

РОБЕРТ СЕДЖВИК Принстонский Университет

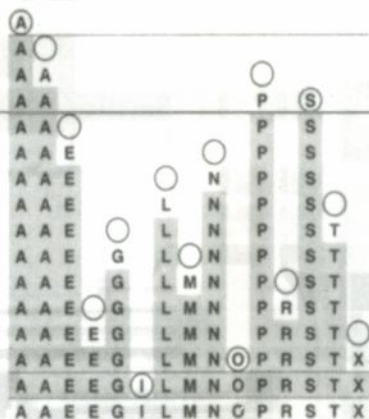
# Фундаментальные алгоритмы

третья редакция

## на **C++**

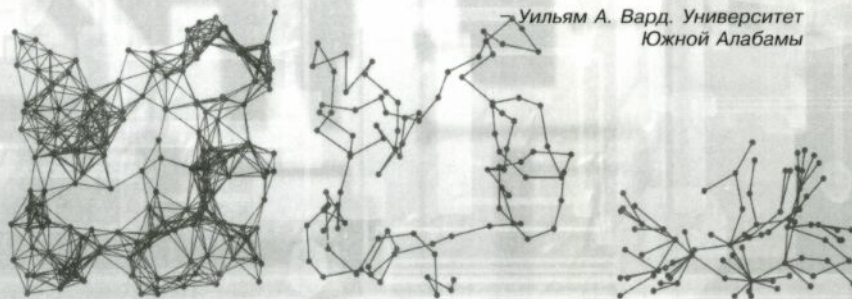
Части 1-4

Анализ  
Структуры данных  
Сортировка  
Поиск



Роберт Седжвик обладает настоящим талантом просто и доступно объяснять сложнейшие вещи. Использование реальных программ, занимающих менее одной страницы, которые к тому же легко понять – вот одно из несомненных достоинств книги. Рисунки, программы и таблицы оказывают существенную помощь в усвоении материала и выгодно отличают эту книгу от множества других.

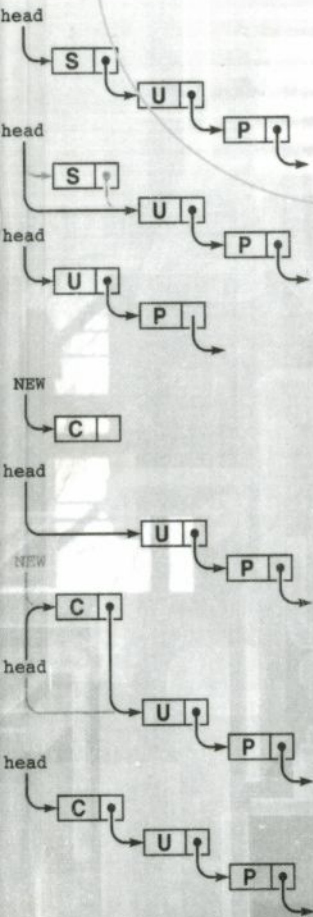
*Уильям А. Вард, Университет Южной Алабамы*



**DS** издательство **DiaSoft**

**ADDISON-WESLEY**  
Pearson Education

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
A SORT IN GEXAMPLE  
0 10 8 14 7 5 13 11 6 2 12 3 1 4 9



# Фундаментальные алгоритмы на C++

ЧАСТИ 1–4

---

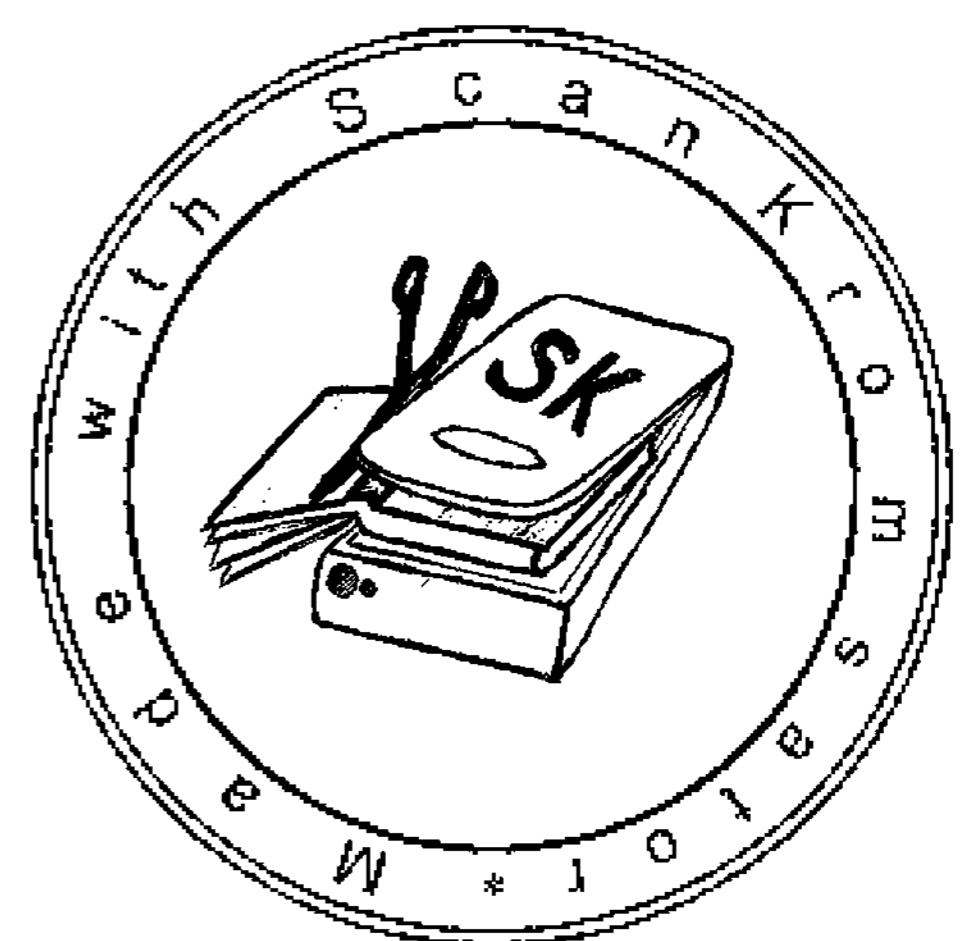
АНАЛИЗ  
СТРУКТУРЫ ДАННЫХ  
СОРТИРОВКА  
ПОИСК

Роберт Седжвик

 торгово-издательский дом  
**DiaSoft**

Москва • Санкт-Петербург • Киев

2001



# Algorithms

THIRD EDITION

# in C++

PARTS 1–4

---

FUNDAMENTALS  
DATA STRUCTURES  
SORTING  
SEARCHING

## Robert Sedgewick

Princeton University

 **ADDISON-WESLEY**

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California

Berkeley, California • Don Mills, Ontario • Sydney • Bonn • Amsterdam

Tokyo • Mexico City



ББК 32.973

С 82

**Седжвик Роберт**

С 82 **Фундаментальные алгоритмы на С++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./Роберт Седжвик. – К.: Издательство «ДиаСофт», 2001.– 688 с.**

ISBN 966-7393-89-5

Эта книга посвящена глубокому исследованию всех основополагающих концепций и алгоритмов, которые можно отнести к разряду “вечных”. Изучив их, вы получите знания, которые *никогда* не устареют и которыми вы будете пользоваться *всегда*.

Краткость, точность, выверенность, актуальность, изобилие примеров и учебных заданий – вот лишь краткий перечень очевидных достоинств книги. Иллюстрация алгоритмов на одном из наиболее эффективных языков С++ лишний раз подчеркивает их популярность. Книгу можно использовать в качестве справочника и даже просто читать как художественную литературу, получая при этом ни с чем не сравнимое удовольствие.

Поскольку книга построена в виде курса лекций, ее можно использовать и в учебном процессе.

Научное издание

**Седжвик Роберт**

ФУНДАМЕНТАЛЬНЫЕ АЛГОРИТМЫ НА С++.

АНАЛИЗ / СТРУКТУРЫ ДАННЫХ / СОРТИРОВКА / ПОИСК

Заведующий редакцией *С.Н.Козлов*

Главный редактор *Ю.Н.Артеменко*

Научный редактор *Ю.Н.Артеменко*

Верстка *Т.Н.Артеменко*

Главный дизайнер *О.А.Шадрин*

НК

Сдано в набор 23.06.2001. Подписано в печать 25.07.2001. Формат 70×100/16.

Бумага офсетная. Гарнитура Таймс. Печать офсетная. Усл.печ.л. 51,60. Усл.кр.отт. 51,60

Тираж 2000 экз. Заказ №

Издательство «ДиаСофт», 04053, Киев-53, а/я 100, тел./факс (044) 212-1254.

e-mail [books@diasoft.kiev.ua](mailto:books@diasoft.kiev.ua), <http://www.diasoft.kiev.ua>

**Отпечатано с готовых диапозитивов**

**в ФГУП ордена Трудового Красного Знамени «Техническая книга»**

**Министерства Российской Федерации по делам печати,**

**телерадиовещания и средств массовых коммуникаций**

**198005, Санкт-Петербург, Измайловский пр., 29.**

Translation copyright © 2001 by DIASOFT LTD.

(Original English language title from Proprietor's edition of the Work)

Original English language title Algorithms in C++ Third Edition, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching, Third Edition by Robert Sedgwick, Copyright © 1999, All rights reserved

Published by arrangement with the original publisher, ADDISON WESLEY LONGMAN, a Pearson Education Company.

Лицензия предоставлена издательством Addison Wesley Longman

Все права зарезервированы, включая право на полное или частичное воспроизведение в любой форме

ISBN 966-7393-89-5 (рус )

ISBN 0-201-35088-2 (англ )

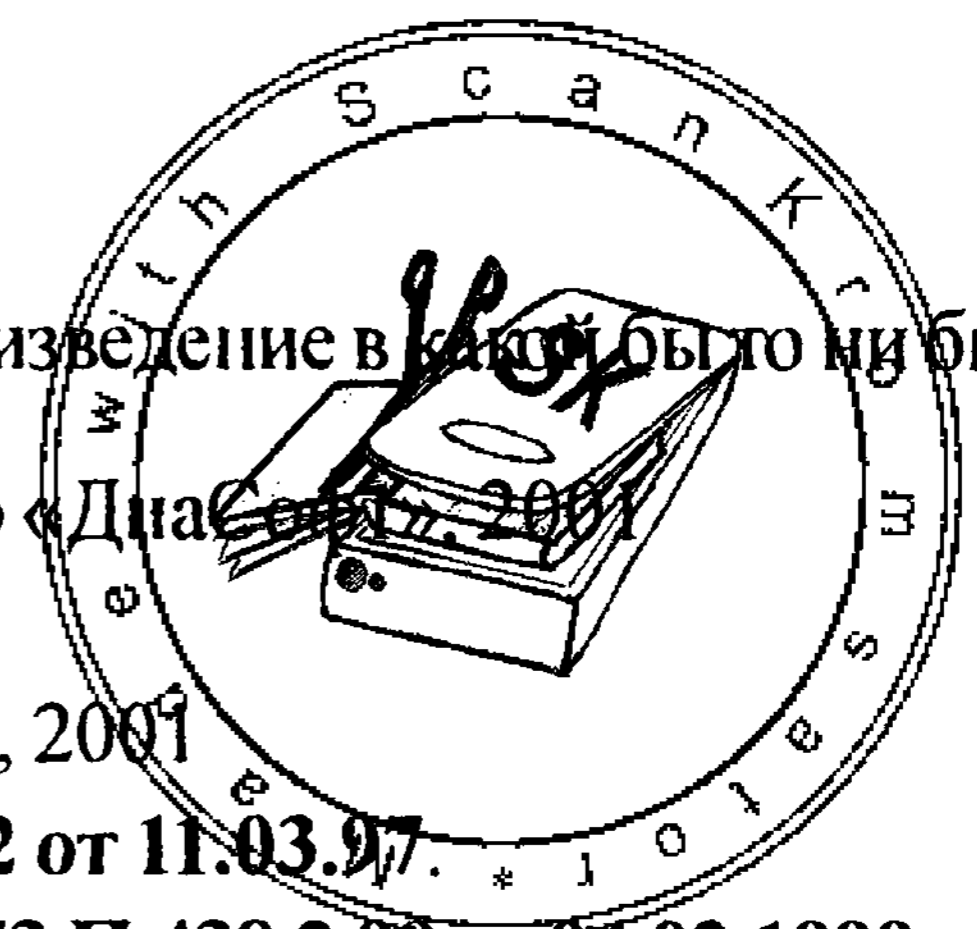
© Перевод на русский язык Издательство «ДиаСофт», 2001

© Addison Wesley Longman, 1999.

© Оформление Издательство «ДиаСофт», 2001

Свидетельство о регистрации 24729912 от 11.03.97. \* 1 0 1 0 1 0

Гигиеническое заключение № 77.99.6.953.П.438.2.99 от 04.02.1999



# Оглавление

<b>Часть 1. Анализ .....</b>	<b>19</b>
<b>Глава 1. Введение .....</b>	<b>20</b>
1.1 Алгоритмы .....	21
1.2 Пример задачи: связность .....	23
1.3 Алгоритмы объединение-поиск .....	27
1.4 Перспектива .....	38
1.5 Обзор тем .....	40
<b>Глава 2. Принципы анализа алгоритмов .....</b>	<b>42</b>
2.1. Разработка и эмпирический анализ .....	43
2.2 Анализ алгоритмов .....	47
2.3 Рост функций .....	49
2.4 O-нотация .....	56
2.5 Простейшие рекурсии .....	60
2.6 Примеры алгоритмического анализа .....	64
2.7 Гарантии, предсказания и ограничения .....	69
<b>Часть 2. Структуры данных .....</b>	<b>75</b>
<b>Глава 3. Элементарные структуры данных .....</b>	<b>76</b>
3.1 Строительные блоки .....	77
3.2 Массивы .....	87
3.3 Связные списки .....	94
3.4 Обработка простых списков .....	100
3.5 Распределение памяти под списки .....	108
3.6 Строки .....	111
3.7 Составные структуры данных .....	116
<b>Глава 4. Абстрактные типы данных .....</b>	<b>126</b>
4.1 Абстрактные объекты и коллекции объектов .....	136
4.2 АД для стека магазинного типа .....	139
4.3 Примеры программ-клиентов, использующих АД стека .....	142
4.4 Реализации АД стека .....	148
4.5 Создание нового АД .....	152
4.6 Очереди FIFO и обобщенные очереди .....	159

4.7 Повторяющиеся и индексные элементы .....	166
4.8 АТД первого класса .....	171
4.9 Пример использования АТД в приложении .....	182
4.10 Перспективы .....	187
<b>Глава 5. Рекурсия и деревья .....</b>	<b>189</b>
5.1 Рекурсивные алгоритмы .....	190
5.2 Разделяй и властвуй .....	197
5.3 Динамическое программирование .....	210
5.4 Деревья .....	218
5.5 Математические свойства бинарных деревьев .....	226
5.6 Обход дерева .....	230
5.7 Рекурсивные алгоритмы бинарных деревьев .....	235
5.8 Обход графа .....	240
5.9 Перспективы .....	245
<b>Часть 3. Сортировка .....</b>	<b>248</b>
<b>Глава 6. Элементарные методы сортировки .....</b>	<b>249</b>
6.1 Правила игры .....	251
6.2 Сортировка выбором .....	257
6.3 Сортировка вставками .....	258
6.4 Пузырьковая сортировка .....	261
6.5 Характеристики производительности элементарных методов сортировки .....	263
6.6 Сортировка методом Шелла .....	269
6.7 Сортировка других типов данных .....	279
6.8 Сортировка по индексам и указателям .....	283
6.9 Сортировка связанных списков .....	291
6.10 Метод распределяющего подсчета .....	295
<b>Глава 7. Быстрая сортировка .....</b>	<b>299</b>
7.1 Базовый алгоритм .....	300
7.2 Характеристики производительности быстрой сортировки .....	305
7.3 Размер стека .....	309
7.4 Подфайлы небольших размеров .....	313
7.5 Метод разделения с вычислением медианы из трех элементов .....	316
7.6 Дублированные ключи .....	321
7.7 Строки и векторы .....	324
7.8 Выборка .....	326

---

<b>Глава 8. Слияние и сортировка слиянием .....</b>	<b>330</b>
8.1 Двухпутевое слияние .....	332
8.2 Абстрактное обменное слияние .....	334
8.3 Нисходящая сортировка слиянием .....	336
8.4 Усовершенствования базового алгоритма .....	340
8.5 Восходящая сортировка слиянием .....	342
8.6 Производительность сортировки слиянием .....	346
8.7 Реализация сортировки слиянием, ориентированной на связные списки .....	349
8.8 Возврат к рекурсии .....	353
<b>Глава 9. Очереди по приоритетам и пирамидальная сортировка .....</b>	<b>355</b>
9.1 Элементарные реализации .....	359
9.2 Пирамидальная структура данных .....	363
9.3 Алгоритмы для сортирующих деревьев .....	365
9.4 Пирамидальная сортировка .....	373
9.5 Абстрактный тип данных очереди по приоритетам .....	380
9.6 Очередь по приоритетам для индексных элементов .....	385
9.7 Биномиальные очереди .....	389
<b>Глава 10. Поразрядная сортировка .....</b>	<b>401</b>
10.1 Биты, байты и слова .....	404
10.2 Двоичная быстрая сортировка .....	407
10.3 Поразрядная сортировка MSD .....	412
10.4 Трехпутевая поразрядная быстрая сортировка .....	420
10.5 Поразрядная сортировка LSD .....	425
10.6 Рабочие характеристики поразрядных сортировок .....	429
10.7 Сортировки с сублинейным временем выполнения .....	433
<b>Глава 11. Методы сортировки специального назначения .....</b>	<b>438</b>
11.1 Четно-нечетная сортировка слиянием Бэтчера .....	440
11.2 Сети сортировки .....	445
11.3 Внешняя сортировка .....	454
11.4 Различные реализации сортировки-слияния .....	461
11.5 Параллельная процедура сортировки-слияния .....	468

---

<b>Часть 4. Поиск</b> .....	<b>474</b>
<b>Глава 12. Таблицы символов и деревья бинарного поиска</b> .....	<b>475</b>
12.1 Абстрактный тип данных таблицы символов .....	477
12.2 Поиск с использованием индексации по ключам .....	483
12.3 Последовательный поиск .....	486
12.4 Бинарный поиск .....	493
12.5 Деревья бинарного поиска .....	498
12.6 Характеристики производительности деревьев бинарного поиска .....	504
12.7 Реализация индексов при использовании таблиц символов .....	508
12.8 Вставка в корень в деревьях бинарного поиска .....	511
12.9 Реализации других функций АД с помощью BST-дерева .....	515
<b>Глава 13. Сбалансированные деревья</b> .....	<b>523</b>
13.1 Рандомизованные BST-деревья .....	527
13.2 Расширенные деревья бинарного поиска .....	533
13.3 Нисходящие 2-3-4-деревья .....	540
13.4 Красно-черные деревья, или RB-деревья .....	545
13.5 Списки пропусков .....	555
13.6 Характеристики производительности .....	562
<b>Глава 14. Хеширование</b> .....	<b>567</b>
14.1 Хеш-функции .....	568
14.2 Раздельное связывание .....	578
14.3 Линейное зондирование .....	583
14.4 Двойное хеширование .....	588
14.5 Динамические хеш-таблицы .....	594
14.6 Перспективы .....	597
<b>Глава 15. Поразрядный поиск</b> .....	<b>602</b>
15.1 Деревья цифрового поиска .....	603
15.2 Trie-деревья .....	608
15.3 patricia-деревья .....	617
15.4 Многопутевые trie-деревья и TST-деревья .....	625
15.5 Алгоритмы индексирования текстовых строк .....	640
<b>Глава 16. Внешний поиск</b> .....	<b>645</b>
16.1 Правила игры .....	647
16.2 Индексированный последовательный доступ .....	649
16.3 B-деревья .....	652
16.4 Расширяемое хеширование .....	665
16.5 Перспективы .....	676
<b>Предметный указатель</b> .....	<b>680</b>



# Предисловие

Цель этой книги заключается в исследовании наиболее важных компьютерных алгоритмов, которые применяются в настоящий момент, а также обучение фундаментальным технологиям постоянно возрастающего количества разработчиков, испытывающих потребность в подобного рода информации. Книга может использоваться в качестве учебника для студентов второго, третьего и четвертого курсов факультетов, на которых изучается компьютерная инженерия, после того как студенты овладели основными навыками программирования, но перед прослушиванием дополнительных тем компьютерной инженерии и компьютерных приложений. Поскольку книга содержит реализации полезных алгоритмов и подробную информацию по характеристикам производительности этих алгоритмов, она может пригодиться и тем, кто занимается самообразованием, или же может послужить справочником для тех, кого интересует разработка компьютерных систем или приложений. Широкий круг рассматриваемых вопросов делает ее исключительным учебником в данной предметной области.

В новом издании текст был полностью переработан, и в него было включено более тысячи новых упражнений, более сотни новых рисунков и десятки новых программ. Кроме того, ко всем рисункам и программам были добавлены подробные комментарии. Этот новый материал охватывает как новые темы, так и более полно поясняет многие классические алгоритмы. Больше внимание, уделенное в книге абстрактным типам данных, расширяет сферу применения приведенных программ и делает их более пригодными для современных сред программирования. Читатели, знакомые с предыдущими изданиями кни-

*Адамс,  
Эндрю,  
Брейтман,  
Робби и, в  
особенности,  
Линде*

ги, найдут в ней множество новой информации; абсолютно все читатели найдут в книге большой объем учебного материала, который позволяет успешно изучить важные концепции.

В связи с большим объемом нового материала новое издание разбито на два тома (каждый примерно равен по объему предыдущему изданию), первый из которых — перед вами. В этом томе освещены фундаментальные концепции, структуры данных, алгоритмы сортировки и поиска; второй том посвящен более сложным алгоритмам и приложениям, построенным на базе абстракций и методов, разработанных в первом томе. Почти весь включенный в это издание материал по основным принципам и структурам данных является новым.

Книга адресована не только программистам и студентам и аспирантам, изучающим компьютерные науки. Подавляющее большинство пользователей компьютеров желают работать быстрее либо решать более сложные задачи. Приведенные в книге алгоритмы представляют собой квинтэссенцию знаний, накопленных за последние более чем 50 лет, которые стали совершенно необходимыми для эффективного использования компьютера для широчайшего множества приложений. Начиная с задач моделирования систем из  $N$  тел в физике и завершая задачами анализа генетического кода в молекулярной биологии, описанные здесь базовые методы стали важной составной частью научных исследований. Они являются также важными составными частями современных программных систем, в числе которых как системы управления базами данных, так и механизмы поиска в Internet. По мере расширения сферы применения компьютерных приложений возрастает и значение освещенных здесь базовых методов. Эта книга предназначена быть источником информации для студентов и профессионалов, заинтересованных в понимании и эффективном использовании описанных фундаментальных алгоритмов как основных инструментальных средств для любого компьютерного приложения, для которого они подходят.

## Круг рассматриваемых вопросов

Книга содержит 16 глав, сгруппированных в виде четырех основных частей: анализ, структуры данных, сортировка и поиск. Приведенные в ней описания призваны познакомить читателей с основными свойствами максимально широкого круга основных алгоритмов. Описанные здесь алгоритмы находят широкое применение в течение долгих лет и являются существенно важными как для профессиональных программистов, так и для изучающих компьютерные науки. Все описанные в книге остроумные методы, от биномиальных очередей до Patricia-деревьев, относятся к базовым концепциям, лежащим в основе компьютерных наук. Второй том состоит из четырех дополнительных частей, в которых освещены строки, геометрия, графы и другие темы. Основной целью при написании этих книг было собрать воедино фундаментальные методы из этих различных областей для ознакомления с лучшими методами решения задач с помощью компьютера.

Собранный в этой книге материал должен произвести наибольшее впечатление на тех читателей, которые знакомы с одним-двумя курсами по компьютерам или обладают аналогичным опытом в программировании: знакомых с программированием на языках высокого уровня, таких как C++, Java или C, и, возможно, тех, кто прошел

основные концепции систем программирования. Таким образом, в первую очередь, книга предназначена тем, кто знаком с современным языком программирования и с основными свойствами современных компьютерных систем. В тексте присутствуют ссылки, которые могут поспособствовать восполнению пробелов в знаниях.

Большинство математических выкладок, сопровождающих аналитические результаты, являются самодостаточными (либо же помечены как выходящие за рамки этой книги), поэтому для понимания изложенного в книге материала требуется лишь минимальная математическая подготовка, хотя хорошая подготовка определенно будет способствовать лучшему усвоению.

## Использование в рамках учебных курсов

Изучение изложенного в книге материала может существенно варьироваться в зависимости от предпочтений преподавателя и подготовки студентов. Книга содержит достаточный объем фундаментальных материалов, чтобы ее можно было использовать для изучения структур данных начинающими, и в ней приведено достаточно дополнительных материалов, чтобы она могла использоваться в качестве учебника по разработке и анализу алгоритмов для более подготовленных студентов. Одни преподаватели могут уделять больше внимания реализациям и практическим вопросам, а другие — анализу и теоретическим концепциям.

Автор разрабатывает ряд учебных материалов, предназначенных для использования в сочетании с этой книгой, в том числе и слайдовые демонстрации для использования на лекциях, задания по программированию, домашние задания и примеры экзаменационных билетов, а также интерактивные упражнения для студентов. Эти материалы будут доступны на Web-странице, посвященной книге, по адресу <http://www.awl.com/cseng/titles/0-201-35088-2>.

В элементарном курсе по структурам данных и алгоритмам основное внимание может уделяться основным структурам данных, описанным в части 2, и их использованию в реализациях, приведенных частях 3 и 4. В курсе по разработке и анализу алгоритмов основное внимание может уделяться материалу, изложенному в части 1 и главе 5, после чего можно приступить к изучению способов достижения хорошей асимптотической производительности алгоритмов, описанных в частях 3 и 4. В курсе по разработке программного обеспечения математические выкладки и дополнительные материалы по алгоритмам можно опустить, а основное внимание акцентировать на интеграции приведенных здесь реализаций в большие программы или системы. В курсе по алгоритмам может использоваться обзорный подход с рассмотрением концепций, относящихся ко всем названным темам.

В течение нескольких последних лет предыдущие издания книги использовались во множестве колледжей и университетов по всему миру как учебник для студентов второго и третьего курсов компьютерных специальностей, а также в качестве дополнительного учебного пособия для студентов других специализаций. Исходя из опыта Принстонского университета, можно утверждать, что приведенные в этой книге материалы обеспечивают основную массу студентов вводными сведениями по компьютерным наукам, которые затем могут быть расширены в ходе последующего изучения курсов по анализу алгоритмов, программированию систем и теории компьютеров,

в то же время снабжая все возрастающую группу студентов других специальностей большим набором технологий, которые они смогут немедленно и успешно использовать.

Упражнения, большинство из которых впервые включены в это издание, разделяются на несколько типов. Ряд из них предназначен для проверки усвоения изложенного в книге материала, и в них читателям просто предлагается проанализировать пример или применить описанные в тексте концепции. Другие же требуют реализации и сборки алгоритмов воедино или экспериментального исследования с целью сравнения вариантов алгоритмов и изучения их свойств. Третьи содержат важную информацию, уровень детализации которой выходит за рамки основного материала книги. Ознакомление и выполнение упражнений будет полезно абсолютно всем читателям.

## Практическое применение алгоритмов

Каждый желающий как можно более эффективно использовать компьютер может воспользоваться этой книгой как справочным пособием. Читатели, обладающие определенным опытом в программировании, смогут найти в книге информацию по конкретным темам. В большинстве случаев отдельные главы книги не зависят одна от другой, хотя иногда в алгоритмах используются методы, описанные в предыдущих главах.

Книга ориентирована на изучение алгоритмов, которые, скорее всего, найдут практическое применение. В ней приводится информация об инструментальных средствах, чтобы читатели могли осознанно реализовывать, отлаживать и настраивать алгоритмы для решения конкретных задач или для обеспечения заданных функциональных возможностей приложения. В книгу включены полные реализации рассмотренных методов, а также описания действия этих программ применительно к последовательному набору примеров.

Поскольку работа выполняется с реальным, а не псевдокодом, программы можно быстро задействовать в практических целях. Их листинги доступны на начальной Web-странице книги. При изучении алгоритмов эти работающие программы можно использовать различными способами. Их можно прочесть для проверки своего понимания особенностей алгоритма или с целью ознакомления с одним из возможных способов обработки инициализации, граничных условий и других сложных ситуаций, которые зачастую вызывают затруднения при программировании. Читатели могут запустить их, чтобы увидеть алгоритмы в действии, для эмпирического исследования производительности и для сравнения полученных результатов с данными, приведенными в таблицах книги, либо же для тестирования внесенных изменений.

Когда это уместно, для обоснования предпочтительности определенных алгоритмов приводятся как экспериментальные, так и аналитические результаты. В представляющих интерес случаях описывается взаимосвязь между рассматриваемыми практическими алгоритмами и чисто теоретическими результатами. Хотя это и не подчеркивается, в тексте книги устанавливается взаимосвязь между анализом алгоритмов и теорией компьютерных наук. В книге собрана и проанализирована специфическая информация по характеристикам производительности алгоритмов и реализаций.

# Язык программирования

Для всех реализаций использовался язык программирования C++. В программах применяется широкое множество стандартных идиом C++, а в текст включены краткие описания каждой из конструкций.

Автор совместно с Крисом Ван Виком разработали основанный на классах, шаблонах и перегруженных операциях стиль программирования на C++, который по нашему мнению позволяет эффективно представлять алгоритмы и структуры данных в виде реальных программ. Мы стремились к изящным, компактным, эффективным и переносимым реализациям. Везде, где это возможно, мы стремились сохранить единство стиля, дабы сходные по действию программы выглядели похожими.

Для множества приведенных в этой книге алгоритмов схожесть сохраняется независимо от языка: быстрая сортировка остается быстрой сортировкой (это лишь один яркий пример) независимо от того, выражена ли она на языке Ada, Algol-60, Basic, C, C++, Fortran, Java, Mesa, Modula-3, Pascal, PostScript, Smalltalk или на одном из других бесчисленных языков программирования или в другой среде, где она зарекомендовала себя как эффективный метод сортировки. С одной стороны, представленный нами код продиктован опытом реализации алгоритмов на этих и множестве других языков (C-версия этой книги также доступна, а Java-версия будет вскоре издана). С другой стороны, отдельные особенности некоторых из этих языков продиктованы опытом их применения разработчиками по отношению к некоторым алгоритмам и структурам данных, которые рассматриваются в книге.

Глава 1 является характерным примером подобного подхода к разработке эффективных реализаций рассматриваемых алгоритмов на C++, а в главе 2 описывается используемый нами подход к их анализу. Главы 3 и 4 посвящены описанию и обоснованию основных механизмов, используемых для реализаций типов данных и абстрактных типов данных. Эти четыре главы закладывают фундамент для понимания остальной части книги.

## Благодарности

Многие читатели прислали мне исключительно полезные отзывы о предыдущих изданиях этой книги. В частности, в течение ряда лет предварительные наброски книги апробировались на сотнях студентов в Принстоне и Брауне. Особую благодарность хотелось бы выразить Трине Авери (Trina Avery) и Тому Фримену (Tom Freeman) за оказанную помощь в выпуске первого издания; Джанет Инсерпи (Janet Incerpi) за проявленные ею творческий подход и изобретательность, чтобы заставить аппаратное и программное обеспечение нашей первой примитивной компьютеризованной издательской системы создать первое издание; Марку Брауну (Marc Brown) за его участие в исследованиях по визуализации алгоритмов, которые явились источником множества рисунков, приведенных в книге, а также Дэйву Хэнсону (Dave Hanson) и Эндрю Эппелю (Andrew Appel) за их готовность ответить на все мои вопросы, связанные с языками программирования. Я хотел бы также поблагодарить очень многих читателей, приславших отзывы о многих изданиях, в том числе Гаю Олмсу (Guy Almes), Джону Бентли (Jon Bentley), Марку Брауну (Marc Brown), Джею Гишеру (Jay Gischer), Аллану Хейдону (Allan Heydon), Кеннеди Лемке (Kennedy Lemke), Юди Манбер (Udi Manber), Дане Ричардс (Dana Richards), Джону Рейфу (John Reif), М. Розенфельду (M. Rosenfeld), Стивену Сейдману (Stephen Seidman), Майку Квину (Michael Quinn) и Вильяму Варду (William Ward).

При подготовке нового издания я имел удовольствие работать с Питером Гордоном (Peter Gordon), Дебби Лафферти (Debbie Lafferty) и Хелен Гольдштейн (Helen Goldstein) из издательства Addison-Wesley, которые терпеливо опекали этот проект с момента его зарождения. Большое удовольствие доставила мне совместная работа с другими штатными сотрудниками этого издательства. Характер проекта сделал подготовку издания данной книги несколько непривычной задачей для многих из них, и я высоко ценю проявленную ими снисходительность.

В процессе написания этой книги я приобрел трех новых наставников и хочу особо выразить им свою признательность. Во-первых, Стиву Саммиту (Steve Summit), который внимательно проверил на техническом уровне первые варианты рукописи и предоставил буквально тысячи подробных комментариев, особенно в отношении программ. Стив хорошо понял мое стремление представить изящные и эффективные реализации, и его комментарии помогли мне не только обеспечить определенное единообразие реализаций, но и существенно улучшить многие из них. Во-вторых, хочу поблагодарить Лин Дюпре (Lyn Dure) за тысячи подробных комментариев в отношении рукописи, которые помогли не только избежать и исправить грамматические ошибки, но и (что значительно важнее) выработать последовательный и связный стиль написания, что позволило собрать воедино устрашающую массу технического материала. В-третьих, я признателен Крису Ван Вику (Chris Van Wyk), который реализовал и отладил все мои алгоритмы на C++, помог выработать подходящий стиль программирования на C++ и дважды внимательно прочел рукопись. Кроме того, Крис сохранял терпение, когда я препарировал многие из его программ, а затем, по мере того как все больше узнавал от него о языке C++, вынужден был снова собирать их воедино, причем в большинстве случаев так, как первоначально на-

писал их он. Я исключительно благодарен полученной возможности поучиться у Стива, Лин и Криса — они оказали решающее влияние на разработку этой книги.

Многое из написанного здесь я узнал из лекций и трудов Дона Кнута (Don Knuth) — моего наставника в Стэнфорде. Хотя непосредственно Дон и не участвовал в создании этой работы, его влияние должно ощущаться по всей книге, поскольку именно он поставил изучение алгоритмов на научную основу, что делает возможным вообще появление подобного рода книг. Мой друг и коллега Филипп Флажолле (Philippe Flajolet), благодаря которому анализ алгоритмов стал вполне сформировавшейся научной областью, оказал такое же влияние.

Я глубоко признателен за оказанную мне поддержку Принстонскому университету, Брауновскому университету и Национальному институту исследований в области информатики и автоматки (INRIA), где я проделал большую часть работы над книгой, а также Институту исследований защиты и Исследовательскому центру компании Хегох в Пало-Альто, где была проделана часть работы во время моих визитов. Многие главы книги основываются на исследованиях, которые щедро финансировались Национальным научным фондом и Отделом военно-морских исследований. И в заключение, я благодарю Билла Боуэна (Bill Bowen), Аарона Лемоника (Aaron Lemonick) и Нейла Руденштайна (Neil Rudenstine) за то, что они способствовали созданию в Принстоне академической обстановки, в которой я получил возможность подготовить эту книгу, несмотря на множество других возложенных на меня обязанностей.

*Роберт Седжвик*

*Марли-ле-Руа, Франция, 1983 г.*

*Принстон, Нью-Джерси, 1990, 1992 гг.*

*Джеймстаун, Род-Айленд, 1997 г.*

*Принстон, Нью-Джерси, 1998 г.*

## Предисловие консультанта по С++

Именно алгоритмы вовлекли меня в компьютерные науки. Изучение алгоритмов требует сочетания нескольких подходов: творческого — для выработки идеи решения задачи; логического — для анализа правильности решения; математического — для анализа производительности; и скрупулезного — для выражения идеи в виде подробной последовательности шагов, чтобы она могла превратиться в программу. Поскольку наибольшее удовлетворение при изучении алгоритмов мне доставляет их реализация в виде работающих компьютерных программ, я с радостью ухватился за возможность поработать с Бобом Седжвиком над книгой по алгоритмам, основанной на программах С++.

Мы с Бобом написали примеры программ, непосредственно используя соответствующие свойства языка С++. Мы применяли классы для разделения определения абстрактного типа данных от нюансов реализации. Мы использовали шаблоны и перегруженные операторы, чтобы программы можно было использовать без изменений со многими различными типами данных.

Однако, мы не воспользовались возможностью продемонстрировать множество других технологий С++. Например, обычно мы опускаем, по крайней мере, некоторые конструкторы, необходимые, чтобы любой класс был "первым классом"; в большинстве конструкторов для хранения значений в членах данных используется присваивание, а не инициализация; вместо строк из библиотеки С++ мы использовали символьные строки в стиле С; мы не использовали большинство возможных "упрощенных" классов контейнеров; и, наконец, мы использовали простые, а не "интеллектуальные" указатели.

В большинстве случаев подобный выбор был продиктован желанием сосредоточиться на алгоритмах, а не на нюансах технологии, характерных для С++. Как только читатели поймут, как работают программы в этой книге, они окажутся вполне подготовленными для изучения любых или же всех этих технологий, чтобы оценить связанные с ними ограничения производительности и предусмотрительность разработчиков библиотеки стандартных шаблонов.

Независимо от того, впервые ли читатели сталкиваются с изучением алгоритмов, или же просто освежают ранее полученные знания, они смогут получить, по крайней мере, такое же удовольствие, какое получил я, работая над программами с Бобом Седжвиком.

Я благодарю Джона Бентли (Jon Bentley), Брайана Кернигана (Brian Kernighan) и Тома Шимански (Tom Szymanski), от которых узнал многое о программировании; Дебби Лафферти (Debbie Lafferty), которая предложила мне принять участие в этом проекте; а также фирму Bell Labs, университету Дрю и Принстонскому университету за оказанную ими безвозмездную поддержку.

*Кристофер Ван Вик  
Чатем, Нью-Джерси, 1998 г.*



## Примечания к упражнениям

Классификация упражнений – это занятие, сопряженное с рядом трудностей, поскольку читатели такой книги, как эта, обладают различным уровнем знаний и опыта. Тем не менее, определенное указание не мешает, поэтому многие упражнения помечены одним из четырех маркеров, дабы проще было выбрать соответствующий подход.

Упражнения, которые проверяют *понимание* материала, помечены незаполненным треугольником:

- ▷ **9.54.** Вычертить биномиальную очередь размера 29, воспользовавшись представлением в виде биномиального дерева.

Чаще всего такие упражнения непосредственно связаны с примерами в тексте. Они не должны вызывать особых трудностей, но их выполнение может прояснить факт или концепцию, которые возможно были упущены при прочтении текста.

Упражнения, которые *дополняют текст новой и требующей размышлений* информацией, помечены незаполненной окружностью:

- **9.62.** Построить такую программную реализацию биномиальной очереди, чтобы выполнялась лемма 9.7. Использовать для этой цели сигнальный указатель, отмечающий точку, в которой циклы должны завершаться.

Такие упражнения заставляют подумать о важных концепциях, связанных с материалом, изложенным в тексте, или ответить на вопрос, который может возникнуть во время прочтения. Возможно, читатели сочтут полезным прочесть эти упражнения даже при отсутствии времени для их выполнения.

Упражнения, которые имеют целью поставить перед читателями *задачу*, помечены черной точкой:

- **9.63.** Реализовать операцию *вставить* (*insert*) для биномиальных очередей путем явного использованием одной лишь операции *объединить* (*join*).

Для выполнения таких упражнений потребуются потратить значительное время, в зависимости от опыта читателя. В общем случае, лучше всего выполнять их за несколько подходов.

Несколько упражнений, которые *особенно трудны* (по сравнению с большинством других) помечены двумя черными точками:

- **9.64.** Реализовать операцию *изменить приоритет* (*change priority*) и *удалить* (*remove*) для биномиальных очередей. *Совет:* Потребуется добавить третью связь, которая указывает на узлы вверх по дереву.

Эти упражнения аналогичны вопросам, которые могут ставиться в научной литературе, однако материал книги может так подготовить читателей, что им доставит удовольствие попытаться ответить на них (возможно, и преуспеть в этом).

Мы старались, чтобы пометки были безотносительными к программной и математической подготовке читателей. Те упражнения, которые требуют наличия опыта по программированию или математическому анализу, очевидны. Мы призываем всех читателей проверить свое понимание алгоритмов, реализовав их. Тем не менее, уп-

ражнения подобные приведенному ниже, просты для профессиональных программистов или студентов, изучающих программирование, но могут потребовать значительных усилий от тех, кто в последнее время по ряду причин программированием не занимался:

- **1.22** Измените программу 1.4, чтобы она генерировала случайные пары целых чисел в диапазоне от 0 до  $N - 1$  вместо того, чтобы считывать их из стандартного ввода, и выполняла цикл до тех пор, пока не будет выполнено  $N - 1$  операций *union*. Выполните программу для значений  $N = 10^3, 10^4, 10^5$  и  $10^6$  и выведите общее количество ребер, генерируемых для каждого значения  $N$ .

Мы призываем всех читателей стремиться учитывать приводимые нами аналитические обоснования свойств всех алгоритмов. С другой стороны, упражнения, подобные нижеследующему, не составят сложности для ученого или изучающего дискретную математику, однако наверняка потребуют значительных усилий от тех, кто давно не занимался математическим анализом:

- 1.12** Вычислите *среднее* расстояние от узла до корня в худшем случае в дереве, построенном алгоритмом взвешенного быстрого объединения из  $2^n$  узлов.

Существует слишком много упражнений, чтобы их все можно было прочесть и усвоить; тем не менее, я надеюсь, что в книге приводится достаточное количество упражнений, дабы сподвигнуть читателей стремиться к более глубокому пониманию интересующих их тем, чем можно добиться в результате простого чтения текста.

# *Часть 1*

# **Анализ**

*В этой части:*

- 1 Введение**
- 2 Принципы анализа алгоритмов**

## Введение

**Ц**ель этой книги — изучение широкого множества важных и полезных *алгоритмов*: методов решения задач, которые подходят для компьютерной реализации. Мы будем иметь дело с различными областями применения, всегда уделяя основное внимание фундаментальным алгоритмам, которые важно знать и интересно изучать. Мы затратим достаточное время на изучение каждого алгоритма, чтобы понять его основные характеристики и разобраться в его тонкостях. Наша цель — достаточно подробно изучить большое количество наиболее важных алгоритмов, используемых в компьютерах в настоящее время, чтобы иметь возможность их применять и учитывать при решении задач.

Стратегия, используемая для изучения представленных в этой книге программ, заключается в их реализации и тестировании, экспериментировании с их разновидностями, рассмотрении их действия применительно к небольшим примерам и попытке их применения к более сложным практическим задачам. Для описания алгоритмов будет использоваться язык программирования C++, тем самым обеспечивая и полезные реализации. Программы написаны в единообразном стиле, который допускает также применение и других современных языков программирования.

Значительное внимание также уделяется характеристикам производительности алгоритмов, чтобы легче было разрабатывать усовершенствованные версии, сравнивать различные алгоритмы выполнения той же самой задачи и предсказывать или гарантировать производительность при решении сложных задач. Для понимания действия алгоритмов может потребоваться выполнение экспериментов

или математического анализа, либо и того и другого. В книге подробно рассматриваются многие наиболее важные алгоритмы; причем, когда это приемлемо, аналитические выкладки приводятся непосредственно в тексте; в других случаях, при необходимости, используются результаты, приводимые в научной литературе.

С целью иллюстрации общего подхода к разработке алгоритмических решений, в этой главе рассматривается подробный пример, включающий в себя ряд алгоритмов решения конкретной задачи. Рассматриваемая задача — не просто модельная задача; она является фундаментальной вычислительной задачей, и полученное решение используется в различных приложениях. Мы начнем с простого решения, затем постараемся выяснить его характеристики производительности, что поможет усовершенствовать алгоритм. После нескольких итераций этого процесса мы получим эффективный и полезный алгоритм решения задачи. Этот пример — своего рода прототип использования этой же общей методологии во всей книге.

В конце главы приводится краткое содержание книги с описанием основных частей и их взаимосвязи между собой.

## 1.1 Алгоритмы

В общем случае при создании компьютерной программы мы реализуем метод, который ранее был разработан для решения какой-либо задачи. Часто этот метод не зависит от конкретного используемого компьютера — весьма вероятно, что он будет равно пригодным для многих компьютеров и многих компьютерных языков. Именно метод, а не саму программу нужно исследовать для выяснения способа решения задачи. Термин *алгоритм* используется в компьютерных науках для описания метода решения задачи, пригодного для реализации в виде компьютерной программы. Алгоритмы составляют основу компьютерных наук: они являются основными объектами изучения во многих, если не в большинстве ее областей.

Большинство представляющих интерес алгоритмов касаются методов организации данных, участвующих в вычислениях. Созданные таким образом объекты называются *структурами данных*, и они также являются центральными объектами изучения в компьютерных науках. Следовательно, алгоритмы и структуры данных идут рука об руку. В этой книге мы покажем, что структуры данных существуют в качестве субпродуктов или конечных продуктов алгоритмов и, следовательно, их нужно изучить, чтобы понять алгоритмы. Простые алгоритмы могут порождать сложные структуры данных и наоборот, сложные алгоритмы могут использовать простые структуры данных. В этой книге будут изучены свойства многих структур данных; фактически книга вполне могла бы называться *"Алгоритмы и структуры данных в C++"*.

Когда компьютер используется для решения той или иной задачи, как правило, мы сталкиваемся с рядом возможных различных подходов. При решении простых задач выбор того или иного подхода вряд ли имеет особое значение, если только выбранный подход приводит к правильному решению. Однако, при решении сложных задач (или в приложениях, в которых приходится решать очень большое количество простых задач) мы немедленно сталкиваемся с необходимостью разработки методов, при которых время или память используются с максимальной эффективностью.

Основная побудительная причина изучения алгоритмов состоит в том, что это позволяет обеспечить огромную экономию ресурсов, вплоть до получения решений за-

дач, которые в противном случае были бы невозможны. В приложениях, в которых обрабатываются миллионы объектов, часто оказывается возможным ускорить работу программы в миллионы раз, используя хорошо разработанный алгоритм. Подобный пример приводится в разделе 1.2 и множестве других разделов книги. Для сравнения, вложение дополнительных денег или времени для приобретения и установки нового компьютера потенциально позволяет ускорить работу программы всего в 10–100 раз. Тщательная разработка алгоритма — исключительно эффективная часть процесса решения сложной задачи в любой области применения.

При разработке очень большой или сложной компьютерной программы значительные усилия должны затрачиваться на выяснение и определение задачи, которая должна быть решена, осознание ее сложности и разбиение ее на менее сложные подзадачи, решения которых можно легко реализовать. Часто реализация многих из алгоритмов, требующихся после разбиения, тривиальна. Однако, в большинстве случаев существует несколько алгоритмов, выбор которых критичен, поскольку для их выполнения требуется большая часть системных ресурсов. Именно этим типам алгоритмов уделяется основное внимание в данной книге. Мы изучим ряд основополагающих алгоритмов, которые полезны при решении сложных задач во многих областях применения.

Совместное использование программ в компьютерных системах становится все более распространенным, поэтому, хотя можно ожидать, что *использовать* придется многие из рассмотренных в книге алгоритмов, одновременно можно надеяться, что *реализовывать* придется лишь немногие из них. Например, библиотека стандартных шаблонов (Standard Template Library) C++ содержит реализации множества базовых алгоритмов. Однако реализация простых версий основных алгоритмов позволяет лучше их понять и, следовательно, эффективнее использовать и настраивать более совершенные библиотечные версии. И что еще важнее, повод повторной реализации основных алгоритмов возникает очень часто. Основная причина состоит в том, что мы сталкиваемся, и очень часто, с совершенно новыми вычислительными средами (аппаратными и программными) с новыми свойствами, которые не могут наилучшим образом использоваться старыми реализациями. Другими словами, чтобы наши решения были более переносимыми и дольше сохраняющими актуальность, часто приходится реализовывать базовые алгоритмы, приспособленные к конкретной задаче, а не основывающиеся на системных подпрограммах. Другая часто возникающая причина повторной реализации базовых алгоритмов заключается в том, что несмотря на усовершенствования встроенные в C++, механизмы, используемые для совместного использования программ, не всегда достаточно мощны, чтобы библиотечные программы можно было легко приспособить к эффективному выполнению конкретных задач.

Компьютерные программы часто чрезмерно оптимизированы. Обеспечение наиболее эффективной реализации конкретного алгоритма может не стоить затраченных усилий, если только алгоритм не должен использоваться для решения очень сложной задачи или же многократно. В противном случае вполне достаточно качественной, сравнительно простой реализации: достаточно быть уверенным в ее работоспособности и в том, что, скорее всего, в худшем случае она будет работать в 5–10 раз медленнее наиболее эффективной версии, что может означать увеличение времени вы-

полнения на несколько дополнительных секунд. И напротив, правильный выбор алгоритма может ускорить работу в 100–1000 и более раз, что может вылиться во время выполнения в экономию минут, часов и даже более того. В этой книге основное внимание уделяется простейшим приемлемым реализациям наилучших алгоритмов.

Выбор наилучшего алгоритма выполнения конкретной задачи может оказаться сложным процессом, возможно, требующим сложного математического анализа. Направление компьютерных наук, занимающееся изучением подобных вопросов, называется *анализом алгоритмов*. Анализ многих изучаемых алгоритмов показывает, что они имеют прекрасную производительность; о хорошей работе других известно просто из опыта их применения. Наша основная цель — изучение приемлемых алгоритмов выполнения важных задач, хотя значительное внимание будет уделено также сравнительной производительности различных методов. Не следует использовать алгоритм, не имея представления о ресурсах, которые могут потребоваться для его выполнения, поэтому мы стремимся знать, как могут выполняться используемые алгоритмы.

## 1.2 Пример задачи: связность

Предположим, что имеется последовательность пар целых чисел, в которой каждое целое число представляет объект некоторого типа, а пара **p-q** интерпретируется в значении "p связано с q". Мы предполагаем, что отношение "связано с" является транзитивным: если p связано с q, а q связано с r, то p связано с r. Задача состоит в написании программы для исключения лишних пар из набора: когда программа вводит пару **p-q**, она должна выводить эту пару только в том случае, если просмотренные до данного момента пары *не предполагают*, что p связано с q. Если в соответствии с ранее просмотренными парами следует, что p связано с q, программа должна игнорировать пару **p-q** и переходить ко вводу следующей пары. Пример такого процесса показан на рис. 1.1.

Задача состоит в разработке программы, которая может запомнить достаточный объем информации о просмотренных парах, чтобы решить, связана ли новая пара объектов. Достаточно неформально задачу разработки такого метода мы называем *задачей связности*. Эта задача возникает в ряде важных приложений. Для подтверждения всеобщего характера этой задачи мы кратко рассмотрим три примера.

Например, целые числа могли бы представлять компьютеры в большой сети, а пары могли бы представлять соединения в сети. Тогда такая программа могла бы использоваться для опреде-

3-4	3-4	
4-9	4-9	
8-0	8-0	
2-3	2-3	
5-6	5-6	
2-9		2-3-4-9
5-9	5-9	
7-3	7-3	
4-8	4-8	
5-6		5-6
0-2		0-8-4-3-2
6-1	6-1	

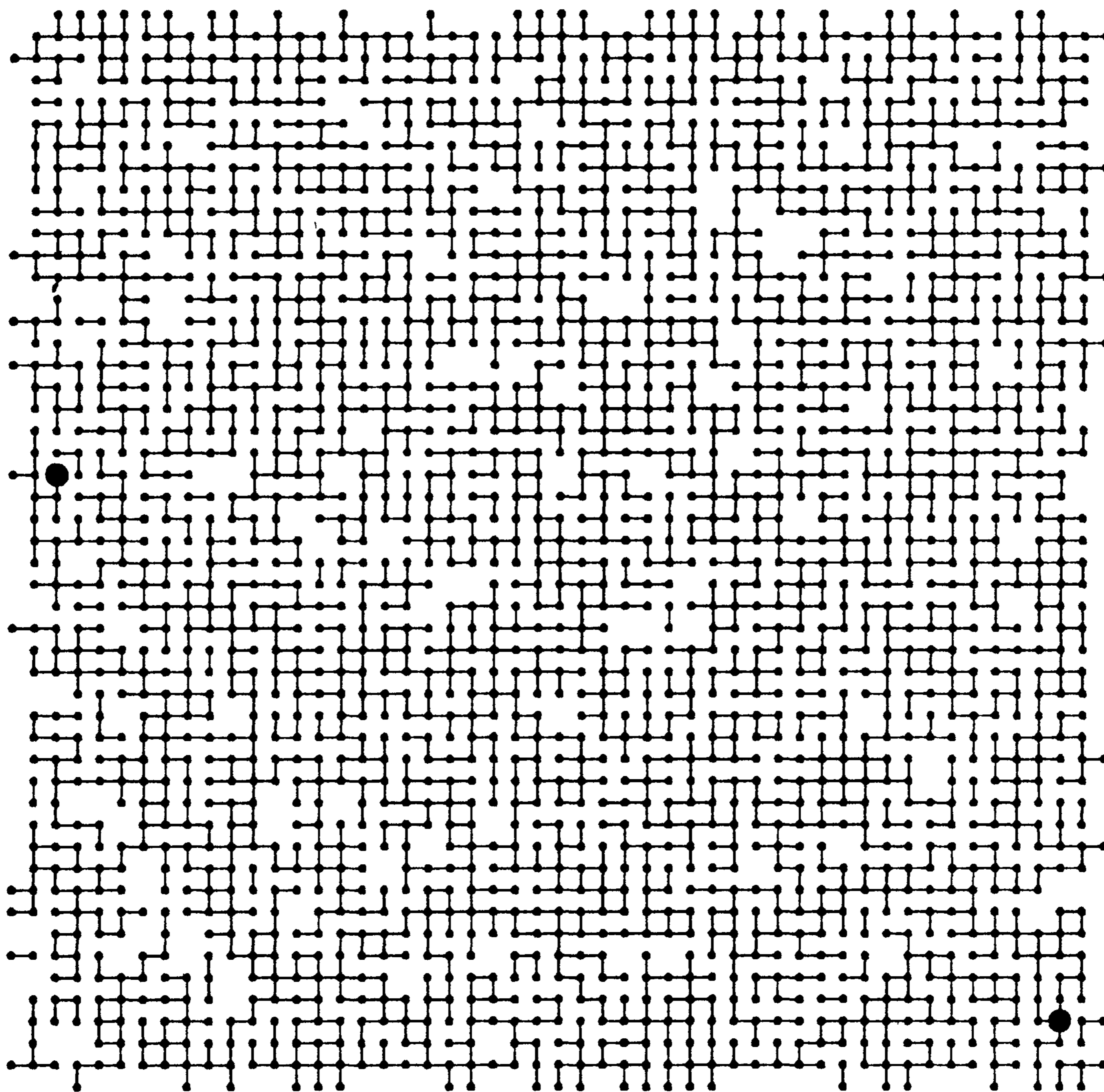
### РИСУНОК 1.1 ПРИМЕР СВЯЗНОСТИ

При заданной последовательности пар целых чисел, представляющих связь между объектами (слева) задачей алгоритма связности заключается в выводе тех пар, которые обеспечивают новые связи (в центре). Например, пара 2-9 не должна выводиться, поскольку связь 2-3-4-9 определяется ранее указанными связями (подтверждение этого показано справа).

ления того, нужно ли устанавливать новое прямое соединение между  $p$  и  $q$ , чтобы иметь возможность обмениваться информацией, или же можно было бы использовать существующие соединения для установки коммуникационного пути. В подобных приложениях может потребоваться обработка миллионов точек и миллиардов или более соединений. Как мы увидим, решить задачу для такого приложения было бы невозможно без эффективного алгоритма.

Аналогично, целые числа могли бы представлять контакты в электрической сети, а пары могли бы представлять связывающие их проводники. В этом случае программу можно было бы использовать для определения способа соединения всех точек без каких-либо избыточных соединений, если это возможно. Не существует никакой гарантии, что ребер списка окажется достаточно для соединения всех точек — действительно, вскоре мы увидим, что определение факта, так ли это, может быть основным применением нашей программы.

Рисунок 1.2 иллюстрирует эти два типа приложений на более сложном примере. Изучение этого рисунка дает представление о сложности задачи связности: как можно быстро выяснить, являются ли *любые* две заданные точки в такой сети связанными?



### РИСУНОК 1.2 БОЛЬШОЙ ПРИМЕР СВЯЗНОСТИ

*Объекты, задействованные в задаче связности, могут представлять точки соединений, а пары могут быть соединениями между ними, как показано в этом идеализированном примере, который мог бы представлять провода, соединяющие здания в городе или компоненты в компьютерной микросхеме. Это графическое представление позволяет человеку выявить несвязанные узлы, но алгоритм должен работать только с переданными ему парами целых чисел. Связаны ли два узла, помеченные черными точками?*



Еще один пример встречается в определенных средах программирования, в которых два имени переменных можно объявлять эквивалентными. Задача заключается в возможности определения, являются ли два заданных имени эквивалентными, после считывания последовательности таких объявлений. Это приложение — одно из первых, обусловивших разработку нескольких алгоритмов, которые мы будем рассматривать. Как будет показано далее, оно устанавливает непосредственную связь между рассматриваемой задачей и простой абстракцией, предоставляющей способ сделать алгоритмы полезными для широкого множества приложений.

Такие приложения, как задача установления эквивалентности имен переменных, описанная в предыдущем абзаце, требует, чтобы целое число было сопоставлено с каждым отдельным именем переменной. Это сопоставление подразумевается также в описанных приложениях сетевого соединения и соединения в электрической цепи. В главах 10–16 мы рассмотрим ряд алгоритмов, которые могут эффективно обеспечить такое сопоставление. Таким образом, в этой главе без ущерба для общности можно предположить, что имеется  $N$  объектов с целочисленными именами от 0 до  $N - 1$ .

Нам требуется программа, которая выполняет конкретную, вполне определенную задачу. Существует множество других связанных с этой задачей, решение которых также может потребоваться. Одна из первых задач, с которой приходится сталкиваться при разработке алгоритма — необходимость убедиться, что *задача* определена приемлемым образом. Чем больше требуется от алгоритма, тем больше времени и объема памяти может потребоваться для выполнения задачи. Это соотношение невозможно в точности определить заранее, и часто определение задачи приходится изменять, когда выясняется, что ее трудно решить либо решение требует слишком больших затрат, или же когда, при удачном стечении обстоятельств, выясняется, что алгоритм может предоставить более полезную информацию, чем требовалось от него в исходном определении.

Например, приведенное определение задачи связности требует только, чтобы программа как-либо узнавала, является ли данная пара  $p$ - $q$  связанной, но не была способна демонстрировать любой или все способы соединения этой пары. Добавление в определение такого требования усложняет задачу и привело бы к другому семейству алгоритмов, которое будет кратко рассматриваться в главе 5 и подробно — в части 7.

Упомянутые в предыдущем абзаце определения требуют *больше* информации, чем первоначальное; может также требоваться *меньше* информации. Например, может потребоваться просто ответить на вопрос: "Достаточно ли  $M$  связей для соединения всех  $N$  объектов?". Эта задача служит иллюстрацией того, что для разработки эффективных алгоритмов часто требуется выполнение умозаключений об абстрактных обрабатываемых объектах на высоком уровне. В данном случае из фундаментальных положений теории графов следует, что все  $N$  объектов связаны тогда и только тогда, когда количество пар, образованных алгоритмом решения задачи связности, равно точно  $N - 1$  (см. раздел 5.4). Иначе говоря, алгоритм решения задачи связности никогда не образует более  $N - 1$  пар, поскольку как только он образует  $N - 1$  пару, любая встретившаяся после этого пара будет уже связанной. Соответственно, можно создать программу, отвечающую "да-нет" на только что поставленный вопрос, изменив программу, которая решает задачу связности, на такую, которая увеличивает значение

счетчика, а не записывает ранее не связанную пару, отвечая "да", когда значение счетчика достигает  $N - 1$ , и "нет", если это не происходит. Этот вопрос — всего лишь один из множества, которые могут возникнуть относительно связности. Входной набор пар называется *графом* (*Graph*), а выходной набор пар — *остовным деревом* (*spanning tree*) этого графа, которое связывает все объекты. Свойства графов, остовных деревьев и всевозможные связанные с ними алгоритмы будут рассматриваться в части 7.

Имеет смысл попытаться определить основные операции, которые будут выполняться, и таким образом сделать любой алгоритм, разрабатываемый для решения задачи связности, полезным для ряда аналогичных задач. В частности, при получении каждой новой пары вначале необходимо определить, представляет ли она новое соединение, а затем внедрить информацию об обнаруженном соединении в общую картину о связности объектов для проверки соединений, которые будут наблюдаться в будущем. Мы инкапсулируем эти две задачи в виде *абстрактных операций*, считая целочисленные вводимые значения представляющими элементы в абстрактных наборах, а затем разработаем алгоритмы и структуры данных, которые могут:

- *находить* набор, содержащий данный элемент
- *замещать* наборы, содержащие два данных элемента, их *объединением*.

Организация алгоритмов посредством этих абстрактных операций, похоже, не препятствует никаким опциям решения задачи связности, а сами эти операции могут оказаться полезными при решении других задач. Разработка уровней абстракции с еще большими возможностями — основной процесс в компьютерных науках в целом и в разработке алгоритмов в частности, и в этой книге мы будем обращаться к нему многократно. В этой главе мы используем неформальное абстрактное представление для разработки программ решения задачи связности; внедрение абстракций в код C++ демонстрируется в главе 4.

Задача связности легко решается посредством абстрактных операций *find* (*поиск*) и *union* (*объединение*). После считывания новой пары  $p$ - $q$  из ввода мы выполняем операцию *find* для каждого члена пары. Если члены пары находятся в одном наборе, мы переходим к следующей паре; если нет, то выполняем операцию *union* и записываем пару. Наборы представляют *связанные компоненты*: поднаборы объектов, характеризующиеся тем, что любые два объекта в данном компоненте связаны. Этот подход сводит разработку алгоритмического решения задачи связности к задачам определения структуры данных, которая представляет наборы, и разработке алгоритмов *union* и *find*, которые эффективно используют эту структуру данных.

Существует много возможных способов представления и обработки абстрактных наборов. В этой главе основное внимание уделяется поиску представления, которое может эффективно поддерживать операции *union* и *find*, требующиеся для решения задачи связности.

## Упражнения

1.1 Приведите вывод, который должен создаваться алгоритмом связности при заданном вводе 0-2, 1-4, 2-5, 3-6, 0-4, 6-0 и 1-3.

1.2 Перечислите все различные способы связывания двух различных объектов, показанных в примере на рис. 1.1.

1.3 Опишите простой метод подсчета количества наборов, остающихся после использования операций *union* и *find* для решения задачи связности, как описано в тексте.

## 1.3 Алгоритмы объединение-поиск

Первый шаг в процессе разработки эффективного алгоритма решения данной задачи — *реализация простого алгоритма ее решения*. Если нужно решить несколько вариантов конкретной задачи, которые оказываются простыми, это может быть выполнено посредством простой реализации. Если требуется более сложный алгоритм, то простая реализация предоставляет возможность проверить правильность работы алгоритма для простых случаев и может служить отправной точкой для оценки характеристик производительности. Эффективность всегда является предметом заботы, но наша первоочередная забота при разработке первой программы решения задачи — убедиться в том, что программа обеспечивает *правильное* решение.

Первое что приходит в голову — как-либо сохранить все вводимые пары, а затем создать функцию для их просмотра, чтобы попытаться выяснить, связана ли следующая пара объектов. Однако нам придется использовать другой подход. Во-первых, количество пар может быть достаточно велико, что не позволит сохранить их все в памяти в используемом на практике приложении. Во-вторых, что гораздо важнее, не существует никакого простого метода, который сам по себе позволяет определить, связаны ли два объекта в наборе всех соединений, даже если бы удалось их все сохранить! Базовый метод, использующий этот подход, рассматривается в главе 5, методы, которые исследуются в этой главе, проще, поскольку они решают менее сложную задачу, и эффективнее, поскольку не требуют сохранения всех пар. Все они используют массив целых чисел, каждое из которых соответствует отдельному объекту, для хранения информации, необходимой для реализации операций *union* и *find*.

Массивы — это элементарные структуры данных, которые подробно изучаются в разделе 3.2. Здесь же они используются в простейшей форме: мы объявляем, что собираемся использовать, скажем, 1000 целых чисел, записывая **a[1000]**; затем обращаемся к *i*-ому целому числу в массиве, записывая **a[i]** для  $0 \leq i < 1000$ .

Программа 1.1 — реализация простого алгоритма, называемого *алгоритмом быстрого поиска*, решающего задачу связности. В основе этого алгоритма лежит использование массива целых чисел, обладающих тем свойством, что *p* и *q* связаны тогда и только тогда, когда *p*-ая и *q*-ая записи массива равны. Мы инициализируем *i*-ую запись массива значением *i* при  $0 \leq i < N$ . Чтобы реализовать операцию *union* для *p* и *q*, мы просматриваем массив, изменяя все записи с именем *p* на записи с именем *q*. Этот выбор произволен — можно было бы все записи с именем *q* изменять на записи с именем *p*.

### Программа 1.1 Решение задачи связности методом быстрого поиска

Эта программа считывает последовательность пар неотрицательных целых чисел, меньших чем *N*, из стандартного ввода (интерпретируя пару **p q**, как "связать объект **p** с объектом **q**") и выводит пары, представляющие объекты, которые еще не связаны. Она поддерживает массив **id**, содержащий запись для каждого объекта и характеризующийся тем, что элементы **id[p]** и **id[q]** равны тогда и только тогда, когда

объекты  $p$  и  $q$  связаны. Для простоты  $N$  определена как константа времени компиляции. Иначе можно было бы считывать ее из ввода и распределять массив  $id$  динамически (см. раздел 3.2).

```
#include <iostream.h>
static const int N = 10000;
int main()
{ int i, p, q, id[N];
  for (i = 0; i < N; i++) id[i] = i;
  while (cin >> p >> q)
  { int t = id[p];
    if (t == id[q]) continue;
    for (i = 0; i < N; i++)
      if (id[i] == t) id[i] = id[q];
    cout << " " << p << " " << q << endl;
  }
}
```

Изменения массива при выполнении операций *union* в примере из рис. 1.1 показаны на рис. 1.3. Для реализации операции *find* достаточно проверить указанные записи массива на предмет равенства — отсюда и название *быстрый поиск (quick-find)*. С другой стороны, операция *union* требует просмотра всего массива для каждой вводимой пары.

**Лемма 1.1** Алгоритм быстрого поиска выполняет не менее  $M N$  инструкций для решения задачи связности при наличии  $N$  объектов, для которых требуется выполнение  $M$  операций объединения.

Для каждой из  $M$  операций *union* цикл **for** выполняется  $N$  раз. Для каждой итерации требуется выполнение, по меньшей мере, одной инструкции (если только проверять, завершился ли цикл).

На современных компьютерах можно выполнять десятки или сотни миллионов инструкций в секунду, поэтому эти затраты не заметны, если значения  $M$  и  $N$  малы, но в современных приложениях может потребоваться обработка миллиардов объектов и миллионов вводимых пар. Поэтому мы неизбежно приходим к заключению, что подобную проблему нельзя решить приемлемым образом, используя алгоритм быстрого поиска (см. упражнение 1.10). Строгое обоснование этого заключения приведено в главе 2.

Графическое представление массива, показанного на рис. 1.3, приведено на рис. 1.4. Можно считать, что некоторые объекты представляют набор, к которому они принадлежат, а остальные указывают на представителя их набора. Причина обращения к этому графическому представлению массива вскоре станет понятна. Обратите внимание, что связи между объектами в этом представлении не обязательно со-

$p$	$q$	0	1	2	3	4	5	6	7	8	9
3	4	0	1	2	4	4	5	6	7	8	9
4	9	0	1	2	9	9	5	6	7	8	9
8	0	0	1	2	9	9	5	6	7	0	9
2	3	0	1	9	9	9	5	6	7	0	9
5	6	0	1	9	9	9	6	6	7	0	9
2	9	0	1	9	9	9	6	6	7	0	9
5	9	0	1	9	9	9	9	9	7	0	9
7	3	0	1	9	9	9	9	9	9	0	9
4	8	0	1	0	0	0	0	0	0	0	0
5	6	0	1	0	0	0	0	0	0	0	0
0	2	0	1	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	1	1	1	1	1

### РИСУНОК 1.3 ПРИМЕР БЫСТРОГО ПОИСКА (МЕДЛЕННОГО ОБЪЕДИНЕНИЯ)

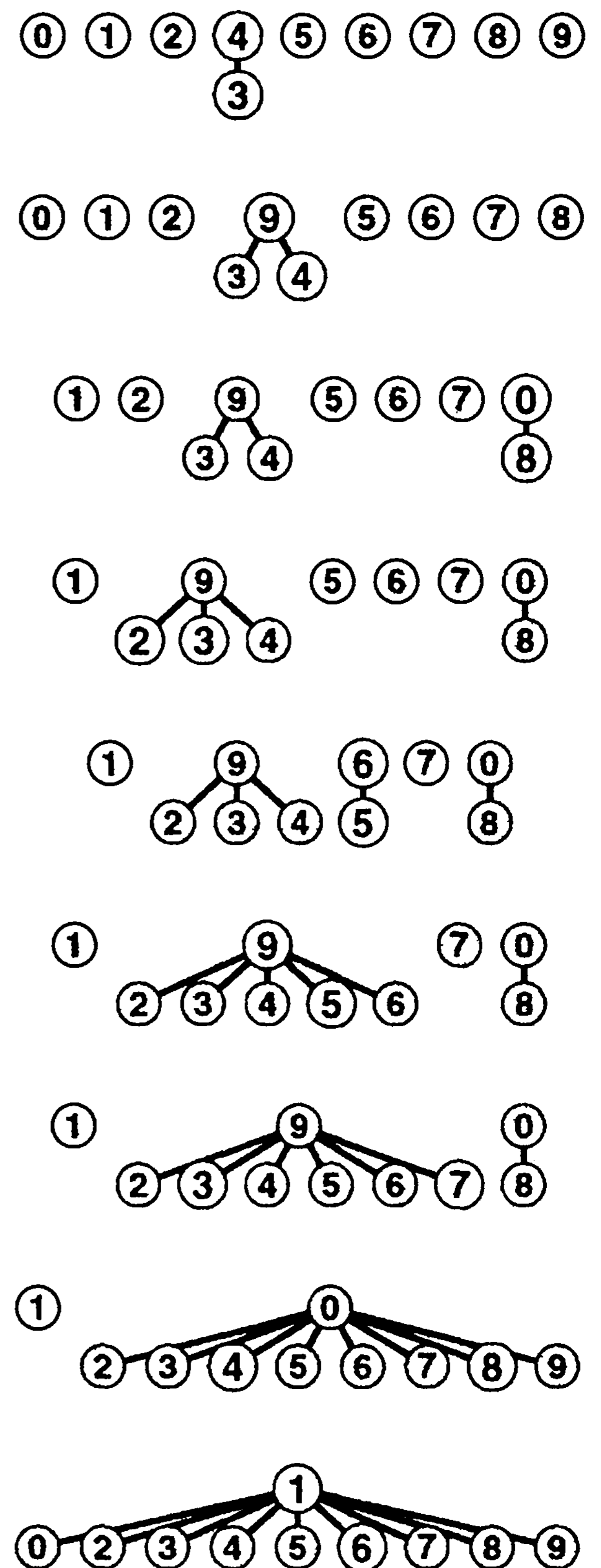
Здесь изображено содержимое массива  $id$  после того, как каждая пара, приведенная слева, обрабатывается алгоритмом быстрого поиска (программа 1.1). Затененные записи — те, которые изменяются для выполнения операции *union*. При обработке пары  $p$   $q$  все записи со значением  $id[p]$  изменяются на содержащие значение  $id[q]$ .

ответствует связям во вводимых парах — они представляют собой информацию, запоминаемую алгоритмом, чтобы иметь возможность определить, соединены ли пары, которые будут вводиться в будущем.

Следующий алгоритм, который мы рассмотрим — метод дополнительный к предшествующему, названный *алгоритмом быстрого объединения*. В его основе лежит та же структура данных — индексированный по именам объектов массив — но в нем используется иная интерпретация значений, что приводит к более сложной абстрактной структуре. Каждый объект указывает на другой объект в этом же наборе, структуре, которая не содержит циклов. Для определения того, находятся ли два объекта в одном наборе, мы следуем указателям для каждого из них до тех пор, пока не будет достигнут объект, который указывает на самого себя. Объекты находятся одном наборе тогда и только тогда, когда этот процесс приводит их к одному и тому же объекту. Если они не находятся в одном наборе, процесс завершится на других объектах (которые указывают на себя). Тогда для образования объединения достаточно связать один объект с другим, чтобы выполнить операцию *union*; отсюда и название *быстрое объединение (quick-union)*.

На рис. 1.5 показано графическое представление, которое соответствует рис. 1.4 при выполнении алгоритма быстрого объединения к массиву, изображенному на рис. 1.1, а на рис. 1.6 показаны соответствующие изменения в массиве *id*. Графическое представление структуры данных позволяет сравнительно легко понять действие алгоритма — вводимые пары, которые заведомо должны быть соединены в данных, связываются одна с другой и в структуре данных. Как упоминалось ранее, важно отметить, что связи в структуре данных не обязательно совпадают со связями в приложении, обусловленными вводимыми парами; вместо этого они конструируются алгоритмом так, чтобы обеспечить эффективную реализацию операций *union* и *find*.

Связанные компоненты, изображенные на рис. 1.5, называются *деревьями (tree)*; это основополагающие комбинационные структуры, которые многократно встречаются в книге. Свойства деревьев будут подробно рассмотрены в главе 5. Деревья, изображенные на рис. 1.5, удобны для выполнения операций *union* и *find*, поскольку их можно быстро построить, и они характеризуются тем, что два



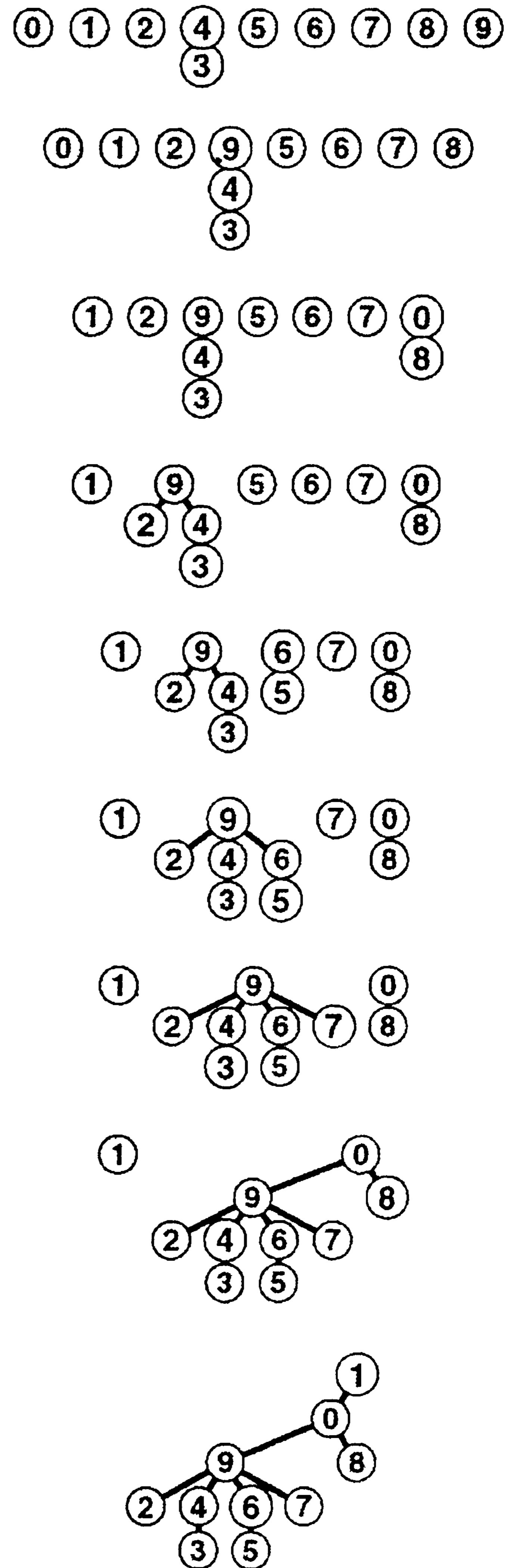
**РИСУНОК 1.4 ПРЕДСТАВЛЕНИЕ БЫСТРОГО ПОИСКА В ВИДЕ ДЕРЕВА**

На этом рисунке показано графическое представление примера, приведенного на рис. 1.3. Связи на этом рисунке не обязательно представляют связи в массиве ввода. Например, структура, показанная на нижнем рисунке, содержит связь 1-7, которая отсутствует во вводе, но образуется в результате строки связей 7-3-4-9-5-6-1.

объекта связаны в дереве тогда и только тогда, когда объекты связаны в массиве ввода. Перемещаясь вверх по дереву, можно легко отыскать корень дерева, содержащего каждый объект, и, следовательно, можно выяснить, связаны они или нет. Каждое дерево содержит только один объект, указывающий сам на себя, называемый *корнем* дерева. Указатель на себя на диаграммах не показан. Начав с любого объекта дерева и перемещаясь к объекту, на который он указывает, затем следующему указанному объекту и т.д., мы всегда со временем попадаем к корню. Справедливость этого свойства можно доказать методом индукции: это справедливо после инициализации массива так, чтобы каждый объект указывал на себя, и если это справедливо перед выполнением данной операции *union*, это безусловно справедливо и после нее.

Диаграммы для алгоритма быстрого поиска, показанные на рис. 1.4, характеризуются теми же свойствами, которые описаны в предыдущем абзаце. Различие состоит в том, что в деревьях быстрого поиска мы достигаем корня из всех узлов, следуя лишь одной связи, в то время как в дереве быстрого объединения для достижения корня может потребоваться проследовать по нескольким связям.

Программа 1.2 — реализация операций *union* и *find*, образующих алгоритм быстрого объединения для решения задачи связности. На первый взгляд кажется, что алгоритм быстрого объединения работает быстрее алгоритма быстрого поиска, поскольку для каждой вводимой пары ему не нужно просматривать весь массив; но на сколько быстрее? В данном случае ответить на этот вопрос труднее, чем в случае быстрого поиска, поскольку время выполнения в большей степени зависит от характера ввода. Выполнив экспериментальные исследования или математический анализ (см. главу 2), можно убедиться, что программа 1.2 значительно эффективнее программы 1.1 и ее можно использовать для решения очень сложных реальных задач. Одно из таких экспериментальных исследований будет рассмотрено в конце этого раздела.



**РИСУНОК 1.5 ПРЕДСТАВЛЕНИЕ БЫСТРОГО ОБЪЕДИНЕНИЯ В ВИДЕ ДЕРЕВА**

Этот рисунок — графическое представление примера, показанного на рис. 1.3. Мы прочерчиваем линию от объекта  $i$  к объекту  $id[i]$ .

### РИСУНОК 1.6 ПРИМЕР БЫСТРОГО ОБЪЕДИНЕНИЯ (НЕ ОЧЕНЬ БЫСТРОГО ПОИСКА)

Здесь изображено содержимое массива *id* после обработки каждой из показанных слева пар алгоритмом быстрого поиска (программа 1.1). Затененные записи — те, которые изменяются для выполнения операции объединения (только по одной для каждой операции). При обработке пары *p q* мы следуем указателям, указывающим из *p*, чтобы добраться до записи *i*, у которой  $id[i] == i$ ; затем мы следуем указателям, исходящим из *q*, чтобы добраться до записи *j*, у которой  $id[j] == j$ ; затем, если *i* и *j* различны, мы устанавливаем  $id[i] = id[j]$ . Для выполнения операции *find* для пары 5-8 (последняя строка) *i* принимает значения 5 6 9 0 1, а *j* — значения 8 0 1.

<i>p</i>	<i>q</i>	0	1	2	3	4	5	6	7	8	9
3	4	0	1	2	4	4	5	6	7	8	9
4	9	0	1	2	4	9	5	6	7	8	9
8	0	0	1	2	4	9	5	6	7	0	9
2	3	0	1	9	4	9	5	6	7	0	9
5	6	0	1	9	4	9	6	6	7	0	9
2	9	0	1	9	4	9	6	6	7	0	9
5	9	0	1	9	4	9	6	9	7	0	9
7	3	0	1	9	4	9	6	9	9	0	9
4	8	0	1	9	4	9	6	9	9	0	0
5	6	0	1	9	4	9	6	9	9	0	0
0	2	0	1	9	4	9	6	9	9	0	0
6	1	1	1	9	4	9	6	9	9	0	0
5	8	1	1	9	4	9	6	9	9	0	0

А пока быстрое объединение можно считать усовершенствованием, поскольку оно устраняет основной недостаток быстрого поиска (тот, что для выполнения *M* операций *union* между *N* объектами программе требуется выполнение, по меньшей мере,  $N M$  инструкций).

### Программа 1.2 Решение задачи связности методом быстрого объединения

Если тело цикла **while** в программе 1.1 заменить этим кодом, мы получим программу, которая соответствует тем же спецификациям, что и программа 1.1, но выполняет меньше вычислений для операции *union* за счет выполнения большего количества вычислений для операции *find*. Циклы **for** и последующий оператор **if** в этом коде определяют необходимые и достаточные условия для того, чтобы массив *id* для *p* и *q* был связанным. Оператор присваивания  $id[i] = j$  реализует операцию *union*.

```
for (i = p; i != id[i]; i = id[i]) ;
for (j = q; j != id[j]; j = id[j]) ;
if (i == j) continue;
id[i] = j;
cout << " " << p << " " << q << endl;
```

Это различие между быстрым объединением и быстрым поиском действительно является усовершенствованием, но быстрое объединение все же обладает тем недостатком, что нельзя *гарантировать*, что оно будет выполняться существенно быстрее быстрого поиска в каждом случае, поскольку характер вводимых данных может замедлять операцию *find*.

**Лемма 1.2** При наличии *M* пар *N* объектов, когда  $M > N$ , для решения задачи связности алгоритму быстрого объединения может потребоваться выполнение более чем  $M N / 2$  инструкций.

Предположим, что пары вводятся в следующем порядке: 1-2, 2-3, 3-4 и т.д. После ввода  $N - 1$  таких пар мы имеем *N* объектов, принадлежащих к одному набору, и сформированное алгоритмом быстрого объединения дерево представляет собой прямую линию, где объект *N* указывает на объект  $N - 1$ , тот, в свою очередь, — на объект  $N - 2$ , тот — на  $N - 3$  и т.д. Чтобы выполнить операцию *find* для объекта *N*, программа должна отследить  $N - 1$  указатель.

Таким образом, среднее количество указателей, отслеживаемых для первых  $N$  пар, равно

$$(0 + 1 + \dots + (N - 1)) / N = (N - 1) / 2$$

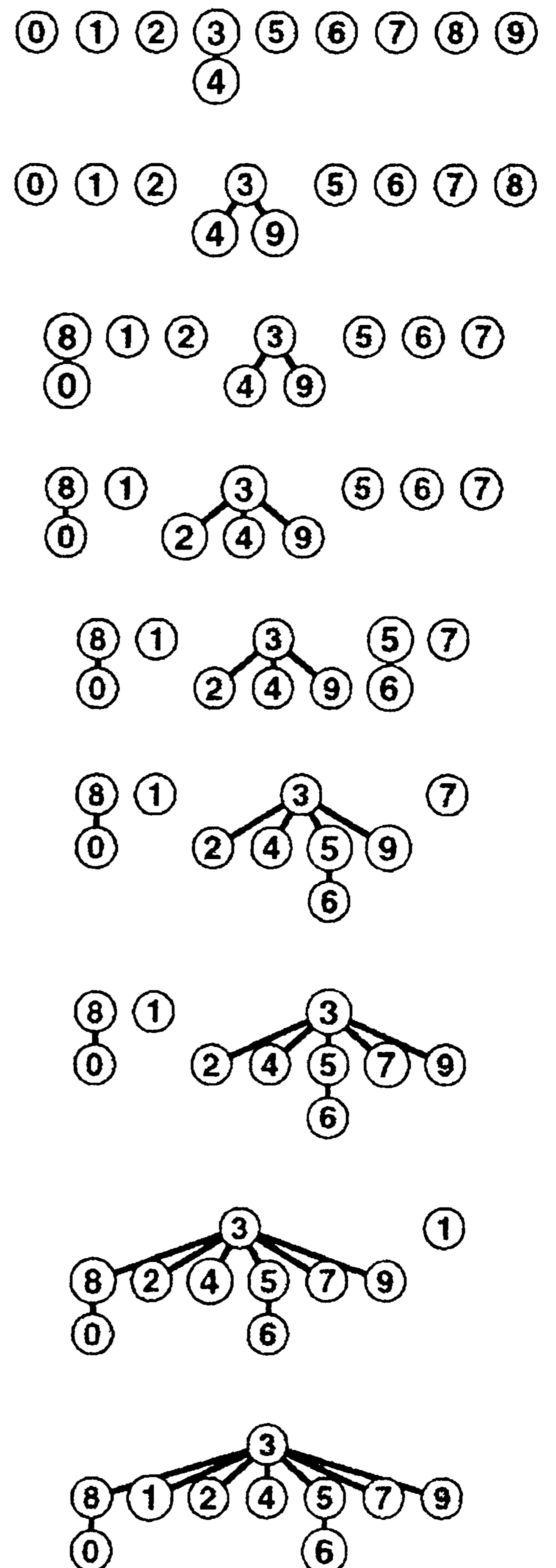
Теперь предположим, что все остальные пары связывают объект  $N$  с каким-либо другим объектом. Чтобы выполнить операцию *find* для каждой из этих пар, требуется отследить, по меньшей мере,  $(N - 1)$  указатель. Общая сумма для  $M$  операций *find* для этой последовательности вводимых пар определено больше  $MN / 2$ .

К счастью, можно легко модифицировать алгоритм, чтобы худшие случаи, подобные этому, гарантированно не имели места. Вместо того чтобы произвольным образом соединять второе дерево с первым для выполнения операции *union*, можно отслеживать количество узлов в каждом дереве и всегда соединять меньшее дерево с большим. Это изменение требует несколько более объемного кода и наличия еще одного массива для хранения счетчиков узлов, как показано в программе 1.3, но оно ведет к существенному повышению эффективности. Мы будем называть этот алгоритм *алгоритмом взвешенного быстрого объединения* (*weighted quick-union algorithm*).

### Программа 1.3 Взвешенная версия быстрого объединения

Эта программа — модификация алгоритма быстрого объединения (см. программу 1.2), которая в служебных целях для каждого объекта, у которого  $\text{id}[i] == i$ , поддерживает дополнительный массив  $\text{sz}$ , представляющий собой массив количества узлов в соответствующем дереве, чтобы операция *union* могла связывать меньшее из двух указанных деревьев с большим, тем самым предотвращая разрастание длинных путей в деревьях.

```
#include <iostream.h>
static const int N = 10000;
int main()
{ int i, j, p, q, id[N], sz[N];
  for (i = 0; i < N; i++)
    { id[i] = i; sz[i] = 1; }
  while (cin >> p >> q)
  {
    for (i = p; i != id[i]; i = id[i]);
    for (j = q; j != id[j]; j = id[j]);
    if (i == j) continue;
    if (sz[i] < sz[j])
      { id[i] = j; sz[j] += sz[i]; }
    else { id[j] = i; sz[i] += sz[j]; }
    cout << " " << p << " " << q << endl;
  }
}
```



### РИСУНОК 1.7 ПРЕДСТАВЛЕНИЕ ВЗВЕШЕННОГО БЫСТРОГО ОБЪЕДИНЕНИЯ В ВИДЕ ДЕРЕВА

На этой последовательности рисунков демонстрируется результат изменения алгоритма быстрого объединения для связывания корня меньшего из двух деревьев с корнем большего из деревьев. Расстояние от каждого узла до корня его дерева не велико, поэтому операция поиска выполняется эффективно.

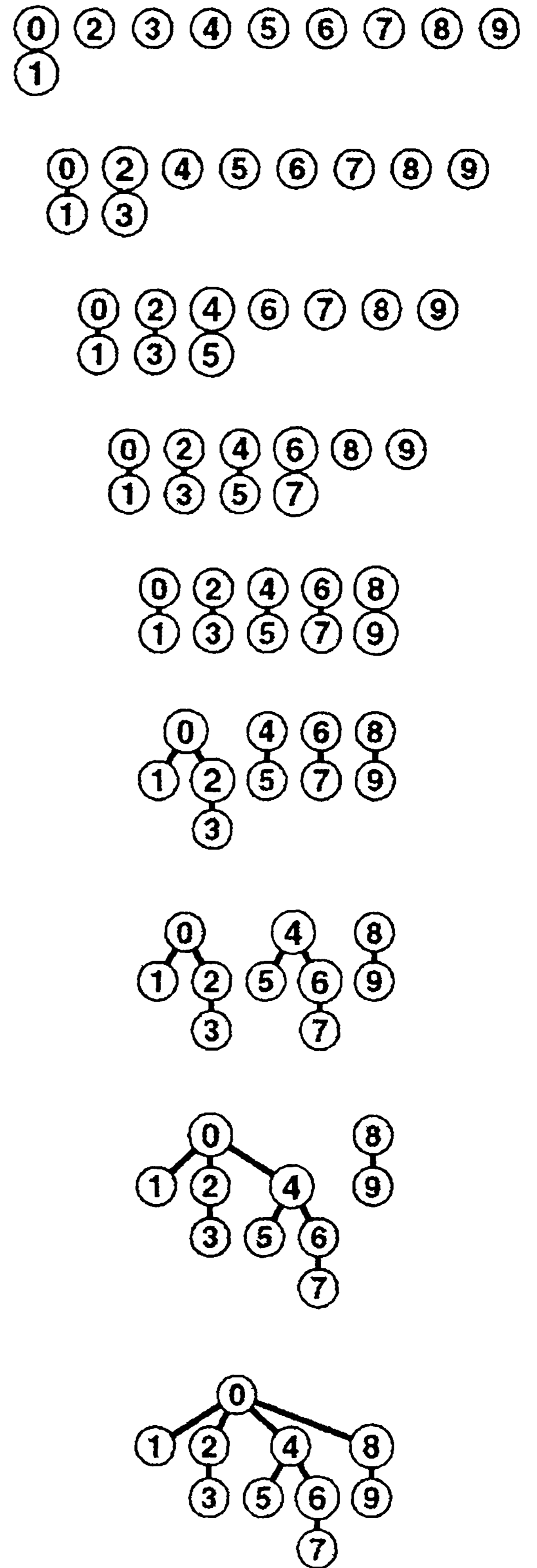


На рис. 1.7 показан бор деревьев, созданных алгоритмом взвешенного поиска для примера ввода, приведенного на рис. 1.1. Даже в этом небольшом примере пути в деревьях существенно короче, чем в случае невзвешенной версии, показанной на рис. 1.5. Рисунок 1.8 иллюстрирует, что происходит в худшем случае, когда размеры наборов, которые должны быть объединены в операции *union*, всегда равны (и являются степенью 2). Эти структуры деревьев выглядят сложными, но они характеризуются простым свойством, что максимальное количество указателей, которые необходимо отследить, чтобы добраться до корня в дереве, состоящем из  $2^n$  узлов, равно  $n$ . Более того, при слиянии двух деревьев, состоящих из  $2^n$  узлов, мы получаем дерево, состоящее из  $2^{n+1}$  узлов, а максимальное расстояние до корня увеличивается до  $n + 1$ . Это наблюдение можно обобщить для доказательства того, что взвешенный алгоритм значительно эффективнее невзвешенного.

**Лемма 1.3** Для определения того, связаны ли два из  $N$  объектов, алгоритму взвешенного быстрого объединения требуется отследить максимум  $\lg N$  указателей.

Можно доказать, что для операции *union* сохраняется свойство, что количество указателей, отслеживаемых из любого узла до корня в наборе  $k$  объектов, не превышает  $\lg k$ . При объединении набора, состоящего из  $i$  узлов, с набором, состоящим из  $j$  узлов, при  $i \leq j$  количество указателей, которые должны отслеживаться в меньшем наборе, увеличивается на 1, но теперь узлы находятся в наборе размера  $i + j$ , и, следовательно, свойство остается справедливым, поскольку  $1 + \lg i = \lg(i + i) \leq \lg(i + j)$ .

Практическое следствие леммы 1.3 заключается в том, что количество инструкций, которые алгоритм взвешенного быстрого объединения использует для обработки  $M$  ребер между  $N$  объектами, не превышает  $M \lg N$ , умноженного на некоторую константу (см. упражнение 1.9). Этот вывод резко отличается от вывода, что алгоритм быстрого поиска всегда (а алгоритм быстрого объединения иногда) использует не менее  $MN/2$  инструкций. Таким образом, при использовании взвешенного быстрого объединения можно гарантировать



**РИСУНОК 1.8 ВЗВЕШЕННОЕ БЫСТРОЕ ОБЪЕДИНЕНИЕ (ХУДШИЙ СЛУЧАЙ)**

Наихудший сценарий для алгоритма взвешенного быстрого объединения — когда каждая операция объединения связывает деревья одинакового размера. Если количество объектов меньше  $2^n$ , расстояние от любого узла до корня его дерева меньше  $n$ .

решение очень сложных встречающихся на практике задач за приемлемое время (см. упражнение 1.11). Ценой добавления нескольких дополнительных строк кода мы получаем программу, которая при решении очень сложных задач, которые могут встретиться на практике, работает буквально в миллионы раз быстрее, чем более простые алгоритмы.

Из приведенных схем видно, что лишь сравнительно небольшое количество узлов располагаются далеко от корня; действительно, экспериментальное изучение очень сложных задач показывает, что как правило, для решения практических задач посредством использования алгоритма взвешенного быстрого объединения, реализованного в программе 1.3, требуется *линейное* время. То есть затраты времени на выполнение алгоритма связаны с затратами времени на считывание ввода постоянным коэффициентом. Вряд ли можно было бы рассчитывать найти более эффективный алгоритм.

Немедленно возникает вопрос, можно ли найти алгоритм, обеспечивающий *гарантированную* линейную производительность. Этот вопрос — исключительно трудный, который уже много лет не дает покоя исследователям (см. раздел 2.7). Существует множество способов дальнейшего совершенствования алгоритма взвешенного быстрого объединения. В идеале было бы желательно, чтобы каждый узел указывал непосредственно на корень своего дерева, но за это не хотелось бы расплачиваться изменением большого количества указателей, как это делалось в случае алгоритма быстрого объединения. К идеалу можно приблизиться, просто делая все проверяемые узлы указывающими на корень. На первый взгляд этот шаг кажется весьма радикальным, но его легко реализовать, а в структуре этих деревьев нет ничего неприкосновенного: если их можно изменить, чтобы сделать алгоритм более эффективным, это следует сделать. Этот метод, названный *сжатием пути (path compression)*, можно легко реализовать, добавляя еще один проход по каждому пути во время выполнения операции *union* и устанавливая запись *id*, соответствующую каждой встреченной вершине, указывающей на корень. В результате деревья становятся почти совершенно плоски-

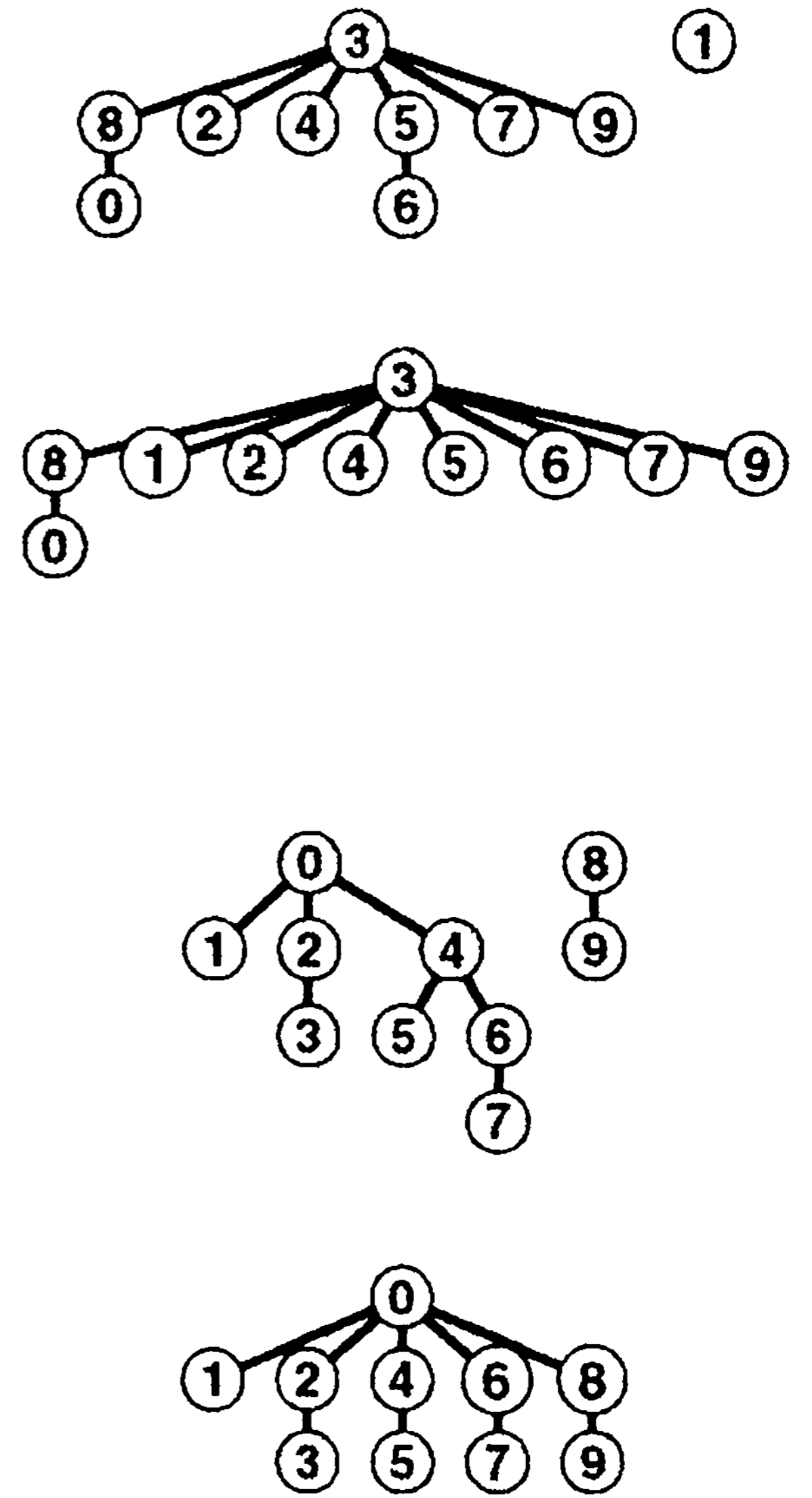


РИСУНОК 1.9 СЖАТИЕ ПУТИ

*Пути в деревьях можно сделать еще короче, просто делая все просматриваемые объекты указывающими на корень нового дерева для операции объединения, как показано в этих двух примерах. В примере на верхнем рисунке показан результат, соответствующий рис.*

*1.7. В случае коротких путей сжатие пути не оказывает никакого влияния, но при обработке пары 1 6, мы делаем узлы 1, 5 и 6 указывающими на узел 3, в результате чего дерево становится более плоским, чем на рис 1.7 В примере на нижнем рисунке показан результат, соответствующий рис.*

*1.8. В деревьях могут появляться пути, которые содержат больше одной-двух связей, но при каждом их прохождении мы делаем их более плоскими. В данном случае при обработке пары 6 8 мы делаем дерево более плоским, делая узлы 4, 6 и 8 указывающими на узел 0.*

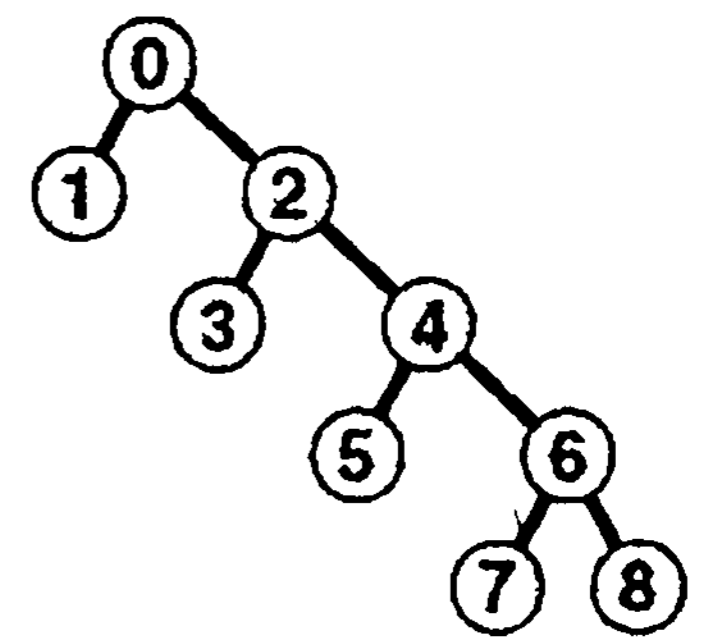
ми, приближаясь к идеалу, обеспечиваемому алгоритмом быстрого поиска, как показано на рис. 1.9. Анализ, устанавливающий этот факт, исключительно сложен, но сам метод прост и эффективен. Результат сжатия пути для большого примера показан на рис. 1.11.

Существует множество других способов реализации сжатия пути. Например, программа 1.4 — реализация, которая сжимает пути, делая каждую связь перескакивающей к следующему узлу в пути вверх по дереву (см. рис. 1.10). Этот метод несколько проще реализовать, чем полное сжатие пути (см. упражнение 1.16), но он дает тот же конечный результат. Мы называем этот вариант *взвешенным быстрым объединением посредством сжатия пути делением пополам* (*weighted quick-union with path compression by halving*). Какой из этих методов эффективнее? Оправдывает ли достигаемая экономия время, требующееся для реализации сжатия пути? Существует ли какая-либо иная технология, применение которой следовало бы рассмотреть? Чтобы ответить на эти вопросы, следует внимательно присмотреться к алгоритмам и реализациям. Мы вернемся к этой теме в главе 2 в контексте рассмотрения основных подходов к анализам алгоритмов.

#### Программа 1.4 Сжатие пути делением пополам

Если циклы `for` в программе 1.3 заменить этим кодом, длина любого проходимого пути будет делиться пополам. Конечный результат этого изменения — превращение деревьев в почти совершенно плоские после выполнения длинной последовательности операций.

```
for (i = p; i != id[i]; i = id[i])
    id[i] = id[id[i]];
for (j = q; j != id[j]; j = id[j])
    id[j] = id[id[j]];
```



**РИСУНОК 1.10 СЖАТИЕ ПУТИ ДЕЛЕНИЕМ ПОПОЛАМ**

Можно уменьшить длину путей вверх по дереву почти вдвое, прослеживая две связи одновременно и устанавливая нижнюю из них указывающей на тот же узел, что и верхняя, как показано в этом примере. Конечный результат выполнения такой операции для каждого проходимого пути приближается к результату, получаемому в результате полного сжатия пути.



**РИСУНОК 1.11 БОЛЬШОЙ ПРИМЕР ВЛИЯНИЯ СЖАТИЯ ПУТИ**

Здесь отображен результат обработки случайных пар 100 объектов алгоритмом взвешенного быстрого объединения с сжатием пути. Все узлы этого дерева, за исключением двух, находятся на расстоянии одного-двух шагов от корня.

Конечный результат применения рассмотренных алгоритмов решения задачи связности приближается к наилучшему, на который можно было бы рассчитывать в любом практическом случае. Мы имеем алгоритмы, которые легко реализовать и время выполнения которых гарантировано связано с затратами времени на сбор данных постоянным коэффициентом. Более того, алгоритмы являются *оперативными*, рассматривающими каждое ребро только один раз и использующими объем памяти, который пропорционален количеству объектов; поэтому какие-либо ограничения на количество обрабатываемых ими ребер отсутствуют. Результаты экспериментального исследования, приведенные в табл. 1.1, подтверждают вывод, что программа 1.3 и ее варианты с использованием сжатия пути полезны даже в очень больших практических приложениях. Выбор лучшего из этих алгоритмов требует тщательного и сложного анализа (см. главу 2).

**Таблица 1.1. Результаты экспериментального исследования алгоритмов объединения-поиска**

Эти сравнительные значения времени, затрачиваемого на решение случайных задач связности с использованием алгоритмов объединения-поиска, демонстрируют эффективность взвешенной версии алгоритма быстрого объединения. Дополнительный выигрыш благодаря использованию сжатия пути менее очевиден. Здесь  $M$  — количество случайных соединений, генерируемых до тех пор, пока все  $N$  объектов не оказываются связанными. Этот процесс требует значительно больше операций *find*, чем операций *union*, поэтому быстрое объединение выполняется существенно медленнее быстрого поиска. Ни быстрый поиск, ни быстрое объединение не годятся для случая, когда значение  $N$  очень велико. Время выполнения при использовании взвешенных методов явно пропорционально значению  $N$ , поскольку оно уменьшается вдвое при уменьшении  $N$  вдвое.

$N$	$M$	$F$	$U$	$W$	$P$	$H$
1000	6206	14	25	6	5	3
2500	20236	82	210	13	15	12
5000	41913	304	1172	46	26	25
10000	83857	1216	4577	91	73	50
25000	309802			219	208	216
50000	708701			469	387	497
100000	1545119			1071	1106	1096

Ключ:

- F** быстрый поиск (программа 1.1)
- U** быстрое объединение (программа 1.2)
- W** взвешенное быстрое объединение (программа 1.3)
- P** взвешенное быстрое объединение с сжатием пути (упражнение 1.16)
- H** взвешенное быстрое объединение с делением пополам (программа 1.4)

## Упражнения

- ▷ **1.4** Приведите содержимое массива `id` после выполнения каждой операции *union* при использовании алгоритма быстрого поиска (программа 1.1) для решения задачи связности для последовательности 0-2, 1-4, 2-5, 3-6, 0-4 и 1-3. Укажите также количество обращений программы к массиву `id` для каждой вводимой пары.
- ▷ **1.5** Выполните упражнение 1.4, но используйте алгоритм быстрого объединения (программа 1.2).
- ▷ **1.6** Приведите содержимое массива `id` после выполнения каждой операции *union* для случаев использования алгоритма взвешенного быстрого объединения применительно к примерам, соответствующим рис. 1.7 и 1.8.
- ▷ **1.7** Выполните упражнение 1.4, но используйте алгоритм взвешенного быстрого объединения (программа 1.3).
- ▷ **1.8** Выполните упражнение 1.4, но используйте алгоритм взвешенного быстрого объединения с сжатием пути делением пополам (программа 1.4).
- 1.9** Определите верхнюю границу количества машинных инструкций, требующихся для обработки  $M$  соединений  $N$  объектов при использовании программы 1.3. Например, можно предположить, что для выполнения оператора присваивания C++ всегда требуется выполнение менее  $c$  инструкций, где  $c$  — некоторая фиксированная константа.
- 1.10** Определите минимальное время (в днях), которое потребовалось бы для выполнения быстрого поиска (программа 1.1) для решения задачи с  $10^9$  объектов и  $10^6$  вводимых пар на компьютере, который может выполнять  $10^9$  инструкций в секунду. Примите, что при каждой итерации внутреннего цикла `for` должно выполняться не менее 10 инструкций.
- 1.11** Определите максимальное время (в секундах), которое потребовалось бы для выполнения взвешенного быстрого объединения (программа 1.3) для решения задачи с  $10^9$  объектов и  $10^6$  вводимых пар на компьютере, который может выполнять  $10^9$  инструкций в секунду. Примите, что при каждой итерации внешнего цикла `while` должно выполняться не более 100 инструкций.
- 1.12** Вычислите *среднее* расстояние от узла до корня в худшем случае в дереве, построенном алгоритмом взвешенного быстрого объединения из  $2^n$  узлов.
- ▷ **1.13** Нарисуйте схему подобную рис. 1.10, но содержащую восемь узлов, а не девять.
- **1.14** Приведите последовательность вводимых пар, для которой алгоритм взвешенного быстрого объединения (программа 1.3) создает путь длиной 4.
- **1.15** Приведите последовательность вводимых пар, для которой алгоритм взвешенного быстрого объединения с сжатием пути делением пополам (программа 1.4) создает путь длиной 4.
- 1.16** Покажите, как необходимо изменить программу 1.3, чтобы реализовать *полное* сжатие пути, при котором каждая операция *union* завершается тем, что каждый обрабатываемый узел указывает на корень нового дерева.
- ▷ **1.17** Выполните упражнение 1.4, но используйте алгоритм взвешенного быстрого объединения с полным сжатием пути (упражнение 1.16).

- 1.18 Приведите последовательность вводимых пар, для которой алгоритм взвешенного быстрого объединения с полным сжатием пути (см. упражнение 1.16) создаст путь длиной 4.
- 1.19 Приведите пример, показывающий, что изменения быстрого объединения (программа 1.2) для реализации полного сжатия пути (см. упражнение 1.16) не достаточно, чтобы гарантировать отсутствие длинных путей в деревьях.
- 1.20 Измените программу 1.3, чтобы в ней для принятия решения, нужно ли устанавливать  $id[i] = j$  или  $id[j] = i$ , вместо веса использовалась *высота* деревьев (самый длинный путь от любого узла до корня). Экспериментально сравните этот вариант с программой 1.3.
- 1.21 Покажите, что лемма 1.3 справедлива для алгоритма, описанного в упражнении 1.20.
- 1.22 Измените программу 1.4, чтобы она генерировала случайные пары целых чисел в диапазоне от 0 до  $N - 1$  вместо того, чтобы считывать их из стандартного ввода, и выполняла цикл до тех пор, пока не будет выполнено  $N - 1$  операций *union*. Выполните программу для значений  $N = 10^3, 10^4, 10^5$  и  $10^6$  и выведите общее количество ребер, генерируемых для каждого значения  $N$ .
- 1.23 Измените программу из упражнения 1.22, чтобы она выводила в виде графика количество ребер, требующихся для соединения  $N$  элементов, при  $100 \leq N \leq 1000$ .
- 1.24 Приведите приближенную формулу для определения количества случайных ребер, требующихся для соединения  $N$  объектов, как функции  $N$ .

## 1.4 Перспектива

Каждый из алгоритмов, рассмотренных в разделе 1.3, кажется определенным усовершенствованием предыдущего, но, вероятно, процесс был искусственно упрощен, поскольку разработка самих этих алгоритмов уже была выполнена рядом исследователем в течение многих лет (см. *раздел используемой литературы*). Реализации просты, а задача четко определена, поэтому мы можем оценить различные алгоритмы непосредственно, экспериментальным путем. Более того, мы можем подкрепить эти исследования, количественно сравнив производительность алгоритмов (см. главу 2). Не все задачи, рассмотренные в этой книге, столь же хорошо проработаны, как эта, и мы обязательно встретимся с алгоритмами, которые трудно сравнить, и с математическими задачами, которые трудно решить. Мы стремимся принимать объективные научно обоснованные решения в отношении используемых алгоритмов, изучая свойства реализаций на примере их выполнения применительно к реальным данным, полученным из приложений, или случайным тестовым наборам данных.

Этот процесс — прототип того, как различные алгоритмы решения фундаментальных задач рассматриваются в данной книге. Когда это возможно, мы предпринимаем те же основные шаги, что и при рассмотрении алгоритмов объединения-поиска, описанных в разделе 1.2, часть из которых перечислена ниже:

- Принятие формулировки полной и частной задачи, включая определение основных абстрактных операций, присущих задаче.

- Тщательная разработка краткой реализации для простого алгоритма.
- Разработка усовершенствованных реализаций путем пошагового улучшения, проверки эффективности идей усовершенствований посредством эмпирического или математического анализа либо их обоих.
- Определение высокоуровневых абстрактных представлений структур данных или алгоритмов посредством операций, которые позволяют эффективно разрабатывать усовершенствованные версии на высоком уровне.
- Стремление к получению гарантированной производительности в худшем случае, когда это возможно, но принятие хорошей производительности при работе с реальными данными, когда это доступно.

Потенциальная возможность впечатляющего повышения производительности алгоритмов решения реальных задач, подобных рассмотренным в разделе 1.2, делает разработку алгоритмов увлекательной областью исследования; лишь немногие другие области разработки позволяют обеспечить экономию времени в миллионы, миллиарды и более раз.

Что еще важнее, по мере повышения вычислительных возможностей компьютеров и приложений разрыв между быстрыми и медленными алгоритмами увеличивается. Новый компьютер может работать в 10 раз быстрее и может обрабатывать в 10 раз больше данных, чем старый, но при использовании квадратичного алгоритма, наподобие быстрого поиска, новому компьютеру потребуется в 10 раз больше времени для выполнения новой задачи, чем требовалось старому для выполнения старой! Вначале это утверждение кажется противоречивым, но его легко подтвердить простым тождеством  $(10N)^2 / 10 = 10N^2$ , как будет показано в главе 2. По мере того как вычислительные мощности увеличиваются, позволяя решать все более сложные задачи, важность использования эффективных алгоритмов также возрастает.

Разработка эффективного алгоритма приносит моральное удовлетворение и прямо окупается на практике. Как видно на примере решения задачи связности, просто сформулированная задача может приводить к изучению множества алгоритмов, которые не только полезны и интересны, но и представляют увлекательную и сложную для понимания задачу. Мы встретимся со многими остроумными алгоритмами, которые были разработаны за долгие годы для решения множества практических проблем. По мере расширения области применения компьютерных решений для научных и экономических проблем возрастает важность использования эффективных алгоритмов для решения известных задач и разработки эффективных решений для новых проблем.

## Упражнения

**1.25** Предположите, что взвешенное быстрое объединение используется для обработки в 10 раз большего количества соединений на новом компьютере, который работает в 10 раз быстрее старого. На сколько больше времени потребуется для выполнения новой задачи на новом компьютере по сравнению с выполнением старой задачи на старом компьютере?

**1.26** Выполните упражнение 1.25 для случая использования алгоритма, для которого требуется выполнение  $N^3$  инструкций.

## 1.5 Обзор тем

В этом разделе приведены краткие описания основных частей книги и раскрытых в них отдельных тем с указанием общего подхода к материалу. Выбор тем диктовался стремлением осветить как можно больше базовых алгоритмов. Некоторые из освещенных тем относятся к фундаментальным темам компьютерных наук, которые мы подробно изучаем для изучения основных алгоритмов широкого применения. Другие рассмотренные алгоритмы относятся к специализированным областям компьютерных наук и связанным с ними областям, таким как численный анализ и исследование операций — в этих случаях приведенный материал служит введением в эти области через исследование базовых методов.

В первых четырех частях, включенных в этот том, освещен наиболее широко используемый набор алгоритмов и структур данных, первый уровень абстракции для коллекций объектов с ключами, который может поддерживать широкое множество важных основополагающих алгоритмов. Рассматриваемые алгоритмы — результаты десятков лет исследований и разработки, и они продолжают играть важную роль во все расширяющемся применении компьютеров.

**Основные принципы** (Часть 1) в контексте данной книги представляют собой основные принципы и методологию, используемые для реализации, анализа и сравнения алгоритмов. Материал, приведенный в главе 1, служит обоснованием изучения разработки и анализа алгоритмов; в главе 2 рассмотрены основные методы получения информации о количественных показателях производительности алгоритмов.

**Структуры данных** (Часть 2) тесно связаны с алгоритмами: необходимо получить ясное представление о методах представления данных, которые используются во всех остальных частях книги. Изложение материала начинается с введения в базовые структуры данных в главе 3, включая анализ, связанные списки и строки; затем в главе 5 рассмотрены рекурсивные программы и структуры данных, в частности, деревья и алгоритмы для манипулирования ими. В главе 4 рассмотрены основные абстрактные типы данных (abstract data types — ADT), такие как стеки и очереди, а также реализации с использованием элементарных структур данных.

Алгоритмы **сортировки** (Часть 3), предназначенные для упорядочения файлов имеют особую важность. Мы достаточно глубоко рассмотрим ряд базовых алгоритмов, в том числе быструю сортировку, сортировку слиянием и поразрядную сортировку. В ходе рассмотрения мы встретимся с несколькими связанными задачами, в том числе с очередями приоритетов, выбором и слиянием. Многие из этих алгоритмов послужат основанием для других алгоритмов, рассматриваемых в последующих частях книги.

Алгоритмы **поиска** (Часть 4), предназначенные для поиска конкретных элементов в больших коллекциях элементов, также имеют важное значение. Мы рассмотрим основные и расширенные методы поиска с использованием деревьев и преобразований численных ключей, в том числе деревья бинарного поиска, сбалансированные деревья, хеширование, деревья цифрового поиска и методы, которые подходят для очень больших файлов. Мы отметим взаимосвязь между этими методами, приведем статистические данные о их сравнительной производительности и установим соответствие с методами сортировки.



В частях с 5 по 8, которые вынесены в отдельный том, освещены дополнительные применения описанных алгоритмов для иного набора данных — второй уровень абстракций, характерный для ряда важных областей применения. Кроме того, в этих частях более глубоко рассмотрены технологии разработки и анализа алгоритмов. Многие из затрагиваемых проблем являются предметом предстоящих исследований.

**Алгоритмы обработки строк** (часть 5) включают в себя ряд методов обработки последовательностей символов (длинных). Поиск строк ведет к установлению соответствия с шаблоном, что, в свою очередь, ведет к синтаксическому анализу. В этой части также рассматриваются и технологии сжатия файлов. Опять-таки, введение в более сложные темы выполняется через рассмотрение некоторых простых задач, которые важны и сами по себе.

**Геометрические алгоритмы** (Часть 6) — это методы решения задач с использованием точек и линий (и других простых геометрических объектов), которые стали использоваться лишь недавно. Мы рассмотрим алгоритмы для отыскания образующей поверхности, определенной набором точек, определения пересечений геометрических объектов, решения задач отыскания ближайших точек и для выполнения многомерного поиска. Многие из этих методов прекрасно дополняют более элементарные методы сортировки и поиска.

**Алгоритмы обработки графов** (Часть 7) полезны при решении ряда сложных и важных задач. Общая стратегия поиска в графах разрабатывается и применяется к фундаментальным задачам связности, в том числе к задаче отыскания кратчайшего пути, минимального остовного дерева, к задаче о сетевом потоке и к задаче соответствия. Унифицированный подход к этим алгоритмам показывает, что все они основываются на одной и той же процедуре, и что эта процедура основывается на основном абстрактном типе данных очереди приоритетов.

**Дополнительные темы** (Часть 8) устанавливают соответствие между изложенным в книге материалом и несколькими связанными областями. Изложение материала начинается с рассмотрения базовых подходов к разработке и анализу алгоритмов, в том числе алгоритмов типа "разделяй и властвуй", динамического программирования, рандомизации и амортизации. Мы рассмотрим линейное программирование, быстрое преобразование Фурье, NP-полноту и другие дополнительные темы для получения общего представления о ряде областей, интерес к которым порождается элементарными проблемами, рассмотренными в этой книге.

Изучение алгоритмов представляет интерес, поскольку это новая отрасль (почти все изученные в этой книге алгоритмы не старше 50 лет, а некоторые были открыты лишь недавно) с богатыми традициями (некоторые алгоритмы известны уже в течение тысяч лет). Постоянно делаются новые открытия, но лишь немногие алгоритмы исследованы полностью. В этой книге мы рассмотрим замысловатые, сложные и трудные алгоритмы, наряду с изящными, простыми и легко реализуемыми алгоритмами. Наша задача — понять первые и оценить вторые в контексте множества различных потенциально возможных приложений. В процессе этого нам предстоит исследовать ряд полезных инструментальных средств и выработать стиль алгоритмического мышления, который пригодится при решении предстоящих вычислительных задач.

## Принципы анализа алгоритмов

**А**нализ — это ключ к пониманию алгоритмов в степени, достаточной для их эффективного применения при решении практических задач. Хотя у нас нет возможности проводить исчерпывающие эксперименты и глубокий математический анализ каждой программы, мы будем работать на основе и эмпирического тестирования, и приближенного анализа, которые помогут нам изучить основные характеристики производительности наших алгоритмов, чтобы можно было сравнивать различные алгоритмы и применять их для практических целей.

Сама идея точного описания производительности сложного алгоритма с помощью математического анализа кажется на первый взгляд устрашающей перспективой, поэтому мы часто обращаемся к исследовательской литературе за результатами подробного математического изучения. Хотя целью данной книги не является обзор методов анализа или их результатов, для нас важно знать, что мы находимся на твердой теоретической почве при сравнении различных методов. Более того, посредством тщательного применения относительно простых методов о многих из алгоритмов можно получить множество подробной информации. Во всей книге мы отводим главное место базовым аналитическим результатам и методам анализа, особенно тогда, когда это может помочь нам в понимании внутренней работы фундаментальных алгоритмов. В этой главе наша основная цель — обеспечить среду и инструменты, необходимые для работы с алгоритмами.

Пример в главе 1 показывает среду, иллюстрирующую многие из базовых концепций анализа алгоритмов, поэтому мы будем часто ссылаться на производительность

алгоритмов `union-find` для конкретизации определенных моментов. Несколько новых примеров подробно обсуждаются в разделе 2.6.

Анализ играет определенную роль в каждой точке процесса разработки и реализации алгоритмов. Прежде всего, как было показано, можно сократить время выполнения на три-шесть порядков за счет правильного выбора алгоритма. Чем эффективнее будут рассматриваемые алгоритмы, тем сложнее будет становиться задача выбора между ними, поэтому необходимо подробнее изучить их свойства. В погоне за *наилучшим* (в некотором точном техническом смысле) алгоритмом мы будем находить и алгоритмы, полезные практически, и теоретические вопросы, требующие решения.

Полный охват методов анализа алгоритмов сам по себе является предметом книги (см. раздел ссылок), однако здесь рассматриваются основы, которые позволят

- Проиллюстрировать процесс.
- Описать в одном месте используемые математические соглашения.
- Обеспечить основу для обсуждения вопросов высокого уровня.
- Выработать понимание научной основы выводов, которые делаются при анализе алгоритмов.

Главное, алгоритмы и их анализ зачастую переплетены. В этой книге мы не станем тщательно изучать глубокие и сложные математические источники, но сделаем достаточно выкладок, чтобы понять, что представляют собой наши алгоритмы и как их можно использовать наиболее эффективно.

## 2.1. Разработка и эмпирический анализ

Мы разрабатываем и реализуем алгоритмы, создавая иерархию абстрактных операций, которые помогают понять природу решаемых вычислительных проблем. При теоретическом изучении данный процесс, хотя он достаточно важен, может увести очень далеко от проблем реального мира. Поэтому в данной книге мы стоим на твердой почве, выражая все рассматриваемые нами алгоритмы реальным языком программирования — C++. Такой подход иногда оставляет очень туманное различие между алгоритмом и его реализацией, но это лишь небольшая плата за возможность работать с конкретной реализацией и учиться на ней.

Несомненно, правильно разработанные программы на реальном языке представляют собой эффективные методы выражения алгоритмов. В этой книге мы изучаем большое количество важных и эффективных алгоритмов, кратко и точно реализованных на C++. Описания на обычном языке или абстрактные высокоуровневые представления зачастую неопределённые и незакончены, а реальные реализации заставляют нас находить экономные представления, не позволяющие завязнуть в деталях.

Мы выражаем алгоритмы на C++, но эта книга об алгоритмах, а не о программировании на C++. Конечно же, мы рассматриваем реализации на C++ многих важных задач, и когда существует удобный и эффективный способ решить задачу именно средствами C++, мы пользуемся этим достоинством. Тем не менее, подавляющее большинство выводов о реализации алгоритмов применимы к любой современной среде программирования. Перевод программ из главы 1 или любых других программ в этой книге на другой современный язык программирования — это достаточно про-

стая задача. Если в некоторых случаях какой-либо другой язык обеспечивает более эффективный механизм решения определенных задач, мы будем указывать на это. Наша цель — использовать C++ как средство выражения алгоритмов, а не задерживаться на вопросах, специфичных для языка.

Если алгоритм реализуется как часть большой системы, мы используем абстрактные типы данных или похожий механизм, чтобы иметь возможность изменить алгоритмы или детали реализации после того, как станет ясно, какая часть системы заслуживает наиболее пристального внимания. Однако в самом начале потребуется понимание характеристик производительности каждого алгоритма, так как требования системы к проектированию могут в значительной степени повлиять на производительность алгоритма. Подобные исходные решения при разработке необходимо принимать с осторожностью, поскольку в конце часто оказывается, что производительность целой системы зависит от производительности некоторых базовых алгоритмов наподобие тех, которые обсуждаются в этой книге.

Алгоритмы, приведенные в этой книге, нашли эффективное применение во всем многообразии больших программ, операционных систем и различных приложениях. Наше намерение состоит в том, чтобы описать алгоритмы и уделить внимание их динамическим свойствам экспериментальным путем. Для одних приложений приведенные реализации могут подойти точно, для других может потребоваться некоторая доработка. Например, при создании реальных систем требуется более защищенный стиль программирования, нежели используемый в книге. Необходимо следить за состоянием ошибок и сообщать о нем, а, кроме того, программы должны быть разработаны таким образом, чтобы их изменение было несложным делом, чтобы их могли быстро читать и понимать другие программисты, чтобы они хорошо взаимодействовали с различными частями системы и сохранялась возможность их переноса в другие среды.

Не отвергая все эти комментарии, все же при анализе каждого алгоритма мы уделяем внимание, в основном, существенным характеристикам производительности алгоритма. Предполагается, что главный интерес будут вызывать наиболее простые алгоритмы с наилучшими показателями производительности.

Для эффективного использования алгоритмов, если нашей целью является решение огромной задачи, которую нельзя решить другим способом, или если цель заключается в эффективной реализации важной части системы, нам требуется понимание характеристик производительности алгоритмов. Формирование такого понимания и является целью анализа алгоритмов.

Один из первых шагов для продвижения вперед в понимании производительности алгоритмов — это *эмпирический анализ*. Если есть два алгоритма для решения одной задачи, то в методе нет никакого волшебства: мы запустим оба и определим, который выполняется дольше! Это концепция может показаться слишком очевидной, чтобы о ней стоило говорить, но, тем не менее, это распространенное упущение в сравнительном анализе алгоритмов. Тот факт, что один алгоритм в 10 раз быстрее другого, вряд ли ускользнет от взора того, кто ждет 3 секунды при выполнении одного и 30 секунд — при выполнении другого, однако его легко упустить как небольшой постоянный фактор при математическом анализе. Когда мы отслеживаем про-

изводительность точных реализаций алгоритмов при типовом вводе, мы получаем результаты, которые не только являются прямым индикатором эффективности, но и содержат информацию, необходимую для сравнения алгоритмов и обоснования предлагаемых математических результатов (см., например, табл. 1.1). Когда эмпирическое изучение начинает поглощать значительное количество времени, на помощь приходит математический анализ. Ожидание завершения программы в течение часа или целого дня вряд ли может рассматриваться как продуктивный способ для понимания того, что программа медлительна, особенно в тех случаях, когда прямой анализ может привести к аналогичному результату.

Первая проблема, возникающая перед нами в эмпирическом анализе — это разработка корректной и полной реализации. Для некоторых сложных алгоритмов эта проблема может оказаться сложным препятствием. В соответствии с этим, обычно требуется либо с помощью анализа, либо путем экспериментирования с похожими программами, установить некоторый признак, насколько эффективной может быть программа, до того как тратить усилия на ее реализацию.

Вторая проблема, с которой мы сталкиваемся при эмпирическом анализе, — это определение природы входных данных и других факторов, оказывающих прямое влияние на производимые эксперименты. Обычно существуют три основных возможности: *реальные* данные, *случайные* данные и *ошибочные* данные. Реальные данные позволяют точно измерить параметры используемой программы; случайные данные гарантируют, что наши эксперименты тестируют алгоритм, а не данные; ошибочные данные гарантируют, что наши программы могут воспринимать любой направленный им ввод. Например, при тестировании алгоритмов сортировки мы запускаем их для работы с данными в виде слов в произведении "Моби Дик", в виде сгенерированных случайным образом целых чисел и в виде файлов, содержащих одинаковые числа. Проблема определения, какие данные необходимо использовать для сравнения алгоритмов, возникает также и при анализе алгоритмов.

При сравнении различных реализаций легко допустить ошибки, особенно, если в игру вступают разные машины, компиляторы или системы, либо гигантские программы с плохо охарактеризованным вводом. Принципиальная опасность при эмпирическом сравнении программ таится в том, что код для одной реализации может быть создан более тщательно, чем для другой. Изобретателю предлагаемого нового алгоритма необходимо обратить внимание на все аспекты его реализации, чтобы не расходовать слишком много усилий на создание классического конкурирующего алгоритма. Чтобы быть уверенным в точности эмпирического сравнения алгоритмов, мы должны быть уверены, что каждой реализации уделялось одинаковое внимание.

Один из подходов, который часто применяется в этой книге, как видно из главы 1, заключается в построении алгоритмов за счет внесения небольших изменений в алгоритмы, существующие для данной задачи, поэтому сравнительное изучение просто необходимо. В более общем смысле, мы стараемся определить важнейшие абстрактные операции и начинаем сравнивать алгоритмы на основе того, как они используют такие операции. Например, эмпирические результаты, приведенные в табл. 1.1, позволяют сравнивать языки программирования и среды, поскольку в них используются одинаковые программы, оперирующие одним и тем же набором базовых опе-

раций. Для реальной среды программирования эти числа можно превратить в реальные времена выполнения. Чаще всего просто требуется узнать, какая из двух программ будет быстрее или до какой степени определенное изменение может улучшить временные либо пространственные характеристики программы.

Возможно, наиболее распространенной ошибкой при выборе алгоритма является упущение характеристик производительности. Более скоростные алгоритмы, как правило, сложнее, чем прямые решения, и разработчики часто предпочитают более медленные алгоритмы, дабы избежать дополнительных сложностей. Однако, как это было для алгоритмов *union-find*, можно добиться значительных улучшений с помощью даже нескольких строк кода. Пользователи удивительно большого числа компьютерных систем теряют существенное время, ожидая, пока простые квадратичные алгоритмы решат задачу, в то время как доступные  $N \log N$  или линейные алгоритмы не намного сложнее, но могут решить задачу быстрее. Когда мы имеем дело с большими задачами, у нас нет другого выбора, кроме как искать наилучший алгоритм, что и будет показано далее.

Возможно, вторая наиболее распространенная ошибка при выборе алгоритма, — уделять слишком много внимания характеристикам его производительности. Снижение времени выполнения программы в 10 раз несущественно, если ее выполнение занимает несколько микросекунд. Даже, если программа требует нескольких минут, время и усилия, необходимые, чтобы она стала выполняться в 10 раз быстрее, могут не стоить того, в особенности, если программа должна применяться лишь несколько раз. Суммарное время, требуемое для реализации и отладки улучшенного алгоритма, может быть значительно выше, чем время, необходимое для выполнения более медленной программы, — в конце концов, часть работы вполне можно доверить компьютеру. Хуже того, можно потратить значительное количество времени и усилий, применяя идеи, которые должны были бы улучшить программу, но, фактически, не делают этого.

Мы не можем провести эмпирические тесты для программы, которая еще не написана, но можем проанализировать свойства программы и оценить потенциальную эффективность предлагаемого улучшения. Не все предполагаемые усовершенствования дают реальный выигрыш в производительности, поэтому нужно понять степень улучшений на каждом шаге. Более того, в реализации алгоритма могут быть включены параметры, а анализ поможет нам их правильно установить. Наиболее важно то, что понимая фундаментальные свойства программ и базовую природу использования ими ресурсов, мы имеем потенциальную возможность рассчитать их эффективность на компьютерах, которые еще не созданы, или сравнить их с новыми алгоритмами, которые еще не написаны. В разделе 2.2 очерчивается методология достижения базового понимания производительности алгоритмов.

## Упражнения

2.1 Перевести программу в главе 1 на другой язык программирования и ответить на вопросы упражнения 1.22 для вашей реализации.

**2.2** Сколько времени займет посчитать до 1 миллиарда (не учитывая переполнение)? Определить количество времени, необходимое программе

```
int i, j, k, count = 0;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            count++;
```

для выполнения в вашей среде для  $N = 10, 100$  и  $1000$ . Если ваш компилятор имеет свойства по оптимизации, которые должны делать программы более эффективными, проверьте, дают ли они какой-либо результат для этой программы.

## 2.2 Анализ алгоритмов

В этом разделе мы наметим, в каких рамках математический анализ будет играть роль в процессе сравнения производительности алгоритмов, и зложим фундамент, необходимый для приложения основных математических результатов к фундаментальным алгоритмам, которые будут изучаться в течение всей книги. Мы рассмотрим основные математические инструменты, используемые для анализа алгоритмов, чтобы изучать классические примеры анализа фундаментальных алгоритмов, а также чтобы воспользоваться результатами исследовательской литературы, которые помогут в понимании характеристик производительности наших алгоритмов.

Среди причин, по которым выполняется математический анализ алгоритмов, находятся следующие:

- Для сравнения разных алгоритмов, предназначенных для решения одной задачи
- Для приблизительной оценки производительности программы в новой среде
- Для установки значений параметров алгоритма

В течение книги можно будет найти немало примеров каждой из этих причин. Эмпирический анализ подходит для некоторых задач, но математический анализ, как будет показано, может оказаться более информативным (и менее дорогим!).

Анализ алгоритмов может стать довольно-таки интересным делом. Некоторые из алгоритмов, приведенных в этой книге, понятны до той степени, когда известны точные математические формулы для расчета времени выполнения в реальных ситуациях. Эти формулы разработаны путем тщательного изучения программы для нахождения времени выполнения в терминах фундаментальных математических величин и последующего их математического анализа. С другой стороны, свойства производительности других алгоритмов из этой книги не поняты до конца — возможно, потому, что их анализ ведет к нерешенным математическим проблемам или же известные реализации слишком сложны, чтобы их подробный анализ имел смысл, или (скорее всего) типы данных для ввода не поддаются точной характеристике.

Несколько важных факторов точного анализа обычно находятся за пределами области действия программиста. Во-первых, программы C++ переводятся в машинные коды для данного типа компьютера, и может оказаться достаточно сложной задачей определить, сколько времени займет выполнение даже одного оператора C++ (особенно в среде, где возможен совместный доступ к ресурсам так, что одна и та же

программа может иметь различные характеристики производительности в разное время). Во-вторых, многие программы чрезвычайно чувствительны ко входным данным, поэтому производительность может меняться в больших пределах в зависимости от них. В-третьих, многие интересующие нас программы еще не поняты до конца, поэтому для них пока не существует специальных математических результатов. И, в заключение, две программы могут совершенно не поддаваться сравнению: одна работает эффективно с определенным типом входных данных, другая выполняется эффективно при других обстоятельствах.

Несмотря на все эти факторы, часто вполне возможно предсказать, сколько времени займет выполнение определенной программы или же понять, что одна программа будет выполняться эффективнее другой, но в определенных ситуациях. Более того, часто этих знаний можно достичь, используя относительно небольшой набор математических инструментов. Задачей аналитика алгоритмов является получить как можно больше информации о производительности алгоритмов; задачей программиста — использовать эту информацию при выборе алгоритмов для определенных приложений. В этой и нескольких следующих разделах будет уделяться внимание идеализированному миру аналитика. Чтобы эффективно применять лучшие алгоритмы, иногда просто необходимо посещать такой мир.

Первый шаг при анализе алгоритма состоит в определении абстрактных операций, на которых основан алгоритм, чтобы отделить анализ от реализации. Так, например, мы отделяем изучение того, сколько раз одна из реализаций алгоритма *union-find* запускает фрагмент кода  $i = a[i]$ , от подсчета, сколько наносекунд требуется для выполнения этого фрагмента кода на данном компьютере. Для определения реального времени выполнения программы на заданном типе компьютера требуются оба упомянутых элемента. Первый из них определяется свойствами алгоритма, последний — свойствами компьютера. Такое разделение зачастую позволяет сравнивать алгоритмы таким способом, который не зависит от определенной реализации или от определенного типа компьютера.

Хотя количество используемых абстрактных операций может оказаться очень большим, в принципе, производительность алгоритма обычно зависит от нескольких величин, причем наиболее важные для анализа величины, как правило, определить несложно. Один из способов их определения заключается в использовании механизма профилирования (механизма, доступного во многих реализациях C++, который включает в себя счетчик количества выполнений каждой инструкции) для нахождения наиболее часто исполняемых частей программы по результатам нескольких пробных запусков. Или же, как алгоритмы *union-find* из раздела 1.3, наша реализация может быть построена лишь на нескольких абстрактных операциях. В любом случае, анализ сводится к определению частоты исполнения нескольких фундаментальных операций. Наш образ действия заключается в том, чтобы отыскать приблизительные оценки этих величин, будучи уверенными, что для важных программ при необходимости можно будет произвести полный анализ. Более того, как будет показано далее, часто можно воспользоваться приближенными аналитическими результатами в сочетании с эмпирическим изучением, чтобы предсказать результаты достаточно точно.



Кроме того, необходимо изучать данные и моделировать их ввод, который ожидает алгоритм. Чаще всего мы будем рассматривать один из двух подходов к анализу: или мы будем предполагать, что ввод является случайным, и изучать *среднюю* производительность программы, или же мы будем рассматривать произвольный ввод и изучать *низкую* производительность программы. Процесс описания случайного ввода для многих алгоритмов достаточно сложен, но для других алгоритмов он может быть прямолинейным и вести к аналитическим результатам, дающим полезную информацию. Средней может быть математическая функция, не зависящая от данных, для которых используется программа, а наихудшей — странная конструкция, которая никогда не встречается на практике, но в большинстве случаев эти виды анализа предоставляют полезную информацию о производительности. Например, мы можем сравнить аналитические и эмпирические результаты (см. раздел 2.1). Если они совпадут, мы повысим нашу уверенность в обоих вариантах; если они не совпадут, мы сможем узнать больше об алгоритме и модели, изучив несоответствия.

В следующих трех разделах мы кратко рассмотрим математические инструменты, которыми будем пользоваться в течение всей книги. Этот материал находится за пределами нашей первоначальной узкой задачи, поэтому читатели, хорошо знакомые с математической основой, или же те, кто не собирается подробно проверять наши математические выражения, связанные с производительностью алгоритмов, могут перейти сразу к разделу 2.6 и вернуться к этому материалу там, где это указано в книге позже. Математические основы, в общем-то, не сложны для понимания и настолько близки к коренным вопросам разработки алгоритма, что их не стоит игнорировать тем, кто стремится эффективно использовать компьютер.

Вначале, в разделе 2.3, рассматриваются математические функции, которые потребуются для описания характеристик производительности алгоритмов. Затем, в разделе 2.4, рассматривается *O-нотация* (*O-notation*) и выражение *пропорционально* (*is proportional to*), которое позволяет опустить детали при математическом анализе. Далее, в разделе 2.5, изучается понятие *рекуррентных соотношений* (*recurrence relations*), основного аналитического инструмента, используемого для отражения характеристик производительности алгоритма в математических выражениях. Следуя этому обзору, в разделе 2.6 приводятся примеры, в которых все эти инструменты применяются для анализа некоторых алгоритмов.

## Упражнения

- **2.3** Найти выражение вида  $c_0 + c_1N + c_2N^2 + c_3N^3$ , которое точно описывает время выполнения программы из упражнения 2.2. Сравнить время, задаваемое этим выражением, с реальным при  $N = 10, 100$  и  $1000$ .
- **2.4** Найти выражение, которое точно описывает время выполнения программы 1.1 в величинах  $M$  и  $N$ .

## 2.3 Рост функций

Большинство алгоритмов имеют *главный параметр*  $N$ , который значительно влияет на время их выполнения. Параметр  $N$  может быть степенью полинома, размером файла при сортировке или поиске, количеством символов в строке или некоторой

другой абстрактной мерой размера рассматриваемой задачи: чаще всего, он прямо пропорционален величине обрабатываемого набора данных. Когда таких параметров существует более одного (например,  $M$  и  $N$  в алгоритмах *union-find*, которые обсуждались в разделе 1.3), мы часто сводим анализ к одному параметру, задавая его как функцию других, или рассматривая одновременно только один параметр (считая остальные постоянными), и, таким образом, ограничивая себя рассмотрением только одного параметра  $N$  без потерь общности. Нашей целью является выражение ресурсных требований программ (как правило, времени выполнения) в зависимости от  $N$  с использованием математических формул, которые максимально просты и справедливы для больших значений параметров. Алгоритмы в этой книге обычно имеют время выполнения, пропорциональное одной из следующих функций:

- 1** Большинство инструкций большинства программ запускается один или несколько раз. Если все инструкции программы обладают таким свойством, мы говорим, что время выполнения программы *постоянно*.
- $\log N$**  Когда время выполнения программы является *логарифмическим*, программа становится медленнее с ростом  $N$ . Такое время выполнения обычно присуще программам, которые сводят большую задачу к набору меньших задач, уменьшая на каждом шаге размер задачи на некоторый постоянный фактор. В интересующей нас области мы будем рассматривать время выполнения, являющееся небольшой константой. Основание логарифма изменяет константу, но не намного: когда  $N$  — тысяча,  $\log N$  равно 3, если основание равно 10, или порядка 10, если основание равно 2; когда  $N$  равно миллиону, значения  $\log N$  только удвоятся. При удвоении  $\log N$  растет на постоянную величину, а удваивается лишь тогда, когда  $N$  достигает  $N^2$ .
- $N$**  Когда время выполнения программы является *линейным*, это обычно значит, что каждый входной элемент подвергается небольшой обработке. Когда  $N$  равно миллиону, такого же и время выполнения. Когда  $N$  удваивается, то же происходит и со временем выполнения. Эта ситуация оптимальна для алгоритма, который должен обработать  $N$  вводов (или произвести  $N$  выводов).
- $N \log N$**  Время выполнения, пропорциональное  $N \log N$ , возникает тогда, когда алгоритм решает задачу, разбивая ее на меньшие подзадачи, решая их независимо и затем объединяя решения. Из-за отсутствия подходящего прилагательного ("*линерифмический*") мы просто говорим, что время выполнения такого алгоритма равно  $N \log N$ . Когда  $N$  равно 1 миллион,  $N \log N$  около 20 миллионов. Когда  $N$  удваивается, тогда время выполнения более чем удваивается.
- $N^2$**  Когда время выполнения алгоритма является *квадратичным*, он полезен для практического использования для относительно небольших задач. Квадратичное время выполнения обычно появляется в алгоритмах, которые обрабатывают все пары элементов данных (возможно, в цикле двойного уровня вложенности). Когда  $N$  равно 1 тысяче, время выполнения равно 1 миллиону. Когда  $N$  удваивается, время выполнения увеличивается вчетверо.

- $N^3$  Похожий алгоритм, который обрабатывает тройки элементов данных (возможно, в цикле тройного уровня вложенности), имеет *кубическое* время выполнения и практически применим лишь для малых задач. Когда  $N$  равно 100, время выполнения равно 1 миллиону. Когда  $N$  удваивается, время выполнения увеличивается в восемь раз.
- $2^N$  Лишь несколько алгоритмов с *экспоненциальным* временем выполнения имеет практическое применение, хотя такие алгоритмы возникают естественным образом при попытках прямого решения задачи. Когда  $N$  равно 20, время выполнения равно 1 миллиону. Когда  $N$  удваивается, время выполнения учетверяется!

Время выполнения определенной программы, скорее всего, будет некоторой константой, умноженной на один из этих элементов (*главный член*) плюс меньшие слагаемые. Значения постоянного коэффициента и остальных слагаемых зависят от результатов анализа и деталей реализации. В грубом приближении коэффициент при главном члене связан с количеством инструкций во внутреннем цикле: на любом уровне разработки алгоритма разумно сократить количество таких инструкций. Для больших  $N$  доминирует эффект главного члена, для малых  $N$  или для тщательно разработанных алгоритмов вклад дают и другие слагаемые, поэтому сравнение алгоритмов становится более сложным. В большинстве случаев мы будем называть время выполнения программ просто "линейным", " $N \log N$ ", "кубическим" и т.д. Обоснование этого подробно приводится в разделе 2.4.

В итоге, чтобы уменьшить общее время выполнения программы, мы минимизируем количество инструкций во внутреннем цикле. Каждую инструкцию необходимо подвергнуть исследованию: является ли она необходимой? Существует ли более эффективный способ выполнить такую же задачу? Некоторые программисты считают, что автоматические инструменты, содержащиеся в современных компиляторах, могут создавать наилучший машинный код; другие утверждают, что наилучшим способом является написание внутренних циклов вручную на машинном языке, или ассемблере. Обычно мы будем останавливаться, доходя до рассмотрения оптимизации на таком уровне, хотя иногда будем указывать, сколько машинных инструкций требуется для выполнения определенных операций. Это потребуется для того, чтобы понять, почему на практике одни алгоритмы могут оказаться быстрее других.

Для малых задач в том, каким методом мы воспользуемся, практически нет различий — быстрый современный компьютер все равно выполнит задачу мгновенно. Но по мере роста задачи, числа, с которыми мы имеем дело, становятся огромными, как продемонстрировано в табл. 2.1. Когда количество исполняемых инструкций в медленном алгоритме становится по-настоящему большим, время, необходимое для их выполнения, становится недостижимым даже для самых быстрых компьютеров. На рис. 2.1 приведен перевод большого количества секунд в дни, месяцы, годы и т.д.; в табл. 2.2 показаны примеры того, как быстрые алгоритмы имеют больше возможностей помочь решить задачу, не вовлекая огромные времена выполнения, нежели даже самые быстрые компьютеры.

**РИСУНОК 2.1 ПЕРЕВОД СЕКУНД**

Огромная разница между такими числами, как  $10^4$  и  $10^8$  становится более очевидной, когда мы рассмотрим их как количество секунд и переведем в привычные единицы измерения. Мы можем позволить программе выполняться 2.8 часа, но вряд ли мы сможем созерцать программу, выполнение которой займет 3.1 года. Поскольку  $2^{10}$  примерно равно  $10^3$ , этой таблицей можно воспользоваться и для перевода степеней 2. Например,  $2^{32}$  секунд это примерно 124 года.

секунды

$10^2$	1.7 минуты
$10^4$	2.8 часа
$10^5$	1.1 дня
$10^6$	1.6 недели
$10^7$	3.8 месяца
$10^8$	3.1 года
$10^9$	3.1 десятилетия
$10^{10}$	3.1 столетия
$10^{11}$	НИКОГДА

**Таблица 2.1 Значения часто встречающихся функций**

В этой таблице показаны относительные величины некоторых функций, которые будут часто встречаться нам при анализе алгоритмов. Очевидно, доминирующей является квадратичная функция, особенно для больших значений  $N$ , а значения между меньшими функциями оказываются не такими, как можно было ожидать для малых  $N$ . Например,  $N^{3/2}$  больше, чем  $N \lg^2 N$  для огромных значений  $N$ , однако для небольших  $N$  наблюдается обратная ситуация. Точное время выполнения алгоритма должно включать в себя линейную комбинацию этих функций. Мы можем легко отделить быстрые алгоритмы от медленных из-за огромной разницы между, например,  $\lg N$  и  $N$  или  $N$  и  $N^2$ , но различие между двумя быстрыми алгоритмами может потребовать тщательного изучения.

$\lg N$	$\sqrt{N}$	$N$	$N \lg N$	$N(\lg N)^2$	$N^{3/2}$	$N^2$
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10000
10	32	1000	9966	99317	31623	1000000
13	100	10000	132877	1765633	1000000	100000000
17	316	100000	1660964	27588016	31622777	10000000000
20	1000	1000000	19931569	397267426	1000000000	1000000000000

**Таблица 2.2 Время для решения гигантских задач**

Для многих приложений нашим единственным шансом решить гигантскую задачу является использование эффективного алгоритма. В этой таблице показано минимальное количество времени, необходимое для решения задач размером 1 миллион и 1 миллиард с использованием линейных,  $N \lg N$  и квадратичных алгоритмов на компьютерах с быстродействием 1 миллион, 1 миллиард и 1 триллион инструкций в секунду. Быстрый алгоритм позволяет решить задачу на медленной машине, но быстрая машина не может помочь, когда используется медленный алгоритм.

Операций в секунду	Размер задачи 1 миллион			Размер задачи 1 миллиард		
	$N$	$N \lg N$	$N^2$	$N$	$N \lg N$	$N^2$
$10^6$	секунд	секунд	недель	часов	часов	никогда
$10^9$	мгновенно	мгновенно	часов	секунд	секунд	десятилетий
$10^{12}$	мгновенно	мгновенно	секунд	мгновенно	мгновенно	недель

При анализе алгоритмов возникает еще несколько функций. Например, алгоритм с  $N^2$  вводами, имеющий время выполнения  $N^3$  можно рассматривать, как  $N^{3/2}$  алгоритм. Кроме того, некоторые алгоритмы, разбиваемые на две подзадачи, имеют время выполнения, пропорциональное  $N \log^2 N$ . Из табл. 2.1 очевидно, что обе эти функции ближе к  $N \log N$ , чем  $N^2$ .

Логарифмическая функция играет специальную роль в разработке и анализе алгоритмов, поэтому ее стоит рассмотреть подробнее. Поскольку мы часто имеем дело с аналитическими результатами, в которых опущен постоянный множитель, мы используем запись " $\log N$ ", опуская основание. Изменение основания логарифма меняет значение логарифма лишь на постоянный множитель, однако, в определенном контексте возникают специальные значения основания логарифма. В математике настолько важным является *натуральный логарифм* (основание  $e = 2.71828\dots$ ), что распространено следующее сокращение:  $\log_e N \equiv \ln N$ . В вычислительной технике очень важен *двоичный логарифм* (основание равно 2), поэтому используется сокращение  $\log_2 N \equiv \lg N$ .

Наименьшее целое число, большее  $\lg N$ , равно количеству бит, необходимых для представления  $N$  в двоичном формате; точно так же наименьшее целое, большее  $\log_{10} N$ , — это количество цифр, необходимое для представления  $N$  в десятичном формате. Оператор C++

```
for (lgN = 0; N > 0; lgN++, N /= 2) ;
```

дает простой способ подсчета наименьшего целого, большего  $\lg N$ . Похожий метод для вычисления этой функции

```
for (lgN = 0, t = 1; t < N; lgN++, t += t) ;
```

В нем утверждается, что  $2^n \leq N < 2^{n+1}$ , когда  $n$  — это наименьшее целое, большее  $\lg N$ .

Иногда мы итерируем логарифм: мы делаем это для больших чисел. Например,  $\lg \lg 2^{256} = \lg 256 = 8$ . Как иллюстрирует данный пример, обычно мы можем считать выражение  $\log \log N$  константой для практических целей, поскольку оно мало даже для очень больших  $N$ .

Кроме того, часто мы сталкиваемся с некоторыми специальными функциями и математической записью из классического анализа, которые полезны для краткого описания свойств программ. В табл. 2.3 собраны наиболее используемые из этих функций; в следующих параграфах мы кратко обсудим их, а также некоторые наиболее важные свойства.

Наши алгоритмы и их анализ часто сталкиваются с дискретными единицами, поэтому часто требуются специальные функции, переводящие действительные числа в целые:

$\lfloor x \rfloor$ : наибольшее целое, меньшее или равное  $x$

$\lceil x \rceil$ : наименьшее целое, большее или равное  $x$ .

Например,  $\lfloor \pi \rfloor$  и  $\lceil e \rceil$  оба равны 3, а  $\lceil \lg(N+1) \rceil$  — это количество бит, необходимое для двоичного представления числа  $N$ . Другое важное применение этих функций возникает в том случае, когда необходимо поделить набор из  $N$  объектов надвое. Этого

нельзя сделать точно, если  $N$  является нечетным, поэтому для точности мы можем создать один поднабор, содержащий  $\lfloor N/2 \rfloor$  объектов, а второй, —  $\lceil N/2 \rceil$  объектов. Если  $N$  четно, тогда размеры обоих поднаборов равны ( $\lfloor N/2 \rfloor = \lceil N/2 \rceil$ ); если же  $N$  нечетно, тогда их размер отличается на единицу ( $\lfloor N/2 \rfloor + 1 = \lceil N/2 \rceil$ ). В C++ можно напрямую подсчитать значения этих функций при выполнении операций над целыми числами (например, если  $N \geq 0$ , тогда  $N/2$  равно  $\lfloor N/2 \rfloor$ , а  $N - (N/2)$  равно  $\lceil N/2 \rceil$ ), а при операциях над числами с плавающей точкой можно воспользоваться функциями `floor` и `ceil` из заголовочного файла `math.h`.

При анализе алгоритмов часто возникает дискретизованная версия функции натурального логарифма, называемая *гармоническими числами*.  $N$ -тое гармоническое число определяется выражением

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N}.$$

Натуральный логарифм  $\ln N$  — это значение площади под кривой между 1 и  $N$ ; гармоническое число  $H_N$  — это площадь под ступенчатой функцией, которую можно определить, вычисляя значения функции  $1/x$  для целых чисел от 1 до  $N$ . Зависимость показана на рис. 2.2. Формула

$$H_N \approx \ln N + \gamma + 1 / (12N)$$

где  $\gamma = 0.57721\dots$  (эта константа называется *постоянной Эйлера*), дает отличное приближение для  $H_N$ . В отличие от  $\lceil \lg N \rceil$  и  $\lfloor \lg N \rfloor$ , для вычисления  $H_N$  лучше воспользоваться библиотечной функцией `log`, а не подсчитывать его непосредственно из определения.

### Таблица 2.3 Специальные функции и постоянные

В этой таблице собрана математическая запись для функций и постоянных, которые часто появляются в формулах, описывающих производительность алгоритмов. Формулы для приближенных значений можно при необходимости уточнить путем учета следующих слагаемых разложения (см. раздел [ссылок](#)).

Функция	Название	Типовое значение	Приближение
$\lfloor x \rfloor$	функция округления снизу	$\lfloor 3.14 \rfloor = 3$	$x$
$\lceil x \rceil$	функция округления сверху	$\lceil 3.14 \rceil = 4$	$x$
$\lg N$	двоичный логарифм	$\lg 1024 = 10$	$1.44 \ln N$
$F_N$	числа Фибоначчи	$F_{10} = 55$	$\phi^N / \sqrt{5}$
$H_N$	гармонические числа	$H_{10} \approx 2.9$	$\ln N + \gamma$
$N!$	факториал	$10! = 3628800$	$(N/e)^N$
$\lg(N!)$		$\lg(100!) \approx 520$	$N \lg N - 1.44 N$

$$e = 2.71828\dots \quad \phi = (1 + \sqrt{5}) / 2 = 1.61803\dots \quad \lg e = 1 / \ln 2 = 1.44269\dots$$

$$\gamma = 0.57721\dots \quad \ln 2 = 0.693147\dots$$

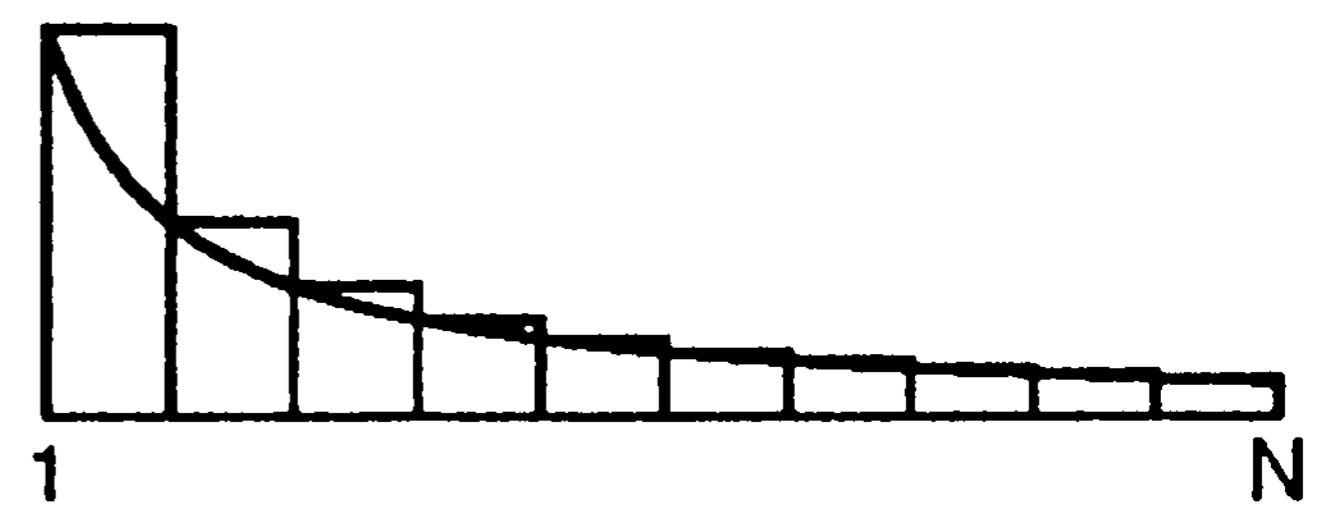


РИСУНОК 2.2  
ГАРМОНИЧЕСКИЕ ЧИСЛА

Гармонические числа представляют собой приближенные значения элементов площади под кривой  $1/x$ . Постоянная  $\gamma$  показывает разницу между  $H_N$  и

$$N = \int_1^N dx/x.$$

Последовательность чисел

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377...

определенная формулой

$$F_N = F_{N-1} + F_{N-2}, \text{ где } N \geq 2, \text{ а } F_0 = 0 \text{ и } F_1 = 1$$

известна, как *числа Фибоначчи*. Эти числа имеют множество интересных свойств. Например, отношение двух последовательных чисел приближенно равно *золотому сечению* (*golden ratio*)  $\phi = (1 + \sqrt{5}) / 2 \approx 1.61803\dots$ . Более подробный анализ показывает, что  $F_N$  равно значению выражения  $\phi^N / \sqrt{5}$ , округленному до ближайшего целого числа.

При анализе алгоритмов часто встречается также функция *факториал*  $N!$ . Как и экспоненциальная функция, факториал возникает при лобовом решении задач и растет слишком быстро, чтобы такие решения представляли практический интерес. Она также возникает при анализе алгоритмов, поскольку представляет собой количество способов упорядочения  $N$  объектов. Для аппроксимации  $N!$  используется *формула Стирлинга*:

$$\lg N! \approx N \lg N - N \lg e + \lg \sqrt{2\pi N}$$

Например, формула Стирлинга показывает нам, что количество бит в представлении числа  $N!$  примерно равно  $N \lg N$ .

Большинство рассматриваемых в этой книге формул выражается посредством нескольких функций, которые мы рассмотрели в этой главе. Однако при анализе алгоритмов может встретиться множество других специальных функций. Например, классическое *биномиальное распределение* и *распределение Пуассона* играют важную роль при разработке и анализе некоторых фундаментальных поисковых алгоритмов, которые будут рассматриваться в главах 14 и 15. Функции, не приведенные здесь, обсуждаются по мере их появления.

## Упражнения

- ▷ 2.5 Для каких значений  $N$  справедливо  $10N \lg N > 2N^2$ ?
- ▷ 2.6 Для каких значений  $N$  выражение  $N^{3/2}$  имеет значение в пределах от  $N(\lg N)^2 / 2$  до  $2N(\lg N)^2$ ?
- 2.7 Для каких значений  $N$  справедливо  $2NH_N - N < N \lg N + 10N$ ?
- 2.8 Для какого наименьшего значения  $N$  справедливо  $\log_{10} \log_{10} N > 8$ ?
- 2.9 Докажите, что  $\lfloor \lg N \rfloor + 1$  — это количество бит, необходимое для представления числа  $N$  в двоичной форме.
- 2.10 Добавьте в табл. 2.2 колонки для  $N(\lg N)^2$  и  $N^{3/2}$ .
- 2.11 Добавьте в табл. 2.2 строки для  $10^7$  и  $10^8$  инструкций в секунду.
- 2.12 Напишите на C++ функцию, которая подсчитывает  $H_N$ , используя функцию  $\log$  из стандартной математической библиотеки.
- 2.13 Напишите эффективную функцию на C++, подсчитывающую  $\lceil \lg \lg N \rceil$ . Не используйте библиотечную функцию.

- 2.14 Сколько цифр в десятичном представлении числа 1 миллион факториал?
- 2.15 Сколько бит в двоичном представлении числа  $\lg(N!)$ ?
- 2.16 Сколько бит в двоичном представлении  $H_N$ ?
- 2.17 Приведите простое выражение для  $\lfloor \lg F_N \rfloor$ .
- 2.18 Приведите наименьшие значения  $N$ , для которых  $\lfloor H_N \rfloor = i$ , где  $1 \leq i \leq 10$ .
- 2.19 Приведите наибольшее значение  $N$ , для которого можно решить задачу, требующую выполнения  $f(N)$  инструкций, на машине с быстродействием  $10^9$  операций в секунду для следующих функций  $f(N)$ :  $N^{3/2}$ ,  $N^{5/4}$ ,  $2N H_N$ ,  $N \lg N \lg \lg N$  и  $N^2 \lg N$ .

## 2.4 O-нотация

Математическая запись, позволяющая отбрасывать подробности при анализе алгоритмов, называется *O-нотацией*. Она определена следующим образом.

**Определение 2.1** *Говорят, что функция  $g(N)$  является  $O(f(N))$ , если существуют такие постоянные  $c_0$  и  $N_0$ , что  $g(N) < c_0 f(N)$  для всех  $N > N_0$ .*

*O*-нотация используется по трем основным причинам:

- Чтобы ограничить ошибку, возникающую при отбрасывании малых слагаемых в математических формулах.
- Чтобы ограничить ошибку, возникающую тогда, когда не учитываются те части программы, которые дают малый вклад в анализируемую сумму.
- Чтобы классифицировать алгоритмы согласно верхней границе их общего времени выполнения.

Третье назначение *O*-нотации рассматривается в разделе 2.7, а два других кратко обсуждаются ниже.

Постоянные  $c_0$  и  $N_0$ , не выраженные явно в *O*-нотации, часто скрывают практически важные подробности реализации. Очевидно, что выражение "алгоритм имеет время выполнения  $O(f(N))$ " ничего не говорит о времени выполнения при  $N$ , меньшем  $N_0$ , а  $c_0$  может иметь большое значение, необходимое, чтобы обойти случай нежелательной низкой производительности. Нам хотелось бы использовать алгоритм, время выполнения которого составляет  $N^2$  наносекунд, а не  $\log N$  столетий, но мы не можем сделать такого выбора на основе *O*-нотации.

Часто результаты математического анализа являются не точными, но приближенными в точном техническом смысле: результат представляет собой выражение, содержащее последовательность убывающих слагаемых. Так же, как мы интересуемся, в основном, внутренним циклом программы, мы интересуемся больше всего *главными членами* (наибольшими по величине слагаемыми) математического выражения. *O*-нотация позволяет рассматривать главные члены и опускать меньшие слагаемые при работе с приближенными математическими выражениями, а также записывать краткие выражения, дающие точные приближения для анализируемых величин.

Некоторые из основных действий, которыми мы пользуемся при работе с выражениями, содержащими *O*-нотацию, являются предметом упражнений 2.20—2.25. Многие из этих действий интуитивно понятны, а склонные к математике читатели



могут с интересом выполнить упражнение 2.21, где требуется доказать обоснованность базовых операций из определения. Существенно то, что из этих упражнений следует, что можно раскрывать скобки в алгебраических выражениях с  $O$ -нотацией так, как, если бы она в них не присутствовала, а затем отбрасывать все слагаемые, кроме наибольшего. Например, если требуется раскрыть скобки в выражении

$$(N + O(1))(N + O(\log N) + O(1)),$$

то мы получим шесть слагаемых

$$N^2 + O(N) + O(N \log N) + O(\log N) + O(N) + O(1).$$

Однако мы можем отбросить все слагаемые кроме наибольшего из них, оставив только

$$N^2 + O(N \log N).$$

То есть,  $N^2$  является хорошей аппроксимацией этого выражения, когда  $N$  велико. Эти действия интуитивно ясны, но  $O$ -нотация позволяет выразить их с математической точностью. Формула с одним  $O$ -слагаемым называется *асимптотическим выражением* (*asymptotic expression*).

В качестве более важного примера предположим (после некоторого математического анализа), что определенный алгоритм имеет внутренний цикл, итерируемый в среднем  $NH_N$  раз, внешнюю секцию, итерируемую  $N$  раз, и некоторый код инициализации, исполняемый единожды. Далее предположим, что мы определили (после тщательного исследования реализации), что каждая итерация внутреннего цикла требует  $a_0$  наносекунд, внешняя секция —  $a_1$  наносекунд, а код инициализации —  $a_2$  наносекунд. Тогда среднее время выполнения программы (в наносекундах) равно

$$2a_0 N H_N + a_1 N + a_2.$$

Поэтому для времени выполнения справедлива следующая формула:

$$2a_0 N H_N + O(N).$$

Более простая формула важна, поскольку из нее следует, что при больших  $N$  нет необходимости искать значения величин  $a_1$  и  $a_2$  для аппроксимации времени выполнения. В общем случае, в точном математическом выражении для времени выполнения может содержаться множество других слагаемых, ряд из которых трудно анализировать.  $O$ -нотация обеспечивает способ получения приближенного ответа для больших  $N$  без привлечения слагаемых подобного рода.

Продолжая данный пример, можно воспользоваться  $O$ -нотацией, чтобы выразить время выполнения с помощью известной функции,  $\ln N$ . Благодаря  $O$ -нотации, приближенное выражение из табл. 2.3 можно записать как  $H_N = \ln N + O(1)$ . Таким образом,  $a_0 N \ln N + O(N)$  — это асимптотическое выражение для общего времени выполнения алгоритма. То есть, при больших  $N$  оно будет близко к легко вычисляемому выражению  $2a_0 N \ln N$ . Постоянный множитель зависит от времени, которое требуется для выполнения инструкций внутреннего цикла.

Более того, нам не нужно знать значения  $a_0$ , чтобы предсказать, что время выполнения для ввода размером  $2N$  будет вдвое больше, чем для ввода размером  $N$ , для больших  $N$ , поскольку

$$\frac{2a_0(2N)\ln(2N)+O(2N)}{2a_0N\ln N+O(N)} = \frac{2\ln(2N)+O(1)}{\ln N+O(1)} = 2 + O\left(\frac{1}{\log N}\right)$$

Таким образом, асимптотическая формула позволяет нам делать точные прогнозы, не вдаваясь в подробности реализации или анализа. Отметим, что такое предсказание *не* было бы возможным, если бы  $O$ -аппроксимация была задана только для главного члена.

Намеченный способ рассуждений позволяет ограничиться только главным членом при сравнении или предсказании времени выполнения алгоритмов. Зачастую мы подсчитываем время выполнения константно-временных операций и хотим ограничиться *только* главным членом, подразумевая неявно, что точный анализ наподобие приведенного выше можно всегда провести, если потребуется.

Когда функция  $f(N)$  является асимптотически большой по сравнению с другой функцией  $g(N)$  (т.е.  $g(N)/f(N) \rightarrow 0$ , когда  $N \rightarrow \infty$ ), мы иногда в книге используем терминологию (бесспорно нетехническую) *порядка*  $f(N)$ , подразумевая  $f(N) + O(g(N))$ . То, что, кажется, теряется в математической точности, компенсируется доходчивостью, так как нас больше интересует производительность алгоритмов, а не математические детали. В таких случаях мы можем быть уверены в том, что при больших значениях  $N$  (если даже не при всех значениях) исследуемая величина будет близка по величине к  $f(N)$ . Например, даже если мы знаем, что величина равна  $N(N-1)/2$ , мы можем говорить о ней, как  $N^2/2$ . Такой способ выражения результатов становится понятным быстрее, чем подробный и точный результат, и отличается от правильного значения всего лишь на 0.1 процента для, например,  $N = 1000$ . Потеря точности в данном случае намного меньше, чем при распространении использования  $O(f(N))$ . Наша цель состоит в том, чтобы быть одновременно и точными, и краткими при описании производительности алгоритмов.

В похожем ключе мы иногда говорим, что время выполнения алгоритма *пропорционально*  $f(N)$ , т.е. можно доказать, что оно равно  $cf(N) + g(N)$ , где  $g(N)$  асимптотически мало по сравнению с  $f(N)$ . В рамках такого подхода мы можем предсказать время выполнения для  $2N$ , если оно известно для  $N$ , как в примере, который обсуждался выше. На рис. 2.3 приводятся значения множителей для таких прогнозов поведения функций, которые часто возникают во время анализа алгоритмов. В соединении с эмпирическим изучением (см. раздел 2.1) данный подход освобож-

1	нет влияния
$\lg N$	небольшой рост
$N$	удвоение
$N \lg N$	чуть более, чем удвоение
$N^{3/2}$	множитель $2\sqrt{2}$
$N^2$	множитель 4
$N^3$	множитель 8
$2^N$	в квадрате

### РИСУНОК 2.3 ВЛИЯНИЕ УДВОЕНИЯ РАЗМЕРОВ ЗАДАЧИ НА ВРЕМЯ ВЫПОЛНЕНИЯ.

*Прогнозирование влияния удвоения размеров задачи на время выполнения является простой задачей, когда время выполнения пропорционально одной из простых функций, как показано в таблице. В теории это влияние можно вычислить только когда  $N$  велико, но данный метод на удивление эффективен. И наоборот, быстрый метод определения функционального роста времени выполнения программы заключается в запуске программы, удвоении количества вводов для максимально возможного  $N$ , а затем оценка нового времени выполнения согласно приведенной таблице.*

дает от определения постоянных величин, зависящих от реализации. Или же, применяя его в обратном направлении, мы можем часто разработать гипотезу о функциональной зависимости времени выполнения программы, изучив, как меняется время выполнения при увеличении  $N$  в два раза.

Различия между  $O$ -границами *пропорционально* (*is proportional to*) и *порядка* (*about*) проиллюстрированы на рис. 2.4 и 2.5.  $O$ -нотация используется, прежде всего, для исследования фундаментального асимптотического поведения алгоритма; *пропорционально* требуется при экстраполяции производительности на основе эмпирического изучения, а *порядка* — при сравнении производительности разных алгоритмов или при предсказании абсолютной производительности.

## Упражнения

▷ 2.20 Докажите, что  $O(1)$  — то же самое, что и  $O(2)$ .

2.21 Докажите, что в выражениях с  $O$ -нотацией можно производить любое из перечисленных преобразований

$$f(N) \rightarrow O(f(N)),$$

$$cO(f(N)) \rightarrow O(f(N)),$$

$$O(cf(N)) \rightarrow O(f(N)),$$

$$f(N) - g(N) = O(h(N)) \rightarrow f(N) = g(N) + O(h(N)),$$

$$O(f(N))O(g(N)) \rightarrow O(f(N)g(N)),$$

$$O(f(N)) + O(g(N)) \rightarrow O(g(N))$$

$$\text{если } f(N) = O(g(N)).$$

○ 2.22 Покажите, что  $(N + 1)(H_N + O(1)) = N \ln N + O(N)$ .

2.23 Покажите, что  $N \ln N = O(N^{3/2})$ .

● 2.24 Покажите, что  $N^M = O(\alpha^N)$  для любого  $M$  и любого постоянного  $\alpha > 1$ .

● 2.25 Докажите, что

$$\frac{N}{N + O(1)} = 1 + O\left(\frac{1}{N}\right).$$

2.26 Предположим, что  $H_k = N$ . Найдите приближенную формулу, которая выражает  $k$  как функцию  $N$ .

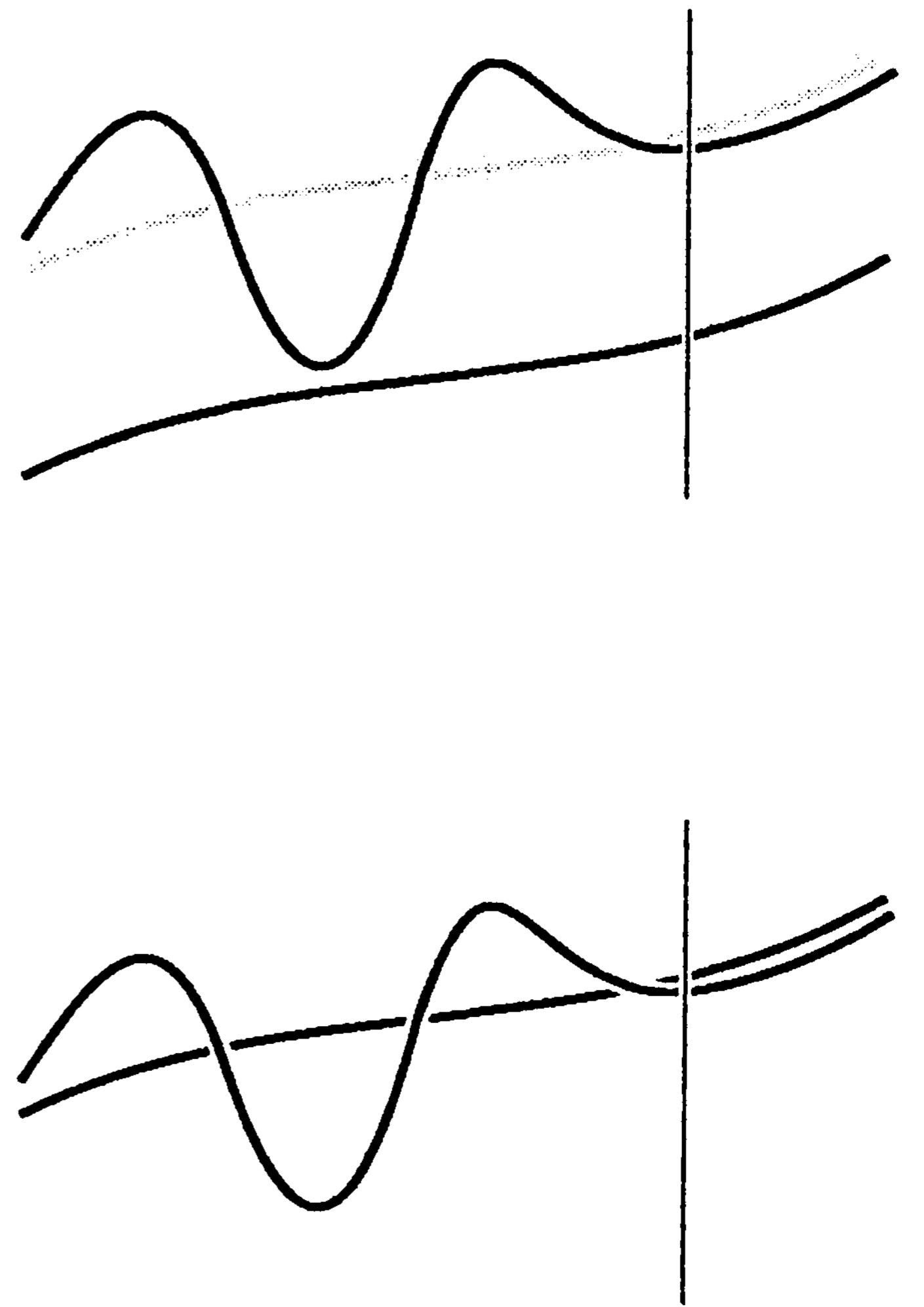
● 2.27 Предположим, что  $\lg(k!) = N$ . Найдите приближенную формулу, которая выражает  $k$  как функцию  $N$ .



**РИСУНОК 2.4 ОГРАНИЧЕНИЕ ФУНКЦИИ С ПОМОЩЬЮ  $O$ -АППРОКСИМАЦИИ**

На этой схематической диаграмме осциллирующая кривая представляет собой функцию  $g(N)$ , которую мы пытаемся аппроксимировать; гладкая черная кривая представляет собой другую функцию,  $f(N)$ , которая используется для аппроксимации, а гладкая серая кривая является функцией  $cf(N)$  с некоторой неопределенной постоянной  $c$ . Вертикальная прямая задает значение  $N_0$ , указывающее, что аппроксимация справедлива для  $N > N_0$ . Когда мы говорим, что  $g(N) = O(f(N))$ , мы ожидаем лишь то, что значение функции  $g(N)$  находится ниже некоторой кривой, имеющей форму функции  $f(N)$ , и правее некоторой вертикальной прямой. Поведение функции  $f(N)$  может быть любым (например, она не обязательно должна быть непрерывной).

- 2.28 Известно, что время выполнения одного алгоритма равно  $O(N \log N)$ , а другого —  $O(N^3)$ . Что данное выражение неявно говорит об относительной производительности алгоритмов?
- 2.29 Известно, что время выполнения одного алгоритма всегда порядка  $N \log N$ , а другого —  $O(N^3)$ . Что данное выражение неявно говорит об относительной производительности алгоритмов?
- 2.30 Известно, что время выполнения одного алгоритма всегда порядка  $N \log N$ , а другого — всегда порядка  $N^3$ . Что данное выражение неявно говорит об относительной производительности алгоритмов?
- 2.31 Известно, что время выполнения одного алгоритма всегда пропорционально  $N \log N$ , а другого — всегда пропорционально  $N^3$ . Что данное выражение неявно говорит об относительной производительности алгоритмов?
- 2.32 Выведите значения множителей, приведенных на рис. 2.3: для каждой функции  $f(N)$ , показанной слева, найдите асимптотическую формулу для  $f(2N) / f(N)$ .



**РИСУНОК 2.5 АППРОКСИМАЦИЯ ФУНКЦИЙ**

*Когда говорят, что функция  $g(N)$  пропорциональна функции  $f(N)$  (верхний график), то подразумевают, что она растет как  $f(N)$ , но, возможно, смещена относительно последней на неизвестную постоянную. Если задано некоторое значение  $g(N)$ , можно предсказать поведение функции при больших  $N$ . Когда говорят, что  $g(N)$  порядка  $f(N)$  (нижний график), то подразумевают, что функцию  $f$  можно использовать для более точной оценки значений функции  $g$ .*

## 2.5 Простейшие рекурсии

Как мы увидим далее в этой книге, многие алгоритмы основаны на принципе рекурсивного разбиения большой задачи на меньшие, когда решения подзадач используются для решения исходной задачи. Эта тема подробно обсуждается в главе 5, в основном, с практической точки зрения, где основное внимание уделено различным реализациям и приложениям. Кроме того, в разделе 2.6 приводится подробный пример. В этом разделе исследуются простейшие методы анализа таких алгоритмов и вывод решений нескольких стандартных формул, которые возникают при анализе многих изучаемых алгоритмов. Понимание математических свойств формул в этом разделе дает необходимое понимание свойств производительности алгоритмов.

Рекурсивное разбиение алгоритма напрямую проявляется в его анализе. Например, время выполнения подобных алгоритмов определяется величиной и номером подзадачи, а также временем, необходимым для разбиения задачи. Математически зависимость времени выполнения алгоритма для ввода величиной  $N$  от времени выполнения при меньшем количестве вводов легко задается с помощью *рекуррентных соотношений*. Такие формулы точно описывают производительность алгоритмов: для вычисления времени выполнения необходимо решить эти рекурсии. Более строгие

аргументы, связанные со спецификой алгоритмов, встретятся при их непосредственном рассмотрении; здесь же внимание уделяется только формулам самим по себе.

**Формула 2.1** В рекурсивной программе, где при каждой итерации цикла количество вводов уменьшается на единицу, возникает следующее рекуррентное соотношение:

$$C_N = C_{N-1} + N, \text{ где } N \geq 2 \text{ и } C_1 = 1.$$

**Решение:**  $C_N$  порядка  $N^2/2$ . Для решения рекурсии ее можно раскрыть, применяя саму к себе следующим образом:

$$\begin{aligned} C_N &= C_{N-1} + N \\ &= C_{N-2} + (N-1) + N \\ &= C_{N-3} + (N-2) + (N-1) + N \\ &\vdots \end{aligned}$$

Продолжая таким же образом, можно получить

$$\begin{aligned} C_N &= C_1 + 2 + \dots + (N-2) + (N-1) + N \\ &= 1 + 2 + \dots + (N-2) + (N-1) + N \\ &= \frac{N(N+1)}{2}. \end{aligned}$$

Подсчет суммы  $1 + 2 + \dots + (N-2) + (N-1) + N$  элементарен: прибавим к сумме ее же, но в обратном порядке. Результирующая сумма — удвоенный искомый результат — будет состоять из  $N$  слагаемых, каждое из которых равно  $N + 1$ .

**Формула 2.2** В рекурсивной программе, где на каждом шаге количество вводов уменьшается вдвое, возникает следующее рекуррентное соотношение:

$$C_N = C_{N/2} + 1, \text{ где } N \geq 2 \text{ и } C_1 = 1.$$

**Решение:**  $C_N$  порядка  $\lg N$ . Из написанного следует, что это уравнение бессмысленно за исключением случая, когда  $N$  четно или же предполагается, что  $N/2$  — целочисленное деление. Сейчас предположим, что  $N = 2^n$ , чтобы рекурсия была всегда определена. (Заметьте, что  $n = \lg N$ .) Тогда рекурсию еще проще раскрыть, чем в предыдущем случае:

$$\begin{aligned} C_{2^n} &= C_{2^{n-1}} + 1 \\ &= C_{2^{n-2}} + 1 + 1 \\ &= C_{2^{n-3}} + 3 \\ &\vdots \\ &= C_{2^0} + n \\ &= n + 1. \end{aligned}$$

Точное решение для любого  $N$  зависит от интерпретации  $N/2$ . Если  $N/2$  представляет собой  $\lfloor N/2 \rfloor$ , тогда существует очень простое решение:  $C_N$  — это количество бит в двоичном представлении числа  $N$ , т.е. по определению  $\lfloor \lg N \rfloor + 1$ . Этот вы-

вод немедленно следует из того, что операция отбрасывания правого бита в двоичном представлении любого числа  $N > 0$  превращает его в  $\lfloor N/2 \rfloor$  (см. рис. 2.6).

**Формула 2.3** В рекурсивной программе, где количество вводов уменьшается вдвое, но необходимо проверить каждый элемент, возникает следующее рекуррентное соотношение:

$$C_N = C_{N/2} + N, \text{ где } N \geq 2 \text{ и } C_1 = 0.$$

**Решение:**  $C_N$  порядка  $2N$ . Рекурсия раскрывается в сумму  $N + N/2 + N/4 + N/8 + \dots$  (Как и формуле 2.2, рекуррентное соотношение определено точно только в том случае, если  $N$  является степенью числа 2). Если данная последовательность бесконечна, то сумма простой геометрической прогрессии равна  $2N$ . Поскольку мы используем целочисленное деление и останавливаемся на 1, данное значение является приближением к точному ответу. В точном решении используются свойства двоичного представления числа  $N$ .

**Формула 2.4** В рекурсивной программе, которая должна выполнить линейный проход по вводам до, в течение или после разбиения их на две половины, возникает следующее рекуррентное соотношение:

$$C_N = 2C_{N/2} + N, \text{ где } N \geq 2 \text{ и } C_1 = 0.$$

**Решение:**  $C_N$  порядка  $N \lg N$ . На это решение ссылаются намного шире, чем на остальные из приведенных здесь, поскольку эта рекурсия используется в целом семействе алгоритмов "разделяй и властвуй".

$$C_{2^n} = 2C_{2^{n-1}} + 2^n$$

$$\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1$$

$$= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1$$

$$\vdots$$

$$= n.$$

Мы находим решение почти так же, как это было сделано в формуле 2.2, но с дополнительным приемом на втором шаге — делением обеих частей равенства на  $2^n$ , который позволяет раскрыть рекурсию.

$N$	$(N)_2$	$\lfloor \lg N \rfloor + 1$
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
6	110	3
7	111	3
8	1000	4
9	1001	4
10	1010	4
11	1011	4
12	1100	4
13	1101	4
14	1110	4
15	1111	4

### РИСУНОК 2.6 ЦЕЛОЧИСЛЕННЫЕ ФУНКЦИИ И ДВОИЧНЫЕ ПРЕДСТАВЛЕНИЯ.

Рассматривая двоичное представление числа  $N$  (в центре), получим  $\lfloor N/2 \rfloor$  путем отбрасывания правого бита. То есть, количество бит в двоичном представлении числа  $N$  на единицу больше, чем в представлении числа  $\lfloor N/2 \rfloor$ . Поэтому  $\lfloor \lg N \rfloor + 1$ , количество бит в двоичном представлении числа  $N$ , является решением формулы 2.2 в случае, если  $N/2$  интерпретируется, как  $\lfloor N/2 \rfloor$ .

**Формула 2.5** В рекурсивной программе, которая разбивает ввод надвое, а затем производит постоянное количество других операций (см. главу 5) возникает следующая рекурсия.

$$C_N = 2C_{N/2} + 1, \text{ где } N \geq 2 \text{ и } C_1 = 1.$$

**Решение:**  $C_N$  порядка  $2N$ . Это решение можно получить так же, как и решение формулы 2.4.

С помощью приведенных методов можно решить различные варианты подобных формул, имеющих другие начальные условия или небольшие отличия в добавочном члене, но необходимо помнить, что некоторые рекурсии, кажущиеся похожими, могут иметь гораздо более сложные решения. Существует множество специальных общих методов для математически строгого решения таких уравнений (см. раздел ссылок). Несколько сложных рекуррентных соотношений встретится в последующих главах, где и будут обсуждаться их решения.

## Упражнения

▷ **2.33** Составьте таблицу значений  $C_N$ , заданных формулой 2.2 для  $1 \leq N \leq 32$ , считая, что  $N/2$  означает  $\lfloor N/2 \rfloor$ .

▷ **2.34** Выполните упражнение 2.33, но считая, что  $N/2$  означает  $\lceil N/2 \rceil$ .

▷ **2.35** Выполните упражнение 2.34 для формулы 2.3.

○ **2.36** Предположим, что  $f_N$  пропорционально постоянной величине и

$$C_N = C_{N/2} + f_N, \text{ где } N \geq t \text{ и } 0 \leq C_N < c \text{ при } N < t,$$

где  $c$  и  $t$  — постоянные. Покажите, что  $C_N$  пропорционально  $\lg N$ .

● **2.37** Сформулируйте и докажите обобщенные версии формул 2.3—2.5, аналогичные обобщенной версии формулы 2.2 в упражнении 2.36.

**2.38** Составьте таблицу значений  $C_N$ , заданных формулой 2.4 при  $1 \leq N \leq 32$  для трех следующих случаев: (i)  $N/2$  означает  $\lfloor N/2 \rfloor$ , (ii)  $N/2$  означает  $\lceil N/2 \rceil$ , (iii)  $2C_{N/2}$  равно  $C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil}$ .

**2.39** Решите формулу 2.4 для случая, когда  $N/2$  означает  $\lfloor N/2 \rfloor$ , используя соответствие двоичному представлению числа  $N$ , как это было сделано в доказательстве формулы 2.2. *Подсказка:* Рассмотрите все числа, меньшие  $N$ .

**2.40** Решите рекурсию

$$C_N = C_{N/2} + N^2 \text{ при } N \geq 2 \text{ и } C_1 = 0,$$

когда  $N$  является степенью числа 2.

**2.41** Решите рекурсию

$$C_N = C_{N/\alpha} + 1 \text{ при } N \geq 2 \text{ и } C_1 = 0,$$

когда  $N$  является степенью числа  $\alpha$ .

○ **2.42** Решите рекурсию

$$C_N = \alpha C_{N/2} \text{ при } N \geq 2 \text{ и } C_1 = 1,$$

когда  $N$  является степенью числа 2.

- 2.43 Решите рекурсию

$$C_N = (C_{N/2})^2 \text{ при } N \geq 2 \text{ и } C_1 = 1,$$

когда  $N$  является степенью числа 2.

- 2.44 Решите рекурсию

$$C_N = \left(2 + \frac{1}{\lg N}\right) C_{N/2} \text{ при } N \geq 2 \text{ и } C_1 = 1,$$

когда  $N$  является степенью числа 2.

- 2.45 Рассмотрите семейство рекурсий наподобие формулы 2.1, где  $N/2$  может означать  $\lfloor N/2 \rfloor$  или  $\lceil N/2 \rceil$ , и единственное требование заключается в том, чтобы рекурсия имела место при  $N > c_0$  и  $C_N = O(1)$  при  $N \leq c_0$ . Докажите, что решением всех таких рекурсий является формула  $\lg N + O(1)$ .
- 2.46 Выведите обобщенные рекурсии и их решения, как в упражнении 2.45, для формул 2.2—2.5.

## 2.6 Примеры алгоритмического анализа

Вооружившись инструментами, о которых было рассказано в трех предыдущих разделах, мы рассмотрим анализ последовательного и бинарного поиска — двух основных алгоритмов для определения того, входит ли некоторая последовательность объектов в заданный набор объектов. Нашей целью является иллюстрация того, как можно сравнивать алгоритмы, а не подробное описание самих алгоритмов. Для простоты предположим, что все рассматриваемые объекты являются целыми числами. Общие приложения рассматриваются в главах 12—16. Простые версии алгоритмов, которые изучаются сейчас, не только демонстрируют многие аспекты задачи их разработки и анализа, но и имеют прямое применение.

Например, представим себе компанию, обрабатывающую кредитные карточки и имеющую  $N$  рискованных или украденных кредитных карточек. При этом компании необходимо проверять, нет ли среди  $M$  транзакций какого-либо из этих  $N$  плохих номеров. Для общности необходимо считать  $N$  большим (скажем, порядка  $10^3$ — $10^6$ ), а  $M$  — огромным (порядка  $10^6$ — $10^9$ ). Цель анализа заключается в приблизительной оценке времен выполнения алгоритмов, когда параметры принимают значения из указанного диапазона.

Программа 2.1 реализует прямое решение проблемы поиска. Она оформлена как функция C++, обрабатывающая массив (см. главу 3), для совместимости с другими вариантами кода для этой задачи, которые мы исследуем в части 4. Однако необязательно вдаваться в детали программы для понимания алгоритма: мы сохраняем все объекты в массиве, затем для каждой транзакции мы последовательно просматриваем массив от начала до конца, проверяя, содержится ли в нем нужный номер.

Для анализа алгоритма, прежде всего, отметим, что время выполнения зависит от того, находится ли требуемый объект в массиве. Если поиск не является успешным, мы можем определить это, только проверив все  $N$  объектов, но успешный поиск может завершиться на первом, втором или любом другом объекте.



## Программа 2.1 Последовательный поиск

Данная функция проверяет, находится ли число  $v$  среди элементов массива  $a[1]$ ,  $a[1+1]$ , ...,  $a[r]$  путем последовательного сравнения с каждым элементом, начиная с начала. Если по достижении последнего элемента нужное значение не найдено, то функция возвращает значение  $-1$ . Иначе она возвращает индекс элемента массива, содержащего искомое число.

```
int search(int a[], int v, int l, int r)
{
    for (int i = l; i <= r; i++)
        if (v == a[i]) return i;
    return -1;
}
```

Поэтому время выполнения зависит от данных. Если бы все варианты поиска происходили для чисел, которые находились в первой позиции, тогда алгоритм был бы быстрым; если бы они происходили для чисел, находящихся в последней позиции, тогда алгоритм был бы медленным. В разделе 2.7 мы обсудим разницу между возможностью *гарантировать* производительность и *предсказать* производительность. В данном случае, лучшее, что мы можем гарантировать — будет исследовано не более  $N$  чисел.

Однако, чтобы сделать какой-либо прогноз, необходимо высказать предположение о данных. В данном случае предположим, что все числа выбраны случайным образом. Из него следует, например, что каждое число в таблице может с одинаковой вероятностью оказаться предметом поиска. После некоторых размышлений мы приходим к выводу, что именно это свойство поиска является наиболее важным, потому что среди случайно выбранных чисел может вообще не происходить успешного поиска (см. упражнение 2.48). В некоторых приложениях количество транзакций с успешным поиском может быть высоким, в других приложениях — низким. Чтобы не запутывать модель свойствами приложения, мы разделим задачу на два случая (успешный и неуспешный поиск) и проанализируем их независимо. Данный пример иллюстрирует, что важным моментом эффективного анализа является разработка разумной модели приложения. Наши аналитические результаты будут зависеть от доли успешных поисков, более того, они обеспечат нас информацией, необходимой для выбора различных алгоритмов для разных приложений на основании этого параметра.

**Лемма 2.1** *Последовательный поиск исследует  $N$  чисел при каждом неуспешном поиске и в среднем порядка  $N/2$  чисел при каждом успешном поиске.*

Если каждое число в таблице с равной вероятностью может быть объектом поиска, тогда

$$(1 + 2 + \dots + N) / N = (N + 1) / 2$$

является средней ценой поиска.

Лемма 2.1 подразумевает, что время выполнения программы 2.1 пропорционально  $N$ , предмет неявного допущения, что средняя цена сравнения двух чисел постоянна. Таким образом, например, можно ожидать, что если удвоить количество объектов, то и время, необходимое для поиска, также удвоится.

Последовательный поиск в неуспешном случае можно ускорить, если упорядочить числа в таблице. Сортировка чисел в таблице является предметом рассмотрения глав 6—11. Несколько алгоритмов, которые мы рассмотрим, выполняют эту задачу за время, пропорциональное  $N \log N$ , которое незначительно по сравнению с ценой поиска при очень больших  $M$ . В упорядоченной таблице можно прервать поиск сразу по достижении числа, большего, чем искомое. Такое изменение уменьшает цену последовательного поиска до  $N/2$  чисел, которые необходимо в среднем проверить при неуспешном поиске. Время для такого случая совпадает со временем для успешного поиска.

**Лемма 2.2** Алгоритм последовательного поиска в упорядоченной таблице проверяет  $N$  чисел для каждого поиска в худшем случае и порядка  $N/2$  чисел в среднем.

Здесь все еще необходимо определить модель неуспешного поиска. Этот результат следует из предположения, что поиск может с равной вероятностью закончиться на любом из  $N + 1$  интервалов, задаваемых  $N$  числами в таблице, а это непосредственно приводит к выражению

$$(1 + 2 + \dots + N + N)/N = (N + 3)/2.$$

Цена неуспешного поиска, который заканчивается до или после  $N$ -ой записи в таблице, такая же:  $N$ .

Другой способ выразить результат леммы 2.2 — это сказать, что время выполнения последовательного поиска пропорционально  $MN$  для  $M$  транзакций в среднем и худшем случае. Если мы удвоим или количество транзакций, или количество объектов в таблице, то время выполнения удвоится; если мы удвоим обе величины одновременно, тогда время выполнения вырастет в 4 раза. Этот результат свидетельствует о том, что данный метод не подходит для очень больших таблиц. Если для проверки одного числа требуется  $c$  микросекунд, а  $M = 10^9$  и  $N = 10^6$ , тогда время выполнения для всех транзакций будет, по крайней мере,  $(c/2)10^9$  секунд, или, следуя рис. 2.1, около  $16c$  лет, что совершенно не подходит.

## Программа 2.2 Бинарный поиск

Эта программа обладает такой же функциональностью, что и программа 2.1, но гораздо эффективнее.

```
int search(int a[], int v, int l, int r)
{
    while (r >= l)
    { int m = (l+r)/2;
      if (v == a[m]) return m;
      if (v < a[m]) r = m-1; else l = m+1;
    }
    return -1;
}
```

Программа 2.2 представляет собой классическое решение проблемы поиска методом, гораздо более эффективным, чем последовательный поиск. Он основан на идее, что если числа в таблице упорядочены, мы можем отбросить половину из них, после сравнения искомого значения с числом из середины таблицы. Если они равны, зна-

чит, поиск прошел успешно, если искомое число меньше, то мы применим такой же метод к левой части таблицы, если больше — то к правой. На рис. 2.7 представлен пример выполнения этого метода над набором чисел.

**Лемма 2.3** *Бинарный поиск исследует не более  $\lfloor \lg N \rfloor + 1$  чисел.*

Доказательство данной леммы иллюстрирует применение рекуррентных соотношений при анализе алгоритмов. Пусть  $T_N$  — это количество сравнений, необходимое бинарному поиску в худшем случае, тогда из того, что алгоритм сводит поиск в таблице размера  $N$  к поиску в два раза меньшей таблице, непосредственно следует, что

$$T_N \leq T_{\lfloor N/2 \rfloor} + 1, \text{ при } N \geq 2 \text{ и } T_1 = 1.$$

При поиске в таблице размером  $N$  мы проверяем число посередине, затем производим поиск в таблице размером не более  $\lfloor N/2 \rfloor$ . Реальная цена может быть меньше этого значения, так как сравнение может закончиться успешно или таблица будет иметь размер  $\lfloor N/2 \rfloor - 1$  (если  $N$  четно). Также, как это было сделано в решении формулы 2.2, легко доказать, что  $T_N \leq n + 1$  при  $N = 2^n$ , а затем получить общий результат с помощью индукции.

Лемма 2.3 позволяет решить очень большую задачу с 1 миллионом чисел при помощи 20 сравнений на транзакцию, а это обычно время, меньшее, чем требуется для чтения или записи числа на большинстве современных компьютеров. Проблема поиска настолько важна, что было разработано несколько методов еще более быстрых, чем приведенный здесь, как будет показано в главах 12—16.

Отметьте, что леммы 2.1 и 2.2 выражены через операции, наиболее часто выполняемые над данными. Как отмечено в комментарии, следующем за леммой 2.1, ожидается, что каждая операция должна занимать постоянное время, тогда можно заключить, что время выполнения бинарного поиска пропорционально  $\lg N$ , в отличие от  $N$  для последовательного поиска. При удвоении  $N$  время бинарного поиска изменяется, но не удваивается, как это имеет место для последовательного поиска. С ростом  $N$  разница между двумя методами растет.

Можно проверить аналитическое доказательство лемм 2.1 и 2.2, написав программу и протестировав алгоритм. Например, в табл. 2.4 показаны времена выполнения

1488	1488			
1578	1578			
1973	1973			
3665	3665			
4426	4426			
4548	4548			
5435	5435	5435	5435	5435
5446	5446	5446	5446	
6333	6333	6333		
6385	6385	6385		
6455	6455	6455		
6504				
6937				
6965				
7104				
7230				
8340				
8958				
9208				
9364				
9550				
9645				
9686				

**РИСУНОК 2.7 БИНАРНЫЙ ПОИСК**  
*Чтобы проверить, содержится ли число 5025 в таблице, приведенной в левой колонке, мы сначала сравниваем его с 6504, из чего следует, что дальше необходимо рассматривать первую половину массива. Затем производится сравнение с числом 4548 (середина первой половины), после чего исследуется вторая половина первой половины. Мы продолжаем этот процесс, работая с подмассивом, в котором может содержаться искомое число, если оно есть в таблице. В заключение мы получаем подмассив с одним элементом, не равным 5025, из чего следует, что 5025 в таблице не содержится.*

бинарного и последовательного поиска для  $M$  поисков в таблице размером  $N$  (включая в случае бинарного поиска и цену сортировки таблицы) при различных значениях  $M$  и  $N$ . Здесь мы не будем рассматривать реализацию программы и детальные эксперименты, поскольку похожие задачи полностью рассмотрены в главах 6 и 11, а, кроме того, использование библиотечных и внешних функций и другие детали создания программ из компонент, включая и функцию `sort`, объясняются в главе 3. В данный момент мы просто подчеркнем, что проведение эмпирического тестирования — это неотъемлемая часть оценки эффективности алгоритма.

Таблица 2.4 подтверждает наше наблюдение, что функциональный рост времени выполнения позволяет предсказать производительность в случае больших значений параметров на основе эмпирического изучения работы алгоритма при малых значениях. Объединение математического анализа и эмпирического изучения убедительно показывает, что предпочтительным алгоритмом, безусловно, является бинарный поиск.

**Таблица 2.4 Эмпирическое изучение последовательного и бинарного поиска**

Приведенные ниже относительные времена выполнения подтверждают наши аналитические результаты, что в случае  $M$  поисков в таблице из  $N$  объектов время последовательного поиска пропорционально  $MN$ , а время бинарного поиска —  $M \lg N$ . При удвоении  $N$  время последовательного поиска также удваивается, а время бинарного поиска почти не изменяется. Последовательный поиск неприменим для очень больших  $M$  при растущих  $N$ , а бинарный поиск является достаточно быстрым даже для огромных таблиц.

$N$	$M = 1000$		$M = 10000$		$M = 100000$	
	S	B	S	B	S	B
125	1	1	13	2	130	20
250	3	0	25	2	251	22
500	5	0	49	3	492	23
1250	13	0	128	3	1276	25
2500	26	1	267	3		28
5000	53	0	533	3		30
12500	134	1	1337	3		33
25000	268	1		3		35
50000	537	0		4		39
100000	1269	1		5		47

*Значения:*

- S      последовательный поиск (программа 2.1)
- B      бинарный поиск (программа 2.2)

Данный пример является прототипом общего подхода к сравнению алгоритмов. Математический анализ используется для оценки частоты, с которой алгоритм производит абстрактные операции, затем на основе этих результатов выводится функциональная форма времени выполнения, которая позволяет проверить и расширить эмпирические данные. По мере нахождения алгоритмических решений все более и более тонких вычислительных задач и усложняющемся математическом анализе свойств их производительности, мы будем обращаться к литературе за математическими результатами, чтобы основное внимание в книге уделить самим алгоритмам. Здесь нет возможности производить тщательное математическое и эмпирическое изучение всех алгоритмов, а главной задачей является определение наиболее существенных характеристик производительности. При этом, в принципе, всегда можно разработать научную основу, необходимую для правильного выбора алгоритмов для важных приложений.

## Упражнения

- ▷ **2.47** Найдите среднее число сравнений, используемых программой 2.1, если  $\alpha N$  поисков прошли успешно, а  $0 \leq \alpha \leq 1$ .
- **2.48** Оцените вероятность того, что хотя бы одно из  $M$  случайных десятизначных чисел будет содержаться в наборе из  $N$  чисел, при  $M = 10, 100, 1000$  и  $N = 10^3, 10^4, 10^5, 10^6$ .
- 2.49** Напишите программу, которая генерирует  $M$  целых чисел и помещает их в массив, затем подсчитывает количество  $N$  целых чисел, которые совпадают с одним из чисел массива, используя последовательный поиск. Запустите программу при  $M = 10, 100, 1000$  и  $N = 10, 100, 1000$ .
- **2.50** Сформулируйте и докажите лемму, аналогичную лемме 2.3 для бинарного поиска.

## 2.7 Гарантии, предсказания и ограничения

Время выполнения большинства алгоритмов зависит от входных данных. Обычно нашей целью при анализе алгоритмов является исключить каким-либо образом эту зависимость: необходимо иметь возможность сказать что-то о производительности наших программ, что почти не зависит от входных данных, так как в общем случае неизвестно, какими будут входные данные при каждом новом запуске программы. Примеры из раздела 2.6 иллюстрируют два основных подхода, которые применяются в этом случае: анализ низкой производительности и анализ средней производительности.

Изучение *низкой* производительности алгоритмов достаточно привлекательно, поскольку оно позволяет *гарантировать* что-либо о времени выполнения программ. Мы говорим, что частота, с которой запускаются определенные абстрактные операции, меньше, чем определенная функция от количества вводов, независимо от того, какими являются входные значения. Например, лемма 2.3 — пример такой гарантии для бинарного поиска, так же как лемма 1.3 — для взвешенного быстрого объединения. Если гарантии являются низкими, как в случае с бинарным поиском, тогда это благоприятная ситуация, поскольку удалось устранить ситуации, когда программа

работает медленно. Поэтому программы с хорошими характеристиками в случае низкой производительности и являются основной целью разработки алгоритмов.

Однако при анализе низкой производительности существуют и некоторые трудности. Для определенных алгоритмов может существовать весомая разница между временем, необходимым для решения задачи в случае худших входных данных, и временем, необходимым в случае данных, которые встречаются на практике. Например, быстрое объединение в случае низкой производительности требует времени выполнения, пропорционального  $N$ , и пропорционального лишь  $\log N$  для обычных данных. Часто не удается доказать, что существуют входные данные, для которых время выполнения алгоритма достигает определенного предельного значения; можно лишь доказать, что время выполнения будет ниже этого предела. Более того, для некоторых задач алгоритмы с хорошими показателями низкой производительности гораздо сложнее, чем другие алгоритмы для этих задач. Иногда возникает ситуация, когда алгоритм с хорошими характеристиками низкой производительности при работе с практическими данными оказывается медленнее, чем более простые алгоритмы, или же при незначительной разнице в скорости он требует дополнительных усилий для достижения хороших характеристик низкой производительности. Для многих приложений другие соображения — переносимость и надежность — являются более важными, чем гарантии, даваемые в случае низкой производительности. Например, как было показано в главе 1, взвешенное быстрое объединение со сжатием пути обеспечивает лучшие гарантии производительности, чем взвешенное быстрое объединение, но для типичных данных, встречающихся на практике, алгоритмы имеют почти одинаковое время выполнения.

Изучение *средней* производительности алгоритмов более существенно, поскольку оно позволяет делать *предположения* о времени выполнения программ. В простейшем случае можно точно охарактеризовать входные данные алгоритма, например, алгоритм сортировки может выполняться над массивом из  $N$  случайных целых чисел или геометрический алгоритм может обрабатывать набор из  $N$  случайных точек на плоскости с координатами между 0 и 1. Затем мы можем подсчитать, сколько раз в среднем выполняется каждая инструкция, и вычислить среднее время выполнения программы, умножив частоту выполнения каждой инструкции на время ее выполнения и суммируя по всем инструкциям.

Однако и в анализе средней производительности существуют трудности. Во-первых, входная модель может неточно характеризовать входные данные, встречающиеся на практике, или же естественная модель входных данных может вообще не существовать. Мало кто будет возражать против использования таких моделей входных данных, как "случайно упорядоченный файл", для алгоритма сортировки или "случайный набор точек" для геометрического алгоритма, и для таких моделей можно получить математические результаты, которые будут точными предположениями о производительности программ в реальных приложениях. Но как можно характеризовать входные данные для программы, которая обрабатывает текст на английском языке? Даже для алгоритмов сортировки в определенных приложениях рассматриваются другие модели кроме модели случайно упорядоченных данных. Во-вторых, анализ может требовать глубоких математических доказательств. Например, анализ случая средней производительности для алгоритмов *union-find* достаточно сложен. Хотя вы-

вод таких результатов обычно выходит за пределы этой книги, мы будем иллюстрировать их природу некоторыми классическими примерами, а также, при необходимости, будем ссылаться на важные результаты (к счастью, анализ большинства алгоритмов можно найти в исследовательской литературе). В третьих, знания среднего значения времени выполнения может оказаться недостаточно: может понадобиться среднее отклонение или другие факты о распределении времени выполнения, вывод которых может оказаться еще более трудным. В частности, нас будет интересовать вероятность того, что алгоритм окажется значительно медленнее, нежели ожидается.

Во многих случаях на первое возражение можно ответить превращением случайности в достоинство. Например, если мы случайным образом "взболтаем" массив до сортировки, тогда допущение о том, что элементы массива находятся в случайном порядке, будет выполнено. Для таких алгоритмов, называемых *рандомизированными алгоритмами*, анализ средней производительности приводит к ожидаемому времени выполнения в строгом вероятностном смысле. Более того, часто можно доказать, что вероятность того, что такой алгоритм будет медленным, пренебрежимо мала. Примеры таких алгоритмов включают в себя быструю сортировку (см. главу 9), рандомизированные BST (см. главу 13) и хеширование (см. главу 14).

*Вычислительная сложность* — это направление анализа алгоритмов, которое занимается фундаментальными *ограничениями*, могущими возникнуть при анализе алгоритмов. Общая цель заключается в определении времени выполнения для низкой производительности *лучшего* алгоритма для данной задачи с точностью до постоянного множителя. Эта функция называется *сложностью* задачи.

Анализ низкой производительности с использованием *O*-нотации освобождает аналитика от необходимости включать в рассмотрение характеристики определенной машины. Выражение "время выполнения алгоритма равно  $O(f(N))$ " не зависит от входных данных и полезно для распределения алгоритмов по категориям в независимости от входных данных и деталей реализации, и таким образом отделяет анализ алгоритма от какой-либо определенной его реализации. В анализе мы, как правило, отбрасываем постоянные множители. В большинстве случаев, если мы хотим знать, чему пропорционально время выполнения алгоритма, —  $N$  или  $\log N$ , — не имеет значения, где будет выполняться алгоритм — на небольшом компьютере или на суперкомпьютере; более того, не имеет значения даже то, хорошо или плохо реализован внутренний цикл алгоритма.

Когда можно доказать, что время выполнения алгоритма в случае низкой производительности равно  $O(f(N))$ , то говорят, что  $f(N)$  является *верхней границей* сложности задачи. Другими словами, время выполнения лучшего алгоритма не больше, чем время любого другого алгоритма для данной задачи.

Мы постоянно стремимся улучшить алгоритмы, но, в конце концов, достигаем точки, где никакое изменение не может снизить время выполнения. Для каждой задачи мы заинтересованы в том, чтобы знать, где необходимо остановиться в улучшении алгоритма, т.е. мы ищем *нижнюю границу* сложности. Для многих задач можно доказать, что *любой* алгоритм решения задачи должен использовать определенное количество фундаментальных операций. Доказательство существования нижней гра-

ницы — сложное дело, связанное с тщательным созданием модели машины и разработкой замысловатых теоретических конструкций входных данных, которые тяжело решить для любого алгоритма. Мы редко затрагиваем вопрос о нахождении нижних границ, но они представляют собой вычислительные барьеры при разработке алгоритмов, и о них будет упоминаться при необходимости.

Когда изучение сложности задачи показывает, что верхняя и нижняя границы алгоритма совпадают, тогда можно быть уверенным в том, что нет смысла пытаться разработать алгоритм, который был бы фундаментально быстрее, чем наилучший из известных. В этом случае можно сконцентрировать внимание на реализации. Например, бинарный поиск является оптимальным в том смысле, что никакой алгоритм, использующий только сравнения, сможет обойтись меньшим их количеством в случае низкой производительности, чем бинарный поиск.

Верхняя и нижняя границы совпадают также и для алгоритмов *union-find*, использующих указатели. В 1975 г. Тарьян (Tarjan) показал, что алгоритм взвешенного быстрого объединения со сжатием пути требует следования по менее, чем  $O(\lg^* V)$  указателям в случае низкой производительности, и что любой алгоритм с указателями должен следовать более, чем по постоянному числу указателей в случае низкой производительности для некоторых входных данных. Другими словами, нет смысла в поиске какого-либо нового улучшения, которое гарантировало бы решение задачи линейным числом операций  $i = a[i]$ . На практике эта разница очень мала, поскольку  $\lg^* V$  очень мало, тем не менее, нахождение простого линейного алгоритма для этой задачи было темой исследования в течение долгого времени, и найденная Тарьяном нижняя граница направила усилия исследователей на другие задачи. Более того, история показывает, что нельзя обойти функции наподобие сложной функции  $\log^*$ , поскольку они присущи этой задаче.

Многие алгоритмы в этой книге были предметом глубокого математического анализа, и исследования их производительности слишком сложны, чтобы обсуждать их здесь. Но именно на основе этих исследований мы рекомендуем многие из тех алгоритмов, которые описаны далее.

Не все алгоритмы стоят такого тщательного рассмотрения; более того, при создании алгоритмов желательно работать с приближенными признаками производительности, чтобы не загружать процесс посторонними деталями. По мере того как разработка алгоритма становится более утонченной, то же самое должно происходить и с анализом, привлекая все более сложные математические инструменты. Часто процесс создания алгоритма приводит к детальному изучению сложности, которые, в свою очередь, ведут к таким теоретическим алгоритмам, которые далеки от каких-либо практических приложений. Распространенной ошибкой является считать, что грубый анализ исследований сложности сразу же приведет к практически эффективному алгоритму. Такие предположения ведут к неприятным неожиданностям. С другой стороны, вычислительная сложность — это мощный инструмент, который помогает определить границы производительности при разработке алгоритма и может послужить отправной точкой для сужения промежутка между верхней и нижней границами.



В этой книге мы придерживаемся точки зрения, что разработка алгоритма, тщательная его реализация, математический анализ, теоретические исследования и эмпирический анализ все вместе вносят важный вклад в создание элегантных и эффективных программ. Мы стремимся получить информацию о свойствах программ всеми доступными средствами, затем модифицировать их или разработать новые программы на основе полученной информации. Мы не имеем возможности производить тщательное тестирование и анализ каждого алгоритма, запускаемого в программной среде на любой возможной машине, но мы можем воспользоваться реализациями алгоритмов, эффективность которых известна, а затем улучшать и сравнивать их, если требуется высокая производительность. По ходу книги при необходимости будут достаточно подробно рассматриваться наиболее важные методы.

## Упражнение

- 2.51 Известно, что временная сложность одной задачи равна  $N \log N$ , а другой —  $N^3$ . Что данное утверждение неявно выражает об относительной производительности определенных алгоритмов, которые решают эти задачи?

## Ссылки к части 1

Существует множество вводных учебников по программированию. Стандартной ссылкой на язык C++ является книга Страуструпа (Stroustrup), а наилучшим источником знаний о C с примерами программ, многие из которых являются полноценными программами также для C++ и написаны в том же духе, что и программы в этой книге, служит книга Кернигана и Ричи (Kernigan, Ritchie) о языке C.

Несколько вариантов алгоритмов для задачи union-find из главы 1 собраны и объяснены в статье Ван-Левена и Тарьяна (van Leewen, Tarjan).

Книги Бентли (Bentley) описывают в том же стиле, что и материал, изложенный здесь, несколько исследований производительности и использование различных подходов при разработке и реализации алгоритмов для решения различных интересных задач.

Классическая работа по анализу алгоритмов, основанная на измерениях асимптотической низкой производительности — это книга Ахо, Хопкрофта и Ульмана (Aho, Hopcroft, Ullman). Книги Кнута (Knuth) покрывают анализ средней производительности более полно и являются официальным источником определенных свойств многих алгоритмов. Книги Гонне, Баэ-Ят (Gonnet, Baez-Yates) и Кормен, Лейзерсон, Ривест (Cormen, Leiserson, Rivest) являются более современными работами. Обе они включают обширный список ссылок на исследовательскую литературу.

Книга Грэм, Кнут, Паташник (Graham, Knuth, Patshnik) рассказывает об областях математики, которые обычно встречаются при анализе алгоритмов; этот же материал разбросан и в упомянутых ранее книгах Кнута. Книга Седжвика и Флажолле представляет собой исчерпывающее введение в предмет.

- A. V. АНО, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1975.
- J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, MA, 1985; *More Programming Pearls*, Addison-Wesley, Reading, MA, 1988.
- R. Baeza-Yates and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.
- T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press/McGraw-Hill, Cambridge, MA, 1990.
- R. L. GRAHAM, D. E. KNUTH, AND O. PATASHNIK, *Concrete Mathematics*, Addison-Wesley, Reading, MA, 1988.
- B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second edition, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, third edition, Addison-Wesley, Reading, MA, 1997; *Volume 2: Seminumerical Algorithms*, third edition, Addison-Wesley, Reading, MA, 1998; *Volume 3: Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1998.
- R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.
- B. Stroustrup, *The C++ Programming Language*, third edition, Addison-Wesley, Reading MA, 1997.
- J. van Leeuwen and R. E. Tarjan, "Worst-case analysis of set-union algorithms," *Journal of the ACM*, 1984.

# Структуры данных

*В этой части:*

- 3    *Элементарные структуры данных***
- 4    *Абстрактные типы данных***
- 5    *Рекурсия и деревья***

# Элементарные структуры данных

Организация данных для обработки является важным этапом разработки программ. Для реализации многих приложений выбор структуры данных — единственное важное решение: когда выбор сделан, разработка алгоритмов не вызывает затруднений. Для одних и тех же данных различные структуры будут занимать неодинаковое дисковое пространство. Одни и те же операции с различными структурами данных создают алгоритмы неодинаковой эффективности. Выбор алгоритмов и структур данных тесно взаимосвязан. Программисты постоянно ищут способы повышения быстродействия или экономии дискового пространства за счет оптимального выбора.

Структура данных не является пассивным объектом: необходимо принимать во внимание выполняемые с ней операции (и алгоритмы, используемые для этих операций). Эта концепция формально выражена в понятии *типа данных (data type)*. В данной главе основное внимание уделяется конкретным реализациям базовых принципов, используемых для организации данных. Будут рассмотрены основные методы организации данных и управления ими, изучен ряд примеров, демонстрирующих преимущества каждого подхода, и сопутствующие вопросы, такие как управление областью хранения. В главе 4 обсуждаются *абстрактные типы данных*, в которых описание типов данных отделено от их реализации.

Ниже будет представлен обзор массивов, связанных списков и строк. Эти классические структуры данных имеют широкое применение: посредством деревьев (см. главу 5) они формируют основу почти всех алгоритмов, рассматриваемых в данной книге. Рассматриваются различные

примитивные операции для управления структурами данных, а также разработки базового набора средств, которые можно использовать для составления сложных алгоритмов.

Изучение хранения данных в виде объектов переменных размеров, а также в связанных структурах, требует знания, как система управляет областью хранения, которую она выделяет программам для данных. Эта тема рассматривается не во всех подробностях, поскольку много важных моментов зависит от системы и аппаратных средств. Тем не менее, мы ознакомимся с принципами управления хранением и несколькими базовыми механизмами решения этой задачи. Кроме того, будут рассмотрены специфические методы, в которых используются механизмы выделения области хранения для программ на C++.

В конце главы приводится несколько примеров *составных структур (compound structures)*, таких как массивы связанных списков и массивы массивов. Построение абстрактных механизмов нарастающей сложности, начиная с нижнего уровня, является постоянной темой данной книги. Рассматривается ряд примеров, которые служат основой для последующего составления более совершенных алгоритмов.

Изучаемые в этой главе структуры данных служат в качестве строительных блоков, которые можно использовать естественным образом в C++ и многих других языках программирования. В главе 5 рассматривается еще одна важная структура данных — *дерево (tree)*. Массивы, строки, связанные списки и деревья служат базовыми элементами большинства алгоритмов, о которых идет речь в книге. В главе 4 рассматривается использование конкретных представлений, разработанных на основе абстрактных типов данных. Эти представления могут применяться в различных приложениях. Остальная часть книги посвящена разработке различных вариантов базовых средств, деревьев и абстрактных типов данных для создания алгоритмов, решающих более сложные задачи. Они также могут служить основой для высокоуровневых абстрактных типов данных в различных приложениях.

## 3.1 Строительные блоки

В этом разделе рассматриваются базовые низкоуровневые конструкции, используемые для хранения и обработки информации в языке C++. Все обрабатываемые компьютером данные, в конечном счете, разбиваются на отдельные биты. Однако написание программ, обрабатывающих исключительно биты, слишком трудоемкое занятие. *Типы* позволяют указывать, как будут использоваться определенные наборы битов, а *функции* позволяют задавать операции, выполняемые над данными. *Структуры C++* используются для группирования разнородных частей информации, а *указатели (pointers)* служат для непрямой ссылки на информацию. В этом разделе изучаются основные механизмы языка C++ в контексте представления основного принципа организации программ. Главная цель — заложить основу разработки структур высшего уровня (главы 3, 4 и 5), на базе которых будет построено большинство алгоритмов, рассматриваемых в данной книге.

Программы обрабатывают информацию, которая происходит из математических или естественных языковых описаний окружающего мира. Соответственно, вычислительная среда обеспечивает встроенную поддержку основных строительных блоков

подобных описаний — чисел и символов. В языке C++ программы построены на нескольких базовых типах данных:

- Целые числа (`int`).
- Числа с плавающей точкой (`float`).
- Символы (`char`).

Принято ссылаться на эти типы по их именам в языке C++ (`int`, `float` и `char`), хотя часто используются обобщенные термины — *целое* (*integer*), *число с плавающей точкой* и *символ* (*character*). Символы чаще всего используются в абстракциях высшего уровня — например, для создания слов и предложений. Поэтому обзор символьных данных будет отложен до раздела 3.6, а пока обратимся к числам.

Для представления чисел используется фиксированное количество бит. Таким образом, тип `int` относится к целым числам определенного диапазона, который зависит от количества бит, используемых для их представления. Числа с плавающей точкой приближаются к действительным числам, а используемое для их представления количество бит определяет точность этого приближения. В языке C++ путем выбора типа достигается оптимальное соотношение точности и занимаемого пространства. Для целых допускаются типы `int`, `long int` и `short int`, а для чисел с плавающей точкой — `float` и `double`. В большинстве систем эти типы соответствуют базовым аппаратным представлениям. Количество бит, используемое для представления чисел, а, следовательно, диапазон значений (для целых) или точность (для чисел с плавающей точкой) зависят от компьютера (см. упражнение 3.1). Тем не менее, язык C++ предоставляет определенные гарантии. В этой книге, ради простоты, обычно используются термины `int` и `float`, за исключением случаев, когда необходимо подчеркнуть, что задача требует применения больших чисел.

В современном программировании при выборе типов данных больше ориентируются на потребности программы, чем на возможности компьютера, прежде всего из соображений переносимости приложений. Например, тип `short int` рассматривается как объект, который может принимать значения от `-32767` до `32767`, а не 16-битный объект. Более того, концепция целых чисел включает операции, которые могут с ними выполняться: сложение, умножение и т.д.

**Определение 3.1** *Тип данных* — это множество значений и набор операций с ними.

Операции связаны с типами, а не наоборот. При выполнении операции необходимо обеспечить, чтобы ее операнды и результат отвечали определенному типу. Пренебрежение этим правилом — распространенная ошибка программирования. В некоторых случаях C++ выполняет неявное преобразование типов; в других используется *приведение* (*casting*), или явное преобразование типов. Например, если `x` и `N` целые числа, выражение

```
((float) x) / N
```

включает оба типа преобразований: оператор `(float)` выполняет приведение — величина `x` преобразуется в значение с плавающей точкой. Затем для `N` выполняется неявное преобразование, в соответствии с правилами C++, чтобы оба аргумента оператора деления представляли значения с плавающей точкой.

Многие операции, связанные со стандартными типами данных (такие как арифметические), встроены в язык C++. Другие операции существуют в виде функций, которые определены в стандартных библиотеках функций. Остальные операции реализуются в функциях C++, которые определены в программах (см. программу 3.1). Таким образом, концепция типа данных связана не только со встроенными типами (целые, значения с плавающей точкой и символы). Разработчики часто задают собственные типы данных, что служит эффективным средством организации программных средств. При описании простой функции C++ в результате создается новый тип данных. Реализуемая функцией операция добавляется к операциям, определенным для типов данных, которые представлены аргументами функции. В некотором смысле, *каждая* программа C++ является типом данных — списком множеств значений (встроенных или других типов) и связанных с ними операций (функций). Возможно, эта концепция слишком обобщена, чтобы быть полезной, но мы убедимся в ценности упрощенного осмысления программ как типов данных.

### Программа 3.1 Описание функций

Механизм, используемый в C++ для реализации новых операций с данными, представляет собой *определение функций (function definition)*, которое демонстрируется ниже.

Все функции имеют список *аргументов* и, возможно, *возвращаемое значение (return value)*. Рассматриваемая функция **lg** имеет один аргумент и возвращаемое значение. И то и другое относится к типу **int**. Функция **main** не принимает аргументов и возвращает значение типа **int** (по умолчанию — значение **0**, которое указывает на успешное завершение).

Функция *объявляется (declare)* путем присвоения ей имени и типов возвращаемых значений. Первая строка программы ссылается на библиотечный файл, который содержит объявления **cout**, **<<** и **endl**. Вторая строка объявляет функцию **lg**. Если функция описана до ее использования (см. следующий абзац), объявление необязательно. Объявление предоставляет информацию, необходимую, чтобы другие функции *вызывали* данную с использованием аргументов допустимого типа. Вызывающая функция может использовать данную функцию в выражении подобно использованию переменных, имеющих тот же тип возвращаемых значений.

Функции *определяются (define)* посредством кода C++. Все программы на C++ содержат описание функции **main**, а данном примере также имеется описание функции **lg**. В описании функции указываются имена аргументов (называемых *параметрами*) и выражения, реализующие вычисления с этими именами, как если бы они были локальными переменными. При вызове функции эти переменные инициализируются, принимая значения передаваемых аргументов, после чего выполняется код функции. Оператор **return** служит командой завершения выполнения функции и передачи возвращаемого значения вызывающей функции. Обычно вызывающая функция не испытывает других воздействий, хотя нам встретится много исключений из этого правила.

Разграничение описаний и объявлений создает гибкость в организации программ. Например, они могут содержаться в разных файлах (см. текст). Кроме того, в простой программе, подобной примеру, описание функции **lg** можно поместить перед описанием **main** и опустить объявление.

```
#include <iostream.h>
int lg(int);
int main()
{
```

---

```

for (int N = 1000; N <= 1000000000; N *= 10)
  cout << lg(N) << " " << N << endl;
}
int lg(int N)
{
  for (int i = 0; N > 0; i++, N /= 2) ;
  return i;
}

```

---

При написании программ одна из задач заключается в их организации таким образом, чтобы они были применимы к настолько возможно более широкому диапазону ситуаций. Причина в том, что достижение этой цели может позволить применить старую программу для решения новой задачи, которая иногда совершенно не связана с проблемой, изначально поставленной перед программой. Во-первых, осмысление и точное определение используемых программой операций позволяет легко распространить ее на любой тип данных, для которого эти операции могут поддерживаться. Во-вторых, путем точного определения действий программы можно добавить выполняемую ей абстрактную операцию к операциям, которые существуют для решения новых задач.

Программа 3.2 реализует несложные вычисления с использованием простых типов данных, описанных с помощью операции **typedef** и функции (которая сама реализована посредством библиотеки функций). Главная функция ссылается на тип данных, а не встроенный тип чисел. Не указывая тип чисел, обрабатываемых программой, мы расширяем ее потенциальную область применения. Например, подобный подход может продлить время жизни программы. Когда новое обстоятельство (новое приложение, компилятор или компьютер) приводит к появлению нового типа чисел, который придется использовать, можно будет обновить программу за счет простого изменения типа данных.

### Программа 3.2 Типы чисел

---

Эта программа вычисляет среднее значение  $\mu$  и среднеквадратичное отклонение  $\sigma$  ряда целых чисел  $x_1, x_2, \dots, x_N$ , сгенерированных библиотечной процедурой **rand**. Ниже приводятся математические формулы:

$$\mu = \frac{1}{N} \sum_{1 \leq i \leq N} x_i$$

$$\sigma^2 = \frac{1}{N} \sum_{1 \leq i \leq N} (x_i - \mu)^2 = \frac{1}{N} \sum_{1 \leq i \leq N} x_i^2 - \mu^2$$

Обратите внимание, что прямая реализация формулы определения  $\sigma^2$  требует одного прохода для вычисления среднего и еще одного прохода для вычисления суммы квадратов разностей членов ряда и среднего значения. Однако преобразование формулы позволяет вычислять  $\sigma^2$  за один проход.

Объявление **typedef** используется для локализации ссылки на тот факт, что данные имеют тип **int**. Например, **typedef** и описание функции **randNum** могут содержаться в отдельном файле (на который ссылается директива **include**). Впоследствии можно будет использовать программу для тестирования случайных чисел другого типа путем изменения этого файла (см. текст).



Независимо от типа данных программа использует тип **int** для индексов и тип **float** для вычисления среднего и среднеквадратичного отклонения. Она действует только при условии, что заданы преобразования данных в тип **float**.

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
typedef int Number;
Number randNum()
{ return rand(); }
int main(int argc, char *argv[])
{ int N = atoi(argv[1]);
  float m1 = 0.0, m2 = 0.0;
  for (int i = 0; i < N; i++)
  {
    Number x = randNum();
    m1 += ((float) x)/N;
    m2 += ((float) x*x)/N;
  }
  cout << " Avg.: " << m1 << endl;
  cout << "Std. dev.: " << sqrt(m2-m1*m1) << endl;
}
```

Этот пример не представляет полного решения задачи разработки программ вычисления средних величин и среднеквадратичных отклонений, не зависящих от типа данных; подобная цель и не ставилась. Например, программа требует преобразования чисел типа **Number** в тип **float**, чтобы они могли участвовать в вычислении среднего значения и отклонения. В данном виде программа зависима от приведения к типу **float**, присущего встроенному типу **int**. Вообще говоря, можно явно описать такое приведение для любого типа, обозначаемого именем **Number**.

При попытке выполнения каких-либо действий, помимо арифметических, вскоре возникнет необходимость добавления операций к типу данных. Например, может потребоваться распечатка чисел. При любой попытке разработать тип данных, основанный на идентификации важных операций программы, необходимо отыскать компромисс между уровнем обобщения и простотой реализации.

Имеет смысл подробно остановиться на способе изменения типа данных таким образом, чтобы программа 3.2 обрабатывала другие типы чисел, скажем, **float** вместо **int**. Язык C++ предлагает ряд механизмов, позволяющих воспользоваться тем, что ссылки на тип данных локализованы. Для такой небольшой программы проще всего сделать копию файла, затем изменить объявление **typedef int Number** на **typedef float Number** и тело процедуры **randNum**, чтобы оно содержало оператор **return 1.0\*rand()/RAND\_MAX** (при этом будут возвращаться случайные числа с плавающей точкой в диапазоне от 0 до 1). Однако даже для такой простой программы этот подход неудобен, поскольку подразумевает наличия двух копий программы. Все последующие изменения программы должны быть отражены в обеих копиях. В C++ существует альтернативное решение — поместить описания **typedef** и **randNum** в отдельный *файл заголовков* (header file) с именем, например, **Number.h** и заменить их директивой

```
#include "Number.h"
```

в коде программы 3.2. Затем можно создать второй файл заголовков с другими описаниями **typedef** и **randNum** и использовать главную программу 3.2 безо всяких изменений с любым из файлов, переименовав требуемый файл в **Number.h**.

Третий метод рекомендован и широко распространен в программировании на C, C++ и других языках. Он предусматривает разбиение программы на три файла:

- *Интерфейс*, где определяется структура данных и объявляются функции, используемые для управления этой структурой
- *Реализация* функций, объявленных в интерфейсе
- *Клиентская программа*, которая использует функции, объявленные в интерфейсе, для реализации более высокого уровня абстракции

Подобная организация позволяет использовать программу 3.2 с целыми и значениями с плавающей точкой, либо расширить ее для обработки других типов данных путем простой компиляции совместно со специальным кодом для поддерживаемого типа данных. В последующих абзацах рассматриваются изменения, необходимые для данного примера.

Под термином "интерфейс" подразумевается описание типа данных. Это соглашение между клиентской программой и программой реализации. Клиент соглашается обращаться к данным только через функции, определенные в интерфейсе, а реализация соглашается предоставлять необходимые функции.

Для программы 3.2 *интерфейс* должен включать следующие объявления:

```
typedef int Number;
Number randNum();
```

Первая строка указывает на тип обрабатываемых данных, а вторая — на операции, связанные с этим типом. Этот код можно поместить в файл с именем, например, **Number.h**, где на него будут независимо ссылаться клиенты и реализации.

*Реализация* интерфейса **Number.h** заключена в функции **randNum**, которая может содержать следующий код:

```
#include <stdlib.h>
#include "Number.h"
Number randNum()
{ return rand(); }
```

Первая строка ссылается на предоставленный системой интерфейс, который описывает функцию **rand()**. Вторая строка ссылается на реализуемый интерфейс (она включена для проверки того, что реализуемая и объявленная функции имеют одинаковый тип). Две последних строки представляют собой код функции. Этот код можно сохранить в файле, например, **int.c**. Реальный код функции **rand** содержится в стандартной библиотеке времени выполнения C++.

*Клиентская* программа, соответствующая примеру 3.2, будет начинаться с директив **include** для интерфейсов, где объявлены используемые функции:

```
#include <iostream.h>
#include <math.h>
#include "Number.h"
```

После этих трех строк может следовать описание функции `main` из программы 3.2. Этот код может быть сохранен в файле с именем, например, `avg.c`.

Результатом совместной компиляции программ `avg.c` и `int.c` будут те же функциональные возможности, что и реализуемые программой 3.2. Но рассматриваемая реализация более гибкая, поскольку связанный с типом данных код инкапсулирован и может использоваться другими клиентскими программами, а также потому, что программа `avg.c` без изменений может использоваться с другими типами данных. По-прежнему предполагается, что любой тип, используемый под именем `Number`, преобразуется в тип `float`. C++ позволяет описывать это преобразование, а также описывать желаемые встроенные операторы (подобные `+=` и `<<`) как часть нового типа данных. Повторное использование имен функций или операторов в различных типах данных называется *перегрузкой* (*overloading*).

Помимо рассмотренного сценария "клиент-интерфейс-реализация" существует много других способов поддержки различных типов данных. Эта концепция выходит за рамки определенного языка программирования, либо метода реализации. Действительно, поскольку имена файлов не являются частью языка, возможно, придется изменить рекомендованный выше простой метод для функционирования в конкретной среде C++ (в системах существуют различные соглашения или правила, относящиеся к содержимому файлов заголовков, а некоторые системы требуют определенных расширений, таких как `.C` или `.cxx` для файлов программ). Одна из наиболее важных особенностей C++ заключается в понятии *классов*, которые предоставляют удобный метод описания и реализации типов данных. В этой главе за основу взято простое решение, но впоследствии практически повсеместно будут использоваться классы. Глава 4 посвящена применению классов для создания базовых типов данных, что важно для разработки алгоритмов. Кроме того, подробно освещается взаимоотношение между классами C++ и парадигмой "клиент-интерфейс-реализация".

Главная причина существования этих механизмов состоит в обеспечении поддержки групп программистов, решающих задачи создания и содержания крупных систем приложений. Однако материал главы важен потому, что на протяжении книги используются механизмы создания естественных способов замены усовершенствованных реализаций алгоритмов и структур данных старыми методами. Это позволяет сравнивать различные алгоритмы для одних и тех же прикладных задач.

Часто приходится создавать структуры данных, которые позволяют обрабатывать наборы данных. Структуры данных могут быть большими, либо иметь широкое применение. Поэтому необходима идентификация важных операций, которые будут выполняться над данными, а также знание методов эффективной реализации этих операций. Первые шаги выполнения этих задач заключаются в процессе последовательного создания абстракций более высокого уровня из абстракций низшего уровня. Этот процесс представляет собой удобный способ разработки все более мощных программ. Простейшим механизмом группировки данных в C++ являются *массивы* (*arrays*), которые рассматриваются в разделе 3.2, и *структуры* (*structures*), о чем пойдет речь ниже.

Структуры представляют собой сгруппированные типы, используемые для описания наборов данных. Этот подход позволяет управлять всем набором как единым

модулем, при этом сохраняется возможность ссылаться на отдельные компоненты по именам. В языке C++ можно использовать структуру для описания нового типа данных, а также задавать операции для него. Другими словами, осуществимо управление сгруппированными данными почти так же, как встроенными типами, вроде `int` и `float`. Можно присваивать имена переменным и передавать эти переменные в качестве аргументов функций, а также выполнять множество других операций, как будет показано ниже.

### Программа 3.3 Интерфейс типа данных `point`

---

Этот интерфейс описывает тип данных, состоящий из набора значений "пары чисел с плавающей точкой" и операции "вычисление расстояния между двумя точками".

```
struct point { float x; float y; };  
float distance(point, point);
```

---

При обработке геометрических данных используется абстрактное понятие точки на плоскости. Следовательно, можно ввести строку

```
struct point { float x; float y; };
```

для указания, что имя `point` будет использоваться для ссылки на пары чисел с плавающей точкой. Например, выражение

```
struct point a, b;
```

объявляет две переменные этого типа. Можно ссылаться по имени на отдельные члены структуры. Например, операторы

```
a.x = 1.0; a.y = 1.0; b.x = 4.0; b.y = 5.0;
```

устанавливают значения переменных таким образом, что `a` представляет точку (1,1), а `b` — точку (4,5).

Кроме того, можно передавать структуры функциям как аргументы. Например, код

```
float distance(point a, point b)  
{ float dx = a.x - b.x, dy = a.y - b.y;  
  return sqrt(dx*dx + dy*dy);  
}
```

описывает функцию, которая вычисляет расстояние между двумя точками на плоскости. Это демонстрирует естественный способ использования структур для группировки данных в типовых приложениях.

Программа 3.3 содержит интерфейс, который воплощает описание типа данных для точек на плоскости: он использует структуру для представления точек и включает операцию вычисления расстояния между двумя точками. Программа 3.4 — это функция, которая реализует операцию. Подобная схема "интерфейс-реализация" используется для описания типов данных при любой возможности, поскольку в ней описание инкапсулировано (в интерфейсе), а реализация выражена в прямой и понятной форме. Тип данных используется в клиентской программе за счет включения интерфейса и компиляции реализации совместно с клиентом (либо с помощью функций отдельной компиляции). Программа реализации 3.4 включает интерфейс (про-

грамму 3.3) для обеспечения соответствия описания функции потребностям клиента. Смысл в том, чтобы клиентские программы вполне могли обрабатывать точки без необходимости принимать какие-либо допущения об их представлении. В главе 4 будет показано, как осуществить следующий этап разделения клиента и реализации.

Пример структуры `point` прост и включает два элемента одного типа. Обычно в структурах смешиваются различные типы данных. Подобные структуры будут часто встречаться в оставшейся части главы.

Структуры позволяют группировать данные. Кроме того, в языке C++ можно связывать данные с операциями, которые должны с ними выполняться, путем использования механизма *классов*. Подробности описания классов с множеством примеров приводятся в главе 4. С применением классов можно даже использовать программу, такую как 3.2, для обработки элементов типа `point` после описания соответствующих арифметических операций и преобразований типов для точек. Эта возможность использования ранее описанных операций высокого уровня абстракции даже для вновь созданных типов данных является одной из важных и выдающихся особенностей программирования на C++. Она основана на способности непосредственного описания собственных типов данных средствами языка. При этом не только связываются данные со структурой, но и точно задаются операции над данными (а также структуры данных и алгоритмы, которые их поддерживают) посредством классов. Классы формируют основу рассматриваемых в книге реализаций. Однако перед детальным обзором описания и использования классов (в главе 4) необходимо рассмотреть ряд низкоуровневых механизмов для управления данными и объединения их.

Помимо предоставления основных типов `int`, `float` и `char`, а также возможности встраивать их в составные типы с помощью оператора `struct`, C++ допускает косвенное управление данными. *Указатель (pointer)* — это ссылка на объект в памяти (обычно реализуется в виде машинного адреса). Чтобы объявить переменную `a` как указатель на целое значение, используется выражение `int *a`. Можно ссылаться на само целое значение с помощью записи `*a`. Допускается объявление указателей на любой тип данных. Унарный оператор `&` предоставляет машинный адрес объекта. Он удобен для инициализации указателей. Например, выражение `*&a` означает то же, что `a`.

### Программа 3.4 Реализация структуры данных `point`

Здесь содержится описание функции `distance`, объявленной в программе 3.3. Используется библиотечная функция вычисления квадратного корня.

```
#include <math.h>
#include "Point.h"
float distance(point a, point b)
{ float dx = a.x - b.x, dy = a.y - b.y;
  return sqrt(dx*dx + dy*dy);
}
```

Косвенная ссылка на объект через указатель часто удобнее прямой ссылки, а также может оказаться более эффективной, особенно для больших объектов. Множество примеров этого преимущества приводится в разделах с 3.3 по 3.7. Как будет показано, еще важнее возможность использования указателей на структуру данных способами, которые поддерживают эффективные алгоритмы обработки данных. Указатели служат основой многих структур данных и алгоритмов.

Простой и важный пример использования указателей связан с описанием функции, которая должна возвращать множество значений. Например, следующая функция (использующая функции `sqrt` и `atan2` из стандартной библиотеки) преобразует декартовы координаты в полярные:

```
polar(float x, float y, float *r, float *theta)
{ *r = sqrt(x*x + y*y); *theta = atan2(y, x); }
```

Аргументы передаются этой функции по значению — если функция присваивает новое значение переменной аргумента, эта операция является локальной и скрыта от вызывающей функции. Поэтому функция не может изменять *указатели* чисел с плавающей точкой `r` и `theta`, но способна изменять значения чисел с помощью косвенной ссылки. Например, если вызывающая функция содержит объявление `float a, b`, вызов функции

```
polar(1.0, 1.0, &a, &b)
```

приведет к тому, что для `a` установится значение **1.414214** ( $\sqrt{2}$ ), а для `b` — значение **0.785398** ( $\pi/4$ ). Оператор `&` позволяет передавать адреса `a` и `b` в функцию, которая обрабатывает эти аргументы как указатели.

В языке C++ можно достичь того же результата посредством *ссылочных* параметров:

```
polar(float x, float y, float& r, float& theta)
{ r = sqrt(x*x + y*y); theta = atan2(y, x); }
```

Запись `float&` означает "ссылка на `float`". Ссылки можно рассматривать как встроенные указатели, которые автоматически сопровождаются при каждом использовании. Например, в этой функции ссылка на `theta` означает ссылку на любое значение `float`, используемое для второго аргумента вызывающей функции. Если вызывающая функция содержит объявление `float a, b`, как в примере из предыдущего абзаца, в результате вызова функции `polar(1.0, 1.0, a, b)` переменной `a` будет присвоено значение **1.414214**, а переменной `b` — значение **0.785398**.

До сих пор речь в основном шла об описании отдельных информационных элементов, обрабатываемых программами. В большинстве случаев интерес представляет работа с потенциально крупными *наборами* данных, и сейчас мы обратимся к основным методам достижения этой цели. Обычно термин *структура данных* относится к механизму организации информации для обеспечения удобных и эффективных средств управления и доступа к ней. Многие важные структуры данных основаны на одном из двух элементарных решений, рассматриваемых ниже, либо на обоих сразу. *Массив (array)* служит средством организации объектов фиксированным и последовательным образом, что более применимо для доступа, чем для управления. *Список (list)* позволяет организовать объекты в виде логической последовательности, что более приемлемо для управления, чем для доступа.

## Упражнения

- ▷ 3.1 Найти наибольшее и наименьшее числа, которые можно представить типами `int`, `long int`, `short int`, `float` и `double` в своей среде программирования.

- 3.2** Протестировать генератор случайных чисел в своей системе. Для этого сгенерировать  $N$  случайных целых чисел в диапазоне от 0 до  $r - 1$  с помощью функции `rand() % r`, вычислить среднее значение и среднеквадратичное отклонение для  $r = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 3.3** Протестировать генератор случайных чисел в своей системе. Для этого сгенерировать  $N$  случайных чисел типа `double` в диапазоне от 0 до 1, преобразуя их в целые числа диапазона от 0 до  $r - 1$  путем умножения на  $r$  и усечения результата. Затем вычислить среднее значение и среднеквадратичное отклонение для  $r = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **3.4** Выполнить упражнения 3.2 и 3.3 для  $r = 2, 4$  и  $16$ .
- 3.5** Реализовать функции, позволяющие применять программу 3.2 для случайных *разрядов* (чисел, которые могут принимать значения только 0 и 1).
- 3.6** Описать структуру, пригодную для представления игральные карты.
- 3.7** Написать клиентскую программу, которая использует типы данных программ 3.3 и 3.4, для следующей задачи: чтение последовательности точек (пар чисел с плавающей точкой) из стандартного устройства ввода и поиск точки, ближайшей к первой.
- **3.8** Добавить функцию к типу данных `point` (программы 3.3 и 3.4), которая определяет, лежат ли три точки на одной прямой, с допуском  $10^{-4}$ . Предположите, что все точки находятся в единичном квадрате.
- **3.9** Описать тип данных для *треугольников*, находящихся в единичном квадрате, включая функцию вычисления площади треугольника. Затем написать клиентскую программу, которая генерирует случайные тройки пар чисел с плавающей точкой от 0 до 1 и вычисляет среднюю площадь сгенерированных треугольников.

## 3.2 Массивы

Возможно, наиболее фундаментальной структурой данных является *массив*, который определен как примитив в C++ и большинстве других языков программирования. В примерах главы 1 уже встречалось использование массива в качестве основы разработки эффективного алгоритма. В этом разделе представлен еще ряд примеров.

Массив является фиксированным набором данных одного типа, которые хранятся в виде непрерывного ряда. Доступ к данным осуществляется по индексам. Для ссылки на  $i$ -тый элемент массива используется выражение `a[i]`. Перед выполнением ссылки программист должен в позиции `a[i]` массива сохранить некоторое значение. Кроме того, в языке C++ индексы должны быть неотрицательными и иметь величину меньшую, чем размер массива. Пренебрежение этими правилами является распространенной ошибкой программирования.

Фундаментальность массивов, как структур данных, заключается в их прямом соответствии системам памяти почти на всех компьютерах. Для извлечения содержимого слова из памяти машинный язык требует указания адреса. Таким образом, всю память компьютера можно рассматривать как массив, где адреса памяти соответствуют индексам. Большинство процессоров машинного языка транслируют программы, использующие массивы, в эффективные программы на машинном языке, в которых осуществляется прямой доступ к памяти. Можно с уверенностью сказать, что доступ

к массиву с помощью выражения  $a[i]$  требует небольшого количества машинных команд.

Простой пример использования массива демонстрируется в программе 3.5, которая распечатывает все простые числа, меньшие 10000. Используемый метод восходит к третьему столетию до н.э. и называется *решето Эратосфена* (см. рис. 3.1). Это типичный алгоритм, где используется возможность быстрого доступа к любому элементу массива по его индексу. Реализовано четыре цикла, из которых три выполняют последовательный доступ к массиву от первого до последнего элемента. В четвертом цикле перебирается весь массив, по  $i$  элементов за один раз. В некоторых случаях предпочтительна последовательная обработка, в других — используется последовательное упорядочение, поскольку оно не хуже других методов. Например, можно первый цикл программы 3.5 изменить следующим образом:

```
for (i = N-1; i > 1, i--) a[i] = 1;
```

что никак не отразится на вычислениях. Подобным же образом можно изменить порядок обхода во внутреннем цикле, либо изменить последний цикл, чтобы печатать простые числа в порядке убывания. Однако нельзя изменить порядок обхода во внешнем цикле основных вычислений, поскольку в нем обрабатываются все целые числа, меньшие  $i$ , перед проверкой элемента  $a[i]$  на соответствие простому числу.

Мы не будем подробно анализировать время выполнения программы 3.5, дабы не углубляться в теорию чисел. Однако очевидно, что время выполнения пропорционально

$$N + N/2 + N/3 + N/5 + N/7 + N/11 + \dots$$

что меньше

$$N + N/2 + N/3 + N/4 + \dots = NH_N \sim N \ln N.$$

Одно из отличительных свойств C++ состоит в том, что имя массива генерирует указатель первого элемента массива (имеющего индекс 0). Более того, допускаются простые *арифметические операции с указателями*: если  $p$  является указателем на объект определенного типа, можно записать код, предполагающий последовательное расположение объектов данного типа. При этом для ссылки на первый объект используется запись

i	2	3	5	a[i]
2	1			1
3	1			1
4	1	0		
5	1			1
6	1	0		
7	1			1
8	1	0		
9	1		0	
10	1	0		
11	1			1
12	1	0	0	
13	1			1
14	1	0		
15	1		0	
16	1	0		
17	1			1
18	1	0	0	
19	1			1
20	1	0		
21	1		0	
22	1	0		
23	1			1
24	1	0	0	
25	1			0
26	1	0		
27	1		0	
28	1	0		
29	1			1
30	1	0	0	0
31	1			1

### РИСУНОК 3.1 РЕШЕТО ЭРАТОСФЕНА

Для вычисления простых чисел, меньших 32, все элементы массива инициализируются с присвоением значения 1 (второй столбец). Это указывает, что пока не обнаружено ни одно число, которое не является простым (элементы  $a[0]$  и  $a[1]$  не используются и не показаны). Затем присваивается значение 0 тем элементам массива, индексы которых являются произведениями чисел 2, 3 и 5, поскольку эти произведения не являются простыми числами. Индексы элементов массива, у которых сохранилось значение 1, являются простыми числами (крайний правый столбец).



$*p$ , на второй —  $*(p+1)$ , на третий —  $*(p+2)$  и т.д. Другими словами, записи  $*(a+i)$  и  $a[i]$  в языке C++ эквивалентны.

Это создает альтернативный механизм доступа к объектам массивов, который иногда оказывается удобнее индексации. Он чаще всего используется для массивов символов (строк). Мы вернемся к этому вопросу в разделе 3.6.

### Программа 3.5 Решето Эратосфена

Цель программы заключается в присвоении элементам  $a[i]$  значения 1, если  $i$  простое число, и значения 0 в противном случае. Сначала значение 1 присваивается всем элементам массива. Затем присваивается значение 0 элементам, индексы которых не являются простыми числами (представляют собой произведения известных простых чисел). Если после этого некоторый элемент  $a[i]$  сохраняет значение 1, его индекс является простым числом.

Поскольку массив состоит из элементов простейшего типа, принимающих значения 0 или 1, для экономии пространства имеет смысл явно описать массив бит вместо массива целых чисел. Кроме того, в некоторых средах программирования требуется, чтобы массив был глобальным, если значение  $N$  велико, либо можно выделять пространство для массива динамически (см. программу 3.6).

```
#include <iostream.h>
static const int N = 1000;
int main()
{ int i, a[N];
  for (i = 2; i < N; i++) a[i] = 1;
  for (i = 2; i < N; i++)
    if (a[i])
      for (int j = i; j*i < N; j++) a[i*j] = 0;
  for (i = 2; i < N; i++)
    if (a[i]) cout << " " << i;
  cout << endl;
}
```

Подобно структурам, указатели массивов важны тем, что позволяют эффективно управлять массивами как высокоуровневыми объектами.

В частности, можно передать указатель на массив как аргумент функции. Это позволит функции обращаться к объектам массива без необходимости создания копии всего массива. Такая возможность необходима при написании программ, управляющих крупными массивами. Например, функции поиска, рассмотренные в разделе 2.6, используют эту особенность. Другие примеры содержатся в разделе 3.7.

В программе 3.5 предполагается, что размер массива должен быть известен заранее. Чтобы выполнить программу для другого значения  $N$ , следует изменить константу  $N$  и повторно скомпилировать программу. В программе 3.6 показано альтернативное решение: пользователь может ввести значение  $N$ , после чего будут выводиться простые числа, меньшие этой величины. Применены два основных механизма C++. В обоих осуществляется передача функциям массивов в качестве аргументов. Первый механизм обеспечивает передачу аргументов командной строки главным программам в массиве `argv` с размером `argc`. Массив `argv` является составным и включает объекты, которые сами представляют собой массивы (строки). Обзор массивов отложим до раздела 3.7, а пока примем на веру, что переменная  $N$  принимает значение, вводимое пользователем при выполнении программы.

Второй базовый механизм — оператор `new[]`, *распределяющий* область памяти, необходимый для массива во время выполнения. В нашем особом случае он возвращает указатель на массив. В некоторых языках программирования динамическое выделение памяти массивам затруднено либо вообще невозможно. В других языках этот процесс выполняется автоматически. Динамическое распределение памяти служит важной функцией программ, управляющих несколькими массивами, отдельные из которых могут иметь большой размер. В данном случае необходимо без выделения памяти предварительно объявить массив с размером, не меньшим любого значения, которое допускается при вводе. В сложной программе, где может использоваться много массивов, выполнять эти действия для каждого из них затруднительно. Обычно используется код, подобный коду программы 3.6, по причине его гибкости. Однако в определенных приложениях, где *размер* массива известен, вполне применимы простые решения, такие как в программе 3.5.

---

### Программа 3.6 Динамическое выделение памяти массиву

---

Для изменения максимального значения простого числа, вычисляемого в программе 3.5, необходима повторная компиляция программы. Вместо этого можно принимать максимальное значение из командной строки и использовать его для выделения памяти массиву во время выполнения с помощью оператора C++ `new[]`. Например, если скомпилировать программу и ввести в командной строке **1000000**, будут получены все целые числа, меньшие миллиона (если компьютер достаточно мощный для таких вычислений). Для отладки программы достаточно значения **100** (что позволит сэкономить время и пространство памяти). Этот подход в дальнейшем используется часто, хотя для краткости проверка на перерасход памяти будет опускаться.

```
int main(int argc, char *argv[])
{ int i, N = atoi(argv[1]);
  int *a = new int[N];
  if (a == 0)
    { cout << "out of memory" << endl; return 0; }
  ...
```

---

Массивы не только отражают низкоуровневые средства для доступа к данным памяти в большинстве компьютеров. Они широко распространены еще и потому, что прямо соответствуют естественным методам организации данных для приложений. Например, массивы прямо соответствуют *векторам* (математический термин для индексированных списков объектов).

Стандартная библиотека C++ содержит класс `Vector` — абстрактный объект, который можно индексировать подобно массиву (с необязательной автоматической проверкой соответствия диапазону). Однако он также способен к расширению и усечению. Это позволяет воспользоваться преимуществами массивов, но возлагать задачи проверки допустимости индексов и управления памятью на систему. Поскольку в этой книге много внимания уделяется быстройдействию, будем избегать скрытых недостатков, связанных с использованием массивов, указывая, что в коде могут быть применены также векторы (см. упражнение 3.14).

Программа 3.7 служит примером эмуляции, использующей массивы. Моделируется последовательность *попыток Бернулли* (Bernoulli trials) — известная абстрактная концепция теории вероятностей. Если подбросить монету  $N$  раз, вероятность выпадения  $k$  решек составляет

$$\binom{N}{k} \frac{1}{2^N} \approx \frac{e^{-(k-N/2)^2/N}}{\sqrt{\pi N/2}}.$$

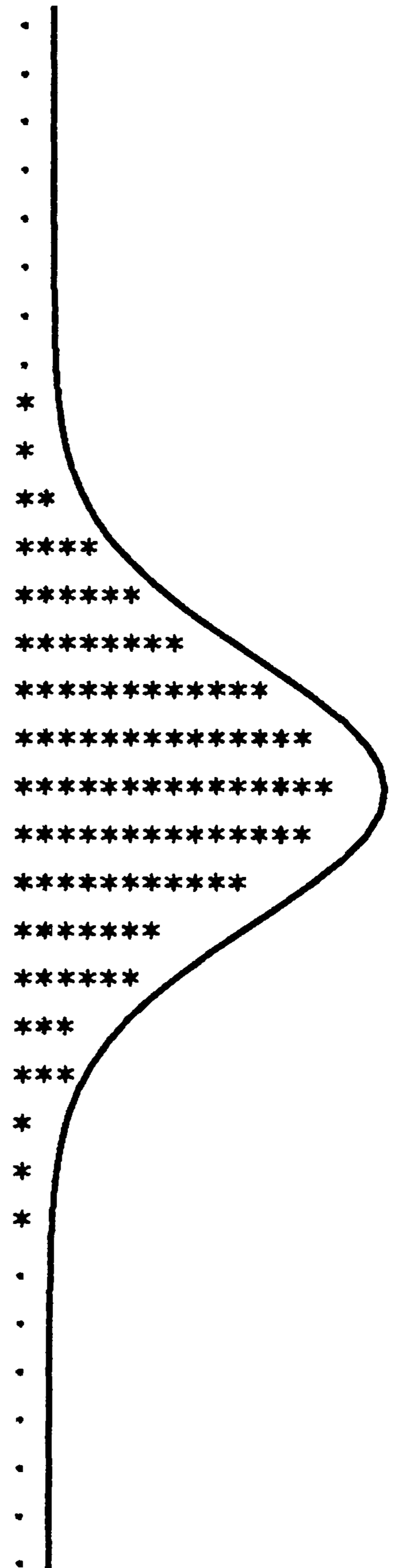
Здесь используется *нормальная аппроксимация* — известная кривая в форме колокола. На рис. 3.2 показан вывод программы 3.7 для 1000 экспериментов подбрасывания монеты по 32 раза. Дополнительные сведения о распределении Бернулли и нормальной аппроксимации можно найти в любом учебнике по теории вероятностей. Эти понятия вновь встретятся в главе 13. Пока остановимся на вычислениях, в которых используются числа и индексы массива для определения частоты выпадений. Способность поддерживать этот вид операций — одно из основных достоинств массивов.

### Программа 3.7 Имитация подбрасываний монеты

Если подбросить монету **N** раз, ожидается выпадение **N/2** решек, но это число может быть любым в диапазоне от **0** до **N**. Данная программа выполняет эксперимент **M** раз, принимая аргументы **M** и **N** из командной строки. Она использует массив **f** для отслеживания частоты выпадений "i решек" для  $0 \leq i \leq N$ , а затем распечатывает гистограмму результатов эксперимента. Каждые 10 выпадений обозначаются одной звездочкой.

Основная операция программы — индексация массива по вычисляемым значениям — важный фактор эффективности многих вычислительных процедур.

```
#include <iostream.h>
#include <stdlib.h>
int heads()
{ return rand() < RAND_MAX/2; }
int main(int argc, char *argv[])
{ int i, j, cnt;
  int N = atoi(argv[1]),
      M = atoi(argv[2]);
  int *f = new int[N+1];
  for (j = 0; j <= N; j++) f[j] = 0;
  for (i = 0; i < M; i++, f[cnt]++)
    for (cnt = 0, j = 0; j <= N; j++)
      if (heads()) cnt++;
  for (j = 0; j <= N; j++)
  {
    if (f[j] == 0) cout << ".";
    for (i = 0; i < f[j]; i+=10) cout << "*";
    cout << endl;
  }
}
```



**РИСУНОК 3.2 ИМИТАЦИЯ ПОДБРАСЫВАНИЙ МОНЕТЫ**

*Эта таблица демонстрирует результат выполнения программы 3.7 при  $N = 32$  и  $M = 1000$ .*

*Имитируется 1000 экспериментов подбрасывания монеты по 32 раза. Количество выпадений решки аппроксимируется нормальной функцией распределения, график которой показан поверх данных.*

В программах 3.5 и 3.7 вычисляются индексы массива по имеющимся данным. В некотором смысле, когда используется вычисленное значение для доступа к массиву размером  $N$ , с помощью единственной операции обрабатывается  $N$  вероятностей. Это существенно повышает эффективность. В дальнейшем будут рассматриваться алгоритмы, где массивы используются подобным же образом.

Массивы применяются для организации всего многообразия объектов, а не только целых чисел. В языке C++ можно объявлять массивы данных любых встроенных либо определяемых пользователем типов (другими словами, составных объектов, объявленных как структуры). Программа 3.8 иллюстрирует использование массива структур для точек на плоскости. Описание структуры рассматривалось в разделе 3.1. Кроме того, демонстрируется типовое использование массивов: организованное хранение данных при обеспечении быстрого доступа к ним в процессе вычислений.

Между прочим, программа 3.8 также интересна в качестве прототипа алгоритма проверки всех пар набора из  $N$  элементов данных, в результате чего затрачивается время, пропорциональное  $N^2$ . В этой книге при любом использовании подобных алгоритмов изыскиваются усовершенствования, поскольку с увеличением значения  $N$  данное решение становится труднореализуемым. В разделе 3.7 демонстрируется использование составных структур данных для вычислений в линейном масштабе времени, соответствующих данному примеру.

### Программа 3.8 Вычисление ближайшей точки

Эта программа демонстрирует использование массива структур и представляет типичный случай, когда элементы сохраняются в массиве для последующей обработки в процессе некоторых вычислений. Подсчитывается количество пар из  $N$  сгенерированных случайным образом точек на плоскости, соединяемых прямой, длина которой меньше  $d$ . При этом используется тип данных для точек, описанный в разделе 3.1. Время выполнения составляет  $O(N^2)$ , поэтому программа не может применяться для больших значений  $N$ . Программа 3.20 обеспечивает более быстрое решение.

```
#include <math.h>
#include <iostream.h>
#include <stdlib.h>
#include "Point.h"
float randFloat()
{ return 1.0*rand()/RAND_MAX; }
int main(int argc, char *argv[])
{ float d = atof(argv[2]);
  int i, cnt = 0, N = atoi(argv[1]);
  point *a = new point[N];
  for (i = 0; i < N; i++)
    { a[i].x = randFloat(); a[i].y = randFloat(); }
  for (i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
      if (distance(a[i], a[j]) < d) cnt++;
  cout << cnt << " pairs within " << d << endl;
}
```

Подобным образом можно создавать составные типы произвольной сложности: не только массивы структур, но и массивы массивов либо структур, содержащих массивы. Эти возможности будут подробно рассматриваться в разделе 3.7. Однако сначала

ознакомимся со *связными списками* (*linked lists*), которые служат главной альтернативой массивов при организации коллекций объектов.

## Упражнения

▷ **3.10** Предположим, `a` объявлена как `int a[99]`. Определить содержимое массива после выполнения следующих двух операторов:

```
for (i = 0; i < 99; i++) a[i] = 98-i;
for (i = 0; i < 99; i++) a[i] = a[a[i]];
```

**3.11** Изменить реализацию решета Эратосфена (программа 3.5) для использования массива (i) символов и (ii) разрядов. Определить влияние этих изменений на расход пространства памяти и времени, используемого программой.

▷ **3.12** С помощью решета Эратосфена определить количество простых чисел, меньших  $N$ , для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

○ **3.13** С помощью решета Эратосфена построить график зависимости  $N$  от количества простых чисел, меньших  $N$ , для значений  $N$  от 1 до 1000.

○ **3.14** Стандартная библиотека C++ в качестве альтернативы массивам содержит тип данных `Vector`. Найти способ использования этого типа данных в своей системе и определить его влияние на время выполнения, если заменить в программе 3.5 массив типом `Vector`.

● **3.15** Эмпирически определить эффект удаления проверки `a[i]` из внутреннего цикла программы 3.5 для  $N = 10^3, 10^4, 10^5$  и  $10^6$  и объяснить его.

▷ **3.16** Написать программу вычисления количества различных целых чисел, меньших 1000, которые встречаются в потоке ввода.

○ **3.17** Написать программу, эмпирически определяющую количество случайных положительных целых, меньших 1000, генерацию которых можно ожидать перед получением повторного значения.

○ **3.18** Написать программу, эмпирически определяющую количество случайных положительных целых, меньших 1000, генерацию которых можно ожидать перед получением каждого значения хотя бы один раз.

**3.19** Изменить программу 3.7 для имитации случая, когда решка выпадает с вероятностью  $p$ . Выполнить 1000 попыток эксперимента с 32 подбрасываниями при  $p = 1/6$  для получения вывода, который можно сравнить с рис. 3.2.

**3.20** Изменить программу 3.7 для имитации случая, когда решка выпадает с вероятностью  $\lambda/N$ . Выполнить 1000 попыток эксперимента с 32 подбрасываниями для получения вывода, который можно сравнить с рис. 3.2. Получается классическое распределение Пуассона.

○ **3.21** Изменить программу 3.8 для распечатывания координат пары ближайших точек.

● **3.22** Изменить программу 3.8 для выполнения тех же вычислений в  $d$ -мерном пространстве.

## 3.3 СВЯЗНЫЕ СПИСКИ

Если главный интерес представляет последовательный перебор набора элементов, их можно организовать в виде *связного списка* — базовой структуры данных, в которой каждый элемент содержит информацию, необходимую для получения следующего элемента. Основное преимущество связных списков перед массивами заключается в возможности эффективного изменения расположения элементов. За эту гибкость приходится жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в отслеживании связей от начала списка.

**Определение 3.2** *Связный список* — это набор элементов, причем каждый из них является частью узла (*node*), который также содержит ссылку (*link*) на узел.

Узлы определяются ссылками на узлы, поэтому связные списки иногда называют *самоссылочными* (*self-referent*) структурами. Более того, хотя узел обычно ссылается на другой узел, возможна ссылка на самого себя, поэтому связные списки могут представлять собой *циклические* (*cyclic*) структуры. Последствия этих двух фактов станут ясны при рассмотрении конкретных представлений и применений связных списков.

Обычно под связным списком подразумевается реализация последовательного расположения набора элементов. Начиная с некоторого узла, мы считаем его первым элементом последовательности. Затем прослеживается его ссылка на другой узел, который дает нам второй элемент последовательности и т.д. Поскольку список может быть циклическим, последовательность иногда представляется бесконечной. Чаще всего приходится иметь дело со списками, соответствующими простому последовательному расположению элементов, принимая одно из следующих соглашений для ссылки последнего узла:

- Это *пустая* (*null*) ссылка, не указывающая на какой-либо узел.
- Ссылка указывает на *фиктивный узел* (*dummy node*), который не содержит элементов.
- Ссылка указывает на первый узел, что делает список *циклическим*.

В каждом случае отслеживание ссылок от первого узла до последнего формирует последовательное расположение элементов. Массивы также задают последовательное расположение элементов, но оно реализуется косвенно, за счет позиции в массиве. (Массивы также поддерживают произвольный доступ по индексу, что невозможно для списков.)

Сначала рассмотрим узлы с единственной ссылкой. В большинстве приложений используются одномерные списки, где все узлы, за исключением, возможно, первого и последнего, имеют ровно по одной ссылке, указывающей на них. Это простейшая и наиболее интересующая нас ситуация — связные списки соответствуют последовательностям элементов. В ходе обсуждения будут исследоваться более сложные случаи.

Связные списки являются примитивными конструкциями в некоторых языках программирования, но не в C++. Однако базовые строительные блоки, о которых шла речь в разделе 3.1, хорошо приспособлены для реализации связных списков. Указатели для ссылок и структуры для узлов описываются следующим образом:

```
struct node { Item item; node *next; };  
typedef node *link;
```

Эта пара выражений представляет собой не более чем код C++ для определения 3.2. Узлы состоят из элементов (указанного здесь типа **Item**) и указателей на узлы. Указатели на узлы также называются ссылками. Более сложные случаи будут представлены в главе 4. Они обеспечивают большую гибкость и более эффективную реализацию определенных операций, но этот простой пример достаточен для понимания основ обработки списков. Подобные соглашения для связанных структур используются на протяжении всей книги.

Распределение памяти имеет ключевое значение для эффективного использования связанных списков. Выше описана единственная структура (**struct node**), но будет получено множество экземпляров этой структуры, по одному для каждого узла, который придется использовать. Как только возникает необходимость использовать новый узел, для него следует зарезервировать память. При объявлении переменной типа **node** для нее резервируется память во время компиляции. Однако часто приходится организовывать вычисления, связанные с резервированием памяти во время выполнения, посредством вызовов системных операторов управления памятью. Например, в строке кода

```
link x = new node;
```

содержится оператор **new**, который резервирует достаточный для узла объем памяти и возвращает указатель на него в переменной **x**. В разделе 3.5 будет кратко показано, как система резервирует память, поскольку это хорошее применение связанных списков!

В языке C++ широко принято *инициализировать* область хранения, а не только *выделять* для нее память. В связи с этим обычно каждую описываемую структуру включается *конструктор* (*constructor*). Конструктор представляет собой функцию, которая описывается внутри структуры и имеет такое же имя. Конструкторы подробно обсуждаются в главе 4. Они предназначены для предоставления исходных значений данным структуры. Для этого конструкторы автоматически вызываются при создании экземпляра структуры. Например, если описать узел списка при помощи следующего кода:

```
struct node  
{ Item item; node *next;  
  node (Item x; node *t)  
  { item = x; next = t; };  
};  
typedef node *link;
```

то оператор

```
link t = new node(x, t);
```

не только резервирует достаточный для узла объем памяти и возвращает указатель на него в переменной **t**, но и присваивает полю **item** узла значение **x**, а указателю поля — значение **t**. Конструкторы помогают избегать ошибок, связанных с инициализацией данных.

Теперь, когда узел списка создан, возникает задача осуществления ссылок на заключаемую в нем информацию — элемент и ссылку. Мы уже ознакомились с базовыми операциями, необходимыми для выполнения этой задачи: достаточно снять косвенность указателя, а затем использовать имена членов структуры. Ссылка  $x$  на элемент узла (тип `Item`) имеет вид  $(*x).item$ , а на ссылку (тип `link`) —  $(*x).link$ . Эти операции так часто используются, что в языке C++ для них существуют сокращенные эквиваленты:  $x->item$  и  $x->link$ . Кроме того, часто возникает необходимость в выражении: "узел, указываемый ссылкой  $x$ ", поэтому упростим его: "узел  $x$ ". Ссылка *служит именем узла*.

Соответствие между ссылками и указателями C++ имеет большое значение, но следует учитывать, что ссылки являются абстракцией, а указатели — конкретным представлением. Можно разрабатывать алгоритмы, где используются узлы и ссылки, а также выбрать одну из многочисленных реализаций этой идеи. Например, ссылки можно представлять с индексами, что будет показано в конце раздела.

Рисунки 3.3 и 3.4 иллюстрируют две основные операции, выполняемые со связными списками. Можно *удалить* любой элемент связного списка, уменьшив его длину на 1; а также *вставить* элемент в любую позицию списка путем увеличения длины на 1. В этих рисунках для простоты предполагается, что списки циклические и никогда не становятся пустыми. В разделе 3.4 рассматриваются *pull-ссылки*, фиктивные узлы и пустые списки. Как показано на рисунках, для вставки или удаления необходимо лишь два оператора C++. Для удаления узла, следующего после узла  $x$ , используются такие операторы:

```
t = x->next; x->next = t->next;
```

или проще:

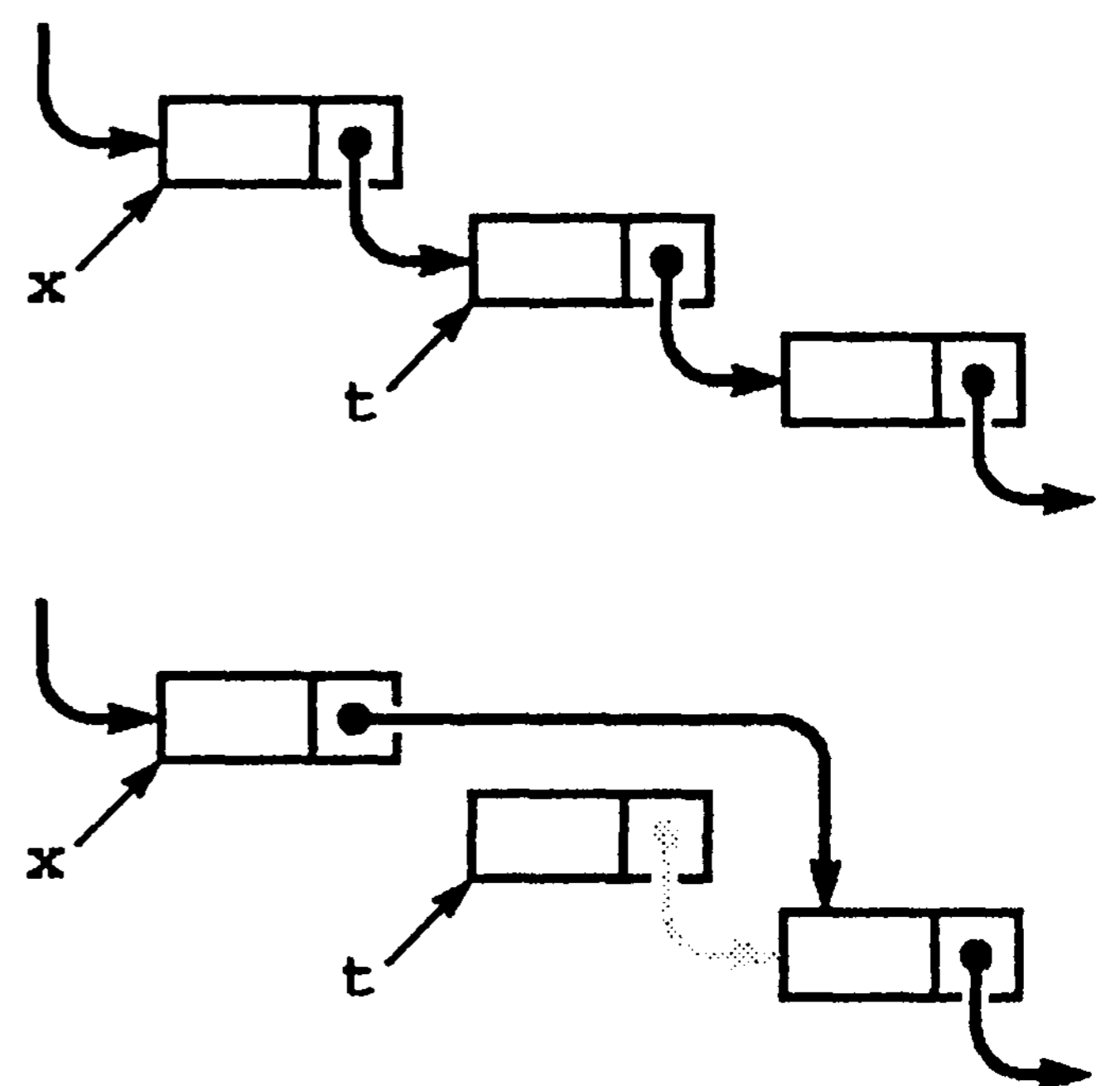
```
x->next = x->next->next;
```

Для вставки в список узла  $t$  в позицию, следующую за узлом  $x$  используются такие операторы:

```
t->next = x->next; x->next = t;
```

Простота вставки и удаления оправдывает существование связных списков. Соответствующие операции неестественны и неудобны для массивов, поскольку требуют перемещения всего содержимого массива, которое следует после затрагиваемого элемента.

Связные списки плохо приспособлены для *поиска  $k$ -того элемента* (по индексу) — операции, которая характеризует эффективность доступа к данным массивов. В массиве для доступа к  $k$ -тому элементу используется простая запись  $a[k]$ , а в списке для этого необходимо отследить  $k$  ссылок.



**РИСУНОК 3.3 УДАЛЕНИЕ В СВЯЗНОМ СПИСКЕ**

Для удаления из связного списка узла, следующего за узлом  $x$ , значение  $t$  устанавливается таким образом, чтобы указатель ссылался на удаляемый узел, а затем изменяется указатель  $x$ :  $t->next$ . Указатель  $t$  может использоваться для ссылки на удаляемый узел. Хотя его ссылка по-прежнему указывает на список, обычно подобная ссылка не используется после удаления узла из списка, за исключением, возможно, информирования системы через оператор *delete* о том, что задействованная память может быть вновь востребована.



Другая операция, неестественная для списков с единичными ссылками, — "поиск элемента, *предшествующего* данному".

После удаления узла из связного списка посредством операции `x->next = x->next->next`, повторное обращение к нему окажется невозможным. Для небольших программ, вроде рассмотренных вначале примеров, это не имеет большого значения, но хорошей практикой программирования обычно считается применение оператора `delete`. Он служит противоположностью оператора `new` для любого узла, который более не придется использовать. В частности, последовательность операторов

```
t = x->next; x->next = t->next; delete t;
```

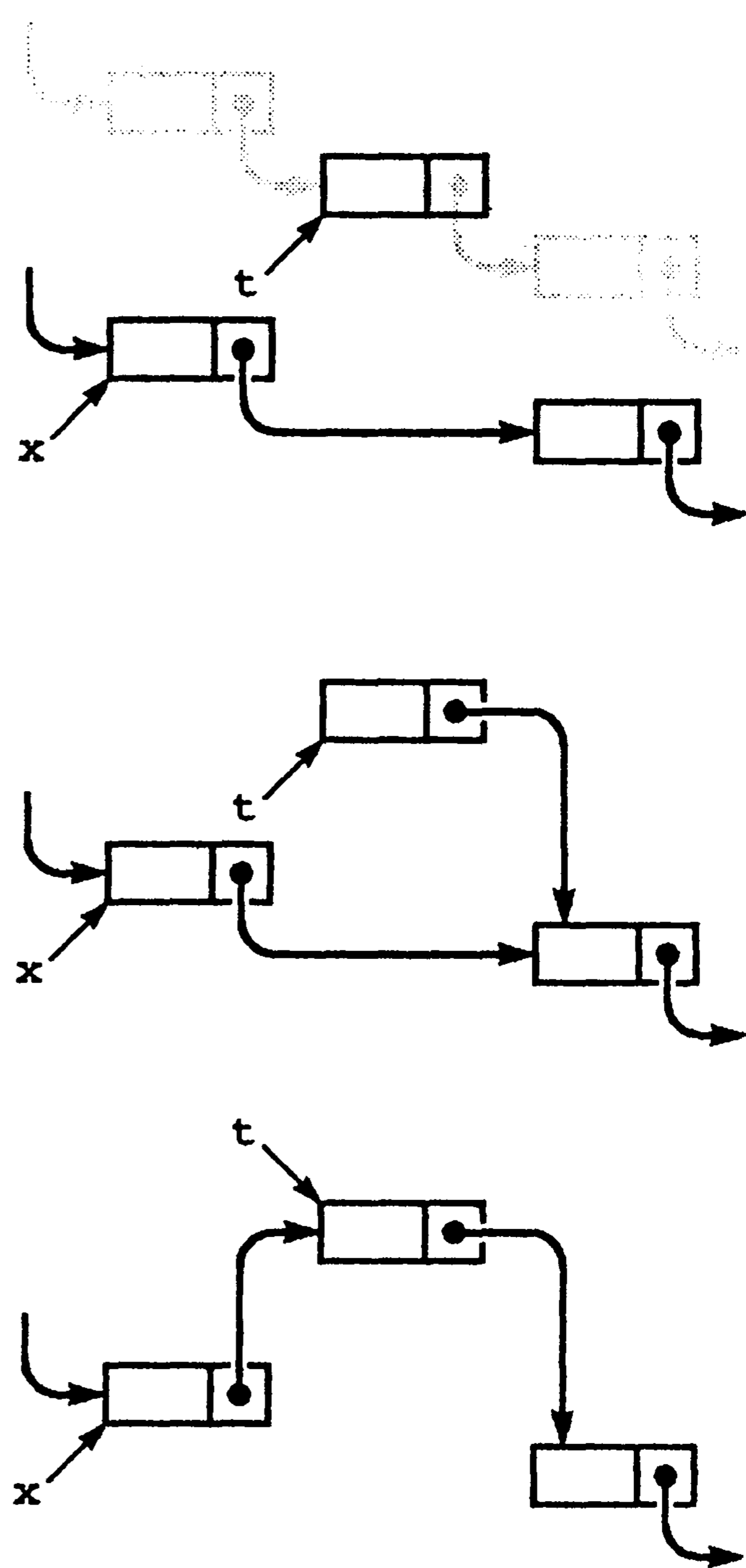
не только удаляет `t` из списка, но также информирует систему, что задействованная память может использоваться для других целей. Оператору `delete` особое внимание уделяется при наличии больших списков либо большого их количества, но до раздела 3.5 будем его игнорировать, чтобы сосредоточиться на оценке преимуществ связных структур.

В последующих главах будет приводиться много примеров использования этих и других базовых операций со связными списками. Поскольку в подобных операциях задействовано небольшое количество операторов, часто используется прямое управление списками вместо описания функций вставки, удаления и т. п. В качестве примера рассмотрим следующую программу решения задачи Иосифа (Флавия), которая служит интересным контрастом решету Эратосфена.

### Программа 3.9 Пример циклического списка (задача Иосифа)

Для представления людей, расставленных в круг, построим циклический связный список, где каждый элемент (человек) содержит ссылку на соседний элемент против хода часовой стрелки. Целое число  $i$  представляет  $i$ -того человека в круге. После создания циклического списка из одного узла вставляются узлы от **2** до **N**. В результате образуется окружность с узлами от **1** до **N**. При этом переменная `x` указывает на **N**. Затем пропускаем **M-1** узлов, начиная с 1-го, и устанавливаем значение ссылки (**M-1**)-го узла таким образом, чтобы пропустить **M**-ый узел. Продолжаем эту операцию, пока не останется один узел.

```
#include <iostream.h>
#include <stdlib.h>
struct node
{ int item; node* next;
```



**РИСУНОК 3.4 ВСТАВКА В СВЯЗНОМ СПИСКЕ**

Для вставки узла `t` в позицию связного списка, следующую за узлом `x` (верхняя диаграмма), для `t->next` устанавливается значение `x->next` (средняя диаграмма), затем для `x->next` устанавливается значение `t` (нижняя диаграмма).

```

node(int x, node* t)
{ item = x; next = t; }
};
typedef node *link;
int main(int argc, char *argv[])
{ int i, N = atoi(argv[1]), M = atoi(argv[2]);
  link t = new node(1, 0); t->next = t;
  link x = t;
  for (i = 2; i <= N; i++)
    x = (x->next = new node(i, t));
  while (x != x->next)
  {
    for (i = 1; i < M; i++) x = x->next;
    x->next = x->next->next;
  }
  cout << x->item << endl;
}

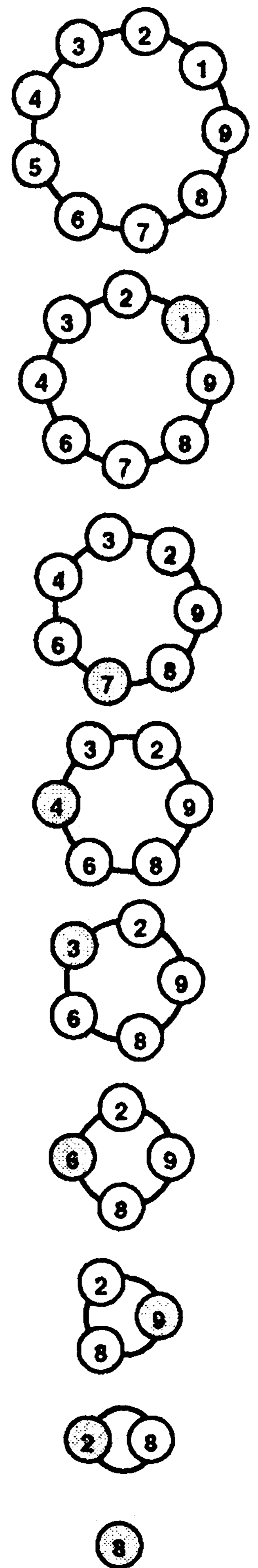
```

Предположим,  $N$  человек решило выбрать главаря. Для этого они встали в круг и стали удалять каждого  $M$ -го человека в определенном направлении отсчета, смыкая ряды после каждого удаления. Задача состоит в определении, кто останется последним (потенциальный лидер с математическими способностями заранее определит выигрышную позицию в круге).

Номер выбираемого главаря является функцией от  $N$  и  $M$ , называемой *функцией Иосифа*. В более общем случае требуется выяснить порядок удаления людей. В примере, показанном на рис. 3.5, если  $N = 9$  и  $M = 5$ , люди удаляются в порядке 5 1 7 4 3 6 9 2, а 8-ой номер становится избранным главарем. Программа 3.9 считывает значения  $N$  и  $M$ , а затем распечатывает эту последовательность.

Для прямой имитации процесса выбора в программе 3.9 используется циклический связный список. Сначала создается список элементов от 1 до  $N$ . Для этого создается циклический список с единственным узлом для участника 1, затем вставляются узлы для участников от 2 до  $N$  с помощью операции, иллюстрируемой на рис. 3.4. Затем в списке отсчитывается  $M - 1$  элемент и удаляется следующий при помощи операции, проиллюстрированной на рис. 3.3. Этот процесс продолжается до тех пор, пока не останется только один узел (который будет указывать на самого себя).

Решето Эратосфена и задача Иосифа хорошо иллюстрируют различие между использованием массивов и связных списков для представления последовательно расположенных наборов объектов. Использование связного списка вместо массива для построения решета Эратосфена скажется на производительности, поскольку эффективность алгоритма



**РИСУНОК 3.5 ПРИМЕР ЗАДАЧИ ИОСИФА**

*Эта диаграмма показывает результат выбора по принципу Иосифа, когда группа людей становится в круг, затем по кругу отсчитывается каждый пятый человек и удаляется из круга, пока не останется один.*

зависит от возможности быстрого доступа к произвольному элементу массива. Использование массива вместо связного списка для решения задачи Иосифа снизит быстродействие, поскольку здесь эффективность алгоритма зависит от возможности быстрого удаления элементов. При выборе структуры данных следует учитывать ее влияние на эффективность алгоритма обработки данных. Это взаимодействие структур данных и алгоритмов занимает центральное место в процессе разработки программ и является постоянной темой данной книги.

В языке C++ указатели служат прямой и удобной реализацией абстрактной концепции связных списков, но важность абстракции не зависит от конкретной реализации. Например, рис. 3.6 демонстрирует использование массивов целых чисел с целью реализации связного списка для задачи Иосифа. Таким образом, можно реализовать связный список с помощью индексов массива вместо указателей. Связные списки применялись задолго до появления конструкций указателей в языках высокого уровня, таких как C++. Даже в современных системах реализации на основе массивов иногда оказываются удобными.

## Упражнения

▷ 3.23 Написать функцию, которая возвращает количество узлов циклического списка для данного указателя одного из узлов списка.

3.24 Написать фрагмент кода, который определяет количество узлов в циклическом списке, находящихся между узлами, на которые ссылаются два данных указателя  $x$  и  $t$ .

3.25 Написать фрагмент кода, который по указателям  $x$  и  $t$  двух непересекающихся связных списков вставляет список, указываемый  $t$ , в список, указываемый  $x$ , в позицию, которая следует после узла  $x$ .

	0	1	2	3	4	5	6	7	8
item	1	2	3	4	5	6	7	8	9
next	1	2	3	4	5	6	7	8	0
5	1	2	3	4	5	6	7	8	9
	1	2	3	5	5	6	7	8	0
1	1	2	3	4	5	6	7	8	9
	1	2	3	5	5	6	7	8	1
7	1	2	3	4	5	6	7	8	9
	1	2	3	5	5	7	7	8	1
4	1	2	3	4	5	6	7	8	9
	1	2	5	5	5	7	7	8	1
3	1	2	3	4	5	6	7	8	9
	1	5	5	5	5	7	7	8	1
6	1	2	3	4	5	6	7	8	9
	1	7	5	5	5	7	7	8	1
9	1	2	3	4	5	6	7	8	9
	1	7	5	5	5	7	7	1	1
2	1	2	3	4	5	6	7	8	9
	1	7	5	5	5	7	7	7	1

РИСУНОК 3.6 ПРЕДСТАВЛЕНИЕ СВЯЗНОГО СПИСКА В ВИДЕ МАССИВА

Эта последовательность отражает связный список для задачи Иосифа (см. рис. 3.5), реализованный с помощью индексов массива вместо указателей. Индекс элемента, следующего в списке за элементом с индексом  $0$ , —  $next[0]$ , и т.д. Сначала (три верхних строки) элемент для участника  $i$  имеет индекс  $i-1$ , и формируется циклический список путем присвоения значений  $i+1$  членам  $next[i]$  для  $i$  от  $0$  до  $8$ , а элементу  $next[8]$  присваивается значение  $0$ . Для имитации процесса выбора Иосифа изменяются ссылки (записи  $next$  массива), но элементы не перемещаются. Каждая пара строк показывает результат перемещения по списку с четырехкратной установкой значений  $x = next[x]$  и последующим удалением пятого элемента (отображаемого в крайнем левом столбце) путем присвоения значения  $next[next[x]]$  указателю  $next[x]$ .

- 3.26 Для данных указателей  $x$  и  $t$  узлов циклического списка написать фрагмент кода, который перемещает узел, следующий после  $t$ , в позицию списка, которая следует после узла  $x$ .
- 3.27 При построении списка программа 3.9 устанавливает двойное количество значений ссылок по сравнению с требуемым, поскольку поддерживает циклический список после вставки каждого узла. Изменить программу таким образом, чтобы циклический список создавался без выполнения этих лишних операций.
- 3.28 Определить время выполнения программы 3.9 как функцию от  $M$  и  $N$ .
- 3.29 Использовать программу 3.9 с целью определения значения функции Иосифа для  $M = 2, 3, 5, 10$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 3.30 Использовать программу 3.9 с целью построения графика зависимости функции Иосифа от  $N$  для  $M = 10$  и  $N$  от 2 до 1000.
- 3.31 Воспроизвести таблицу на рис. 3.6, когда элемент  $i$  занимает в массиве исходную позицию  $N-i$ .
- 3.32 Разработать версию программы 3.9, в которой для реализации связного списка используется массив индексов (см. рис. 3.6).

## 3.4 Обработка простых списков

Вычисления со связными списками заметно отличаются от вычислений с массивами и структурами. При использовании массивов и структур элемент сохраняется в памяти. После этого на него можно ссылаться по имени (или индексу), что во многом подобно хранению информации в картотеке либо адресной книге. При использовании связных списков метод хранения информации усложняет доступ к ней, однако упрощает перераспределение. Работа с данными, организованными в виде связных списков, называется *обработкой списков*.

При использовании массивов возможны ошибки программы, связанные с попыткой доступа к данным вне допустимого диапазона. Для связных списков наиболее часто встречается подобная ошибка — ссылка на неопределенный указатель. Другая распространенная ошибка — использование указателя, который изменен неизвестным образом. Одна из причин возникновения упомянутой проблемы состоит в возможном присутствии нескольких указателей на один и тот же узел, что программист может не осознавать. В программе 3.9 несколько подобных проблем устраняется за счет использования циклического списка, который никогда не бывает пустым. При этом каждая ссылка всегда указывает на однозначно определенный узел. Кроме того, каждая ссылка может интерпретироваться как ссылка на список.

Искусство разработки корректного и эффективного кода для обрабатывающих списки приложений приходит с практикой и терпением. В этом разделе представлены примеры и упражнения, призванные помочь освоить создание кода обработки списков. На протяжении книги будет использоваться множество других примеров, поскольку связные структуры лежат в основе многих эффективных алгоритмов.

Как указывалось в разделе 3.3, для первого и последнего указателей списка применяется ряд различных соглашений. Некоторые из них рассматриваются в этом разделе, хотя мы взяли за правило резервировать термин *связные списки* для описания простейших ситуаций.

**Определение 3.3** *Связный список содержит либо null-ссылки, либо ссылки на узлы, которые содержат элемент и ссылку на связный список.*

Это определение накладывает более строгие ограничения, нежели определение 3.2, но оно ближе к умоглядной модели, используемой при написании кода обработки списков. Вместо исключения всех других соглашений (за счет использования только этого определения) и создания специальных определений, соответствующих каждому соглашению, оставляем оба подхода в силе с учетом, что тип используемого связного списка будет ясен из контекста.

Одна из наиболее распространенных операций со списками — *обход (traverse)*. Это последовательное сканирование элементов списка и выполнение некоторых операций с каждым из них. Например, если  $x$  является указателем на первый узел списка, последний узел имеет null-указатель, а  $visit$  — процедура, которая принимает элемент в качестве аргумента, то обход списка можно реализовать следующим образом:

```
for (link t = x; t != 0; t = t->next) visit(t->item);
```

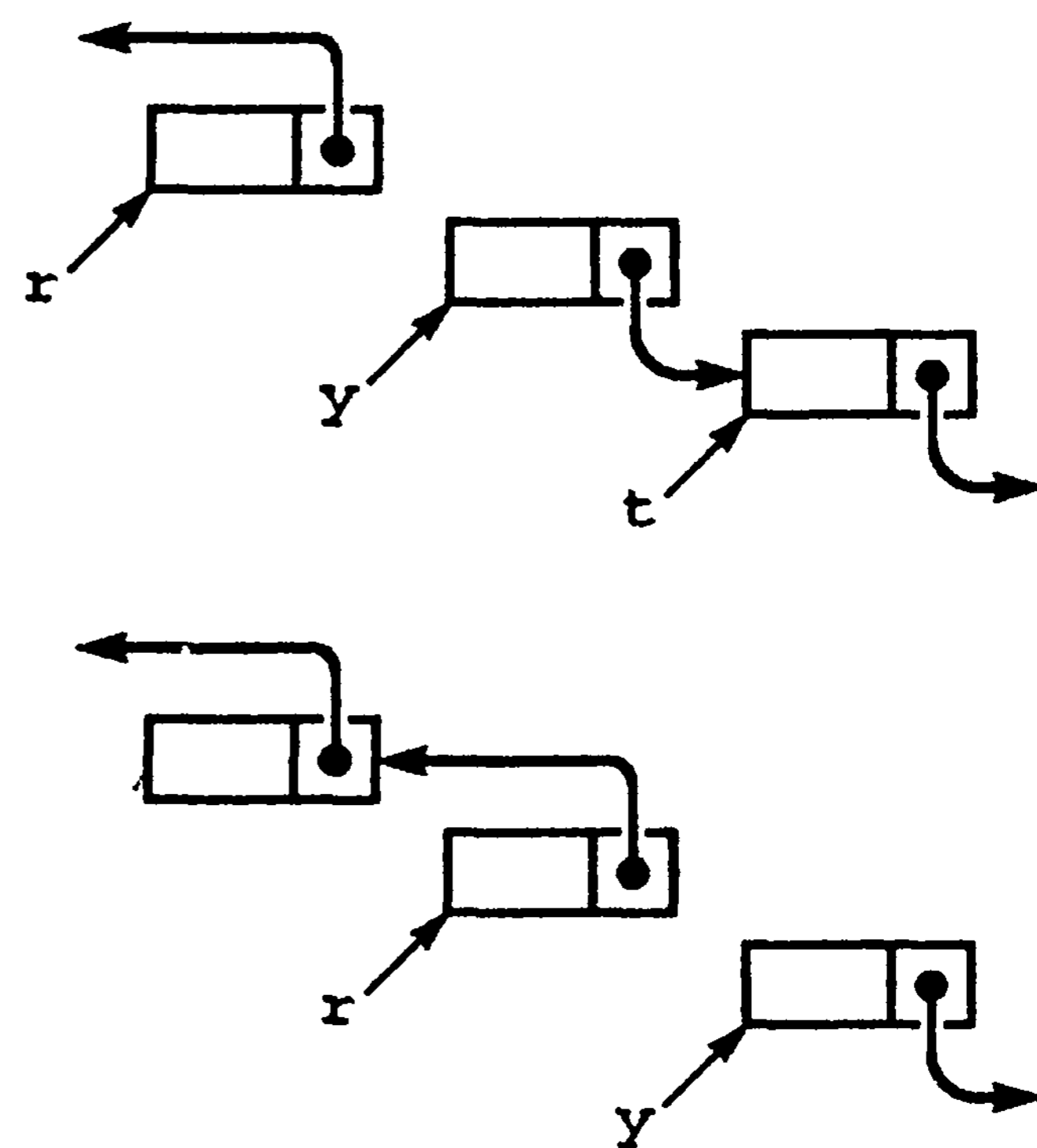
Этот цикл (либо его эквивалентная форма `while`) является универсальным для программ обработки списков, подобно циклу `for (int i = 0; i < N; i++)` для программ обработки массивов.

Программа 3.10 служит реализацией простой задачи обработки списка, состоящей в изменении на обратный порядок следования узлов. Она принимает связный список в качестве аргумента и возвращает связный список, состоящий из тех же узлов, но расположенных в обратном порядке. На рис. 3.7 показано изменение, выполняемое функцией в своем главном цикле для каждого узла. Эта диаграмма упрощает проверку каждого оператора программы на правильность изменения ссылок. Программисты обычно используют подобные диаграммы для осмысления операций обработки списков.

### Программа 3.10 Обращение порядка следования элементов списка

Эта функция обращает порядок следования ссылок в списке, возвращая указатель последнего узла, который затем указывает на предпоследний узел и т.д. При этом для ссылки в первом узле исходного списка устанавливается значение 0 (null-указатель). Для выполнения этой задачи необходимо сохранять ссылки на три последовательных узла списка.

```
link reverse(link x)
{ link t, y = x, r = 0;
  while (y != 0)
    { t = y->next; y->next = r; r = y;
      y = t; }
  return r;
}
```



### РИСУНОК 3.7 ОБРАЩЕНИЕ СПИСКА

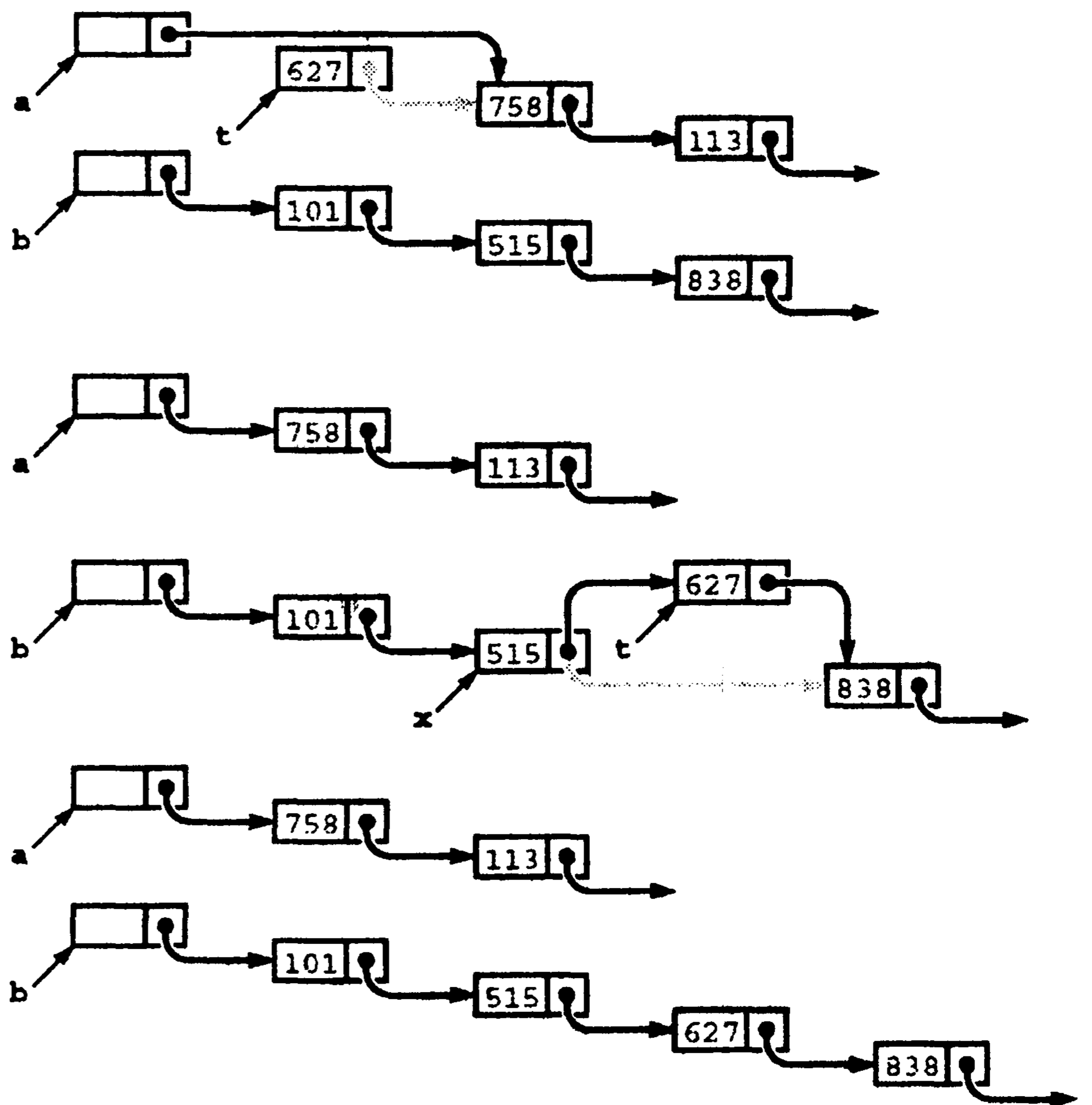
Для обращения порядка следования элементов списка применяется указатель  $r$  на уже обработанную часть списка, и указатель  $y$  на еще не затронутую часть списка. Эта диаграмма демонстрирует изменение указателей каждого узла списка. Указатель узла, следующего после  $y$ , сохраняется в переменной  $t$ , ссылка  $y$  изменяется так, чтобы указывать на  $r$ , после чего  $r$  перемещается в позицию  $y$ , а  $y$  — в позицию  $t$ .

Программа 3.11 служит реализацией другой задачи обработки списков: перераспределение узлов в порядке сортировки их элементов. Она генерирует  $N$  случайных целых чисел, помещает в список в порядке их появления, перераспределяет узлы в порядке сортировки элементов и распечатывает полученную последовательность. Ожидаемое время выполнения программы пропорционально  $N^2$ , поэтому для больших значений  $N$  программа неэффективна, что будет показано в главе 6. Обсуждение темы сортировки также откладывается до главы 6, поскольку в главах с 6-й по 10-ю рассматривается множество методов сортировки. Сейчас ставится цель продемонстрировать пример приложения, выполняющего обработку списков.

Списки в программе 3.11 иллюстрируют еще одно часто используемое соглашение: в начале каждого списка содержится фиктивный узел, называемый *ведущим (head node)*. Поле элемента ведущего узла игнорируется, но ссылка узла сохраняется в качестве указателя узла, содержащего первый элемент списка. В программе используется два списка: один для сбора вводимых случайных чисел в первом цикле, а другой для сбора сортированного вывода во втором цикле. На рис. 3.8 показаны изменения, вносимые программой 3.11 в течение одной итерации главного цикла. Из списка ввода извлекается следующий узел, вычисляется его позиция в списке вывода, и реализуется ссылка на эту позицию.

### РИСУНОК 3.8 СОРТИРОВКА СВЯЗНОГО СПИСКА.

Приведенная диаграмма отражает один шаг преобразования неупорядоченного связного списка (заданного указателем *a*) в упорядоченный связный список (заданный указателем *b*) с использованием сортировки вставками. Сначала берется первый узел неупорядоченного списка и указатель на него сохраняется в *t* (верхняя диаграмма). Затем выполняется поиск в *b* для нахождения первого узла *x*, для которого справедливо условие  $x \rightarrow \text{next} \rightarrow \text{item} > t \rightarrow \text{item}$  (или  $x \rightarrow \text{next} = \text{NULL}$ ) и *t* вставляется в список после *x* (средняя диаграмма). Эти операции уменьшают на один узел размеры списка *a* и увеличивают на один узел размеры списка *b*, сохраняя список *b* упорядоченным (нижняя диаграмма). По прошествии цикла список *a* окажется пустым, а список *b* будет содержать все узлы в упорядоченном виде.



---

**Программа 3.11 Сортировка методом вставки в список**

---

Этот код генерирует  $N$  случайных целых чисел в диапазоне от 0 до 999, строит связный список, в котором на каждый узел приходится по одному числу (первый цикл `for`), затем переставляет узлы, чтобы при обходе списка числа следовали по порядку (второй цикл `for`). Для выполнения сортировки используется два списка — список ввода (несортированный) и список вывода (сортированный). В каждой итерации цикла из ввода удаляется узел и вставляется в определенную позицию списка вывода. Код упрощен благодаря использованию ведущих узлов в каждом списке, содержащих ссылки на первые узлы. В объявлениях ведущих узлов применен конструктор, поэтому их данные инициализируются при создании.

```
node heada(0, 0); link a = &heada, t = a;
for (int i = 0; i < N; i++)
    t = (t->next = new node(rand() % 1000, 0));
node headb(0, 0); link u, x, b = &headb;
for (t = a->next; t != 0; t = u)
{
    u = t->next;
    for (x = b; x->next != 0; x = x->next)
        if (x->next->item > t->item) break;
    t->next = x->next; x->next = t;
}
```

---

Главная причина использования ведущего узла становится понятной, если рассмотреть процесс добавления *первого* узла к сортированному списку. Этот узел имеет наименьший элемент в списке ввода и может находиться в любом месте списка. Существуют три возможности:

- Дублировать цикл `for`, который обнаруживает наименьший элемент и создает список из одного узла таким же образом, как в программе 3.9.
- Перед каждой вставкой узла проверять, не является ли список вывода пустым.
- Использовать фиктивный ведущий узел, ссылка которого указывает на первый узел списка, как в данном примере.

Первому варианту не хватает изящества. Он требует дополнительного кода. То же относится и ко второму варианту.

Использование ведущего узла влечет дополнительные затраты памяти (на дополнительный узел). Во множестве типовых приложений без него можно обойтись. Например, в программе 3.10 присутствуют список ввода (исходный) и список вывода (зарезервированный), но нет необходимости использовать ведущий узел, поскольку все вставки выполняются в начало списка вывода. Будут примеры и других приложений, в которых использование фиктивного узла упрощает код эффективнее, нежели применение `null`-ссылки в *хвосте* списка. Не существует жестких правил принятия решения об использовании фиктивных узлов — выбор зависит от стиля в комбинации с соображениями быстродействия. Хорошие программисты выбирают соглашение, которое более всего упрощает задачу. На протяжении книги встречается несколько подобных компромиссов.

Ряд вариантов соглашений о связных списках приводится в табл. 3.1 для справочных целей. Остальные рассматриваются в упражнениях. Во всех вариантах табл. 3.1 для ссылки на список используется указатель **head** и сохраняется положение, при ко-

тором программа управляет ссылками узлов, используя данный код для различных операций. Распределение памяти под узлы и ее освобождение, а также заполнение узлов информацией одинаково для всех соглашений. Для живучести функций, реализующих те же операции, необходим дополнительный код проверки условий ошибок. Цель таблицы заключается в демонстрации сходства и различия вариантов.

**Таблица 3.1 Соглашения о ведущем и завершающем узлах в связных списках**

Эта таблица представляет реализации базовых операций обработки списков, основанных на пяти часто используемых соглашениях. Подобный код используется в простых приложениях с линейной организацией процедур обработки списков.

**Список циклический, никогда не бывает пустым**

*первая вставка:* `head->next = head;`  
*вставка t после x:* `t->next = x->next; x->next = t;`  
*удаление после x:* `x->next = x->next->next;`  
*цикл обхода:* `t = head;`  
`do { ... t = t->next; } while (t != head);`  
*проверка на наличие лишь одного элемента:* `if (head->next == head)`

**Ведущий указатель, null-указатель завершающего узла**

*инициализация:* `head = 0;`  
*вставка t после x:* `if (x == 0) {head = t; head->next = 0; }  
else { t->next = x->next; x->next = t; }`  
*удаление после x:* `t = x->next; x->next = t->next;`  
*цикл обхода:* `for (t = head; t != 0; t = t->next)`  
*проверка на пустоту:* `if (head == 0)`

**Фиктивный ведущий узел, null-указатель завершающего узла**

*инициализация:* `head = new node;`  
`head->next = 0;`  
*вставка t после x:* `t->next = x->next; x->next = t;`  
*удаление после x:* `t = x->next; x->next = t->next;`  
*цикл обхода:* `for (t = head->next; t != 0; t = t->next)`  
*проверка на пустоту:* `if (head->next == 0)`

**Фиктивные ведущий и завершающий узлы**

*инициализация:* `head = new node;`  
`z = new node;`  
`head->next = z; z->next = z;`  
*вставка t после x:* `t->next = x->next; x->next = t;`  
*удаление после x:* `x->next = x->next->next;`  
*цикл обхода:* `for (t = head->next; t != z; t = t->next)`  
*проверка на пустоту:* `if (head->next == z)`

Другая важная ситуация, в которой иногда удобно использовать ведущий узел, возникает, когда необходимо передать спискам указатели в качестве аргументов функций. Эти аргументы могут изменять список таким же образом, как это выполняется для массивов. Использование ведущего узла позволяет функции принимать или



возвращать пустой список. При отсутствии ведущего узла функция нуждается в механизме информирования вызывающей функции в случае, когда оставляется пустой список. Одно из решений для C++ состоит в передаче указателя на список как ссылочного параметра. Второй механизм предусматривает прием функциями обработки списков указателей на списки, ввода в качестве аргументов и возврат указателей на списки вывода. Он уже задействован в программе 3.10. Этот принцип устраняет необходимость использования ведущих узлов. Более того, он очень удобен для рекурсивной обработки списков, которая часто используется в этой книге (см. раздел 3.1).

### Программа 3.12 Интерфейс обработки списков

---

В этом коде, который можно сохранить в файле интерфейса **list.h**, описаны типы узлов и ссылок, включая операции, выполняемые над ними. Для распределения памяти под узлы списка и ее освобождения объявляются собственные функции. Функция **construct** применена для удобства реализации. Эти описания *позволяют* клиентам использовать узлы и связанные с ними операции без зависимости от подробностей реализации. Как будет показано в главе 4, несколько отличный интерфейс, основанный на классах C++, может *обеспечить* независимость клиентской программы от подробностей реализации.

```
typedef int Item;
struct node { Item item; node *next; };
typedef node *link;
typedef link Node;

void construct(int);
Node newNode(int);
void deleteNode(Node);
void insert(Node, Node);
Node remove(Node);
Node next(Node);
Item item(Node);
```

---

Программа 3.12 объявляет набор функций "черного ящика", которые реализуют базовый список операций. Это позволяет избегать повторения кода и зависимости от деталей реализации. Программа 3.13 реализует выбор Иосифа (см. программу 3.9), преобразованный в клиентскую программу, которая использует этот интерфейс. Идентификация важных операций, используемых в вычислениях, и описание их в интерфейсе обеспечивают гибкость, которая позволяет рассматривать различные конкретные реализации важных операций и проверять их эффективность. В разделе 3.5 рассматривается реализация операций программы 3.12 (см. программу 3.14), но возможны альтернативные решения, не требующие никаких изменений программы 3.13 (см. упражнение 3.51). Эта тема еще будет неоднократно затрагивать в данной книге. Язык C++ включает несколько механизмов, специально предназначенных для упрощения разработки инкапсулированных реализаций; речь об этом пойдет в главе 4.

### Программа 3.13 Организация списка для задачи Иосифа

---

Эта программа решения задачи Иосифа служит примером клиентской программы, использующей примитивы обработки списков, которые объявлены в программе 3.12 и реализованы в программе 3.14.

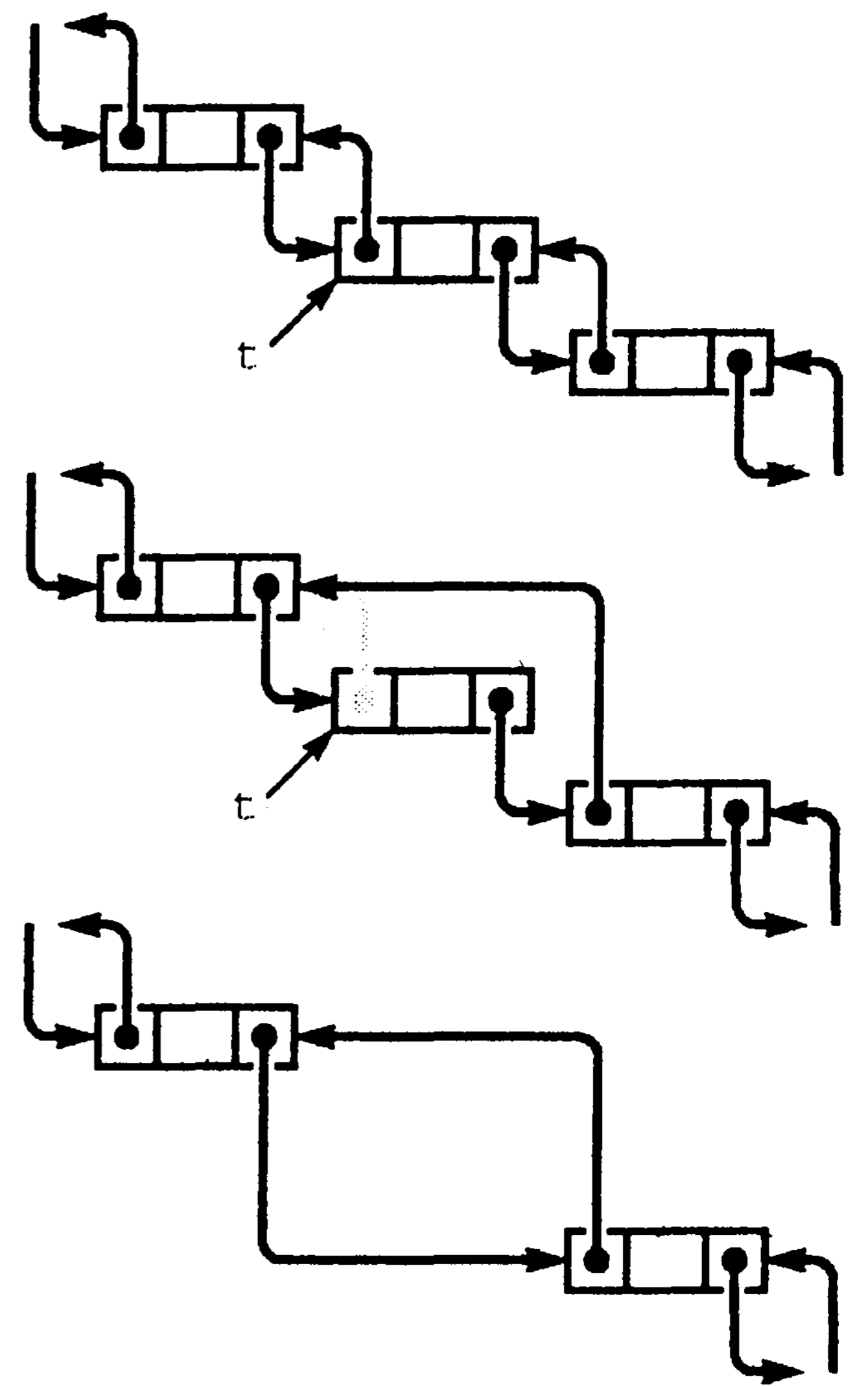
```

#include <iostream.h>
#include <stdlib.h>
#include <list.h>
int main(int argc, char *argv[])
{ int i, N = atoi(argv[1]), M = atoi(argv[2]);
  Node t, x;
  construct(N);
  for (i = 2, x = newNode(1); i <= N; i++)
    { t = newNode(i); insert(x, t); x = t; }
  while (x != next(x))
    {
      for (i = 1; i < M; i++) x = next(x);
      deleteNode(remove(x));
    }
  cout << item(x) << endl;
  return 0;
}

```

Некоторые программисты предпочитают инкапсулировать все операции в низкоуровневых структурах данных, таких как связанные списки, путем описания функций для каждой низкоуровневой операции в интерфейсах, подобных показанному в программе 3.12. Действительно, как будет продемонстрировано в главе 4, классы C++ упрощают это решение. Однако такой дополнительный уровень абстракции иногда скрывает факт использования небольшого количества операций низкого уровня. В данной книге при реализации высокоуровневых интерфейсов низкоуровневые операции обычно описываются непосредственно в связанных структурах, дабы ясно выразить существенные подробности алгоритмов и структур данных. В главе 4 будет рассматриваться множество примеров.

За счет добавления ссылок можно реализовать возможность обратного перемещения по связанному списку. Например, применение *двухсвязного списка* (*double linked list*) позволяет поддерживать операцию "найти элемент, предшествующий данному". В таком списке каждый узел содержит две ссылки: одна (*prev*) указывает на предыдущий элемент, а другая (*next*) — на следующий. Наличие фиктивных узлов либо цикличность двухсвязного списка позволяет обеспечить эквивалентность выражений  $x \rightarrow next \rightarrow prev$  и  $x \rightarrow prev \rightarrow next$  для каждого узла. На рис. 3.9 и 3.10 показаны основные действия со ссылками, необходимые для реализации операций *remove* (удалить), *insert after* (вставить после) и *insert before* (вставить перед) в двухсвязных списках.



**РИСУНОК 3.9 УДАЛЕНИЕ В ДВУХСВЯЗНОМ СПИСКЕ**

В двухсвязном списке указатель узла предоставляет достаточно информации для удаления узла, что видно из диаграммы. Для данного *t* указателю  $t \rightarrow next \rightarrow prev$  присваивается значение  $t \rightarrow prev$  (средняя диаграмма), а указателю  $t \rightarrow prev \rightarrow next$  — значение  $t \rightarrow next$  (нижняя диаграмма).

Обратите внимание, что для операции удаления не требуется дополнительной информации об узле, предшествующем данному (либо следующим за ним) в списке, как это имеет место для односвязных списков — эта информация содержится в самом узле.

Действительно, главная особенность двухсвязных списков состоит в возможности удаления узла, когда ссылка на него является единственной информацией об узле. Типичны случаи, когда ссылка передается при вызове функции в качестве аргумента, а также если узел имеет другие ссылки и сам является частью другой структуры данных. Предоставление этой дополнительной возможности влечет удвоение пространства, отводимого под ссылки в каждом узле, а также количества операций со ссылками на каждую базовую операцию. Поэтому двухсвязные списки обычно не используются, если этого не требуют условия. Рассмотрение подробных реализаций отложим до обзора нескольких особых случаев, где в этом возникнет необходимость — например, в разделе 9.5

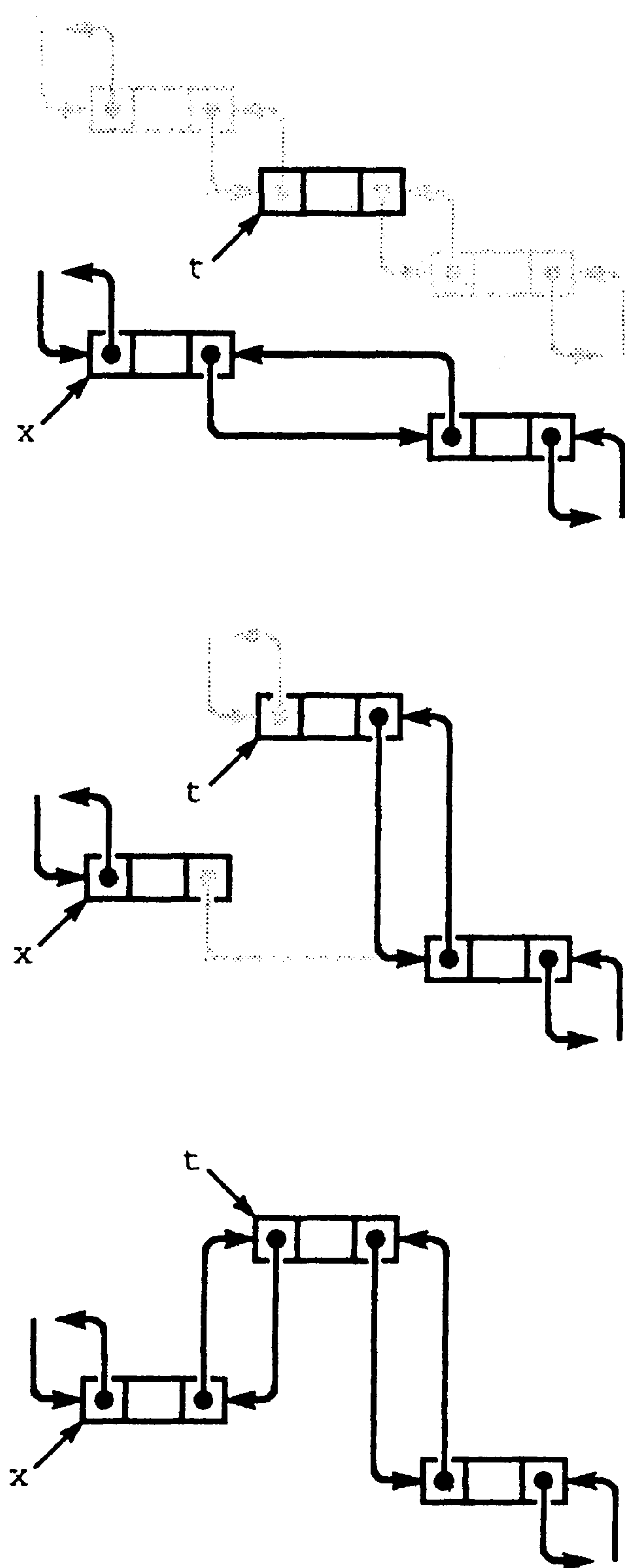
Связные списки используются в материале книги, во-первых, для основных ADT-реализаций (см. главу 4), во-вторых, в качестве компонентов более сложных структур данных. Для многих программистов связные списки являются первым знакомством с абстрактными структурами данных, которыми разработчик может непосредственно управлять. Они образуют важное средство разработки высокоуровневых абстрактных структур данных, необходимых для решения множества задач, в чем будет возможность убедиться.

## Упражнения

▷ 3.33 Создать функцию, которая перемещает наибольший элемент данного списка так, чтобы он стал последним узлом.

3.34 Создать функцию, которая перемещает наименьший элемент данного списка так, чтобы он стал первым узлом.

3.35 Создать функцию, которая перераспределяет связный список так, чтобы узлы в четных позициях следовали после узлов в нечетных позициях, сохраняя порядок сортировки четных и нечетных узлов.



**РИСУНОК 3.10 ВСТАВКА В ДВУХСВЯЗНОМ СПИСКЕ**

Для вставки узла в двухсвязный список необходимо установить четыре указателя. Можно вставить новый узел после данного узла (как показано на диаграмме) либо перед ним. Для вставки узла  $t$  после узла  $x$  указателю  $t \rightarrow \text{next}$  присваивается значение  $x \rightarrow \text{next}$ , а указателю  $x \rightarrow \text{next} \rightarrow \text{prev}$  — значение  $t$  (средняя диаграмма). Затем указателю  $x \rightarrow \text{next}$  присваивается значение  $t$ , а указателю  $t \rightarrow \text{prev}$  — значение  $x$  (нижняя диаграмма).

- 3.36 Реализовать фрагмент кода для связного списка, меняющий местами узлы, которые следуют после узлов, указываемых ссылками **t** и **u**.
- 3.37 Создать функцию, которая принимает ссылку на список в качестве аргумента и возвращает ссылку на копию списка (новый список, содержащий те же элементы в том же порядке).
- 3.38 Создать функцию, принимающую два аргумента — ссылку на список и функцию, принимающую список в качестве аргумента — и удаляет все элементы данного списка, для которых функция возвращает ненулевое значение.
- 3.39 Выполнить упражнение 3.38, но создать копии узлов, которые проходят проверку и возвращают ссылку на список, содержащий эти узлы, в порядке их следования в исходном списке.
- 3.40 Реализовать версию программы 3.10, в которой используется фиктивный узел.
- 3.41 Реализовать версию программы 3.11, в которой не используются фиктивные узлы.
- 3.42 Реализовать версию программы 3.9, в которой используется фиктивный узел.
- 3.43 Реализовать функцию, которая меняет местами два данных узла в двухсвязном списке.
- 3.44 Создать запись табл. 3.1 для списка, который никогда не бывает пустым. На этот список ссылается указатель первого узла, а последний узел содержит указатель на самого себя.
- 3.45 Создать запись табл. 3.1 для циклического списка, имеющего фиктивный узел, который служит ведущим и завершающим узлом.

## 3.5 Распределение памяти под списки

Преимущество связных списков перед массивами состоит в том, что связные списки эффективно изменяют размеры на протяжении времени жизни. В частности, обязательно заранее знать максимальный размер списка. Одно из важных практических следствий этого свойства состоит в возможности иметь несколько структур данных, обладающих общим пространством, не уделяя особого внимания их относительному размеру в любой момент времени.

Интерес представляет реализация оператора **new**. Например, при удалении узла из списка задача сводится к перераспределению ссылок таким образом, чтобы узел более не был привязан к списку. А как поступает система с пространством, которое этот узел занимал? Как система утилизирует пространство, чтобы всегда иметь возможность выделять его под новый узел в операторе **new**? За этими вопросами стоят механизмы, которые служат еще одним примером эффективности элементарной обработки списков.

Оператор **delete** дополняет оператор **new**. Когда блок выделенной памяти более не используется, вызывается оператор **delete** для информирования системы о том, что блок доступен для дальнейшего использования. *Динамическое распределение памяти* (*dynamic memory allocation*) — это процесс управления памятью и ответных действий на вызов операторов **new** и **delete** из клиентских программ. При вызове оператора **new** непосредственно из приложений, таких как программы 3.9 или 3.11, запрашиваются

блоки памяти одинакового размера. Это типичный случай. Альтернативный метод отслеживания памяти, доступной для распределения, напрашивается сам: достаточно использовать связный список! Все узлы, которые не входят ни в один используемый список, можно совместно содержать в единственном связном списке. Этот список называется *свободным (free list)*. Когда необходимо выделить пространство под узел, оно извлекается за счет *удаления* из свободного списка. При удалении узла из какого-либо списка, он *вставляется* в свободный список.

Программа 3.14 является реализацией интерфейса, описанного в программе 3.12, включая функции распределения памяти. При совместной компиляции с программой 3.13 она дает такой же результат, что и прямая реализация, с которой мы начали в программе 3.9. Содержание свободного списка для узлов фиксированного размера — тривиальная задача при наличии базовых операций вставки и удаления узлов из списка.

### Программа 3.14 Реализация интерфейса обработки списков

Эта программа реализует функции, объявленные в программе 3.12, а также иллюстрирует стандартное распределение памяти под узлы фиксированного размера. Создается свободный список, который инициализируется в соответствии с максимальным количеством узлов, используемых программой. Все узлы взаимосвязаны. Когда клиентская программа выделяет память для узла, он удаляется из свободного списка. Когда клиентская программа удаляет узел, он привязывается к свободному списку.

По соглашению клиентские программы не ссылаются на узлы списков за исключением случаев объявления переменных типа **Node** и использования их в качестве аргументов функций, описанных в интерфейсе. При этом возвращаемые клиентским программам узлы имеют ссылки на самих себя. Эти соглашения служат средством защиты от ссылок на неопределенные указатели и, в какой-то мере, гарантируют, что клиент использует интерфейс должным образом. В языке C++ эти соглашения реализуются путем использования классов совместно с конструкторами (см. главу 4).

```
#include <stdlib.h>
#include "list.h"
link freelist;
void construct(int N)
{
    freelist = new node[N+1];
    for (int i = 0; i < N; i++)
        freelist[i].next = &freelist[i+1];
    freelist[N].next = 0;
}
link newNode(int i)
{ link x = remove(freelist);
  x->item = i; x->next = x;
  return x;
}
void deleteNode(link x)
{ insert(freelist, x); }
void insert(link x, link t)
{ t->next = x->next; x->next = t; }
link remove(link x)
{ link t = x->next; x->next = t->next; return t; }
link next(link x)
{ return x->next; }
Item item(link x)
{ return x->item; }
```

Рисунок 3.11 иллюстрирует разрастание свободного списка по мере удаления узлов в программе 3.13. Для простоты подразумевается реализация связного списка (без ведущего узла), основанная на индексах массива.

Реализация обобщенного распределителя памяти в среде C++ намного сложнее, чем подразумевают рассмотренные простые примеры, а реализация оператора `new` в стандартной библиотеке явно не настолько проста, как показано в программе 3.14. Одно из основных различий состоит в том, что функции `new` приходится обрабатывать запросы распределения области хранения для узлов различного размера — от крохотных до огромных. Для этой цели разработано несколько хитроумных алгоритмов. Другой подход, используемый некоторыми современными системами, состоит в освобождении пользователя от необходимости явно удалять узлы за счет алгоритмов *сборки мусора (garbage-collection)*. Эти алгоритмы автоматически удаляют все узлы, на которые не указывает ни одна ссылка. В этой связи также разработано несколько нетривиальных алгоритмов управления областью хранения. Они не будут рассматриваться подробно, поскольку их характеристики быстродействия зависят от свойств определенных компьютеров.

Программы, использующие специфические сведения о проблеме, часто эффективнее программ общего назначения, решающих те же задачи. Распределение памяти — не исключение из этого правила. Алгоритм обработки запросов областей хранения различного размера не может "знать", что запросы всегда будут относиться к блокам фиксированного размера, и поэтому не может использовать этот факт. Парадоксально, но вторая причина отказа от функций библиотек общего назначения заключается в том, что это делает программу более переносимой — можно застраховаться от неожиданных изменений быстродействия при смене библиотеки либо перемещения в другую систему. Многие программисты считают, что использование простого распределителя памяти, вроде продемонстрированного в программе 3.14, — удачный способ разработки эффективных и переносимых программ, использующих связные списки. Этот подход задействован в ряде алгоритмов, которые будут исследоваться в данной книге. В них применяются подобные виды запросов к системе управления памятью. В остальной части книги для распределения памяти применяются стандартные функции C++ `new` и `delete`.

	0	1	2	3	4	5	6	7	8
item	1	2	3	4	5	6	7	8	9
next	1	2	3	4	5	6	7	8	0
4	1	2	3	4	5	6	7	8	9
	1	2	3	5	6	7	8	0	
0	1	2	3	4	5	6	7	8	9
	4	2	3	5	6	7	8	1	
6	1	2	3	4	5	6	7	8	9
	4	2	3	5	7	0	8	1	
3	1	2	3	4	5	6	7	8	9
	4	2	5	6	7	0	8	1	
2	1	2	3	4	5	6	7	8	9
	4	5	3	6	7	0	8	1	
5	1	2	3	4	5	6	7	8	9
	4	7	3	6	2	0	8	1	
8	1	2	3	4	5	6	7	8	9
	4	7	3	6	2	0	1	5	
1	1	2	3	4	5	6	7	8	9
	4	8	3	6	2	0	7	5	

**РИСУНОК 3.11 ПРЕДСТАВЛЕНИЕ СВЯЗНОГО СПИСКА И СВОБОДНОГО СПИСКА В ВИДЕ МАССИВОВ**

Эта версия рис. 3.6 демонстрирует результат поддержки свободного списка с узлами, удаленными из циклического списка. Слева отображен индекс первого узла свободного списка. В конце процесса свободный список представляет собой связный список, содержащий все удаленные элементы.

Проследивая ссылки, начиная с 1, можно наблюдать следующий ряд элементов: 2 9 6 3 4 7 1 5. Они следуют в порядке, обратном по отношению тому, в котором элементы удалялись.

## Упражнения

- 3.46 Написать программу, которая удаляет все узлы связного списка (вызывает операцию **delete** с указателем).
- 3.47 Написать программу, которая удаляет узлы связного списка, находящиеся в позициях с номерами, кратными 5.
- 3.48 Написать программу, которая удаляет узлы в четных позициях связного списка.
- 3.49 Реализовать интерфейс программы 3.12 с помощью прямого использования операций **new** и **delete** в функциях **newNode** и **deleteNode** соответственно.
- 3.50 Эмпирически сравнить времена выполнения функций распределения памяти из программы 3.14 с операторами **new** и **delete** (см. упражнение 3.49) для программы 3.13 при  $M = 2$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 3.51 Реализовать интерфейс программы 3.12, используя индексы массива (при отсутствии ведущего узла) вместо указателей таким образом, чтобы рис. 3.11 иллюстрировал операции программы.
- 3.52 Предположим, что имеется набор узлов без null-указателей (каждый узел указывает на самого себя либо на другой узел набора). Доказать, что если следовать ссылкам, начиная с любого узла, получится цикл.
- 3.53 При соблюдении условий упражнения 3.52 записать фрагмент кода, отыскивающий по данному указателю узла количество различных узлов, которые будут достигнуты, если следовать ссылкам от данного узла. Модификация любых узлов не допускается. Не использовать более некоторого постоянного объема дополнительного пространства памяти.
- 3.54 При соблюдении условий упражнения 3.53 записать функцию, которая определяет, окажутся ли в одном цикле две данных ссылки, если их отслеживать.

## 3.6 Строки

В языке C термин "*строка*" (*string*) обозначает массив символов переменной длины, определяемый начальной точкой и символом завершения строки. Язык C++ наследует эту структуру данных из C. Кроме того, строки в качестве высокоуровневой абстракции включены в стандартную библиотеку. В этом разделе приводятся несколько примеров строк в стиле C. Ценность строк в качестве низкоуровневых структур данных обусловлена двумя главными причинами. Во-первых, многие вычислительные приложения предусматривают обработку текстовых данных, которые могут представляться непосредственно строками. Во-вторых, многие вычислительные системы предоставляют прямой и эффективный доступ к *байтам* памяти, которые в точности соответствуют символам строк. Таким образом, в подавляющем большинстве случаев абстракция строк связывает потребности приложения с возможностями компьютера.

Абстрактное понятие последовательности символов, завершаемых символом конца строки, может быть реализовано множеством способов. Например, можно использовать связный список, но в этом случае для каждого символа потребуется содержать по одному указателю. Язык C++ наследует из C эффективную реализацию на основе массивов, которая рассматривается в этом разделе, а также предоставляет более обоб-

щенную реализацию из стандартной библиотеки. Остальные реализации исследуются в главе 4.

Различие между строкой и массивом символов связано с понятием *длины*. В обоих случаях представляется непрерывная область памяти, но длина массива устанавливается в момент его создания, а длина строки может изменяться в процессе выполнения программы. Это различие влечет интересные последствия, которые мы вкратце обсудим.

Для строки необходимо зарезервировать память либо во время компиляции, путем объявления массива символов с фиксированной длиной, либо во время выполнения, через вызов функции `new[]`. После выделения памяти массиву его можно заполнять символами с начала и до символа завершения строки. Без символа завершения строка представляет собой обыкновенный массив символов. Символ завершения строки позволяет применять более высокий уровень абстракции, позволяющий рассматривать только часть массива (от начала до символа завершения) как содержащую значимую информацию. Символ завершения имеет значение 0. Его также обозначают так: `'\0'`.

Например, чтобы найти длину строки, можно подсчитать количество символов от начала и до символа завершения. В табл. 3.2 перечислены простые операции, которые часто выполняются со строками. Все они предусматривают сканирование строк с начала и до конца. Многие из этих функций содержатся в библиотеках, объявленных в файле `<string.h>`. Однако программисты для простых приложений часто используют слегка измененные версии в линейном коде. Для живучести функций, реализующих те же операции, необходим дополнительный код проверки условий ошибок. Этот код представлен здесь не только с целью демонстрации его простоты, но и для наглядной демонстрации характеристик производительности.

**Таблица 3.2 Элементарные операции со строками**

В этой таблице представлены основные операции обработки строк, использующие два различных примитива языка C++. Применение указателей делает код более компактным, а использование индексированного массива служит более естественным методом выражения алгоритмов и делает код более простым для понимания. Операция объединения для версии с использованием указателей совпадает с операцией для версии, в которой применяется индексированный массив. Операция предварительного сравнения для версии с указателями получается таким же образом, как и для версии с индексированным массивом, поэтому она опущена. Время выполнения всех реализаций пропорционально длине строки.

#### **Версии с индексированным массивом**

##### *Вычисление длины строки (strlen(a))*

```
for (i = 0; a[i] != 0; i++) ; return i ;
```

##### *Копирование (strcpy(a, b))*

```
for (i = 0; (a[i] = b[i]) != 0; i++) ;
```

##### *Сравнение (strcmp(a, b))*

```
for (i = 0; a[i] == b[i] != 0; i++) ;
    if (a[i] == 0) return 0;
return a[i] - b[i];
```

##### *Сравнение (префикс) (strncmp(a, b, n))*

```
for (i = 0; i < n && a[i] != 0; i++)
    if (a[i] != b[i]) return a[i] - b[i];
return 0;
```



*Присоединение* (`strcat(a, b)`)

```
strcpy(a+strlen(a), b)
```

**Эквивалентные версии с указателями**

*Вычисление длины строки* (`strlen(a)`)

```
b = a; while (*b++) ; return b-a-1;
```

*Копирование* (`strcpy(a, b)`)

```
while (*a++ = *b++) ;
```

*Сравнение* (`strcmp(a, b)`)

```
while (*a++ = *b++)
    if (*(a-1) == 0) return 0;
return *(a-1) - *(b-1);
```

Одной из наиболее важных является операция *сравнения* (*compare*). Она указывает, какая из двух строк должна первой значиться в словаре (*dictionary*). Для простоты изложения предполагается, что словарь идеализирован (поскольку реальные правила для строк, содержащих знаки пунктуации, буквы нижнего и верхнего регистра, числа и пр. довольно сложны), а строки сравниваются посимвольно от начала и до конца. Такой порядок называется лексикографическим. Кроме того, функция сравнения используется для определения эквивалентности строк. По соглашению функция сравнения возвращает отрицательное число, если строка первого аргумента находится в словаре перед второй строкой, положительное число в противном случае и ноль, когда строки идентичны. Важно отметить, что выполнение проверки равенства *не то же*, что определение, равны ли два *указателя* строк — если два указателя строк равны, то же относится к указываемым строкам (это *одна и та же* строка), но возможны различные указатели строк, которые ссылаются на равные строки (идентичные последовательности символов). Хранение информации в строках с последующей обработкой либо доступом к ней путем сравнения строк, применяется во множестве приложений. Поэтому операция сравнения имеет особое значение. Характерный пример содержится в разделе 3.7, а также во многих других местах книги.

Программа 3.15 является реализацией простой задачи обработки строк. Она распечатывает позиции, где короткая образцовая строка содержится внутри длинной строки текста. Для этой задачи разработано несколько сложных алгоритмов, но данный простой алгоритм иллюстрирует несколько соглашений, используемых при обработке строк в языке C++.

Обработка строк служит убедительным примером необходимости сведений о быстродействии библиотечных функций. Дело в том, что время выполнения библиотечных функций может превышать интуитивно ожидаемый результат. Например, *определение длины строки занимает время, пропорциональное ее длине*. Игнорирование этого факта может привести к серьезным проблемам, связанным с быстродействием. Например, после краткого обзора библиотеки можно реализовать соответствие образцу из программы 3.15 следующим образом:

```
for (i = 0; i < strlen(a); i++)
    if (strncmp(&a[i], p, strlen(p)) == 0)
        cout << i << " ";
```

К сожалению, время выполнения этого фрагмента кода пропорционально, как минимум, *квадрату* длины *a* независимо от кода тела цикла, поскольку для определения длины строки *a* каждый раз выполняется ее полный перебор. Эти затраты времени велики и даже неприемлемы: выполнение программы для проверки наличия определенного слова в данной книге (содержащей более миллиона символов) потребует триллионов операций. Подобные проблемы трудно обнаружить, поскольку программа может хорошо работать с небольшими строками на этапе отладки, но сильно замедляться, если не полностью затормаживаться при решении реальных задач. Более того, этих проблем можно избегать только будучи осведомленным о них!

### Программа 3.15 Поиск строки

Эта программа обнаруживает все вхождения введенного из командной строки слова в строке текста (предположительно намного большей длины). Строка текста объявляется в виде массива символов фиксированной длины (можно также использовать оператор `new[]`, как в программе 3.6). Чтение строки выполняется из стандартного ввода с помощью функции `cin.get()`. Память для слова (аргумента командной строки) выделяется системой перед вызовом программы, а указатель строки содержится в элементе `argv[1]`. Для каждой начальной позиции *i* в строке *a* предпринимается попытка сравнить подстроку, которая начинается с этой позиции, со словом *p*. Эквивалентность проверяется символ за символом. Как только достигается конец слова *p*, печатается начальная позиция *i* вхождения этого слова в текст.

```
#include <iostream.h>
#include <string.h>
static const int N = 10000;
int main(int argc, char *argv[])
{ int i; char t;
  char a[N], *p = argv[1];
  for (i = 0; i < N-1; a[i] = t, i++)
    if (!cin.get(t)) break;
  a[i] = 0;
  for (i = 0; a[i] != 0; i++)
    { int j;
      for (j = 0; p[j] != 0; j++)
        if (a[i+j] != p[j]) break;
      if (p[j] == 0) cout << i << " ";
    }
  cout << endl;
}
```

Этот вид ошибок называется *потерей быстродействия (performance bug)*, поскольку код может быть корректным, но его выполнение окажется не настолько эффективным, как ожидалось. Прежде чем приступить к изучению эффективных алгоритмов, необходимо надежно устранить потери быстродействия подобного типа. Стандартные библиотеки обладают многими достоинствами, однако следует учитывать потенциальные недостатки их использования для реализации простых функций рассмотренного типа.

Одна из важных концепций, к которой мы время от времени возвращаемся, гласит, что различные реализации одного и того же абстрактного понятия могут иметь широкие расхождения в характеристиках производительности. Например, класс `string` стандартной библиотеки C++ постоянно отслеживает длину строки и может возвращать это значение через равные промежутки времени, но остальные операции вы-

полняются медленнее. Для различных приложений могут оказаться приемлемыми неодинаковые реализации.

Довольно часто библиотечные функции не могут гарантировать наилучшее быстроедействие для всех приложений. Даже если производительность библиотечной функции хорошо задокументирована (как в случае `strlen`), нет уверенности, что некоторые будущие реализации не повлекут изменений, которые отрицательно повлияют на быстроедействие программ. Это соображение имеет большое значение для разработки алгоритмов и структур данных, потому его следует постоянно учитывать. Остальные примеры и дальнейшие вариации рассматриваются в главе 4.

По сути, строки являются указателями на символы. В некоторых случаях рассмотренный подход позволяет создавать компактный код для функций обработки строк. Например, чтобы скопировать одну строку в другую, можно написать:

```
while (*a++ = *b++) ;
```

вместо

```
for (i = 0; a[i] != 0; i++) a[i] = b[i];
```

либо третьего варианта из табл. 3.2. Оба способа ссылки на строки эквиваленты, но получаемый код может иметь различные характеристики производительности на разных компьютерах. Обычно версия с массивами используется для ясности, а версия с указателями — с целью снижения размеров кода. Подробные исследования для поиска наилучшего решения оставляем для определенных фрагментов часто используемого кода в некоторых приложениях.

Распределение памяти для строк сложнее, чем для связных списков, поскольку строки имеют различный размер. Действительно, совершенно обобщенный механизм резервирования пространства для строк представляет собой ни что иное, как предоставляемые системой функции `new[]` и `delete[]`. Как указывалось в разделе 3.6, для решения этой задачи разработаны различные алгоритмы. Их характеристики производительности зависят от системы и компьютера. Часто распределение памяти при работе со строками является не такой сложной проблемой, как это может показаться, поскольку используются *указатели* на строки, а не сами символы. Действительно, *обычно мы не* предполагаем, что все строки занимают индивидуально выделенные блоки памяти. Мы склонны предполагать, что каждая строка занимает область памяти с неопределенным адресом, но достаточно большую, чтобы вмещать строку и ее символ завершения. Следует очень тщательно обеспечивать выделение памяти при выполнении операций создания либо удлинения строк. В качестве примера в разделе 3.7 приводится программа, которая читает строки и управляет ими.

## Упражнения

- ▷ 3.55 Написать программу, которая принимает строку в качестве аргумента и печатывает таблицу, которая отображает встречаемые в строке символы с частотой появления каждого из них.
- ▷ 3.56 Написать программу, которая определяет, является ли данная строка палиндромом (одинаково читается в прямом и обратном направлениях), если игнори-

ровать пробелы. Например, программа должна давать положительный ответ для строки "if i had a hifi".

**3.57** Предположим, что для строк индивидуально выделена память. Создать версии функций `strcpy` и `strcat`, которые выделяют память и возвращают указатель на новую результирующую строку.

**3.58** Написать программу, которая принимает строку в качестве аргумента и читает ряд слов (последовательностей символов, разделенных пробелами) из стандартного ввода, распечатывая те из них, которые входят как подстроки в строку аргумента.

**3.59** Написать программу, которая заменяет в данной строке подстроки, состоящие из нескольких пробелов, одним пробелом.

**3.60** Реализовать версию программы 3.15, в которой будут использоваться указатели.

- **3.61** Написать эффективную программу, которая вычисляет длину самой большой последовательности пробелов в данной строке, анализируя как можно меньшее количество символов. *Подсказка:* с возрастанием длины последовательности пробелов программа должна выполняться быстрее.

## 3.7 Составные структуры данных

Массивы, связанные списки и строки обеспечивают простые методы последовательной организации данных. Они создают первый уровень абстракции, который можно использовать для группировки объектов методами, пригодными для их эффективной обработки. Иерархию подобных абстракций можно использовать для построения более сложных структур. Возможны массивы массивов, массивы списков, массивы строк и т.д. В этом разделе представлены примеры подобных структур.

Подобно тому, как одномерные массивы соответствуют векторам, *двумерные* массивы, с двумя индексами, соответствуют *матрицам* и широко используются в математических расчетах. Например, следующий код можно применить для перемножения матриц **a** и **b** с помещением результата в третью матрицу **c**.

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        c[i][j] = 0.0;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            c[i][j] += a[i][k]*b[k][j];
```

Математические расчеты часто естественно выражаются многомерными массивами.

Помимо математических применений привычный способ структурирования информации состоит в использовании таблиц чисел, организованных в виде строк и столбцов. В таблице оценок можно выделить по одной строке для каждого студента и по одному столбцу для каждого предмета. Такая таблица будет представлять двумерный массив с одним индексом для строк и вторым — для столбцов. Если студентов 100, а предметов 10, можно объявить массив в виде `grades[100][10]`, а затем ссылаться

на оценку  $i$ -го студента по  $j$ -тому предмету следующим образом: `grade[i][j]`. Для вычисления средней оценки по предмету необходимо сложить элементы соответствующего столбца и разделить сумму на количество строк. Чтобы вычислить среднюю оценку определенного студента, нужно сложить элементы строки и разделить сумму на количество столбцов и т.п. Двумерные массивы широко используются в приложениях подобного типа. В программе часто целесообразно использовать более двух измерений. Например, в таблице оценок можно использовать третий индекс для учета всех таблиц на протяжении учебного года.

Двумерные массивы — вид удобной записи, поскольку числа хранятся в памяти компьютера, которая, по сути, является одномерным массивом. Во многих языках программирования двумерные массивы хранятся в порядке *старшинства строк* в одномерных массивах. Так в массиве `a[M][N]` первые  $N$  позиций будут заняты первой строкой (элементы от `a[0][0]` до `a[0][N-1]`), вторые  $N$  позиций — второй строкой (элементы от `a[1][0]` до `a[1][N-1]`) и т.д. При организации хранения в порядке старшинства строк последняя строка кода перемножения матриц из предыдущего абзаца в точности эквивалентна выражению:

$$c[N*i+j] = a[N*i+k]*b[N*k+j]$$

Обобщение той же схемы применимо для массивов с большим количеством измерений. В языке C++ многомерные массивы могут реализовываться более общим методом: их можно описывать в виде составных структур данных (массивов массивов). Это обеспечивает гибкость, например, содержания массивов массивов, которые имеют различный размер.

В программе 3.6 представлен метод динамического выделения памяти массивам, который позволяет использовать программы для различных задач без повторной компиляции. Воспользуемся подобным методом для многомерных массивов. Как выделять память многомерным массивам, размер которых неизвестен на этапе компиляции? Другими словами, необходима возможность ссылаться в программе на элемент массива, такой как `a[i][j]`, но не объявлять массив типа `int a[M][N]` (например), поскольку значения  $M$  и  $N$  неизвестны. При организации хранения в порядке старшинства строк выражение вида

```
int* a = malloc(M*N*sizeof(int));
```

распределяет память под массив целых чисел размером  $M \times N$ , но это решение приемлемо не для всех случаев. Например, когда массив передается в функцию, во время компиляции можно не указывать только его первое измерение. Программа 3.16 демонстрирует более эффективное решение для двумерных массивов, основанных на описании "массивы массивов".

### Программа 3.16 Распределение памяти под двумерный массив

Эта функция динамически выделяет память двумерному массиву как массиву массивов. Сначала выделяется пространство массиву указателей, а затем — каждой строке. В этой функции выражение

```
int **a = malloc2d(M, N);
```

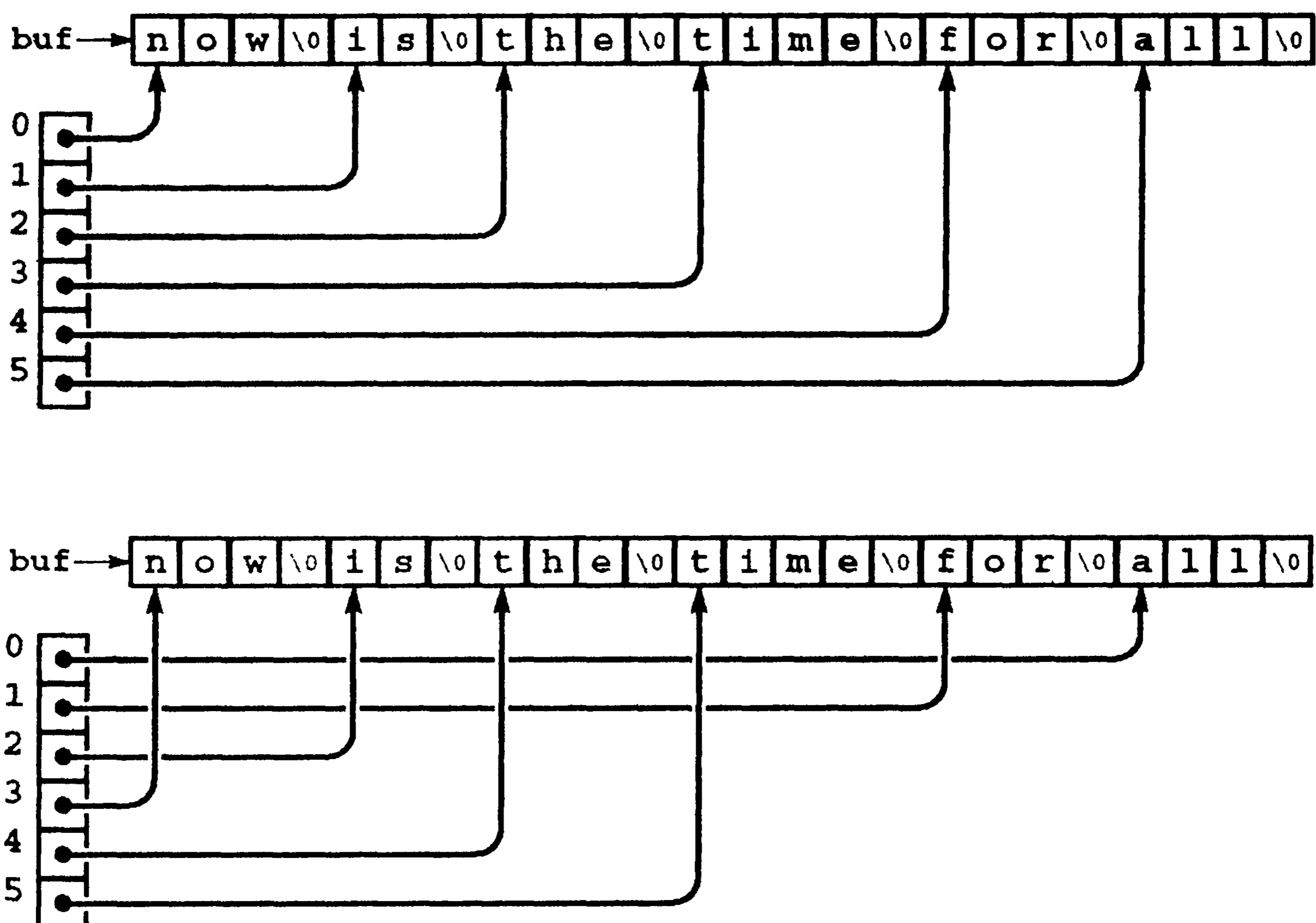
выделяет память массиву целых чисел размером  $M \times N$ .

```

int **malloc2d(int r, int c)
{ int **t = new int*[r];
  for (int i = 0; i < r; i++)
    t[i] = new int[c];
  return t;
}

```

Программа 3.17 демонстрирует использование подобной составной структуры: массива строк. На первый взгляд, поскольку абстрактное понятие строки относится к массиву символов, массивы строк следовало бы представлять в виде массивов, состоящих из массивов. Однако конкретным представлением строки является *указатель* на начало массива символов, поэтому массив строк также может быть массивом указателей. Как показано на рис. 3.12, эффекта перераспределения строк можно достичь простым изменением расположения указателей массива. В программе 3.17 используется библиотечная функция `qsort`. Реализация подобных функций рассматривается в главах 6–9 вообще и в главе 7 — особенно. Этот пример иллюстрирует типовой сценарий обработки строк: символы считываются в большой одномерный массив, указатели сохраняются в отдельных строках (ограниченных символами завершения), а затем осуществляется управление указателями.



**РИСУНОК 3.12 СОРТИРОВКА СТРОК**

При обработке строк обычно используются указатели буфера, который содержит строки (верхняя диаграмма), поскольку указателями легче управлять, чем строками, имеющими различную длину. Например, в результате сортировки указатели перераспределяются таким образом, что при последовательном обращении к ним возвращаются строки в алфавитном (лексикографическом) порядке.

Нам уже встречалось другое применение массивов строк: массив `argv`, используемый для передачи строковых аргументов функции `main` программ на C++. Система сохраняет в строковом буфере символы командной строки, введенные пользователем, и передает в процедуру `main` указатель на массив указателей строк этого буфера. Для вычисления чисел, соответствующих некоторым аргументам, используются функции преобразования. Остальные аргументы используются непосредственно как строки.

### Программа 3.17 Сортировка массива строк

Эта программа иллюстрирует важную функцию обработки строк: расположение набора строк в порядке сортировки. Строки считываются в буфер, который достаточно велик, чтобы вместить их все. Для каждой строки в массиве сохраняется указатель. Затем изменяется расположение указателей, чтобы указатель на самую младшую строку помещался в первую позицию массива, указатель на следующую в алфавитном порядке строку — во вторую позицию и т.д.

Библиотечная функция `qsort`, которая в действительности выполняет сортировку, принимает четыре аргумента: указатель на начало строки, количество объектов, размер каждого объекта и функцию сравнения. Независимость от типа сортируемых объектов достигается за счет слепого перераспределения блоков данных, которые представляют объекты (в данном случае, указатели на строки), а также путем использования функции сравнения, принимающей в качестве аргумента указатель на `void`. Программа выполняет обратное преобразование этих блоков в тип указателей на указатели символов для функции `strcmp`. Чтобы обратиться к первому символу строки для операции сравнения, разыменовываются три указателя: один для получения индекса (который является указателем) элемента массива, один для получения указателя строки (с помощью индекса) и еще один для получения символа (с помощью указателя).

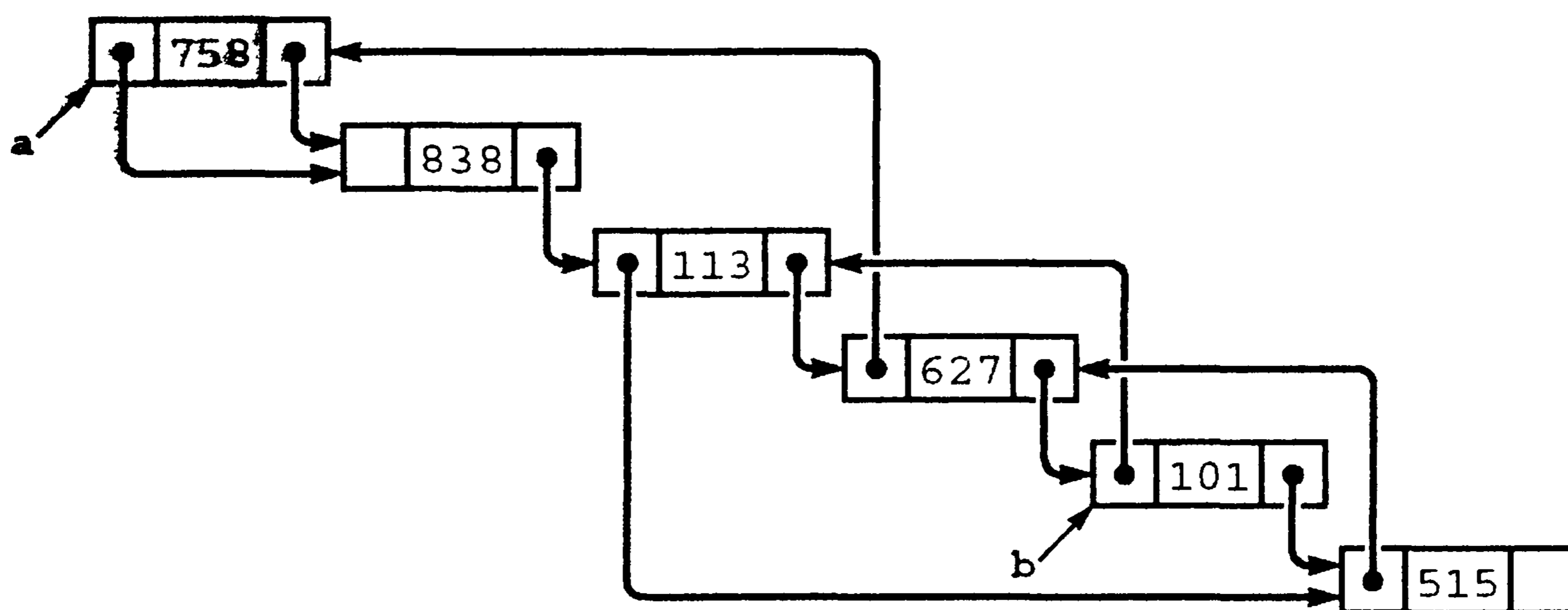
Мы используем другой метод достижения независимости от типа для функций сортировки и поиска (см. главы 4 и 6).

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
int compare(const void *i, const void *j)
{ return strcmp(*(char **)i, *(char **)j); }
int main()
{ const int Nmax = 1000;
  const int Mmax = 10000;
  char* a[Nmax]; int N;
  char buf[Mmax]; int M = 0;
  for (N = 0; N < Nmax; N++)
  {
    a[N] = &buf[M];
    if (!(cin >> a[N])) break;
    M += strlen(a[N])+1;
  }
  qsort(a, N, sizeof(char*), compare);
  for (int i = 0; i < N; i++)
    cout << a[i] << endl;
}
```

Строить составные структуры данных можно также исключительно с помощью ссылок. На рис. 3.13 показан пример *мульти списка*, узлы которого имеют несколько полей ссылок и принадлежат независимо поддерживаемым связным спискам. При разработке алгоритмов для построения сложных структур данных часто применяют более одной ссылки для каждого узла, но это преследует цель эффективного управления ими. Например, список с двойными связями является мульти списком, который удовлетворяет ограничению: оба выражения  $x \rightarrow l \rightarrow r$  и  $x \rightarrow r \rightarrow l$  эквивалентны  $x$ . В главе 5 рассматривается намного больше важных структур данных с двумя ссылками для каждого узла.

Если многомерная матрица является *разреженной* (*sparse*) (количество ненулевых элементов относительно невелико), для ее представления вместо многомерного массива можно использовать мульти список. Каждому значению матрицы соответствует один узел, а каждому измерению — одна ссылка. При этом ссылки указывают на следующий элемент в данном измерении. Эта организация снижает размер области хранения. Она теперь не зависит от максимальных значений индексов измерений, а пропорциональна количеству ненулевых записей. Однако для многих алгоритмов увеличивается время выполнения, поскольку для доступа к отдельным элементам приходится отслеживать ссылки.

Для ознакомления с дополнительными примерами составных структур данных и выяснения различий между индексированными и связными структурами данных рассмотрим структуры данных, применяемые для представления графов. *Граф* — это фундаментальный комбинаторный объект, который описывается набором объектов (называемых *вершинами*) и набором связей между вершинами (называемых *ребрами*). Мы уже встречались с понятием графов в задаче связности из главы 1.



**РИСУНОК 3.13 МУЛЬТИСПИСОК**

Связывать узлы можно с помощью двух полей ссылок в двух независимых списках, по одному на каждое поле. Здесь правое поле связывает узлы в одном порядке (например, этот порядок может отражать последовательность создания узлов), а левое поле — в другом порядке (в нашем случае это порядок сортировки, возможно, результат сортировки вставками, использующей только левое поле ссылки). Отслеживая правые ссылки от узла *a*, мы обходим узлы в порядке их создания. Отслеживая левые ссылки от узла *b*, мы обходим узлы в порядке сортировки.



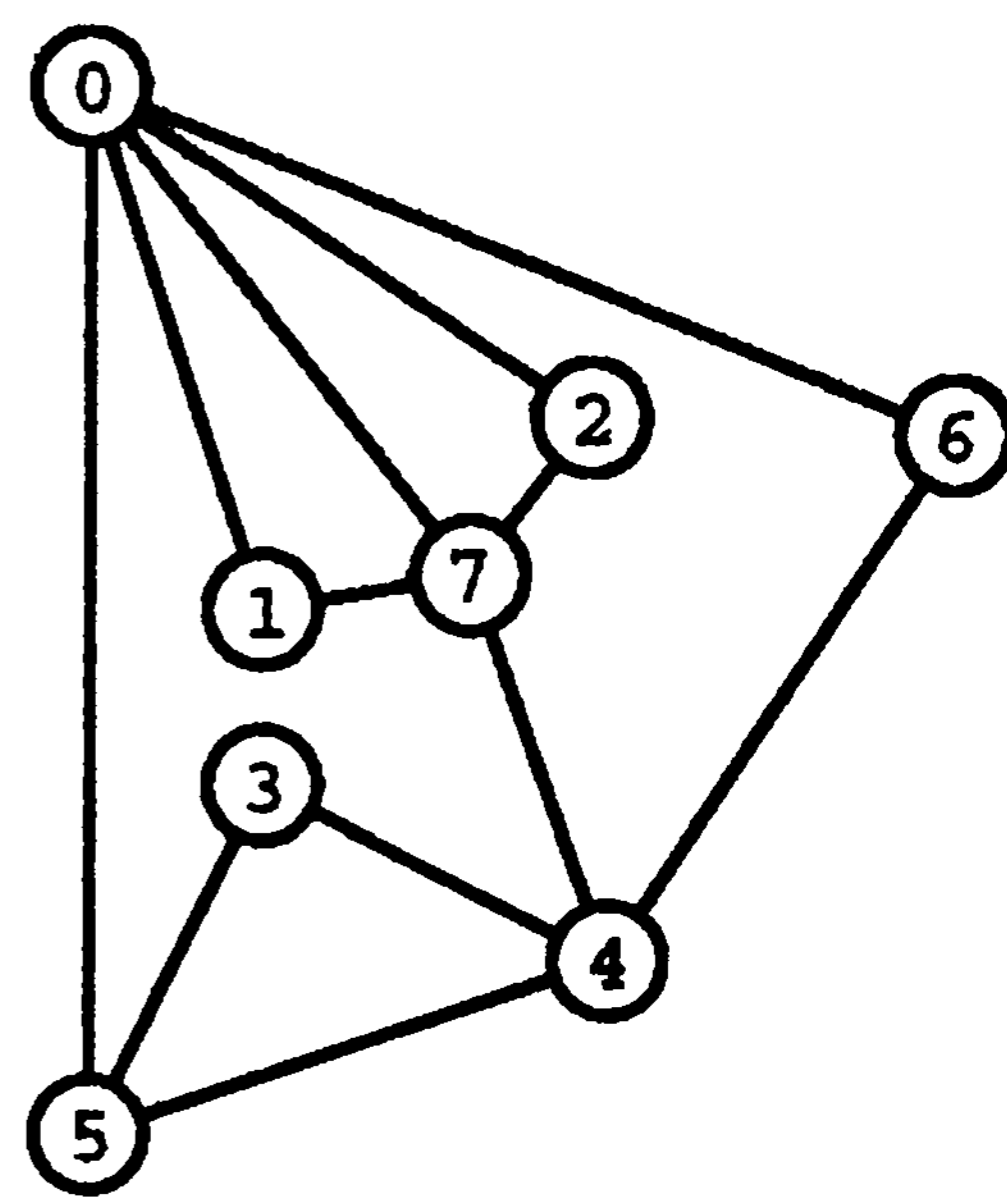
Предположим, что граф с количеством вершин  $V$  и количеством ребер  $E$  описывается набором из  $E$  пар целых чисел в диапазоне от 0 до  $V-1$ . Это означает, что вершины обозначены целыми числами  $0, 1, \dots, V-1$ , а ребра определяются парами вершин. Как и в главе 1, пара  $i-j$  обозначает связь между вершинами  $i$  и  $j$ , и имеет то же значение, что и пара  $j-i$ . Составленные из таких ребер графы называются *неориентированными* (*undirected*). Другие типы графов рассматриваются в части 7.

Один из простых методов представления графа заключается в использовании двумерного массива, называемого *матрицей смежности* (*adjacency matrix*). Она позволяет быстро определять, существует ли ребро между вершинами  $i$  и  $j$  путем простой проверки наличия ненулевого значения в записи матрицы, находящейся на пересечении строки  $i$  и столбца  $j$ . Для неориентированных графов, к которым принадлежит рассматриваемый, при наличии записи в строке  $i$  и столбце  $j$  обязательно существует запись в строке  $j$  и столбце  $i$ . Поэтому матрица симметрична. На рис. 3.14 показан пример матрицы смежности для неориентированного графа. Программа 3.18 демонстрирует создание матрицы смежности для вводимой последовательности ребер.

### Программа 3.18 Представление графа в виде матрицы смежности

Эта программа выполняет чтение набора ребер, описывающих неориентированный граф, и создает для него представление в виде матрицы смежности. При этом элементам  $a[i][j]$  и  $a[j][i]$  присваивается значение 1, если существует ребро от  $i$  до  $j$  либо от  $j$  до  $i$ . В противном случае элементам присваивается значение 0. В программе предполагается, что во время компиляции количество вершин  $V$  постоянно. Иначе пришлось бы динамически выделять память под массив, представляющий матрицу смежности (см. упражнение 3.71).

```
#include <iostream.h>
int main()
{ int i, j, adj[V][V];
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      adj[i][j] = 0;
  for (i = 0; i < V; i++) adj[i][i] = 1;
  while (cin >> i >> j)
    { adj[i][j] = 1; adj[j][i] = 1; }
}
```



	0	1	2	3	4	5	6	7
0	1	1	1	0	0	1	1	1
1	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	1
3	0	0	1	1	1	0	0	0
4	0	0	0	1	1	1	1	0
5	1	0	0	1	1	1	0	0
6	1	0	0	0	1	0	1	0
7	1	1	1	0	1	0	0	1

### РИСУНОК 3.14 ПРЕДСТАВЛЕНИЕ ГРАФА В ВИДЕ МАТРИЦЫ СМЕЖНОСТИ

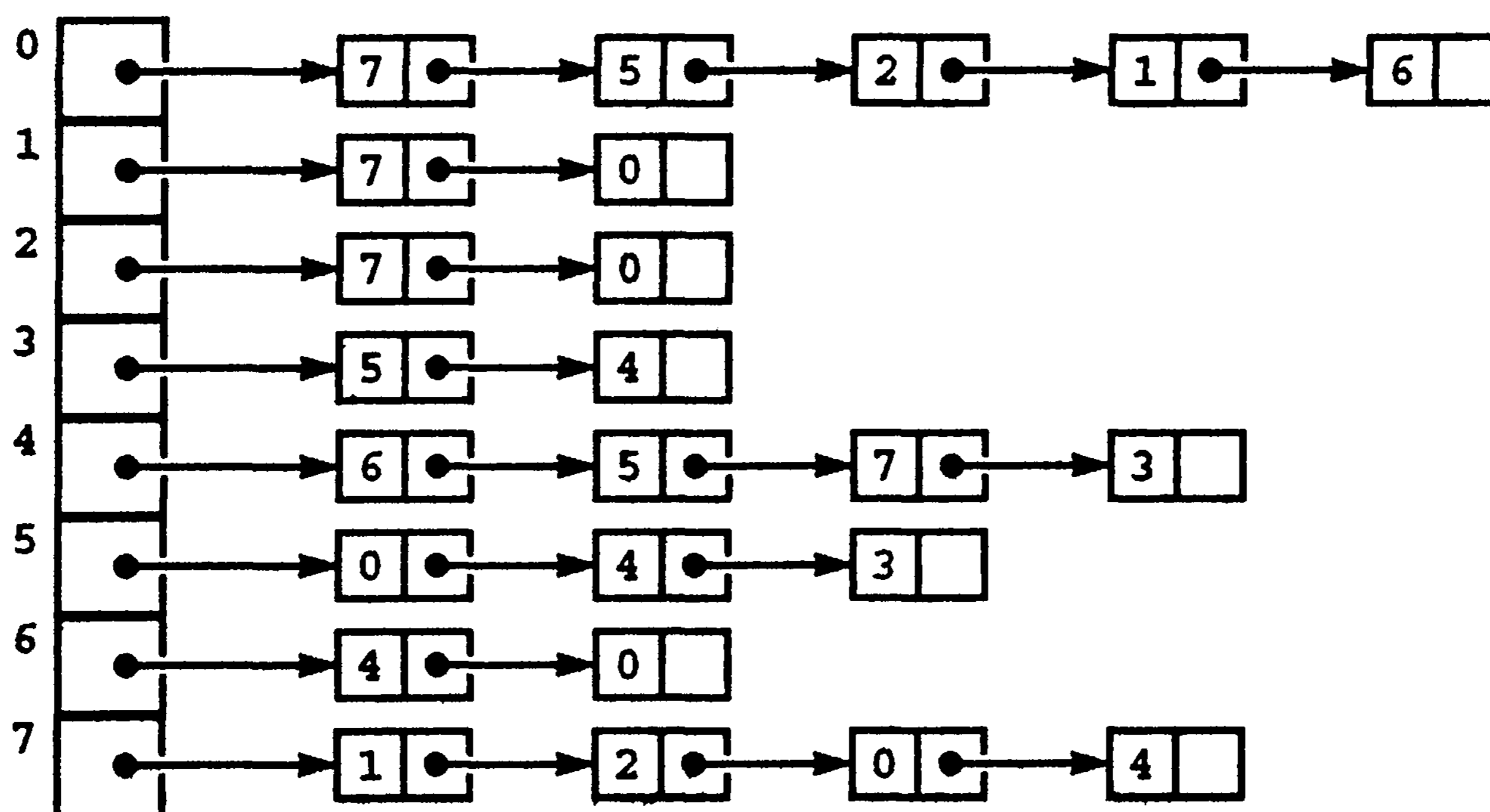
Граф представляет собой набор вершин и ребер, которые их соединяют. Для простоты вершинам присваиваются индексы (последовательность неотрицательных целых чисел, начиная с нуля). Матрица смежности — это двумерный массив, где граф представляется за счет установки разряда (значения 1) в строке  $i$  и столбце  $j$  в том и только том случае, когда между вершинами  $i$  и  $j$  существует ребро. Массив симметричен относительно диагонали. По соглашению разряды устанавливаются во всех позициях диагонали (каждая вершина соединена сама с собой). Например, шестая строка (и шестой столбец) показывают, что вершина 6 соединена с вершинами 0, 4 и 6.

Другой простой метод представления графа предусматривает использование массива связанных списков, называемых *списками смежности (adjacency lists)*. Каждой вершине соответствует связный список с узлами для всех вершин, связанных с данной. В неориентированных графах, если существует узел для вершины  $j$  в  $i$ -том списке, то должен существовать узел для вершины  $i$  в  $j$ -том списке. На рис. 3.15 показан пример представления неориентированного графа с помощью списков смежности. Программа 3.19 демонстрирует метод создания такого представления для вводимой последовательности ребер.

### Программа 3.19 Представление графа в виде списков смежности

Эта программа считывает набор ребер, которые описывают граф, и создает его представление в виде списков смежности. Список смежности представляет собой массив списков, по одному для каждой вершины, где  $j$ -тый список содержит связный список узлов, присоединенных к  $j$ -той вершине.

```
#include <iostream.h>
struct node
{ int v; node* next;
  node(int x, node* t)
  { v = x; next = t; }
};
typedef node *link;
int main()
{ int i, j; link adj[V];
  for (i = 0; i < V; i++) adj[i] = 0;
  while (cin >> i >> j)
  {
    adj[j] = new node(i, adj[j]);
    adj[i] = new node(j, adj[i]);
  }
}
```



**РИСУНОК 3.15 ПРЕДСТАВЛЕНИЕ ГРАФА В ВИДЕ СПИСКОВ СМЕЖНОСТИ**

В этом представлении графа, изображенного на рис. 3.14, применен массив списков. Необходимое для данных пространство пропорционально сумме количеств вершин и ребер. Для поиска индексов вершин, присоединенных к данной вершине  $i$ , анализируется  $i$ -тая позиция массива, содержащая указатель связного списка, который содержит по одному узлу для каждой присоединенной к  $i$  вершины.

Оба представления графа являются массивами более простых структур данных (по одной для каждой вершины), которые описывают ребра, связанные с данной вершиной. Для матрицы смежности более простая структура данных реализована в виде индексированного массива, а для списка смежности — в виде связного списка.

Таким образом, представление графа различными методами создает альтернативные возможности расхода пространства. Для матрицы смежности необходимо пространство, пропорциональное  $V^2$ ; для списков смежности расход памяти пропорционален величине  $V + E$ . При небольшом количестве ребер (такой граф называется *разреженным*), представление с использованием списков смежности потребует намного меньшего размера пространства. Если большинство пар вершин соединены ребрами (такой граф называется *насыщенным*), использование матрицы смежности предпочтительнее, поскольку оно не связано со ссылками. Некоторые алгоритмы более эффективны для представлений на базе матрицы смежности, поскольку требуют постоянных затрат времени для ответа на вопрос: "существует ли ребро между вершинами  $i$  и  $j$ ". Другие алгоритмы более эффективны для представлений на базе списков смежности, поскольку они позволяют обрабатывать все ребра графа за время, пропорциональное  $V + E$ , а не  $V^2$ . Показательный пример этой альтернативы продемонстрирован в разделе 5.8.

Оба типа представлений можно элементарно распространить на другие типы графов (для примера см. упражнение 3.70). Они служат основой большинства алгоритмов обработки графов, которые будут исследоваться в части 7.

В завершение главы рассмотрим пример, демонстрирующий использование составных структур данных для эффективного решения простой геометрической задачи, о которой шла речь в разделе 3.2. Для данного значения  $d$  необходимо узнать количество пар из множества  $N$  точек внутри единичного квадрата, которые можно соединить отрезком прямой с длиной меньшей  $d$ . Программа 3.20 призвана снизить время выполнения программы 3.8 с приблизительным коэффициентом  $1/d^2$  при больших значениях  $N$ . Для этого в программе задействован двумерный массив связных списков. Единичный квадрат разбивается на сетку меньших квадратов одинакового размера. Затем для каждого квадрата создается связный список всех точек, попадающих в квадрат. Двумерный массив обеспечивает немедленный доступ к набору точек, который является ближайшим по отношению к данной точке. Связные списки обладают гибкостью, позволяющей хранить все точки без необходимости знать заранее, сколько точек попадает в каждую ячейку сетки.

Используемое программой 3.20 пространство пропорционально  $1/d^2 + N$ , но время выполнения составляет  $O(d^2N^2)$ , что дает большое преимущество перед прямолинейным алгоритмом из программы 3.8 при небольших значениях  $d$ . Например, для  $N = 10^6$  и  $d = 0.001$  затраты времени и пространства на решение задачи будут практически линейно зависеть от  $N$ . При этом прямолинейный алгоритм потребует неоправданно высоких затрат времени. Эту структуру данных можно использовать в качестве основы для решения многих других геометрических задач. Например, совместно с алгоритмом *union-find* из главы 1 это даст близкий к линейному алгоритм определения факта возможности соединения отрезками длиной  $d$  набора из  $N$  случайных точек на плоскости. Это фундаментальная задача из области проектирования сетей и цепей.

**Программа 3.20 Двумерный массив списков**

Эта программа демонстрирует эффективность удачного выбора структуры данных на примере геометрических вычислений из программы 3.8. Единичный квадрат разбивается на сетку. Создается двумерный массив связанных списков, причем каждой ячейке (квадрату) сетки соответствует один список. Размер ячеек достаточно мал, чтобы все точки в пределах расстояния  $d$  от каждой данной точки попадали в одну ячейку с ней либо в смежную ячейку. Функция `malloc2d` подобна такой же функции из программы 3.16, но она создана для объектов типа `link`, а не `int`.

```
#include <math.h>
#include <iostream.h>
#include <stdlib.h>
#include "Point.h"
struct node
{ point p; node *next;
  node(point pt, node* t) { p = pt; next = t; } };
typedef node *link;
static link **grid;
static int G, cnt = 0; static float d;
void gridinsert(float x, float y)
{ int X = x*G+1; int Y = y*G+1;
  point p; p.x = x; p.y = y;
  link s, t = new node(p, grid[X][Y]);
  for (int i = X-1; i <= X+1; i++)
    for (int j = Y-1; j <= Y+1; j++)
      for (s = grid[i][j]; s != 0; s = s->next)
        if (distance(s->p, t->p) < d) cnt++;
  grid[X][Y] = t;
}
int main(int argc, char *argv[])
{ int i, N = atoi(argv[1]);
  d = atof(argv[2]); G = 1/d;
  grid = malloc2d(G+2, G+2);
  for (i = 0; i < G+2; i++)
    for (int j = 0; j < G+2; j++)
      grid[i][j] = 0;
  for (i = 0; i < N; i++)
    gridinsert(randFloat(), randFloat());
  cout << cnt << " pairs within " << d << endl;
}
```

Как следует из примеров данного раздела, данные различных типов можно объединять (косвенно, либо с помощью явных ссылок) в объекты, а последовательности объектов — в составные объекты. Таким образом, из базовых абстрактных конструкций можно строить объекты неограниченной сложности. Тем не менее, в этих примерах еще не достигнуто полное обобщение структурирования данных, как будет показано в главе 5. Прежде чем пройти последний этап, следует рассмотреть абстрактные структуры данных, которые можно создавать с помощью связанных списков и массивов — основных средств достижения следующего уровня обобщения.

## Упражнения

- 3.62** Написать версию программы 3.16, обрабатывающую трехмерные массивы.
- 3.63** Изменить программу 3.17 для индивидуальной обработки вводимых строк (выделять память каждой строке после считывания ее из устройства ввода). Можно предположить, что длина каждой строки превышает 100 символов.
- 3.64** Написать программу заполнения двумерного массива значений 0—1 путем установки значения 1 для элемента  $a[i][j]$ , если наибольшее общее кратное  $i$  и  $j$  равно единице, и установки значения 0 во всех остальных случаях.
- 3.65** Воспользоваться программами 3.20 и 1.4 для разработки эффективного приложения, которое определяет, можно ли соединить набор из  $N$  точек отрезками длиной меньшей  $d$ .
- 3.66** Написать программу преобразования разреженной матрицы из двумерного массива в мультисписок с узлами для ненулевых значений.
- **3.67** Реализовать перемножение матриц, представленных мультисписками.
  - ▷ **3.68** Отобразить матрицу смежности, построенную программой 3.18, для введенных пар значений: 0-2, 1-4, 2-5, 3-6, 0-4, 6-0 и 1-3.
  - ▷ **3.69** Отобразить список смежности, построенный программой 3.19, для введенных пар значений: 0-2, 1-4, 2-5, 3-6, 0-4, 6-0 и 1-3.
  - **3.70** *Ориентированный (directed) граф* — это граф, у которого соединения между вершинами имеют направление: ребра следуют из одной вершины в другую. Выполнить упражнения 3.68 и 3.69, предположив, что вводимые пары представляют ориентированный граф, где обозначение  $i-j$  указывает, что ребро направлено от  $i$  к  $j$ . Кроме того, изобразить граф, используя стрелки для указания ориентации ребер.
  - 3.71** Модифицировать программу 3.18 таким образом, чтобы она принимала количество вершин в качестве аргумента командной строки, а затем динамически распределяла память под матрицу смежности.
  - 3.72** Модифицировать программу 3.19 таким образом, чтобы она принимала количество вершин в качестве аргумента командной строки, а затем динамически распределяла память под массив списков.
  - **3.73** Написать функцию, которая использует матрицу смежности графа для вычислений по заданным вершинам  $a$  и  $b$  количества вершин  $c$  со свойством наличия ребра от  $a$  до  $c$  и от  $c$  до  $b$ .
  - **3.74** Выполнить упражнение 3.73 с использованием списков смежности.

## Абстрактные ТИПЫ ДАННЫХ

Разработка абстрактных моделей для данных и способов обработки этих данных является важнейшим компонентом в процессе решения задач с помощью компьютера. Примеры этого мы видим на низком уровне в повседневном программировании (когда, например, как обсуждалось в главе 3, мы используем массивы и связанные списки) и на высоком уровне при решении прикладных задач (как было продемонстрировано в главе 1, во время использования бора union-find при решении задачи связности). В настоящей главе рассматриваются *абстрактные типы данных* (*abstract data type*, в дальнейшем АТД), позволяющие создавать программы с использованием высокоуровневых абстракций. За счет применения абстрактных типов данных появляется возможность отделять абстрактные (концептуальные) преобразования, которые программы выполняют над данными, от любого конкретного представления структуры данных и любой конкретной реализации алгоритма.

Все вычислительные системы базируются на *уровнях абстракции*: на основе определенных физических свойств кремния и других материалов берется абстрактная модель бита, который может принимать двоичные значения 0-1; затем на основе динамических свойств значений определенного набора битов берется абстрактная модель машины; далее, на основе принципа работы машины под управлением программы на машинном языке берется абстрактная модель языка программирования; и, наконец, берется абстрактное понятие алгоритма, реализуемое в виде программы на языке C++. Абстрактные типы данных обеспечивают возможность продолжать этот процесс дальше и разрабатывать абстрактные конструкции для

определенных вычислительных задач на более высоком уровне, чем это обеспечивается системой C++; они позволяют также разрабатывать абстрактные конструкции, ориентированные на конкретные приложения и подходящие для решения задач в многочисленных прикладных областях; наконец, они позволяют создавать абстрактные конструкции более высокого уровня, в которых используются эти базовые конструкции. Абстрактные типы данных предоставляют в наше распоряжение постоянно расширяющийся набор инструментальных средств, позволяющий приниматься за решение все новых и новых задач.

С одной стороны, применение абстрактных конструкций освобождает от забот по их детальной реализации; с другой стороны, когда приходится учитывать производительность (быстродействие) программы, необходимо знать затраты на выполнение базовых операций. Мы используем много базовых абстракций, встроенных в аппаратные средства компьютера и служащих основой для машинных инструкций. Мы также реализуем другие абстракции в программном обеспечении. Кроме того, мы используем еще третьи абстракции, существующие в написанном ранее системном программном обеспечении. Абстрактные конструкции более высокого уровня часто создаются на основе более простых конструкций. На всех уровнях действует один и тот же основной принцип: в своих программах мы должны найти наиболее важные операции и наиболее важные характеристики данных, затем точно определить и те и другие на абстрактном уровне и разработать эффективные конкретные механизмы для их поддержки. В настоящей главе приводится множество примеров применения этого принципа.

Для разработки нового уровня абстракции потребуется *определить* абстрактные объекты, с которыми необходимо манипулировать, и операции, которые должны выполняться над ними; мы должны *представить* данные в некоторой структуре данных и *реализовать* операции; и (вот тема для упражнения) необходимо обеспечить, чтобы эти объекты было удобно *использовать* для решения прикладных задач. Перечисленные замечания применимы также к простым типам данных, и базовые механизмы для поддержки типов данных, которые обсуждались в главе 3, могут быть адаптированы для наших целей. Однако язык C++ предлагает важное расширение для механизма структур, называемое *классом*. Классы исключительно полезны при создании уровней абстракции и поэтому рассматриваются в качестве первого инструмента, который используется для этой цели на протяжении оставшейся части книги.

**Определение 4.1** *Абстрактный тип данных (АТД) — это тип данных (набор значений и совокупность операций для этих значений), доступ к которому осуществляется только через интерфейс. Программу, которая использует АТД, будем называть клиентом, а программу, в которой содержится спецификация этого типа данных — реализацией.*

Ключевое отличие, делающее тип данных абстрактным, характеризуется словом *только*: в случае АТД клиентские программы не имеют доступа к значениям данных никаким другим способом, кроме как посредством операций, имеющихся в интерфейсе. Представление этих данных и функции, реализующие эти операции, находятся в реализации и полностью отделены интерфейсом от клиента. Мы говорим, что интерфейс является *непрозрачным*: клиент не может видеть реализацию через интерфейс.

В программах на языке C++ это различие обычно проводится немного лучше, так как проще всего создавать интерфейс, включая в него представление данных; но при этом клиентским программам не разрешается прямой доступ к данным. Другими словами, разработчики клиентских программ могут знать представление данных, но никоим образом не могут его *использовать*.

В качестве примера рассмотрим интерфейс типа данных для точек (программа 3.3) из раздела 3.1. В этом интерфейсе явно объявляется, что точки представлены как структуры, состоящие из пары чисел с плавающей точкой, обозначаемых  $x$  и  $y$ . В самом деле, подобное применение типов данных является обычным в больших системах программного обеспечения: мы разрабатываем набор правил относительно того, как должны быть представлены данные (а также определяем ряд связанных с ними операций), и делаем эти правила доступными через интерфейс, чтобы ими могли пользоваться клиентские программы, формирующие большую систему. Тип данных обеспечивает согласованность всех частей системы с представлением основных общесистемных структур данных. Какой бы хорошей такая стратегия ни была, она имеет один изъян: если необходимо *изменить* представление данных, то потребуются изменить и все клиентские программы. Программа 3.3 снова дает нам простой пример: одна из причин разработки этого типа данных — сделать так, чтобы клиентским программам было удобно манипулировать с точками, и мы ожидаем, что в случае необходимости у клиентов будет доступ к отдельным координатам точки. Но мы не можем перейти к другому представлению данных (скажем, к полярным координатам, или трехмерным координатам, или даже к другим типам данных для отдельных координат) без изменения всех клиентских программ.

В отличие от этого, программа 4.1 содержит реализацию абстрактного типа данных, соответствующего типу данных из программы 3.3. В этой реализации используется класс языка C++, в котором сразу определены как данные, так и связанные с ними операции. Программа 4.2 является клиентской программой, работающей с упомянутым типом данных. Эти две программы выполняют те же самые вычисления, что и программы 3.3 и 3.8. Они иллюстрируют ряд основных свойств классов, которые сейчас будут рассматриваться.

---

#### Программа 4.1 Реализация класса POINT (точка)

---

В этом классе определен тип данных, который состоит из набора значений, представляющих собой "пары чисел с плавающей точкой" (предполагается, что они интерпретируются как точки на декартовой плоскости), а также две *функции-члена*, определенные для всех экземпляров класса **POINT**: функция **POINT()**, которая является *конструктором*, инициализирующим координаты случайными значениями от 0 до 1, и функция **distance(POINT)**, вычисляющая расстояние до другой точки. Представление данных является **приватным (private)**, и обращаться к нему или модифицировать его могут только функции-члены. В свою очередь, функции-члены являются **общедоступными (public)** и доступны для любого клиента. Код можно сохранить, например, в файле с именем **POINT.cxx**.

```
#include <math.h>
class POINT
{
    private:
        float x, y;
```



```

public:
    POINT()
    { x = 1.0*rand()/RAND_MAX;
      y = 1.0*rand()/RAND_MAX; }
    float distance(POINT a)
    { float dx = x-a.x, dy = y-a.y;
      return sqrt(dx*dx + dy*dy); }
};

```

Когда мы записываем в программе определение наподобие `int i`, мы даем системе команду зарезервировать область памяти для данных (встроенного) типа `int`, на которую будем ссылаться по имени `i`. В языке C++ такая сущность называется *объектом*. При записи в программе такого определения, как `POINT p`, говорят, что *создается* объект класса `POINT`, ссылка на который осуществляется по имени `p`. В нашем примере каждый объект имеет два элемента данных, которые называются `x` и `y`. Так же как и в случае структур, на них ссылаются по именам, подобным `p.y`.

Элементы данных `x` и `y` называются *данными-членами* класса. В классе могут быть также определены *функции-члены*, которые реализуют операции, связанные с этим типом данных. Например, класс, определенный в программе 4.1, имеет две функции-члена, которые называются `POINT` и `distance`.

Клиентские программы, такие как программа 4.2, могут обращаться к функциям-членам, связанным с объектом, ссылаясь на них по именам точно так же, как они ссылаются на данные, находящиеся в какой-нибудь структуре `struct`. Например, выражение `p.distance(q)` вычисляет расстояние между точками `p` и `q` (это же самое расстояние даст и выражение `q.distance(p)`). Функция `POINT()`, первая функция в программе 4.1, является особой функцией-членом, называемой *конструктором*: у нее такое же имя, как и у класса, и она вызывается тогда, когда требуется создать объект этого класса. Определение `POINT p` в программе-клиенте приводит к распределению области памяти под новый объект и затем (посредством функции `POINT()`) к присвоению каждому из двух его элементов данных случайного значения в диапазоне от 0 до 1.

#### Программа 4.2 Программа-клиент для класса `POINT` (нахождение ближайшей точки)

Эта версия программы 3.8 является клиентом, который использует АТД `POINT`, определенный в программе 4.3. Оператор `new[]` создает массив объектов абстрактного типа `POINT` (вызывая конструктор `POINT()` для инициализации каждого объекта случайными значениями координат). Оператор `a[i].distance(a[j])` вызывает для объекта `a[i]` функцию-член `distance` с аргументом `a[j]`.

```

#include <iostream.h>
#include <stdlib.h>
#include "POINT.cxx"
int main(int argc, char *argv[])
{ float d = atof(argv[2]);
  int i, cnt = 0, N = atoi(argv[1]);
  POINT *a = new POINT[N];
  for (i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
      if (a[i].distance(a[j]) < d) cnt++;
  cout << cnt << " pairs within " << d << endl;
}

```

Этот стиль программирования, который иногда называется *объектно-ориентированным программированием*, полностью поддерживается конструкцией `class` языка C++. Можно думать о классе как о расширении структуры, где мы не только собираем данные, но также определяем операции над этими данными. Может существовать много разных объектов, принадлежащих одному классу, но все они подобны в том, что их данные-члены могут принимать один и тот же набор значений и над этими данными-членами может выполняться одна и та же совокупность операций; коротко говоря, они являются экземплярами одного и того же типа данных. В объектно-ориентированном программировании мы даем объектам команды обрабатывать свои данные-члены (в противоположность использованию независимых функций для обработки данных, хранимых в объектах).

Этот небольшой класс рассматривается в качестве примера с той целью, чтобы познакомиться с основными особенностями классов; поэтому он — далеко не полностью заверченный класс. В реальном коде для класса точки у нас будет намного больше операций. Например, в программе 4.1 отсутствуют даже операции, позволяющие узнавать значения координат  $x$  и  $y$ . Как мы увидим, добавление этих и других операций — задача довольно простая. В части 5 мы более подробно рассмотрим классы для точки и других геометрических абстракций, например, линий и многоугольников.

В языке C++ (но не в C) у структур также могут быть связанные с ними функции. Ключевое различие между классами и структурами связано с доступом к информации, который характеризуется ключевыми словами **private** и **public**. На *приватный* (*private*) член класса можно ссылаться только *внутри* класса, в то время как на *общедоступный* (*public*) член класса может ссылаться любой клиент. Приватными членами могут быть как данные, так и функции: в программе 4.1 приватными являются только данные, но далее мы увидим многочисленные примеры классов, в которых приватными будут также функции-члены. По умолчанию члены классов являются приватными, тогда как члены структур — общедоступными.

Например, в клиентской программе, использующей класс **POINT**, мы не можем ссылаться на данные-члены **p.x**, **p.y** и т.д., как могли бы в случае, если бы программа использовала структуру **POINT**, поскольку члены класса  $x$  и  $y$  являются приватными. Все что можно предпринять — это использовать для обработки точек общедоступные функции-члены. Такие функции имеют прямой доступ к данным-членам любого объекта этого класса. Например, когда в программе 4.1 мы вызываем функцию **distance** с помощью оператора **p.distance(q)**, в операторе **dx = x - a.x** имя  $x$  относится к данным-членам  $x$  в точке **p** (поскольку функция **distance** была вызвана как функция-член экземпляра **p**), а имя **a.x** относится к данным-члену  $x$  в точке **q** (так как **q** — это действительный параметр, соответствующий формальному параметру **a**). Дабы исключить возможную двусмысленность или путаницу, можно было бы записать **dx = this->x-a.x**, — ключевое слово **this** относится к указателю на объект, для которого вызывается функция-член.

Когда к данным-членам применяется ключевое слово **static**, это, как и в случае обычных чисел, означает, что существует только одна копия этой переменной (относящаяся к классу), а не множество копий (относящихся к отдельным объектам). Эта возможность часто используется, например, для отслеживания статистики, касающей-

ся объектов: можно в класс **POINT** включить переменную **static int N**, добавить в конструктор **N++**, и тогда появится возможность знать количество созданных точек.

Конечно, можно принять за факт, что функция, вычисляющая расстояние между двумя точками, должна иметь *не* один аргумент (любую из двух точек), а, как в предыдущей реализации, два аргумента (обе точки), что более естественно. В языке C++ этот подход можно реализовать, следующим образом определив в классе **POINT** другую функцию **distance**:

```
static float distance(POINT a, POINT b)
{
    float dx = a.x - b.x, dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}
```

*Статическая* функция-член может иметь доступ к членам класса, но ее не требуется вызывать для отдельного объекта.

Другая возможная альтернатива — определить функцию **distance** как независимую функцию вне объявления класса **POINT** (используя тот же самый код, что и в предыдущем абзаце, но опуская ключевое слово **static**). Поскольку эта версия функции **distance** должна иметь доступ к приватным данным-членам класса **POINT**, в объявление класса **POINT** потребуется включить строку

```
friend float distance(POINT, POINT);
```

*Дружественная (friend) функция* — это функция, которая, не будучи членом класса, имеет доступ к его приватным членам.

Чтобы клиентские программы имели возможность читать значения данных, можно определить функции-члены, возвращающие эти значения:

```
float X() const { return x; }
float Y() const { return y; }
```

Эти функции определяются с ключевым словом **const**, так как они не модифицируют данные-члены объекта, для которого вызываются. Мы часто включаем в классы языка C++ функции такого типа. Обратите внимание, что если бы использовалось другое представление данных, например, полярные координаты, то реализовать эти функции было бы труднее; тем не менее, эта трудность была бы прозрачной для клиента. Преимущества подобной гибкости можно также задействовать в функциях-членах — если в реализации функции **distance** из программы 4.1 вместо ссылок типа **a.x** использовать ссылки типа **a.X()**, то при изменении представления данных не придется изменять этот код. Помещая эти функции в приватную часть класса, можно позволить себе такую гибкость даже в тех классах, где не требуется доступ клиента к данным.

Во многих приложениях главная цель создания класса связана с определением нового типа данных, наиболее адекватно отвечающего потребностям приложения. В таких ситуациях часто обнаруживается, что использовать этот тип данных требуется таким же самым образом, как и встроенные типы данных языка C++, например, **int** или **float**. Эта тема рассматривается более подробно в разделе 4.8. Один важный инструмент, помогающий нам достичь этой цели, называется *перегрузкой операций (operator overloading)*; в языке C++ она позволяет задать, что к объекту класса необхо-

можно применить фундаментальную операцию и что в точности эта операция должна делать. Например, предположим, что требуется считать две точки идентичными, если расстояние между ними меньше чем 0.001. Добавляя в класс код

```
friend int operator==(POINT a, POINT b)
{ return distance(a, b) < .001; }
```

можно с помощью операции == проверять, равны ли две точки (согласно приведенному определению).

Другой операцией, которую обычно желают перегрузить, является операция << из класса ostream. При программировании на языке C++ обычно полагают, что эту операцию можно использовать для вывода значений любого объекта; в действительности же так можно делать только в том случае, если в классе определена перегруженная операция <<. В классе POINT это можно сделать следующим образом:

```
ostream& operator<<(ostream& t, POINT p)
{
    cout << "(" << p.X() << ", " << p.Y() << ")";
    return t;
}
```

Эта операция не является ни функцией-членом, ни даже дружественной: для доступа к данным она использует общедоступные функции-члены X() и Y().

Как понятие класса языка C++ соотносится с моделью клиент-интерфейс-реализация и абстрактными типами данных? Оно обеспечивает непосредственную языковую поддержку, но достаточно распространенным является тот факт, что существует несколько различных подходов к созданию класса, которые можно использовать. Общепринято следующее правило: *объявления общедоступных функций в классе образуют его интерфейс*. Другими словами, представление данных хранится в приватной части класса, где оно недоступно для программ, использующих класс (клиентских программ). Все, что клиентские программы "знают" о классе — это общедоступная информация о его функциях-членах (имя, тип возвращаемого значения и типы аргументов). Чтобы подчеркнуть природу интерфейса (т.е. то, что он определяется посредством класса), сначала рассмотрим интерфейс (как это иллюстрируется в программе 4.3). Затем мы рассмотрим одну реализацию — программу 4.1. Здесь важно то, что такой порядок упрощает исследование других реализаций, с другими представлениями данных и другими реализациями функций, а также тестирование и сравнение реализаций, позволяя делать это без каких-либо изменений клиентских программ.

### Программа 4.3 Интерфейс абстрактного типа данных POINT

В соответствие с общепринятым правилом, интерфейс, относящийся к реализации АД в виде класса, получают путем устранения приватных частей и замещения реализаций функций их объявлениями (сигнатурами). Данный интерфейс именно так и получен из программы 4.1. Мы можем использовать различные реализации, имеющие один и тот же интерфейс, без внесения изменений в код клиентских программ, работающих с этим АД.

```
class POINT
{
    private:
        // программный код, зависящий от реализации
```

```
public:
    POINT();
    float distance(POINT) const;
};
```

Для многих приложений возможность изменения реализаций является обязательной. Например, предположим, что создается программное обеспечение для компании, которой необходимо обрабатывать списки почтовых адресов потенциальных клиентов. С помощью классов C++ можно определить функции, которые позволяют клиентским программам манипулировать с данными без непосредственного доступа к ним. Мы создаем функции-члены, возвращающие требуемые данные. Например, можно предоставить клиентским программам интерфейс, в котором определены такие операции как *извлечь имя клиента* или *добавить запись клиента*. Самое важное следствие такой организации заключается в том, что те же самые клиентские программы можно использовать даже в том случае, если потребуется изменить формат списков почтовых адресов. Это значит, что можно изменять представление данных и реализацию функций, имеющих доступ к этим данным, без необходимости соответствующей модификации клиентских программ.

Подобного рода реализации типов данных в виде классов иногда называют *конкретными типами данных (concrete data types)*. Однако, в действительности, тип данных, который подчиняется этим правилам, удовлетворяет также и нашему определению абстрактного типа данных (определение 4.1) — различие между ними является вопросом точного определения таких слов, как "доступ", "называть" и "определять", что является делом довольно хитроумным, и мы оставляем его теоретикам языков программирования. Действительно, в определении 4.1 точно не сформулировано, что такое интерфейс или как должны быть описаны тип данных и операции. Такая неопределенность является неизбежной, поскольку попытка точно выразить эту информацию в наиболее общем виде потребует использования формального математического языка и, в конце концов, приведет к трудным математическим вопросам. Этот вопрос является центральным при проектировании языка программирования. Вопросы спецификации будут рассматриваться позже, после изучения нескольких примеров абстрактных типов данных.

Применение классов языка C++ для реализации абстрактных типов данных с соблюдением общепринятого правила, гласящего, что интерфейс состоит из определений общедоступных функций, не является идеальным вариантом, поскольку интерфейс и реализация разделяются не полностью. Когда мы изменяем реализацию, нам приходится перекомпилировать программы-клиенты. Некоторые альтернативные варианты рассматриваются в разделе 4.5.

Абстрактные типы данных возникли в качестве эффективного механизма поддержки *модульного программирования* как принципа организации больших современных систем программного обеспечения. Они являются средством, позволяющим ограничить размеры и сложность интерфейса между (потенциально сложными) алгоритмами и, связанными с ними структурами данных, с одной стороны, и программами (потенциально — большим количеством программ), использующими эти алгоритмы и структуры данных, с другой стороны. Этот принцип упрощает понимание больших

прикладных программ в целом. Более того, абстрактные типы данных, в отличие от простых типов данных, обеспечивают гибкость, необходимую для удобного изменения или улучшения в системе фундаментальных структур данных и алгоритмов. Самое главное, что интерфейс АД определяет соглашение между пользователями и разработчиками, который обеспечивает точные правила взаимодействия, причем каждый знает, что можно ожидать от другого.

В случае тщательно спроектированных АД отделение программ-клиентов от реализаций можно использовать различными интересными способами. Например, при разработке или отладке реализаций АД обычно применяются программы-драйверы. Чтобы узнать свойства программ-клиентов, при построении систем в качестве заполнителей часто используются также неполные реализации абстрактных типов данных, называемые *заглушками (stubs)*.

В настоящей главе абстрактные типы данных рассматриваются подробно потому, что они играют важную роль в изучении структур данных и алгоритмов. Действительно, важной движущей силой разработки почти всех алгоритмов, рассматриваемых в этой книге, является стремление обеспечить эффективные реализации базовых операций для некоторых фундаментальных АД, играющих исключительно важную роль при решении многих вычислительных задач. Проектирование абстрактного типа данных — это только первый шаг навстречу потребностям прикладных программ; необходимо также разработать живучие реализации связанных с ними операций, а также структур данных, лежащих в основе этих операций. Эти задачи и являются темой настоящей книги. Более того, абстрактные модели используются непосредственно для разработки алгоритмов и структур данных, а также для сравнения их характеристик производительности, как это было в примере из главы 1: как правило, разрабатывается прикладная программа, использующая АД, для решения некоторой задачи, затем разрабатываются несколько реализаций АД и сравнивается их эффективность. В настоящей главе этот общий процесс рассматривается подробно, со множеством примеров.

Программисты, пишущие на C++, регулярно используют как простые, так и абстрактные типы данных. Когда мы на низком уровне обрабатываем целые числа, используя только операции, имеющиеся в языке C++, мы, по существу, используем системно определяемую абстракцию для целых чисел. На какой-нибудь новой машине целые числа могут быть представлены, а операции реализованы, каким-либо иным способом, но программа, которая использует только операции, определенные для целых чисел, будет работать корректно и на новой машине. В этом случае различные операции языка C++ для целых чисел составляют интерфейс, наши программы являются клиентами, а программное и аппаратное обеспечение системы обеспечивают реализацию. Часто эти типы данных достаточно абстрактны для того, чтобы мы, не изменяя свои программы, могли перейти на новую машину со, скажем, другими представлениями для целых чисел и чисел с плавающей точкой. Однако этот пример иллюстрирует также тот факт, что такая идеальная ситуация случается не столь часто, как хотелось бы, поскольку клиентские программы могут собирать информацию о представлении данных, определяя предельные значения того или иного ресурса. Например, узнать некоторую информацию о том, как в машине представлены целые

числа, можно, скажем, выполняя цикл, в котором некоторое целое число умножается на два до тех пор, пока не появится сообщение об ошибке переполнения.

Ряд примеров в главе 3 представляет собой программы на C++, написанные в стиле языка C. Программисты на C часто определяют интерфейсы в заголовочных файлах, описывающих наборы операций для некоторых структур данных; при этом реализации находятся в каком-нибудь другом, независимом файле программы. Такой порядок представляет собой соглашение между пользователем и разработчиком, и служит основой для создания стандартных библиотек, доступных в средах программирования на языке C. Однако многие такие библиотеки включают в себя операции для определенной структуры данных и поэтому представляют собой типы данных, но не *абстрактные* типы данных. Например, библиотека строк языка C не является абстрактным типом данных, поскольку программы, работающие со строками, знают, как представлены строки (массивы символов) и, как правило, имеют к ним прямой доступ через индексы массива или арифметику указателей.

В отличие от этого, классы языка C++ позволяют нам не только использовать разные реализации операций, но и создавать их на основе разных структур данных. Опять-таки, ключевое отличие, характеризующее АД, заключается в том, что мы можем производить изменения, не модифицируя клиентские программы; это вытекает из того требования, что доступ к типу данных должен осуществляться *только* посредством интерфейса. Ключевое слово `cv` в определении класса предотвращает прямой доступ клиентских программ к данным. Например, можно было бы включить в стандартную библиотеку языка C++ реализацию класса `string`, созданную на базе, скажем, представления строки в виде связного списка и использовать эту реализацию без изменения клиентских программ.

На протяжении всей настоящей главы мы будем видеть многочисленные примеры реализаций АД с помощью классов языка C++. После четкого уяснения этих понятий в конце главы мы вернемся к обсуждению соответствующих философских и практических следствий.

## Упражнения

- ▷ 4.1 Предположим, необходимо подсчитать количество пар точек, находящихся внутри *квадрата* со стороной  $d$ . Чтобы решить эту задачу, сделайте две разные версии программы-клиента и реализации: во-первых, модифицируйте соответствующим образом функцию-член `distance`; во-вторых, замените функцию-член `distance` функциями-членами `X` и `Y`.
- 4.2 В программе 4.3 к классу точки добавьте функцию-член, которая возвращает расстояние до начала координат.
- 4.3 В программе 4.3 модифицируйте реализацию АД `Point` таким образом, чтобы точки были представлены полярными координатами.
- 4.4 Напишите клиентскую программу, которая считывает из командной строки целое число  $N$  и заполняет массив  $N$  точками, среди которых нет двух равных друг другу. Для проверки равенства или неравенства точек используйте перегруженную операцию `==`, описанную в тексте настоящей главы.

- 4.5 Используя представление на базе связного списка, подобное тому, что в программе 3.14, преобразуйте интерфейс обработки списка (*list-processing interface*) из раздела 3.4 (программа 3.12) в реализацию АДТ на базе классов. Протестируйте полученный интерфейс, модифицируя клиентскую программу (программу 3.13) так, чтобы она использовала этот интерфейс; затем перейдите к реализации на базе массивов (см. упражнение 3.52).

## 4.1 Абстрактные объекты и коллекции объектов

Используемые в приложениях структуры данных часто содержат огромное количество разнотипной информации, и некоторые элементы этой информации могут принадлежать нескольким независимым структурам данных. Например, файл персональных данных может содержать записи с именами, адресами и какой-либо иной информацией о служащих; вполне возможно, что каждая запись должна принадлежать и к структуре данных, используемой для поиска информации об отдельных служащих, и к структуре данных, используемой для ответа на запросы статистического характера, и т.д.

Несмотря на это разнообразие и сложность, в большом классе прикладных программ обработки данных производятся обобщенные манипуляции объектами данных, а доступ к информации, связанной с этими объектами, требуется только в ограниченном числе особых случаев. Многие из этих манипуляций являются естественным продолжением базовых вычислительных процедур, поэтому их часто приходится выполнять в самых разнообразных приложениях. Многие из фундаментальных алгоритмов, рассматриваемых в настоящей книге, можно эффективно применять в какой-либо задаче построения уровня абстракции, позволяющего клиентским программам рационально выполнять эти манипуляции. Поэтому мы подробно рассмотрим многочисленные АДТ, связанные с манипуляциями подобного рода. В этих АДТ определены различные операции над коллекциями абстрактных объектов, не зависящие от типов самих объектов.

В разделе 3.1, в котором обсуждалось применение простых типов данных для написания программ, не зависящих от типов объектов, для задания типов элементов данных использовался **typedef**. Этот подход позволяет использовать один и тот же код для, скажем, целых чисел и чисел с плавающей точкой за счет простого изменения **typedef**. При использовании указателей типы объектов могут быть сколь угодно сложными. При таком подходе часто приходится делать неявные предположения относительно операций, выполняемых над объектами (например, в программе 3.2 предполагается, что для объектов типа **Number** определены операции сложения, умножения и приведения к типу **float**) и, кроме того, мы не скрываем представление данных от клиентских программ. Абстрактные типы данных позволяют делать явными любые предположения относительно операций, выполняемых над объектами данных.

В оставшейся части данной главы рассматривается несколько примеров использования классов C++ с целью построения АДТ для обобщенных объектов данных. Будет продемонстрировано создание АДТ для обобщенных объектов типа **Item**, позволяющие записывать клиентские программы, в которых объекты **Item** используются точно так же, как и встроенные типы данных. Там, где это необходимо, в классе **Item** явно определяются операции, которые должны выполняться над обобщенными



объектами с помощью наших алгоритмов. Все эти характеристики объектов определяются вместе с сокрытием от клиентских программ любой информации о представлении данных.

После реализации класса `Item` для обобщенных объектов (либо принятия решения использовать встроенный класс), воспользуемся механизмом *шаблонов* языка C++ для написания кода, который является обобщенным относительно типов объектов. Например, операцию обмена для обобщенных элементов можно определить следующим образом:

```
template <class Item>
void exch(Item &x, Item &y)
{ Item t = x; x = y; y = t; }
```

Аналогичным образом реализуются и другие простые операции для элементов. С помощью шаблонов можно задавать семейства классов, по одному классу для каждого типа элементов.

Разобравшись с классами обобщенных объектов, можно перейти к рассмотрению *коллекций* (*collection*) объектов. Многие структуры данных и алгоритмы применяются для реализации фундаментальных АД, представляющих собой коллекции абстрактных объектов и создаваемых на основе двух следующих операций:

- *вставить* новый объект в коллекцию.
- *удалить* объект из коллекции.

Такие АД называются *обобщенными очередями* (*generalized queues*). Как правило, для удобства в них также явно включаются следующие операции: *создать* (*construct*) структуру данных (конструкторы), *подсчитать* (*count*) количество объектов в структуре данных (или просто проверить, не пуста ли она). Могут потребоваться и операции, наподобие *уничтожить* (*destroy*) структуру данных (деструкторы) и *копировать* (*copy*) структуру данных (конструкторы копирования); эти операции обсуждаются в разделе 4.8.

Когда объект *вставляется*, намерения совершенно ясны; но какой объект имеется в виду при удалении объекта из коллекции? В разных АД для коллекций объектов применяются различные критерии для определения того, какой объект удалять в операции *удаления*, и устанавливаются различные правила, связанные с этими критериями. Более того, помимо операций *вставить* и *удалить*, мы столкнемся с рядом других естественных операций. Многие алгоритмы и структуры данных, рассматриваемые в книге, были спроектированы с целью обеспечения эффективной реализации различных поднаборов этих операций для разнообразных критериев выполнения операции *удаления* и других правил. Эти АД в принципе просты, используются широко и являются центральными в огромном числе задач по обработке данных; именно поэтому они заслуживают пристального внимания.

Мы рассматриваем некоторые из этих фундаментальных структур данных, их свойства и примеры применения; в то же самое время мы используем их в качестве примеров, чтобы проиллюстрировать основные механизмы, используемые для разработки АД. В разделе 4.2 исследуется *стек магазинного типа* (*pushdown stack*), в котором для удаления объектов используется следующее правило: всегда удаляется объект,

добавленный последним. В разделе 4.3 рассматриваются различные применения стеков, а в разделе 4.4 — реализации стеков, придерживаясь при этом того подхода, что реализации должны быть отделены от приложений. После обсуждения стеков мы вернемся к предыдущему материалу и рассмотрим процесс создания нового АД в контексте абстракции `union-find` для задачи связности, которая исследовалась в главе 1. После этого мы вернемся к коллекциям абстрактных объектов и рассмотрим очереди FIFO (которые на данном уровне абстракции отличаются от стеков только тем, что в них применяется другое правило удаления элементов), а также обобщенные очереди, где запрещены повторяющиеся элементы.

Как было показано в главе 3, массивы и связные списки обеспечивают основные механизмы, позволяющие *вставлять* и *удалять* заданные элементы. Действительно, связные списки и массивы — это структуры данных, лежащие в основе нескольких реализаций рассматриваемых нами обобщенных очередей. Как мы знаем, затраты на вставку и удаление элементов зависят от конкретной используемой структуры и от конкретного вставляемого или удаляемого элемента. В отношении некоторого данного АД задача заключается в том, чтобы выбрать структуру данных, позволяющую эффективно выполнять требуемые операции. В настоящей главе подробно рассматриваются несколько примеров абстрактных типов данных, для которых связные списки и массивы обеспечивают подходящие решения. Абстрактные типы данных, обеспечивающие более эффективные операции, требуют более сложных реализаций, что является главной движущей силой создания многих алгоритмов, рассматриваемых в настоящей книге.

Типы данных, которые состоят из коллекций абстрактных объектов (обобщенные очереди), являются центральным объектом изучения в компьютерных науках, поскольку они непосредственно поддерживают фундаментальную модель вычислений. Оказывается, что при выполнении многих вычислений приходится иметь дело с большим числом объектов, но обрабатывать их можно только поочередно — по одному объекту за один раз. Поэтому во время обработки одного объекта требуется, чтобы все остальные были сохранены. Эта обработка может включать проверку некоторых, уже сохраненных объектов, или добавление в коллекцию новых объектов, но операции сохранения объектов и их извлечения в соответствии с определенным критерием являются основой таких вычислений. Как будет показано, этому шаблону соответствуют многие классические структуры данных и алгоритмы.

В языке C++ классы, реализующие коллекции абстрактных объектов, называются *контейнерными классами* (*container classes*). Некоторые из обсуждаемых структур данных реализованы в библиотеке языка C++ или ее расширениях (стандартной библиотеке шаблонов — Standard Template Library). Во избежание путаницы, мы будем редко ссылаться на эти классы, и материал книги представляется, начиная с самых основ.

## Упражнения

- ▷ 4.6 Дайте определение для класса `Item`, в котором для проверки равенства чисел с плавающей точкой используется перегруженная операция `==`. Считайте два числа с плавающей точкой равными, если абсолютная величина их разности, деленная на большее (по абсолютной величине) из двух чисел, меньше чем  $10^{-6}$ .

▷ 4.7 Дайте определение класса `Item` и перегрузите операции `==` и `<<` так, чтобы их можно было использовать в программе обработки игральные карт.

4.8 Перепишите программу 3.1 так, чтобы в ней использовался класс обобщенных объектов `Item`. Ваша программа должна работать для любого типа объектов класса `Item`, которые могут выводиться при помощи операции `<<`, генерироваться случайным образом с использованием статической функции-члена `rand()` и для которых определены операции `+` и `/`.

## 4.2 АД для стека магазинного типа

Самый важный тип данных из тех, в которых определены операции *вставить* и *удалить* для коллекций объектов, называется *стек* магазинного типа

Стек работает отчасти подобно ящику для входящей корреспонденции весьма занятого профессора: работа скапливается стопкой, и каждый раз, когда у профессора появляется возможность просмотреть какую-нибудь работу, он берет ее сверху. Работа студента вполне может застрять на дне стопки на день или два, однако, скорее всего, добросовестный профессор к концу недели управится со всей стопкой и освободит ящик. Далее несложно будет заметить, что работа компьютерных программ организована именно таким образом, и это вполне естественно. Они часто откладывают некоторые задачи и выполняют в это время другие; более того, зачастую им требуется в первую очередь вернуться к той задаче, которая была отложена последней. Таким образом, стеки магазинного типа являются фундаментальной структурой данных для множества алгоритмов.

**Определение 4.2** *Стек магазинного типа* — это АД, который включает две основные операции: *вставить*, или *затолкнуть* (`push`) новый элемент и *удалить*, или *вытолкнуть* (`pop`) элемент, вставленный последним.

Когда мы говорим об АД стека магазинного типа, мы ссылаемся на описание операций *затолкнуть* и *вытолкнуть*, которые должны быть определены достаточно хорошо для того, чтобы клиентская программа могла их использовать, а также на некоторую реализацию этих операций, функционирующую в соответствии с правилом удаления элементов такого стека: *последним пришел, первым ушел* (*last-in, first-out*, сокращенно *LIFO*).

### Программа 4.4 Интерфейс абстрактного типа данных стека

Используя то же самое общепринятое правило, что и в программе 4.3, мы определяем АД стека через объявление общедоступных функций. При этом предполагается, что представление стека и любой другой код, зависящий от реализации, являются приватными, чтобы можно было изменять реализации, не изменяя код клиентских программ. Кроме того, в этом интерфейсе применяется шаблон, что позволяет программам-клиентам использовать стеки, содержащие объекты любых классов (см. программы 4.5 и 4.6), а в реализациях использовать ключевое слово `Item` в качестве обозначения типа объектов стека (см. программы 4.7 и 4.8). Аргумент конструктора `STACK` задает максимальное количество элементов, которые можно поместить в стек.

```
template <class Item>
class STACK
{
private:
    // программный код, зависящий от реализации
```

```

public:
    STACK(int);
    int empty() const;
    void push(Item item);
    Item pop();
};

```

На рис. 4.1 показано, как изменяется содержимое стека в процессе выполнения серии операций *затолкнуть* и *вытолкнуть*. Каждая операция *затолкнуть* увеличивает размер стека на 1, а каждая операция *вытолкнуть* уменьшает размер стека на 1. На рисунке элементы стека перечисляются в порядке их помещения в стек, поэтому ясно, что самый правый элемент списка — это элемент, который находится на вершине стека и будет извлечен из стека, если следующей операцией будет операция *вытолкнуть*. В реализации элементы можно организовывать любым требуемым способом, однако при этом у программ-клиентов должна сохраняться иллюзия, что элементы организованы именно таким образом.

Как упоминалось в предыдущем разделе, для того чтобы можно было писать программы, использующие абстракцию стека, сначала необходимо определить интерфейс. С этой целью, как принято, объявляется совокупность общедоступных функций-членов, которые будут использоваться в реализациях класса (см. программу 4.4). Все остальные члены класса делаются приватными (**private**) и тем самым обеспечивается, что эти функции будут единственной связью между клиентскими программами и реализациями. В главах 1 и 3 мы уже видели достоинства определения абстрактных операций, на которых основаны требуемые вычисления. Сейчас мы рассматриваем механизм, позволяющий записывать программы, в которых применяются эти абстрактные операции. Для реализации такой абстракции, для сокрытия структуры данных и реализации от программы-клиента используется механизм классов. В разделе 4.3 рассматриваются примеры клиентских программ, использующих абстракцию стека, а в разделе 4.4 — соответствующие реализации.

Первая строка кода в программе 4.4 (интерфейс абстрактного типа данных стек) добавляет в этот класс шаблон C++, позволяющий клиентским программам задавать тип объектов, которые могут заноситься в стек.

L		L
A		L A
*	A	L
S		L S
T		L S T
I		L S T I
*	I	L S T
N		L S T N
*	N	L S T
F		L S T F
I		L S T F I
R		L S T F I R
*	R	L S T F I
S		L S T F I S
T		L S T F I S T
*	T	L S T F I S
*	S	L S T F I
O		L S T F I O
U		L S T F I O U
*	U	L S T F I O
T		L S T F I O T
*	T	L S T F I O
*	O	L S T F I
*	I	L S T F
*	F	L S T
*	T	L S
*	S	L
*	L	

**РИСУНОК 4.1 ПРИМЕР СТЕКА МАГАЗИННОГО ТИПА (ОЧЕРЕДИ, ФУНКЦИОНИРУЮЩЕЙ ПО ПРИНЦИПУ LIFO)**

*Этот список отображает результаты выполнения последовательности операций. Левый столбец представляет собой выполняемые операции (сверху вниз), где буква означает операцию затолкнуть, а звездочка — операцию вытолкнуть. В каждой строке отображается выполняемая операция, буква, извлекаемая при операции выталкивания, и содержимое стека после операции (от первой занесенной буквы до последней — слева направо).*

## Объявление

```
STACK<int> save(N)
```

определяет, что элементы стека `save` должны быть типа `int` (и что максимальное количество элементов, которое может вмещать стек, равно `N`). Программа-клиент может создавать стеки, содержащие объекты типа `float` или `char` или любого другого типа (даже типа `STACK`); для этого необходимо просто изменить параметр шаблона в угловых скобках. Мы можем считать, что в реализации указанный класс замещает класс `Item` везде, где он встречается.

В абстрактном типе данных назначение интерфейса состоит в том, чтобы служить в качестве соглашения между программой-клиентом и реализацией. Объявления функций обеспечивают соответствие между вызовами в клиентской программе и определениями функций в реализации. Но с другой стороны, интерфейс не содержит никакой информации о том, как должны быть реализованы функции, или хотя бы как они должны функционировать. Как мы можем объяснить клиентской программе, что такое стек? Для простых структур, подобных стеку, можно было бы открыть код, но ясно, что в общем случае такое решение не может являться эффективным. Чаще всего программисты прибегают к описаниям на английском языке и помещают их в документацию на программу.

Строгая трактовка этой ситуации требует полного описания того, как должны работать функции (с использованием формальной математической нотации). Такое описание иногда называют *спецификацией*. Разработка спецификации обычно является трудной задачей. Она должна описывать *любую* программу, реализующую функции, на математическом метаязыке, тогда как мы привыкли определять работу функций с помощью кода, написанного на языке программирования. На практике работа функций представляется в виде описаний на английском языке. Но давайте пойдём дальше, пока мы не углубились слишком далеко в гносеологические вопросы. В настоящей книге даются подробные примеры описаний на английском языке и по несколько реализаций для большинства рассматриваемых АТД.

Чтобы подчеркнуть, что наша спецификация абстрактного типа данных стек предоставляет достаточно информации для написания осмысленной клиентской программы, перед исследованием реализаций рассмотрим в разделе 4.3 две клиентские программы, использующие стеки магазинного типа.

## Упражнения

### ▷ 4.9 В последовательности

```
EAS*Y*QUE***ST***IO*N***
```

буква означает операцию *затолкнуть*, а звездочка — операцию *вытолкнуть*. Дайте последовательность значений, возвращаемых операциями *вытолкнуть*.

**4.10** Используя те же правила, что и в упражнении 4.9, вставьте звездочки в последовательность `EASY` таким образом, чтобы последовательность значений, возвращаемых операциями *вытолкнуть*, была следующей: (i) `EASY`; (ii) `YSAE`; (iii) `ASYE`; (iv) `AYES`; или в каждом случае докажите, что такая последовательность не возможна.

- 4.11 Предположим, что даны две последовательности. Разработайте алгоритм, позволяющий определить, можно ли к первой последовательности добавить звездочки так, чтобы эта последовательность, будучи интерпретированной как последовательность стековых операций (аналогично упражнению 4.10), дала в результате вторую последовательность.

## 4.3 Примеры программ-клиентов, использующих АТД стека

В последующих главах мы увидим огромное количество приложений со стеками. Сейчас в качестве вводного примера рассмотрим применение стеков для вычисления арифметических выражений. Например, предположим, что требуется найти значение простого арифметического выражения с операциями умножения и сложения целых чисел:

$$5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 )$$

Это вычисление включает сохранение промежуточных результатов: например, если сначала вычисляется  $9 + 8$ , придется сохранить результат 17 на время, пока, скажем, вычисляется  $4 * 6$ . Стек магазинного типа представляет собой идеальный механизм для сохранения промежуточных результатов таких вычислений.

Начнем с рассмотрения более простой задачи, где выражение, которое необходимо вычислить, имеет другую форму: знак операции стоит *после* двух своих аргументов, а не между ними. Как будет показано, в такой форме может быть представлено любое арифметическое выражение, причем форма носит название *постфиксной*, в отличие от *инфиксной* — обычной формы записи арифметических выражений. Вот постфиксное представление выражения из предыдущего абзаца:

$$5 \ 9 \ 8 \ + \ 4 \ 6 \ * \ * \ 7 \ + \ *$$

Форма записи, обратная постфиксной, называется *префиксной* или *Польской записью* (так как ее придумал польский логик Лукашевич (Lukasiewicz)).

При инфиксной записи чтобы отличить, например, выражение

$$5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 )$$

от выражения

$$( ( 5 * 9 ) + 8 ) * ( ( 4 * 6 ) + 7 )$$

требуется скобки; но при постфиксной (или префиксной) записи скобки не нужны. Чтобы увидеть почему, можно рассмотреть следующий процесс преобразования постфиксного выражения в инфиксное: все группы из двух операндов и следующим за ними знаком операции замещаются их инфиксными эквивалентами и заключаются в круглые скобки для демонстрации того, что этот результат может рассматриваться как операнд. То есть, группы  $ab^*$  и  $ab^+$  замещаются группами  $(a * b)$  и  $(a + b)$ , соответственно. Затем то же самое преобразование выполняется с полученным выражением, продолжая процесс до тех пор, пока не будут обработаны все операции. В нашем примере преобразование будет происходить следующим образом:

```

5 9 8 + 4 6 * * 7 + *
5 ( 9 + 8 ) ( 4 * 6 ) * 7 + *
5 ( ( 9 + 8 ) * ( 4 * 6 ) ) 7 + *
5 ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) *
( 5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) )

```

Таким способом в постфиксном выражении можно определить все операнды, связанные с любой операцией, поэтому необходимость в применении скобок отпадает.

С другой стороны, с помощью стека можно действительно выполнить эти операции и вычислить значение любого постфиксного выражения (см. рис. 4.2). Перемещаясь слева направо, мы интерпретируем каждый операнд как команду "занести операнд в стек", а каждый знак операции — как команды "извлечь из стека два операнда, выполнить операцию и занести результат в стек". Программа 4.5 является реализацией этого процесса на языке C++. Обратите внимание, что, поскольку АТД `Стек` создан на основе шаблона, один и тот же код может использоваться и для создания стека целых чисел в этой программе, и стека символов в программе 4.6.

#### Программа 4.5 Вычисление постфиксного выражения

Эта программа-клиент для стека магазинного типа считывает любое постфиксное выражение, включающее умножение и сложение целых чисел, затем вычисляет это выражение и распечатывает полученный результат. Промежуточные результаты она хранит в стеке целых чисел; при этом предполагается, что интерфейс из программы 4.4 реализован в файле `STACK.cxx` как класс, созданный на основе шаблона.

Когда встречаются операнды, они заносятся в стек; когда встречаются знаки операций, из стека извлекаются два верхних элемента, над ними выполняется данная операция и результат снова заносится в стек. Порядок, в котором две операции `pop()` выполняются в выражениях данного кода, в языке C++ не определен, поэтому код для некоммутативных операций, таких как вычитание или деление, был бы более сложным.

В программе неявно предполагается, что целые числа и знаки операций ограничены какими-нибудь другими символами (скажем, пробелами), но программа не проверяет корректность входных данных. Последний оператор `if` и цикл `while` выполняют вычисление, подобное тому, что выполняет функция `atoi` языка C++, которая преобразует строки в коде ASCII в целые числа, готовые для вычислений. Когда встречается новая цифра, накопленный результат умножается на 10 и к нему прибавляется эта цифра.

```

#include <iostream.h>
#include <string.h>
#include "STACK.cxx"
int main(int argc, char *argv[])
{ char *a = argv[1]; int N = strlen(a);
  STACK<int> save(N);
  for (int i = 0; i < N; i++)
  {

```

5	5			
9	5	9		
8	5	9	8	
+	5	17		
4	5	17	4	
6	5	17	4	6
*	5	17	24	
*	5	408		
7	5	408	7	
+	5	415		
*		2075		

#### РИСУНОК 4.2 ВЫЧИСЛЕНИЕ ПОСТФИКСНОГО ВЫРАЖЕНИЯ

*Эта последовательность операций показывает, как стек используется для вычисления постфиксного выражения  $5\ 9\ 8\ +\ 4\ 6\ *\ * 7\ +\ *$ . Выражение обрабатывается слева направо и, если встречается число, оно заносится в стек; если же встречается знак операции, то эта операция выполняется над двумя верхними числами стека и результат опять заносится в стек.*

```

    if (a[i] == '+')
        save.push(save.pop() + save.pop());
    if (a[i] == '*')
        save.push(save.pop() * save.pop());
    if ((a[i] >= '0') && (a[i] <= '9'))
        save.push(0);
    while ((a[i] >= '0') && (a[i] <= '9'))
        save.push(10*save.pop() + (a[i++] - '0'));
}
cout << save.pop() << endl;
}

```

Постфиксная запись и стек магазинного типа обеспечивают естественный способ организации ряда вычислительных процедур. В некоторых калькуляторах и языках программирования метод вычислений явно базируется на постфиксной записи и стековых операциях — при выполнении любой операции ее аргументы извлекаются из стека, а результат возвращается в стек.

Примером такого языка является язык PostScript. Это заверченный язык программирования, в котором программы пишутся в постфиксном виде и интерпретируются с помощью внутреннего стека, точно как в программе 4.5. Хотя мы не можем осветить здесь все аспекты этого языка (*см. раздел ссылок*), он достаточно прост и мы можем изучить некоторые реальные программы, чтобы оценить полезность постфиксной записи и абстракции стека магазинного типа. Например, строка

```
5 9 8 add 4 6 mul mul 7 add mul
```

является программой на языке PostScript! Программа на языке PostScript состоит из операций (таких, как **add** и **mul**) и операндов (таких, как целые числа). Программу на этом языке мы интерпретируем так же, как делали это в программе 4.5 — читая ее слева направо. Если встречается операнд, он заносится в стек; если встречается знак операции, из стека извлекаются операнды для этой операции (если они есть), а затем результат (если он есть) заносится в стек. Таким образом, на рис. 4.2 полностью описан процесс выполнения этой программы: после выполнения программы в стеке остается число **2075**.

В языке PostScript имеется несколько простых функций, которые служат инструкциями для абстрактного графопостроителя; кроме того, можно определять и собственные функции. Эти функции с аргументами, расположенными в стеке, вызываются таким же способом, как и любые другие функции. Например, следующий код на языке PostScript

```
0 0 moveto 144 hill 0 72 moveto 72 hill stroke
```

соответствует последовательности действий "вызвать функцию **moveto** с аргументами 0 и 0, затем вызвать функцию **hill** с аргументом 144" и т.д. Некоторые операции относятся непосредственно к самому стеку. Например, операция **dup** дублирует элемент в верхушке стека; поэтому, например, код

```
144 dup 0 rlineto 60 rotate dup 0 rlineto
```

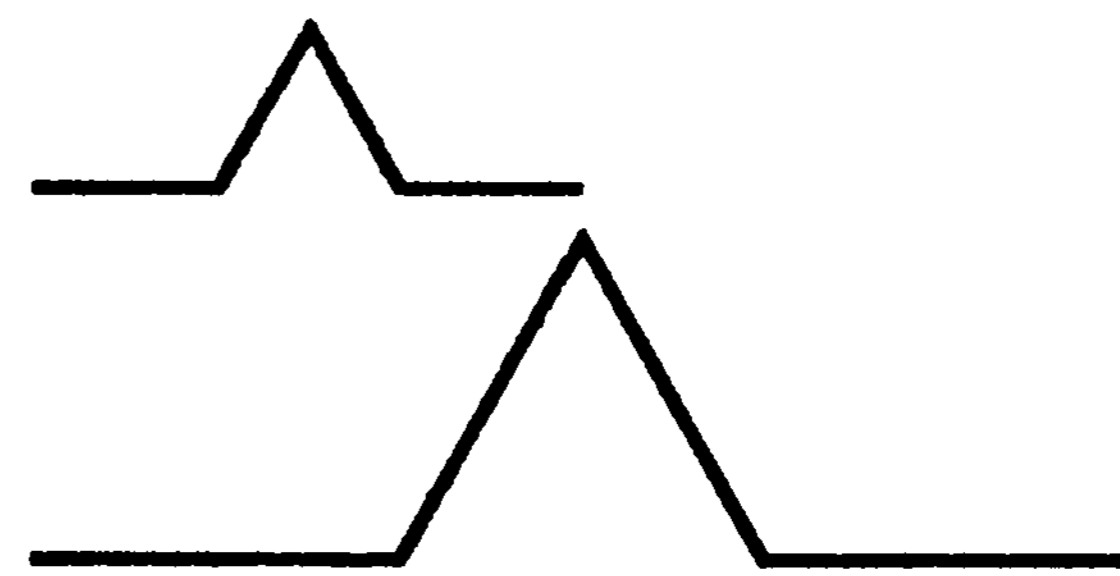


соответствует следующей последовательности действий: вызвать функцию **rlineo** с аргументами 144 и 0, затем вызвать функцию **rotate** с аргументом 60, затем вызвать функцию **rlineo** с аргументами 144 и 0 и т.д. В программе PostScript, показанной на рис. 4.3, определяется и используется функция **hill**. Функции в языке PostScript подобны макросам: последовательность `/hill { A } def` делает имя **hill** эквивалентным последовательности операций внутри фигурных скобок. Рисунок 4.3 представляет пример программы PostScript, в которой определяется функция и вычерчивается простая диаграмма.

В данном контексте наш интерес к языку PostScript объясняется тем, что этот широко используемый язык программирования базируется на абстракции стека магазинного типа. Действительно, в аппаратных средствах многих компьютеров реализованы основные стековые операции, поскольку они являются естественным воплощением механизма вызова функций: при входе в процедуру текущее состояние программной среды сохраняется путем занесения информации в стек; при выходе из процедуры состояние программной среды восстанавливается за счет извлечения информации из стека. Как будет показано в главе 5, эта связь между магазинными стеками и программами, которые организованы в виде функций, обращающихся к другим функциям, отражает важную модель вычислительного процесса.

Возвращаясь к первоначальной задаче, отметим, что, как иллюстрирует рис. 4.4, магазинный стек можно также использовать для преобразования инфиксного арифметического выражения с круглыми скобками в постфиксную форму. При выполнении этого преобразования *знаки операций* заносятся в стек, а сами операнды просто передаются в выходной поток программы. Правая скобка показывает, что два последних числа на выходе программы являются аргументами операции, знак которой занесен в стек последним. Поэтому этот знак операции можно извлечь из стека и также передать в выходной поток программы.

Программа 4.6 представляет собой реализацию этого процесса. Обратите внимание, что аргумен-



```

/hill {
    dup 0 rlineo
    60 rotate
    dup 0 rlineo
    -120 rotate
    dup 0 rlineo
    60 rotate
    dup 0 rlineo
    pop
} def
0 0 moveto
144 hill
0 72 moveto
72 hill
stroke

```

#### РИСУНОК 4.3 ПРОСТАЯ ПРОГРАММА НА ЯЗЫКЕ POSTSCRIPT

*Диаграмма в верхней части рисунка была вычерчена программой PostScript, расположенной под ней. Программа является постфиксным выражением, в котором используются встроенные функции **moveto**, **rlineo**, **rotate**, **stroke** и **dup**; в ней также используется определяемая пользователем функция **hill** (см. текст). Графические команды являются инструкциями графопостроителю: команда **moveto** устанавливает печатающую головку устройства на заданную позицию страницы (координаты даются в пунктах, равных 1/72 дюйма); команда **rlineo** перемещает печатающую головку на новую позицию, координаты которой задаются относительно текущей позиции, и тем самым она добавляет очередной участок к пройденному пути; команда **rotate** изменяет направление движения печатающей головки, заставляя ее повернуть влево на заданное число градусов; команда **stroke** чертит пройденный путь.*

ты в постфиксном выражении расположены в том же самом порядке, что и в инфиксном выражении. Забавно также отметить, что левые скобки в инфиксном выражении не нужны. Однако они необходимы, если существуют операции, имеющие разное количество операндов (см. упражнение 4.14).

Помимо того, что алгоритм, разработанный в данном разделе для вычисления инфиксных выражений, предоставил два разных примера использования абстракции стека, он и сам по себе является упражнением по абстракциям. Во-первых, входные данные преобразуются в промежуточное представление (постфиксное выражение). Во-вторых, для интерпретации и вычисления этого выражения имитируется работа абстрактной машины, функционирующей на основе стека. В целях эффективности и мобильности эта же схема работы наследуется многими современными компиляторами: задача компиляции программы на C++ для некоторого компьютера разбивается на две задачи, которые концентрируются вокруг промежуточного представления. Поэтому задача трансляции программы отделяется от задачи выполнения этой программы, точно так же как это делалось в данном разделе. В разделе 5.7 будет показано родственное, но другое промежуточное представление.

#### Программа 4.6 Преобразование из инфиксной формы в постфиксную

Эта программа является еще одним примером программы-клиента для стека магазинного типа. В данном случае стек содержит символы. Для преобразования  $(A+B)$  в постфиксную форму  $AB+$  левая скобка игнорируется, в выходной поток записывается  $A$ , в стеке запоминается знак  $+$ , в выходной поток записывается  $B$  и затем при встрече правой скобки из стека извлекается знак  $+$  и он же записывается в выходной поток.

```
#include <iostream.h>
#include <string.h>
#include "STACK.cxx"
int main(int argc, char *argv[])
{ char *a = argv[1]; int N = strlen(a);
  STACK<char> ops(N);
  for (int i = 0; i < N; i++)
  {
    if (a[i] == ')')
      cout << ops.pop() << " ";
    if ((a[i] == '+') || (a[i] == '*'))
      ops.push(a[i]);
    if ((a[i] >= '0') && (a[i] <= '9'))
      cout << a[i] << " ";
  }
  cout << endl;
}
```

```
(
5 5
* *
( *
( *
( *
9 9 *
+ * +
8 8 * +
) + *
* * *
( * *
4 4 * *
* * * *
6 6 * * *
) * * *
) * *
+ * +
7 7 * +
) + *
) *
```

#### РИСУНОК 4.4 ПРЕОБРАЗОВАНИЕ ИНФИКСНОГО ВЫРАЖЕНИЯ В ПОСТФИКСНОЕ

*Данная последовательность операций демонстрирует использование стека для преобразования инфиксного выражения*

*$(5*(((9+8)*(4*6))+7))$  в его постфиксную форму  $5\ 9\ 8\ +\ 4\ 6\ *\ * \ 7\ +\ *$ .*

*Выражение обрабатывается слева направо: если встречается число, оно записывается в выходной поток; если встречается левая скобка, она игнорируется; если встречается знак операции, он заносится в стек; и если встречается правая скобка, то в выходной поток записывается знак операции, находящийся на верхушке стека.*

Это приложение также иллюстрирует достоинства абстрактных типов данных и шаблонов C++. Здесь используются два разных стека, один из которых содержит объекты типа `char` (знаки операций), а другой — объекты типа `int` (операнды). С помощью АД в виде шаблонного класса, определенного в программе 4.4, можно даже объединить две рассмотренных клиентских программы в одну (см. упр. 4.19). Несмотря на привлекательность решения, стоит сознавать, что это, по-видимому, не самый лучший выбор, поскольку различные реализации могут отличаться своей производительностью и, возможно, не следовало бы априори решать, что в обоих случаях будет применяться одна и та же реализация. Действительно, главное внимание уделяется реализации и производительности, и сейчас мы приступим к рассмотрению этих тем применительно к стеку магазинного типа.

## Упражнения

▷ 4.12 Преобразуйте в постфиксное выражение

$( 5 * ( ( 9 * 8 ) + ( 7 * ( 4 + 6 ) ) ) ) .$

▷ 4.13 Таким же способом, как на рис. 4.2, покажите содержимое стека при вычислении программой 4.5 следующего выражения

$5 \ 9 * 8 \ 7 \ 4 \ 6 + * \ 2 \ 1 \ 3 * + * + * .$

▷ 4.14 Расширьте программы 4.5 и 4.6 таким образом, чтобы они включали операции — (вычитание) и / (деление).

4.15 Расширьте решение упражнения 4.14 таким образом, чтобы оно включало унарные операции — (отрицание) и  $\$$  (извлечение квадратного корня). Кроме того, измените механизм абстрактного стека в программе 4.5 так, чтобы можно было использовать числа с плавающей точкой. Например, имея в качестве исходного выражение

$( - ( - 1 ) + \$ ( ( - 1 ) * ( - 1 ) - ( 4 * ( - 1 ) ) ) ) / 2$

программа должна выдать число 1.618034.

4.16 Напишите на языке PostScript программу, которая чертит следующую

фигуру: 

● 4.17 Методом индукции докажите, что программа 4.5 правильно вычисляет любое постфиксное выражение.

○ 4.18 Напишите программу, которая с использованием стека магазинного типа преобразует постфиксное выражение в инфиксное.

○ 4.19 Объедините программы 4.5 и 4.6 в один модуль, в котором будут использоваться два разных АД: стек целых чисел и стек операций (знаков операций).

●● 4.20 Реализуйте компилятор и интерпретатор для языка программирования, в котором каждая программа состоит из одного арифметического выражения. Выражению предшествует ряд операторов присваивания с арифметическими выражениями, состоящими из целых чисел и переменных, обозначаемых одиночными символами нижнего регистра. Например, получив входные данные

$(x = 1)$

$(y = (x + 1))$

$(( (x + y) * 3 ) + ( 4 * x ) )$

программа должна вывести число 13.

## 4.4 Реализации АТД стека

В данном разделе рассматриваются две реализации АТД стека: в одной используются массивы, в другой — связные списки. Эти реализации получаются в результате простого применения базовых инструментальных средств, рассмотренных в главе 3. Мы полагаем, что они различаются только своей производительностью (быстродействием).

### Программа 4.7 Реализация стека магазинного типа на базе массива

В этой реализации  $N$  элементов стека хранятся как элементы массива:  $s[0], \dots, s[N-1]$ , начиная с первого занесенного элемента и завершая последним. Верхушкой стека (позицией, в которую будет заноситься следующий элемент стека) является позиция  $s[N]$ . Максимальное количество элементов, которое может вмещать стек, программа-клиент передает в виде аргумента в конструктор **STACK**, размещающий в памяти массив данного размера; однако код не проверяет такие ошибки, как помещение элемента в переполненный стек (или выталкивание элемента из пустого стека).

```
template <class Item>
class STACK
{
private:
    Item *s; int N;
public:
    STACK(int maxN)
        { s = new Item[maxN]; N = 0; }
    int empty() const
        { return N == 0; }
    void push(Item item)
        { s[N++] = item; }
    Item pop()
        { return s[--N]; }
};
```

Если для представления стека применяется массив, то все функции, объявленные в программе 4.4, реализуются очень просто, что видно из программы 4.7. Элементы заносятся в массив в точности так, как показано на рис. 4.1, при этом отслеживается индекс верхушки стека. Выполнение операции *затолкнуть* означает запоминание элемента в позиции массива, указываемой индексом верхушки стека, а затем увеличение этого индекса на единицу; выполнение операции *вытолкнуть* означает уменьшение индекса на единицу и извлечение элемента, обозначенного этим индексом. Операция *создать* (конструктор) осуществляет размещение массива указанного размера, а операция *проверить, пуст ли стек* проверяет, не равен ли индекс нулю. Скомпилированная вместе с клиентской программой (такой, как программа 4.5 или 4.6), эта реализация обеспечивает рациональный и эффективный стек магазинного типа.

Один потенциальный недостаток применения массива для представления стека известен многим: как это обычно бывает со структурами данных, создаваемыми на базе массивов, до использования массива необходимо знать его максимальный размер, чтобы распределить под него оперативную память. В рассматриваемой реализации эта информация передается в аргументе конструктора. Данный недостаток — результат выбора реализации на базе массива; он не является неотъемлемой частью АТД

стека. Зачастую трудно определить максимальное число элементов, которое программа будет заносить в стек: если выбрать слишком большое число, то такая реализация будет неэффективно использовать оперативную память, а это может быть нежелательно в тех приложениях, где память является ценным ресурсом. Если выбрать слишком маленькое число, программа может вообще не работать. Применение АТД дает возможность рассматривать другие варианты и изменять реализацию без изменения кода клиентских программ.

Например, чтобы стек мог элегантно увеличиваться и уменьшаться, можно было бы отдать предпочтение связному списку, как в программе 4.8. Стек организован в обратном порядке по сравнению с реализацией на базе массива — начиная с последнего занесенного элемента и завершая первым. Этот (см. рис. 4.5) позволяет более просто реализовать базовые стековые операции. Чтобы *вытолкнуть* элемент, удаляется узел в начале списка и извлекается из него элемент; чтобы *втолкнуть* элемент, создается новый узел и добавляется в начало списка. Поскольку все операции связного списка выполняются в начале списка, узел, соответствующий верхушке стека, не требуется.

Программа 4.8 не проверяет такие ошибки, как попытка извлечения элемента из пустого стека, занесения элемента в переполненный стек или выход за пределы памяти. В отношении проверки двух последних условий имеются две возможности. Их можно трактовать как независимые ошибки и отслеживать количество элементов в списке, для чего при каждом занесении в стек проверять, что счетчик не превышает значение, переданное конструктору в качестве аргумента, и что `new` выполняется успешно. Вполне уместно занять позицию, при которой не требуется заранее знать максимальный размер стека, и, игнорируя аргумент конструктора (см. упражнение 4.24), сообщать о том, что стек переполнен, только тогда, когда `new` завершается с ошибкой.

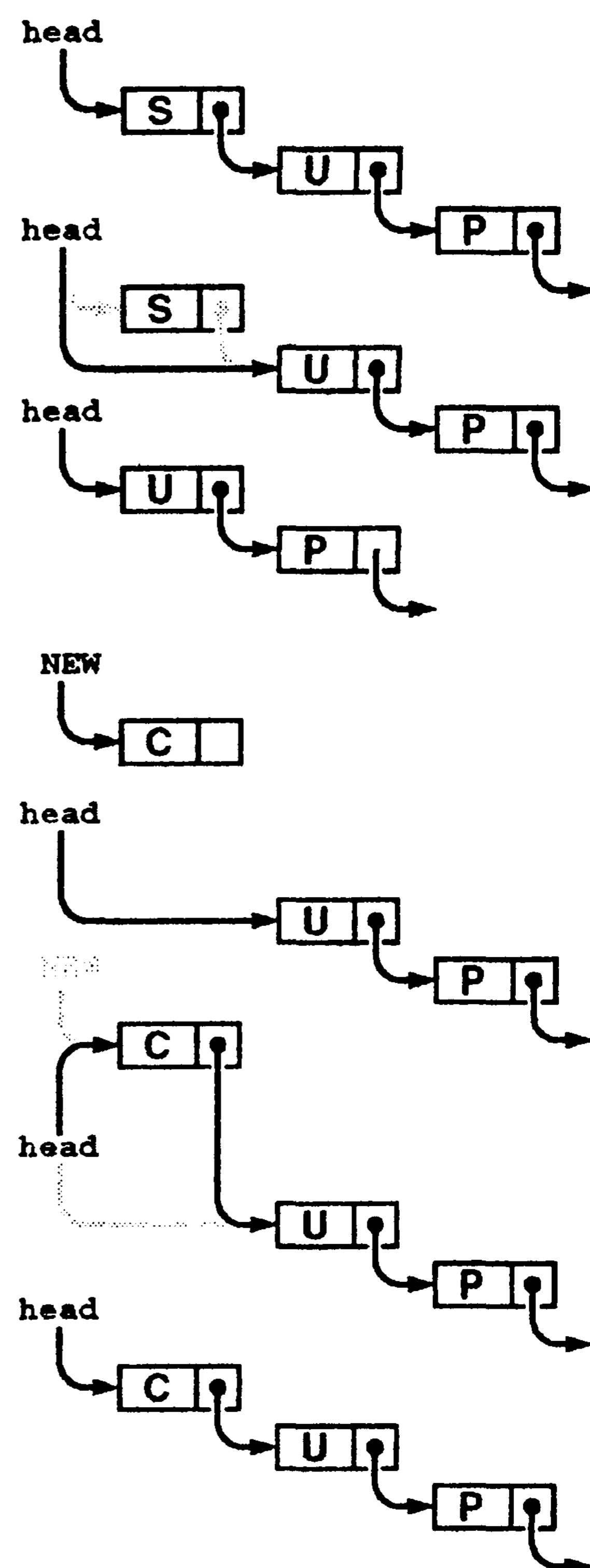


РИСУНОК 4.5 СТЕК МАГАЗИННОГО ТИПА НА БАЗЕ СВЯЗНОГО СПИСКА

Стек представлен указателем *head*, который указывает на первый (последний вставленный) элемент. Чтобы вытолкнуть элемент из стека (*top*), удаляется элемент в начале списка, устанавливая *head* равным указателю связи из этого элемента. Для заталкивания в стек нового элемента (*bottom*), он присоединяется в начало списка путем установки его поля связи так, чтобы оно указывало на *head*, а указателя *head* — так, чтобы он указывал на новый элемент.

---

**Программа 4.8 Реализация стека магазинного типа на базе связанного списка**


---

В этой программе АДТ реализуется с помощью связанного списка. Представление данных для узлов связанного списка организовано традиционным способом (см. главу 3) и включает конструктор для узлов, который заполняет каждый новый узел данным элементом и его связью.

```

template <class Item>
class STACK
{
private:
    struct node
    { Item item; node* next;
      node(Item x, node* t)
      { item = x; next = t; }
    };
    typedef node *link;
    link head;
public:
    STACK(int)
    { head = 0; }
    int empty() const
    { return head == 0; }
    void push(Item x)
    { head = new node(x, head); }
    Item pop()
    { Item v = head->item; link t = head->next;
      delete head; head = t; return v; }
};

```

---

Программы 4.7 и 4.8 представляют две различные реализации одно и того же АДТ. Можно заменять одну реализацию другой, не делая *никаких* изменений в клиентских программах, подобных тем, которые рассматривались в разделе 4.3. Они отличаются только производительностью. Реализация на базе массива использует объем памяти, необходимый для размещения максимального числа элементов, которые может вместить стек в процессе вычислений; реализация на базе списка использует объем памяти пропорционально количеству элементов, но при этом всегда расходует дополнительную память для одной связи на каждый элемент, а также дополнительное время на распределение памяти при каждой операции *затолкнуть* и освобождение памяти при каждой операции *вытолкнуть*. Если требуется стек больших размеров, который обычно заполняется практически полностью, по-видимому, предпочтение стоит отдать реализации на базе массива. Если же размер стека варьируется в широких пределах и присутствуют другие структуры данных, которым требуется память, не используемая во время, когда в стеке находится несколько элементов, предпочтение следует отдать реализации на базе связанного списка.

По ходу изложения материала будет показано, что эти же самые соображения относительно использования оперативной памяти справедливы для многих реализаций АДТ. Разработчики часто оказываются в ситуациях, когда приходится выбирать между возможностью быстрого доступа к любому элементу при необходимости заранее указывать максимальное число требуемых элементов (в реализации на базе массивов) и гибким использованием памяти (пропорционально количеству элементов, находящих-

ся в стеке) при отсутствии быстрого доступа к любому элементу (в реализации на базе связанных списков).

Помимо этих основных соображений относительно использования памяти, обычно больше всего интересуют различия в производительности разных реализаций АД, связанные со временем выполнения. В данном случае различия между двумя рассмотренными реализациями незначительны.

**Лемма 4.1** *Используя либо массивы, либо связанные списки, для АД стека магазинного типа можно реализовать операции **втолкнуть** и **вытолкнуть**, имеющие постоянное время выполнения.*

Этот факт является непосредственным следствием внимательного изучения программ 4.7 и 4.8.

То, что в реализациях на базе массива и связанного списка элементы стека хранятся в разном порядке, для клиентских программ не имеет никакого значения. В реализациях могут использоваться какие угодно структуры данных, лишь бы они создавали впечатление абстрактного стека магазинного типа. В обоих случаях реализации способны создавать впечатление эффективного абстрактного объекта, который может выполнять необходимые операции с помощью всего лишь нескольких машинных инструкций. Цель всей книги — отыскать структуры данных и эффективные реализации для других важных АД.

Реализация на базе связанного списка создает впечатление стека, который может увеличиваться неограниченно. Однако в практических условиях такой стек невозможен: рано или поздно, когда запрос на выделение еще некоторого объема памяти не сможет быть удовлетворен, `new` сгенерирует исключение. Можно также организовать стек на базе массива, который будет увеличиваться и уменьшаться динамически: когда стек заполняется наполовину, размер массива увеличивается в два раза, а когда стек становится наполовину пустым, размер массива уменьшается в два раза. Детали реализации упомянутой стратегии мы оставляем в качестве упражнения в главе 14, где этот процесс подробно рассматривается для более сложных применений.

## Упражнения

- ▷ **4.21** Определите содержимое элементов `s[0]`, ... , `s[4]` после выполнения с помощью программы 4.7 операций, показанных на рис. 4.1.
- **4.22** Предположим, что вы изменяете интерфейс стека магазинного типа, заменяя операцию *проверить, пуст или стек* операцией *подсчитать*, которая возвращает количество элементов, находящихся в данный момент в структуре данных. Реализуйте операцию *подсчитать* для представления на базе массива (программа 4.7) и представления на базе связанного списка (программа 4.8).
- 4.23** Измените реализацию стека магазинного типа на базе массива (программа 4.7) так, чтобы она вызывала функцию-член `error()`, если клиентская программа пытается выполнить операцию *вытолкнуть*, когда стек пуст, или операцию *затолкнуть*, когда стек переполнен.
- 4.24** Измените реализацию стека магазинного типа на базе связанного списка (программа 4.8) таким образом, чтобы она вызывала функцию-член `error()`, если кли-

ентская программа пытается выполнить операцию *вытолкнуть*, когда стек пуст, или операцию *затолкнуть*, когда отсутствует свободная память при вызове `new`.

**4.25** Измените реализацию стека магазинного типа на базе связного списка (программа 4.8) так, чтобы для создания списка она использовала массив индексов (см. рис. 3.4).

**4.26** Напишите реализацию стека магазинного типа на базе связного списка, в которой элементы списка хранятся начиная с первого занесенного элемента и завершая последним занесенным элементом. Потребуется использовать двухсвязный список.

- **4.27** Разработайте АТД, который содержит два разных стека магазинного типа. Воспользуйтесь реализацией на базе массива. Один стек расположите в начале массива, другой — в конце. (Если клиентская программа работает так, что один стек увеличивается, в то время как другой уменьшается, эта реализация будет занимать меньший объем памяти, нежели альтернативные варианты.)
- **4.28** Реализуйте функцию вычисления инфиксных выражений, состоящих из целых чисел. Она должна включать программы 4.5 и 4.6, а также использовать АТД из упражнения 4.27. *Примечание:* потребуется рассмотреть случай, когда оба стека содержат элементы одного и того же типа.

## 4.5 Создание нового АТД

Разделы с 4.2 по 4.4 содержат пример полного кода программы на C++ для одной из наиболее важных абстракций: стека магазинного типа. В *интерфейсе* из раздела 4.2 определены основные операции; *клиентские программы* из раздела 4.3 могут использовать эти операции независимо от их реализации, а вот *реализация* из раздела 4.4 обеспечивает конкретное представление и программный код АТД.

Создание нового АТД часто сводится к следующему процессу. Начав с разработки клиентской программы, предназначенной для решения прикладной задачи, определяются операции, которые считаются наиболее важными: какие операции *хотелось бы* выполнять над данными? Затем определяется интерфейс и записывается код программы-клиента с целью проверки гипотезы о том, что использование АТД упростит реализацию клиентской программы. Затем выполняется анализ, можно ли достаточно эффективно реализовать операции в рамках АТД. Если нет, придется поискать источник неэффективности, а затем применить интерфейс, поместив в него операции, более подходящие для эффективной реализации. Поскольку модификация влияет и на клиентскую программу, ее потребуется соответствующим образом изменить. Выполнив несколько модификаций интерфейса и клиентской программы, получаем работающую клиентскую программу и работающую реализацию; интерфейс "замораживается" в таком состоянии, т.е. он больше не изменяется. С этого момента разработка клиентских программ и реализаций выполняется отдельно: можно писать другие клиентские программы, использующие тот же самый АТД (скажем, для целей дополнительного тестирования), можно записывать другие реализации, равно как и сравнивать производительность различных реализаций.

В других ситуациях можно начать с определения АТД. Подобный подход может породить следующие вопросы: какие базовые операции над доступными данными



могут потребоваться клиентским программам? Какие операции можно реализовать эффективно? Завершив разработку реализации, можно проверить ее эффективность с помощью клиентских программ. Перед окончательным "замораживанием" интерфейса, возможно, потребуется дополнительная модификация и тесты.

В главе 1 подробно рассматривался пример, где размышления о том, какой уровень абстракции использовать, помогли отыскать эффективный алгоритм решения сложной задачи. Теперь посмотрим как обсуждаемый в настоящей главе обобщенный подход применяется для инкапсуляции абстрактных операций из главы 1.

В программе 4.9 определен интерфейс, включающий две операции (в дополнение к операции *создать* (*construct*)). Похоже, что эти операции описывают алгоритмы связности, рассмотренные в главе 1, на высоком уровне абстракции. Какие бы базовые алгоритмы и структуры данных не лежали в основе, необходимо иметь возможность проверки связности двух узлов, а также описания, что конкретные два узла являются связанными.

#### Программа 4.9 Интерфейс АТД "Отношения эквивалентности"

Интерфейс АТД построен так, чтобы было удобно записывать код, в точности соответствующий принятому решению рассматривать алгоритм связности в виде класса, поддерживающего три абстрактных операции: *инициализировать* (*initialize*) абстрактную структуру данных для отслеживания связей между заданным числом узлов; *найти* (*find*), являются ли два узла связанными; и *соединить* (*unite*) два данных узла и считать их с этого момента связанными.

```
class UF
{
    private:
        // программный код, зависящий от реализации
    public:
        UF(int);
        int find(int, int);
        void unite(int, int);
};
```

Программа 4.10 — это клиентская программа, которая для решения задачи связности использует АТД с интерфейсом, представленным в программе 4.9. Одно из преимуществ этого АТД состоит в относительной простоте понимания программы, поскольку она написана с использованием абстракций, позволяющих естественно представить процесс вычислений.

Программа 4.11 является реализацией интерфейса *union-find*, который определен в программе 4.9. В этой реализации (см. раздел 1.3) применяется бор деревьев, в основе которого лежат два массива, представляющие известную информацию о связях. В разных алгоритмах, рассмотренных в главе 1, используются различные реализации АТД, причем их можно тестировать независимо друг от друга, не изменяя клиентскую программу.

Этот АТД приводит к созданию несколько менее эффективных программ, нежели программа связности из главы 1, поскольку он не обладает тем преимуществом упомянутой программы, что в ней каждой операции *соединение* (*union*) непосредственно предшествует операция *поиск* (*find*). Дополнительные издержки подобного рода являются платой за переход к более абстрактному представлению. В данном случае

существует множество способов устранения этого недостатка, например, за счет более сложной реализации интерфейса (см. упражнение 4.30). На практике пути бывают очень короткими (особенно, если применяется сжатие путей), так что в данном случае дополнительные издержки, по-видимому, окажутся незначительными.

#### **Программа 4.10 Клиентская программа для АТД "Отношения эквивалентности"**

АТД из программы 4.9 позволяет отделить алгоритм связности от реализации `union-find`), тем самым делая его более доступным.

```
#include <iostream.h>
#include <stdlib.h>
#include "UF.cxx"
int main(int argc, char *argv[])
{ int p, q, N = atoi(argv[1]);
  UF info(N);
  while (cin >> p >> q)
    if (!info.find(p, q))
    {
      info.unite(p, q);
      cout << " " << p << " " << q << endl;
    }
}
```

Комбинация программ 4.10 и 4.11 с точки зрения функциональности эквивалентна программе 1.3; однако, разбиение программы на две части является более эффективным подходом, так как:

- позволяет отделять решение задачи высокого уровня (задачи связности) от решения задачи низкого уровня (задачи `union-find`) и решать эти две задачи независимо
- предоставляет естественный способ сравнения различных алгоритмов и структур данных, применяемых при решении этой задачи
- определяет с помощью интерфейса способ проверки ожидаемой работы программного обеспечения
- обеспечивает механизм, позволяющий совершенствовать представления (переходить к новым структурам данных и новым алгоритмам) без каких-либо изменений кода клиентских программ
- предоставляет абстракцию, которую можно использовать при построении других алгоритмов

Сказанное выше справедливо для многих задач, с которыми приходится сталкиваться при разработке компьютерных программ, так что эти базовые принципы построения абстрактных типов данных применяются исключительно широко.

#### **Программа 4.11 Реализация АТД "Отношения эквивалентности"**

Этот код взвешенного быстрого объединения из главы 1 является реализацией интерфейса из программы 4.9. В данной реализации код упакован в форму, удобную для его использования в других приложениях. Приватная перегруженная функция-член `find` реализует обход дерева вплоть до его корня.

```

class UF
{
private:
    int *id, *sz;
    int find(int x)
        { while (x != id[x]) x = id[x]; return x; }
public:
    UF(int N)
        {
            id = new int[N]; sz = new int[N];
            for (int i = 0; i < N; i++)
                { id[i] = i; sz[i] = 1; }
        }
    int find(int p, int q)
        { return (find(p) == find(q)); }
    void unite(int p, int q)
        { int i = find(p), j = find(q);
          if (i == j) return;
          if (sz[i] < sz[j])
              { id[i] = j; sz[j] += sz[i]; }
          else { id[j] = i; sz[i] += sz[j]; }
        }
};

```

В коде программы 4.11 смешаны интерфейс и реализация; поэтому он не допускает отдельную компиляцию клиентских программ и реализаций. Для того чтобы разные реализации могли использовать один и тот же интерфейс, программу можно, как в разделе 3.1, разделить на три файла следующим образом.

Создать заголовочный файл с именем, скажем, **UF.h**, который будет содержать объявление класса, представление данных и объявления функций, но не определения функций. В рассматриваемом примере этот файл будет содержать код программы 4.9, в который также включается представление данных (приватные объявления из программы 4.11). Затем необходимо сохранить определения функций в отдельном файле **.схх**, который будет также содержать директиву **include** для файла **UF.h** (как это имеет место в любой клиентской программе). При таком порядке вещей появляется возможность отдельной компиляции клиентских программ и реализаций. В самом деле, определение любой функции-члена класса можно сохранить в отдельном файле, если только функция-член объявляется в классе, а в определении функции перед ее именем помещается имя класса и знак **::**. Например, определение функции **find** в нашем примере следовало бы записать так:

```

int UF::find(int p, int q)
{ return (find(p) == find(q)); }

```

К сожалению, при отдельной компиляции разные компиляторы налагают различные требования на реализацию шаблонов. Проблема заключается в том, что компилятор не может создать код для функции-члена, не зная типа параметра шаблона, который недоступен, так как определен в главной программе. Определение функций-членов внутри объявлений классов позволяет избегать подобного рода головоломок, связанных с компиляцией.

Однако использование трех файлов, по-прежнему, не является идеальным, поскольку представление данных, которое в действительности является частью реализации, хранится в одном файле с интерфейсом. С помощью ключевого слова **private** можно предотвратить доступ к нему со стороны клиентских программ, но если провести в реализации изменения, которые, в свою очередь, потребуют изменений в представлении данных, придется изменить файл **.h** и перекомпилировать все клиентские программы. Во многих сценариях разработки программного обеспечения может отсутствовать информация о программах-клиентах, так что это может оказаться весьма затруднительным. В некоторых сценариях такая организация может иметь смысл. В случае очень большого и сложного АТД можно сначала остановиться на представлении данных и интерфейсе, а уже потом усадить команду программистов за разработку различных частей реализации. В этом случае общедоступная часть интерфейса послужит соглашением между программистами и клиентами, а приватная часть — соглашением сугубо между программистами. К тому же, когда необходимо найти самый лучший способ решения некоторой проблемы при условии, что должна использоваться некая конкретная структура данных, эта стратегия зачастую обеспечивает именно то, что и требуется. Таким образом, можно повысить производительность, изменяя лишь незначительную часть огромной системы.

В языке C++ имеется механизм, предназначенный специально для того, чтобы предоставить возможность писать программы с хорошо определенным интерфейсом, позволяющим полностью отделять клиентские программы от реализаций. Этот механизм базируется на понятии *производных (derived) классов*, посредством которых можно дополнять или переопределять некоторые члены существующего класса. Включение ключевого слова **virtual** в объявление функции-члена означает, что эта функция *может* быть переопределена в производном классе; добавление последовательности символов **= 0** в конце объявления функции-члена указывает на то, что данная функция является *чистой виртуальной (pure virtual) функцией*, которая *должна* быть переопределена в любом производном классе. Производные классы обеспечивают удобный способ создания программ на основе работ других программистов и являются важным компонентом объектно-ориентированных программных систем.

*Абстрактный (abstract) класс* — это класс, все члены которого являются чистыми виртуальными функциями. В любом классе, порожденном от абстрактного класса, должны быть определены все функции-члены и любые необходимые приватные данные-члены; таким образом, по нашей терминологии абстрактный класс является интерфейсом, а любой класс, порожденный от него, является реализацией. Программы-клиенты могут использовать интерфейс, а система C++ может обеспечить соблюдение соглашений между клиентами и реализациями, даже когда клиенты и реализации функций компилируются отдельно. Например, в программе 4.12 показан абстрактный класс **uf** для отношений эквивалентности; если изменить первую строку программы 4.11 следующим образом:

```
class UF : public class uf
```

то она будет указывать на то, что класс **UF** является производным от класса **uf** и поэтому в нем определены (как минимум) все функции-члены класса **uf** — т.е. класс **UF** является реализацией интерфейса **uf**.

---

**Программа 4.12 Абстрактный класс для АД отношения эквивалентности**

---

Приведенный ниже код флорирует интерфейс АД отношения эквивалентности, который обеспечивает полное разделение клиентов и реализаций (см. текст).

```
class uf
{
    public:
        virtual uf(int) = 0;
        virtual int find(int, int) = 0;
        virtual void unite(int, int) = 0;
};
```

---

К сожалению, использование абстрактных классов влечет за собой значительный дополнительный расход машинного времени при выполнении программы, поскольку каждый вызов виртуальной функции требует обращения к таблице указателей на функции-члены. Более того, компиляторы обладают намного меньшими возможностями в отношении оптимизации кода для абстрактных классов. Так как рассматриваемые в книге алгоритмы и структуры данных часто находятся в частях системы, критичных к производительности, возможно, упомянутая цена за гибкость, предоставляемую абстрактными классами, окажется слишком большой, чтобы ее платить.

Есть еще один способ действий — применить стратегию четырех файлов, при которой приватные части хранятся не в интерфейсе, но в отдельном файле. Например, в класс из программы 4.9 можно было бы добавить в начале строки

```
private:
#include "UFprivate.h"
```

после чего поместить строки

```
int *id, *sz;
int find(int);
```

в файл **UFprivate.h**. Эта стратегия позволяет аккуратно выделить четыре компонента (клиентскую программу, реализацию, представление данных и интерфейс) и обеспечивает максимальную гибкость в экспериментировании со структурами данных и алгоритмами.

Гибкость, которую можно достичь с помощью производных классов и стратегии "четырёх файлов", сохраняет возможность, хоть и непреднамеренного, тем не менее, нарушения соглашений между программами-клиентами и реализациями в отношении того, каким должен быть АД. Все эти механизмы гарантируют, что клиентские программы и реализации будут корректно компоноваться; однако они также зависят друг от друга и *функционируют* таким образом, что в общем случае нельзя описать это формально. Например, предположим, что какой-нибудь неосведомленный (или неопытный) программист нашел наш алгоритм взвешенного быстрого поиска трудным для понимания и решил заменить его алгоритмом быстрого объединения (или еще хуже, реализацией, которая даже не дает правильного решения). Мы всегда стремились к тому, чтобы такие изменения вносились легко, но в данном случае это может совершенно испортить программу-клиент в важном приложении, которое полагается на эту реализацию, обладающую хорошей производительностью для крупных за-

дач. Практика программирования полна подобных примеров, и от них очень трудно защититься.

Однако, такого рода рассуждения уводят нас к свойствам языков программирования, компиляторов, компоновщиков, сред выполнения программ, что весьма далеко от алгоритмов. Поэтому, чаще всего будем придерживаться простого, общепринятого разделения программы на два файла, где АДТ реализуется в виде классов C++, общедоступные функции-члены составляют интерфейс, а реализация объединяется с интерфейсом в отдельном файле, который включается в программы-клиенты и компилируется каждый раз, когда компилируются клиентские программы. Первопричина связана с тем, что реализация в виде класса — это удобное и компактное средство представления структур данных и алгоритмов. Если для какого-либо отдельного приложения потребуется большая гибкость, которая может быть обеспечена одним из только что упомянутых способов, можно соответствующим образом изменить структуры классов.

## Упражнения

- 4.29** Модифицируйте программу 4.11 так, чтобы в ней использовалось сжатие пути делением пополам.
- 4.30** Устраните упоминаемую в тексте неэффективность программы, добавив в программу 4.9 операцию, которая объединяет операции *union* и *find* и соответствующим образом изменив программы 4.11 и 4.10.
- **4.31** Модифицируйте интерфейс (программа 4.9) и реализацию (программа 4.11) для отношений эквивалентности так, чтобы в них присутствовала функция, возвращающая количество узлов, связанных с данным.
- 4.32** Модифицируйте программу 4.11 так, чтобы для представления структуры данных в ней вместо параллельных массивов использовался массив структур.
- **4.33** Используя стек целых чисел (без шаблонов), напишите программу из трех файлов, вычисляющую значение постфиксного выражения. Обеспечьте, чтобы клиентскую программу (аналог программы 4.5) можно было компилировать отдельно от реализации стека (аналог программы 4.7).
- **4.34** Модифицируйте решение предыдущего упражнения — отделите представление данных от реализаций функций-членов (сделайте программу из четырех файлов). Протестируйте полученный результат, подставляя реализацию стека на базе связного списка (аналог программы 4.8) без перекомпиляции программы-клиента.
- **4.35** Создайте полную реализацию АДТ "Отношения эквивалентности" на базе абстрактного класса с виртуальными функциями и сравните производительность полученной программы и программы 4.11 на крупных задачах связности (в стиле таблицы 1.1).

## 4.6 Очереди FIFO и обобщенные очереди

Очередь с дисциплиной *FIFO* (*First-In, First-Out* — *Первым пришел, первым ушел*) является еще одним фундаментальным АТД, который подобен стеку магазинного типа, но подчиняется противоположному правилу удаления элемента в операции *удалить*. Из очереди удаляется не последний вставленный элемент, а наоборот — элемент, который был вставлен в очередь раньше всех остальных.

Пожалуй, ящик для входящей корреспонденции нашего занятого профессора *должен* бы был функционировать как очередь FIFO, поскольку дисциплина "первым пришел, первым ушел" интуитивно кажется справедливой для обработки входящей корреспонденции. Однако профессор не всегда вовремя отвечал на телефонные звонки и даже позволял себе опаздывать на лекции! В стеке какая-нибудь служебная записка может застрять на дне, но срочные случаи обрабатываются сразу же после появления. В очереди FIFO документы обрабатываются по порядку, и каждый должен ожидать своей очереди.

### Программа 4.13 Интерфейс АТД "Очередь FIFO"

Этот интерфейс идентичен интерфейсу стека магазинного типа из программы 4.4, за исключением имен функций. Эти два АТД отличаются только спецификациями, что совершенно не отражается на коде интерфейса.

```
template <class Item>
class QUEUE
{
    private:
        // программный код, зависящий от реализации
    public:
        QUEUE(int);
        int empty();
        void put(Item);
        Item get();
};
```

Очереди FIFO с завидной частотой встречаются в повседневной жизни. Когда мы стоим в цепочке людей, чтобы посмотреть кинокартину или купить продукты, нас обслуживают в порядке FIFO. Аналогично этому в вычислительных системах очереди FIFO часто используются для обслуживания задач, которые необходимо выполнять по принципу: первым пришел, первым обслужился. Другим примером, иллюстрирующим различие между стеками и очередями FIFO, может служить отношение к скоропортящимся продуктам в бакалейной лавке. Если бакалейщик выкладывает новые товары на переднюю часть полки и покупатели берут товары также с передней части полки, получается стек. Бакалейщик может столкнуться с проблемой, поскольку товар на задней части полки может стоять очень долго и попросту испортиться. Выкладывая новые товары на заднюю часть полки, бакалейщик гарантирует, что время, в течение которого товар находится на полке, ограничивается временем, необходимым покупателям для приобретения всех товаров, выставленных на полку. Этот же базовый принцип применяется во множестве подобных ситуаций.

**Определение 4.3** *Очередь FIFO* — это АД, который содержит две базовых операции: *вставить* (**put** — занести) *новый элемент и удалить* (**get** — извлечь) *элемент, который был вставлен раньше всех остальных.*

Программа 4.13 является интерфейсом для АД "Очередь FIFO". Этот интерфейс отличается от интерфейса стека, рассмотренного в разделе 4.2, только спецификациями — для компилятора два интерфейса совершенно идентичны! Это подчеркивает тот факт, что сама абстракция, которую программисты обычно не определяют формально, является существенным компонентом абстрактного типа данных. Для больших приложений, которые могут содержать десятки АД, проблема их точного определения является критически важной. В настоящей книге мы работаем с АД, представляющими важнейшие понятия, которые определяются в тексте, но не при помощи формальных языков (разве что через конкретные реализации). Чтобы понять природу абстрактных типов данных, потребуется рассмотреть примеры их использования и конкретные реализации.

На рис. 4.6 показано, как очередь FIFO изменяется в ходе ряда последовательных операций *извлечь* и *занести*. Каждая операция *извлечь* уменьшает *размер* очереди на 1, а каждая операция *занести* увеличивает *размер* очереди на 1. Элементы очереди перечислены на рисунке в порядке их занесения в очередь, поэтому ясно, что первый элемент списка — это тот элемент, который будет возвращаться операцией *извлечь*. Опять-таки, в реализации можно организовать элементы любым требуемым способом при условии сохранения иллюзии того, что элементы организованы именно в соответствии с дисциплиной FIFO.

В случае реализации АД "Очередь FIFO" с помощью связного списка, элементы списка хранятся в следующем порядке: от первого вставленного до последнего вставленного элемента (см. рис. 4.6). Такой порядок является обратным по отношению к порядку, который применяется в реализации стека, причем он позволяет создавать эффективные реализации операций над очередями. Как показано на рис. 4.7 и в программе 4.14 (реализации), поддерживаются два указателя на этот список: один на начало списка (чтобы можно было *извлечь* первый элемент) и второй на его конец (для занесения в очередь нового элемента).

F		F
I		F I
R		F I R
S		F I R S
*	F	I R S
T		I R S T
*	I	R S T
I		R S T I
N		R S T I N
*	R	S T I N
*	S	T I N
*	T	I N
F		I N F
I		I N F I
*	I	N F I
R		N F I R
S		N F I R S
*	N	F I R S
*	F	I R S
*	I	R S
T		R S T
*	R	S T
O		S T O
U		S T O U
T		S T O U T
*	S	T O U T
*	T	O U T
*	O	U T
*	U	T
*	T	

#### РИСУНОК 4.6 ПРИМЕР ОЧЕРЕДИ FIFO

На рисунке показан результат выполнения последовательности операций. Операции представлены в левом столбце (порядок выполнения — сверху вниз); здесь буква обозначает операцию *put* (занести), а звездочка — операцию *get* (извлечь). Каждая строка содержит операцию, букву, возвращаемую операцией *get* и содержимое очереди от первого занесенного элемента до последнего в направлении слева направо.



### Программа 4.14 Реализация очереди FIFO на базе связного списка

Различие между очередью FIFO и стеком магазинного типа (программа 4.8) состоит в том, что новые элементы вставляются в конец списка, а не в его начало. Следовательно, в этом классе хранится указатель **tail** на последний узел списка, и, таким образом, функция **put** может добавлять в список новый узел, связывая его с узлом, на который ссылается указатель **tail**, а затем обновляя указатель **tail** так, чтобы он указывал уже на новый узел. Функции **QUEUE**, **get** и **empty** идентичны аналогичным функциям в реализации стека магазинного типа на базе связного списка из программы 4.8. Поскольку новые узлы всегда вставляются в конец списка, конструктор узлов может устанавливать в **next** поле указателя каждого нового узла и получать только один аргумент.

```
template <class Item>
class QUEUE
{
private:
    struct node
    { Item item; node* next;
      node(Item x)
      { item = x; next = 0; }
    };
    typedef node *link;
    link head, tail;
public:
    QUEUE(int)
    { head = 0; }
    int empty() const
    { return head == 0; }
    void put(Item x)
    { link t = tail;
      tail = new node(x);
      if (head == 0)
          head = tail;
      else t->next = tail;
    }
    Item get()
    { Item v = head->item;
      link t = head->next;
      delete head; head = t; return v;
    }
};
```

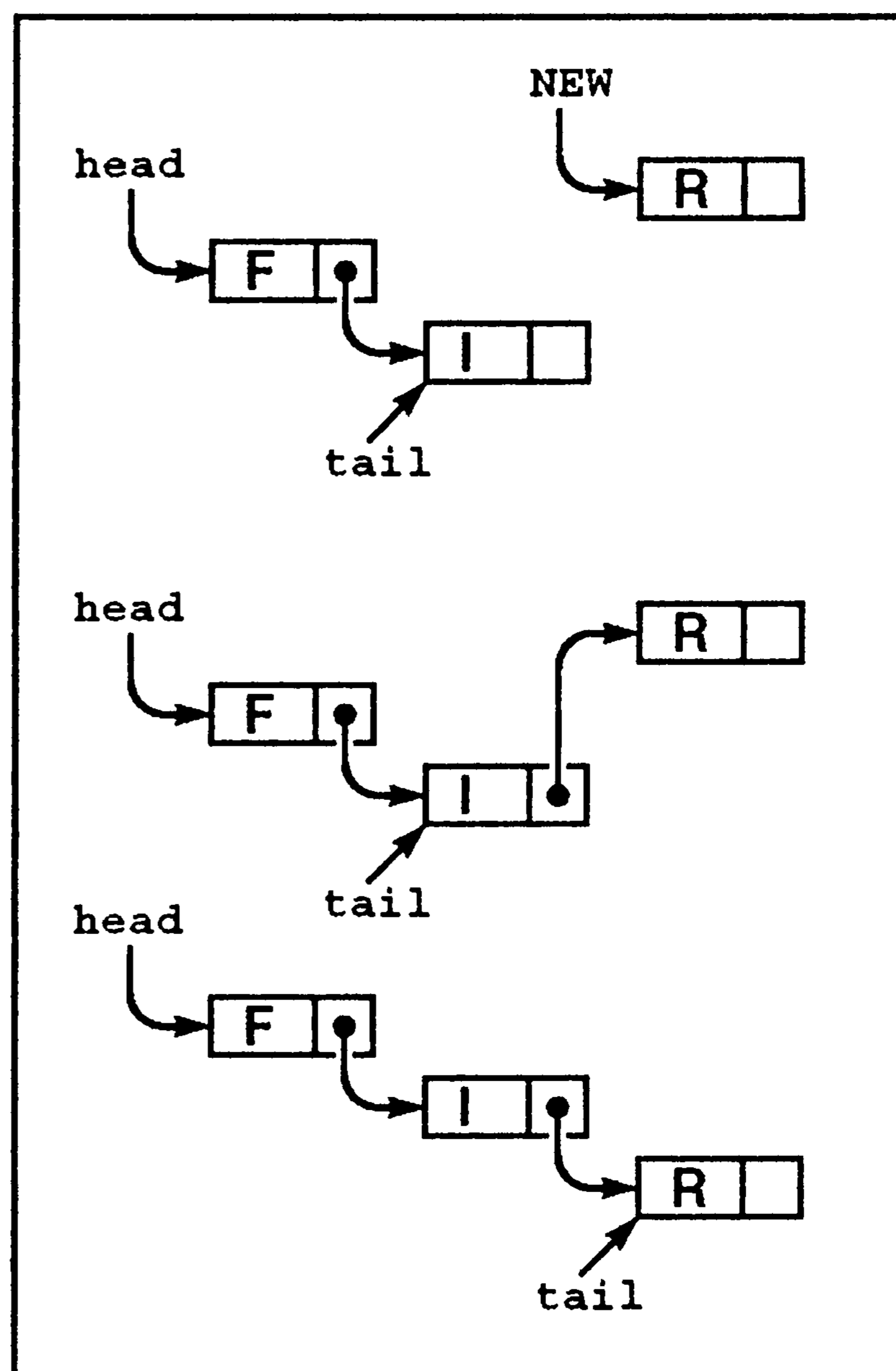


РИСУНОК 4.7 ОЧЕРЕДЬ НА БАЗЕ СВЯЗНОГО СПИСКА

В представлении очереди в виде связного списка новые элементы вставляются в конец списка, поэтому элементы связного списка от первого вставленного элемента до последнего располагаются от начала к концу очереди. Очередь представляется двумя указателями: **head** (начало) и **tail** (конец), которые указывают, соответственно, на первый и последний элемент. Для извлечения элемента из очереди удаляется элемент в начале очереди так же, как это делалось в случае стека (см. рис. 4.5). Чтобы занести в очередь новый элемент, поле связи узла, на который ссылается указатель **tail**, устанавливается так, чтобы оно указывало на новый элемент (середина рисунка), а затем обновляется указатель **tail** (нижняя часть рисунка)

Для реализации очереди FIFO можно также воспользоваться массивом, однако при этом необходимо соблюдать осторожность и обеспечить, чтобы время выполнения как операции *занести*, так и операции *извлечь* было постоянным. Это условие означает невозможность пересылки элементов очереди внутри массива, как это можно было

бы предположить при буквальной интерпретации рис. 4.6. Следовательно, как и в реализации на базе связного списка, потребуется поддерживать два индекса в массиве: индекс начала очереди и индекс конца очереди. Содержимым очереди считаются элементы, индексы которых находятся в рамках упомянутых двух индексов. Чтобы *извлечь* элемент, он удаляется его из начала (**head**) очереди, после чего индекс **head** увеличивается на единицу; чтобы *занести* элемент, он добавляется в конец (**tail**) очереди, а индекс **tail** увеличивается на единицу. Как иллюстрирует рис. 4.8, последовательность операций *занести* и *извлечь* приводит к тому, что все выглядит так, будто очередь движется по массиву. Она устроена так, что при достижении конца массива осуществляется переход на его начало. С деталями реализации рассмотренного процесса можно ознакомиться в коде программы 4.15.

#### Программа 4.15 Реализация очереди FIFO на базе массива

К содержимому очереди относятся все элементы массива, расположенные между индексами **head** и **tail**; при этом учитывается переход с конца на начало массива. Если индексы **head** и **tail** равны, очередь считается пустой, однако если они стали равными в результате выполнения операции **put**, очередь считается полной. Как обычно, проверка на такие ошибочные ситуации не выполняется, но размер массива делается на 1 больше максимального количества элементов очереди, ожидаемое программой-клиентом. При необходимости программу можно расширить, включив в нее подобного рода проверки.

```
template <class Item>
class QUEUE
{
private:
    Item *q; int N, head, tail;
public:
    QUEUE(int maxN)
    { q = new Item[maxN+1];
      N = maxN+1; head = N; tail = 0; }
    int empty() const
    { return head % N == tail; }
    void put(Item item)
    { q[tail++] = item; tail = tail % N; }
    Item get()
    { head = head % N; return q[head++]; }
};
```

```
F      F
I      F I
R      F I R
S      F I R S
·      F I R S
T      I R S T
·      I R S T
I      R S T I
N      R S T I N
·      R S T I N
·      S T I N
·      T I N
F      I N F
I      I N F I
·      I N F I
R      N F I R
S      N F I R S
·      F I R S
·      I R S
T      T R S
·      R T S
O      T O S
U      T O U S
T      T O U T S
·      S T O U T
·      T O U T
·      O U T
·      U T
·      T
```

#### РИСУНОК 4.8 ПРИМЕР ОЧЕРЕДИ FIFO, РЕАЛИЗОВАННОЙ НА БАЗЕ МАССИВА

*Данная последовательность операций отображает манипуляции с данными, лежащие в основе абстрактного представления очереди из рис. 4.6. Эти манипуляции соответствуют случаю, когда очередь реализуется за счет запоминания ее элементов в массиве, сохранения индексов начала и конца очереди и обеспечения перехода индексов на начало массива, когда они достигают его конца. В данном примере индекс **tail** переходит на начало массива, когда вставляется второй символ **T**, а индекс **head** — когда удаляется второй символ **S**.*

**Лемма 4.2** Для АТД "Очередь FIFO" имеется возможность реализовать операции `get` и `put` с постоянным временем выполнения, используя либо массивы, либо связные списки.

Этот факт становится ясным, стоит только внимательно посмотреть на код программ 4.14 и 4.15.

Те же соображения относительно использования оперативной памяти, которые были изложены в разделе 4.4, применимы и к очередям FIFO. Представление на базе массива требует резервирования оперативной памяти с объемом, достаточным для запоминания максимально ожидаемого количества элементов очереди. В случае же представления на базе связного списка оперативная память используется пропорционально числу элементов в структуре данных; это происходит за счет дополнительного расхода памяти на связи (между элементами) и дополнительного расхода времени на распределение и освобождение памяти для каждой операции.

Хотя по причине фундаментальной взаимосвязи между стеками и рекурсивными программами (см. главу 5), со стеками приходится сталкиваться чаще, чем с очередями FIFO, будут также встречаться и алгоритмы, для которых очереди являются естественными базовыми структурами данных. Как уже отмечалось, очереди и стеки используются в вычислительных приложениях чаще всего для того, чтобы отложить выполнение того или иного процесса. Хотя многие приложения, использующие очередь отложенных задач, работают корректно вне зависимости от того, какие правила удаления элементов задействуются в операциях *удалить*, общее время выполнения программы или использования ресурсов, может зависеть от применяемой дисциплины. Когда в подобных приложениях встречается большое количество операций *вставить* или *удалить*, выполняемых над структурами данных с большим числом элементов, различия в производительности обретают первостепенную важность. Поэтому в настоящей книге столь существенное внимание уделяется таким АТД. Если бы производительность программ не интересовала, можно было бы создать один единственный АТД с операциями *вставить* и *удалить*; однако производительность является предельно важным показателем, поэтому каждое правило, в сущности, соответствует своему АТД. Чтобы оценить эффективность отдельного АТД, потребуется учитывать издержки (т. е. расход машинного времени) двух видов: издержки, обусловленные реализацией, которые зависят от алгоритмов и структур данных, выбранных для реализации, и издержки, обусловленные отдельным правилом выполнения операции, в смысле его влияния на производительность клиентской программы. В заключение данного раздела описываются несколько таких АТД, которые будут рассматриваться подробно в следующих главах.

И стеки магазинного типа, и очереди FIFO являются частными случаями более общего АТД: *обобщенной (generalized) очереди*. Частные случаи обобщенных очередей различаются только правилами удаления элементов. Для стеков это правило будет таким: "удалить элемент, который был вставлен последним"; для очередей FIFO правило гласит: "удалить элемент, который был вставлен первым"; существует и множество других вариантов.

Простым, тем не менее, мощным вариантом является *неупорядоченная очередь (random queue)*, подчиняющаяся следующему правилу: "удалить случайный элемент"; и программа-клиент может ожидать, что она с одинаковой вероятностью получит лю-

бой из элементов очереди. Используя представление на базе массива (см. упр. 4.48), для неупорядоченной очереди можно реализовать операции с постоянным временем выполнения. Представление на базе массива требует (так же, как для стеков и очередей FIFO), чтобы оперативная память была распределена заранее. Однако в данном случае альтернативное представление на базе связного списка менее привлекательно, чем в случае стеков и очередей FIFO, поскольку эффективная реализация как операции вставки, так и операции удаления, является очень трудной задачей (см. упр. 4.49). Чтобы с высокой степенью вероятности избежать сценариев с наихудшей производительностью, неупорядоченные очереди можно использовать в качестве базиса для рандомизированных алгоритмов (см. раздел 2.7).

При описании стеков и очередей FIFO элементы идентифицировались по времени вставки в очередь. В качестве альтернативы эти абстрактные понятия можно описывать в терминах последовательного перечня упорядоченных элементов и базовых операций вставки и удаления элементов в начале и конце списка. Если элементы вставляются в конец списка и удаляются также с конца, получается стек (точно как в реализации на базе массива); если элементы вставляются в начало и удаляются в начале, также получается стек (точно как в реализации на базе связного списка); если же элементы вставляются в конец, а удаляются с начала, то получается очередь FIFO (как в реализации на базе связного списка). В конечном итоге, если элементы вставляются в начало, а удаляются с конца, также получается очередь FIFO (этот вариант не соответствует ни одной из реализаций — для его точной реализации можно было бы изменить представление на базе массива, а вот представление на базе связного списка для этой цели не подойдет из-за необходимости поддерживать указатель на конец очереди в случае удаления элементов в конце очереди). Развивая дальше эту точку зрения, приходим к абстрактному типу данных *дек* (*double-ended queue*, *двухсторонняя очередь*), в котором и вставки, и удаления разрешаются с обеих сторон. Его реализацию мы оставляем в качестве упражнений (см. упр. 4.43—4.47); при этом необходимо отметить, что реализация на базе массива является простым расширением программы 4.15, а для реализации на базе связного списка потребуется двухсвязный список, иначе удалять элементы дека можно будет только с одной стороны.

В главе 9 рассматриваются *очереди с приоритетами*, в которых элементы имеют ключи, а правило удаления элементов выглядит как "удалять элемент с самым маленьким ключом". АТД "Очередь с приоритетами" полезен во множестве приложений, и задача нахождения эффективных реализаций для этого АТД в течение многих лет была целью исследований в компьютерных науках. Важным фактором в исследованиях были идентификация и использование АТД в приложениях: подставляя новый алгоритм вместо старой реализации в крупном, сложном приложении и сравнивая результаты, можно сразу же определить, является ли новый алгоритм правильным. Более того, отмечая, как от подстановки новой реализации изменяется общее время выполнения приложения, можно сразу же определить, является ли новый алгоритм более эффективным, нежели старый. Структуры данных и алгоритмы, которые рассматриваются в главе 9 в плане решения данной проблемы, столь же интересны, сколь оригинальны и эффективны.

В главах с 12 по 16 исследуются *символьные таблицы* (*symbol tables*). Это обобщенные очереди, в которых элементы имеют ключи, а правило удаления элементов звучит так: "удалить элемент, ключ которого равен данному, если таковой элемент существует". Этот АТД, пожалуй, самый важный из изучаемых, и можно будет ознакомиться с десятками его реализаций.

Каждый из этих АТД дает начало ряду родственных, но разных АТД, которые появились в результате внимательного изучения клиентских программ и производительности различных реализаций. В разделах 4.7 и 4.8 обсуждаются многочисленные примеры изменений в спецификации обобщенных очередей, ведущих к еще более оригинальным АТД, которые будут подробно рассматриваться далее в книге.

## Упражнения

▷ **4.36** Найдите содержимое элементов  $q[0]$ , ... ,  $q[4]$  после выполнения программой 4.15 операций, показанных на рис. 4.6. Считайте, что  $\text{maxN}$ , как и на рис. 4.8, равно 10.

▷ **4.37** В последовательности

`EAS * Y * QUE * * * ST * * * IO * N * * *`

буква означает операцию *put*, а звездочка — операцию *get*. Найдите последовательность значений, возвращаемых операциями *get*, когда эта последовательность операций выполняется над первоначально пустой очередью FIFO.

**4.38** Модифицируйте приведенную в тексте реализацию очереди FIFO на базе массива (программа 4.15) так, чтобы в ней вызывалась функция `error()`, если клиент пытается выполнить операцию *get*, когда очередь пуста, или операцию *put*, когда очередь переполнена.

**4.39** Модифицируйте приведенную в тексте реализацию очереди FIFO на базе связного списка (программа 4.14) так, чтобы в ней вызывалась функция `error()`, если клиент пытается выполнить операцию *get*, когда очередь пуста, или если при выполнении *put* отсутствует доступная память в `new`.

▷ **4.40** В последовательности

`EAs + Y + QUE * * + st + * + IO * n + + *`

буква верхнего регистра означает операцию *put* в начале дека, буква нижнего регистра — операцию *put* в конце дека, знак плюс означает операцию *get* в начале, а звездочка — операцию *get* в конце. Найдите последовательность значений, возвращаемых операциями *get*, когда эта последовательность операций выполняется над первоначально пустым деком.

▷ **4.41** Используя правила, принятые в упр. 4.40, найдите, каким образом в последовательность `EasY` необходимо вставить знаки плюс и звездочки, чтобы операции *get* возвращали следующую последовательность символов: (i) `EsaY`, (ii) `YasE`, (iii) `aYsE`, (iv) `asYE`; либо же в каждом случае докажите, что такая последовательность невозможна.

● **4.42** Имея две последовательности, составьте алгоритм, позволяющий определить, возможно ли в первую последовательность вставить знаки плюс и звездочки так,

чтобы, интерпретируя ее как последовательность операций над деком в смысле упр. 4.41, получить вторую последовательность.

▷ **4.43** Запишите интерфейс для АТД "Дек".

**4.44** Для интерфейса дека (упр. 4.43) запишите реализацию, в которой в качестве базовой структуры данных используется массив.

**4.45** Для интерфейса дека (упр. 4.43) запишите реализацию, в которой в качестве базовой структуры данных используется двухсвязный список.

**4.46** Для приведенного в тексте интерфейса очереди FIFO (программа 4.13) запишите реализацию, в которой в качестве базовой структуры данных используется циклический список.

**4.47** Напишите программу-клиент, которая проверяет полученный АТД "Дек" (упр. 4.43), считывая с командной строки в качестве первого аргумента строку команд подобную приведенной в упр. 4.40, после чего выполняет указанные операции. В интерфейс и реализации добавьте функцию-член **dump** и распечатывайте содержимое дека после каждой операции, как это сделано на рис. 4.6.

○ **4.48** Создайте АТД "Неупорядоченная очередь" (напишите интерфейс и реализацию), в котором в качестве базовой структуры данных используется массив. Обеспечьте для каждой операции постоянное время выполнения.

●● **4.49** Создайте АТД "Неупорядоченная очередь" (напишите интерфейс и реализацию), в котором в качестве базовой структуры данных используется связный список. Напишите как можно более эффективные реализации операций *insert* и *remove* и проанализируйте связанные с ними издержки для наихудшего случая их выполнения.

▷ **4.50** Напишите программу-клиент, которая выбирает для лотереи числа следующим образом: заносит в неупорядоченную очередь числа от 1 до 99, а затем удаляет пять из них и выводит результат.

**4.51** Напишите программу-клиент, которая считывает с командной строки в качестве первого аргумента целое число  $N$ , а затем распечатывает результат раздачи в покере карт на  $N$  игроков. Для этого она должна заносить в неупорядоченную очередь  $N$  элементов (см. упр. 4.7) и затем выдавать результат выбора из этой очереди пяти карт за один раз.

● **4.52** Напишите программу, которая решает задачу связности. Для этого она должна вставлять все пары в неупорядоченную очередь, а затем с помощью алгоритма взвешенного быстрого поиска (программа 1.3) извлекает их из очереди.

## 4.7 Повторяющиеся и индексные элементы

Во многих приложениях обрабатываемые абстрактные элементы являются *уникальными*. Это качество подводит нас к мысли пересмотреть представления о том, как должны функционировать стеки, очереди FIFO и другие обобщенные АТД. В частности, в данном разделе рассматриваются такие изменения спецификаций стеков, очередей FIFO и обобщенных очередей, которые запрещают в этих структурах данных присутствие повторяющихся элементов.

Например, компании, поддерживающей список рассылки по адресам покупателей, может потребоваться расширить этот список информацией из других списков, собранных с различных источников. Это выполняется с помощью соответствующих операций *insert*. При этом, по-видимому, не требуется заносить информацию о покупателях, адреса которых уже присутствуют в списке. Далее можно будет увидеть, что тот же принцип применим в самых разнообразных приложениях. В качестве еще одного примера рассмотрим задачу маршрутизации сообщений в сложной сети передачи данных. Можно попытаться передать сообщение одновременно по нескольким маршрутам, однако каждому отдельно взятому узлу сети необходима только одна копия этого сообщения в свои внутренние структуры данных.

Один из подходов к решению данной проблемы — это возложение на программы-клиенты задачи по обеспечению того, что в АД не будут присутствовать повторяющиеся элементы. Предположительно, программы-клиенты могли бы выполнять эту задачу, используя какой-нибудь другой АД. Но поскольку цель создания любого АД связана с обеспечением четких решений прикладных задач с помощью клиентских программ, можно прийти к мнению, что обнаружение повторяющихся элементов и разрешение таких ситуаций — это часть задачи, относящаяся к компетенции АД.

Использование стратегии запрета присутствия повторяющихся элементов сводится к изменению *абстракции*: интерфейс, имена операций и прочее для такого АД будут такими же, как и для соответствующего АД, в котором данный принцип не используется, но функционирование реализации изменяется фундаментальным образом. Вообще говоря, при модификации описания какого-нибудь АД получается совершенно новый АД — АД, обладающий совершенно другими свойствами. Эта ситуация демонстрирует также ненадежную природу спецификации АД: обеспечить, чтобы клиентские программы и реализации придерживались спецификации интерфейса, достаточно трудно, однако обеспечить применение такого высокоуровневого принципа, как данный — совсем другое дело. Тем не менее, мы заинтересованы в подобных алгоритмах, поскольку программы-клиенты применяют такие свойства для решения задач новыми способами, а реализации могут использовать преимущества, предоставляемые подобными ограничениями, для обеспечения более эффективного решения задач.

На рис. 4.9 проиллюстрирована работа модифицированного АД "Стек без повторяющихся элементов" для

L		L
A		L A
·	A	L
S		L S
T		L S T
I		L S T I
·	I	L S T
N		L S T N
·	N	L S T
F		L S T F
I		L S T F I
R		L S T F I R
·	R	L S T F I
S		L S T F I
T		L S T F I
·	I	L S T F
·	F	L S T
·	T	L S
O		L S O
U		L S O U
·	U	L S O
T		L S O T
·	T	L S O
·	O	L S
·	S	L

#### РИСУНОК 4.9 СТЕК МАГАЗИННОГО ТИПА БЕЗ ПОВТОРЯЮЩИХСЯ ЭЛЕМЕНТОВ

Данная последовательность операций аналогична операциям на рис. 4.1, однако она выполняется над стеком, в котором запрещены повторяющиеся объекты.

Серыми квадратами отмечены случаи, когда стек остается неизменным, так как в стеке уже присутствует элемент, идентичный тому, который должен быть занесен.

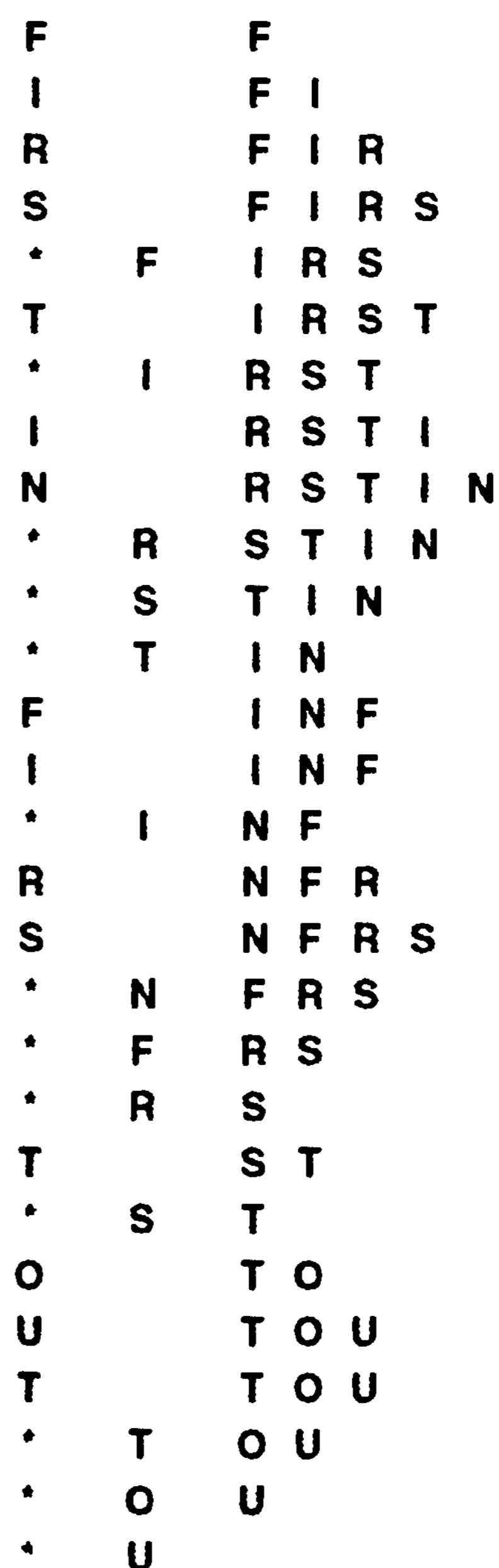
Количество элементов в стеке ограничено числом возможных различных (отличающихся) элементов.

случая, показанного на рис. 4.1; на рис. 4.10 приведен результат аналогичных изменений для очереди FIFO.

Вообще говоря, решение относительно повторяющихся элементов приходится принимать тогда, когда программа-клиент делает запрос на *занесение* элемента, уже имеющегося в структуре данных. Как следует поступить в такой ситуации? Продолжать работу так, как будто запроса вообще не было? Или *удалить* старый элемент и *занести* новый? Это решение влияет на порядок, в котором, в конечном счете, будут обрабатываться элементы в АТД наподобие стеков и очередей FIFO (см. рис. 4.11); упомянутое различие очень существенно для клиентских программ. Например, компания, использующая подобный АТД для списка рассылки, могла бы предпочесть занесение нового элемента на место старого (вероятно, предполагая, что он содержит более свежую информацию о клиенте); а коммутационный центр, использующий такой АТД, мог бы предпочесть проигнорировать новый элемент (вероятно, он уже предпринял соответствующие шаги и отправил сообщение). Более того, выбор того или иного принципа влияет на реализации: как правило, принцип "удалить старый элемент" более труден в реализации, нежели принцип "игнорировать новый элемент", поскольку связан с модификацией структуры данных.

Для реализации обобщенных очередей без повторяющихся элементов необходимо иметь абстрактную операцию, проверяющую равенство элементов (как это рассматривалось в разделе 4.1). Помимо такой операции необходимо иметь возможность определять, существует ли уже в структуре данных новый вставляемый элемент. Этот общий случай предполагает необходимость реализации АТД "Таблица символов", поэтому он рассматривается в контексте реализаций, в главах с 12 по 15.

Имеется важный частный случай, для которого существует простое решение; его иллюстрирует программа 4.16 для АТД "Стек магазинного типа". В этой реализации предполагается, что элементы являются целыми числами в диапазоне от 0 до  $M - 1$ . Далее, чтобы определять, имеется ли уже в стеке некоторый элемент, в реализации используется второй массив, индексами которого являются сами элементы стека. При вставке в стек элемента  $i$  выполняется также установка в 1  $i$ -ого элемента второго массива, а при удалении из стека элемента  $i$   $i$ -ый элемент второго массива устанавливается в 0. Во всем остальном для вставки и уда-



**РИСУНОК 4.10 ОЧЕРЕДЬ FIFO БЕЗ ПОВТОРЯЮЩИХСЯ ЭЛЕМЕНТОВ, ФУНКЦИОНИРУЮЩАЯ ПО ПРИНЦИПУ "ИГНОРИРОВАТЬ НОВЫЙ ЭЛЕМЕНТ"**

*Данная последовательность операций аналогична операциям, показанным на рис. 4.6, однако она выполняется над очередью, в которой запрещены повторяющиеся объекты.*

*Серыми квадратами отмечены случаи, когда очередь остается неизменной, так как в очереди уже присутствует элемент, идентичный тому, который должен быть занесен.*



ления элементов применяется тот же код с одной дополнительной проверкой: перед вставкой элемента выполняется проверка, нет ли в стеке такого элемента. Если есть, операция *занесения* элемента игнорируется. Это решение не зависит от используемого представления стека — на базе массива, связанного списка или чего-нибудь еще. Для реализации принципа "удалять старый элемент" требуется большой объем работы (см. упр. 4.57).

Суммируя изложенное выше, можно сказать, что один из способов реализации стека без повторяющихся элементов, функционирующего по принципу "игнорировать новый элемент", состоит в поддержке *двух* структур данных. Первая, как и прежде, содержит элементы стека и позволяет отслеживать порядок, в котором были вставлены элементы стека. Вторая структура является массивом, позволяющим отслеживать, какие элементы находятся в стеке; индексами этого массива являются элементы стека. Такое использование массива рассматривается как частный случай реализации таблицы символов, которая обсуждается в разделе 12.2. Когда известно, что элементы представляют собой целые числа в диапазоне от 0 до  $M - 1$ , эту же методику можно применять по отношению к любой обобщенной очереди.

#### Программа 4.16 Стек индексных элементов, в котором запрещены повторяющиеся элементы

В этой реализации стека магазинного типа предполагается, что класс **Item** имеет тип **int**, который возвращает целые числа в диапазоне от 0 до **maxN-1**, так что он может поддерживать массив **t**, где каждому элементу стека соответствует отличное от нуля значение. Этот массив дает возможность функции **push** быстро проверять, не находится ли уже в стеке ее аргумент, и если так, то не предпринимать никаких действий. В каждом элементе массива **t** используется только один бит, поэтому при желании можно сэкономить оперативную память, используя вместо целых чисел символы или биты (см. упр. 12.12).

```
template <class Item>
class STACK
{
private:
    Item *s, *t; int N;
public:
    STACK(int maxN)
    {
        s = new Item[maxN]; N = 0;
        t = new Item[maxN];
        for (int i = 0; i < maxN; i++) t[i] = 0;
    }
    int empty() const
    { return N == 0; }
```

```
F      F
I      F I
R      F I R
S      F I R S
*      F I R S
T      I R S T
*      I R S T
I      R S T I
N      R S T I N
*      R S T I N
*      S T I N
*      T I N
F      I N F
I      N F I
*      N F I
R      F I R
S      F I R S
*      F I R S
*      I R S
*      R S
T      S T
*      S T
O      T O
U      T O U
T      O U T
*      O U T
*      U U
*      T
```

#### РИСУНОК 4.11 ОЧЕРЕДЬ FIFO БЕЗ ПОВТОРЯЮЩИХСЯ ЭЛЕМЕНТОВ, ФУНКЦИОНИРУЮЩАЯ ПО ПРИНЦИПУ "УДАЛЯТЬ СТАРЫЙ ЭЛЕМЕНТ"

*Данная последовательность операций аналогична операциям, показанным на рис. 4.10. Однако здесь используется другой, более сложный для реализации принцип, в соответствии с которым новый элемент всегда добавляется в конец очереди. Если же в очереди присутствует точно такой же элемент, он удаляется.*

```
void push(Item item)
{
    if (t[item] == 1) return;
    s[N++] = item; t[item] = 1;
}
Item pop()
{ t[s[--N]] = 0; return s[N]; }
};
```

---

Этот частный случай встречается довольно часто. Его наиболее важный пример — когда элементы структуры данных сами являются индексами массива, поэтому такие элементы называются *индексными элементами* (*index items*). Обычно имеется набор из  $M$  объектов, хранимых в каком-то другом массиве, и как часть более сложного алгоритма, необходимо передавать этот набор объектов через структуру обобщенной очереди. Объекты заносятся в очередь по индексам и обрабатываются при удалении, причем каждый объект должен обрабатываться только один раз. Очередь без повторяющихся элементов, в которой используются индексы массива, позволяет напрямую достичь этой цели.

Каждый из этих вариантов (запрещать или нет повторяющиеся элементы; использовать или нет новый элемент ведет к новому АД. Различия могут показаться незначительными, тем не менее, они заметно влияют на динамическую характеристику АД (как это видится со стороны клиентских программ), а также на выбор алгоритма и структуры данных для реализации различных операций. Посему нет иной альтернативы, кроме как трактовать все эти АД как разные. Более того, приходится учитывать и другие варианты: например, может появиться желание изменить интерфейс с целью информирования клиентской программы о том, что она пытается вставить дубликат уже имеющегося элемента, либо же предоставить программе-клиенту возможность выбора: игнорировать новый элемент или удалять старый.

Когда мы неформально используем такой термин, как *магазинный стек*, *очередь FIFO*, *дек*, *очередь по приоритету* или *таблица символов*, мы потенциально говорим о *семействе* абстрактных типов данных, где каждый тип имеет свой набор операций, набор соглашений о значении этих операций, причем для эффективной поддержки этих операций каждый тип требует своих, в некоторых случаях довольно сложных, реализаций.

## Упражнения

▷ **4.53** Для стека без повторяющихся элементов, работающего по принципу "удалять старый элемент", изобразите рисунок, аналогичный рис. 4.9.

**4.54** Модифицируйте стандартную реализацию стека на базе массива из раздела 4.4 (программа 4.7) таким образом, чтобы в ней были запрещены повторяющиеся элементы по принципу "игнорировать новый элемент". Используйте метод "грубой силы", заключающийся в сканировании всего стека.

**4.55** Модифицируйте стандартную реализацию стека на базе массива из раздела 4.4 (программа 4.7) таким образом, чтобы в ней были запрещены повторяющиеся элементы по принципу "удалять старый элемент". Используйте метод "грубой силы", заключающийся в сканировании всего стека и перемещении его элементов.

- **4.56** Выполните упражнения 4.54 и 4.55 для реализации стека на базе связного списка из раздела 4.4 (программа 4.8).
- **4.57** Разработайте реализацию стека магазинного типа, в которой повторяющиеся элементы запрещены по принципу "удалять старый элемент". Элементами стека являются целые числа в диапазоне от 0 до  $M - 1$ , а операции *push* и *pop* должны иметь постоянное время выполнения. *Подсказка:* возьмите представление стека на базе двухсвязного списка и воспользуйтесь указателями на узлы этого списка, а не массивом индексных значений.
- 4.58** Выполните упражнения 4.54 и 4.55 для очереди FIFO.
- 4.59** Выполните упражнение 4.56 для очереди FIFO.
- 4.60** Выполните упражнение 4.57 для очереди FIFO.
- 4.61** Выполните упражнения 4.54 и 4.55 для рандомизированной очереди.
- 4.62** Напишите клиентскую программу для АТД, полученного в упражнении 4.61, которая использует рандомизированную очередь без повторяющихся элементов.

## 4.8 АТД первого класса

Интерфейсы и реализации АТД "Стек" и "Очередь FIFO" в разделах с 4.2 по 4.7 достигают важной цели: сокрытие от клиентских программ структур данных, используемых в реализациях. Эти АТД весьма полезны и будут служить в качестве основы для множества других реализаций, рассматриваемых в книге.

Однако, когда такие типы данных используются в программах таким же образом, как и встроенные типы данных, например, `int` или `float`, программиста могут подстергать ловушки. В данном разделе мы рассмотрим, как конструировать АТД, с которыми программы-клиенты могут работать так же, как и со встроенными типами данных, без нарушения принципа сокрытия деталей реализации от клиентских программ.

**Определение 4.4** *Тип данных первого класса* — это тип данных, который может использоваться в программах таким же образом, как и встроенные типы данных.

Например, типы данных первого класса можно использовать как переменные (т.е. объявлять переменные этих типов и присваивать им значения). Такие типы данных можно также использовать в аргументах и возвращаемых значениях функций. В этом определении, равно как и в других, относящихся к типам данных, нельзя достигнуть точности, не углубившись в высокие материи, касаемые семантики операций. Как будет показано, одно дело утверждать, что можно записывать  $a = b$ , где  $a$  и  $b$  являются объектами определяемого пользователем класса, и совсем другое дело — точно определить, что означает эта операция.

В идеале можно представлять, что все типы данных имеют некоторый универсальный набор хорошо определенных операций; в действительности же каждый тип данных характеризуется своим собственным набором операций. Это различие между типами данных само по себе препятствует точному определению типа данных первого класса, поскольку оно предполагает необходимость дать определения для *каждой* операции, заданной для встроенных типов данных, а это делается редко. Чаще всего важными являются лишь несколько критических операций и вполне достаточно применять только их для собственных типов данных так же, как и для встроенных типов.

Во многих языках программирования создание типов данных первого класса является делом трудным или даже невозможным; в языке C++ необходимые базовые инструменты — это концепция класса и возможность перегрузки операций. Действительно, в языке C++ легко определять классы, которые являются типами данных первого класса; более того, имеется четкий способ модернизации тех классов, которые таковыми не являются.

Метод, который применяется в языке C++ для реализации типов данных первого класса, применим к любому классу: в частности, он применим к обобщенным очередям и, таким образом, обеспечивает возможность создания программ, которые оперируют со стеками и очередями FIFO во многом так же, как и с другими типами данных C++. При изучении алгоритмов эта возможность достаточно важна, поскольку она предоставляет естественный способ выражения высокоуровневых операций, относящихся к таким АТД. Например, вполне уместно говорить об операциях *соединения* двух очередей — т.е. создании из них одной очереди. Далее будут рассматриваться алгоритмы, которые реализуют такие операции для АТД "Очередь по приоритету" (глава 9) и "Таблица символов" (глава 12).

Если доступ к типу данных первого класса осуществляется только через интерфейс, то это действительно АТД первого класса (см. определение 4.1). Обеспечение возможности работать с экземплярами АТД, в основном, так же, как со встроенными типами данных, например, `int` или `float`, — важная цель многих языков программирования высокого уровня, поскольку это позволяет написать любую программу так, чтобы она манипулировала центральными объектами приложения. В таком случае множество программистов смогут одновременно работать над большими системами, используя точно определенный набор абстрактных операций, что, в свою очередь, позволяет реализовывать абстрактные операции самыми разными способами без какого-либо изменения кода приложения (например, для новых компьютеров или сред выполнения).

#### Программа 4.17 Драйвер комплексных чисел (корни из единицы)

Эта клиентская программа выполняет вычисления над комплексными числами с использованием АТД, который позволяет проводить вычисления непосредственно с интересующей нас абстракцией. В этой связи объявляются переменные типа `Complex` и задействуются в арифметических выражениях с перегруженными операциями. Данная программа проверяет реализацию АТД, вычисляя корни из единицы в различных степенях. При помощи соответствующего определения перегруженной операции `<<` (см. упр. 4.70) производится вывод таблицы, показанной на рис. 4.12.

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "COMPLEX.cxx"
int main(int argc, char *argv[])
{ int N = atoi(argv[1]);
  cout << N << " complex roots of unity" << endl;
  for (int k = 0; k < N; k++)
  { float theta = 2.0*3.14159*k/N;
    Complex t(cos(theta), sin(theta)), x = t;
    cout << k << ": " << t << " ";
    for (int j = 0; j < N-1; j++) x *= t;
    cout << x << endl;
  }
}
```

Для начала в качестве примера рассмотрим АТД первого класса, соответствующий абстракции "комплексное число". Наша цель — получить возможность записывать программы, подобные программе 4.17, которая выполняет алгебраические действия над комплексными числами, используя операции, определенные в АТД. Данная программа объявляет и инициализирует комплексные числа, а также использует операции \*= и <<. Можно было бы воспользоваться и другими операциями, но в качестве примера достаточно рассмотреть только эти две. На практике используется класс `complex` из библиотеки C++, в котором имеется обширный набор перегруженных операций, включая даже тригонометрические функции.

В программе 4.17 принимаются во внимание несколько математических свойств комплексных чисел. Стоит немного отклониться от основной темы и кратко рассмотреть эти свойства. В некотором смысле это даже и не отступление, поскольку само по себе исследование связи между комплексными числами как математической абстракцией и их представлением в компьютерной программе достаточно интересно.

Число  $i = \sqrt{-1}$  называется *мнимым* числом. Хотя  $\sqrt{-1}$  как вещественное число не имеет смысла, мы называем его  $i$  и выполняем над  $i$  алгебраические операции, заменяя  $i^2$  на  $-1$  всякий раз, когда его встречаем. Комплексное число состоит из двух частей: вещественной и мнимой — комплексные числа можно записывать в виде  $a + bi$ , где  $a$  и  $b$  — вещественные числа. Для умножения комплексных чисел применяются обычные алгебраические правила, при этом всякий раз  $i^2$  заменяется на  $-1$ . Например:

$$(a + bi)(c + di) = ac + bci + adi + bdi^2 = (ac - bd) + (ad + bc)i.$$

При умножении комплексных чисел вещественные или мнимые части могут сокращаться (принимать значения 0), например:

$$(1 - i)(1 - i) = 1 - i - i + i^2 = -2i,$$

$$(1 + i)^4 = 4i^2 = -4,$$

$$(1 + i)^8 = 16.$$

Разделив обе части последнего уравнения на  $16 = (\sqrt{2})^8$ , мы находим, что

$$\left(\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}\right)^8 = 1.$$

Вообще говоря, имеется много комплексных чисел, которые при возведении в степень дают 1. Они называются *комплексными корнями из единицы*. Действительно, для каждого  $N$  имеется ровно  $N$  комплексных чисел  $z$ , для которых справедливо  $z^N = 1$ . Легко можно показать, что этим свойством обладают числа

$$\cos\left(\frac{2\pi k}{N}\right) + i \sin\left(\frac{2\pi k}{N}\right),$$

для  $k = 0, 1, \dots, N-1$  (см. упр. 4.68). Например, если в этой формуле взять  $k = 1$  и  $N = 8$ , получим корень восьмой степени из единицы, который мы только что нашли.

Программа 4.17 вычисляет все корни  $N$ -ой степени из единицы для любого данного  $N$  и затем возводит их в  $N$ -ую степень, используя операцию \*=, определенную в данном АТД. Выходные данные программы показаны на рис. 4.12. При этом ожи-

дается, что каждое из этих чисел, возведенное в  $N$ -ую степень, дает один и тот же результат — 1 или  $1 + 0i$ . Полученные мнимые части не равны строго нулю из-за ограниченной точности вычислений. Интерес к этой программе объясняется тем, что она использует класс **Complex** точно так же, как встроенный тип данных. Далее будет подробно показано, почему это возможно.

Даже в этом простом примере важно, чтобы тип данных был абстрактным, поскольку имеется еще одно стандартное представление, которым можно было бы воспользоваться — полярные координаты (см. упр. 4.67). Программа 4.18 — это интерфейс, который может использоваться такими клиентами, как программа 4.17; а программа 4.19 — это реализация, в которой используется стандартное представление данных (одно число типа **float** для вещественной части и другое — для мнимой).

### Программа 4.18 Интерфейс АД первого класса для комплексных чисел

Этот интерфейс для комплексных чисел позволяет реализациям создавать объекты типа **Complex** (инициализированные двумя значениями типа **float**), обеспечивает доступ к вещественной и мнимой частям и использование операции  $*$ . Хотя это и не задано явно, стандартные системные механизмы, действующие для всех классов, позволяют использовать объекты класса **Complex** в операторах присваивания, а также в аргументах и возвращаемых значениях функций.

```
class Complex
{
private:
    // программный код,
    // зависящий от реализации
public:
    Complex(float, float);
    float Re() const;
    float Im() const;
    Complex& operator*=(Complex&);
};
```

Когда в программе 4.17 мы полагаем  $x = t$ , где  $x$  и  $t$  являются объектами класса **Complex**, система распределяет память для нового объекта и копирует в новый объект значения, относящиеся к объекту  $t$ . Если использовать объект класса **Complex** как аргумент или возвращаемое значение функции, процесс будет таким же. Кроме того, когда объект выходит за пределы области видимости, система освобождает связанную с ним память. Например, в программе 4.17 система освобождает память, связанную с объектами  $t$  и  $x$  класса **Complex**, после цикла **for** так же, как и память, связанную с переменной  $g$  типа **float**. Коротко говоря, класс **Complex** используется подобно встроенным типам данных, т.е. он относится к типам данных первого класса.

0	1.000	0.000	1.000	0.000
1	0.707	0.707	1.000	0.000
2	0.000	1.000	1.000	0.000
3	-0.707	0.707	1.000	0.000
4	-1.000	0.000	1.000	0.000
5	-0.707	-0.707	1.000	0.000
6	0.000	-1.000	1.000	0.000
7	0.707	-0.707	1.000	0.000

### РИСУНОК 4.12 КОМПЛЕКСНЫЕ КОРНИ ИЗ ЕДИНИЦЫ

Эта таблица содержит выходные данные программы 4.17, когда она вызывается с параметрами **a.out 8**, а реализация перегруженной операции  $<<$  выполняет соответствующее форматирование выходных данных (см. упр. 4.70). Восемь комплексных корней из единицы равны:  $\pm 1, \pm i$  и

$$\pm \frac{\sqrt{2}}{2} \pm \frac{\sqrt{2}}{2} i$$

(два левых столбца). При возведении в восьмую степень все эти восемь чисел дают в результате  $1 + 0i$  (два левых столбца).

---

**Программа 4.19 АТД первого класса для комплексных чисел**

---

Этот код реализует АТД, определенный в программе 4.18. Для представления вещественной и мнимой частей комплексного числа используются данные типа `float`. Это тип данных первого класса, так как в представлении данных отсутствуют указатели. Когда объект класса `Complex` применяется либо в операторе присваивания, либо в аргументе функции, либо как ее значение возврата, система делает его копию, размещая в памяти новый объект и копируя данные в точности как в случае встроенных типов данных.

Перегруженная операция `<<` в текущей реализации выходные данные не форматирует (см. упр. 4.70).

```
#include <iostream.h>
class Complex
{
private:
    float re, im;
public:
    Complex(float x, float y)
        { re = x; im = y; }
    float Re() const
        { return re; }
    float Im() const
        { return im; }
    Complex& operator*=(const Complex& rhs)
        { float t = Re();
          re = Re()*rhs.Re() - Im()*rhs.Im();
          im = t*rhs.Im() + Im()*rhs.Re();
          return *this;
        }
};
ostream& operator<<(ostream& t, const Complex& c)
{ t << c.Re() << " " << c.Im(); return t; }
```

---

Действительно, в языке C++ *любой* класс, обладающий свойством, что ни один из его данных-членов не является указателем, относится к типам данных первого класса. При копировании объекта копируется каждый его член; когда объекту присваивается значение какого-нибудь иного объекта, то каждый его член перезаписывается; когда объект выходит за пределы области видимости, занимаемая им память освобождается. В системе существуют стандартные механизмы функционирования в каждой из упомянутых ситуаций: для выполнения необходимых функций в каждом классе имеются используемые по умолчанию *конструктор копирования, операция присваивания и деструктор*.

Однако, когда некоторые данные-члены являются указателями, эффект от выполнения функций, используемых по умолчанию, окажется совершенно другим. В операции присваивания конструктор копирования, используемый по умолчанию, делает копию указателей, но действительно ли это то, что нам требуется? Это важный вопрос *семантики копирования*, который необходимо задавать себе при проектировании любого АТД. Или, если говорить более широко, вопрос управления памятью является решающим, когда при разработке программного обеспечения используются АТД. Далее приводится пример, который поможет осветить эти вопросы более детально.

Программа 4.20 является примером клиентской программы, которая оперирует с очередями FIFO как с типами данных первого класса. Она моделирует процесс поступления и обслуживания клиентов в совокупности  $M$  очередей. На рис. 4.13 показан пример выходных данных этой программы. Интерес к этой программе объясняется желанием проиллюстрировать, как механизм типов данных первого класса позволяет работать с таким высокоуровневым объектом, как очередь — можно также представить себе написание подобных программ для проверки различных методов организации очереди по обслуживанию клиентов и т.п.

Предположим, что используется рассмотренная ранее реализация очереди на базе связного списка из программы 4.14. Когда встречается запись  $p = q$ , где  $p$  и  $q$  являются объектами класса **QUEUE**, система распределяет память для нового объекта и копирует в новый объект значения, относящиеся к объекту  $q$ . Но это указатели **head** и **tail** — сам связный список *не* копируется. Если впоследствии связный список, относящийся к объекту  $p$ , изменяется, тем самым изменяется и связный список, относящийся к объекту  $q$ . Безусловно, в программе 4.20 результат должен быть не таким. Опять же, если использовать объект класса **QUEUE** как аргумент функции, процесс будет выглядеть так же. В случае встроенных типов данных мы рассчитываем на то, что внутри функции будет свой собственный объект, который можно использовать по своему усмотрению. Следствием этих ожиданий будет то, что в случае структуры данных с указателями потребуются делать копию. Однако система не знает, как это сделать — именно мы должны предоставить необходимый код. То же самое справедливо и для возвращаемых значений функций.

### Программа 4.20 Клиентская программа, моделирующая очередь

Данная клиентская программа моделирует ситуацию, при которой клиенты, ожидающие обслуживания, случайным образом помещаются в одну из  $M$  очередей обслуживания, затем, опять-таки случайным образом, выбирается очередь (возможно, та же) и, если она не пуста, выполняется обслуживание (клиент из очереди удаляется). Каждый раз после выполнения перечисленных операций выводится номер добавленного клиента, номер обслуженного клиента и содержимое очередей.

```
75 in 74 out
0: 58 59 60 67 68 73
1:
2: 64 66 72
3: 75
```

```
76 in
0: 58 59 60 67 68 73
1:
2: 64 66 72
3: 75 76
```

```
77 in 58 out
0: 59 60 67 68 73
1: 77
2: 64 66 72
3: 75 76
```

```
78 in 77 out
0: 59 60 67 68 73
1: 78
2: 64 66 72
3: 75 76
```

```
79 in 78 out
0: 59 60 67 68 73
1: 79
2: 64 66 72
3: 75 76
```

### РИСУНОК 4.13 МОДЕЛИРОВАНИЕ НЕУПОРЯДОЧЕННОЙ ОЧЕРЕДИ

*Данный листинг представляет заключительную часть выходных данных программы 4.20 для случая, когда в командной строке вводится аргумент 80. Он отображает содержимое очередей после указанных операций, которые заключаются в следующем: случайным образом выбирается очередь и в нее заносится следующий элемент; затем еще раз выбирается очередь (также случайно) и, если она не пуста, из нее извлекается элемент.*



Здесь неявно предполагается, что класс **QUEUE** принадлежит в типу данных первого класса. Эта программа не будет корректно функционировать с реализациями, предоставленными в программах 4.14 и 4.15, по причине неправильной семантики копирования во внутреннем цикле **for**.

Реализация АД "Очередь" в программе 4.22 имеет конструктор копирования, который исправляет этот дефект. Во внутреннем цикле **for** эта реализация каждый раз делает соответствующую копию для объекта **q** и полагается на то, что ее деструктор позволит системе освободить память, занятую копиями.

```
#include <iostream.h>
#include <stdlib.h>
#include "QUEUE.cxx"
static const int M = 4;
int main(int argc, char *argv[])
{ int N = atoi(argv[1]);
  QUEUE<int> queues[M];
  for (int i = 0; i < N; i++, cout << endl)
    { int in = rand() % M, out = rand() % M;
      queues[in].put(i);
      cout << i << " in ";
      if (!queues[out].empty())
        cout << queues[out].get() << " out";
      cout << endl;
      for (int k = 0; k < M; k++, cout << endl)
        { QUEUE<int> q = queues[k];
          cout << k << ": ";
          while (!q.empty())
            cout << q.get() << " ";
        }
    }
}
```

Еще хуже обстоят дела, когда объект класса **QUEUE** выходит за пределы видимости: система освобождает память, относящуюся к указателям, но *не* всю память, занимаемую собственно связным списком. Действительно, как только указатели прекращают свое существование, исчезает возможность доступа к данной области памяти. Вот вам пример *утечки памяти (memory leak)* — серьезной проблемы, всегда требующей особого внимания во время написания программы, которая имеет дело с распределением памяти.

В языке C++ существуют специальные механизмы, упрощающие создание классов, имеющих корректную семантику копирования и предотвращающих утечку памяти. В частности, в классы следует включать такие функции-члены:

- *Конструктор копирования* — для того чтобы создавать новый объект, который является копией данного объекта.
- *Перегруженную операцию присваивания* — для того чтобы объект мог присутствовать с левой стороны оператора присваивания.
- *Деструктор* — для того чтобы освободить память, выделенную под объект во время его создания.

Когда системе необходимо выполнить указанные операции, она использует эти функции-члены. Если они не включены в класс, система обращается к стандартным

функциям. Они работают так, как это описано для класса **Complex**, однако ведут к неверной семантике копирования и утечке памяти, если какие-нибудь данные-члены являются указателями. Программа 4.21 — суть интерфейс очереди FIFO, в который включены три перечисленных функции. Подобно конструкторам, они имеют отличительные сигнатуры, в которые входит имя класса.

#### Программа 4.21 Интерфейс АД первого класса "Очередь"

Чтобы определяемый пользователем класс, данные-члены которого могут содержать указатели, больше походил на встроенный тип данных, в его интерфейс необходимо включить конструктор копирования, перегруженную операцию присваивания и деструктор, как это сделано в данном расширении простого интерфейса очереди FIFO из программы 4.13.

```
template <class Item>
class QUEUE
{
private:
    // Implementation-dependent code
public:
    QUEUE(int);
    QUEUE(const QUEUE&);
    QUEUE& operator=(const QUEUE&);
    ~QUEUE();
    int empty() const;
    void put(Item);
    Item get();
};
```

Когда при создании объекту присваивается начальное значение либо объект передается как параметр или возвращается из функции, система автоматически вызывает конструктор копирования **QUEUE(const QUEUE&)**. Операция присваивания **QUEUE& operator=(const QUEUE&)** вызывается в случае применения операции **=**, дабы присвоить значение одной очереди другой. Деструктор **~QUEUE()** вызывается в том случае, если необходимо освободить память, связанную с какой-либо очередью. Если в программе присутствует объявление без установки начального значения, например **QUEUE<int> q;**, то для создания объекта **q** система использует конструктор **QUEUE()**. Если объект инициализируется с помощью такого объявления, как **QUEUE<int> q = p** (или эквивалентной формы **QUEUE<int> q(p)**), система использует конструктор копирования **QUEUE(const QUEUE&)**. Эта функция должна создавать новую копию объекта **p**, а не просто еще один указатель на него. Как обычно для ссылочных параметров, ключевое слово **const** выражает намерение не изменять объект **p**, но использовать его только для доступа к хранящейся в нем информации.

Программа 4.22 представляет собой реализацию конструктора копирования, перегруженной операции присваивания и деструктора для реализации очереди на базе связного списка из программы 4.14. Деструктор проходит по всей очереди и с помощью **delete** освобождает память, распределенную под каждый узел. Если клиентская программа присваивает значения некоторого объекта самому себе, то в операции присваивания не предпринимаются каких-либо действий; в противном случае вызывается деструктор, а затем с помощью **put** копируется каждый элемент очереди, сто-

ящей справа от знака операции. Конструктор копирования обнуляет очередь (т.е. делает ее пустой) и затем с помощью операции присваивания выполняет копирование.

За счет помещения в класс конструктора копирования, перегруженной операции присваивания и деструктора (аналогично тому, как это делается в программе 4.22), можно превратить любой класс языка C++ в АД первого класса. Эти функции обычно действуют по принципу простого обхода структур данных. Однако эти дополнительные шаги предпринимаются не всегда, поскольку

- часто используется только один экземпляр объекта определенного класса;
- даже при условии присутствия нескольких экземпляров может оказаться необходимым предотвращение непреднамеренного копирования огромных структур данных.

Короче говоря, осознав возможность создания типов данных первого класса, мы сознаем также необходимость достижения компромисса между качеством и ценой, которую приходится платить за его реализацию и использование, в особенности, когда дело касается большого объема данных.

#### Программа 4.22 Реализация очереди первого класса на базе связанного списка

Есть возможность модернизировать реализацию класса "Очередь FIFO" из программы 4.14 и превратить этот тип данных в принадлежащий первому классу. Для этого в класс потребуется добавить приведенные ниже реализации конструктора копирования, перегруженной операции присваивания и деструктора. Эти функции перекрывают функции, используемые по умолчанию, и вызываются, когда необходимо копировать или уничтожать объекты.

Деструктор `-QUEUE()` вызывает приватную функцию-член `deletelist`, которая проходит по всему связанному списку, вызывая для каждого узла функцию `delete`. Таким образом, когда освобождается память, связанная с указателями, освобождается также вся память, занимаемая объектом.

Если перегруженная операция присваивания вызывается для случая, когда объект присваивается самому себе (например, `q = q`), никакие действия не выполняются. В противном случае вызывается функция `deletelist`, чтобы очистить память, связанную со старой копией объекта (в результате получается пустой список). Затем для каждого элемента списка, который соответствует объекту, находящемуся справа от знака операции присваивания, вызывается `put` и таким образом создается копия этого списка. В обоих случаях возвращаемое значение представляет собой ссылку на целевой объект (объект, которому присваиваются значения другого объекта).

Конструктор копирования `QUEUE(const QUEUE&)` обнуляет список и затем с помощью перегруженной операции присваивания создает копию своего аргумента.

```
private:
    void deletelist()
    {
        for (link t = head; t != 0; head = t)
            { t = head->next; delete head; }
    }
public:
    QUEUE(const QUEUE& rhs)
    { head = 0; *this = rhs; }
    QUEUE& operator=(const QUEUE& rhs)
```

```
{
    if (this == &rhs) return *this;
    deletelist();
    link t = rhs.head;
    while (t != 0)
        { put(t->item); t = t->next; }
    return *this;
}
~QUEUE()
{ deletelist(); }
```

---

Приведенные рассуждения также объясняют, почему библиотека языка C++ включает класс **string**, несмотря на привлекательность низкоуровневой структуры данных типа строки в стиле языка C. C-строки не принадлежат к типам данных первого класса, поскольку они не особенно лучше указателей. В самом деле, многие программисты впервые сталкиваются с некорректной семантикой копирования тогда, когда копируют указатель на C-строку, ожидая, что будет копироваться сама строка. В противоположность этому, класс **string** языка C++ является типом данных первого класса, поэтому при работе с большими строками следует соблюдать особую осторожность.

В качестве другого примера можно изменить программу 4.20 таким образом, чтобы она периодически распечатывала лишь несколько первых элементов каждой очереди, и можно было бы отслеживать продвижение очередей даже тогда, когда они становятся очень большими. Однако, когда очереди будут очень большими, производительность программы существенно снизится, поскольку при инициализации локальной переменной в цикле **for** вызывается конструктор копирования, который создает копию всей очереди, даже если требуется всего лишь несколько элементов. Далее, в конце каждой итерации цикла память, занимаемая очередью, освобождается деструктором, так как она связана с локальной переменной. Для программы 4.20 в ее нынешнем виде, когда выполняется доступ к каждому элементу копии, дополнительные затраты на выделение и освобождение памяти увеличивают время выполнения только на некоторый постоянный коэффициент. Однако, если требуется доступ всего лишь к нескольким элементам очереди, платить столь высокую цену было бы неразумно. Пожалуй, в такой ситуации для операции *копировать* большее предпочтение стоит отдать реализации, используемой по умолчанию, которая копирует только указатели, и добавить в этот АТД операции, обеспечивающих доступ к элементам очереди без ее модификации.

Утечка памяти — это трудно выявляемый дефект, который словно чума поражает многие большие системы. Хотя освобождение памяти, занятой некоторым объектом, обычно является делом, в принципе, простым, на практике очень тяжело быть уверенным в том, что удалось отыскать все вплоть до последней распределенной области памяти. Механизм деструкторов в языке C++ полезен, однако система не может гарантировать, что обход структур данных совершен так, как задумывалось. Когда объект прекращает свое существование, его указатели теряются безвозвратно, и любой указатель, "оставленный без внимания" деструктором, потенциально ведет к утечке памяти. Один из типовых источников утечки памяти возникает тогда, когда в

классе вообще забывают определить деструктор — конечно же, не стоит надеяться, что деструктор, используемый по умолчанию, корректно очистит память. Особое внимание этому вопросу необходимо уделить в ситуациях, когда используются абстрактные классы, наподобие класса из программы 4.12.

В ряде систем существует *автоматическое распределение памяти* — в таких случаях система самостоятельно определяет, какая область памяти больше не используется программами, после чего освобождает ее. Ни одно из этих решений не является полностью удобоваримым. Одни полагают, что распределение памяти — слишком важное дело, чтобы его можно было поручать системе; другие же считают, что это слишком важное дело, чтобы его можно было поручать программистам.

Список вопросов, которые могут возникнуть при рассмотрении реализаций АД, будет очень длинным даже в случае таких простых АД, как те, которые рассматриваются в настоящей главе. Нужна ли нам возможность поддерживать в одной очереди хранение объектов разных типов? Требуется ли нам в одной клиентской программе использовать разные реализации для очередей одного и того же типа, если известно, что они характеризуются разной производительностью? Следует ли включать в интерфейс информацию об эффективности реализаций? Какую форму должна иметь эта информация? Подобного рода вопросы подчеркивают, насколько важно уметь разбираться в основных характеристиках алгоритмов и структур данных, и понимать, каким образом клиентские программы могут эффективно их использовать. В некотором смысле, именно этом и является темой книги. Хотя полные реализации часто представляют собой упражнения по технике программирования, а не по проектированию алгоритмов, мы стремимся не забывать об этих существенных вопросах с тем, чтобы разработанные алгоритмы и структуры данных могли послужить базисом для создания инструментальных программных средств в самых разнообразных приложениях (см. раздел ссылок).

## Упражнения

- ▷ 4.63 Перегрузите операции  $+$  и  $+=$  для работы с комплексными числами (программы 4.18 и 4.19).
- 4.64. Преобразуйте АД "Отношения эквивалентности" (из раздела 4.5) в тип данных первого класса.
- 4.65 Создайте АД первого класса для использования в программах, оперирующих с игральными картами.
- 4.66 Используя АД из упражнения 4.65, напишите программу, которая опытным путем определит вероятность раздачи различных наборов карт при игре в покер.
- 4.67 Разработайте реализацию для АД "Комплексное число" на базе представления комплексных чисел в полярных координатах (т.е. в форме  $re^{i\theta}$ ).
- 4.68 Воспользуйтесь тождеством  $e^{i\theta} = \cos\theta + i\sin\theta$  для доказательства того, что  $e^{2\pi i} = 1$ , а  $N$  комплексных корней  $N$ -ой степени из единицы равны

$$\cos\left(\frac{2\pi k}{N}\right) + i\sin\left(\frac{2\pi k}{N}\right),$$

для  $k = 0, 1, \dots, N - 1$ .

- 4.69 Перечислите корни  $N$ -ой степени из единицы для значений  $N$  от 2 до 8.
- 4.70 С использованием `precision` и `setw` из файла `iostream.h` создайте реализацию перегруженной операции `<<` для программы 4.19, которая выдаст выходные данные, показанные на рис. 4.12 для программы 4.17.
  - ▷ 4.71 Опишите точно, что происходит в результате запуска программы 4.20 моделирования очереди, используя такую простую реализацию, как программа 4.14 или 4.15.
  - 4.72 Разработайте реализацию данного в тексте АДТ первого класса "Очередь FIFO" (программа 4.21), в которой в качестве базовой структуры данных используется массив.
  - ▷ 4.73 Напишите интерфейс АДТ первого класса для стека.
  - 4.74 Разработайте реализацию АДТ первого класса для стека из упражнения 4.73, которая в качестве базовой структуры данных использует массив.
  - 4.75 Разработайте реализацию АДТ первого класса для стека из упражнения 4.73, которая в качестве базовой структуры данных использует связный список.
  - 4.76 Используя приведенные ранее АДТ первого класса для комплексных чисел (программы 4.18 и 4.19), модифицируйте программу вычисления постфиксных выражений из раздела 4.3 таким образом, чтобы она вычисляла постфиксные выражения, состоящие из комплексных чисел с целыми коэффициентами. Для простоты считайте, что все комплексные числа имеют ненулевые целые коэффициенты как для вещественной, так и для мнимой частей, и записываются без пробелов. Например, для исходного выражения
 
$$1+2i \ 0+1i \ + \ 1-2i \ * \ 3+4i \ + \ .$$
 программа должна вывести результат  $8+4i$ .
  - 4.77 Выполните математический анализ процесса моделирования очереди в программе 4.20, чтобы определить вероятность (как функцию аргументов  $N$  и  $M$ ) того, что очередь, выбранная для  $N$ -ой операции `get`, будет пустой, а также ожидаемое количество элементов в этих очередях после  $N$  итераций цикла `for`.

## 4.9 Пример использования АДТ в приложении

В качестве заключительного примера рассмотрим специализированный АДТ, который характеризует отношения между областями применения и типами алгоритмов и структур данных, которые обсуждаются в настоящей книге. Этот пример АДТ, представляющего *полином*, взят из области *символической логики*, в которой компьютеры помогают оперировать с абстрактными математическими объектами. Цель заключается в том, чтобы получить возможность писать программы, которые могут оперировать с полиномами и выполнять вычисления, наподобие

$$\left(1 - x + \frac{x^2}{2} - \frac{x^3}{6}\right) \left(1 + x + x^2 + x^3\right) = 1 + \frac{x^2}{2} + \frac{x^3}{3} - \frac{2x^4}{3} + \frac{x^5}{3} - \frac{x^6}{6}.$$

Кроме того, необходима возможность вычислять полиномы для заданного значения  $x$ . Для  $x = 0.5$  обе стороны приведенного выше уравнения имеют значение 1.1328125. Операции умножения, сложения и вычисления полиномов являются центральными операциями в огромном числе математических вычислений. Программа 4.23 — это простой пример, в котором выполняются символические операции, соответствующие полиномиальным уравнениям

$$\begin{aligned}(x + 1)^2 &= x^2 + 2x + 1, \\(x + 1)^3 &= x^3 + 3x^2 + 3x + 1, \\(x + 1)^4 &= x^4 + 4x^3 + 6x^2 + 4x + 1, \\(x + 1)^5 &= x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1, \\&\dots\end{aligned}$$

Упомянутые основные идеи можно развить дальше и включить такие операции, как сложение, интегрирование, дифференцирование, сведения о специальных функциях и т.п.

#### Программа 4.23 Клиентская программа для АТД "Полином" (биномиальные коэффициенты)

Эта клиентская программа использует АТД "Полином", определенный в интерфейсе (программа 4.24), для выполнения алгебраических операций над полиномами с целыми коэффициентами. В программу из командной строки вводится целое число  $N$  и число с плавающей точкой  $p$ . Затем она вычисляет  $(x + 1)^N$  и проверяет результат, вычисляя значение результирующего полинома для  $x = p$ .

```
#include <iostream.h>
#include <stdlib.h>
#include "POLY.cxx"
int main(int argc, char *argv[])
{ int N = atoi(argv[1]); float p = atof(argv[2]);
  cout << "Binomial coefficients" << endl;
  POLY<int> x(1,1), one(1,0), t = x + one, y = t;
  for (int i = 0; i < N; i++)
    { y = y*t; cout << y << endl; }
  cout << y.eval(p) << endl;
}
```

Первый шаг — определить АТД "Полином" так, как это иллюстрируется интерфейсом из программы 4.24. Для такой хорошо понятной математической абстракции, как полином, спецификация настолько ясна, что ее не стоит выражать словесно (так же, как и в случае АТД для комплексных чисел, который обсуждался в разделе 4.8): необходимо, чтобы экземпляры АТД вели себя в точности так же, как и эта хорошо понятная математическая абстракция.

Для реализации функций, определенных в этом интерфейсе, потребуется выбрать конкретную структуру данных для представления полиномов и затем реализовать соответствующие алгоритмы для работы с этой структурой данных, дабы полученная реализация функционировала в соответствии с ожиданиями клиентской программы. Как обычно, выбор структуры данных влияет на потенциальную эффективность алгоритмов, поэтому стоит обдумать несколько вариантов. Так же как для стеков и оче-

редей, можно воспользоваться представлением на базе связного списка либо представлением на базе массива. Программа 4.25 — это реализация, в которой используется представление на базе массива; реализация на базе связного списка остается на самостоятельную проработку (см. упр. 4.78).

Для сложения (*add*) двух полиномов складываются их коэффициенты. Если полиномы представлены в виде массивов, функция *add*, как показано в программе 4.25, равнозначна одиночному циклу по этим массивам. Для умножения (*multiply*) двух полиномов применяется элементарный алгоритм, основанный на законе распределения. Один полином умножается на каждый член другого, результаты располагаются так, чтобы степени  $x$  соответствовали друг другу, и затем складываются для получения окончательного результата. В следующей таблице кратко показывается этот вычислительный процесс для  $(1 - x + x^2/2 - x^3/6)(1 + x + x^2 + x^3)$ :

$$\begin{array}{r}
 1 - x + \frac{x^2}{2} - \frac{x^3}{6} \\
 + x - x^2 + \frac{x^3}{2} - \frac{x^4}{6} \\
 + x^2 - x^3 + \frac{x^4}{2} - \frac{x^5}{6} \\
 + x^3 - x^4 + \frac{x^5}{2} - \frac{x^6}{6} \\
 \hline
 1 + \frac{x^2}{2} + \frac{x^3}{3} - \frac{2x^4}{3} + \frac{x^5}{3} - \frac{x^6}{6}
 \end{array}$$

Время, необходимое для умножения таким способом двух полиномов, по-видимому, пропорционально  $N^2$ . Отыскать более быстрый алгоритм решения этой задачи весьма непросто. Эта тема рассматривается более подробно в части 8, где будет показано, что время, необходимое для решения такой задачи с помощью алгоритма "разделяй-и-властвуй", пропорционально  $N^{3/2}$ , а время, необходимое для ее решения с помощью быстрого преобразования Фурье, пропорционально  $N \lg N$ .

#### Программа 4.24 Интерфейс АТД "Полином"

Для того чтобы можно было задавать коэффициенты различных типов, в этом интерфейсе АТД "Полином" используется шаблон. Здесь также перегружаются бинарные операции  $+$  и  $*$ , поэтому в арифметических выражениях такие операции можно задавать и для полиномов. Конструктор, вызванный с аргументами  $c$  и  $N$ , создает полином, соответствующий выражению  $cx^N$ .

```

template <class Number>
class POLY
{
private:
    // программный код, зависящий от реализации
public:
    POLY<Number>(Number, int);
    float eval(float) const;
    friend POLY operator+(POLY &, POLY &);
    friend POLY operator*(POLY &, POLY &);
};

```



В реализации функции *evaluate* (вычислить) из программы 4.25 используется эффективный классический алгоритм, известный как *алгоритм Горнера* (*Horner's algorithm*). Самая простая реализация этой функции заключается в непосредственном вычислении выражения с использованием функции, вычисляющей  $x^N$ . При таком подходе требуется время с квадратичной зависимостью от  $N$ . В более сложной реализации значения  $x^i$  запоминаются в таблице и затем используются при непосредственных вычислениях. В данной ситуации требуется дополнительный объем памяти, линейно зависящий от  $N$ . Алгоритм Горнера — это прямой оптимальный линейный алгоритм, основанный на следующем использовании круглых скобок:

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = (((a_4x + a_3)x + a_2)x + a_1)x + a_0.$$

Алгоритм Горнера часто представляют как ловкий прием, экономящий время, но в действительности — это первый выдающийся пример элегантного и эффективно-го алгоритма, который сокращает время, необходимое для выполнения этой важной вычислительной задачи, делая его не квадратично, но линейно зависимым от  $N$ . Преобразование строк с ASCII-символами в целые числа, выполняемое в программе 4.5, является разновидностью алгоритма Горнера. Мы снова встретимся с алгоритмом Горнера в главе 14 и части 5, где он выступает в качестве основы для важного вида вычислений, относящихся к некоторым реализациям таблиц символов и поиску строк.

#### Программа 4.25 Реализация АД "Полином" на базе массива

В этой реализации АД для полиномов представление данных состоит из степени и указателя на массив коэффициентов. Это не АД первого класса: клиентские программы должны знать, что возможна утечка памяти, а семантика копирования заключается в копировании указателей (см. упр. 4.79).

```
template <class Number>
class POLY
{
private:
    int n; Number *a;
public:
    POLY<Number>(Number c, int N)
    { a = new Number[N+1]; n = N+1; a[N] = c;
      for (int i = 0; i < N; i++) a[i] = 0;
    }
    float eval(float x) const
    { double t = 0.0;
      for (int i = n-1; i >= 0; i--)
          t = t*x + a[i];
      return t;
    }
    friend POLY operator+(POLY &p, POLY &q)
    { POLY t(0, p.n > q.n ? p.n-1 : q.n-1);
      for (int i = 0; i < p.n; i++)
          t.a[i] += p.a[i];
      for (int j = 0; j < q.n; j++)
          t.a[j] += q.a[j];
      return t;
    }
}
```

```

friend POLY operator*(POLY &p, POLY &q)
{ POLY t(0, (p.n-1)+(q.n-1));
  for (int i = 0; i < p.n; i++)
    for (int j = 0; j < q.n; j++)
      t.a[i+j] += p.a[i]*q.a[j];
  return t;
}
};

```

В ходе выполнения перегруженных операций  $+$  и  $*$  создаются новые полиномы, поэтому данная реализация связана с утечкой памяти. Утечку памяти можно легко ликвидировать, добавляя в реализацию конструктор копирования, перегруженную операцию присваивания и деструктор. Так бы стоило и поступить в случае полиномов очень больших размеров, обработки огромного количества небольших полиномов, а также создания АДТ для использования в каком-нибудь приложении (см. упр. 4.79).

Использование в реализации АДТ "Полином" представления на базе массива — это, как обычно, лишь одна из возможностей. Если показатели степени очень большие, а членов в полиномах немного, то представление на базе связного списка может оказаться более подходящим. Например, не стоило бы применять программу 4.25 для выполнения такого умножения:

$$(1 + x^{1000000})(1 + x^{2000000}) = 1 + x^{1000000} + x^{2000000} + x^{3000000},$$

поскольку в ней будет использоваться массив с выделенным пространством под миллионы неиспользуемых коэффициентов. В упражнении 4.78 вариант со связным списком рассматривается более подробно.

## Упражнения

- 4.78 Напишите реализацию для приведенного в тексте АДТ "Полином" (программа 4.24), в которой в качестве базовой структуры данных используются связные списки. Списки не должны содержать узлов, соответствующих членам с нулевыми коэффициентами.
- 4.79 Устраните утечку памяти в программе 4.25, добавив в нее конструктор копирования, перегруженную операцию присваивания и деструктор.
- ▷ 4.80 Добавьте перегруженные операции  $+=$  и  $*=$  в АДТ "Полином" из программы 4.25.
- 4.81 Расширьте рассмотренный в главе АДТ "Полином", включив в него операции интегрирования и дифференцирования полиномов.
- 4.82 Модифицируйте полученный АДТ "Полином" из упражнения 4.81 так, чтобы в нем игнорировались все члены с экспонентами, большими или равными целому числу  $M$ , которое поступает в клиентскую программу во время инициализации.
- 4.83 Расширьте АДТ "Полином" из упражнения 4.81 так, чтобы он включал деление и сложение полиномов.
- 4.84 Разработайте АДТ, который позволяет клиентским программам выполнять сложение и умножение целых чисел произвольной точности.

- **4.85** Используя АТД, разработанный в упражнении 4.84, модифицируйте программу вычисления постфиксных выражений из раздела 4.3 так, чтобы она могла вычислять постфиксные выражения, состоящие из целых чисел произвольной точности.
- **4.86** Напишите клиентскую программу, которая с помощью АТД "Полином" из упражнения 4.83 вычисляет интегралы, используя разложение функций в ряды Тейлора и оперируя с ними в символической форме.
- 4.87** Разработайте АТД, который позволяет клиентским программам выполнять алгебраические операции над векторами чисел с плавающей точкой.
- 4.88** Разработайте АТД, который позволяет клиентским программам выполнять алгебраические операции над матрицами абстрактных объектов, для которых определены операции сложения, вычитания, умножения и деления.
- 4.89** Напишите интерфейс для АТД "Символьная строка", который включает операции создания строк, сравнения двух строк, конкатенации двух строк, копирования одной строки в другую и получения длины строки. *Примечание:* интерфейс должен быть похож на интерфейс, доступный в стандартной библиотеке C++.
- 4.90** Напишите реализацию для интерфейса из упражнения 4.89, используя там, где это необходимо строковую библиотеку C++.
- 4.91** Напишите реализацию для полученного в упражнении 4.89 интерфейса строки, используя представление на базе связного списка. Проанализируйте время выполнения каждой операции для наихудших случаев.
- 4.92** Напишите интерфейс и реализацию АТД "Множество индексов", в котором обрабатываются множества целых чисел в диапазоне от 0 до  $M - 1$  (где  $M$  — заданная константа) и имеются операции создания множества, объединения двух множеств, пересечения двух множеств, дополнения множества, разности двух множеств и вывода содержимого множества. Для представления каждого множества используйте массив из  $M - 1$  элементов, принимающих значения 0–1.
- 4.93** Напишите клиентскую программу, которая тестирует АТД, созданный в упражнении 4.92.

## 4.10 Перспективы

Приступая к изучению алгоритмов и структур данных, следует устойчиво владеть фундаментальными понятиями, лежащими в основе АТД. Рассмотрим три главных причины:

- АТД являются важными и широко используемыми инструментальными средствами разработки программного обеспечения, и многие из изучаемых алгоритмов служат реализациями широко применяемых фундаментальных АТД.
- АТД помогают инкапсулировать разрабатываемые алгоритмы, чтобы один и тот же код программы мог служить нам для самых разных целей.
- АТД предоставляют собой удобный механизм, который используется в процессе разработки алгоритмов и сравнения характеристик, связанных с их производительностью.

С теоретической точки зрения, АДТ воплощают простой (и здравый) принцип, заключающийся в том что мы обязаны точно описывать способы манипуляций с данными. Для выполнения подобной задачи в языке C++ имеется удобный механизм клиент-интерфейс-реализация, который подробно рассматривался в настоящей главе; он позволяет получать на языке C++ код, обладающий рядом желанных свойств. Во многих современных языках присутствуют специальные средства поддержки, позволяющие разрабатывать программы с упомянутыми свойствами, тем не менее, существует общий для разных языков подход — когда в языке отсутствуют специальные средства поддержки, устанавливаются определенные правила программирования, обеспечивающие требуемое разделение на клиентские программы, интерфейсы и реализации.

По мере рассмотрения постоянно растущего круга возможностей касательно задания характеристик абстрактных типов данных, приходится сталкиваться с непрерывно расширяющимся кругом проблем, связанных с созданием эффективных реализаций. Многочисленные рассмотренные ранее примеры иллюстрируют пути преодоления этих трудностей. Мы постоянно стремимся достичь следующей цели — эффективно реализовать все операции, однако, весьма маловероятно, чтобы в реализации общего назначения удалось сделать это для всех наборов операций. Подобного рода ситуация противоречит тем принципам, которые, в первую очередь, приводят к созданию абстрактных типов данных, так как во многих случаях разработчикам АДТ необходимо знать свойства клиентских программ для того, чтобы определять, какие реализации АДТ будут функционировать наиболее эффективно. А вот разработчикам клиентских программ следует располагать информацией о характеристиках производительности различных реализаций, дабы адекватно определять, какой реализации отдавать предпочтение в том или ином приложении. Как всегда, необходимо достичь некоторого баланса. В настоящей книге рассматриваются многочисленные подходы к реализации различных вариантов фундаментальных АДТ, каждый из которых имеет важное применение.

Один АДТ можно использовать для создания другого. Абстракции, подобные указателям и структурам, определенным в языке C++, использовались для построения связанных списков. Далее абстракции связанных списков и массивов, доступных в C++, применялись при построении стеков магазинного типа. Наконец, стеки магазинного типа использовались для организации вычислений арифметических выражений. Понятие АДТ позволяет создавать большие системы на основе разных уровней абстракции, от существующих в компьютере машинных инструкций, до разнообразных возможностей языка программирования, вплоть до сортировки, поиска и другой функциональности высокого уровня, которая обеспечивается алгоритмами (см. части 3 и 4). Некоторые приложения требуют еще более высокого уровня абстракции, поэтому стоит обратиться к частям с 5 по 8. Абстрактные типы данных — это лишь этап в истинной бесконечности создания все более и более мощных абстрактных механизмов, в чем и заключается суть эффективного использования компьютеров для решения современных задач.

## Рекурсия и деревья

**Р**екурсия относится к одному из фундаментальных понятий в математических и компьютерных науках. В языках программирования рекурсивной программой называют программу, которая обращается к самой себе (подобно тому, как в математике рекурсивной функцией называют функцию, которая определена через понятия самой этой функции). Рекурсивная программа не может вызывать себя до бесконечности, поскольку в этом случае она никогда не завершилась бы (точно так же рекурсивная функция не может всегда определяться понятиями самой функции, поскольку тогда определение стало бы циклическим). Следовательно, вторая важная особенность рекурсивной программы — наличие *условия завершения*, позволяющего программе прекратить вызывать себя (применительно к математике это условие, при выполнении которого функция перестает определяться понятиями самой этой функции). Все практические вычисления можно представить рекурсивными структурами.

Изучение рекурсии неразрывно связано с изучением рекурсивно определяемых структур, называемых *деревьями*. Деревья используются как для упрощения понимания и анализа рекурсивных программ, так и в качестве явных структур данных. В главе 1 мы уже встречались с применением деревьев (хотя и не рекурсивным). Взаимосвязь между рекурсивными программами и деревьями лежит в основе значительной части материала книги. Деревья используются для упрощения понимания рекурсивных программ; в свою очередь, рекурсивные программы используются для построения деревьев; в конечном итоге, глобальная взаимосвязь между ними (и рекуррентные отношения) применяется при анализе алгоритмов. Рекурсия

помогает разрабатывать изящные и эффективные структуры данных и алгоритмы для широчайшего спектра применений.

Основная цель этой главы заключается в исследовании рекурсивных программ и структур данных как практических инструментов. Вначале исследуется взаимосвязь между математической рекурсией и простыми рекурсивными программами и приводится ряд примеров применения рекурсии. Затем рассматривается фундаментальная рекурсивная схема, известная под названием "*разделяй и властвуй*", которая используется для решения задач общего характера в нескольких последующих разделах этой книги. После этого приводится общий подход к реализации рекурсивных программ, называемый *динамическим программированием*, который предоставляет эффективные и элегантные решения для обширного класса задач. Затем подробно рассматриваются деревья, их математические свойства и связанные с ними алгоритмы, в том числе базовые методы *обхода дерева (tree traversal)*, которые лежат в основе рекурсивных программ обработки деревьев. В завершение будут анализироваться тесно связанные с рекурсией алгоритмы обработки графов — в частности, особое внимание будет уделяться фундаментальной рекурсивной программе *поиска в глубину (depth-first search)*, которая служит основой для многих алгоритмов обработки графов.

Как будет показано далее, многие интересные алгоритмы легко и просто реализуются через рекурсивные программы, а многие разработчики алгоритмов предпочитают выражать методы рекурсивно. Кроме того, не менее подробно будут исследоваться и нерекурсивные методы реализации. Часто можно не только создать простые алгоритмы с использованием стеков, которые, в основном, эквивалентны рекурсивным, но и отыскать нерекурсивные альтернативы, которые приводят к такому же конечному результату через иную последовательность вычислений. Рекурсивная формулировка проблемы предоставляет структуру, в рамках которой доступны и более эффективные альтернативы.

Исчерпывающее исследование рекурсии и деревьев могло бы послужить темой отдельной книги, поскольку эти концепции находят применение во множестве компьютерных приложений, а также выходят далеко за рамки компьютерных наук. Фактически можно было бы сказать, что *вся эта книга посвящена освещению рекурсии и деревьев*, поскольку с фундаментальной точки зрения упомянутые темы затрагиваются в каждой главе.

## 5.1 Рекурсивные алгоритмы

*Рекурсивный алгоритм* — это алгоритм, решающий задачу путем решения одного или нескольких более узких вариантов той же задачи. Для реализации рекурсивных алгоритмов в C++ используются *рекурсивные функции* — функции, которые вызывают самих себя. Рекурсивные функции в C++ соответствуют рекурсивным определениям математических функций. Изучение рекурсии начнем с исследования программ, которые непосредственно вычисляют математические функции. Как будет показано, базовые механизмы можно расширить, что приведет к обобщенной парадигме программирования.

---

**Программа 5.1 Функция вычисления факториала (рекурсивная реализация)**

---

Эта рекурсивная функция вычисляет функцию  $N!$ , используя стандартное рекурсивное определение. Она возвращает правильное значение, когда вызывается с неотрицательным и достаточно малым аргументом  $N$ , чтобы  $N!$  можно было представить типом `int`.

```
int factorial(int N)
{
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```

---

Рекуррентные отношения (см. раздел 2.5) являются рекурсивно определенными функциями. Рекуррентное отношение задает функцию, областью допустимых значений которой являются неотрицательные числа, определяемые либо некоторыми начальными значениями, либо (рекурсивно) множеством ее собственных значений, полученных на основе меньших целых значений. Вероятно, наиболее известной из таких функций является функция вычисления факториала, которая определяется рекуррентным отношением

$$N! = N \cdot (N - 1)!, \quad \text{для } N \geq 1, \text{ причем } 0! = 1$$

Это определение непосредственно соответствует рекурсивной функции C++ в программе 5.1.

Программа 5.1 эквивалентна простому циклу. Например, следующий цикл `for` выполняет такое же вычисление:

```
for ( t = 1, i = 1; i <= N; i++) t *= i;
```

Как будет показано, рекурсивную программу всегда можно преобразовать в не-рекурсивную, которая выполняет такое же вычисление. И наоборот, используя рекурсию, любое вычисление, предполагающее выполнение циклов, можно реализовать, не прибегая к циклам.

Рекурсия используется ввиду того, что зачастую она позволяет выразить сложные алгоритмы в компактной форме без ущерба для эффективности. Например, рекурсивная реализация функции вычисления факториала избавляет от необходимости использования локальных переменных. В системах программирования, поддерживающих обращения к функциям, издержки рекурсивной реализации определяются механизмами, которые используют эквивалент встроенного стека. Большинство современных систем программирования имеют тщательно разработанные механизмы для выполнения такой задачи. Как будет показано, несмотря на это преимущество, очень легко можно создать рекурсивную функцию, которая окажется весьма неэффективной, и поэтому необходимо постараться, чтобы впоследствии не пришлось возиться с плохо поддающимися исправлениям реализациями.

---

**Программа 5.2 Сомнительная рекурсивная программа**

---

Если аргумент  $N$  является нечетным, эта функция вызывает саму себя с  $3N+1$  в качестве аргумента. Если  $N$  является четным, она вызывает себя с  $N/2$  в качестве аргумента. Для гарантированного завершения этой программы нельзя использовать индукцию, поскольку не каждый рекурсивный вызов использует аргумент, меньший заданного.

```
int puzzle(int N)
{
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
}
```

Программа 5.1 иллюстрирует базовые особенности рекурсивной программы: она вызывает саму себя (с меньшими значениями аргументов) и содержит условие завершения, при выполнении которого непосредственно вычисляет свое результирующее значение. Дабы убедиться в правильности работы программы, можно применить метод математической индукции:

- Программа вычисляет  $0!$  (исходное значение)
- Если допустить, что программа вычисляет  $k!$  для  $k < N$  (индуктивное предположение), то она вычисляет и  $N!$ .

Такого рода рассуждения могут привести к быстрому способу разработки алгоритмов для решения сложных задач.

В языках программирования, подобных C++, существует очень мало ограничений для видов создаваемых программ, но, как правило, использование рекурсивных функций ограничивается теми, которые позволяют проверять правильность своей работы методом математической индукции, как упоминалось выше. Хотя в этой книге вопрос формального подтверждения правильности не рассматривается, нас интересует объединение сложных программ для выполнения сложных задач, и поэтому требуется определенная гарантия, что задача будет решена корректно. Механизмы, подобные рекурсивным функциям, могут обеспечить такую гарантию, в то же время обеспечивая компактные реализации. В частности, если следовать правилам математической индукции, необходимо удостовериться, что создаваемые рекурсивные функции удовлетворяют двум основным условиям:

- Они должны явно решать задачу для исходного значения.
- В каждом рекурсивном вызове должны использоваться меньшие значения аргументов.

Эти утверждения являются спорными — они равнозначны тому, что мы должны располагать допустимой индуктивной проверкой для каждой создаваемой рекурсивной функции. Тем не менее, они служат полезным руководством при разработке реализаций.

Программа 5.2 — занятный пример, иллюстрирующий необходимость наличия индуктивного аргумента. Она представляет собой рекурсивную функцию, нарушающую

```
puzzle(3)
  puzzle(10)
    puzzle(5)
      puzzle(16)
        puzzle(8)
          puzzle(4)
            puzzle(2)
              puzzle(1)
```

#### РИСУНОК 5.1 ПРИМЕР ЦЕПОЧКИ РЕКУРСИВНЫХ ВЫЗОВОВ

*Эта вложенная последовательность вызовов функции со временем завершается, однако нельзя гарантировать, что рекурсивная функция, используемая в программе 5.2, не будет иметь произвольную глубину вложенности для какого-либо аргумента. Желательно использовать рекурсивные программы, которые всегда вызывают себя с меньшими значениями аргументов.*



правило, в соответствии с которым каждый рекурсивный вызов должен использовать меньшие значения аргументов, и поэтому для ее проверки нельзя использовать метод математической индукции. Действительно, неизвестно, завершается ли это вычисление для каждого значения  $N$ , поскольку значение  $N$  не имеет никаких пределов. Для меньших целочисленных значений, которые могут быть представлены значениями типа `int`, можно проверить, что программа прерывается (см. рис. 5.1 и упражнение 5.4), но для больших целочисленных значений (скажем, для 64-разрядных слов), не известно, входит ли эта программа в бесконечный цикл.

Программа 5.3 — компактная реализация *алгоритма Евклида* для отыскания наибольшего общего делителя для двух целых чисел. Алгоритм основывается на наблюдении, что наибольший общий делитель двух целых чисел  $x$  и  $y$ , когда  $x > y$ , совпадает с наибольшим общим делителем числа  $y$  и  $x$  по модулю  $y$  (остаток от деления  $x$  на  $y$ ). Число  $t$  делит и  $x$  и  $y$  тогда, и только тогда, когда оно делит и  $y$  и  $x \bmod y$  ( $x$  по модулю  $y$ ), поскольку  $x$  равно  $x \bmod y$  плюс число, кратное  $y$ . Рекурсивные вызовы, созданные для примера выполнения этой программы, показаны на рис. 5.2. При реализации алгоритма Евклида глубина рекурсии зависит от арифметических свойств аргументов (она связана с ними логарифмической зависимостью).

### Программа 5.3 Алгоритм Евклида

Это один из наиболее давно известных алгоритмов, разработанный свыше 2000 лет назад — рекурсивный метод отыскания наибольшего общего делителя двух целых чисел.

```
int gcd(int m, int n)
{
    if (n == 0) return m;
    return gcd(n, m % n);
}
```

Программа 5.4 — пример с несколькими рекурсивными вызовами. Она оценивает другое выражение, по существу выполняя те же вычисления, что и программа 4.2, но применительно к префиксным (а не постфиксным) выражениям, используя рекурсию вместо явного стека. В этой главе будут встречаться множество других примеров использования рекурсивных и эквивалентных программ, в которых задействуются стеки. Конкретная взаимосвязь между несколькими парами таких программ исследуется достаточно подробно.

### Программа 5.4 Рекурсивная программа для оценки префиксных выражений

Для оценки префиксного выражения либо осуществляется преобразование числа из ASCII в двоичную форму (в цикле `while`) в конце, либо над двумя операндами, которые оцениваются рекурсивно, выполняется операция, указанная первым символом

```
gcd(314159, 271828)
  gcd(271828, 42331)
    gcd(42331, 17842)
      gcd(17842, 6647)
        gcd(6647, 4458)
          gcd(4458, 2099)
            gcd(2099, 350)
              gcd(350, 349)
                gcd(349, 1)
                  gcd(1, 0)
```

### РИСУНОК 5.2 ПРИМЕР ПРИМЕНЕНИЯ АЛГОРИТМА ЭВКЛИДА

*Эта вложенная последовательность вызовов функции иллюстрирует работу алгоритма Евклида, показывая, что числа 314159 и 271828 являются взаимно простыми.*

выражения. Эта функция является рекурсивной, однако использует глобальный массив, содержащий выражение и индекс текущего символа выражения. Индекс изменяется после вычисления каждого подвыражения.

```
char *a; int i;
int eval()
{ int x = 0;
  while (a[i] == ' ') i++;
  if (a[i] == '+')
    { i++; return eval() + eval(); }
  if (a[i] == '*')
    { i++; return eval() * eval(); }
  while ((a[i] >= '0') && (a[i] <= '9'))
    x = 10*x + (a[i++] - '0');
  return x;
}
```

Пример обработки префиксного выражения программой 5.4 показан на рис. 5.3. Множество рекурсивных вызовов маскируют сложную серию вычислений. Подобно большинству рекурсивных программ, работу этой программы проще всего понять, воспользовавшись методом индукции: полагая, что она работает правильно применительно к простым выражениям, можно предположить, что это справедливо и применительно к сложным выражениям. Программа представляет собой простой пример *рекурсивного нисходящего анализатора* — аналогичный процесс используется для преобразования программ C++ в машинные коды.

Реальная проверка правильности оценки выражения программой 5.4 — гораздо более сложная задача, нежели проверка работы описанных функций с целочисленными аргументами, и в этой книге встретятся еще более сложные рекурсивные программы и структуры данных. Соответственно, мы не ставим перед собой идеалистичной задачи предоставления полных индуктивных доказательств правильности каждой создаваемой рекурсивной программы. В данном случае способность программы "узнавать", как следует разделять операнды в соответствии с заданным оператором, вначале кажется непостижимой (возможно потому, что не сразу понятно, как выполнить это разделение на верхнем уровне). Однако в действительности вычисление достаточно понятно (поскольку нужная ветвь программы при каждом вызове функции однозначно определяется первым символом выражения).

В принципе, любой цикл `for` можно заменить эквивалентной рекурсивной программой. Часто ре-

```
eval() * + 7 * * 4 6 + 8 9 5
eval() + 7 * * 4 6 + 8 9
eval() 7
eval() * * 4 6 + 8 9
eval() * 4 6
eval() 4
eval() 6
return 24 = 4*6
eval() + 8 9
eval() 8
eval() 9
return 17 = 8 + 9
return 408 = 24*17
return 415 = 7+408
eval() 5
return 2075 = 415*5
```

### РИСУНОК 5.3 ПРИМЕР ОЦЕНКИ ПРЕФИКСНОГО ВЫРАЖЕНИЯ

*Эта вложенная последовательность вызовов функций иллюстрирует работу рекурсивного алгоритма оценки префиксного выражения для конкретного примера выражения. Для простоты здесь показаны аргументы выражения. Сам по себе алгоритм никогда явно не определяет протяженность строки своих аргументов: вместо этого всю необходимую информацию он извлекает из начала строки.*

курсивная программа предоставляет более естественный способ выражения вычисления, чем цикл **for**, поэтому можно воспользоваться преимуществами механизма, предоставляемого системой, поддерживающей рекурсию. Однако, этот подход имеет один скрытый недостаток, о котором следует помнить. Как должно быть понятно из примеров, приведенных на рис. 5.1–5.3, при выполнении рекурсивной программы вызовы функций вкладываются один в другой, пока не будет достигнута точка, когда вместо рекурсивного вызова выполняется возврат. В большинстве сред программирования такие вложенные вызовы функций реализуются с помощью эквивалента встроженных стеков. Сущность подобного рода реализаций будет исследоваться на протяжении данной главы. *Глубина рекурсии* — это максимальная степень вложенности вызовов функций в ходе вычисления. В общем случае глубина будет зависеть от вводимых данных. Например, глубина рекурсии в примерах, приведенных на рис. 5.2 и 5.3, составляет, соответственно, 9 и 4. При использовании рекурсивной программы следует учитывать, что среда программирования должна поддерживать стек, *размер* которого пропорционален глубине рекурсии. При решении сложных задач необходимый для этого стека объем памяти может заставить отказаться от использования рекурсивного решения.

Структуры данных, построенные из узлов с указателями, рекурсивны изначально. Например, определение связных списков в главе 3 (определение 3.3) является рекурсивным. Следовательно, рекурсивные программы предоставляют естественные реализации для многих часто используемых функций, работающих с такими структурами данных. Программа 5.5 содержит четыре примера. Подобные реализации в книге используются часто, в основном потому, что их гораздо проще понять, чем их не-рекурсивные аналоги. Однако, рекурсивные программы, подобные программе 5.5, следует использовать обдуманно при обработке очень больших списков, поскольку глубина рекурсии может быть пропорциональна длине списков и, соответственно, требуемый для рекурсивного стека объем памяти может превысить допустимые пределы.

### **Программа 5.5 Примеры рекурсивных функций для связных списков**

Эти рекурсивные функции для выполнения простых задач обработки списков легко выразить, но они могут быть бесполезны для очень больших списков, поскольку глубина рекурсии может быть пропорциональна длине списка.

Первая функция **count** подсчитывает количество узлов в списке. Вторая, **traverse**, вызывает функцию **visit** для каждого узла списка, с начала до конца. Обе функции легко реализуются с помощью цикла **for** или **while**. Третья функция **traverseR** не имеет простого итеративного аналога. Она вызывает функцию **visit** для каждого узла списка, но в обратном порядке.

Четвертая функция **remove** удаляет из списка все узлы с заданным значением элемента. Основа реализации этой функции — изменение связи  $x = x \rightarrow \text{next}$  в узле, предшествующем удаляемому, что возможно благодаря использованию параметра ссылки. Структурные изменения для каждой итерации цикла **while** совпадают с показанными на рис. 3.3, в данном случае и **x**, и **t** ссылаются на один и тот же узел.

```
int count(link x)
{
    if (x == 0) return 0;
    return 1 + count(x->next);
}
```

```

void traverse(link h, void visit(link))
{
    if (h == 0) return;
    visit(h);
    traverse(h->next, visit);
}
void traverseR(link h, void visit(link))
{
    if (h == 0) return;
    traverseR(h->next, visit);
    visit(h);
}
void remove(link& x, Item v)
{
    while (x != 0 && x->item == v)
        { link t = x; x = x->next; delete t; }
    if (x != 0) remove(x->next, v);
}

```

Некоторые среды программирования автоматически выявляют и исключают *листовую (оконечную) рекурсию (tail recursion)*; когда последним действием функции является рекурсивный вызов, увеличение глубины рекурсии вовсе не обязательно. Это усовершенствование эффективно преобразовало бы функции подсчета, обхода и удаления, используемые в программе 5.5 в циклы, но оно не применимо к функции обхода в обратном направлении.

В разделах 5.2 и 5.3 рассматриваются два семейства рекурсивных алгоритмов, представляющие важные случаи вычислений. Затем, в разделах 5.4—5.7 будут исследоваться рекурсивные структуры данных, служащие основой для большой группы алгоритмов.

## Упражнения

- ▷ 5.1 Напишите рекурсивную программу для вычисления  $\lg(N!)$ .
- 5.2 Измените программу 5.1 для вычисления  $N! \bmod M$ , чтобы переполнение больше не играло никакой роли. Попробуйте выполнить программу для  $M = 997$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ , чтобы увидеть, как используемая система программирования обрабатывает рекурсивные вызовы с большой глубиной вложенности.
- ▷ 5.3 Приведите последовательности значений аргументов, получаемые в результате вызова программы 5.2 для каждого из целых чисел от 1 до 9.
- 5.4 Найдите значение  $N < 10^6$ , при котором программа 5.2 выполняет максимальное количество рекурсивных вызовов.
- ▷ 5.5 Создайте нерекурсивную реализацию алгоритма Эвклида.
- ▷ 5.6 Приведите рисунок, соответствующий рис. 5.2, для результата выполнения алгоритма Эвклида применительно к числам 89 и 55.
- 5.7 Укажите глубину рекурсии алгоритма Эвклида при вводе двух последовательных чисел Фибоначчи ( $F_N$  и  $F_{N+1}$ ).
- ▷ 5.8 Приведите рисунок, соответствующий рис. 5.3, для результата оценки префиксного выражения в случае ввода  $+ * * 12 12 12 144$ .

- 5.9** Создайте рекурсивную программу для оценки постфиксных выражений.
- 5.10** Создайте рекурсивную программу для оценки инфиксных выражений. Можете считать, что операнды всегда заключаются в круглые скобки.
- **5.11** Создайте рекурсивную программу, которая преобразует корневые выражения в постфиксные.
- **5.12** Создайте рекурсивную программу, которая преобразует постфиксные выражения в инфиксные.
- 5.13** Создайте рекурсивную программу для решения задачи Иосифа Флавия (Josephus) (см. раздел 3.3).
- 5.14** Создайте рекурсивную программу, которая удаляет конечный узел из связанного списка.
- **5.15** Создайте рекурсивную программу для изменения на обратный порядок следования узлов в связанном списке (см. программу 3.7). *Совет:* используйте глобальную переменную.

## 5.2 Разделяй и властвуй

Во множестве рассматриваемых в книге программ используется два рекурсивных вызова, каждый из которых работает приблизительно с половиной входных данных. Эта рекурсивная схема — вероятно, наиболее важный случай хорошо известного метода *"разделяй и властвуй"* разработки алгоритмов, который служит основой для важнейших алгоритмов.

Давайте в качестве примера рассмотрим задачу отыскания максимального из  $N$  элементов, сохраненных в массиве  $a[0], \dots, a[N-1]$ . Эту задачу можно легко выполнить за один проход массива демонстрируется в примере:

```
for (t = a[0], i = 1; i < N; i++)
    if (a[i] > t) t = a[i];
```

Рекурсивное решение типа *"разделяй и властвуй"*, приведенное в программе 5.6 — еще один простой (хотя и совершенно иной) алгоритм решения той же задачи; он использовался только с целью иллюстрации концепции *"разделяй и властвуй"*.

### Программа 5.6 Применение алгоритма *"разделяй и властвуй"* для отыскания максимума

Эта функция делит массив  $a[l], \dots, a[r]$  на массивы  $a[l], \dots, a[m]$  и  $a[m+1], \dots, a[r]$ , находит максимальные элементы в обеих частях (рекурсивно) и возвращает больший из них в качестве максимального элемента всего массива. В программе предполагается, что `Item` — тип первого класса, для которого операция  $>$  определена. Если размер массива является четным числом, обе части имеют одинаковые размеры, если же нечетным, эти размеры различаются на 1.

```
Item max(Item a[], int l, int r)
{
    if (l == r) return a[l];
    int m = (l+r)/2;
    Item u = max(a, l, m);
    Item v = max(a, m+1, r);
    If (u > v) return u; else return v;
}
```

Чаще всего подход "разделяй и властвуй" используют из-за того, что он обеспечивает более быстрые решения, чем простые итерационные алгоритмы (в конце главы рассматриваются несколько примеров таких алгоритмов); кроме того, данный подход заслуживает подробного исследования, поскольку он способствует пониманию сущности определенных базовых вычислений.

На рис. 5.4 показаны рекурсивные вызовы, выполняемые при запуске программы 5.6 применительно к примеру массива. Структура программы кажется сложной, но обычно об этом можно не беспокоиться — для проверки программы мы полагаемся на метод математической индукции, а для анализа ее производительности используется рекуррентное соотношение.

Как обычно, сам код предполагает проверку правильности вычисления методом индукции:

- Он явно и немедленно отыскивает максимальный элемент массива, размер которого равен 1.
- Для  $N > 1$  код разделяет массив на два, размер каждого из которых меньше  $N$ , исходя из индуктивного предположения, находит максимальные элементы в обеих частях и возвращает большее из этих двух значений, которое должно быть максимальным значением для всего массива.

Более того, рекурсивную структуру программы можно использовать для исследования характеристик ее производительности.

**Лемма 5.1** *Рекурсивная функция, которая разделяет задачу размерности  $N$  на две независимые (непустые) решающие ее части, рекурсивно вызывает себя менее  $N$  раз.*

Если одна часть имеет размерность  $k$ , а другая —  $N-k$ , то общее количество рекурсивных вызовов используемой функции равно

$$T_N = T_k + T_{N-k} + 1, \quad \text{при } N \geq 1; T_1 = 0$$

Решение  $T_N = N - 1$  можно получить непосредственно методом индукции. Если сумма размеров частей меньше  $N$ , доказательство того, что количество вызовов меньше чем  $N - 1$  вытекает из тех же рассуждений по методу индукции. Аналогичными рассуждениями можно подтвердить справедливость данного утверждения и для общего случая (см. упражнение 5.20).

0 1 2 3 4 5 6 7 8 9 10  
T I N Y E X A M P L E

```

Y max(0, 10)
  Y max(0, 5)
    T max(0, 2)
      T max(0, 1)
        T max(0, 0)
          I max(1, 1)
            N max(2, 2)
              Y max(3, 5)
                Y max(3, 4)
                  Y max(3, 3)
                    E max(4, 4)
                      X max(5, 5)
                        P max(6, 10)
                          P max(6, 8)
                            M max(6, 7)
                              A max(6, 6)
                                M max(7, 7)
                                  P max(8, 8)
                                    L max(9, 10)
                                      L max(9, 9)
                                        E max(10, 10)

```

#### РИСУНОК 5.4 РЕКУРСИВНЫЙ ПОДХОД К РЕШЕНИЮ ЗАДАЧИ ОТЫСКАНИЯ МАКСИМУМА

*Эта последовательность вызовов функций иллюстрирует динамику отыскания максимума с помощью рекурсивного алгоритма.*

Программа 5.6 — типичный пример для многих алгоритмов типа "разделяй и властвуй", имеющих совершенно одинаковую рекурсивную структуру, но другие примеры могут отличаться от приведенного в двух аспектах. Во-первых, программа 5.6 выполняет одинаковый объем вычислений для каждого вызова функции, и поэтому ее общее время выполнения линейно связано с количеством вызовов. Как будет показано, другие алгоритмы типа "разделяй и властвуй" могут выполнять различный объем вычислений для различных вызовов функций, и поэтому для определения общего времени выполнения требуется более сложный анализ. Время выполнения таких алгоритмов зависит от конкретного способа деления на части. Во-вторых, программа 5.6 — типичный пример алгоритмов типа "разделяй и властвуй", для которых сумма размеров частей равна общей размерности. Другие алгоритмы типа "разделяй и властвуй" могут разделять задачу на части, сумма размеров которых меньше или больше размерности всей задачи. Эти алгоритмы все же относятся к рекурсивным алгоритмам типа "разделяй и властвуй", поскольку *каждая* часть меньше целого, но анализировать их труднее, нежели программу 5.6. Мы подробно рассмотрим процесс анализа таких типов алгоритмов, как только столкнемся с ними.

Например, алгоритм бинарного поиска, приведенный в разделе 2.6, является алгоритмом типа "разделяй и властвуй", который делит задачу пополам, а затем работает только с одной из этих половин. Рекурсивная реализация бинарного поиска исследуется в главе 12.

На рис. 5.5 показано содержимое внутреннего стека, поддерживаемого средой программирования для реализации вычислений, изображенных на рис. 5.4. Приведенная на рисунке модель является идеализированной, но она позволяет взглянуть на структуру вычисления по методу "разделяй и властвуй" изнутри. Если программа имеет два рекурсивных вызова, фактический внутренний стек содержит одну запись, соответствующую первому вызову функции во время выполнения (эта запись содержит значения аргументов, локальные переменные и адрес возврата), и аналогичную запись, соответствующую второму вызову функции во время ее выполнения. Альтернатива продемонстрированному на рис. 5.5 подходу — помещение в стек сразу двух значений с сохранением всех подстеков, которые должны явно создаваться в стеке. Такая организация проясняет структуру вычислений и закладывает основу

```

0 10
0 5 6 10
0 2 3 5 6 10
0 1 2 2 3 5 6 10
0 0 1 1 2 2 3 5 6 10
1 1 2 2 3 5 6 10
2 2 3 5 6 10
3 5 6 10
3 4 5 5 6 10
3 3 4 4 5 5 6 10
4 4 5 5 6 10
5 5 6 10
6 10
6 8 9 10
6 7 8 8 9 10
6 6 7 7 8 8 9 10
7 7 8 8 9 10
8 8 9 10
9 10
9 9 10 10
10 10

```

### РИСУНОК 5.5 ПРИМЕР ДИНАМИКИ ВНУТРЕННЕГО СТЕКА

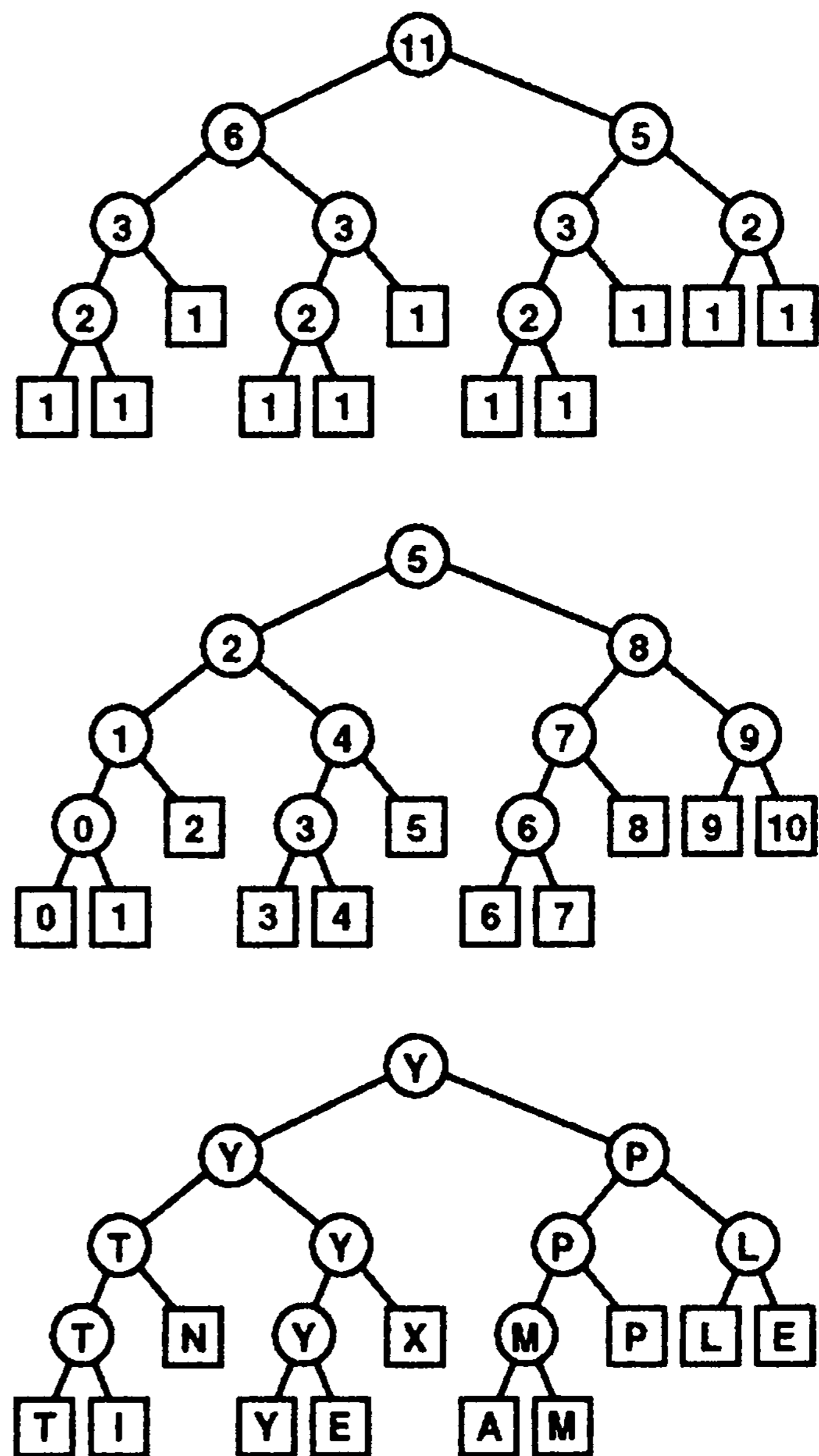
*Эта последовательность — идеализированное представление содержимого внутреннего стека во время вычисления примера из рис. 5.4. Обработка начинается с левого и правого индексов всего подмассива в стеке. Каждая строка представляет результат помещения в стек двух индексов и, если они не равны, проталкивания четырех индексов, которые ограничивают левый и правый подмассивы после деления помещенного подмассива на две части. На практике вместо выполнения упомянутых действий система хранит в стеке адреса возврата и локальные переменные, тем не менее, приведенная модель вполне адекватно описывает вычисление.*

для более общих вычислительных схем, подобных исследуемым в разделах 5.6 и 5.8.

На рис. 5.6 показана структура алгоритма "разделяй и властвуй" для поиска максимального значения. Эта структура является рекурсивной: верхний узел содержит размер входного массива; структура левого подмассива изображена слева, а правого — справа. Формальное определение и описание структур деревьев такого типа можно найти в разделах 5.4 и 5.5. Они облегчают понимание структуры любых программ, в которых используются вложенные вызовы функций — в частности, рекурсивных программ. На рис. 5.6 показано также это же дерево, но с узлами, помеченными возвращаемым значением из соответствующего обращения к функции. Процесс создания явно связанных структур, которые представляют деревья, подобные этому, рассматривается в разделе 5.7.

Ни одно рассмотрение рекурсии не было бы полным без рассмотрения старинной задачи о ханойских башнях. Имеется три стержня и  $N$  дисков, которые помещаются на трех стержнях. Диски различаются размерами и вначале размещаются на одном из стержней от самого большого (диск  $N$ ) внизу до самого маленького (диск 1) вверху. Задача состоит в перемещении дисков на соседнюю позицию (стержень) при соблюдении следующих правил: (i) одновременно можно перемещать только один диск; и (ii) ни один диск не может быть помещен поверх диска меньшего размера. Легенда гласит, что конец света наступит раньше, чем группа монахов справится с задачей размещения 40 золотых дисков на трех алмазных стержнях.

Программа 5.7 предоставляет рекурсивное решение этой задачи. Программа указывает диск, который должен быть сдвинут на каждом шагу, и направление его перемещения (+ означает перемещение на один стержень вправо с переходом к крайнему слева стержню при достижении крайнего справа стержня, а — означает перемещение на один стержень влево с переходом к крайнему справа стержню при достижении крайнего слева стержня). Рекурсия основывается на следующей идее: для перемещения  $N$  дисков вправо на один стержень вначале верхние  $N - 1$  дисков нужно переместить на один стержень влево, затем переместить



**РИСУНОК 5.6 РЕКУРСИВНАЯ СТРУКТУРА АЛГОРИТМА ПОИСКА МАКСИМУМА.**

*Алгоритм "разделяй и властвуй" разделяет представляющий задачу массив, размер которого равен 11, на массивы с размерами 6 и 5, массив с размером 6 — на два массива с размерами 3 и т.д., пока не будет получен массив с размером 1 (верхний). Каждая окружность на этих схемах представляет вызов рекурсивной функции для расположенных непосредственно под ней узлов, связанных с ней линиями (квадраты представляют вызовы, для которых рекурсия завершается). На схеме в центре показано значение индекса в середине файла, который использовался для выполнения деления; на нижней схеме показано возвращаемое значение.*



диск  $N$  на один стержень вправо, после чего еще раз переместить  $N - 1$  дисков на один стержень влево (поверх диска  $N$ ). В правильности работы этого решения можно удостовериться методом индукции. На рис. 5.7 показаны перемещения для  $N = 5$  и рекурсивные вызовы для  $N = 3$ . Структура алгоритма достаточно очевидна; давайте рассмотрим его подробно.

### Программа 5.7 Решение задачи о ханойских башнях

Мы сдвигаем башню из дисков вправо, сдвигая (рекурсивно) все диски, кроме нижнего, влево, затем сдвигаем нижний диск вправо, после чего перемещаем (рекурсивно) башню поверх нижнего диска.

```
void hanoi(int N, int d)
{
    if (N == 0) return;
    hanoi(N-1, -d);
    shift(N, d);
    hanoi(N-1, -d);
}
```

Во-первых, рекурсивная структура этого решения немедленно диктует количество необходимых перемещений.

**Лемма 5.2** *Рекурсивный алгоритм "разделяй и властвуй" решения задачи о ханойских башнях дает решение, приводящее к  $2^N - 1$  перемещениям.*

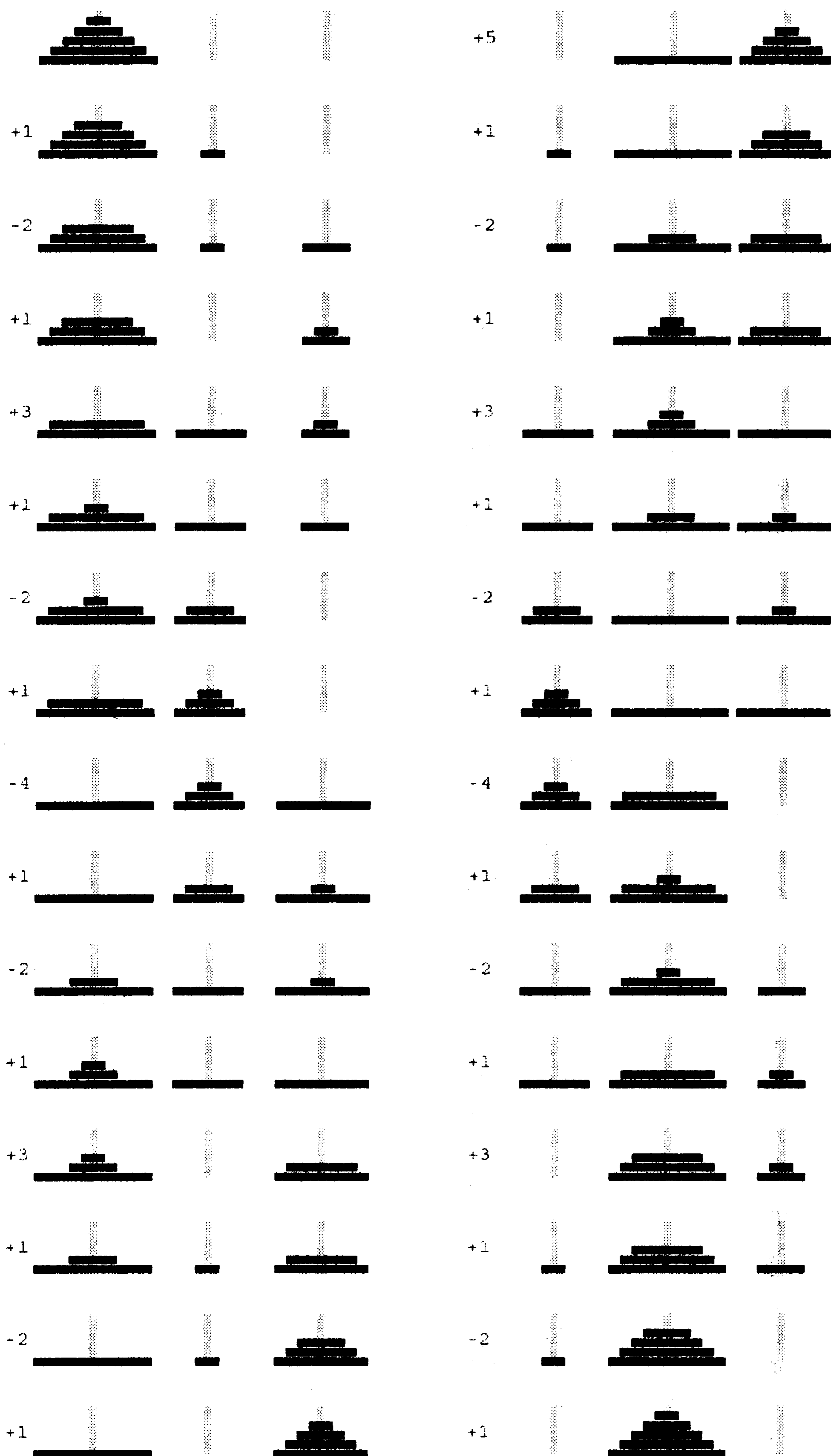
Как обычно, из кода немедленно следует, что количество перемещений удовлетворяет условию рекуррентности. В данном случае количество перемещений дисков, удовлетворяющее условию рекуррентности, определяется формулой, аналогичной формуле 2.5:

$$T_N = 2T_{N-1} + 1, \quad \text{при } N \geq 2, T_1 = 1.$$

Предсказанный результат можно непосредственно проверить методом индукции: мы имеем  $T(1) = 2^1 - 1 = 1$ ; и, если  $T(k) = 2^k - 1$  для  $k < N$ , то  $T(N) = 2(2^{N-1} - 1) + 1 = 2^N - 1$ .

Если монахи перемещают по одному диску в секунду, для завершения работы им потребуется, по меньшей мере, 348 столетий (см. рис. 2.1), разумеется, если они не допускают ошибок. Скорее всего, конец света может наступить даже позже этого срока, поскольку монахи никак не смогли бы воспользоваться программой 5.7 для быстрого выяснения, какой диск нужно перемещать следующим. А теперь давайте проанализируем метод, который ведет к простому (не рекурсивному) решению, упрощающему принятие решения. Хоть и не хочется оставлять бедных монахов в неведении, однако этот метод имеет большое значение для множества важных применяемых на практике алгоритмов.

Чтобы понять решение задачи о ханойских башнях, давайте рассмотрим простую задачу рисования меток на линейке. Каждые  $1/2$  дюйма на линейке отмечаются черточкой, каждые  $1/4$  дюйма отмечаются несколько более короткими черточками,  $1/8$  дюйма — еще более короткими и т.д. Задача состоит в создании программы для рисования этих меток при любом заданном разрешении, при условии, что в нашем распоряжении имеется процедура `mark(x, h)` для рисования меток высотой  $h$  условных единиц в позиции  $x$ .



### РИСУНОК 5.7 ХАНОЙСКИЕ БАШНИ

На этой схеме отображено решение задачи о ханойских башнях для случая с пятью дисками. Мы сдвигаем четыре верхних диска влево на одну позицию (левый столбец), затем перемещаем диск 5 вправо, а затем верхние четыре диска перемещаем влево на одну позицию (правый столбец). Приведенная ниже последовательность вызовов функций образует вычисление для трех дисков. Вычисленная последовательность перемещений — +1 -2 +1 +3 +1 -2 +1 появляется в решении четыре раза (для примера показано первых семь перемещений).

```

hanoi(3, +1)
  hanoi(2, -1)
    hanoi(1, +1)
      hanoi(0, -1)
      shift(1, +1)
      hanoi(0, -1)
    shift(2, -1)
  hanoi(1, +1)
    hanoi(0, -1)
    shift(1, +1)
    hanoi(0, -1)
  shift(3, +1)
hanoi(2, -1)
  hanoi(1, +1)
    hanoi(0, -1)
    shift(1, +1)
    hanoi(0, -1)
  shift(2, -1)
hanoi(1, +1)
  hanoi(0, -1)
  shift(1, +1)
  hanoi(0, -1)

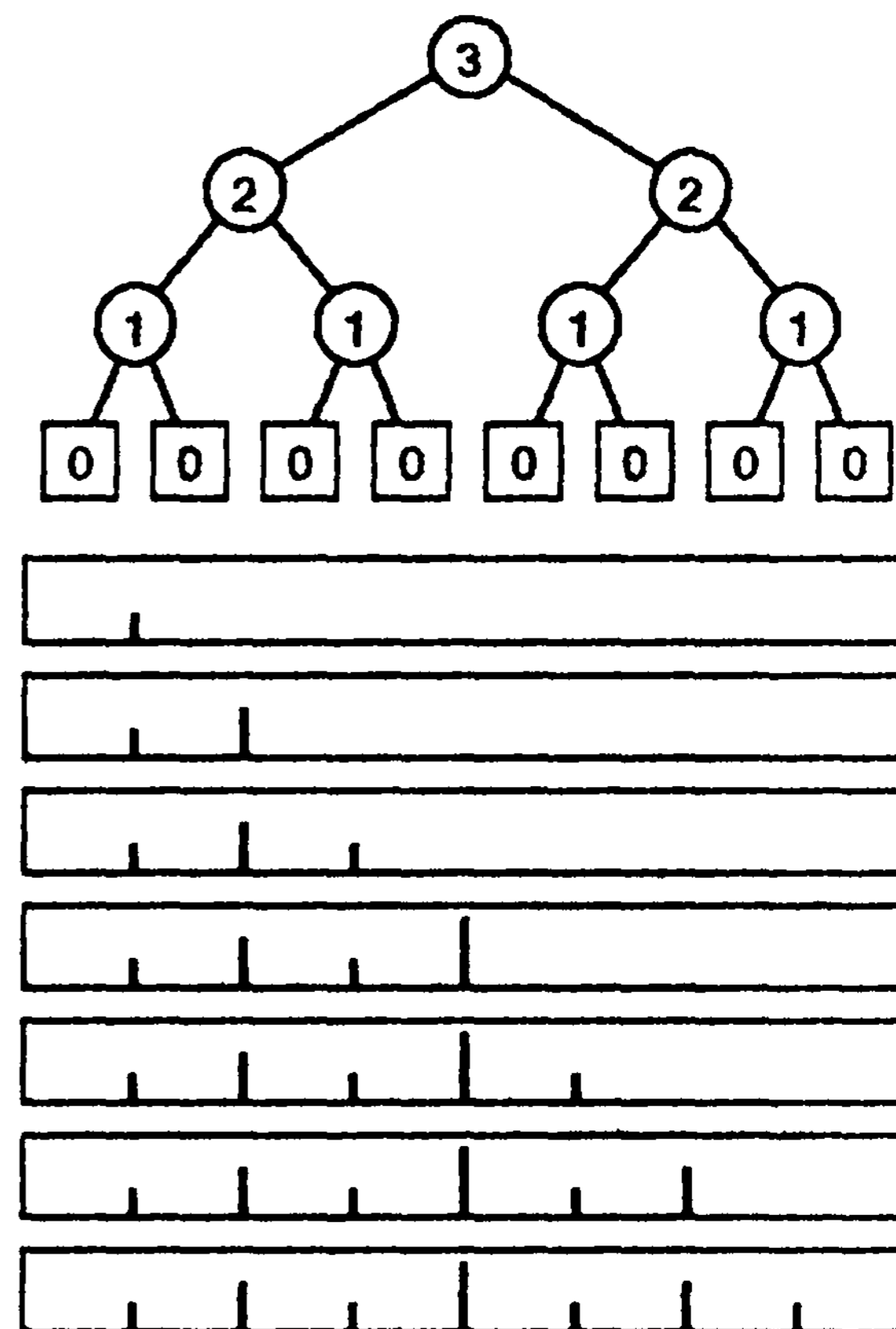
```

Если требуемое разрешение равно  $1/2''$  дюймов, давайте изменим масштаб, чтобы задача состояла в помещении метки в каждой точке в интервале от 0 до  $2''$ , за исключением конечных точек. Таким образом, средняя метка должна иметь высоту  $n$  условных единиц, метки в середине левой и правой половин должны иметь высоту  $n - 1$  условных единиц и т.д. Программа 5.8 — простой алгоритм "разделяй и властвуй" для выполнения этой задачи; работа данного алгоритма применительно к небольшому примеру проиллюстрирована на рис. 5.8. С точки зрения рекурсии в основе метода лежит следующая идея. Для помещения меток на интервале, последний вначале делится на две равные половины. Затем создаются метки (более короткие) в левой половине (рекурсивно), помещается длинная метка в середине и рисуются метки (более короткие) в правой половине (рекурсивно). Если говорить об итерации, рис. 5.8 иллюстрирует, что с помощью этого метода метки создаются по порядку, слева направо — фокус заключается в вычислении длин интервалов. Дерево рекурсии, приведенное на рисунке, помогает понять вычисление: просматривая его сверху вниз, мы видим, что длина меток уменьшается на 1 для каждого вызова рекурсивной функции. Если просматривать дерево в поперечном направлении, мы получаем метки в том порядке, в каком они рисуются, поскольку для каждого данного узла вначале рисуются метки, связанные с вызовом функции слева, затем метка, связанная с данным узлом, а затем метки, связанные с вызовом функции справа.

### Программа 5.8 Применение алгоритма "разделяй и властвуй" для рисования линейки

Для рисования меток на линейке мы рисуем метки в левой половине, затем самую длинную метку в середине, а затем метки в правой половине. Эта программа предназначена для использования со значением  $r - l$  равным степени 2 — свойство, сохраняемое в ее рекурсивных вызовах (см. упражнение 5.27).

```
void rule(int l, int r, int h)
{ int m = (l+r)/2;
  if (h > 0)
  {
    rule(l, m, h-1);
    mark(m, h);
    rule(m, r, h-1);
  }
}
```



```
rule(0, 8, 3)
  rule(0, 4, 2)
    rule(0, 2, 1)
      rule(0, 1, 0)
      mark(1, 1)
      rule(1, 2, 0)
    mark(2, 2)
    rule(2, 4, 1)
      rule(2, 3, 0)
      mark(3, 1)
      rule(3, 4, 0)
    mark(4, 3)
  rule(4, 8, 2)
    rule(4, 6, 1)
      rule(4, 5, 0)
      mark(5, 1)
      rule(5, 6, 0)
    mark(6, 2)
    rule(6, 8, 1)
      rule(6, 7, 0)
      mark(7, 1)
      rule(7, 8, 0)
```

### РИСУНОК 5.8 ВЫЗОВЫ ФУНКЦИЙ, РИСУЮЩИХ ЛИНЕЙКИ

Эта последовательность вызовов функций составляет вычисление для рисования линейки длиной 8, в результате чего наносятся метки 1, 2, 1, 3, 1, 2 и 1.

Сразу видно, что последовательность длин в точности совпадает с последовательностью перемещаемых дисков при решении задачи о ханойских башнях. Действительно, простым доказательством их идентичности является идентичность рекурсивных программ. Таким образом, применив другой подход, наши монахи могли воспользоваться метками на линейке для определения диска, который требуется переместить.

Более того, и решение задачи о ханойских башнях в программе 5.7, и программа рисования линейки в программе 5.8 являются вариантами общей схемы "разделяй и властвуй", представленной программой 5.6. Все три программы решают задачу размера  $2^n$ , разделяя ее на две задачи размера  $2^{n-1}$ . При отыскании максимума время получения решения линейно связано с размером входного массива; при рисовании линейки и при решении задачи о ханойских башнях время линейно связано с размером выходного массива. Обычно считается, что время выполнения задачи о ханойских башнях определяется экспоненциальной зависимостью, поскольку объем задачи измеряется количеством дисков, т.е.  $n$ .

Рисование меток на линейке с помощью рекурсивной программы не представляет особой сложности, но не существует ли более простого способа вычисления длины  $i$ -ой метки для любого данного значения  $i$ ? На рис. 5.9 показан еще один простой вычислительный процесс, дающий ответ на этот вопрос.  $i$ -ый номер, выводимый и программой решения задачи о ханойских башнях, и программой рисования линейки, — ни что иное как количество окончных 0-вых разрядов в двоичном представлении  $i$ . Это утверждение можно доказать методом индукции по соответствию с формулировкой метода "разделяй и властвуй" для процесса вывода таблицы  $n$ -разрядных чисел: достаточно напечатать таблицу  $(n - 1)$ -разрядных чисел, каждому из которых предшествует 0-й разряд, а затем напечатать таблицу  $(n - 1)$ -разрядных чисел, каждому из которых предшествует 1-й разряд (см. упражнение 5.25).

Применительно к задаче о ханойских башнях использование соответствия с  $n$ -разрядными числами — простой алгоритм решения задачи. Можно смещать стопку дисков на один стержень вправо, до завершения повторяя следующие два шага:

- Перемещение маленького диска вправо, если  $n$  нечетно (влево, если оно четно).
- Выполнение единственного разрешенного перемещения, не затрагивающего маленький диск.

То есть, после перемещения маленького диска, остальные два стержня содержат два диска, один из которых меньше другого. Единственное разрешенное перемещение, не затрагивающее маленький диск — перемещение меньшего диска поверх большего. Каждое второе перемещение затрагивает маленький диск по

0	0	0	0	1	
0	0	0	1	0	1
0	0	0	1	1	
0	0	1	0	0	2
0	0	1	0	1	
0	0	1	1	0	1
0	0	1	1	1	
0	1	0	0	0	3
0	1	0	0	1	
0	1	0	1	0	1
0	1	0	1	1	
0	1	1	0	0	2
0	1	1	0	1	
0	1	1	1	0	1
0	1	1	1	1	
1	0	0	0	0	4
1	0	0	0	1	
1	0	0	1	0	1
1	0	0	1	1	
1	0	1	0	0	2
1	0	1	0	1	
1	0	1	1	0	1
1	0	1	1	1	
1	1	0	0	0	3
1	1	0	0	1	
1	1	0	1	0	1
1	1	0	1	1	
1	1	1	0	0	2
1	1	1	0	1	
1	1	1	1	0	1
1	1	1	1	1	

**РИСУНОК 5.9**  
**БИНАРНЫЙ**  
**ПОДСЧЕТ И**  
**ФУНКЦИЯ**  
**РИСОВАНИЯ**  
**ЛИНЕЙКИ**

*Вычисление функции рисования линейки эквивалентно подсчету количества окончных нулей в четных  $N$ -разрядных числах.*

той же причине, что каждое второе число является нечетным, а каждая вторая метка на линейке является самой короткой. Вероятно, наши монахи знали этот секрет, поскольку трудно себе представить, как иначе они могли бы определить нужные перемещения.

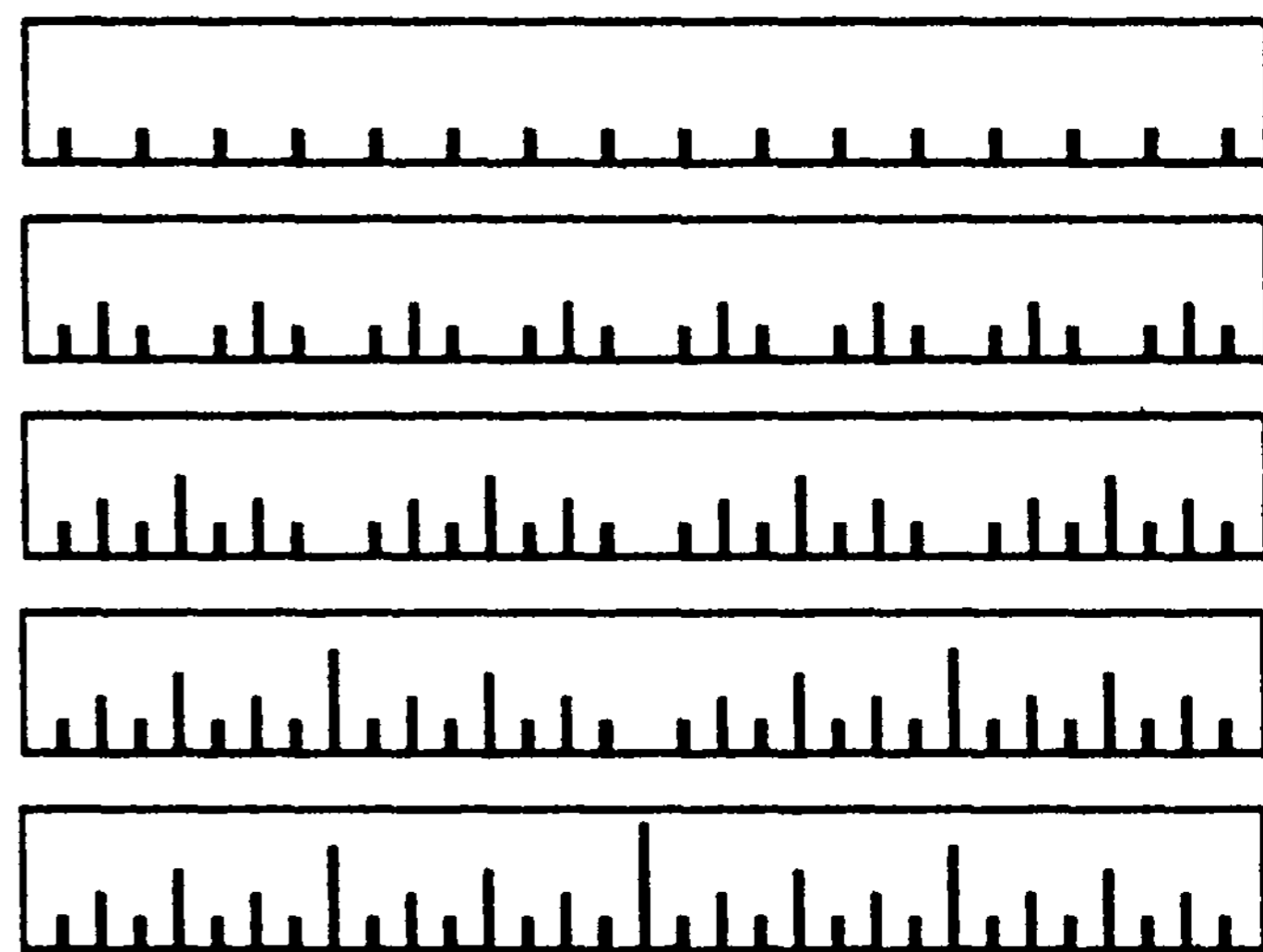
Формальное доказательство методом индукции того, что в решении задачи о ханойских башнях каждое второе перемещение затрагивает маленький диск (все начинается и завершается такими перемещениями), весьма поучительно: Для  $n = 1$  существует только одно перемещение, затрагивающее маленький диск, следовательно, утверждение подтверждается. При  $n > 1$  из предположения, что утверждение справедливо для  $n - 1$ , следует его справедливость и для  $n$ . Это можно подтвердить, прибегнув к следующей рекурсивной конструкции: первое решение для  $n - 1$  начинается с перемещения маленького диска, а второе решение для  $n - 1$  завершается перемещением маленького диска, следовательно, решение для  $n$  начинается и завершается перемещением маленького диска. Мы поместили перемещение, не затрагивающее маленький диск, между двумя перемещениями, которые его затрагивают (перемещением, завершающим первое решение для  $n - 1$ , и перемещением, начинающим второе решение для  $n - 1$ ), следовательно, утверждение, что каждое второе перемещение затрагивает маленький диск, подтверждается.

Программа 5.9 представляет альтернативный способ рисования линейки, на который натолкнуло соответствие с двоичными числами (см. рис. 5.10). Эту версию алгоритма называют *восходящей (bottom-up)* реализацией. Она не является рекурсивной, но определенно навеяна рекурсивным алгоритмом. Это соответствие между алгоритмами типа "разделяй и властвуй" и двоичными представлениями чисел часто способствует углубленному пониманию при анализе и разработке усовершенствованных версий, таких как восходящие подходы. Данную возможность следует учитывать, чтобы понять и, возможно, усовершенствовать каждый из исследуемых алгоритмов типа "разделяй и властвуй".

### Программа 5.9 Не рекурсивная программа для рисования линейки

В противоположность программе 5.8, линейку можно нарисовать, вначале изобразив все метки длиной 1, затем все метки длиной 2 и т.д. Переменная  $t$  представляет длину меток, а переменная  $j$  — количество меток между двумя последовательными метками длиной  $t$ . Внешний цикл `for` увеличивает значение  $t$  при сохранении соотношения  $j=2^{t-1}$ . Внутренний цикл `for` рисует все метки длиной  $t$ .

```
void rule(int l, int r, int h)
{
    for (int t = 1, j = 1;
         t <= h; j += j, t++)
        for (int i = 0; l+j+i <= r;
             i += j+j)
            mark(l+j+i, t);
}
```



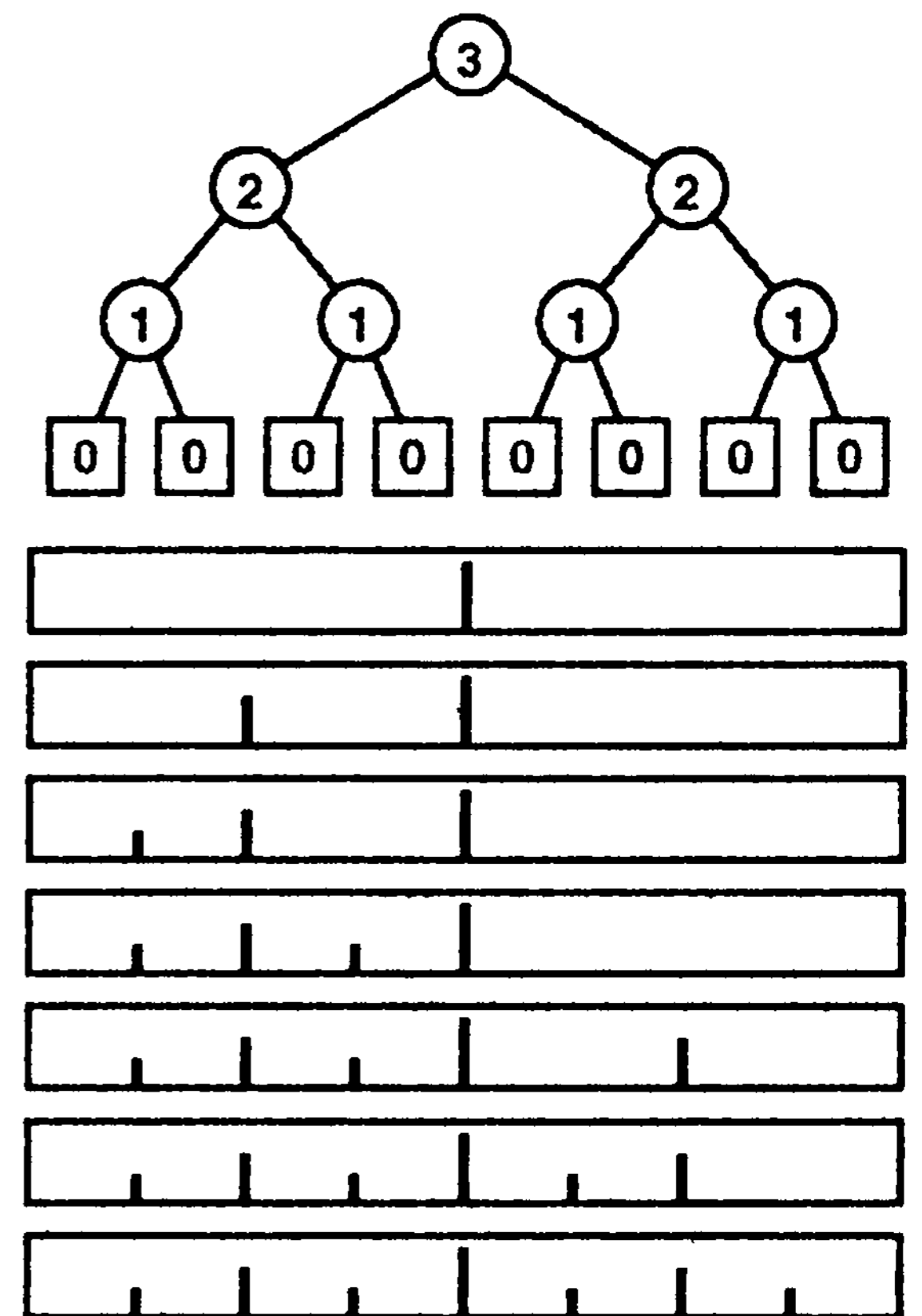
**РИСУНОК 5.10 РИСОВАНИЕ ЛИНЕЙКИ В ВОСХОДЯЩЕМ ПОРЯДКЕ**

Для рисования линейки не рекурсивным методом вначале рисуются все метки длиной 1 и пропускаются позиции, затем рисуются метки длиной 2 и пропускаются остающиеся позиции, затем рисуются метки длиной 3 с пропуском остающихся позиций и т.д.

Восходящий подход предполагает изменение *порядка выполнения* вычислений при рисовании линейки. На рис. 5.11 показан еще один пример, в котором порядок следования трех вызовов функций в рекурсивной реализации изменяется. Этот пример отражает выполнение рекурсивного вычисления первоначально описанным способом: рисование средней метки, затем рисование левой половины, а затем правой. Последовательность рисования меток сложна, но является результатом простой перемены мест двух операторов в программе 5.8. Как будет показано в разделе 5.6, взаимосвязь между рис. 5.8 и 5.11 сродни различию между постфиксными и префиксными арифметическими выражениями.

Рисование меток в порядке, показанном на рис. 5.8, может оказаться более предпочтительным по сравнению с выполнением вычислений в измененном порядке, содержащихся в программе 5.9 и приведенных на рис. 5.11, поскольку можно нарисовать линейку произвольной длины. Достаточно представить себе графическое устройство, которое просто перемещается вдоль непрерывной ленты к следующей метке. Аналогично, при решении задачи о ханойских башнях мы ограничиваемся перемещением дисков в требуемом порядке перемещения. Вообще говоря, многие рекурсивные программы основываются на решениях подзадач, которые должны быть выполнены в конкретном порядке. Для других вычислений (например, см. программу 5.6) порядок решения подзадач роли не играет. Для таких вычислений единственным ограничением служит необходимость решения подзадач перед тем, как можно будет решить главную задачу. Понимание того, когда можно изменять порядок вычисления, не только служит ключом к успешной разработке алгоритма, но и оказывает непосредственное практическое влияние во многих случаях. Например, этот вопрос исключительно важен при исследовании возможности реализации алгоритмов в параллельных процессорах.

Восходящий подход соответствует общему методу разработки алгоритмов, при котором задача решается путем решения вначале элементарных подзадач с последующим объединением этих решений для получения решения несколько больших подзадач и т.д., пока вся задача не будет решена. Подобный подход можно было бы назвать подходом *"объединяй и властвуй"*.



```
rule(0, 8, 3)
  mark(4, 3)
  rule(0, 4, 2)
    mark(2, 2)
    rule(0, 2, 1)
      mark(1, 1)
      rule(0, 1, 0)
      rule(1, 2, 0)
    rule(2, 4, 1)
      mark(3, 1)
      rule(2, 3, 0)
      rule(3, 4, 0)
  rule(4, 8, 2)
    mark(6, 2)
    rule(4, 6, 1)
      mark(5, 1)
      rule(4, 5, 0)
      rule(5, 6, 0)
    rule(6, 8, 1)
      mark(7, 1)
      rule(6, 7, 0)
      rule(7, 8, 0)
```

**РИСУНОК 5.11 ВЫЗОВЫ ФУНКЦИЙ ДЛЯ РИСОВАНИЯ ЛИНЕЙКИ (ВЕРСИЯ С ИСПОЛЬЗОВАНИЕМ ПРЯМОГО ОБХОДА)**

*Эта последовательность отображает результат рисования меток перед рекурсивными вызовами, а не между ними.*

Лишь небольшой шаг отделяет рисование линеек от рисования двумерных узоров, похожих на показанный на рис. 5.12. Этот рисунок иллюстрирует, как простое рекурсивное описание может приводить к сложным вычислениям (см. упражнение 5.30).

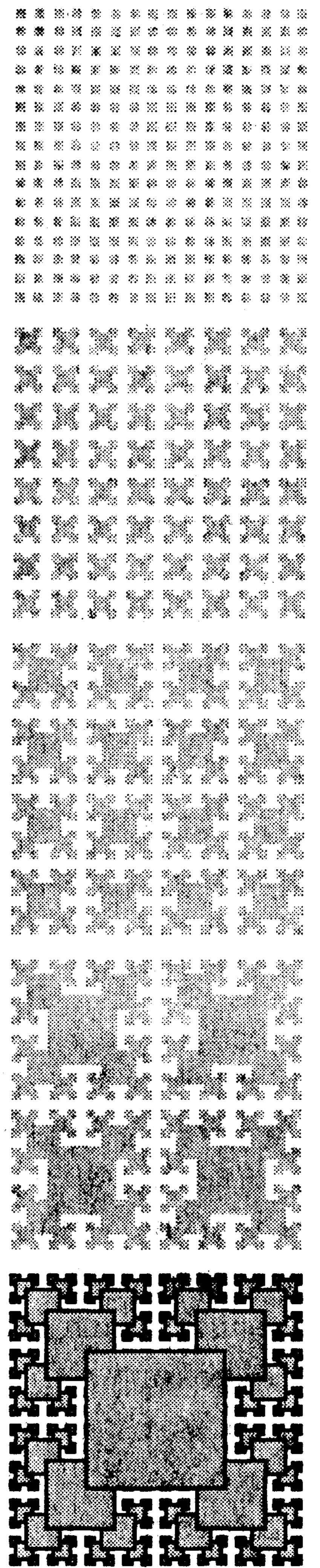
Рекурсивно определенные геометрические узоры, наподобие показанного на рис. 5.12, иногда называют *фракталами*. При использовании более сложных примитивов рисования и более сложных рекурсивных функций (особенно, включая рекурсивно определенные функции, работающие как с действительными значениями, так и в комплексной плоскости), можно разрабатывать поразительно разнообразные и сложные узоры. На рис. 5.13 приведен еще один пример — *звезда Коха*, которая определяется рекурсивно следующим образом: звезда Коха 0 порядка — простой выступ, показанный на рис. 4.3, а звезда Коха  $n$  порядка — это звезда Коха порядка  $n - 1$ , в которой каждый линейный сегмент заменен соответствующим образом отмасштабированной звездой 0 порядка.

Подобно решениям задач рисования линейки и ханойских башен, эти алгоритмы линейны по отношению к количеству шагов, но это количество связано экспоненциальной зависимостью с максимальной глубиной рекурсии (см. упражнения 5.29 и 5.33). Кроме того, они могут быть непосредственно связаны с подсчетом в соответствующей системе счисления (см. упражнение 5.34).

Задача о ханойских башнях, рисование линейки и фракталы весьма занимательны, а связь с двоичными числами поражает, однако все эти вопросы интересуют нас прежде всего потому, что облегчают понимание одного из основных методов разработки алгоритмов — деление задачи пополам и независимое решение одной или обеих половин задачи; возможно, это — наиболее важная из подобного рода технологий, рассматриваемых в книге. В табл. 5.1 подробно описаны бинарный поиск и сортировка слиянием, которые не только являются важными и широко используемыми на практике алгоритмами, но и служат типичными примерами разработки алгоритмов типа "разделяй и властвуй".

### Таблица 5.1 Основные алгоритмы типа "разделяй и властвуй"

Бинарный поиск (см. главы 2 и 12) и сортировка слиянием (см. главу 8) — прототипы алгоритмов типа "разделяй и властвуй", которые обеспечивают гарантированную оптимальную производительность, соответственно, поиска и сортировки. Рекуррентное соотношение демонстрирует сущность вычислений методом "разделяй и властвуй" для



**РИСУНОК 5.12 ДВУМЕРНАЯ ФРАКТАЛЬНАЯ ЗВЕЗДА**

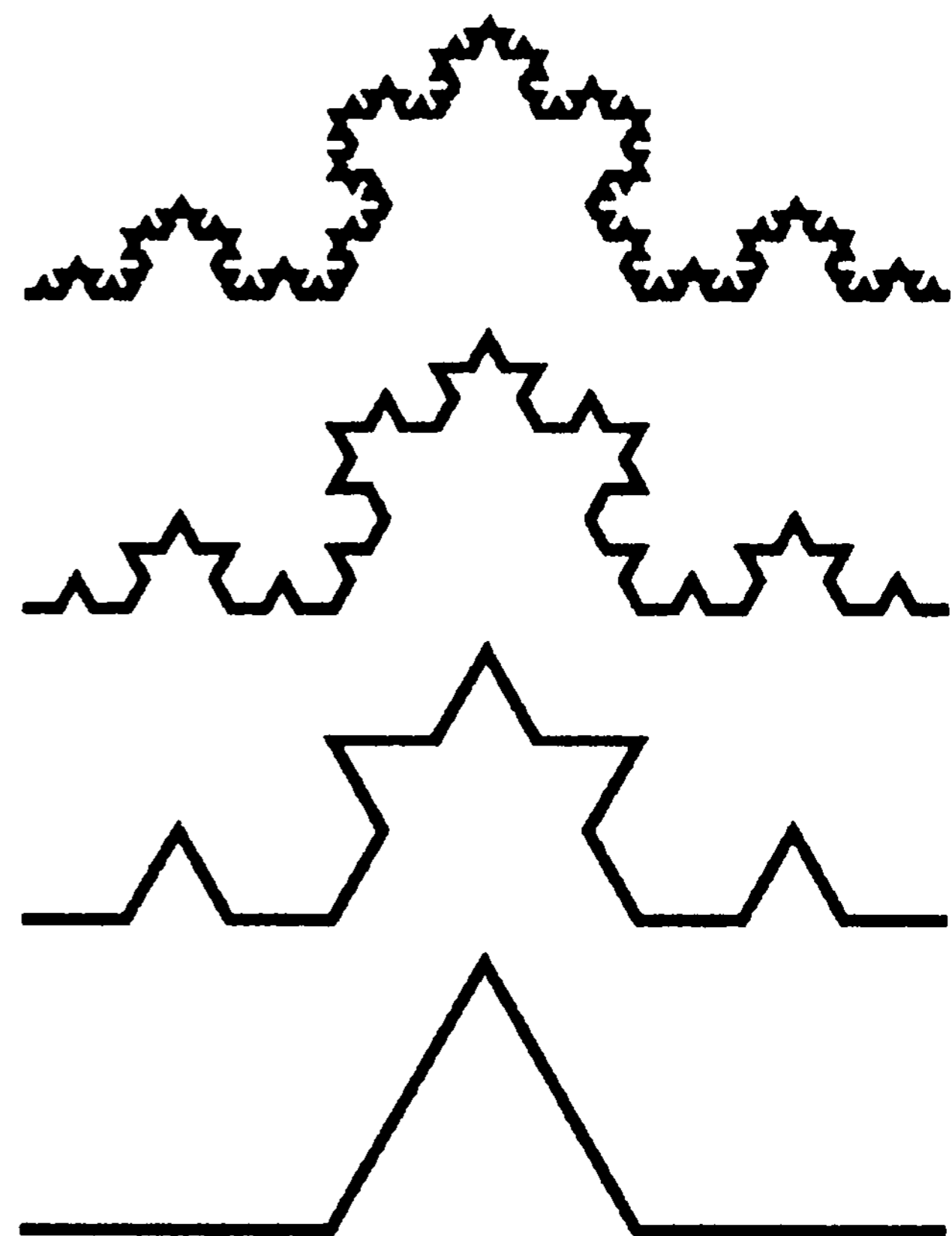
*Этот фрактал — двумерная версия рис. 5.10. Очерченные квадраты на нижнем рисунке подчеркивают рекурсивную структуру вычисления.*

каждого алгоритма. (Получение решений, приведенных в правом столбце, описывается в разделах 2.5 и 2.6.) При бинарном поиске задача делится пополам, выполняется одно сравнение, а затем рекурсивный вызов для одной из половин. При сортировке слиянием задача делится пополам, затем выполняется рекурсивная обработка обеих половин, после чего программа выполняет  $N$  сравнений. В книге будет рассматриваться множество других алгоритмов, разработанных с применением этих рекурсивных схем.

	рекуррентное соотношение	приближенное решение
<b>БИНАРНЫЙ ПОИСК</b>		
количество сравнений	$C_N = C_{N/2} + 1$	$\lg N$
<b>СОРТИРОВКА СЛИЯНИЕМ</b>		
количество рекурсивных вызовов	$A_N = 2A_{N/2} + 1$	$N$
количество сравнений	$C_N = 2C_{N/2} + 1$	$N \lg N$

Быстрая сортировка (см. главу 7) и поиск в бинарном дереве (см. главу 12) представляют важную вариацию базового подхода "разделяй и властвуй", в которой задача разделяется на подзадачи размеров  $k - 1$  и  $N - k$  при некотором исходном значении  $k$ . При произвольном вводе эти алгоритмы делят задачи на подзадачи, размеры которых *в среднем* вдвое меньше исходного (что имеет место при сортировке слиянием или бинарном поиске). Влияние упомянутого различия анализируется при рассмотрении этих алгоритмов.

Рассмотрения заслуживают также следующие вариации основной темы: разделение на части различных размеров, разделение более чем на две части, разделение на перекрывающиеся части и выполнение различного объема вычислений в не-рекурсивной части алгоритма. В общем случае алгоритмы типа "разделяй и властвуй" требуют выполнения вычислений для разделения входного массива на части, либо для объединения результатов обработки двух независимо решенных частей исходной задачи, либо для упрощения задачи после того, как половина входного массива обработана. То есть, код может присутствовать перед, после и между двумя рекурсивными вызовами. Естественно, подобные вариации приводят к созданию более сложных алгоритмов, нежели бинарный поиск или сортировка слиянием, и эти алгоритмы труднее анализировать. В книге приводятся многочисленные примеры; к более сложным приложениям и способам анализа мы вернемся в части 8.



```

/kochR
{
  2 copy ge {dup 0 rlineto }
  {
    3 div
    2 copy kochR
    60 rotate
    2 copy kochR
    -120 rotate
    2 copy kochR
    60 rotate
    2 copy kochR
  } ifelse
  pop pop
} def
0 0 moveto
27 81 kochR
0 27 moveto
9 81 kochR
0 54 moveto
3 81 kochR
0 81 moveto
1 81 kochR
stroke

```

**РИСУНОК 5.13 РЕКУРСИВНАЯ POSTSCRIPT-ПРОГРАММА ДЛЯ РИСОВАНИЯ ФРАКТАЛА КОХА**

*Это изменение программы PostScript, приведенной на рис. 4.3, преобразует вывод в фрактал (см. текст).*



## Упражнения

- 5.16** Создайте рекурсивную программу, которая находит максимальный элемент в массиве, выполняя сравнение первого массива с максимальным элементом остальной части массива (вычисленным рекурсивно).
- 5.17** Создайте рекурсивную программу, которая находит максимальный элемент в связном списке.
- 5.18** Модифицируйте программу "разделяй и властвуй" для отыскания максимального элемента в массиве (программа 5.6), чтобы она делила массив размера  $N$  на две части, одна из которых имеет размер  $k=2^{\lceil \lg N \rceil - 1}$ , а вторая —  $N - k$  (чтобы размер хотя бы одной части был степенью 2).
- 5.19** Нарисуйте дерево, которое соответствует рекурсивным вызовам, выполняемым программой из упражнения 5.18 при размере массива равном 11.
- **5.20** Методом индукции докажите, что количество вызовов функции, выполняемых любым алгоритмом типа "разделяй и властвуй", который делит задачу на части, в сумме составляющие задачу в целом, а затем решает части рекурсивно, линейно связано с размером задачи.
  - **5.21** Докажите, что рекурсивное решение задачи о ханойских башнях (программа 5.7) является оптимальным. То есть, покажите, что любое решение *требует*, по меньшей мере,  $2^N - 1$  перемещений.
  - ▷ **5.22** Создайте рекурсивную программу, которая вычисляет длину  $i$ -ой метки на линейке с  $2^n - 1$  метками.
  - **5.23** Исследуйте таблицы  $n$ -разрядных чисел, подобную приведенной на рис. 5.9, на предмет определения свойства  $i$ -го числа, определяющего направление  $i$ -го перемещения (указанного знаковым битом на рис. 5.7) при решении задачи о ханойских башнях.
- 5.24** Создайте программу, которая обеспечивает решение задачи о ханойских башнях путем заполнения массива, содержащего все перемещения, как сделано в программе 5.9.
- **5.25** Создайте рекурсивную программу, которая заполняет массив размера  $n$ -на- $2^n$  нулями и единицами таким образом, чтобы массив представлял все  $n$ -разрядные числа, как показано на рис. 5.9.
  - 5.26** Отобразите результаты использования рекурсивной программы рисования линейки (программы 5.8) для следующих значений аргументов: **rule(0, 11, 4)**, **rule(4, 20, 4)** и **rule(7, 30, 5)**.
  - 5.27** Докажите следующее свойство программы рисования линейки (программы 5.8): если разность между ее первыми двумя аргументами является степенью 2, то оба ее рекурсивных вызова также обладают этим свойством.
  - **5.28** Создайте функцию, которая эффективно вычисляет количество конечных нулей в двоичном представлении целого числа.
  - **5.29** Сколько квадратов изображено на рис. 5.12 (включая и те, которые скрыты большими квадратами)?
  - **5.30** Создайте рекурсивную программу на C++, результатом которой будет PostScript-программа в форме списка вызовов функций **x y r box**, обеспечивающая

отображение нижнего рисунка рис. 5.12; функция `box` рисует квадрат размера  $g$ -на- $g$  в точке с координатами  $(x,y)$ . Реализуйте функцию `box` в виде команд PostScript (см. раздел 4.3).

**5.31** Создайте восходящую нерекурсивную программу (аналогичную программе 5.9), которая обеспечивает отображение верхнего рисунка рис. 5.12, как описано в упражнении 5.30.

- **5.32** Создайте программу PostScript, обеспечивающую рисование нижней части рис. 5.12.

▷ **5.33** Сколько линейных сегментов содержит звезда Коха  $n$ -го порядка?

- **5.34** Рисование звезды Коха  $n$ -го порядка сводится к выполнению последовательности команд типа "повернуть на  $\alpha$  градусов, затем нарисовать сегмент линии длиной  $1/3$ ." Установите соответствие с системами счисления, позволяющее рисовать звезду путем увеличения значения счетчика и последующего вычисления на базе этого значения угла  $\alpha$ .

- **5.35** Модифицируйте программу рисования звезды Коха, приведенную на рис. 5.13, для создания другого фрактала, основывающегося на фигуре, состоящей из 5 линий нулевого порядка, вычерчиваемых путем смещения на одну условную единицу в восточном, северном, восточном, южном и восточном направлениях (см. рис. 4.3).

**5.36** Создайте рекурсивную функцию типа "разделяй и властвуй" для рисования аппроксимации сегмента линии в пространстве целочисленных координат для заданных конечных точек. Примите, что координаты изменяются в интервале от 0 до  $M$ . *Совет:* вначале вычертите точку вблизи середины сегмента.

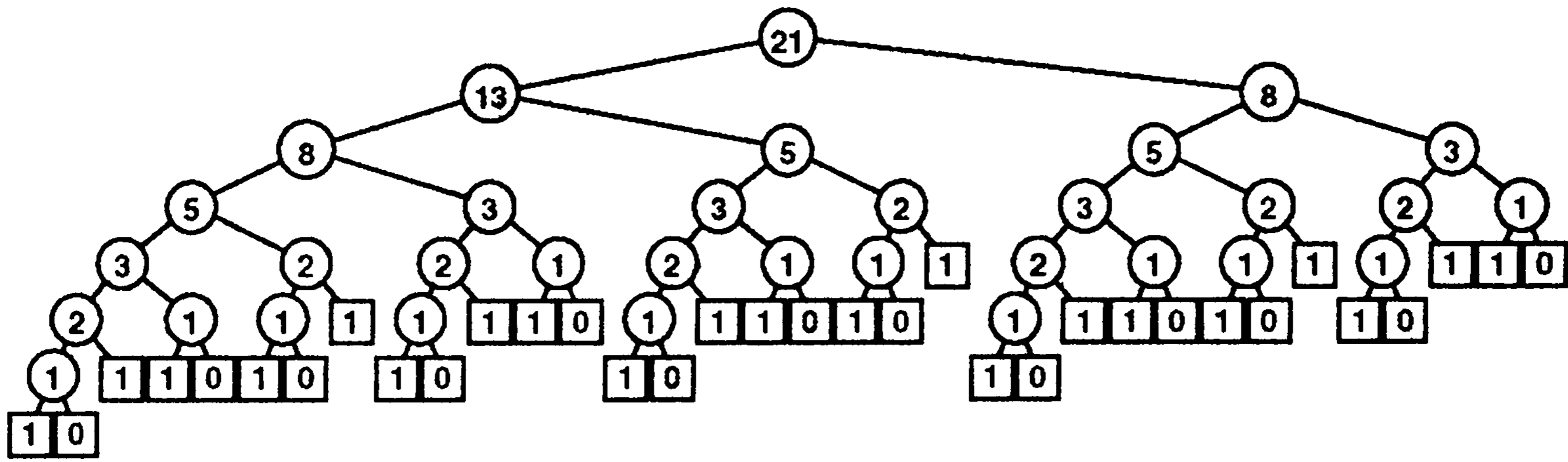
## 5.3 Динамическое программирование

Главная особенность алгоритмов типа "разделяй и властвуй", рассмотренных нами в разделе 5.2, — разделение ими задачи на независимые подзадачи. Когда подзадачи не являются независимыми, ситуация усложняется, в первую очередь потому, что непосредственная рекурсивная реализация даже простейших алгоритмов этого типа может требовать неприемлемо больших затрат времени. В этом разделе рассматривается систематический подход для избежания этой опасности в некоторых случаях.

Например, программа 5.10 — непосредственная рекурсивная реализация рекуррентного соотношения, определяющего числа Фибоначчи (см. раздел 3.3). *Не используйте эту программу* — она весьма неэффективна. Действительно, количество рекурсивных вызовов для вычисления  $F_N$  равно  $F_{N+1}$ . Но  $F_N$  приближенно равно  $\phi^N$ , где  $\phi \approx 1,618$  — золотая пропорция. Как это ни удивительно, но для программы 5.10 время этого элементарного вычисления определяется экспоненциальной зависимостью. Рисунок 5.14, на котором приведены рекурсивные вызовы для небольшого примера, весьма наглядно демонстрирует требуемый объем повторных вычислений.

И напротив, можно легко вычислить первые  $N$  чисел Фибоначчи за время, пропорциональное значению  $N$ , используя следующий массив:

```
F[0] = 0; F[1] = 1;
For (i = 2; i <= N; i++)
    F[i] = F[i-1] + F[i-2];
```



### РИСУНОК 5.14 СТРУКТУРА РЕКУРСИВНОГО АЛГОРИТМА ДЛЯ ВЫЧИСЛЕНИЯ ЧИСЕЛ ФИБОНАЧЧИ

Схема рекурсивных вызовов для вычисления  $F_8$  по стандартному рекурсивному алгоритму иллюстрирует, как рекурсия с перекрывающимися подзадачами может приводить к экспоненциальному возрастанию затрат. В данном случае второй рекурсивный вызов игнорирует вычисление, выполненное во время первого вызова, что приводит к значительным повторным вычислениям, поскольку эффект нарастает в геометрической прогрессии с увеличением количества рекурсивных вызовов. Рекурсивные вызовы для вычисления  $F_6 = 8$  (отраженные в правом поддереве корня и в левом поддереве левого поддерева корня) показаны ниже.

#### Программа 5.10 Числа фибоначчи (рекурсивная реализация)

Эта программа, хотя она выглядит компактно и изящно, неприменима на практике, поскольку время вычисления  $F_N$  экспоненциально зависит от  $N$ . Время вычисления  $F_{N+1}$  в  $\phi \approx 1,6$  раз больше времени вычисления  $F_N$ . Например, поскольку  $\phi^9 > 60$ , если для вычисления  $F_N$  компьютеру требуется около секунды, то для вычисления  $F_{N+9}$  потребуется более минуты, а для вычисления  $F_{N+18}$  — более часа.

```
int F(int i)
{
    if (i < 1) return 0;
    if (i == 1) return 1;
    return F(i-1) + F(i-2);
}
```

Числа возрастают экспоненциально, поэтому размер массива невелик — например,  $F_{45} = 1836311903$  — наибольшее число Фибоначчи, которое может быть представлено 32-разрядным целым, поэтому достаточно использовать массив с 46 элементами.

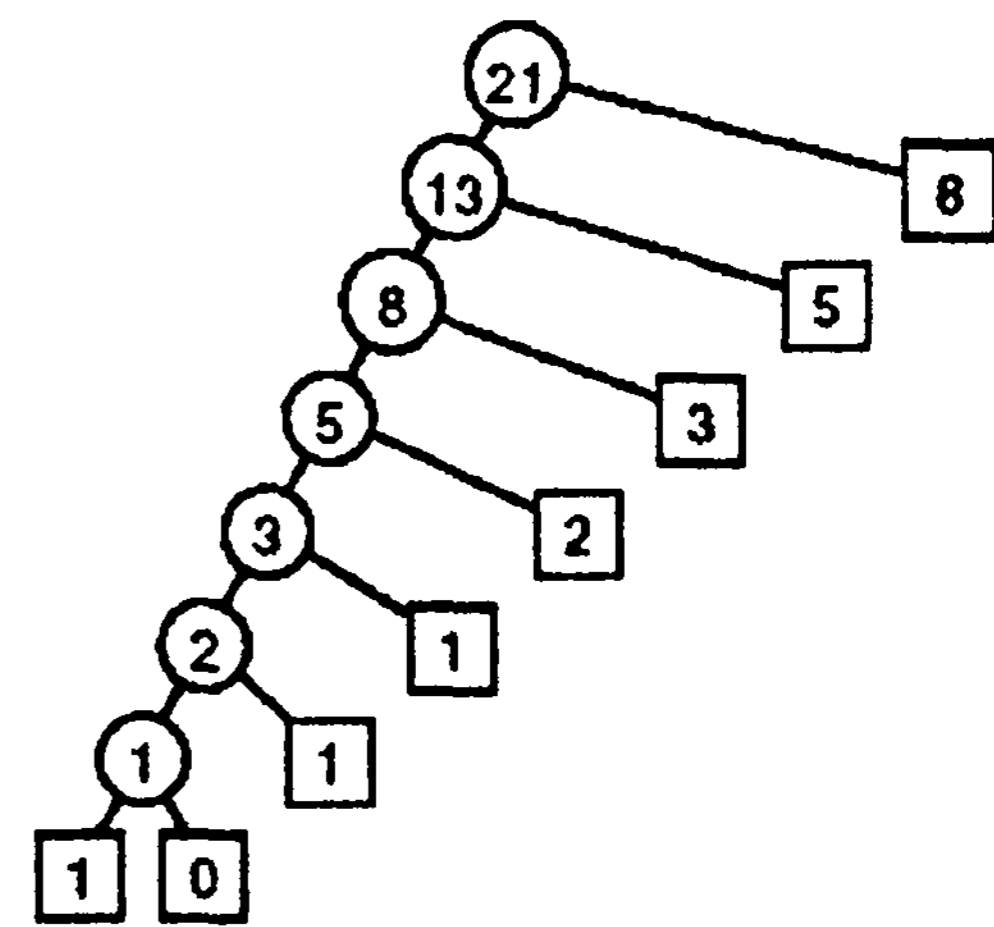
Этот подход предоставляет непосредственный способ получения численных решений для любых рекуррентных соотношений. В случае с числами Фибоначчи можно даже обойтись без массива и ограничиться только первыми двумя значениями (см. упражнение 5.37); для многих других часто встречающихся рекуррентных соотношений (см., например, упражнение 5.40) необходимо поддерживать массив, хранящий все известные значения.

Рекуррентное соотношение — это рекурсивная функция с целочисленными значениями. Рассуждения, приведенные в предыдущем абзаце, подсказывают, что любую такую функцию можно приближенно вычислить, вычисляя все значения функции, начиная с наименьшего, используя на каждом шаге ранее вычисленные значения для подсчета текущего значения. Эта технология называется *восходящим динамическим про-*

```
8 F(6)
  5 F(5)
    3 F(4)
      2 F(3)
        1 F(2)
          1 F(1)
            0 F(0)
          1 F(1)
            1 F(2)
              1 F(1)
                0 F(0)
            2 F(3)
              1 F(2)
                1 F(1)
                  0 F(0)
                1 F(1)
                  3 F(4)
                    2 F(3)
                      1 F(2)
                        1 F(1)
                          0 F(0)
                        1 F(1)
                          1 F(2)
                            1 F(1)
                              0 F(0)
```

граммированием (*bottom-up dynamic programming*). Она применима к любому рекурсивному вычислению *при условии*, что мы можем себе позволить хранить все ранее вычисленные значения. Такая технология разработки алгоритмов успешно используется для решения широкого круга задач. Поэтому следует уделить внимание простой технологии, которая может изменить зависимость времени выполнения алгоритма от значения аргументов с экспоненциальной на линейную!

*Нисходящее динамическое программирование* (*top-down dynamic programming*) — еще более простая технология, которая позволяет автоматически выполнять рекурсивные функции при том же (или меньшем) количестве итераций, что и восходящее динамическое программирование. При этом рекурсивная программа используется для сохранения каждого вычисленного ею (в качестве заключительного действия) значения и для проверки сохраненных значений во избежание повторного вычисления любого из них (в качестве первого действия). Программа 5.11 — механически измененная программа 5.10, в которой за счет применения нисходящего динамического программирования достигается снижение времени выполнения — его зависимость от размера входного массива становится линейной. Рисунок 5.15 служит иллюстрацией радикального уменьшения количества рекурсивных вызовов, достигнутого посредством этого простого автоматического изменения. Иногда нисходящее динамическое программирование называют также *мемуаризацией* (*memoization*).



**РИСУНОК 5.15 ПРИМЕНЕНИЕ НИСХОДЯЩЕГО ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ ДЛЯ ВЫЧИСЛЕНИЯ ЧИСЕЛ ФИБОНАЧЧИ**

*Из этой схемы рекурсивных вызовов, использованных для вычисления  $F_8$  методом нисходящего динамического программирования, видно, как сохранение вычисленных значений снижает нарастание затрат с экспоненциального (см. рис. 5.14) до линейного.*

### Программа 5.11 Числа Фибоначчи (динамическое программирование)

Сохранение вычисляемых значений в статическом массиве (элементы которого в C++ инициализируются 0) явным образом исключает любые повторные вычисления. Эта программа вычисляет  $F_N$  за время, пропорциональное  $N$ , что разительно отличается от времени  $O(\phi^N)$ , которое требуется для вычисления в программе 5.10.

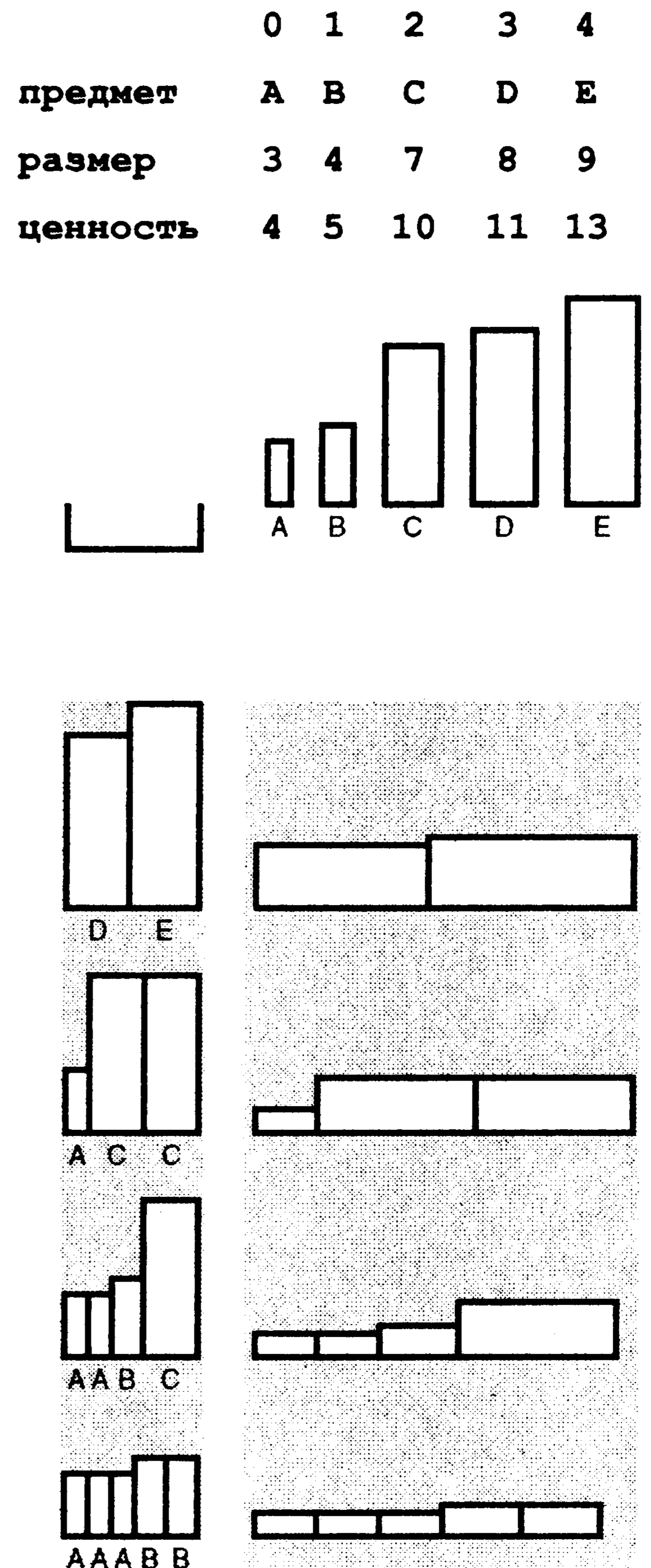
```
int F(int i)
{ static int knownF[maxN];
  if (knownF[i] != 0) return knownF[i];
  int t = i;
  if (i < 0) return 0;
  if (i > 1) t = F(i-1) + F(i-2);
  return knownF[i] = t;
}
```

В качестве более сложного примера давайте рассмотрим *задачу о ранце*: вор, граблящий сейф, находит в нем  $N$  видов предметов различных размеров и ценности, но имеет только небольшой ранец емкостью  $M$ , в котором может унести награбленное. Задача заключается в том, чтобы определить комбинацию предметов, которые вор должен уложить в ранец, чтобы общая стоимость похищенного оказалась наиболь-

шей. Например, при наличии типов предметов, представленных на рис. 5.16, вор, располагающий ранцем, размер которого равен 17, может взять только пять (но не шесть) предметов А общей стоимостью равной 20 или предметы D и E суммарной стоимостью равной 24 или предметы в одной из множества других комбинаций. Наша цель — найти эффективный алгоритм для определения максимума среди всех возможностей при любом заданном наборе предметов и вместимости ранца.

Решения задачи о ранце важны во многих приложениях. Например, транспортной компании может потребоваться определение наилучшего способа загрузки грузовика или транспортного самолета. В подобных приложениях могут встречаться и другие варианты этой задачи: например, количество элементов каждого вида может быть ограничено или могут быть доступны два грузовика. Многие из таких вариантов могут решаться путем применения того же подхода, который мы собираемся исследовать для решения только что сформулированной базовой задачи; другие варианты оказываются гораздо более сложными. Можно четко разграничить решаемые и нерешаемые задачи этого типа, что исследуется в части 8.

В рекурсивном решении задачи о ранце при каждом выборе предмета мы предполагаем, что можем (рекурсивно) определить оптимальный способ заполнения остающегося пространства в ранце. Для ранца размера `cap`, для каждого доступного типа элемента `i` определяется общая стоимость элементов, которые можно было бы унести, укладывая `i`-ый элемент в ранец при оптимальной упаковке остальных элементов. Этот оптимальный способ упаковки — просто способ упаковки, определенный (или который будет определен) для меньшего ранца размера `cap-items[i].size`. В этом решении используется следующий принцип: оптимальные принятые решения в дальнейшем не требуют пересмотра. Как только установлено, как оптимально упаковать ранцы меньших размеров, эти задачи не требуют повторного исследования независимо от следующих элементов.



**РИСУНОК 5.16 ПРИМЕР ЗАДАЧИ О РАНЦЕ**

*Входными данными примера задачи о ранце (вверху) являются емкость ранца и набор предметов различных размеров (которые представлены значениями на горизонтальной оси) и стоимости (значения на вертикальной оси). На этом рисунке показаны четыре различных способа заполнения ранца, размер которого равен 17; два из этих способов ведут к получению максимальной суммарной стоимости, равной 24.*

Программа 5.12 — прямое рекурсивное решение, основывающееся на приведенных рассуждениях. Эта программа также неприменима для решения встречающихся на практике задач, поскольку из-за большого объема повторных вычислений (см. рис. 5.17) время решения связано с количеством элементов экспоненциально. Но для решения проблемы можно автоматически задействовать нисходящее динамическое программирование, как показано в программе 5.13. Как и ранее, эта методика исключает все повторные вычисления (см. рис. 5.18).

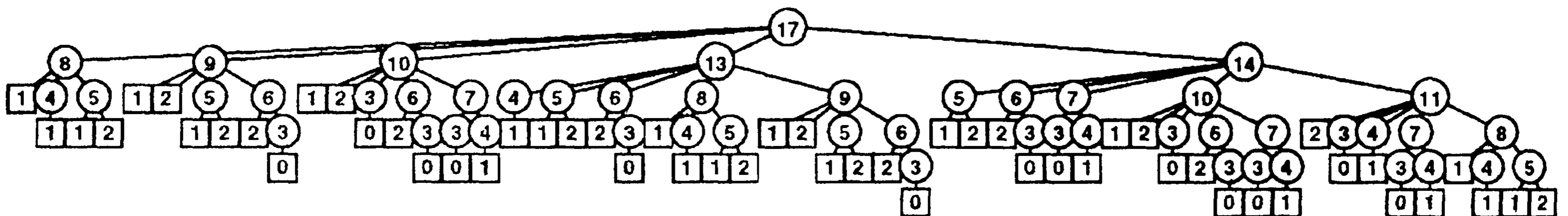
### Программа 5.12 Задача о ранце (рекурсивная реализация)

Аналогично тому, как мы предостерегали в отношении рекурсивного решения задачи вычисления чисел Фибоначчи, *не следует использовать* эту программу, поскольку для ее выполнения потребуется время, экспоненциально связанное с количеством элементов, и поэтому, возможно, даже небольшой задачи решение получить не удастся. Тем не менее, программа представляет компактное решение, которое легко можно усовершенствовать (см. программу 5.13). В ней предполагается, что элементы являются структурами с размером и стоимостью, которые определены следующим образом

```
typedef struct { int size; int val; } Item;
```

кроме того, в нашем распоряжении имеется массив  $N$  элементов типа **Item**. Для каждого возможного элемента мы вычисляем (рекурсивно) максимальную стоимость, которую можно было бы получить, включив этот элемент в выборку, а затем выбираем максимальную из всех стоимостей.

```
int knap(int cap)
{ int i, space, max, t;
  for (i = 0, max = 0; i < N; i++)
    if ((space = cap-items[i].size) >= 0)
      if ((t = knap(space) + items[i].val) > max)
        max = t;
  return max;
}
```



**РИСУНОК 5.17 РЕКУРСИВНАЯ СТРУКТУРА АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ О РАНЦЕ.**

*Это дерево представляет структуру рекурсивных вызовов простого рекурсивного алгоритма решения задачи о ранце, реализованного в программе 5.12. Число в каждом узле представляет остающееся свободным пространство ранца. Недостатком алгоритма является то же экспоненциальное снижение производительности вследствие большого объема повторных вычислений, требуемых для решения перекрывающихся подзадач, которое рассматривалось при вычислении чисел Фибоначчи (см. рис. 5.14).*

### Программа 5.13 Задача о ранце (динамическое программирование)

Эта программа, которая представляет собой механически измененную программу 5.12, снижает с экспоненциальной до линейной зависимость времени выполнения от количества элементов. Мы просто сохраняем любые вычисленные значения функции, а затем вместо выполнения рекурсивных вызовов получаем любые сохраненные значения, когда они требуются (используя зарезервированное значение для представления неизвестных значений). Индекс элемента сохраняется, поэтому при желании всегда можно восстановить содержимое ранца после вычисления: `itemKnown[M]` находится в ранце, остальное содержимое совпадает с оптимальной упаковкой ранца размера `M-itemKnown[M].size`, следовательно, `itemKnown[M-itemKnown[M].size]` находится в ранце и т.д.

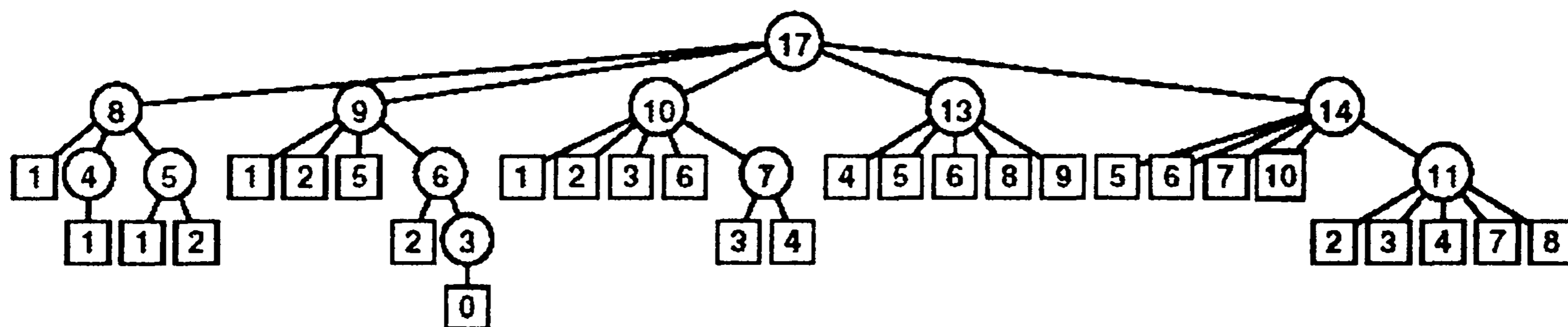
```
int knap(int M)
{ int i, space, max, maxi = 0, t;
  if (maxKnown[M] != unknown) return maxKnown[M];
  for (i = 0, max = 0; i < N; i++)
    if ((space = M-items[i].size) >= 0)
      if ((t = knap(space) + items[i].val) > .max)
        { max = t; maxi = i; }
  maxKnown[M] = max; itemKnown[M] = items[maxi];
  return max;
}
```

Динамическое программирование принципиально исключает все повторные вычисления в *любой* рекурсивной программе, при условии, что мы можем себе позволить сохранять значения функции для аргументов, которые меньше чем интересующий вызов.

**Лемма 5.3** *Динамическое программирование уменьшает время выполнения рекурсивной функции до максимального значения, которое потребуется на вычисление функции для всех аргументов, которые меньше или равны данному аргументу, при условии, что стоимость рекурсивного вызова остается постоянной.*

См. упражнение 5.50.

Применительно к задаче о ранце из леммы следует, что время выполнения пропорционально произведению  $NM$ . Таким образом, задача о ранце легко поддается решению, когда емкость ранца не очень велика; для очень больших емкостей время и требуемый объем памяти могут оказаться недопустимо большими.



**РИСУНОК 5.18 ПРИМЕНЕНИЕ МЕТОДА НИСХОДЯЩЕГО ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ ДЛЯ РЕАЛИЗАЦИИ АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ О РАНЦЕ**

*Подобно тому, как это было сделано для вычисления чисел Фибоначчи, методика с сохранением известных значений уменьшает нарастание затрат времени на выполнение алгоритма с экспоненциального (см. рис. 5.17) до линейного.*

Восходящее динамическое программирование также применимо к задаче о ранце. Действительно, метод восходящего программирования можно применять во всех случаях, когда применим метод нисходящего программирования, хотя при этом необходимо убедиться, что значения функции могут быть вычислены в соответствующем порядке, чтобы каждое нужное значение вычислялось в требуемый момент. Для функций, имеющих только один целочисленный аргумент, подобных рассмотренным, можно просто увеличить порядок аргумента (см. упражнение 5.53); для более сложных рекурсивных функций определение правильного порядка может быть сложной задачей.

Например, мы не обязаны ограничиваться рекурсивными функциями только с одним целочисленным аргументом. При наличии функции с несколькими целочисленными аргументами решения меньших подзадач можно сохранить в многомерных массивах, по одному для каждого аргумента. В других ситуациях целочисленные аргументы вообще не требуются, а вместо этого можно использовать абстрактную формулировку отдельной задачи, позволяющую разделить задачу на менее сложные. Примеры таких задач рассмотрены в частях 5—8.

При использовании нисходящего динамического программирования известные значения сохраняются; при использовании восходящего динамического программирования они вычисляются заранее. В общем случае нисходящее динамическое программирование предпочтительней восходящего, поскольку

- оно представляет собой механическую трансформацию естественного решения задачи;
- порядок решения подзадач определяется сам собой;
- решение всех подзадач может не потребоваться.

Приложения, в которых применяется динамическое программирование, различаются по сущности подзадач и объему сохраняемой для них информации.

Важный момент, которым нельзя пренебрегать, заключается в том, что динамическое программирование становится неэффективным, когда количество возможных значений функции, которые могут потребоваться, столь велико, что мы не можем себе позволить их сохранять (при нисходящем программировании) или вычислять предварительно (при восходящем программировании). Например, если в задаче о ранце  $M$  и размеры элементов — 64-разрядные величины или числа с плавающей точкой, мы не сможем сохранять значения путем их индексирования в массиве. Это несоответствие — не просто небольшое неудобство — оно создает принципиальные трудности. Для подобных задач пока не известно ни одного достаточно эффективно решения; как будет показано в части 8, имеются веские причины считать, что эффективного решения вообще не существует.

Динамическое программирование — это методика разработка алгоритмов, которая рассчитана в первую очередь для решения сложных задач того типа, который будет рассмотрен в частях 5—8. Большинство рассмотренных в частях со 2 по 4 алгоритмов представляют собой реализацию методов типа "разделяй и властвуй" с неперекрывающимися подзадачами, и основное внимание было уделено скорее субквадратичной или сублинейной зависимости производительности, чем субэкспоненциальной. Однако, нисходящее динамическое программирование является базовой



технологией разработки эффективных реализаций рекурсивных алгоритмов, которая присутствует в арсенале средств любого, кто принимает участие в создании и реализации алгоритмов.

## Упражнения

▷ **5.37** Создайте функцию, которая вычисляет  $F_N$  по модулю  $M$ , используя строго постоянный объем памяти для промежуточных вычислений.

**5.38** Каково наибольшее значение  $N$ , для которого  $F_N$  может быть представлено в виде 64-разрядного целого числа?

○ **5.39** Нарисуйте дерево, которое соответствует рис. 5.15 для случая, когда выполнена взаимная замена рекурсивных вызовов в программе 5.11.

**5.40** Создайте функцию, которая использует восходящее динамическое программирование для вычисления значения  $P_N$ , определяемого рекуррентным соотношением

$$P_N = \lfloor N/2 \rfloor + P_{\lfloor N/2 \rfloor} + P_{\lceil N/2 \rceil}, \quad \text{для } N \geq 1, \text{ при } P_0 = 0.$$

Нарисуйте график зависимости  $P_N - N \lg N/2$  от  $N$  для  $0 \leq N \leq 1024$ .

**5.41** Создайте функцию, в которой восходящее динамическое программирование используется для решения упражнения 5.40.

○ **5.42** Нарисуйте дерево, которое соответствует рис. 5.15 для функции из упражнения 5.41 при значении  $N = 23$ .

**5.43** Нарисуйте график зависимости количества рекурсивных вызовов, выполняемых функцией из упражнения 5.41 для вычисления  $P_N$ , от  $N$  для  $0 \leq N \leq 1024$ . (Для выполнения этих вычислений для каждого значения  $N$  программа должна запускаться заново.)

**5.44** Создайте функцию, в которой восходящее динамическое программирование используется для вычисления значения  $C_N$ , определенного рекуррентным соотношением

$$C_N = N + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}),$$

для  $N \geq 1$ , при  $C_0 = 1$

**5.45** Создайте функцию, в которой нисходящее динамическое программирование применяется для решения упражнения 5.44.

○ **5.46** Нарисуйте дерево, которое соответствует рис. 5.15 для функции из упражнения 5.45 при значении  $N = 23$ .

**5.47** Нарисуйте график зависимости количества рекурсивных вызовов, выполняемых функцией из упражнения 5.45 для вычисления  $C_N$ , от  $N$  для  $0 \leq N \leq 1024$ . (Для выполнения этих вычислений для каждого значения  $N$  программа должна запускаться заново.)

▷ **5.48** Приведите содержимое массивов `maxKnown` и `itemKnown`, вычисленное программой 5.13 для вызова `knapsack(17)` применительно к элементам, приведенным на рис. 5.16.

- ▷ **5.49** Приведите дерево, соответствующее рис. 5.18 при допущении, что элементы рассматриваются в порядке уменьшения их размеров.
- **5.50** Докажите лемму 5.3.
- **5.51** Создайте функцию, решающую задачу о ранце, используя версию программы 5.12, в которой применяется восходящее динамическое программирование.
- **5.52** Создайте функцию, которая решает задачу о ранце методом нисходящего динамического программирования, но используя при этом рекурсивное решение, основывающееся на вычислении оптимального количества конкретного элемента, который должен быть помещен в ранец, исходя из определения (рекурсивно) оптимального способа упаковки ранца без этого элемента.
- **5.53** Создайте функцию, решающую задачу о ранце, используя версию рекурсивного решения, описанного в упражнении 5.52, в которой применяется восходящее динамическое программирование.
- **5.54** Воспользуйтесь динамическим программированием для решения упражнения 5.4. Обеспечьте отслеживание общего количества сохраняемых вызовов функций.

**5.55** Создайте программу, в которой нисходящее динамическое программирование применяется для вычисления биномиального коэффициента  $\binom{N}{k}$ , исходя из рекурсивного соотношения

$$\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1}, \quad \text{при} \quad \binom{N}{0} = \binom{N}{N} = 1.$$

## 5.4 Деревья

Деревья — это математические абстракции, играющие главную роль при разработке и анализе алгоритмов, поскольку

- мы используем деревья для описания динамических свойств алгоритмов;
- мы строим и используем явные структуры данных, которые являются конкретными реализациями деревьев.

Мы уже встречались с примерами обоих применений деревьев. В главе 1 были разработаны алгоритмы для решения задачи подключения, которые основывались на структурах деревьев, а в разделах 5.2 и 5.3 структура вызовов рекурсивных алгоритмов была описана с помощью структур деревьев.

Мы часто встречаемся с деревьями в повседневной жизни — это основное понятие очень хорошо знакомо. Например, многие люди отслеживают предков и наследников с помощью генеалогического дерева; как будет показано, значительная часть терминов заимствована именно из этой области применения. Еще один пример — организация спортивных турниров; среди прочих, в исследовании этого применения принял участие и Льюис Кэрролл (Lewis Carroll). В качестве третьего примера можно привести организационную диаграмму большой корпорации; это применение отличается иерархическим разделением, характерным для алгоритмов типа "разделяй и властвуй". Четвертым примером служит дерево синтаксического разложения предло-

жения английского (или любого другого языка) на составляющие его части; такое дерево близко связано с обработкой компьютерных языков, как описано в части 5. Типичный пример дерева, в данном случае описывающего структуру этой книги, показан на рис. 5.19. В книге рассматривается и множество других примеров применения деревьев.

Применительно к компьютерам, одно из наиболее известных применений структур деревьев — для организации файловых систем. Файлы хранятся в *каталогах* (иногда называемых также *папками*), которые рекурсивно определяются как последовательности каталогов и файлов. Это рекурсивное определение снова отражает естественное рекурсивное разбиение на составляющие и идентично определению определенного типа дерева.

Существует множество различных типов деревьев, и важно понимать различие между абстракцией и конкретным представлением, с которым выполняется работа для данного приложения. Соответственно, мы подробно рассмотрим различные типы деревьев и их представления. Рассмотрение начнется с определения деревьев как абстрактных объектов и с ознакомления с большинством основных связанных с ними терминов. Мы неформально рассмотрим различные типы деревьев, которые следует рассматривать в порядке сужения самого этого понятия:

- Деревья
- Деревья с корнем
- Упорядоченные деревья
- М-арные и бинарные деревья

После этого неформального рассмотрения мы перейдем к формальным определениям и рассмотрим представления и приложения. На рис. 5.20 приведена иллюстрация многих из этих базовых концепций, которые будут сначала рассматриваться, а затем и определяться.

*Дерево (tree)* — это непустая коллекция вершин и ребер, удовлетворяющих определенным требованиям. *Вершина (vertex)* — это простой объект (называемый также *узлом (node)*), который может иметь имя и содержать другую связанную с ним информацию; *ребро (edge)* — это связь между двумя вершинами. *Путь (path)* в дереве — это список отдельных вершин, в котором следующие друг за другом вершины соединяются ребрами дерева. Определяющее свойство дерева — существование только одного пути, соединяющего любые два узла. Если между какой-либо парой узлов существует более одного пути или если между какой-либо парой узлов путь отсутствует, мы имеем граф, а не дерево. Несвязанный набор деревьев называется *бором (forest)*.

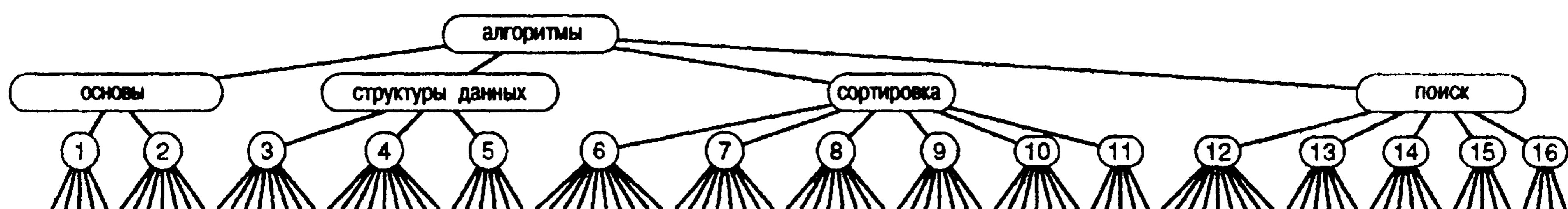
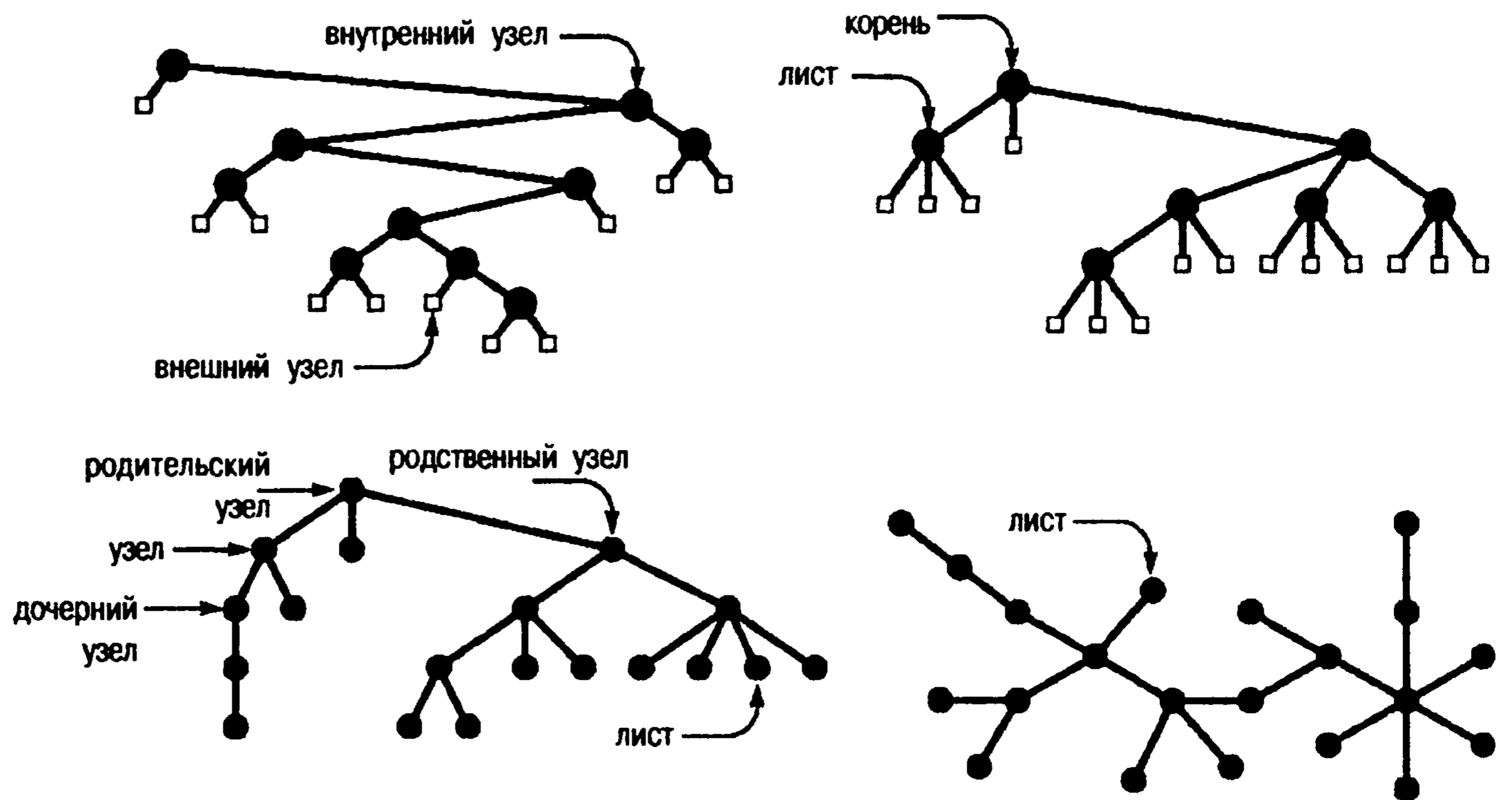


РИСУНОК 5.19 ДЕРЕВО

Это дерево описывает части, главы и разделы этой книги. Каждый элемент представлен узлом. Каждый узел связан нисходящими связями с составляющими его частями и восходящей связью — с большей частью, к которой он принадлежит.

## РИСУНОК 5.20 ТИПЫ ДЕРЕВЬЕВ

На этих схемах приведены примеры двоичного дерева (вверху слева), троичного дерева (вверху справа), дерева с корнем (внизу слева) и свободного дерева (внизу справа).



Дерево с корнем (единственным, *rooted*) — это дерево, в котором один узел назначен *корнем* (*root*) дерева. В области компьютеров термин *дерево* обычно применяется к деревьям с корнем, а термин *свободное дерево* (*free tree*) — к более общим структурам, описанным в предыдущем абзаце. В дереве с корнем любой узел является корнем поддерева, состоящего из него и расположенных под ним узлов.

Существует только один путь между корнем и каждым из других узлов дерева. Данное определение никак не определяет направление ребер; обычно считается, что все ребра указывают от корня или к корню, в зависимости от приложения. Обычно деревья с корнем рисуются с корнем, расположенным в верхней части (хотя на первый взгляд это соглашение кажется неестественным), и говорят, что узел *у* располагается *под* узлом *x* (а *x* располагается над *у*), если *x* находится на пути от *у* к корню (т.е., *у* находится под *x*, как нарисовано на странице, и соединяется с *x* путем, который не проходит через корень). Каждый узел (за исключением корня) имеет только один узел над ним, который называется его *родительским узлом* (*parent*); узлы, расположенные непосредственно под данным узлом, называются его *дочерними узлами* (*children*). Иногда аналогия с генеалогическими деревьями расширяется еще больше и тогда говорят об узлах-*предках* (*grand parent*) или *родственных* (*sibling*) узлах данного узла.

Узлы, не имеющие дочерних узлов, называются *листьями* (*leaves*) или *терминальными* (*оконечными, terminal*) узлами. Для соответствия с последним применением узлы, имеющие хотя бы один дочерний узел, иногда называются *нетерминальными* (*nonterminal*) узлами. В этой главе мы уже встречались с примером утилиты, различающей эти типы узлов. В деревьях, которые использовались для представления структуры вызовов рекурсивных алгоритмов (например, рис. 5.14), нетерминальные узлы (окружности) представляют вызовы функций с рекурсивными вызовами, а терминальные узлы (квадраты) представляют вызовы функций без рекурсивных вызовов.

В определенных приложениях способ упорядочения дочерних узлов каждого узла имеет значение; в других это не важно. *Упорядоченное* (*ordered*) дерево — это дерево с корнем, в котором определен порядок следования дочерних узлов каждого узла. Упорядоченные деревья — естественное представление: например, при рисовании дерева дочерние узлы размещаются в определенном порядке. Действительно, многие другие конкретные представления имеют аналогично предполагаемый порядок; на-

пример, обычно это различие имеет значение при рассмотрении представления деревьев в компьютере.

Если каждый узел *должен* иметь конкретное количество дочерних узлов, появляющихся в конкретном порядке, мы имеем *M-арное дерево*. В таком дереве часто можно определить специальные внешние узлы, которые не имеют дочерних узлов. Тогда внешние узлы могут действовать в качестве фиктивных, на которые ссылаются узлы, не имеющие указанного количества дочерних узлов. В частности, простейшим типом *M-арного* дерева является бинарное дерево. *Бинарное дерево (binary tree)* — это упорядоченное дерево, состоящее из узлов двух типов: внешних узлов без дочерних узлов и внутренних узлов, каждый из которых имеет ровно два дочерних узла. Поскольку два дочерних узла каждого внутреннего узла упорядочены, говорят о *левом дочернем узле (left child)* и *правом дочернем узле (right child)* внутренних узлов. Каждый внутренний узел должен иметь и левый, и правый дочерние узлы, хотя один из них или оба могут быть внешними узлами. *Лист* в *M-арном* дереве — это внутренний узел, все дочерние узлы которого являются внешними.

Все это общая терминология. Далее рассматриваются формальные определения, представления и приложения, в порядке расширения понятий:

- бинарные и *M-арные* деревья
- упорядоченные деревья
- деревья с корнем
- свободные деревья

Начав с наиболее характерной абстрактной структуры, мы должны быть в состоянии подробно рассмотреть конкретные представления, как станет понятно из дальнейшего изложения.

**Определение 5.1** *Бинарное дерево* — это либо внешний узел, либо внутренний узел, связанный с парой бинарных деревьев, которые называются левым и правым поддеревьями этого узла.

Из этого определения становится понятно, что сами деревья — абстрактное математическое понятие. При работе с компьютерными представлениями мы работаем всего лишь с одной конкретной реализацией этой абстракции. Эта ситуация не отличается от представления действительных чисел значениями типа `float`, целых чисел значениями типа `int` и т.д. Когда мы рисуем дерево с узлом в корне, связанным ребрами с левым поддеревом, расположенным слева, и с правым поддеревом, расположенным справа, то выбираем удобное конкретное представление. Существует множество различных способов представления бинарных деревьев (см., например, упражнение 5.62), которые поначалу кажутся удивительными, но вполне отражают сущность, как и можно было ожидать, учитывая абстрактный характер определения.

Чаще всего применяется следующее конкретное представление при реализации программ, использующих и манипулирующих бинарными деревьями, — структура с двумя связями (правой и левой) для внутренних узлов (см. рис. 5.21). Эти структуры аналогичны связным спискам, но они имеют по две связи для каждого узла, а не по одной. Нулевые связи соответствуют внешним узлам. В частности, мы добавили связь

к стандартному представлению связного списка, приведенному в разделе 3.3, следующим образом:

```
struct node { Item item; node *l, *r; }  
typedef node *link;
```

что представляет собой всего лишь код C++ для определения 5.1. Узлы состоят из элементов и пар указателей на узлы, и указатели на узлы называются также связями. Так, например, мы реализуем абстрактную операцию *перехода к левому поддереву* с помощью ссылки на указатель типа  $x = x->l$ .

Это стандартное представление позволяет построить эффективную реализацию операций, которые вызываются для перемещения по дереву *вниз* от корня, но не для перемещения по дереву *вверх* от дочернего узла к его родительскому узлу. Для алгоритмов, требующих использования таких операций, можно добавить третью связь для каждого узла, направленную к его родительскому узлу. Эта альтернатива аналогична двухсвязным спискам. Как и в случае со связными списками (см. рис. 3.6), в определенных ситуациях узлы дерева хранятся в массиве, а в качестве связей используются индексы, а не указатели. Конкретный пример такой реализации исследуется в разделе 12.7. Для определенных специальных алгоритмов используются другие представления бинарных деревьев, что наиболее полно исследуется в главе 9.

Из-за наличия такого множества различных возможных представлений можно было бы разработать ADT (Abstract Data Type) (абстрактный тип данных) бинарного дерева, инкапсулирующий важные операции, которые нужно выполнять, и разделяющий использование и реализацию этих операций. В данной книге данный подход не используется, поскольку

- чаще всего мы используем представление с двумя связями;
- мы используем деревья для реализации ADT более высокого уровня, и хотим сосредоточить внимание на этой теме;
- мы работаем с алгоритмами, эффективность которых зависит от конкретного представления, — это обстоятельство может быть упущено в ADT.

По этим же причинам мы используем уже знакомые конкретные представления массивов и связных списков. Представление бинарного дерева, отображенное на рис. 5.21 — один из фундаментальных инструментов, который теперь добавлен к этому краткому списку.

Анализируя связные списки, мы начали рассмотрение с элементарных операций вставки и удаления узлов (см. рис. 3.3 и 3.4). При использовании стандартного представления бинарных деревьев такие операции необязательно являются элементарными из-за наличия второй связи. Если нужно удалить узел из бинарного дерева, приходится решать принципиальную проблему наличия двух дочерних узлов и только одного родительского, с которыми нужно работать после удаления узла. Существуют три естественных операции, для которых подобное осложнение не возникает: вставка нового узла в нижней части дерева (замена нулевой связи связью с новым узлом), удаление листа (замена связи с ним нулевой связью) и объединение двух деревьев посредством создания нового корня, левая связь которого указывает на одно дерево, а правая — на другое. Эти операции интенсивно используются при манипулировании бинарными деревьями.

**Определение 5.2** *M-арное дерево* — это внешний узел, либо внутренний узел, связанный с упорядоченной последовательностью  $M$  деревьев, которые также являются  $M$ -арными деревьями.

Обычно узлы в  $M$ -арных деревьях представляются либо в виде структур с  $M$  именованными связями (как в бинарных деревьях), либо в виде массивов  $M$  связей. Например, в главе 15 рассмотрены 3-арные (или *троичные*) деревья, в которых используются структуры с тремя именованными связями (левой, средней и правой), каждая из которых имеет специальное значение для связанных с ними алгоритмов. В остальных случаях использование массивов для хранения связей вполне подходит, поскольку значение  $M$  фиксировано, хотя, как будет показано, при использовании такого представления особое внимание потребуется уделить интенсивному использованию памяти.

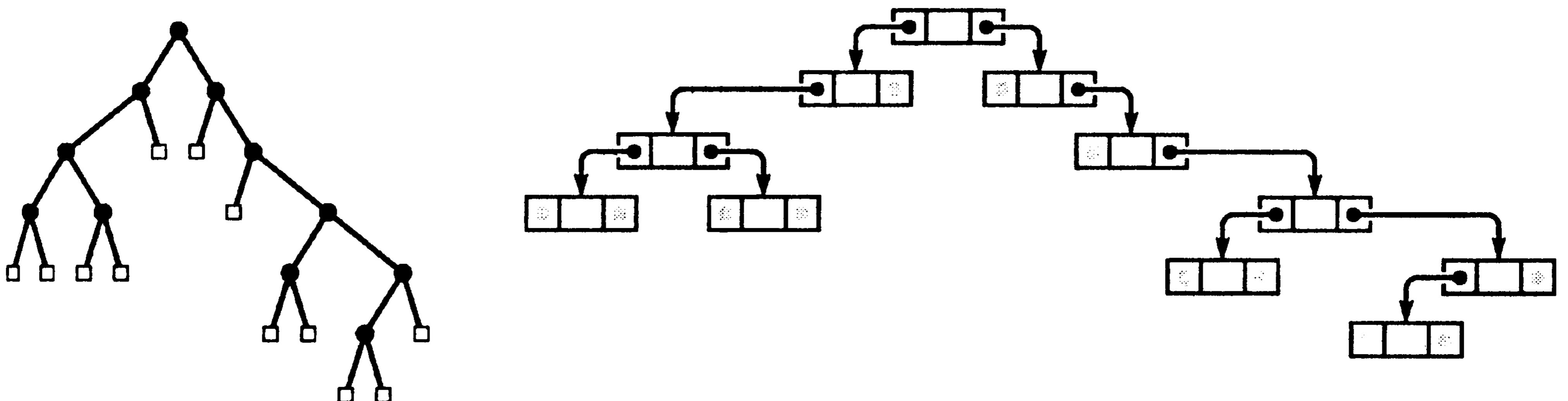
**Определение 5.3** *Дерево (также называемое упорядоченным деревом)* — это узел (называемый *корнем*), связанный с последовательностью несвязанных деревьев. Такая последовательность называется *бором*.

Различие между упорядоченными деревьями и  $M$ -арными деревьями состоит в том, что узлы в упорядоченных деревьях могут иметь любое количество дочерних узлов, в то время как узлы в  $M$ -арных деревьях должны иметь точно  $M$  дочерних узлов. Иногда в контекстах, в которых требуется различать упорядоченные и  $M$ -арные деревья, мы используем термин *главное дерево (general tree)*.

Поскольку каждый узел в упорядоченном дереве может иметь любое количество связей, представляется естественным рассматривать его с использованием связного списка, а не массива, для хранения связей с дочерними узлами. Пример такого представления приведен на рис. 5.22. Из этого примера видно, что тогда каждый узел содержит две связи: одну для связного списка, соединяющего его с родственными узлами, и вторую для связного списка его дочерних узлов.

**Лемма 5.4** *Существует однозначное соответствие между бинарными деревьями и упорядоченными борами.*

Это соответствие показано на рис. 5.22. Любой бор можно представить в виде бинарного дерева, сделав левую связь каждого узла указывающей на его левый дочерний узел, а правую связь каждого узла — указывающей на родственный узел, расположенный справа.



**РИСУНОК 5.21 ПРЕДСТАВЛЕНИЕ БИНАРНОГО ДЕРЕВА**

В стандартном представлении бинарного дерева используются узлы с двумя связями: левой связью с левым поддеревом и правой связью с правым поддеревом. Нулевые связи соответствуют внешним узлам.

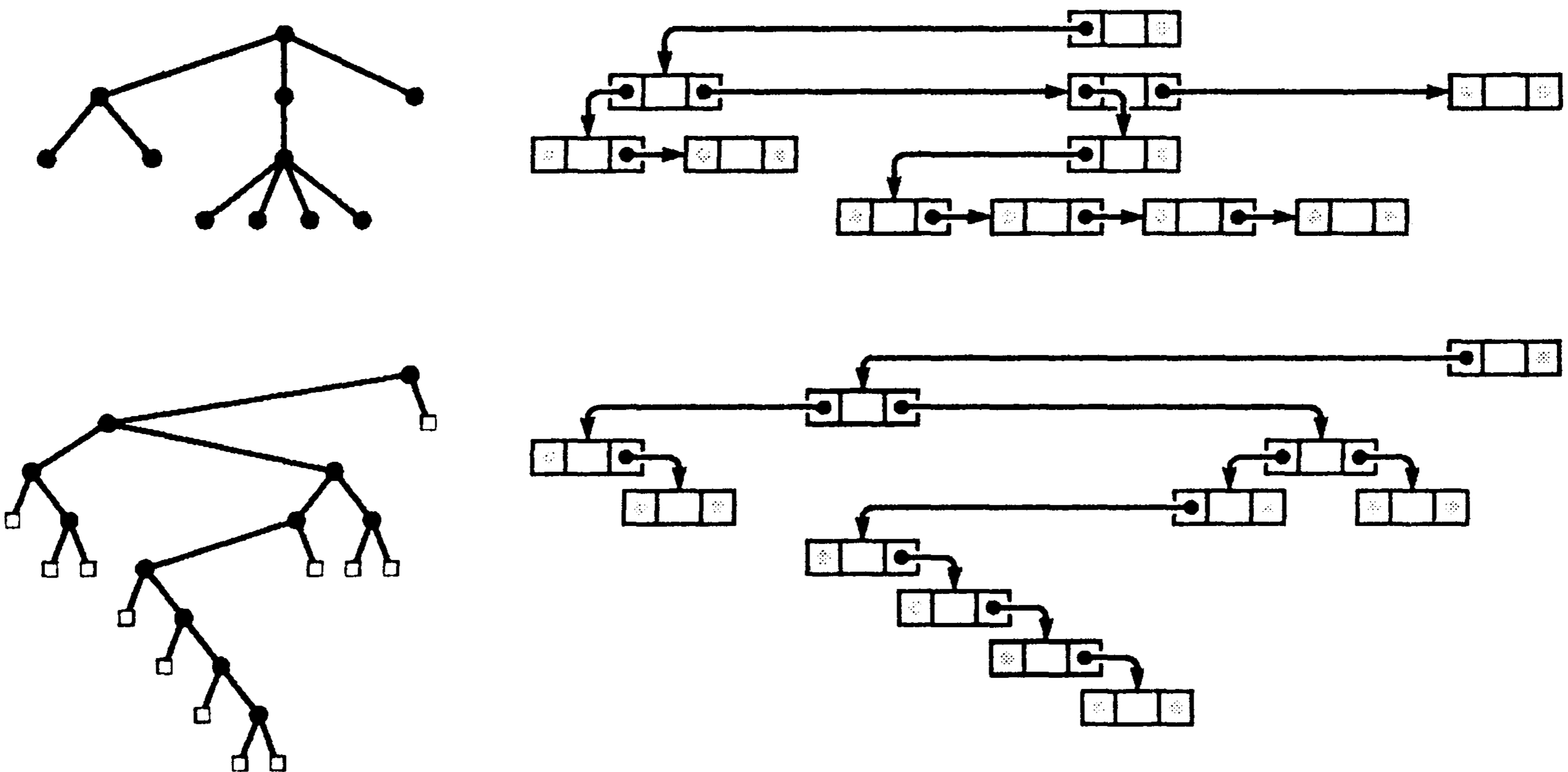
**Определение 5.4** *Дерево с корнем (или неупорядоченное дерево) — это узел (называемый корнем), связанный с множественным набором деревьев с корнем. (Такой множественный набор называется неупорядоченным бором.)*

Деревья, с которыми мы встречались в главе 1, посвященной проблеме связности, являются неупорядоченными деревьями. Такие деревья могут быть определены в качестве упорядоченных деревьев, в которых порядок рассмотрения дочерних узлов узла не имеет значения. Неупорядоченные деревья можно было бы также определить в виде набора взаимосвязей между родительскими и дочерними узлами. Такое представление может казаться не связанным с рассматриваемыми рекурсивными структурами, но, вероятно, является конкретным представлением, которое в основном соответствует абстрактному подходу.

Неупорядоченное дерево можно было бы представить в компьютере упорядоченным деревом; при этом приходится признать, что несколько различных упорядоченных деревьев могут представлять одно и то же неупорядоченное дерево. Действительно, обратная задача определения того, представляют ли два различных упорядоченные дерева одно и то же неупорядоченное дерево (задача *изоморфизма дерева*) трудно поддается решению.

Наиболее общим типом деревьев является дерево, в котором не выделен ни один корневой узел. Например, остовные деревья, полученные в результате работы алгоритмов связности из главы 1, обладают этим свойством. Для правильного определения *деревьев без корня, неупорядоченных деревьев и свободных деревьев* потребуется начать с определения *графов (graphs)*.

**Определение 5.5** *Граф — это набор узлов с набором ребер, которые соединяют пары отдельных узлов (причем любую пару узлов соединяет только одно ребро).*



**РИСУНОК 5.22 ПРЕДСТАВЛЕНИЕ ДЕРЕВА**

*Представление упорядоченного дерева за счет поддержания связного списка дочерних узлов каждого узла эквивалентно представлению его в виде двоичного дерева. На схеме справа вверху показано представление в виде связного списка дочерних узлов для дерева, показанного слева вверху; при этом список реализован в правых связях узлов, а левая связь каждого узла указывает на первый узел в связном списке его дочерних узлов. На схеме справа внизу приведена несколько измененная версия верхней схемы; она представляет бинарное дерево, изображенное слева внизу. Таким образом, бинарное дерево можно рассматривать в качестве представления дерева.*



Представьте себе, что перемещаетесь вдоль ребра от одного какого-либо узла до другого, затем от этого узла к следующему и т.д. Последовательность ребер, ведущих от одного узла до другого, когда ни один узел не посещается дважды, называется *простым путем*. Граф является *связным (connected)*, если существует простой путь, связывающий любую пару узлов. Простой путь, у которого первый и последний узел совпадают, называется *циклом (cycle)*.

Каждое дерево является графом; а какие же графы являются деревьями? Граф считается деревом, если он удовлетворяет любому из следующих четырех условий:

- Граф имеет  $N - 1$  ребер и ни одного цикла.
- Граф имеет  $N - 1$  ребер и является связным.
- Только один простой путь соединяет каждую пару вершин в графе.
- Граф является связным, но перестает быть таковым при удалении любого ребра.

Любое из этих условий — необходимое и достаточное условие для выполнения остальных трех. Формально, одно из них должно было бы служить определением *свободного дерева*; неформально, они все вместе служат определением.

Мы представляем свободное дерево в виде коллекции ребер. Если представлять свободное дерево неупорядоченным, упорядоченным или даже бинарным деревом, придется признать, что в общем случае существует множество различных способов представления каждого свободного дерева.

Абстракция дерева используется часто, и рассмотренные в этом разделе различия важны, поскольку знание различных абстракций деревьев — существенная составляющая определения эффективного алгоритма и соответствующих структур данных для решения данной задачи. Часто приходится работать непосредственно с конкретными представлениями деревьев без учета конкретной абстракции, но в то же время часто имеет смысл поработать с соответствующей абстракцией дерева, а затем рассмотреть различные конкретные представления. В книге приведено множество примеров этого процесса.

Прежде чем вернуться к алгоритмам и представлениям, мы рассмотрим ряд основных математических свойств деревьев; эти свойства будут использоваться при разработке и анализе алгоритмов в виде деревьев.

## Упражнения

- ▷ 5.56 Приведите представления свободного дерева, показанного на рис. 5.20, в форме дерева с корнем и бинарного дерева.
- 5.57 Сколько существует различных способов представления свободного дерева, показанного на рис. 5.20, в форме упорядоченного дерева?
- ▷ 5.58 Нарисуйте три упорядоченных дерева, которые изоморфны по отношению к упорядоченному дереву, показанному на рис. 5.20. Другими словами, должна существовать возможность преобразования всех четырех деревьев одного в другое путем обмена дочерними узлами.
- 5.59 Предположите, что деревья содержат элементы, для которых определена операция `operator==`. Создайте рекурсивную программу, которая удаляет в бинарном дереве все листья, содержащие элементы, равные данному (см. программу 5.5).

- **5.60** Измените функцию "разделяй и властвуй" для поиска максимального элемента в массиве (программа 5.6), чтобы она делила массив на  $k$  частей, размер которых различался бы не более чем на 1, рекурсивно находила максимум в каждой части и возвращала максимальный из максимумов.
- 5.61** Нарисуйте 3-арные и 4-арные деревья, соответствующие использованию  $k = 3$  и  $k = 4$  в рекурсивной конструкции, предложенной в упражнении 5.60, для массива, состоящего из 11 элементов (см. рис. 5.6).
- **5.62** Бинарные деревья эквивалентны двоичным строкам, в которых нулевых битов на 1 больше, чем единичных при соблюдении дополнительного ограничения, что в любой позиции  $k$  количество нулевых битов слева от  $k$  не больше, чем количество единичных битов слева от  $k$ . Бинарное дерево — это либо нуль, либо две таких строки, объединенные вместе, которым предшествует 1. Нарисуйте бинарное дерево, соответствующее строке  
1110010110001011000
- **5.63** Упорядоченные деревья эквивалентны согласованным парам круглых скобок: упорядоченное дерево — это либо нуль, либо последовательность упорядоченных деревьев, заключенных в круглые скобки. Нарисуйте упорядоченное дерево, соответствующее строке  
((() (()) ())) (()) (())
- **5.64** Создайте программу для определения того, представляют ли два массива  $N$  целочисленных значений между 0 и  $N - 1$  изоморфные неупорядоченные деревья, если интерпретируются (как в главе 1) в форме связей типа родительский-дочерний в дереве с узлами, пронумерованными от 0 до  $N - 1$ . То есть, программа должна определять, существует ли способ изменения нумерации узлов в одном дереве, чтобы представление в виде массива одного дерева было идентичным представлению в виде массива другого дерева.
- **5.65** Создайте программу для определения того, представляют ли два бинарных дерева изоморфные неупорядоченные деревья.
- ▷ **5.66** Нарисуйте все упорядоченные деревья, которые могли бы представлять дерево, определенное набором ребер 0-1, 1-2, 1-3, 1-4, 4-5.
- **5.67** Докажите, что если в связанном графе, состоящем из  $N$  узлов, удаление любого ребра влечет за собой разъединение графа, то он имеет  $N - 1$  ребер и ни одного цикла.

## 5.5 Математические свойства бинарных деревьев

Прежде чем приступить к рассмотрению алгоритмов обработки деревьев, продолжим математическую тему, рассмотрев ряд базовых свойств деревьев. Мы сосредоточим внимание на бинарных деревьях, поскольку они используются в книге чаще других. Понимание их основных свойств послужит фундаментом для понимания характеристик производительности различных алгоритмов, с которыми мы встретимся — не только тех, в которых бинарные деревья используются в качестве явных структур данных, но и рекурсивных алгоритмов типа "разделяй и властвуй" и других аналогичных применений.

**Лемма 5.5** *Бинарное дерево с  $N$  внутренними узлами имеет  $N + 1$  внешних узлов.*

Эта лемма доказывается методом индукции: бинарное дерево без внутренних узлов имеет один внешний узел, следовательно, лемма справедлива для  $N = 0$ . Для  $N > 0$  любое бинарное дерево с  $N$  внутренними узлами имеет  $k$  внутренних узлов в левом поддереве и  $N-1-k$  внутренних узлов в правом поддереве для некоторого  $k$  в диапазоне между 0 и  $N-1$ , поскольку корень является внутренним узлом. В соответствии с индуктивным предположением левое поддерево имеет  $k + 1$  внешних узлов, а правое поддерево —  $N-k$  внешних узлов, что в сумме составляет  $N + 1$ .

**Лемма 5.6** *Бинарное дерево с  $N$  внутренними узлами имеет  $2N$  связей:  $N-1$  связей с внутренними узлами и  $N + 1$  связей с внешними узлами.*

В каждом дереве с корнем каждый узел, за исключением корня, имеет единственный родительский узел, и каждое ребро соединяет узел с его родительским узлом; следовательно, существует  $N-1$  связей, соединяющих внутренние узлы. Аналогично, каждый из  $N + 1$  внешних узлов имеет одну связь со своим единственным родительским узлом.

Характеристики производительности многих алгоритмов зависят не только от количества узлов в связанных с ними деревьях, но и от различных структурных свойств.

**Определение 5.6** *Уровень (level) узла в дереве на единицу выше уровня его родительского узла (корень размещается на уровне 0). Высота (height) дерева — максимальный из уровней узлов дерева. Длина пути (path length) дерева — сумма уровней всех узлов дерева. Длина внутреннего пути (internal path length) бинарного дерева — сумма уровней всех внутренних узлов дерева. Длина внешнего пути (external path length) бинарного дерева — сумма уровней всех внешних узлов дерева.*

Удобный способ вычисления длины пути дерева заключается в суммировании произведений  $k$  на число узлов на уровне  $k$  для всех  $k$ .

Из этих соотношений следуют также простые рекурсивные определения, вытекающие непосредственно из рекурсивных определений деревьев и бинарных деревьев. Например, высота дерева на 1 больше максимальной высоты поддеревьев его корня, а длина пути дерева, имеющего  $N$  узлов равна сумме длин путей поддеревьев его корня плюс  $N-1$ . Приведенные соотношения также непосредственно связаны с анализом рекурсивных алгоритмов. Например, для многих рекурсивных вычислений высота соответствующего дерева в точности равна максимальной глубине рекурсии, или размеру стека, необходимого для поддержки вычисления.

**Лемма 5.7** *Длина внешнего пути любого бинарного дерева, имеющего  $N$  внутренних узлов, на  $2N$  больше длины внутреннего пути.*

Эту лемму можно было бы доказать методом индукции, но есть другое, более наглядное доказательство (которое применимо и для доказательства леммы 5.6). Обратите внимание, что любое бинарное дерево может быть сконструировано при помощи следующего процесса: начните с бинарного дерева, состоящего из одного внешнего узла. Затем повторите следующее  $N$  раз: выберите внешний узел и замените его новым внутренним узлом с двумя дочерними внешними узлами. Если выбранный внешний узел располагается на уровне  $k$ , длина внутреннего пути увеличивается на  $k$ , но длина внешнего пути увеличивается на  $k + 2$  (один внешний

узел на уровне  $k$  удаляется, но два на уровне  $k + 1$  добавляются). Процесс начинается с дерева, длина внутреннего и внешнего путей которого равны 0, и на каждом из  $N$  шагов длина внешнего пути увеличивается на 2 больше, чем длина внутреннего пути.

**Лемма 5.8** *Высота бинарного дерева с  $N$  внутренними узлами не меньше  $\lg N$  и не больше  $N - 1$ .*

Худший случай — вырожденное дерево, имеющее только один лист и  $N - 1$  связь от корня до листа (см. рис. 5.23). В лучшем случае мы имеем уравновешенное дерево с  $2^i$  внутренними узлами на каждом уровне  $i$ , за исключением самого нижнего (см. рис. 5.23). Если высота равна  $h$ , то должно быть справедливо соотношение

$$2^{h-1} < N + 1 \leq 2^h$$

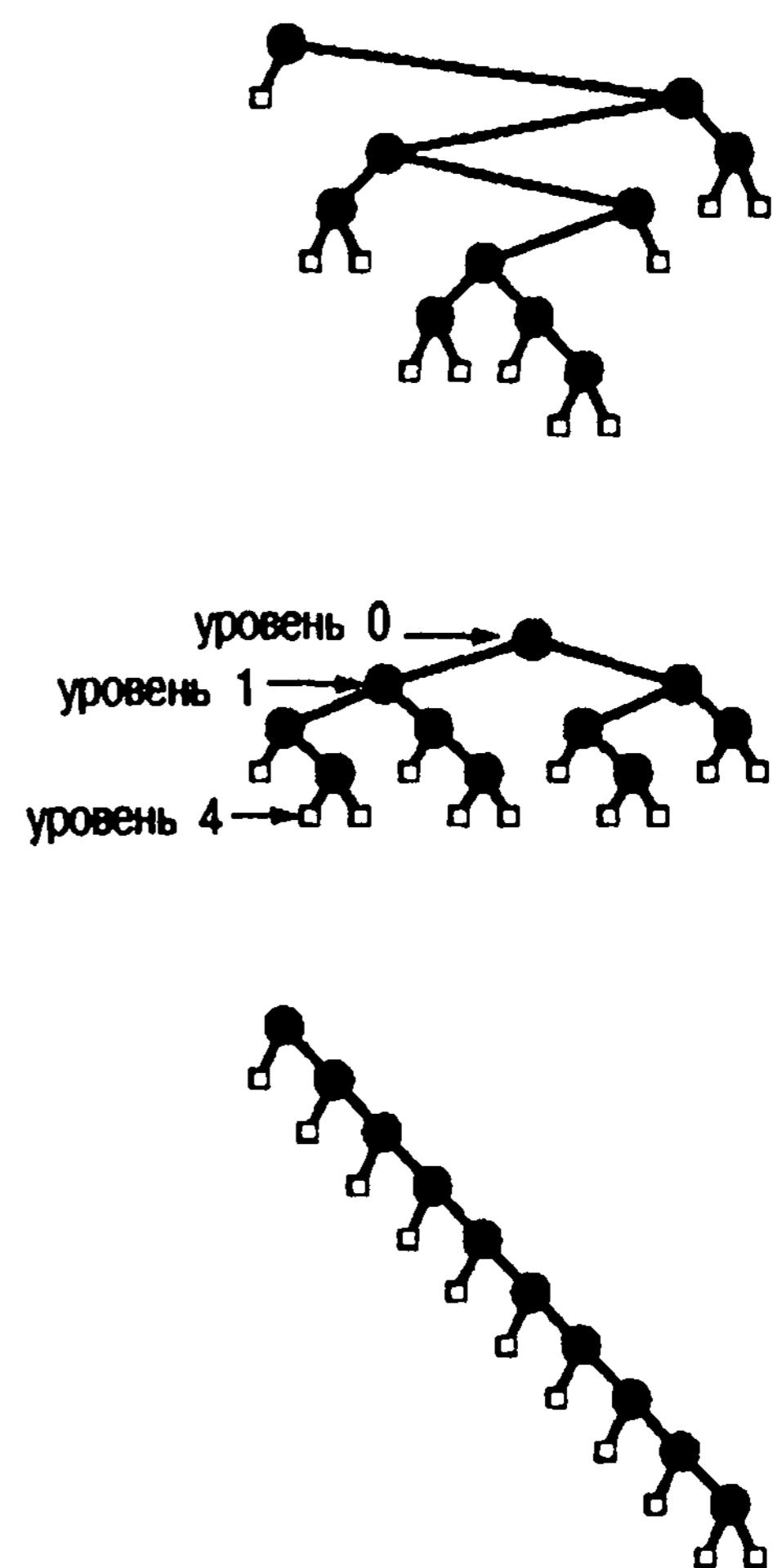
поскольку существует  $N + 1$  внешних узлов. Из этого неравенства следует провозглашенная лемма: в лучшем случае высота равна  $\lg N$ , округленному до ближайшего целого числа.

**Лемма 5.9** *Длина внутреннего пути бинарного дерева с  $N$  внутренними узлами не меньше чем  $N \lg(N/4)$  и не превышает  $N(N - 1)/2$ .*

Худший и лучший случай соответствуют тем же деревьям, которые упоминались при рассмотрении леммы 5.8 и показаны на рис. 5.23. В худшем случае длина внутреннего пути дерева равна  $0 + 1 + 2 + \dots + (N - 1) = N(N - 1)/2$ . В лучшем случае дерево имеет  $(N + 1)$  внутренних узлов при высоте, не превышающей  $\lg N$ . Перемножая эти значения и применяя лемму 5.7, мы получаем предельное значение  $(N + 1) \lfloor \lg N \rfloor - 2N < N \lg(N/4)$ .

Как мы увидим, бинарные деревья часто используются в компьютерных приложениях, и при этом производительность наивысшая, когда бинарные деревья полностью уравновешены (или близки к этому). Например, деревья, которые использовались нами для описания алгоритмов "разделяй и властвуй", подобных бинарному поиску и сортировке слиянием, полностью уравновешены (см. упражнение 5.74). В главах 9 и 13 исследуются явные структуры данных, основывающиеся на уравновешенных деревьях.

Эти основные свойства деревьев предоставляют информацию, которая потребуется для разработки эффективных алгоритмов решения многих практических задач. Более подробный анализ нескольких специфичных алгоритмов, с которыми придется встретиться, требует сложного математического анализа, хотя часто полезные прибли-



**РИСУНОК 5.23** БИНАРНОЕ ДЕРЕВО С 10 ВНУТРЕННИМИ УЗЛАМИ

*Бинарное дерево, показанное сверху, имеет высоту равную 7, длину внутреннего пути равную 31 и длину внешнего пути равную 51. Полностью уравновешенное бинарное дерево (в центре) с 10 внутренними узлами имеет высоту равную 4, длину внутреннего пути равную 19 и длину внешнего пути равную 39 (ни одно двоичное дерево с 10 узлами не может иметь меньшие значения любых из этих параметров). Вырожденное дерево (внизу) с 10 внутренними узлами имеет высоту равную 10, длину внутреннего пути равную 45 и длину внешнего пути равную 65 (ни одно бинарное дерево с 10 узлами не может иметь большие значения любых из этих параметров).*

женные оценки можно получить, прибегнув к прямолинейным индуктивным аргументам, подобным использованным в этом разделе. В последующих главах мы продолжим рассмотрение математических свойств деревьев по мере возникновения необходимости. Пока можно вернуться к теме алгоритмов.

## Упражнения

- ▷ **5.68** Сколько внешних узлов существует в  $M$ -арном дереве с  $N$  внутренними узлами? Используйте свой ответ для определения объема памяти, необходимого для представления такого дерева, если считать, что для каждой связи и каждого элемента требуется по одному слову памяти.
- 5.69** Приведите верхнее и нижнее граничные значения высоты  $M$ -арного дерева с  $N$  внутренними узлами.
- **5.70** Приведите верхнее и нижнее граничные значения длины внутреннего пути  $M$ -арного дерева с  $N$  внутренними узлами.
- 5.71** Приведите верхнее и нижнее граничные значения количества листьев в бинарном дереве с  $N$  узлами.
- **5.72** Покажите, что если листья внешних узлов в бинарном дереве различаются на константу, то высота составляет  $O(\log N)$ .
- **5.73** *Дерево Фибоначчи* высотой  $n > 2$  — это бинарное дерево с деревом Фибоначчи высотой  $n - 1$  в одном поддереве и деревом Фибоначчи высотой  $n - 2$  — в другом. Дерево Фибоначчи высотой 0 — это единственный внешний узел, а дерево Фибоначчи высотой 1 — единственный внутренний узел с двумя внешними дочерними узлами (см. рис. 5.14). Выразите высоту и длину внешнего пути дерева Фибоначчи высотой  $n$  в виде функции  $N$  (количества узлов в дереве).
- 5.74** *Дерево типа "разделяй и властвуй"*, состоящее из  $N$  узлов, — это бинарное дерево с корнем, обозначенным  $N$ , деревом типа "разделяй и властвуй", состоящим из  $\lfloor N/2 \rfloor$  узлов, в одном поддереве и деревом типа "разделяй и властвуй", состоящим из  $\lceil N/2 \rceil$  узлов, в другом. (Дерево типа "разделяй и властвуй" показано на рис. 5.6.) Нарисуйте дерево типа "разделяй и властвуй" с 11, 15, 16 и 23 узлами.
- **5.75** Докажите методом индукции, что длина внутреннего пути дерева типа "разделяй и властвуй" находится в пределах между  $N \lg N$  и  $N \lg N + N$ .
- 5.76** *Дерево типа "объединяй и властвуй"*, состоящее из  $N$  узлов, — это бинарное дерево с корнем, обозначенным  $N$ , деревом типа "объединяй и властвуй", состоящим из  $\lfloor N/2 \rfloor$  узлов, в одном поддереве и деревом типа "объединяй и властвуй", состоящим из  $\lceil N/2 \rceil$  узлов, в другом (см. упражнение 5.18). Нарисуйте дерево типа "объединяй и властвуй" с 11, 15, 16 и 23 узлами.
- 5.77** Докажите методом индукции, что длина внутреннего пути дерева типа "объединяй и властвуй" находится в пределах между  $N \lg N$  и  $N \lg N + N$ .
- 5.78** *Полное (complete) бинарное дерево* — это дерево, в котором все уровни, кроме, возможно, последнего, который заполняется слева направо, заполнены, как проиллюстрировано на рис. 5.24. Докажите, что длина внутреннего пути полного дерева с  $N$  узлами лежит в пределах между  $N \lg N$  и  $N \lg N + N$ .

## 5.6 Обход дерева

Прежде чем рассматривать алгоритмы, конструирующие бинарные деревья и деревья, рассмотрим алгоритмы для реализации наиболее общей функции обработки деревьев — *обхода дерева*; при наличии указателя на дерево требуется систематически обработать все узлы в дереве. В связном списке переход от одного узла к другому выполняется за счет отслеживания единственной связи; однако, в случае деревьев приходится принимать решения, поскольку может существовать несколько связей, по которым можно следовать.

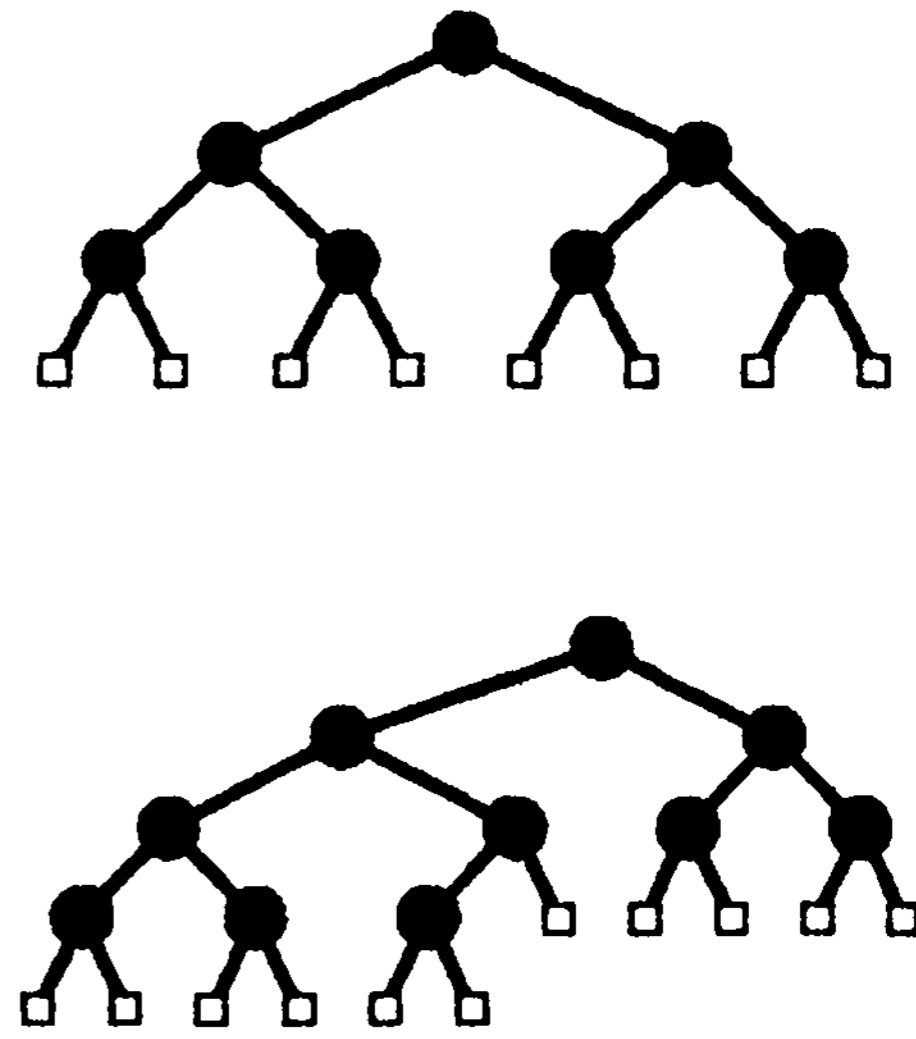
Начнем рассмотрение с процесса для бинарных деревьев. В случае со связными списками имелись две основные возможности (см. программу 5.5): обработать узел, а затем следовать связи (в этом случае узлы посещались бы по порядку), или следовать связи, а затем обработать узел (в этом случае узлы посещались бы в обратном порядке). При работе с бинарными деревьями существуют две связи и, следовательно, три основных порядка возможного посещения узлов:

- *Прямой обход (сверху вниз)*, при котором мы посещаем узел, а затем левое и правое поддеревья
- *Поперечный обход (слева направо)*, при котором мы посещаем левое поддерево, затем узел, а затем правое поддерево
- *Обратный обход (снизу вверх)*, при котором мы посещаем левое и правое поддерево, а затем узел.

Эти методы можно легко реализовать с помощью рекурсивной программы, как показано в программе 5.14, которая является непосредственным обобщением программы 5.5 обхода связного списка. Для реализации обходов в других порядках достаточно соответствующим образом переставить вызовы функций в программе 5.14. Порядок посещения узлов в примере дерева при использовании каждого из порядков обхода показан на рис. 5.26. На рис. 5.25 приведена последовательность вызовов функций, которые выполняются при вызове программы 5.14 применительно к примеру дерева из рис. 5.26.

### Программа 5.14 Рекурсивный обход дерева

Эта рекурсивная функция принимает в качестве аргумента ссылку на дерево и вызывает функцию **visit** для каждого из узлов дерева. В приведенном виде функция реализует прямой обход; если поместить обращение к **visit** между рекурсивными вызовами, получится поперечный обход; а если поместить обращение к **visit** после рекурсивных вызовов — то обратный обход.



**РИСУНОК 5.24 ПОЛНЫЕ БИНАРНЫЕ ДЕРЕВЬЯ С СЕМЬЮ И ДЕСЯТЬЮ ВНУТРЕННИМИ УЗЛАМИ**

*Когда количество внешних узлов является степенью 2 (верхний рисунок), все внешние узлы в полном бинарном дереве располагаются на одном уровне. В противном случае (нижний рисунок) внешние узлы располагаются в двух уровнях при размещении внутренних узлов слева от внешних узлов предпоследнего уровня.*

```

void traverse(link h, void visit(link))
{
    if (h == 0) return;
    visit(h);
    traverse(h->l, visit);
    traverse(h->r, visit);
}

```

С этими же основными рекурсивными процессами, на которых основываются различные методы обхода дерева, мы уже встречались в рекурсивных программах типа "разделяй и властвуй" (см. рис. 5.8 и 5.11) и в арифметических выражениях. Например, выполнение прямого обхода соответствует рисованию вначале меток на линейке, а затем выполнению рекурсивных вызовов (см. рис. 5.11); выполнение поперечного обхода соответствует перемещению самого большого диска в решении задачи о ханойских башнях между рекурсивными вызовами, которые перемещают все остальные диски; выполнение обратного обхода соответствует вычислению постфиксных выражений и т.д. Эти соответствия позволяют немедленно заглянуть внутрь механизмов, лежащих в основе обхода дерева. Например, известно, что при поперечном обходе каждый второй узел является внешним, по той же причине, по какой при решении задачи о ханойских башнях каждое второе перемещение затрагивает маленький диск.

Полезно также рассмотреть нерекурсивные реализации, в которых используется явный стек. Для простоты мы начнем с рассмотрения абстрактного стека, содержащего элементы дерева, инициализированного деревом, которое требуется обойти. Затем мы войдем в цикл, в котором выталкивается и обрабатывается верхняя запись стека, и который продолжается, пока стек не опустеет. Если вытолкнутая запись является элементом, мы посещаем его; если вытолкнутая запись — дерево, мы выполняем последовательность операций выталкивания, которая зависит от требуемого порядка:

- Для *прямого обхода* заталкивается правое поддерево, затем левое поддерево, а затем узел.
- Для *поперечного обхода* заталкивается правое поддерево, затем узел, а затем левое поддерево.
- Для *обратного обхода* заталкивается узел, затем правое поддерево, а затем левое поддерево.

Нулевые деревья в стек не заталкиваются. На рис. 5.27 показано содержимое стека при использовании каждого из этих трех методов для обхода примера дерева, приведенного на рис. 5.26. Методом индукции можно легко убедиться, что для любого бинарного дерева этот метод приводит к такому же результату, как и рекурсивный метод.

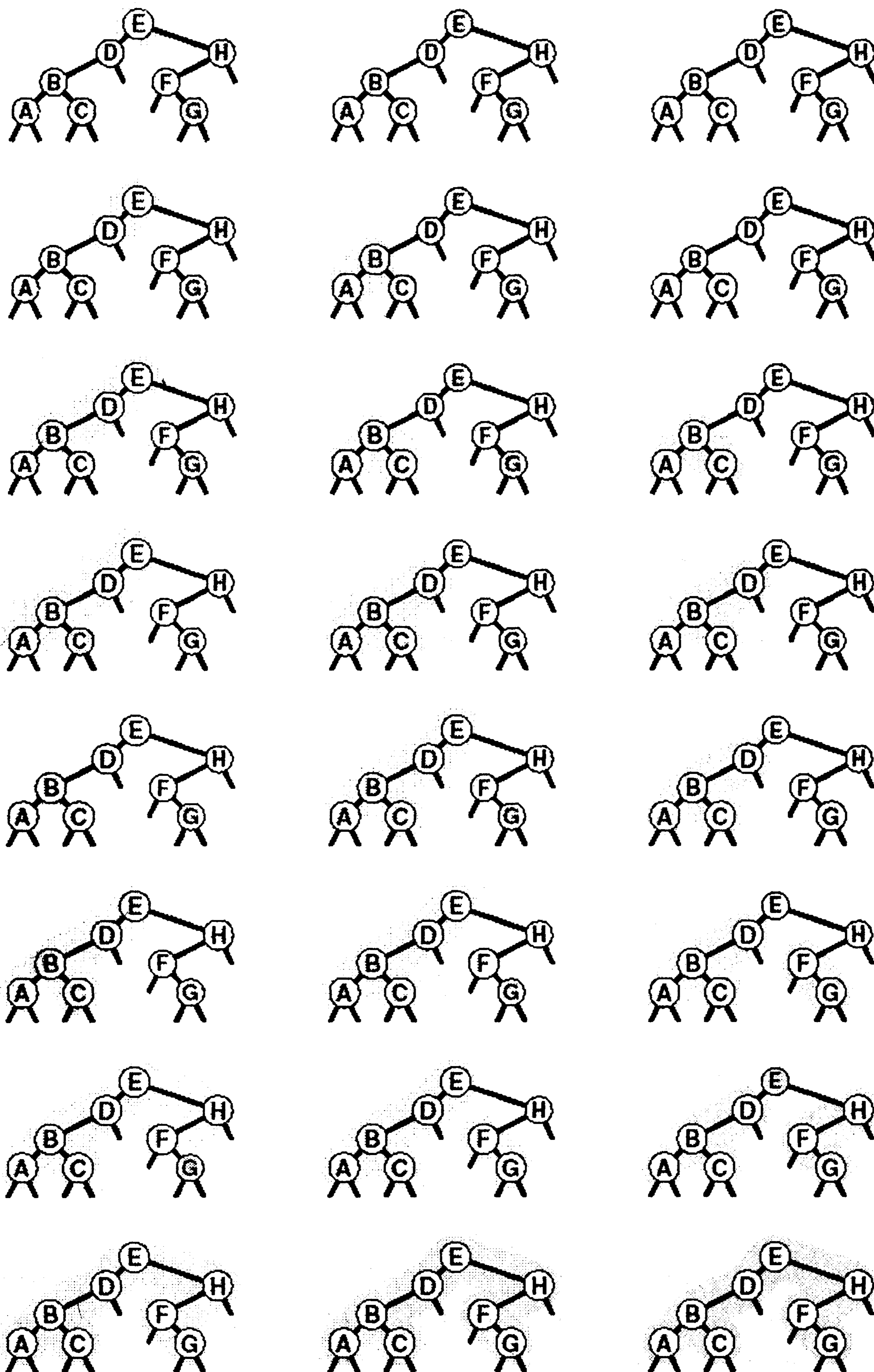
```

traverse E
  visit E
    traverse D
      visit D
        traverse B
          visit B
            traverse A
              visit A
                traverse *
                traverse *
            traverse C
              visit C
                traverse *
                traverse *
            traverse *
            traverse *
        traverse H
          visit H
            traverse F
              visit F
                traverse *
                traverse G
                  visit G
                    traverse *
                    traverse *
                traverse *

```

#### РИСУНОК 5.25 ВЫЗОВЫ ФУНКЦИЙ ПРЯМОГО ОБХОДА

*Эта последовательность вызовов функций определяет прямой обход для примера дерева, показанного на рис. 5.26.*



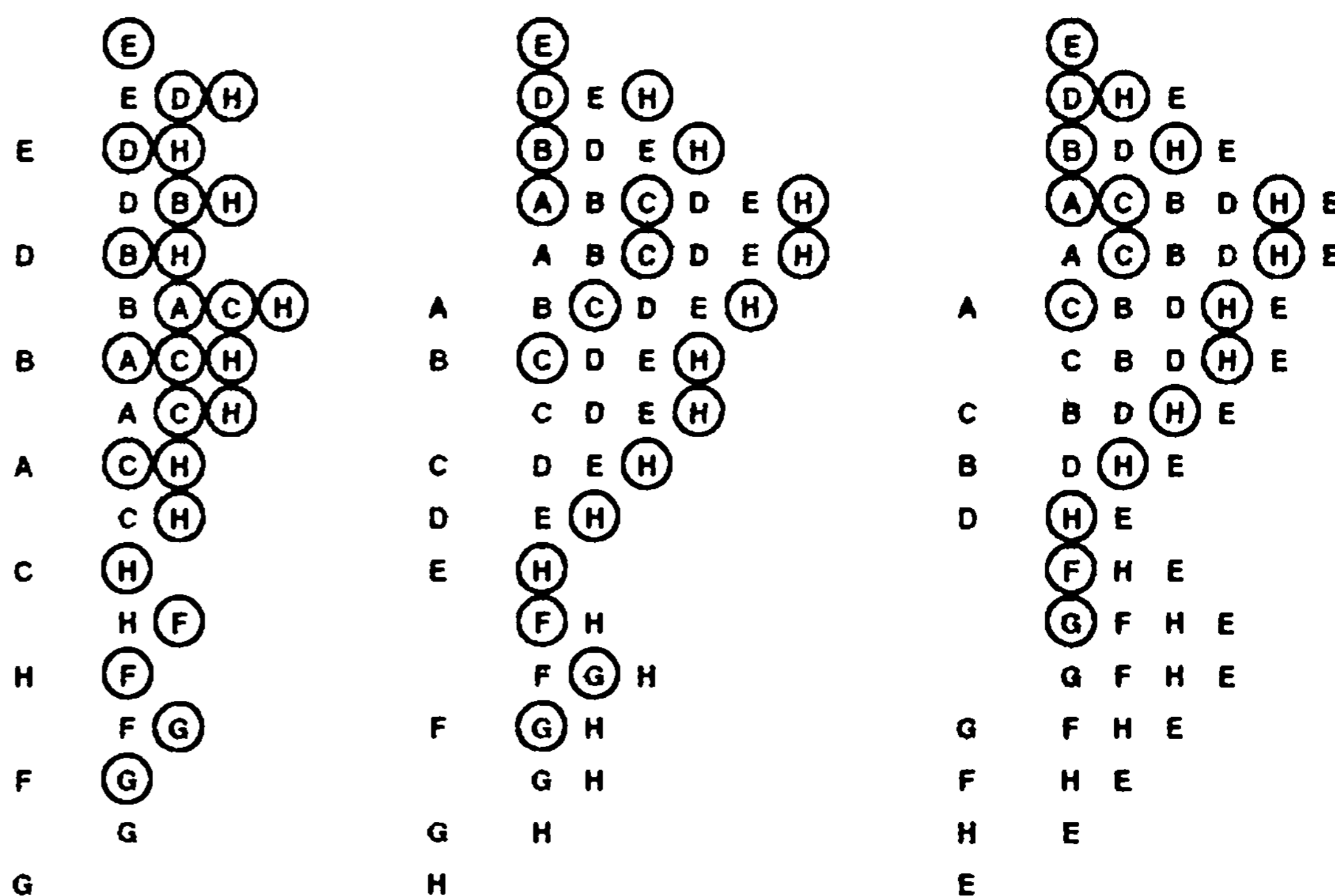
**РИСУНОК 5.26 ПОРЯДКИ ОБХОДА ДЕРЕВА**

*Эти последовательности показывают порядок посещения узлов для прямого (слева), поперечного (в центре) и обратного (справа) обхода дерева.*



### РИСУНОК 5.27 СОДЕРЖИМОЕ СТЕКА ДЛЯ АЛГОРИТМОВ ОБХОДА ДЕРЕВА

Эти последовательности отражают содержимое стека для прямого (слева), поперечного (в центре) и обратного (справа) обхода дерева (см. рис. 5.26) для идеализированной модели вычислений, аналогичной использованной в примере на рис. 5.5, когда элемент и два его поддерева помещаются в стек в указанном порядке.



Описанная в предыдущем абзаце схема является концептуальной, охватывающей три метода обхода дерева, однако реализации, используемые на практике, несколько проще. Например, для выполнения прямого обхода не обязательно заталкивать узлы в стек (мы посещаем корень каждого выталкиваемого дерева), поэтому можно воспользоваться простым стеком, состоящим только из одного типа элементов (связей дерева), как это сделано в нерекурсивной реализации в программе 5.15. Системный стек, поддерживающий рекурсивную программу, содержит адреса возврата и значения аргументов, а не элементы или узлы, но фактическая последовательность выполнения вычислений (посещения узлов) остается одинаковой для рекурсивного метода и метода с использованием стека.

#### Программа 5.15 Прямой обход (нерекурсивная реализация)

Эта нерекурсивная функция с использованием стека — функциональный эквивалент ее рекурсивного аналога, программы 5.14.

```
void traverse(link h, void visit(link))
{ STACK<link> s(max);
  s.push(h);
  while (!s.empty())
  {
    visit(h = s.pop());
    if (h->r != 0) s.push(h->r);
    if (h->l != 0) s.push(h->l);
  }
}
```

Четвертая естественная стратегия обхода — простое посещение узлов дерева в порядке, в котором они отображаются на странице — сверху вниз и слева направо. Этот метод называется обходом *по уровням*, поскольку все узлы каждого уровня посещаются вместе, по порядку. Посещение узлов дерева, показанного на рис. 5.26, при обходе по уровням показано на рис. 5.28.

Как ни удивительно, обход по уровням можно получить, заменив в программе 5.15 стек очередью, что демонстрирует программа 5.16. Для реализации прямого обхода используется структура данных типа "последним вошел, первым вышел" (LIFO); для реализации обхода по уровням используется структура данных типа "первым вошел, первым вышел" (FIFO). Эти программы заслуживают внимательного изучения,

поскольку они представляют существенно различающиеся подходы к организации остающейся невыполненной работы. В частности, обход по уровням не соответствует рекурсивной реализации, связанной с рекурсивной структурой дерева.

**Программа 5.16 Обход по уровням**

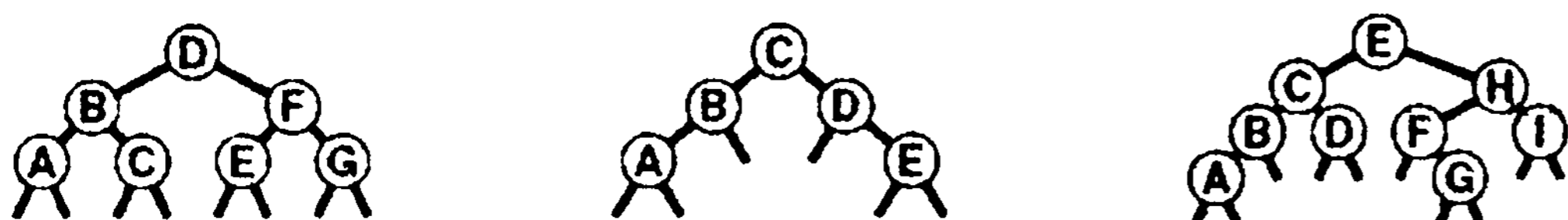
Изменение структуры данных, лежащей в основе прямого обхода (см. программу 5.15) со стека на очередь приводит к преобразованию обхода в обход по уровням.

```
void traverse(link h, void visit(link))
{ QUEUE<link> q(max);
  q.put(h);
  while (!q.empty())
  {
    visit(h = q.get());
    if (h->l != 0) q.put(h->l);
    if (h->r != 0) q.put(h->r);
  }
}
```

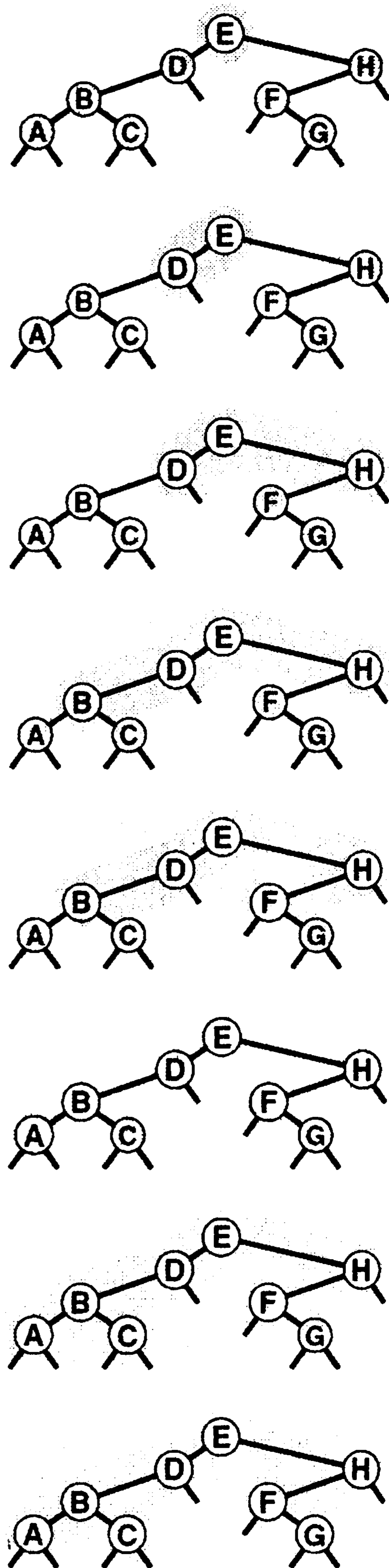
Прямой обход, обратный обход и обход по уровням однозначно определяются и для боров. Чтобы определения были единообразными, представьте себе бор в виде дерева с воображаемым корнем. Тогда правило для прямого обхода формулируется следующим образом: "посетить корень, а затем каждое из поддеревьев"; а правило для обратного обхода — "посетить каждое из поддеревьев, а затем корень". Правило для обхода по уровням то же, что и для бинарных деревьев. Непосредственные реализации этих методов — простые обобщения программ прямого обхода с использованием стека (программы 5.14 и 5.15) и программы обхода по уровням с использованием очереди (программа 5.16) для бинарных деревьев, которые мы только что рассмотрели. Соображения по поводу реализаций не приводятся, поскольку в разделе 5.8 мы рассмотрели более общую процедуру.

**Упражнения**

- ▷ 5.79 Приведите порядок посещения узлов для прямого, поперечного, обратного и обхода по уровням для следующих бинарных деревьев:



- ▷ 5.80 Отрадите содержимое очереди во время обхода по уровням (программа 5.16) на рис. 5.28, аналогично показанному на рис. 5.27.



**РИСУНОК 5.28 ОБХОД ПО УРОВНЯМ**

*Эта последовательность отображает результат посещения узлов дерева в порядке сверху вниз и слева направо.*

**5.81** Покажите, что прямой обход бора равноценен прямому обходу соответствующего бинарного дерева (см. лемму 5.4), а обратный обход бора совпадает с поперечным обходом бинарного дерева.

○ **5.82** Приведите нерекурсивную реализацию поперечного обхода.

● **5.83** Приведите нерекурсивную реализацию обратного обхода.

● **5.84** Создайте программу, которая преобразует прямой и поперечный обходы бинарного дерева в обход дерева по уровням.

## 5.7 Рекурсивные алгоритмы бинарных деревьев

Алгоритмы обхода дерева, рассмотренные в разделе 5.6, наглядно демонстрируют необходимость рассмотрения рекурсивных алгоритмов бинарных деревьев, что обусловлено самой рекурсивной структурой этих деревьев. Для решения многих задач можно непосредственно применять рекурсивные алгоритмы типа "разделяй и властвуй", которые, по существу, обобщают алгоритмы обхода деревьев. Обработка дерева выполняется посредством обработки корневого узла и (рекурсивно) его поддеревьев; вычисление можно выполнять перед, между или после рекурсивных вызовов (или же использовать все три метода).

Часто требуется определять различные структурные параметры дерева, имея только ссылку на дерево. Например, программа 5.17 содержит рекурсивные функции для вычисления количества узлов и высоту заданного дерева. Функции определяются непосредственно исходя из определения 5.6. Ни одна из этих функций не зависит от порядка обработки рекурсивных вызовов: они обрабатывают все узлы дерева и возвращают одинаковый результат, если, например, рекурсивные вызовы поменять местами. Не все параметры дерева вычисляются так легко: например, программа для эффективного вычисления длины внутреннего пути бинарного дерева является более сложной (см. упражнения 5.88—5.90).

### Программа 5.17 Вычисление параметров дерева

Для выяснения базовых структурных свойств дерева можно использовать простые рекурсивные процедуры, подобные следующим.

```
int count(link h)
{
    if (h == 0) return 0;
    return count(h->l) + count(h->r) + 1;
}
int height(link h)
{
    if (h == 0) return -1;
    int u = height(h->l), v = height(h->r);
    if (u > v) return u+1; else return v+1;
}
```

Еще одна функция, которая полезна при создании программ, обрабатывающих деревья, — функция, которая выводит на печать структуру или рисует дерево. Например, программа 5.18 — рекурсивная процедура, выводящая дерево в формате, приведенном на рис. 5.29. Эту же базовую рекурсивную схему можно использовать для

рисования более сложных представлений деревьев, подобным использованным на рисунках в этой книге (см. упражнение 5.85).

### Программа 5.18 Функция быстрого вывода дерева

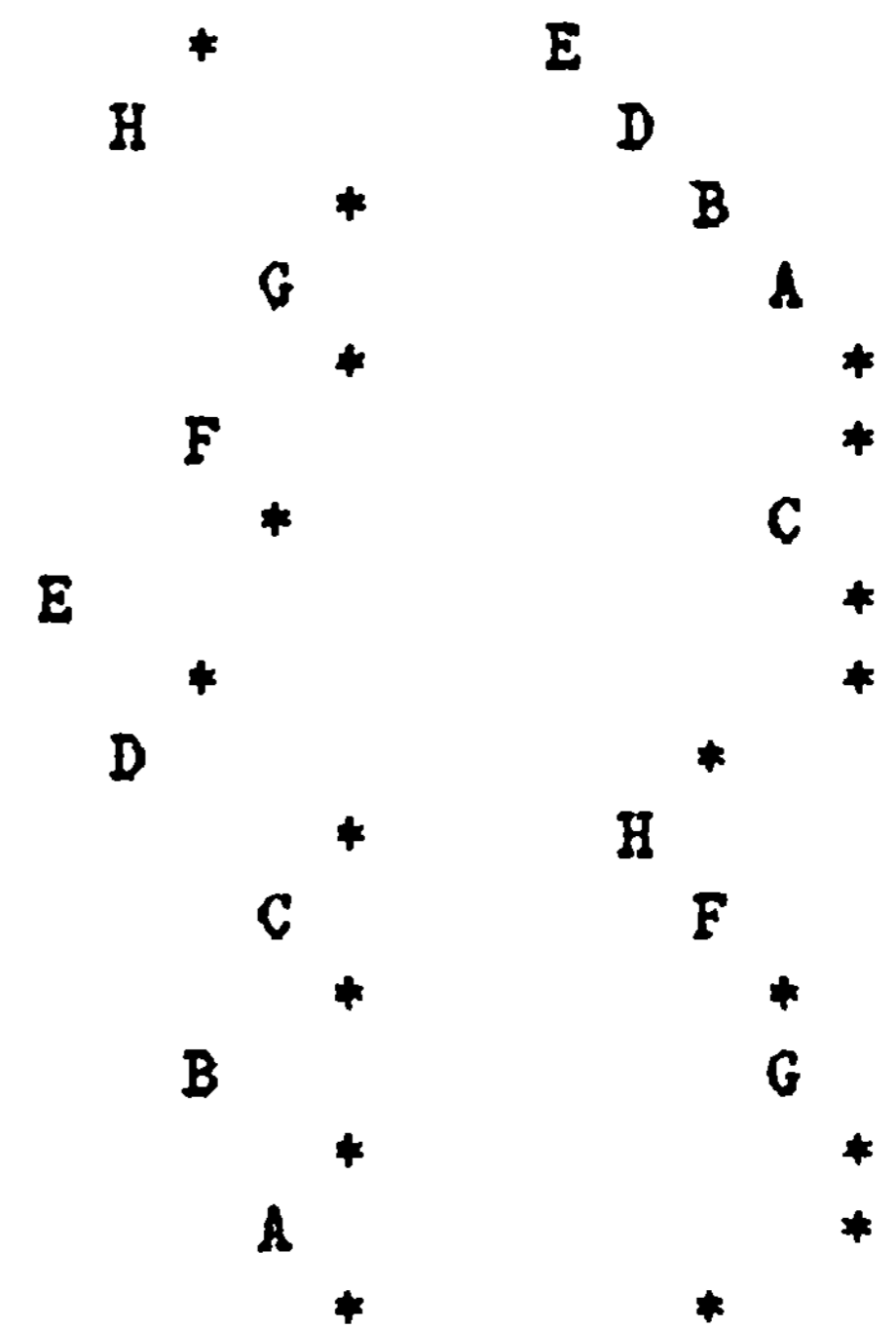
Эта рекурсивная программа отслеживает высоту дерева и использует эту информацию для вывода его представления, которое можно использовать при отладке программ обработки деревьев (см. рис. 5.29). В программе предполагается, что элементы в узлах имеют тип `Item`, для которого определена перегруженная операция `<<`.

```
void printnode(Item x, int h)
{ for (int i = 0; i < h; i++) cout << " ";
  cout << x << endl;
}
void show(link t, int h)
{
  if (t == 0) { printnode('*', h); return;
}
  show(t->r, h+1);
  printnode(t->item, h);
  show(t->l, h+1);
}
```

Программа 5.18 выполняет поперечный обход, но если выводить элемент перед рекурсивными вызовами, получаем прямой обход; этот вариант также приведен на рис. 5.29. Этот формат привычен, например, при отображении генеалогического дерева, списка файлов в файловой структуре в виде дерева или при создании структуры печатного документа. Например, выполнение прямого обхода дерева, отображенного на рис. 5.19, приводит к выводу варианта таблицы оглавления этой книги.

Вначале рассмотрим пример программы, которая строит явную структуру дерева, связанного с приложением определения максимума, которая была рассмотрена в разделе 5.2. Наша цель — построение *турнира* — бинарного дерева, в котором элементом каждого внутреннего узла является копия большего из элементов его двух дочерних элементов. В частности, элемент в корне — копия наибольшего элемента в турнире. Элементы в листьях (узлах, которые не имеют дочерних узлов) образуют представляющие интерес данные, а остальная часть дерева — структура данных, которая позволяет эффективно находить наибольший из элементов.

Программа 5.19 — рекурсивная программа, которая строит турнир из элементов массива. Будучи расширенной версией программы 5.6, она использует стратегию "разделяй и властвуй": чтобы построить турнир для единственного элемента, программа создает лист, содержащий этот элемент, и выполняет возврат. Чтобы построить турнир для  $N > 1$  элементов, в программе используется стратегия "разделяй и властвуй": программа делит все множество элементов пополам, строит турнир для каждой половины и создает новый узел со связями с двумя турнирами и с элементом, который является копией большего элемента в корнях обоих турниров.



**РИСУНОК 5.29 ВЫВОД ДЕРЕВА (ПРИ ПОПЕРЕЧНОМ ОБХОДЕ И ПРЯМОМ ОБХОДЕ)**

*Вывод, приведенный на рисунке слева, получен в результате использования программы 5.18 применительно к примеру дерева, приведенному на рис. 5.26, и он демонстрирует структуру дерева аналогично использованному графическому представлению, повернутому на 90 градусов. Вывод, показанный на рисунке справа, получен в результате выполнения этой же программы при перемещении оператора вывода в начало программы; он демонстрирует структуру дерева в привычном формате.*

### Программа 5.19 Конструирование турнира

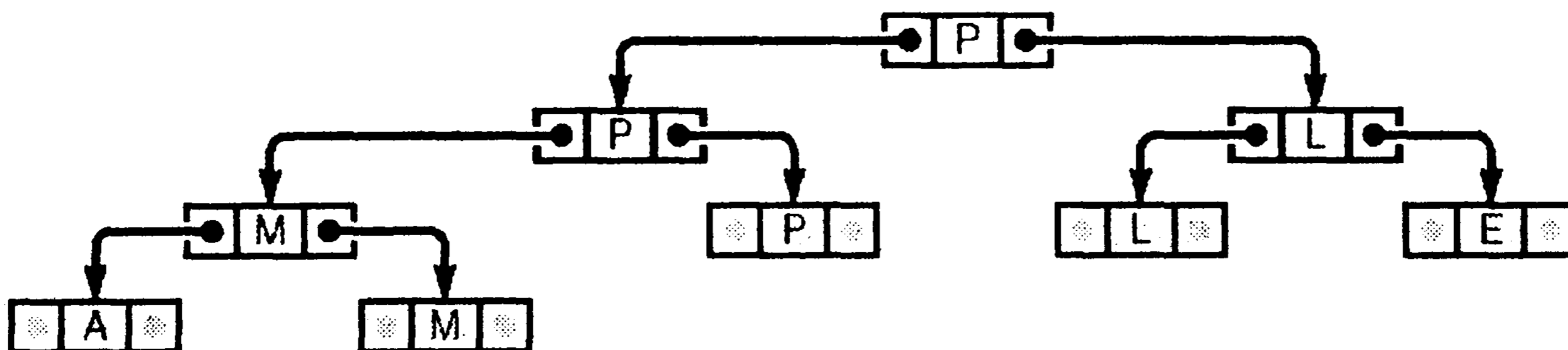
Эта рекурсивная функция делит массив  $a[1], \dots, a[r]$  на две части  $a[1], \dots, a[m]$  и  $a[m+1], \dots, a[r]$ , строит турниры для двух частей (рекурсивно) и создает турнир для всего массива, связывая новый узел с рекурсивно построенными турнирами и помещая в него копию большего элемента в корнях двух рекурсивно построенных турниров.

```

struct node
{ Item item; node *l, *r;
  node(Item x)
  { item = x; l = 0; r = 0; }
};
typedef node* link;
link max(Item a[], int l, int r)
{ int m = (l+r)/2;
  link x = new node(a[m]);
  if (l == r) return x;
  x->l = max(a, l, m);
  x->r = max(a, m+1, r);
  Item u = x->l->item, v = x->r->item;
  if (u > v)
    x->item = u; else x->item = v;
  return x;
}

```

На рис. 5.30 приведен пример явной структуры дерева, построенной программой 5.19. Построение таких рекурсивных структур — вероятно, предпочтительней отысканию максимума путем просмотра данных, как это было сделано в программе 5.6, поскольку структура дерева обеспечивает определенную гибкость для выполнения других операций. Важным примером служит и сама операция, использованная для построения турнира. При наличии двух турниров их можно объединить в один турнир, создав новый узел, левая связь которого указывает на один турнир, а правая на другой, и приняв больший из двух элементов (помещенных в корнях двух данных турниров) в качестве наибольшего элемента объединенного турнира. Можно также рассмотреть алгоритмы добавления и удаления элементов и выполнения других операций. Здесь мы не станем рассматривать такие операции, поскольку аналогичные структуры данных, обладающие подобной гибкостью, исследуются в главе 9.



**РИСУНОК 5.30 ЯВНОЕ ДЕРЕВО ДЛЯ ОТЫСКАНИЯ МАКСИМУМА (ТУРНИР)**

На этом рисунке отображена структура дерева, сконструированная программой 5.19 на основании ввода элементов  $A M P L E$ . Элементы данных помещаются в листьях. Каждый внутренний узел содержит копию большего из элементов в двух дочерних узлах, следовательно, в соответствии с методом индукции наибольшим элементом является корень.

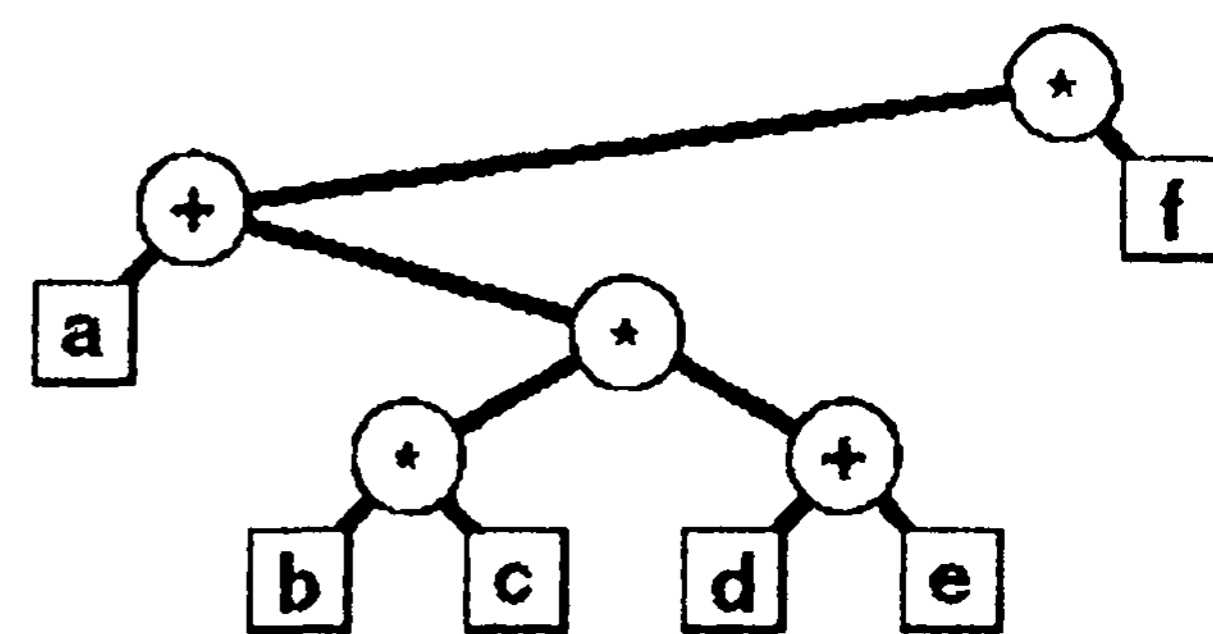
Действительно, реализации с использованием деревьев для нескольких из обобщенных абстрактных типов данных (АТД) запросов, рассмотренных в разделе 4.6, относятся к основной теме большей части этой книги. В частности, многие из алгоритмов, приведенных в главах 12—15, основываются на *деревьях бинарного поиска (binary search tree)*, которые являются явными деревьями, соответствующими бинарному поиску, аналогично тому, как явная структура на рис. 5.30 взаимосвязана с рекурсивным алгоритмом отыскания максимума (см. рис. 5.6). Сложность реализации и использования таких структур заключается в обеспечении эффективности алгоритмов после выполнения большого числа операций *вставки, удаления* и других операций.

Вторым примером программы создания бинарного дерева служит измененная версия программы оценки префиксного выражения, приведенной в разделе 5.1 (программа 5.4), которая создает дерево, представляющее префиксное выражение, а не просто оценивает его (см. рис. 5.31). В программе 5.20 используется та же рекурсивная схема, что и в программе 5.4, но рекурсивная функция возвращает ссылку на дерево, а не значение. Программа создает новый узел дерева для каждого символа в выражении: узлы, которые соответствуют операциям, имеют связи со своими операндами, а узлы листьев содержат переменные (или константы), которые являются входными данными выражения.

### Программа 5.20 Создание дерева синтаксического анализа

Используя ту же стратегию, которая была задействована при оценке префиксных выражений (см. программу 5.4), эта программа создает дерево синтаксического анализа из префиксного выражения. Для простоты предполагается, что операндами являются одиночные символы. Каждый вызов рекурсивной функции создает новый узел, передавая в него в качестве лексемы следующий символ из массива входных данных. Если лексема представляет собой операнд, программа возвращает новый узел, а если операция — то устанавливает левый и правый указатели на дерево, построенное (рекурсивно) для двух аргументов.

```
char *a; int i;
struct node
{ Item item; node *l, *r;
  node(Item x)
  { item = x; l = 0; r = 0; }
};
typedef node* link;
link parse()
{ char t = a[i++]; link x = new node(t);
  if ((t == '+') || (t == '*'))
    { x->l = parse(); x->r = parse(); }
  return x;
}
```



**РИСУНОК 5.31 ДЕРЕВО СИНТАКСИЧЕСКОГО АНАЛИЗА**

*Это дерево создается программой 5.20 для префиксного выражения  $* + a * * b c + d e f$ . Оно представляет собой естественный метод представления выражения: каждый операнд размещается в листе (который отображен в качестве внешнего узла), а каждая операция, которая должна применяться к выражению, представлена левым и правым поддеревьями узла, содержащего операцию.*

В программах трансляции, подобных компиляторам, часто используются такие внутренние представления программ в виде деревьев, поскольку деревья удобны для достижения многих целей. Например, можно представить операнд, соответствующие переменным, принимающим значения, и сгенерировать машинный код для оценки выражения, представленного в виде дерева с обратным обходом. Либо же можно воспользоваться деревом с поперечным обходом для вывода выражения в инфиксной форме или дерево с обратным обходом — для вывода выражения в постфиксной форме.

В этом разделе было рассмотрено несколько примеров, подтверждающих концепцию о возможности создания и обработки структур явных связных деревьев с помощью рекурсивных программ. Чтобы этот подход стал эффективным, потребуется учесть производительность различных алгоритмов, альтернативные представления, нерекурсивные альтернативы и ряд других нюансов. Однако, более подробное рассмотрение программ обработки деревьев откладывается до главы 12, поскольку в главах 7—11 деревья используются, в основном, в описательных целях. К реализациям явных деревьев мы вернемся в главе 12, поскольку они служат основой для многих алгоритмов, исследуемых в главах 12—15.

## Упражнения

- 5.85 Измените программу 5.18, чтобы она выводила PostScript-программу, которая рисует дерево в формате, подобном использованному на рис. 5.23, но без маленьких квадратов, представляющих внешние узлы. Используйте операторы **moveto** и **lineto** для рисования линий и оператор

```
/node {newpath moveto currentpoint 4 0 360 arc fill} def
```

для рисования узлов. После инициализации этого определения вызов **node** приводит к рисованию черной точки в точке с координатами, помещенными в стек (см. раздел 4.3).

- ▷ 5.86 Создайте программу, которая подсчитывает листья в бинарном дереве.
- ▷ 5.87 Создайте программу, которая подсчитывает количество узлов в бинарном дереве с одним внешним и одним внутренним дочерними деревьями.
- ▷ 5.88 Создайте рекурсивную программу, которая вычисляет длину внутреннего пути бинарного дерева, используя определение 5.6.
- 5.89 Определите количество вызовов функций, выполненных программой при вычислении длины внутреннего пути бинарного дерева. Докажите ответ методом индукции.
- 5.90 Создайте рекурсивную программу, которая вычисляет длину внутреннего пути бинарного дерева за время, которое пропорционально количеству узлов в дереве.
- 5.91 Создайте рекурсивную программу, которая удаляет из турнира все листья с заданным ключом (см. упражнение 5.59).

## 5.8 Обход графа

В качестве заключительного примера в этой главе рассмотрим одну из наиболее важных рекурсивных программ: рекурсивный обход графа, или *поиск в глубину* (*depth-first search*). Этот метод симметричного посещения всех узлов графа — непосредственное обобщение методов обхода деревьев, рассмотренных в разделе 5.6, и он служит основой для многих базовых алгоритмов обработки графов (см. часть 7). Это простой рекурсивный алгоритм. Начиная с любого узла  $v$ , мы

- посещаем  $v$ ;
- (рекурсивно) посещаем каждый (*непосещенный*) узел, связанный с  $v$ .

Если граф является связным, со временем будут посещены все узлы. Программа 5.21 демонстрирует реализацию этой рекурсивной процедуры.

### Программа 5.21 Поиск в глубину

Для посещения в графе всех узлов, связанных с узлом  $k$ , мы помечаем его как *посещенный*, а затем (рекурсивно) посещаем все непосещенные узлы в списке смежности для узла  $k$ .

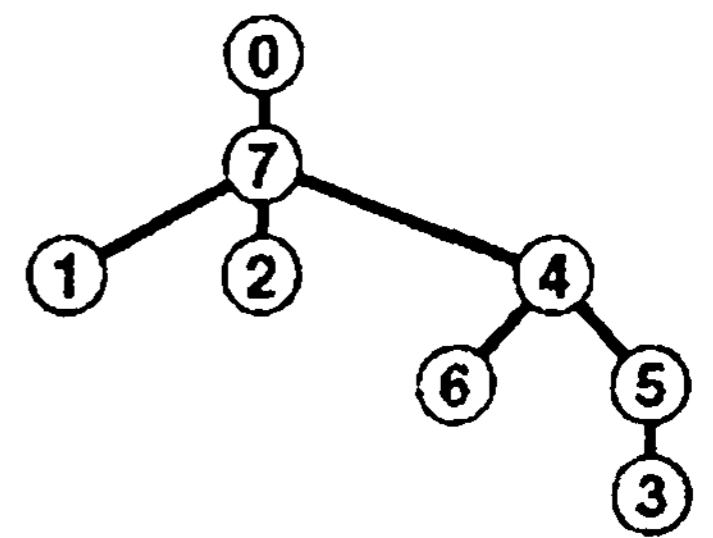
```
void traverse(int k, void visit(int))
{ visit(k); visited[k] = 1;
  for (link t = adj[k]; t != 0; t = t->next)
    if (!visited[t->v]) traverse(t->v, visit);
}
```

Например, предположим, что используется представление в виде списка соседних узлов, описанное в примере графа на рис. 3.15. На рис. 5.32 показаны последовательность вызовов, выполненных при поиске в глубину в этом графе, а последовательность прохождения ребер графа находится в левой части рис. 5.33. При прохождении каждого из ребер графа возможны два исхода: если ребро приводит к уже посещенному узлу, мы игнорируем его; если оно приводит к еще не посещенному узлу, мы переходим к нему через рекурсивный вызов. Набор всех просмотренных таким образом ребер образует остовное дерево графа.

Различие между поиском в глубину и общим обходом дерева (см. программу 5.14) состоит в том, что необходимо явно исключить посещение уже посещенных узлов. В дереве мы никогда не встречаемся с такими узлами. Действительно, если граф является деревом, рекурсивный поиск в глубину, начинающийся с корня, эквивалентен прямому обходу.

**Лемма 5.10** *Время, требующееся для выполнения поиска в глубину в графе с  $V$  вершинами и  $E$  ребрами, пропорционально  $V + E$ , если использовать представление графа в виде списков смежности.*

В представлении в виде списков смежности каждому ребру графа соответствует один узел в списке, а каждой вер-



```

visit 0
  visit 7 (first on 0's list)
    visit 1 (first on 7's list)
      check 7 on 1's list
      check 0 on 1's list
    visit 2 (second on 7's list)
      check 7 on 2's list
      check 0 on 2's list
    check 0 on 7's list
  visit 4 (fourth on 7's list)
    visit 6 (first on 4's list)
      check 4 on 6's list
      check 0 on 6's list
    visit 5 (second on 4's list)
      check 0 on 5's list
      check 4 on 5's list
    visit 3 (third on 5's list)
      check 5 on 3's list
      check 4 on 3's list
    check 7 on 4's list
    check 3 on 4's list
  check 5 on 0's list
  check 2 on 0's list
  check 1 on 0's list
  check 6 on 0's list
  
```

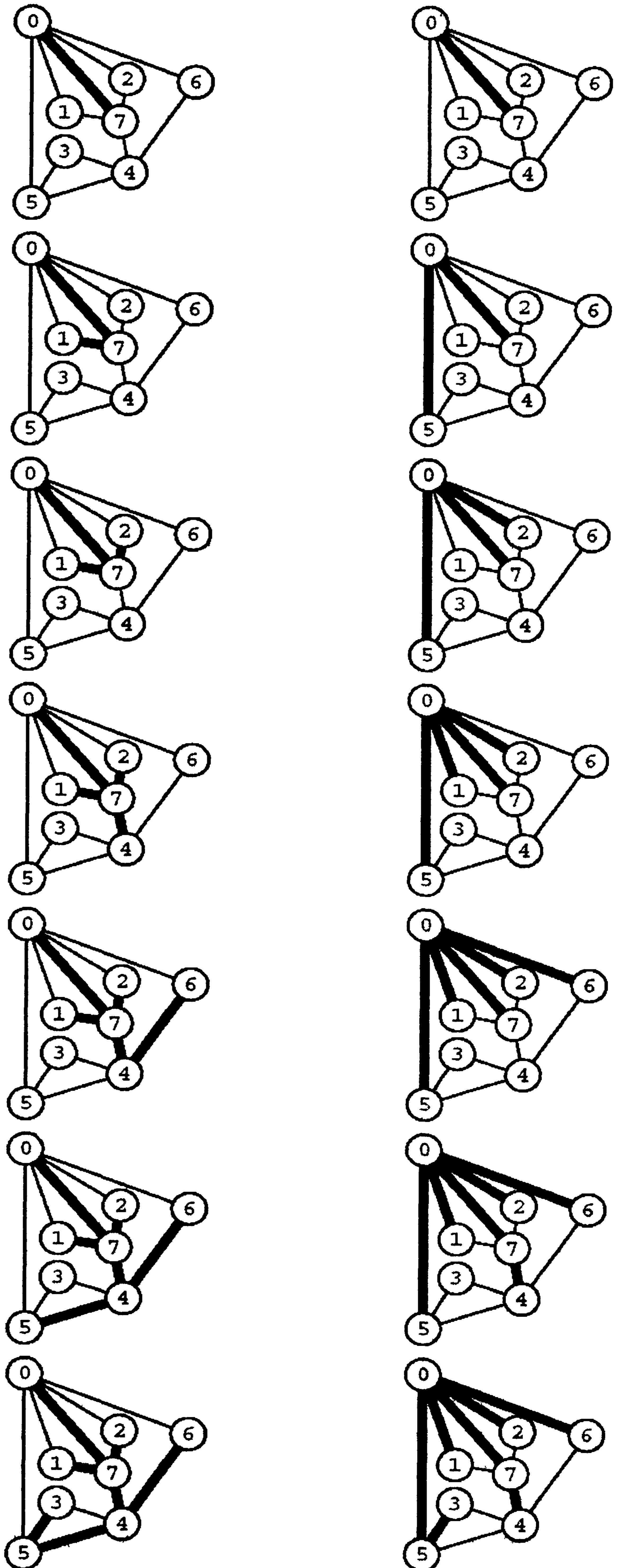
### РИСУНОК 5.32 ВЫЗОВЫ ФУНКЦИЙ ПОИСКА В ГЛУБИНУ

*Эта последовательность вызовов функций реализует поиск в глубину для примера графа, приведенного на рис. 3.15. Дерево, которое описывает структуру рекурсивных вызовов (вверху), называется деревом поиска в глубину.*



**РИСУНОК 5.33 ПОИСК В ГЛУБИНУ  
И ПОИСК В ШИРИНУ**

*При поиске в глубину (слева) переход выполняется от узла к узлу, с возвратом к предшествующему узлу с целью проверки следующей возможности, когда все соседние связи данного узла проверены. При поиске в ширину (справа), до перехода к следующему узлу сначала должны быть перебраны все возможности для данного узла.*



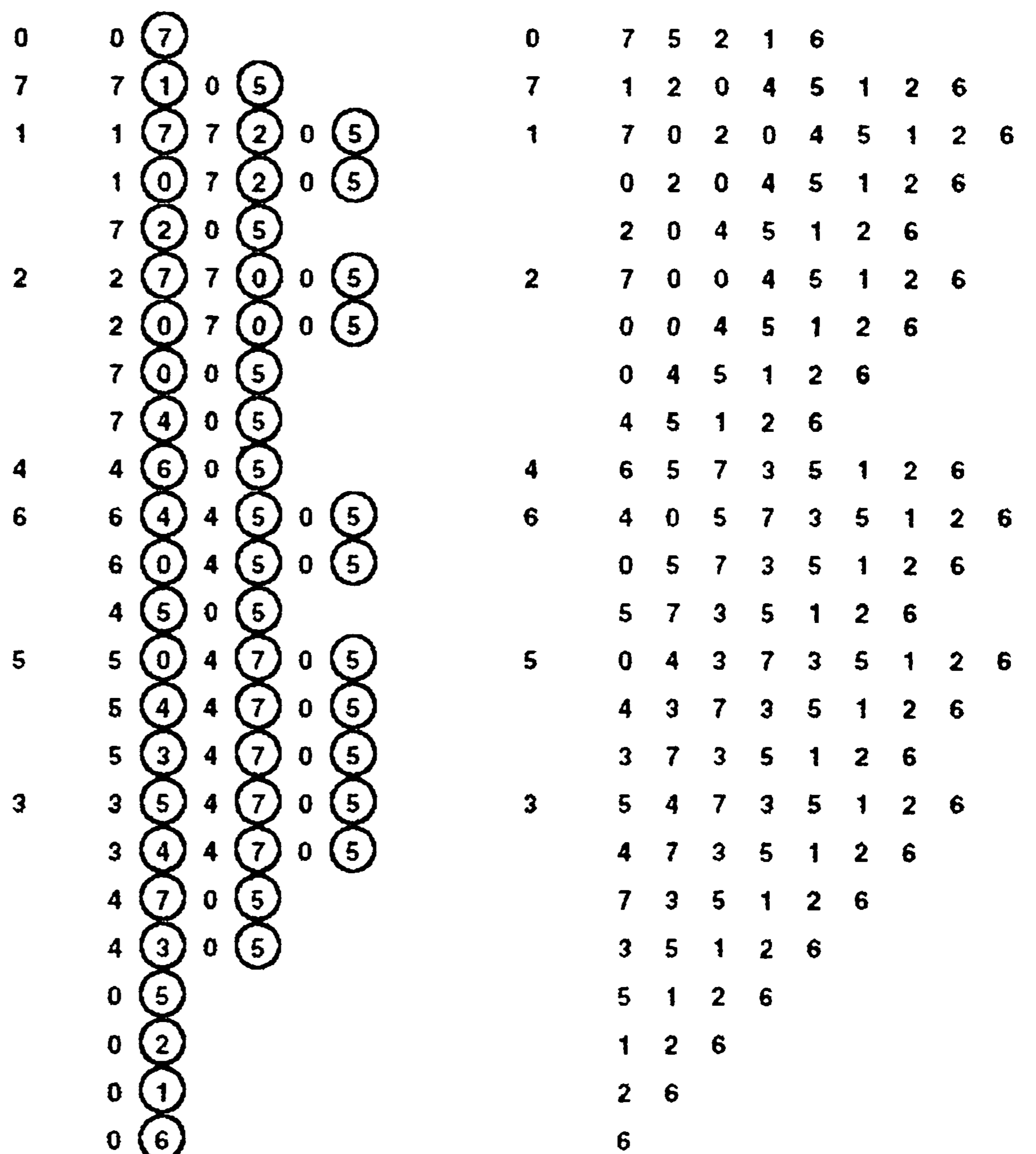
шине графа соответствует один указатель на начало списка. Поиск в глубину затрагивает каждый из них не более одного раза.

Поскольку время, необходимое для построения представления в виде списков смежности из последовательности ребер (см. программу 3.19), также пропорционально  $V + E$ , поиск в глубину обеспечивает линейное по отношению к затрачиваемому времени решение задачи связности из главы 1. Однако для очень больших графов решения union-find все же могут оказаться предпочтительнее, поскольку для представления всего графа требуется объем памяти, пропорциональный  $E$ , в то время как для решений union-find необходим объем памяти, который пропорционален только  $V$ .

Подобно тому как это было сделано в случае с обходом дерева, можно определить метод обхода графа, при котором используется явный стек, как показано на рис. 5.34. Можно представить себе абстрактный стек, содержащий двойные записи: узел и указатель на список смежности для этого узла. Когда стек инициализируется начальным узлом, а указатель — первым элементом списка смежности для этого узла, алгоритм поиска в глубину эквивалентен входу в цикл, в котором сначала посещается узел в верхней части стека (если он еще не был посещен); затем сохраняется узел, указанный текущим указателем списка смежности; далее, ссылка из списка смежности обновляется следующим узлом (с выталкиванием записи, если достигнут конец списка смежности); и, наконец, запись стека для сохраненного узла заталкивается со ссылкой на первый узел его списка смежности.

**РИСУНОК 5.34 ДИНАМИКА СТЕКА ПОИСКА В ГЛУБИНУ**

*Стек, поддерживающий поиск в глубину, можно представить себе как содержащий узел и ссылку на список смежности для этого узла (он показан узлом, заключенным в окружность) (слева). Таким образом, обработка в стеке начинается с узла 0, т.е. со ссылки на первый узел в его списке смежности — узел 7. Каждая строка отражает результат выталкивания из стека, занесения ссылки на следующий узел в списке посещенных узлов и занесения в стек записи для непосещенных узлов. Процесс также можно представить в виде простого заталкивания в стек всех узлов, смежных с любым непосещенным узлом (справа).*



Или же, подобно тому, как это было сделано для обхода дерева, можно создать стек, содержащий только ссылки на узлы. Когда стек инициализируется начальным узлом, мы входим в цикл, в котором посещаем узел в верхней части стека (если он еще не был посещен), затем заталкиваем в стек все соседние узлы этого узла. Рисунок 5.34 иллюстрирует, что для нашего примера графа оба метода эквивалентны поиску в глубину, причем эквивалентность сохраняется и в общем случае.

Алгоритм посещения верхнего узла и заталкивания всех его соседей — простая формулировка поиска в глубину, но из рис. 5.34 понятно, что этому методу присущ недостаток возможного оставления в стеке нескольких копий каждого узла. Это происходит даже в случае проверки, посещался ли каждый узел, который должен быть помещен в стек, и если да, то отказа от занесения в стек такого узла. Во избежание упомянутой проблемы можно воспользоваться реализацией стека, которая запрещает дублирование за счет применения стратегии "забывания старого элемента": поскольку ближайшая к вершущке стека копия всегда посещается первой, все остальные копии просто выталкиваются из стека.

Динамика состояния стека для поиска в глубину, показанная на рис. 5.34, основана на том, что узлы каждого списка соседних узлов появляются в стеке в том же порядке, что и в списке. Для получения этого порядка для данного списка смежности при заталкивании узлов по одному, вначале следовало бы затолкнуть в стек последний узел, затем предпоследний и т.д. Более того, чтобы ограничить размер стека числом вершин при одновременном посещении узлов в том же порядке, как и при поиске в глубину, необходимо использовать стек, в котором реализуется стратегия "забывания старого элемента". Если посещение узлов в том же порядке, что и при поиске в глубину, не имеет значения, обоих этих осложнений можно избежать и непосредственно сформулировать нерекурсивный метод обхода графа с использованием стека, который выглядит так. После инициализации стека начальным узлом мы входим в цикл, в котором посещаем узел на вершущке стека, затем обрабатываем его список смежности, помещая в стек каждый узел (если он еще не был посещен) и используя реализацию стека, которая за счет применения стратегии "игнорирования нового элемента" запрещает дублирование. Этот алгоритм обеспечивает посещение всех узлов графа, подобно поиску в глубину, но не является рекурсивным.

Алгоритм, описанный в предыдущем абзаце, заслуживает внимания, поскольку можно было бы воспользоваться любым обобщенным абстрактным типом данных очереди и все же посетить каждый из узлов графа (плюс сгенерировать развернутое дерево). Например, если вместо стека задействовать очередь, то получится *поиск в ширину*, который аналогичен обходу дерева по уровням. Программа 5.22 — реализация этого метода (при условии, что используется реализация с применением очереди, подобная программе 4.12); пример этого алгоритма в действии показан на рис. 5.35. В части 6 будет исследоваться множество алгоритмов обработки графов, основанных на более сложных обобщенных абстрактных типах данных очереди.

**РИСУНОК 5.35 ДИНАМИКА ОЧЕРЕДИ ПОИСКА В ШИРИНУ**

*Обработка начинается с узла 0 в очереди, затем мы получаем узел 0, посещаем его и помещаем в очередь узлы 7 5 2 1 6 из его списка смежности, причем именно в этом порядке. Затем мы получаем узел 7, посещаем его и помещаем в очередь узлы из его списка смежности, и т.д. В случае запрета дублирования через стратегию "игнорирования нового элемента" (справа) мы получаем такой же результат без наличия каких-либо лишних записей очереди.*

	0		0
0	7 5 2 1 6	0	7 5 2 1 6
7	5 2 1 6 1 2 4	7	5 2 1 6 4
5	2 1 6 1 2 4 4 3	5	2 1 6 4 3
2	1 6 1 2 4 4 3	2	1 6 4 3
1	6 1 2 4 4 3	1	6 4 3
6	1 2 4 4 3 4	6	4 3
	2 4 4 3 4	4	3
	4 4 3 4	3	
4	4 3 4 3		
	3 4 3		
3	4 3		
	3		

**Программа 5.22 Поиск в ширину**

Чтобы посетить в графе все узлы, связанные с узлом  $k$ ,  $k$  помещается в очередь FIFO, затем выполняется цикл, в котором из очереди получается следующий узел  $i$ , если он еще не был посещен, он посещается и в стек заталкиваются все непосещенные узлы из его списка смежности; процесс продолжается до тех пор, пока очередь не опустеет.

```
void traverse(int k, void visit(int))
{
    QUEUE<int> q(V*V);
    q.put(k);
    while (!q.empty())
        if (visited[k = q.get()] == 0)
            {
                visit(k); visited[k] = 1;
                for (link t = adj[k]; t != 0; t = t->next)
                    if (visited[t->v] == 0) q.put(t->v);
            }
}
```

При использовании обоих методов поиска в ширину и в глубину посещаются все узлы графа, но их способ достижения этого различается коренным образом, что демонстрирует рис. 5.36. Поиск в ширину подобен армии исследователей, разосланной во все стороны с целью охвата всей территории; поиск в глубину соответствует единственному исследователю, который как можно дальше проникает вглубь неизведанной территории, возвращаясь только в случае, если наталкивается на тупик. Таковы базовые методы решения задач, играющие существенную роль во многих областях компьютерных наук, а не только в контексте поиска в графах.

**Упражнения**

**5.92** Принимая во внимание диаграммы, соответствующие рис. 5.33 (слева) и 5.34 (справа), покажите, как происходит посещение узлов в графе, построенном для последовательности ребер 0-2, 1-4, 2-5, 3-6, 0-4, 6-0 и 1-3 (см. упражнение 3.70), при рекурсивном поиске в глубину.

**5.93** Принимая во внимание диаграммы, соответствующие рис. 5.33 (слева) и 5.34 (справа), покажите, как происходит посещение узлов в графе, построенном для

последовательности ребер 0-2, 1-4, 2-5, 3-6, 0-4, 6-0 и 1-3 (см. упражнение 3.70), при поиске в глубину с использованием стека.

**5.94** Принимая во внимание диаграммы, соответствующие рис. 5.33 (слева) и 5.35 (справа), покажите, как происходит посещение узлов в графе, построенном для последовательности ребер 0-2, 1-4, 2-5, 3-6, 0-4, 6-0 и 1-3 (см. упражнение 3.70), при поиске в ширину (с использованием очереди).

- **5.95** Почему время выполнения, упоминаемое в лемме 5.10, пропорционально  $V + E$ , а не просто  $E$ ?

**5.96** Принимая во внимание диаграммы, соответствующие рис. 5.33 (слева) и 5.35 (справа), покажите, как происходит посещение узлов в примере графа, приведенном в тексте (рис.3.15), при поиске в глубину с использованием стека с применением стратегии "забывания старого элемента".

**5.97** Принимая во внимание диаграммы, соответствующие рис. 5.33 (слева) и 5.35 (справа), покажите, как происходит посещение узлов в примере графа, приведенном в тексте (рис.3.15), при поиске в глубину с использованием стека с применением стратегии "игнорирования нового элемента".

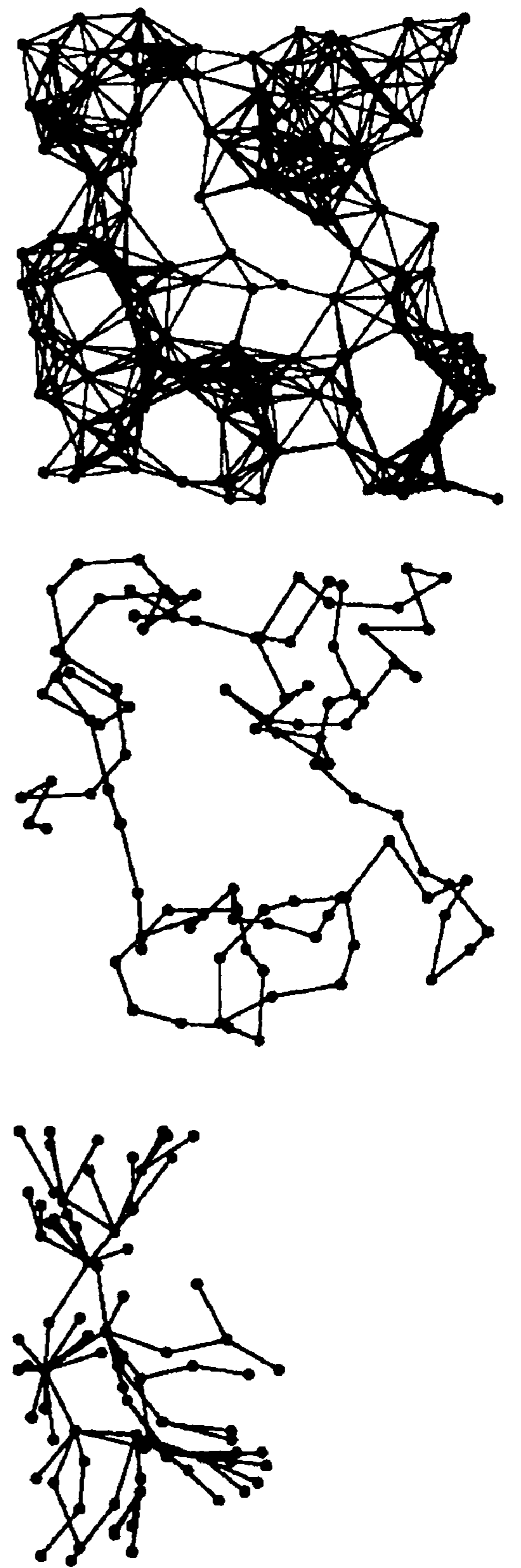
- ▷ **5.98** Реализуйте поиск в глубину с использованием стека для графов, которые представлены списками смежности.

- **5.99** Реализуйте рекурсивный поиск в глубину для графов, которые представлены списками смежности.

## 5.9 Перспективы

Рекурсия лежит в основе ранних теоретических исследований природы вычислений. Рекурсивные функции и программы играют главную роль в математических исследованиях, в которых предпринимается попытка разделения задач на поддающиеся решению на компьютере и на непригодные для этого.

В ходе столь краткого рассмотрения просто невозможно полностью осветить столь обширные темы, как деревья и рекурсия. Многие наиболее показательные примеры рекурсивных программ будут пребывать в центре внимания на протяжении всей книги — к ним относятся алгоритмы типа "разделяй и властвуй" и рекурсивные структуры данных, которые успешно применяются для решения широкого спектра задач. Для многих приложений нет смысла вы-



**РИСУНОК 5.36 ДЕРЕВЬЯ ОБХОДА ГРАФОВ**

На этой схеме показаны поиск в глубину (в центре) и поиск в ширину (внизу), выполненные на половину в большом графе (вверху). При поиске в глубину обход выполняется от одного узла к следующему, так что большинство узлов связано только с двумя другими. И напротив, при поиске в ширину перемещение по графу выполняется по всей области графа, с посещением всех узлов, связанных с данным, прежде чем двигаться дальше; поэтому некоторые узлы связаны со множеством других.

ходить за рамки простой непосредственной рекурсивной реализации; для других будут рассматриваться методы, производные от альтернативных нерекурсивных и восходящих реализаций.

В этой книге основное внимание уделено практическим аспектам построения рекурсивных программ и структур данных. Наша цель — применение рекурсии для создания изящных и эффективных реализаций. Для достижения этой цели необходимо особо учитывать опасности, сопряженные с использованием простых программ, которое ведет к экспоненциальному увеличению количества вызовов функций или недопустимо большой степени вложенности. Несмотря на этот недостаток, рекурсивные программы и структуры данных весьма привлекательны, поскольку часто они предоставляют индуктивные аргументы, которые помогают убедиться в правильности и эффективности разработанных программ.

На протяжении всей книги деревья используются как для упрощения понимания динамических свойств программ, так и в качестве динамических структур данных. В главах с 12 по 15 особенно большое внимание уделяется манипулированию явными древовидными структурами. Свойства, описанные в этой главе, предоставляют основную информацию, которая требуется для эффективного применения явных древовидных структур.

Несмотря на свою центральную роль в разработке алгоритмов, рекурсия — вовсе не панацея на все случаи жизни. Как было показано при исследовании алгоритмов обхода деревьев и графов, алгоритмы с использованием стека (которые рекурсивны по своей природе) — не единственная возможность при необходимости управлять задачами с множеством вычислений. Эффективная технология разработки алгоритмов для решения многих задач заключается в использовании обобщенных реализаций с применением очередей, отличающихся от стеков, которые предоставляют свободу выбирать следующую задачу в соответствии с каким-либо более субъективным критерием, нежели простой выбор чего-либо последнего. Структуры данных и алгоритмы, которые эффективно поддерживают такие операции — основная тема главы 9, а со многими примерами их применения мы встретимся во время исследования алгоритмов обработки графов в части 7.

## Ссылки для части 2

Существует множество учебников для начинающих, посвященных структурам данных. Например, в книге Стендиша (Standish) темы связанных структур, абстракций данных, стеков и очередей, распределения памяти и создания программ освещаются более подробно, чем здесь. Конечно, классические книги Кернигана и Ритчи (Kernighan-Ritchie) и Страуструпа (Stroustrup) — бесценные источники подробной информации по реализациям C и C++. Книги Мейерса (Meyers) также содержат полезную информацию о реализациях C++.

Разработчики PostScript, вероятно, не могли даже и предполагать, что разработанный ими язык будет представлять интерес для людей, которые изучают основы алгоритмов и структур данных. Сам по себе этот язык не представляет особой сложности, а справочные руководства по нему — основательны и доступны.

Парадигма реализации типа интерфейс-клиент подробно, со множеством примеров описывается в книге Хансона (Hanson). Эта книга — замечательный справочник для тех программистов, которые намерены создавать устойчивый и переносимый код для больших систем.

Книги Кнута (Knuth), в особенности 1-й и 3-й тома, остаются авторитетным источником информации по свойствам элементарных структур данных. Книги Баеца-Йатса (Baeza-Yates) и Гоннета (Gonnet) содержат более современную информацию, подкрепленную внушительным библиографическим перечнем. Седжвик (Sedgewick) и Флажолет (Flajolet) подробно освещают математические свойства деревьев.

Adobe Systems Incorporated, *PostScript Language Reference Manual*, second edition, Addison-Wesley, Reading, MA, 1990.

R. Baeza-Yates and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.

D. R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*, Addison-Wesley, 1997.

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second edition, Prentice-Hall, Englewood Cliffs, NJ, 1988.

D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, third edition, Addison-Wesley, Reading, MA, 1997; *Volume 2: Seminumerical Algorithms*, third edition, Addison-Wesley, Reading, MA, 1998; *Volume 3: Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1998.

S. Meyers, *Effective C++*, second edition, Addison-Wesley, Reading, MA, 1996.

S. Meyers, *More Effective C++*, Addison-Wesley, Reading, MA, 1996.

R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.

T. A. Standish, *Data Structures, Algorithms, and Software Principles in C*, Addison-Wesley, 1995.

B. Stroustrup, *The C++ Programming Language*, third edition, Addison-Wesley, Reading MA, 1997.

# Сортировка

*В этой части:*

- 6**    **Элементарные методы сортировки**
- 7**    **Быстрая сортировка**
- 8**    **Слияние и сортировка слиянием**
- 9**    **Очереди по приоритетам и пирамидальная сортировка**
- 10**   **Поразрядная сортировка**
- 11**   **Методы сортировки специального назначения**



## Элементарные методы сортировки

**В** качестве нашего первого экскурса в область алгоритмов сортировки изучим несколько элементарных методов, которые целесообразно использовать для сортировки файлов небольших размеров либо для сортировки файлов со специальной структурой. Имеются несколько причин для подробного изучения этих простых алгоритмов сортировки. Прежде всего, они представляют собой контекст, в рамках которого можно изучить терминологию и базовые механизмы алгоритмов сортировки, что позволит создать соответствующие предпосылки для изучения более сложных алгоритмов. Во-вторых, эти простые методы во многих приложениях сортировки показали себя, по сути дела, более эффективными, чем мощные универсальные методы. В-третьих, некоторые из простых методов допускают расширение в более эффективные универсальные методы или же могут оказаться полезными в плане повышения эффективности совершенных методов сортировки.

Цель настоящей главы заключается не только в ознакомлении читателя с элементарными методами сортировки, но и в создании таких структур, в рамках которых можно было бы изучать сортировку и в последующих главах. Мы рассмотрим различные ситуации, которые благоприятствуют применению того или иного алгоритма сортировки, исследуем различные виды входных файлов, а также рассмотрим различные способы сравнения методов сортировки и изучения их свойств.

Начнем с рассмотрения простой программы тестирования методов сортировки, которая обеспечивает контекст, позволяющий выработать соглашения, необходимые для того, чтобы впоследствии им следовать. Мы также про-

анализируем базовые свойства различных методов сортировки; их очень важно знать при оценке возможности использования того или иного алгоритма в условиях конкретных приложений. Затем мы подробно рассмотрим реализацию трех элементарных методов: сортировки выбором, сортировки вставками и пузырьковой сортировки. После этого будут подробно анализироваться рабочие характеристики упомянутых алгоритмов. Далее мы рассмотрим сортировку методом Шелла, которой не очень-то подходит характеристика "элементарная", однако она достаточно просто реализуется и имеет много общего с сортировкой методом вставки. Углубляясь в математические свойства сортировки методом Шелла, мы займемся изучением проблемы разработки интерфейсов типов данных, наряду с программами, которые рассматривались в главах 3 и 4 и которые предназначены для распространения изучаемых алгоритмов на сортировку различных видов файлов данных, какие встречаются на практике. Затем мы рассмотрим методы сортировки по косвенным ссылкам на данные, а также методы сортировки связных списков. Данная глава завершается обсуждением специализированного метода, который целесообразно использовать, когда известно, что ключи принимают значения в ограниченном диапазоне.

Многие из возможных приложений сортировки часто отдают предпочтение простейшим алгоритмам. Во-первых, очень часто программа сортировки используется всего лишь один или небольшое число раз. После того как удалось "решить" проблему сортировки для некоторого набора данных, в дальнейшем потребность в программе сортировки в приложениях, которые манипулируют этими данными, отпадает. Если элементарная сортировка работает не медленнее других частей приложения, осуществляющего обработку данных — например, считывание данных или их вывод на печать — то отпадает необходимость в поиске более быстрых методов сортировки. Если число сортируемых элементов не очень большое (скажем, не превышает нескольких сотен элементов), можно просто воспользоваться простым методом и не ломать голову над тем, как работает интерфейс для системной сортировки, или как написать и отладить программу, реализующую какой-нибудь сложный метод сортировки. Во-вторых, элементарные методы всегда годятся для файлов небольших размеров (состоящих из нескольких десятков элементов) — сложные алгоритмы в общем случае обуславливают непроизводительные затраты ресурсов, а это приводит к тому, что на файлах небольших размеров они работают медленнее элементарных методов сортировки. Эта проблема не попадет в фокус нашего внимания до тех пор, пока не возникнет необходимость сортировки большого числа файлов небольших размеров, однако следует иметь в виду, что приложения с подобными требованиями встречаются достаточно часто. Другими типами файлов, сортировка которых существенно упрощена, являются файлы, с почти (или уже) завершенной сортировкой или файлы, которые содержат большое число дублированных ключей. Далее можно будет убедиться в том, что некоторые методы из числа простейших особенно эффективны при сортировке хорошо структурированных файлов.

Как правило, для сортировки случайно упорядоченного файла из  $N$  элементов с применением элементарных методов, которые обсуждаются в данной главе, требуется время, пропорциональное  $N^2$ . Если  $N$  не велико, то такое время выполнения сортировки может оказаться вполне приемлемым. Как только что было отмечено, эти методы, примененные для сортировки файлов небольших размеров, а также в ряде дру-

гих специальных случаев, по эффективности могут превзойти более сложные методы сортировки. Однако методы, которые исследуются в настоящей главе, *не* годятся для сортировки файлов больших размеров с произвольной организацией, поскольку время их сортировки будет недопустимо большим, даже если она выполняется на сверхбыстродействующих компьютерах. В этом плане заслуживающим внимания исключением может послужить сортировка методом Шелла (см. раздел 6.6), для которого при большом  $N$  требуется гораздо меньше, чем  $N^2$  шагов, при этом можно утверждать, что данный метод является одним из наилучших для сортировки файлов средних размеров и для ряда других специальных случаев.

## 6.1. Правила игры

Прежде чем переходить к рассмотрению конкретных алгоритмов, полезно обсудить общую терминологию и базовые принципы построения алгоритмов сортировки. Мы будем рассматривать методы сортировки *файлов элементов*, обладающих *ключами*. Эти понятия являются естественными абстракциями для современных сред программирования. Ключи, которые — суть лишь часть (зачастую очень небольшая часть) элементов, используются для управления сортировкой. Цель метода сортировки заключается в переупорядочении элементов таким образом, чтобы их ключи следовали в соответствии с четко определенными правилами (обычно это цифровой или алфавитный порядок). Специфические характеристики ключей и элементов в разных приложениях могут существенно отличаться друг от друга, однако абстрактное понятие размещения ключей и связанной с ними информации в определенном порядке и представляет собой суть проблемы сортировки.

Если сортируемый файл полностью помещается в оперативной памяти, то используемый в этом случае метод сортировки называется *внутренним*. Сортировка файлов, хранящихся на магнитной ленте или диске, называется *внешней*. Основное различие между этими двумя методами заключается в том, что в условиях внутренней сортировки доступ к любому элементу не представляет трудностей, в то время как в условиях внешней сортировки возможен только последовательный метод доступа или, по меньшей мере, доступ к блокам больших размеров. Некоторые из внешних методов сортировки рассматриваются в главе 11, однако большая часть исследуемых алгоритмов принадлежит к категории внутренней сортировки.

Мы будем рассматривать как массивы, так и связные списки. Как проблема сортировки массивов, так и проблема сортировки связных списков представляют несомненный интерес: в процессе разработки собственных алгоритмов мы столкнемся с некоторыми базовыми задачами, для которых лучше всего подойдет последовательное распределение элементов, для других же задач больше подходит структура связных списков. Некоторые из классических методов обладают столь высокой степенью абстракции, что могут одинаково эффективно применяться как к массивам, так и к связным спискам; в то же время другие максимально эффективно проявляют себя на каком-то одном виде указанных выше объектов сортировки. Другие виды ограничений доступа также иногда представляют определенный интерес.

Для начала стоит акцентировать внимание на сортировке массивов. Программа 6.1 может служить иллюстрацией соглашений, которые будут соблюдаться во всех реали-

зациях. Она состоит из управляющей программы (в дальнейшем программы-драйвера), которая заполняет массив, считывая целые значения из стандартного ввода либо генерируя случайные целые значения (соответствующий режим работы программы задается целым аргументом). Затем эта программа вызывает функцию сортировки, которая размещает эти целые значения в определенном порядке, а в завершение выполняет печать результатов сортировки.

### Программа 6.1. Пример сортировки массива с помощью управляющей программы

Данная программа служит иллюстрацией принятых нами соглашений, касающихся реализации базовых алгоритмов сортировки массивов. Функция **main** — суть драйвер, который инициализирует массив целыми значениями (случайными значениями либо значениями, полученными из стандартного ввода), вызывает функцию **sort** с целью выполнения сортировки заполненного массива, после чего выводит на печать упорядоченный результат сортировки.

Шаблоны позволяют использовать данную программную реализацию для сортировки элементов, принадлежащих к различным типам данных, для которых определены операции сравнения и присваивания. В рассматриваемом случае функция **sort** представляет собой частный случай сортировки вставками (см. раздел 6.3, в котором приводится подробное описание, пример и улучшенный вариант реализации). Она использует шаблонную функцию, которая сравнивает два элемента и при необходимости производит обмен их местами, чтобы второй элемент был не меньше первого.

Можно внести соответствующие изменения в программу-драйвер, чтобы она смогла выполнять сортировку любых типов данных, для которых определена операция **operator<**, не внося при этом никаких изменений в функцию **sort** (см. раздел 6.7).

```
#include <iostream.h>
#include <stdlib.h>
template <class Item>
    void exch(Item &A, Item &B)
    { Item t = A; A = B; B = t; }
template <class Item>
    void compexch(Item &A, Item &B)
    { if (B < A) exch(A, B); }
template <class Item>
    void sort(Item a[], int l, int r)
    { for (int i = l+1; i <= r; i++)
        for (int j = i; j > l; j--)
            compexch(a[j-1], a[j]);
    }
int main(int argc, char *argv[])
{ int i, N = atoi(argv[1]), sw = atoi(argv[2]);
  int *a = new int[N];
  if (sw)
    for (i = 0; i < N; i++)
        a[i] = 1000*(1.0*rand()/RAND_MAX);
  else
    { N = 0; while (cin >> a[N]) N++; }
  sort(a, 0, N-1);
  for (i = 0; i < N; i++) cout << a[i] << " ";
  cout << endl;
}
```

Как уже должно быть известно из глав 3 и 4, существуют многочисленные механизмы, обеспечивающие условия для выполнения сортировки и других типов данных. Мы подробно обсудим возможности использования таких механизмов в разделе 6.7. Функция `sort` из программы 6.1 представляет собой приведенную к шаблону реализацию, ссылающуюся на сортируемые элементы только посредством своего первого аргумента и нескольких простейших операций над данными. Как обычно, подобный подход позволяет использовать одни и те же программные коды для сортировки элементов разных типов. Например, если в программный код функции `main` программы 6.1, выполняющий генерацию, хранение и печать случайных ключей внести изменения, обеспечивающие возможность обработки чисел с плавающей точкой вместо целых чисел, то вообще отпадет необходимость какой-либо модификации функции `sort`. С целью достижения упомянутой гибкости (и в то же время явной идентификации переменных, в которых хранятся элементы сортировки), реализации должны быть снабжены такими параметрами, благодаря которым стала бы возможной работа с типом данных `Item`. На данном этапе под типом данных `Item` подразумевается `int` или `float`; в разделе 6.7 подробно рассматривается реализация типов данных, которые позволят выполнять сортировку элементов произвольной природы с ключами в виде чисел с плавающей точкой, строк и других типов, используя механизмы, описанные в главах 3 и 4.

Функцию `sort` можно заменить любой программой сортировки массивов, представленной в настоящей главе или в главах 7—10. Каждая из них предполагает необходимость сортировки элементов типа `Item`, каждая из них использует три аргумента: массив, левую и правую границы подмассива, подлежащего сортировке. В них также применяется операция `operator<`, осуществляющая сравнение ключей элементов, а также функции `exch` или `compexch`, выполняющие обмен элементов местами. Чтобы иметь возможность различать методы сортировки, будем присваивать различным программам сортировки разные имена. Переименование какой-либо из них, замена программы-драйвера или использование указателей на функции для переключения с одного алгоритма на другой в клиентской программе, подобной программе 6.1, без внесения изменений в программную реализацию алгоритма сортировки, не сопряжено ни с какими трудностями.

Принятые соглашения позволяют проводить анализ многих естественных и компактных программных реализаций алгоритмов сортировки массивов. В разделах 6.7 и 6.8 рассматривается драйвер, который служит иллюстрацией того, как следует использовать реализации сортировок в более общих контекстах, а также различные реализации типов данных. И хотя мы всегда будем уделять должное внимание требованиям к организации программных пакетов, основные усилия будут направляться на решение алгоритмических проблем, к рассмотрению которых мы сейчас и переходим.

Пример функции сортировки, представленный программой 6.1, является одним из вариантов сортировки вставками (*insertion sort*), который более подробно исследуется в разделе 6.3. Так как в нем используются только операции сравнения и обмена, то его можно считать примером неадаптивной (*nonadaptive*) сортировки: последовательность операций, которые она выполняет, не зависит от порядка следования данных. И наоборот, адаптивная (*adaptive*) сортировка выполняет различные последовательности операций в зависимости от результата сравнения (вызов операции `operator<`).

Неадаптивные методы сортировки интересны тем, что они достаточно просто реализуются аппаратными средствами (см. главу 11), однако большинство универсальных алгоритмов сортировки, которые будут предметом наших обсуждений, суть адаптивные методы сортировки.

Как обычно, основной характеристикой алгоритма сортировки, вызывающей наибольший интерес, является время, затрачиваемое на его выполнение. Для выполнения сортировки  $N$  элементов методом выбора, методом вставок и пузырьковым методом, которые будут рассматриваться в разделах 6.2—6.4, требуется время, пропорциональное  $N^2$ , как показано в разделе 6.5. Более совершенные методы, которые исследуются в главах 7—10, могут выполнить сортировку  $N$  элементов за время, пропорциональное  $N \log N$ , однако эти методы не всегда столь же эффективны, как рассматриваемые здесь методы применительно к небольшим значениям  $N$ , а также в некоторых особых случаях. В разделе 6.6 обсуждается более совершенный метод (сортировка методом Шелла), который можно выполнить за время, пропорциональное  $N^{3/2}$  или даже за меньшее время, а в разделе 6.10 приводится специализированный метод (сортировка по ключевым индексам), которая для некоторых типов ключей выполняется за время, пропорциональное  $N$ .

Аналитические результаты, изложенные в предыдущем параграфе, получены на основе подсчета базовых операций (сравнение и обмен значениями), которые выполняет алгоритм. Как уже отмечалось в разделе 2.2, следует также учитывать затраты ресурсов на выполнение этих операций. Тем не менее, в общем случае мы считаем, что основное внимание целесообразно сосредоточить на изучении наиболее часто используемых операций (внутренний цикл алгоритма). Цель заключается в том, чтобы разработать эффективные и недорогие реализации эффективных алгоритмов. Имея перед собой эту цель, мы не только должны избегать неоправданных расширений внутренних циклов алгоритмов, но и искать пути удаления всевозможных команд из внутренних циклов, где это только возможно. В общем, наилучший способ уменьшения стоимости приложения — это переключение на более эффективный алгоритм; другой способ предусматривает минимизацию внутренних циклов по числу команд. Оба эти способа подробно исследуются на примере алгоритмов сортировки.

Дополнительный объем оперативной памяти, используемый алгоритмом сортировки, является вторым по важности фактором, который также будет рассматриваться. По существу, все эти методы можно разбить на три категории: те, которые выполняют сортировку на месте и не нуждаются в дополнительной памяти, за исключением, возможно, небольшого стека или таблицы; те, которые используют представление в виде связного списка или каким-то другим способом получают доступ к данным, используя для этой цели указатели или индексы массивов, в связи с чем необходима дополнительная память для размещения  $N$  указателей или индексов, и те, которые требуют дополнительной памяти для размещения еще одной копии массива, подвергаемого сортировке.

Довольно-таки часто применяются методы сортировки элементов с несколькими ключами — иногда возникает необходимость сортировки одного и того же набора элементов в разные моменты по разным ключам. В таких случаях очень важно знать, обладает ли выбранный метод сортировки следующим свойством:

**Определение 6.1:** *Говорят, что метод сортировки устойчив, если он сохраняет относительный порядок размещения элементов в файле, который содержит дублированные ключи.*

Например, если сформированный по алфавиту список студентов и год окончания школы упорядочен по году, устойчивый метод сортировки выдает список, в котором студенты, окончившие школу в одном и том же году, опять-таки перечисляются в алфавитном порядке, в то время как при использовании неустойчивого метода, скорее всего, будет получен список, в котором алфавитный порядок в рамках года не сохраняется. Очень часто люди, не знакомые с понятием устойчивости, когда впервые сталкиваются с подобной ситуацией, с удивлением обнаруживают, до какой степени неустойчивый алгоритм сортировки нарушает начальный порядок.

Некоторые (но отнюдь не все) простые методы сортировки, которые рассматриваются в данной главе, суть устойчивые методы. С другой стороны, многие из сложных алгоритмов сортировки (опять-таки, не все), которые исследуются в нескольких последующих главах, таковыми не являются. В тех случаях, когда устойчивость должна быть неотъемлемым свойством, мы вводим его, добавляя перед сортировкой к каждому ключу небольшой индекс, либо расширяем ключ сортировки каким-нибудь другим способом. Выполнение такой дополнительной работы равносильно использованию обоих ключей для сортировки, представленной на рис.6.1; использование устойчивого алгоритма сортировки гораздо предпочтительней. Легко согласиться с необходимостью наличия свойства устойчивости; фактически, ряд более сложных алгоритмов, которые будут рассматриваться в следующих главах, достигают устойчивости без существенных дополнительных затрат памяти и времени.

Как уже отмечалось ранее, программы сортировки осуществляют доступ к элементам в соответствии с одним из двух следующих способов: либо доступ к ключам с последующим выполнением операции сравнения, либо доступ к элементам в целом с целью их перемещения. Если сортируемые элементы имеют большие размеры, не имеет смысла перемещать их в памяти, целесообразнее выполнять *непрямую (indirect) сортировку*: переупорядочиваются не сами элементы, а, скорее, массив указателей (индексов) так, что первый указатель указывает на наименьший элемент, следующий — на наименьший из оставшихся и т.д. Ключи можно хранить вместе с са-

Adams	1
Black	2
Brown	4
Jackson	2
Jones	4
Smith	1
Thompson	4
Washington	2
White	3
Wilson	3

Adams	1
Smith	1
Washington	2
Jackson	2
Black	2
White	3
Wilson	3
Thompson	4
Brown	4
Jones	4

Adams	1
Smith	1
Black	2
Jackson	2
Washington	2
White	3
Wilson	3
Brown	4
Jones	4
Thompson	4

### РИСУНОК 6.1. ПРИМЕР УСТОЙЧИВОЙ СОРТИРОВКИ

*Сортировка представленных здесь записей может соответствовать и тому, и другому ключу. Предположим, что первоначально записи были отсортированы по первому ключу (верхняя часть рисунка). Неустойчивая сортировка по второму ключу не сохраняет этот порядок для записей с дублированным ключом (в центре), в то время как устойчивая сортировка сохраняет этот порядок (нижняя часть).*

ними элементами (если ключи большие) либо с указателями (если ключи небольших размеров). Можно переупорядочить элементы после сортировки, но часто в этом нет необходимости, поскольку теперь уже имеется возможность обращения к ним в порядке, установленном сортировкой (косвенным путем). Непрямые методы сортировки рассматриваются в разделе 6.8.

## Упражнения

- ▷ 6.1. Детская игрушка, в которой используются свойства сортировки, состоит из  $i$  карт, каждую из них можно нанизать на колышек в  $i$ -й позиции, причем  $i$  принимает значения от 1 до 5. Разработать метод, который можно использовать для нанизывания карт на колышки, принимая во внимание то обстоятельство, что по виду карты нельзя сказать, можно ли ее надеть на тот или иной колышек (для этого потребуются предпринять попытку надеть карту на колышек).
- 6.2. Карточный трюк требует, чтобы колода карт была упорядочена по мастям (сначала пики, далее черви, трефы и бубны), а внутри каждой масти — по старшинству. Попросите нескольких своих друзей решить эту задачу (перед каждой попыткой перетасуйте карты!) и запишите методы, которыми они пользовались.
- 6.3. Объясните, как вы будете сортировать колоду карт при условии, что карты необходимо укладывать в ряд лицевой стороной вниз, причем допускается проверка значений только двух карт и (при необходимости) обмен местами этих карт.
- 6.4. Объясните, как вы будете сортировать колоду карт при условии, что карты хранятся в колоде, а единственная допустимая операция с ними состоит в том, что выявляются значения двух верхних карт в колоде, обмен этих карт местами и перемещение верхней карты в низ колоды.
- 6.5. Приведите пример последовательностей из трех операций сравнения-обмена, посредством которых выполняется сортировка совокупности из трех элементов.
- 6.6. Приведите пример последовательностей из пяти операций сравнения-обмена, посредством которых выполняется сортировка совокупности из четырех элементов.
- 6.7. Напишите клиентскую программу, которая проверяет, является ли используемая подпрограмма сортировки устойчивой.
- 6.8. Проверка того, что массив отсортирован в результате применения функции `sort`, не дает никаких гарантий того, что сортировка работает. Почему так получается?
- 6.9. Написать клиентскую программу, представляющую собой драйвер, измеряющий эффективность сортировки, который многократно выполняет функцию `sort` на файлах различных размеров, измеряет время каждого выполнения этой функции и выводит на печать (в виде текста или графического изображения) значения среднего времени выполнения сортировки.
- 6.10. Написать учебную клиентскую программу-драйвер, которая выполняет функцию `sort` в сложных или патологических случаях, которые могут иметь место в практических приложениях. Примерами могут служить уже упорядоченные файлы, файлы, представленные в обратном порядке, файлы, все записи которых имеют одни и те же ключи, файлы, содержащие только два отличных друг от друга значения, файлы размерами 0 или 1.



## 6.2. Сортировка выбором

Один из самых простых алгоритмов сортировки работает следующим образом. Сначала отыскивается наименьший элемент массива, затем он меняется местами с элементом, стоящим первым в сортируемом массиве. Далее, находится второй наименьший элемент и меняется местами с элементом, стоящим вторым в исходном массиве. Этот процесс продолжается до тех пор, пока весь массив не будет отсортирован. Изложенный метод называется *сортировкой выбором*, поскольку он работает по принципу выбора наименьшего элемента из числа неотсортированных. На рис 6.2. представлен пример работы этого метода.

Программа 6.2 — суть реализация сортировки выбором, в которой выдержаны все принятые нами соглашения. Внутренний цикл представляет собой сравнение текущего элемента с наименьшим из выявленных к тому времени элементом (плюс программный код, необходимый для увеличения на единицу индекса текущего элемента и проверки того, что он не выходит за границы массива); трудно себе представить более простой метод сортировки. Действия по перемещению элементов выходят за пределы внутреннего цикла: каждая операция обмена элементов местами устанавливает один из них в окончательную позицию, так что всего потребуется выполнить  $N - 1$  таких операций (для последнего элемента эту операцию выполнять не нужно). Таким образом, основную составляющую времени выполнения сортировки представляет количество осуществленных операций сравнения. В разделе 6.5 будет показано, что это время пропорционально  $N^2$ , а также представлены способы вычисления общего времени выполнения программы сортировки и корректного сравнения сортировки выбором и других элементарных методов.

### Программа 6.2. Сортировка выбором

Для каждого  $i$  от 1 до  $r-1$  поменять местами элемент  $a[i]$  с минимальным элементом в последовательности  $a[i], \dots, a[r]$ . По мере продвижения индекса  $i$  слева направо, элементы слева от него занимают свои окончательные позиции в массиве (дальнейшим перемещениям они не подлежат), таким образом, массив будет полностью отсортирован, когда  $i$  достигнет его правого конца.

```
template <class Item>
void selection(Item a[], int l, int r)
{ for (int i = l; i < r; i++)
  { int min = i;
    for (int j = i+1; j <= r; j++)
      if (a[j] < a[min]) min = j;
```

```
Ⓐ S O R T I N G E X A M P L E
A S O R T I N G E X Ⓐ M P L E
A A O R T I N G Ⓔ X S M P L E
A A E R T I N G O X S M P L Ⓔ
A A E E T I N Ⓖ O X S M P L R
A A E E G Ⓘ N T O X S M P L R
A A E E G I N T O X S M P Ⓕ R
A A E E G I L T O X S Ⓜ P N R
A A E E G I L M O X S T P Ⓝ R
A A E E G I L M N X S T P Ⓞ R
A A E E G I L M N O S T Ⓟ X R
A A E E G I L M N O P T S X Ⓠ R
A A E E G I L M N O P R Ⓡ X T
A A E E G I L M N O P R S X Ⓢ T
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

РИСУНОК 6.2. ПРИМЕР СОРТИРОВКИ ВЫБОРОМ

В этом примере первый проход не дал результата, поскольку слева от **A** в массиве нет элемента, меньшего **A**. На втором проходе другой элемент **A** оказался наименьшим среди оставшихся, поэтому он меняется местами с элементом **S**, занимающим вторую позицию. Далее, на третьем проходе, элемент **E**, находящийся в середине массива, меняется местами с **O**, занимающим третью позицию; затем, на четвертом проходе, еще один элемент **E** меняется местами с **R**, занимающим четвертую позицию и т.д.

```
    exch(a[i], a[min]);  
  }  
}
```

Недостаток сортировки выбором заключается в том, что время ее выполнения лишь в малой степени зависит от того, насколько упорядочен исходный файл. Процесс нахождения минимального элемента за один проход файла дает очень мало сведений о том, где может находиться минимальный элемент на следующем проходе этого файла. Например, пользователь, применяющий этот метод, будет немало удивлен, когда узнает, что на сортировку почти отсортированного файла или файла, записи которого имеют одинаковые ключи, требуется столько же времени, сколько и на сортировку файла, упорядоченного случайным образом! Как мы убедимся позже, другие методы с большим успехом используют преимущества присутствия порядка в исходном файле.

Несмотря на всю ее простоту и очевидный примитивизм подхода, сортировка выбором превосходит более совершенные методы в одном из важных приложений: этому методу сортировки файлов отдается предпочтение в тех случаях, когда записи файла огромны, а ключи занимают незначительное пространство. В подобного рода приложениях затраты ресурсов на перемещения записей намного превосходят стоимость операций сравнения, а что касается перемещения данных, то никакой алгоритм не способен выполнить сортировку файла с меньшим числом перемещений данных, нежели метод выбора (см. лемму 6.5, раздел 6.5).

## Упражнения

- ▷ 6.11. В стиле рис. 6.2 показать, как сортируется учебный файл **E A S Y Q U E S T I O N** методом выбора.
- 6.12. Какой максимальной величины достигает число обменов для любого конкретного элемента в процессе сортировки выбором? Что собой представляет среднее количество обменов, приходящееся на один элемент?
- 6.13. Приведите пример файла из  $N$  элементов, в котором число случаев невыполнения условия  $a[j] < a[\text{min}]$  достигает максимального значения (вследствие чего  $\text{min}$  принимает новое значение) в процессе выполнения сортировки выбором.
- 6.14. Является ли сортировка выбором устойчивой?

## 6.3. Сортировка вставками

Метод сортировки, который часто применяют игроки в бридж по отношению к картам на руках, заключается в том, что отдельно анализируется каждый конкретный элемент, который затем помещается в надлежащее место среди других, уже отсортированных элементов. В условиях компьютерной реализации следует позаботиться о том, чтобы освободить место для вставляемого элемента путем смещения больших элементов на одну позицию вправо, после чего на освободившееся место помещается вставляемый элемент. Функция `sort` из программы 6.1 является программной реализацией этого метода, получившего название *сортировки вставками* (*insertion sort*).

Как и в случае сортировки выбором, элементы, находящиеся слева от текущего индекса, отсортированы в соответствующем порядке, тем не менее, они еще не занимают свои окончательные позиции, ибо могут быть передвинуты с целью освобождения места под меньшие элементы, которые обнаруживаются позже. Массив будет полностью отсортирован, когда индекс достигнет правой границы массива. На рис. 6.3 показано, как работает этот метод на примере учебного файла.

Реализация сортировки вставками, представленная в программе 6.1, проста, но не может считаться эффективной. Сейчас мы рассмотрим три способа его совершенствования, иллюстрирующих мотив, который прослеживается во многих разрабатываемых реализациях, а именно: требуется получить компактный, понятный и эффективный программный код, однако эти целевые установки время от времени вступают между собой в противоречие, поэтому часто приходится искать компромисс. Это достигается путем разработки естественной программной реализации с последующим ее улучшением за счет определенной последовательности преобразований с проверкой эффективности (и правильности) каждого такого преобразования.

Прежде всего, можно отказаться от выполнения операций `comprch`, если встречается ключ, который не больше ключа вставляемого элемента, поскольку подмассив, находящийся слева, уже отсортирован. А именно, если справедливо условие  $a[j-1] < a[j]$ , то выполняя команду `break`, можно выйти из внутреннего цикла `for` в функции `sort` программы 6.1. В связи с таким изменением реализация превращается в адаптивную сортировку, благодаря чему быстродействие программы, примененной для сортировки ключей, упорядоченных случайным образом, повышается примерно в два раза (см. лемму 6.2).

Внеся усовершенствования, описанные в предыдущем параграфе, получаем два условия прекращения выполнения внутреннего цикла — можно изменить программный код и представить в виде цикла `while`, дабы отобразить этот факт наглядно. Менее очевидное улучшение реализации следует из того факта, что проверка условия  $u > l$  обычно оказывается излишней: и в самом деле, она достигает цели *только* в случае, когда вставляемый элемент является наименьшим из просмотренных к этому моменту, благодаря чему он достигает начала массива. Широко используемая альтернатива этому заключается в том, чтобы сохранять сортируемые ключи в элементах массива от  $a[1]$  до  $a[N]$ , а *сигнальный ключ* (*sentinel key*) поместить в  $a[0]$ , устанавливая его значение, по меньшей мере, не превышающим наименьшего ключа в сортируемом массиве. Теперь проверка того факта, что обнаружен ключ меньше сигнального, одновременно становится проверкой обоих представляющих интерес условий,

```

A S O R T I N G E X A M P L E
A (S) O R T I N G E X A M P L E
A (O) S R T I N G E X A M P L E
A O (R) S T I N G E X A M P L E
A O R S (T) I N G E X A M P L E
A (I) O R S T N G E X A M P L E
A I (N) O R S T G E X A M P L E
A (G) I N O R S T E X A M P L E
A (E) G I N O R S T X A M P L E
A E G I N O R S T (X) A M P L E
A (A) E G I N O R S T X M P L E
A A E G I (M) N O R S T X P L E
A A E G I M N O (P) R S T X L E
A A E G I (L) M N O P R S T X E
A A E (E) G I L M N O P R S T X
A A E E G I L M N O P R S T X

```

### РИСУНОК 6.3. ПРИМЕР ВЫПОЛНЕНИЯ СОРТИРОВКИ ВСТАВКАМИ

*Во время первого прохода сортировки вставками элемент S, занимающий вторую позицию, больше A, так что трогать его не надо. На втором проходе, когда в третьей позиции встречается элемент O, он меняется местами с S, так что последовательность становится A O S отсортированной и т.д. Незаштрихованные элементы, которые не взяты в кружок, — это те, которые были передвинуты на одну позицию вправо.*

благодаря чему внутренний цикл становится меньше, а быстродействие программы повышается.

Сигнальные ключи иногда не слишком удобны в применении: по-видимому, не очень-то просто определить значение минимально возможного ключа либо вызывающая программа не располагает местом под дополнительный ключ. В программе 6.3 предлагается один из способов обойти обе упомянутых проблемы в условиях сортировки вставками: сначала выполняется отдельный проход вдоль массива, в результате которого элемент с минимальным ключом помещается в первую позицию. Затем сортируется остальной массив, при этом первый элемент, он же наименьший, служит в качестве сигнального ключа. В общем случае следует избегать употребления в программах сигнальных ключей, поскольку зачастую легче воспринимаются программные коды с явными проверками, однако ситуации, когда сигнальные ключи могут оказаться полезными в плане упрощения программы и повышения ее эффективности, обязательно будут фиксироваться.

### Программа 6.3. Сортировка вставками

Эта программа представляет собой усовершенствованный вариант функции `sort` из программы 6.1, поскольку (i) он помещает наименьший элемент массива в первую позицию; в этом качестве наименьший элемент может быть использован как сигнальный ключ; (ii) во внутреннем цикле он выполняет лишь одну операцию присваивания; операция обмена исключается; и (iii) он прекращает выполнение внутреннего цикла, когда вставляемый элемент уже находится в требуемой позиции. Для каждого  $i$  он сортирует элементы  $a[l], \dots, a[i]$ , перемещая на одну позицию вправо элементы  $a[l], \dots, a[i-1]$  из отсортированного списка, которые по значению больше  $a[i]$ , после чего  $a[i]$  попадает в соответствующее место.

```
template <class Item>
void insertion(Item a[], int l, int r)
{ int i;
  for (i = r; i > l; i-) compexch(a[i-1], a[i]);
  for (i = l+2; i <= r; i++)
    { int j = i; Item v = a[i];
      while (v < a[j-1])
        { a[j] = a[j-1]; j--; }
      a[j] = v;
    }
}
```

Третье улучшение, которое сейчас будет рассматриваться, также касается удаления лишних команд из внутреннего цикла. Оно следует из того факта, что последовательные обмены значениями с одним и тем же элементом не эффективны. Если производятся два или большее число обменов значениями, то мы имеем

```
t = a[j]; a[j] = a[j-1]; a[j-1] = t;
```

за которым следует

```
t = a[j-1]; a[j-1] = a[j-2]; a[j-2] = t
```

и т.д. Значение  $t$  в этих двух последовательностях не изменяется, но при этом происходит бесполезная трата времени на его запоминание и последующее чтение с целью следующего обмена значениями. Программа 6.3 передвигает большие элементы

на одну позицию вправо вместо того, чтобы воспользоваться операцией обмена, тем самым избегая напрасной траты времени.

Программа 6.3 является реализацией сортировки методом вставки, обладающей большей эффективностью, нежели программная реализация этого же метода сортировки, включенная в программу 6.1 (в разделе 6.5 мы убедимся в том, что ее быстродействие примерно в два раза выше). В рамках данной книги нас интересуют не только элегантные и одновременно эффективные алгоритмы, но и элегантные и одновременно эффективные реализации этих алгоритмов. В подобных случаях положенные в их основу алгоритмы несколько отличаются друг от друга — было бы правильно назвать функцию `sort` из программы 6.1 *неадаптивной сортировкой вставками* (*nonadaptive insertion sort*). Правильное понимание свойств алгоритма — лучшее руководство при разработке его программной реализации, которая может эффективно использоваться в различных приложениях.

В отличие от сортировки выбором, время выполнения сортировки вставками зависит главным образом от исходного порядка ключей при вводе. Например, если файл большой, а ключи записей уже упорядочены (или почти упорядочены), то сортировка вставками выполняется быстро, а сортировка выбором протекает медленно. Более полное сравнение различных алгоритмов сортировки приводится в разделе 6.5.

## Упражнения

- ▷ 6.15. В стиле рис. 6.3 показать, как сортируется учебный файл `EASYQUESTIONS` методом вставок.
- 6.16. Разработать программную реализацию сортировки вставками, в которой во внутреннем цикле используется оператор `while`, завершающийся по одному из двух условий, описание которых приводится выше.
- 6.17. Для каждого из условий цикла `while` в упражнении 6.16 дать описание файла из  $N$  элементов, для которого в момент выхода из цикла это условие всегда ложно.
- 6.18. Является ли сортировка вставками устойчивой?
- 6.19. Привести пример неадаптивной реализации сортировки выбором, в основе которой лежит выявление минимального элемента, с программным кодом, подобным первому оператору цикла `for` в программе 6.3.

## 6.4. Пузырьковая сортировка

Метод сортировки, который многие обычно осваивают раньше других из-за его исключительной простоты, называется *пузырьковой сортировкой* (*bubble sort*), в рамках которой выполняются следующие действия: проход по файлу с обменом местами соседних элементов, нарушающих заданный порядок, до тех пор, пока файл не будет окончательно отсортирован. Основное достоинство пузырьковой сортировки заключается в том, что его легко реализовать в виде программы, однако вопрос о том, какой из методов сортировки реализуется легче других — пузырьковый, метод вставок или метод выбора — остается открытым. В общем случае пузырьковый метод обладает несколько меньшим быстродействием, однако его все же стоит рассмотреть для полноты картины.

Предположим, что мы всегда передвигаемся по файлу справа налево. Если удастся обнаружить минимальный элемент на первом проходе, он меняется местами с каждым элементом, стоящим от него слева, и, в конце концов, этот элемент помещается в позицию на левой границе массива. Затем на втором проходе в соответствующую позицию устанавливается второй по величине элемент и т.д. Таким образом, вполне достаточно выполнить  $N$  проходов, т.е., пузырьковую сортировку можно рассматривать как один из видов сортировки выбором, хотя при этом для помещения каждого элемента в соответствующую позицию приходится выполнять большой объем действий. Программа 6.4 представляет собой реализацию алгоритма пузырьковой сортировки, а на рис. 6.4 показан пример работы этого алгоритма.

#### Программа 6.4. Пузырьковая сортировка

Для каждого  $i$  от 1 до  $r-1$  внутренний цикл ( $j$ ) помещает минимальный элемент среди элементов последовательности  $a[i], \dots, a[r]$  в  $a[i]$ , переходя от элемента к элементу справа налево и выполняя при этом операции сравнения значений соседних элементов и обмена местами следующих друг за другом элементов. Наименьший элемент беспрепятственно перемещается при всех таких операциях сравнения влево и "всплывает как и пузырек" в начале файла. Как и в случае сортировки методом выбора, в условиях которой индекс  $i$  перемещается по файлу слева направо, элементы слева от него находятся в своих окончательных позициях.

```
template <class Item>
void bubble(Item a[], int l, int r)
{ for (int i = l; i < r; i++)
  for (int j = r; j > i; j--)
    compexch(a[j-1], a[j]);
}
```

Быстродействие программы 6.4 можно повысить за счет экономной реализации внутреннего цикла, выполняя те же приемы, которые применялись в разделе 6.3 при разработке программы сортировки вставками (см. также упражнение 6.25). В самом деле, если сравнивать программные коды, то программа 6.4 оказывается практически идентичной неадаптивной сортировке вставками, реализованной в программе 6.1. Различия между ними состоят в том, что в условиях сортировки вставками внутренний цикл `for` продвигается вдоль левой (отсортированной) части массива, а в условиях пузырьковой сортировки — вдоль правой (не обязательно отсортированной) части массива.

```
A S O R T I N G E X A M P L E
A (A) S O R T I N G E X (E) M P L
A A (E) S O R T I N G E X (L) M P
A A E (E) S O R T I N G (L) X (M) (P)
A A E E (G) S O R T I N (L) (M) X (P)
A A E E G (I) S O R T (L) N (M) P X
A A E E G I (L) S O R T (M) N P X
A A E E G I L (M) S O R T (N) P X
A A E E G I L M (N) S O R T (P) X
A A E E G I L M N (O) S (P) R T X
A A E E G I L M N O (P) S (R) T X
A A E E G I L M N O P (R) S T X
A A E E G I L M N O P R (S) T X
A A E E G I L M N O P R S (T) X
A A E E G I L M N O P R S T (X)
```

#### РИСУНОК 6.4. ПРИМЕР ВЫПОЛНЕНИЯ ПУЗЫРЬКОВОЙ СОРТИРОВКИ

*В процессе пузырьковой сортировки ключи с малыми значениями устремляются влево. Поскольку сортировка производится в направлении справа налево, каждый ключ меняется местами с ключом слева до тех пор, пока не будет обнаружен ключ с меньшим значением. На первом проходе E меняется местами с L, P и M, пока не остановится справа от A; далее уже A продвигается к началу файла, пока не остановится перед другим A, который уже занимает окончательную позицию.  $i$ -й по величине ключ устанавливается в свое окончательное положение после  $i$ -ого прохода, как и в случае сортировки выбором, но при этом и другие ключи также приближаются к своим окончательным позициям.*

Программа 6.4 использует только инструкции `comrexch` и по этой причине не является адаптивной реализацией, однако, если файл практически отсортирован, можно заставить ее работать с большей эффективностью, проверяя, не оказался ли очередной проход таким, что на нем не было выполнено ни одной операции перестановки элементов местами (что равносильно полному упорядочению файла, в связи с чем можно покинуть внешний цикл). Внедрение в программу подобного усовершенствования позволяет повысить быстродействие пузырьковой сортировки на некоторых типах файлов, однако в общем случае она не достигнет той эффективности, которая характерна для программы сортировки вставками, в которую внесены изменения, обеспечивающие своевременный выход из внутреннего цикла, о чем подробно рассказывается в разделе 6.5.

## Упражнения

▷ 6.20. В стиле рис. 6.4 показать, как сортируется учебный файл `EASYQUESTION` методом пузырьковой сортировки.

6.21. Дать пример файла, для которого пузырьковая сортировка выполняет максимально возможное количество перестановок элементов местами.

○ 6.22. Является ли пузырьковая сортировка устойчивой?

6.23. Объясните, почему пузырьковая сортировка оказывается более предпочтительной, нежели неадаптивная версия сортировки выбором, описанная в упражнении 6.19.

● 6.24. Проведите эксперимент с целью определения, сколько проходов файла с произвольной организацией из  $N$  элементов экономится в результате добавления в пузырьковую сортировку возможностей прекращения сортировки по результатам проверки, отсортирован файл или нет.

6.25. Разработать эффективную реализацию пузырьковой сортировки с минимально возможным числом команд во внутреннем цикле. Убедиться в том, что проделанные "усовершенствования" не снижают быстродействия программы!

## 6.5. Характеристики производительности элементарных методов сортировки

Алгоритмы сортировки выбором, вставками и пузырьковая сортировка по времени выполнения находятся в квадратичной зависимости от числа элементов как в наиболее трудных, так и в обычных случаях, но в то же время они не нуждаются в дополнительной памяти. Таким образом, время их выполнения отличается только постоянным коэффициентом пропорциональности этой зависимости, хотя принципы их работы существенно различаются, о чем свидетельствуют рис. 6.5, 6.6 и 6.7.

В общем случае время выполнения алгоритма сортировки пропорционально количеству операций сравнения, выполняемых этим алгоритмом, количеству перемещений или перемен местами элементов, а, возможно, и обоим сразу. В случае ввода данных, упорядоченных случайным образом, сравнение указанных методов сортировки предполагает изучение различий между соответствующими постоянными коэффициентами пропорциональности в зависимости от числа выполняемых операций

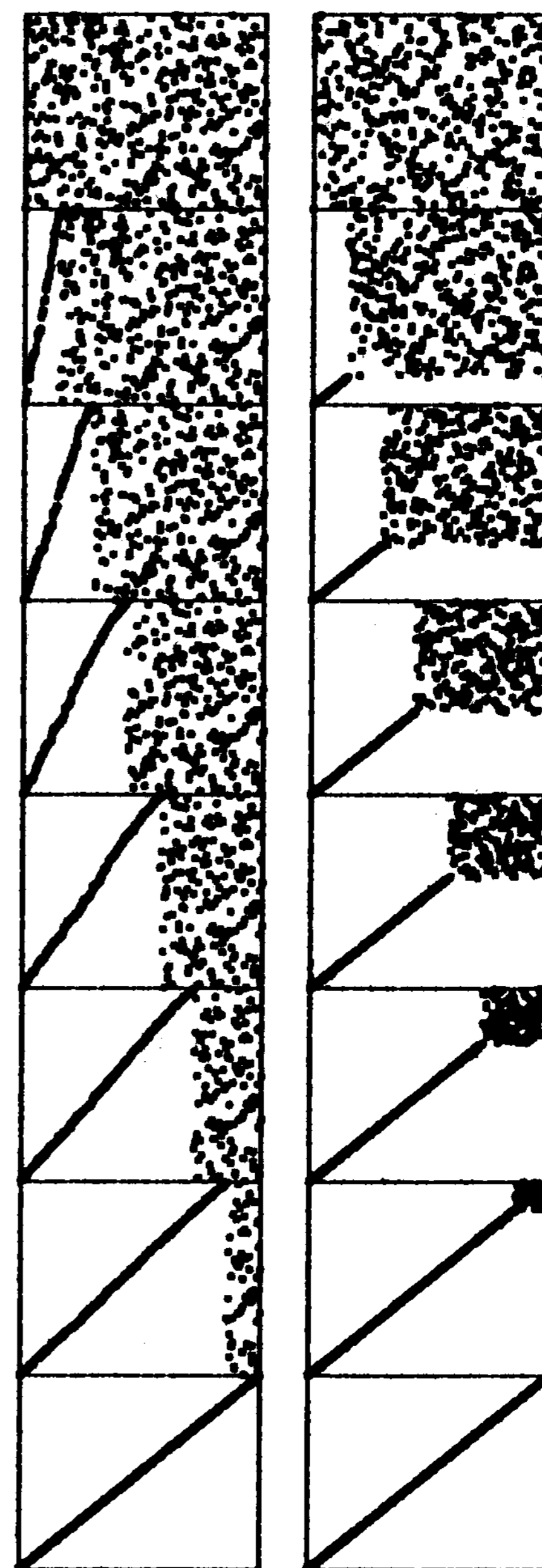
сравнений и перемены сортируемых элементов местами, а также различий соответствующих постоянных коэффициентов пропорциональности в зависимости от числа выполняемых операций во внутренних циклах. В случае ввода данных со специальными характеристиками времена выполнения различных видов сортировок могут отличаться в большей степени, нежели по значениям указанных выше постоянных коэффициентов пропорциональности. В данном разделе подробно рассматриваются аналитические результаты, свидетельствующие в пользу этого заключения.

**Лемма 6.1.** *Сортировка выбором производит примерно  $N^2/2$  операций сравнения и  $N$  операций обмена элементов местами.*

Можно легко проверить это свойство на примере данных, приведенных на рис. 6.2, представляющих собой таблицу размерностью  $N$  на  $N$ , на которой незаштрихованные буквы соответствуют сравнениям. Примерно половина элементов этой таблицы не заштрихована, эти элементы расположены над диагональю. Каждому из  $N - 1$  элементов (за исключением завершающего элемента) на диагонали соответствует операции обмена. Точнее, исследование программного кода показывает, что на каждое  $i$  от 1 до  $N - 1$  приходится одна операция обмена и  $N - i$  сравнений, так что всего производится  $N - 1$  операций обмена и  $(N - 1) + (N - 2) + \dots + 2 + 1 = N(N - 1) / 2$  операций сравнения. Эти свойства сохраняются независимо от природы входных данных; единственный показатель сортировки выбором, зависящий от характера входных данных — это число операций присваивания переменной `min` новых значений. В наихудшем случае эта величина также становится квадратично зависимой, однако в среднем она характеризуется значением  $O(N \log N)$  (см. раздел ссылок), так что мы вправе рассчитывать на то, что время выполнения сортировки выбором не чувствительно к природе входных данных.

**Лемма 6.2.** *Сортировка вставками производит в среднем приблизительно  $N^2/4$  операций сравнения и  $N^2/4$  операций полубомена элементов местами (перемещений) и в два раза больше операций в наихудшем случае.*

Сортировка, реализованная программой 6.3, выполняет одно и то же число сравнений и перемещений. Так же как и лемма 6.1, эта величина легко просле-



**РИСУНОК 6.5. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ СОРТИРОВОК ВЫБОРОМ И ВСТАВКАМИ**

*Эти снимки сортировки вставками (слева) и выбором (справа) на примере случайной последовательности иллюстрируют протекание процессов сортировки указанными выше методами. На рисунке процесс сортировки показан в виде графика зависимости  $a[i]$  от  $i$  для каждого  $i$ . Перед началом сортировки на графике представлена равномерно распределенная случайная величина; по окончании сортировки график представлен диагональю, проходящей из левой нижней угла в правый верхний угол. Сортировка вставками никогда не забегает вперед по отношению к текущей позиции в массиве, в то время как сортировка выбором никогда не возвращается назад.*



живается на диаграмме размером  $N$  на  $N$ , представленной на рис. 6.3, которая подробно иллюстрирует работу алгоритма сортировки вставками. Здесь ведется подсчет элементов, лежащих под главной диагональю, причем в наихудшем случае учитываются все такие элементы. Можно ожидать, что для случайно распределенных входных данных каждый элемент пройдет в среднем половину пути назад, следовательно, необходимо учитывать только половину элементов, лежащих ниже диагонали.

**Лемма 6.3.** *Пузырьковая сортировка производит в среднем примерно  $N^2/2$  операций сравнения и  $N^2/2$  операций обмена как в среднем, так и в наихудшем случаях.*

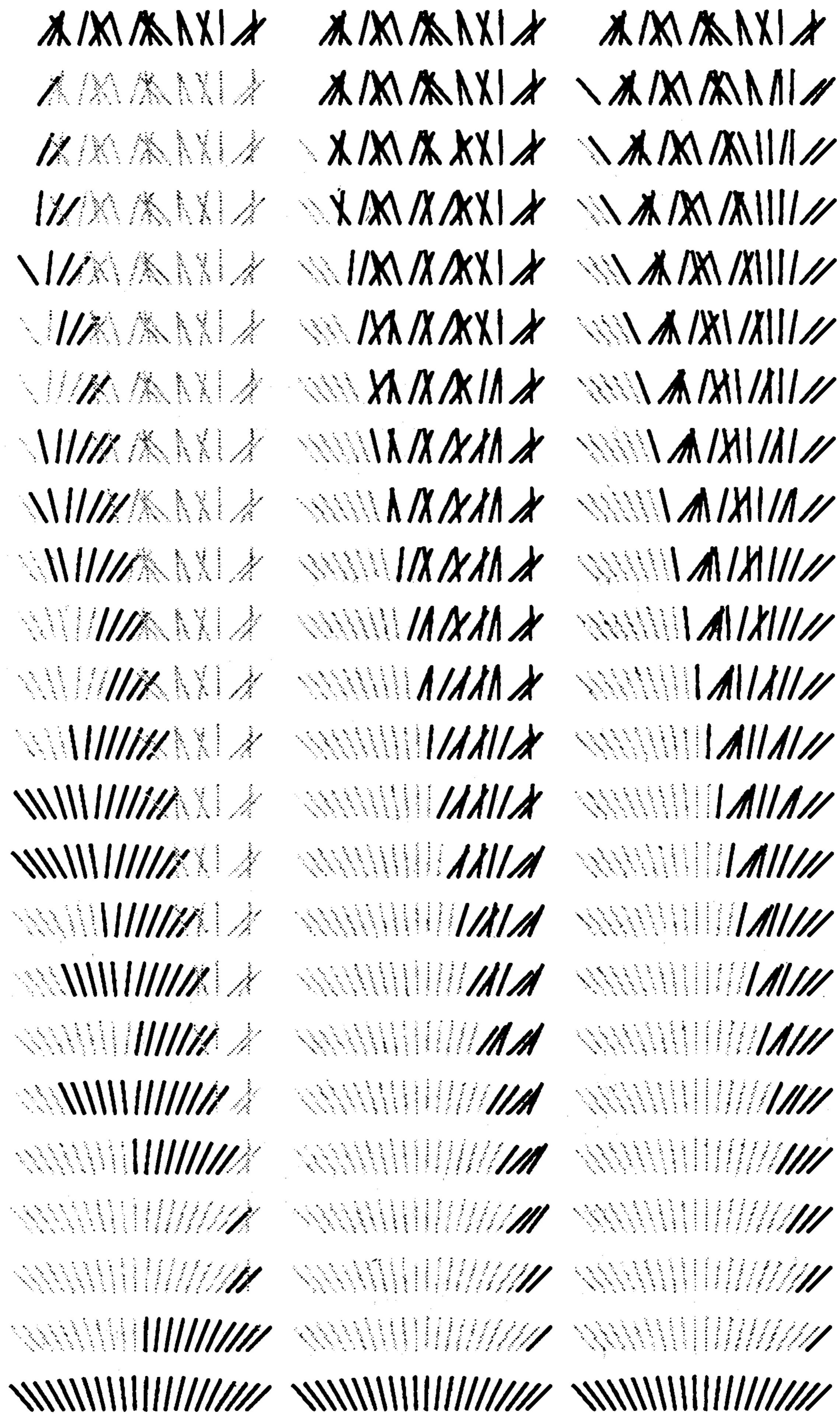
На  $i$ -м проходе пузырьковой сортировки нужно выполнить  $N - i$  операций сравнения-обмена, следовательно, лемма 6.3 доказывается так же, как и случае сортировки выбором. Если алгоритм усовершенствован таким образом, что его выполнение прекращается, как только он обнаруживает, что файл уже отсортирован, то время его выполнения зависит от природы входных данных. Если файл уже отсортирован, то достаточно всего лишь одного прохода, однако  $i$ -й проход требует выполнения  $N - i$  операций сравнения и обмена, если файл отсортирован в обратном порядке. Как отмечалось ранее, эффективность сортировки для обычного случая ненамного выше, чем для самого трудного случая, однако анализ, благодаря которому был установлен этот факт, достаточно сложный (см. раздел ссылок).

И хотя понятие частично отсортированного файла уже по своей природе содержит в себе неопределенность, сортировка вставками и пузырьковая сортировка показывают хорошие результаты при работе с файлами, не обладающими произвольной организацией, которые довольно часто встречаются на практике. Применение в такого рода приложениях универсальных методов сортировки нецелесообразно. Например, рассмотрим, как работает сортировка вставками на файле, который уже является отсортированным. Сразу же выясняется, что все элементы файла находятся на своих местах, а общие затраты времени на сортировку линейно зависят от числа элементов. Это же утверждение справедливо и в отношении пузырьковой сортировки, но для сортировки выбором эта зависимость остается квадратичной.

**Определение 6.2.** *Инверсией называется пара ключей, которые нарушают порядок в файле.*

Для подсчета количества инверсий в файле для каждого элемента потребуется просуммировать число элементов слева, которые превосходят его по величине (мы будем называть это значение числом инверсий, соответствующих выбранному элементу). Но это число есть в точности расстояние, которое должны пройти эти элементы во время сортировки вставками. В частично отсортированном файле меньше инверсий, чем в произвольно упорядоченном файле.

Существуют такие типы частично отсортированных файлов, в которых каждый элемент находится достаточно близко к своей окончательной позиции. Например, некоторые игроки в карты сортируют имеющиеся у них на руках карты, сначала располагая их по масти, тем самым помещая близко к их окончательным позициям, а затем упорядочивают карты каждой масти по старшинству. Далее мы рассмотрим некоторое число методов сортировки, которые действуют примерно таким же образом — на начальной стадии они размещают элементы вблизи окончательных позиций, в



### РИСУНОК 6.6. ОПЕРАЦИИ СРАВНЕНИЯ И ОБМЕНА ЭЛЕМЕНТОВ В УСЛОВИЯХ ЭЛЕМЕНТАРНЫХ МЕТОДОВ СОРТИРОВКИ

На этой диаграмме показаны различия в том, как сортировки вставками, выбором, а также пузырьковая сортировка приводят конкретный файл в порядок. Сортируемый файл представлен в виде отрезков, которые сортируются по углу наклона. Черные отрезки соответствуют элементам, к которым в процессе сортировки производится доступ на каждом проходе; серые отрезки соответствуют элементам, которые не затрагиваются. В условиях сортировки вставками (слева) вставляемый элемент проходит примерно половину пути назад через отсортированную часть на каждом проходе. Сортировка выбором (в центре) и пузырьковая сортировка (справа) проходят на каждом проходе через всю неотсортированную часть массива с целью обнаружения там следующего наименьшего элемента; различие между этими методами заключается в том, что пузырьковая сортировка меняет местами каждую пару соседних элементов, нарушающих порядок, которые ей удастся обнаружить, в то время как сортировка выбором помещает минимальный элемент в окончательную позицию. Это различие проявляется в том, что по мере выполнения пузырьковой сортировки, неотсортированная часть массива становится все более упорядоченной.

результате чего образуется частично упорядоченный файл, в котором каждый элемент расположен недалеко от той позиции, которую он в конечном итоге должен занять. Высокую эффективность при сортировке таких файлов демонстрируют сортировка вставками и пузырьковая сортировка (но не сортировка выбором).

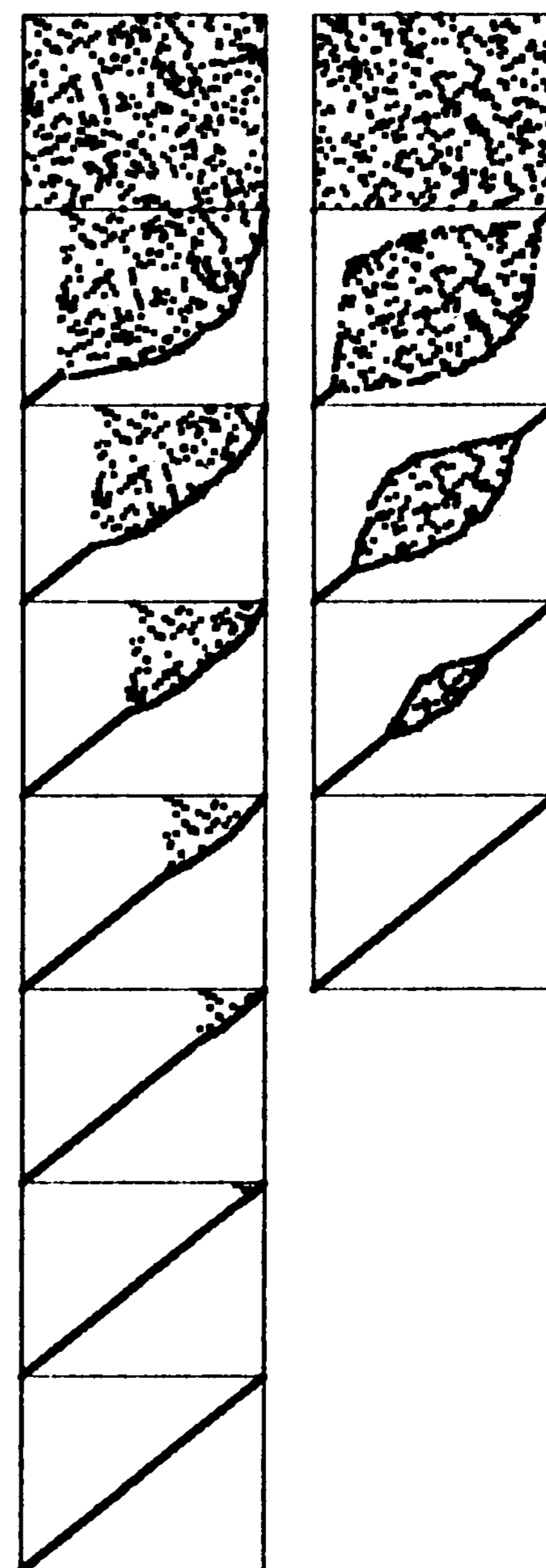
**Лемма 6.4.** *Методы вставок и пузырьковой сортировки выполняют линейно зависящее от  $N$  число операций сравнения и обмена при сортировке файлов с не более чем постоянным числом инверсий, приходящихся на каждый элемент.*

Как было только что отмечено, время выполнения сортировки вставками прямо пропорционально количеству инверсий в сортируемом файле. Что касается пузырьковой сортировки (здесь мы ссылаемся на программу 6.4, скорректированную таким образом, что ее выполнение прекращается, как только завершена сортировка файла), то доказательство подобного утверждения требует дополнительных рассуждений (см. упражнение 6.29). Каждый проход пузырьковой сортировки уменьшает справа от любого элемента число меньших его элементов точно на 1 (если, естественно, это число не было равным 0), следовательно, пузырьковая сортировка использует не более чем постоянное число проходов для типов файлов, которые сейчас рассматриваются (т.е. частично упорядоченных), и поэтому количество выполняемых операций сравнения и обмена линейно зависит от  $N$ .

Другой тип частично отсортированных файлов может быть получен за счет добавления нескольких элементов к уже отсортированному файлу либо путем соответствующего изменения ключей нескольких элементов отсортированного файла. Подобные типы файлов наиболее часто встречаются в приложениях сортировки. Для таких файлов наиболее эффективным является сортировка вставками: сортировка выбором и пузырьковая сортировка таковыми не являются.

### Таблица 6.1. Эмпирические исследования элементарных алгоритмов сортировки

На файлах небольших размеров быстроедействие сортировки вставками и выбором примерно в два раза выше, чем у пузырьковой сортировки, но время выполнения этого вида сортировки находится в квадратичной зависимости от размера файла (если размер файла возрастает в 2 раза, то время его сортировки возрастает в 4 раза). Ни



**РИСУНОК 6.7. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ ДВУХ ПУЗЫРЬКОВЫХ СОРТИРОВОК**

*Стандартная пузырьковая сортировка (слева) работает подобно сортировке выбором в плане того, что каждый проход устанавливает один элемент в его окончательную позицию, но в то же время он асимметрично приносит некоторый порядок в остальную часть массива. Смена направления просмотра массива на альтернативный (т.е. просмотр в направлении с начала до конца массива меняется на обратный и наоборот) порождает новую разновидность пузырьковой сортировки, получившей название **шейкер-сортировки** (справа), которая заканчивается быстрее (см. упражнение 6.30).*

один из этих методов не следует использовать для сортировки больших случайно упорядоченных файлов — например, соответствующие показатели, относящиеся к алгоритму сортировки методом Шелла (см. раздел 6.6), возрастают менее чем в 2 раза. В тех случаях, когда выполнение операций сравнения сопряжено с большими затратами, — например, когда ключи представлены в виде строк — сортировка вставками работает гораздо быстрее, чем два других рассматриваемых способа, поскольку в условиях сортировки вставками выполняется меньшее количество операций сравнения. Здесь мы не обсуждаем ситуации, когда дорогостоящими являются операции обмена; в таких случаях наилучшей является сортировка выбором.

32-разрядные целочисленные ключи						ключи в виде строк		
N	S	I*	I	B	B*	S	I	B
1000	5	7	4	11	8	13	8	19
2000	21	29	15	45	34	56	31	78
4000	85	119	62	182	138	228	126	321

#### Ключи:

- S Сортировка выбором (программа 6.2)
- I\* Сортировка вставками на основе операций обмена (программа 6.1)
- I Сортировка вставками (программа 6.3)
- B Пузырьковая сортировка (программа 6.4)
- B\* Шейкер-сортировка (упражнение 6.30)

**Лемма 6.5.** Сортировка вставками использует линейно зависимое (от  $N$ ) число операций сравнения и обмена для файлов с не более чем постоянным количеством элементов, которые имеют более чем постоянное число соответствующих инверсий.

Время выполнения сортировки вставками зависит от общего числа инверсий в файле и не зависит от того, каким образом эти инверсии распределены в файле.

Чтобы сделать выводы относительно времени выполнения сортировок на основании лемм 6.1—6.5, необходимо проанализировать затраты на операции сравнения и обмена, а этот фактор, в свою очередь, зависит от размеров элементов и ключей (см. таблицу 6.1). Например, если элементы представляют собой ключи в виде одного слова, то операция обмена (требующая четырех доступов к массиву) должна стоить в два раза дороже операции сравнения. В такой ситуации значения времени, затрачиваемого на выполнение сортировок вставками и выбором, можно в первом приближении считать соизмеримыми, в то время как пузырьковая сортировка выполняется медленнее. Но если сами элементы больше ключей, то лучше всего подходит сортировка выбором.

**Лемма 6.6.** Время выполнения сортировки выбором линейно зависит от размеров файлов с большими элементами и малыми ключами.

Пусть  $M$  есть отношение размера элемента к размеру ключа. Тогда можно предположить, что стоимость операции сравнения составляет 1 единицу времени, а стоимость операции обмена —  $M$  единиц времени. Сортировка выбором затрачивает на операции сравнения примерно  $N^2/2$  единиц времени и порядка  $NM$  единиц времени на операции обмена. Если  $M$  больше постоянного кратного  $N$ , то произ-

ведение  $NM$  превосходит  $N^2$ , так что время выполнения сортировки пропорционально произведению  $NM$ , которое, в свою очередь, пропорционально количеству времени, которое требуется для перемещения всех данных.

Например, если требуется выполнить сортировку 1000 элементов, каждый из которых состоит из ключа в виде одного слова и 1000 слов данных, а мы фактически должны переупорядочить эти элементы, то трудно найти что-либо лучшее, чем сортировка выбором, поскольку основной составляющей времени выполнения для нее будет стоимость перемещения в конечном итоге 1 миллиона слов данных. В разделе 6.8 мы столкнемся с альтернативой переупорядочиванию данных.

## Упражнения

- ▷ 6.26. Какой из трех элементарных методов (сортировка выбором, сортировка вставками и пузырьковая сортировка) выполняется быстрее на файле, элементы которого снабжены идентичными ключами?
- 6.27. Какой из трех перечисленных выше элементарных методов выполняется быстрее на файле, упорядоченном в обратном порядке?
- 6.28. Дать пример файла из 10 элементов (использовать ключи от А до J), в процессе сортировки которого пузырьковая сортировка выполняет меньше операций сравнения, чем метод вставок, либо же доказать, что такой файл не существует.
- 6.29. Показать, что каждый проход пузырьковой сортировки уменьшает ровно на 1 число элементов слева от текущего, превосходящих его по значению (естественно, если это число не равно 0).
- 6.30. Реализовать вариант пузырьковой сортировки, который попеременно применяет проходы по данным слева направо и справа налево. Этот (более быстродействующий, но в то же время более сложный) алгоритм называется *шейкер-сортировкой* (*shaker sort*) (см. рис.6.7).
- 6.31. Показать, что лемма 6.5 не характерна для шейкер-сортировки (см. упражнение 6.30).
- 6.32. Напишите программу сортировки на PostScript (см. раздел 4.3) и воспользуйтесь полученной программной реализацией для построения диаграмм типа 6.5 — 6.7. Можно применить рекурсивную реализацию или обратиться к справочной литературе для изучения циклов и массивов PostScript.

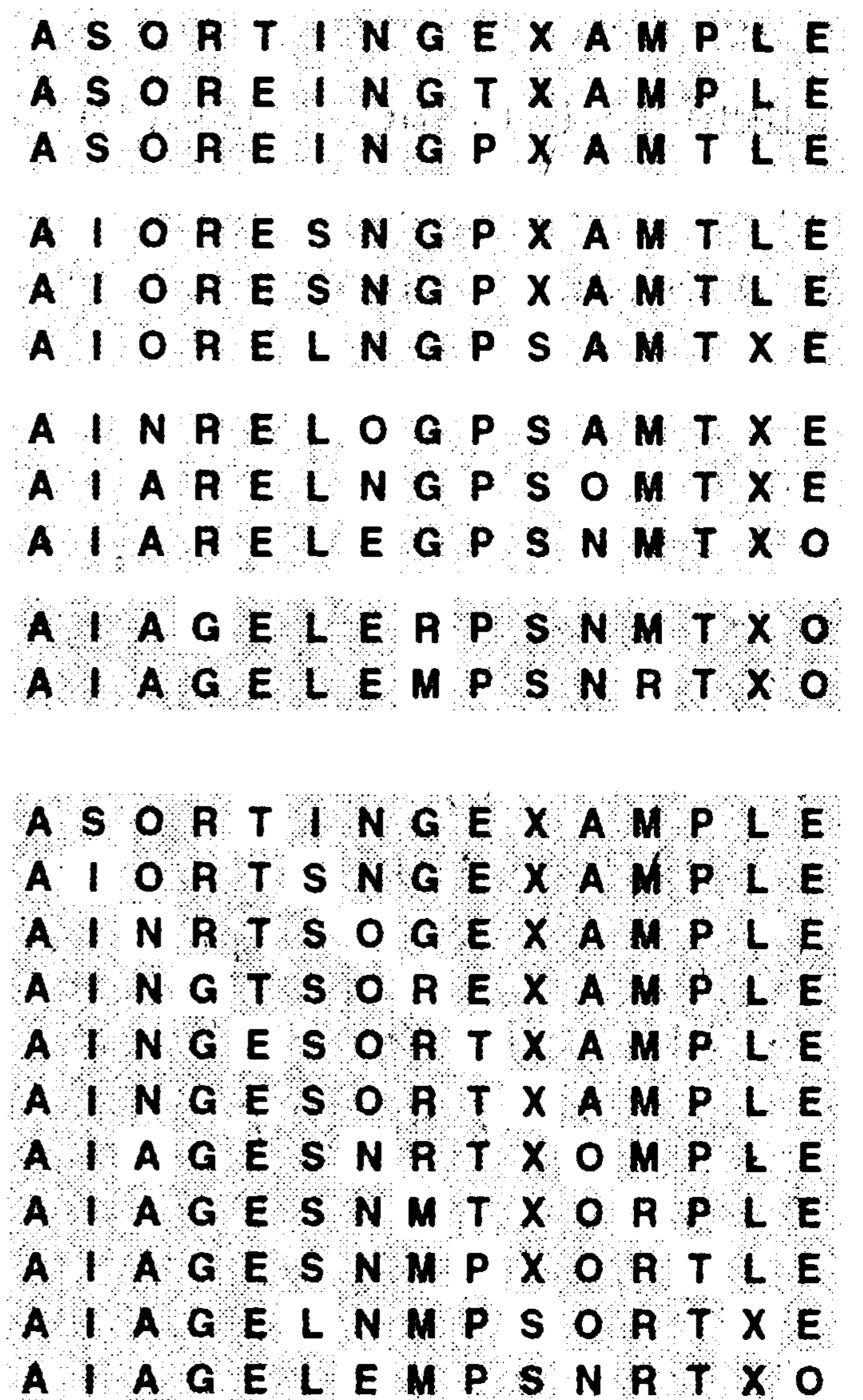
## 6.6. Сортировка методом Шелла

Сортировка вставками не относится к категории быстродействующих, поскольку единственный вид операции обмена, который она использует, выполняется над двумя соседними элементами, в связи с чем элемент может передвигаться вдоль массива лишь на одно место за один раз. Например, если элемент с наименьшим значением ключа оказывается в конце массива, потребуется сделать  $N$  шагов, чтобы поместить его в надлежащее место. Сортировка методом Шелла представляет собой простейшее расширение метода вставок, быстродействие которого выше за счет обеспечения возможности обмена местами элементов, которые находятся далеко один от другого.

Идея заключается в переупорядочении файла таким образом, чтобы придать ему следующее свойство: совокупность  $h$ -ых элементов исходного файла (начиная с любого) образует отсортированный файл. В таком случае говорят, что файл *h-упорядочен* (*h-sorted*). Другими словами,  $h$ -упорядоченный файл представляет собой  $h$  независимо отсортированных взаимно проникающих друг в друга файлов. В процессе  $h$ -сортировки при достаточно больших значениях  $h$  можно менять местами элементы массива, расположенные достаточно далеко друг от друга, и тем самым ускорить последующую  $h$ -сортировку при меньших значениях  $h$ . Использование такого рода процедуры для любой последовательности значений  $h$ , которая заканчивается единицей, завершается получением упорядоченного файла: именно в этом и заключается суть сортировки методом Шелла.

Один из способов реализации сортировки методом Шелла заключается в том, что для каждого  $h$  независимо используется сортировка вставками на каждом из  $h$  подфайлов. Несмотря на очевидную простоту этого процесса, возможен еще более простой подход именно благодаря тому, что подфайлы независимы. В процессе  $h$ -сортировки файла мы просто вставляем элемент среди предшествующих ему элементов в соответствующем  $h$ -подфайле, перемещая большие по значению элементы вправо (см. рис.6.8). Эта задача решается путем использования программы сортировки вставками, при которой каждый шаг перемещения по файлу в сторону увеличения или уменьшения равен  $h$ , но не равен 1. С учетом данного обстоятельства программная реализация сортировки методом Шелла сводится всего лишь к тому, что для каждого значения шага осуществляется проход по файлу, характерный для сортировки вставками, аналогичный реализованному в программе 6.5. Работа программы показана на рис. 6.9.

Возникает вопрос: какую последовательность шагов следует использовать? В общем случае на этот вопрос трудно найти правильный ответ. В литературе опубликованы результаты исследований различных последовательностей шагов, некоторые из них хорошо зарекомендовали себя на практике, однако



#### РИСУНОК 6.8. ВЗАИМНО ПРОНИКАЮЩИЕ 4-СОРТИРОВКИ

*В верхней части данной диаграммы показан процесс 4-сортировки файла, состоящего из 15 элементов, сначала выполняется сортировка вставками подфайла, содержащего элементы в позициях 0, 4, 8, 12, затем сортировка вставками подфайла на позициях 1, 5, 9, 13, далее сортировка вставками подфайла в позициях 2, 6, 10, 14 и, наконец, сортировка вставками подфайла в позициях 3, 7, 11. Однако все четыре подфайла независимы друг от друга, так что аналогичный результат можно получить, вставляя каждый элемент в соответствующую ему позицию в его подфайле и перемещаясь в обратном направлении на четыре элемента за один раз (нижняя часть рисунка). Беря первый ряд каждого раздела верхней диаграммы, затем второй ряд каждого раздела и так далее, получаем диаграмму в нижней части рисунка.*

наилучшую последовательность, по-видимому, отыскать не удалось. В общем случае на практике используются убывающие последовательности шагов, близкие к геометрической прогрессии, в результате чего число шагов находится в логарифмической зависимости от размеров файлов. Например, если размер следующего шага равен примерно половине предыдущего, то для сортировки файла, состоящего из 1 миллиона элементов, потребуется примерно 20 шагов, если такое соотношение примерно равно одной четвертой, то достаточно будет 10 шагов. Использование как можно меньшего числа шагов — это весьма важное требование, которое нетрудно учесть, но при этом в последовательности шагов необходимо выдерживать различные арифметические соотношения, такие как величины их общих делителей и ряд других свойств.

Практический результат от обнаружения хорошей последовательности шагов, по-видимому, ограничен повышением быстродействия алгоритма на 25%, в то же время сама проблема представляет собой увлекательную загадку — примером того, какие сложные вопросы характерны для использования на первый взгляд простого алгоритма.

### Программа 6.5. Сортировка методом Шелла

Если отказаться от использования сигнальных ключей, а затем заменить каждое появление "1" на "h" в сортировке вставками, то полученная при этом программа реализует h-сортировку файла. Добавление внешнего цикла, изменяющего значение шага, позволяет получить компактную программную реализацию сортировки методом Шелла, в которой используется последовательность шагов **1 4 13 40 121 364 1093 3280 9841 ...**

```
template <class Item>
void shellsort(Item a[], int l, int r)
{ int h;
  for (h = 1; h <= (r-l)/9; h = 3*h+1) ;
  for ( ; h > 0; h /= 3)
    for (int i = l+h; i <= r; i++)
      { int j = i; Item v = a[i];
        while (j >= l+h && v < a[j-h])
          { a[j] = a[j-h]; j -= h; }
        a[j] = v;
      }
}
```

Последовательность шагов **1 4 13 40 121 364 1093 3280 9841 ...**, используемая в программе 6.5, в которой соотношение соседних шагов составляет приблизительно одну треть, была рекомендована Кнудом в 1969 г. (см. раздел ссылок). Она просто вычисляется (начав с 1, получить значение следующего шага, умножив предыдущее значение на 3 и добавив 1) и обеспечивает реализацию сравнительно эффективной сортировки даже в случае относительно больших файлов (см. рис.6.10).

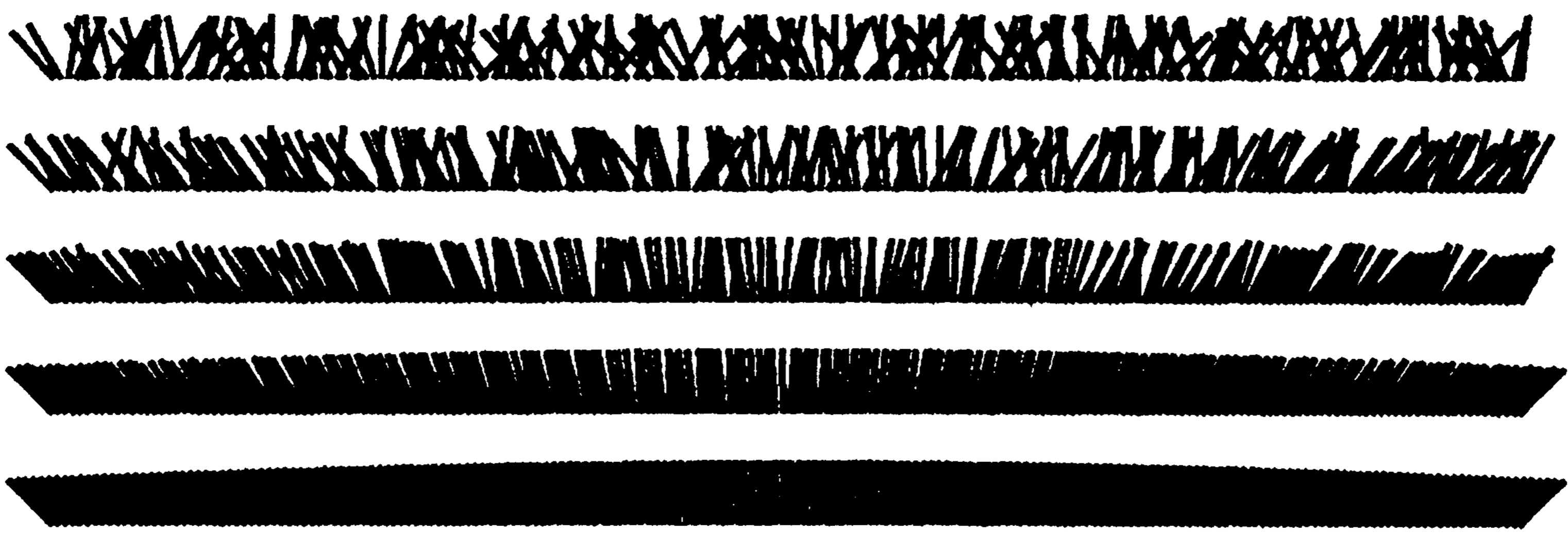
```
A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A E O R T I N G E X A M P L S
```

```
A E O R T I N G E X A M P L S
A E O R T I N G E X A M P L S
A E N R T I O G E X A M P L S
A E N G T I O R E X A M P L S
A E N G E I O R T X A M P L S
A E N G E I O R T X A M P L S
A E A G E I N R T X O M P L S
A E A G E I N M T X O R P L S
A E A G E I N M P X O R T L S
A E A G E I N M P L O R T X S
A E A G E I N M P L O R T X S
```

```
A E A G E I N M P L O R T X S
A A E G E I N M P L O R T X S
A A E G E I N M P L O R T X S
A A E E G I N M P L O R T X S
A A E E G I N M P L O R T X S
A A E E G I N M P L O R T X S
A A E E G I M N P L O R T X S
A A E E G I M N P L O R T X S
A A E E G I L M N P O R T X S
A A E E G I L M N O P R T X S
A A E E G I L M N O P R T X S
A A E E G I L M N O P R T X S
A A E E G I L M N O P R T X S
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

### РИСУНОК 6.9. ПРИМЕР СОРТИРОВКИ МЕТОДОМ ШЕЛЛА.

*Сортировка файла при помощи 13-сортировки (сверху), с последующими 4-сортировкой (в центре) и 1-сортировкой (внизу), не требует выполнения большого объема операций сравнения (о чем можно судить по количеству незаштрихованных элементов). Завершающий проход есть ни что иное как сортировка вставками, но при этом ни один из элементов не должен далеко перемещаться благодаря порядку, установленному в файле на двух первых проходах.*



**РИСУНОК 6.10. СОРТИРОВКА МЕТОДОМ ШЕЛЛА СЛУЧАЙНО РАСПРЕДЕЛЕННОЙ ПЕРЕСТАНОВКИ**

*Целью каждого прохода при сортировке методом Шелла состоит в том, чтобы привести в файл как единое целое большую степень упорядоченности. Сначала файл подвергается 40-сортировке, затем 13-сортировке, далее 4-сортировке, и наконец, 1-сортировке. Каждый проход приближает порядок в файле к окончательному.*

Многие другие последовательности шагов позволяют получить еще более эффективную сортировку, однако довольно трудно превзойти эффективность программы 6.5 более чем на 20% даже в случае сравнительно больших значений  $N$ . Одной из таких последовательностей является **1 8 23 77 281 1073 4193 16577 ...**, т.е. последовательность  $4^{i+1} + 3 \cdot 2^i + 1$  для  $i > 0$ . Можно доказать, что приведенная последовательность обеспечивает повышенное быстродействие для самых трудных случаев сортировки (см. лемму 6.10). Рисунок 6.12 показывает, что эта последовательность, равно как и последовательность Кнута, а также многие другие последовательности шагов, обладают похожими динамическими характеристиками для файлов больших размеров. Вполне возможно, что существуют лучшие последовательности. Несколько идей по улучшению последовательностей шагов приводится в разделе упражнений.

С другой стороны, существуют и плохие последовательности шагов: например, **1 2 4 8 16 32 64 128 256 512 1024 2048 ...** (первая последовательность шагов, предложенная Шеллом еще в 1959 г. (см. раздел ссылок)), скорее всего, служит причиной низкой эффективности сортировки, поскольку элементы на нечетных позициях не сравниваются с элементами на четных позициях вплоть до последнего прохода. Этот эффект заметен на файлах с произвольной организацией, он становится катастрофическим в наихудших случаях: эффективность метода резко снижается и время выполнения сортировки становится пропорциональным квадрату  $N$ , если, например, половина элементов файла с меньшими значениями находится в четных позициях, а другая половина элементов (с большими значениями) — в нечетных позициях (см. упражнение 6.36).

Программа 6.5 вычисляет следующий шаг, разделив текущий шаг на 3 после инициализации с таким расчетом, чтобы всегда использовалась одна и та же последовательность шагов. Другой вариант заключается в том, что сортировка начинается с  $h=N/3$  или для этой цели используется другая функция от  $N$ . Но лучше отказаться от стратегий подобного рода, ибо для некоторых значений  $N$  могут появиться плохие последовательности шагов наподобие описанной выше.

Наше описание эффективности сортировки методом Шелла не отличается особой точностью, поскольку никому еще не удалось выполнить точный анализ этого алгоритма. Этот пробел в наших знаниях затрудняет не только оценку различных после-



довательностей шагов, но и аналитическое сравнение сортировки методом Шелла с другими методами сортировки. Не известна даже функциональная форма для определения времени выполнения сортировки методом Шелла (более того, эта форма зависит от выбора последовательности шагов). Кнут обнаружил, что функциональные формы  $N(\log N)^2$  и  $N^{1.25}$  довольно точно описывают ситуацию, а дальнейшие исследования показали, что для некоторых видов последовательностей подходят более сложные функции вида  $N^{1+1/\sqrt{\lg N}}$ .

В завершение текущего раздела отвлечемся от основной темы и обсудим некоторые известные результаты исследования сортировки методом Шелла. Основная цель таких обсуждений состоит в том, чтобы показать, что даже алгоритмы, которые на первый взгляд кажутся простыми, обладают сложными свойствами, а анализ алгоритмов имеет не только практическое значение, но и может представлять собой интересную научную задачу. Для тех читателей, которых увлекла идея поиска новых, более совершенных последовательностей шагов сортировки методом Шелла, следующая ниже информация окажется весьма полезной; все остальные могут сразу перейти к изучению раздела 6.7.

**Лемма 6.7.** *Результатом  $h$ -сортировки  $k$ -упорядоченного файла есть  $h$ - и  $k$ -упорядоченный файл.*

Этот факт кажется очевидным, однако чтобы его доказать, потребуются значительные усилия (см. упражнение 6.47).

**Лемма 6.8.** *Сортировка методом Шелла выполняет менее  $N(h-1)(k-1)/g$  операций сравнения при  $g$ -сортировке  $h$ - и  $k$ -упорядоченного файла при условии, что  $h$  и  $k$  взаимно просты.*

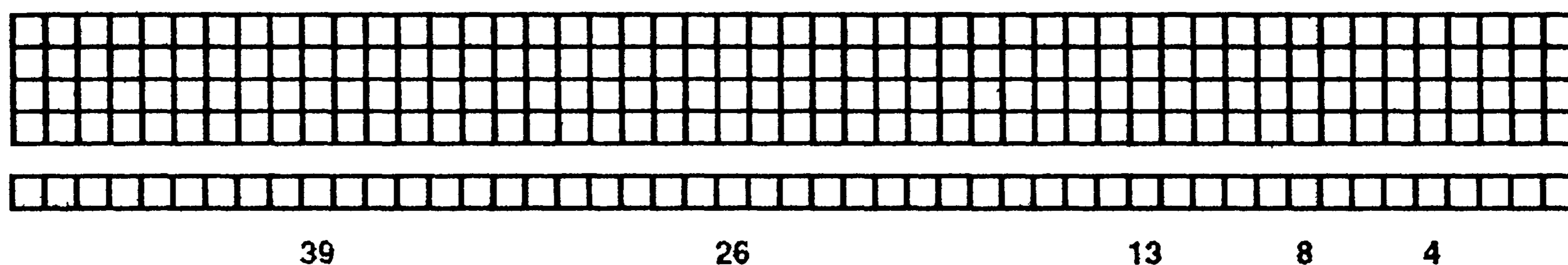
Основа этого факта показана на рис. 6.11. Ни один элемент, расположенный дальше  $(h-1)(k-1)$  позиций слева от любого заданного элемента  $x$ , не может быть больше  $x$ , если  $h$  и  $k$  взаимно просты (см. упражнение 6.43). При  $g$ -сортировке проверяются не более одного из  $g$  таких элементов.

**Лемма 6.9.** *Сортировка методом Шелла выполняет менее  $O(N^{3/2})$  операций сравнения для последовательности шагов 1 4 13 40 121 364 1093 3280 9841...*

Для больших шагов, когда имеются  $h$  подфайлов размером  $N/h$ , в наихудшем случае расходы составляют примерно  $N^2/h$ . При малых шагах из леммы 6.8 следует, что стоимость составляет приблизительно  $Nh$ . Все зависит от того, насколько успешно удастся вписаться в эти границы на каждом шаге. Это справедливо для каждой относительно простой последовательности, возрастающей экспоненциально.

**Лемма 6.10.** *Сортировка методом Шелла выполняет менее  $O(N^{4/3})$  операций сравнения для последовательности шагов 1 8 23 77 281 1073 4193 16577...*

Доказательство этой леммы практически не отличается от доказательства леммы 6.9. Из леммы, аналогичной лемме 6.8, следует, что стоимость сортировки при небольших значениях шагов принимает значение порядка  $Nh^{1/2}$ . Доказательство этой леммы требует привлечения аппарата теории чисел, что выходит за рамки данной книги (см. раздел ссылок).



**РИСУНОК 6.11. 4- И 13-УПОРЯДОЧЕННЫЙ ФАЙЛ.**

Нижний ряд изображает массив, при этом заштрихованные квадратики обозначают элементы, которые должны быть меньше или равны крайнему правому элементу массива, если этот массив 4- и 13-упорядоченный. 4 верхних ряда отражают, как появился нижний ряд. Если элемент справа находится в массиве в позиции  $i$ , то 4-упорядочение означает, что элементы массива в позициях  $i-4$ ,  $i-8$ ,  $i-12$ , ... меньше или равны ему (верхний ряд); 13-упорядочение означает, что элемент  $i-13$ , а вместе с ним, в силу 4-упорядочения, и элементы  $i-17$ ,  $i-21$ ,  $i-25$ , ... меньше или равны ему (второй ряд сверху); по той же причине элемент в позиции  $i-26$ , а вместе с ним, в силу 4-упорядочения, и элементы  $i-30$ ,  $i-34$ ,  $i-38$ , ... меньше или равны ему (третий ряд сверху), и т.д. Оставшиеся незаштрихованными квадратики суть те, которые могут быть больше, чем элемент слева; в рассматриваемом примере таких элементов самое большее 18 (дальше всех находится элемент в позиции  $i-36$ ). Таким образом, потребуется выполнить самое большее  $18N$  сравнений при сортировке вставками 13- и 4-упорядоченного файла, состоящего из  $N$  элементов.

Последовательности шагов, которые рассматривались до сих пор, эффективны в силу того, что следующие один за другим элементы последовательности взаимно просты. Другое семейство последовательностей шагов эффективно именно благодаря тому, что такие элементы *не являются* взаимно простыми.

В частности, из доказательства леммы 6.8 следует, что в процессе завершающей сортировки вставками 2- и 3-упорядоченного файла каждый элемент перемещается самое большее на одну позицию. Это значит, что такой файл может быть отсортирован за один проход пузырьковой сортировки (отпадает необходимость в дополнительном цикле). Теперь, если файл 4-упорядочен и 6-упорядочен, то из этого следует, что каждый элемент перемещается максимум на одну позицию, если выполнить его 2-упорядочение (поскольку каждый подфайл 2- и 3-упорядочен); и если файл 6-упорядочен и 9-упорядочен, то каждый элемент перемещается самое большее на одну позицию при его 3-сортировке. Продолжая рассуждения в этом направлении, мы приходим к идее, которую Пратт опубликовал в 1971 г. (см. раздел ссылок).

Метода Пратта основан на использовании треугольника шагов, причем каждое входящее в этот треугольник число в два раза больше числа, стоящего сверху справа, и в три раза больше числа, стоящего в треугольнике сверху слева.

				1			
			2		3		
		4		6		9	
	8		12		18		27
	16	24		36		54	81
	32	48	72		108	162	243
64	96	144	216	324	486	729	

Если мы используем эти числа снизу вверх и справа налево как последовательность шагов в рамках сортировки методом Шелла, то каждому шагу  $x$  в нижнем ряду предшествуют значения  $2x$  и  $3x$ , так что каждый подфайл оказывается 2-упорядочен и 3-упорядочен и ни один элемент не передвигается больше, чем на одну позицию в процессе всей сортировки!

**Таблица 6.2. Эмпирическое исследование последовательностей шагов сортировки методом Шелла.**

Сортировка методом Шелла выполняется в несколько раз быстрее по сравнению с другими элементарными методами сортировки даже в тех случаях, когда шаги являются степенями 2, в то же время некоторые специальные виды последовательностей шагов позволяют увеличить ее быстродействие в 5 и более раз. Три лучших последовательности, приведенные в данной таблице, существенно различаются по положенным в их основу принципам. Сортировка методом Шелла вполне пригодна для практических приложений даже в случае файлов больших размеров; по эффективности она намного превосходит методы выбора и вставок, равно как и пузырьковую сортировку (см. таблицу 6.1).

<b>N</b>	<b>O</b>	<b>K</b>	<b>G</b>	<b>S</b>	<b>P</b>	<b>I</b>
12500	16	6	6	5	6	6
25000	37	13	11	12	15	10
50000	102	31	30	27	38	26
100000	303	77	60	63	81	58
200000	817	178	137	139	180	126

**Ключи:**

<b>O</b>	1 2 4 8 16 32 64 128 256 512 1024 2048 ...
<b>K</b>	1 4 13 40 121 364 1093 3280 9841 ... (лемма 6.9)
<b>G</b>	1 2 4 10 23 52 113 249 548 1207 2655 5843 ... (упражнение 6.40)
<b>S</b>	1 8 23 77 281 1073 4193 16577 ... (лемма 6.10)
<b>P</b>	1 7 8 49 56 64 343 392 448 512 2401 2744 ... (упражнение 6.44)
<b>I</b>	1 5 19 41 109 209 505 929 2161 3905 ... (упражнение 6.45)

**Лемма 6.11.** Сортировка методом Шелла выполняет менее  $O(N(\log N)^2)$  операций сравнения для последовательности шагов 1 2 3 4 6 9 8 12 18 27 16 24 36 54 81...

Число шагов из треугольника, которое меньше  $N$  по величине, и подавно будет меньше  $(\log_2 N)^2$ .

Последовательности шагов, предложенные Праттом, на практике проявляют тенденцию работать хуже, чем другие, поскольку их слишком много. Мы можем воспользоваться тем же принципом построения последовательностей шагов на базе *лю-бых* двух взаимно простых чисел  $h$  и  $k$ . Такие последовательности шагов показывают хорошие результаты, поскольку границы в худших случаях, соответствующие лемме 6.11, дают завышенную оценку стоимости сортировки файлов с произвольной организацией.

Проблема построения хорошей последовательности шагов для сортировки методом Шелла представляет собой прекрасный пример сложного поведения простых ал-

горитмов. Разумеется, мы, не имея возможности анализа всех алгоритмов, которые будут встречаться далее, на таком же уровне детализации (этому препятствует не только ограниченное пространство данной книги, но, как и в случае сортировки методом Шелла, может произойти так, что потребуются математический анализ, выходящий за рамки материала книги, а, возможно, и специальные исследования). Тем не менее, многие алгоритмы, рассматриваемые в данной книге, являются результатом широких аналитических и эмпирических исследований, выполненных исследователями за несколько последних десятилетий, поэтому можно воспользоваться плодами их трудов. Эти исследования показывают, что задача повышения эффективности сортировки во многих случаях представляет собой интересную научную проблему и часто приносит неплохие практические результаты, даже в случаях простых алгоритмов. В табл. 6.2 приведены эмпирические данные, которые показывают, что некоторые подходы к построению последовательностей шагов хорошо работают на практике; относительно короткая последовательность  $1\ 8\ 23\ 77\ 281\ 1073\ 4193\ 16577\ \dots$  — одна из самых простых, какие используются в программных реализациях сортировки методом Шелла.

Из рис. 6.13. следует, что сортировка методом Шелла обеспечивает хорошие результаты на различных видах файлов и несколько хуже — в случае файлов с произвольной организацией. В самом деле, построение файла, на котором сортировка методом Шелла выполняется медленно на заданной последовательности шагов, — достаточно сложная задача (см. упражнение 6.42). Как уже отмечалось ранее, существуют плохие последовательности шагов, при использовании которых в сортировке методом Шелла требуемое число операций сравнения в худшем случае находится в квадратичной зависимости от размера файла (см. упражнение 6.36), однако доказано, что для широкого набора таких последовательностей эта оценка намного лучше.

Сортировка методом Шелла выбирается для многих приложений, поскольку она обеспечивает приемлемое время сортировки даже для достаточно больших файлов и реализуется в виде компактной программы, которую легко заставить работать. В следующих главах мы ознакомимся с методами сортировки, которые, возможно, более эффективны, но работают всего лишь в два раза быстрее (а то и того меньше) при небольших значениях  $N$ , но при этом они существенно сложнее. Короче говоря, если необходимо решить про-



**РИС. 6.12. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ СОРТИРОВКИ МЕТОДОМ ШЕЛЛА (ДВЕ РАЗЛИЧНЫЕ ПОСЛЕДОВАТЕЛЬНОСТИ ШАГОВ)**

*Представленный на рисунке процесс выполнения сортировки методом Шелла можно сравнить с резиновой лентой, неподвижно закрепленной в противоположных углах, когда все точки ленты устремляются в направлении диагонали. Отображены две последовательности шагов:  $121\ 40\ 13\ 4\ 1$  (слева) и  $209\ 109\ 41\ 19\ 5\ 1$  (справа). Вторая последовательность требует выполнения на один проход больше, но в то же время она выполняется быстрее, поскольку каждый ее проход более эффективен.*

блему сортировки, но нет желания изучать интерфейс системной сортировки, следует воспользоваться *сортировкой методом Шелла*; несколько позже вы решите, есть ли смысл прилагать дополнительные усилия, чтобы заменить его на более совершенный метод сортировки.

## Упражнения

▷ **6.33.** Устойчива ли сортировка методом Шелла?

**6.34.** Показать, как следует построить программу, реализующую сортировку методом Шелла с последовательностью шагов **1 8 23 77 281 1073 4193 16577 ...** с непосредственным вычислением последовательных шагов, используя для этой цели метод, подобный примененному для разработки программы, вычисляющей шаги последовательности Кнута.

▷ **6.35.** Представить диаграммы, соответствующие рис. 6.8 и 6.9 для ключей **E A S Y Q U E S T I O N**.

**6.36.** Определить время выполнения программы сортировки методом Шелла с последовательностью шагов **1 2 4 8 16 32 64 128 264 512 1024 2048 ...** для сортировки файла, состоящего из целых чисел  $1, 2, \dots, N$  на нечетных позициях и  $N + 1, N + 2, \dots, 2N$  на четных позициях.

**6.37.** Написать программу-драйвер, способную выполнять сравнения последовательностей шагов сортировки методом Шелла. Последовательность считываются из стандартного ввода, по одному значению шага в строке; затем используется для сортировки 10 файлов с произвольной организацией длины  $N$ , для  $N = 100, 1000$  и  $10000$ . Подсчитать количество операций сравнения или замерить фактическое время выполнения сортировки для каждого файла.

● **6.38.** Выполните эксперимент с целью определения, позволяет ли добавление или удаление отдельного шага улучшить последовательность шагов **1 8 23 77 281 1073 4103 16577...** для  $N=10000$ .

● **6.39.** Выполните эксперимент с целью определения значения  $x$ , которое обеспечивает минимальное время сортировки случайно распределенных файлов, если в последовательности **1 4 13 40 121 364 1093 3280 9841...** для  $N=10000$  шаг 13 заменить на  $x$ .

**6.40.** Выполните эксперимент с целью определения значения  $\alpha$ , которое обеспечивает минимальное время сортировки файлов с произвольной организацией для последовательности шагов  $1, \lfloor \alpha \rfloor, \lfloor \alpha^2 \rfloor, \lfloor \alpha^3 \rfloor, \lfloor \alpha^4 \rfloor, \dots$ ; где  $N=10000$ .

● **6.41.** Для файлов с произвольной организацией, содержащих 1000 элементов, найти последовательность из трех шагов, которая обеспечивает выполнение минимально возможного числа операций сравнения, какое только удастся обнаружить.

●● **6.42.** Построить файл из 100 элементов, для которого сортировка методом Шелла с последовательностью шагов **1 8 23 77** выполняет максимально возможное количество операций сравнения, какое только удастся найти.

● **6.43.** Доказать, что любое число, большее или равное  $(h-1)(k-1)$ , может быть представлено в виде линейной комбинации (с неотрицательными коэффициентами)  $h$  и  $k$ , если  $h$  и  $k$  — взаимно простые числа. *Совет:* покажите, что если при делении на  $h$  два первых из  $h-1$  кратных  $k$ , имеют один и тот же остаток от деления на  $h$ , то  $h$  и  $k$  должны иметь общий множитель.

**6.44.** Выполните эксперимент с целью определения значений  $h$  и  $k$ , которые обеспечивают наименьшие показатели времени сортировки файлов с произвольной организацией; в качестве шагов для сортировки 10000 элементов используется последовательность, подобная последовательности Пратта, при построении которой использованы  $h$  и  $k$ .

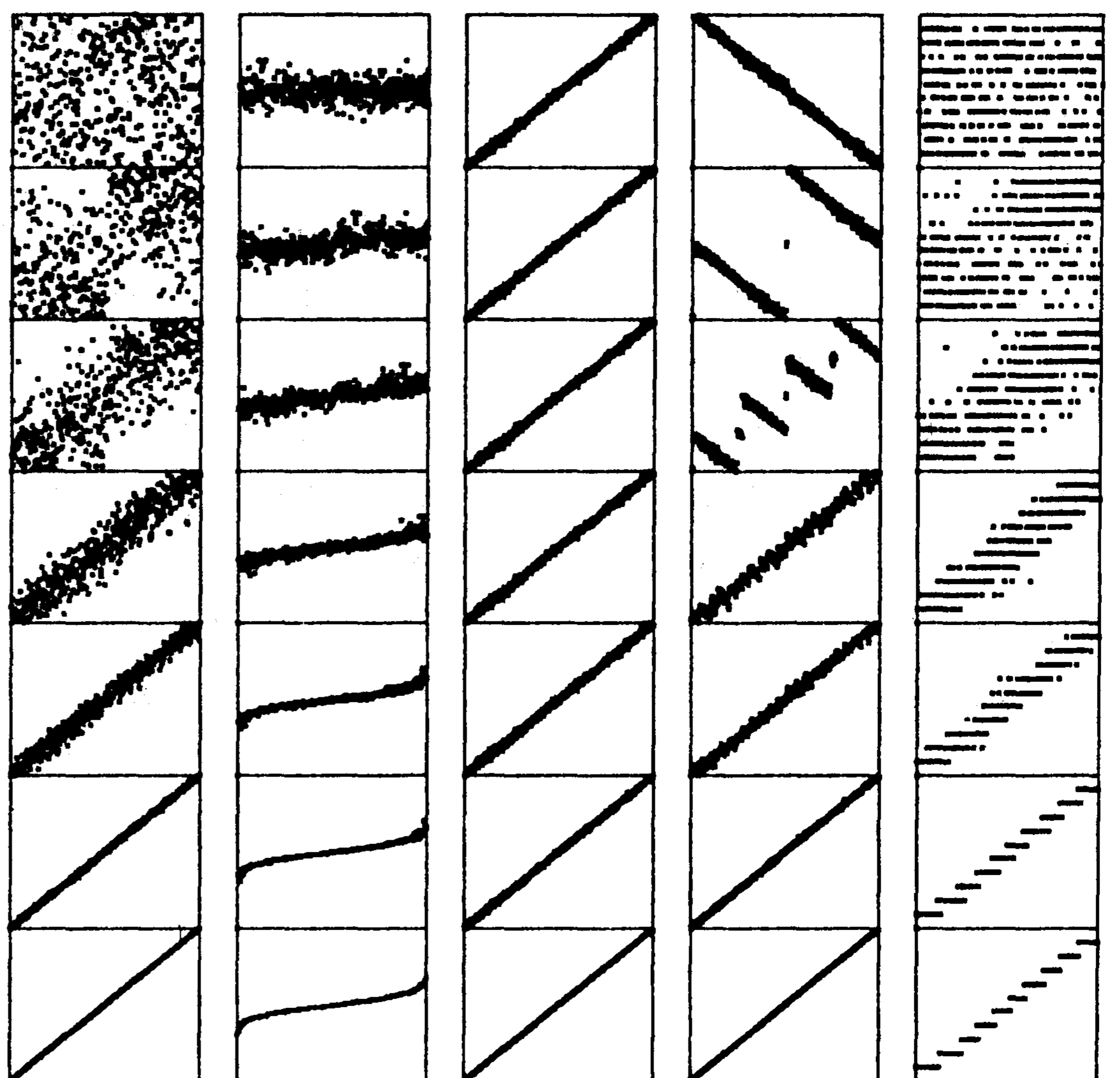
**6.45.** Последовательность шагов **1 5 19 41 109 209 505 929 2161 3905 ...** построена путем слияния последовательностей  $9 \cdot 4^i - 9 \cdot 2^i + 1$  и  $4^i - 3 \cdot 2^i + 1$  для  $i > 0$ . Сравните результаты использования этих последовательностей по отдельности с результатом использования их слияния на примере сортировки 10000 элементов.

**6.46.** Последовательность шагов **1 3 7 21 48 112 336 861 1968 4592 13776...** получена на базе последовательности, состоящей из взаимно простых чисел, скажем, **1 3 7 16 41 101**, с последующим построением треугольника из последовательности Пратта. В данном случае  $i$ -й ряд треугольника получен путем умножения первого элемента в  $i$ -1-ом ряду на  $i$ -й элемент базовой последовательности и путем умножения каждого элемента в  $i$ -1-ом ряду на  $i+1$ -ый элемент базовой последовательности. Проведите эксперименты с целью выявления базовой последовательности, которая превосходит указанную выше в процессе сортировки 10000 элементов.

- **6.47.** Завершите доказательство лемм 6.7 и 6.8.
- **6.48.** Построить реализацию, в основу которой положен алгоритм шейкер-сортировки (упражнение 6.30), и сравнить со стандартным алгоритмом. *Совет:* следует воспользоваться такими последовательностями шагов, которые существенно отличаются от последовательностей, применяемых в стандартных алгоритмах.

### РИСУНОК 6.13. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ СОРТИРОВКИ МЕТОДОМ ШЕЛЛА РАЗЛИЧНЫХ ТИПОВ ФАЙЛОВ

На представленных диаграммах показана сортировка методом Шелла с последовательностью шагов **209 109 41 19 5 1** для файла с произвольной организацией, нормально распределенного файла, практически упорядоченного файла и случайно распределенного файла с 10 различными значениями ключей (слева направо, сверху). Время выполнения каждого прохода зависит от того, насколько упорядочен файл к моменту начала прохода. После выполнения нескольких проходов все эти файлы будут упорядоченными одинаково; таким образом, время выполнения сортировки не очень чувствительно к виду входных данных.



## 6.7. Сортировка других типов данных

Несмотря на то что при исследовании алгоритмов целесообразно рассматривать их как процедуру, выполняющую размещение массивов чисел в порядке их возрастания или размещение символов в алфавитном порядке, необходимо признать, что эти алгоритмы, в основном, не зависят от типа сортируемых элементов, в связи с чем нетрудно сделать определенные обобщения. Мы подробно рассуждали о том, как разбить программы на отдельные независимые модули, ориентированные на работу с конкретными типами данных, а также о работе с абстрактными типами данных (см. главы 3 и 4); в настоящем разделе обсуждаются способы применения рассмотренных понятий с целью построения программных реализаций, интерфейсов и клиентских программ для алгоритмов сортировки. В частности, будут рассматриваться интерфейсы для:

- *элементов* или обобщенных объектов, подлежащих сортировке
- *массивов* элементов.

Тип данных **item** (элемент) предоставляет возможность использования программы сортировки для любых типов данных из числа тех, для которых определены некоторые виды базовых операций. Такой подход эффективен не только для простых, но и для абстрактных типов данных, в связи с чем мы перейдем к изучению множества программных реализаций. Интерфейс **array** (массив) менее критичен для нашей задачи; мы остановимся на нем с тем, чтобы получить навыки работы с многомодульной программой, которая имеет дело с несколькими типами данных. Мы рассмотрим только одну из (достаточно простую) программных реализаций интерфейса **array**.

Программа 6.6 является клиентской программой, снабженной теми же общими функциональными средствами, что и функция **main** из программы 6.1, и дополненной возможностями манипуляции массивами и элементами, инкапсулированными в отдельных модулях. Это обеспечивает, в частности, возможность тестирования программ сортировки на различных типах данных путем замены одних модулей на другие, не внося при этом каких-либо изменений в клиентскую программу. Программа 6.6 обращается к интерфейсу с целью выполнить операции **exch** и **comprexch**, используемые программными реализациями различных видов сортировки. Можно было бы настоять на том, чтобы эти программные средства были включены в интерфейс **Item.h**, однако реализации сортировок в программе 6.1, благодаря легко понимаемой семантике при определении операторов присваивания и оператора **operator<**, вполне подходят для нашей цели, так что проще содержать их в одном модуле, где они могут быть использованы всеми построенными реализациями для всех выбранных типов данных **item**. В завершение программной реализации следует дать точное определение интерфейсов **array** и типа данных **item** и только после этого предлагать собственные программные реализации.

### Программа 6.6. Драйвер сортировки массивов

Данный драйвер базовых сортировок массивов использует три явных интерфейса: первый для типа данных, который инкапсулирует операции, выполняемые на обобщенных элементах; следующий предназначен для функций **exch** и **comprexch** несколько более высокого уровня, которые используются в программных реализациях сортировок; и третий — для функций, которые инициализируют и печатают (а также

сортируют!) массивы. Разбивая драйвер на соответствующие модули, получаем возможность применить каждую реализацию сортировки к различным типам данных без необходимости внесения в программы каких-либо изменений, совместно использовать реализации операций обмена и сравнения-обмена, а также компилировать эти функции отдельно для массивов (возможно, для их использования в других программах-драйверах).

```
#include <stdlib.h>
#include "Item.h"
#include "exch.h"
#include "Array.h"
main(int argc, char *argv[])
{ int N = atoi(argv[1]), sw = atoi(argv[2]);
  Item *a = new Item[N];
  if (sw) rand(a, N); else scan(a, N);
  sort(a, 0, N-1);
  show(a, 0, N-1);
}
```

---

### Программа 6.7. Интерфейс для данных типа массив.

---

Интерфейс **Array.h** определяет высокоуровневые функции для массивов абстрактных элементов: инициализирует случайными значениями, инициализирует значениями, считанными из стандартного ввода, распечатывает содержимое и выполняет сортировку содержимого.

```
template <class Item>
  void rand(Item a[], int N);
template <class Item>
  void scan(Item a[], int &N);
template <class Item>
  void show(Item a[], int l, int r);
template <class Item>
  void sort(Item a[], int l, int r);
```

---

Интерфейс программы 6.7. определяет примеры высокоуровневых операций, которые можно выполнять над массивами. Требуется реализовать возможность инициализировать массив значениями ключей, которые имеют случайное распределение или поступают из стандартного ввода; кроме того необходимо иметь возможность сортировать вхождения (а как же!) и, наконец, иметь возможность печатать содержимое массивов. Это только лишь несколько примеров; в конкретном приложении может возникнуть необходимость определять и другие операции (класс **Vector** в библиотеке стандартных шаблонов является одним из подходов, обеспечивающим обобщенный интерфейс этого вида). Пользуясь программой 6.7, можно подставлять различные реализации той или иной операции без необходимости внесения изменений в клиентскую программу, которая использует этот интерфейс — в данном случае, в функцию **main** программы 6.6. Исследуемые здесь различные реализации сортировок могут служить реализациями функции **sort**. В программе 6.8 имеются простые реализации других функций. Модульная организация программы позволяет подставлять другие реализации в зависимости от приложения. Например, можно воспользоваться реализацией функции **show**, которая печатает только часть массива при проверке сортировки массивов очень больших размеров.



**Программа 6.8. Реализация типа данных массива**

Приведенный ниже код представляет собой реализацию функций, определенных в программе 6.7. Эта реализация использует типы `Item` и обрабатывающие их базовые функции, которые определены в отдельном интерфейсе (см. программу 6.9).

```
#include <iostream.h>
#include <stdlib.h>
#include "Array.h"
template <class Item>
void rand(Item a[], int N)
{ for (int i = 0; i < N; i++) rand(a[i]); }
template <class Item>
void scan(Item a[], int &N)
{ for (int i = 0; i < N; i++)
  if (!scan(a[i])) break;
  N = i;
}
template <class Item>
void show(Item a[], int l, int r)
{ for (int i = l; i <=r; i++)
  show(a[i]);
  cout << endl;
}
```

Подобным же образом, чтобы иметь возможность работать с конкретными типами элементов и ключей, мы даем определения их типов и объявляем все необходимые операции над ними в явном интерфейсе, а затем следуют реализации этих операций, определенных в интерфейсе элемента. Например, рассмотрим приложение в виде некоторого бухгалтерского отчета, в котором могут быть использованы ключи, соответствующие номеру счета клиента, и числа с плавающей точкой, соответствующие сумме на счетах клиента. Программа 6.9 являет собой пример интерфейса, который определяет тип данных для такого рода приложения. Код этого интерфейса объявляет операцию `<`, которая нужна для сравнения ключей, а также для функций, которые генерируют случайные значения ключей, считывают ключи и выводят значения ключей на печать. В программе 6.10 содержатся реализации функций, используемых в этом простом примере. Разумеется, можно приспособить эти реализации под конкретные приложения. Например, `Item` может иметь абстрактный тип данных (АТД)), определенный в виде класса C++, а ключи могут быть функциями-членами класса, а не элементами данных соответствующей структуры. Такие АТД рассматриваются в главе 12.

**Программа 6.9. Пример интерфейса для данных типа `Item`**

Файл `Item.h`, включенный в программу 6.6, дает определение типа данных сортируемых элементов. В этом примере элементами являются небольшие записи, состоящие из целочисленных ключей, и связанная с ними информация, представленная числами с плавающей точкой. Мы объявляем, что перегруженная операция `operator<` будет реализована отдельно, равно как и три функции: `scan` (считывает `Item` в своем аргументе), `rand` (сохраняет случайный `Item` в своем аргументе), `show` (печатает `Item`).

```
typedef struct record { int key; float info; } Item;
int operator<(const Item&, const Item&);
int scan(Item&);
void rand(Item&);
void show(const Item&);
```

Программы 6.6—6.10 вместе с любыми подпрограммами сортировки *из числа представленных* в разделах 6.2—6.6, тестирует сортировку небольших по размерам записей. Построение такого рода интерфейсов и реализаций для других типов данных позволяет применять рассмотренные выше методы для сортировки различных видов данных — таких как комплексные числа (см. упражнение 6.50), векторы (см. упражнение 6.55) или полиномы (см. упражнение 6.56) — и при этом вообще без каких-либо изменений в кодах программ сортировки. Для более сложных типов элементов интерфейсы и реализации также должны быть более сложными, однако решение проблем реализации полностью отделено от вопросов построения алгоритмов сортировки, которые были предметом наших исследований. Одни и те же механизмы можно задействовать для большинства методов сортировки, рассматриваемых в данной главе, а также тех, которые будут изучаться в главах 7—9. В разделе 6.10 мы подробно проанализируем одно очень важное исключение — в результате анализа станет ясно, что целое семейство алгоритмов сортировки, которое должно иметь совсем другое конструктивное оформление, заслуживает того, чтобы стать предметом изучения в главе 10.

Рассмотренный в этом разделе подход является своего рода средним путем между программой 6.1 и имеющими перспективы промышленного применения полностью абстрактным набором реализаций, дополненных средствами выявления ошибок, управления памятью и даже более универсальными возможностями. Подобного рода проблемы конструктивного оформления программ приобретают все большую актуальность в некоторых современных областях программирования и приложений. Ряд вопросов придется оставить без ответа. Основная цель заключается в том, чтобы используя сравнительно простые механизмы из изученных показать, что программные реализации исследуемых сортировок находят широкое применение.

---

### Программа 6.10. Пример реализации типа данных

---

Этот программный код является реализацией перегруженной операции **operator<** и функций **scan**, **rand** и **show**, которые объявлены в программе 6.9. Поскольку запись представляет собой структуры небольших размеров, можно сделать так, чтобы функция **exch** использовала встроенный оператор присваивания без ненужных забот относительно затрат на копирование элементов.

```
#include <iostream.h>
#include <stdlib.h>
#include "Item.h"
int operator<(const Item& A, const Item& B)
    { return A.key < B.key; }
int scan(Item& x)
    { return (cin >> x.key >> x.info) != 0; }
void rand(Item& x)
    { x.key = 1000*(1.0*rand()/RAND_MAX);
      x.info = 1.0*rand()/RAND_MAX; }
void show(const Item& x)
    { cout << x.key << " " << x.info << endl; }
```

---

## Упражнения

- 6.49. Напишите свою версию программ 6.9 и 6.10, в которых выполняется перегрузка операций `operator<<` и `operator>>` вместо того, чтобы пользоваться функциями `scan` и `show`, а также внесите соответствующие изменения в программу 6.8, обеспечивающие использование созданного интерфейса.
- 6.50. Напишите интерфейс и реализацию типа данных обобщенных элементов для поддержки метода сортировки комплексных чисел  $x + iy$ , использующих в качестве ключа модуль  $\sqrt{x^2 + y^2}$ . *Совет:* игнорирование операции извлечения квадратного корня, по-видимому, повысит эффективность.
- 6.51. Напишите интерфейс, который дает определение абстрактного типа данных первого класса для обобщенных элементов (см. раздел 4.8) и построить реализацию, в которой, как и в предыдущем упражнении, элементами являются комплексные числа. Протестируйте созданный интерфейс с программами 6.3 и 6.6.
- ▷ 6.52. Добавить функцию `check` в тип данных `array` в программах 6.8 и 6.9, которые проверяют, упорядочен ли массив с помощью сортировки.
- 6.53. Добавить функцию `testinit` в тип данных `array` в программах 6.8 и 6.7, которая генерирует данные для тестирования в соответствии с распределениями, подобными показанным на рис. 6.13. Ввести целочисленный аргумент, посредством которого клиентская программа сможет выбирать соответствующее распределение.
- 6.54. Внести в программы 6.7 и 6.8 такие изменения, которые позволили бы реализовать тип данных `abstract`. (Ваша реализация должна распределять и поддерживать массив именно так, как это делают наши реализации стеков и очередей в главе 3.)
- 6.55. Напишите интерфейс и реализацию для обобщенного типа данных `Item` с таким расчетом, чтобы известные методы сортировки можно было использовать с целью сортировки многомерных векторов из  $d$  целых чисел, размещая их в таком порядке: сначала идут векторы с наименьшей первой компонентой, из векторов с равными первыми компонентами первым идет тот, у которого меньше вторая компонента, из векторов с равными первыми и вторыми компонентами первым идет тот, у которого меньше третья компонента и т.д.
- 6.56. Напишите интерфейс и реализацию для обобщенного типа данных `Item` с таким расчетом, чтобы известные методы сортировки можно было использовать для сортировки полиномов (см. раздел 4.9). В рамках этой задачи потребуется определить соответствующий порядок.

## 6.8 Сортировка по индексам и указателям

Разработка данных типа строка, подобного используемым в программах 6.9 и 6.10, представляет особый интерес, поскольку строки символов широко используются как ключи в процессе сортировки. Более того, поскольку строки могут иметь различные длины и вообще быть очень длинными, построение, удаление и сравнение строк может потребовать значительных затрат ресурсов, так что следует проявить особую осторожность и позаботиться о том, чтобы выбранная реализация не приводила к излишним и необязательным операциям этого вида.

С этой целью применяется такое представление данных, которое состоит из указателя (на массив символов), — это стандартное представление строки в стиле языка С. Далее, первая строка программы 6.9 изменяется на

```
typedef struct { char *str; } Item;
```

что обеспечивает ее преобразование в интерфейс для строк. Этот указатель помещается в структуру `struct`, поскольку С++ не позволяет перегружать операцию `operator<` для встроенных типов, каковыми являются указатели. Подобного рода ситуации не являются чем-то необычным в С++: класс (или `struct`), который приспособливает интерфейс для другого типа данных называется *классом-оболочкой*. В рассматриваемом случае мы не требуем слишком многого от класса-оболочки, тем не менее, в некоторых случаях возможны более сложные реализации. Вскоре будет рассмотрен еще один пример.

Программа 6.11 представляет собой реализацию, ориентированную на строковые элементы. Перегруженная операция `operator<` легко реализуется с помощью функции сравнения строк из библиотеки С, но реализация функций `scan` (и `rand`) представляет собой более трудную задачу, поскольку нельзя упускать из виду распределение памяти для строк. Программа 6.11 использует метод, который изучался в главе 3 (программа 3.17), содержащий буфер в реализации этого типа данных. Другие варианты предусматривают динамическое распределение памяти для каждой строки, использование реализации класса, подобного классу `String` из библиотеки стандартных шаблонов, либо организацию буфера в клиентской программе. Можно воспользоваться любым из этих подходов (с соответствующими интерфейсами) для сортировки строк символов, используя любую из реализаций сортировки из рассмотренных выше.

Мы сталкиваемся с необходимостью подобного выбора управления памятью всякий раз, когда пытаемся придавать программе модульную структуру. Кто должен нести ответственность за управление памятью, соответствующее конкретной реализации некоторых типов объектов: клиентская программа, реализация типа данных или система? Не существует однозначного и готового ответа на этот вопрос (некоторые разработчики языков программирования вообще становятся суеверными, когда этот вопрос возникает). Некоторые из современных систем программирования (включая конкретные реализации С++) содержат обобщенные механизмы, обеспечивающие автоматическое управление памятью. Мы еще раз столкнемся с этой проблемой в главе 9, когда приступим к изучению реализации более сложных абстрактных типов данных.

Программа 6.11 представляет собой пример *сортировки по указателю*, к рассмотрению которой в обобщенном виде мы сейчас перейдем. Другой простой подход к проблеме сортировки без (непосредственных) перемещений элементов заключается в построении *индексного массива*, причем доступ к ключам элементов осуществляется только для того, чтобы выполнить операцию сравнения. Предположим, что сортируемые элементы находятся в массиве `data[0], ..., data[N-1]` и мы не хотим перемещать их в силу тех или иных причин (возможно, из-за их огромных размеров). Чтобы получить эффект сортировки, используется *второй массив*, массив `a` индексов элементов. Мы начинаем с инициализации `a[i]` значениями `i` для `i=0, ..., N-1`. Другими словами, мы начинаем с того, что `a[0]` получает значение индекса первого элемента данных,

**a[1]** — второго элемента данных и т.д. Целью сортировки является переупорядочение массива индексов таким образом, что **a[0]** дает индекс элемента данных с наименьшим значением ключа, **a[1]** — индекс элемента данных с минимальным значением ключа из числа оставшихся и т.д. После этого эффект сортировки достигается за счет доступа к ключам через индексы — например, таким способом можно вывести массив в порядке выполненной сортировки.

### Программа 6.11. Реализация типов данных для строковых элементов

Эта реализация позволяет выполнять сортировку строк в языке C. Для представления данных используется структура, которая содержит указатель на символ (см текст программы), благодаря чему сортировка осуществляется для массива указателей на символы, переупорядочивая их таким образом, что строки, на которые они указывают, следуют друг за другом в алфавитно-цифровом порядке. Чтобы подробно показать процесс управления памятью, мы даем определение буфера памяти фиксированных размеров, в который данный модуль помещает символы сортируемых строк; по-видимому, динамическое распределение памяти подходит больше. Реализация функции **rand** здесь опущена.

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
static char buf[100000];
static int cnt = 0;
int operator<(const Item& a, const Item& b)
    { return strcmp(a.str, b.str) < 0; }
void show(const Item& x)
    { cout << x.str << " "; }
int scan(Item& x)
    { int flag = (cin >> (x.str = &buf[cnt])) != 0;
      cnt += strlen(x.str)+1;
      return flag;
    }
```

Требуется точно определить, что выбранный вид сортировки выполняет упорядочивание массива индексов, а не простые целые числа. Тип **Index** следует определить так, чтобы можно было перегрузить операцию **operator<** следующим образом:

```
int operator<(const Index& i, const Index& j)
    { return data[i] < data[j]; }
```

Если мы получим в свое распоряжение массив объектов типа **Index**, то любая задействованная функция сортировки так переупорядочит индексы в массиве **a**, что значение **a[i]** определит число ключей, меньших по значению, чем ключ элемента **data[i]** (индекс **a[i]** в отсортированном массиве). (Для простоты в этом обсуждении предполагается, что данные суть ключи, а не элементы в полном объеме — этот принцип можно распространить на более крупные и сложные элементы, внося в операцию **operator<** такие изменения, которые позволяют осуществлять доступ к специфическим ключам таких элементов, либо воспользоваться функцией-членом класса для вычисления такого ключа.) Для определения объектов типа **Index** используется класс-оболочка:

```

struct intWrapper
{
    int item;
    intWrapper(int i = 0)
        { item = i; }
    operator int() const
        {return item; }
};
type def intWrapper Index;

```

Конструктор в данной структуре `struct` преобразует любое значение типа `int` в `Index`, а операция преобразования типа `operator int()` преобразует любое значение `Index` обратно в `int`, следовательно, объекты типа `Index` можно использовать везде, где допускается применение объектов встроенного типа `int`.

Пример индексации, в рамках которого одни и те же элементы сортируются по двум различным ключам, представлен на рис. 6.14. Одна клиентская программа может определить `operator<` для работы с одним видом ключей, а другая — для использования с другим ключом, при этом оба они могут воспользоваться одной и той же программой сортировки, чтобы построить массив индексов, который позволил бы получать доступ к элементам в порядке расположения их ключей.

Переупорядочение  $N$  различных неотрицательных целых чисел, меньших  $N$ , в математике называется перестановкой: индексная сортировка выполняет перестановку. В математике перестановки обычно определяются как переупорядочения целых чисел от 1 до  $N$ ; мы же будем употреблять числа от 0 до  $N - 1$ , чтобы подчеркнуть прямую связь между перестановками и массивом индексов в C++.

Такой подход, предусматривающий использование массива индексов вместо реальных элементов, работает в любом языке программирования, который поддерживает массивы. Другая возможность заключается в использовании указателей; она аналогична только что рассмотренной реализации строкового типа данных (программа 6.11). Будучи выполненной на массиве элементов фиксированного размера, сортировка по указателям во многом эквивалентна индексной сортировке, но при этом адрес массива добавляется к каждому индексу. Но сортировка по указателям — это более общая форма сортировки, ибо указатели могут указывать на что угодно, а элементы, подвергающиеся сортировке, отнюдь не обязательно должны иметь фиксированные размеры. Для индексной сортировки характерно следующее: если `a` есть массив указателей на ключи, то результатом вызова функции `sort` будет переупорядочение указателей таким образом, что пос-

0	10	9	Wilson	63
1	4	2	Johnson	86
2	5	1	Jones	87
3	6	0	Smith	90
4	8	4	Washington	84
5	7	8	Thompson	65
6	2	3	Brown	82
7	3	10	Jackson	61
8	9	6	White	76
9	0	5	Adams	86
10	1	7	Black	71

#### РИСУНОК 6.14. ПРИМЕР ИНДЕКСНОЙ СОРТИРОВКИ

*Манипулируя индексами, а не самими записями, можно выполнять сортировку одновременно по нескольким ключам. В этом примере данные могут быть фамилиями студентов и их степенями, вторая колонка представляет собой результат индексной сортировки по именам, а третья колонка представляет собой результат индексной сортировки по степени. Например, **Wilson** — фамилия, идущая по алфавиту последней, и ей соответствует десятая степень, в то время как фамилия **Adams** идет первой в алфавитном порядке, а ей соответствует шестая степень.*

ледовательный доступ к ним означает доступ к ключам в соответствующем порядке. Мы выполняем операции сравнения, следуя указателям; мы реализуем операции обмена за счет перемены местами указателей.

Функция `qsort` из стандартной библиотеки C представляет собой сортировку по указателям (см. программу 3.17), которая принимает функцию сравнения в качестве аргумента (но не берет за основу перегруженную операцию `operator<`, что делалось ранее). У этой функции четыре аргумента: массив, количество сортируемых элементов, размеры элементов и указатель на функцию, которая выполняет сравнение двух элементов, причем для этого ей потребуется передать указатели на эти элементы. Например, если `Item` есть `char*`, то приведенный ниже программный код выполняет сортировку строк в соответствии с принятыми соглашениями:

```
int compare (void *i, void *j)
    { return strcmp(*(Item *)i, *(Item *)j); }
void sort (Item a[], int l, int r)
    { qsort(a, r-l+1, sizeof(Item), compare); }
```

Положенный в основу этого кода алгоритм в интерфейсе не определен, хотя быстрая сортировка (см. главу 7) находит широкое применение. В главе 7 мы рассмотрим многие причины, почему это так. Благодаря материалу данной главы, а также глав с 7 по 11, станет понятно, почему в условиях некоторых специальных приложений целесообразно применение ряда других методов сортировки. Кроме того будут исследоваться подходы к ускорению вычислений в тех случаях, когда время выполнения сортировки является критическим фактором для приложения.

### Программа 6.12. Интерфейс типа данных для элементов типа запись

В записях имеются два ключа: ключ строкового типа (например, фамилия) в первом поле и целое число (например, степень) — во втором. Будем считать, что эти записи слишком большие, чтобы их копировать, поэтому `Item` определяется как структура `struct`, содержащая указатель на запись.

```
struct record { char name[30]; int num; };
typedef struct { record *r; } Item;
int operator<(const Item&, const Item&);
void rand(Item&);
void show(const Item&);
int scan(Item&);
```

### Программа 6.13. Реализация типа данных записей

Приведенные ниже реализации функций `scan` и `show` для записей работают в стиле реализации строкового типа данных из программы 6.11, распределяя и работая с памятью, в которой хранятся строки. Реализация операции `operator<` находится в отдельном файле, что дает возможность изменять представления и ключи сортировки без затрагивания основного кода.

```
static record data[maxN];
static int cnt = 0;
void show(const Item& x)
    { cout << x.r->name << " " << x.r->num << endl; }
int scan(Item& x)
    {
        x.r = &data[cnt++];
        return (cin >> x.r->name >> x.r->num) != 0;
    }
```

В обычных приложениях указатели используются для доступа к записям, которые могут иметь несколько ключей. Например, записи, содержащие фамилии студентов со степенями или фамилии людей вместе с возрастом могут определяться следующим образом:

```
struct record { char[30] name; int num; }
```

Вполне может потребоваться выполнить их сортировку, используя любое из этих полей в качестве ключа. Программы 6.12 и 6.13 могут служить примерами интерфейса сортировки по указателям и реализации, которая позволила бы достигнуть этого. Для различных приложений сортировки используются массив указателей на записи и различные реализации операции `operator<`. Например, если откомпилировать программу 6.13 вместе с файлом, содержащим программный код

```
#include "Item.h"
int operator<(const Item &a, const Item &b)
{ return a.r->num < b.r->num; }
```

то получим тип данных для элементов, для которых любая реализация функции `sort` выполнит сортировку по указателям на целочисленном поле; а если откомпилировать программу 6.13 вместе с файлом, содержащим программный код

```
#include "Item.h"
#include <string.h>
int operator<(const Item &a, const Item &b)
{ return strcmp(a.r->name, b.r->name) < 0; }
```

то получим тип данных для элемента, для которого любая реализация функции `sort` выполнит сортировку по указателям на строковом поле.

Основная причина использования индексов или указателей состоит в том, чтобы не затрагивать данных, подвергаемых сортировке. Можно "сортировать" файл даже в том случае, когда к нему разрешен доступ "только для чтения". Более того, используя несколько индексных массивов или массивов указателей, можно сортировать один и тот же файл по многим ключам (см. рис. 6.14). Подобного рода гибкость, позволяющая манипулировать данными, сохраняя их неизменными, полезна во многих приложениях.

Другая причина, побуждающая манипулировать индексами, заключается в том, что таким путем можно избежать расходов на перемещения записей целиком. Достигается значительная экономия, если записи большие (а ключи маленькие), поскольку для выполнения операции сравнения требуется доступ только к небольшой части записи, а большая часть записи в процессе выполнения сортировки не затрагивается. При непрямом подходе стоимость операции обмена примерно равна стоимости операции сравнения в общем случае, когда сравниваются записи произвольной длины (за счет расходов на дополнительное пространство памяти, выделяемое под индексы или указатели). В самом деле, если ключи длинные, то операции обмена могут быть даже не такими дорогостоящими, как операции сравнения. При получении оценки времени выполнения сортировки целочисленных файлов тем или иным методом, часто предполагается, что стоимости операций сравнения и обмена приблизительно одного порядка. Выводы, полученные на основе такого предположения, можно, по-видимому, отнести к широкому классу приложений в тех случаях, когда применяется сортировка по указателям или индексная сортировка.



Для многих приложений никогда не возникает необходимость физического перемещения данных в соответствии с порядком размещения соответствующих индексов, и можно получить к ним доступ в нужном порядке, используя индексные массивы. Если в силу некоторых причин такой подход не устраивает, возникнет необходимость решить обычную проблему из области классического программирования: каким образом сделать так, чтобы записи файла, подвергаемого индексной сортировке, заняли свои места в соответствующем порядке? Программный код

```
for (i=0; i < N; i++) datasorted[i] =
    data[a[i]];
```

тривиален, однако требует дополнительного пространства памяти, достаточного для размещения еще одной копии массива. Что делать в том случае, когда для второй копии файла в памяти не хватает места? Ведь нельзя же просто записать `data[i] = data[a[i]]`, ибо в этом случае предыдущее значение `data[i]` окажется затертым, и скорее всего, преждевременно.

На рис. 6.15 показан способ решения этой проблемы, все еще обходясь одним проходом по файлу. Чтобы переместить первый элемент на положенное ему место, мы переносим элемент, который находится на его месте, на положенное ему место и т.д. Продолжая подобные рассуждения, мы в конце концов находим элемент, который требуется передвинуть на первую позицию; на этом этапе в окончательные позиции оказывается сдвинутым некоторый цикл элементов. Далее, мы переходим ко второму элементу и выполняем те же операции для его цикла и так далее (любые элементы, с которыми мы сталкиваемся и которые находятся в своих окончательных позициях ( $a[i] = i$ ), попадают в цикл длиной 1 и не перемещаются).

В частности, для каждого значения  $i$  сохраняется значение `data[i]` и инициализируется индексная переменная  $k$  значением  $i$ . Далее мы обращаем свое внимание к образовавшемуся пустому месту в позиции  $i$  и ищем элемент, который должен заполнить это место. Таким элементом является `data[a[k]]` — другими словами, операция присваивания `data[k] = data[a[k]]` перемещает это пустое место в `a[k]`. Теперь пус-

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
A S O R T I N G E X A M P L E
0 10 8 14 7 5 13 11 6 2 12 3 1 4 9
```

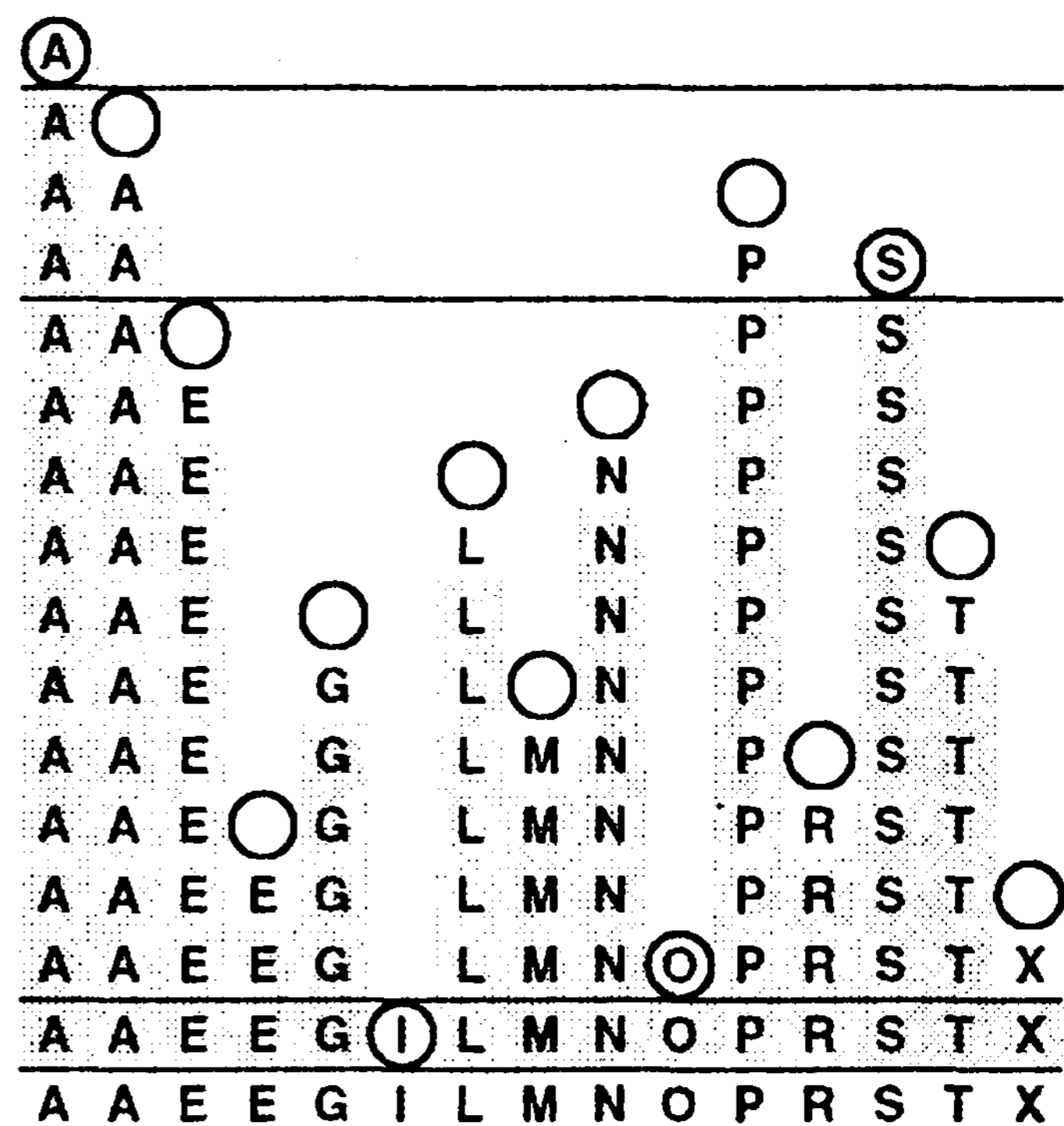


РИСУНОК 6.15. ОБМЕННАЯ СОРТИРОВКА

Чтобы выполнить обменное упорядочение массива (упорядочение на месте), мы перемещаемся по нему слева направо, в циклах передвигая элементы, которые требуется переместить. В

рассматриваемом примере имеются четыре цикла: первый и последний суть вырожденные одноэлементные циклы. Второй цикл начинается с позиции 1.

Элемент *S* переходит во временную переменную, оставляя в позиции 1 пустое место. Перемещение второго *A* приводит к тому, что в позиции 10 остается пустое место. Это пустое место заполняется элементом *P*, который, в свою очередь, оставляет пустое место в позиции 12. Это пустое место должно быть заполнено элементом в позиции 1, следовательно, запомненный элемент *S* переходит в это пустое место, тем самым завершая цикл 1 10 12, который устанавливает эти элементы в окончательные позиции. Аналогично выполняется цикл 2 8 6 13 4 7 11 3 14 9, который и завершает сортировку.

тое место возникает в позиции `data[a[k]]`, таким образом, `k` устанавливается в `a[k]`. Повторяя эти действия, мы в конце концов оказываемся в ситуации, когда пустое место должно быть заполнено значением `data[i]`, которое было предварительно сохранено. Когда мы помещаем элемент в какую-либо позицию, мы вносим соответствующие изменения в массив `a`. Для любого элемента, занявшего свою позицию, `a[k]` равен `i`, и только что написанный процесс сводится к отсутствию операции (no-op). Перемещаясь по массиву и начиная новый цикл всякий раз, когда встречается элемент, который еще ни разу не перемещался, мы перемещаем каждый элемент самое большее один раз. Программа 6.14 представляет реализацию рассмотренного процесса.

### Программа 6.14. Обменная сортировка

Массив `data[0],...,data[N-1]` должен быть упорядочен на месте в соответствии с массивом индексов `a[0],...,a[N-1]`. Любой элемент, для которого `a[i] == i`, занимает свое окончательное место и его больше не следует трогать. В противном случае следует сохранить `data[i]` в `v` и работать в цикле `a[i], a[a[i]], a[a[a[i]]]` и т.д. до тех пор, пока индекс `i` не встретится опять. Мы повторяем этот процесс для следующего элемента, который не на месте, и продолжаем в том же духе, пока не приведем в порядок весь файл, причем каждая запись будет перемещаться только один раз.

```
template <class Item>
void insitu(Item data[], Index a[], int N)
{ for (int i = 0; i < N; i++)
  { Item v = data[i];
    int j, k;
    for (k = i; a[k] != i; k = a[j], a[j] = j)
      { j = k; data[k] = data[a[k]]; }
    data[k] = v; a[k] = k;
  }
}
```

Этот процесс называется *перестановкой по месту (in situ)* или *обменное упорядочение* файла. Отметим еще раз: несмотря на то что сам по себе алгоритм весьма интересен, тем не менее, во многих приложениях в нем нет необходимости, ибо вполне достаточно непрямого доступа к элементам. Кроме того, если записи несопоставимо велики по отношению к их номерам, наиболее эффективным может оказаться вариант их упорядочения при помощи обыкновенной сортировки выбором (см. лемму 6.5).

Непрямая сортировка требует дополнительного пространства памяти для размещения массива индексов или массива указателей и дополнительного времени для выполнения операций непрямого сравнения. Во многих приложениях эти затраты являются вполне оправданной ценой за гибкость и возможность вообще не затрагивать записи. В случае файлов, состоящих из больших записей, мы практически всегда будем пользоваться непрямой сортировкой, а во многих приложениях часто происходит так, что вообще нет необходимости перемещать данные. В рамках этой книги обычно применяется прямой доступ к данным. Однако в некоторых приложениях все-таки придется воспользоваться массивами индексов и указателей во избежание перемещения данных; именно в силу этих причин мы и остановились здесь на этой теме.

## Упражнения

- 6.57. Представить реализацию типа данных для элементов, когда элементами являются записи, а не указатели на записи. Такая организация данных может оказаться предпочтительной для применения программ 6.12 и 6.13 к небольшим записям. (Не забывайте, что язык C++ поддерживает операцию присваивания структур.)
- 6.58. Показать, как можно использовать функцию `qsort` для решения проблем сортировки, на которые ориентированы программы 6.12 и 6.13.
  - ▷ 6.59. Построить массив индексов, который получается при индексной сортировке ключей `EASYQUESTION`.
  - ▷ 6.60. Построить последовательность перемещений данных, необходимых для перестановки ключей `EASYQUESTION` на месте после выполнения индексной сортировки (см. упражнение 6.59).
- 6.61. Опишите перестановку размера  $N$  (набор значений для массива `a`), в которой условие `a[i] != i` в процессе работы программы 6.14 выполняется максимальное число раз.
- 6.62. Доказать, что в процессе перемещения ключей и появления в программе 6.14 пустых мест мы обязательно вернемся к ключу, с которого начинали.
- 6.63. Реализовать программу, аналогичную программе 6.14, с сортировкой по указателям в предположении, что указатели указывают на массив из  $N$  записей типа `Item`.

## 6.9. Сортировка связанных списков

Как стало известно из главы 3, массивы и связанные списки представляют собой два базовых способа структурирования данных, поэтому рассмотрим реализацию сортировки вставками связанных списков как пример обработки списков, о которой шла речь в разделе 3.4 (программа 3.11). Все программные реализации сортировок, рассмотренные к этому моменту, предполагают, что сортируемые данные представлены в виде массивов, в связи с чем они не могут быть использованы непосредственно, если мы работаем в рамках системы, которая использует связанные списки для организации данных. В некоторых случаях могут оказаться полезными специальные алгоритмы, но только тогда, когда они по существу выполняют последовательную обработку данных, которую можно эффективно поддерживать для связанных списков.

### Программа 6.15. Определение интерфейса для типа связанного списка

Данный интерфейс для связанных списков может быть сопоставлен с интерфейсом для массивов, представленным в программе 6.7. Функция `randomlist` строит список случайно распределенных элементов с одновременным выделением для них памяти. Функция `showlist` выполняет печать ключей из этого списка. Программы сортировки используют перегруженную операцию `operator<` для сравнения и манипулирования указателями с целью упорядочения элементов. Представление данных в узлах реализуется обычным способом (см. главу 3) и включает конструктор узлов, который заполняет каждый новый узел заданными значениями и наделяет фиктивной связью.

```
struct node
{ Item item; node* next;
```

```

node(Item x)
{ item = x; next = 0; }
};
typedef node *link;
link randlist(int);
link scanlist(int&);
void showlist(link);
link sortlist(link);

```

Программа 6.15 задает интерфейс типа данных **linked-list** (связный список), подобный используемому в программе 6.7. В условиях программы 6.15 драйвер, соответствующий программе 6.6, умещается в одной строке:

```

main(int argc, char *argv[])
{ showlist(sortlist(scanlist(atoi(argv[1])))); }

```

Большая часть работы (включая и распределение памяти) ложится на реализации связного списка и функции **sort**. Так же как и в случае драйвера для массива, этот список должен инициализироваться (из стандартного ввода либо случайными значениями), необходимо уметь отобразить его содержимое и, разумеется, отсортировать его. Как обычно, в качестве типа данных сортируемых элементов используется **Item**, что предпринималось в разделе 6.7. Программный код, реализующий интерфейс подобного рода, является стандартным для связных списков и аналогичен коду, который подробно исследовался в главе 3; ниже приводится соответствующее упражнение.

Рассматриваемый интерфейс — это низкоуровневый интерфейс, который не делает различий между связью (указатель на узел) и связным списком (указатель, значение которого есть 0, или указатель на узел, содержащий указатель на список). С другой стороны, для использования в списках и реализациях можно сделать выбор в пользу абстрактного типа данных первого класса, который в точности определяет все соглашения, касающиеся фиктивного узла, и т.д. Выбор в пользу низкого уровня, которому отдается предпочтение в настоящий момент, позволяет сосредоточиться на манипуляциях со связями, которые характеризуют сами алгоритмы и структуры данных, являющиеся предметом изучения настоящей книги.

Существует фундаментальное правило, регламентирующее работу со структурами связных списков, которое критично для многих приложений, но не всегда четко просматривается в наших программных кодах. В более сложной среде может иметь место случай, когда указатели на узлы списка, с которыми мы работаем, определяются другими частями прикладной системы (т.е., они содержатся в мультисписках). Возможность того, что ссылки на узлы будут осуществляться через указатели, управление которыми реализуется за пределами сортировки, означает, что наши программы *должны менять только связи в узлах и не должны менять ключей или какой-либо другой информации*. Например, если требуется выполнить операцию обмена, то на первый взгляд кажется, что проще всего совершить обмен значениями элементов (что мы, собственно говоря, и делали во время сортировки массивов). Но в таком случае любая ссылка на любой из этих узлов с использованием какой-либо другой связи обнаружит, что значение изменилось, и результат ссылки не даст ожидаемого эффекта. Необходимо, чтобы сами связи изменились таким образом, чтобы узлы появились в порядке, заданном сортировкой, когда список просматривается через связи, к кото-

рым имеем доступ мы, но при этом сохранялся прежний порядок, если доступ производится по любым другим связям. При этом существенно усложняется реализации, однако обычно это является необходимым условием.

Не особенно сложно приспособить сортировку вставками, выбором и пузырьковую сортировку для работы со связными списками, но в каждом конкретном случае возникают занятные проблемы. Сортировка выбором достаточно проста: имеется входной список (в котором в исходном положении хранятся данные) и выходной список (в котором фиксируются результаты сортировки); входной список просматривается с целью обнаружения максимального элемента, который затем удаляется из списка и помещается в начало выходного списка (см. рис. 6.16). Реализация этой операции представляет собой одну из простейших манипуляций над связными списками и является полезным методом сортировки коротких списков. Реализация показана в программе 6.16. Другие методы сортировки оставляются в качестве упражнений для самостоятельной проработки.

### Программа 6.16. Сортировка выбором связного списка

Сортировка выбором связного списка достаточно проста, но несколько отличается от сортировки массива тем же методом, поскольку размещение элемента в начале списка — более простая операция. Поддерживаются входной список (указатель **h->next**) и выходной список (указатель **out**). Когда входной список не пуст, он просматривается с целью нахождения максимального элемента, который затем удаляется из входного и помещается в начало выходного списка. Реализация использует вспомогательную программу **findmax**, которая возвращает связь узла, связь которого указывает на максимальный элемент в списке (см. упражнение 3.34).

```
link listselection(link h)
{ node dummy(0); link head = &dummy, out = 0;
  head->next = h;
  while (head->next != 0)
    { link max = findmax(head), t = max->next;
      max->next = t->next;
      t->next = out; out = t;
    }
  return out;
}
```

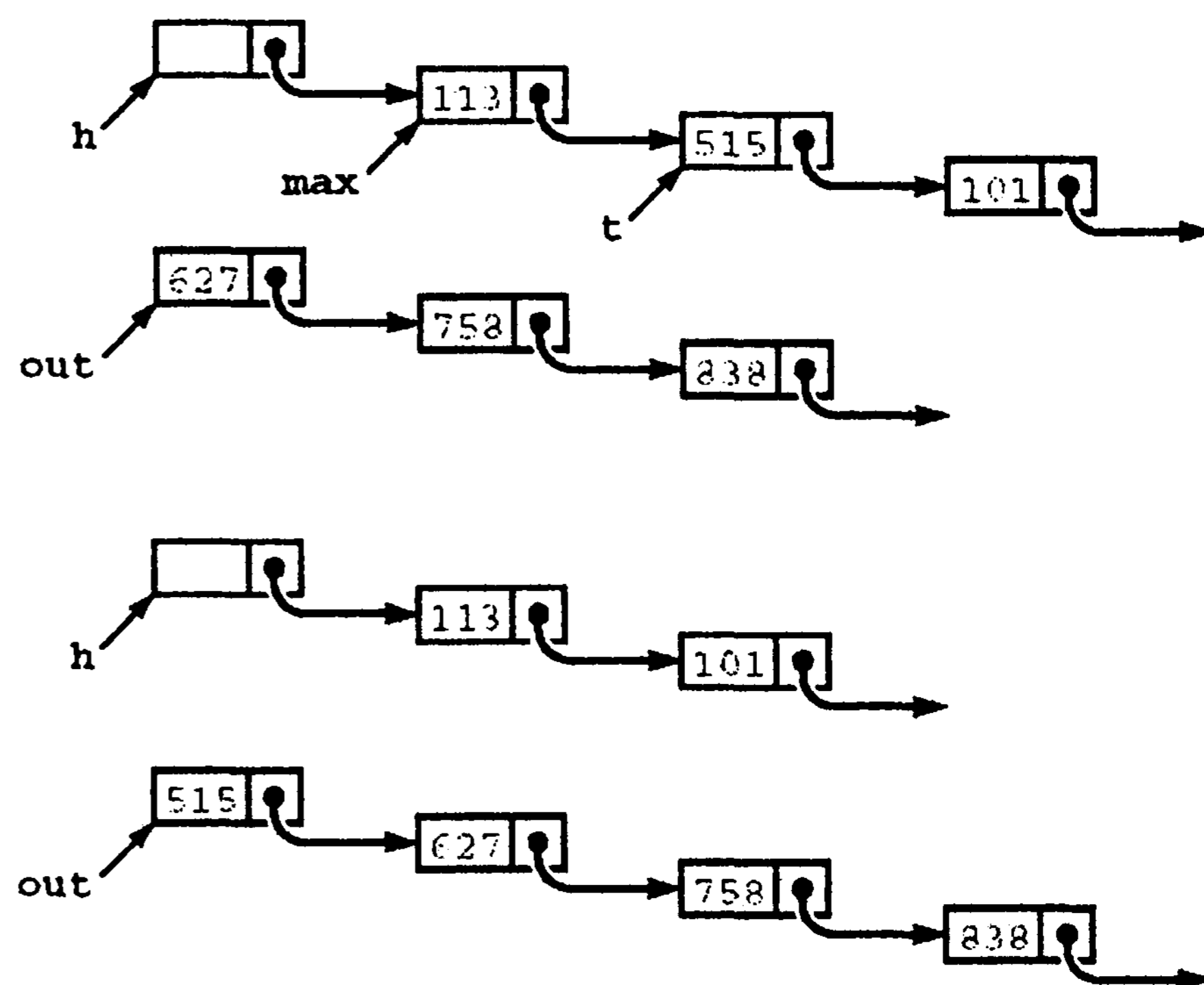


РИСУНОК 6.16. СОРТИРОВКА ВЫБОРОМ СВЯЗНОГО СПИСКА

На этой диаграмме показан один шаг сортировки выбором связного списка. Мы поддерживаем входной список с указателем **h->next** и выходной список с указателем **out** (вверху). Входной список просматривается с таким расчетом, чтобы **max** показывал на узел, предшествующий (а **t** указывал на) узлу, содержащему максимальный элемент.

Таковыми являются указатели, которые необходимы для того, чтобы исключить **t** из входного списка (уменьшив его длину на 1) и поместить его в начало выходного списка (увеличивая его длину на 1), сохраняя в выходном списке заданный порядок (внизу). С течением процесса, в конечном итоге, исчерпывается весь входной список, а в выходном списке элементы размещаются в заданном порядке.

В некоторых ситуациях, возникающих во время обработки списков, вообще нет необходимости в явной реализации сортировки. Например, решено всегда содержать список в определенном порядке и включать новые узлы в список аналогично тому, как это делается при сортировке вставками. Такой подход требует незначительных дополнительных затрат, если вставки производятся сравнительно редко, либо если список имеет небольшие размеры, а также в ряде других случаев. Например, по той или иной причине понадобится выполнить просмотр всего списка перед тем, как вставить в него новые узлы (возможно, чтобы убедиться в том, что в списке нет дубликатов). В главе 14 рассматривается алгоритм, который использует упорядоченные связные списки, а в главах 12 и 14 исследуются многочисленные структуры данных, эффективность которых повышается благодаря наличию порядка.

## Упражнения

- ▷ **6.64.** Показать содержимое входного и выходного списков при условии, что сортировка ключей `ASORTINGEXAMPLE` выполняется с использованием программы 6.15.
- 6.65.** Разработать реализацию интерфейса связного списка, заданного в программе 6.15.
- 6.66.** Написать клиентскую программу, представляющую собой драйвер, измеряющий эффективность сортировки связных списков (см. упражнение 6.9).
- **6.67.** Разработать АТД первого класса для связных списков (см. раздел 4.8), который включает конструктор для инициализации случайными значениями, конструктор для инициализации через перегруженную операцию `operator<<`, вывод данных через перегруженную операцию `operator>>`, деструктор, конструктор копий и функцию-член `sort`. Воспользоваться сортировкой выбором для реализации функции `sort`, при этом `findmax` рассматривается как приватная функция-член.
- 6.68.** Разработать программную реализацию пузырьковой сортировки для связных списков. *Предостережение:* Обмен местами двух соседних элементов в связном списке — это более сложная операция, чем может показаться на первый взгляд.
- ▷ **6.69.** Включить в программу 3.11 модуль сортировки вставками таким образом, чтобы он обладал такими же функциональными возможностями, что и программа 6.16.
- 6.70.** Вариант сортировки вставками, использованный в программе 3.11, выполняет сортировку связных списков значительно медленнее, чем сортировку массивов на некоторых входных файлах. Дайте описание одного из таких файлов и объясните, в чем заключается проблема.
- **6.71.** Построить программную реализацию варианта сортировки методом Шелла, ориентированного на связные списки, который не требует существенно большего объема памяти и времени для сортировки случайно упорядоченного файла, нежели вариант, предназначенный для сортировки массивов. *Совет:* воспользоваться пузырьковой сортировкой.
- **6.72.** Реализовать АТД для *последовательностей*, которые позволили бы использовать одну и ту же клиентскую программу для отладки программных реализаций сортировки как связных списков, так и массивов. Иначе говоря, клиентские про-

граммы могут генерировать последовательности из  $N$  элементов (в результате генерации случайно распределенных совокупностей элементов либо стандартного ввода), выполнять сортировку последовательности и отображать ее содержимое. Например, АТД в файле SEQ.cxx должен работать со следующим программным кодом:

```
#include "Item.h"
#include "SEQ.cxx"
main(int argc, char *argv[])
{ int N = atoi(argv[1], sw = atoi(argv[2]);
  if (sw) SEQrand(N); else SEQscan();
  SEQsort();
  SEQshow();
}
```

Получить две реализации: одну, использующую представление в виде массива, и другую, использующую представление в виде связанного списка. Воспользоваться сортировкой выбором.

- 6.73. Расширить реализацию из упражнения 6.72 так, чтобы она стала АТД первого класса. *Совет:* решение ищите в библиотеке стандартных шаблонов.

## 6.10. Метод распределяющего подсчета

Повышение эффективности некоторых алгоритмов сортировки достигается за счет использования специфических свойств ключей. Например, рассмотрим такую задачу: требуется выполнить сортировку файла из  $N$  элементов, ключи которых принимают различные значения в диапазоне от 0 до  $N - 1$ . Мы можем решить эту проблему сразу, используя с этой целью массив **b**, посредством оператора:

```
for (i = 0; i < N; i++) b[key(a[i])] = a[i];
```

То есть, мы выполняем сортировку, используя ключи как *индексы*, а не как абстрактные элементы, которые сравниваются между собой. В этом разделе мы ознакомимся с элементарным методом, который использует ключевую индексацию для повышения эффективности сортировки в случае, когда ключами служат целые числа, принимающие значения в ограниченном диапазоне.

Если все ключи равны 0, то сортировка тривиальна; теперь предположим, что два различных ключа принимают значения 0 и 1. Такого рода проблема сортировки может возникнуть, когда требуется выделить элементы файла, удовлетворяющие некоторым (возможно, достаточно сложным) проверочным критериям: допустим, мы считаем, что значение 0 ключа означает, что элемент "принят", а значение 1 — что элемент "отвергнут". Один из способов сортировки состоит в том, что сначала подсчитывается число 0, затем выполняется второй подход по входному массиву **a** с целью размещения его элементов в массиве **b**, при этом предусматривается массив, состоящий из двух счетчиков, который используется следующим образом. Мы начинаем с того, что помещаем 0 в **cnt[0]**, а количество нулевых ключей в файле — в **cnt[1]**, с целью показать, что в рассматриваемом файле не существуют ключи, принимающие значения меньше 0, но имеется **cnt[1]** ключей, значения которых меньше 1. Разумеется, мы можем заполнить массив **b** следующим образом: в начало массива записы-

ваются 0 (начиная с  $b[cnt[0]]$  или  $b[0]$ ) и 1, начиная с  $b[cnt[1]]$ . Таким образом, программный код

```
for (i = 0; i < N; i++)
    b[cnt[a[i]]++] = a[i];
```

переносит элементы из  $a$  в  $b$ . Опять-таки мы получаем быструю сортировку за счет использования ключей в качестве индексов (для выбора между  $cnt[0]$  и  $cnt[1]$ ).

Такой подход очень просто обобщается. Более реалистичную задачу в том же духе можно сформулировать следующим образом: выполнить сортировку файла, состоящего из  $N$  элементов, ключи которого принимают целые значения в диапазоне от 0 до  $M - 1$ . Расширим базовый метод, описанный в предыдущем параграфе, до алгоритма, получившего название *распределяющего подсчета*, который эффективно решает эту задачу для не слишком больших  $M$ . Точно так же как и в случае двух ключей, идея состоит в том, чтобы подсчитать количество ключей с каждым конкретным значением, а затем использовать счетчики при перемещении в соответствующие позиции во время второго прохода по сортируемому файлу. Сначала подсчитывается число ключей для каждого значения, затем вычисляются частичные суммы, чтобы знать, сколько имеется ключей, меньших или равных каждому такому значению. Далее снова, аналогично случаю двух значений ключа, используем эти числа как индексы при распределении ключей. Для каждого ключа показания связанного с ним счетчика рассматриваются в качестве индекса, указывающего на конец блока ключей, принимающих одно и то же значение. Этот индекс используется при размещении ключей в массиве  $b$ , после чего производится переход к следующему элементу. Описанный процесс иллюстрируется на рис. 6.17. Реализация находится в программе 6.17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	3	3	0	1	1	0	3	0	2	0	1	1	2	0

0	1	2	3
0	6	4	2
0	6	10	12

0	0													
3	0													3
3	0													3 3
0	0	0												3 3
1	0	0					1							3 3
1	0	0					1	1						3 3
0	0	0	0				1	1						3 3
3	0	0	0				1	1						3 3 3
0	0	0	0	0			1	1						3 3 3
2	0	0	0	0			1	1			2			3 3 3
0	0	0	0	0	0		1	1			2			3 3 3
1	0	0	0	0	0		1	1	1		2			3 3 3
1	0	0	0	0	0		1	1	1	1	2			3 3 3
2	0	0	0	0	0		1	1	1	1	2	2		3 3 3
0	0	0	0	0	0	0	1	1	1	1	2	2	3	3 3
0	0	0	0	0	0	0	1	1	1	1	2	2	3	3 3

### РИСУНОК 6.17. СОРТИРОВКА МЕТОДОМ РАСПРЕДЕЛЯЮЩЕГО ПОДСЧЕТА

Сначала для каждого значения определяется, сколько имеется в файле ключей, принимающих это конкретное значение: в рассматриваемом примере имеется шесть ключей, принимающих значение 0, семь значений 1, два значения 2 и три значения 3. Затем подсчитываются частичные суммы ключей, принимающих значения, меньшие значений конкретных ключей: 0 ключей меньше 0, 6 ключей меньше 1, 10 ключей меньше 2 и 12 ключей меньше 3 (таблица в середине). Затем, помещая ключи в соответствующие позиции, частичные суммы рассматриваются в качестве индексов: 0 в начале файла помещается в ячейку 0; далее увеличивается на единицу значение указателя, соответствующего 0; в эту позицию пойдет следующий 0. Затем 3 из следующей позиции в файле слева помещается в ячейку 12 (поскольку в файле имеются 12 ключей со значением, меньшим 3), причем соответствующий счетчик увеличивается на 1 и т.д.



**Программа 6.17. Распределяющий подсчет**

Во время выполнения первого цикла `for` счетчики инициализируются значениями 0; во втором цикле `for` значение второго счетчика устанавливается равным количеству ключей 0, значение третьего счетчика — количеству ключей, равных 1 и т.д. В третьем цикле `for` все эти числа складываются, в результате чего получаем количество ключей, меньших или равных ключу, соответствующему очередному счетчику. Эти числа теперь представляют собой индексы концов тех частей файлов, которым эти ключи принадлежат. Четвертый цикл `for` перемещает ключи во вспомогательный массив `b` в соответствии со значениями этих индексов, а на завершающем цикле отсортированный файл возвращается в файл `a`. Чтобы этот программный код работал, необходимо, чтобы ключи были целыми значениями, не превышающими `M`, хотя его можно легко модифицировать таким образом, чтобы ключи можно было извлекать из элементов с более сложной структурой (см. упражнение 6.77).

```
void distcount(int a[], int l, int r)
{ int i, j, cnt[M];
  static int b[maxN];
  for (j = 0; j < M; j++) cnt[j] = 0;
  for (i = l; i <= r; i++) cnt[a[i]+1]++;
  for (j = 1; j < M; j++) cnt[j] += cnt[j-1];
  for (i = l; i <= r; i++) b[cnt[a[i]]++] = a[i];
  for (i = l; i <= r; i++) a[i] = b[i-1];
}
```

**Лемма 6.12.** *Метод распределяющего подсчета представляет собой сортировку с линейно зависимым временем выполнения при условии, что диапазон изменения значений ключей пропорционален размеру файла.*

Каждый элемент перемещается дважды, один раз в процессе распределения и один раз при возвращении в исходный файл; ссылки на ключи также производятся дважды, один раз при подсчете, другой раз при выполнении распределения. Два других цикла `for` в алгоритме используются при накоплении показаний счетчиков и фактически мало влияют на время выполнения сортировки до тех пор, пока количество подсчетов существенно не превосходит размер файла.

Если сортировке подвергается файл крупных размеров, то вспомогательный файл `b` может привести к проблемам в плане распределения памяти. Программу 6.17 можно изменить так, чтобы она совершала сортировку на месте (т.е., без необходимости построения вспомогательного файла), используя методы, подобные применяемым в программе 6.14. Эта операция тесно связана с базовыми методами, которые будут обсуждаться в главах 7 и 10, так что отложим ее изучение до упражнений 12.16 и 12.17 из раздела 12.3. Как станет ясно из главы 12, подобная экономия пространства памяти достигается ценой нарушения устойчивости алгоритма, из-за чего область применения этого алгоритма существенно сужается, поскольку приложения, использующие большое число дубликатов ключей, часто прибегают к помощи других связанных с ними ключей, относительный порядок которых должен быть сохранен. Исключительно важный пример такого рода исследуется в главе 10.

## Упражнения

- **6.74.** Дать пример специализированной версии метода распределяющего подсчета для сортировки файлов, элементы которых могут принимать только одно из трех значений (**a**, **b** или **c**).
- 6.75.** Предположим, что используется сортировка вставками для упорядочения случайно распределенного файла, элементы которого принимают одно из трех возможных значений. Какой зависимости подчиняется время выполнения сортировки: линейной, квадратичной или некоторой промежуточной зависимости?
- ▷ **6.76.** Показать, как файл **A B R A C A D A B R A** сортируется при помощи метода распределяющего подсчета.
- 6.77.** Реализовать сортировку методом распределяющего подсчета элементов, которые представляют собой потенциально большие записи с целочисленными ключами, принимающими значения в небольшом диапазоне.
- 6.78.** Реализовать сортировку методом распределяющего подсчета в виде сортировки по указателям.

## Быстрая сортировка

Темой настоящей главы является алгоритм сортировки, который, по-видимому, используется гораздо чаще любого другого, а именно — алгоритму *быстрой сортировки* (*quicksort*). Базовый алгоритм этого вида сортировки был открыт в 1960 г. Хоаром (С.А.Р.Ноаре), и с той поры многие специалисты посчитали своим долгом досконально изучить его (см. *раздел ссылок*). Быстрая сортировка стала популярной прежде всего потому, что ее нетрудно реализовать, она хорошо работает на различных видах входных данных и во многих случаях требует меньше затрат ресурсов по сравнению с другими методами сортировки.

Алгоритм быстрой сортировки обладает и другими весьма привлекательными особенностями: он принадлежит к категории обменных (*in-place*) сортировок (т.е., требует всего лишь небольшого вспомогательного стека), на выполнение сортировки  $N$  элементов в среднем затрачивается время, пропорциональное  $N \log N$  и для него характерны исключительно короткие внутренние циклы. Его недостатком является то, что он неустойчив, для его выполнения в наихудшем случае требуется  $N^2$  операций, он хрупок в том смысле, что даже простая ошибка в реализации может пройти незамеченной и вызвать ошибки в работе алгоритма на некоторых видах файлов.

Работа быстрой сортировки проста для понимания. Алгоритм был подвергнут тщательному математическому анализу, и можно дать достаточно точную оценку его эффективности. Этот анализ был подтвержден многосторонними эмпирическими экспериментами, а сам алгоритм был усовершенствован до такой степени, что ему отдают предпочтение в широчайшем диапазоне практических

применений сортировки. По этой причине потребуется уделить гораздо большее внимание эффективной реализации алгоритма быстрой сортировки, нежели реализациям других алгоритмов. Аналогичные методы реализации целесообразно применять и к другим алгоритмам; поместив в них быструю сортировку, ими можно пользоваться с большей уверенностью, поскольку появится возможность в точности предсказать, какое влияние они окажут на эффективность сортировки.

Заманчиво попытаться разработать способы улучшения быстрой сортировки: чем выше быстродействие алгоритма сортировки, тем привлекательнее выглядят возможности вычислительных систем, а быстрая сортировка представляет собой почтенный метод, который лишь усиливает это впечатление. Практически с момента опубликования Хоаром алгоритма быстрой сортировки в литературе стали регулярно появляться его усовершенствованные версии. Предлагалось и анализировалось множество идей, но при этом совсем не трудно ошибиться при оценке этих улучшений, поскольку данный алгоритм настолько хорошо сбалансирован, что эффект от усовершенствования одной части программы может послужить причиной ухудшения функционирования другой ее части. Мы детально изучим три модификации, которые существенно повышают эффективность быстрой сортировки.

Тщательно сбалансированная версия быстрой сортировки, по всей вероятности, будет выполняться быстрее любого другого метода сортировки на большинстве компьютеров, к тому же быстрая сортировка широко используется как библиотечная программа сортировки и для других серьезных приложений сортировки. В самом деле, сортировка из стандартной библиотеки C++ называется `qsort` (б[ыстрая] сортировка), ибо обычно именно алгоритм быстрой сортировки лежит в основе различных реализаций. Однако, время выполнения быстрой сортировки зависит от организации входных данных и колеблется между линейной и квадратичной зависимостью от количества сортируемых элементов и пользователи иногда бывают неприятно удивлены неожиданно неудовлетворительными, неприемлемыми результатами сортировки некоторых видов входных данных, особенно когда используются хорошо отлаженные версии этого алгоритма. Если приложение работает настолько плохо, что возникает подозрение в наличии дефектов в реализации быстрой сортировки, то сортировка методом Шелла может оказаться более удачным выбором, обеспечивающим лучший результат при меньших затратах на реализацию. Следует отметить, что в случае особо крупных файлов быстрая сортировка выполняется примерно в пять-десять раз быстрее сортировки методом Шелла, при этом на некоторых видах файлов, довольно часто встречающихся на практике, может быть достигнута еще большая эффективность данного вида сортировки.

## 7.1. Базовый алгоритм

Быстрый метод сортировки функционирует по принципу "разделяй и властвуй". Он делит сортируемый массив на две части, затем сортирует эти части независимо друг от друга. Как будет показано далее, точное положение точки деления зависит от исходного порядка элементов во входном файле. Суть метода заключается в процессе разбиения файла, который переупорядочивает файл таким образом, что выполняются следующие условия:

- Элемент  $a[i]$  для некоторого  $i$  занимает свою окончательную позицию в массиве.
- Ни один из элементов  $a[i], \dots, a[i-1]$  не превышает  $a[i]$ .
- Ни один из элементов  $a[i+1], \dots, a[r]$  не является меньшим  $a[i]$ .

Полная сортировка достигается путем деления файла на подфайлы с последующим применением к ним этих же методов (см. рис. 7.1). Поскольку процесс разбиения всегда помещает, по меньшей мере, один из элементов в окончательную позицию, по индукции нетрудно получить формальное доказательство того, что этот рекурсивный метод обеспечивает правильную сортировку. Программа 7.1 содержит рекурсивную реализацию упомянутой идеи.

### Программа 7.1. Быстрая сортировка

Если в массиве имеется один или меньшее число элементов, то ничего делать не надо. В противном случае массив подвергается обработке со стороны процедуры **partition** (см. программу 7.2), которая помещает элемент  $a[i]$  для некоторого  $i$  в позицию между  $l$  и  $r$  включительно и переупорядочивает остальные элементы таким образом, что рекурсивные вызовы этой процедуры должным образом завершают сортировку.

```
template <class Item>
void quicksort(Item a[], int l, int r)
{
    if (r <= l) return;
    int i = partition(a, l, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
}
```

Разбиение осуществляется с использованием следующей стратегии. Прежде всего, в качестве *разделяющего элемента* (*partitioning element*) произвольно выбирается элемент  $a[r]$  — он сразу займет свою окончательную позицию. Далее начинается просмотр с левого конца массива, который продолжается до тех пор, пока не будет найден элемент, превосходящий по значению разделяющий элемент, затем выполняется просмотр, начиная с правого конца массива, который продолжается до тех пор, пока не отыскивается элемент, который по значению меньше разделяющего. Оба элемента, на которых просмотр был прерван, очевидно, находятся не на своих местах в разделенном массиве, и потому они меняются местами. Мы продолжаем дальше в том же духе, пока не убедимся в том, что слева от левого указателя не осталось ни одного элемента, который был бы больше по значению разделяющего, и ни одного элемента справа от правого указателя, которые были бы меньше по значению разделяющего элемента, как показано на следующей диаграмме

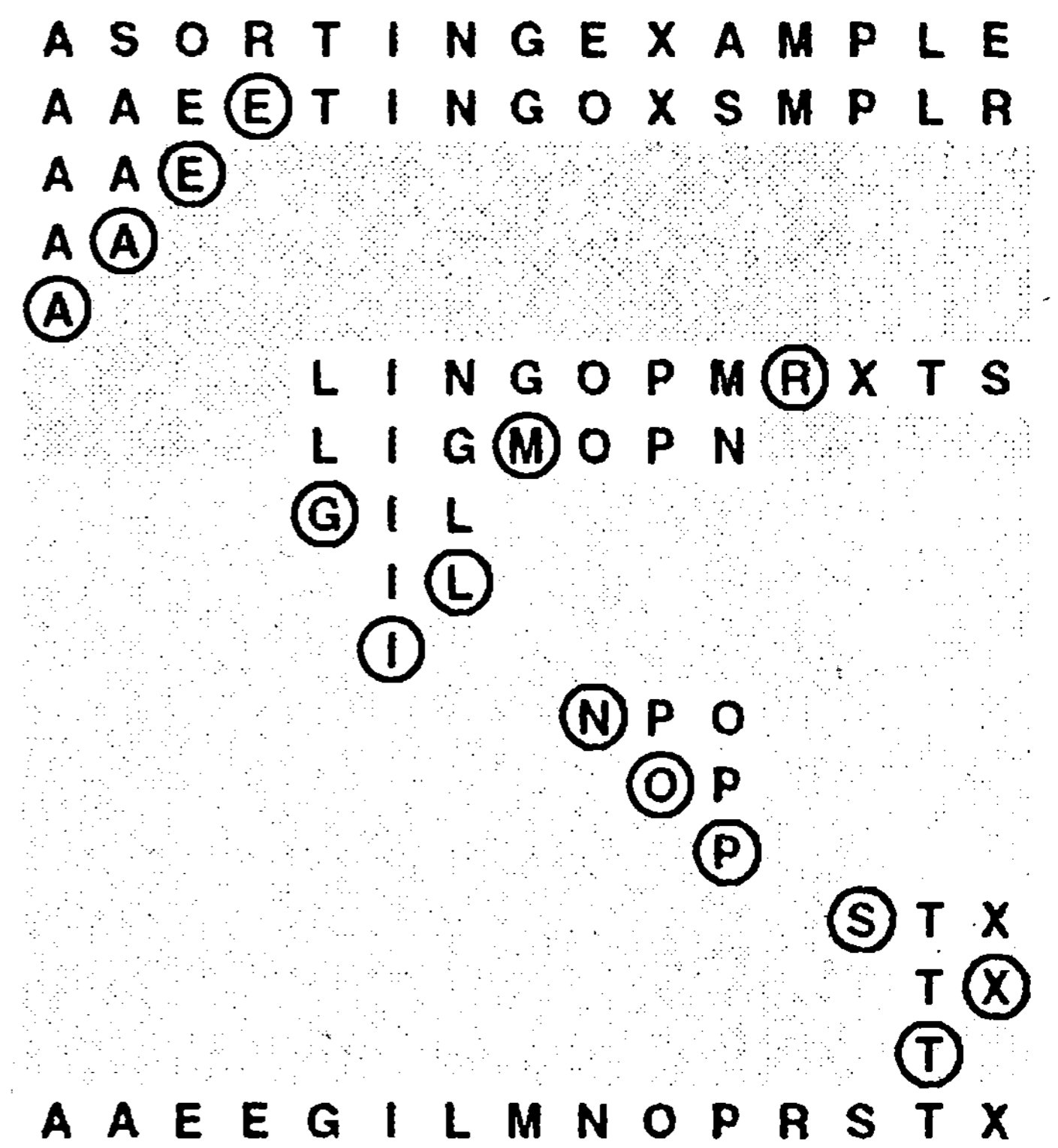


РИСУНОК 7.1. ПРИМЕР БЫСТРОЙ СОРТИРОВКИ

*Быстрая сортировка представляет собой рекурсивный процесс разбиения файла на части: мы разбиваем его, помещая некоторый (разделяющий) элемент в свою окончательную позицию и выполняем перегруппировку массива таким образом, что элементы, меньшие по значению, остаются слева от разделяющего элемента, а элементы, большие по значению, — справа. Далее мы рекурсивно сортируем левую и правую части массива. Каждая строка этой диаграммы представляет результат разбиения отображаемого подфайла с помощью элемента, заключенного в кружок. Конечным результатом такого вида сортировки является полностью отсортированный файл.*



щему элементу — возможно, в таких реализациях и потребуется еще одна проверка во избежание ситуации, когда указатель смещается с правого конца массива. С другой стороны, усовершенствование быстрой сортировки, которое будет обсуждаться в разделе 7.5, имеет своим положительным побочным эффектом то обстоятельство, что отпадает необходимость как в проверке, так и в наличии самого служебного значения на каждом конце.

Процесс разделения неустойчив, поскольку во время любой операции обмена любой ключ может пройти мимо большого числа равных ему ключей (которые остаются непроверенными). Простые способы сделать быструю сортировку, ориентированную на массивы, устойчивой, пока не известны.

Реализацию разделяющей процедуры следует выполнять с особой осторожностью. В частности, наиболее простой способ гарантировать завершение рекурсивной программы заключается в том, что (i) она не вызывает себя для файлов с размерами  $l$  и менее и (ii) вызывает себя только для файлов, размер которых строго меньше размеров входного файла. Эти стратегии на первый взгляд кажутся очевидными, однако при этом легко упустить из виду такие свойства ввода, которые в конечном счете могут послужить причиной неудачи. Например, обычная ошибка в реализации быстрой сортировки заключается в отсутствии гарантии того, что каждый элемент всегда будет поставлен в нужное место, а также в возможности вхождения программы сортировки в бесконечный цикл в случаях, когда разделяющим элементом служит наибольший или наименьший элемент файла.

Когда в файле встречаются дубликаты ключей, фиксация момента пересечения указателей сопряжена с определенными трудностями. Процесс разбиения можно слегка усовершенствовать, если остановить просмотр при  $i < j$ , а затем воспользоваться значением  $j$ , а не  $i-1$ , чтобы определить правую границу левого подфайла при первом рекурсивном вызове. В таком случае выполнение еще одной итерации цикла следует рассматривать как усовершенствование, поскольку оба цикла просмотра прекращаются, когда  $j$  и  $i$  ссылаются на один тот же элемент, в результате два элемента занимают свои окончательные позиции: один из них — это элемент, который остановил оба просмотра и в силу этого обстоятельства должен быть равен разделяющему элементу, а также и сам разделяющий элемент. Подобная ситуация могла бы возникнуть, например, если бы на рис. 7.2  $R$  был  $E$ . Это изменение, по-видимому, заслуживает того, чтобы его внести в программу, ибо в данном конкретном случае программа в том виде, в каком она здесь представлена, оставляет запись с ключом, равным ключу разделяющего элемента, в  $a[r]$ , и это приводит к тому, что первое разделение, выполняемое за счет вызова `quicksort(a, i+1, r)`, вырождается, поскольку самый правый ключ оказывается наименьшим. Однако, реализацию разделения, используемую в программе 7.2, несколько проще понять, так что в дальнейшем мы будем ссылаться на нее как на базовый метод разделения, применяемый в быстрой сортировке. Если в сортируемом файле присутствует значительное число дублированных ключей, то на передний план выступают другие факторы. Они будут рассматриваться несколько позже.

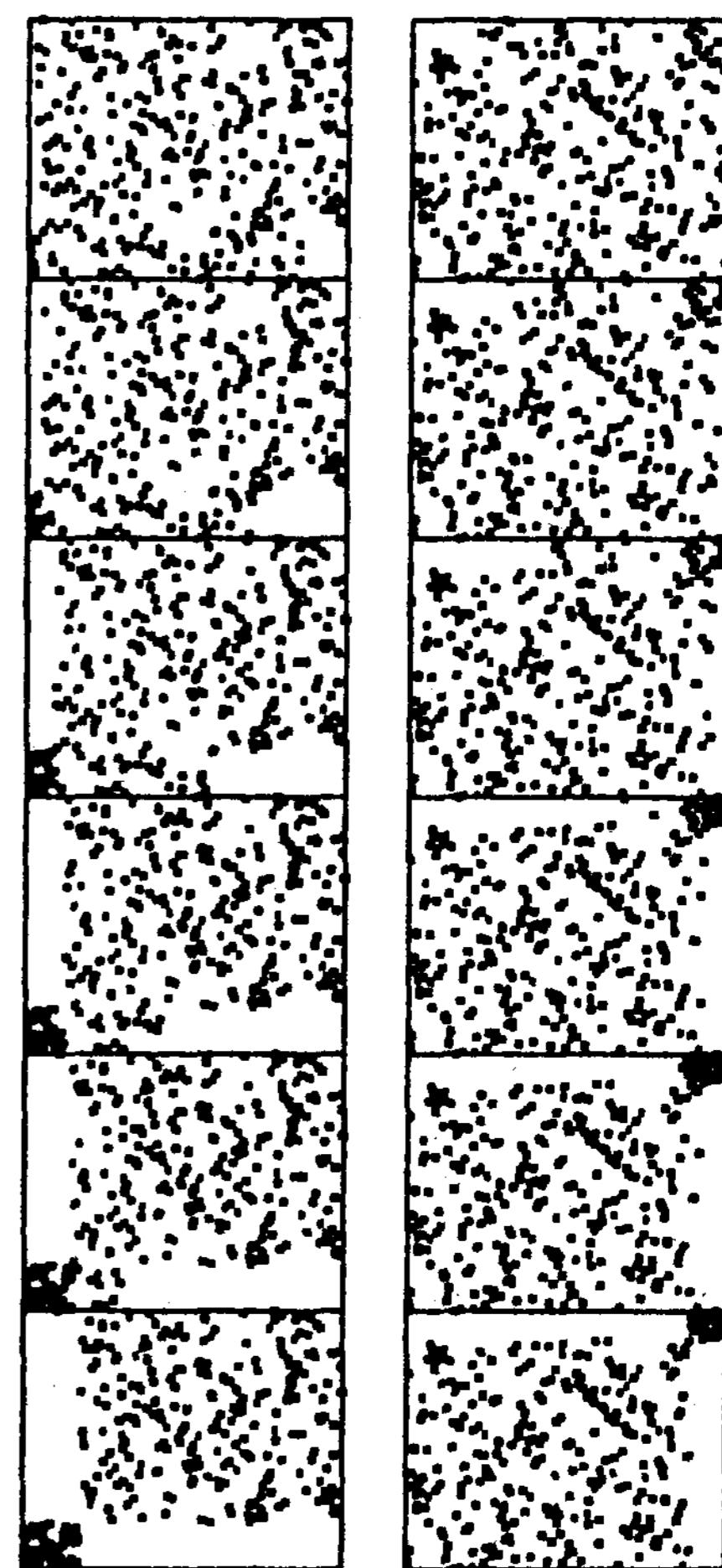
## Программа 7.2. Разделение

Переменная  $v$  сохраняет значение разделяющего элемента  $a[r]$ , а  $i$  и  $j$  представляет собой, соответственно, указатели левого и правого просмотра. Цикл разделения увеличивает значение  $i$  и уменьшает значение  $j$  на 1, причем условие, что ни один элемент слева от  $i$  не больше  $v$  и ни один элемент справа от  $j$  не больше  $v$ , не нарушается. Как только значения указателей пересекаются, процедура разбиения завершается, меняя местами  $a[r]$  и  $a[i]$ , при этом  $v$  присваивается значение  $a[i]$ , так что не будет ни одного большего элемента справа от  $v$  и ни одного меньшего элемента слева от  $v$ .

Разделяющий цикл реализуется в виде бесконечного цикла, который прерывается функцией **break**, когда указатели пересекаются. Проверка  $j==1$  обеспечивает защиту от случая, когда в качестве разделяющего элемента используется наименьший элемент файла.

```
template <class Item>
int partition(Item a[], int l, int r)
{ int i = l-1, j = r; Item v = a[r];
  for (;;)
  {
    while (a[++i] < v) ;
    while (v < a[--j]) if (j == 1)
      break;
    if (i >= j) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[r]);
  return i;
}
```

Существуют три основных стратегии, которые можно выбрать применительно к ключам, равным разделяющему элементу: заставить оба указателя останавливаться на таком ключе (как это имеет место в программе 7.2); заставить один указатель остановиться, а другому позволить продолжать просмотр; позволить обоим указателям продолжить просмотр. Вопрос о том, какая из этих стратегий лучше, был тщательно изучен с привлечением математического аппарата, и результаты показали, что наилучшей стратегией является останов обоих указателей главным образом потому, что при этом получается сбалансированное разделение при наличии множества дублированных ключей, в то время как две других стратегии для некоторых видов файлов приводят к ярко выраженному нарушению баланса. В разделе 7.6 рассматривается несколько более сложный и в то же время гораздо более эффективный способ работы с дублированными ключами.



**РИСУНОК 7.3.**  
ДИНАМИЧЕСКИЕ  
ХАРАКТЕРИСТИКИ  
ПРОЦЕССА РАЗДЕЛЕНИЯ  
БЫСТРОЙ СОРТИРОВКИ

*Процесс разбиения делит файл на два подфайла, которые могут подвергаться сортировке независимо друг от друга. Ни один из элементов слева от значения указателя просмотра левого подфайла не может быть больше его, так что выше и левее его на диаграмме точек нет; ни один из элементов справа от значения указателя просмотра правого подфайла не может быть меньше его, так что ниже и правее его на диаграмме точек нет. Из этих двух примеров легко видеть, что разбиение файла с произвольной организацией делит его на два файла меньших размеров с произвольной организацией, при этом один элемент (а именно, разделяющий) занимает свою окончательную позицию на диагонали.*



В конечном итоге эффективность сортировки зависит от качества разбиения файла, которое, в свою очередь, зависит от выбора значения разделяющего элемента. Рисунок 7.2 демонстрирует, что процедура разделения разбивает крупный файл с произвольной организацией на два файла с произвольной организацией меньших размеров, но при этом точка раздела может оказаться в любом месте файла. Мы предпочитаем выбирать такую точку раздела вблизи от середины файла, однако не располагаем необходимой для этого информацией. Если сортируется файл с произвольной организацией, то выбор элемента  $a[r]$  в качестве разделяющего — это то же самое, что и выбор любого другого конкретного элемента; он дает нам *в общем случае* точку раздела в непосредственной близости от середины. В разделе 7.4 проводится анализ рассматриваемого алгоритма, который позволит сравнить такой случай с идеальным выбором. В разделе 7.5 будет показано, насколько подобного рода анализ может оказаться полезным при выборе разделяющего элемента в целях повышения эффективности рассматриваемого алгоритма.

## Упражнения

- ▷ 7.1. Показать в стиле рассмотренного здесь примера, как быстрая сортировка сортирует файл `EASYQUESTION`.
- 7.2. Показать, как производится разделение файла `1001110000010100`, используя для этой цели программу 7.2 и несущественные модификации, предлагаемые по тексту.
- 7.3. Реализовать разделение, не прибегая к помощи операторов `break` или `goto`.
- 7.4. Разработать устойчивую быструю сортировку для связанных списков.
- 7.5. Каким является максимальное число перемещений наибольшего элемента файла, состоящего из  $N$  элементов, во время выполнения быстрой сортировки.

## 7.2. Характеристики производительности быстрой сортировки

Несмотря на все ее ценные качества, базовая программа быстрой сортировки обладает определенным недостатком, который заключается в том, что она исключительно неэффективна на некоторых простых файлах, которые могут встретиться на практике. Например, если она применяется для сортировки файла размером  $N$ , который уже отсортирован, то все имеющиеся разделения вырождаются, и программа вызовет сама себя  $N$  раз, перемещая за каждый вызов всего лишь один элемент.

**Лемма 7.1.** *Быстрая сортировка в наихудшем случае выполняет примерно  $N^2/2$  операций сравнения.*

В силу только что приведенного аргумента, число операций сравнения, выполненных при сортировке уже отсортированного файла, выражается как

$$N + (N - 1) + (N - 2) + \dots + 2 + 1 = (N + 1) N / 2.$$

Все разделения вырождаются как в случае файла, отсортированного в обратном порядке, так и в случае файлов определенных видов, вероятность столкнуться с которыми на практике существенно ниже (см. упражнение 7.6).

Подобное поведение означает не только то, что время выполнения быстрой сортировки приближенно определяется зависимостью  $N^2/2$ , но и то, что пространство памяти, необходимое для выполнения этой рекурсии, примерно пропорционально  $N$  (см. раздел 7.3), что в случае крупных файлов недопустимо. К счастью, имеются сравнительно простые способы существенного снижения вероятности того, что наихудший случай возникнет в типовых применениях рассматриваемой программы.

Наиболее благоприятный для быстрой сортировки случай имеет место, когда на каждой стадии разбиения файл делится на две равные части. Это обстоятельство приводит к тому, что количество операций сравнения, выполняемых в процессе быстрой сортировки, удовлетворяет рекуррентному соотношению типа "разделяй и властвуй".

$$C_N = 2C_{N/2} + N$$

Член  $2C_{N/2}$  соответствует затратам на сортировку двух подфайлов;  $N$  суть затраты на проверку каждого элемента, для чего используется тот или иной разделяющий указатель. Из главы 5 уже известно, что это рекуррентное соотношение имеет решение

$$C_N \approx N \lg N$$

Несмотря на то что не всегда все так удачно складывается, тем не менее, верно, что *в среднем* разделение попадает на середину файла. Если еще учесть точную вероятность каждой позиции разделения, то указанное выше рекуррентное соотношение становится более сложным и более трудным для решения, однако окончательный результат примерно такой же.

**Лемма 7.2.** *Быстрая сортировка в среднем выполняет  $2N \ln N$  операций сравнения.*

Точное рекуррентное соотношение для определения числа сравнений, выполняемых во время быстрой сортировки  $N$  случайно распределенных различных элементов, имеет вид

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}) \quad \text{для } N \geq 2,$$

при  $C_1 = C_0 = 0$ . Член  $N + 1$  учитывает затраты на выполнение операций сравнения разделяющего элемента с каждым из остальных элементов (два дополнительных в точке пересечения указателей); наличие остальных компонентов обусловлено тем фактом, что каждый элемент  $k$  может стать разделяющим элементом с вероятностью  $1/k$ , после чего остаются файлы с произвольной организацией, имеющие размеры  $k - 1$  и  $N - k$ .

Это рекуррентное соотношение, несмотря на внешнюю сложность, фактически довольно просто решается, буквально за три действия. Во-первых,  $C_0 + C_1 + \dots + C_{N-1}$  есть ни что иное как  $C_{N-1} + C_{N-2} + \dots + C_0$ ; следовательно, имеем

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \leq k \leq N} C_{k-1}.$$

Во-вторых, можно избавиться от суммы, если умножить обе части равенства на  $N$  и вычесть такую же формулу для  $N - 1$

$$NC_N - (N - 1)C_{N-1} = N(N + 1) - (N - 1)N + 2C_{N-1}$$

За счет такого упрощения рассматриваемое рекуррентное соотношение приобретает вид

$$NC_N = (N + 1)C_{N-1} + 2N.$$

В третьих, разделив обе части на  $N(N + 1)$ , получим рекуррентное соотношение, которое приобретает более компактную форму:

$$\begin{aligned} \frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \vdots \\ &= \frac{C_2}{3} + \sum_{3 \leq k \leq N} \frac{2}{k+1}. \end{aligned}$$

Это точное выражение почти равно сумме, которая легко аппроксимируется интегралом (см. раздел 2.3):

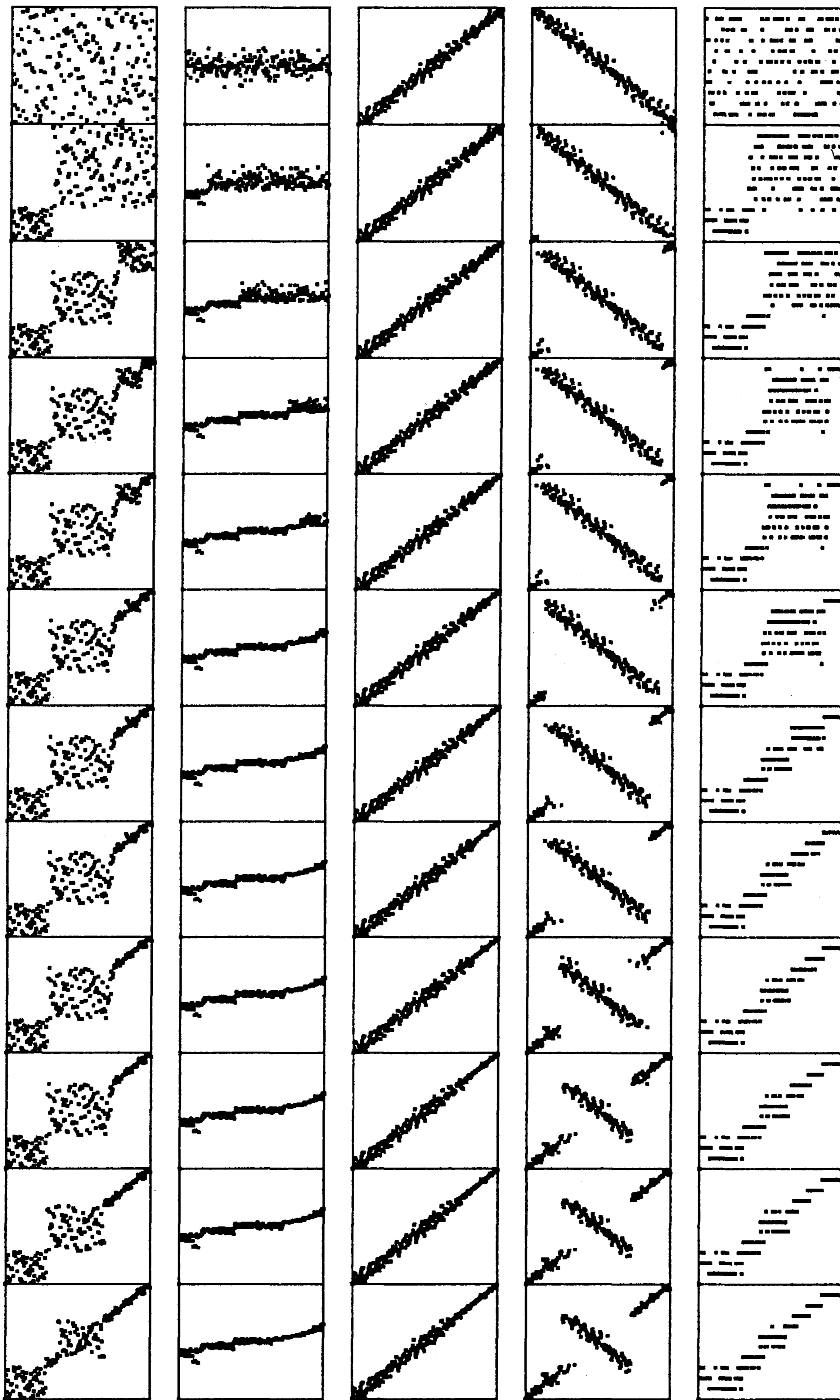
$$\frac{C_N}{N+1} \approx 2 \sum_{1 \leq k < N} \frac{1}{k} \approx 2 \int_1^N \frac{1}{x} dx = 2 \ln N,$$

откуда вытекает объявленный ранее результат. Обратите внимание на то, что  $2N \ln N \approx 1.39N \lg N$ , так что среднее количество операций сравнения приблизительно лишь на 39 процентов больше, чем в самом лучшем случае.

Данный анализ предполагает, что сортируемый файл содержит случайно упорядоченные записи с различными ключами, однако реализации в программах 7.1 и 7.2 могут работать медленно в случаях, когда ключи не обязательно различны и не обязательно расположены в случайном порядке (рис. 7.4). Если сортировка применяется многократно или если она должна использоваться для упорядочения очень большого файла (или, в частности, если она должна использоваться как универсальная библиотечная функция для сортировки файлов с неизвестными характеристиками), следует рассмотреть несколько усовершенствований, предлагаемых в разделах 7.5 и 7.6, которые снижают вероятность того, что наихудший случай возникнет на практике, а также уменьшают среднее время выполнения сортировки где-то на 20 процентов.

## Упражнения

- 7.6. Построить шесть файлов из 10 элементов, при упорядочении которых метод быстрой сортировки (программа 7.1) выполняет то же число операций сравнения, что и в самом худшем случае (когда все элементы файла упорядочены).
- 7.7. Написать программу, вычисляющую точное значение  $C_N$ , и сравнить это точное значение с приближенным значением  $2N \ln N$  для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 7.8. Сколько примерно операций сравнения выполнит быстрая сортировка (программа 7.1) для упорядочения файла из  $N$  равных элементов?



**РИСУНОК 7.4. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ БЫСТРОЙ СОРТИРОВКИ НА ФАЙЛАХ РАЗЛИЧНЫХ ТИПОВ**

*Выбор произвольного разделяющего элемента в быстрой сортировке приводит к тому, что для различных файлов применяются различные сценарии разбиения. Приводимые здесь диаграммы иллюстрируют начальные части сценариев, ориентированных на файлы с произвольной организацией, на файлы, упорядоченные в соответствии с распределением Гаусса, на почти упорядоченные, на почти упорядоченные в обратном порядке и на файлы с произвольной организацией с 10 различными значениями ключей (слева направо), использующих относительно большое значение отсечения для небольших подфайлов. Элементы, не вовлеченные в разделение, занимают места вблизи от диагонали, после чего остаются массивы, которые легко упорядочиваются за счет последующего применения сортировки вставками. Почти упорядоченные файлы требуют выполнения чересчур большого количества разбиений.*

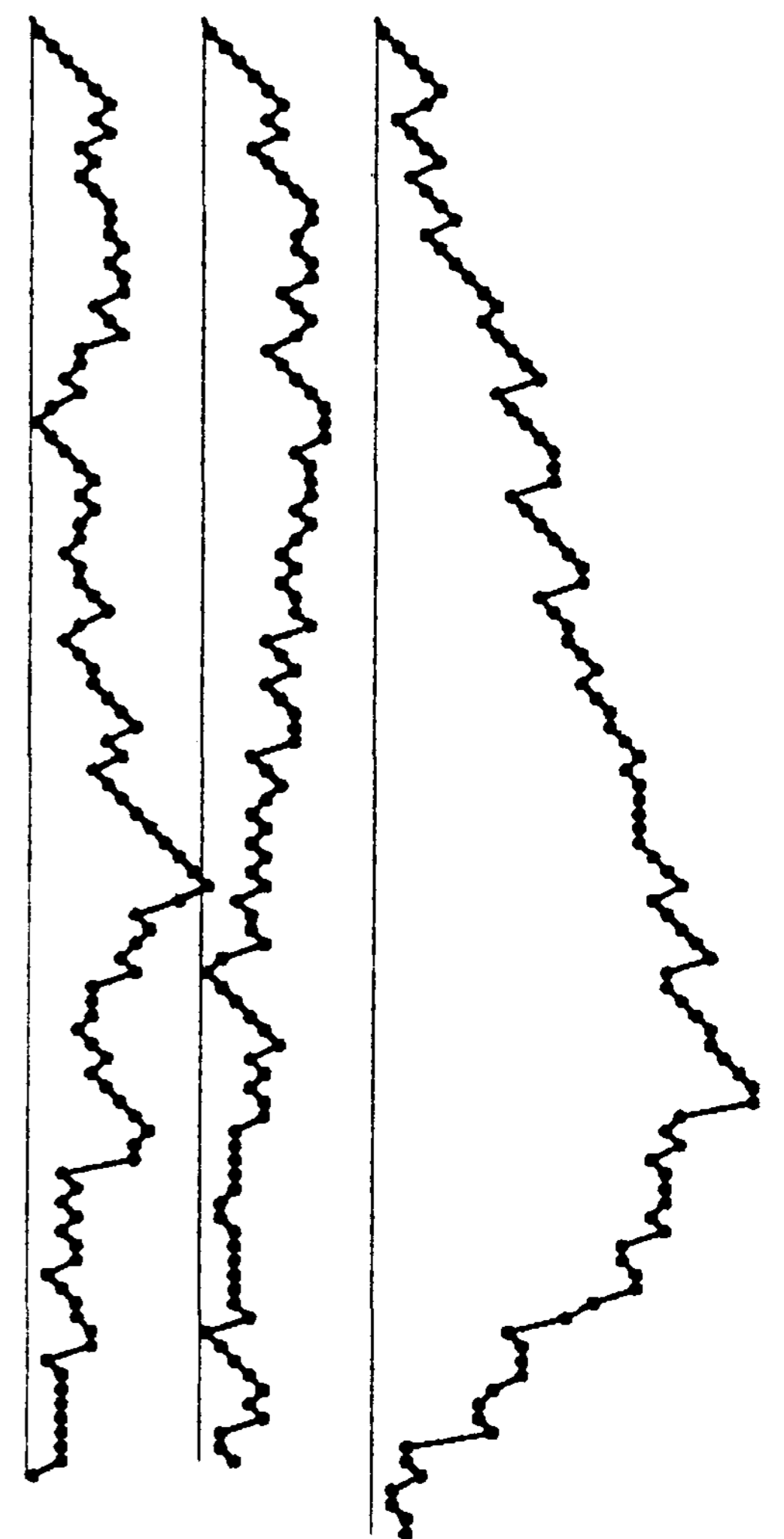
7.9. Сколько примерно операций сравнения понадобится быстрой сортировке (программа 7.1), чтобы выполнить сортировку файла, состоящего из  $N$  элементов, которые имеют два различных значения ключа ( $k$  элементов с одним значением и  $N - k$  элементов с другим значением)?

- 7.10. Написать программу, которая строит файл, представляющий собой наиболее благоприятный случай для быстрой сортировки: файл из  $N$  различных элементов, обладающих таким свойством, что каждое его разбиение генерирует подфайлы, которые отличаются друг от друга по размерам максимум на 1 элемент.

## 7.3. Размер стека

Так же как и в главе 3, для выполнения быстрой сортировки можно воспользоваться стеком магазинного типа, рассматривая его как стек, в котором в виде сортируемых подфайлов содержится перечень работ, которые предстоит выполнить. Каждый раз когда возникает необходимость в обработке подфайла, он выталкивается из стека. После деления файла получаются два подфайла, требующих дальнейшей обработки, которые и заталкиваются в стек. В рекурсивной реализации, представленной программой 7.1, стек, который поддерживается системой, содержит именно эту информацию.

Что касается файлов с произвольной организацией, то максимальный размер стека пропорционален значению  $\log N$  (см. раздел ссылок), однако в условиях вырожденного случая стек может расширяться до размеров, пропорциональных  $N$ , как показано на рис. 7.5. В самом деле, наиболее трудный случай имеет место тогда, когда файл уже отсортирован. Потенциальная возможность увеличения размеров стека до пропорциональных размерам сортируемого файла в условиях рекурсивной реализации быстрой сортировки представляет собой не очевидную, но в то же время вполне реальную проблему: в условиях этого метода сортировки всегда используется стек, а в вырожденном случае при работе с файлом большого размера подобное обстоятельство может послужить причиной аварийного останова программы ввиду нехватки памяти. Такое поведение совершенно недопустимо для библиотечной программы сортировки. (По всей вероятности, мы скорее столкнемся с проблемой недопустимо длительной сортировки, нежели с проблемой нехватки памяти.) Трудно гарантировано исключить подобное поведение программы, но как будет показано в разделе 7.5, нетрудно предусмотреть специальные средства, которые делают вероятность возникновения таких вырожденных случаев исключительно малой.



**РИСУНОК 7.5. РАЗМЕР СТЕКА ДЛЯ БЫСТРОЙ СОРТИРОВКИ.**

*Рекурсивный стек для быстрой сортировки не становится больше для файлов с произвольной организацией, но в то же время он может потребовать дополнительного пространства при работе с вырожденными файлами. На диаграмме представлены размеры стеков для двух файлов с произвольной организацией (слева, в центре) и размеры для частично упорядоченных файлов (справа).*

Программа 7.3 представляет собой нерекурсивную реализацию, которая решает данную проблему путем проверки размеров обоих подфайлов и размещения большего из них в стек первым. Рисунок 7.6 служит иллюстрацией такой стратегии. Сравнивая этот пример с приведенным на рис. 7.1, мы видим, что при такой стратегии подфайлы не меняются, меняется только порядок их обработки. Таким образом, мы сокращаем расход памяти без увеличения расхода времени.

Стратегия, которая заключается в том, что в стек помещается больший из двух подфайлов, приводит к тому, что каждая запись в стеке составляет не более чем половину записи, предшествующей ей в стеке, поэтому под стек отводится пространство памяти, достаточное для размещения примерно  $\lg N$  записей. Использование стека по максимуму имеет место, когда точка разделения приходится на середину файла. Что касается файлов с произвольной организацией, то фактический максимальный размер стека намного меньше; для вырожденных файлов его размер, по видимому, также будет небольшим.

### Программа 7.3. Нерекурсивная программная реализация быстрой сортировки.

Представленная ниже нерекурсивная реализация (см. главу 5) использует явно определенный стек магазинного типа, заменяя рекурсивные вызовы помещением в стек параметров, а вызовы процедур и выходы из них — циклом, который осуществляет выборку параметров из стека и их обработку, пока стек не пуст. Мы помещаем больший из двух подфайлов в стек первым с тем, чтобы максимальная глубина стека при сортировке  $N$  элементов не превосходила величины  $\lg N$ .

```
#include "STACK.cxx"
inline void push2(STACK<int> &s, int A, int B)
    { s.push(B); s.push(A); }
template <class Item>
void quicksort(Item a[], int l, int r)
    { STACK<int> s(50);
      push2(s, l, r);
      while (!s.empty())
        {
          l = s.pop(); r = s.pop();
          if (r <= l) continue;
          int i = partition(a, l, r);
          int (i-1 > r-i)
            { push2(s, l, i-1); push2(s, i+1, r); }
          else
            { push2(s, i+1, r); push2(s, l, i-1); }
        }
    }
```

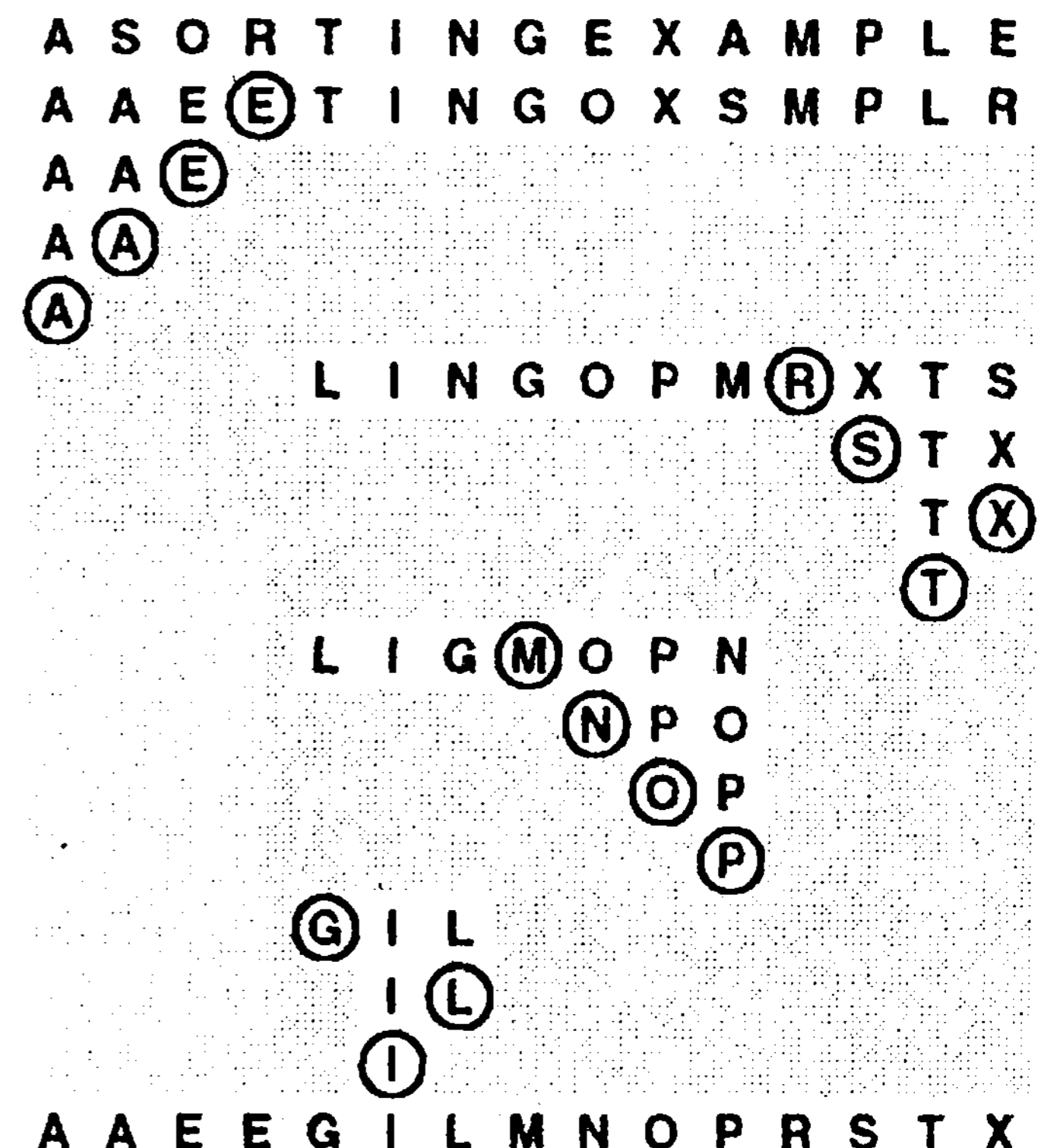


РИСУНОК 7.6. ПРИМЕР БЫСТРОЙ СОРТИРОВКИ (ПЕРВЫМ СОРТИРУЕТСЯ МЕНЬШИЙ ПОДФАЙЛ)

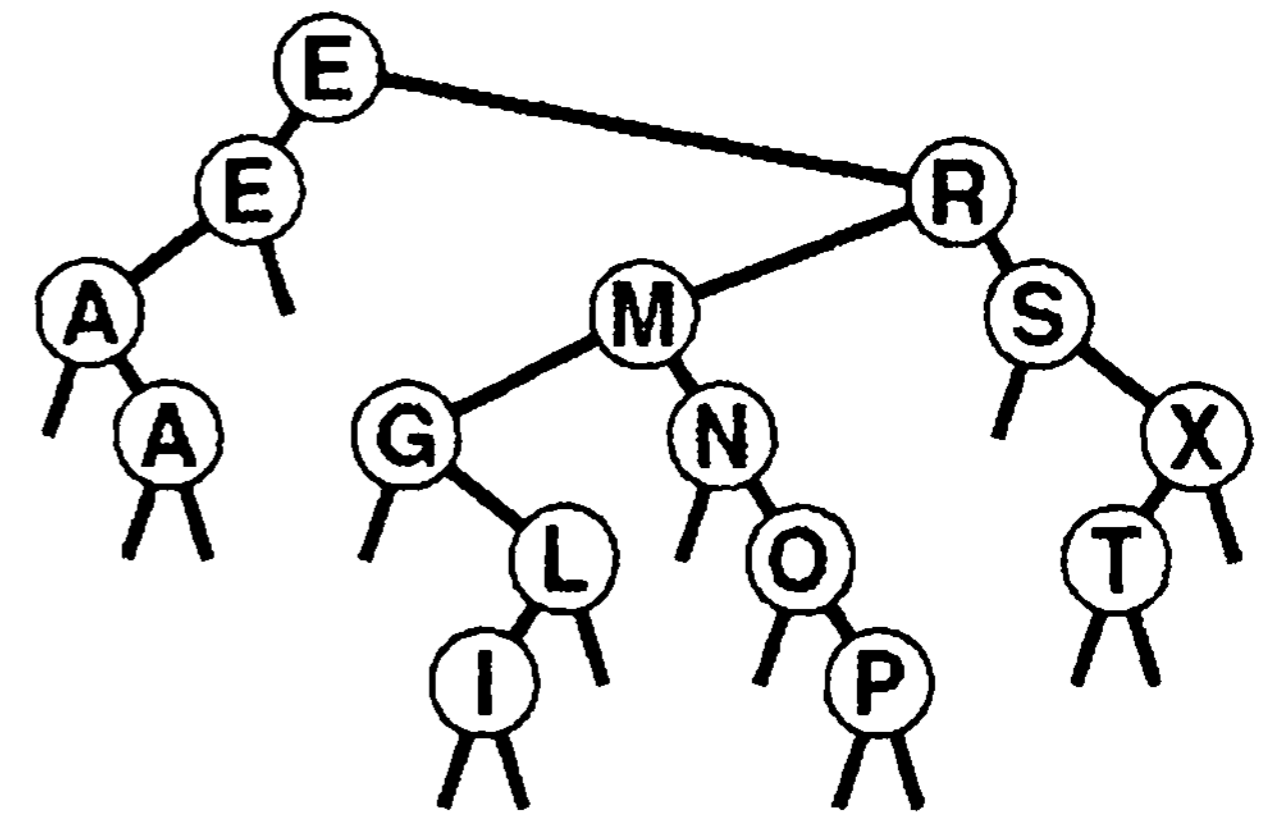
*Очередность, в которой производится обработка подфайлов, не препятствует корректному выполнению алгоритма быстрой сортировки и не приводит к увеличению времени выполнения сортировки, однако может повлиять на размеры стека магазинного типа, положенного в основу рекурсивной структуры. В рассматриваемом случае первым в обработке подвергается меньший из подфайлов, образованных в результате каждого разделения.*

**Лемма 7.3.** *Если меньший из двух подфайлов сортируется первым, то стек никогда не содержит более  $\lg N$  вхождений в случаях, когда для сортировки  $N$  файлов применяется быстрая сортировка.*

В наихудшем случае размер стека не должен превышать  $T_N$ , где  $T_N$  удовлетворяет рекуррентному соотношению  $T_N = T_{\lfloor N/2 \rfloor} + 1$  при  $T_1 = T_0 = 0$ . Это рекуррентное соотношение является стандартным и принадлежит к типу, рассмотренному в главе 5 (см. упражнение 7.13).

Этот метод не обязательно будет работать в по-настоящему рекурсивной реализации, поскольку он зависит от *очистки стека по окончании рекурсивной процедуры* (*end- либо tail-recursion removal*). Когда последним действием какой-либо процедуры является вызов другой процедуры, то в некоторых системах программирования действует следующее соглашение: локальные переменные удаляются из стека *раньше*, чем произойдет такой вызов, но отнюдь не после этого. Без такой очистки нельзя быть уверенным, что размер стека, используемого быстрой сортировкой, будет небольшим. Например, вызов быстрой сортировки с целью упорядочения уже отсортированного файла размером  $N$  приводит к рекурсивному вызову для упорядочения такого же файла, но размером  $N - 1$ , что, в свою очередь, приводит к рекурсивному вызову для файла размером  $N - 2$  и так далее, для чего в конечном итоге потребуется стек с глубиной, пропорциональной  $N$ . С учетом подобного обстоятельства сам собою напрашивается вывод о необходимости использования нерекурсивной реализации, что гарантировало бы от чрезмерного "разбухания" стека. С другой стороны, некоторые компиляторы C++ автоматически исключают завершающую очистку стека, и многие машины обеспечивают прямую аппаратную поддержку вызовов функций — поэтому нерекурсивная реализация, представленная программой 7.3, может на самом деле в такой среде оказаться медленнее рекурсивной реализации, представленной в программе 7.1.

Рисунок 7.7 служит иллюстрацией того, как нерекурсивный метод производит обработку тех же подфайлов (но в другом порядке), что и рекурсивный метод, применительно к любому файлу. На нем показана древовидная структура, в ко-



**РИСУНОК 7.7. РАЗДЕЛЯЮЩЕЕ ДЕРЕВО БЫСТРОЙ СОРТИРОВКИ**

*Если мы сожмем диаграммы разделения, представленные на рис. 7.6 и 7.1, соединив каждый разделяющий элемент с разделяющими элементами, использованными в двух его подфайлах, то получим показанное на данной диаграмме статическое представление процесса разделения (для обоих случаев). В этом бинарном дереве каждый подфайл представлен своим разделяющим элементом (или самим собой, если он имеет размер 1), и поддеревья каждого узла суть деревья, представляющие подфайлы после разделения. Дабы не загромождать рисунок, нулевые подфайлы на нем не показаны, хотя наши рекурсивные версии алгоритма выполняют рекурсивные вызовы при выполнении условия  $r < l$ , когда разделяющим элементом становится наименьший или наибольший элемент файла. Само по себе дерево не зависит от очередности, в которой подфайлы подвергаются разделению. Наша рекурсивная реализация сортировки соответствует посещению узлов при их обходе в прямом порядке, а нерекурсивная реализация соответствует правилу посещения сначала наименьшего дерева.*

тором корневым элементом служит разделяющий элемент, а также порожденные при этом левое и правое поддеревья, соответствующие левому и правому подфайлам и являющиеся, соответственно, левым и правым потомками. Использование рекурсивной реализации, представленной программой 7.3, соответствует просмотру этих узлов в прямом порядке; нерекурсивная реализация соответствует правилу просмотра сначала наименьшего дерева.

Когда используется явно заданный стек, что, собственно говоря, и делалось в программе 7.3, удаётся избежать некоторых непроизводительных затрат, характерных для рекурсивных реализаций, хотя современные системы программирования не привносят больших непроизводительных затрат в столь простые программы. Программу 7.3 можно улучшать и дальше. Например, она помещает в стек оба подфайла, но только подфайл, хранящийся в вершине стека, доступен в любой момент; такое положение дел можно изменить, явно объявив специальные переменные  $r$  и  $l$ . Итак, производится проверка условия  $r < l$  по мере того, как подфайлы выбираются из стека, в то время как намного эффективнее вообще не сохранять в стеке файлы, удовлетворяющие упомянутому условию (упражнение 4.14). Эта мера, на первый взгляд, может показаться несущественной, однако рекурсивный характер быстрой сортировки фактически приводит к тому, что значительная часть подфайлов в процессе быстрой сортировки имеет размеры 1 или 0. Далее рассматривается важное усовершенствование быстрой сортировки, которое обеспечивает повышение ее эффективности за счет распространения этой идеи на все файлы небольших размеров.

## Упражнения

- ▷ 7.11. В стиле рис. 7.11 представить содержимое стека после каждой пары операций *помещения* (*push*) в стек и *выталкивания* (*pop*) из стека, когда программа 7.3 используется для сортировки файла, содержащего ключи **E A S Y Q U E S T I O N**.
- ▷ 7.12. Выполнить задание, сформулированное в упражнении 7.11, для случая, когда в стек сначала помещается правый подфайл, а затем левый подфайл (как это принято в рекурсивной реализации).
- 7.13. Завершить доказательство леммы 7.3, воспользовавшись для этой цели методом индукции.
- 7.14. Внесите в программу 7.3 такие изменения, чтобы она не помещала в стек подфайлы, удовлетворяющие условию  $r \leq l$ .
- ▷ 7.15. Вычислить максимальный размер стека, затребованного программой 7.3, когда  $N = 2^n$ .
- 7.16. Вычислить максимальные размеры стека, затребованного программой 7.3, когда  $N = 2^n - 1$  и  $N = 2^n + 1$ .
- 7.17. Имеет ли смысл использовать для нерекурсивной реализации быстрой сортировки вместо стека очередь? Предъявите аргументы для обоснования своих ответов.
- 7.18. Выясните и сообщите, практикует ли ваша система программирования очистку стека по завершении рекурсивной процедуры.



- **7.19.** Выполните эмпирические исследования с целью определения среднего размера стека, используемого базовым рекурсивным алгоритмом для сортировки файла с произвольной организацией, состоящего из  $N$  элементов, причем  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **7.20.** Найти среднее число подфайлов размера 0, 1 и 2, когда быстрая сортировка используется для сортировки произвольно организованного файла из  $N$  элементов.

## 7.4. Подфайлы небольших размеров

Заметное повышение эффективности быстрой сортировки следует из того факта, что рекурсивная программа гарантировано вызывает сама себя для работы со множеством подфайлов небольших размеров. Следовательно, когда она сталкивается с подфайлами небольших размеров, она обязана использовать по возможности самый лучший метод работы с ними. Один из очевидных способов достижения данной цели предусматривает соответствующую проверку в начале рекурсивной программы на участке от оператора `return` до вызова сортировки методом вставки, например,

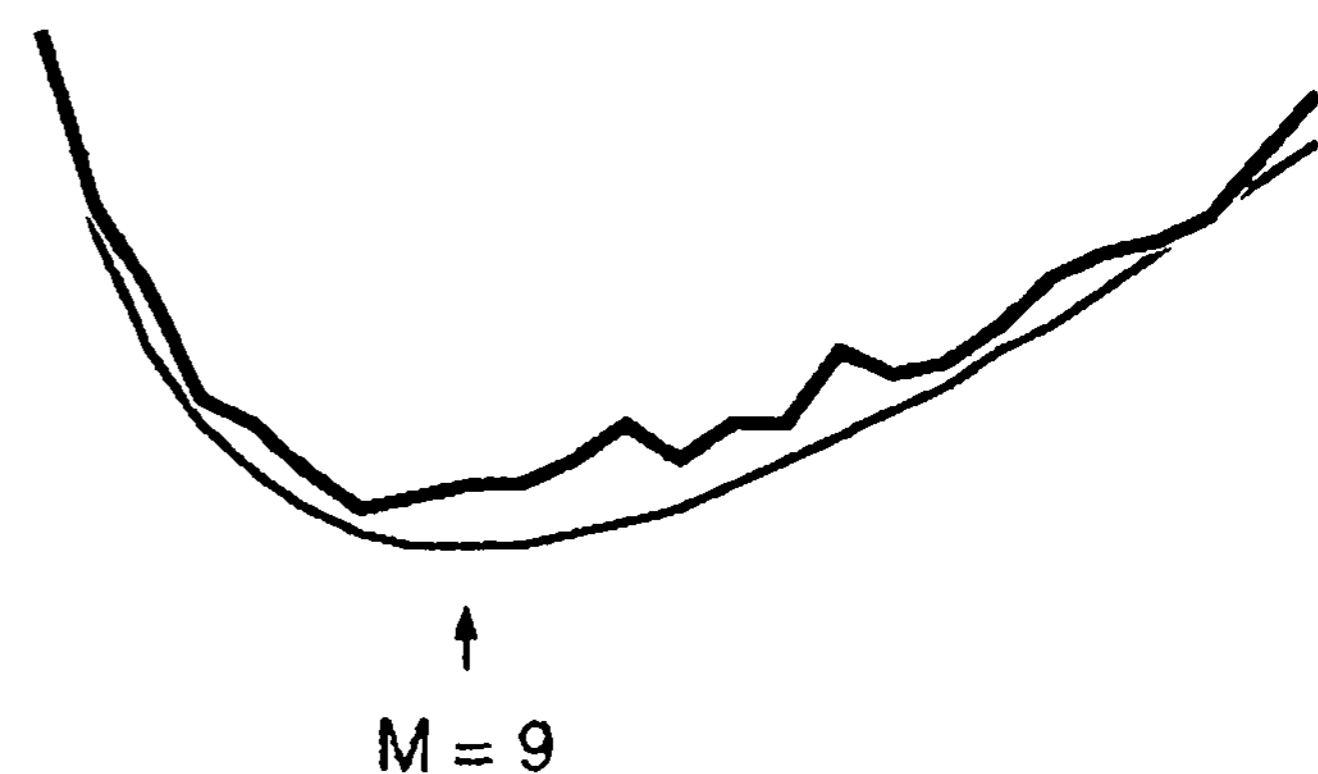
```
if (r-1 <= M) insertion(a, l, r);
```

Здесь  $M$  — это некоторый параметр, точное значение которого зависит от реализации. Мы можем определить наилучшее значение  $M$  путем анализа либо эмпирических исследований. Обычно в результате подобных исследований выясняется, что время выполнения мало меняется, если  $M$  принимает значения в диапазоне примерно от 5 до 25, при этом значение времени выполнения, когда  $M$  попадает в этот диапазон, отличается от значения этого показателя при естественном выборе  $M=1$  не более, чем на 10 процентов (см. рис.7.8).

Несколько более простой и чуть более эффективный по сравнению с сортировкой вставками способ обращения с подфайлами небольших размеров по мере их появления состоит в том, чтобы поменять проверку в начале программы на

```
if (r-1 <= M) return;
```

Другими словами, в процессе разделения небольшие подфайлы просто игнорируются. В условиях нерекурсивной реализации это можно сделать, отказавшись от помещения в стек любых файлов с размерами меньшими  $M$ , либо игнорируя все файлы с размерами меньшими  $M$ , которые будут обнаруживаться в стеке. По окончании операции разделения получается практически отсортированный файл. При этом, как уже отмечалось в разделе 6.5, метод вставок является наилучшим методом сортировки подобного рода файлов. То есть, сортировка вставками работает с таким файлом



**РИСУНОК 7.8. ОТСЕЧЕНИЕ ФАЙЛОВ МАЛЫХ РАЗМЕРОВ**

*Выбор оптимального значения размера в целях отсечения небольших файлов приводит к уменьшению среднего времени выполнения сортировки на 10 процентов. Выбор точного значения не критичен; значения этого показателя в достаточно широких пределах (приблизительно от 5 до 20) дает практически одинаково хорошие результаты для большей части приложений. Жирная ломаная линия (сверху) получена эмпирическим путем; тонкая линия (снизу) была рассчитана аналитически.*

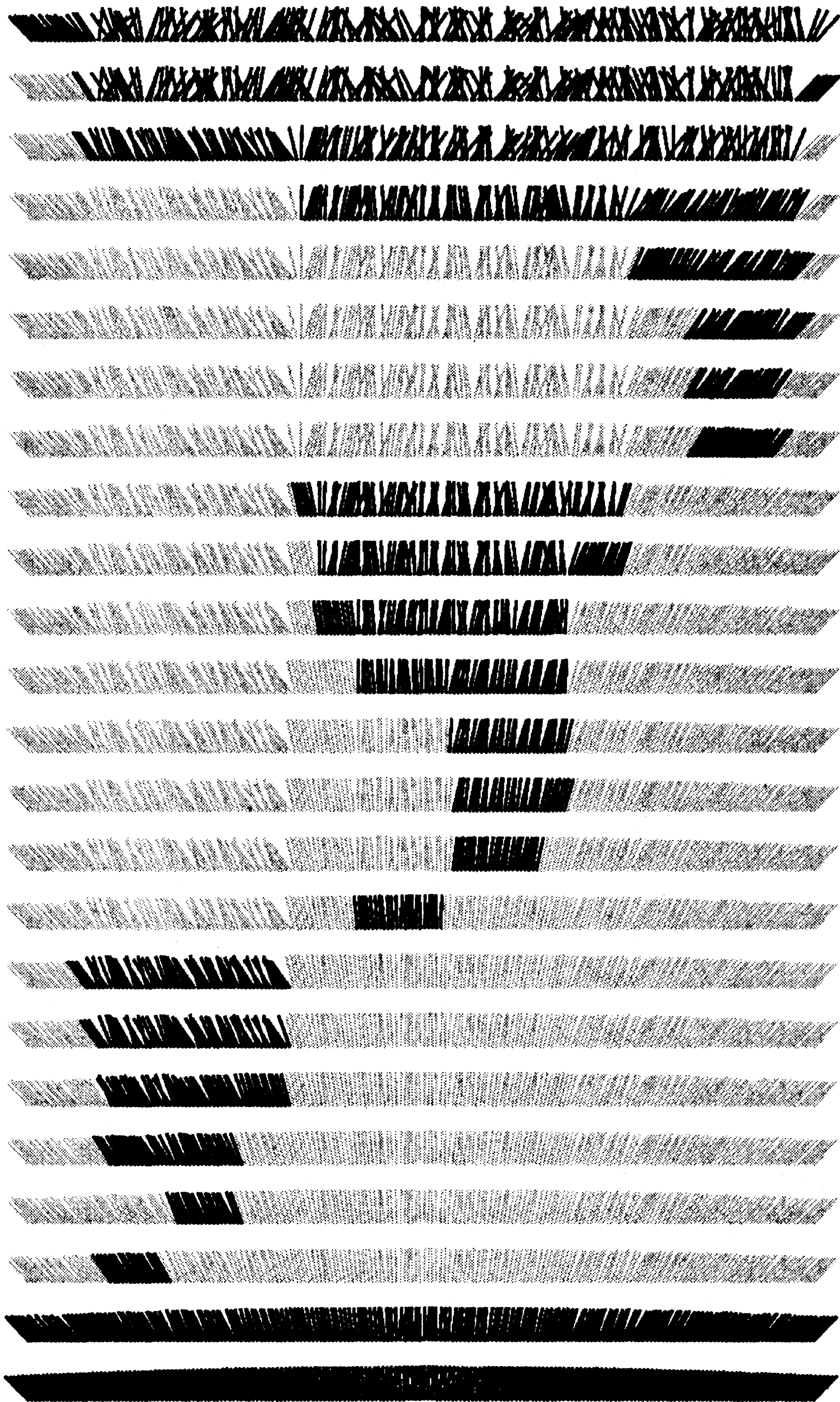
почти так же хорошо, как и с совокупностью файлов небольших размеров, когда они подвергаются сортировке вставками непосредственно. При использовании данного метода следует соблюдать осторожность, ибо сортировка вставками скорее всего будет работать, даже если алгоритм быстрой сортировки содержит фатальную ошибку, из-за которой эта сортировка просто функционировать не будет. Только резкое возрастание затрат ресурсов может служить сигналом о том, что что-то не в порядке.

Рисунок 7.9 иллюстрирует этот процесс на примере крупного файла. Даже при сравнительно радикальном отсечении файлов небольших размеров та часть программы, которая выполняет быструю сортировку, выполняется быстро, поскольку в процесс разделения вовлечено относительно небольшое количество элементов. Сортировка вставками, завершающая выполнение программы, также выполняется быстро, поскольку на обработку ей передается почти упорядоченный файл.

Этот метод с большой пользой можно применять всякий раз, когда мы имеем дело с рекурсивным алгоритмом. В силу особенностей рекурсивных алгоритмов можно быть уверенным в том, что *все* они основную часть своего времени будут заняты решением небольших задач; в любом случае для работы с небольшими файлами в нашем распоряжении имеются алгоритмы "решения в лоб" с низкими непроизводительными затратами. Благодаря такому обстоятельству в общем случае можно улучшить общие показатели производительности с помощью гибридных алгоритмов.

## Упражнения

- 7.21. Нужны ли служебные ключи, если сортировка вставками вызывается непосредственно из быстрой сортировки?
- 7.22. Снабдить программу 7.1 инструментальными средствами, которые позволили бы подсчитать процент операций сравнения при разбиении файлов, размеры которых не превосходят 10, 100 и 1000 элементов, и вывести на печать значения, которые принимают это процентное отношение для случаев сортировки файлов с произвольной организацией, состоящих из  $N$  элементов для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 7.23. Реализовать рекурсивный вариант быстрой сортировки с отсечением для сортировки вставками подфайлов, размеры которых не превышают  $M$  элементов, и эмпирически определить значения  $M$ , при которых программа достигает максимального быстродействия в вашей вычислительной среде при сортировке файлов с произвольной организацией, состоящих из  $N$  элементов для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 7.24. Решить задачу, сформулированную в упражнении 7.23, воспользовавшись не-рекурсивной реализацией.
- 7.25. Решить задачу, сформулированную в упражнении 7.23, для случая, когда сортируемые записи содержат  $a$  ключей и  $b$  указателей на другую информацию (но мы не используем сортировку по указателям).
- 7.26. Написать программу, которая вычерчивает гистограмму (см. программу 3.7) для размеров подфайлов, передаваемых в сортировку вставками, при выполнении сортировки файла размером  $N$  с отсечением подфайлов с размерами, не превышающими  $M$ . Выполнить эту программу для  $M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .



**РИСУНОК 7.9. СРАВНЕНИЯ В БЫСТРОЙ СОРТИРОВКЕ**

Подфайлы в условиях быстрой сортировки обрабатываются независимо друг от друга. На этой диаграмме показан результат разбиения каждого подфайла в процессе сортировки 200 элементов с отсечением файлов размером 15 и меньше. Получить приближенное представление об общем числе сравнений можно, сосчитав количество отмеченных элементов в вертикальных столбцах. В рассматриваемом случае каждая позиция массива во время сортировки вовлекается только в шесть или семь подфайлов.

7.27. Проведите эмпирические исследования с целью определения среднего размера стека, используемого при быстрой сортировке с отсечением подфайлов с размерами, не превышающими  $M$ , для сортировки файла с произвольной организацией, состоящего из  $N$  элементов, причем  $M=10, 100$  и  $1000$  и  $N=10^3, 10^4, 10^5$  и  $10^6$ .

## 7.5 Метод разделения с вычислением медианы из трех элементов

Еще одно усовершенствование метода быстрой сортировки заключается в использовании такого разделяющего элемента, который с достаточно большой вероятностью делил бы файл вблизи его середины. Наиболее безопасный выбор, минимизирующий вероятность возникновения наихудшего случая, состоит в использовании в качестве разделяющего элемента случайного элемента массива. Тогда вероятность возникновения наихудшего случая становится ничтожно малой. Этот метод представляет собой пример *вероятностного алгоритма (probabilistic algorithm)* — такого алгоритма, который использует случайный характер величин для достижения высокой эффективности с большой вероятностью, независимо от степени упорядоченности входных данных. Далее в этой книге мы столкнемся с многочисленными примерами использования свойства случайности при разработке структуры алгоритмов, в частности, когда предполагается наличие той или иной тенденции во входных данных. На практике использование в рамках быстрой сортировки генератора случайных чисел с этой целью может оказаться излишним: простой произвольный выбор оказывается достаточно эффективным.

Другой хорошо известный способ нахождения подходящего разделяющего элемента заключается в том, что производится выборка трех элементов из файла, затем в качестве разделяющего элемента используется медиана из этих трех элементов. Выбирая для такой цели три элемента из левой части, из середины и из правой части массива, можно также включить в эту схему служебные метки: сначала сортируем три выбранных элемента (с использованием метода трех обменов, описанного в главе 6), затем меняем местами элемент из середины с элементом  $a[r-1]$ , далее выполняем алгоритм разделения на элементах  $a[l+1], \dots, a[r-2]$ . Приведенное усовершенствование получило название *метода медианы из трех элементов (median-of-three)*.

Метод медианы из трех элементов повышает эффективность сортировки по трем направлениям. Во-первых, он существенно снижает вероятность возникновения наихудшего случая для любой реальной сортировки. Чтобы сортировка выполнялась за время, пропорциональное  $N^2$ , два из трех проверяемых элементов должны быть в числе наибольших и наименьших элементов файла, и это событие должно последовательно повторяться во время большей части процессов разделения. Во-вторых, он устраняет необходимость в служебном ключе в процессе разделения, поскольку для выполнения этой функции вполне достаточно одного элемента из числа тех, которые подвергаются проверке до начала разделения. В-третьих, он уменьшает среднее время выполнения алгоритма примерно на 5 процентов.

Сочетание использования метода медианы из трех элементов с отсечением подфайлов небольших размеров может уменьшить среднее время выполнения быстрой сортировки по сравнению с аналогичным показателем естественной рекурсивной

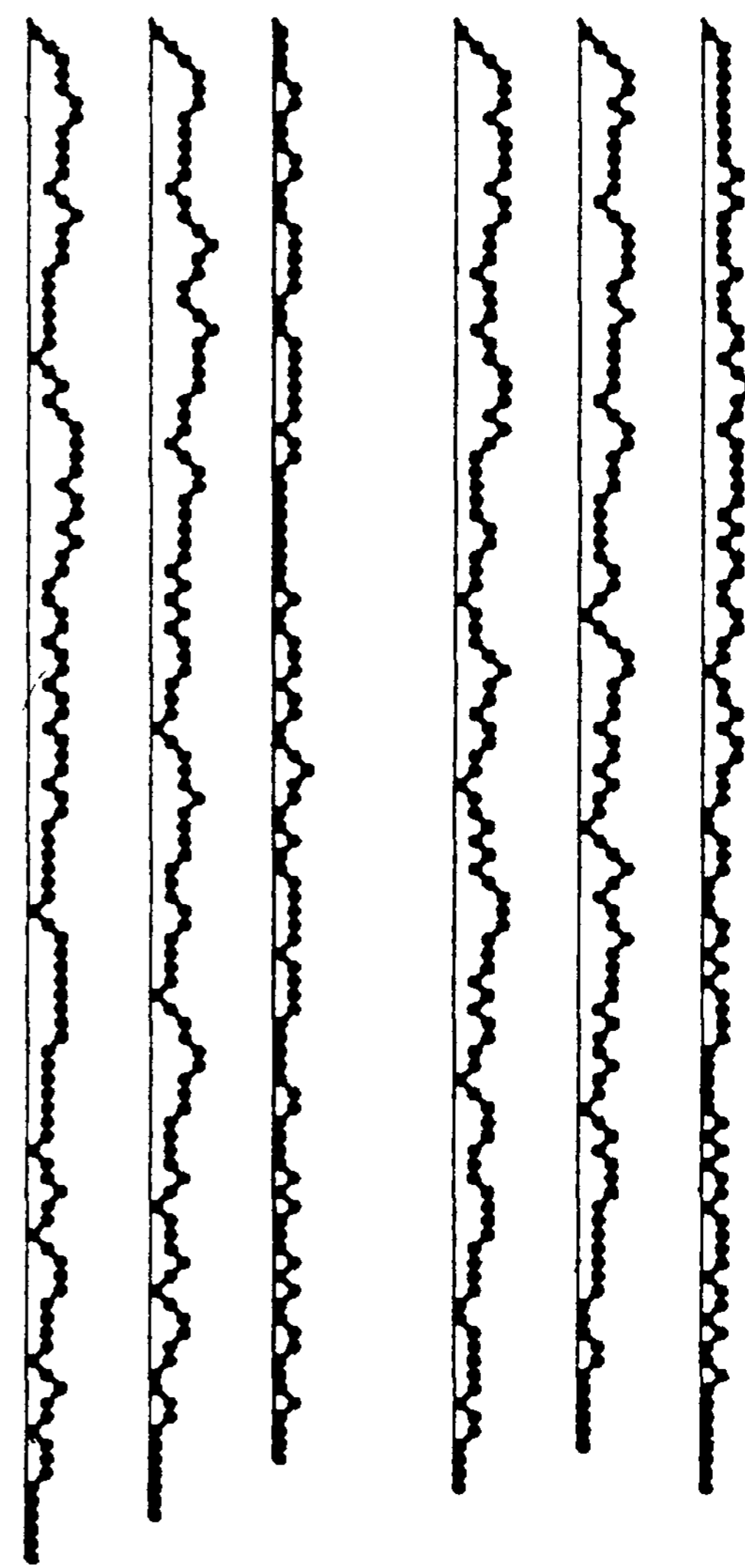
сортировки на 20–25 процентов. Программа 7.4 представляет собой реализацию, в которой применены все упомянутые усовершенствования.

Можно рассмотреть и другие способы совершенствования программы: отказ от рекурсии, замена вызовов подпрограмм встроенными кодами, использование служебных меток и т.п. Однако в современных машинах такие вызовы процедур обычно эффективны и они не включаются во внутренние циклы. Что более важно, применение отсечения подфайлов небольших размеров во многих случаях позволяют скомпенсировать возможные непроизводительные затраты (за пределами внутреннего цикла). Главные аргументы в пользу нерекурсивных реализаций с явно определенным стеком заключаются в том, чтобы получить гарантии, что можно пользоваться стеками ограниченных размеров (см. рис. 7.10).

Возможно дальнейшее улучшение алгоритма (например, можно использовать медиану из пяти или большего числа элементов), однако величина сэкономленного времени для файлов с произвольной организацией незначительна. Мы можем получить большую экономию времени за счет кодирования внутренних циклов или всей программы на языке ассемблера или на машинном языке. Эти выводы были многократно подтверждены специалистами на примерах солидных приложений сортировок (см. раздел ссылок).

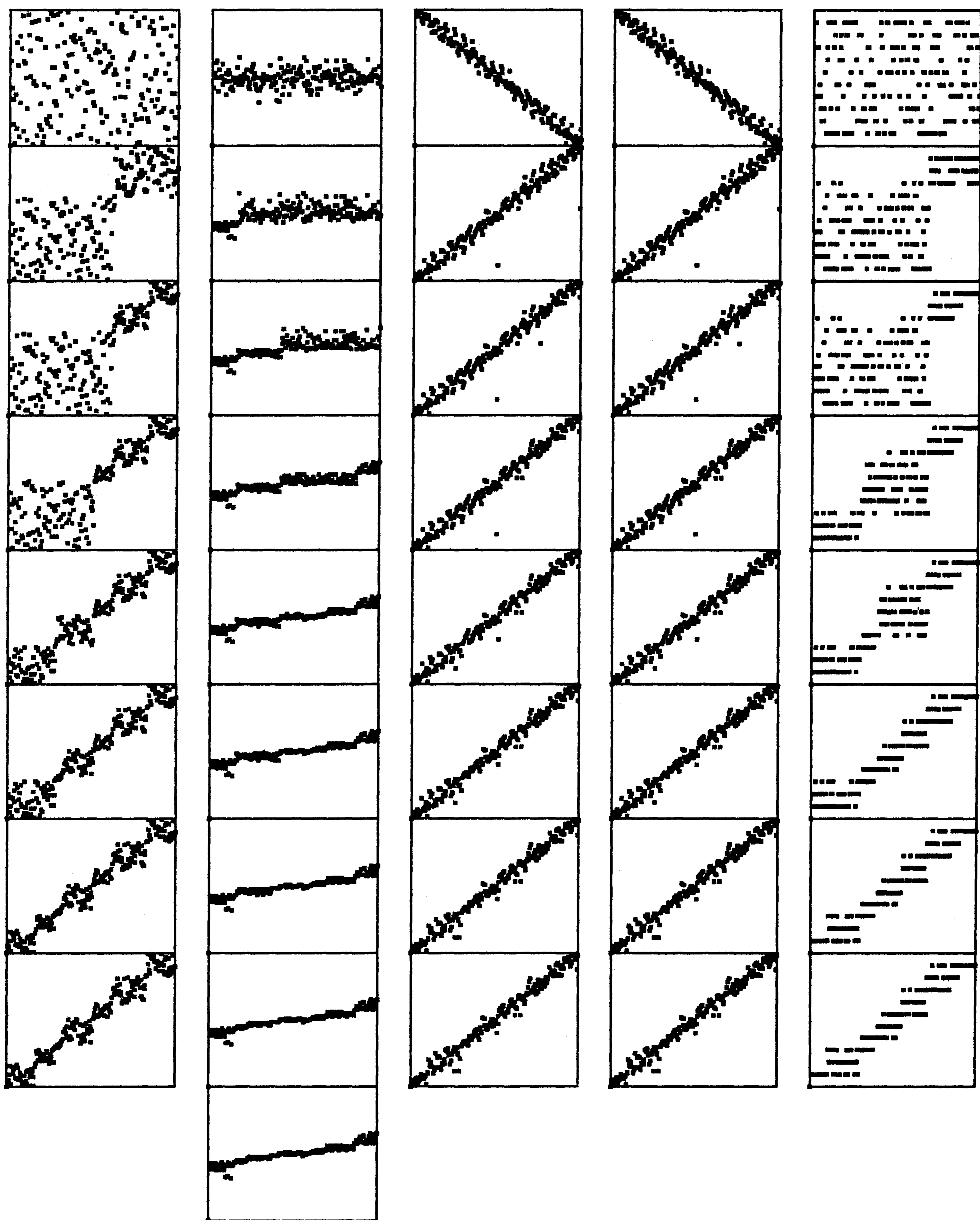
Для файлов с произвольной организацией первый обмен, выполняемый программой 7.4, излишний. Мы включили его в программу не только из-за того, что он обеспечивает оптимальное разделение для уже упорядоченных файлов, но еще и потому, что оно служит защитой от нештатных ситуаций, могущих возникнуть на практике (см. например, упражнение 7.33). Рисунок 7.11 иллюстрирует эффективность использования среднего элемента в процессе выбора разделяющего элемента для различных типов файлов.

Метод медианы из трех элементов представляет собой специальный случай общей идеи, заключающейся в том, что для файла неизвестного типа можно произвести выборку и использовать свойства полученной выборочной совокупности, чтобы дать оценку всему файлу.



**РИСУНОК 7.10. РАЗМЕРЫ СТЕКОВ ДЛЯ УЛУЧШЕННЫХ ВАРИАНТОВ БЫСТРОЙ СОРТИРОВКИ**

*Сортировка меньшего из подфайлов, образовавшихся в результате разделения, первым, гарантирует, что размер стека в худшем случае находится в логарифмической зависимости от размера исходного файла. На диаграмме отображены размеры для тех же файлов, что и представленные на рис. 7.5, при этом в трех первых случаях (слева) применяется метод сортировки меньшего подфайла первым, в остальных трех случаях (справа) к этому еще добавляется метод медианы трех. По этим диаграммам трудно судить о времени выполнения; этот показатель зависит от размера файлов в стеке, а не от их числа. Например, третий файл (частично отсортирован) не требует большого стекового пространства, однако вызывает замедление сортировки, поскольку размеры обрабатываемых подфайлов большие.*



**РИСУНОК 7.11. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ БЫСТРОЙ СОРТИРОВКИ С ВЫЧИСЛЕНИЕМ МЕДИАНЫ ИЗ ТРЕХ ЭЛЕМЕНТОВ НА ФАЙЛАХ РАЗЛИЧНЫХ ТИПОВ.**

*Модификация быстрой сортировки, предусматривающая вычисление медианы из трех элементов (в частности, с использованием среднего элемента) обеспечивает приличные результаты в плане придания процессу разделения большей устойчивости. Особенно хорошо этот метод проявляет себя при сортировке вырожденных файлов, показанных на рис. 7.4. Другой вариант, позволяющий достичь тех же целей, — использование случайного разделяющего элемента.*

Для быстрой сортировки мы хотим получить оценку медианы, чтобы решить, как проводить процедуру разделения. Свойство этого алгоритма заключается в том, что нам не нужна особо точная оценка (мы вообще можем обойтись без такой оценки, если стоимость ее вычисления высока); мы просто хотим избежать исключительно плохой оценки. Если используется случайная выборка из одного элемента, получается рандомизированный алгоритм, который виртуально обладает высоким быстродействием, независимо от природы входных данных. Если произвести случайную выборку из файла из трех или из пяти элементов, а затем воспользоваться медианой из этой выборки в процедуре разделения, получится лучшее разбиение, но такое усовершенствование достигается ценой выполнения выборки.

#### Программа 7.4. Улучшенная быстрая сортировка

Выбор медианы из первого, среднего и конечного элементов в качестве разделяющего элемента и отсечение рекурсии меньших подфайлов может привести к существенному повышению эффективности быстрой сортировки. Данная реализация осуществляет разделение по медиане из первого, среднего и конечного элементов массива (не следует использовать эти элементы в процессе разделения для других целей). Файлы длиной 11 и меньше в процессе разделения игнорируются; затем для окончания сортировки используется команда **insertion** из главы 6.

```
static const int M = 10;
template <class Item>
void quidcksort(Item a[], int l, int r)
{
    if (r-1 <= M) return;
    exch(a[(l+r)/2], a[r-1]);
    comexch(a[l], a[r-1]);
    comexch(a[l], a[r]);
    comexch(a[r-1], a[r]);
    int i = partition(a, l+1, r-1);
    quidcksort(a, l, i-1);
    quidcksort(a, i+1, r);
}
template <class Item>
void hybridsort(Item a[], int l, int r)
{ quidcksort(a, l, r); insertion(a, l, r); }
```

Быстрая сортировка нашла широкое применение в связи с тем, что она успешно протекает в различных ситуациях. Другие методы хорошо работают в некоторых специальных случаях, которые время от времени встречаются на практике, но быстрая сортировка успешно решает гораздо большее число сортировочных задач, чем это можно сделать с применением других методов, а ее быстродействие зачастую гораздо выше, чем в условиях применения альтернативных подходов. В табл. 7.1 представлены эмпирические результаты, которые могут служить подтверждением некоторых из сделанных выше выводов.

#### Таблица 7.1. Эмпирическое исследование алгоритмов быстрой сортировки

Быстрая сортировка (программа 7.1) работает примерно в два раза быстрее, чем сортировка методом Шелла (программа 6.6). Совершенствование метода отсечения меньших подфайлов и метода медианы из трех элементов (программа 7.4) сокращают время выполнения сортировки примерно на 10 процентов каждое.

N	Метод Шелла	Быстрая сортировка			Быстрая сортировка с вычислением медианы из трех элементов		
		M = 0	M = 10	M = 20	M = 0	M = 10	M = 20
12500	6	2	2	2	3	2	3
25000	10	5	5	5	5	4	6
50000	26	11	10	10	12	9	14
100000	58	24	22	22	25	20	28
200000	126	53	48	50	52	44	54
400000	278	116	105	110	114	97	118
800000	616	255	231	241	252	213	258

## Упражнения

**7.28.** Наша реализация метода медианы из трех элементов тщательно следит за тем, чтобы элементы, составляющие выборку, не принимали участия в процессе разделения. Одна из причин заключается в том, что они могут быть использованы в качестве служебных меток. Назовите другие причины.

**7.29.** Реализовать быструю сортировку на базе разделения по медиане случайной выборки из файла, содержащей пять элементов. Элементы выборочной совокупности не должны принимать участия в разделении (см. упражнение 7.28). Сравните эффективность вашего алгоритма с эффективностью метода медианы из трех элементов на примерах крупных файлов с произвольной организацией.

**7.30.** Выполните программу из упражнения 7.29 на крупных файлах со специальной организацией — например, отсортированные файлы, файлы, упорядоченные в обратном порядке или файлы с одинаковыми ключами элементов. Насколько ее эффективность при сортировке указанных файлов отличается от эффективности, полученной для файлов с произвольной организацией?

- **7.31.** Реализовать быструю сортировку с использованием выборочных совокупностей размером  $2^k - 1$ . Сначала выполнить сортировку выборочной совокупности, затем выполнить рекурсивную программу, осуществляющую разделение с использованием медианы из элементов выборочной совокупности, и поместить обе половины оставшейся части выборочной совокупности в каждый подфайл таким образом, чтобы они могли быть использованы в этих подфайлах без дальнейшей сортировки. Такой метод сортировки называется *сортировкой методом случайной выборки (samplesort)*.
- **7.32.** Выполнить эмпирические исследования, чтобы определить наилучший размер выборочной совокупности для сортировки методом случайной выборки (см. упражнение 7.31) для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Имеет ли значение, какой вид сортировки используется для упорядочения выборочной совокупности: быстрая сортировка или сортировка методом случайной выборки (samplesort)?
- **7.33.** Показать, что если внести изменения в программу 7.4, предусматривающие исключение первой операции обмена и пропуск ключей, равных разделяющему элементу, то время выполнения сортировки файла, организованного в обратном порядке, находится в квадратичной зависимости от длины файла.

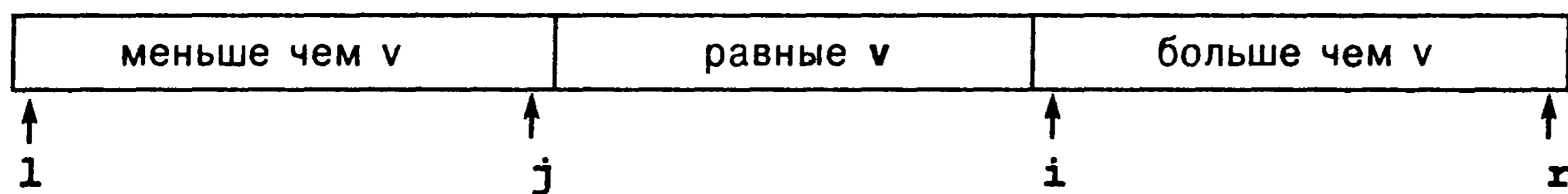


## 7.6. Дублированные ключи

Файлы с большим числом дублированных сортируемых ключей довольно часто встречаются в различных приложениях. Например, может потребоваться сортировка большого файла с персональными данными по году рождения или, скажем, сортировка для деления персонала по половому признаку.

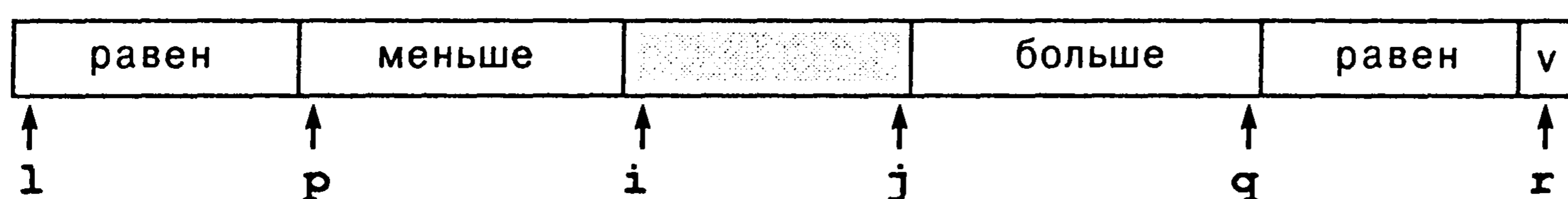
Когда в сортируемом файле имеется множество дублированных ключей, нельзя со всей определенностью утверждать, что рассмотренные нами различные реализации быстрой сортировки показывают недопустимо низкую эффективность, однако их можно существенно улучшить. Например файл, который состоит исключительно из равных друг другу ключей (одно и то же значение), вовсе не нуждается в дальнейшей сортировке, однако наши реализации продолжают процесс разделения, подвергая обработке все более мелкие подфайлы независимо от того, насколько большим является исходный файл (см. упражнение 7.8). В ситуации, когда во входном файле присутствует большое число дублированных ключей, рекурсивная природа быстрой сортировки приводит к тому, что подфайлы, содержащие только элементы с одним и тем же ключом, встречаются довольно часто, благодаря чему существуют большие потенциальные возможности для совершенствования алгоритма сортировки.

Одна достаточно простая идея заключается в делении файла на *три* части, одна — для ключей, меньших разделяющего элемента, другая — для ключей, равных ему, и третья — для ключей, больших разделяющего элемента:



Выполнение такого разбиения намного сложнее, чем разбиение на две части, которым мы пользовались ранее. Были предложены различные методы решения этой задачи. Классическим упражнением по программированию, получившим широкую известность с легкой руки Дейкстры (Dijkstra), стала Задача голландского национального флага, и прежде всего из-за того, что три возможных категорий ключей могут соответствовать трем цветам флага (Dutch National Flag problem) (см. раздел ссылок). В рамках быстрой сортировки мы добавляем еще одно ограничение, заключающееся в том, что эта задача должна быть решена за один проход по файлу — алгоритм, который предусматривает два прохода по данным, замедлит быструю сортировку в два раза, даже если в исходном файле вообще не будет дублированных ключей.

Оригинальный метод, предложенный Бентли (Bentley) и Макилроем (McIlroy) в 1993 г. для разбиения файла на три части, представляет собой модификацию стандартной схемы разделения и предусматривает следующее: ключи, равные разделяющему элементу и встретившиеся в левом подфайле, накапливаются в левом конце файла, ключи, равные разделяющему элементу и встретившиеся в правом подфайле, накапливаются в правом конце файла. Во время выполнения процесса разделения мы придерживаемся следующей схемы:



Далее, когда указатели пересекутся и точное местонахождение равных ключей станет известным, мы перемещаем в эту позицию все элементы с ключами, равными разделяющему элементу. Эта схема не совсем удовлетворяет требованию, чтобы разбиение файла на три части было завершено за один проход, однако непроизводительные расходы системных ресурсов пропорциональны только количеству обнаруженных дублированных ключей. Этот факт обуславливает две особенности: во-первых, этот метод работает хорошо, даже если в исходном файле вообще нет дублированных ключей, поскольку в этом случае непроизводительные затраты отсутствуют. Во-вторых, для этого метода характерна линейная зависимость времени выполнения от длины файла при постоянном числе значений ключей: каждая фаза разделения исключает из процесса сортировки все ключи со значениями, равными значению разделяющего элемента, так что каждый ключ может быть использован максимум при постоянном числе разделений.

Рисунок 7.12 служит иллюстрацией работы алгоритма разбиения на три части на примере учебного файла, а программа 7.5 содержит реализацию быстрой сортировки, в основу которой положен этот метод. Рассматриваемая реализация требует добавления двух операторов `if` в цикл обмена и двух циклов `for` с тем, чтобы процедура разделения завершалась помещением ключей, равных разделяющему элементу, в окончательные позиции. По-видимому, на это потребуется меньше программного кода, чем в случае других альтернатив разделения файла на три части. И что более важно, этот метод не только исключительно эффективно решает проблему дублированных ключей, но и приносит минимально возможный объем непроизводительных затрат в случае, когда в исходном файле вообще нет дублированных ключей.

### Программа 7.5. Быстрая сортировка с разделением на три части

В основу программы положено разделение массива на три части: на элементы, меньшие разделяющего элемента (в позиции  $a[1], \dots, a[j]$ ), элементы, равные разделяющему элементу (в позиции  $a[j+1], \dots, a[i-1]$ ), и элементы большие разделяющего элемента (в позиции  $a[i], \dots, a[r]$ ). После этого сортировка завершается двумя рекурсивными вызовами.

Чтобы достичь поставленной цели, программа содержит ключи, равные разделяющему элементу, слева между  $l$  и  $q$  и справа между  $q$  и  $r$ . В разделяющем цикле, ког-

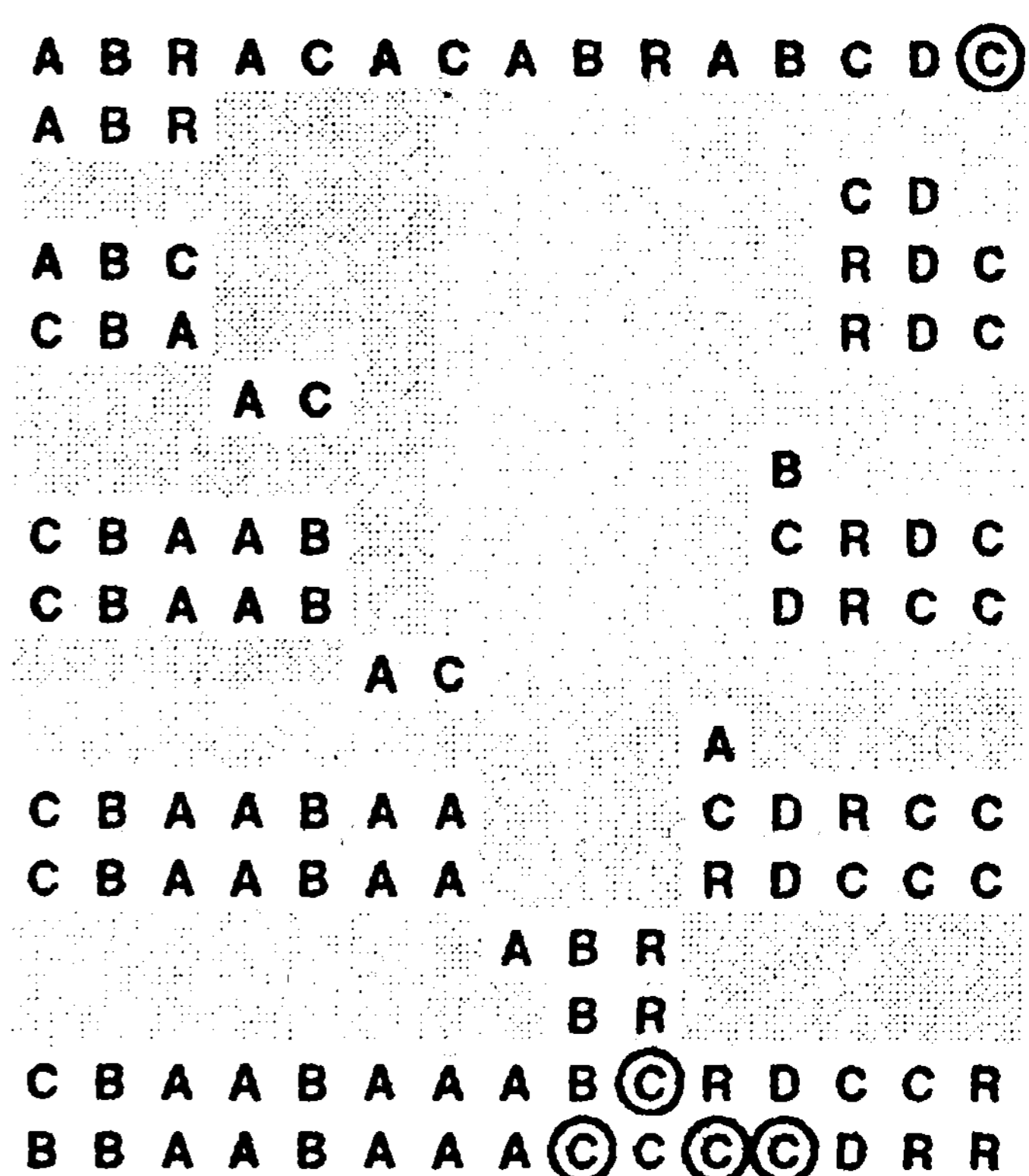


РИСУНОК 7.12. РАЗДЕЛЕНИЕ НА ТРИ ЧАСТИ

Эта диаграмма описывает процесс установки ключей, равных разделяющему элементу, в окончательные позиции. Как и в случае рис. 7.2, просмотр начинается слева с целью обнаружить элемент, который не меньше разделяющего элемента, и справа с целью обнаружить элемент, который не больше разделяющего элемента, затем они меняются местами. Если после обмена элемент слева равен разделяющему элементу, он меняется местами с левым крайним элементом массива; то же самое проделывается и справа. Когда указатели пересекутся, разделяющий элемент помещается в ту же позицию, в которой он находился раньше (предпоследняя строка), а затем все ключи, равные ему, ставятся рядом с ним с любой стороны (нижняя строка).

да указатели просмотра перестают изменяться и выполняется обмен значениями  $i$  и  $j$ , она проверяет каждый из этих элементов на предмет равенства разделяющему элементу. Если элемент, который сейчас находится слева, равен разделяющему элементу, то при помощи операции обмена он помещается в левую часть массива; если элемент, который сейчас находится справа, равен разделяющему элементу, то в результате операции обмена он помещается в правую часть массива.

После того как указатели пересекутся, элементы, равные разделяющему элементу и находящиеся на разных концах массива, после операции обмена попадают в свои окончательные позиции. После этого указанные ключи могут быть исключены из подфайлов, для которых выполняются последующие рекурсивные вызовы.

```
template <class Item>
int operator==(const Item &A, const Item &B)
{ return !less(A, B) && !less(B, A); }
template <class Item>
void quicksort(Item a[], int l, int r)
{ int k; Item v = a[r];
  if (r <= l) return;
  int i = l-1, j = r, p = l-1, q = r;
  for (;;)
  {
    while (a[++i] < v) ;
    while (v < a[--j]) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
    if (a[i] == v) { p++; exch(a[p], a[i]); }
    if (v == a[j]) { q--; exch(a[q], a[j]); }
  }
  exch(a[i], a[r]); j = i-1; i = i+1;
  for (k = l ; k <= p; k++, j--) exch(a[k], a[j]);
  for (k = r-1; k >= q; k--, i++) exch(a[k], a[i]);
  quicksort(a, l, j);
  quicksort(a, i, r);
}
```

## Упражнения

- ▷ 7.34. Дайте объяснения тому, что происходит, когда программа 7.5 выполняется на файле с произвольной организацией ( $i$ ) с двумя различными значениями ключей и ( $ii$ ) с тремя различными значениями ключей.
- 7.35. Изменить программу 7.1 таким образом, чтобы она выполняла команду `return`, если все ключи в подфайле одинаковы. Сравните эффективность полученной программы с эффективностью программы 7.1 на больших файлах с произвольной организацией с ключами, принимающими  $t$  различных значений при  $t = 2, 5$  и  $10$ .
- 7.36. Предположим, в программе 7.2 вместо того, чтобы остановить просмотр с целью нахождения ключей, равных разделяющему элементу, при обнаружении такого ключа он пропускается. Показать, что в таком случае время выполнения программы 7.1 подчиняется квадратичной зависимости.
- 7.37. Доказать, что время выполнения программы из упражнения 7.37 пропорционально квадрату длины файлов для всех файлов с  $O(1)$  различных значений ключей.

7.38. Написать программу, определяющую количество различных ключей, которые встречаются в файле. Воспользуйтесь полученной программой для подсчета числа различных ключей в файле с произвольной организацией, состоящего из  $N$  целых чисел, принимающих значения в диапазоне от 0 до  $M-1$  для  $M = 10, 100$  и  $1000$  и для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

## 7.7 Строки и векторы

Когда сортировочными ключами служат строки, мы можем пользоваться реализацией типов данных, подобной программе 6.11, совместно с реализациями быстрой сортировки, рассматриваемыми в настоящей главе. Несмотря на то что такой подход позволяет получить корректную и эффективную реализацию (обладающую при работе с крупными файлами большим быстродействием, чем удавалось получить до сих пор), имеют место скрытые издержки, которые заслуживают того, чтобы на них остановиться подробнее.

Проблема заключается в стоимости функции `strcmp`, которая сравнивает две строки в направлении слева направо, сопоставляя строки символ за символом, затрачивая на это время, пропорциональное количеству символов, совпадающих в начале обеих строк. На заключительных стадиях процесса разделения быстрой сортировки, когда ключи близки друг к другу по значению, почти вся стоимость алгоритмов сосредоточена в его завершающих стадиях, так что исследование с целью совершенствования алгоритма вполне оправдано.

Например, рассмотрим подфайл размером 5, содержащий ключи `discreet`, `discredit`, `discrete`, `discrepancy` и `discretion`. Все сравнения, выполненные с целью сортировки этих ключей, исследуют по меньшей мере семь символов, но в рассматриваемом случае можно было начать просмотр с седьмого символа, если бы была доступной дополнительная информация, фиксирующая тот факт, что первые шесть символов совпадают.

Процедура разделения файла на три части, которая рассматривалась в разделе 7.6, представляет собой элегантный способ извлечь пользу из отмеченного выше факта. На каждой стадии процесса разделения проверяется только один символ (скажем, символ в позиции  $d$ ), предполагая, что ключи, занимающие позиции от 0 до  $d-1$  и подлежащие сортировке, совпадают. Мы выполняем разделение на три части, помещая те ключи,  $d$ -й символ которых меньше  $d$ -го символа разделяющего элемента, слева, те ключи,  $d$ -й символ которых равен  $d$ -му символу разделяющего элемента, в середине, а те ключи,  $d$ -й символ которых больше  $d$ -го символа разделяющего элемента, справа. Далее мы выполняем обычные действия за исключением того, что мы сортируем средний подфайл, начиная с  $d+1$ -го символа. Нетрудно видеть, что этот метод обеспечивает корректную сортировку и к тому же обладает исключительно высокой эффективностью (см. табл. 7.2). В данном случае мы получаем убедительный пример неограниченных возможностей рекурсивного мышления (и программирования).

Для реализации этого вида сортировки требуется тип данных с более высоким уровнем абстракции, который мог бы обеспечить доступ к отдельным символам ключей. Возможности по манипулированию строками, которыми обладает язык C++, делают подобную реализацию исключительно простой. Тем не менее, мы отложим обсуждение деталей этой реализации до главы 10, в которой рассмотрим различные методы

сортировки, использующие то обстоятельство, что сортировочные ключи легко разлагаются на более мелкие части.

**Таблица 7.2. Эмпирическое исследование вариантов быстрой сортировки**

В этой таблице представлена относительная стоимость нескольких различных вариантов быстрой сортировки на примере упорядочения первых  $N$  слов из книги *Moby Dick*. Непосредственное использование метода вставки для сортировки небольших подфайлов или игнорирование небольших подфайлов с последующей сортировкой того же файла методом вставки потом — суть стратегии, обеспечивающие один и тот же уровень эффективности, но в то же время экономия расходов, достигаемая за счет реализации обеих стратегий несколько ниже, чем для целочисленных ключей (см. табл. 7.1), поскольку стоимость операции сравнения строк влечет большие издержки. Если во время разбиения файлов просмотр не останавливается на дублированных ключах, то время сортировки файла, у которого все ключи одинаковы, подчиняется квадратичной зависимости; низкая эффективность проявляется в этом примере в связи с тем, что существует множество слов, которые встречаются в данных довольно часто. По той же причине разделение на три части обеспечивает высокий уровень эффективности сортировки; она на 30-35 процентов быстрее системной сортировки.

$N$	<b>V</b>	<b>I</b>	<b>M</b>	<b>Q</b>	<b>X</b>	<b>T</b>
12500	8	7	6	10	7	6
25000	16	14	13	20	17	12
50000	37	31	31	45	41	29
100000	91	78	76	103	113	68

Обозначения:

- V** Быстрая сортировка (программа 7.1).
- I** Сортировка вставками для подфайлов небольших размеров.
- M** Файлы небольших размеров игнорируются, завершающая сортировка вставками.
- Q** Системная сортировка `qsort`.
- X** Пропускаются дублированные ключи (квадратичная зависимость в случае, когда все ключи одинаковы).
- T** Разделение на три части (программа 7.5).

Этот подход распространяется на многомерные сортировки, в условиях которых в качестве ключей выступают векторы, а записи должны быть переупорядочены таким образом, что сначала файл упорядочивается по первой компоненте, затем записи с равными первыми компонентами ключей упорядочиваются по второй компоненте и так далее. Если компоненты не имеют дублированных ключей, проблема сводится у сортировке по первой компоненте; однако в обычном случае каждая из компонент может принимать одно из нескольких различных значений, и вполне оправдан переход к разбиению на три части (переход к следующей компоненте в среднем разделе). Этот случай Хоар рассматривал в своей первой статье, он представляет собой важное практическое приложение.

## Упражнения

- 7.39. Рассмотреть возможность усовершенствования сортировки выбором, вставками, пузырьковой сортировки и шейкер-сортировки применительно к строкам.
- 7.40. Сколько символов проверяются в рамках стандартного алгоритма быстрой сортировки (программа 7.1, использующая строковый тип из программы 6.11) при сортировке файла, состоящего из  $N$  строк длиной  $t$ , при этом все строки одинаковы? Дать ответ на тот же вопрос применительно к модификации, предложенной в тексте.

## 7.8. Выборка

Одно из важных приложений, имеющее отношение к сортировке, но не требующее сортировки в полном объеме, является операция нахождения медианы из некоторого множества данных. В статистических, а также в других приложениях обработки данных, это весьма распространенный вид вычислений. Один из способов решения этой задачи заключается в том, что числа подвергаются сортировке, затем выбирается число из середины, но можно поступить еще лучше, воспользовавшись процессом разделения быстрой сортировки.

Операция нахождения медианы представляет собой частный случай операции *выборки* (*selection*): нахождения  $k$ -го наименьшего числа в заданном наборе чисел. Поскольку сам алгоритм не может дать гарантии, что конкретный элемент — суть  $k$ -й наименьший, если не проверит и не распознает  $k - 1$  элементов, которые меньше  $k$ , и  $N - k$  элементов, которые больше  $k$ , большая часть алгоритмов может вернуть все  $k$  наименьших элементов исходного файла без каких-либо дополнительных вычислений.

Операция выборки часто применяется при обработке экспериментальных и других видов данных. Широко практикуется использование медианы и других показателей *порядковой статистики* (*order statistics*) для деления файла на меньшие части. Нередко для дальнейшей обработки запоминается только некоторая часть большого файла; в таких случаях программа, способная выбрать, скажем, 10 процентов наибольших элементов файла, может оказаться предпочтительнее, чем сортировка в полном объеме. Другим важным примером можно считать использование разделения вокруг медианы в качестве первой стадии многих алгоритмов типа "разделяй и властвуй".

Мы уже рассматривали алгоритм, который может быть приспособлен непосредственно для выборки. Если  $k$  исключительно мало, то сортировка выбором (см. главу 6) будет работать хорошо, требуя для своего выполнения время, пропорциональное  $Nk$ : сначала находим первый наименьший элемент, затем второй наименьший, равный наименьшему из оставшихся элементов после первого выбора, и так далее. Для несколько большего  $k$  мы найдем описание методов в главе 9, эти методы можно настроить таким образом, что время их выполнения окажется пропорциональным  $N \log k$ .

Метод выборки, время выполнения которого в среднем линейно для всех значений  $k$ , следует непосредственно из процедуры разделения, используемой в быстрой сортировке. Напомним, что метод разделения, используемый в быстрой сортировке,

переупорядочивает массив  $a[1], \dots, a[r]$  и возвращает целое  $i$  такое, что элементы от  $a[1]$  до  $a[i-1]$  меньше или равны  $a[i]$ , а элементы от  $a[i+1]$  до  $a[r]$  больше или равны  $a[i]$ . Если  $k$  равно  $i$ , то задача решена. С другой стороны, если  $k < i$ , следует продолжать работу на левом подфайле; если  $k > i$ , работу необходимо продолжать на правом подфайле. Подобный подход прямо выводит на рекурсивную программу выборки, каковой является программа 7.6. Пример работы этой процедуры на файле небольших размеров показан на рис. 7.13.

### Программа 7.6. Выборка

Данная процедура производит разделение массива относительно  $(k-1)$ -го наименьшего элемента (элемент в  $a[k]$ ): она переупорядочивает массив таким образом, что  $a[1], \dots, a[k-1]$  меньше или равны  $a[k]$ , а  $a[k+1], \dots, a[r]$  больше или равны  $a[k]$ .

Например, можно вызвать `select(a, 0, N-1, N/2)` с целью разделения массива по значению медианы, оставляя медиану в  $a[N/2]$ .

```
template <class Item>
void select(Item a[], int l, int r,
            int k)
{
    if (r <= l) return;
    int i = partition(a, l, r);
    if (i > k) select(a, l, i-1, k);
    if (i < k) select(a, i+1, r, k);
}
```

### Программа 7.7. Нерекурсивная выборка

Нерекурсивная реализация выборки просто строит раздел, затем помещает левый указатель в раздел, если этот раздел оказывается слева от искомой позиции, или помещает правый указатель в раздел, если раздел оказывается справа от искомой позиции.

```
template <class Item>
void select(Item a[], int l, int r, int k)
{
    while (r > l)
    {
        int i = partition(a, l, r);
        if (i >= k) r = i-1;
        if (i <= k) l = i+1;
    }
}
```

```
A S O R T I N G E X A M P L E
A A E (E) T I N G O X S M P L R
L I N G O P M (R) X T S
L I G (M) O P N
A A E E L I G (M) O P N R X T S
```

### РИСУНОК 7.13. ВЫБОРКА МЕДИАНЫ

*Чтобы найти медиану из ключей, в рассматриваемом примере сортировка, использующая метод разделения, выполняет три рекурсивных вызова. Во время первого вызова отыскивается восьмое наименьшее число в файле из 15 элементов, в то время как разделение позволяет получить четвертое наименьшее (E); во втором вызове ищется четвертое наименьшее число в файле из 11 элементов и разделение дает восьмое наименьшее (R), далее, в третьем вызове ищется четвертое наименьшее число в файле размером 7, которым будет (M). Файл переупорядочен теперь таким образом, что медиана занимает окончательное место, элементы, меньшие медианы по значению, сосредоточены слева от нее, элементы, большие нее по значению — справа от нее (элементы, равные медиане, могут быть помещены с любой ее стороны), однако окончательный порядок в файле еще не установлен.*

Программа 7.7 представляет собой нерекурсивную версию, построенную на базе рекурсивной версии, представленной в программе 7.6. Поскольку эта программа всегда завершается рекурсивным вызовом самой себя, мы просто восстанавливаем начальные значения параметров и возвращаемся в начало программы. Это значит, что мы отказываемся от рекурсии и не используем стек, а также исключаем из программы вычисления, в которых используется  $k$ , и рассматриваем  $k$  только как индекс массива.

**Лемма 7.4.** *Время выполнения выборки, для которой используется быстрая сортировка, в среднем подчиняется линейной зависимости.*

Как и в случае быстрой сортировки, можно предположить (в первом приближении), что в случае файла очень больших размеров каждое разделение разбивает файл напополам, при этом для выполнения процесса разделения требуется выполнить  $N + N/2 + N/4 + N/8 + \dots = 2N$  операций сравнения. Но поскольку разделение выполняется именно для быстрой сортировки, это грубое приближение недалеко от истины. Анализ, подобный приводимому в разделе 7.2 анализу быстрой сортировки, но гораздо более сложный (см. раздел ссылок), дает результаты, согласно которым среднее число сравнений определяется выражением

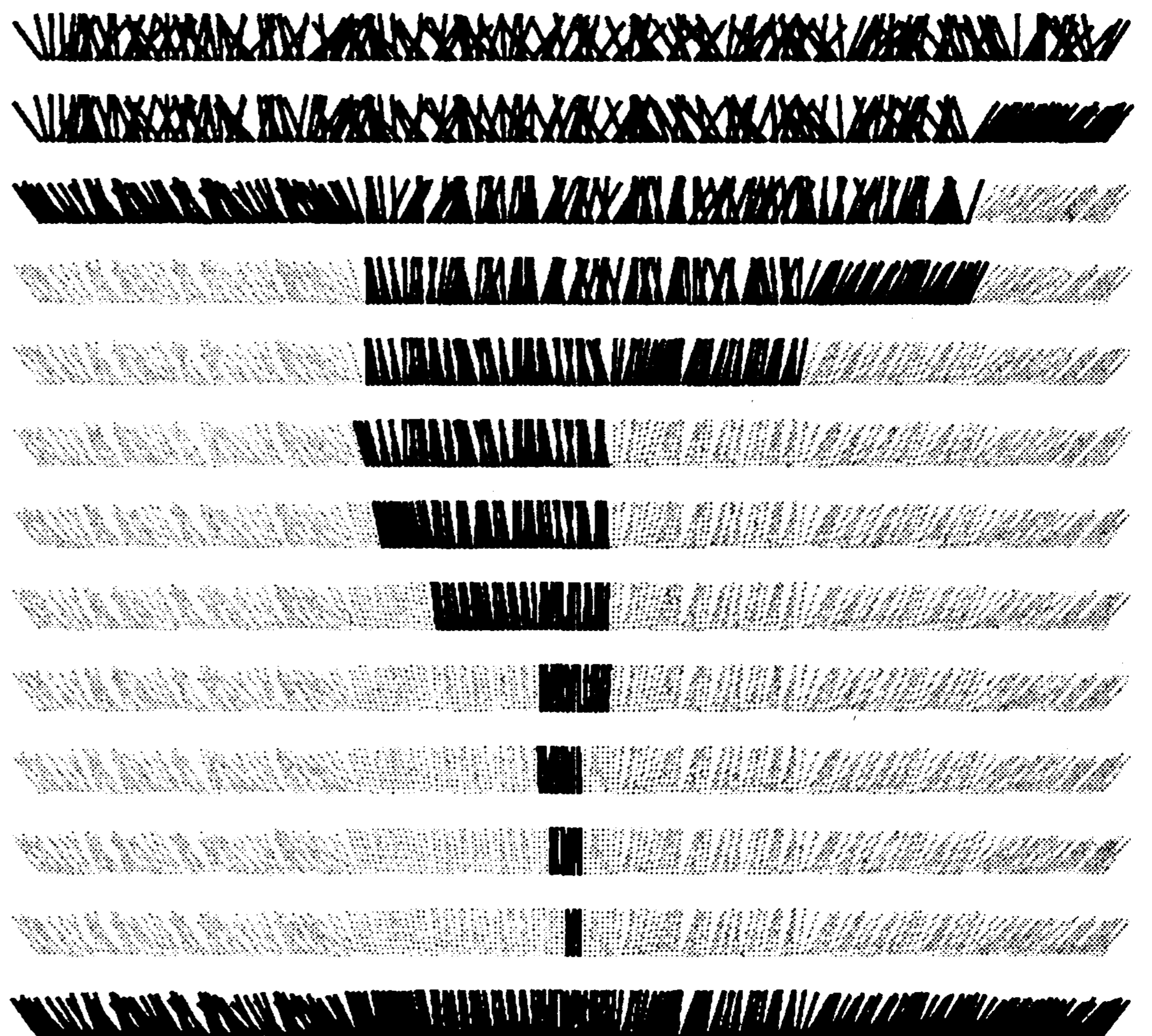
$$2N + 2k \ln(N/k) + 2(N - k) \ln(N/(N - k)),$$

которое линейно зависит от любого допустимого значения  $k$ . Вычисление этой формулы при  $k = N/2$  показывает, что для нахождения медианы необходимо выполнить  $(2 + 2 \ln 2) N$  сравнений.

Пример того, как этот метод находит медиану в крупном файле, представлен на рис. 7.14. В рассматриваемом случае имеется только один подфайл, который при каждом вызове уменьшается в размере в одно и то же число раз, таким образом, эта процедура завершается за  $O(\log N)$  шагов. Выполнение программы можно ускорить, введя в нее выборку, однако при этом следует соблюдать осторожность (см. упражнение 7.45).

#### РИСУНОК 7.14. ВЫБОР МЕДИАНЫ С ПОМОЩЬЮ ПРОЦЕДУРЫ РАЗДЕЛЕНИЯ.

*Процесс выборки предусматривает разбиение на разделы подфайла, который содержит искомый элемент, перемещение левого указателя вправо или правого указателя влево в зависимости от того, где окажется точка раздела.*





Наихудший случай проявляется так же, как и для случая быстрой сортировки — использование данного метода для нахождения наименьшего элемента уже упорядоченного файла приводит к тому, что время выполнения этой процедуры характеризуется квадратичной зависимостью. Можно модифицировать эту процедуру, основанную на применении быстрой сортировки, таким образом, что время ее выполнения будет *гарантировано* подчиняться линейной зависимости. Однако все модификации подобного рода, будучи очень важными в плане теоретических исследований, совершенно неприемлемы на практике.

## Упражнения

- 7.41. Сколько в среднем потребуется операций сравнения, чтобы найти наименьший элемент в файле из  $N$  элементов, применяя для этой цели процедуру **select**?
- 7.42. Сколько в среднем потребуется операций сравнения, чтобы найти  $\alpha N$ -й наименьший элемент, применяя для этой цели процедуру **select**, для  $\alpha = 0.1, 0.2, \dots, 0.9$ ?
- 7.43. Сколько потребуется операций сравнения в наихудшем случае, чтобы найти медиану из  $N$  элементов, применяя для этой цели процедуру **select**?
- 7.44. Напишите эффективную программу переупорядочения файла таким образом, чтобы все элементы с ключами, равными медиане, оказались в окончательной позиции, элементы меньше медианы — слева и элементы больше медианы — справа.
- 7.45. Проведите исследование идеи использования выборки с целью повышения эффективности выбора. *Совет:* Использование медианы не всегда приводит к ожидаемым результатам.
  - 7.46. Реализовать алгоритм выборки на базе метода трехпутевого разделения на примере крупного файла с ключами, принимающими  $t$  различных значений, для  $t = 2, 5$  и  $10$ .

# Слияние и сортировка слиянием

Семейство алгоритмов быстрой сортировки, рассмотренное в главе 7, основано на операции *выборки*: нахождение  $k$ -го минимального элемента в файле. Мы убедились в том, что выполнение операции выборки аналогично делению файла на две части, на часть, содержащую все  $k$  наименьших элементов, и часть, содержащую  $N - k$  больших по значению элементов. В этой главе исследуется семейство алгоритмов сортировки, в основе которых лежит вспомогательный процесс, известный как *слияние*, т.е., объединение двух отсортированных файлов в один файл большего размера. Слияние является основой для простого алгоритма сортировки типа "разделяй и властвуй" (см. раздел 5.2), а также для его двойника — алгоритма восходящей (снизу-вверх) сортировки слиянием, при этом оба из них достаточно просто реализуются.

Выборка и слияние — суть вспомогательные операции в том смысле, что выборка разбивает файл на два независимых файла, в то время как слияние объединяет два независимых файла в один. Контраст между этими двумя операциями становится очевидным, если применить принцип "разделяй и властвуй" для создания конкретных методов сортировки. Можно изменить организацию файла таким образом, что когда обе части файла подвергаются сортировке, упорядочивается и весь файл; и наоборот, можно разбить файл на две части для последующей сортировки, а затем объединить упорядоченные части, чтобы получить весь файл в упорядоченном виде. Мы уже видели, что получается в первом случае: это ни что иное как быстрая сортировка, которая состоит из процедуры выборки, за которой следуют два рекурсивных вызова.

В этой главе рассматривается *сортировка слиянием* (*mergesort*), которая является дополнением быстрой сортировки в том, что она состоит из двух рекурсивных вызовов с последующей процедурой слияния.

Одним из наиболее привлекательных свойств сортировки слиянием является тот факт, что она сортирует файл, состоящий из  $N$  элементов, за время, пропорциональное  $N \log N$ , независимо от характера входных данных. В главе 9 мы познакомимся с еще одним алгоритмом, время выполнения которого гарантировано пропорционально  $N \log N$ ; алгоритм носит название *пирамидальной сортировки* (*heapsort*). Основным недостатком сортировки слиянием заключается в том, что прямолинейные реализации этого алгоритма требуют дополнительного пространства памяти, пропорционального  $N$ . Это препятствие можно обойти, однако сделать это довольно сложно, причем потребуются дополнительные затраты, которые в общем случае не оправдываются на практике, особенно если учесть, что существует альтернатива в виде пирамидальной сортировки. Получить программную реализацию сортировки слиянием не труднее, чем реализацию пирамидальной сортировки, при этом длина внутреннего цикла принимает среднее значение между аналогичными показателями быстрой сортировки и пирамидальной сортировки, следовательно, сортировка методом слияния достойна того, чтобы к ней проявить внимание в случае, когда на первый план выходят такие показатели как быстродействие, необходимость избегать наихудших случаев, возможность использования дополнительного пространства памяти.

Но гарантированное время выполнения, пропорциональное  $N \log N$ , может обернуться недостатком. Например, в главе 6 мы видели, что существуют методы, которые могут быть адаптированы таким образом, что в некоторых особых ситуациях, например, когда уровень упорядоченности файла достаточно высок либо когда имеется всего лишь несколько различных ключей, время их выполнения линейно. В противоположность этому, время выполнения сортировки слиянием зависит главным образом от числа ключей входного файла и оно практически не чувствительно к их порядку.

Сортировка слиянием — это устойчивая сортировка, и данное обстоятельство склоняет чашу весов в ее пользу в тех приложениях, в которых устойчивость имеет важное значение. Конкурирующие методы, такие как быстрая сортировка или пирамидальная сортировка, не относятся к числу устойчивых. Различные приемы, придающие этим методам устойчивость, имеют стойкую тенденцию к использованию дополнительного пространства памяти; следовательно, требования дополнительной памяти, предъявляемые со стороны сортировки слиянием отодвигаются на задний план в тех случаях, когда устойчивость становится доминирующим фактором.

Другое свойство сортировки слиянием, которое приобретает важное значение в некоторых ситуациях, является тот факт, что сортировка слиянием обычно реализуется таким образом, что она осуществляет, в основном, последовательный доступ к данным (один элемент за другим). Например, сортировка слиянием — именно тот метод, который можно применить к связным спискам, для которых из всех методов доступа применим только метод последовательного доступа. По тем же причинам, как мы убедимся в главе 11, слияние часто используется в качестве основы для сортировки на специализированных и высокопроизводительных машинах, поскольку именно последовательный доступ к данным в подобного рода системах обработки данных является самым быстрым.

## 8.1. Двухпутевое слияние

Имея два упорядоченных входных файла, их можно объединить в один упорядоченный выходной файл просто отслеживая наименьший элемент в каждом файле и входя в цикл, в котором меньший из двух элементов, наименьших в своих файлах, переносится в выходной файл; процесс продолжается до тех пор, пока оба входных файла не будут исчерпаны. Мы ознакомимся с несколькими реализациями этой базовой абстрактной операции в этом и следующем разделах. Время выполнения линейно зависит от количества элементов в выходном файле, если на каждую операцию поиска следующего наименьшего элемента в файле уходит одно и то же время, что как раз и имеет место в том случае, когда отсортированные файлы представлены структурой данных, которая поддерживает последовательный доступ с постоянным временем доступа, такой как связный список или массив. Эта процедура представляет собой *двухпутевое слияние* (*two-way merging*); в главе 11 мы подробно ознакомимся с многопутевым слиянием, в котором принимают участие более двух файлов. Наиболее важным приложением многопутевого слияния является внешняя сортировка, которая подробно рассматривается в текущей главе.

Для начала предположим, что имеются два непересекающихся упорядоченных массива целых чисел  $a[0], \dots, a[N-1]$  и  $b[0], \dots, b[M-1]$ , которые требуется слить в третий массив  $c[0], \dots, c[N+M-1]$ . Легко реализуемая очевидная стратегия заключается в том, чтобы последовательно выбирать для  $c$  наименьший оставшийся элемент из  $a$  и  $b$ , как показано в программе 8.1. Эта реализация отличается простотой, в то же время она обладает характеристиками, которые мы сейчас и рассмотрим.

### Программа 8.1. Слияние

Чтобы объединить два упорядоченных массива  $a$  и  $b$  в упорядоченный массив  $c$ , используется цикл `for`, который помещает элемент в массив  $c$  на каждой итерации. Если  $a$  исчерпан, элемент берется из  $b$ ; если исчерпан  $b$ , то элемент берется из  $a$ ; если же элементы остаются и в том и в другом массиве, наименьший из оставшихся элементов в  $a$  и  $b$  переходит в  $c$ . Помимо неявного предположения об упорядоченности обоих массивов, эта реализация предполагает также, что массив  $c$  не пересекается (т.е., не перекрывается или совместно не использует памяти)  $a$  и  $b$ .

```
template <class Item>
void mergeAB(Item c[], Item a[], int N,
             Item b[], int M )
{
    for (int i = 0, j = 0, k = 0; k < N+M; k++)
    {
        if (i == N) { c[k] = b[j++]; continue; }
        if (j == M) { c[k] = a[i++]; continue; }
        c[k] = (a[i] < b[j]) ? a[i++] : b[j++];
    }
}
```

Во-первых, эта реализация предполагает, что массивы не пересекаются. В частности, если  $a$  и  $b$  являются крупными массивами, то для размещения выходных данных необходим третий массив  $c$  (также крупный). Вместо того чтобы использовать дополнительное пространство памяти, пропорциональное размерам слитого файла, жела-

тельно иметь в своем распоряжении такой метод, чтобы с помощью операций обмена мы могли, например, объединить два упорядоченных файла  $a[1], \dots, a[m]$  и  $a[m+1], \dots, a[r]$  в один упорядоченный файл за счет соответствующего перемещения элементов  $a[1], \dots, a[r]$  без использования больших объемов дополнительного пространства памяти. Здесь желательно на какое-то время остановиться и подумать о том, как можно все это сделать. На первый взгляд кажется, что у этой проблемы есть простое решение, однако на самом деле все известные до сих пор решения достаточно сложные, особенно если их сравнить с программой 8.1. И в самом деле, довольно трудно разработать алгоритм обменного слияния, т.е. в рамках пространства памяти, занимаемого входными файлами, который обладал бы лучшими характеристиками и который мог бы стать альтернативой обменной *сортировке*. Мы вернемся к этой проблеме в разделе 8.2.

Слияние, как операция, имеет свою собственную область применения. Например, в обычной среде обработки данных может возникнуть необходимость поддерживать крупный (упорядоченный) файл данных, в который непрерывно поступают новые элементы. Один из подходов заключается в том, что новые элементы группируются в *пакеты*, которые затем добавляются в главный (намного больший) файл, после чего выполняется очередная сортировка всего файла. Такая ситуация как бы специально создана для слияния: более эффективная стратегия предусматривает сортировку пакета (небольших размеров) новых элементов, с последующим слиянием полученного файла небольших размеров с большим главным файлом. Слияние используется во многих подобного рода приложениях, что обуславливает целесообразность ее изучения. Основное внимание в данной главе будет уделяться методам сортировки, в основу которых положено слияние.

## Упражнения

**8.1.** Предположим, что упорядоченный файл размером  $N$  нужно объединить с неупорядоченным файлом размером  $M$ , при этом  $M$  намного меньше  $N$ . Во сколько раз быстрее, чем повторная сортировка, работает предложенный метод, в основе которого лежит слияние, если его рассматривать как функцию от  $M$ , при  $N = 10^3, 10^6$  и  $10^9$ ? Решите эту задачу полагая, что в вашем распоряжении имеется программа сортировки, которой требуется  $c_1 N \lg N$  секунд, чтобы выполнить сортировку файла размером  $N$ , и программа слияния, которой требуется  $c_2(N + M)$  секунд, чтобы слить файл размером  $N$  с файлом размером  $M$ , при  $c_1 \approx c_2$ .

**8.2.** В чем превосходит и в чем уступает сортировка всего файла методом вставок двум методам, представленными в упражнении 8.1? (Ответьте на этот вопрос, полагая, что малый файл имеет произвольную организацию, так что каждая вставка проходит примерно полпути в большом файле, а время выполнения сортировки определяется выражением  $c_3 MN/2$ , при этом константа  $c_3$  — того же порядка, что и другие константы.)

**8.3.** Что произойдет, если воспользоваться программой 8.1 для обменного слияния посредством вызова `merge(a, a, N/2, a+N/2, N-N/2)` применительно к ключам **A E Q S U Y E I N O S T**?

- 8.4. Верно ли утверждение, что программа 8.1, вызванная так, как показано в упражнении 8.3, работает правильно тогда и только тогда, когда оба входных подмассива уже отсортированы? Докажите правильность выводов или представьте контрпример.

## 8.2. Абстрактное обменное слияние

Хотя реализация слияния, по-видимому, требует дополнительного пространства, мы все еще считаем абстракцию обменного слияния полезной при реализации методов сортировки, которые здесь изучаются. В нашей следующей реализации слияния мы сделаем акцент на этом моменте за счет использования интерфейса `merge(a, l, m, r)`, тем самым показывая, что подпрограмма помещает результат слияния  $a[1], \dots, a[m]$  и  $a[m+1], \dots, a[r]$  в объединенный упорядоченный файл  $a[1], \dots, a[r]$ . Можно было бы реализовать эту программу слияния, сначала скопировав абсолютно все во вспомогательный файл с последующим применением базового метода, использованного в программе 8.1, однако пока мы не будем делать этого, а сначала внесем одно усовершенствование в данный подход. И хотя дополнительное *пространство* памяти для вспомогательного массива, по-видимому, на практике связано с определенной ценой, в разделе 8.4 мы рассмотрим дальнейшие улучшения, которые позволят избежать дополнительных затрат *времени*, требуемого на копирование массива.

Вторая, заслуживающая внимания, основная характеристика базового слияния заключается в том, что внутренний цикл содержит две проверки с целью определить, достигнут ли конец хотя бы одного из двух массивов. Разумеется, чаще условие этой проверки не подтверждаются, и складывается исключительно благоприятная ситуация для использования служебных ключей, позволяющих отказаться от такого рода проверок. Иначе говоря, если элементы со значениями ключей, большими, чем значения всех других ключей, добавляются в конец массива **a** и массива **aux**, от этих проверок можно отказаться в силу того, что если массив **a** (**b**) будет исчерпан, служебный ключ позволяет перейти в режим выборки следующих элементов только из массива **b** (**a**) и помещать их в массив **c** вплоть до окончания операции слияния.

Однако, как было показано в главах 6 и 7, служебными метками не всегда просто пользоваться либо в силу того, что не всегда легко находить значение наибольшего элемента, либо в связи с тем, что необходимое пространство памяти не так-то просто получить. Что касается слияния, то существует достаточно простое средство, которое показано на рис. 8.1. В основу этого метода положена следующая идея: при условии, что мы отказались копировать массивы, чтобы реализовать обменную абстракцию, мы просто представляем второй файл во время его копирования в обратном порядке (без дополнительных затрат), так что связанный с ним указатель перемещается справа налево. Эта операция приводит к тому, что наибольший элемент, в каком бы он файле не находился, служит служебной меткой для другого массива. Программа 8.2 содержит эффективную реализацию абстрактного обменного слияния, в основу которого положена упомянутая идея; она служит фундаментом алгоритмов сортировки, которые обсуждаются ниже в этой главе. Она также использует вспомогательный массив, размер которого пропорционален выходному файлу, полученному в резуль-

тате слияния, но она гораздо более эффективна, нежели прямолинейная реализация, поскольку в этом случае отпадает необходимость проверок с целью обнаружения окончания сливаемых массивов.

### Программа 8.2. Абстрактное обменное слияние

Данная программа выполняет слияние двух файлов без использования служебных меток, путем копирования второго массива в массив `aux` в обратном порядке, когда сразу за концом первого массива следует конец второго (т.е., устанавливая на `aux` битонный (*bitonic*) порядок). Первый цикл `for` перемещает первый массив, после чего `i` указывает на `l`, что означает готовность начинать слияние. Вторым циклом `for` перемещает второй массив, после чего `j` указывает на `r`. Затем в процессе слияния (третий цикл `for`) наибольший элемент служит служебной меткой независимо от того, в каком файле он находится. Внутренний цикл этой программы достаточно короткий (перенос в `aux`, сравнение, перенос обратно в `a`, увеличение значений `i` или `j` на единицу, увеличение на единицу и проверка значения `k`).

```
template <class Item>
void merge(Item a[], int l, int m, int r)
{ int i, j;
  static Item aux[maxN];
  for (i = m+1; i > l; i--) aux[i-1] = a[i-1];
  for (j = m; j < r; j++) aux[r+m-j] = a[j+1];
  for (int k = l; k <= r; k++)
    if (aux[j] < aux[i])
      a[k] = aux[j--]; else a[k] = aux[i++];
}
```

Последовательность ключей, которая сначала увеличивается, а затем уменьшается (или сначала уменьшается, а затем увеличивается), называется *битонной* (*bitonic*) последовательностью. Сортировка битонной последовательности эквивалентна слиянию, но иногда удобно представить проблему слияния как проблему битонной сортировки; рассмотренный метод, позволяющий избежать сравнений со служебным элементом, можно рассматривать как простой пример таких сортировок.

Одно из важных свойств программы 8.1 заключается в том, что реализуемое ею слияние устойчиво: она сохраняет относительный порядок элементов с одинаковыми ключами. Эту характеристику нетрудно проверить, и часто имеет смысл убедиться в том, что устойчивость сохраняется при реализации абстрактного обменного слияния, поскольку устойчивое слияние немедленно приводит к устойчивым методам сортировки, как будет показано в разделе 8.3. Не всегда просто сохранить свойство устойчивости: например, программа 8.1 не обеспечивает устойчивости (см. упражнение 8.6). Это обстоятельство еще больше усложняет проблему разработки истинного алгоритма обменного слияния.

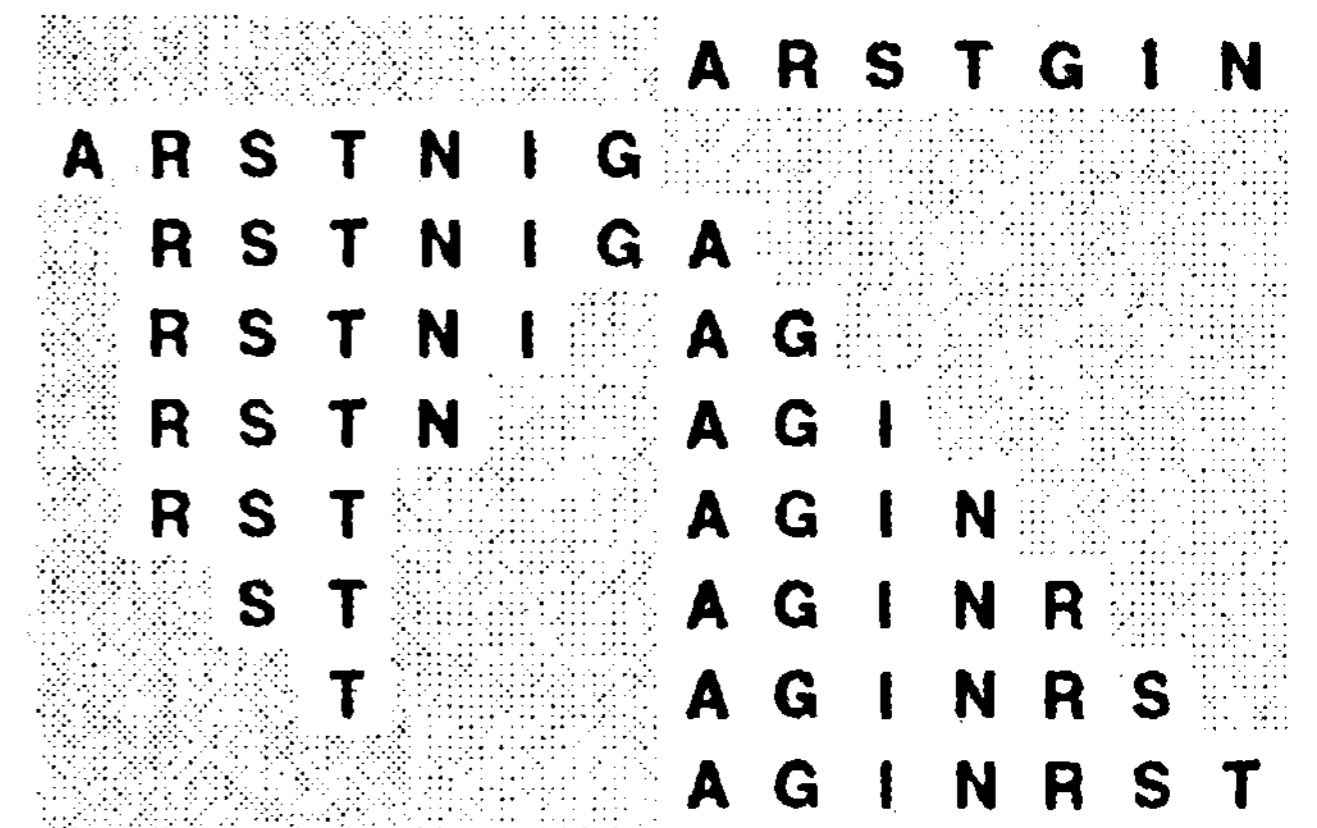


РИСУНОК 8.1. СЛИЯНИЕ БЕЗ ИСПОЛЬЗОВАНИЯ СЛУЖЕБНЫХ МЕТОК.

*Чтобы слить два возрастающих файла, они копируются во вспомогательный массив, при этом второй файл в обратном порядке непосредственно следует за первым. Далее мы следуем простому правилу: перемещаем на выход левый или правый элемент в зависимости от того, какой из них меньше. Наибольший ключ служит служебной меткой для другого файла, независимо от того, в каком файле этот ключ находится. На данной диаграмме показано, как производится слияние файлов *ARST* и *GIN*.*

## Упражнения

- ▷ 8.5. В стиле примера, представленного на диаграмме 8.1, показать, как выполняется слияние ключей **A E Q S U Y E I N O S T** с использованием программы 8.1.
- 8.6. Объяснить, почему программа 8.2 не является устойчивой, и разработать устойчивую версию этой программы.
- 8.7. Что получится, если программу 8.2 применить к ключам **E A S Y Q U E S T I O N**?
- 8.8. Верно ли, что программа 8.2 выполняет правильный вывод тогда и только тогда, когда оба входных подмассива представлены в порядке, установленном сортировкой? Приведите аргументы в пользу своего ответа либо представьте контрпример.

## 8.3. Нисходящая сортировка слиянием

Имея в своем распоряжении процедуру слияния, нетрудно воспользоваться ею в качестве основы для рекурсивной процедуры сортировки. Чтобы отсортировать заданный файл, мы делим его на две части, выполняем рекурсивную сортировку обеих частей, после чего производим их слияние. Реализация этого алгоритма представлена в программе 8.3; пример иллюстрируется на рис. 8.2. Как отмечалось в главе 5, этот алгоритм является одним из широко известных примеров использования принципа "разделяй и властвуй" при разработке эффективных алгоритмов.

Нисходящая сортировка слиянием аналогична принципу управления сверху вниз, в рамках которого руководитель организует работы таким образом, что получив большую задачу, он разбивает ее на подзадачи, которые должны независимо решать его подчиненные. Если каждый руководитель будет решать свою задачу, разбивая ее на две равные части с последующим объединением решений, полученных его подчиненными и последующей передачей результата своему начальству, то примерно также организована сортировка слиянием. Работа недалеко продвинется, пока кто-то, кто не имеет в своем подчинении исполнителей, не получит и не выполнит свою задачу (в рассматриваемом случае это слияние двух файлов размером 1); однако руководство выполняет значительную часть работы, соединяя результаты работы подчиненных в единое целое.

Сортировка слиянием играет важную роль благодаря простоте и оптимальности заложенного в нее ме-

```

A S O R T I N G E X A M P L E
A S
  O R
A O R S
    I T
      G N
        G I N T
A G I N O R S T
          E X
            A M
              A E M X
                L P
                  E L P
                    A E E L M P X
A A E E G I L M N O P R S T X

```

### РИСУНОК 8.2. ПРИМЕР НИСХОДЯЩЕЙ СОРТИРОВКИ СЛИЯНИЕМ

Каждая строка показывает результат вызова функции *mergesort* в процессе нисходящей сортировки слиянием. Сначала выполняется слияние *A* и *S*, чтобы получить *AS*, затем слияние *O* и *R*, чтобы получить *OR*, а затем слияние *OR* и *AS*, чтобы получить *AORS*. Далее осуществляется слияние *IT* и *GN*, чтобы получить *GINT*, затем этот результат сливается с *AORS* и получается *AGINORST* и т.д. При помощи этого метода производится рекурсивное построение отсортированных файлов из отсортированных файлов меньших размеров.



тогда (время ее выполнения пропорционально  $N \log N$ ), который допускает возможность реализации, обладающей устойчивостью. Эти утверждения сравнительно нетрудно доказать.

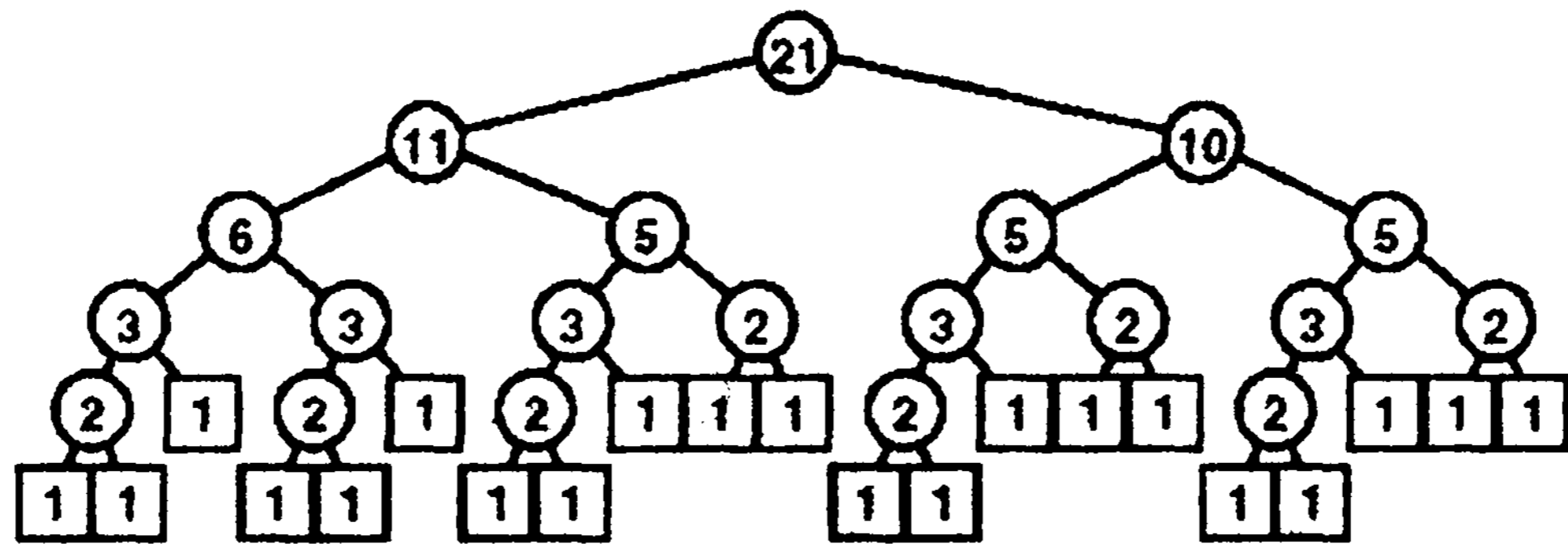
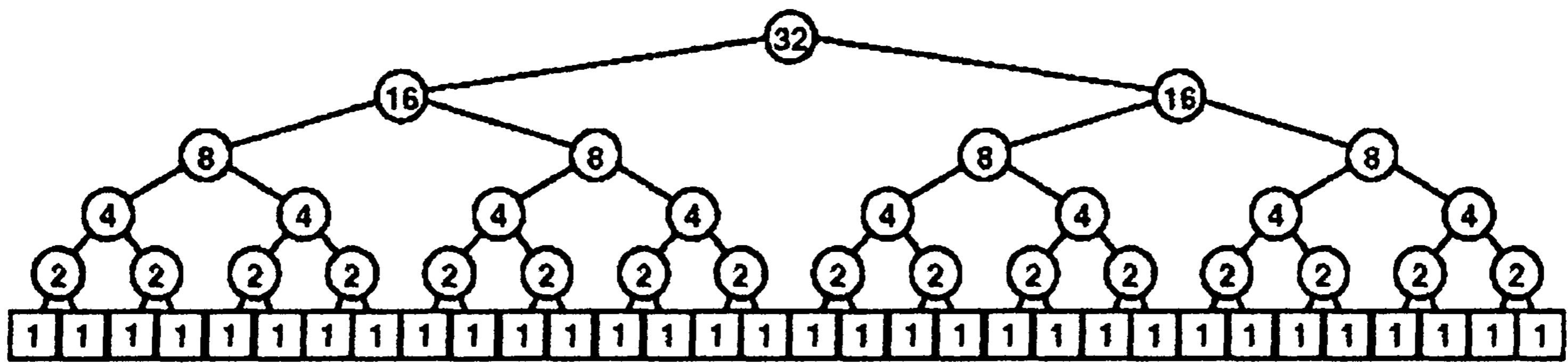
Как было показано в главе 5 (и для быстрой сортировки в главе 7), можно воспользоваться древовидной структурой, чтобы получить наглядное представление о структуре рекурсивных вызовов рекурсивного алгоритма, что поможет понять все варианты рассматриваемого алгоритма и провести его анализ. Что касается сортировки слиянием, то структура рекурсивных вызовов целиком зависит от размеров ввода. Для любого заданного  $N$  мы строим дерево, получившее название "*дерево разделяй и властвуй*", которое описывает размер подфайлов, подвергаемых обработке в процессе выполнения программы 8.3 (см. упражнение 5.73): если  $N$  есть 1, то в таком дереве имеется всего лишь один узел с меткой 1; в противном случае дерево состоит из узла, содержащего файл размером  $N$ , представляющего корень, поддеревя, представляющего левый подфайл размера  $\lfloor N/2 \rfloor$  и поддеревя, представляющего правый подфайл размера  $\lceil N/2 \rceil$ . Следовательно, каждый узел этого дерева соответствует вызову функции `mergesort`, при этом метка дает представление о размере задачи, соответствующей конкретному рекурсивному вызову. Если  $N$  есть степень 2, то такая конструкция приводит к получению полностью сбалансированного дерева со степенью 2 во всех узлах и 1 во всех внешних узлах. Когда  $N$  не является степенью 2, сложность дерева увеличивается. Примеры обоих вышеуказанных случаев иллюстрируются диаграммой на рис. 8.3. Мы сталкивались с подобными рода деревьями раньше, в разделе 5.2, когда изучали алгоритм со структурой рекурсивных вызовов, аналогичной применяемой в сортировке слиянием.

### Программа 8.3. Нисходящая сортировка слиянием

Эта базовая реализация сортировки слиянием является примером рекурсивной программы, прототипом которой служит принцип "разделяй и властвуй". Она выполняет сортировку массива  $a[1], \dots, a[r]$  путем деления его на две части  $a[1], \dots, a[m]$  и  $a[m+1], \dots, a[r]$  с последующей их сортировкой независимо друг от друга (через рекурсивные вызовы) и слияния полученных упорядоченных подфайлов с тем, чтобы в конечном итоге получить отсортированный исходный файл. Функция может потребовать использования вспомогательного файла, достаточно большого, чтобы принять копию входного файла, однако эту абстрактную операцию удобно рассматривать как обменное слияние (см. текст).

```
template <class Item>
void mergesort(Item a[], int l, int r)
{ if (r <= l) return;
  int m = (r+1)/2;
  mergesort(a, l, m);
  mergesort(a, m+1, r);
  merge(a, l, m, r);
}
```

Структурные свойства сбалансированных деревьев, построенных по принципу "разделяй и властвуй", имеют непосредственное отношение к анализу сортировки слиянием. Например, общее количество операций сравнения, выполняемых алгоритмом, в точности равно сумме всех меток узлов.



### РИСУНОК 8.3. ДЕРЕВЬЯ, ПОСТРОЕННЫЕ ПО ПРИНЦИПУ "РАЗДЕЛЯЙ И ВЛАСТВУЙ"

Эти три диаграммы иллюстрируют размеры подзадач, возникающих в процессе выполнения нисходящей сортировки слиянием. В отличие от деревьев, соответствующих, например, быстрой сортировке, эти схемы определяются только размерами исходного файла, а не значениями ключей, присутствующих в файле. Верхняя диаграмма показывает, как сортируется файл, состоящий из 32 элементов. Мы (рекурсивно) сортируем два файла по 16 элементов, затем выполняем их слияние. Файлы сортируются по 16 элементов с выполнением (рекурсивной) сортировки файлов по 8 элементов и т.д. Для файлов, размер которых нельзя представить в виде степени 2, схема оказывается несколько более сложной, в чем нетрудно убедиться из нижней диаграммы.

**Лемма 8.1.** Сортировка слиянием требует выполнения примерно  $N \lg N$  операций сравнения для сортировки любого файла из  $N$  элементов.

В реализациях, описанных в разделах 8.1 и 8.2, каждое слияние типа  $(N/2)$  на  $(N/2)$  требует  $N$  сравнений (это значение будет для разных файлов отличаться на 1 или на 2, в зависимости от того, как используются служебные метки). Следовательно, общее количество сравнений при сортировке в полном объеме может быть описано стандартным сбалансированным рекуррентным соотношением:  $M_N = M_{\lfloor N/2 \rfloor} + M_{\lceil N/2 \rceil} + N$ , где  $M_1 = 0$ . Такое рекуррентное соотношение описывает также сумму меток узлов и длину внешнего пути дерева типа "разделяй и властвуй" с  $N$  узлами (см. упражнение 5.73). Это утверждение нетрудно проверить, когда  $N$  является степенью числа 2 (см. формулу 2.4) и доказать методом индукции для произвольного  $N$ . Упражнения 8.12—8.14 содержат непосредственное доказательство.

**Лемма 8.2.** Сортировка слиянием использует дополнительное пространство, пропорциональное  $N$ .

Это факт непосредственно следует из обсуждения, приведенного в разделе 8.2. Мы можем предпринять некоторые шаги, дабы уменьшить размеры используемого дополнительного пространства за счет существенного усложнения алгоритма (см., например, упражнение 8.21). Как будет показано в разделе 8.7, сортировка слиянием также эффективна, если сортируемый файл организован как связный список. В этом случае указанное свойство сохраняется, однако для связей расходуется дополнительное пространство памяти. В случае массивов, как отмечалось в разделе 8.2, можно выполнять обменное слияние (обсуждение этой темы будет продолжено в разделе 8.4), однако эта стратегия вряд ли оправдывается на практике.

**Лемма 8.3.** *Сортировка слиянием устойчива, если устойчив используемый при этом метод слияния.*

Это утверждение легко проверить методом индукции. Для реализации метода слияния, предложенного в программе 8.1, легко показать, что относительная позиция дублированных ключей не нарушается. Однако, чем сложнее алгоритм, тем выше вероятность того, что эта устойчивость будет нарушена (см. упражнение 8.6).

**Лемма 8.4.** *Потребность ресурсов со стороны сортировки слиянием не чувствительна по отношению к исходному порядку входного файла.*

В наших реализациях входные данные определяют разве что порядок, в котором элементы обрабатываются во время слияний. Каждый проход требует пространства памяти и числа шагов, пропорциональных размеру подфайла, что обуславливается необходимостью затрат на перемещение данных во вспомогательный файл. Соответствующие две ветви оператора `if` могут потребовать слегка отличающихся значений времени для выполнения компиляции, что в свою очередь приводит к некоторой зависимости времени выполнения от характера входных данных, однако число сравнений и других операций не зависит от того, как упорядочен входной файл. Обратите внимание на то, что это отнюдь *не* эквивалентно утверждению, что алгоритм не адаптивный (см. раздел 6.1) — последовательность сравнений не зависит от упорядоченности входных данных.

## Упражнения

- ▷ **8.9.** Показать, что слияние, реализуемое программой 8.3, не сортирует ключи **EASYQUESTION**.
- 8.10.** Начертить деревья типа "разделяй и властвуй" для  $N = 16, 24, 31, 32, 33$  и  $39$ .
- **8.11.** Реализовать рекурсивную сортировку слиянием для массивов, воспользовавшись идеей трехпутевой, а не двухпутевой сортировки.
- **8.12.** Доказать, что все узлы с метками 1 в деревьях типа "разделяй и властвуй", расположены на двух нижних уровнях.
- **8.13.** Доказать, что метки в узлах на каждом уровне сбалансированного дерева размером  $N$ , в сумме дают  $N$ , за исключением, возможно, нижнего уровня.
- **8.14.** Используя упражнения 8.12 и 8.13, доказать, что количество сравнений, необходимых для выполнения сортировки слиянием, находятся в пределах между  $N \lg N$  и  $N \lg N + N$ .
- **8.15.** Найти и доказать зависимость между числом сравнений, используемых сортировкой слиянием, и количеством бит в  $\lceil \lg N \rceil$ -разрядных положительных числах, меньших  $N$ .

## 8.4. Усовершенствования базового алгоритма

Как уже было видно на примере быстрой сортировки, можно усовершенствовать большую часть рекурсивных алгоритмов, применяя для обработки файлов небольших размеров другие методы, отличные от основного. Рекурсия гарантирует, что эти методы будут использоваться для случаев небольших файлов, так что более совершенная обработка файлов небольших размеров приводит к тому, что улучшается и весь алгоритм. Следовательно, как это имело место и для случая быстрой сортировки, переключение на сортировку вставками подфайлов небольших размеров приводит к уменьшению времени выполнения типовой реализации операции сортировки слиянием от 10 до 15 процентов.

В качестве следующего усовершенствования целесообразно рассмотреть возможность сведения к нулю времени копирования данных во вспомогательный массив, используемый процедурой слияния. Поступая таким образом, следует так организовать рекурсивные вызовы, что процесс вычисления сам меняет в нужный момент роли входного и вспомогательного массивов на каждом уровне. Один из способов реализации такого подхода заключается в создании двух вариантов программ — одного для приема входных данных в файл **a** и пересылки выходных данных в файл **aux**, а другого для приема входных данных в файл **aux** и пересылки выходных данных в файл **a**, после чего обе версии поочередно вызывают одна другую. Другой подход продемонстрирован в программе 8.4, которая вначале создает копию входного массива, а затем использует программу 8.1 и переключает аргументы в рекурсивных вызовах с целью отказа от явно заданной процедуры копирования массива. Вместо нее путем поочередных переключений результат слияний помещается то во вспомогательный, то во входной файл. (Это достаточно хитроумная программа.)

Данный метод позволяет избежать копирования массива ценой включения во внутренний цикл проверки с целью определения, когда входные файлы будут исчерпаны. (Напоминаем, что предложенный метод устранения подобного рода проверок в программе 8.2 предусматривает превращение этого файла в битонный на время копирования.) Эту потерю можно восполнить посредством реализации той же идеи: мы пишем программы как слияния, так и сортировки слиянием, одну для представления массива в порядке возрастания, а другую — для представления массива в порядке убывания. Вооружившись такой стратегией, можно снова обратиться к битонной стратегии и устроить так, что внутреннему циклу слияния никогда не понадобятся служебные метки.

Принимая во внимание тот факт, что такая супероптимизация использует четыре копии базовых программ и умопомрачительные рекурсивные переключения аргументов, она может быть рекомендована экспертам (или студентам!), но в то же время она существенно ускоряет сортировку слиянием. Экспериментальные результаты, которые будут обсуждаться в разделе 8.6, показывают, что сочетание всех предложенных выше усовершенствований ускоряют сортировку слиянием примерно на 40 процентов, однако сортировка слиянием все еще выполняется примерно на 25 процентов медленнее, чем быстрая сортировка. Эти показатели зависят от реализации и от машины, однако подобные результаты возможны в различных ситуациях.

Другие реализации слияния, требующие выполнения явно заданных проверок на предмет исчерпания первого файла, могут привести к более заметным колебаниям значений времени выполнения в зависимости от характера входных данных, но эта зависимость все еще остается незначительной. В файлах с произвольной организацией размер другого подфайла, когда исчерпывается первый файл, будет небольшим, и стоимость перемещения во вспомогательный файл все еще остается пропорциональной размеру этого подфайла. Можно подумать о повышении производительности сортировки слиянием в тех случаях, когда в файле уже достигнута высокая степень упорядоченности, за счет игнорирования вызова функции `merge`, когда в файле уже установлен порядок, однако данная стратегия не эффективна на многих типах файлов.

#### Программа 8.4. Сортировка слиянием без копирования

Рекурсивная программа предусматривает сортировку файла **b**, в результат сортировки помещается в файл **a**. Следовательно, рекурсивные вызовы сформулированы таким образом, что их результаты остаются в файле **b**, и мы применяем программу 8.1 для слияния файлов, помещенных в **b** с файлом из **a**, а их результаты остаются в файле **a**. Таким образом, все перемещения данных выполняются в процессе слияния.

```
template <class Item>
void mergesortABr(Item a[], Item b[], int l, int r)
    { if (r-l <= 10) { insertion(a, l, r); return; }
      int m = (l+r)/2;
      mergesortABr(b, a, l, m);
      mergesortABr(b, a, m+1, r);
      mergeAB(a+1, b+1, m-l+1, b+m+1, r-m);
    }
template <class Item>
void mergesortAB(Item a[], int l, int r)
    { static Item aux[maxN];
      for (int i = l; i <= r; i++) aux[i] = a[i];
      mergesortABr(a, aux, l, r);
    }
```

### Упражнения

**8.16.** Реализовать абстрактное обменное слияние, использующее дополнительное пространство памяти, размер которого пропорционален размеру меньшего из файлов, подвергаемых слиянию. (Ваш метод должен наполовину уменьшать потребность в пространстве сортировки слиянием.)

**8.17.** Выполните сортировку слиянием крупного файла с произвольной организацией и эмпирическим путем определите длину другого подфайла на момент исчерпания первого подфайла как функцию от  $N$  (сумма длин двух сливаемых подфайлов).

**8.18.** Предположим, что программа 8.3 модифицирована таким образом, что пропускает функцию `merge`, когда  $a[m] < a[m+1]$ . Сколько сравнений экономится в этом случае, если в файле, представленном к сортировке, уже установлен порядок, предусматриваемый данной сортировкой?

**8.19.** Выполните модифицированный алгоритм, предложенный в упражнении 8.18, для крупных файлов с произвольной организацией. Определите эмпирическим путем среднее число пропусков функции `merge` в зависимости от  $N$  (размер исходного файла, представленного к сортировке).

**8.20.** Предположим, что сортировка слиянием должна быть выполнена на  $h$ -сортированном файле для небольшого значения  $h$ . Какие изменения вы внесете в подпрограмму `merge`, чтобы воспользоваться преимуществами, предоставляемыми упомянутым свойством входных данных? Выполните эксперименты с гибридами сортировки методом Шелла и сортировки слиянием, основанными на этой подпрограмме.

**8.21.** Разработайте реализацию слияния, которое уменьшает потребность в дополнительном пространстве памяти до  $\max(M, N/M)$ , воспользовавшись следующей идеей. Разбейте массив на  $N/M$  блоков размером  $M$  (для простоты предположим, что  $N$  кратно  $M$ ). Затем, (i) рассматривая эти блоки как записи, первые ключи которых суть сортировочные ключи, отсортировать их, используя для этой цели сортировку выбором, и (ii) выполнить проход по массиву, выполняя слияние первого блока со вторым, затем второго блока с третьим и так далее.

**8.22.** Доказать, что метод, описанный в упражнении 8.21, выполняется за время, подчиняющееся линейной зависимости.

**8.23.** Реализовать битонную сортировку слиянием без копирования.

## 8.5. Восходящая сортировка слиянием

Как отмечалось в главе 5, у каждой рекурсивной программы имеется нерекурсивный аналог, который хотя и выполняет эквивалентные вычисления, тем не менее, он делает это по-другому. Будучи прототипами теории разработки алгоритмов по принципу "разделяй и властвуй", нерекурсивные реализации сортировки слиянием заслуживают детального изучения.

Рассмотрим последовательность слияний, выполняемую рекурсивным алгоритмом. В примере, представленном на рис. 8.2, видно, что файл размером 15 сортируется в виде следующей последовательности слияний:

```
1-с-1 1-с-1 2-с-2 1-с-1 1-с-1 2-с-2 4-с-4
1-с-1 1-с-1 2-с-2 1-с-1 2-с-1 4-с-3 8-с-7.
```

Порядок выполнения слияний определяется рекурсивной структурой алгоритма. Однако подфайлы обрабатываются независимо и слияния могут выполняться в различных последовательностях. Рисунок 8.4 показывает восходящую стратегию, при которой последовательность слияний такова:

```
1-с-1 1-с-1 1-с-1 1-с-1 1-с-1 1-с-1 1-с-1
2-с-2 2-с-2 2-с-2 2-с-1 4-с-4 4-с-3 8-с-7.
```

Последовательность слияний, выполняемая рекурсивным алгоритмом, определяется деревом типа "разделяй и властвуй", показанным на рис. 8.3: мы просто проходим по дереву в обратном порядке. Как было показано в главе 3, можно разработать нерекурсивный алгоритм, использующий явно определяемый стек, который даст ту же последовательность слияний. Однако нет необходимости ограничиваться только

обратным порядком: любое прохождение дерева, во время которого обход поддеревя, принадлежащего конкретному узлу, должен быть завершен перед посещением самого узла, дает правильный алгоритм. Единственное ограничение заключается в том, что сливаемые файлы должны быть сначала отсортированы. Что касается сортировки слиянием, то удобно сначала выполнить все слияния типа 1-с-1, затем все слияния типа 2-с-2, затем типа 4-с-4 и так далее. Такая последовательность соответствует обходу дерева по уровням, постепенно поднимаясь по дереву снизу вверх.

### Программа 8.5. Восходящая сортировка слиянием

Восходящая сортировка слиянием состоит из последовательности проходов по всему файлу с выполнением слияний вида  $m$ -с- $m$ , при этом  $m$  на каждом проходе удваивается, так что в заключение производится слияние типа  $m$ -с- $x$  для некоторого  $x$ , меньшего или равного  $m$ .

```
inline int min(int A, int B)
{ return (A < B) ? A : B; }
template <class Item>
void mergesortBU(Item a[], int l, int r)
{
    for (int m = 1; m <= r-1; m = m+m)
        for (int i = 1; i <= r-m; i += m+m)
            merge(a, i, i+m-1, min(i+m+m-1, r));
}
```

В главе 5 на нескольких примерах можно было заметить, что когда мы рассуждаем в направлении снизу вверх, имеет смысл переориентировать мышление в направлении стратегии "объединяй и властвуй", в рамках которой принимаются решения относительно малых подзадач, которые объединяются для получения решения более крупной задачи. В частности, нерекурсивный вариант сортировки слиянием в программе 8.5 получается следующим образом: все элементы файла рассматриваются как упорядоченные подписки длиной 1. Затем мы просматриваем этот список, выполняя слияния вида 1-с-1, что дает упорядоченные подписки размером 2, затем мы просматриваем полученный список, выполняя при этом слияния вида 2-с-2, что даст упорядоченный подписаниек размером 8, и так далее до тех пор, пока весь список не станет упорядоченным. Завершающий подписаниек не всегда может оказаться того же размера, что и все другие, если размер файла не является степенью 2, тем не менее, можно слить и его.

Если размер файла является степенью 2, то множество слияний, выполняемых восходящей сортировкой слиянием, в точности совпадает со слияниями, выполняемыми рекурсивной сортировкой слиянием, однако последовательность слияний будет другой.

```
A S O R T I N G E X A M P L E
A S
  O R
    I T
      G N
        E X
          A M
            L P
A O R S
  G I N T
    A E M X
      E L P
A G I N O R S T
  A E E L M P X
A A E E G I L M N O P R S T X
```

### РИСУНОК 8.4. ПРИМЕР ВОСХОДЯЩЕЙ СОРТИРОВКИ СЛИЯНИЕМ

Каждая строка диаграммы показывает результат вызова функции в процессе восходящей сортировки слиянием.

Слияния вида 1-с-1 выполняются первыми: слияние *A* и *S* дает *AS*; затем производится слияние *O* и *R*, в результате получаем *OR* и так далее.

Поскольку длина файла является нечетной величиной, последнее *E* в слиянии не участвует. На втором проходе выполняются слияния типа 2-с-2: *AS* сливается с *OR*, в результате получает *AORS* и т.д. Сортировка файла завершается слиянием вида 4-с-4, 4-с-3 и, наконец, 8-с-7.

Восходящая сортировка слиянием соответствует прохождению дерева типа "разделяй и властвуй" в порядке уровней, снизу вверх. В противоположность этому, мы обращались к рекурсивному алгоритму как к *нисходящей сортировке слиянием*, поскольку при обратном порядке прохождения дерева просмотр начинается сверху и следует вниз по дереву.

Если размер файла не может быть представлен степенью 2, восходящий алгоритм дает другое множество слияний, как показано на рис. 8.5. Восходящий алгоритм соответствует дереву, построенному по принципу "объединяй и властвуй" (см. упражнение 5.75), который отличается от дерева типа "разделяй и властвуй", относящегося к категории нисходящих алгоритмов. Однако вполне можно устроить так, чтобы последовательность слияний, порожденных рекурсивным методом, была такой же, как и аналогичная последовательность, полученная в рамках нерекурсивного метода, тем не менее, нет особых причин делать это, поскольку разница в затратах на их реализацию по отношению к общим затратам незначительная.

Леммы 8.1—8.4 справедливы и для восходящей сортировки слиянием, при этом имеют место следующие дополнительные леммы:

**Лемма 8.5.** *Все слияния на каждом проходе восходящей сортировки слиянием манипулируют файлами, размер которых выражен степенью 2, за исключением разве что размера последнего файла.*

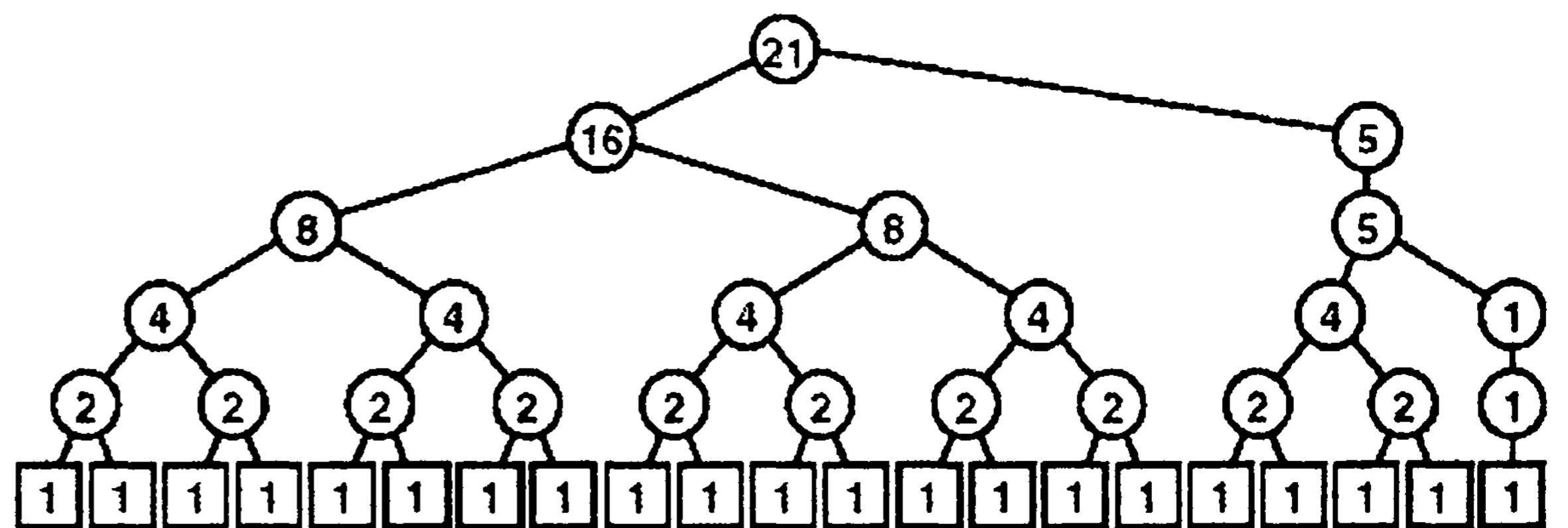
Это факт легко установить методом индукции.

**Лемма 8.6.** *Количество проходов при восходящей сортировке слиянием по файлу из  $N$  элементов в точности равно числу бит в двоичном представлении  $N$  (при этом ведущие нули игнорируются).*

На каждом проходе восходящей сортировки слиянием размер упорядоченных подфайлов удваивается, так что размер подписков после  $k$  проходов составит  $2^k$ . Таким образом, количество проходов, необходимое для сортировки файла из  $N$  элементов, есть наименьшее  $k$  такое, что  $2^k \geq N$ , точной величиной  $k$  является  $\lceil \lg N \rceil$ , т.е. количество бит в двоичном представлении  $N$ . Это можно доказать методом индукции или путем анализа структурных свойств деревьев типа "объединяй и властвуй".

### РИСУНОК 8.5. РАЗМЕРЫ ФАЙЛОВ ПРИ ВОСХОДЯЩЕЙ СОРТИРОВКЕ СЛИЯНИЕМ

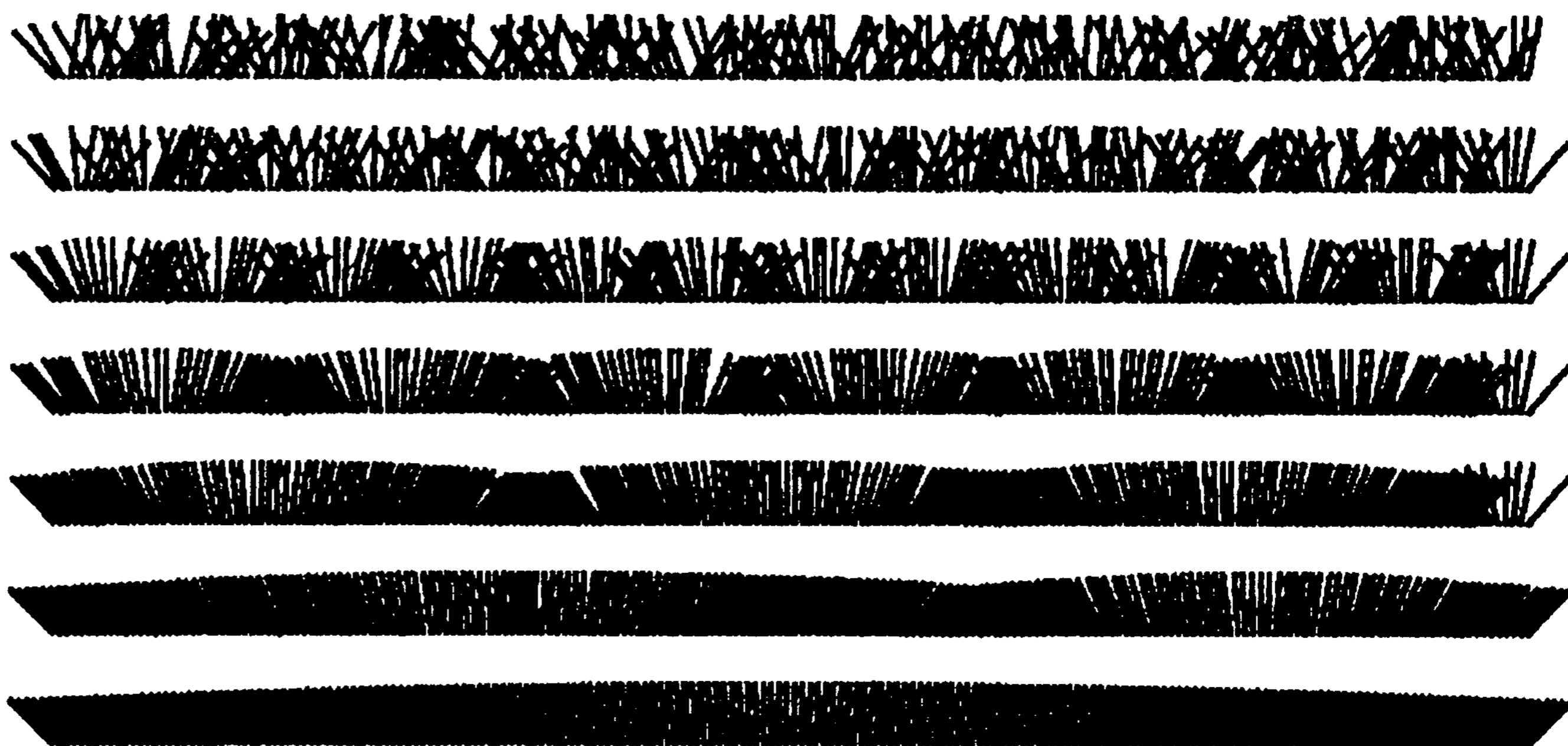
*Схемы восходящей сортировки слиянием кардинально отличаются от схем, применяемых при нисходящей сортировке слиянием (рис. 8.3), когда размер файла не является степенью 2. Что касается восходящей сортировки слиянием, то все размеры подфайлов, исключая, возможно, последний, являются степенью 2. Эти различия представляют интерес для понимания базовых структур алгоритмов, однако на производительности сортировки они отражаются лишь незначительно.*





**РИСУНОК 8.6. ВОСХОДЯЩАЯ СОРТИРОВКА СЛИЯНИЕМ**

*Нам потребуется всего лишь семь проходов, чтобы отсортировать 200 элементов, применяя для этой цели восходящую сортировку слиянием. На каждом проходе количество отсортированных подфайлов уменьшается вдвое, зато их длина удваивается (за исключением разве что последнего подфайла).*



Процесс выполнения восходящей сортировки слиянием показан на рис. 8.6. Сортировка 1 миллиона элементов выполняется за 20 проходов по данным, 1 миллиарда — за 30 проходов и т.д.

Кратко подводя итоги, отметим, что нисходящие и восходящие сортировки суть два достаточно простых алгоритма, в основу которых положена операция слияния двух упорядоченных подфайлов в результирующий объединенный упорядоченный файл. Оба алгоритма тесно связаны между собой и даже выполняют одно и то же множество слияний, если размер исходного файла является степенью 2, но они отнюдь не идентичны. Рисунок 8.7 демонстрирует различия динамических характеристик алгоритмов на примере большого файла. Каждый алгоритм может использоваться на практике, если речь не идет об экономии пространства памяти и желательно обеспечить гарантированное время выполнения для наихудшего случая. Оба алгоритма представляют интерес как прототипы более универсальных алгоритмов типа "разделяй и властвуй" и "объединяй и властвуй".

**Упражнения**

**8.24.** Показать, какие слияния выполняет нисходящая сортировка слиянием (программа 8.5) на ключах `EASYQUESTION`.

**8.25.** Реализовать восходящую сортировку слиянием, которая начинает с того, что сортирует блоки по  $M$  элементов каждый методом вставок. Определить эмпирическим путем значение  $M$ , для которого разработанная программа выполняется быстрее всего при сортировке файлов с произвольной организацией, содержащих  $N$  элементов при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

**8.26.** Нарисовать деревья, которые отображают слияния, выполняемые программой 8.5 для  $N = 16, 24, 31, 32, 33$  и  $39$ .

**8.27.** Написать программу рекурсивной сортировки слиянием, которая выполняет те же слияния, что и восходящая сортировка слиянием.

**8.28.** Написать программу восходящей сортировки слиянием, которая выполняет те же слияния, что и нисходящая сортировка слиянием. (Это упражнение намного труднее, чем упражнение 8.27).

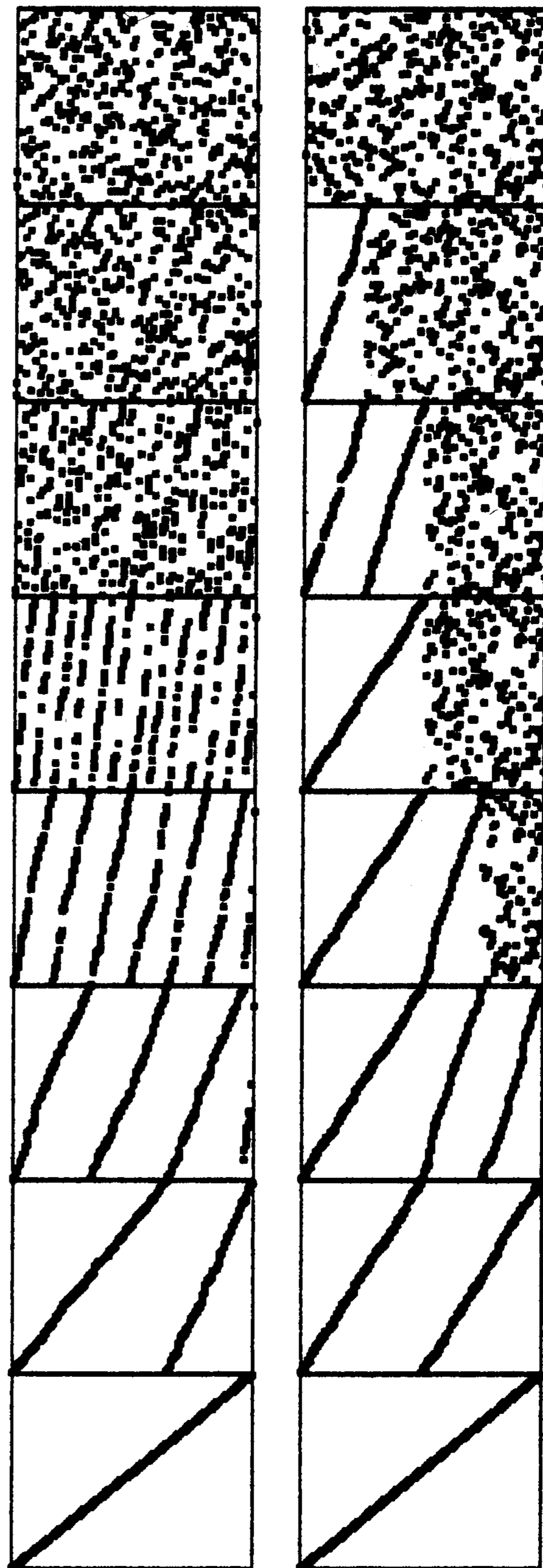
8.29. Предположим, что размер файла является степенью 2. Удалите рекурсию из нисходящей сортировки слиянием, чтобы получить нерекурсивную сортировку слиянием, которая выполняет ту же *последовательность* слияний.

8.30. Доказать, что количество проходов, выполняемых нисходящей сортировкой слиянием, *также* представляется числом бит в двоичном представлении (см. лемму 8.6).

## 8.6. Производительность сортировки слиянием

Таблица 8.1 характеризует относительную эффективность различных усовершенствований из числа рассмотренных выше. Как это часто бывает, такие исследования показывают, что время выполнения сортировки можно сократить наполовину и даже больше, если направить усилия на улучшения внутреннего цикла алгоритма сортировки.

Помимо погони за усовершенствованиями, рассмотренными в разделе 8.2, можно добиться дальнейших улучшений производительности за счет того, что наименьшие элементы в обоих массивах будут содержаться в простых переменных или в регистрах процессора во избежание ненужных доступов к массивам. Таким образом, внутренний цикл сортировки слиянием может быть, в основном, сведен к операциям сравнения (с условным переходом), к увеличению на единицу значений двух счетчиков ( $k$  и одного из  $i$  или  $j$ ) и проверке условия завершения цикла с условным переходом. Общее количество команд во внутреннем цикле несколько превышает этот показатель для быстрой сортировки, однако такие команды выполняются всего лишь  $N \lg N$  раз, в то время как команды внутреннего цикла выполняются в рамках быстрой сортировки на 39 процентов чаще (или на 29 процентов в случае ее варианта с вычислением медианы из трех элементов). Чтобы выполнить точное сравнение этих двух алгоритмов в конкретной среде, следует воспользоваться более совершенной реализацией и провести более подробный анализ. Тем не менее, точно известно, что внутренний цикл сортировки слиянием несколько длиннее внутреннего цикла быстрой сортировки.



**РИСУНОК 8.7. СРАВНЕНИЕ ВОСХОДЯЩЕЙ СОРТИРОВКИ СЛИЯНИЕМ С НИСХОДЯЩЕЙ СОРТИРОВКОЙ СЛИЯНИЕМ**

*Восходящая сортировка слиянием (слева) состоит из последовательности проходов по файлу, которые выполняют слияние отсортированных подфайлов до тех пор, пока не останется только один подфайл. Каждый элемент файла, за исключением разве что нескольких в самом конце, используется в каждом проходе. В противоположность этому, нисходящая сортировка слиянием (справа) сортирует первую половину файла, прежде чем перейти ко второй половине (в рекурсивном режиме), так что схемы выполнения обеих видов сортировки слиянием существенно различаются.*

Таблица 8.1. Эмпирические исследования алгоритмов сортировки слиянием

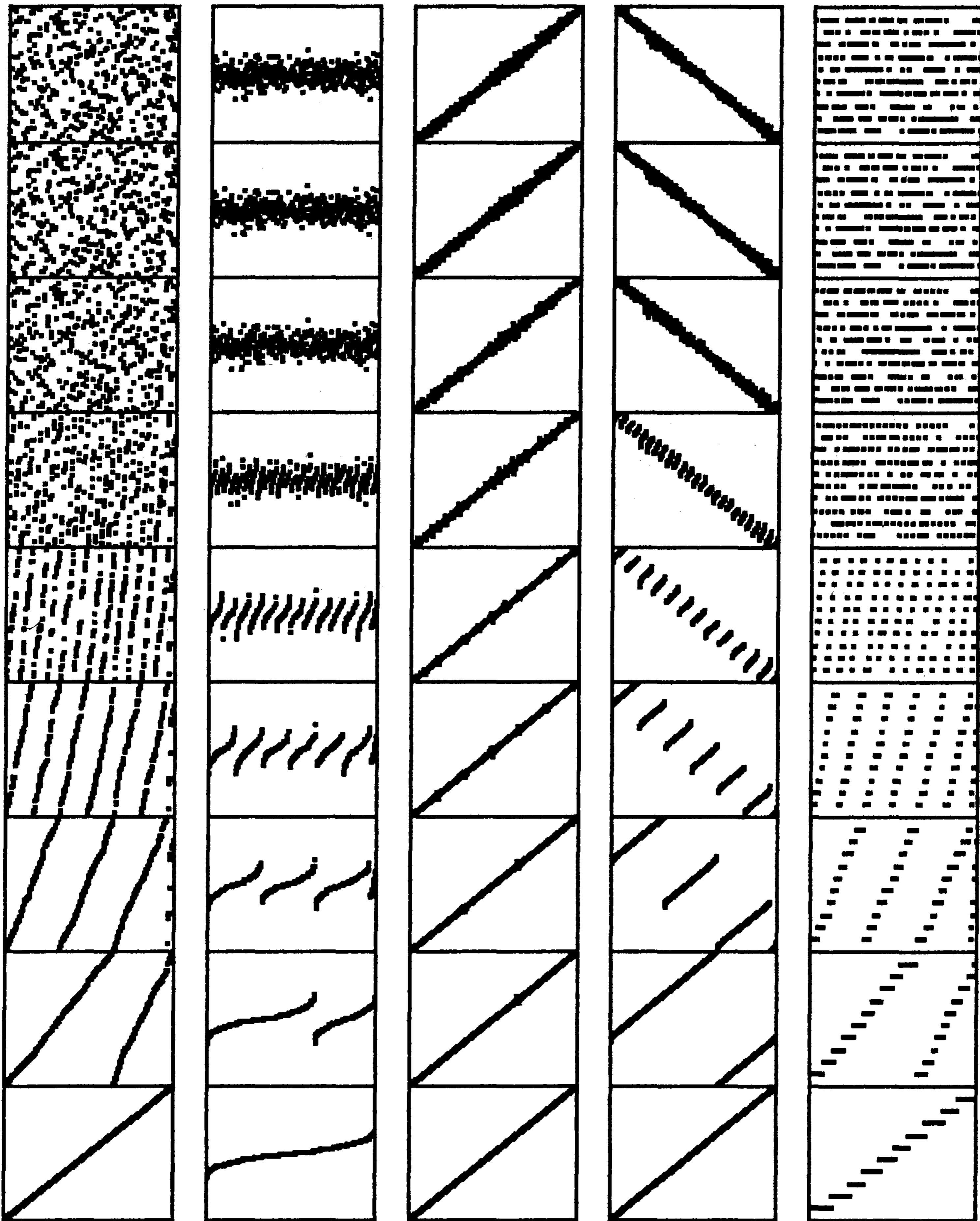
Представленные здесь относительные временные показатели различных видов сортировки на файлах чисел с плавающей точкой с произвольной организацией для различного числа  $N$  отражают следующие факты: стандартная быстрая сортировка выполняется в два раза быстрее стандартной сортировки слиянием; добавление отсеечения файлов небольших размеров снижает время выполнения нисходящей и восходящей сортировок слиянием примерно на 15 процентов; для заданных в таблице размеров файлов быстрое действие нисходящей сортировки слиянием примерно на 10 процентов выше, чем восходящей; даже если устранить затраты на копирование файла, то и в этом случае сортировка слиянием файлов с произвольной организацией на 50–60 процентов медленнее простой быстрой сортировки (см. табл. 7.1).

$N$	$Q$	сверху вниз			снизу вверх	
		$T$	$T^*$	$O$	$B$	$B^*$
12500	2	5	4	4	5	4
25000	5	12	8	8	11	9
50000	11	23	20	17	26	23
100000	24	53	43	37	59	53
200000	52	111	92	78	127	110
400000	109	237	198	168	267	232
800000	241	524	426	358	568	496

Ключи:

- Q** Стандартная быстрая сортировка (программа 7.1)
- T** Сортировка нисходящая слиянием, стандартная (программа 8.1)
- T\*** Сортировка нисходящая слиянием с отсечением файлов небольших размеров
- O** Сортировка нисходящая слиянием с отсечением и без копирования массива
- B** Стандартная сортировка восходящая слиянием (программа 8.5)
- B\*** Сортировка восходящая слиянием с отсечением файлов небольших размеров

По обыкновению мы должны выразить предостережение относительно того, что погоня за усовершенствованиями подобного рода, перед которой не в состоянии устоять многие программисты, могут в некоторых случаях приносить всего лишь незначительные выгоды и должны быть реализованы только после того, как будут сняты более важные вопросы. В таких случаях сортировка слиянием будет обладать явно выраженным превосходством перед быстрой сортировкой в том, что она устойчива и обеспечивает высокую скорость сортировки, равно как и недостатками, которые проявляются прежде всего в том, что она использует дополнительное пространство памяти, пропорциональное размерам массива. Если совокупность этих факторов складывается в пользу сортировки слиянием (при этом большое значение имеет быстрое действие), то предложенные усовершенствования заслуживают внимательного рассмотрения наряду с тщательным изучением программного кода, порожденного компиляторами, специальных свойств архитектуры машины и пр.



### РИСУНОК 8.8. ВОСХОДЯЩАЯ СОРТИРОВКА СЛИЯНИЕМ РАЗЛИЧНЫХ ВИДОВ ФАЙЛОВ

*Время выполнения сортировки слиянием не чувствительно к организации входных данных.*

*Приводимые здесь диаграммы показывают, что количество проходов, выполненное в рамках восходящей сортировки слиянием на файлах с произвольной организацией, на файлах с распределением Гаусса, на почти упорядоченных файлах, на почти обратно упорядоченных файлах и на файлах с произвольной организацией, обладающих 10 различными ключами (слева направо), зависит только от размера файла и не зависит от того, какими являются входные значения. Подобное поведение находится в резком противоречии с поведением быстрой сортировки и поведением множества других алгоритмов.*

С другой стороны, следует также повторить обычное предостережение о том, что программисты никогда не должны упускать из виду вопросов производительности во избежание совершенно неоправданных издержек. Все программисты (равно как и авторы) испытывают затруднения, когда несущественные, но вовремя не замеченные свойства реализации, выступают на передний план и подавляют все другие хитроумные механизмы реализации. Достаточно распространенной является ситуация, когда обнаруживается возможность уменьшить время выполнения той или иной реализации в два раза в тот момент, когда она подвергается тщательному анализу именно под этим углом зрения. Регулярное тестирование является наиболее эффективным защитным средством, позволяющим избегать подобных неприятных сюрпризов, которые, как известно, возникают в самый неподходящий момент.

Мы достаточно подробно рассматривали эти моменты в главе 5, однако привлекательность преждевременной оптимизации настолько сильна, что каждый раз, когда изучается возможность применения того или иного метода усовершенствования реализации на таком уровне детализации, целесообразно подумать об улучшении самих методов. Что касается оптимизации сортировки слиянием, то здесь можно чувствовать себя вполне спокойно, ибо программы 8.1—8.4 обладают всеми наиболее важными характеристиками производительности, которые исследовались выше: время их выполнения пропорционально  $N \log N$ , они нечувствительны к организации входных данных (см. рис. 8.8), они используют дополнительное пространство памяти, они могут быть реализованы с сохранением свойства устойчивости. Сохранение этих свойств в процессе снижения времени выполнения в общем случае не является особо трудной задачей.

## Упражнения

**8.31.** Реализовать восходящую сортировку слиянием без копирования массивов.

**8.32.** Разработать программу трехуровневой гибридной сортировки, использующую быструю сортировку, сортировку слиянием и сортировку вставками с целью получить метод, который по производительности не уступает наиболее эффективной быстрой сортировке (даже для малых файлов), но в то же время может гарантировать в наихудшем случае производительность с квадратичной зависимостью.

## 8.7. Реализация сортировки слиянием, ориентированной на связные списки

Для практической реализации в любом случае требуется дополнительное пространство памяти, так почему бы не рассмотреть возможность реализации сортировки слиянием, ориентированной на связные списки? Другими словами, чем тратить дополнительное пространство памяти на вспомогательный массив, не лучше ли использовать его для хранения связей? Иначе можно столкнуться с проблемой предварительной сортировки связного списка (см. раздел 6.9). Как оказалось, сортировка слиянием может быть успешно использована для сортировки связных списков. Полная реализация функции сортировки связных списков представлена в программе 8.6. Обратите внимание на то обстоятельство, что в данном случае программа фактического слияния столь же проста, как и программа процедуры слияния, ориентированная на массивы (программа 8.2).

Имея в своем распоряжении такую функцию слияния, легко получить рекурсивную нисходящую сортировку слиянием списков. Программа 8.7 является прямой рекурсивной реализацией функции, которая принимает в качестве входного параметра указатель на неупорядоченный список и возвращает указатель на список, содержащий те же элементы в отсортированном порядке. Эта программа выполняет свою работу, переупорядочивая узлы списка — память не резервируется ни для временных узлов, ни для списков. Для нахождения середины списка программа 8.7 использует следующий прием: разные реализации могут делать это либо передавая длину списка как параметр в рекурсивную программу, либо сохраняя длину в самом списке. Такая программа в рекурсивной формулировке проста для понимания, даже если реализует достаточно сложный алгоритм.

Восходящий подход "объединяй и властвуй" можно также применить и по отношению к сортировке слиянием связанных списков, хотя необходимость отслеживать связи со всеми подробностями делают его более сложным, чем он кажется на первый взгляд. Как было установлено в разделе 8.3 при рассмотрении нисходящих методов, ориентированных на работу с массивами, при разработке алгоритма восходящей сортировки списков слиянием не существует особых причин придерживаться точно того же набора операций слияния, которые выполняют рекурсивная версия или версия, использующая массивы.

### Программа 8.6. Слияние связанных списков

Данная программа сливает список, на который указывает **a**, со списком, на который указывает **b**, с помощью вспомогательного указателя **c**. Операция сравнения ключей в функции **merge** включает равенство, так что слияние будет устойчивым, если по условию список **b** следует за списком **a**. Для простоты принимаем, что все списки завершаются символом 0. Другие соглашения, касающиеся завершающих элементов в списках, также работают (см. табл. 3.1). Что еще важнее, мы *не* используем заголовочные узлы списка во избежание их распространения.

```
link merge(link a, link b)
{ node dummy(0); link head = &dummy, c = head;
  while ((a != 0) && (b != 0))
    if (a->item < b->item)
      { c->next = a; c = a; a = a->next; }
    else
      { c->next = b; c = b; b = b->next; }
  c->next = (a == 0) ? b : a;
  return head->next;
}
```

Одна из забавных версий восходящей сортировки связанных списков слиянием, которую нетрудно сформулировать, напрашивается сама собой: поместить элементы списка в циклический список, после чего перемещаться по списку, сливая пары упорядоченных подфайлов до тех пор, пока дело не будет сделано. Этот метод концептуально прост, однако (как это имеет место в случае низкоуровневых программ, работающих со связными списками) для их реализации требуется проявить недюжинную изобретательность (см. упражнение 8.36). Другая версия восходящей сортировки связанных списков слиянием, в основу которой положена та же идея, представлена в программе 8.8: содержать все сортируемые списки в рамках АТД очереди. Этот метод

также отличается концептуальной простотой, однако (как это имеет место в случае высокоуровневых программ, работающих со связными списками) для их реализации также требуется изобретать различные хитроумные способы.

Одно из важных свойств заключается в том, что этот метод способен извлечь пользу из любого порядка, который может присутствовать в исходном файле. В самом деле, количество проходов через список определяется не выражением  $\lceil \lg N \rceil$ , но скорее  $\lceil \lg S \rceil$ , где  $S$  есть число упорядоченных подфайлов в исходном массиве. Этот метод иногда называется *естественной* сортировкой слиянием. Что касается файлов с произвольной организацией, то в этом случае от данного метода трудно ожидать особых выгод, поскольку можно сэкономить разве что один-два прохода (на самом деле этот метод, по-видимому, будет обладать меньшим быстродействием, что обусловлено затратами на дополнительные проверки с целью определить, какой порядок установлен в файле), однако на практике достаточно часто встречаются файлы, состоящие из блоков упорядоченных подфайлов; в таких ситуациях рассматриваемый метод окажется достаточно эффективным.

### Программа 8.7. Сортировка связных списков слиянием сверху вниз

Эта программа выполняет сортировку, разбивая список, на который указывает  $c$ , на две части, на которые указывают, соответственно,  $a$  и  $b$ , подвергая сортировке обе эти части в рекурсивном режиме, с последующим использованием функции `merge` (программа 8.6) для получения окончательного результата. В конце входного файла должен стоять символ 0 (и следовательно, список  $b$  также должен завершаться нулем), а явно определенная команда  $c \rightarrow \text{next} = 0$  помещает 0 в конец списка  $a$ .

```
link mergesort(link c)
{
    if (c == 0 || c->next == 0) return c;
    link a = c, b = c->next;
    while ((b != 0) && (b->next != 0))
        { c = c->next; b = b->next->next; }
    b = c->next; c->next = 0;
    return merge(mergesort(a), mergesort(b));
}
```

### Программа 8.8. Восходящая сортировка связных списков слиянием

Эта программа использует АТД очереди (программа 4.18) для реализации восходящей сортировки слиянием. Элементы очереди представляют собой упорядоченные связные списки. После инициализации очереди списком длиной 1, программа просто удаляет из очереди два списка, сливает их, а полученный результат возвращает в эту же очередь и продолжает процесс до тех пор, пока в очереди не останется только один список. Это соответствует последовательности проходов через все элементы, при этом на каждом проходе длина упорядоченных списков удваивается, как и в случае восходящей сортировки слиянием.

```
link mergesort(link t)
{ QUEUE<link> Q(max);
  if (t == 0 || t->next == 0) return t;
  for (link u = 0; t != 0; t = u)
      { u = t->next; t->next = 0; Q.put(t); }
  t = Q.get();
  while (!Q.empty())
      { Q.put(t); t = merge(Q.get(), Q.get()); }
  return t;
}
```

## Упражнения

- 8.33. Разработать реализацию нисходящей сортировки связных списков слиянием, работающей со списками, которые содержат собственные длины в заголовочных узлах, и применяющей эти длины с целью определения способа разбивания списков на части
- 8.34. Разработать реализацию восходящей сортировки связных списков слиянием, работающей со списками, которые содержат собственные длины в заголовочных узлах, и применяющей эти длины с целью определения способа разбиения списков на части.
- 8.35. Добавить в программу 8.7 отсечение подфайлов малых размеров. Определить пределы, в рамках которых правильный выбор размеров отсекаемых файлов ускоряет действие полученной программы.
- 8.36. Реализовать восходящую сортировку слиянием применительно к циклическому связному списку, описание которого содержится в тексте.
- 8.37. Добавить отсечение подфайлов малых размеров в восходящую сортировку слиянием связных списков из упражнения 8.36. Определить пределы, в рамках которых правильный выбор размеров отсекаемых файлов ускоряет действие полученной программы.
- 8.38. Добавить в программу 8.7 отсечение подфайлов малых размеров. Определить пределы, в рамках которых правильный выбор размеров отсекаемых файлов ускоряет действие полученной программы.
- 8.39. Нарисовать дерево типа "объединяй и властвуй", которое отображает слияния, которые выполняет программа 8.8 для  $N = 16, 24, 31, 32, 33$  и  $39$ .
- 8.40. Нарисовать дерево типа "объединяй и властвуй", которое подводит итог слияниям, выполняемым сортировкой слиянием на циклическом списке (упражнение 8.38) для  $N = 16, 24, 31, 32, 33$  и  $39$ .
- 8.41. Выполнить эмпирические исследования с целью выдвижения гипотезы, касающейся числа упорядоченных подфайлов в массиве из  $N$  случайных 32-разрядных целых чисел.
- 8.42. Определить эмпирическим путем количество проходов, необходимых для выполнения естественной сортировки слиянием случайных 64-разрядных ключей при  $N = 10^3, 10^4, 10^5$  и  $10^6$ . *Совет:* чтобы выполнить это упражнение, не обязательно пользоваться сортировкой (и даже не нужно генерировать полные 64-разрядные ключи).
- 8.43. Преобразовать программу 8.8 в процедуру естественной сортировки слиянием, предварительно заполнив очередь упорядоченными подфайлами, которые содержатся во входном файле.
- 8.44. Реализовать естественную сортировку слиянием применительно к массивам.



## 8.8. Возврат к рекурсии

Программы, представленные в данной главе, и быстрая сортировка, которая рассматривалась в предыдущей главе, — суть типичные алгоритмы типа "разделяй и властвуй". Мы ознакомимся с несколькими алгоритмами подобной структуры в последующих главах, так что более пристальное изучение основных характеристик соответствующих программных реализаций представляется вполне оправданным.

Быструю сортировку более корректно было бы назвать алгоритмом типа "*разделяй и властвуй*": в рекурсивных реализациях после активизации программы большая часть работы выполняется *перед* рекурсивными вызовами. С другой стороны, рекурсивная сортировка слиянием еще больше выдержана в духе принципа "разделяй и властвуй": прежде всего, файл делится на две части, затем обработке ("воздействию власти") по отдельности подвергаются обе части. Сначала сортировке слиянием подвергаются файлы небольших размеров, в заключение обработке подвергается самый большой подфайл. Быстрая сортировка начинается с обработки наибольшего подфайла и завершается обработкой подфайлов небольших размеров. Интересно провести сравнение этих алгоритмов в контексте аналогии с управлением коллективом сотрудников, приводимой в начале настоящей главы: быстрая сортировка соответствует тому, что каждый руководящий работник затрачивает свои усилия на то, чтобы правильно разбить задачу на подзадачи, так что работа будет успешно выполнена, если успешно выполнены все подзадачи, в то время как сортировка слиянием соответствует тому, что каждый руководящий работник выполняет быструю произвольную разбивку задачу напополам, а затем затрачивает все свои усилия на то, чтобы преодолеть последствия подобных действий после того, как соответствующие подзадачи будут решены.

Это различие ясно показывает наличие в нерекурсивных реализациях двух методов. Быстрая сортировка должна поддерживать стек, поскольку она должна сохранять большие подзадачи, которые дробятся в зависимости от организации входных данных. Сортировка слиянием допускает простые нерекурсивные реализации, поскольку способ разбиения файла на части не зависит от природы данных, благодаря чему становится возможным изменение очередности, в которой она решает подзадачи, и это обстоятельство позволяет упростить программу.

Можно, конечно, приводить доводы в пользу того, что быструю сортировку следует рассматривать как нисходящий алгоритм, поскольку он начинает работу на вершине дерева рекурсии, а затем спускается вниз, дабы завершить сортировку. Можно также подумать и о нерекурсивной сортировке, которая проходит через дерево рекурсии снизу вверх, но вдоль уровней. Таким образом, сортировка многократно проходит по массивам, дробя файлы на подфайлы меньших размеров. Применительно к массивам этот метод не имеет широкого практического применения, что объясняется высокими затратами ресурсов для отслеживания подфайлов; однако, применительно к связным спискам он аналогичен восходящей сортировке слиянием.

Мы также отметили, что сортировка слиянием и быстрая сортировка отличаются друг от друга в плане проблемы устойчивости. В случае сортировки слиянием, если предположить, что подфайлы отсортированы с соблюдением концепции устойчивости, то вполне достаточно того, чтобы слияние проводилось в устойчивом режиме, а это вполне достижимо. Рекурсивная структура алгоритма немедленно приводит к

индуктивному доказательству устойчивости. Что касается реализации быстрой сортировки, ориентированной на массивы, то простого способа устойчивого разбиения файлов не видно, так что возможность положительного решения проблемы устойчивости исключается еще до того, как рекурсия вступит в дело. Тем не менее, прямая реализация быстрой сортировки применительно к связным спискам является устойчивой (см. упражнение 7.4).

Как было показано в главе 5, алгоритмы с одним рекурсивным вызовом могут быть сведены к циклу, но алгоритмы с двумя рекурсивными циклами, подобные сортировке слиянием или быстрой сортировке, открывают двери в мир алгоритмов типа "разделяй и властвуй" и древовидных структур, в котором отводится место и для наших лучших алгоритмов. Сортировка слиянием и быстрая сортировка заслуживают тщательного изучения не только из-за их важного практического значения, но и в силу того, что они позволяют глубже погрузиться в сущность рекурсии, которая может сослужить добрую службу при разработке и понимании других рекурсивных алгоритмов.

## Упражнения

- **8.45.** Предположим, что сортировка слиянием реализована таким образом, что дробление файла осуществляется в произвольной точке, а не точно в середине файла. Сколько операций сравнения выполняются в среднем по этому методу для сортировки  $N$  элементов?
- **8.46.** Провести анализ эффективности сортировки слиянием при сортировке строк. Сколько в среднем нужно выполнить операций сравнения символов при сортировке файлов больших размеров?
- **8.47.** Провести эмпирические исследования с целью сравнения эффективности быстрой сортировки связных списков (см. упражнение 7.4) и нисходящей сортировки слиянием связных списков (программа 8.7).

## Очереди по приоритетам и пирамидальная сортировка

**В**о многих приложениях требуется обработка записей с упорядоченными определенным образом ключами, но не обязательно в строгом порядке и не обязательно все сразу. Часто мы накапливаем некоторый набор записей, после чего обрабатываем запись с максимальным значением ключа, затем, возможно, накопление записей продолжается, потом обрабатывается запись с наибольшим текущим ключом и т.д. Соответствующая структура данных в подобного рода средах поддерживает операции вставки нового элемента и удаления наибольшего элемента. Такая структура данных называется *очередью по приоритетам*. Использование очереди по приоритетам подобно использованию обычных очередей (удаляется самый старый элемент) и стеков (удаляется самый новый элемент), однако их эффективная реализация представляет собой довольно трудную задачу. Очередь по приоритетам является наиболее важным примером обобщенного абстрактного типа данных (АТД), которые обсуждались в разделе 4.6. Фактически, очередь по приоритетам представляет собой оправданное обобщение стека и очереди, поскольку эти структуры данных можно реализовать посредством очередей по приоритетам, используя соответствующий механизм установки приоритетов (см. упражнения 9.3 и 9.4).

**Определение 9.1.** *Очередь по приоритетам представляет собой структуру элементов с ключами, которая поддерживает две основные операции: вставку нового элемента и удаление элемента с наибольшим значением ключа.*

Приложениями очередей по приоритетам являются системы моделирования, в рамках которых ключи могут соответствовать моментам возникновения событий, что обеспечивает возможность их обработки в хронологическом порядке; системы планирования заданий в компьютерных системах, где ключи могут соответствовать приоритетам, указывающим, какой из пользователей должен быть обслужен первым; а также численные расчеты, в которых ключами могут быть ошибки в вычислениях, в которых приоритеты показывают, что наиболее грубая ошибка должна быть исправлена первой.

Любую очередь по приоритетам можно использовать как основу для алгоритма сортировки, устанавливая все записи в очередь, а затем последовательно исключая из нее наибольшие текущие записи, чтобы получить последовательность записей в обратном порядке. Далее в книге будет показано, как следует использовать очередь по приоритетам в качестве строительных блоков для более совершенных алгоритмов. В части 5 мы разработаем алгоритм сжатия файлов, использующий программы из данной главы, в главе 7 увидим, как очереди по приоритетам могут служить подходящими абстракциями для упрощения понимания взаимоотношений между множеством фундаментальных алгоритмов поиска в графах. Здесь упомянуто всего лишь несколько примеров той важной роли, которую играют очереди по приоритетам как базовые инструментальные средства при разработке алгоритмов.

На практике очереди по приоритетам намного сложнее, чем это следует из только что сформулированного простого определения, поскольку существуют несколько других операций, которые могут потребоваться для поддержки очередей при всех условиях, которые могут возникнуть во время их использования. И в самом деле, одна из основных причин того, что многие реализации очередей с приоритетами находят широкое практическое применение, заключается в их гибкости, позволяющей клиентским программам выполнять различные операции над наборами записей с ключами. Необходимо построить и поддерживать структуры данных, содержащие записи с числовыми ключами (*приоритетами*), которые поддерживают некоторые из следующих операций:

- Создать (*construct*) очередь по приоритетам из  $N$  заданных элементов,
- Вставить (*insert*) новый элемент,
- Удалить наибольший (*remove the maximum*) элемент,
- Изменить приоритет (*change the priority*) произвольно выбранного элемента,
- Удалить (*remove*) произвольно выбранный элемент,
- Объединить (*join*) две очереди по приоритетам в одну.

Если записи имеют дублированные ключи, мы считаем, что "наибольший" означает "любая запись с максимальным значением ключа". Как и в случае других структур данных, в этот набор потребуется добавить стандартные операции *создать*, *проверить наличие* элементов и, возможно, операции *уничтожить* и *копировать*.

Имеются области, в которых эти операции взаимно перекрываются, а в некоторых случаях удобно дать определения других, подобных операций. Например, у некоторых клиентских программ часто возникает необходимость *найти наибольший* (*find the maximum*) элемент в очереди по приоритетам без его удаления из очереди. Или же

понадобится операция *заменить наибольший* (*replace the maximum*) элемент новым элементом. Мы можем реализовать подобного рода операции за счет использования в качестве строительных блоков две базовых операции: операцию *найти наибольший* можно представить через операцию *удалить наибольший* и следующую за ней операцию *вставить*, а операцию *заменить наибольший* можно представить либо через операцию *вставить* и следующую за ней *удалить наибольший*, либо через операцию *удалить наибольший* и следующую за ней *вставить*. Однако более эффективный программный код обычно получается за счет реализации таких операций непосредственно, при условии, что в них существует потребность и существует их точное описание. Точное описание не всегда однозначно, как это может показаться на первый взгляд. Например, только что сформулированные два варианта операции *заменить наибольший* (*replace the maximum*) существенно отличаются друг от друга: первый из них приводит к тому, что размер очереди временно увеличивается на один элемент, а во втором в очередь каждый раз вставляется новый элемент. Аналогично, операция *изменить приоритет* может быть реализована как *удалить* с последующей операцией *вставить*, а операция *построить* может быть реализована как многократное применение операции *вставить*.

Для некоторых приложений более удобным может оказаться поменять ориентацию на обратную и работать с наименьшими элементами, а не с наибольшими. Мы предпочитаем работать, главным образом, с очередями по приоритетам, которые настроены на доступ к наибольшим ключам, а не к наименьшим. Когда потребуется противоположная ориентация, мы будем называть ее (т.е. очередь по приоритетам, которая позволит *удалять наименьший* элемент (*remove the minimum*)) очередью по приоритетам, *ориентированной на минимальный элемент* (*minimum-oriented*).

### Программа 9.1. Базовый тип абстрактных данных очереди по приоритетам

Данный интерфейс определяет операции над простейшим типом очереди по приоритетам: инициализировать, проверить наличие, добавить новый элемент, удалить наибольший элемент. Элементарные программные реализации этих функций обеспечивают в наихудшем случае линейное время их выполнения на массивах и списках, но в этой главе встретятся реализации, для которых время выполнения всех операций гарантировано не превосходит величины, пропорциональной логарифму количества элементов в очереди. Как обычно, параметр конструктора определяет максимальное число элементов, ожидаемых для размещения в очереди, причем некоторые реализации могут его игнорировать.

```
template <class Item>
class PQ
{
    private :
        // Программный код, который зависит от реализации
    public:
        PQ(int);
        int empty() const;
        void insert(Item);
        Item getmax();
};
```

Эта очередь по приоритетам суть прототипный АТД (см. главу 4): он представляет четко определенный набор операций над данными и является удобным абстрактным понятием, которое позволяет отделять прикладные программы (клиенты) от различных приложений, которые предстоит рассмотреть в настоящей главе. Интерфейс, заданный программой 9.1, определяет большую часть базовых операций над очередью по приоритетам; более полный интерфейс рассматривается далее в разделе 9.5. Строго говоря, различные подмножества различных операций, которые мы предпочтем включить в свой рабочий набор, приводит к различным абстрактным структурам данных, но для очередей по приоритетам наиболее характерными являются операции *удалить наибольший* (*remove-the-maximum*) и *вставить* (*insert*), поэтому им и будет уделяться основное внимание.

Различные реализации очередей по приоритетам обладают различными рабочими характеристиками в зависимости от того, какие операции должны выполняться, а различные приложения требуют эффективного выполнения различных наборов операций. И в самом деле, различия в рабочих характеристиках в принципе могут быть *единственным* видом различий, которые возникают при использовании понятия абстрактного типа данных. Такого рода ситуация приводит к различным компромиссам в отношении затрат. В данной главе анализируются различные пути достижения подобного рода компромиссов. Мы почти приблизимся к идеалу в смысле возможности выполнить операцию *удалить наибольший* за время, которое находится в логарифмической зависимости от числа элементов в очереди, а все остальные операции — за постоянное время.

Сначала это положение иллюстрируется в разделе 9.1 на примере анализа нескольких элементарных структур данных, предназначенных для реализации очередей по приоритетам.

Далее, в разделах 9.2—9.4 внимание сосредоточивается на рассмотрении классической структуры данных, получившей название *сортирующее дерево* (*heap*), которая обеспечивает эффективную реализацию всех операций, кроме операции *объединить*. Кроме того, в разделе 9.4 исследуется важный алгоритм сортировки, который естественным образом вытекает из этих реализаций. После этого мы более подробно проанализируем некоторые проблемы, связанные с разработкой полных АТД очереди по приоритетам, в разделах 9.5 и 9.6. И в заключение, в разделе 9.7 рассматриваются более сложные структуры данных, получившие название *биномиальных очередей* (*binomial queue*), которые используются для реализации всех операций (в том числе и операции *объединить*) в наихудшем случае логарифмической зависимости времени выполнения.

В процессе исследований всего разнообразия структур данных не следует упускать из виду как основные компромиссы, достигнутых между связным и последовательным распределением памяти (см. главу 3), так и проблемы, возникающие при организации пакетов, используемых прикладными программами. В частности, некоторые из алгоритмов с улучшенными свойствами, описания которых появятся позже в этой книге, представляют собой клиентские программы, которые используют очереди по приоритетам.

## Упражнения

▷ 9.1. Буква означает операцию *вставить*, а звездочка — операцию *удалить наибольший* элемент из последовательности **PRIO\*R\*\*I\*T\*Y\*\*\*QUE\*\*\*U\*E**.

Дать последовательность значений, возвращаемых операциями *возвратить наибольший* элемент.

▷ 9.2. Добавить к условиям упражнения 9.1 знак плюс, означающий операцию *объединить*, и круглые скобки для ограничения пределов очереди по приоритетам, построенной операциями, заключенными в эти скобки. Показать содержимое очереди по приоритетам, соответствующее последовательности

**((PRIO)+(R\*IT\*Y\*))\*\*\* + (QUE\*\*\*U\*E)**.

○ 9.3. Объяснить, как использовать АТД очереди по приоритетам для реализации АТД стека.

○ 9.4. Объяснить, как использовать АТД очереди по приоритетам для реализации АТД очереди.

## 9.1. Элементарные реализации

Базовые структуры данных, которые обсуждались в главе 3, предоставляют множество различных возможностей для реализации очередей по приоритетам. Программа 9.2 демонстрирует реализацию, которая в качестве базовой структуры данных использует неупорядоченный массив. Операция *найти наибольший* элемент реализуется следующей последовательностью действий: сначала производится просмотр массива с целью обнаружения наибольшего элемента, затем осуществляется замена наибольшего элемента на последний элемент с последующим уменьшением размера очереди на единицу. На рис. 9.1 показано содержимое массива для тестовой последовательности операций. Базовая реализация соответствует тем реализациям, которые можно было бы видеть в главе 4 для стеков и очередей небольших размеров (см. программы 4.7 и 4.15) и полезны при работе с очередями небольших размеров. Основные различия между ними связаны с их производительностью. Для стеков и очередей есть возможность разрабатывать реализации всех операций, которые выполняются за постоянное время; что касается очередей по приоритетам, то легко отыскать такие реализации, в рамках которых *одна* из функций *вставить* или *удалить наибольший* выполняется за постоянное время, однако найти реализацию, в которой про-

B		B
E		B E
·	E	B
S		B S
T		B S T
I		B S T I
·	T	B S I
N		L S I N
·	S	B N I
F		B N I F
I		B N I F I
R		B N I F I R
·	R	B N I F I
S		B N I F I S
T		B N I F I S T
·	T	B N I F I S
·	S	B N I F I
O		B N I F I O
U		B N I F I O U
·	U	B N I F I O
T		B N I F I O T
·	T	B N I F I O
·	O	B N I F I
·	N	B I I F
·	I	B F I
·	I	B F
·	F	B
·	B	

**РИСУНОК 9.1. ПРИМЕР ОЧЕРЕДИ ПО ПРИОРИТЕТАМ (ПРЕДСТАВЛЕНИЕ ДАННЫХ В ВИДЕ НЕУПОРЯДОЧЕННОГО МАССИВА)**

*Эта последовательность отражает результаты выполнения некоторой последовательности операций в левой колонке (сверху вниз), при этом буквы обозначают операцию *вставка*, а звездочка — операцию *удалить наибольший*. Каждая строка отображает операцию, удаляемую в результате выполнения каждой операции *удалить наибольший* букву и содержимое массива после выполнения этой операции.*

изводительность обеих операций достаточно высока, — весьма непростая задача, и это является предметом разговора в данной главе.

Можно использовать упорядоченные и неупорядоченные последовательности, реализованные в виде связанных списков или массивов. Выбор в пользу того, оставить элементы неотсортированными или разместить их в определенном порядке, определяется тем, что упорядоченная совокупность элементов позволяет выполнить операции *удалить наибольший* или *найти наибольший* за постоянное время, но это также означает необходимость прохода по всему списку, чтобы выполнить операцию *вставить*. Неупорядоченная последовательность элементов позволяет выполнить операцию *вставить* за постоянное время, а для операции *удалить наибольший* или *найти наибольший* потребуется просмотреть весь список. Неупорядоченность представляет собой прототипный "ленивый" подход к решению проблемы, в рамках которого выполнение работы откладывается до тех пор, пока это не станет необходимым (в данном случае, поиск наибольшего элемента); упорядоченная последовательность представляет собой прототипный "энергичный" подход к решению проблемы, когда заранее выполняется максимально возможный объем работы (поддержка списка отсортированным на случай выполнения операции *вставка*), дабы обеспечить максимальную эффективность последующих операций. В любом из этих случаев можно использовать представление данных в виде массива или связанного списка, при этом основная альтернатива состоит в том, что (двух) связный список позволяет выполнять операцию *удалить* (и в случае неупорядоченных данных операцию *объединить*) за постоянное время, но требует большего объема памяти для хранения связей.

### Программа 9.2. Реализация очереди по приоритетам с использованием массивов

Эта реализация, которую можно сравнить с реализациями стеков и очередей с использованием массивов, которые рассматривались в главе 4 (см. программы 4.7 и 4.15), хранит элементы в неупорядоченном массиве. Элементы добавляются в конец массива и удаляются с конца массива, как это имеет место в стеке.

```

template <class Item>
class PQ
{
    private:
        Item *pq;
        Int N;
    public:
        PQ (int maxN)
            { pq = new Item[maxN]; N = 0; }
        int empty() const
            {return N == 0; }
        void insert (Item item)
            { pq[N++] = item; }
        Item getmax( )
            {int max = 0;
             for (int j = 1; j<N; j++)
                 if (pq[max] < pq[j]) max = j;
             exch(pq[max], pq[N-1]);
             return pq[--N];
            }
};

```



**Таблица 9.1. Стоимость операций, поддерживающих очередь по приоритетам, для наихудшего случая**

Показатели производительности реализации АТД очереди по приоритетам колеблются в широких пределах, что следует из данной таблицы, содержащей временные показатели для наихудшего случая (в пределах постоянного множителя для больших  $N$ ) в условиях различных методов. Элементарные методы (первые четыре строки) требуют для выполнения некоторых операций постоянного времени и линейного времени для остальных; более совершенные методы гарантируют постоянное или линейное время выполнения для большего числа или даже для всех операций.

		Удалить Вставить наибольший	Удалить наибольший	Найти	Изменить приоритет	Объединить
упорядоченный массив	$N$	1	$N$	1	$N$	$N$
упорядоченный список	$N$	1	1	1	$N$	$N$
неупорядоченный массив	1	$N$	1	$N$	1	$N$
неупорядоченный список	1	$N$	1	$N$	1	1
сортирующее дерево	$\lg N$	$\lg N$	$\lg N$	1	$\lg N$	$N$
биномиальная очередь	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$
теоретически наилучший	1	$\lg N$	$\lg N$	1	1	1

Для наихудшего случая в условиях различных реализаций показатели стоимости различных операций (пределах постоянного коэффициента) для очереди по приоритетам размера  $N$  сведены в таблицу 9.1.

При разработке завершенной реализации необходимо соблюдать все интерфейсные требования — в особенности то, как клиентские программы осуществляют доступ к узлам при выполнении операций *удалить* и *изменить приоритет*, и как они осуществляют доступ к самим очередям по приоритетам как к типам данных при выполнении операции *объединить*. Эти проблемы изучаются разделах 9.4 и 9.7, в которых рассматриваются две завершённых реализации: в одной реализации используются двухсвязные неупорядоченные списки, а в другой — биномиальные очереди.

Время выполнения клиентской программы, использующей очереди по приоритетам, зависит не только от ключей, но также и от смеси различных операций. Не следует упускать из виду простые реализации, поскольку во многих практических ситуациях они довольно часто оказываются эффективнее более сложных методов. Например, реализация, работающая с неупорядоченными списками, может оказаться приемлемой в приложениях, в которых выполняются лишь немногие операции *удалить наибольший* и в то же время очень большое количество вставок, тогда как упорядоченный список лучше подходит в тех случаях, когда выполняется большое число операций *найти наибольший* либо когда вставляемые элементы преимущественно больше уже находящихся в очереди по приоритетам.

## Упражнения

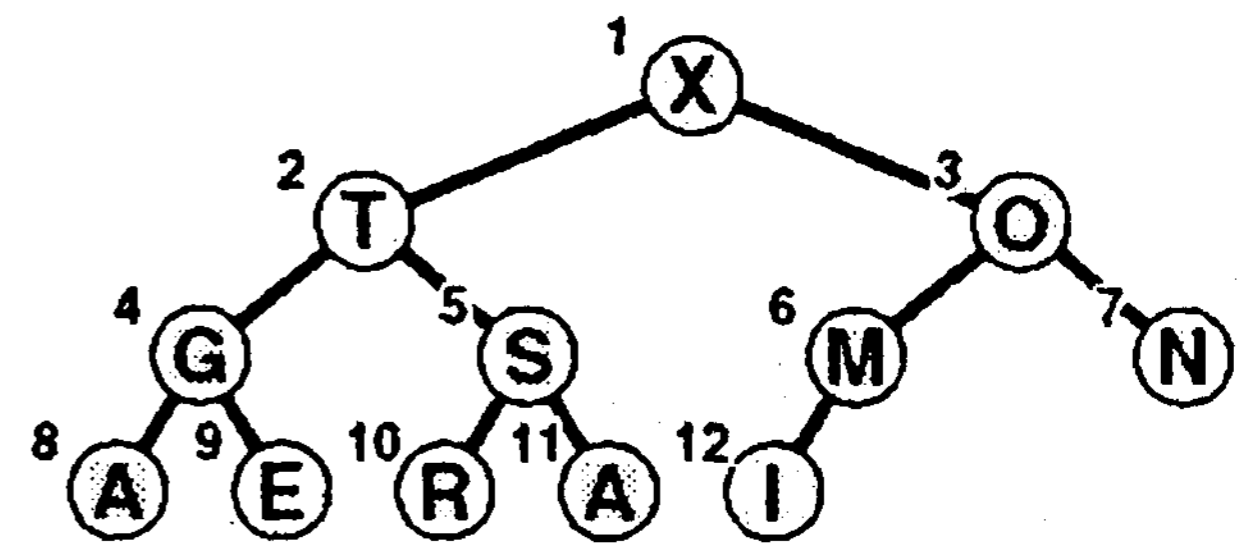
- ▷ 9.5. Найти недостатки следующей идеи: почему бы для реализации операции *найти наибольший* (*find the maximum*) не отслеживать максимальное значение из числа элементов, включенных на текущий момент, а затем возвращать это значение как результат операции?

- ▷ 9.6. Показать содержимое массива после выполнения последовательности операций из рис. 9.1.
- 9.7. Напишите реализацию основного интерфейса очереди по приоритетам, который использует упорядоченный массив в качестве базовой структуры данных.
- 9.8. Напишите программную реализацию основного интерфейса очереди по приоритетам, которая использует неупорядоченный связный список в качестве базовой структуры данных *Совет*: см. программу 4.8 и 4.14.
- 9.9. Напишите программную реализацию основного интерфейса очереди по приоритетам, который использует упорядоченный связный список в качестве базовой структуры данных *Совет*: см. программу 3.11.
- 9.10. Рассмотреть "ленивую" реализацию, в условиях которой список упорядочивается только когда выполняются операции *удалить наибольший* или *найти наибольший*. Вставки, сделанные с момента предыдущей сортировки, содержатся в отдельном списке, затем они сортируются и при необходимости подвергаются слиянию. Рассмотреть преимущества такой реализации перед элементарными реализациями, в основу которых положены неупорядоченные и упорядоченные списки.
- 9.11. Написать клиентскую программу-драйвер, которая использует функцию **insert** для заполнения очереди по приоритетам, затем функцию **getmax** для удаления половины ключей, затем снова **insert** для заполнения очереди, после чего для удаления всех ключей используется **getmax**, и все это проделывается многократно над случайными последовательностями ключей различной длины, изменяющейся в широких пределах. Кроме того, программа должна измерять время, затраченное на каждое выполнение программы, и распечатывать или строить графики среднего времени выполнения.
- 9.12. Написать клиентскую программу, представляющую собой драйвер производительности, которая использует функцию **insert** для заполнения очереди по приоритетам, затем выполняет функции **getmax** и **insert** столько раз, сколько она способна их выполнить в течение 1 секунды на случайных последовательностях ключей разной длины, значения которой колеблются в широких пределах, от малых до больших. Программа должна измерять время, затраченное на каждое выполнение программы, и распечатывать или строить графики среднего числа функций **getmax**, которое удалось выполнить.
- 9.13. Воспользуйтесь клиентской программой из упражнения 9.12, чтобы сравнить реализацию неупорядоченного массива, представленную программой 9.2, с реализацией неупорядоченного списка из упражнения 9.8.
- 9.14. Воспользуйтесь клиентской программой из упражнения 9.12, чтобы сравнить реализацию упорядоченного массива с реализацией упорядоченного списка из упражнений 9.7 и 9.9.
- 9.15. Написать клиентскую программу-драйвер, которая использует функции интерфейса очереди по приоритетам из программы 9.1 в трудных и патологических случаях, которые могут встретиться в практических реализациях. В категорию простых случаев входят уже упорядоченные ключи, ключи, установленные в обратном порядке, случаи, когда все ключи одинаковы и когда последовательности ключей содержат только два различных значения.

**9.16.** (За этим упражнением на самом деле стоят 24 упражнения.) Докажите правильность границ, установленных для 4 элементарных реализаций, представленных в таблице 9.1, используя для доказательства реализацию операций *вставить* (*insert*) и *удалить наибольший* (*remove the maximum*) из программы 9.2, и реализации, выполненные в упражнениях 9.7—9.9, а также формальное описание методов реализации других операций. Что касается операций *удалить* (*remove*), *изменить приоритет* (*change priority*) и *объединить* (*join*), предположите, что имеется средство, обеспечивающее прямой доступ к объекту ссылки.

## 9.2. Пирамидальная структура данных

Основной темой настоящей главы является простая структура данных, получившая название *сортирующего дерева* (*heap*), которая может эффективно поддерживать основные операции в очереди по приоритетам. В сортирующем дереве записи хранятся в виде массива таким образом, что каждый ключ обязательно принимает значение большее, чем значения двух других ключей, занимающих относительно него строго определенные положения. В свою очередь, каждый из этих ключей должен быть больше, чем два других определенных ключа и т.д. Подобное упорядочение легко обнаружить, если рассматривать эти ключи как входящие в бинарную древовидную структуру с ребрами, идущими от каждого ключа к двум другим ключам, о которых известно, что они меньше его по значению.



1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

**Определение 9.2.** *Дерево называется пирамидально упорядоченным (heap-ordered), если ключ в каждом его узле больше или равен ключам всех потомков этого узла (если таковые имеются). Эквивалентная формулировка: ключ в каждом узле пирамидально упорядоченного дерева меньше или равен ключу узла, который является родителем данного узла.*

**Лемма 9.1.** *Ни один из узлов пирамидально упорядоченного дерева не может иметь ключа, большего чем ключ корня дерева.*

На любое дерево можно наложить ограничения, обусловленные пирамидальной упорядоченностью. Однако особенно удобно пользоваться *полным бинарным деревом* (*complete binary tree*). Из главы 3 известно, что мы можем начертить такую структуру, поместив в верхней части страницы корневой узел, а затем спускаясь вниз по странице и перемещаясь слева направо, подсоединять к каждому конкретному узлу предыдущего уровня два узла текущего уровня до тех пор, пока не будут помещены все  $N$  узлов. Достаточ-

### РИСУНОК 9.2. ПРЕДСТАВЛЕНИЕ ПОЛНОГО ДВОИЧНОГО ДЕРЕВА В ВИДЕ ПИРАМИДАЛЬНО УПОРЯДОЧЕННОГО МАССИВА

*Если рассматривать элемент в позиции  $\lfloor i/2 \rfloor$  массива как родителя элемента в позиции  $i$ , для  $2 \leq i \leq N$  (или, что эквивалентно, считая  $i$ -й элемент родителем  $2i$ -го и  $2i+1$ -го элементов), то становится возможным удобное представление совокупности элементов массива в виде дерева. Такое соответствие эквивалентно нумерации полного двоичного дерева (элементы на нижнем уровне нумеруются, начиная с самого левого) по уровням. Дерево называется пирамидально упорядоченным, если ключ любого заданного узла больше или равен ключам узлов потомков. Сортирующее дерево есть представление в виде массива полного упорядоченного двоичного дерева.  $i$ -й элемент сортирующего дерева больше или равен по величине  $2i$ -го и  $2i+1$ -го элементов.*

но просто представить полное бинарное дерево в виде массива, поместив корневой узел в позицию 1, его потомков в позицию 2 и 3, узлы следующего уровня в позиции 4, 5, 6, 7 и т.д., как показано на рис. 9.2.

**Определение 9.3.** *Сортирующее дерево есть совокупность узлов с ключами, образующих полное пирамидально упорядоченное бинарное дерево, представленное в виде массива.*

Можно было бы воспользоваться связным представлением пирамидально упорядоченных деревьев, но полные деревья предоставляют возможность задействовать компактное представление в виде массива, в котором легко переходить с некоторого узла к его родителю или к его предкам без необходимости поддержки явных связей. Родителя узла, находящегося в позиции  $i$ , необходимо искать в позиции  $\lfloor i/2 \rfloor$ , и, соответственно, два потомка узла в позиции  $i$  находятся в позициях  $2i$  и  $2i + 1$ . При подобной организации прохождение по такому дереву выполняется проще, чем если бы это дерево было реализовано в связном представлении, поскольку в таком случае могут понадобиться связи дерева, принадлежащие каждому ключу, чтобы иметь возможность перемещаться вверх и вниз по дереву (каждый элемент будет иметь один указатель на родителя и один указатель на каждого потомка). Полные бинарные деревья, представленные в виде массивов, являются жесткими структурами, но все же обладают достаточной гибкостью, чтобы позволить реализовать эффективные алгоритмы, манипулирующие очередями по приоритетам.

В разделе 9.3 мы убедимся в том, что можем воспользоваться сортирующими деревьями для реализации всех операций на очередях по приоритетам (за исключением операции *объединить*) таким образом, что на свое выполнение они в худшем случае потребуют логарифмическое время. Все такие реализации функционируют вдоль некоторого пути в сортирующем дереве (перемещаясь от родителя к потомкам по направлению вниз или от потомка к родителю по направлению вверх, не меняя при этом направления движения). Как уже было показано в главе 3, все пути в полном дереве, состоящем из  $N$  узлов, содержат в себе порядка  $\lg N$  узлов: примерно  $N/2$  узлов находятся на самом нижнем уровне,  $N/4$  узлов — потомки которых занимают нижний уровень,  $N/8$  узлов — "внуки" которых занимают нижний уровень и т.д. Каждое следующее поколение узлов содержит наполовину меньше узлов, чем предыдущее, всего же может быть максимум  $\lg N$  поколений.

Можно также воспользоваться явными связными представлениями древовидной структуры для разработки эффективных реализаций операций над очередями по приоритетам. В качестве примеров рассматриваются полные деревья с тремя связями (см. раздел 9.5), сортировка повторной выборкой (см. программу 5.19) и биномиальные очереди (см. раздел 9.7). Как и в случае простых стеков и очередей, одна из главных причин, побуждающих рассматривать связные представления, заключается в том, что они освобождают от необходимости заранее знать максимальные размеры очереди, что требуется в обязательном порядке в случае представления в виде массива. Мы также можем извлечь в некоторых ситуациях определенную пользу из гибкости, обеспечиваемой связными структурами, при разработке эффективных алгоритмов. Читателям, которые не имеют достаточного опыта использования явных древовидных структур, мы настоятельно рекомендуем прочитать главу 12, чтобы изучить базовые методы реализации даже более важных абстрактных типов данных, таких как сим-

вольные таблицы, прежде чем иметь дело со связными представлениями деревьев, рассматриваемых в упражнениях этой главы и в разделе 9.7. Однако внимательное изучение связных структур можно оставить до второго чтения, поскольку основной темой настоящей главы является сортирующее дерево (представление в виде массива без связей полного пирамидально упорядоченного дерева).

## Упражнения

- ▷ 9.17. Является ли массив, отсортированный в нисходящем порядке, сортирующим деревом?
- 9.18. Наибольший элемент сортирующего дерева должен появиться в позиции 1, а второй наибольший элемент должен занимать позицию 2 или 3. Представьте список позиций в сортирующем дереве из 15 элементов, в котором  $k$ -й наибольший элемент (i) может появиться и (ii) не может появиться, для  $k = 2, 3, 4$  (в предположении, что все значения попарно различны).
- 9.19. Решить задачу 9.18 для общего значения  $k$  как функции от  $N$ , представляющего собой размер сортирующего дерева.
- 9.20. Решить задачи 9.18 и 9.19 для  $k$ -го наименьшего элемента.

## 9.3. Алгоритмы для сортирующих деревьев

Все алгоритмы очередей по приоритетам для сортирующих деревьев работают таким образом, что сначала вносят простое изменение, способное нарушить структуру пирамиды, затем выполняют проход вдоль пирамиды, внося при этом в сортирующее дерево такие изменения, которые гарантируют, что структура сортирующего дерева сохраняется везде. Этот процесс иногда называют *установлением пирамидального порядка (heapifying)*. Возможны два случая. Когда приоритет какого-либо узла увеличивается (или в нижний уровень сортирующего дерева добавляется новый узел), мы должны двигаться вверх по дереву, чтобы восстановить структуру сортирующего дерева. Когда приоритеты каких-либо узлов уменьшаются (например, при замене корневого узла новым узлом), необходимо пройти вниз по дереву, чтобы восстановить структуру сортирующего дерева. Для начала обсудим, как реализовать две эти базовые функции, а затем подумаем, как их использовать в различных операциях с очередями по приоритетам.

Если свойства сортирующего дерева нарушены из-за того, что ключ некоторого узла становится больше ключа родительского узла, можно сделать шаг в направлении исправления этого нарушения, обменяв местами этот узел с его родителем. После обмена этот узел становится больше, чем оба его потомка (один из них — это прежний родитель, другой меньше, чем старый родитель, поскольку он был потомком этого узла), но все еще может оставаться больше своего родителя. Мы можем устранить и это нарушение аналогичным способом и продвигаться далее вверх по дереву, пока не достигнем узла с наибольшим ключом, каковым является корень дерева. Пример описанного процесса показан на рис. 9.3. Программа проста и понятна, в ее основе лежит тот факт, что родитель узла, занимающего позицию  $k$  в сортирующем дереве, находится в позиции  $k / 2$  этого дерева. Программа 9.3 представляет собой реализацию функции, которая восстанавливает возможные нарушения, обусловленные

увеличением приоритета в заданном узле сортирующего дерева, благодаря продвижению вверх по дереву.

### Программа 9.3. Восходящая установка структуры сортирующего дерева

Чтобы восстановить пирамидальную структуру после того, как приоритет одного из узлов сортирующего дерева изменился, мы продвигаемся вверх по дереву, меняя при необходимости местами узел в позиции  $k$  с его родителем (который находится в позиции  $k/2$ ) и продолжаем этот процесс до тех пор, пока выполняется условие  $a[k/2] < a[k]$  или пока не выйдем в вершину сортирующего дерева.

```
template <class Item>
void fixUp(Item a[], int k)
{
    while (k > 1 && a[k/2] < a[k])
        { exch (a[k], a[k/2]); k = k/2; }
}
```

Если свойство пирамидальности сортирующего дерева было нарушено в силу того, что ключ какого-либо узла становится меньше, чем один или оба ключа его потомков, выполняются шаги с целью устранения нарушения путем замены узла на больший из его двух потомков. Такая замена может вызвать нарушение свойств сортирующего дерева на узле-потомке; это нарушение устраняется аналогичным путем и выполняется продвижение вниз по дереву до тех пор, пока не будет достигнут узел, оба потомка которого меньше его самого, либо нижний уровень дерева. Пример процесса показан на рис. 9.4. Опять-таки, программный код учитывает то обстоятельство, что потомки узла сортирующего дерева в позиции  $k$  занимают в нем позиции  $2k$  и  $2k+1$ .

Программа 9.4 содержит реализацию функции, которая восстанавливает сортирующее дерево после возможных нарушений по причине повышения приоритета заданного узла, перемещаясь вниз по этому дереву. Эта функция должна знать размер сортирующего дерева ( $N$ ), чтобы иметь возможность отследить момент, когда будет достигнут нижний уровень дерева.

Обе эти операции не зависят от способа представления структуры дерева, если возможен доступ к родителям (восходящий метод) и к потомкам (нисходящий метод) любого узла. В случае восходящего метода мы перемещаемся вверх по дереву, заменяя ключ заданного узла ключом его родителя до тех пор, пока не выйдем на корневой узел или на родителя с большим (или равным) ключом. В случае нисходящего

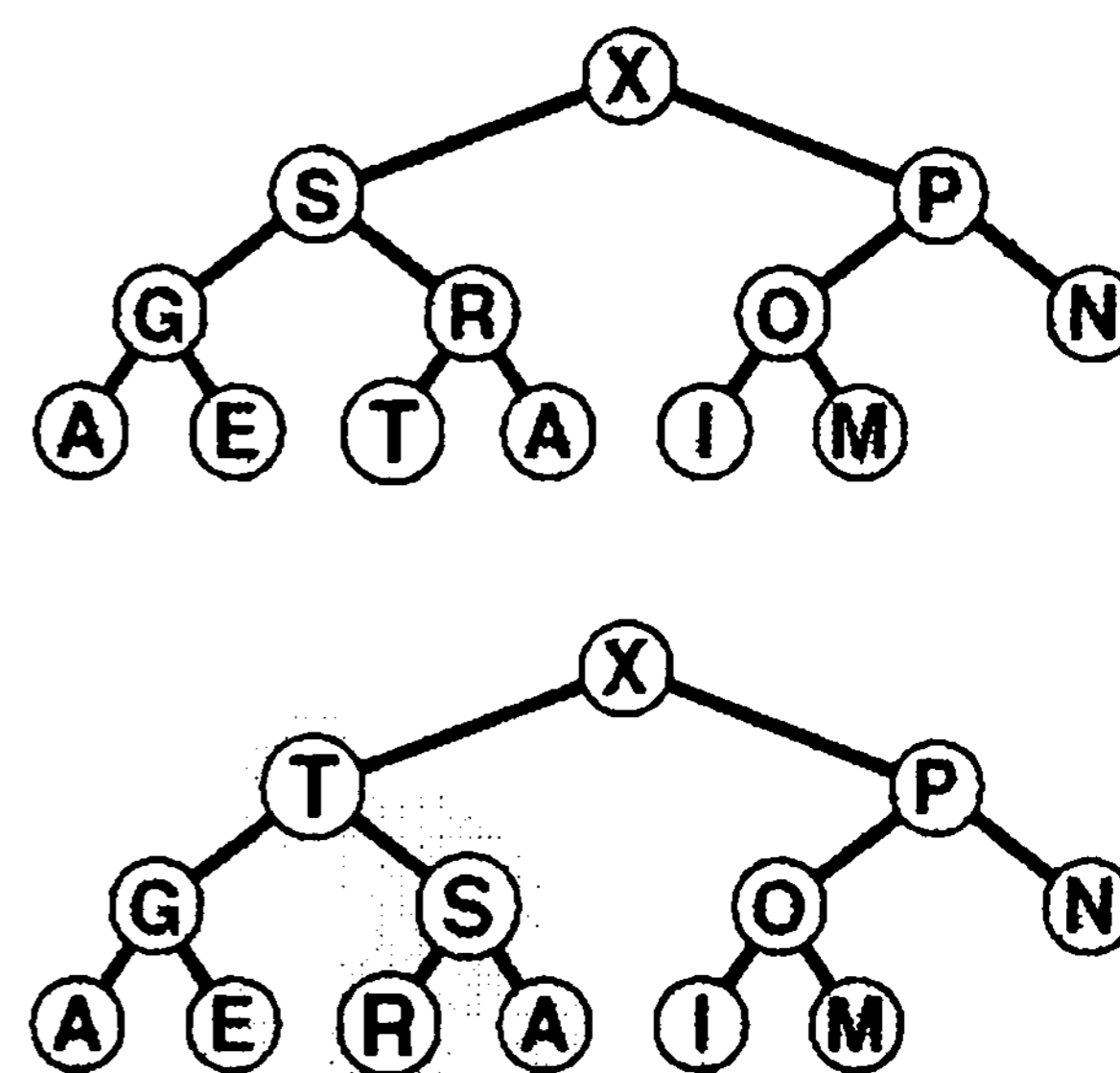


РИСУНОК 9.3. ВОСХОДЯЩАЯ УСТАНОВКА СТРУКТУРЫ СОРТИРУЮЩЕГО ДЕРЕВА

На верхнем дереве, показанном на диаграмме, установлен пирамидальный порядок, в который не вписывается узел  $T$  на нижнем уровне. Если мы поменяем  $T$  местами с его родителем, дерево остается пирамидально упорядоченным, за исключением разве что того случая, когда  $T$  оказывается больше своего нового родителя.

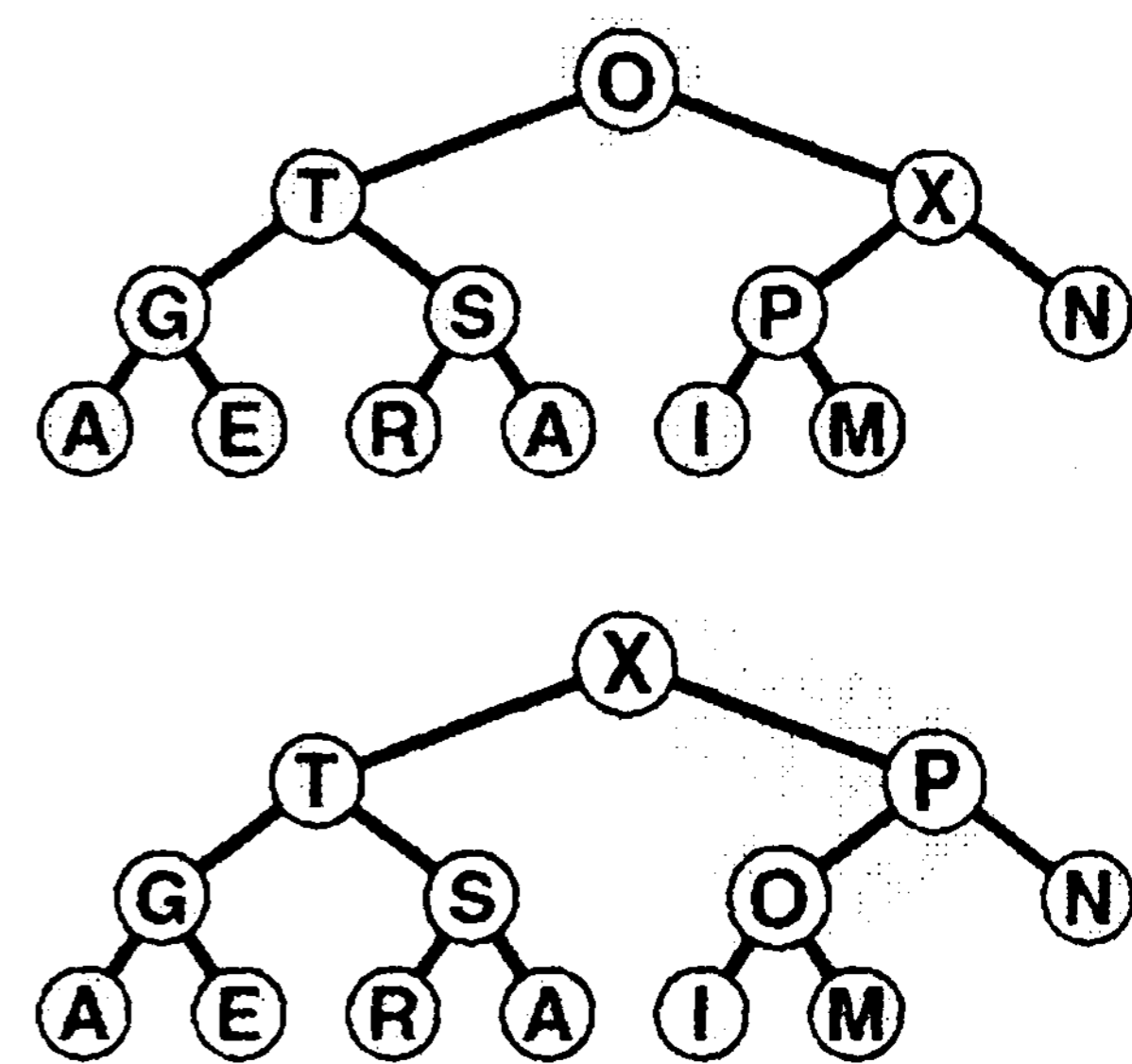
Продолжая обмен местами узла  $T$  со своими родителями до тех пор, пока мы не выйдем на своем пути на корневой узел, либо на узел, который больше  $T$ , мы можем установить пирамидальный порядок во всем дереве. Мы можем использовать эту процедуру в качестве основы для операции *insert* (вставить), выполняемой на сортирующих деревьях с целью восстановления пирамидального порядка после добавления в это дерево нового элемента (крайнюю правую позицию на нижнем уровне, вводя в дереве при необходимости новый уровень).

метода мы перемещаемся вниз по дереву, заменяя ключ заданного узла большим из ключей его потомков до тех пор, пока не достигнем нижнего уровня или точки, в которой нет потомков с большими ключами. В обобщенном виде упомянутые операции применимы не только к бинарным деревьям, но и к любой древовидной структуре. Усовершенствованные алгоритмы, манипулирующие очередями по приоритетам, обычно ориентированы на древовидные структуры общего вида, но в то же время полагаются именно на эти базовые операции, обеспечивающие доступ к наибольшим ключам этой структуры, сосредоточенным в ее верхней части.

Если вообразить себе, что некоторое сортирующее дерево представляет иерархию некой корпорации, в котором каждый потомок представляет подчиненное подразделение (и каждый родитель представляет собой вышестоящее подразделение), то эти операции допускают любопытные аналогии. Восходящий метод соответствует появлению на сцене нового перспективного управляющего, который продвигается по служебной лестнице с одного поста на другой, более высокий (обмениваясь служебными обязанностями с любым начальником, имеющим более низкую квалификацию) до тех пор, пока этот новый работник не столкнется с более квалифицированным начальником. Нисходящий метод аналогичен ситуации, когда президента некоторой компании сменяет на его посту некто, уступающий ему по квалификации. Если какой-либо из подчиненных президента, облеченный наибольшими полномочиями, превосходит по совокупности качеств нового работника, они обмениваются своими обязанностями, и мы движемся вниз по служебной лестнице, понижая в должности нового работника и повышая других, пока не будет достигнут уровень компетенции нового работника, когда не остается ни одного подчиненного, превосходящего нового работника по квалификации (этот идеализированный сценарий в реальной жизни встречается редко). Продолжая эту аналогию, движение по сортирующему дереву часто будет называться *продвижением*.

#### Программа 9.4. Нисходящая установка структуры сортирующего дерева

Чтобы восстановить пирамидальную структуру в случае, когда приоритет узла понижается, мы движемся вниз по сортирующему дереву, меняя при необходимости местами узел в позиции **k** с большим из его двух потомков, и останавливаемся, когда узел в позиции **k** не превышает какой-либо из двух своих потомков, или когда



**РИСУНОК 9.4. НИСХОДЯЩАЯ УСТАНОВКА СТРУКТУРЫ СОРТИРУЮЩЕГО ДЕРЕВА**

*Дерево, изображенное в верхней части диаграммы, почти везде пирамидально упорядочено, исключением является корень дерева. Если заменить узел **O** большим из его потомков (**X**), то рассматриваемое дерево приобретает пирамидальный порядок, за исключением поддерева с корнем в узле **O**. Продолжая обмен местами с большим из его двух потомков до тех пор, пока не будет достигнут нижний уровень пирамиды или точка, в которой **O** больше любого из своих потомков, можно восстановить условие пирамиды на всем дереве. Этой процедурой можно воспользоваться в качестве основы для операции **remove the maximum** (удалить наибольший) в сортирующем дереве с целью восстановить пирамидальный порядок после замены ключа в корне дерева на ключ, который находится на нижнем уровне в крайней правой позиции.*

достигнут нижний уровень. Обратите внимание на то обстоятельство, что если **N** есть четное число и **k** равно **N/2**, то узел в позиции **k** имеет только одного потомка — этот случай требует особого подхода!

Внутренний цикл в этой программе имеет два четко определенных выхода: один для случая, когда достигнут нижний уровень сортирующего дерева, а другой для случая, когда условия сортирующего дерева удовлетворяются где-то внутри дерева. Этот случай может служить прототипным примером необходимости существования конструкции **break**.

```
template <class Item>
void fixDown(Item a[ ], int k, int N)
{
    while (2*k <= N)
    { int j = 2*k;
      if (j < N && a[j] < a[j+1]) j++;
      if (!(a[k] < a[j])) break;
      exch (a[k], a[j]); k = j;
    }
}
```

### Программа 9.5. Очередь по приоритетам на базе сортирующего дерева

Чтобы реализовать операцию вставки, мы увеличиваем **N** на 1, добавляем новый элемент в конец сортирующего дерева, а затем при помощи функции **fixUp** восстанавливаем пирамидальный порядок. При выполнении функции **getmax** (получить наибольший) размер сортирующего дерева должен быть уменьшен на 1, таким образом, мы выбираем в качестве возвращаемого значения величину **pq[1]**, затем уменьшаем размер сортирующего дерева на 1, перемещая значение **pq[N]** в **pq[1]**, после чего выполняем функцию **fixDown** с целью восстановить в дереве пирамидальный порядок. Реализации конструктора и функции **empty** предельно тривиальны. Первая позиция **pq[0]** массива здесь не используется, но она может быть задействована в качестве сигнальной метки в других реализациях.

```
template <class Item>
class PQ
{
private:
    Item *pq;
    int N;
public:
    PQ(int maxN)
        {pq = new Item[maxN+1]; N = 0; }
    int empty( ) const
        {return N==0; }
    void insert (Item, item)
        {pq[++N] = item; fixUp(pq, N);}
    Item getmax()
        {
            exch(pq[1], pq[n] );
            fixDown(pq, 1, N-1 );
            return pq[N--];
        }
};
```

Как показывает программа 9.5, эти две основные операции позволяют построить эффективную реализацию АТД очереди по приоритетам. Если очередь по приоритетам представлена как пирамидально упорядоченный массив, использование операции

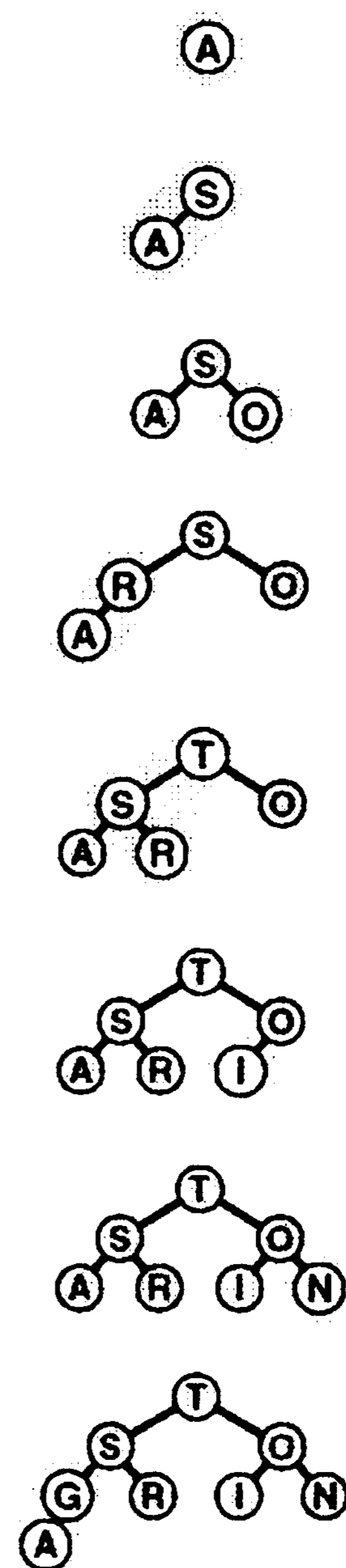


вставить (*insert*) равносильно добавлению нового элемента в конец массива и перемещение этого элемента по массиву с целью восстановления структуры сортирующего дерева; операция *удалить наибольший* (*remove the maximum*) равносильна удалению наибольшего элемента из вершины дерева с последующим размещением элемента, находящегося в конце дерева, в его вершине и перемещением его вниз вдоль массива с целью восстановления структуры сортирующего дерева.

**Лемма 9.2.** *Операции вставить (*insert*) и удалить наибольший (*remove the maximum*) для абстрактного типа данных могут быть реализованы на пирамидально упорядоченных деревьях таким образом, что операция вставить требует для своего выполнения на очереди, состоящей из  $N$  элементов, не более  $\lg N$  сравнений, а операция удалить наибольший — не более  $2 \lg N$  сравнений.*

Обе операции предусматривают перемещение вдоль пути между корнем и нижним уровнем дерева, при этом ни один путь вдоль сортирующего дерева, состоящего из  $N$  элементов, не содержит более  $\lg N$  элементов (см. например, лемму 5.8 и упражнение 5.77). Операция *удалить наибольший* требует двух сравнений для каждого узла: одну для того, чтобы найти потомка с большим ключом, другую — для того, чтобы принять решение, нужно ли продвигать этого потомка.

На рис. 9.5 и 9.6 показаны примеры, в рамках которых выполняется построение сортирующего дерева путем последовательной вставки элементов в первоначально пустое сортирующее дерево. В представлении сортирующего дерева в виде массива, которое использовалось ранее, этот процесс соответствует пирамидальному упорядочению массива, при этом каждый раз, когда мы переходим к новому элементу, размер сортирующего дерева увеличивается на 1, а для восстановления пирамидального порядка используется функция **fisUp**. В худшем случае на выполнение этого процесса требуется время, пропорциональное  $N \lg N$  (когда каждый новый элемент превосходит по величине все предыдущие, т.е. он проделывает весь путь к корню дерева), но в среднем на это требуется линейное время (новый элемент произвольной природы поднимается всего лишь на несколько уровней). В разделе 9.4 будет рассматриваться способ построения сортирующего дерева (установления в массиве пирамидального порядка) за линейное время в худшем случае.



**РИСУНОК 9.5. ПОСТРОЕНИЕ СОРТИРУЮЩЕГО ДЕРЕВА СВЕРХУ ВНИЗ**

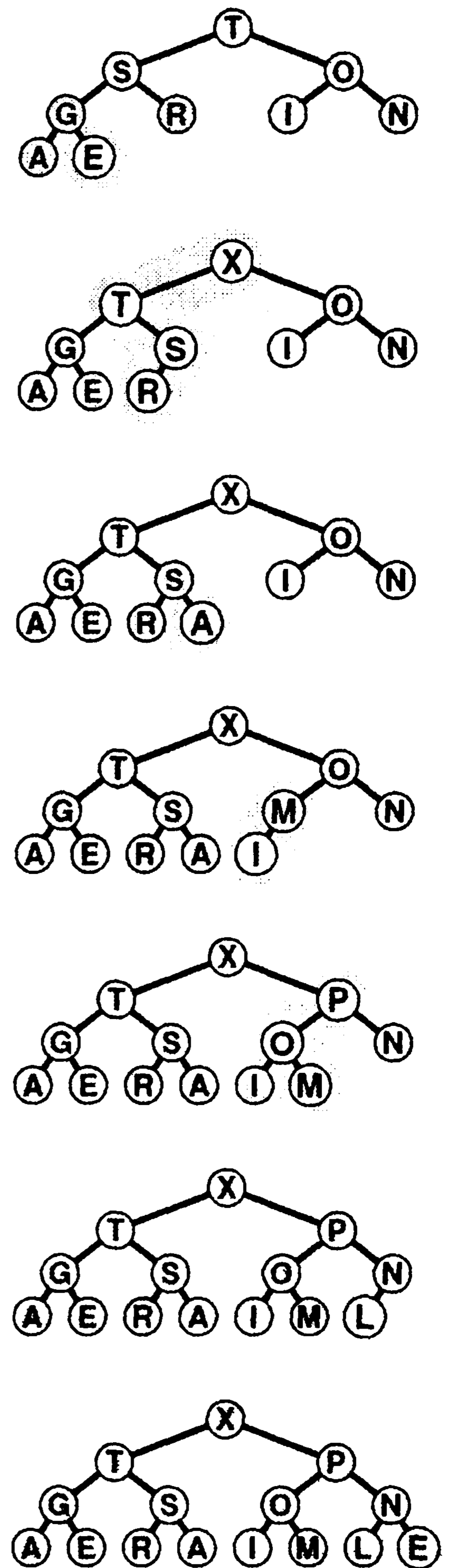
*Представленная последовательность диаграмм описывает операцию вставки ключей **A S O R T I N G** в первоначально пустое сортирующее дерево. Новые узлы добавляются в сортирующее дерево на нижнем уровне в направлении слева направо. Каждая вставка затрагивает только узлы, которые находятся на пути между точкой вставки и корнем, поэтому затраты в худшем случае пропорциональны логарифму размера сортирующего дерева.*

Базовые процедуры **fixUp** и **fixDown** из программ 9.3 и 9.4 позволяют получить прямую реализацию операций *изменить приоритет* (*change priority*) и *удалить* (*remove*). Для изменения приоритета элемента, находящегося где-то в середине сортирующего дерева, применяется процедура **fixUp** для перемещения вверх по дереву, если приоритет элемента увеличивается, и процедура **fixDown** для перемещения вниз по дереву, если приоритет уменьшается. Полная реализация этих процедур применительно к конкретным элементам данных имеет смысл, только если для каждого элемента имеется дескриптор, отслеживающий место этого элемента в структуре данных. Мы подробно рассмотрим реализации, которые выполняют эти действия, в разделах 9.5—9.7.

**Лемма 9.3.** *Операции изменить приоритет (change priority), удалить (remove) и изменить приоритет (change priority) для АТД очередь по приоритетам могут быть реализованы через сортирующие деревья, такие, что для любой из указанных выше операций требуется выполнение не более чем  $2 \lg N$  операций сравнения на очереди из  $N$  элементов.*

Поскольку эти операции требуют наличия специальных дескрипторов, отложим рассмотрение реализаций, поддерживающих эти операции, до раздела 9.7 (см. программу 9.12 и рис. 9.14). Все они предусматривают движение по сортирующему дереву вдоль одного пути, возможно, сверху вниз или снизу вверх в худшем случае.

Специально обращаем внимание на тот факт, что в этот список не включена операция *объединить* (*join*). Эффективное объединение двух очередей по приоритетам, по-видимому, потребует более сложной структуры данных, которая будет подробно рассматриваться в разделе 9.7. С другой стороны, представленный здесь простой метод, в основу которого положен пирамидальный порядок, вполне достаточен для большинства различных приложений. Он использует минимальное дополнительное пространство памяти и обеспечивает высокую эффективность выполнения операций за исключением тех случаев, когда часто и на больших объемах данных выполняются операции *объединить*.



**РИСУНОК 9.6. ПОСТРОЕНИЕ СОРТИРУЮЩЕГО ДЕРЕВА СВЕРХУ ВНИЗ (ПРОДОЛЖЕНИЕ)**

Представленная последовательность диаграмм отображает процедуру вставки ключей *EXAMPLE* в сортирующее дерево, построение которого было начато на рис. 9.5. Общая стоимость построения сортирующего дерева размером  $N$  составляет  $\lg 1 + \lg 2 + \dots + \lg N$ , что меньше  $N \lg N$ .

Как уже отмечалось выше и как продемонстрировано в программе 9.6, любую очередь по приоритетам можно использовать для построения еще одного метода сортировки. Мы просто вставляем все подлежащие сортировке ключи в очередь по приоритетам, а затем многократно используем операцию *удалить наибольший*, чтобы удалять их в порядке убывания приоритетов. Такое использование очереди по приоритетам, представленной в виде неупорядоченного списка, соответствует выполнению сортировки выбором, а применение упорядоченного списка соответствует выполнению сортировки вставками.

### Программа 9.6. Сортировка с использованием очереди по приоритетам

Чтобы отсортировать подмассив  $a[1], \dots, a[r]$ , используя для этой цели АТД *очередь по приоритетам*, следует при помощи функции `insert` (вставить) поместить элементы в очередь по приоритетам, а затем с использованием функции `getmax` (найти наибольший) удалять их в порядке убывания значений приоритетов. Подобный алгоритм сортировки выполняется за время, пропорциональное  $N \lg N$ , но при этом он требует дополнительного объема памяти, пропорционального количеству сортируемых элементов  $N$  (в очереди по приоритетам).

```
#include "PQ.cxx"
template <class Item>
void PQsort(Item a[], int l, int r)
{ int k;
  PQ<Item> pq(r-l+1);
  for (k = l; k <= r; k++) pq.insert(a[k]);
  for (k = r; k >= l; k--)
    a[k] = pq.getmax();
}
```

На рис. 9.5 и 9.6. показан пример первой фазы (процесс построения), на которой используется реализация очереди по приоритетам с пирамидальным порядком; на рис. 9.7 и 9.8 представлена вторая фаза (которую будем называть *нисходящей сортировкой* — `sortdown`). В условиях практических применений этот метод не выглядит особенно элегантно, поскольку он без особой необходимости копирует сортируемые элементы (в очереди по приоритетам). Действительно, выполнение  $N$  последовательных вставок — не самый эффективный способ построения сортирующего дерева, состоящего из  $N$  элементов. В следующем разделе при обсуждении реализации классического алгоритма пирамидальной сортировки этим двум вопросам уделяется особое внимание.

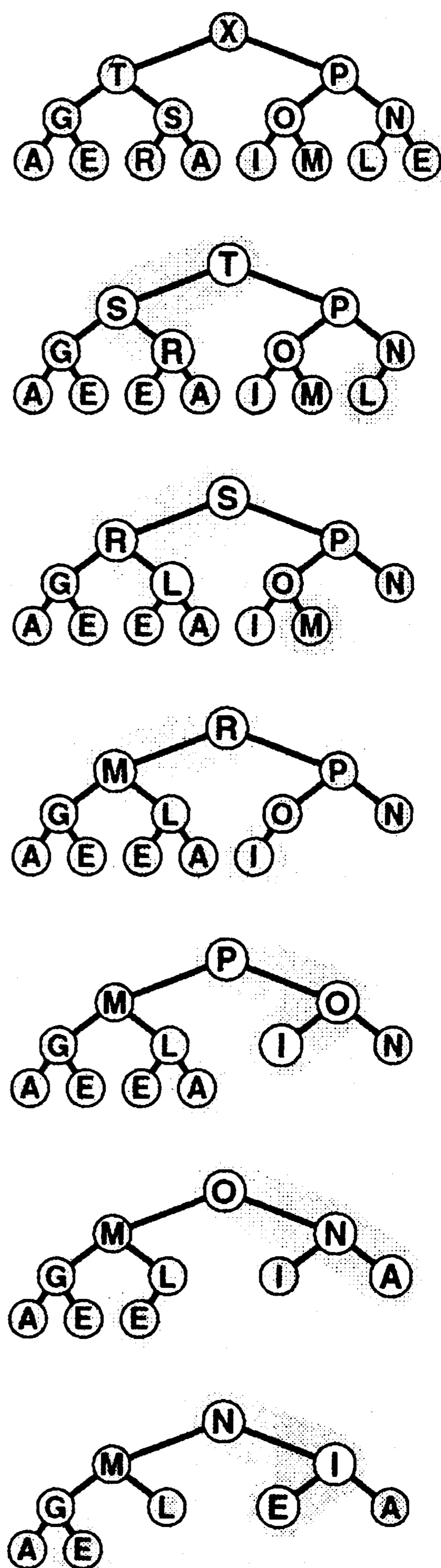


РИСУНОК 9.7. СОРТИРОВКА В СОРТИРУЮЩЕМ ДЕРЕВЕ

После замены наибольшего элемента сортирующего дерева самым правым элементом нижнего уровня можно восстановить пирамидальный порядок за счет перемещения по пути из корня на нижний уровень дерева.

## Упражнения

▷ 9.21. Построить сортирующее дерево, которое получается после того как ключи **E A S Y Q U E S T I O N** вставлены в первоначально пустое сортирующее дерево.

▷ 9.22. Воспользовавшись соглашением, принятым в упражнении 9.1, представьте последовательность сортирующих деревьев, полученных в результате выполнения операций

**PRIO\*R\*\*I\*T\*Y\*\*\*QUE\*\*\*U\*E**

на первоначально пустом сортирующем дереве.

9.23. Поскольку примитив **exch** используется в операциях по установке пирамидального порядка, элементы загружаются и запоминаются в два раза чаще, чем это необходимо. Предложите более эффективные реализации, которые не сталкиваются с такой проблемой, в духе сортировки вставками.

9.24. Почему мы не пользуемся сигнальной меткой, чтобы избежать проверку  $j < N$  в функции **fixDown**?

○ 9.25. Добавить операцию *заменить наибольший* (*replace the maximum*) в реализацию очереди по приоритетам с пирамидальным порядком в программе 9.5. Рассмотреть случай, когда добавляемое значение больше всех остальных значений в очереди. Совет: К элегантному решению приводит использование элемента **pq[0]**.

9.26. Каким является минимальное количество перемещений ключей, которое потребуется выполнить на сортирующем дереве в операции *удалить наибольший* (*remove maximum*)? Приведите пример сортирующего дерева, содержащего 15 элементов, для которого достигается этот минимум.

9.27. Каким является минимальное количество ключей, которое потребуется переместить в процессе выполнения на сортирующем дереве трех последовательных операций *удалить наибольший*? Приведите пример сортирующего дерева из 15 элементов, для которого достигается этот минимум.

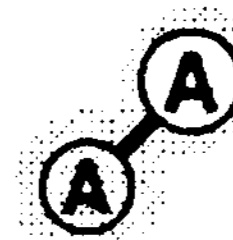
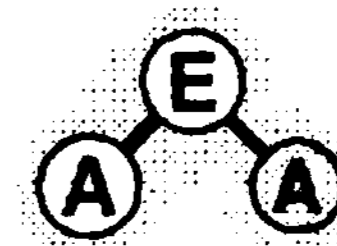
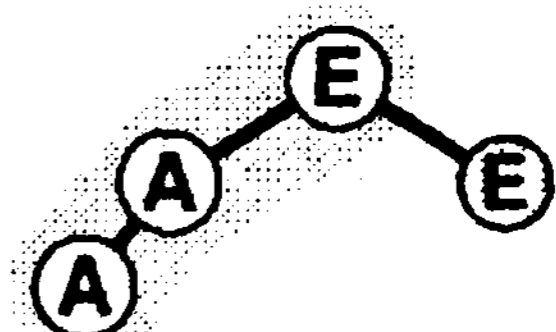
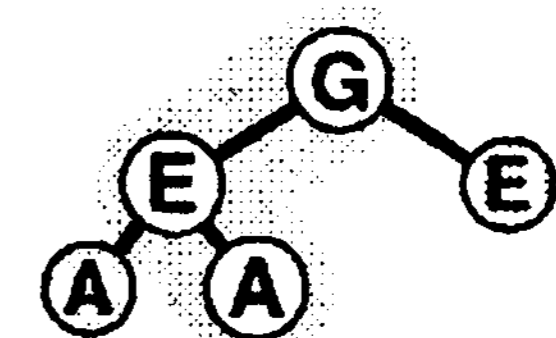
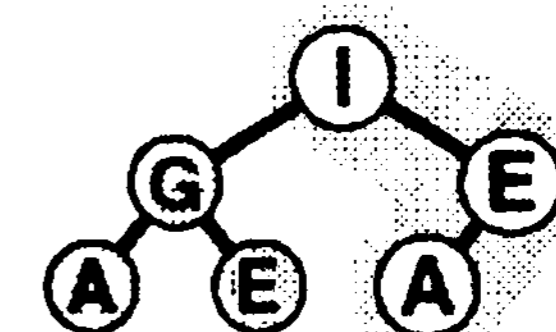
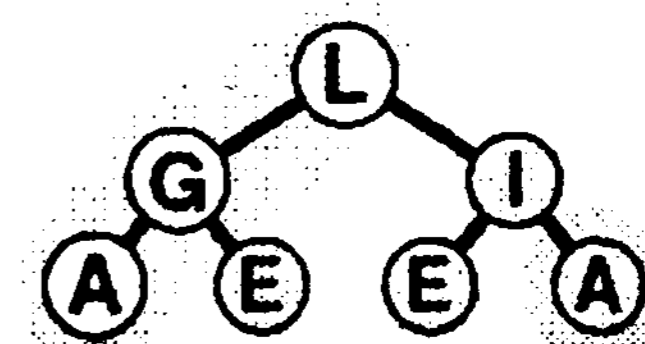
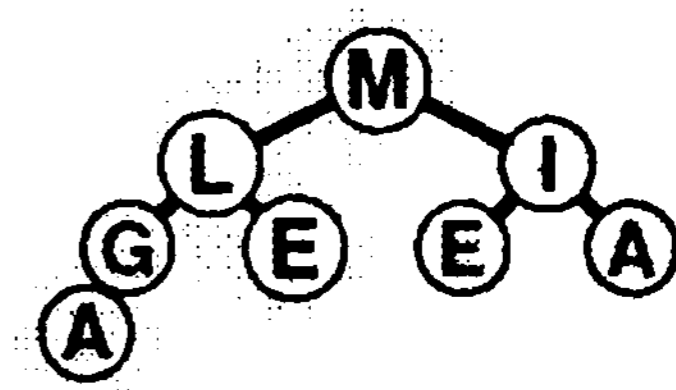


РИСУНОК 9.8. СОРТИРОВКА ИЗ СОРТИРУЮЩЕГО ДЕРЕВА

Представленная здесь последовательность диаграмм отображает удаление остальных ключей из сортирующего дерева на рис. 9.7. Даже если каждый элемент возвращается на нижний уровень, то общие затраты на выполнение этой фазы сортировки не больше  $\lg N + \dots + \lg 2 + \lg 1$ , что меньше  $N \lg N$ .

## 9.4. Пирамидальная сортировка

Основную идею, положенную в основу программы 9.6, можно приспособить с таким расчетом, чтобы сортировка массива выполнялась без необходимости использования какого бы то ни было дополнительного пространства памяти. Другими словами, сосредоточившись на задаче сортировки, мы отказываемся от идеи сокрытия очереди по приоритетам, и вместо того чтобы придерживаться ограничений, налагаемых интерфейсом АТД *очереди по приоритетам*, будем непосредственно применять функции **fixUp** и **fixDown**.

Используя программу 9.5 в программе 9.6 непосредственно для перемещения по массиву слева направо, применим функцию **fixUp** с тем, чтобы элементы, располагающиеся слева от указателя просмотра, образовывали пирамидально упорядоченное полное дерево. Затем, во время выполнения процесса нисходящей сортировки мы помещаем наибольший элемент на то место, которое освобождается по мере уменьшения размеров сортирующего дерева. Иначе говоря, процесс нисходящей сортировки подобен сортировке выбором, однако при этом он пользуется более эффективным методом обнаружения наибольшего элемента в неотсортированной части массива.

Более эффективным, нежели построение сортирующего дерева за счет последовательного выполнения вставок, как показано на рис. 9.5 и 9.6, является построение такого дерева методом прохождения по этому дереву в обратном направлении, формируя поддеревья меньших размеров снизу вверх, что иллюстрирует рис. 9.9. Другими словами, мы рассматриваем каждую позицию массива как корень небольшого поддерева и извлекаем пользу из того обстоятельства, что функция **fixDown** работает на таких сортирующих поддеревьях столь же хорошо, как и на большом дереве. Если оба потомка узла суть сортирующие деревья, то вызов для этого узла функции **fixDown** приводит к тому, что поддерево с корнем в этом узле также становится сортирующим деревом. Продвигаясь по дереву в обратном направлении и вызывая **fixDown** в каждом узле, мы по индукции можем восстановить пирамидальный порядок. Просмотр начинается на полпути в обратном направлении вдоль массива, поскольку можно пропустить поддерево размером 1.

Полная реализация классического алгоритма пирамидальной сортировки (heapsort) представлена в программе

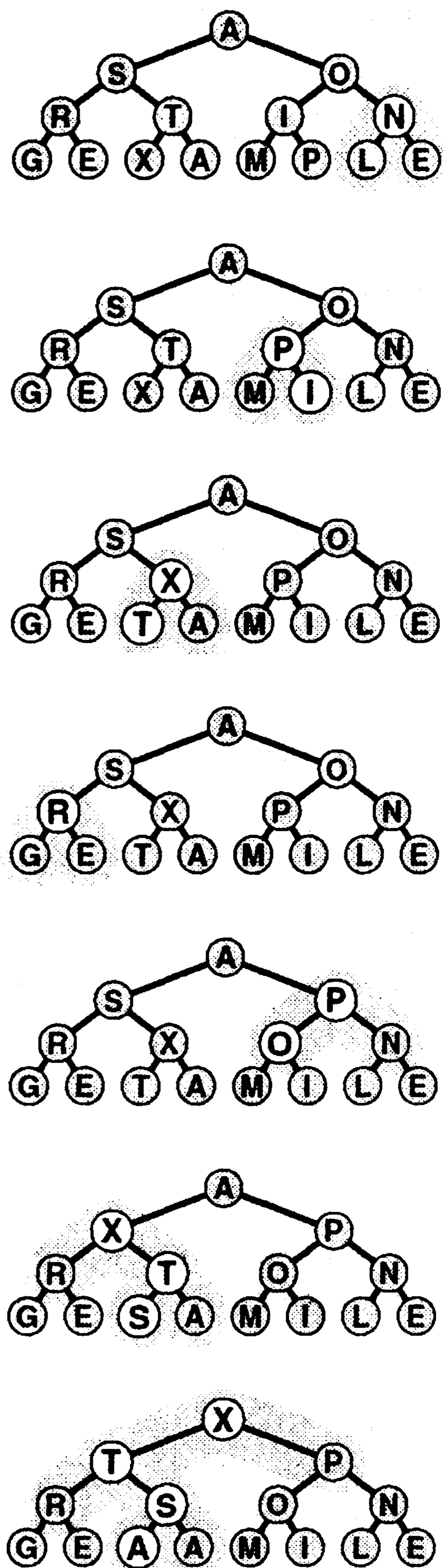


РИСУНОК 9.9. ПОСТРОЕНИЕ СОРТИРУЮЩЕГО ДЕРЕВА СНИЗУ ВВЕРХ

*Продвигаясь справа налево и снизу вверх мы строим сортирующее дерево, следя за тем, чтобы поддерево под текущим узлом было пирамидально упорядочено. Общие затраты в наихудшем случае линейно зависимы, поскольку большая часть узлов расположена близко к нижнему уровню дерева.*

9.7. И хотя циклы в этой программе на первый взгляд решают совершенно разные задачи (первый выполняет построение сортирующего дерева, второй разрушает это сортирующее дерево для процесса нисходящей сортировки), они построены на основании одной и той же базовой процедуры, которая восстанавливает порядок в дереве, на котором, возможно, уже установлен пирамидальный порядок, за исключением разве что самого корня, используя при этом представление полного дерева в виде массива. На рис. 9.10 показано содержимое массива в примере, соответствующем рис. 9.7—9.9.

**Лемма 9.4.** *Для построения сортирующего дерева снизу вверх требуется линейно зависимое время.*

В основе этого утверждения лежит тот факт, что большинство подвергаемых обработке деревьев имеют небольшие размеры. Например, чтобы построить сортирующее дерево из 127 элементов, мы выполняем построение 32 сортирующих деревьев размером 3, 16 деревьев размером 7, 8 деревьев размером 15, 4 деревьев размером 31, двух сортирующих деревьев размером 63 и одного дерева размером 127, так что в худшем случае требуется  $32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$  продвижений (в два раза больше числа операций сравнения). Для  $N = 2^n - 1$  верхняя граница числа продвижений равна

$$\sum_{1 \leq k < n} k 2^{n-k-1} = 2^n - n - 1 < N.$$

Это же доказательство справедливо и в случае, когда  $N + 1$  не является степенью 2.

Эта лемма не имеет особого значения для пирамидальной сортировки, поскольку основная доля времени ее выполнения все еще приходится на  $N \log N$  — время, затрачиваемое на выполнение нисходящей сортировки, однако оно играет важную роль в тех приложениях очередей по приоритетам, в которых операция *создать* (*construct*) приводит к алгоритму, обеспечивающему линейную зависимость времени выполнения. Как отмечается на рис. 9.6, построение сортирующего дерева с  $N$  последовательно выполняемых операций *вставить* требует в совокупности  $N \log N$  шагов в наихудшем случае (даже если это общее количество шагов оказывается в среднем линейным для файлов с произвольной организацией).

```
A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A S O R T P N G E X A M I L E
A S O R X P N G E T A M I L E
A S O R X P N G E T A M I L E
A S P R X O N G E T A M I L E
A X P R T O N G E S A M I L E
X T P R S O N G E A A M I L E
```

```
T S P R E O N G E A A M I L X
S R P L E O N G E A A M I T X
R L P I E O N G E A A M S T X
P L O I E M N G E A A R S T X
O L N I E M A G E A P R S T X
N L M I E A A G E O P R S T X
M L E I E A A G N O P R S T X
L I E G E A A M N O P R S T X
I G E A E A L M N O P R S T X
G E E A A I L M N O P R S T X
E A E A G I L M N O P R S T X
E A A E G I L M N O P R S T X
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

#### РИСУНОК 9.10. ПРИМЕР ПИРАМИДАЛЬНОЙ СОРТИРОВКИ

*Пирамидальная сортировка представляет собой эффективный алгоритм сортировки, основанный на методе выбора. Сначала строится сортирующее дерево сверху вниз без использования вспомогательной памяти. Верхние восемь строк диаграммы соответствуют рис. 9.9. Далее, из дерева многократно удаляется наибольший элемент. Незаштрихованная часть строк нижней диаграммы соответствуют рис. 9.7 и 9.8; заштрихованная часть содержит отсортированный по возрастанию файл.*

---

**Программа 9.7. Пирамидальная сортировка**


---

Непосредственное применение функции **fixDown** позволяет построить классический алгоритм пирамидальной сортировки. Цикл **for** выполняет построение сортирующего дерева; далее, цикл **while** меняет местами наибольший элемент с последним элементом массива и восстанавливает свойства сортирующего дерева, продолжая этот процесс до тех пор, пока сортирующее дерево не станет пустым. Тот факт, что в условиях представления полного дерева в виде массива указатель **pq** указывает на **a[l-1]**, позволяет программе рассматривать переданный ей подфайл как первый элемент с индексом 1 (см. рис. 9.2). В некоторых средах программирования это невозможно.

```
template <class Item>
void heapsort(Item a[ ], int l, int r)
{ int k, N = r-l+1;
  Item *pq = a+l-1;
  for (k = N/2; k >= 1; k--)
    fixDown (pq, k, N);
  while (N>1)
    {exch(pq[1], pq[N]);
     fixDown (pq, 1, --N); }
}
```

---

**Лемма 9.5.** *Пирамидальная сортировка использует менее  $2N \lg N$  сравнений для сортировки  $N$  элементов.*

Несколько более высокая граница  $3N \lg N$  следует непосредственно из леммы 9.2. Предложенная здесь граница следует из более точного подсчета на основе леммы 9.4.

Лемма 9.5 и возможность выполнения без использования вспомогательной памяти — вот два основных фактора, которые обуславливают интерес, проявляемый к пирамидальной сортировке на практике, ибо они *гарантируют*, что сортировка  $N$  элементов будет выполняться за время, пропорциональное  $N \log N$  независимо от природы входного потока данных. В подобных условиях не бывает входных данных, вызывающих возникновение наихудшего случая, который существенно замедляет выполнение сортировки (в отличие от быстрой сортировки), а пирамидальная сортировка вообще не использует дополнительное пространство памяти (в отличие от сортировки слиянием). Достижение такой гарантированной эффективности для худшего случая требует уплаты своей цены: например, внутренний цикл рассматриваемого алгоритма (стоимость выражается количеством операций сравнения) выполняет больше базовых операций, чем внутренний цикл быстрой сортировки, таким образом, пирамидальная сортировка, по-видимому, работает медленнее быстрой сортировки как на обычных файлах, так и на файлах с произвольной организацией.

Сортирующие деревья можно успешно использовать для решения проблемы выборки  $k$  максимальных элементов из  $N$  элементов (см. главу 7), особенно в случаях, когда  $k$  мало. Мы просто прекращаем выполнение алгоритма пирамидальной сортировки после того, как  $k$  элементов будут отобраны из вершины сортирующего дерева.

**Лемма 9.6.** Выборка на базе пирамидальной сортировки позволяет отыскать  $k$ -й наибольший элемент из  $N$  элементов за время, пропорциональное  $N$ , когда  $k$  мало или близко по величине к  $N$ , либо за время, пропорциональное  $N \log N$ , во всех других случаях.

Один метод заключается в том, чтобы построить сортирующее дерево с использованием числа операций сравнения меньшего  $2N$  (см. лемму 9.4), с последующим удалением  $k$  наибольших элементов, используя для этой цели  $2k \lg N$  или меньшее количество операций сравнения (см. лемму 9.4), при общем числе сравнений, равном  $2N + 2k \lg N$ . Другой метод связан с построением минимально ориентированного сортирующего дерева размером  $k$  с последующим выполнением операции *заменить наименьший* (*replace the minimum*) (операция *вставить* плюс операция *удалить наименьший*) на оставшихся элементах, при общем количестве операций сравнения, не превосходящем  $2k + 2(N - k) \lg k$  (см. упражнение 9.36). Этот метод использует дополнительную память, пропорциональную  $k$ , и является особо привлекательным для нахождения  $k$  наибольших из  $N$  элементов в условиях, когда  $k$  принимает малое, а  $N$  — большое значение (или ничего не известно заранее). Что касается случайных ключей или других типичных ситуаций верхняя граница  $\lg k$  для операций на сортирующем дереве во втором методе, по-видимому, есть  $O(1)$ , когда  $k$  мало по сравнению с  $N$  (см. упражнение 9.36).

Исследованы другие способы дальнейшего улучшения пирамидальной сортировки. Одна из идей, развитая Флойдом, состоит в использовании того обстоятельства, что элемент, повторно вставляемый в процессе нисходящей сортировки, проделывает весь путь на нижний уровень, так что мы можем сэкономить время, затрачиваемое на проверку достижения таким элементом своей позиции, просто продвигая больший из двух потомков до тех пор, пока не будет достигнут нижний уровень с последующим продвижением вверх по сортирующему дереву в соответствующую позицию. Благодаря этой идее коэффициент сокращения числа выполняемых операций сравнения асимптотически стремится к 2 и принимает значение, близкое к  $\lg N! \approx N \lg N - N / \ln 2$ , что является абсолютным минимумом количества сравнений для любого алгоритма сортировки (см. часть 8). Этот метод требует дополнительных вычислений и может оказаться полезным на практике только в ситуациях, когда расходы на операции сравнения сравнительно велики (например, при сортировке записей со строковыми ключами или другими видами длинных ключей).

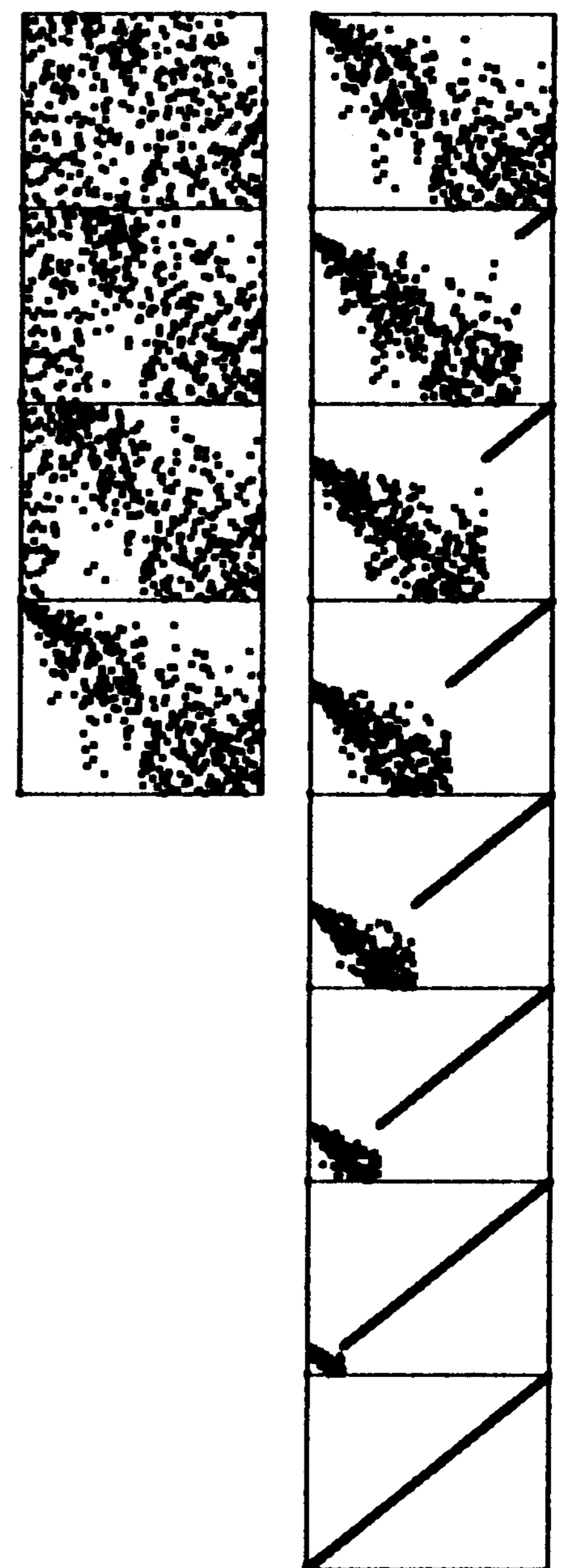
Другая идея заключается в том, чтобы построить сортирующие деревья, опираясь на представление полных пирамидально упорядоченных *троичных* деревьев в виде массивов, при этом узел в позиции  $k$  больше или равен узлам в позициях  $3k - 1$ ,  $3k$  и  $3k + 1$  и меньше или равен узлам в позициях  $\lfloor (k + 1) / 3 \rfloor$  для всех позиций между 1 и  $N$  в массиве из  $N$  элементов. Снижение стоимости, обусловленное меньшей высотой дерева, уравновешивается более высокой стоимостью выбора наибольшего из трех потомков в каждом узле. Подобного рода компромисс зависит от деталей реализации (см. упражнение 9.30). Дальнейшее увеличение количества потомков в каждом узле по всем признакам не дает никакой выгоды.

Рисунок 9.11 иллюстрирует пирамидальную сортировку в действии на файле с произвольной организацией. Поначалу кажется, что этот процесс делает все, что угодно, только не сортировку, поскольку по мере продвижения построения сортирующего



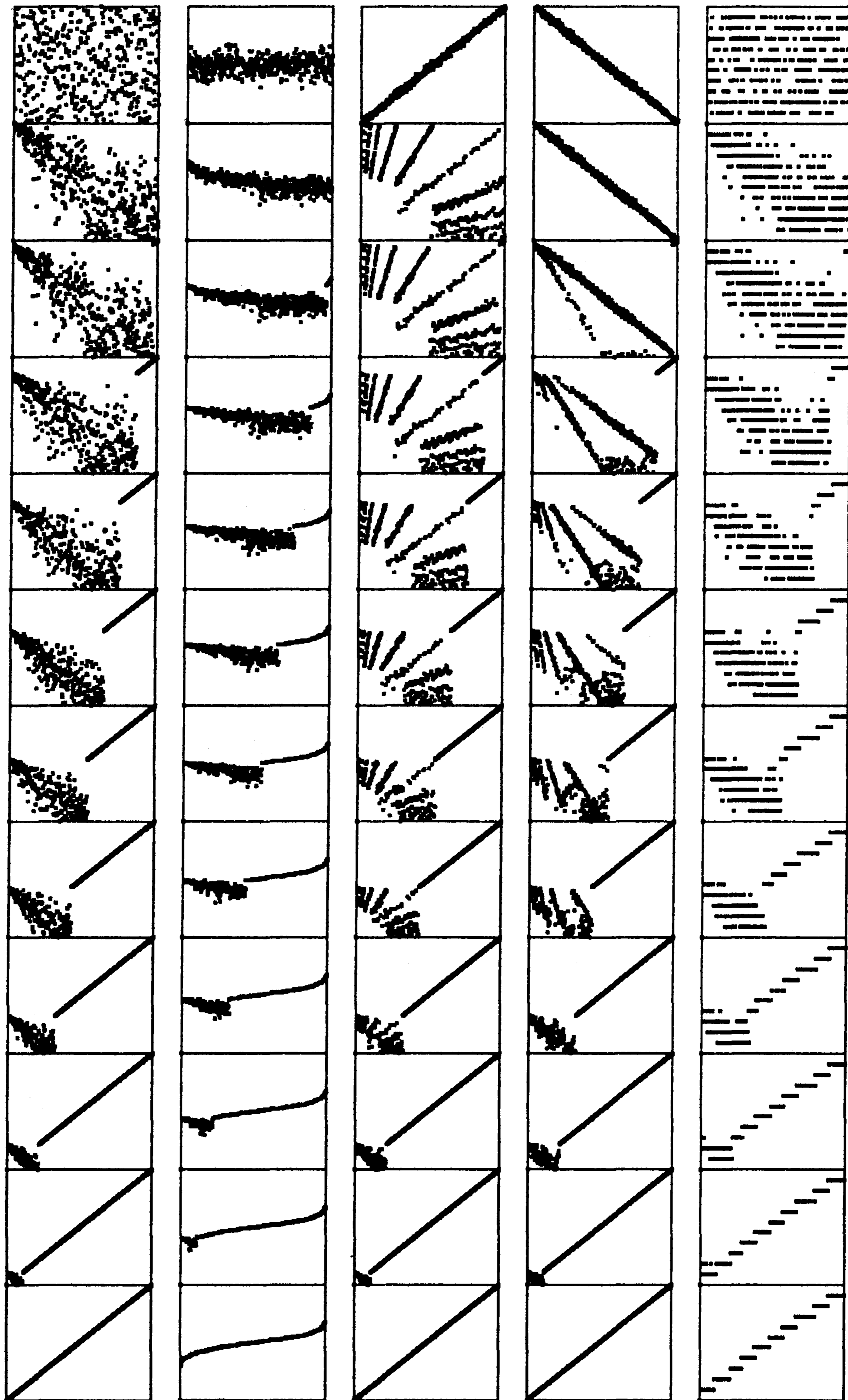
дерева большие элементы перемещаются в начало файла. Однако, в соответствии с ожиданиями, далее метод больше выглядит как зеркальное отображение сортировки выбором. На рис. 9.12 показано, что различные виды данных можно представить в виде сортирующих деревьев, обладающими особыми характеристиками, однако по мере продвижения процесса сортировки к своему завершению все они приобретают вид сортирующего дерева с произвольной организацией.

Естественно, не может не интересовать проблема, какой метод сортировки выбрать для конкретного приложения: пирамидальную сортировку, быструю сортировку или сортировку слиянием. Выбор между пирамидальной сортировкой и сортировкой слиянием сводится к выбору между неустойчивой сортировкой (см. упражнение 9.28) и сортировкой, которая использует дополнительный объем памяти; выбор между пирамидальной сортировкой и быстрой сортировкой сводится к выбору между средним быстродействием сортировки и быстродействием, обеспечиваемым в наихудшем случае. В свое время мы достаточно хорошо потрудились над совершенствованием внутренних циклов быстрой сортировки и сортировки слиянием; решение этих вопросов применительно к сортирующим деревьям мы предоставляем в упражнениях, сопровождающих настоящую главу. Речь отнюдь не идет о повышении производительности пирамидальной сортировки до уровня, превосходящего быструю сортировку, что, собственно говоря, показано посредством эмпирических данных, приведенных в табл. 9.2. Тем не менее, специалисты, изучающие проблему быстрой сортировки на собственных машинах, найдут эти данные поучительными. Как обычно, различные характерные особенности конкретных машин и систем программирования могут сыграть важную роль. Например, быстрая сортировка и сортировка слиянием обладают свойством локальности, которое дает им определенные преимущества на некоторых типах вычислительных машин. Когда операции сравнения становятся недопустимо дорогостоящими, версия Флойда подходит лучше других, поскольку в такого рода ситуациях она становится чуть ли не оптимальной, если принимать во внимание только время выполнения и стоимость используемого пространства памяти.



**РИСУНОК 9.11.**  
**ДИНАМИЧЕСКИЕ**  
**ХАРАКТЕРИСТИКИ**  
**ПИРАМИДАЛЬНОЙ**  
**СОРТИРОВКИ**

*На первый взгляд кажется, что процесс построения (слева) нарушает ранее установленный порядок на файле за счет того, что элементы с большими значениями помещаются в начало файла. Затем процесс нисходящей сортировки (справа) начинает действовать как сортировка выбором, при этом сортируемое дерево находится в начальной части файла, а построение отсортированного массива выполняется в конце файла.*



**РИСУНОК 9.12. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ ПИРАМИДАЛЬНОЙ СОРТИРОВКИ РАЗЛИЧНЫХ ТИПОВ ФАЙЛОВ.**

*Время выполнения пирамидальной сортировки не слишком чувствительно к природе входных данных. Независимо от того, какие значения принимают входные величины, наибольший элемент обнаруживается за число шагов, меньшее  $\lg N$ . На диаграммах показаны: файлы с произвольной организацией, файлы с распределением Гаусса, почти упорядоченные файлы, почти обратно упорядоченные файлы и файлы с произвольной организацией, обладающие 10 различными ключами (вверху, слева направо). Вторая диаграмма сверху демонстрирует сортирующее дерево, построенное с помощью восходящего алгоритма, остальные диаграммы отображают для каждого файла процесс нисходящей сортировки. Вначале сортирующие деревья отображают в какой-то степени исходный файл, но по мере продвижения процесса сортировки они все больше напоминают сортирующие деревья, полученные для файла с произвольной организацией.*

Таблица 9.2. Эмпирические исследования алгоритмов пирамидальной сортировки

Значения относительного времени выполнения различных видов сортировки на файлах случайных целых чисел подтверждают наши предположения, основанные на оценке длины внутренних циклов, что пирамидальная сортировка обладает меньшим быстродействием, чем быстрая сортировка, но сопоставима по этому параметру с сортировкой слиянием. Временные показатели для первых  $N$  страниц книги *Moby Dick* ("Моби Дик") в правой части таблицы показывают, что метод Флойда является эффективным усовершенствованием пирамидальной сортировки в тех случаях, когда стоимость операций сравнения достаточно высока.

$N$	32-битовые целые ключи					строковые ключи		
	Q	M	PQ	H	F	Q	H	F
12500	2	5	4	3	4	8	11	8
25000	7	11	9	8	8	16	25	20
50000	13	24	22	18	19	36	60	49
100000	27	52	47	42	46	88	143	116
200000	58	111	106	100	107			
400000	122	238	245	232	246			
800000	261	520	643	542	566			

**Обозначения:**

- Q** Быстрая сортировка, стандартная реализация (программа 7.1)
- M** Сортировка слиянием, стандартная реализация (программа 8.1)
- PQ** Очередь по приоритетам на базе пирамидальной сортировки (программа 9.5)
- H** Пирамидальная сортировка, стандартная реализация (программа 9.6)
- F** Пирамидальная сортировка с усовершенствованием Флойда

**Упражнения**

- 9.28.** Показать, что пирамидальная сортировка не является устойчивой.
- **9.29.** Определить эмпирическим путем процентное отношение времени, затрачиваемого на фазу построения для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
  - **9.30.** Реализовать версию пирамидальной сортировки, основанную на полных пирамидально упорядоченных сортирующих деревьях, соответствующих описанному в тексте. Сравнить количество использованных созданной программой операций сравнения, полученное эмпирическим путем, с аналогичным показателем стандартной реализации для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
  - **9.31.** Продолжая упражнение 9.30, определить эмпирическим путем, будет ли метод Флойда эффективным применительно к троичным деревьям.
  - **9.32.** Имея в виду только стоимость операций сравнения и предполагая, что для нахождения наибольшего из  $t$  элементов требуется  $t$  операций сравнения, найти значение  $t$ , которое сводит к минимуму коэффициент  $N \log N$ , присутствующий при подсчете операций сравнения, в том случае, когда в пирамидальной сортировке используется  $t$ -арное сортирующее дерево. Сначала воспользуемся прямым обобщением программы 9.7, затем предположим, что метод Флойда позволяет сэкономить одно сравнение во внутреннем цикле.

- 9.33. Для  $N = 32$  найти такое расположение ключей, которое требует выполнения максимального числа операций сравнения в рамках пирамидальной сортировки.
- 9.34. Для  $N = 32$  найти такое расположение ключей, которое требует выполнения минимального числа операций сравнения в рамках пирамидальной сортировки.
- 9.35. Показать, что построение очереди по приоритетам размером  $k$  с последующим выполнением  $N - k$  операций *заменить наименьший* (*replace the minimum*) (операция *вставить* плюс операция *удалить наименьший*), оставляет в сортирующем дереве  $k$  наибольших элементов из числа  $N$  элементов.
- 9.36. Реализовать обе версии выборки на базе пирамидальной сортировки, о которых шла речь при обсуждении леммы 9.6, воспользовавшись методами, описанными в упражнении 9.25. Сравните определяемое эмпирическим путем количество операций сравнения, которые они используют, с аналогичным показателем метода, задействующего быструю сортировку, описанного в разделе главы 7, для  $N = 10^6$  и при  $k = 10, 100, 1000, 10^4, 10^5$  и  $10^6$ .
- 9.37. Реализовать версию пирамидальной сортировки, в основу которой положена идея представления пирамидально упорядоченного дерева в прямом порядке (*preorder*), а не по уровням. Проведите эмпирическое сравнение количества операций сравнения, используемых этой версией, с количеством сравнений в стандартной реализации для ключей с произвольной организацией, причем  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

## 9.5. Абстрактный тип данных очереди по приоритетам

В большинстве случаев требуется написать приложение так, чтобы оно содержало процедуру реализации очереди по приоритетам вместо того, чтобы возвращать значения из операции *удалить наибольший* (*remove the maximum*), уведомлять, ~~какая~~ из записей обладает наибольшим приоритетом, и подобным же образом поступать по отношению к другим операциям. Другими словами, устанавливаются приоритеты, а очереди по приоритетам применяются только с целью доступа к другой информации в соответствующем порядке. Подобного рода организация аналогична использованию понятий *непрямой сортировки* (*indirect-sort*) и *сортировки по указателю* (*pointer-sort*), описанных в главе 6. В частности, такой подход нужен для придания смысла таким операциям, как *изменить приоритет* (*change priority*) или *удалить* (*remove*). Здесь мы подробно исследуем реализацию упомянутой идеи не только в силу того, что в дальнейшем будем использовать очередь по приоритетам, но еще и потому, что такая ситуация характерна для проблем, с которыми приходится сталкиваться при разработке интерфейсов и реализаций для абстрактных типов данных (АТД).

Когда мы хотим удалить элемент из очереди по приоритетам, как правильно указать, какой конкретно элемент надо удалить? Когда мы хотим поддерживать сразу несколько очередей по приоритетам, как следует организовать реализации, чтобы иметь возможность манипулировать очередями по приоритетам так же, как мы манипулируем другими типами данных? Подобного типа вопросы служили предметом обсуждения в главе 4. Программа 9.8 представляет обобщенный интерфейс для очередей по приоритетам равно как и для строк, которые обсуждались в разделе 4.8. Она

находит применение в ситуации, когда клиентская программа работает с ключами и ассоциированной с ними информацией и, будучи главным образом заинтересованной в операции доступа к информации, ассоциированной с ключом с наибольшим значением, она, тем не менее, может выполнять и другие многочисленные операции по обработке данных в объектах, о чем шла речь в начале главы. Все эти операции обращаются к конкретной очереди по приоритетам через дескрипторы (указатель на структуру, которая не имеет конкретного описания). Операция *вставить* возвращает дескриптор каждого объекта, добавляемого в очередь по приоритетам в клиентской программе. Дескрипторы объектов отличаются от дескрипторов очередей по приоритетам. При такой организации клиентские программы берут на себя обязанность отслеживать дескрипторы, которые впоследствии они могут использовать для определения, какие объекты подвергались воздействию со стороны операций *удалить* и *изменить приоритет*, и над какими очередями по приоритетам должны быть выполнены все указанные операции.

Данная организация налагает ограничения как на клиентскую программу, так и на реализацию. Клиентская программа не может получить доступ к информации никаким другим способом, кроме как за счет использования этого интерфейса. На нее возлагается ответственность за правильное использование дескрипторов: например, в реализации не существует сколь-нибудь подходящего способа проверки с целью выявления недопустимости такого действия, как использование клиентской программой дескриптора удаленного элемента. С другой стороны, реализация не может свободно перемещаться по информации, поскольку у клиентских программ имеются дескрипторы, которые они могут использовать на более поздних стадиях. Этот момент станет более понятным, когда мы приступим к исследованию деталей реализации. Как обычно, какой бы уровень детализации ни был бы выбран в наших реализациях, абстрактный интерфейс, такой как программа 9.8, представляет собой полезную отправную точку для нахождения компромисса между потребностями приложений и потребностями реализаций.

Прямолинейные реализации базовых операций над очередями по приоритетам, которые даны в программе 9.9, используют неупорядоченные представления данных в виде двухсвязных списков. Этот программный код иллюстрирует природу интерфейса; очень несложно построить другие, столь же бесхитростные реализации, используя другие элементарные представления.

### Программа 9.8. Полный АТД очереди по приоритетам

Данный интерфейс для АТД очереди по приоритетам дает клиентским программам возможность удалять элементы и менять приоритеты (с использованием дескрипторов, предлагаемых программной реализацией), а также выполнять слияние очередей.

```
template <class Item>
class PQ
{
    private:
        //Программный код, зависящий от реализации
    public:
        //Определение дескриптора в зависимости от реализации
        PQ(int);
        int empty() const;
```

```

        handle insert(Item);
        Item getMax();
        void change(handle, Item);
        void remove(handle);
        void join(PQ<Item>&);
};

```

В соответствие с обсуждениями в разделе 9.1, реализация, представленная в программах 9.9 и 9.10, подходит для тех приложений, в которых очередь по приоритетам небольшая и операции *удалить наибольший* или *найти наибольший* выполняются нечасто; в остальных случаях более предпочтительными оказываются реализации на базе пирамидальной сортировки. Реализация функции **fixUp** и **fixDown** на пирамидально упорядоченных деревьях с явными связями с одновременной поддержкой целостности дескрипторов — достаточно сложная задача, которую мы оставляем читателю в качестве упражнения, а сами займемся подробным рассмотрением двух альтернативных подходов в разделах 9.6 и 9.7.

Полный АД, такой как в программе 9.8, обладает массой достоинств, однако иногда полезно рассматривать другие подходы, налагающие другие ограничения на клиентские программы и реализации. В разделе 9.6 рассматривается пример, в условиях которого в обязанности клиентской программы входит ведение записей и ключей, а программы, работающие с очередями по приоритетам, обращаются к ним неявно.

Иногда приемлемыми и даже желательными могут оказаться небольшие изменения интерфейсов. Например, может потребоваться функция, возвращающая значение ключа с наибольшим приоритетом в очереди, а не способ обращения к этому ключу и связанной с ним информацией. Кроме того, на передний план выходят проблемы, которые исследовались в разделе 4.8 и которые связаны с управлением памятью и семантикой копирования. Мы не рассматриваем операции *уничтожить* (*destroy*) или *копировать* (*copy*), а выбрали один из нескольких возможных вариантов операции *объединить* (*join*).

### Программа 9.9. Неупорядоченная очередь по приоритетам в виде двухсвязного списка.

Эта реализация содержит операции *создать* (*construct*), *проверить на наличие элементов* (*test if empty*) и *вставить* (*insert*), взятые из интерфейса программы 9.8 (см. программу 9.10, в которой содержатся реализации четырех других функций). Она поддерживает простой неупорядоченный список, в котором имеются головной и хвостовой узлы. Структура **node** (узел) описывается как узел двухсвязного списка (элемент и две связи). Представление частных данных состоит из головы списка и хвостовых связей.

```

Template <class Item>
class PQ
{
    private:
        struct node
        { Item item; node *prev, *next;
          node(Item v)
          { item = v; prev = 0; next = 0; }
        };
};

```

```

    typedef node* link;
    link head, tail;
public:
    typedef node* handle;
    PQ(int = 0)
    {
        head = new node(0);    tail = new node(0);
        head->prev = tail;    head->next = tail;
        tail->prev = head;    tail->next = head;
    }
    int empty( ) const
    { return head->next->next == head; }
    handle insert(Item v)
    { handle t = new node(v);
      t->next = head->next;    t->next->prev = t;
      t->prev = head;    head->next = t;
      return t;
    }
    Item getmax( );
    void change(handle, Item);
    void remove(handle);
    void join(PQ<Item>&);
};

```

Добавление таких процедур в интерфейс, представленный в программе 9.8, не связан с какими-либо трудностями; гораздо труднее разработать такую реализацию, в рамках которой гарантируется логарифмическая производительность для всех видов операций. В тех приложениях, в которых очереди по приоритетам не достигают больших размеров или в которых смесь операций *вставить* и *удалить наибольший* обладает рядом специальных свойств, предпочтительным может оказаться полностью гибкий интерфейс. С другой стороны, в тех приложениях, в которых очереди возрастают до внушительных размеров и наблюдается или подтверждается соответствующими расчетами десятикратное, а то и стократное возрастание производительности, можно ограничиться именно теми операциями, которые обеспечивают упомянутый рост производительности. Были проведены обширные исследования, результаты которых были положены в основу алгоритмов, манипулирующих очередями по приоритетам при помощи различных сочетаний операций. Биномиальная очередь, описанная в разделе 9.7, представляет собой важный пример такого рода.

#### Программа 9.10. Очередь по приоритетам в виде двухсвязного списка (продолжение)

Подставляя эти реализации вместо соответствующих объявлений в программе 9.9, мы получаем полную реализацию очереди по приоритетам. Операция *удалить наибольший* требует просмотра всего списка, тем не менее, затраты на поддержку двухсвязного списка оправдываются тем фактом, что операции *изменить приоритет*, *удалить* и *объединить* реализованы таким образом, что выполняются за постоянное время, при этом к спискам применяются только элементарные операции (обращайтесь к главе 3 за более подробным описанием связных списков).

При необходимости можно добавить в программы деструктор, конструктор копирования и перегруженный оператор присваивания для дальнейшего совершенствования этого приложения с целью получения АТД первого класса (см. раздел 4.8). Обратите внимание на то обстоятельство, что реализация функции *join* (объединить) присваивает

список узлов из параметров, которые должны быть включены в результат, но при этом она не делает их копий.

```

Item getmax( )
{ Item max; link x = head->next;
  for (link t = x; t->next !=head; t = t->next)
    if (x->item < t->item) x =t;
  max = x->item;
  remove(x);
  return max;
}
void change (handle x, Item v)
{ x->key = v; }
void remove(handle x)
{
  x->next->prev-> = x->prev;
  x->prev->next-> = x->next;
  delete x;
}
void join(PQ<Item>& p)
{
  tail-> prev ->next = p.head->next;
  p.head->next->prev = tail-> prev;
  head->prev = p.tail;
  p.tail->next = head;
  delete tail; delete p.head;
  tail = p.tail
}

```

## Упражнения

**9.38.** Какой реализацией очереди по приоритетам вы воспользуетесь для того, чтобы найти 100 наименьших чисел в наборе из  $10^6$  случайных чисел? Докажите правильность полученного ответа.

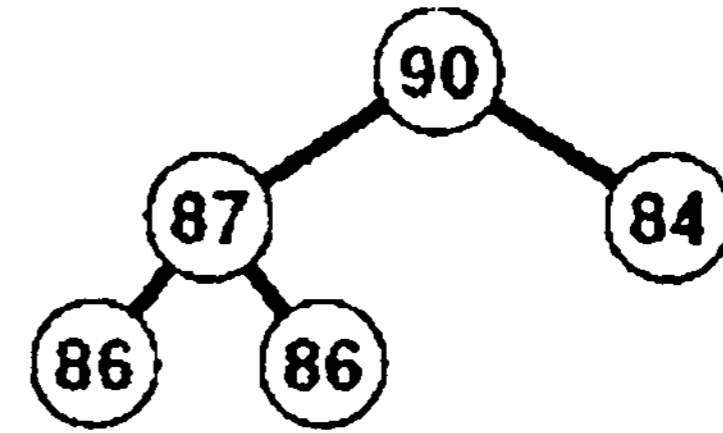
**9.39.** Построить реализации, подобные программам 9.9 и 9.10, которые используют упорядоченные двухсвязные списки. *Совет:* Поскольку клиентская программа имеет дескрипторы конкретных элементов структуры данных, ваши программы могут изменять в узлах только связи (но не ключи).

**9.40.** Построить реализации операций *вставить* и *удалить наибольший* (интерфейс очереди по приоритетам в программе 9.1), воспользовавшись пирамидально упорядоченными полными деревьями с явными узлами и связями. *Совет:* Поскольку в клиентской программе нет дескрипторов конкретных элементов структуры данных, можно воспользоваться преимуществом того факта, что проще осуществить обмен информационными полями в узлах, нежели обмен самими узлами.

- **9.41.** Построить реализации операций *вставить*, *удалить наибольший*, *изменить приоритет* и *удалить* (интерфейс очереди по приоритетам в программе 9.8), воспользовавшись пирамидально упорядоченными деревьями с явными связями. *Совет:* Поскольку в клиентской программе имеются дескрипторы конкретных элементов структуры данных, это упражнение сложнее упражнения 9.40 не только потому, что узлы должны быть трехсвязными, но также и потому, что ваши программы могут изменять только связи (но не ключи) в узлах.



- 9.42. Добавить реализацию (решение "в лоб") операции *объединить* в реализацию из упражнения 9.41.
- 9.43. Добавить объявления деструктора, конструктора копирования и перегруженный оператор присваивания в программу 9.8, чтобы превратить ее в АТД первого класса, включить соответствующие реализации в программы 9.9 и 9.10 и написать программу-драйвер, которая протестирует полученный интерфейс и реализацию.
  - 9.44. Внести изменения в интерфейс и реализацию операции *объединить* в программы 9.9 и 9.10, которые приведут к тому, что она возвращает **PQ** (результат объединения параметров) и имеет своим результатом разрушение аргументов.
- 9.45. Построить интерфейс очереди по приоритетам и реализацию, которая поддерживает операции *построить* и *удалить наибольший*, используя для этой цели турнир (см. раздел 5.7). Программа 5.19 обеспечивает основу для операции *построить* (*construct*).
- 9.46. Преобразовать решение упражнения 9.45 в АТД первого класса.
  - 9.47. Добавить операцию *вставить* в решение упражнения 9.45.



k	qp[k]	pq[k]	data[k]	
0			Wilson	63
1	5	3	Johnson	86
2	2	2	Jones	87
3	1	4	Smith	90
4	3	9	Washington	84
5		1	Thompson	65
6			Brown	82
7			Jackson	61
8			White	76
9	4		Adams	86
10			Black	71

## 9.6. Очередь по приоритетам для индексных элементов

Предположим, что записи, обрабатываемые в очередях по приоритетам, образуют массив. В этом случае программам, работающим с очередями по приоритетам, имеет смысл обращаться к элементам, используя индексы массивов. Более того, индексы массива могут рассматриваться в качестве дескрипторов, чтобы реализовать все операции, выполняемые над очередями по приоритетам. Интерфейс, соответствующий этому описанию, приводится в программе 9.11. Рисунок 9.13 демонстрирует, как такой подход можно применить в примере, который использовался для изучения индексной сортировки в главе 6. Обходясь без копирования и не внося специальных изменений в записи, можно поддерживать очередь по приоритетам, содержащую некоторое подмножество записей.

### РИСУНОК 9.13. СТРУКТУРЫ ДАННЫХ ИНДЕКСНОГО СОРТИРУЮЩЕГО ДЕРЕВА

*Манипулируя индексами, а не самими записями, можно построить очередь по приоритетам на подмножестве записей в массиве. В рассматриваемом случае сортирующее дерево размером 5 в массиве **pq** содержит имена 5 студентов с наивысшими рейтингами. Таким образом, **data[pq[1]].name** содержит **Smith**, имя студента, обладающего наивысшим рейтингом, и т.д.*

*Обращенный массив позволяет программам очереди по приоритетам трактовать индексы массива как дескрипторы. Например, если требуется поменять индекс Смита на 85, мы меняем содержимое **data[3].grade**, затем вызываем функцию **PQchange(3)**. Реализация очереди по приоритетам осуществляет доступ к записи **pq[qp[3]]** в (или **pq[1]**, поскольку **qp[3]=1**) и к новому ключу в **data[pq[1]].name** (или **data[3].name**, поскольку **pq[1]=3**).*

Использование индексов существующего массива — вполне естественное решение, однако оно ведет к реализациям с ориентацией, противоположной программе 9.8. Теперь уже клиентская программа не может свободно перемещаться по информации, поскольку программа, реализующая очередь по приоритетам, работает с теми же индексами, с которыми имеет дело и клиентская программа. В свою очередь, реализация очереди по приоритетам должна пользоваться индексами только в тех случаях, когда они сначала передаются ей клиентской программой.

При разработке реализации применяется в точности такой же подход, который использовался при индексной сортировке в разделе 6.8. Мы манипулируем индексами и перегружаем операцию `operator<` таким образом, что сравнения адресуются к элементам массива индексов клиентской программы. При этом возникают некоторые осложнения, поскольку необходимо, чтобы программа очереди по приоритетам отслеживала объекты так, чтобы она могла отыскать их, когда клиентская программа обращается к ним по дескриптору (индексу массива). С этой целью добавляется второй индексный массив, обеспечивающий отслеживание положения ключей в очереди по приоритетам. Чтобы локализовать поддержку этого массива, данные перемещаются только через операцию `exch`, поэтому необходимо дать соответствующее определение `exch`.

Полная реализация этого подхода с использованием сортирующих деревьев находится в программе 9.12. Эта программа только незначительно отличается от программы 9.5, тем не менее, она достойна специального изучения, поскольку является исключительно полезной в практических ситуациях. Будем называть структуру данных, построенную этой программой, *индексным сортирующим деревом* (*index heap*). Эта программа служит строительным блоком для других алгоритмов в частях 5—7. Как обычно, мы не включаем код обнаружения ошибок и предполагаем (например), что индексы никогда не выходят за пределы отведенного им диапазона, а пользователь не делает попыток вставить что-либо в заполненную очередь и удалить что-либо из пустой очереди. Добавление кода подобного назначения не вызывает особых затруднений.

### **Программа 9.11. Интерфейс АТД очереди по приоритетам для индексных элементов.**

Вместо того чтобы строить различные структуры данных из самих элементов данных, этот интерфейс обеспечивает возможность построения очереди по приоритетам с использованием индексов конкретных элементов массива клиентской программы. Программы, реализующие операции *вставить*, *удалить наибольший*, *изменить приоритеты* и *удалить*, используют дескриптор, представляющий собой индекс массива, а клиентская программа перегружает операцию `operator<` так, чтобы стало возможным сравнение двух элементов массива. Например, клиентская программа может определить `operator<` таким образом, что значением неравенства  $i < j$  становится результат сравнения `data[i].grade` и `data[j].grade`.

```
template <class Item>
class PQ
{
private:
    //Программный код, зависящий от реализации
public:
    PQ(int);
    int empty() const;
```

```

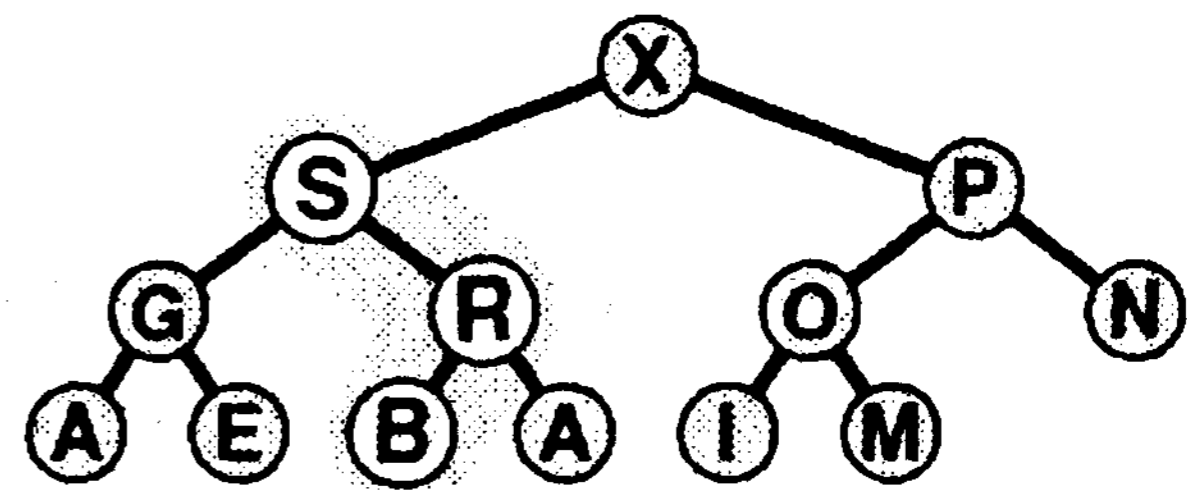
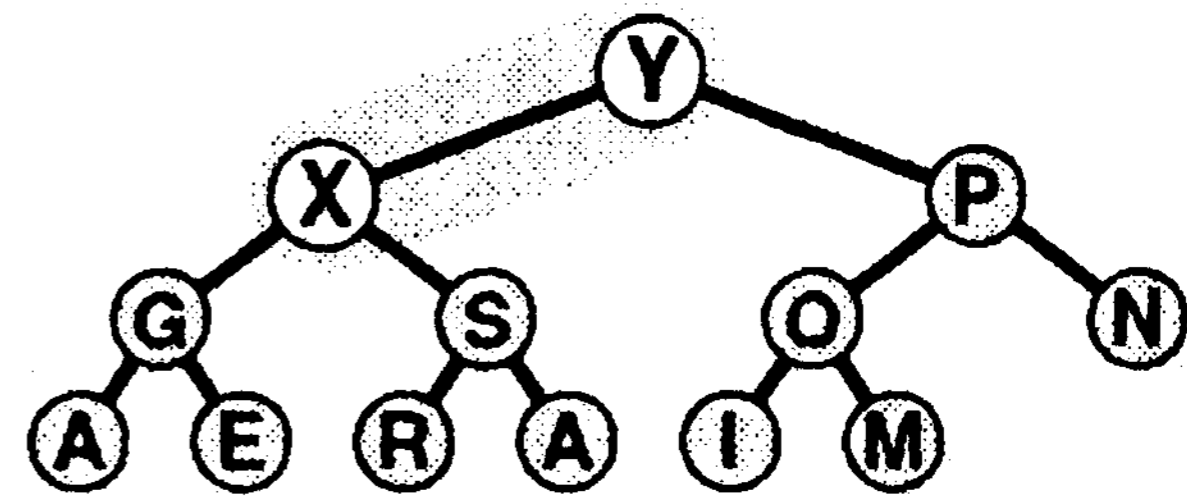
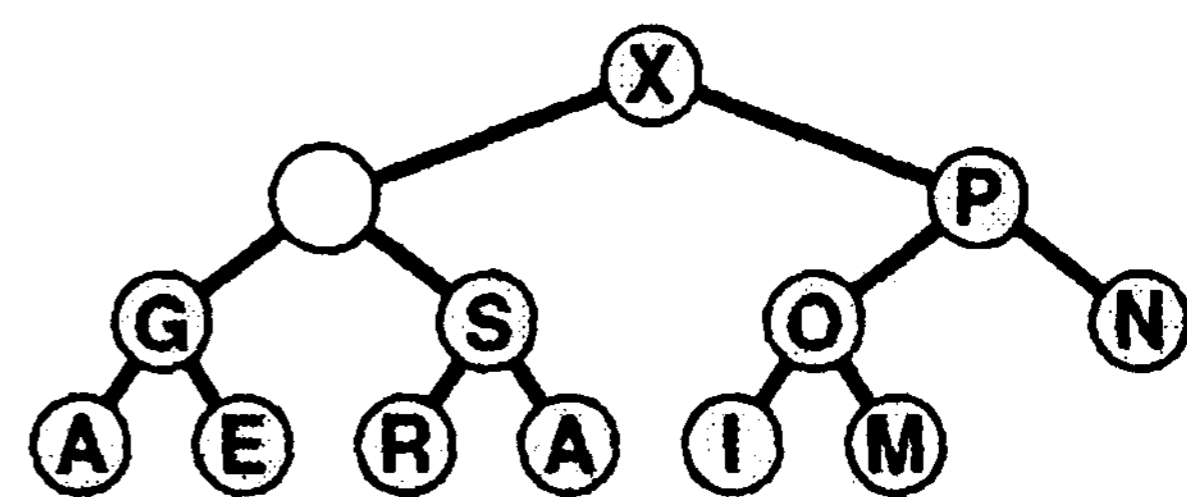
void insert(Index);
Index getMax();
void change(Index);
void remove(Index);
};

```

Этот же подход применим и для любой очереди по приоритетам, для которой используется представление в виде массива (например, см. упражнения 9.50 и 9.51). Основным недостатком применения подобного рода косвенной адресации состоит в необходимости расхода дополнительного объема памяти. Размер массива индексов должен иметь размер массива данных, тогда как максимальный размер очереди по приоритетам может быть намного меньше. Другой подход к построению очереди по приоритетам над данными, образующими массив, заключается в том, чтобы клиентская программа строила записи, состоящие из ключа и индекса массива в качестве дополнительной информации, или в использовании индексного ключа с перегруженной операцией `operator<`, поддерживаемой клиентской программой. Далее, если рассматриваемая реализация использует представление со списочным распределением памяти, такое как в программах 9.9 и 9.10 или в упражнении 9.41, то пространство памяти, используемое очередью по приоритетам, будет пропорционально максимальному количеству элементов в очереди в каждый конкретный момент времени. Такого рода подходы по сравнению с подходом, примененным в программе 9.12, выглядят предпочтительнее, если заранее нужно зарезервировать определенное пространство памяти необходимо заранее или если очередь по приоритетам включает в себя только небольшую часть массива данных.

### Программа 9.12. Очередь по приоритетам, построенная на базе индексного сортирующего дерева

Эта реализация программы 9.11 поддерживает `pq` как массив индексов некоторого клиентского массива. Например, если клиент определяет



**РИСУНОК 9.14. ИЗМЕНЕНИЕ ПРИОРИТЕТА ПРОИЗВОЛЬНОГО УЗЛА СОРТИРУЮЩЕГО ДЕРЕВА**

На верхней диаграмме показано сортирующее дерево, о котором известно, что оно пирамидально упорядочено, за исключением разве что одного узла. Если этот узел больше своего родителя, то он должен продвигаться вверх, как показано на рис. 9.3. Эта ситуация демонстрируется и на средней диаграмме; в рассматриваемом случае узел *Y* поднимается вверх по дереву (в общем случае его продвижение может прекратиться, так и не достигнув корня). Если узел меньше, чем больший из его двух потомков, то он должен опуститься вниз по дереву, как показано на рис. 9.3. Эта иллюстрация показана на нижней диаграмме, в этом случае узел *B* продвигается вниз по дереву (в общем случае его продвижение может прекратиться, так и не достигнув нижнего уровня). Данной процедурой можно воспользоваться в качестве основы для операции **изменить приоритет** в сортирующем дереве с целью восстановления структуры сортирующего дерева после изменения значения ключа в узле, либо в качестве основы для операции **удалить** в сортирующем дереве с целью восстановления структуры сортирующего дерева после замены ключа в узле на самый правый ключ нижнего уровня дерева.

`operator<` для аргументов типа `Index`, как указано в комментарии, предшествующем программе 9.11, то когда `fixUp` выполняет сравнение `pq[j]` с `pq[k]`, она, по идее, сравнивает `data.grade[pq[j]]` с `data.grade[pq[k]]`. Мы предполагаем, что `Index` — это класс-оболочка, объект которого может индексировать массивы, так что мы можем зафиксировать позицию сортирующего дерева, соответствующую значению индекса `k` в `qp[k]`, что, в свою очередь, позволяет реализовать операции *изменить приоритет* и *удалить* (см. рис. 9.13). Инвариант `pq[qp[k]] = qp[pq[k]] = k` поддерживается для всех `k` сортирующего дерева (см. рис. 9.13).

```
template <class Item>
class PQ
{
private:
    int N; Index* pq; int* qp;
    void exch(Index i, Index j)
    { int t;
      t = qp[i]; qp[i] = qp[j]; qp[j] = t;
      pq[qp[i]] = i; pq[qp[j]] = j;
    }
    void fixUp(Index a[], int k);
    void fixDown(Index a[], int k, int N);
public:
    PQ(int maxN)
    { pq = new Index[maxN+1];
      qp = new int[maxN+1]; N = 0; }
    int empty( ) const
    { return N == 0; }
    void insert(Index v)
    { pq[++N] = v; qp[v] = N; fixUp(qp, N); }
    Index getMax( )
    {
      exch(pq[1], pq[N]);
      fixDown(qp, 1, N-1);
      return qp[N--];
    }
    void change(Index k)
    { fixUp(pq, qp[k]); fixDown(pq, qp[k], N); }
};
```

Сопоставление этого подхода, обеспечивающего реализацию очереди по приоритетам в полном объеме, с подходом из раздела 9.5, обнаруживает существенные различия в конструкции АТД. В первом случае (программа 9.8, например) в обязанность реализации очереди по приоритетам входит распределение памяти под ключи и ее освобождение, изменение значений ключей и тому подобное. Абстрактный тип данных предоставляет клиентской программе дескрипторы элементов данных, а эта программа осуществляет доступ к элементам данных только за счет обращений к программам поддержки очередей по приоритетам, передавая эти дескрипторы в параметрах. Во втором случае (например, программа 9.12) клиентская программа несет ответственность за ключи и записи, а программы поддержки очереди по приоритетам осуществляют доступ к этой информации только через дескрипторы, выбранные пользователем (индексы массива в случае программы 9.12). В обоих случаях требуется обеспечить взаимодействие между клиентской программой и реализацией.

Обратите внимание, что в данной книге мы заинтересованы в таком взаимодействии, которое выходит за рамки, очерченные механизмами поддержки языков программирования. В частности, хотелось бы, чтобы характеристики производительности реализаций соответствовали динамическому сочетанию операций, затребованному клиентской программой. Один из способов достижения такого соответствия — это применение реализаций с проверенными границами производительности в наихудшем случае, но в то же время многие задачи можно решить намного проще, сопоставляя требования этих задач в части производительности с возможностями простых реализаций.

## Упражнения

- 9.48. Предположим, что массив заполнен ключами **E A S Y Q U E S T I O N**. Показать содержимое массивов **pq** и **qr** после помещения этих ключей в первоначально пустое сортирующее дерево, используя программу 9.12.
- 9.49. Добавить в программу 9.12 операцию *удалить*.
- 9.50. Реализовать АТД *очереди по приоритетам* для индексных элементов (см программу 9.11), воспользовавшись представлением очереди по приоритетам в виде упорядоченного массива.
- 9.51. Реализовать АТД *очереди по приоритетам* для индексных элементов (см. программу 9.11), воспользовавшись представлением очереди по приоритетам в виде неупорядоченного массива.
- 9.52. Для заданного массива из  $N$  элементов рассмотрим полное бинарное дерево из  $2N$  элементов (представленном в виде массива **pq**), содержащих индексы из этого массива, со следующими свойствами: (i) для  $i$  от 0 до  $N-1$  имеем  $pq[N+i]$  и (ii) для  $i$  от 1 до  $N-1$  имеем  $pq[i] = pq[2*i]$ , если  $a[pq[2*i]] > a[pq[2*i+1]]$ , и  $pq[i] = pq[2*i+1]$  — в противном случае. Такая структура называется *турниром индексного сортирующего дерева (index heap tournament)*, поскольку сочетает свойства индексных сортирующих деревьев и турниров (см. программу 5.19). Запишите турнир индексного сортирующего дерева, соответствующий ключам **E A S Y Q U E S T I O N**.
- 9.53. Реализовать АТД *очереди по приоритетам* для индексных элементов (см. программу 9.11), используя турнир индексного сортирующего дерева.

## 9.7. Биномиальные очереди

Ни одна из рассмотренных выше реализаций не может обеспечить высокоэффективное выполнение в наихудшем случае сразу всех операций *объединить*, *удалить наибольший* и *вставить*. Неупорядоченные связные списки позволяют быстро выполнять операции *объединить* и *вставить*, но на них медленно выполняется операция *удалить наибольший*; упорядоченные связные списки позволяют быстро выполнять операции *удалить наибольший*, но на них медленно выполняются операции *объединить* и *вставить*; на сортирующем дереве быстро выполняются *вставить* и *удалить наибольший*, но медленно — операция *объединить* и т.д. (см. таблицу 9.1). В приложениях, в которых операции *объединить* выполняются часто или в большом объеме, рекомендуется рассмотреть применение более совершенных структур данных.

В текущем контексте под термином "эффективная" понимается такая операция, для выполнения которой в наихудшем случае требуется время, не превосходящее логарифмической зависимости. Создается впечатление, что это ограничение исключает представления в виде массива, поскольку очевидно, что два массива можно объединить только путем перемещения всех элементов по крайней мере одного из массивов. Представление в виде неупорядоченного двухсвязного списка, предложенного программой 9.9, выполняет операцию *объединить* за постоянное время, но требует просмотра всего списка при выполнении операции *удалить наибольший*. Использование двухсвязных упорядоченных списков (см. упражнение 9.39) позволяет выполнять операцию *удалить наибольший* за постоянное время, однако требует линейного времени для выполнения слияния списков в рамках операции *объединить*.

Были разработаны многочисленные структуры данных, которые способны поддерживать эффективные реализации всех операций над очередями по приоритетам. Большая их часть базируется на прямом связном представлении пирамидально упорядоченных деревьев. Две связи необходимы для продвижения вниз по дереву (либо к двум потомкам в бинарном дереве, либо к первому потомку и к одному из следующих потомков в бинарном представлении общего дерева) и одна связь с родителем требуется для продвижения вверх по дереву. Разработка реализаций операций, поддерживающих пирамидальный порядок на любой (пирамидально упорядоченной) древовидной форме с явно определенными узлами и связями или на других представлениях в общем случае не требует особых ухищрений. Основная трудность сопряжена с такими динамическими операциями как *вставить*, *объединить* и *удалить наибольший*, которые требуют модификации древовидных структур. В основу различных структур данных положены различные стратегии модификации древовидных структур с одновременным сохранением баланса в дереве. В общем случае алгоритмы используют деревья, обладающие большей гибкостью, нежели полные деревья, однако сохраняют их достаточно сбалансированными, чтобы временные показатели не выходили за пределы логарифмических границ.

Затраты ресурсов на поддержание трехсвязных структур могут оказаться обременительным — гарантия того, что конкретная реализация правильно использует три указателя во всех обстоятельствах, может оказаться достаточно трудной задачей (см. упражнение 9.40). Более того, в различных практических ситуациях трудно обосновать необходимость эффективной реализации всех без исключения операций, так что следует хорошо обдумать этот вопрос, перед тем как браться за подобную реализацию. С другой стороны, трудно доказать, что в таких эффективных реализациях нет необходимости, а затраты ресурсов на то, чтобы все операции над очередями с приоритетами выполнялись быстро, практически всегда можно оправдать. Независимо от любых соображений подобного рода, следующий шаг, заключающийся в переходе от сортирующих деревьев к структурам данных с целью достижения эффективной реализации операций *вставить*, *объединить* и *удалить наибольший*, сам по себе представляет несомненный интерес и достоин подробного изучения.

Даже для списочных представлений деревьев условие его пирамидальной упорядоченности и условие полноты пирамидально упорядоченного бинарного дерева являются чрезмерно жесткими, в силу чего получение эффективной реализации операции *объединить* становится невозможной. Предположим, имеется два пирамидально упо-

рядоченных дерева, но как слить их в одно? Например, если одно из этих деревьев содержит 1023 узла, а другое — всего лишь 255 узлов, то какой способ выбрать для их слияния, чтобы получить дерево из 1278 узлов, не затрагивая более 10 или 20 узлов? На первый взгляд задача слияния пирамидально упорядоченных деревьев кажется вообще невозможной, если эти деревья являются пирамидально упорядоченными и полными, тем не менее, были предложены различные более совершенные структуры данных, которые позволили ослабить условия пирамидальной упорядоченности и сбалансированности с тем, чтобы придать структурам данных большую гибкость, необходимую для получения эффективной реализации операции *объединить*. Далее мы рассмотрим оригинальное решение этой проблемы, получившее название биномиальной очереди, полученное Вильемином (Vuillemin) в 1978 г.

Вначале следует отметить, что операция *объединить* (*join*) тривиальна для одного специального типа дерева с ослабленным требованием необходимости быть пирамидально упорядоченным.

**Определение 9.4.** *Бинарное дерево, содержащее узлы с ключами, называется левосторонним пирамидально упорядоченным (left heap ordered), если ключ каждого узла больше или равен всем ключам левого поддерева этого ключа (если таковое имеется).*

**Определение 9.5.** *Сортирующее дерево степени 2 есть левостороннее пирамидально упорядоченное дерево, состоящее из корневого узла с пустым правым поддеревом и полным левым поддеревом. В дереве, соответствующем сортирующему дереву степени 2 по левому потомку, соответствие правому потомку называется биномиальным деревом (binomial tree).*

Биномиальные деревья и сортирующие деревья степени 2 эквивалентны. Мы будем работать с обоими представлениями, поскольку в своем воображении несколько легче построить зрительный образ биномиальные дерева, в то время как упрощенное представление сортирующих деревьев степени 2 приводит к более простой реализации. В частности, мы ощущаем свою зависимость от следующих факторов, являющихся прямым следствием приведенных определений:

- Количество узлов сортирующего дерева степени 2 есть степень числа 2.
- Ни один из узлов не обладает ключом, который превосходит по значению ключ корня.
- Биномиальные деревья пирамидально упорядочены.

Тривиальной операцией, на которой базируются все биномиальные алгоритмы, является объединение двух сортирующих деревьев степени 2, состоящих из одинакового числа узлов. В результате объединения получаем сортирующее дерево, содержащее в два раза большее число узлов, которое, как показано на рис. 9.16, совсем нетрудно построить. Корневой узел с большим значением ключа становится корнем результирующего дерева (другой исходный корень при этом становится потомком корня результирующего дерева), а его левое поддерево становится правым поддеревом другого корневого узла. Если задано связное представление деревьев, то операция объединения выполняется за постоянное время: мы всего лишь устанавливаем две связи в вершине. Программная реализация этой операции находится в программе 9.13. Эта базовая операция является центральным звеном общего решения пробле-

мы реализации очереди по приоритетам без единой медленной операции, предложенного Виллемином (Vuillemin).

**Определение 9.6.** *Биномиальная очередь представляет собой набор сортирующих деревьев степени 2, ни одно из которых не совпадает с остальными по размеру. Структура биномиальной очереди определяется числом узлов этой очереди в соответствии с двоичным представлением целых чисел.*

Биномиальная очередь из  $N$  элементов содержит по одному сортирующему дереву на каждый бит двоичного представления числа  $N$ . Например, биномиальная очередь из 13 узлов содержит одно 8-узловое сортирующее дерево, одно 4-узловое и одно 1-узловое дерево (см. рис. 9.15). В биномиальной очереди размера  $N$  содержатся максимум  $\lg N$  сортирующих деревьев степени 2, высота которых не превосходит значения  $\lg N$ .

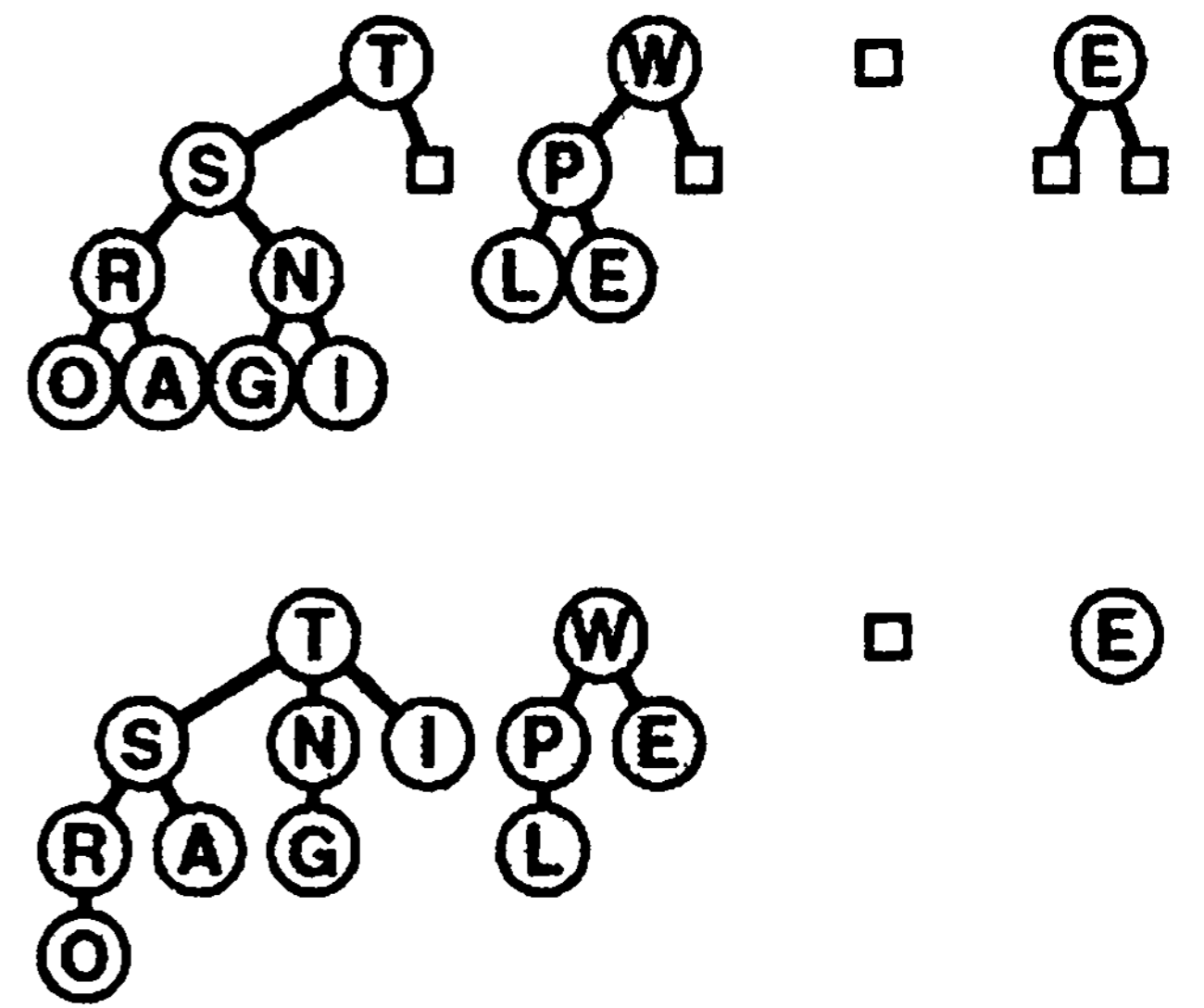
В соответствии с определениями 9.5 и 9.6, сортирующие деревья степени 2 (и дескрипторы элементов данных) представляются как связи с узлами, содержащими ключ и две связи каждый (подобно явному древовидному представлению турниров на рис. 5.10), а биномиальные очереди представляются как массивы сортирующих деревьев степени 2 путем включения следующего кода в приватную часть программы 9.8:

```
struct node
{ Item item; node *l, *r;
  node(Item v)
  { item = v; l = 0; r = 0; }
};
typedef node *link;
link* bq;
```

### Программа 9.13. Объединение двух сортирующих деревьев степени 2 одинаковых размеров

Чтобы объединить два сортирующих дерева степени 2 одинаковых размеров в одно сортирующее дерево степени 2 двойного размера достаточно изменить лишь несколько связей. Эта функция, которая в реализации определена как приватная функция-член, представляет собой ключевой фактор, обеспечивающий эффективность алгоритма биномиальной очереди.

```
static link pair(link p, link q)
{
  if (p->item < q->item)
    { p->r = q->l; q->l = p; return q; }
  else {q->r = p->l; p->l = q; return q; }
}
```



**РИСУНОК 9.15. БИНОМИАЛЬНАЯ ОЧЕРЕДЬ РАЗМЕРА 13**

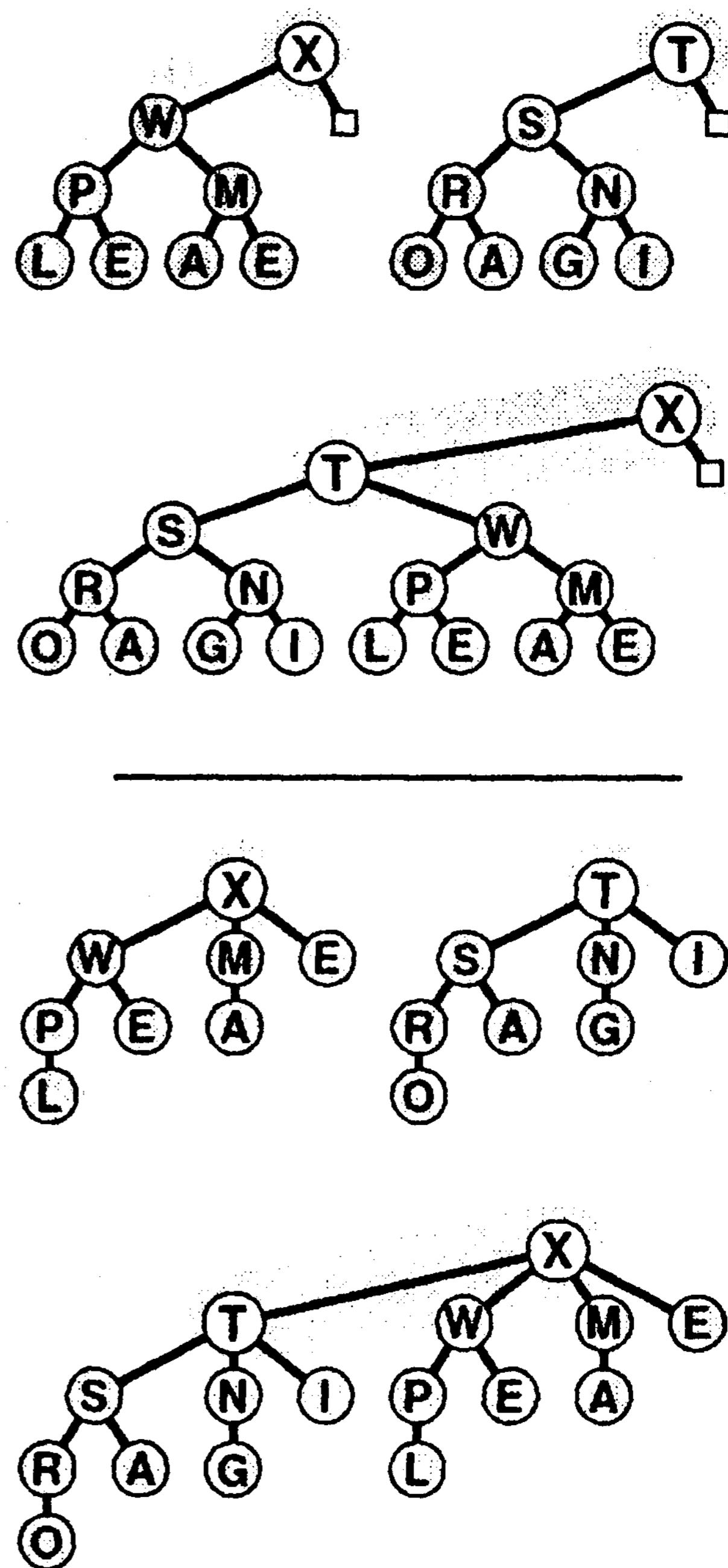
Биномиальная очередь размера  $N$  представляет собой список левосторонних пирамидально упорядоченных сортирующих деревьев степени 2, по одному на каждый бит двоичного представления числа  $N$ . Таким образом, биномиальная очередь размера  $13 = 1101_2$  состоит из одного 8-узлового сортирующего дерева, одного 4-узлового и одного 1-узлового деревьев. На диаграмме показано представление в виде левостороннего пирамидально упорядоченного сортирующего дерева степеней 2 (сверху) и представление в виде биномиального пирамидально упорядоченного дерева (внизу) одной и той же биномиальной очереди.



Размеры массивов невелики и высота деревьев тоже, а само это представление обладает достаточной гибкостью, чтобы стала возможной реализация всех операций с очередями по приоритетам менее чем за  $\lg N$  шагов, как мы в этом сейчас убедимся.

Начнем с рассмотрения операции *вставить* (*insert*). Процесс вставки нового элемента в биномиальную очередь в точности отображает процесс увеличения значения двоичного числа на единицу. Чтобы увеличить значения двоичного числа на единицу, мы двигаемся справа налево, заменяя 1 на 0 в силу необходимости выполнения переноса, обусловленного тем, что  $1 + 1 = 10_2$ , пока не обнаружим 0 в самой правой позиции, который заменяем единицей. Аналогичным образом, чтобы добавить в биномиальную очередь новый элемент, мы продвигаемся справа налево, выполняя слияние сортирующих деревьев, соответствующих битам 1 с переносимым сортирующим деревом, пока не найдем самую правую пустую позицию, в которую и помещаем переносимое дерево.

В частности, для вставки нового элемента в биномиальную очередь мы превращаем новый элемент в 1-узловое сортирующее дерево. Далее, если  $N$  есть четное число (значение самого правого разряда равно 0), мы просто помещаем это сортирующее дерево в самую правую пустую позицию биномиальной очереди. Если  $N$  — нечетное число (значение самого правого разряда составляет 1), мы объединяем сортирующее 1-дерево, соответствующее новому элементу, с сортирующим 1-деревом в самой крайней правой позиции биномиальной очереди, в результате чего получаем сортирующее 2-дерево переноса. Если позиция, соответствующая 2 в биномиальной очереди, пуста, то сортирующее дерево переноса помещается в эту позицию, в противном случае производится слияние сортирующего 2-дерева переноса с сортирующим 2 деревом из биномиальной очереди, образуя при этом 4-дерево переноса, и так продолжая процесс до тех пор, пока не будет достигнута пустая позиция в биномиальной очереди. Этот процесс демонстрируется на рис. 9.17, а в программе 9.14 приведена его реализация.



**РИСУНОК 9.16. ОБЪЕДИНЕНИЕ ДВУХ СОРТИРУЮЩИХ ДЕРЕВЬЕВ СТЕПЕНИ 2 ОДИНАКОВОГО РАЗМЕРА**

*Мы объединяем два сортирующих дерева степени 2 (сверху), помещая больший из двух корней в вершину результирующего дерева, при этом поддерево (левое) этого корня становится правым поддеревом другого исходного корня. Если операнды содержат  $2^n$  узлов, то результат содержит  $2^{n+1}$  узлов. Если операндами являются пирамидально упорядоченные левосторонние деревья, такой же порядок сохраняется и в результирующем сортирующем дереве, при этом ключ с наибольшим значением находится в корне. Отображение той же операции, ориентированной на представление пирамидально упорядоченного биномиального дерева, показано под разделительной чертой.*

**Программа 9.14. Вставка в биномиальную очередь**

Для вставки узла в биномиальную очередь его сначала потребуется превратить в 1-дерево и идентифицировать как сортирующее 1-дерево переноса, а затем повторять в цикле следующий процесс, начиная с  $i = 0$ . Если в биномиальной очереди сортирующее  $2^i$ -дерево отсутствует, в очередь помещается  $2^i$ -дерево переноса. Если в биномиальной очереди такое дерево есть, оно объединяется с таким же новым деревом (с применением функции `pair` из программы 9.13) и получается  $2^{i+1}$ -дерево, после чего значение  $i$  увеличивается на 1 и процесс продолжается до тех пор, пока не обнаруживается пустая позиция для сортирующего дерева в биномиальной очереди.

```

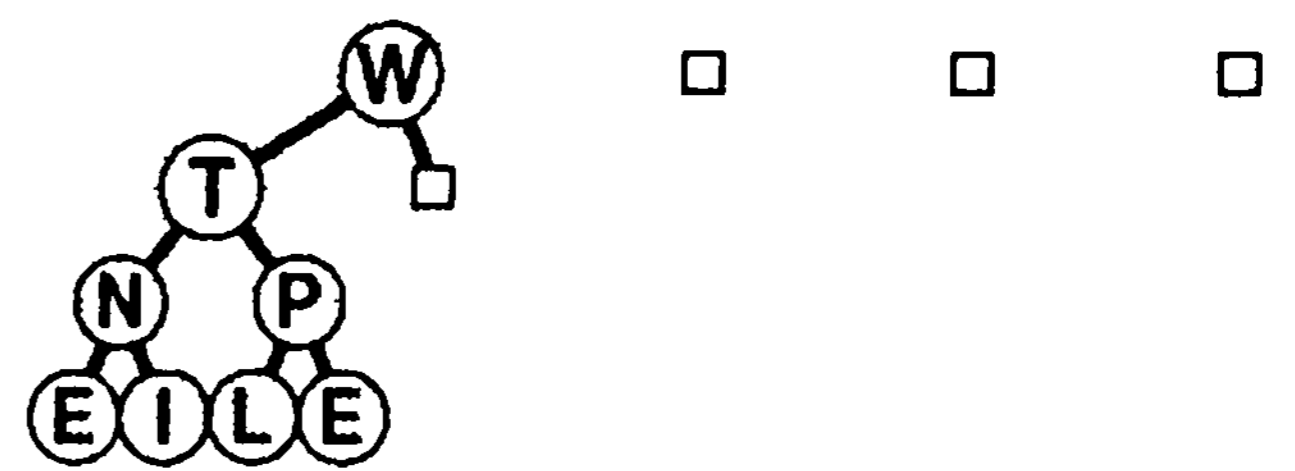
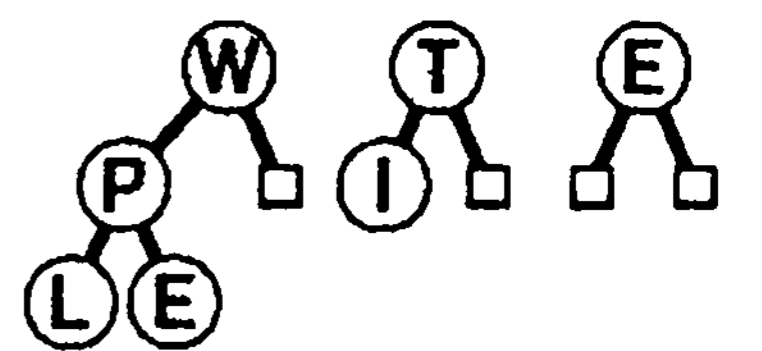
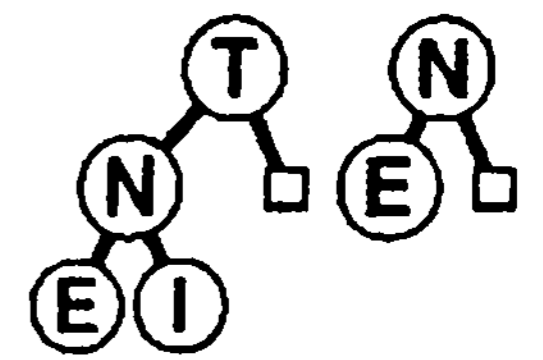
handle insert(Item)
{ link t = new node(v), c = t;
  for (int i = 0; i < maxBQsize; i++)
  {
    if (c == 0) break;
    if (bq[i] == 0)
      { bq[i] = c; break; }
    c = pair(c, bq[i]); bq[i] = 0;
  }
  return t;
}

```

Другие операции, выполняющиеся в биномиальных очередях, понять проще, если сравнить их с операциями двоичной арифметики. Как мы вскоре убедимся, реализация операции *объединить* соответствует реализации операции сложения двоичных чисел.

Предположим на момент, что в нашем распоряжении имеется (эффективная) функция для операции *объединить* (`join`), предназначенная для слияния очереди по приоритетам, ссылка на которую передается во втором операнде функции, с очередью по приоритетам, ссылка на которую находится в первом операнде (результат выполнения операции сохраняется в первом операнде). С использованием этой функции можно реализовать операцию *вставить* (`insert`) за счет вызова функции *объединить*, в одной из операндов которой передается биномиальная очередь размером 1 (см. упражнение 9.63).

Можно также реализовать операцию *удалить наибольший* (`remove the maximum`) путем одного вызова операции *объединить*. Для нахождения максимального элемента в биномиальной очереди просматриваются сортирующие деревья степени 2 этой очереди. Каждое такое дерево — левостороннее пирамидально упорядоченное сортирующее дерево, следовательно, максимальный его элемент находится в корне. Большой из корневых элементов является наибольшим элементом биномиальной очереди. Поскольку в биномиальной очереди не может быть более  $\lg N$  деревьев, общее время поиска наибольшего элемента меньше  $\lg N$ .

**РИСУНОК 9.17. ВСТАВКА НОВОГО ЭЛЕМЕНТА В БИНОМИАЛЬНУЮ ОЧЕРЕДЬ**

*Добавление элемента в биномиальную очередь из семи узлов аналогично выполнению арифметической операции двоичного сложения  $111_2 + 1 = 1000_2$  с переносом в каждом разряде. В результате получаем биномиальную очередь, представленную в нижней части диаграммы и состоящую из 8-дерева, причем 4-, 2- и 1-деревья отсутствуют.*

## Программа 9.15. Удаление наибольшего элемента из биномиальной очереди

Сначала производится просмотр корневых узлов, чтобы выяснить, какой из них больше, затем из биномиальной очереди удаляется сортирующее дерево степени 2, содержащее наибольший элемент. Далее из сортирующего дерева степени 2 удаляется корневой узел, содержащий наибольший элемент, и временно строится биномиальная очередь, которая содержит остальные составляющие части сортирующего дерева степени 2. В завершение при помощи операции *объединить* выполняется слияние полученной таким путем биномиальной очереди и исходной биномиальной очереди.

```

Item getmax( )
{ int i, max; Item v = 0;
  link* temp = new link[maxBQsize];
  for ( i = 0, max = -1; i < maxBQsize; i++)
    if ( bq[i] !=0)
      if ((max == -1) || (v < bq[i]->item))
        { max = i; v = bq[max]->item; }
  link x = bq[max]->l;
  for (i = max; i < maxBQsize; i++) temp[i] = 0;
  for (i = max; i > 0; i--)
    { temp[i-1] = x;
      x = x->r;
      temp[i-1]->r = 0;
    }
  delete bq[max]; bq[max] = 0;
  BQjoin (bq, temp);
  delete temp;
  return v;
}

```

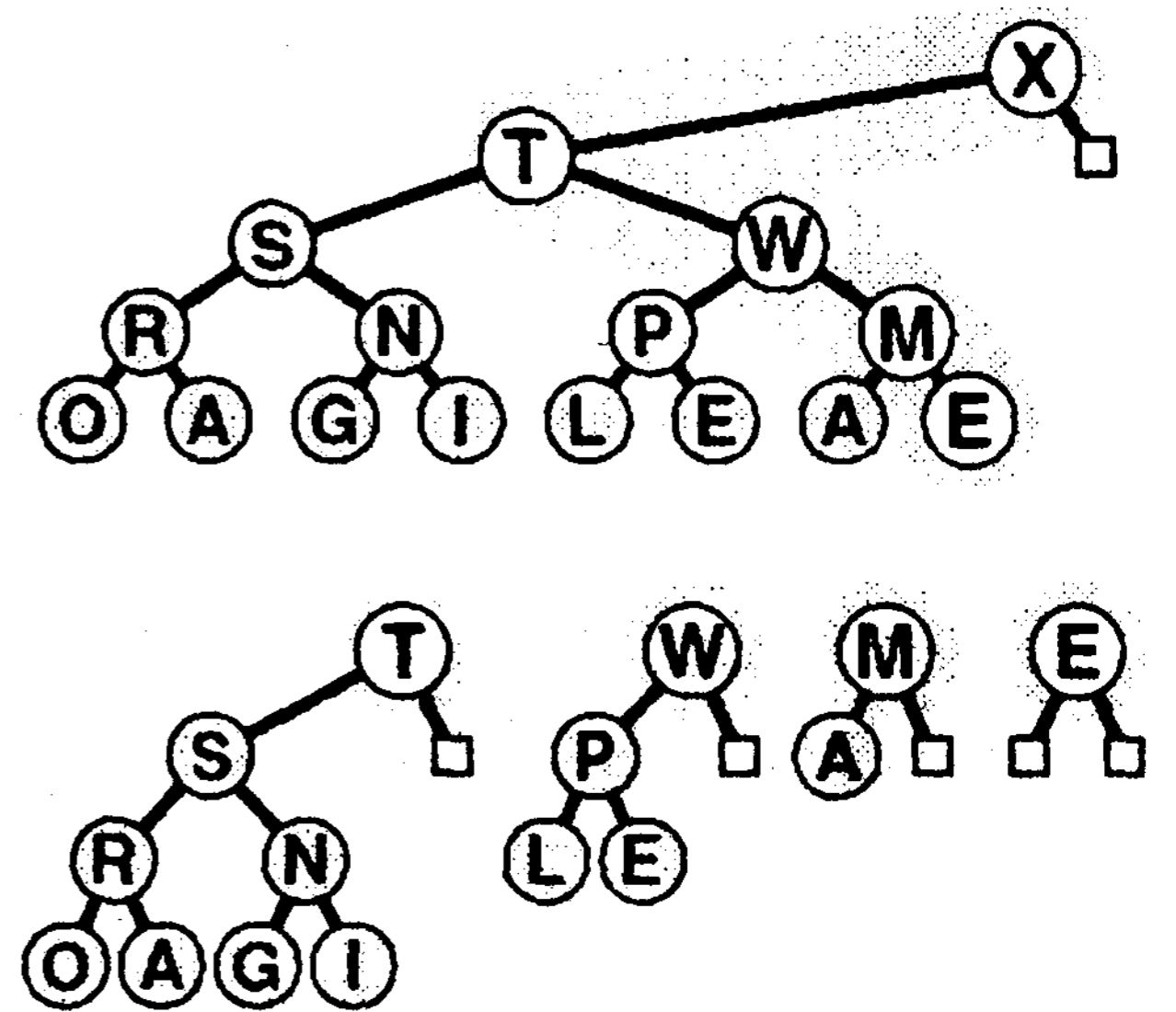


РИСУНОК 9.18. УДАЛЕНИЕ НАИБОЛЬШЕГО ЭЛЕМЕНТА ИЗ СОРТИРУЮЩЕГО ДЕРЕВА СТЕПЕНИ 2.

Убрав корень, получаем бор сортирующих деревьев степени 2; все они левосторонне пирамидально упорядочены, при этом их корни берутся из правого ствола дерева. Эта операция позволяет найти способ удаления наибольшего элемента из биномиальной очереди: убираем корень сортирующего дерева степени 2, которое содержит наибольший элемент, затем выполняем операцию *объединить* для слияния полученной биномиальной очереди с остальными сортирующими деревьями степени 2 исходной биномиальной очереди.

Перед выполнением операции *удалить наибольший* (*remove the maximum*) потребуется отметить, что удаление корня левостороннего сортирующего  $2^k$ -дерева приводит к появлению  $k$  левосторонних сортирующих деревьев степени 2 —  $2^{k-1}$ -дерево,  $2^{k-2}$ -дерево и так далее — которые легко реструктурировать в биномиальную очередь размера  $2^k - 1$ , как показано на рис. 9.18. Затем можно воспользоваться операцией *объединить*, чтобы объединить биномиальную очередь с остальной частью исходной очереди с целью завершения операции *удалить наибольший*. Эта реализация показана в программе 9.15.

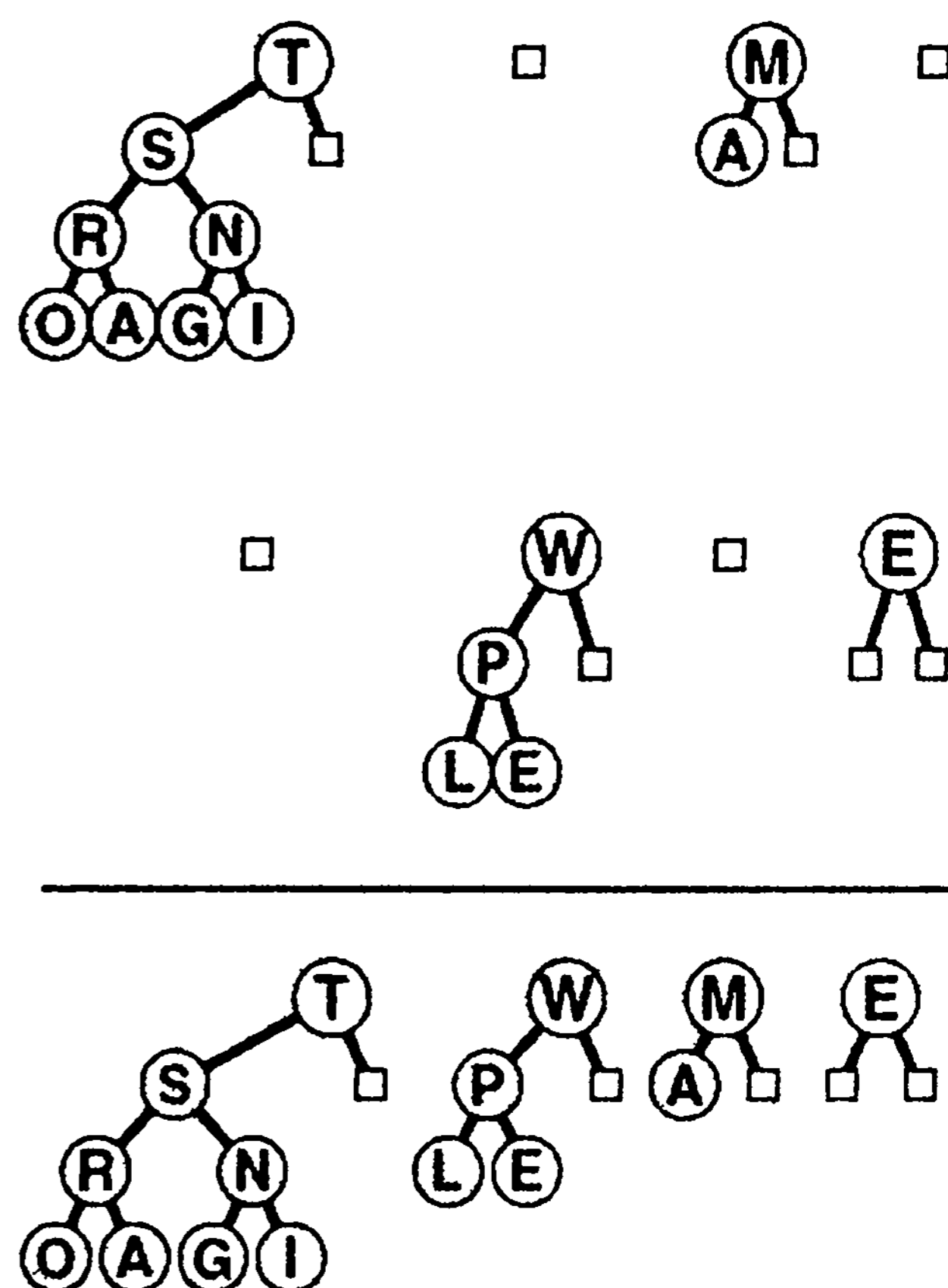
Каким образом объединяются две биномиальных очереди? Прежде всего, следует отметить, что эта операция тривиальна, если обе очереди не содержат сортирующих деревьев степени 2 одинакового размера, как показано на рис.9.19; мы просто выполняем слияние деревьев из обеих биномиальных очередей и получаем одну биномиальную очередь. Очередь размера 10 (состоящая из 8-дерева и 2-дерева) и очередь размера 5 (состоящая из 4-дерева и 1-дерева) путем простого

слияния образуют очередь размера 15 (состоящую из 8-дерева, 4-дерева, 2-дерева и 1-дерева). Способ, рассчитанный на более общие случаи, выполняется по прямой аналогии со сложением двух двоичных чисел, дополненного переносом (см. рис. 9.20).

Например, если очередь размером 7 (состоящую из 4-дерева, 2-дерева и 1-дерева) добавить к очереди размером 3 (состоящую из 2-дерева и 1-дерева), получится очередь размером 10 (состоящая из 8-дерева и 2-дерева). Для сложения потребуется слить 1-дерева и выполнить перенос 2-дерева, затем слить 2-дерева и выполнить перенос 4-дерева, затем слить 4-дерева, чтобы в результате получить 8-дерево точно таким же способом, каким выполняется двоичное сложение  $011_2 + 111_2 = 1010_2$ . Пример, представленный на рис. 9.19, проще примера, показанного на рис. 9.20, поскольку он аналогичен операции сложения  $1010_2 + 0101_2 = 1111_2$ .

Столь непосредственная аналогия с операциями двоичной арифметики вплотную подводит к естественной реализации операции *объединить* (см. программу 9.16). Для каждого разряда приходится рассматривать восемь возможных случаев в зависимости от значения каждого из 3 разрядов (перенос и два разряда в операндах). Соответствующая программа ненамного сложнее, чем программа простого арифметического сложения, поскольку в этом случае мы имеем дело с четко различимыми сортирующими деревьями, а не с неразличимыми битами, однако в обоих случаях принципиальных трудностей не возникает. Например, если все три разряда принимают значения 1, мы должны оставить одно дерево в получающейся биномиальной очереди, а два других объединить и перенести в следующую позицию. В самом деле, эта операция предоставляет полный цикл на абстрактных типах данных: мы (открыто) противимся искушению перевести программу 9.16 в статус абстрактной двоичной дополнительной процедуры, в рамках которых реализация биномиальной очереди есть ни что иное, как клиентская программа, использующая более сложную процедуру сложения битов, представленную программой 9.13.

**Лемма 9.7.** *Все операции на очереди по приоритетам как на абстрактном типе данных могут быть реализованы на биномиальной очереди таким образом, что для выполнения любой из них на очереди из  $N$  элементов требуется  $O(\lg N)$  шагов.*



**РИСУНОК 9.19. ОБЪЕДИНЕНИЕ ДВУХ БИНОМИАЛЬНЫХ ОЧЕРЕДЕЙ (БЕЗ ПЕРЕНОСА)**

*В том случае, когда две объединяемые биномиальные очереди содержат только сортирующие деревья степени 2, размеры которых попарно различны, операция объединения есть ни что иное как слияние. Выполнение этой операции аналогично сложению двух двоичных чисел, в условиях которой не встречаются сложения типа  $1+1$  (т.е. без переносов). В рассматриваемом случае биномиальная очередь из 10 узлов сливается с очередью из 5 узлов, результатом чего является биномиальная очередь из 15 узлов, соответствующая операции  $1010_2 + 0101_2 = 1111_2$ .*

Указанные рамки производительности служат целью разработки соответствующей структуры данных. Они представляют собой последствия того факта, что во всех реализациях имеется один или два цикла, которые выполняют итерацию на корнях деревьев биномиальной очереди.

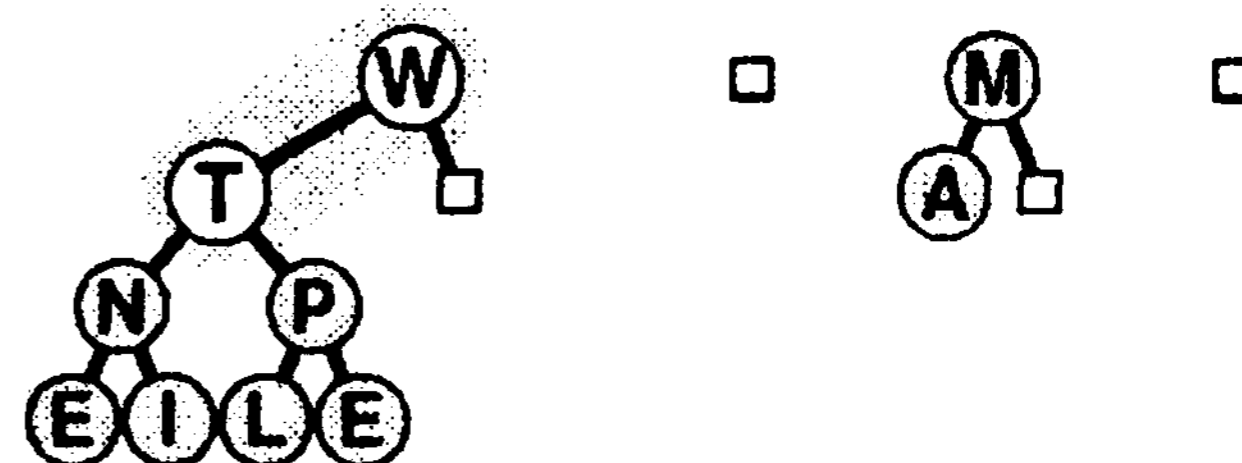
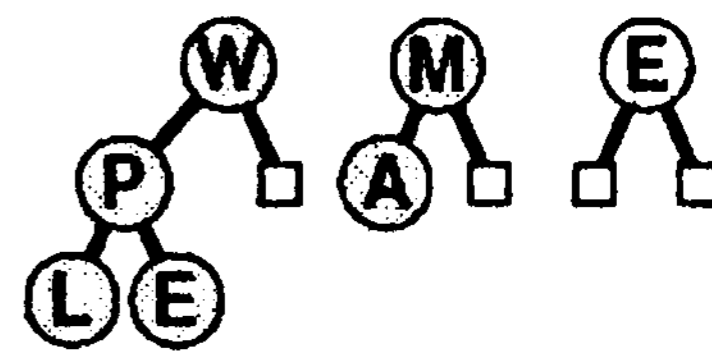
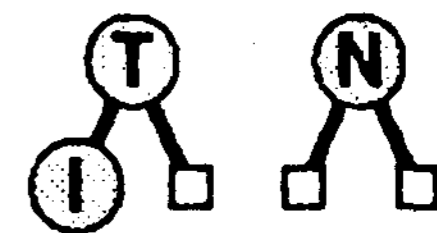
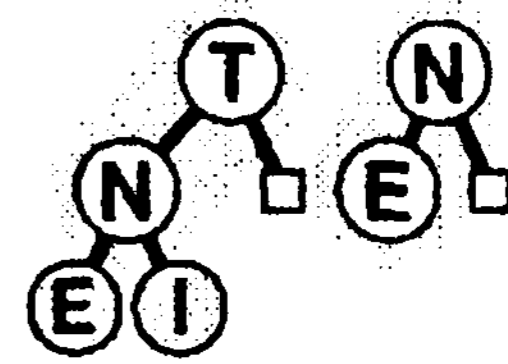
### Программа 9.16. Объединение (слияние) двух биномиальных очередей

Данная программа моделирует арифметическую операцию сложения двух двоичных чисел. Продвигаясь справа налево с начальным значением разряда переноса, равным 0, принимается во внимание три возможных случая (учитываются все возможные значения операндов и разряда переноса). Например, случай 3 соответствует условиям, когда биты обоих операндов принимают значения 1, а бит переноса — значение 0. В этом случае результатом будет 0, но перенос принимает значение 1 (т.е., результат сложения значений разрядов операндов).

Подобно функции `pair`, рассматриваемая функция является приватной функцией-членом реализации, которая вызывается функциями `getmax` и `join`. Функция абстрактного типа данных `join(PQ<Item>& p)` реализована в виде вызова `BQjoin(bq, p.bq)`.

```
static inline int test( int C, int B, int A)
{return 4*C + 2*B + 1*A}
static void BQjoin (link *a, link *b)
{ link c = 0;
  for (int i = 0; i < maxBQsize; i++)
    switch(test(c != 0; b[i] != 0, a[i] != 0))
    {
      case 2: a[i] = b[i]; break;
      case 3: c = pair(a[i], b[i]);
              a[i] = 0; break;
      case 4: a[i] =c, c =0; break;
      case 5: c = pair(c, a[i]);
              a[i] = 0; break;
      case 6:
      case 7: c = pair(c, b[i]); break;
    }
}
```

В целях упрощения будем считать, что наши реализации проходят в цикле через все деревья, так что время их выполнения пропорционально логарифму максимального размера биномиальной очереди. Можно сделать так, чтобы эти реализации соответствовали указанным граничным значениям в тех случаях, когда фактический размер очереди намного меньше, чем ее максимальный размер, путем отслеживания размеров очереди или за счет выбора соответствующего значения сигнальной метки, отмечающей точки, в которых циклы должны прекратиться (см. упражнения 9.61 и 9.62). Во многих ситуациях результат внесения подобного рода изменений не стоит



### РИСУНОК 2.20. ОБЪЕДИНЕНИЕ ДВУХ БИНОМИАЛЬНЫХ ОЧЕРЕДЕЙ

Добавление биномиальной очереди из 3 узлов к биномиальной очереди из 7 узлов имеет своим результатом очередь из 10 узлов как результат выполнения процесса, который моделирует операцию сложения  $011_2 + 111_2 = 1010_2$  двоичной арифметики. Сложение  $N$  и  $E$  дает пустое 1-дерево и перенос 2-дерева, содержащего узлы  $N$  и  $E$ . Последующее сложение трех 2-деревьев оставляет одно из них в итоговой очереди, а в перенос попадает 4-дерево, содержащее узлы  $TNEI$ . Это 4-дерево складывается с другим 4-деревом, образуя биномиальную очередь, показанную в нижней части диаграммы. Процесс охватывает небольшое количество узлов.

затраченных на них усилий, поскольку максимальный размер очереди экспоненциально превосходит максимальное число итераций циклов. Например, если в качестве максимального размера очереди принимается  $2^{16}$ , а количество элементов очереди обычно выражается тысячами, то простейшие из реализаций будут повторять цикл 15 раз, в то время как более сложные методы снижают число повторений цикла всего лишь до 11 или 12, при этом на поддержание максимального размера очереди и сигнальной метки требуется дополнительный расход ресурсов. С другой стороны, безосновательный выбор большого значения максимума может привести к тому, что на очередях малых размеров программы начнут работать медленнее, нежели рассчитывалось.

**Лемма 9.8.** *Построение биномиальной очереди с выполнением  $N$  операций вставок в первоначально пустую очередь требует выполнения  $O(N)$  сравнений в наихудшем случае.*

Для одной половины вставок (когда размер очереди представлен четным числом и 1-деревья отсутствуют) операции сравнения вообще не требуются; для половины оставшихся вставок (если нет 2-деревьев) требуется лишь одна операция сравнения; если нет 4-деревьев, требуется только 2 операции сравнения и т.д. Следовательно, общее число сравнений меньше  $0 \cdot N/2 + 1 \cdot N/4 + 2 \cdot N/8 + \dots < N$ . Что касается леммы 9.7, то нам нужна одна из модификаций из числа тех, которые рассматривались в упражнениях 9.61 и 9.62, чтобы получить в наихудшем случае время выполнения, не превышающее линейного.

Как уже упоминалось в разделе 4.8, мы не рассматривали вопросов распределения памяти при реализации операции *объединить* в программе 9.16. В силу этой причины при выполнении операции имеет место утечка памяти, так что в некоторых ситуациях она становится непригодной. Для исправления этого дефекта потребуется уделить соответствующее внимание вопросу распределения памяти под аргументы и возвращаемое значение функции, которая реализует операцию *объединить* (см. упражнение 9.65).

Для биномиальных очередей характерно высокое быстродействие, тем не менее, были предложены специальные структуры данных, которые обладали в теоретическом плане еще лучшими характеристиками, гарантировано обеспечивая постоянное время выполнения некоторых операций. Эта проблема вызывает интерес и предлагает разработчикам структур данных широкое поле деятельности. С другой стороны, практическое применение многих из этих, понятных только посвященным, структур весьма сомнительно, а мы должны знать, что от некоторых имеющих место ограничений эффективности можно избавиться только путем снижения времени выполнения некоторых операций на очередях по приоритетам, прежде чем пускаться на поиски сложных решений в области структур данных. И в самом деле, для практических применений предпочтение следует отдавать тривиальным структурам при отладке и при работе с очередями малых размеров; далее, для ускорения операций, если речь не идет о получении быстродействующей операции *объединить*, необходимо использовать сортирующие деревья. Наконец, биномиальными очередями следует пользоваться, если требуется получить логарифмическое время выполнения всех операций. Однако, принимая во внимание все сопутствующие факторы, приходим к выводу, что пакет очередей по приоритетам на базе биномиальных очередей является, несомненно, ценным вкладом в библиотеку программного обеспечения.

## Упражнения

- ▷ 9.54. Вычертить биномиальную очередь размера 29, воспользовавшись представлением в виде биномиального дерева.
- 9.55. Напишите программу построения представления биномиальной очереди в виде биномиального дерева заданного размера  $N$  (только узлы, соединенные ребрами, но не ключи).
- 9.56. Построить биномиальную очередь, которая получится, если ключи **E A S Y Q U E S T I O N** вставляются в первоначально пустую биномиальную очередь.
- 9.57. Построить биномиальную очередь, которая получится, если ключи **E A S Y** вставляются в первоначально пустую биномиальную очередь, и построить биномиальную очередь, которая получится, если вставить ключи **Q U E S T I O N** в первоначально пустую очередь. Затем выполните операцию *удалить наибольший* в обеих очередях и представьте результат. И наконец, представьте результат выполнения операции *объединить* применительно к полученным очередям.
- 9.58. Используя соглашения из упражнения 9.1, построить последовательность биномиальных очередей, полученных в результате того, что операции **PRIO\*R\*\*I\*T\*Y\*\*\*QUE\*\*\*U\*E** выполняются на первоначально пустой биномиальной очереди.
- 9.59. Используя соглашения из упражнения 9.1, построить последовательность биномиальных очередей, полученных в результате того, что операции **(((PRIO\*) + (R \*IT\*Y\*))\*\*\*)+ (QUE\*\*\*U\*E)** выполняются на первоначально пустой биномиальной очереди.
- 9.60. Доказать, что биномиальное дерево с  $2^n$  узлами имеет  $\binom{n}{i}$  узлов на уровне  $i$ , причем  $0 \leq i \leq n$ . (Этот факт и послужил причиной того, что подобного рода структура данных получила название *биномиального дерева*).
- 9.61. Построить такую программную реализацию биномиальной очереди, чтобы выполнялась лемма 9.7, внося изменение в тип данных *биномиальной очереди*, чтобы он содержал размер этой очереди с целью последующего использования этого значения для управления циклами.
- 9.62. Построить такую программную реализацию биномиальной очереди, чтобы выполнялась лемма 9.7. Использовать для этой цели сигнальный указатель, отмечающий точку, в которой циклы должны завершаться.
- 9.63. Реализовать операцию *вставить (insert)* для биномиальных очередей путем явного использованием одной лишь операции *объединить (join)*.
- 9.64. Реализовать операцию *изменить приоритет (change priority)* и *удалить (remove)* для биномиальных очередей. *Совет*: Потребуется добавить третью связь, которая указывает на узлы вверх по дереву.

- **9.65.** Добавить деструктор, конструктор копирования и перегруженную операцию присваивания в реализации биномиальной очереди (программы 9.13—9.16), приведенные в тексте книги, с целью разработки реализации АТД первого класса из упражнения 9.43. Написать программу-драйвер, которая сможет протестировать полученные интерфейс и реализацию.
- **9.66.** Эмпирическим путем сравнить биномиальные очереди с сортирующими деревьями с целью их использования в качестве основы для сортировки (как в программе 9.6) произвольно упорядоченных ключей при  $N = 1000, 10^4, 10^5$  и  $10^6$ . *Совет:* см. упражнение 9.37.
- **9.67.** Разработать метод обменной сортировки, аналогичной сортирующему дереву, но основанной на биномиальных очередях. *Совет:* см. упражнение 9.37.



## Поразрядная сортировка

**В**о многих приложениях сортировки ключи, которые используются для определения порядка следования записей в файлах, могут иметь сложную природу. Например, рассмотрим, насколько сложными являются ключи, используемые в телефонной книге или в библиотечном каталоге. Чтобы отделить все эти сложности от наиболее важных свойств методов сортировки, изучением которых мы занимались, ограничимся использованием только базовых операций сравнения двух ключей и обмена местами двух записей (все детали манипулирования ключами в этих функциях все это время оставались скрытыми) как абстрактным интерфейсом между методами сортировки и применениями этих методов из глав 6–9. В данной главе мы займемся изучением еще одной абстракции, отличной от рассмотренных выше, применительно к ключам сортировки. Например, довольно часто нет необходимости в обработке ключей в полном объеме на каждом этапе: чтобы найти телефонный номер какого-либо конкретного абонента вполне достаточно проверить несколько первых букв его фамилии, чтобы найти страницу, на которой находится искомый номер. Чтобы достигнуть такой же эффективности сортировочных алгоритмов, мы должны перейти от абстрактных операций, в рамках которых мы выполняли сравнение ключей, к абстракциям, в условиях которых мы разделяем ключи на последовательности порций фиксированных размеров или *байтов*. Двоичные числа представляют собой последовательности битов, строки — последовательности символов, десятичные числа — последовательности цифр, и многие другие типы ключей (но далеко не все) можно рассматривать под таким же углом

зрения. Методы сортировки, построенные на обработке чисел по одной порции за раз, называются *поразрядными (radix) методами сортировки*. Эти методы не только выполняют сравнение ключей: они обрабатывают и сравнивают соответствующие части ключей.

Алгоритмы поразрядной сортировки рассматривают ключи как числа, представленные в системе счисления с основанием  $R$  при различных значениях  $R$  (*основание системы счисления*), и работают с отдельными цифрами чисел. Например, если машина в почтовом отделении обрабатывает пачку пакетов, каждый из которых помечен десятичным числом из пяти цифр, она распределяет эту пачку на десять отдельных стопок: в одной стопке находятся пакеты, номера которых начинаются с 0, в другой — пакеты с номерами, начинающимися с 1, в третьей — с 2 и т.д. При необходимости каждая из стопок может быть подвергнута отдельной обработке с применением того же метода к следующей цифре или более простого метода, если в стопке осталось всего лишь несколько пакетов. Если бы перед нами стояла задача распределения пакета в стопки в порядке от 0 до 9 и в том порядке отсортировать каждую стопку, то будет упорядочен весь пакет. Эта процедура является простым примером поразрядной сортировки с  $R = 10$ , именно такой метод сортировки чаще всего выбирается как наиболее подходящий в различных практических приложениях, в которых ключами являются десятичные числа, содержащие от 5 до 10 цифр, например, почтовые коды, телефонные номера или коды службы социальной защиты. Подробно этот метод рассматривается в разделе 10.3.

Для различных приложений подходят различные основания системы счисления  $R$ . В этой главе мы главным образом сосредоточимся на ключах, представленных в виде целых чисел и строк, для сортировки которых широко применяются методы поразрядной сортировки. Для целых чисел в силу того обстоятельства, что они представлены в компьютерах в виде двоичных чисел, мы чаще других будем выбирать для работы  $R = 2$  или одну из степеней числа 2, поскольку такой выбор позволяет разложить ключи на независимые друг от друга порции. Что касается ключей, в состав которых входят строки символов, мы используем  $R = 128$  или  $R = 256$ , приравнивающий основание системы счисления к размеру байта. Помимо такого рода прямых приложений мы можем в конечном итоге рассматривать фактически *все*, что может быть представлено в цифровом компьютере как двоичное число, благодаря чему мы имеем возможность сориентировать многие приложения сортировки на использование различных типов ключей с тем, чтобы сделать возможной использование поразрядной сортировки для упорядочения ключей, представляющих собой двоичные числа.

В основе алгоритмов поразрядной сортировки лежит абстрактная операция извлечь из ключа  $I$ -ю цифру. К счастью, в C++ существуют низкоуровневые операции, благодаря которым можно реализовать такие действия просто и эффективно. Этот факт очень важен, поскольку многие другие языки программирования (например, Pascal), поощряя написание машинно-независимых программ, намеренно создают трудности в написании программ, которые зависят от способа представления чисел в конкретной машине. В таких языках достаточно трудно реализовать многие типы методов побитовых манипуляций, которые хорошо подходят для большинства компьютеров. В частности, на текущий момент поразрядную сортировку можно считать

жертвой этой "прогрессивной" философии. Однако разработчики С и С++ признают, что подобного рода манипуляции с битами часто бывают весьма полезными, и при реализации поразрядной сортировки мы имеем возможность воспользоваться низкоуровневыми языковыми средствами.

Требуется также надежная поддержка со стороны аппаратных средств; мы не можем считать такую поддержку само собой разумеющимся фактом. Некоторые машины (как старые, так и новые модели) предлагают эффективные способы представления малых значений данных, в то же время характеристики производительности других типов машин (как старых, так и новых моделей) на этих операциях существенно снижаются. Поскольку алгоритм поразрядной сортировки достаточно просто выражается в терминах операции извлечения конкретной цифры, задача достижения максимальной производительности алгоритма поразрядной сортировки может оказаться весьма интересным вхождением в среду аппаратного и программного обеспечения.

Существуют два принципиально различных базовых подхода к поразрядной сортировке. Первый класс методов составляют алгоритмы, которые анализируют значение цифр в ключах в направлении слева направо, при этом первыми обрабатываются *наиболее значащие цифры*. Такие методы в общем случае называются *поразрядной сортировкой MSD (most significant digit radix sort — Поразрядная сортировка сначала по старшей цифре)*. Поразрядная сортировка MSD привлекательна прежде всего тем, что в этом случае анализируется минимальный объем информации, необходимый для выполнения сортировки (см. рис. 10.1). Поразрядная сортировка MSD обобщает понятие быстрой сортировки, поскольку она выполняется за счет разделения сортируемого файла в соответствии со старшими цифрами ключей, после чего тот же метод применяется к подфайлам в режиме рекурсии. В самом деле, в условиях, когда в качестве основания системы счисления выбрана 2, мы реализуем поразрядную сортировку MSD тем же способом, что и быструю сортировку. Во втором классе методов поразрядной сортировки используется другой принцип: они анализируют цифры ключей в направлении справа налево, работая с начала с наименее значащей цифрой. Эти методы в общем случае называются *поразрядной сортировкой LSD (least significant digit radix sort — Поразрядная сортировка сначала по младшей цифре)*. Поразрядная сортировка LSD в какой-то степени противоречит интуиции, поскольку основную часть процессорного времени затрачивается на обработку цифр, которые не могут повлиять на результат, однако эта проблема легко решается, и этот почтенный метод выбирается в качестве базового во многих приложениях сортировки.

.396465048	.015583409	.0
.353336658	.159072306	.1590
.318693642	.159369371	.1593
.015583409	.269971047	.2
.159369371	.318693642	.31
.691004885	.353336658	.35
.899854354	.396465048	.39
.159072306	.538069659	.5
.604144269	.604144269	.60
.269971047	.691004885	.69
.538069659	.899854354	.8

#### РИСУНОК 10.1. ПОРАЗРЯДНАЯ СОРТИРОВКА

Несмотря на то что между 0 и 1 в рассматриваемом списке находятся 11 чисел (столбец слева), содержащих в совокупности 99 цифр, мы можем установить на них порядок (столбец в центре) путем анализа всего лишь 22 цифр (столбец справа).

## 10.1. Биты, байты и слова

Ключевое условие для понимания сути поразрядной сортировки состоит в признании того, что (i) компьютеры в общем случае ориентированы на обработку групп битов, называемых *машинными словами*, которые в свою очередь часто объединяются в небольшие фрагменты, называемые *байтами*; (ii) ключи сортировки обычно *также* организуются в последовательности байтов, (iii) короткие последовательности байтов могут также служить индексами массивов или машинными адресами. Поэтому нам будет удобно работать со следующими абстракциями.

**Определение 10.1.** *Байт представляет собой последовательность битов фиксированной длины, строка есть последовательность байтов переменной длины, слово есть последовательность байтов фиксированной длины.*

В зависимости от контекста *ключом* в поразрядной сортировке может быть слово или строка. Некоторые из алгоритмов поразрядной сортировки, которые предстоит рассмотреть в настоящей главе, используют свойство ключей принимать фиксированную длину (слова), другие разрабатываются с целью приспособиться к ситуации, когда ключи имеют переменную длину (строки).

Типичная машина оперирует 8-разрядными байтами и 32- и 64-разрядными словами (фактические значения можно найти в заголовочном файле `<limits.h>`), однако удобнее иметь возможность рассматривать также и некоторые другие размеры байтов и слов (в общем случае небольшие кратные целые конструктивных машинных размеров или их части). Мы используем в качестве числа разрядов в слове и числа разрядов в байте машинно-зависимые и зависящие от приложений константы, например:

```
const int bitword = 32;
const int bitsbyte = 8;
const int bytesword = bitword/bitsbyte;
const int R=1 << bitsbyte;
```

В эти определения для последующего использования, когда мы начнем рассматривать поразрядные сортировки, включается также константа *R*, представляющая число различных значений байтов. Пользуясь этими определениями мы в общем случае предполагаем, что **bitword** является кратным **bitsbyte**, что число битов в машинном слове не меньше (обычно равно) **bitword**, и что байты допускают индивидуальную адресацию. В различных компьютерах реализованы различные соглашения, касающиеся ссылок на их биты и байты, для целей наших рассуждений мы будем считать, что биты в слове перенумерованы слева направо, от 0 до **bitword-1**, и байты в слове перенумерованы слева направо, от 0 до **bytesword-1**. В обоих случаях мы полагаем, что нумерация производится от наибольшего значения к наименьшему значению.

В большинстве компьютеров реализованы битовые операции *и* (*and*) и *сдвиг* (*shift*), которыми мы можем воспользоваться для извлечения отдельных байтов из слов. В C++ мы можем прямо написать операции извлечения *B*-ого байта из двоичного *A* слова следующим образом:

```
inline int digit(long A, int B)
{ return (A >> bitsbyte*(bytesword-B-1) & (R-1); }
```

Например, данная макрокоманда извлекает байт 2 (третий байт) 32-разрядного числа путем сдвига вправо на  $32 - 3 * 8 = 8$  позиций с последующим использованием маски 000000000000000000000000000011111111 с целью обнуления всех разрядов за исключением искомого байта, занимающего 8 разрядов справа.

Многие машины организованы таким образом, что в качестве размера байта взято основание одной из систем счисления, в силу чего обеспечивается быстрая выборка нужных битов в рамках одного доступа. Эта операция непосредственно поддерживается C++ строками в C-стиле:

```
inline int digit(char* A, int B)
{ return A[B]; }
```

Если мы используем структуру-оболочку, подобную рассмотренной в разделе 6.8, мы будем записывать:

```
inline int digit(Item& A, int B)
{ return A.str[B]; }
```

Этот подход может быть использован также и для чисел, хотя различие схем представления чисел может сделать эти программные коды непереносимыми. В любом случае мы должны сознавать, что такого рода операции доступа к отдельным байтам в некоторых вычислительных средах могут быть реализованы на базе операций сдвига и маскирования, подобных рассмотренным в предыдущих параграфах.

На другом уровне абстракции, несколько отличном от рассматриваемого, мы можем представлять себе ключи как числа и байты как цифры. Если выбрано (ключ представлен как) число, то базовая операция, необходимая для реализации поразрядной сортировки есть извлечение из числа цифры. Когда мы выбираем в качестве основания для системы счисления степень 2, цифрами являются группы битов, к которым мы легко можем получить доступ используя рассмотренные выше макросы. В самом деле, основной причиной того, что в качестве основания системы счисления используется степень 2 является то, что операция доступа к группе битов не требует больших затрат. В некоторых вычислительных средах можно выбирать другие основания систем счисления. Например, если  $a$  есть положительное целое число, то  $b$ -я цифра представления  $a$  в системе счисления  $R$  есть

$$\lfloor a/R^b \rfloor \bmod R.$$

На машинах, предназначенных для высокопроизводительных числовых вычислений, эти вычисления могут выполняться для произвольного значения  $R$  так же быстро, как и для случая  $R = 2$ .

Еще одна точка зрения призывает рассматривать ключи как числа в диапазоне от 0 до 1, при этом подразумевается, что десятичная точка находится слева, как показано на рис. 10.1. В этом случае  $b$ -ой цифрой числа  $a$  будет

$$\lfloor aR^b \rfloor \bmod R.$$

Если мы работаем на машине, на которых можно эти операции выполнять эффективно, то мы можем использовать их в качестве основы для поразрядной сортировки. Такая модель применяется в тех случаях, когда ключи имеют переменную длину, например, в случае строк символов.

Таким образом, в оставшейся части главы мы будем рассматривать ключи как числа в системе счисления с основанием  $R$  (конкретное значение  $R$  не указывается) и употреблять операцию **digit** для доступа к отдельным цифрам ключей полагая, что мы сможем разработать быстродействующую программную реализацию операции **digit** для конкретных компьютеров.

**Определение 10.2.** *Ключ есть число в системе счисления с основанием  $R$ , цифры которого пронумерованы слева (начиная с 0).*

В свете только что рассмотренных примеров, для нас удобнее считать, что эта абстракция допускает эффективные реализации для множества приложений на большей части компьютеров, хотя мы должны соблюдать определенную осторожность, ибо конкретная реализация может быть эффективной только в рамках заданной программной и аппаратной среды.

Мы полагаем, что ключи достаточно длинные, так что операция извлечения из них битов имеет смысл. Если же ключи короткие, то мы можем применять метод подсчета индексных ключей, рассмотренных в главе 6. Напомним, что этот метод позволяет сортировать  $N$  ключей, представляющие собой целые числа, принимающие значения в диапазоне от 0 до  $R - 1$  за линейное время, используя для этой цели одну вспомогательную таблицу размером  $R$  для расчетов и другую таблицу размером  $N$  для перепорядочения  $N$  записей. Следовательно, если мы можем себе позволить поддержку таблицы размером  $2^w$ , то сортировку  $w$ -разрядных ключей легко выполнить за линейное время. В самом деле, расчеты, связанные подсчетом индексных ключей, лежат в основе базовых методов поразрядной сортировки MSD и LSD. Поразрядная сортировка вступает на передний план, когда ключи обладают достаточной длиной (скажем,  $w = 64$ ), когда использование таблицы размером  $2^w$  не является целесообразным.

## Упражнения

- ▷ 10.1. Сколько нужно цифр для представления 32-разрядного числа в системе счисления, основанием которой является 256? Опишите, как можно извлечь каждую цифру этого числа. Ответьте на этот вопрос для случая, когда основанием системы счисления будет число  $2^{16}$ .
- ▷ 10.2. Для  $N = 10^3$ ,  $10^6$  и  $10^9$  дать наименьший размер байта, который позволит представить любое число в диапазоне от 0 до  $N$  в виде слова из 4 байтов.
- 10.3. Перегрузить операцию **operator<**, используя абстракцию **digit** (чтобы, например, можно было выполнять эмпирические исследования, сравнивая алгоритмы из глав 6 и 9 с методами, описанными в данной главе, используя одни и те же данные).
- 10.4. Разработать и выполнить эксперимент, сравнивающий затраты ресурсов на извлечение цифр с использованием в одном случае операции сдвига и в другом случае — арифметических операций, реализованных на вашей машине. Сколько цифр вы можете извлечь за секунду, используя каждый из двух этих методов? *Указание:* будьте осторожны, ваш компилятор может преобразовать арифметические операции в операции сдвига битов и наоборот.

- **10.5.** Напишите программу, которая при заданном наборе произвольных десятичных чисел ( $R = 10$ ), равномерно распределенных на интервале от 0 до 1, будет вычислять количество операций сравнения чисел, необходимых для их сортировки в смысле рис. 10.1. Выполните эту программу для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **10.6.** Выполнить упражнение 10.5 для  $R = 2$ , используя 32-разрядные величины.
- **10.7.** Выполнить упражнение 10.5 для случая, когда числа подчиняются распределению Гаусса.

## 10.2. Двоичная быстрая сортировка

Предположим, что мы можем переупорядочить записи в файле таким образом, что все те ключи, которые начинаются с бита 0, идут раньше всех тех, которые начинаются с бита 1. Далее мы можем воспользоваться методом рекурсивной сортировки, который является одним из вариантов быстрой сортировки (см. главу 7): разбиение файла этим способом с последующей независимой сортировкой двух полученных подфайлов. Чтобы переупорядочить файл, выполните просмотр слева с целью обнаружить ключ, который начинается с бита 1, затем продолжайте просмотр справа с целью найти ключ, который начинается с бита 0, поменяйте ключи местами и продолжайте процесс до тех пор, пока указатели не пересекутся. Этот метод в литературе (включая и более ранние издания данной книги) часто называют *поразрядной обменной сортировкой* с тем, чтобы подчеркнуть, что это прежде всего простой вариант алгоритма, изобретенного Хоаром, несмотря на то что он был открыт раньше быстрой сортировки (см. раздел ссылок).

Программа 10.1 представляет полную реализацию этого метода. Применяемый в ней процесс разбиения по существу лишь немногим отличается от разбиения, реализованного в программе 7.2, за исключением того, что в рассматриваемом случае в качестве разделяющего элемента используется число  $2^b$ , а не некоторый ключ из файла. Поскольку числа  $2^b$  может и не быть в файле, то нет гарантии того, что конкретный элемент будет помещен в свою окончательную позицию в процессе разбиения. Рассматриваемый алгоритм отличается от алгоритма быстрой сортировки, поскольку рекурсивные вызовы выполняются для ключей, имеющих на 1 бит меньше. Это различие существенно влияет на эффективность алгоритма. Например, если имеет место вырожденное разбиение файла из  $N$  элементов, то произойдет рекурсивный вызов для подфайла размером  $N$  для ключей, имеющих размер на 1 разряд меньше. Следовательно, число таких вызовов ограничено количеством разрядов в ключе. В противоположность этому, последовательное использование разделяющих значений, взятых не из самого файла в условиях стандартной быстрой сортировки может привести к возникновению бесконечного рекурсивного цикла.

Как и в случае стандартной быстрой сортировки, возможны различные варианты реализации внутренних циклов. В программе проверка, не пересеклись ли значения указателей, включена в оба внутренних цикла. Такая проверка приводит к дополнительному обмену местами для случая, когда  $i = j$ , чего можно избежать путем применения `break`, что, собственно говоря, и сделано в программе 7.2, хотя в рассматриваемом случае обмен элемента `a[i]` на самого себя вполне безобиден. Другой альтернативой является использование служебной метки.

## Программа 10.1. Двоичная быстрая сортировка

Эта программа производит разделение файла по старшим разрядам, после чего выполняет рекурсивную сортировку полученных подфайлов. Переменная *d* фиксирует анализируемый бит, начиная с 0 (крайний левый). Процесс разделение заканчивается, когда *j* становится равным *i*, т.е. когда все элементы справа от *a[i]* имеют 1 в *d*-й позиции, а все элементы слева от *a[i]* имеют 0 в *d*-й позиции. Сам элемент *a[i]* имеет в *d*-й позиции 1, если только у всех ключей файла в позиции *d* не стоит 0. Дополнительная проверка, выполняемая непосредственно по окончании цикла, охватывает и этот случай.

```
template <class Item>
void quicksortB(Item a[], int l, int r, int d)
{ int i = l, j = r;
  if (r <= l || d > bitsword) return;
  while (j != i)
  {
    while (digit(a[i], d) == 0 && (i < j)) i++;
    while (digit(a[j], d) == 1 && (j > i)) j--;
    exch(a[i], a[j]);
  }
  if (digit(a[r], d) == 0) j++;
  quicksortB(a, l, j-1, d+1);
  quicksortB(a, j, r, d+1);
}
template <class Item>
void sort(Item a[], int l, int r)
{ quicksortB(a, l, r, 0); }
```

Рисунок 10.2 дает описание работы программы 10.1 на примере простого учебного файла и сравнивает ее с быстрой сортировкой, представленной на рис. 7.1. Этот рисунок показывает, каким является движение данных, но не объясняет, почему производятся те или иные перемещения — это зависит от двоичного представления соответствующих ключей. Более подробное представление этого же примера дано на рис. 10.3. В рамках этого примера предполагается, что буквы кодируются 5-разрядными кодами, при этом для *i*-й буквы алфавита используется двоичное представление числа *i*. Подобное кодирование представляет собой упрощенную версию настоящих кодов символов, для которых используется большее число битов (7.8 и даже 16) с тем, чтобы охватить большее число символов (буквы верхнего и нижнего регистров, числа и специальные символы).

Для ключей в виде полных слов, состоящих из произвольных совокупностей битов, отправной точкой для программы 10.1 должны служить крайние левые биты слова или бит 0. В общем случае выбираемая исходная точка напрямую зависит от приложения, от

```
A S O R T I N G E X A M P L E
A E O L M I N G E A X T P R S
                                     S T P R X
                                     S R P T
                                     P R S
                                     R S
A E A E G I N M L O
                                     I N M L O
                                     L M N O
                                     N O
                                     L M
A A E E G
  E E G
  E E
  E E
A A
A A
A A
A A E E G I L M N O P R S T X
```

### РИСУНОК 10.2. ПРИМЕР ДВОИЧНОЙ БЫСТРОЙ СОРТИРОВКИ

Разделение по старшему разряду еще не гарантирует того, что хотя бы одно значение станет на свое окончательное место, оно лишь обеспечивает, что все ключи с 0 в старших разрядах предшествуют ключам с 1 в старших разрядах. Мы можем сравнить эту диаграмму с рис. 7.1, иллюстрирующим быструю сортировку, хотя то, как выполняется метод разделения, совершенно непонятен, если для ключей не выбран двоичный метод представления. На рис. 10.3 приводятся подробности, благодаря которым становится ясно, по каким позициям производится разделение.



количества битов в машинном слове, от представления целых чисел и отрицательных чисел в той или иной машине. Для однобуквенных 5-разрядных ключей на рис. 10.2 и 10.3 отправной точкой на 32-разрядной машине должен быть бит 27.

Этот пример обращает внимание на потенциальную проблему, возникающую при использовании двоичной быстрой сортировки в реальных ситуациях: вырожденные разделения (когда все ключи имеют одно и то же значение разряда, по которому производится разделение) могут встречаться довольно часто. Такая ситуация при сортировке малых чисел (старшие разряды принимают значение 0), таких как в рассматриваемых нами примерах, не является чем-то необычным. Эта же проблема возникает при использовании ключей, состоящих из символов: например, предположим, что мы строим 32-разрядные ключи из четырех символов, каждый из которых представлен в стандартном 8-разрядном коде, объединяя их в единое целое. Тогда появление вырожденных разделений возможно в начальной позиции каждого символа, поскольку, например, все буквы нижнего регистра начинаются с одних и тех же битов для большинства кодов символов. Эта проблема является типовой по своим последствиям, с которыми приходится сталкиваться при сортировке кодированных данных, подобные проблемы возникают и в других видах поразрядной сортировки.

A	00001	A	00001	A	00001	A	00001	A	00001	A	00001
S	10011	E	00101	E	00101	A	00001	A	00001	A	00001
O	01111	O	01111	A	00001	E	00101	E	00101	E	00101
R	10010	L	01100	E	00101	E	00101	E	00101	E	00101
T	10100	M	01101	G	00111	G	00111	G	00111	G	00111
I	01001	I	01001	I	01001	I	01001	I	01001	I	01001
N	01110	N	01110	N	01110	N	01110	L	01100	L	01100
G	00111	G	00111	M	01101	M	01101	M	01101	M	01101
E	00101	E	00101	L	01100	L	01100	N	01110	N	01110
X	11000	A	00001	O	01111	O	01111	O	01111	O	01111
A	00001	X	11000	S	10011	S	10011	P	10000	P	10000
M	01101	T	10100	T	10100	T	10100	R	10010	R	10010
P	10000	P	10000	P	10000	P	10000	S	10011	S	10011
L	01100	R	10010	R	10010	R	10010	T	10100	T	10100
E	00101	S	10011	X	11000	X	11000	X	11000	X	11000

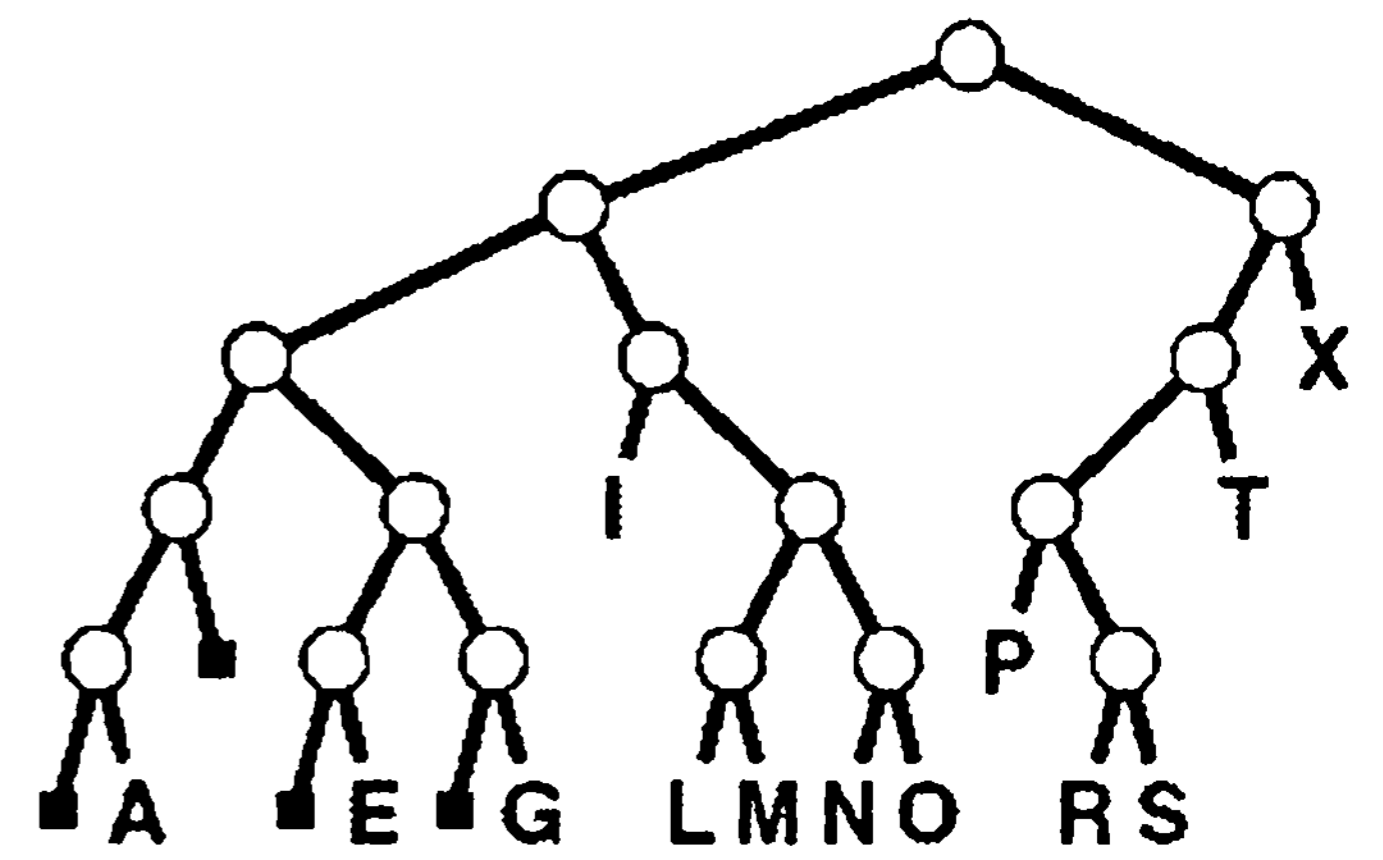
**РИСУНОК 10.3. ПРИМЕР ДВОИЧНОЙ БЫСТРОЙ СОРТИРОВКИ (ПОКАЗАНЫ ЗНАЧЕНИЯ РАЗРЯДОВ КЛЮЧЕЙ)**

*Мы построили эту диаграмму на основании рис. 10.2 путем перевода значений ключей в их двоичные коды и сжатия таблицы, что позволяет показать, как производятся сортировки независимых подфайлов, как если бы они выполнялись параллельно, при этом столбцы и ряды меняются местами. На первой стадии файл разбивается на подфайл, все ключи которого начинаются с 0, и на подфайл, все ключи которого начинаются с 1. Затем первый подфайл разбивается на подфайл, все ключи которого начинаются с 00 и на подфайл, все ключи которого начинаются с 01; независимо от этого процесса в некоторый другой момент времени второй подфайл разбивается на подфайл, все ключи которого начинаются с 10 и на подфайл, все ключи которого начинаются с 11. Этот процесс прекратится только тогда, когда все разряды будут исчерпаны (для дублированных ключей в рассматриваемом примере) или когда размер подфайла становится равным 1.*

После того как выяснится, что один из ключей отличается от всех остальных значением левого разряда, никакие другие разряды больше не анализируются. Это свойство в одних ситуациях выступает как достоинство, в других — как недостаток. Когда ключи представляют собой случайные совокупности битов, анализу подвергаются только  $\lg N$  битов, что намного меньше числа битов в ключах. Этот факт рассматривается в разделе 10.6, а также в упражнении 10.5 и рис. 10.1. Например, сортировка файла из 1000 записей с произвольно организованными ключами может потребовать анализа всего лишь 10 или 11 битов каждого ключа (даже если такими ключами являются, скажем, 64-разрядные ключи).

С другой стороны, все биты одинаковых ключей также проверяются. Поразрядная сортировка не способна работать хорошо на файлах, которые содержат большое число дублированных ключей достаточно большой длины. Двоичная быстрая сортировка и стандартные методы работают достаточно быстро, если сортируемые ключи представляют собой наборы битов случайной природы (различие между ними определяется главным образом разницей стоимости операций извлечения битов и сравнения), но в то же время стандартный алгоритм быстрой сортировки легче настроить на обработку неслучайных совокупностей ключей, а трехпутевая быстрая сортировка идеально подходит для случаев, когда преобладают дублированные ключи.

Как и в случае быстрой сортировки, структуру разделения удобно описывать в виде бинарного дерева (аналогично тому, как она представлена на рис. 10.4): корень дерева соответствует подфайлу, подвергающемуся сортировке, а два его поддеревя — двум подфайлам, полученным в результате выполнения разделения. Мы, по крайней мере, знаем, что в результате выполнения стандартной быстрой сортировки одна из записей помещается в процессе разделения в свою окончательную позицию, следовательно, помещаем этот ключ в корневой узел; мы знаем также, что в условиях бинарной сортировки ключи попадают в свои окончательные позиции, только когда мы доходим до подфайлов размером 1, либо когда все разряды ключа исчерпаны; таким образом, мы помещаем эти ключи на нижний уровень дерева. Такая структура назы-



**РИСУНОК 10.4. ДЕРЕВО РАЗДЕЛЕНИЯ ДЛЯ ДВОИЧНОЙ БЫСТРОЙ СОРТИРОВКИ**

*Это дерево описывает структуру разделения для двоичной быстрой сортировки, соответствующей рис. 10.2 и 10.3. Поскольку нет гарантии того, что какие-либо элементы займут свои окончательные позиции, ключи соответствуют внешним узлам дерева. Такая структура обладает следующим свойством: продвигаясь по пути от корня к конкретному ключу, принимая 0 за левую ветвь и 1 за правую ветвь, получаем значения старших разрядов этого ключа. Именно эти значения и отличают данный ключ от всех остальных во время сортировки. Маленькие черные квадраты представляют нулевые разделения (когда все ключи переходят на другую сторону, поскольку значения старших разрядов совпадают). В условиях рассматриваемого примера это может случиться только в окрестности нижнего уровня дерева, однако возможно и на более высоких уровнях: например, если I или X в число таких ключей не входили, то их узлы на чертеже будут заменены нулевым узлом. Обратите внимание на тот факт, что дублированные ключи (A и E) не могут быть разделены (сортировка поставит их в один подфайл только после исчерпания всех их битов).*

вается бинарным деревом (trie); свойства такого бинарного дерева подробно анализируются в главе 15. Например, одно из таких свойств, представляющих несомненный интерес, заключается в том, что структура бинарного дерева полностью определяется значениями ключей, а не порядком их следования.

Разделы, получаемые в процессе выполнения быстрой сортировки, зависят от двоичного представления и числа сортируемых элементов. Например, если файлы представляют собой случайные перестановки целых чисел, меньших  $171 = 10101011_2$ , то разделение по первому биту эквивалентно разбиению по значению 128, так что получаем неравные подфайлы (один файл размером 128, а другой размером 43). Ключи на рис. 10.5 суть произвольные 8-разрядные значения, таким образом, этот эффект в рассматриваемом случае не имеет места, однако он достоин того, чтобы на него обратить внимание сейчас, чтобы он не стал для неприятным сюрпризом, когда мы столкнемся с ним на практике.

Мы можем внести усовершенствования в базовую рекурсивную реализацию, представленную программой 10.1, за счет отказа от рекурсии и обработки подфайлов небольших размеров другим способом, как это делалось в случае стандартной быстрой сортировки в главе 7.

## Упражнения

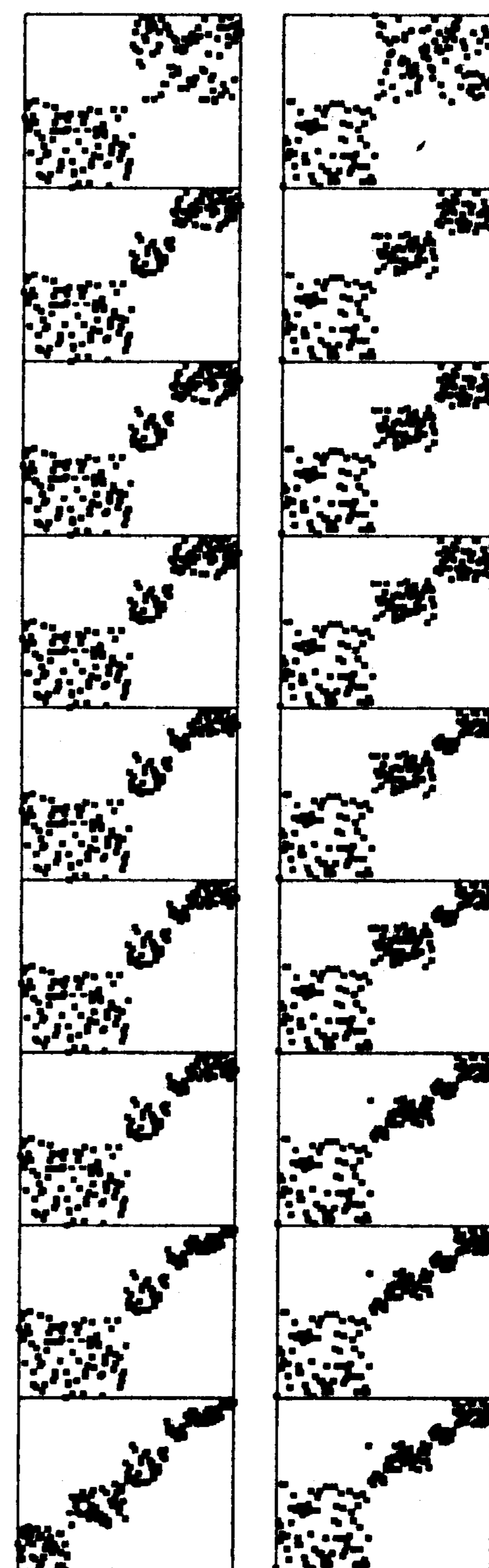
▷ **10.8.** Начертите дерево в стиле рис. 10.2, которое соответствует процессу деления в поразрядной быстрой сортировке, для ключей `E A S Y Q U E S T I O N`.

**10.9.** Сравните число операций обмена местами, используемых при двоичной быстрой сортировке с аналогичным показателем обычной быстрой сортировки для 3-разрядных двоичных чисел 001, 011, 101, 110, 000, 001, 010, 111, 110, 010.

○ **10.10.** Почему сортировка меньшего из двух подфайлов первым менее важна в случае двоичной быстрой сортировки, чем в случае обычной быстрой сортировки?

○ **10.11.** Опишите, что произойдет на втором уровне деления (когда левый подфайл подвергается разделению и когда правый подфайл подвергается разделению), в случае применения двоичной быстрой сортировки для упорядочения случайных перестановок неотрицательных целых чисел, меньших 171.

**10.12.** Написать программу, которая на проходе, предшествующем процессу обработки, определяет число старших разрядов, на которых все ключи равны, затем вызывает



**РИСУНОК 10.5.**  
**ДИНАМИЧЕСКИЕ**  
**ХАРАКТЕРИСТИКИ**  
**ДВОИЧНОЙ БЫСТРОЙ**  
**СОРТИРОВКИ НА**  
**КРУПНОМ ФАЙЛЕ**

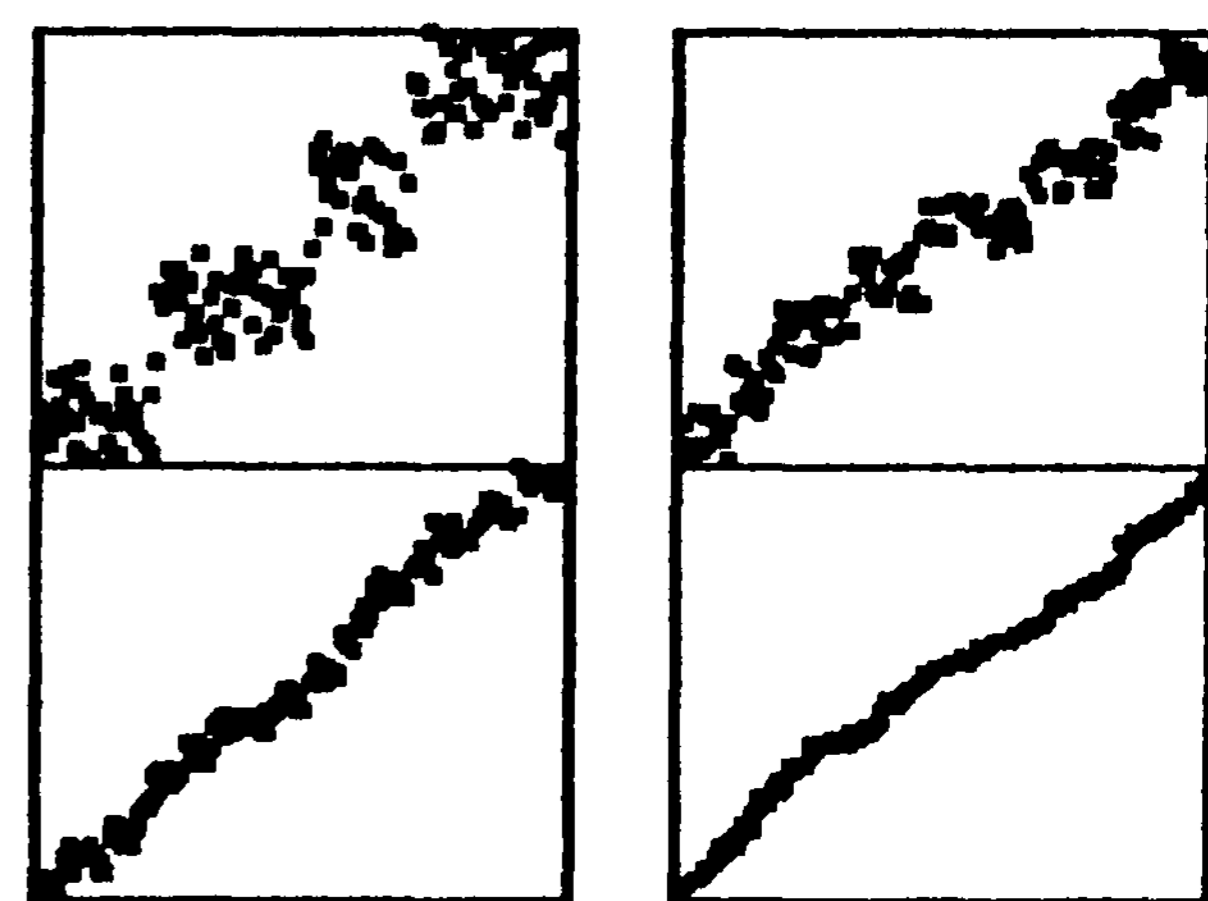
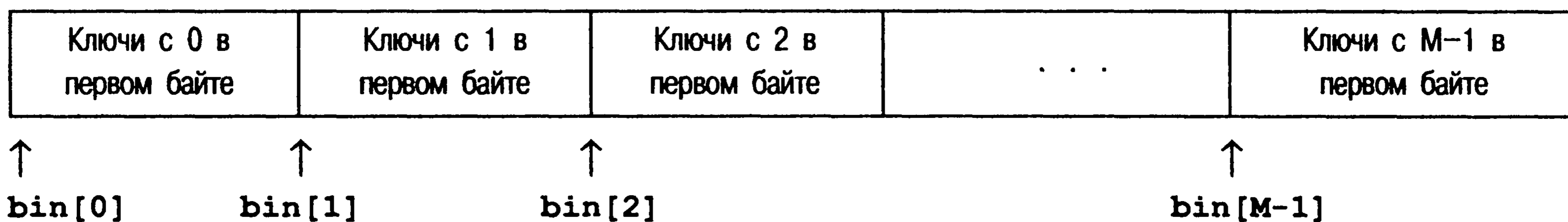
*Разбиения на части при двоичной быстрой сортировке менее чувствительны к порядку расположения ключи, чем в условиях стандартной быстрой сортировки. В рассматриваемом случае выполнение этой процедуры на двух различных 8-разрядных файлах с произвольной организацией приводят к практически идентичным профилям разделений.*

процедуру двоичной быстрой сортировки, которая модифицирована таким образом, что игнорирует эти разряды. Сравнить время выполнения вашей программы со временем выполнения стандартной реализации для  $N = 10^3, 10^4, 10^5$  и  $10^6$ , когда в качестве входных данных используется 32-разрядные слова следующего формата: крайние правые 16 битов — это равномерно распределенные случайные значения, а все крайние левые 16 битов суть 0 за исключением ситуаций, когда в позициях  $i$  проставляется 1, если имеется  $i$  единиц в правой половине.

**10.13.** Внести изменения в двоичную быструю сортировку с целью непосредственного обнаружения ситуаций, когда все ключи равны. Сравните время выполнения вашей программы с аналогичным показателем для стандартной реализации при  $N = 10^3, 10^4, 10^5$  и  $10^6$  для случая, когда входными данными служат данные, описанные в упражнении 10.12.

## 10.3. Поразрядная сортировка MSD

Использование всего лишь 1 разряда при поразрядной быстрой сортировке заставляет нас рассматривать ключи как числа в двоичной системе счисления и рассматривать сначала наиболее значащие числа. Обобщая предположим, что мы хотим отсортировать числа, представленные в системе счисления по основанию  $R$ , рассматривая в первую очередь наиболее значащие байты. Чтобы сделать это, необходимо разделить массив по крайней мере на  $R$ , а не на 2, различных частей. По традиции будем называть эти разделы *корзинами* или *ведрами* и будем представлять себе рассматриваемый алгоритм как группу из  $R$  корзин, по одной на каждое возможное значение первой цифры, как показано на следующей диаграмме:



**РИСУНОК 10.6. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ ПОРАЗРЯДНОЙ СОРТИРОВКИ MSD**

*Всего лишь одна стадия поразрядной сортировки MSD почти полностью решает задачу упорядочения, как показывает рассматриваемый пример файла произвольной организации, состоящий из 8-разрядных целых чисел. Первая стадия поразрядной сортировки MSD по двум старшим разрядам (слева) делит исходный файл на четыре подфайла. На следующей стадии каждый такой подфайл делится на четыре подфайла. Поразрядная сортировка MSD по трем старшим разрядам (справа) делит файл на восемь подфайлов за один проход, на котором также выполняются распределение и подсчет. На следующем уровне каждый из этих подфайлов снова разбивается на восемь частей, при этом в каждой такой части содержится всего лишь несколько элементов.*

Мы выполняем проход по всем ключам, распределяя их по корзинам, затем выполняем сортировку содержимого корзины по ключам с байтами, которые меньше исходных на 1.

На рис. 10.6 показан пример поразрядной сортировки MSD на произвольных перестановках целых чисел. В отличие от двоичной быстрой сортировки этот алгоритм может привести файл в относительный порядок достаточно быстро, даже после первого разделения, если значение основания системы счисления достаточно велико.

Как уже говорилось в разделе 10.2, одна из наиболее привлекательных особенностей поразрядной сортировки обусловлена ее интуитивным и прямолинейным характером, позволяющим ее адаптироваться к сортирующим приложениям, в условиях которых ключами являются символьные строки. Это ее свойство особенно ярко проявляется в C++ и других средах программирования, в которых обеспечена прямая поддержка обработки строк. В условиях поразрядной сортировки мы просто используем основание системы счисления, соответствующее размеру байта. Чтобы извлечь цифру, мы загружаем байт, чтобы переместиться к следующей цифре, мы увеличиваем на единицу указатель строки. Сейчас мы рассматриваем строки фиксированной длины, а немного погодя мы убедимся в том, что с ключами в виде строк переменной длины легко работать, используя те же базовые механизмы.

На рис. 10.7 показан пример поразрядной сортировки MSD трехбуквенных слов. Для простоты изложения в условиях этого примера за основание системы счисления принимается значение 26, хотя для большей части приложений мы выбираем с этой целью большее значение основания, в зависимости от того, как кодируются символы. Прежде всего, слова переупорядочиваются таким образом, что те из них, которые начинаются с символа **a**, идут раньше слов, начинающихся с буквы **b**, и т.д. Затем слова, начинающиеся с буквы **a**, подвергаются рекурсивной сортировке, далее производится сортировка слов, начинающихся с буквы **b**, и т.д. Как показывает пример, большая часть работы, связанной с сортировкой, приходится на разделения по первой букве, полученные после первого разделения подфайлы имеют небольшие размеры.

Как мы уже убедились на примере быстрой сортировки в главах 7 и разделе 10.2, а также при ознакомлении с сортировкой слиянием в главе 8, можно повысить эффективность большинства рекурсивных программ за счет использования простых алгоритмов для сортировки небольших по размерам файлов. Вопрос использования других методов для сортировки файлов

now	ace	ace	ace
for	ago	ago	ago
tip	and	and	and
ilk	bet	bet	bet
dim	cab	cab	cab
tag	caw	caw	caw
jot	cue	cue	cue
sob	dim	dim	dim
nob	dug	dug	dug
sky	egg	egg	egg
hut	for	few	fee
ace	fee	fee	few
bet	few	for	for
men	gig	gig	gig
egg	hut	hut	hut
few	ilk	ilk	ilk
jay	jam	jay	jam
owl	jay	jam	jay
joy	jot	jot	jot
rap	joy	joy	joy
gig	men	men	men
wee	now	now	nob
was	nob	nob	now
cab	owl	owl	owl
wad	rap	rap	rap
caw	sob	sky	sky
cue	sky	sob	sob
fee	tip	tag	tag
tap	tag	tap	tap
ago	tap	tar	tar
tar	tar	tip	tip
jam	wee	wad	wad
dug	was	was	was
and	wad	wee	wee

**РИСУНОК 10.7.**  
**ПРИМЕР**  
**ПОРАЗРЯДНОЙ**  
**СОРТИРОВКИ MSD**  
*Мы распределяем множество слов по 26 корзинам соответственно первой букве. Затем мы сортируем каждую корзину, применяя тот же метод, начиная со второй буквы.*

небольших размеров (корзины, содержащие небольшое число элементов) имеет большое значение для поразрядной сортировки, ибо их так много! Более того, мы можем настроить алгоритм соответствующим образом путем выбора значения  $R$ , поскольку существует простая зависимость: если  $R$  принимает чрезмерно большое значение, то большая часть стоимости сортировки приходится на инициализацию и проверку корзин, если наоборот,  $R$  недостаточно велико, то метод не использует своих потенциальных выгод, что достигается, если разделить исходный файл на максимально возможное число фрагментов. Мы вернемся к исследованию этих проблем в конце разделе 10.6.

Чтобы реализовать поразрядную сортировку MSD, необходимо обобщить методы разделения массивов, которые мы рассматривали при изучении реализаций быстрой сортировки в разделе 10.7. Эти методы, в основу которых положено перемещение указателей с противоположных концов массива навстречу друг другу, так что они встречаются где-то посередине, работают хорошо при необходимости получения двух или трех разделов, но не допускают немедленного обобщения. К счастью, *метод подсчета индексных ключей*, который рассматривался в главе 6 для целей сортировки файлов с ключами, принимающих значения в узком диапазоне, в рассматриваемом случае подходит как нельзя лучше. При этом используются таблицы значений и вспомогательные массивы, на первом проходе массива подсчитывается количество повторений каждой цифры старшего разряда. Эти значения показывают, где окажутся точки разделения. Затем, на втором проходе массива мы используем эти значения для перемещения элементов в соответствующие позиции вспомогательного массива.

Программа 10.2 реализует этот процесс. Ее рекурсивная структура обобщает структуру быстрой сортировки, так что мы снова оказываемся перед необходимостью решать проблемы, которые рассматривались в разделе 7.3. Должны ли мы сохранять наибольший из подфайлов во избежание излишней глубины рекурсии? Скорее всего, нет, поскольку глубина рекурсии ограничивается длиной ключа. Должны ли мы применять для сортировки подфайлов небольшого размера простые методы, такие как, например, сортировка простыми вставками? Конечно, поскольку существует огромное число таких методов.

Чтобы выполнить разбиение, программа 10.2 использует вспомогательный массив, размер которого равен размеру файла, подлежащего сортировке. В качестве альтернативы мы можем воспользоваться обменным методом подсчета индексных ключей (см. упражнения 10.17 и 10.18). Мы должны уделить большое внимание использованию пространства памяти, поскольку рекурсивные обращения могут привести к чрезмерному расходу памяти для локальных переменных. В программе 10.2 временный буфер для размещения ключей (`aux`) может быть глобальным, но массив, в котором хранятся число и местоположение позиций точек разделения (`count`) должны быть локальными.

Дополнительное пространство памяти для вспомогательного массива не представляют собой сколь-нибудь серьезной проблемы в условиях многочисленных приложений поразрядной сортировки применительно к длинным ключам или записям, поскольку для этих типов данных успешно применяется сортировка по указателю. Следовательно, дополнительное пространство памяти отводится для переупорядоче-

ния указателей, и в силу этого обстоятельства оно мало по сравнению с пространством, занимаемым самими ключами и записями (тем не менее, им не следует пренебрегать). Если памяти достаточно и главной заботой является обеспечение нужного быстродействия (обычная ситуация, характерная для использования поразрядной сортировки), можно также сэкономить время, затрачиваемое на копирование этого массива, путем рекурсивных операции переключения тех или иных параметров, как это делалось в рамках сортировки слиянием в разделе 8.4.

В случае ключей произвольной природы, количество ключей в каждой корзине (размеры подфайлов) после первого прохода в среднем составляет  $N/R$ . На практике ключи могут не быть произвольными (например, когда ключи представлены в виде строк, представляющих слова на английском языке, мы знаем, что лишь немногие из них начинаются с буквы *x* и совсем нет слов, которые начинались бы с букв *xx*), так что много корзин окажутся пустыми, а многие из непустых корзин будут содержать больше ключей, чем остальные (см. рис. 10.8). Несмотря на такое положение дел, процесс многопутевого разбиения в общем случае будет эффективным применительно к делению крупных сортируемых файлов на множество подфайлов меньших размеров.

### Программа 10.2. Поразрядная сортировка MSD

Мы разрабатывали эту программу, взяв за основу программу 8.17 (сортировка с подсчетом индексных ключей) путем замены ссылок на ключи на ссылки на цифры в ключах и добавления цикла на завершающей части программы, в котором выполняются рекурсивные вызовы каждого подфайла ключей, начиная с той же цифры. Для ключей переменной длины, оканчивающихся цифрой 0 (что характерно для строк в стиле C) первый оператор `if` и первый рекурсивный вызов пропускаются. Полученная реализация использует вспомогательный массив (`aux`), который достаточно велик, чтобы хранить копию входного файла.

```
#define bin(A) 1+count[A]
template <class Item>
void radixMSD(Item a[], int l, int r, int d)
{ int i, j, count[R+1];
  static Item aux[maxN];
  if (d > bytesword) return;
  if (r-l <= M) { insertion(a, l, r); return; }
  for (j = 0; j < R; j++) count[j] = 0;
  for (i = l; i <= r; i++)
    count[digit(a[i], d) + 1]++;
  for (j = 1; j < R; j++)
    count[j] += count[j-1];
  for (i = l; i <= r; i++)
    aux[1+count[digit(a[i], d)]] = a[i];
  for (i = l; i <= r; i++) a[i] = aux[i];
  radixMSD(a, l, bin(0)-1, d+1);
  for (j = 0; j < R-1; j++)
    radixMSD(a, bin(j), bin(j+1)-1, d+1);
}
```

Другой естественный путь реализации поразрядной сортировки MSD предполагает использование связанных списков. Для каждой корзины предусматривается один связанный список: на первом проходе по сортируемым элементам мы вставляем элемент в

соответствующий связный список, определяемый цифрой старшего разряда. Далее мы сортируем подсписки, после чего объединяем все эти связные списки в единое целое. Такой подход выливается в достаточно трудную задачу программирования (см. упражнение 10.36). Соединение связных списков требует отслеживания начала и конца каждого из этих списков, и, естественно, многие из них вполне могут оказаться пустыми.

Чтобы добиться высокой производительности конкретного приложения, использующего поразрядную сортировку, следует ограничить число пустых корзин за счет выбора соответствующего значения как основания системы счисления, так и значения, в соответствии с которым отсекаются подфайлы небольших размеров. В качестве конкретного примера предположим, что требуется отсортировать  $2^{16}$  (что-то около шестнадцати миллионов) 64-разрядных чисел. Чтобы поддерживать таблицу числа повторов, намного меньшую по размерам, чем размер файла, мы можем выбрать основание  $R = 2^{16}$ , для чего требуется проверка значений 16 разрядов, составляющих ключи. Но после первого разделения средний размер файла составит всего лишь  $2^8$ , и выбранное основание системы счисления для таких небольших файлов становится слишком большим. Положение усугубляет еще и то обстоятельство, что таких файлов может быть очень много: порядка  $2^{16}$  в рассматриваемом случае. Для каждого из этих  $2^{16}$  файлов процедура сортировки устанавливает значения  $2^{16}$  счетчиков равными 0, затем проверяет, что около  $2^8$  из них принимают ненулевые значения и так далее, что достигается ценой выполнения *по меньшей мере*  $2^{32}$  арифметических операций. Программа 10.2, которая реализована на основании предположения, что большая часть корзин не пуста, выполняет достаточно большое число арифметических операций для каждой пустой корзины (например, она выполняет рекурсивные вызовы для всех пустых корзин), так что для рассматриваемого примера время выполнения окажется очень большим. Соответствующим основанием для второго уровня может быть  $2^8$  или  $2^4$ . Короче говоря, вполне очевидно, что мы не должны использовать большие основания систем счисления в условиях поразрядной сортировки MSD файлов небольших размеров. Подробно мы рассмотрим этот вопрос в разделе 10.6, когда займемся исследованием характеристик различных методов.

Если мы положим  $R = 256$  и откажемся от рекурсивного вызова для корзины 0, то программа 10.2 становится эффективным методом сортировки строк в стиле С. Если нам также известно, что длина всех строк не превышает некоторой фиксированной длины, мы можем ввести специальную переменную **bytesword**, фиксирующую эту длину и присвоить ей соответствующее значение, либо отказаться от проверки по переменной **bytesword** и выполнять сортировку символьных строк переменной длины. Для сортировки строк мы обычно будем выполнять реализацию абстрактной операции **digit** в виде единственной ссылки на массив, согласно изложенному в разделе 10.1. Путем соответствующего подбора значений основания  $R$  и величины **bytesword** (с проверкой по этим значениям) легко можно модифицировать программу 10.2 таким образом, чтобы она могла работать со строками символов из нестандартных алфавитов или нестандартных форматов с соблюдением ограничений по длине и других соглашений.

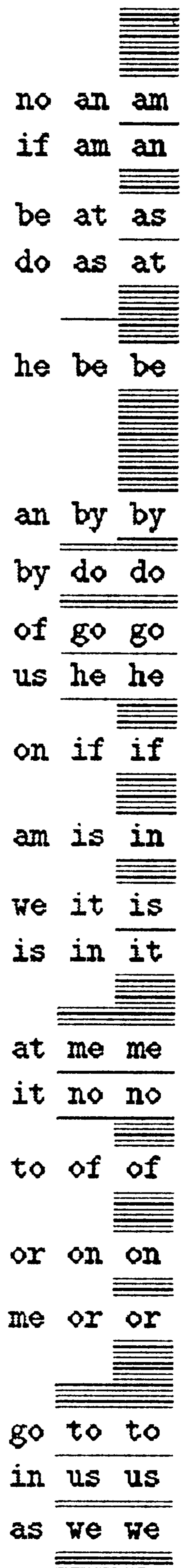


Сортировка строк лишней раз показывает, насколько важным является правильное обращение с пустыми корзинами. На рис. 10.8 показан процесс деления для примера, подобного представленному на рис. 10.7, но только для слов, состоящих из двух букв, и явно заданных пустых корзин. В рамках этого примера мы сортируем двухбуквенные слова методом поразрядной сортировки, используя в качестве основания системы счисления число 26, так что на каждой стадии сортировки имеется 26 корзин. На первой стадии число пустых корзин невелико, однако на второй стадии пустые корзины преобладают.

Функция поразрядной сортировки MSD выполняет разделение файла по первой цифре ключей, затем рекурсивно вызывает сама себя для обработки подфайлов, соответствующих каждому значению. На рис. 10.9 представлена структура этих рекурсивных вызовов для примера применения поразрядной сортировки MSD, показанного на рис. 10.8. Структура вызова соответствует *многопутевому бинарному дереву*, прямому обобщению древовидной структуры для двоичной быстрой сортировки, показанной на рис. 10.4. Каждый узел соответствует рекурсивному вызову сортировки MSD для некоторого подфайла. Например, поддерево корня с корнем, помеченном буквой **o**, соответствует сортировке подфайла, состоящего из трех ключей **of**, **on** и **or**.

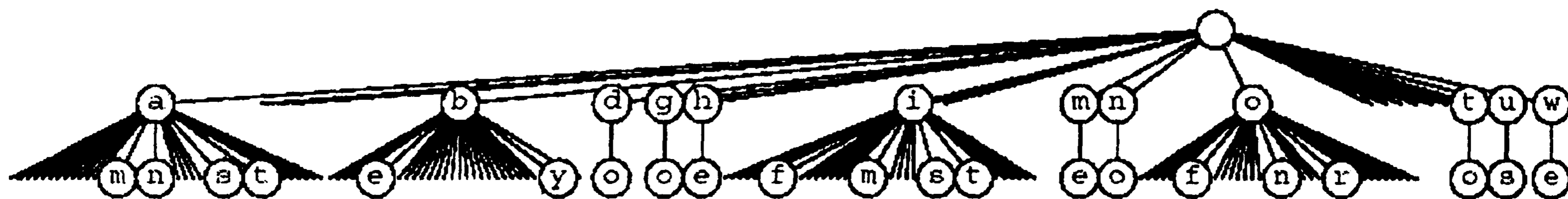
Из этих рисунков легко видеть, что в процессе сортировки строк методом MSD (сначала по старшей цифре) появляется значительное число пустых корзин. В разделе 10.4 мы рассмотрим способы решения этой проблемы; в главе 15 мы будем изучать явно заданные бинарные древовидные структуры, используемые в приложениях, ориентированных на обработку строк. В общем случае мы будем работать с компактными представлениями древовидных структур, которые не содержат узлов, соответствующих пустым корзинам и в которых метки сдвинуты с ребер на нижние узлы. Подобное имеет место на рис. 10.10, изображающем структуру, соответствующую рекурсивной структуре вызовов (при этом игнорируются пустые корзины), на примере сортировки множества трехбуквенных слов методом поразрядной сортировки MSD из рис. 10.7. Например, поддерево корня с вершиной, помеченной буквой **j**, соответствует сортировке корзины, содержащей четыре ключа: **jam**, **jau**, **jot** и **joy**. Подробно мы рассмотрим свойства таких деревьев в главе 15.

Основная трудность получения максимальной эффективности поразрядной сортировки MSD ключей, представленных в виде длинных строк, в практических условиях обуслов-



**РИСУНОК 10.8. ПРИМЕР ПОРАЗРЯДНОЙ СОРТИРОВКИ MSD (С ПУСТЫМИ КОРЗИНАМИ)**

*Уже на второй стадии сортировки файлов небольших размеров встречается избыточное число пустых корзин.*



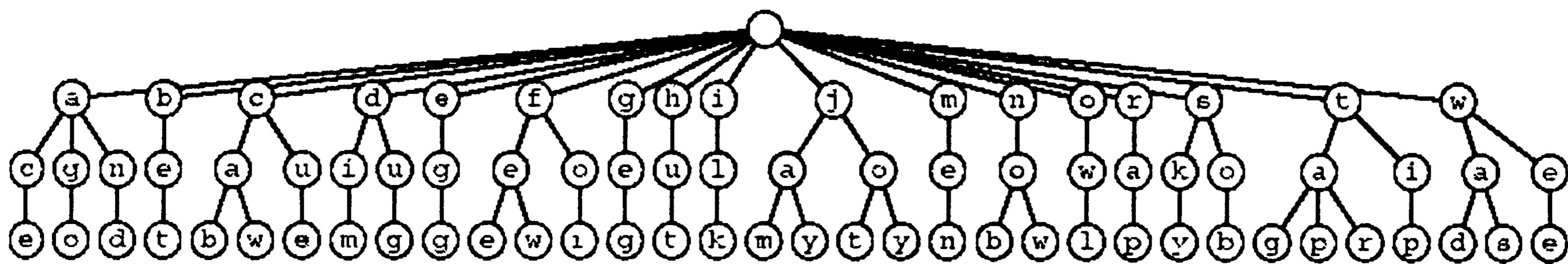
**РИСУНОК 10.9. РЕКУРСИВНАЯ СТРУКТУРА ПОРАЗРЯДНОЙ СОРТИРОВКИ MSD**

Это дерево соответствует действию рекурсивной поразрядной сортировки MSD, реализованной в программе 10.2, на примере упорядочения совокупности двухбуквенных слов методом поразрядной сортировки MSD, представленного на рис. 10.8. Если файл принимает размер 0 или 1, рекурсивные вызовы отсутствуют. В остальных случаях имеет место 26 вызовов: один на каждое возможное значение текущего байта.

лена низким уровнем фактора случайности сортируемых данных. В типичном случае такие строки содержат большие промежутки одних и тех же данных или ненужных данных, либо некоторые их части принимают значения из узкого диапазона. Например, приложение, выполняющее обработку записей, содержащих данные о студентах, могут иметь дело с ключами, поля которых соответствуют году окончания школы (четыре байта, отличающиеся друг от друга только в одном байте), название штата (максимум 10 байтов, принимающее одно из 50 возможных значений) и пол (1 байт, принимающий одно из двух возможных значений), а также поле фамилии студента (в большей степени, чем предыдущие, соответствует случайной строке, но в то же время соблюдаются некоторые закономерности: в общем случае оно не бывает коротким, буквы не подчиняются равномерному распределению, имеет место большое число замыкающих пробелов в поле фиксированной длины). Все эти ограничения приводят к образованию пустых корзин в процессе поразрядной сортировки MSD (см. упражнение 10.23).

Один из практических способов решения этой проблемы заключается в разработке более сложной реализации абстрактной операции доступа к конкретным байтам, которая учитывает любые специальные знания о сортируемых строках. Другим методом, который достаточно просто реализуется и который называется *эвристика в масштабах корзины* (bin-span-heuristics) заключается в том, что запоминаются начальные и конечные значения диапазонов значений непустых корзин на стадии подсчета, а затем используются только корзины, попадающие в полученный диапазон (возможно, с учетом некоторых специальных случаев, таких как специальные значения ключей, например, 0 или пробел). Такое усовершенствование весьма привлекательно для ситуаций типа описанных в предыдущем параграфе. Например, в случае алфавитно-цифровых данных, принимая в качестве основания системы счисления число 256, мы можем работать с числами в одном разделе ключей и в результате получить только 10 непустых корзин, соответствующих цифрам, в то же время мы можем работать с буквами верхнего регистра в другом разделе ключей и при этом иметь 26 непустых корзин, соответствующих этим буквам.

Имеются различные альтернативы, которые мы можем попытаться использовать для расширения метода эвристики в масштабах корзины (см. раздел справок). Например, можно использовать дополнительную структуру данных для хранения сведений о непустых корзинах, поддерживать только счетчики и общаться к ним в рекурсивном режиме. Однако реализации такого подхода (и даже применения эвристики в



**РИСУНОК 10.10. РЕКУРСИВНАЯ СТРУКТУРА ПОРАЗРЯДНОЙ СОРТИРОВКИ MSD (ПУСТЫЕ ФАЙЛЫ ИГНОРИРУЮТСЯ)**

*Предлагаемое представление рекурсивной структуры поразрядной сортировки MSD более компактно, чем представление на рис. 10.9. Каждый узел этого дерева помечен значением  $(i - 1)$ -й цифры конкретного ключа, где  $i$  — это расстояние от узла до корня. Каждый путь от корня до нижнего уровня дерева соответствует конкретному ключу; если теперь соединить метки узлов в одно слово, получим соответствующий ключ. Данное дерево соответствует представленному на 10.7 примеру поразрядной сортировки MSD слов из трех букв.*

масштабе корзины) более чем достаточно для подобного рода ситуаций, поскольку экономия средств в этом случае незначительна, если основание не есть очень большое число или если размер файла невелик, в этом случае следует применять меньшее основание либо сортировать файл с использованием другого метода. Мы можем достичь такой же экономии ресурсов, какую получаем за счет выбора соответствующего основания или переключения на использование специальных методов для обработки файлов небольших размеров, однако это не просто сделать. В разделе 10.4 мы рассмотрим еще одну версию быстрой сортировки, посредством которой изящно решается проблема пустых корзин.

## Упражнения

- ▷ 10.14. Начертить компактную древовидную структуру (пустые корзины отсутствуют, ключи узлов такие же, как и на рис. 10.10), соответствующую рис. 10.9.
- ▷ 10.15. Сколько узлов содержит полное дерево, соответствующее рис. 10.10?
- ▷ 10.16. Покажите, как выполняется разбиение набора ключей **now is the time for all good people to come the aid of their party** при поразрядной сортировке MSD.
- 10.17. Написать программу, которая выполняет четырех-путевое обменное разделение путем подсчета частоты повторения каждого ключа подобно тому, как это делается в условиях сортировки методом подсчета индексных ключей, с последующим использованием метода, подобного реализованному в программе 6.14 для перемещения ключей.
- 10.18. Написать программу, которая решает задачу  $R$ -путевого разделения в общем виде, используя метод, описанный в общих чертах в упражнении 10.17.
- 10.19. Написать программу, генерирующую произвольные 80-байтные ключи, после чего сортирует их методом поразрядной сортировки MSD для  $N = 10^3$ ,  $10^4$ ,  $10^5$  и  $10^6$ . Снабдите программу инструментальными средствами для распечатки общего количества байтов, проверенных в процессе каждой сортировки.
- 10.20. Каким может быть крайнее правое положение байта ключа, к которому по вашему предположению обратится программа из упражнения 10.19 во время доступа для каждого заданного значения  $N$ ? Если вы успешно справились с этим уп-

ражнением, снабдите программу инструментальными средствами, позволяющими отслеживать эти значения, и сравните результаты теоретических расчетов с результатами, полученными опытным путем.

**10.21.** Написать программу генератора ключей, которая порождает ключи путем перетасовки 80-байтной случайной последовательности. Воспользуйтесь полученным генератором для генерирования  $N$  произвольных ключей, затем выполните их упорядочение методом поразрядной сортировки MSD для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Сравните достигнутую производительность с аналогичным результатом для произвольного случая (см. упражнение 10.19).

**10.22.** Каким может быть крайнее правое положение байта ключа, к которому по вашему предположению обратится программа из упражнения 10.19 во время доступа к каждому из  $N$  заданных значений? Если вы выполнили это упражнение, сравните теоретические расчеты с эмпирическими результатами, полученными при выполнении вашей программы.

**10.23.** Написать программу генератора ключей, которая порождает случайные 30-байтные ключи, состоящие из трех полей: поле размером в четыре байта, содержащее одну из 10 заданных строк, поле размером 10 байтов, содержащее одну из 50 заданных строк, однобайтное поле с одним из двух возможных значений, и поле размером 15 байтов, содержащее выровненную слева строку случайных символов, которая с равной вероятностью может содержать от четырех до 15 символов. Используйте полученный генератор ключей для генерации  $N$  случайных ключей, затем выполните их упорядочение методом поразрядной сортировки MSD для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Снабдите программу инструментальными средствами для распечатки общего количества байтов, проверенных в процессе сортировки. Сравните достигнутую производительность с аналогичным результатом для произвольного случая (см. упражнение 10.19).

**10.24.** Внесите в программу такие изменения, чтобы стала возможной реализация эвристики в масштабах корзины. Проверьте, как работает программа на данных из упражнения 10.23.

## 10.4. Трехпутевая поразрядная быстрая сортировка

Еще одна возможность приспособить быструю сортировку для поразрядной сортировки MSD заключается в использовании трехпутевого разделения ключей по старшим байтам, переходя к следующему байту только в среднем подфайле (в котором содержатся ключи, старшие байты которых равны старшему байту разделяющего элемента). Реализация этого метода не представляет трудностей (по существу, достаточно описания из одного предложения плюс код из программы 7.5, обеспечивающий трехпутевое разделение), а сам метод легко адаптируется к различным ситуациям. Программа 10.3 содержит полную реализацию этого метода.

По существу, выполнение такой трехпутевой поразрядной быстрой сортировки равносильно сортировке файла по старшим разрядам ключей (с использованием метода быстрой сортировки) с последующим применением в режиме рекурсии этого же метода к оставшейся части ключей. При сортировке строк этот метод выглядит предпочтительнее в сравнении с обычной быстрой сортировкой и поразрядной сортировкой MSD. Разумеется, его можно рассматривать как гибрид двух указанных алгоритмов.

Сравнивая трехпутевую поразрядную быструю сортировку со стандартной поразрядной сортировкой MSD мы можем отметить, что она разбивает файл всего лишь на три части, так что она не использует всех преимуществ быстрого многопутевого разбиения, особенно на ранних стадиях сортировки. С другой стороны, на поздних стадиях поразрядной сортировки MSD появляется множество пустых корзин, но в то же время поразрядная быстрая сортировка хорошо приспособлена для случаев дублированных ключей, ключей, принимающих значения в узких диапазонах, файлов небольших размеров и многих других случаев, в условиях которых поразрядная сортировка MSD выполняется медленно. Особо важное значение приобретает то обстоятельство, что подобного рода разбиение хорошо подходит для случаев проявления различного рода закономерностей в разных частях ключа. Более того, для нее не нужны никакие вспомогательные файлы. В противовес всем этим достоинствам можно поставить тот факт, что требуются дополнительные операции обмена, чтобы реализовать многопутевое разбиение с помощью трехпутевого разбиения, когда число подфайлов велико.

На рис. 10.11 приводится пример применения этого метода для сортировки совокупности трехбуквенных слов, представленных на рис. 10.7. Рисунок 10.12 отражает структуру рекурсивных вызовов. Каждый узел соответствует в точности трем рекурсивным вызовам: по ключам с меньшим значением первого байта (левый потомок), по ключам с тем же значением первого байта (средний потомок) и по ключам с большим значением первого байта (правый потомок).

Когда сортируемые ключи соответствуют абстракции из раздела 10.2, стандартную быструю сортировку (а также все другие методы сортировки, рассмотренные в главах 6—9) можно рассматривать как поразрядную сортировку MSD, поскольку функция сравнения осуществляет доступ сначала к наиболее значащей части ключа (см. упражнение 10.3). Например, если в качестве ключей выступают строки, функция сравнения должна осуществлять доступ только к старшим байтам, если они попарно различаются, и к двум байтам, если первые байты совпадают, а вторые байты различны, и т.д. Таким образом, стандартный алгоритм автоматически затрачивает некоторую часть выигрыша в производительности, в погоне за которой мы используем поразрядную сортировку MSD (см. раздел 7.7). Существенное различие заключается в том, что стандартный алгоритм не может предпринять никаких действий, когда старшие байты равны.

```

now gig ace ago ago
for for bet bet ace
tip dug dug and and
ilk ilk cab ace bet
dim dim dim cab
tag ago ago cav
jot and and cue
sob fee egg egg
nob cue cue dug
sky cav cav dim
hut hut fee
ace ace for
bet bet few
men cab ilk
egg egg gig
few few hut
jay jay jam
owl jot jay
joy joy joy
rap jam jot
gig owl owl men
vee vee now owl
was was nob nob
cab men men now
wad wad rap
cav sky sky sky sky
cue nob was tip sob
fee sob sob sob tip tar
tap tap tap tap tap tap
ago tag tag tag tag tag
tar tar tar tar tar tip
dug tip tip was
and now vee vee
jam rap wad wad

```

**РИСУНОК 10.11.  
ТРЕХПУТЕВАЯ  
ПОРАЗРЯДНАЯ  
БЫСТРАЯ  
СОРТИРОВКА**

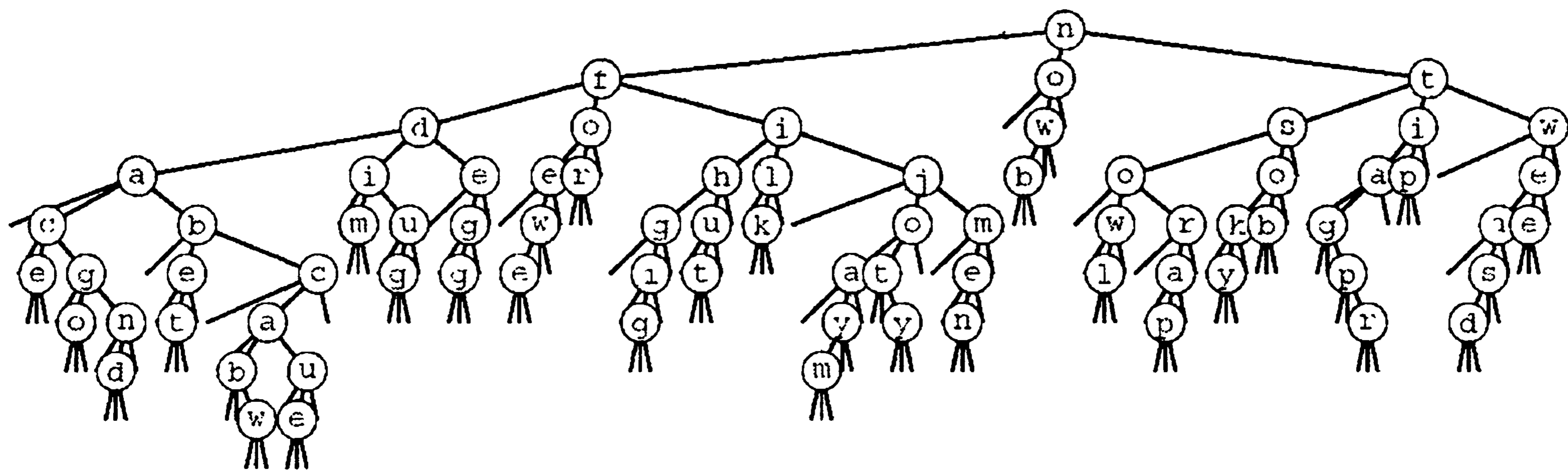
*Мы делим файл на три части: слова, начинающиеся с букв от a до i, слова, начинающиеся с буквы j, и слова, начинающиеся с букв от k до z. Затем мы выполняем сортировку в рекурсивном режиме.*

Действительно, программу 10.3 можно рассматривать как способ метода быстрой сортировки запоминать все, что ему стало известно о старших цифрах элементов после их использования в процедуре разделения файла на несколько частей. В малых файлах, для которых большая часть операций сравнения была выполнена в процессе сортировки, существует большая вероятность того, что многие старшие байты совпадают. Стандартный алгоритм обязан просмотреть все эти байты в рамках каждой операции сравнения, в то время как трехпутевой алгоритм не делает этого.

### Программа 10.3. Трехпутевая поразрядная быстрая сортировка

Программный код поразрядной сортировки MSD фактически мало чем отличается от программного кода быстрой сортировки с трехпутевым разбиением (программа 9.5), отличия состоят в следующем: (i) ссылки на ключи становятся ссылками на конкретные байты ключей, (ii) текущий байт добавляется как параметр к рекурсивной служебной программе и (iii) рекурсивные вызовы для среднего подфайла перемещаются к следующему байту. Мы избегаем выполнять перемещения за пределы концов строк путем проверки, равно ли разделяющее значение 0, перед рекурсивными обращениями, которые обеспечивают переход к следующим байтам. Когда разделяющим значением является 0, левый подфайл пуст, средний подфайл соответствует ключам, которые, по определению программы, равны этому значению, а правый подфайл соответствует более длинным строкам, которые требуют дальнейшей обработки.

```
#define ch(A) digit(A, d)
template <class Item>
void quicksortX(Item a[], int l, int r, int d)
{
    int i, j, k, p, q; int v;
    if (r-l <= M) { insertion(a, l, r); return; }
    v = ch(a[r]); i = l-1; j = r; p = l-1; q = r;
    while (i < j)
    {
        while (ch(a[++i]) < v) ;
        while (v < ch(a[--j])) if (j == l) break;
        if (i > j) break;
        exch(a[i], a[j]);
        if (ch(a[i])==v) { p++; exch(a[p], a[i]); }
        if (v==ch(a[j])) { q--; exch(a[j], a[q]); }
    }
    if (p == q)
    { if (v != "\0") quicksortX(a, l, r, d+1);
      return; }
    if (ch(a[i]) < v) i++;
    for (k = l; k <= p; k++, j--) exch(a[k], a[j]);
    for (k = r; k >= q; k--, i++) exch(a[k], a[i]);
    quicksortX(a, l, j, d);
    if ((i == r) && (ch(a[i]) == v)) i++;
    if (v != "\0") quicksortX(a, j+1, i-1, d+1);
    quicksortX(a, i, r, d);
}
```

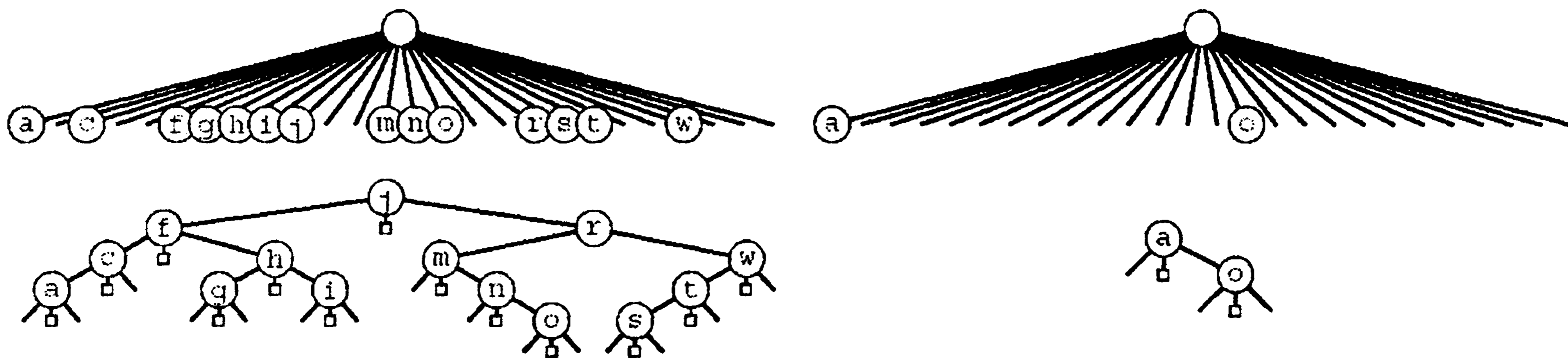


**РИСУНОК 10.12. РЕКУРСИВНАЯ СТРУКТУРА ТРЕХПУТЕВОЙ ПОРАЗРЯДНОЙ БЫСТРОЙ СОРТИРОВКИ**

Представленная на диаграмме комбинация двоичного и троичного деревьев соответствует подстановке 26-путевых узлов в двоичное дерево, изображенное на рис. 10.10, троичных деревьев поиска, как показано на рис. 10.13. Любой путь от корня на нижний уровень дерева, который оканчивается средней связью, определяет ключ файла, задаваемый символами, охваченными средними связями на этом пути. В диаграмме, показанной на рис. 10.10, имеется 1035 пустых связей, которые не отображены на ней, что касается рассматриваемой диаграммы, то все 155 пустых связей, которыми обладает это дерево, на диаграмме показаны. Каждая пустая связь соответствует пустой корзине, так что это различие показывает, как трехпутевое разбиение может существенно сократить число пустых корзин, которые появляются во время поразадной сортировки MSD.

Рассмотрим случай, когда ключи имеют большую длину (для простоты предположим, что длина ключей фиксирована) и в то же время по большей части старшие байты одинаковы. В подобного рода ситуациях время выполнения обычной быстрой сортировки пропорционально длине слова, помноженной на  $2N \ln N$ , тогда как время выполнения ее поразадной версии пропорционально  $N$ , умноженному на длину слова (чтобы обнаружить все равные между собой старшие байты) плюс  $2N \ln N$  (затрачивается на сортировку более коротких ключей). Иначе говоря, этот метод работает в  $\ln N$  раз быстрее, чем обычная быстрая сортировка, если принимать во внимание только стоимость операций сравнения. Для ключей, используемых в сортировочных приложениях на практике, характеристики, подобные полученным в условиях рассмотренного выше искусственного примера, не являются чем-то необычным (см. упражнение 10.25).

Другим интересным свойством трехпутевой поразадной быстрой сортировки является то, что ее характеристики напрямую не зависят от значения основания системы счисления. Для других методов сортировки, использующих основание системы счисления, нужно заводить и поддерживать вспомогательный массив, индексированный по значению основания системы счисления. Также следует позаботиться о том, чтобы размер этого массива ненамного превосходил размер самого файла. Для этого метода нет такой таблицы. Если в качестве основания брать исключительно большое значение (больше длины слова), то рассматриваемый метод превращается в обычную быструю сортировку, а если в качестве основания взять число 2, то он вырождается в обычную двоичную быструю сортировку, и только промежуточные значения основания системы счисления позволяют эффективно преодолевать равные промежутки между фрагментами ключей.



**РИСУНОК 10.13. ПРИМЕР УЗЛОВ ДЕРЕВА ТРЕХПУТЕВОЙ ПОРАЗРЯДНОЙ БЫСТРОЙ СОРТИРОВКИ.**

Трехпутевая поразрядная быстрая сортировка подходит к решению проблемы пустых корзин, характерной для поразрядной сортировки MSD, с применением трехпутевого разбиения, чтобы устранить значение одного байта и (в рекурсивном режиме) работать с другими. Это действие соответствует замене каждого узла на пути вдоль средних связей по дереву, которое описывает рекурсивную структуру вызовов поразрядной сортировки MSD (см. рис. 10.9) на троичное дерево, в котором каждой непустой корзине соответствует внутренний узел. Для полных узлов (слева) такое изменение требует затрат времени и не влечет за собой заметной экономии пространства памяти, в то время как в случае пустых узлов (справа), затраты времени минимальны, а экономия памяти значительная.

Мы можем разработать гибридный метод, пригодный для многих практических приложений, обладающий превосходными характеристиками, применяя стандартную поразрядную сортировку MSD для упорядочения крупных файлов, чтобы воспользоваться преимуществами многопутевого разбиения, и трехпутевую поразрядную сортировку с небольшим значением основания системы счисления для небольших файлов, чтобы избежать отрицательных эффектов, обусловленных наличием большого числа пустых корзин.

Трехпутевая быстрая сортировка успешно применяется и в тех случаях, когда сортируемыми ключами являются векторы (как в математическом смысле, так и в смысле стандартной библиотеки шаблонов C++). Другими словами, если ключи составлены из независимых компонентов (каждый компонент сам по себе независимый ключ), мы имеем возможность переупорядочить записи таким образом, что они будут располагаться в порядке следования первых компонентов ключей и в порядке следования вторых компонентов ключей в тех случаях, когда равны их первые компоненты и т.д. Мы можем рассматривать сортировку векторов как обобщенный вид поразрядной сортировки, где  $R$  может быть произвольно большим. После выполнения настройки программы 10.3 на это приложение, мы будем называть ее *многомерной быстрой сортировкой (multikey quicksort)*.

## Упражнения

**10.25.** Для  $d > 4$  предположим, что ключи состоят из  $d$  байтов, при этом заключительные 4 байта принимают случайные значения, а все остальные равны 0. Дайте оценку количеству просмотренных байтов при сортировке посредством методов трехпутевой поразрядной быстрой сортировки (Программа 10.3) и стандартной быстрой сортировки (Программа 7.1) файла размером  $N$  для больших  $N$ , и вычислите для обоих случаев отношение значений времени выполнения сортировки.



**10.26.** Эмпирически определите размер байта, для которого время выполнения трехпутевой сортировки 64-разрядных ключей минимально, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

● **10.27.** Разработать реализацию трехпутевой поразрядной сортировки для связанных списков.

**10.28.** Разработать реализацию многопутевой поразрядной сортировки для случая, когда ключи представлены в виде векторов из  $t$  чисел с плавающей точкой, используя проверку чисел с плавающей точкой на равенство, описанную в упражнении 4.6.

**10.29.** Используя генератор ключей, описанный в упражнении 10.19, выполнить трехпутевую поразрядную быструю сортировку для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Сравнить ее производительность с аналогичным показателем для поразрядной сортировки MSD.

**10.30.** Используя генератор ключей, описанный в упражнении 10.21, выполнить трехпутевую поразрядную быструю сортировку для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Сравнить ее производительность с аналогичным показателем для поразрядной сортировки MSD.

**10.31.** Используя генератор ключей, описанный в упражнении 10.23, выполнить трехпутевую поразрядную быструю сортировку для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Сравнить ее производительность с аналогичным показателем для поразрядной сортировки MSD.

## 10.5. Поразрядная сортировка LSD

Метод, альтернативный поразрядной сортировке, предусматривает просмотр байтов в направлении справа налево. На рис. 10.14 показано, как задача сортировки трехбуквенных слов может быть решена за три прохода по файлу. Мы сначала сортируем файл по последней букве (используя для этой цели метод подсчета индексных ключей), затем по средней букве, и только потом — по первой букве.

На первых порах не так то просто поверить, что этот метод работает; и в самом деле, он вообще не работает, если используемый метод сортировки неустойчив (см. определение 6.1). После того, как установлена важность свойства устойчивости, нетрудно дать формулировку доказательства того, что поразрядная сортировка LSD работает: мы знаем, что после упорядочения ключей по  $i$  замыкающим байтам (при сохранении устойчивости) любые два ключа в файле появляются в нужном порядке (в соответствии с просмотренными на текущий момент разрядами) либо благодаря тому, что первые из  $i$  замыкающих байтов отличны друг от друга, в подобном случае сортировка по этому байту расставила их в соответствующем порядке, или если первые из  $i$  замыкающих байтов совпадают, они уже упорядочены нужным образом в силу свойства устойчивости. Можно дать этому другую формулировку: если  $w - i$  еще не просмотренных байтов какой-

now	sob	cab	ace
for	nob	wad	ago
tip	cab	tag	and
ilk	wad	jam	bet
dim	and	rap	cab
tag	ace	tap	caw
jot	wee	tar	cue
sob	cue	was	dim
nob	fee	caw	dug
sky	tag	raw	egg
hut	egg	jay	fee
ace	gig	ace	few
bet	dug	wee	for
men	ilk	fee	gig
egg	owl	men	hut
few	dim	bet	ilk
jay	jam	few	jam
owl	men	egg	jay
joy	ago	ago	jot
rap	tip	gig	joy
gig	rap	dim	men
wee	tap	tip	nob
was	for	sky	now
cab	tar	ilk	owl
wad	was	and	rap
tap	jot	sob	raw
caw	hut	nob	sky
cue	bet	for	sob
fee	you	jot	tag
raw	now	you	tap
ago	few	now	tar
tar	caw	joy	tip
jam	raw	cue	wad
dug	sky	dug	was
you	jay	hut	wee
and	joy	owl	you

**РИСУНОК 10.14.**

**ПРИМЕР**

### ПОРАЗРЯДНОЙ СОРТИРОВКИ LSD

*Трехбуквенные слова упорядочиваются за три прохода (слева направо) методом поразрядной сортировки LSD*

либо пары ключей идентичны, то любое различие между этими ключами определяется  $i$  байтами, которые уже просмотрены, и если эти ключи должным образом упорядочены, то они сохраняют этот порядок в силу свойства устойчивости. С другой стороны, если  $w - i$  еще не просмотренных байтов различны, то те  $i$  байтов, которые уже просмотрены, не играют никакой роли, так что следующий проход должным образом упорядочит эту пару, учитывая различия в более значащих байтах.

Требование устойчивости означает, например, что метод разделения, использованный в двоичной быстрой сортировке, не может быть использован в двоичной версии рассматриваемого метода сортировки справа налево. С другой стороны, метод сортировки с подсчетом индексных ключей является устойчивым методом сортировки, что сразу же приводит нас к классическому и эффективному алгоритму. Программа 10.4 представляет собой реализацию этого метода. По-видимому, понадобится вспомогательный массив для целей распределения — методы, которые используются в упражнениях 10.17 и 10.18, выполняющие распределение без использования дополнительной памяти, жертвуют устойчивостью в пользу отказа от дополнительного массива.

Метод поразрядной сортировки использовался в старых машинах для сортировки перфокарт для вычислительных машин. Такие машины обладали способностью распределения колоды карт по 10 корзинам в соответствии видом отверстий, пробитых в специальных столбцах. Если в некотором наборе столбцов пробиты определенные числа, оператор может сортировать перфокарты, пропуская их через машину по крайней правой цифре, затем собрав и упорядочив колоды перфокарт, полученные на выходе, снова пропустить их через машину, на сей раз по цифре, следующей за крайней правой, и т.д. Физическая сортировка перфокарт представляет собой устойчивый процесс, который можно смоделировать сортировкой методом подсчета индексных ключей. Эта версия поразрядной сортировки LSD не только широко использовалась в коммерческих приложениях в пятидесятых и шестидесятых годах прошлого столетия, этим методом пользовались многие осторожные программисты, которые проби-вали некоторые последовательности чисел в нескольких концевых колонках колоды перфокарт, на которых набита программа, чтобы можно было восстановить порядок следования перфокарт в колоде механическим способом, если колода случайно рассыплется.

#### Программа 10.4. Метод поразрядной сортировки LSD

Данная программа реализует сортировку байтов в словах методом подсчета индексных ключей, продвигаясь слева направо. Реализация сортировки методом подсчета индексных ключей должна быть устойчивой. Если  $R$  равна 2 (благодаря чему **bytesword** и **bitwords** идентичны), данная программа является *прямой поразрядной сортировкой* — с проходом справа налево, с поразрядным просмотром (см. рис. 10.15).

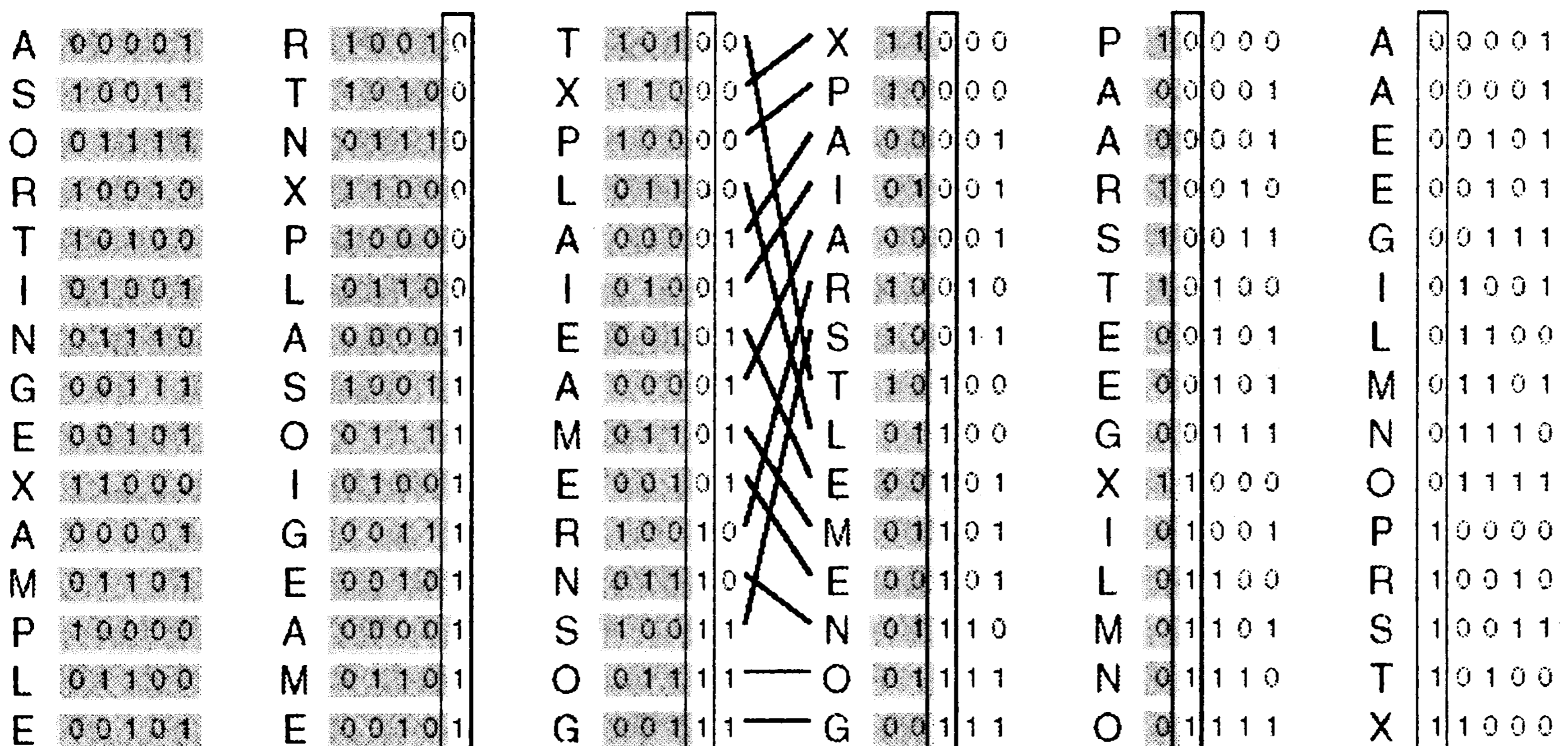
```
template <class Item>
void radixLSD(Item a[], int l, int r)
{ static Item aux[maxN];
  for (int d = bytesword-1; d >= 0; d--)
  {
    int i, j, count[R+1];
    for (j = 0; j < R; j++) count[j] = 0;
```

```

for (i = 1; i <= r; i++)
    count[digit(a[i], d) + 1]++;
for (j = 1; j < R; j++)
    count[j] += count[j-1];
for (i = 1; i <= r; i++)
    aux[count[digit(a[i], d)]++] = a[i];
for (i = 1; i <= r; i++) a[i] = aux[i];
}
}

```

Рисунок 15 иллюстрирует работу двоичной поразрядной сортировки LSD на условном примере упорядочения ключей, благодаря чему становится возможным сравнение с рис.10.3. Для рассматриваемых 5-разрядных ключей полное упорядочение достигается за пять проходов по ключам справа налево. Сортировка записей с одноразрядным ключом сводится к разбиению файла таким образом, что все записи с ключом 0 следуют раньше всех записей с ключом 1. Как только что было показано, мы не можем пользоваться стратегией разбиения, которую мы в рассматривали в начале данной главы при обсуждении программы 10.1, несмотря на то, что она по всем признакам решает ту же проблему, в силу того, что она не устойчива. Имеет смысл рассмотреть поразрядную сортировку с основанием системы счисления, равным 2, поскольку во многих случаях ее удобно реализовать на высокопроизводительных машинах и в аппаратном обеспечении специального назначения (см. упражнение 10.38). В программах мы используем максимально возможное число разрядов с тем, чтобы уменьшить число проходов, которое ограничено только размерами массива рассматриваемых чисел (см. рис. 10.16).



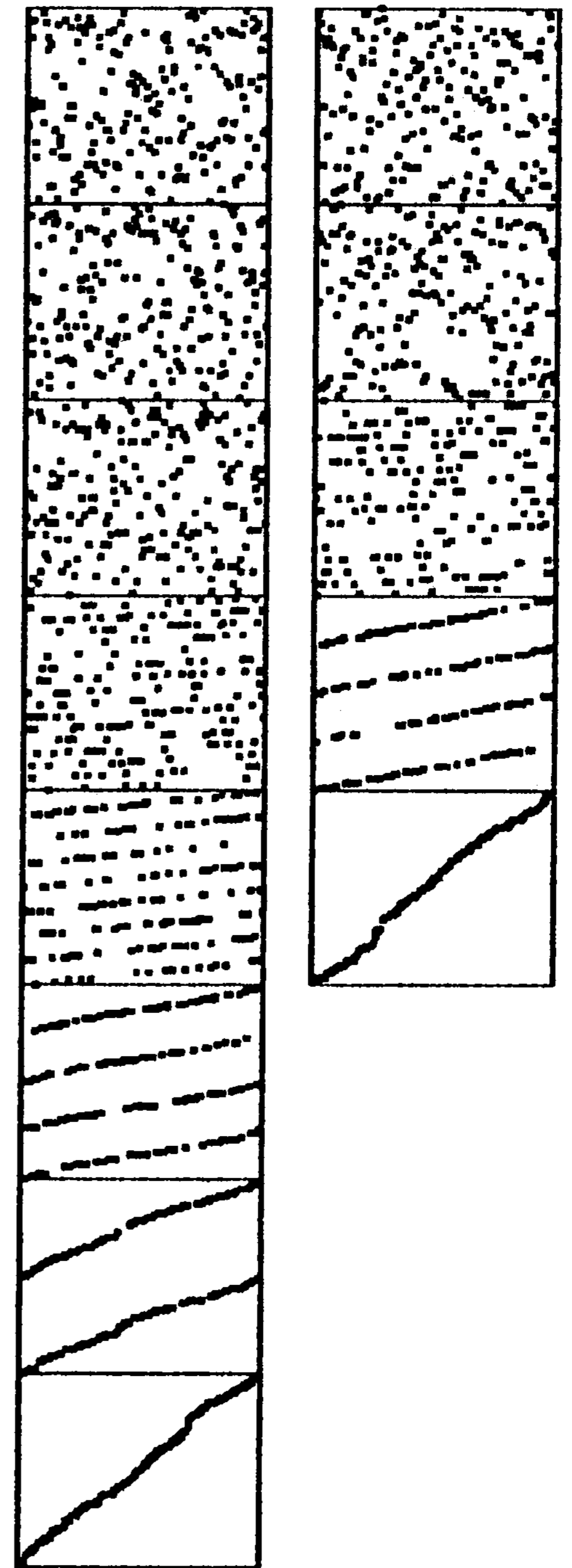
### РИСУНОК 10.15. ПОРАЗРЯДНАЯ (ДВОИЧНАЯ) СОРТИРОВКА LSD (ПОКАЗАНЫ РАЗРЯДЫ КЛЮЧЕЙ)

Данная диаграмма служит иллюстрацией использования поразрядной сортировки справа налево, разряд за разрядом, применительно к рассматривавшемуся выше файлу учебному ключей. Мы вычисляем  $i$ -й столбец из  $(i - 1)$ -го столбца файла, извлекая (не нарушая при этом свойства устойчивости) все ключи с 0 в  $i$ -м разряде, а затем все ключи с 1 в  $i$ -м разряде. Если перед операцией извлечения  $(i - 1)$ -й столбец был упорядочен по  $i - 1$  замыкающим разрядам ключей, то и  $i$ -й столбец будет упорядочен по  $i$  замыкающим разрядам ключей по окончании этой операции. Перемещение ключей на третьей стадии пояснений не требует.

Применение подхода LSD к сортировке строковых данных обычно сопряжено с некоторыми трудностями, что в первую очередь объясняется переменной длиной ключей. В случае сортировок MSD достаточно просто отличать один ключ от другого по их старшим байтам, однако в основе сортировок лежит принцип постоянной длины ключа, при этом ведущие ключи используются только на заключительных проходах. По всей видимости, даже в случае ключей (большой) фиксированной длины, поразрядная сортировка LSD выполняет бесполезную работу на правой стороне ключей, ибо, как мы смогли убедиться выше, в процессе сортировки обычно используется только левая часть каждого ключа. В разделе 10.6 мы найдем способ решения этой проблемы после того, как подробно изучим свойства поразрядной сортировки.

## Упражнения

- 10.32.** Используя генератор ключей из упражнения 10.19 выполнить поразрядную сортировку LSD для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Выполнить сравнение показателей этой сортировки с параметрами поразрядной сортировкой MSD.
- 10.33.** Используя генератор ключей из упражнения 10.21 и 10.23 выполнить поразрядную сортировку LSD для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Выполнить сравнение показателей этой сортировки с параметрами поразрядной сортировкой MSD.
- 10.34.** Представить (неотсортированный) результат попытки применения поразрядной сортировки, в основу которой положен метод разбиения, используемый в методе двоичной быстрой сортировки, для примера, представленного на рис.10.15.
- ▷ **10.35.** Представить результаты использования поразрядной сортировки LSD для упорядочения по двум первым символам совокупности ключей **now is the time for all good people to come the aid of their party**.
- **10.36.** Разработать реализацию поразрядной сортировки LSD, используя связные списки.
  - **10.37.** Найти эффективный метод, который (i) переупорядочивает записи файла таким образом, что те их них, ключи которых начинаются с бита 0, идут раньше тех записей, ключи которых начинаются с бита 1, (ii) использует дополнительное пространство памяти, пропорциональное квадратному корню из числа записей (или меньше того) и (iii) является устойчивым.



**РИСУНОК 10.16.**  
ДИНАМИЧЕСКИЕ  
ХАРАКТЕРИСТИКИ  
ПОРАЗРЯДНОЙ  
СОРТИРОВКИ LSD

На диаграмме показаны этапы поразрядной сортировки LSD 8-разрядных ключей, принимающих случайные значения, с основанием 2 (слева) и 4, последняя включает в себе все другие этапы, представленные на диаграмме для основания 2 (справа). Например, когда остаются два разряда (второй этап с конца в левой диаграмме, предпоследний в правой диаграмме) рассматриваемый файл состоит из четырех взаимно проникающих отсортированных подфайлов, состоящих из ключей, начинающихся с 00, 01, 10 и 11.

- **10.38.** Разработать программу, которая сортирует массив 32-разрядных слов, используя всего лишь одну следующую абстрактную операцию: задано положение разряда  $i$  и указатель на массив  $a[k]$ ; упорядочить  $a[k], a[k+1], \dots, a[k+63]$  посредством устойчивого метода таким образом, что слова с битом 0 в положении  $i$  располагаются раньше слов с битом 1 в позиции  $i$ .

## 10.6. Рабочие характеристики поразрядных сортировок

Время выполнения поразрядной сортировки LSD при сортировке записей  $N$  с ключами, состоящими из  $w$  байтов, пропорционально  $Nw$ , поскольку соответствующий алгоритм производит  $w$  проходов по всем  $N$  ключам. Как показывает рис.10.17, это свойство не зависит от природы входных данных.

Для случая длинных ключей и коротких байтов это время сопоставимо с  $M \lg N$ : например, если мы используем двоичную поразрядную сортировку LSD для сортировки 1 миллиарда 32 разрядных ключей, то как  $w$ , так и  $\lg N$  примерно равны 32. Для более коротких ключей и более длинных байтов время выполнения сортировки сопоставимо с  $N$ : например, если по отношению к 64-разрядным ключам используется 16-разрядное основание системы счисления, то  $w$  принимает значение 4, что можно считать константой малого значения.

Чтобы должным образом выполнить сравнение рабочих характеристик поразрядной сортировки с характеристиками алгоритмов, построенных на основе операции сравнения, мы более подробно должны проанализировать каждый байт ключей, а не ограничиваться только учетом числа ключей.

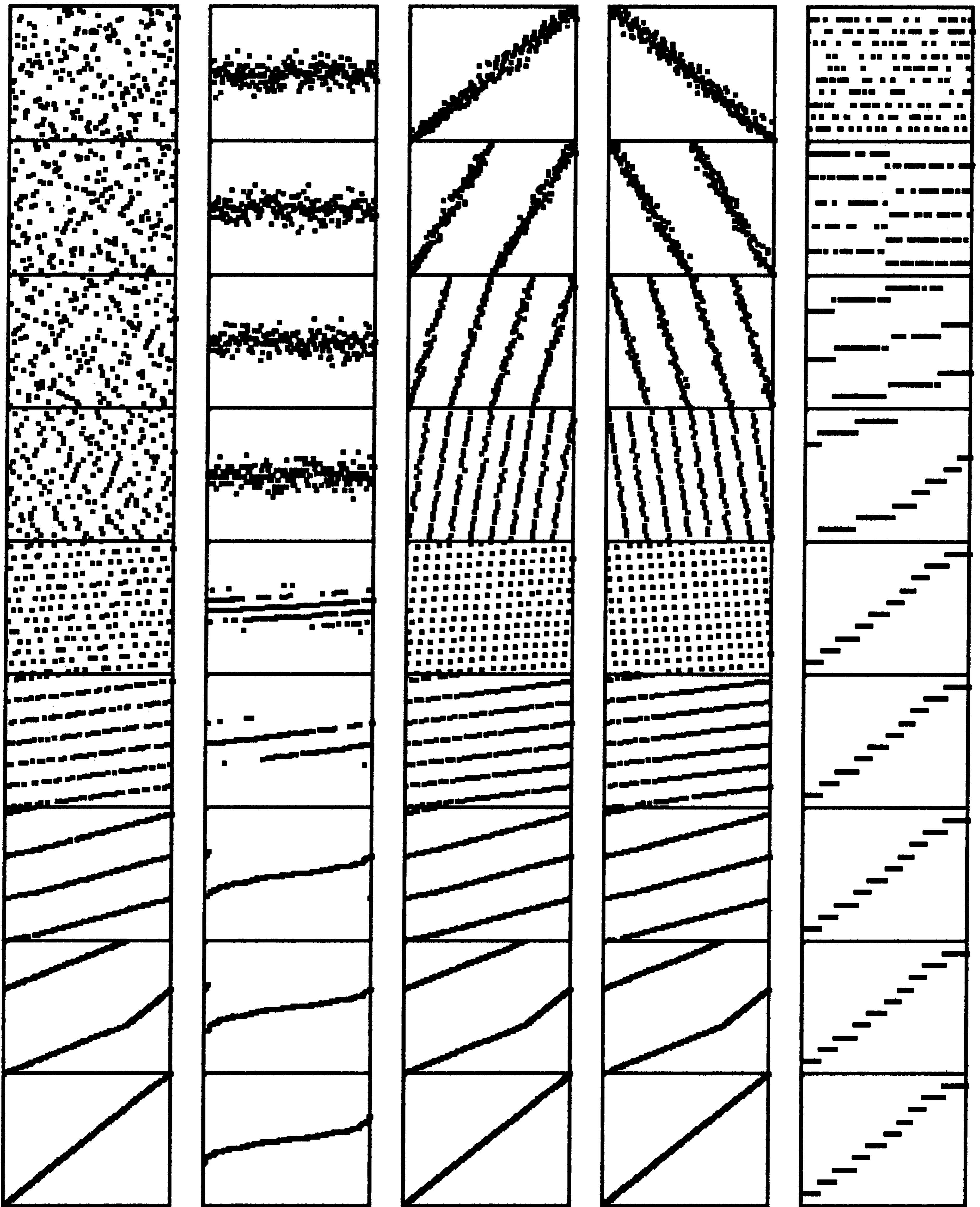
**Лемма 10.1.** *В худшем случае поразрядная сортировка выполняет проверку всех байтов и всех ключей.*

Другими словами, различные виды поразрядной сортировки суть *линейные* сортировки в том смысле, что затрачиваемое на нее время в большинстве случаев пропорционально количеству цифр во входных данных. Этот факт непосредственно следует из анализа программ: ни одна из цифр не проверяется дважды. Из всех программ, которые мы проверили, худший случай имел место, когда все ключи обладали одними и теми же значениями.

Как мы могли убедиться выше, в случае ключей, принимающих случайные значения, равно как и в во многих других случаях, время выполнения поразрядной сортировки MSD может быть сублинейным по отношению к общему числу битов данных, поскольку не во всех ситуациях можно выполнять полный просмотр конкретного ключа. Для ключей произвольной длины справедливо следующее утверждение:

**Лемма 10.2.** *Двоичная быстрая сортировка в среднем производит проверку  $N \lg N$  разрядов при сортировке ключей, состоящих их битов, принимающих случайное значение.*

Если размер файла выражается степенью 2, а составляющие его биты принимают случайные значения, то естественно предположить, что одна половина старших битов принимает значение 0, а другая половина — значение 1, так что рекуррентное соотношение  $C_N = 2C_{N/2} + N$  описывает возникшую ситуацию, как мы установили при обсуждении свойств быстрой сортировки в главе 7.



### РИСУНОК 10.17. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ ПОРАЗРЯДНОЙ СОРТИРОВКИ LSD ДЛЯ РАЗЛИЧНЫХ ВИДАХ ФАЙЛОВ

Представленные диаграммы служат иллюстрацией этапов поразрядной сортировки для файлов с произвольной организацией, файлов с распределением Гаусса, почти отсортированных файлов, почти отсортированных в обратном порядке файлов, файлов с произвольной организацией, обладающих 10 различными ключами (слева направо). Длина каждого файла равна 200. Время выполнения не зависит от исходного порядка входных данных. Три файла, обладающих одним и тем же набором ключей (первый, третий и четвертый файлы представляют собой перестановки целых чисел от 1 до 200) имеют одни и те же рабочие характеристики на завершающих этапах сортировки.

Опять-таки, это описание ситуации не достаточно точно, поскольку точка разделения приходится на центр только в среднем (поскольку число бит в ключе конечно). Тем не менее, для двоичной быстрой сортировки вероятность того, что разделяющая точка находится в окрестности центра, выше, чем для стандартного метода быстрой сортировки, так что старший член выражения, определяющего время выполнения сортировки, тот же, что и для случая, когда разделение идеально. Подробный анализ, доказывающий этот результат, представляет собой классический пример анализа алгоритмов, впервые выполненный Кнудом в 1973 г. (см. *раздел ссылок*).

Этот результат обобщается путем его применения к поразрядной сортировке MSD. Тем не менее, поскольку основной интерес для нас представляет время выполнения сортировки, а не символы просматриваемых ключей, мы должны проявлять осторожность, поскольку время выполнения поразрядной сортировки пропорционально величине основания системы счисления  $R$  и не имеет ничего общего с ключами.

**Лемма 10.3.** *Поразрядная сортировка MSD с основанием системы счисления  $R$  применительно к файлу размера  $N$  требует для выполнения по меньшей мере  $2N + 2R$  шагов.*

Поразрядная сортировка MSD требует выполнения по меньшей мере одного прохода сортировки методом подсчета индексных ключей, а подсчет индексных ключей предусматривает по меньшей мере два прохода по записям (один для подсчета, другой для распределения), на что затрачивается по меньшей мере  $2N$  шагов, еще два подхода предназначены для просмотра счетчиков (один для их инициализации в 0 в начале, другой — для проверки, находятся ли подфайлы в конце), на что уходит по меньшей мере еще  $2R$  шагов.

Доказательство наличия этого свойства практически тривиально, в то же время оно играет весьма важную роль в нашем понимании поразрядной сортировки MSD. В частности, оно говорит нам, что мы не можем утверждать, что время выполнения сортировки будет меньше в силу того, что  $N$  мало, поскольку  $R$  может быть намного больше, чем  $N$ . Короче говоря, *для сортировки файлов небольших размеров следует использовать другие методы.* Этот вывод может служить решением проблемы пустых корзин, которую мы обсуждали в конце раздела 10.3. Например, если  $R$  равно 256, а  $N$  равно 2, то поразрядная сортировка MSD будет в 128 раз медленнее, чем более простой метод, предусматривающий только сравнение элементов. Рекурсивная структура поразрядной сортировки MSD приводит к тому, что соответствующая рекурсивная программа будет многократно вызывать себя для большого числа файлов небольших размеров. Поэтому в условиях рассматриваемого примера игнорирование проблемы пустых корзин может привести к тому, что вся поразрядная сортировка замедлится в 128 раз по сравнению с тем, какой она может быть. Что касается промежуточных ситуаций (например, предположим, что  $R$  равно 256, а  $N$  равно 64), то стоимость не будет настолько катастрофичной, тем не менее весьма существенной. Использование сортировки вставками — не слишком удачный выбор, ибо стоимость  $N^2/4$  сравнений все еще недопустимо высока; не следует игнорировать проблему пустых корзин в силу того факта, что их очень много. Простейший путь решения этой проблемы состоит в использовании основания системы счисления, которая меньше размера сортируемого файла.

**Лемма 10.4.** *Если основание системы счисления всегда меньше размера файла, то в худшем случае число шагов, выполняемых поразрядной сортировкой MSD, учитывается постоянным множителем небольшой величины при  $\log_R N$  (для ключей, содержащих случайные байты) и постоянным множителем небольшой величины при числе байтов в ключе.*

Полученный для худшего случая результат вытекает непосредственно из предыдущих рассуждений, и анализ, выполненный для леммы 2, служит обобщением, позволяющим получить оценку для среднего случая. Для большого  $R$  множитель  $\log_R N$  принимает малое значение, так что для практических применений можно считать, что общее время пропорционально  $N$ . Например, если  $R = 216$ , то  $\log_R N$  меньше 3 для всех  $N < 2^{48}$ , при этом можно с уверенностью утверждать, что это число охватывает все возможные на практике размеры файлов.

Как и в случае леммы 2, на основании леммы 4 мы можем сделать важный для практических приложений вывод о том, что поразрядная сортировка MSD фактически является сублинейной функцией от общего числа разрядов в случайных ключах достаточно большой длины. Например, при сортировке 1 миллиона 64-разрядных случайных ключей потребуется проверка от 20 до 30 старших разрядов ключей, что составляет менее половины всех данных.

**Лемма 10.5.** *Трехпутевая поразрядная быстрая сортировка выполняет в среднем  $2N \log N$  операций сравнения байтов при сортировке  $N$  ключей (произвольной длины).*

Возможны два поучительных толкования этого результата. Во-первых, если считать рассматриваемый метод эквивалентным разбиению по старшему разряду, применяемому в рамках быстрой сортировки, то (рекурсивно) используя этот метод применительно к подфайлам, нас не должен удивлять тот факт, что общее число операций примерно такое же, как и в случае нормальной быстрой сортировки, но все это — операции сравнения единичных байтов, а не сравнения ключей в полном объеме. Во вторых, рассматривая этот метод под углом зрения рис. 10.13, мы вправе ожидать, что по лемме 3 время выполнения  $\log_R N$  должно быть помножено на  $2 \ln R$ , поскольку требуется  $2R \ln R$  шагов быстрой сортировки, чтобы отсортировать  $R$  байтов, в отличие от шагов для тех же байтов в случае бинарного дерева. Полное доказательство мы опускаем (см. раздел ссылок).

**Лемма 10.6.** *Поразрядная сортировка LSD может сортировать  $N$  записей с  $w$ -разрядными ключами за  $w / \lg R$  проходов, при этом используется дополнительное пространство памяти для  $R$  счетчиков (и буфер для переупорядочения файла).*

Доказательство этого факта непосредственно вытекает из реализации. В частности, если мы примем  $R = 2^{w/4}$ , мы получим четырехпроходную линейную сортировку.

## Упражнения

**10.39.** Предположим, что входной файл состоит из 1000 копий каждого числа от 1 до 1000, каждое представлено в виде 32-разрядного слова. Покажите, как вы воспользуетесь этими сведениями, чтобы получить быстрый вариант поразрядной сортировки.



- 10.40.** Предположим, что входной файл состоит из 1000 копий каждого из тысячи различных 32-разрядных чисел. Покажите как вы воспользуетесь этими сведениями, чтобы получить быстрый вариант поразрядной сортировки.
- 10.41.** Каким является общее число байтов, проверяемых в процессе трехпутевой поразрядной быстрой сортировки при сортировке строк байтов фиксированной длины в худшем случае?
- 10.42.** Эмпирически сравнить число байтов, проверяемых в процессе трехпутевой поразрядной быстрой сортировки длинных строк для  $N = 10^3, 10^4, 10^5$  и  $10^6$ , при этом число выполняемых операций сравнений должно быть таким же, как и в случае стандартной быстрой сортировки для тех же файлов.
- **10.43.** Найти количество байтов, просматриваемых в процессе выполнения поразрядной сортировки MSD и трехпутевой поразрядной быстрой сортировки для файла с  $N$  ключами A, AA, AAA, AAAA, AAAAA, ...

## 10.7. Сортировки с сублинейным временем выполнения

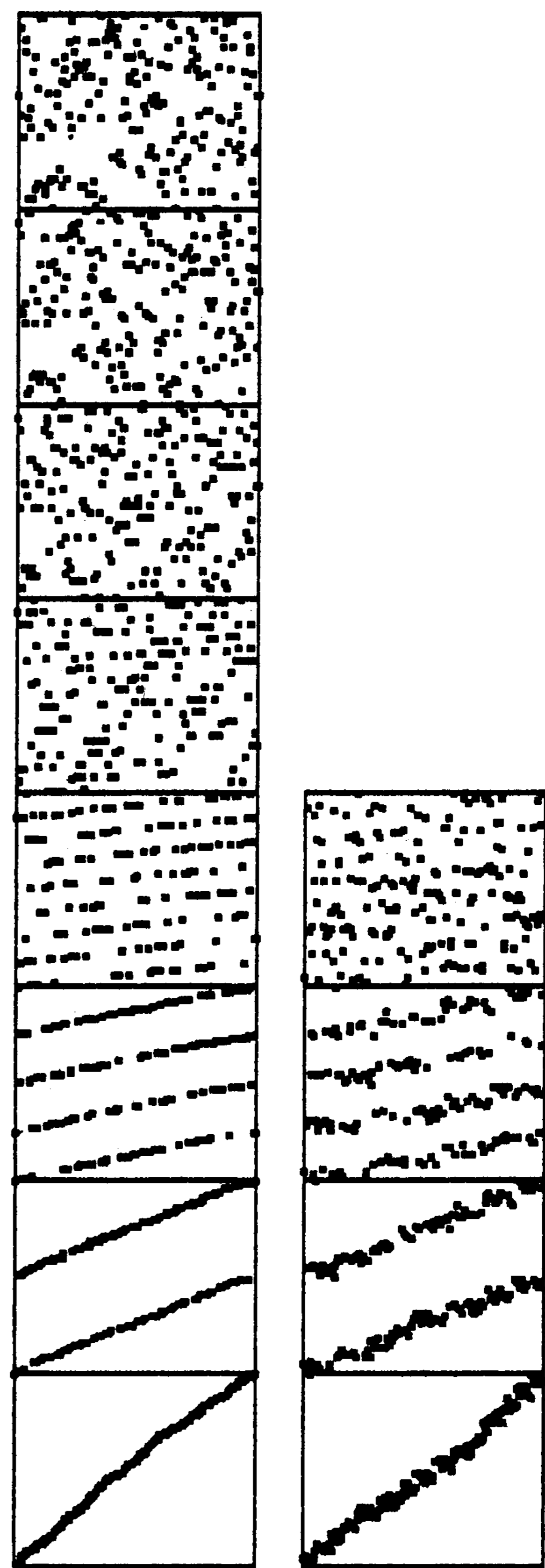
Основной вывод, который можно сделать по результатам анализа, проведенного в разделе 10.6, состоит в том, что время выполнения поразрядной сортировки может находиться в линейной зависимости от общего количества информации, заключенной в ключах. В данном разделе мы проанализируем практическое значение этого факта.

Реализация поразрядной сортировки LSD, представленная в разделе 10.5, выполняет **bytesword** проходов по файлу. Увеличивая  $R$ , мы получаем эффективный метод сортировки для достаточно больших  $N$  при наличии дополнительного пространства памяти для размещения  $R$  счетчиков. Как отмечалось при доказательстве леммы 10.5, целесообразно выбирать  $R$  таким, чтобы значение  $\ln R$  (число разрядов в байте) было примерно равно одной четвертой от размера слова, так чтобы поразрядная сортировка представляла собой четыре прохода сортировки методом подсчета индексных ключей. Проверяется каждый бит каждого ключа. Этот пример является прямой аналогией организационной архитектуры многих компьютеров: в одной из типичных организаций используется 32-разрядное слово, которое, в свою очередь, состоит из 8-разрядных байтов. Мы извлекаем из слов байты, а не биты, и этот подход позволяет достичь довольно высокой эффективности на многих типах компьютеров. Теперь каждый проход процедуры подсчета индексных ключей линеен, а поскольку их всего лишь четыре, то и вся сортировка линейна — вряд ли можно надеяться на что-либо лучшее, когда речь идет о сортировке.

В действительности дело обстоит так, что мы можем обойтись всего лишь двумя проходами процедуры подсчета индексных ключей. Мы сделаем это, воспользовавшись тем фактом, что файл будет практически отсортирован, если используются только  $w/2$  старших разрядов  $w$ -разрядных ключей. Как это имело место в случае быстрой сортировки, можно эффективно завершить сортировку, выполнив после этого сортировку простыми вставками для всего файла.

Этот метод представляет собой тривиальную модификацию программы 10.4. Чтобы выполнить сортировку слева направо, воспользовавшись для этой цели старшей половиной ключей, мы просто запускаем внешний цикл со значения `byteword/2-1`, а не с `byteword-1`. Затем мы применяем метод сортировки простыми вставками к почти упорядоченному файлу, который к этому моменту получаем. Рисунки 10.3 и 10.18 представляют собой убедительное доказательство того, что файл, отсортированный по старшим разрядам, достаточно хорошо упорядочен. Сортировка методом вставки выполняет всего лишь шесть операций обмена для сортировки файла, изображенного в четвертом столбце диаграммы на рис. 10.3, а на рис. 10.18 показано, что достаточно большие файлы, отсортированные только по старшей половине разрядов, могут быть эффективно упорядочены простыми вставками.

Для некоторых размеров файлов, возможно, имеет смысл использовать дополнительное пространство памяти, которое в других случаях будет отведено под вспомогательный массив, чтобы попытаться обойтись всего лишь одним проходом для подсчета индексных ключей, выполняя переупорядочение без использования дополнительной памяти. Например, сортировка 1 миллиона 32-разрядных ключей, принимающих случайные значения, может быть осуществлена за один проход сортировки методом подсчета индексных ключей на 20 старших разрядах и последующей сортировкой методом вставки. Чтобы выполнить эту процедуру, потребуется пространство памяти только для счетчиков (1 миллион) — значительно меньше, чем нужно для размещения вспомогательного массива. Использование этого метода равносильно использованию стандартной сортировки при  $R = 2^{20}$ , хотя очень важно, чтобы для файлов малых размеров применялись небольшие значения основания (см. обсуждение леммы 10.4).



**РИСУНОК 10.18. ДИНАМИЧЕСКИЕ ХАРАКТЕРИСТИКИ ПОРАЗРЯДНОЙ СОРТИРОВКИ LSD НА РАЗРЯДАХ, ИСПОЛЬЗУЕМЫХ ПОРАЗРЯДНОЙ СОРТИРОВКОЙ MSD**

*Когда ключами служат биты, принимающие случайные значения, сортировка файла по старшим разрядам ключей устанавливает на файле порядок, близкий к искомому. На этой диаграмме сортировка LSD с шестью проходами (слева) на файле со случайными 6-разрядными ключами сравнивается с трехпроходной поразрядной сортировкой, после которой может последовать еще один проход, на этот раз сортировки простыми вставками (справа). Последняя стратегия приводит к увеличению быстродействия примерно в два раза.*

Подход применительно к поразрядной сортировке получил широкое распространение в силу того, что он требует исключительно простых структур управления, а его базовые операции очень удобны для реализаций в машинном языке, который легко адаптируется к высокопроизводительным аппаратным средствам специального назначения. В такой среде наибольшее быстродействие, по-видимому, достигается при использовании полной поразрядной сортировки LSD. Если мы используем указатели, то чтобы воспользоваться поразрядной сортировкой LSD нам потребуется дополнительное пространство памяти для размещения  $N$  связей (и  $R$  счетчиков), и эти затраты позволяют реализовать метод, который может сортировать файлы с произвольной организацией всего лишь за три-четыре прохода.

В обычных средах программирования внутренний цикл программы, реализующей подсчет индексных ключей, который служит основой поразрядных сортировок, содержит намного большее число инструкций, чем внутренние циклы быстрой сортировки или сортировки слиянием. Из этого свойства программных реализаций следует, что описанные выше сублинейные методы во многих случаях не могут, вопреки нашим ожиданиям, обладать таким же быстродействием как, скажем, быстрая сортировка.

Алгоритмы общего назначения, такие как быстрая сортировка, нашли на практике более широкое применение, чем поразрядная сортировка, поскольку они могут адаптироваться к более широкому диапазону приложений. Основная причина подобного положения дел заключается в том, что абстракция ключа, на которой построена поразрядная сортировка обладает меньшей универсальностью, чем та, которая использовалась в главах 6—9. Например, один из широко распространенных способов установления интерфейса со служебной программой сортировки заключается в том, чтобы клиент сам выбирал функцию сравнения. Таким интерфейсом является интерфейс, используемый программой `qsort` из библиотеки программ на C++. Это программное средство не только годится в ситуации, когда клиент может воспользоваться специальными сведениями о сложных ключах с целью реализации быстрого сравнения, но также позволяет выполнять сортировку, используя отношения порядка, которые вообще могут обходиться без ключей. Мы рассмотрим один такой алгоритм в главе 21.

Когда какой-либо из них может быть использован, выбор между быстрой сортировкой и различными алгоритмами поразрядной сортировки (и имеющие к ним отношение различные версии быстрой сортировки!) из числа тех, которые мы рассматривали в данной главе, будет зависеть не только от свойств приложения (таких как ключи, размеры записи и файла), но также и от свойств среды программирования и аппаратных средств, от которых зависят эффективность доступа и использование отдельных битов и байтов. В табл. 10.1 и 10.2 приводятся эмпирические результаты, свидетельствующие в пользу вывода о том, что линейная или сублинейная зависимость рабочих характеристик, которые мы рассматривали применительно к различным приложениям поразрядной сортировки, делают эти методы сортировок привлекательными для различных подходящих приложений.

**Таблица 10.1. Эмпирический анализ поразрядных сортировок (ключи в виде целых чисел)**

Приводимые в таблице в относительных единицах временные показатели различных поразрядных сортировок применительно к файлам с  $N$  32-разрядных чисел произвольной организацией (во всех применяется отсечение для  $N$  меньше 16 для последующей сортировки методом простой вставки) показывают, что поразрядные сортировки входят в число самых быстрых, если соблюдать осторожность при их использовании. Если мы используем огромное основание системы счисления для крошечных файлов, мы просто сводим на нет все преимущества поразрядной сортировки MSD, однако если мы выберем в качестве такого основания величину, которая меньше размера файла, то все ее достоинства восстанавливаются. Самый быстрый метод сортировки ключей в виде целых чисел является поразрядная сортировка, примененная к старшей половине ключей, быстродействие которого мы можем повысить еще больше, если уделим надлежащее внимание внутреннему циклу (см. упражнение 10.45).

$N$	4-разрядные байты			8-разрядные байты			16-разрядные байты		
	Q	M	L	M	L	L*	M	L	M*
12500	2	7	11	28	4	2	52	5	8
25000	5	14	21	29	8	4	54	8	15
50000	10	49	43	35	18	9	58	15	39
100000	21	77	92	47	39	18	67	30	77
200000	49	133	185	72	81	39	296	56	98
400000	102	278	377	581	169	88	119398	110	297
800000	223	919	732	6064	328	203	1532492	219	2309

Обозначения:

- Q** Быстрая сортировка, стандартная (программа 7.1)
- M** Поразрядная сортировка, стандартная (программа 10.2)
- L** Поразрядная сортировка LSD (программа 10.4)
- M\*** Поразрядная сортировка MSD, основание системы счисления адаптируется под размер файла
- L\*** Поразрядная сортировка LSD на разрядах MSD.

**Таблица 10.2. Эмпирические исследования поразрядных сортировок (строковые ключи)**

Приводимые в таблице в относительных единицах временные показатели различных поразрядных сортировок первых слов  $N$  слов из книги *Моби Дик (Moby Dick)* (все виды сортировок, за исключением пирамидальной сортировки, с отсечением при  $N$  меньше 16 для последующего выполнения сортировки простыми вставками) показывают, что подход "сначала MSD" эффективен применительно к строковым данным. Отсечение малых подфайлов менее эффективен в условиях трехпутевой поразрядной быстрой сортировки, чем другие методы, и совсем не эффективен, если не проводить модификацию сортировки методом вставки во избежание прохода через старшие разряды ключей (см. упражнение 10.46).

<i>N</i>	<i>Q</i>	<i>T</i>	<i>M</i>	<i>F</i>	<i>R</i>	<i>X</i>	<i>X*</i>
12500	7	6	9	9	8	6	5
25000	14	12	18	19	15	11	10
50000	34	26	39	49	34	25	24
100000	83	61	87	114	71	57	54

Обозначения:

- Q** Быстрая сортировка, стандартная (программа 7.1)
- T** Быстрая сортировка с трехпутевым разбиением (программа 7.5)
- M** Сортировка слиянием (программа 8.2)
- F** Пирамидальная сортировка с усовершенствованием Флойда (см. раздел 9.4)
- R** Поразрядная сортировка MSD (программа 10.2)
- X** Трехпутевая поразрядная быстрая сортировка MSD (программа 10.3)
- X\*** Трехпутевая поразрядная быстрая сортировка MSD (с отсечениями)

## Упражнения

- ▷ **10.44.** Каким является основной недостаток выполнения на старших битах ключей с последующей подчисткой нарушений искомого порядка при помощи сортировки простыми вставками?
- **10.45.** Разработать программную реализацию поразрядной сортировки LSD на 32-разрядных ключах с минимально возможным числом команд во внутреннем цикле.
- 10.46.** Реализовать трехпутевую поразрядную быструю сортировку таким образом, чтобы сортировка методом вставок файлов небольших размеров не использовала старшие байты, о которых известно, что они равны.
- 10.47.** Имея 1 миллион 32-разрядных ключей, найти такой размер байта, для которого время выполнения сортировки будет минимальным в условиях, когда используется метод, предусматривающий поразрядную сортировку LSD по двум первым байтам и последующую подчистку нарушений искомого порядка через сортировку простыми вставками.
- 10.48.** Выполнить упражнение 10.47 для 1 миллиарда 64-разрядных ключей.
- 10.49.** Выполнить упражнение 10.48 для трехпутевой поразрядной сортировки LSD.

## Методы сортировки специального назначения

Методы сортировки являются критическими компонентами многих прикладных систем. Довольно часто предпринимаются специальные меры, чтобы сделать сортировку максимально быстрой или придать ей способность выполнять обработку особо крупных файлов. Нередки случаи, когда специально для того, чтобы повысить возможности сортировки, в вычислительные системы вносятся усовершенствования, повышающие ее производительность, разрабатываются специальные аппаратные средства или просто выбираются компьютерные системы, в основу которых положены новые архитектурные решения. Во всех подобных случаях представления об относительной стоимости операций, выполняемых над сортируемыми данными, которыми мы руководствуемся, могут оказаться непригодными. В данной главе мы исследуем примеры методов сортировки, которые разрабатываются для эффективного применения на конкретных типах машин. Мы рассмотрим некоторые примеры ограничений, налагаемых на высокопроизводительные аппаратные средства, а также несколько методов, которые могут оказаться полезными на практике в плане реализации высокоэффективных видов сортировки.

Проблема поддержки эффективных методов сортировки рано или поздно встает перед *любой* новой архитектурой вычислительных систем. В самом деле, исторически сложилось так, что сортировка служит своеобразным испытательным стендом, позволяющим получить объективную оценку новой архитектуры ввиду того, что она имеет важное практическое значение, а также благодаря тому, что она легка для восприятия. Мы хотим знать не только то, какой из известных алгоритмов и в силу каких

причин выполняется на новой машине лучше других, но также и то, можно ли в рамках того или иного нового алгоритма воспользоваться конструктивными особенностями конкретной машины. Чтобы разработать новый алгоритм, мы даем определение некоторой абстрактной машины, которая инкапсулирует основные свойства реальной машины, разрабатываем и анализируем алгоритмы для этой абстрактной машины, затем пишем программы лучших алгоритмов, тестируем эти программные реализации, после чего вносим усовершенствования как в сами алгоритмы, так и в их программные реализации. С этой целью мы воспользуемся сведениями, полученными нами при изучении материалов, изложенных в главах 6–10, в том числе и описаниями многих методов, ориентированных на выполнение на машинах общего назначения. В то же время абстрактные машины налагают на эти алгоритмы ряд ограничений, которые помогут нам учесть реальные затраты и убедиться в том, что каждый алгоритм достигает максимальной эффективности на том или ином конкретном виде машины.

На одном конце спектра находятся модели низкого уровня, в рамках которых разрешается выполнение только операции сравнения-обмена. На другом конце спектра находятся модели высокого уровня, в условиях которых мы выполняем считывание и запись блоков данных больших объемов на медленнодействующие внешние носители информации или распределяем данные между параллельными процессорами.

В первую очередь мы рассмотрим версию сортировки слиянием, известную как *нечетно-четная сортировка слиянием Бэтчера (Batcher's odd-even mergesort)*. В ее основу положен алгоритм слияния, функционирующий по принципу *разделяй и властвуй*, который использует только операции сравнения-обмена, при этом для перемещения данных употребляются операции *идеального тасования (perfect-shuffle)* и операция *идеального обратного тасования (perfect unshuffle)*. Они представляют интерес сами по себе и применяются для решения проблем, отличных от сортировки. Далее мы будем рассматривать метод Бэтчера как сеть сортировки. *Сеть сортировки (sorting network)* представляет собой простую абстракцию для аппаратных средств сортировки. Такая сеть состоит из соединенных друг с другом посредством межкомпонентных связей *компараторов (comparators)*, представляющих собой модули, способные выполнять операции сравнения-обмена.

Другой важной проблемой абстрактной сортировки является проблема *внешней сортировки (external sorting)*; в этом случае сортируемый файл обладает такими огромными размерами, что не помещается в оперативной памяти. Стоимость доступа к индивидуальным записям может оказаться непомерно высокой, в силу этого обстоятельства мы должны использовать абстрактную модель, в которой записи передаются между внешними устройствами в виде блоков больших размеров. Мы рассмотрим два алгоритма внешней сортировки и воспользуемся соответствующей моделью, чтобы сравнить их между собой.

В заключение мы рассмотрим параллельную сортировку (*parallel sorting*) на тот случай, когда сортируемый файл распределяется между независимыми параллельными процессорами. Мы дадим определение простой модели параллельной машины, а затем выясним, при каких условиях метод Бэтчера обеспечит эффективное решение этой проблемы. Использование одного и того же базового алгоритма для проблем

высокого уровня и низкого уровня служит наглядной иллюстрацией того, какой мощью обладает абстракция.

Различные абстрактные машины, рассматриваемые в этой главе, достаточно просты, однако заслуживают подробного изучения, поскольку инкапсулируют в себе конкретные ограничения, которые могут оказаться критичными для некоторых приложений сортировки. Аппаратные средства сортировки низкого уровня должны состоять из простых компонентов; внешняя сортировка в общем случае требует поблочного доступа к особо крупным файлам данных, а параллельная сортировка накладывает определенные ограничения на связи с процессорами. С одной стороны, мы не можем в полной мере воспользоваться подробной моделью машины, которая полностью соответствует конкретной реальной машине, с другой стороны, абстракции, которые мы рассматриваем, приводят нас не только к теоретическим формулировкам, содержащим информацию о наиболее важных ограничениях, но также и к весьма интересным алгоритмам, которые можно использовать непосредственно на практике.

## 11.1 Четно-нечетная сортировка слиянием Бэтчера

Для начала мы рассмотрим метод сортировки, в основе которых лежат две следующих абстрактных операции: операция *сравнения обмена* (*compare-exchange*) и операция *идеального тасования* (*perfect shuffle*) (вместе с ее антиподом, операцией *идеального обратного тасования* (*perfect unshuffle*)). Алгоритм, разработанный Бэтчером в 1968 г., известен как *нечетно-четная сортировка слиянием Бэтчера* (*Batcher's odd-even mergesort*). Реализовать алгоритм, используя операции тасования, сравнения-обмена и двойной рекурсии, несложно; гораздо труднее понять, почему алгоритм работает, и распутать все тасования и рекурсии, чтобы понять, как он работает на нижнем уровне.

Мы уже подвергали беглому анализу операцию сравнения-обмена в главе 6, где отметили, что некоторые рассматриваемые при этом элементарные методы сортировки могут быть четко выражены в терминах этой операции. В настоящий момент нас интересуют методы, в условиях которых данные проверяются исключительно через операций сравнения-обмена. Стандартные операции сравнения исключаются: операции сравнения-обмена не возвращают результат, следовательно, у программы нет возможности выполнять те или иные действия в зависимости от конкретных значений данных.

**Определение 11.1** *Неадаптивный алгоритм сортировки* — это алгоритм, в котором последовательность выполняемых операций зависит только от числа входных данных, а не от значений ключей.

В этом разделе мы допускаем использование операций, которые в одностороннем порядке выполняют переупорядочивание данных, такие как операции обмена и тасования, но без этих операций, как мы увидим в разделе 11.2, можно обойтись. Неадаптивные методы эквивалентны неветвящимся программам сортировки: они могут быть записаны в виде простого перечня операций сравнения-обмена.



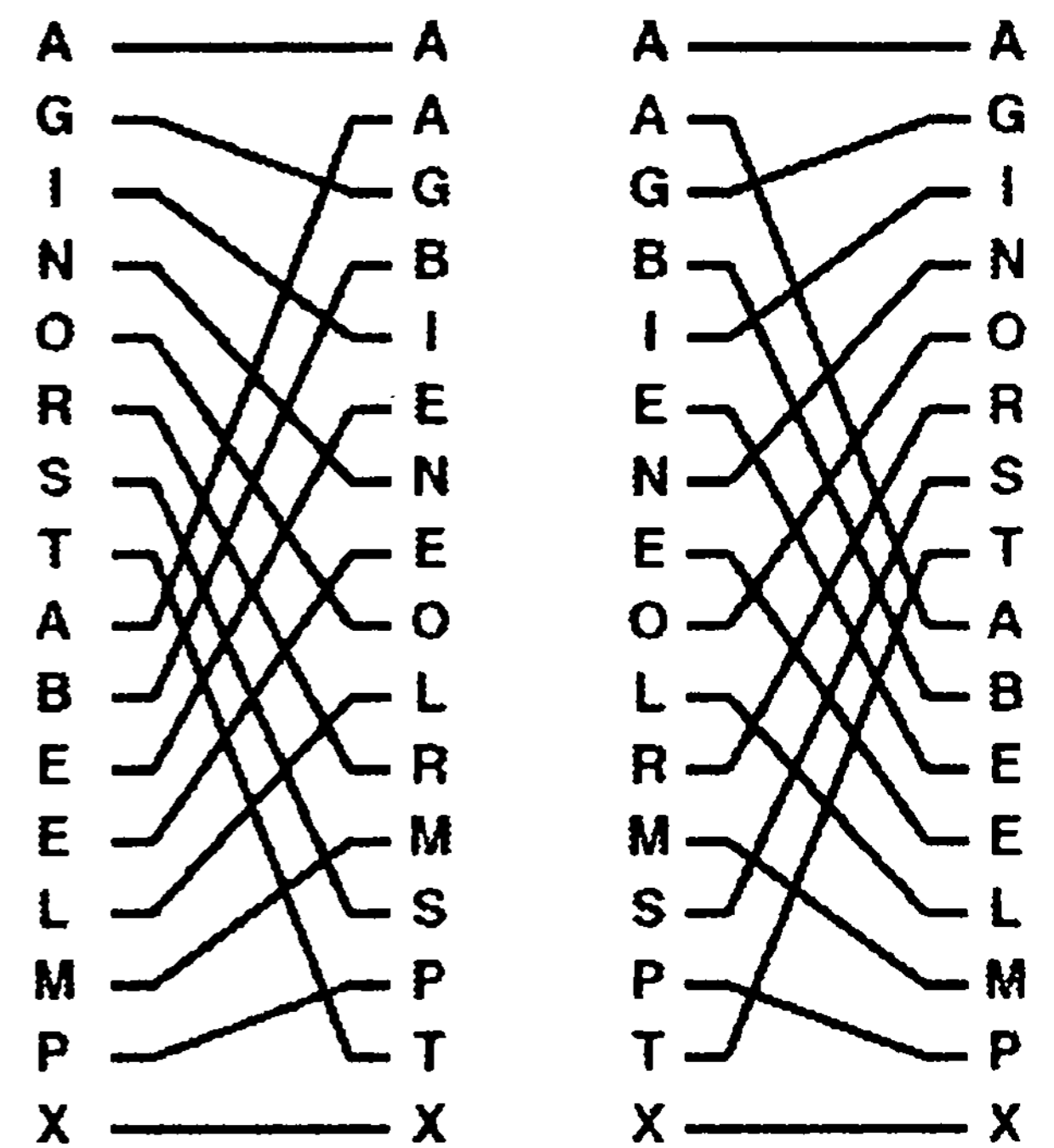
Например, последовательность

```
comrexch(a[0], a[1])
comrexch(a[1], a[2])
comrexch(a[0], a[1])
```

суть неветвящаяся программа, выполняющая сортировку трех элементов. Мы используем циклы, тасования и другие операции высокого уровня, преследуя цель удобной и экономной записи алгоритма, однако основная цель при разработке алгоритма состоит в том, чтобы определить для каждого  $N$  фиксированную последовательность операций **comrexch**, которые способны выполнить сортировку любого набора из  $N$  ключей. Мы можем без ущерба для общности рассуждений предположить, что ключи принимают целочисленные значения в диапазоне от 1 до  $N$  (см. упражнение 11.4); чтобы убедиться в том, что неветвящаяся программа работает правильно, достаточно доказать, что она сортирует каждую из возможных перестановок этих значений (см., например, упражнение 11.5).

Лишь немногие из алгоритмов, которые мы успели обсудить в главах 6–10, являются неадаптивными алгоритмами — все они используют операцию **operator<** либо проверяют ключи каким-то другим способом, после чего выполняют действия в зависимости от значений ключей. Исключением является пузырьковая сортировка (см. раздел 6.4), в условиях которой используются только операции сравнения-обмена. Версия Пратта сортировки методом Шелла (см. раздел 6.6) служит еще одним примером неадаптивного метода сортировки.

Программа 11.1 представляет собой реализацию других абстрактных операций, которыми мы будем пользоваться в дальнейшем — операции идеального тасования и операции обратного идеального тасования (на рис. 11.1 дается пример каждой из них). Идеальное тасование переупорядочивает массив так, как может перетасовать колоду карт только большой специалист этого дела: колода делится точно наполовину, затем карты по одной берутся из каждой половины колоды. Первая карта всегда берется из верхней половины колоды. Если число карт в колоде четное, в обеих половинах содержится одинаковое их число, если число карт нечетное, то лишняя карта идет последней в верхней половине колоды.



**РИСУНОК 11.1. ИДЕАЛЬНОЕ ТАСОВАНИЕ И ОБРАТНОЕ ИДЕАЛЬНОЕ ТАСОВАНИЕ**

*При выполнении идеального тасования (слева) мы берем первый элемент файла, затем первый элемент из второй половины файла, затем второй элемент файла и второй элемент из второй половины файла и т. д.*

*Предположим, что элементы перенумерованы сверху вниз, начиная с 0. Следовательно, элементы из первой половины занимают позиции с четными номерами, а элементы из второй половины занимают нечетные позиции. Чтобы выполнить обратное идеальное тасование (справа), мы должны выполнить обратную процедуру: элементы, занимающие четные позиции, переходят в первую половину файла, элементы, занимающие нечетные позиции, — во вторую половину.*

Обратное идеальное тасование выполняет обратную процедуру: мы попеременно откладываем карты в верхнюю и нижнюю половину колоды.

### Программа 11.1 Идеальное тасование и обратное идеальное тасование

Функция **shuffle** переупорядочивает подмассив  $a[1], \dots, a[r]$  путем деления этого подмассива пополам, после чего попеременно берет элементы из каждой половины массива: элементы из первой половины переходят в позиции с четными номерами получающегося при этом массива, а элементы из второй половины переходят в позиции с нечетными номерами. Функция **unshuffle** выполняет обратную процедуру: элементы, занимающие четные позиции, переходят в первую половину получающегося при этом массива, а элементы, занимающие нечетные позиции, переходят во вторую половину. Мы применяем эти функции только к подмассивам, содержащим четное число элементов.

```
template <class Item>
void shuffle(Item a[], int l, int r)
{ int i, j, m = (l+r)/2;
  static Item aux[maxN];
  for (i = l, j = 0; i <= r; i+=2, j++)
    { aux[i] = a[l+j]; aux[i+1] = a[m+1+j]; }
  for (i = l; i <= r; i++) a[i] = aux[i];
}

template <class Item>
void unshuffle(Item a[], int l, int r)
{ int i, j, m = (l+r)/2;
  static Item aux[maxN];
  for (i = l, j = 0; i <= r; i+=2, j++)
    { aux[l+j] = a[i]; aux[m+1+j] = a[i+1]; }
  for (i = l; i <= r; i++) a[i] = aux[i];
}
```

Сортировка Бэтчера представляет собой в точности нисходящую сортировку слиянием, описанную в разделе 8.3; различие состоит лишь в том, что вместо одной из адаптивных реализаций слияний из главы 8 она использует нечетно-четное слияние Бэтчера, представляющее собой неадаптивное нисходящее рекурсивное слияние. Программа 8.3 сама по себе вообще не выполняет доступа к данным, так что из факта использования нами неадаптивного слияния следует, что и сама сортировка неадаптивная.

На протяжении этого раздела и раздела 11.2 мы неявно предполагаем, что число сортируемых элементов является степенью 2. Следовательно, мы всегда можем употребить значение " $N/2$ ", не опасаясь того, что  $N$  — нечетное. Это предположение условно — рассматриваемые нами программы и примеры рассчитаны на любые размеры файлов, тем не менее, оно существенно упрощает наши рассуждения. К этой проблеме мы вернемся в конце раздела 11.2.

Слияние Бэтчера само по себе является рекурсивным методом "разделяй и властвуй". Чтобы выполнить слияние 1-с-1, мы употребляем только одну операцию сравнения-обмена. Во всех прочих случаях, для выполнения слияния  $N$ -с- $N$  мы осуществляем обратное тасование, чтобы свести эту задачу к двум задачам слияния  $N/2$ -с- $N/2$ , после чего решаем их в рекурсивном режиме, в результате чего получаем два отсортированных файла. Тасуя эти файлы, мы получаем почти отсортированный файл; все, что теперь нам нужно — это один проход для выполнения  $N/2 - 1$  независимых друг

от друга операций сравнения-обмена между элементами  $2i$  и  $2i + 1$ , где  $i$  пробегает значения от 1 до  $N/2 - 1$ . Соответствующий пример показан на рис.11.2. Это описание позволяет без труда написать программу 11.2.

Почему этот метод сортирует все возможные перестановки входных данных? Ответ на этот вопрос далеко не очевиден; классическое доказательство этого факта — это на самом деле косвенное доказательство, которое построено на общих характеристиках неадаптивных программ сортировки.

### Программа 11.2. Нечетно-четное слияние Бэтчера (рекурсивная версия)

Эта рекурсивная программа реализует абстрактное обменное слияние, используя для этой цели операции **shuffle** и **unshuffle** из программы 11.1, хотя это и не обязательно — программа 11.3 представляет собой нерекурсивную версию данной программы, в которой тасование не используется. Основным интересом для нас представляет тот факт, что рассматриваемая реализация является компактным описанием алгоритма Бэтчера, когда размер файла является степенью 2.

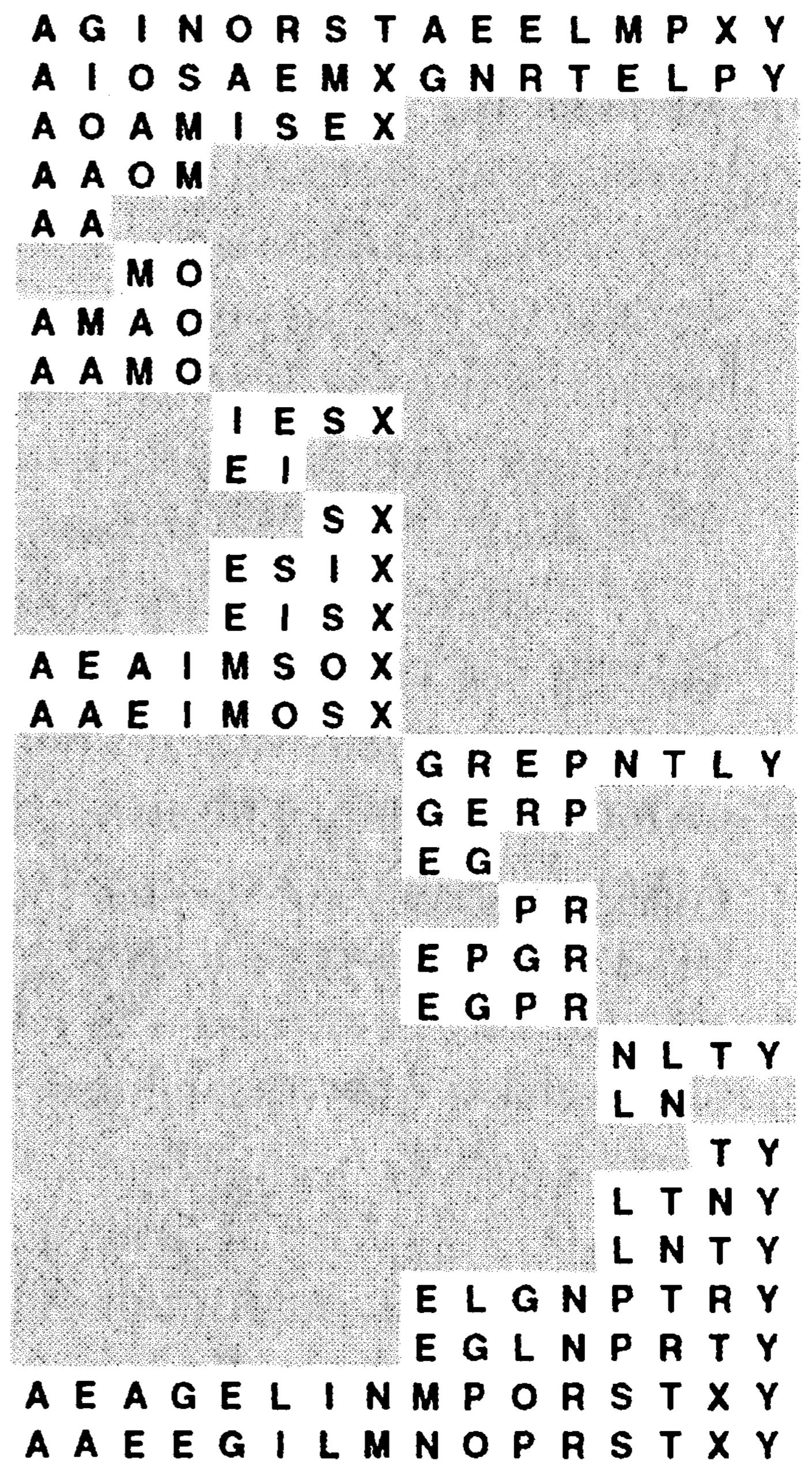
```
template <class Item>
void merge(Item a[], int l, int m, int r)
{
    if (r == l+1) comexch(a[l], a[r]);
    if (r < l+2) return;
    unshuffle(a, l, r);
    merge(a, l, (l+m)/2, m);
    merge(a, m+1, (m+1+r)/2, r);
    shuffle(a, l, r);
    for (int i = l+1; i < r; i+=2)
        comexch(a[i], a[i+1]);
}
```

**Лемма 11.1.** (Принцип нулей и единиц) Если неадаптивная программа выдает отсортированный выход в случае, когда входы состоят только из 0 и 1, то она делает то же, когда входами являются произвольные ключи.

См. упражнение 11.7.

**Лемма 11.2.** Нечетно-четное слияние Бэтчера (программа 11.2) это правильный метод слияния

Опираясь на принцип нулей и единиц, мы проверяем только, правильно ли работает метод слияния, когда входами являются нули и единицы.



**РИСУНОК 11.2. ПРИМЕР ВЫПОЛНЕНИЯ НИСХОДЯЩЕГО НЕЧЕТНО-ЧЕТНОГО СЛИЯНИЯ БЭТЧЕРА**

Чтобы слить  $A G I N O R S T$  с  $A E E L M P X Y$ , мы начнем с того, что выполним операцию обратного тасования, которая приводит к возникновению двух независимых друг от друга задач слияния наполовину меньших массивов (показаны во второй строке): нам надо слить  $A I O S$  с  $A E M X$  (в первой части массива) и  $G N R T$  с  $E L P Y$  (во второй части массива). После того, как эти подзадачи будут решены в рекурсивном режиме, мы тасуем массивы, полученные в результате решения этих подзадач (полученные в предпоследней строке) и завершаем эту сортировку выполнением операций сравнения-обмена  $E$  с  $A$ ,  $G$  с  $E$ ,  $L$  с  $I$ ,  $N$  с  $M$ ,  $P$  с  $O$ ,  $R$  с  $S$  и  $T$  с  $X$ .

Предположим, что в первом подфайле содержатся  $i$  нулей и  $j$  нулей во втором подфайле. Доказательство этой леммы требует рассмотрения четырех случаев в зависимости от того, являются ли  $i$  и  $j$  четными или нечетными числами. Если обе переменные имеют четные значения, то две подзадачи слияния выполняется над двумя файлами, при этом один файл содержит  $i/2$  нулей, а другой файл —  $j/2$  нулей, так что в обоих случаях получившийся при слиянии файл содержит  $(i+j)/2$  нулей. Выполнив тасование, получим сортированный файл типа 0-1. Файл типа 0-1 подвергается тасованию и в тех случаях, когда  $i$  — четное, а  $j$  — нечетное число, а также когда  $i$  — нечетное, а  $j$  — четное число. Но если оба  $i$  и  $j$  — нечетные числа, то мы завершаем эту процедуру тем, что тасуем файл, содержащий  $(i+j)/2 + 1$  нулей с файлом, содержащим  $(i+j)/2 - 1$  нулей, следовательно, полученный после тасования файл содержит  $i+j-1$  нулей, единицу, ноль и  $N-i-j-1$  единиц (см. рис. 11.3), и один из компараторов на завершающей стадии заканчивает сортировку.

На самом деле нет необходимости тасовать данные. И действительно, мы можем переделать программы 11.2 и 8.3 таким образом, чтобы на выходе они имели неветвящуюся сортирующую программу для любого  $N$ , скорректировав функции `comprexch` и `shuffle` с тем, чтобы они поддерживали индексы и осуществляли косвенные обращения к данным (см. упреждение 11.12). Либо мы можем заставить эту программу генерировать последовательность команд сравнения-обмена для последующего применения к исходному входному файлу (см. упражнение 11.13). Мы можем использовать эти приемы для любого неадаптивного метода сортировки, который выполняет переупорядочивание данных при помощи операций обмена, тасования или им подобных. Что касается слияния Бэтчера, то структура этого алгоритма настолько проста, что мы сразу можем приступить к разработке восходящей программной реализации, в чем убедимся уже в разделе 11.2.

```

0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1
0 1 1 1 0 0 1 1 0 1 1 1 0 0 1 1
0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1

0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1
0 1 1 1 0 0 1 1 0 1 1 1 0 1 1 1
0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1

0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1
0 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1
0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1

0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 1
0 0 1 1 0 0 1 1 0 1 1 1 0 1 1 1
0 0 0 0 1 1 1 1 0 0 1 1 1 1 1 1
0 0 0 0 0 1 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1

```

**РИСУНОК 11.3. ЧЕТЫРЕ СЛУЧАЯ СЛИЯНИЯ ТИПА 0-1.**

Каждому из четырех примеров отводится по 5 строк: задача слияния типа 0-1, результат выполнения операции обратного тасования, который порождает две подзадачи слияния; результат рекурсивного завершения слияний; результат тасования и результат завершающих нечетно-четных сравнений. На последней стадии обмен выполняется только, когда число нулей в обоих входных файлах нечетно.

## Упражнения

- ▷ 11.1. Покажите результат тасования и обратного тасования ключей **E A S Y Q U E S T I O N**.
- 11.2. Обобщить программу 11.1 на случай, когда нужно выполнить  $h$ -путевое тасование и обратное тасование.
- 11.3. Реализовать операции тасования и обратного тасования без использования вспомогательного массива.

- 11.4. Показать, что неветвящаяся программа, которая сортирует  $N$  различных ключей, сможет отсортировать  $N$  ключей, которые не обязательно различны.
- ▷ 11.5. Показать, как неветвящаяся программа, приведенная в тексте, сортирует каждую из шести перестановок чисел 1, 2 и 3.
- 11.6. Приведите пример неветвящейся программы, которая выполняет сортировку четырех элементов.
- 11.7. Доказать лемму 11.1. *Совет:* Показать, что если программа не способна выполнить сортировку некоторого входного массива с произвольными ключами, то существует некоторая последовательность типа 0-1, которую она также не может отсортировать.
- ▷ 11.8. Показать, как выполняется слияние ключей A E Q S U Y E I N O S T с помощью программы 11.2 в стиле примера, представленного на диаграмме рис.11.2.
- ▷ 11.9. Выполнить упражнение 11.8 для ключей A E S Y E I N O Q S T U.
- 11.10. Выполнить упражнение 11.8 для ключей 1 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0.
- 11.11. Эмпирически выполнить сравнение времени выполнения сортировки слиянием Бэтчера аналогичным параметром стандартной нисходящей сортировки слиянием (программы 8.3 и 8.2) для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 11.12. Предложите реализации функций **complexch**, **shuffle** и **unshuffle**, которые заставляют программы 11.2 и 8.3 работать в режиме непрямой сортировки (см. раздел 6.8).
- 11.13. Предложите реализации функций **complexch**, **shuffle** и **unshuffle**, которые заставляют программы 11.2 и 8.3 печатать для заданного  $N$  неветвящуюся программу сортировки  $N$  элементов. Вы можете воспользоваться вспомогательным глобальным массивом для отслеживания значений индексов.
- 11.14. Если мы представим второй файл из предназначенных для слияния в обратном порядке, мы получим *битонную (bitonic)* последовательность в соответствии с определением, данным в разделе 8.3. Внося изменения в заключительный цикл программы 11.2 так, чтобы он начинался с 1, а не с  $l + 1$ , мы получим программу, которая сортирует битонные последовательности. Показать, как с помощью этого метода выполняется слияние ключей A E S Q U Y T S O N I E в стиле примера, показанного на диаграмме на рис. 11.2.
- 11.15. Доказать, что модификация программы 11.2, описанная в упражнении 11.14, способна выполнить любой битонной последовательности.

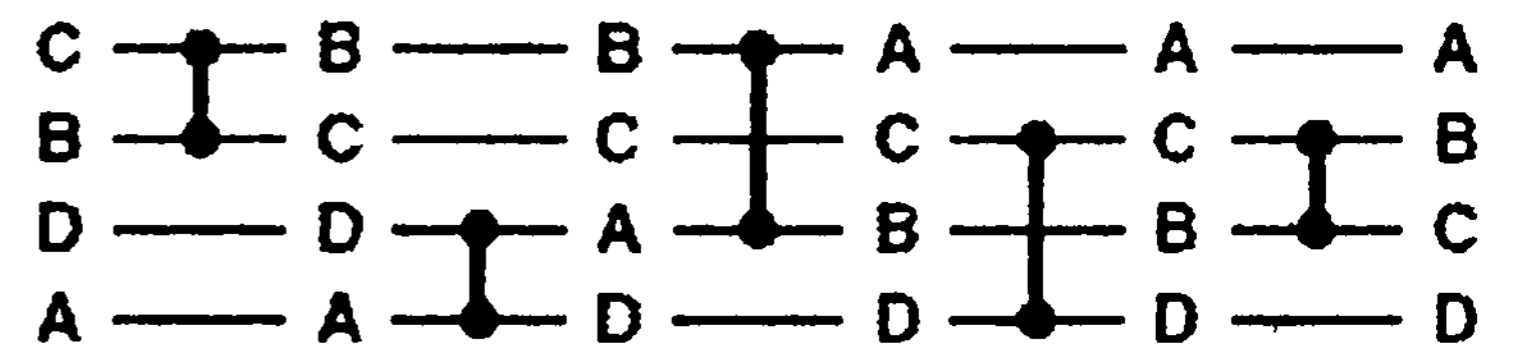
## 11.2. Сети сортировки

Простейшей моделью для изучения неадаптивных алгоритмов сортировки является абстрактная машина, которая способна осуществлять доступ к данным *только* с помощью операций сравнения-обмена. Такая машина называется *сетью сортировки (sorting network)*. Сеть сортировки построена из атомарных *модулей сравнения-обмена (compare-exchange modules)* или *компараторов (comparators)*, которые соединены между собой линиями связи таким образом, что становится возможным выполнение полной сортировки общего вида.

На рис. 11.4 показана простая сеть, выполняющая сортировку четырех ключей. Обычно мы изображаем сеть сортировки  $N$  элементов в виде последовательности  $N$  горизонтальных прямых линий, при этом каждый компаратор соединен с двумя линиями. Можно считать, что сортируемые ключи проходят по сети справа налево, при этом при необходимости происходит обмен пары чисел, в результате которого меньшее из двух чисел подымается вверх всякий раз, когда на их пути возникает компаратор.

Необходимо снять целый ряд вопросов, прежде чем станет возможным построение реальной сортирующей машины, работающей по этой схеме. Например, не описан метод кодирования входов. В рамках одного из подходов каждую связь на рис. 11.4 можно рассматривать как группу линий, каждая такая линия несет один бит данных, следовательно, все биты ключа распространяются по линии одновременно. Другой подход заключается в том, что данные поступают на входы компараторов по одной линии побитно, т.е. 1 бит за единицу времени (первыми идут наиболее значащие биты). Кроме того, совершенно не затрагивался вопрос синхронизации: должны быть предусмотрены механизмы, препятствующие срабатыванию компараторов, прежде чем поступят входные данные. Сети сортировки представляют собой полезные абстракции, поскольку они позволяют нам отделять детали реализации от проектных решений более высокого порядка, таких как, например, минимизация числа компараторов. Более того, как мы убедимся в разделе 11.5, абстрактное понятие сети сортировки может быть с пользой применено и в приложениях, отличных от построения электронных схем различного назначения.

Еще одним важным применением сетей сортировки является модель параллельных вычислений. Если два компаратора не используют одних и тех же линий для ввода данных, то мы полагаем, что они могут работать одновременно. Например, сеть, изображенная на рис. 11.4, показывает, что четыре элемента могут быть отсортированы за три параллельных шага. Компаратор 0-1 и компаратор 2-3 могут работать одновременно на первом шаге, после чего компаратор 0-2 и компаратор 1-3 могут одновременно работать на втором шаге, а компаратор 2-3 завершает сортировку на третьем шаге. Для любой заданной сети нетрудно сгруппировать компараторы в последовательность параллельных каскадов (*parallel stage*), каждый из которых состоит из компараторов, которые могут работать одновременно (см. упражнение 11.17). Чтобы параллельные вычисления были эффективными, нашей задачей становится разработка сетей с минимально возможным числом параллельных каскадов.

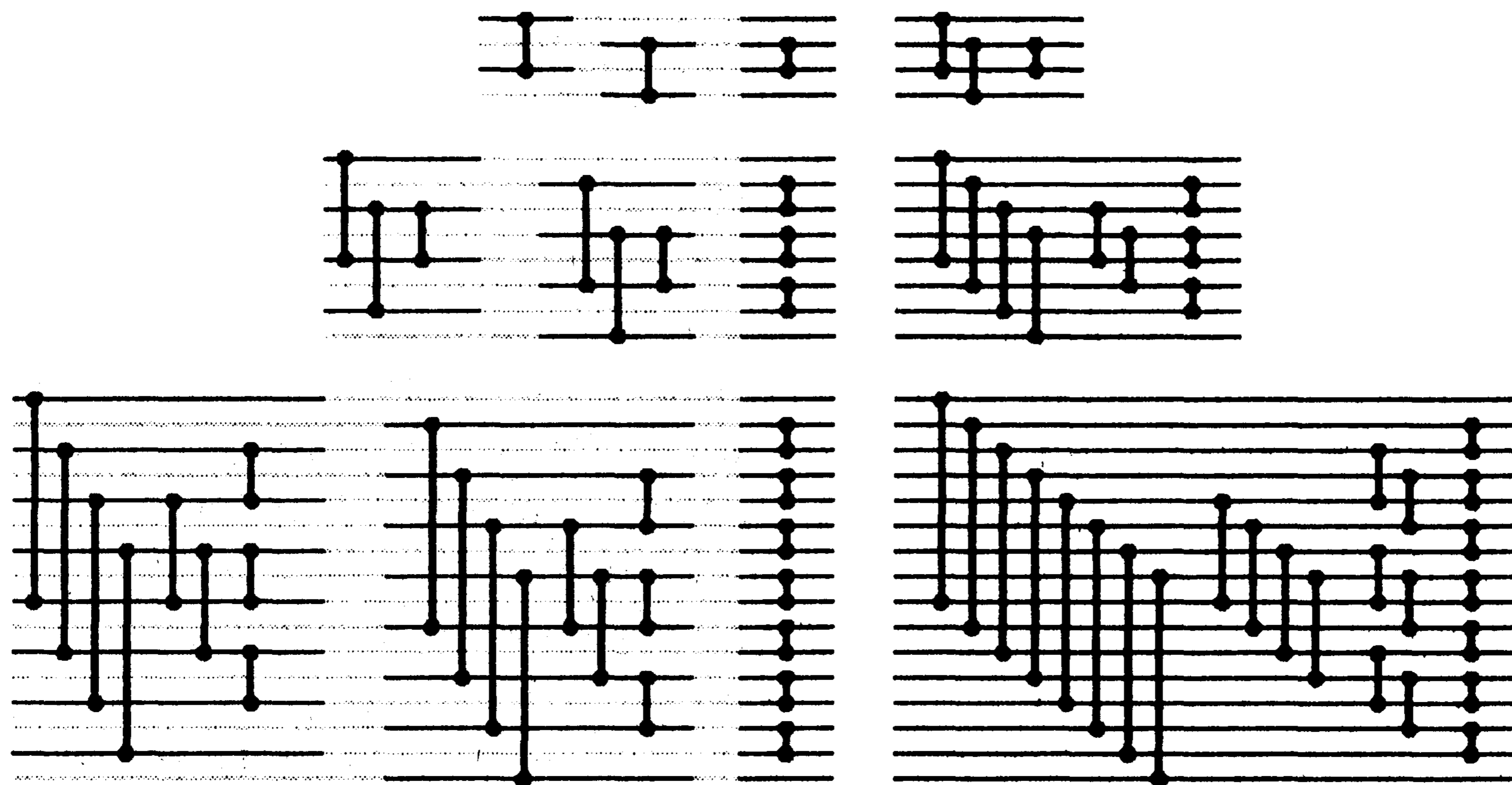


**РИСУНОК 11.4. СЕТЬ СОРТИРОВКИ**

*Ключи перемещаются по линиям сети сортировки слева направо. Компараторы, которые они встречают на своем пути, производят при необходимости обмен ключей, в результате которого меньший из двух ключей поднимается на верхнюю линию. В рассматриваемом примере на двух верхних линиях производится обмен ключей  $B$  и  $C$ , за ним следует обмен  $A$  и  $D$  на двух нижних линиях, затем происходит обмен  $A$  и  $B$  и так далее, так что в конечном итоге ключи предстают перед нами отсортированными сверху вниз. В этом примере все компараторы, за исключением четвертого, выполняют операцию обмена. Такая сеть способна отсортировать любые перестановки из четырех ключей.*

Программа 11.2 непосредственно соответствует сети слияния для каждого  $N$ , в то же время нам полезно ознакомиться с прямой восходящей структурой, изображенной на рис.11.5. Чтобы построить сеть слияния размера  $N$  мы воспользуемся двумя копиями сети размером  $N/2$ ; одна из них предназначена для линий с четными номерами, а другая — для линий с нечетными номерами. Поскольку соответствующие два набора компараторов не пересекаются, мы можем разместить их таким образом, чтобы обе эти сети чередовались. После этого мы расставим компараторы между линиями 1 и 2, 3 и 4 и т.д. Чередование нечетных и четных линий заменяет идеальное тасование, использованное в программе 11.2. Доказательство того, что эти сети выполняют слияние должным образом, аналогично доказательствам лемм 11.1 и 11.2, для которых применялся принцип нулей и единиц. На рис.11.6 показан пример выполнения слияния этого типа.

Программа 11.3. является восходящей реализацией слияния Бэтчера без операции тасования, соответствующей сетям, представленным на рис. 11.5. Эта программа представляет собой компактную и элегантную обменную (не использующую дополнительного пространства оперативной памяти) функцию слияния, которую, возможно, легче понять, если считать ее чередующимся представлением сетей, в то же время прямое доказательство того, что она правильно завершает задачу слияния, само по себе очень интересно. Мы проведем исследование одного такого доказательства в конце этого раздела.



**РИСУНОК 11.5. НЕЧЕТНО-ЧЕТНОЕ СЛИЯНИЕ БЭТЧЕРА**

Показанные на этом рисунке различные представления сетей на четыре (сверху), восемь (в центре) и 16 (внизу) линий служат иллюстрацией рекурсивной структуры, положенной в основу сети. Слева показаны прямые представления конструкции сети размера  $N$  с двумя копиями сетей размера  $N/2$  (одна для линий с четной нумерацией, другая для линий с нечетной нумерацией) плюс каскад компараторов, соединенных с линиями 1 и 2, 3 и 4, 5 и 6 и т.д. Справа показаны более простые сети, которые были получены нами из сетей, изображенных слева, путем группирования компараторов одинаковой длины; такое группирование стало возможным в связи с тем, что мы можем перемещать компараторы, установленные на нечетных линиях, мимо компараторов на нечетных линиях, не нарушая их работы.

Рисунок 11.7 служит иллюстрацией сети нечетно-четной сортировки Бэтчера, построенной на основе сетей слияния, представленных на рис. 11.5, с использованием стандартной конструкции рекурсивной сортировки слиянием. Эта конструкция дважды рекурсивна: один раз для сетей слияния, другой раз — для сетей сортировки. И хотя они не оптимальны — мы вскоре рассмотрим оптимальные сети — тем не менее, они достаточно эффективны.

### Программа 11.3. Нечетно-четное слияние Бэтчера (нерекурсивная версия)

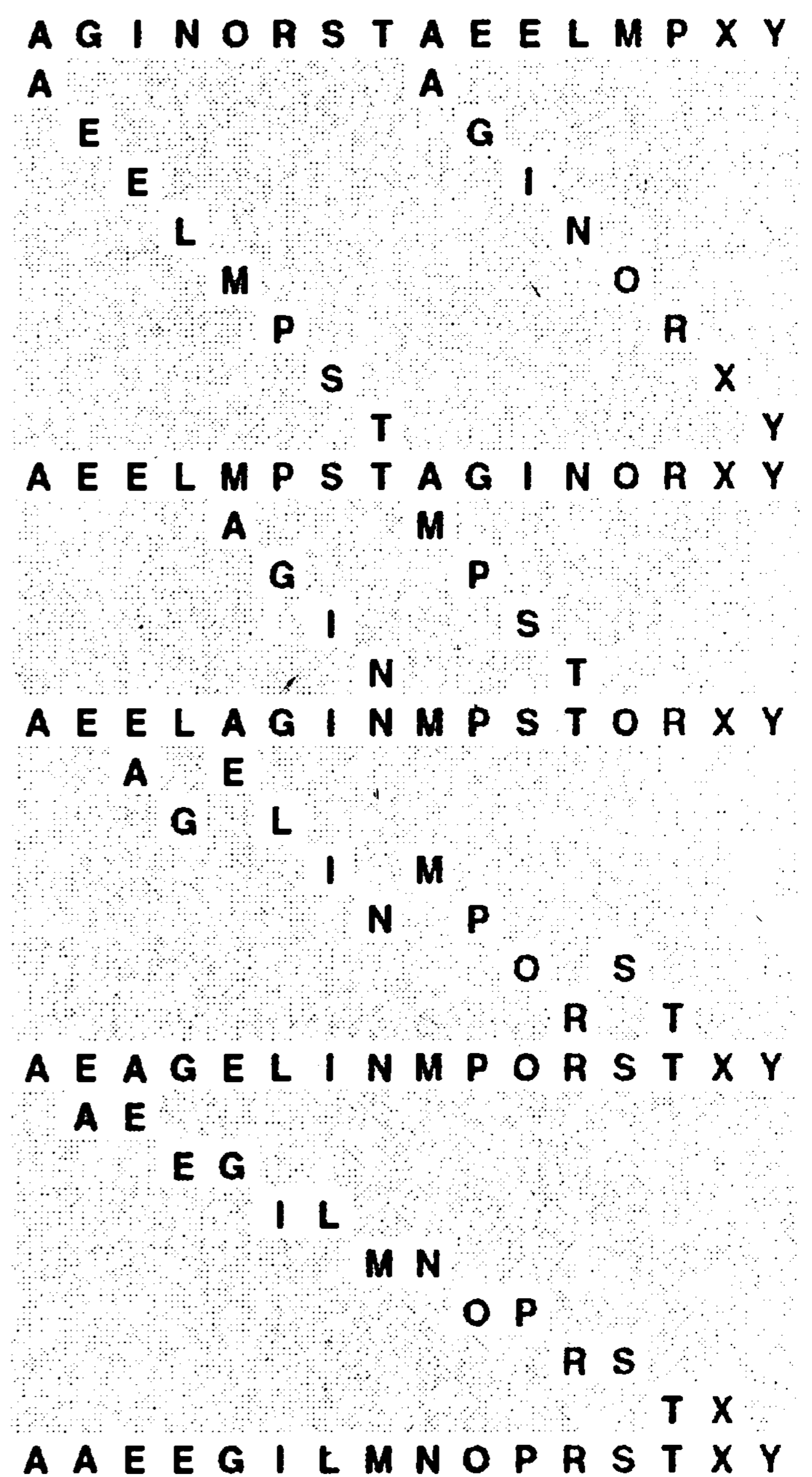
Данная реализация нечетно-четного слияния (которая предполагает, что размер файла  $N$  является степенью 2) компактна и своеобразна. Мы сможем понять как она совершает процедуру слияния, выполнив исследования, в какой степени она соответствует рекурсивной версии (см. программу 11.2 и рис. 11.5). Она завершает слияние за  $\lg N$  проходов, представленных единообразными и независимыми командами сравнения-обмена.

```
template <class Item>
void merge(Item a[], int l, int m, int r)
{ int N = r-l+1;    // предполагается, что
                  // N/2 это m-l+1
  for (int k = N/2; k > 0; k /= 2)
    for (int j = k % (N/2); j+k<N; j+= k+k)
      for (int i = 0; i < k; i++)
        compexch(a[l+j+i], a[l+j+i+k]);
}
```

**Лемма 11.3.** Сети нечетно-четной сортировки Бэтчера используют приблизительно  $N(\lg N)^2 / 4$  компараторов и могут быть выполнены за  $(\lg N)^2 / 2$  параллельных шагов.

Сеть слияния требует выполнения около  $\lg N$  параллельных шагов, а сети сортировки требуют выполнения  $1 + 2 + \dots + \lg N$  или примерно  $(\lg N)^2 / 2$  параллельных шагов. Подсчет компараторов оставляем читателю в качестве упражнения (см. упражнение 11.23).

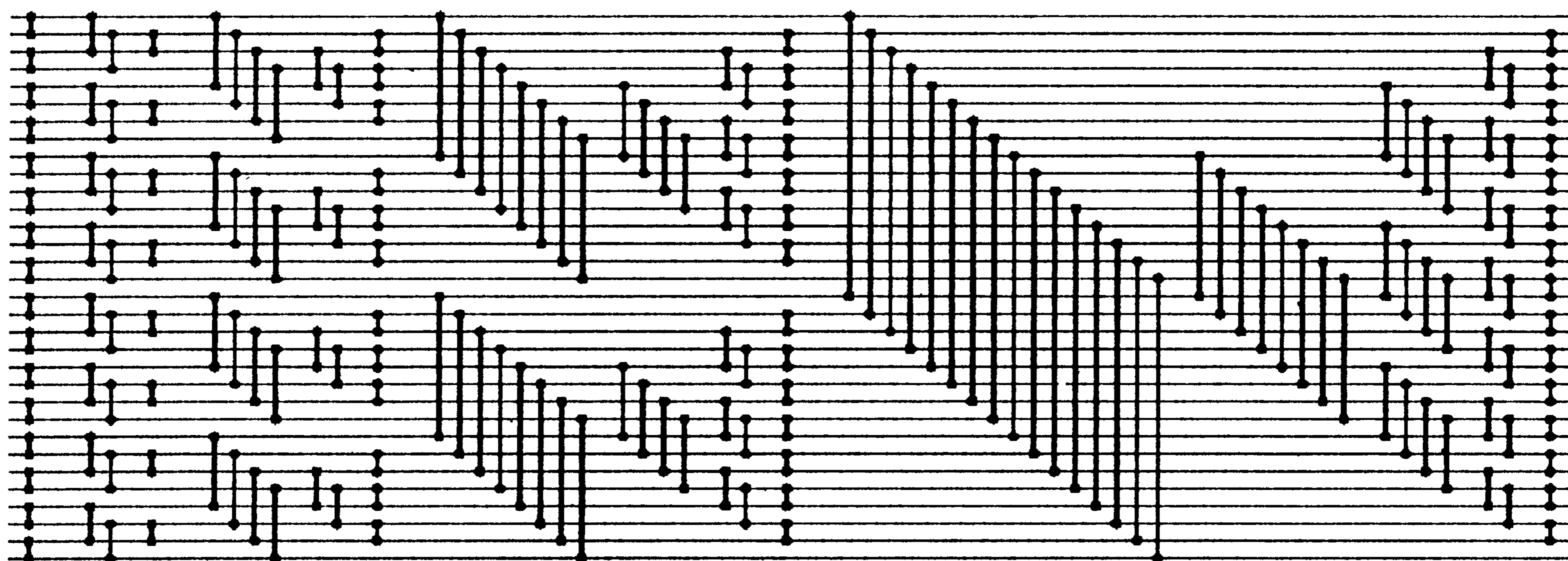
Использование функции слияния в программе 11.3 в рамках стандартной сортировки слиянием, представленной программой 8.3, позволяет получить компактный обменный неадаптивный метод сортировки, который использует  $O(N(\lg N)^2)$  операций сравнения-обмена. С другой стороны, мы можем удалить все рекурсии из сортировки слиянием и непосредственно реализовать восходящую версию в полном ее объеме, как показано в программе 11.4. Как и в случае программы 11.3, эту программу легче понять, если рассматривать ее как чередующееся представление сети, показанной на рис. 11.7.



**РИСУНОК 11.6. ПРИМЕР НЕЧЕТНО-ЧЕТНОГО СЛИЯНИЯ БЭТЧЕРА.**

Если удалить все операции тасования, то для выполнения слияния Бэтчера применительно к нашему примеру потребуется 25 операций слияния-обмена, изображенных на этой диаграмме. Их можно разбить на четыре фазы выполнения независимых операций сравнения-обмена с фиксированным сдвигом каждой фазы.





**РИСУНОК 11.7. НЕЧЕТНО-ЧЕТНЫЕ СЕТИ СОРТИРОВОК БЭТЧЕРА**

*Данная сеть сортировки на 32 линии содержит две копии сетей на 16 линий, четыре копии сетей на восемь линий и т.д. Просматривая справа налево, мы как бы проходим через всю структуру сверху вниз: сеть сортировки на 32 линии состоит из сети слияния типа 16-с-16, за которой следуют две копии сети сортировки на 16 линий (одна в верхней половине и одна в нижней половине). Каждая сеть на 16 линий состоит из сети слияния типа 8-с-8, за которой следуют две копии сети сортировки на 8 линий и т.д. Просматривая слева направо, мы проходим через эту структуру снизу вверх: первый столбец компараторов создает отсортированный файл размером 2; далее идут сети слияния типа 2-с-2, которые создают отсортированные подфайлы размером 4; затем идут сети слияния типа 4-с-4, которые создают отсортированные подфайлы размером 8 и т.д.*

Такая реализация предусматривает включение еще одного цикла и одной проверки в программу 11.3, поскольку слияние и сортировка обладают похожими рекурсивными структурами. Чтобы выполнить восходящий проход слияния последовательности отсортированных файлов размера  $2^k$  в последовательность отсортированных файлов размера  $2^{k+1}$ , мы используем всю сеть слияния, но при этом включаем только те компараторы, которые полностью попадают в подфайлы. Данная программа может претендовать на получение приза как наиболее компактная нетривиальная реализация сортировки, какую нам когда-либо приходилось видеть и которая может оказаться наилучшим выбором в тех случаях, когда мы хотим воспользоваться всеми преимуществами высокопроизводительных архитектурных особенностей для высокоскоростной сортировки небольших файлов (или для построения сетей сортировки). Понимание того, как и почему сортирует рассматриваемая программа, может оказаться неподъемной задачей, если мы лишимся перспективы использования рекурсивных приложений и сетевых конструкций, рассмотренных до сих пор.

Как это часто случается с методами типа "разделяй и властвуй", мы сталкиваемся с двумя возможностями для выбора в случае, когда  $N$  не есть степень 2 (см. упражнение 11.24 и 11.21). Мы можем поделить его напополам (сверху вниз) либо разделить на максимальное число, представляющее собой степень 2, меньшее  $N$  (снизу вверх). Последнее несколько удобнее для сетей сортировки, поскольку это эквивалентно построению полной сети для минимальной степени 2, большей или равной  $N$ , после чего использовать только первые  $N$  линий и компараторы, оба конца которых подключены именно к этим линиям. Предположим теперь, что на неиспользованных линиях имеются служебные ключи, которые больше любых других ключей сети. Тогда компараторы на этих линиях никогда не производят операций обмена, так что их

можно удалить без ущерба для дальнейших действий. В самом деле, мы можем воспользоваться любым смежным набором из  $N$  линий большей сети: будем считать, что на игнорируемых линиях в верхней части имеются сигнальные метки с небольшими значениями, а на игнорируемых линиях в нижней части имеются служебные метки с большими значениями. Во всех этих сетях имеется примерно  $N (\lg N)^2 / 4$  компараторов.

#### Программа 11.4. Нечетно-четная сортировка Бэтчера (нерекурсивная версия)

Данная реализация нечетно-четной сортировки Бэтчера прямо соответствует представлению сети, отображенному на рис. 11.7. Она разбивается на фазы, индексруемые переменной  $p$ . Последняя фаза, когда  $p$  равно  $N$ , и есть нечетно-четное слияние Бэтчера. Предпоследняя фаза, когда  $p$  равно  $N/2$ , есть нечетно-четное слияние с первым каскадом, и все компараторы, которые пересекаются с любой линией, кратной  $N/2$ , удаляются; третья фаза с конца, когда  $p$  равно  $N/4$ , есть нечетно-четное слияние с двумя первыми каскадами, а все компараторы, которые пересекаются с любой линией, кратной  $N/4$ , удаляются, и так далее.

```
template <class Item>
void batchersort(Item a[], int l, int r)
{ int N = r-l+1;
  for (int p = 1; p < N; p += p)
    for (int k = p; k > 0; k /= 2)
      for (int j = k&p; j+k < N; j += (k+k))
        for (int i = 0; i < N-j-k; i++)
          if ((j+i)/(p+p) == (j+i+k)/(p+p))
            comrch(a[l+j+i], a[l+j+i+k]);
}
```

Теория сетей сортировки развивалась достаточно интересно (см. раздел ссылок). Задача построения сети с минимально возможным числом компараторов была поставлена Бозе (Bose) еще до 1960 г., впоследствии она получила название задачи *Бозе-Нельсона (Bose-Nelson)*. Сети Бэтчера были первым более-менее приемлемым решением этой задачи, а некоторые исследователи даже полагали, что сети Бэтчера оптимальны. Сети слияния Бэтчера оптимальны, так что любая сеть сортировки с существенно меньшим числом компараторов может быть построена только в рамках подхода, отличного от рекурсивной сортировки слиянием. Задача нахождения оптимальных сетей сортировки не подвергалась исследованиям до тех пор, пока в 1983 г. Аджтай (Ajtai), Колмос (Kolmos) и Шемереди (Szemeredy) не доказали существования сетей с  $O(N \log N)$  числом компараторов. Однако сети AKS (Ajtai, Kolmos и Szemeredy) — это всего лишь математические умозаключения, не имеющие практического применения, так что сети Бэтчера все еще считаются одними из наиболее подходящих для практического применения.

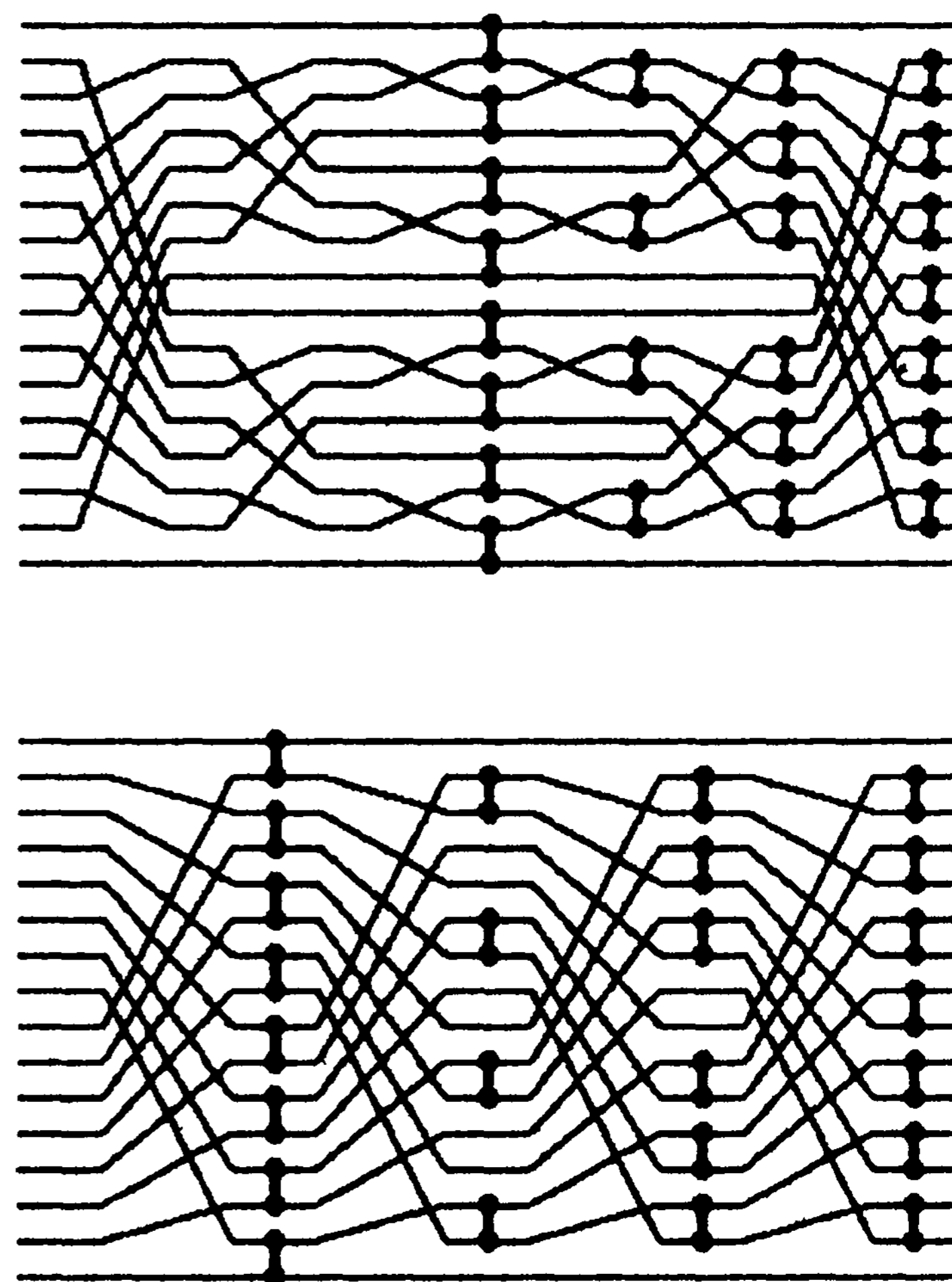


РИСУНОК 11.8.

Прямая реализация программы 11.2 как сети сортировки порождает сеть, насыщенную рекурсивными операциями тасования и обратного тасования (вверху). Эквивалентная реализация (внизу) использует только операции полного тасования.

Связь между идеальным тасованием и сетями Бэтчера позволяет завершить наши исследования сетей сортировки анализом одной забавной версии рассматриваемого алгоритма. Если мы перетасуем линии нечетно-четного слияния Бэтчера, то получим сети, в которых компараторы соединяют смежные линии. Рисунок 11.8 служит иллюстрацией сети, которая соответствует реализации тасования, соответствующего программе 11.2. Эту схему межкомпонентных соединений иногда называют *сачком для ловли бабочек* (*butterfly network*). На этом рисунке дано еще одно представление той же неветвящейся программы, которая предлагает еще более унифицированный шаблон: она использует только операции полного тасования.

На рис. 11.9. показана еще одна интерпретация рассматриваемого метода, которая служит иллюстрацией базовой структуры. Во-первых, мы записываем один файл под другим, далее, мы сравниваем смежные по вертикали элементы и при необходимости выполняем их обмен так, чтобы меньший элемент находился выше большего. Затем мы делим каждый ряд на две равных части и чередуем половины этих рядов, после чего выполняем те же операции сравнения-обмена над числами во второй и третьей строках. В сравнении других пар рядов нет необходимости, поскольку они были предварительно отсортированы. Операция деления и чередования оставляет как ряды, так и столбцы в отсортированном виде. Это свойство в общем виде сохраняется благодаря той же операции: каждый шаг удваивает число рядов, сокращает наполовину число столбцов и сохраняет ряды и столбцы в отсортированном виде, в конечном итоге мы получаем один столбец и  $N$  рядов, благодаря чему столбец полностью отсортирован. Связь между таблицей, представленной на рис. 11.9 и сетью, изображенной в нижней части рис. 11.8, заключается в том, что когда мы разворачиваем таблицы по столбцам (за элементами первого столбца следуют элементы второго столбца и т.д.) мы замечаем, что перестановка, необходимая для перехода с одного шага на другой, есть ни что иное как идеальное тасование.

Теперь с помощью абстрактной параллельной машины со встроенными межкомпонентными соединениями для идеальной сортировки, как показано на рис. 11.10, мы имеем возможность непосредственно

A	E	G	G	I	M	N	R	A	B	E	E	L	M	P	X
A	E	G	G	I	M	N	R	A	B	E	E	I	M	N	R
A	B	E	E	L	M	P	X	A	E	G	G	L	M	P	X
A	B	E	E	I	M	N	R	A	B	E	E	L	M	P	X
I	M	N	R	A	E	G	G	I	M	N	R	L	M	P	X
A	E	G	G	I	M	N	R	L	M	P	X	L	M	P	X
L	M	P	X	L	M	P	X	L	M	P	X	L	M	P	X
A	B	E	E	I	M	N	R	L	M	P	X	L	M	P	X
A	B	E	E	I	M	N	R	L	M	P	X	L	M	P	X
A	E	G	G	I	M	N	R	L	M	P	X	L	M	P	X
G	G	I	M	N	R	L	M	P	X	G	G	I	M	N	R
G	G	I	M	N	R	L	M	P	X	G	G	I	M	N	R
I	M	N	R	L	M	P	X	I	M	N	R	L	M	P	X
I	M	N	R	L	M	P	X	I	M	N	R	L	M	P	X
L	M	N	R	L	M	P	X	L	M	N	R	L	M	P	X
L	M	N	R	L	M	P	X	L	M	N	R	L	M	P	X
P	X	P	X	P	X	P	X	P	X	P	X	P	X	P	X
P	X	P	X	P	X	P	X	P	X	P	X	P	X	P	X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

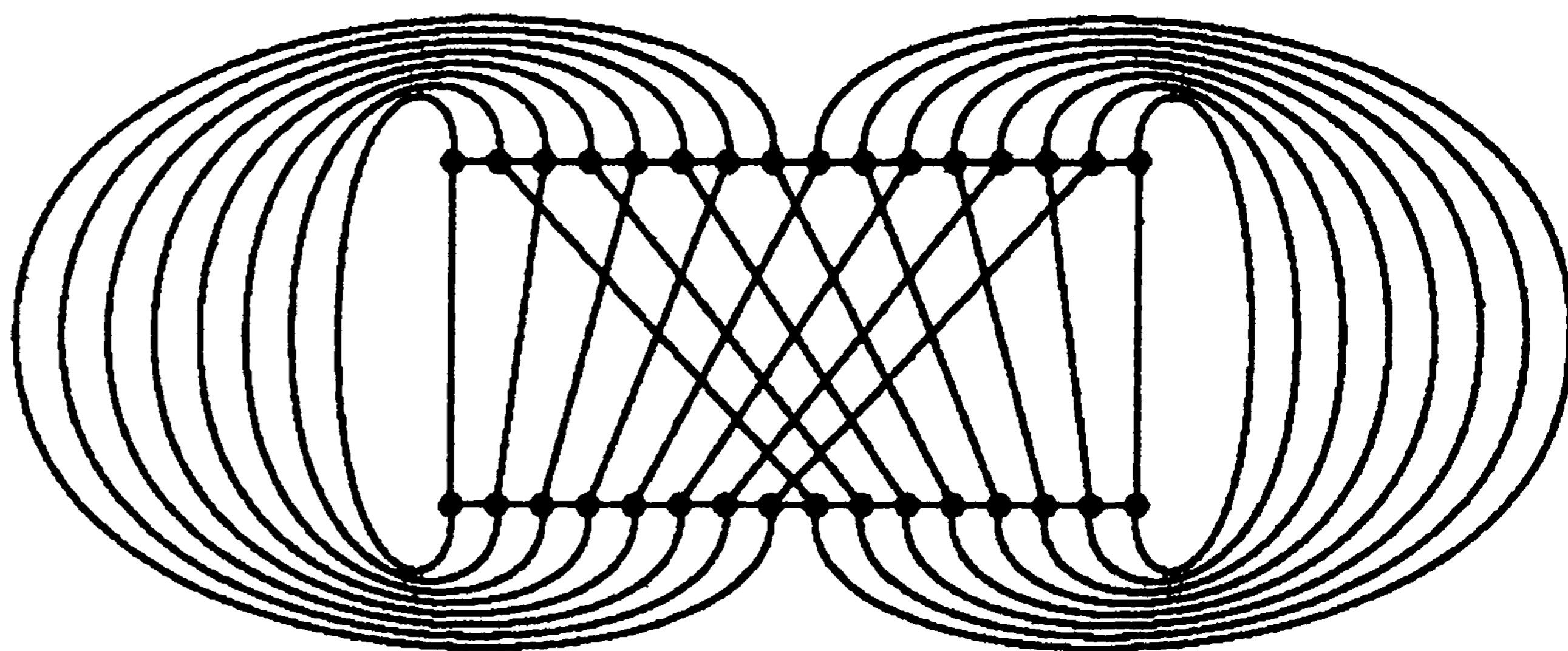
#### РИСУНОК 11.9. СЛИЯНИЕ МЕТОДОМ ДЕЛЕНИЯ И ЧЕРЕДОВАНИЯ.

*Записав вначале оба отсортированных файла в один ряд, мы выполняем их слияние посредством многократного повторения следующей процедуры: разбиваем каждый ряд на две равные части и чередуем полученные половины (слева), после чего выполняем операции сравнения-обмена над смежными по вертикали элементами, содержащимися в различных рядах (справа).*

*Сначала у нас было 16 столбцов и 1 ряд, затем восемь столбцов и 2 ряда, далее 4 столбца и четыре ряда, потом 2 столбца и восемь рядов и, наконец, 16 рядов и один столбец, который предстает в отсортированном виде.*

## РИСУНОК 10. 10. МАШИНА ИДЕАЛЬНОГО ТАСОВАНИЯ

Изображенная на этом рисунке машина с межкомпонентными связями способна эффективно выполнять алгоритм Бэтчера (равно как и множество других). Подобные связи используются в некоторых параллельных компьютерах.



реализовать сеть, подобную показанной в нижней части рис. 11.8. Такая машина на каждом шаге выполняет предусмотренные рассматриваемым алгоритмом операции сравнения-обмена на некоторой паре смежных процессоров, после чего осуществляет идеальное тасование данных. Программирование работы этой машины сводится к определению, какие пары процессоров должны выполнять операции сравнения-обмена на каждом цикле.

На рис. 11.11. показаны динамические характеристики как восходящего метода, так и версии нечетно-четного слияния Бэтчера с полным тасованием.

Тасование — это важная абстракция, описывающая движение данных в алгоритмах, построенных по принципу "разделяй и властвуй", и она возникает в различных задачах, отличных от сортировки. Например, квадратная матрица размером  $2^n$ -на- $2^n$  хранится в развернутом по рядам виде, затем  $n$  идеальных тасовок транспонируют эту матрицу (приводят матрицу к развернутому по столбцам виду). В число более важных примеров входят быстрые преобразования Фурье и полиномиальное приближение (см часть 8). Мы можем решить каждую из этих задач, воспользовавшись циклической машиной идеального тасования, подобной показанной на рис. 11.10, но с гораздо более мощными процессорами. Мы можем даже обдумать вариант с использованием универсальных процессоров, способных выполнять прямое и обратное тасование (некоторые из машин этого типа были даже построены для практического применения); мы вернемся к обсуждению параллельных машин данного типа в разделе 11.5.

## Упражнения

- 11.16. Дать примеры сетей сортировки четырех (см. упражнение 11.6), пяти и шести элементов. Использовать минимально возможное число компараторов.
- 11.17. Написать программу, способную подсчитать число параллельных шагов, необходимых для выполнения любой неветвящейся программы. *Совет:* Воспользуйтесь следующей стратегией присвоения меток. Отметьте входные линии как принадлежащие к каскаду 0, затем для каждого компаратора выполните следующие действия: пометьте обе выходные линии как входные для каскада  $i + 1$ , если метка одной из входных линий есть  $i$ , а метка другой линии не превосходит  $i$ .
- 11.18. Сравнить время выполнения программы 11.4 с аналогичным показателем для программы 8.3 для случайно упорядоченных ключей при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- ▷ 11.19. Начертить сеть Бэтчера, ориентированную на слияние типа 10-с-11.

●● 11.20. Доказать существование зависимости между рекурсивным обратным тасованием и тасованием, представленном на рис. 11.8.

○ 11.21. Из изложенного в тексте следует, что на рис. 11.7 в неявном виде представлено 11 сетей сортировки 21 элемента. Начертить ту из них, которая содержит минимальное число компараторов.

11.22. Найти число компараторов в нечетно-четных сетях сортировки Бэтчера для  $2 \leq N \leq 32$ , где сети, когда  $N$  не есть степень 2, строятся на базе первых  $N$  линий сети, построенной для наименьшей степени 2, превосходящей  $N$ .

○ 11.23. Для  $N = 2^n$  вывести точное выражение для определения числа компараторов, используемых в нечетно-четных сетях сортировки Бэтчера. *Совет:* Проверьте свой ответ на данных рис. 11.7, которые показывают, что в сетях имеются 1, 3, 9, 25 и 65 компараторов для  $N$  равному, соответственно, 2, 4, 8, 16 и 32.

○ 11.24. Построить сеть для сортировки 21 элемента, выбрав для этой цели нисходящий рекурсивный стиль, когда сеть размером  $N$  представляет собой композицию сетей размером  $\lfloor N/2 \rfloor$  и  $\lceil N/2 \rceil$ , за которыми следует сеть слияния. (Воспользоваться решением упражнения 11.19 в качестве завершающей части сети.)

11.25. Воспользоваться рекуррентными соотношениями для подсчета числа компараторов в сетях сортировки, построенных, как описано в упражнении 11.24 для  $2 \leq N \leq 32$ . Сравните полученные вами результаты с результатами, полученными в упражнении 11.22.

● 11.26. Найти 16-линейную сеть сортировки, которая использует меньшее число компараторов, чем сеть Бэтчера.

11.27. Вычертить сети слияния, соответствующие рис. 11.8, для битонных последовательностей, используя схему, описанную в упражнении 11.14.

11.28. Вычертить сети сортировки, соответствующие сортировке Шелла с приращениями Пратта (см. раздел 6.6) для  $N = 32$ .

11.29. Построить таблицу, содержащую число компараторов в сетях, описанных в упражнении 11.28, и число компараторов в сетях Бэтчера для  $N = 16, 32, 64, 128$  и 256.

11.30. Разработать сети сортировки, способные выполнять сортировку 3-сортированных или 4-сортированных файлов из  $N$  элементов.

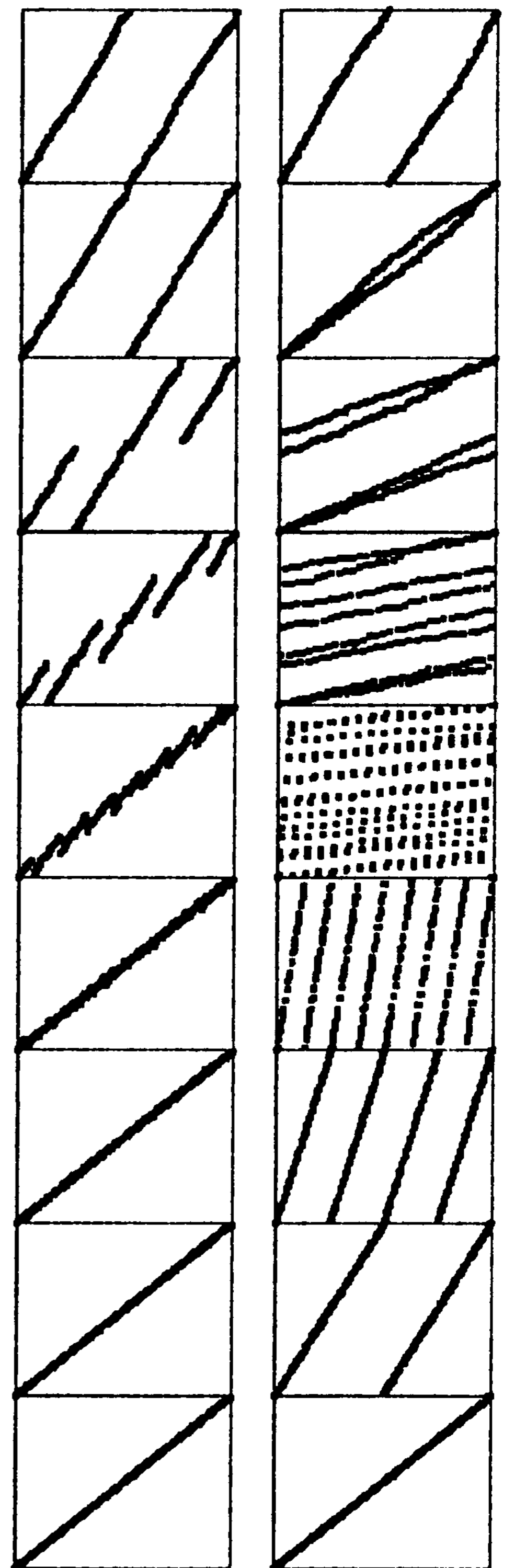


РИСУНОК 11.11.  
ДИНАМИЧЕСКИЕ  
ХАРАКТЕРИСТИКИ  
НЕЧЕТНО-ЧЕТНОГО  
СЛИЯНИЯ.

*Восходящая версия нечетно-четного слияния (слева) использует последовательность каскадов, посредством которых выполняется операция сравнения-обмена большей половины одного отсортированного файла с меньшей половиной следующего. При добавлении полного тасования (справа) рассматриваемый алгоритм приобретает совершенно другой вид.*

- 11.31. Воспользуйтесь сетью из упражнения 11.30 для разработки схемы, подобной алгоритму Пратта, на базе чисел, кратных 3 и 4. Вычертите полученную сеть для  $N = 32$  и решите упражнение 11.29 применительно к этой сети.
- 11.32. Начертить версию нечетно-четной сети сортировки Бэтчера для  $N = 16$ , в условиях которой идеальное тасование выполняется между каскадами независимых компараторов, соединяющих смежные линии (четырьмя концевыми каскадами этой сети должны быть каскады из сети слияния в нижней части рис.11.8).
- 11.33. Написать программу слияния для машины, изображенной на рис. 11.10, соблюдая следующие соглашения: каждая инструкция есть последовательность из 15 бит, при этом  $i$ -й бит,  $1 \leq i \leq 15$ , показывает (если он равен 1), что процессор  $i$  и процессор  $i - 1$  должны выполнить операцию сравнения-обмена. Программа представляет собой последовательность инструкций, а машина выполняет идеальное тасование между каждыми двумя инструкциями.
- 11.34. Написать программу сортировки для машины, изображенной на рис. 11.10, соблюдая соглашения, сформулированные в упражнении 11.33.

## 11.3. Внешняя сортировка

Мы переходим к рассмотрению другого аспекта задачи абстрактной сортировки, которая возникает, когда сортируемый файл настолько велик, что не помещается целиком в оперативной памяти компьютера. Для описания такого рода ситуаций мы используем термин *внешняя сортировка* (*external sorting*). Существует множество различных типов устройств внешней сортировки, которые накладывают различные ограничения на атомарные операции, применяемые при реализации таких видов сортировки. Кроме того, будет полезно изучить методы сортировки, использующие две простейшие базовые операции: операция *считывания* (*read*) данных из внешнего запоминающего устройства в оперативную память и операция *записи* (*write*) данных из оперативной памяти на внешнее запоминающее устройство. Мы полагаем, что стоимость этих двух операций настолько выше стоимости простейших примитивных вычислительных операций, что последние в дальнейшем мы можем полностью игнорировать. Например, в условиях такой абстрактной модели мы можем позволить себе не учитывать стоимость сортировки в оперативной памяти! При огромных размерах оперативной памяти и неэффективных методах сортировки подобный подход может оказаться неоправданным, но в общем случае на практике при необходимости всегда можно учесть эти затраты при расчете общей стоимости внешней сортировки.

Многообразие типов внешних запоминающих устройств и их стоимость ставят разработку методов внешней сортировки в зависимость от состояния технологии на текущий момент. Эти методы могут оказаться очень сложными, многие параметры оказывают влияние на их рабочие характеристики; вполне может случиться так, что искусные методы могут оказаться недооцененными и невостребованными только в силу тех или иных изменений в технологии. По этой причине в данном разделе мы не будем заниматься вопросами разработки конкретных реализаций, а сосредоточим свои усилия на изучении общих принципов внешней сортировки.

Довольно часто жесткие требования, предъявляемые к методам доступа к данным в зависимости от типа внешних устройств, оказываются более важным фактором, чем

высокая стоимость операций чтения-записи. Например, для большинства типов устройств операции чтения и записи из внешнего запоминающего устройства в оперативную память в общем случае наиболее эффективно выполняется в виде крупных блоков данных. Помимо этого, внешние устройства, обладающие громадными возможностями, довольно часто разрабатываются с таким расчетом, что максимальной производительности они достигают в тех случаях, когда доступ к таким блокам данных осуществляется *последовательно*. Например, мы не можем прочитать элемент данных, хранящийся в конце магнитной ленты без того, чтобы не просмотреть элементы, хранящиеся в начале ленты — на практике доступ к элементам данных на магнитной ленте ограничивается доступом к тем элементам, которые расположены в некоторой окрестности элементов, доступ к которым осуществляется чаще, чем к остальным. Некоторые из современных технологий обладают тем же свойством. В связи с этим, в данном разделе мы сосредоточимся на изучении методов, которые последовательно считывают и записывают крупные блоки данных, откуда непосредственно следует, что быстродействующие реализации этого вида доступа к данным могут быть достигнуты на машинах и устройствах тех типов, которые представляют для нас интерес.

Когда мы выполняем процесс считывания или записи некоторого числа различных файлов, мы полагаем, что эти файлы находятся на различных внешних запоминающих устройствах. На старых вычислительных машинах, в которых файлы хранились на сменных магнитных лентах, это предположение было непреложным требованием. При работе с магнитными дисками становится возможной реализация алгоритмов, которые, согласно замыслу, используют единственное внешнее устройство, но в общем случае работа с несколькими устройствами остается более эффективной.

Первым шагом при создании высокоэффективной программы сортировки файлов сверхбольших размеров является создание копии файла. Вторым шагом может быть представление исходного файла в обратном порядке. Все трудности, с которыми приходится сталкиваться при решении этих задач, возникают и при реализации внешней сортировки. (В процессе сортировки иногда приходится выполнять одну из этих операций.) Цель использования абстрактной модели состоит в том, чтобы отделить проблемы построения программной реализации от проблем разработки алгоритма.

Рассматриваемые алгоритмы сортировки представляют собой некоторую последовательность проходов по всем данным, и мы обычно определяем стоимость того или иного метода внешней сортировки путем простого подсчета числа таких проходов. В общем случае нам требуется сравнительно небольшое число проходов — десять или, возможно, меньше. В силу этого обстоятельства уменьшение этого числа проходов хотя бы на один существенно улучшает рабочие характеристики алгоритма. Основное наше предположение заключается в том, что мы полагаем, что основную долю времени выполнения конкретного метода внешней сортировки составляют операции ввода и вывода данных, следовательно, мы можем вычислить время выполнения внешней сортировки, умножив число осуществляемых ею проходов на время, необходимое для считывания и записи всего файла.

Короче говоря, абстрактная модель сортировки, которой мы в дальнейшем будем пользоваться, построена на предположении, что сортируемый файл слишком велик,

чтобы полностью поместиться в оперативную память, и что он определяет значения двух других параметров: времени выполнения сортировки (число проходов по данным) и количество используемых внешних устройств. Мы полагаем, что в нашем распоряжении имеются

- $N$  записей на внешнем устройстве, сортировку которых мы должны выполнить
- пространство оперативной памяти, достаточное для размещения  $M$  записей
- $2P$  внешних устройств, которыми мы можем пользоваться во время сортировки.

Мы присваиваем метку 0 внешнему устройству, на котором находится файл входных данных, и метки 1, 2, ...,  $2P - 1$  всем остальным внешним устройствам. Цель сортировки заключается в том, чтобы вернуть записи на устройство 0 в отсортированном виде. Как мы увидим далее, существует некоторая зависимость между  $P$  и общим временем выполнения сортировки — мы заинтересованы в том, чтобы получить эту зависимость в числовом выражении с тем, чтобы иметь возможность сравнить конкурирующие стратегии.

Существует несколько причин, в силу которых эта идеализированная модель может не соответствовать действительности. Тем не менее, как и всякая хорошая абстрактная модель, она отображает наиболее важные аспекты реальной ситуации, и это обстоятельство очерчивает точные рамки, внутри которых можно проводить исследования алгоритмических идей, многие из которых применяются непосредственно в практических ситуациях.

Большая часть методов внешней сортировки соблюдает следующие принципы общего характера. Выполняется первый проход по сортируемому файлу, в процессе которого производится его разбиение на блоки, размер которых примерно соответствует пространству оперативной памяти, после чего выполняется *сортировка* этих блоков. Затем осуществляется *слияние* отсортированных блоков, при необходимости с этой целью выполняются несколько проходов файла, при этом с каждым проходом степень упорядоченности возрастает, пока весь файл не окажется отсортированным. Такой подход называется *сортировкой-слиянием* (*sort-merge*), он с успехом применяется на практике с тех пор, когда компьютеры получили широкое распространение в коммерческих приложениях в пятидесятых годах прошлого столетия.

Простейшая стратегия сортировки-слияния, получившая название *сбалансированного многопутевого слияния* (*balanced multiway merging*), показана на рис. 11.12. Этот метод состоит из прохода, осуществляющего *начальное распределение* (*initial distribution*), за которым следуют несколько проходов многопутевого слияния (*multiway merging passes*).

На начальном проходе мы осуществляем распределение входных данных (входа) по внешним устройствам  $P, P + 1, \dots, 2P - 1$  в виде отсортированных блоков данных, каждый из которых содержит  $M$  записей (за исключением, возможно, заключительного блока, который меньше остальных, если  $N$  не кратно  $M$ ). Такое распределение нетрудно выполнить — мы считываем первые  $M$  записей с устройства ввода, сортируем их и записываем полученный блок данных на устройство  $P$ ; затем считываем следующие  $M$  записей с входного устройства, сортируем их и записываем отсортированный блок на устройство  $P + 1$  и т.д. Если, достигнув устройства  $2P - 1$ , у нас все еще остаются необработанные данные (т.е., если  $N > PM$ ), мы помещаем на устрой-



A S O R T I N G A N D M E R G I N G E X A M P L E W I T H F O R T Y F I V E R E C O R D S • S

A O S • D M N • A E X • F H T • E R V • S

I R T • E G R • L M P • O R T • C E O • S

A G N • G I N • E I W • F I Y • D R S • S

A A G I N O R S T • F F H I O R T T Y • S

D E G G I M N N R • C D E E O R R S V • S

A E E I L M P W X • S

A A A D E E E G G G I I I L M M N N N O P R R S T W X • S

C D E E F F H I O O R R R S T T V Y • S

• S

A A A C D D E E E E E F F G G G H I I I I L M M N N N O O O P R R R R R S S T T T V W X Y • S

## РИСУНОК 11.12.

*На проходе начального распределения берем элементы  $A S O$  из набора входных данных, сортируем их и помещаем отсортированную совокупность  $A O S$  на первое устройство для выходных данных. Далее мы берем элементы  $R T I$  из набора входных данных, сортируем их и помещаем отсортированную совокупность  $I R T$  на второе устройство для выходных данных. Продолжая таким образом, производя соответствующие операции на выходных устройствах в цикле, мы в конечном итоге получаем 15 отрезков: по пять на каждом выходном устройстве. На первой стадии слияния мы осуществляем слияние отрезков  $A O S$ ,  $I R T$  и  $A G N$ , в результате чего получаем последовательность  $A A G I N O R S T$ , которую мы записываем на первое выходное устройство, затем мы выполняем слияние вторых отрезков на входных устройствах и получаем последовательность  $D E G G I M N N R$ , которую мы записываем на втором выходном устройстве, и т.д.; таким способом мы выполняем сбалансированное распределение данных на три устройства. Сортировка завершается двумя дополнительными проходами, на которых выполняется слияние.*

ство  $P + 1$  второй отсортированный блок и т.д. до тех пор, пока весь ввод не будет исчерпан. По завершении процедуры распределения количество отсортированных блоков, размещенных на каждом устройстве, равно значению  $N/M$ , округленному до ближайшего целого числа в сторону уменьшения или увеличения. Если  $N$  есть кратное  $M$ , то размеры всех блоков одинаковы и равны  $N/M$  (если это не так, то все блоки, за исключением последнего, равны  $N/M$ ). Для небольших  $N$  число блоков может оказаться меньше  $P$ , ввиду чего одно или большее число устройств могут оказаться пустыми.

На первом многопутевом проходе, осуществляющем слияние, мы рассматриваем устройства в интервале от  $P$  до  $2P - 1$  как входные, а устройства в интервале от  $0$  до  $P - 1$  как выходные. Мы осуществляем  $P$ -путевое слияние блоков данных размером  $M$ , помещенных на входные устройства, в результате которого получаем отсортированные блоки данных размером  $PM$ , которые затем помещаем на выходные устройства с соблюдением условия максимально возможного баланса. Прежде всего, мы производим слияние первых блоков с каждого входного устройства и помещаем результат этого слияния на устройство  $0$ , затем мы помещаем результат слияния вторых блоков на каждом входном устройстве, на устройство  $1$  и т.д. По достижении устройства  $P - 1$  мы далее помещаем второй блок данных на устройство  $0$ , затем второй блок на устройство  $1$  и т.д. Мы продолжаем этот процесс до тех пор, пока все входные данные не будут исчерпаны. По завершении процедуры распределения число отсортированных блоков на каждом устройстве равно  $N/(PM)$ , округленное до

ближайшего целого числа в сторону увеличения или уменьшения. Если  $N$  есть кратное  $PM$ , то все блоки имеют размер  $PM$  (в противном случае заключительный блок имеет меньший размер). Если  $N$  не больше  $PM$ , то остается один отсортированный блок (на устройстве 0) и на этом сортировка заканчивается.

В противном случае мы повторяем этот процесс и выполняем второй проход многопутевого слияния, рассматривая устройства  $0, 1, \dots, P-1$  как входные устройства, а устройства  $P, P+1, \dots, 2P-1$  как выходные. Мы выполняем  $P$ -путевое слияние с целью получить из отсортированных блоков размера  $PM$ , посещенных на входных устройствах, блоки размером  $P^2M$  с последующим их размещением на выходных устройствах. Сортировка заканчивается по завершении второго прохода (результат находится на устройстве  $P$ ), если  $N$  не больше  $P^2M$ .

Продолжая таким образом, перемещаясь между устройствами от 0 до  $P$ , с одной стороны, и между устройствами от  $P-1$  до  $2P-1$ , с другой, мы увеличиваем размер блоков в  $P$  раз при помощи  $P$ -путевого слияния, пока в конечном итоге не получил один блок на устройстве 0 или на устройстве  $P$ . Завершающее слияние на каждом проходе может не быть  $P$ -путевым слиянием в полном смысле этого термина, но если это так, то оно хорошо сбалансировано.

Рисунок 11.13 служит иллюстрацией этого процесса, для чего используются только количество и относительные размеры проходов. Оценку стоимости слияния мы получаем, выполняя операции умножения, указанные в представленной на рисунке таблице, суммируя результаты (без учета нижней строки) и деля сумму на число отрезков исходного файла. Эти вычисления выражают издержки через число проходов по данным.

Чтобы выполнить  $P$ -путевое слияние, мы можем воспользоваться очередью по приоритетам размером  $P$ . Мы хотим постоянно иметь на выходе наименьший элемент из числа тех, которые еще не поступали на выход, из каждого из  $P$  отсортированных блоков, подлежащих слиянию, с последующей заменой каждого элемента на выходе следующим элементом из того же блока. Чтобы завершить эту процедуру, мы отслеживаем индексы устройств с помощью операции `operator<`, считывающей значение ключа следующей записи, которая должна быть считана с указанного устройства (и порождающей служебную метку, которая превосходит по значению все ключи записей по достижении конца блока). Само слияние представляет собой простой цикл, который считывает очередную запись с устройства, имеющего минимальный ключ, и

	0	1	2	3	4	5
15*1						
				5*1	5*1	5*1
2*3	2*3	1*3				
				1*9	1*6	
1*15						

### РИСУНОК 11.13. РАСПРЕДЕЛЕНИЕ ПРОХОДОВ В 3-ПУТЕВОМ СБАЛАНСИРОВАННОМ СЛИЯНИИ.

*В процессе начального распределения трехпутевого сбалансированной сортировки-слияния файла, размеры которого в 15 раз превосходят пространство оперативной памяти, мы размещаем 5 отрезков, размер которых в относительных единицах равен 1, на устройствах 4, 5 и 6, при этом устройства 0, 1 и 2 остаются незагруженными. На первой стадии слияния мы помещаем отрезки размера 3 на устройства 0 и 1 и один отрезок размера 3 на устройство 2, оставляя устройства 3, 4 и 5 незагруженными. Далее мы выполняем слияние отрезков, находящихся на устройствах 0, 1 и 2, и снова отправляем результаты на устройства 3, 4 и 5 и так далее, продолжая процесс до тех пор, пока останется только один отрезок на устройстве 0. Общее число обработанных таким образом записей равно 60: четыре прохода по 15 записям.*

передает эту запись на выход, после чего заменяет эту запись в очереди по приоритетам на следующую запись с того же устройства, продолжая такую последовательность действий до тех пор, пока служебная метка не станет наименьшей в очереди по приоритетам. Мы можем воспользоваться реализацией сортирующего дерева, чтобы сделать время выполнения для очереди по приоритетам пропорциональным  $\log P$ , но обычно  $P$  так мало, что эти затраты кажутся ничтожными по сравнению с операцией записи на внешнее устройство. В нашей абстрактной модели мы игнорируем затраты на содержание очереди по приоритетам и предполагаем, что в нашем распоряжении имеется эффективный метод последовательного доступа к данным на внешних устройствах, благодаря чему мы можем измерять время выполнения путем подсчета числа проходов по данным. На практике мы можем воспользоваться элементарной реализацией очереди по приоритетам и сосредоточить все наши усилия на создании программ, обеспечивающих максимально возможную производительность внешних устройств.

**Лемма 11.4.** *При наличии  $2P$  внешних устройств и оперативной памяти, достаточной для размещения  $M$  записей, сортировка-слияние, в основу которой положено  $P$ -путевое сбалансированное слияние, требует выполнения  $1 + \lceil \log_P(N/M) \rceil$  проходов.*

Один проход нужен для распределения. Если  $N = MP^k$ , то блоки получают размер  $MP$  после первого слияния,  $MP^2$  после второго,  $MP^3$  после третьего и т.д. Сортировка заканчивается после того, как будут выполнены  $k = \log_P(N/M)$  проходов. В противном случае, если имеет место условие  $M^{Pk-1} < N < M^{Pk}$ , эффект неполных и пустых блоков приводит к появлению различий в размерах блоков ближе к концу процесса, тем не менее, он завершится после выполнения  $k = \lceil \log_P(N/M) \rceil$  проходов.

Например, если мы хотим отсортировать 1 миллиард записей, имея для этой цели шесть внешних устройств и оперативную память, позволяющую разместить 1 миллион записей, мы можем решить эту задачу через трехпутевую сортировку-слияние, выполнив в общем целом восемь проходов по данным — один проход для начального распределения и  $\lceil \log_3 1000 \rceil = 7$  проходов слияния. После выполнения прохода начального распределения мы получим отсортированные блоки данных, содержащие по 1 миллиону записей, 3 миллиона записей в каждом блоке после первого слияния, 9 миллионов записей в каждом блоке после второго слияния, 27 миллионов записей в каждом блоке после третьего слияния и т.д. Можно подсчитать, что на сортировку файла уходит в 9 раз больше времени, чем на то, чтобы просто получить его копию.

Наиболее важное решение, которое приходится принимать на практике в процессе сортировки-слияния — это выбор значения  $P$ , т.е. порядка слияния. В условиях используемой нами абстрактной модели последовательный метод доступа накладывает на нас определенные ограничения, откуда следует, что  $P$  должно быть равно половине числа доступных для нас внешних устройств. Такая модель реалистична для многих внешних запоминающих устройств. Если всего лишь небольшое число внешних устройств может быть использовано для сортировки, неизбежным становится использование методов доступа, отличных от последовательных. В таких случаях мы все еще имеем возможность использовать многопутевое слияние, но при этом должны учитывать определенную зависимость, заключающуюся в том, что увеличение

значения  $P$  приводит к уменьшению числа проходов, но в то же время увеличивает количество операций (медленных) непоследовательного доступа.

## Упражнения

- ▷ 11.35. Показать, как ключи **E A S Y Q U E S T I O N W I T H P L E N T Y O F K E Y S** сортируются посредством 3-путевого сбалансированного слияния в стиле примера, представленного на рис. 11.12.
- ▷ 11.36. Как отразится на числе проходов многопутевого слияния тот факт, что число используемых для слияния внешних устройств удвоилось?
- ▷ 11.37. Как отразится на числе проходов многопутевого слияния тот факт, что мы увеличили в 10 раз объем доступного пространства оперативной памяти?
- 11.38. Разработать интерфейс для внешнего ввода и вывода, который предусматривает последовательную передачу блоков данных с внешних устройств, работающих асинхронно (или детально изучить существующий в вашей системе). Использовать этот интерфейс для реализации  $P$ -путевого слияния, при этом  $P$  есть максимально возможное в конкретных обстоятельствах, но  $P$  входных файлов и выходной файл должны размещаться на различных выходных устройствах. Сравните время выполнения полученной программы с временем, необходимым для копирования файлов на выходные устройства, один за другим.
- 11.39. Используйте интерфейс из упражнения 11.38 при написании программы замены порядка на обратный для файла максимально допустимого в вашей системе размера.
- 11.40. Как вы будете выполнять идеальное тасование всех записей на внешнем устройстве?
- 11.41. Разработать стоимостную модель многопутевого слияния, которая охватывает алгоритмы, способные переключаться с одного файла на другой на одном и том же устройстве с постоянной стоимостью переключения, которая намного превышает стоимость последовательного чтения.
- 11.42. Разработать подход к внешней сортировке, который основан на разделении, подобном используемому в быстрой сортировке или в поразрядной сортировке MSD (сначала по старшей цифре), проанализировать его и сравнить с многопутевой сортировкой. Вы можете перейти на более высокий уровень абстракции, использованный при описании сортировки-слияния в этом разделе, однако вы должны стремиться к тому, чтобы быть способными спрогнозировать время выполнения для заданного числа устройств и заданного объема оперативной памяти.
- 11.43 Как вы будете сортировать содержимое внешнего устройства, если любые другие внешние устройства недоступны для использования (можно пользоваться только оперативной памятью)?
- 11.44. Как вы будете сортировать содержимое внешнего устройства, если для использования доступно всего лишь еще одно устройство (а также оперативная память)?

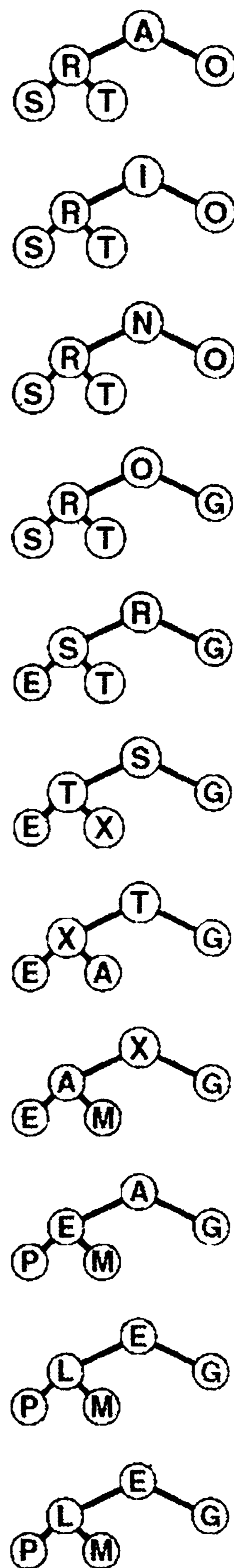
## 11.4. Различные реализации сортировки-слияния

Общая стратегия сортировки-слияния, описанная в разделе 11.13, доказала свою эффективность на практике. В этом разделе мы рассмотрим два усовершенствования этой стратегии, позволяющих снизить объем затрат. Первое из них, метод *выбора с замещением* (*replacement selection*), оказывает на время выполнения тот же эффект, что и объем пространства используемой оперативной памяти; следующий метод, *многофазное слияние* (*polyphase merging*), обеспечивает тот же эффект, что и увеличение числа используемых нами устройств.

В разделе 11.3 мы обсуждали применение очереди по приоритетам для *P*-путевого слияния, но при этом делали оговорку, что *P* настолько мало, что усовершенствование алгоритма с целью повышения быстродействия фактически незаметны. Тем не менее, во время начального распределения мы можем воспользоваться быстрыми очередями по приоритетам, чтобы получить отсортированные отрезки файлов, размеры которых не позволяют размещать их в оперативной памяти. Идея заключается в том, чтобы пропустить (неупорядоченные) входные данные через очередь по приоритетам больших размеров, как и раньше, записывая в очередь по приоритетам наименьший элемент, постоянно заменяя его следующим элементом со входа с одним дополнительным условием: если новый элемент меньше, чем только что появившийся на выходе, то поскольку он может и не стать частью текущего сортируемого блока, мы маркируем его как принадлежащий следующему блоку и считаем, что он больше всех остальных элементов текущего блока. Когда маркированный элемент поднимается в вершину очереди по приоритетам, мы начинаем новый блок. На рис. 11.14 этот метод показан в действии.

**Лемма 11.5.** *В случае произвольных ключей размеры блоков данных, полученных посредством выборки с замещением, в два раза превосходят размер сортирующего дерева.*

Если мы воспользуемся пирамидальной сортировкой для получения исходных блоков данных, мы заполним записями всю оперативную память; в этом случае нужно выводить их из памяти одну за одной до тех пор, пока сортирующее дерево не опустеет. После этого мы заполняем оперативную память очередным пакетом записей и продолжаем этот процесс снова и снова. В среднем сортирующее дерево занимает во время выполнения этого про-



**РИСУНОК 11.14. ВЫБОР С ЗАМЕЩЕНИЕМ**

Эта последовательность показывает, как можно получить два блока данных, *A I N O R S T X* и *A E E G L M P*, соответственно, длиной 7 и 8, из последовательности, используя для этой цели сортирующее дерево размером 5.

цесса только половину пространства оперативной памяти. В противоположность этому, выборка с замещением сохраняет эту же структуру в оперативной памяти, так что не следует удивляться тому, что ее эффективность в два раза выше. Полное доказательство этой леммы требует полного и разностороннего анализа (см. раздел ссылок), хотя это утверждение нетрудно проверить экспериментальным путем (см. упражнение 11.47).

Что касается файлов с произвольной организацией, то практический результат применения выборки с замещением, по-видимому, состоит в уменьшения числа проходов на 1: вместо того, чтобы начинать с отсортированных блоков данных, размер которых примерно равен объему пространства оперативной памяти, мы для начала можем использовать отрезки в два раза большего объема пространства оперативной памяти. Для  $P = 2$  такая стратегия экономит точно один проход слияния, для значений  $P$  больше 2 полученный эффект несколько скромнее. Тем не менее, мы знаем, что на практике сортировка файлов с произвольной организацией случается довольно редко, и если ключи упорядочены в той или иной степени, то в условиях использования метода выборки с замещением можно получать отрезки огромных размеров. Например, если ни одному из ключей не предшествуют в файле более  $M$  ключей, превосходящих его по значению, то этот файл может быть полностью отсортирован за один проход выборки с замещением, и при этом никакое слияние не понадобится! Эта возможность служит наиболее веским аргументом в пользу практического применения выборки с замещением.

Большим недостатком сбалансированной многопутевой сортировки является тот факт, что примерно только половина внешних устройств активно используется во время выполнения процедур слияния:  $P$  входных устройств и устройство, используемое для накопления выхода. Альтернативой этому остается выполнение  $(2P - 1)$ -путевых сортировок с передачей выхода на устройство 0, с последующим распределением данных на другие магнитные ленты в конце каждого прохода слияния. Но этот подход не превосходит первый по эффективности, поскольку он удваивает количество проходов, что обусловлено необходимостью распределения данных. Сбалансированное многопутевое слияние, по-видимому, потребует либо дополнительного числа лентопротяжных устройств, либо выполнения дополнительных операций копирования. Разработаны несколько хитроумных алгоритмов, которые обеспечивают занятость всех внешних устройств за счет замены устройства, на котором производится слияние отсортированных блоков данных небольших размеров. Простейший из этих методов получил название *многофазное слияние* (*polyphase merging*).

Основополагающая идея многофазного слияния заключается в том, чтобы распределять отсортированные блоки, полученные в результате выполнения выборки с замещением на доступные лентопротяжные устройства с определенной степенью неравномерности (оставляя одно устройство пустым) с последующим применением стратегии "сливать до опустошения" (*merge-until-empty*): поскольку сливаемые ленты неодинаковы по длине, одна из них будет исчерпана раньше остальных, после чего она может быть использована как выходная. То есть, мы меняем ролями выходную ленту (теперь на ней размещены несколько отсортированных блоков) и пустую к этому моменту входную ленту, продолжая этот процесс до тех пор, пока останется только один блок. На рис. 11.15 показан соответствующий пример.

A S O R T I N G A N D M E R G I N G E X A M P L E W I T H F O R T Y F I V E R E C O R D S • S

A O S • D M N • A E X • F H T • S

I R T • E G R • L M P • O R T • E R V • D R S • S

A G N • G I N • E I W • F I Y • C E O • • • S

A A G I N O R S T • D E G G I M N N R • A E E I L M P W X • F F H I O R T T Y • S

E R V • D R S • S

C E O • • • S

A A C E E G I N O O R R S T V • D D E G G I M N N R R S • S

A E E I L M P W X • F F H I O R T T Y • S

• S

D D E G G I M N N R R S • S

F F H I O R T T Y • S

A A A C E E E E G I I L M N O O P R R R S T V W X • S

A A A C D D E E E E E F F G G G H I I I I L M M N N N O O O P R R R R R R S S T T T V W X Y • S

### РИСУНОК 11.15. ПРИМЕР МНОГОФАЗНОГО СЛИЯНИЯ

*На стадии начального распределения мы размещаем различное число отрезков на лентах в соответствии с заранее определенной схемой, а не по принципу поддержания сбалансированного числа отрезков на лентах, как это имело место в случае на рис. 11.12. Затем мы выполняем трехпутевые слияния на каждой фазе, пока сортировка не будет завершена. В этом случае число фаз больше, чем в условиях сбалансированного слияния, но эти фазы не выполняются на всей совокупности данных.*

Стратегия "сливать до опустошения" работает на произвольном числе магнитных лент, как показано на рис. 11.16. Слияние разбивается на множество фаз, и не каждая из них выполняется на всей совокупности данных, зато ни на одной из них не надо выполнять дополнительного копирования. На рис. 11.16 показано, как производится вычисление отрезков для начального распределения. Мы подсчитываем число отрезков на каждом устройстве, рассуждая в обратном порядке.

В случае примера, представленного на рис. 11.16, мы ставим перед собой следующую задачу: мы хотим закончить слияние с 1 отрезком на устройстве 0. Следовательно, перед последним слиянием устройство 0 должно быть незагруженным, а на устройствах 1, 2 и 3 должны находиться по одному отрезку. Далее, мы определяем, каким должно быть распределение отрезков, которое потребуется для выполнения предпоследнего слияния, чтобы получить требуемое распределение. Одно из устройств 1, 2 или 3 должно быть пустым (чтобы его можно было использовать в качестве выходного устройства для предпоследнего слияния) — произвольно выбираем для этой цели устройство 3. Иначе говоря, предпоследнее слияние сливает по одному отрезку с каждого устройства 0, 1 и 2 и помещает результат на устройство 3. Поскольку предпоследнее слияние оставляет 0 отрезков на устройствах 0 и 1, оно должно начинаться с одного отрезка на устройстве 0 и двух отрезков на каждом из устройств 1 и 2. Аналогичные рассуждения приводят нас к заключению, что слияние, предшествующее только что рассмотренному, должно начинаться с того, что на устройствах 3, 0 и 1 должно быть, соответственно, 2, 3 и 4 отрезка. Продолжая в том же духе, мы можем построить таблицу распределения отрезков: выбираем максимальное число из каждого ряда, заменяем его нулем и добавляем его к каждому из оставшихся чисел, чтобы получить предыдущий ряд. Этот прием соответствует определению предыдущего ряда слияния высшего порядка, которое порождает текущий ряд. Такая техника работает с любым количеством магнитных лент (по меньшей мере, с тремя). Возни-

кающие при этом числа суть обобщенные числа Фибоначчи, которые обладают множеством интересных свойств. Если число отрезком не есть обобщенное число Фибоначчи, то мы предполагаем наличие фиктивных отрезков, которые необходимы для заполнения таблицы. Основная трудность в реализации многофазного слияния состоит в определении отрезков для начального распределения (см. упражнение 11.54).

Имея в своем распоряжении распределение отрезков, мы, применяя прямой ход рассуждений, можем вычислить их относительные размеры, фиксируя размеры после очередного слияния. Например, первое слияние в примере на рис. 11.16 порождает 4 отрезка размером в 3 относительных единицы на устройстве 0, два отрезка размером 1 на устройстве 2 и 1 отрезок размера 1 на устройстве 3 и т.д. Как и в случае сбалансированного многопутевого слияния, мы можем проделать указанные операции умножения, просуммировать результаты (не включая нижнюю строку) и поделить на число начальных отрезков, чтобы вычислить меру стоимости в виде числа, кратного стоимости полного прохода по всей совокупности данных. Для простоты вычислений мы включаем фиктивные отрезки в расчет затрат, который дает нам верхнюю границу истинной стоимости.

**Лемма 11.6.** При наличии трех внешних устройств и пространства оперативной памяти, достаточного для размещения  $M$  записей, сортировка-слияние, в основу которой положена операция выборки с замещением с последующим двухпутевым многофазным слиянием, выполняет в среднем  $1 + \lceil \log_{\phi}(N/M) \rceil / \phi$  эффективных проходов.

Общий анализ многофазной сортировки, выполненный Кнудом (Knuth) и другими исследователями в шестидесятых-семидесятых годах, — сложное и пространное исследование, выходящее за рамки данной книги. Для  $P = 3$  используются числа Фибоначчи, отсюда и появление коэффициента  $\phi$ . Другие константы появляются для  $P$  больше 3. Коэффициент  $1/\phi$  используется в случае, когда на каждой фазе используется именно эта часть данных. Мы подсчитываем число "эффективных проходов" как количество считанных данных, деленных на общее количество данных. Некоторые результаты общего анализа вызывают удивление. Например, оптимальный метод распределения фиктивных отрезков по магнитным лентам предусматривает использование дополнительных фаз и большего числа фиктивных отрезков, чем можно бы было предположить, поскольку некоторые отрезки используются в слияниях намного чаще других (см. раздел ссылок).

0	1	2	3
	17*1		
7*1		4*1	6*1
3*1	4*3		2*1
1*1	2*3	2*5	
	1*3	1*5	1*9
1*17			

### РИСУНОК 11.16. РАСПРЕДЕЛЕНИЕ ОТРЕЗКОВ ДЛЯ МНОГОФАЗНОГО ТРЕХПУТЕВОГО СЛИЯНИЯ

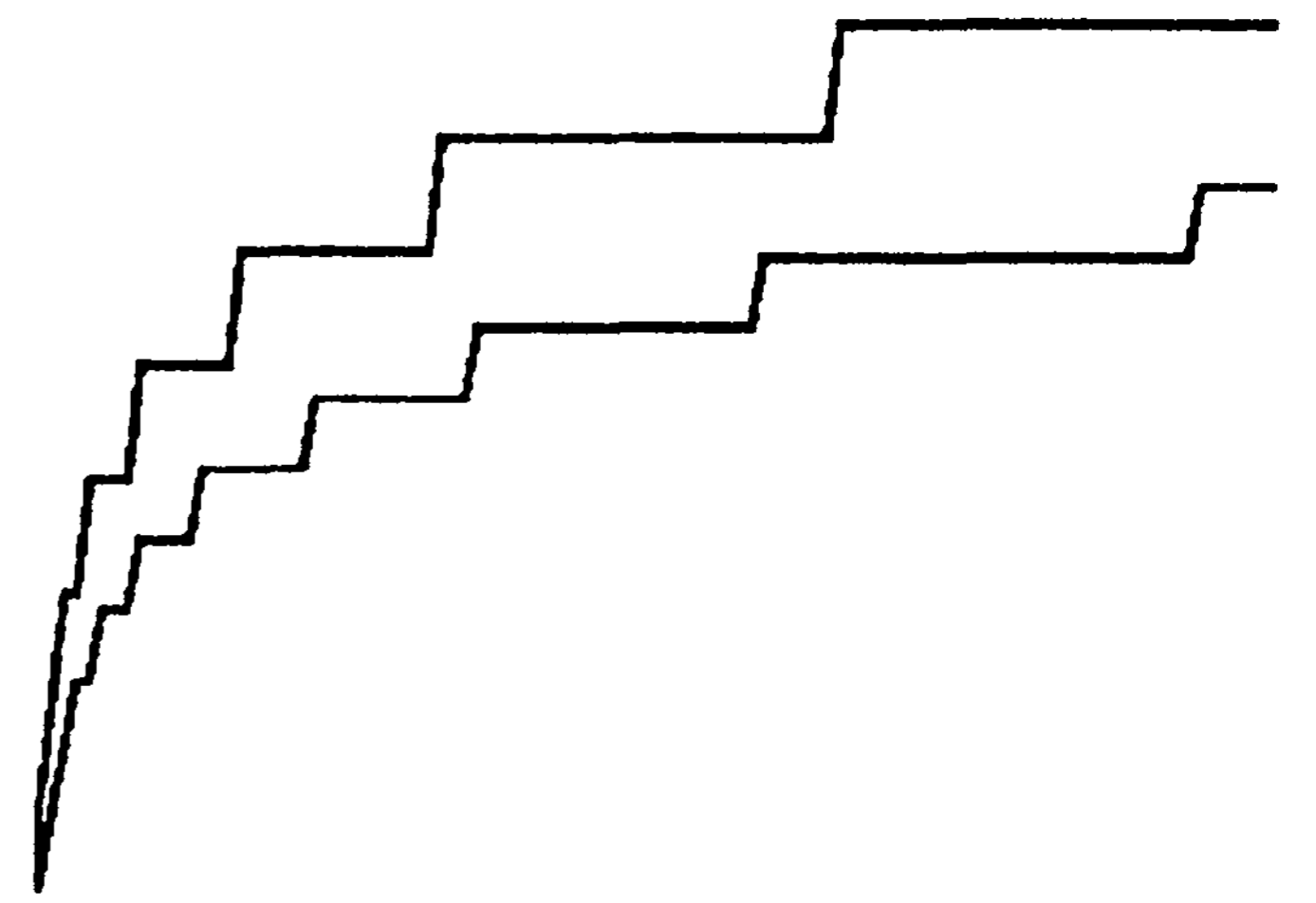
В процессе начального распределения многофазного трехпутевого слияния файла, размеры которого в 17 раз превосходят объем пространства оперативной памяти, мы помещаем семь отрезков на устройство 0, четыре отрезка на устройство 2 и шесть отрезков на устройство 3. Затем, на первой фазе мы выполняем слияния до тех пор, пока устройство 2 не станет пустым, при этом три отрезка размером 1 остаются на устройстве 0, два отрезка размером 1 на устройстве 3, и создаем четыре отрезка размером 3 на устройстве 1. Для файла, в 15 раз превышающего по размерам оперативную память, мы вначале помещаем два фиктивных отрезка на устройство 0 (см. рис. 11.15). Общее число блоков, подвергнутых обработке в процессе полного слияния, равно 59, на один меньше, чем в примере, иллюстрирующем сбалансированное слияние (см. рис. 11.13), но при этом мы используем на 2 устройства меньше (см. упражнение 11.50).



Например, если мы хотим отсортировать 1 миллиард записей, используя для этой цели 3 устройства и пространство оперативной памяти, достаточное для размещения 1 миллиона записей, мы можем сделать это посредством двухпутевого многофазного слияния с  $\lceil \log_{\phi} 500 \rceil / \phi = 8$  проходами. Добавив проход начального распределения, мы получаем несколько большие затраты (один проход), чем в случае сбалансированного слияния, которое использует в *два раза* большее число устройств. То есть, мы можем рассматривать многофазное слияние как процедуру, позволяющую выполнять ту же работу, но с половиной аппаратных средств. Для заданного количества устройств многофазное слияние всегда обеспечивает большую эффективность, чем сбалансированная сортировка, о чем свидетельствует рис. 11.17.

Как мы уже отмечали в начале раздела 11.3, тот факт, что мы сосредоточили свое внимание на изучении абстрактной машины с последовательным доступом к внешним устройствам, позволяет отделить проблемы, связанные с разработкой алгоритмов, от проблем их использования на практике. При разработке практических приложений нам необходимо проверить основные предположения и позаботиться о том, чтобы они всегда соблюдались. Например, мы зависим от эффективной реализации функций ввода-вывода, которые передают данные между процессором, внешними устройствами и системными программными средствами. В общем случае в современных системах используются хорошо отлаженные реализации таких программных средств.

Принимая эту крайнюю точку зрения, заметим, что многие из современных вычислительных систем располагают возможностью *виртуальной памяти* (*virtual memory*) — более абстрактная модель доступа к внешним устройствам, чем та, которая использовалась до сих пор. В виртуальной памяти мы получаем возможность обращаться к крупным блокам данных, содержащим большое количество записей, полагаясь на систему в вопросе гарантированной доставки адресуемых данных с внешних запоминающих устройств в оперативную память, когда нам это нужно; при этом возникает впечатление, что доступ к данным в смысле удобства ничем не отличается от прямого доступа к данным, находящимся в оперативной памяти. Однако эта иллюзия не совсем полная: пока программа обращается к ячейкам памяти, расположенных в относительной близости по отношению к ячейкам, к которым она недавно обращалась, то потребность в передаче данных с внешних устройств в оперативную память возникает нечасто и рабочие характеристики внешней памяти вполне удовлетворительны. (Например, программы, которые осуществляют последовательный доступ к дан-



#### РИСУНОК 11.17. СРАВНЕНИЕ ЗАТРАТ НА СБАЛАНСИРОВАННОЕ И МНОГОФАЗНОЕ СЛИЯНИЕ

*Число проходов, выполняемых в рамках сбалансированного слияния на четырех магнитных лентах (сверху), всегда больше, чем число эффективных проходов, выполняемых в рамках многофазного слияния с тремя магнитными лентами (внизу). Представленные на рисунке графики получены для функций, обладающих свойствами из лемм 11.4 и 11.6, для  $N/M$  от 1 до 100. В силу наличия фиктивных отрезков истинные характеристики многофазного слияния имеют более сложный характер, чем следует из представленной шаговой функции.*

ным, попадают в эту категорию.) Но если данные, к которым производится доступ, разбросаны в разных местах крупного файла, то система виртуальной памяти начнет испытывать *перезгруз (thrash)*, затрачивая все свое время на доступ к данным во внешней памяти, результаты которого будут катастрофическими.

Не следует игнорировать виртуальную память как возможную альтернативу сортировки-слияния очень крупных файлов. Мы могли бы получить программную реализацию сортировки-слияния непосредственно, или что еще проще, могли бы воспользоваться методами внутренней сортировки, такими как быстрая сортировка или сортировка слиянием. Эти методы внутренней сортировки заслуживают нашего пристального внимания в надежно работающей среде виртуальной памяти. Такие методы как пирамидальная сортировка или поразрядная сортировка в условиях, когда адреса ссылок находятся в разных концах оперативной памяти, по всей видимости, не подойдут из-за перезагрузки.

С другой стороны, использование виртуальной памяти может привести к недопустимым непроизводительным затратам системных ресурсов, так что расчет на собственные явные методы (подобные тем, которые рассматривались выше) может оказаться наилучшим способом получить максимум от высокопроизводительных внешних запоминающих устройств. Одна из основных характеристик изученных выше методов заключается в том, что они разрабатывались с расчетом, чтобы максимально возможное число независимых частей компьютерной системы работало с максимальной эффективностью и ни одна из частей при этом не простаивала. Когда в качестве подобных независимых частей рассматривать процессоры, мы приходим к идее параллельных вычислений, которая рассматривается в разделе 11.5.

## Упражнения

- ▷ **11.45.** Показать, какие отрезки порождаются методом выборки с замещением с очередью по приоритетам размером 4, примененным к ключам **E A S Y Q O U E S T I O N**.
- **11.46.** Как отражается использование метода выборки с замещением на файле, который был порожден посредством применения метода выборки с замещением к заданному файлу?
- **11.47.** Эмпирически определить среднее число отрезков, порожденных посредством применения метода выборки с замещением с очередью по приоритетам размером 10000 к файлам с произвольной организацией при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 11.48.** Каким будет число отрезков *в худшем случае* при использовании метода выборки с замещением для генерации отрезков начального распределения для файла, содержащего  $N$  записей, с использованием очереди по приоритетам размером  $M$ , при  $M < N$ ?
- ▷ **11.49.** Показать, как выполняется сортировка ключей **E A S Y Q U E S T I O N W I T H P L E N T Y O F K E Y S** в результате применения многофазного слияния в стиле примера, показанного на диаграмме рис. 11.15.
- **11.50.** В примере многофазного слияния на рис. 11.15 мы поместили два фиктивных отрезка на магнитную ленту с 7 отрезками. Найдите другие способы распределения фиктивных отрезков на лентах и выберите среди них такой, который обеспечивает минимальное по стоимости слияние.

- 11.51.** Начертить таблицу, соответствующую рис. 11.13, с целью определить максимальное число отрезков, которые могут быть слиты с помощью сбалансированного трехпутевого слияния с пятью проходами по данным (с использованием шести устройств).
- 11.52.** Начертить таблицу, соответствующую рис. 11.16, с целью определить максимальное число отрезков, которые могут быть слиты с помощью метода многофазного слияния с теми же затратами, что и пять проходов по всей совокупности данных (с использованием шести устройств).
- **11.53.** Напишите программу, вычисляющую число проходов, выполняемых многофазным слиянием, и эффективное количество проходов, выполняемых методом многофазного слияния для заданного числа устройств и заданного числа начальных блоков. Используйте эту программу для распечатки таблицы этих затрат для каждого метода для  $P = 3, 4, 5, 10$  и  $100$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **10.54.** Напишите программу, последовательно назначающую начальные отрезки устройствам в условиях  $P$ -путевого многофазного слияния. Всякий раз, когда число отрезков есть обобщенное число Фибоначчи, отрезки должны быть назначены устройствам в соответствии с требованиями алгоритма; ваша задача заключается в том, чтобы отыскать удобный способ распределения отрезков по одному за единицу времени.
- **11.55.** Реализовать выборку с замещением, воспользовавшись интерфейсом, определенным в упражнении 11.38.
- **11.56.** Чтобы получить программную реализацию сортировки-слияния, используйте сочетание решений упражнений 11.38 и 11.55. Используйте полученную программу для сортировки файла максимально возможных в вашей системе размеров, воспользовавшись многофазным слиянием. Если возможно, определите как отражается на времени выполнения программы увеличение числа устройств.
- 11.57.** Как нужно обходиться с файлами небольших размеров в рамках реализации быстрой сортировки применительно к файлу сверхбольших размеров в среде виртуальной памяти?
- **11.58.** Если на вашем компьютере функционирует подходящая система виртуальной памяти, выполните эмпирическое сравнение быстрой сортировки, поразрядной сортировки MSD, поразрядной сортировки LSD и пирамидальной сортировки для сверхбольшого файла. Выберите файл максимально возможных в вашей системе размеров.
- **11.59.** Разработать программную реализацию рекурсивной многопутевой сортировки слиянием, в основу которой положено  $k$ -путевое слияние, которую можно использовать для сортировки сверхбольших файлов в среде виртуальной памяти (см. упражнение 8.11).
- **11.60.** Если на вашем компьютере функционирует подходящая система виртуальной памяти, эмпирически определите значение, при котором достигается минимальное время выполнения программной реализации из упражнения 11.59. Выберите файл максимально возможных в вашей системе размеров.

## 11.5. Параллельная процедура сортировки-слияния

Как сделать так, чтобы несколько независимых процессоров работали совместно, решая одну задачу сортировки? Управляют ли процессоры внешними запоминающими устройствами или сами являются самостоятельными вычислительными системами — от этого во многом зависит алгоритм функционирования высокопроизводительных вычислительных систем. Проблема параллельных вычислений в последнее время широко изучается. Разработано множество типов вычислительных машин параллельного действия, предложены разнообразные модели параллельных вычислений. Во всех этих случаях проблема сортировки выступает как эффективное средство тестирования и того, и другого.

Мы уже рассматривали вопросы параллельной обработки данных низкого уровня в процессе изучения сетей сортировки в разделе 11.2, когда обсуждали возможность одновременного выполнения нескольких операций сравнения-обмена. Сейчас мы рассмотрим модель параллельных вычислений высокого уровня с использованием независимых универсальных процессоров (а не только компараторов), имеющих доступ к одним и тем же данным. И в этом случае мы игнорируем многие практические проблемы, зато можем проводить исследования алгоритмов в данном контексте.

Абстрактная модель, которую мы используем для представления параллельной обработки данных, базируется на предположении, что сортируемые файлы распределяются между независимыми процессорами. Мы полагаем, что имеются

- $N$  записей, подлежащих сортировке
- $P$  процессоров, способных принять  $(N/P)$  записей.

Мы присваиваем этим процессорам метки  $0, 1, \dots, P-1$  и полагаем, что файл, который служит входом, помещен в память локальных процессоров (то есть, в каждом процессоре содержатся  $N/P$  записей). Цель сортировки заключается в переупорядочении записей таким образом, чтобы  $N/P$  наименьших записей находились в памяти процессора  $0$ ,  $N/P$  следующих наименьших записей находились в памяти процессора  $1$  и так далее, в отсортированном порядке. Как мы увидим далее, существует зависимость между  $P$  и общим временем выполнения — мы хотим получить количественную оценку этой зависимости с целью сравнения различных стратегий.

Предлагаемая модель — одна из многих возможных моделей параллельной обработки данных и для нее характерны многие из упрощений в смысле практического применения, какие имеют место в предложенной нами модели внешней сортировки (раздел 11.3). И в самом деле, рассматриваемая модель не учитывает одну из наиболее важных проблем, которая весьма актуальна для параллельной обработки данных: ограничения, которые накладываются на обмен данными между процессорами.

Мы полагаем, что стоимость такого обмена данных гораздо выше, чем обращения к локальной памяти, и что наибольшая эффективность достигается в тех случаях, когда такой обмен осуществляется последовательно, большими блоками данных. В этом смысле каждый процессор рассматривает остальные процессоры как внешние

запоминающие устройства. И на сей раз, эту модель с высоким уровнем абстракции можно рассматривать как неудовлетворительную в плане практического применения ввиду ее чрезмерной упрощенности; ее также можно считать неудовлетворительной и с теоретической точки зрения, поскольку ей не хватает полноты определения. Но несмотря на все это, она представляет собой структуру, в рамках которой возможна разработка полезных алгоритмов.

И в самом деле, рассматриваемая задача (с учетом сделанных выше предположений) представляет собой пример могущества абстрагирования, ибо мы для ее решения можем воспользоваться сетями сортировки, которые изучались в разделе 11.2, внося соответствующие изменения в абстракцию сортировки-слияния, которые делают ее пригодной для работы с большими блоками данных.

**Определение 11.2.** *Компаратор слияния принимает на входе два отсортированных файла размером  $M$  и выдает на выход два отсортированных файла: один из них содержит  $M$ -й наименьший из  $2M$  входных элементов, а другой содержит  $M$ -й наибольший из  $2M$  входных элементов.*

Эту операцию нетрудно реализовать: производится слияние двух входных файлов, затем на выход передаются первая половина и вторая половина файла, полученного в результате слияния.

**Лемма 11.7.** *Мы можем выполнить сортировку файла размером  $N$ , поделив его на  $M/N$  блоков размером  $M$  с последующей сортировкой каждого файла, после чего используем сеть сортировки со встроенными в нее компараторами.*

Чтобы установить этот факт, взяв за основу принцип нулей и единиц, требуются проявить определенную изобретательность (см. упражнение 11.61), однако изучение примера, подобного представленному на рис. 11.18, позволит убедиться в правильности этого утверждения.

Будем называть метод, описываемый леммой 11.7, *поблочной сортировкой (block sorting)*. Но прежде чем использовать этот метод на конкретной машине, необходимо выбрать значения некоторых параметров модели. Наш интерес к этому методу обусловлен следующей рабочей характеристикой:

**Лемма 11.8.** *Поблочная сортировка на  $P$  процессорах с использованием сортировки Бэтчера с компараторами слияния может выполнить сортировку  $N$  записей примерно за  $(\lg P)^2/2$  параллельных шагов.*

Под *параллельными шагами (parallel steps)* в данном контексте мы понимаем некоторую совокупность отдельных компараторов. Лемма 11.8 является прямым следствием лемм 11.3 и 11.7.

Чтобы реализовать компаратор слияния на двух процессорах, мы должны сделать так, чтобы оба они в процессе обмена копиями хранящихся в них блоков данных выполняли слияние (параллельно), при этом на одном из них оставалась половина с меньшими значениями ключей, а на другом — половина с большими значениями ключей. Если передача блоков происходит медленно по сравнению с быстродействием конкретного процессора, мы можем вычислить общее время, необходимое для сортировки, умножая стоимость передачи одного блока на  $(\lg P)^2/2$ . Такая оценка предполагает принятие большого числа предположений, например, предполагается, что

передача многочисленных блоков данных производится параллельно, без затрат, что довольно редко имеет место в реальных компьютерах параллельного действия. Несмотря на все это, она представляет собой отправную точку для понимания того, что можно ожидать от практической реализации.

Если стоимость передачи блока данных определяется быстродействием отдельного процессора (еще одна идеальная цель, к которой можно только приблизиться на реальных машинах), то мы должны принимать в расчет время, затрачиваемое на начальную сортировку. Каждый процессор выполняет  $(N/P) \lg(N/P)$  сравнений (параллельно), чтобы выполнить начальную сортировку  $N/P$  блоков, и примерно  $P^2 (\lg P)/2$  этапов слияния типа  $(N/P)$ -с- $(N/P)$ . Если стоимость сравнения —  $\alpha$ , а стоимость слияния на одну запись составляет  $\beta$ , общее время выполнения приблизительно равно

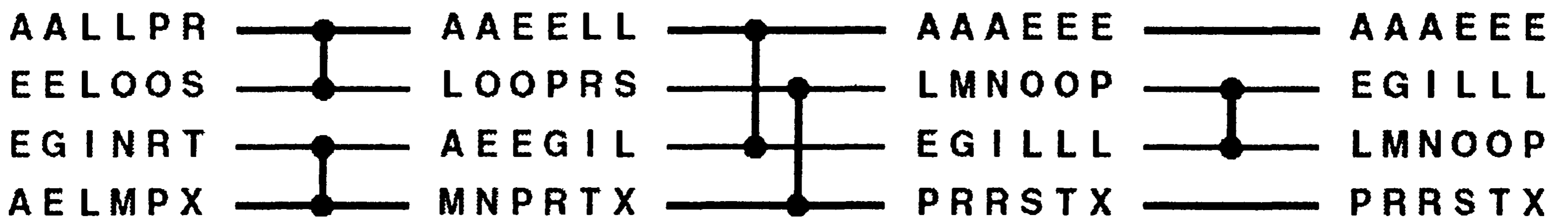
$$\alpha (N/P) \lg(N/P) + \beta (N/P) P^2 (\lg P) / 2.$$

Для сверхбольших  $N$  и малых  $P$  этот показатель — лучшее из того, на что можно рассчитывать в условиях метода параллельной сортировки, основанной на сравнении, поскольку в этом случае стоимость составляет  $\alpha (N \lg N)/P$ , которую можно считать оптимальной: любая сортировка требует  $N \lg N$  сравнений и самое лучшее, что можно предпринять в этом случае — сделать  $P$  из них одновременно. В случае больших значений  $P$  преобладает второе слагаемое и стоимость приближается к значению  $\beta N (P \lg P)/2$ , которое можно считать субоптимальным, хотя все еще конкурентоспособным. Например, второе слагаемое составляет  $256\beta N/P$  стоимости сортировки 1 миллиарда элементов на 64 процессорах, в то время как вклад первого слагаемого составляет  $32\alpha N/P$ .

Когда значение  $P$  достаточно велико, на некоторых машинах при передаче данных могут возникнуть узкие места. Если такое случится, применение идеального тасования, представленного на рис. 11.8, может быть использовано как средство управления издержками подобного рода. Именно по этим причинам в некоторых машинах имеются встроенные схемы низкоуровневых соединений, которые позволяют эффективно реализовать операции тасования.

Этот пример показывает, что в некоторых случаях можно обеспечить эффективную работу процессоров при решении задач сортировки файлов сверхбольших размеров. Чтобы знать, как это сделать наилучшим образом неизбежно потребуются проанализировать множество различных алгоритмов для данного типа параллельной машины и исследовать поведение используемой машины на модели при различных значениях ее параметров. Более того, может потребоваться совершенно другой подход к проблеме параллельной обработки данных. Тем не менее, предположение о том, что увеличение числа процессоров приводит к увеличению стоимости обмена данными между ними, является основополагающим для параллельных вычислений, а сети Бэтчера представляет собой эффективное средство управления такого рода затратами, что имеет место как на низком уровне, в чем мы имели возможность убедиться в разделе 11.2, так и на высоком уровне, в чем мы смогли убедиться в настоящем разделе.

Методы сортировки, описанные в этом разделе и в других местах данной главы, имеют определенные отличия от методов, которые мы изучали на протяжении глав 6—10, поскольку они предусматривают копирование и наличие ограничений, кото-



**РИСУНОК 11.18. ПРИМЕР ПОБЛОЧНОЙ СОРТИРОВКИ**

*Этот рисунок служит иллюстрацией того факта, что мы можем воспользоваться сетью, представленной на рис. 11.14 для сортировки блоков данных. Компараторы помещают выход в виде половины элементов с меньшими номерами на верхнюю из двух входных линии, а половину элементов с большими номерами — на нижнюю линию. Трех параллельных шагов оказывается достаточно.*

рые не рассматриваются в обычном программировании. В главах 6—10 простых предположений, касающихся используемых нами данных, было достаточно, чтобы можно было сравнивать большое число различных методов решения одной и той же базовой задачи. В противоположность этому в данной главе мы сосредоточились на формулировании различных задач, но имели возможность рассматривать лишь немногие решения каждой из них. Эти примеры служат иллюстрацией того факта, что изменения ограничений, налагаемых внешней средой, предоставляют возможности для появления новых решений, а наиболее важная часть этого процесса состоит в нахождении полезных абстрактных формулировок конкретных задач.

Сортировка играет исключительно важное значение во многих практических приложениях, и разработка эффективных методов сортировки является одной из главных задач, на решение которых должна быть ориентирована архитектура новых компьютеров и новые среды программирования. Памятуя истину, что новые достижения строятся на прошлом опыте, важно получить представление о совокупности технических средств, которые мы обсуждали в главах 6—10, в силу того факта, что появились новые революционные изобретения. Абстрактное мышление, которое понадобилось при изучении изложенного ранее материала, может оказаться необходимым, если вам придется разрабатывать быстродействующие процедуры сортировки для новых машин.

## Упражнения

- 11.61. Воспользуйтесь принципом нулей и единиц (лемма 11.1) для доказательства леммы 11.7.
- 11.62. Реализовать последовательную версию поблочной сортировки с нечетно-четным слиянием Бэтчера: (i) воспользоваться стандартной сортировкой слиянием (программы 8.3 и 8.2) для сортировки блоков данных, (ii) воспользоваться стандартным обменным слиянием (программа 8.2) для реализации компараторов слияния и (iii) воспользоваться восходящим нечетно-четным слиянием Бэтчера (программа 11.3) для сортировки отдельных блоков.
- 11.63. Дать оценку время выполнения программы, описанной в упражнении 11.62, как функции от  $N$  и  $M$  для больших  $N$ .

- **11.64.** Выполнить упражнения 11.62 и 11.63, но при этом в обоих случаях замените нечетно-четное слияние Бэтчера (программа 11.3) на программу 8.2.
- 11.65.** Найти значения  $P$ , для которых  $(N/P)\lg N = NP \lg P$ , для  $N = 10^3, 10^6, 10^9$  и  $10^{12}$ .
- 11.66.** Найти приближенные выражения вида  $c_1 N \lg N + c_2 N$  для определения числа сравнений элементов данных, используемых параллельной поблочной сортировки Бэтчера для  $P = 1, 4, 16, 64$  и  $256$ .
- 11.67.** Сколько параллельных шагов потребуется для сортировки  $10^{15}$  записей, которые распределены на 1000 дисков, используя для этой цели 100 процессоров?

### Ссылки на источники к части 3

Основным источником для ссылок в данном разделе является том 3 многотомной монографии Кнута (Knuth), посвященный вопросам сортировки и поиска. Практически по каждой теме, которые мы затрагивали до сих пор, в этой книге можно найти полезную информацию. В частности, все результаты, касающиеся рабочих характеристик различных алгоритмов, которые обсуждались в данной книге, в указанной монографии обоснованы исчерпывающим математическим анализом.

Имеется обширная литература по вопросам сортировки. Опубликованная Кнутом и Ривестом (Rivest) в 1973 г. библиография содержит сотни ссылок, благодаря которым можно ознакомиться с положением дел в области разработки и совершенствования множества классических методов, часть из которых была рассмотрена здесь. Более поздние ссылки, сопровождаемые обширной библиографией, охватывающей последние работы, вы найдете в книге Баеца-Ятца (Baeza-Yates) и Гонне (Gonnet). Обзор состояния наших знаний о сортировке Шелла можно найти в статье Седжвика (Sadgewick) за 1996 г.

Что касается быстрой сортировки, то наилучшей ссылкой может служить пионерская статья Хоара (Hoare), вышедшая в свет в 1962 г., где рассматриваются все наиболее важные варианты, которые обсуждались в главе 7. Более подробные детали, касающиеся математического анализа и практических применений многих модификаций и усовершенствований, появившихся уже после того, как этот алгоритм нашел широкое применение на практике, можно найти в статье Седжвика, опубликованной в 1978 г. Бентли (Bentley) и Мак-Илрой (McIlroy) дали его современную трактовку. Материал по трехпутевому разбиению в главе 7 и трехпутевой поразрядной быстрой сортировке в главе 10 основан именно на этой статье и статье Бентли и Мак-Илроя, появившейся в печати в 1978 г. Первый алгоритм, использующий разбиения (двоичная быстрая сортировка или поразрядная сортировка с обментами) были опубликованы в статье Хильдебрандта (Hildebrandt) и Исбитца (Isbitz) в 1959 г.

Структура данных биномиальной очереди Вийемана (Vuillemin) в том виде, в каком она была реализована и исследована Брауном (Brown), поддерживает все операции над очередями с приоритетами элегантно и эффективно. Двоичные сортирующие деревья, описанные Фредманом (Fredman), Седжвиком, Слеатором (Sleator) и Тарьяном (Tarjan), являются усовершенствованиями базового понятия и представляют немалый практический интерес.

В статье, появившейся в 1993 г., Мак-Илрой, Бостик (Bostic) и Мак-Илрой представляют положение дел с реализацией поразрядной сортировки.



## Список литературы

- R. Baeza-Yates and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.
- J. L. Bentley and M. D. McIlroy, “Engineering a sort function,” *Software—Practice and Experience* **23**, 1 (January, 1993).
- J. L. Bentley and R. Sedgwick, “Sorting and searching strings,” Eighth Symposium on Discrete Algorithms, New Orleans, January, 1997.
- M. R. Brown, “Implementation and analysis of binomial queue algorithms,” *SIAM Journal of Computing* **7**, 3 (August, 1978).
- M. L. Fredman, R. Sedgwick, D. D. Sleator, and R. E. Tarjan, “The pairing heap: a new form of self-adjusting heap,” *Algorithmica* **1**, 1 (1986).
- P. Hildebrandt and H. Isbitz, “Radix exchange — an internal sorting method for digital computers,” *Journal of the ACM*, **6**, 2 (1959).
- C. A. R. Hoare, “Quicksort,” *Computer Journal*, **5**, 1 (1962).
- D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1997.
- P. M. McIlroy, K. Bostic, and M. D. McIlroy, “Engineering radix sort,” *Computing Systems* **6**, 1 (1993).
- R. L. Rivest and D. E. Knuth, “Bibliography 26: Computing Sorting,” *Computing Reviews*, **13** 6 (June, 1972).
- R. Sedgwick, “Implementing quicksort programs,” *Communications of the ACM* **21**, 10 (October 1978).
- R. Sedgwick, “Analysis of shellsort and related algorithms,” Fourth European Symposium on Algorithms, Barcelona, September, 1996.
- J. Vuillemin, “A data structure for manipulating priority queues,” *Communications of the ACM* **21**, 4 (April 1978).

# Поиск

*В этой части:*

- 12**    **Таблицы символов и деревья бинарного поиска**
- 13**    **Сбалансированные деревья**
- 14**    **Хеширование**
- 15**    **Поразрядный поиск**
- 16**    **Внешний поиск**

# Таблицы символов и деревья бинарного поиска

**П**олучение конкретного фрагмента или фрагментов информации из больших томов ранее сохраненных данных — основополагающая операция, называемая *поиском*, характерная для многих вычислительных задач. Как и в случае с алгоритмами сортировки, описанными в главах с 6 по 11, и очередями конкретного приоритета, описанными в главе 9, мы работаем с данными, разделенными на записи, или *элементы*, каждый из которых имеет *ключ*, используемый при поиске. Цель поиска — отыскание элементов с ключами, которые соответствуют заданному *ключу поиска*. Обычно, назначением поиска является получение доступа к информации внутри элемента (а не просто к ключу) с целью ее обработки.

Поиск используется повсеместно и связан с выполнением множества различных операций. Например, в банке требуется отслеживать информацию о счетах всех клиентов и выполнять поиск в этих записях для подведения баланса и выполнения банковских операций. На авиалинии необходимо отслеживать количество мест на каждом рейсе и выполнять поиск свободных мест, отказа в продаже билетов или внесения каких-либо изменений в списки пассажиров. Еще один пример — средство поиска в сетевом интерфейсе программы, которое отыскивает в сети все документы, содержащие заданное ключевое слово. Требования, предъявляемые к этим приложениям, в чем-то совпадают (и для банка, и для авиалинии требуются точность и надежность), а в чем-то различны (банковские данные имеют длительный срок хранения по сравнению

с данными остальных упомянутых приложений); тем не менее, во всех случаях требуются эффективные алгоритмы поиска.

**Определение 12.1** *Таблица символов* — это структура данных элементов с ключами, которая поддерживает две базовых операции: вставку нового элемента и возврат элемента с заданным ключом.

Иногда таблицы символов называют также *словарями* (*dictionary*), по аналогии с проверенной временем системой предоставления определений слов путем перечисления их в справочнике в алфавитном порядке. Так, в словаре английского (или любого другого) языка "ключи" — это слова, а "элементы" — связанные со словами записи, которые содержат определение, правила произношения и другую информацию. Алгоритмы поиска, используемые для отыскания информации в словаре, обычно основываются на алфавитном расположении записей. Телефонные книги, энциклопедии и другие справочники, в основном, организованы таким же образом, и некоторые из рассматриваемых методов поиска (например, алгоритм бинарного поиска в разделах 2.6 и 12.4), также основываются на том, что записи упорядочены.

Таблицы символов в компьютерах обладают преимуществом в том, что они значительно более динамичны, чем словарь или телефонная книга. Поэтому большинство рассматриваемых методов создают структуры данных, которые не только позволяют использовать эффективные алгоритмы поиска, но и поддерживают эффективные реализации операций добавления новых элементов, удаления или изменения элементов, объединения двух таблиц символов в одну и т.п. В этой главе будут вновь рассматриваться многие из вопросов, связанных с операциями, которые исследовались применительно к очередям по приоритетам в главе 9. Разработка динамических структур данных для поддержки поиска — одна из старейших и наиболее широко изученных проблем в компьютерных науках; она будет находиться в центре нашего внимания как в этой главе, так и в главах 13—16. Как будет показано, для решения задачи реализации таблиц символов разработаны (и продолжают разрабатываться) множество оригинальных алгоритмов.

Теоретики компьютерных наук и программисты интенсивно исследуют и другие применения таблиц символов, кроме упомянутых основных, поскольку эти таблицы — незаменимое вспомогательное средство при организации программного обеспечения в компьютерных системах. Таблица символов служит словарем для программы: ключи — это символические имена, используемые в программе, а элементы содержат информацию, описывающую именованные объекты. Начиная с зары развития компьютерной техники, когда таблицы символов позволяли программистам переходить от использования числовых адресов в машинных кодах к символическим именам языка ассемблера, и завершая современными приложениями нового тысячелетия, когда символические имена имеют определенное значение в рамках всемирных компьютерных сетей, быстрые алгоритмы поиска играли и будут играть важную роль при компьютерной обработке.

Таблицы символов часто встречаются также в абстракциях нижнего уровня, а иногда и на аппаратном уровне. Для описания этого понятия иногда используется термин *ассоциативная память*. Мы уделим основное внимание программным реализациям, но некоторые из рассматриваемых методов применимы также и к аппаратной реализации.

Как и при изучении методов сортировки в главе 6, в этой главе изучение методов поиска начинается с рассмотрения ряда элементарных методов, пригодных для небольших таблиц и в некоторых особых ситуациях, и которые иллюстрируют базовые технологии, используемые более совершенными методами. Затем, в остальной части главы основное внимание уделяется *дереву бинарного поиска (binary search tree — BST)*, основополагающей и широко используемой структуре данных, допускающей применение алгоритмов быстрого поиска.

В разделе 2.6 в качестве иллюстрации эффективности математического анализа при разработке эффективных алгоритмов рассматривались два алгоритма поиска. Для полноты изложенного в этой главе материала мы повторим часть информации, приведенной в главе 2.6; в ряде случаев для ознакомления с некоторыми доказательствами будут приводиться ссылки на эту главу. Позже в этой главе мы обратимся также к основным свойствам двоичных деревьев, которые исследуются в разделах 5.4 и 5.5.

## 12.1 Абстрактный тип данных таблицы символов

Как и при рассмотрении очередей приоритета, алгоритмы поиска можно рассматривать как принадлежащие к интерфейсам, объявляющим множество общих операций, которые могут быть отделены от конкретных реализаций, что позволяет легко и просто заменять одни реализации другими. Интерес представляют следующие операции:

- *Вставка* нового элемента.
- *Поиск* элемента (или элементов) с заданным ключом.
- *Удаление* указанного элемента.
- *Выбор k-го* по величине элемента в таблице символов.
- *Сортировка* таблицы символов (отображение всех элементов в порядке их ключей).
- *Объединение* двух таблиц символов.

Подобно множеству других структур данных, к этому набору может потребоваться добавить стандартные операции *создания, проверки, не пуст ли элемент* и, возможно, *уничтожения и копирования*. Кроме того, может потребоваться рассмотрение различных других практических изменений основного интерфейса. Например, часто операция *поиска и вставки* оказывается весьма полезной, поскольку во многих реализациях поиск ключа, даже безуспешный, тем не менее, предоставляет точную информацию, необходимую для вставки нового элемента с этим ключом.

В общем случае термин "алгоритм поиска" используется в значении "реализация абстрактного типа данных таблицы символов", хотя, строго говоря, последний термин предполагает определение и построение основополагающей структуры данных таблицы символов и, в дополнение к поиску, реализацию операций абстрактного типа данных. В связи с важностью таблиц символов для столь многих компьютерных приложений они доступны в качестве высокоуровневой абстракций во многих средах программирования. Стандартная библиотека C содержит `bsearch`, т.е. реализацию алгоритма бинарного поиска, описанного в разделе 12.4, а библиотека стандартных шаблонов C++ предоставляет большое множество таблиц символов, называемых "ассоциатив-

ными контейнерами". Как обычно, трудно добиться, чтобы реализация общего назначения удовлетворяла требованиям производительности, предъявляемым к специализированным приложениям. В процессе изучения многих оригинальных методов, разработанных для реализации абстракции таблицы символов, мы определим контекст, который поможет понять характеристики готовых реализаций и принять решение о необходимости реализации, предназначенной для конкретного приложения.

Как и в случае с сортировкой, мы рассмотрим методы без определения типов обрабатываемых элементов. Столь же подробно, как в разделе 6.8, будут исследоваться реализации, использующие интерфейс, который определяет **Item** и основные абстрактные операции с данными. Мы рассмотрим методы как с использованием сравнения, так и с использованием корня, использующие в качестве индексов ключи или фрагменты ключей. Чтобы разделить роли, выполняемые при поиске элементами и ключами, понятие **Item** (элемент), использованное в главах 6—11, расширяется до элементов, содержащих ключи типа **Key**. Поскольку требуется несколько больше элементов, чем было необходимо для ознакомления с алгоритмами сортировки, будем считать, что они объединены в пакеты абстрактных типов данных, реализованные с помощью классов C++, как показано в программе 12.1. Функция-член **key()** будет применяться для извлечения ключей из элементов, а перегруженная операция **operator==** — для проверки равенства двух ключей. В этой главе и главе 13 также перегружается **operator<** для сравнения значений двух ключей, что помогает при поиске; алгоритмы поиска, описанные в главах 14 и 15, основываются на извлечении фрагментов ключей за счет использования базовых операций с корнями, которые использовались в главе 10. Кроме того предполагается, что элементы инициализируются *нулевыми значениями (null)*, и что клиенты имеют доступ к функции **null()**, которая может проверять, является ли элемент нулевым. Нулевые элементы используются для поддержки возвращаемого значения в том случае, когда ни один элемент в таблице символов не имеет искомого ключа. В некоторых реализациях предполагается, что нулевые элементы имеют служебный ключ.

### Программа 12.1 Пример реализации АДЭ элемента

Это определение класса элементов, представляющих собой небольшие записи, состоящие из целочисленных ключей и связанной с ними информации в виде значений с плавающей точкой, иллюстрирует основные соглашения в отношении элементов таблиц символов. Наши реализации таблиц символов — клиентские программы, в которых операции **==** и **<** используются для сравнения ключей, а функции-члены **key()** и **null()** — соответственно для получения доступа к ключам и проверки, являются ли элементы нулевыми.

В определении типа элемента были также включены функции **scan** (считывающая **Item**), **rand** (генерирующая произвольный **Item**) и **show** (выводящая **Item**), которые будут использоваться драйверами. Это позволяет создавать и тестировать различные реализации таблиц символов, состоящие из различных типов элементов.

```
#include <stdlib.h>
#include <iostream.h>
static int maxKey = 1000;
typedef int Key;
class Item
{
private:
```

```
    Key keyval;
    float info;
public:
    Item()
        { keyval = maxKey; }
    Key key()
        { return keyval; }
    int null()
        { return keyval == maxKey; }
    void rand()
        { keyval = 1000*::rand()/RAND_MAX;
          info = 1.0*::rand()/RAND_MAX; }
    int scan(istream& is = cin)
        { return (is >> keyval >> info) != 0; }
    void show(ostream& os = cout)
        { os << keyval << " " << info << endl; }
};
ostream& operator<<(ostream& os, Item& x)
{ x.show(os); return os; }
```

Чтобы использовать при поиске интерфейсы и реализации для чисел с плавающей точкой, строк и более сложных элементов, описанных в разделах 6.8 и 6.9, нужно только убедиться, что в `Key` присутствуют определения `key()`, `null()`, `operator==` и `operator<`, а также изменить `rand`, `scan` и `show`, сделав их функциями-членами, которые ссылаются на ключи соответствующим образом.

Программа 12.2 — интерфейс, который определяет базовые операции таблицы символов (за исключением операции *объединить* (*join*)). Этот интерфейс будет использоваться в клиентских программах и всех реализациях поиска в этой и нескольких следующих главах. Абстрактный тип данных первого класса не применяется в том смысле, как это принималось в разделе 4.8 (см. упражнение 12.6), поскольку в большинстве программ используется только одна таблица, а добавление конструкторов копирования, перегруженных операций присваивания и деструкторов, хоть и несложная задача в большинстве реализаций, но все же она отвлекала бы от важных характеристик алгоритмов. В программе 12.2 можно было бы также определить версию интерфейса для манипулирования дескрипторами элементов подобно программе 9.8 (см. упражнение 12.7), но это излишне усложняет программу в типичной ситуации, когда достаточно манипулировать элементом посредством ключа. Интерфейс не задает способ определения элемента, который должен быть *удален*. В большинстве реализаций используется интерпретация "удалить элемент с ключом, равным данному элементу", при этом подразумевается предварительный *поиск*. В других реализациях, которые предоставляют дескрипторы и могут выполнять проверку идентичности элемента, необходимость *поиска* перед удалением исключается, и поэтому для них допустимы более быстрые алгоритмы. Кроме того, при рассмотрении алгоритмов для операции *объединить*, предполагающих наличие приложений, которые обрабатывают несколько таблиц символов, можно использовать реализации АДТ первого класса таблицы символов, которые сводят к минимуму напрасную трату времени и расход памяти (см. раздел 12.9).

---

**Программа 12.2 АТД таблицы символов**

---

В этом интерфейсе определены операции для простой таблицы символов: инициализация, возврат значения счетчика элементов, поиск элемента с заданным ключом, добавление нового элемента, удаление элемента, выбор  $k$ -го наименьшего элемента и отображение элементов в порядке их ключей (в указанном выходном потоке).

```
template <class Item, class Key>
class ST
{
    private:
        // Код, зависящий от реализации
    public:
        ST(int);
        int count();
        Item search(Key);
        void insert(Item);
        void remove(Item);
        Item select(int);
        void show(ostream&);
};
```

---

В некоторых алгоритмах не предполагается наличие какого-либо определенного порядка ключей, и поэтому для сравнения ключей в них используется только `operator==` (а не `operator<`), однако во многих реализациях таблиц символов используется упорядоченная организация ключей, применяемых в `operator<` для структурирования данных и управления поиском. Кроме того, абстрактные операции *select* (выбор) и *sort* (сортировка) явно ссылаются на порядок ключей. Функция *sort* объединяется в пакет в виде функции, которая отправляет все элементы в выходной поток по порядку без обязательной переконфигурации. Реализации *sort* можно легко обобщить с целью получения функции, которая посещает элементы в порядке их ключей, возможно, применяя к каждому из них процедуру, переданную в аргументе. Используемые функции *sort* для таблиц символов мы назвали *show*, поскольку приведенные реализации обеспечивают отображение содержимого таблицы символов отсортированным по порядку. Для алгоритмов, в которых `operator<` не используется, нет необходимости, чтобы ключи были сравнимы друг с другом, поэтому такие алгоритмы необязательно поддерживают операции *select* и *sort*.

Случай возможного существования элементов с дублированными ключами при создании реализации таблицы символов должен рассматриваться особо. Некоторые приложения не допускают существования дублированных ключей, поэтому ключи могут использоваться в качестве дескрипторов. Примером такой ситуации может служить использование номеров карточек социального страхования в качестве ключей для персональных файлов. Напротив, в других приложениях может предполагаться наличие нескольких элементов с одинаковыми ключами: например, поиск по ключевому слову в базах данных документов, как правило, будет приводить к нескольким совпадениям с условием поиска.

Обработку элементов с дублированными ключами можно выполнять одним из нескольких способов. Один из подходов — настаивание на том, чтобы искомая в первую очередь структура данных содержала только элементы с различными ключами и обеспечение для каждого ключа ссылки на список элементов приложения, содержащих дублированные ключи. То есть, в базовых структурах данных используются элементы, которые содержат ключ и ссылку, и отсутствуют элементы с одинаковыми ключами.



Подобная организация удобна в некоторых приложениях, поскольку все элементы с данным искомым ключом возвращаются в результате выполнения одной операции *search* (поиск) или могут быть удалены одной операцией *remove* (удаление). С точки зрения реализации, эта организация эквивалентна предоставлению управления дублированными ключами клиенту. Вторая возможность — оставление элементов с одинаковыми ключами в главной структуре данных поиска и возврат в результате поиска *любого* элемента с данным ключом. Это соглашение проще для приложений, которые обрабатывают элементы по одному, когда порядок обработки элементов с одинаковыми ключами не важен. Однако, это может оказаться неудобным с точки зрения разработки алгоритма, поскольку может потребоваться расширение интерфейса за счет включения в него механизма для получения всех элементов с данным ключом или для вызова указанной функции для каждого элемента с конкретным ключом. Третья возможность — принять, что каждый элемент имеет уникальный идентификатор (кроме ключа) и потребовать, чтобы функция *search* отыскивала элемент с данным идентификатором при заданном ключе. Разумеется, может потребоваться и какой-либо более сложный механизм. Эти рассуждения применимы ко всем операциям на таблицах символов при наличии дублированных ключей. Нужно ли *удалить* все элементы с данным ключом, любой элемент с ключом или конкретный элемент (для чего требуется реализация, поддерживающая дескрипторы элементов)? При описании реализаций таблиц символов мы неформально указываем возможный наиболее удобный способ обработки элементов с одинаковыми ключами, не обязательно рассматривая каждый механизм для каждой реализации.

Программа 12.3 — пример клиентской программы, иллюстрирующей упомянутые выше соглашения для реализаций таблиц символов. Программа использует таблицу символов для поиска различных значений в последовательности ключей (сгенерированной произвольно или считанной со стандартного ввода), а затем выводит их в отсортированном порядке.

### Программа 12.3 Пример клиента для таблицы символов

В этой программе таблица символов используется для поиска отдельных ключей в произвольно сгенерированной или считанной со стандартного ввода последовательности. Для каждого ключа операция **search** используется для проверки того, просматривался ли ключ раньше. Если ранее ключ не просматривался, функция вставляет элемент с этим ключом в таблицу символов. Типы ключей и элементов, а также абстрактные операции с ними определены в **Item.cxx** (см., например, программу 12.1).

```
#include <iostream.h>
#include <stdlib.h>
#include "Item.cxx"
#include "ST.cxx"
int main(int argc, char *argv[])
{ int N, maxN = atoi(argv[1]), sw = atoi(argv[2]);
  ST<Item, Key> st(maxN);
  for (N = 0; N < maxN; N++)
    { Item v;
      if (sw) v.rand(); else if (!v.scan()) break;
      if (!(st.search(v.key())) .null()) continue;
      st.insert(v);
    }
}
```

```
st.show(cout); cout << endl;
cout << N << " keys" << endl;
cout << st.count() << " distinct keys" << endl;
}
```

Как обычно, следует иметь в виду, что различные реализации операций на таблицах символов обладают различными характеристиками производительности, которые могут зависеть от конкретного набора операций. В одном приложении операция *insert* может использоваться сравнительно редко (возможно, для построения таблицы) при огромном количестве выполняемых операций *search*; в другом, в сравнительно небольших таблицах, может выполняться огромное количество операций *insert* и *remove*, перемежаемое операциями *search*. Не все реализации будут поддерживать все операции, и некоторые из них могут обеспечивать эффективную поддержку определенных функций за счет других; при этом явно предполагается, что менее эффективные функции выполняются редко. Каждая из базовых операций в интерфейсе таблицы символов находит важные применения, поэтому для обеспечения эффективного использования различных комбинаций операций предлагается множество базовых вариантов реализации. В этой и нескольких следующих главах основное внимание будет уделено реализациям базовых функций *construct*, *insert* и *search* с приведением некоторых пояснений относительно функций *remove*, *select*, *sort* и *join*, когда в этом возникнет необходимость. Широкое множество алгоритмов, требующих рассмотрения, обусловлено различием характеристик производительности различных комбинаций базовых операций, а также ограничениями, накладываемыми на значения ключей, размерами элементов и другими факторами.

В этой главе мы встретимся с реализациями, в которых среднее время выполнения операций *search*, *insert*, *remove* и *select* для произвольных ключей пропорционально логарифму количества элементов в словаре, а время выполнения операции *sort* линейно зависит от количества элементов. В главе 13 мы исследуем способы обеспечения этого уровня производительности; кроме того, в разделе 12.2 будет приведена одна, а в главах 14 и 15 — несколько реализаций, производительность которых при определенных условиях остается постоянной.

Изучены и многие другие операции с таблицами символов. Примерами могут служить *поиск от метки (finger search)*, при котором поиск может начинаться с точки, в которой завершился предыдущий поиск; *поиск в диапазоне*, когда нужно подсчитать или отобразить все узлы, попадающие в заданный интервал; и *поиск ближайшего соседа* (если определено понятие расстояния), при котором выполняется поиск ключей, ближайших к данному.

## Упражнения

▷ **12.1** Создать реализацию класса **Item** (аналогичную программе 12.1), поддерживающего обработку реализациями таблиц символов элементов, состоящих исключительно из целочисленных ключей.

**12.2** Создать реализацию класса **Item** (аналогичную программе 12.1), поддерживающего обработку реализациями таблиц символов элементов, состоящих исключительно из строковых ключей в стиле C, а также поддерживающую буфер для строк, как это делается в программе 6.11.

- ▷ 12.3 Используя программу 12.2 АДТ таблицы символов, создать реализации АДТ стека и очереди.
- ▷ 12.4 Используйте АДТ таблицы символов, определенный программой интерфейса 12.2, создать реализацию АДТ очереди по приоритету, который поддерживает операции удаления *как* максимального, *так и* минимального элементов.  
12.5 Используя АДТ таблицы символов, определенный программой интерфейса 12.2, создать реализацию сортировки массива, совместимой с реализациями, описанными в главах 6—10.
- ▷ 12.6 Добавьте в программу 12.2 объявления деструктора, конструктора копирования и перегруженной операции присваивания, чтобы преобразовать ее в АДТ первого класса (см. разделы 4.8 и 9.5).  
12.7 Определите интерфейс АДТ таблицы символов, который позволяет клиентским программам удалять конкретные элементы с использованием дескрипторов и изменять ключи (см. разделы 4.8 и 9.5).
- ▷ 12.8 Приведите интерфейс типа элемента и реализацию элементов с двумя полями: 16-битным целочисленным ключом и строкой *S*, которая содержит информацию, связанную с этим ключом.
- 12.9 Укажите среднее количество отдельных ключей, которые программа-драйвер (программа 12.3) будет находить среди  $N$  произвольных целых чисел, меньших 1000, для  $N = 10, 10^2, 10^3, 10^4$  и  $10^5$ . Определите свой ответ эмпирически, аналитически или обоими методами.

## 12.2 Поиск с использованием индексации по ключам

Предположим, что значения ключей — отдельные небольшие числа. В этом случае простейший алгоритм поиска основывается на сохранении элементов в массиве, индексированном по ключам, как сделано в реализации, приведенной в программе 12.4. Код весьма прост: оператор `new[]` инициализирует все записи значением `nullItem`, затем мы вставляем (*insert*) элемент со значением ключа `k`, просто сохраняя его в массиве `st[k]`, и выполняем поиск (*search*) элемента со значением ключа `k`, отыскивая его в `st[k]`. Для удаления (*remove*) элемента со значением ключа `k` значение `nullItem` помещается в `st[k]`. Реализации операций выбора (*select*), сортировки (*sort*) и подсчета (*count*) в программе 12.4 используют линейный просмотр массива с пропуском нулевых элементов. Реализация предоставляет клиенту решать задачи обработки элементов с дублированными ключами и проверки таких условий, как указание операции *remove* для ключа, отсутствующего в таблице.

Эта реализация служит отправной точкой для всех реализаций таблиц символов, которые рассматриваются в этой главе и главах 13—15. Как таковая она может использоваться для различных клиентов и с различными типами элементов. Компилятор будет проверять, подчиняются ли интерфейс, реализация и клиент одним и тем же соглашениям.

Операция индексации, на которой основывается поиск с использованием индексации по ключам, совпадает с базовой операцией в методе сортировки с подсчетом индексированных ключей, который был исследован в разделе 6.10. Когда это возможно,

следует выбирать метод поиска с использованием индексации по ключам, поскольку операции *search* и *insert* трудно реализовать эффективнее.

Если элементы вообще отсутствуют (имеются только ключи), можно использовать таблицу бит. В этом случае таблица символов называется *таблицей существования* (*existence table*), поскольку  $k$ -ый разряд можно считать признаком существования  $k$  в наборе ключей таблицы. Например, на 32-разрядном компьютере этот метод можно было бы использовать для быстрого выяснения того, используется ли уже конкретный 4-значный номер телефонного коммутатора, используя таблицу из 313 слов.

**Лемма 12.1** *Если значения ключей — положительные целые числа меньше  $M$ , и элементы имеют различные ключи, то тип данных таблицы символов может быть реализован посредством индексированных по ключам массивов элементов так, чтобы для выполнения операций **insert**, **search** и **remove** требовалось постоянное время; время для выполнения операций **initialize**, **select** и **sort** пропорционально  $M$  всегда, когда любая из операций выполняется по отношению к таблице, состоящей из  $N$ -элементов.*

Это свойство становится очевидным после ознакомления с кодом. Обратите внимание, что на ключи накладывается условие  $N \leq M$ .

Программа 12.4 не обрабатывает дублированные ключи и в ней предполагается, что значения ключей лежат в пределах между  $0$  и  $M-1$ . Для хранения любых элементов с одинаковыми ключами можно было бы использовать связные списки или один из подходов, упомянутых в разделе 12.1, а перед использованием ключей в качестве индексов можно было бы выполнить их простые преобразования (см. упражнение 12.13). Но мы отложим подробное рассмотрение таких случаев вплоть до главы 14, в которой рассматривается *хеширование*, при котором для реализации таблиц символов для общих ключей используется этот же подход, заключающийся в преобразовании ключей из потенциально широкого диапазона в узкий с дополнительными действиями для элементов с дублированными ключами. Пока будем предполагать, что старым элементом со значением ключа, равным ключу вставляемого элемента, можно молча пренебречь (как в программе 12.4), или же считать это условием ошибки (см. упражнение 12.10).

#### Программа 12.4 Таблица символов, основывающаяся на индексированном по ключам массиве

В этой программе предполагается, что значения ключей — положительные целые числа, меньшие зарезервированного значения  $M$ , и они используются в качестве индексов массива. В силе соглашение, что конструктор **Item** создает элементы со значениями ключей, равными зарезервированному значению, чтобы конструктор **ST** мог отыскать значение  $M$  в нулевом элементе. При этом, прежде всего, необходимо следить за объемом памяти, который требуется, когда зарезервированное значение велико, и за временем, необходимым конструктору **ST**, когда значение  $N$  мало по сравнению со значением  $M$ .

```
template <class Item, class Key>
class ST
{
private:
    Item nullItem, *st;
    int M;
public:
```

```

ST(int maxN)
  { M = nullItem.key(); st = new Item[M]; }
int count()
  { int N = 0;
    for (int i = 0; i < M; i++)
      if (!st[i].null()) N++;
    return N;
  }
void insert(Item x)
  { st[x.key()] = x; }
Item search(Key v)
  { return st[v]; }
void remove(Item x)
  { st[x.key()] = nullItem; }
Item select(int k)
  { for (int i = 0; i < M; i++)
    if (!st[i].null())
      if (k-- == 0) return st[i];
    return nullItem;
  }
void show(ostream& os)
  { for (int i = 0; i < M; i++)
    if (!st[i].null()) st[i].show(os); }
};

```

Реализация операции *count* в программе 12.4 — пример "ленивого" подхода, когда действия выполняются только при вызове функции *count*. Альтернативный ("энергичный") подход заключается в поддержке локальной переменной счетчика непустых позиций таблицы с увеличением значения переменной, когда вставка (*insert*) выполняется в позицию таблицы, содержащую *nullItem*, и с уменьшением значения счетчика, если удаление (*remove*) выполняется по отношению к позиции таблицы, не содержащей *nullItem* (см. упражнение 12.11). "Ленивый" подход предпочтительнее, если операция *count* используется редко (или вообще не используется), а количество возможных значений ключей мало; "энергичный" подход предпочтительнее, если операция *count* используется часто или если количество возможных значений ключей очень велико. Для подпрограммы библиотеки общего назначения "энергичный" подход предпочтительней, поскольку он обеспечивает оптимальную производительность для наихудшего случая при небольшом постоянном коэффициенте увеличения затрат на выполнение операций *insert* и *remove*. Для внутреннего цикла в приложении с очень большим количеством операций *insert* и *remove*, но незначительным количеством операций *count* "ленивый" подход оказывается предпочтительней, поскольку обеспечивает наиболее быструю реализацию часто выполняемых операций. Как мы уже неоднократно убеждались, подобная дилемма типична для разработки АТД, которые должны поддерживать различные наборы операций.

При разработке интерфейса общего назначения приходится принимать и ряде других решений. Например, должен ли диапазон ключей быть одинаковым для всех объектов или различным для различных объектов? При выборе последнего варианта может потребоваться добавить аргументы к конструктору и определить функции, предоставляющие клиенту доступ к диапазону ключей.

Индексированные по ключам массивы удобны для многих приложений, но они неприменимы, если ключи не попадают в узкий диапазон. Действительно, можно считать, что эта и несколько следующих глав посвящены разработке решений для случая, когда диапазон возможных значений ключей столь широк, что невозможно использовать индексированную таблицу с одной потенциальной записью для каждого ключа.

## Упражнения

**12.10** Реализуйте АДТ таблицы символов первого класса (см. упражнение 12.6), используя динамически распределяемые массивы, индексированные по ключам.

▷ **12.11** Измените реализацию, представленную в программе 12.4, чтобы обеспечить "энергичную" реализацию функции **count** (путем отслеживания количества ненулевых записей).

▷ **12.12** Измените реализацию, созданную в упражнении 12.10, чтобы обеспечить "энергичную" реализацию функции **count** (см. упражнение 12.11).

**12.13** Разработайте версию программы 12.4, в которой используется функция **h(Key)**, преобразующая ключи в неотрицательные целые числа меньше **M** так, чтобы никакие два ключа не отображались одним и тем же целым числом. (Это усовершенствование делает реализацию полезной, когда ключи относятся к узкому диапазону (не обязательно начинающемуся с 0) и в других простых случаях.)

**12.14** Разработайте версию программы 12.4 для случая, когда элементы представляют собой ключи, являющиеся положительными целыми числами меньшими **M** (без какой-либо связанной информации). В реализации используйте динамически распределенный массив, состоящий приблизительно из **M/bitword** слов, где **bitword** — количество бит в одном слове в используемой компьютерной системе.

**12.15** Используйте реализацию, созданную в упражнении 12.14, для экспериментального определения среднего и стандартного отклонений количества отдельных целых чисел в произвольной последовательности **N** неотрицательных целых чисел меньших **N** для **N** близкого к объему памяти, доступному программе в используемом компьютере и выраженному в битах (см. программу 12.3)

## 12.3 Последовательный поиск

В общем случае, когда значения ключей относятся к слишком большому диапазону, чтобы их можно было использовать в качестве индексов, один из простых подходов к реализации таблиц символов — последовательное сохранение элементов в массиве в упорядоченном виде. Когда требуется вставить новый элемент, мы вставляем его в массив, перемещая большие элементы на одну позицию, как это делалось для сортировки вставками; когда необходимо выполнить поиск, массив просматривается последовательно. Поскольку массив упорядочен, при встрече ключа, значение которого больше искомого, можно сделать вывод о неудаче поиска. Более того, благодаря упорядочению массива, реализация операций *select* и *sort* не представляет сложности. Программа 12.5 содержит реализацию таблицы символов, которая основывается на этом подходе.

---

**Программа 12.5 Таблица символов (упорядоченная) с использованием массива**

---

Подобно программе 12.4, в этой реализации используется массив элементов, но для нее не обязательно, чтобы ключи были небольшими целыми числами. Поддержание упорядоченности массива обеспечивается тем, что при вставке нового элемента большие элементы смещаются с целью освобождения места, как это делается при сортировке вставками. В этом случае функция **search** может выполнять в массиве поиск элемента с указанным ключом, возвращая значение **nullItem** при обнаружении элемента с большим ключом. Реализация функций **select** и **sort** тривиальны, а реализация функции **remove** оставляется на самостоятельную проработку (см. упражнение 12.16).

```
template <class Item, class Key>
class ST
{
private:
    Item nullItem, *st;
    int N;
public:
    ST(int maxN)
        { st = new Item[maxN+1]; N = 0; }
    int count()
        { return N; }
    void insert(Item x)
        { int i = N++; Key v = x.key();
          while (i > 0 && v < st[i-1].key())
              { st[i] = st[i-1]; i--; }
          st[i] = x;
        }
    Item search(Key v)
        {
          for (int i = 0; i < N; i++)
              if (!(st[i].key() < v)) break;
          if (v == st[i].key()) return st[i];
          return nullItem;
        }
    Item select(int k)
        { return st[k]; }
    void show(ostream& os)
        { int i = 0;
          while (i < N) st[i++].show(os); }
};
```

---

В программе 12.5 можно было бы несколько усовершенствовать внутренний цикл в реализации операции *search* за счет использования служебного значения для исключения проверки на предмет выхода за пределы массива в том случае, если ни один из элементов таблицы не содержит искомого ключа. В частности, следующую после конца массива позицию можно было бы сохранить в качестве служебной, а затем перед поиском заполнить ее поле ключа искомым значением. В таком случае поиск всегда будет завершаться на элементе, содержащем искомый ключ, а то, находился ли ключ в таблице, всегда можно определить, проверив, является ли данный элемент служебным.

Другой подход связан с созданием реализации, в которой размещение элементов в массиве по порядку не является обязательным. При вставке новый элемент помещается в конец массива; во время поиска массив просматривается последовательно.

Этот подход характеризуется тем, что операция *insert* выполняется быстро, а операции *select* и *sort* требуют значительно большего объема работы (для выполнения каждой из них требуется реализация одного из методов, описанных в главах 7—10). Удаление (*remove*) элемента с указанным ключом можно выполнить, отыскав его, а затем переместив последний элемент массива в позицию удаляемого элемента и уменьшив размер массива на 1; удаление всех элементов с заданным ключом реализуется путем повторения этой операции. Если доступен дескриптор, предоставляющий индекс элемента в массиве, поиск не требуется и операция *remove* выполняется за постоянное время.

Еще одна простая реализация таблицы символов — использование связного списка. В этом случае можно также хранить список в упорядоченном виде с целью упрощения поддержки операции *sort* либо оставить его неупорядоченным для ускорения операции *insert*. Программа 12.6 демонстрирует второй подход. Как обычно, преимущество применения связных списков по сравнению с массивами состоит в том, что вовсе не обязательно заранее точно определять максимальный размер таблицы, а недостаток — в необходимости расхода дополнительного объема памяти (под ссылки) и невозможности эффективной поддержки операции *select*.

#### Программа 12.6 Таблица символов (неупорядоченная) с использованием связного списка

В этой реализации операций *construct*, *count*, *search* и *insert* используется односвязный список, каждый узел которого содержит элемент с ключом и ссылкой. Функция *insert* помещает новый элемент в начало списка и выполняется за постоянное время. Функция-член *search* использует приватную рекурсивную функцию *searchR* для просмотра списка.

Поскольку список не упорядочен, реализации операций *sort* и *select* опущены.

```
#include <stdlib.h>
template <class Item, class Key>
class ST
{
private:
    Item nullItem;
    struct node
    { Item item; node* next;
      node(Item x, node* t)
        { item = x; next = t; }
    };
    typedef node *link;
    int N;
    link head;
    Item searchR(link t, Key v)
    { if (t == 0) return nullItem;
      if (t->item.key() == v) return t->item;
      return searchR(t->next, v);
    }
public:
    ST(int maxN)
    { head = 0; N = 0; }
    int count()
    { return N; }
    Item search(Key v)
    { return searchR(head, v); }
    void insert(Item x)
    { head = new node(x, head); N++; }
};
```



Подходы с использованием неупорядоченного массива и неупорядоченного списка оставлены для самостоятельной реализации (см. упражнения 12.20 и 12.21). Все четыре упомянутых подхода (с использованием массивов и списков, упорядоченных и неупорядоченных) могут использоваться в приложениях один вместо другого, отличаясь только временем выполнения и требуемым объемом памяти. В этой и нескольких следующих главах исследуется множество различных подходов к решению задачи реализации таблиц символов.

Сохранение элементов в упорядоченном виде — иллюстрация идеи, что в общем случае в реализациях таблиц символов ключи используются для определенной структуризации данных в целях ускорения поиска. Структура может допускать быстрые реализации и ряда других операций, но при этом следует учитывать затраты на поддержку структуры, которые могут приводить к замедлению других операций. Мы встретимся с многими примерами упомянутого явления. Например, в приложении, где функция *sort* требуется часто, нужно было бы выбрать упорядоченное представление (с использованием массива или списка), поскольку такая структура таблицы делает реализацию функции *sort* тривиальной, в отличие от необходимости полной реализации сортировки. В приложении, в котором заведомо потребуется частое выполнение операции *select*, нужно было бы использовать представление с использованием упорядоченного массива, поскольку при такой структуре таблицы затрачиваемое на выполнение упомянутой операции время постоянно. И напротив, время выполнения операции *select* в связном списке линейно зависит от количества элементов, даже если список упорядочен.

Чтобы подробнее проанализировать последовательный поиск произвольных ключей, начнем с рассмотрения затрат на вставку новых ключей и отдельно рассмотрим случаи *успешного* и *неуспешного* поиска. Первый часто называют *попаданием при поиске*, а второй — *промахом при поиске*. Нас интересуют затраты как при попаданиях, так и при промахах в среднем и худшем случаях. Строго говоря, в реализации с использованием упорядоченного массива (см. программу 12.5) для каждого исследуемого элемента используются две операции сравнения ( $==$  и  $<$ ). При анализе в главах 12—16 каждую из таких пар мы будем считать одной операцией сравнения, поскольку обычно для эффективного их объединения можно выполнить низкоуровневую оптимизацию.

**Лемма 12.2** *При последовательном поиске в таблице символов с  $N$  элементами для выявления попаданий при поиске требуется выполнение около  $N/2$  сравнений (в среднем).*

См. лемму 2.1. Доказательство применимо к массивам или связным спискам, упорядоченным или неупорядоченным.

**Лемма 12.3** *При последовательном поиске в таблице символов, содержащей  $N$  неупорядоченных элементов, используется постоянное количество шагов для выполнения вставок и  $N$  сравнений для выявления промахов при поиске (всегда).*

Эти утверждения справедливы для представлений с использованием как массивов, так и связных списков, в чем легко убедиться, ознакомившись с реализациями (см. упражнение 12.20 и программу 12.6).

**Лемма 12.4** Для вставки, обнаружения попаданий и промахов при последовательном поиске в таблице символов, содержащей  $N$  упорядоченных элементов, требуется выполнение приблизительно  $N/2$  операций сравнения (в среднем).

См. лемму 2.2. И вновь эти утверждения справедливы для представлений с использованием как массивов, так и связных списков, в чем несложно убедиться, ознакомившись с реализациями (см. программу 12.5 и упражнение 12.21).

Построение упорядоченных таблиц путем последовательной вставки, по существу, эквивалентно выполнению алгоритма сортировки вставками, который был описан в разделе 6.2. Общее время, необходимое для построения таблицы, связано квадратичной зависимостью с количеством элементов, поэтому вряд ли стоит использовать этот метод для построения больших таблиц. Однако при выполнении огромного количества операций *search* в небольшой таблице поддержка упорядоченности элементов вполне оправдана, поскольку в соответствии с леммами 12.3 и 12.4 этот подход может в два раза уменьшить время, затрачиваемое на обнаружение промахов при поиске. Если элементы с дублированными ключами не должны храниться в таблице, дополнительные затраты на поддержку упорядоченности таблицы не столь велики, как может казаться, поскольку вставка выполняется только после обнаружения промаха при поиске и, следовательно, время, затрачиваемое на вставку, пропорционально времени, затрачиваемому на поиск. С другой стороны, если элементы с дублированными ключами могут храниться в таблице, при использовании неупорядоченной таблицы время выполнения операции *insert* может оставаться постоянным. Использование неупорядоченной таблицы предпочтительнее для приложений, в которых выполняется огромное количество операций *insert* при сравнительно небольшом числе операций *search*.

Помимо учета этих различий, приходится, как обычно, идти на компромисс: для реализаций с использованием связных списков требуется дополнительный объем памяти для ссылок, в то время как для реализаций с использованием массивов необходимо заранее знать максимальный размер таблицы или же предусмотреть увеличение таблицы с течением времени (см. раздел 14.5). Кроме того, как упоминалось в разделе 12.5, использование связных списков обладает гибкостью, позволяющей эффективно реализовать другие операции типа *join* и *remove*.

Эти результаты во взаимосвязи с другими алгоритмами, освещенными далее в этой главе и главах 13 и 14, обобщены в табл. 12.1. В разделе 12.4 рассматривается *бинарный поиск*, при котором зависимость времени поиска от количества элементов уменьшается до  $\lg N$ , в связи с чем он широко используется при работе со статическими таблицами (когда вставки выполняются сравнительно редко).

### Таблица 12.1 Затраты на вставку и поиск в таблицах символов

Записи в этой таблице представляют приведенное к постоянному коэффициенту время выполнения как функцию от количества элементов в таблице  $N$  и размеров таблицы  $M$  (если он отличен от  $N$ ) для реализаций, в которых новые элементы можно вставлять независимо от наличия в таблице элементов с дублированными ключами. При реализации элементарных методов (первые четыре строки) время выполнения некоторых операций постоянно, а время выполнения других линейно зависит от количества элементов или размеров таблицы; более сложные методы гарантируют логарифмическую зависимость времени от количества элементов или же постоянство времени выполнения большинства или всех операций. Значения  $M \lg N$  в столбце операции выбора представляют затраты на сортировку элементов — линейная

зависимость времени выполнения операции *select* для неупорядоченного набора элементов возможна лишь теоретически, но не на практике (см. раздел 7.8). Значения, помеченные звездочкой, относятся к крайне нежелательным худшим случаям.

	худший случай			средний случай		
	вставка	поиск	выбор	вставка	попадание при поиске	промах при поиске
массив, индексированный по ключам	1	1	$M$	1	1	1
упорядоченный массив	$N$	$N$	1	$N/2$	$N/2$	$N/2$
упорядоченный связный список	$N$	$N$	$N$	$N/2$	$N/2$	$N/2$
неупорядоченный массив	1	$N$	$N \lg N$	1	$N/2$	$N$
неупорядоченный связный список	1	$N$	$N \lg N$	1	$N/2$	$N$
бинарный поиск	$N$	$\lg N$	1	$N/2$	$\lg N$	$\lg N$
дерево бинарного поиска	$N$	$N$	$N$	$\lg N$	$\lg N$	$\lg N$
дерево типа "красное—черное"	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$
рандомизованное дерево	$N^*$	$N^*$	$N^*$	$\lg N$	$\lg N$	$\lg N$
хеширование	1	$N^*$	$N \lg N$	1	1	1

В разделах 12.5—12.9 рассматриваются *деревья бинарного поиска*, которые обеспечивают определенную гибкость в том, что время поиска и вставки становится пропорциональным  $\lg N$ , но только в среднем. В главе 13 будут исследоваться *деревья типа "красное—черное" (red-black tree)* и *рандомизованные деревья бинарного поиска*, которые, соответственно, гарантируют логарифмическую зависимость времени от количества элементов либо существенно увеличивают вероятность этого. В главе 14 изучаются вопросы *хеширования*, которое в среднем обеспечивает постоянство времени поиска и вставки, но не обеспечивает эффективную поддержку операции *sort* и ряда других операций. В главе 15 будут рассматриваться методы поразрядного поиска, которые аналогичны методам поразрядной сортировки, описанным в главе 10; в главе 16 исследуются методы, применимые к файлам, которые хранятся внешне.

## Упражнения

- ▷ **12.16** Добавьте операцию *remove* в приведенную реализацию таблицы символов с использованием упорядоченного массива (программа 12.5).
- ▷ **12.17** Создайте функции **searchinsert** для приведенных реализаций таблиц символов с использованием списка (программа 12.6) и массива (программа 12.5). Функции должны выполнять в таблице символов поиск элемента с заданным ключом, а затем вставлять элемент, если таковой в таблице отсутствует.

**12.18** Создайте операцию *select* для приведенной реализации таблицы символов с использованием списка (программа 12.6).

**12.19** Укажите количество операций сравнения, необходимое для помещения ключей **E A S Y Q U E S T I O N** в первоначально пустую таблицу с использованием АД, которые реализованы в соответствии с одним из четырех элементарных подходов: упорядоченный или неупорядоченный массив или список. Примите, что для каждого ключа выполняется поиск, а затем, в случае его отсутствия в таблице, выполняется вставка, как в упражнении 12.17.

**12.20** Реализуйте операции *construct*, *search* и *insert* для интерфейса таблицы символов программы 12.2, используя для представления таблицы символов неупорядоченный массив. Характеристики производительности программы должны соответствовать табл. 12.1.

- **12.21** Реализуйте операции *construct*, *search* и *insert* для интерфейса таблицы символов программы 12.2, используя для представления таблицы символов упорядоченный связный список. Характеристики производительности программы должны соответствовать табл. 12.1.
- **12.22** Измените представленные реализации таблицы символов с использованием списка (программа 12.6), чтобы они поддерживали дескрипторы элемента клиента (см. упражнение 12.7); добавьте деструктор, конструктор копирования и перегруженную операцию присваивания (см. упражнение 12.6); добавьте операции *remove* и *join*; создайте программу-драйвер, тестирующую созданные интерфейс и реализацию АДТ первого класса таблицы символов.

**12.23** Создайте программу-драйвер проверки производительности, в которой функция *insert* используется для заполнения таблицы символов, а функции *select* и *remove* — для ее освобождения; эти операции должны повторяться несколько раз применительно к произвольным последовательностям ключей различной длины, от малой до большой. Программа должна замерять время, затрачиваемое на каждое выполнение, и выводить средние значения в виде текста или графика.

**12.24** Создайте программу-драйвер проверки производительности, в которой бы функция *insert* использовалась для заполнения таблицы символов, а функция *search* обеспечивала, чтобы при поиске каждого элемента в таблице в среднем происходило 10 попаданий и примерно столько же промахов; эти операции должны повторяться несколько раз применительно к произвольной последовательности ключей различной длины, от малой до большой. Программа должна замерять время, затрачиваемое на каждое выполнение, и выводить средние значения в виде текста или графика.

**12.25** Создайте программу-драйвер, в которой функции из программы 12.2 для интерфейса таблицы символов используются применительно к трудному или крайне нежелательному случаю, который может возникнуть в реальных приложениях. Простыми примерами являются уже упорядоченные файлы, файлы, упорядоченные в обратном порядке, файлы с одинаковыми ключами и файлы, которые состоят только из двух различных значений.

- **12.26** Какую реализацию таблицы символов следовало бы использовать для приложения, в котором в произвольном порядке выполняется  $10^2$  операций *insert*,  $10^3$  операций *search* и  $10^4$  операций *select*? Обоснуйте ответ.
- **12.27** (В действительности это упражнение состоит из пяти упражнений). Дайте ответ на вопрос упражнения 12.26 для пяти других вариантов сочетания операций и частоты их использования.

**12.28** Алгоритм *самоорганизующегося* поиска — это алгоритм, который изменяет порядок элементов так, чтобы часто запрашиваемые элементы, скорее всего, находились в начале поиска. Измените реализацию операции *search* для упражнения 12.20, чтобы при каждом попадании при поиске она выполняла следующее действие: помещала найденный элемент в начало списка, перемещая на одну позицию вправо все элементы, расположенные между началом списка и освободившейся позицией. Эта процедура называется эвристическим *перемещением вперед*.

▷ **12.29** Укажите порядок ключей после того, как элементы с ключами **E A S Y Q U E S T I O N** помещаются в первоначально пустую таблицу, когда после выполнения операции *search* вызывается *insert* по причине отсутствия элемента, причем применяется эвристический самоорганизующийся поиск с перемещением вперед (см. упражнение 12.28).

**12.30** Создайте программу-драйвер для методов самоорганизующегося поиска, в которой функция *insert* используется для заполнения таблицы символов  $N$  ключами, а затем выполняется  $10N$  поисков для обнаружения элементов в соответствии с заранее определенным распределением вероятностей.

**12.31** Воспользуйтесь решением упражнения 12.30 для сравнения времени выполнения реализации из упражнения 12.20 и времени выполнения реализации из упражнения 12.28 для  $N = 10, 100$  и  $1000$ , используя распределение вероятностей, при котором вероятность успешного выполнения операции *search* для  $i$ -го наибольшего ключа равна  $1/2^i$  при  $1 \leq i \leq N$ .

**12.32** Выполните упражнение 12.31 для распределения вероятностей, при котором вероятность успешного выполнения операции *search* для  $i$ -го наибольшего ключа определяется соотношением  $N_N/i$  при  $1 \leq i \leq N$ . Это распределение называется *законом Zipфа*.

**12.33** Сравните эвристическое перемещение вперед с оптимальной организацией для распределений, указанных в упражнениях 12.31 и 12.32, поддерживающей размещение ключей в порядке возрастания (в порядке уменьшения ожидаемой частоты обращения к ним). То есть, в упражнении 12.31 вместо решения из упражнения 12.20 воспользуйтесь программой 12.5.

## 12.4 Бинарный поиск

В реализации последовательного поиска на базе массивов общее время поиска на больших наборах элементов можно значительно уменьшить, используя процедуру поиска, основанную на стандартном подходе "разделяй и властвуй" (см. раздел 5.2). Для этого набор элементов необходимо разделить на две части, определить, к какой из двух частей принадлежит ключ поиска, а затем сосредоточить свои усилия именно на этой части. Рациональный способ разделения наборов элементов на части состоит в поддержке элементов в отсортированном виде с последующим использованием индексов в отсортированном массиве для определения части массива, над которой будет выполняться дальнейшая работа. Такая технология поиска называется *бинарным поиском*. Программа 12.7 представляет рекурсивную реализацию этой основополагающей стратегии. В программе 2.2 показана нерекурсивная реализация метода, при которой никаких стеков не требуется,

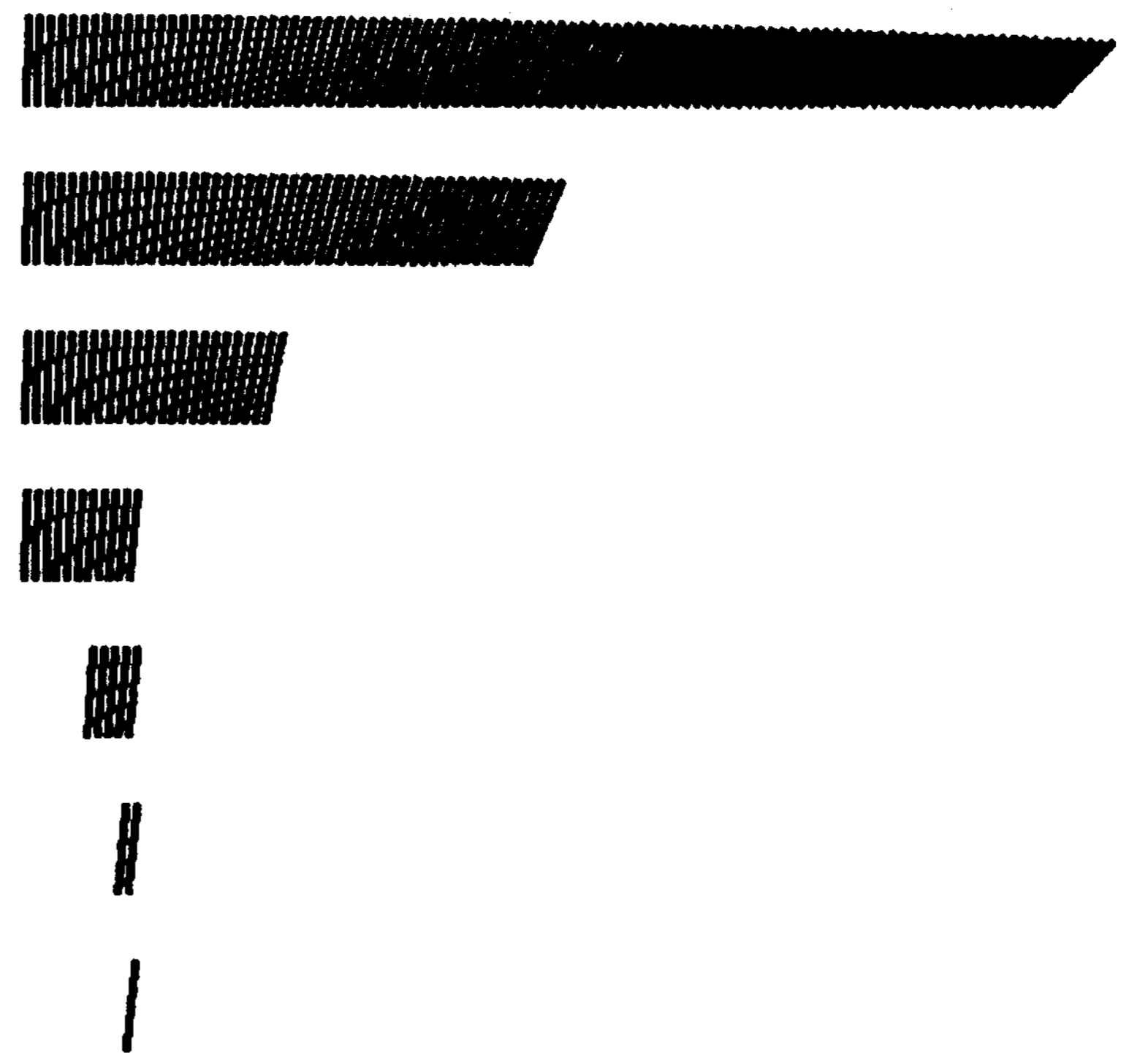
A A A C E E E G H I L M N P R  
 H I L M N P R  
 H I L  
 L

**РИСУНОК 12.1 БИНАРНЫЙ ПОИСК**

При бинарном поиске для нахождения искомого ключа  $L$  в этом примере файла используются только три итерации. В первом вызове алгоритм сравнивает  $L$  с ключом в середине файла —  $G$ . Поскольку  $L$  больше этого ключа, в ходе следующей итерации исследуется правая половина файла. Затем, поскольку  $L$  меньше  $M$ , находящегося в середине правой половины, в ходе третьей итерации исследуется подфайл, состоящий из трех элементов, в который входят ключи  $H, I$  и  $L$ . После выполнения еще одной итерации размер подфайла становится равным 1 и алгоритм находит ключ  $L$ .

### РИСУНОК 12.2 БИНАРНЫЙ ПОИСК

При бинарном поиске требуется только семь итераций для нахождения записи в файле, который состоит из 200 элементов. Размеры подфайлов описываются последовательностью 200, 99, 49, 24, 11, 5, 2, 1; несложно заметить, что каждая из исследуемых частей несколько меньше предыдущей.



поскольку рекурсивная функция в программе 12.7 завершается рекурсивным вызовом.

На рис. 12.1 показаны подфайлы, исследуемые в ходе бинарного поиска в небольшой таблице; на рис. 12.2 приведен большой пример. В ходе каждой итерации отбрасывается несколько больше половины таблицы, поэтому количество требуемых итераций мало.

**Лемма 12.5** При бинарном поиске никогда не используется более чем  $\lfloor \lg N \rfloor + 1$  сравнений (для выявления попадания или промаха).

См. лемму 2.3. Интересно отметить, что максимальное количество сравнений, используемых для бинарного поиска в таблице, размер которой равен  $N$ , в точности равен количеству бит в двоичном представлении числа  $N$ , поскольку операция сдвига одного бита вправо преобразует двоичное представление  $N$  в двоичное представление числа  $\lfloor N/2 \rfloor$  (см. рис. 2.6).

Поддержка таблицы в отсортированном виде, как это делается при сортировке вставками, приводит к тому, что время выполнения становится квадратичной функцией от количества операций *insert*, но эта стоимость может оказаться приемлемой или ею даже можно пренебречь, если количество операций *search* очень велико. В типичной ситуации, когда все элементы (или большая их часть) доступны до начала поиска, можно построить (*construct*) таблицу символов с помощью конструктора, который принимает массив в качестве аргумента и использует один из стандартных методов сортировки, описанных в главе 6 и последующих главах, для сортировки таблицы во время инициализации. После этого обновление таблицы может выполняться различными способами. Например, можно поддерживать порядок во время вставки, как это делается в программе 12.5 (см. также упражнение 12.21), либо объединить вставляемые элементы в пакет, выполнить сортировку и объединить с существующей таблицей (как описано в упражнении 8.1). Всякое обновление может быть связано со вставкой элемента, ключ которого меньше ключа какого-либо из элементов таблицы, поэтому может потребоваться перемещение любого элемента с целью освобождения места. Упомянутая вероятность высокой стоимости обновления таблицы — наибольший недостаток использования бинарного поиска. С другой стороны, существует ог-

ромное число приложений, в которых статическая таблица может быть заранее отсортированной и в этом случае, благодаря быстрому доступу, обеспечиваемому такими реализациями, как программа 12.7, бинарный поиск является наиболее предпочтительным методом.

Если новые элементы требуется вставлять динамически, может показаться, что для этого нужна связная структура. Однако сам по себе связный список не позволяет создавать эффективную реализацию, поскольку эффективность бинарного поиска зависит от возможности быстро попасть в середину любого подмассива через индексирование, а единственный способ попасть в середину связного списка — это отслеживание ссылок. Для суммирования эффективности бинарного поиска и гибкости связных структур требуются более сложные структуры данных, которые мы начнем исследовать вскоре.

Если в таблице присутствуют дублированные ключи, бинарный поиск можно расширить до поддержки операций на таблице символов для подсчета количества элементов с данным ключом или возврата их в виде группы. Несколько элементов, ключи которых совпадают с искомым, образуют в таблице непрерывный блок (поскольку таблица упорядочена), и в программе 12.7 успешный поиск завершится где-то внутри этого блока. Если приложению требуется доступ ко всем элементам подобного рода, в программу можно добавить код для выполнения просмотра в обоих направлениях от точки завершения поиска, а также код для возврата двух индексов, ограничивающих элементы, ключи которых равны искомому ключу. В этом случае время выполнения поиска окажется пропорциональным  $\lg N$  плюс количество найденных элементов. Аналогичный подход используется для решения более общей задачи *поиска в диапазоне*, которая состоит в нахождении всех элементов, ключи которых попадают в указанный интервал. Подобные расширения базового набора операций с таблицами символов рассматриваются в части 6.

### Программа 12.7 Бинарный поиск (в таблице символов, основанной на массиве)

Эта реализация функции поиска (`search`) использует процедуру рекурсивного бинарного поиска. Для выяснения, присутствует ли заданный ключ `v` в отсортированном массиве, функция сначала сравнивает `v` с элементом, находящимся в средней позиции. Если `v` меньше, он должен быть в первой половине массива, а если больше — то во второй.

Массив должен быть отсортированным. Эта функция может рассматриваться в качестве замены функции `search` из программы 12.5, которая обеспечивает динамическую сортировку во время вставки. Можно было бы также добавить конструктор таблицы символов, который получал бы массив в качестве аргумента, а затем строил бы таблицу символов на основе элементов входного массива и сортировал бы ее для целей поиска с применением одной из стандартных процедур сортировки.

`private:`

```
Item searchR(int l, int r, Key v)
{ if (l > r) return nullItem;
  int m = (l+r)/2;
  if (v == st[m].key()) return st[m];
  if (l == r) return nullItem;
  if (v < st[m].key())
    return searchR(l, m-1, v);
  else return searchR(m+1, r, v);
}
```

`public:`

```
Item search(Key v)
{ return searchR(0, N-1, v); }
```

Последовательность сравнений, выполняемых алгоритмом бинарного поиска, predetermined: конкретная последовательность используется в зависимости от значения искомого ключа и значения  $N$ . Сравнения могут быть описаны в виде структуры бинарного дерева, подобной приведенной на рис. 12.3. Это дерево аналогично использованному в главе 8 для описания размеров подфайлов во время сортировки слиянием (рис. 8.3). В бинарном поиске используется один путь в дереве, тогда как при сортировке слиянием — все пути. Это дерево является статическим и неявным; в разделе 12.5 будут рассматриваться алгоритмы, в которых для выполнения поиска используется динамическая, явно построенная структура бинарного дерева.

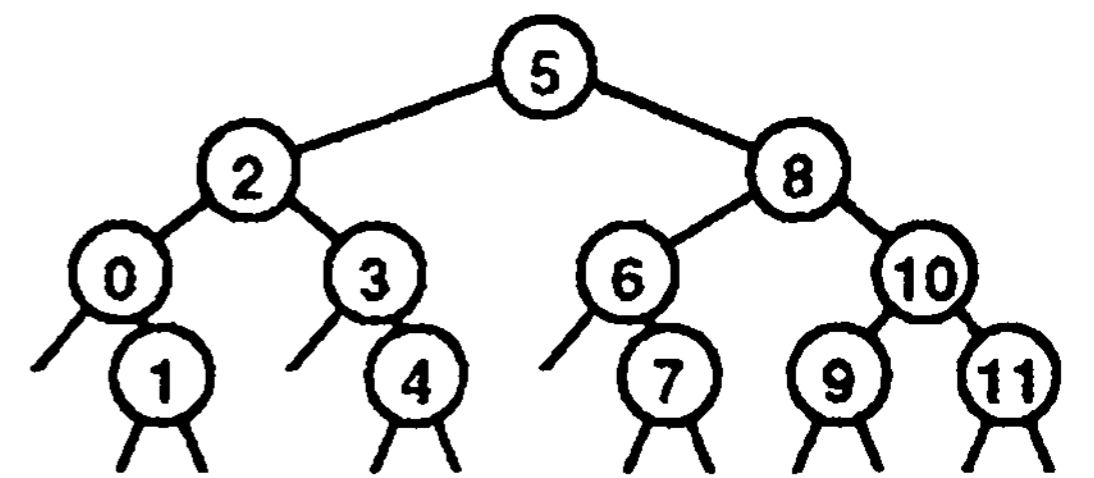
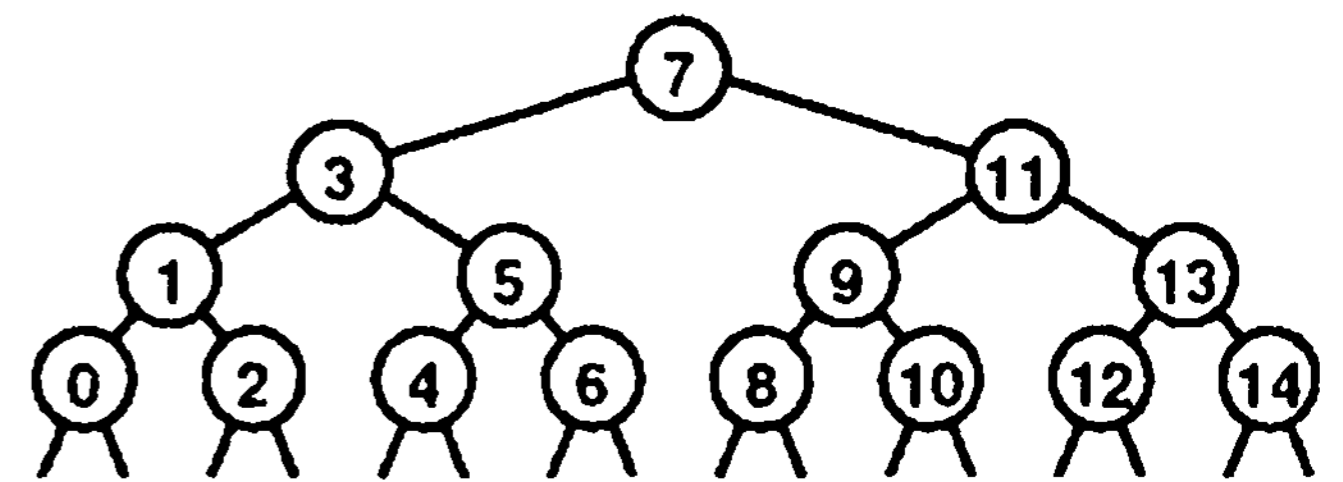
На верхней диаграмме показан поиск в файле, состоящем из 15 элементов, проиндексированных от 0 до 14. Мы просматриваем средний элемент (индекс 7), затем (рекурсивно) просматриваем левое поддерево, если искомый элемент меньше него, и правое — если он больше. Каждый поиск соответствует пути, проходящему по дереву сверху вниз; например, поиск элемента, который располагается между элементами 10 и 11, потребовал бы просмотра элементов 7, 11, 9, 10. Для файлов с размерами, которые не являются на 1 меньше степени числа 2, последовательность не столь симметрична, что не сложно заметить на нижней диаграмме, нарисованной для 12 элементов.

Одно из возможных усовершенствований бинарного поиска — более точное предположение о размещении ключа поиска в текущем интервале (вместо слепого сравнения его со средним элементом на каждом шаге). Эта тактика имитирует способ поиска имени в телефонном справочнике или слова в словаре: если искомая запись начинается с буквы, находящейся в начале алфавита, поиск выполняется вблизи начала книги, но если она начинается с буквы, находящейся в конце алфавита, поиск выполняется в конце книги. Для реализации данного метода, называемого *интерполяционным поиском* (*interpolation search*), программу 12.7 потребуется изменить следующим образом: оператор

$$m = (l+r)/2$$

заменяется на оператор

$$m = l + (v - [l].key()) * (r - l) / (a[r].key() - a[l].key());$$



**РИСУНОК 12.3**  
**ПОСЛЕДОВАТЕЛЬНОСТЬ**  
**СРАВНЕНИЙ ПРИ БИНАРНОМ**  
**ПОИСКЕ**

*На этих диаграммах деревьяев типа "разделяй и властвуй" показана последовательность индексов для сравнений при бинарном поиске.*

*Последовательности зависят только от размера исходного файла, а не значений ключей в файле. Эти деревья несколько отличаются от деревьев, соответствующих сортировке слиянием и аналогичным алгоритмам (рис. 5.6 и 8.3), поскольку расположенный в корне элемент в поддеревья не включается.*



Для обоснования изменения отметим, что выражение  $(l + r)/2$  равнозначно выражению  $l + \frac{1}{2}(r - l)$ : мы вычисляем середину интервала, добавляя к левой граничной точке половину размера интервала. Использование интерполяционного поиска сводится к замене в этой формуле коэффициента  $\frac{1}{2}$  ожидаемой позицией ключа — а именно  $(v - k_l) / (k_r - k_l)$ , где  $k_l$  и  $k_r$  означают значения  $a[l].key()$  и  $a[r].key()$ , соответственно. При этом предполагается, что значения ключей являются числовыми и равномерно распределенными.

Можно показать, что при интерполяционном поиске в файлах с произвольными ключами для каждого поиска (попадания или промаха) используется менее  $\lg \lg N$  сравнений. Доказательство этого утверждения выходит далеко за рамки этой книги. Эта функция растет очень медленно и на практике ее можно считать постоянной: если  $N$  равно 1 миллиарду, то  $\lg \lg N < 5$ . Таким образом, любой элемент можно найти, выполнив только несколько обращений (в среднем) — это существенное достижение по сравнению с бинарным поиском. Для ключей, которые распределены не вполне произвольно, производительность интерполяционного поиска еще выше. Действительно, граничным случаем является метод поиска с использованием индексирования по ключам, описанный в разделе 12.2.

Однако интерполяционный поиск в значительной степени основывается на предположении, что ключи распределены во всем интервале более-менее равномерно — в противном случае, что обычно и имеет место на практике, метод окажется неэффективным. Кроме того, для его реализации требуются дополнительные вычисления. Для небольших значений  $N$  стоимость обычного бинарного поиска ( $\lg N$ ) достаточно близка к стоимости интерполяционного поиска ( $\lg \lg N$ ), и поэтому вряд ли стоит использовать последний метод. С другой стороны, интерполяционный поиск определенно заслуживает внимания при работе с большими файлами, в приложениях, в которых сравнения выполняются особенно часто, и при использовании внешних методов, сопряженных с большими затратами на доступ.

## Упражнения

- ▷ 12.34 Реализуйте нерекурсивную функцию бинарного поиска (см. программу 12.7).
- 12.35 Нарисуйте деревья, которые соответствуют рис. 12.3 для  $N = 17$  и  $N = 24$ .
- 12.36 Найдите значения  $N$ , для которых бинарный поиск в таблице символов размером  $N$  становится в 10, 100 и 1000 раз быстрее последовательного поиска. Предскажите значения аналитически и проверьте их экспериментально.
- 12.37 Предположите, что вставки в динамическую таблицу символов размера  $N$  реализуются методом сортировки вставками, но для выполнения операции *search* используется бинарный поиск. Предположите, что поиск выполняется в 1000 раз чаще вставки. Определите в процентах долю времени, затрачиваемую на вставку, для  $N = 10^3$ ,  $10^4$ ,  $10^5$  и  $10^6$ .
- 12.38 Разработайте реализацию таблицы символов, в которой используется бинарный поиск и "ленивая" вставка, а также поддерживаются операции *construct*, *count*, *search*, *insert* и *sort*, прибегнув к следующей стратегии. Храните большой отсорти-

рованный массив для основной таблицы символов и неупорядоченный массив для недавно вставленных элементов. При вызове функции `search` отсортируйте недавно вставленные элементы (если таковые существуют), объедините их с основной таблицей, а затем воспользуйтесь бинарным поиском.

**12.39** Добавьте "ленивое" удаление в реализацию, созданную в упражнении 12.38.

**12.40** Ответьте на вопрос упражнения 12.37 для реализации из упражнения 12.38.

- **12.41** Реализуйте функцию, аналогичную бинарному поиску (программа 12.7), которая возвращает количество элементов в таблице символов с ключами, равными данному.

**12.42** Создайте программу, которая при заданном значении  $N$  создает последовательность  $N$  макроинструкций, проиндексированных от 0 до  $N-1$ , в форме `compare(l, h)`, где  $i$ -ая инструкция в списке означает "сравнить ключ поиска со значением индекса  $i$  в таблице; затем сообщить о попадании при поиске, если они равны, выполнить  $l$ -ую инструкцию, если он меньше, и  $h$ -ую инструкцию, если он больше" (индекс 0 зарезервируйте на случай промаха при поиске). Необходимо, чтобы последовательность обладала тем свойством, что для любого поиска должно выполняться столько же операций сравнения, как и при бинарном поиске в этом же наборе данных.

- **12.43** Расширьте макрос, созданный в упражнении 12.42, чтобы программа создавала машинный код, выполняющий бинарный поиск в таблице размером  $N$  при наименьшем возможном количестве машинных инструкций на одно сравнение.

**12.44** Предположите, что  $a[i] = 10^i$  для  $1 < N$ . Сколько позиций в таблице исследуются интерполяционным поиском во время неуспешного поиска  $2k-1$ ?

- **12.45** Найдите значения  $N$ , для которых интерполяционный поиск в таблице символов размером  $N$  выполняется в 1, 2 и 10 раз быстрее бинарного поиска, при условии произвольности ключей. Предскажите значения аналитически и проверьте их экспериментально.

## 12.5 Деревья бинарного поиска

Для решения проблемы излишне высоких затрат на вставку в качестве основы для реализации таблицы символов используется явная древовидная структура.

Лежащая в основе структура данных позволяет разрабатывать алгоритмы с высокой средней производительностью выполнения операций *search*, *insert*, *select* и *sort* в таблицах символов. Этот метод используется во множестве приложений и в компьютерных науках считается одним из наиболее фундаментальных.

В главе 5 уже рассматривались деревья, тем не менее, полезно еще раз вспомнить терминологию. Определяющее свойство *дерева (tree)* заключается в том, что каждый узел указывается только одним другим узлом, называемым *родительским (parent)*. Определяющее свойство *бинарного дерева* — наличие у каждого узла левой и правой связей. Связи могут указывать на другие двоичные деревья или на *внешние (external) узлы*, которые не имеют связей. Узлы с двумя связями называются также *внутренними (internal) узлами*. Для выполнения поиска каждый внутренний узел имеет также элемент со значением ключа, а связи с внешними узлами называются *нулевыми (null) связями*. Значения ключей во внутренних узлах сравниваются в ключом поиска и управляют протеканием поиска.

**Лемма 12.2** *Дерево бинарного поиска (BST) — это бинарное дерево, с каждым из внутренних узлов которого связан ключ, причем ключ в любом узле больше (или равен) ключам и во всех узлах левого поддерева этого узла и меньше (или равен) ключам во всех узлах правого поддерева этого узла.*

В программе 12.8 BST-деревья используются для реализации операций *search*, *insert*, *construct* и *count*. В первой части реализации узлы в BST-дереве определяются как содержащие элемент (с ключом), левую и правую связи. Кроме того, для обеспечения энергичной реализации операции *count* программа поддерживает поле, содержащее количество узлов в дереве. Левая связь указывает на BST-дерево с элементами с меньшими (или равными) ключами, а правая — на BST-дерево с элементами с большими (или равными) ключами.

### Программа 12.8 Таблица символов на базе дерева бинарного поиска

В этой реализации функции **search** и **insert** используют приватные рекурсивные функции **searchR** и **insertR**, которые непосредственно отражают рекурсивное определение BST-дерева. Обратите внимание на использование аргумента ссылки в функции **insertR** (см. текст). Ссылка **head** указывает на корень дерева.

```
template <class Item, class Key>
class ST
{
private:
    struct node
    { Item item; node *l, *r;
      node(Item x)
        { item = x; l = 0; r = 0; }
    };
    typedef node *link;
    link head;
    Item nullItem;
    Item searchR(link h, Key v)
    { if (h == 0) return nullItem;
      Key t = h->item.key();
      if (v == t) return h->item;
      if (v < t) return searchR(h->l, v);
      else return searchR(h->r, v);
    }
    void insertR(link& h, Item x)
    { if (h == 0) { h = new node(x); return; }
      if (x.key() < h->item.key())
        insertR(h->l, x);
      else insertR(h->r, x);
    }
public:
    ST(int maxN)
    { head = 0; }
    Item search(Key v)
    { return searchR(head, v); }
    void insert(Item x)
    { insertR(head, x); }
};
```

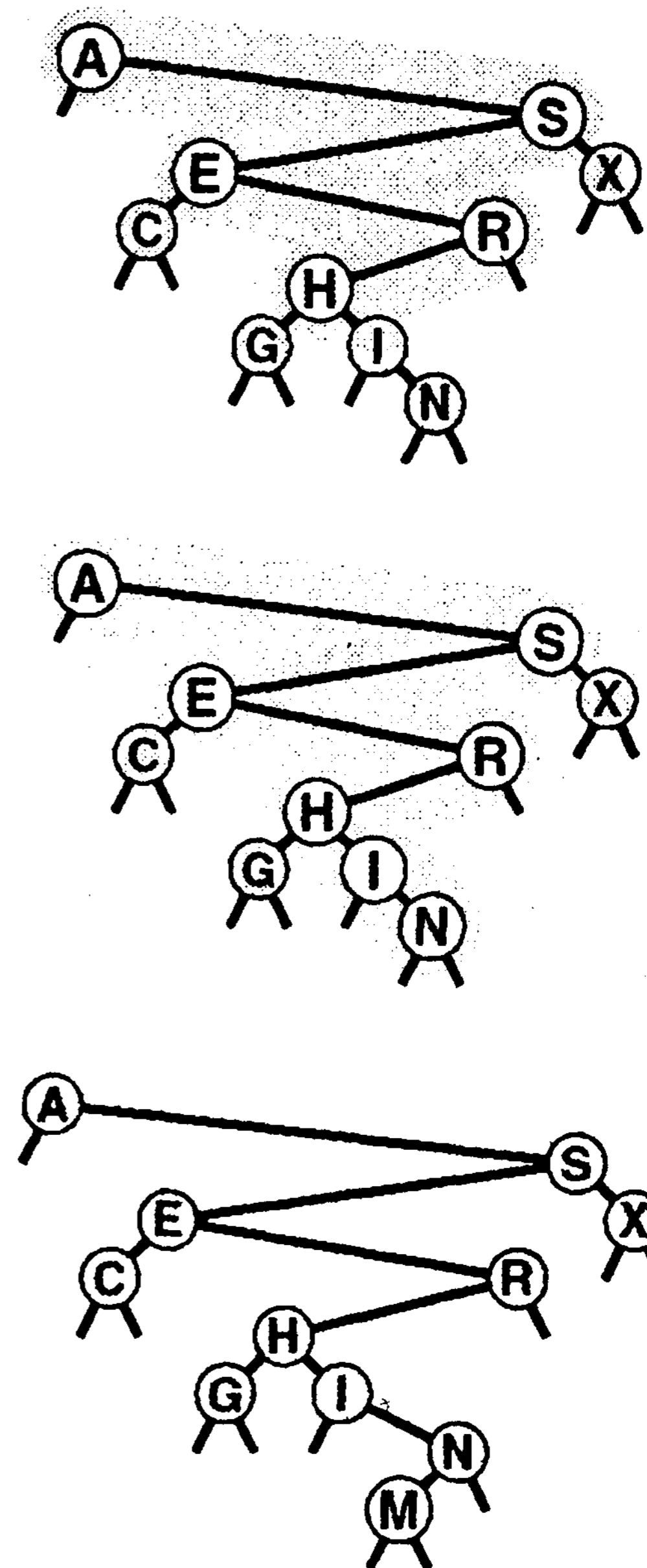
При наличии этой структуры рекурсивный алгоритм поиска ключа в BST-дереве становится очевидным: если дерево пусто, имеет место промах при поиске; если ключ поиска равен ключу в корне, имеет место попадание при поиске. В противном случае выполняется поиск (рекурсивно) в соответствующем поддереве. Функция `searchR` в программе 12.8 непосредственно реализует этот алгоритм. Мы вызываем рекурсивную подпрограмму, которая принимает дерево в качестве первого аргумента и ключ в качестве второго, начиная с корня дерева и искомого ключа. На каждом шаге гарантируется, что никакие части дерева, кроме текущего поддерева, не могут содержать элементы с искомым ключом.

Подобно тому как при бинарном поиске на каждой итерации размер интервала уменьшается чуть более чем в два раза, текущее поддерево в дереве бинарного поиска меньше предшествующего (в идеальном случае приблизительно вдвое). Процедура завершается либо в случае нахождения элемента с искомым ключом (попадание при поиске), либо когда текущее поддерево становится пустым (промах при поиске).

На диаграмме в верхней части рис. 12.4 проиллюстрирован процесс поиска для примера дерева. Начиная с верхней части, процедура поиска в каждом узле приводит к рекурсивному вызову для одного из дочерних узлов этого узла; таким образом, поиск определяет путь по дереву. В случае попадания при поиске путь завершается в узле, содержащем ключ, а в случае промаха путь завершается во внешнем узле, как показано на средней диаграмме на рис. 12.4.

В программе 12.8 используется 0 связей для представления внешних узлов и приватные данные — член `head`, который является ссылкой на корень дерева. Для конструирования пустого BST-дерева значение `head` устанавливается равным 0. Можно было бы также использовать фиктивный узел в корне и еще один для представления всех внешних узлов в различных комбинациях, подобных рассмотренным для связных списков в табл. 3.1 (см. упражнение 12.53).

Функция поиска в программе 12.8 столь же проста, как и обычный бинарный поиск; существенная особенность BST-деревьев заключается в том, что операцию `insert` легко реализовать в виде операции `search`. Рекурсивная функция `insertR` для вставки нового элемента в BST-дерево следует логике, аналогичной использованной при разработке функции `searchR`, и использует ссылочный аргумент `h` для



**РИСУНОК 12.4 ПОИСК И ВСТАВКА В ДЕРЕВО БИНАРНОГО ПОИСКА**

*В процессе успешного поиска  $H$  в этом примере дерева (вверху) мы перемещаемся вправо от корня (поскольку  $H$  больше чем  $A$ ), затем влево в правом поддереве (поскольку  $H$  меньше чем  $S$ ) и т.д., продолжая перемещаться вниз по дереву, пока не встретится  $H$ . В процессе неуспешного поиска  $M$  в этом примере дерева (в центре) мы перемещаемся вправо от корня (поскольку  $M$  больше чем  $A$ ), затем влево в правом поддереве корня (поскольку  $M$  меньше чем  $S$ ) и т.д., продолжая перемещаться вниз по дереву, пока не встретится внешняя связь слева от  $N$  в нижней части диаграммы. Для вставки  $M$  после обнаружения промаха при поиске достаточно просто заменить связь, которая прерывает поиск, связью с  $M$  (внизу).*

построения дерева: если дерево пусто, **h** устанавливается равным ссылке на новый узел, содержащий элемент; если ключ поиска меньше ключа в корне, элемент вставляется в левое поддерево, в противном случае элемент вставляется в правое поддерево. То есть, аргумент ссылки изменяется только в последнем рекурсивном вызове, когда вставляется новый элемент. В разделе 12.8 и в главе 13 будут изучаться более сложные древовидные структуры, которые естественным образом представляются с помощью той же рекурсивной схемы, но которые чаще изменяют ссылочный аргумент.

На рис. 12.5 и 12.6 показан пример создания BST-дерева путем вставки последовательности ключей в первоначально пустое дерево. Новые узлы присоединяются к нулевым связям в нижней части дерева, а в остальном структура дерева никак не изменяется. Поскольку каждый узел имеет две связи, дерево растет скорее в ширину, нежели в высоту.

При использовании BST-деревьев реализация функции *sort* требует незначительного объема дополнительной работы. Построение BST-дерева сводится к сортировке элементов, поскольку при соответствующем рассмотрении BST-дерево представляет отсортированный файл. На приведенных выше рисунках ключи отображаются на странице слева направо (если не обращать внимания на их расположение по высоте и связи). Программа работает только со связями, но простой поперечный обход, по определению, обеспечивает выполнение этой задачи, что демонстрируется рекурсивной реализацией функции **showR** в программе 12.9. Для отображения элементов в BST-дереве в порядке их ключей мы отображаем элементы в левом поддерево в порядке их ключей (рекурсивно), затем корень, и далее элементы в правом поддерево в порядке их ключей (рекурсивно).

### Программа 12.9 Сортировка с помощью BST-дерева

При поперечном обходе BST-дерева элементы посещаются в порядке следования их ключей. В этой реализации функция-член **show** элемента **item** используется для вывода элементов в порядке следования их ключей.

```
private:
    void showR(link h, ostream& os)
    {
        if (h == 0) return;
        showR(h->l, os);
        h->item.show(os);
        showR(h->r, os);
    }
public:
    void show(ostream& os)
    { showR(head, os); }
```

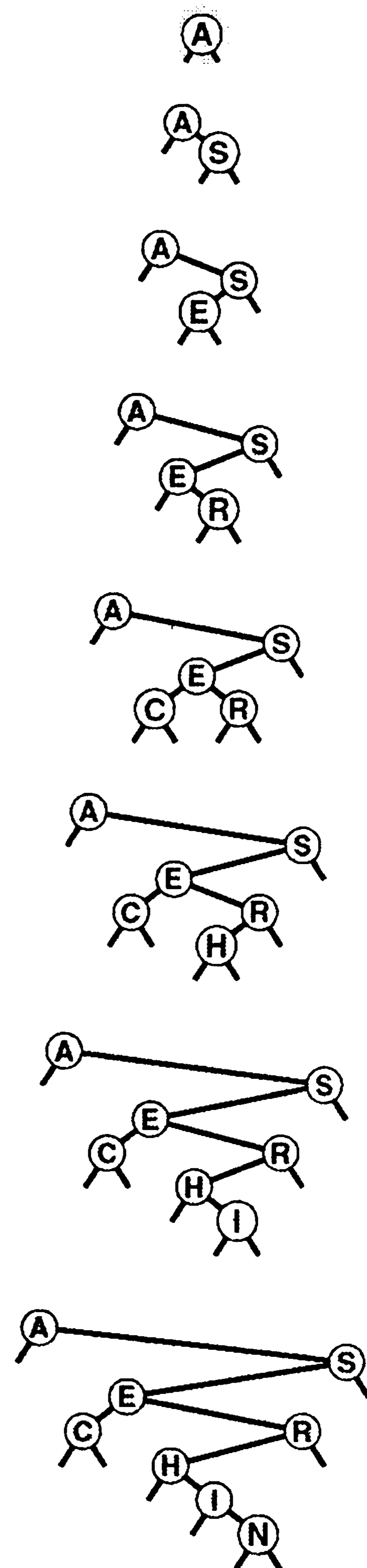


РИСУНОК 12.5 СОЗДАНИЕ ДЕРЕВА БИНАРНОГО ПОИСКА

*Эта последовательность отражает результат вставки ключей A S E R C H I N в первоначально пустое BST-дерево.*

*Каждая вставка следует за промахом при поиске в нижней части дерева.*

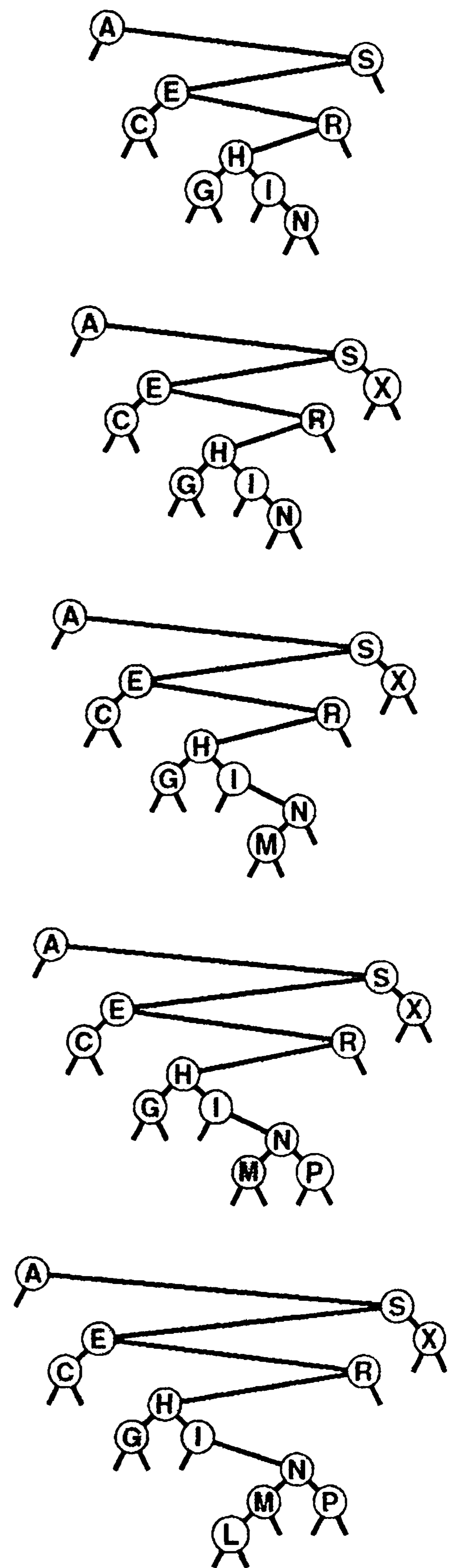
Как упоминалось в разделе 12.1, периодически, в случае необходимости систематического посещения каждого из элементов таблицы символов, мы будем обращаться к общей операции *visit* для таблиц символов. Применительно к BST-деревьям можно посетить элементы в порядке следования их ключей, заменив в только что приведенном описании операцию "show" операцией "visit" и, возможно, обеспечив передачу в функцию посещения элемента в качестве параметра (см. раздел 5.6).

Нерекурсивный подход при обдумывании реализации поиска и вставки в BST-деревьях несомненно представляет интерес. При нерекурсивной реализации процесс поиска состоит из цикла, в котором искомый ключ сравнивается с ключом в корне, затем выполняется перемещение влево, если ключ поиска меньше, и вправо — если он больше ключа в корне. Вставка состоит из обнаружения промаха при поиске (завершающегося на пустой связи) и последующей замены пустой связи указателем на новый узел. Этот процесс соответствует явному манипулированию связями, расположенными вдоль пути вниз по дереву (см. рис. 12.4). В частности, чтобы иметь возможность вставить новый узел в нижней части дерева, необходимо сохранять связь с родительским узлом текущего узла, как имеет место в реализации в программе 12.10. Как обычно, рекурсивная и нерекурсивная версии, по существу, эквивалентны, причем понимание обоих подходов способствует углублению наших представлений об алгоритмах и структурах данных.

### Программа 12.10 Вставка в BST-дерево (нерекурсивная)

Вставка элемента в BST-дерево эквивалентна выполнению неуспешного поиска этого элемента и последующего присоединения нового узла элемента вместо нулевой связи в месте завершения поиска. Присоединение нового узла требует отслеживания родительского узла *p* текущего узла *q* в процессе перемещения вниз по дереву. При достижении нижней части дерева *p* указывает на узел, связь которого необходимо изменить так, чтобы она указывала на новый вставленный узел.

```
void insert(Item x)
{ Key v = x.key();
  if (head == 0)
    { head = new node(x); return; }
  link p = head;
  for (link q = p; q != 0; p = q ? q : p)
    q = (v < q->item.key()) ? q->l : q->r;
```



**РИСУНОК 12.6 СОЗДАНИЕ ДЕРЕВА БИНАРНОГО ПОИСКА (ПРОДОЛЖЕНИЕ)**

*Эта последовательность отражает вставку ключей G X M P L в BST-дерево, создание которого было показано на рис. 12.5.*

```

if (v < p->item.key()) p->l = new
node(x);
else p->r = new node(x);
}

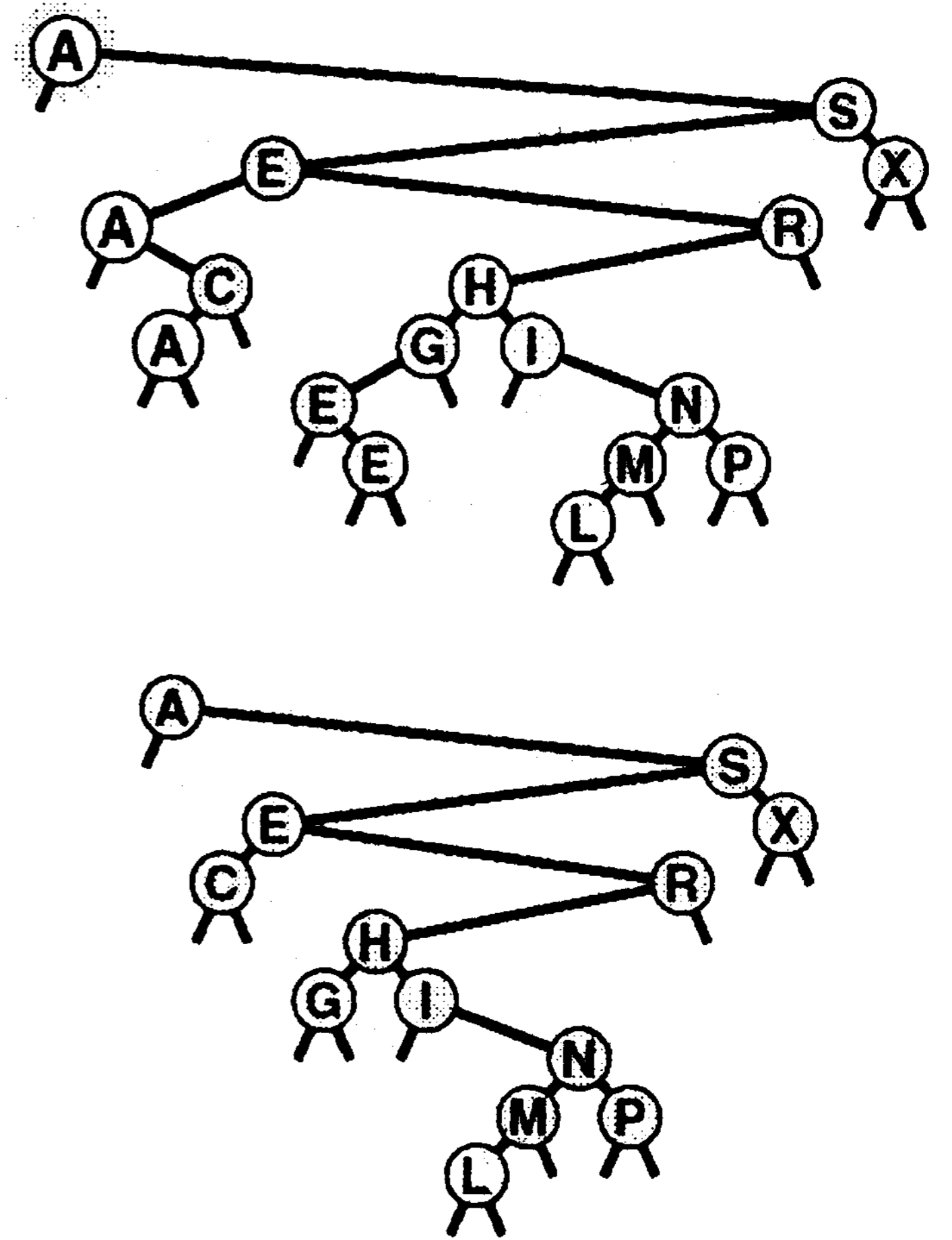
```

Функции BST-дерева в программе 12.8 не проверяют явно наличие элементов с дублированными ключами. При вставке нового узла, ключ которого равен какому-либо ключу, уже вставленному в дерево, узел помещается справа от присутствующего в дереве узла. Одним из побочных эффектов подобного соглашения является то, что узлы с дублированными ключами не отображаются в дереве последовательно (см. рис. 12.7). Однако, их можно найти, продолжив поиск с точки, в которой функция `search` находит первое совпадение, пока не встретится связь 0. Как упоминается в разделе 9.1, существует несколько других возможностей обработки элементов с одинаковыми ключами.

Деревья бинарного поиска — аналог быстрой сортировки. Узел в корне дерева соответствует разделяющему элементу при быстрой сортировке (никакие ключи слева от него не могут быть больше, и никакие ключи справа не могут быть меньше него). В разделе 12.6 будет показано, как это наблюдение связано с анализом свойств деревьев.

## Упражнения

- ▷ 12.46 Нарисуйте BST-дерево, образующееся при вставке элементов с ключами E A S Y Q U T I O N в первоначально пустое дерево.
- ▷ 12.47 Нарисуйте BST-дерево, образующееся при вставке элементов с ключами E A S Y Q U E S T I O N в первоначально пустое дерево.
- ▷ 12.48 Укажите количество сравнений, необходимых для помещения ключей E A S Y Q U E S T I O N в первоначально пустую таблицу символов за счет использования BST-дерева. Считайте, что операция `search` выполняется для каждого ключа, вслед за чем выполняется операция `insert` для каждого промаха при поиске, как имеет место в программе 12.3.
- 12.49 Вставка ключей в порядке A S E R H I N G C в первоначально пустое дерево также дает верхнюю часть дерева, показанного на рис. 12.6. Приведите десять других вариантов порядка этих ключей, которые обеспечат тот же результат.



**РИСУНОК 12.7 ДУБЛИРОВАННЫЕ КЛЮЧИ В ДЕРЕВЬЯХ БИНАРНОГО ПОИСКА**

Когда BST-дерево содержит записи с дублированными ключами (вверху), они оказываются разбросанными по всему дереву, что иллюстрируется выделенными узлами A. Все дублированные ключи размещаются вдоль пути поиска ключа от корня до внешнего узла, и поэтому они легкодоступны. Однако, во избежание путаницы при использовании, типа "A располагается под C", в примерах используются различные ключи (внизу).

- 12.50** Реализуйте функцию `searchinsert` для BST-деревьев (программа 12.8). Функция должна искать в таблице символов элемент с таким же ключом, как и у данного элемента, а затем вставлять элемент, если не находит ни одного такого элемента.
- ▷ **12.51** Создайте функцию, которая возвращает количество элементов в BST-дереве, имеющих ключ, равный данному.
- 12.52** Предположите, что заблаговременно известно, насколько часто должно выполняться обращение к ключам поиска в бинарном дереве. Должны ли ключи вставляться в дерево в порядке возрастания или убывания ожидаемой частоты обращения к ним? Обоснуйте ответ.
- **12.53** Упростите код поиска и вставки в реализации BST-дерева программы 12.8, воспользовавшись двумя фиктивными узлами: `head`, содержащим элемент с зарезервированным ключом, который меньше всех остальных и правая связь которого указывает на корень дерева; и `z`, содержащим элемент со служебным ключом, который больше всех остальных, и левая и правая связи которого указывает на него самого, причем он представляет все внешние узлы (внешние узлы являются связями с `z`). (См. табл. 3.1).
- 12.54** Измените реализацию BST-дерева в программе 12.8 для хранения элементов с дублированными ключами в связных списках, размещенных в узлах дерева. Измените интерфейс, чтобы операция `search` работала подобно операции `sort` (для всех элементов с ключом поиска).
- 12.55** В нерекурсивной процедуре вставки в программе 12.10 для определения того, какую связь узла `p` необходимо заменить новым узлом, используется избыточное сравнение. Приведите реализацию, в которой для исключения этого сравнения используются указатели на связи.

## 12.6 Характеристики производительности деревьев бинарного поиска

Время выполнения алгоритмов обработки BST-деревьев зависит от форм деревьев. В лучшем случае дерево может быть полностью сбалансированным и содержать приблизительно  $\lg N$  узлов между корнем и каждым из внешних узлов, но в худшем случае в каждый из путей поиска может содержать  $N$  узлов.

Можно также надеяться, что в среднем время поиска также будет связано с количеством узлов логарифмической зависимостью, поскольку первый вставляемый элемент становится корнем дерева. Если  $N$  ключей должны быть вставлены в произвольном порядке, то этот элемент делил бы ключи пополам (в среднем), что дало бы логарифмическую зависимость времени поиска (при использовании аналогичных рассуждений применительно к поддеревьям). Действительно, возможен случай, когда BST-дерево приводит в точности к тем же сравнениям, что и бинарный поиск (см. упражнение 12.58). Этот случай был бы наилучшим для данного алгоритма, гарантируя логарифмическую зависимость времени выполнения для всех видов поиска. В действительно произвольной ситуации корнем может быть любой ключ, и поэтому полностью сбалансированные деревья встречаются исключительно редко, соответственно, без особых усилий не удастся сохранять дерево полностью сбалансированным после каждой вставки. Однако полностью несбалансированные деревья также встречаются редко при про-



извольных ключах, потому-то в среднем деревья достаточно хорошо сбалансированы. В этом разделе мы детализируем это наблюдение.

В частности, длина пути и высота бинарных деревьев, рассмотренные в разделе 5.5, непосредственно связаны с затратами на поиск в BST-деревьях. Высота определяет стоимость поиска в худшем случае, длина внутреннего пути непосредственно связана со стоимостью попаданий при поиске, а длина внешнего пути непосредственно связана со стоимостью промахов при поиске.

**Лемма 12.6** *Для обнаружения попадания при поиске в дереве бинарного поиска, образованном  $N$  произвольными ключами, в среднем требуется около  $2 \lg N \approx 1.39 \lg N$  сравнений.*

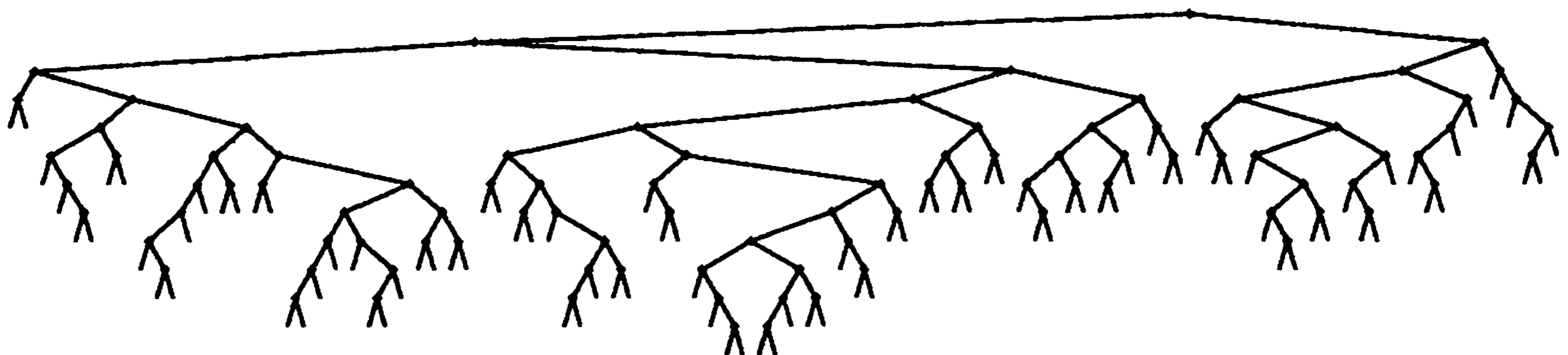
Как упоминалось в разделе 12.3, мы считаем последовательные операции  $==$  и  $<$  одной операцией сравнения. Количество сравнений, использованных для обнаружения попадания при поиске, завершающемся в данном узле, равно 1 плюс расстояние от этого узла до корня. Таким образом, интересующая величина равна 1 плюс средняя длина внутреннего пути BST-дерева, которую можно проанализировать с использованием уже знакомых рассуждений: если  $C_N$  — средняя длина внутреннего пути BST-дерева, состоящего из  $N$  узлов, мы имеем следующее рекуррентное соотношение:

$$C_N = N - 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}),$$

при  $C_1 = 1$ . Член  $N - 1$  учитывает, что корень увеличивает длину пути для каждого из остальных  $N - 1$  узлов на 1; остальная часть выражения вытекает из того, что ключ в корне (вставленный первым) с равной вероятностью может быть  $k$ -м по величине ключом, разбивая дерево на произвольные поддеревья размерами  $k - 1$  и  $N - k$ . Это рекуррентное соотношение почти идентично тому, которое решалось в главе 7 для метода быстрой сортировки, и для получения оговоренного результата его можно решить так же.

**Лемма 12.7** *Для выполнения вставок и обнаружения промахов при поиске в дереве бинарного поиска, образованном  $N$  произвольными ключами, в среднем требуется около  $2 \lg N \approx 1.39 \lg N$  сравнений.*

Поиск произвольного ключа в дереве, содержащем  $N$  узлов, с равной вероятностью может завершиться в любом из  $N - 1$  внешних узлов обнаружением промаха при поиске. Эта лемма в сочетании с тем, что разница длин внешнего и внутрен-



**РИСУНОК 12.8 ПРИМЕР ДЕРЕВА БИНАРНОГО ПОИСКА**

*В этом BST-дереве, которое было построено за счет вставки около 200 произвольных ключей в первоначально пустое дерево, ни один поиск не использует более 12 сравнений. Средняя стоимость обнаружения попадания при поиске приблизительно равна 10.*

него пути в любом дереве равна просто  $2N$  (см. лемму 5.7), обуславливает оговоренный результат. В любом BST-дереве среднее количество сравнений, необходимых для выполнения вставки или обнаружения промаха, приблизительно на 1 больше среднего количества сравнений, необходимых для выявления попадания при поиске.

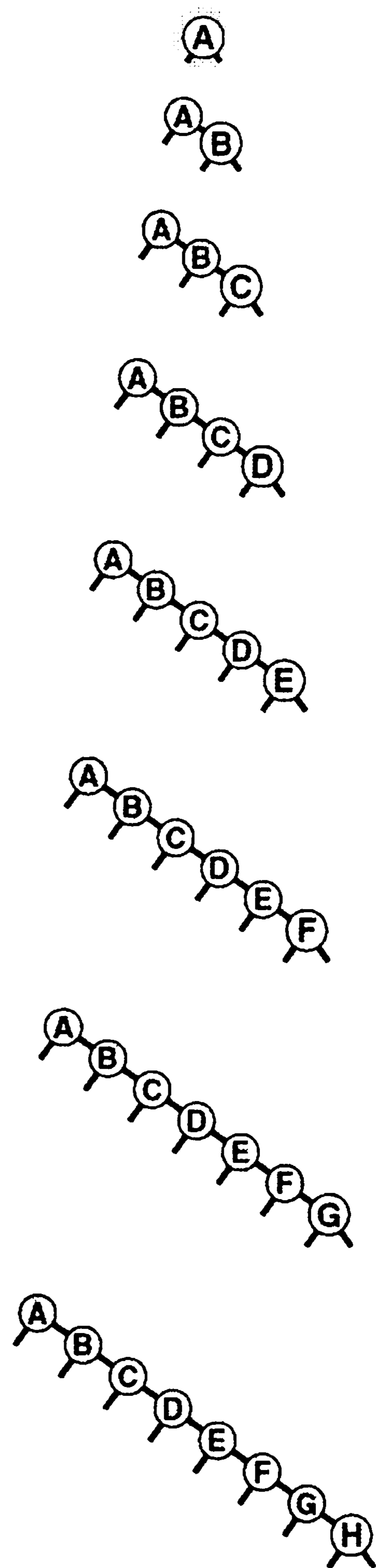
В соответствии с леммой 12.6 следует ожидать, что затраты на поиск для BST-деревьев должна быть приблизительно на 39% выше затрат для бинарного поиска произвольных ключей. Но в соответствии с леммой 12.7 дополнительные затраты вполне окупаются, поскольку новый ключ может быть вставлен почти при тех же затратах — подобная гибкость при использовании бинарного поиска не доступна. На рис. 12.8 показано BST-дерево, полученное в результате длинной цепи произвольных перестановок. Хотя оно содержит несколько длинных и несколько коротких путей, его можно считать хорошо сбалансированным: для выполнения любого поиска требуется менее 12 сравнений, а среднее количество сравнений, необходимых для обнаружения произвольного попадания при поиске равно 7.00, что сравнимо с 5.74 для случая бинарного поиска.

Леммы 12.6 и 12.7 определяют производительность для среднего случая при условии, что порядок ключей произволен. Если это не так, производительность алгоритма имеет тенденцию к снижению.

**Лемма 12.8** *В худшем случае для поиска в дереве бинарного поиска с  $N$  ключами может потребоваться  $N$  сравнений.*

На рис. 12.9 и 12.10 показаны два примера наихудших случаев BST-деревьев. Для этих деревьев поиск с использованием бинарного дерева ничем не лучше последовательного поиска с использованием односвязных списков.

Таким образом, высокая производительность базовой реализации таблиц символов с использованием BST-дерева требует, чтобы ключи в достаточной степени были подобны произвольным ключам, а дерево не содержало длинных путей. Более того, наихудший случай не столь уж редко встречается на практике — он возникает при вставке ключей в первоначально пустое дерево по порядку или в обратном порядке с применением стандартного алгоритма — последовательность операций, которую мы определено можем предпринять, не получив никакого явного предупреждения не делать этого. В главе 13 исследуются технологии превращения этого худшего случая в крайне маловероятный и полного его исключения, превращения всех деревьев в подобные деревья для наилучшего случая, длины всех путей в которых гарантировано определяются логарифмической зависимостью.



**РИСУНОК 12.9 ХУДШИЙ СЛУЧАЙ ДЕРЕВА БИНАРНОГО ПОИСКА**

*Если ключи в BST-дереве вставляются в порядке возрастания, дерево вырождается в форму, эквивалентную односвязному списку, приводя к квадратичной зависимости времени создания дерева и к линейной зависимости времени поиска.*

Ни одна из других рассмотренных реализаций таблиц символов не может использоваться для выполнения задачи вставки в таблицу очень большого количества произвольных ключей, а затем поиска каждого из них — время выполнения каждого из методов, описанных в разделах 12.2—2.4 определяется квадратичной зависимостью. Более того, из анализа следует, что среднее расстояние до узла в бинарном дереве пропорционально логарифму количества узлов в дереве, что позволяет эффективно выполнять смешанные наборы операций поиска, вставки и других операций с АДТ таблицы символов, что и будет вскоре показано.

## Упражнения

- ▷ 12.56 Создайте рекурсивную программу, которая вычисляет максимальное количество сравнений, требуемых для любого поиска в данном BST-дереве (высоту дерева).
- ▷ 12.57 Создайте рекурсивную программу, которая вычисляет максимальное количество сравнений, требуемых для попадания при поиске в данном BST-дереве (длину внутреннего пути дерева, деленную на  $N$ ).

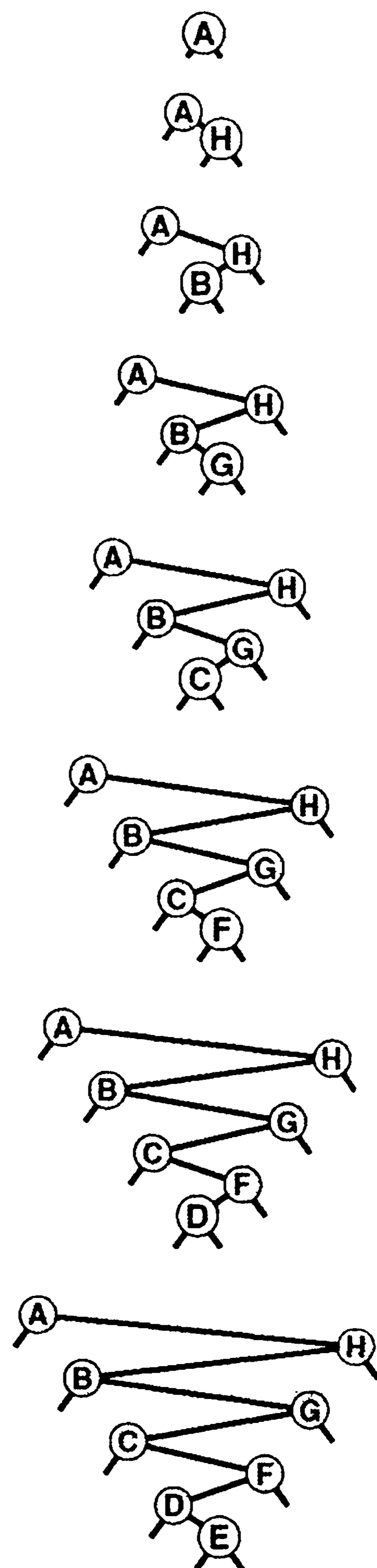
12.58 Приведите последовательность вставок ключей **E A S Y Q U E S T I O N** в первоначально пустое BST-дерево, чтобы созданное при этом дерево было эквивалентно бинарному поиску в смысле последовательности сравнений, выполняемых при поиске любого ключа в BST-дереве и при бинарном поиске.

- 12.59 Создайте программу, которая вставляет набор ключей в первоначально пустое BST-дерево так, чтобы созданное дерево было эквивалентно бинарному поиску, как описывалось в упражнении 12.58.

12.60 Нарисуйте все различающиеся по структуре BST-деревья, образованные после вставки  $N$  ключей в первоначально пустое дерево, для  $2 \leq N \leq 5$ .

- 12.61 Определите вероятность того, что каждое из деревьев в упражнении 12.60 является результатом вставки  $N$  произвольных различных элементов в первоначально пустое дерево.
- 12.62 Сколько бинарных деревьев, состоящих из  $N$  узлов, имеют высоту  $N$ ? Сколько существует различных способов вставки  $N$  различных ключей в первоначально пустое дерево, приводящих к образованию BST-дерева с высотой  $N$ ?

- 12.63 Докажите с использованием метода индукции, что разница между длинами внешнего и внутреннего путей в любом бинарном дереве составляет  $2N$  (см. лемму 5.7).



**РИСУНОК 12.10 ЕЩЕ ОДИН ХУДШИЙ СЛУЧАЙ ДЕРЕВА БИНАРНОГО ПОИСКА**

*Множество других подобных вариантов порядка вставки ключей приводят к вырождению BST-дерева. Тем не менее, дерево бинарного поиска, образованное произвольно упорядоченными ключами, скорее всего, окажется хорошо сбалансированным.*

**12.64** Определите эмпирически среднее и стандартное отклонение количества сравнений, использованных для обнаружения попаданий и промахов при поиске в BST-дереве, созданном в результате вставки  $N$  произвольных ключей в первоначально пустое дерево, для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

**12.65** Создайте программу, которая строит  $t$  BST-деревьев за счет вставки  $N$  произвольных ключей в первоначально пустое дерево и вычисляет максимальную высоту дерева (максимальное количество сравнений, необходимых для обнаружения любого промаха при вставке в любом из  $t$  деревьев) для  $N = 10^3, 10^4, 10^5$  и  $10^6$  при  $t = 10, 100$  и  $1000$ .

## 12.7 Реализация индексов при использовании таблиц символов

Во многих приложениях необходимо выполнять поиск в структуре просто с целью нахождения элементов без их перемещения. Например, может существовать массив элементов с ключами, для которого требуется метод поиска, определяющий в массиве индекс элемента, соответствующего определенному ключу. Может также требоваться удаление элемента с данным индексом из структуры поиска с его сохранением в массиве для какого-либо другого применения. В разделе 9.6 были рассмотрены преимущества обработки индексированных элементов в очередях по приоритету, которые косвенно обращаются к данным клиентского массива. Применительно к таблицам символов эта же концепция ведет к уже знакомым *индексам*: внешней по отношению к набору элементов поисковой структуры, которая обеспечивает быстрый доступ к элементам с данным ключом. В главе 16 будет рассматриваться случай, когда элементы и, возможно, даже индексы хранятся во внешнем хранилище; в этом разделе кратко исследуется случай, когда и элементы и индексы размещаются в памяти.

Деревья бинарного поиска можно определить таким образом, чтобы индексы строились в точности так же, как при обеспечении косвенного метода для сортировки в разделе 6.8 и для сортирующих деревьев в разделе 9.6: необходимо использовать контейнер **Index** для определения элементов BST-дерева и обеспечить, чтобы ключи извлекались из элементов, как обычно, через функцию-члена **key**. Более того, для связей можно использовать параллельные массивы, как это было сделано для связных списков в главе 3. Используются три массива: для элементов, левых связей и правых связей. Связи являются индексами массивов (целыми значениями) и ссылки на связи, подобные

```
x = x->l
```

во всем коде заменяются на ссылки типа

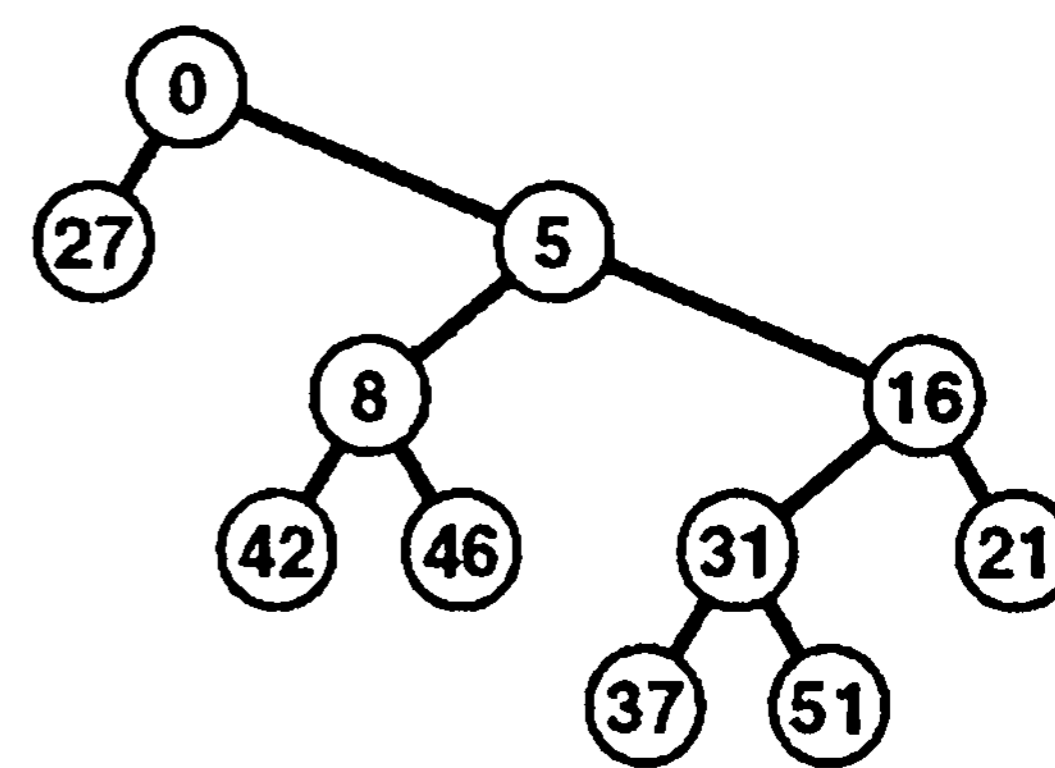
```
x = l[x]
```

Этот подход устраняет затраты на динамическое распределение памяти для каждого узла — элементы занимают массив независимо от функции поиска, и мы заранее выделяем два целочисленных значения на каждый элемент для хранения связей дерева, признавая, что потребуется объем памяти не меньше этого, когда все элементы будут помещены в структуру поиска. Память под связи используется не всегда, тем

не менее, она всегда готова для использования подпрограммой поиска, не требуя дополнительного времени на распределение. Другая важная особенность этого подхода заключается в том, что он допускает добавление дополнительных массивов (содержащих дополнительную связанную с каждым узлом информацию) без какого-либо изменения кода манипулирования деревом. Когда подпрограмма поиска возвращает индекс элемента, она предоставляет способ немедленного доступа ко всей информации, связанной с этим элементом, путем использования индекса для доступа к соответствующему массиву.

Такой способ реализации BST-деревьев как средства упрощения поиска в больших массивах элементов иногда весьма полезен, поскольку исключает дополнительные затраты на копирование элементов во внутреннее представление АД и перегрузки, связанной с распределением и конструированием при помощи `new`. Использование массивов не подходит, когда объем памяти играет первостепенную роль, а таблица символов увеличивается и уменьшается в значительных пределах. В особенности это актуально, если заранее трудно предвидеть максимальный размер таблицы символов. Если невозможно точно предвидеть размер, неиспользуемые связи могут привести к напрасному расходованию памяти в области массива элементов.

Важное применение концепции индексирования — обеспечение поиска ключевых слов в строке текста (см. рис. 12.11). В программе 12.11 показан пример такого приложения. Она считывает текстовую строку из внешнего файла. Затем, просматривая каждую позицию в текстовой строке с целью определения ключа строки от данной позиции и до конца строки, она вставляет все ключи в таблицу символов, используя указатели на строки. Подобное применение ключей строк отличается от определения типа строкового элемента, как в упражнении 12.2, поскольку никакое распределение памяти под область хранения не требуется. Используемые ключи строк имеют произвольную длину, но мы поддерживаем только указатели на них и просматриваем лишь то количество символов, которое требуется для определения того, какая из двух строк должна следовать первой. Никакие две строки не совпадают (например, все они имеют различную длину), но если изменить операцию `==`, чтобы сравнить при условии, что одна является префиксом второй, можно было бы воспользоваться таб-



0 call me ishmael some...  
 5 me ishmael some year...  
 8 ishmael some years a...  
 16 some years ago never...  
 21 years ago never mind...  
 27 ago never mind how l...  
 31 never mind how long...  
 37 mind how long precis...  
 42 how long precisely h...  
 46 long precisely havin...  
 51 precisely having lit...  
 ...

#### РИСУНОК 12.11 ИНДЕКС ТЕКСТОВОЙ СТРОКИ

*В этом примере индекса строки мы определяем ключ строки так, чтобы он начинался с каждого слова в тексте; затем строится BST-дерево за счет обращения к ключам по их индексам строк. В принципе, ключи имеют произвольную длину, но в общем случае исследуются только несколько начальных символов. Например, для определения того, встречается ли фраза **never mind** в этом тексте, она сравнивается с **call...** в корне (индекс строки 0), затем с **me...** в правом дочернем узле корня (индекс 5), затем с **some...** в правом дочернем узле этого узла (индекс 16), а затем **never mind** отыскивается в левом дочернем узле данного узла (индекс 31).*

лицей символов для выяснения того, присутствует ли данная строка в тексте, и простым вызовом функции `search`.

### Программа 12.11 Пример индексирования текстовых строк

В этой программе предполагается, что `Item.cxx` определяет представление данных `char*` для ключей строк в элементах, перегруженную операцию `operator<`, которая использует `strcmp`, перегруженную операцию `operator==`, которая использует `strncmp` и оператор преобразования из `Item` в `char*` (см. текст). Главная программа считывает текстовую строку из указанного файла и использует таблицу символов для построения индекса из строк, определяемых начальными символами текстовых строк. Затем она считывает запрашиваемые строки из стандартного ввода и выводит позицию, в которой запрос найден в тексте (или выводит строку `not found`). При реализации таблицы символов с использованием BST-дерева поиск выполняется быстро даже для очень больших строк.

```
#include <iostream.h>
#include <fstream.h>
#include "Item.cxx"
#include "ST.cxx"
static char text[maxN];
int main(int argc, char *argv[])
{ int N = 0; char t;
  ifstream corpus; corpus.open(*++argv);
  while (N < maxN && corpus.get(t)) text[N++] = t;
  text[N] = 0;
  ST<Item, Key> st(maxN);
  for (int i = 0; i < N; i++) st.insert(&text[i]);
  char query[maxQ]; Item x, v(query);
  while (cin.getline(query, maxQ))
    if ((x = st.search(v.key())).null())
      cout << "not found: " << query << endl;
    else cout << x-text << ": " << query << endl;
}
```

Программа 12.11 считывает серии запросов со стандартного ввода, использует функцию `search` для определения присутствия каждого запроса в тексте и выводит позицию в тексте для первого совпадения с запросом. Если таблица символов реализована с использованием BST-дерева, то в соответствии с леммой 12.6 можно ожидать, что для поиска потребуется около  $2N \ln N$  сравнений. Например, как только индексы построены, любую фразу в тексте, состоящем приблизительно из 1 миллиона символов, можно было бы найти с использованием около 30 операций сравнения строк. Это приложение равносильно индексированию, поскольку указатели строк `S` являются индексами массива символов: если `x` указывает на `text[i]`, то разность двух указателей, `x-text`, равна `i`.

При построении индексов в реальных приложениях потребуется учесть и множество других моментов. Существует множество способов, в соответствии с которыми можно извлечь конкретные преимущества из свойств ключей строк в плане ускорения работы алгоритмов. Более сложные методы поиска строк и создания индексов с полезными возможностями ключей строк будут основными темами в части 5.

В таблицу 12.2 сведены результаты исследований, подтверждающие приведенные аналитические рассуждения и демонстрирующие применение деревьев бинарного поиска для работы с динамическими таблицами символов с произвольными ключами.

Таблица 12.2 Эмпирические исследования реализаций таблиц символов

В этой таблице приведено относительное время создания таблицы символов и поиска каждого ключа в таблице. Деревья бинарного поиска обеспечивают быстрые реализации поиска и вставки; при использовании всех других методов для выполнения одной из этих двух задач требуется время, определяемое квадратичной зависимостью. В общем случае бинарный поиск выполняется несколько быстрее поиска в BST-дереве, но он не может быть использован применительно к очень большим файлам, если только таблицу нельзя предварительно отсортировать. Стандартная реализация BST-дерева распределяет память для каждого узла дерева, в то время как реализация с использованием индексов предварительно распределяет память для всего дерева (что ускоряет создание) и вместо указателей использует индексы массивов (что замедляет поиск).

N	конструирование					попадания при поиске				
	A	L	B	T	T*	A	L	B	T	T*
1250	1	5	6	1	0	6	13	0	1	1
2500	0	21	24	2	1	27	52	1	1	1
5000	0	87	101	4	3	111	211	2	2	3
12500		645	732	12	9	709	1398	7	8	9
25000		2551	2917	24	20	2859	5881		15	21
50000				61	50				38	48
100000				154	122				104	122
200000				321	275				200	272

Ключ:

- A Неупорядоченный массив (упражнение 12.20)
- B Упорядоченный связный список (упражнение 12.21)
- C Бинарный поиск (программа 12.7)
- T Дерево бинарного поиска, стандартное (программа 12.8)
- T\* Индексированное дерево бинарного поиска (упражнение 12.67)

## 12.8 Вставка в корень в деревьях бинарного поиска

В стандартной реализации BST-деревьев каждый новый вставленный узел попадает куда-то в нижнюю часть дерева, заменяя какой-то внешний узел. Это не является абсолютно обязательным; это всего лишь следствие естественного алгоритма с использованием рекурсивной вставки. В этом разделе рассматривается другой метод вставки, при котором каждый новый элемент вставляется в корень, и поэтому недавно вставленные узлы находятся вблизи *вершины* дерева.

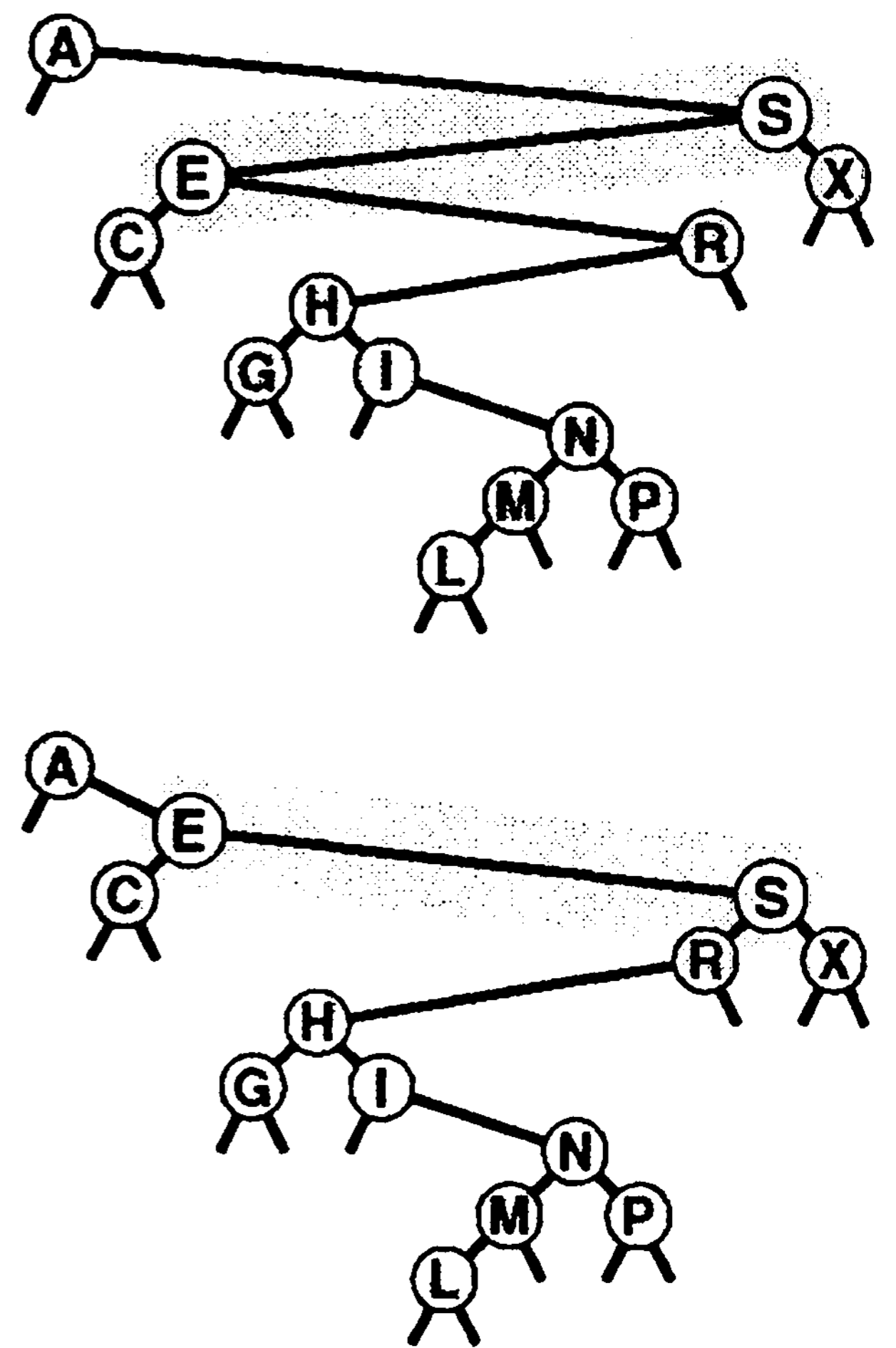
Построенные таким образом деревья обладают рядом интересных свойств, но главная побудительная причина рассмотрения этого метода в том, что он играет важную роль в двух из усовершенствованных алгоритмов, рассматриваемых в главе 13.

Предположим, что ключ вставляемого элемента больше ключа в корне. Создание нового дерева можно было бы начать с помещения нового элемента в новый корневой

узел, устанавливая старый корень в качестве левого поддерева, а правое поддерево старого корня в качестве правого поддерева. Однако, правое поддерево может содержать некоторые меньшие ключи, поэтому для завершения вставки потребуется выполнить дополнительные действия. Аналогично, если ключ вставляемого элемента меньше ключа корня и больше всех ключей в левом поддереве корня, можно снова создать новое дерево с новым элементом, помещенным в корень, но если левое поддерево содержит какие-либо большие ключи, необходимы дополнительные действия. Перемещения всех узлов с меньшими ключами в левое поддерево и всех узлов с большими ключами в правое в общем случае оказывается сложным преобразованием, поскольку узлы, которые должны быть перемещены, могут быть разбросаны по всему пути поиска для вставляемого узла.

К счастью, существует простое рекурсивное решение этой проблемы, которое основывается на *ротации* (*rotation*) — фундаментальном преобразовании деревьев. По существу, ротация позволяет менять местами роль корня и одного из его дочерних узлов при сохранении порядка ключей в узлах BST-дерева. *Ротация вправо* затрагивает корень и левый дочерний узел (см. рис. 12.12). Ротация помещает корень справа, изменяя на обратное направление левой связи корня: перед ротацией она указывает от корня к левому дочернему узлу, а после ротации — от левого дочернего узла (нового корня) к старому корню (правый дочерний узел нового корня). Основная часть, которая обеспечивает работу ротации, — копирование правой связи левого дочернего узла, чтобы она стала левой связью старого корня. Эта связь указывает на все узлы с ключами между двумя узлами, участвующими в ротации. И наконец, связь со старым корнем должна быть изменена так, чтобы указывать на новый корень. Описание *ротации влево* аналогично приведенному выше, с учетом замены определений "правый" и "левый" (см. рис. 12.13).

Ротация — это локальное изменение, затрагивающее только три связи и два узла, которое позволяет перемещать узлы по деревьям, не изменяя глобальные свойства порядка, превращающие BST-дерево в полезную структуру для поиска (см. программу 12.12). Ротации используются для перемещения конкретных узлов по дереву и для предотвращения разбалансировки деревьев. В разделе 12.9 с помощью ротаций реализованы *remove*, *join* и



**РИСУНОК 12.12 РОТАЦИЯ ВПРАВО В BST-ДЕРЕВЕ**

На этой схеме показан результат (внизу) ротации вправо в узле *S* в примере BST-дерева (вверху). Узел, содержащий *S*, перемещается в дереве вниз, становясь правым дочерним узлом своего прежнего левого дочернего узла.

Ротация выполняется путем получения связи с новым корнем *E* из левой связи *S*, установки левой связи *S* путем копирования правой связи *E*, установки правой связи *E* указывающей на *S* и установки связи из *A* указывающей не на *S*, а на *E*. Эффект ротации заключается в перемещении *E* и его левого поддерева на один уровень вверх и перемещении *S* и его правого поддерева на один уровень вниз. Остальная часть дерева остается неизменной.



другие операции с АД; в главе 13 они будут применяться для построения деревьев, для которых допустима неоптимальная производительность.

### Программа 12.12 Ротации в BST-деревьях

Эти две симметричных процедуры выполняют операцию *rotation* (ротация) в BST-дереве. *Ротация вправо* делает старый корень правым поддеревом нового корня (старого левого поддерева корня); *ротация влево* превращает старый корень в левое поддерево нового корня (старого правого поддерева корня). Для реализаций, в которых поле счетчика поддерживается в узлах (например, для поддержки операции *select*, как будет показано в разделе 14.9), необходимо также обменять поля счетчиков для участвующих в ротации узлов (см. упражнение 12.75).

```
void rotR(link& h)
{ link x = h->l; h->l = x->r; x->r = h;
  h = x; }
void rotL(link& h)
{ link x = h->r; h->r = x->l; x->l = h;
  h = x; }
```

Операции ротации обеспечивают простую рекурсивную реализацию вставки в корень: необходимо рекурсивно вставить новый элемент в соответствующее поддерево (оставив его по завершении рекурсивной операции в корне этого дерева), а затем выполнить ротацию, чтобы сделать узел корнем главного дерева. Пример показан на рис. 12.14, а программа 12.13 содержит реализацию данного метода. Эта программа — убедительный пример больших возможностей рекурсии. Любой читатель, которого это не убеждает, может попытаться выполнить упражнение 12.76.

### Программа 12.13 Вставка в корень BST-дерева

При наличии функций ротации, определенных в программе 12.12, реализация рекурсивной функции, которая вставляет новый узел в корень BST-дерева, очевидна: необходимо вставить новый элемент в корень соответствующего поддерева, а затем выполнить соответствующую ротацию, что приведет к его переносу в корень главного дерева.

```
private:
void insertT(link& h, Item x)
{ if (h == 0) { h = new node(x); return; }
  if (x.key() < h->item.key())
    { insertT(h->l, x); rotR(h); }
  else { insertT(h->r, x); rotL(h); }
}
public:
void insert(Item item)
{ insertT(head, item); }
```

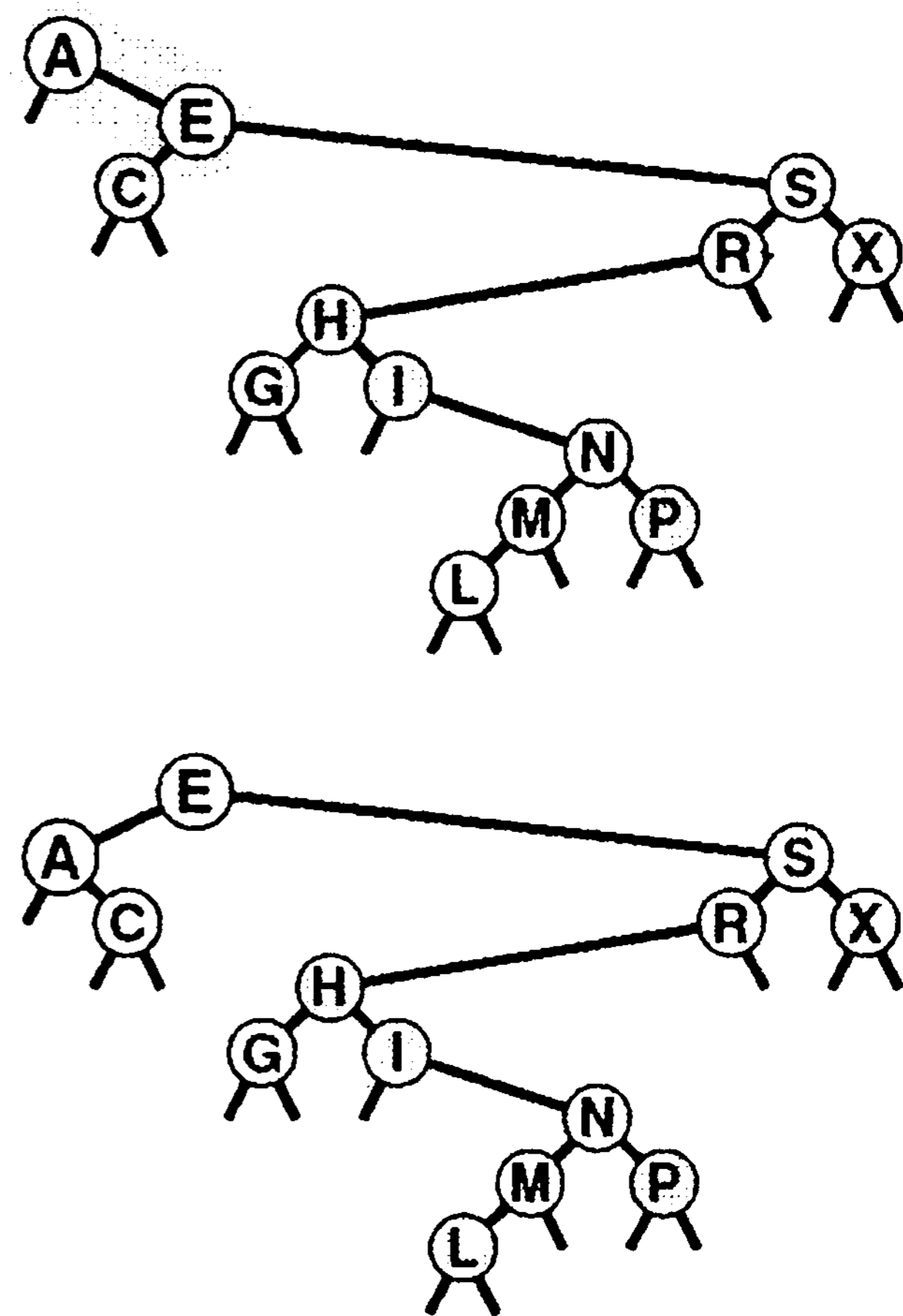


РИСУНОК 12.13 РОТАЦИЯ ВЛЕВО В BST-ДЕРЕВЕ

На этой схеме показан результат (внизу) ротации влево узла *A* в примере BST-дерева (вверху). Узел, содержащий *A*, перемещается вниз, становясь левым дочерним узлом своего прежнего правого дочернего узла.

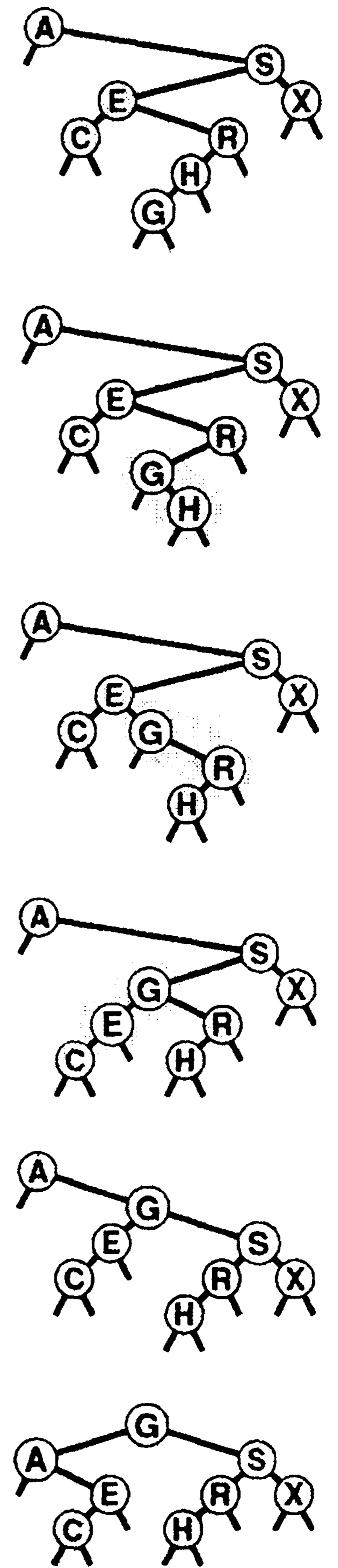
Ротация выполняется за счет получения связи с новым корнем *E* из правой связи *A*, установки правой связи *A* путем копирования левой связи *E*, установки левой связи *E* на *A* и установки связи с *A* (верхняя связь дерева) вместо этого на *E*.

На рис. 12.15 и 12.16 показано создание BST-дерева путем вставки последовательности ключей в первоначально пустое дерево с использованием метода вставки в корень. Если последовательность ключей произвольна, построенное таким образом BST-дерево обладает совершенно теми же стохастическими свойствами, что и BST-дерево, построенное стандартным методом. Например, леммы 12.6 и 12.7 справедливы и для BST-деревьев, построенных при помощи вставки в корень.

На практике преимущество метода вставки в корень состоит в том, что недавно вставленные ключи располагаются вблизи вершины. Следовательно, затраты на обнаружение попаданий при поиске недавно вставленных ключей, скорее всего, будут ниже, нежели при стандартном методе. Это важно, поскольку во многих приложениях выполняется именно такой динамический набор операций *search* и *insert*. Таблица символов может содержать изрядное количество элементов, но значительная часть поисков может относиться к наиболее недавно вставленным элементам. Например, в системе обработки коммерческих транзакций активные транзакции могут оставаться вблизи вершины и обрабатываться быстро без обращения к старым транзакциям, которые должны быть опущены. Метод вставки в корень автоматически придает структуре данных это и аналогичные свойства.

Если также изменить функцию *search*, чтобы она помещала найденный узел в корень, получится метод самоорганизующегося поиска (см. упражнение 12.28), который сохраняет часто посещаемые узлы вблизи вершины дерева. В главе 13 будет показано систематическое применение этой идеи при создании реализации таблицы символов, обладающей гарантированными характеристиками производительности.

Как и в случае ряда других методов, упомянутых в этой главе, трудно точно определить производительность метода вставки в корень по сравнению со стандартным методом вставки, поскольку производительность настолько зависит от комбинации различных операций с таблицей символов, что ее трудно проанализировать аналитически. Невозможность проанализировать алгоритм не обязательно должна удерживать от использования вставки в корень, когда известно, что основная масса поисков будет связана с недавно вставленными данными, однако всегда требуются гарантии в отношении производительности; основной темой в главе 13 являются такие методы конструирования BST-деревьев, которые могут предоставить такие гарантии.



**РИСУНОК 12.14 ВСТАВКА В КОРЕНЬ BST-ДЕРЕВА**

Эта последовательность отображает результат вставки *G* в BST-дерево, приведенное на верхнем рисунке, после (рекурсивного) выполнения ротации после вставки с целью перемещения вставленного узла *G* в корень. Этот процесс эквивалентен вставке *G* с последующим выполнением последовательности ротаций для перемещения его в корень.

## Упражнения

- ▷ 12.73 Нарисуйте BST-дерево, образующееся при вставке элементов с ключами **E A S Y Q U E S T I O N** в первоначально пустое дерево методом вставки в корень.

12.74 Приведите последовательность из 10 ключей (используя буквы от A до J), для которой при вставке в первоначально пустое дерево методом вставки в корень для создания дерева требуется максимальное количество сравнений. Укажите количество используемых сравнений.

12.75 Добавьте код, необходимый, чтобы программа 12.12 корректно изменяла поля счетчиков, которые должны изменяться после ротации.

- 12.76 Реализуйте нерекурсивную функцию вставки в корень BST-дерева (см. программу 12.13).

12.77 Эмпирически определите среднее значение и стандартное отклонение количества сравнений, используемых для обнаружения попаданий и промахов при поиске в BST-дереве, построенном при помощи вставки  $N$  произвольных ключей в первоначально пустое дерево и последующего выполнения  $N$  произвольных поисков  $N/10$  наиболее недавно вставленных ключей для  $N = 10^3$ ,  $10^4$ ,  $10^5$  и  $10^6$ . Проведите эксперименты и для стандартного метода вставки и для метода вставки в корень; затем сравните полученные результаты.

## 12.9 Реализации других функций АД с помощью BST-дерева

Рекурсивные реализации, приведенные в разделе 12.5 для основных функций *search*, *insert* и *sort*, которые используют структуры бинарных деревьев, достаточно просты. В этом разделе рассматриваются реализации функций *select*, *join* и *remove*. Одна из них, *select*, имеет также рекурсивную реализацию, однако реализация других может оказаться трудной задачей, приводящей к проблемам, связанным с производительностью. Операцию *select* важно рассмотреть, поскольку возможность эффективной поддержки операций *select* и *sort* — одна из причин, по которой для многих приложений BST-деревья оказываются предпочтительнее других структур. Некоторые программисты избегают использования BST-деревьев, чтобы не иметь дело с операцией *remove*; в этом разделе рассматривается компактная реализация, которая связывает эти операции вместе и использует технологию ротации к корню, описанную в разделе 12.8.

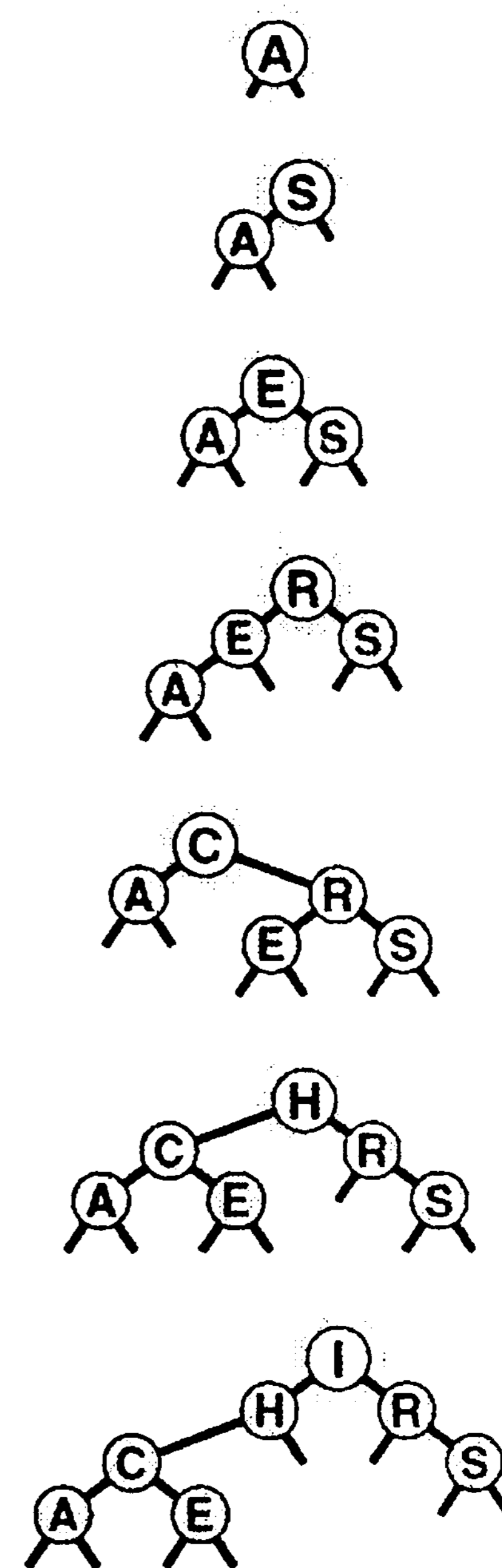


РИСУНОК 12.15 СОЗДАНИЕ BST-ДЕРЕВА ЗА СЧЕТ ВСТАВКИ В КОРЕНЬ

Эта последовательность отображает результат вставки ключей **A S E R C H I** в первоначально пустое BST-дерево при помощи метода вставки в корень. Каждый новый узел вставляется в корень с изменением связей, расположенных вдоль его пути поиска, что приводит к образованию соответствующего BST-дерева.

В общем случае операции связаны с перемещением вниз по дереву; поэтому для случая произвольного BST-дерева можно ожидать, что затраты будут определяться логарифмической зависимостью. Однако, нельзя гарантировать, что BST-деревья останутся произвольными после выполнения над ними нескольких операций. В конце этого раздела мы вернемся к данному вопросу.

Для реализации операции *select* можно использовать рекурсивную процедуру, аналогичную методу выбора с использованием быстрой сортировки, который описан в разделе 7.8. Для отыскания в BST-дереве элемента с  $k$ -м наименьшим ключом проверяется количество узлов в левом поддереве. Если там имеется  $k$  узлов, возвращается корневой элемент. В противном случае, если левое поддерево содержит более  $k$  узлов, отыскивается (рекурсивно)  $k$ -й наименьший узел в нем. Если ни одно из этих условий не соблюдается, то левое поддерево содержит  $t$  элементов при  $t < k$ , и  $k$ -й наименьший элемент в BST-дереве является  $(k - t - 1)$ -м наименьшим элементом в правом поддереве. Программа 12.14 содержит реализацию описанного метода. Как обычно, поскольку каждое выполнение функции завершается максимум одним рекурсивным вызовом, нерекурсивная версия очевидна (см упражнение 12.78).

#### Программа 12.14 Выбор с помощью BST-дерева

В этой процедуре предполагается, что размер поддерева поддерживается для каждого узла дерева. Сравните эту программу с выбором с использованием быстрой сортировки в массиве (программа 9.6).

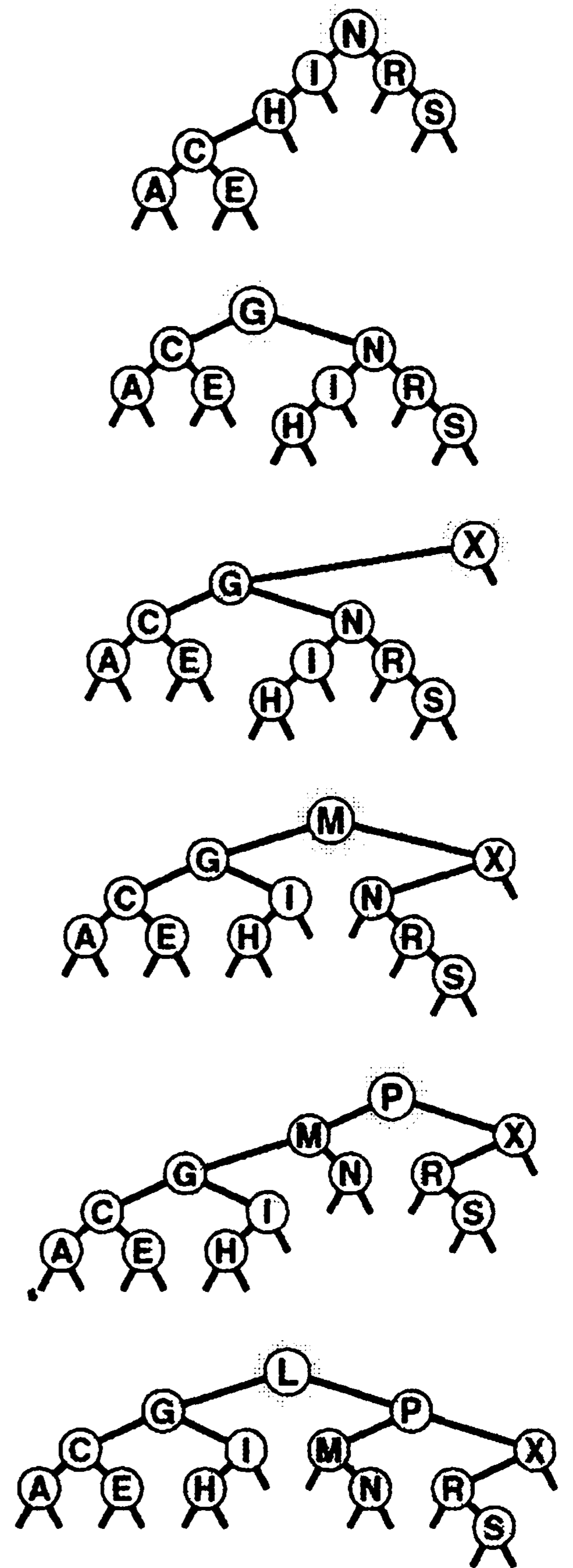
**private:**

```
Item selectR(link h, int k)
{ if (h == 0) return nullItem;
  int t = (h->l == 0) ? 0 : h->l->N;
  if (t > k) return selectR(h->l, k);
  if (t < k) return selectR(h->r, k-t-1);
  return h->item;
}
```

**public:**

```
Item select(int k)
{ return selectR(head, k); }
```

С точки зрения алгоритма основная причина включения полей счетчика в узлы BST-дерева — необходимость поддержки реализации операции *select*. Это позволяет также обеспечивать тривиальную реализацию операции *count* (возвращение поля счетчика в конечном узле); в главе 13 будет продемонстрировано еще одно применение. Недостатки присутствия поля счетчика заключаются в использовании дополнительной памяти для каждого узла и необходимости обновления поля каждой функцией, изме-



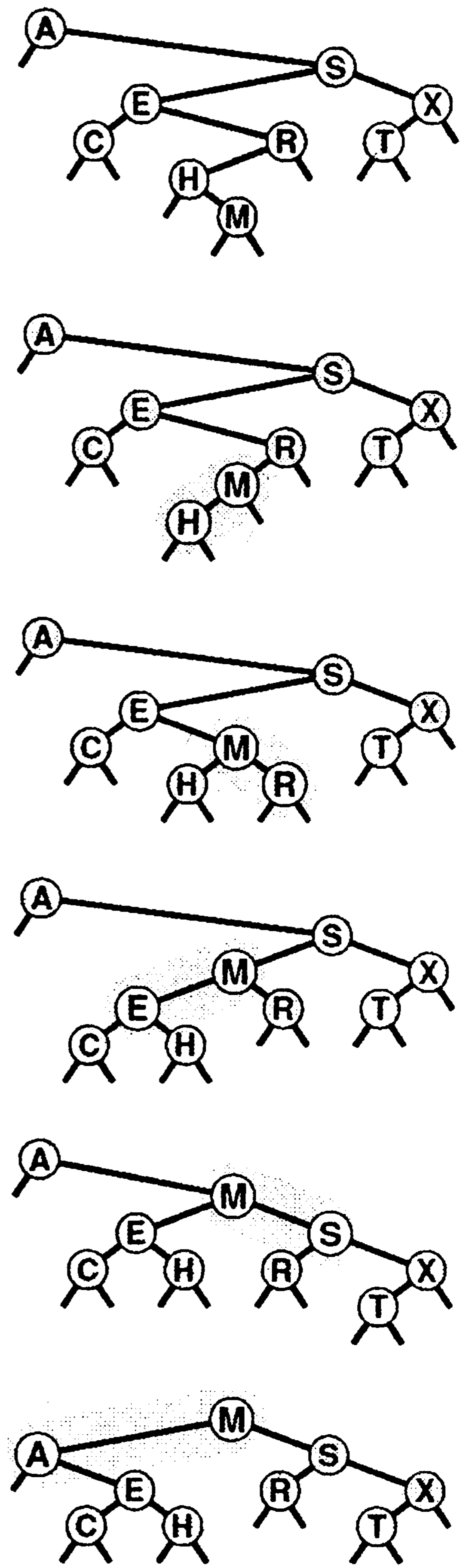
**РИСУНОК 12.16**  
**КОНСТРУИРОВАНИЕ BST-**  
**ДЕРЕВА ПРИ ПОМОЩИ**  
**ВСТАВКИ В КОРЕНЬ**  
**(ПРОДОЛЖЕНИЕ)**

*Эта последовательность отображает вставку ключей  $N G X M P L$  в BST-дерево, которое начинало создаваться на рис. 12.15.*

няющей дерево. Поддержка поля счетчика может не окупаться в некоторых приложениях, в которых основным операциями являются *insert* и *search*, но это может оказаться незначительной платой, если важно поддерживать операцию *select* в динамической таблице символов.

Эту реализацию операции *select* можно преобразовать в операцию *partition* (разбиение на части), которая реорганизует дерево для помещения  $k$ -го наименьшего элемента в корень, что в точности соответствует рекурсивной технологии, которая использовалась для вставки в корень в разделе 12.8: если мы (рекурсивно) помещаем требуемый узел в корень одного из поддеревьев, его затем можно сделать корнем, всего дерева, применив единственную ротацию. Программа 12.15 содержит реализацию этого метода. Подобно ротациям, разбиение на части не принадлежит к операциям АТД, поскольку эта функция преобразует конкретное представление таблицы символов и не должна быть прозрачной для клиентов. Скорее, это вспомогательная процедура, которую можно использовать для реализации операций АТД либо с целью повышения эффективности их выполнения. На рис. 12.17 приведен пример, иллюстрирующий аналогично показанному на рис. 12.14, что этот процесс эквивалентен спуску по пути поиска от корня до требуемого узла дерева, а затем подъему обратно с выполнением ротаций для перемещения узла в корень.

Для удаления узла с данным ключом из BST-дерева вначале необходимо проверить, находится ли он в одном из поддеревьев. Если да, мы заменяем это поддерево результатом удаления (рекурсивного) из него узла. Если удаляемый узел находится в корне, дерево заменяется результатом объединения двух поддеревьев в одно. Для выполнения такого объединения существует несколько возможностей. Один из возможных подходов проиллюстрирован на рис. 12.18, а реализация представлена в программе 12.16. Для объединения двух BST-деревьев, все ключи второго из которых заведомо больше ключей первого, ко второму дереву применяется операция *partition* с целью перемещения в корень наименьшего элемента в этом дереве. На данном этапе левое поддерево корня должно быть пустым (иначе в нем располагался бы элемент, который меньше элемента в корне — явное противоречие), и задачу можно завершить, заменив эту связь связью с первым деревом. На рис. 12.19 показана последовательность удалений в примере дерева, которая иллюстрирует некоторые из возможных ситуаций.



**РИСУНОК 12.17 РАЗДЕЛЕНИЕ BST-ДЕРЕВА НА ЧАСТИ**

Эта последовательность отображает результат (внизу) разбиения примера BST-дерева (вверху) на части расположенным практически посередине ключом; при этом применяется (рекурсивно) ротация, как это делалось во время вставки в корень.

## Программа 12.16 Удаление узла с данным ключом из BST-дерева

В этой реализации операции *remove* выполняется удаление из BST-дерева первого встреченного узла с ключом *v*. Выполняя просмотр сверху вниз, программа осуществляет рекурсивные вызовы для соответствующего поддеревья до тех пор, пока удаляемый узел не окажется в корне. Затем программа заменяет узел результатом объединения его двух поддеревьев — наименьший узел в правом поддереве становится корнем, а затем его левая связь начинает указывать на левое поддерево.

```
private:
    link joinLR(link a, link b)
    {
        if (b == 0) return a;
        partR(b, 0); b->l = a;
        return b;
    }
    void removeR(link& h, Key v)
    {
        if (h == 0) return;
        Key w = h->item.key();
        if (v < w) removeR(h->l, v);
        if (w < v) removeR(h->r, v);
        if (v == w)
            { link t = h;
              h = joinLR(h->l, h->r); delete t; }
    }
public:
    void remove(Item x)
    { removeR(head, x.key()); }
```

Этот подход асимметричен и является *особым* в одном отношении: почему в качестве корня нового дерева используется наименьший ключ во втором дереве, а не наибольший ключ в первом дереве? Другими словами, почему удаляемый узел заменяется *следующим* узлом в поперечном обходе дерева, а не *предыдущим*? Можно было бы также рассмотреть другие подходы. Например, если узел, который должен быть удален, содержит нулевую левую связь, почему бы просто не сделать его правый дочерний узел новым корнем вместо того, чтобы использовать узел с наименьшим ключом в правом поддереве? Предлагались другие, аналогичные, модификации базовой процедуры удаления. К сожалению, всем им присущ один и тот же недостаток: остающееся после удаления дерево не является произвольным, даже если до этого оно было таковым. Более того, было показано, что программа 12.16 склонна оставлять дерево слегка несбалансированным (средняя высота пропорциональна  $\sqrt{N}$ ), если дерево подвергается большому количеству произвольных пар операций удаления-вставки (см. упражнение 12.84).

Упомянутые различия могут быть не заметны в реальных приложениях, если только *N* не очень велико. Тем не менее, подобного рода сочетание элегантного алгорит-

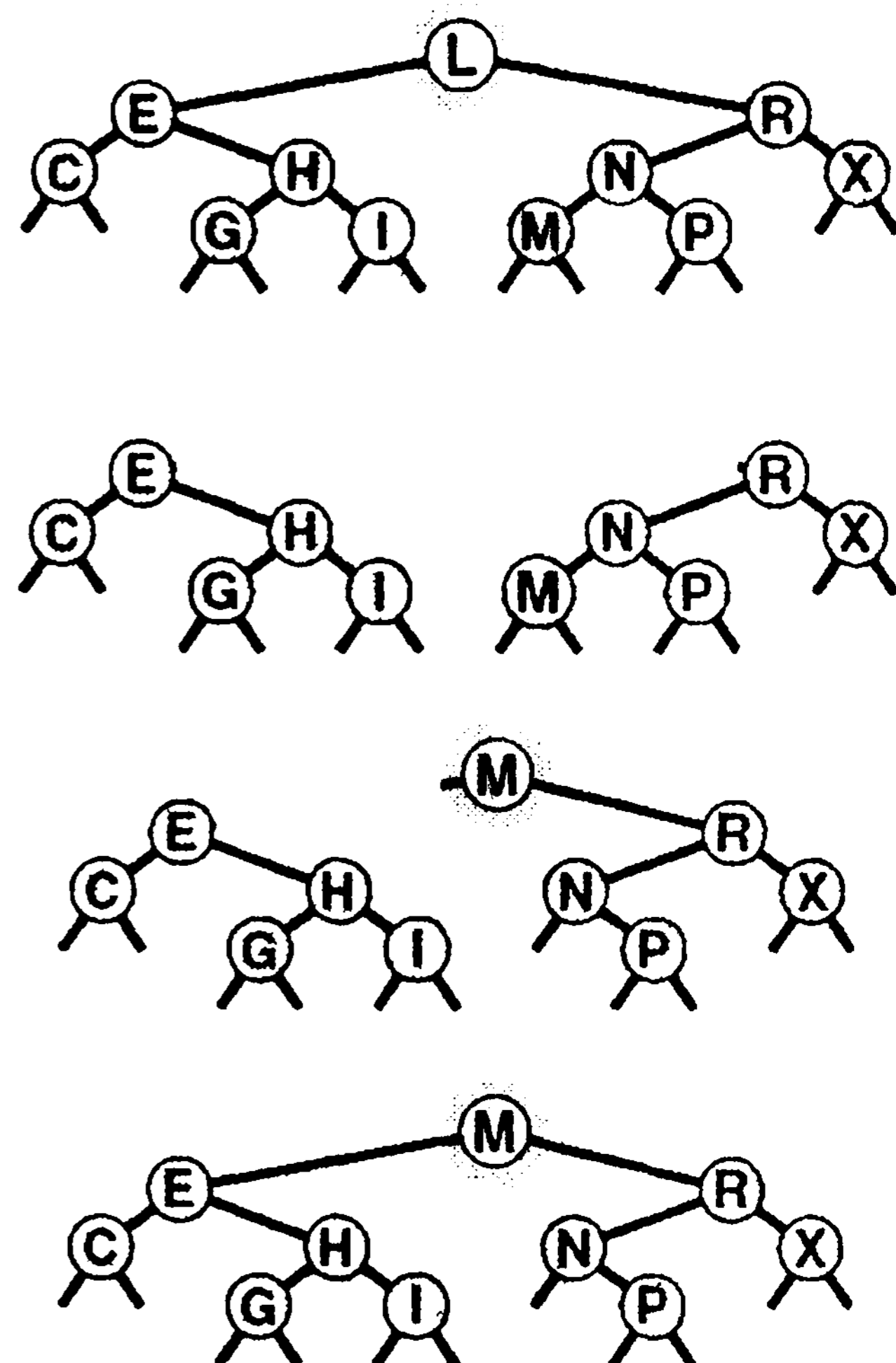


РИСУНОК 12.18 УДАЛЕНИЕ КОРНЯ В BST-ДЕРЕВЕ

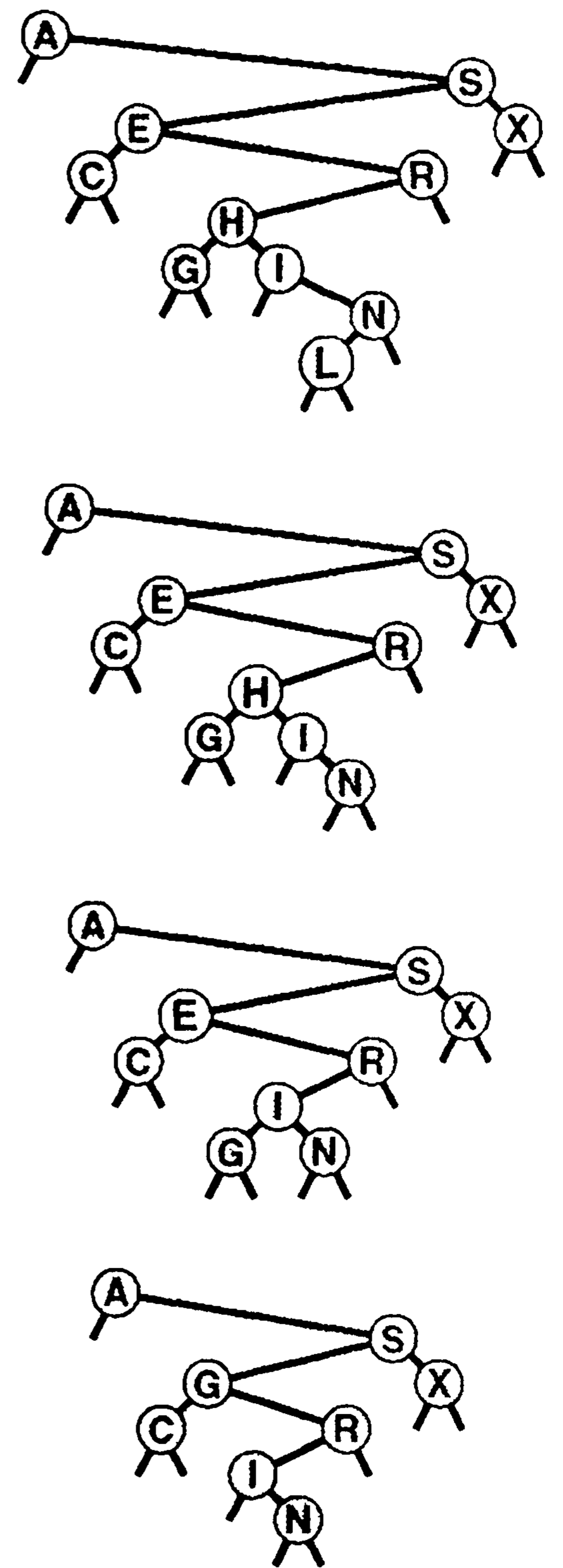
На этой схеме показан результат (внизу) удаления корня из примера BST-дерева (вверху). Вначале мы удаляем узел, оставляя два поддерева (второй сверху рисунок). Затем мы разделяем правое поддерево на части для помещения его наименьшего элемента в корень (третий сверху рисунок), оставляя левую связь указывающей на пустое поддерево. И, наконец, мы заменяем эту связь связью с левым поддеревом исходного дерева (нижний рисунок).

ма с нежелательными характеристиками производительности является неудовлетворительным. В главе 13 будут рассматриваться два различных способа исправления этой ситуации.

Для алгоритмов поиска типично, когда для удаления требуются более сложные реализации, нежели для поиска. Значения ключей играют сложную роль в формировании структуры, поэтому удаление ключа может быть сопряжено со сложными исправлениями. Одна из возможных альтернатив — использование "ленивой" стратегии удаления, оставляющей удаленные узлы в структуре данных, но помечающей их как "удаленные", которые будут игнорироваться при поиске. В реализации поиска в программе 12.8 эту стратегию можно реализовать за счет пропуска проверки на равенство для таких узлов. Необходимо убедиться, что большие количества помеченных узлов не ведут к излишним затратам времени или памяти, но если удаления выполняются не слишком часто, дополнительные затраты могут не играть особой роли. Помеченные узлы можно было бы использовать в будущих вставках, когда это удобно (например, это можно было бы легко сделать для узлов в нижней части дерева). Или же можно было бы периодически перестраивать всю структуру данных, отбрасывая помеченные узлы. Подобные соображения применимы к любой структуре данных, сопряженной с вставками и удалениями, а не только к таблицам символов.

Эта глава завершается рассмотрением реализации операции *remove* с задействованием дескрипторов и операции *join* для реализаций АДТ таблицы символов, которые используют BST-деревья. Мы предполагаем, что дескрипторы — это ссылки, и опускаем дальнейшие рассуждения на тему формирования пакетов, чтобы можно было сосредоточиться на двух базовых алгоритмах.

Основная сложность в реализации функции для удаления узла с данным дескриптором (связью) та же, что имела место в случае связных списков: необходимо изменить указатель в структуре, который указывает на удаляемый узел. Существует, по меньшей мере, четыре способа решения упомянутой проблемы. Во-первых, в каждый узел дерева можно добавить третью связь, указывающую на его родительский узел. Недостаток метода заключается в том, что поддерживать дополнительные связи весьма обременительно, как уже отмечалось неоднократно. Во-вторых, можно использовать ключ в элементе для выполнения поиска в дереве, прекращая его после нахождения соответствующего указателя. Недостаток этого подхода в том, что усредненная позиция узла находится в нижней части дерева и, следовательно, этот подход сопряжен с необязательным перемещением по



**РИСУНОК 12.19 УДАЛЕНИЕ УЗЛА ИЗ BST-ДЕРЕВА**

Эта последовательность отображает результат удаления узлов с ключами *L*, *H* и *E* из BST-дерева, показанного на верхнем рисунке. Вначале *L* просто удаляется, поскольку он расположен внизу. Затем *H* заменяется на его правый дочерний узел *I*, поскольку левый дочерний узел *I* пуст. Наконец, *E* заменяется своим потомком *G*.

дереву. В-третьих, можно воспользоваться ссылкой или указателем на указатель узла в качестве дескриптора. Этот метод подходит в языках C++ и C, но не во множестве других языков. В-четвертых, можно прибегнуть к "ленивому" подходу, помечая удаленные узлы и периодически перестраивая структуру данных, как описывалось несколько ранее.

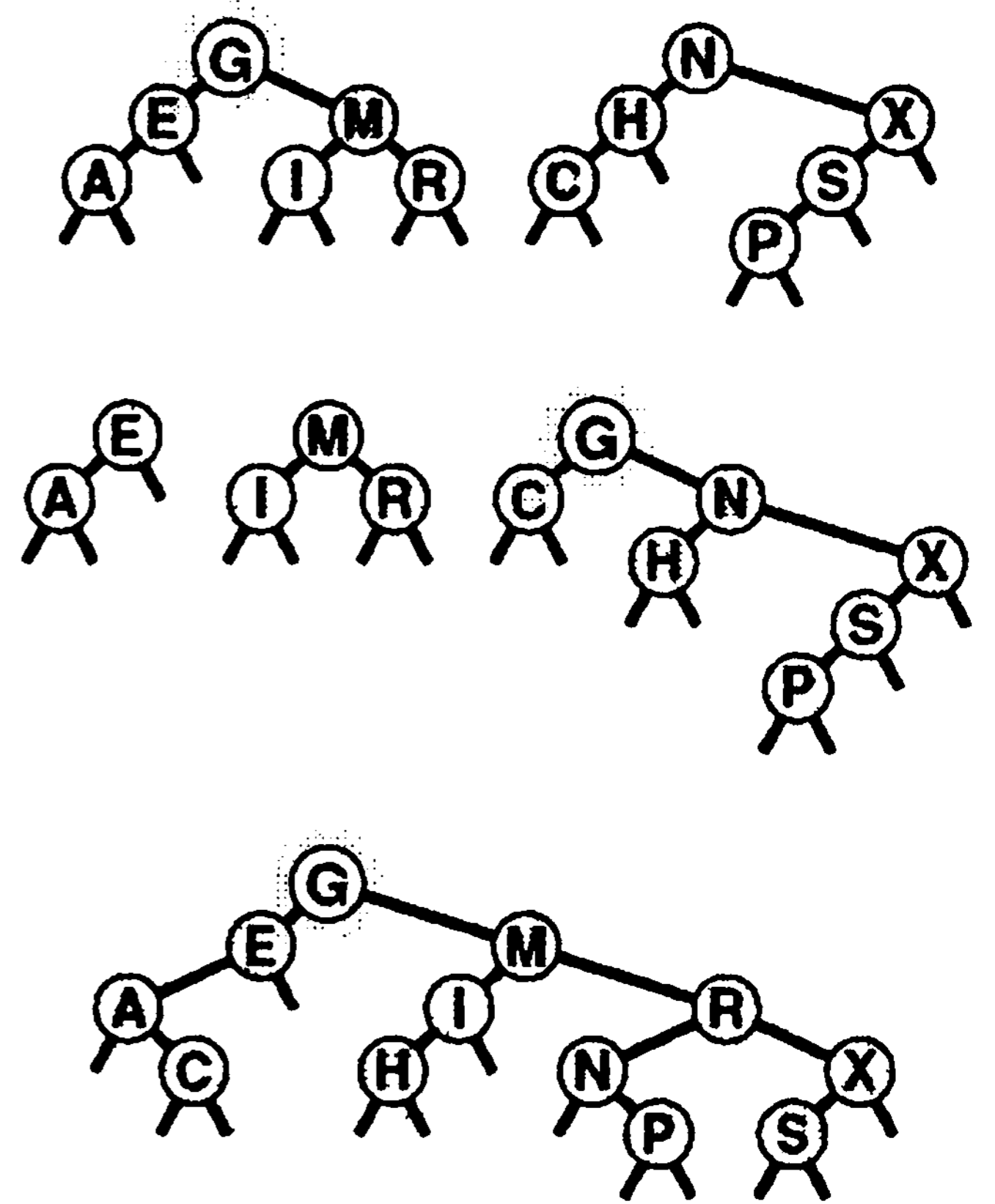
Последняя операция с АДТ таблиц символов, которую мы рассмотрим — операция *join*. В реализации с использованием BST-дерева это сводится к объединению двух деревьев. Как объединить два дерева бинарного поиска в одно? Существуют различные алгоритмы для выполнения этой задачи, но каждому из них присущи определенные недостатки. Например, можно было бы обойти первое BST-дерево, вставляя каждый из его узлов во второе BST-дерево (этот алгоритм является однонаправленным: функция *insert* во втором BST-дереве используется как параметр функции обхода первого BST-дерева). Время выполнения подобного решения не линейно, поскольку для каждой вставки может потребоваться линейное время. Другой подход связан с обходом обоих BST-деревьев с целью помещения элементов в массив, объединения их и затем построения нового BST-дерева. Время выполнения этой операции может быть линейным, но это также связано с потенциально большим массивом.

Программа 12.17 — компактная, линейная по затратам времени рекурсивная реализация операции *join*. Вначале мы вставляем корень первого BST-дерева во второе BST-дерево, используя метод вставки в корень. Эта операция дает два поддерева, ключи которых заведомо меньше этого корня, и два поддерева, ключи которых заведомо больше этого корня, поэтому требуемый результат получается путем объединения (рекурсивного) первой пары в левое поддерево корня, а второй пары — в правое поддерево корня (!). Каждый узел может быть корневым максимум при одном рекурсивном вызове, поэтому общее время определяется линейной зависимостью.

### Программа 12.17 Объединение двух BST-деревьев

Если одно из BST-деревьев пустое, второе будет представлять результат. В противном случае два BST-дерева объединяются путем выбора (произвольного) корня первого дерева в качестве результирующего корня, вставки этого корня в корень второго дерева, а затем (рекурсивного) объединения пары левых поддеревьев и пары правых поддеревьев.

```
private:
    link joinR(link a, link b)
    {
        if (b == 0) return a;
```



**РИСУНОК 12.20 ОБЪЕДИНЕНИЕ ДВУХ BST-ДЕРЕВЬЕВ**

На этой схеме показан результат (внизу) объединения двух примеров BST-деревьев (вверху). Вначале мы вставляем корень *G* первого дерева во второе дерево, используя вставку в корень (второй сверху рисунок). У нас остаются два поддерева, ключи которых меньше *G*, и два поддерева, ключи которых больше *G*. Объединение обеих пар (рекурсивно) обеспечивает конечный результат (рисунок внизу).



```

    if (a == 0) return b;
    insertT(b, a->item);
    b->l = joinR(a->l, b->l);
    b->r = joinR(a->r, b->r);
    delete a; return b;
}
public:
    void join(ST<Item, Key>& b)
    { head = joinR(head, b.head); }

```

Пример показан на рис. 12.20. Подобно удалению, этот процесс асимметричен и может приводить к деревьям, которые не являются хорошо сбалансированными, однако простое решение проблемы обеспечивает рандомизация, как будет показано в главе 13. Обратите внимание, что в наихудшем случае количество сравнений, использованных для выполнения операции *join*, должно, по крайней мере, определяться линейной зависимостью; иначе можно было бы разработать алгоритм сортировки, в котором присутствует менее  $M \lg N$  сравнений, применяя такой подход, как восходящая сортировка слиянием (см. упражнение 12.88).

Мы не включили в программу код, необходимый для поддержки полей счетчика в узлах BST-дерева во время преобразований для операций *join* и *remove*, что необходимо в приложениях, где требуется поддерживать также и операцию *select* (программа 12.14). Концептуально эта задача проста, однако требует определенных усилий. Один из стандартных способов ее выполнения — реализация небольшой вспомогательной процедуры, которая устанавливает значение поля счетчика в узле на единицу больше, чем сумма полей счетчиков в его дочерних узлах, а затем вызов полученной процедуры для каждого узла, связи которого изменяются. В частности, это можно выполнить для обоих узлов в *rotL* и *rotR* в программе 12.12, что достаточно для выполнения преобразований в программах 12.13 и 12.15, поскольку они преобразуют деревья исключительно путем ротаций. Для *joinLR* и *removeR* в программе 12.16 и *join* в программе 12.17 достаточно вызвать процедуру обновления счетчика узлов для возвращаемого узла непосредственно перед оператором *return*.

Базовые операции *search*, *insert* и *sort* для BST-деревьев легко реализовать и выполнить даже при весьма незначительной случайности в последовательности операций, поэтому BST-деревья широко используются применительно к динамическим таблицам символов. Они допускают также простые рекурсивные решения для поддержки других видов операций, как было показано в этой главе на примере операций *select*, *remove* и *join*, и как будет показано во многих примерах далее в книге.

Несмотря на всю полезность, существует два основных недостатка использования BST-деревьев в приложениях. Во-первых, они требуют существенного дополнительного объема памяти для связей. Часто мы считаем, что ссылки и записи имеют практически одинаковые размеры (скажем, одно машинное слово) — если это так, реализация с использованием BST-дерева использует две трети выделенного для нее объема памяти под ссылки и только одну треть под ключи. Данный эффект менее важен в приложениях с большим количеством записей и более важен в средах, в которых указатели велики. Если же память играет первостепенную роль, применению BST-деревьев можно предпочесть один из методов хеширования с открытой адресацией, описанных в главе 14.

Второй недостаток использования BST-деревьев — возможность того, что деревья могут стать плохо сбалансированными и в результате снижать производительность. В главе 13 исследуются еще несколько подходов к обеспечению гарантированной производительности. При наличии достаточного объема памяти под связи, эти алгоритмы делают BST-деревья весьма привлекательными в качестве основы для реализации АДТ таблиц символов, поскольку обеспечивают гарантированно высокую производительность для большого набора полезных операций с АДТ.

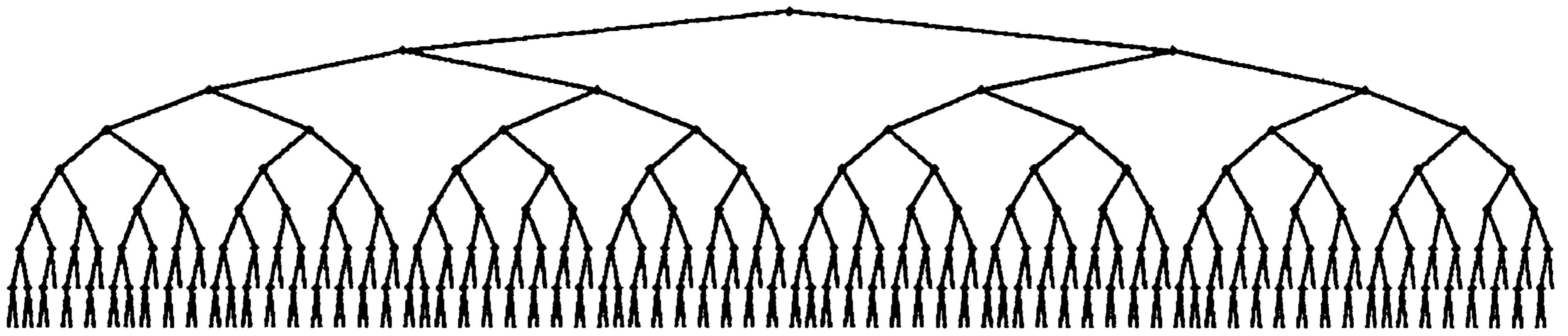
### Упражнения

- ▷ 12.78 Реализуйте нерекурсивную функцию BST-дерева (см. программу 12.14).
- ▷ 12.79 Нарисуйте BST-дерево, образованное в результате вставки элементов с ключами `EASYQUESTION` в первоначально пустое дерево и последующего удаления `Q`.
- ▷ 12.80 Нарисуйте BST-дерево, образованное в результате вставки элементов с ключами `EASY` в первоначально пустое дерево, вставки элементов с ключами `QUESTION` в другое первоначально пустое дерево и последующего объединения результатов.
- 12.81 Реализуйте нерекурсивную функцию `remove` для BST-дерева (см. программу 12.16).
- 12.82 Реализуйте версию операции `remove` для BST-деревьев (программа 12.16), которая удаляет *все* узлы дерева, имеющие ключи, равные данному.
- 12.83 Измените реализации таблиц символов, основанные на BST-дереве, чтобы они поддерживали дескрипторы клиентских элементов (см. упражнение 12.7); добавьте реализации деструктора, конструктора копирования и перегруженной операции присваивания (см. упражнение 12.6); добавьте операции `remove` и `join`; воспользуйтесь программой-драйвером из упражнения 12.22 для проверки своего интерфейса и реализации АДТ первого класса таблицы символов.
- 12.84 Экспериментально определите увеличение высоты BST-дерева при выполнении длинной последовательности чередующихся операций вставки и удаления в произвольном дереве с  $N$  узлами.  $N = 10, 100$  и  $1000$ , а для каждого значения  $N$  выполняется до  $10^2$  пар вставок-удалений.
- 12.85 Реализуйте версию функции `remove` (см. программу 12.16), которая принимает произвольное решение относительно замены узла, который должен быть удален, узлом-предком или узлом-потомком в дереве. Выполните для этой версии серию экспериментов, описанных в упражнении 12.84.
- 12.86 Реализуйте версию функции `remove`, которая использует рекурсивную функцию для перемещения узла, который должен быть удален, в нижнюю часть дерева при помощи ротации, подобной вставке в корень (программа 12.13). Нарисуйте дерево, образованное в результате удаления программой корня из полного дерева, состоящего из 31 узла.
- 12.87 Выполните эксперименты для определения увеличения высоты BST-дерева при многократной вставке элемента в корень дерева, образованного в результате объединения поддеревьев корня в произвольном дереве, состоящем из  $N$  узлов, причем  $N = 10, 100$  и  $1000$ .
- 12.88 Реализуйте версию восходящей сортировки слиянием, основанной на операции `join`. Начните с помещения ключей в  $N$  одноузловых деревьев, затем объедините одноузловые деревья в пары для получения  $N/2$  двухузловых деревьев, далее объедините двухузловые деревья для получения  $N/4$  четырехузловых деревьев и т.д.
- 12.89 Реализуйте версию функции `join` (см. программу 12.17), которая принимает произвольное решение относительно использования корня первого или корня второго дерева в качестве корня результирующего дерева. Прodelайте для этой версии серию экспериментов, описанных в упражнении 12.87.

## Сбалансированные деревья

Описанные в предыдущей главе алгоритмы, использующие деревья бинарного поиска (BST-деревья) успешно работают для широкого множества приложений, однако их производительность существенно снижается в худших случаях. Более того, как это ни прискорбно, на практике именно худший случай стандартного алгоритма с использованием BST-дерева наподобие быстрой сортировки встречается чаще всего, причем когда пользователь не ожидает этого. Уже упорядоченные файлы, файлы с большим количеством дублированных ключей, упорядоченные в обратном порядке файлы или файлы с любым большим сегментом, имеющим простую структуру, могут приводить к тому, что время построения BST-дерева определяется квадратичной, а время поиска — линейной зависимостью.

В идеальном случае можно было бы сохранять деревья полностью сбалансированными, подобно дереву, показанному на рис. 13.1. Эта структура соответствует бинарному поиску и, следовательно, все поиски могут быть выполнены с использованием менее  $N + 1$  сравнений, но при этом поддержка динамических вставок и удалений сопряжена с большими затратами. Высокая производительность поиска гарантируется для любого BST-дерева, в котором все внешние узлы расположены в одном, в крайнем случае в двух, нижних уровнях. Существует большое множество таких BST-деревьев, поэтому в плане поддержки сбалансированности дерева имеется некоторая свобода. Если нас устраивают деревья, близкие к оптимальным, возможности еще больше расширяются. Например, существует очень много BST-деревьев, высота которых не превышает  $2 \lg N$ . Если допустимо смягчить стандарт, но при этом



**РИСУНОК 13.1 БОЛЬШОЕ ПОЛНОСТЬЮ СБАЛАНСИРОВАННОЕ BST-ДЕРЕВО**

*Все внешние узлы этого BST-дерева располагаются в одном из двух уровней, и для выполнения любого поиска требуется столько же сравнений, сколько использовалось бы при поиске этого же ключа методом бинарного поиска (если бы элементы хранились в упорядоченном массиве). Цель применения алгоритма с использованием сбалансированного дерева — сохранение BST-дерева максимально сбалансированным при сохранении эффективности вставки, удаления и других операций с АДД словаря.*

гарантировать, что алгоритмы будут строить только такие BST-деревья, можно избежать снижения производительности для худшего случая, что было бы желательно в реальных приложениях, работающих с динамическими структурами данных. При этом производительность для среднего случая также увеличивается.

Один из подходов к повышению степени сбалансированности в BST-деревьях связан с периодическим явным выполнением их повторной балансировки. Действительно, используя рекурсивный метод, продемонстрированный в программе 13.1, большинство BST-деревьев можно полностью сбалансировать, затратив на это время, которое определяется линейной зависимостью (см. упражнение 13.4). Скорее всего, такая повторная балансировка повысит производительность для случайных ключей, но не обеспечит гарантию против квадратичной зависимости производительности выполнения операций в динамической таблице символов для худшего случая. С одной стороны, между операциями повторной балансировки время вставки для данной последовательности ключей может возрасти в квадратичной зависимости от длины последовательности; с другой стороны, повторную балансировку крупных деревьев нежелательно выполнять слишком часто, поскольку для выполнения каждой операции балансировки требуется время, которое, по меньшей мере, линейно зависит от размера дерева. Необходимость отыскания компромисса в данной ситуации затрудняет использование глобальной повторной балансировки для гарантирования высокой производительности в динамических BST-деревьях. Во всех рассматриваемых в дальнейшем алгоритмах во время обхода дерева выполняются последовательно нарастающие локальные операции, которые совместно увеличивают сбалансированность всего дерева, хотя при этом никогда не приходится выполнять обход всех узлов подобно тому, как это имеет место в программе 13.1.

### **Программа 13.1 Балансировка BST-дерева**

Используя функцию разделения на части **partR** из программы 12.15, эта рекурсивная функция приводит BST-дерево в полностью сбалансированное состояние при затратах времени, которые определяются линейной зависимостью. Разделение на части выполняется с целью помещения среднего узла в корень, а затем это же выполняется для поддеревьев.

```
void balanceR(link& h)
{
    if ((h == 0) || (h->N == 1)) return;
    partR(h, h->N/2);
    balanceR(h->l);
    balanceR(h->r);
}
```

Задача обеспечения гарантированной производительности для реализаций таблиц символов, основанных на использовании BST-деревьев, — превосходный повод для исследования того, что именно подразумевается под гарантированной производительностью. Будут показаны решения этой задачи, являющиеся типичными примерами трех базовых подходов к обеспечению гарантированной производительности при разработке алгоритмов: *рандомизации*, *амортизации* и *оптимизации*. А теперь давайте по очереди кратко рассмотрим каждый из этих подходов.

При использовании *рандомизованного* алгоритма принятие случайного решения встроено в сам алгоритм, что радикально уменьшает вероятность возникновения худшего случая сценария (независимо от входного массива данных). Мы уже видели типичный пример такого подхода, когда случайный элемент использовался в качестве разделяющего в алгоритме быстрой сортировки. В разделах 13.1 и 13.5 исследуются *рандомизованные BST-деревья* и *списки пропусков* — два простых способа использования рандомизации в реализациях таблиц символов для увеличения эффективности всех операций с АД таблицы символов. Эти алгоритмы просты и широко используются, однако они оставались неисследованными в течение нескольких десятилетий (см. *раздел ссылок*). Аналитическое доказательство эффективности этих алгоритмов не является элементарным, но сами алгоритмы просты для понимания, как и их реализация и практическое применение.

*Амортизационный* подход заключается в однократном выполнении дополнительных действий во избежание выполнения большего объема работы впоследствии, чтобы можно было обеспечить гарантированный верхний предел усредненной стоимости одной операции (общей стоимости всех операций, разделенной на количество операций). В разделе 13.2 рассматривается *расширенное* дерево — вариант BST-дерева, которое можно использовать для обеспечения таких гарантий при реализации таблиц символов. Разработка этого метода послужила одним из стимулов разработки концепции *амортизации* (см. *раздел ссылок*). Этот алгоритм — вполне очевидное расширение метода вставок в корень, рассмотренного в главе 12, но аналитическое обоснование предельных значений производительности является достаточно сложным.

*Оптимизационный* подход заключается в обеспечении гарантированной производительности каждой операции. Были разработаны различные методы, использующие этот подход, часть из которых восходит к 60-м годам. Эти методы требуют хранения в деревьях некоторой структурной информации, и, как правило, их реализация связана с определенной сложностью. В этой главе рассматриваются две простые абстракции, которые не только делают реализацию понятной, но и обеспечивают почти оптимальные верхние пределы затрат. В заключение, после исследования реализаций АД таблиц символов с использованием каждого из этих трех подходов, обеспечивающих гарантированную высокую производительность, в главе приводится сравнение

характеристик производительности. Помимо различий, обусловленных различной природой гарантий производительности, обеспечиваемых тем или иным алгоритмом, каждый из методов характеризуется затратами (сравнительно небольшими) времени или памяти, сопряженными с упомянутыми гарантиями; разработка АДД действительно оптимально сбалансированного дерева все еще остается предметом будущих исследований. Тем не менее, все рассматриваемые в этой главе алгоритмы весьма важны и способны обеспечить быстро выполняющиеся реализации операций *search* и *insert* (и ряда других операций для АДД таблицы символов) в динамических таблицах символов, предназначенных для разнообразных приложений.

## Упражнения

- 13.1 Реализуйте эффективную функцию, выполняющую повторную балансировку BST-деревьев, которые не содержат поле счетчика в своих узлах.
- 13.2 Модифицируйте стандартную функцию вставки в BST-дерево, представленную в программе 12.8, чтобы ее можно было использовать в программе 13.1 для выполнения повторной балансировки дерева каждый раз, когда количество элементов в таблице символов достигает числа, равного степени 2. Сравните время выполнения этой программы с временем выполнения программы 12.8 при выполнении задач (i) построения дерева из  $N$  случайных ключей и (ii) поиска  $N$  случайных ключей в результирующем дереве, для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 13.3 Оцените количество сравнений, используемых программой из упражнения 13.2 при вставке возрастающей последовательности  $N$  ключей в таблицу символов.
- 13.4 Покажите, что для вырожденного дерева время выполнения программы 13.1 пропорционально  $MgN$ . Затем приведите максимально неоптимальный вариант дерева, при котором время выполнения программы определяется линейной функцией.
- 13.5 Модифицируйте стандартную функцию вставки в BST-дерево, приведенную в программе 12.8, чтобы она делила примерно пополам любой встреченный узел, который в одном из своих поддеревьев содержит менее одной четверти всех узлов. Сравните время выполнения этой программы с временем выполнения программы 12.8 при выполнении задач (i) построения дерева из  $N$  случайных ключей и (ii) поиска  $N$  случайных ключей в результирующем дереве, для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 13.6 Оцените количество сравнений, используемых программой из упражнения 13.5 при вставке возрастающей последовательности  $N$  ключей в таблицу символов.
- 13.7 Расширьте реализацию, созданную в упражнении 13.5, чтобы она так же выполняла повторную балансировку при выполнении функции *remove*. Экспериментально определите, возрастает ли высота дерева при выполнении длиной последовательности чередующихся случайных вставок и удалений в случайном дереве, состоящем из  $N$  узлов при  $N = 10, 100$  и  $1000$  и при выполнении  $N^2$  пар вставок-удалений для каждого  $N$ .

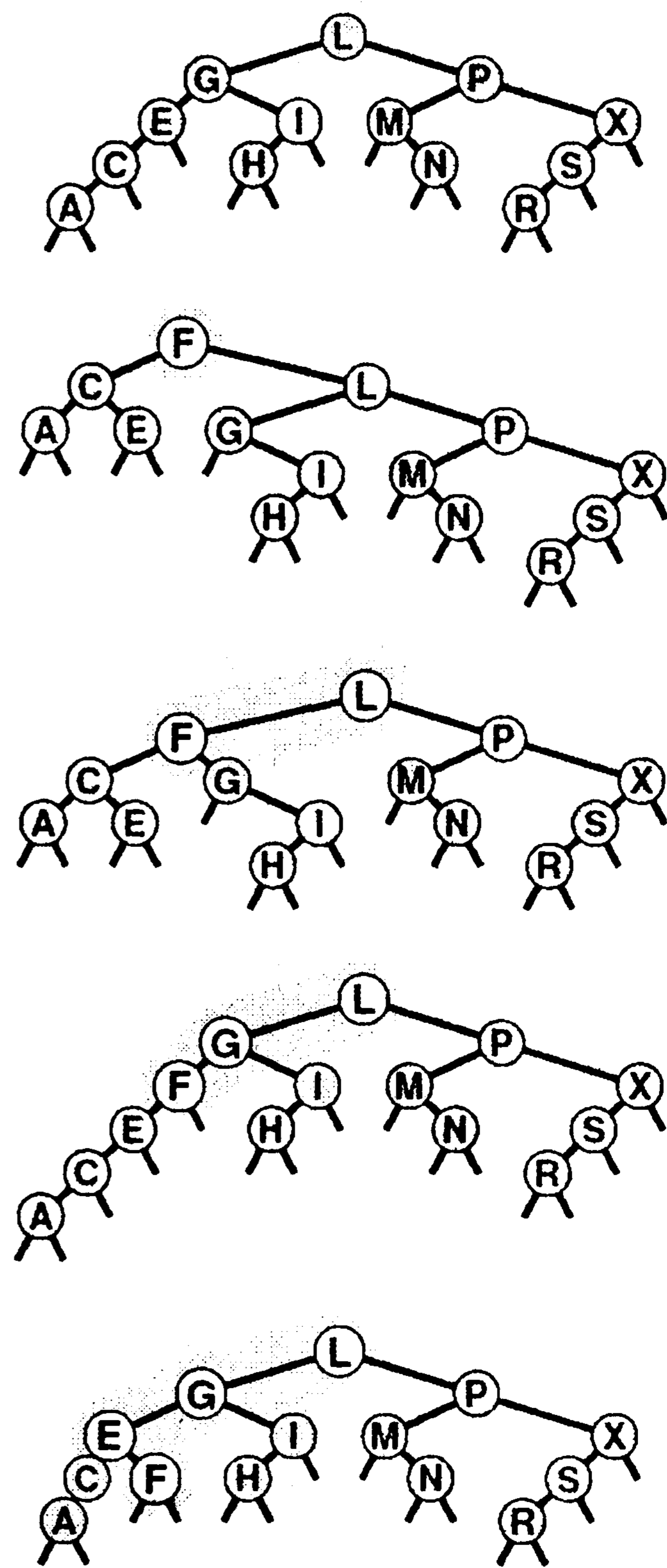
## 13.1 Рандомизованные BST-деревья

Чтобы проанализировать затраты для случая усредненной производительности при работе с BST-деревьями, было сделано допущение, что элементы вставляются в случайном порядке (см. раздел 12.6). Применительно к алгоритму с использованием BST-дерева главное следствие этого допущения заключается в том, что каждый узел дерева с равной вероятностью может оказаться корневым, причем это же справедливо и по отношению к поддеревьям. Как это ни удивительно, "случайность" можно включить в алгоритм, чтобы это свойство сохранялось без *каких-либо* допущений относительно порядка вставки элементов. Идея весьма проста: при вставке нового узла в дерево, состоящее из  $N$  узлов, вероятность появления нового узла в корне должна быть равна  $1/(N + 1)$ , поэтому мы просто принимаем рандомизованное решение использовать вставку в корень с этой вероятностью. В противном случае мы рекурсивно используем метод для вставки новой записи в левое поддерево, если ключ записи меньше ключа в корне, и в правое поддерево — если он больше. Программа 13.2 содержит реализацию этого метода.

Если использовать нерекурсивный подход, выполнение рандомизованной вставки эквивалентно выполнению стандартного поиска ключа с принятием на каждом шаге рандомизованного решения о том, продолжить ли поиск или прервать его и выполнить вставку в корень. Таким образом, как показано на рис. 13.2, новый узел может быть вставлен в любое место на пути поиска. Это простое вероятностное объединение стандартного алгоритма BST-дерева с методом вставки в корень обеспечивает гарантированную в смысле вероятности производительность.

**Лемма 13.1** *Построение рандомизованного BST-дерева эквивалентно построению стандартного BST-дерева из случайно переставленных в исходном состоянии ключей. Для конструирования рандомизованного BST-дерева из  $N$  элементов используется около  $2N \ln N$  сравнений (независимо от порядка вставки элементов), а для выполнения поиска в таком дереве требуется приблизительно  $2 \ln N$  сравнений.*

Каждый элемент с равной вероятностью может быть корнем дерева, и это свойство сохраняется и для обоих поддеревьев. Первая часть этого утверждения подтверждается самим методом конструирования, но для подтверждения того, что



**РИСУНОК 13.2 ВСТАВКА В РАНДОМИЗОВАННОЕ BST-ДЕРЕВО**

*Новая запись в рандомизованном BST-дереве может располагаться в любом месте пути поиска записи, в зависимости от результата рандомизованных решений, принятых во время поиска. На этом рисунке показаны возможные местоположения записи, содержащей ключ  $F$ , при ее вставке в пример дерева (верхний рисунок).*

метод вставки в корень сохраняет случайность поддеревьев, требуется тщательное вероятностное обоснование (см. раздел ссылок).

Различие между производительностью в усредненном случае для рандомизованных и стандартных BST-деревьев очень невелико, но имеет большое значение. Усредненные затраты в обоих случаях одинаковы (хотя для рандомизованных деревьев коэффициент пропорциональности несколько выше), однако для случая стандартных деревьев результат зависит от допущения о представлении элементов к вставке в случайном порядке их ключей (их вставка в любом порядке равновероятна). Это допущение не справедливо во многих практических приложениях, и, следовательно, рандомизованный алгоритм примечателен тем, что позволяет избавиться от такого допущения и вместо этого исходить из законов распределения вероятностей в генераторе случайных чисел. При вставке элементов в порядке следования их ключей, в обратном порядке или *любом другом порядке*, BST-дерево все равно будет рандомизованным.

### Программа 13.2 Вставка в рандомизованное BST-дерево

Эта функция принимает рандомизованное решение о том, использовать ли метод вставки в корень программы 12.13 или стандартный метод вставки программы 12.8. В рандомизованном BST-дереве каждый из узлов с равной вероятностью является корнем; поэтому, помещая новый узел в корень дерева размера  $N$  с вероятностью  $1/(N + 1)$ , мы получаем рандомизованное дерево.

**private:**

```
void insertR(link& h, Item x)
{ if (h == 0) { h = new node(x); return; }
  if (rand() < RAND_MAX/(h->N+1))
    { insertT(h, x); return; }
  if (x.key() < h->item.key())
    insertR(h->l, x);
  else insertR(h->r, x);
  h->N++;
}
```

**public:**

```
void insert(Item x)
{ insertR(head, x); }
```

На рис. 13.3 показано создание рандомизованного дерева для примера набора ключей. Поскольку принимаемые алгоритмом решения являются рандомизованными, вероятно, последовательность деревьев при каждом выполнении алгоритма будет иной. На рис. 13.4 отражается тот факт, что рандомизованное дерево, сконструированное из набора элементов, ключи которых упорядочены по возрастианию, выглядит так же, как стандартное BST-дерево, построенное из элементов, следующих в случайном порядке (сравните с рис. 12.8).

Существует вероятность, что при первой же возможности генератор случайных чисел может привести к неверному решению, обуславливая тем самым создание плохо сбалансированных деревьев, однако эту вероятность можно оценить математически и доказать, что она очень мала.

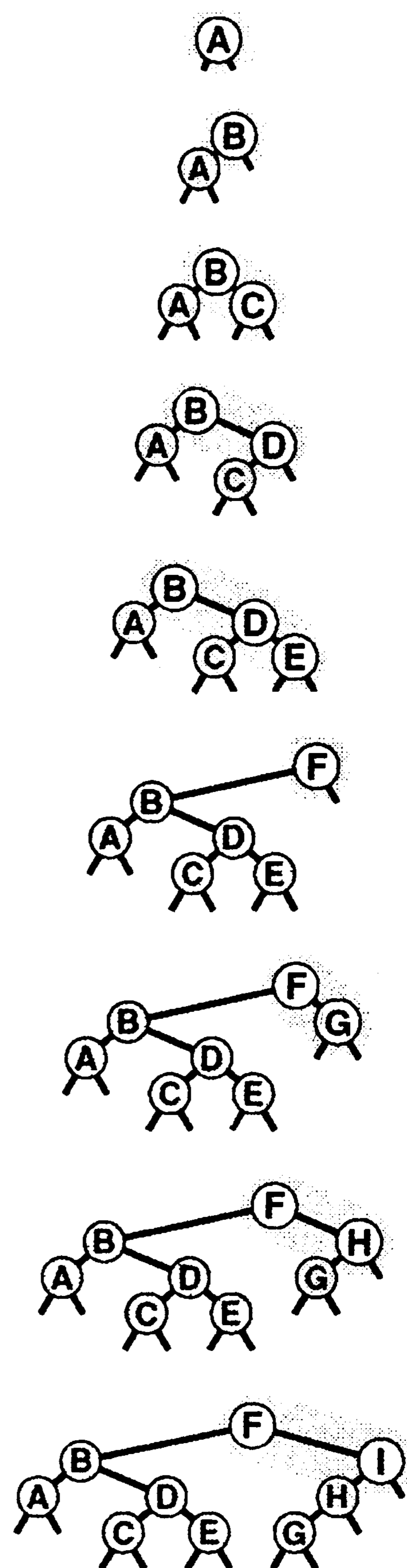
**Лемма 13.2** *Вероятность того, что затраты на создание рандомизованного BST-дерева превышают усредненные затраты в  $\alpha$  раз, меньше  $e^{-\alpha}$ .*



Этот результат совпадает с результатами, полученными из общего решения вероятностных соотношений, которые были разработаны Карпом (Karp) в 1995 г. (см. раздел ссылок).

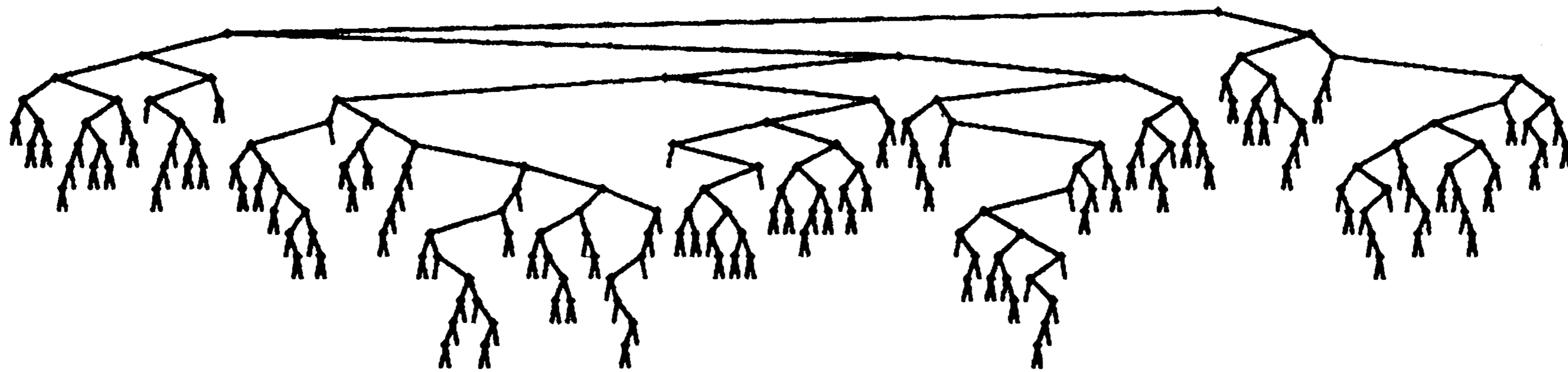
Например, для построения рандомизованного BST-дерева, состоящего из 100000 узлов, требуется около 2.3 миллиона сравнений, но вероятность того, что количество сравнений превысит 23 миллиона, значительно меньше 0.01 процента. Подобная гарантия производительности более чем удовлетворяет практические требования, предъявляемые к обработке реальных наборов данных такого размера. Упомянутую гарантию нельзя обеспечить при использовании стандартного BST-дерева для выполнения такой задачи: например, придется столкнуться с проблемами снижения производительности, если данные в значительной степени упорядочены, что маловероятно для случайных данных, но по множеству причин достаточно часто имеет место при работе с реальными данными.

По тем же соображениям, утверждение, аналогичное лемме 13.2, справедливо и по отношению к времени выполнения быстрой сортировки. Но в данном случае это еще более важно, поскольку отсюда следует, что затраты на поиск в дереве близки к усредненным. Независимо от дополнительных затрат на построение деревьев, стандартную реализацию BST-дерева можно использовать для выполнения операций *search* при затратах, которые зависят только от формы деревьев, при отсутствии вообще каких-либо дополнительных затрат на балансировку. Это свойство важно в типовых приложениях, в которых операции *search* гораздо более многочисленны, нежели любые другие. Например, описанное в предыдущем абзаце BST-дерево, состоящее из 100000 узлов, могло бы содержать телефонный справочник и использоваться для выполнения миллионов поисков. Можно быть почти уверенным, что каждый поиск потребует затрат, которые отличаются от усредненных затрат, равных приблизительно 23 сравнениям, лишь небольшим постоянным коэффициентом. Поэтому на практике можно не беспокоиться, что для большого количества поисков потребуется количество сравнений, приближающееся к 100000, в то время как при использовании стандартных BST-деревьев это было бы весьма актуально.



**РИСУНОК 13.3 ПОСТРОЕНИЕ РАНДОМИЗОВАННОГО BST-ДЕРЕВА**

На этих рисунках показана вставка ключей *A B C D E F G H I* в первоначально пустое BST-дерево методом рандомизованных вставок. Дерево на нижнем рисунке выглядит так же, как если бы оно было построено с применением стандартного алгоритма BST-дерева при вставке этих же ключей в случайном порядке.



**РИСУНОК 13.4 БОЛЬШОЕ РАНДОМИЗОВАННОЕ BST-ДЕРЕВО**

Это BST-дерево является результатом рандомизованной вставки 200 элементов в порядке возрастания их ключей в первоначально пустое дерево. Дерево выглядит так, как если бы оно было построено из ключей, расположенных в случайном порядке (см. рис. 12.8).

Один из главных недостатков рандомизованной вставки — затраты на генерацию случайных чисел в каждом из узлов во время каждой вставки. Высокопроизводительный системный генератор случайных чисел может работать с большой нагрузкой для генерации псевдослучайных чисел с большей степенью случайности, чем требуется для BST-деревьев. Поэтому в определенных реальных ситуациях (например, если допущение о случайном порядке следования элементов *справедливо*) конструирование рандомизованного BST-дерева может оказаться более медленным процессом, чем построение стандартного BST-дерева. Подобно тому как это делалось в случае быстрой сортировки, эти затраты можно снизить, используя числа, которые не являются совершенно случайными, но не требуют больших затрат при генерации и достаточно подобны случайным числам. Тем самым минимизируется возникновение худших случаев BST-деревьев при последовательностях вставок ключей, которые вполне могут встретиться на практике (см. упражнение 13.14).

Еще один потенциальный недостаток рандомизованных BST-деревьев — необходимость наличия в каждом узле поля количества узлов поддеревья данного узла. Дополнительный объем памяти, требуемый для поддержки этого поля, может оказаться чрезмерной платой для больших деревьев. С другой стороны, как было показано в разделе 12.9, это поле может требоваться по ряду других причин — например, для поддержки операции *select* или для обеспечения проверки целостности структуры данных. В подобных случаях рандомизованные BST-деревья не требуют никаких дополнительных затрат памяти и их использование представляется весьма привлекательным.

Основной принцип сохранения случайной структуры деревьев ведет также к эффективным реализациям операций *remove*, *join* и других операций для АТД таблицы символов, обеспечивая при этом создание рандомизованных деревьев.

Для *объединения* дерева, состоящего из  $N$  узлов, с деревом, состоящим из  $M$  узлов, используется базовый метод, описанный в главе 12, за исключением принятия рандомизованного решения о выборе корня исходя из того, что корень объединенного дерева должен выбираться из  $N$ -узлового дерева с вероятностью  $N / (M + N)$ , а из  $M$ -узлового дерева — с вероятностью  $M / (M + N)$ . Программа 13.3 содержит реализацию этой операции.

**Программа 13.3 Комбинация рандомизованных BST-деревьев**

В этой функции используется тот же метод, что и в программе 12.17, за исключением того, что в ней принимается рандомизованное, а не произвольное решение о том, какой узел использовать в качестве корня объединенного дерева, исходя из равной вероятности размещения корня в любом узле. Приватная функция-член **fixN** обновляет **b->N** значением, которое на 1 больше суммы соответствующих полей в поддеревьях (0 для нулевых деревьев).

```
private:
    link joinR(link a, link b)
    {
        if (a == 0) return b;
        if (b == 0) return a;
        insertR(b, a->item);
        b->l = joinR(a->l, b->l);
        b->r = joinR(a->r, b->r);
        delete a; fixN(b); return b;
    }
public:
    void join(ST<Item, Key>& b)
    { int N = head->N;
      if (rand() / (RAND_MAX / (N+b.head->N)+1) < N)
          head = joinR(head, b.head);
      else head = joinR(b.head, head); }
```

Аналогично, произвольное решение заменяется рандомизованным в алгоритме *remove*, как показано в программе 13.4. Этот метод соответствует нерассмотренному варианту реализации удаления узлов в стандартных BST-деревьях, поскольку он приводил бы (при отсутствии рандомизации) к несбалансированным деревьям (см. упражнение 13.21).

**Программа 13.4 Удаление в рандомизованном BST-дереве.**

Мы используем ту же функцию **remove**, которая применялась для стандартных BST-деревьев (см. программу 12.6), но при этом функция **joinLR** заменяется функцией, приведенной здесь, которая принимает рандомизованное, а не произвольное решение о замещении удаленного узла предком или потомком, исходя из того, что каждый узел в результирующем дереве с равной вероятностью может быть корнем. Чтобы правильно поддерживать счетчики узлов, в качестве последнего оператора в функции **removeR** необходимо также включить вызов функции **fixN** (см. программу 13.3) для **h**.

```
link joinLR(link a, link b)
{
    if (a == 0) return b;
    if (b == 0) return a;
    if (rand() / (RAND_MAX / (a->N+b->N)+1) < a->N)
        { a->r = joinLR(a->r, b); return a; }
    else { b->l = joinLR(a, b->l); return b; }
}
```

**Лемма 13.3** *Создание дерева через произвольную последовательность рандомизованных операций вставки, удаления и объединения эквивалентно построению стандартного BST-дерева за счет случайной перестановки ключей в дереве.*

Как и в случае с леммой 13.1, для доказательства этого утверждения требуется тщательный вероятностный анализ (см. раздел ссылок).

Доказательства свойств вероятностных алгоритмов требуют хорошего понимания теории вероятностей, тем не менее, понимание этих доказательств отнюдь не обязательно для программистов, использующих алгоритмы. Осмотрительный программист в любом случае проверит утверждения, подобные лемме 13.2, независимо от того, как они обоснованы (например, убеждаясь в качестве генератора случайных чисел или иных особенностей реализации), и, следовательно, сможет использовать эти методы со знанием дела. Рандомизованные BST-деревья — вероятно, простейший способ поддержки АДТ заполненной таблицы символов при гарантировании почти оптимальной производительности. Именно поэтому они находят применение во многих практических приложениях.

## Упражнения

▷ **13.8** Нарисуйте рандомизованное BST-дерево, образующееся в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево при условии, что плохо выполняющая рандомизацию функция, приводящая к вставке в корень, применяется во всех случаях, когда размер дерева является нечетным.

**13.9** Создайте программу-драйвер, которая 1000 раз выполняет следующий эксперимент для  $N = 10$  и  $100$ : используя программу 13.2, вставляет ключи от  $0$  до  $N - 1$  (в указанном порядке) в первоначально пустое рандомизованное BST-дерево, а затем выводит статистическую функцию  $\chi^2$ , исходя из предположения, что вероятность попадания каждого ключа в корень равна  $1/N$  (см. упражнение 14.5).

○ **13.10** Укажите вероятность попадания ключа **F** в каждую из позиций, показанных на рис. 13.2.

**13.11** Создайте программу вычисления вероятности того, что рандомизованная вставка завершается в одном из внутренних узлов данного дерева для каждого из узлов в пути поиска.

**13.12** Создайте программу вычисления вероятности того, что рандомизованная вставка завершается в одном из внешних узлов данного дерева.

○ **13.13** Реализуйте нерекурсивную версию функции рандомизованной вставки, приведенной в программе 13.2.

**13.14** Нарисуйте рандомизованное BST-дерево, образующееся в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево при использовании версии программы 13.2, в которой выражение, содержащее функцию `rand()`, заменяется проверкой `(111 % h->N) == 3` для принятия решения о применении вставки в корень.

**13.15** Выполните упражнение 13.9 для версии программы 13.2, в которой выражение, содержащее функцию `rand()`, заменяется проверкой `(111 % h->N) == 3` для принятия решения о применении вставки в корень.

**13.16** Приведите последовательность рандомизованных решений, которая привела бы к построению вырожденного дерева (в котором все ключи упорядочены, а левые связи являются нулевыми) из ключей **E A S Y Q U T I O N**. Какова вероятность возникновения этого события?

- 13.17** Может ли *любое* BST-дерево, содержащее ключи **E A S Y Q U T I O N**, быть построено посредством *какой-либо* последовательности принятия рандомизованных решений, если эти ключи вставляются в указанном порядке в первоначально пустое дерево? Обоснуйте свой ответ.
- 13.18** Определите эмпирическим путем среднее значение и стандартное отклонение количества сравнений, используемых для обнаружения попаданий и промахов при поиске в рандомизованном BST-дереве, построенном в результате вставки  $N$  случайных ключей в первоначально пустое дерево, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- ▷ **13.19** Нарисуйте BST-дерево, образующееся в результате использования программы 13.4 для удаления ключа **Q** из дерева, построенного в упражнении 13.4, при использовании проверки  $(111 \% (a \rightarrow N + b \rightarrow N)) < a \rightarrow N$  для принятия решения об объединении с помещением **a** в корень.
- 13.20** Нарисуйте BST-дерево, образующееся в результате вставки элементов с ключами **E A S Y** в первоначально пустое дерево, вставки элементов с ключами **Q U E S T I O N** в другое первоначально пустое дерево и последующего объединения результата за счет использования программы 13.3 с выполнением проверки, описанной в упражнении 13.19.
- 13.21** Нарисуйте BST-дерево, образующееся в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево и последующего применения программы 13.4 для удаления ключа **Q**, при использовании плохо выполняющей рандомизацию функции, которая всегда возвращает 0.
- 13.22** Экспериментально определите увеличение высоты BST-дерева при выполнении длинной последовательности чередующихся вставок и удалений с помощью программ 13.2 и 13.3 в дереве, состоящем из  $N$  узлов, при  $N = 10, 100$  и  $1000$  и при выполнении  $N^2$  пар вставок-удалений для каждого  $N$ .
- **13.23** Сравните результаты, полученные в результате выполнения упражнения 13.22, с результатом удаления и повторной вставки наибольшего ключа в рандомизованном дереве, состоящем из  $N$  узлов, при использовании программ 13.2 и 13.3 для  $N = 10, 100$  и  $1000$  и при выполнении  $N^2$  пар вставок-удалений для каждого  $N$ .
- 13.24** Модифицируйте программу из упражнения 13.22 для определения усредненного количества вызовов функции **rand()**, выполняемого ею для удаления одного элемента.

## 13.2 Расширенные деревья бинарного поиска

В методе вставки в корень, описанном в разделе 12.8, основная цель достигалась путем перемещения вновь вставленного узла в корень дерева с помощью ротации влево и вправо. В этом разделе исследуются способы модификации метода вставки в корень, чтобы ротации также в определенном смысле балансировали дерево.

Вместо того чтобы рассматривать (рекурсивно) единственную ротацию, которая перемещает недавно вставленный узел к вершине дерева, рассмотрим *две* ротации, которые перемещают узел из позиции в одном из дочерних узлов корня к вершине дерева. Вначале выполняется одна ротация, которая делает узел дочерним узлом корня. Затем при помощи еще одной ротации он перемещается в корень. Существует

два принципиально различных случая, в зависимости от того, одинаково ли ориентированы две связи от корня к вставляемому узлу. На рис. 13.5 показан случай, когда ориентации различны; в правой части рис. 13.6 изображен случай, когда ориентации одинаковы. В основе обработки расширенных BST-деревьев лежит наблюдение о существовании альтернативного способа выполнения действий, когда связи от корня к вставляемому узлу ориентированы одинаково: достаточно выполнить две ротации в корне, как показано в правой части рис. 13.6.

*Вставка со скосом (splay insertion)* перемещает вновь вставленные узлы в корень за счет применения трансформаций, показанных на рис. 13.5 (стандартной вставки в корень, когда связи от корня к дочернему узлу в пути поиска имеют различную ориентацию) и в правой части рис. 13.6 (двух ротаций в корне, когда связи от корня к дочернему узлу в пути поиска имеют одинаковую ориентацию). Построенные таким образом BST-деревья являются *расширенными BST-деревьями*. Программа 13.5 содержит рекурсивную реализацию вставки с расширением; пример одиночной вставки приведен на рис. 13.7, а процесс построения примера дерева показан на рис. 13.8. Различие между вставкой с расширением и стандартной вставкой в корень может показаться несущественным, но оно достаточно важно: расширение исключает худший случай квадратичной зависимости времени выполнения, являющийся главным недостатком стандартных BST-деревьев.

### Программа 13.5 Вставка с расширением в BST-деревья

Эта функция отличается от алгоритма вставки в корень программы 12.13 лишь одной существенной особенностью: если путь поиска проходит влево-влево или вправо-вправо, узел перемещается в корень путем двойной ротации от вершины, а не от нижней части (см. рис. 13.6).

Программа проверяет четыре варианта для двух шагов пути поиска от корня и выполняет соответствующие ротации:

*влево-влево:* дважды выполняет ротацию влево в корне;

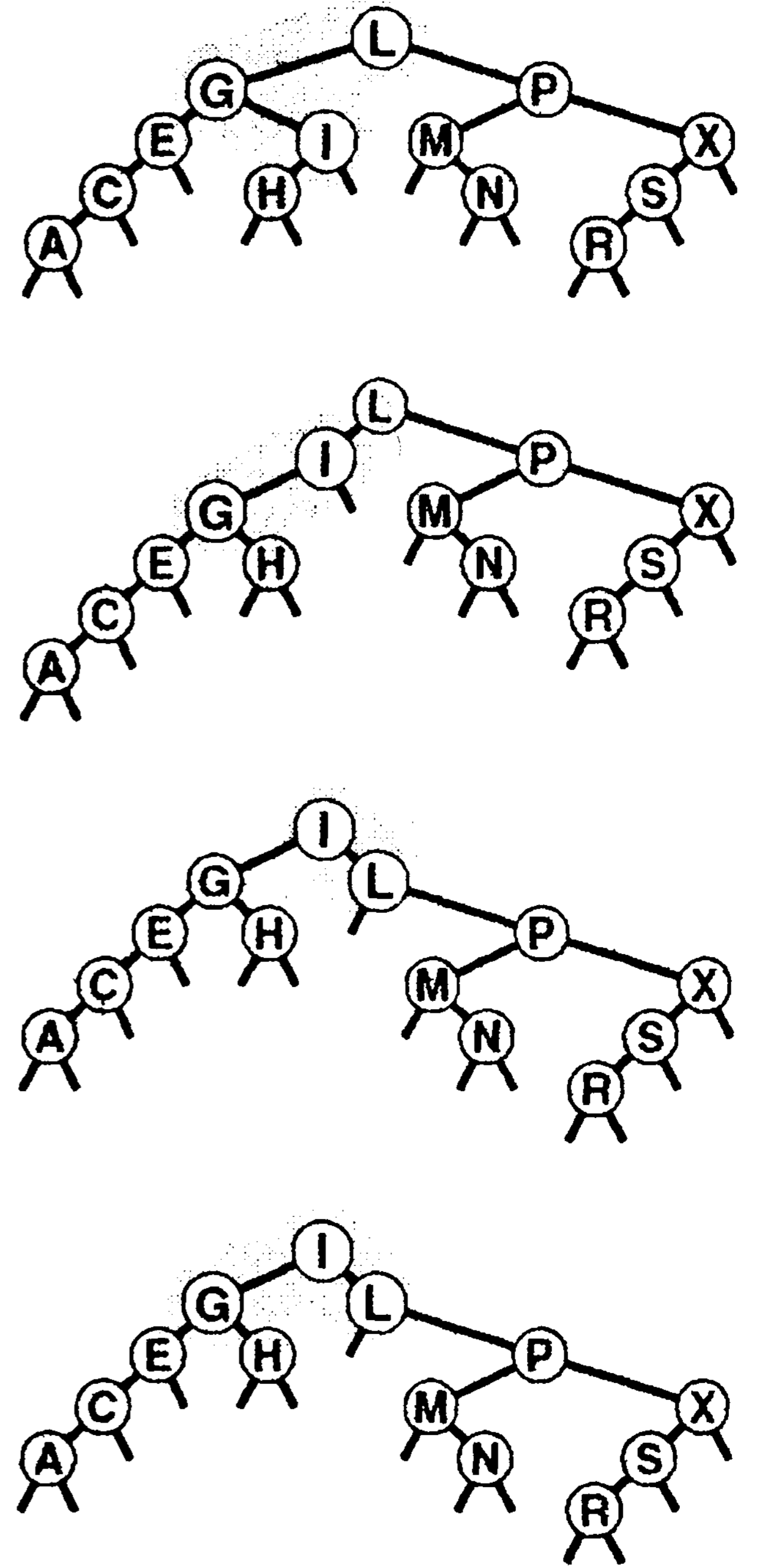
*влево-вправо:* выполняет ротацию влево в левом дочернем узле, а затем вправо в корне;

*вправо-вправо:* дважды выполняет ротацию вправо в корне;

*вправо-влево:* выполняет ротацию вправо в правом дочернем узле, а затем ротацию влево в корне.

**private:**

```
void splay(link& h, Item x)
```



**РИСУНОК 13.5 ДВОЙНАЯ РОТАЦИЯ В BST-ДЕРЕВЕ (ОРИЕНТАЦИИ РАЗЛИЧНЫ)**

*В этом простом дереве (вверху) в результате ротации влево в узле G, за которой следует ротация вправо в узле L, узел I помещается в корень (внизу). Эти ротации могут завершать процесс вставки в стандартном или скошенном BST-дереве.*

```

{
  if (h == 0)
    { h = new node(x, 0, 0, 1); return; }
  if (x.key() < h->item.key())
    { link& hl = h->l; int N = h->N;
      if (hl == 0)
        { h = new node(x, 0, h, N+1); return; }
      if (x.key() < hl->item.key())
        { splay(hl->l, x); rotR(h); }
      else { splay(hl->r, x); rotL(hl); }
      rotR(h);
    }
}
else
{ link &hr = h->r; int N = h->N;
  if (hr == 0)
    { h = new node(x, h, 0, N+1); return; }
  if (hr->item.key() < x.key())
    { splay(hr->r, x); rotL(h); }
  else { splay(hr->l, x); rotR(hr); }
  rotL(h);
}
}
public:
void insert(Item item)
{ splay(head, item); }

```

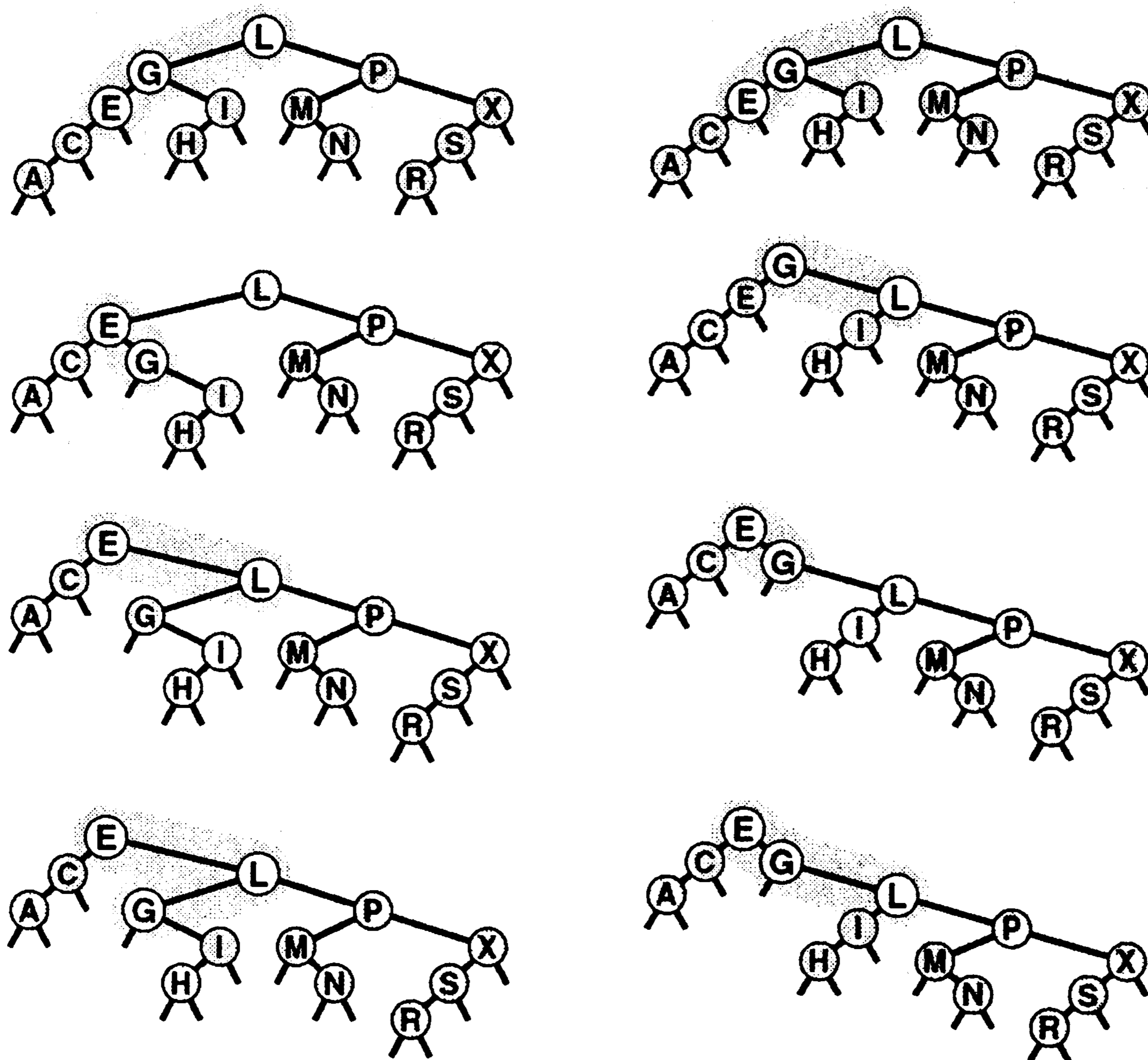


РИСУНОК 13.6 ДВОЙНАЯ РОТАЦИЯ В BST-ДЕРЕВЕ (ОРИЕНТАЦИИ ОДИНАКОВЫ)

Когда обе связи в двойной ротации ориентированы в одном направлении, существуют две возможности. При использовании стандартного метода вставки в корень вначале выполняется ротация в узле, расположенном ниже (слева); при использовании вставки со скосом вначале выполняется вставка в узле, расположенном выше (справа).

**Лемма 13.4** *Количество сравнений, используемых при построении расширенного дерева путем  $N$  вставок в первоначально пустое дерево, равно  $O(N \lg N)$ .*

Это утверждение — следствие более жесткой леммы 13.5, которая будет вскоре рассмотрена.

Подразумевается, что константа в  $O$ -нотации имеет значение 3. Например, для построения BST-дерева, состоящего из 100000 узлов, с использованием вставки с расширением всегда требуется менее 5 миллионов сравнений. Это не гарантирует, что результирующее дерево поиска будет хорошо сбалансировано и что каждая операция будет эффективной, но тем не менее обеспечивает весьма значительную гарантию в отношении общего времени выполнения; на практике фактическое время выполнения, скорее всего, окажется еще меньшей.

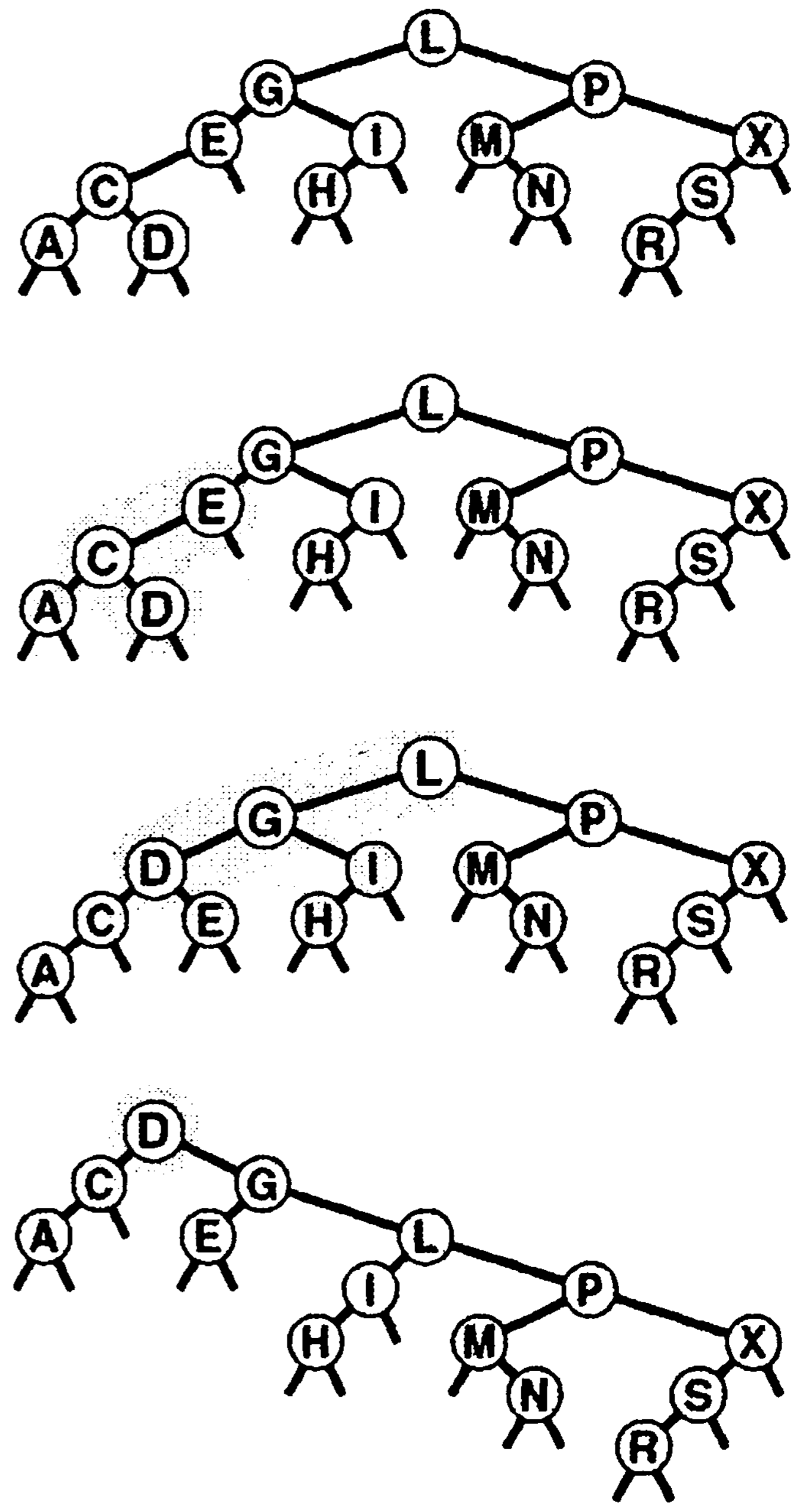
При вставке узла в BST-дерево с использованием вставки с расширением мы не только перемещаем этот узел в корень, но и перемещаем все встретившиеся узлы (в пути поиска) ближе к корню. Точнее говоря, выполняемые ротации уменьшают в два раза расстояние от корня до любого встретившегося узла. Это свойство сохраняется также при реализации операции *search* таким образом, чтобы во время поиска она выполняла трансформации с расширением. Некоторые пути в деревьях удлиняются: если обращение к узлам в этих путях не выполняется, указанный эффект не имеет значения. Если же мы обращаемся к узлам в длинном пути, после обращения он укорачивается в два раза; таким образом, ни один путь не может породить больших затрат.

**Лемма 13.5** *Количество сравнений, требуемых для любой последовательности  $M$  операций вставки или поиска в  $N$ -узловом расширенном BST-дереве, равно*

$$O((N + M) \lg(N + M)).$$

Доказательство этого утверждения, приведенное Слеатором (Sleator) и Тарьяном (Tarjan) в 1985 г. — классический пример амортизационного анализа алгоритмов (см. раздел ссылок). Подробно он исследуется в части 8.

Лемма 13.5 представляет гарантию амортизационного алгоритма: мы гарантируем не эффективность каждой операции, но эффективность *усредненных* затрат всех выполненных операций. Это усредненное значение не является вероятностным; скорее можно утверждать, что *общие* затраты гарантированно будут низкими. Для многих приложений такого рода гарантии достаточно, но для других приложений этого может оказаться мало. Например, нельзя гарантировать время ответа для каждой



**РИСУНОК 13.7 ВСТАВКА С РАСШИРЕНИЕМ**

На этом рисунке изображен результат (внизу) вставки записи с ключом  $D$  в пример дерева, приведенный на верхнем рисунке, с использованием вставки в корень с расширением. В этом случае процесс вставки состоит из двойной ротации влево-вправо, за которой следует двойная ротация вправо-вправо (от вершины).



операции при использовании расширенных BST-деревьев, поскольку время выполнения некоторых операций может определяться линейной зависимостью. Если какая-либо операция занимает время, которое определяется линейной зависимостью, существует гарантия, что другие операции будут выполняться гораздо быстрее, но это — слабое утешение для клиента, который вынужден ожидать.

Предельное значение, приведенное в лемме 13.5 — предельное значение общих затрат на все операции для худшего случая: как это типично для предельных значений худших случаев, эти затраты могут быть гораздо выше фактических. Операции с расширением перемещают недавно посещенные элементы ближе к вершине дерева; поэтому этот метод привлекателен для приложений поиска, имеющих неоднородные шаблоны обращения — особенно для приложений со сравнительно небольшим, пусть даже и медленно изменяющимся, рабочим набором элементов, к которым выполняется обращение.

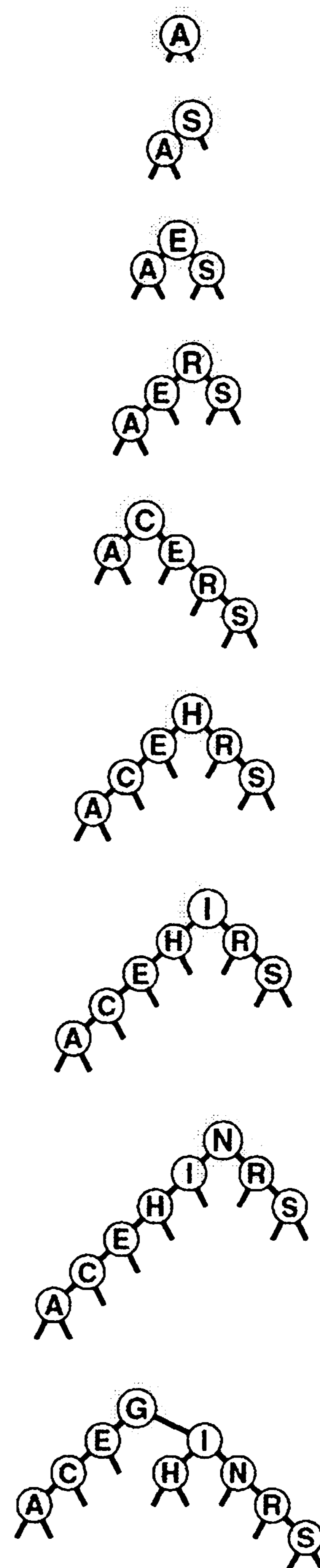
На рис. 13.9 приведены два примера, демонстрирующие эффективность операций расширения-ротации при балансировке дерева. На этих рисунках вырожденное дерево (построенное за счет вставки элементов в порядке их ключей) приводится в сравнительно хорошо сбалансированное состояние в результате небольшого числа операций *search*.

Обобщая, можно сказать, что небольшое количество выполненных поисков существенно улучшает сбалансированность дерева.

Если в дереве поддерживаются дублированные ключи, то операция с расширением может привести к тому, что элементы с ключами, равными ключу в данном узле, попадут по обе стороны от этого узла (см. упражнение 13.38). Из этого следует, что нельзя найти все элементы с данным ключом столь же легко, как это имело место в стандартных BST-деревьях. Необходимо проверить наличие дубликатов в обоих поддеревьях или воспользоваться каким-либо альтернативным методом для работы с дублированными ключами, как было описано в главе 12.

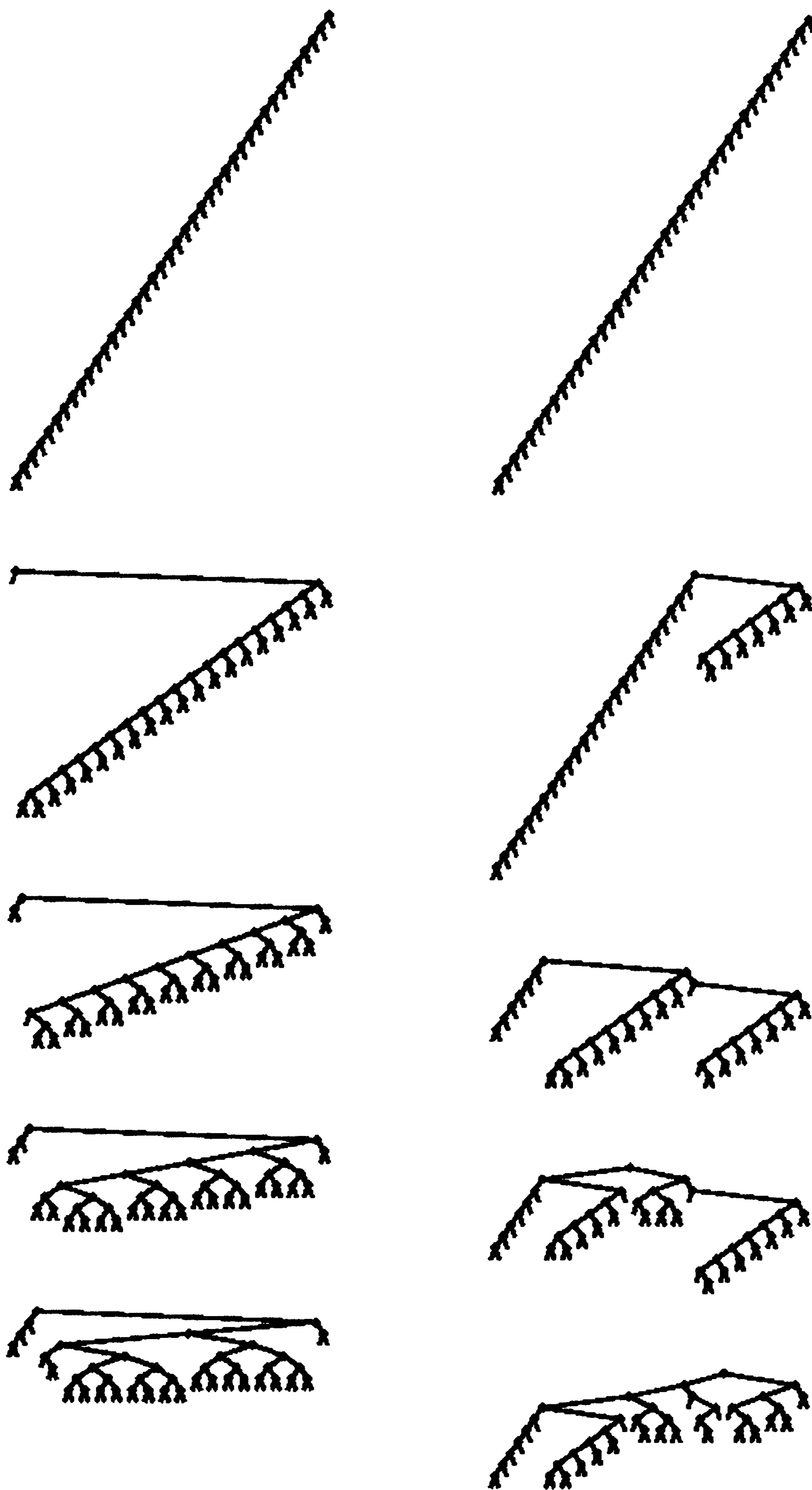
## Упражнения

- ▷ 13.25 Нарисуйте расширенное BST-дерево, образованное в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево с использованием вставки с расширением.



**РИСУНОК 13.8 ПОСТРОЕНИЕ РАСШИРЕННОГО ДЕРЕВА**

*Эта последовательность рисунков демонстрирует вставку записей с ключами **A S E R C H I N G** в первоначально пустое дерево с использованием вставки с расширением.*



### РИСУНОК 13.9 БАЛАНСИРОВКА ХУДШЕГО СЛУЧАЯ РАСШИРЕННОГО ДЕРЕВА С ПОМОЩЬЮ ПОИСКА

*Вставка ключей по порядку в первоначально пустое дерево с использованием вставки с расширением требует только постоянного количества шагов для выполнения одной вставки, но создает несбалансированное дерево, показанное на верхних рисунках. Последовательность рисунков слева отображает результат поиска (с расширением) наименьшего, второго, третьего и четвертого наименьших ключей в дереве. Каждый поиск уменьшает длину пути к искомому ключу (и к большинству других ключей в дереве) в два раза. Последовательность рисунков справа показана балансировка этого же худшего случая дерева с использованием последовательности случайных попаданий при поиске. Каждый поиск уменьшает вдвое количество узлов в своем пути, уменьшая длину путей поиска и множества других узлов в дереве.*

▷ **13.26** Сколько связей дерева должно быть изменено для выполнения двойной ротации? Сколько связей действительно изменяется при выполнении каждой из двойных ротаций в программе 13.5?

**13.27** Добавьте в программу 13.5 реализацию операции *search* с расширением.

○ **13.28** Реализуйте нерекурсивную версию функции вставки с расширением в программе 13.5.

**13.29** Используйте программу-драйвер, созданную в упражнении 12.30, для определения эффективности расширенных BST-деревьев в качестве самоорганизующихся структур, сравнив их со стандартными BST-деревьями для распределений запросов на поиск, определенных в упражнениях 12.31 и 12.32.

○ **13.30** Нарисуйте все структурно различные BST-деревья, которые могут быть образованы в результате вставки  $N$  ключей в первоначально пустое дерево с использованием вставки с расширением, при  $2 \leq N \leq 7$ .

● **13.31** Определите вероятность того, что каждое из деревьев в упражнении 13.30 образовано в результате вставки  $N$  случайных различных элементов в первоначально пустое BST-дерево.

○ **13.32** Определите эмпирически среднее значение и стандартное отклонение количества сравнений, используемых для обнаружения попаданий и промахов при поиске в BST-дереве, построенного в результате вставки  $N$  случайных ключей в первоначально пустое дерево с использованием вставки с расширением, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ . Не следует выполнять какие-либо поиски: просто постройте деревья и вычислите длину их путей. Являются ли расширенные BST-деревья сбалансированными в большей степени, чем произвольные BST-деревья, в меньшей степени или так же?

**13.33** Усовершенствуйте программу, созданную для упражнения 13.32, чтобы она выполняла  $N$  случайных поисков (скорее всего, они будут неудачными), выполняя расширение в каждом из созданных деревьев. Как расширение влияет на среднее количество сравнений, необходимых для выявления промаха при поиске?

**13.34** Измените программы, созданные для упражнений 13.32 и 13.33, чтобы они измеряли время выполнения, а не просто подсчитывали количество сравнений. Проведите аналогичные эксперименты. Поясните любые изменения в выводах, получаемых из экспериментальных результатов.

**13.35** Сравните расширенные BST-деревья со стандартными BST-деревьями применительно к задаче построения индекса из фрагмента реального текста, содержащего по меньшей мере 1 миллион символов. Измерьте время, требуемое для построения индекса и средние длины путей в BST-деревьях.

**13.36** Определите экспериментально среднее количество сравнений для обнаружения попаданий при поиске в расширенном BST-дереве, построенном в результате вставки произвольных ключей, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

**13.37** Прибегните к экспериментальному исследованию с целью проверки идеи использования вставки с расширением вместо стандартной вставки в корень для рандомизованных BST-деревьев.

▷ **13.38** Нарисуйте расширенное BST-дерево, образованное в результате вставки элементов с ключами 0 0 0 0 0 0 0 0 0 0 0 0 1 в указанном порядке в первоначально пустое дерево.

## 13.3 Нисходящие 2-3-4-деревья

Несмотря на гарантию производительности, которую можно обеспечить при использовании рандомизованных и расширенных BST-деревьев, в обоих случаях не исключается вероятность того, что время выполнения отдельной операции поиска может определяться линейной зависимостью. Следовательно, эти методы не помогают ответить на основной вопрос в отношении сбалансированных деревьев: существует ли тип BST-дерева, для которого можно гарантировать логарифмическую зависимость времени выполнения каждой операции *insert* и *search* от размеров дерева? В этом и следующем разделах мы рассмотрим абстрактное обобщение BST-деревьев и абстрактное представление этих деревьев в виде типа BST-дерева, которые позволяют утвердительно ответить на этот вопрос.

Для гарантии того, что создаваемые BST-деревья будут сбалансированными, используемые структуры деревьев должны обладать определенной гибкостью. Для получения подобной гибкости давайте предположим, что узлы в наших деревьях могут содержать более одного ключа. В частности, мы допускаем существование 3-узлов и 4-узлов, которые могут содержать, соответственно, два и три ключа. 3-узел имеет три исходящие из него связи: одну ко всем элементам, ключи которых меньше обоих его ключей, одну ко всем элементам, ключи которых располагаются между двумя его ключами, и одну ко всем элементам, ключи которых больше обоих его ключей. Аналогично, 4-узел имеет 4 исходящих связи: по одной для каждого из интервалов, определенных его тремя ключами. Таким образом, узлы в стандартном BST-дереве можно было бы называть 2-узлами: они содержат один ключ и две связи. Позже мы рассмотрим эффективные способы определения и реализации базовых операций с этими расширенными узлами; а пока давайте примем, что ими можно манипулировать обычным образом, и посмотрим, как их можно собрать воедино для образования деревьев.

**Определение 13.1** *2-3-4-дерево поиска* — это либо пустое дерево, либо дерево, содержащее три типа узлов: **2-узлы** с одним ключом, левой связью к дереву с меньшими ключами и правой связью к дереву с большими ключами; **3-узлы** с двумя ключами, с левой связью к дереву с меньшими ключами, средней связью к дереву, значения ключей которых лежат между значениями ключей данного узла, и правой связью к дереву с большими ключами; и **4-узлы** с тремя ключами и четырьмя связями к деревьям, значения ключей которых определены диапазонами, образованными ключами узла.

**Определение 13.2** *Сбалансированное 2-3-4-дерево поиска* — это 2-3-4-дерево поиска, все пустые деревья которого расположены на одинаковом расстоянии от корня.

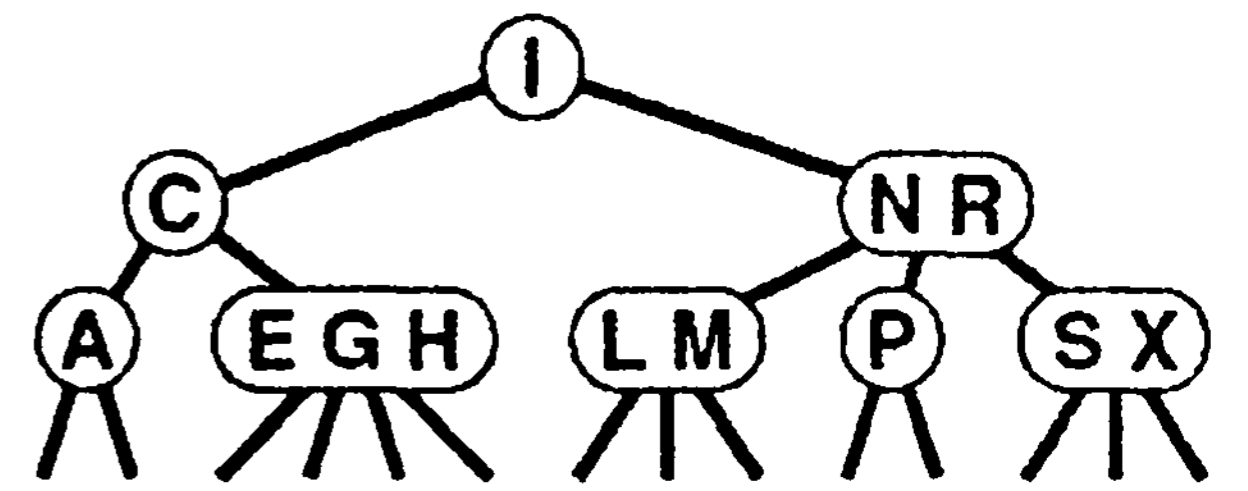


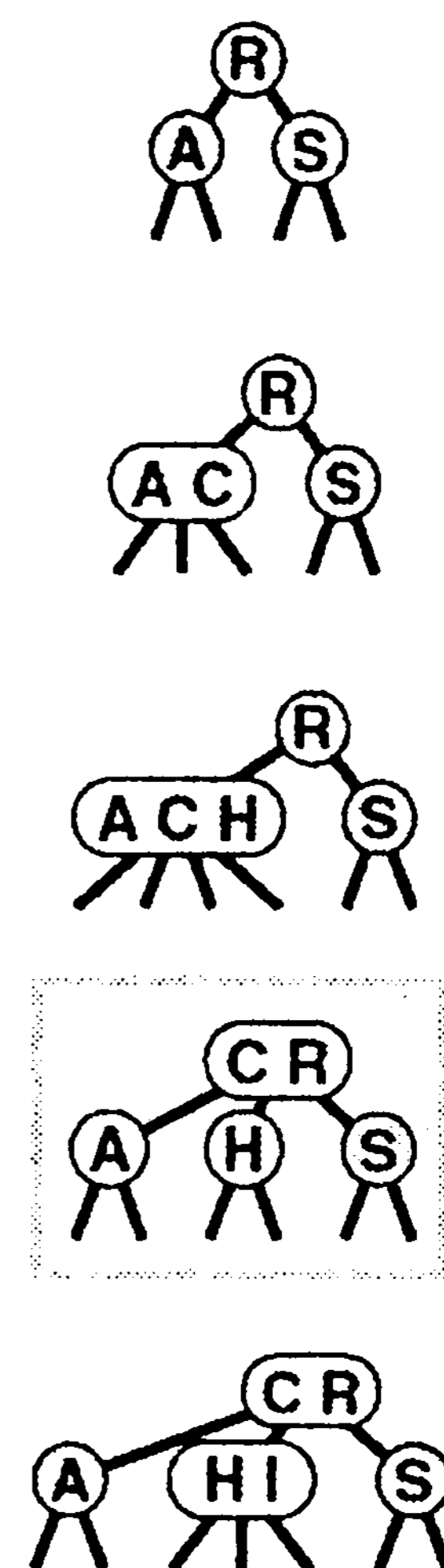
РИСУНОК 13.10 2-3-4-ДЕРЕВО

На этом рисунке изображено 2-3-4-дерево, содержащее ключи *A S R C H I N G E X M P L*. В таком дереве ключ можно отыскать, используя ключи в узле, расположенном в корне, для нахождения связи к поддереву, а затем продолжить рекурсивное выполнение поиска. Например, для выполнения поиска ключа *P* в этом дереве потребовалось бы проследить правую связь от корня, поскольку *P* больше *I*, затем — среднюю связь от правого дочернего дерева корня, поскольку значение *P* располагается между *N* и *R*, и, наконец, завершить успешный поиск во втором узле, содержащем ключ *P*.

В этой главе термин *2-3-4-дерево* будет применяться к сбалансированным 2-3-4-деревьям поиска (вообще говоря, в других контекстах он обозначает более общую структуру). Пример 2-3-4-дерева приведен на рис. 13.10. Алгоритм поиска ключей в таком дереве представляет собой обобщение алгоритма поиска для BST-деревьев. Чтобы выяснить, находится ли ключ в дереве, мы сравниваем его с ключами в корне: если он равен любому из них, имеет место попадание при поиске; в противном случае мы отслеживаем связь от корня к поддереву, соответствующему набору значений ключей, который должен содержать искомый ключ, и рекурсивно выполняем поиск в этом дереве. Существует ряд способов представления 2-, 3- и 4-узлов и для организации поиска соответствующей связи; отложим рассмотрение этих решений до раздела 13.4, где приводится особенно удобная организация поиска.

Для вставки нового узла в 2-3-4-дерево можно было бы выполнить безрезультатный поиск, а затем присоединить узел, как это делалось в BST-деревьях, но при этом новое дерево оказалось бы несбалансированным. Основная особенность, делающая 2-3-4-деревья столь важными, состоит в том, что можно выполнять вставки, всегда сохраняя полную сбалансированность дерева. Например, легко видеть, что делать, если поиск прерывается на 2-узле: достаточно преобразовать его в 3-узел. Аналогично, если поиск прерывается на 3-узле, его достаточно преобразовать в 4-узел. Но что делать, если поиск прерывается на 4-узле? Решение состоит в том, что можно освободить место для нового ключа, сохраняя сбалансированность дерева, вначале разделив 4-узел на два 2-узла, передвинув средний узел вверх к родительскому узлу данного узла. Эти три описанных случая проиллюстрированы на рис. 13.11.

А что делать, если необходимо разделить 4-узел, родительский узел которого — также 4-узел? Одним из возможных методов было разделение также и родительского узла, но узел-предок также мог бы оказаться 4-узлом и т.д. — возможно, пришлось бы разделять узлы вдоль всего дерева. Более простой подход заключается в обеспечении того, чтобы путь поиска не завершался в 4-узле за счет разделения любого 4-узла, попадающегося на пути *вниз* по дереву.



**РИСУНОК 13.11 ВСТАВКА В 2-3-4-ДЕРЕВО**

*2-3-4-дерево, состоящее только из 2-узлов, аналогично BST-дереву (верхний рисунок). Ключ С можно вставить, преобразовав 2-узел, в котором прерывается поиск С, в 3-узел (второй сверху рисунок). Аналогично, можно вставить ключ Н, преобразовав 3-узел, в котором прерывается его поиск, в 4-узел (третий сверху рисунок). Для вставки ключа I требуется выполнение дополнительных действий, поскольку его поиск прерывается в 4-узле. Вначале мы разделяем 4-узел, передавая его средний ключ родительскому узлу, и преобразуем этот узел в 3-узел (четвертый сверху, выделенный рисунок). Такое преобразование создает допустимое 2-3-4-дерево, в нижней части которого появляется место для I. И, наконец, мы вставляем I в 2-узел, на котором теперь прерывается поиск, и преобразуем этот узел в 3-узел (нижний рисунок).*

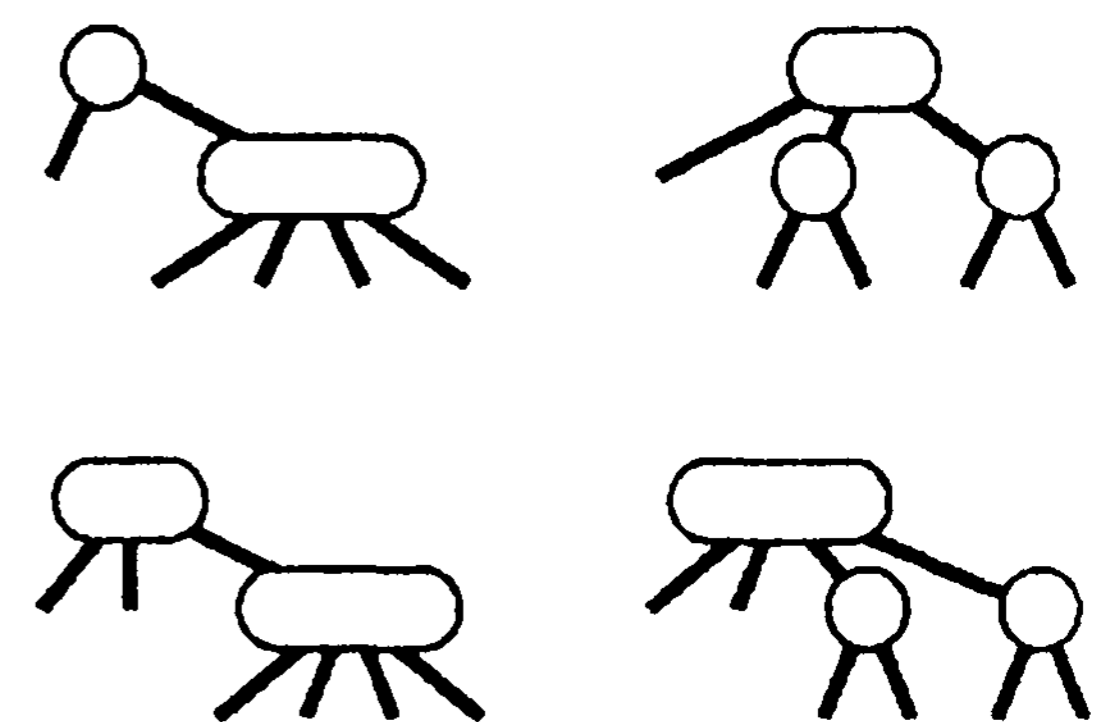
В частности, как показано на рис. 13.12, каждый раз, когда встречается 2-узел, соединенный с 4-узлом, такая пара преобразуется в 3-узел, соединенный с двумя 2-узлами; а когда встречается 3-узел, соединенный с 4-узлом, пара преобразуется в 4-узел, соединенный с двумя 2-узлами. Разделение 4-узлов возможно потому, что можно перемещать не только ключи, но и *связи*. Два 2-узла имеют столько же (четыре) связей, как и 4-узел, поэтому разделение можно выполнить, не внося никаких изменений ниже (или выше) разделяемого узла. 3-узел не преобразуется в 4-узел одним лишь добавлением еще одного ключа; требуется также еще один указатель (в этом случае — дополнительная связь, созданная разделением). Очень важно, что эти преобразования являются чисто локальными: никакую часть дерева, кроме показанной на рис. 13.12, не нужно исследовать или изменять. Каждое преобразование передает один из ключей 4-узла в его родительский узел и соответствующим образом преобразует связи.

Спускаясь вниз по дереву, не нужно беспокоиться явно о родительском узле текущего 4-узла, поскольку выполняемые преобразования обеспечивают, что при прохождении каждого узла в дереве мы попадаем в узел, который не является 4-узлом. В частности, при достижении нижней части дерева мы оказываемся не в 4-узле и можем вставить новый узел, непосредственно выполнив преобразование 2-узла в 3-узел либо 3-узла в 4-узел. Вставку можно считать разделением воображаемого 4-узла в нижней части, передающим вверх новый ключ.

Одно заключительное замечание: всегда, когда корень дерева становится 4-узлом, мы просто разделяем его, преобразуя в треугольник, состоящий из трех 2-узлов, как это делалось для первого разделяемого узла в предыдущем примере. Разделение корня после вставки несколько удобнее альтернативного подхода, когда приходится ожидать, пока новая вставка выполнит разделение, поскольку не нужно заботиться о родительском узле корня. Разделение корня (и только эта операция) приводит к увеличению высоты дерева на один уровень.

На рис. 13.13 представлено построение 2-3-4-дерева для тестового набора ключей. В отличие от стандартных BST-деревьев, которые разрастаются вниз от вершины, эти деревья разрастаются вверх от нижней части. Поскольку 4-узлы разделяются на пути от вершины вниз, деревья называются нисходящими 2-3-4-деревьями. Этот алгоритм важен, поскольку он создает практически идеально сбалансированные деревья поиска, хотя в процессе прохождения по дереву выполняются всего лишь несколько локальных преобразований.

**Лемма 13.6** При поиске в  $N$ -узловых 2-3-4-деревьях посещаются максимум  $\lg N + 1$  узлов.



**РИСУНОК 13.12 РАЗДЕЛЕНИЕ 4-УЗЛОВ В 2-3-4-ДЕРЕВЕ**

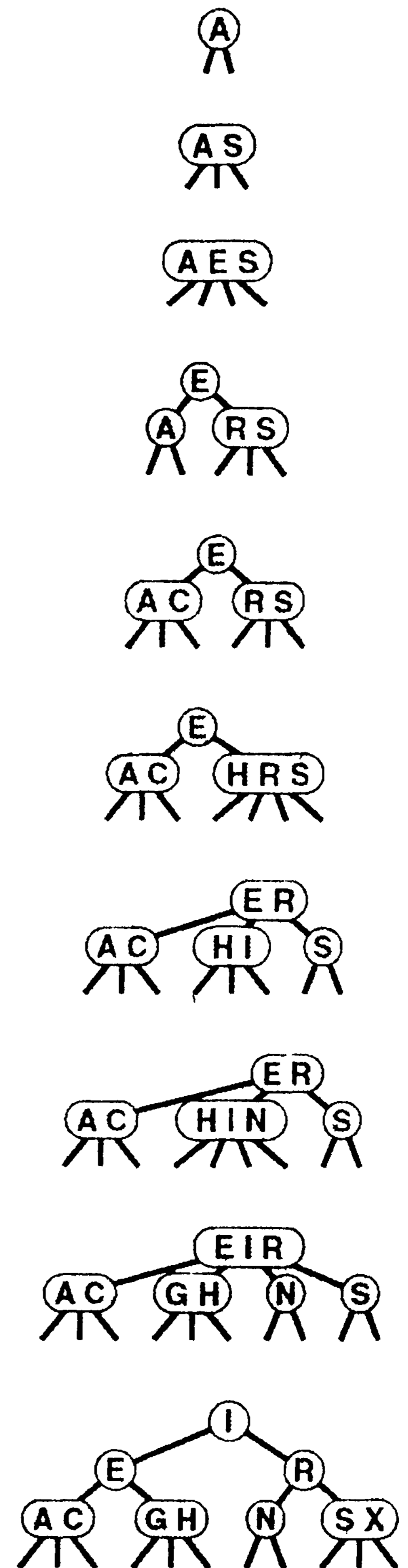
*В 2-3-4-дереве любой 4-узел, который не является дочерним узлом 4-узла, можно разделить на два 2-узла, передав его среднюю запись родительскому узлу. Присоединенный к 4-узлу 2-узел (верхний левый рисунок) становится 3-узлом, соединенным с двумя 2-узлами (вверху справа), а 3-узел, соединенный с 4-узлом (внизу слева), становится 4-узлом, соединенным с двумя 2-узлами (внизу справа).*

Каждый внешний узел располагается на одинаковом расстоянии от корня: выполняемые преобразования не оказывают никакого влияния на расстояние между любым узлом и корнем, за исключением случая, когда выполняется разделение корня (в этом случае расстояние между всеми узлами и корнем увеличивается на 1). Если все узлы являются 2-узлами, приведенное утверждение также справедливо, поскольку дерево подобно полному бинарному дереву; если в дереве присутствуют 3- и 4-узлы, высота может быть только меньше.

**Лемма 13.7.** Для вставок в  $N$ -узловых 2-3-4-деревьях в худшем случае требуется разделение менее  $\lg N + 1$  узлов, а в среднем, вероятно, потребуется менее одного разделения узла.

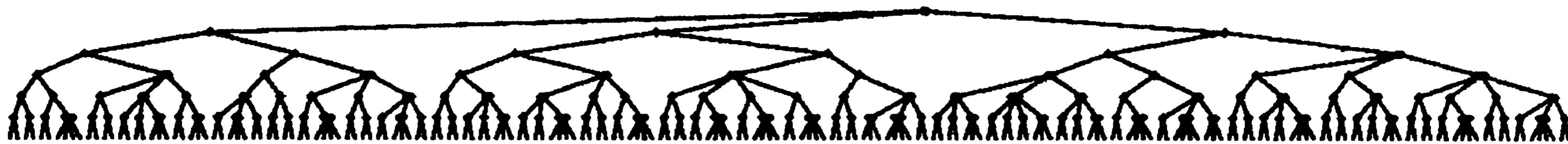
В самом худшем случае все узлы на пути к точке вставки являются 4-узлами и они все будут разделены. Но в дереве, образованном в результате случайных перестановок  $N$  элементов, маловероятен не только этот худший случай, но и в среднем, вероятно, потребуется меньше операций разделения, поскольку в деревьях 4-узлы встречаются не так часто. Например, в большом дереве на рис. 13.14 все 4-узлы кроме двух расположены на нижнем уровне. До сих пор специалистам не удавалось аналитически точно проанализировать производительность 2-3-4-деревьев, но из эмпирически полученных результатов становится очевидным, что для балансировки деревьев используется очень мало разделений. Худший случай определяется только соотношением  $\lg N$ , и в практических ситуациях не приходится говорить даже о приближении к этому значению.

Приведенное описание достаточно для определения алгоритма поиска с использованием 2-3-4-деревьев, который гарантирует достаточно высокую производительность для худшего случая. Однако, мы находимся лишь на половине пути к реализации. Хотя можно было бы создать алгоритмы, действительно выполняющие преобразования с различными типами данных, представляющими 2-, 3- и 4-узлы, для большинства встречающихся задач это непосредственное представление не очень удобно в плане реализации. Как и в случае расширенных BST-деревьев, накладные расходы, связанные с манипулированием более сложными структурами узлов, могли бы сделать алгоритмы более медленными, чем стандартный поиск с использованием BST-деревьев. Главное назначение балансировки — обеспечение средства против наихудшего случая, но мы предпочли бы, чтобы затра-



**РИСУНОК 13.13 ПОСТРОЕНИЕ 2-3-4-ДЕРЕВА**

На этой последовательности рисунков показан результат вставки элементов с ключами *A S E R C H I N G X* в первоначально пустое 2-3-4-дерево. Каждый встречающийся по пути поиска 4-узел разделяется, обеспечивая тем самым свободное место для нового элемента в нижней части дерева.



**РИСУНОК 13.14** Это 2-3-4-дерево — результат 200 случайных вставок в первоначально пустое дерево. Все пути поиска в дереве содержат не более шести узлов.

ты для этого были низкими, а также чтобы не приходилось идти на дополнительные затраты при каждом выполнении алгоритма. К счастью, как будет показано в разделе 13.4, существует простое представление 2-, 3- и 4-узлов, которое позволяет выполнять преобразования единообразно при небольших дополнительных затратах по сравнению со стандартным поиском в бинарном дереве.

Описанный алгоритм — всего лишь один из возможных способов поддержания баланса в 2-3-4-деревьях поиска. Разработано несколько других методов, которые позволяют достичь таких же результатов.

Например, можно выполнять балансировку снизу вверх. Вначале в дереве выполняется поиск с целью нахождения расположенного в нижней части дерева узла, к которому должен принадлежать вставляемый элемент. Если этот узел является 2-узлом или 3-узлом, он преобразуется в 3-узел или 4-узел, как описывалось ранее. Если он — 4-узел, он делится, как и ранее (со вставкой нового элемента в один из результирующих 2-узлов в нижней части), и средний элемент вставляется в родительский узел, если тот является 2- или 3-узлом. Если родительский узел является 4-узлом, он делится на два (со вставкой среднего узла в соответствующий 2-узел), а средний элемент вставляется в его родительский узел, если тот является 2- или 3-узлом. Если узел предок также является 4-узлом, мы продолжаем такой подъем по дереву, разделяя 4-узлы до тех пор, пока на пути поиска не встретится 2-узел или 3-узел.

Такой вид восходящей балансировки можно выполнять в деревьях, которые содержат только 2- или 3-узлы (и не имеют 4-узлов). Этот подход ведет к большему количеству операций разделения узлов во время выполнения алгоритма, но его проще программировать, поскольку приходится учитывать меньше случаев. В рамках еще одного подхода уменьшение количества операций разделения узлов достигается за счет отыскания родственных узлов, не являющихся 4-узлами, когда есть готовность приступить к разделению 4-узла.

Реализации всех этих методов требуют применения одной и той же рекурсивной схемы, как будет показано в разделе 13.14. В главе 16 рассматриваются также обобщения этих методов. Основное преимущество рассматриваемого нисходящего подхода по сравнению с другими методами заключается в том, что необходимая сбалансированность может быть достигнута в результате одного нисходящего прохода по дереву.

## Упражнения

- ▷ **13.39** Нарисуйте сбалансированное 2-3-4-дерево поиска, образованное в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево с использованием метода нисходящей вставки.



- ▷ **13.40** Нарисуйте сбалансированное 2-3-4-дерево поиска, образованное в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево с использованием метода восходящей вставки.
- **13.41** Какова минимальная и максимальная возможная высота сбалансированных 2-3-4-деревьев, содержащих  $N$  узлов?
- **13.42** Какова минимальная и максимальная возможная высота сбалансированных деревьев бинарного поиска, содержащих  $N$  узлов?
- **13.43** Нарисуйте все структурно различные сбалансированные 2-3-4-деревья бинарного поиска, содержащие  $N$  ключей, при  $2 \leq N \leq 12$ .
- **13.44** Определите вероятность того, что каждое из деревьев, нарисованных в упражнении 13.43, является результатом вставки  $N$  различных случайных элементов в первоначально пустое дерево.
- 13.45** Создайте таблицу, в которой будет отображаться количество деревьев из упражнения 13.43 для каждого значения  $N$ , которые являются изоморфными в том смысле, что они могут быть преобразованы одно в другое за счет обмена поддеревьев в узлах.
- ▷ **13.46** Опишите алгоритмы поиска и вставки в сбалансированные 2-3-4-5-6-деревья поиска.
- ▷ **13.47** Нарисуйте несбалансированное 2-3-4-дерево поиска, образованное в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево с использованием следующего метода. Если поиск завершается в 2- или 3-узле, его следует преобразовать в 3- или 4-узел, как в сбалансированном алгоритме; если поиск завершается в 4-узле, соответствующую связь в этом 4-узле следует заменить новым 2-узлом.

## 13.4 Красно-черные деревья, или RB-деревья

Описанный в предыдущем разделе алгоритм нисходящей вставки в 2-3-4-деревья прост для понимания, но его непосредственная реализация весьма громоздка в связи со множеством различных случаев, которые могут возникать. Приходится поддерживать три различных типа узлов, сравнивать ключи поиска с каждым из ключей в узлах, копировать связи и другую информацию из узлов одного типа в узлы другого типа, создавать и удалять узлы и т.д. В этом разделе мы исследуем простое абстрактное представление 2-3-4-деревьев, которое позволяет создавать естественную реализацию алгоритмов таблиц символов с почти оптимальными гарантиями производительности для худшего случая.

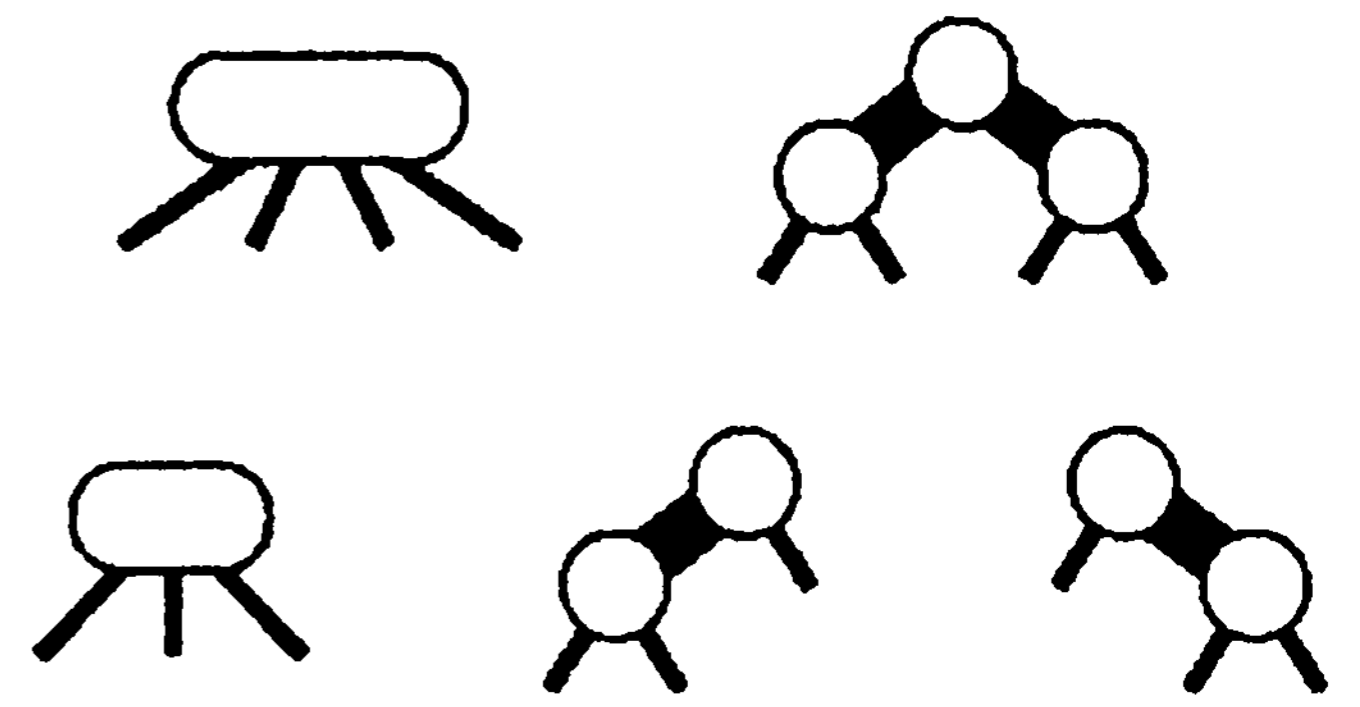
Основная идея заключается в представлении 2-3-4-деревьев в виде стандартных BST-деревьев (содержащих только 2-узлы), но с добавлением к каждому узлу дополнительного информационного разряда для кодирования 3-узлов и 4-узлов. Мы будем представлять связи двумя различными типам связей: *красными (red)* связями (R-связями), которые объединяют воедино небольшие бинарные деревья, образующие 3-узлы и 4-узлы, и *черными (black)* связями (B-связями), которые объединяют воедино 2-3-4-дерево. В частности, как показано на рис. 13.15, 4-узлы представляются тремя 2-узлами, соединенными R-связями, а 3-узлы представляются двумя 2-узлами, соеди-

ненными одной R-связью. R-связь в 3-узле может быть левой или правой, следовательно, существует два способа представления каждого 3-узла.

В любом дереве каждый узел указывается одной связью, поэтому *окрашивание связей эквивалентно окрашиванию узлов*. Соответственно, мы используем по одному дополнительному разряду на каждый узел для хранения цвета связи, указывающей на этот узел. 2-3-4-деревья, представленные таким образом, называются *красно-черными* (или RB-) деревьями бинарного поиска. Ориентация каждого 3-узла определяется динамикой алгоритма, которую мы опишем. Можно было бы выдвинуть правило, что все 3-узлы наклонены одинаково, но делать это не имеет смысла. Пример RB-дерева показан на рис. 13.16. Если исключить R-связи и свернуть соединяемые ими узлы, в результате будет получено 2-3-4-дерево, показанное на рис. 13.10.

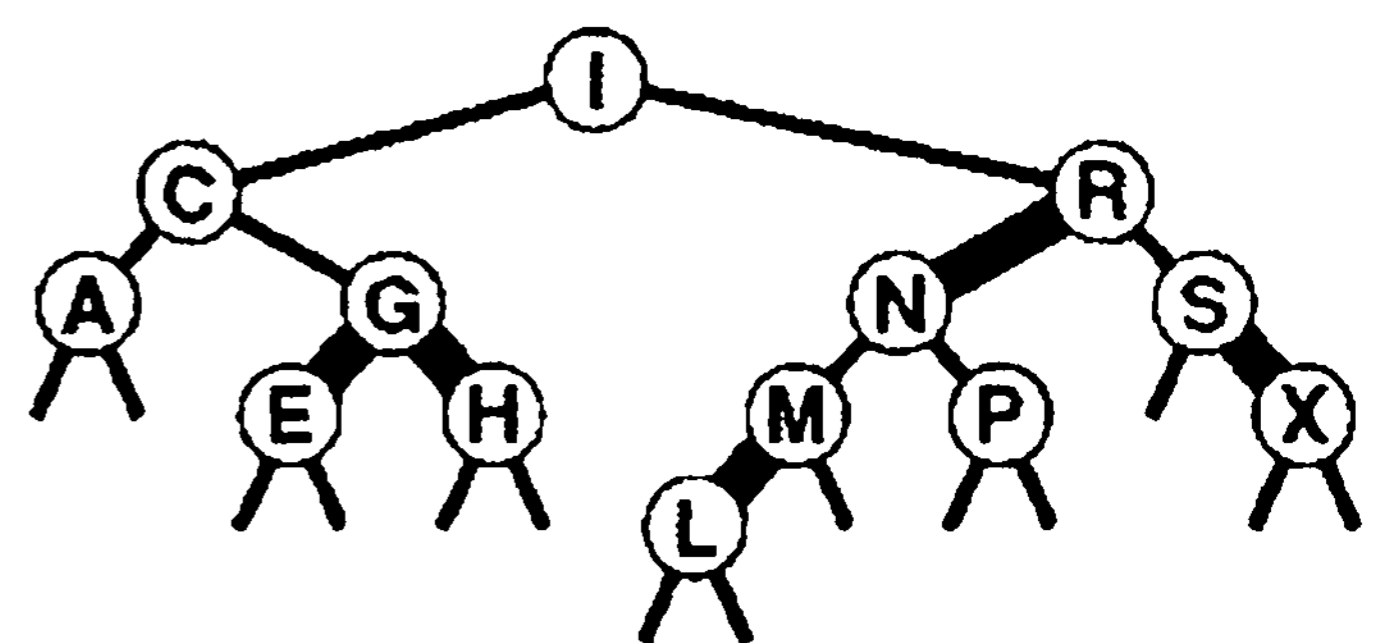
RB-деревья обладают двумя важными свойствами: (i) стандартный метод *search* для BST-деревьев работает безо всяких изменений; (ii) существует прямое соответствие между RB-деревьями и 2-3-4-деревьями, поэтому алгоритм с использованием сбалансированного 2-3-4-дерева можно реализовать, сохранив соответствие. Таким образом, можно воспользоваться лучшим из обоих подходов: простой метод поиска, присущий стандартному BST-дереву, и простой метод вставки-балансировки, характерный для 2-3-4-дерева поиска.

Метод поиска никогда не исследует поле, представляющее цвет узла, поэтому механизм балансировки не увеличивает время, занимаемое основной процедурой поиска. Поскольку каждый ключ вставляется только один раз, но в типичном приложении его поиск может выполняться многократно, в результате общее время поиска сокращается (поскольку деревья сбалансированы) ценой сравнительно небольших дополнительных затрат (во время поисков никаких действий по балансировке не присутствует). Более того, дополнительные затраты, связанные со вставкой, невелики: действия по балансировке приходится предпринимать только при встрече 4-узлов, а в дереве количество таких узлов невелико, поскольку они всегда разделяются. Внутренним циклом процедуры вставки является код,



**РИСУНОК 13.15 3-УЗЛЫ И 4-УЗЛЫ В RB-ДЕРЕВЬЯХ**

*Использование двух типов связей обеспечивает эффективный способ представления 3-узлов и 4-узлов в 2-3-4-деревьях. R-связи (жирные линии на схемах) используются для представления внутренних соединений в узлах, а B-связи (тонкие линии на схемах) — для представления связей 2-3-4-дерева. 4-узел (вверху слева) представляется сбалансированным поддеревом, состоящим из трех 2-узлов, которые соединены R-связями (вверху справа). Оба представления имеют три ключа и четыре B-связи. 3-узел (внизу слева) представляется одним 2-узлом, связанным с другим узлом (слева или справа) единственной R-связью (внизу справа). Оба представления имеют два ключа и три B-связи.*



**РИСУНОК 13.16 RB-ДЕРЕВО**

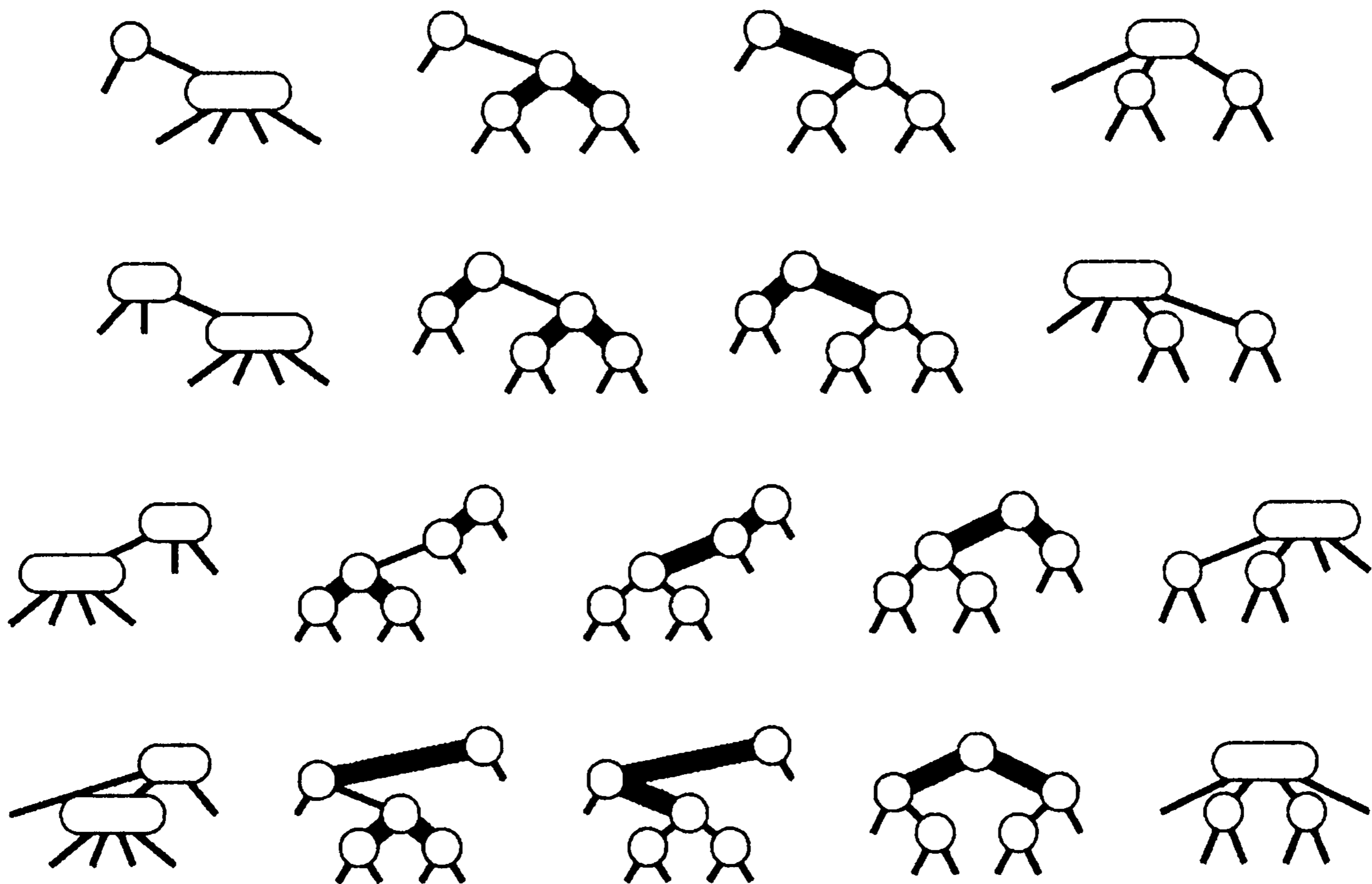
*На этом рисунке изображено RB-дерево, содержащее ключи A S R C H I N G E X M P L. Ключ в таком дереве можно найти при помощи стандартного поиска в BST-деревьях. В этом дереве любой путь от корня до внешнего узла содержит три B-связи. Если свернуть узлы, соединенные R-связями, мы получим 2-3-4-дерево, показанное на рис. 13.10.*

выполняющий прохождение вниз по дереву (аналогичный операциям вставки или поиска и вставки в стандартных BST-деревьях), с добавлением одной дополнительной проверки: если узел имеет два дочерних R-узла, он является частью 4-узла.

Столь небольшая перегрузка — основной фактор, определяющий эффективность RB-деревьев бинарного поиска.

Теперь давайте рассмотрим RB-представление двух преобразований, выполнение которых может потребоваться при встрече 4-узла. При наличии 2-узла, который соединен с 4-узлом, эту пару необходимо преобразовать в 3-узел, соединенный с двумя 2-узлами; при наличии 3-узла, соединенного с 4-узлом, пара должна быть преобразована в 4-узел, соединенный с двумя 2-узлами. Когда в нижнюю часть дерева добавляется новый узел, его можно представить в виде 4-узла, который должен быть разделен, причем его средний узел передается на один уровень вверх для вставки в нижний узел, где завершится поиск. Самой сущностью нисходящего процесса гарантируется, что этот узел должен быть либо 2-узлом, либо 3-узлом. Преобразование, требуемое при встрече 2-узла, соединенного с 4-узлом, выполняется без труда, и такое же преобразование работает применительно к 3-узлу, "правильно" соединенному с 4-узлом, как показано в двух первых примерах на рис. 13.17.

Остаются еще две ситуации, которые могут возникать при наличии 3-узла, соединенного с 4-узлом, как показано в последних двух примерах на рис. 13.17. (Фактически, существует четыре таких ситуации, поскольку 3-узлы другой ориентации могут также создавать зеркальные отражения представлений.) В этих случаях простое разделение 4-узла приводит к образованию двух последовательных R-связей — т.е. результирующее дерево не представляет 2-3-4-дерево в соответствии с принятыми



**РИСУНОК 13.17 РАЗДЕЛЕНИЕ 4-УЗЛОВ В RB-ДЕРЕВЕ**

*В RB-дереве операция разделения 4-узла, который не является дочерним 4-узла, реализуется изменением цветов узлов дерева, образующих 4-узел с последующим возможным выполнением одной или двух ротаций. Если родительский узел является 2-узлом (верхний рисунок) или 3-узлом с подходящей ориентацией (второй сверху рисунок), ротации не потребуются. Если 4-узел располагается на центральной связи 3-узла (нижний рисунок), требуется двойная ротация; в противном случае достаточно одиночной ротации (третий сверху рисунок).*

соглашениями. Эта ситуация не так уж плоха, поскольку имеется три узла, соединенные R-связями: достаточно преобразовать дерево так, чтобы R-связи указывали вниз из одного и того же узла.

К счастью, уже использованные операции ротации — это именно то, что необходимо для достижения требуемого эффекта. Давайте начнем с более простого из двух оставшихся случаев: третьего из представленных на рис. 13.17, когда 4-узел, соединенный с 3-узлом, разделяется, оставляя две идущие подряд и одинаково ориентированные R-связи. Эта ситуация не возникла бы, если бы 3-узел был ориентирован иначе. Соответственно, мы изменяем структуру дерева для изменения ориентации 3-узла и тем самым сводим этот случай ко второму, когда простого разделения 4-узла оказывается достаточно. Изменение структуры дерева для переориентации 3-узла заключается в выполнении единственной ротации с дополнительным требованием изменения цвета двух узлов.

И наконец, для обработки случая, когда 4-узел, соединенный с 3-узлом, разделяется, оставляя две идущие подряд R-связи, которые ориентированы по-разному, мы выполняем ротацию, чтобы свести этот случай к случаю с одинаково ориентированными связями, который затем обрабатывается, как было описано выше. Это преобразование сводится к выполнению тех же операций, что и при выполнении двойных ротаций влево-вправо и вправо-влево, которые использовались для расширенных BST-деревьев в разделе 13.2, хотя для правильной поддержки цветов потребуются незначительные дополнительные действия. Примеры операций вставки в RB-деревья приведены на рис. 13.18 и 13.19.

Программа 13.6 содержит реализацию операции *insert* для RB-деревьев, выполняющую преобразования, которые обобщены на рис. 13.17. Рекурсивная реализация позволяет изменять цвета 4-узлов на пути вниз по дереву (перед рекурсивными вызовами), а затем выполнять ротации на пути вверх по дереву (после рекурсивных вызовов). Эту программу было бы трудно понять без двух уровней абстракции, разработанных для ее реализации. Несложно убедиться, что рекурсивный подход реализует ротации, изображенные на рис. 13.17; затем можно убедиться, что программа реализует высокоуровневый алгоритм в 2-3-4-деревьях — разделяет 4-узлы на пути вниз по дереву, а затем вставляет новый элемент в 2- или 3-узел в месте завершения пути поиска в нижней части дерева.

### Программа 13.6 Вставка в RB-деревья бинарного поиска

Эта функция реализует вставку в 2-3-4-деревья за счет использования RB-представления. К типу `node` добавляется разряд цвета `red` (и соответствующим образом расширяется его конструктор), причем 1 означает, что узел является красным, а 0 — черным. На пути вниз

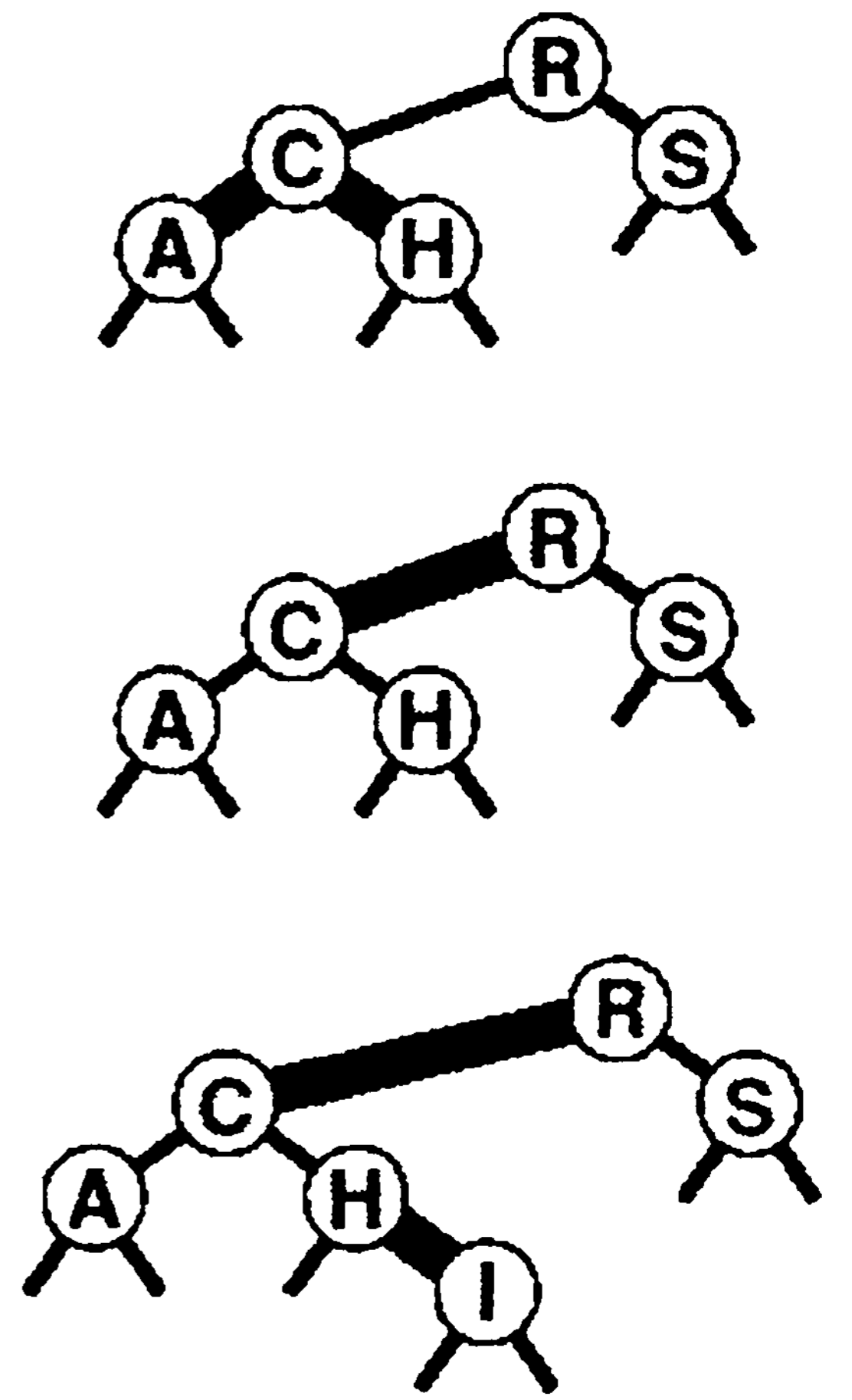


РИСУНОК 13.18 ВСТАВКА В RB-ДЕРЕВО

На этом рисунке показан результат (внизу) вставки записи с ключом *I* в пример RB-дерева, показанный вверху. В этом случае процесс вставки состоит из разделения 4-узла *C* с изменением цвета (в центре), последующим добавлением нового 2-узла в нижней части и преобразованием узла, содержащего ключ *H*, в 3-узел.

по дереву (перед рекурсивным вызовом) выполняется проверка на присутствие 4-узлов и их разделение с изменением разрядов цвета во всех трех узлах. По достижении нижней части дерева создается новый R-узел для вставляемого элемента и возвращается указатель на него. На пути вверх по дереву (после рекурсивного вызова) осуществляется проверка необходимости выполнения ротации. Если путь поиска содержит две R-связи с одинаковой ориентацией, выполняется единственная ротация от верхнего узла, затем разряды цвета изменяются с целью образования соответствующего 4-узла. Если путь поиска содержит две R-связи с различными ориентациями, выполняется единственная ротация от нижнего узла, в результате которой случай сводится к предыдущему.

private:

```
int red(link x)
{ if (x == 0) return 0; return x->red; }
void RBinsert(link& h, Item x, int sw)
{
  if (h == 0) { h = new node(x); return; }
  if (red(h->l) && red(h->r))
  { h->red = 1; h->l->red = 0; h->r->red = 0; }
  if (x.key() < h->item.key())
  {
    RBinsert(h->l, x, 0);
    if (red(h) && red(h->l) && sw) rotR(h);
    if (red(h->l) && red(h->l->l))
    { rotR(h); h->red = 0; h->r->red = 1; }
  }
  else
  {
    RBinsert(h->r, x, 1);
    if (red(h) && red(h->r) && !sw) rotL(h);
    if (red(h->r) && red(h->r->r))
    { rotL(h); h->red = 0; h->l->red = 1; }
  }
}
```

public:

```
void insert(Item x)
{ RBinsert(head, x, 0); head->red = 0; }
```

На рис. 13.20, который можно считать более подробной версией рис. 13.13, показано, как программа 13.6 строит RB-деревья, представляющие сбалансированные 2-3-4-деревья при вставке тестового набора ключей. На рис. 13.21 присутствует дерево, построенное на основе более объемного примера, чем тот, который был использован. Среднее количество узлов, посещенных во время поиска случайного ключа в этом древе, равно всего лишь 5.81. Можете сравнить это значение со значением 7.00 для дерева, построенного из этих же ключей в главе 12, и с 5.74 — наименьшим возможным для случая идеально сбалансированного дерева. Ценой всего нескольких ротаций мы получаем дерево, сбалансированное гораздо лучше любого другого, состоящего из этих же ключей. Программа 13.6 — эффектив-

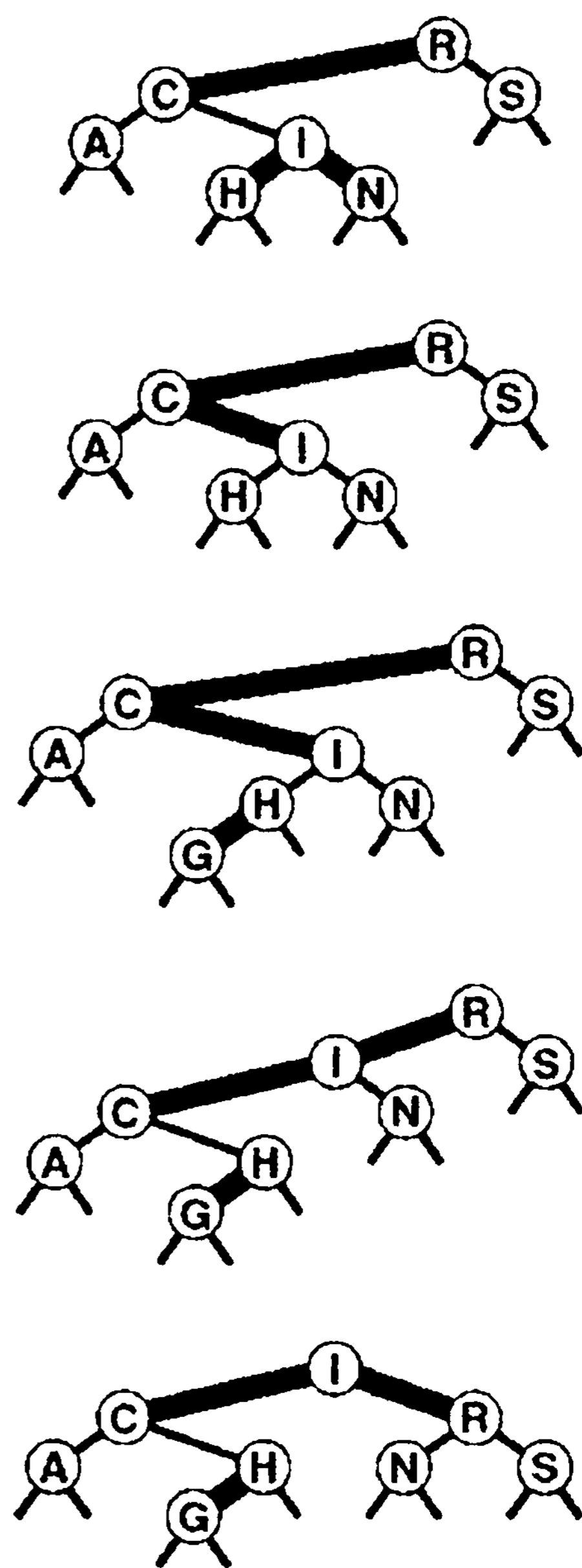


РИСУНОК 13.19 ВСТАВКА В RB-ДЕРЕВО С ИСПОЛЬЗОВАНИЕМ РОТАЦИЙ

На этом рисунке показан результат (внизу) вставки записи с ключом G в RB-дерево, приведенное вверху. В этом случае процесс вставки состоит из разделения 4-узла с ключом I с изменением цвета (второй сверху рисунок), затем добавления нового узла в нижней части дерева (третий сверху рисунок), и, наконец, выполнения (с возвратом к каждому узлу на пути поиска после вызовов рекурсивных функций) ротации влево в узле S и ротации вправо в узле R с целью завершения процесса разделения 4-узла.

ный, сравнительно компактный алгоритм вставки с использованием структуры бинарного дерева, который гарантирует логарифмическую зависимость для количества шагов для всех поисков и вставок. Это одна из немногих реализаций таблиц символов, обладающих подобным свойством, и ее использование оправдано в библиотечной реализации, когда свойства обрабатываемой последовательности ключей нельзя точно охарактеризовать.

**Лемма 13.8** Для поиска в *RB*-дереве с  $N$  узлами требуется менее  $2 \lg N + 2$  сравнений.

Только для разделений, которые в 2-3-4-дереве соответствуют 3-узлу, соединенному с 4-узлом, требуется ротация в *RB*-дереве; таким образом эта лемма — следствие леммы 13.2. Худший случай возникает тогда, когда путь к точке вставки состоит из чередующихся 3-узлов и 4-узлов.

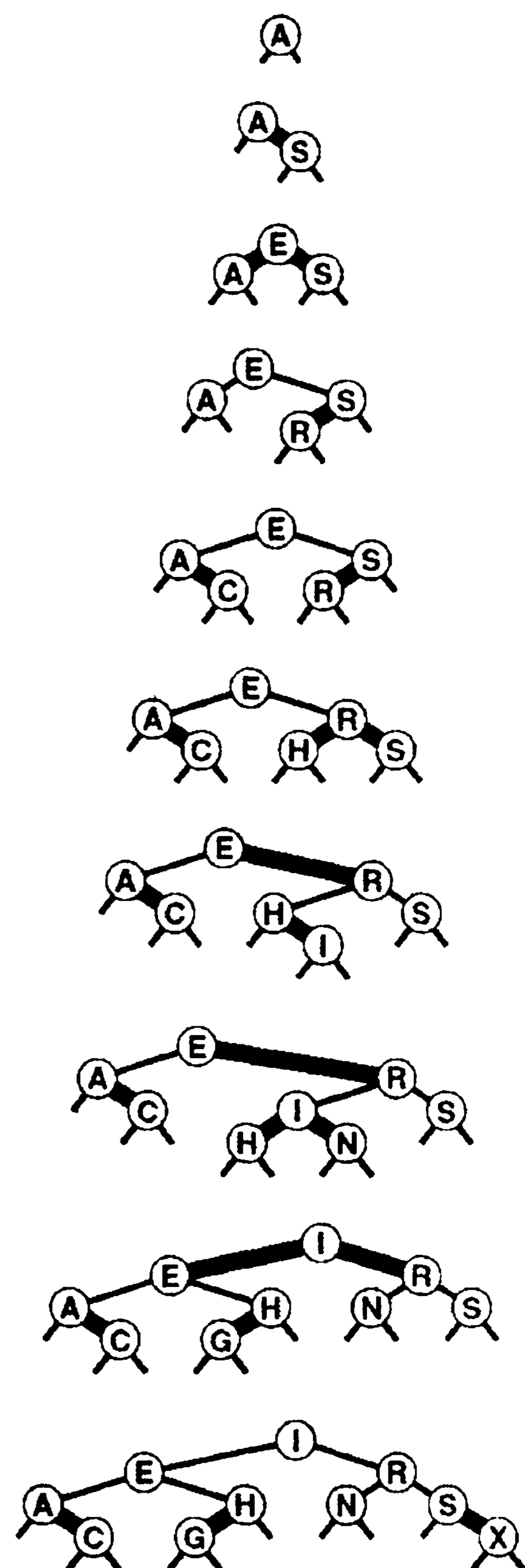
Более того, программа 13.6 устраняет небольшие накладные расходы, требуемые для балансировки, и создаваемые ею деревья почти оптимальны, поэтому она привлекательна также в качестве быстрого метода поиска общего назначения.

**Лемма 13.9** Для поиска в *RB*-дереве с  $N$  узлами, построенном из случайных ключей, в среднем используется около  $1.002 \lg N$  сравнений.

Константа 1.002, установленная путем частичного анализа и моделирования (см. раздел ссылок), достаточно мала, чтобы можно было считать *RB*-деревья оптимальными для практического применения, но вопрос о том, действительно ли *RB*-деревья являются асимптотически оптимальными, остается открытым. Можно ли считать константу равной 1?

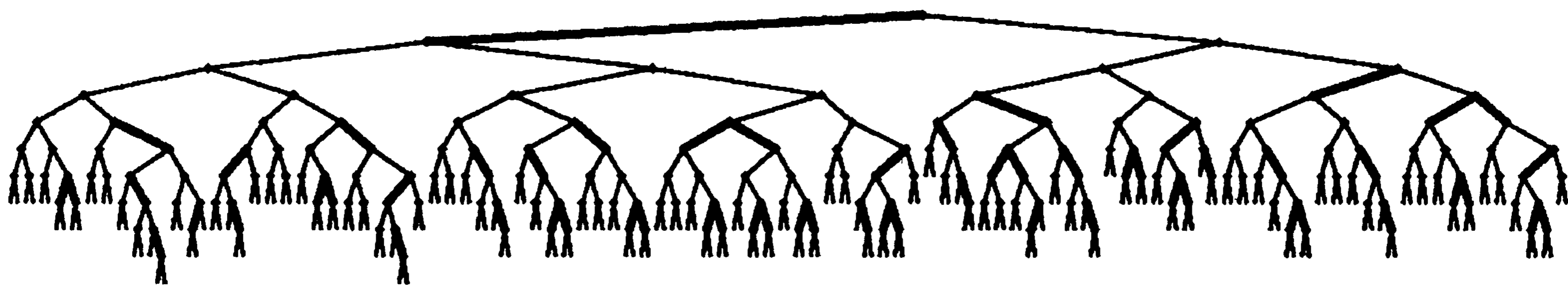
Поскольку в рекурсивной реализации из программы 13.6 определенные действия выполняются перед рекурсивными вызовами, а определенные — после рекурсивных вызовов, ряд модификаций дерева выполняется при перемещении по пути поиска вниз, а ряд — на обратном пути вверх. Следовательно, в ходе одного нисходящего прохождения балансировка не выполняется. Это обстоятельство мало влияет на большинство приложений, поскольку глубина рекурсии гарантировано мала. Для некоторых приложений, связанных с несколькими независимыми процессами, обращающимися к одному и тому же дереву, может потребоваться не-рекурсивная реализация, которая в каждый заданный момент времени активно оперирует только постоянным количеством узлов (см. упражнение 13.66).

Для приложения, которое хранит в дереве и другую информацию, операция ротации может оказаться дорогостоящей, возможно принуждая обновлять информацию во



**РИСУНОК 13.20 ПОСТРОЕНИЕ *RB*-ДЕРЕВА**

Эта последовательность рисунков демонстрирует результат вставки записей с ключами *A S E R C H I N X* в первоначально пустое *RB*-дерево.



**РИСУНОК 13.21 БОЛЬШОЕ RB-ДЕРЕВО БИНАРНОГО ПОИСКА**

*Это RB-дерево — результат вставки случайным образом упорядоченных ключей в первоначально пустое дерево. Для обнаружения всех промахов при поиске в этом дереве требуется от 6 до 12 сравнений.*

всех узлах поддеревьев, затрагиваемых ротацией. Для таких приложений можно обеспечить, чтобы каждая вставка требовала не более одной ротации, используя RB-дерева для реализации восходящих 2-3-4-деревьев поиска, которые описаны в конце раздела 13.3. Вставка в эти деревья сопряжена с разделением 4-узлов вдоль пути поиска, которое требует изменений цвета, но не выполнения ротаций в RB-представлении, за которыми следует одна одиночная или двойная ротация (один из случаев, показанных на рис. 13.17), когда первый 2-узел или 3-узел встречается вверх вдоль пути поиска (см. упражнение 13.59).

Если в дереве необходимо поддерживать дублированные ключи, то, подобно тому как это делалось в расширенных BST-деревьях, следует разрешить элементам с ключами, равными ключу данного узла, размещаться по обе стороны от этого узла. В противном случае длинные строки дублированных ключей могут привести к серьезному дисбалансу. И снова, это наблюдение свидетельствует, что для отыскания всех элементов с данным ключом требуется специализированный код.

Как упоминалось в конце раздела 13.3, RB-представления 2-3-4-деревьев входят в число нескольких аналогичных стратегий, которые были предложены для реализации сбалансированных бинарных деревьев (см. раздел ссылок). Как было показано, балансировка деревьев достигается операциями ротации: мы рассмотрели специфическое представление деревьев, которое упрощает принятие решения о необходимости выполнения ротации. Другие представления деревьев ведут к другим алгоритмам, часть из которых мы кратко рассмотрим в данном разделе.

Старейшая и наиболее изученная структура данных для сбалансированных деревьев — *сбалансированное по высоте, или AVL-, дерево*, исследованное Адельсоном-Вельским (Adel'son-Vel'skii) и Ландисом (Landis). Для этих деревьев характерно, что высоты двух поддеревьев каждого узла различаются максимум на 1. Если вставка приводит к тому, что высота одного из поддеревьев какого-либо узла увеличивается на 1, условие баланса может быть нарушено. Однако, одна одиночная или двойная ротация в любом случае вернет узел в сбалансированное состояние. Основанный на этом наблюдении алгоритм аналогичен методу восходящей балансировки 2-3-4-деревьев: для узла выполняется рекурсивный поиск, затем, *после* рекурсивного вызова, выполняется проверка разбалансировки и, при необходимости, одиночная или двойная ротация с целью корректирования баланса (см. упражнение 13.61). Для принятия решения о том, какие ротации нужно выполнять (если это необходимо), требуется знать, является ли высота каждого узла на 1 меньше, равна или на 1 больше высоты его родственного узла. Для прямого кодирования этой информации требуется по два

разряда на каждый узел, хотя, используя RB-абстракцию, можно обойтись и без использования дополнительного объема памяти (см. упражнения 13.62 и 13.65).

Поскольку 4-узлы не играют никакой специальной роли в алгоритме с использованием 2-3-4-деревьев, сбалансированные деревья можно строить по существу так же, используя только 2-узлы и 3-узлы. Построенные таким образом деревья называют *2-3-деревьями*. Такие деревья были исследованы Хопкрофтом (Hopcroft) в 1970 г. 2-3-деревья не обладают достаточной гибкостью, чтобы можно было построить удобный алгоритм нисходящей вставки. Кроме того, RB-структура может упростить реализацию, но восходящие 2-3-деревья не обеспечивают особых преимуществ по сравнению с восходящими 2-3-4-деревьями, поскольку для поддержания баланса по-прежнему требуются одиночные и двойные ротации. Восходящие 2-3-4-деревья несколько лучше сбалансированы и обладают тем преимуществом, что для каждой вставки требуется максимум одна ротация.

В главе 16 исследуется еще один важный тип сбалансированных деревьев — расширение 2-3-4-деревьев, называемое *B-деревьями*. B-деревья допускают существование до  $M$  ключей в одном узле для больших значений  $M$  и широко используются в приложениях поиска, работающих с очень большими файлами.

Мы уже определили RB-деревья их соответствием 2-3-4-деревьям. Интересно также сформулировать непосредственные структурные определения.

**Определение 13.3** *RB-дерево бинарного поиска* — это дерево бинарного поиска, в котором каждый узел помечен как **красный (R)** либо **черный (B)**, с наложением дополнительного ограничения, что никакие два красных узла не могут появляться друг за другом в любом пути от внешней связи к корню.

**Определение 13.4** *Сбалансированное RB-дерево бинарного поиска* — это RB-дерево бинарного поиска, в котором все пути от внешних связей к корню содержат одинаковое количество черных узлов.

А теперь давайте рассмотрим альтернативный подход к разработке алгоритма с использованием сбалансированного дерева, заключающийся в полном игнорировании абстракции 2-3-4-дерева и формулировании алгоритма вставки, который сохраняет определяющее свойство сбалансированных RB-деревьев бинарного поиска за счет применения ротаций. Например, использование восходящего алгоритма соответствует присоединению нового узла в нижней части пути поиска с помощью R-связи, затем продвижению вверх по пути поиска с выполнением ротаций или изменений цвета, как это делалось в случаях, представленных на рис. 13.17, с целью разбиения любой встретившейся пары последовательных R-связей. Основные операции не отличаются от выполняемых в программе 13.6 и в ее восходящем аналоге, но при этом имеются незначительные различия, поскольку 3-узлы могут быть ориентированы в любом направлении, операции могут выполняться в ином порядке и различные решения в отношении различных ротаций могут приниматься с равным успехом.

Давайте подведем итоги: используя RB-деревья для реализации сбалансированных 2-3-4-деревьев, можно разработать таблицу символов, в которой операция *search* для ключа в файле, состоящем, скажем, из 1 миллиона элементов, может быть выполнена путем сравнения этого ключа приблизительно с 20 другими ключами. В худшем случае требуется не более 40 сравнений. Более того, с каждым сравнением связаны лишь



небольшие накладные расходы и поэтому гарантируется быстрое выполнение операции *search* даже в очень больших файлах.

## Упражнения

- ▷ **13.48** Нарисуйте RB-дерево бинарного поиска, образованное в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево с использованием метода нисходящей вставки.
- ▷ **13.49** Нарисуйте RB-дерево бинарного поиска, образованное в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево с использованием метода восходящей вставки.
- **13.50** Нарисуйте RB-дерево, образованное в результате вставки по порядку букв от **A** до **K** в первоначально пустое дерево, а затем опишите, что происходит в общем случае построения дерева путем вставки ключей в порядке возрастания.
- 13.51** Приведите последовательность вставок, в результате которой будет создано RB-дерево на рис. 13.16.
- 13.52** Создайте два произвольных 32-узловых RB-деревя. Нарисуйте их (вручную или с помощью программы). Сравните их с BST-деревьями (несбалансированными), построенными из этих же ключей.
- 13.53** Сколько различных RB-деревьев соответствуют 2-3-4-дереву, содержащему  $t$  3-узлов?
- **13.54** Нарисуйте все структурно различные RB-деревья поиска, содержащие  $N$  ключей, при  $2 \leq N \leq 12$ .
- **13.55** Определите вероятность того, что каждое из деревьев в упражнении 13.43 является результатом вставки  $N$  случайных различных элементов в первоначально пустое дерево.
- 13.56** Создайте таблицу, в которой приведено количество деревьев для каждого значения  $N$  из упражнения 13.54, являющихся изоморфными в том смысле, что они могут быть преобразованы одно в другое путем взаимной замены поддеревьев в узлах.
- **13.57** Покажите, что в худшем случае длина почти всех путей от корня к внешнему узлу в RB-дереве, состоящем из  $N$  узлов, равна  $2 \lg N$ .
- 13.58** Сколько ротаций требуется в худшем случае для вставки в RB-дерево, состоящее из  $N$  узлов?
- **13.59** Используя RB-представление и рекурсивный подход, аналогичный программе 13.6, реализуйте операции *construct*, *search* и *insert* для таблиц символов, в основе которых лежат структуры данных в виде восходящих сбалансированных 2-3-4-деревьев. *Совет:* код может выглядеть аналогично программе 13.6, но должен выполнять операции в другом порядке.
- 13.60** Используя RB-представление и рекурсивный подход, аналогичный программе 13.6, реализуйте операции *construct*, *search* и *insert* для таблиц символов, в основе которых лежат структуры данных в виде восходящих сбалансированных 2-3-деревьев.

- 13.61** Используя рекурсивный подход, аналогичный программе 13.6, реализуйте операции *construct*, *search* и *insert* для таблиц символов, в основе которых лежат структуры данных в виде сбалансированных по высоте (AVL) деревьев.
- **13.62** Измените реализацию, созданную в упражнении 13.61, чтобы использовать RB-деревья (содержащие по 1 разряду на узел) для кодирования информации о балансировке.
  - **13.63** Реализуйте сбалансированные 2-3-4-деревья, используя представление RB-дерева, в котором 3-узлы всегда наклонены вправо. *Примечание:* это изменение позволяет исключить из внутреннего цикла операции *insert* одну из проверок разрядов.
  - **13.64** Для сохранения сбалансированности 4-узлов программа 13.6 выполняет ротации. Разработайте использующую представление в виде RB-дерева реализацию сбалансированных 2-3-4-деревьев, в которой 4-узлы могут быть представлены любыми тремя узлами, соединенными двумя R-связями (полностью сбалансированными или несбалансированными).
  - **13.65** Не прибегая к использованию дополнительного объема памяти для хранения разряда цвета, реализуйте операции *construct*, *search* и *insert* для RB-деревьев, воспользовавшись следующим приемом. Чтобы окрасить узел в красный цвет, поменяйте местами две его связи. Затем, чтобы проверить, является ли узел красным, проверьте, больше ли его левый дочерний узел, чем правый. Дабы обеспечить возможный обмен указателями, придется модифицировать функции сравнения; в результате использования такого приема сравнения разрядов заменяются сравнениями ключей, что, по всей вероятности, требует больших затрат, однако метод демонстрирует, что в случае необходимости можно избавиться от дополнительного разряда в узлах.
  - **13.66** Реализуйте нерекурсивную функцию *вставки* в RB-дерево бинарного поиска (см. программу 13.6), которая соответствует вставке в сбалансированное 2-3-4-дерево в течение одного прохода. *Совет:* введите связи **gg**, **g** и **p**, которые указывают, соответственно, на прадеда, деда и родителя текущего узла в дереве. Все эти связи могут потребоваться для выполнения двойной ротации.
- 13.67** Создайте программу, которая вычисляет процентную долю B-узлов в заданном RB-дереве бинарного поиска. Протестируйте программу, вставив  $N$  случайных ключей в первоначально пустое дерево, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 13.68** Создайте программу, которая вычисляет процент элементов, находящихся в 3-узлах и 4-узлах заданного 2-3-4-дерева поиска. Протестируйте программу, вставив  $N$  случайных ключей в первоначально пустое дерево, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- ▷ **13.69** Используя по одному разряду на узел для представления цвета, можно представлять 2-, 3- и 4-узлы. Сколько разрядов на узел потребовалось бы для представления бинарным деревом 5-, 6- и 7-узлов?
- 13.70** Эмпирически вычислите среднее значение и стандартное отклонение количества сравнений, используемых для выявления попаданий и промахов при поиске в RB-дереве, построенном в результате вставки  $N$  случайных узлов в первоначально пустое дерево, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

**13.71** Модифицируйте программу, созданную в упражнении 13.70, для вычисления количества ротаций и разделений узлов, используемых для построения деревьев. Проанализируйте полученные результаты.

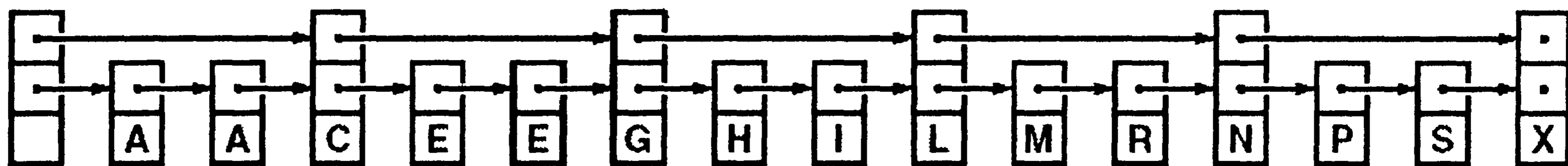
**13.72** Воспользуйтесь программой-драйвером из упражнения 12.30 для сравнения свойства самоорганизации расширенных BST-деревьев с гарантиями, обеспечиваемыми RB-деревьями бинарного поиска для худшего случая и со стандартными BST-деревьями для распределений запросов на поиск, определенных в упражнениях 12.31 и 12.32 (см. упражнение 13.29).

- **13.73** Реализуйте функцию *search* для RB-деревьев, которая выполняет ротации и изменяет цвета узлов в процессе спуска по дереву с целью обеспечения того, что узел в нижней части пути поиска не является 2-узлом.
- **13.74** Воспользуйтесь решением упражнения 13.73 для реализации функции *remove* для RB-деревьев. Найдите узел, который должен быть удален, продолжите поиск вплоть до нахождения 3-узла или 4-узла в нижней части пути и переместите узел-наследник из нижней части, чтобы заменить удаленный узел.

## 13.5 Списки пропусков

В этом разделе мы рассмотрим подход к разработке быстрых реализаций операций с таблицами символов, которые на первый взгляд кажутся совершенно отличными от рассмотренных методов, основанных на использовании деревьев, но в действительности очень тесно связанных с ними. Подход основан на рандомизованной структуре данных и почти наверняка обеспечивает практически оптимальную производительность всех базовых операций для АДТ таблиц символов. Лежащая в основе алгоритма структура данных, которая была разработана Пухом (Pugh) в 1990 г. (см. раздел ссылок), называется *списком пропусков*. В ней дополнительные связи в узлах связанного списка используются для пропуска больших частей списка во время поиска.

На рис. 13.22 приведен простой пример, в котором третий узел в упорядоченном связанном списке содержит дополнительную связь, которая позволяет пропустить три узла списка. Дополнительные связи можно использовать для ускорения операции *search*: просмотр верхней части списка выполняется до тех пор, пока не будет найден ключ или узел с меньшим ключом, содержащий связь с узлом с большим ключом; затем связи в нижней части используются для проверки двух промежуточных узлов. Этот метод ускоряет выполнение операции *search* в три раза, поскольку в ходе успешного поиска  $k$ -го узла в списке исследуются лишь около  $k/3$  узлов.



**РИСУНОК 13.22** ДВУХУРОВНЕВЫЙ СВЯЗНЫЙ СПИСОК

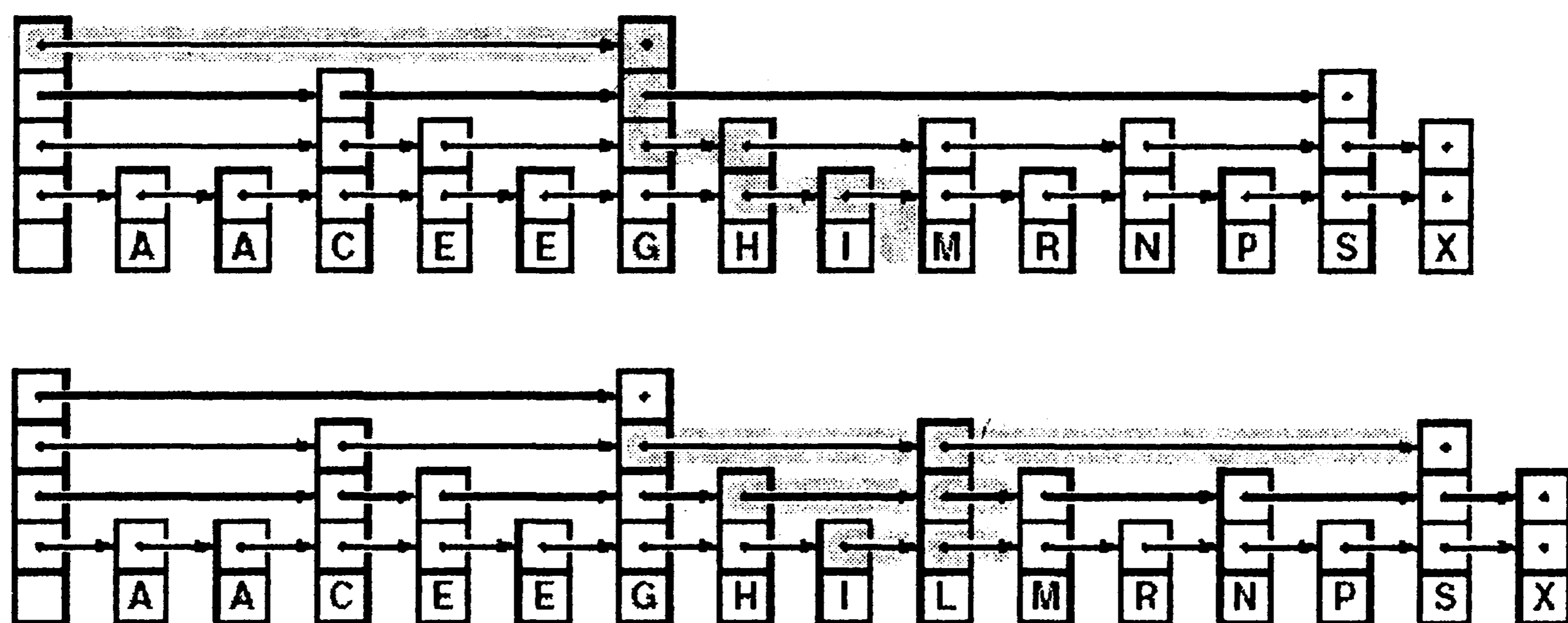
Каждый третий узел в этом списке содержит вторую связь, поэтому можно выполнять пропуски в списке и почти в три раза ускорить прохождение по списку по сравнению с использованием только первых связей. Например, до двенадцатого узла в списке (*P*), можно добраться из начала списка, следуя лишь по пяти связям: вторым связям к *C*, *G*, *L*, *N*, а затем по первой связи узла *N* к *P*.

К этой конструкции можно применить итерацию и обеспечить вторую дополнительную связь, позволяющую быстрее просматривать узлы с дополнительными связями, и т.д. Кроме того, конструкцию можно сделать более общей, пропуская с помощью каждой связи различное количество узлов.

**Определение 13.5** *Список пропусков (skip list) — это связный список, в котором каждый узел содержит различное количество связей, причем  $i$ -тые связи в узлах реализуют односвязные списки, пропускающие узлы, содержащие менее чем  $i$  связей.*

На рис. 13.23 показан этот же список пропусков и приведен пример поиска и вставки нового узла. Для выполнения поиска можно выполнить просмотр в верхнем списке, пока не будет найден ключ поиска или узел с меньшим ключом, содержащий связь с узлом с большим ключом. Затем мы переходим ко второму сверху списку и повторяем процедуру, продолжая этот процесс, пока не будет найден ключ поиска или пока в нижнем уровне не будет обнаружен промах при поиске. Для вставки мы выполняем поиск, связываясь с новым узлом при переходе с уровня  $k$  на уровень  $k - 1$ , если новый узел содержит, по меньшей мере,  $k$  дополнительных связей.

Внутреннее представление узлов предельно простое. Единственная связь в односвязном списке заменяется массивом связей и целочисленной переменной, содержащей количество связей в узле. Управление памятью — вероятно, наиболее сложный аспект использования списков пропусков. Объявления типов и код для выделения памяти под новые узлы будут вскоре исследованы при рассмотрении вставки. Пока же достаточно отметить, что доступ к узлу, который следует за узлом  $t$  на  $(k + 1)$ -м уровне списка пропусков, можно получить так:  $t \rightarrow \text{next}[k]$ . Рекурсивная реализация в программе 13.7 демонстрирует, что поиск в списках пропусков не только является весь-



**РИСУНОК 13.23** ПОИСК И ВСТАВКА В СПИСКЕ ПРОПУСКОВ

Добавляя дополнительные уровни к структуре, показанной на рис. 13.22, и позволяя связям пропускать различное количество узлов, мы получаем пример обобщенного списка пропусков. Для поиска ключа в этом списке процесс начинается с самого верхнего уровня с переходом вниз при каждой встрече ключа, который не меньше ключа поиска. В данном случае (см. верхний рисунок) мы находим ключ  $L$ , начав с уровня 3, следуя вдоль первой связи, затем спускаясь к  $G$  (считая нулевую связь связью со служебным узлом), затем следуя к  $I$ , спускаясь к уровню 2, поскольку  $S$  больше чем  $L$ , затем спускаясь к уровню 1, поскольку  $M$  больше  $L$ . Для вставки узла  $L$  с тремя связями мы связываем его с тремя списками именно там, где во время поиска были найдены связи с большими ключами.

ма понятным обобщением поиска в односвязных списках, но и подобен бинарному поиску или поиску в BST-деревьях. Вначале проверяется, содержит ли текущий узел ключ поиска. Затем, если это не так, ключ в текущем узле сравнивается с ключом поиска. Если он больше, выполняется один рекурсивный вызов, а если меньше — другой.

### Программа 13.7 Поиск в списках пропусков

Для  $k$  равного 0 этот код эквивалентен программе 12.6, выполняющей поиск в односвязных списках. Для общего случая  $k$  мы переходим к следующему узлу на уровне  $k$  списка, если его ключ меньше ключа поиска, и вниз к уровню  $k-1$ , если его ключ меньше.

```
private:
    Item searchR(link t, Key v, int k)
    { if (t == 0) return nullItem;
      if (v == t->item.key()) return t->item;
      link x = t->next[k];
      if ((x == 0) || (v < x->item.key()))
      {
          if (k == 0) return nullItem;
          return searchR(t, v, k-1);
      }
      return searchR(x, v, k);
    }
public:
    Item search(Key v)
    { return searchR(head, v, lgN); }
```

Первой задачей, с которой мы сталкиваемся при необходимости вставки нового узла в список пропусков, является определение количества связей, которые должен содержать узел. Все узлы содержат, по меньшей мере, одну связь; следуя интуитивному представлению, отображенному на рис. 13.22, на втором уровне можно пропускать сразу по  $t$  узлов, если один из каждых  $t$  узлов содержит, по меньшей мере, две связи; применяя итеративный подход, мы приходим к заключению, что один из каждых  $t^j$  узлов должен содержать по меньшей мере  $j + 1$  связей.

Для создания узлов, обладающих таким свойством, мы выполняем рандомизацию, используя функцию, которая возвращает  $j + 1$  с вероятностью  $1/t^j$ . Имея  $j$ , мы создаем новый узел с  $j$  связями и вставляем его в список пропусков, используя ту же рекурсивную схему, которую использовали для выполнения операции *search*, как показано на рис. 13.23. После достижения уровня  $j$  мы связываем новый узел при каждом спуске на следующий уровень. К этому моменту уже установлено, что элемент в текущем узле меньше ключа поиска и связан (на уровне  $j$ ) с узлом, который не меньше ключа поиска.

Для инициализации списка пропусков мы строим заглавный узел с максимальным количеством уровней, которые будут разрешены в списке, и нулевыми указателями на всех уровнях. Программы 13.8 и 13.9 реализуют инициализацию и вставку для списков пропусков.

**Программа 13.8 Структуры данных и конструктор списка пропусков**

Узлы в списках пропусков содержат массив связей, поэтому конструктор для **node** должен распределить массив и установить все связи в 0. Константа **lgNmax** — максимальное количество уровней, которые разрешаются в списке: ее значение может быть установлено равным пяти для крошечных списков или 30 — для огромных. Переменная **N**, как обычно, хранит количество элементов в списке, а **lgN** — количество уровней. Пустой список является заглавным узлом с **lgNmax** связей, которые все установлены в 0, при нулевых **N** и **lgN**.

```
private:
    struct node
    { Item item; node **next; int sz;
      node(Item x, int k)
      { item = x; sz = k; next = new node*[k];
        for (int i = 0; i < k; i++) next[i] = 0; }
    };
    typedef node *link;
    link head;
    Item nullItem;
    int lgN;
public:
    ST(int)
    { head = new node(nullItem, lgNmax); lgN = 0; }
```

**Программа 13.9 Вставка в список пропусков**

Мы генерируем новый  $j$ -связный узел с вероятностью  $1/2^j$ , затем перемещаемся вдоль пути поиска точно так же, как в программе 13.7, но связываем новый узел при переходе на каждый из расположенных ниже  $j$  уровней.

```
private:
    int randX()
    { int i, j, t = rand();
      for (i = 1, j = 2; i < lgNmax; i++, j += j)
          if (t > RAND_MAX/j) break;
      if (i > lgN) lgN = i;
      return i;
    }
    void insertR(link t, link x, int k)
    { Key v = x->item.key(); link tk = t->next[k];
      if ((tk == 0) || (v < tk->item.key()))
      {
          if (k < x->sz)
              { x->next[k] = tk; t->next[k] = x; }
          if (k == 0) return;
          insertR(t, x, k-1); return;
      }
      insertR(tk, x, k);
    }
public:
    void insert(Item v)
    { insertR(head, new node(v, randX()), lgN); }
```

На рис. 13.24 демонстрируется построение списка пропусков для тестового набора ключей, вставляемых в случайном порядке; на рис. 13.25 приведен более объемный

пример, а на рис. 13.26 показано конструирование списка пропусков для тех же ключей, которые показаны на рис. 13.24, но вставленных в порядке возрастания. Подобно рандомизованным BST-деревьям, стохастические свойства списков пропусков не зависят от порядка вставки ключей.

**Лемма 13.10** Для поиска и вставки в рандомизованный список пропусков с параметром  $t$  в среднем требуется около  $(t \log_t N)/2 = (t / (2 \lg t)) \lg N$  сравнений.

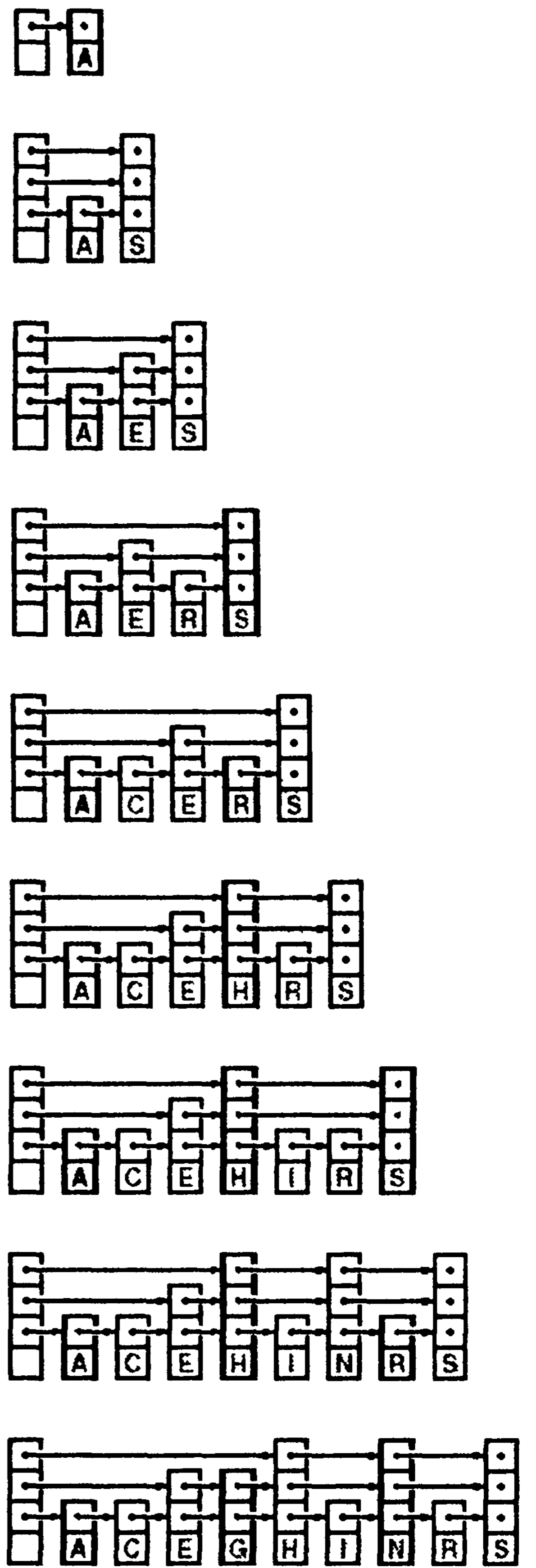
Мы ожидаем, что список пропусков должен иметь около  $\log_t N$  уровней, поскольку  $\log_t N$  больше наименьшего значения  $j$ , для которого  $t^j = N$ . На каждом уровне мы ожидаем, что на предыдущем уровне было пропущено около  $t$  узлов, а перед переходом на следующий уровень придется пройти приблизительно половину из них. Как видно из примера на рис. 13.25, количество уровней мало, но точное аналитическое обоснование этого не столь элементарно (см. раздел *ссылок*).

**Лемма 13.11** Списки пропусков имеют в среднем  $(t / (t - 1))N$  связей.

На нижнем уровне имеется  $N$  связей,  $N/t$  связей — на первом уровне, около  $N/t^2$  связей — на втором уровне и т.д., при общем количестве связей в списке равном примерно

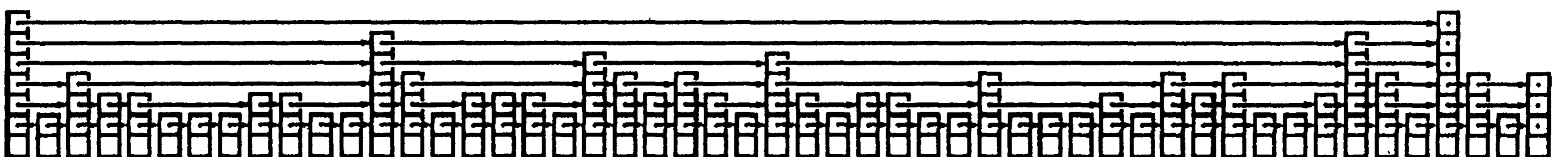
$$N(1 + 1/t + 1/t^2 + 1/t^3 \dots) = N / (1 - 1/t)$$

Выбор подходящего значения  $t$  немедленно приводит к необходимости отыскания компромисса между временем выполнения и занимаемым объемом памяти. При  $t = 2$  в списках пропусков требуется в среднем около  $\lg N$  сравнений и  $2N$  связей — этот уровень производительности сравним с лучшей производительностью при использовании BST-деревьев. Для больших значений  $t$  время поиска и вставки удлиняется, но дополнительный объем памяти, требуемый для связей, уменьшается. Продифференцировав выражение из леммы 13.10, мы находим, что выбор значения  $t = e$  ми-



**РИСУНОК 13.24 ПОСТРОЕНИЕ СПИСКА ПРОПУСКОВ**

Эта последовательность рисунков показывает результат вставки элементов с ключами *A S E R C H I N G* в первоначально пустой список пропусков. Узлы содержат  $j$  связей с вероятностью  $1/2^j$ .



**РИСУНОК 13.25 БОЛЬШОЙ СПИСОК ПРОПУСКОВ**

Этот список пропусков — результат вставки в случайном порядке 50 ключей в первоначально пустой список. Доступ к каждому из ключей можно получить, следуя вдоль не более чем 7 связей.

минимизирует ожидаемое количество сравнений, требуемое для выполнения поиска в списке пропусков.

В следующей таблице приведены значения коэффициента  $N \lg N$  в выражении, определяющем количество сравнений, необходимых для конструирования таблицы из  $N$  элементов:

$t$	2	$e$	3	4	8	16
$\lg t$	1.00	1.44	1.58	2.00	3.00	4.00
$t / \lg t$	2.00	1.88	1.89	2.00	2.67	4.00

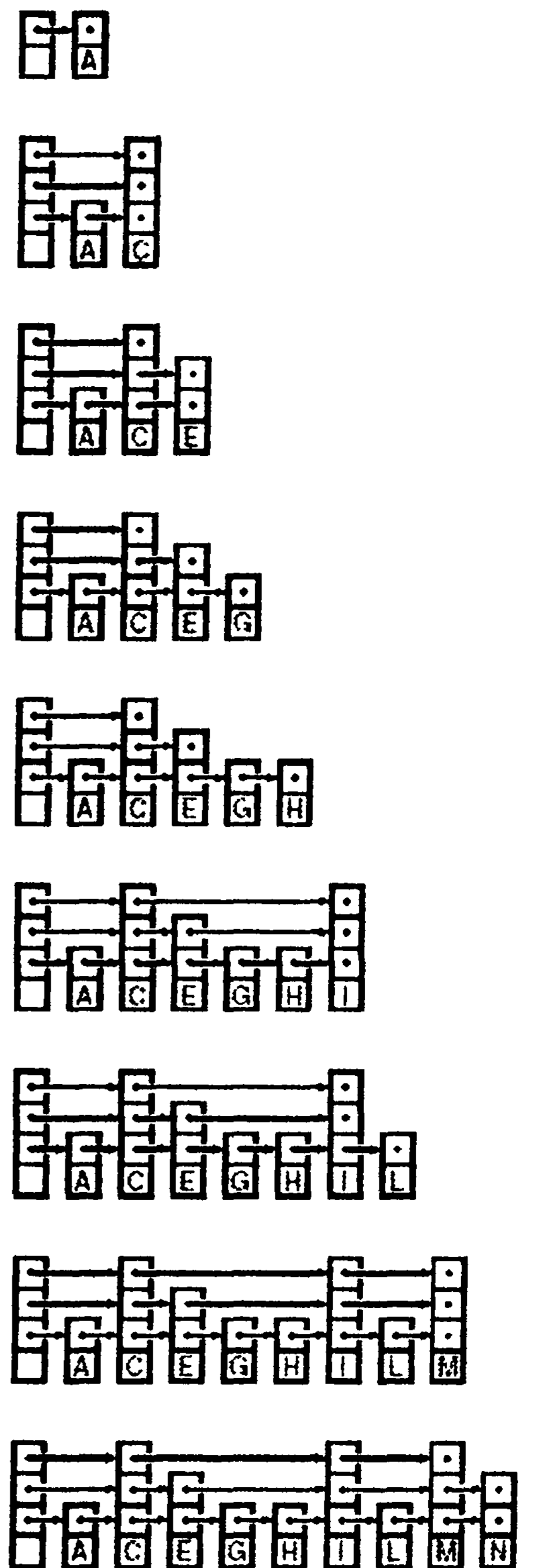
Если для рекурсивного выполнения сравнений, следования связям и перемещения вниз требуются затраты, которые существенно отличаются от приведенных значений, можно выполнить более точные расчеты (см. упражнение 13.83).

Поскольку время выполнения поиска определяется логарифмической зависимостью, перерасход памяти можно уменьшить до объема, не слишком превышающего требуемый для односвязных списков (если объем памяти ограничен), увеличив значение  $t$ . Точная оценка времени выполнения зависит от распределения относительных затрат на следование связям в списках и на переход вниз на следующий уровень. Мы вернемся к этому компромиссу между временем и объемом памяти в главе 16 при рассмотрении проблемы индексирования очень больших файлов.

Реализация других функций таблиц символов с помощью списков пропусков не представляет сложности. Например, в программе 13.10 приведена реализация функции *remove*, в которой применяется та же рекурсивная схема, которая использовалась для функции *insert* в программе 13.9. Для выполнения удаления мы разрываем связь узла со списком на каждом уровне (где он связывался для функции *insert*). Затем мы освобождаем узел после разрыва его связи с нижней частью списка (в отличие от его создания перед пересечением списка для вставки). Для реализации операции *join* списки объединяются (см. упражнение 13.78); для реализации операции *select* к каждому узлу добавляется поле, содержащее количество узлов, пропущенных его связью самого высокого уровня (см. упражнение 13.77).

### Программа 13.10 Удаление в списках пропусков

Для удаления узла с заданным ключом из списка пропусков мы разрываем его связь на каждом уровне, где находим связь к нему, а затем удаляем его по достижении нижнего уровня.



**РИСУНОК 13.26**  
ПОСТРОЕНИЕ СПИСКА ПРОПУСКОВ, СОДЕРЖАЩЕГО УПОРЯДОЧЕННЫЕ КЛЮЧИ

*Эта последовательность рисунков показывает результат вставки элементов с ключами A C E G H I N R S в первоначально пустой список пропусков. Стохастические свойства списка не зависят от порядка вставки ключей.*



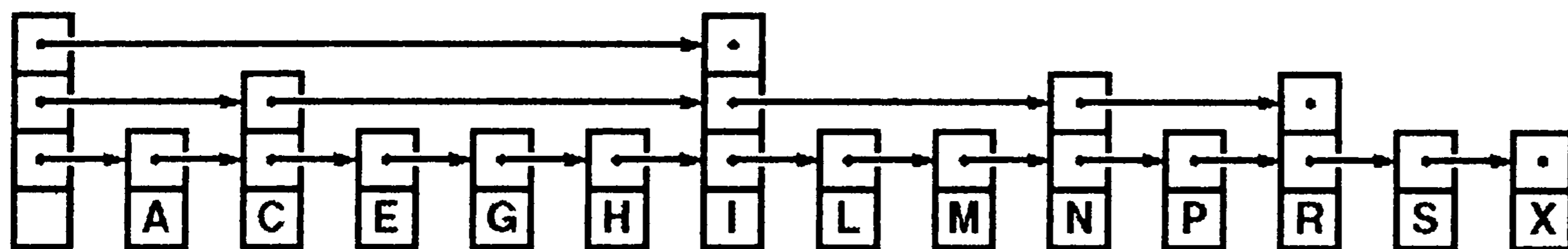
```

private:
    void removeR(link t, Key v, int k)
    { link x = t->next[k];
      if (!(x->item.key() < v))
        {
          if (v == x->item.key())
            { t->next[k] = x->next[k]; }
          if (k == 0) { delete x; return; }
          removeR(t, v, k-1); return;
        }
      removeR(t->next[k], v, k);
    }
public:
    void remove(Item x)
    { removeR(head, x.key(), lgN); }

```

Хотя списки пропусков легко обобщить в качестве систематического способа быстрого перемещения по связному списку, важно понимать, что лежащая в основе структура данных — всего лишь альтернативное представление сбалансированного дерева. Например, на рис. 13.27 приведено представление сбалансированного 2-3-4-дерева из рис. 13.10 в виде списка пропусков. Алгоритмы для сбалансированного 2-3-4-дерева из раздела 13.3, можно реализовать, используя абстракцию списка пропусков, а не абстракцию RB-дерева, описанную в разделе 13.4. Результирующий код несколько сложнее кода рассмотренных представлений (см. упражнение 13.80). В главе 16 мы еще вернемся к этой взаимосвязи между списками пропусков и сбалансированными деревьями.

Идеальный список пропусков, показанный на рис. 13.22 — жесткая структура, которую трудно поддерживать при вставке нового узла, как это имело место и в случае упорядоченного массива для реализации бинарного поиска, поскольку вставка сопряжена с изменением всех связей во всех узлах, следующих за вставленным. Один из способов сделать структуру более гибкой заключается в построении списков, в которых каждая связь пропускает одну, две или три связи на расположенном ниже уровне: эта организация соответствует 2-3-4-деревьям (см. рис. 13.27). Рандомизованный алгоритм, рассмотренный в этом разделе — еще один эффективный способ уменьшения жесткости структуры; в главе 16 будут рассматриваться и другие альтернативы.



**РИСУНОК 13.27 ПРЕДСТАВЛЕНИЕ 2-3-4-ДЕРЕВА В ВИДЕ СПИСКА ПРОПУСКОВ**

*Этот список пропусков — представление 2-3-4-дерева, показанного на рис. 13.10. В общем случае списки пропусков соответствуют сбалансированным многопозиционным деревьям с одной или более связью на узел (допускаются 1-узлы, не имеющие ключей и имеющие 1 связь). Для построения списка пропусков, соответствующего дереву, мы присваиваем каждому узлу количество связей, равное его высоте в дереве, а затем связываем узлы по горизонтали. Для построения дерева, соответствующего списку пропусков, мы группируем пропущенные узлы и рекурсивно связываем их с узлами на следующем уровне.*

## Упражнения

- 13.75** Нарисуйте список пропусков, образованный в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустой список, если считать, что **randX** возвращает последовательность значений 1, 3, 1, 1, 2, 2, 1, 4, 1 и 1.
- ▷ **13.76** Нарисуйте список пропусков, образованный в результате вставки элементов с ключами **E A I N O Q S T U Y** в указанном порядке в первоначально пустой список, если считать, что функция **randX** возвращает такие же значения, как и в упражнении 13.75.
- 13.77** Реализуйте операцию *select* для таблицы символов на базе списка пропусков.
- **13.78** Реализуйте операцию *join* для таблицы символов на базе списка пропусков.
- ▷ **13.79** Модифицируйте реализации операций *search* и *insert*, приведенные в программах 13.7 и 13.9, чтобы вместо нулевых они создавали списки с сигнальными узлами.
- **13.80** Задействуйте списки пропусков при реализации операций *construct*, *search* и *insert* для таблиц символов, использующих абстракцию сбалансированного 2-3-4-дерева.
- **13.81** Сколько случайных чисел требуется в среднем для построения списка пропусков с параметром  $t$ , с использованием функции **randX()** из программы 13.9?
- **13.82** Для  $t = 2$  измените программу 13.9 так, чтобы исключить цикл **for** из функции **randX**. *Совет:* последние  $j$  разрядов в бинарном представлении числа  $t$  принимают значение любого отдельного разряда  $j$  с вероятностью  $1/2^j$ .
- 13.83** Выберите значение  $t$ , которое минимизирует затраты на поиск для случая, когда затраты на следование по связи в  $\alpha$  раз превышают затраты на выполнение сравнения, а затраты на выполнение перехода на один уровень рекурсии вниз в  $\beta$  раз превышают затраты на выполнение сравнения.
- **13.84** Разработайте реализацию списка пропусков, в которой узлы содержат сами указатели, а не указатель на массив указателей, который использовался в программах 13.7 — 13.10. *Совет:* поместите массив в *конец* узла.

## 13.6 Характеристики производительности

Как для конкретного приложения произвести выбор между рандомизованными BST-деревьями, расширенными BST-деревьями, RB-деревьями бинарного поиска и списками пропусков? Долгих пор внимание было сосредоточено на различной природе гарантий производительности, обеспечиваемых этими алгоритмами. Время и занимаемый объем памяти всегда являются главным определяющим фактором, но необходимо учитывать также и ряд других факторов. В этом разделе мы кратко рассмотрим вопросы реализации, эмпирические исследования, оценку времени выполнения и требования к объему памяти.

Все алгоритмы, основанные на использовании деревьев, зависят от ротаций; реализация ротаций вдоль пути поиска — важная составляющая большинства алгоритмов для сбалансированных деревьев. Мы использовали рекурсивные реализации, которые неявным образом сохраняют указатели на узлы в пути поиска в локальных переменных в стеке рекурсии. Но каждый из алгоритмов может быть реализован и не рекурсивно, оперируя постоянным количеством узлов и выполняя постоянное количество операций связывания для одного узла в процессе нисходящего прохода по дереву.

Из трех основанных на деревьях алгоритмов проще всего реализовать рандомизированные BST-деревья. Главные требования — надежность генератора случайных чисел и избежание слишком больших затрат времени на генерирование случайных разрядов. Расширенные деревья несколько более сложны, но являются простым расширением стандартного алгоритма вставки в корень. RB-деревья бинарного поиска требуют еще большего объема кода для проверки и манипулирования разрядами цвета. Одно из преимуществ RB-деревьев по сравнению двумя другими алгоритмами — возможность использования разрядов цвета для проверки логики при отладке и для обеспечения быстрого поиска в любой момент времени в течение жизни дерева. Исследуя расширенное BST-дерево, невозможно выяснить, все ли необходимые преобразования выполняет создающий его код; программная ошибка может привести (только!) к проблемам, связанным с производительностью. Аналогично, ошибка в генераторе случайных чисел, используемом для рандомизированных BST-деревьев или списков пропусков, может привести к незамеченным в противном случае проблемам производительности.

Списки пропусков легко реализовать и они особенно привлекательны, если требуется поддерживать полный набор операций для таблиц символов, поскольку все операции *search*, *insert*, *remove*, *join*, *select* и *sort* имеют естественные реализации, которые легко сформулировать. Внутренний цикл для выполнения поиска в списках пропусков длиннее, чем в деревьях (для него требуется дополнительный индекс в массиве указателей или дополнительный рекурсивный вызов для перемещения на нижний уровень), поэтому время поиска и вставки удлиняется. Кроме того, списки пропусков ставят программиста в зависимость от генератора случайных чисел, поскольку отладка программы, поведение которой носит случайный характер, — весьма сложная задача, и некоторые программисты особенно не любят работать с узлами, содержащими случайное количество связей.

В табл. 13.1 приведены экспериментальные данные по производительности четырех рассмотренных в этой главе методов, а также реализаций элементарных BST-деревьев, описанных в главе 12, для ключей, которые являются случайными 32-разрядными целыми числами. Приведенные в этой таблице данные подтверждают аналитические результаты, полученные в разделах 13.2, 13.4 и 13.5. RB-деревья работают со случайными ключами гораздо быстрее других алгоритмов. Пути в них на 35 процентов короче, чем в рандомизированных или расширенных BSR-деревьях, в во внутреннем цикле приходится выполнять меньший объем работы. Рандомизированные деревья и списки пропусков требуют генерации, по меньшей мере, одного случайного числа для каждой вставки, а расширенные BST-деревья сопряжены с ротацией в каж-

дом узле во время выполнения вставки и поиска. И наоборот, накладные расходы при использовании RB-деревьев бинарного поиска заключаются в том, что для каждой вставки приходится проверять значение 2 разрядов в каждом узле, а иногда приходится выполнять и ротацию. При неравномерном доступе расширенные BST-деревья могут обеспечивать более короткие пути, но эта экономия, скорее всего, будет сводиться на нет тем, что и для поиска, и для вставки требуются ротации в каждом узле во внутреннем цикле, за исключением, быть может, экстремальных случаев.

**Таблица 13.1 Результаты экспериментального изучения реализаций сбалансированных деревьев**

Эти сравнительные значения времени построения и поиска в BST-деревьях, образованных случайными последовательностями  $N$  32-разрядных чисел, при различных значениях  $N$ , показывают, что все методы обеспечивают хорошую производительность даже для очень больших таблиц, но RB-деревья работают значительно быстрее других методов. Во всех методах используется стандартный поиск в BST-дереве, за исключением расширенных BST-деревьев, где при поиске выполняется расширение с целью перемещения часто посещаемых узлов ближе к вершине, и списков пропусков, в которых используется, по сути дела, этот же алгоритм, но по отношению к другой структуре данных.

$N$	конструирование						промахи при поиске					
	B	T	R	S	C	L	B	T	R	S	C	L
1250	0	1	3	2	1	2	1	1	0	0	0	2
2500	2	4	6	3	1	4	1	1	1	2	1	3
5000	4	7	14	8	5	10	3	3	3	3	2	7
12500	11	23	43	24	16	28	10	9	9	9	7	18
25000	27	51	101	50	32	57	19	19	26	21	16	43
50000	63	114	220	117	74	133	48	49	60	46	36	98
100000	159	277	447	282	177	310	118	106	132	112	84	229
200000	347	621	996	636	411	670	235	234	294	247	193	523

Обозначения:

- B Стандартное BST-дерево (программа 12.8)
- T BST-дерево, построенное при помощи вставок в корень (программа 12.13)
- R Рандомизованное BST-дерево (программа 13.2)
- S Расширенное BST-дерево (упражнение 13.33 и программа 13.5)
- C RB-дерево бинарного поиска (программа 13.6)
- L Список пропусков (программы 13.7 и 13.9)

Расширенные BST-деревья не требуют использования дополнительного объема памяти под информацию о балансировке; RB-деревья бинарного поиска требуют 1 дополнительного разряда; рандомизованные BST-деревья требуют наличия поля счетчика. Для многих приложений поле счетчика поддерживается по ряду других причин, поэтому оно может быть и не сопряжено с дополнительными затратами для рандомизованных BST-деревьев. Действительно, *добавление* этого поля может потребоваться

при использовании расширенных BST-деревьев, RB-деревьев бинарного поиска или списков пропусков. В случае необходимости, RB-деревья бинарного поиска можно сделать столь же эффективными в плане используемого объема памяти, как и расширенные BST-деревья, исключив разряд цвета (см. упражнение 13.65). В современных приложениях объем памяти — не столь критичный фактор, каким он был когда-то, однако истинный профессионал все же должен избегать напрасных затрат. Например, необходимо знать, что некоторые системы могут использовать все 32-разрядное слово для небольшого поля счетчика или 1-разрядного поля цвета в узле, в то время как другие системы могут упаковывать поля в памяти так, что их распаковка требует значительного дополнительного времени. Если объем памяти ограничен, списки пропусков с большим параметром  $t$  могут уменьшить объем памяти, требуемый для связей, почти в два раза ценой более медленного (но все же определяемого логарифмической зависимостью) поиска. В некоторых случаях основанные на деревьях методы могут быть также реализованы с использованием по одной связи на узел (см. упражнение 12.68).

Подводя итоги, можно сказать, что все рассмотренные в этой главе методы обеспечат высокую производительность для типовых приложений, при этом каждый метод обладает собственными достоинствами для тех, кто заинтересован в разработке высокопроизводительных реализаций таблиц символов. Расширенные BST-деревья обеспечат высокую производительность как метод самоорганизующегося поиска, особенно, когда частое обращение к небольшим наборам ключей является типичным случаем; рандомизированные BST-деревья, вероятно, будут работать быстрее и их проще реализовать для таблиц символов, которые должны обладать полным набором функций. Списки пропусков просты для понимания и могут обеспечить логарифмическую зависимость времени поиска при меньших затратах памяти, а RB-деревья бинарного поиска привлекательны для библиотечных реализаций таблиц символов, поскольку они обеспечивают гарантированные предельные характеристики производительности для худшего случая и являются наиболее быстрыми алгоритмами поиска и вставки случайных данных.

Кроме специфических использований в приложениях, это множество решений задачи разработки эффективных реализаций АДТ таблиц символов важно также и потому, что оно иллюстрирует фундаментальные подходы к разработке алгоритмов, которые можно задействовать и при поиске решения других задач. При постоянной потребности в простых оптимальных алгоритмах, мы часто сталкиваемся с почти оптимальными алгоритмами, подобными рассмотренным в этой главе. Более того, как можно было заметить в случае с сортировкой, алгоритмы, основанные на сравнении — лишь верхушка айсберга. Переходя к абстракциям более низкого уровня, где можно обрабатывать части ключей, можно разрабатывать реализации, которые работают еще быстрее исследованных в этой главе, что и будет показано в главах 14 и 15.

## Упражнения

**13.85** Разработайте реализацию таблицы символов, использующую рандомизованные BST-деревья, которая содержит деструктор, конструктор копирования и перегруженную операцию присваивания и поддерживает операции *construct*, *count*, *search*, *insert*, *remove*, *join*, *select* и *sort* для АДТ таблиц символов с дескрипторами клиентских элементов (см. упражнения 12.6 и 12.7).

**13.86** Разработайте реализацию таблицы символов, использующую списки пропусков, которая содержит деструктор, конструктор копирования и перегруженную операцию присваивания и поддерживает операции *construct*, *count*, *search*, *insert*, *remove*, *join*, *select* и *sort* для АДТ таблиц символов с дескрипторами клиентских элементов (см. упражнения 12.6 и 12.7).

## Хеширование

Рассматриваемые нами алгоритмы поиска основываются на абстрактной операции сравнения. Однако, метод поиска с использованием индексирования по ключу, описанный в разделе 12.2, при котором элемент с ключом  $i$  хранится в позиции  $i$  таблицы, что позволяет обратиться к нему немедленно, является существенным исключением из этого утверждения. При поиске с использованием индексирования по ключу значения ключей используются в качестве индексов массива, а не участвуют в сравнениях; при этом метод основывается на том, что ключи являются различными целыми числами из того же диапазона, что и индексы таблицы. В этой главе мы рассмотрим *хеширование (hashing)* — расширенный вариант поиска с использованием индексирования по ключу, применяемый в более типовых приложениях поиска, в которых не приходится рассчитывать на наличие ключей со столь удобными свойствами. Конечный результат применения данного метода коренным образом отличается от результата применения основанных на сравнении методов — вместо перемещения по структурам данных словаря со сравнением ключей поиска с ключами в элементах, предпринимается попытка обращения к элементам в таблице непосредственно, за счет выполнения арифметических операций для преобразования ключей в адреса таблицы.

Алгоритмы поиска, которые используют хеширование, состоят из двух отдельных частей. Первый шаг — вычисление *хеш-функции*, которая преобразует ключ поиска в адрес в таблице. В идеале различные ключи должны были бы отображаться на различные адреса, но часто два и более различных ключа могут преобразовываться в один и тот же адрес в таблице. Поэтому вторая часть поиска ме-

тодом хеширования — процесс *разрешения конфликтов*, который обрабатывает такие ключи. В одном из рассматриваемых методов разрешения конфликтов используются связные списки, поэтому он находит непосредственное применение в динамических ситуациях, когда заранее трудно предвидеть количество ключей поиска. В других двух методах разрешения конфликтов высокая производительность поиска обеспечивается для элементов, хранящихся в фиксированном массиве. Мы исследуем также способ усовершенствования этих методов с целью их расширения для случаев, когда нельзя заранее предсказать размеры таблицы.

Хеширование — хороший пример *компромисса между временем и объемом памяти*. Если бы на объем используемой памяти ограничения не накладывались, любой поиск можно было бы выполнить за счет всего лишь одного обращения к памяти, просто используя ключ в качестве адреса памяти, как это делается при поиске с использованием индексирования по ключу. Однако, часто этот идеальный случай оказывается недостижимым, поскольку требуемый объем памяти неприемлем, когда ключи являются длинными. С другой стороны, если бы не существовало ограничений на время выполнения, можно было бы обойтись минимальным объемом памяти, используя метод последовательного поиска. Хеширование предоставляет способ использования как приемлемого объема памяти, так и приемлемого времени с целью достижения компромисса между этими двумя крайними случаями. В частности, можно поддерживать любое выбранное соотношение, просто настраивая размер таблицы, а не переписывая код или выбирая другие алгоритмы.

Хеширование — одна из классических задач компьютерных наук: различные алгоритмы подробно исследованы и находят широкое применение. Мы увидим, что при ряде общих допущений можно надеяться на обеспечение поддержки операций *search* и *insert* в таблицах символов при *постоянном* времени выполнения независимо от размера таблицы.

Это ожидаемое постоянное время выполнения — теоретический оптимум производительности для любой реализации таблицы символов, но хеширование не является панацеей по двум основным причинам. Во-первых, время выполнения зависит от длины ключа, которая может быть значительной в реальных приложениях, использующих длинные ключи. Во-вторых, хеширование не обеспечивает эффективные реализации для других операций, таких как *select* или *sort*, с таблицами символов. Эти и другие вопросы подробно рассматриваются в этой главе.

## 14.1 Хеш-функции

Прежде всего, необходимо решить задачу вычисления хеш-функции, которая занимается преобразованием ключей в адреса в таблице. Обычно реализация этого математического вычисления не представляет сложности, но необходимо соблюдать определенную осторожность во избежание различных малозаметных ловушек. При наличии таблицы, которая может содержать  $M$  элементов, нам требуется функция, которая преобразует ключи в целые числа в диапазоне  $[0, M - 1]$ . Идеальную хеш-функцию легко вычислить и аппроксимировать случайной функцией: для любого вводимого значения в определенном смысле выводимое значение должно быть равновероятным.



Хеш-функция зависит от типа ключа. Строго говоря, для каждого вида ключей, который может использоваться, требуется отдельная хеш-функция. Для повышения эффективности в общем случае желательно избежать явного преобразования типов, снова обратившись вместо этого к идее рассмотрения двоичного представления ключей в машинном слове в виде целого числа, которое можно использовать в арифметических вычислениях. Хеширование предшествовало появлению языков высокого уровня — на первых компьютерах было типичным в один момент времени рассматривать значение как строковый ключ, а в следующий — как целое число. В некоторых языках высокого уровня затруднительно создавать программы, которые зависят от того, как ключи представляются на конкретном компьютере, поскольку такие программы, по самой своей природе, являются машинно-зависимыми и поэтому их трудно перенести на другой компьютер. В общем случае хеш-функции зависят от процесса преобразования ключей в целые числа, поэтому в реализациях хеширования иногда трудно одновременно обеспечить независимость от компьютера и эффективность. Как правило, простое целочисленное значение или ключи типа с плавающей точкой можно преобразовать, выполнив всего одну машинную операцию, но строковые ключи и другие типы составных ключей требуют больше затрат и больше внимания в плане достижения высокой эффективности.

Вероятно, простейшей является ситуация, когда ключами являются числа с плавающей точкой, заведомо относящиеся к фиксированному диапазону. Например, если ключи — числа, которые больше 0 и меньше 1, их можно просто умножить на  $M$ , округлить до ближайшего целого числа и получить адрес в диапазоне между 0 и  $M - 1$ ; пример показан на рис. 14.1. Если ключи больше  $s$  и меньше  $t$ , их можно масштабировать, вычтя  $s$  и разделив на  $t - s$ , в результате чего они попадут в диапазон значений между 0 и 1, а затем умножить на  $M$  для получения адреса в таблице.

Если ключи — это  $w$ -разрядные целые числа, их можно преобразовать в числа с плавающей точкой и разделить на  $2^w$  для получения чисел с плавающей точкой в диапазоне между 0 и 1, а затем умножить на  $M$ , как в предыдущем абзаце. Если операции с плавающей точкой выполняются часто, а числа не столь велики, чтобы при-

.513870656	51
.175725579	17
.308633685	30
.534531713	53
.947630227	94
.171727657	17
.702230930	70
.226416826	22
.494766086	49
.124698631	12
.083895385	8
.389629811	38
.277230144	27
.368053228	36
.983458996	98
.535386205	53
.765678883	76
.646473587	64
.767143786	76
.780236185	78
.822962105	82
.151921138	15
.625476837	62
.314676344	31
.346903890	34

#### РИСУНОК 14.1 МУЛЬТИПЛИКАТИВНАЯ ХЕШ- ФУНКЦИЯ ДЛЯ КЛЮЧЕЙ С ПЛАВАЮЩЕЙ ТОЧКОЙ

*Для преобразования чисел с плавающей точкой в диапазоне между 0 и 1 в индексы таблицы, размер которой равен 97, выполняется умножение на 97. В этом примере имеют место три конфликта: при значениях индексов равных 17, 53 и 76. Старшие разряды ключа определяют хеш-значения; младшие разряды ключей не играют никакой роли. Одна из целей разработки хеш-функции — избежание дисбаланса, когда во время вычисления каждый разряд данных играет определенную роль.*

водить к переполнению, этот же результат может быть получен с использованием арифметических операций над целыми числами: необходимо умножить ключ на  $M$ , затем выполнить сдвиг вправо на  $w$  разрядов для деления на  $2^w$  (или, если умножение приводит к переполнению, выполнить сдвиг, а затем умножение). Такие функции бесполезны для хеширования, если только ключи не распределены по диапазону равномерно, поскольку хеш-значение определяется только ведущими цифрами ключа.

Более простой и эффективный метод для  $w$ -разрядных целых чисел — один из, вероятно, наиболее часто используемых методов хеширования — выбор в качестве размера  $M$  таблицы простого числа и вычисление остатка от деления  $k$  на  $M$ , или  $h(k) = k \bmod M$  для любого целочисленного ключа  $k$ . Такая функция называется *модульной хеш-функцией*. Ее очень просто вычислить ( $k \% M$  в языке C++), и она эффективна для достижения равномерного распределения значений ключей между значениями, которые меньше  $M$ . Небольшой пример показан на рис. 14.2.

Модульное хеширование можно использовать также для ключей с плавающей точкой. Если ключи относятся к небольшому диапазону, можно выполнить масштабирование с целью их преобразования в числа из диапазона между 0 и 1, умножить на  $2^w$  для получения  $w$ -разрядных целочисленных значений, а затем использовать модульную хеш-функцию. Другая альтернатива — просто использовать в качестве операнда модульной хеш-функции двоичное представление ключа (если оно доступно).

Модульное хеширование применяется во всех случаях, когда имеется доступ к разрядам, образующим ключи, независимо от того, являются ли они целыми числами, представленными машинным словом, последовательностью символов, упакованных в машинное слово, или представлены одним из множества других возможных вариантов. Последовательность случайных символов, упакованная в машинное слово — не совсем то же, что случайные целочисленные ключи, поскольку некоторые разряды используются для кодирования. Но оба эти типа (и любой другой тип ключа, который кодируется так, чтобы уместиться в машинном слове) можно заставить *выглядеть* случайными индексами в небольшой таблице.

16838	57	38	6
5758	35	58	58
10113	25	13	50
17515	55	15	24
31051	11	51	90
5627	1	27	77
23010	21	10	20
7419	47	19	85
16212	13	12	19
4086	12	86	25
2749	33	49	98
12767	60	67	90
9084	63	84	14
12060	32	60	53
32225	21	25	16
17543	83	43	42
25089	63	89	5
21183	37	83	91
25137	14	37	35
25566	55	66	0
26966	0	66	65
4978	31	78	76
20495	28	95	66
10311	29	11	72
11367	18	67	25

#### РИСУНОК 14.2 МОДУЛЬНАЯ ХЕШ-ФУНКЦИЯ ДЛЯ ЦЕЛОЧИСЛЕННЫХ КЛЮЧЕЙ

*В трех правых столбцах показан результат хеширования 16-разрядных ключей, приведенных слева, с помощью следующих функций  $v \% 97$  (слева),  $v \% 100$  (в центре) и  $(int)(a * v) \% 100$  (справа), где  $a = .618033$ . Размеры таблицы для этих функций соответственно равны 97, 100 и 100. Значения оказываются случайными (поскольку ключи случайны). Центральная функция ( $v \% 100$ ) использует только две крайние справа цифры ключей и поэтому может показывать низкую производительность применительно к неслучайным ключам.*

Основная причина выбора в качестве размера  $M$  хеш-таблицы простого числа для модульного хеширования иллюстрируется на рис. 14.3. В этом примере символьных данных с 7-разрядным кодированием ключ трактуется как число с основанием 128 — по одной цифре для каждого символа в ключе. Слово `now` соответствует числу 1816567, которое может быть также записано как

$$110 \cdot 128^2 + 111 \cdot 128^1 + 119 \cdot 128^0$$

поскольку в ASCII-коде символам `n`, `o` и `w` соответствуют числа  $156_8 = 110$ ,  $157_8 = 111$  и  $167_8 = 119$ . Далее, выбор размера таблицы  $M = 64$  неудачен для этого типа ключа, поскольку добавление к  $x$  значений, кратных 64 (или 128), не оказывает влияния на значение  $x \bmod 64$  — для любого ключа значением хеш-функции является значение последних 6 разрядов этого ключа. Безусловно, хорошая хеш-функция должна учитывать все разряды ключа, особенно для ключей, образованных символами. Аналогичные ситуации могут возникать, когда  $M$  содержит множитель, являющийся степенью 2. Простейший способ избежать этого — выбрать в качестве  $M$  простое число.

Модульное хеширование весьма просто реализовать, за исключением того, что размер таблицы необходимо определить простым числом. Для некоторых приложений можно довольствоваться небольшим известным простым числом или же поискать простое число, близкое к требуемому размеру таблицы, в списке известных простых чисел. Например, числа равные  $2^t - 1$  являются простыми при  $t = 2, 3, 5, 7, 13, 19$  и  $31$  (и ни при каких других значениях  $t < 31$ ): это хорошо известные *простые числа Мерсенне* (Mersenne). Чтобы динамически распределить таблицу определенных размеров, нужно вычислить простое число, близкое к определенному значению. Это вычисление не тривиально (хотя и существует остроумный алгоритм выполнения этой задачи, который исследуется в части 5), поэтому на практике обычно используют таблицу заранее вычисленных значений (см. рис. 14.4). Использование модульного хеширования — не единственная причина, по которой размер таблицы определяется простым числом; другая причина рассматривается в разделе 14.4.

<code>now</code>	6733767	1816567	55	29
<code>for</code>	6333762	1685490	50	20
<code>tip</code>	7232360	1914096	48	1
<code>ilk</code>	6473153	1734251	43	18
<code>dim</code>	6232355	1651949	45	21
<code>tag</code>	7230347	1913063	39	22
<code>jot</code>	6533764	1751028	52	24
<code>sob</code>	7173742	1898466	34	26
<code>nob</code>	6733742	1816546	34	8
<code>sky</code>	7172771	1897977	57	2
<code>hut</code>	6435364	1719028	52	16
<code>ace</code>	6070745	1602021	37	3
<code>bet</code>	6131364	1618676	52	11
<code>men</code>	6671356	1798894	46	26
<code>egg</code>	6271747	1668071	39	23
<code>few</code>	6331367	1684215	55	16
<code>jay</code>	6530371	1749241	57	4
<code>owl</code>	6775754	1833964	44	4
<code>joy</code>	6533771	1751033	57	29
<code>rap</code>	7130360	1880304	48	30
<code>gig</code>	6372347	1701095	39	1
<code>wee</code>	7371345	1962725	37	22
<code>was</code>	7370363	1962227	51	20
<code>cab</code>	6170342	1634530	34	24
<code>wad</code>	7370344	1962212	36	5

#### РИСУНОК 14.3 МОДУЛЬНЫЕ ХЕШ-ФУНКЦИИ ДЛЯ КОДИРОВАННЫХ СИМВОЛОВ

*В каждой строке этой таблицы приведены 3-символьное слово, представление этого слова в ASCII-коде как 21-разрядное число в восьмеричной и десятичной формах и стандартные модульные хеш-функции для таблицы с размерами 64 и 31 (два крайних справа столбца). Размер таблицы, равный 64, приводит к нежелательным результатам, поскольку для получения хеш-значения используются только самые правые разряды ключа, а символы в словах обычного языка распределены неравномерно. Например, всем словам, завершающимся на букву `u`, соответствует хеш-значение 57. И напротив, простое значение 31 вызывает меньше конфликтов в таблице, размер которой составляет менее половины предыдущей.*

Другая альтернатива обработки целочисленных ключей — объединение мультипликативного и модульного методов: следует умножить ключ на константу в диапазоне между 0 и 1, а затем выполнить деление по модулю  $M$ . Другими словами, необходимо использовать функцию  $h(k) = \lfloor k\alpha \rfloor \bmod M$ . Между значениями  $\alpha$ ,  $M$  и эффективным основанием ключа существует взаимодействие, которое теоретически могло бы привести к аномальному поведению, но если использовать умеренное значение  $\alpha$ , в реальном приложении вряд ли придется столкнуться с какой-либо проблемой. Часто в качестве  $\alpha$  выбирают значение  $\phi = 0.618033\dots$  (*золотое сечение*). Изучено множество других вариаций на эту тему, в частности хеш-функции, которые могут быть реализованы с помощью таких эффективных машинных инструкций, как сдвиг и маскирование (см. раздел ссылок).

Во многих приложениях, в которых используются таблицы символов, ключи не являются числами и не обязательно являются короткими; чаще они оказываются алфавитно-цифровыми строками, которые могут быть длинными. Как вычислить хеш-функцию для такого слова, как

averylongkey?

В 7-разрядном ASCII-коде этому слову соответствует 84-разрядное число

$$97 \cdot 128^{11} + 118 \cdot 128^{10} + 101 \cdot 128^9 + 114 \cdot 128^8 + 121 \cdot 128^7 + 108 \cdot 128^6 + 111 \cdot 128^5 + 110 \cdot 128^4 + 103 \cdot 128^3 + 107 \cdot 128^2 + 101 \cdot 128^1 + 121 \cdot 128^0,$$

которое слишком велико, чтобы его можно было представить для обычных арифметических функций в большинстве компьютеров. Более того, часто требуется обрабатывать значительно более длинные ключи.

Чтобы вычислить модульную хеш-функцию для длинных ключей, последние преобразуются фрагмент за фрагментом. Можно воспользоваться арифметическими свойствами функции **mod** и задействовать алгоритм Горнера (Horner) (см. раздел 4.9).

$n$	$\delta_n$	$2^n - \delta_n$
8	5	251
9	3	509
10	3	1021
11	9	2039
12	3	4093
13	1	8191
14	3	16381
15	19	32749
16	15	65521
17	1	131071
18	5	262139
19	1	524287
20	3	1048573
21	9	2097143
22	3	4194301
23	15	8388593
24	3	16777213
25	39	33554393
26	5	67108859
27	39	134217689
28	57	268435399
29	3	536870909
30	35	1073741789
31	1	2147483647

#### РИСУНОК 14.4 ПРОСТЫЕ ЧИСЛА ДЛЯ ХЕШ-ТАБЛИЦ

*Эта таблица наибольших простых чисел, которые меньше  $2^n$  для  $8 \leq n \leq 32$ , может использоваться для динамического распределения хеш-таблицы, когда требуется, чтобы размер таблицы определялся простым числом. Для любого данного положительного значения в указанном диапазоне эту таблицу можно использовать для получения простого числа, отличающегося от него менее чем в 2 раза.*

Этот метод основывается на еще одном способе записи чисел, соответствующих ключам. Для рассматриваемого примера запишем следующее выражение:

$$\begin{aligned} &((((((((((97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114) \cdot 128 + 121) \cdot 128 \\ &\quad + 108) \cdot 128 + 111) \cdot 128 + 110) \cdot 128 + 103) \cdot 128 \\ &\quad + 107) \cdot 128 + 101) \cdot 128 + 121. \end{aligned}$$

То есть, можно вычислить десятичное число, соответствующее коду символа строки при просмотре ее слева направо, умножая полученное значение на 128, а затем добавляя кодовое значение следующего символа. Со временем, в случае длинной строки, этот способ вычисления создал бы число, которое больше того, которое вообще можно представить в компьютере. Однако вычисленное число нас не интересует, поскольку требуется только остаток от его деления на  $M$ , который мал. Результат можно получить, даже не сохраняя большое накопленное значение, т.к. в любой момент вычисления можно отбросить число, кратное  $M$  — при каждом выполнении умножения и сложения нужно хранить только остаток деления по модулю  $M$ . Результат такой же, как если бы у нас имелась возможность вычислять длинное число, а затем выполнять деление (см. упражнение 14.10). Это наблюдение ведет к непосредственному арифметическому способу вычисления модульных хеш-функций для длинных строк (см. программу 14.1). В программе используется еще одно, последнее ухищрение: вместо основания 128 в ней используется простое число 127. Причина такого изменения рассматривается в следующем абзаце.

#### Программа 14.1 Хеш-функция для строковых ключей

Эта реализация хеш-функции для строковых ключей требует одного умножения и одного сложения на каждый символ в ключе. Если бы константу 127 мы заменили константой 128, программа просто вычисляла бы остаток от деления числа, соответствующего 7-разрядному ASCII-представлению ключа, на размер таблицы с использованием метода Горнера. Простое основание, равное 127, помогает избежать аномалий, если размер таблицы является степенью 2 или кратным 2.

```
int hash(char *v, int M)
{ int h = 0, a = 127;
  for (; *v != 0; v++)
    h = (a*h + *v) % M;
  return h;
}
```

Существует множество способов вычисления хеш-функций приблизительно при тех же затратах, что и при выполнении модульного хеширования с использованием метода Горнера (при выполнении одной-двух арифметических операций для каждого символа в ключе). Для случайных ключей методы мало отличаются от описанного, но и реальные ключи едва ли можно считать случайными. Возможность ценой небольших затрат придать реальным ключам случайный вид приводит к рассмотрению *рандомизованных* алгоритмов хеширования — нам требуются хеш-функции, которые создают случайные индексы таблицы независимо от существующих ключей. Рандомизацию организовать не трудно, поскольку вовсе не требуется буквально придерживаться определения модульного хеширования — нужно всего лишь, чтобы

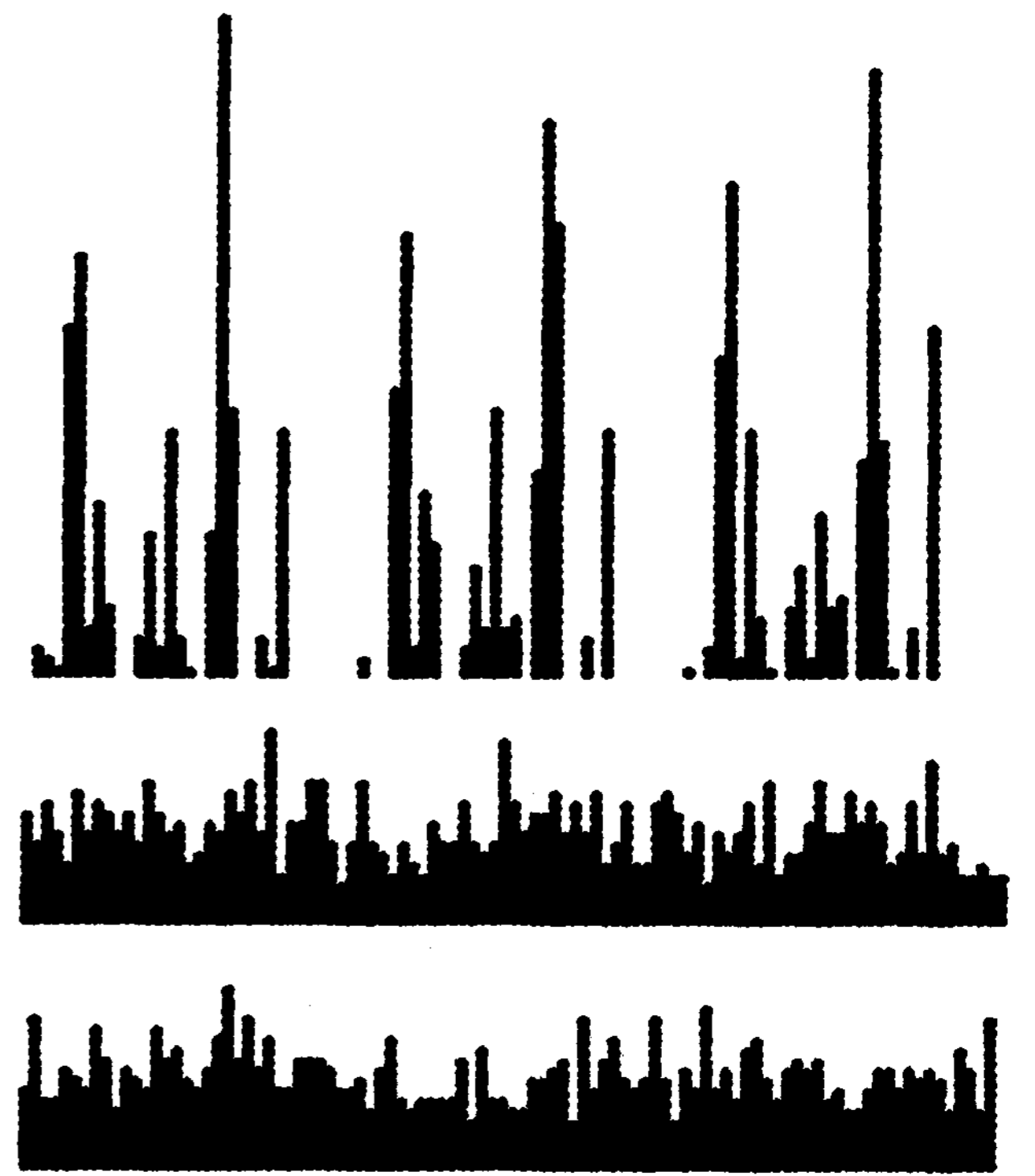
в вычислении целого числа, меньшего  $M$ , использовались все разряды ключа. В программе 14.1 показан один из способов выполнения этого: использование простого основания вместо степени 2, связанного с задействованием целого числа, которое соответствует ASCII-представлению строки. На рис. 14.5 показано, как это изменение препятствует плохому распределению типовых строковых ключей. Теоретически хеш-значения, созданные программой 14.1, могут не подходить для размеров таблицы, которые кратны числу 127 (хотя на практике это, скорее всего, скажется минимальным образом); для создания рандомизованного алгоритма можно было бы выбрать значение множителя наугад. Еще более эффективный подход — использование случайных значений коэффициентов в вычислении и другого случайного значения для каждой цифры ключа. Такой подход дает рандомизованный алгоритм, называемый *универсальным хешированием*.

Теоретически идеальная универсальная хеш-функция — это функция, для которой вероятность конфликта между двумя различными ключами в таблице размером  $M$  равна в точности  $1/M$ . Можно доказать, что использование в качестве коэффициента  $a$  в программе 14.1 последовательности случайных различных значений вместо фиксированного произвольного значения преобразует модульное хеширование в универсальную хеш-функцию. Однако затраты на генерирование нового случайного числа для каждого символа в ключе, скорее всего, окажутся неприемлемыми. В программе 14.2 продемонстрирован практический компромисс: мы варьируем коэффициенты, генерируя простую псевдослучайную последовательность.

### Программа 14.2 Универсальная хеш-функция (для строковых ключей)

Эта программа выполняет те же вычисления, что и программа 14.1, однако для аппроксимации вероятности возникновения конфликтов для двух несовпадающих ключей до значения  $1/M$  вместо фиксированных оснований системы счисления применяются псевдослучайные значения коэффициентов. С целью минимизации нежелательных временных затрат при вычислении хеш-функции используется грубый генератор случайных чисел.

```
int hashU(char *v, int M)
{ int h, a = 31415, b = 27183;
  for (h = 0; *v != 0; v++, a = a*b % (M-1))
    h = (a*h + *v) % M;
  return (h < 0) ? (h + M) : h;
}
```



**РИСУНОК 14.5 ХЕШ-ФУНКЦИИ ДЛЯ СИМВОЛЬНЫХ СТРОК**

*На этих диаграммах показано распределение для набора английских слов (первых 1000 слов романа Мелвилла "Моби Дик") в случае применения программы 14.1 при*

$M = 96$  и  $a = 128$  (вверху)

$M = 97$  и  $a = 128$  (в центре) и

$M = 96$  и  $a = 127$  (внизу)

*Неравномерное распределение в первом случае является*

*результатом неравномерного употребления букв и того, что и размер таблицы, и множитель кратны 32, что ведет к сохранению неравномерности.*

*Остальные два примера выглядят случайными, поскольку размер таблицы и множитель являются взаимно простыми числами.*

Подведем итоги: чтобы использовать хеширование для реализации абстрактной таблицы символов, в качестве первого шага необходимо расширить интерфейс абстрактного типа, включив в него операции `hash`, которая отображает ключи на неотрицательные целые числа, меньше размера таблицы `M`. Непосредственная реализация

```
inline int hash(Key v, int M)
{ return (int) M*(v-s)/(t-s); }
```

выполняет эту задачу для ключей с плавающей точкой, имеющих значения между `s` и `t`; для целочисленных ключей можно просто вернуть значение `v % M`. Если `M` не является простым числом, хеш-функция может возвращать

```
(int) (.616161 * (float) v) % M
```

или результат аналогичного целочисленного вычисления, такой как

```
(16161 * (unsigned) v) % M
```

Все эти функции, включая программу 14.1, работающую со строковыми ключами, проверены временем; они равномерно распределяют ключи и служат программистам в течение многих лет. Универсальный метод, представленный в программе 14.2 — заметное усовершенствование для строковых ключей, которое обеспечивает случайные хеш-значения при небольших затратах; аналогичные рандомизованные методы можно применить к целочисленным ключам (см. упражнение 14.1).

В конкретном приложении универсальное хеширование может работать значительно медленнее более простых методов, поскольку в случае длинных ключей для выполнения двух арифметических операций для каждого символа в ключе может затрачиваться слишком большое время. Для обхода этого ограничения ключи можно обрабатывать большими фрагментами. Действительно, наряду с элементарным модульным хешированием можно использовать наибольшие фрагменты, которые помещаются в машинное слово. Как подробно рассматривалось ранее, подобного вида операция может оказаться труднореализуемой или требовать специальных средств в некоторых строго типизованных языках высокого уровня, однако она может требовать малых затрат или не требовать абсолютно никакой работы в C++, если использовать вычисления с подходящими форматами представления данных. Во многих ситуациях важно учитывать эти факторы, поскольку вычисление хеш-функции может выполняться во внутреннем цикле, следовательно, ускоряя хеш-функцию, можно ускорить все вычисление.

Несмотря на очевидные преимущества рассмотренных методов, их реализация требует внимания по двум причинам. Во-первых, необходимо быть внимательным во избежание ошибок при преобразовании типов и использовании арифметических функций применительно к различным машинным представлениям ключей. Подобные операции часто являются источниками ошибок, особенно при переносе программы со старого компьютера на новый с другим количеством разрядов в слове или другой точностью выполнения операций. Во-вторых, весьма вероятно, что во многих приложениях вычисление хеш-функции будет выполняться во внутреннем цикле и время ее выполнения может в значительной степени определять общее время выполнения. В подобных случаях важно убедиться, что функция сводится к эффективному машинному коду. Подобные операции — известные источники снижения эффективности;

например, разность времени выполнения простого модульного метода и версии, в которой вначале выполняется умножение на 0.61616, может быть поразительной на компьютерах с низкой производительностью аппаратуры или программного обеспечения операций с плавающей точкой. Наиболее быстрый метод для многих компьютеров — сделать  $M$  степенью 2 и воспользоваться хеш-функцией

```
inline int hash(Key v, int M)
{ return v & (M-1); }
```

Эта функция использует только самые младшие разряды ключей, но операция побитового *and* выполняется существенно быстрее целочисленного деления, тем самым минимизируя нежелательные эффекты плохого распределения ключей.

Типичная ошибка в реализациях хеширования заключается в том, что хеш-функция всегда возвращает одно и то же значение, возможно потому, что требуемое преобразование типов выполняется неправильно. Такая ошибка называется ошибкой *производительности*, поскольку использующая подобную хеш-функцию программа вполне может выполняться корректно, но крайне медленно (т.к. ее эффективная работа возможна только, если хеш-значения распределены равномерно). Однострочные реализации этих функций очень легко тестировать, поэтому мы настоятельно рекомендуем проверять, насколько успешно они работают для типов ключей, которые могут встретиться в любой конкретной реализации таблицы символов.

Для проверки гипотезы, что хеш-функция создает случайные значения, можно использовать функцию статистического распределения  $\chi^2$  (см. упражнение 14.5), но, возможно, это требование — слишком жесткое. Действительно, нас вполне может удовлетворить, если хеш-функция создает каждое значение одинаковое количество раз, что соответствует значению функции статистического распределения  $\chi^2$ , равному 0, и местами не является случайным. Тем не менее, следует с подозрением относиться к очень большим значениям функции статистического распределения  $\chi^2$ . На практике, вероятно, достаточно использовать проверку того, что значения распределены до такой степени, чтобы ни одно из значений не доминировало (см. упражнение 14.15).

По этим же соображениям хорошо разработанная реализация таблицы символов, основанная на универсальном хешировании, могла бы периодически проверять, что хеш-значения не являются неравномерно распределенными. Клиента можно было бы информировать *либо* о том, что имело место маловероятное событие, *либо* о наличии ошибки в хеш-функции. Подобного рода проверка оказалась бы разумным дополнением к любому реальному рандомизованному алгоритму.

## Упражнения

▷ 14.1 Используя абстракцию **digit** из главы 10 для обработки машинного слова как последовательности байтов, реализуйте рандомизованную хеш-функцию для ключей, которые представлены разрядами в машинных слова.

14.2 Проверьте, сопряжено ли преобразование 4-байтового ключа в 32-разрядное целое число в используемой программной среде с какими-либо накладными расходами в плане времени выполнения.



- **14.3** Разработайте хеш-функцию для строковых ключей, основанную на идее одновременной загрузки 4 байт с последующим выполнением арифметических операций сразу над 32 разрядами. Сравните время выполнения этой функции со временем выполнения программы 14.1 для 4-, 8-, 16- и 32-байтовых ключей.

**14.4** Создайте программу для определения значений  $a$  и  $M$  при минимально возможном значении  $M$ , чтобы хеш-функция  $a*x \% M$  создавала различные (несовпадающие) значения для ключей, представленных на рис. 14.2. Результат является примером *совершенной хеш-функции*.

- **14.5** Создайте программу для вычисления функции статистического распределения  $\chi^2$  для хеш-значений  $N$  ключей при размере таблицы, равном  $M$ . Это число определяется равенством

$$\chi^2 = \frac{M}{N} \sum_{0 \leq i < M} \left( f_i - \frac{N}{M} \right)^2,$$

где  $f_i$  — количество ключей с хеш-значением  $i$ . Если хеш-значения являются случайными, значение этой функции статистического распределения для  $N > cM$  должно быть равно  $M \pm \sqrt{M}$  с вероятностью  $1 - 1/c$ .

**14.6** Воспользуйтесь программой из упражнения 14.5 для вычисления хеш-функции  $618033*x \% 10000$  для ключей, которые являются случайными положительными целыми числами, меньшими  $10^6$ .

**14.7** Используйте программу из упражнения 14.5 для вычисления хеш-функции из программы 14.1 для различных строковых ключей, полученных из какого-либо большого файла в системе, например, словаря.

- **14.8** Предположите, что ключи являются  $t$ -разрядными целыми числами. Для модульной хеш-функции с простым основанием  $M$  докажите, что каждый разряд ключа обладает тем свойством, что существуют два ключа, различающиеся только этим разрядом при различных хеш-значениях.

**14.9** Рассмотрите идею реализации модульного хеширования для целочисленных ключей с помощью соотношения  $(a*x) \% M$ , где  $a$  — произвольное фиксированное простое число. Приводит ли это изменение к достаточному перемешиванию разрядов, чтобы можно было использовать значение  $M$ , не являющееся простым числом?

**14.10** Докажите, что

$$(((ax) \bmod M) + b) \bmod M = (ax + b) \bmod M,$$

при условии, что  $a$ ,  $b$ ,  $x$  и  $M$  — неотрицательные целые числа.

- ▷ **14.11** Если в упражнении 14.7 использовать слова из текстового файла, например книги, вряд ли удастся получить хорошую функцию статистического распределения  $\chi^2$ . Обоснуйте справедливость этого утверждения.

**14.12** Воспользуйтесь программой из упражнения 14.5 для вычисления хеш-функции  $97*x \% M$  для всех размеров таблицы в диапазоне от 100 до 200, используя в качестве ключей  $10^3$  случайных положительных целых чисел, меньших  $10^6$ .

**14.13** Воспользуйтесь программой из упражнения 14.5 для вычисления хеш-функции  $97 * x \% M$  для всех размеров таблицы в диапазоне от 100 до 200, используя в качестве ключей целые числа в диапазоне между  $10^2$  и  $10^3$ .

**14.14** Воспользуйтесь программой из упражнения 14.5 для вычисления хеш-функции  $100 * x \% M$  для всех размеров таблицы в диапазоне от 100 до 200, используя в качестве ключей  $10^3$  случайных положительных целых чисел, меньших  $10^6$ .

**14.15** Выполните упражнения 14.12 и 14.14, но реализуйте более простой критерий отбрасывания хеш-функций, которые создают любое значение более  $3N/M$  раз.

## 14.2 Раздельное связывание

Рассмотренные в разделе 14.1 хеш-функции преобразуют ключи в адреса таблицы; второй компонент алгоритма хеширования — определения способа обработки случая, когда два ключа представляются одним и тем же адресом. Самый прямой метод — построить для каждого адреса таблицы связный список элементов, ключи которых отображаются на этот адрес. Данный подход ведет непосредственно к обобщению метода элементарного поиска в списке (см. главу 12) из программы 14.3. Вместо поддержки единственного списка поддерживаются  $M$  списков.

### Программа 14.3 Хеширование с помощью раздельного связывания

Эта реализация таблицы символов сводится к замене конструктора ST, функций **search** и **insert** в основанный на связном списке таблице символов из программы 12.6 на приведенные здесь функции и к замене связи **head** на массив связей **heads**. В этой программе используются такие же рекурсивные процедуры поиска и удаления в списке, как и в программе 12.6, но при этом поддерживается  $M$  списков с заглавными связями в **heads**, с использованием хеш-функции для выбора между списками. Конструктор устанавливает  $M$  так, что каждый список должен содержать около пяти элементов; поэтому для выполнения остальных операций требуется всего несколько проверок.

```
private:
    link* heads;
    int N, M;
public:
    ST(int maxN)
    {
        N = 0; M = maxN/5;
        heads = new link[M];
        for (int i = 0; i < M; i++) heads[i] = 0;
    }
    Item search(Key v)
    { return searchR(heads[hash(v, M)], v); }
    void insert(Item item)
    { int i = hash(item.key(), M);
      heads[i] = new node(item, heads[i]); N++; }
```

Метод традиционно называется *раздельным связыванием* (*separate chaining*), поскольку конфликтующие элементы объединяются в отдельные связные списки (см. рис. 14.6).

Как и в случае элементарного последовательного поиска, эти списки можно хранить упорядоченными или оставить неупорядоченными. При этом приходится идти на те же основные компромиссы, что и описанные в разделе 12.3, но для отдельного связывания экономия времени имеет меньшее значение (поскольку списки невелики), а экономия памяти — более важна (поскольку имеется много списков).

Для упрощения кода вставки в упорядоченный список можно было бы использовать начальный узел, но применение  $M$  начальных узлов для отдельных списков при отдельном связывании может быть нежелательно. Действительно, можно было бы даже исключить  $M$  связей на эти списки, объединив первые узлы списков в таблицу (см. упражнение 14.20).

Для случая промаха при поиске можно считать, что хеш-функция достаточно равномерно перемешивает значения ключей, чтобы поиск в каждом из  $M$  списков был равновероятным. Тогда характеристики производительности, рассмотренные в разделе 12.3, применимы к каждому списку.

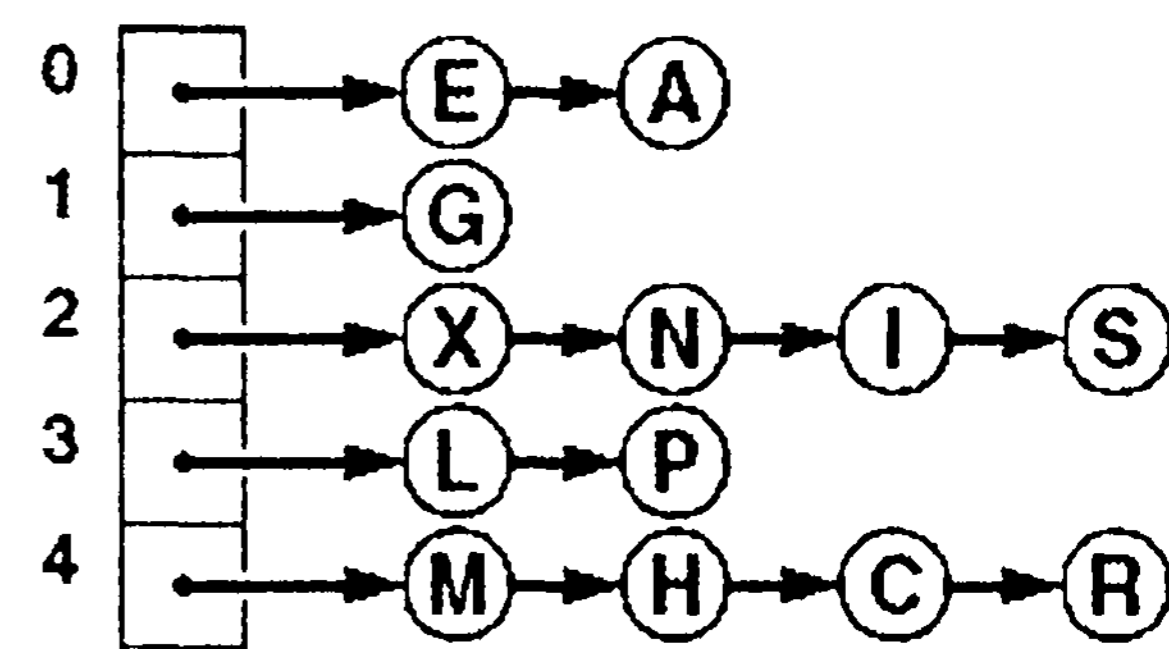
**Лемма 14.1** *Раздельное связывание уменьшает количество выполняемых при последовательном поиске сравнений в  $M$  раз (в среднем) при использовании дополнительного объема памяти для  $M$  связей.*

Средняя длина списков равна  $N/M$ . Как было описано в главе 12, ожидается, что успешные поиски будут доходить приблизительно до середины какого-либо списка. Безрезультатные поиски будут доходить до конца списка, если списки неупорядочены, и до половины списка, если они упорядочены.

Чаще всего для раздельного связывания используются неупорядоченные списки, поскольку этот подход прост в реализации и эффективен: для выполнения операции *insert* требуется постоянное время, а для выполнения операции *search* — время, пропорциональное  $N/M$ . Если ожидается очень большое количество промахов при поиске, обнаружение промахов можно ускорить в два раза, храня списки в упорядоченном виде, ценой замедления операции *insert*.

В приведенном виде лемма 14.1 тривиальна, поскольку средняя длина списков равна  $N/M$  независимо от распределения элементов по спискам. Например, предположим, что все элементы попадают в первый список. Тогда средняя длина списков равна  $(N + 0 + 0 + \dots + 0) / M = N / M$ . Истинная причина практической полезности хеширования заключается в том, что вероятность наличия в *каждом* списке около  $N/M$  элементов очень высока.

A S E R C H I N G X M P L  
0 2 0 4 4 4 2 2 1 2 4 3 3



**РИСУНОК 14.6** ХЕШИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ РАЗДЕЛЬНОГО СВЯЗЫВАНИЯ

На диаграмме показан результат вставки ключей  $A S E R C H I N G X M P L$  в первоначально пустую хеш-таблицу с помощью раздельного связывания (неупорядоченных списков) с использованием хеш-значений, приведенных на верхнем рисунке.  $A$  помещается в список 0, затем  $S$  помещается в список 2,  $E$  — в список 0 (в его начало, с целью поддержания постоянства времени вставки),  $R$  — в список 4 и т.д.

**Лемма 14.2** В хеш-таблице, использующей раздельное связывание, содержащей  $M$  списков и  $N$  ключей, вероятность того, что количество ключей в каждом списке незначительно отличается от  $N/M$ , очень близка к 1.

Для читателей, которые знакомы с основами вероятностного анализа, мы приводим краткое изложение этих классических рассуждений. В соответствии с элементарным доказательством, вероятность, что данный список будет содержать  $k$  элементов, равна

$$\binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}$$

$k$  выбирается из  $N$  элементов: эти  $k$  элементов помещаются в данный список с вероятностью  $1/M$ , а остальные  $N - k$  элементов не помещаются в данный список с вероятностью  $1 - (1/M)$ . Обозначив  $\alpha = N/M$ , это выражение можно переписать как

$$\binom{N}{k} \left(\frac{\alpha}{N}\right)^k \left(1 - \frac{\alpha}{N}\right)^{N-k},$$

которое, в соответствии с классической аппроксимацией Пуассона (Poisson), меньше чем

$$\frac{\alpha^k e^{-\alpha}}{k!}.$$

Отсюда следует, что вероятность наличия в списке более чем  $t\alpha$  элементов меньше чем

$$\left(\frac{\alpha e}{t}\right)^t e^{-\alpha}.$$

Эта вероятность исключительно мала для используемых на практике диапазонов параметров. Например, если средняя длина списков равна 20, вероятность хеширования в какой-либо список, содержащий более 40 элементов, меньше чем  $(40e/2)^2 e^{-20} \approx 0.0000016$ .

Приведенный анализ — пример классической задачи занятости, при которой рассматривается  $N$  мячей, произвольно забрасываемых в одну из  $M$  корзин, и анализируется распределение мячей по корзинам. Классический математический анализ этих задач предоставляет много других интересных фактов, имеющих отношение к изучению алгоритмов хеширования. Например, в соответствии с аппроксимацией Пуассона количество пустых списков близко к  $e^{-\alpha}$ . Еще интереснее, что среднее количество элементов, вставленных прежде, чем происходит первое совпадение, равно приблизительно  $\sqrt{\pi M/2} \approx 1.25 \sqrt{M}$ . Этот результат — решение классической задачи о дне рождения. Например, в соответствии с этими же рассуждениями, при  $M = 365$  среднее количество людей, которых придется проверить, прежде чем удастся найти двух с оди-

наковыми датами рождения, приблизительно равно 24. В соответствии со вторым классическим результатом среднее количество элементов, вставленных прежде, чем в каждом списке окажется, по меньшей мере, по одному элементу, приблизительно равно  $MN_M$ . Этот результат — решение классической задачи *коллекционера карточек*. Например, в соответствии с этим анализом при  $M = 1280$ , вероятно, придется собрать 9898 бейсбольных карточек (купонов), прежде чем удастся заполучить по одной карточке для каждого из 40 игроков каждой из 32 команд в серии. Полученные результаты весьма показательны для проанализированных свойств хеширования. На практике, в соответствии с ними, раздельное связывание можно успешно использовать, если хеш-функция создает значения, близкие к случайным (см. раздел ссылок).

Как правило, в реализациях раздельного связывания значение  $M$  выбирают достаточно малым, чтобы не приходилось напрасно расходовать огромные непрерывные области памяти с пустыми связями, но достаточно большим, чтобы последовательный поиск был наиболее эффективным методом для списков. Гибридные методы (такие, как использование бинарных деревьев вместо связных списков), вероятно, не стоят беспокойства. В качестве руководящего принципа, значение  $M$  можно выбирать равным приблизительно одной пятой или одной десятой количества ключей, ожидаемого в таблице, чтобы каждый из списков предположительно в среднем содержал около пяти или десяти ключей. Одно из достоинств раздельного связывания в том, что это решение не критично: при наличии большего количества ключей, чем ожидалось, поиски будут занимать несколько больше времени, чем если бы заранее был выбран больший размер таблицы; при наличии в таблице меньшего количества ключей поиск будет выполняться еще быстрее при, вероятно, небольшом объеме напрасно расходуемой памяти. Когда объем памяти не является критичным, значение  $M$  может быть выбрано достаточно большим, чтобы время поиска было постоянным; когда же объем памяти *критичен*, все же можно повысить производительность в  $M$  раз, выбрав значение  $M$  максимально допустимым.

Приведенные в предыдущем абзаце комментарии применимы ко времени поиска. На практике для раздельного связывания обычно применяются неупорядоченные списки по двум основным причинам. Во-первых, как уже упоминалось, операция *insert* выполняется исключительно быстро: мы вычисляем хеш-функцию, выделяем память для узла и связываем узел с началом соответствующего списка. Во многих приложениях шаг распределения памяти не требуется (поскольку элементами, вставленными в таблицу символов, могут быть существующие записи с доступными полями связей), и для выполнения операции *insert* остается выполнить всего три или четыре машинные инструкции. Второе важное преимущество использования в программе 14.3 реализации с использованием неупорядоченных списков в том, что все списки работают подобно стекам, поэтому можно легко удалить последние вставленные элементы, которые размещаются в начале списков (см. упражнение 14.21). Эта операция важна при реализации таблицы символов со вложенными диапазонами, например в компиляторе.

Как и в нескольких предшествующих реализациях, мы неявно предоставляем клиенту выбор способа обработки дублированных ключей. Клиент, подобный программе 12.11, может выполнить операцию *search* для проверки наличия дубликатов перед выполнением любой операции *insert*, убеждаясь, что таблица не содержит дублирован-

ных ключей. Другой клиент может избежать сопряженных с этой операцией *search* затрат, оставив дубликаты в таблице, тем самым ускоряя выполнение операций *insert*.

В общем случае хеширование не подходит для использования в приложениях, в которых требуются реализации операций *sort* и *select* для АД. Однако хеширование часто используется в типовых ситуациях, когда необходимо использовать таблицу символов с потенциально большим количеством операций *search*, *insert* и *remove* с последующим однократным выводом элементов в порядке их ключей. Одним из примеров такого приложения является таблица символов в компиляторе; другой пример — программа удаления дубликатов, подобная программе 12.11. Для обработки этой ситуации в реализации раздельного связывания с использованием неупорядоченных списков следовало бы воспользоваться одним из методов сортировки, описанных в главах 6—10. В реализации с использованием упорядоченного списка сортировку можно было бы выполнить за время, пропорциональное  $N \lg M$  в случае сортировки слиянием списка (см. упражнение 14.23).

## Упражнения

- ▷ **14.16** Сколько времени могло бы потребоваться в худшем случае для вставки  $N$  ключей в первоначально пустую таблицу при использовании раздельного связывания с (i) неупорядоченными списками и (ii) упорядоченными списками?
- ▷ **14.17** Приведите содержимое хеш-таблицы, образованной при вставке элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустую таблицу, состоящую из  $N = 5$  списков при использовании раздельного связывания с неупорядоченными списками. Для преобразования  $k$ -той буквы алфавита в индекс таблицы используйте хеш-функцию  $11k \bmod M$ .
- ▷ **14.18** Выполните упражнение 14.17, но для случая упорядоченных списков. Зависит ли ответ от порядка вставки элементов?
- **14.19** Создайте программу, которая с использованием раздельного связывания вставляет  $N$  случайных целых чисел в таблицу размером  $N/100$ , а затем определяет длину самого короткого и самого длинного списков, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 14.20** Измените программу 14.3 с целью исключения из нее заглавных связей путем представления таблицы символов в виде массива узлов (**nodes**) (каждая запись таблицы — первый узел в ее списке).
- 14.21** Измените программу 14.3, включив в нее для каждого элемента целочисленное поле, значение которого устанавливается равным количеству элементов в таблице в момент вставки элемента. Затем реализуйте функцию, которая удаляет все элементы, для которого значение этого поля больше заданного целого числа  $N$ .
- 14.22** Измените реализацию функции **search** в программе 14.3, чтобы она отображала все элементы, ключи которых равны данному ключу, так же, как это делает функция **show**.
- 14.23** Разработайте реализацию таблицы символов, использующую раздельное связывание с упорядоченными списками (при фиксированном размере таблицы, равном 97), которая включает деструктор, конструктор копирования и перегруженную операцию присваивания и поддерживает операции *construct*, *count*, *search*, *insert*, *remove*, *join*, *select* и *sort* для АД первого класса таблицы символов при поддержке дескрипторов клиента (см. упражнения 12.6 и 12.7).

## 14.3 Линейное зондирование

Если можно заранее предусмотреть количество элементов, которые должны быть помещены в хеш-таблицу, и при наличии достаточно большой непрерывной области памяти, в которой можно хранить все ключи при некотором остающемся свободном объеме памяти, в хеш-таблице, вероятно, вообще не стоит использовать какие-либо связи. Существует несколько методов хранения  $N$  элементов в таблице размером  $M > N$ , при которых разрешение конфликтов основывается на наличии пустых мест в таблице. Такие методы называются методами хеширования с *открытой адресацией*.

Простейший метод открытой адресации называется *линейным зондированием* (*linear probing*): при наличии конфликта (когда хеширование выполняется в место таблицы, которое уже занято элементом с ключом, не совпадающим с ключом поиска) мы просто проверяем следующую позицию в таблице. Обычно подобную проверку (определяющую, содержит ли данная позиция таблицы элемент с ключом, равным ключу поиска) называют *зондированием* (*probe*).

Линейное зондирование характеризуется выявлением одного из трех возможных исходов зондирования: если позиция таблицы содержит элемент, ключ которого совпадает с искомым, имеет место попадание при поиске; в противном случае (если позиция таблицы содержит элемент, ключ которого не совпадает с искомым) мы просто зондируем позицию таблицы со следующим по величине индексом, продолжая этот процесс (возвращаясь к началу таблицы при достижении ее конца) до тех пор, пока не будет найден искомый ключ или пустая позиция таблицы. Если содержащий искомый ключ элемент должен быть вставлен вслед за неудачным поиском, он помещается в пустую область таблицы, в которой поиск был завершен. Программа 14.4 — это реализация АДТ таблицы символов, где используется этот метод. Процесс построения хеш-таблицы с использованием линейного зондирования для тестового набора ключей показан на рис. 14.7.

### Программа 14.4 Линейное зондирование

Эта реализация таблицы символов хранит элементы в таблице, размер которой вдвое превышает максимально ожидаемое количество элементов и инициализируется значением `nullItem`. Таблица содержит сами элементы; если элементы велики, тип элемента можно изменить, чтобы он содержал ссылки на элементы.

Для вставки нового элемента выполняется хеширование в позицию таблицы и сканирование вправо с целью нахождения незанятой позиции, используя в незанятых позициях нулевые элементы в качестве служебных, как это делалось при поиске с индексированием по ключу (программа 12.4). Для поиска элемента с данным ключом мы обращаемся к позиции ключа в хеш-таблице и выполняем сканирование для отыскания совпадения, завершая процесс после нахождения незанятой позиции.

Конструктор устанавливает  $M$  таким образом, чтобы таблица была заполнена менее чем наполовину, поэтому для выполнения остальных операций потребуется всего несколько зондирований, если хеш-функция создает значения, достаточно близкие к случайным.

```
private:  
    Item *st;  
    int N, M;  
    Item nullItem;
```

```

public:
    ST(int maxN)
    {
        N = 0; M = 2*maxN;
        st = new Item[M];
        for (int i = 0; i < M; i++)
            st[i] = nullItem;
    }
    int count() const { return N; }
    void insert(Item item)
    { int i = hash(item.key(), M);
      while (!st[i].null()) i = (i+1) % M;
      st[i] = item; N++;
    }
    Item search(Key v)
    { int i = hash(v, M);
      while (!st[i].null())
          if (v == st[i].key()) return st[i];
          else i = (i+1) % M;
      return nullItem;
    }
}

```

Как и в случае отдельного связывания, производительность методов с открытой адресацией зависит от коэффициента  $\alpha = N/M$ , но при этом он интерпретируется иначе. В случае отдельного связывания  $\alpha$  — среднее количество элементов в одном списке, которое в общем случае больше 1. В случае открытой адресации  $\alpha$  — доля занятых позиций таблицы в процентах; она должна быть меньше 1. Иногда  $\alpha$  называют коэффициентом загрузки хеш-таблицы.

В случае разреженной (слабо заполненной) таблицы (значение  $\alpha$  мало) можно рассчитывать, что в большинстве случаев поиска пустая позиция будет найдена в результате всего нескольких зондирований. В случае почти полной таблицы (значение  $\alpha$  близко к 1) для выполнения поиска могло бы потребоваться очень большое количество зондирований, а когда таблица полностью заполнена, поиск может даже привести к бесконечному циклу. Как правило, при использовании линейного зондирования во избежание больших затрат времени при поиске стремятся к тому, чтобы не допустить заполнения таблицы. То есть, вместо того, чтобы использовать дополнительный объем памяти для связей, задействуется дополнительное пространство в хеш-таблице, что позволяет сократить последовательности зондирования. При использовании линейного зондирования размер таблицы больше, чем при отдельном связывании, поэтому необходимо, чтобы  $M > N$ , но общий используемый объем памяти может быть меньше, поскольку никаких связей не используется.

```

A S E R C H I N G X M P
7 3 9 9 8 4 11 7 10 12 0 8

```

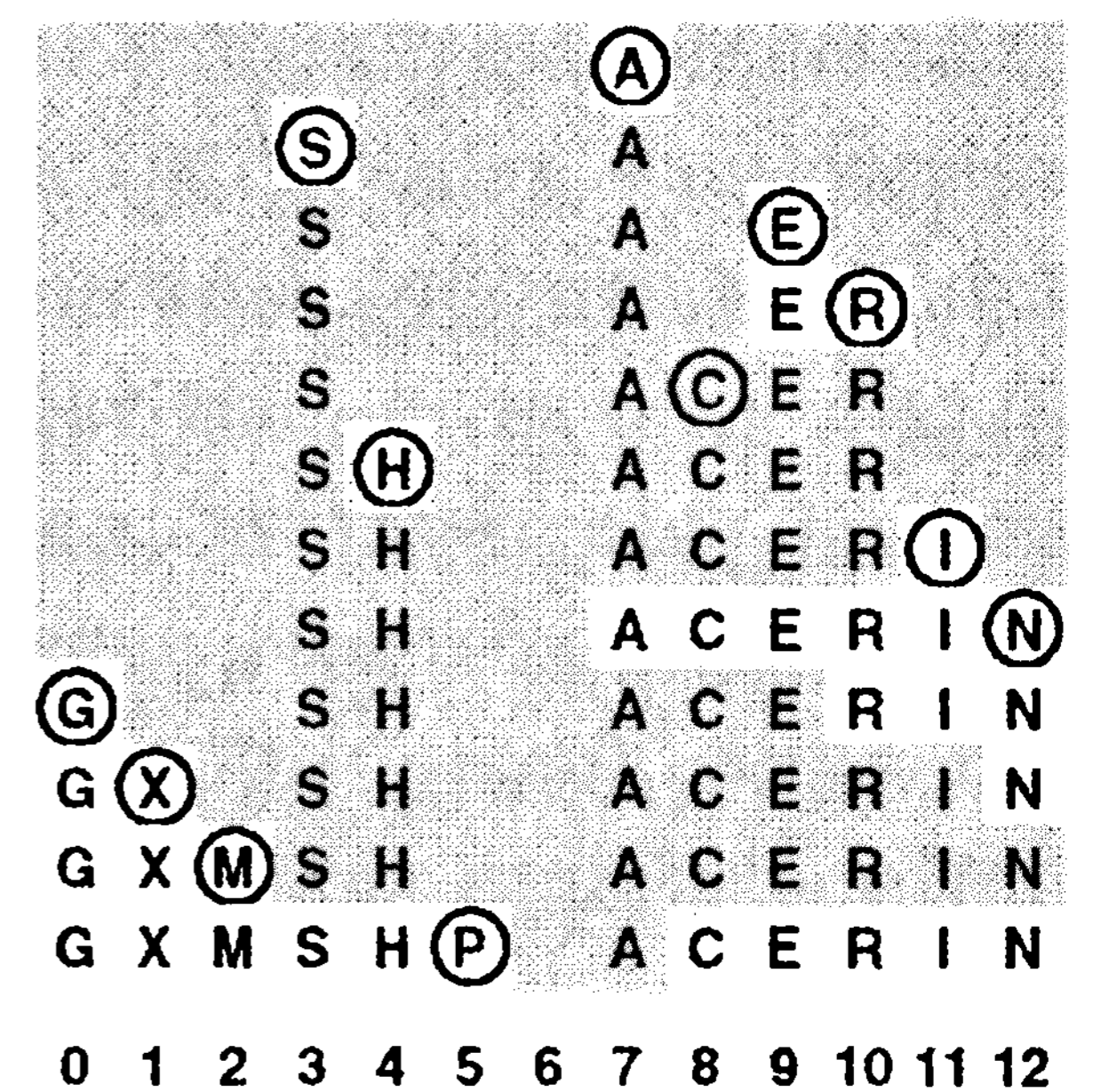


РИСУНОК 14.7 ХЕШИРОВАНИЕ МЕТОДОМ ЛИНЕЙНОГО ЗОНДИРОВАНИЯ

На этой диаграмме показан процесс вставки ключей *A S E R C H I N G X M P* в первоначально пустую хеш-таблицу с открытой адресацией, размер которой равен 13, при использовании показанных сверху хеш-значений и разрешении конфликтов за счет применения линейного зондирования. Вначале *A* помещается в позицию 7, затем *S* помещается в позицию 3, *E* — в позицию 9, далее, после конфликта в позиции 9, *R* помещается в позицию 10 и т.д. При достижении правого конца таблицы зондирование продолжается с левого конца: например, последний вставленный ключ *P* помещается в позицию 8, затем после возникновения конфликта в позициях 8 — 12 и 0 — 5 выполняется зондирование позиции 5. Все позиции таблицы, которые не зондировались, затенены.



Вопросы сравнения используемого объема памяти подробно рассматриваются в разделе 14.5. Пока же давайте проанализируем время выполнения линейного зондирования как функцию  $\alpha$ .

Средние затраты на выполнение линейного зондирования зависят от способа объединения элементов в непрерывные группы занятых ячеек таблицы, называемые *кластерами (clusters)*, при их вставке. Давайте рассмотрим следующие два крайних случая заполненной наполовину ( $M = 2N$ ) таблицы линейного зондирования: в лучшем случае позиции таблицы с четными индексами могли бы быть пустыми, а с нечетными — занятыми. В худшем случае первая половина позиций таблицы могла бы быть пустой, а вторая — заполненной. Средняя длина кластеров в обоих случаях равна  $N/(2N) = 1/2$ , но среднее количество зондирований для безуспешного поиска равно 1 (все поиски требуют, по меньшей мере, одного зондирования) плюс

$$(0 + 1 + 0 + 1 + \dots) / (2N) = 1 / 2$$

в лучшем случае и 1 плюс

$$(N + (N - 1) + (N - 2) + \dots) / (2N) \approx N/4$$

в худшем случае.

Обобщая приведенные рассуждения, приходим к выводу, что среднее количество зондирований для безуспешного поиска пропорционально *квадратам* длин кластеров. Среднее значение вычисляется путем вычисления затрат для промаха при поиске, начиная с каждой позиции таблицы, и деля сумму на  $M$ . Для обнаружения всех промахов при поиске требуется не менее 1 зондирования, поэтому выполняется подсчет количества зондирований, следующих за первым. Если кластер имеет длину  $t$ , то выражение

$$(t + (t - 1) + \dots + 2 + 1) / M = t(t + 1) / (2M)$$

определяет вклад этого кластера в общую сумму. Сумма длин кластеров равна  $N$ , поэтому суммируя эти затраты для всех ячеек в таблице, находим, что общие средние затраты на обнаружение промаха при поиске равны  $1 + N/(2M)$  плюс сумма квадратов длин кластеров, деленная на  $2M$ . Имея заданную таблицу, можно быстро вычислить средние затраты на безуспешный поиск в этой таблице (см. упражнение 14.28), но кластеры образуются в результате сложного динамического процесса (алгоритма линейного зондирования), который трудно охарактеризовать аналитически.

**Лемма 14.3** При разрешении конфликтов с помощью линейного зондирования среднее количество зондирований, требуемых для поиска в хеш-таблице размером  $M$ , которая содержит  $N = \alpha M$  ключей, приблизительно равно

$$\frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \text{и} \quad \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right)$$

для попаданий и промахов, соответственно.

Несмотря на сравнительно простую форму полученных результатов, точный анализ линейного зондирования — сложная задача. Вывод, полученный Кнутом (Knuth) в 1962 г., явился значительной вехой в анализе алгоритмов (см. раздел ссылок).

Точность этих выражений уменьшается с приближением значения  $\alpha$  к 1, но в данном случае это не важно, поскольку в любом случае линейное зондирование не следует использовать в почти заполненной таблице. Для меньших значений  $\alpha$  уравнения достаточно точны. Ниже приведена таблица обобщенных значений ожидаемого количества зондирований, требуемых для обнаружения попаданий и промахов при поиске с использованием линейного зондирования:

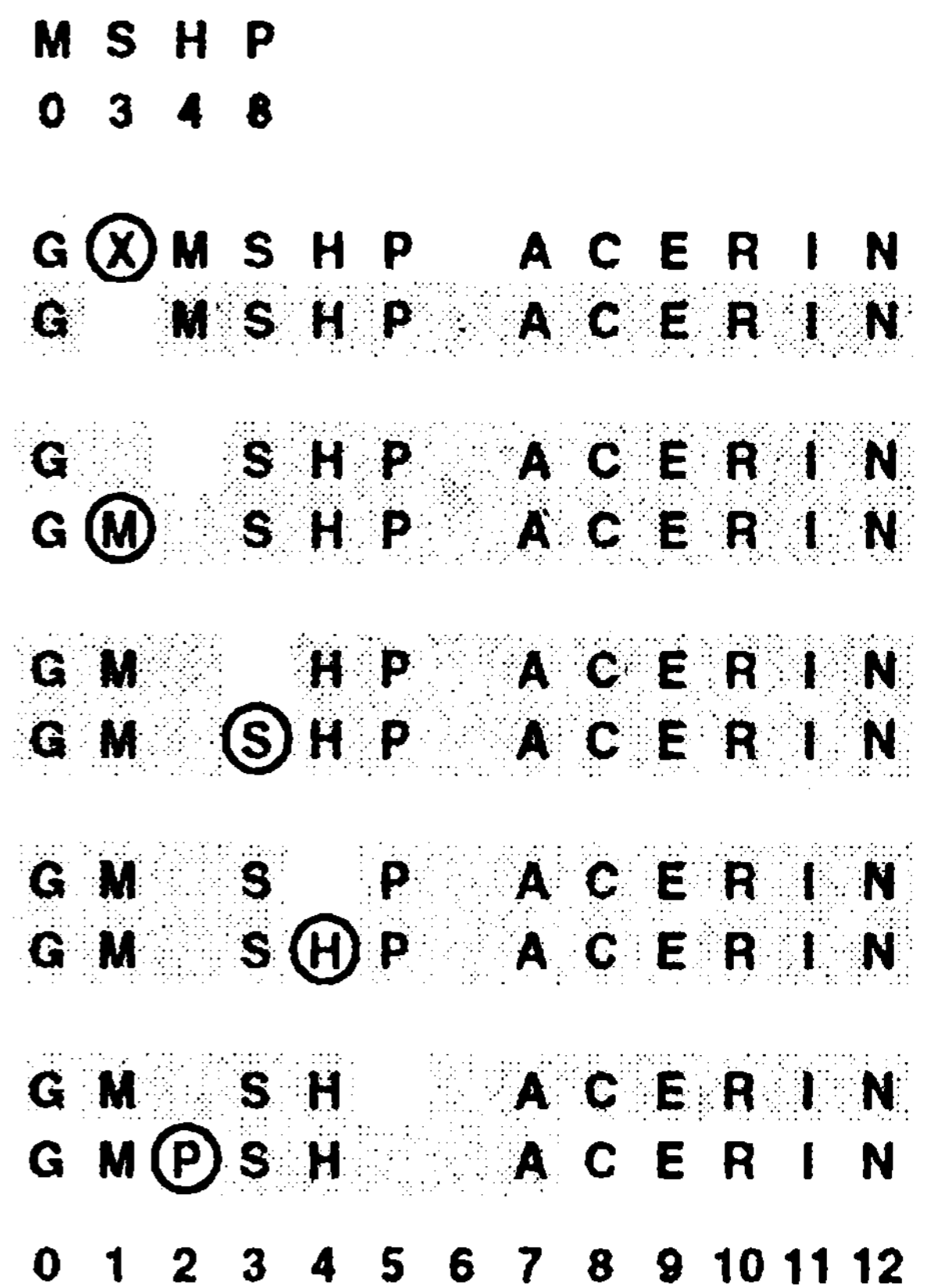
коэффициент загрузки ( $\alpha$ )	1/2	2/3	3/4	9/10
попадание при поиске	1.5	2.0	3.0	5.5
промах при поиске	2.5	5.0	8.5	55.5

Для обнаружения промахов при поиске всегда требуются большие затраты, чем для обнаружения попаданий, и в обоих случаях в таблице, которая заполнена менее чем на половину, в среднем требуется лишь несколько зондирований.

Подобно тому, как это делалось при использовании отдельного связывания, мы предоставляем клиенту право выбора, хранить ли в таблице элементы с дублированными ключами. Такие элементы не обязательно размещаются в таблице линейного зондирования в соседних позициях — среди элементов с дублированными ключами могут размещаться и другие элементы с таким же хеш-значением.

Исходя из самого способа образования таблицы, ключи в таблице, построенной линейным зондированием, размещаются в случайном порядке. В результате операции *sort* и *select* абстрактного типа данных требуют реализации с самого начала по одному из методов, описанных в главах 6—10. Поэтому линейное зондирование не подходит для приложений, в которых эти операции выполняются часто.

А как удалить ключ из таблицы, построенной с помощью линейного зондирования? Его нельзя просто удалить, поскольку элементы, которые были вставлены позже, могут вызывать пропуск этого элемента и поэтому поиск таких элементов мог бы постоянно прерываться на пустой позиции, оставленной удаленным элементом. Одно из решений этой проблемы заключается в повторном хешировании всех элементов, для которых эта проблема могла бы возникнуть — между удаленным элементом и следующей незанятой позицией справа от него. Пример, иллюстрирующий этот процесс, приведен на рис. 14.8. Программа 14.5 содержит реализацию этого подхода. В разреженной таблице в большинстве слу-



**РИСУНОК 14.8 УДАЛЕНИЕ В ХЕШ-ТАБЛИЦЕ, ИСПОЛЬЗУЮЩЕЙ ЛИНЕЙНОЕ ЗОНДИРОВАНИЕ**

На этой диаграмме демонстрируется процесс удаления  $X$  из таблицы, показанной на рис. 14.7. Во второй строке показан результат простого удаления  $X$  из таблицы, что неприемлемо, поскольку  $M$  и  $P$  отрезаются от своих хеш-позиций пустой позицией, оставленной ключом  $X$ . Поэтому ключи  $M$ ,  $S$ ,  $H$  и  $P$  (расположенные справа от  $X$  в этом же кластере) повторно вставляются в указанном порядке с использованием хеш-значений, указанных вверху, и с разрешением конфликтов с помощью линейного зондирования.  $M$  заполняет свободное место, оставленное ключом  $X$ , затем  $S$  и  $H$  вставляются в таблицу, не вызывая конфликтов, а затем  $P$  вставляется в позицию 2.

чаев процесс потребует лишь нескольких операций повторного хеширования. Другой способ реализации удаления — замена удаленного ключа служебным ключом, который может служить заполнителем для поиска, но может быть определен и повторно использоваться для вставок (см. упражнение 14.33).

### Программа 14.5 Удаление из хеш-таблицы, использующей линейное зондирование

Для удаления элемента с заданным ключом мы выполняем его поиск и заменяем его элементом `nullItem`. Затем необходимо внести изменения на случай, если какой-либо элемент, расположенный справа от теперь незанятой позиции, помещается в эту позицию или левее нее, поскольку свободная позиция приводила бы к прерыванию поиска такого элемента. Поэтому выполняется повторная вставка всех элементов, которые располагаются в одном кластере с удаленным элементом и правее него. Поскольку таблица заполнена менее чем на половину, в среднем количество повторно вставляемых элементов будет мало.

```
void remove(Item x)
{ int i = hash(x.key(), M), j;
  while (!st[i].null())
    if (x.key() == st[i].key()) break;
    else i = (i+1) % M;
  if (st[i].null()) return;
  st[i] = nullItem; N--;
  for (j = i+1; !st[j].null(); j = (j+1) % M, N--)
    { Item v = st[j]; st[j] = nullItem; insert(v); }
}
```

## Упражнения

- ▷ **14.24** Какое время может потребоваться в худшем случае для вставки  $N$  ключей в первоначально пустую таблицу при использовании линейного зондирования?
- ▷ **14.25** Приведите содержимое хеш-таблицы, образованной в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустую таблицу размером  $M = 16$  при использовании линейного зондирования. Используйте хеш-функцию  $11k \bmod M$  для преобразования  $k$ -той буквы алфавита в индекс таблицы.

**14.26** Выполните упражнение **14.25** для  $M = 10$ .

- **14.27** Создайте программу, которая вставляет  $10^5$  случайных неотрицательных чисел, меньших чем  $10^6$ , в таблицу размером  $10^5$ , использующую линейное зондирование. Программа должна в графическом виде выводить количество зондирований, используемых для каждой из  $10^3$  последовательных вставок.

**14.28** Создайте программу, которая вставляет  $N/2$  случайных целых чисел в таблицу размером  $N$ , использующую линейное зондирование, а затем на основании длин кластеров вычисляет средние затраты на обнаружение промаха при поиске в результирующей таблице, для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

**14.29** Создайте программу, которая вставляет  $N/2$  случайных целых чисел в таблицу размером  $N$ , использующую линейное зондирование, а затем вычисляет средние затраты на обнаружение попадания при поиске в результирующей таблице, для  $N = 10^3, 10^4, 10^5$  и  $10^6$ . В конце *не* выполняйте поиск всех ключей (отслеживайте затраты на построение таблицы).

- **14.30** Определите экспериментальным путем, изменяются ли средние затраты на обнаружение попаданий и промахов при поиске в случае выполнения длинной последовательности чередующихся случайных вставок и удалений с помощью программ 14.4 и 14.5 в хеш-таблице размером  $2N$ , содержащей  $N$  ключей, для  $N = 10, 100$  и  $1000$  и до  $N^2$  пар удалений-вставок для каждого значения  $N$ .

## 14.4 Двойное хеширование

Основной принцип линейного зондирования (а, в действительности, и любого метода хеширования) — обеспечение гарантии того, что при поиске конкретного ключа выполняется поиск каждого ключа, который отображается тем же адресом в таблице (в частности, самого ключа, если он присутствует в таблице). Однако, как правило, при использовании схемы с открытой адресацией другие ключи также исследуются, особенно, когда таблица начинает приближаться к заполненному состоянию. В примере, приведенном на рис. 14.7, поиск ключа **N** сопряжен с просмотром ключей **C**, **E**, **R** и **I**, ни один из которых не имеет такого же хеш-значения. Хуже того, вставка ключа с одним хеш-значением может существенно увеличить время поиска ключей с другими хеш-значениями: на рис. 14.7 вставка ключа **M** приводит к увеличению времени поиска для позиций 1—12 и 0—1. Это явление называется *кластеризацией*, поскольку оно связано с процессом образования кластеров. Оно может значительно замедлять линейное зондирование для почти заполненных таблиц.

К счастью, существует простой способ избежать возникновения проблемы кластеризации — *двойное хеширование*. Основная стратегия остается той же, что и при выполнении линейного зондирования. Единственное различие состоит в том, что вместо исследования каждой позиции таблицы, следующей за конфликтной, мы используем вторую хеш-функцию для получения постоянного шага, который будет использоваться для последовательности зондирования. Реализация метода приведена в программе 14.6.

### Программа 14.6 Двойное хеширование

Двойное хеширование аналогично линейному зондированию, за исключением того, что вторая хеш-функция используется для определения шага поиска, который будет использоваться после обнаружения каждого конфликта. Шаг поиска должен быть ненулевым, а размер таблицы и шаг поиска должны быть взаимно простыми числами. Функция **remove** для линейного зондирования (см. программу 14.5) не работает с двойным хешированием, поскольку любой ключ может присутствовать во множестве различных последовательностей зондирования.

```
void insert(Item item)
{ Key v = item.key();
  int i = hash(v, M), k = hashtwo(v, M);
  while (!st[i].null()) i = (i+k) % M;
  st[i] = item; N++;
}
Item search(Key v)
{ int i = hash(v, M), k = hashtwo(v, M);
  while (!st[i].null())
    if (v == st[i].key()) return st[i];
    else i = (i+k) % M;
  return nullItem;
}
```

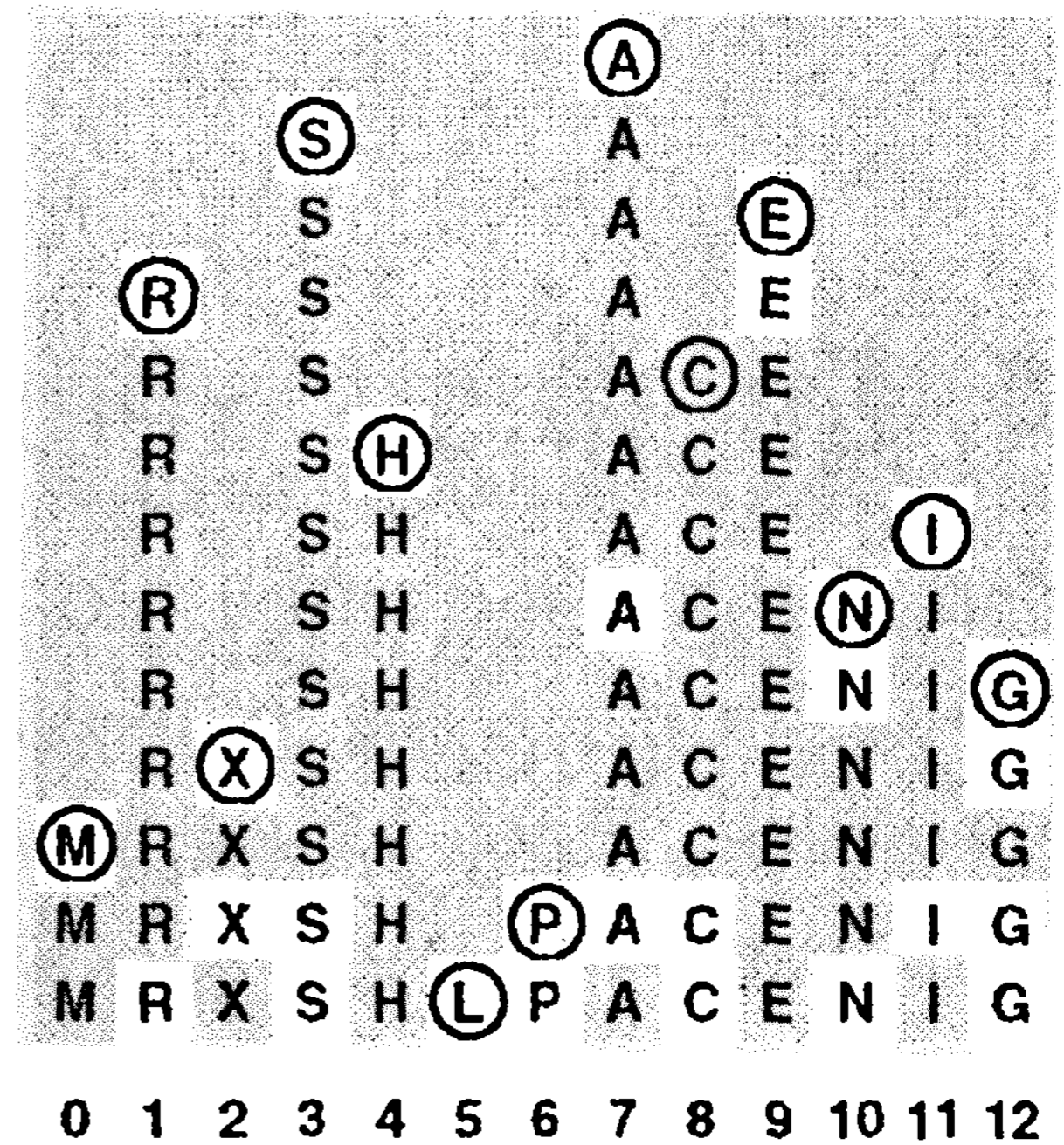
Выбор второй хеш-функции требует определенной осторожности, поскольку в противном случае программа может вообще не работать. Во-первых, необходимо исключить случай, когда вторая хеш-функция дает значение, равное 0, поскольку это приводило бы к бесконечному циклу при первом же конфликте. Во-вторых, важно, чтобы значение второй хеш-функции и размер таблицы были взаимно простыми числами, поскольку в противном случае некоторые из последовательностей зондирования могли бы оказаться очень короткими (рассмотрите для примера случай, когда размер таблицы вдвое больше значения второй хеш-функции). Один из способов претворения подобной политики в жизнь — выбор в качестве  $M$  простого значения и выбор второй хеш-функции, возвращающей значения, которые меньше  $M$ . На практике простой второй хеш-функции наподобие

```
inline int hashtwo(Key v)
{ return (v % 97) + 1; }
```

будет достаточно для многих хеш-функций, когда размер таблицы не слишком мал. Кроме того, любое снижение эффективности, вызываемое данным упрощением, скорее всего, будет мало заметно на практике. Если таблица очень велика или мало заполнена, сам размер таблицы не обязательно должен быть простым числом, поскольку для каждого поиска будет использоваться лишь несколько зондирования (хотя, при использовании этого упрощения, для предотвращения бесконечного цикла может потребоваться выполнение проверки для прерывания длинных поисков (см. упражнение 14.38)).

На рис. 14.9 показан процесс построения небольшой таблицы методом двойного хеширования; из рис. 14.10 видно, что двойное хеширование приводит к образованию значительно меньшего количества кластеров (которые вследствие этого значительно короче), чем в результате линейного зондирования.

```
A S E R C H I N G X M P L
7 3 9 9 8 4 11 7 10 12 0 8 6
1 3 1 5 5 5 3 3 2 3 5 4 2
```



**РИСУНОК 14.9 ДВОЙНОЕ ХЕШИРОВАНИЕ**

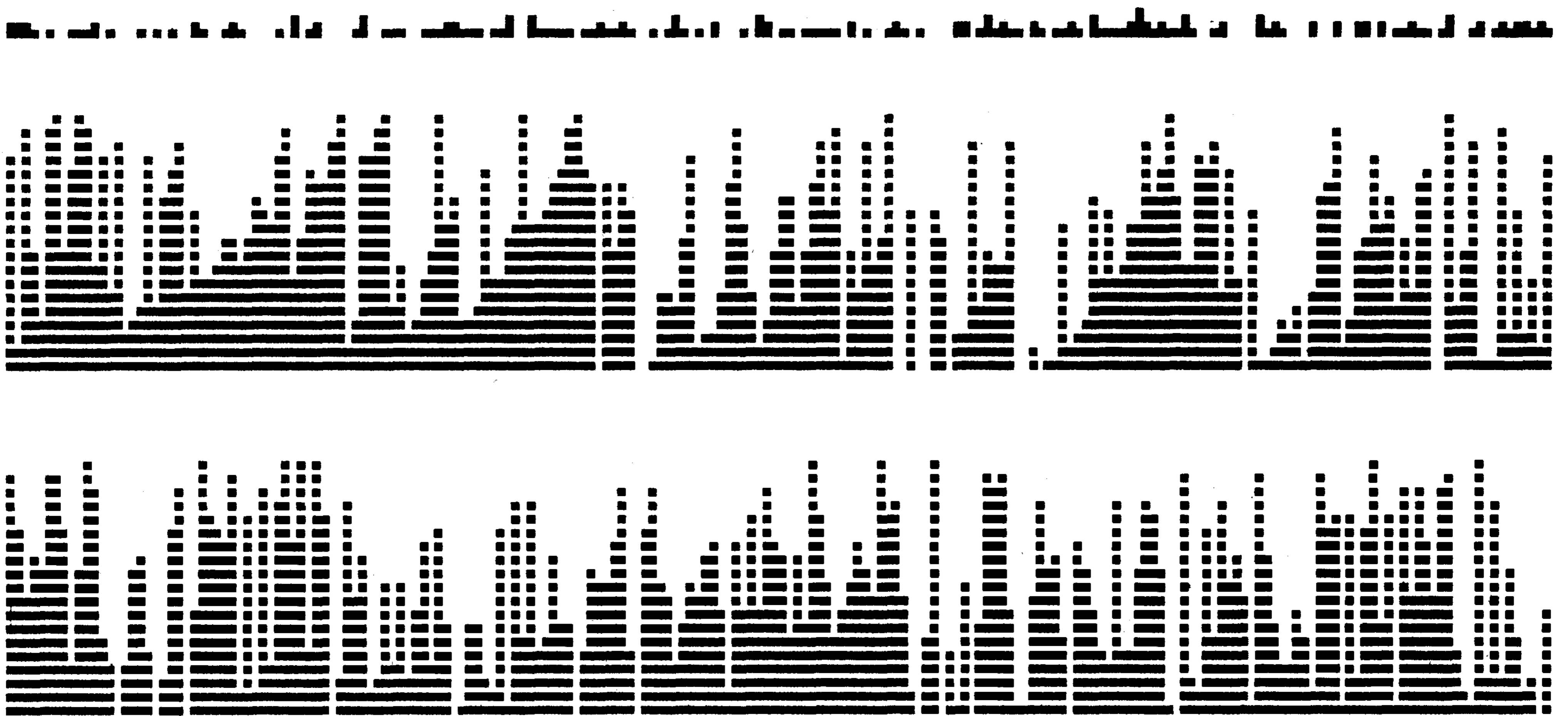
На этой диаграмме показан процесс вставки ключей *A S E R C H I N G X M P L* в первоначально пустую хеш-таблицу с открытой адресацией с использованием хеш-значений, приведенных сверху, и разрешением конфликтов за счет применения двойного хеширования. Первое и второе хеш-значения каждого ключа отображаются в двух строках под этим ключом. Как и на рис. 14.7, зондируемые позиции таблицы не затенены. *A* помещается в позицию 7, затем *S* — в позицию 3, *E* — в позицию 9, как и на рис. 14.7, но ключ *R* помещается в позицию 1 после возникновения конфликта в позиции 9, причем второе хеш-значение этого ключа, равное 5, используется в качестве шага зондирования после возникновения конфликта. Аналогично, ключ *P* помещается в позицию 6 при заключительной вставке после возникновения конфликтов в позициях 8, 12, 3, 7, 11 и 2 при использовании его второго хеш-значения, равного 4, как шага зондирования.

**Лемма 14.4** При разрешении конфликтов с помощью двойного хеширования среднее количество зондирований, необходимых для выполнения поиска в хеш-таблице размером  $M$ , содержащей  $N = \alpha M$  ключей, равно

$$\frac{1}{\alpha} \ln \left( \frac{1}{1-\alpha} \right) \quad \text{и} \quad \frac{1}{1-\alpha}$$

для обнаружения попаданий и промахов, соответственно.

Эти формулы — результат глубокого математического анализа, выполненного Гюйба (Guibas) и Шемереди (Szemerédi) (см. раздел ссылок). Доказательство основывается на том, что двойное хеширование почти эквивалентно более сложному алгоритму случайного хеширования, при котором используется зависящая от ключей последовательность позиций зондирования, обеспечивающая равную вероятность попадания каждого зондирования в каждую позицию таблицы. По многим причинам этот алгоритм — всего лишь аппроксимация двойного хеширования: например, очень трудно гарантировать, чтобы при двойном хешировании каждая позиция таблицы проверялась хотя бы один раз, но при случайном хешировании одна и та же позиция таблицы может проверяться более одного раза. Тем не менее, для разреженных таблиц вероятность возникновения конфликтов при использовании обоих методов одинакова. Интерес представляют оба метода: двойное хеширование легко реализовать, в то время как случайное хеширование легко анализировать.



**РИСУНОК 14.10** КЛАСТЕРИЗАЦИЯ

На этих диаграммах показано размещение записей при их вставке в хеш-таблицу с использованием линейного зондирования (рисунок в центре) и двойного хеширования (рисунок внизу), при распределении ключей, показанном на верхней диаграмме. Каждая строка — результат вставки 10 записей. По мере заполнения таблицы записи объединяются в кластеры, разделенные пустыми позициями таблицы. Длинные кластеры нежелательны, поскольку средние затраты на поиски одного ключа в кластере пропорциональны длине кластера. При использовании линейного зондирования чем длиннее кластеры, тем более вероятно увеличение их длины, поэтому по мере заполнения таблицы несколько длинных кластеров оказываются доминирующими. При использовании двойного хеширования этот эффект значительно менее выражен и кластеры остаются сравнительно короткими.

Средние затраты на обнаружение промаха при поиске для случайного хеширования определяются равенством

$$1 + \frac{N}{M} + \left(\frac{N}{M}\right)^2 + \left(\frac{N}{M}\right)^3 + \dots = \frac{1}{1 - (N/M)} = \frac{1}{1 - \alpha}.$$

Выражение слева — сумма вероятностей использования для обнаружения промахов более  $k$  зондирований, при  $k = 0, 1, 2, \dots$  (которая на основании элементарной теории вероятностей равна средней вероятности). При поиске всегда используется одно зондирование, затем с вероятностью  $(N/M)^2$  требуется второе зондирование и т.д. Эту же формулу можно использовать для вычисления следующего приближенного значения средней стоимости попадания при поиске в таблице, содержащей  $N$  ключей:

$$\frac{1}{N} \left( 1 + \frac{1}{1 - (1/M)} + \frac{1}{1 - (2/M)} + \dots + \frac{1}{1 - ((N-1)/M)} \right).$$

Вероятность попадания одинакова для всех ключей в таблице; затраты на отыскание ключа равны затратам на его вставку, а затраты на вставку  $j$ -го ключа в таблицу равны затратам на обнаружение промаха в таблице, содержащей  $j-1$  ключ, следовательно, эта формула определяет среднее значение этих затрат. Теперь можно упростить и вычислить эту сумму, умножив числители и знаменатели всех дробей на  $M$ :

$$\frac{1}{N} \left( 1 + \frac{M}{M-1} + \frac{M}{M-2} + \dots + \frac{M}{M-N+1} \right),$$

и выполнив дальнейшее упрощение, получаем

$$\frac{M}{N} (H_M - H_{M-N}) \approx \frac{1}{\alpha} \ln \left( \frac{1}{1 - \alpha} \right),$$

поскольку  $H_M \approx \ln M$ .

Точная взаимосвязь между производительностью двойного хеширования и идеальным случаем случайного хеширования, которая была установлена Гюйба и Шемереди — асимптотический результат, который не обязательно должен быть справедлив для используемых на практике размеров таблиц; более того, полученные результаты основываются на предположении, что хеш-функции возвращают случайные значения. Тем не менее, асимптотические формулы, приведенные в лемме 14.5, на практике достаточно точно определяют производительность двойного хеширования, даже при использовании такой просто вычисляемой второй хеш-функции, как  $(v \% 97) + 1$ . Как и в случае соответствующих формул для линейного зондирования, эти формулы стремятся к бесконечности при приближении значения  $\alpha$  к 1, но это происходит значительно медленнее.

Различие между линейным зондированием и двойным хешированием наглядно иллюстрируется на рис. 14.11. Двойное хеширование и линейное зондирование имеют одинаковую производительность для разреженных таблиц, но при использовании

двойного хеширования можно допустить значительно большую степень заполнения таблицы, чем при использовании линейного зондирования, прежде чем производительность заметно снизится. В следующей таблице приведено ожидаемое количество зондирований для обнаружения попаданий и промахов при использовании двойного хеширования:

коэффициент загрузки ( $\alpha$ )	1/2	2/3	3/4	9/10
попадание при поиске	1.4	1.6	1.8	2.6
промах при поиске	1.5	2.0	3.0	5.5

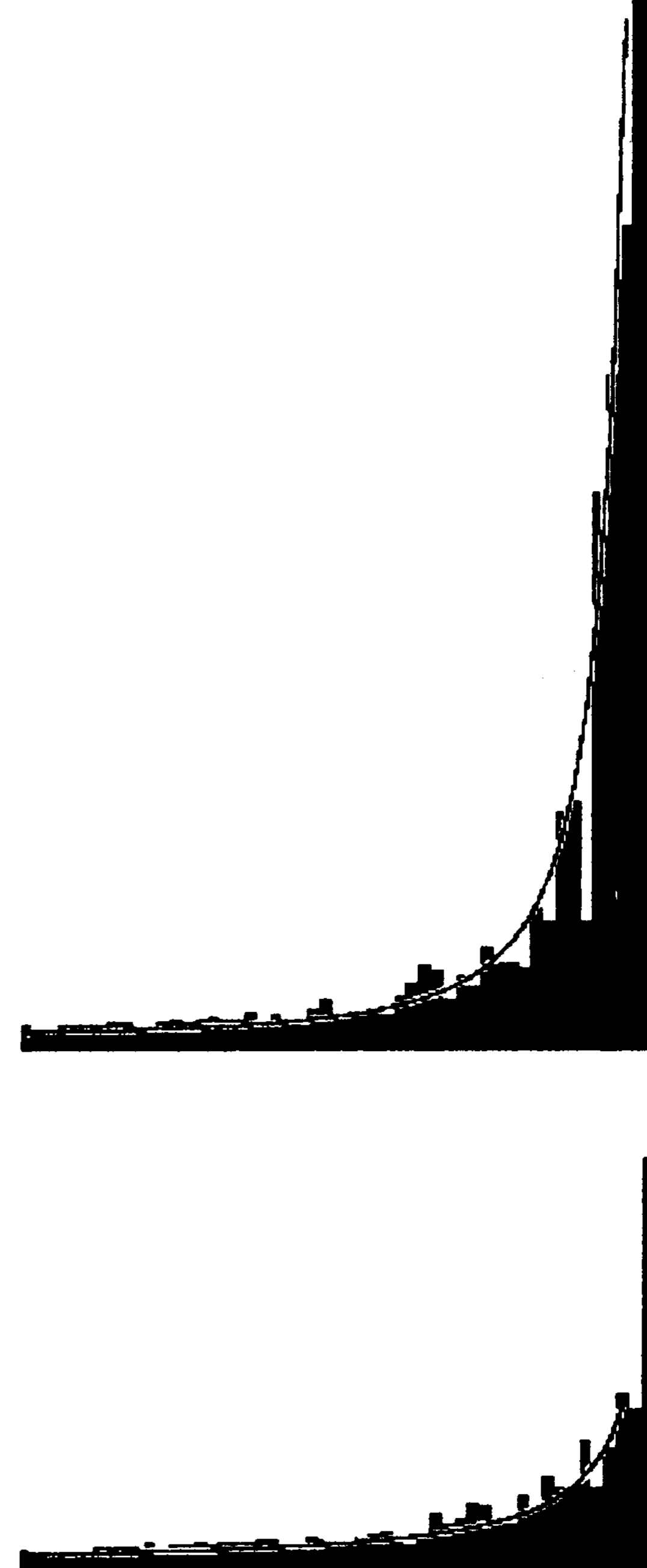
Для обнаружения промахов при поиске всегда требуются большие затраты, чем для обнаружения попаданий, но для обнаружения и тех и других в среднем требуется лишь несколько зондирований даже в таблице, заполненной на девять десятых.

Если взглянуть на эти результаты под другим углом, можно заключить, что для получения такого же среднего времени поиска двойное хеширование позволяет использовать меньшие таблицы, чем потребовалось бы при использовании линейного зондирования.

**Лемма 14.5** *Сохраняя коэффициент загрузки меньшим, чем  $1 - 1/\sqrt{t}$  для линейного зондирования, и меньшим, чем  $1 - 1/t$  для двойного хеширования, можно обеспечить, чтобы в среднем для выполнения всех поисков требовалось менее  $t$  зондирований.*

Установите значения выражений для промахов при поиске равными 1 и решите уравнения относительно  $\alpha$ .

Например, для обеспечения, чтобы среднее количество зондирований для поиска было меньшим 10, необходимо сохранять таблицу пустой по меньшей мере на 32 процента при использовании линейного зондирования, но лишь на 10 процентов при использовании двойного хеширования. При необходимости обработки  $10^5$  элементов, чтобы иметь возможность выполнить безрезультатный поиск менее чем за 10 зондирований, требуется свободное пространство всего для  $10^4$  дополнительных элементов. Для сравнения: при использовании отдельного связывания потребовалась бы дополнительная память для более чем  $10^5$  связей, а при использовании деревьев бинарного поиска потребовался бы вдвое больший объем памяти.



**РИСУНОК 14.11 ЗАТРАТЫ НА ВЫПОЛНЕНИЕ ПОИСКА С ИСПОЛЬЗОВАНИЕМ ОТКРЫТОЙ АДРЕСАЦИИ**

*На этих графиках показаны затраты на построение хеш-таблицы, размер которой равен 1000, путем вставки ключей в первоначально пустую таблицу с использованием линейного зондирования (вверху) и двойного хеширования (внизу). Каждый вертикальный столбец представляет затраты на вставку 20 ключей. Серые кривые представляют теоретически предсказанные затраты (см. леммы 14.4 и 14.5).*



Метод реализации операции *remove*, приведенный в программе 14.5 (повторное хеширование ключей, которые могут иметь путь поиска, содержащий удаляемый элемент), не годится для двойного хеширования, поскольку удаленный ключ может присутствовать во множестве различных последовательностей зондирования, затрагивающих ключи, разбросанные по всей таблице. Поэтому необходимо прибегнуть к другому методу, рассмотренному в конце раздела 12.3: удаленный элемент заменяется служебным элементом, помечающим позицию таблицы как занятую, но не соответствующим ни одному из ключей (см. упражнение 14.33).

Подобно линейному зондированию, двойное хеширование — неподходящее основание для реализации полнофункционального АДТ таблицы символов, в котором необходимо поддерживать операции *sort* или *select*.

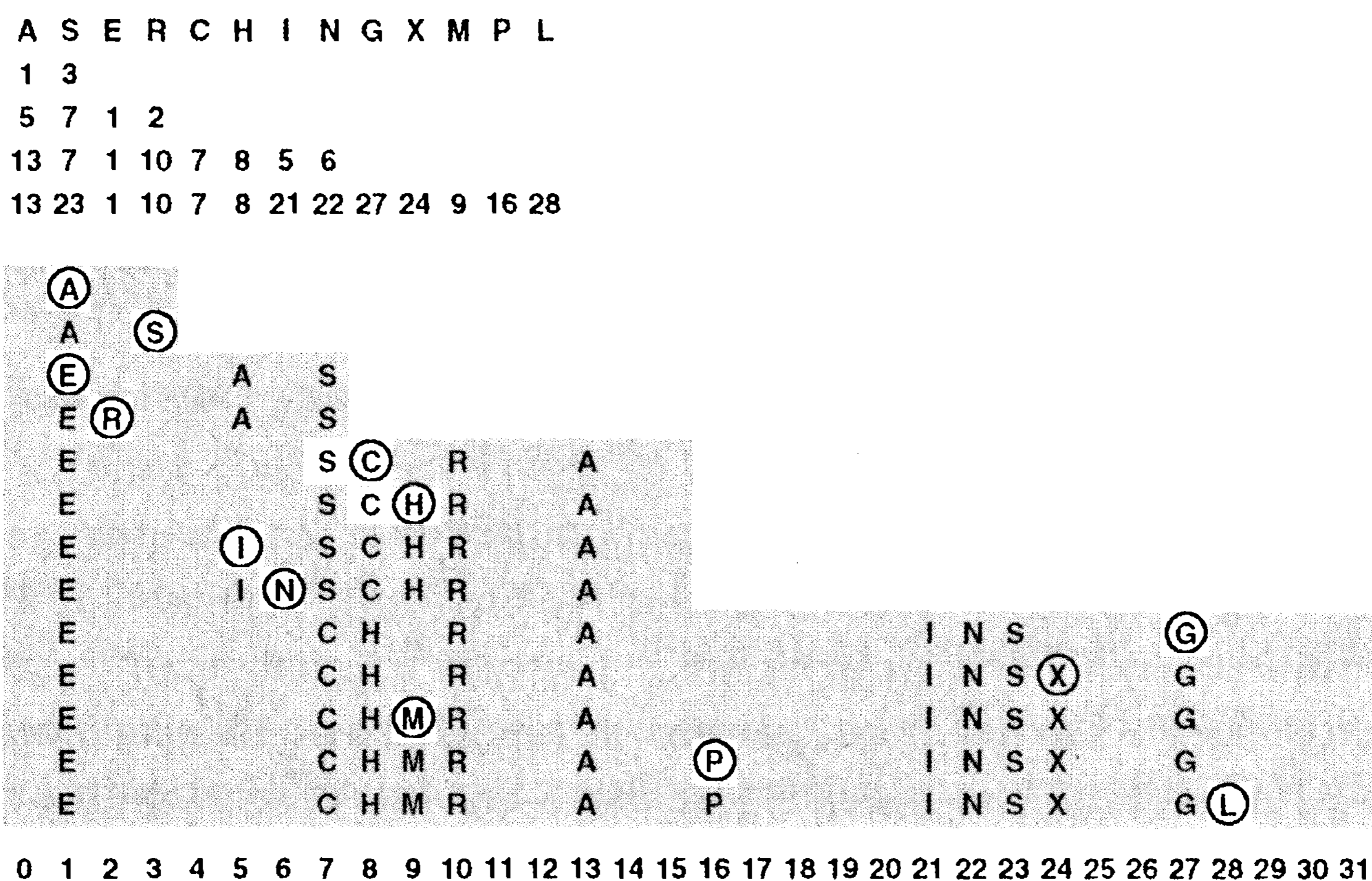
## Упражнения

- ▷ **14.31** Приведите содержимое хеш-таблицы, образованной в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустую таблицу размером  $M = 16$  при использовании двойного хеширования. Воспользуйтесь хеш-функцией  $11k \bmod M$  для первоначального зондирования и второй хеш-функцией  $(k \bmod 3) + 1$  для шага поиска (когда ключ —  $k$ -тая буква алфавита).
- ▷ **14.32** Выполните упражнение 14.31 для  $M = 10$ .
- 14.33** Реализуйте удаление для двойного хеширования с использованием служебного элемента.
- 14.34** Измените решение упражнения 14.27, чтобы в нем использовалось двойное хеширование.
- 14.35** Измените решение упражнения 14.28, чтобы в нем использовалось двойное хеширование.
- 14.36** Измените решение упражнения 14.29, чтобы в нем использовалось двойное хеширование.
- **14.37** Реализуйте алгоритм, который аппроксимирует случайное хеширование, предоставляя ключ в качестве исходного значения для встроенного генератора случайных чисел (как в программе 14.2).
- 14.38** Предположите, что таблица, размер которой равен  $10^6$ , заполнена наполовину, причем занятые позиции распределены случайным образом. Вычислите вероятность того, что все позиции, индексы которых кратны 100, заняты.
- ▷ **14.39** Предположите, что в коде реализации двойного хеширования присутствует ошибка, приводящая к тому, что одна или обе хеш-функции всегда возвращают одно и то же значение (не 0). Опишите, что происходит в каждой из следующих ситуаций: (i) когда первая функция неверна, (ii) когда вторая функция неверна, (iii) когда обе функции неверны.

## 14.5 Динамические хеш-таблицы

С увеличением количества ключей в хеш-таблице производительность поиска уменьшается. При использовании отдельного связывания время поиска увеличивается постепенно — когда количество ключей в таблице удваивается, время поиска удваивается. Это же справедливо по отношению к таким методам с открытой адресацией, как линейное зондирование и двойное хеширование для разреженных таблиц, но по мере заполнения таблицы затраты возрастают коренным образом и, что гораздо хуже, наступает момент, когда никакие дополнительные ключи вообще не могут быть вставлены. Эта ситуация отличается от деревьев поиска, в которых рост происходит естественным образом. Например, в RB-дереве затраты на поиск возрастают лишь несущественно (на одно сравнение) при каждом удвоении количества узлов в дереве.

Один из способов обеспечения роста в хеш-таблицах — удвоение размера таблицы, когда она начинает заполняться. Удвоение размера таблицы — дорогостоящая операция, поскольку все элементы в таблице должны быть вставлены повторно, однако она выполняется нечасто. Программа 14.7 — реализация роста путем удвоения при использовании линейного зондирования. Пример показан на рис. 14.12.



**РИСУНОК 14.12 ДИНАМИЧЕСКОЕ РАСШИРЕНИЕ ХЕШ-ТАБЛИЦЫ**

На этой диаграмме продемонстрирован процесс вставки ключей *A S E R C H I N G X M P L* в динамическую хеш-таблицу, которая расширяется путем удвоения хеш-значений, приведенных в верхней части рисунка, и разрешения конфликтов с помощью линейного зондирования. В четырех строках под ключами приводятся хеш-значения для размера таблицы равного 4, 8, 16 и 32.

Начальный размер таблицы равен 4, затем он удваивается до 8 для *E*, до 16 для *C* и до 32 для *G*. При каждом удвоении размера таблицы для всех ключей выполняется повторное хеширование и повторная вставка. Все вставки выполняются в разреженные таблицы (менее чем на одну четверть для повторной вставки и от одной четверти до одной второй в остальных случаях), поэтому возникает мало конфликтов.

Это же решение работает и для двойного хеширования, а основная идея применима и для отдельного связывания (см. упражнение 14.46). Каждый раз когда таблица заполняется более чем на половину, она расширяется путем удвоения ее размера. После первого расширения доля заполнения таблицы всегда составляет от одной четвертой до одной второй, поэтому в среднем затраты на поиск составляют менее трех зондирований. Более того, хотя операция повторного построения таблицы является дорогостоящей, она выполняется столь редко, что ее стоимость представляет лишь постоянную долю общих затрат на построение таблицы.

#### Программа 14.7 Динамическая вставка в хеш-таблицу (для линейного зондирования)

Эта реализация операции `insert` для линейного зондирования (см. программу 14.4) обрабатывает произвольное количество ключей, удваивая размер таблицы при каждом заполнении таблицы наполовину (такой же подход может использоваться для двойного хеширования или отдельного связывания). Удвоение требует распределения памяти для новой таблицы, повторного хеширования всех ключей в новую таблицу, а затем освобождения памяти, занимаемой старой таблицей. Функция-член `init` используется для построения или повторного построения таблицы, заполненной нулевыми элементами указанных размеров: она реализована так же, как конструктор `ST` в программе 14.4, поэтому ее код опущен.

```
private:
    void expand()
    { Item *t = st;
      init(M+M);
      for (int i = 0; i < M/2; i++)
          if (!t[i].null()) insert(t[i]);
      delete t;
    }
public:
    ST(int maxN)
    { init(4); }
    void insert(Item item)
    { int i = hash(item.key(), M);
      while (!st[i].null()) i = (i+1) % M;
      st[i] = item;
      if (N++ >= M/2) expand();
    }
```

Эту же концепцию можно выразить, говоря, что затраты на одну вставку меньше четырех зондирований. Это утверждение не равносильно утверждению, что для каждой вставки в среднем требуется менее четырех зондирований; действительно, мы знаем, что для тех вставок, которые приводят к удвоению размера таблицы, будет требоваться большое количество зондирований. Это рассуждение — простой пример *амортизационного анализа*: для этого алгоритма нельзя гарантировать быстрое выполнение всех и каждой операции, но можно гарантировать, что средние затраты на одну операцию будут низкими.

Хотя общие затраты низки, профиль производительности вставок неравномерен: большинство операций выполняется исключительно быстро, но для определенных редко выполняемых операций требуется почти столько же времени, сколько ранее было затрачено на построение всей таблицы. При увеличении размера таблицы от 1 тысячи до 1 миллиона ключей это замедление будет происходить около 10 раз. По-

добное поведение приемлемо во многих приложениях, но может оказаться недопустимым, когда абсолютные гарантии производительности желательны или обязательны. Например, в то время как банк или авиакомпания могут допустить, чтобы клиенту приходилось ожидать дополнительное время при выполнении 10 транзакций из 1 миллиона, долгое ожидание может иметь катастрофические последствия в таких приложениях, как онлайн-система обработки финансовых транзакций, или система управления воздушным движением.

Если требуется поддерживать операцию *remove* абстрактного типа данных, может иметь смысл сужать таблицу, вдвое уменьшая ее размер при уменьшении количества ее ключей (см. упражнение 14.44). Но это требует одной оговорки: порог сужения должен отличаться от порога увеличения, поскольку в противном случае небольшое количество операций *insert* и *remove* могло бы приводить к последовательности операций удвоения и уменьшения размера вдвое даже для очень больших таблиц.

**Лемма 14.6** *Последовательность операций  $search$ ,  $insert$  и  $delete$  в таблицах символов может быть выполнена за время, которое пропорционально  $t$ , и при использовании объема памяти, не превышающего числа ключей в таблице, умноженного на постоянный коэффициент.*

Линейное зондирование с увеличением путем удвоения используется во всех случаях, когда операция *insert* приводит к тому, что количество ключей в таблице становится равным половине размера таблицы. Сужение путем уменьшения размера таблицы вдвое используется, когда операция *remove* приводит к тому, что количество ключей в таблице становится равным одной восьмой размера таблицы. В обоих случаях после того, как размер таблицы изменен до значения  $N$ , она содержит  $N/4$  ключей. После этого должно быть выполнено  $N/4$  операций *insert*, прежде чем размер таблицы будет снова удвоен (за счет повторной вставки  $N/2$  ключей в таблицу размера  $2N$ ), и  $N/8$  операций *remove*, прежде чем размер таблицы будет снова вдвое уменьшен (за счет повторной вставки  $N/8$  ключей в таблицу размера  $N/2$ ). В обоих случаях количество повторно вставляемых ключей не превышает двукратного количества операций, которые были выполнены для инициализации перестройки таблицы, поэтому общие затраты остаются линейными. Более того, таблица всегда заполнена от одной восьмой до одной четверти (см. рис. 14.13), следовательно, в соответствии с леммой 14.4 среднее количество зондирований для каждой операции меньше 3.

Этот метод подходит для использования в реализации таблицы символов, предназначенной для библиотеки общего назначения, когда последовательности применения операций непредсказуемы, поскольку позволяет приемлемым образом обрабатывать таблицы любых размеров. Основной его недостаток — затраты на повторное хеширование и распределение памяти при расширении и сжатии таблицы. В типичном случае, когда основными выполняемыми операциями являются операции поиска, гарантия малого заполнения таблицы обеспечивает прекрасную производительность. В главе 16 будет рассмотрен другой подход, позволяющий избежать повторного хеширования и подходящий для очень больших таблиц внешнего поиска.

## Упражнения

- ▷ **14.40** Приведите содержимое хеш-таблицы, образованной в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустую таблицу, имевшую начальный размер  $M = 4$ , которая увеличивается вдвое при ее заполнении наполовину, при разрешении конфликтов методом линейного зондирования. Воспользуйтесь хеш-функцией  $11k \bmod M$  для преобразования  $k$ -той буквы алфавита в индекс таблицы.
- 14.41** Не было бы ли более экономичным увеличивать хеш-таблицу, утраивая (а не удваивая) ее размер при заполнении таблицы наполовину?
- 14.42** Не было бы ли более экономичным увеличивать хеш-таблицу, утраивая ее размер при заполнении таблицы на одну треть (вместо того, чтобы удваивать размер таблицы при ее заполнении наполовину)?
- 14.43** Не было бы ли более экономичным увеличивать хеш-таблицу, удваивая ее размер при заполнении таблицы на три четверти (а не наполовину)?
- 14.44** Добавьте в программу 14.7 функцию *remove*, которая удаляет элемент, как в программе 14.4, но затем сжимает таблицу, вдвое уменьшая ее размер, если после удаления таблица остается пустой на семь восьмых.
- **14.45** Реализуйте версию программы 14.7 для отдельного связывания, которая в 10 раз увеличивает размер таблицы каждый раз, когда средняя длина списка равна 10.
- 14.46** Измените программу 14.7 и реализацию, созданную в упражнении 14.44, чтобы в них использовалось двойное хеширование с "ленивым" удалением (см. упражнение 14.33). Обеспечьте, чтобы программа учитывала количество фиктивных объектов и пустых позиций при принятии решения о необходимости расширения или сужения таблицы.
- 14.47** Разработайте реализацию таблицы символов с использованием линейного зондирования и динамических таблиц, которая содержит деструктор, конструктор копирования и перегруженную операцию присваивания и поддерживает операции *construct*, *count*, *search*, *insert*, *remove* и *join* для АДТ первого класса таблицы символов при поддержке дескрипторов клиента (см. упражнения 12.6 и 12.7).

## 14.6 Перспективы

Как было показано в ходе рассмотрения методов хеширования, выбор метода, который наиболее подходит для конкретного приложения, зависит от множества различных факторов. Все методы могут уменьшать время выполнения функций *search* и *insert*, делая его постоянным, и все методы находят применение в широком множестве приложений. Обобщая, можно охарактеризовать три основных метода (линейное зондирование, двойное хеширование и отдельное связывание) следующим образом: линейное зондирование является самым быстрым из этих трех методов (при наличии достаточного объема памяти, чтобы таблица была разреженной), двойное хеширование позволяет наиболее эффективно использовать память (но требует дополнительных затрат времени для вычисления второй хеш-функции), а отдельное связывание проще всего реализовать и применять (при условии наличия хорошего средства распределения памяти). Экспериментальные данные и комментарии, характеризующие производительность алгоритмов, приведены в табл. 14.1.

Таблица 14.1 Данные экспериментального исследования реализаций хеш-таблиц

Эти относительные значения времени построения и поиска в таблицах символов, состоящих из случайных последовательностей 32-разрядных целых чисел, подтверждают, что для ключей, которые легко поддаются хешированию, хеширование работает значительно быстрее, чем поиск с использованием дерева. Двойное хеширование работает медленнее, чем раздельное связывание и линейное зондирование для разреженных таблиц (из-за необходимости вычисления второй хеш-функции), но значительно быстрее линейного зондирования, когда таблица заполняется; кроме того, этот метод — единственный, который может обеспечить быстрый поиск с использованием лишь небольшого дополнительного объема памяти. Динамические хеш-таблицы, построенные с использованием линейного зондирования и расширения путем удвоения, требуют больших затрат времени при конструировании, чем другие хеш-таблицы, из-за необходимости распределения памяти и повторного хеширования, но несомненно обеспечивают наиболее быстрый поиск. Этот метод является предпочтительным, когда чаще всего выполняется поиск, и когда заранее нельзя точно предвидеть количество ключей.

N	конструирование					промахи при поиске				
	R	H	P	D	P*	R	H	P	D	P*
1250	1	0	5	3	0	1	1	0	1	0
2500	3	1	3	4	2	1	1	0	0	0
5000	6	1	4	4	3	2	1	0	1	0
12500	14	6	5	5	5	6	1	2	2	1
25000	34	9	7	8	11	16	5	3	4	3
50000	74	18	11	12	22	36	15	8	8	8
100000	182	35	21	23	47	84	45	23	21	15
150000		54	40	36	138		99	89	52	21
160000		58	43	44	147		115	133	66	23
170000		68	55	45	136		121	226	85	25
180000		65	61	50	152		133	449	125	27
190000		79	106	59	155		144	2194	261	30
200000	407	84			159	186	156			33

Обозначения:

- R RB-дерево бинарного поиска (программы 12.8 и 13.6)
- H Раздельное связывание (программа 14.3 при размере таблицы, равном 20000)
- P Линейное зондирование (программа 14.4 при размере таблицы, равном 200000)
- D Двойное хеширование (программа 14.6 при размере таблицы, равном 200000)
- P Линейное зондирование с расширением путем удвоения (программа 14.7)

Выбор между линейным зондированием и двойным хешированием зависит прежде всего от затрат на вычисление хеш-функции и от коэффициента загрузки таблицы. Для разреженных таблиц (для малых значений коэффициента  $\alpha$ ) оба метода используют лишь несколько зондирований, но для двойного хеширования может потребоваться больше времени, поскольку необходимо вычислять две хеш-функции для длинных ключей. По мере того как значение  $b$  приближается к 1, двойное хеширование начинает существенно превосходить по производительности линейное зондирование, как видно из табл. 14.1.

Сравнение линейного зондирования и двойного хеширования с отдельным связыванием выполнить сложнее, поскольку необходимо точно учитывать используемый объем памяти. Отдельное связывание использует дополнительный объем памяти для связей; методы с открытой адресацией используют дополнительную память неявно внутри таблицы для завершения последовательностей зондирования. Следующий конкретный пример иллюстрирует ситуацию. Предположим, что имеется таблица  $M$  списков, построенная при помощи отдельного связывания, что средняя длина списков равна 4, а каждый элемент и каждая связь занимают по одному машинному слову. Предположение, что элементы и ссылки занимают одинаковый объем памяти, оправдано во многих ситуациях, поскольку очень большие элементы будут заменены ссылками на них. В этом случае таблица занимает  $9M$  слов в памяти ( $4M$  для элементов и  $5M$  для связей), и для выполнения поиска в среднем требуется 2 зондирования. Но при линейном зондировании для  $4M$  элементов в таблице размером  $9M$  требуется всего  $(1 + 1/(1 - 4/9)) / 2 = 1.4$  зондирований для обнаружения попадания при поиске — что на 30 процентов меньше, чем требуется для отдельного связывания при том же объеме используемой памяти; а при линейном зондировании для  $4M$  элементов в таблице размером  $6M$  требуется 2 зондирования для обнаружения попадания при поиске (в среднем) и, следовательно, используется памяти на 33 процента меньше, чем при отдельном связывании при том же времени выполнения. Более того, для обеспечения увеличения таблицы при остающемся небольшим коэффициенте ее загрузки можно использовать динамический метод, подобный реализованному в программе 14.7.

Приведенные в предшествующем абзаце рассуждения показывают, что обычно выбор отдельного связывания вместо открытой адресации по соображениям производительности не является вполне оправданным. Однако на практике отдельное связывание с фиксированным значением  $M$  часто выбирают по ряду других причин: его легко реализовать (особенно операцию *remove*); оно требует небольшого дополнительного объема памяти для элементов, которые имеют заранее выделенные поля связей, пригодные для использования таблицей символов и другими АТД, которые могут в них нуждаться; и, хотя его производительность снижается с увеличением количества элементов в таблице, это снижение допустимо и происходит так, что вряд ли повредит приложению, поскольку производительность все же в  $M$  раз выше, чем производительность последовательного поиска.

Существует много других методов хеширования, которые находят применение в особых ситуациях. Хотя мы не можем останавливаться на этом подробно, все же кратко рассмотрим три примера, иллюстрирующие сущность специальных методов хеширования.

Один класс методов перемещает элементы во время вставки при двойном хешировании, делая успешный поиск более эффективным. Так, Brent (Brent) разработал метод, при использовании которого среднее время успешного поиска может ограничиваться константой даже в заполненной таблице (см. раздел ссылок). Такой метод может быть удобным в приложениях, в которых попадания при поиске — основная операция.

Другой метод, называемый *упорядоченным хешированием (ordered hashing)*, использует упорядочение для уменьшения затрат на безрезультатный поиск при использовании линейного зондирования, приближая их к затратам на успешный поиск. В стандартном линейном зондировании поиск прекращается при обнаружении пустой позиции таблицы или элемента, ключ которого равен искомому; в упорядоченном хешировании поиск прекращается при обнаружении ключа, который больше или равен искомому ключу (чтобы эта процедура работала, таблица должна конструироваться продуманно) (см. раздел ссылок). Это усовершенствование путем ввода упорядочения в таблицу аналогично достигаемому эффекту от упорядочения списков при отдельном связывании. Этот метод предназначен для приложений, в которых преобладают промахи при поиске.

Таблица символов, в которой быстро происходит обнаружение промахов при поиске и несколько медленнее — попаданий при поиске, может использоваться для реализации *словаря исключений (exception dictionary)*. Например, система текстовой обработки может содержать алгоритм для разбиения слов на слоги, который успешно работает для большинства слов, но не работает в отдельных случаях (таких как "bizarre"). Скорее всего, лишь небольшое количество слов в очень большом документе будет включено в словарь исключений, поэтому, скорее всего, почти все поиски будут завершаться промахами.

Это — лишь некоторые примеры множества алгоритмических усовершенствований, предложенных для хеширования. Многие из них представляют интерес и находят важные применения. Однако, как обычно, следует избегать опрометчивого применения усложненных методов, если только это не обусловлено серьезными причинами, а компромисс между производительностью и сложностью не проанализирован самым тщательным образом, поскольку отдельное связывание, линейное зондирование и двойное хеширование просты, эффективны и приемлемы для большинства приложений.

Задача реализации словаря исключений представляет собой пример приложения, в котором алгоритм можно слегка изменить для оптимизации производительности наиболее часто выполняемой операции — в данном случае обнаружения промаха при поиске. Например, предположим, что имеется словарь исключений, состоящий из 1000 элементов, 1 миллион элементов, которые необходимо искать в словаре, и ожидается, что буквально все поиски должны завершаться промахами. Эта ситуация могла бы возникнуть, если бы все элементы были исключениями языка или случайными 32-разрядными целыми числами. Один из подходов — хеширование всех слов, скажем, 15-разрядными значениями (в этом случае размер таблицы составит около  $2^{16}$ ). 1000 исключений занимают 1/64 часть таблицы и большинство из 1 миллиона поисков немедленно завершаются промахами при обнаружении пустой позиции таблицы при первом же зондировании. Но если таблица содержит 32-разрядные слова, задачу можно выполнить значительно эффективней, преобразовав ее в таблицу исключительных разрядов и используя 20-разрядные хеш-значения. При промахе (как имеет место в большинстве случаев) поиск завершается в результате проверки одного разряда; в случае попадания при поиске требуется выполнения второй проверки в меньшей таблице. Исключения занимают 1/1000 часть таблицы; промахи при поиске — наиболее



вероятная операция; и задача выполняется путем 1 миллиона проверок непосредственно индексированных разрядов. Решение основывается на идее, что хеш-функция создает короткий *сертификат*, представляющий ключ; это важная концепция, находящая применение и в приложениях, которые отличаются от реализаций таблиц символов.

В качестве реализации таблиц символов хеширование предпочтительней структур бинарных деревьев, рассмотренных в главах 12 и 13, поскольку оно несколько проще и может обеспечить оптимальное (постоянное) время поиска, если ключи относятся к стандартному типу или достаточно просты, чтобы можно было быть уверенным в разработке хорошей хеш-функции для них. Преимущества структур бинарных деревьев по сравнению с хешированием заключается в том, что деревья основываются на более простом абстрактном интерфейсе (разработка хеш-функции не требуется); деревья являются динамическими (никакая предварительная информация о количестве вставок не требуется); деревья могут обеспечить гарантированную производительность в худшем случае (все элементы могут быть помещены в одну и ту же позицию даже при использовании наилучшего метода хеширования); и, наконец, деревья поддерживают более широкий диапазон операций (самое главное, операции *sort* и *select*). Когда эти факторы не имеют значения, безусловно следует выбирать хеширование, только с одной важной оговоркой: когда ключи представляют собой длинные строки, их можно встроить в структуры данных, которые могут обеспечить методы поиска, работающие еще быстрее хеширования. Подобного рода структуры — тема главы 15.

## Упражнения

- ▷ **14.48** Для 1 миллиона целочисленных ключей вычислите размер хеш-таблицы, при которой каждый из трех методов хеширования (раздельное связывание, линейное зондирование и двойное хеширование) использует для обнаружения промаха при вставке в среднем столько же сравнений с ключами, как и BST-деревья, подсчитывая вычисление хеш-функции как сравнение.
- ▷ **14.49** Для 1 миллиона целочисленных ключей вычислите количество сравнений, выполняемых каждым из трех методов хеширования (раздельным связыванием, линейным зондированием и двойным хешированием) в среднем для обнаружения промаха при поиске, когда они всего могут использовать 3 миллиона слов памяти (как имело бы место в случае BST-деревьев).
- 14.50** Реализуйте АТД таблицы символов, обеспечивающий быстрое обнаружение промаха при поиске, как описано в тексте, используя для второй проверки раздельное связывание.

## Поразрядный поиск

В нескольких методах поиска обработка ведется за счет исследования ключей поиска по небольшим фрагментам за раз вместо того, чтобы на каждом шаге сравнивать полные значения ключей. Эти методы носят название *поразрядного поиска (radix-search)* и работают аналогично методам поразрядной сортировки, которые рассматривались в главе 10. Они удобны, когда фрагменты ключей поиска легкодоступны, и могут обеспечить эффективные решения для множества реальных применений поиска.

В данном случае применяется та же абстрактная модель, которая использовалась в главе 10: в зависимости от контекста *ключ* может быть *словом* (последовательностью байтов фиксированной длины) или *строкой* (последовательностью байтов переменной длины). Ключи, являющиеся словами, обрабатываются как числа, представленные в системе счисления с основанием  $R$  при различных значениях  $R$  (*основания системы счисления*), причем действия выполняются над отдельными цифрами чисел. Строки в стиле  $S$  можно рассматривать как числа переменной длины, ограничиваемые специальным символом, чтобы для ключей как фиксированной, так и переменной длины алгоритмы могли основываться на абстрактной операции "извлечения  $i$ -той цифры из ключа", в том числе и когда ключ содержит менее  $i$  цифр.

Принципиальные преимущества методов поразрядного поиска заключается в следующем: они обеспечивают приемлемую производительность для худшего случая без сложностей, присущих сбалансированным деревьям; они обеспечивают простой способ обработки ключей переменной длины; некоторые из них позволяют экономить память, сохраняя часть ключа внутри поисковой структуры; они могут обеспечить быстрый доступ к данным, конку-

рируя как с деревьями бинарного поиска (BST-деревьями), так и с хешированием. Недостатки этих методов связаны с тем, что некоторые методы могут приводить к неэффективному использованию памяти, а при поразрядной сортировке производительность может снижаться в случае отсутствия эффективного доступа к байтам ключей.

Вначале мы рассмотрим несколько методов поиска, которые работают посредством исследования ключей поиска по одному разряду, используя их для перемещения по структурам бинарных деревьев. Мы рассмотрим ряд методов, каждый из которых минимизирует проблемы, характерные для предыдущего, а в завершение познакомимся с остроумным методом, находящим применение в ряде приложений поиска.

Затем мы исследуем обобщение *R-путевых* деревьев. Как и в предыдущем случае, мы рассмотрим ряд методов, представив в завершение гибкий и эффективный метод, который может поддерживать базовую реализацию таблицы символов и множество ее расширений.

Обычно при поразрядном поиске вначале исследуются старшие цифры ключей. Многие методы непосредственно соответствуют методам поразрядной сортировки "сначала по старшей цифре" (most significant digit — MSD) подобно тому, как поиск, основанный на BST-дереве, соответствует быстрой сортировке. В частности, мы рассмотрим аналоги методов сортировки с линейной зависимостью времени выполнения, приведенных в главе 10 — методы поиска с линейной зависимостью времени выполнения, основанные на том же принципе.

В конце главы приводится специальное приложение, в котором структуры поразрядного поиска задействуются в построении индексов для больших текстовых строк. Рассмотренные в этой главе методы обеспечивают естественные решения для этого приложения и помогают заложить основу для рассмотрения более сложных задач обработки строк, приведенных в части 5.

## 15.1 Деревья цифрового поиска

Простейший метод поразрядного поиска основан на использовании *деревьев цифрового поиска* (*digital search trees* — *DST*), на которые в дальнейшем будем ссылаться как на *DST-деревья*. Алгоритмы *поиска* и *вставки* аналогичны поиску в бинарном дереве, за исключением одного различия: ветвление в дереве выполняется не в соответствии с результатом сравнения полных ключей, а в соответствии с выбранными разрядами ключа. На первом уровне используется ведущий разряд; на втором уровне используется разряд, следующий за ведущим, и т.д., пока не

```
A 00001
S 10011
E 00101
R 10010
C 00011
H 01000
I 01001
N 01110
G 00111
X 11000
M 01101
P 10000
L 01100
```

### РИСУНОК 15.1 ДВОИЧНЫЕ ПРЕДСТАВЛЕНИЯ ОДНОСИМВОЛЬНЫХ КЛЮЧЕЙ

*Подобно тому, как это было сделано в главе 10, в небольших примерах, приведенных на рисунках этой главы, 5-разрядное двоичное представление числа  $i$  используется для представления  $i$ -той буквы алфавита, как здесь продемонстрировано на примере нескольких ключей. Предполагается, что биты нумеруются слева направо от 0 до 4.*

встретится внешний узел. Программа 15.1 содержит реализацию операции *поиска* (*search*); реализация операции *вставки* (*insert*) аналогична. Вместо того чтобы для сравнения ключей использовать операцию  $<$ , мы исходим из предположения о наличии функции **digit**, обеспечивающей доступ к отдельным разрядам ключей. Этот код буквально совпадает с кодом реализации поиска в бинарном дереве (см. программу 12.8), но, как будет показано, имеет существенно иные характеристики производительности.

### Программа 15.1 Бинарное DST-дерево

Чтобы разработать реализацию таблицы символов, в которой используются DST-деревья, мы изменили реализации операций *search* и *insert* в стандартном BST-дереве (см. программу 12.8), что должно быть заметно в этом примере операции *search*. Для принятия решения о том, следует ли выполнять перемещение влево или вправо, вместо сравнения полных ключей выполняется проверка единственного (ведущего) разряда ключа. Рекурсивные вызовы функции имеют третий аргумент, позволяющий смещать вправо позицию проверяемого разряда при перемещении вниз по дереву. Для проверки разрядов используется функция **digit**, как было описано в разделе 10.1. Эти же изменения применяются к реализации операции *insert*; в остальном используется код программы 12.8.

**private:**

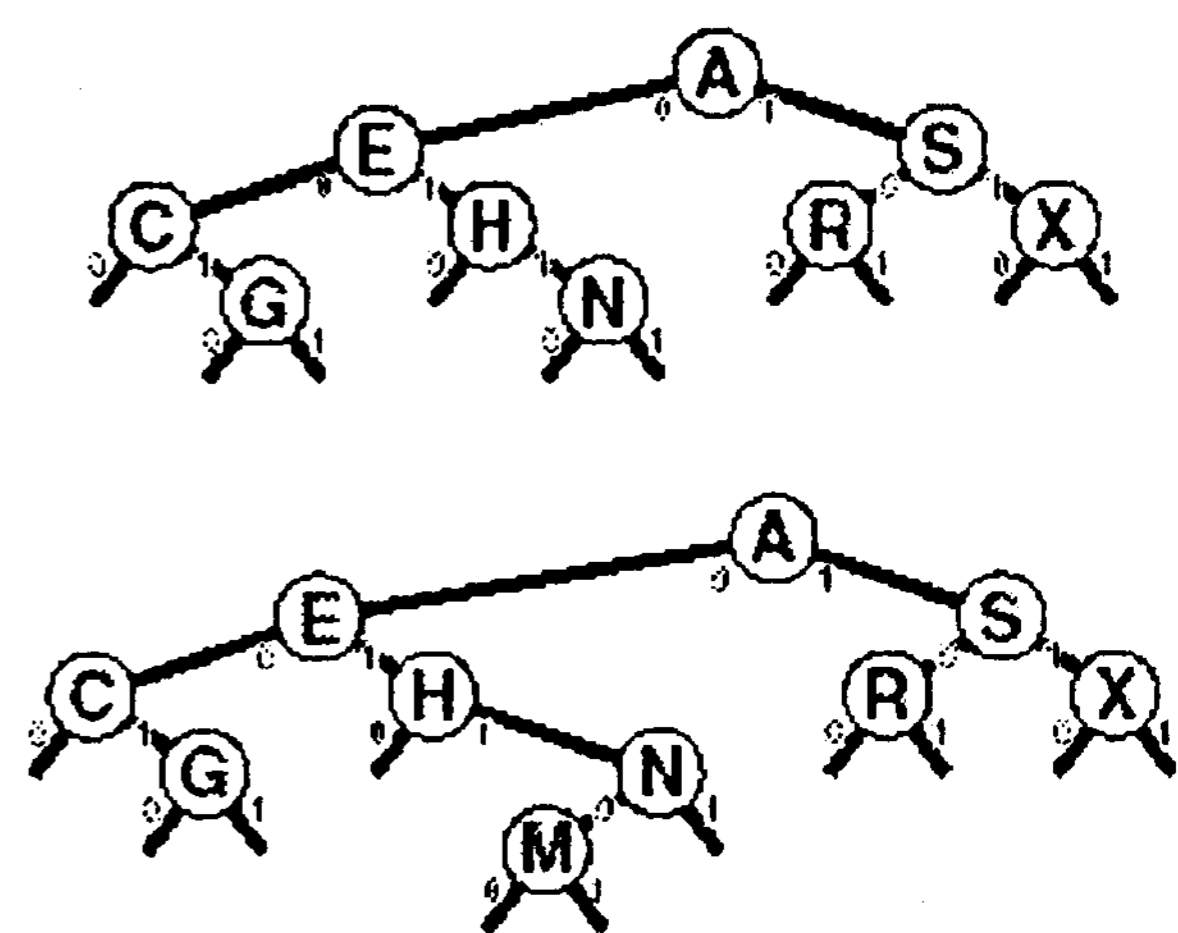
```
Item searchR(link h, Key v, int d)
{ if (h == 0) return nullItem;
  if (v == h->item.key()) return h->item;
  if (digit(v, d) == 0)
    return searchR(h->l, v, d+1);
  else return searchR(h->r, v, d+1);
}
```

**public:**

```
Item search(Key v)
{ return searchR(head, v, 0); }
```

В главе 10 было показано, что при использовании поразрядной сортировки особое внимание должно быть уделено совпадающим ключам; то же самое справедливо и по отношению к поразрядному поиску. В этой главе предполагается, что все значения ключей в таблице символов являются различными. Это предположение не ведет к нарушению общности, поскольку для поддержания приложений, содержащих записи с дублированными ключами, можно воспользоваться одним из методов, рассмотренных в разделе 12.1. При исследовании поразрядного поиска важно сосредоточиться на различных значениях ключей, поскольку значения ключей являются важными компонентами нескольких структур данных, которые рассматриваются в дальнейшем.

На рис. 15.1 приведены двоичные представления однобуквенных ключей, используемых на остальных рисунках этой главы. На рис. 15.2 показан пример



**РИСУНОК 15.2 DST-ДЕРЕВО И ВСТАВКА**

При выполнении безуспешного поиска ключа  $M=01101$  в этом примере DST-дерева (вверху) мы перемещаемся влево от корня (поскольку первый разряд в двоичном представлении ключа содержит 0), а затем вправо (поскольку второй разряд содержит 1), затем вправо, влево и завершаем поиск на нулевой связи под ключом N. Для вставки ключа M (внизу) мы заменяем нулевую связь в месте завершения поиска связью с новым узлом, как это делается при вставке в BST-дерево.

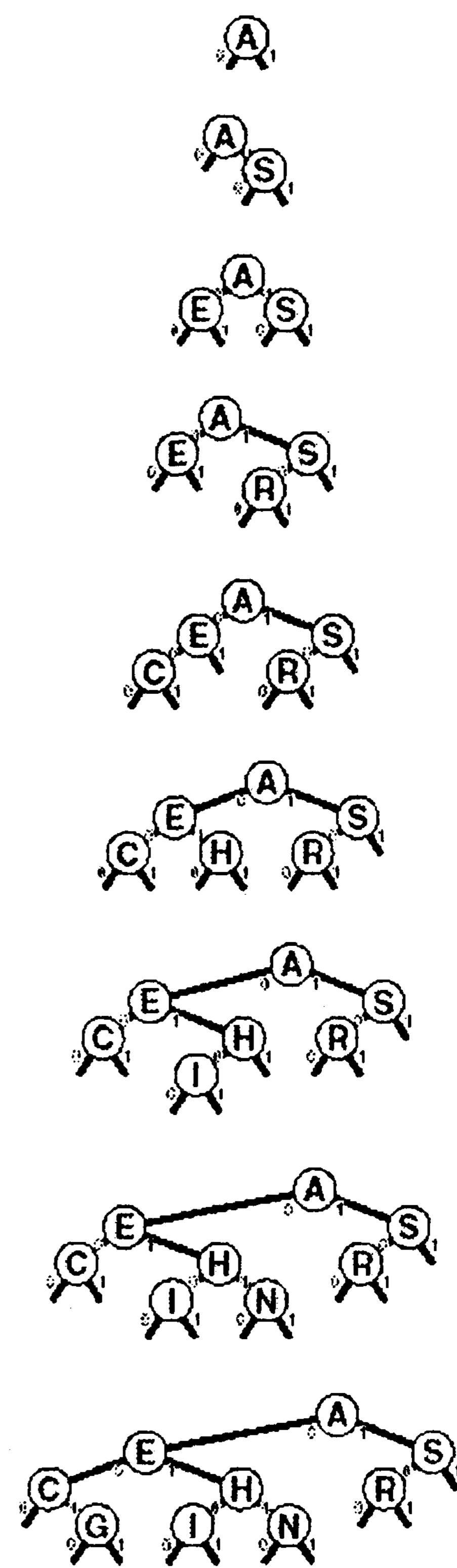
вставки в DST-дерево, а на рис. 15.3 — процесс вставки ключей в первоначально пустое дерево.

Разряды ключей управляют поиском и вставкой, но обратите внимание, что DST-деревья не обладают свойством упорядоченности, характерным для BST-деревьев. Другими словами, ключи в узлах слева от данного *не* обязательно меньше, а ключи в узлах справа от данного *не* обязательно больше ключей данного узла, как это было бы в BST-дереве с различными ключами. Ключи слева от данного узла действительно меньше ключей справа от него — если узел находится на уровне  $k$ , все они совпадают в первых  $k$  разрядах, но следующий разряд равен 0 для ключей слева и 1 для ключей справа — но сам ключ узла может быть наименьшим, наибольшим или любым в диапазоне всех ключей из поддерева этого узла.

DST-деревья характеризуются тем, что каждый ключ размещается *где-то* вдоль пути, определенного разрядами ключа (следующими слева направо). Этого свойства достаточно, чтобы реализации операций *search* и *insert* в программе 15.1 работали правильно.

Предположим, что ключи — слова фиксированной длины, каждое из которых состоит из  $w$  разрядов. Требование, чтобы ключи были различными, обуславливает, что  $N = 2^w$ , и обычно предполагается, что  $N$  значительно меньше  $2^w$ , поскольку в противном случае имело бы смысл использовать поиск с индексированием по ключу (см. раздел 12.2). Этому условию соответствует множество реальных задач. Например, использование DST-деревьев вполне подходит для таблицы символов, содержащей вплоть до  $10^5$  записей с 32-разрядными ключами (но, скорее всего, не  $10^6$  записей), или любое количество записей с 64-разрядными ключами. DST-деревья работают также и в случае ключей переменной длины; отложим подробное рассмотрение этого случая до раздела 15.2, где будет рассматриваться и ряд других альтернатив.

Производительность для худшего случая деревьев, построенных по методу поразрядного поиска, значительно выше производительности для худшего случая BST-деревьев, если количество ключей велико, а длина ключей мала по сравнению с их количеством. Вероятней всего, во многих приложениях (например, если ключи образованы случайными значениями разрядов) длина самого длинного пути в DST-дереве оказывается сравнительно небольшой. В частности, самый длинный путь ограничивается длиной самого длинного ключа; более того, если ключи имеют фиксированную длину, то время поиска ограничивается длиной. Сказанное иллюстрируется на рис. 15.4.



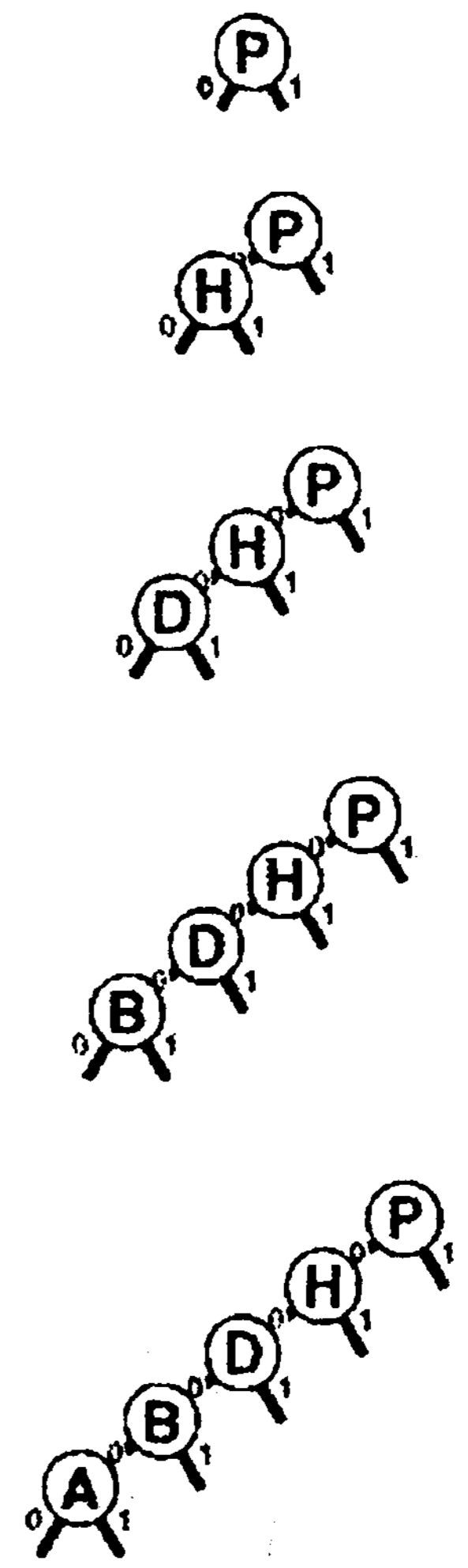
**РИСУНОК 15.3**  
**ПОСТРОЕНИЕ DST-ДЕРЕВА**

На этой последовательности рисунков показан результат вставки ключей *A S E R C H I N G* в первоначально пустое DST-дерево.

**Лемма 15.1** Для выполнения поиска или вставки в DST-дереве, построенном из  $N$  случайных ключей, требуется около  $\lg N$  сравнений в среднем и около  $2 \lg N$  сравнений в худшем случае. Количество сравнений никогда не превышает количество разрядов в ключе поиска.

Утверждения этой леммы для случайных ключей можно доказать при помощи рассуждений, которые аналогичны приведенным для более естественной задачи в следующем разделе, поэтому здесь мы предоставляем читателям самостоятельно вывести доказательство (см. упражнение 15.30). Доказательство основывается на интуитивном представлении о том, что невидимая часть случайного ключа с равной вероятностью должна начинаться с 0 или 1 разряда, поэтому половина приходится на любую сторону любого ключа. При каждом перемещении вниз по дереву мы используем разряд ключа, поэтому ни для какого поиска в DST-дереве не может требоваться больше сравнений, чем разрядов в ключе поиска. Для типового случая, когда имеются  $w$ -разрядные слова и количество ключей  $N$  значительно меньше общего возможного количества ключей, равного  $2^w$ , длины путей близки к  $\lg N$ . Поэтому для случайных ключей количество сравнений значительно меньше количества разрядов в ключах.

На рис. 15.5 показано большое DST-дереве, образованное случайными 7-разрядными ключами. Это дерево почти идеально сбалансировано. Использование DST-деревьев заманчиво во многих реальных приложениях, поскольку эти деревья обеспечивают практически оптимальную производительность даже при решении очень сложных задач, требуя лишь минимальных усилий на реализацию. Например, DST-дереве, построенное из 32-разрядных ключей (или четырех 8-разрядных символов), гарантировано требует менее 32 сравнений, а DST-дереве, построенное из 64-разрядных ключей (или восьми 8-разрядных символов), гарантировано требует менее 64 сравнений, даже при наличии миллионов ключей. Для большого значения  $N$  подобные гарантии сравнимы с теми, которые обеспечивают красно-черные (RB-) деревья, но для их реализации требуются почти такие же усилия, как и для реализации стандартных BST-деревьев (которые могут гарантировать только производительность, пропорциональную  $N^2$ ). Это свойство делает DST-деревья привлекательной альтернативой использованию сбалансированных деревьев для практической реализации функций *search* и *insert* в таблице символов, при условии наличия эффективного доступа к разрядам ключей.



**РИСУНОК 15.4 DST-ДЕРЕВО ДЛЯ ХУДШЕГО СЛУЧАЯ**

В этой последовательности рисунков отображены результаты вставки ключей  $P = 10000$ ,  $H = 01000$ ,  $D = 00100$ ,  $B = 00010$  и  $A = 00001$  в первоначально пустое DST-дереве. Последовательность деревьев кажется вырожденной, но длина пути ограничивается длиной двоичного представления ключей. Ни один 5-разрядный ключ, за исключением  $00000$ , не приводит к дальнейшему увеличению высоты дерева.

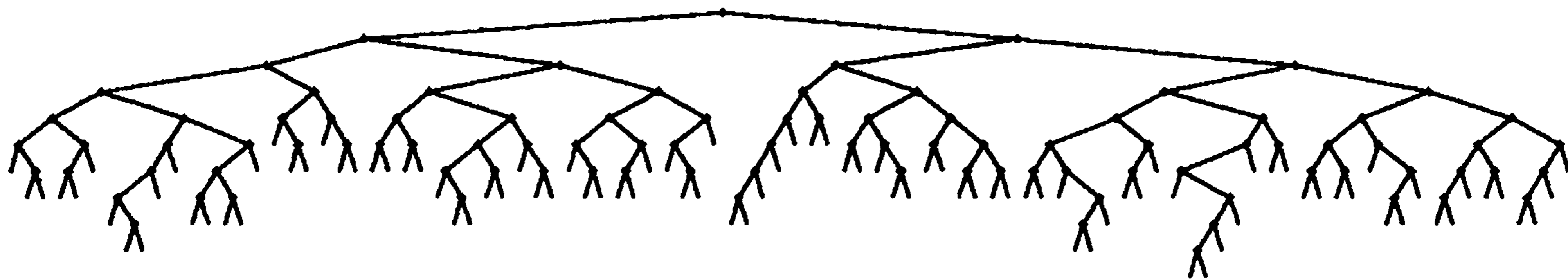


РИСУНОК 15.5 ПРИМЕР DST-ДЕРЕВА

Это DST-дерево, построенное в результате вставки около 200 случайных ключей, хорошо сбалансировано.

## Упражнения

- ▷ **15.1** Нарисуйте DST-дерево, образованное в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое дерево при использовании двоичного кодирования, приведенного на рис. 15.1.
- 15.2** Покажите последовательность вставки ключей **A B C D E F G**, приводящую к образованию полностью сбалансированного DST-дерева, одновременно являющегося допустимым BST-деревом.
- 15.3** Покажите последовательность вставки ключей **A B C D E F G**, приводящую к образованию полностью сбалансированного DST-дерева, характеризующегося тем, что каждый его узел имеет ключ, который меньше ключей всех узлов в его поддереве.
- ▷ **15.4** Нарисуйте DST-дерево, образованное в результате вставки элементов с ключами **01010011 00000111 00100001 01010001 11101100 00100001 10010101 01001010** в указанном порядке в первоначально пустое дерево.
- 15.5** Можно ли в DST-деревьях хранить записи с дублированными ключами, как это возможно в BST-деревьях? Обоснуйте свой ответ.
- 15.6** Экспериментально сравните высоту и длину внутреннего пути DST-дерева, построенного в результате вставки  $N$  случайных 32-разрядных ключей в первоначально пустое дерево, с этими же характеристиками стандартного BST-дерева и RB-дерева (см. главу 13), построенных из этих же ключей, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **15.7** Приведите полное определение длины внутреннего пути для худшего случая DST-дерева, содержащего  $N$  различных  $w$ -разрядных ключей.
- **15.8** Реализуйте операцию *remove* для таблицы символов, основанной на DST-дереве.
- **15.9** Реализуйте операцию *select* для таблицы символов, основанной на DST-дереве.
- **15.10** Опишите, как можно было бы вычислить высоту DST-дерева, образованного заданным набором ключей, при линейных затратах времени, не прибегая к строительству DST-дерева.

## 15.2 Trie-деревья

В этом разделе мы рассмотрим деревья поиска, которые позволяют использовать разряды ключей для проведения поиска подобно DST-деревьям, но ключи которых упорядочены, что позволяет поддерживать рекурсивные реализации функции *sort* и других функций таблиц символов, как это делалось в отношении BST-деревьев. Основная идея заключается в хранении ключей только в нижней части дерева, в листьях. Результирующая структура данных обладает рядом полезных свойств и служит основой для нескольких эффективных алгоритмов поиска. Впервые структура была создана Брианде (Briandais) в 1959 г., и поскольку она оказалась удобной для поиска (*retrieval*), в 1960 г. Фредкин (Fredkin) применил к ней специальное название *trie*. С некоторой долей юмора обычно это слово произносится как "трай-и" (*try* — попытка, англ.), дабы отличать его от "tree". Вероятно, в соответствии с принятой в книге терминологией, подобного рода деревья следовало бы называть "trie-деревьями бинарного поиска", тем не менее, повсеместно используется термин *trie-дерево* и все его понимают. В этом разделе рассматривается базовая бинарная версия, в разделе 15.3 — важную вариацию, а в разделах 15.4 и 15.5 — базовую версию trie-деревьев со многими путями и ее вариации.

Trie-деревья можно использовать для ключей, которые содержат фиксированное количество разрядов либо являются строками разрядов переменной длины. Для простоты начнем рассмотрение, предположив, что ни один ключ поиска не является префиксом другого ключа. Например, это условие выполняется, когда ключи имеют фиксированную длину и являются различными.

В trie-дереве ключи хранятся в *листьях* бинарного дерева. Вспомните из раздела 5.4, что лист в дереве — это узел, не имеющий дочерних узлов, что отличает его от внешнего узла, который интерпретируется как нулевой дочерний узел. В бинарном дереве под листом понимается внутренний узел, левая и правая связи которого являются нулевыми. Хранение ключей в листьях, а не во внутренних узлах позволяет использовать разряды ключей для проведения поиска, как это делалось применительно к DST-деревьям в разделе 15.1, сохраняя при этом свойство, что все ключи, текущий разряд которых равен 0, попадают в левое поддерево, а все ключи, текущий разряд которых равен 1 — в правое.

**Определение 15.1** *Под trie-деревом понимается бинарное дерево, имеющее ключи, связанные с каждым из его листьев, и рекурсивно определенное следующим образом. Trie-дерево, состоящее из пустого набора ключей, представляет собой нулевую связь. Trie-дерево, состоящее из единственного ключа — это лист, содержащий данный ключ. И, наконец, trie-дерево, содержащее намного больше одного ключа — это внутренний узел, левая связь которого ссылается на trie-дерево с ключами, начинающимися с 0 разряда, а правая — на trie-дерево с ключами, начинающимися с 1 разряда, причем для конструирования поддеревьев ведущий разряд такого дерева должен быть удален.*

Каждый ключ в trie-дереве хранится в листе, в пути, описанном ведущей последовательностью разрядов ключа. И наоборот, каждый лист в trie-дереве содержит только ключ, который начинается с разрядов, определенных в пути от корня до этого листа. Нулевые связи в узлах, не являющихся листьями, соответствуют последовательностям ведущих разрядов, которые не присутствуют ни в одном ключе trie-дерева.



Следовательно, для поиска ключа в trie-дереве нужно всего лишь просмотреть его ветви в соответствии с разрядами ключа, как это делалось в DST-деревьях, но при этом не нужно выполнять сравнение во внутренних узлах. Поиск начинается слева от ключа, с верхушки дерева, с продвижением по левой связи, если текущий разряд — 0, и по правой, если он — 1; при этом перемещение внутри ключа выполняется на один разряд вправо. Поиск, завершающийся на нулевой связи, считается промахом; поиск, который заканчивается в листе, может быть завершён при помощи одного сравнения с ключом, поскольку этот узел содержит единственный ключ в дереве, который может быть равен искомому. Программа 15.2 содержит реализацию этого процесса.

### Программа 15.2 Поиск в trie-дереве

В этой функции разряды ключа используются для управления переходом по ветвям при перемещении вниз по дереву, как это делается в программе 15.1 для DST-деревьев. Возможны три варианта: если поиск достигает листа (с обоими нулевыми связями), значит, это уникальный узел trie-дерева, который может содержать запись с ключом *v*. В такой ситуации выполняется проверка, действительно ли этот узел содержит *v* (попадание при поиске) или какой-либо ключ, ведущие разряды которого совпадают с *v* (промах при поиске). Если поиск достигает нулевой связи, то вторая связь родительского узла не должна быть нулевой и, следовательно, в trie-дереве существует какой-либо другой ключ, соответствующий разряд которого отличается от ключа поиска, и имеет место промах при поиске. В программе предполагается, что ключи являются различными и (если ключи могут иметь различную длину) ни один ключ не является префиксом другого ключа. Член *item* не используется в узлах, которые не являются листьями.

**private:**

```
Item searchR(link h, Key v, int d)
{ if (h == 0) return nullItem;
  if (h->l == 0 && h->r == 0)
  { Key w = h->item.key();
    return (v == w) ? h->item : nullItem; }
  if (digit(v, d) == 0)
    return searchR(h->l, v, d+1);
  else return searchR(h->r, v, d+1);
}
```

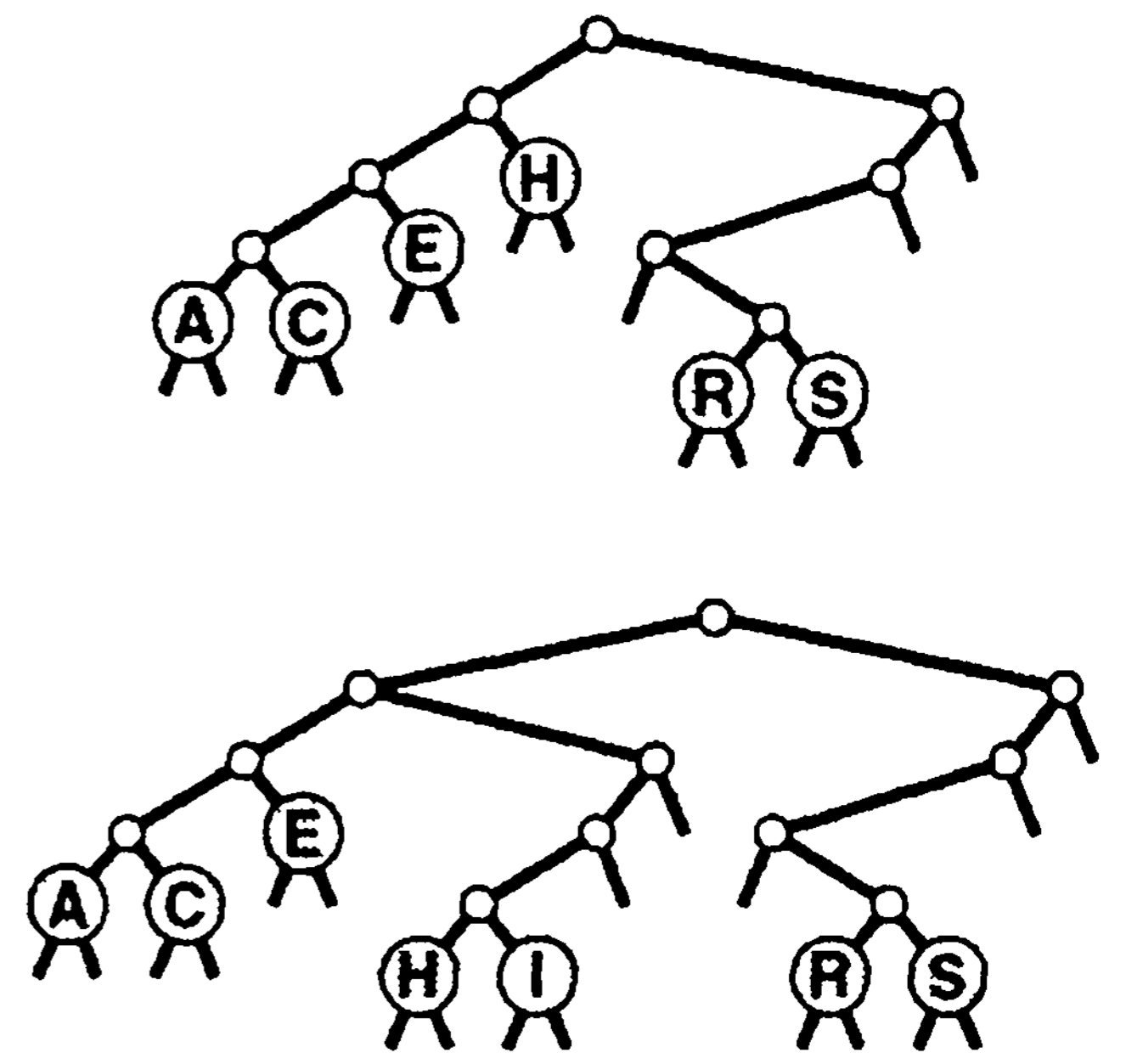
**public:**

```
Item search(Key v)
{ return searchR(head, v, 0); }
```

Для вставки ключа в trie-дерево вначале, как обычно, выполняется поиск. Если поиск завершается на нулевой связи, она, как обычно, заменяется связью с новым содержащим ключ листом. Но если поиск заканчивается в листе, необходимо продолжить перемещение вниз по дереву, добавляя внутренний узел для каждого разряда, значение которого для искомого и найденного ключей совпадает; этот процесс должен завершиться тем, что оба ключа в листьях, являющихся дочерними узлами внутреннего узла, будут соответствовать первому разряду, в котором они отличаются. Пример поиска и вставки в trie-дереве показан на рис. 15.6; процесс построения trie-дерева за счет вставки ключей в первоначально пустое дерево представлен на рис. 15.7. Программа 15.3 представляет собой полную реализацию алгоритма вставки.

Для успешного поиска ключа  $H = 01000$  в этом примере trie-дерева (верхний рисунок) выполняется перемещение влево от корня (поскольку первый разряд в двоичном представлении ключа равен 0), затем вправо (поскольку второй разряд равен 1), где происходит обнаружение  $H$ , который является единственным ключом в дереве, начинающимся с последовательности  $01$ . Ни один ключ в trie-дереве не начинается с  $101$  или  $11$ ; эти последовательности разрядов ведут к нулевым связям, находящимся в узлах, которые не являются листьями.

Для вставки ключа  $I$  (нижний рисунок) потребуется добавить три узла, не являющиеся листьями: узел, соответствующий последовательности  $01$ , с нулевой связью, соответствующей последовательности  $011$ ; узел, соответствующий  $010$ , с нулевой связью, которая соответствует  $0101$ ; и узел, соответствующий последовательности  $0100$ , с ключом  $H = 01000$  в листе слева и с ключом  $I = 01001$  в листе справа от него.



**РИСУНОК 15.6 ПОИСК И ВСТАВКА В TRIE-ДЕРЕВЕ**

*Ключи в trie-дереве хранятся в листьях (узлах с обоими нулевыми связями); нулевые связи в узлах, которые не являются листьями, соответствуют последовательностям разрядов, не найденным ни в одном ключе trie-дерева.*

### Программа 15.3 Вставка в trie-дерево

Для вставки нового узла в trie-дерево вначале, как обычно, выполняется поиск. В случае промаха возможны два варианта.

Если промах имел место не в листе, нулевая связь, приведшая к выявлению промаха, как обычно, заменяется связью с новым узлом.

Если промах имел место в листе, функция **split** используется для создания по одному нового внутреннего узлу для каждой разрядной позиции, в которой ключ поиска и найденный ключ совпадают. Процесс завершается созданием одного внутреннего узла для самой левой разрядной позиции, в которой эти ключи различаются. Оператор **switch** в функции **split** преобразует два проверяемых разряда в число для управления четырьмя возможными случаями. Если разряды одинаковы (случай  $00_2 = 0$  или  $11_2 = 3$ ), разделение продолжается; если разряды различны (случай  $01_2 = 1$  или  $10_2 = 2$ ), разделение прекращается.

**private:**

```
link split(link p, link q, int d)
{ link t = new node(nullItem); t->N = 2;
  Key v = p->item.key(); Key w = q->item.key();
  switch(digit(v, d)*2 + digit(w, d))
  { case 0: t->l = split(p, q, d+1); break;
    case 1: t->l = p; t->r = q; break;
    case 2: t->r = p; t->l = q; break;
    case 3: t->r = split(p, q, d+1); break;
  }
  return t;
}

void insertR(link& h, Item x, int d)
{ if (h == 0) { h = new node(x); return; }
  if (h->l == 0 && h->r == 0)
    { h = split(new node(x), h, d); return; }
```

```

    if (digit(x.key(), d) == 0)
        insertR(h->l, x, d+1);
    else insertR(h->r, x, d+1);
}
public:
    ST(int maxN)
    { head = 0; }
    void insert(Item item)
    { insertR(head, item, 0); }

```

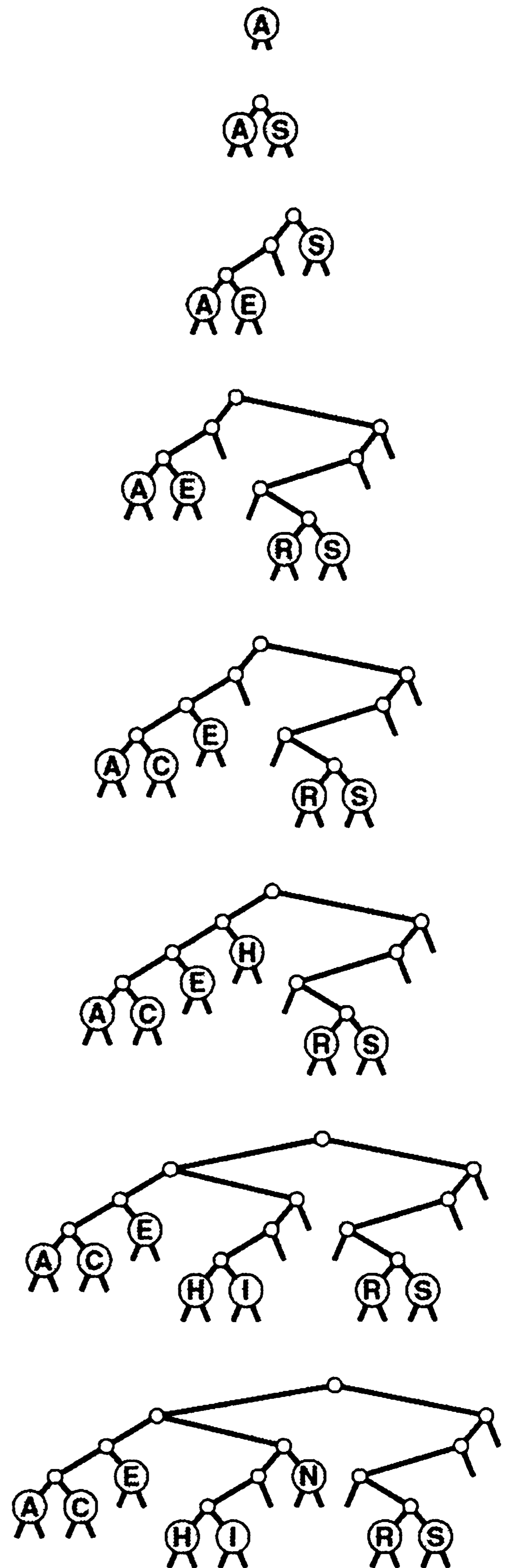
Мы не обращаемся к нулевым связям в листьях и не храним элементы в узлах, не являющихся листьями, поэтому можно сократить объем используемой памяти, применив **union** или пару производных классов для определения узлов, как принадлежащих к одному из этих двух типов (см. упражнения 15.20 и 15.21). Пока стоит пойти более простым путем, используя единственный тип узлов, который имел место в BST-деревьях, DST-деревьях и других структурах бинарных деревьев, внутренние узлы которых характеризуются нулевыми ключами, а листья — нулевыми связями, памятуя, что при необходимости можно воспользоваться памятью, теряемой из-за этого упрощения. В разделе 15.3 будет рассмотрено усовершенствование алгоритма, исключающее потребность в нескольких типах узлов, а в главе 16 приводится реализация, в которой применяется **union**.

Теперь давайте рассмотрим основные свойства trie-деревьев, вытекающие из определения и приведенных примеров.

**Лемма 15.2** Структура trie-дерева не зависит от порядка вставки ключей: для каждого данного набора различных ключей существует уникальное trie-дерево.

Этот основополагающий факт, который можно доказать методом математической индукции, примененным к поддеревьям — отличительная особенность trie-деревьев: для всех остальных рассмотренных структур деревьев поиска создаваемое дерево зависит как от набора ключей, так и от порядка их вставки.

Левое поддерево trie-дерева содержит все ключи, ведущий разряд которых равен 0, а правое поддерево — все ключи, ведущий разряд которых равен 1. Это свойство trie-деревьев обуславливает прямое соответствие с поразрядным поиском: при поиске с использованием бинарного trie-дерева файл делится совершенно так же, как при бинарной быстрой сортировке (см. раздел 10.2). Такое соответствие становится оче-



**РИСУНОК 15.7 ПОСТРОЕНИЕ TRIE-ДЕРЕВА**

На этой последовательности рисунков показан результат вставки ключей A S E R C H I N в первоначально пустое trie-дерево.

видным при сравнении trie-дерева, показанного на рис. 15.6, со схемой разделения на рис. 10.4 (с учетом незначительного различия в ключах); это аналогично соответствию между поиском с использованием бинарного дерева и быстрой сортировкой (см. главу 12).

В частности, в отличие от DST-деревьев, trie-деревья не обладают свойством упорядоченности ключей, поэтому операции *sort* и *select* в таблице символов могут быть реализованы непосредственно (см. упражнения 15.17 и 15.18). Более того, trie-деревья столь же хорошо сбалансированы, как и DST-деревья.

**Лемма 15.3.** *Для выполнения вставки или поиска случайного ключа в trie-дереве, построенном из  $N$  случайных (различных) строк разрядов, требуется в среднем около  $\lg N$  сравнений разрядов. В худшем случае количество сравнений разрядов не превышает количество разрядов в искомом ключе.*

К анализу trie-деревьев необходимо подходить очень внимательно в связи с требованием, что ключи должны быть различными, или, в более общем случае, что ни один ключ не должен быть префиксом другого ключа. Одна из простых моделей, соответствующая этому условию, требует, чтобы ключи были случайной (бесконечной) последовательностью разрядов — из нее выбираются разряды, необходимые для построения trie-дерева.

Тогда производительность для среднего случая можно обосновать, исходя из следующих вероятностных соотношений. Вероятность того, что каждый из  $N$  ключей в случайном trie-дереве отличается от случайного ключа поиска по меньшей мере в одном из  $t$  ведущих разрядов, равна

$$\left(1 - \frac{1}{2^t}\right)^N.$$

Вычитание этого значения из 1 дает вероятность того, что один из ключей в trie-дереве совпадает с ключом поиска во всех  $t$  ведущих разрядах. Другими словами,

$$1 - \left(1 - \frac{1}{2^t}\right)^N$$

это вероятность того, что для выполнения поиска требуется выполнение более  $t$  сравнений разрядов. В соответствии с элементарными соотношениями теории вероятностей для  $t \geq 0$  сумма вероятностей того, что случайная переменная будет больше 1, совпадает со средним значением этой случайной переменной, и, следовательно, средние затраты на поиск определяются выражением

$$\sum_{t \geq 0} \left(1 - \left(1 - \frac{1}{2^t}\right)^N\right).$$

Воспользовавшись элементарной аппроксимацией  $(1 - 1/x)^x \sim e^{-1}$ , находим, что затраты на поиск должны быть приблизительно равны

$$\sum_{t \geq 0} \left(1 - e^{-N/2^t}\right).$$

Значения приблизительно  $\lg N$  членов этой суммы, для которых  $2^i$  значительно меньше  $N$ , очень близки к 1; значения всех членов, для которых  $2^i$  значительно больше  $N$ , близки к 0; и значения нескольких членов, для которых  $2^i \approx N$ , лежат в интервале между 0 и 1. Таким образом, значение суммы приближенно равно  $\lg N$ . Для более точного определения этого значения требуется выполнение очень сложных математических вычислений (см. раздел ссылок). В приведенном анализе предполагается, что значение  $w$  достаточно велико, чтобы во время поиска никогда не возникала ситуация недостаточного количества разрядов; тем не менее, учет действительного значения  $w$  лишь уменьшит получаемое значение затрат.

В худшем случае можно было бы получить два ключа, имеющие огромное количество одинаковых разрядов, но вероятность подобного события ничтожно мала. Вероятность того, что утверждение леммы 15.3 для худшего случая не соблюдается, экспоненциально стремится к нулю (см. упражнение 15.29).

Еще один подход к анализу trie-деревьев заключается в обобщении подхода, который использовался при анализе BST-деревьев (см. лемму 12.6). Вероятность того, что  $k$  ключей начинаются с 0 разряда, а  $N - k$  ключей начинаются с 1 разряда, равна

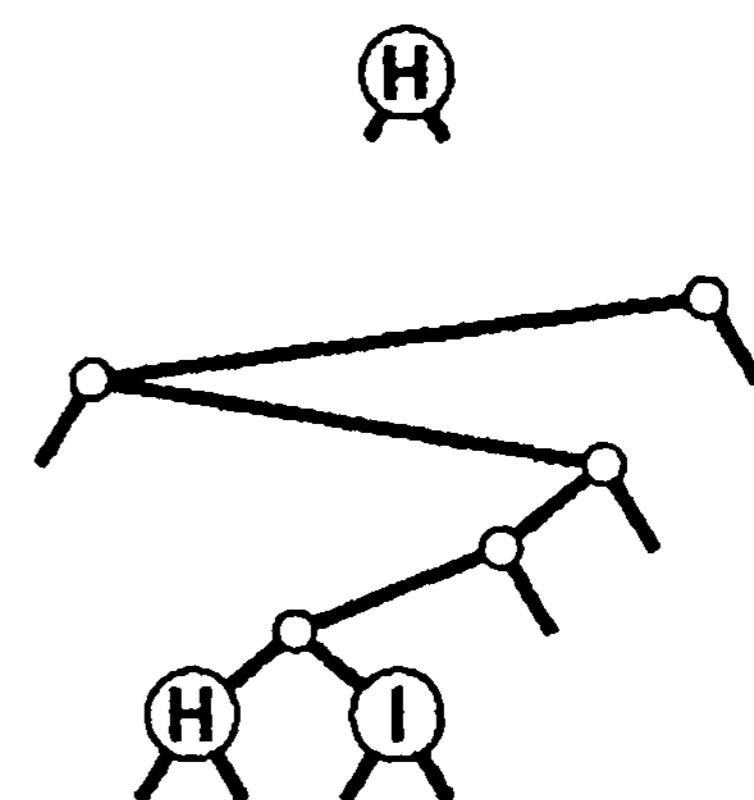
$$\binom{N}{k} / 2^N.$$

Следовательно, длина внешнего пути описывается рекуррентным соотношением

$$C_N = N + \frac{1}{2^N} \sum_k \binom{N}{k} (C_k + C_{N-k}).$$

Это рекуррентное соотношение аналогично рекуррентному соотношению быстрой сортировки, которое было решено в разделе 7.2, но решить его значительно труднее. Как ни удивительно, но точным решением является выражение для средних затрат на поиск, полученное на основании леммы 15.3, умноженное на  $N$  (см. упражнение 15.26). Исследование самого рекуррентного соотношения позволяет понять, почему trie-деревья лучше сбалансированы, чем BST-деревья: вероятность того, что разделение произойдет вблизи середины дерева, гораздо выше, чем в любом другом месте. Поэтому рекуррентное соотношение больше напоминает соотношение сортировки слиянием (приблизительное решение которого равно  $N \lg N$ ), нежели рекуррентное соотношение быстрой сортировки (приблизительное решение которого равно  $2 N \lg N$ ).

Неприятное свойство trie-деревьев, также отличающее их от других рассмотренных типов деревьев поиска — однонаправленное ветвление в случае присутствия ключей с одинаковыми разрядами. Например, ключи, которые различаются только в последнем разряде, всегда требуют пути, длина которого равна длине ключа, независимо от количе-



**РИСУНОК 15.8 ХУДШИЙ СЛУЧАЙ БИНАРНОГО TRIE-ДЕРЕВА**

На этих рисунках показан результат вставки ключей  $H=01000$  и  $I=01001$  в первоначально пустое бинарное trie-дерево. Как и в DST-дереве (см. рис. 15.4), длина пути ограничивается длиной двоичного представления ключей; однако, как видно из этого примера, пути могут иметь эту длину даже при наличии в trie-дереве всего двух ключей.

ства ключей в дереве (см. рис. 15.8). Количество внутренних узлов может несколько превышать количество ключей.

**Лемма 15.4** *Trie-дерево, построенное из  $N$  случайных  $w$ -разрядных ключей, содержит в среднем около  $N/\ln 2 \approx 1.44 N$  узлов.*

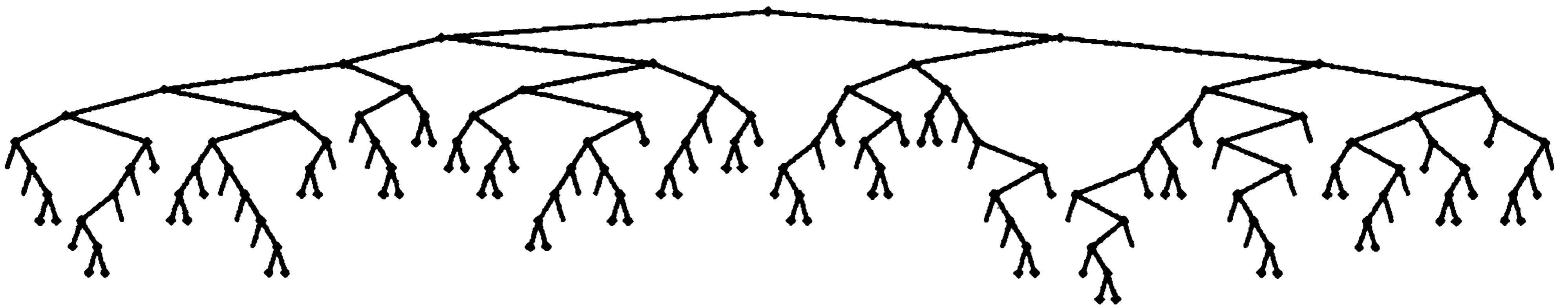
Изменив аргумент для леммы 15.3, можно записать выражение для среднего количества узлов в trie-дереве с  $N$  ключами (см. упражнение 15.27):

$$\sum_{t \geq 0} \left( 2^t \left( 1 - \left( 1 - \frac{1}{2^t} \right)^N \right) - N \left( 1 - \frac{1}{2^t} \right)^{N-1} \right)$$

Математический анализ, позволяющий получить приблизительное значение этой суммы, значительно сложнее, чем приведенный для леммы 15.3, поскольку значения многих членов не равны 0 или 1 (см. раздел ссылок).

Полученный результат можно подтвердить эмпирически. Например, на рис. 15.9 показано большое дерево, имеющее на 44 процента больше узлов, чем DST-дерево или DST-дерево, построенные из этого же набора ключей. Тем не менее, оно хорошо сбалансировано и затраты на поиск в нем почти оптимальны. На первый взгляд могло бы показаться, что дополнительные узлы приведут к существенному повышению средних затрат на поиск, но в действительности это не так — например, средние затраты на поиск увеличились бы всего на 1 при увеличении вдвое количества узлов в сбалансированном trie-дереве.

Для удобства реализации в программах 15.2 и 15.3 предполагалось, что ключи различны и имеют фиксированную длину, дабы иметь уверенность, что рано или поздно ключи окажутся различными, а программы смогут обрабатывать по одному разряду одновременно и никогда не выйдут за пределы разрядов ключей. Для удобства в программах 15.2 и 15.3 также неявно предполагалось, что ключи имеют произвольное количество разрядов, дабы, в конце концов, если пренебречь очень малой (уменьшающейся по экспоненциальному закону) вероятностью, они оказывались различными. Прямым следствием этого допущения является то, что обе программы и их анализ, когда ключами являются строки разрядов переменной длины, требуют нескольких уточнений.



**РИСУНОК 15.9 ПРИМЕР TRIE-ДЕРЕВА**

*Это trie-дерево, построенное в результате вставки около 200 случайных ключей, хорошо сбалансировано, но из-за однонаправленного ветвления содержит на 44 процента больше узлов, чем было бы необходимо в ином случае. (Нулевые связи в листьях не показаны.)*

Для использования программ в приведенном виде в отношении ключей переменной длины ограничение различия ключей следует расширить, чтобы ни один из ключей не был префиксом другого ключа. Как будет показано в разделе 15.5, в некоторых приложениях подобное ограничение достигается автоматически. В противном случае такие ключи можно было бы обрабатывать, сохраняя информацию во внутренних узлах, поскольку каждый обрабатываемый префикс соответствует какому-либо внутреннему узлу в trie-дереве (см. упражнение 15.31).

Для достаточно длинных ключей, состоящих из случайных разрядов, утверждения из лемм 15.2 и 15.3 для среднего случая по-прежнему справедливы. В худшем случае высота trie-деревя по-прежнему ограничивается количеством разрядов в самых длинных ключах. Эти затраты могут оказаться весьма существенными, если ключи имеют очень большую длину и, возможно, определенное сходство, как в случае закодированных символьных данных. В следующих двух разделах рассматриваются методы снижения затрат в trie-деревьях для случая длинных ключей. Один из способов сокращения путей в trie-деревьях сводится к свертыванию однонаправленных ветвей в отдельные связи (изящный и эффективный метод выполнения этой задачи будет приведен в разделе 15.3). Другой способ уменьшения длин путей в trie-деревьях предполагает существование более двух связей для каждого узла; этот подход — тема раздела 15.4.

## Упражнения

- ▷ **15.11** Нарисуйте trie-дерево, образованное в результате вставки элементов с ключами **E A S Y Q U T I O N** в указанном порядке в первоначально пустое trie-дерево.
- 15.12** Что происходит в случае применения программы 15.3 для вставки записи, ключ которой равен какому-либо ключу, уже присутствующему в trie-дереве?
- 15.13** Нарисуйте дерево, образованное в результате вставки элементов с ключами **01010011 00000111 00100001 01010001 11101100 00100001 10010101 01001010** в первоначально пустое trie-дерево.
- 15.14** Эмпирически сравните высоту, количество узлов и длину внутреннего пути trie-деревя, построенного в результате вставки  $N$  случайных 32-разрядных ключей в первоначально пустое trie-дерево, с этими же характеристиками стандартного BST-деревя и RB-деревя (глава 13), построенных из тех же ключей, для  $N = 10^3$ ,  $10^4$ ,  $10^5$  и  $10^6$  (см. упражнение 15.6).
- 15.15** Полностью охарактеризуйте длину внутреннего пути для худшего случая trie-деревя, содержащего  $N$  различных  $w$ -разрядных ключей.
- **15.16** Реализуйте операцию *remove* для таблицы символов, основанной на trie-дереве.
- **15.17** Реализуйте операцию *select* для таблицы символов, основанной на trie-дереве.
- 15.18** Реализуйте операцию *sort* для таблицы символов, основанной на trie-дереве.
- ▷ **15.19** Создайте программу, которая выводит все ключи trie-деревя, имеющие те же начальные  $t$  разрядов, что и заданный искомым ключ.

○ **15.20** Воспользуйтесь конструкцией `union` из C++ для реализации операций *search* и *insert*, в которых применяются trie-деревья с узлами, не являющимися листьями; которые содержат связи, но не содержат элементов; и с листьями, которые содержат элементы, но не содержат связей.

○ **15.21** Воспользуйтесь парой производных классов для реализации операций *search* и *insert*, в которых используются trie-деревья с узлами, не являющимися листьями; которые содержат связи, но не содержат элементов; и с листьями, которые содержат элементы, но не содержат связей.

**15.22** Измените программы 15.3 и 15.2 так, чтобы они сохраняли ключ поиска в машинном регистре и выполняли сдвиг на один разряд для обращения к следующему разряду при перемещении в trie-дереве на уровень вниз.

**15.23** Измените программы 15.3 и 15.2 так, чтобы они поддерживали таблицу  $2^r$  trie-деревьев для фиксированной константы  $r$ , причем первые  $r$  разрядов ключа должны использоваться для индексирования таблицы, а к остальным разрядам ключа применяются стандартные алгоритмы доступа. Это изменение позволяет сэкономить около  $r$  шагов, если только таблица не содержит большого количества нулевых записей.

**15.24** Какое значение следовало бы выбрать для  $r$  в упражнении 15.23 при наличии  $N$  случайных ключей (которые достаточно длинны, чтобы их можно было считать различными)?

**15.25** Создайте программу для вычисления количества узлов в trie-дереве, соответствующих данному набору различных ключей фиксированной длины, путем их сортировки и сравнения соседних ключей в отсортированном списке.

● **15.26** Методом индукции докажите, что

$$N \sum_{t \geq 0} (1 - (1 - 2^{-t})^N)$$

— это решение рекуррентного соотношения наподобие быстрой сортировки, приведенного после леммы 15.3, для длины внешнего пути в случайном trie-дереве.

● **15.27** Из выражения леммы 15.4 получите выражение для среднего количества узлов в случайном trie-дереве.

● **15.28** Создайте программу для вычисления среднего количества узлов в случайном trie-дереве, состоящем из  $N$  узлов, и вывода значения с точностью до  $10^{-3}$ , для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

●● **15.29** Докажите, что высота trie-деревя, построенного из  $N$  случайных строк разрядов, приблизительно равна  $2 \lg N$ . *Совет:* рассмотрите задачу о дне рождения (см. лемму 14.2).

● **15.30** Докажите, что средние затраты на поиск в DST-дереве, построенном из случайных ключей, асимптотически приближаются к  $\lg N$  (см. леммы 15.1 и 15.2).

**15.31** Измените программы 15.2 и 15.3 так, чтобы они обрабатывали строки разрядов переменной длины с единственным ограничением, что в структуре данных не должны храниться записи с дублированными ключами. В частности, решите в рамках этого соглашения возвращать значение  $\text{bit}(\mathbf{v}, \mathbf{d})$  для случая, когда  $\mathbf{d}$  больше длины  $\mathbf{v}$ .

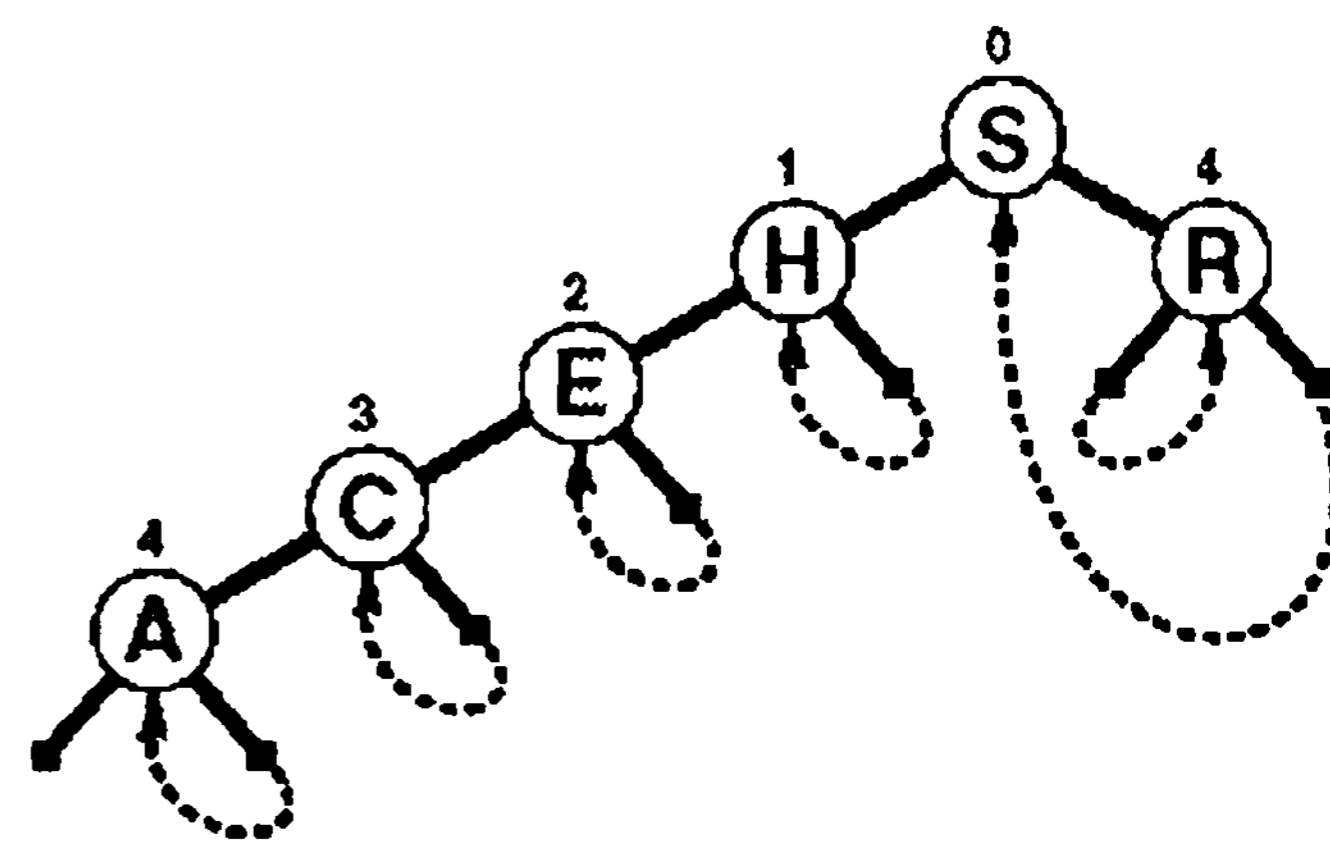


**15.32** Воспользуйтесь trie-деревом для построения структуры данных, которая может поддерживать АДТ таблицы существования для  $w$ -разрядных целых чисел. Программа должна поддерживать операции *construct*, *insert* и *search* при условии, что *insert* и *search* принимают целочисленные аргументы, а *search* возвращает `nullItem.key()` в случае промаха и полученный аргумент в случае попадания при поиске.

## 15.3 patricia-деревья

Основанный на trie-деревьях поиск, описанный в разделе 15.2, обладает двумя недостатками. Во-первых, однонаправленное ветвление приводит к созданию дополнительных узлов в trie-дереве, что кажется необязательным. Во-вторых, в trie-дереве присутствуют два различных типа узлов, что приводит к усложнениям (см. упражнения 15.20 и 15.21). В 1968 г. Моррисон (Morrison) изобрел способ ликвидации обеих проблем за счет применения метода, который назвал *patricia* (practical algorithm to retrieve information coded in alphanumeric — практический алгоритм получения информации, закодированной алфавитно-цифровыми символами). Моррисон разработал свой алгоритм в контексте приложений, использующих индексирование по строкам, наподобие рассмотренных в разделе 15.5, но этот алгоритм не менее эффективен и в плане реализации таблицы символов. Подобно DST-деревьям, patricia-деревья позволяют выполнять поиск  $N$  ключей в дереве, содержащем всего  $N$  узлов; подобно деревьям, они требуют выполнения всего лишь около  $\lg N$  сравнений разрядов и одного сравнения полного ключа для выполнения одного поиска, а также поддерживают другие операции с АДТ. Более того, эти характеристики производительности не зависят от длины ключей, и структура данных подходит для ключей переменной длины.

Начиная со структуры данных стандартного trie-дерева, мы избегаем однонаправленного ветвления благодаря применению простого приема: в каждый узел помещается индекс разряда, который должен проверяться с целью выбора пути из этого узла. Таким образом, мы переходим непосредственно к разряду, в котором должно приниматься важное решение, пропуская сравнения разрядов в узлах, в которых все ключи в поддереве имеют одинаковое значение этого разряда. Более того, внешние узлы исключаются при помощи еще одного простого приема: данные хранятся во внутрен-



**РИСУНОК 15.10**  
**ПОИСК В PATRICIA-ДЕРЕВЕ**

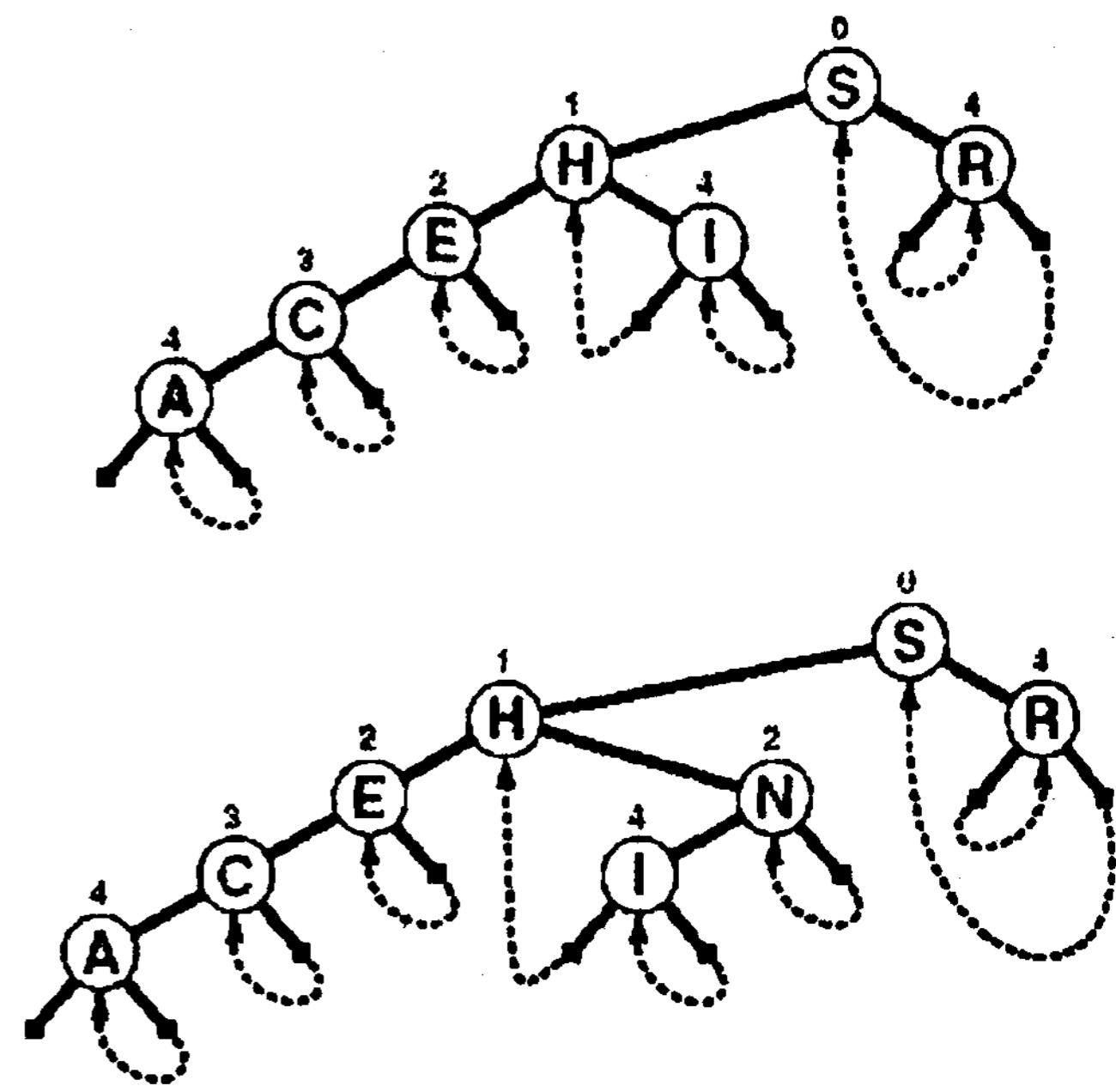
При успешном поиске ключа  $R=10010$  в этом patricia-дереве выполняется перемещение вправо (поскольку нулевой разряд равен 1), затем влево (поскольку 4 разряд равен 0), что приводит к ключу  $R$  (единственному ключу в дереве, начинающемуся с последовательности  $1***0$ ). По пути вниз в дереве выполняется проверка только тех разрядов ключа, которые указаны цифрами над узлами (ключи в узлах игнорируются). При первой встрече связи, которая указывает вверх в дереве, искомым ключ сравнивается с ключом в узле, указанном идущей вверх связью, поскольку это единственный ключ в дереве, который может быть равен искомому ключу.

При безуспешном поиске ключа  $I=01001$  выполняется перемещение влево от корня (поскольку нулевой бит равен 0), затем по правой (направленной вверх) связи (поскольку первый бит равен 1) и выясняется, что ключ  $H$  (единственный ключ в дереве, начинающийся с последовательности  $01$ ) не равен  $I$ .

них узлах, а связи с внешними узлами заменяются связями, которые указывают в обратном направлении вверх на требуемый внутренний узел в trie-дереве. Эти два изменения позволяют представлять trie-деревья как бинарные деревья, состоящие из узлов с ключом и двумя связями (а также дополнительным полем под индексом); такие деревья называются *patricia-деревьями*. При использовании *patricia-деревьев* ключи хранятся в узлах, как при использовании DST-деревьев, а обход дерева выполняется в соответствии с разрядами искомого ключа, но для управления поиском нет необходимости использовать ключи в узлах при перемещении вниз по дереву. Они хранятся там просто для возможного обращения к ним впоследствии, при достижении нижней части дерева.

Как было вскользь отмечено в предыдущем абзаце, отследить работу алгоритма проще, если вначале принять во внимание, что стандартные trie-деревья и *patricia-деревья* можно считать различными представлениями одной и той же абстрактной структуры trie-дерева. Например, trie-деревья, показанные на рис. 15.10 и на верхней схеме рис. 15.11, иллюстрирующие поиск и вставку для *patricia-деревьев*, представляют ту же абстрактную структуру, что и trie-деревья на рис. 15.6. В алгоритмах поиска и вставки для *patricia-деревьев* используется, создается и поддерживается конкретное представление АДТ trie-дерева, отличающееся от используемого в алгоритмах поиска и вставки, которые рассматривались в разделе 15.2. Тем не менее, лежащая в их основе абстракция остается той же самой.

Программа 15.4 содержит реализацию алгоритма поиска в *patricia-дереве*. Используемый метод отличается от поиска в trie-дереве в трех отношениях: не существует никаких явных нулевых связей, в ключе проверяется не следующий разряд, а указанный, и поиск завершается сравнением ключа в точке, в которой выполняется перемещение по дереву вверх. Легко проверить, указывает ли связь вверх, поскольку индексы разрядов в узлах (по определению) увеличиваются по мере перемещения вниз по дереву. При выполнении поиска он начинается от корня и перемещение выполняется вниз по дереву с использованием индекса разряда в каждом узле для определения разряда в искомом ключе, который следует проверять — если этот разряд равен 1, перемещение выполняется вправо, а если 0 — влево. При перемещении вниз по дереву ключи в узлах вообще не проверяются. Со временем встречается связь, указывающая вверх: каждая направленная вверх связь указывает на уникальный ключ в дереве, содержащий разряды, которые могли бы направить поиск вдоль этой связи. Таким образом, если ключ в узле, указанный первой направленной вверх связью, равен искомому ключу, поиск является успешным; в противном случае он будет безуспешным.



**РИСУНОК 15.11 ВСТАВКА В PATRICIA-ДЕРЕВО**

*Чтобы вставить ключ  $I$  в приведенный на рис. 15.10 пример *patricia-дерева*, мы добавляем новый узел для проверки 4 разряда, поскольку ключи  $H=01000$  и  $I=01001$  отличаются только этим разрядом (рисунок сверху). В процессе последующего поиска в trie-дереве, который достигает нового узла, необходимо проверить ключ  $H$  (левую связь), если 4 разряд ключа поиска равен 0; если этот разряд равен 1 (правая связь), следует проверить ключ  $I$ .*

*Для вставки ключа  $N=01110$  (рисунок внизу) мы добавляем новый узел между ключами  $H$  и  $I$  для проверки 2 разряда, поскольку именно этот разряд отличает  $N$  от  $H$  и  $I$ .*

---

**Программа 15.4 Поиск в patricia-дереве**

---

Рекурсивная функция **searchR** возвращает уникальный узел, который может содержать запись с ключом **v**. Она выполняет продвижение вниз по trie-дереву, используя разряды для управления поиском, но проверяет только один разряд каждого встреченного узла — указанный в поле **bit**. Функция прерывает поиск, встретив внешнюю связь, указывающую вверх. Функция поиска **search** вызывает функцию **searchR**, а затем проверяет ключ в этом узле для определения того, имеет ли место попадание и промах при поиске.

**private:**

```
Item searchR(link h, Key v, int d)
{
    if (h->bit <= d) return h->item;
    if (digit(v, h->bit) == 0)
        return searchR(h->l, v, h->bit);
    else return searchR(h->r, v, h->bit);
}
```

**public:**

```
Item search(Key v)
{ Item t = searchR(head, v, -1);
  return (v == t.key()) ? t : nullItem;
}
```

---

На рис. 15.10 показан поиск в patricia-дереве. В случае промаха, обусловленного тем, что поиск направляется по нулевой связи в trie-дереве, соответствующий поиск в patricia-дереве будет следовать несколько иным путем, нежели поиск в стандартном trie-дереве, поскольку при перемещении вниз по дереву разряды, соответствующие однонаправленному ветвлению, вообще не проверяются. В то время как поиск в trie-дереве завершается в листе, поиск в patricia-дереве завершается сравнением с тем же ключом, что и при поиске в trie-дереве, но без проверки разрядов, соответствующих однонаправленному ветвлению в trie-дереве.

Реализация вставки для patricia-деревьев отражает два случая, возникающих при вставке в trie-деревьях (см. рис. 15.11). Как обычно, информация о местоположении нового ключа извлекается из промаха при поиске. При использовании trie-деревьев промах может происходить либо из-за нулевой связи, либо из-за несовпадения ключа в листе. При использовании patricia-деревьев приходится выполнять дополнительные действия для определения требуемого типа вставки, поскольку во время поиска соответствующие однонаправленному ветвлению разряды были пропущены. Поиск в patricia-деревьях всегда завершается сравнением ключа, и этот ключ содержит требуемую информацию. Мы находим самый левый разряд, в котором отличаются искомым ключ и ключ, прервавший поиск, затем снова выполняем поиск в trie-дереве, сравнивая этот разряд с разрядами в узлах по пути поиска. Посещение узла, определяющего более старшую позицию разряда, чем у того, где различаются искомым и найденный ключи, свидетельствует о пропуске разряда во время поиска в patricia-дереве, который должен был бы приводить к нулевой связи при аналогичном поиске, но в trie-дереве. Поэтому мы добавляем новый узел, чтобы обеспечить проверку этого разряда. Если не удастся отыскать узел, определяющий более старшую позицию разряда, чем у того, где различаются искомым и найденный ключи, значит, поиск в patricia-дереве соответствует поиску в trie-дереве, который завершается в листе. В

таким случае мы добавляем новый узел, который различает искомый ключ и ключ, прервавший поиск. Мы всегда добавляем только один узел, ссылающийся на самый левый разряд, в котором ключи отличаются, в то время как при использовании стандартного trie-дерева для достижения этого разряда могло бы потребоваться добавление нескольких узлов с однонаправленными ветвями. Помимо обеспечения требуемого различения разрядов, новый узел будет использоваться также и для хранения нового элемента. Начальный этап построения примера trie-дерева показан на рис. 15.12.

Программа 15.5 содержит реализацию алгоритма вставки в patricia-дерево. Код вытекает непосредственно из описания, приведенного в предыдущем абзаце, с учетом одного дополнительного обстоятельства, что выполняется просмотр связей с узлами, содержащими индексы разрядов, длина которых не превышает индекс текущего разряда и которые служат связями с внешними узлами. Код вставки просто проверяет это свойство связей, но не должен перемещать ключи или связи. На первый взгляд направленные вверх связи в patricia-деревьях кажутся странными, но выбор связей, которые должны использоваться при вставке каждого узла, удивительно прост. Суть же заключается в том, что использование одного типа узла вместо двух существенно упрощает код.

### Программа 15.5 Вставка в patricia-дерево

Процесс вставки ключа в patricia-дерево начинается с поиска. Функция `searchR` из программы 15.5 приводит к уникальному ключу в дереве, который должен отличаться от вставляемого. Мы определяем самый левый разряд, в котором отличаются этот и искомый ключи, а затем при помощи рекурсивной функции `insertR` перемещаемся вниз по дереву и вставляем новый узел, содержащий `v` в этой позиции.

В функции `insertR` имеется два случая, которые соответствуют случаям, проиллюстрированным на рис. 15.11. Новый узел может замещать внутреннюю связь (если ключ поиска отличается от ключа, найденного в позиции разряда, который был пропущен) или внешнюю связь (если разряд, который отличает искомый ключ от найденного, не потребовался для различения найденного ключа от всех других ключей в trie-дереве).

```
private:
```

```
link insertR(link h, Item x, int d, link p)
{ Key v = x.key();
  if ((h->bit >= d) || (h->bit <= p->bit))
  {
    link t = new node(x); t->bit = d;
    t->l = (digit(v, t->bit) ? h : t);
    t->r = (digit(v, t->bit) ? t : h);
    return t;
  }
}
```

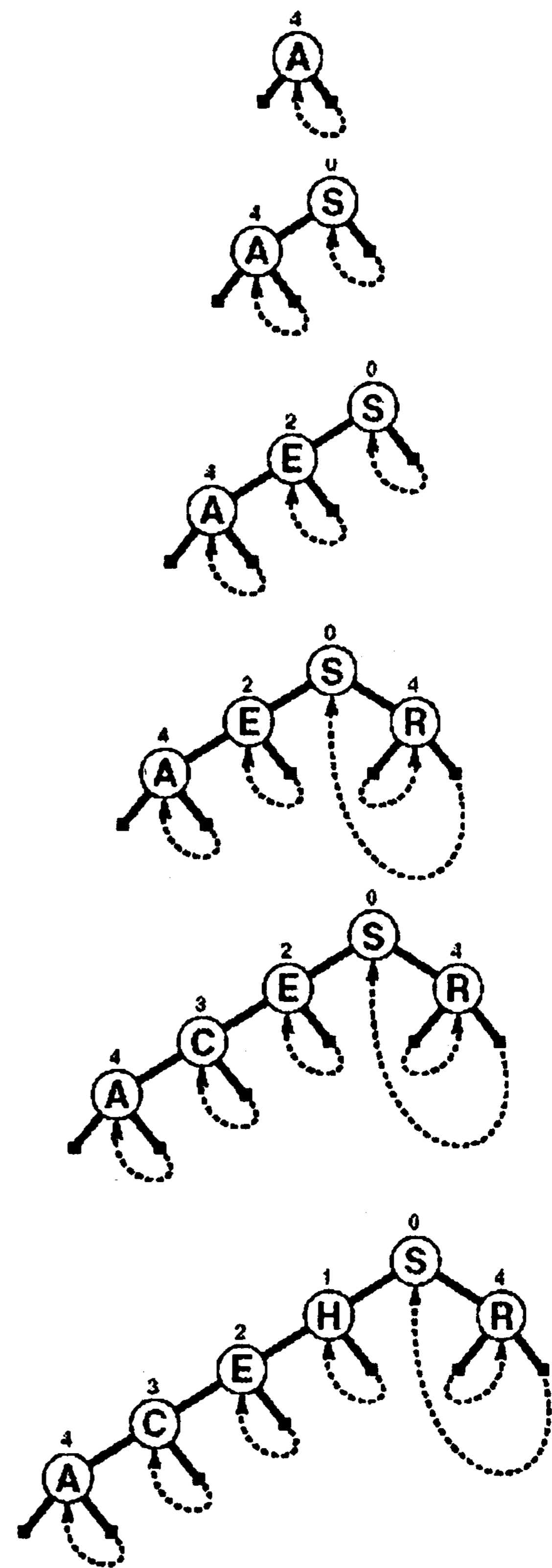


РИСУНОК 15.12 ПОСТРОЕНИЕ PATRICIA-ДЕРЕВА

Эта последовательность рисунков отражает результат вставки ключей *A S E R C H* в первоначально пустое patricia-дерево. На рис. 15.11 находится результат вставки ключей *I* и *N* в дерево, показанное на нижнем рисунке.

```

    if (digit(v, h->bit) == 0)
        h->l = insertR(h->l, x, d, h);
    else h->r = insertR(h->r, x, d, h);
    return h;
}
public:
void insert(Item x)
{ Key v = x.key(); int i;
  Key w = searchR(head->l, v, -1).key();
  if (v == w) return;
  for (i = 0; digit(v, i) == digit(w, i); i++) ;
  head->l = insertR(head->l, x, i, head);
}
ST(int maxN)
{ head = new node(nullItem);
  head->l = head->r = head; }

```

В соответствии с принципом конструирования, все внешние узлы, расположенные ниже узла с индексом разряда  $k$ , начинаются с тех же самых  $k$  разрядов (в противном случае следовало бы создать узел с индексом разряда, который меньше  $k$ , чтобы два узла различались). Следовательно, patricia-дерево можно преобразовать в стандартное trie-дерево, создав соответствующие внутренние узлы между узлами, в которых разряды были пропущены, и заменив указывающие вверх связи на связи с внешними узлами (см. упражнение 15.48). Однако, лемма 15.2 выполняется для patricia-деревьев не полностью, поскольку присваивание ключей внутренним узлам зависит от порядка вставки ключей. Структура внутренних узлов зависит от порядка вставки ключей, а внешние связи и размещение значений ключей — нет.

Важное следствие того, что patricia-дерево представляет лежащую в его основе структуру стандартного trie-дерева, заключается в том, что рекурсивный поперечный обход дерева можно использовать для посещения узлов по порядку, как демонстрируется в программе 15.6. Мы посещаем только внешние узлы, которые выявляются за счет проверки на наличие не увеличивающихся индексов разрядов.

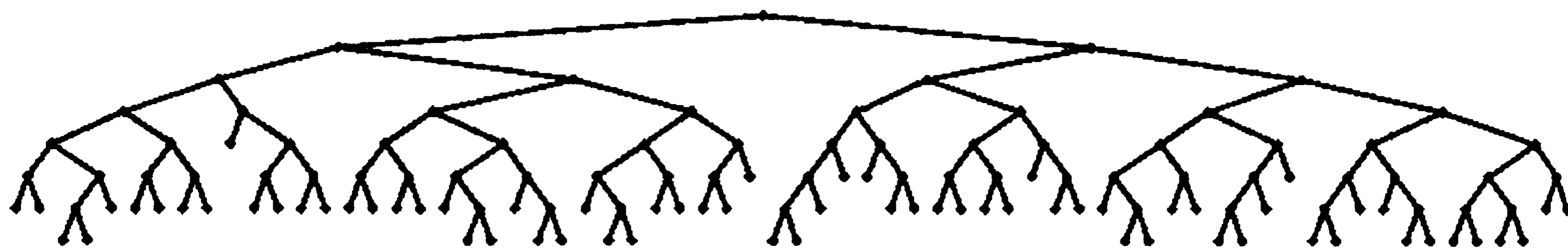
### Программа 15.6 Сортировка в patricia-дереве

Эта рекурсивная процедура отображает записи в patricia-дереве в порядке следования их ключей. В программе предполагается, что элементы располагаются во внешних (виртуальных) узлах, которые могут быть выявлены при помощи проверки того, что индекс разряда в текущем узле не превышает индекс разряда его родительского узла. В остальном эта программа — суть реализация стандартного поперечного обхода.

```

private:
void showR(link h, ostream& os, int d)
{
    if (h->bit <= d) { h->item.show(os); return; }
    showR(h->l, os, h->bit);
    showR(h->r, os, h->bit);
}
public:
void show(ostream& os)
{ showR(head->l, os, -1); }

```



**РИСУНОК 15.13 ПРИМЕР PATRICIA-ДЕРЕВА**

*Это patricia-дерево, построенное в результате вставки примерно 200 случайных ключей, эквивалентно trie-дереву, приведенному на рис. 15.9, при условии удаления из последнего однонаправленных ветвей. Результирующее дерево является почти идеально сбалансированным.*

patricia-деревья — наиболее показательная реализация метода поразрядного поиска: этот метод позволяет идентифицировать разряды, которые отличают ключи поиска, после чего встраивать их в структуру данных (без избыточных узлов), обеспечивающую быстрое попадание от любого искомого ключа к единственному ключу в структуре, который мог бы быть равен искомому. На рис. 15.13 показано patricia-дерево, образованное теми же ключами, которые использовались для построения trie-дерева из рис. 15.9 — patricia-дерево не только содержит на 44 процента меньше узлов по сравнению со стандартным trie-деревом, но и является почти идеально сбалансированным.

**Лемма 15.5** *Вставка или поиск случайного ключа в patricia-дереве, построенном из  $N$  случайных строк разрядов, требует приблизительно  $\lg N$  сравнений разрядов в среднем и приблизительно  $2 \lg N$  сравнений разрядов в худшем случае. Количество сравнений разрядов никогда не превышает длины ключа.*

Эта лемма — непосредственное следствие леммы 15.3, поскольку длина путей в patricia-деревьях не превышает длину путей в соответствующих trie-деревьях. Точный анализ среднего случая patricia-дерева сложен; из него следует, что в среднем в patricia-дереве требуется на одно сравнение меньше, чем в стандартном trie-дереве (см. раздел ссылок).

В табл. 15.1 приведены экспериментальные данные, подтверждающие вывод, что DST-деревья, стандартные trie-деревья и patricia-деревья обеспечивают сравнимую производительность (а также обеспечивают время поиска, которое сравнимо или меньше времени поиска методов с использованием сбалансированных деревьев из главы 13) в случае целочисленных ключей. Поэтому данные методы определено должны рассматриваться в качестве возможных реализаций таблиц символов даже при использовании ключей, которые могут быть представлены в виде коротких строк разрядов, с учетом ряда упомянутых вполне очевидных ограничений.

#### **Таблица 15.1 Экспериментальные результаты исследования реализаций trie-деревьев**

Эти сравнительные значения времени построения и поиска в таблицах символов, содержащих случайные последовательности 32-разрядных целых чисел, подтверждают, что поразрядные методы конкурируют с методами, использующими сбалансированные деревья, даже в случае ключей, хранящих случайные разряды. Различия в производительности более заметны, когда ключи являются длинными и не обязательно случайными (см. табл. 15.2), или когда особое внимание уделяется обеспечению эффективности доступа к разрядам ключа (см. упражнение 15.22).

N	конструирование				попадания при поиске			
	B	D	T	P	B	D	T	P
1250	1	1	1	1	0	1	1	0
2500	2	2	4	3	1	1	2	1
5000	4	5	7	7	3	2	3	2
12500	18	15	20	18	8	7	9	7
25000	40	36	44	41	20	17	20	17
50000	81	80	99	90	43	41	47	36
100000	176	167	269	242	103	85	101	92
200000	411	360	544	448	228	179	211	182

Ключ:

- B RB-дерево бинарного поиска (программы 12.8 и 13.6)
- D DST-дерево (программа 15.1)
- T trie-дерево (программы 15.2 и 15.3)
- P patricia-дерево (программы 15.4 и 15.5)

Обратите внимание, что затраты на поиск, упомянутые в лемме 15.5, *не* возрастают с увеличением длины ключа. И напротив, затраты на поиск при использовании стандартного trie-дерева, как правило, зависят от длины ключей — позиция первого разряда, в котором различаются два заданных ключа, может находиться на произвольном удалении от начала ключа. Все рассмотренные ранее методы поиска, основанные на сравнениях, также зависят от длины ключа — если два ключа различаются только самым правым разрядом, для их сравнения требуется время, пропорциональное длине ключей.

Более того, для выполнения поиска методами хеширования *всегда* требуется время, пропорциональное длине ключа, поскольку требуется вычислять хеш-функцию. Однако метод patricia-деревьев позволяет обращаться непосредственно к интересующим разрядам и, как правило, требует проверки менее  $\lg N$  из них. В связи с этим patricia-метод (или поиск с использованием trie-дерева, из которого удалены однонаправленные ветвления) — наиболее предпочтительный метод поиска при наличии длинных ключей.

Например, предположим, что используется компьютер, обеспечивающий эффективный доступ к 8-разрядным байтам данных, и требуется выполнять поиск среди миллионов 1000-разрядных ключей. В этом случае при использовании patricia-метода для выполнения поиска потребовался бы доступ только к приблизительно 20 байтам искомого ключа плюс одна 125-байтовая операция сравнения. В то же время при использовании хеширования потребовался бы доступ ко всем 125 байтам искомого ключа для вычисления хеш-функции плюс несколько операций сравнения, а основанные на сравнениях методы потребовали бы от 20 до 30 сравнений полных ключей. Действительно, сравнения ключей, в особенности на ранних этапах поиска, требуют сравнения всего нескольких байтов, но на последующих этапах, как правило, необходимо сравнение значительно большего количества байтов. В разделе 15.5 мы снова сравним производительность различных методов поиска для случая длинных ключей.

Действительно, для *patricia*-алгоритма вообще не должно существовать никаких ограничений на длину искомым ключей. Как будет показано в разделе 15.5, этот алгоритм особенно эффективен в приложениях с ключами переменной длины, которые могут быть очень длинными. При использовании *patricia*-деревьев можно надеяться, что количество проверок разрядов, необходимых для поиска среди  $N$  записей, даже с очень длинными ключами, будет приблизительно пропорциональным  $\lg N$ .

## Упражнения

- 15.33** Что происходит при использовании программы 15.5 для вставки записи, ключ которой равен какому-либо ключу, уже присутствующему в *trie*-дереве?
- ▷ **15.34** Нарисуйте *patricia*-дерево, образованное в результате вставки ключей **E A S Y Q U T I O N** в указанном порядке в первоначально пустое *trie*-дерево.
- ▷ **15.35** Нарисуйте *patricia*-дерево, образованное в результате вставки ключей **01010011 00000111 00100001 01010001 11101100 00100001 10010101 01001010** в указанном порядке в первоначально пустое *trie*-дерево.
- **15.36** Нарисуйте *patricia*-дерево, образованное в результате вставки ключей **01010011 00000111 00100001 11101100 01010001 00100001 00000111 01010011** в указанном порядке в первоначально пустое *trie*-дерево.
- 15.37** Экспериментально сравните высоту и длину внутреннего пути *patricia*-дерева, построенного в результате вставки  $N$  случайных 32-разрядных ключей в первоначально пустое *trie*-дерево, с этими же характеристиками стандартного *BST*-дерева и *RB*-дерева (глава 13), образованных из этих же ключей, для  $N = 10^3, 10^4, 10^5$  и  $10^6$  (см. упражнения 15.6 и 15.14).
- 15.38** Приведите полную характеристику длины внутреннего пути для худшего случая *patricia*-дерева, содержащего  $N$  различных  $w$ -разрядных ключей.
- ▷ **15.39** Реализуйте операцию *select* для таблицы символов, основанной на *patricia*-дереве.
- **15.40** Реализуйте операцию *remove* для таблицы символов, основанной на *patricia*-дереве.
- **15.41** Реализуйте операцию *join* для таблицы символов, основанной на *patricia*-дереве.
- **15.42** Создайте программу, которая выводит все ключи *patricia*-дерева, имеющие такие же начальные  $t$  разрядов, как и у заданного искомого ключа.
- 15.43** Измените стандартные поиск и вставку с использованием *trie*-дерева (программы 15.2 и 15.3) с целью исключения однонаправленного ветвления подобно тому, как это делается для *patricia*-деревьев. Если вы уже выполнили упражнение 15.20, воспользуйтесь результирующей программой в качестве отправной точки.
- 15.44** Измените поиск и вставку с использованием *patricia*-дерева (программы 15.4 и 15.5) для поддержки таблицы  $2^r$  *trie*-деревьев, как описано в упражнении 15.23.
- 15.45** Покажите, что каждый ключ в *patricia*-дереве находится в собственном пути поиска и, следовательно, во время выполнения операции *search* встречается как по пути вниз по дереву, так и в конце операции.



**15.46** Измените программу поиска с использованием *patricia*-дерева (программа 15.4), чтобы в процессе перемещения вниз по дереву она сравнивала ключи с целью повышения производительности обнаружения попаданий при поиске. Экспериментально оцените эффективность произведенного изменения (см. упражнение 15.45).

**15.47** Воспользуйтесь *patricia*-деревом для построения структуры данных, которая может поддерживать АДТ таблицы существования для  $w$ -разрядных целых чисел (см. упражнение 15.32).

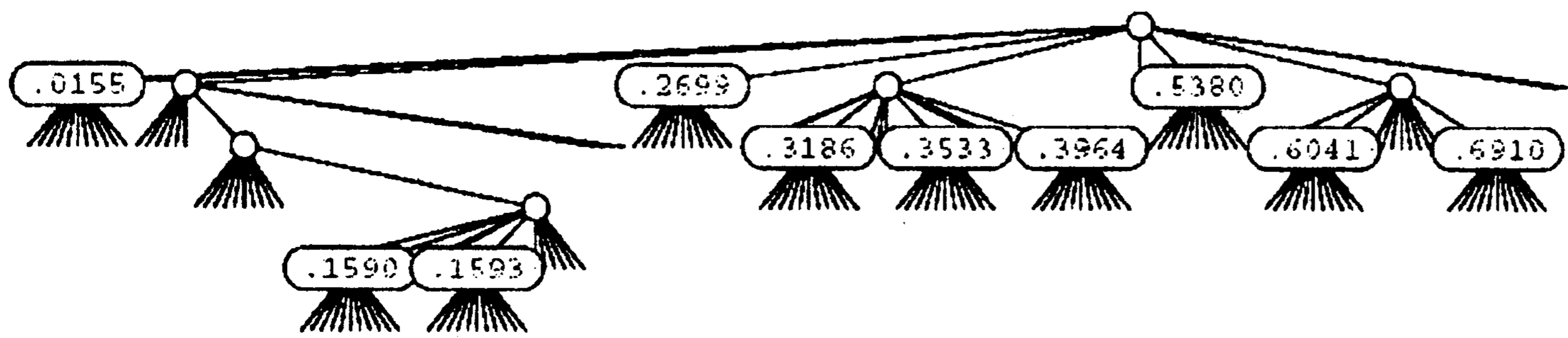
- **15.48** Создайте программу, которая преобразует *patricia*-дерево в стандартное *trie*-дерево с такими же ключами, и наоборот.

## 15.4 Многопутевые *trie*-деревья и TST-деревья

Было установлено, что производительность поразрядной сортировки можно существенно увеличить, рассматривая одновременно более одного разряда. То же самое справедливо и в отношении поразрядного поиска: исследуя одновременно по  $r$  разрядов, скорость поиска можно увеличить в  $r$  раз. Однако, существует скрытая опасность, вынуждающая применять эту идею более осторожно, чем в случае поразрядной сортировки. Проблема заключается в том, что одновременное рассмотрение  $r$  разрядов соответствует использованию узлов дерева с  $R=2^r$  связями, а это может приводить к значительным напрасным затратам памяти для неиспользуемых связей.

В *trie*-деревьях (бинарных), описанных в разделе 15.2, узлы, соответствующие разрядам ключей, имеют две связи: одну для случая, когда разряд ключа равен 0, и вторую для случая, когда он равен 1. Подходящим обобщением служат  $R$ -путевые *trie*-деревья, когда цифрам ключа соответствуют узлы с  $R$  связями, по одной для каждого возможного значения цифры. Ключи хранятся в листьях (узлах, все связи которых являются нулевыми). Поиск в  $R$ -путевом *trie*-дереве начинается с корня и с самой левой цифры ключа, а цифры ключа используются для управления перемещением вниз по дереву. Если значение цифры равно  $i$ , выполняется перемещение вниз по  $i$ -той связи (и переход на следующую цифру). В случае достижения листа, он содержит единственный ключ в *trie*-дереве, ведущие цифры которого соответствуют пройденному пути, поэтому для определения того, имеет ли место попадание или промах при поиске, можно сравнить этот ключ с искомым. В случае достижения нулевой связи понятно, что имеет место промах при поиске, поскольку эта связь соответствует последовательности ведущих цифр, не найденной ни в одном ключе *trie*-дерева. На рис. 15.14 показано 10-путевое *trie*-дерево, представляющее тестовый набор десятичных чисел. Как отмечалось в главе 10, встречающиеся на практике числа обычно различаются сравнительно небольшим количеством узлов *trie*-дерева. Эта же особенность более общих типов ключей служит основой для ряда эффективных алгоритмов поиска.

Прежде чем создавать реализацию полной таблицы символов с несколькими типами узлов, давайте начнем изучение многопутевых деревьев с задачи *таблицы существования*, в которой хранятся только ключи (без каких-либо записей и связанной с ними информации). Требуется разработать алгоритмы *вставки* ключа в структуру данных и *поиска* в структуре данных с целью определения, был ли вставлен заданный ключ.



Чтобы использовать тот же интерфейс, что и для более общих реализаций таблиц символов, давайте примем соглашение, что функция поиска возвращает `nullItem` при промахе и фиктивный элемент, содержащий искомый ключ, при попадании. Это соглашение способствует упрощению кода и наглядному представлению структуры многопутевых trie-деревьев. В разделе 15.5 исследуются более общие реализации таблиц символов, в том числе индексация строк.

**Определение 15.2** *trie-дерево существования, соответствующее набору ключей, рекурсивно определяется следующим образом: trie-дерево для пустого набора ключей — нулевая связь; trie-дерево для непустого набора ключей — внутренний узел, содержащий связи, ссылающиеся на trie-дерево для каждой возможной цифры ключа, причем при построении поддеревьев ведущая цифра должна удаляться.*

Для простоты в этом определении предполагается, что ни один ключ не является префиксом другого. Как правило, удовлетворение такого ограничения достигается тем, что все ключи различны и либо имеют фиксированную длину, либо содержат завершающий символ со значением `NULLdigit` — служебный символ, который не используется ни для каких других целей. Основное в этом определении то, что trie-деревья существования можно применять для реализации таблиц существования без сохранения внутри trie-дерева какой-либо информации. Вся информация неявно определяется внутри структуры trie-дерева. Каждый узел содержит  $R+1$  связь (по одной для каждого возможного значения символа плюс одну связь для `NULLdigit`) и не содержит никакой другой информации. Для управления перемещением вниз по trie-дереву во время поиска используются цифры из ключа. Если связь с `NULLdigit` встречается одновременно с завершением цифр ключа, то имеет место попадание при поиске, в противном случае имеет место промах. Для вставки нового ключа поиск выполняется до тех пор, пока не встретится нулевая связь, а затем добавляются узлы для каждого из остальных символов в ключе. На рис. 15.15 показан пример 27-путевого trie-дерева; программа 15.7 содержит реализацию процедур поиска и вставки в базовом (многопутевом) trie-дереве существования.

**РИСУНОК 15.14**  
10-ПУТЕВОЕ  
TRIE-ДЕРЕВО ДЛЯ  
ДЕСЯТИЧНЫХ ЧИСЕЛ.

*На этом рисунке показано trie-дерево, которое позволяет различать набор чисел*

.396465048  
.353336658  
.318693642  
.015583409  
.159369371  
.691004885  
.899854354  
.159072306  
.604144269  
.269971047  
.538069659

*(см. рис. 12.1). Каждый узел имеет 10 связей (по одной для каждой возможной цифры). В корне связь 0 указывает на trie-дерево для ключей, первая цифра которых равна 0 (присутствует только одно такое дерево); связь 1 указывает на trie-дерево для ключей, первая цифра которых — 1 (таких деревьев два), и т.д. Ни одно из этих чисел не имеет первой цифры, равной 4, 7, 8 или 9, поэтому данные связи являются нулевыми. В дереве присутствует только по одному числу, первая цифра которого равна 0, 2 и 5, поэтому для каждой из этих цифр имеется лист, содержащий соответствующее число. Остальная часть структуры строится рекурсивно за счет смещения на одну цифру вправо.*

### Программа 15.7 Поиск и вставка в trie-дерево существования

В этой реализации операций *search* и *insert* для многопутевых trie-деревьев ключи неявно сохраняются внутри структуры trie-дерева. Каждый узел содержит R указателей на следующий, более низкий уровень trie-дерева. Когда *t*-той цифрой ключа будет *i*, перемещение выполняется вдоль *i*-той связи на уровне *t*. Функция *search* возвращает фиктивный элемент, содержащий переданный в аргументе ключ, если он присутствует в таблице, или *nullItem* в противном случае. В качестве альтернативы можно было бы изменить интерфейс, чтобы в нем использовался только тип **Key**, или в созданном нами классе элементов реализовать преобразование типа из **Item** в **Key**.

```
private:
    struct node
    {
        node **next;
        node()
        {
            next = new node*[R];
            for (int i = 0; i < R; i++) next[i] = 0;
        }
    };
    typedef node *link;
    link head;
    Item searchR(link h, Key v, int d)
    {
        int i = digit(v, d);
        if (h == 0) return nullItem;
        if (i == NULLdigit)
            { Item dummy(v); return dummy; }
        return searchR(h->next[i], v, d+1);
    }
    void insertR(link& h, Item x, int d)
    {
        int i = digit(x.key(), d);
        if (h == 0) h = new node;
        if (i == NULLdigit) return;
        insertR(h->next[i], x, d+1);
    }
public:
    ST(int maxN)
    {
        head = 0;
    }
    Item search(Key v)
    {
        return searchR(head, v, 0);
    }
    void insert(Item x)
    {
        insertR(head, x, 0);
    }

```

Если ключи различны и имеют фиксированную длину, можно обратиться к связи с конечным символом и прекращать поиск по достижении длины ключа (см. упражнение 15.55). Мы уже встречались с примером подобного типа trie-дерева, когда использовали trie-деревья для описания сортировки ключей фиксированной длины сначала по самому старшему разряду (MSD).

В определенном смысле это чисто абстрактное представление структуры trie-дерева является оптимальным, поскольку оно может поддерживать выполнение опера-

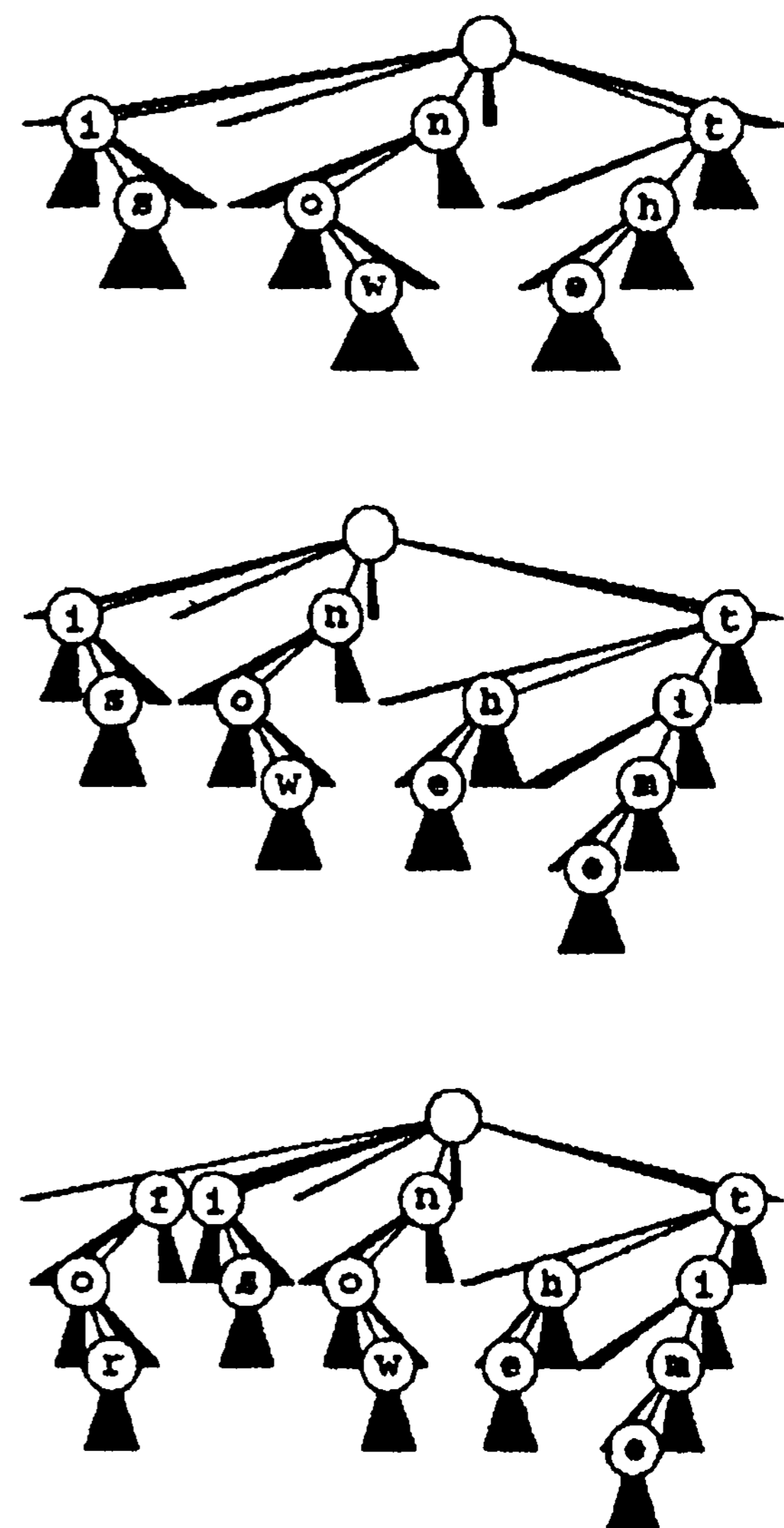


РИСУНОК 15.15 ПОИСК И ВСТАВКА В R-ПУТЕВОМ TRIE-ДЕРЕВЕ СУЩЕСТВОВАНИЯ

26-путевое trie-дерево для слов *now*, *is* и *the* (рисунок сверху) имеет девять узлов: корень плюс по одному узлу для каждой буквы. На этих рисунках узлы помечены, но в структурах данных явные метки узлов не используются, поскольку метка каждого узла может быть получена, исходя из позиции его связи в массиве связей родительского узла. Для вставки ключа *time* в существующем узле для *t* создается новая ветвь и добавляются новые узлы для ключей *i*, *m* и *e* (рисунок в центре); для вставки ключа *for* выполняется переход от корня и добавляются новые узлы для *f*, *o* и *r*.

ции *search* за время, которое пропорционально длине ключа, при затратах памяти, в худшем случае пропорциональных общему количеству символов в ключе. Однако общий объем используемой памяти может оказаться весьма большим, поскольку для каждого символа имеется около  $R$  связей. В конечном итоге необходимы более эффективные реализации. Как было показано для случая бинарных trie-деревьев, в качестве конкретного представления абстрактной структуры, которая хорошо определяет используемый набор ключей, имеет смысл рассматривать чистое trie-дерево. Далее можно исследовать и другие представления той же абстрактной структуры, которые, возможно, обеспечат более высокие показатели производительности.

**Определение 15.3** *Многопутевое trie-дерево* — это многопутевое дерево, имеющее связанные с каждым из его листьев ключи, которое рекурсивно определяется следующим образом: trie-дерево для пустого набора ключей представляет собой нулевую связь; trie-дерево для единственного ключа — лист, содержащий этот ключ; и, наконец, trie-дерево для набора ключей, количество которых значительно превышает 1 — внутренний узел со связями, ссылающимися на trie-деревья для ключей с каждым из возможных значений цифр, причем для конструирования поддеревьев ведущая цифра должна удаляться.

Предполагается, что ключи в структуре данных различны и ни один ключ не является префиксом другого. При выполнении поиска в стандартном многопутевом trie-дереве цифры ключа используются для направления поиска вниз по дереву. При этом возможны три исхода. Если достигнута нулевая связь, значит, имеет место промах при поиске; если достигнут лист, содержащий ключ поиска, имеет место попадание; если достигнут лист, содержащий другой ключ, имеет место промах при поиске. Все листья имеют  $R$  нулевых связей, следовательно, как упоминалось в разделе 15.2, узлы-листья и узлы, не являющиеся листьями, могут быть представлены различным образом. Такая реализация рассматривается в главе 16, а в этой главе предлагается другой подход к реализации. В любом случае аналитические результаты, полученные в разделе 15.3, являются достаточно общими, чтобы дать представление о характеристиках производительности стандартных многопутевых деревьев.

**Лемма 15.6.** *Для выполнения поиска или вставки в стандартном  $R$ -арном trie-дереве в среднем требуется выполнение приблизительно  $\log_R N$  сравнений байтов в дереве, построенном из  $N$  случайных строк байтов. Количество связей в  $R$ -арном trie-дереве, построенном из  $N$  случайных ключей, приблизительно равно  $RN/\ln R$ . Количество сравнений байтов, необходимое для выполнения поиска или сравнения, не превышает количества байтов в искомом ключе.*

Эти результаты — ни что иное, как обобщение утверждений, приведенных в леммах 15.3 и 15.4. Их можно получить, подставив в доказательствах этих лемм  $R$  вместо 2. Однако, как упоминалось ранее, для выполнения точного математического анализа требуются исключительно сложные математические выкладки.

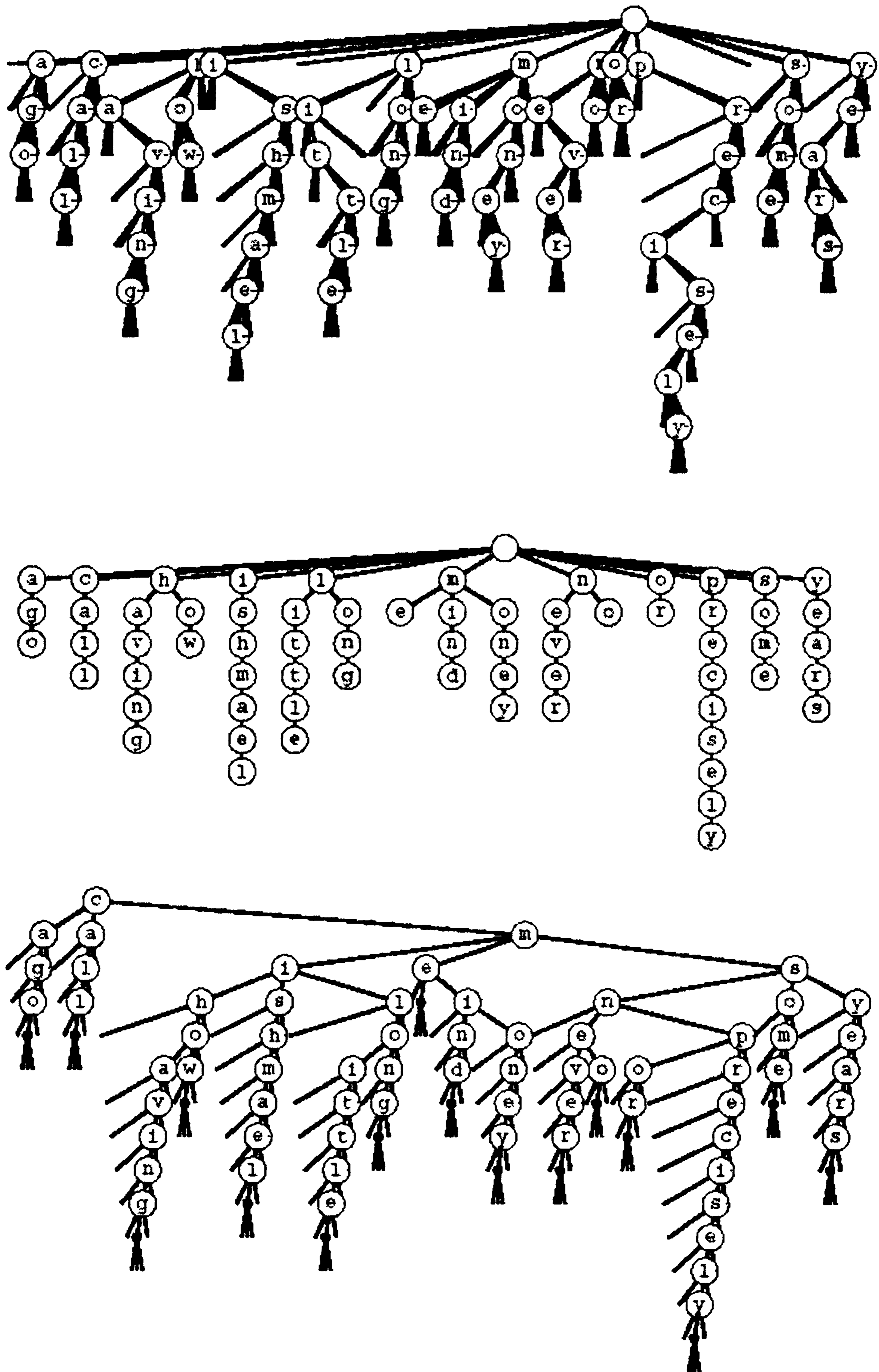
Характеристики производительности, указанные в лемме 15.6, представляют крайний случай компромисса между временем выполнения и занимаемым объемом памяти. С одной стороны, имеется большое количество неиспользуемых нулевых связей — лишь несколько узлов вблизи вершины дерева используют более одной-двух из своих связей. С другой стороны, высота дерева невелика. Предположим, например,

что используется типичное значение  $R=256$  и имеется  $N$  случайных 64-разрядных ключей. В соответствии с леммой 15.6 для выполнения поиска потребуется  $(\lg N)/8$  сравнений символов (максимум 8) и при этом будет задействовано менее  $47N$  связей. Если объем доступной памяти не ограничен, этот метод предоставляет весьма эффективную альтернативу. Для этого примера затраты на выполнение поиска можно было бы сократить до 4 сравнений символов, приняв  $R=65536$ , однако при этом потребовалось бы свыше 5900 связей.

В разделе 15.5 мы вернемся к стандартным многопутевым деревьям. В остальной части этого раздела рассматривается альтернативное представление trie-деревьев, построенных программой 15.7: *trie-дерево тернарного поиска (ternary search trie — TST)*, или просто TST-дерево, полная форма которого показана на рис. 15.16. В TST-дереве каждый узел содержит символ и *три* связи, соответствующие ключам, текущие цифры которых меньше, равны и больше символа узла. Подход эквивалентен реализации узлов trie-дерева в виде BST-деревьев, в который в качестве ключей используются символы, соответствующие ненулевым связям. В стандартных trie-деревьях существования из программы 15.7 узлы trie-дерева представляются  $R+1$  связью, и выводы о символе, представленном каждой ненулевой связью, делаются на основании его индекса. В соответствующем TST-дереве существования все символы, соответствующие ненулевым связям, явно появляются в узлах — мы находим символы, соответствующие ключам, только при прохождении по средним связям.

Алгоритм поиска для TST-деревьев существования столь прост, что читатели вполне могли бы описать его самостоятельно. Алгоритм вставки несколько сложнее, но прямо отражает вставку в trie-деревьях существования. Для выполнения поиска первый символ в ключе сравнивается с символом в корне. Если он меньше, поиск продолжается по левой связи, если больше — по правой, а если равен, поиск выполняется вдоль средней связи и осуществляется переход к следующему символу ключа. В любом случае алгоритм применяется рекурсивно. Поиск завершается промахом, если встречается нулевая связь или конец искомого ключа встречается раньше, чем **NULLdigit**. Поиск завершается попаданием, если пересекается средняя связь в узле, символ которого — **NULLdigit**. Для вставки нового ключа выполняется поиск, а затем добавляются новые узлы для символов в заключительной части ключа, как это имело место в trie-деревьях. Подробности реализации этих алгоритмов приведены в программе 15.8, а на рис. 15.17 показаны TST-деревья, соответствующие trie-деревьям из рис. 15.15.

Продолжая использовать соответствие между деревьями поиска и алгоритмами сортировки, мы видим, что TST-деревья соответствуют трехпутевой сортировке, точно так же как BST-деревья соответствуют быстрой сортировке, trie-деревья — бинарной быстрой сортировке, а *M-путевые* trie-деревья — *M-путевой* сортировке. Структура рекурсивных вызовов для трехпутевой сортировки, показанная на рис. 10.13, представляет собой TST-дерево для этого набора ключей. Проблема нулевых связей, характерная для trie-деревьев, похожа на проблему пустых корзин, характерную для поразрядной сортировки; трехпутевое ветвление обеспечивает эффективное решение обеих проблем.



**РИСУНОК 15.16 СТРУКТУРЫ TRIE-ДЕРЕВЬЕВ СУЩЕСТВОВАНИЯ**

На этих рисунках показаны три различных реализации trie-дерева существования для 16 слов *call me ishmael some years ago never mind how long precisely having little or no money*: 26-путьное trie-дерево существования (вверху); абстрактное trie-дерево с удаленными нулевыми связями (в центре); представление в виде TST-дерева (внизу). 26-путьное trie-дерево содержит слишком много связей, в то время как TST-дерево служит эффективным представлением абстрактного trie-дерева. В двух верхних trie-деревьях предполагается, что ни один ключ не является префиксом другого. Например, добавление ключа *pot* привело бы к потере ключа *po*. Для решения этой проблемы в конце каждого ключа можно добавить нулевой символ, как показано на примере TST-дерева на нижнем рисунке.

### Программа 15.8 Поиск и вставка в TST-дерево существования

Это код реализует те же алгоритмы абстрактного trie-дерева, что и в программе 15.7, но каждый узел содержит одну цифру и три связи: по одной для ключей, следующая цифра которых меньше, равна и больше соответствующей цифры в искомом ключе.

```
private:
struct node
{ Item item; int d; node *l, *m, *r;
  node(int k)
  { d = k; l = 0; m = 0; r = 0; }
};
typedef node *link;
link head;
Item nullItem;
Item searchR(link h, Key v, int d)
{ int i = digit(v, d);
  if (h == 0) return nullItem;
  if (i == NULLdigit)
  { Item dummy(v); return dummy; }
  if (i < h->d) return searchR(h->l, v, d);
  if (i == h->d) return searchR(h->m, v, d+1);
  if (i > h->d) return searchR(h->r, v, d);
}
void insertR(link& h, Item x, int d)
{ int i = digit(x.key(), d);
  if (h == 0) h = new node(i);
  if (i == NULLdigit) return;
  if (i < h->d) insertR(h->l, x, d);
  if (i == h->d) insertR(h->m, x, d+1);
  if (i > h->d) insertR(h->r, x, d);
}
public:
ST(int maxN)
{ head = 0; }
Item search(Key v)
{ return searchR(head, v, 0); }
void insert(Item x)
{ insertR(head, x, 0); }
```

Эффективность TST-деревьев в плане используемого объема памяти можно повысить, помещая ключи в листья в тех точках, где они различны, и избегая однопутевого ветвления между внутренними узлами, как это имело место в patricia-деревьях. В конце этого раздела исследуется реализация, основанная на первом из указанных изменений.

**Лемма 15.7** Для выполнения поиска или вставки в полное TST-дерево требуется время, пропорциональное длине ключа. Количество связей в TST-дереве превышает количество символов во всех ключах не более чем в три раза.

В худшем случае каждый символ ключа соответствует полному несбалансированному  $R$ -арному узлу, вытянутому наподобие односвязного списка. Вероятность возникновения этого худшего случая в случайном дереве исключительно мала. Скорее можно ожидать, что придется выполнить  $\ln R$  или менее сравнений на первом уровне (поскольку корневой узел ведет себя подобно BST-дереву, состояще-

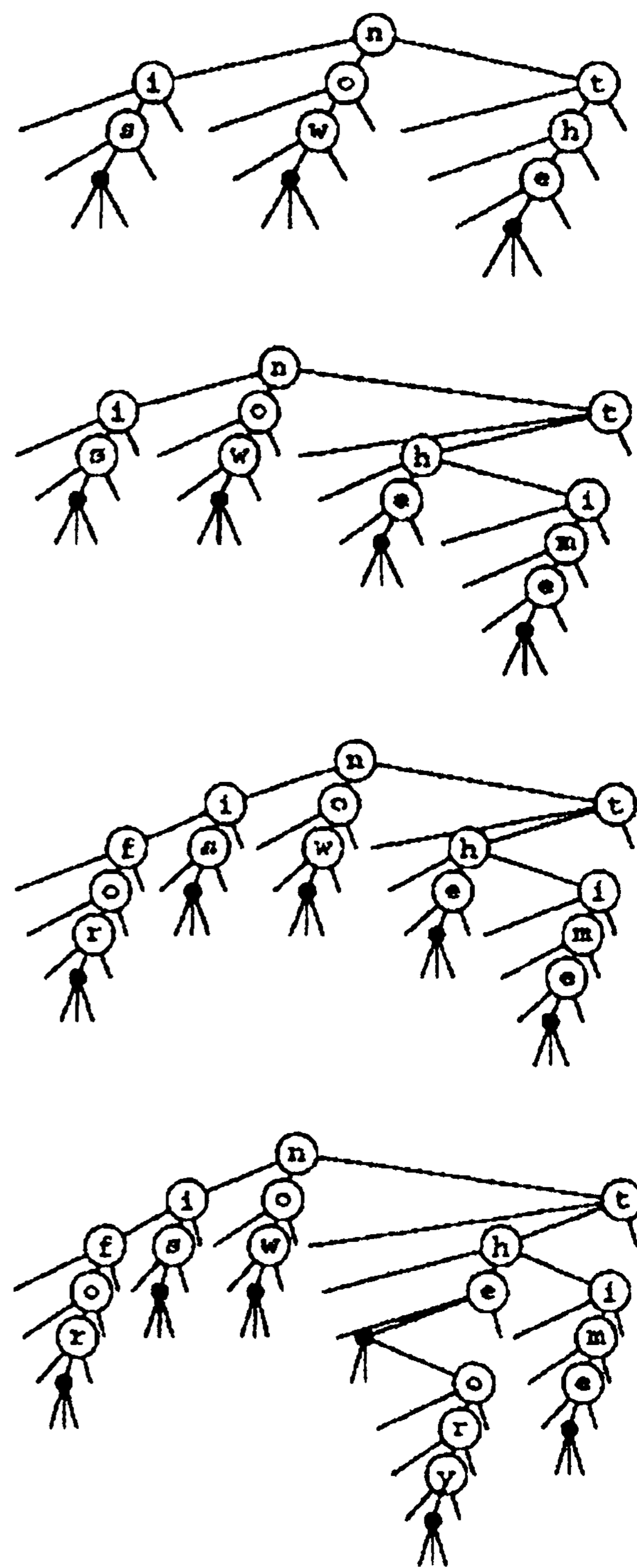


РИСУНОК 15.17 TST-ДЕРЕВЬЯ СУЩЕСТВОВАНИЯ

TST-дерево существования содержит по одному узлу для каждой буквы, но каждый узел имеет только 3 дочерних узла, а не 26. Деревья на трех верхних рисунках — это TST-деревья, соответствующие примеру вставки из рис. 15.15, за исключением того, что к каждому ключу дописывается завершающий символ. Это позволяет снять ограничение, связанное с тем, что ни один ключ не может быть префиксом другого. В таком случае можно, например, вставить ключ *theory* (рисунок внизу).

му из  $R$  различных значений байтов) и, возможно, на нескольких других уровнях (если существуют ключи с общим префиксом и содержащие до  $R$  различных значений байтов в символе, который следует за префиксом). При этом для большинства символов придется выполнять лишь несколько сравнений байтов (поскольку большинство узлов trie-дерева редко содержат ненулевые связи). Для обнаружения промахов при поиске, вероятнее всего, потребуется незначительное количество сравнений байтов, завершающихся на нулевой связи в одном из верхних уровней дерева. Для обнаружения попаданий при поиске будет требоваться приблизительно по одному сравнению байта для одного символа ключа поиска, поскольку большинство из них расположено в узлах с однопутевым ветвлением в нижней части trie-дерева.

В общем случае фактически используемый объем памяти меньше верхнего предельного значения, определяемого тремя связями на каждый символ, поскольку ключи совместно используют узлы в верхних уровнях дерева. Мы воздержимся от проведения точного анализа для среднего случая, поскольку TST-деревья наиболее полезны в ситуациях, когда ключи не являются ни случайными, ни полученными из надуманных конструкций, соответствующих худшему случаю.

Главное достоинство использования TST-деревьев заключается в том, что они легко приспособляются к неоднородностям в ключах, возникновение которых весьма вероятно в реальных приложениях. Это является следствием двух основных эффектов. Во-первых, ключи в реальных приложениях образуются из больших наборов символов, а использование конкретных символов в наборах далеко от однородного — например, в конкретном наборе строк, скорее всего, будет использоваться только небольшая часть возможных символов. При использовании TST-деревьев можно задействовать 128- или 256-символьное кодирование, не беспокоясь о лишние затратах для узлов с 128- или 256-путевым ветвлением и не будучи вынужденными определять, какие наборы символов действительно имеют значение. Наборы символов алфавитов, отличных от латинского, могут содержать тысячи символов — TST-деревья особенно подходят для строковых ключей, состоящих из таких символов. Во-вторых, ключи в практических приложениях часто имеют структурированный формат, различающийся от приложения к приложению, когда в одной части ключа используются только буквы, в другой — только цифры, а специальные символы служат разделителями (см. упражнение 15.72). Например, на рис. 15.18 приведен список кодов для базы данных онлайн библиотеки. В случае таких ключей некоторые из узлов trie-дерева могут быть представлены унарными узлами в TST-дереве (для мест, где все ключи содержат разделители), другие могут быть представлены 10-узловыми BST-деревьями (для мест, где все ключи содержат цифры),

```
LDS___361_H_4
LDS___485_N_4_H_317
LDS___625_D_73_1986
LJN___679_N_48_1985
LQP___425_M_56_1991
LTK___6015_P_63_1988
LVM___455_M_67_1974
WAFR_____5054_____33
WKG_____6875
WLSOC_____2542_____30
WPHIL_____4060_____2_____55
WPHYS_____39_____1_____30
WROM_____5350_____65_____5
WUS_____10706_____7_____10
WUS_____12692_____4_____27
```

#### РИСУНОК 15.18 ПРИМЕР СТРОКОВЫХ КЛЮЧЕЙ (НОМЕРОВ БИБЛИОТЕЧНЫХ ФУНКЦИЙ)

*Эти ключи из онлайн-базы данных библиотеки, иллюстрируют степень варьирования структуры, которую можно встретить в строковых ключах в приложениях. Некоторые символы могут быть смоделированы случайными буквами, другие — случайными цифрами, а третьи имеют фиксированное значение или структуру.*



а третьи — 26-узловыми BST-деревьями (для мест, где все ключи содержат буквы). Эта структура создается автоматически, не требуя специального анализа ключей.

Второе практическое преимущество поиска, основанного на TST-деревьях, по сравнению со множеством других алгоритмов заключается в том, что обнаружение промахов при поиске, скорее всего, будет исключительно эффективным даже при длинных ключах. Часто для обнаружения промаха при поиске в алгоритме используется лишь несколько сравнений байтов (и поддерживается несколько указателей). Как было показано в разделе 15.3, для обнаружения промаха при поиске в хеш-таблице, которая содержит  $N$  ключей, требуется время, пропорциональное длине ключа (для вычисления хеш-функции); для выполнения той же операции в дереве поиска требуется, по меньшей мере,  $\lg N$  сравнений ключей. Даже в Patricia-дереве для обнаружения случайного промаха при поиске требуется  $\lg N$  сравнений разрядов.

В табл. 15.2 приведены экспериментальные данные, подтверждающие выводы, приведенные в двух предыдущих абзацах.

**Таблица 15.2 Экспериментальные данные исследования поиска с использованием строковых ключей**

Эти сравнительные значения времени построения и поиска в таблицах символов, образованных строковыми ключами, наподобие библиотечных кодов из рис. 15.18, подтверждают, что TST-деревья, хотя и требуют несколько больших затрат при построении, обеспечивают наиболее быстрое обнаружение промахов при поиске с использованием строковых ключей. В основном это обусловлено тем, что для поиска не требуется исследование всех символов в ключе.

$N$	конструирование				промахи при поиске			
	В	Н	Т	Т*	В	Н	Т	Т*
1250	4	4	5	5	2	2	2	1
2500	8	7	10	9	5	5	3	2
5000	19	16	21	20	10	8	6	4
12500	48	48	54	97	29	27	15	14
25000	118	99	188	156	67	59	36	30
50000	230	191	333	255	137	113	70	65

Ключ:

- В** Стандартное BST-дерево (программа 12.8)
- Н** Хеширование с отдельным связыванием ( $M = N/5$ ) (программа 14.3)
- Т** TST-дерево (программа 15.8)
- Т\*** TST-дерево с  $R^2$ -путевым ветвлением в корне (программы 15.11 и 15.12)

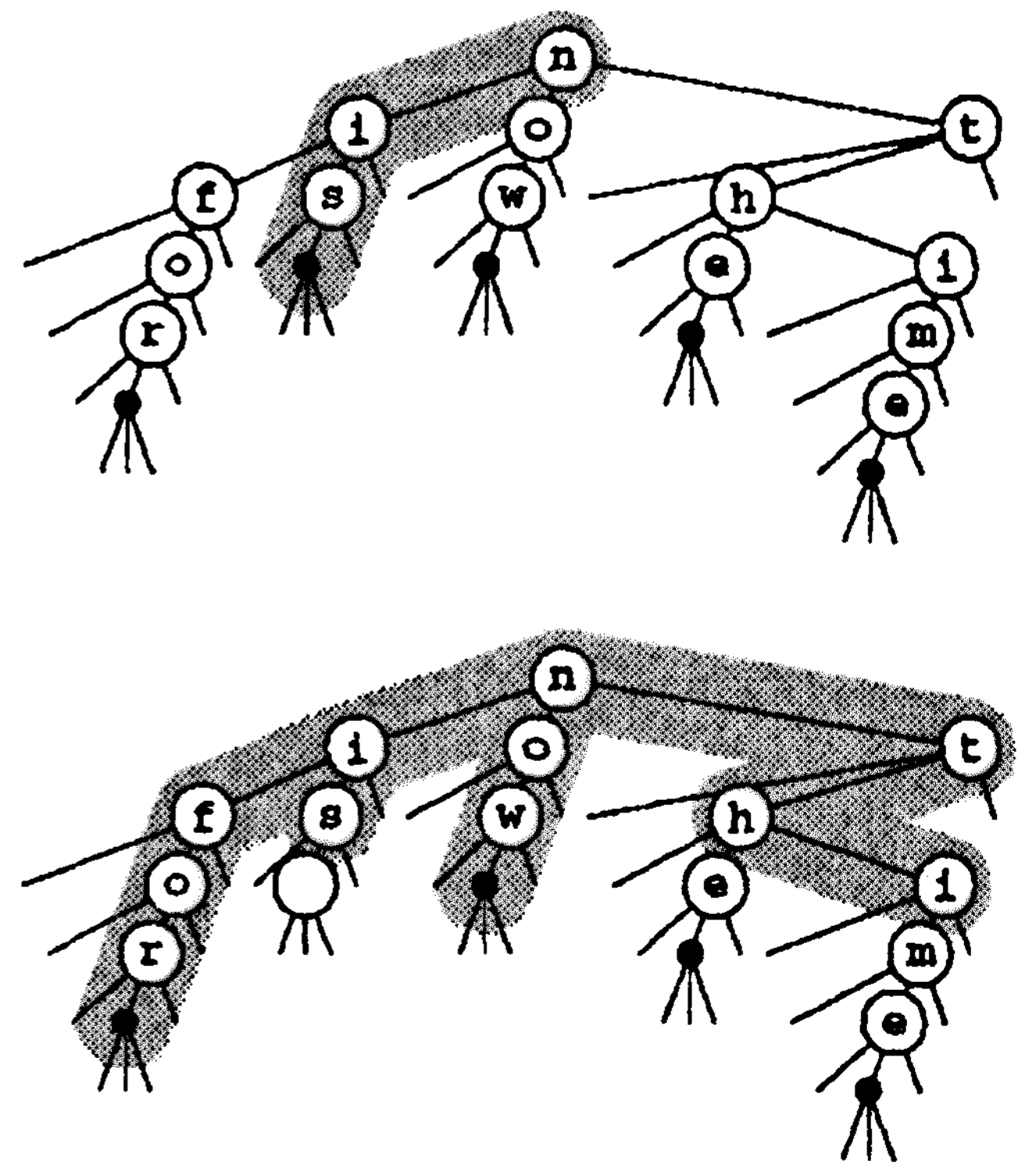
Третья причина привлекательности TST-деревьев заключается в том, что они поддерживают более общие операции, нежели рассмотренные операции таблиц символов. Например, программа 15.9 разрешает не указывать отдельные символы в искомом ключе и выводит все ключи в структуре данных, которые соответствуют указанным цифрам ключа поиска. Подобный пример показан на рис. 15.19. Очевидно, что за счет внесения небольших изменений, эту программу можно приспособить к посещению всех соответствующих ключей, как это делается для выполнения операции *sort*, а не просто для их вывода (см. упражнение 15.58).

С помощью TST-деревьев легко решается еще несколько аналогичных задач. Например, можно посетить все ключи в структуре данных, которые отличаются от ключа поиска не более чем в одной позиции цифры (см. упражнение 15.59). Операции этого типа требуют больших затрат или невозможны в случае применения других реализаций таблиц символов. Эти и многие другие задачи, когда не требуется строгое соответствие со строкой поиска, будут подробно рассматриваться в части 5.

Patricia-деревья предоставляют несколько аналогичных преимуществ; основное практическое преимущество TST-деревьев по сравнению с patricia-деревьями заключается в том, что они обеспечивают доступ к байтам, а не к разрядам ключей. Одна из причин, почему это различие рассматривается как преимущество, связана с тем, что предназначенные для этого машинные операции реализованы во многих компьютерах, а C++ обеспечивает непосредственный доступ к байтам символьных строк в стиле C. Другая причина состоит в том, что в некоторых приложениях работа с байтами в структуре данных естественным образом отражает байтовую ориентацию самих данных в некоторых приложениях — например, при решении задачи поиска частичного соответствия, описанной в предыдущем абзаце (хотя, как будет показано в главе 18, поиск частичного соответствия можно ускорить за счет продуманного использования доступа к разряду).

Исходя из желания избежать однопутевого ветвления в TST-деревьях, стоит заметить, что большинство однопутевых ветвлений происходит на концах ключей и не происходит, если развить структуру до стандартной реализации многопутевого trie-дерева, в которой записи хранятся в листьях, помещенных в самом верхнем уровне trie-дерева различения ключей. Можно также поддерживать индексацию байтов, подобно тому как это делается в patricia-деревьях (см. упражнение 15.65), однако для простоты мы опускаем это изменение. Комбинация многопутевого ветвления и представления в виде TST-дерева и сама по себе достаточно эффективна во многих приложениях, но свертывание однопутевого ветвления в стиле patricia-деревьев еще больше повышается производительность в тех случаях, когда ключи, скорее всего, совпадают с длинными последовательностями (см. упражнение 15.72).

Еще одно простое усовершенствование поиска, основанного на использовании TST-деревьев, — использование большого явного многопутевого узла в корне. Для этого проще всего хранить таблицу  $R$  TST-деревьев: по одному для каждого возможного значения первой буквы в ключах. Если значение  $R$  невелико, можно использовать первые две буквы ключей (и таблицу размером  $R^2$ ). Чтобы этот метод был эф-



**РИСУНОК 15.19 ПОИСК ЧАСТИЧНОГО СООТВЕТСТВИЯ В TST-ДЕРЕВЬЯХ**

Для нахождения всех ключей в TST-дереве, которые соответствуют шаблону  $i^*$  (верхний рисунок), мы выполняем поиск  $i$  в BST-дереве для первого символа. В данном примере мы находим слово  $is$  (единственное слово, соответствующее шаблону), пройдя две однопутевых ветви. Для поиска в соответствии с менее строгим шаблоном  $*o^*$  (нижний рисунок) в BST-дереве первого символа мы посещаем все узлы, но в дереве второго символа только те, которые соответствуют символу  $o$ , со временем выходя на слова  $for$  и  $pow$ .

эффективен, ведущие цифры ключей должны быть равномерно распределенными. Результирующий гибридный алгоритм поиска соответствует тому, как человек мог бы искать фамилии в телефонном справочнике. Вначале принимается многопутевое решение ("Так, посмотрим, слово начинается с 'А'"), вслед за чем, вероятно, принимается несколько двухпутевых решений ("Оно должно располагаться перед 'Andrews', но после 'Aitken'"), после чего сравнивается следующий символ ("Algonquin", ...Нет, 'Algoritms' отсутствует, поскольку ни одно слово не начинается с 'Algor!").

Программы 15.10–15.12 включают в себя основанную на применении TST-дерева реализацию операций *search* и *insert*, в которой используется *R*-путевое ветвление в корне и хранение элементов в листьях (чтобы не было однопутевых ветвей, если ключи различны). Скорее всего, эти программы будут относиться к наиболее быстрым программам выполнения поиска с использованием строковых ключей. Лежащая в их основе структура TST-дерева может поддерживать также набор других операций.

### Программа 15.10 Определения типов узлов в гибридном TST-дереве

Этот код определяет структуры данных, связанные с программами 15.11 и 15.12, которые предназначены для реализации таблицы символов с использованием TST-деревьев. В корневом узле используется *R*-путевое ветвление: корень — это массив **heads**, состоящий из *R* связей и индексированный по первым цифрам ключей. Каждая связь указывает на TST-дерево, построенное из всех ключей, которые начинаются с соответствующих цифр. Этот гибрид сочетает в себе преимущества trie-деревьев (быстрый поиск при помощи индексации, реализуемый в корне) и TST-деревьев (эффективное использование памяти, обусловленное существованием по одному узлу для каждого символа, не считая корня).

```
struct node
{ Item item; int d; node *l, *m, *r;
  node(Item x, int k)
  { item = x; d = k; l = 0; m = 0; r = 0; }
  node(node* h, int k)
  { d = k; l = 0; m = h; r = 0; }
  int internal()
  { return d != NULLdigit; }
};
typedef node *link;
link heads[R];
Item nullItem;
```

### Программа 15.11 Вставка в гибридное TST-дерево для АДТ таблицы символов

В этой реализации операции *insert* с использованием TST-деревьев элементы хранятся в листьях, что, по существу, является обобщением программы 15.3. В ней *R*-путевое ветвление используется для первого символа, а отдельные TST-деревья — для всех слов, начинающихся с каждого символа. Если поиск завершается на нулевой связи, мы создаем лист для хранения элемента. Если поиск завершается в листе, мы создаем внутренние узлы, необходимые для различения найденного и искомого ключей.

```
private:
link split(link p, link q, int d)
{ int pd = digit(p->item.key(), d),
  qd = digit(q->item.key(), d);
  link t = new node(nullItem, qd);
```

```

    if (pd < qd)
        { t->m = q; t->l = new node(p, pd); }
    if (pd == qd)
        { t->m = split(p, q, d+1); }
    if (pd > qd)
        { t->m = q; t->r = new node(p, pd); }
    return t;
}
link newext(Item x)
{ return new node(x, NULLdigit); }
void insertR(link& h, Item x, int d)
{ int i = digit(x.key(), d);
  if (h == 0)
    { h = new node(newext(x), i); return; }
  if (!h->internal())
    { h = split(newext(x), h, d); return; }
  if (i < h->d) insertR(h->l, x, d);
  if (i == h->d) insertR(h->m, x, d+1);
  if (i > h->d) insertR(h->r, x, d);
}
public:
  ST(int maxN)
  { for (int i = 0; i < R; i++) heads[i] = 0; }
  void insert(Item x)
  { insertR(heads[digit(x.key(), 0)], x, 1); }

```

---

### Программа 15.12 Поиск в гибридном TST-дереве для АД таблицы символов

---

Эта реализация операции *search* для TST-деревьев (построенных с помощью программы 15.11) подобна поиску с применением многопутевого trie-дерева, но в ней используются только три, а не  $R$  связей для каждого узла (за исключением корня). Цифры ключа используются для перемещения вниз по дереву, которое завершается либо на нулевой связи (промах при поиске), либо в листе, содержащем ключ, который либо равен (попадание при поиске), либо не равен (промах при поиске) искомому.

```

private:
  Item searchR(link h, Key v, int d)
  { if (h == 0) return nullItem;
    if (h->internal())
      { int i = digit(v, d), k = h->d;
        if (i < k) return searchR(h->l, v, d);
        if (i == k) return searchR(h->m, v, d+1);
        if (i > k) return searchR(h->r, v, d);
      }
    if (v == h->item.key()) return h->item;
    return nullItem;
  }
public:
  Item search(Key v)
  { return searchR(heads[digit(v, 0)], v, 1); }

```

---

В таблице символов, которая разрастается до очень больших размеров, может потребоваться согласовать коэффициент ветвления с размером таблицы. В главе 16 будет показан систематический способ увеличения многопутевого trie-дерева, чтобы

можно было воспользоваться преимуществами многопутевого поиска при произвольных размерах файлов.

**Лемма 15.8** Для выполнения поиска или вставки в TST-дереве, содержащем элементы в листьях (т.е., не имеющем ни одной однопутевой ветви в нижней части дерева) и  $R'$ -путевые ветви в корне, требуется приблизительно  $\ln N - t \ln R$  обращений к байтам для  $N$  ключей, которые являются случайными строками байтов. При этом количество требуемых связей равно  $R'$  (для корневого узла) плюс небольшое постоянное число, кратное  $N$ .

Эти грубые оценки непосредственно следуют из леммы 15.6. При оценке затрат времени мы принимаем, что все узлы в пути поиска, за исключением нескольких узлов у вершины, количество которых постоянно, действуют по отношению к  $R$  значениям символов подобно рандомизированным BST-деревьям. Поэтому мы просто умножаем значение времени на  $\ln R$ . При оценке затрат памяти предполагается, что узлы на нескольких первых уровнях заполнены  $R$  значениями символов, а узлы на нижних уровнях содержат только постоянное количество значений символов.

Например, при наличии 1 миллиарда случайных ключей, представляющих собой строки байтов, при  $R = 256$  и при использовании на верхнем уровне таблицы, размер которой равен  $R^2 = 65536$ , для выполнения типового поиска потребуется около  $\ln 10^9 - 2 \ln 256 \approx 20.7 - 11.1 = 9.6$  сравнений байтов. Использование таблицы в верхней части структуры уменьшает затраты на поиск в два раза. Если ключи являются действительно случайными, этой производительности можно достичь с помощью более непосредственных алгоритмов, в которых используются ведущие байты ключа и таблица существования, как было описано в разделе 14.6. В случае применения TST-деревьев такую же производительность можно получить и тогда, когда ключи имеют менее случайную структуру.

Интересно сравнить TST-деревья без многопутевых ветвей в корне со стандартными BST-деревьями при использовании случайных ключей. В соответствии с леммой 15.8, для выполнения поиска в TST-дереве потребуется около  $\ln N$  сравнений байтов, в то время как в стандартных BST-деревьях требуется около  $\ln N$  сравнений ключей. В верхней части BST-дерева сравнения ключей могут быть выполнены путем сравнения всего одного байта, но в нижней части для выполнения сравнения ключа может потребоваться несколько сравнений байтов. Это различие в производительности не является решающим. Причины, по которым при использовании строковых ключей TST-деревья предпочтительнее стандартных BST-деревьев, таковы: они обеспечивают быстрое обнаружение промаха при поиске; они непосредственно приспособлены для многопутевого ветвления в корне; и (что наиболее важно) они хорошо подходят для ключей в виде строк байтов, не являющихся случайными, поэтому длина поиска не превышает длину ключа в TST-дереве.

Некоторые приложения не могут воспользоваться преимуществом  $R$ -путевого ветвления в корне — например, все ключи в примере с библиотечными кодами из рис. 15.18 начинаются с буквы L или W. Для других приложений может потребоваться более высокий коэффициент ветвления в корне — например, как отмечалось, если бы ключи были случайными целыми числами, пришлось бы использовать максимально большую таблицу. Подобные зависимости от приложения можно задействовать при на-

стройке алгоритма с целью достижения максимальной производительности, но не следует забывать о том, что одно из наиболее привлекательных свойств TST-деревьев — возможность не беспокоиться о подобной зависимости от приложений при обеспечении достаточно высокой производительности без каких-либо настроек.

Вероятно, наиболее важное свойство trie-деревьев или TST-деревьев с записями в листьях заключается в том, что их характеристики производительности *не зависят* от длины ключа. Следовательно, их можно использовать для ключей произвольной длины. В разделе 15.5 рассмотрено одно такое особенно эффективное приложение.

## Упражнения

- ▷ **15.49** Нарисуйте trie-дерево существования, образованное в результате вставки слов **now is the time for all good people to come the aid of their party** в первоначально пустое trie-дерево. Используйте 27-путевое ветвление.
- ▷ **15.50** Нарисуйте TST-дерево существования, образованное в результате вставки слов **now is the time for all good people to come the aid of their party** в первоначально пустое TST-дерево.
- ▷ **15.51** Нарисуйте 4-путевое trie-дерево, образованное в результате вставки элементов с ключами **01010011 00000111 00100001 01010001 11101100 00100001 10010101 01001010** в первоначально пустое trie-дерево, в котором используются 2-разрядные байты.
- ▷ **15.52** Нарисуйте TST-дерево, образованное в результате вставки элементов с ключами **01010011 00000111 00100001 01010001 11101100 00100001 10010101 01001010** в первоначально пустое TST-дерево, в котором используются 2-разрядные байты.
- ▷ **15.53** Нарисуйте TST-дерево, образованное в результате вставки элементов с ключами **01010011 00000111 00100001 01010001 11101100 00100001 10010101 01001010** в первоначально пустое TST-дерево, в котором используются 4-разрядные байты.
- **15.54** Нарисуйте TST-дерево, образованное в результате вставки элементов с ключами библиотечных кодов из рис. 15.18 в первоначально пустое TST-дерево.
- **15.55** Измените приведенную реализацию поиска и вставки в многопутевом trie-дереве (программа 15.7), чтобы она работала при условии, что все ключи (фиксированной длины) являются *w*-байтовыми словами (т.е. не требуется указание конца ключа).
- **15.56** Измените приведенную реализацию поиска и вставки в TST-дереве (программа 15.8), чтобы она работала при условии, что все ключи (фиксированной длины) являются *w*-байтовыми словами (т.е. не требуется указание конца ключа).
- 15.57** Экспериментально сравните время и объем памяти, требуемые для 8-путевого trie-дерева, построенного из случайных целых чисел с использованием 3-разрядных байтов, для 4-позиционного trie-дерева, построенного из случайных целых чисел с использованием 2-разрядных байтов, и для бинарного trie-дерева, построенного из тех же ключей, при  $N = 10^3, 10^4, 10^5$  и  $10^6$  (см. упражнение 15.14).
- 15.58** Измените программу 15.9, чтобы подобно операции *sort* она обеспечивала посещение всех узлов, которые соответствуют искомому ключу.

- **15.59** Создайте функцию, которая выводит все ключи в TST-дереве, отличающиеся от искомого не более чем в  $k$  позициях для заданного целочисленного значения  $k$ .
- **15.60** Приведите полную характеристику худшего случая длины внутреннего пути  $R$ -путевого trie-дерева с  $N$  различными  $w$ -разрядными ключами.
- **15.61** Разработайте реализацию таблицы символов с использованием многопутевых trie-деревьев, которая включает в себя деструктор, конструктор копирования и перегруженную операцию присваивания, а также поддерживает операции *construct*, *count*, *insert*, *remove* и *join* для АДТ первого класса таблицы символов с поддержкой дескрипторов клиента (см. упражнения 12.6 и 12.7).
- **15.62** Разработайте реализацию таблицы символов с использованием многопутевых TST-деревьев, которая включает в себя деструктор, конструктор копирования и перегруженную операцию присваивания, а также поддерживает операции *construct*, *count*, *insert*, *remove* и *join* для АДТ первого класса таблицы символов с поддержкой дескрипторов клиента (см. упражнения 12.6 и 12.7).
- ▷ **15.63** Создайте программу, которая выводит все ключи в  $R$ -путевом trie-дереве, имеющие те же начальные  $t$  байтов, что и заданный искомым ключ.
- **15.64** Измените приведенную реализацию поиска и вставки в многопутевом trie-дереве (программа 15.7), чтобы исключить однопутевое ветвление, как это было сделано для patricia-деревьев.
- **15.65** Измените приведенную реализацию поиска и вставки в TST-дереве (программа 15.8), чтобы исключить однопутевое ветвление, как это было сделано для patricia-деревьев.
- 15.66** Создайте программу, которая выполняет балансировку BST-деревьев, представляющих внутренние узлы TST-дерева (реорганизует их так, чтобы все их внешние узлы располагались на одном из двух уровней).
- 15.67** Создайте версию операции *insert* для TST-деревьев, которая поддерживает представление всех внутренних узлов в виде сбалансированного дерева (см. упражнение 15.66).
- **15.68** Приведите полную характеристику худшего случая длины внутреннего пути TST-дерева, содержащего  $N$  различных  $w$ -разрядных ключей.
- 15.69** Создайте программу, генерирующую случайные 80-байтовые строковые ключи (см. упражнение 10.19). Воспользуйтесь этим генератором ключей для построения 256-путевого trie-дерева, содержащего  $N$  случайных ключей при  $N = 10^3$ ,  $10^4$ ,  $10^5$  и  $10^6$ , с применением операции *search*, а затем — операции *insert* в случае промаха при поиске. Программа должна выводить общее количество узлов в каждом trie-дереве и общее время, затраченное на построение каждого trie-дерева.
- 15.70** Выполните упражнение 15.69 для TST-деревьев. Сравните полученные характеристики производительности с характеристиками trie-деревьев.
- 15.71** Создайте генератор, который генерирует ключи, перемешивая случайную 80-байтовую последовательность (см. упражнение 10.21). Воспользуйтесь полученным генератором ключей для построения 256-путевого trie-дерева, содержащего  $N$  случайных ключей при  $N = 10^3$ ,  $10^4$ ,  $10^5$  и  $10^6$ , с применением операции *search*, а затем — операции *insert* в случае промаха при поиске. Сравните полученные ха-

рактеристики производительности с характеристиками trie-дерева из упражнения 15.69.

- 15.72 Создайте генератор ключей, который генерирует 30-байтовые случайные строки, состоящие из трех полей: 4-байтового поля, содержащего одну из 10 заданных строк, 10-байтового поля, содержащего одну из 50 заданных строк, 1-байтового поля, содержащего одно из двух заданных значений, и 15-байтового поля, содержащего случайные, выровненные по левому краю буквенные строки, длина которых с равной вероятностью может составлять от четырех до 15 символов (см. упражнение 10.23). Воспользуйтесь этим генератором ключей для построения 256-путевого trie-дерева, содержащего  $N$  случайных ключей при  $N = 10^3, 10^4, 10^5$  и  $10^6$ , с применением операции *search*, а затем — операции *insert* в случае промаха при поиске. Программа должна выводить общее количество узлов в каждом trie-дереве и общее время, затраченное на построение каждого trie-дерева. Сравните полученные характеристики производительности с характеристиками для случайных строковых ключей (см. упражнение 15.69).

15.73 Выполните упражнение 15.72 для случая TST-деревьев. Сравните полученные характеристики производительности с характеристиками для trie-деревьев.

15.74 Разработайте реализацию операций *search* и *insert* для ключей в виде строк байтов при использовании многопутевых деревьев *порядкового* поиска.

- ▷ 15.75 Нарисуйте 27-путевое DST-дерево (см. упражнение 15.74), образованное в результате вставки элементов с ключами **now is the time for all good people to come the aid of their party** в первоначально пустое DST-дерево.
- 15.76 Разработайте реализацию поиска и вставки в многопутевое trie-дерево, в котором связные списки применяются для представления узлов trie-дерева (в отличие от представления в виде BST-дерева, которое используется для TST-деревьев). Определите экспериментальным путем, что использовать эффективнее: упорядоченные или неупорядоченные списки, и сравните эту реализацию с реализацией, основанной на использовании TST-деревьев.

## 15.5 Алгоритмы индексирования текстовых строк

В разделе 12.7 был рассмотрен процесс построения *строкового индекса*, и BST-дерево со строковыми указателями использовалось для обеспечения возможности определения того, присутствует ли строка с заданным ключом в очень большом тексте. В этом разделе будут рассмотрены более сложные алгоритмы использования многопутевых trie-деревьев, но отправная точка при этом остается той же. Каждая позиция в тексте считается началом строкового ключа, который простирается до конца текста; за счет использования строковых указателей из этих ключей строится таблица символов. Все ключи различны (например, имеют различную длину) и большинство из них очень велики. Цель поиска — определить, является ли заданный искомый ключ префиксом одного из ключей в индексном указателе, что эквивалентно выяснению того, присутствует ли искомый ключ где-либо в текстовой строке.

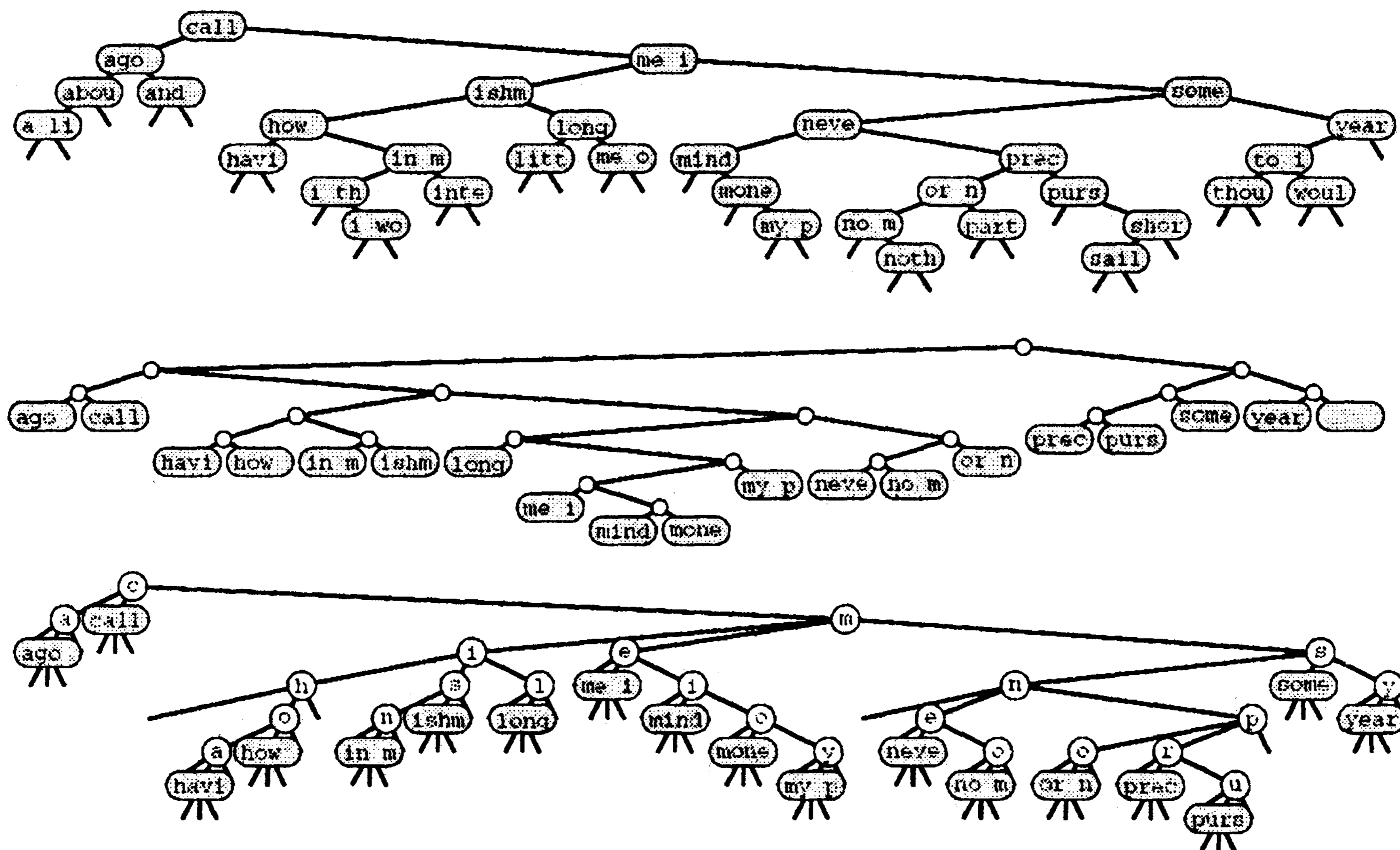
Дерево поиска, которое построено из ключей, определенных строковыми указателями внутри текстовой строки, называется *деревом суффиксов*. Можно воспользоваться любым алгоритмом, допускающим существование ключей переменной длины. В этом случае особенно подходят методы, основанные на применении trie-деревьев.



ев, поскольку (за исключением методов с использованием trie-деревьев, выполняющих однопутевое ветвление в окончаниях ключей) их время выполнения зависит не от длины ключей, а только от количества цифр, необходимых для различения. Это прямо противоположно, например, алгоритмам хеширования, которые нельзя непосредственно применить для решения этой задачи, поскольку их время выполнения пропорционально длине ключей.

На рис. 15.20 приведены примеры строковых индексов, построенных с использованием BST-деревьев, patricia-деревьев и TST-деревьев (с листьями). В этих индексах используются только ключи, которые начинаются с граничных символов слов; индексирование, начинающееся с границ символов, обеспечило бы более сложное индексирование, но при этом использовался бы гораздо больший объем памяти.

Строго говоря, даже текст, состоящий из случайной строки, не позволяет случайному набору ключей превратится в соответствующий индекс (поскольку ключи не являются независимыми). Однако, в используемых на практике приложениях индексирования редко приходится иметь дело со случайными текстами, и это теоретическое несоответствие не мешает нам воспользоваться преимуществом эффективных реализаций индексирования, которые поддерживают поразрядные методы. Мы воздержимся от подробного рассмотрения характеристик производительности при использовании каждого из алгоритмов для построения строкового индекса, поскольку многие ограничения, проанализированные для общих таблиц символов со строковыми ключами, действуют также и при решении задачи индексирования строк.



**РИСУНОК 15.20 ПРИМЕРЫ ИНДЕКСИРОВАНИЯ ТЕКСТОВЫХ СТРОК**

На этих схемах показаны индексы текстовых строк, построенные из текста *call me ishmael some years ago never mind how long precisely...* с использованием BST-дерева (вверху), patricia-дерева (в центре) и TST-дерева (внизу). Узлы, содержащие строковые указатели, отображены первыми четырьмя символами, расположенными в указываемой указателем точке.

В случае типового текста реализации с использованием стандартных BST-деревьев, вероятно, должны быть рассмотрены в первую очередь, поскольку их легко реализовать (см. программу 12.10). Скорее всего, для типовых приложений это решение должно обеспечивать хорошую производительность. Один из побочных эффектов взаимной зависимости ключей — особенно, при построении строкового индекса для каждой позиции символа — то, что худший случай BST-деревьев не порождает особые проблемы в очень больших текстах, поскольку несбалансированные BST-деревья возникают только в исключительно причудливых конструкциях.

patricia-деревья изначально разрабатывались для приложений строкового индексирования. Для использования программ 15.5 и 15.4 потребуется лишь обеспечить реализацию функции `bit`, чтобы при заданном строковом указателе и целочисленном значении  $i$  она возвращала  $i$ -тый разряд строки (см. упражнение 15.82). На практике зависимость высоты patricia-дерева, реализующего индекс текстовой строки, будет логарифмической. Более того, patricia-дерево обеспечит быстрые реализации поиска для обнаружения промахов, поскольку нет необходимости исследовать все байты ключа.

TST-деревья предоставляют несколько преимуществ в плане производительности, характерных для patricia-деревьев, их просто реализовать и они используют преимущества встроенных операций доступа к байтам, которые в современных компьютерах обычно реализованы. Кроме того, они поддаются простым реализациям, подобным программе 15.9, которые могут решать и более сложные задачи, нежели установка полного соответствия с искомым ключом. Чтобы использовать TST-деревья для построения строкового индекса, необходимо удалить код, обрабатывающий конечные части ключей в структуре данных, поскольку ни одна строка не является префиксом другой и, следовательно, никогда не придется сравнивать строки вплоть до их конечных символов. Эта модификация включает в себя изменение определения `operator==` в интерфейсе типа элемента, чтобы две строки считались равными, если одна из них — префикс другой, как это было сделано в разделе 12.7, поскольку мы будем сравнивать ключ поиска (короткий) с текстовой строкой (длинной), начиная с некоторой позиции текстовой строки. Третье удобное изменение — хранение в каждом узле строковых указателей, а не символов, чтобы каждый узел в дереве ссылался на позицию в текстовой строке (позицию, следующую за первым вхождением строки, определенной символами в равных ветвях от корня до этого узла). Реализация перечисленных изменений — интересное и поучительное упражнение, ведущее к созданию гибкой и эффективной реализации индексирования текстовых строк (см. упражнение 15.81).

Несмотря на все описанные преимущества, имеется одно важное обстоятельство, которым мы пренебрегли при рассмотрении использования BST-деревьев, patricia-деревьев или TST-деревьев для типичных приложений индексирования текста: сам по себе текст фиксирован, поэтому нет необходимости поддерживать динамические операции `insert`, как мы привыкли это делать. То есть, как правило, индекс строится один раз, а затем без каких-либо изменений используется для выполнения огромного объема поисков. Следовательно, динамические структуры данных типа BST-деревьев, patricia-деревьев или TST-деревьев могут оказаться вообще не нужными. Базовый алгоритм, подходящий в данной ситуации — *бинарный поиск* с использованием стро-

ковых указателей (см. раздел 12.4). Индекс — это набор строковых указателей; построение индекса — это сортировка строковых указателей. Основное преимущество использования бинарного поиска по сравнению с динамическими структурами данных заключается в экономии используемого объема памяти. Для индексирования текстовой строки в  $N$  позициях при помощи бинарного поиска требуется всего лишь  $N$  строковых указателей; и напротив, для индексирования строки в  $N$  позициях за счет использования метода, основанного на деревьях, требуется по меньшей мере  $3N$  указателей (один строковый указатель для текста и две связи). Как правило, индексные указатели текста огромны, поэтому бинарный поиск может оказаться более предпочтительным, т.к. гарантировано обеспечивает логарифмическую зависимость времени поиска, но при этом задействуется лишь одна треть объема памяти, используемого методами, основанными на деревьях. Однако, при наличии достаточного объема доступной памяти TST-деревья позволяют реализовать более быстрые операции *search* для многих приложений, поскольку, в отличие от бинарного поиска, перемещение по ключам выполняется без возвратов.

Для случая очень большого текста, когда планируется выполнять лишь небольшое количество поисков, построение полного индексного указателя, скорее всего, будет неоправданным. В части 5 рассматривается задача *поиска строк*, при которой без какой-либо предварительной обработки требуется быстро определить, содержит ли текст заданный искомый ключ. В ней также исследуется ряд задач поиска строк, которые лежат между двумя крайними ситуациями без выполнения какой-либо предварительной обработки и построения полного индексного указателя для огромного текста.

## Упражнения

- ▷ **15.77** Нарисуйте 26-путевое DST-дерево, образованное в результате построения индекса текстовой строки из слов **now is the time for all good people to come the aid of their party**.
- ▷ **15.78** Нарисуйте 26-путевое trie-дерево, образованное в результате построения индекса текстовой строки из слов **now is the time for all good people to come the aid of their party**.
- ▷ **15.79** Нарисуйте TST-дерево, образованное в результате построения индекса текстовой строки из слов **now is the time for all good people to come the aid of their party** в стиле рис. 15.20.
- ▷ **15.80** Нарисуйте TST-дерево, образованное в результате построения индекса текстовой строки из слов **now is the time for all good people to come the aid of their party**, при использовании описанной в тексте реализации, когда TST-дерево содержит строковые указатели на каждый из узлов.
- **15.81** Измените реализации поиска и вставки в TST-дерево, приведенные в программах 15.11 и 15.12, так, чтобы обеспечить индексирование строк, основанное на TST-дереве.
- **15.82** Реализуйте интерфейс, который позволяет patricia-алгоритму обрабатывать строковые ключи в стиле C (т.е. массивы символов), как если бы они были строками разрядов.

- **15.83** Нарисуйте *patricia*-дерево, образованное в результате построения индекса текстовой строки из слов **now is the time for all good people to come the aid of their party** при использовании 5-разрядного двоичного кодирования, когда  $i$ -тая буква алфавита хранится в виде двоичного представления числа  $i$ .
- 15.84** Объясните, почему идея улучшения бинарного поиска путем использования того же базового принципа, на котором основываются TST-деревья (сравнение символов, а не строк), оказывается неэффективной.
- 15.85** Найдите в системе большой (размером, по меньшей мере,  $10^6$  байтов) текстовый файл и сравните высоту и длину внутреннего пути стандартного BST-дерева, *patricia*-дерева и TST-дерева, полученных в результате построения индексного указателя для данного файла.
- 15.86** Экспериментально сравните высоту и длину внутреннего пути стандартного BST-дерева, *patricia*-дерева и TST-дерева, полученных в результате построения индексного указателя для текстовой строки, состоящей из  $N$  случайных символов 32-символьного алфавита при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **15.87** Создайте эффективную программу для определения самой длинной повторяющейся последовательности в очень большой текстовой строке.
- **15.88** Создайте эффективную программу для определения 10-символьной последовательности, чаще всего встречающейся в очень большой текстовой строке.
- **15.89** Постройте строковый индексный указатель, который поддерживает операцию, возвращающую количество вхождений ее аргумента в индексированном тексте, а также, подобно операции *sort*, поддерживает операцию *search*, обеспечивающую посещение всех позиций в тексте, которые соответствуют искомому ключу.
- **15.90** Опишите текстовую строку, состоящую из  $N$  символов, для которой основанный на применении TST-дерева строковый индекс работает особенно плохо. Оцените затраты на построение индекса для этой же строки с использованием BST-дерева.
- 15.91** Предположим, что требуется построить индекс для случайной  $N$ -разрядной строки для позиций разрядов, кратных 16. Экспериментально определите, какие размеры байтов (1, 2, 4, 8 или 16) ведут к наименьшему времени построения индекса, основанного на использовании TST-дерева, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

## Внешний поиск

**А**лгоритмы поиска, которые подходят для доступа к элементам в больших файлах, имеют огромное практическое значение. Поиск — это фундаментальная операция для больших наборов данных, и для ее выполнения безусловно требуется значительная часть ресурсов, используемых во многих вычислительных средах. С появлением глобальных сетей появилась возможность собирать практически любую информацию, требуемую для выполнения задачи — проблема заключается только в возможности эффективного поиска. В этой главе рассмотрены базовые механизмы, которые могут поддерживать эффективный поиск в самых больших таблицах символов, какие только можно себе представить.

Подобно алгоритмам из главы 11, алгоритмы, рассматриваемые в этой главе, подходят для множества различных типов аппаратных и программных сред. Соответственно, мы будем стремиться к исследованию алгоритмов на более абстрактном уровне, нежели это могут дать программы на C++. Однако, приведенные далее алгоритмы также непосредственно обобщают знакомые методы поиска и удобно выражаются в виде программ на C++, полезных во многих ситуациях. В этой главе будет использоваться подход, отличный от использованного в главе 11: мы в деталях разработаем конкретные реализации, рассмотрим их основные характеристики производительности, а затем способы, в соответствии с которыми лежащие в основе реализаций алгоритмы могут быть сделаны полезными в возникающих реальных ситуациях. Вообще говоря, название этой главы не совсем верно, поскольку в ней алгоритмы будут представляться в виде программ на C++, которые можно было бы заменить другими реали-

зациями таблиц символов из рассмотренными в главах 12—15. Будучи таковыми, они вообще не являются "внешними" методами. Тем не менее, они построены в соответствии с простой абстрактной моделью, что превращает их в подробное описание того, как строить методы поиска для конкретных внешних устройств.

Подробные абстрактные модели менее полезны, чем применительно к сортировке, поскольку связанные с ними затраты весьма невелики для многих важных приложений. В основном нас будут интересовать методы поиска в очень больших файлах, хранящихся на внешних устройствах наподобие дисков, которые обеспечивают быстрый доступ к произвольным блокам данных. Для устройств типа ленточных накопителей, где допускается только последовательный доступ (похожая модель рассматривалась в главе 11), поиск вырождается до тривиального (и медленного) метода считывания с начала файла до тех пор, пока поиск не будет завершен. Для дисковых устройств можно воспользоваться гораздо более эффективным подходом: как ни удивительно, методы, которые мы изучим, могут поддерживать операции *search* и *insert* в таблицах символов, содержащих миллиарды и триллионы элементов, при использовании всего трех-четырёх ссылок на блоки данных на диске. Такие системные параметры, как размер блока и отношение затрат на доступ к новому блоку к затратам на доступ к элементам внутри блока, влияют на производительность, но методы можно считать относительно не зависящими от этих параметров (в пределах значений, которые, скорее всего, будут встречаться на практике). Более того, большинство важных шагов, которые необходимо предпринимать для подгонки методов под конкретные реальные ситуации, вполне очевидны.

Поиск — это фундаментальная операция для дисковых устройств. Как правило, файлы организованы так, чтобы можно было воспользоваться преимуществами конкретного устройства с целью максимально эффективного доступа к информации. Короче говоря, вполне можно предположить, что устройства, используемые для хранения очень больших объемов информации, построены именно для поддержки эффективных и простых реализаций операции *search*. В этой главе рассматриваются алгоритмы, которые построены на несколько более высоком уровне абстракции, нежели базовые операции, обеспечиваемые дисковыми устройствами, и которые могут поддерживать операцию *insert* и другие динамические операции для таблиц символов. Эти методы по сравнению с прямыми методами предоставляют такие же преимущества, какие деревья бинарного поиска и хеш-таблицы предоставляют по сравнению с бинарным и последовательным поиском.

Во многих вычислительных средах разрешена непосредственная адресация для огромных объемов *виртуальной памяти*, которая полагается на систему в плане определения эффективных способов обработки запросов данных любой программы. Рассматриваемые алгоритмы могут послужить также эффективными решениями задачи реализации таблицы символов в упомянутых средах.

Массив информации, которая должна обрабатываться компьютером, называется *базой данных*. Значительная часть исследований посвящена методам построения, поддержания и использования баз данных. Большая часть этой работы выполняется в области создания абстрактных моделей и реализаций для поддержки операций *search* с более сложным критерием, нежели рассмотренное простое "соответствие отдельному ключу". В базе данных поиски могут основываться на критерии частичного соот-

ветствия, который может включать несколько ключей и возвращать большое количество элементов. Методы этого типа затрагиваются в частях 5 и 6. В общем случае запросы на поиск достаточно сложны, чтобы зачастую приходилось выполнять последовательный поиск по всей базе данных, проверяя каждый элемент на соответствие критерию. Тем не менее, быстрый поиск крошечных фрагментов данных, соответствующих конкретному критерию, в огромном файле — основная возможность в любой системе баз данных, и многие современные базы данных строятся на основе описанных в этой главе механизмов.

## 16.1 Правила игры

Подобно тому как это было сделано в главе 11, будем считать, что последовательный доступ к данным требует значительно меньших затрат, чем непоследовательный доступ. Предметом моделирования будет определение ресурсов памяти, используемых для реализации таблицы символов при делении ее на *страницы*: непрерывные блоки информации, к которым может обеспечиваться эффективный доступ со стороны дисковых устройств. Каждая страница будет содержать множество элементов; наша задача заключается в организации элементов внутри страниц таким образом, чтобы к любому элементу можно было обратиться путем чтения всего нескольких страниц. Мы будем предполагать, что время ввода/вывода, требуемое для считывания страницы, неизмеримо больше времени, требуемого для доступа к конкретным элементам или для выполнения любых других вычислений, связанных с этой страницей. Во многих отношениях эта модель слишком упрощена, но она сохраняет достаточное количество характеристик внешних устройств хранения, чтобы можно было рассмотреть фундаментальные методы.

**Определение 16.1** *Страница* — это непрерывный блок данных. *Зондирование* — это первое обращение к странице.

Нас интересуют реализации таблиц символов, в которых используется небольшое количество зондирований. Мы будем избегать конкретных предположений касательно размеров страницы и отношения времени, требуемого для зондирования, ко времени, которое впоследствии потребуется для доступа к элементам внутри блока. Ожидается, что эти значения должны быть порядка 100–1000; большая точность не требуется, поскольку алгоритмы не особо чувствительны к этим значениям.

Эта модель непосредственно применима, например, к файловой системе, в которой файлы образуют блоки с уникальными идентификаторами и назначение которой состоит в поддержке эффективного доступа, вставки и удаления на основе этих идентификаторов. В блоке помещается определенное количество элементов, и затратами на обработку элементов внутри блока можно пренебречь по сравнению с затратами на считывание блока.

Эта модель непосредственно применима и к системе виртуальной памяти, где мы просто ссылаемся на огромный объем памяти и предоставляем системе самой сохранять часто используемую информацию в хранилище с быстрым доступом (таким, как внутренняя память), а редко используемую информацию — в хранилище с медленным доступом (таким, как диск). Многие компьютерные системы располагают слож-

ными механизмами страничной обработки, которые реализуют виртуальную память, храня недавно использовавшиеся страницы в *кэше*, к которому можно быстро обратиться. Системы страничной обработки основываются на той же рассматриваемой абстракции: они делят диск на блоки и делают допущение, что затраты на первый доступ к блоку существенно превышают затраты на доступ к данным внутри блока.

Как правило, такое абстрактное представление страницы полностью соответствует блоку в файловой системе или странице в системе виртуальной памяти. Для простоты в общем случае при рассмотрении алгоритмов будет предполагаться, что такое соответствие актуально. В конкретных приложениях, в зависимости от системы или приложения, блок может состоять из нескольких страниц либо несколько блоков могут образовывать одну страницу; подобные нюансы не снижают эффективность алгоритмов, тем самым лишь подчеркивая выгодность работы на абстрактном уровне.

Мы манипулируем страницами, ссылками на страницы и элементами с ключами. Применительно к большой базе данных наиболее важная проблема, которую следует рассмотреть заключается в поддержке *индексов* для данных. То есть, как кратко упоминалось в разделе 12.7, предполагается, что элементы, образующие нашу таблицу символов, хранятся где-то в статической форме, а наша задача состоит в построении структуры данных с ключами и ссылками на элементы, которая позволяет быстро ссылаться на данный элемент. Например, телефонная компания может хранить информацию о клиентах в огромной статической базе данных при наличии нескольких индексов, возможно, использующих различные ключи, для ежемесячных оплат, для обработки ежедневных транзакций, периодических обращений к клиентам и т.п. Для очень больших наборов данных индексы имеют первостепенное значение: в основном, копии основных данных не создаются не только потому, что невозможно выделить дополнительный объем памяти, но и потому, что желательно избежать проблем, связанных с поддержанием целостности данных в условиях присутствия нескольких копий.

Соответственно, в общем случае предполагается, что каждый элемент представляет собой *ссылку* на фактические данные, которая может быть адресом страницы или каким-либо более сложным интерфейсом для базы данных. Для простоты будем хранить в своих структурах данных не копии элементов, а копии ключей — часто такой подход оказывается наиболее практичным. Кроме того, для простоты описания алгоритмов мы не будем использовать абстрактный интерфейс для ссылок на элементы и страницы — вместо этого будем использовать только указатели. Таким образом, мы сможем использовать свои реализации непосредственно в среде виртуальной памяти, но нам придется преобразовать указатели и обращения к указателям в более сложные механизмы, чтобы сделать их действительно внешними методами сортировки.

Мы рассмотрим алгоритмы, которые для широкого диапазона значений двух основных параметров (размера блока и относительного времени доступа) реализуют *поиск* (*search*), *вставки* (*insert*) и другие операции в полностью динамической таблице символов за счет использования всего нескольких зондирований для каждой операции. В типичном случае, когда выполняется огромное количество операций, может оказаться эффективной тщательная настройка. Например, если типовые затраты на поиск удастся снизить с трех зондирований до двух, производительность системы может быть повышена на 50 процентов! Однако в этой главе подобная настройка не



рассматривается; ее эффективность в очень большой степени зависит от систем и приложений.

В первых компьютерах внешние устройства хранения были хитроумными устройствами, которые были не только большими и медленно работающими, но и не могли хранить большие объемы информации. Соответственно, было важно обойти их ограничения, и фольклор начала эры программирования переполнен историями о программах доступа ко внешним файлам, которые обеспечивали прекрасную синхронизацию для считывания данных с вращающегося диска или барабана, или как-либо иначе минимизировали физические перемещения, необходимые для доступа к данным. В то же время существует полно историй о громких провалах подобных попыток, когда малейшие просчеты существенно замедляли процесс по сравнению с простейшими реализациями. И напротив, современные устройства хранения информации не только очень малы по размерам и работают исключительно быстро, но и способны хранить огромные объемы информации, поэтому в общем случае такие проблемы решать не приходится. Действительно, в современных программных средах мы стремимся избегать зависимости от конкретных реальных устройств — в целом гораздо важнее, чтобы программы эффективно работали на различных компьютерах (включая и будущие разработки), чем чтобы они обеспечивали максимальную производительность на конкретном устройстве.

Для баз данных с длительным сроком использования существует множество вопросов реализации, связанных с основными целями поддержания целостности данных и обеспечения гибкого и надежного доступа. Здесь эти вопросы не рассматриваются. Для таких приложений рассматриваемые методы можно считать фундаментальными алгоритмами, которые непременно обеспечат высокую производительность и послужат отправной точкой при разработке системы.

## 16.2 Индексированный последовательный доступ

Прямой подход к построению индекса заключается в сохранении массива со ссылками на ключи и элементы, упорядоченного по ключам, с последующим использованием бинарного поиска (см. раздел 12.4) для реализации операции *search*. При наличии  $N$  элементов для реализации этого метода потребовалось бы  $\lg N$  зондирований, даже в случае очень большого файла. Наша базовая модель немедленно принуждает к рассмотрению двух разновидностей этого простого метода. Во-первых, сам по себе индекс очень велик и в общем случае не помещается на одной странице. Поскольку доступ к страницам можно получить только через ссылки на страницы, вместо этого можно построить явное полностью сбалансированное бинарное дерево с ключами и указателями на страницы во внутренних узлах и с ключами и указателями на элементы во внешних. Во-вторых, затраты на доступ к  $M$  записям таблицы равны затратам на доступ к 2, поэтому можно воспользоваться  $M$ -арным деревом при таких же затратах на каждый узел, что и в случае применения бинарного дерева. Такое усовершенствование уменьшает количество зондирований до значения, приблизительно пропорционального  $\log_M N$ . Как было показано в главах 10–15, на практике это значение можно считать константным. Например, если  $M$  равно 1000, то  $\log_M N$  меньше 5 для  $N$  меньше 1 триллиона.

На рис. 16.1 приведен пример набора ключей, а на рис. 16.2 — пример такой структуры дерева для этих ключей. Чтобы примеры были достаточно понятными, пришлось использовать сравнительно небольшие значения  $M$  и  $N$ , тем не менее, они иллюстрируют, что деревья для большого значения  $M$  будут плоскими.

Показанное на рис. 16.2 дерево является абстрактным, не зависящим от устройства представлением индекса, которое аналогично множеству других рассмотренных структур данных. Обратите внимание, что вместе с тем оно не особо отличается от *зависящих* от устройств индексов, которые можно встретить в программном обеспечении низкоуровневого доступа к дискам. Например, в ряде первых систем использовалась двухуровневая схема, в которой нижний уровень соответствовал элементам на страницах конкретного дискового устройства, а второй уровень — главному индексу отдельных устройств. В таких системах главный индекс хранился в основной памяти, поэтому для доступа к элементу через такой индекс требовались две операции доступа: одна для получения индекса и вторая для получения страницы, содержащей элемент. С увеличением емкости дисков возрастали и размеры индексов, причем для хранения индекса требовалось несколько страниц, что со временем привело к использованию иерархической схемы наподобие показанной на рис. 16.2. Мы продолжим работать с абстрактным представлением, зная, что при необходимости оно может быть непосредственно реализовано с помощью типового системного аппаратного и программного обеспечения низкого уровня.

Во многих современных системах аналогичная структура дерева используется для организации огромных файлов в виде последовательностей дисковых страниц. Такие деревья не содержат ключей, но они могут эффективно поддерживать стандартные операции доступа к файлам, хранящимся по порядку, и, если каждый узел содержит счетчик размера его дерева, то нахождения страницы, которая содержит  $k$ -тый элемент файла.

Поскольку метод индексации, показанный на рис. 16.2, объединяет последовательную организацию ключей с индексным доступом, по историческим причинам такой метод называется *индексным последовательным доступом*. Этот метод удобен для приложений, в которых изменения в базе данных выполняются редко. Иногда сам индекс называют *каталогом*. Недостаток использования индексного последовательного доступа заключается в том, что изменение каталога представляет собой операцию, сопряженную с большими затратами. Например, для добавления единственного ключа может потребоваться перестройка буквально всей базы данных с присвоением новых позиций многим ключам и новых значений индексам. Для преодоления упомянуто-

706	111000110
176	001111110
601	110000001
153	001101011
513	101001011
773	111111011
742	111100010
373	011111011
524	101010100
766	111110110
275	010111101
737	111011111
574	101111100
434	100011100
641	110100001
207	010000111
001	000000001
277	010111111
061	000110001
736	111011110
526	101010110
562	101110010
017	000001111
107	001000111
147	001100111

#### РИСУНОК 16.1 ДВОИЧНОЕ ПРЕДСТАВЛЕНИЕ ВОСЬМЕРИЧНЫХ КЛЮЧЕЙ

*Ключи (слева), используемые в примерах в этой главе, представляют собой трехзначные восьмеричные числа, которые также интерпретируются как 9-разрядные двоичные значения (справа).*

го недостатка и обеспечения возможности увеличения базы данных в будущем в ранних системах резервировались лишние страницы на дисках и лишнее пространство на страницах; но, в конечном счете, подобная технология оказывалась не очень эффективной в динамических ситуациях (см. упражнение 16.3). Методы, которые будут рассматриваться в разделах 16.3 и 16.4, обеспечивают систематичные и эффективные альтернативы таким специализированным схемам.

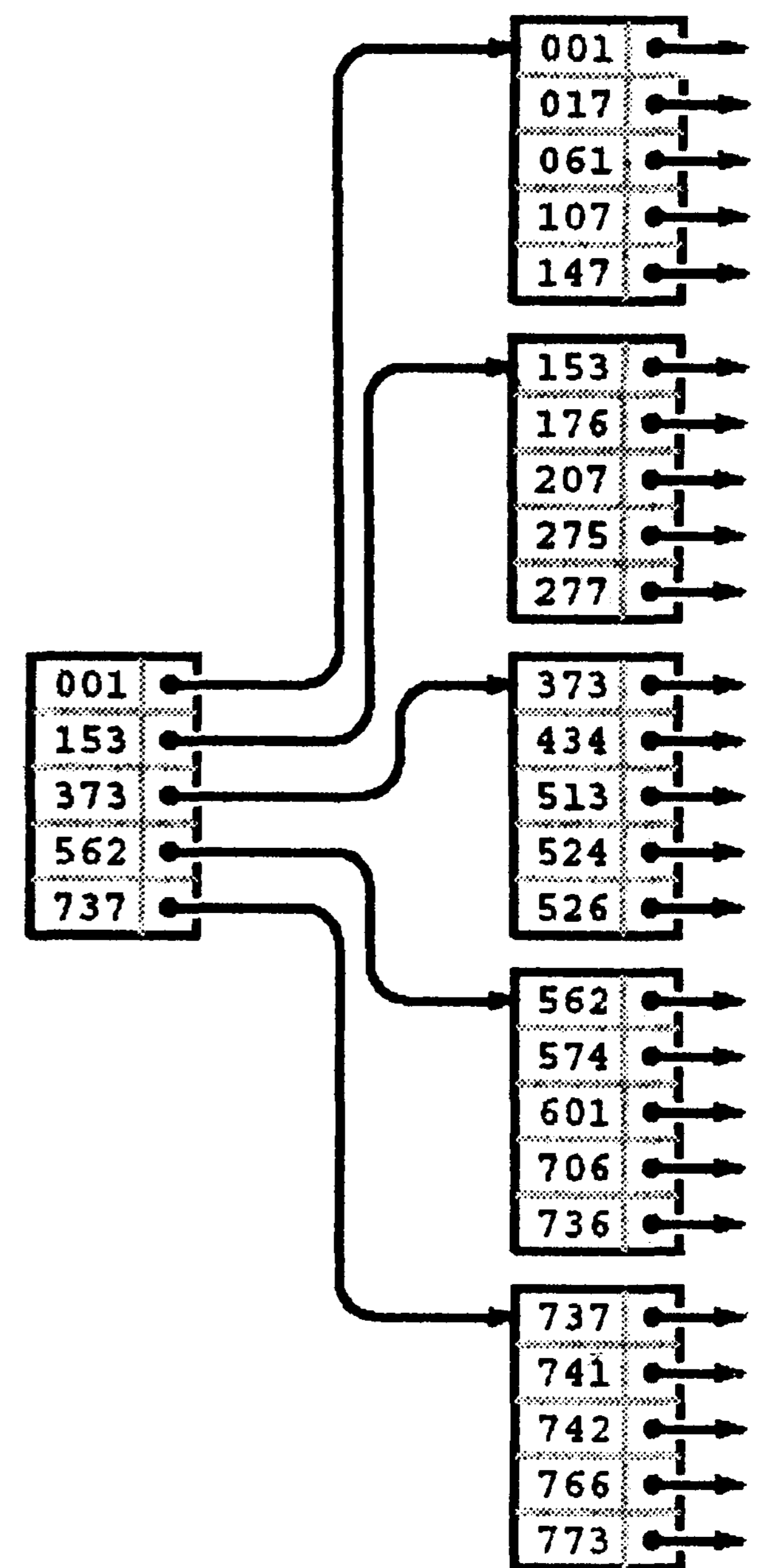
**Лемма 16.1** *Для выполнения поиска в индексном последовательном файле требуется выполнение только постоянного количества зондирований, однако вставка может быть сопряжена с перестройкой всего индекса.*

В данном случае (и в других разделах книги) термин *постоянное* используется несколько вольно для обозначения значения, которое пропорционально  $\log_M N$  для большого значения  $M$ . Как уже отмечалось, это оправдано для размеров файлов, встречающихся на практике. На рис. 16.3 показаны дополнительные примеры. Даже при наличии 128-разрядного ключа поиска, пригодного для указания неизмеримо большого количества различных элементов, равного  $2^{128}$ , элемент с данным ключом можно было бы найти при помощи всего 13 зондирований, при 1000-позиционном ветвлении.

Мы не будем рассматривать реализации, которые обеспечивают поиск и построение индексов подобного типа, поскольку они представляют собой специальные случаи более общих механизмов, рассмотренных в разделе 16.3 (см. упражнение 16.17 и программу 16.2).

## Упражнения

- ▷ 16.1 Составьте таблицу значений  $\log_M N$  для  $M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- ▷ 16.2 Нарисуйте структуру индексного последовательного файла для ключей 516, 177, 143, 632, 572, 161, 774, 470, 411, 706, 461, 612, 761, 474, 774, 635, 343, 461, 351, 430, 664, 127, 345, 171 и 357 при  $M = 5$  и  $N = 6$ .
- 16.3 Предположим, что мы строим структуру индексированного последовательного файла для  $N$  элементов, помещаемых на страницах емкостью  $M$ , но оставляем на каждой странице  $k$  свободных ячеек для возможности расширения. Приведите формулу для определения количества зондирований, необходимых для выполнения поиска, в виде функции от  $N, M$  и  $k$ . Используйте эту формулу для определения количества зондирований, необходимого для выполнения поиска при  $k = M/10, M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .



**РИСУНОК 16.2** СТРУКТУРА ИНДЕКСНОГО ПОСЛЕДОВАТЕЛЬНОГО ФАЙЛА

*В последовательном индексе ключи хранятся по порядку в полных страницах (справа), причем индекс указывает на самый малый ключ на каждой странице (слева). Для добавления ключа потребуется перестроить всю структуру данных.*

- 16.4 Предположите, что затраты на зондирование составляют около  $\alpha$  условных единиц времени, а средние затраты на поиск элемента на странице составляют около  $\beta M$  условных единиц времени. Найдите значение  $M$ , при котором затраты на поиск в структуре индексного последовательного файла минимальны, при  $\alpha/\beta = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

## 16.3 В-деревья

Для построения структур поиска, которые могут быть эффективны в динамических ситуациях, мы будем строить многопозиционные деревья, но при этом откажемся от ограничения, в соответствии с которым каждый узел должен иметь в точности  $M$  записей. Вместо этого выдвинем условие, что каждый узел должен иметь максимум  $M$  записей, чтобы они помещались на странице, но при этом узлы могут иметь и меньше записей. Чтобы гарантировать, что узлы имеют достаточное количество записей для обеспечения ветвления, необходимого ради предотвращения увеличения длины путей, мы потребуем, чтобы все узлы имели, по меньшей мере (скажем) по  $M/2$  записей, за исключением, быть может, корня, который должен иметь не менее одной записи (двух связей). Причина этого исключения для корня станет понятна при подробном рассмотрении алгоритма. Такие деревья были названы *В-деревьями* благодаря Байеру (Bayer) и МакКрейту (McCreight), которые в 1970 г. первыми воспользовались многопутевыми сбалансированными деревьями для внешнего поиска. Термин *В-дерево* часто используется для описания именно той структуры данных, которая строится алгоритмом, предложенным Байером и МакКрейтом; мы же будем использовать его в качестве общего термина для обозначения семейства связанных алгоритмов.

Мы уже встречались с реализацией В-дерева: из определений 13.1 и 13.2 видно, что В-деревья четвертого порядка, в которых каждый узел имеет не более 4 и не менее 2 связей, являются ни чем иным, как сбалансированными 2-3-4-деревьями, описанными в главе 13. Действительно, лежащую в их основе абстракцию можно непосредственно обобщить и реализовать В-деревья, обобщив реализации нисходящего 2-3-4-дерева, описанные в разделе 13.4. Однако, различия между внешним и внутренним поиском, упомянутые в разделе 16.1, приводят к ряду различий в реализациях. В этом разделе мы рассмотрим реализацию, которая

- обобщает 2-3-4-деревья до деревьев, имеющих от  $M/2$  до  $M$  узлов
- представляет многопутевые узлы при помощи массива элементов и связей

$10^5$	слов в словаре
$10^6$	слов в романе "Моби Дик"
$10^9$	номеров карточек социального страхования
$10^{12}$	телефонных номеров в мире
$10^{15}$	людей, когда-либо живших на Земле
$10^{20}$	песчинок на побережье Кони-Айленда
$10^{25}$	разрядов во всех изготовленных модулях памяти
$10^{79}$	электронов во Вселенной

### РИСУНОК 16.3 ПРИМЕРЫ РАЗМЕРОВ НАБОРОВ ДАННЫХ

*Эти впечатляющие граничные значения демонстрируют, что на практике вряд ли встретится таблица символов, содержащая более  $10^{30}$  элементов. Даже в такой невероятно большой базе данных элемент с заданным ключом можно было бы найти посредством менее 10 зондирований при 1000-позиционном ветвлении. Даже если бы как-то удалось сохранить информацию о каждом электроны во Вселенной, 1000-позиционное ветвление позволило бы получить доступ к любому конкретному элементу через менее чем 27 зондирований.*

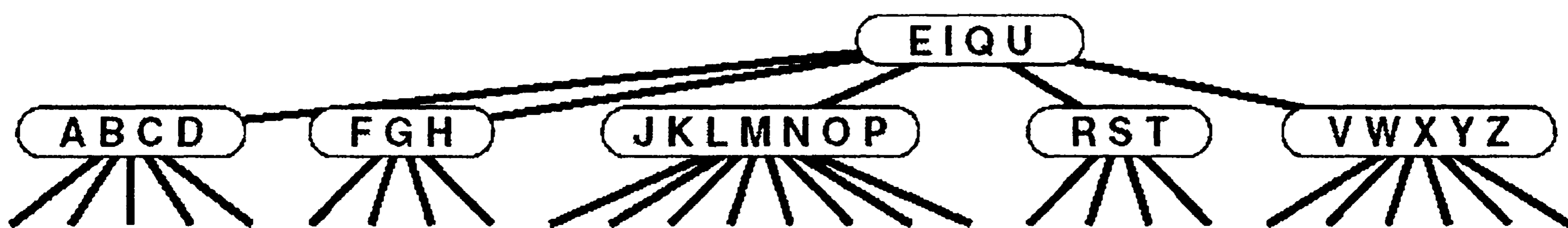
- реализует индекс, а не структуру поиска, содержащую элементы
- выполняет восходящее разделение
- отделяет индексы от элементов

Два последних свойства в этом перечне несущественны, однако удобны во многих ситуациях и обычно встречаются в реализациях В-деревьев.

На рис. 16.4 показано абстрактное 4-5-6-7-8-дерево, являющееся обобщением 2-3-4-дерева из раздела 13.3. Обобщение очевидно: 4-узлы имеют три ключа и четыре связи, 5-узлы — четыре ключа и пять связей и т.д., при использовании одной связи для каждого возможного интервала между ключами. Поиск начинается с корня и выполняется переход от одного узла к другому с определением соответствующего интервала для искомого ключа в текущем узле. Далее осуществляется выход по соответствующей связи, чтобы добраться до следующего узла. Поиск завершается попаданием, если искомым ключом отыскивается в любом рассмотренном узле, и промахом, если нижняя часть дерева достигается без попадания. Подобно тому как это было возможно в 2-3-4-деревьях, новый ключ можно вставить в нижнюю часть дерева после выполнения поиска, если по дороге вниз по дереву выполняется разделение полных узлов: если корень — 8-узел, он заменяется 2-узлом, связанным с двумя 4-узлами. Затем, каждый раз когда встречается  $k$ -узел, он заменяется  $(k + 1)$ -узлом, соединенным с двумя 4-узлами. Этот подход гарантирует наличие места для вставки нового узла в случае достижения нижней части дерева.

Или же, как описывалось в разделе 13.3 применительно к 2-3-4-деревьям, разделение можно выполнять снизу вверх: вставка реализуется через поиск и помещение нового ключа в нижний узел, если только последний не является 8-узлом — в этом случае он делится на два 4-узла со вставкой среднего ключа и двух связей в его родительский узел. Восходящее разделение выполняется до тех пор, пока не встретится узел-потомок, отличный от 8-узла.

Путем замены в предыдущих двух абзацах 4 на  $M/2$ , а 8 — на  $M$ , приведенные описания преобразуются в описания поиска и вставки для  $M/2$ -...- $M$ -деревьев при любом положительном целочисленном значении  $M$ , кратном 2 (см. упражнение 16.9).



**РИСУНОК 16.4 4-5-6-7-8-ДЕРЕВО**

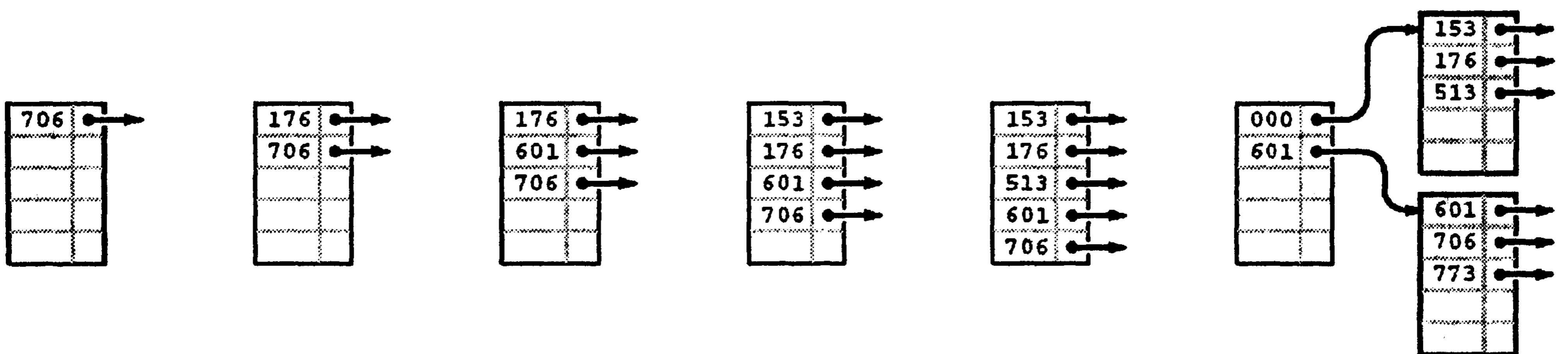
На рисунке показано обобщение 2-3-4-деревьев, построенных из узлов с от 4 до 8 связей (и от 3 до 7 связей, соответственно). Как и в случае 2-3-4-деревьев, мы поддерживаем высоту деревьев постоянной за счет разделения 8-узлов с использованием либо нисходящего, либо восходящего алгоритма. Например, для вставки в показанное дерево еще одного J следовало бы сначала разделить 8-узел на два 4-узла, а затем вставить M в корень, преобразуя его в 6-узел. Если дело доходит до разделения корня, то не остается ничего другого, кроме как создание нового корня, который будет 2-узлом. В результате корень освобождается от ограничения, что узлы должны иметь по меньшей мере четыре связи.

**Определение 16.2** *B-дерево* порядка  $M$  — это дерево, которое либо пусто, либо состоит из  $k$ -узлов с  $k-1$  ключами и  $k$  связями с деревьями, представляющими каждый из  $k$  ограниченных ключами интервалов, и обладающее следующими структурными свойствами:  $k$  должно находиться в интервале между 2 и  $M$  для корня и между  $M/2$  и  $M$  для любого другого узла; все связи с пустыми деревьями должны находиться на равном расстоянии от корня.

Алгоритмы B-деревьев строятся на основе этого базового набора абстракций. Как и в главе 13, существует значительная свобода в выборе конкретных представлений таких деревьев. Например, можно использовать расширенное RB-представление (см. упражнение 13.69). Для внешнего поиска мы используем еще более простое представление в виде упорядоченного массива, при условии, что значение  $M$  достаточно велико, чтобы  $M$ -узлы заполняли страницу. Коэффициент ветвления равен, по меньшей мере,  $M/2$ , поэтому, как следует из леммы 16.1, количество зондирований, необходимое для выполнения любого поиска или вставки, по сути, постоянно.

Вместо того чтобы реализовать только что описанный метод, рассмотрим вариант, обобщающий стандартный индексный указатель, рассмотренный в главе 16.1. Мы храним ключи со ссылками на элементы на *внешних страницах* в нижней части дерева, а копии ключей со ссылками на страницы — на *внутренних страницах*. Мы вставляем новые элементы в нижнюю часть, а затем используем базовую абстракцию  $M/2$ -...- $M$  дерева. Когда страница содержит  $M$  записей, мы разделяем ее на две страницы с  $M/2$  записями в каждой и вставляем ссылку на новую страницу в родительскую страницу. При разделении корня мы создаем новый корень с двумя дочерними узлами, тем самым увеличивая высоту дерева на 1.

На рис. 16.5–16.7 показано B-дерево, которое было построено в результате вставки ключей, показанных на рис. 16.1 (в приведенном порядке) в первоначально пустое дерево при  $M = 5$ .

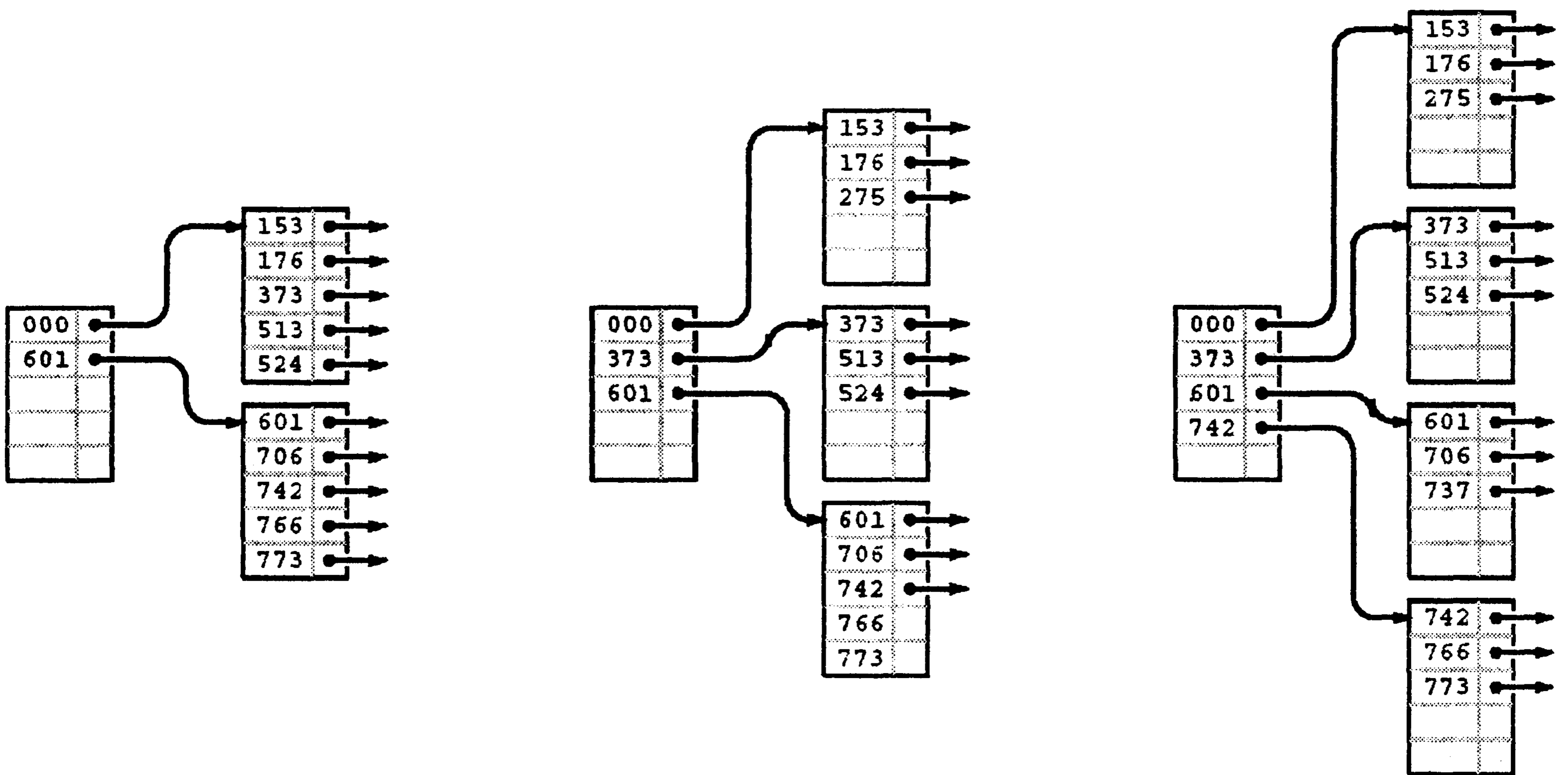


**РИСУНОК 16.5 ПОСТРОЕНИЕ B-ДЕРЕВА, ЧАСТЬ 1**

В этих примерах показаны шесть вставок в первоначально пустое B-дерево, построенное из страниц, которые могут содержать пять ключей и связей, при использовании ключей, являющихся 3-значными восьмеричными числами (9-разрядными двоичными числами). Ключи в страницах хранятся по порядку. Шесть вставок приводят к разделению на два внешних узла с тремя ключами в каждом и внутренний узел, служащий в качестве индекса: его первый указатель указывает на страницу, содержащую все ключи, которые больше или равны ключу 000, но меньше ключа 601, а второй указатель — на страницу, содержащую все ключи, которые больше или равны ключу 601.

Выполнение вставок требует всего лишь добавления элемента в страницу, но на основании структуры результирующего дерева можно определить важные события, происходящие во время его построения. Дерево содержит семь внешних страниц, поэтому должно было быть выполнено шесть разделений внешних узлов, и поскольку его высота равна 3, корень дерева должен был разделяться дважды. Эти события описаны в сопровождающих рисунки комментариях.

В программе 16.1 приведены определения типа для узлов рассматриваемой реализации В-дерева. Мы не указываем подробно структуру узлов, что следовало бы сделать в реальной реализации, поскольку для этого потребовалась бы ссылка на конкретные страницы диска. Для простоты мы используем один тип узлов, когда каждый узел состоит из массива записей, каждая из которых содержит элемент, ключ и связь. Каждый узел содержит также счетчик, указывающий количество активных записей. Мы не обращаемся к элементам во внутренних узлах, мы не обращаемся к связям во внешних узлах, и мы не обращаемся к ключам внутри элементов дерева. При использовании конкретной структуры данных в реальном приложении можно сэкономить память за счет применения таких конструкций, как **union** (объединение) или производные классы. Можно было бы также сэкономить память ценой увеличения времени выполнения, используя везде в дереве связи с элементами вместо ключей. Такие конструктивные решения сопряжены с очевидными изменениями кода и зависят от конкретного характера ключей, элементов и связей в приложении.



**РИСУНОК 16.6 ПОСТРОЕНИЕ В-ДЕРЕВА, ЧАСТЬ 2**

*После вставки четырех ключей 742, 373, 524 и 766 в крайнее справа В-дерево, показанное на рис. 16.5, обе внешние страницы оказываются заполненными (рисунок слева). Затем, при вставке ключа 275 первая страница разделяется с пересылкой связи с новой страницей (вместе с наименьшим ее ключом 373) вверх в индексе (рисунок в центре). Далее, при вставке ключа 737 разделяется нижняя страница, также с пересылкой связи с новой страницей вверх в индексе (рисунок справа).*

Программа 16.1 Определения типов узлов В-дерева

Каждый узел в В-дереве содержит массив и счетчик количества активных записей в массиве. Каждая запись массива представляет собой ключ, элемент и связь с узлом. Во внутренних узлах используются только ключи и связи; во внешних узлах используются только ключи и элементы. Новые узлы инициализируются в виде пустых узлов путем установки значения поля счетчика равным 0.

```

template <class Item, class Key>
struct entry
{ Key key; Item item; struct node *next; };
struct node
{ int m; entry<Item, Key> b[M];
  node() { m = 0; }
};
typedef node *link;
    
```

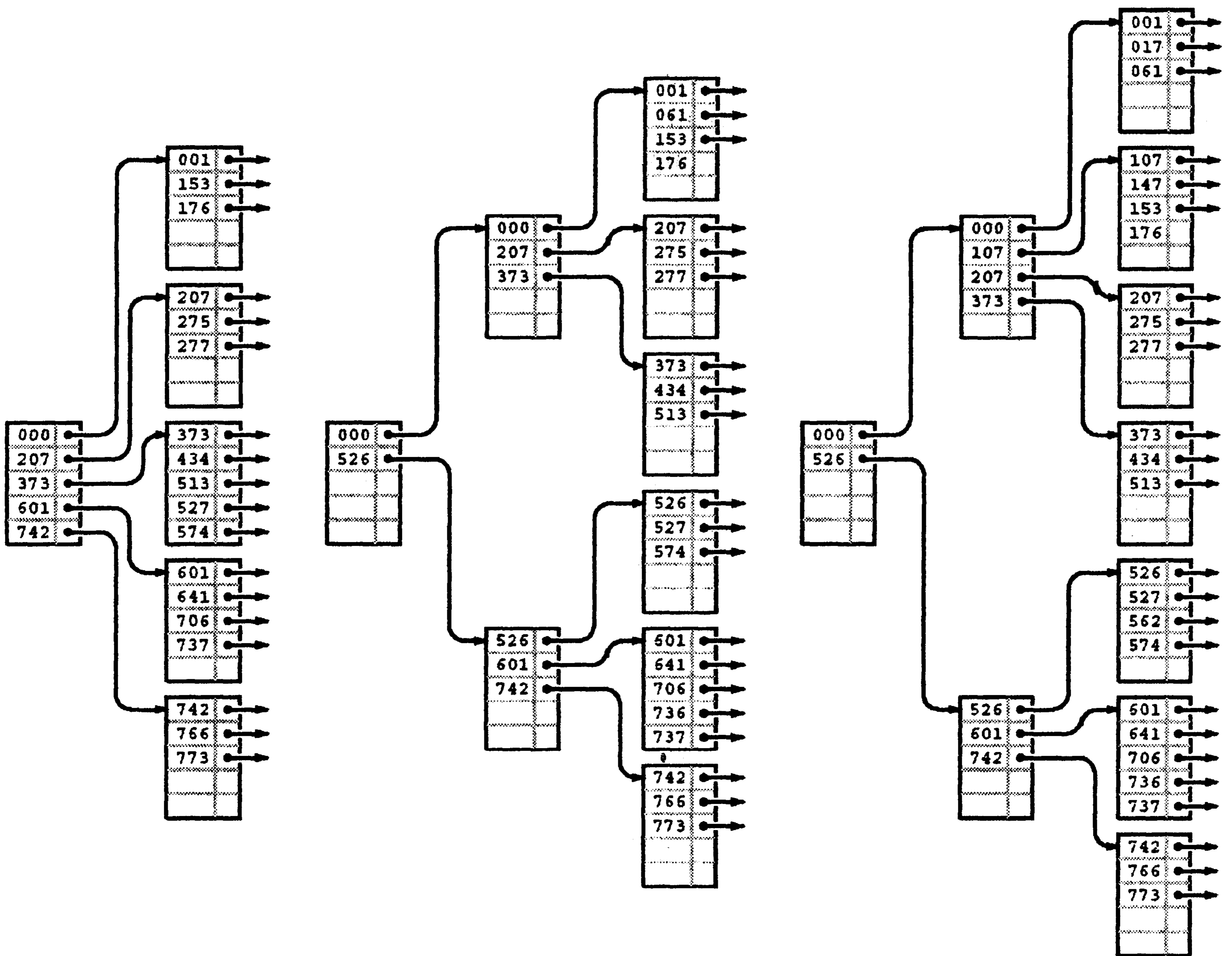


РИСУНОК 16.7 ПОСТРОЕНИЕ В-ДЕРЕВА, ЧАСТЬ 3

Продолжая приведенный пример, мы вставляем 13 ключей 574, 434, 641, 207, 001, 227, 061, 736, 526, 562, 017, 107 и 147 в крайнее справа В-дерево из рис. 16.6. Разделение узла происходит при вставке ключей 277 (рисунок слева), 526 (рисунок в центре) и 107 (рисунок справа). Разделение узла, вызванное вставкой ключа 526, приводит также к разделению индексной страницы и к увеличению высоты дерева на единицу.



После ознакомления с этими определениями и с рассмотренными примерами деревьев, код выполнения операции *search*, приведенный в программе 16.2, становится вполне понятным. Для внешних узлов мы выполняем просмотр массива узлов с целью нахождения ключа, который соответствует искомому ключу, возвращая связанный с ним элемент в случае успеха и нулевой элемент в случае неудачи. Для внутренних узлов мы выполняем просмотр массива узлов для нахождения связи с уникальным поддеревом, которое могло бы содержать искомый ключ.

### Программа 16.2 Поиск в B-дереве

Как обычно, эта реализация операции *search* для B-деревьев основывается на рекурсивной функции. Для внутренних узлов (имеющих положительную высоту) мы выполняем просмотр с целью отыскания ключа, который больше искомого, и осуществляем рекурсивный вызов поддерева, указанного предыдущей связью. Для внешних узлов (высота которых равно 0) мы выполняем просмотр, чтобы выяснить, содержат ли они элемент, ключ которого равен искомому.

**private:**

```
Item searchR(link h, Key v, int ht)
{ int j;
  if (ht == 0)
    for (j = 0; j < h->m; j++)
      { if (v == h->b[j].key)
        return h->b[j].item; }
  else
    for (j = 0; j < h->m; j++)
      if ((j+1 == h->m) || (v < h->b[j+1].key))
        return searchR(h->b[j].next, v, ht-1);
  return nullItem;
}
```

**public:**

```
Item search(Key v)
{ return searchR(head, v, HT); }
```

Программа 16.3 содержит реализацию операции *insert* для B-деревьев; кроме того, в ней применяется рекурсивный подход, используемый во многих других реализациях деревьев поиска, описанных в главах 13 и 15. Эта реализация является восходящей, поскольку проверка на предмет разделения узла выполняется *после* рекурсивного вызова, и, следовательно, первый разделяемый узел является внешним. Разделение требует передачи новой связи вверх родительскому узлу разделяемого узла, который, в свою очередь, может нуждаться в разделении и передаче связи его родительскому узлу и т.д., возможно, вплоть до корня дерева (при разделении корня создается новый корень с двумя дочерними поддеревьями). И наоборот, в реализации 2-3-4-дерева из программы 13.6 проверка на предмет разделения выполняется *перед* рекурсивным вызовом, и поэтому разделение выполняется в ходе перемещения вниз по дереву. Для B-деревьев можно было бы воспользоваться также нисходящим подходом (см. упражнение 16.10). Это различие между нисходящим и восходящим подходами во многих приложениях, использующих B-деревья, несущественно, поскольку эти деревья являются очень плоскими.

---

**Программа 16.3 Вставка в B-дерево**


---

Новые элементы вставляются за счет перемещения больших элементов вправо на одну позицию, как в сортировке вставками. Если вставка приводит к переполнению узла, мы вызываем функцию `split` для разделения узла на две половины, а затем вставляем связь с новым узлом во внутренний родительский узел, который также может разделяться; таким образом, влияние вставки может распространяться вплоть до корня.

```
private:
link insertR(link h, Item x, int ht)
{ int i, j; Key v = x.key(); entry<Item, Key> t;
  t.key = v; t.item = x;
  if (ht == 0)
    for (j = 0; j < h->m; j++)
      { if (v < h->b[j].key) break; }
  else
    for (j = 0; j < h->m; j++)
      if ((j+1 == h->m) || (v < h->b[j+1].key))
        { link u;
          u = insertR(h->b[j+1].next, x, ht-1);
          if (u == 0) return 0;
          t.key = u->b[0].key; t.next = u;
          break;
        }
    for (i = h->m; i > j; i--) h->b[i] = h->b[i-1];
    h->b[j] = t;
    if (++h->m < M) return 0; else return split(h);
}
public:
ST(int maxN)
{ N = 0; HT = 0; head = new node; }
void insert(Item item)
{ link u = insertR(head, item, HT);
  if (u == 0) return;
  link t = new node(); t->m = 2;
  t->b[0].key = head->b[0].key;
  t->b[1].key = u->b[0].key;
  t->b[0].next = head; t->b[1].next = u;
  head = t; HT++;
}
```

---

Код разделения узла показан в программе 16.4. В нем для переменной  $M$  используется четное значение, и каждый узел дерева может содержать только  $M - 1$  элемент. Этот подход позволяет вставлять  $M$ -тый элемент в узел *перед* разделением этого узла и значительно упрощает код, не оказывая особого влияния на затраты (см. упражнения 16.20 и 16.21). Для простоты в аналитических выкладках, приведенных далее в разделе, мы ограничиваем сверху количество элементов каждого узла ( $M$ ); действительные же различия весьма несущественны. В нисходящей реализации не пришлось бы прибегать к этой технологии, поскольку в таком случае наличие свободного места для вставки нового узла обеспечивается автоматически.

### Программа 16.4 Разделение узла В-дерева

Чтобы разделить узел в В-дереве, мы создаем новый узел, перемещаем большую половину ключей в новый узел, а затем изменяем значения счетчиков и устанавливаем служебные ключи в середине обоих узлов. В этой программе предполагается, что значение  $M$  является четным, а в каждом узле имеется лишняя позиция для элемента, который вызывает разделение. Другими словами, максимальное количество ключей в узле равно  $M-1$ , и когда количество ключей в узле достигает значения  $M$ , узел разделяется на два узла с  $M/2$  ключей в каждом.

```
link split(link h)
{ link t = new node();
  for (int j = 0; j < M/2; j++)
    t->b[j] = h->b[M/2+j];
  h->m = M/2; t->m = M/2;
  return t;
}
```

**Лемма 16.2** Для выполнения поиска или вставки в В-дереве порядка  $M$ , содержащем  $N$  элементов, требуется от  $\log_M N$  до  $\log_{M/2} N$  зондирований — число, которое на практике можно считать постоянным.

Эта лемма является следствием наблюдения, что все узлы во внутренней части В-дерева (узлы, которые не являются ни корнем, ни внешними узлами) имеют от  $M/2$  до  $M$  связей, поскольку они образованы в результате разделения полного узла, содержащего  $M$  ключей, и увеличиваться может только их количество (при разделении нижнего узла). В лучшем случае эти узлы образуют полное дерево порядка  $M$ , что немедленно дает указанный верхний предел (см. лемму 16.1). В худшем случае мы получаем полное дерево порядка  $M/2$ .

Когда  $M$  равно 1000, при  $N$  менее 125 миллионов, высота дерева меньше трех. В типовых ситуациях затраты можно уменьшить до двух зондирований, храня корневой узел во внутренней памяти. Для реализаций поиска на диске этот шаг можно предпринимать явно перед применением любого приложения, связанного с очень большим количеством поисков; в виртуальной памяти с кэшированием корневым узлом будет узел, который, скорее всего, должен храниться в быстрой памяти, поскольку обращение к нему выполняется наиболее часто.

Вряд ли можно рассчитывать на реализацию поиска, которая сможет гарантировать выполнение менее двух зондирований при выполнении операций *search* и *insert* в очень больших файлах, и В-деревья находят широкое применение, поскольку они позволяют приблизиться к этому идеалу. Подобная производительность и гибкость достигаются ценой наличия пустых ячеек внутри узлов, что может оказаться обременительным для очень больших файлов.

**Лемма 16.3** В-дерево порядка  $M$ , сконструированное из  $N$  случайных элементов, предположительно должно иметь около  $1.44N/M$  страниц.

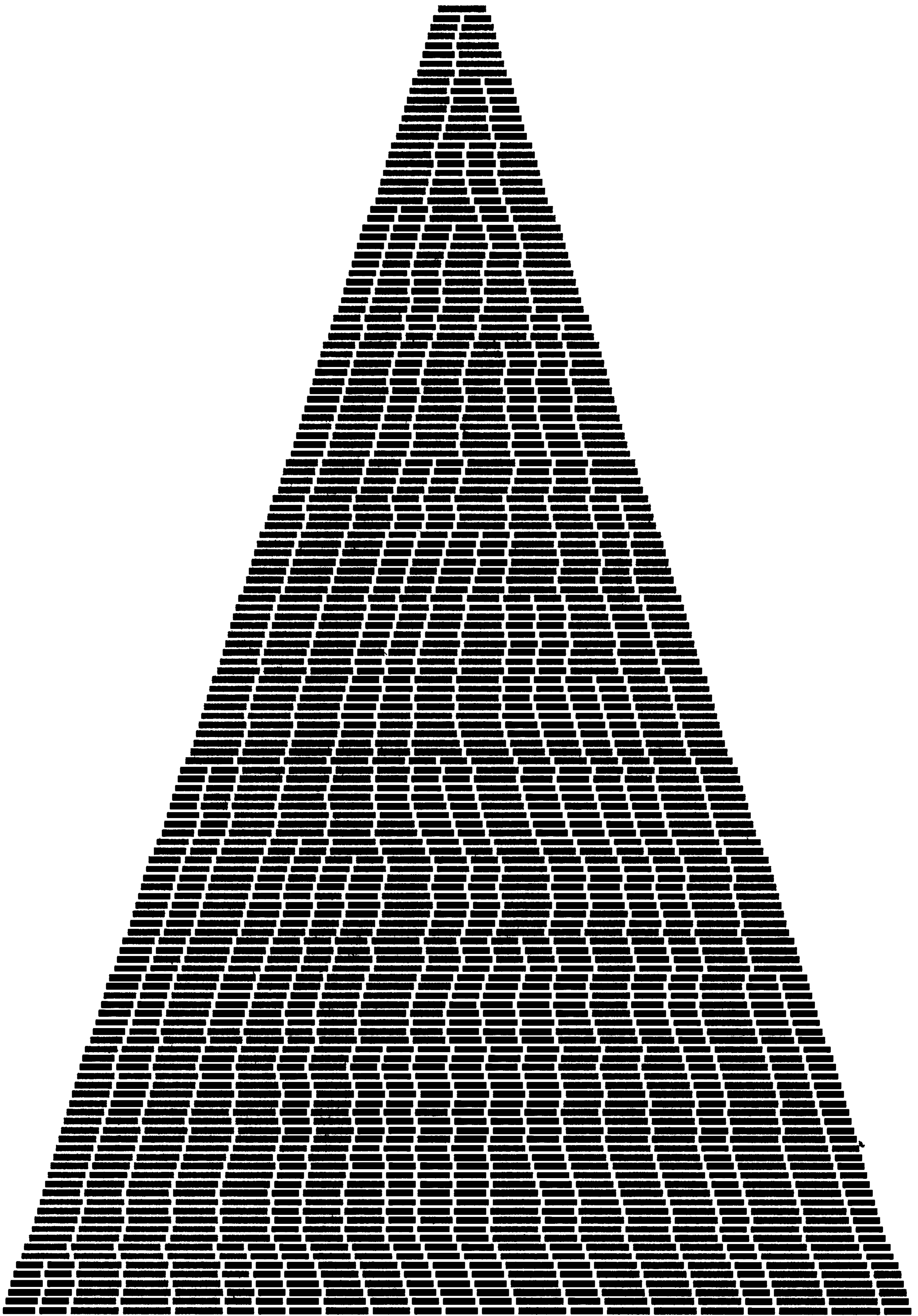
Яо (Yao) доказал это в 1979 г., прибегнув к математическому анализу, который выходит за рамки этой книги (см. раздел ссылок). Этот анализ основывается на анализе простой вероятностной модели, описывающей рост дерева. После того, как вставлены первые  $M/2$  узлов, в любой заданный момент времени имеется  $t_i$  вне-

шних страниц с  $i$  элементами, при  $M/2 \leq i \leq M$  и  $t_{M/2} + \dots + t_M = N$ . Поскольку вероятность попадания случайного ключа в любой интервал между узлами одинакова, вероятность попадания в узел с  $i$  элементами равна  $t_i/N$ . В частности, для  $i < M$  эта величина равна вероятности того, что количество внешних страниц с  $i$  элементами уменьшается на 1, а количество внешних страниц с  $(i + 1)$  элементами увеличивается на 1. Для  $i = 2M$  эта величина равна вероятности того, что количество внешних страниц с  $2M$  элементами уменьшается на 1, а количество внешних страниц с  $M$  элементами увеличивается на 2. Такой вероятностный процесс называется *Марковской цепью*. Результат, полученный Яо, основывается на анализе асимптотических свойств этой цепи.

Лемму 16.3 можно также доказать, написав программу для имитации вероятностного процесса (см. упражнение 16.11 и рис. 16.8 и 16.9). Конечно, можно было бы также просто построить случайные В-деревья и измерить их структурные характеристики. Вероятностную имитацию выполнить проще, чем произвести математический анализ или создать полную реализацию, кроме того, она служит важным инструментом изучения и сравнения вариантов алгоритмов (см., например, упражнение 16.16).

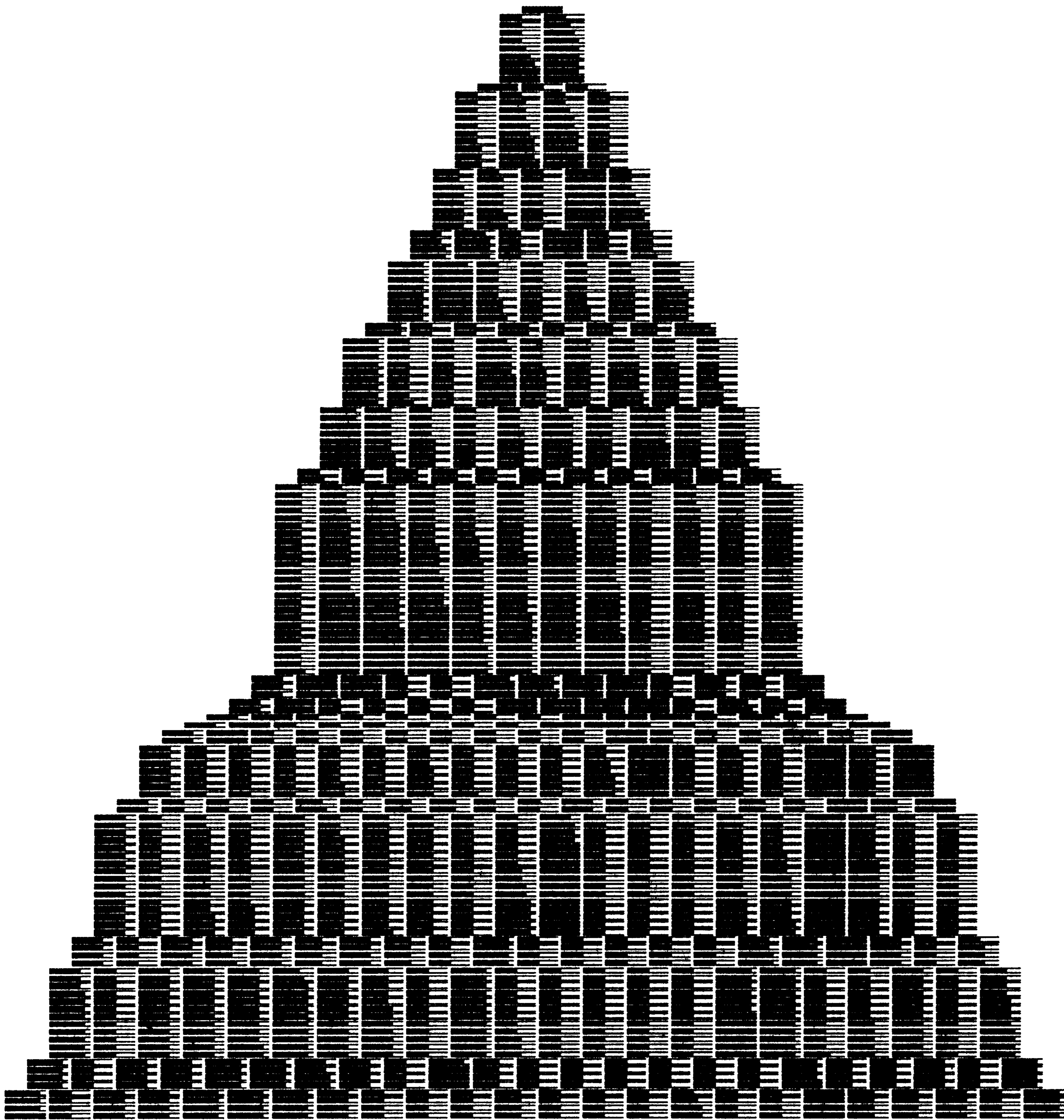
Реализации других операций таблиц символов аналогичны соответствующим реализациям для других ранее рассмотренных представлений с использованием деревьев, поэтому они остаются в качестве упражнений (см. упражнения 16.22–16.25). В частности, реализации операций *select* и *sort* очевидны, но, как обычно, правильная реализация операции *remove* может оказаться сложной задачей. Подобно операции *insert*, большинство операций *remove* заключаются в простом удалении элемента из внешней страницы и уменьшении значения ее счетчика, но что делать, если необходимо удалить элемент из узла, который имеет только  $M/2$  элементов? Естественный подход — найти для заполнения свободного места элементы в родственных узлах (возможно, с уменьшением количества узлов на единицу), но реализация усложняется, поскольку приходится отслеживать ключи, связанные с любыми перемещаемыми по узлам элементами. Во встречающихся на практике ситуациях обычно можно использовать значительно более простой подход, оставляя внешние узлы незаполненными, что не особенно снижает производительность (см. упражнение 16.25).

Многие вариации базовой абстракции В-дерева приходят на ум немедленно. Один класс вариаций позволяет экономить время за счет упаковки во внутренние узлы максимально возможного количества ссылок страниц, в результате чего коэффициент ветвления повышается, а дерево становится более плоским. Как уже отмечалось, в современных системах преимущества, получаемые в результате таких изменений, ограничены, поскольку стандартные значения параметров позволяют реализовать операции *search* и *insert* при помощи всего двух зондирований — эффективность, которую вряд ли удастся повысить. Другой класс вариантов повышает эффективность использования дискового пространства, перед разделением объединяя узлы с их родственными узлами. Упражнения 16.13–16.16 посвящены такому методу, который при работе со случайными ключами позволяет уменьшить дополнительно используемый объем дискового пространства с 44 до 23 процентов. Как обычно, выбор того или иного варианта зависит от свойств приложения. Помня о широком множестве различных ситуаций, в которых применимы В-деревья, мы не будем подробно рассматривать эти вопросы.



**РИСУНОК 16.8 РОСТ БОЛЬШОГО В-ДЕРЕВА**

*В этой имитации мы вставляем элементы со случайными ключами в первоначально пустое В-дерево, образованное страницами, которые могут содержать девять ключей и связей. Каждая линия отображает внешние узлы в виде сегментов, длина которых пропорциональна количеству элементов в данном узле. Большинство вставок выполняется во внешних узлах, которые не заполнены, что приводит к увеличению размера данного узла на 1. Когда вставка выполняется в заполненном внешнем узле, узел разделяется на два узла половинного размера.*

**РИСУНОК 16.9 РОСТ БОЛЬШОГО В-ДЕРЕВА С ОТОБРАЖЕНИЕМ ЗАНЯТОСТИ СТРАНИЦ**

*В этой версии рис. 16.8 показано, как страницы заполняются во время процесса роста В-дерева. Как и в предыдущем случае, большинство вставок приходится на страницы, которые не заполнены, лишь увеличивая их занятость на 1. Когда вставка приходится на полную страницу, страница разделяется на две полупустые страницы.*

Мы не сможем также рассмотреть подробности реализаций, поскольку это потребовало бы учета слишком многих зависящих от устройств и систем факторов. Как обычно, детальная разработка таких приложений — рискованное дело, и в современных системах желательно избегать наличия столь прихотливого и непереносимого кода, особенно, когда базовый алгоритм работает вполне успешно.

## Упражнения

- ▷ **16.5** Приведите содержимое 3-4-5-6-дерева, образованного в результате вставки ключей `EASYQUESTIONWITHPLENTYOFKEYS` в указанном порядке в первоначально пустое дерево.
- **16.6** Постройте рисунки, соответствующие рис. 16.5–16.7, иллюстрирующие процесс вставки ключей 516, 177, 143, 632, 572, 161, 774, 470, 411, 706, 461, 612, 761, 474, 774, 635, 343, 461, 351, 430, 664, 127, 345, 171 и 357 в указанном порядке в первоначально пустое дерево при  $M = 5$ .
- **16.7** Укажите высоту В-деревьев, образованных в результате вставки в указанном порядке ключей из упражнения 16.6 в первоначально пустое дерево для *каждого* значения  $M > 2$ .
- 16.8** Нарисуйте В-дерево, образованное в результате вставки 16 одинаковых ключей в первоначально пустое дерево при  $M = 4$ .
- **16.9** Нарисуйте 1-2-дерево, образованное в результате вставки ключей `EASYQUESTION` в первоначально пустое дерево. Объясните, почему 1-2-деревья не представляют практического интереса как сбалансированные деревья.
- **16.10** Измените реализацию вставки в В-дерево, приведенную в программе 16.3, чтобы в ней выполнялось разделение при перемещении вниз по дереву, подобно реализации вставки в 2-3-4-дереве (программа 13.6).
- **16.11** Напишите программу для вычисления среднего количества внешних страниц В-дерева порядка  $M$ , построенного путем  $N$  случайных вставок в первоначально пустое дерево при использовании вероятностного процесса, описанного после леммы 16.1. Выполните программу для  $M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **16.12** Предположите, что в трехуровневом дереве  $a$  связей могут храниться во внутренней памяти, от  $b$  до  $2b$  связей — в страницах, представляющих внутренние узлы, и от  $c$  до  $2c$  связей — в страницах, представляющих внешние узлы. Определите в виде функции от  $a, b$  и  $c$  максимальное количество элементов, которое может храниться в таком дереве.
- **16.13** Рассмотрите эвристический вариант *родственного разделения* В-деревьев (или *В\*-дерево*): когда требуется разделить узел, поскольку он содержит  $M$  записей, мы объединяем узел с его родственным узлом. Если родственный узел содержит  $k$  записей, при  $k < M - 1$ , мы перераспределяем элементы, помещая и в родственный, и в полный узел приблизительно по  $(M + k)/2$  записей. В противном случае мы создаем новый узел и помещаем в каждый узел дерева приблизительно по  $2M/3$  записей. Кроме того, мы позволяем корню разрастаться, чтобы в нем могло содержаться около  $4M/3$ , разделяя его и создавая новый корневой узел с двумя записями, когда его размер достигает этого предельного значения. Укажите верхние пределы количества зондирований, используемых для выполнения поиска

или вставки в В\*-дереве порядка  $M$ , содержащем  $N$  элементов. Сравните эти предельные значения с соответствующими предельными значениями для В-деревьев (см. лемму 16.2) при  $M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .

- **16.14** Разработайте реализацию операции *insert* в В\*-дереве (основанную на эвристическом методе родственного разделения).
- **16.15** Создайте рисунок, аналогичный рис. 16.8 для иллюстрации эвристического метода родственного разделения.
- **16.16** Выполните вероятностную имитацию (см. упражнение 16.11) с целью определения среднего количества страниц, задействованных при использовании эвристического метода родственного разделения при помощи построения В\*-дерева порядка  $M$  в результате вставки случайных узлов в первоначально пустое дерево, при  $M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **16.17** Создайте программу для восходящего построения индекса В-дерева, начиная с массива указателей страниц, содержащих от  $M$  до  $2M$  упорядоченных элементов.
- **16.18** Можно ли сконструировать индекс заполненных страниц за счет использования алгоритма вставки в В-дерево, рассмотренного в тексте раздела (программа 16.3)? Обоснуйте ответ.
- 16.19** Предположите, что много различных компьютеров имеют доступ к одному и тому же индексу, что позволяет нескольким программам практически одновременно предпринимать попытки вставки нового узла в одно и то же В-дерево. Поясните, почему в такой ситуации может быть предпочтительнее использовать нисходящие В-деревья, а не восходящие. Предположите, что каждая программа может (и делает это) задержать изменение другими программами любого узла, который она считывает и, возможно, изменяет впоследствии.
- **16.20** Измените реализацию В-дерева в программах 16.1—16.3, чтобы в одном узле дерева могло содержаться  $M$  элементов.
- ▷ **16.21** Постройте таблицу значений разностей  $\log_{999}N$  и  $\log_{1000}N$  для  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- ▷ **16.22** Реализуйте операцию *sort* для таблицы символов, основанной на использовании В-дерева.
- **16.23** Реализуйте операцию *select* для таблицы символов, основанной на использовании В-дерева.
- **16.24** Реализуйте операцию *remove* для таблицы символов, основанной на использовании В-дерева.
- **16.25** Реализуйте операцию *remove* для таблицы символов, основанной на применении В-дерева, при использовании простого метода, когда указанный элемент удаляется из внешней страницы (возможно, допустив, чтобы количество элементов на странице становилось меньше  $M/2$ ), но чтобы изменение не распространялось вверх по дереву, за исключением, возможно, настройки значений ключей, если удаленный элемент оказался наименьшим на данной странице.
- **16.26** Измените программы 16.2 и 16.3, чтобы внутри узлов использовался бинарный поиск (программа 12.6). Определите значение  $M$ , при котором время, затрачиваемое программой на построение таблицы символов за счет вставки  $N$  элемен-



тов со случайными ключами в первоначально пустую таблицу, минимально, при  $N = 10^3, 10^4, 10^5$  и  $10^6$ , и сравните полученные значения времени с соответствующими значениями для RB-деревьев (программа 13.6).

## 16.4 Расширяемое хеширование

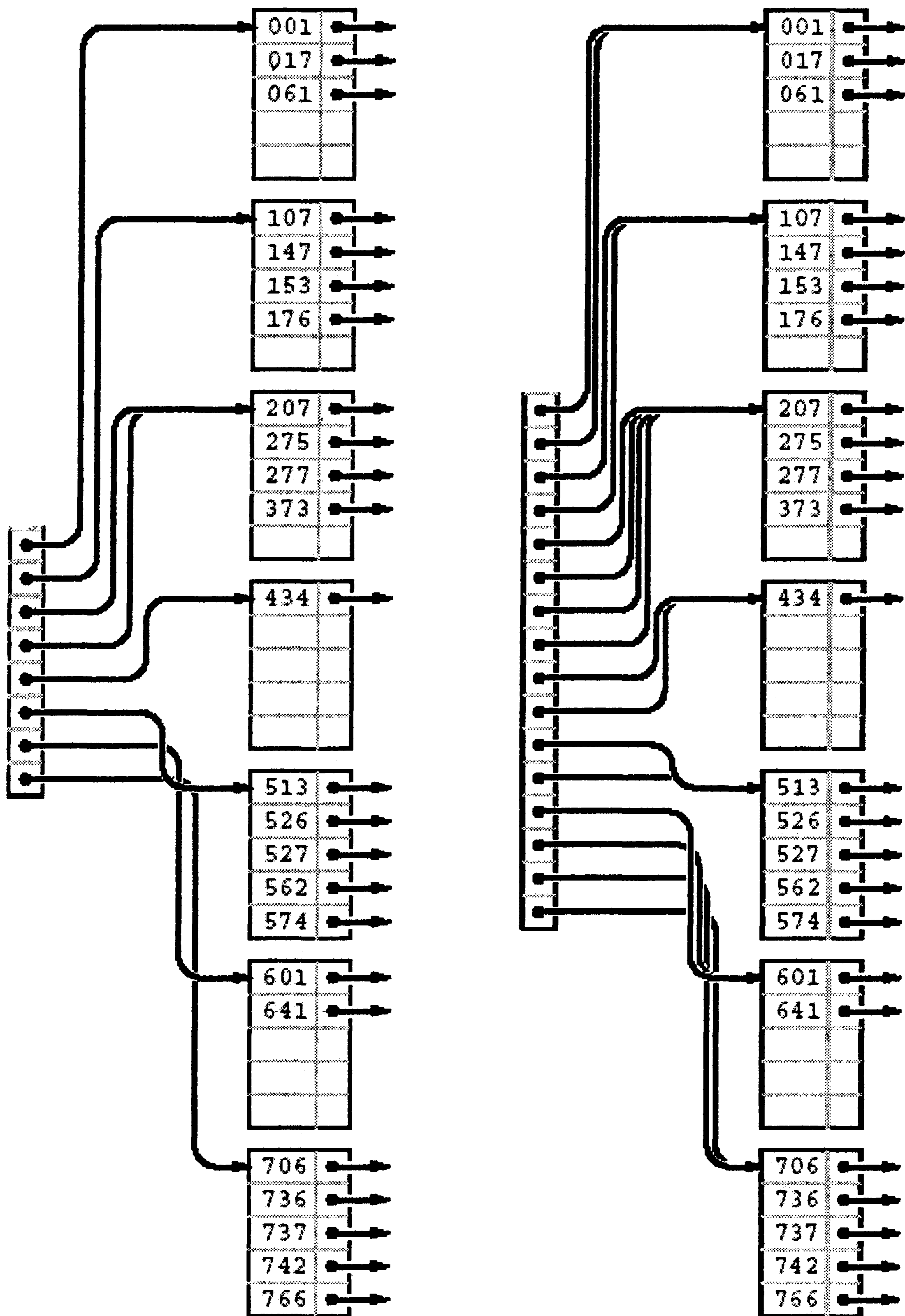
Альтернатива B-деревьям, расширяющая применение алгоритмов поразрядного поиска на внешний поиск, была разработана Фагином (Fagin), Нивергельтом (Nievergelt), Пиппенгером (Pippenger) и Стронгом (Strong) в 1978 г. Их метод, названный *расширяемым хешированием*, приводит к реализации операции *search*, для которой в типичных приложениях требуется всего одно-два зондирования. Для соответствующей реализации операции *insert* также (почти всегда) требуется всего одно-два зондирования.

Расширяемое хеширование объединяет свойства методов хеширования, использования многопозиционных trie-деревьев и последовательного доступа. Подобно методам хеширования, описанным в главе 14, расширяемое хеширование представляет собой рандомизированный алгоритм — прежде всего необходимо определить хеш-функцию, которая преобразует ключи в целые числа (см. раздел 14.1). Для простоты в этом разделе мы будем просто считать, что ключи являются случайными строками разрядов фиксированной длины. Подобно алгоритмам с использованием многопозиционных trie-деревьев, описанным в главе 15, расширяемое хеширование начинает поиск с использования ведущих разрядов ключей в качестве индексных указателей в таблицу, размер которой является степенью 2. Подобно алгоритмам с применением B-деревьев, при использовании расширяемого хеширования элементы хранятся на страницах, которые разделяются на две части при заполнении. Подобно методам индексного последовательного доступа расширяемое хеширование поддерживает каталог, указывающий, где можно найти страницу, содержащую соответствующие искомому ключу элементы. Поскольку оно объединяет эти знакомые свойства в одном алгоритме, расширяемое хеширование как нельзя более подходит для завершения знакомства с алгоритмами поиска.

Предположим, что количество доступных страниц диска является степенью 2 — скажем,  $2^d$ . Тогда можно поддерживать каталог  $2^d$  ссылок различных страниц, использовать  $d$  разрядов ключей для индексных указателей внутри каталога и хранить на одной и той же странице все ключи, первые  $k$  разрядов которых совпадают, как показано на рис. 16.10. Подобно тому, как это делалось в случае B-деревьев, элементы на страницах хранятся по порядку, и достигнув страницы, которая соответствует элементу с заданным искомым ключом, мы выполняем последовательный поиск.

Если удвоить размер каталога и клонировать каждый указатель, можно получить структуру, которую можно индексировать первыми 4 разрядами искомого ключа (рисунок справа). Например, последняя страница по-прежнему определяется как содержащая элементы с ключами, первые три разряда которых — 111, и она будет доступна через каталог, если первые 4 разряда искомого ключа — 1110 или 1111. Этот больший каталог может допускать увеличение таблицы.

Рисунок 16.10 иллюстрирует две базовых концепции, лежащие в основе расширяемого хеширования. Во-первых, нам не обязательно поддерживать  $2^d$  страниц.



**РИСУНОК 16.10 ИНДЕКСЫ СТРАНИЦ КАТАЛОГА**

При использовании каталога, состоящего из восьми записей, можно хранить до 40 ключей, храня все записи, первые 3 разряда которых совпадают, на одной странице, обратиться к которой можно через указатель, хранящийся в каталоге (рисунок слева). Запись 0 каталога содержит указатель на страницу, которая содержит все ключи, начинающиеся с 000; запись таблицы 1 содержит указатель на страницу, которая содержит все ключи начинающиеся с 001; запись таблицы 2 содержит указатель на страницу, которая содержит все ключи, начинающиеся с 010, и т.д. Если некоторые страницы заполнены не полностью, количество требующихся страниц можно уменьшить, организовав множество указателей на одну и ту же страницу. В данном примере (слева) ключ 373 находится на той же странице, что и ключи, начинающиеся с 2; эта страница определена как содержащая элементы с ключами, первые два разряда которых — 01.

То есть, может существовать несколько записей каталога, ссылающихся на одну и ту же страницу, что не повлияет на возможность быстро выполнять поиск в структуре путем объединения на одной странице ключей с различными значениями, первые  $d$  разрядов которых совпадают, в то же время позволяя находить страницу, содержащую заданный ключ, за счет использования ведущих разрядов ключа для индексации внутри каталога. Во-вторых, для увеличения емкости таблицы можно удвоить размер каталога.

В частности, структура данных, используемая для расширяемого хеширования, значительно проще использованной для В-деревьев. Она состоит из страниц, которые содержат до  $M$  элементов, и каталога, содержащего  $2^d$  указателей на страницы (программа 16.5). Указатель в ячейке каталога  $x$  ссылается на страницу, которая содержит все элементы, ведущие  $d$  разрядов которых равны  $x$ . Значение  $d$  выбирается достаточно большим, чтобы на каждой странице гарантированно хранилось менее  $M$  элементов. Реализация операции *search* проста: мы используем ведущие  $d$  разрядов ключа для индексации внутри каталога, что обеспечивает доступ к странице, которая содержит любые элементы с соответствующими ключами, а затем выполняем последовательный поиск такого элемента на данной странице (см. программу 16.6).

### Программа 16.5 Структуры данных расширяемого хеширования

Расширяемая хеш-таблица — это каталог ссылок на страницы (подобно внешним узлам в В-деревьях), который содержит до  $2M$  элементов. Каждая страница содержит также счетчик ( $m$ ) количества элементов на странице и целочисленное значение ( $k$ ), указывающее количество ведущих разрядов, которые должны совпадать в ключах элементов. Как обычно,  $N$  — количество элементов в таблице. Переменная  $d$  определяет количество разрядов, которые используются для индексации в каталоге, а  $D$  — количество записей в каталоге; таким образом,  $D=2^d$ . Вначале таблица устанавливается соответствующей каталогу размера 1, который указывает на пустую страницу.

```
template <class Item, class Key>
class ST
{
private:
    struct node
    { int m; Item b[M]; int k;
      node() { m = 0; k = 0; }
    };
    typedef node *link;
    link* dir;
    Item nullItem;
    int N, d, D;
public:
    ST(int maxN)
    { N = 0; d = 0; D = 1;
      dir = new link[D];
      dir[0] = new node;
    }
};
```

**Программа 16.6 Поиск в таблице расширяемого хеширования**

Поиск в таблице расширяемого хеширования заключается всего лишь в использовании ведущих разрядов ключа для индексирования внутри каталога и выполнении на указанной странице последовательного поиска элемента, ключ которого равен искомому. Единственное выдвигаемое при этом требование — каждая запись каталога должна ссылаться на страницу, которая гарантированно содержит все элементы таблицы символов, начинающиеся с указанных разрядов.

```
private:
    Item search(link h, Key v)
    {
        for (int j = 0; j < h->m; j++)
            if (v == h->b[j].key()) return h->b[j];
        return nullItem;
    }
public:
    Item search(Key v)
    { return search(dir[bits(v, 0, d)], v); }
```

Для поддержки операции *insert* структура данных должна быть несколько более сложной, но одно из ее свойств заключается в том, что этот алгоритм поиска успешно работает без каких-либо модификаций. Чтобы обеспечить поддержку операции *insert*, необходимо ответить на следующие вопросы:

- Что делать, когда количество элементов на странице превышает ее емкость?
- Какой размер каталога следует использовать?

Например, в примере, приведенном на рис. 16.10, нельзя было бы использовать значение  $d = 2$ , поскольку некоторые страницы оказались бы переполненными, и нельзя было бы использовать значение  $d = 5$ , поскольку слишком много страниц оказались бы пустыми. Как обычно, наибольший интерес вызывает поддержка операции *insert* для АТД таблицы символов, чтобы, например, структура могла постепенно разрастаться по мере выполнения ряда чередующихся операций *search* и *insert*. Принятие такой точки зрения соответствует уточнению первого вопроса:

- Что делать, когда необходимо *вставить* элемент в заполненную страницу?

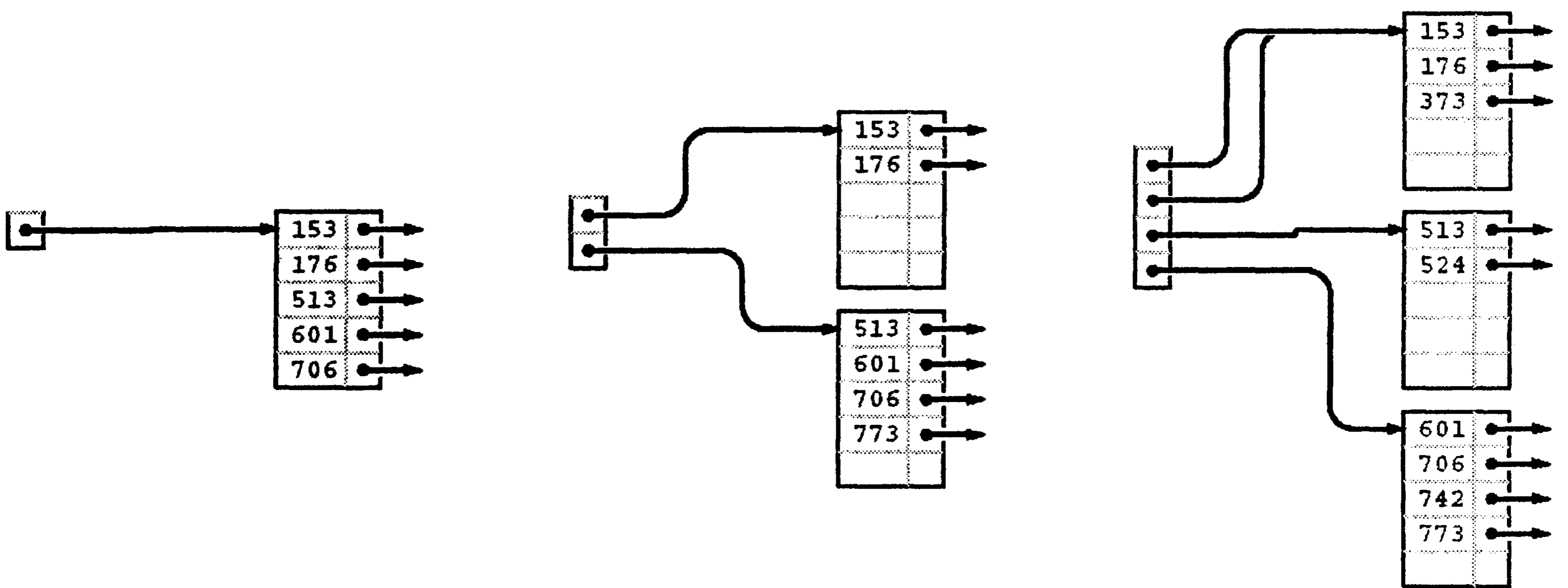
Например, в примере, представленном на рис. 16.10, нельзя было бы вставить элемент, ключ которого начинается с 5 или 7, поскольку соответствующие страницы заполнены.

**Определение 16.3** *Расширяемая хеш-таблица* порядка  $d$  представляет собой каталог из  $2^d$  ссылок на страницы, которые содержат до  $M$  элементов с ключами. Первые  $k$  разрядов элементов на каждой странице совпадают, а каталог содержит  $2^{d-k}$  указателей на страницу, начинающихся с ячейки, указанной ведущими  $k$  разрядами в ключах на страницах.

Некоторые  $d$ -разрядные последовательности могут не появляться ни в одном из ключей. Соответствующие записи каталога оставлены в определении 16.3 не определенными, хотя существует естественный способ организации указателей на пустые страницы, который вскоре будет рассмотрен.

Для поддержания этих характеристик в процессе разрастания таблицы мы используем две базовые операции: *разделение страницы*, при котором некоторые ключи с полной страницы переносятся на другую страницу, и *разделение каталога*, при котором размер каталога удваивается, а значение  $d$  увеличивается на 1. В частности, при заполнении страницы мы разделяем ее на две, используя самую левую разрядную позицию, где различаются ключи, для определения элементов, которые должны быть перемещены на новую страницу. При разделении страницы мы соответствующим образом изменяем указатели каталога, при необходимости удваивая размер каталога.

Как обычно, лучший способ понять алгоритм заключается в отслеживании его работы при вставке набора ключей в первоначально пустую таблицу. Вскоре каждая из ситуаций, которые должен обрабатывать алгоритм, проявляется в простейшей форме, и принципы, лежащие в основе алгоритма, становятся понятными. Построение расширяемой хеш-таблицы для набора из 25 восьмеричных ключей, рассматриваемого в этой главе, показано на рис. 16.11–16.13. Подобно тому, как это имело место в В-деревьях, большинство вставок не приводит к переполнению: они просто добавляют ключ к странице. Поскольку процесс начинается с одной страницы, а завершается восемью, можно предположить, что семь вставок вызвали разделение страниц. Поскольку в начале размер каталога равен 1, а в конце — 16, можно предположить, что четыре вставки привели к разделению каталога.

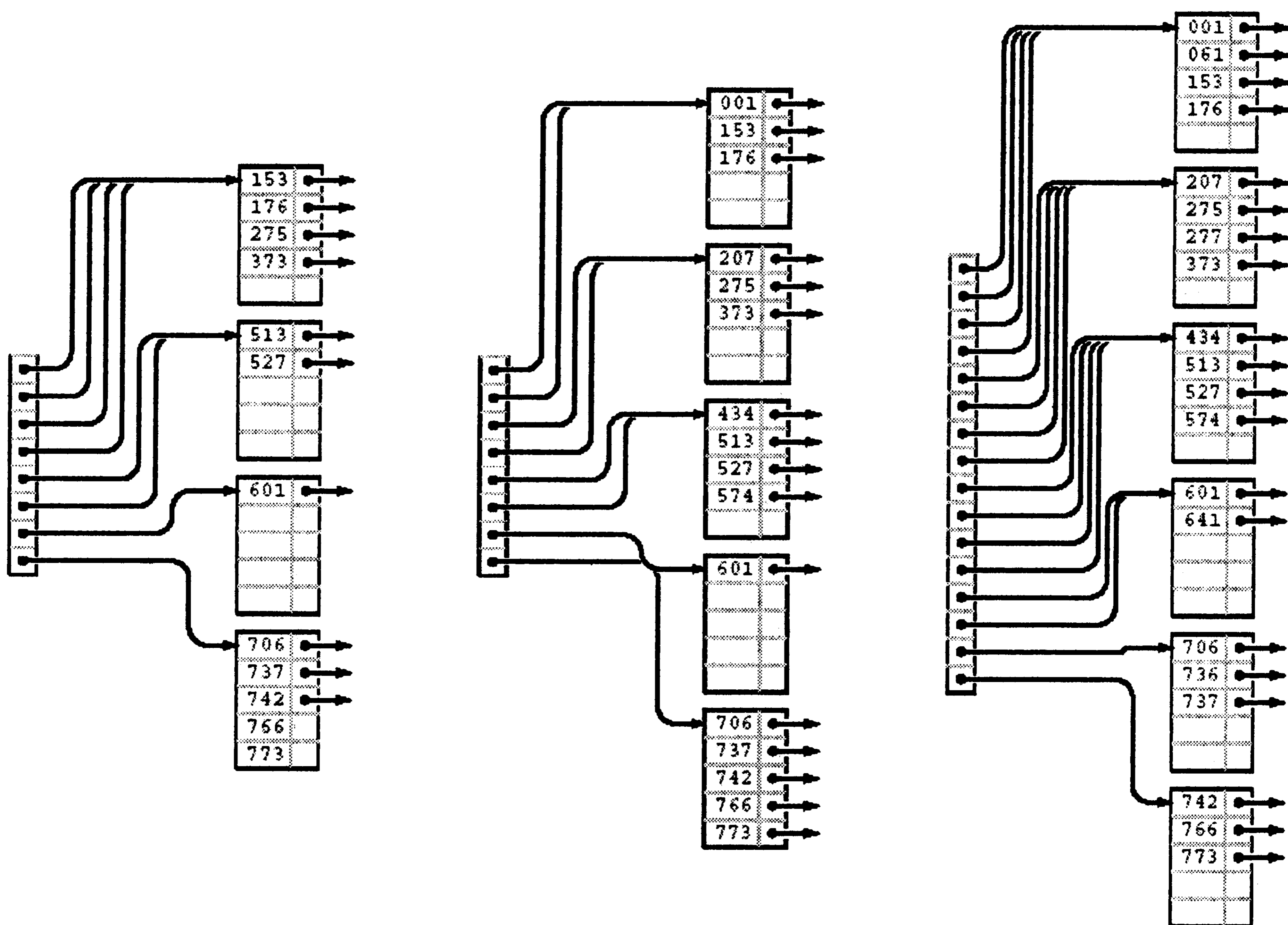


**РИСУНОК 16.11 ПОСТРОЕНИЕ РАСШИРЯЕМОЙ ХЕШ-ТАБЛИЦЫ, ЧАСТЬ 1**

Как и в В-деревьях, первые пять вставок в расширяемую хеш-таблицу приходятся на одну страницу (рисунок слева). Затем, при вставке ключа 773, мы выполняем разделение на две страницы (одна содержит все ключи, начинающиеся с 0 разряда, другая — все ключи, начинающиеся с 1 разряда) и удваиваем размер каталога, чтобы он содержал по одному указателю на каждую из страниц (рисунок в центре). Ключ 742 вставляется в нижнюю страницу (поскольку он начинается с 1 разряда), а ключ 373 — в верхнюю страницу (поскольку он начинается с 0 разряда), но затем нижнюю страницу приходится разделить, чтобы можно было поместить ключ 524. Для выполнения этого разделения все элементы, ключи которых начинаются с разрядов 10, помещаются на одну страницу, а все элементы, ключи которых начинаются с разрядов 11 — на другую, после чего размер каталога снова удваивается, чтобы в нем могли поместиться указатели на обе эти страницы (рисунок справа). Каталог содержит два указателя на страницу, содержащую элементы с ключами, которые начинаются с 0 разряда: один для ключей, которые начинаются с разрядов 00, и другой для ключей, которые начинаются с разрядов 01.

**Лемма 16.4** *Расширяемая хеш-таблица, построенная из набора ключей, зависит только от значений этих ключей и не зависит от порядка их вставки.*

Давайте рассмотрим trie-дерево, соответствующее ключам (см. лемму 15.2), в котором каждый внутренний узел помечен количеством элементов в его поддереве. Внутренний узел соответствует странице в расширяемой хеш-таблице тогда и только тогда, когда его метка меньше  $M$ , а метка его родительского узла не меньше чем  $M$ . Все элементы, расположенные ниже этого узла, попадают на данную страницу. Если узел находится на уровне  $k$ , он соответствует  $k$ -разрядной последовательности, полученной из пути trie-дерева обычным способом, а все записи в каталоге расширяемой хеш-таблицы, индексы которых начинаются с этой  $k$ -разрядной последовательности, содержат указатели на соответствующую страницу. Размер каталога определяется наибольшим уровнем всех внутренних узлов в trie-дереве, соответствующих страницам. Таким образом, trie-дерево можно преобразовать в расширяемую хеш-таблицу независимо от порядка вставки элементов, и это свойство сохраняется в качестве следствия леммы 15.2.



**РИСУНОК 16.12** ПОСТРОЕНИЕ РАСШИРЯЕМОЙ ХЕШ-ТАБЛИЦЫ, ЧАСТЬ 2

*Мы вставляем ключи 766 и 275 в крайнее справа B-дерево, показанное на рис. 16.11, без выполнения какого-либо разделения узлов (рисунок слева). Затем, при вставке ключа 737, нижняя страница разделяется и это, поскольку существует только одна связь с нижней страницей, приводит к разделению каталога (рисунок в центре). Затем мы вставляем ключи 574, 434, 641 и 207, что приводит к разделению верхней страницы (рисунок справа)*

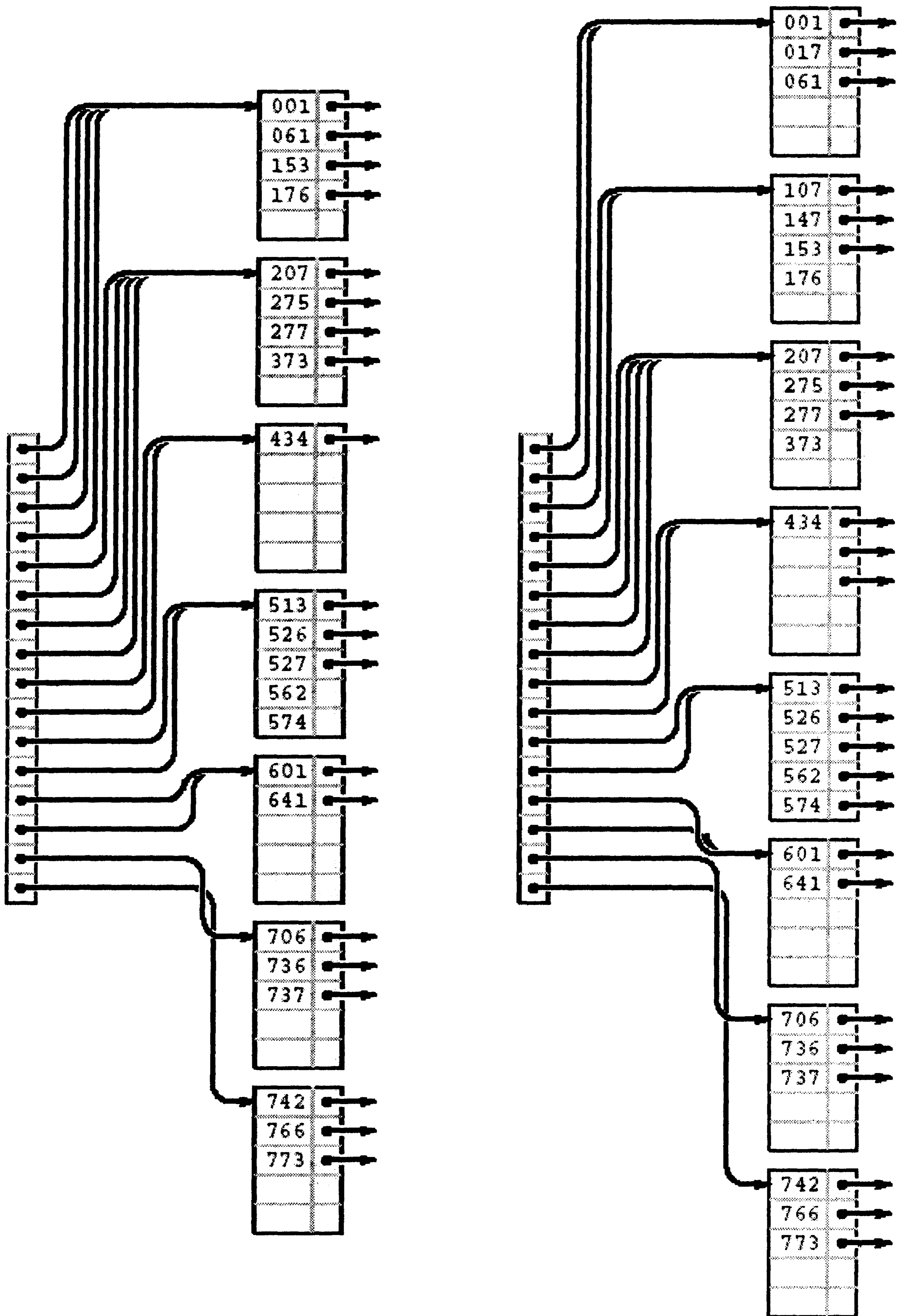


РИСУНОК 16.13 ПОСТРОЕНИЕ РАСШИРЯЕМОЙ ХЕШ-ТАБЛИЦЫ, ЧАСТЬ 3

Продолжая пример, приведенный на рис. 16.11 и 16.12, мы вставляем 5 ключей 526, 562, 017, 107 и 147 в крайнее справа B-дерево, отображенное на рис. 16.6. Разделение узлов происходит при вставке ключей 526 (рисунок слева) и 107 (рисунок справа).

Программа 16.7 — реализация операции *insert* для расширяемой хеш-таблицы. Прежде всего, мы обращаемся к странице, которая могла бы содержать искомый ключ, посредством единственной ссылки к каталогу, как это делалось при поиске. Затем мы вставляем в нее новый элемент, как это делалось для внешних узлов в В-деревьях (см. программу 16.2). Если в результате этой вставки в узле оказывается  $M$  элементов, мы вызываем функцию разделения, как это делалось для В-деревьев, правда, на этот раз она сложнее. Каждая страница содержит  $k$  ведущих разрядов, которые заведомо совпадают в ключах всех элементов на данной странице, и, поскольку разряды нумеруются слева направо, начиная с 0,  $k$  определяет также индекс разряда, который необходимо проверять для определения способа разделения элементов.

### Программа 16.7 Вставка в расширяемую хеш-таблицу

Чтобы вставить элемент в расширяемую хеш-таблицу, мы выполняем поиск; затем мы вставляем элемент в указанную страницу; далее, разделяем страницу, если вставка вызывает переполнение. Общая схема не отличается от используемой для В-деревьев, но алгоритмы поиска и разделения иные. Функция разделения создает новый узел, затем проверяет  $k$ -тый разряд (считая слева) ключа каждого элемента: если разряд равен 0, элемент остается в старом узле; если он равен 1, элемент перемещается в новый узел. Значение  $k + 1$  присваивается полю "заведомо идентичных ведущих разрядов" обоих узлов после разделения. Если этот процесс не приводит к помещению по меньшей мере одного ключа в каждом узле, разделение выполняется снова, пока подобным образом не будут разделены все элементы. В конце процесса мы вставляем указатель нового узла в каталог.

```
private:
void split(link h)
{ link t = new node;
  while (h->m == 0 || h->m == M)
  {
    h->m = t->m = 0;
    for (int j = 0; j < M; j++)
      if (bits(h->b[j].key(), h->k, 1) == 0)
        h->b[h->m++] = h->b[j];
      else t->b[t->m++] = h->b[j];
    t->k = ++(h->k);
  }
  insertDIR(t, t->k);
}
void insert(link h, Item x)
{ int j; Key v = x.key();
  for (j = 0; j < h->m; j++)
    if (v < h->b[j].key()) break;
  for (int i = (h->m)++; i > j; i--)
    h->b[i] = h->b[i-1];
  h->b[j] = x;
  if (h->m == M) split(h);
}
public:
void insert(Item x)
{ insert(dir[bits(x.key(), 0, d)], x); }
```



Следовательно, для разделения страницы мы создаем новую страницу, затем помещаем все элементы, для которых данный разряд равен 0, на старую страницу, а все элементы, для которых он равен 1 — на новую страницу, и, наконец, устанавливаем значение счетчика разрядов равным  $k + 1$  для обеих страниц. Возможен случай, когда все ключи имеют одинаковое значение разряда  $k$ , что привело бы к наличию заполненного узла. В этом случае мы просто перешли бы к следующему разряду, продолжая процесс до тех пор, пока каждая страница не будет содержать, по меньшей мере, один элемент. Со временем процесс должен завершиться, *если только не имеется  $M$  значений одного и того же ключа*. Вскоре мы рассмотрим и этот случай.

Как и в случае В-деревьев, на каждой странице остается свободное место, чтобы можно было выполнять разделение после вставки; это упрощает код. Следует снова отметить, что эта технология имеет небольшое практическое значение, и ее влиянием можно пренебречь при анализе.

При создании новой страницы в каталог необходимо вставить указатель на нее. Выполняющий эту вставку код представлен в программе 16.8. Простейшим является случай, когда перед вставкой каталог содержит ровно два указателя на страницу, которая разделяется. В этом случае нужно всего лишь обеспечить, чтобы второй указатель ссылался на новую страницу. Если количество разрядов  $k$ , которое требуется для различения ключей на новой странице, превышает количество разрядов  $d$ , имеющееся для доступа к каталогу, размер каталога нужно увеличить, чтобы в нем могла поместиться новая запись. И, в завершение, указатели каталога обновляются соответствующим образом.

### Программа 16.8 Вставка в каталог расширяемого хеширования

Этот обманчиво простой код лежит в основе процесса расширяемого хеширования. У нас имеется связь  $t$  с узлом, содержащим элементы, первые  $k$  разрядов которых совпадают, которую требуется вставить в каталог. В простейшем случае, когда значения  $d$  и  $k$  равны, достаточно просто поместить  $t$  в  $d[x]$ , где  $x$  — значение первых  $d$  разрядов  $t \rightarrow b[0]$  (и всех остальных элементов на странице). Если  $k$  больше  $d$ , размер каталога следует удвоить, пока не будет достигнуто равенство значений  $d$  и  $k$ . Если  $k$  меньше  $d$ , необходимо установить более одного указателя — в первом цикле `for` вычисляется количество указателей, которые необходимо установить ( $2^{d-k}$ ), а во втором выполняется собственно установка.

```
void insertDIR(link t, int k)
{ int i, m, x = bits(t->b[0].key(), 0, k);
  while (d < k)
  { link *old = dir;
    d += 1; D += D;
    dir = new link[D];
    for (i = 0; i < D; i++) dir[i] = old[i/2];
    if (d < k) dir[bits(x, 0, d)^1] = new node;
  }
  for (m = 1; k < d; k++) m *= 2;
  for (i = 0; i < m; i++) dir[x*m+i] = t;
}
```

Если более  $M$  элементов имеют дублированные ключи, таблица переполняется и программа 16.7 входит в бесконечный цикл, пытаясь найти способ различения ключей. С этим же тесно связана проблема, заключающаяся в том, что каталог может

оказаться неоправданно большим, если ключи имеют излишнее количество совпадающих ведущих разрядов. Для файлов, которые имеют большое количество дублированных ключей или длинные цепочки разрядных позиций, в которых они совпадают, эта ситуация сродни излишнему времени, требуемому для выполнения поразрядной сортировки сначала по старшей цифре. Для преодоления этих проблем приходится рассчитывать на рандомизацию, обеспечиваемую хеш-функцией (см. упражнение 16.43). При наличии большого количества дублированных ключей, даже при использовании хеширования, приходится предпринимать экстраординарные действия, поскольку хеш-функции отображают равные ключи на равные хеш-значения. Дублированные ключи могут сделать каталог неестественно большим; кроме того, алгоритм полностью разрушается при наличии большего количества равных ключей, чем помещается на одной странице. Следовательно, прежде чем использовать этот код, необходимо добавить проверки, предотвращающие возникновение упомянутых условий (см. упражнение 16.35).

С точки зрения производительности наибольший интерес представляют количество используемых страниц (как и в случае В-деревьев) и размер каталога. Для этого алгоритма рандомизация обеспечивается хеш-функциями, поэтому характеристики производительности, определенные для среднего случая, сохраняются для любой последовательности  $N$  различных вставок.

**Лемма 16.5** *При использовании страниц, которые могут содержать  $M$  элементов, для реализации расширяемого хеширования для файла, содержащего  $N$  элементов, в среднем требуется около  $1.44(N/M)$  страниц. Ожидаемое количество записей в каталоге приблизительно равно  $3.92(N^{1/M})(N/M)$ .*

Этот (достаточно обоснованный) результат дополняет анализ trie-деревьев, кратко рассмотренный в предыдущей главе (см. раздел ссылок). Точные значения констант для количества страниц и размера каталога соответственно равны  $lge = 1/\ln 2$  и  $e lge = e/\ln 2$ , поэтому точные значения величин колеблются вокруг указанных средних значений. Это не должно вызывать удивления, поскольку, например, размер каталога должен являться степенью 2 — факт, который в результате должен приниматься во внимание.

Обратите внимание, что размер каталога возрастает быстрее, нежели линейно по отношению к  $N$ , особенно при малых значениях  $M$ . Однако, для значений  $N$  и  $M$ , представляющих практический интерес, значение  $N^{1/M}$  достаточно близко к 1, поэтому реально можно ожидать, что каталог будет иметь около  $4(N/M)$  записей.

Мы приняли, что каталог — это массив указателей. Его можно хранить в памяти или, если он слишком велик, в памяти можно хранить корневой узел, который указывает местонахождение страниц каталога, используя такую же схему индексирования. Или же можно добавить еще один уровень, индексируя первый уровень, скажем, по первым 10 разрядам, а второй — по остальным разрядам (см. упражнение 16.36).

Подобно тому, как это делалось для В-деревьев, реализация остальных операций таблицы символов оставлена в качестве упражнений (см. упражнения 16.38 и 16.41). Так же как и в случае В-деревьев, правильная реализация операции *remove* представляет собой сложную задачу, но разрешение наличия незаполненных страниц — легко реализуемая альтернатива, которая может оказаться эффективной во многих возникающих на практике ситуациях.

## Упражнения

- ▷ **16.27** Сколько страниц были бы пустыми, если бы на рис. 16.10 использовался каталог, размер которого равен 32?
- 16.28** Постройте рисунки, соответствующие рис. 16.11—16.13, иллюстрирующие процесс вставки ключей 562, 221, 240, 771, 274, 233, 401, 273 и 201 в указанном порядке в первоначально пустое дерево, при  $M = 5$ .
- **16.29** Постройте рисунки, соответствующие рис. 16.11—16.13, иллюстрирующие процесс вставки ключей 562, 221, 240, 771, 274, 233, 401, 273 и 201 в обратном порядке в первоначально пустое дерево, при  $M = 5$ .
- **16.30** Предположите, что имеется массив упорядоченных элементов. Опишите, как можно было бы определить размер каталога расширяемой хеш-таблицы, соответствующей этому набору элементов.
- **16.31** Создайте программу, которая строит расширяемую хеш-таблицу из массива упорядоченных элементов, выполняя два прохода по элементам: один для определения размера каталога (см. упражнение 16.30) и второй для распределения элементов по страницам и заполнения каталога.
- **16.32** Приведите набор ключей, для которого соответствующая расширяемая хеш-таблица имеет каталог размером 16 при размещении восьми указателей на одной странице.
- **16.33** Создайте для расширяемого хеширования рисунок, подобный рис. 16.8.
- **16.34** Создайте программу для вычисления среднего количества внешних страниц и среднего размера каталога для расширяемой хеш-таблицы, построенной в результате  $N$  случайных вставок в первоначально пустое дерево, при емкости страницы  $M$ . Вычислите долю пустого пространства в процентах при  $M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- 16.35** Добавьте в программу 16.7 соответствующие проверки с целью предотвращения неправильной работы в случае вставки в таблицу слишком большого количества дублированных ключей или ключей, у которых совпадет слишком много ведущих разрядов.
- **16.36** Измените реализацию расширяемого хеширования, приведенную в программах 16.5—16.7, чтобы в ней использовался двухуровневый каталог, в каждом узле которого содержится не более  $M$  указателей. Обратите особое внимание на действия, предпринимаемые, когда каталог впервые разрастается с одноуровневого до двухуровневого.
- **16.37** Измените реализацию расширяемого хеширования, приведенную в программах 16.5—16.7, чтобы в структуре данных на одной странице могло существовать  $M$  элементов.
- **16.38** Реализуйте операцию *sort* для расширяемой хеш-таблицы.
- **16.39** Реализуйте операцию *select* для расширяемой хеш-таблицы.
- **16.40** Реализуйте операцию *remove* для расширяемой хеш-таблицы.
- **16.41** Реализуйте операцию *remove* для расширяемой хеш-таблицы, используя метод, указанный в упражнении 16.25.

- **16.42** Разработайте версию расширяемого хеширования, которая разделяет страницы при разделении каталога, чтобы каждый указатель каталога указывал на уникальную страницу. Проведите эксперименты с целью сравнения производительности этой реализации с производительностью стандартной реализации.
- **16.43** Экспериментально определите количество случайных чисел, которые предположительно будут сгенерированы прежде, чем будет найдено более  $M$  чисел с одинаковыми  $d$  начальными разрядами, при  $M = 10, 100$  и  $1000$  и при  $1 \leq d \leq 20$ .
- **16.44** Измените хеширование с отдельным связыванием (программа 14.3), чтобы в нем использовалась хеш-таблица размером  $2M$ , а элементы хранились на страницах размером  $2M$ . Другими словами, когда страница заполняется, она связывается с новой пустой страницей, чтобы каждая запись хеш-таблицы указывала на связный список страниц. Экспериментально определите среднее количество зондирований, необходимое для выполнения поиска после построения таблицы из  $N$  элементов со случайными ключами, при  $M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ .
- **16.45** Измените двойное хеширование (программа 14.6), чтобы в нем использовались страницы размером  $2M$  при интерпретации обращений к полным страницам в качестве "коллизий". Экспериментально определите среднее количество зондирований, необходимое для выполнения поиска после построения таблицы из  $N$  элементов со случайными ключами, при  $M = 10, 100$  и  $1000$  и  $N = 10^3, 10^4, 10^5$  и  $10^6$ , при начальном размере таблицы равном  $3N/2M$ .

## 16.5 Перспективы

Наиболее важное применение рассмотренных в этой главе методов — это построение индексов для очень больших баз данных, поддерживаемых во внешней памяти, например, в дисковых файлах. Хотя рассмотренные фундаментальные алгоритмы обладают большими возможностями, разработка реализации файловой системы, основанной на использовании В-деревьев или расширяемого хеширования представляет собой сложную задачу. Во-первых, приведенные в этом разделе программы на C++ нельзя использовать непосредственно — они должны быть модифицированы для считывания и ссылки на дисковые файлы. Во-вторых, необходимо быть уверенным, что параметры алгоритма (например, размеры страницы и каталога) подобраны в соответствии с характеристиками конкретного используемого аппаратного обеспечения. В-третьих, следует уделить внимание надежности и выявлению и исправлению ошибок. Например, необходимо иметь возможность убедиться в целостности структуры данных и принять решение о том, как исправлять любые из возможных ошибок. Подобные системные факторы являются критичными и выходят за рамки этой книги.

С другой стороны, при наличии среды программирования, которая поддерживает виртуальную память, рассмотренные реализации на языке C++ можно непосредственно использовать в ситуации, когда необходимо выполнять очень большое количество операций применительно к очень большим таблицам символов. В первом приближении можно сказать, что при каждом обращении к странице такая система будет помещать эту страницу в *кэш*, в котором эффективно обрабатываются ссылки на данные этой страницы. При обращении к странице, которая отсутствует в *кэш*-памяти, система должна считать страницу из внешней памяти, поэтому в первом при-

ближении случаи отсутствия страницы в кэше можно считать эквивалентом используемой нами меры затрат, выражаемой зондированиями.

При использовании В-деревьев каждый поиск или вставка привязывается к корню, поэтому корень всегда будет присутствовать в кэше. В противном случае, при достаточно большом значении  $M$ , типичные случаи поиска или вставки сопряжены максимум с двумя случаями отсутствия в кэше. При достаточно большом объеме кэш-памяти существует высокая вероятность того, что первая страница (дочерняя страница корня), к которой происходит обращение при поиске, уже присутствует в кэше, поэтому средние затраты на один поиск, скорее всего, будут значительно меньше двух зондирований.

При использовании расширяемого хеширования маловероятно, что весь каталог будет храниться в кэше, поэтому можно ожидать, что обращение и к каталогу, и к странице будет сопряжено с отсутствием в кэше (это худший случай). То есть, для выполнения поиска в очень большой таблице требуются два зондирования: одно для обращения к соответствующей части каталога, а другое для обращения к соответствующей странице.

Эти алгоритмы завершают наше знакомство с поиском, поскольку для эффективного их использования требуется понимание базовых свойств бинарного поиска, деревьев бинарного поиска, сбалансированных деревьев, хеширования и trie-деревьев — фундаментальных алгоритмов поиска, которые были изучены в главах 12–15. Все вместе, эти алгоритмы предоставляют нам решения задачи реализации таблицы символов в широком множестве ситуаций: они представляют собой прекрасный пример возможностей технологии алгоритмов.

## Упражнения

- **16.46** Разработайте реализацию таблицы символов с использованием В-деревьев, которая включает в себя деструктор, конструктор копирования и перегруженную операцию присваивания и поддерживает операции *construct*, *count*, *search*, *insert*, *remove* и *join* для АДТ первого класса таблицы символов с обеспечением поддержки дескрипторов клиента (см. упражнения 12.6 и 12.7).
- **16.47** Разработайте реализацию таблицы символов с использованием расширяемого хеширования, которая включает в себя деструктор, конструктор копирования и перегруженную операцию присваивания и поддерживает операции *construct*, *count*, *search*, *insert*, *remove* и *join* для АДТ первого класса таблицы символов с обеспечением поддержки дескрипторов клиента (см. упражнения 12.6 и 12.7).

**16.48** Модифицируйте реализацию В-дерева, приведенную в разделе 16.3 (программы 16.1–16.3), чтобы в ней для ссылок на страницы использовался абстрактный тип данных.

**16.49** Модифицируйте реализацию расширяемого хеширования, приведенную в разделе 16.4 (программы 16.5–16.8), чтобы в ней для ссылок на страницы использовался абстрактный тип данных.

**16.50** Оцените среднее количество зондирований, затрачиваемое на каждый поиск в В-дереве, при выполнении  $S$  случайных поисков в типичной кэш-системе, где  $T$  страниц, к которым недавно выполнялось обращение, хранятся в памяти (и, следовательно, они добавляют 0 к значению счетчика проб). Предположите, что  $S$  значительно больше  $T$ .

- 16.51** Оцените среднее количество зондирований, затрачиваемое на каждый поиск в расширяемой хеш-таблице, для модели кэш-памяти, описанной в упражнении 16.50.
- **16.52** Если используемая система поддерживает виртуальную память, разработайте и проведите эксперименты с целью сравнения производительности В-деревьев с производительностью бинарного поиска при выполнении случайных поисков в очень большой таблице символов.
- 16.53** Реализуйте АТД очереди с приоритетами, который поддерживает операцию *construct* для очень большого количества элементов, за которой следует очень большое количество операций *insert*, *remove* и *maximum* (см. главу 9).
- 16.54** Разработайте АТД внешней таблицы символов, основанной на применении представления В-деревьев в виде списка пропусков (см. упражнение 13.80).
- **16.55** Если используемая система поддерживает виртуальную память, проведите эксперименты с целью определения значения  $M$ , обеспечивающего наименьшее время выполнения для реализации В-дерева, которое поддерживает случайные операции *insert* в очень большой таблице символов. (Прежде чем выполнять подобные эксперименты, которые могут потребовать больших затрат, возможно, стоит изучить основные характеристики используемой системы.)
- **16.56** Модифицируйте реализацию В-дерева, приведенную в разделе 16.3 (программы 16.1–16.3), чтобы она работала в среде, в которой таблица размещается на внешнем устройстве хранения информации. Если система допускает произвольный доступ к файлам, поместите всю таблицу в один (очень большой) файл и в структуре данных вместо указателей используйте смещение внутри файла. Если система допускает непосредственное обращение к страницам на внешних устройствах, в структуре данных вместо указателей используйте адреса страниц. Если система допускает оба вида доступа, выберите подход, который, по вашему мнению, наиболее подходит для реализации очень большой таблицы символов.
- **16.57** Модифицируйте реализацию расширяемого хеширования, приведенную в разделе 16.4 (программы 16.5–16.8), чтобы она работала в среде, в которой таблица размещается на внешнем устройстве хранения информации. Поясните причины выбранного подхода к распределению каталога и страниц по файлам (см. упражнение 16.56).

## Литература, использованная в части 4

Основными источниками для этого раздела являются книги Кнута (Knuth); Баецца-Ятса (Baeza-Yates) и Гонне (Gonnet); Мехлхорна (Mehlhorn); Кормена (Cormen), Лейзерсона (Leiserson) и Ривеста (Rivest). В этих книгах подробно рассматриваются многие из приведенных в этой части алгоритмов, вместе с математическим анализом и предложениями по практическому применению. Классические методы подробно изложены в книге Кнута. Более новые методы описаны в других книгах, в них же приводятся ссылки на другую литературу. В этих четырех источниках, а также в книге Седжвика (Sedgewick)-Флажолета (Flajolet) описан практически весь материал, который упоминается, как "выходящий за рамки этой книги".

Материал, приведенный в главе 13, взят из статьи Роура (Roura) и Мартинеса (Martinez) 1996 г., статьи Слеатора (Sleator) и Тарьяна (Tarjan) 1985 г. и статьи Гюи-

ба (Guibas) и Седжвика 1978 г. Как видно из дат публикации этих статей, исследование сбалансированных деревьев продолжается. Перечисленные труды содержат подробные обоснования свойств RB-деревьев и аналогичных им структур, а также ссылки на более современные работы.

Трактовка trie-деревьев, приведенная в главе 15, является классической (хотя в литературе редко можно встретить реализации на языке C++). Материал по TST-деревьям взят из статьи Бентли (Bentley) и Седжвика, опубликованной в 1997 г.

В статье Байера (Bayer) и Мак-Крейта (McCreight), опубликованной в 1972 г., рассматриваются B-деревья; алгоритм расширяемого хеширования, представленный в главе 16, взят из статьи Фагина (Fagin), Нивергельта (Nievergelt), Пиппенгера (Pippenger) и Стронга (Strong), опубликованной в 1979 г. Аналитические результаты в отношении расширяемого хеширования были получены Флажолетом в 1983 г. С этими статьями обязательно следует ознакомиться всем, кто желает получить более подробную информацию по методам внешнего поиска. Практическое применение этих методов обусловлено контекстом систем баз данных. С введением в эту область можно ознакомиться, например, в книге Дейта (Date).

- R. Baeza-Yates and G. H. Gonnet, *Handbook of Algorithms and Data Structures*, second edition, Addison-Wesley, Reading, MA, 1984.
- J. L. Bentley and R. Sedgwick, "Sorting and searching strings," Eighth Symposium on Discrete Algorithms, New Orleans, January, 1997.
- R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica* 1, 1972.
- T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
- C. J. Date, *An Introduction to Database Systems*, sixth edition, Addison-Wesley, Reading, MA, 1995.
- R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, "Extendible hashing—a fast access method for dynamic files," *ACM Transactions on Database Systems* 4, 1979.
- P. Flajolet, "On the performance analysis of extendible hashing and trie search," *Acta Informatica* 20, 1983.
- L. Guibas and R. Sedgwick, "A dichromatic framework for balanced trees," in *19th Annual Symposium on Foundations of Computer Science*, IEEE, 1978. Also in *A Decade of Progress 1970–1980*, Xerox PARC, Palo Alto, CA.
- D. E. Knuth, *The Art of Computer Programming. Volume 3: Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1997.
- K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.
- S. Roura and C. Martinez, "Randomization of search trees by subtree size," Fourth European Symposium on Algorithms, Barcelona, September, 1996.
- R. Sedgwick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, MA, 1996.
- D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM* 32, 1985.

# Предметный указатель

---

## Символы

:: 155  
<< 132  
== 132

## А

Абстрактные операции 26  
  find (поиск) 26, 31, 36  
  union (объединение) 26, 33, 36  
Абстрактный объект 136  
Абстрактный тип данных (АТД) 127, 164  
  дек (double-ended queue). *См. Очередь:  
  двухсторонняя*  
Автоматическое распределение памяти 181  
Алгоритм 21, 42, 110, 185, 190, 197, 296,  
  299, 316, 331, 402, 440, 603, 645  
  амортизации 525  
  амортизационного анализа 595  
  анализ 23, 42, 47  
  быстрого объединения 29. *См. также Метод:  
  быстрого объединения*  
  быстрого поиска 27, 28. *См. также Метод:  
  быстрого поиска*  
  быстрой сортировки 299. *См. также Метод:  
  быстрой сортировки*  
  вероятностный 316  
  взвешенного быстрого объединения 32  
  вставки 603  
  Горнера 185, 572  
  Евклида 193  
  индексирования текстовых строк 640  
  обменного слияния 333  
  обхода дерева 235  
  объединения-поиска (union-find) 36  
  оперативный 36  
  оптимизации 525  
  пирамидальной сортировки (heapsort) 331  
  поиска 603, 645  
  поиска максимума 200  
  поразрядной сортировки 402. *См. также  
  Поразрядная сортировка*  
  "разделяй и властвуй" 197, 207. *См. также  
  Рекурсия*

## Алгоритм

  рандомизированный 71  
  рандомизации 525  
  распределяющего подсчета 296  
  рекурсивный 190  
  "сборка мусора" 110  
  сжатия пути. *См. Метод: сжатия пути (path  
  compression)*  
  случайного хеширования 590  
  сортировки 440  
    неадаптивный 440  
  хеширования 578  
Амортизация 525  
Анализ 23, 44  
  алгоритмов 42, 47  
  производительности 70  
  эмпирический 44  
Аппроксимация 91  
  нормальная 91  
  функций 60  
Асимптотическое выражение 57  
Ассоциативная память 476

## Б

База данных 646  
Байт 111, 401  
Балансировка BST-дерева 524  
Библиотека стандартных шаблонов C++ 22  
Бинарный поиск 66, 493  
Бит 404  
Битонная (bitonic) последовательность 335  
Бор (forest) 33, 219, 223. *См. также Дерево*  
Быстрая сортировка 299, 407  
  двоичная 407  
  нерекурсивная реализация 310  
  с разделением на три части 322  
Быстрый поиск (quick-find) 28

## В

Вектор 324  
Вершина (vertex) 120, 219  
Виртуальная память 467, 646



Внешний поиск 645

Внешняя сортировка 454

Вставка

в 2-3-4-дерево 541

в B-дерево 658

в BST-дерево 502

в RB-дерево 548, 549

в каталог расширяемого хеширования 673

в расширяемую хеш-таблицу 672

в списке пропусков 556

с расширением 536

со скосом 534

Выборка 326, 327, 330

медианы 327

нерекурсивная 327

Выражение 57

асимптотическое 57

## Г

Гармонические числа 54

Граф 26, 120, 224

насыщенный 123

неориентированный 121

обход графа 240

представление в виде списков смежности 122

разрезанный 123

связный (connected) 225

## Д

Данные 45

ошибочные 45

реальные 45

случайные 45

структура 76, 86

тип 76

Данные-члены 129

Двоичное представление 62

Двоичный логарифм 53, 54

Двойное хеширование 588

Дек 164

Дерево 26, 29, 77, 189, 219, 311, 337, 358,  
386, 477, 603

2-3- 552

2-3-4

нисходящее 540

построение 543

разделение 542

сбалансированное 540

Дерево

B 652, 654

BST 505, 509, 515, 524

двойная ротация 535

рандомизованное 527

сбалансированное 524

M-арное 221

patricia 617

RB 545

trie 608

многопутевое 628

TST 629

бинарного поиска 238, 477, 498, 533

расширенное 533

бинарное 221, 236, 391

биномиальное 391

быстрой сортировки 311

главное 223

корень 30

красно-черное 545

неупорядоченное 224

обход дерева 190, 230

объединение дерева 530

остовное 26

"разделяй и властвуй" 337, 338

с корнем 220, 224

свободное 220

сортирующее 358, 363

индексное 386

суффиксов 640

упорядоченное 220

цифрового поиска (DST) 603

бинарное 604

Дескриптор 519. *См. также Ссылка*

Деструктор 175

Динамические хеш-таблицы 594

Динамическое программирование 216

Динамическое распределение памяти 108

Доступ 649

индексированный последовательный 649

Драйвер

комплексных чисел 172

сортировки массивов 279

**З**

- Заглушка (stub) 134
- Задача 200, 321, 450
  - Бозе-Нельсона 450
  - голландского национального флага 321
  - занятости 580
  - Иосифа 97
  - коллекционера карточек 581
  - о дне рождения 580
  - о ранце 212, 214
  - о ханойских башнях 200
  - связности 23, 26
  - сложность 71
- Запись 142
  - инфиксная 142
  - Польская 142
  - постфиксная 142
- Звезда Коха 207
- Золотая пропорция 210
- Золотое сечение (golden ratio) 55, 572
- Зондирование 583, 647
  - линейное 583

**И**

- Инверсия 265
- Индекс 508, 643
  - текстовой строки 509
- Индексный элемент 170
- Интерфейс 82, 127
  - Argau.h 280
  - непрозрачный 127
- Инфиксная запись 142

**К**

- Каталог 650
- Класс 83, 127, 138
  - Complex 174
  - Item 136
  - POINT 128
  - QUEUE 176
  - string 114, 180
  - Vector 90
  - абстрактный 156
  - контейнерный 138
  - производный 156
  - кластер 585
- Кластеризация 588
- Клиент 127

- Клиентская программа 82
- Ключ 251, 259, 321, 406, 475, 602
  - дублированный 321
  - поиска 475
  - сигнальный 259. *См. также Сортировка: вставками*

- Ключевое слово 130
  - private 130
  - public 130
  - static 130, 131
  - this 130

Коллекция объектов 136

- Компаратор 439, 445
  - слияния 469

Комплексные корни из единицы 174

- Компоненты 26
  - связанные 26

Константа 54

Эйлера 54

Конструирование 564

- Конструктор 95, 129
  - копирования 175, 178
  - списка пропусков 558

Контейнерный класс 138

Корень (root) 30, 220

Корзина 412

**Л**

Линейное зондирование 583

Листья 220

Логарифм 53, 54

- двоичный 53, 54
- натуральный 53

**М**

Марковская цепь 660

- Массив 27, 83, 86, 87
  - двумерный 116

Матрица 116

- разреженная 120
- смежности 121

Медиана 326

Мемуаризация 212

Метод 27, 197, 295, 300, 418, 438, 578

- быстрого объединения *См. Алгоритм: быстрого объединения*
- быстрого поиска 27. *См. также Алгоритм: быстрого поиска*

- Метод
- быстрой сортировки 300. *См. также Алгоритм: быстрой сортировки*
  - выборки 326
  - медианы из трех элементов 316
  - раздельного связывания 578
  - "разделяй и властвуй" 197
  - распределяющего подсчета 295
  - самоорганизующегося поиска 514
  - сжатия пути (path compression) 34
  - сортировки специального назначения 248, 438
  - Флойда 379
  - эвристика в масштабах корзины 418
- Моделирование неупорядоченной очереди 176
- Модульное
- программирование 133
  - хеширование 570
- Мультисписок 120
- Н**
- Натуральный логарифм 53
- Нормальная аппроксимация 91
- О**
- О-нотация 49
- Обход графа 240
- Общедоступный (public) 130
- Объединение 26
- двух BST-деревьев 520
  - дерева 530
- Объект 129, 136
- абстрактный 136
  - коллекция 136
- Объектно-ориентированное программирование (ООП) 130
- Объявление typedef 80
- Оператор
- delete 108
  - new 108
  - new[] 90
  - return 79
- Операции 26
- абстрактные 26
  - find (поиск) 26, 32, 36
  - union (объединение) 26, 33, 36
  - со строками 112
- Операция 132, 148, 326, 439, 479, 646
- << 132
  - == 132, 478
  - count 485
  - insert 167, 393, 500, 581, 646
  - join 520, 560
  - partition 517
  - remove (удаление) 481, 488, 593
  - search (поиск) 481, 490, 581, 529, 646
  - select (выбор) 480, 515
  - sort (сортировка) 480
  - выборки (selection) 326
  - вытолкнуть 148
  - записи (write) 454
  - затолкнуть 148
  - идеального обратного тасования (perfect unshuffle) 440
  - идеального тасования (perfect-shuffle) 439
  - нахождения медианы 326
  - объединить (join) 479
  - поиск (find) 153
  - соединение (union) 153
  - создать 148, 153
  - сравнения 113
  - сравнения обмена (compare-exchange) 440
  - считывания (read) 454
- Оптимизация 525
- Остовное дерево (spanning tree) 26
- Очередь 137, 159, 355
- FIFO 159
  - без повторяющихся элементов 168
  - на базе массива 162
  - на базе связного списка 161
  - биномиальная 358, 389, 392
  - двухсторонняя 164. *См. также Абстрактный тип данных: дек (double-ended queue)*
  - неупорядоченная 163
  - обобщенная 137, 163
  - объединение двух биномиальных очередей 397
  - по приоритетам 355
  - в виде двухсвязного списка 383
  - для индексных элементов 385
  - на базе индексного сортирующего дерева 387
  - на базе сортирующего дерева 368
  - неупорядоченная 382

**П**

- Пакет 333
- Память 476
  - автоматическое распределение памяти 181
  - ассоциативная 476
  - виртуальная 465, 646
  - утечка 177
- Параметры функции 79
- Перегруз 466
- Перегрузка (overloading) 83
- Перегрузка операций (operator overloading) 131
- Пирамидальная сортировка 331
- Поиск
  - 26, 28, 65, 190, 474, 475, 602, 645, 646
  - бинарный 66, 68, 493
  - ближайшего соседа 482
  - быстрый (quick-find) 28
  - в В-дереве 657
  - в глубину 190, 240
  - в диапазоне 482, 495
  - в списках пропусков 557
  - в таблице расширяемого хеширования 668
  - в ширину 243
  - внешний 645
  - интерполяционный 496
  - неуспешный 489
  - от метки 482
  - поразрядный 602
  - последовательный 65, 68, 486
  - с использованием индексации по ключам 483
  - строки 114
  - успешный 489
- Полином 182
- Польская запись 142
- Попытка Бернулли 90
- Поразрядная сортировка 401
  - LSD 403, 425
  - MSD 403, 412
  - обменная 407
- Постоянная *См. Константа*
- Постоянная Эйлера 54
- Постфиксная запись 142
- Потеря быстродействия 114
- Представление 62
  - двоичное 62
- Преобразование типов 78

- Префиксная запись 142
- Приватный (private) 130
- Приведение типов 78
- Программа 82, 145, 189, 205, 240
  - PostScript 145
  - клиентская 82
  - поиск в глубину 240
  - рекурсивная 189
  - рисования линейки 205
- Программирование 211
  - динамическое 216
  - восходящее 211
  - нисходящее 212
  - модульное 133
- Пропорция 210
  - золотая 210
- Простые числа Мерсенне 571
- Процедура
  - fixDown 370
  - fixUp 370
- Путь (path) 219
  - длина 227
  - простой 225
  - сжатие 34

**Р**

- Разделение
  - каталога 669
  - страницы 669
- Раздельное связывание 578
- Разрешение конфликтов 568
- Рандомизация 525, 573
- Распределение памяти 108, 181
  - автоматическое 181
  - динамическое 108
  - под двумерный массив 117
- Расширяемое хеширование 665
- Реализация 127
- Ребра 120
- Ребро (edge) 219
- Рекуррентные соотношения 60
- Рекурсия 60, 189, 340, 353
  - глубина рекурсии 195
  - листовая (оконечная) 196
  - "разделяй и властвуй" 197
- Решето Эратосфена 88
- Ротация 512

**С**

Свойства графов 26  
Связанные компоненты 26  
Связность 23  
Связный список (linked list) 93, 94, 101, 555  
    двухуровневый 555  
Связывание 578  
    раздельное 578  
Связь  
    нулевая 498  
Сертификат 601  
Сеть сортировки 439, 445  
Сжатие пути 34  
    делением пополам 35  
Символ (char) 78  
Слияние 330, 332, 456  
    абстрактное обменное 334  
    без использования служебных меток 335  
    двухпутевое 332  
    многофазное 461  
    сбалансированное многопутевое 456  
    связных списков 350  
Словарь 476. *См. также Таблица символов*  
Слово 404  
Сложность задачи 71  
    верхняя граница 71  
    нижняя граница 71  
Сортировка 103, 248, 251, 299, 330,  
    355, 401, 439, 454  
    адаптивная 253  
    битонной последовательности 335. *См. также*  
        *Слияние*  
    быстрая 299  
        двоичная 407  
        многомерная 424  
        нерекурсивная реализация 310  
        с разделением на три части 322  
    Бэтчера 442  
    внешняя 251, 454  
    внутренняя 251  
    вставками 253, 258  
    выбором 257  
    выбором связного списка 291, 293  
    из сортирующего дерева 372  
    индексная 286  
    массива с помощью управляющей программы 252  
    массива строк 119

**Сортировка**

    методом вставки в список 103  
    методом распределяющего подсчета 296  
    методом Шелла 269  
    неадаптивная 253  
    непрямая 255  
    нисходящая 371  
    обменная 289  
    пирамидальная 331, 355, 373, 375  
    по индексам и указателям 283  
    по указателям 286  
    поблочная 471  
    поразрядная 401. *См. также Поразрядная*  
        *сортировка*  
        обменная 407  
        трехпутевая быстрая 420  
    производительность 263  
    пузырьковая 261  
    с использованием очереди по приоритетам 371  
    с помощью BST-дерева 501  
    связных списков слиянием сверху вниз 351  
    слиянием 330  
        без копирования 341  
        восходящая 342  
        нисходящая 336  
        ориентированная на связные списки 349  
    слиянием Бэтчера 439  
        четно-нечетная 440  
    строк 118  
    устойчивая 255  
    Шейкер 267  
Специальные функции 54  
Спецификация 141  
Список 86  
    двухсвязный 106  
    интерфейс обработки списков 105  
    мультисписок 120  
    обработка 100  
    обращение списка 101  
    обход (traverse) элементов списка 101  
    пропусков 555  
    распределение памяти под списки 108  
    связный 93, 100  
    смежности 122  
    циклический 97  
Ссылка 519

Стек 140, 309  
LIFO 140  
магазинного типа (pushdown stack) 137, 309  
  без повторяющихся элементов 167  
  на базе массива 148  
  на базе связного списка 149

Страница 647

Строка 111, 324, 404  
  операции со строками 112  
  сравнения 113  
  поиск 114

Структура 77, 83, 94  
  данных 21, 76, 86, 363, 551  
  AVL 552  
  индексного сортирующего дерева 385  
  пирамидальная 363  
  составная 116  
  самоссылочная 94  
  составная 77  
  циклическая 94

## Т

Таблица  
  символов 476, 600  
  существования (existence table) 484  
  хеш 594

Тип 77

Тип данных 76, 78, 84, 126, 284  
  double 78  
  float 78  
  int 78  
  item 284  
  long int 78  
  point 84  
  short int 78  
  абстрактный (АТД) 126  
  базовый 78  
  первого класса 171  
  преобразование 78. *См. также Приведение типов*

Типы чисел 80

Турнир 236

## У

Удаление в списках пропусков 560

Узел 94  
  ведущий 102  
  внешний 498  
  внутренний 498

Узел (node) 219  
  -предок (grand parent) 220  
  дочерний (children) 220  
  родительский (parent) 220  
  родственный (sibling) 220  
  терминальный (оконечный) 220

Указатель (pointer) 77, 85  
  строки 113

Универсальное хеширование 574

Упорядочение файла 290  
  обменное 290. *См. также Сортировка: Обменная*

Уровень абстракции 126

Утечка памяти (memory leak) 177, 180

## Ф

Файл 290  
  обменное упорядочение 290 *См. также Сортировка: Обменная*

Факториал 54

Формула Стирлинга 55

Фрактал 207  
  Коха 208

Функция 54, 60, 77  
  fixDown 373  
  fixUp 373  
  key() 478  
  main 252  
  POINT() 128, 129  
  qsort 118, 287  
  search 481, 503, 510  
  showR 501  
  sort 258, 480, 489  
  strcmp 324  
  -член 129  
    статическая 131  
  аппроксимация 60  
  вычисления факториала 191  
  дружественная 131  
  Иосифа 98  
  объявление 79  
  округления сверху 54  
  округления снизу 54  
  определение 79  
  параметры 79  
  рекурсивная 190, 198  
  специальная 54  
  хеш 567  
    модульная 571  
  целочисленная 62

**Х**

- Хеш-таблица 668
  - динамическая 594
  - расширяемая 668
- Хеш-функция 569
  - для символьных строк 574
  - для строковых ключей 573
  - мультипликативная 569
- Хеширование 474, 567, 665
  - двойное 588
  - методом линейного зондирования 584
  - модульное 570
  - расширяемое 665
  - с открытой адресацией 583
  - с помощью отдельного связывания 578
  - случайное 590
  - универсальное 574
  - упорядоченное 600

**Ц**

- Целые числа (int) 78
- Цель 660
  - марковская 660
- Цикл 225

**Ч**

- Числа 54, 78
  - гармонические 54
  - Фибоначчи 54, 211
  - Мерсенне 571
  - мнимые 173
  - с плавающей точкой (float) 78
  - целые 78

**Э**

- Эвристика в масштабах корзины 418
- Элемент 170, 475
  - индексный 170
- Эмпирический анализ 44

**Я**

- Язык 145
  - PostScript 145

**Иностранные термины**

- BST-дерево 524
- FIFO 233
- LIFO 140, 233
- PostScript 145
- Standard Template Library 22

