国外经典计算机科学教材

# Algorithms in C++ Parts 1-4
## Fundamentals, Data Structures, Sorting, Searching

# 算法 I–IV（C++ 实现）——
## 基础、数据结构、排序和搜索
## （第三版）

[美] Robert Sedgewick 著
张铭泽 赵剑云 梁勇等 译

Addison Wesley

中国电力出版社
www.infopower.com.cn

# 算法I–IV（C++实现）——基础、数据结构、排序和搜索（第三版）

Sedgewick 在解释概念方面颇具天分，能使它们更易理解。而把真正的代码用重而大小的文字块在文中，使读者容易理解它们，则是本书的另一个优势。插图、程序和表等相关本丰富了读者的学习体验，这些因素使本书显得与众不同。

——William A. Ward, 南阿拉巴马大学

Robert Sedgewick完全重写了他的著作，对它进行了充分的扩展和更新，涵盖了目前重要的算法和数据结构。Christopher Van Wyk 和 Sedgewick 开发的新实现采用的是C++语言，这种实现不仅能简洁直接地表达算法，而且给编程者提供了实践的方法，以便在真正的应用中测试这些算法。

新的版本提供了很多新算法，而且对每个算法的解释也比以前的版本详细得多。新的版面设计以及详细、富有创意并且具有注释的插图，使本书的表达能力大大地提高了。第三版保留了将理论和实践成功混合在一起的特点，正是这一点，使 Sedgewick 的著作成为 25 万多名程序员无价的参考资源。

本书是全卷的前半部分，涵盖了基本的数据结构、排序算法、搜索算法以及它们的相关应用。虽然本书实质上可以用于各种语言的程序设计，Christopher Van Wyk 和 Sedgewick 的实现都采用了 C++ 类和 ADT 实现的自然对应。

本书的精彩内容包括：

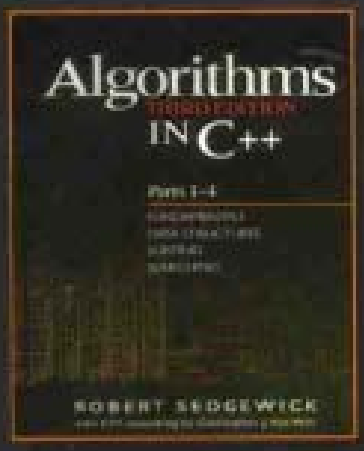
- 扩展了对数组、链表、字符串树及其他基本数据结构的介绍。
- 比以前的版本更加着重于抽象数据类型（ADT）、模块化程序设计方法、面向对象的程序设计方法和 C++ 类。
- 有关排序、选择、优先级队列 ADT 实现和符号表 ADT（搜索）实现的算法，超过 100 个。
- 关于二项式队列、多路基数排序、随机化 BST、伸展树、跳跃表、多叉线索、B 树、可扩充散列等，采用了新的实现。
- 关于算法的量化分析，是比较算法的依据。
- 1000 多条新的练习，帮助读者学习算法。

无论是你初学算法，还是想找一本将最新C++经典算法和新算法融入程序设计的参考手册，你都会发现本书提供了丰富的有用信息。

责任编辑 / 姚贵胜
封面设计 / 王红柳

ISBN 7-5083-1808-0
9 787508 318080 >

ISBN 7-5083-1808-0
定价：55.00 元




**Robert Sedgewick** 是普林斯顿大学计算机系的 William O. Baker 教授，也是 Adobe Systems 公司的主管，曾经在 Xerox PARC、IDA 和 INRIA 公司担任研究员。

**Christopher J. Van Wyk** 是 Drew 大学数学和计算机系的教授，曾在 Bell 实验室担任研究员，现在是那里的顾问。

Sedgewick和Van Wyk都在斯坦福大学获得了博士学位，导师同是 Donald E. Knuth。

国外经典计算机科学教材

# Algorithms in C++ Parts 1-4
## Fundamentals, Data Structures, Sorting, Searching

# 算法 I–IV（C++ 实现）——
## 基础、数据结构、排序和搜索
## （第三版）

[美] Robert Sedgewick 著
张铭泽 赵剑云 梁勇等 译

中国电力出版社
www.infopower.com.cn

Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, 3rd Edition (ISBN 0-201-35088-2)
Robert Sedgewick
Authorized translation from the English language edition, entitled Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, published by Addison Wesley, Copyright © 1998.
All rights reserved.
No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.
CHINESE SIMPLIFIED language edition published by China Electric Power Press Copyright © 2003.

本书由美国培生集团授权出版。
北京市版权局著作权合同登记号 图字：01-2000-3090 号

图书在版编目（CIP）数据
算法 I～IV（C++实现）——基础、数据结构、排序和搜索（第三版）/（美）塞奇威克著；张铭泽等译.
—北京：中国电力出版社，2004
（国外经典计算机科学教材）
ISBN 7-5083-1808-0
Ⅰ.算… Ⅱ.①塞…②张… Ⅲ.①电子计算机－算法设计－教材②C 语言－程序设计－教材 Ⅳ.TP301.6
中国版本图书馆 CIP 数据核字（2003）第 110219 号

丛 书 名：国外经典计算机科学教材
书　　名：算法I-IV（C++实现）——基础、数据结构、排序和搜索（第三版）
编　　著：（美）Robert Sedgewick
翻　　译：张铭泽　赵剑云　梁勇 等
责任编辑：姚贵胜
出版发行：中国电力出版社
　　地址：北京市三里河路6号　　邮政编码：100044
　　电话：（010）88515918　　传　　真：（010）88518169
印　　刷：北京丰源印刷厂
开　　本：787×1092　1/16　　印　　张：34.125　　字　　数：755 千字
书　　号：ISBN 7-5083-1808-0
版　　次：2004 年 2 月北京第 1 版　　2004 年 2 月第 1 次印刷
定　　价：55.00 元
版权所有　翻印必究

# 出 版 说 明

新世纪的朝阳刚刚露出丝抹微红，如火如荼的全球信息化浪潮便汹涌而至，让人无时无刻不感受到新一轮产业革命的气息。如何在这场变革中占尽先机，既是对民族信息业的挑战，也是机遇。从而，作为民族信息产业发展基石的高等教育事业就被赋予了比以往更重的责任，对培养和造就我国21世纪的一代新人提出了更高的要求。但在计算机科学突飞猛进的同时，专业教材的发展却严重滞后，越来越成为人才培养的瓶颈。同时，以美国为代表的西方国家计算机科学教育经历了充分的发展，产生了一批有着巨大影响力的经典教材，因此，以批判、借鉴的态度有选择地引进这些国外经典计算机教材，将促进国内教学体系和国外接轨，大大推动我国计算机教育事业的发展。

中国电力出版社进入计算机图书市场已有近6个年头，通过坚持“高端、精品、经典”战略，致力于与国外著名出版机构合作，出版了大批博得计算机业界和教育界赞誉的作品。通过与信息技术教育界人士的广泛沟通，同时依托丰富的出版资源，中国电力出版社适时推出了“国外经典计算机科学教材”的出版计划。本次教材出版计划是和美国最大的计算机教育出版机构——Pearson教育集团（Addison-Wesley、Prentice-Hall等皆为其下属子公司）合作，依托其数十年积累的大批经典教材资源，确保了教材选题的权威经典。

为保证这套教材的含金量，并做到有的放矢，我们在国内组织了由中国科学院、北京大学等一流院校教师组成的专家指导委员会，对高校课程教学体系做了系统、详细的调查，听取了众多教育专家、行业专家的意见，对教育部的教育规划进行了认真研究，并深入了解国外大学实际教学选用的教材状况，对国外教材做了理性的分析，确立了依托国家教育计划、传播先进教学理念、为培养符合社会需要的高素质创新型人才服务，来作为本次“国外经典计算机科学教材”出版计划的宗旨。

我们从2002年的下半年开始着手这套教材的策划工作，并多次组织了专家研讨会、座谈会等，分析现有教材的优点与不足，采其精华，并力争体现本套教材的质量和特色。

1. 深入理解国内的教学体系结构，并比照国外相同专业的课程设置，既具有现实的适用性，又立足发展眼光，具备一定的前瞻性。

2. 以计算机专业的核心课程为基础，同时配合专业教学计划，争取覆盖专业选修课程和专业任选课程。

3. 选取国外的最新教材版本，同时对照国内同专业课程的学时要求，对不适用的版本进行剔除，充分满足国内教学要求。

4. 根据专业对口和必须具备同课程教学经验的要求，严格挑选译者，并严把质量关，确保教材翻译的高质量。

5. 通过从原出版社网站下载勘误表及与原书作者进行沟通的方式，对原书中的错误一一做了修改。

6. 对教材出版的后期工作，如审校、编辑、排版、印刷进行了严格的质量把关。

经过专家指导委员会的集体讨论，并广泛听取广大高等院校师生的意见，反复比较，从数百种国外教材中遴选出数十种，列入第一阶段的出版计划。这些教材的作者无一不是学富五车的大师，如 Stallings，Date，Ullman，Aho，Bryant，Sedgewick 等，他们的作品均是一版再版，并被众多国外一流大学如 Stanford University，MIT，UC Bekerley，Carnegie Mellon Univeristy，University of Michigan 等采用为教材。拟订的第一阶段出版计划包括 30 种图书，内容覆盖程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等计算机专业核心基础课程，基本满足国内计算机专业的教学要求。

此外，为了帮助广大任课教师加深对本系列教材的理解，减轻他们的备课难度，我们从国外出版机构引进了大批的课程教学辅助资料，并积极延请国内优秀教师，根据其使用该系列教材中的教学经验，着手编写更加适合国内应用状况的教辅材料。

由于我们对国内高校计算机教育存在认识深度上的不足，在选题、翻译、编辑加工出版等方面的工作中还有许多有待提高之处，恳请广大师生和读者提出批评和建议,并期待有更多的人加入到我们的工作中来。我们的联系方式是：

电子邮件：csbook@cepp.com.cn

联系电话：010-88515918-300

联系地址：北京市西城区三里河路 6 号中国电力出版社

邮政编码：100044

# 前　言

写作本书的意图是纵览当今所使用的最重要的计算机算法，并为越来越多需要了解这方面知识的读者讲解基础的技术。本书可以在学生掌握了基本的程序设计技能，熟悉了计算机系统，但是还没有学习过计算机科学或计算机应用的高级领域的专门课程时，作为计算机科学的第二、第三或第四门课程的教科书。此外，由于本书包含了大量有用算法的实现和关于其性能特征的详细信息，所以它还适用于自学，或作为那些从事计算机系统或应用程序开发的人员的参考手册。宽广的视角使本书成为了计算机算法领域最合适的入门读物。

对于这新的一版，我不仅完全重写了它的内容，而且还添加了一千多个练习、一百多幅图和许多程序，此外还给所有的图和程序都添加了详细的注释。新的素材涵盖了两个方面的内容——新的主题和对许多经典算法的更完整的解释。抽象数据类型是这本书的新重点，这个新重点使程序的应用更广泛，并且与现代程序设计环境的关系更紧密。读过本书旧版本的人一定会发现，新版本包含了更为丰富的新信息。所有读者都能感到大量的教学性资料为掌握基本概念提供了有效的途径。

由于新的素材数量过多，所以我们把新版本分成了两卷（每一卷的容量大约都相当于旧的版本），本书就是第一卷。这一卷包括基本概念、数据结构、排序算法和搜索算法；第二卷涵盖的高级算法和应用程序是以第一卷介绍的基本抽象概念和方法为基础的。在这一版中，几乎所有关于基本原理和数据结构的素材都是新的。

这本书不仅仅是为程序设计人员和计算机科学系的学生而编写的，凡是想使计算机运行得更快，或想用它解决更多问题的人都可以使用这本书。本书中的算法代表了过去 50 多年所研究的知识的主体，对于形形色色的应用程序来说，它们已经成为有效使用计算机所不可缺少的。从物理学中的 N 体仿真到分子生物学中的基因序列问题，这些基础算法已成为科学研究的要素；从数据库系统到 Internet 搜索引擎，它们已经成为现代软件系统的实质。随着计算机应用领域的不断扩大，本书介绍的大量基础算法的效用也在逐渐增长。本书的目标就是成为学生和那些想了解基础算法并且能够灵活地运用这些算法作为基本工具来处理任意计算机应用程序的专业人士的资源。

## 范围

本书共有十六章，分为四大部分：基本原理、数据结构、排序算法和搜索算法。这里的说明是想让读者对尽可能多的基础算法的基本特性有一个了解。本书介绍的算法已经经过多年的广泛应用，代表了程序员和计算机科学系学生掌握的知识的主体。从二项式队列到帕氏线索算法这个范围内的有创见的方法都与计算机科学核心的基础范例相关。第二卷

由另外四部分构成，它们分别为字符串处理算法、几何算法、图形算法和高级主题。我写这本书的主要意图是将各个领域应用的基础方法集合起来，从而提供用计算机解决基本问题的最好方法。

如果你已经学习过一两门计算机科学的初级课程（如 C++、Java 或 C 等高级语言的程序设计课程，可能还有讲解程序设计系统的基础概念的课程）或者具有同等的程序设计经验，那么一定会非常欣赏本书提供的资料。因此，本书是为每个熟悉现代程序设计语言和现代计算机系统的基本特性的人而编写的。书中所建议的参考文献有助于你弥补背景知识的不足。

由于用来支持分析结果的大部分数学知识都来源于本书之内（或者被标记为不在本书范围之内），所以尽管具有完备的数学知识无疑会有很大帮助，但是专门的数学准备却不是必需的。

## 课程中的用法

如何教授本书的内容具有极大的灵活性，这是由教师的经验和学生的知识储备决定的。这里有充足的基础内容用来教授初学者数据结构，也有充足的高级主题来教授高年级学生算法设计与分析。有些教师可能想着重于实现和实践方面的内容，有些教师则可能想着重于分析和理论概念方面的内容。

我为这本书的使用设计了多种课程学习资料，其中包括讲稿用的幻灯片、程序设计作业、家庭作业以及示例测验和交互式练习。访问本书的主页 http://www.awl.com/cseng/titles/0-201-35088-2 可以得到这些资料。

关于数据结构和算法的基础课程可能会着重于第二部分中的基本数据结构和它们在第三、四部分中的实现中的使用。关于算法设计和分析的课程可能会着重于第一部分和第五章中的基础内容，然后研究第三、四部分中的算法如何达到良好的渐近性能的方法。关于软件工程的课程可能会省略数学和高级算法部分的内容，而着重于如何把这里给出的算法实现集成到大的应用程序或系统中。关于算法的课程则可能采用通盘考虑的方法并介绍所有领域内的概念。

本书的早期版本近年来被全世界各大学院和大学用作计算机科学的第二或第三门课程的教材或其他课程的补充性阅读材料。在普林斯顿大学，我们的经验是这本书内容广泛，为专业学生提供了计算机科学的引论，此后有关算法分析、系统程序设计和理论性计算机科学的课程都可以对它进行扩展，此外它还给其他学科的学生提供了一整套方法，他们可以立刻将这些方法很好地利用起来。

本书中的练习（大部分是这一版新添的）分为几种类型。有些专用于测试对文中内容的理解程度，只要求读者完成一个练习或应用文中介绍的概念。有些则涉及到算法的实现和结合，或者根据学习经验比较各种算法来学习它们的特性。还有一些则是重要内容的深入探讨，这些内容不适于放在正文中。仔细阅读和思考这些练习会使每个读者受益匪浅。

## 实际应用的算法

任何想更加有效地使用计算机的人都可以把这本书作为参考手册或自学材料。具有程序设计经验的人可以在这本书中找到许多专业主题的内容。对于更大范围内的读者，则可以阅读个别独立的章节，尽管有时某一章中的算法会用到前一章中介绍的方法。

本书的定位是研究有可能实际用到的算法。它提供的专业工具信息很中肯，读者可以放心地实现、调试和试验算法，来解决问题或在应用程序中提供功能性。本书中讨论的算法都有完整的实现，而且在一系列连贯的示例中还有关于操作这些应用程序的描述。

由于我们使用的是真正的代码，而不是伪代码，所以你可以便捷地将它们应用到实际当中。访问本书的主页可以得到各个程序的清单。你可以以多种方式利用这些有指导意义的程序来研究算法。阅读这些代码可以检测你对算法细节的理解程度，或者弄明白初始化、边界条件处理和其他常常给程序设计带来挑战的问题的解决方法。还可以试着运行一下代码来查看运行中的算法，根据经验研究它们的性能，并将你得到的结果与书中的表进行对照，或者尝试着对它们进行一些修改。

在适当的时候，书中会列出经验数据和分析结果以说明为什么选择某些算法。当要引起注意时，它会描述实际应用的算法和纯理论的结果之间的关系。尽管没有强调算法分析和理论计算机科学之间的联系，但是上下文中仍然对它们进行了说明。有关算法和实现的性能特征被综合在一起，进行了概括，在本书中随处可见。

## 程序设计语言

本书中所有实现所用的程序设计语言都是 C++。其中的程序使用了大量标准的 C++语法，而且文中还对每个结构做了简要说明。

我和 Chris Van Wyk 基于类、模板和重载操作符开发了一种 C++程序设计形式，我们认为这种形式是把算法和数据结构当作真正的程序来表示的一种有效方式。我们努力使实现的代码精致、紧凑、高效，并具有可移植性。由于这种形式无论何时都是一致的，所以用它开发的相似程序看起来就比较相像。

本书中的许多算法本身就具有相似性，而与实现它们的语言无关，例如快速排序算法就是进行快速排序（一个典型的例子），无论它是用 Ada、Algol-60、Basic、C、C++、Fortran、Java、Mesa、MoDula-3、Pascal、PostScript、Smalltalk 以及其他无法尽数的程序设计语言实现的，还是在证明了它是高效排序方法的环境中实现的。一方面，我们的代码是根据用这些语言或大量其他的语言（本书还有一个用 C 语言实现的版本，此外用 Java 实现的版本很快就会面世）实现算法的经验形成的；另一方面，某些语言的某些特性是根据它们的设计者使用本书介绍的某些算法和数据结构的经验形成的。

第一章中包含一个复杂的例子，说明如何用这种方法开发算法的高效 C++实现。第二章描述了对这些算法进行分析的方法。第三和第四章说明并验证了我们用来实现数据类型

和 ADT 实现的基本机制。这四章为本书的其余章节打下了基础。

## 致谢

有许多人对这本书的早期版本提出了有用的反馈信息。尤其是普林斯顿大学和布朗大学的学生们，在过去的几年中，他们一直为本书的草稿所苦。特别感谢 Trina Avery 和 Tom Freeman，他们在编写第一版的过程中给予了我极大的帮助。感谢 Janet Incerpi，她以自己的创造力和才智说服了我们使用早期原始的用计算机处理的数字排版硬件和软件来制作本书的第一版。感谢 Marc Brown，是他负责的算法可视化研究创建了本书中的诸多插图。感谢 Dave Hanson 和 Andrew Appel，他们回答了我许多关于程序设计语言的问题。我还要感谢众多读者对各个版本所做的评论，他们是：Guy Almes、Jon Bentley、Marc Brown、Jay Gischer、Allan Heydon、Kennedy Lemke、Udi Manber、Dana Richards、John Reif、M. Rosenfeld、Stephen Seidman、Michael Quinn 和 William Ward。

我有幸和 Peter Gordon、Debbie Lafferty 和 Helen Goldstein 一起，在 Addison-Wesley 共同完成了这个新的版本，在整个项目的进展过程中我得到了他们耐心的指导。还有 Addison-Wesley 的其他职员，能和他们一起工作也是我的荣幸。这个项目的特性使得本书给他们中的许多人带来了非同寻常的挑战，对于他们的耐心我感谢万分。

在编写本书的过程中，我结交了三位良师益友，在这里，我特别想表达对他们的感谢。首先是 Steve Summit，他在技术层次上仔细检查了手稿的早期版本，提出了大量详尽的文字意见，尤其是关于程序方面的。Steve 很明白我的目标是提供精致、高效的实现代码，他的意见不仅有助于我衡量实现代码的一致性，还有助于我充分地改进它们。第二位是 Lyn Dupre，他也对我的手稿提出了大量详尽的意见，这对于纠正和避免语法错误，更重要的是找到一种连贯一致的编写形式来说非常宝贵，就是这种形式将本书众多的技术资料很好地组织在了一起。第三位是 Chris Van Wyk，他用 C++语言实现并调试了我所有的算法，还回答了我大量有关 C++的问题，帮助我开发了一种合适的 C++程序设计形式，并且将我的手稿仔细阅读了两遍。此外，当我拆开了他的许多 C++程序，而后随着我从他那里学习的 C++知识逐渐增多，又将这些程序还原成他编写的样子时，Chris 依然极力支持着我。对于能有向 Steve、Lyn 与 Chris 学习的机会，我感谢万分，他们的投入对于这本书的进展至关重要。

我在本书中所写的内容大部分都是从 Don Knuth 的教学和著述中学到的，他是我在斯坦福大学的指导老师。尽管 Don 对这本书没有什么直接的影响，但是其中仍然能够感受到他的存在，因为是他把算法研究放到了科学基础之上，使这样的著述成为了可能。Philippe Flajolet 是我的朋友兼同事，他是将算法分析发展成一个成熟研究领域的主力，对这本书具有相同的影响力。

我深深地感谢普林斯顿大学、布朗大学和 INRIA（Institut National de Recherche en Informatique et Automatique）给予的支持，在那里，我完成了本书的大部分工作。还要感谢 Defense Analyses 学会和 Xerox Palo Alto 研究中心，我在访问它们的时候完成了本书的某些工作。本书

的许多部分都依赖于 National Science Foundation（国家科学基金会）和 Office of Naval Research 的慷慨支持。最后，我要感谢 Bill Bowen、Aaron Lemonick 和 Neil Rudenstine，他们帮助我在普林斯顿创建了一个学术环境，使得我能够在其中编写这本书而不受其他众多任务的打扰。

Robert Sedgewick

Marly-le-Roi，法国，1983

普林斯顿，新泽西州，1990，1992

詹姆斯敦，罗得马州，1997

普林斯顿，新泽西州，1998

## C++顾问的前言

是算法让我接近了计算机科学。算法研究要求以几种方式进行思维：用创造性的方式来发现解决问题的想法；用逻辑方式来分析它的正确性；用数学方式来分析它的性能；用挥洒汗水的方式将想法表达成具体的步骤序列，使它能够成为软件。由于我在算法研究方面最满意的地方就是将它们实现成为有指导意义的计算机程序，所以我欣然接受了与 Bob Sedgewick 共同编写一本基于 C++程序来介绍算法的书。

Bob 和我以简明的方式用 C++的适当特性编写了许多示例程序。我们使用了类将抽象数据类型的说明和它的实现细节分离开。此外我们还使用了模板和重载操作符，以便我们的程序不做改变就能应用于多种不同的数据类型。

但是我们放弃了展示其他 C++技术的机会。例如，我们通常会省略一些构造函数，它们是使每个类成为“一级类”所必需的（参阅第四章，可以学到如何实现这一点）：大多数构造函数都使用赋值运算代替初始化将值存储在数据成员中；我们使用 C 形式的字符串，而不依赖于 C++库中的字符串类型；大多数“轻量级”的包装类我们都不使用，而只使用简单的指针。

我们之所以做出这样的选择，是为了将重点放在算法本身上，而不让 C++特有的某些技术细节分散读者的注意力。一旦读者明白了本书中的程序是如何运行的，那么就有很好的基础去弄明白所有的 C++技术，去欣赏它们所涉及的效率权衡，去崇拜标准模板库设计者的天才了。

无论读者是第一次遇到算法的研究，还是重逢旧相识，都会从这本书中得到乐趣，至少像我和 Bob Sedgewick 共同开发程序时所得到的乐趣一样多。

感谢 Jon Bentley、Brian Kernighan 和 Tom Szymanski，我从他们那里学到了很多程序设计的知识。还要感谢 Debbie Lafferty，是他建议我考虑这个项目的。最后要感谢贝尔实验室、德鲁大学和普林斯顿大学，它们给予了我制度上的支持。

Christopher Van Wyk

查塔姆，新泽西州，1998

## 有关练习的注释

给练习分类是件充满风险的事情，因为本书的读者具有的知识和经验水平参差不齐。虽

然如此，指导仍然是适宜的，所以许多练习都附加了一个记号，以帮助你判断如何使用它们。

测试你对内容的理解程度的练习是用空心三角标识的，如下所示：

▷ 9.56 给出一个空二项式队列在插入EASYQUESTION后的结果。

通常这样的练习都与文中的例子直接相关。它们不会有特别的难度，但是做这些练习可以教会你一个事实或一个概念，它们可能是在你阅读正文时感到困惑不解的。

给正文添加新的和引人思考的资料的练习是用空心圆标识的，如下所示：

○ 14.19 编写一个程序，对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，采用链地址法将 $N$ 个随机整数插入表长为 $N/100$ 的散列表中，然后给出最长和最短的地址链的长度。

这样的练习将鼓励你思考一个与正文内容相关的重要概念，或者回答你阅读正文时可能遇到的一个问题。你会发现做这种练习是非常有价值的，即使你没有时间把它们都做一遍。

具有挑战性的练习是用黑色圆点标识的，如下所示：

● 8.45 假设归并排序将文件按随机方式进行拆分，而不是恰好进行平分。使用这样的方法对包含 $N$ 个元素的文件进行排序，平均需要使用多少次比较操作？

这种练习可能需要花费大量的时间才能完成，这是由你的经验决定的。一般说来，最有效的办法就是分几个不同的时期来攻克它们。

少数难度极大（与其他大多数练习比较而言）的练习是用两个黑色圆点标识的，如下所示：

●● 15.29 请证明 $N$ 个随机 0、1 序列关键字构造的线索的高度为 $2\lg N$。提示：考虑生日问题（见性质 14.2）。

这种练习近似于研究文献上陈述的问题，不过本书中的资料已经为你尝试解决它们（可能会成功）做好了准备。

对于考察你的程序设计能力和数学能力的练习，则没有明确记号。这些要求程序设计或数学分析的练习是一种自我检查。我们鼓励所有的读者都通过实现算法来检测自己对它们的理解程度。对于程序员或程序设计课程的学生来说，这样的练习很简单，不过对于最近很少编程的人来说，可能会有一定难度：

1.22 修改程序 1.4，用随机产生的 0～$N-1$ 之间的整数对来代替标准输入，而且循环只执行 $N-1$ 次合并操作。试分别在 $N=10^3$，$10^4$，$10^5$ 和 $10^6$ 时运行你的程序，输出对应于每个 $N$ 值所产生的边的个数。

出于同样的心情，我们鼓励所有的读者探索有关算法特性知识的分析基础。对于科学工作者或离散数学课程的学生来说，这样的练习很简单，不过对于近来很少做数学分析的人来说，可能会有一定难度：

1.12 在用“带权的快速合并”算法在最坏情况下建立的 $2^n$ 个节点的树中，计算从一个节点到根节点的平均距离。

还有更多的练习需要你阅读和吸收。我希望这里有足够的练习能够激励你去积极地加深对自己感兴趣的主题的理解程度，而不仅仅满足于阅读正文所得到的收获。

# 目　录

出版说明
前　言

第一部分　基本原理

第一章　简介 ..... 3
1.1　算法 ..... 3
1.2　示例：连通问题 ..... 5
1.3　合并-查找算法 ..... 8
1.4　前景展望 ..... 16
1.5　总结 ..... 17

第二章　算法分析原理 ..... 19
2.1　实现和经验分析 ..... 20
2.2　算法分析 ..... 22
2.3　函数的增长 ..... 24
2.4　大O符号 ..... 30
2.5　递归基础知识 ..... 33
2.6　算法分析举例 ..... 36
2.7　保证、预测和限制 ..... 40

第一部分参考资料 ..... 44

第二部分　数据结构

第三章　基本数据结构 ..... 47
3.1　构建组件 ..... 48
3.2　数组 ..... 56
3.3　链表 ..... 62
3.4　基本的链表处理 ..... 68
3.5　链表的内存分配 ..... 75
3.6　字符串 ..... 78
3.7　复合数据结构 ..... 82

第四章　抽象数据类型 ..... 91
4.1　抽象对象和对象集合 ..... 98
4.2　下推栈 ADT ..... 100
4.3　栈 ADT 客户示例 ..... 103
4.4　栈 ADT 的实现 ..... 108

4.5 创建一个新 ADT ..... 112
4.6 FIFO 队列和广义队列 ..... 117
4.7 复制和索引项 ..... 124
4.8 一级 ADT ..... 128
4.9 基于应用的 ADT 范例 ..... 137
4.10 前景展望 ..... 142

**第五章 递归与树** ..... 144
5.1 递归算法 ..... 145
5.2 分治法 ..... 150
5.3 动态编程 ..... 161
5.4 树 ..... 168
5.5 二叉树的数学性质 ..... 174
5.6 树遍历 ..... 177
5.7 递归二叉树算法 ..... 181
5.8 图遍历 ..... 186
5.9 前景展望 ..... 191

**第二部分参考资料** ..... 192

## 第三部分 排序算法

**第六章 基本排序方法** ..... 195
6.1 游戏规则 ..... 196
6.2 选择排序 ..... 200
6.3 插入排序 ..... 201
6.4 冒泡排序 ..... 203
6.5 基本排序方法的执行特性 ..... 205
6.6 Shell 排序法 ..... 209
6.7 对其他类型的文件进行排序 ..... 217
6.8 索引和指针排序 ..... 221
6.9 链表排序 ..... 227
6.10 关键字索引统计 ..... 230

**第七章 快速排序** ..... 233
7.1 基本算法 ..... 234
7.2 快速排序算法的性能特性 ..... 237
7.3 栈大小 ..... 240
7.4 小的子文件 ..... 243
7.5 利用三个元素的中间元素来划分 ..... 245
7.6 重复值 ..... 249
7.7 字符串和向量 ..... 251

7.8 选择 ..... 253

**第八章 归并及归并排序** ..... 256

8.1 二路归并 ..... 257
8.2 抽象的合适归并算法 ..... 258
8.3 自顶向下的归并排序 ..... 260
8.4 对基本排序方法进行改过 ..... 263
8.5 自底向上的归并排序 ..... 264
8.6 执行典型的归并排序算法 ..... 268
8.7 使用链表执行归并排序 ..... 270
8.8 再次讨论递归过程 ..... 272

**第九章 优先队列与堆排序** ..... 274

9.1 基本的实现方法 ..... 276
9.2 堆的数据结构 ..... 279
9.3 基于堆的算法 ..... 281
9.4 堆排序 ..... 285
9.5 优先队列抽象数据类型 ..... 293
9.6 索引元素的优先队列 ..... 297
9.7 二项式队列 ..... 300

**第十章 基数排序** ..... 309

10.1 比特、字节、字 ..... 310
10.2 二进制快速排序 ..... 313
10.3 MSD 基数排序 ..... 316
10.4 三路基数快速排序 ..... 322
10.5 LSD 基数排序 ..... 326
10.6 基数排序的特性 ..... 328
10.7 运行时间低于线性的排序 ..... 332

**第十一章 特殊用途的排序方法** ..... 335

11.1 巴彻尔奇偶归并排序 ..... 336
11.2 排序网络 ..... 340
11.3 外部排序 ..... 347
11.4 “排序-归并”的实现 ..... 351
11.5 并行“排序-归并” ..... 356

**第三部分参考资料** ..... 360

## 第四部分 搜索算法

**第十二章 符号表和二叉搜索树** ..... 365

12.1 符号表抽象数据类型（ADT）.....366
12.2 关键字索引检索.....372
12.3 顺序搜索.....374
12.4 二叉搜索.....380
12.5 二叉搜索树.....384
12.6 BST 的性能特性.....390
12.7 符号表的索引实现.....393
12.8 在 BST 的根进行的插入.....397
12.9 其他 ADT 函数的 BST 实现.....400

**第十三章 平衡树**.....408
13.1 随机化 BST.....411
13.2 发散 BST.....416
13.3 自上而下 2-3-4 树.....421
13.4 红黑树.....426
13.5 跳跃表.....434
13.6 性能特性.....441

**第十四章 散列**.....444
14.1 散列函数.....445
14.2 链地址法.....451
14.3 线性探测.....455
14.4 双重散列.....459
14.5 动态散列表.....463
14.6 综述.....466

**第十五章 基数检索**.....470
15.1 数字搜索树.....471
15.2 线索（trie）.....473
15.3 帕氏线索.....480
15.4 多叉线索和 TST.....487
15.5 文本字符索引算法.....501

**第十六章 外部排序**.....504
16.1 游戏规则.....505
16.2 索引顺序存取.....506
16.3 B 树.....508
16.4 可扩充散列.....519
16.5 综述.....528

**第四部分参考资料**.....531

# 第一部分

# 基 本 原 理

# 第一章

# 简　介

本书的目的是研究各种重要而有用的算法。所谓算法（algorithm）就是解决那些适合计算机实现的问题的方法。我们将会涉及到许多不同应用领域的问题，但是只集中研究其中重要而有趣的部分。我们还会在每个算法上花费足够的时间来理解它的重要特性并考虑一些细节问题。我们的目标就是学习大量目前用于计算机的重要算法，以便能更好地应用这些算法。

本书中理解算法的方法就是实现并测试它们，试验它们的各种变体在一些小例子中讨论算法的操作，并在我们在现实中可能遇到的类似大例子中试验这些算法。我们将用 C++ 编程语言来描述算法，这样同时也提供了有用的实现。我们的程序有统一的形式，可以方便地转换成其他流行的编程语言。

我们还关注算法的性能特性，以帮助我们开发改进的版本、比较同一个任务的不同算法，并预测或保证大问题的性能。理解算法如何执行需要实验或数学分析。对于许多特别重要的算法我们会考虑它们的细节，适当时直接改进分析结果，或在必要时使用研究文献中的结果。

为了说明我们改进算法的主要方法，我们将在这一章中详细地讨论一个例子，这个例子由一些解决特定问题的算法组成。这个问题不是一个玩具（toy）问题，它是一个基本的计算任务，我们采用的解决办法在多种应用程序中也是有效的。我们首先提出一个简单的解决方案，然后理解这个方案的性能特性，以帮助我们发现如何改进算法。在重复几次这样的过程之后，我们会得到一个解决该问题的高效而有用的算法。这个原型例子为我们贯穿本书所使用的方法学奠定了基础。

我们以对本书内容的简短讨论作为这一章的总结，总结包括本书主要部分的简要描述和每部分之间的关系。

## 1.1　算法

在编写一个计算机程序时，我们通常会实现一个事先设计好的解决该问题的方法。这

个方法与具体使用的计算机无关，它可以适用于许多计算机和计算机语言。我们必须学习的是解决问题的方法，而不是计算机程序本身。“算法”是计算机科学的术语，它描述了一个适用于计算机程序实现的解决问题的方法。算法是许多领域研究的核心问题。

大多数算法所关心的是计算机中数据的组织方法。用这种方式建立的对象称为“数据结构”（data structure）。当然数据结构也是计算机科学研究的核心对象。这样，算法与数据结构就结合在了一起。本书认为数据结构是算法的副产物或最终产物，为了理解算法，我们就必须研究数据结构。简单的算法可以使用复杂的数据结构。反之，复杂的算法也可以使用简单的数据结构。我们将在本书中研究许多数据结构的属性。事实上，这本书称为“C++中的算法与数据结构”可能更合适。

我们使用计算机来帮助解决问题时，通常会有多个不同的方案。对于小问题，使用哪种方案无关紧要，只要方案是正确的即可。然而对于大问题（或者由大量需要解决的小问题组成的应用程序），我们希望设计出尽可能节省时间和空间的方案。

我们学习算法设计的主要原因是：它的原则使得节省大量时间和空间成为可能，甚至使得原本不可能完成的任务得以完成。在处理几百万个对象的大应用程序中，使用设计良好的算法能够使程序快几百万倍，我们会在 1.2 节中和其他部分看到这样的例子。与之相比较，如果投资额外的钱和时间来购买并安装一台新计算机以缩短程序运行时间，那么得到的加速比可能仅是 10 或 100。在任何应用领域中，精细的算法设计都是解决大问题的过程中最有效的部分。

如果开发的程序很庞大或很复杂，就需要理解和定义待解决的问题，控制它的复杂度，并将它分解成更容易实现的子任务。分解后的许多算法常常易于实现，然而，多数情况下，有些算法的选择很难确定，因为运行这类算法会占用大量的系统资源。我们这本书所关注的就是这类算法。我们会学习大量的对解决各个应用领域问题有用的基本算法。

计算机系统中程序的共享越来越广泛，所以在这本书中，尽管我们可能希望使用大量的算法，但是我们也希望只实现其中的一小部分。例如，C++ STL（Standard Template Library，标准模板库）中包含了大量基本算法的实现。然而实现一些基本算法有助于我们更好地理解算法，并更有效地使用它们，从而从库中开发更高级的版本。更重要的是，基础算法经常需要重新实现，主要原因是我们会遇到具有新特征的全新计算机环境（硬件和软件），旧的实现未必最佳。换句话说，我们经常实现适合问题的基本算法，而不单单使用系统例程，从而使得我们的方案更具移植性、更持久。另一个原因是，尽管 C++内部有优化处理，但在特定问题上，共享软件的机制在提供适合的、能有效执行的库程序方面还不够方便。

计算机程序常常被过分优化。保证一个特定算法实现的高效性可能并不重要，除非这个算法在大量的任务中出现，或被多次使用，否则一个精细的相对简单的实现就足够了，即保证可以运行，而且运行时间比最佳可能算法慢 5 到 10 倍，也就是说它可以多运行几秒钟。相反，首要算法的选择恰当可以使加速比达到 100 或 1000，甚至更多，节省的运行时间可能会是几分钟、几小时或更多。本书关心那些最佳算法的最简单、最合理的实现。

为一个特定任务选择最佳算法是一个复杂的过程，其中可能需要成熟的数学分析方法。计算机科学中研究这类问题的分支是算法分析（analysis of algorithms）。我们研究的许多算法都是经过分析后证明效率很高。另外那些算法经过实验证明可以很好地工作。我们的主要目标是学习重要任务的合理算法，还要仔细地比较不同算法的效率。我们使用一个算法就不能不考虑它所使用的资源——我们也需要清楚希望这些算法如何执行。

## 1.2 示例：连通问题

假设有一个整数对的序列，每个整数代表某个类型的一个对象，p-q 对表示“p 连接到 q”，即 p 和 q 之间连通。假设连通关系具有传递性——如果 p 与 q 连通，q 与 r 连通，那么 p 与 r 连通。我们的目标是写一个过滤出那些不在集合中的对的程序：输入 p-q 对时，如果已输入的对的集合中没有隐含了这样的对（通过连通关系的传递性），那么程序应输出该对。如果前面输入的对隐含了 p 与 q 连通，那么程序将忽略 p-q，准备输入下一对。图 1.1 示例了这样的过程。

| | | |
|---|---|---|
| 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 | |

**图 1.1 连通性示例**

给定一个表示对象之间连通的整数对的序列（左），连通算法的任务是输出那些提供新的连通关系的对（中）。例如，对 2-9 不是输出的一部分，因为 2-3-4-9 这样的连通关系隐含在前面（这一证明列在右侧）。

我们的问题是设计一个可以记录足够多的关于对的信息的算法，以便确定一对新的对象是否连通，我们称设计这样一个算法的任务为“连通问题”（connectivity problem）。这个问题在大量的重要应用程序中出现。我们简要地介绍三个例子，以反映该问题的本质。

例如，整数可以代表大的网络中的计算机，对表示网络中的连接，这样，我们的程序就用来决定 p 和 q 通信时是需要建立新的连接，还是使用已有的连接来建立通信线路。在这类应用问题中，可能会有成百万个点和上亿个连接，甚至更多。正如我们即将看到的，如果没有一个有效的算法，那么解决这种应用问题几乎是不可能的。

类似地，整数还可以表示电路网络中的连接点，对就表示连接这些点的电线。在这种情况下，如果可能，我们会用程序来找出一个将所有点连接起来而没有额外连接的方案。事实上，列表中的边并不能保证所有的点一定可连接。我们马上会看到，判断是否连通是程序的一个主要问题。

图 1.2 给出了这两类应用问题的大型示例，这个图的检验说明了连通问题的难度。我们如何能快速地证明网络中任何两点之间都连通？

还有一个例子就是在特定的程序环境下，这样的对可以表示两个变量是否等价。有了一系列这样的定义后，就可以判断两个给定名字是否等价。就是这个应用激发了我们即将看到的几个算法的产生。它将我们的问题与一种简单抽象直接相关，并提供了一种使算法能在更广泛的应用程序中使用的方法，这一点我们即将看到。

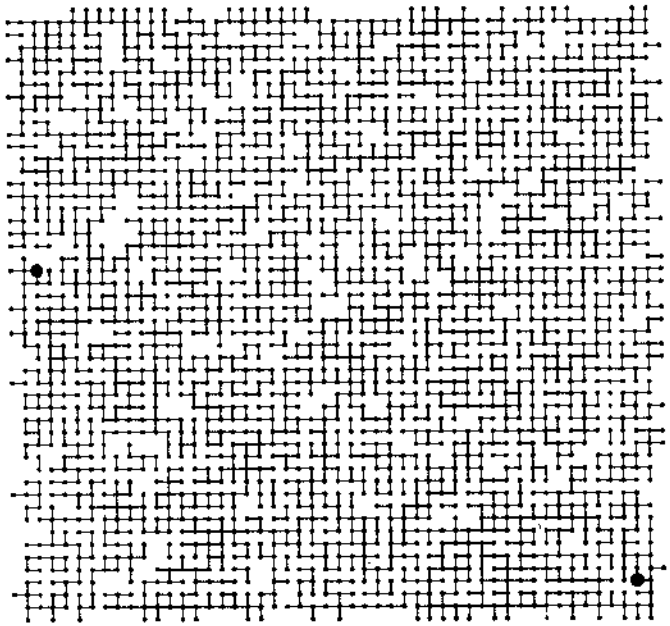


**图 1.2 大型连通示例**

连通问题中的对象由连接点表示，对表示二者之间的连接。这个例子可以被认为是城市中不同建筑之间的电线，或是计算机芯片上的电线。图形表示使得人们可以发现不连通点，但算法必须在给定整数对的基础上工作。用大的黑点表示的两个节点是否连通？

上面描述的像变量名等价问题这样的应用程序需要将每个不同变量名与整数关联起来。这种关联关系隐含在前面描述的网络连接和电路应用中。第十章~第十六章中的大量算法都直接地给出了这样的关联关系。这样，为了不失一般性，我们在这一章中假设有 $N$ 个对象，每个对象都有一个整数名与之对应（0~$N$ -1）。

我们在寻找一个完成定义良好的指定任务的程序。可能有许多相关问题也是我们要解决的。我们在开发一个算法时所面对的第一个问题就是，要确保以合理方式指定问题。我们对一个算法要求得越多，它完成任务所需要的时间和空间也就越多。对这个关系量化很难，我们常常会修改问题说明，可能会发现问题变得很难解决或解决的代价很高，或者在好的情况下，发现算法可以比原始说明提供更多有用的信息。

例如，我们的连通问题说明只需要程序知道给定的 p-q 对是否连通，而不是连接这两个点的路径。在这个说明上再增加一些要求就会使问题变得很难，而且和另外一些算法相关，我们将在第五章简略地介绍，在第七部分详细地介绍。

前面这段提到的说明可能比从前的说明要求我们有更多的信息，也可能要求更少的信息。例如：我们可能只需要能够回答这样的问题："$M$ 个连接是否能够连接所有的 $N$ 个对象？"这个问题说明，要设计一个高效的算法，就需要对研究的抽象对象进行高层次的推

理。在这个例子中，从图论中可以得出这样一个基本结论：当且仅当连通算法输出的对的个数恰好为 $N-1$ 时，所有 $N$ 个对象是连通的（参见 5.4 节）。换句话说，连通算法不可能输出多于 $N-1$ 个对，因为一旦它输出 $N-1$ 个对，那么从此时起输入的任何对都是连通的。因此，我们可以修改解决连通问题的程序，增加一个计数器就可以得到一个只回答是否问题的程序，而不输出那些不连通的输入对。当计数器达到 $N-1$ 时回答“是”，如果还没达到，则回答“否”。这个问题仅是很多个需要回答相关连通性的问题之一。输入对的集合称为“图”（graph），输出对的集合称为图的生成树（spanning tree），生成树连接了所有的对象。第七部分将介绍图、生成树和有关算法的一些性质。

确定算法执行过程的基本操作很重要，它可以使得为连通问题设计的任何算法对于各种类似问题仍有效。确切地说，每当输入一个新的对时，首先应该判断它是否意味着一个新的连接，然后将这个连接的信息合并到已有的连通关系中，以便将来检查连通关系时使用。我们将这两个任务封装成抽象操作（abstract operation），用整数输入值表示抽象集合的元素，再设计算法和数据结构，然后就可以：

- 查找（find）包含指定元素的集合；
- 用两个元素的“合并结果”（union）代替它们。

在抽象操作的基础上组织算法在解决连通问题方面似乎没有太大用处，但在解决其他问题时却很有帮助。在计算机科学的各个领域中，尤其是算法设计中，较高层次的抽象是一个重要的过程。贯穿本书，我们将时常看到这种情况。在这一章中，我们用不很正式的抽象思维方式指导设计程序，以解决连通问题；在第四章，我们将会看到如何在 C++代码中封装抽象。

在查找和合并两个抽象操作基础上解决连通问题很容易。读入一个新的对 p-q 后，首先对“对”中每个元素执行查找操作，如果对的两个元素均在同一个集合内，那么准备处理下一对，否则，执行合并操作，输出该对。这些集合表示“连通部分”：每一个集合都是对象集合的子集，但集合中任意两个元素之间均连通。这种解决方案使得设计连通性问题算法的任务降低为定义一个表示这样集合的数据结构，并设计一个使用该数据结构的高效的合并和查找算法。

有很多表示和处理抽象集合的方式，第四章将会详细介绍。这一章的主要任务是找到一种能有效地支持合并和查找操作的表示。

## 练习

1.1 如果输入是 0-2、1-4、2-5、3-6、0 4、6-0 和 1-3，那么连通性算法的输出将是什么？

1.2 列出图 1.1 的示例中所有连接两个不同对象的方法。

1.3 试描述这样一个简单的方法，它要统计正文中使用合并和查找操作解决连通性问题后所剩的集合数。

## 1.3 合并-查找算法

为某个问题设计一个好的算法的第一步就是实现一个解决该问题的简单算法。如果待解决的问题是特定的，而且容易解决，那么这个简单的实现就足够了。如果需要一个更复杂的算法，这个简单的实现可以正确地检查那些小的情况，并且为评估性能特性提供一个基准。我们始终关心算法的效率，但在设计第一个程序时，更应该保证它的正确性。

首先要考虑的是如何保存这些输入对，然后再写一个遍历函数来判断下一个对象对是否连通。我们应该使用不同的方案。首先，在实际的应用程序中，对的个数可能很多，以至于不能将它们全部放在内存中。其次，更重要的是，即使我们能够保存这些对，也没有一个简单的方法能判断两个对象是否连通。在第五章我们会设计一个解决这一问题的基本方法。这一章所采用的方法要简单得多，因为它解决的问题难度较低，而且不需要保存所有的对。它使用了一个整数数组，一个整数对应于一个对象，这样就保存了实现合并和查找的必要信息。

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

**图 1.3 快速查找（慢速合并）算法的示例**

序列的内容反映了采用快速查找算法（程序 1.1），当输入为左侧的 p-q 对时，数组的当前内容。阴影部分是经过合并操作后发生改变的项，输入 p-q 对时，将值为 id[p]的项的值改为 id[q]。

在 3.2 节中我们会详细讨论数组这一基本数据结构。这里只使用一个最简单的数组形式：假设我们要定义一个大小为 1000 的数组 a[1000]，数组中第 $i$ 个元素就是 a[i]，其中 $0 \le i < 1000$。

**程序 1.1 连通问题的快速查找解法**

该程序读入一组介于 0 和 N-1 之间（包含 0 和 N-1）的非负整数对，p-q 对表示 p 和 q 连通，输出那些目前尚未连通的输入对。它使用了数组 id，每个对象在数组中都有一项，当且仅当 p 与 q 连通时 id[p]和 id[q]相等。为简化问题，我们定义 N 为编译时常量。当然也可以通过输入确定 N，再动态地分配 id 数组（参见 3.2 节）。

```
#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;
      }
}
```

程序 1.1 是快速查找算法的实现。算法的基础是一个整数数组。当且仅当 $p$ 与 $q$ 连通时，数组中第 $p$ 个元素与第 $q$ 个元素值相等。我们将数组中的第 $i$ 个元素初始化为 $i$，其中 $0 \le i < N$。为了实现 $p$ 和 $q$ 的合并操作，我们遍历数组，将所有名字为 $p$ 的项改为 $q$，当然也可以将所有名字为 $q$ 的项改为 $p$。

图 1.3 显示了以图 1.1 为例，数组经过合并操作后的改变。对于查找操作，我们只需要测试数组项是否相等，因此称之为“快速查找”，相反，合并操作对于每一个输入对都要遍历整个数组，因此称之为“慢速合并”。

**性质 1.1** 对于 $N$ 个对象的连通问题，如果要执行 $M$ 次合并操作，那么快速查找算法将执行至少 $M*N$ 条指令。

对于每一个合并操作，我们都使 for 循环迭代了 $N$ 次，每迭次叠代至少需要一条指令（如果只检测循环是否结束）。■

现代计算机每秒钟可以执行上亿条指令，因此在 $M$ 和 $N$ 都较小时这个代价微乎其微。但如果在某个应用程序中可能有几十亿个对象和几百万个输入对，我们就不能用快速查找算法来解决这样的问题。第二章将简明地对这个结论进行量化。

图 1.4 是图 1.3 的图形化表示。可以认为一些对象是它们所在集合的代表，集合中其他元素均指向这个代表元素。数组的图形表示使整个过程更加清晰，可以看出，在这种表示方法下，对象之间的连接并不一定是输入对之间的连接，它只为算法判断当前输入对是否连通提供信息。

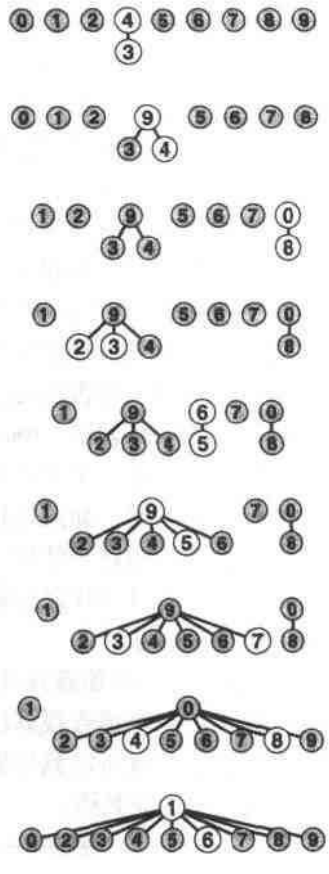


**图 1.4 快速查找算法的树状表示**

该图是图 1.3 中示例的图形化表示，图中的连接并不一定代表输入对中的连接。例如，最后一棵树中有 1-7 这样的连接，而实际上存在这样一系列连接：7-3-4-9-5-6-1，才使得 7 和 1 连通。

下面要介绍的是快速查找算法的互补算法——快速合并算法。它使用的数据结构也是数组，数组的索引是对象的名字，但数组元素是一个复杂的抽象结构——在一个没有环的结构中，每个对象指向同集合内的另一个对象。要判断两个对象是否属于同一个集合，只需跟随每个对象的指针，直至到达一个指向自己的对象。当且仅当经过以上过程得到同一个对象时，两个对象在同一个

集合中。如果二者不在同一个集合内，那么最终得到的一定是不同的对象（每个对象都指向自己）。合并操作只需要将其中的一个链接到另一个即可，因此称之为“快速合并”算法。

图 1.5 与图 1.4 一样是图形化表示，它是快速合并算法在图 1.1 上的应用。图 1.6 给出了 id 数组的相应变化。数据结构的图形化表示使算法的操作易于理解。如果输入对是连通的，那么在数据结构上二者也具有连通性。前面提到过，数据结构中连接与输入对所隐含的连接并不一定完全一致，前者只是用来帮助实现高效的合并算法和查找算法的结构。

图 1.5 中连通的部分称为“树”（tree），树是一个基本组合结构，在本书中我们将多次看到它。我们将在第五章中详细讨论树的性质。图 1.5 中的树对于合并和查找操作非常有用，因为它们十分容易建立，而且具有这样的性质：当且仅当两个对象在输入中连通时树中两个对象连通。沿着树向上走，可以容易地发现包含每个对象的树的树根，于是我们就有了查找两个对象是否连通的方法。每棵树都有一个指向自己的对象，这就是树根（root）。图中没有显示指向自己的指针。从树中的任意一个节点开始到它所指向的对象，再到该对象所指向的对象，如此这样，最终总会到达根节点。我们可以用归纳法证明这一性质：数组初始化后每个对象均指向自身，如果在合并操作之前满足这一性质，那么“合并”操作后仍满足这一性质。

图 1.4 所示的快速查找算法的图也具有上述的性质，二者的区别在于，在快速查找算法的树中，从任何一个节点出发只需要经过一步就可以到达根节点，而在快速合并算法中可能要经过几步才能到达。

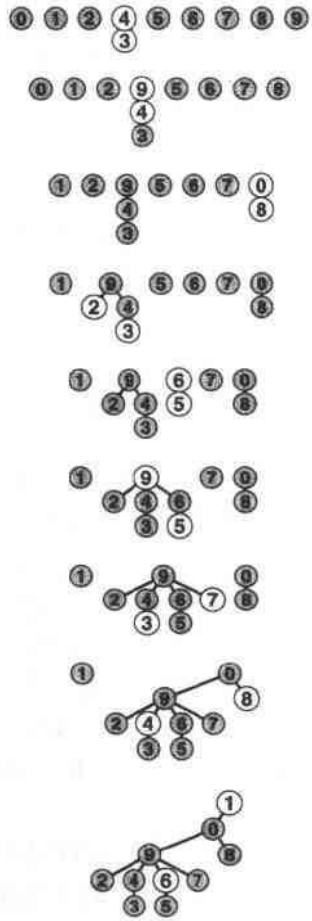


**图 1.5 快速合并算法的树状表示**

该图是图 1.3 示例的图形化表示。我们画一条从对象 i 到对象 id[i]的线段。

**程序 1.2 连通问题的快速合并解法**

如果用这段代码替换程序 1.1 中的 while 循环体，我们就得到一个与程序 1.1 有相同需求的程序，它的合并操作时间更少，但查找操作的时间较长。for 循环和后面的 if 语句指出了 p 和 q 连通在 id 数组中的充分必要条件。赋值语句 id[i]=j 实现了合并操作。

```
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;
```

程序 1.2 是由“合并”操作和查找操作组成快速合并算法的实现。快速合并算法似乎比快速查找算法更快，因为对于每一个输入对，它不需要遍历 id 数组，但究竟快了多少呢？在此回答这个问题比在快速查找算法那部分回答更难，因为执行时间是依赖于输入特性的。通过实践检验或数学分析（见第二章），可以证明程序 1.2 比程序 1.1 效率高许多，因此在大多数实际问题中程序 1.2 更合适。这一节的最后，我们会讨论一个实例。现在我们暂且认为快速合并算法是改进的，因为它消除了快速查找算法的主要约束——要对 $N$ 个对象进行 $M$ 次合并，程序需要至少执行 $M*N$ 条指令。

可见快速合并算法相对于快速查找算法有所改进，但快速合并算法仍有它的局限性——我们并不能保证在每种情况下它都比“快速查找”算法快，因为输入数据可能使查找操作变慢。

**性质 1.2** *若连通问题中，$N$ 表示对象个数，$M$ 表示对的个数，$M>N$，则快速合并算法至少要执行 $M*N/2$ 条指令。*

假设输入对以 1-2，2-3，3-4 这样的顺序输入，输入这样 $N-1$ 个对后，就可以得到一个包含了所有对象的集合，快速合并算法得到的树是一条直线，其中 $N$ 指向 $N-1$，$N-1$ 指向 $N-2$，$N-2$ 指向 $N-3$，依次类推。要对对象 $N$ 执行查找操作，程序就必须遍历 $N-1$ 个指针，这样，前 $N$ 对象要遍历的平均指针数为：

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

现在假设所剩余的对都和 $N$ 连通，这些对的查找，操作至少要遍历（$N-1$）个指针，那么该输入对的序列的 $M$ 次查找，操作数必然大于 $M*N/2$。■

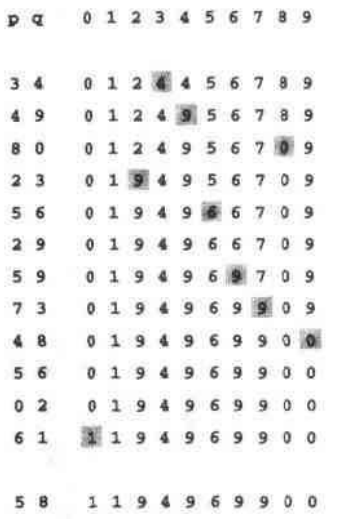



**图 1.6 快速合并算法（查找不是很快）示例**

id 序列的内容反映了采用快速合并算法（程序 1.2），当输入为左侧的 p-q 对时，id 数组的当前内容。阴影部分是经过合并操作后发生变化的项。如果输入为 p-q 对，我们就从 p 开始，直至找到一个满足 id[i]==i 的项 i，然后从 q 开始直至找到一个满足 id[j]==j 的项 j；如果 i≠j，令 id[i]=id[j]。例如，在 5-8 的查找操作过程中，i 依次取 5、6、9、0、1，j 依次取 8、0、1。

幸运的是，我们对算法做一个简单的修改就可以保证这样的最坏情况不出现。具体做法就是在进行合并操作时，不是简单地将第二棵树连接到第一棵树，而是对每棵树保持一个节点计数，并且总是将小的树连接到大的树上。这一修改需要增加部分代码和一个保存节点计数的数组，如程序 1.3 所示，但它提高了效率。我们称这个算法为“带权的快速合并算法”。

图 1.7 显示了以图 1.1 中的输入为例，用带权的合并-查找算法构造的树。即使是这样一个小例子，树中路径也要比图 1.5 中不带权的合并-查找算法的短。图 1.8 说明了最坏情况，此时进行合并操作的两个集合大小总是一样的（2 的幂）。这些树结构看似复杂，但它们有一个简单的性质——对于有 $2^n$ 个节点的树，要到达根节点所遍历的最大节点数

为 $n$。进一步说，如果合并两个有 $2^n$ 个节点的树，就会得到一棵有 $2^{n+1}$ 个节点的树，并且到根节点的最大距离增加为 $n+1$。这一结论为证明带权算法比不带权算法更有效提供了证据。

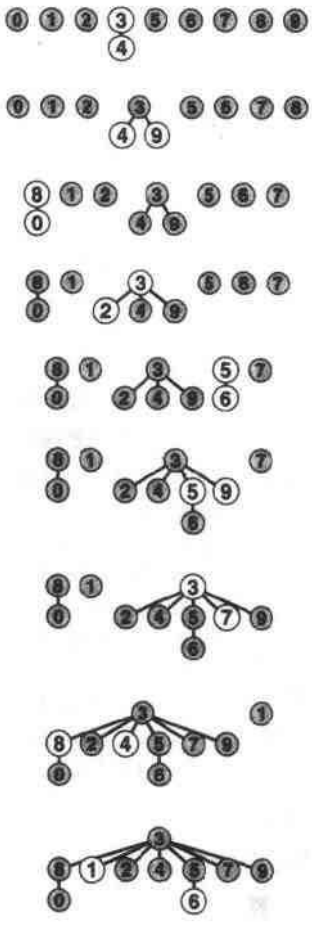


**图 1.7 带权的快速合并算法的树状表示**

这个序列描述了将快速合并算法修改为把较小树的根节点连接到较大树的根节点上。每个节点到它所在的树的根节点的距离变小，所以查找操作更有效率。

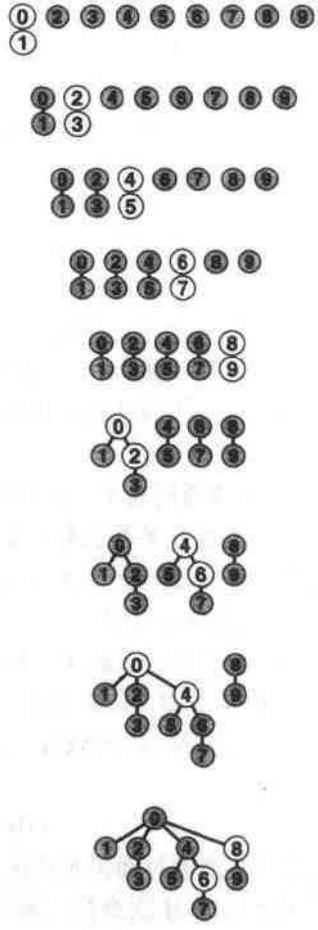


**图 1.8 带权的快速合并算法（最坏情况）**

带权的快速合并算法的最坏情况就是进行合并操作的两棵树一样高。如果对象的个数少于 $2^n$，则从树中任意一个节点到根的距离小于 $n$。

**程序 1.3 快速合并的带权版本**

这个程序是在快速合并算法基础上修改得到的（参见程序 1.2），它增加了一个数组 `sz`，用来保存满足 `id[i]==i` 的对象所在树的节点个数，以便合并操作可以将较小的树连接到较大的树上，这样可以防止树中最长路径的增长。

```
#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;
      }
  }
```

**性质 1.3** *在确定 $N$ 个对象中的两个对象是否连通时，带权的快速合并算法至少顺着指针走 $\lg N$ 步。*

我们可以证明，合并操作保证这一条性质：在由 $k$ 个对象组成的树中，任意节点到根的距离不大于 $\lg k$。当合并一个 $i$ 个节点的集合和一个 $j$ 个节点的集合（$i \le j$）时，我们使得 $i$ 个节点的集合中的指针个数增加了 1，但集合中的元素个数却是 $i+j$，这条性质得到了保证，因为 $1+\lg i=\lg(i+i) \le \lg(i+j)$。■

性质 3 的实验表明，在 $N$ 个对象 $M$ 个连接的情况下，带权的快速合并算法最多执行 $M*\lg N$ 条指令（见练习 1.9），而快速查找算法的大多数情况（快速合并算法的少数情况）至少要执行 $M*N/2$ 条指令，二者形成鲜明对比。可见，使用带权的快速合并算法，可以保证在合理时间内解决大多数实际问题（见练习 1.11）。仅仅在第一个程序的基础上增加几行代码，我们就可以得到一个快几百万倍的算法，而且它可以解决实际应用中遇到的问题。

从图中明显可以看出仅有少数节点和根节点距离很远。实际上，大量问题的实验研究表明：程序 1.3 的带权快速合并算法可以在线性时间内解决实际问题，即运行算法所需要的时间是读输入所需要的时间的常数倍。我们几乎不期望再找到更高效的算法。

下面我们来看是否可以找到能保证线性性能的算法。这

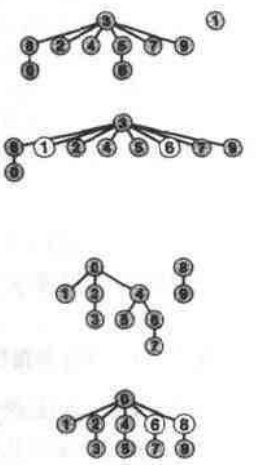


**图 1.9 路径压缩**

我们还可以使树中的路径更短，具体做法就是在进行合并操作时，将所经过的节点均指向根节点，如例子中所示，上面的例子是图 1.7 对应的结果，对于短的路径来说，路径压缩没有什么作用，但如果输入对是 1-6，我们就会将 1、5 和 6 均指向 3，并得到一棵比图 1.7 更扁的树，最下面的例子是与图 1.8 对应的结果。树中长于一个或两个连接的路径可以扩展，但在跨越不同子树时，就会使树变得扁平。如果输入对是 6-8，那么 4、6、8 就都指向 0，树就会变得更扁平。

**图 1.10 等分的路径压缩**

通过向上隔一个节点连接的方法，即底层节点与它上面的一个节点均指向同一节点，我们可以近似地等分路径的长度，如例子中所示，对于所跨越的每个路径都执行这样的操作其结果与全路径压缩一样。

确实是一个困扰了研究人员许多年的问题（见 2.7 节）。进一步改进带权的快速合并算法的简单方法很多。理想情况下，我们希望每个节点都直接指向树的根节点，但又不希望像快速合并算法那样进行大量的修改指针的操作。我们可以采用使被检测的节点指向根节点的方法逼近理想情况。这一步看似不切实际，但是易于实现，而且除了树结构外都不必修改：如果修改树结构可以提高算法效率，就有必要这样做。我们可以实现这一方法，称之为“路径压缩”（path compression），在进行合并操作时，只要将所经过的节点的 id 数组项改为指向根节点即可。这一过程的目的是将树变得扁平，逼近快速查找算法的理想情况，如图 1.9 所示。证明这一算法的分析是非常复杂的，但这一方法简单有效。图 1.11 给出了大的例子的路径压缩的结果。

有许多方法可以实现路径压缩，例如程序 1.4，它通过在连接时向上跳一步指向下一节点的方法压缩路径。如图 1.10 所示，这一方法显然比全路径压缩更容易实现（见练习 1.16），而且可以得到相同的结论。我们将修改的算法称为“等分路径压缩的带权快速合并（weighted quick-union with path compression by halving）算法”。这类算法中哪个更有效呢？节省的时间与实现路径压缩所需要的时间相比是否值得？是否有其他可考虑的技术？要回答这些问题，我们必须更仔细地研究算法和实现，我们将在第二章详细介绍这类关于算法分析的内容。

**程序 1.4 等分的路径压缩**

如果用这段代码替换程序 1.3 中的 for 循环，我们就等分了横跨的路径。这一改变使得经过一个长序列的合并操作后的树几乎是完全平的。

```
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]];
```

后面算法的最终结果几乎是我们在实践中所能期望的最好的。我们得到了易于实现，又保证运行时间是采集数据时间的常数倍的算法。而且，算法对每条边只考虑一次，因为算法使用的空间是与对象个数成正比的，所以它对所处理的边个数没有限制。表 1.1 中的实践数据验证了我们的结论，说明程序 1.3 和它的路径压缩改进算法对于大多数实际问题是有用的。在这些算法中选择哪个还需要更详细的理论分析（见第二章）。

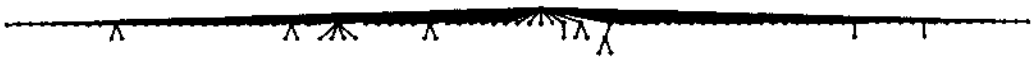


图 1.11 路径压缩的大的示例

这个序列描述了对于从 100 个对象中随机产生的对，用路径压缩的带权快速合并算法处理的结果。除了两个节点之外，树中所有节点到根的距离都是 1 或 2。

## 练习

▷ 1.4 如果输入序列为 0-2、1-4、2-5、3-6、0-4、6-0 和 1-3，写出使用快速查找算法（程序 1.1）解决连通问题时每一步合并操作得到的 id 数组的内容，以及对于每一个输入对，id 数组被访问的次数。

▷ 1.5 做练习 1.4，但使用快速合并算法（程序 1.2）。

表 1.1 合并-查找算法的研究结果

这些相对时间表明，在解决随机的连通性问题方面，在各种各样的合并-查找算法中，带权的快速合并算法是最有效的。路径压缩所带来的好处不是很重要。在这些实验里，$M$ 表示使得 $N$ 个对象连通而随机产生的连接的个数。这一过程所进行的查找操作比合并操作多，所以快速合并算法比快速查找算法要慢。在 $N$ 很大时，快速查找算法和快速合并算法都不适用。带权方法的运行时间显然是与 $N$ 成比例的，即在 $N$ 加倍时运行时间也加倍。

| $N$ | $M$ | 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 |

其中：

F 快速查找算法（程序 1.1）
U 快速合并算法（程序 1.2）
W 带权的快速合并算法（程序 1.3）
P 路径压缩的带权快速合并算法（练习 1.16）
H 等分的带权快速合并算法（程序 1.4）

▷ 1.6 用带权的快速合并算法运行图 1.7 和图 1.8 中的示例，试给出经过每一步合并操作后 id 数组的内容。

▷ 1.7 做练习 1.4，但使用带权的快速合并算法（程序 1.3）。

▷ 1.8 做练习 1.4，但使用等分路径压缩的带权快速合并算法（程序 1.4）。

1.9 使用程序 1.3 处理 $N$ 个对象 $M$ 个连接的连通问题，求需要执行的机器指令数的上界。例如，你可以假设任何 C++赋值语句总是要求执行少于 $c$ 条的指令，$c$ 是一个固定的常数。

1.10 如果连通问题中有 $10^9$ 个对象和 $10^6$ 个输入对，请估计在一台每秒钟执行 $10^9$ 条指令的计算机上运行快速查找算法（程序 1.1）可能需要的最少时间（以天计数）。假设内层 for 循环中每一次迭代至少要执行 10 条指令。

1.11 如果连通问题中有 $10^9$ 个对象和 $10^6$ 个输入对，请估计在一台每秒钟执行 $10^9$ 条指令的计算机上运行带权的快速合并算法（程序 1.3）可能需要的最多时间（以秒计数）。假设外层 while 循环中每一次迭代至多执行 100 条指令。

1.12 在用“带权的快速合并”算法在最坏情况下建立的 $2^n$ 个节点的树中，计算从一个节点到根节点的平均距离。

▷ 1.13 画类似图 1.10 的图，以 8 个节点而不是 9 个节点开始。

○ 1.14 给出使带权的快速合并算法（程序 1.3）产生一条长度为 4 的路径的输入对序列。

● 1.15 给出使等分路径压缩的带权快速合并算法（程序 1.4）产生一条长度为 4 的路径的输入对序列。

1.16 全路径压缩是指我们在进行每一次合并算法时将所经过的节点均指向新树的根节点，试将程序 1.3 修改为全路径压缩。

▷ 1.17 做练习 1.4，但使用全路径压缩的带权快速合并算法（见练习 1.16）。

●●1.18 给出使全路径压缩的带权快速合并算法（程序 1.4）产生一条长度为 4 的路径的输入对序列。

○ 1.19 给出一个例子说明修改快速合并算法（程序 1.2）来实现全路径压缩（见练习 1.16）不足以保证树中没有长的路径。

● 1.20 修改程序 1.3，不用权，而用树的高度（树中任意一个节点到根的最长距离）来决定是否令 id[i]=j 或 id[j]=i。实际运行，试比较它与程序 1.3 的区别。

●●1.21 证明性质 1.3 对练习 1.20 中的算法仍适用。

● 1.22 修改程序 1.4，用随机产生的 0～$N$-1 之间的整数对来代替标准输入，而且循环只执行 $N$-1 次合并操作。试分别在 $N=10^3$，$10^4$，$10^5$ 和 $10^6$ 时运行你的程序，输出对应于每个 $N$ 值所产生的边的个数。

● 1.23 修改练习 1.22 的程序以算出连接 $N$ 个对象需要的边的个数，其中 $100 \le N \le 1000$。

●●1.24 给出连接 $N$ 个对象需要随机产生的边的个数的近似公式，自变量是 $N$。

## 1.4 前景展望

我们在 1.3 节中提出的算法看起来是逐步改进的，但实际上改进的过程并不容易，因为我们所看到的算法改进过程是许多研究人员几经努力才研究出来的（参见“第一部分参考资料”）。因为问题特定而且实现是容易的，所以我们可以通过实际运行来研究评价这些算法。更进一步，我们可以量化地比较这些算法的性能（见第二章）。这本书中不是所有的算法都可以像这个例子这样改进的，我们还会研究一些难于比较和量化的算法和难于解决的问题。我们试图对所使用的算法做出客观的、科学的判断，同时用实际应用或随机产生的数据测试实现算法的程序，从而总结性质获得经验。

这个过程是我们贯穿本书针对基本问题设计算法时所采用的方法。如果可能，我们会使用和 1.2 节中设计合并-查找算法相同的步骤来设计其他算法，如下所列：

- 确定指定问题的具体需求，包括确定该问题的基本抽象操作；
- 仔细设计一个简单算法的简明实现；
- 通过实验分析和数学化的分析来提炼改进的思想，从而改进实现；
- 寻找数据结构的更高层次的抽象表示或能提高版本设计的有效操作的算法；
- 可能时要尽量保证最坏情况的性能，最好在实际数据上测试性能。

对于 1.2 节那样的实际问题在性能上的显著提高使得算法设计成为令人兴奋的研究领域；几乎没有其他设计方法能使加速比达到几百万、几十亿，甚至更大。

更重要的是，随着计算能力和应用问题规模的增长，快速算法和慢速算法之间的差距越来越大。一台新的计算机可能比旧的机器快 10 倍，并且所处理的数据是旧机器所处理的数据的 10 倍，但如果我们使一个差的算法（如，快速查找算法）在新的计算机上运行，它的运行时间会是在旧机器上的 10 倍。这句话看似不可思议，但可以用一个简单的等式 $(10*N)^2/10=10N^2$ 加以证明（我们在第二章中会看到）。随着计算能力的提高，我们可以处理更大规模的问题，同时高效算法的重要性也提高了。

设计一个高效算法是具有实际回报的活动。如连通问题所示，一个简单的问题可以引导我们研究很多不但有用有趣而且在理解方面也有挑战性的算法。我们还将遇到很多已在许多实际问题中研究若干年的算法。随着科学计算和商业问题等应用领域的扩大，应用那些高效算法解决已有问题并为新的问题设计有效算法也变得日趋重要。

**练习**

1.25 假设有一台旧计算机和一台新计算机，新计算机的运行速度是旧计算机的 10 倍，如果在新计算机上运行连接个数为 10 倍于原问题的带权的快速合并算法，那么新计算机执行这一新任务比旧计算机完成原任务慢多少？

1.26 当我们使用的算法要执行 $N^3$ 条指令时，回答练习 1.25。

## 1.5 总结

本节简要介绍本书的各个主要部分，以及涉及到的主要问题和方面。我们尽可能地涵盖基础算法。所涉及的领域有些是计算机科学的核心领域，我们会深入学习那些应用广泛的基本算法；其他算法不仅需要计算机领域的知识，还需要相关领域的内容（例如数值分析，操作研究等），对于这种情况，我们通过基本方法的检验来介绍这些领域。

本卷书的前四个部分，介绍了被广泛应用的算法和数据结构，以及一些对象集合的第一层抽象，这些对象具有支持大量重要的基础算法的关键元素。我们讨论的算法是研究并发展了几十年的，而且在日趋庞大的计算应用中仍扮演着重要角色。

**基本原理**（第一部分）。所谓基本原理是指我们用于实现、分析、比较算法的基本方法和原理。第一章的内容促使我们学习算法设计和分析。在第二章中我们将学习获取有关算法性能的量化信息的方法。

**数据结构**（第二部分）。数据结构与算法同样重要。我们将透彻理解数据的表示方法，并在本书中的后面部分使用。第三章中将介绍一些基本的数据结构，包括数组、链表和字符串等。第五章会介绍一些复杂的数据结构，尤其是树以及管理树所需要的算法。在第四章中，我们会学习基本的抽象数据类型（ADT），如栈和队列，以及如何用基本数据结构来实现它们。

**排序算法**（第三部分）。对文件进行排序很重要。我们会看到大量的算法，包括 Shell 排序、快速排序、归并排序和堆排序等。我们还会遇到一些相关问题的算法，如优先队列、选择和归并等。这些算法很多会成为本书中后面算法的基本组成部分。

**搜索算法**（第四部分）。搜索算法主要是从大量的数据项中搜索某一个特定的数据项，它很重要。我们会讨论用树和数字键转换搜索的基本方法和改进方法，包括二叉搜索树、平衡树、散列和数字搜索树等这些适用于大文件的搜索方法。最后说明了几种方法之间的关系，对比了不同排序算法的性能统计。

第五部分到第八部分另成一卷。这几部分介绍了那些算法在各个领域深入的应用，以及在大量重要领域中的第二层抽象。我们仍然会深入地讨论算法设计和分析的技术。我们涉及的问题许多是正在研究的课题。

**字符串处理算法**（第五部分）这一部分包括大量处理（长）字符序列的方法。字符串搜索会使用模式匹配，而模式匹配会使用解析。这里还介绍了文件压缩技术。在处理一些基本问题时，我们再一次讨论了它的高级主题。

**几何算法**（第六部分）这一部分介绍了解决关于点和线（或其他简单几何对象）问题的方法。我们会学习如何寻找点集的凸包、几何对象的交集、近点问题以及多值搜索问题。这些方法中许多都实现了基本的排序算法和搜索算法。

**图形算法**（第七部分）这部分对解决许多重要而且有一定难度的问题很有帮助。在图形中检索的通用策略可以应用于基本的连通问题，包括最短路径、最小生成树、网流和匹配等。这些算法的统一处理体现出它们是基于同一过程的，这一过程又是依赖于基本优先队列 ADT 的。

**高级主题**（第八部分）这一部分的讨论目的是将本书中的相关材料应用到其他一些高层研究领域中去，我们先介绍一些算法设计和分析的主要方法，包括分治、动态程序设计、随机化等。对于线性方程、快速傅立叶变换和 NP 完全性问题等高级主题，我们仅仅从本书中所遇到的基本问题引出的高深领域的角度加以介绍。

算法研究十分有趣，因为它是一个有丰富传统的全新领域（几乎所有我们研究的算法都是在近 50 年之内产生的，而且一些只是最近才发现的，只有少数算法是几千年前就已知的）。新的发现经常有，但很少能被完全理解。本书中，我们既会学习一些复杂的算法，也会学习一些简单的算法。我们的目标是理解已有的算法，并研究在不同程序中有应用潜力的新算法。通过这样做，我们可以发掘出许多有用的工具，并开发出算法思想的风格，以便应对将要到来的计算挑战。

# 第二章

# 算法分析原理

想要充分理解算法并用它有效地解决实际问题，算法分析是一个关键。虽然我们不能对我们运行的每个程序都进行广泛的实验和深入的数学分析，但我们可以在一个基本框架之内进行经验性的测试和近似分析，以帮助了解算法的性能特征，从而能够比较不同的算法，应用它们解决实际问题。

用数学的分析方法对一个复杂算法的性能进行精确的描述，这个想法初看起来似乎有点令人畏惧，而且我们的确经常需要参考一些研究文献来查阅特定的数学知识。我们不打算在本书中覆盖所有的分析方法，进而加以总结，不过大家应该从一开始就清楚地意识到我们是在一个坚实的数学基础上进行算法的比较。其实，对于本书中很多最重要的算法，只要掌握很少的一些基础分析技术，并加以适当的应用，就能获得相当多的有用信息。贯穿本书我们都会强调最基本的分析结果和分析方法，尤其是当它们有助于理解基本算法的内部机制时。本章的主要目标是提供使用算法本身进行工作的上下文和工具。

第一章中的例子提供了一个可以演示很多算法分析基本概念的上下文，为了使本书的内容保持连贯，我们会经常回到这个合并-查找算法。另外，我们也会介绍一些新的例子，见 2.6 节。

算法分析在设计和实现算法的整个过程中都起到了一定的作用。首先，我们已经看到，如果设计算法时选择得宜，可以将运行时间降低到几千分甚至几万分之一。当我们考虑更高效的算法时，将会看到这样的选择更有挑战性，所以我们必须更深入地研究它们的性能。当追求“最好”的算法时，我们发现开发在实践和理论问题中有用的算法都是富于挑战性的。

对算法分析的方法进行综述性的概括也是本书的目标之一（参见“第一部分参考资料”），不过现在我们先只考虑一些基本的内容，使我们能够：

- 演示分析的过程；
- 统一描述我们所使用的数学约定；
- 为高级技术的讨论提供基础；
- 对以后进行算法比较的科学根据和原理有一个理解。

最重要的是，算法和对它们的分析总是互相纠缠在一起的。在这本书里，我们不打算探讨艰深的数学知识，但是我们的确需要足够的数学基础，才能更好地理解我们的算法，并更有效地使用它们。

## 2.1 实现和经验分析

在设计和开发算法时，我们一般使用层次化的抽象操作，以帮助理解我们需要解决的计算问题的实质。在理论研究中，这个过程的确很有用，但有时也会使我们太过偏离实际世界中的原问题。因此，在这本书中，我们将始终立足于实际的编程语言——所有的算法都将以 C++语言给出。这样可能会使读者对算法和算法的实现这两个概念产生混淆，不过相对于我们得到的好处，这个小小的代价还是可以接受的。

实际上，使用实际的编程语言认真构建程序是表达算法的一个有效途径。在本书中，我们将考察大量重要、高效的算法，都用 C++实现，简洁而且精确。算法的自然语言描述或者抽象的高层描述一般都存在含糊、不完整的问题；真正的实现能够强迫我们找到精确的表达，从而不会淹没在细节里。

我们将使用 C++表达我们的算法，不过这本书是讨论算法，而不是 C++编程的。当然，在我们对很多重要的任务考虑 C++实现时，如果 C++中有某种特殊的机制能更方便或有效地完成任务，我们会使用该机制；但绝大部分的实现在任何一种现代编程语言中都适用。将第一章中的程序或本书的其他程序翻译成其他编程语言是一件很直接的事。少数情况下，如果其他语言中有某种特定的机制特别适合实现某个算法，我们也会指出来。总之，我们的目标是借助 C++来表达我们考虑的算法，而不是专门介绍特定于 C++的实现问题。

如果某个算法的实现是一个大系统的一个部分，我们将使用抽象数据类型或类似的机制，使得当我们确定系统的某个部分最值得关心后可以修改算法及其实现。不过，刚开始的时候我们需要了解每个算法的性能特征，因为系统的设计需求可能对算法性能有很大的影响。诸如此类的设计决策一开始就必须谨慎小心，因为最后经常会发现，整个系统的性能最终由几个基本算法的性能决定，本书中将要讨论的有些问题就是这样。

这本书中的算法实现已经在很大范围内得到了有效的应用，包括大程序、操作系统和应用系统。我们试图描述一些算法，而且鼓励读者能够在给出的实现上进行实验，以获取它们的动态性质。对一些应用程序，书中给出的实现可以直接使用；而对另一些应用程序，可能还需要一些额外的工作。例如，当构建实际系统时可能需要使用更安全的编程风格；发生错误时应该能进行检查并报错；程序应该能较容易地被修改，方便地被其他程序员阅读和理解，与系统的其他部分接口良好，并且可以方便地移植到其他平台。

尽管实际应用中有这样多的要求，但在本书中进行算法分析时，我们认为性能是最重要的参数，并且会将注意力集中在几个关键的性能参数上。我们将始终追求具有更好性能的算法，尤其是当好算法更简单的时候。

不论我们的目标是解决一个大问题，还是有效地实现一个大系统的一个关键子部分，

为了有效地使用一个算法，我们需要对它的性能特征有一个了解。获得这样的了解正是算法分析的目的。

要了解算法的性能，首先可以运用的一个方法就是经验分析（empirical analysis）。这个方法一点也不神秘：给你两个解决同一个问题的算法，只要把它们都运行一遍，看哪个需要的运行时间更长就行了。这个方法可能看起来太明显了，不过它的确是很常用的比较算法性能的方法。如果一个算法比另一个算法快 10 倍，我们必然能观察到一个算法需要 3 秒钟时间完成，而另一个算法则需要等待 30 秒；但在数学分析时这个小的常量开销很容易被忽略掉。测试典型输入下的性能，得到的结果不仅可以给我们一个直观的感觉，而且这些信息有助于我们进行算法的比较，并可以用来验证数学分析的结果是否正确（表 1.1 就是一个例子）。不过，如果经验分析需要消耗很长的时间，就最好用数学的分析方法了。花一个小时、甚至一天来等一个程序运行，最后得出结论这个程序速度很慢，是一件吃力不讨好的工作，尤其是当简洁的数学分析能给我们相同结果的时候。

经验分析中我们遇到的第一个任务是开发出一个正确而完整的实现。对于一些复杂的算法，这个任务可能会成为一个大的障碍。这时候，我们也会希望在花大量的精力实现该算法之前，能通过数学的分析方法，或者凭借类似程序的经验，对它的性能有一个大概的了解。

在经验分析中，我们遇到的第二个问题是确定输入数据和其他可能对实验结果产生影响的因素。典型地，我们有三种基本的选择：使用真实（actual）数据、随机（random）数据或伪（perverse）数据。使用真实数据使我们能测量出程序在实际使用中的开销；随机数据确保我们测试的是程序，而不是数据；伪数据是指那些在实际应用中很少产生的异常数据，用它们进行测试是为了保证程序能处理任何的输入。例如，当测试一个排序算法时，我们可以输入某本书中的单词，或者一些随机产生的整数，也可以输入一堆完全相同的数。使用什么样的输入数据进行算法比较也是我们需要考虑的问题。

用经验分析的方法比较不同算法的实现时很容易产生错误，尤其是当比较中使用的机器、编译器或系统不同，或者比较时选用的输入数据不合适时。其中典型的危险是实现一个算法的代码比另一个更优。提出一个新算法的人很可能会很认真地实现这个新算法，而对用于对比的经典算法的实现就远没有那么细心了。为了保证经验分析的正确性，我们应该对它们给予相同程度的重视。

在本书中，我们经常使用的一个方法是在已有算法的基础上做微小的改动来产生新的算法（这一点在第一章中已经有所体现），这样我们对算法进行比较就合理了。更一般化地，我们试图定义一些最基本的抽象操作，并且以这些操作为基础进行算法比较。例如，表 1.1 中的经验比较结果是可信的，即使对于不同的编程语言和环境来说也是如此，因为实现它们的程序都是相似的，使用了同一组基本操作。对于某个特定的编程环境，我们可以很容易地把这些数据与实际运行时间对应起来。更经常地，我们只想知道两个程序中哪个运行得更快，或者一个改动将在多大程度上提高某个程序的时间或空间开销。

在选择算法时最容易发生的错误也许就是忽略性能因素了。相对于那些直接的解决方案，速度较快的算法一般也会比较复杂，于是有的技术人员宁愿牺牲效率采用较慢的算法，也不想对付那些额外的复杂性。但是，我们在合并-查找算法中已经看到，有时候只需要改动几行代码就可以获得巨大的性能改进。有相当多的系统用户常常在花大量时间等待一个二次方程算法，而实际上只需要对算法做一点点小改进就可以在几分之一的时间内解决问题。当我们要对付的问题规模很大时，我们必须寻找一个更好的算法，这一点以后我们会看得更清楚。

在选择算法时最容易发生的第二个错误也许就是太重视性能因素了。如果一个程序的运行时间只是几个微秒，那么将它的性能提高 10 倍也没有太大意义。即使这个程序的运行时间是几分钟，也许也不值得费时费力地使它的运行速度提高 10 倍，特别是如果我们不需要经常使用这个程序。实现并调试一个改进的算法所需要的总的时间，很可能远远大于运行一个稍慢一点的程序所需要的时间——我们大可以让计算机来完成这部分工作。更糟糕的是，有时我们以为能够改进一个程序，但花了很大的时间和精力实现这个想法之后，却发现实际上性能并没有得到提高。

如果一个程序没有写出来，我们就不能运行它，对它进行经验分析；但是我们仍可以分析它的性质，估计这个改进的大概效果。并非所有设想的改进在实际中都能得到性能的改善，我们应该了解每一步改进所能带来的好处。而且，我们可以在实现中包含参数，并利用分析帮助确定这些参数的设置。最重要的是，通过理解程序的基本性质及其资源使用的基本情况，我们有能力在计算机系统尚未建立、新算法尚未实现的时候就对它们的效率有一个估计。在 2.2 节中，我们将概述对算法性能进行基本评价的方法。

**练习**

2.1 用另一种编程语言改写第一章中的程序，并用你的实现回答练习 1.22。

2.2 从 1 数到 1 000 000 000 需要多长的时间（不考虑溢出）？对于 $N$=10，100 和 1000，分别记录在你的运行环境中运行以下程序所需的时间。如果你的编译器可以对程序进行优化，请查看是否对本程序做了优化：

```
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++;
```

## 2.2 算法分析

在这一节中，我们将介绍用数学分析进行算法性能比较的框架，为本书后面介绍分析方法和结果打好基础。我们将学习在算法分析中用到的基本数学工具，以便进一步学习对基本算法的经典分析方法，以及利用研究文献中已有的结果帮助我们理解算法的性

能特征。

我们对算法进行数学分析是为了：

- 比较实现同一任务的不同算法；
- 预测算法在新的环境下的性能；
- 设置算法参数的值。

后面我们将看到很多这样的例子。经验分析对于一些任务来说可能已经足够了，但数学分析可以发掘出更多的信息，而且更经济。

算法分析是一件很有挑战性的工作。对本书中的一些算法，人们已经有了相当完善的了解，甚至已经有了精确的数学公式可以预测在实际环境中的运行时间。这样的数学公式是通过对程序认真研究，用基本的数学量定义运行时间，并对这些量进行数学分析得出的。另一方面，本书中的另一部分算法并没有得到很好的了解，可能是因为对它们的分析涉及还没有解决的数学问题，或者已有的实现过于复杂，不适合详细分析，或者更有可能是因为输入类型不能被很好地总结出来。

在精确的分析中，有一些重要的因素不受程序员所控制。首先，C++程序将被解释为特定于某个计算机的机器代码，我们很难精确地知道一个 C++语句的执行时间，特别是当程序处于一个资源共享的环境中时，同一个程序的两次运行其性能参数都可能不同。第二，很多程序对输入数据十分敏感，性能可能会根据输入的不同而产生很大的波动。第三，很多我们感兴趣的程序人们还不能很好地理解，无法获得我们关心的数学结果。最后，两个程序有可能根本就是无法比较的：其中一个可能在某一类输入数据下很有效；另一个则在其他条件下性能更好。

尽管有这么多的不可控因素，我们依然可以较准确地预测一个特定的程序所需要的运行时间，或者知道在某个特定的环境中这个程序将比另一个程序性能更优越。而且，一般我们只需要使用很少的一组数学工具中的一种，就可以得到上述知识。算法分析的任务就在于发掘出尽可能多的关于算法性能的信息；而程序员的任务则是在为一个特定的问题选择算法时尽量利用这些信息。在本节以及后面的几节里，我们将在数学分析的理想世界中考虑问题；不过为了有效地使用我们的最好算法，有时候我们还得回到此岸世界中来。

分析一个算法的第一步是确定该算法是基于什么抽象操作的，从而将分析从实现中隔离出来。例如，在分析合并-查找算法的时候，我们就将需要执行多少次代码片段 i = a[i] 和在我们的计算机上执行一个片段需要多少纳秒分离开来了。要确定程序在特定计算机上的实际运行时间，这两个参数我们都需要知道。前者由算法的性质决定，而后者由计算机的性质决定。这样的分离允许我们不依赖于特定的实现和计算机讨论问题。

虽然一个算法中涉及的基本抽象操作的数量可能很多，但原则上，算法的性能只由其中的少数几个决定，而且一般其中最重要的那个量很容易找到。确定关键操作的一个方法是试着运行几次，使用剖析（Profiling）机制（这是很多 C++环境都支持的一个机制，

可以统计指令的执行频率）以确定程序最经常执行的部分。或者，像 1.3 节的合并-查找算法那样，我们的实现本身就是建立在几个抽象操作上的。不管是哪一种情况，算法分析的任务就是确定这几个基本操作的执行频率。我们的常用方法是寻找对这些量的一个近似估计，不过我们心里应该清楚，对重要的程序，必要的时候我们可以进行更完整的分析。另外，我们经常将近似分析的结果与经验分析相结合，以更精确地预测性能，这一点后面也将看到。

除此之外，我们还必须研究数据，为该算法的输入建立模型。一般来说，分析的时候可以考虑采取以下两种策略之一：一是假定输入是随机的，研究程序的平均情况（average-case）性能；另一种是输入伪数据，研究程序的最坏情况（worst-case）性能。对于一部分算法，确定随机输入的特征是一个比较困难的过程，不过对于另一些算法，这个工作很容易，而且能使我们得到有用的结果。平均情况可能只是数学上的一个理想状态，不能代表程序使用时的真正情况；而最坏情况是人为构造出来的异常情况，在实际应用中可能永远不会出现，但是这些分析在很多时候都能给我们一些有用的信息。例如，我们可以在分析结果的基础之上进行经验分析（见 2.1 节），如果两者吻合，就能确信分析结果是正确的；如果不一致，我们也可以通过对两者差别的研究对算法和模型有更深的了解。

在以下的三节中，我们将简要介绍本书将使用的数学工具。这部分的内容不在本书的主要讲述目标之内，如果读者具有较深的数学背景知识，或者对推导算法性能过程中的数学公式不感兴趣，可以跳过这一部分，直接学习 2.6 节，当需要的时候再回过头来阅读有关内容。不过，我们认为这些数学知识并不很艰深，而且与算法设计的关键部分有着紧密的联系，如果你想高效地使用一台计算机，最好不要忽略它们。

首先，在 2.3 节中，我们将考察一些在描述算法性能特征时常用的数学函数。接着，在 2.4 节中，将介绍 O 符号和“与…成比例”的概念，这两个概念使我们在进行数学分析时能忽略掉一些细节。然后，2.5 节将考察递归（recurrence）关系，这是我们使用数学公式归纳算法性能特征时最基本的分析手段。有了这些基础知识，2.6 节将举出一些例子，使用以上介绍的基本工具分析几个特殊的算法。

### 练习

● 2.3 用一个形如 $c_0+c_1N+c_2N^2+c_3N^3$ 的表达式精确地描述练习 2.2 中你的程序的运行时间。对于 $N=10$，100 和 1000，比较该表达式预测的时间与实际的执行时间。

● 2.4 用一个表达式精确地描述程序 1.1 的运行时间，用 $M$ 和 $N$ 作为参数。

## 2.3 函数的增长

大部分算法会有一个主要参数 $N$，它是影响运行时间的最主要因素。这个参数 $N$ 可以

是多项式的指数、需要排序或者搜索的文件的大小、一个文本字符串中字符的个数或者其他对问题规模的抽象度量：一般来说它直接与需要处理的数据集的规模成比例。当存在多于一个参数的时候（例如 1.3 节中的合并-查找算法中就有 $M$、$N$ 两个参数），我们一般将其中一个参数用另一个参数的函数表示，或者先固定其中一个（认为是常数），一次只考虑一个参数，从而不失一般性地将参数减少到一个。我们的目标是用带参数 $N$ 的数学公式表达程序的资源开销（主要是运行时间），并且希望公式尽量简单，在 $N$ 取大值时准确。本书中算法的运行时间通常与下列函数成比例：

1　大部分程序的大部分指令只执行一次，或者最多几次。如果一个程序的所有指令都具有这样的性质，我们说这个程序的执行时间是常数。

$\log N$　如果一个程序的运行时间是对数级的，则随着 N 的增大程序会渐渐慢下来。如果一个程序将一个大的问题分解成一系列更小的问题，每一步都将问题的规模缩减成几分之一，一般就会出现这样的运行时间函数。在我们所关心的范围内，可以认为运行时间小于一个大的常数。对数的基数会影响这个常数，但改变不会太大：当 $N$=1000 时，如果基数是 10，$\log N$ 等于 3；如果基数是 2，$\log N$ 约等于 10。当 $N$=1 000 000，$\log N$ 只是前值的两倍。当 $N$ 是原来的两倍，$\log N$ 只增长了一个常数因子；仅当从 $N$ 增长到 $N^2$，$\log N$ 才会增长到原来的两倍。

$N$　如果一个程序的运行时间是线性的，很可能是这样的情况：对每个输入的元素都做了少量的处理。当 $N$ =1 000 000 时，运行时间大概也就是这个数值；当 $N$ 增长到原来的两倍时，运行时间大概也增长到原来的两倍。如果一个算法必须处理 $N$ 个输入（或产生 $N$ 个输出），那么这种情况是最优的。

$N\log N$　如果某个算法将问题分解成更小的子问题，独立地解决各个子问题，最后将结果综合起来，运行时间一般就是 $N\log N$。我们找不到一个更好的形容词，就暂且将这样的算法运行时间叫做 $N\log N$。当 $N$ =1 000 000 时，$N\log N$ 大约是 20 000 000。当 $N$ 增长到原来的两倍时，运行时间超过原来的两倍，但超过得不是太多。

$N^2$　如果一个算法的运行时间是二次的（quadratic），那么它一般只能用于一些规模较小的问题。这样的运行时间通常存在于需要处理每一对输入数据项的算法（在程序中很可能表现为一个嵌套循环）中。当 $N$ =1000 时，运行时间就是 1 000 000；如果 $N$ 增长到原来的两倍，则运行时间将增长到原来的四倍。

$N^3$　类似地，如果一个算法需要处理输入数据项的三元组（很可能表现为三重嵌套循环），其运行时间一般就是三次的，只能用于一些规模较小的问题。当 $N$ =100 时，运行时间就是 1 000 000；如果 $N$ 增长到原来的两倍，运行时间将增长到原来的八倍。

$2^N$　如果一个算法的运行时间是指数级的（exponential），一般它很难在实践中使用，即使这样的算法通常是对问题的直接求解。当 $N$ =20 时，运行时间是 1 000 000；如果 $N$ 增长到原来的两倍时，运行时间将是原时间的平方！

一个程序的运行时间很可能是上面的某一项（首项）乘以一个常数，再加上一个更小

的项。这个常数系数的值和包含的项依赖于分析结果和实现细节。粗略地说，首项的系数与内层循环中执行的指令数有关：在算法设计的任何层次上，都应该小心地限制这类指令的数量。对于较大的 $N$，首项起着决定性的作用：对较小的 $N$ 或者当我们需要对算法进行细致的分析时，其他项也会起作用，使得算法的比较更加困难。但在大多数情况下，我们就简单地说一个程序的运行时间是“线性的”、“$N\log N$”、“三次的”等等。至于为什么能这么做，理由详见 2.4 节。

| 秒 | |
|---|---|
| $10^2$ | 1.7 分钟 |
| $10^4$ | 2.8 小时 |
| $10^5$ | 1.1 天 |
| $10^6$ | 1.6 周 |
| $10^7$ | 3.8 月 |
| $10^8$ | 3.1 年 |
| $10^9$ | 3.1 十年 |
| $10^{10}$ | 3.1 世纪 |
| $10^{11}$ | 几乎永不结束 |

**图 2.1 秒的转换**

如果我们以秒为单位并换算成我们熟悉的时间单位，就会更清楚地意识到 $10^4$ 和 $10^8$ 之间的巨大差别。我们可以承受一个程序运行 2.8 小时，但不可能接受一个运行 3.1 年才能完成的程序。因为 $2^{10}$ 约等于 $10^3$，因此这个表也适合 2 的指数，例如 $2^{32}$ 大约是 124 年。

要减少一个程序的总运行时间，我们的注意力最终集中在使内层循环的指令数降至最少。每条指令的引入都应该经过仔细的审查：它是否是必需的？是否有一种更有效的方法能完成同样的工作？一些程序员相信现代编译器提供的自动工具能产生最优化的机器代码，另一些则认为最好的方法是用机器或汇编语言手工书写最内层循环的代码。一般来说，在本书中我们不考虑这一层次上的优化，虽然有的时候也会注意实现某个操作需要的机器指令数，以帮助理解为什么在实践中一个算法会比另一个算法更快。

对于一些规模很小的问题，我们采用哪种方法产生的差别不大——在一台比较快的现代计算机上，完成任务都只需要一眨眼的时间。但是随着问题规模的增长，我们需要处理的数据量会变得很大，如表 2.1 所示。这时慢的算法所要执行的指令数也会变得非常之大，执行这些指令所需要的时间几乎是不可接受的，即使是在最快的计算机上也不可行。图 2.1 给出了一些以秒为单位的数到日、月、年的转换；表 2.2 举了一些例子，说明快的算法比快的计算机更能帮助我们在可以忍受的时间内解决问题。

**表 2.1 常见函数的值**

这个表列举了在算法分析中一些常见的函数的相对大小。二次函数很明显地在其中遥遥领先；当 $N$ 较小时，一些函数之间的差异可能和我们所猜想的有些出入。例如，对较大的 $N$，$N^{3/2}$ 的值比 $N\lg^2 N$ 大，而当 $N$ 取较小的值时，可能 $N\lg^2 N$ 反而要大一些。一个准确描述算法运行时间的函数可能是几个这样的函数的线性组合。我们可以很容易地将快的算法和慢的算法区分开来，因为正如我们所见，$\lg N$ 与 $N$、$N$ 与 $N^2$ 之间存在着巨大的差异；但要在几个快的算法中区分出哪个更快则需要谨慎的分析。

| $\lg N$ | $\sqrt{N}$ | $N$ | $N\lg N$ | $N(\lg N)^2$ | $N^{3/2}$ | $N^2$ |
|---|---|---|---|---|---|---|
| 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 |

表 2.2 解决大规模问题所需的时间

对于很多应用，当问题规模很大时，能够解决它们的惟一方法是找到一个有效的算法。这个表列出了当问题规模为 1 百万和 10 亿时，在运算能力分别为每秒执行 1 百万、10 亿、1 万亿条指令的计算机上，分别使用线性的、$N\log N$ 和二次的算法解决问题的最小运行时间。一个快的算法能够使我们在较慢的机器上解决问题，而使用慢的算法，即使是很快的机器也帮不了我们什么忙。

| 每秒操作数 | 问题规模为 1 百万 | | | 问题规模为 10 亿 | | |
|---|---|---|---|---|---|---|
| | $N$ | $N\lg N$ | $N^2$ | $N$ | $N\lg N$ | $N^2$ |
| $10^6$ | 几秒 | 几秒 | 几周 | 几小时 | 几小时 | 几乎永不结束 |
| $10^9$ | 瞬间 | 瞬间 | 几小时 | 几秒 | 几秒 | 几十年 |
| $10^{12}$ | 瞬间 | 瞬间 | 几秒 | 瞬间 | 瞬间 | 几周 |

我们还会碰到别的一些函数。例如，可能有一个算法接受的输入规模是 $N^2$，而运行时间与 $N^3$ 成比例，这样的算法一般认为是 $N^{3/2}$ 的。又比如某些算法有两个阶段的子问题分解，这样它的运行时间很可能是 $N\log^2 N$。从表 2.1 中可以知道，这两个函数都更接近于 $N\log N$，而不是 $N^2$。

在算法设计和分析中对数函数占有相当特殊的地位，因此我们将花一点工夫来详细介绍它。因为我们处理分析结果时一般可以忽略常数因子，所以就直接使用 $\log N$ 这个记号，而不指明它的基数了——基数从一个常数变化成另一个常数，相当于使结果乘上一个常数因子。在特殊的环境中人们习惯使用特殊的基数：在数学中，自然对数（基数 $e$=2.71828…）占有很重要的地位，为此引入了特殊的简记符号 $\log_e N = \ln N$；在计算机科学中，二进制的对数很常用，简记符号是 $\log_2 N = \lg N$。

大于 $\lg N$ 的最小整数是用二进制表示 N 所需要的位数；类似地，大于 $\log_{10} N$ 的最小整数是在十进制中表示 $N$ 所需要的数字个数。C++语句：

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

提供了一种计算大于 $\lg N$ 的最小整数的简单方法。另一个完成同样功能的类似方法是：

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

这个方法利用了如下事实：如果 $n$ 是大于 $\lg N$ 的最小整数，则 $2^n \le N < 2^{n+1}$。

偶尔我们也会迭代地进行对数运算：对一个很大的数连续取对数，例如 $\lg\lg 2^{256} = \lg 256 = 8$。正如本例所示，即使 $N$ 很大，两次取对数之后也会变得很小，因此在实际应用中我们可以把 $\log\log N$ 看作一个常数。

在经典的分析中我们还经常会遇到一些特殊的函数和数学符号，它们可以帮助我们对程序的性质进行简洁的描述。表 2.3 中总结了其中最常用的函数；下面我们将花一定的篇幅讨论其中的一些函数和它们的性质。

我们的算法和分析一般处理的都是离散的单元，所以我们经常需要用下面两个函数把

实数转化成整数：

$\lfloor x \rfloor$：小于等于 $x$ 的最大整数

$\lceil x \rceil$：大于等于 $x$ 的最小整数

例如，$\lfloor \pi \rfloor$ 和 $\lceil e \rceil$ 都等于 3；$\lceil \lg(N+1) \rceil$ 是 $N$ 的二进制表示的位数。当我们想把一个含有 $N$ 个对象的集合分成两半时也需要用到这两个函数。如果我们不知道 $N$ 是奇数还是偶数，就可以表述成一个子集中含有 $\lfloor N/2 \rfloor$ 个对象、另一个子集含有 $\lceil N/2 \rceil$ 个对象。如果 $N$ 是偶数，这两个子集的对象个数相等（$\lfloor N/2 \rfloor = \lceil N/2 \rceil$）；如果是奇数，则相差 1（$\lfloor N/2 \rfloor + 1 = \lceil N/2 \rceil$）。在 C++中，如果我们操作的是整数，可以直接计算这两个函数（例如当 $N \geq 0$ 时，$N/2$ 就是 $\lfloor N/2 \rfloor$，$N - (N/2)$ 则是 $\lceil N/2 \rceil$）；如果操作的是浮点数，则可以使用 math.h 中定义的 floor 和 ceil 计算。

**表 2.3 特殊函数和常数**

这个表总结了我们在描述算法性能时经常用到的一些数学符号和常数。如果您想了解更多的信息，请查阅“第一部分参考资料”。

| 函数 | 名字 | 特殊值 | 近似 |
|---|---|---|---|
| $\lfloor x \rfloor$ | floor 函数 | $\lfloor 3.14 \rfloor = 3$ | $x$ |
| $\lceil x \rceil$ | ceiling 函数 | $\lceil 3.14 \rceil = 4$ | $x$ |
| $\lg N$ | 2 的对数 | $\lg 1024 = 10$ | $1.44 \ln N$ |
| $F_N$ | 菲波那契数 | $F_{10} = 55$ | $\phi\ N / \sqrt{5}$ |
| $H_N$ | 调和平均数 | $H_{10} \approx 2.9$ | $\ln N + \gamma$ |
| $N!$ | 阶乘函数 | $10! = 3628800$ | $(N/e)^N$ |
| $\lg(N!)$ | | $\lg(100!) \approx 520$ | $N \lg N - 1.44N$ |

$e = 2.71828\ldots$

$\gamma = 0.57721\ldots$

$\phi = (1+\sqrt{5})/2 = 1.61803\ldots$

$\ln 2 = 0.693147\ldots$

$\lg e = 1/\ln 2 = 1.44269\ldots$

在算法分析中还经常会用到调和平均数（harmonic number），这其实是离散化的自然对数函数。$N$ 的调和平均数定义为：

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

自然对数 $\ln N$ 实际上是曲线 $1/x$ 在 1 和 $N$ 之间与 $x$ 轴所夹区域的面积；而调和平均数 $H_N$ 则是阶梯函数在 1 和 $N$ 之间与 $x$ 轴所夹区域的面积。两者的关系如图 2.2 所示。公

**图 2.2 调和平均数**

调和平均数是对曲线 $y=1/x$ 下面区域的面积的一个近似估计。常数 $\gamma$ 计量的是 $H_N$ 与 $\ln N = \int_1^N dx/x$ 之间的差值。

式：

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

其中 $\gamma = 0.57721\ldots$（这个常数被称为欧拉常数），很好地逼近了 $H_N$。与$\lceil \lg N \rceil$和$\lfloor \lg N \rfloor$的计算不同，我们最好使用库函数 log 来计算 $H_N$，而不是直接从定义出发计算。

数列：

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

是由公式：

$$F_N = F_{N-1} + F_{N-2} \qquad N \geq 2，\text{且 } F_0 = 0，\ F_1 = 1$$

定义的，叫做菲波那契数列。这个数列有很多有趣的性质。例如，相邻两项的比值接近于黄金分割率$\phi = (1+\sqrt{5})/2 \approx 1.61803\ldots$进一步的分析还发现 $F_N$ 是$\phi^N/\sqrt{5}$取整到最接近整数的结果。

有时候我们还会用到熟悉的阶乘函数 $N!$。与指数函数相似，阶乘函数的运行时间一般出现在对问题的直接解法中，而且随问题规模的增大时间开销增长很快，在实践中一般不可接受。出现阶乘函数的另一种可能原因是它记数了 $N$ 个物体排列的所有可能方法。我们可以用 Stirling 公式近似地计算 $N!$：

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

例如，Stirling 公式告诉我们用二进制数表示 $N!$ 所需要的位数大概是 $N\lg N$。

本书中我们考虑的大部分公式都是用本节介绍的这几个函数表达的。在算法分析时可能会遇到其他一些特殊的函数。例如，在十四和十五章讨论的基本搜索算法的设计和分析中，经典的二项分布和相关的泊松分布起着重要的作用。当我们遇到本节没有列举的函数时，会给出简单的介绍。

## 练习

▷ 2.5　当 $N$ 为何值时，$10N\lg N > 2N^2$？

▷ 2.6　当 $N$ 为何值时，$N^{3/2}$ 处于 $N(\lg N)^2/2$ 和 $2N(\lg N)^2$ 之间？

2.7　当 $N$ 为何值时，$2NH_N - N < N\lg N + 10N$？

○ 2.8　使得 $\log_{10}\log_{10} N > 8$ 的最小的 $N$ 是多少？

○ 2.9　证明：$\lfloor \lg N \rfloor + 1$ 是用二进制表示 $N$ 所需的位数。

2.10　在表 2.2 中增加两列：$N(\lg N)^2$ 和 $N^{3/2}$。

2.11　在表 2.2 中增加两行：每秒 $10^7$ 和 $10^8$ 条指令。

2.12　写一个 C++函数计算 $H_N$，其中可以使用标准数学库中的 log 函数。

2.13　写一个高效的 C++函数计算$\lceil \lg\lg N \rceil$。不允许使用任何库函数。

2.14　在一百万的阶乘的十进制表示中有多少位数字？

2.15　$\lg(N!)$的二进制表示有多少位？

2.16　$H_N$的二进制表示有多少位？

2.17 给出一个$\lfloor \lg F_N \rfloor$的简单表达式。

○ 2.18 求满足$\lfloor H_N \rfloor = i$，$1 \le i \le 10$的$N$的最小值。

2.19 对给定的$f(N)$，在一台每秒种能执行$10^9$条指令的计算机上解决一个至少需要运行$f(N)$条指令的问题，求最大的$N$值。其中$f(N)$分别为：$N^{3/2}$，$N^{5/4}$，$2NH_N$，$N\lg N \lg\lg N$，$N^2 \lg N$。

## 2.4 大O符号

当我们进行算法分析时，有一种数学技巧允许我们忽略细节，那就是$O$符号，读作大$O$符号。定义如下：

**定义 2.1** *如果存在常数$c_0$和$N_0$使得对所有的$N>N_0$，都有$g(N)<c_0 f(N)$，则称一个函数$g(N)$是$O(f(N))$的。*

我们一般为以下三种目的使用大$O$符号：

- 限制由于忽略数学公式中的低次项而产生的误差；
- 限制由于忽略程序中在总运行时间中贡献较小的部分而产生的误差；
- 允许我们根据算法运行时间的上限对算法进行分类。

我们将在2.7节中介绍第三种使用方法，本节先介绍前两个。

大$O$符号中隐含的常数$c_0$和$N_0$有时会隐藏一些在实践中很重要的实现细节。显然，当$N$小于$N_0$的时候，说一个算法的运行时间是$O(f(N))$的毫无意义；而$c_0$可能掩盖了为避免最坏情况而在设计中引入的大量开销。如果有两个算法，一个运行时间为$N^2$纳秒，另一个为$\log N$世纪，我们显然更倾向于前者，但在大$O$表示法中我们无法做出正确判断。

通常来说，数学分析的结果不是精确的，而是对精确的技术结果的一个近似：结果可能是一个包含一系列递降的项的表达式。如前所述，我们在程序中最关心的是最内层的循环，类似地，在数学表达式中我们最关心的是首项（leading term，最高次的项）。大$O$符号就是允许我们在操作近似的数学表达式时只记录首项，忽略其他所有的低次项，最终使得我们能得到一个简洁而又能达到相当精确程度的近似表达式。

练习2.20~2.25给出了常用的对含有大$O$符号的表达式进行的操作。其中大部分操作是直观的，数学基础较好的读者可以尝试做一做练习2.21，从定义出发证明这些操作的有效性。特别地，有的练习说可以像展开普通代数表达式一样展开带大$O$符号的表达式，然后可以忽略一般项而只留下最高次的。例如，展开表达式：

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

我们得到六项：

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

然后可以去掉低次项，近似为：

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

也就是说，当 $N$ 很大时，$N^2$ 是对表达式的一个相当好的近似。这些操作都是直观的，但大 $O$ 符号使得我们能严格、精确地表达这个过程。我们称带有一个大 $O$ 项的表达式为渐近表达式（asymptotic expression）。

下面来看一个和本书内容更相关的例子。假设经过一定的数学分析之后，我们发现某个算法有一个内层循环，平均重复 $NH_N$ 次，还有一个外层循环重复 $N$ 次，另外有一些初始化代码，只需要执行一次。经过对实现的仔细分析，我们进一步发现内层循环的每次迭代需要 $a_0$ 纳秒，外层部分需要 $a_1$ 纳秒，初始化部分需要 $a_2$ 纳秒。于是，我们知道这个程序的平均执行时间（以纳秒为单位）是：

$$2a_0N\,H_N+a_1N+a_2$$

也可以说执行时间是：

$$2a_0N\,H_N+O\,(N)$$

这个简单形式很有用，因为它告诉我们，当 N 很大时，我们在计算近似的运行时间时不必关心 $a_1$ 和 $a_2$ 的值。一般来说，运行时间的精确的数学表达式可能会有很多项，其中一些分析起来很困难。大 $O$ 符号提供了一个跳过这些复杂的项求近似答案的方法。

继续上面这个例子，我们还可以使用一个熟悉的函数 $\ln N$ 来近似表达上面这个式子。使用大 $O$ 符号，表 2.3 中的近似关系可以表达成 $H_N=\ln N+O\,(1)$，因此 $a_0N\ln N+O\,(N)$也是上述算法运行时间的一个渐进表达式。我们可以认为，当 $N$ 较大时，运行时间近似为 $2a_0N\ln N$，后者的计算相当简单。常数因子 $a_0$ 是由内层循环的指令执行时间决定的。

另外，我们不需要知道 $a_0$ 的具体值，就可以预测出输入规模为 $2N$ 时的运行时间大约是输入规模为 N 时的两倍（当 $N$ 较大时），因为：

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

这告诉我们，我们可以利用渐进公式进行准确的预测，而无须陷入实现或分析的细节。注意，如果我们在大 $O$ 表达式里只保留了首项，将不能进行这样的预测。

如上的推理告诉我们，当比较或试图预测算法的运行时间时，只需关注其首项。我们经常需要计数一个固定开销的操作的执行次数，并希望使用首项对结果进行估计，所以我们通常只记录首项，隐含地假设必要时可以进行如上的精确分析。

如果一个函数 $f\,(N)$渐进地大于另一个函数 $g\,(N)$（即当 $N$ 趋于无穷大时，$g\,(N)/f(N)$趋于 0），本书使用术语“约等于 $f(N)$”（这不是一个通用的技术上的术语）来指示 $f(N)+O\,(g\,(N))$。这样做看起来损失了一部分数学上的精确性，但是我们现在

| | |
|---|---|
| 1 | 无 |
| $\lg N$ | 很小的增长 |
| $N$ | 增长一倍 |
| $N\lg N$ | 略多于一倍 |
| $N^{3/2}$ | 乘以 $2\sqrt{2}$ |
| $N^2$ | 乘以 4 |
| $N^3$ | 乘以 8 |
| $2^N$ | 平方 |

**图 2.3　问题规模增长一倍对运行时间的影响**

如本表所示，当运行时间与某个简单函数成比例时，预测问题规模增长一倍对运行时间的影响是很容易的、理论上，仅当 $N$ 很大时我们才能使用这些结果，但在实践中这些结果很有效、反过来，这个表也为确定一个程序的运行时间函数提供了一个简单的方法：运行程序，将输入规模扩大一倍再运行一遍（应保证 $N$ 尽量大），然后反查本表即可。

最感兴趣的是算法的性能，而不是数学细节。这种情况下，我们可以认为当 $N$ 很大时，问题的规模接近于 $f(N)$。例如，已知一个量等于 $N(N\text{-}1)/2$，我们可以认为它“接近于 $N^2/2$”。这样的表示方法比精确的结果更容易理解，而且误差不大，例如当 $N=1000$ 时，两者的误差不到 0.1%。与 $O(f(N))$ 表示法相比，这样的表述损失的精确性更小。我们的目标是在描述算法的性能时尽量保持精确和简洁。

类似地，我们有时说算法的运行时间“与 $f(N)$ 成比例”，如果我们能证明时间开销等于 $cf(N)+g(N)$，其中 $g(N)$ 渐进小于 $f(N)$。如果这样的关系成立，我们可以将观察到的运行时间，例如 $N$，映射到 $2N$。图 2.3 给出了在对分析结果进行映射时常用的一些因子。结合经验分析（见 2.1 节），这个方法能将我们从确定与实现有关的常数的细节中解脱出来，或者我们也可以用它来预测当 $N$ 增长一倍时运行时间的增长情况。

图 2.4 和图 2.5 演示了大 $O$ 符号、与…成比例、约等于之间的区别。一般地，我们使用大 $O$ 符号获得算法基本的渐进性质；如果希望从经验分析推断性能则使用“与…成比例”；如果想比较性能或进行较精确的预测，则使用“约等于”。

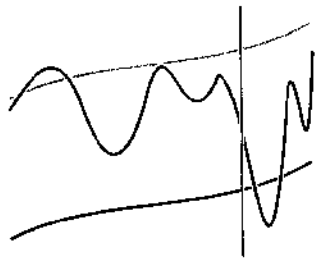
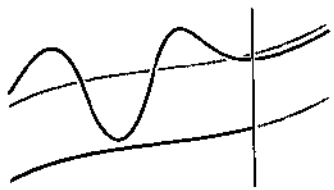


**图 2.4 用大 O 近似限制一个函数**

在这个图示中，抖动的曲线代表我们想要近似的函数 $g(N)$；黑色的、较平缓的曲线带表另一个函数 $f(N)$，是我们在近似的表达式中使用的；灰色的平缓曲线表示 $cf(N)$，$c$ 为某个常数。垂直线代表值 $N_0$，当 $N>N_0$ 时近似关系成立。当我们说 $g(N)=O(f(N))$ 时，我们希望在某条垂直线的右侧，$g(N)$ 的值在某个具有 $f(N)$ 形状的曲线下方。$f(N)$ 的性质可以更特别（例如它甚至可以不是连续的）。



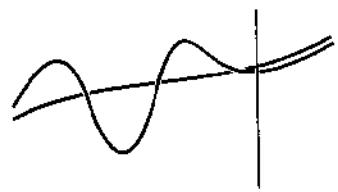


**图 2.5 函数近似**

当我们说一个函数 $g(N)$ 与 $f(N)$ 成比例时（见上方的图），我们期望它最终能像 $f(N)$ 那样增长，不过可以相差一个常数。给定 $g(N)$ 的某个值，我们可以估计 N 较大时它的值。当我们说 $g(N)$ 约等于 $f(N)$ 时（见下方的图），我们期望最终能用 $f$ 来精确地估计 $g$。

## 练习

▷ 2.20 证明 $O(1)$ 与 $O(2)$ 相等。

2.21 证明对使用大 $O$ 符号的表达式可以做以下变换：

$$
\begin{aligned}
f(N) &\rightarrow O(f(N)),\\
cO(f(N)) &\rightarrow O(f(N)),\\
O(c\,f(N)) &\rightarrow O(f(N)),\\
f(N)-g(N)=O(h(N)) &\rightarrow f(N)=g(N)+O(h(N)),\\
O(f(N))\,O(g(N)) &\rightarrow O(f(N)g(N)),\\
O(f(N))+O(g(N)) &\rightarrow O(g(N)) \quad \text{其中 } f(N)=O(g(N)).
\end{aligned}
$$

○ 2.22 说明 $(N+1)(H_N+O(1))=N\ln N+O(N)$。

2.23 说明 $N\ln N=O(N^{3/2})$。

● 2.24 说明：对任意的 $M$ 和常数 $\alpha>1$，$N^M=O(\alpha^N)$ 成立。

● 2.25 证明：

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

2.26 设 $H_k=N$，将 $k$ 近似地表示成 $N$ 的函数。

● 2.27 设 $\lg(k!)=N$，将 $k$ 近似地表示成 $N$ 的函数。

○ 2.28 已知某个算法的运行时间是 $O(N\log N)$，另一个算法的运行时间是 $O(N^3)$。这说明两个算法的相对性能如何？

○ 2.29 已知某个算法的运行时间约等于 $N\log N$，另一个算法的运行时间是 $O(N^3)$。这说明两个算法的相对性能如何？

○ 2.30 已知某个算法的运行时间约等于 $N\log N$，另一个算法的运行时间约等于 $N^3$。这说明两个算法的相对性能如何？

○ 2.31 已知某个算法的运行时间与 $N\log N$ 成比例，另一个算法的运行时间与 $N^3$ 成比例。这说明两个算法的相对性能如何？

○ 2.32 计算图 2.3 中给出的因子：对左边的每个函数 $f(N)$，求 $f(2N)/f(N)$ 的近似公式。

## 2.5 递归基础知识

在本书的学习中我们会发现，很多算法都是基于这样的思想：将一个大的问题分解成一个或多个小一点的问题，先求各小问题的解，最后得到原始问题的解。我们在第五章中将详细讨论这个方法，主要是从实际使用的角度，关注其实现和应用。在 2.6 节中也会有一个较详细的例子。在本节，我们先来看看分析这类算法的基本方法，并推导在这类算法的分析中常见的几种标准公式。了解这些公式的数学性质有助于对本书很多算法性能特征的透彻理解。

算法中的递归分解在分析中有着直接的体现。例如，这类算法的运行时间一般由子问题的规模和数量以及分解所需的时间决定；用数学语言来表达，算法在输入规模为 $N$ 时的运行时间与较小输入规模之间的函数依赖关系可以用简单的递归关系式表达。这样的式子精确地描述了算法的性能；如果需要知道运行时间，只需要解这个递归关系式。当考虑具体算法时，会涉及一些严格的参数——现在，我们先只考虑公式本身。

**公式 2.1** 如果一个递归程序循环处理输入，每次减少一项，就会出现如下的递归式：

$$C_N = C_{N-1} + N，\text{当 } N \geq 2 \text{ 且 } C_1 = 1$$

解：$C_N$ 约等于 $N^2/2$。为了解这个递归式，我们将式子逐层代入消减：

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

最后我们将得到：

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

计算 $1+2+\cdots+(N-2)+(N-1)+N$ 的求和结果很简单：把求和式与其逆序逐项相加，结果有 $N$ 项，每一项都等于 $N+1$，除以 2 就得到结果。

**公式 2.2** 如果递归程序每一步处理一半的输入，递归公式为：

$$C_N = C_{N/2} + 1，\ 当\ N \geq 2\ 且\ C_1 = 1$$

解：$C_N$ 约等于 $\lg N$。这个公式有意义的前提是 $N$ 为偶数，或者我们默认 $N/2$ 是可以整除的。现在，我们先假定 $N=2^n$，这样就保证了公式是良定义的（其中 $n=\lg N$）。现在我们可以消减这个式子：

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

对一般的 $N$，精确的结果依赖于对 $N/2$ 的解释。如果 $N/2$ 代表的是 $\lfloor N/2 \rfloor$，结果很简单：$C_N$ 是 $N$ 的二进制表示的位数，根据定义，等于 $\lfloor \lg N \rfloor + 1$。从如下事实可以很容易地得到这个结论：在任意整数 $N>0$ 的二进制表示中消去最右端的位，得到的就是 $\lfloor N/2 \rfloor$（见图 2.6）。

**公式 2.3** 如果递归程序每一步处理一半的输入，但需要检查输入的每一项，递归公式为：

$$C_N = C_{N/2} + N，\ 当\ N \geq 2\ 且\ C_1 = 0$$

解：$C_N$ 约等于 $2N$。递归式可消减为 $N+N/2+N/4+N/8+\cdots$ 的求和（与公式 2.2 类似，当 $N$ 是 2 的次幂时公式是良定义的）。如果序列是无限的，这个简单的等比数列之和恰等于 $2N$。但我们使用的是整数除法，当达到 1 时就停止了，所以最后的结果约等于精确解。精确结果与 $N$ 的二进制表示的性质有关。

**公式 2.4** 如果递归程序每一步将输入分成两半，但在划分之前、划分过程中或划分之后需要线性地遍历输入，则递归公式为：

$$C_N = 2C_{N/2} + N，\ 当\ N \geq 2\ 且\ C_1 = 0$$

解：$C_N$ 约等于 $N \lg N$。在我们讨论的这些公式中，这个公式最为常用，因为它适用于一大类标准的分治算法。

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

| $N$ | $(N)_2$ | $\lfloor \lg N \rfloor + 1$ |
|---|---|---|
| 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 |

**图 2.6 整数函数和二进制表示**

给出任意整数 $N$（中间一列）的二进制表示，去掉最右端的一位，得到的就是 $\lfloor N/2 \rfloor$。也就是说，$N$ 的二进制表示的位数比 $\lfloor N/2 \rfloor$ 二进制表示的位数大 1。因此，$N$ 的二进制表示的位数，即 $\lfloor \lg N \rfloor + 1$ 是公式 2.2 的解（假设 $N/2$ 代表的是 $\lfloor N/2 \rfloor$）。

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

解的过程与公式 2.2 类似，不过需要一点特殊的技巧：在第二步中将递归式的两边都除于 $2^n$。

**公式 2.5**　如果递归程序每一步将输入分成两半，并且需要做一些其他处理（消耗常数时间，见第五章），递归公式为：

$$C_N = 2C_{N/2} + 1，当\ N \geq 2\ 且\ C_1 = 1$$

解：$C_N$ 约等于 $2N$。求解过程与公式 2.4 类似。

我们可以用类似的技术求解以上公式的一些变体，例如不同的初始条件或者附加项略有不同；不过应当注意，一些看起来和这些公式很相似的递归式可能求解起来非常困难，处理这样的式子需要一些高级的数学知识（参见“第一部分参考资料”）。在以后的章节中我们会遇到一些这类的复杂递归式，我们留到出现时再讨论它们。

## 练习

▷ 2.33　对 $1 \leq N \leq 32$，求公式 2.2 中 $C_N$ 的值。其中 $N/2$ 代表的是 $\lfloor N/2 \rfloor$。

▷ 2.34　假设 $N/2$ 代表的是 $\lceil N/2 \rceil$，回答练习 2.33。

▷ 2.35　对公式 2.3 完成练习 2.34。

○ 2.36　假设 $f_N$ 与一个常数成比例，且

$$C_N = C_{N/2} + f_N，当\ N \geq t；\ N < t\ 时\ 0 \leq C_N < t$$

其中 $c$ 和 $t$ 都是常数。证明 $C_N$ 与 $\lg N$ 成比例。

● 2.37　给出并证明公式 2.3 到 2.5 的一般化表示（参考练习 2.36 中公式 2.2 的一般化表示）。

2.38　对 $1 \leq N \leq 32$，求公式 2.4 中 $C_N$ 的值。假定：(*i*) $N/2$ 代表 $\lfloor N/2 \rfloor$ (*ii*) $N/2$ 代表 $\lceil N/2 \rceil$ (*iii*) $2\ C_{N/2}$ 代表 $C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil}$。

2.39　假设 $N/2$ 代表 $\lfloor N/2 \rfloor$，求解公式 2.4。要求像公式 2.2 的证明那样使用 $N$ 二进制表示的性质。提示：考虑所有小于 $N$ 的数。

2.40　求解递归式：

$$C_N = 2C_{N/2} + N^2，当\ N \geq 2\ 且\ C_1 = 0，$$

其中 $N$ 是 2 的次幂。

2.41　求解递归式

$$C_N = 2C_{N/\alpha} + 1，当\ N \geq 2\ 且\ C_1 = 0，$$

其中 $N$ 是 $\alpha$ 的次幂。

○ 2.42 求解递归式：

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

其中 $N$ 是 2 的次幂。

○ 2.43 求解递归式：

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

其中 $N$ 是 2 的次幂。

● 2.44 求解递归式：

$$C_N = (2+\frac{1}{\lg N})C_{N/2}，\text{当 } N \geq 2 \text{ 且 } C_1 = 1，$$

其中 $N$ 是 2 的次幂。

● 2.45 考虑与公式 2.1 相似的递归关系，其中 $N/2$ 可以代表$\lfloor N/2 \rfloor$或$\lceil N/2 \rceil$，要求当 $N > C_0$ 时递归关系成立，当 $N \leq C_0$ 时 $C_N = O(1)$。证明所有这样的递归式的解是 $\lg N + O(1)$。

●● 2.46 参照练习 2.45，写出公式 2.2 到公式 2.5 的一般化递归关系和解。

## 2.6 算法分析举例

具备了上面三节介绍的知识，我们现在可以考虑顺序搜索（sequential search）和二分搜索（binary search）算法的分析了（这两个都是用来确定一个对象是否在一个预先给定的对象集合中出现的基本算法）。我们的目的是演示对算法进行比较的方法，而不是详细地描述具体的算法本身。为了简化描述，我们假定涉及的对象都是整数，第十二章到十六章会详细介绍这个问题的一般化情形。我们现在考虑的算法简化版本不仅体现了算法设计和分析中的很多问题，而且在实际工程中有很多直接的应用。

举一个例子，让我们想像一个信用卡公司有 $N$ 张信誉有问题或被盗的信用卡，现在要检查 $M$ 个事务，看其中是否涉及这 $N$ 张不良的卡。假设在这个应用问题中 $N$ 和 $M$ 都很大（$N$ 是 $10^3$~$10^6$ 的数量级，$M$ 是 $10^6$~$10^9$ 的数量级）。我们的分析目标是估计出当参数在这个范围内时算法的运行时间。

程序 2.1 实现了对搜索问题的直接求解过程。为了和第四部分用到的其他代码兼容，我们将它封装成一个接受一个数组并进行处理的 C++函数（数组的定义见第三章），不过读者不必理会封装的细节就可以理解这个算法：将所有的对象存放在一个数组中；然后对每个事务，从头到尾顺序地遍历整个数组，检查每个元素，看是否是我们要搜索的对象。

**程序 2.1 顺序搜索**

这个函数检查一个数 v 是否在一个事先给定的数字集合 a[l]，a[l+1]，…，a[r]中。检查的方法是从第一个元素开始，依次与每个数字逐一比较。如果一直到数组尾还没有找到要搜索的数，则返回 -1；

否则返回对应这个数字的数组元素下标。

```
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;
  }
```

在分析这个算法的时候，我们立刻注意到算法的运行时间与要搜索的对象是否在数组中有关：我们必须检查所有的 $N$ 个对象才能确定搜索是不成功的，但如果搜索成功，可能在第一个、第二个或任意一个对象上结束。

由此可见，运行时间依赖于数据。如果所有的搜索要找的数恰好都是数组的第一个元素，这个算法是很快的；如果恰好都是最后一个元素，它会很慢。我们将在 2.7 节中讨论预测性能和保证性能的区别。在这个例子中，我们可以保证至多需要检查 $N$ 个数。

但是，为了进行预测，我们需要对数据的性质做一些假设。在这个例子中，我们可以假设所有数字都是随机选择的。这个假设默认表中的每个数字都等概率地可能是我们要搜索的那个数。另一方面，我们发现搜索的性质也很关键，因为对于随机选择的数字，我们很可能一次成功的搜索也遇不到（见练习 2.48）。对于一些应用问题，搜索成功的事务的数量可能很大；而对于另一些应用，这样的事务可能很少。为了避免在模型中涉及应用问题的性质，我们将这两种情况（成功搜索和不成功搜索）分开来讨论，并分别进行分析。这个例子告诉我们，要进行有效的分析，关键是要对应用建立一个合理的模型。我们的分析结果将依赖于成功检索所占的比例；事实上，我们可能需要根据这个参数为不同的应用选择不同的算法。

**性质 2.1**　*顺序搜索算法对每个不成功的搜索要检查 $N$ 个数字，对成功的搜索平均检查约 $N/2$ 个数字。*

如果表中的每个数都等可能成为搜索的目标，则一次搜索的平均时间开销为：

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

■

性质 2.1 说明程序 2.1 的运行时间与 $N$ 成比例（即比较两个数字的平均时间为常数）。因此，如果对象的数量增长一倍，可以预期搜索所花的时间也增长一倍。

如果表中的数是按顺序排列的，我们可以加快顺序搜索在不成功搜索情形时的速度。给一个表中的数排序是第六章到第十一章的主题，我们会讨论很多完成这个任务的算法，大部分可以在与 $N\log N$ 成比例的时间内解决这个问题，这与 $M$ 很大时搜索所花的时间相比是微不足道的。在一个已排序的表中，如果我们遇到一个比要搜索的数大的数，就可以立即停止搜索。这个改进能使顺序搜索的开销降低到每个不成功的搜索检查约 $N/2$ 个数，和成功的搜索相同。

**性质 2.2** 在已排序的表中顺序搜索最坏情况下需检查 $N$ 个数，平均情况检查 $N/2$ 个数。

**程序 2.2 二分搜索**

本程序实现与程序 2.1 相同的功能，但更为有效。

```
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;
  }
```

我们需要为不成功搜索也建立一个模型。我们假设搜索等可能地在表中 $N$ 个元素形成的 $N+1$ 个间隙的任一个终止，于是很容易得到如下结果：

$$(1+2+\cdots+N+N)/N = (N+3)/2$$

不成功搜索在第 $N$ 个元素之前和之后终止的开销是一样的，都是 $N$。■

性质 2.2 的另一个等价说法是：对 $M$ 个事务，对最坏情况和平均情况，顺序检索的运行时间等价于 $MN$。如果我们使事务的数量或表中对象的数量增长一倍，则运行时间也增长一倍；如果两者同时增长一倍，则运行时间是原来的 4 倍。这个结果也告诉我们：这个算法不适合很大的表。如果比较单个数据需要 $c$ 微秒，那么当 $M=10^9$ 且 $N=10^6$ 时，处理所有事务的运行时间至少是 $(c/2)\ 10^9$ 秒，根据图 2.1，大约是 $16c$ 年，这是不可接受的。

程序 2.2 是搜索问题的一个经典解法，比顺序搜索的效率高多了。它的基本思想是：如果表中的数据是按顺序排列的，我们将要搜索的数与位于表中间的数相比较，就可以排除表中一半的数：如果两者相等，说明搜索成功完成；如果前者小于后者，只需对表的左半部分实施相同的过程；如果前者大于后者，对表的右半部分实施相同的过程。表 2.7 举例说明了这个算法在一组样本数据上的操作过程。

**性质 2.3** 二分搜索至多检查 $\lfloor \lg N \rfloor + 1$ 个数。

这个性质的证明体现了在算法分析中递归关系的使用。如果用 $T_N$ 表示最坏情况下二分搜索所需的比较次数，根据该算法每次将搜索 $N$ 个元素的表减少到搜索一半的元素的性质，我们可以立刻得到：

$$T_N \le T_{\lfloor N/2 \rfloor} + 1，\ 当\ N \ge 2\ 且\ T_1 = 1$$

检索一个大小为 $N$ 的表，我们先检查中间的一个数，然后在一个规模不大于 $\lfloor N/2 \rfloor$ 的表中继续搜索。真正的开销可能小于这个值，因为成功的搜索可能在某次比较发现相等时就终止运行了，或者被搜索的表的大小可能是 $\lfloor N/2 \rfloor - 1$（当 $N$ 是偶数的时候）。类似公式

2.2 的求解过程，我们可以很快地得到结论：当 $N=2^n$ 时 $T_N \leq n+1$，并用数学归纳法证明之。■

性质 2.3 使我们能很快地解决搜索问题，即使表中有一百万个数据，处理一个事务也只需要不超过 20 次比较操作，在很多计算机上，这些比较所花的时间可能比读或写一个数字所花的时间还少。搜索问题在计算机中如此重要，因此人们还提出了不少比二分法更高效的算法，我们将在第十二章到第十六章中继续介绍。

注意，性质 2.1 和 2.2 中，我们表达时间的概念是对数据最常进行的操作。正如解释性质 2.1 时所说的，我们认为每个操作消耗一个定量的时间，因此我们可以得出结论：二分搜索的运行时间与 $\lg N$ 成比例，而顺序搜索与 $N$ 成比例。当 $N$ 增长一倍时，二分算法的运行时间受的影响很小，而顺序算法的时间增长了一倍。随着 $N$ 的增长，两个算法之间的差距越来越大。

我们可以实现并测试这两个算法来验证分析结果（性质 2.1 和 2.2）。例如，表 2.4 列举了对不同的 $N$ 和 $M$，分别使用二分搜索和顺序搜索在大小为 $N$ 的表中进行 $M$ 次搜索的运行时间（如果是二分搜索，还要包括对表中数据排序的时间）。在这里我们不再给出程序的具体实现细节，因为在第六章到第十二章中将更详细地讨论这个问题及其实现，而且这里面涉及库和外部函数的调用，以及其他将程序组合起来的技术，还要用到第三章中介绍的 sort 函数。这里我们只是想强调，经验测试是评价算法效率过程中不可缺少的一部分。

```
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
```

**图 2.7　二分搜索**

要搜索 5025 是否在表中储存的数据（见图的左栏）中，我们首先将它和 6504 比较；比较的结果告诉我们，只需要考虑数组的前半部分。接着和 4548 比较（前半部分的中间一个数），比较结果使得我们处理前半部分的第二部分。继续这个过程，每次都得到一个子数组，这个过程中一直保持如下性质：如果要搜索的数在表中，它一定留在子数组中。最后，我们得到一个只含一个元素的子数组，这个元素不是 5025，所以 5025 不在这个表中。

表 2.4 验证了我们的观点，即运行时间的函数式增长允许我们根据规模较小时的经验分析结果预测规模较大时的算法性能。数学分析和经验分析的结合为二分搜索优于顺序搜索提供了令人信服的证据。

这个例子是我们比较算法的一般化方法的一个范例：先数学化地分析该算法进行关键的抽象操作的频率，然后使用这些结果推导出运行时间的函数表达式，最后验证并扩展经验分析结果。随着我们的算法要解决的问题越来越复杂，同时确定算法性能特征的分析过程越来越复杂，我们将依赖于文献中的一些数学结果，以使我们的注意力集中在算法本身。我们不能对遇到的每个算法都进行完整的数学和经验分析，但会尽量给出本质的性能特征，并且知道，在关键的应用问题上我们能够通过严格的科学分析选择出最好的算法。

**练习**

▷ 2.47 假设有$\alpha N$ $(0\leq\alpha\leq1)$次搜索是成功的，求用程序 2.1 进行搜索的平均比较次数。

●●2.48 估计 $M$ 个随机的 10 位数至少有一个匹配上一组给定值($N$ 个)的概率。其中 $M$=10，100 和 1000，$N=10^3$，$10^4$，$10^5$ 和 $10^6$。

2.49 开发一个程序，产生 $M$ 个随机整数并存放到一个数组中，再随机产生 $N$ 个整数，记录与数组中元素匹配的数的个数（使用顺序搜索算法）。对 $M$=10，100，1000 和 $N$=10，100 和 1000 分别运行程序。

● 2.50 参考性质 2.3 写出并证明二分搜索算法的性质。

**表 2.4 顺序搜索和二分搜索的经验分析结果**

这些相对时间验证了我们的分析结果：在一个含 $N$ 个对象的表中进行 $M$ 次搜索，顺序搜索需要与 $MN$ 成比例的运行时间，二分搜索的时间与 $M\lg N$ 成比例。如果 $N$ 增长一倍，顺序搜索的运行时间也增长一倍，而二分搜索的时间变化很小。当 $M$ 较大时，随着 $N$ 的增长顺序搜索在实践中几乎不可行，但二分搜索即使对于很大的表速度也很快。

| | $M$ = 1000 | | $M$ = 10000 | | $M$ = 100000 | |
|---|---|---|---|---|---|---|
| $N$ | 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 |

其中：

S 顺序搜索（程序 2.1）

B 二分搜索（程序 2.2）

## 2.7 保证、预测和限制

大部分算法的运行时间与它们的输入数据有关。典型地，我们在进行算法分析时的目标是在某种意义上消除这样的依赖性：我们希望在讨论一个算法的性能时尽可能地不依赖于输入的数据，因为我们不知道程序每次运行会遇到什么样的数据。2.6 节中的例子展示了我们最常用的两种方法：最坏情况分析和平均情况分析。

研究算法的最坏情况性能很有意义，因为它能使我们对程序的运行时间做出保证。我们能确知某个抽象操作的执行次数不多于输入规模的一个特定函数，无论输入的是什么样的数据。例如，性质 2.3 是对二分算法的一个这样的保证，性质 1.3 是对加权快速合并的保证。如果保证的操作次数很小，如二分算法的性质，就对我们很有利，因为我们知道我们的程序不会运行得特别慢。具有良好的最坏情况性能的程序是算法设计的一个基本目标。

但是，进行最坏情况分析也存在一些困难。对于一个特定的算法，它处理一个最坏情况输入所需要的时间与处理在实际中可能遇到的数据相比，可能有很大的差距。例如，快速合并算法在最坏情况下的运行时间与 $N$ 成比例，而平均情况下只需要 $\log N$ 的时间。更重要的是，我们不是总能够证明存在一组输入使运行时间达到这个边界，只能保证一定低于这个边界。另外，对于有的问题，具有较好的最坏情况性能的算法比一般的算法复杂得多。我们可能发现一个算法具有很好的最坏情况性能，但是在处理一般的输入时比简单的算法还慢；或者虽然快但快得不多，不值得我们付出很多的额外劳动来争取。对于很多应用问题，其他方面的考虑（如可移植性或可靠性等）比提高最坏情况性能更为重要。例如我们在第一章中看到的例子，带路径压缩的加权快速合并比加权快速合并有更佳的最坏情况性能，但在处理典型的实际数据时它们的运行时间相近。

研究算法的平均情况性能也有用，因为我们可以根据它对程序的运行时间做出预测。最简单的情况，我们可以找出算法输入的精确特征。例如排序算法是处理一个由 $N$ 个随机整数组成的数组；几何算法处理的是一个含 $N$ 个点的点集，这些点随机地分布在一个平面上，坐标值在 0 和 1 之间。然后，我们可以计算每条指令的平均执行次数，将每条指令的频度乘以执行该指令需要的时间并求出总和，就得到程序的平均运行时间。

当然，进行平均情况分析也存在一些困难。首先，我们建立的输入模型可能并不能反映在实际中真正遇到的输入情况，或者根本就没有输入模型。很少有人会质疑排序算法的“随机序文件”或几何算法的“随机点集”，对于这样的模型，我们的确可能得出很好的数学结果，能够相当准确地预测出实际应用程序的性能。但是，如何为一个处理英文文本的程序的输入建立模型？即使是排序算法，在某些应用中也会用到非随机排序的模型。第二，分析过程中可能需要更深的数学知识。例如，合并-查找算法的平均情况分析难度就较大。虽然这类结果的推导过程不在本书的讲述范围之内，我们仍将通过几个经典的例子来加以说明，在必要的时候会引用一些有关的结果（幸运的是，很多最好的算法的分析都可以在经典文献中找到）。第三，知道运行时间的平均值可能还不够：我们可能还需要知道运行时间的分布情况或其他性质，这些将更难分析。特别地，我们经常对什么情况下算法会大大慢于预期值比较感兴趣。

在很多情况下，我们可以用随机的方法来解决上面一段中提出的第一个问题。例如，我们随机地生成一个数组，然后对它排序，这时我们可以认为数组中的元素顺序是随机的。对于这类算法（被称为随机化算法，randomized algorithm），平均情况性能分析得到的是严格的概率意义下的预期运行时间。而且，我们经常可以证明这样的算法变慢的概率相当小。

这类算法包括快速排序（第九章）、随机化 BST（第十三章）和散列（第十四章）。

计算复杂性领域是算法的一个分支，帮助我们理解设计算法时遇到的根本限制。它的总目标是确定解决某一问题的最好的算法在最坏情况下的运行时间，允许相差一个常数因子。这就是问题的复杂度（complexity）。

在最坏情况分析过程中使用大 $O$ 符号使得我们不必关注与机器有关的细节。我们说一个算法的运行时间是 $O(f(N))$，这个论断与输入无关。这是对算法进行分类的一个有效方法，既不依赖于输入，也不依赖于实现细节，将算法的分析与算法的实现细节分离开来。我们在分析时一般忽略常数因子；大多数情况下，我们只想知道算法的运行时间是与 $N$ 成比例还是与 $\log N$ 成比例，而不关心算法将在一台微机上还是超级计算机上运行，也不关心内层循环只有几条指令还是很多条指令。

如果可以证明解决某个问题的一个算法的最坏情况运行时间是 $O(f(N))$，我们就说 $f(N)$ 是问题复杂度的一个上界。换句话说，解决这个问题的最好算法的运行时间不会比解决该问题的任何其他算法更高。

我们一直努力改进我们的算法，但我们最终会达到一个境地，此时任何修改看起来都不能再减少运行时间了。对每一个问题，我们都希望知道什么时候可以停止寻求更好的算法，也就是说，我们需要知道问题复杂度的下界。对很多问题，我们都可以证明：解决该问题的任何算法必须进行一定量的基本操作。证明问题的下界是一个比较麻烦的过程，我们需要精心构建一个机器模型，然后在其上构建复杂的输入模型。本书将很少涉及这个主题，但它代表了算法设计中的计算障栏，我们应该有这个概念。

如果复杂度分析发现某个算法的上界等于问题的下界，我们就可以知道不必再费心去设计更快的算法，而应该集中精力于算法的实现了。例如，二分搜索是最优的，因为没有算法在最坏情况下的比较次数能比二分搜索更低（假设比较操作是互斥的）。

基于指针的合并-查找算法的上界与下界也重合了。Tarjan 在 1975 年证明了，带路径压缩的加权快速合并最坏情况下需要跟踪不多于 $O(\lg^* V)$ 个指针，而任何基于指针的算法在最坏情况下都必须至少跟踪常数个指针。也就是说，没有可能找到改进算法能保证在线性时间内解决这个问题。在实际操作中，这个差距并不明显，因为 $\lg^* V$ 很小；寻找这个问题的线性时间解法困扰了学术界很多年，Tarjan 提出的下界使大家终于放弃了这个问题。另外，这个故事也说明我们不可能避免复杂的 log*函数，因为这个函数是问题的内在性质。

对本书中的很多算法，要进行详细的数学分析和性能研究是件很麻烦的事，因此在书中将不做详细讨论。事实上，正是在这样的研究基础之上，我们才能对这些算法加以评价。

不是所有的算法都值得这样认真的研究；事实上，在设计过程中，人们更喜欢使用一些近似的性能指标来指导设计，而避开细节问题。随着设计越来越细化，分析也随之深入，需要使用更复杂的数学工具。很多时候，设计过程导致深入的复杂性分析，最后带来与实

际应用偏离很远的理论算法。一个常见的误区是认为复杂性分析的粗略结果能直接转化成高效的实际算法；这样的想法可能会带来令人不愉快的意外。另一方面，计算复杂度是一个强大的工具，能在设计过程中告诉我们什么时候达到了性能的底限，不必再追求更快的算法来缩小上界和下界之间的差距了。

本书的观点是：算法的设计、谨慎实现、数学分析、理论研究，以及经验分析对开发优美而高效的算法都起着重要的作用。我们希望能借助一些工具获得有关程序性质的信息，然后利用这些信息改进算法或开发新的程序。我们不可能对每个算法都在每种机器上实现并在每种运行环境下测试——这样太浪费时间了，但是我们可以认真地实现我们知道是高效的算法，如果需要峰值性能再改进并比较它们。贯穿本书，在必要的时候我们会详细研究那些最重要的算法，以理解为什么它们具有良好的性能。

## 练习

○ 2.51　已知一个问题的时间复杂度是 $N\log N$，另一个问题的复杂度是 $N^3$。有两个算法分别解决这两个问题。根据上述两个论断，这两个算法的相对性能如何？

## 第一部分参考资料

在程序设计方面有大量的入门读物。C++的标准参考书是 Stroustrup 的书；Kernighan 和 Ritchie 的关于 C 语言的书是 C 语言知识的最好文献，包含大量 C 程序的例子，其中很多是合法的 C++程序，并且与本书的程序风格一致。

Van Leeuwen 和 Tarjan 的书中对本书第一章中合并-查找问题算法的很多变体做了分类和比较。

Bentley 的书中包含一些详细的个案分析，针对一些很有趣的问题演示了从开发算法到实现的整个过程的几种途径。其程序风格也与本书很接近。

在算法分析方面，Aho、Hopcroft 和 Ullman 的书是度量渐进最坏情况性能的经典；Knuth 的书主要覆盖平均情况分析，是介绍了大量算法性质的权威著作。Gonnet、Baeza-Yates 和 Cormen、Leiserson、Rivest 的书是两本较新的著作，包含了最近的研究文献中的内容。

Graham、Knuth、Patashnik 的书覆盖了算法分析中经常用到的数学知识，同样的内容在 Knuth 的书中也可以找到。Sedgewick 和 Flajolet 的书对这个主题做了透彻的介绍。

A. V. Aho, 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, 1998.

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second edition, Prentice-Hall, Englewood Cliffs, NJ, 1998.

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.

# 第二部分

# 数 据 结 构

# 第三章

# 基本数据结构

组织数据以方便处理是开发计算机程序的一个必要步骤。对于很多应用程序而言，选择正确的数据结构是其实现过程中所涉及的惟一的主要决定：一旦做出了抉择，所需的算法就很简单。对于同样的数据，某些数据结构要比其他数据结构占用更多或更少的空间；对于作用在数据上的同样的操作，某些数据结构要比其他数据结构产生更高效率（或更低效率）的算法。对算法和数据结构的选择是紧密地交织在一起的，而我们也不断地通过做出正确的选择，来寻求节省时间或空间的方法。

一个数据结构并不是一个被动的对象：我们也必须考虑将要对其执行的操作（以及这些操作所用的算法）。这个概念通过数据类型（data type）的观点来公式化。在本章中，我们把主要焦点放在我们用于结构化数据的基础途径的具体实现上。我们将考虑组织数据的基本方法和控制数据的方法，接触一些展示出每种方法优点的特定例子，并讨论诸如存储管理这样的相关问题。在第四章中我们将讨论抽象数据类型（abstract data type）——使数据类型的定义从实现中分离。

这里将讨论数组、链表和字符串的特性。这些传统的数据结构有着极其广泛的应用，它们和“树”（参阅第五章）实际上构成了所有这本书里涉及的算法的基础。我们还将考虑各种各样控制这些数据结构的原始操作，以开发出一整套基本工具，它们可用于为困难的问题提出完善算法。

要研究作为大小可变的对象以及在链接数据结构中的数据存储，需要对系统怎样管理它分配给程序的存储空间有深入的了解。这里将不会详尽地介绍这个主题，因为其中很多重要讨论部分都和实际操作系统和机器有关。但我们要论述存储管理的途径和几种最基本的机制。除此以外，我们还将讨论一些特定（风格化）的方式，其中将在本书的程序中使用C++存储分配机制。

在本章结尾我们考虑几个复合数据结构（compound structure）的实例，例如链表数组和复合数组。从较低层次的数据结构建立复杂度更高的抽象结构的思想，是一个不断重现、贯穿全书的主题。我们将举出许多例子，作为稍后介绍的更高级算法的基础。

本章中涉及的数据结构是重要的构建组件，可自然地应用于C++和许多其他编程语言中。第五章将讨论另一个主要的数据结构——树（tree）。数组、字符串、链表和树是最基

本的元素，我们在这本书里讨论的绝大部分算法都以它们为根基。对于这里得出的有关建立基本抽象数据类型（这些数据类型已足以应付多种实际应用的需要了）的具体表示，第四章将讨论它们的使用。本书的剩余部分根据这里讨论的基本工具（树和抽象数据类型）衍生出大量变体，以创造可解决更困难问题的算法，以及在不同应用中作为更高层次抽象数据类型的基础、更好地迎合我们的需要的算法。

## 3.1 构建组件

在这一节中，我们将回顾在C++中用于存储和处理信息的主要的低层(low-level)结构。所有被计算机处理的数据最终都可分解到单个位（bit），但编写专门处理位的程序实在会令人烦恼不堪。类型（type）允许我们规定怎样使用多个位的特定组合，函数（function）允许我们规定作用于数据上的操作。这里通过使用C++的结构（structure）类型，使不同种类的信息组合在一起；通过使用指针（pointer）间接地引用这些信息。在这一节，我们将在呈现出组织程序的一般途径的上下文中，讨论这些C++的基本机制。其主要目标是：为本章其余部分和第四、第五章的高级数据结构奠定基础，这些结构将作为这本书所讨论的大部分算法的基础。

我们编写程序处理信息——包括从对于这个我们生存的世界的数学描述或自然语言描述中提取的信息。因此运算环境提供了对于这些描述的基本构建组件的内置支持——数字和字符。在C++中，我们的程序都建立于少数的基础数据类型：

- 整数（int）；
- 浮点数（float）；
- 字符（char）。

尽管我们也常常用整数、浮点数和字符这些一般术语，但约定俗成地用这些基本类型的C++名字（int、float和char）来指代它们。字符在高级抽象中最常使用（例如造词和造句），所以对字符数据的讨论将被推迟到3.6节，这里先讨论数字。

我们用一定数量的位来表示数字，所以int必须是位于一个特定范围内的整数，而这个范围则取决于用来表示整数的位的个数。浮点数是实际数字的近似值，用于表示实际数字的位的个数将影响我们对它的近似精确度。在C++语言里，我们用空间换取精确率：对于整数可以从int、long int或者short int这些类型中进行选择，而对于浮点数则有float或double类型。在绝大多数系统中这些类型都和低层的硬件表示相对应。尽管C++提供了一定的保证，但用于表示的位的个数、由此决定的值域（相对int而言）或精确度（相对float而言）都依赖于实际机器（参阅练习3.1）。为明确起见，在本书中一般使用int和float，除非我们想强调正被讨论的问题需要用极大的数字。

在现代编程中，我们考虑数据类型更多地是根据该程序的需要而不是机器的性能，主要为了使程序具有更佳的可移植性。因此，举例来说，我们把short int看作一个取值范围介于-32767~32767之间的对象，而不是作为一个16位的对象。此外整数的概念还包括对它们执行的操作：加法、乘法，等等。

**定义 3.1**　数据类型是一组值集合和在这些值上执行的操作的集合。

操作和类型相关，反之则不成立。当我们执行一个操作时，我们必须确保操作数和结果的类型是正确的。忽略这个责任是一种常见的编程错误。在一些情况下 C++会执行隐式的类型转换；在其他情况下我们使用显式的类型转换（cast）。例如 x 和 N 都是整数，则表达式：

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

包含了两种转换类型：(float)是一个显式类型转换，把 x 值转换为浮点数；对于 N 则依据 C++的隐式类型转换规则执行了转换，使除法操作符的两个参数都变成浮点数。

很多和标准数据类型相关的操作都内置于 C++语言中，例如算术操作。其他操作以函数的形式定义于标准函数库中；还有一些操作以 C++函数的形式，被定义在我们的程序中（参阅程序 3.1）。也就是说，数据类型的概念并不局限于内置的整数、浮点数和字符类型。作为一种组织软件的有效方法，我们常常定义自己的数据类型。当我们用 C++定义一个简单函数时，我们实际上正在创造一种新的数据类型，该函数实现的操作则被添加到用函数的参数表示的数据类型的操作集合中。实际上，每个 C++程序在某种意义上来说都是一种数据类型——数值集合（内置类型或者其他类型）及其相关操作（函数）的一个列表。这种观点也许因为太过广泛而没有多少实用价值，但我们将会看到：集中我们的焦点，以数据类型的观点来理解程序是颇有价值的。

**程序 3.1**　函数定义

在 C++中我们用于实现数据上的新操作的机制就是这里举例说明的函数定义（function definition）。

所有函数都有一个参数（argument）列表，还可能有一个返回值（return value）。这里的函数 lg 有一个参数和一个返回值，类型都是 int。这里的函数 main 没有参数，返回一个 int 值（默认情况下数值 0 表明函数成功完成）。

我们通过给出函数名和它的返回值类型来声明（declare）函数。文件的第一行引用一个库文件，其中包含了 cout、<<和 endl 的声明。代码的第二行是 lg 的声明。如果该函数在它被调用前已被定义（见下一段），则这个声明是可选的。这个声明提供了其他函数使用正确类型的参数来调用（call 或 invoke）它所需的信息。调用函数可以在一个表达式中使用被调用函数，和使用该返回值类型的变量完全一样。

我们用 C++代码定义（define）函数。所有 C++程序都包括一个主函数 main 的定义，这个程序的代码还定义了 lg。在一个函数定义中，我们给参数（也称为参变量，parameter）命名，并像使用局部变量一样用那些名字表达计算的方法。当函数被调用时，这些变量被参数所传递的值初始化，接着函数代码被执行。return 语句是结束函数执行、向调用函数提供返回值的指令。原则上调用函数不受到影响，然而我们将看到很多不遵守这个原则的例外情况。

函数声明和定义的分离为组织程序提供了更高的灵活性，例如两者可以位于不同的文件中（参阅正文部分），或者在类似于这个简单例子的程序中，我们也可以把 lg 的定义提前到 main 的定义之前并省略它的声明语句。

```
#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;
  }
```

在我们编写程序时，其中一个目标是把它们组织起来，以便在尽可能广泛的情况下派上用场。采取这个目标的原因是：我们可能在某种情况下重用旧有的程序来解决新的问题，即使新问题和该程序原本用于解决的问题也许完全没有联系。首先，通过仔细了解、精确确定一个程序所使用的操作，我们可以轻而易举地把它扩展到任何类型的、支持那些操作的数据上去。其次，通过仔细了解、精确确定一个程序的功能，我们可以随意地支配它，在解答新问题的过程中把它执行的抽象操作添加到我们所用的操作中去。

程序 3.2 使用了以 typedef 操作定义的一种简单数据类型和一个函数（函数本身在一个库函数里实现），实现了对数字的一个简单运算。主函数引用了新定义的类型，而不是数字的内置类型。通过不指定程序处理的数字类型，我们就延展了程序的潜在可用性。打个比方，这种做法就好像延长了程序的可用寿命。当一些新的环境（一个新的应用程序或是一种新的编译器或计算机）中出现一种我们打算处理的新数据类型时，我们只要更改数据类型即可达到更新程序的目的。

对于为计算平均数和标准差而开发一个独立于数据类型的程序的问题，这个例子并没有提供一个完整的一般性解决方案；它的目的也不在于此。例如这个程序需要把一个 Number 类型的数字转换为 float 类型，才可以用于计算平均值和标准差。从源代码可以看到，该程序要依赖属于 int 内置类型一部分的 float 显式类型转换；通常我们可以为任何 Number 类型显式地定义一个这样的类型转换。

如果要尝试进行算术操作以外的操作，我们很快会发现需要向数据类型添加更多操作。例如我们可能希望打印那些数字。无论何时，只要我们尝试在确定一个程序中操作的重要性的基础上发展新的数据类型，就需要在通用的程度和实现、使用的方便性之间取得平衡。

**程序 3.2 数字类型**

这个程序依照以下数学定义，计算由库程序 rand 产生的整数数列 $x_1$，$x_2$，…，$x_N$ 的平均值$\mu$和标准差$\sigma$。

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

要注意的是：按照$\sigma^2$定义的直接实现需要一次计算平均值的处理过程、一次计算数列元素和平均值之间差的平方和的处理过程。但重新调整方程使我们可以在一个处理数据过程中计算出$\sigma^2$。

依据数据类型是 int 这个事实，这里用 typedef 声明使对数据类型 int 的引用局部化。例如可以把 typedef 和函数 randNum 保存在一个独立的文件里（用一条包含指令来引用）；然后我们可以通过修改这个文件，用这个程序来测试不同类型的随机数（参阅正文）。

无论数据类型是什么，这个程序使用 int 类型作为索引，用 float 类型计算平均值和标准差。只有该数据类型向 float 的转换已被定义时，程序才有效。

```
#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;
  }
```

通过改变数据类型使程序 3.2 适用于其余类型的数字（例如说 float，而不是 int），这种想法很有价值。在 C++中有许多不同的机制可供我们使用，以充分利用对于该数据类型存在局部引用这个情况。对于这样一个小程序，最简单的方法是拷贝一份源文件，然后将 typedef 定义改为 typedef float Number，在过程体中把 randNum 改为 return 1.0*rand()/RAND_MAX;（这样会返回 0 和 1 之间的随机浮点数）。即使是这样一个小程序，这种方法也是十分不便的；因为存在两份主程序的拷贝，而且我们必须保证：以后任何对这个程序的修改都在两份拷贝中反映出来。C++中有一种可供选择的办法，就是把 typedef 和 randNum 放置到一个独立的头（header）文件（例如称为 Number.h 的文件）中，而在程序 3.2 中用以下指令取代这些部分：

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

那么我们便可以用不同的 typedef 和 randNum 生成另一个头文件；通过重命名这些文件的其中之一或另外建一个 Number.h，即可由程序 3.2 的主函数调用任意一个而完全无须修改主程序。

第三种备受推荐的软件工程惯例被 C、C++和其他语言的程序员广泛使用，方法是把程序分割成三个文件：

- 一个接口（interface），定义数据结构和声明用于数据结构的函数；
- 在接口文件声明的函数的具体实现（implementation）；
- 一个客户（client）程序，调用在接口文件声明的函数，以在更高抽象层次进行处理。

通过这种安排，我们可以将程序 3.2 的主程序用于整数或浮点数；或者扩展到其他类型的数据。只需把主程序和对应其他类型的特定代码一起编译。在以下的段落中，我们将考虑这个例子所需的精确修改。

我们把接口看作数据类型的一个定义。它是客户程序和实现之间的契约：客户程序同意只通过接口里定义的函数访问数据，实现则同意提供已允诺的函数。

对程序 3.2 的例子，接口部分应由以下声明构成：

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

第一行指定了将被加工的数据类型，而第二行指定了和该类型相关连的操作。我们可以将代码保存于一个文件（例如名为 Number.h），客户程序和实现都可以独立地引用它。

Number.h 文件中接口部分的实现是函数 randNum 的实现，它也许会由以下代码构成：

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

第一行引用系统提供的描述 rand()函数的接口；第二行引用我们正要实现的接口（我们加入这一行作为一个校验，保证我们正在实现的函数和我们已声明的函数类型相同）；最后两行是函数的代码。这些代码可被保存在一个文件（例如名为 int.c 的文件）中。函数 rand 的实际代码被保存于标准的 C++运行时库中。

一个和程序 3.2 对应的客户程序，应由针对接口的包含指令开始，该接口声明了所使用的函数。代码如下：

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

程序 3.2 的主程序可以跟随这三行。这些代码可以被保存在一个诸如名为 avg.c 的文件里。

以上段落提及的 avg.c 和 int.c 被一起编译以后，就具有和程序 3.2 相同的功能；但它们代表了一种更灵活的实现方法，这既是因为和这种数据类型相关的代码被封装起来并可以被其他客户程序调用，也是因为 avg.c 无须修改即可用于其他数据类型。我们依然假设任何由 Number 表示的数据类型都被转换成 float；C++允许我们定义这个转换，也允许按我们的意愿，把诸如+=和<<的内置操作符重新定义并作为新数据类型的一部分。在不同的数据类型中重用函数名或操作符称为重载（overloading）。

除了刚刚提及的“客户-接口-实现”方案外，还有很多其他途径支持数据类型。这个概念超越了任何特定的编程语言或实现方法。实际上，既然文件名不属于编程语言的一部分，你也许需要修改前面提到的简单方法，才能令它在你的 C++环境中顺利运行（包含规则各不相同，不同系统有关于头文件内容的不同规则或常规，一些系统要求对程序文件使用特殊的文件扩展名，例如.C 或.cxx）。C++中一个最重要的特色是类（class）的概念，它为我们定义和实现数据类型提供了便捷的方法。尽管在这一章的剩余部分我们仍将坚持使用刚刚提及的简单途径，但贯穿本书的其余部分将几乎完全使用类。第四章的主题将是：使用类来创建在算法设计中十分重要的基本数据类型；我们还将详细讨论 C++类和“客户-接口-实现”设计范例之间的关系。

这些机制存在的主要原因是：它们为面对着创建和维护大型应用系统的艰巨任务的程序员队伍提供了强有力的支持。然而对这个主题的了解十分重要，因为我们在整本书中都将探讨、利用这些机制，从而创建出简单自然的方法，以便用算法和数据结构的改良实现方法替换旧有的，也达到了比较相应于同一应用问题的不同算法的目的。

我们常常希望建立允许控制数据集合的数据结构。这些数据结构也许很庞大，也许应用范围很广，所以我们最感兴趣的是：找出执行于数据上的重要操作，了解怎样高效地实现那些操作。完成这些任务是逐步从低层次的抽象构建高层次的抽象的过程中的第一步。这个过程允许我们方便地开发出更强大的程序。C++语言中有组织地集中数据的最简单的方法是数组（array，将在 3.2 节介绍），以及接下来介绍的结构体（structure）。

结构体是集合类型，用于定义数据的组合，以便将整个组合作为一个单元来控制，但我们仍可以通过一个给定组合的个体成员的名字来引用这个成员。在 C++中，我们可用结构体定义新的数据类型和对数据执行的操作。也就是说我们可以像控制内置类型（例如 int、float）定义的数据一样控制数据的组合。正如我们将看到的那样，我们可以命名变量、把变量作为参数传递给函数，还可以做很多其他工作。

**程序 3.3 坐标点数据类型接口**

这个接口定义了一个由“一对浮点数”值组合组成的数据类型，以及“计算两点之间距离”的操作。

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

例如处理几何数据时，也许希望利用平面中的点的抽象概念。相应地，以下语句表明 point 用于代表坐标的一对浮点数：

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

而以下语句则声明了两个这种类型的变量。

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

我们可以用一个结构体的个体成员的名字引用它们，例如下面的语句把 a 设为代表点(1,1)，b 代表点(4,5)：

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

结构体也可以作为参数传递给函数。请看下面的代码：

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

这组代码定义了一个计算平面中两点之间距离的函数。这个例子展示了如何使用结构体，在典型的应用程序中把数据集结起来。

程序 3.3 是一个实现了平面中的坐标点的数据类型定义的接口：它用一个结构体代表点，包含了一个计算两点之间距离的操作。程序 3.4 是实现这个操作的函数。我们尽可能地用这样的“接口-实现”方案来定义数据，因为它们以一种清晰明了的方式封装定义（在接口中）和实现。如果要在一个客户程序中使用这种数据类型，则只需用包含指令调用该接口，并把实现部分和客户程序一起编译（或使用恰当的分割编译软件）。程序 3.4 实现包含了程序 3.3 的接口，以保证它的函数定义和客户程序的要求吻合。这种思想的目的是：确保客户程序完全不必对点的表示方法做任何假设即可使用点。在第四章里，我们将看到如何使这种分割更进一步。

刚刚给出的结构体 point 的例子比较简单，只由两个相同类型的数据项构成。一般来讲结构体可以混合不同类的数据。在本章的余下部分我们专门处理这样的结构体。

结构体允许我们把数据集合在一起；另外在 C++里，还可以把数据和对数据所执行的操作用类（class）机制绑合。第四章将以大量例子讨论定义类的细节。假如一个类定义恰当的话，我们甚至可以通过为坐标点定义符合我们需要的适当算术和类型转换操作，从而用程序 3.2 这样的程序来处理 point 类型的数据项。这种把先前定义的高级抽象操作运用到新定义的类型上的能力，是 C++编程的一个本质上的、与众不同的特色。它基于我们在语言范围内直接定义自己的数据类型的能力：不仅通过 struct 集结数据，还通过类精确地定义对数据执行的操作（以及数据结构和支持这些结构的算法）。类为贯穿全书的实现方法奠定了基础，但在详细地考虑类的定义和使用（参阅第四章）之前，我们需要讨论大量用于控制和连接数据的低级机制。

除了为我们提供 int、float 和 char 这些特定的基本类型，以及用 struct 把它们构建成为

复合类型的能力外，C++还为我们提供了间接控制数据的能力。指针（pointer）是对内存中对象的引用，通常实现为一个机器地址。举个例子，我们通过语句 int *a 把 a 声明为一个指向整数的指针，用*a 就可以访问这个整数本身。指针可被声明指向任何类型的数据。一元操作符&给出一个对象的机器地址，因此对指针的初始化很有用。例如*&a 和 a 是完全等价的。

**程序 3.4 坐标点数据类型的实现**

实现部分为程序 3.3 中声明的坐标点的函数 distance 提供了定义。它利用了一个库函数来计算平方根。

```
#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);
  }
```

经由指针间接引用一个对象比直接引用该对象更方便，也更高效（特别是对于巨大的对象而言）。从 3.3 节到 3.7 节有许多表明这种优点的例子。我们将会看到，更为重要的一点是指针可以使数据结构化，这些途径支持用于处理数据的更高效的算法。指针是很多数据结构和算法的根基。

当我们考虑一个返回多个值的函数的定义时，将呈现一个关于指针使用的简单而又重要的例子。例如以下的函数（调用来自标准库的函数 sqrt 和 atan2）把笛卡儿坐标转换为极坐标。

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

传递给函数的参数采用“按值传递”——如果函数向参数变量赋一个新的值，该赋值作用只局限于函数内部，其调用函数不会受到影响。因此这个函数不能改变指向浮点数 r 和 theta 的指针，但它可以通过间接引用改变这些数字的值。例如，如果一个调用函数有声明语句 float a, b，那么以下函数调用将导致 a 值被设为 1.414214（$\sqrt{2}$），b 值被设为 0.785398（$\pi/4$）。

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

&操作符允许我们将 a、b 的地址传递给函数，而函数也会将那些参数作为指针对待。

在 C++里我们可以用另一种参变量传递方式——“按引用传递”取得同样的效果。方法如下：

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

符号 float&的含义是“对一个浮点数的引用”。我们可以把引用看作为内置指针，每次使用时都被自动跟踪。例如当我们在这个函数里引用 theta 时，实际在引用调用函数传递的第二个参数，而不管那个参数是什么浮点数。如果像在前面的例子中那样，一个调用函数有声明语句 float a, b，则函数调用 polar(1.0, 1.0, a, b) 将导致 a 值被设为 1.414214，b 值被设为 0.785398。

到目前为止，我们主要讨论了如何定义程序处理的个体信息。在很多场合中，我们关心的是怎样和潜在的巨大数据集合（collection）打交道，而现在我们依靠基本方法达到目的。一般来说，术语“数据结构”是指一种组织信息，以提供方便、有效的途径来访问、控制数据的机制。很多重要的数据结构都基于以下本章中讨论的基本途径的其中一种或全部两种。我们可以用数组（array），把对象按一种固定有序的方式组织起来，这样做更多地使数据适合于访问而不是控制；或者用列表（list），把对象按一种逻辑有序的方式组织起来，这样做更多地使数据适合于控制而不是访问。

**练习**

▷ 3.1 在你的编程环境中，分别找出可以使用类型 int、long int、short int、float 和 double 表示的最大和最小的数字。

3.2 在你的系统中测试随机数产生器。针对以下各种组合情况：$r$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$。用 rand() % $r$ 产生 $N$ 个位于 0 和 $r$-1 之间的随机整数，并计算平均值和标准差。

3.3 在你的系统中测试随机数产生器。针对以下各种组合情况：$r$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$，生成 $N$ 个在 0~1 之间的 double 类型的随机数，通过和 $r$ 相乘、舍去小数位将随机数转变为 0~$r$-1 之间的整数，再计算平均值和标准差。

○ 3.4 对 $r$=2，4 和 16，重做练习 3.2 和 3.3。

3.5 实现必要的函数，使程序 3.2 可用于随机位（bit，取值只能为 0 或 1）。

3.6 定义一个适用于表示纸牌的 struct。

3.7 编写一个调用程序 3.3 和 3.4 中的数据类型的客户程序，完成以下任务：从标准输入读取一系列的坐标点，找出距离第一个点最近的点。

● 3.8 向坐标点数据类型（程序 3.3 和 3.4）添加一个函数，判断三点是否共线，数值误差在 $10^{-4}$ 以内。假设点都位于单位正方形（边长为 1 的正方形）内。

● 3.9 为单位正方形内的三角形（triangle）定义一个数据类型，包括一个计算三角形面积的函数。然后编写一个客户程序，生成二对 0~1 之间的 float 类型随机数，并计算所产生三角形的平均面积。

## 3.2 数组

也许最基本的数据结构就是数组（array），在 C++和绝大多数其他编程语言中都把数组定义为一个基本元素。在第一章的例子中，我们已看到过数组作为编制一个高效算法的基础的使用了；在这一节当中我们将看到更多这样的例子。

一个数组是一组相同类型数据的固定集合，它们的存储空间相邻，可通过索引访问。我们称一个数组 a 的第 i 个元素为 a[i]。在使用 a[i]之前，必须保证在数组的 a[i]空间里存储了有效的内容，这是程序员的责任。在 C++中，程序员还必须保证使用的索引是非负数且小于数组实际大小。对这两种责任的忽略是两种最常见的编程错误。

数组是最基本的数据结构，因为它实质上和所有计算机的存储器系统有直接的对应关系。使用机器语言从内存中提取一个字的内容，需要知道其地址。因此我们把整个计算机存储器想像成一个数组，存储器的地址和数组的索引相对应。大多数机器语言处理器都把涉及数组的程序翻译成高效的直接访问内存的机器语言程序。所以我们可以很有把握地假设：诸如 a[i]这样的数组访问操作被翻译成仅仅几条机器指令。

程序 3.5 给出了一个使用数组的简单例子——打印所有小于 10000 的质数。这个方法可追溯到公元前三世纪，被称为 sieve of Eratosthenes（参阅图 3.1）。算法的特点是利用以下事实：给定元素的索引后，即可高效率地访问一个数组的任何元素。具体实现有四个循环，其中三个从头到尾顺序访问数组的元素，第四个每次在数组中跳过 i 个元素。在某些情况下顺序处理是必需的；在其他情况下，使用顺序排序的原因是它的性能和其他任何方法等同。例如我们可以把程序 3.5 的第一个循环改为：

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

这样对计算没有任何影响。我们也可以用相似的方式使内层循环的次序逆转，或者可以把最后的循环改为按递减顺序打印质数。但我们不可以改变主运算中的外层循环的顺序，因为在完成测试 a[i]是否为质数之前，它需要用所有小于 i 的整数。

这里不打算详尽分析程序 3.5 的运行时间，因为这样做会让我们过多地走入数论的范畴。不过很明显运行时间和下式成正比例：

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

上式小于 $N + N/2 + N/3 + N/4 + \cdots = NH_N \sim N \ln N$。

C++中的一个与众不同的特色是：一个数组名会产生一个指向数组首元素（索引 0 所指的元素）的指针。而且，它还支持简单的指针运算（pointer arithmetic）：如果 p 是一个指向某个类型的对象的指针，那么我们就可以以该类型的对象顺序排列为假设条件编写代码——用*p 引用第一个对象，用

| 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 |

**图 3.1　Sieve of Eratosthenes**

要计算小于 32 的质数，先把所有数组项初始化为 1（如第二列所示），表明假定全部数字都是质数。因为 a[0]和 a[1]对应 0、1，所以不被使用，在图中也没有表示出来。然后我们把索引为 2、3、5 的倍数的数组项设成 0，因为这些倍数是合数。最后和依然是 1 的数组项对应的索引就是质数（如最右边一列所示）。

*(p+1)引用第二个对象，用*(p+2)引用第三个对象，以此类推。换句话说，在 C++里*(a+i)和 a[i]是等价的。

这种等价性为访问数组里的元素提供了另一种可供我们选择的机制，而且有时比索引更方便。这种机制最常用于字符数组（字符串），在 3.6 节我们再讨论它。

和结构体一样，指向数组的指针意义重大，因为它们允许我们把数组作为高层对象来加以高效率的控制。尤其是可以把一个指向数组的指针作为参数传递给一个函数，从而使该函数无需为整个数组造一份拷贝就可以访问其中的对象。当编写控制超大型数组的程序时，这种能力是不可或缺的。例如我们在 2.6 节分析的搜索函数就利用了这种特性。在 3.7 节我们将看到其他范例。

**程序 3.5 Sieve of Eratosthenes**

这个程序的功能是：如果自然数 i 是质数，就把 a[i]的值变为 1，否则变为 0。首先所有的数组元素都设成 1，以表明没有任何数已被证明是非质数。然后把被证明是合数（已知质数的倍数）的数作为访问数组元素的索引，设对应的 a[i]为 0。如果所有小于 a[i]的质数的倍数所对应的项都变为 0 以后，a[i]依然是 1，那我们就可以知道它也是质数。

因为本程序调用一个由最简单的元素类型——“0-1”开关类型组成的数组，所以直接使用由位构成的数组会比整数数组更节省空间。另外，如果 N 值过大的话，一些编程环境也许要求数组被定义为全局变量，或者我们可以动态分配它占用的存储空间（参阅程序 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;
  }
```

程序 3.5 的实现假定数组的大小必须预先知道：要为一个不同的 $N$ 值运行这个程序的话，必须改变常数 $N$ 的值并在执行以前重新编译。程序 3.6 显示了另一种途径：程序的用户可以输入 $N$ 值，并得到所有小于 $N$ 的质数作为回应。它使用了 C++中两种涉及将数组作为参数传递给函数的机制。第一种是命令行参数以一个大小为 argc 的数组 argv 的形式，传递到主程序的机制。数组 argv 是一个复合数组，由本身也是数组（字符串）的对象构成。所以我们将把它推迟到 3.7 节再加以详细讨论。目前只需要知道的是：当程序执行时变量 $N$ 获得用户输入的数字。

**程序 3.6　数组的动态存储分配**

要改变程序 3.5 计算的最大质数的值，我们需要重新编译程序。另一种方法是：程序执行时从命令行取得期望的最大值，再通过 C++操作符 new[]，用这个值为数组分配空间。譬如，如果我们编译程序并用 1000000 作为命令行参数，那么可以得到全部小于 1 百万的质数。当然前提是所用的计算机有足够快的运算速度和足够好的性能来完成计算工作。也可以用 100 进行调试，这样可以节省大量的时间和空间。我们将经常使用这种惯常的做法，然而为了简短起见，这里会省略“内存空间不足”的测试。

```
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; }
    ...
```

程序 3.6 中所使用的第二个基本机制是 new[]，这个操作符在程序执行时动态地为我们使用的数组分配（allocate）存储空间，并返回一个我们专用的指向数组的指针。对于部分编程语言，动态分配数组占用的内存空间十分困难，甚至不可能这样做；而对于另外一些编程语言，内存分配是一个自动化机制。动态分配在控制多个数组（其中一些也许必须占用十分庞大的空间）的程序中是一种核心手段。在没有动态内存分配的情况下，我们只好预先声明一个大得足以容纳任何用户被允许输入的内容的数组。而对一个大型应用程序可能使用很多数组，因此不可能对所有数组都这样做。因为程序 3.6 的代码提供了一种很灵活的手段，所以在本书中一般使用类似的代码。然而在数组大小已预先知道的特定应用中，像程序 3.5 这样较简单的方案是完全适合的。

数组不仅类似地反映出大多数计算机的访问内存数据的低层机制，还和应用程序中组织数据的自然方法相对应，因此得到了广泛的应用。譬如，数组也和表示对象的索引列表的数学术语——矢量（vector）直接对应。

C++标准库提供了类 Vector，这是一个可以像数组一样被加以索引的抽象对象。Vector 还带有可选的自动越界检查，还能增大、缩小。我们不仅可由使用它而得到数组的优点，还不必考虑检查错误索引和管理内存，这些工作都由系统完成。因为本书中关注的通常是程序的性能，因

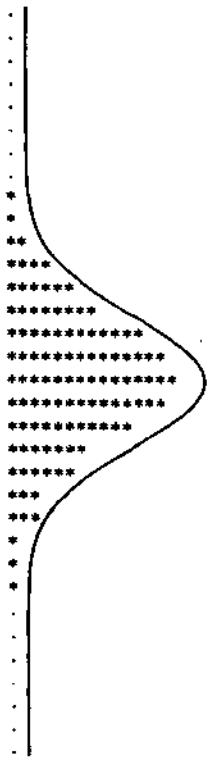


**图 3.2　抛硬币的模拟试验**

这个表显示了用 $N=32$、$M=1000$ 运行程序 3.7，以模拟 1000 次“抛一个硬币 32 次”的试验的结果。看到正面的次数由一般分布函数近似得到，并根据数据绘出图形。

此我们倾向于避免过多涉及这些由于使用数组而带来的隐藏代价，同时也应清楚：这些代码也可调用类 Vector 或被类 Vector 调用（见练习 3.14）。

程序 3.7 是一个使用数组的模拟程序的范例。它模拟出一系列的贝努利实验（Bernoulli trial），这是一个来源于概率论的熟悉的抽象概念。如果我们抛一个硬币 $N$ 次，那么看到 $k$ 次正面的概率是：

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

这种近似也称为一般近似（normal approximation），也即我们熟悉的贝尔曲线。图 3.2 显示了程序 3.7 模拟 1000 次“抛一个硬币 32 次”的试验的输出结果。在任何介绍概率论的文献中都可以找到许多更详细的关于贝努利分布和一般近似的资料，而在第十三章我们将再次接触到这些数学分布。在这里，我们主要关心的是：用数字作为数组的索引来计算它们出现的频率。对于这类操作的支持是数组的最重要的优点之一。

**程序 3.7 抛硬币的模拟**

如果我们把一个硬币抛 $N$ 次，预计会得到 $N/2$ 次正面，但从 0 次~$N$ 次的每一种情况都可能发生。这个程序从命令行获取参数 $N$ 和 $M$，把实验执行 $M$ 次。它使用了一个数组 $f$ 来记录出现 $i$ 次（$0 \le i \le N$）正面的发生频率，然后显示实验结果的条形图：每一个星号代表 10 次。

这个程序基于的操作（用运算值作为数组索引）对很多运算程序的效率都是至关重要的。

```
#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;
      }
  }
```

程序 3.5 和程序 3.7 都从手头上的数据计算数组索引。在某种意义上，当我们使用计算得出的值访问大小为 $N$ 的数组时，仅仅一个操作就考虑了 $N$ 种可能性。当我们实现这一点时，所获得的效率的提高是巨大的。贯穿全书我们都会遇到以这种途径利用数组的算法。

我们通过数组组织所有不同类型的对象，而不局限于整数。在 C++里可以声明任何内置类型或用户定义类型的数组，例如声明为结构体的复合对象。程序 3.8 展示了一个坐标平面内的点结构的数组的使用，其中利用了 3.1 节中讨论的结构体定义。这个程序还展示了数组的一种惯常的使用方法：分隔保存数据，以便在某些运算中可以通过一种有组织的方式快速访问它们。

顺便提一提，程序 3.8 的另一有趣之处在于它是一个典型的二次算法。该算法检查一个由 $N$ 个数据项组成的集合的所有组对，因此所需的时间和 $N^2$ 成正比。在本书中，只要遇到这样一个算法，我们都寻求改善的方法；这样做的原因是随着 $N$ 不断增大，算法会变得不可行。我们在 3.7 节中将看到当碰到这种情况时，如何用一个复合的数据结构，使运算在线性时间内得以完成。

**程序 3.8　最接近点的计算**

这个程序显示了一个结构体数组的使用；也是在某些计算中把数据项保存在数组中，以便稍后处理的典型情况下的代表。对 $N$ 个在单位正方形内随机生成的点（它使用了 3.1 节中描述的坐标点数据类型），计算出可以被长度小于 $d$ 的直线连接起来的两个点组成的的组对数。因为运行时间是 $O(N^2)$，所以程序对于极大的 $N$ 是不可用的。程序 3.20 提供了一种更快捷的解决方案。

```
#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;
 }
```

我们可以用类似的方法创建任意复杂度的复合类型：不仅可以用结构体组成的数组，

还可以用数组组成的数组或包含数组的结构体。我们将在 3.7 节中仔细考虑这些不同的方法。然而在这之前我们先研究链表，这是除数组外另一个组织对象集合的主要方法。

### 练习

▷ 3.10 假设 a 被声明为 int a[99]。给出执行以下两条语句后数组的内容。

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

3.11 修改关于 sieve of Eratosthenes（参阅程序 3.5）的实现：分别使用元素为字符的数组，以及元素为位的数组。确定这些修改对于程序运行所需的空间和时间的影响。

▷ 3.12 分别对 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$，通过 sieve of Eratosthenes 求小于 $N$ 的质数。

○ 3.13 对 1~1000 之间的所有 $N$，用 sieve of Eratosthenes 画出 $N$ 和相应的小于 $N$ 的质数个数的分布图。

○ 3.14 C++标准库包含了 Vector 数据类型作为数组以外的另一选择。试找出在你的系统中使用这种数据类型的方法；并确定如果在程序 3.5 中用 Vector 取代数组，程序运行时间所受到的影响。

● 3.15 分别对 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$，通过试验确定从程序 3.5 的内层循环去除对 a[i]的测试造成的影响。解释你观察到的结果。

▷ 3.16 编写一个程序，计算在输入流中出现的小于 1000 的不同整数的个数。

○ 3.17 编写一个程序，对小于 1000 的随机正整数，通过试验确定要产生多少个数才出现重复值。

○ 3.18 编写一个程序，对小于 1000 的随机正整数，通过试验确定要产生多少个数才能使每个值都至少出现一次。

3.19 修改程序 3.7 以模拟抛出的硬币得到正面的概率是 $p$ 的情况。对于硬币抛出 32 次、$p = 1/6$ 的实验，运行程序 1000 次，得到输出结果并和图 3.2 进行比较。

3.20 修改程序 3.7 以模拟抛出的硬币得到正面的概率是 $\lambda / N$ 的情况。对于硬币抛出 32 次的实验，运行程序 1000 次，得到输出结果并和图 3.2 进行比较。这种分布就是经典的泊松分布。

○ 3.21 修改程序 3.8 以打印最接近点组对的坐标。

● 3.22 修改程序 3.8 以模拟 d 维空间中执行的同样的计算。

## 3.3 链表

如果我们希望一个一个地顺序访问一组数据项的集合，可以把这些数据项组织成一个链表（linked list）。链表是一种基本的数据结构，其中每个元素都含有访问下一个元素所需的信息。链表相对于数组的主要优点是：链接为我们提供了迅速重新安排元素次序的能力。获得这种灵活性所付出的代价是不能迅速访问链表中任意一个指定元素，因为访问链表中某个元素的惟一途径是从开始一直沿着链接找下去，直到找到为止。

**定义 3.2** ***链表*是一组数据项的集合，其中每个数据项都是一个*节点*（node）的一部分，每个节点都包含指向下一节点的*链接*（link）。**

我们通过引用节点来定义节点，因此链表有时候也被称为自引用（self-referent）结构。

此外，虽然一个节点的链接通常指向不同的节点，但它也可以指向节点本身；所以链表也可是循环（cyclic）结构。当我们开始讨论链表的具体表示和应用时，这两个事实的含义就会变得更清晰。

一般来说，我们把链表看作一组元素的顺序排列的实现：从某个给定节点开始，其元素被认为处于排列的第一位。然后我们跟随着它的链接访问下一个节点，其元素被认为处于排列的第二位，以此类推。因为链表可以是循环的，相应地排列也似乎是无限的。我们接触的链表大多和一个简单的元素顺序排列相对应；对于末尾节点的链接部分，采用以下其中一种惯例来处理：

- 将其置为不指向任何节点的空链接（null link）；
- 使其指向一个不包含元素的哑元节点（dummy node）；
- 使其指向第一个节点（首节点），令链表成为循环链表（circular list）。

在每种情况下，从首节点出发，沿链接访问所有节点直至最后一个，即可定义出一组元素的顺序排列。数组也定义了一组元素的顺序排列，但是其中数据的顺序组织是通过数组中的位置而隐含确定的。数组还支持由索引直接访问元素，链表则没有这个功能。

先来考虑仅有一个链接的节点。在绝大多数的应用中我们都使用一维链表：只有第一个、最后一个节点可能例外，其余全部节点均仅有一个指向它们的链接。这和最简单的、也是我们最关心的情况相对应——链表对应元素的排列顺序。在以下的课程中我们将讨论更为复杂的情况。

链表在一些编程环境中是基本结构，但在 C++里却不是。然而，在 3.1 节中讨论的基本构建组件很适用于链表的具体实现。更明确地说，我们用指针定义链接，用结构体定义节点，如以下语句：

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

这组语句其实和对应定义 3.2 的 C++代码没有什么两样。节点由数据项（类型是 Item，这里没有进行定义）和节点指针组成，节点指针就是前面提到的链接。我们在第四章将看到更为复杂的定义表示，它们提供了更强的机动性，为某些操作提供更高效率的实现方法。但这个简单的表示对于目前讨论链表处理的基本原理已经足够了。贯穿全书我们都对链接结构使用类似的约定。

要有效使用链表，一个最重要的考虑因素是内存分配。我们已定义了一个单一结构体（struct node），但这个结构体对每一个将被使用的节点都产生一个实例，因此会出现大量的实例。无论何时，只要我们需要一个新节点，就必须为它保留一定的内存。当声明一个类型为 node 的变量时，就会在编译时为其保留内存；但在执行期间，我们常常通过调用内存管理的系统操作符，围绕着保留内存这个中心安排具体的计算。例如以下代码：

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

使用了操作符 new 为一节点保留足够的内存，并在 x 中返回一个指向该内存块的指针。在

3.5 节我们将简要介绍系统如何保留内存，因为这就是链表应用的一个绝佳例子！

在 C++中，标准的操作不仅是分配内存，还要初始化内存。为达到这个目的，我们一般为每个定义的 struct 引入一个构造函数（constructor）。构造函数是这样一个函数：它在一结构体内被定义，和该结构体分享相同的名字。在第四章我们将详细讨论构造函数。它们的用途是：为结构体中的数据提供初始值。为达到这个目的，当创建结构体的一个实例时它们就自动被调用。举个例子，如果我们用下面的代码定义一个链表节点：

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

那么语句：

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

就不仅为一个节点分配足够的内存，返回一个指向该内存块的指针 t，还把节点的 item 域设为 x，指针域设为 t。构造函数帮助我们避免了源于未初始化数据的编程错误。

现在一旦创建了一个链表节点，我们怎样访问它拥有的信息——它的数据项和链接部分呢？其实我们已经见过执行该任务所需的基本操作了：只要先引用指针，再引用结构体的成员名就可以了。例如由链接 x 指向的节点的数据项（类型是 Item）的表示方法是(*x).item，而链接部分（类型是 link）则是(*x).link。然而这些操作的使用频率是如此之高，以致 C++提供了一种更简便的表示方法：x->item 和 x->link。上述两种形式是完全等价的。类似地，因为我们需要频繁地用短语“由链接 x 指向的节点”，所以直接说“节点 x”——这个链接就指定了节点。

链接和 C++指针之间的对应关系是至关重要的，但我们必须谨记前者是一个抽象概念，后者是一个具体的实现。我们可以设计使用节点和链接的算法，并可从这个理念的许多可行的实现方法中挑选一个。比如，在这一节的末尾部分我们可以看到：用数组索引代表链接也是可行的。

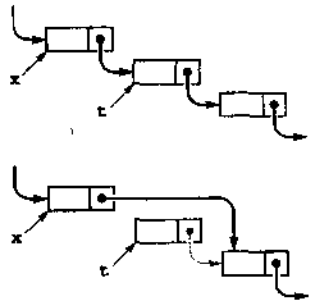



**图 3.3 链表的删除操作**

要从一个链表中删除给定节点 x 的下一个节点，先把指针 t 指向将被删除的节点，然后把 x 的链接指向 t->next。指针 t 可用于引用被删除的节点。尽管它的链接仍然指向链表，但我们通常在删除一个链表中的节点后就不再使用这样一个链接了。例外的情况多半是通过 delete 通知系统：节点占用的内存可以被回收。

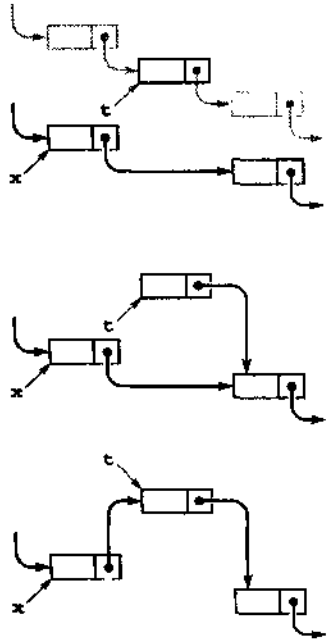



**图 3.4 链表的插入操作**

要把一给定节点 t 插入到链表中紧跟另一给定节点 x 的位置上（上图）：先把 t->next 指向 x->next（中图），再把 x->next 指向 t（下图）。

图 3.3、图 3.4 显示了对链表执行的两个最基础的操作。我们可以从一个链表中删除任何元素，使表长度减少 1；也可以向一个链表的任何位置插入一个元素，使表长度增加 1。为简单起见，对这些图我们都假设链表是循环的，而且永远不会成为空表。在 3.4 节中我们将讨论空链接、哑元节点和空表。如图所示，插入和删除操作各自需要两个 C++语句。删除节点 x 的下一个节点使用的语句是：

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

更简单直接的语句是：

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

把节点 t 插入到链表中节点 x 的下一个位置，使用的语句是：

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

插入和删除的简单易用正是链表存在的理由。数组中与之相对应的操作既不自然又不方便，因为它们需要迁移受影响元素后面的全部数组元素。

相比之下，链表却不适合用于“查找第 $k$ 个元素”（给定其索引号码，要求找到该元素）的操作。而这种高效率的访问正是数组的突出优点。在一个数组中，只要访问 a[k]即可找到第 $k$ 个元素；而在一个链表中，我们必须遍历 $k$ 个链接。在单链表上执行的另外一项极不自然的操作是“寻找一给定元素的前一个元素”。

当我们用语句 x->next = x->next->next 从链表删除一节点时，也许永远也不能再访问它了。对于我们一开始讨论的例子那样的小程序而言，这还不会产生大的影响。但我们通常将使用 delete 操作符作为一种良好的编程习惯加以提倡；delete 和 new 相对应，应该用它删除任何我们不再打算使用的节点。更具体地说，以下的语句序列：

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

不仅从链表中删除了 t，还通知系统它占据的内存可以被用于其他一些用途。当我们有庞大的链表对象或者链表对象数目巨大时，需要对 delete 特别加以注意，但在 3.5 节之前都暂时忽略它。因此我们可以集中精力，感受链接结构的好处。

在随后的章节中，我们将看到大量应用链表上的这些（或其他）基本操作的范例。因为这些操作仅涉及到几个语句，所以我们通常都直接控制链表，而不为插入、删除之类的操作定义函数。作为一个例子，接下来将讨论一个解决 Josephus

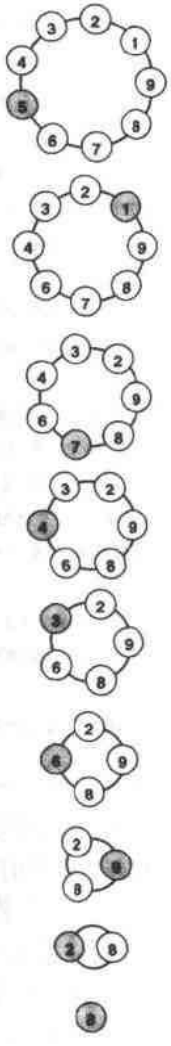


**图 3.5　Josephus 筛选范例**

本图显示了一个 Josephus 筛选的结果。所有人围成一个圆圈站立，按顺序数数，每隔第 5 个人出局，他两边的人靠拢使圆圈重新形成。

问题的程序。这个问题和 sieve of Eratosthenes 形成了一个很有趣的对比。

**程序 3.9 循环链表范例（Josephus 问题）**

我们用一个循环链表表示排成圆圈的人，每人的链接指向圆圈内站在他左边的人。整数 $i$ 代表圆内的第 $i$ 个人。先为 1 号建立一个单节点循环链表，再把 2 号到 $N$ 号依次插入到 1 号节点后面；这样就形成了一个由 1~$N$（逆时针）的圆，最后 x 指向 $N$。然后我们从 1 号开始跳过 $M$-1 个节点，把第 $M$-1 号的链接置为指向 $M$+1 号节点。一直这样做，直到只剩下一个节点为止。

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

假设有 $N$ 个人决定要选出一个领袖，方法如下：所有人围成一个圆圈，按顺序数数，每隔第 $M$ 的人出局，此时他两边的人靠拢使圆圈重新形成。这个问题就是要找出哪一个人将是最后的胜利者（一个在数学方面具有领袖潜质的人应提前计算出要占据圆圈中的哪一个位置）。

所选出领袖的号码是一个关于 $N$ 和 $M$ 的函数，我们称之为 Josephus 函数。更为一般地，我们也许希望知道所有人出局的顺序。譬如像图 3.5 所示，如果 $N$ = 9 且 $M$ =5，则出局的顺序是 5 1 7 4 3 6 9 2，8 就是选出的领袖。程序 3.9 读入 $N$ 和 $M$，并打印出这个顺序。

程序 3.9 使用了一个循环链表来直接模拟选举的过程。首先我们对 1~$N$ 建立链表：创建一个仅由代表 1 号的节点组成的循环链表，再把 2 号~$N$ 号的节点依次插入到链表中 1 号节点后面，这一步使用了图 3.4 所示的插入代码。然后我们顺着列表向前遍历，数 $M$ -1 个元素，用图 3.3 所示的代码删除下一个节点；就这样继续下去，直到只剩下一个节点为

止（这个节点指向它本身）。

对于表示一个按顺序组织的对象集合，Josephus 问题和 sieve of Eratosthenes 清楚地阐释了使用链表和使用数组两种方法之间的区别。如果用链表代替数组来解决 sieve of Eratosthenes，将是费时费力的，因为这个算法的高效有赖于数组的迅速访问任何位置的能力。而如果用数组代替链表来解决 Josephus 问题，也是费时费力的，因为这个算法的高效有赖于链表的迅速删除元素的能力。当我们选择一个数据结构时，必须清楚这个选择对用于处理数据的算法的效率有什么影响。这种数据结构和算法之间的相互作用是设计过程的核心，也是一个贯穿本书的反复出现的主题。

在 C++里，指针为链表的抽象概念的实现提供了一种既直接又方便的具体实现途径。但这个抽象的核心价值并不依赖于任何特定的实现。譬如，图 3.6 显示了如何使用整数数组来实现解决 Josephus 问题的链表。也就是说，可以用数组索引，而不是指针来实现链表。在 C++这些高级语言提供指针结构以前，链表是相当有用的。即使在现代的计算机系统中，基于数组的实现有时也是十分方便的。

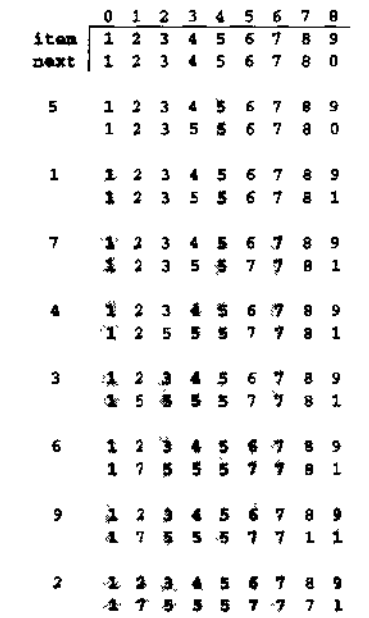



**图 3.6　链表的数组实现**

这个序列展示了用数组索引，而不是指针实现的解决 Josephus 问题（参阅图 3.5）的链表。next[0]指出了链表中索引为 0 的元素的下一个元素的索引，以此类推。最初（头三行）第 i 人的元素有索引 i-1，而我们通过把对应 i = 0~7 的 next[i]设为 i+1、next[8]设为 0 得到一个循环链表，为模拟 Josephus 筛选的过程，我们改变链接（next 数组域），而不移动元素。每两行表示在链表中向前移动 4 步（语句 x = next[x]），然后用语句 next[x] = next[next[x]] 删除第 5 个元素（左边一列所示）的结果。

## 练习

▷ 3.23　编写一个函数，根据一个给出的指向循环链表中一节点的指针，返回链表的节点个数。

3.24　编写一段代码，根据两个给出的指向循环链表中节点的指针 x 和 t，确定这两个节点之间的节点个数。

3.25　编写一段代码，根据两个给出的指向两个独立的循环链表的指针 x 和 t，把 t 指向的链表插入到 x 指向的链表中，插入点为 x 的下一个点。

● 3.26　给定指向循环链表中节点的指针 x 和 t，编写一段代码，把跟随 t 的节点移到链表中 x 的下一个节点后面的位置上。

3.27　构建链表时，程序 3.9 因为要在每个节点插入以后保持链表是循环的，所以设置了双倍于实际需要的链接值。修改这个程序，使构建链表时无需做这些额外的工作。

3.28　在一个常数因子范围内，给出作为 $M$ 和 $N$ 的函数的程序 3.9 的运行时间。

3.29　分别对 $M = 2$，3，5 和 10，$N = 10^3$，$10^4$，$10^5$ 和 $10^6$，使用程序 3.9 决定 Josephus 函数的值。

3.30　分别对 $M = 10$ 和 $N$ =2~1000，使用程序 3.9 画出 Josephus 函数和 $N$ 值的分布图。

○ 3.31　以元素 i 最初处于数组中 N-i 的位置开始，重建图 3.6 中的表。

3.32　开发程序 3.9 的另一版本，它使用索引数组实现链表（参阅图 3.6）。

## 3.4 基本的链表处理

链表引领我们进入一个明显不同于数组和结构体的计算领域。通过数组和结构体，我们可以将一个元素保存在内存中，稍后再用名字或索引访问它；这种方式和我们将一页信息放入一个文件抽屉或者一个地址本大同小异。而对于链表，我们保存信息的方式使它更难于访问，但更易于重新排列。处理以链表形式组织的数据称为链表处理（list processing）。

当使用数组时，我们容易受到涉及数组越界访问的程序错误的影响。使用链表时我们遇到的最常见的错误也是类似的：引用一个未定义的指针。另外一个常见的错误是使用一个无意中修改了的指针。这个问题出现的一个原因是：我们也许有多个指向同一节点的指针，但却不一定知道这个情况。程序 3.9 通过使用一个永远非空的循环链表而避免了几个这样的问题，因为每个链接总是指向一个定义好的节点，都可以被解释为指向链表。

为链表处理应用程序开发正确而高效的代码是一种通过实践获得的编程技巧，需要开发程序的耐心。这一节讨论的范例和练习将使我们可以更轻松地和处理链表的代码打交道。贯穿本书可以看到大量其他的例子，因为链接结构是一些我们最成功的算法的核心。

在 3.3 节已提到过，我们对链表中第一个和末尾的节点使用许多不同的常规做法。在本节中会讨论其中一些，但我们仍坚持以下方针：用术语“链表”来描述最简单的情况。

**定义 3.3** *链表是一个空链接，或是一个指向节点的链接，且这个节点包含一个元素项和指向一个链表的链接。*

这个定义比定义 3.2 更有限制性；但它和我们在编写链表处理代码时的思维模型的对应关系更密切。我们并不打算仅用这个定义而排除其他不同的常规定义，也不打算给出每种常规用法对应的特有定义，而是让两者共存。因为我们认为：使用的链表类型将和使用环境清晰地分隔开来。

其中一个最常用的对链表执行的操作是遍历（traverse）操作：我们按顺序扫描链表中的元素，对每一个都执行一定操作。譬如，若 x 指针指向链表的首节点，而尾节点有一个空指针，visit 是一个接受链表元素作为参数的过程体，那么我们可以用以下语句来遍历链表：

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

这个循环（或与之等价的 while 形式）在链表处理程序中的使用，和数组处理程序中的对应循环形式 for (int i = 0; i < N; i++)的使用一样普遍。

程序 3.10 是一个简单的链表处理任务的实现，它使一个链表中节点的顺序逆转。它接收一个链表作为参数，返回一个由相同节点构成、但排列顺序逆转的链表。图 3.7 展示了该函数在它的主循环中对每个节点所做的修改。这个图解使我

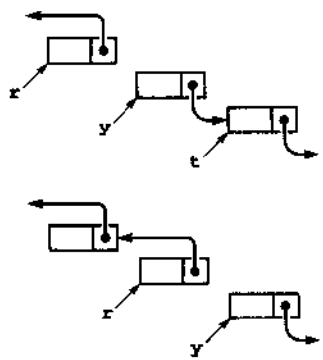



**图 3.7 链表的逆转操作**

要逆转链表的顺序，我们保持指向链表中处理完毕部分的指针 r，和指向链表中待处理部分的指针 y。本图显示了链表节点的指针是如何改变的。先保存一个指向 y 后面节点的指针 t，把 y 的链接指向 r，然后使 r 移到 y、y 移到 t。

们更容易地检查程序的每一条语句，从而确保代码按照我们的意图更改链接。程序员通常会用这些图解来了解链表处理实现的操作。

**程序 3.10　链表逆转操作**

本函数使一个链表中的节点逆转，返回一个指向末尾节点的指针。这个指针指向倒数第二个节点，以此类推；而原链表的首节点被置零，即空指针。要完成这个任务，我们需要维护链表中连续三个节点的指针。

```
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;
  }
```

程序 3.11 是另一个简单的链表处理任务的实现：重新排列链表的节点，使它们的元素项有序。它产生 $N$ 个随机整数，依照它们产生的顺序放入到一个链表中，重新排列链表的节点以使它们的元素项有序，最后打印出排列后的序列。我们在第六章中将指出：这个程序的预期运行时间和 $N^2$ 成正比，所以程序对于太大的 $N$ 没什么用处。除了这个结论，我们还把对这个程序的排序方面的讨论推迟到第六章，因为从第六章到第十章将出现非常多排序的方法。我们暂时只是要举出一个具体实现，作为链表处理程序的例子。

程序 3.11 的链表阐述了另一种常用的常规做法：在每个链表的开始保留一个称为头节点（head node）的哑元节点。链表的头节点中的元素项域可以忽略，但它的链接域必须保留作为指向链表中含有第一个元素项的节点的指针。该程序使用了两个链表：一个用于收集第一循环中的随机输入；另一个用于收集第二循环中的排序后的输出。图 3.8 解释了在程序 3.11 的一次主循环执行中程序所做的修改。我们从输入链表中取出下一个节点，找出它在输出链表中所处的位置，再把它链接到那个位置上。

**程序 3.11　链表插入排序**

本代码产生 $N$ 个 0~999 之间的随机整数，以每个节点对应一个数字来构建一个链表（第一个 for 循环）；然后重新排列节点，使遍历链表时所有数字按顺序出现（第二个 for 循环）。为了完成排序，我们保存两个链表：一个输入（未排序）链表和一个输出（已排序）链表。在每次循环的执行过程中，从输入链表中取出一节点并插入到输出链表的适当位置。通过对每个链表使用头节点（它包含指向链表中首节点的指针），代码得到简化。头节点的声明使用了构造函数，因此它们的数据成员在创建时就被初始化。

```
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;
  }
```

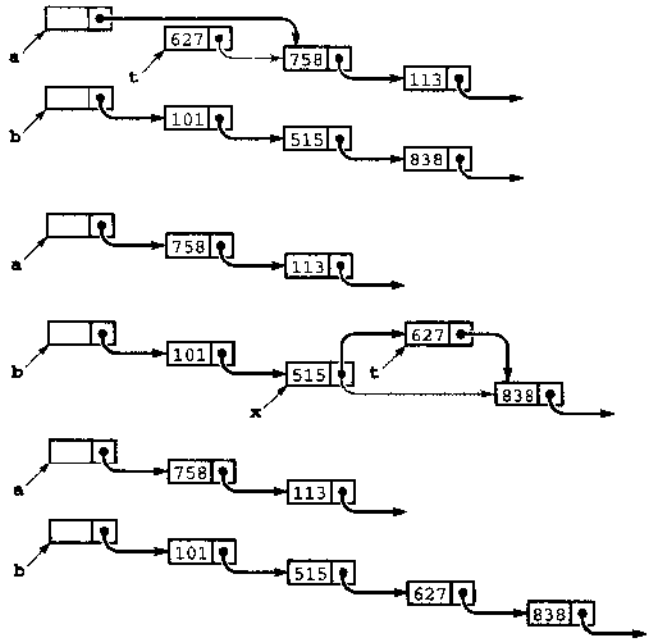



**图 3.8 链表排序**

本图描述了利用插入排序方法，把一个（由 a 指向的）未排序链表转变成（由 b 指向的）有序链表的过程中的一步。我们从未排序链表中取出首节点，用 t 保存一个指向它的指针（上图）。然后在 b 中搜索，找出插入点 x（用语句 x->next->item > t->item 查找，或 x->next = NULL 表示到达链尾），再把 t 插入到链表中 x 后的位置（中图）。这些操作使 a 的长度减少 1，b 的长度增加 1，并使 b 保持有序状态（下图）。如此不断重复，最终清空 a 并使所有节点在 b 中有序排列。

当我们考虑向已排序的链表添加首（first）节点时，在链表开始使用头节点的主要原因就变得清楚了。首节点是链表中拥有最小元素项的节点，而且可以处于链表的任何位置。所以我们有三种选择：

- 重复查找最小元素项的 for 循环，用和程序 3.9 同样的方式建立一个单节点链表；
- 每次想插入一个节点时都检查输出链表是否为空；
- 使用一个哑元头节点，它的链接指向链表的首节点，正如这里给出的实现一样。

第一种选择不优雅，而且需要额外的代码；第二种选择也不优雅，而且需要额外的时间。

使用头节点的确会引致某些额外开销（额外代码），而我们在很多普通的应用程序中可避免使用头节点。例如，也可以把程序 3.10 看作有一个输入表（原始表）和一个输出表（逆序表），而不需要在程序中使用头节点——因为全部对输出表的插入操作都发生在表的开端。我们将看到更多这样的其他应用：当它们在链表的末尾（tail）使用哑元节点而不是空链接时，编码会更简单。关于是否使用哑元节点没有硬性规定，做出选择的关键在于把编程风格和对程序性能的影响的了解结合起来。挑选出最大限度简化手头工作的常规做法是优秀程序员十分喜欢的挑战。在本书中我们将看到一些这样的权衡方案。

表 3.1 列出了一些链表常规做法的备选方案，以供参考；其余的将在练习中讨论到。在表 3.1 的所有情况中，我们都使用指针 head 来引用链表；而且为了保持一致，我们的程序都通过不同操作的给定代码来管理指向节点的链接。对于所有常规做法，分配和释放节点占有的内存、对它们填入信息都是一样的。实现同样操作且具有良好健壮性的程序会有额外的代码检查错误状态。这个表的目的在于展示不同选择方案之间的异同。

**表 3.1　链表首尾节点的常规方案**

本表对基本链表处理操作分别以五种常用的定义方案给出其实现方法。这类代码用于简单的、使用内嵌链表处理代码的应用程序。

**循环，永远非空**

| | |
|---|---|
| 头插入 | `head->next = head;` |
| 在 x 节点后插入 t 节点 | `t->next=x->next;x->next=t;` |
| 删除 x 后的节点 | `x->next = x->next->next;` |
| 遍历循环 | `t = head;`<br>`do {… t = t->next; } while (t != head );` |
| 测试是否只有一个元素 | `if (head->next == head)` |

**有头节点，尾节点为 null**

| | |
|---|---|
| 初始化 | `head = 0;` |
| 在 x 节点后插入 t 节点 | `if (x == 0) { head = t; head->next = 0; }`<br>`else {t->next = x->next; x->next = t; }` |
| 删除 x 后的节点 | `t = x->next; x->next = t->next;` |
| 遍历循环 | `for (t = head; t != 0; t = t->next)` |
| 测试是否为空 | `if (head == 0)` |

**有哑元头节点，尾节点为 null**

| | |
|---|---|
| 初始化 | `head = new node;`<br>`head->next = 0;` |
| 在 x 节点后插入 t 节点 | `t->next = x->next; x->next = t;` |
| 删除 x 后的节点 | `t = x->next; x->next = t->next;` |
| 遍历循环 | `for (t = head->next; t != 0; t = t->next)` |

| | |
|---|---|
| 测试是否为空 | `if (head->next == 0)` |
| **有哑元头节点和尾节点** | |
| 初始化 | `head = new node;`<br>`z = new node;`<br>`head->next = z; z->next = z;` |
| 在x节点后插入t节点 | `t->next = x->next; x->next = t;` |
| 删除x后的节点 | `x->next = x->next->next;` |
| 遍历循环 | `for (t = head->next; t != z; t = t->next)` |
| 测试是否为空 | `if (head->next == z)` |

**程序 3.12 链表处理接口**

在以下代码中定义了节点和链接的类型，包括希望对它们执行的操作；我们可以把代码保存在一个接口文件 list.h 中。我们为链表节点分配和释放内存空间声明自己的函数。函数 construct 可使实现更方便。这些定义允许客户程序使用 node 和相关操作，而无需依赖具体的实现细节。在第四章我们将看到：一个基于 C++类的、稍稍不同的接口可以确保客户程序不依赖具体的实现细节。

```
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);
```

还有另一种使用头节点有时可收到便利效果的重要情况。那就是我们希望把链表指针作为参数传递给函数，令函数可修改链表——这和我们对数组进行的处理大致一样。使用头节点允许函数接收或返回一个空链表。如果没有头节点，我们就需要对该函数有一种机制，在它返回一个空链表时通知调用函数。C++提供的一种解决方案是把链表指针作为引用参数传递。另一种机制，也就是程序 3.10 中函数所使用的机制，是让链表处理函数接收指向输入链表的指针作为参数，并返回指向输出链表的指针。通过这种常规做法，我们就不需要使用头节点了。而且它很适合递归链表处理，这个处理在本书中有很广泛的应用（参阅 5.1 节）。

程序 3.12 声明一组实现基本链表操作的“黑盒”函数，因此我们可以避免重复的内嵌代码以及对实现细节的依赖。程序 3.13 是前面介绍的 Josephus 筛选程序（程序 3.9）的改

造版本，可作为客户程序使用这个接口。识别出在一个计算中运用的主要操作，并在接口中对它们做出定义十分重要，因为我们借此可以灵活地考虑关键操作的不同具体实现方法，并检测它们是否有效。在 3.5 节中我们会讨论程序 3.12 定义的操作的一个具体实现（参阅程序 3.14），但也可以尝试其他的方案而完全不必更改程序 3.13（参阅练习 3.51）。这个主题将贯穿全书，反复出现。为了更简易地开发出封装好的实现程序，C++还引入了几个特别为此设计的机制。第四章将讨论这一点。

**程序 3.13 Josephus 问题的链表分配**

这个针对 Josephus 问题的程序是一个客户程序的例子，它利用了由程序 3.12 声明，并由程序 3.14 实现的链表处理基本操作。

```
#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;
  }
```

部分程序员趋向于在程序 3.12 这样的接口中为每一个低层操作定义函数，从而把所有对低层数据结构（例如链表）执行的操作都封装起来。实际上，正如我们将在第四章中看到的那样，C++的类机制使得这种做法很方便易行。然而这额外的抽象层次有时会掩盖了只有一些低层操作被包括在内的事实。在本书中需要实现高层接口的部分，我们通常直接编写作用于链接结构的低层操作，以便清楚地揭示出算法和数据结构的核心细节。在第四章我们将看到大量这样的例子。

通过添加更多的链接，我们可以获得在链表中反向移动的能力。例如可以用一个双向链表（doubly linked list）来支持“寻找一给定元素的前一个元素”的操作。双向链表中的每个节点有两个链接：一个（prev）指向前面的节点；另一个（next）指向后面的节点。结合哑元节点或循环链表，我们可确保对一个双向链表中的每一个节点，x、x->next->prev

和 x->prev->next 是等价的。图 3.9 和图 3.10 显示了在一个双向链表中实现删除（remove）、前插入（insert before）和后插入（insert after）所需的基本链接控制。请注意：这里的删除操作并不像单向链表那样，需要有关链表中前一节点或后一节点的额外信息——那些信息已被节点本身所包含。

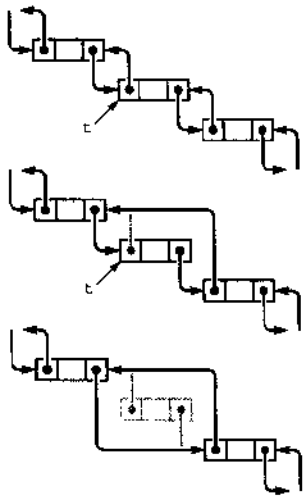



**图 3.9 双向链表中的删除操作**

正如这里的图例所示：在双向链表中，一个指向节点的指针已为删除该节点提供了足够的信息。给出 t 后，把 t->next->prev 设为 t->prev（中图），并把 t->prev->next 设为 t->next（下图）。

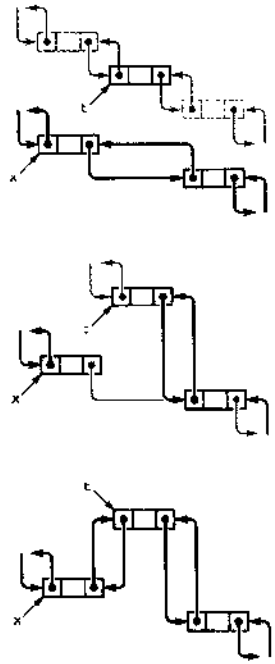



**图 3.10 双向链表中的插入操作**

为了向一双向链表中插入一节点，我们需要设置四个指针。我们可以在一个给定节点之后（如图例所示）或之前插入新节点，在给定节点 x 之后插入给定节点 t 的步骤是：把 t->next 设为 x->next，把 x->next->prev 设为 t（中图），再把 x->next 设为 t，t->prev 设为 x（下图）。

的确，双向链表的主要意义在于：当我们拥有的关于一个节点的信息仅仅是指向它的指针时，我们依然可以删除它。有两种典型的情况：一是在函数调用中链接被作为参数传递给函数，二是节点还有其他链接，并且是其他一些数据结构的一部分。为了提供这种额外的能力，每个节点中链接所用的空间要加倍，同时每个基本操作中的链接控制指令的数量也要加倍。所以除非有特别的需要，一般不使用双向链表。我们把详细实现方法的讨论推迟到几个的确有这种需要的特定场合中——例如 9.5 节的例子。

贯穿全书我们都使用链表，首先是用于基本的 ADT 实现（参阅第四章），然后在更复杂的数据结构中充当组件。链表是很多程序员初次接触的、直接处于他们控制之下的一种抽象数据结构。我们将会看到，在开发应用于许多重要问题的高层抽象数据结构的过程中，链

表充当了一种核心工具。

**练习**

▷ 3.33 编写一个函数，移动一个给定链表中最大的元素，使它成为链表中最后的节点。

3.34 编写一个函数，移动一个给定链表中最小的元素，使它成为链表中第一个节点。

3.35 编写一个函数，重新排列一个链表。使偶数位置的节点集中到链表的后半部，奇数位置的节点集中到链表的前半部，同时分别使奇数位节点和偶数位节点之间的相对顺序保持不变。

3.36 对一链表实现一段代码，互换两个给定链接 t 和 u 所指向的节点之后的节点位置。

○ 3.37 编写一个函数，接收一个作为参数的链表指针，并返回一个指向该链表的副本（一个含有相同元素、相同排列顺序的新链表）的指针。

3.38 编写一个函数，接收两个参数——一个链表指针和一个函数，该函数接收另一个链接作为参数。若作为参数的函数对链表上的一个节点返回一个非零值，则删除该节点，对所有节点都做这个操作。

3.39 重做练习 3.38，但要为每个通过测试的节点都建立副本，并返回一个指向包含这些节点的链表的指针。要求节点排列顺序必须和原表一致。

3.40 实现程序 3.10 的一个使用头节点的版本。

3.41 实现程序 3.11 的一个不使用头节点的版本。

3.42 实现程序 3.9 的一个使用头节点的版本。

3.43 实现一个程序，互换一个双向链表中两个给定节点的位置。

○ 3.44 对于以下一个链表，在表 3.1 中给出表项：该链表永远非空，被一个指向其首节点的指针所引用，而且尾节点有一个指向本身的指针。

3.45 对于以下一个循环链表，在表 3.1 中给出表项：该链表有一个同时作为链表开始和结尾的哑元节点。

## 3.5 链表的内存分配

链表和数组相比较，其中一个优点是链表在它们的生存期内可以便捷地增大和缩小。尤其是不必预先知道它们使用的最大空间。这个特性导致的一个重要的实用结果是：我们可以使几个数据结构共享同一块内存空间，而在任何时刻都无须特别注意它们的相对大小。

问题的关键在于怎样实现操作符 new。举个例子：从一个链表删除一个节点时，重新安排链接、使得节点不再挂接在链表中是我们的工作。但系统如何处理该节点占用过的空间呢？系统又如何循环利用空间，使得当调用 new 请求更多空间时，它总可以满足一个节点的空间要求呢？隐藏在这些问题背后的机制提供了另一个使用基本链表处理的例子。

和 new 相对应的是操作符 delete。当我们使用完一块已分配的内存后，就调用 delete 来通知系统可以回收这块内存并留作他用。动态内存分配（dynamic memory allocation）是

管理内存的过程，负责响应来自客户程序的 new 和 delete 调用。

当我们在程序 3.9 或 3.11 这样的程序中直接调用 new 时，所有调用都请求相同大小的内存块。这是典型的情况，这马上令我们想到一种可供选择的记录可供分配内存的办法——就是使用链表！可以通过一个单向链表，把所有不属于任何正被使用的链表的节点联结起来。我们也将这个表称为空闲列表（free list）。当需要为一个节点分配空间时，就从空闲表删除它而得到内存空间；从任何链表中删除一个节点时，我们的处理办法是把它插入到空闲表中。

程序 3.14 是程序 3.12 中定义的接口的一个具体实现，它包含了内存分配函数。和程序 3.13 一起编译以后，所产生的结果和我们在程序 3.9 使用的直接实现的结果是一样的。给出向一个链表插入节点和在其中删除节点的基本操作以后，维护用于固定大小节点的空闲表的任务就显得微不足道了。

对程序 3.13，图 3.11 显示了空闲表怎样随着节点的删除而增长。为简单起见，该图采用了一个基于数组索引的链表实现（没有头节点）。

**程序 3.14 链表处理接口的实现**

本程序给出了程序 3.12 中声明函数的实现，并展示了一种为一定大小节点分配内存的标准途径。我们建立一个空闲表，把它初始化为我们的程序将使用的节点数目上限，其中全部节点都链接起来。那么，当一个客户程序分配一个节点时，就把该节点从空闲列表删除；当一个客户程序释放一个节点时，就把该节点链接到空闲列表中。

依照常规，客户程序除了通过声明类型为 Node 的变量、把它们作为接口里定义的函数参数使用外，不能引用链表节点；而返回给客户程序的节点有自身链接。这些常规对引用未定义指针提供了一种保护手段，也在一定程度上保证了客户程序按照我们的意图使用接口。在 C++中，我们通过带有构造函数的类，强制执行这些常规（参阅第四章）。

```
#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; }
```

在 C++环境中实现一个通用的内存分配符，要比我们的简单例子所涉及的复杂许多。而标准库中 new 的实现也决不是程序 3.14 所显示的那样简单。两者之间一个主要的不同点是：new 必须处理不同大小（范围从极小到极大）的节点发出的存储分配请求。为了这个目的，人们已经研究出几种聪明算法。某些现代系统采用的另一种方法是：把用户从直接 delete 节点的要求中解放出来——使用"垃圾收集"（garbage-collection）算法自动删除所有不被任何链接使用的节点。沿着这条思路，人们已经开发出几个聪明的存储管理算法。我们不会对它们进行更详细的讨论，因为它们的性能依赖于特定系统和机器的属性。

能够利用关于一个应用的专用知识的程序，常常比用于同一任务的通用程序更高效。内存分配也遵守这条定律。一个必须处理不同大小的存储请求的算法不可能知道我们将一直请求某种固定大小的内存块，因此也不能利用这一点。矛盾的是，避免使用通用库函数的另一个原因是这样做会使程序具有更高的可移植性——当改动这些库，或者迁移到一个不同的系统中时，我们可以防止出现意想之外的性能改变。很多程序员已经发现：为了开发出高效、可移植的使用链表的程序，用一个简单的、类似于程序 3.14 所示的内存分配符是一种很有效的途径。这个方法也适合于很多在本书中将要讨论的算法，这些算法都对内存管理系统做出了相似的要求。经过上述说明，我们将在贯穿本书的其余部分中使用标准的 C++内存分配管理功能 new 和 delete。

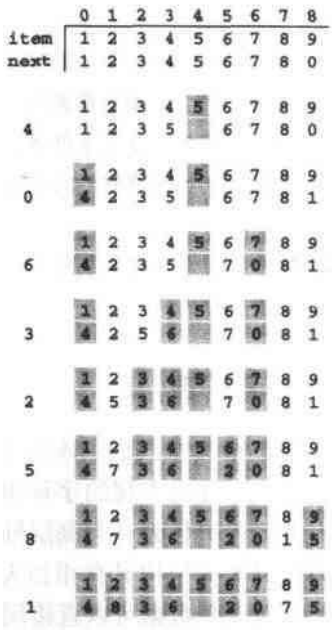



**图 3.11　一个链表的数组表示（结合空闲列表）**

图 3.6 的这个版本显示了维护一个由从循环链表中删除的节点组成的空闲列表的结果，其中空闲列表的首节点的索引在最左边一列给出。该过程结束时，空闲列表是一个包含所有被删除元素的链表。从 1 开始顺着链接向前遍历，我们可看到元素的顺序是 2 9 6 3 4 7 1 5，与它们被删除的顺序正好相反。

### 练习

○ 3.46 编写一个程序，释放（以一个指向它们的指针调用 delete）一个给定链表的所有节点。

3.47 编写一个程序，释放一个链表中可以被 5 整除（第 5 个、第 10 个、第 15 个……）的位置上的节点。

○ 3.48 编写一个程序，释放一个链表中偶数（第 2 个、第 4 个、第 6 个……）位置上的节点。

3.49 实现程序 3.12 中的接口，在 newNode 和 deleteNode 中分别使用 new 和 delete。

3.50 对程序 3.13 用 $M = 2$ 以及 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$ 进行试验研究，用 new 和 delete（参阅练习 3.49）比较程序 3.14 中内存分配函数的运行时间。

3.51 实现程序 3.12 中的接口，使用数组索引（不设头节点）而不是指针，使图 3.11 成为你的程序的操作的跟踪记录。

○ 3.52 假设你有一组不带有空指针的节点，也即每个节点指向本身或者集合中其他节点。证明如果从任意给定节点出发，沿链接遍历，最终必定得到一个回路。

● 3.53 在练习 3.52 的条件下，编写一个代码段。它根据给定的指向一个节点的指针，找出从该节点出发、沿链接遍历而最终到达的不同节点的数目，过程中不修改任何节点。不要使用超过一个固定数量的额外内存空间。

●● 3.54 在练习 3.53 的条件下，编写一个函数。它确定如果跟随两个给定的链接向前遍历，最终是否在同一回路结束。

## 3.6 字符串

在 C 语言中我们用术语字符串（string）代表一个长度可变的字符数组，它是由一个起始点和一个标志末尾的字符串终结符定义的。C++从 C 继承了这种数据结构，还在标准库中引入字符串作为一个高层抽象。这一节中我们要考虑一些 C 类型字符串的例子。字符串作为低层数据结构具有相当大的价值，原因有二。第一，很多计算应用涉及对文本数据的处理，而这些数据可以直接用字符串来表示；第二，很多计算机系统支持对以字节（byte）为单位的内存进行直接、高效的访问，而字节和字符串中的字符直接对应。也就是说，在很多情况下字符串抽象都和应用程序对机器性能的需要相吻合。

对于一个以字符串终结符结束的字符序列，其抽象表示可以经由很多途径实现。比方说我们可以用链表，尽管这个选择需要为每个字符花费一个指针。C++从 C 继承了本节中讨论的基于数组的具体实现，而且在标准库中提供了更为通用的实现方法。在第四章中我们还将讨论其他实现方法。

字符串和字符数组的区别在于长度（length）。两者都代表连续区域的内存，但数组的长度在数组建立时就已确定下来，而字符串的长度在一个程序的执行过程中还可以改变。这个区别具有值得注意的含义，我们马上就会探讨这一点。

我们需要为字符串保留内存，要么通过在编译期间声明一个固定长度的字符数组，要么

通过在执行期间调用 new[ ]。一旦分配了数组，我们就可以填入字符——从数组头开始，以字符串终结符结束。如果没有字符串终结符，那么一个字符串完全等价于一个字符数组；有了字符串终结符，我们就可以在更高的抽象层次上处理，认为只有数组中从开始到字符串终结符的部分才包含了有意义的信息。字符串终结符具有 0 值，也就是大家所知的'/0'。

例如要找出字符串的长度，我们对开始和字符串终结符之间的字符数量计数。表 3.2 给出了我们经常对字符串执行的简单操作。它们的处理过程都涉及到从头到尾扫描字符串。其中很多都在<string.h>中被声明为库函数，可直接使用；然而许多程序员在简单应用程序的嵌入代码中都会使用经过轻微修改的版本。实现同样操作的健壮的程序应该有额外的代码检查错误情况。我们在这里引入这些代码不单要突出它的简单，还要直接显露其执行特性。

我们对字符串执行的一个最重要的操作是比较（compare），它可以告诉我们两个字符串中的哪一个将在字典中首先出现。为了方便讨论，我们假想出一本理想化的字典（因为现实中字符串的规则包含标点、大小写字母、数字等等，相当复杂），从头到尾逐个字符进行比较。这个排序称为字典排序（lexicographic order）。我们也可以用比较函数确定字符串

**表 3.2 基本的字符串处理操作**

这个表使用两种不同的 C++语言基础元素，给出了基本字符串处理操作的实现。指针方法使代码更紧凑，但索引数组方法是表达算法的一种更自然的途径，也使代码更易于理解。附加操作的指针版本和索引数组版本是相同的；而前缀比较的指针版本也来源于一般比较操作，就和索引数组版本的方法相同，所以这两者都被省略了。全部实现所需时间都和字符串长度成正比。

**索引数组版本**

```
计算字符串长度 (strlen(a))
   for (i = 0; a[i] != 0; i++); return i;
复制 (strcpy(a, b))
   for (i = 0; (a[i] = b[i]) != 0; i++);
比较 (strcmp(a, b))
   for (i = 0; a[i] == b[i]; i++)
      if (a[i] == 0) return 0;
   return a[i] - b[i];
比较（前缀）(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;
追加 (strcat(a, b))
   strcpy(a+strlen(a), b)
```

**等价的指针版本**

```
计算字符串长度 (strlen(a))
   b = a; while (*b++); return b-a-1;
复制 (strcpy(a, b))
   while (*a++ = *b++);
```

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

是否相等。依照常规，如果在字典中第一个参数字符串比第二个先出现，那么比较函数返回一个负值；如果两者相等则返回0；如果按字典排序第二个参数字符串比第一个先出现，则返回一个正值。特别值得注意的是：对字符串做等同性测试和确定两个字符串指针的等同性并不一样。如果两个字符串指针相等，那么它们指向的字符串也相等，实际上是同一个字符串；但我们也可以使用不同的指针指向相同的字符串（相同的字符序列）。很多应用程序都把信息保存为字符串形式，然后通过比较字符串处理或访问那些信息，所以比较操作是一个特别重要的操作。在3.7节和本书中的很多其他地方，我们将看到一些特定的例子。

程序3.15是一个简单的字符串处理任务的实现过程：若一个短模式字符串在一个长文本字符串中出现，它就打印出现的位置。对于这个任务已经开发出几种完善的算法，但这个简单例子显示了用C++处理字符串时我们使用的几种常规做法。

字符串处理为通晓库函数作用的必要性提供了一个极具说服力的范例。问题是库函数的运行时间也许大于我们凭直觉所得的期待值。例如，确定一个字符串的长度所需时间和字符串的长度成正比。忽略这个事实可能引致严重的程序性能问题。例如，快速地浏览一下标准库以后，我们也许把程序3.15中的模式匹配实现为以下形式：

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

不幸的是，无论循环体中使用什么代码，这个代码段所需的时间都至少和字符串a长度的平方成正比。因为每一次执行循环，程序都把a从头到尾遍历一次以确定其长度。这个开销相当可观，甚至令人望而却步：如果执行这个程序来检查本书（大概有超过1百万个字符）中是否含有某一单词，将需要数万亿条指令。诸如上面这个例子的问题难以检测，因为当我们用比较小的字符串调试时，这个程序也许运作得十分良好；但投入实际使用后它将会逐渐变慢，甚至永远不能完成。而且，我们只有知道这些问题的存在，才能着手消除它们。

**程序3.15 字符串搜索**

这个程序从命令行接受一个单词，在一个（假设非常大）的文本字符串里找出所有出现这个单词的地方。我们将字符串声明为一个固定大小的字符数组，并通过cin.get()从标准输入设备读取（我们也可以像程序3.6那样用new[]）。来自命令行参数的单词所使用的内存由系统分配，分配过程发生在程序被调用之前，我们从argv[1]得到字符串指针。对于字符串a的每个比较起始位置i，我们都尝试逐个字符匹配该子串（单词），检验是否相等；子串起始位置为p。只要成功到达p的结尾，我们就打印出i，也就是单词在文本中出现的

起始位置。

```
#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;
  }
```

这种错误称为性能 bug，原因是其代码被校验后是正确的。但程序并不像我们（暗地里）预期的那样有效运作。在我们正式开始研究高效算法以前，必须确保已经消除了这类性能 bug。虽然标准库有很多优点，但我们必须对把它们用于这种简单函数的危险性提高警觉。

在本书我们反复提到的核心概念中，其中之一是同一抽象概念的不同实现方法可以导致差异极大的性能特性。譬如，C++标准库里的类 string 可追踪记录字符串的长度，因此可以在常数时间内返回一个字符串的长度；但其他操作的速度就慢得多。一种实现方法也许适合一种应用；另一种实现方法也许适合另一种应用。

库函数时常不能保证使所有应用程序的性能都达到最佳。即便一个库函数（例如 strlen）的执行性能记录良好，我们也不能确保某些未来的实现方法所带来的性能改变不会对我们的程序造成负面影响。这个结论在算法和数据结构的设计中至关重要，因此我们必须始终把它铭记于心。我们将在第四章讨论其他例子和更进一步的结果。

字符串其实是指向字符的指针。在一些场合中，这种实现可以使字符串处理函数的代码更紧凑、简洁。例如把一个字符串复制到另一个，我们可以用：

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

代替：

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

也可以使用表 3.2 中给出的第三种选择。这两种引用字符串的方法是等同的，但也许会产生在不同的机器上具有不同执行特性的代码。一般来说，数组版本清晰明了，而指针版本

可减少表达式数量。对于特定应用程序频繁执行的特定代码，哪一个版本最好呢？这里不做详细研究，因为必须根据实际需要区别对待。

字符串的内存分配比链表复杂得多，因为不同字符串的大小也不同。的确，一个为字符串保留空间的完全通用的机制恰恰是系统提供的 new[]和 delete[]函数。正如 3.6 节所提到的那样，各种各样解决这个问题的算法已经被研发出来了，它们的执行特性和系统及机器有关。内存分配问题在开始时可能显得相当严重，但当我们实际处理字符串时常常发现：情况要比我们预料的好得多。这是因为我们处理的是字符串的指针，而不是字符本身。当然，我们通常不假设所有字符串都位于个别的内存分配块中。我们倾向于假定每个字符串位于不定的内存空间中，每块空间仅能容纳字符串和它的终结符。当执行建立或延长字符串的操作时我们必须十分小心，保证有足够的分配空间。作为一个例子，我们将在 3.7 节中讨论一个读入字符串并控制它们的程序。

### 练习

▷ 3.55 编写一个程序，接受一个字符串作为参数并打印一张表。对于每一个在字符串中出现的字符，该表给出该字符以及出现频率。

▷ 3.56 编写一个程序，检查一给定的字符串在忽略空格后是否为回文（顺读和倒读都一样）。例如对于字符串 if i had a hifi，你的程序应该报告检查结果成功。

3.57 假设字符串的内存空间是个别分配的。编写函数 strcpy 和 strcat 的新版本，要求分配内存并返回一个指向新字符串的指针作为结果。

3.58 编写一个程序，从标准输入设备接受一个字符串作为参数，并读入一组单词的序列（字符序列之间由空格分隔）。打印出所有以子串形式在参数字符串中出现过的单词。

3.59 编写一个程序，在一个给定字符串中用单个空格代替由一个以上空格组成的子串。

3.60 实现程序 3.15 的指针版本。

○ 3.61 编写一个高效率的程序，确定一个给定字符串中最长的空格序列的长度，要求在字符串中检查的字符尽可能少。提示：随着空格序列长度的增加，你的程序的速度也应该加快。

## 3.7 复合数据结构

数组、链表和字符串都为顺序组织数据提供了简单的途径。它们作为第一层抽象，可使我们以有利于高效地处理对象的方式组织对象。我们已经介绍了这些抽象，下一步可以在一个层次模型中用它们构建更复杂的结构。我们可以构造多维数组、链表数组和字符串数组等等。这一节将考虑这些结构的例子。

正如一维数组对应于矢量一样，带有两个索引的二维（two-dimensional）数组对应于矩阵（matrix），并因此在数学计算中得到广泛应用。例如，我们可以用以下代码进行矩阵 a 和 b 的乘法运算，把结果保留在第三个矩阵 c 中：

```
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];
```

我们经常碰到这样的数学计算，它们可以用多维数组的形式自然地表达出来。

除了数学应用外，另一种结构化信息的常见途径是使用一张表，表中的数据以行列形式组织。一张关于学生在各门科目中所得成绩的表也许具有以下结构：每一行对应一个学生，每一列对应一门科目。这张表可以用一个二维数组表示，对行和列分别使用一个索引。如果有学生 100 人、科目 10 门，我们可以编写 grades[100][10]声明这个数组，那么 grades[i][j]就表示第 i 个学生在第 j 门科目取得的成绩。要计算其中一门的平均分，我们先求出该列所有元素的总和，再除以行的总数；要计算某个学生全部科目的平均分，我们先求出该行所有元素的总和，再除以列的总数，以此类推。二维数组在这类应用中的使用非常广泛。在一台计算机上，二维以上数组常常更直接，也更方便。例如，一条指令可能用第三个索引记录学生成绩表的年份。

二维数组是一种表示方法上的便利，因为数字最终是存储在计算机内存中，而内存实际是一个一维数组。在很多编程环境中，二维数组以行序为主序（row-major order）的方式存储为一个一维数组：在数组 a[M][N]里，开始的 $N$ 个位置由第一行（由元素 a[0][0]到元素 a[0][N-1]）占据，接下来的 $N$ 个位置由第二行（由元素 a[1][0]到元素 a[1][N-1]）占据，以此类推。以行序为主序时，前面段落中的矩阵相乘代码的最后一行恰好等价于以下代码：

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

**程序 3.16 二维数组的分配**

本函数把二维数组视作出数组构成的数组，为其动态分配内存。我们先分配一个指针数组，然后为每一行分配内存。通过这个函数，语句

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

将分配一个 M×N 的整数数组。

```
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;
  }
```

同样的方案可以推广开来，为多维数组提供便利。在 C++中多维数组可以用一种更具

普遍性的方式实现：定义为复合数据结构（复合数组）。这样提高了实现的灵活性，譬如可有一个由大小各异的数组构成的数组。

我们在程序 3.6 中看到过一个动态分配数组空间的方法，它允许我们把程序用于不同大小的问题而无需重新编译；因此我们希望对于多维数组也有一个类似的方法。但如果在编译期间不知道多维数组的大小，又怎样为它们分配内存呢？也就是说，我们希望在一个程序中引用一个诸如 a[i][j]的数组元素，但不能把它声明为 int a[M][N]（举例）——因为我们不知道 M 和 N 的值。以行序为主序时，像下面的语句：

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

将分配一个 M×N 的整数数组，但这个办法并不是在所有情况下都有效的。例如当一个数组被传递到一个函数时，编译期间只有它的第一维可暂时不被确定。程序 3.16 根据二维数组作为“由数组构成的数组”的定义，提供了一种更有效的解决方案。

程序 3.17 显示了一种类似的复合数据结构——字符串数组的使用。乍一看，因为我们对字符串的抽象定义是字符组成的数组，我们也许可把字符串数组表示为多维数组。但是我们用于字符串的具体表示是一个指向字符数组开始的指针。如图 3.12 所示，我们仅仅通过重新排列数组中的指针，就可以收到重新排列字符串的效果。程序 3.17 使用了库函数 qsort——实现这些函数是第六章到第九章大致涉及的主题，而第七章将特别侧重这方面的内容。这个范例展示了一个处理字符串的典型情况：我们把字符本身读入一个巨大的一维数组，保存指向单个字符串的指针（用字符串终结符为它们划分界限），然后控制这些指针。

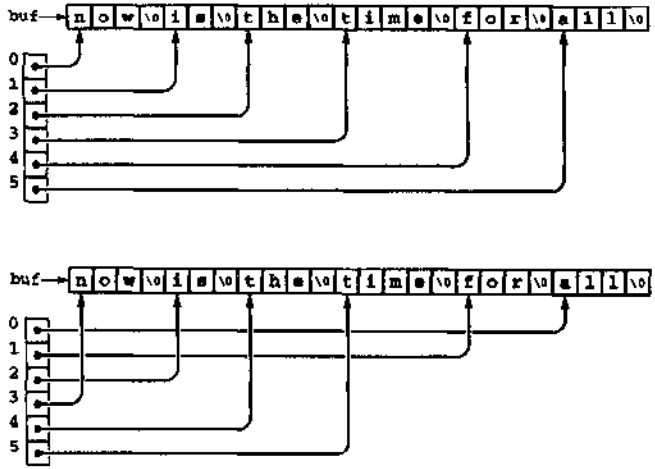



**图 3.12 字符串排序**

处理字符串时，我们通常使用指向一个含有那些字符串的缓冲区（上图）的指针。因为和长度不一的字符串自身相比，指针更容易控制。例如一次排序的结果是：重新排列指针，使得按顺序访问它们时可以给出依照字典排序排列的字符串。

我们已经遇到过字符串数组的另外一种应用了：argv 数组，在 C++程序里它被用于向

主函数 main 传递参数字符串。系统在一个字符串缓冲区内存储用户键入的命令行，并向主函数 main 传递一个指针，这个指针指向一个由缓冲区内字符串的指针构成的数组。我们用转换函数计算对应于某些参数的数字，至于其他参数则作为字符串直接使用。

**程序 3.17 对字符串数组进行排序**

这个程序展示了一个主要的字符串处理函数：重新排列一组字符串使其有序。我们把所有字符串读入一个足够容纳它们的缓冲区，在一个数组中为每个字符串建立一个指针，然后重新排列指针，使指向最小字符串的指针处于数组中第一位，指向次小字符串的指针处于数组中第二位，以此类推。

实际执行排序的库函数 qsort 需要四个参数：指向数组开始的指针、对象的数量、每个对象的大小以及一个比较函数。它通过盲目地重排代表对象（本例中是字符串指针）的数据块，以及使用一个接收 void 指针作为参数的比较函数，从而让程序与被排序的对象类型无关。比较函数代码返回 strcmp 的结果，strcmp 的参数类型是指向 char 指针的指针。要在一次比较过程中实际访问一个字符串的第一个字母，我们需要三个指针：其一取得指向数组的索引（也是一个指针），其二取得指向字符串的指针（使用索引），其三取得该字符（使用指针）。

我们采用了一种不同的方法，使我们的排序和搜索函数具有类型无关性（参阅第四章和第六章）。

```
#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;
  }
```

我们也可以单独使用链接构造复合数据结构。图 3.13 显示了一个多重链表（multilist）的例子：节点有多个链接域并属于相对独立的不同链表。在算法设计中常常用一个以上链接建立复杂的数据结构，但它们的建立方式足以让我们有效地处理这些数据。例如，一个

双向链表就是多重链表，它满足以下的约束条件：x->l->r 和 x->r->l 都等于 x。在第五章中我们将分析一个重要得多的数据结构，其中每个节点有两个链接。

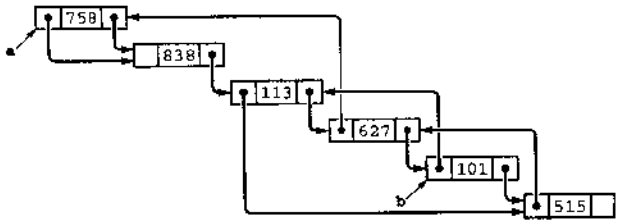



**图 3.13 多重链表**

我们可以把带有两个链接域的节点在两个独立链表中联系起来，其中每个链表使用一个属于自己的链接域。这里右链接域按一个次序（例如可以是节点被创建的次序）组织节点，而左链接域按另一不同次序（例如本例中是排列次序，假设插入排序只使用左链接域）组织节点。从 a 顺着右链接域出发，我们就以创建顺序访问节点；从 b 顺着左链接域出发，我们就以排列顺序访问节点。

如果一个多维矩阵稀疏（sparse，相对较少元素非零）的话，那么我们可以利用一个多重链表代替多维数组来表示它。对于矩阵里的每个值可用一个节点代表，每一维则用一个链接代表，该链接指向那一维中的后续元素。这种安排大大减少了所需的存储空间：从原先的各维中最大索引的乘积减为和非零元素的数量成正比；但缺点是增加了许多算法的运算时间，因为它们必须为了访问单个元素而遍历大量链接。

为了引入更多复合数据结构的例子，并突出索引结构和链接结构之间的差别，我们接下来讨论用于代表图的数据结构。图（graph）是一种基本的组合对象，被定义为由多个称为顶点（vertex）的对象的集合和顶点之间称为边（edge）的连接的集合构成。我们在第一章的连通性问题中就已接触过图了。

**程序 3.18 图的邻接矩阵表示**

这个程序读入一组定义一个无向图的边，创建一个对应这个图的邻接矩阵。如果在图中顶点 i、j 或 j、i 之间有一条边，就把 a[i][j]和 a[j][i]设置为 1；如果不存在这样的边，则设置为 0。这个程序假设顶点数 V 是一个编译时常数，否则需要动态分配表示邻接矩阵的数组（参阅练习 3.71）。

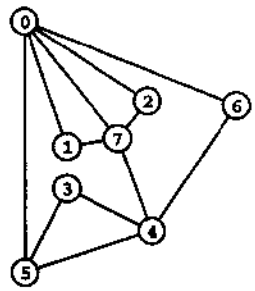
```
#include <iostream.h>
int main()
  { int i, j, adj[V][V];
    for (i = 0; i < V; i++)
      for (j = 0; j < V; j++)
        adj[i][j] = 0;
    for (i = 0; i < V; i++) adj[i][i] = 1;
    while (cin >> i >> j)
      { adj[i][j] = 1; adj[j][i] = 1; }
  }
```

我们假定有一个含 V 个顶点和 E 条边的图，那么它由一个 E 对 0~V-1 之间的整数组成的集合定义。那就是说，我们假定这些顶点由整数 0，1，…，V-1 分别标号，这些边分别由一对顶点指定。正如第一章的情况，我们认为对 *i-j* 定义了顶点 *i* 和 *j* 之间的连接，因此 *i-j* 和 *j-i* 具有相同的意义。由这样的边组成的图被称为无向图（undirected graph）。在第七部分我们将介绍其他类型的图。

一种表示图的直观方法是使用二维数组，也称为邻接矩阵（adjacency matrix）。通过邻接矩阵我们可马上判断出顶点 *i* 和 *j* 之间是否存在一条边，只需要检查矩阵中行 *i* 和列 *j* 相交处是否为非零值。对于我们正在考虑的无向图，如果矩阵中行 *i* 和列 *j* 相交处存在一个项，那么行 *j* 和列 *i* 相交处也存在一个项，所以矩阵是对称的。图 3.14 显示了一个无向图的邻接矩阵的范例；程序 3.18 显示了给定一组边的序列作为输入时，我们如何创建相应的邻接矩阵。

另一种表示图的直观方法是使用链表数组，也称为邻接表（adjacency list）。我们为每个顶点保存一个链表，其中每个节点代表一个连接到该顶点的顶点。对于我们正在考虑的无向图，如果在 *i* 的链表中存在节点 *j*，则在 *j* 的链表中必定存在节点 *i*。图 3.15 显示了一个无向图的邻接表的范例；程序 3.19 显示了给定一组边的序列作为输入时，如何创建相应的邻接表。

这两种图的表示方法都是简单数据结构的数组——都对每个顶点描述了和该顶点关联的边。对邻接矩阵，这个简单数据结构实现为一个索引数组；对邻接列表，则实现为一个链表。




| | 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 |

**图 3.14　图的邻接矩阵表示**

图由一组顶点和一组连接顶点的边组成。为简单起见，我们为顶点指定索引（从 0 开始的连续非负整数）。一个邻接矩阵是一个代表图的二维数组：当且仅当顶点 *i*、*j* 之间存在边，我们把行 *i* 和列 *j* 相交处的项设置为 1。因此数组关于对角线对称。依照惯例，我们把对角线上的项都设为 1（每个顶点都和自身连接）。例如第六行（和第六列）表明顶点 6 和顶点 0、4、6 连接。

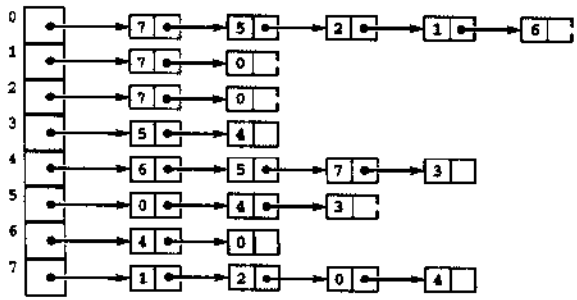



**图 3.15　图的邻接表表示**

对于图 3.14 中的图，这里的表示方法使用了一个链表数组。所需的空间正比于节点数和边数的总和。要找出连接到给定顶点 *i* 的顶点索引，我们着眼于数组的第 *i* 个位置，它包含了一个链表指针，链表中为每个和 *i* 连接的顶点保存一个节点。

因此当我们表示一个图时，我们就直接面对空间的权衡问题。邻接矩阵使用的空间正比于 $V^2$，而邻接表使用的空间正比于 $V+E$。如果边数较少（这样的图称为稀疏图，sparse graph），那么邻接表表示方法将节省大量的所需空间；如果大多数顶点对都由边相连（这样的图称为稠密图，dense graph），那么邻接矩阵表示方法更可取，因为它不涉及到链接。某些算法使用邻接矩阵表示将更有效，因为它允许在常数时间内解决“顶点 $i$、$j$ 之间是否有边相连”这个问题；其他算法则使用邻接表表示将更有效，因为它允许我们在正比于 $V+E$、而不是 $V^2$ 的时间内处理一个图的所有边。在 5.8 节中我们将看到一个关于此类权衡问题的例子。

**程序 3.19 图的邻接表表示**

这个程序读入一组定义一个图的边，创建一个对应该图的邻接表。图的邻接表是一个链表数组，每个元素表示一个顶点，其中第 $j$ 个表包含一个由所有连接到第 $j$ 个顶点的节点构成的链表。

```
#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]);
      }
  }
```

图的邻接矩阵表示和邻接表表示都可以被直接扩展，以处理其他类型的图（例如练习 3.70）。它们是我们在第七部分要考虑的大多数图处理算法的基础。

为了总结这一章，让我们来考虑一个例子，它展示的是：使用复合数据结构，为我们在 3.2 节中讨论的简单几何问题提供一个高效的解决方案。这个问题是：$d$ 已知，希望求出在单位正方形内的 $N$ 点组成的集合中，有多少对点可被长度小于 $d$ 的直线相连。程序 3.20 使用了一个二维链表数组来改善程序 3.8 的运行时间：当 $N$ 足够大时，系数大约为 $1/d^2$。它把单位正方形划分成相同大小的小正方形组成的网格。然后，对于每个正方形建立一个链表，其中包含所有落入该正方形的点。二维数组提供了直接访问接近一给定点的点集合的能力，链表则提供了把点存储于其可能落入的正方形的灵活性，我们完全不必预先知道

有多少个点落入每个网格正方形。

程序 3.20 使用的空间正比于 $1/d^2 + N$，但运行时间是 $O(d^2 N^2)$；当 $d$ 较小时，这个运行时间相对于程序 3.8 的直接算法是一个显著的改善。例如 $N = 10^6$、$d = 0.001$ 时，我们可以在相当于线性的时间和空间内解决问题；而程序 3.8 的直接算法则需要花费一段不可接受的时间。我们也可以把这个数据结构用作解决很多其他几何问题的基础。譬如在确定“平面内 $N$ 个随机点的集合是否可以用长度为 $d$ 的直线连接起来”（这是一个在网络和电路设计中引起关注的基本问题）的答案时，只要把它和第一章的合并-查找（union-find）算法相结合，便可给出一个接近线性的算法。

基本抽象结构可用于把不同类型的数据组织成对象，并间接或直接地把这些对象组织成复合对象。正如这一节中我们看过的范例所显示的那样，我们从基本抽象结构建立的复合体的复杂程度并没有止境。我们将在第五章看到：这些范例仍没有达到使结构化数据的过程完全一般化的目标，我们还有一步之遥。然而在迈出这一步之前，我们要考虑可用链表和数组构建的主要抽象数据结构；这些基本工具将有助于我们进行下一层的归纳。

**程序 3.20　一个二维链表数组**

本程序阐明了适当的数据结构对于程序 3.8 的几何计算的有效性。它把单位正方形划分成网格，并建立一个二维链表数组，使每个网格正方形对应于一个列表。我们把网格划分得足够精细，以使距离任一给定点小于 $d$ 的所有点要么落在同一个网格，要么落在相邻网格。函数 malloc2d 和程序 3.16 中的类似，但用于类型为 link 的对象，而不是类型 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;
 }
```

## 练习

3.62 为程序 3.16 编写另一个版本，处理三维数组。

3.63 修改程序 3.17，单独处理输入字符串（从输入设备读入每个字符串后就为它分配内存）。你可以假设所有字符串都有少于 100 个字符。

3.64 编写一个程序，对一个二维数组填入 0 或 1 值。如果 $i$ 和 $j$ 的最大公约数是 1，就把 a[ $i$ ][ $j$ ]设为 1，否则设为 0。

3.65 结合使用程序 3.20 和程序 1.4，开发一个高效的程序，判断 $N$ 个点的集合是否可以被长度小于 $d$ 的边连接起来。

3.66 编写一个程序，转换一个稀疏矩阵的表示方法：从二维数组变为只含有对应非零值的节点的多重链表。

● 3.67 对于用多重链表表示的矩阵，实现矩阵乘法。

▷ 3.68 给出程序 3.18 建立的邻接矩阵，假定输入点对为 0-2、1-4、2-5、3-6、0-4、6-0 和 1-3。

▷ 3.69 给出程序 3.19 建立的邻接列表，假定输入点对为 0-2、1-4、2-5、3-6、0-4、6-0 和 1-3。

○ 3.70 有向图（directed graph）指顶点连接有方向的图：边从一个顶点出发*到达*另一个顶点。根据以下假设完成练习 3.68 和练习。

3.69 输入点对代表一有向图，*i-j* 表示存在一条从 $i$ 到 $j$ 的边。同时以箭头表明边的方向，画出相应的图。

3.71 修改程序 3.18，以命令行参数形式接收顶点个数，然后动态分配邻接矩阵。

3.72 修改程序 3.19，以命令行参数形式接收顶点个数，然后动态分配链表数组。

○ 3.73 编写一个函数，利用一个图的邻接矩阵，对给定顶点 $a$ 和 $b$ 计算满足以下条件的顶点 $c$ 的数量：存在一条从 $a$ 到 $c$ 的边和一条从 $c$ 到 $b$ 的边。

○ 3.74 使用邻接表解答练习 3.73。

# 第四章

# 抽象数据类型

在我们用计算机解决问题的过程中，随着数据和处理数据的程序的发展，抽象模型也随之发展起来。我们可以在日常的编程中看到体现这一原则的较低层次的例子（例如在使用第三章讨论的队列和链表时），也可以在问题的解决中看到体现这一原则的较高层次的例子（正如我们在第十章中所看到的，我们使用合并-查找森林来解决连通性问题）。在这一章里，我们考虑抽象数据类型（abstract data types，ADT），它允许我们在编程中使用高层次的抽象。通过使用抽象数据类型，我们可以将编程时对数据的概念转换从特定的数据结构表示和算法实现中分离出来。（也就是说，数据的具体解释可以独立于数据结构和算法而存在。——译者注。）

所有的计算机系统都基于抽象层次（layers of abstraction）：首先，我们采用了“位”（bit）的抽象模型，它可以从硅以及其他材料的特定物理属性中归纳出来，也就是得到二进制值0和1；接着，我们采用“机器”（machine）的抽象模型，它来自特定位集合值的动态属性；然后，我们采用了“编程语言”（programming language）的抽象模型，通过控制那些以机器语言为编程语言的机器来认识这一抽象模型；最后，我们采用“算法”（algorithm）这一抽象概念，它是由C++语言编程实现的。抽象数据类型允许我们将这一过程进行得更远，例如开发特定计算任务的抽象机制，它的层次相对C++系统所能提供的来说要高一些；又例如开发适合在各种应用领域解决问题的应用专有抽象机制；或者建立高层次的抽象机制以用于基本的机制。抽象数据类型为我们提供一种可以随时扩展的工具集合，不断用来解决新的问题。

一方面，使用抽象的机制可以让我们从关心程序如何实现这一细节问题上得到解放，另一方面，当性能对于一个程序来说很重要时，我们需要知道基本操作的开销。因此我们使用很多已经内置于机器硬件内部的基本抽象，它们提供了机器指令的基础。至于其他的抽象，我们用软件来实现，或者仍然使用那些以前写好了的系统软件所提供的抽象。通常我们根据基本的抽象机制来建立更高级的抽象机制。所有的抽象层次均遵循同样的基本原则：确定程序中的关键操作和数据中的关键特征，以便在抽象层次中精确地定义它们，然后开发有效的具体机制来支持它们。本章将介绍许多体现这一原则的例子。

为了发展一个新的抽象层次，我们需要定义（define）那些我们想要操纵的对象，以及其上执行的操作；我们需要使用一些数据结构来表达（represent）数据以及实现（implement）操作；而且我们需要保证那些对象能方便地用（use）来解决应用问题（练习的要点）。这些说明同样适用于简单数据类型。为了支持数据类型，我们在第三章讨论了一些基本抽象，它们可以修订成服务于我们的目标，C++对结构机制做了重要的扩展，称为类（class）。类在构造抽象层次的时候尤其有用，因此将会是贯穿本书剩余部分我们用来实现目标的主要工具。

**定义 4.1** *抽象数据类型（ADT）指的是只通过接口进行访问的数据类型（一组值和值上的操作的集合）。我们将那些使用 ADT 的程序叫做客户（client），将那些指定数据类型的程序叫做实现（implementation）。*

数据类型的抽象与其他抽象的关键区别在于“只”这个字：对于 ADT 来说，客户程序除了通过接口定义的操作外，不能访问对象的任何数据值。数据的表示和实现操作的函数都在接口的实现里面，和客户完全分离。接口对我们来说是不透明的：客户不能通过接口看到方法的具体实现。在 C++编程中，我们通常得到一个稍微好一些的区别，因为建立一个接口最简单的方法需要包括接口里的数据实现，但是特别规定客户程序不允许直接访问数据。也就是说，客户程序可以知道数据实现，但是没有办法使用它。

例如，3.1 节程序 3.3 中的 point 数据类型的接口，显式地声明了 point 的结构是浮点数对，它的成员函数分别为 x 和 y 。实际上，数据类型的这种使用方式在很多大型软件系统中非常普遍：我们开发了一组关于数据表示形式的协定（并定义了一组关联操作），而且在接口中实现了这些协定，这样客户程序就可以用它们组成一个大的系统。数据类型保证了系统的所有部分都与核心系统的数据结构的实现保持一致。这种策略虽然很有用，但是有一个缺点：如果我们需要改变数据的表示，我们就需要改变所有的客户程序。程序 3.3 又一次提供了一个简单的例子：开发数据类型的原因之一是使得客户程序能更方便地操作点，而且我们希望客户能够在需要的时候访问单独的坐标。但是如果我们不改变所有的客户程序，就不可能把它变成一个不同的表示（例如极坐标、三维坐标，或者单个坐标的不同数据类型等）。

相反，程序 4.1 展示了一个与程序 3.3 相对应的抽象数据类型的实现，它使用了 C++类来定义数据和与之关联的操作。程序 4.2 是使用这个数据类型的一个客户程序。这些程序实现了与程序 3.3 和程序 3.8 相同的计算。它们举例说明了一些类的基本属性，我们现在就来讨论它们。

当我们在程序里写一个定义（例如 int I）的时候，我们指示系统为 int 类型的数据保留地址空间，我们用名字 i 来引用类型 int。在 C++中，我们使用对象（object）这个名词来表示这一类的实体。当我们在程序中写一个 point 的定义时，就是在说：“我们建立了 point 类的一个对象，它的名字叫做 p”。在这个例子里面，每一个对象都有两个数据项，x 和 y，正如结构体一样，这两个数据项的引用采用类似 p.y 的形式。

**程序 4.1 Point 类实现**

这个类定义的数据类型包括成组的浮点数对（大概翻译成笛卡尔平面的点），所有 Point 的定义都有两个成员函数：构造函数 POINT()用 0~1 之间的随机数来初始化点的坐标值，而 distance(POINT)函数计算到另一个点的距离。数据的表示是私有的，只能被成员函数访问，成员函数是公共的，可以被任何客户使用。我们也许应该保存好这一段代码，给它起个文件名，例如 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); }
  };
```

我们把 x 和 y 这两个项叫作类的数据成员(data member)。一个类也许会定义一些成员函数（member function）来实现与数据类型相关的操作。例如，程序 4.1 的类定义包括了两个成员函数 POINT 和 distance。

就像程序 4.2 一样，客户程序可以通过引用名字来调用那些关联着对象的成员函数，用同样的方法，客户程序也能够访问结构体中的数据。例如，表达式 p.distance(q)计算点 p 到 q 的距离（用另外一个表达式 q.distance(p)也可以得到同样的结果）。程序 4.1 里的第一个函数 POINT()是一类特殊的成员函数，我们称之为构造函数（constructor）：它的函数名与类名相同，当对象建立时它将被调用。客户程序的定义 POINTP 引发了对新对象的空间分配，然后通过 POINT()函数引发点的数据项被赋予 0~1 之间的随机值。

**程序 4.2 Point 类的客户（最近的点的计算）**

程序 3.8 的这个版本是使用程序 4.3 里定义的 POINT 抽象数据类型的客户。new[]操作符构造了一个 POINT 对象的数组（调用构造函数 POINT()来为每一个对象初始化一个随机坐标）。a[i].distance(a[j])调用对象 a[i]的成员函数 distance，其输入参数是 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;
}
```

这种编程风格，有时也叫面向对象的编程（object-oriented programming），已经得到了C++中类结构的直接支持。我们可以把类看作结构体的一种延伸，在类里，我们不仅可以收集数据，还能够定义数据上的操作。也许一个类会派生出很多不同的对象，但它们在很多方面相似：数据成员接纳的值组是一样的；数据成员能执行的操作集是一样的。简而言之，它们是同一个数据类型的不同实例。在面向对象编程中，我们指示对象来操作它们的成员数据（与之相对的是用自由函数来操作存储在对象里面的数据）。

我们只是使用这一个小类作为介绍类的基本特点的例子，因此它并不很完善。在实际编程里面，我们也许会有多得多的关于点的操作。例如，程序 4.1 甚至没有为我们提供如何获得点的 x 和 y 值的操作。正如我们看到的一样，加上像这样那样的操作其实是一件很直接的任务。在第五章中，我们将考虑点和其他几何抽象（例如，线和多边形等）的类的细节。

结构体在 C++中能够有关联函数（在 C 中就没有）。它们的关键区别在于对关键词 private 和 public 的使用，这跟访问信息有关系。私有（private）类成员只能在类里面被引用，而公共（public）成员能够被任何客户引用。私有成员可以是数据和函数：在程序 4.1 里，只有数据是私有的，但是我们将会看到大量私有函数的例子。默认情况下，类成员是私有的，结构体成员是公共的。

举个例子，在一个使用 POINT 类的客户程序中，我们不能像在结构体中那样用 p.x、q.y 来引用它的数据成员，因为 x 和 y 是私有的。我们只能通过使用公共成员函数来处理点。这些函数能够直接访问这个类的任何对象的数据成员。例如，我们在程序 4.1 里调用距离函数 distance 来得到两点之间的距离，格式为 p.distance(q)。在 distance 函数里，有这样一个表达式 dx=x-a.x，其中 x 指的是点 p 的数据成员 x（因为在本例中 distance 是作为实例 p 的成员函数被调用的）；a.x 指的是点 q 的数据成员 x（因为 q 是 distance 函数输入参数中形参 a 的实参）。我们也可以用这样的写法来避免可能发生的迷惑和冲突——dx=this->x-a.x，关键字 this 是一个指针，指向调用该成员函数的当前实例对象（本例中为 p）。

当我们为一个数据成员指定关键字 static 时，表明这个变量只能有一个拷贝（与类直接关联），而不是像普通成员那样有多个拷贝（与每一个对象关联）。我们常常使用这一特性来跟踪对象的统计信息：例如，我们可能会在 POINT 类中包含一个静态（static）整型变量 N，在构造函数中加上 N++，这样我们就可以通过 N 知道被创建的点的数目了。

也许我们会有这样的观点：一个计算两点间距离的函数不应该与任何一个点关联，而

应该像我们以前所做的那样，用两个点作为参数，这样会更自然一些。在 C++里，我们可以在 POINT 类中定义一个不同的 distance 函数来实现这个目标。如下所示：

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

一个静态成员函数不需要被某个特定对象调用就可以访问类中的成员。

另一个可选方案是将 distance 定义成 POINT 类声明外的自由函数（使用与上一段同样的代码，但是要去掉关键字 static）。因为这一版本的距离函数需要访问 POINT 类的私有数据成员，所以我们需要在 POINT 类中包含以下声明：

```
friend float distance(POINT, POINT);
```

友元（friend）函数是指那些不是类成员，却能访问类私有成员的函数。

为了给客户程序提供访问数据值的能力，我们需要提供成员函数来返回它们：

```
float X() const { return x; }
float Y() const { return y; }
```

这些函数被声明为常量（const）函数是因为它们并不修改所访问的对象的数据成员。我们在 C++里常常包含这一类的函数。记住，如果我们采用不同的数据表示（比如极坐标），它们也许会很难实现。但这种困难可以转嫁给客户程序。我们也可以在成员函数中应用这一特性——在程序 4.1 中，如果我们在 distance 函数的实现中用 a.X()来代替 a.x，我们就不需要随着数据表示的改变而修改这一函数的代码。我们甚至可以利用这一特性给自己提供方便——如果不想让客户访问数据，只需把这些函数放在类的私有部分即可。

对于大多数应用，我们建立新类的主要目的是定义一种新的数据类型，特别是根据应用的需要来定义。当我们这样做的时候，常常会希望使用我们自定义的类型能像使用 C++内置类型（例如 int、float 等等）一样方便。我们将在 4.8 节讨论这一主题。帮助我们完成这一愿望的重要工具之一是操作符重载（operator overloading），C++允许我们在基本操作符上指定我们需要的功能，这些功能能够应用到我们自己的类中，而且正好是操作符应该有的功能。举个例子来说，假设我们希望做以下规定：两个点之间的距离如果小于 0.001 的话就把它们看成一个点。我们只要在类里面加上以下代码，就可以使用==操作符来检验两个点是否相同（在前述定义下）。

```
friend int operator==(POINT a, POINT b)
  { return distance(a, b) < .001; }
```

另外一个我们通常希望重载的操作符是 ostream 类里的<<操作符。在 C++编程中，我们通常假设能够使用这一操作符打印任何对象的值；而实际上，我们只能在类中定义了<<操作符时才能这么做。对于我们的 POINT 对象来说，我们可以这么写：

```
ostream& operator<<(ostream& t, POINT p)
  {
    cout << "(" << p.X() << "," << p.Y() << ")";
    return t;
  }
```

这个操作符不是成员函数，甚至不是友元函数：它使用 ADT 公共函数 X()和 Y()来访问数据。

C++里类的概念与客户-接口-实现的范例以及 ADT 之间有什么关系？它提供了直接的语言支持，但是又足够通用，因此我们可以通过很多种方法来实现。我们通常接受这样的惯例：类中公共函数的声明包括了它的接口。也就是说，我们在类的私有部分保持数据的表示，使得使用这个类的程序（客户程序）不能访问它。所有的客户程序都“知道”类的成员函数的公共信息（名字、返回值和参数类型）。为了强调一个类定义接口的自然性，我们先来看看程序 4.3 中的例子。问题的实质是：这样做方便了我们考虑其他的实现，即使是换了不同的数据表示、函数实现，也能够在完全不改写客户程序的情况下验证和比较各种不同的实现。

**程序 4.3 Point ADT 接口**

按照惯例，我们把 ADT 类的接口的实现和它的声明分离开，移除私有部分，用声明（签名）来替换函数实现。程序 4.1 中的接口就是用这种方法分离的。我们不需要修改客户程序的任何代码就可以使用各种不同的实现，只要它们的接口一致。

```
class POINT
  {
    private:
      // 与实现相关的代码
    public:
      POINT();
      float distance(POINT) const;
  };
```

很多应用都要求有修改类的实现的能力。例如，假设我们正在为一个公司开发软件来处理潜在顾客的邮件列表。使用一个 C++类，我们就能够定义函数来允许客户程序在不直接访问数据的情况下操纵数据。实际上，我们提供了成员函数来返回感兴趣的数据。例如，我们可以为客户程序提供一个接口，定义一些像取出顾客姓名、添加顾客记录之类的操作。这种安排最重要的意义是，即使我们需要修改邮件列表的格式，我们也能够使用同一个客户程序。也就是说，我们能够在不改变客户程序的前提下改变数据表示和程序实现。

这一类的类数据类型有时也叫做具体数据类型（concrete data type）。无论如何，满足这一约定的数据类型也符合我们关于 ADT 的定义（定义 4.1）——区别在于一些术语的精确定义而已，像“访问”、“参考”和“指定”等等，这些术语应该留给编程语言理论来解

决。实际上定义 4.1 并没有特别指定接口是什么，或者数据类型和操作应该怎样描述。这种不精确有存在的必要，因为指定这些一般性的信息需要正式的数学语言，甚至会导向更难的数学问题。这一问题是编程语言设计的中心问题，我们将在讨论完 ADT 的例子后讨论它。

我们用 C++类来实现 ADT 时贯彻的是这样的原则：公共函数的定义由接口组成。但这种安排并不好，因为接口和实现不能完全分开。否则当我们修改实现时，就要重新编译客户程序。我们将会在 4.5 节讨论其他的实现方法。

ADT 已经变成现代大型软件系统中的组织原则，它是支持模块编程的有效机制。它提供了一个有效途径来限制大小，减少算法和关联数据结构之间，以及使用算法和数据结构的程序之间的接口的复杂性，这里的算法可能相当复杂，这里的程序可能相当多。这种安排使得我们从整体上理解一个大的应用程序时比较容易。更进一步，ADT 不像那些简单数据类型，它还能提供便利——可以很方便地修改或改进系统的基本数据结构和算法。最重要的是，ADT 定义了用户和实现者之间的一层抽象作为彼此交流的准确方法。

由一个仔细设计的 ADT，我们可以以很多有趣的方式应用客户和实现之间的分离。例如我们通常使用驱动程序来开发和调试 ADT 实现。简单地说，我们常常使用未完成的 ADT 实现，称为占位程序（stub），作为“占位符”来建立系统，或者了解客户程序的特性。

我们在这一章测试 ADT 的细节问题，因为它们在数据结构和算法的学习中占重要地位。实际上，这本书里我们讨论到的几乎所有算法的开发动机，从本质上说都是为了特定基本 ADT 的基本操作的有效实现，这些基本 ADT 在很多计算任务中扮演关键的角色。设计一个 ADT 只不过是迎合应用程序需要的第一步——我们还需要开发关联操作的可行实现，优先考虑使它们起作用的数据结构。这些任务是本书的主题。更进一步，我们直接使用抽象模型来开发并比较算法和数据结构的性能特征，就像第一章的例子一样：通常，我们使用 ADT 开发应用程序来解决一个问题，然后开发该 ADT 的各种实现来比较它们之间的效率。这一章我们将详细讨论这一通用过程，并给出大量的例子。

C++程序员有规律地使用数据类型和 ADT。在一个较低的层次上，当我们用 C++提供的操作处理整型数据时，我们实际上使用了系统定义的整型数据抽象。在某些新的机器上，整型数据能够被重新表示，操作也能够用不同的方法实现，但是仅仅使用整型数据特殊操作的程序会在那些特定的机器上正确运行。从这个意义来说，各种 C++整型数据操作组成了接口，我们的程序就是客户，系统硬件和软件提供实现。通常，数据类型足够抽象，以至于我们可以在不改变程序的前提下将它们转移到一个新的机器上，例如，通过整型和浮点型数据的不同表示形式。然而这一例子也说明了我们不能随时随地地应用这一原则，因为客户程序能够通过突破资源限制的方式来收集数据的表示信息。例如，我们如果想要知道整型数据在某一台机器上的表示信息，我们可以用一个循环：一直乘以 2，直到报告一个溢出错误为止。第三章里大量的例子是用“C 风格”写成的 C++程序。C 程序员通常在头文件里定义接口，头文件里描述了某些数据结构的一组操作，并在一些独立的程序文件

里实现。这种安排给用户和实现者提供了紧密联系，也是C编程环境里所构造的标准库的基础。然而，许许多多的这一类库只是组成了某一特定数据结构的操作，因此它们组成的是数据类型，而不是抽象数据类型。例如，C中的字符串库就不是抽象数据类型，因为使用字符串的程序知道字符串是怎么表示的（字符的数组），而且能够通过数组下标或者指针算法来直接访问它们。

与之相对应，C++类不仅允许我们使用操作的不同实现，还可以根据它们来决定下面的数据结构。另外，刻划 ADT 的关键区别是我们能够在不修改任何客户程序的前提下做这样的改变，因为要求规定数据类型只能通过接口访问。类定义里的 private 关键字防止客户程序直接访问数据。比方说，我们可以提供基于 C++标准库的 string 类的其他实现，例如一个用链表实现的字符串类，即使这样客户程序使用字符串的时候也不需要任何修改。

通过这一章，我们将会看到大量用 C++类来实现 ADT 的例子。等我们对这一概念有一些感觉之后，我们将在这一章末尾讨论哲学和实践的蕴含。

**练习**

▷ 4.1 假设我们想要计算落在大小为 d 的正方形中的点对的数目，给出两个不同版本的客户和实现来解决这一问题：第一个，适当地修改 distance 函数；第二个，用 X 和 Y 的成员函数替换 distance 函数。

4.2 在程序 4.3 中加上一个成员函数，返回跑原点的距离。

○ 4.3 修改程序 4.3 里的 ADT 实现，用极坐标表示点。

○ 4.4 写一个客户程序，从命令行获得一个整型参数 $N$，然后用 $N$ 个点填充一个数组，保证没有任何两个点是相等的，使用前文描述的重载操作符==来验证相等性。

●●4.5 将 3.4 节里程序 3.12 的列表处理接口转换为基于类的 ADT 实现，用与程序 3.14 同样的方式使用链表来表示。修改客户程序（使用程序 3.13 作为客户），测试你的接口，然后转到一个基于数组的表示（练习 3.52）。

## 4.1 抽象对象和对象集合

我们在实际应用中使用的数据结构通常包括各种类型的大量信息，而且其中很大一部分信息属于多独立性数据结构。举个例子来说，人事数据文件也许会包括人名、地址和各种其他信息的雇员记录，而每一个记录需要为了搜索特定雇员而属于一个数据结构，或者为了回答统计查询而属于另一个数据结构，或者有其他的原因和方式。

不考虑这一多样性和复杂性，一个用于计算应用的大类中包含数据对象的普通处理，而且因为某些特定的原因，需要提供访问关联信息的入口。许多必要的处理过程是从那些基本计算过程发展而来，因此它们可以用在大量的应用中。许多我们在本章讨论到的基本算法，也可以有效地应用到建立抽象层的工作中，可以为客户程序提供有效处理数据的能力。因此我们将在细节上讨论与这些处理相关联的 ADT。它们定义了抽象对象集合的许多

操作，与对象类型无关。

在3.1节里，我们为了写出不依赖对象类型的代码，已经讨论过简单数据类型的使用，那时候我们使用关键字typedef来定义我们的数据项的类型。这种方法允许我们使用同样的代码来定义像整型、浮点型这样的数字，只要改变typedef就行了。如果使用指针，对象类型可以任意的复杂。当我们使用这一方法的时候，我们通常会发现自己对那些对象的操作做了隐含的假设（例如我们在写程序3.2的时候，假定数字类型有加和乘的操作，而且能够转换为浮点数），而且我们没有对客户程序隐藏数据的表示。ADT为我们提供了一个方法，对我们在数据对象上执行的操作做清楚的假设。

这一章的剩余部分，我们将要讨论一些例子，以说明如何使用C++类来建立ADT里的通用数据对象。我们将要看到如何为通用类型Item建立一个ADT，以便我们可以在客户程序里像使用内置数据类型那样使用Item。需要的时候我们在Item类里显式地定义一些操作，我们的算法需要在通用对象上执行这些操作。我们在制定所有的这些对象特性的时候，都不会提供任何数据表示的信息给客户程序。

一旦我们为通用对象实现了Item类（或者选择使用一个内置类），谈到类型时，我们将使用C++的模板机制来编写通用代码。例如，我们可以定义一个通用项（Item）的交换操作符，如下所示：

```
template <class Item>
  void exch(Item &x, Item &y)
    { Item t = x; x = y; y = t; }
```

我们可以用类似的方法实现项的其他简单操作。使用模板，我们可以指定类族（familiy of classes），每一类的项都有一个类族相对应。

解决完通用对象类，我们可以继续讨论对象集合（collection）。这本书讨论到的许多数据结构和算法将被用来实现基本的ADT，这些ADT组成抽象对象集合，从以下两个操作构造得到：

- 插入（insert）一个新对象到集合中；
- 从集合中删除（remove）一个对象。

我们称这样的ADT为广义队列（generalized queue），为了方便，我们通常也包含显式操作来构造数据结构（构造函数），以及对数据结构里的对象进行计数（或者仅仅为了测试一下集合是否为空）。我们也许还需要操作符来解构（析构函数）或者复制（复制构造函数）数据结构。这些构造函数将会在4.8节出现。

当我们插入一个对象的时候，我们的目标很明确，但是当我们从集合里面删除一个对象的时候，我们会得到哪一个对象呢？对象集合的不同ADT实现在实现删除操作时有不同的处理原则，这些原则决定哪一个对象被删除。此外，我们将会遇到许多除了插入和删除操作之外的其他自然操作。这本书里的许多算法和数据结构的设计，目的就是为了支持这些操作的不同子集的有效实现，这些操作具有不同的删除原则和其他约定。这些ADT在概念上很简单，使用上很广泛，而且是大量计算任务的核心内容，因此它们值得我们给予更多的关注。

我们现在来考虑某些这一类的基本数据结构、它们的性质以及应用它们的例子，同时把它们作为例子来说明我们开发 ADT 的基本机制。4.2 节我们将讨论下推栈（pushdown stack），它的规则是执行删除操作时删除那个最近执行插入操作的对象。4.3 节我们将讨论栈的应用，4.4 节讨论栈的实现，包括使用特殊的方法来保证应用和实现的分离。讨论完栈之后，我们回头讨论一下一个新的 ADT 的创建过程，结合第一章讨论的连通性问题中实现的合并-查找（union-find）抽象这一上下文来讨论。在这之后，我们返回到抽象对象集合中去，讨论 FIFO（First In First Out，先进先出）队列和广义队列（它和栈在抽象层次上的惟一区别在于使用了不同的删除规则），广义队列不允许复制项。

正如我们在第三章里看到的一样，数组和链表提供了基本的机制，允许我们插入和删除特定的项。实际上，链表和数组是某些广义队列的实现里的根本数据结构。大家知道，删除和插入操作的时间花费依赖于我们使用的特定结构以及需要插入和删除的特定项。对于一个给定的 ADT 来说，我们的挑战是选择一种数据结构，有效地完成所需要的操作。这一章我们详细测试了几个 ADT 的范例，这些范例由链表和数组提供了一些适当的解决方案。ADT 需要支持操作的能力越强，它的实现就会越复杂，这是我们这本书里讨论的许多算法的主要推动力。

由抽象对象集合（广义队列）组成的数据类型是计算机科学里学习的主要目标，因为它们直接支持了一个计算的基本范例。对于大量的计算来说，我们发现自己有太多的对象需要处理，但是一次只能处理其中的一个。因此，我们需要在处理其中一个时保存好其他的对象。这一操作也许会引发某些额外的操作，比如检查对象是否已保存，或者往集合里增加更多的信息，但是存储对象、根据某些标准取回数据这一类的操作是计算的基本操作。我们将会看到，许多经典的数据结构和算法都符合这一模型。

在 C++里，实现抽象对象集合的类名叫容器类（container class）。我们将会讨论到的一些数据结构是用 C++库或者它的基于模板的扩展（标准模板库）来实现的。但是为了避免冲突，我们很少引用这些类，我们使用第一法则来开发这本书所需的原材料。

**练习**

▷ 4.6 给出一个 Item 的类定义，重载==操作符来支持浮点数之间的相等性比较。如果两个数的差除较大的数（绝对值）后得到的数小于 $10^{-6}$，那么就认为它们相等。

▷ 4.7 给出一个 Item 的类定义，重载==和<<操作符，以使得它能够在处理扑克牌游戏的程序中使用。

4.8 重写程序 3.1，使用一个一般对象类 Item。你的程序应该适用于 Item 的任何类型，应该能够通过<<输出，通过一个静态成员函数 rand()来随机产生初始值，还要有+和/的定义。

## 4.2 下推栈 ADT

在支持集合元素插入和删除操作的所有数据类型里面，最重要的叫做下推栈（pushdown

stack）。

一个栈操作起来有点像一个工作繁忙的教授的收件邮箱：信件在栈里堆得高高的，一旦教授有空看，拿的肯定是顶部的那些信。一个学生的论文可能会在栈底呆上一两天，但一个尽责任的教授会在每个周末清空栈。我们将会看到，计算机程序通常用这种方法来组织。它们频繁地推迟某些任务，因为正在执行其他任务；此外，它们频繁地需要首先返回执行那些最近被延迟的任务。因此，下推栈成为了许多算法的最基本的数据结构。

**定义 4.2** 一个下推栈就是一个 ADT，它由两种基本操作组成，插入（推）一个新项，和移除（弹出）一个最近插入的项。

也就是说，当我们谈及下推栈的时候，我们指的是这样一个描述：它有推（push）和弹出（pop）操作，它非常好地指定了一个客户程序能够能够使用这些操作。对这些操作的某些实现来说，它强迫实行这样的规则来表征一个下推栈：项按照一个后进先出（Last-In, Fist-Out）的规则来移除。

**程序 4.4 下推栈接口**

使用我们在程序 4.3 里已经使用过的约定，我们定义了一个下推栈的公共函数声明，假定栈的数据结构表示和其他与实现相关的代码都在实现的私有部分里面，因此我们可以修改私有部分，而不必改变客户代码。另外，这一接口使用了模板，允许客户程序在栈里使用任何类型的对象（请参阅程序 4.5 和 4.6），这些实现使用了关键字 Item 来作为栈里的对象类型（参阅程序 4.7 和 4.8）。堆栈的构造函数有一个参数，指定了栈里元素数目的最大值。

```
template <class Item>
class STACK
  {
    private:
      // 与实现相关的代码
    public:
      STACK(int);
      int empty() const;
      void push(Item item);
      Item pop();
  };
```

图 4.1 展示了一个简单栈的一系列推和弹出操作。每一次推操作栈大小增加一，每一次弹出操作栈大小减少一。图中，栈里的项按照它们入栈的顺序列表，因此很清楚地看到列表里最右边的项就是栈里最上边的项——如果下一个操作是弹出操作，这一个项将会被返回。在一个实现里面，我们可以按照自己的意愿自由地组织项，只要我们能够保证客户程序保持着这样的错觉就行了：项还是按这种方式组织的。

正如前一节我们所讨论的那样，为了编写使用下推栈抽象的程序，我们需要首先定义

接口。为了达到这一目标，我们的惯例是声明一个公共成员函数的集合，并在类实现中使用它，就像程序 4.4 所举的例子一样。我们让所有其他的类成员保持为私有的，然后 C++就能够保证这些函数是客户程序和实现之间惟一的纽带。从第一章和第三章我们已经看到，一个计算通常基于某些抽象操作，我们已经认识到识别这些抽象操作的价值。现在就让我们来讨论一个机制，它允许我们写使用这些抽象操作的程序。为了坚持抽象原则，我们使用类机制来对客户隐藏数据结构和实现。我们将在 4.3 节讨论使用栈抽象的客户程序的例子；在 4.4 节讨论它的实现。

在程序 4.4 里，栈这个 ADT 接口的第一行代码给这个类添加了一个 C++模板，它允许客户程序自己指定栈中对象的类型。一个类似：

```
STACK<int> save(N)
```

的声明指定了在栈 save 中存放的项的类型是 int（还有任何时候栈中元素个数的最大值是 *N*）。客户能够建立一个以浮点数、字符或者任何其他类型（甚至是栈类型本身）为对象类型的栈，仅仅需要改变尖括号里的模板参数即可。在实现中我们需要考虑指明的类，当 Item 出现的时候替换掉它。

**图 4.1 下推栈（LIFO 队列）示例**

这一列表展示了左边列（从上到下）一系列操作的结果，字母表明推入栈，星号表明弹出栈。每一行显示了操作、弹出操作的结果字符，以及操作之后栈的内容，按照最先插入的在左边、最后插入的在右边的顺序。

ADT 里接口的目的是充当客户和实现之间的契约。函数声明保证了客户程序的调用与实现里的函数定义相匹配，然而接口里并没有包含函数如何实现的任何信息，甚至也没有关于它们如何运行的信息。那么我们如何向客户程序解释栈究竟是什么呢？对于像栈一样的简单结构来说，可行方案之一是显示代码，但这一解决方案很明显通常没有效。程序员更经常做的是求助于语言文字的描述，把解释放在和代码一块的文档里面。

这种状况的精确处理，需要用一些正式的数学符号来完全描述函数应该如何运行。这样的描述有时称为规范（specification）。开发规范通常是一件充满挑战的任务。它必须用数学的元语言来描述实现这些函数的任何程序，尽管我们习惯了在用编程语言写代码时顺便描述一下函数的行为。在实践中，我们用语言文字来描述函数行为。为了不在认识论问题上纠缠，我们继续讨论。这本书中我们详细地给出了各种例子：文字语言描述，和我们讨论过的大多数 ADT 的多种实现。

为了突出我们为下推栈 ADT 建立的规范对于我们写有意义的客户程序是多么的有用，我们将在 4.3 节讨论两个使用下推栈的例子，然后再讨论其他的实现。

**练习**

▷ 4.9　以下序列字母表示推操作，星号表示弹出操作．给出每一次弹出操作的结果值的序列。

E A S * Y * Q U E * * * S T * * * I O * N * * *.

4.10　使用练习 4.9 中的约定，在序列 E A S Y 中适当的地方插入星号，使得弹出操作所返回的值的序列是(*i*)E A S Y；(*ii*)Y S A E；(*iii*)A S Y E；(*iv*)A Y E S；或者对以上每一种情况证明不可能存在这样的序列。

●●4.11　给定两个序列，给出算法来判断是否可以在序列中插入星号，使得由第一个序列可以生成第二个序列。生成规则由练习 4.10 来解释。

## 4.3　栈 ADT 客户示例

这一章我们将看到非常多的栈的应用。作为一个入门性质的例子，我们现在讨论如何使用栈来评价算术表达式。举个例子来说，假设我们现在需要得到一个简单的算术表达式的值，这一算术表达式由整数的乘法和加法构成。就像：

5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 )

这一计算包含了中间过程的结果：例如，如果我们首先计算 9+8，得到 17，当我们计算下一步 4*6 时需要把上一步的结果 17 存起来。下推栈在这一类需要存储中间计算结果的计算中无疑是一个理想的机制。

我们从考虑一个简单问题开始，我们需要求值的表达式的形式是：操作符跟在它的两个参数后面，而不是像通常那样在它们中间。我们将会看到，任何算术表达式都能够用这种形式表示出来，这种表示方法叫做后缀表达式（postfix），与之相对应的是中缀表达式（infix）。中缀表达式是我们写算术表达式时习惯使用的方式。上一个段落的表达式如果用后缀来表示，就是：

5 9 8 + 4 6 * * 7 + *

与后缀表达式相反的是前缀表达式，又叫做波兰表示法（因为它是由波兰逻辑学家 Lukasiewicz 发明的）。

如果算术表达式是中缀表达式，我们需要在其中添加括号来区别不同的计算顺序，例如以下就是两个完全不同意义的算术表达式：

5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 )

( ( 5 * 9 ) + 8 ) * ( ( 4 * 6 ) + 7 )

但是括号在后缀（或者前缀）表达式中是不需要的。想

| | | | | |
|---|---|---|---|---|
| 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 | | | |

**图 4.2　后缀表达式求值**

这一序列表明了在对后序表达式 598+46**7+*求值过程中栈的作用。从左往右处理表达式，如果遇到数字，把它推入栈，如果遇到操作符，弹出栈中两个数字，将它们在该操作符下的执行结果再压入栈。

知道为什么，我们来看看下面一个将后缀表达式变换成中缀表达式的例子：我们把两个操作数后面跟着一个操作符的事件替换成对应的中缀表达式，加上括号来指出所得结果可以被看成新的操作数。也就是说，当表达式中出现 ab*和 ab+时，将它们分别替换成(a*b)和(a+b)。然后我们对结果表达式做同样的变换，一直继续下去直到所有的操作符都已经被处理过。在我们这一个例子中，变换的顺序如下：

```
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 ) )
```

用这种方法，我们能够决定后缀表达式中的操作数到底是跟哪个操作符关联，因此不需要任何括号。

在栈的帮助下，我们实际上可以对任意的后缀表达式进行求值操作，就像图 4.2 一样。我们是这样从左到右作解释的：遇到一个操作数，把它推入栈中，遇到一个操作符，弹出栈中两个操作数，执行所示操作，将得到的结果推入栈。程序 4.5 是这一过程的 C++实现。记住，因为栈这一 ADT 已经模板化了，所以我们可以很容易地使用同样的代码在这个程序中建造一个整数栈，然后再在程序 4.6 中建造一个字符栈。

**程序 4.5 后缀表达式求值**

这是一个下推栈的客户程序，它读取任何由整数乘法和加法组成的表达式，然后对表达式求值，并打印计算结果。本程序在一个整数栈中保存中间结果，假设程序 4.4 这一模板化了的类接口在文件 STACK.cxx 中定义。

当我们遇到操作数时，把它们压入栈；当我们遇到操作符时，从栈中弹出最顶端的两个整数，对它们应用操作符，然后把计算结果压回入栈。两个 pop()操作弹出的整数的顺序在这本 C++代码中并没有被指定，因此对于那些不可交换参数计算顺序的操作符，例如减法和除法，它们的实现代码可能要复杂些。

本程序隐含着这样的假设：整数和操作符之间用某些其他符号（例如空格）分隔。但是程序完全不检查输入的合法性。最后一个 if 表达式和 while 循环的作用和 C++里的 atoi 函数类似，就是将整数 ASCII 字符串转换到整数来计算。当我们碰到一个新的数字时，将累积结果乘以 10 再加上这个数字。

```
#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++)
```

```
    {
      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;
}
```

后缀表示法以及与之关联的下推栈给我们提供了一个组织一系列计算过程的自然途径。一些计算器和计算语言明确地把它们的计算方法放在后缀和栈的操作中——每一个操作都从栈中弹出它的参数，然后把计算结果压回栈中。

PostScript 语言可以作为这样一种语言的例子，这本书就是用这种语言写的。它是一种完全可编程的语言，程序用后缀写成并在一个内置栈中解释执行，就像程序 4.5 一样。虽然我们在此不能覆盖此一语言的所有方面（参见“第二部分参考资料”），但是它十分简单，我们可以通过研究某些实际程序来估计一下后缀表示法以及下推栈抽象的用途。例如，下面这个字符串就是一个 PostScript 程序：

```
5 9 8 add 4 6 mul mul 7 add mul
```

PostScript 包括操作符（如 add 和 mul）和操作数（如整数）。正如程序 4.5 所示，我们通过从左往右依次读取每一项来解释这一字符串：如果遇到一个操作数，则压入栈中；如果遇到操作符，则从栈中弹出它的操作数（如果有的话），然后把结果（如果有的话）压回栈。因此，这一程序的运行情况在图 4.2 中已经详细描述了：程序的最后运行结果 2075 留在了栈中。

PostScript 有许多基本函数，作为抽象图形显示设备的指令来使用，我们甚至可以定义我们自己的函数。这些带参数函数在栈中被调用，和其他函数没什么两样。举例来说，以下 PostScript 代码：

```
0 0 moveto 144 hill 0 72 moveto 72 hill stroke
```

表示这样的操作序列：“用参数 0 和 0 调用 moveto，用参数 144 调用 hill”，以此类推。一些操作符指向栈本身。例如 dup 操作符，功能是复制栈顶项，如，以下 PostScript 代码：

```
144 dup 0 rlineto 60 rotate dup 0 rlineto
```

表示的是这样的操作序列：“用参数 144 和 0 调用 rlineto，用参数 60 调用 rotate，然后用参数 144 和 0 调用 rlineto”，以此类推。图 4.3 中的 PostScript 程序定义和使用了函数 hill。

PostScript 中的函数有点像宏：序列/hill {A}def 使得 hill 成为花括号中的操作符序列的等值操作。图 4.3 是一个 PostScript 程序的例子，定义了一个函数来画一个简单的图表。

```
/hill {
      dup 0 rlineto
      60 rotate
      dup 0 rlineto
      -120 rotate
      dup 0 rlineto
      60 rotate
      dup 0 rlineto
      pop
    } def
0 0 moveto
144 hill
0 72 moveto
72 hill
stroke
```

**图 4.3 PostScript 程序示例**

上图中的图表是用其下的 PostScript 程序写成的。该程序是一个后缀表达式，使用了以下内置函数 moveto、rlineto、rotate、stroke、dup，用户还自定义了一个 hill 函数（定义参看正文）。画图命令是绘图设备的指令：movto 指示设备移动到页面上的指定位置（以像素点为坐标，一个点等于 1/72 英寸）；rlineto 指示设备从当前位置移动到指定坐标位置，并在经过的轨迹上画上线；rotate 指示设备向左转动指定的度数；stroke 指示设备画出轨迹的路径。

通过上下文的讨论，我们对 PostScript 的兴趣集中在一个事实：这一广泛应用的编程语言基于的是下推栈抽象。实际上，许多计算机在硬件层实现了基本的栈操作，因为它们很自然地实现了一种函数调用的机制：进入过程入口的时候，通过压栈操作把当前的环境保存在栈中；退出过程调用的时候，通过弹出操作恢复调用过程前的环境。正如我们在第五章中所看到的，下推栈和以函数调用函数的形式组织起来的程序之间的联系是计算的一个基本范例。

回来讨论我们原来的那个问题，我们也能够使用一个下推栈来将一个加上完整的括号的中缀表达式转换成后缀表达式，图 4.4 可以作为一个示例。在这个计算中，我们将操作符推入栈中，然后简单地将操作数传递到输出设备。然后，每一个右括号表示最后一个操作符的两个参数都已经输出了，也就是说操作符自身已经能够弹出和输出了。

程序 4.6 是这一过程的一个实现。记住后缀表达式中的参数的出现顺序和中缀表达式中的一致。有趣的是，我们发现中缀表达式中左括号并不是必要的。不过，如果我们使用的操作符有不同数目的操作数，则左括号是必要的。（参看练习 4.14）

除了提供两个不同的例子来使用下推栈抽象，我们本节开发的整个中缀求值算法本身也是关于抽象的一个练习。第一，我们将输入转换成一个中间表达形式（后缀表达式）；第二，我们模拟了一个基于栈的抽象机器来解释和对后缀表达式求值。同样的做法在许多当代的编程语言解释器中得到了应用，因为这一做法有效而且容易移植：为一个特定计算机编译一个 C++程序的问题被分解为围绕两个中间表示的任务，因此解释程序的问题就和运行程序的问题分离开来了，就像我们这一章所做的一样。我们将在 5.7 节看到一个相关联的但又不一样的中间表示。

**程序 4.6 中缀－后缀转换**

此程序是下推栈的另一个客户示例，在这一示例中，栈中元素是字符。为了将（A+B）转换到它的后缀形式 AB+，我们忽略左括号，转换 A 到后缀形式，保存+符号在栈中，转换 B 到后缀形式，然后，因为遇到右括号，所以弹出栈中元素，并输出+。

```
#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;
  }
```

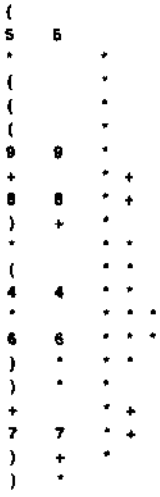


**图 4.4　中缀表达式到后缀表达式的转换**

此序列展示了栈在对中缀表达式(5*(((9+8)*(4*6))+7))向其后缀表达式 5 9 8 + 4 6 * * 7 + *的转换过程中的作用。我们从左至右处理表达式：当我们遇到数字时，直接输出它们；当我们遇到左括号时，忽略它；当我们遇到操作符时，将它推入栈中；当我们遇到右括号时，输出栈顶的操作符。

这一个应用同时也说明了 ADT 和 C++模板的价值。我们不仅使用两种不同的栈，而且其中一个栈放的是 char 类型的对象（操作符），另一个放的是 int 对象（操作数）。通过使用程序 4.4 中定义的类模板化的 ADT，我们甚至能够将两个客户组合到一个程序中（参看练习 4.19）。正因为这种解决方法很吸引人，我们需要小心它可能不是可选的方法之一，因为不同的实现会有不同的性能特性，因此我们不能期望设计一个实现来同时满足两个目标。实际上，我们的重点是各种实现及其性能，下面我们就开始讨论下推栈这一主题。

## 练习

▷ 4.12　将以下表达式转换成后缀表达式：

( 5 * ( ( 9 * 8 ) + ( 7 * ( 4 + 6 ) ) ) )

▷ 4.13　以下是经过程序 4.5 求值得到的表达式，按照与图 4.2 同样的方法，给出栈的内容：

5 9 * 8 7 4 6 + * 2 1 3 * + * + *

▷ 4.14　对程序 4.5 和程序 4.6 进行扩展，使它们能处理-（减法）和/（除法）操作。

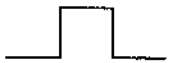
4.15　扩展练习 4.14 的解答，使之包括一元操作符-（相反数）和$（平方根）。然后，修改程序 4.5 中的栈抽象，使之使用浮点数。例如，

给出以下表达式：

```
(-(-1) + $((-1) * (-1)-(4 * (-1))))/2
```

你的程序应该打印出结果值 1.618034。

4.16 写一个 PostScript 程序画以下图案。



● 4.17 用归纳法证明程序 4.5 能够对任何后缀表达式正确求值。

○ 4.18 使用一个下推栈写一个程序，将一个后缀表达式转换成中缀表达式。

○ 4.19 将程序 4.5 和程序 4.6 组合到一个模块中，使用两种不同的栈 ADT：一个整型栈和一个操作符栈。

●●4.20 实现一个编程语言编译器和解释器，用该编程语言写的程序包括一个简单算术表达式，之前有一系列的算术赋值表达式，这些赋值表达式由整数和以单个小写字母命名的变量组成。例如，给出以下输入：

```
(x = 1)
(y = (x + 1))
(((x + y) * 3) + (4 * x))
```

你的程序应该打印出结果值 13。

## 4.4 栈 ADT 的实现

本节我们讨论栈 ADT 的两种实现：一种使用队列，另一种使用链表。这两种实现都是我们在第三章学过的基本工具的直接应用。我们认为它们仅仅在性能特性上有所不同。

**程序 4.7 下推栈的队列实现**

当栈中有 N 个项时，这一实现用 s[0]，...，s[N-1]来保存它们，按照最近插入的顺序排列。栈顶（下一个压入的元素将要占据的位置）元素就是 s[N]。客户程序将项的最大期望数目作为参数传递给 STACK 的构造函数，构造函数分配一个以此为大小的队列。但是本代码并未实现对诸如向一个满栈中压入元素（或者从一个空栈弹出元素）这类操作的错误检测。

```
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]; }
};
```

如果我们使用一个队列来表示栈，程序 4.4 里声明的函数对于实现来说就显得微不足道了，正如程序 4.7 所展示的一样。我们按照图 4.1 的方式把项放进队列中，跟踪栈顶的下标。需要执行压入操作时，只要把项存到队列中栈顶所指示的下标位置即可，然后增大下标；需要执行弹出操作时，减小下标，然后返回当前下标代表的元素。构造操作（构造函数）包括几个步骤：分配指定大小的队列，测试是否为空操作，检查下标是否为 0。和程序 4.5 或者程序 4.6 这样的客户程序一起编译，这一实现就能提供一个有效而且高效的下推栈。

我们知道使用队列表示的一个潜在缺点：正如其他基于队列的数据结构，我们需要事先知道队列的最大长度，这样才能给它分配内存。在这一实现中，我们将这个最大长度作为参数传递给构造函数。这一限制是我们选择使用队列实现的人工因素，并不是栈 ADT 固有的基本部分。我们也许并不容易得到程序会放到栈里的最大元素数目，如果我们选择一个任意大的数字，这一实现对空间的使用将很低效，在一个主要考虑空间资源因素的应用中将不受欢迎。如果我们选择了一个太小的值，我们的程序将根本不能运行。只有使用了 ADT，我们才可能有别的选择，使用别的实现，而不需要更改客户程序。

例如，可以使栈的尺寸动态地增长和收缩，我们也许希望使用一个链表，就像程序 4.8 中一样。不同于栈的队列实现，我们把元素按逆序排列，从最近插入的元素到最初插入的元素，将基本的栈操作以一个更简单的方式来实现，如图 4.5 所示。弹出操作，我们将链表的表头元素移除，并返回它的项；压入操作，我们建立一个新的节点，然后将它加到链表头。因为所有的链表操作都是在表头进行，所以我们不需要一个头节点。

程序 4.8 的代码并没有检查诸如从空栈中弹出元素、往满栈中插入元素或者内存溢出这类的错误。为了检查后两

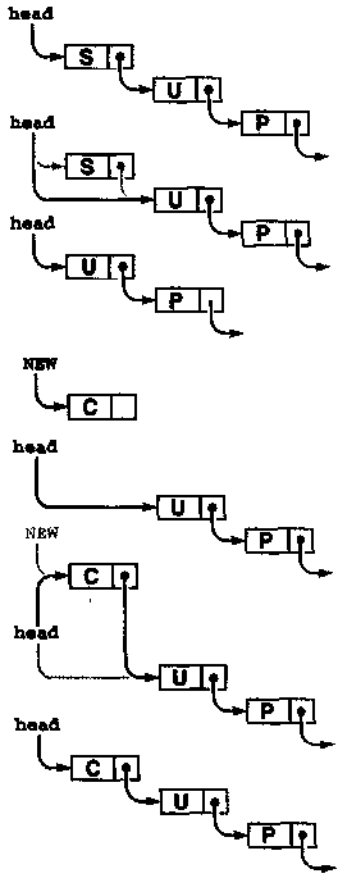



**图 4.5　链表下推栈**

栈表现为指针头，它指向第一个(通常是最新插入的)元素，为了弹出栈顶元素，我们通过指定 head 为它的链接元素移除链表的头元素。为了往栈中压入新元素，我们把它链接到表头，并把它的 link 域置为 head，把 head 指向它。

种错误，我们有两种选择。我们可以把它们看成单独的错误，跟踪链表中项的数目，每一次压栈操作前，检查数目是否超过了我们传递给构造函数的参数或者 new 成功的数目。或者，我们不需要提前知道栈的最大尺寸，仅当我们遇到 new 失败的时候才报告一个栈满的信息，同时忽略传递给构造函数的参数（参看练习 4.24）。

程序 4.7 和 4.8 是同一个 ADT 的两种不同实现。我们可以用其中一个代替另一个，而不需要对客户程序（例如那些我们在 4.3 节测试过的程序）做任何改变。它们的区别仅仅在于性能特性的差异。队列实现为计算中项的最大期望数目分配了所需空间；链表实现则按比例为项的数目分配空间，但常常为每个项一个链接而使用额外的空间，并为每一次 push 和 pop 操作的分配和收回空间花费额外的时间。如果我们通常使用一个很巨大但常常近似为空的栈，我们可以选用队列；如果我们的栈的尺寸动态变化，而且其他程序可以在栈中只有少许项的时候使用那些没有使用的空间，我们可以使用链表实现。

**程序 4.8 下推栈的一个链表实现**

此代码使用了链表来实现下推栈。链表节点的数据实现用正常的方式来组织（参见第三章），还包括了一个节点的构造函数，用指定的项和链接来实例化每一个新节点。

```
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; }
  };
```

这些对空间使用率的考虑，对很多的 ADT 实现来说也是一样的，正如我们通过本书将

会看到的。我们通常站在这样一个两难的位置：是希望快速地访问项却不得不提前分配最大数目所需的空间（队列实现），还是优先考虑灵活性，按项数目的比例分配空间却不得不放弃快速访问任意一个元素的能力（链表实现）。

除了考虑基本的空间使用率问题外，我们常常对 ADT 实现的运行时间的性能差异感兴趣。在这里，我们考虑的两种实现在性能上只有很少的一点差异。

**性质 4.1**　*我们可以在一个常数时间实现下推栈 ADT 的 push 和 pop 操作，不管我们使用的是队列还是链表。*

这一事实在程序 4.7 和 4.8 中将很快得到验证。

栈的项在队列和链表实现中的存储顺序不一样，跟客户程序却没有关系。任意一个实现可以自由地使用任何数据结构，只要它们能够保持着一个抽象下推栈的幻像（illusion）。在这两种情况下，实现能够建立有效的抽象实体的幻像，能够实现也许只要一些机器指令就可以实现的所需操作。纵观全书，我们的目标是为其他重要的 ADT 找到数据结构和有效的实现。

链表实现支持一个栈的幻像，它能够无限地增长。这样一个栈在实际中是不可能的：在某些时候，当对更多的内存的需求得不到满足的时候，new 操作将引发异常。也可能对基于队列的栈的大小实行动态的增长和收缩：当队列已经有一半是满的时候，将队列长度加倍；或者当队列已经有一半是空的时候将队列长度减半。我们将对这一策略的具体实现留到第十四章的练习中去，那时我们将为更多的高级应用考虑这一过程的细节。

## 练习

▷　4.21 使用程序 4.7,执行完图 4.1 所示的操作后，给出 s[0]，...，s[4]的内容。

○　4.22 假设你修改了下推栈中的关于测试栈是否为空的接口函数 test，改成使用 count，它将返回数据结构中当前的项的数目。请你给出 count 在队列中的实现（程序 4.7）以及链表中的实现（程序 4.8）。

4.23 修改程序 4.7 的代码中基于数组队的下推栈实现，如果在栈为空时执行了 pop 操作，或者在栈满时执行了 push 操作，则调用一个 error()函数。

4.24 修改程序 4.8 的代码中基于链表的下推栈实现，如果在栈为空时执行了 pop 操作，或者执行了 new 操作却申请不到新的内存空间，则调用一个 error()函数。

4.25 修改程序 4.8 的代码中基于链表的下推栈实现，使用一个索引数组来实现链表（参见图 3.4）。

4.26 写一个基于链表的下推栈实现，使得链表中项的顺序按照从最后插入到最近插入。你可能需要使用一个双向链表。

●　4.27 开发一个 ADT，提供两种不同的下推栈给客户。使用一个队列实现。其中一个实现使用队列头，另一个使用队列尾（如果客户程序使用两个栈，当一个增长的时候另一个收缩，这一实现就会比其他的实现所使用的空间要小）。

●　4.28 为整数实现一个中缀表达式求值函数，这一函数包括了程序 4.5 和 4.6，使用练习 4.27 中定义的 ADT。记住，你一定要处理这样一个事实：两个栈必须使用同样类型的项。

## 4.5 创建一个新 ADT

从 4.2 到 4.4 节展示了一个完整的 C++代码例子，以实现我们最重要的一个抽象——下推栈。4.2 节的接口定义了基本的操作；4.3 节中的客户程序可以使用这些操作，而不用依靠操作是如何实现的；然后 4.4 节的实现提供了必要的具体表现以及程序代码来实现抽象。

为了设计一个新的 ADT，我们通常遵循以下步骤。首先进行的任务是开发一个客户程序来解决应用问题，我们首先确定看上去至关重要的操作，例如，我们想要那些数据做什么用？然后，我们定义一个接口，写客户代码来测试以下假设，假设 ADT 的存在将使得我们的客户程序的实现变得更简单。接下来，我们考虑是否能够合理有效地实现 ADT 里的操作。如果不能，也许我们能够试着找出究竟是什么原因使得我们失去了效率。然后我们需要修改接口的设计，需要将那些更有效的操作加到 ADT 中去。这些修正影响了客户程序，因此我们也需要修改它们。经过几次反复，我们得到了一个可以工作的客户程序和一个可以工作的实现，因此我们将冻结接口：我们的原则是再也不改变这个接口。这个时候，客户程序的开发和接口的开发就可以分开进行了：我们可以写另一个客户程序来使用同一个 ADT（也许我们需要写一些驱动程序来测试我们的 ADT），我们可以写其他的实现，而且可以比较各种不同的实现之间的性能差异。

在其他情况下，我们也许会先进行 ADT 的设计。这种方法可能需要问以下问题：客户程序处理即将到来的数据需要什么样的基本操作？我们知道的哪些操作可以有效地实现？开发完实现后，我们也许需要在客户程序上测试它的功效。我们也许需要修改接口，在冻结接口之前做更多的测试。

在第一章里，我们讨论了一个详细的例子来考虑一个抽象等级，以便帮助我们找到一个有效算法来解决一个复杂问题。我们接着将讨论使用本章的通用方法来封装我们在第一章开发的抽象操作。

程序 4.9 根据两个操作（除了构造操作之外）定义了接口，这两个操作似乎在一个高级抽象层次下适合我们在第一章讨论的连通性算法。无论算法和数据结构如何重要，我们需要的是能够检查出两个节点是否被认为相连，然后声明那两个节点为相连节点。

程序 4.10 是一个客户程序，使用了程序 4.9 定义的接口 ADT 来解决连通性问题。其中使用 ADT 的好处之一是这个程序很容易理解，因为它是根据允许计算用一个自然的方式表达这一原则来写成的。

**程序 4.9 等价关系 ADT 接口**

我们的 ADT 接口机制使得我们可以非常便利地实现我们的决定，考虑连通性算法，一个支持二个抽象操作的类：初始化一个数据结构来跟踪给定节点之间的连接；找出两个给定节点是否相连；以及将两个给定节点连接起来，以便以后将它们看成是相连的。

```
class UF
  {
    private:
      // 与实现相关的代码
    public:
      UF(int);
      int find(int, int);
      void unite(int, int);
  };
```

**程序 4.10　等价关系 ADT 客户**

程序 4.9 定义的 ADT 把连通性算法与合并-查找的实现分了开来，使得那个算法更加容易理解。

```
#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;
        }
  }
```

程序 4.11 是程序 4.9 里定义的合并-查找接口的一个实现，它使用了一个树的森林的数据结构，这一数据结构通过两个数组来作为已知的连通性信息的基本表示，正如 1.3 节中描述的一样。我们在第一章讨论的不同算法表现了这一 ADT 的不同实现，而且我们可以测试它们，就像根本没有改变过客户程序一样。

这一 ADT 导致了这些程序比起第一章中那些连通性应用程序来说稍微有一些低效，因为它并没有应用到客户的这一特性：每一次 union 操作之后一定马上跟着一个 find 操作。我们有时候会导致这一类额外的费用，这是为了得到一个更抽象的表示而付出的代价。在本例中，有很多的方法来避免低效性，但可能会付出另外一些代价：使接口或表示更复杂（参见练习 4.30）。在实际操作中，路径非常短（特别是当我们使用了路径压缩时），因此那些额外的花费在本例中是可忽略的。

将程序 4.10 和 4.11 组合到一起恰恰等价于程序 1.3，但是将程序分成两个部分是一个有效的方式，因为它：

- 把高层次的问题（连通性）分成低层次的问题（合并-查找）来解决，允许我们独

立地解决这两个问题；

- 给我们一个自然的方式来比较解决这一问题的不同算法和数据结构；
- 通过接口定义了一个方法来检查软件是否按照期望来操作；
- 提供了一个机制来允许我们升级到新的实现（新的数据结构或者新的算法），而完全不用更改客户程序；
- 提供给我们一个抽象，我们可以使用它来写其他的算法。

当我们开发计算机程序的时候，这些好处广泛应用到我们面对的大量任务中，因此以这一基本原则为基础的 ADT 被广泛应用。

我们在程序 4.11 的代码混合了接口和实现，因此并没有提供对客户和实现的单独编译。为了保证不同的实现使用同样的接口正如 3.1 节中所做的那样，我们可以将程序分割成三个文件，如下所示。

**程序 4.11 等价关系 ADT 实现**

本代码从第一章的带权-快速-合并代码得到，实现了程序 4.9 的接口，以一种形式封装了代码，以便我们能够在其他应用中方便地使用它。这一实现重载了函数名 find，将它作为一个私有成员函数，来实现到树根的遍历。

```
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]; }
        }
  };
```

建立一个头文件（例如 UF.h）来包含类声明、数据表示以及函数声明（但不是函数定义）。对于我们讨论到的这个例子来说，它由程序 4.9 的代码组成，但是数据实现（程序 4.11 的私有声明）也包括在里面。然后，把函数定义放在一个单独的.cxx 文件里面，它包含了一个 include 语句指向 UF.h（在所有客户中都要这样做）。这样安排之后，我们就可以独立地编译客户和实现。实际上，我们可以把任意的类成员函数定义放到一个单独的文件里面，只要类声明了这些函数，而且我们在定义的函数前面写上类名和::就行了。举例来说，我们例子里面的 find 函数的定义可以像下面这样写：

```
int UF::find(int p, int q)
  { return (find(p) == find(q)); }
```

很不幸，分开编译时，不同的编译器对不同的模板实例化有不同的要求。问题是如果不知道模板参数的类型，编译器无法为成员函数产生代码，而模板参数的类型在主程序中指定了之后就再也不可访问了。在类声明中定义成员函数避免了这一类的编译问题。

然而，这种 3-文件的惯例仍然不是理想的。因为数据表示虽然真的是实现的一部分，但是它却和接口在一个文件里面。使用关键字 private，我们可以保证客户程序不能访问它。但是如果我们在实现中做了改变，而这些改变又需要数据表示做相应的改变的话，我们就不得不修改.h 文件，然后重新编译所有的客户。在许多软件工程的具体问题中，我们可能对客户程序一无所知，因此这将是一个吃力不讨好的要求。在其他问题中，这一安排却没有问题。对于一个大型而且复杂的 ADT 来说，我们可能会先选定数据表示和一个接口，然后才分配程序员们做具体实现的不同部分。在这种情况下，接口的公共部分将作为程序员和客户之间的联系人的角色存在，接口的私有部分则成为程序员之间的联系人的角色。这一策略也通常正好满足了我们这本书的需求，当我们需要找到一个解决给定问题的最好方法的时候，在这一约束下我们不得不使用一个特定的数据结构。通过这种方法，我们也许能够意识到在一个大的系统中修改一个小模块而获得的性能提高。

C++语言并没有提供这样一个机制，即经过特别设计，允许我们用一个定义好的接口来写程序，并且这一接口与实现完全分开。它基于派生类（derived class）的概念，通过派生我们可以增加或者重新定义一个已存在类的一些成员函数。在一个成员函数声明的前面加上关键字 virtual 意味着这个函数可以在一个派生类中重定义；在成员函数声明后面加上 =0，表示这是一个纯虚函数（pure virtual function），纯虚函数必须在任何派生函数中重新定义。派生类给程序员提供了一个便利的途径，使得他们能够在别人的工作基础上完成自己的工作，派生类还提供了面向对象编程系统的一个基本构成部分。

**程序 4.12 等价关系 ADT 的抽象类**

本代码由一个等价关系 ADT 的接口组成，它在客户和实现之间实现了完全的分离（见代码）。

```
class uf
  {
    public:
      virtual uf(int) = 0;
      virtual int find(int, int) = 0;
      virtual void unite(int, int) = 0;
  };
```

抽象类（abstract class）是指那些所有成员都是纯虚函数的类。任何从一个抽象类派生出来的类都必须重新定义所有的成员函数以及任何所需的私有数据成员；因此，在我们的术语中，抽象类是一个接口，任何从其派生出来的类是实现。客户能够使用这一接口，C++系统能够强制客户和实现之间的联系，甚至客户和函数的实现被分别编译。举例来说，程序 4.12 为等价关系展示了一个抽象类 uf；改变程序 4.11 的第一行为：

```
class UF : public class uf
```

表明 UF 派生自 uf，因此它（至少）需要定义 uf 的所有成员函数，也就是说，它是接口 uf 的一个具体实现。

很不幸，使用抽象类将导致巨大的运行时间花费，因为每一次访问一个虚函数都需要跟着一个指针，指针指向指针表里的成员函数。此外，编译器远远不止这么严格，它们有能力为抽象类产生优化代码。既然本书中我们所讨论的算法和数据结构通常都是系统的性能关键部分，因此我们也许希望通过这些牺牲来获得抽象类提供的灵活性。

然而另一种处理方式也适合 4-文件策略，我们可以把私有部分放到一个单独的文件里，而不是在接口里。举例来说，我们可以在程序 4.9 的类的前面加上两句：

```
private:
#include "UFprivate.h"
```

然后在文件 UFprivate.h 中加上：

```
int *id, *sz;
int find(int);
```

这一策略清楚地分开了四个部分（客户、实现、数据表示和接口），而且在数据结构和算法上给我们提供了最大的灵活性。

这种可以通过派生类以及 4-文件策略获得的灵活性，使得客户与实现之间关于 ADT 是什么的隐含联系也许会在未来被破除。所有的这些机制保证了客户程序和实现能够正确地连接在一起，但是它们还互相依赖，而其方式是我们通常不能正确指定的。例如，假设一些未获得提醒的程序员发现我们的带权快速-查找算法太难理解，然后决定用一个快速-合并算法（或者是更坏的情况，比如用了一个错误的算法，就根本不能得到正确答案的算法）来代替它，我们一直坚持允许这样的修改更容易，但是在本例中，它可能会在一个依赖此实现的关键应用中削弱客户程序。对于大型问题来说，此时可能有很好的性能。编程

的学问里充满着这一类问题，实际上我们很难避免它们。

然而，这一类的讨论把我们推到一个境地，那就是考虑语言、编译器、链接器和运行时系统的特点，或者远远地离开了算法的分析。因而我们通常坚持简单的 2-文件的惯例，我们用 C++类来实现 ADT，公共成员函数由接口组成，接口-实现混合在一个独立的文件中，被客户程序包含起来（而且会随着客户程序的重编译而被重编译）。我们这么做的主要原因是因为类的实现是对我们的数据结构和算法的方便而且简洁的表达。对于某些特定应用来说，如果我们需要额外的灵活性，正如我们刚才提及的其他方法所提供的那些灵活性，我们可以在这些行的代码里面重新组织我们的类。

### 练习

4.29 修改程序 4.11 以便可以通过二分法（halving）使用路径压缩。

4.30 解决程序中提及的低效率问题，通过在程序 4.9 中加上一个操作，将 union 和 find 操作组合在一起提供一个实现放到程序 4.11 里面，从而修改程序 4.10。

○ 4.31 修改我们的等价关系接口（程序 4.9）以及实现（程序 4.11），提供一个函数，返回已知的连接到指定节点的节点数目。

4.32 修改程序 4.11，使用一个数组的结构来替换掉基本数据结构里的并行数组。

○ 4.33 为后缀表达式求值问题构建一个 3-文件解决方案，使用一个整数栈（不使用模板）。确保你的客户程序（程序 4.5 的版本）可以和你的栈实现（程序 4.7 的版本）分开编译。

○ 4.34 修改你以前的练习的解决方案，以便把数据表示从成员函数实现中分离出来（4-文件解决方案）。通过替换一个链表栈的实现（程序 4.8 的版本）来测试你的答案，而不需要重新编译客户程序。

● 4.35 通过使用虚函数，建立一个基于一个抽象类的等价关系 ADT 的完全实现，然后比较它在遇到巨型的连通性问题时与程序 4.11 的性能差别。

## 4.6 FIFO 队列和广义队列

FIFO（First-In,First-Out 先进先出）队列是另一个基本的 ADT，它和下推栈有点相似，但在决定执行 remove 操作移除哪一个元素的时候，它们的使用规则刚好相反。下推栈是移除最近插入的元素，而 FIFO 队列则是移除那些呆在队列中时间最长的元素。

既然先进先出的次序直观上看起来是一个决定下一步做什么的公平方法，也许我们工作繁忙的教授的收信箱操作起来应该像一个 FIFO 队列。然而，教授不可能永远都及时回电话、按时上课！在栈里，一个备忘便条也许会埋在底下，但紧急事件可以在它们出现的时候马上得到处理；在 FIFO 队列里，我们井然有序地对待那些任务，但是每一个任务都必须排队等待。

**程序 4.13 FIFO 队列 ADT 接口**

除了函数的名字，这一接口和程序 4.4 的下推栈接口很类似。这两个 ADT 的区别仅仅在于它们的使用规范，而在接口代码中并没有提及。

```
template <class Item>
class QUEUE
  {
    private:
      // 与实现相关的代码
    public:
      QUEUE(int);
      int empty();
      void put(Item);
      Item get();
  };
```

FIFO 队列在日常生活中到处可见。当我们排队买票看电影或买杂货时，我们实际上遵循了 FIFO 的原则。类似地，FIFO 队列在计算机系统中频繁使用，当我们提供一个以先来先服务为基础的服务时，FIFO 队列用来保存那些已经到达的任务。我们来看另一个例子，它足以说明栈和 FIFO 队列之间的巨大差别，那就是一个杂货店的易过期商品的存货。如果杂货店老板把新的商品放在货架的前部分，而且顾客先拿货架前头的货物，那么这里用的是栈原则，但这将会给杂货店老板带来一个问题，因为货架后头的商品也许会呆在那儿很长一段时间，就会很容易过期。将新到的货品排在货架后头就不会有这个问题。因为杂货店老板能够保证每一样货品呆在货架上的时间最长只能为顾客买完整个货架的货物所需的时间。FIFO 这一基本原则在很多类似的情况下得到了应用。

**定义 4.3** 一个 FIFO 队列是指这样一个 ADT，包括两种基本操作：插入（put）一个新项，以及移除（get）一个最早插入的项。

程序 4.13 是一个 FIFO 队列 ADT 的接口。这个接口与我们在 4.2 节讨论的栈接口的不同之处仅仅在于术语的命名上，比如，对于一个编译器来说，两个接口是完全一样的！这一发现突出了这一事实，抽象本身是一个 ADT

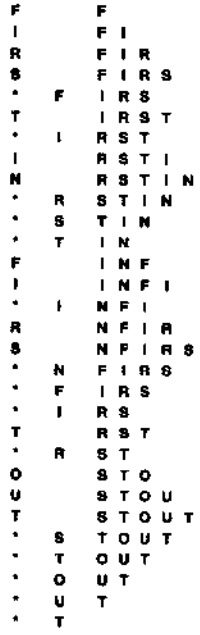


**图 4.6 FIFO 队列的例子**

这个列表展示了左边一列元素的一系列操作（从上到下），字母表示 put，星号表示 get。每一行显示了操作、get 操作返回的字母以及队列的内容，从左到右的顺序是最早插入的到最近插入的。

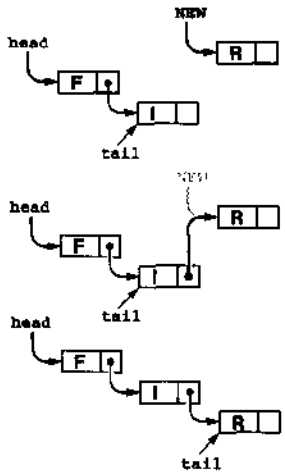



**图 4.7　链表队列**

在这个链表表示的队列中，我们往尾部插入新项，因此链表里的项从头到尾的顺序是从最早插入到最近插入。队列被表示为两个指针，分别是指向第一个项的 head 指针和指向最后一个项的 tail 指针。为了从队列中 get 一个项，我们删除链头的那个项，正如我们在栈中所做的那样（参见图 4.5）。为了 put 一个新项到队列中，我们把 tail 指针所在节点的 link 域指向新的那个项（中间那个图），然后更新 tail，让它指向那个新的项（下面那个图）。

的基本组成部分，但是程序员们却常常并没有很好地定义它们。对于大型的应用来说，它们的表现好坏与 ADT 的贡献有关，因此如何很好地定义 ADT 这个问题变得关键起来。本书中我们所论及的 ADT 仅仅涉及了我们在文章中定义的基本概念，并没有用很正规的语言表述出来，更不用说通过了精确的实现。为了看清楚 ADT 的特性，我们需要讨论使用这些 ADT 的例子，以及检查精确的实现。

图 4.6 为我们展示了一个 FIFO 队列的例子，还包括一系列 get 和 put 操作。每一个 get 操作减少了队列的一个单位的大小，每一个 put 操作又增加了队列的一个单位的大小。图中，队列中的项按照它们插入队列的顺序链接起来，因此可以很清楚地看到列表中的第一项将会是下一个 get 操作所返回的那一项。进一步来说，在一个实现里面我们可以随心所欲地按照我们的意愿组织安排这些项，只要我们保证它们用起来和用这种方式组织起来的项没有差别就行了。

为了用一个链表来实现 FIFO 队列 ADT，我们需要把项按照从最早插入到最近插入的顺序排在链表里面，正如图 4.6 所示。这一顺序与我们原先使用过的栈实现的顺序刚好相反，但是它给我们提供了队列操作的有效实现。我们在链表中加上两个指针：一个在头（从这我们可以 get 第一个元素），一个在尾（从这我们可以 put 一个新元素到队列里面）。如图 4.7 和程序 4.14 所示。

**程序 4.14　FIFO 队列链表实现**

FIFO 队列和下推栈（程序 4.8）的区别在于新项的插入是在尾部，而不是在头部。因此，这个类为链表的最后一个节点指定了一个指针 tail，为了 put 一个新项到队列中，我们把 tail 指针所在节点的 link 域指向新的那个项，然后更新 tail，让它指向那个新的项。其他的函数，例如 QUEUE、get 和 empty，和程序 4.8 下推栈的链表实现中的与之相对应的函数完全一样。既然每一个新项都会加到链表的末尾，那么我们的构造函数就可以把每一个新项的指针域置为空，这样构造函数就只需要一个参数了。

```
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; }
};
```

我们也可以使用数组来实现一个 FIFO 队列，虽然我们不得不特别注意运行时的 put 和 get 操作会不会造成错误。性能方面的要求表明我们不能随意移动数组内的队列元素，不像我们在图 4.6 中建议的文字卡的解释。因此，正如我们在链表实现中所做的那样，我们给数组分配两个索引下标：一个在队列头，一个在队列尾。我们把两个下标之间的元素看成队列的内容。为了 get 一个元素，我们把它从队列的开始（头部）删除，然后增长头索引；为了 put 一个元素，我们把它加到队列的最后（尾部），然后增长尾索引。一系列的 put 和 get 操作使得队列看起来像是在数组里面移动似的，就像图 4.8 所示。当它到达数组的末端时，我们把它卷回到数组的开始部分。这一计算的细节将在程序 4.15 的代码里面说明。

**性质 4.2** *我们可以在常数时间内实现 FIFO 队列 ADT 的 get 和 put 操作，不管使用数组还是链表。*

这一事实在我们考察程序 4.14 和 4.15 的代码时就会马上清楚。

我们在 4.4 节中所讨论的同样内容将应用于 FIFO 队列使用的空间资源。数组表示需要我们为项在计算中所期望的最大数目准备足够的空间。然而，链表表示使用与数据结构中

元素数目成比例的空间，指针占用了额外的空间，每一次操作分配和释放内存又花费了额外的时间。

因为栈和递归的基本关系（参看第五章），我们遇到栈的机会比我们遇到 FIFO 队列的机会要多。尽管如此，我们也将遇到一些以队列作为自然的基本数据结构的算法。正如我们已经注意到的，在计算应用程序中对队列和栈的使用之一就是在延期计算（postpone computation）中。尽管许多包括一个不完全队列的应用程序能够正确地操作，而不需要管 remove 用的是什么规则，但运行时间和其他资源使用的总量也许会由这一规则决定。如果这样一个应用程序包括大量的对数据结构的 insert 和 remove 操作，而且这些数据结构包括大量的项，那么性能差异就会变得极为重要。因此，我们在本书中需要对这样的 ADT 特别注意。如果我们忽略性能，我们将要阐述一个包含 insert 和 remove 操作的简单 ADT；既然我们不会忽略性能，那么从本质上说，每一条规则组成一个不同的 ADT。为了评价一个特定 ADT 的有效性，我们需要讨论两类开销：实现开销，它依赖于我们实现中对算法和数据结构的选择；还有在影响到客户的性能方面特定的决定-选择（decison-making）规则的开销。作为本节的总结，我们将描述一组此类 ADT，并贯穿在本书的讨论中。

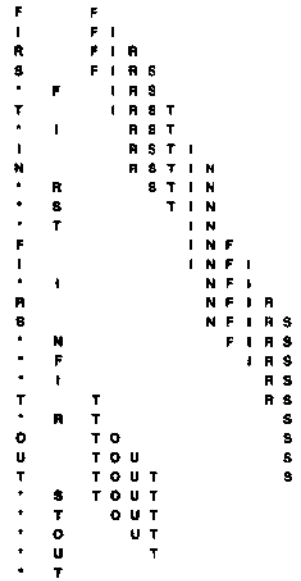


**图 4.8　FIFO 队列的例子，数组实现**

这一串序列展示了当我们通过在一个队列中存储项来实现队列时，在图 4.6 中的抽象表示之下的数据操作。跟踪队列中开始和结尾的下标，然后当它们到达数组结尾时把下标回卷到开始的地方。在这个例子里面，当第二个 T 插入的时候，tail 下标回卷到开始的地方，head 下标在第二个 S 删除的时候也需要回卷。

**程序 4.15　FIFO 队列的数组实现**

队列的内容是数组中从 head 到 tail 的所有元素，当 tail 遇到数组结尾的时候将会回卷到 0。如果 head 和 tail 重合，我们认为此时队列为空；但是如果一个 put 操作造成了它们相等，我们认为此时的队列为满。通常我们不检查这样的错误条件，但是我们设定数组长度的时候，会将它设为比客户将在队列中放置的元素最大数目大 1，因此我们可以为这个程序加上这样的检查。

```
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++]; }
};
```

特别是，下推栈和 FIFO 队列都是一个更通用的 ADT 的具体实例：那就是广义队列（generalized queue）。广义队列的实例之间的差别仅仅在于删除项时使用的规则。对于栈来说，规则就是“删除最近插入的项”；对于 FIFO 队列来说，规则就是“删除最先插入的项”；当然还有其他的很多可能性。

一个简单但是功能强大的选择是使用随机队列（random queue），这种数据结构的使用规则是“删除一个随机的项”，因此客户程序就能够在相等的概率下得到队列的任意项。我们能够在常数时间下实现随机队列的操作的实现，只需要使用一个数组表示（参见练习 4.48）即可。和栈以及 FIFO 队列一样，数组实现也需要我们预先保留空间。但是，链表这一选择与栈和 FIFO 队列相比就没有那么吸引人了，因为有效地实现它的插入和删除都是一个复杂的任务（参见练习 4.49）。我们可以使用随机队列作为随机的算法的基础，尽最大可能地避免最坏性能的情况发生（参见 2.7 节）。

前文已述，可以按照项插入队列的时间识别它们，我们已经通过这一方法来描述栈和 FIFO 队列。另外一种可选方法是，我们可以根据有次序地列出项的顺序来描述这些抽象概念，以及从链表的开头和结尾插入和删除的基本操作。如果我们在末尾插入同时亦是从末尾进行删除，那么我们得到一个栈（就和我们的数组实现一样）；如果我们在开头插入同时也从开头删除，我们还是得到一个栈（就和我们的链表实现一样）；如果我们在末尾插入并从开头删除，那么我们得到的就是 FIFO 队列（就和我们的链表实现一样）；如果我们在开头插入并从末尾删除，那么我们得到的还是 FIFO 队列（这种选择并没有和我们的任何一种实现相对应——我们当然也可以把我们的数组（此处原文为 array，但纵观上下文，应该是链表——译者注）实现修改一下然后精确地实现它，但是链表实现在此并不适用，因为当我们删除链表末尾的项时，我们需要备份链表尾部的指针）。根据这种观点来创建 ADT 的话，我们将会得到双端队列（deque）ADT，它允许我们在任何一边进行插入和删除操作。我们把这一实现留作练习（参见练习 4.43 到 4.47），除了基于数组的实现，没有其他任何东西是程序 4.15 的直接扩展，而且链表实现需要一个双端链表（doubly linked list），除非我们限制双端队列只允许在一端进行删除操作。

在第九章里，我们将讨论优先队列（priority queue），在优先队列里面，每个项有一个键（key）和值（value），并且删除的规则是“删除有最小的键的那个项”。优先队列 ADT 在大量的应用中都很有用，而且很多年以来，为这一 ADT 找一个有效的实现已经成了计算机科学的一个研究目标。定义并且在应用中使用这一 ADT 已经成了研究工作的一个重要组

成部分：我们可以马上指出一个算法是否正确，只需用它代替一个大型复杂系统里的旧版实现，然后检查一个它们是否得到一样的结果就行了。进一步来说，不管一个新算法是否比旧的要有效，我们总算得到一个直接的指示：用新的实现来代替旧的增加了总的运行时间。我们在第九章为了解决这个问题而讨论的数据结构和算法有趣、有独创性而且有效。

从第十二章到第十六章，我们将讨论符号表（symbol table），它是广义队列的一种，每项有键这一属性，删除规则是："如果存在的话，删除那些键与给定键相等的项。"这一 ADT 也许是我们所讨论到的所有 ADT 中最重要的一个了，我们将会检查关于它的数十个实现。

这类 ADT 中的每一个又会引出一堆与之相关联，又有所差别的 ADT，那些 ADT 把自己看成是对客户程序和实现的性能的仔细检查的副产品。在 4.7 节和 4.8 节里，我们将讨论大量关于对广义队列的规范的修改例子，引出更多不同的 ADT，当然，这些都将在本书后面加以讨论。

## 练习

▷ 4.36　使用程序 4.15，执行图 4.6 指示的操作，然后给出 q[0]，...，q[4]的内容。假定图 4.8 中的 maxN 为 10。

▷ 4.37　在以下序列中，字母表示 put，星号表示 get，这一系列操作在一个初始为空的 FIFO 队列中执行，给出每一次 get 操作返回的值的序列。

E A S * Y * Q U E * * * S T * * * I O * N * * *

4.38　修改程序 4.15 中的基于数组的 FIFO 队列实现，加上一个 error()函数，使得当一个客户试图从空队列 get，或者向满队列 put 的时候调用它。

4.39　修改程序 4.14 中的基于数组的 FIFO 队列实现，加上一个 error()函数，使得当一个客户试图从空队列 get，或者向队列 put 却没有内存可用于 new 的时候调用它。

▷ 4.40　在以下序列中，大写字母表示 put 到开头，小写字母表示 put 到结尾，加号表示从开头 get，星号表示从末尾 get。这一系列操作在一个初始为空的双端队列中执行，给出每一次 get 操作返回的值的序列。

E A s + Y + Q U E * * + s t + * + l O * n + + *

▷ 4.41　使用练习 4.40 的方法，给出一个方法，使得我们在序列 E a s Y 之间插入加号和星号后，该序列 get 操作返回：(*i*)E s a Y；　(*ii*) Y a s E；　(*iii*) a Y s E；　(*iv*)a s Y E；或者证明不可能存在这样的序列。

● 4.42　给定两个序列，给出一个算法来确定是否可能通过加上加号和星号，使得第一个序列能够产生第二个序列，作为一个双端操作的序列，按照练习 4.41 所示的方法来解释。

▷ 4.43　写一个双端队列的 ADT 的接口。

4.44　为你的双端队列接口（练习 4.43）提供一个实现，使用一个数组作为基本数据结构。

4.45　为你的双端队列接口（练习 4.43）提供一个实现，使用一个双链表作为基本数据结构。

4.46 为你的双端队列接口（练习 4.43）提供一个实现，使用一个循环链表作为基本数据结构。

4.47 写一个客户程序来测试你的双端链表 ADT（练习 4.43），通过读取一个如同练习 4.40 般的命令字符串，然后执行指示操作。字符串作为命令行的第一个参数提供。往接口和实现中加上一个成员函数 dump，用来在每一次操作后打印出双端队列的内容，使用图 4.6 的风格。

○ 4.48 写一个接口和一个实现来建立一个随机队列（random-queue）ADT，使用数组作为基本的数据结构。确保每一个操作都是常数时间复杂度。

●●4.49 写一个接口和一个实现来建立一个随机队列（random-queue）ADT，使用链表作为基本的数据结构。提供 insert 和 remove 的实现，尽你的最大努力提高它们的效率，然后分析它们的最坏情况开销（worst-case cost）。

▷ 4.50 写一个客户程序，为抽奖抽取数字。通过把数字 1–99 放进一个随机队列中，然后打印删除它们中的五个数字得到的结果。

4.51 写一个客户程序从命令行的第一个参数中取一个整数 $N$，然后打印出 $N$ 个扑克牌局（poker hand），方法是把 $N$ 个项放到一个随机队列中（参见练习 4.7），然后打印出从队列中一次拣出五张牌的结果。

● 4.52 写一个程序解决连通性问题，通过把所有的对插入到一个随机队列中，然后又从队列中把它们拿出来。使用带权的快速查找算法（程序 1.3）。

## 4.7 复制和索引项

对于很多应用来说，我们所处理的抽象项是独一无二（unique）的，它们的这个性质导致我们开始考虑改变关于栈、FIFO 队列或者其他广义 ADT 应该怎样操作的问题。特别地，在这一节里面，我们将讨论改变栈、FIFO 队列和其他广义队列的规范的效果，如何使它们不接受数据结构的重复项。

举例说明，我们公司里有一个顾客的邮件列表，这个列表可能会随着 insert 操作而增长，这些 insert 可能来自很多其他的来源，但是不允许插入一个已经存在的顾客资料。我们可以看到在很多应用中都需要遵循同样的原则。但是，我们考虑另一个例子，就是在一个复杂的通信网络中路由信息。我们可能会试着在网络中同步走几条路径，但是信息只有一条，因此任何特定的网络节点在它的内部数据结构中都只能有这一信息的一个拷贝。

解决这一状况的一种方法是让客户保证重复的项不会在 ADT 里出现，这种方法可能会导致客户去使用一些不同的 ADT。但是既然一个 ADT 的设计目的是给客户提供一个应用

**图 4.9 无重复项的下推栈**

这一序列展示了执行了和图4.1中一样的操作之后的结果，但是这回用的是不允许重复项的栈。灰色正方形标志的状态表示栈在操作后没有改变，因为正要压入栈的项在栈中已经存在。栈中项的数目被可能不同的项的数目所限制。

程序的完整解决方案，我们应该把检测和解决重复性的问题作为 ADT 应该解决的问题的一个部分。

不接受重复项这一原则是抽象（abstraction）的一个变化：遵循这个原则的 ADT 的接口无论从名字还是操作规范上看都和那些不遵循这一原则的 ADT 接口一样，但是它们实现的行为从根本上给改变了。通常，不管我们如何修改一个 ADT 的规范，我们总是得到一个全新的 ADT——一个和原来的相比有着完全不同属性的 ADT。这种状况也证明了 ADT 规范的不稳定性：保证客户和实现通过接口依附到规范中是完全不足够的，但是像这样强行符合一个高级原则又会引来新的问题。但是，我们仍然对能够实现这一原则的算法很感兴趣，因为用新的方法、新的实现开发这样的特性来解决问题，能够从这一类限制中得到好处，以便提供更多有效的解决方案。

图 4.9 展示了一个修改后的无重复项栈 ADT 如何执行图 4.1 中所示的操作；图 4.10 展示了 FIFO 队列改变后的影响。

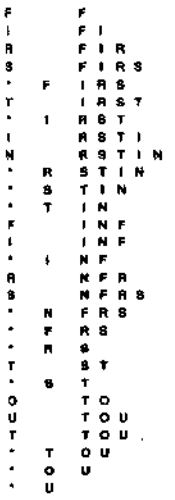


**图 4.10　无重复项的 FIFO 队列，忽略新项原则**

这一序列展示了执行了和图 4.6 中一样的操作之后的结果，但这回用的是不允许重复项的 FIFO 队列。灰色正方形标志的状态表示队列在操作后没有改变，因为正要放入队列中的项在队列中已经存在。

通常，客户在为一个已存在的项提交 insert 请求时，我们有相应的对策。但是我们应不应该若无其事地执行这个操作（也就是说，我们应不应该执行起来就像这个请求从未提交过）？或者说，我们应不应该执行起来像是在插入之后紧跟着一个删除？这一决定会影响到 ADT 中的项的顺序，尤其是栈和 FIFO 队列（参见图 4.11），这样的区别对于客户程序来说可能是很重要的。例如，公司使用这样的 ADT 来处理邮件列表，如果用这种插入原则的话，那么如果该顾客已经存在（假定这个顾客有很多更新信息），也许会造成系统忽略掉新的项（或者新项已经被处理掉）。进一步来说，这一原则的选择影响了 ADT 的实现：忘掉旧的项原则会比忘掉新的项更难实现，因为它需要我们修改数据结构。

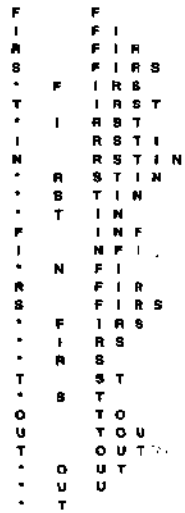


**图 4.11　无重复项的 FIFO 队列，忽略旧项原则**

这一序列展示了执行了和图 4.10 中一样的操作之后的结果，但使用了一个（更难实现的）原则，我们通常在队列尾添加新项，如果有重复，则删掉它。

为了实现广义队列的无重复性，我们假设我们有一个抽象操作来测试项的等价性，就像 4.1 节里讨论的一样。给定这样一个操作，我们仍然需要检测一个将要插入到数据结构中的新项是否已存在。通常的做法是实现符号表（symbol table）ADT，因此我们可以在第十一章到第十五章里给出的

实现中讨论这一 ADT。

还有一种特别情况我们是可以直接解决的，那就是程序 4.6 里讨论的下推栈 ADT 实现。这一实现假设项都是 0~$M$-1 的整数。然后，它使用了第二个数组，用项本身作为下标索引，这一数组用来检查一个项是否已经存在于栈中。当我们插入项 $i$ 时，我们把数组里的第 $i$ 个项置为 1；当我们删除项 $i$ 时，我们把数组里的第 $i$ 个项置为 0。因此我们可以像以前一样，使用同样的代码来插入、删除项，只需要增加一个测试操作：插入一个项之前，我们可以检测一下它是否已经存在于栈中。如果是，我们忽略 push 操作。这一操作不依赖我们使用的是数组或者链表（还是别的什么）实现。实现一个忽略旧项原则包括了更多的工作（参见练习 4.57）。

**程序 4.16 有下标且无重复项的栈**

这一下推栈实现假设每一个项有一个整型类型转换返回 0~maxN-1 的整数，因此它可以维持一个数组 t，这个数组里的非零值对应着栈里的每一个项。这个数组允许 push 操作很快地检查它的参数是否已经存在于堆栈中，然后如果已经存在，则忽略掉这个操作。我们在 t 里面对每一个项只使用一个位的空间，因此在必要的情况下，我们可以通过使用字符或者位代替整数来节省空间。

```
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; }
      void push(Item item)
        {
          if (t[item] == 1) return;
          s[N++] = item; t[item] = 1;
        }
      Item pop()
        { t[s[--N]] = 0; return s[N]; }
  };
```

总而言之，使用忽略新项原则实现一个无重复项的栈的方法是，保持两个数据结构：

第一个像以前一样用来保持栈中的元素，以记录中元素插入时的顺序；第二个用来跟踪哪些元素在栈中；同时用这一个项做下标索引。这样使用数组其实是符号表实现的一个特例，我们将会在 12.2 节中讨论符号表。我们可以在任何的广义队列中使用同样的技术，只要我们知道广义队列中的项是 0~$M$-1 的整数就可以了。

这样的特例越来越多。最重要的例子是当这些数据结构中的项以它们本身作为索引的时候，因此我们把这样的项叫做索引项（index item）。典型地，我们有一组M对象，保存在另一个数组中，我们需要把一个广义队列结构传递给一个更加复杂的算法作为它的一部分。对象通过下标放到队列中，然后当它们被删除的时候才处理，而且一个对象仅仅处理一次。在队列中使用数组下标可以直接满足无重复性原则。

每一种选择（不允许重复，或者允许；使用新项，或者不使用）都导致一个新的 ADT。它们之间的差别很小，但它们明显地影响到客户对 ADT 的动态行为，并影响到我们对算法和数据结构的选择来实现各种操作，因此我们没有别的选择，只能够分别对待所有的这些 ADT。进一步来说，我们有其他选择需要考虑：例如，我们也许希望修改接口，使得客户程序能够在它需要插入新项时一眼就能看到这个 ADT 是否允许插入重复项，或者提供一个选择给客户程序，让它自己决定是否忽略新项或者忘掉旧项。

当我们非正式地使用一个项，例如下推栈、FIFO 队列、双端队列、优先队列或者符号表的时候，我们暗地里指的是一族（family）的 ADT，每一种都定义了不同的操作和对这些操作的不同约定，对某些情形来说，每一种需要不同的、更高级的实现来有效地支持这些操作。

## 练习

▷ 4.53 对照着图 4.9，为无重复项栈画一个图表，使用忘记旧项原则。

4.54 修改 4.4 节中标准的基于数组的堆栈实现（程序 4.7），使之不允许重复项，使用忘记旧项原则。使用一个强制的方法，包括扫描整个栈。

4.55 修改 4.4 节中标准的基于数组的栈实现（程序 4.7），使之不允许重复项，使用忘记新项原则。使用一个强制的方法，包括扫描整个栈，必要的时候重排整个栈。

● 4.56 为 4.4 里的基于链表的栈实现（程序 4.8）做同练习 4.54 和 4.55 一样的操作。

○ 4.57 开发一个无重复项下推栈实现，为 0~$M$-1 之间的整数项使用一个忘记旧项原则，保证 push 和 pop 操作花费常数的时间。提示：使用一个双端链表作为栈的表示，并且保留对节点的指针到一个项索引数组中，注意数组中的项值不仅仅只是 0 和 1。

4.58 为 FIFO 队列 ADT 做同练习 4.54 和 4.55 一样的操作。

4.59 为 FIFO 队列 ADT 做同练习 4.56 一样的操作。

4.60 为 FIFO 队列 ADT 做同练习 4.57 一样的操作。

4.61 为随机队列 ADT 做同练习 4.54 和 4.55 一样的操作。

4.62 为你的练习 4.61 的 ADT 写一个客户程序，实现一个无重复项的随机队列。

## 4.8 一级 ADT

从 4.2 节到 4.7 节，我们讨论了栈和 FIFO 队列 ADT，它们的接口和实现达到了在实现中对客户程序隐藏数据结构实现细节的目的。这样的 ADT 用途广泛，还可以作为我们这本书里讨论到的许多实现的基础。

当我们在使用定义在我们自己的程序里的数据类型时，如果像使用内置数据类型（例如 int 或者 float）那样使用它们，我们就会遇到陷阱（pitfall）。在这一节里，我们将讨论如何处理 ADT 才能使得它们能够在客户程序里用操纵内置数据类型的方法来操纵，我们仍然追求的目标是对客户程序隐藏实现细节。

**定义 4.4** *一级数据结构指的是我们在程序中能像使用内置数据结构那样使用它们的数据结构。*

举例说明，我们可以使用一级数据结构来声明变量并指定值给它们，而且我们可以把它们作为函数的输入参数和返回值参数。就像和数据结构关联的其他定义一样，如果我们不深层地考虑与数据结构相关的操作的语义，我们就不可能很好地定义它们。正如我们将要看的，当 a 和 b 都是用户定义类型时，写一个表达式 a=b 是一回事，它代表什么意思就是另一回事了。

在一个理想世界中，我们可以假设所有的数据类型都有一些已经定义好的全局操作；而实际上，每一种数据类型都以它自己的操作集为特征。数据类型之间的这些差别影响到一级数据结构这个概念的精确定义，因为这些差别暗示我们应该为每一个操作提供定义，但我们很少这样做。更经常的是，只有一些关键的操作对我们来说才是重要的，我们希望能够像使用内置类型一样使用这些我们自己的数据类型。

在很多编程语言中，建立一级数据类型很困难，甚至不可能；在 C++中，类的概念和重载操作符的能力正是我们所需要的基本工具。实际上，正如我们将会看到的，在 C++里把一个类定义成一级数据类型是一件很容易的事；进一步来说，即使它们不是，要把它们升级成一级数据类型也有一套定义好了的路子。

我们在 C++里用来实现一级数据类型的方法对于任何类来说都是适用的：特别是，它适用于广义队列，因此这些方法给我们提供了一种能力——像操纵 C++里的其他数据类型一样操纵栈和 FIFO 队列。这种能力对于我们的算法学习来说是很重要的，因为它提供给我们一个自然的途径来表达对这类 ADT 的高级操作。举例来说，我们可以说，联接（join）两个队列就是把它们组合成一个队列。我们将为优先队列 ADT（第九章）和符号表 ADT（第十二章）讨论实现这类操作的算法。

如果一个一级数据结构只能够通过接口访问，那它就是一个一级 ADT（参见定义 4.1）。能够像操纵内置数据类型（如 int 或者 float）那样操纵 ADT 的实例，是我们设计许多高级编程语言的重要目的，因为它允许任何客户程序写成操纵那些仅仅与应用有关的对象；它允许许多程序员同步地开发一个复杂系统，所有人都用的是一组精确定义的操作；它还提供给那些准备用不同的方式实现的抽象操作，而不需要对应用程序的代码做任何的改变

——例如，为了新的机器和编程环境。

**程序 4.17　复数驱动程序（单位根）**

这一客户程序使用 ADT 实现了一个关于复数的计算，允许它们和那些直接定义的 Complex 类型变量进行计算，以及在算术表达式里面使用它们，当然，需要通过重载操作符。这个程序通过计算单位根的幂次来检查这一 ADT 的实现。通过重载操作符<<的适当定义（参见练习 4.70），程序打印出图 4.12 中的表。

```
#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;
      }
  }
```

我们通过把复数抽象作为一个范例来讨论，开始我们对一级 ADT 的学习。我们的目标是能够写一个像程序 4.17 那样的程序，使用 ADT 上定义的操作来实现在复数上的算术操作。这个程序声明和初始化了复数，而且使用了*=和<<操作。我们也许希望使用几种其他的操作，但是我们仍然只讨论这两种来作为例子。实际上，我们将会讨论在 C++库里使用复数类，它重载了所有的相应操作符，甚至包括以负数作参数的三角函数操作。

程序 4.17 依赖于复数的一些数学特性；我们现在暂时离一下题，先简要讨论一下这些特性。换一种角度考虑，我们根本没有离题，因为预期到作为数学抽象的复数间的关系以及在计算机程序里它们的表示，这种讨论将会很有趣。

数字 $i=\sqrt{-1}$ 是一个虚数。虽然作为实数来说 $\sqrt{-1}$ 没有意义，我们仅仅这样定义 $i$，并实现 $i$ 上的算术操作，任何时候只要出现 $i^2$ 就用-1 来代替。一个复数由两部分组成，实部和虚部——复数能够写成 $a+bi$ 的形式，此时 $a$、$b$ 都是实数。为了进行复数的乘法，我们应用常用的算术规则，记得在 $i^2$ 出现的时候替换成-1。例如：

$$(a+bi)(c+di)=ac+bci+adi+bdi^2=(ac-bd)+(ad+bc)i$$

当我们进行一个复数乘法的时候，实部或者虚部都有可能省略掉（即为 0）。例如：

$$(1-i)(1-i)=1-i-i+i^2=-2i$$
$$(1+i)^4=4i^2=-4$$
$$(1+i)^8=16$$

把上一个的等式两边同时除以 16，考虑到$16=(\sqrt{2})^8$，我们发现

$$(\frac{1}{\sqrt{2}}+\frac{i}{\sqrt{2}})^8=1$$

通常来说，有很多的复数经过幂次操作之后会等于 1。这些复数叫做复单位根（complex roots of unity）。实际上，对每一个整数 $N$，都有恰好 $N$ 个复数 $z$ 满足 $z^N=1$。这些数很容易就能够写出来满足这个性质（参见练习 4.68）：例如，把 $k=1$ 和 $N=8$ 代入下式，就可以得到我们刚才发现的 8 次复单位根。

$$\cos(\frac{2\pi k}{N})+i\sin(\frac{2\pi k}{N}),\qquad k=0,1,\cdots,N-1$$

程序 4.17 计算了任意给定 $N$ 的每一个 $N$ 次单位根，然后对它们每一个执行 $N$ 次方操作，使用 ADT 里定义的*=操作符。程序的输出可以在图 4.12 中得到。我们希望每一个数经过 $N$ 次幂操作后得到同样的结果：1 或者 1+0$i$。我们在计算中会遇到这样的问题，有时候计算结果的虚部并不恰好是 0，这仅仅是计算精度的问题。我们所关心的是这个程序能够想使用内置数据类型一样使用复数，然后，我们会从细节上讨论为什么能够这样做。

```
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
```

**图 4.12 复单位根**

这一个表格给出了当我们用 8 作为 a.out 的命令行参数时，程序 4.17 的运行结果。该程序同时重载了<<操作符，以便有合适的输出结果格式（参见练习 4.70）。这八个复单位根分别为±1、±$i$ 和$\pm\frac{\sqrt{2}}{2}\pm\frac{\sqrt{2}}{2}i$（左边两列）。这八个复数中的任何一个在做八次方的幂运算后都得到结果 1+0i（右边两列）。

**程序 4.18 复数的一级 ADT 接口**

这个关于复数的接口允许实现建立 Complex 类型的对象（初始化为两个 float 值），访问实部和虚部，以及使用*=操作符。只要没有显式地指定，默认机制为所有的类都允许我们在赋值表达式或者在函数返回表达式中使用 Complex 对象。

```
class Complex
  {
    private:
      // 与实现相关的代码
    public:
      Complex(float, float);
      float Re() const;
      float Im() const;
      Complex& operator*=(Complex&);
  };
```

就算是这样简单的例子，保证数据类型的抽象性也是很重要的，因为还有另外一种标准表示我们可能希望考虑使用的：极坐标（参见练习 4.67）。程序 4.18 是一个能够被客户程序（例如程序 4.17）使用的接口，程序 4.19 是使用标准表示的实现（这个表示是指用一个浮点数作实部，一个浮点数作虚部）。

当我们在程序 4.17 里说 x = t 的时候，x 和 t 都是 Complex 类型的对象，系统分配内存给一个新对象，然后拷贝与 t 关联的对象的值到新的对象里面。同样的过程在我们使用一个 Complex 对象作为输入参数或者函数返回值的对象时也会发生。还有，当一个对象超出它的有效范围时，系统收回它占用的存储空间。例如，当程序 4.17 执行完 for 循环后，Complex 对象 t 和 x 生命周期完结，此时系统收回它们占用的存储空间，正如系统收回浮点数 r 的存储空间一样。简而言之，我们像使用内置数据类型一样使用 Complex 类型——因为它是一个一级数据类型。

**程序 4.19 复数的一级 ADT**

本代码实现了程序 4.18 定义的 ADT，使用浮点数表示每一个复数的实部和虚部。它是一个一级数据类型，因为数据表示里面没有指针。当我们在一个赋值表达式里使用 Complex 对象时，或者把复数作为函数的返回值参数时，系统分配空间给一个新的 Complex 对象，然后复制数据，就像内置类型一样。

```
#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; }
```

实际上，任何有这种性质的类，只要它的数据成员中没有指针，那么它就是一个 C++里的一级数据类型。当一个对象被复制时，它的每一个数据成员都会被复制；当一个对象被赋值时，它的每一个数据成员都被改写；当一个对象结束了它的生命周期时，它的存储空间被收回。系统对这样的每一种情况都有默认机制对付：每一个类都有默认的复制构造函数（copy constructor）、赋值操作符（assignment operator）以及析构函数（destructor）来实现这些功能。

然而，当一个数据成员是一个指针时，这些默认函数的效果将完全不同了。对于一个赋值操作来说，默认的复制构造函数对指针进行了复制，但这是我们真正想要的效果吗？复制语义（copy semantics）在任何 ADT 设计里面都是值得讨论的一个重要问题。更经常的说法是，存储管理问题对于在软件工程中使用 ADT 来说很关键。下面，我们讨论一个例子，将会有助于我们更仔细地测试这些问题。

程序 4.20 例示了一个客户程序，它操纵 FIFO 队列作为一级数据类型。这个程序在一组 *M* 队列中模拟了一个特定的环境，包括顾客的到达和接收服务。图 4.13 是这个程序输出结果的一个例子。我们对这个程序的兴趣在于展示一级数据类型机制如何允许它本身和队列 ADT 本身作为高级对象一起工作——我们可以想像写类似的程序来测试各种组织队列以对顾客提供服务的方法，诸如此类。

```
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
```

**图 4.13 随机队列模拟**

这一个列表给出了用 80 作为程序 4.20 的命令行参数后得到的结果。它展示了执行完指定操作之后的队列的内容，我们随机选择一个队列 put 下一个项进去，然后随机选择另一个队列，如果不为空就 get 一个项出来。

假设我们使用的是我们在程序 4.14 中讨论的链表队列实现。当我们说 p = q 时，系统给一个新对象分配内存空间，然后把 q 的值赋给这个新对象，其中 p 和 q 都是 QUEUE 对象。但是这些赋的值是 head 和 tail 指针——链表本身并没有被复制。随后如果我们修改了 p 指向的链表，我们同时也修改了 q 指向的链表。这样的结果并不是我们在程序 4.20 里所期待的。还有，这样的操作过程将会在我们使用一个 QUEUE 对象作为函数参数的时候同样发生。对于内置类型，我们期望在函数里得到我们自己的对象。对于基于指针的数据结构，这样的期望得到的暗示是我们需要做一个复制操作。系统不可能知道我们究竟想干什么，也不知道怎么去做——我们不得不自己提供代码。这些讨论同样适用于把它作为返回值的情况。

更坏的是，当一个 QUEUE 对象的生命周期结束时，系统收回的是指针的分配空间，而不是链表本身的分配空间。实际上，一旦指针消失，我们就再也没有办法访问链表数据的存储空间了。这是一个内存泄漏的例子，内存泄漏问题是一个严重的问题，只要我们写

分配存储的程序，这个问题就值得注意。

**程序 4.20 队列客户程序（队列模拟）**

这一客户程序模拟这样一种情况，我们指定客户随机地按照其中一个 M 服务队列等待服务，然后选择随机选择一个队列（可能是同一个），如果这个队列不为空，则提供服务（从队列中删除一个客户）。每一次我们执行这些操作，我们都把加进去的顾客、服务过的顾客以及队列的内容打印出来。

代码隐式地假设 QUEUE 是一个一级数据类型。程序 4.14 和 4.15 给出的实现不能很好地实现，因为它内部 for 循环的语义不正确。

程序 4.22 里的队列 ADT 实现有一个复制构造函数修正了这一缺陷。它在内部 for 循环中，每一次都拷贝 q 的正确队列，而且依靠析构函数来允许系统回收复制体的内存空间。

```
#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() << " ";
          }
      }
  }
```

C++提供了特别的机制来方便类的构造，只要它们有正确的复制语义和能够避免内存泄漏。特别是，我们需要提供以下的成员函数：

- 复制构造函数，来创建一个新对象，新对象是给定对象的拷贝；
- 重载赋值操作符，以便对象能够出现在赋值操作的左边；
- 析构函数，来回收对象生存期内被分配的所有存储空间。

当系统需要实现这些指定的操作时，它使用这些成员函数。如果它们没有被定义，

那系统就会用默认的函数来代替，就像我们对 Complex 所描述的那样，但如果有任何的成员函数是指针的话就会导致不正确的复制语义和内存泄漏问题。程序 4.21 是一个包括了这三个函数的 FIFO 队列接口。它们和构造函数一样，都有一个与众不同的包括类名的签名。

**程序 4.21 一级队列 ADT 接口**

为了创建一个用户定义的数据成员包括有指针的类，又要它的行为更像一个内置类型，我们需要在接口里面包括一个复制构造函数、一个重载的赋值操作符和一个析构函数。把这三个函数作为程序 4.13 的扩展。

```
template <class Item>
class QUEUE
  {
    private:
      // 与实现相关的代码
    public:
      QUEUE(int);
      QUEUE(const QUEUE&);
      QUEUE& operator=(const QUEUE&);
      ~QUEUE();
      int empty() const;
      void put(Item);
      Item get();
  };
```

如果我们在创建一个对象的时候同时给它赋一个初值，或者把对象作为函数的参数或返回值，系统就会自动地调用复制构造函数 QUEUE(const QUEUE&)。赋值表达式 QUEUE& operator=(const QUEUE&)将会在我们使用=把一个对象赋给另一个时调用。析构函数 ~QUEUE()在一个队列的存储空间被回收时调用。如果我们使用一个没有提供初始值的声明，系统就会使用构造函数 QUEUE()来创建对象 q。如果我们确实提供了另一个队列来初始化对象，用的声明就像 QUEUE <int> q=p（或者它的等价形式 QUEUE <int> q(p)），则系统使用构造函数 QUEUE(const QUEUE&)。这个函数将创建 p 的一个新拷贝，而不是 p 的另一个指针。正如通常的引用参数（reference parameter），关键字 const 表示表示我们不想改变 p，只是需要使用它来访问信息而已。

程序 4.14 实现了链表队列，而程序 4.22 则实现了它的复制构造函数、重载赋值操作符和析构函数。析构函数遍历链表，用 delete 回收每一个节点的存储空间。客户程序赋一个对象给它自身的时候，赋值操作符什么也不做，否则它会调用析构函数，然后通过 put 每一项来把等号右边的队列复制给左边的对象。复制构造函数把队列置空，然后使用赋值操作符来执行复制。

我们可以把任意的C++类转换成一级ADT，只要像程序4.22那样提供一个复制构造函数、重载赋值操作符和析构函数即可。这些函数通常基于对数据结构的直接遍历。然而，我们常常不需要这些额外的步骤，因为：

- 我们通常仅仅使用类对象的一个简单实例；
- 如果我们有多个实例，我们需要避免无意中拷贝巨量的数据结构。

简而言之，即使认识到我们能够建立一级数据类型，我们仍然需要注意权衡实现它们和使用它们之间的便利和代价，特别是它们包括有巨量的数据的时候更是如此。

这样的讨论同时解释了为什么C++里除了有一个低级的C格的字符串数据结构外，还包括一个string类。C格字符串不是一级类型，因为它们包括指针。实际上，很多程序员第一次暴露出不正确的复制语义就是在复制一个C格字符串时发生的，他们以为复制的是字符串本身，实际上复制的却只是字符串的指针。相反，C++字符串是一个一级类型，因此我们需要在操纵巨型字符串时多加注意。

**程序4.22 一级队列的链表实现**

我们可以把程序4.14里的FIFO队列类实现升级成一个一级队列，只需加上一个复制构造函数、重载赋值操作符和析构函数的实现即可。这些函数会覆盖掉系统的默认函数，并在需要复制或者析构对象时调用。

析构函数~QUEUE()调用了一个私有成员函数deletelist，这个函数遍历整个链表，对每一个节点调用delete，于是，所有与这个对象关联的内存空间都能够在其指针被收回的同时被收回。

当表达式为自赋值操作q=q时，重载赋值操作符QUEUE& operator=(const QUEUE& rhs)什么也不做。否则，它使用deletelist清空对象的旧拷贝所关联的内存空间（保留链表为空），然后通过调用put每一项来把等号右边的队列复制给左边的对象。不管哪种情况情况，返回值都是复制目标的引用。

复制构造函数QUEUE(const QUEUE&)先将链表初始化为空，然后使用重载赋值操作符来获得参数的拷贝。

```
  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(); }
```

作为另一个例子，我们可以想像修改程序 4.20，让它周期性地打印出队列的前几个元素，以便我们可以在队列变得很大时跟踪过程。但当它真的变得很大时，我们也许会逐渐地被队列的低性能所震惊，因为 for 循环里的本地变量初始化需要调用复制构造函数，而复制构造函数复制了整个队列，即使我们只需要访问其中的某一些元素而已。然后，在每一次循环反复的最后，整个队列将被析构函数析构，只是因为它是本地变量。就拿程序 4.20 来说，我们为了复制而访问每一个项，即使额外的分配和取消分配的代价对运行时间的影响是常数时间，但是它仍然是一个不可忽略的代价，尤其是当我们仅仅需要访问一些项的时候更是如此。这种情况下，我们情愿为 copy 使用默认的指针赋值（pointer-assignment）操作，以及修改 ADT 来加上允许我们无改变地访问队列里的项的操作。

内存泄漏是折磨许多大型系统的小缺点。虽然回收分配给一个对象的内存空间通常在概念上是直接的，但是要确保我们能够追踪到每一次的存储分配，这在实际上是一件棘手的任务。C++析构机制会对我们有帮助，但是系统不可能在我们需要的时候保证遍历我们的数据结构。当一个对象不再存在的时候，它的指针就永远失去了——任何被析构函数忽视的指针都是一个潜在的内存泄漏。内存泄漏的一个普遍原因是我们完全忘记了要把析构函数包括进去——当然不能指望默认的析构函数能正确地清空我们的存储。当我们使用如同程序 4.12 这样的抽象类的时候要特别注意这个问题。

一些系统已经实现了自动内存分配（automatic memory allocation），即由系统决定哪些内存空间已经不再有程序使用，然后回收它们。但是，不可能会有完全令人满意的解决方案。一些人认为，内存分配问题太重要了，以至于不能完全交给系统去做；而另一些人则认为，它太重要了，以至于不能交给程序员去做。

当我们讨论 ADT 实现的时候，问题的列表将变得越来越长，即使那些我们在本章讨论的简单 ADT 也是这样。我们需要在一个队列里面放不同数据类型的对象吗？我们需要因为我们知道性能的差异而在一个简单客户程序里面使用队列的同种数据类型的不同实现吗？实现的执行效率信息应该被包括进接口里面吗？这些信息应该是怎样的格式？这一类问题突出了如下问题的重要性，理解我们的算法和数据结构的基本特征的重要性，以及客户程序怎样有效地使用它们，在某种意义上，这是我们这本书的主题。虽然在软件工程里面，完全的实现通常代替了算法设计，但是我们应该努力保持有关问题的认识，以便我们的算法和数据结构能够成为在一个广阔范围里应用的软件工具的基础。

### 练习

▷ 4.63 重载操作符+和+=，使它们能够接受复数（程序 4.18、4.19）。

4.64 把 4.5 节里的等价关系 ADT 转换为一个一级类。

4.65 创建一个一级 ADT，使之用于处理扑克牌游戏的程序中。

●●4.66 写一个程序来经验性地测定各种扑克牌局发出来的可能性，使用练习 4.65 里定义的 ADT。

○ 4.67 为复数 ADT 开发一个实现，复数的表示基于极坐标（也就是说，形式是 $re^{i\theta}$）。

● 4.68 利用关系 $e^{i\theta}=\cos\theta+i\sin\theta$ 来证明 $e^{2\pi i}=1$，以及证明 $N$ 个 $N$ 次复单位根是：

$$\cos(\frac{2\pi k}{N})+i\sin(\frac{2\pi k}{N}),\qquad k=0,1,\cdots,N-1$$

4.69 列出 $N$ 从 2~8 的 $N$ 次复单位根。

● 4.70 使用 iostream.h 里的 precision 和 setw 来为程序 4.19 提供一个重载的<<操作符的实现，要求输出程序 4.17 里的图 4.12。

▷ 4.71 当你使用一个简单数据类型的实现，例如程序 4.14 或者程序 4.15 来运行队列模拟程序程序 4.20 时，精确描述所发生的事情。

4.72 开发一个 FIFO 队列一级 ADT 的实现，ADT 在程序 4.21 中给出，使用一个数组作为它的基本数据结构。

▷ 4.73 为一级下推栈 ADT 写一个接口。

4.74 为你在练习 4.73 里做的一级下推栈 ADT 开发一个实现，使用一个数组作为基本的数据结构。

4.75 为你在练习 4.73 里做的一级下推栈 ADT 开发一个实现，使用一个链表作为基本的数据结构。

○ 4.76 修改 4.3 节里的后缀表达式求值（postfix-evaluation）程序，对包括整系数复数在内的后缀表达式求值，使用程序 4.18 和程序 4.19 里的一级复数 ADT。为简单起见，假设所有的复数的实部和虚部都是非零的整数，各部分之间没有空格。例如，当给出以下输入时，你的程序应该输出 8+4i。

```
1+1i 0+1i + 1-2i * 3+4i +
```

●●4.77 对程序 4.20 里的队列模拟过程做一个数学的分析，当函数参数为 $N$ 和 $M$ 时，测定第 $N$ 次 get 操作时队列为空的概率，以及 for 循环中第 $N$ 次反复后队列里的项数目的期望值。

## 4.9 基于应用的 ADT 范例

作为最后一个例子，我们在本节讨论一个特定应用（application-specific）ADT，这个 ADT 代表了应用领域与我们在本书中讨论的算法及数据结构之间的关系。我们将要讨论的例子是多项式（polynomial）ADT。它来自符号数学（symbolic mathematics）这个领域，在这个领域里，我们使用计算机来帮助处理抽象数学对象。

我们的目的是能够写一个处理多项式的程序，以及做像这样的计算：

$$(1-x+\frac{x^2}{2}-\frac{x^3}{6})(1+x+x^2+x^3)=1+\frac{x^2}{2}+\frac{x^3}{3}-\frac{2x^4}{3}+\frac{x^5}{3}-\frac{x^6}{6}$$

我们也希望能够在给出 $x$ 之后对多项式求值。例如 $x$=0.5，该等式的两边都得到同样的结果 1.1328125。乘法操作、加法操作以及多项式求值，这些操作是很多数学计算的中心问题。程序 4.23 是一个提供了符号操作的简单例子，它能够处理以下的多项式等式：

$$(x+1)^2 = x^2 + 2x + 1$$
$$(x+1)^3 = x^3 + 3x^2 + 3x + 1$$
$$(x+1)^4 = x^4 + 4x^3 + 6x^2 + 4x + 1$$
$$(x+1)^5 = x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$$
...

同样基本的思想扩展到包括因式分解（composition）、积分（integration）、求导（differentiation），或者关于知识的特殊功能等等。

**程序 4.23 多项式客户（二项式系数）**

这一个客户程序使用了程序 4.24 里定义的多项式 ADT 接口来实现整系数多项式的代数处理。它从命令行获得一个整数 $N$ 和一个浮点数 $p$，计算 $(x+1)^N$，然后通过代入 $x=p$ 来验算计算结果。

```
#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;
  }
```

正如程序 4.24 里展示的接口一样，第一步是定义多项式 ADT。对于一个众所周知的数学抽象（例如多项式）来说，它的规范已经清楚到不需要再说的地步（就像我们在 4.8 节里讨论的复数 ADT 一样）：我们需要这个 ADT 的实例在行为上和那些众所周知的数学抽象精确地相似。

为了实现接口里定义的函数，我们需要选择一个特定的数据结构来表示多项式，然后实现处理数据结构的算法来提供客户程序期望 ADT 提供的行为。通常来说，数据结构的选择影响算法的潜在效率，我们将随便讨论几个。除了栈和队列，我们还能够选择使用链表表示或者数组表示。程序 4.25 是一个使用数组表示的实现；链表表示的实现留作练习（参见练习 4.78）。

想要 add 两个多项式，我们只要把它们的系数加起来就行了。如果多项式用的是数组表示，add 函数仅仅是一个简单的数组循环，我们可以在程序 4.25 里看到。想要 multiply

两个多项式，我们使用基于分配律（distributive law）的基本算法。我们用其中一个多项式的每一个项乘以另一个多项式的每一个项，按照 $x$ 的次数排列好结果，然后把同次数的项加起来就得到最后结果了。以下表格总结了对多项式$(1-x+x^2/2-x^3/6)(1+x+x^2+x^3)$的计算：

$$
\begin{array}{r}
1-x+\frac{x^2}{2}-\frac{x^3}{6} \\
+x-x^2+\frac{x^3}{2}-\frac{x^4}{6} \\
+x^2-x^3+\frac{x^4}{2}-\frac{x^5}{6} \\
+x^3-x^4+\frac{x^5}{2}-\frac{x^6}{6} \\
\hline
1+\frac{x^2}{2}+\frac{x^3}{3}-\frac{2x^4}{3}+\frac{x^5}{3}-\frac{x^6}{6}
\end{array}
$$

这个计算看上去需要用 $N^2$ 的时间复杂度来乘以两个多项式。为这个任务找一个更快的算法是一个重要的挑战。我们将在第八部分详细讨论这个主题，那时我们将看到，有可能找到时间复杂度为 $N^{3/2}$ 的分治（divide-and-conquer）算法，以及时间复杂度为 $N \lg N$ 的快速傅里叶变换（fast Fourier transform，FFT——译者注）来完成这个任务。

**程序 4.24　多项式的 ADT 接口**

这个多项式 ADT 的接口使用了模板，以便允许不同的数据类型作系数。它重载了二进制操作符+和*，这样客户程序就能够使用那些包括了这些操作符的多项式的算术表达式。构造函数如果在调用时给出了参数 $c$ 和 $N$，那么就创建一个多项式 $cx^N$。

```
template <class Number>
class POLY
  {
    private:
      // 与实现相关的代码
    public:
      POLY<Number>(Number, int);
      float eval(float) const;
      friend POLY operator+(POLY &, POLY &);
      friend POLY operator*(POLY &, POLY &);
  };
```

程序 4.25 里的求值函数的实现使用了一个传统的有效算法，名叫 *Horner* 算法。这一函数的一个直观的实现包括了一个用于计算 $x^N$ 的函数。这种方法需要平方时间。一种不那么直观的实现需要把中间结果 $x^i$ 存储在一个表里面，然后在下回需要计算的时候

直接读取就行了（这其实是动态规划的方法——译者注）。这种方法需要线性的额外空间。*Horner* 算法是一种直接优化的线性算法，它基于加括号操作（parenthesization），如下所示：

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = (((a_4x + a_3)x + a_2)x + a_1)x + a_0$$

*Horner* 的方法常常被认为是节约时间的小技巧，但它又的确是一个早期的漂亮而且高效的算法的突出例子．他把这一基本计算任务的时间花费从二次减少到了线性。我们在程序 4.5 里实现的转换 ASCII 字符串到整数的计算是 *Horner* 算法的一个版本。在第十四章和第五部分，我们将再次遇到 *Horner* 算法，作为一个与特定的符号表以及字符串检索实现相关联的重要计算的基础。

**程序 4.25 多项式 ADT 的数组实现**

在这个多项式 ADT 的数组实现中，数据的表示包括了度（degree）和一个指向系数数组的指针。它不是一个一级 ADT：客户程序必须注意内存泄漏和指针赋值复制语义这些问题（参见练习 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;
        }
};
```

重载的+和*操作符构造了新的多项式来保留它们的结果，因此这个实现有内存泄漏问题。我们可以轻易地通过实现一个复制构造函数、一个重载赋值操作符以及一个析构函数来摆脱这个缺陷。当我们处理很大的多项式或者处理大量的小多项式，或者建立一个在应用系统中使用的 ADT 时就会这样做（参见练习 4.79）。

按惯例来说，用数组表示来实现多项式 ADT 只是一种可能性而已。如果指数很大，而且项不多，链表表示也许更适合一些。例如，我们不会想用程序 4.25 来处理如下的一个乘法操作：

$$(1 + x^{1000000})(1 + x^{2000000}) = 1 + x^{1000000} + x^{2000000} + x^{3000000}$$

因为它将使用的数组将会有近百万个未曾使用的空间浪费。练习 4.78 详细讨论了链表实现的方法。

## 练习

○ 4.78　提供一个程序 4.24 里给出的多项式 ADT 的实现，使用链表作为基本数据结构。你的链表不能含有任何系数为 0 的项的节点。

○ 4.79　通过加上一个复制构造函数、一个重载赋值操作符以及一个析构函数，解决程序 4.25 里的内存泄漏问题。

▷ 4.80　在程序 4.25 定义的多项式 ADT 中加上重载的操作符+=和*=。

○ 4.81　扩展程序中给定的多项式 ADT，使之包括多项式积分和多项式求导操作。

4.82　修改你的练习 4.81 的多项式 ADT，使之忽略所有指数大于等于整数 $M$ 的项，$M$ 由客户初始化的时候提供。

●●4.83　修改你的练习 4.81 的多项式 ADT，使之包括多项式的因式分解操作。

● 4.84　开发一个 ADT，允许客户程序对任意长的整数执行加法和乘法操作。

● 4.85　修改 4.3 节里的后缀表达式求值程序，使之能够对任意长的整数组成的后缀表达式求值，使用你在练习 4.84 里开发的 ADT。

●●4.86　写一个使用你在练习 4.83 开发的多项式 ADT 的客户程序来计算积分。使用泰勒展开式（Taylor series）近似函数，符号化地处理它们。

4.87　开发一个 ADT 提供给客户程序，使它们能够处理浮点数的算术运算。

4.88　开发一个 ADT 提供给客户程序，使它们能够处理抽象对象矩阵的算术运算：加法、减法、乘

法以及已定义的除法。

4.89 为一个字符串 ADT 写一个接口，该 ADT 包括创建字符串、比较两个字符串、串接两个字符串、拷贝一个字符串到另一个字符串中以及返回字符串长度等操作。注意：你的接口将会和 C++标准库里提供的接口很相似。

4.90 为你的练习 4.89 中的字符串 ADT 接口提供一个实现，在适当的时候使用 C++的 string 库。

4.91 为你的练习 4.89 中的字符串 ADT 接口提供一个实现，使用一个链表作为基本的表示方式。分析每一种情况的最坏情况时间复杂度。

4.92 为一个索引集（index set）ADT 写一个接口和一个实现，处理 0~$M$-1 的整数集合（$M$ 是已定义的常数），包括以下操作：创建一个集合、计算两个集合的并集、计算两个集合的交集、计算两个集合的差集，以及打印出一个集合的内容。在你的实现里面使用一个长度为 $M$-1 的 0-1 数组来表示每一个集合。

4.93 写一个客户程序测试你的练习 4.92 中的 ADT。

## 4.10 前景展望

我们有三个主要理由，来让我们在从事算法及数据结构的学习时注意基本 ADT 的基本概念：

- ADT 是一个广泛应用的重要的软件工程工具，许多我们学习的算法都是一些广泛适用的基本 ADT 的实现；
- ADT 帮助我们封装所开发的算法，这样我们就可以在许多不同的地方重用这些代码；
- ADT 提供了一个方便的机制，在我们开发和比较算法性能的过程中提供给我们使用。

理想地，ADT 具体表述了我们被迫精确地描述我们操纵数据的方法的通用原则。我们在这一章里详细讨论的客户-接口-实现机制对 C++的这个任务来说是很方便的，而且它以 C++代码提供给我们一些有趣的特性。许多现代编程语言有着特别的支持来允许以相似的特性来开发程序，但是通用的方法超出了特定编程语言的范围——当我们没有指定语言支持的话，我们采用传统编程习惯来保持客户、接口和实现间的区别。

当我们讨论一个随时扩展的关于指定我们的 ADT 的行为的选择的问题集时，我们将会面对一个随时扩展的关于提供有效实现的挑战的问题集合。我们讨论的大量例子展示了迎接这些挑战的方法。我们持续不断地努力，以达到有效地实现所有操作的目标，但是我们不可能有一个通用的实现能够为所有的操作集合达到这个目标。这种状况与以下原则相抵触，ADT 的实现需要知道客户程序的细节，以便决定哪种实现才会使客户程序最有效地运行；而客户程序的实现这又需要知道各种不同实现间性能的特性，以便决定为特定应用选择哪种实现。这一原则完全以 ADT 为第一位的元素。一如既往，我们必须在其中找到平衡点。在这本书里，我们讨论了大量的方法来实现不同的基本 ADT，所有这些都有很重要的应用。

我们可以根据一个已经写好的 ADT 来建立另外一个 ADT。我们使用 C++提供的指针和结构抽象来建立链表，然后使用链表或者 C++提供的数组抽象来建立下推栈，接着使用下推栈来获得计算算术表达式的能力。ADT 的概念允许我们在不同的抽象层次上构造大型的系统，从计算机本身提供的机器语言指令到大量编程语言提供的能力，完成排序、搜索等各种操作，并且提供在这本书的第三部分和第四部分讨论的算法所提供的高级能力，或者是各种应用所要求的更高级的抽象，正如我们在第五部分到第八部分所讨论的那些。ADT 处在开发更有用的抽象机制的关键点上，而这一机制又是使用计算机有效地解决问题的基本所在。

# 第五章

# 递归与树

递归（recursion）是数学与计算机科学中的基础概念。递归程序可被简单定义为在编程语言的一个自我调用的程序（类似于数学中的递归函数是根据自身定义的）。递归程序不能总是自我调用，否则会无法终止（类似于递归函数不能总是由自身定义，否则定义就会无限循环）。因此第二个基本要素就是，必须存在一个终止条件，让程序得以停止对自身的调用（就如数学函数不再根据自身来定义）。所有的实际计算都可以借助递归的框架表达出来。

递归与被称为树（tree）的以递归方式定义的结构的研究是互相重合的。我们使用树既有助于理解和分析递归程序，也可以解释清楚这些数据结构。我们在第一章已经碰到过树的应用程序（尽管那不是递归形式的）。递归与树之间的联系是本书的实质基础。我们使用树来理解递归程序，同时也用递归程序来创建树，还利用二者的基础联系及循环互动来分析算法。递归有助于我们开发出精练高效的数据结构和算法，并应用于各种程序中。

这一章的基本目的是从实用工具的角度考察递归程序与数据结构。首先，我们讨论数学循环与简单递归程序之间的关系，并考察大量的实际递归程序。然后，我们将介绍基本递归方案，即分治法（divide and conquer），并会用它来解决在后续几章出现的基本问题。接着，介绍一种被称为动态编程（dynamic programming）的实现递归程序的一般性办法，它能为广泛的问题提供精湛有效的解决办法。接着，我们研究树及其数学属性、还有与之相关的具体算法，包括为树遍历（tree traversal）（它是递归树运行程序的基础）提供的基本方法。最后，我们再进一步考察与处理图（graph）相关的算法，并将特别关注一种基本的递归程序——深度优先搜索（depth-first search），它是许多处理图的算法的基础。

正如我们将看到的，许多有趣的算法就是用递归程序来表述的，许多算法设计者也喜欢以递归方式表达各种方法。我们同样详尽讨论非递归的方式，不只因为我们常常能简单设置那些基于栈的本质上与递归等价的算法，而且我们也发现这些非递归的方式同样能通过一系列不同的运算获得相等的结果。因此递归的公式模型为我们寻求其他可选方案提供了一整套框架。

关于递归和树的讨论完全可以写成一本书，因为这里的讨论都是源自大量的计算机科学领域之内和之外的应用程序。实际上，这本书就是关于递归和树的全面讨论，从基础部

分开始，出现在书里的每一章。

## 5.1 递归算法

递归算法（recursive algorithm）就是通过解决同一问题的一个或多个小的实例来最终解决一个大问题的算法。我们常使用递归函数（即能调用自身的函数）在 C++中实现递归算法。C++中的递归函数相当于数学函数的递归定义。我们研究递归就从检验直接求值数学函数的程序开始，并从它的基本机制引伸到一种通用的程序设计范型。

递推关系（recurrence relation)（参见 2.5 节）是递归定义的函数。一个递推关系定义一个函数，该函数的定义域是非负整数，可以赋初始值或以更小整数的递归方式实现。或许你最熟悉的函数是阶乘函数，它由以下递推关系定义：

$$N! = N \cdot (N-1)! \quad N \geq 1 \text{ 且 } 0! = 1$$

该定义直接与程序 5.1 中的 C++递归函数对应。

**程序 5.1 阶乘函数（递归实现）**

这个递归函数使用标准的递归定义计算函数 $N!$。当以非负 $N$ 调用并且 $N!$足够小到可以表示为 int 类型值时，它返回正确值。

```
int factorial(int N)
  {
    if (N == 0) return 1;
    return N*factorial(N-1);
  }
```

程序 5.1 等价于一个简单的循环。例如，以下的 for 循环实现了相同的计算：

```
for ( t = 1, i = 1; i <= N; i++) t *= i
```

由此可见，把递归程序转换成完成同样计算的非递归程序常常是可能的，反之，我们可以使用递归而不使用循环来表示任何涉及循环的计算。

因为递归使我们可以用紧凑的形式表达复杂算法，且不牺牲效率，所以我们使用递归。比如说，阶乘函数的递归实现避免了对局部变量的需要。递归实现的花费产生于程序设计系统中的机制，这种机制支持函数调用，而函数使用的是内置下推栈的等价物。大多数当代的程序设计系统为了这一任务都精心设置机制。除了这个优势，我们还会看到，写出一个极其没有效率的递归函数是很容易的，并且我们需要费尽心机去避免难以驾驭的实现。

程序 5.1 显示出一个递归程序的基本特征：它调用自身，具有终止条件，在这个条件直接计算其结果。我们可以使用数学归纳法来验证这一程的工作过程：

- 它计算 0!（基）。
- 假设当 $k<N$（归纳假设）时，它计算 $k!$，在这一假设下它计算 $N!$。

以此推论，就为我们开发解决复杂问题的算法提供了一条便捷的道路。

在 C++这样的程序语言中，很少有对我们所写程序的种类限制，但是我们仍努力限制自己使用前面几段所列出的包括正确归纳证明的递归函数。虽然我们不考虑书本中出现的正确性证明，我们还是很喜欢把复杂的程序组合起来去处理较难的任务，而且我们也需要确凿地保证能够正确完成任务。诸如递归函数这样的机制能够为我们提供确凿的保证，同时给予我们紧凑的实现方式。就实际来说，与数学归纳法的联系让我们知道，我们的递归函数必须满足以下两个基本属性：

- 它们必须清晰无误地解决基的情况。
- 每一个递归的调用，必须包括更小的参数值。

这些要点都是含糊不清的——它对于我们所写的每一个递归函数，都必须有一个有效归纳证明。这些要点在我们开发实现的时候能提供有用的指导。

```
puzzle(3)
  puzzle(10)
    puzzle(5)
      puzzle(16)
        puzzle(8)
          puzzle(4)
            puzzle(2)
              puzzle(1)
```

**图 5.1 递归调用链的例子**

嵌套的函数调用序列最终停止，但我们不能证明程序 5.2 的递归函数没有为一些参数提供任意深的嵌套。我们推荐总是用更小的参数进行自我调用的递归程序。

程序 5.2 是一个有趣的例子，它显示了对归纳参数的需要。这是一个递归函数，它违反了这样的规则：每一个递归调用必须包括更小的参数值，因此我们不能用数学归纳法来理解它。实际上，如果没有 $N$ 的范围，我们就无从知道这种计算对于每个 $N$ 是否会终止。对于能表示为 int 的小整数，我们能检验这一程序的终止条件（参见图 5.1 与练习 5.4），但对于大整数（比如 64 位的字），我们无从得知这个程序是否会进入无限循环当中。

**程序 5.2 有问题的递归程序**

如果参数 $N$ 是奇数，这个函数用 $3N+1$ 作为参数调用本身；如果 $N$ 是偶数，函数用 $N/2$ 作为参数调用本身。我们不能使用归纳法证明该程序终止，因为并非每一个递归调用都使用一个比上面所给的更小的参数。

```
int puzzle(int N)
  {
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
  }
```

**程序 5.3 欧几里得算法**

作为有 2000 年历史的最老的著名算法之一，这是一个找出两个整数的最大公约数的递归方法。

```
int gcd(int m, int n)
  {
```

```
    if (n == 0) return m;
    return gcd(n, m % n);
  }
```

程序 5.3 是欧几里得算法的紧凑的实现，用于找出两个整数的最大公约数。它是基于这样一种观察：两个整数 $x$ 和 $y$（$x>y$）的最大公约数等同于 $y$ 与 $x \bmod y$（$x$ 除以 $y$ 的余数）的最大公约数。$t$ 除以 $x$ 和 $y$ 等价于 $t$ 除以 $y$ 和 $x \bmod y$，因为 $x$ 等同于 $x \bmod y$ 加上一个 $y$ 的倍数。图 5.2 是该程序调用的示例。对欧几里得算法而言，递归的深度由参数的算术属性（即参数的对数）决定。

```
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)
```

**图 5.2　欧几里得算法的例子**

这一嵌套的函数调用顺序说明了欧几里得算法的操作，它发现 314159 与 271828 是互质的。

程序 5.4 是一个带有多重递归调用的例子，它是另外一个表达式求值程序，实质上处理着与程序 4.2 相同的计算，但它是前缀（而非后缀）表达式，并且让递归代替显式的下推栈。在本章中，我们将看到许多递归程序的其他例子，以及使用下推栈的等价程序。我们将就几对这样的程序详细检验其特定的关系。

**程序 5.4　递归程序对前缀表达式求值**

为了对前缀表达式求值，我们将一个数字的 ASCII 码转换为二进制值（在 while 循环结尾），或者进行由在两个操作数上的表达式的第一字符指示的操作，并以递归形式求值。该函数是递归的，但使用全局数组，包含该表达式与在表达式中当前字符的索引。该索引经过每一个求值的子表达式然后被推进。

```
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;
  }
```

图 5.3 显示出程序 5.4 在样本前缀表达式中的操作。该多重递归调用掩饰了一系列的复杂计算。类似于许多递归程序，该程序可以从归纳角度来很好地理解：假设它对于简单表

达式能正确工作，我们就可以确信它对于复杂表达式也能很好地工作。该程序是一个简单的递归渐下降语法分析程序（recursive descent parser）的例子，我们可以使用同样的过程将C++程序转换成机器码。

写出程序 5.4 正确求出表达式的值的精确的归纳证明理所当然比用我们一直在讨论的整数参数的函数证明更具挑战性，我们在整本书中也将继续碰到比这一程序更复杂的递归程序与数据结构。为每一个我们所写的递归程序提供正确性的完整归纳证明，并不是我们所要追求的理想目标。在这种情况下，程序“懂得”如何分离出与给定的操作符对应的操作数首先是不可思议的（或许因为在高级别上我们根本不能马上看到如何进行这个区分），但实际上的确是一个直截了当的计算（因为要处理每一个函数调用毫无疑问是由表达式中的第一个字符串决定的）。

```
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
```

**图 5.3 前缀表达式求值的例子**

这个嵌套的函数调用顺序，显示出递归的前缀表达式求值算法在样本表达式上的操作。简而言之，这里显示了表达式参数。算法本身并不明确地决定参数串的长度，它从字符串的开头取得所需。

原则上，我们可以用许多等价的递归程序来代替 for 循环。与 for 循环相比，递归程序常常是更为自然的表达式计算的方式，所以我们要充分利用那些支持递归的程序设计系统所提供的机制优势。但我们必须明白一个潜藏的代价。当我们实现一个递归程序时，我们正在嵌套函数调用，直到我们达到某一点，在这一点上我们不再进行递归调用，并且返回，我们从图 5.1 到图 5.3 考察的例子能明显地看到这一点。在大多数编程环境中，使用等价的内置栈来实现这样一个嵌套函数调用。我们将通过这一章来检查这种实现的本质。递归的深度就是在整个计算中函数调用的嵌套的最大程度。通常，深度取决于输入。比如说，在图 5.2 与图 5.3 的例子中，递归的深度分别是 9 和 4。在使用一个递归程序的时候，我们需要考虑到：这个编程环境必须能够保持一个其大小与递归深度成比例的下推栈。对于大型的问题，栈所需的空间可能妨碍我们使用递归的解决办法。

由带有指针的节点建立的这种数据结构是递归的。比如说，我们对于第三章中链表的定义就是递归的。因此，递归的程序提供许多普遍使用函数的自然实现，用于支配这样的数据结构。程序 5.5 包含了 4 个例子。我们在本书中经常使用这样的实现，主要是因为它们比非递归程序更容易理解。然而，当运行大型的链表时，我们必须对使用诸如程序 5.5 这样的程序加以注意，因为那些函数的递归深度能够与链表成正比，因此递归栈要求的空间可能不被允许。

**程序 5.5 对于链表的递归函数例子**

这些关于简单链表处理任务的递归函数很容易表达，但对于大型链表未必有用，因为递归的深度要与链表的深度成正比。

第一个函数 count 计算链表中的节点数。第二个函数 traverse 从头到尾为每一个链表上的节点调用函数 visit。这两个函数都很容易用 for 或 while 循环来实现。第三个函数 traverseR 并无一个简单的循环可以与之对应。它为链表中每一个节点调用函数 visit，但以相反的顺序进行。

第四个函数 remove 从链表中消除所有带给定值的节点。该实现的关键是在每一个要被删除的节点的前驱中能改变 x=x->next 的这样一种链接，由于使用引用参数使实现得以成为可能。为每一个 while 循环的重复运行所做的结构性改变与图 3.3 中的图相同，但这里的 x 和 t 都指向同一节点。

```
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);
  }
```

当函数的最后的操作实现的是递归调用的时候，一些程序环境会自动检测与消除尾递归（tail recursion），严格地增加这类递归的长度并不是必要的。这种改进将有效地转换程序 5.5 中的 count、traversal 和 removal 函数，成为循环（loop），但它并不运用于反向遍历函数。

在 5.2 节和 5.3 节中，我们考察了两个熟悉的递归算法，它们表示基本的计算范型。然后，在 5.4 节到 5.7 节，我们考察了递归数据结构，它是我们考虑的绝大部分算法的基础。

## 练习

▷ 5.1　编写一个递归程序来计算 $\lg(N!)$。

5.2 改进程序 5.1 来计算 $N!$ mod $M$，以使溢出不再是问题。尝试对 $M$=997，$N$=$10^3$，$N$=$10^4$，$N$=$10^5$ 和 $N$=$10^6$ 运行你的程序，并由此理解你的程序设计系统如何处理深嵌套的递归调用。

▷ 5.3 对于每一个 1~9 之间的整数，当程序 5.2 被调用时，给出参数值序列。

● 5.4 对程序 5.2 导致的最大递归调用数目，找出 $N$<$10^6$ 的值。

▷ 5.5 给出欧几里得算法的非递归实现。

▷ 5.6 输入 89 和 55，运行欧几里得算法，根据其结果给出对应于图 5.2 的图。

○ 5.7 当输入值是两个逻辑顺序的斐波纳契数（$F_N$ 和 $F_{N+1}$）时，给出欧几里得算法的递归深度。

▷ 5.8 当输入+ * * 12 12 12 144 时，就递归前缀表达式求值结果给出与图 5.2 相对应的图。

5.9 写出一个对后缀表达式求值的递归程序。

5.10 写出一个递归程序求中缀表达式。可以假定操作数被置于圆括号中。

○ 5.11 写出一个递归程序，把中缀表达式变为后缀表达式。

○ 5.12 写出一个递归程序，把后缀表达式变为中缀表达式。

5.13 为 Josephus 问题编写一个递归程序（参见 3.3 节）。

5.14 写出一个递归程序，删除一个链表的最后节点。

○ 5.15 写出一个递归程序，反转链表中节点的顺序（见程序 3.7）。提示：使用全局变量。

## 5.2 分治法

我们在书中所考虑的许多递归程序，都使用两个递归调用，每一个调用处理输入的一半信息。这种递归模式可能就是著名的针对算法设计的“分治法”（divide-and-conquer）范型的最重要的例子，这个范型是我们的许多重要算法的基础。

作为一个例子，我们可以考察这样一种任务：在存放于数组 a[0]，…，a[$N$-1]的 $N$ 个项里，找出最大的一项。我们可以轻而易举地完成这个任务，只要遍历数组一遍即可。

```
for (t = a[0], i = 1; i < N; i++)
  if (a[i] > t) t = a[i];
```

在程序 5.6 中给出的递归式的分治法也是一个简单的（但完全不同的）算法，可以解决同样的问题；我们使用这个办法来阐明分治法的概念。

**程序 5.6 使用分治法求最大元素**

这个函数将数组 a[l]，…，a[r]分成 a[l]，…，a[m]和 a[m+1]，…，a[r]两部分，分别求出每一部分的最大元素（递归地），并返回较大的那一个作为整个数组的最大元素。它假设 Item 是定义了>的一级类型。如果数组大小是偶数的，两部分相等；如果是奇数的，两部分大小差 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;
  }
```

我们使用这个分治法通常是因为相对于那些使用简单迭代算法的方式，它能够提供更加快速的解决方式（我们将在本节的末尾集中讨论几个例子），同时分治法作为一种理解基础计算的本质的途径，也是值得我们深入考察的。

图 5.4 显示了对于一个样本数组调用调用程序 5.6 时的递归调用。这个根本的结构看起来很复杂，但是一般不需要担心，因为我们依靠一种归纳法证明这个程序如何运行，同时我们使用一个递推关系来分析这个程序的实现过程。

通常来说，代码本身能够通过一种实现所要计算的归纳来显示证据：

- 它简明快速地为大小为 1 的数组找出最大的元素。
- 对于 $N>1$，它把一个数组分割成两个大小比 $N$ 小的数组，使用归纳法假设找到两部分中的最大元素，然后返回两个值中较大的一个，这个值必定是整个数组中的最大值。

更进一步，我们可以使用这个程序的递归结构来了解其性能特征。

**性质 5.1** *把一个大小为 $N$ 的问题分解为两个独立的（非空）它可以递归求解的部分的递归函数调用自身少于 $N$ 次。*

如果所分的两个部分的大小分别是 $k$ 和 $N-k$，那么我们使用的递归函数调用的总数目就是：

$$T_N=T_k+T_{N-k}+1, \quad N\geq 1, \qquad T_1=0$$

通过归纳立刻得出 $T_N=N-1$。如果总的大小加起来的值小于 $N$，调用的次数之和也小于 $N-1$，其证明与先前的归纳证明一样。我们可以在通常的情况下得到类似结果（参见练习 5.20）。■

程序 5.6 是许多分治法算法的代表，它精确地具有同样的递归结构，但是其他的例子可能在两个基本的方面尚有不同。首先，程序 5.6 在每一次函数调用上工作量相同，所以它的总运行时间是线性的。其他分治法算法在每一次函数调用上进行更多的工作，就如我们将看到的，所以决定总的运行时间需要更复杂的分析。这种算法的运行时间依靠于分解为部分的精确分析。第二，程序 5.6 就是分治法的算法，在这种算法中部分总和构成总体。其他的分治法算法可以分解

```
 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)
```

**图 5.4 查找最大值的递归算法**

这个函数调用的序列说明了用递归算法计算最大值的动态过程。

为更小的部分，而这些部分之和小于整个问题，或者将总和大于整个问题的那些部分加以重叠。这些算法仍然是正确的递归算法，因为每一个部分都比整个问题要小，但分析这些算法比分析程序 5.6 要困难得多。我们将会详细地考虑不同类型的算法的分析。

例如，我们在 2.6 节中讨论的折半查找算法是个分治法算法，该算法把一个问题分成对半，并且只运算其中的一半。我们将在第十二章检验折半查找的一个递归的实现。

图 5.5 显示出由编程环境所保持的内部栈的内容来支持图 5.4 中的计算。图 5.5 所示的模型是理想化的，但它有助于我们对分治法计算结构的深入理解。如果一个程序有两个递归调用，当执行第一个函数（函数包含了参数值、局部变量和返回地址）时，实际的内部栈就包含了与该函数调用相符的一个项，接着执行第二函数，就有一个与第二函数调用相符的相似项。可选的办法就如图 5.5 所描述的那样，把两个项一次放置到栈上，使得所有的子任务以显式的方式在栈上完成。这个安排浅显地描绘了计算，并且为更一般的计算模式做好铺垫，就如我们在 5.6 节和 5.8 节所见的那样。

图 5.6 描述了找出最大值的分治法的计算结构。它是一个递归的结构：在顶端的节点包含了输入数组的大小，用于左侧子数组的结构画在了左方，用于右侧子数组的结构画在了右方。我们将正式地定义并讨论 5.4 节和 5.5 节中的树结构。这些结构对于理解任何包括了嵌套函数调用（特别是递归调用）的程序都有很好的作用。在图 5.6 中显示的是同一棵树，不过它带有与函数调用相符的以返回值为标号的节点。在 5.7 节，我们将考察建造一个显式的链接结构的过程，该结构表示类似与此类似的树。

要是没有古老的汉诺塔问题，关于递归的讨论将是不完整的。我们有三个立柱和适合这些立柱的 $N$ 个盘子。这些盘子在大小上有所不同，一开始都是安排在一个立柱上，顺序为底下的盘子最大顶端的盘子最小。任务就是要移动这些盘子到右边的一个位置（立柱）上，遵守以下的规则：（*i*）一次只能移动一个盘子；（*ii*）没有盘子可以放置到一个更小盘子的上端。有一个传说，当一群僧侣在寺庙里面，用 40 个金盘子和三个钻石立柱完成这个项时，世界末日就到了。

程序 5.7 就这个问题给出了一个递归的解决办法，它详细说明了在每一个步骤哪个盘子将会被移动，向哪个方向（+意味着移动一个盘子到右边立柱，若在最右边立柱上，就转到最左边的立柱；-也就是移动一个盘子到左边立柱，若在最左边立柱上，就转到最右边的立柱）。这样的递归是基于

```
 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
```

**图 5.5 内部栈动态**

该序列是在图 5.4 计算中内部栈内容的一个理想化表述。我们从栈的整个子数组的左边和右边的索引开始。每一行描述了弹出两个索引的结果，如果二者不相等，就压入四个索引，当弹出的子数组被分为两部分后，这些索引就为左边的子数组和右边的子数组定下界限。在实际运用中，该系统保留了栈上的返回地址和局部变量，而代替专门的对将完成工作的表述的是足够描述计算的模型。

以下的想法：移动一个立柱上的 $N$ 个盘子到右边，我们首先移动最上面的 $N-1$ 个盘子到左边，然后移动第 $N$ 个盘子到右边，再移动其他 $N-1$ 个盘子到左边（在盘子 $N$ 之上）。我们可以用归纳法来检验这种解决方案。图 5.7 显示当 $N=5$ 时的移动，以及当 $N=3$ 时的递归调用。我们现在详尽考察的这样一个样式是显而易见的。

首先，这个解决方案的递归结构能够立即告诉我们这个解决办法所需的移动数目。

**性质 5.2** *汉诺塔问题的递归分治法算法产生出一个有“$2^N-1$”步移动的解决方案。*

通常，若移动的数目能满足一个递推，就可以直接得出这个性质。在这种情况下，满足盘子移动数目的递推就与公式 2.5 相似：

$$T_N=2T_{N-1}+1, \quad N\geq 2,\ T_1=1$$

我们可以用归纳法直接检验所示的结果：$T(1)=2^1-1=1$；并且，如果 $T(k)=2^k-1$，$k<N$，那么 $T(N)=2(2^{N-1}-1)+1=2^N-1$。■

如果僧侣们每秒移动一个盘子，假设不犯一个错误，至少也要花 34800 年才能完成（参见图 2.1）。世界的末日看起来要远得多，因为这群僧侣无法想像他们使用程序 5.7 的好处，也无法迅速算出下一步该移动哪一个。我们现在考察一种方法的分析，这种方法导致了一种很容易做出正确决定的简单的非递归方法。它与大量重要的实际算法相关，我们当然不想让这些僧侣知道这个秘密。

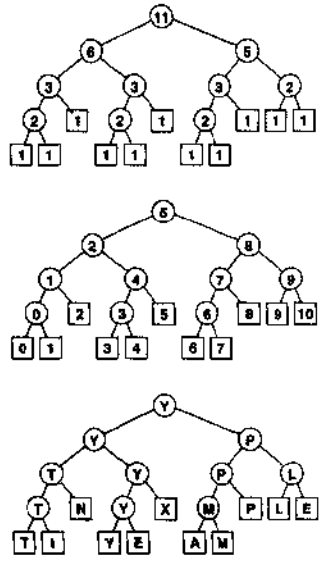


**图 5.6 找出最大值算法的递归结构**

分治法将一个大小为 11 个元素的问题分成一个大小为 6 和一个大小为 5 的问题，再将大小为 6 的问题分成两个大小为 3 的问题，如此分割下去，直到分割成大小为 1 的问题（最上面的图）。图中的每个圆表示对递归函数的一次调用，其作用的对象是其下连接的节点（正方形是递归终止的调用）。中间的图显示了我们用来影响分解位置的文件中的索引值，最下面的图显示了返回值。

**程序 5.7 汉诺塔的解决方案**

我们把盘子移到右边的方案是，（递归地）将除了最下面的盘子之外的所有盘子移到左边，然后将最下面的盘子移到右边，然后（递归地）再将其他盘子移回到最下面的盘子上面。

```
void hanoi(int N, int d)
  {
    if (N == 0) return;
    hanoi(N-1, -d);
    shift(N, d);
    hanoi(N-1, -d);
  }
```

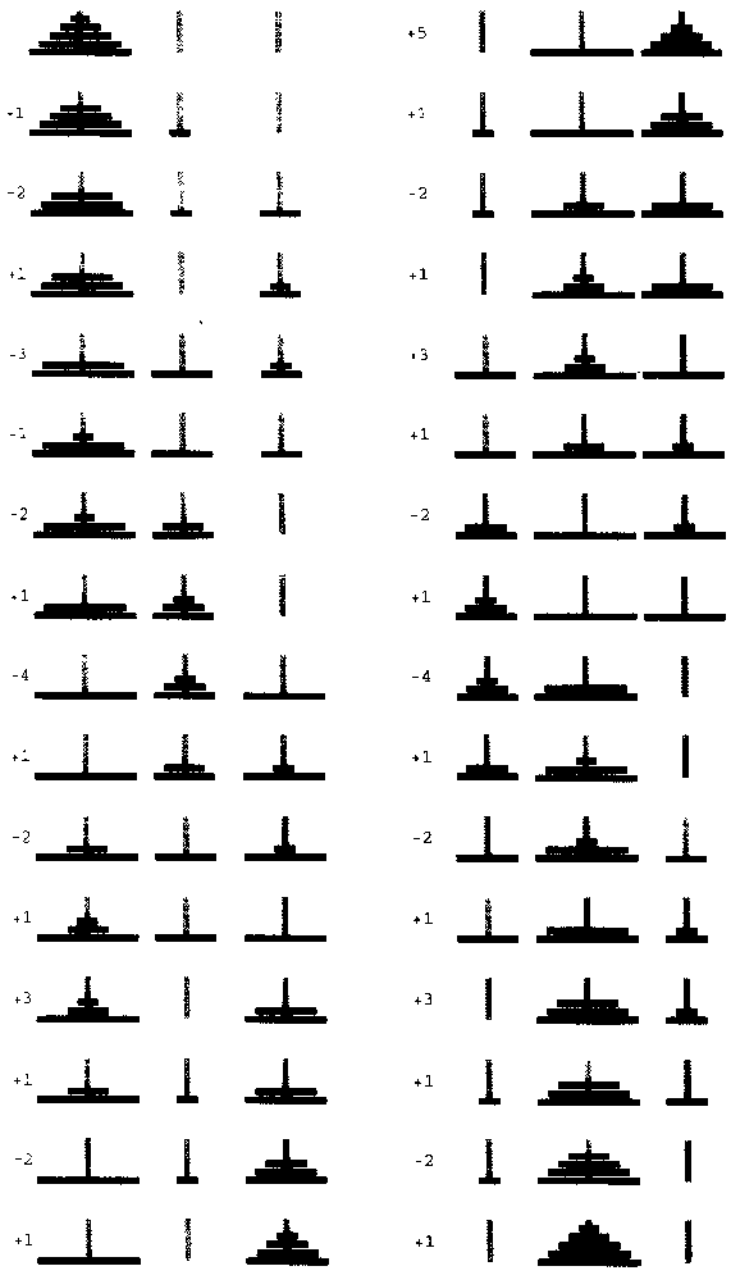



**图 5.7 汉诺塔**

该图描述了具有五个盘子的汉诺塔问题的解决方案。首先移动左边的位置（左列）上的上面 4 个盘子，然后将第 5 个盘子移到右边的柱子，然后移动左边位置（右列）上的上面 4 个盘子。图右半部分的函数调用序列构成了一个盘子的计算序列。移动的计算序列是+1 -2 +1 +3 +1 -2 +1，该序列在该解决方案中出现了四次（例如，前七次移动）。

要理解汉诺塔的解决方案，让我们考虑这样一个简单的任务：在一把尺子上画出一系列的刻度线。每一英寸在 1/2 英寸的刻度上有一个线，1/4 英寸的线短一些，1/8 英寸的线再短一些，如此继续。我们的任务就是编写一个程序，以任意给定的细分程度画出这些线，假设我们已有一个过程 mark(x，h)，它的作用就是在位置处画一个 h 单位的线。

如果需要的细分程度是 $1/2^n$ 英寸，我们重新调整程序，以使我们的任务变为从 0 到 $2^n$ 之间在每一刻度上放一个线，不包括端点。从而，中间的线应该是 $n$ 个单位的高度，在左半部分和右半部分的中间的线应该是 $n$-1 个单位的高度，以此类推。程序 5.8 是直截了当的分治法算法，能够完成这一目标；图 5.8 显示了对于一个小示例的操作。就递归而言，潜藏在该方法背后的意图如下。要在任意间隔上画线，我们首先就该间隔分成同等的两部分。然后，我们在左半部分画出短线（递归地），在中点画出长线，在右半部分画出短线（递归地）。图 5.8 阐明了按顺序画线的方法，从左到右——技巧在于计算长度。图中的递归树有助我们理解计算过程：纵向读下去，我们可见对于每次递归函数调用，线的长度递减 1；横向读过去，我们按它们被画的顺序得到这些线，因为，对于任何给定的节点，我们首先标出在左方与函数调用有关的线，再画出与节点有关的线，最后是标出在右方与函数调用有关的线。

我们马上看到，这些长度的顺序精确地与在汉诺塔问题中被移动的盘子的顺序相同。实际上，要证明它们是同一的，简单的证据就是它们的递归程序相同。换种方式，我们那些僧侣也能在一把尺子上使用这些线以决定该移动哪一个盘子。

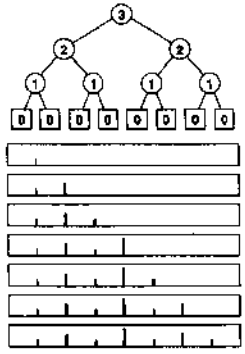
更进一步，程序 5.7 中的汉诺塔解决方案与程序 5.8 的刻度尺画线程序是被程序 5.6 所例证的基本的分治法设计的不同变体。三者都通过把大小为 $2^n$ 的问题分解为 $2^{n-1}$ 的两个问题。要找出最大值，我们可用大小为输入值的线性时间解决办法；要划出一把尺子和解决汉诺塔问题，我们可用大小为输出值的线性时间解决办法。对于汉诺塔，我们一般把该解决方案设想为指数时间的，原因在于我们根据盘子的数目 $n$ 来衡量该问题的大小。

**程序 5.8 分治法画刻度线**

要在尺子上画刻度线，我们首先在左半边画刻度线，然后在中间画一条最长的刻度线，最后在右半边画刻度线。该程序将应用 r-1 等于 2 的幂次方的属性，该属性体现在它的递归调用中（参见练习 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);
      }
  }
```

使用递归程序在尺子上画线并不难，但是否还有更简单的方式计算第 $i$ 个线的长度，而无论给定的 $i$ 是多少？图 5.9 显示了为这一问题提供了答案的另一个简单的计算过程。由汉诺塔程序与尺子程序打印出的第 $i$ 个数目，不过是在 $i$ 的二进制表示中尾缀 0 的位数。我们可根据分治法公式，针对打印 $n$ 位数字的表，用归纳法证明这个性质：打印（$n$-1）位数字的表，前缀都为 0；然后打印（$n$-1）位数字的表，每一个前缀都为 1（参见练习 5.25）。



```
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)
```

**图 5.8 画线函数的调用**

该函数调用序列是为长度为 8 的尺子画刻度的计算序列，结果在长度 1，2，1，3，1，2 和 1 处画了刻度。

```
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
```

**图 5.9 二进制计算与刻度尺函数**

计算刻度尺函数相当于计算在偶数 $N$ 位数字中尾缀 0 的个数。

对于汉诺塔问题，与 $n$ 位数字对应的涵义是该任务的一个简单算法。我们可以将一个立柱的所有盘子移到右边，方法是重复以下的两步直到完成：

- 如果 $n$ 是奇数，把小的盘子移动到右边（如果 $n$ 是偶数，移动到左边）。
- 惟一合法的移动不涉及小盘子。

这就是说，在我们移动小盘子之后，其他两个立柱保持了两个盘子，一个比另一个更小。不涉及小盘子的惟一的合法移动就是要移动一个较小的盘子到一个较大的盘子上面。每隔一次移动就会涉及小盘子，这是出于同一个原因：每隔一个数目是奇数，每隔尺子上一个刻度都是最短的。也许我们的僧侣知道这个秘密，否则我们就很难想像他们怎么去决定下一步该移动哪一个。

用归纳法形式化证明在汉诺塔方案中涉及小盘子的每隔一步移动能让我们深感受益：当 $n$=1 时，包括小盘子，有一步移动，因此性质成立。当 $n$>1 时，假设在 $n$-1 时性质成立，暗示出在递归结构中，性质依然对 $n$ 成立：对于 $n$-1，第一个解决办法从一个小盘子的移动开始，对于 $n$-1，第二个解决办法以一个小盘子的移动结束，因此对于 $n$ 而言的解决办法都是从一个小盘子的移动开始和结束的。我们把不涉及小盘子的移动放在两个涉及小盘子的移动之间（即结束 $n$-1 个盘子的第一个解决方案的移动和开始 $n$-1 个盘子的第二个解决方案的移动），因此每隔一次移动就会涉及小盘子的性质成立。

**程序 5.9　画一把尺子的非递归程序**

与程序 5.8 相反，我们也可以首先画出所有长度为 1 的线，再画出所有长度为 2 的线，以此类推，通过这种方法就可以画出一把尺子。变量 t 指示线的长度，变量 j 在两个连续的长度 t 的线之间，指示线的数目。外层 for 循环对 t 进行增量运算，维持性质 $j=2^{t-1}$。内层 for 循环画出所有长度 t 的线。

```
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);
  }
```

程序 5.9 是一种可选择的画尺子的方法，由与二进制数字的相关性产生（参见图 5.10）。我们将算法的这个版本称为自底向上的实现。这种方法是非递归的，但它显然受到了递归算法的启示。分治法算法与数字的二进制表示之间的相关性常常提供了改进版本的分析与发展的洞察力，例如自底向上的方法。我们考虑这个观点，以便了解甚至可能去改进我们考察的每一个分治法算法。

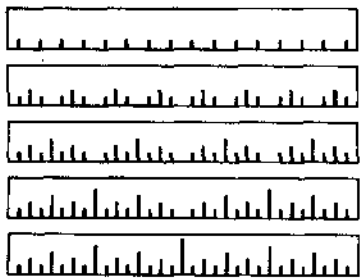


**图 5.10　以由底向上的顺序画一把尺子**

要非递归地画一把尺子，我们交替画长度为 1 的线，跳过位置；然后交替画长度为 2 的线，跳过保留的位置；交替画长度为 3 的线，跳过保留的位置；以此类推。

当我们画一把尺子时，自底向上的方法涉及到重新安排计算的顺序。图 5.11 显示了另一个例子，在其中我们重新安排了递归实现中的三个函数调用的顺序。它以我们先前描述的方式反映出递归计算：画出中间点的线，然后画左半部分，再画右半部分。画线的样式是复杂的，但却是

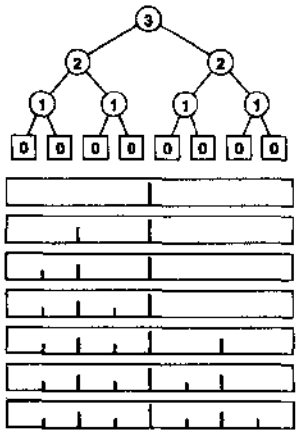
简单交换在程序 5.8 中两个语句的结果。就如我们将在 5.6 节看到的，图 5.8 与图 5.11 的关系和算术表达式的后缀与前缀间的区别实际上是类似的。

像在图 5.8 中那样按顺序画线与做程序 5.9 中包含的并在图 5.11 中指出的重新安排的计算相比，也许前者更可取。原因在于，只要我们想像出一种画线工具，它能够连续移到下一条线，我们就可以画一条任意长的尺子。类似地，要解决汉诺塔问题，我们就受限于按照它们被完成的顺序产生盘子移动的序列。总的来说递归程序依赖于子问题在特定顺序下的解决。对其他计算来说（例如，可参见程序 5.6），我们解决这个子问题的顺序是不相关的。对于这一类计算，惟一的约束就是我们必须先解决子问题再解决主问题。理解什么时候有重排计算的灵活性，不仅是使算法设计成功的秘密，而且在许多上下文中有直接的实际效果。比如说，在我们要考察在并行处理器中实现算法时，这就是致关重要的。

自底向上的方法符合算法设计的一般办法，在该算法设计中我们先解决琐碎的子问题，然后把这些解决办法联合起来去解决稍大的问题，以此类推，直到整个问题被解决。这样的方法可称为合治法（combine and conquer）。

从画一把尺子到画图 5.12 所示的二维图案，只有一小步距离。该图显示了一个简单递归描述如何导致一个看上去很复杂的计算（参见练习 5.30）。

如图 5.12 所示的递归定义的几何图案，有时被称为分形（fractal）。如果使用更多的复杂的画图元素，而且涉及更多的复杂递归调用（特别是那些在复杂平面中递归定义的函数），就能开发出显著多样性与复杂性的图案。图 5.13 所显示的另一个例子是 Koch Star，它被递归地定义如下：0 阶（order）的 Koch Star 是图 4.3 中的简单 hill 例子，$n$ 阶的 Koch Star 是 $n-1$ 阶的 Koch Star，并且其每个线段都以 0 阶 Koch Star 所代替的 Koch Star。




```
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)
```

**图 5.11 划尺函数调用（前序版本）**

这一序列指示出在递归调用前（而非它们之间）画线的结果。

如同画尺寸和汉诺塔的解决办法，这些算法在步骤数上都是线性的，但该步骤数在递归的最大深度上是指数级的（参见练习 5.29 和练习 5.33）。它们同样能直接与适当数目系统的计数相关（参见练习 5.34）。

汉诺塔问题、画尺子问题和分形都是很有趣的；而且与二进制数的联系也是令人惊

异的，但我们在所有这些主题中的首要兴趣在于，它们提供给我们一种理解这些基本算法设计范型的洞察力，这些分半（divide in half）的范型能独立解决一个或两个分半部分，这些技巧也许是我们在本书中所要考虑的最重要的技术。表 5.1 包括了关于折半查找与归并排序的细节，它们不但是重要而广泛应用的实用算法，而且例示了分治法的算法设计范型。

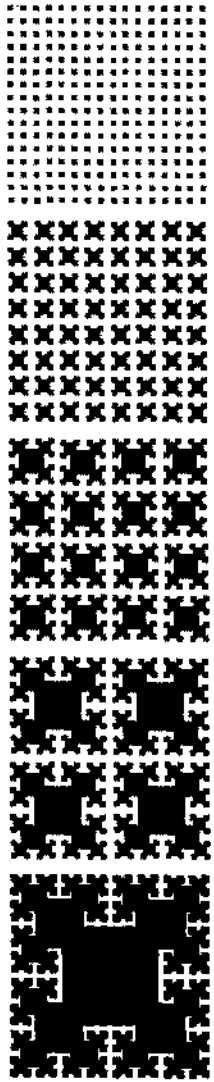
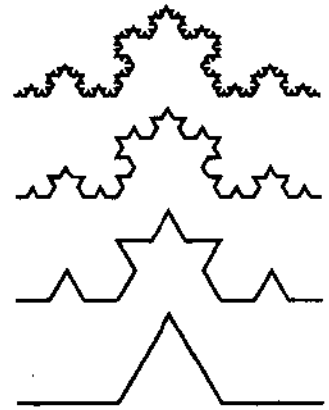


图 5.12　二维的分形星

这个分形是图 5.10 的二维版本。底端图突出显示的方框突出显现了计算的递归结构。



```
/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
```

图 5.13　Koch 分形的递归 PostScript

对于图 4.3 PostScript 程序的修正，把输出转成分形（见正文）。

表 5.1 基本的分治法算法

折半查找（参见第二章、第十二章）与归并排序（参见第八章）都是原型的分治法算法，提供保证的最优性能，分别用于搜索与排序。该递推指示了每一个算法的分治法计算的本质（参见 2.5 节、2.6 节最右栏的解决办法）。折半查找将一个问题分成两半，先进行一次比较，然后为其中一半进行一个递归调用。归并排序将一个问题分成两半，分别对两部分进行递归运算，再进行 $N$ 次比较。在本书中，我们将考察大量用这些递归方案开发的其他算法。

| | 递推 | 近似解 |
|---|---|---|
| 折半查找 | | |
| 比较 | $C_N = C_{N/2}+1$ | $\lg N$ |
| 归并排序 | | |
| 递归调用 | $A_N = 2A_{N/2}+1$ | $N$ |
| 比较 | $C_N = 2C_{N/2}+N$ | $N\lg N$ |

快速排序（参见第七章）与折半查找（参见第十二章）表示了一个在基本的分治法主题上很有意义的变体，在这一主题上，问题被分解成关于 $k$ 值的大小为 $k$-1 和 $N$-$k$ 的子问题，其中 $k$ 值由输入决定。在任意输入下，这些算法把一个问题平均分解成大小小于问题本身一半的子问题（就如在归并排序或折半查找中那样）。当我们讨论算法时，就研究这种不同效果的分析。

基本主题的其他值得我们认真考察的变体包括：分解成可变大小的部分、分解成多于两个的部分，分解成重叠部分、在算法的非递归部分进行大量的工作。一般来说，分治法算法包括的工作有把输入进行分片，或者合并两个独立解决的输入部分的处理结果，或者在一半的输入被处理后继续协助跟进。也就是说，在两个递归调用之前、之后或之间可以有代码。自然，这类变体引致的算法比折半查找与归并排序更复杂，并且分析起来更困难。我们考察本书中的大量例子；在第八部分返回到高级程序与分析。

## 练习

5.16 写出一个递归程序，基于比较数组中第一个元素与余下部分中的最大元素（递归计算地），找出一个数组中的最大元素。

5.17 写出一个递归程序，找出链表中的最大元素。

5.18 修改在数组中找出最大元素的分治法程序（程序 5.6），把一个大小为 $N$ 的数组分解为大小为 $k=2^{\lceil \lg N-1 \rceil}$ 的一部分和大小为 $N$-$k$ 的另一部分（以便至少有一部分的大小是 2 的幂）。

5.19 根据来自练习 5.18 的程序所做的递归调用，画出数组大小为 11 的相应的树。

● 5.20 用归纳法证明分治法算法产生的函数调用的次数是线性的，该分治法算法把一个问题分解成若干个部分，然后递归地解决每一部分。

● 5.21 证明：汉诺塔问题（见程序 5.7）的递归解决办法是最佳的。换句话说，证明用别的办法就需

要至少 $2^N-1$ 次移动。

▷ 5.22 写出一个递归程序，计算在一把有 $2^n-1$ 个线的尺子中第 $i$ 个线的长度。

●●5.23 考察诸如图 5.9 所示的 $n$ 位数字的表，寻找决定求解汉诺塔问题的第 $i$ 步移动方向的第 $i$ 个数字的性质（如图 5.7 的符号位指示的）。

5.24 写出一个程序，如同程序 5.9 那样，通过填充一个包括所有移动的数组，产生汉诺塔问题的解。

○ 5.25 写出一个递归程序，用 0 与 1 填充一个 $n\times 2^n$ 数组，其中该数组如图 5.9 所述那样代表了所有 $n$ 位二进制数字。

5.26 对这些非计划的参数值：rule(0,11,4)，rule(4,20,4)和 rule(7,30,5)，使用递归的画尺寸程序（参见程序 5.8）的结果。

5.27 证明关于画尺子程序（参见程序 5.8）的下列事实：如果它开始两个参数之间的差是 2 的幂，那么它的递归调用也有这一性质。

○ 5.28 写出一个函数，可以有效地计算一个整数的二进制表示中尾缀 0 的数目。

○ 5.29 在图 5.12 中有多少个正方形（要数出被更大的正方形所覆盖的那些）。

○ 5.30 写出一个 C++递归程序，以使它能输出一个画出图 5.12 底下那个图的 PostScript 程序，调用一系列 x y r box 函数，在$(x, y)$画一个边长为 $r$ 的正方形。用 PostScript 实现 box（参见 4.3 节）。

5.31 写出一个自底向上的非递归程序（类似于程序 5.9），以练习 5.30 所述的方法画出图 5.22 中的底下那个图。

● 5.32 写出一个 PostScript 程序，画出图 5.12 中的底下那个图。

▷ 5.33 在 $n$ 阶 Koch Star 中有多少个线段？

●●5.34 画一个 $n$ 阶 Koch Star，实现以下形式的命令序列“旋转 $a$ 度，再画一个长度 1/3”的线段”。找出与数目系统的对应性，通过递增一个计数器，再从计数器值计算 $a$ 角，使你得以画出 star。

● 5.35 修正图 5.13 中的 Koch Star 程序，产生出基于 0 阶的五边形的不同分形，能以东、北、东、南、东的顺序移动一个单位。

5.36 写出一个递归的分治法函数，给定端点，在整数坐标空间中对一个线段画出近似的值。假设所有坐标在 0 和 $M$ 之间。提示：先给出一个接近中间的点。

## 5.3 动态编程

我们在 5.2 节中考察的分治法算法的本质特征就是这些算法把一个问题划分成独立的子问题。如果子问题并不独立，问题就会复杂得多，基本的原因是：即使是这种最简单算法的直接递归实现，也可能需要难以想像的时间。在这一节中，为避免存在于一些情况中的这种缺陷，我们考察一个系统的方法。

例如，程序 5.10 是一个定义斐波纳契数（参见 2.3 节）的递推的直接递归实现。千万不要使用这样的程序，因为它极端低效。实际上，对于计算 $F_N$ 的递归调用的数目正是 $F_{N+1}$。

但是，$F_N$ 大约是 $\phi^N$，而 $\phi$ 约等于 1.618 是黄金率。令人尴尬的事实是程序 5.10 是针对这个微不足道的计算的指数级算法。图 5.14 为一个小例子描述了递归调用，使得被涉及的重新计算的数量变得简明。

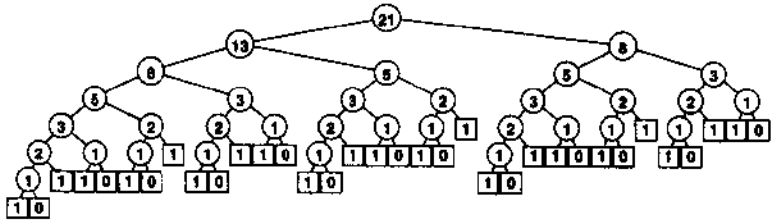


**图 5.14 斐波纳契数的递归算法的结构**

要求用于通过标准递归算法计算 $F_8$ 的这一递归调用的图解，显示了用重叠的子问题的递归如何能导致指数级的代价。在这个例子中，第二个递归调用忽略了由于多重递归在开始就完成了的大量的计算。计算 $F_6$=8 的递归调用（它在根的右子树和根的左子树中反映出来）如下所示。

```
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)
```

相比之下，很容易以与 $N$ 成正比的时间来计算出前 $N$ 个斐波纳契数，使用如下数组：

```
F[0] = 0; F[1] = 1;
for (i = 2; i <= N; i++)
  F[i] = F[i-1] + F[i-2];
```

该数目以指数级增长，因此这个数组是小型的，比如说 $F_{45}$=1836311903 是最大的斐波纳契数，能以 32 位整数表示，所以大小为 46 的数组就能做到。

这一技术给了我们一种获取大量的关于递推关系的解决办法。在斐波纳契数的例子中，

我们甚至能舍弃数组，只要跟踪前两个值（参见练习 5.37）；对于大量的其他经常遇到的递推（比如参见练习 5.40），我们需要用所有已知值来维持数组。

递推是一个有整数值的递归函数。我们在前几段的讨论中可以得出这样的结论：我们可以按从最小开始之顺序计算所有函数值来求任何类似函数的值，在每一步使用先前已计算的值计算当前值。我们称这种方法为自底向上的动态编程。假设我们能够保存所有先前计算的值，那么它能应用于任何递归计算。这是一个算法设计的技术，已成功应用于广泛的问题解决方案。我们必须注意一个简单的技巧，以便能从指数级向线性改进一个算法的运行时间。

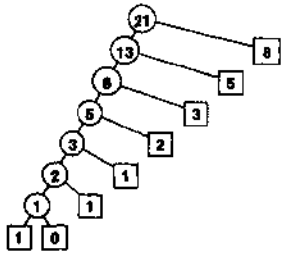



**图 5.15　计算斐波纳契数的自底向上的动态编程**

这一使用一个递归算法的自底向上的动态编程实现来计算 $F_8$ 的递归调用的图解，显示出怎样去存储已计算的值，以使代价从指数级降到线性（参见图 5.14）。

自顶向下的动态编程甚至是一个更简单的技巧，与由底向上的动态编程相比，它在同一（或更小）代价的基础上自动允许我们实现递归函数。我们实现递归程序存储每一个它所计算的值(正如它最末的步骤)，并对这些已存值进行检验，以避免重新计算它们中的任何项（正如它最初的步骤）。程序 5.11 是程序 5.10 的机械转换，通过自顶向下的动态编程将它的运行时间减少为线性。图 5.15 显示了由于自动转变使得递归调用的数目显著减少。自顶向下的动态编程因此有时也称为默记法（memoization）。

**程序 5.10　斐波纳契数（递归实现）**

这一程序虽然紧凑和精致，但却不可用，其原因在于它在计算 $F_N$ 时用的是指数级时间。计算 $F_{N+1}$ 的运行时间 $\phi$ 约等于计算 $F_N$ 的 1.6 倍。比如说，由于 $\phi^9>60$，如果我们注意到计算 $F_N$ 需要一秒的时间，我们就会知道计算 $F_{N+9}$ 的运行时间将会超过一分钟，计算 $F_{N+18}$ 的运行时间将会超过一小时。

```
int F(int i)
  {
    if (i < 1) return 0;
    if (i == 1) return 1;
    return F(i-1) + F(i-2);
  }
```

对于更复杂的程序例子，考察背包（knapspack）问题：一个小偷打劫一个保险柜，发现柜子满是 $N$ 类不同大小与价值的物品，但他只有一个容积为 $M$ 的背包来装东西。背包问题就是要找出一个小偷该选择的物品组合，以把所有所偷物品的最大价值装走。例如用图 5.16 所描述的项类型，一个带大小为 17 的背包的小偷总共能够拿走 5 个 A（而不是 6 个），总共为 20；或者 D 和 E，总共为 24，或是其他组合中的一种。我们的目标就是在任何给定的项与背包容量中，找出一个有效的算法以便在众多可能性中获得价值最大的一个。

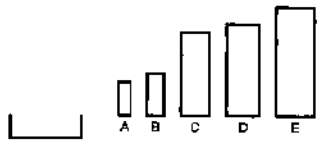
在许多应用程序中，背包问题的解决方法都是很重要的。比如说，一个装卸公司或许希望知道装载一卡车或一货轮货物的最佳途径。在这类应用程序中，其他的类似问题或许也会产生：比如说，对于每一个有效的装载项也许会有一个限制的数目，又或许不是一部而是有两部卡车。大量的类似问题都能用同一办法处理，我们可以考察刚才提到的基本问题的解决办法；还会有一些更复杂的办法。于是在这种问题的可行与不可行之间就有一条界限，我们将在第八部分中详尽考察。

在一个递归的解决背包问题的办法中，每次我们选择一个项，我们都假设可以（递归地）找到一个理想的方式把余下的背包空档都用尽。对大小为 cap 的背包，每一项 $i$ 在可选的项类型中，我们确定，可以通过放置 $i$ 并在 $i$ 周围放置其他项组成最优的一个打包得出一个总的值。最优的打包方式简单来说相当于我们已经发现（或将要发现）的大小为 cap-item[i].size 的更小背包。这一解决办法可以开发出最优决策的原则，一旦原则确定，就无需再行变更。只要知道如何用最优的一套项组合打包以获取较小的容积，我们就不需要重新检验那些问题，无论下一个项是什么。

程序 5.12 是一个基于上述讨论的直接递归办法。再有，这一程序不适用于解决实际问题，因为它大量的重新计算耗费了指数级时间（参见图 5.17），但我们能自动运用自顶向下的动态编程来消除这一问题，如程序 5.13 所示。如前所述，这种技术消除了所有的重新计算，如图 5.18 所示。

通过设计，动态编程消除了所有在递归程序中的重新计算，只是要符合这一条件：我们能够为小于所讨论的调用参数去保存函数的值。

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| item | A | B | C | D | E |
| size | 3 | 4 | 7 | 8 | 9 |
| val | 4 | 5 | 10 | 11 | 13 |



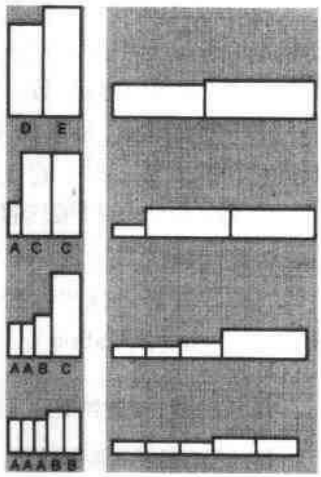




**图 5.16 背包例子**

一个背包问题的例子（上端）由一个背包容积和不同大小（横向）与值（纵看）的项构成。这个图显示了 4 种不同的方式以填充一个大小 17 的背包，其中两个方式使最高的可能总值为 24。

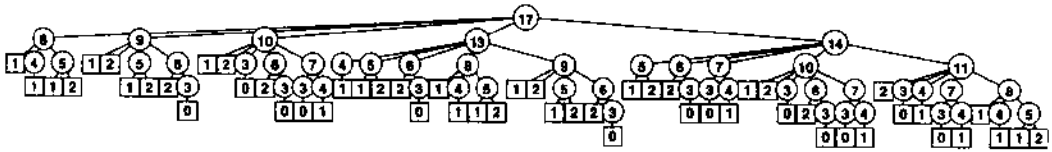


**图 5.17 背包算法的递归结构**

这棵树表示了程序 5.12 中的简单递归背包算法的递归调用结构。每一个节点中的数代表了背包中剩下的容量。算法遭遇了指数性能的同一基本问题，原因在于有大量的重复计算去处理重叠的子问题，我们已在计算斐波纳契数时考察过这些子问题。

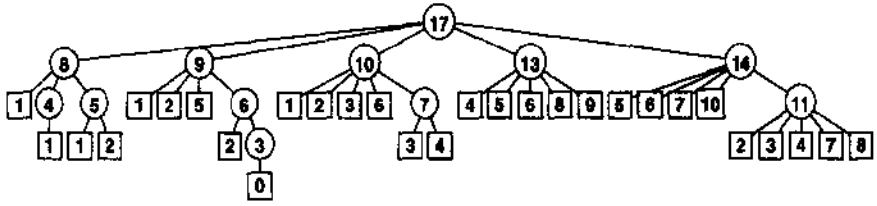



**图 5.18 背包算法的自顶向下的动态编程**

就如它为斐波纳契数计算所做的一切，保存已知值的技巧减少了背包算法的代价：从指数级变为了线性的。

**程序 5.11 斐波纳契数（动态编程）**

通过存储我们在一个静态数组中所计算的值（该数组的项在C++中被初始化为0），直接避免了任何重新的计算。这一程序以正比于 $N$ 的时间来计算 $F_N$，与程序5.10中使用的时间 $O(\phi^N)$ 形成对比。

```
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;
}
```

**性质 5.3** *动态编程减少递归函数的运行时间至多为：为所有小于或等于所给参数的参数去计算函数值所需的时间，并使处理一个递归调用的代价为常量（参见练习5.50）。*■

对于背包问题，性质5.3暗示了运行时间与 $NM$ 成比例。这样的话，我们就能轻松解决背包问题，只要背包容积并非极其庞大即可；对于极其庞大的背包容积，需要的时间和空间将可能大到被禁止的程度。

自底向上的动态编程同样应用于解决背包问题。实际上，我们可以随时在使用自顶向下的办法时去使用自底向上的办法，当然我们需要谨慎地确保按一个正确的顺序计算函数值，以便我们所需的每一个函数值都能被计算。对于带有单个参数的函数，诸如我们已考虑的那两个，我们一般是以参数的递增顺序来进行处理（参见练习5.53）；对于更复杂的递归函数，要确定一个恰当的顺序是很不易的。

例如，我们不需要用单个整型参数来限制我们的递归函数。当我们有一个带若干整型参数的函数时，我们能够存储对较小的子问题的解决办法在多维数组中，一个参数对应一个维。其他那些完全不涉及整型参数的情形，就使用抽象的离散问题公式，这些公式能让我们把大问题分解成一个个小问题。我们将在第五部分到第八部分考察这类问题的例子。

**程序 5.12 背包问题（递归实现）**

就如我们在计算斐波纳契数问题的递归解决方法时所提醒的，千万不要使用这个程序，因为它将耗费指数级时间，即使是对于小问题也不能完成。但是它毕竟表明了一种我们可以很轻易改进的紧凑方案（参见程序 5.13）。这一代码假设那些项都是有大小和值的结构，并由：

```
typedef struct { int size; int val; } Item;
```

定义。我们有一个类型 Item 的 N 项的数组。对于每一个可能的项，我们（递归地）计算我们所能得到的最大值，包括那一项，然后挑出那些值中最大的一项。

```
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;
  }
```

在自顶向下的动态编程中，我们存储已知的值；在自底向上的动态编程中，我们预先计算这些值。我们通常选择自顶向下的动态编程而不选择自底向上的动态编程，其原因如下：

- 自顶向下的动态编程是一个自然问题解决办法的机械转换；
- 计算子问题的顺序能自己处理；
- 我们或许不需要计算所有子问题的答案。

动态编程的应用程序在子问题的本质及我们关于子问题的要存储的信息总量是不同的。

我们不能忽视的至关紧要的一点是，当我们需要的可能的函数值的数目太高以至于不能存储（自顶向下）或预先计算（自底向上）所有值时，动态编程就会变得低效。比如，如果背包问题中的 $M$ 和数据项是 64 位的或是浮点数，我们将无法通过索引到数组中去存储值。这一差别不仅导致了小麻烦，还带来了一个主要困难。对这种问题，却没有好的解决办法；我们在第八部分中将看到，的确没有好方法。

动态编程是一个算法设计技巧，基本适合于我们将在第五到第八部分考察的那类高级问题。我们在第二部分到第四部分考察的大多数算法都是分治法方法，并且都具有不重叠的子问题。我们聚焦于近正方形或近线性的而非近指数的操作过程。无论如何，自顶向下的动态编程确实是开发高效的递归算法实现的基本方法，这类算法应归入任何从事算法设计与实现所需的工具箱。

## 练习

▷ 5.37 写出能计算 $F_N \bmod M$ 的函数，要求对中间计算只使用一个常量空间。

5.38 $F_N$可以被表述为 64 位的整数时，最大的 $N$ 是多少？

○ 5.39 在程序 5.11 中交换递归调用的情形下，画出对应图 5.15 的树。

5.40 写出一个函数，使用自底向上的动态编程来计算由下面的递推定义的 $P_N$ 的值：

$$P_N=\lfloor N/2\rfloor+P_{\lfloor N/2\rfloor}+P_{\lceil N/2\rceil},\quad N\geq 1 \qquad P_0=0$$

画出一个 $P_N$-$N\lg N/2$ 与 $N$ 之间的曲线图，其中 $0\leq N\leq 1024$。

5.41 写出一个函数，使用自顶向下的动态编程解决练习 5.40。

○ 5.42 当 $N=23$ 时，为练习 5.41 中的函数画出一棵与图 5.15 对应的树。

**程序 5.13 背包问题（动态编程）**

对于程序 5.12 的代码的改进，减少了运行时间，使时间从指数级的变成线性的。我们简单地保存我们所计算的函数值，然后在我们需要的时候检索已经保存的值（使用一个 sentinel 值来表达未知值），而不是制造递归调用。我们存储数据项的索引，以便我们能够在计算之后重建背包的内容，如果我们希望：itemKnown[M]在背包中，那么剩下的内容就跟大小为 M-itemKnown[M].size 的最优的背包的内容一致，因此 itemKnown[M-items[M].size]在背包里，以此类推。

```
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;
  }
```

5.43 画出 $N$ 与由练习 5.41 中的函数为了计算 $P_N$（$0\leq N\leq 1024$）而做的递归调用的数目之间的曲线图（要完成计算，你的程序中的一个 $N$ 的取值都从零开始）。

5.44 写出一个函数，使用自底向上的动态编程来计算由下面的递推定义的 $C_N$ 的值：

$$C_N=N+\frac{1}{N}\sum_{1\leq k\leq N}(C_{k-1}+C_{N-k}),\quad N\geq 1 \qquad C_0=1$$

5.45 写出一个函数，使用自顶向下的动态编程来解决练习 5.44。

○ 5.46 当 $N=23$ 时，为练习 5.45 中的函数画出一棵与图 5.15 相对就应的树。

5.47 画出 $N$ 与由练习 5.45 中的函数为了计算 $C_N$（$0\leq N\leq 1024$）而做的递归调用的数目之间的曲线图（要完成计算，你的程序中的每一个 $N$ 的取值范围都从零开始）。

▷ 5.48 根据图 5.16 中的数据项，给出由程序 5.13 的调用 knap(17)计算的 maxKnown 与 ItemKnown 的数组的内容。

▷ 5.49 假设要按数据项的大小递减的顺序来考虑，画出与图 5.18 对应的树。

5.50 证明性质 5.3。

5.51 使用程序 5.12 的自底向上动态编程版本，写出一个解决背包问题函数。

5.52 使用自顶向下动态编程，写出一个解决背包问题的函数，但要基于包括在背包中特定项的最优数目的计算，使用一个递归解决方法，并且（递归地）知道没有那个特定项时打包背包的最优方法。

5.53 使用练习 5.52 中所述的递归解决方法的自底向上动态编程版本，写出一个解决背包问题的函数。

5.54 使用动态编程来解决练习 5.4。跟踪你保存的函数调用的总数。

5.55 写一个程序，使用自顶向下动态编程来计算二项式的系数 $\binom{N}{k}$，基于递归规则 $\binom{N}{k}=\binom{N-1}{k}+\binom{N-1}{k-1}$，且 $\binom{N}{0}=\binom{N}{N}=1$。

## 5.4 树

树是一种数学上的抽象，在算法的设计与分析中起到了一个中心作用，理由如下：

- 我们使用树来描述算法的动态性质。
- 我们创建并使用显式的数据结构，这些数据结构都是树的具体实现。

我们已经看过上述用法的例子。我们为第一章基于树结构的连通性问题设计了算法，并且在 5.2 节、5.3 节我们使用树结构描述了递归算法的调用结构。

我们在日常生活中经常遇到树——这是一个熟悉的基本概念。比如说，许多人用家族树根踪祖先和后代；就如我们将看到的，我们的许多术语都是源于树的这种用法。另一个树的例子可在运动联赛的组织中找到；该用法由 Lewis Carroll 等人研究。第三个例子就是一个大型公司中的组织机构图；该用法对刻画分治法算法的层次分解是很有启示作用的。第四个例子就是转变英语句子为组成部分的分析树；如第五部分所讨论的那样，这一类树与计算机语言的处理密切相关。图 5.19 给出了一个树的典型例子——这棵树描述了本书的结构。此外我们在本书中还接触到大量树的应用的其他例子。

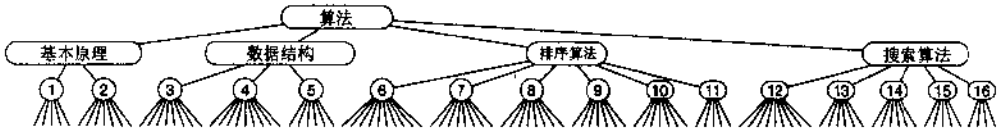



**图 5.19 一棵树**

这棵树描述了本书结构中的“部分”、“章”、“节”。对每一个实体，都有一个节点。每一个节点通过向下的链连接到它的组成要素，通过向上的链连接它属于的更大的部分。

在计算机的应用程序中，其中最为我们熟悉的树的用法就是组织文件系统。我们把文件放置在目录（有时也称为文件夹）中，这些目录被递归定义为目录与文件的序列。这种递归定义再次反映出一个自然的递归分解，而且与一个树的特定类型的定义相同。

有许多种树，而且它对于我们理解抽象表述与具体表述之间的区别是极其重要的，当我们为一个应用程序工作时，我们就要用到这些表示。因此，我们将详细考虑不同类型的树以及它们的表示。我们通过把树定义成抽象的对象，并引入大量基础的相关术语来开始我们的讨论。我们将非正式地讨论不同类型的树，对于这些树我们需要在一般性的递减顺序中加以考虑：

- 树；
- 有根树；
- 有序树；
- $M$ 叉树与二叉树。

用上述非正式的讨论开展了背景介绍后，我们转向正式的定义，并考察表示方法与应用。图 5.20 显示出许多我们讨论并定义的基本概念。

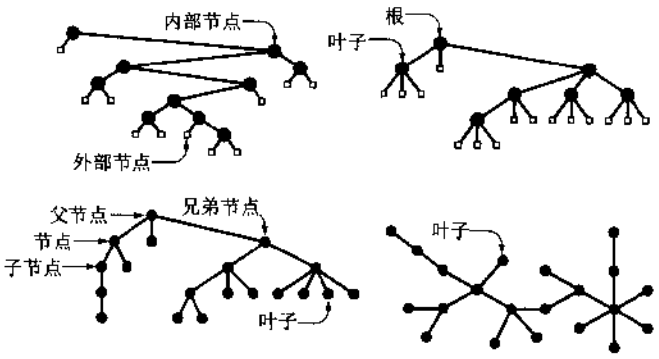



**图 5.20 树的类型**

这些图是二叉树（左上）、二叉树（右上）、有根树（左下）和自由树（右下）的例子。

一棵树就是一个满足一定要求的顶点与边的非空集合体。一个顶点就是一个简单的对象（也被称为是节点），它可以有个名字，并且可以携带其他关联信息；一个边就是两个节点之间的连接。树中的一条路径就是一个不同的节点序列，在序列中连续的节点由树中的边连接。定义一棵树的性质就是要有一条简明的路径连接任意两个节点。如果在几对节点中有不止一条的路径，或者在几对节点中没有路径，那样我们就有了一个图（graph）；我们没有树。一组不相交的树被称作森林。

一棵有根树（rooted tree）就是我们在其中指派一个节点作为树的根的树。在计算机科学中，我们通常指的树都是有根树，同时使用自由树指前几段中所提到的更一般的结构。在一棵有根树中，任何节点都是一棵由该节点及其下面的节点组成的子树的根。

在根与树中其他节点之间只有一条路径。这一定义暗示在边上没有方向；我们通常认为，依靠应用，边就是全部远离根或全部指向根的。我们通常用在顶端的根来画有根树（尽管这种惯例一开始看起来并不自然）。如果 $x$ 在一条从 $y$ 到根的路径上（也就是说，如果 $y$ 在节点 $x$

的下面，并与 $x$ 以一条不通过根部的路径相连，我们就说节点 $y$ 在节点 $x$ 的下面（及 $x$ 在 $y$ 的上面）。每一个节点（除了根）都只有一个节点在它的上面，这个节点被称作父节点，而直接在其下面的节点被称为子节点。我们有时需要对祖父节点或是对兄弟节点进行研究，以类比家族树。

没有子节点的节点被称作叶子或是终端节点。与后一种名称相对应，至少有一个子节点的节点被称为非终端节点。我们已经在本章看到了区分这些不同类型节点的应用的例子。我们使用树来表达递归算法的调用结构（比如，参见图 5.14）时，非终端节点（环）表示带有递归调用的函数调用，终端节点（方形）表示没有递归调用的函数调用。

在特定的应用中，每一个节点的子节点被排序方法是有着重要意义的；而在另外一些应用中，其意义却并不大。一棵有序树就是在其中每一个节点上的子节点的顺序是确定的有根树。有序树是一种很自然的表示：比如说，在画一棵树时我们按照某种次序放置子节点。实际上，许多其他的自然具体表示都有一个相似的暗示的次序；比如说，当我们考虑在一部计算机中表示树时，这一区别就变得很有意义了。

如果每一个节点必须有一个特定数目的子节点并以特定的次序出现，我们就有了一棵 $M$ 叉树。在这种树中，经常要确切地定义专门的外部节点，而且这些节点都没有子节点。然后，外部节点可以通过没有特定数目子节点的节点，扮作哑元节点引用。特别地，最简单的 $M$ 叉树类型就是二叉树。一棵二叉树就是由两种类型的节点组成的有序树：没有子节点的外部节点，有两个子节点内部的节点。由于每一个内部节点的子节点都是有次序的，所以我们称它们为内部节点的左子节点与右子节点：每一个内部节点都必须有左子节点与右子节点，尽管其中一个或两个可能是外部节点。在 $M$ 叉树中一个叶子就是一个内部节点，它们的子节点全部都是外部节点。

以上就是基本的术语，接下来，我们按照一般性递增的顺序考虑下列各种树的正式定义、表示以及应用：

- 二叉树与 $M$ 叉树；
- 有序树；
- 有根树；
- 自由树。

从最特定的抽象结构开始，我们将能够非常详尽地考虑具体的表述。

**定义 5.1** *二叉树是外部节点或内部节点，与一对分别被称作该节点的左子树与右子树的二叉树相连。*

该定义清楚表明二叉树本身是抽象的数学概念。当我们用计算机表示工作来时，我们就在处理着一个抽象的具体实现。这种情形与用浮点表示实数、用 int 表示整数等等类似。当我们画根部有节点的一棵树时，其根部通过边与左边的左子树及右边的右子树相连，我们就选择一种惯用的具体表示。有许多不同的方法表示二叉树（例如，参见练习 5.62），二叉树在开始很让人惊奇，但是在给定定义的抽象本质后，可以预想其反映结果。

当我们实现那些使用和控制二叉树的程序时，我们最常使用的具体表示就是一个内部节点的带有两个链接（包括右链接与左链接）的结构（参见图 5.21）。这些结构与链表（linked list）相似，但它们的每一节点有两个链接，而不是一个。空链接与外部节点相对应。特别地，我们加一个链接到来自 3.3 节的我们的标准链表表示上，如下：

```
struct node { Item item; node *l, *r; };
typedef node *link;
```

这只是针对定义 5.1 的C++语言代码。节点由项（item）与几对节点的指针（pointer）组成，我们同样称指向节点的指针为链接。因此，我们可以用一个诸如 x=x->l 的指针引用实现抽象操作“移动到左子树”。

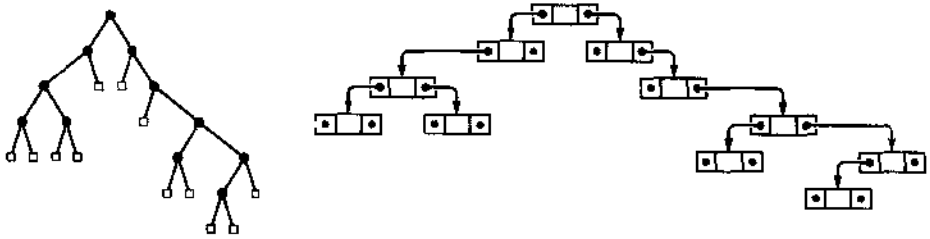


**图 5.21 二叉树表示**

二叉树的标准表示使用带有两个链的节点：左边的链连左子树，右边的链表示连右子树，空链对应外部节点。

这种标准表示允许实现有效地从根部下移的调用操作，但是不实现从子节点到其父节点的上移的调用操作。对于要求这类操作的算法，我们可以给每一个节点添加第三个链接，指向父节点。这一可供选择的方案与双向链表（doubly linked list）类似。利用链表（参见图 3.6），我们使树节点保持在一个数组中，并使用索引代替指针在一些情形中的作为链接。我们在 12.7 节中考察一个类似实现的特定例子。我们使用其他二叉树表示某些特定的算法，在第九章中尤其值得注意。

由于所有不同可能的表示的存在，我们也许可以开发一个二叉树的 ADT，把我们所要进行的重要操作封装进去，并且将这些操作的使用与实现分离开来。我们在本书中不采取这一途径，因为：

- 我们通常使用的都是双链接的表示；
- 我们使用树来实现更高水平的 ADT，并将聚焦于这些部分；
- 我们用其效率依靠于特定的表示的算法工作——一个可能在 ADT 中丢失的事实。

这些都是我们对数组与链表使用熟悉的具体表示的一致理由。在图 5.21 中描述的二叉树表示是一个基础的工具，我们现在把它加到这个短链表中。

对于链表，我们从考察基本的操作开始，这些操作有插入或删除节点（参见图 3.3、3.4）。对于二叉树的标准表示，这样的操作并非必要的基本要素，因为有第二个链接存在。如果我们要删除二叉树的一个节点，我们必须弄清基本问题，就是节点消失后我们可能有一个

父节点却有两个子节点要处理。于是可有三个自然的操作，都没有这个麻烦：在底端插入一个新的节点（用一个新节点的链接代替 null 链接）；删除一个叶子（用一个 null 链接代替与它的链接）；创制一个新根部把两棵树组合成一棵树，其中通过用左链接指向一棵树，用右链接指向另一棵树。当要操作二叉树时，我们大量使用这些操作。

**定义 5.2** *M 叉树是一个外部节点或内部节点与同样是 M 叉树的 M 个树的序列相连。*

我们通常在 $M$ 叉树中表示节点为名叫带 $M$ 个链接的结构（就如二叉树中的一样）或者 $M$ 个链接的数组。比如说，第十五章中，我们考察三叉（或三重的）树，在树上我们使用三个命名为左、中、右的链接的结构。每一个结构对于相关联的算法都有特定的含义。否则，使用数组去保存链接就是恰当的，因为 $M$ 的值是固定的，尽管我们将看到，在使用类似表示时我们必须特别注意额外空间的使用。

**定义 5.3** *树（也称为有序树）是一个节点（也称为根部）与一个不相交的树的序列连接的。这样的序列被称为森林。*

有序树与 $M$ 叉树的区别在于：有序树中的节点可以有任何数目的子节点，$M$ 叉树的节点必须有确定的 $M$ 个子节点。我们有时根据上下文使用一般树（general tree）这一术语把有序树从 $M$ 叉树中区别开来。

因为有序树中每一个节点可以有任意多个链接，因此很自然就考虑使用链表，而不是使用数组，来确定通向节点的子节点的链接。图 5.22 就是这种表示的例子。在这个例子中，我们很清楚地看到每一个节点包含两个链接，一个用于连接到它的兄弟节点，另一个用于它的子节点的链表。

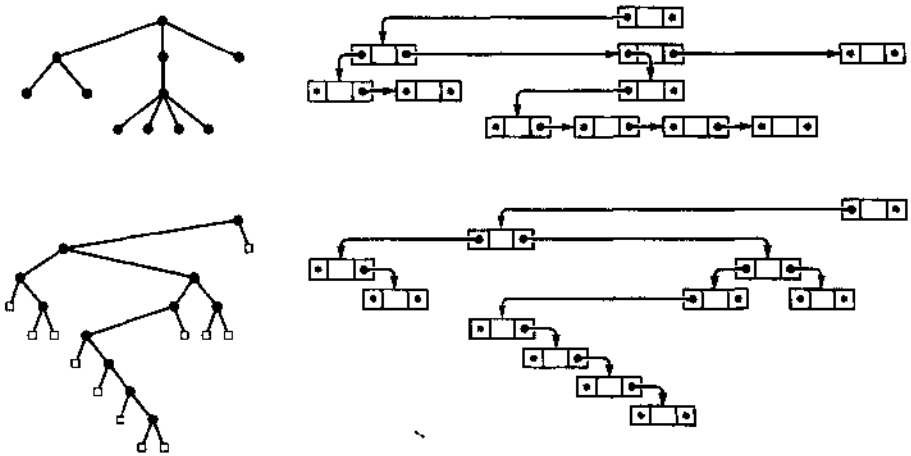


**图 5.22 树的表示**

通过保持每一节点的子节点的链表来表示一棵有序树，相当于把它表示为一棵二叉树。在右上方的图显示了一个在左上方的树的子节点链表表示，带有实现在节点的右链接的表，并且每一个节点的左链接指向其子节点链表中的第一个节点。在右下方的图显示了一个对上方图进行细微的重新安排的版本，并且非常清晰地表示左下方的二叉树。也就是说，在表示树时我们可以考虑二叉树。

**性质 5.4** 在二叉树与有序树之间，有一个一一对应的关系。

这种对应在图 5.22 中详细描述，通过使每一个节点的左链接指向最左的子节点，使每一个节点的右链接指向右边的兄弟节点，我们可以把任何的森林表示成一个叉树。

**定义 5.4** 一棵有根树或无序树是一个连接到有根树的多重集合（multiset）的一个节点（这样一种多重集合被称为无序森林）。

我们在第一章中遇到的关于连通性问题的树就是无序树。这一类树可以被定义为有序树，其中节点的子节点被考虑的顺序是不重要的。我们同样选择把无序树定义为在节点中包含一套父-子关系。这一选择看起来与我们正在考察的递归结构没有什么关联，但它却是对抽象观念的真实具体的表示。

我们可以在计算机中选择用一棵有序树来表示无有序树，并认识到许多不同的有序树或许表示了同一棵无序树。实际上，反过来要确定是否两个不同的有序树表示同一棵无序树（树同构问题）并非是一件很困难的事。

树最一般的类型就是没有根节点区别。比如说，在第一章的连通性算法引致的生成树就有这一性质。要恰当地定义无根树、无序树或自由树，我们就要从图的定义开始。

**定义 5.5** 一个图（graph）就是一组节点，它与连接着不同节点对的一组边在一起（这些节点至多有一条边连接着任一对节点）。

我们可以想像，从某个节点开始，跟随一条边到这条边的连续节点，然后跟从那个节点到另一个节点的边，以此类推。以这种方式，一个从一个节点到另一个节点的边的序列（没有节点出现两次）被称为简单路径。如果有一条简单路径连接任何节点对，一个图就是连通的。除了开头与末尾的节点是相同的之外一条简单路径被称作环路（cycle）。

每一棵树都是一个图，哪一些图是树呢？如果一个图满足以下四个条件的任意一条，我们就可以把它考虑为树：

- G 有 $N-1$ 条边，并且没有环路；
- G 有 $N-1$ 条边，并且是连通的；
- 只有一条路径连接 G 中的每一对顶点；
- G 是连通的，但在任意边被删除后不再保持连通。

上述的条件的任意一条对于证明其他三条都是必要且充分的。正式地说，我们应该挑选其中一个条件作为自由树的定义。非正式地说，我们让这些条件集体用于定义。

我们简单地表示一个自由树为边的集合。如果我们选择表示一个自由树为无序根、有序树甚至二叉树，我们需要认识到，在通常情况下，表示每一个自由树都有许多不同的方法。

树的抽象经常出现，在这一节中所讨论的区别极其重要，因为如果给定问题要去寻找一个有效的算法及相应的数据结构，这时懂得不同的树的抽象就是致关重要的了。我们常常直接用树的具体表示工作，而不理会特定的抽象，但我们也常常因为使用恰当的树抽象而受益，由此再去考虑不同的具体表示。我们将在本书中看到这个过程的大量的例子。

回到算法与实现之前，我们考虑一些树的基本数学性质；这些性质将在树算法的设计与分析中用到。

### 练习

▷ 5.56 作为有根树与二叉树，给出图 5.20 中的自由树的表示。

● 5.57 在表示图 5.20 中的自由树为有序树，有多少种不同的办法？

▷ 5.58 画出三棵与图 5.20 中的有序树同构的有序树。也就是说，你必须能够通过交换子节点来将四棵树转换成另外一棵树。

○ 5.59 假设有树包含一些项，operator= =是为这些项定义的。写出一个递归程序来删除二叉树中所有的叶子，其中项与给定的项相等（参见程序 5.5）。

○ 5.60 变换分治法函数，用于找出数组中最大的项（参见程序 5.6），用以分解数组为 $k$ 个部分，并由大小至多为 1 区别开来，递归地找到每一部分的最大值，再返回众多最大值中的最大值。

5.61 在练习 5.60 中提示的递归构造中使用的 $k$=3 与 $k$=4，为一个 11 元素的数组画出 3 叉树和 4 叉树（参见图 5.6）。

○ 5.62 二叉树与二进制字符串相当，二进制字符串比 1 位多一个 0 位，由于有这个额外的约束，在任何位置 $k$，严格出现在 $k$ 的左边的 0 位的数目不比严格出现在 $k$ 的左边的 1 位的数目大。一棵二叉树就是 0 或两个这样的字符串连接在一起，前面是 1。画出与下面的字符串相对应的二叉树：

```
1 1 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 0
```

○ 5.63 有序树相当于平衡的圆括号字符串：一棵有序树是一种空或是由圆括号包围的有序树的序列。画出与下面字符串相对应的有序树：

```
( ( ( ) ( ( ) ( ) ) ( ) ) ( ( ) ( ) ( ) ) )
```

●●5.64 编写一个程序判断：当被解释为带有在 0 与 $N$-1 之间编号的节点的树（就如第一章那样）中的父-子链接时，写出一个程序，以便决定 0 与 $N$ -1 之间 $N$ 个整数的两个数组是否能表示同构的无序树。也就是说，你的程序必须决定是否有方法重新给一棵树上的节点编号，以便这棵树的数组表示与另一棵树的数组表示相同。

●●5.65 写出一个程序决定是否两个二叉树表示了同构的无序树。

▷ 5.66 请画出所有能表示由边 0-1、1-2、1-3、1-4、4-5 定义的树的有序树。

● 5.67 证明：如果一个 $N$ 个节点的连通的图有删除任何边就断开了图的联系的性质，则该图有 $N$-1 个边，并且没有环路。

## 5.5 二叉树的数学性质

在开始考察树处理算法之前，我们通过考察一些树的基本性质继续数学的脉络。我们聚焦于二叉树，因为我们在整本书中经常要用到它。深刻理解它们的基本性质，将为我们理解将会遇到的不同算法的性能特征铺平道路，因为我们将遇到算法不止是使用二叉树作

为显式的数据结构，而且还有分治法递归算法以及其他类似的应用。

**性质 5.5** 一棵有 $N$ 个内部节点的二叉树有 $N+1$ 个外部节点。

我们通过归纳法来证明这个性质：一棵有 0 个内部节点的二叉树有 1 个外部节点，由此当 $N=0$ 时性质成立。当 $N>0$ 时，任何有 $N$ 个内部节点的二叉树，就有 $k$ 个内部节点在它的左子树上，有 $N$-1-$k$ 个内部节点在它的右子树上，其中 $k$ 介于 0 与 $N$-1 之间，因为根是个内部节点。通过归纳假设，左子树有 $k$+1 个外部节点，左子树有 $N$-$k$ 个外部节点，总计 $N$+1 个外部节点。■

**性质 5.6** 一棵有 $N$ 个内部节点的二叉树就有 $2N$ 个链接：$N$-1 个链接到内部节点，$N$+1 个链接到外部节点。

在任何有根树中，每一个节点（除了根）都有一个惟一的父节点，每一条边连接一个节点到它的父节点上，所以有 $N$-1 个链接与内部节点连接。相似地，$N$+1 个外部节点的每一个都有一个链接到惟一的父节点。■

许多算法的性能特征不只是依靠于相关树的节点数目，而且依靠于不同的结构化性质。

**定义 5.6** 节点在一棵树的层数比它父节点层数（level）更高（根部在层数 0 上），树的高度就是树节点层数中的最大值。树的路径长度是所有树节点的层数的总和。一棵二叉树的内部路径长度是树的所有内部节点的层数的总和。一棵二叉树的外部路径长度是树的所有外部节点的层数的总和。

一种计算树的路径长度的方便方法，就是对所有的 $k$，求 $k$ 与在层数 $k$ 上节点的数目的乘积的和。

这些数量同样有简单的递归定义，这些定义直接来自树与二叉树的递归定义。比如，树的高度比树根的子树的高度的最大值还大 1，有 $N$ 个节点的树的路径长度就是树根的子树的路径长度的总和加 $N$-1。这些量同样与递归算法的分析直接相关。比如，对于许多递归计算，相应树的高度正好是递归深度的最大值，或是需要支持计算的栈的大小。

**性质 5.7** 任何带有 $N$ 个内部节点的二叉树的外部路径长度比内部路径长度大 $2N$。

我们通过归纳法来证明这个性质，但有个可选的试验是有启发性的（该试验同样对性质 5.6 的证明有效）。注意到任何二叉树都能通过以下的过程创建。从一个外部节点组成的二叉树开始，然后重复以下步骤 $N$ 次：挑选一个外部节点，再用一个有两个外部节点作为子节点的新内部节点来代替之。如果所选的外部节点的层数是 $k$，内部路径长度以 $k$ 增长，内部路径长度以 $k$+2 增长（一个层数是 $k$ 的外部节点被删除，但两个层数是 $k$+1 的外部节点被增加）。整个过程从内部路径长度和外部路径长度同为 0 的树开始，对于 $N$ 个步骤的每一步，增加外部路径长度比内部路径长度多 2。■

**性质 5.8** 带有 $N$ 个内部节点的二叉树的高度至少是 $\lg N$，至多是 $N$-1。

最糟的情况是退化树，只有一个叶子，$N$-1 个链接从根部到叶子（参见图 5.23），最好的情况是平衡树，在每一个层数 $i$ 上有 $2^i$ 个内部节点（底端层数除外）（参见图 5.23）。如果高度是 $h$，那么我们一定要有 $2^{h-1}<N+1\le 2^h$，因为有 $N$+1 个外部节点。这个不等式暗示了

我们提到的性质：最好情况的高度正好等于上舍入的最近的整数的 $\lg N$。■

**性质 5.9** 带有 $N$ 个内部节点的二叉树的内部路径长度至少是 $N\lg(N/4)$ 至多是 $N(N-1)/2$。

最糟的情况和最好的情况都是由在性质 5.8 讨论中引用的和在图 5.23 中描述的同样的树造成的。最糟的情况的树的内部路径长度是 $0+1+2+\cdots+(N-1)=N(N-1)/2$。最好的情况的树有 $(N+1)$ 个外部节点，高度不超过 $\lfloor \lg N \rfloor$。把这些乘起来并应用性质 5.7，我们得到 $(N+1)\lfloor \lg N \rfloor - 2N < N\lg(N/4)$。■

就如我们将看到的，二叉树广泛的出现在计算机应用中，而且当二叉树完全平衡（或接近平衡）时计算的性能最佳。比如，我们用于描述分治法算法的树，如折半查找与归并排序都是（参见练习 5.74）。在第九章与第十三章，我们将考察基于平衡树的数据结构。

树的基本性质为我们要开发一些用于解决实际问题的有效算法提供了信息。对于我们将遇到的特定算法的更详尽分析，就要求有熟练的数学分析，虽然我们用诸如在本节中用到的简明的归纳参数，常常能得到有用的估计。我们更进一步讨论树的数学性质，这些数学性质将在后面的几章中需要。就这点来说，我们准备转回到关于算法问题上来。

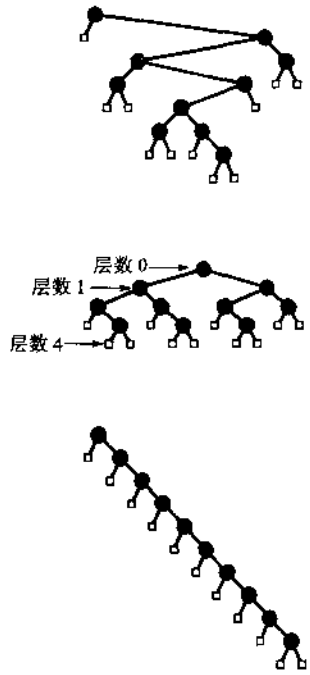



**图 5.23 三棵带有 10 个内部节点的二叉树**

展示在上端的二叉树的高度是 7，内部路径长度 31，外部路径长度 51；中间的完全平衡二叉树有 10 个内部节点，高度为 4，内部路径长度 19，外部路径长度 39（没有带 10 个节点的二叉树有比以上数量更小的值）。底端的退化的二叉树有 10 个内部节点，高度为 10，内部路径长度 45，外部路径长度 65（没有带 10 个节点的二叉树有比以上数量更大的值）。

## 练习

▷ 5.68 在一棵具有 $N$ 个内部节点的 $M$ 叉树中有多少个外部节点？使用你的答案来给出要表示这样的树所需的内存数量，假设每一个链接与项需要一个字的内存。

5.69 给出有 $N$ 个内部节点的 $M$ 叉树的高度的上限与下限。

○ 5.70 给出有 $N$ 个内部节点的 $M$ 叉树的内部路径长度的上限与下限。

5.71 给出有 $N$ 个节点的二叉树的叶子数目的上限与下限。

● 5.72 证明：如果二叉树中的外部节点的层数按常量变动，那么该高度是 $O(\log N)$。

○ 5.73 高度 $n>2$ 的斐波纳契树是一棵二叉树，在一个子树上有高度为 $n-1$ 的斐波纳契树，在另一个子树上有高度为 $n-2$ 的斐波纳契树。一棵高度为 0 的斐波纳契树是一个外部节点；一棵高度为 1 的斐波纳契树是一个内部节点，带有两个外部的子节点（参见图 5.14）。给出高度为 $n$ 的斐波纳契树的高度与外部路径长度，作为一个树中节点的数目 $N$ 的函数。

5.74 一棵有 $N$ 个节点的分治法树，有一个标注为 $N$ 的二叉树的根、一棵在一棵子树中有 $\lfloor N/2 \rfloor$ 个节

点的分治法树，一棵在另一棵子树中有 $N/2$ 个节点的分治法树（图 5.6 描述了一个分治法树）。用 11、15、16 和 23 个节点画出分治法树。

○ 5.75 用归纳法证明一个分治法树的内部路径长度在 $M\lg N$ 与 $M\lg N+N$ 之间。

5.76 一棵有 $N$ 个节点的合治法树，是一个具有标注为 $N$ 的根、一棵在一棵子树有 $N/2$ 个节点的合治法树，一棵在另一棵子树中有 $N/2$ 个节点的合治法树（参见练习 5.18）。用 11、15、16 和 23 个节点画出合治法树。

5.77 用归纳法证明一个合治法树的内部路径长度在 $M\lg N$ 与 $M\lg N+N$ 之间。

5.78 一棵完全二叉树就是所有层从左边到右边都是满的（可能除了最后一个之外），如图 5.24 所示。证明一个有 $N$ 个节点的完全树的内部路径长度在 $M\lg N$ 与 $M\lg N+N$ 之间。

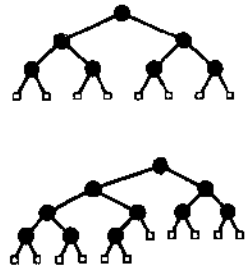


**图 5.24 有 7 个和 10 个内部节点的完全二叉树**

当外部节点的数目是 2 的幂（上图）时，完全二叉树中的内部节点都在同一层数。否则（下图）外部节点出现在两个层数，其中在次底层上的内部节点在外部节点的左方。

## 5.6 树遍历

在考察创建二叉树与树的算法之前，我们考察用于最基本的树处理功能的算法：树遍历，即给出一个指向树的指针，我们打算系统地处理树里的每一个节点。在一个链表中，我们跟随单一链接从一个节点移动到下一个节点；对于树来说，无论如何，我们都要做决定，因为也许有多重链接要跟随。

**程序 5.14 递归树遍历**

这个递归函数取指向树的链接作为参数，并且用树上的每一个节点作为参数调用函数 visit。也就是说，该函数实现一个前序遍历；如果我们将对 visit 的调用移动到递归调用之间，我们就有一个中序遍历；如果我们将对 visit 的调用移动到递归调用之后，我们就有一个后序遍历。

```
void traverse(link h, void visit(link))
  {
    if (h == 0) return;
    visit(h);
    traverse(h->l, visit);
    traverse(h->r, visit);
  }
```

我们从考虑二叉树的过程开始，对于链表，我们有两个基本选择（参见程序 5.5）：处理节点然后跟从链表（在这种情况下，我们按顺序访问节点），或者跟随链表然后处理节点（在这种情况下，我们按反顺序访问节点）。对于二叉树，我们有两个链表，由此我们有三个基本的顺序供我们访问节点：

- 前序，我们访问节点，然后访问左子树和右子树；
- 中序，我们访问左子树，然后访问节点，再访问右子树；
- 后序，我们访问左子树和右子树，然后访问节点。

我们可以用递归程序轻易地实现这些方法，如程序 5.14 所示，这是程序 5.5 中的链表遍历程序的直接一般化。要按其他顺序实现遍历，我们以适当的方式改变程序 5.14 中的函数调用。图 5.26 展示了我们访问一棵样例树的节点的顺序。图 5.25 展示了当我们在图 5.26 中的一棵样例树上调用程序 5.14 时执行函数调用的顺序。

```
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 H
    visit H
    traverse F
      visit F
      traverse *
      traverse G
        visit G
        traverse *
        traverse *
    traverse *
```

**图 5.25 前序遍历函数调用**

对于图 5.26 的一棵示例树，这个函数调用的顺序组成前序遍历。

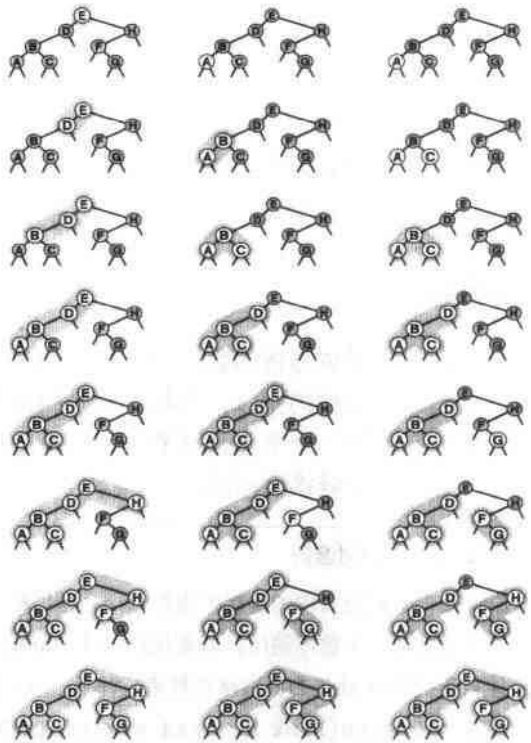


**图 5.26 树遍历次序**

这些顺序指出我们访问前序（左）、中序（中）、后序（右）的树遍历节点的顺序。

我们已经遇到同样的基础的递归步骤，不同的树遍历方法都是基于这一步骤的，如在分治法的三个程序中（参见图 5.8 与图 5.11）以及算术表达式中。比如，首先根据在尺子上画线进行前序遍历，然后进行递归调用（参见图 5.11）；对应于汉诺塔解决办法移动最大的盘子进行中序遍历；对应于估算后缀表达式的值进行后序遍历，等等。这些对应性使我们对于潜藏在树遍历背后的机制有了深刻认识。比如，我们知道，在中序遍历中每隔一个节点就是外部节点，出于同样的道理，在汉诺塔中每隔一步移动都涉及了最小的盘子。

考察那些使用显式的下推栈的非递归实现同样很有用处。为简单起见，我们从一个抽象的栈开始考察，这个栈能够保存项或树，以将被遍历的树做初始化。然后，我们进入一个循环，在这个循环中我们弹出并处理在栈顶端的项，如此继续，直到栈空了为止。如果已弹出的实体是个个项，我们就访问它；如果已弹出的实体是一棵树，我们就以希望的顺序进行一系列的入栈操作：

- 对于前序，我们压入右子树，然后是左子树，最后是节点；
- 对于中序，我们压入右子树，然后是节点，最后是左子树；
- 对于后序，我们压入节点，然后是右子树，最后是左子树；

我们并不推空树到栈上。图 5.27 显示了我们使用三种方法中的每一个来遍历图 5.26 中的样例树时栈的内容。我们通过归纳法可以轻易地证实这个办法可以得出与二叉树的递归方法同样的输出结果。

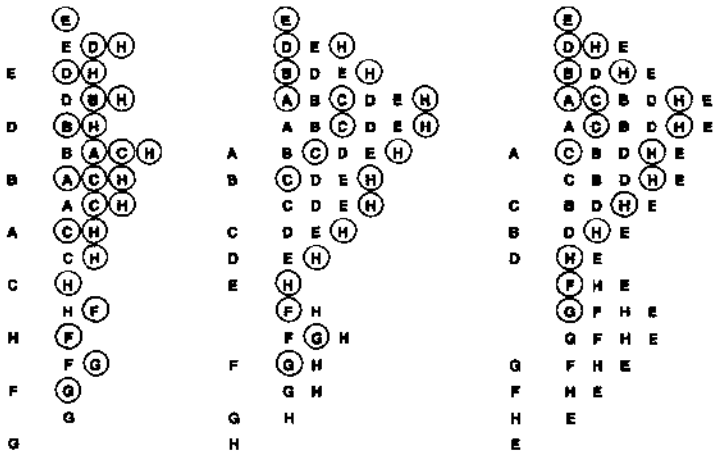


**图 5.27　遍历算法的栈内容**

这些顺序指示前序（左）、中序（中）、后序（右）树遍历的栈内容（参见图 5.26），用于一个理想化的计算模型，类似于我们在图 5.5 中使用的模型，在该模型中我们按指示的顺序把项及其两棵子树放到栈上。

在前几段中描述的方案是概念性的，包括了三个遍历的方法，但在我们的实际实现中要稍微简单一些。比如，对于前序，我们不需要把节点推到栈上（我们访问弹出来的每一棵树的根），这样的话，我们就能使用一个仅包含一种类型的项（树链接）的简单栈，类似于在程序 5.15 中的非递归实现。支持递归程序的系统栈包含返回地址与参数值，而不是项或节点，但是我们进行计算（访问节点）的实际顺序对于递归方法和基于栈的办法来说是一样的。

**程序 5.15　前序遍历（非递归）**

非递归的基于栈的函数与相应的程序 5.14 中的递归函数在功能上是相等的。

```
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);
      }
  }
```

第四个自然的遍历策略简单来说就是当节点出现在页面上时，访问一棵树上的节点，从头读到尾，从左读到右。这种方法称为层次顺序的遍历，因为在每一层上的节点都一起出现，并且是按顺序的。图 5.28 展示了在图 5.26 中的树的节点如何按层的次序逐一被访问。

值得注意的是，我们可以通过以一个队列来代替程序 5.15 中的栈来做到层次遍历，如同程序 5.16 所示。对于前序，我们使用 LIFO 数据结构；对于层次顺序，我们使用 FIFO 数据结构。这些程序值得认真研究，因为它们表示了对余下待完成的工作的组织办法。特别地，层次顺序并不跟那些和树的递归结构相关的递归实现一致。

**程序 5.16 层次顺序的遍历**

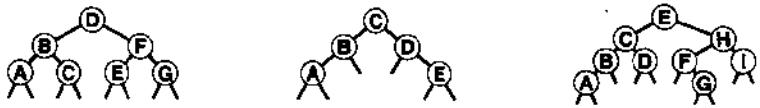
把前序遍历中基本数据结构从栈转变成队列，这样的转变就使遍历转成层次顺序的。

```
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);
      }
  }
```

前序、后序和层次顺序对森林同样能很好的定义。为了使定义一致，就把森林设想为一棵有假想根的树。然后，前序规则就是“访问根部，再访问每一个子树。”后序规则就是“访问每一个子树，再访问根部。”，层次顺序就和二叉树的一样。这些方法的直接实现都是基于栈的前序遍历程序（参见程序 5.14 与程序 5.15）以及我们刚考察的二叉树的基于队列的层次顺序遍历程序（参见程序 5.16）的简单一般化。我们省略了实现的考察，原因在于我们将在 5.8 节考察一个更一般的过程。

**练习**

▷ 5.79　给出下列二叉树的前序、中序、后序和层次顺序。




▷ 5.80　按图 5.27 的风格显示出图 5.28 中描述的层次顺序遍历期间队列的内容。

5.81　证明森林的前序与对应的二叉树的前序相同（参见性质 5.4），森林的后序与二叉树的中序相同。

○ 5.82　给出一个中序遍历的非递归实现。

● 5.83　给出一个后序遍历的非递归实现。

● 5.84　写出一个程序，取二叉树的前序和后序遍历作为输入，产生树的层次顺序遍历为输出。

## 5.7　递归二叉树算法

我们考虑 5.6 节中讨论的二叉树的算法就例证了我们被引向二叉树的递归算法这一基本事实，因为这些树作为递归结构的本质。许多任务允许直接递归分治法算法，这些算法在本质上对遍历的算法进行了一般化。我们通过处理根节点和（递归地处理）它的子树来处理整棵树；我们可以在递归定义之前、之间、之后甚至三段时间同时进行计算。

只给定一个链接到树，我们常常需要找到树的不同结构参数的值。比如说，程序 5.17 由计算给定树的节点的数目和高度的递归函数组成。该函数即刻符合了定义 5.6。这些函数均不依靠于处理递归调用的次序：如果我们交换递归调用，这些函数也处理树上的所有节点并返回相同的答案。并非所有的树参量都这么容易计算：比如说，一个能有效计算二叉树的内部路径长度的程序就是较大的挑战（参见练习 5.88 到练习 5.90）。

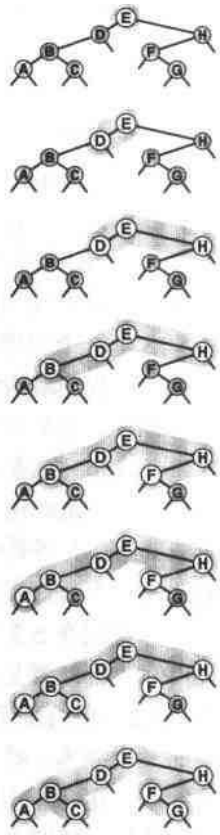


**图 5.28　层次顺序遍历**

这个顺序描述了按从上到下及从左到右的顺序访问节点的结果。

**程序 5.17　树参数的计算**

我们可以使用下面的简单的递归过程来学习树的基本结构性质。

```
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;
  }
```

无论何时当我们写一个处理树的程序，另一个有用的函数就是能显示或画出树的函数。比如，程序 5.18 是个递归的过程，它能按图 5.29 所示的格式显示一棵树。我们可以使用同样的递归办法画更详细的树的表示，比如我们在书中使用的那些树（参见练习 5.85）。

程序 5.18 是一个中序遍历——如果我们在递归调用之前显示该项，我们就得到一个前序遍历，该遍历同样展示在图 5.29 中。我们可能要用到的这种格式，比如对于一个族树，或是列出基于树的文件系统的文件，或是概括显示的文档（比如说，在图 5.19 中树的前序遍历给出本书的框架结构）等等都是我们所熟悉的格式应用。

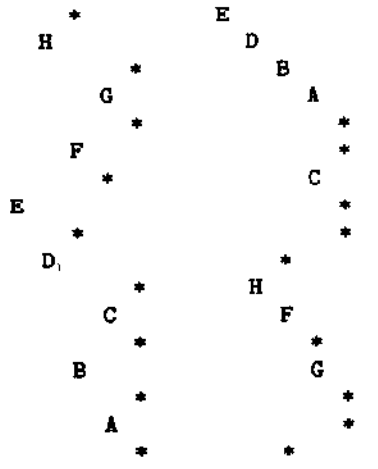



**图 5.29 显示一棵树（中序和前序）**

左方结果的输出是源自使用图 5.26 中的样例树上的程序 5.18，以类似于图的表示方式展示树结构（其中的图表示我们已经使用过，被转了 90 度）。右方结果的输出是源自同一程序，打印语句已经被移到了开头；它以我们熟悉的概括格式展示了树结构。

我们的第一个程序的例子（它能建造一个显式的二叉树结构）与我们在 5.2 节考察的找最大值的应用程序相关联。我们的目标就是造一个联赛（tournament）：一棵二叉树的每一个内部节点的项就是树的两个子节点中较大项的一个副本。特别地说，在根部的项是联赛中最大项的副本。在叶子中的项（无子节点的节点）构成了感兴趣的数据，树的余下部分是一个数据结构，该结构允许我们非常有效地找到项中最大的一项。

程序 5.19 是一个递归程序，该程序由数组中的项构造了一个联赛。程序 5.6 的扩展，它使用一种分治法的策略：为一个项构造一个联赛，我们创建（并且返回）一个叶子，该叶子就包括了那一个项。要为 $N>1$ 项构造一个联赛，我们就使用这种分治法的策略：把项分解成对半，为每一个对半构造一个联赛，并创建一个新的节点，该节点具有到两个联赛的链接，并且具有一个项，这个项是在两个联赛的根部的一个较大项的副本。

**程序 5.18　快速显示树的函数**

该递归程序跟踪树的高度并使用该信息确定显示出树的表示，这种表示我们能用于调试树处理程序（参见图 5.29）。假设的是在节点的项是 Item 类型的，operator<<通过重载而被定义。

```
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);
  }
```

图 5.30 是一个显式树结构的例子，该结构由程序 5.19 构造。构造一个诸如这种的递归数据结构，可能最好是通过扫描数据来找到最大值，我们就曾经在程序 5.6 中这样做过，其原因是树结构给我们完成其他操作的灵活性。正是这些我们用于构造联赛的操作成了一个重要的例子：给定两个联赛，我们可以在恒定时间内把它们组合成为一个联赛，方法是创建一个新的节点，并且令它的左链接指向其中一个联赛，右链接指向另外一个联赛，挑出两项中较大的一项（在两个给定联赛的根部）作为组合联赛中的最大项。我们同样可以为增加项、删除项以及进行其他操作而去考察算法。我们并不打算在这里考察这样的操作，原因在于我们要在第九章考察类似的有这种灵活性的数据结构。

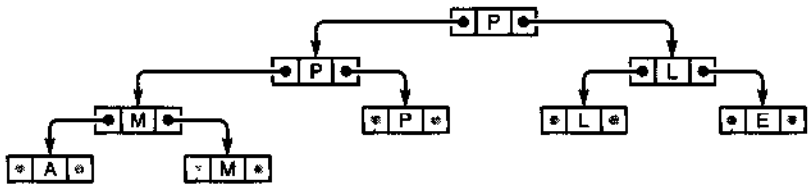



**图 5.30　找最大值（联赛）的显式树**

这个图描述了程序 5.19 构造的的显式树的结构，从输入 A M P L E 开始。该数据项是在叶子中。每一个内部节点都有一个在两子节点中更大项的副本，因此通过归纳法，最大项就在根部。

实际上，我们在 4.6 节讨论过的概括性的队列 ADT 中的几种的基于树的实现是整本书中主要讨论的主题。特别地，从第十二章到第十五章许多算法都是基于二叉搜索树的，这些二叉搜索树都是与折半查找对应的显式的树，处在一种类比于图 5.30 的显式结构和图 5.6 的递归找最大值算法之间的关系中。实现和使用类似结构的具挑战性的地方在于，我们要保证在一系列的插入、删除以及其他操作后仍然能够保持我们的算法的效率。

**程序 5.19 联赛的构造**

该递归函数是把一个数组 a[l]，…，a[r]分解成两个部分 a[l]，…，a[m]和 a[m+1]，…，a[r]，为这两部分递归地构造联赛，并且通过在新的节点设置链接到递归创建的联赛，并且在两个递归创建的联赛的根部设置数组的项值到更大的一项，从而为整个数组构建一个联赛。

```
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;
  }
```

我们创建一个二叉树的第二个程序的例子就是，在 5.1 节中的前缀表达式求值程序 5.4 进行修正，以便创建一个表示前缀表达式的树的表示，而不只是简单的对它进行求值（参见图 5.31）。程序 5.20 使用了和程序 5.4 一样的递归方案，但是程序 5.20 的递归函数返回一个连到一棵树上的链接，而不是一个值。我们为表达式中的每个字符创建一个新的树节点：这些与操作符相对应的节点，都有到它们的操作数的链接，叶子节点包括了输入到表达式中的变量（或常量）。

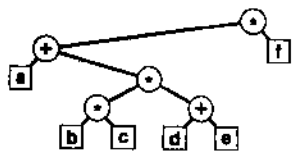



**图 5.31 分析树**

这棵树由程序 5.20 构造，前缀表达式是*+a**bc+def。它是表示一个表达式的自然方法：每一个操作数都在叶子中（我们在图所画作外部节点），每一个操作符都被应用到由一个包含该操作符的节点的右子树和左子树表示的表达式中。

诸如编译器这样的翻译程序经常为程序使用类似的内部树表示，因为这些树在许多用途上都是有用的。比如，我们可以对应于承担值的变量去设想操作数，我们也可以产生机器代码来计算由后序遍历的树所表示的表达式的值。又或者，我们可以使用树显示出表达式：用中序遍历显示表达式的中缀形式或是用后序遍历显示表达式的后缀形式。

我们在这几节中考察的一些例子，目的是要引入这样的概念：我们可以用递归程序创建与处理显式的链接树结构。要有效地做到这一点，我们需要考察不同算法的性能、

可选表示、非递归可选方案的操作以及许许多多的其他细节。然而，我们将推延对树处理程序的更为详尽的考察，直到第十二章再考虑，理由是我们在第七章到第十一章使用树基本上是为了描述的目的。我们在第十二章返回到显式树的实现，因为这些实现形成了我们要考察的第十二章到第十五章的大量算法的基础。

**程序 5.20 分析树的构造**

使用与我们用于估量前缀表达式的策略同样的策略，这个程序从一个前缀表达式构建一棵分析树。为简单起见，我们假设操作数是一个字符。每一个递归函数的调用都创建一个新的节点，其中以输入的下一个字符作为记号（token）。如果该记号是操作数，我们返回到新的节点；如果该记号是操作符，我们设置左指针和右指针指向为两个参数（递归的）创建的那棵树上。

```
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;
  }
```

## 练习

○ 5.85 修改程序 5.18 以输出一个画树的 PostScript 程序，必须以图 5.23 所使用的那种格式，但不带表示外部节点的方框。使用 moveto 和 lineto 来画线，并且使用用户定义的操作符/node {newpath moveto currenpoint 4 0 360 arc fill }def 来画节点。在这个定义之后，调用 node 在栈的坐标上画一个黑点（参见 4.3 节）。

▷ 5.86 写出一个计数二叉树的叶子的程序。

▷ 5.87 写出一个计数二叉树的节点数目的程序，该二叉树有一个外部节点与一个内部子节点。

▷ 5.88 写出一个计算二叉树的内部路径长度的程序（使用定义 5.6）。

5.89 当计算二叉树的内部路径长度时，确定由你的程序所做的函数调用的数目。通过归纳法证明你的答案。

● 5.90 写出一个递归程序，计算二叉树的内部路径长度，这个程序的运行时间与树的节点数目成比例。

○ 5.91 写出一个递归程序从联赛中删除带有给定键的所有叶子（参见练习 5.59）。

## 5.8 图遍历

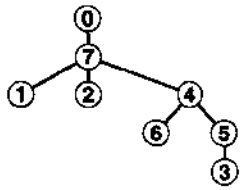
对于这一章的最后一个递归程序的例子，我们考察所有递归程序中最为重要的一个：递归图遍历，即深度优先搜索。这个系统的访问所有图中的节点的方法是一个对我们在 5.6 节考察的树遍历方法的直接一般化，该方法成为许多处理图的基本算法的基础，（参见第七部分）。它是一个简单的递归算法。从任意的节点 $v$ 开始，我们

- 访问 $v$；
- （递归地）访问每一个（未访问的）附于 $v$ 的节点。

如果图是连通的，我们最终可以到达所有的节点。程序 5.21 就是一个该递归过程的具体实现。

比如说，假设我们使用图 3.15 中的示例图所描述的邻接表表示。图 5.32 显示出在深度优先搜索这个图的过程里所做的递归调用，图 5.33 的左方的顺序描述了我们在图中跟随边所采取的方式。我们在图中跟随每一条边，在两个可能的结果当中就有一个：如果边把我们带到一个已经访问过的节点上，我们就忽略它；如果边把我们带到一个还未访问过的节点上，我们通过递归调用跟随它到那儿。我们用这种方式跟随的所有边的集合就为图形成了一个生成树（spanning tree）。

在深度优先检索和一般树遍历之间的区别（参见程序 5.14）在于我们需要明确地保障不会访问我们已经访问过的节点。在一棵树中，我们从未遇到这样的节点。实际上，如果这个图是一棵树，从根部开始的递归的深度优先搜索就与前序遍历等价。




```
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
```

**图 5.32 深度优先搜索的函数调用**

这个函数调用序列为图 3.15 的示例图组成了深度优先搜索。描述了递归结构（下面的图）的树被称为深度优先搜索树。

**程序 5.21 深度优先搜索**

要访问图中与节点 $k$ 相连的所有节点，我们将它标记为访问过的，然后（递归地）访问 $k$ 的邻接表中的所有未访问过的节点。

```
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);
  }
```

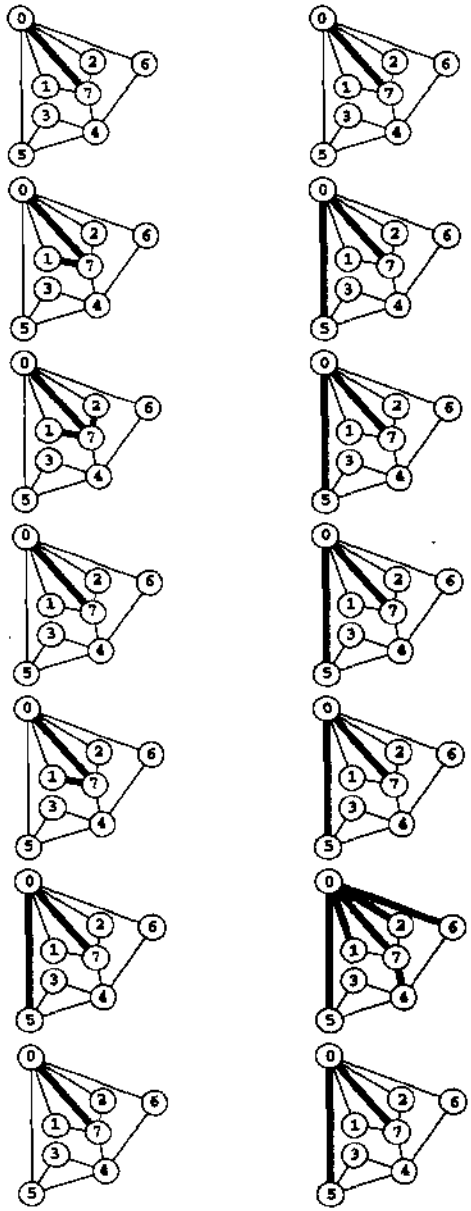


**图 5.33 深度优先搜索与宽度优先搜索**

深度优先搜索（左边）从节点移动到节点，只要它在给定的节点已经尝试过每一种可能性，就倒退到前一个节点去尝试下一个可能性。宽度优先搜索（右边）在移动到下一个节点前，充分尝试一个节点上的一切的可能性。

**性质 5.10** 深度优先搜索要求与 V+E 成正比的时间，其中在图中有 V 个顶点，E 条边，使用邻接表表示。

在邻接表表示中，有一个表节点与每一个图中的边相对应，一个表头指针与图中的每一个顶点相对应。深度优先搜索最多一次就接触到它们所有的。

因为由一个输入边的输入序列构建一个邻接表表示也要花费与 V+E 成比例的时间（参见程序 3.19），深度优先搜索给我们一个线性时间的解决办法来用于第一章的连通性问题。然而，对于大型的图，联合查找的办法可能更为可取，因为表示一个整图要花费与 E 成比例的空间，但联合查找办法只要花费与 V 成比例的空间。

就如我们在树遍历中所做的，我们可以定义一个使用显式栈的图遍历方法，如同图 5.34 所描述的那样。我们可以考虑一个包容两个项的抽象的栈：一个节点和一个指向该节点邻接表的指针。栈被初始化为第一个节点并且在指针被初始化为邻接表中的第一个节点，深度优先搜索算法与进入循环是等价的，在循环中我们访问栈顶端的节点（如果该节点仍然被访问）；存储当前邻接表指针引用的节点；更新对下一个节点的邻接表引用（如果节点在链接表的末尾就弹出项）；然后为存储的节点压入一个栈项，指向在它的链接表中的第一个节点。

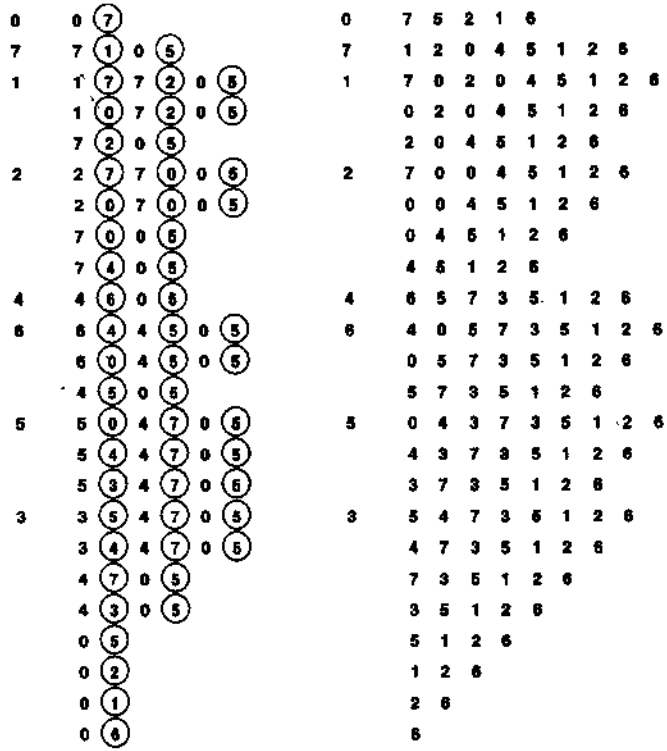


**图 5.34 深度优先搜索栈动态**

我们可以把支持深度优先搜索的下推栈想像为包含一个节点和指向该节点的邻接表（由一个带圈的节点指示）。这样的话，我们就从栈的节点 0 开始，带有对它的表中的第一节点，即节点 7 的引用。每一行指示出弹出栈的结果，把对下一个节点的引用推入已经被访问的节点表中，再把一个项推到未被访问的节点的栈中。可以选择的是，我们可以考虑把整个过程看作简单的把所有与任何未被访问的节点邻接的节点推到栈上（右边）

可选的方案是，就如我们为树遍历所做的，我们可以考察栈来包含指向节点的链接。栈初始化到开始节点，我们进入一个循环，在该循环中我们访问栈顶端的节点（如果它还未被访问），然后压入所有与其邻接的节点到栈上。图 5.34 说明了所有这些办法都与我们样例图的深度优先搜索是等价的，而且这种等价实际上具有一般性。

访问顶端节点并压入其所有邻居算法是个简单的深度优先搜索的公式，但从图 5.34 清楚看到这有一个缺点，即在栈中遗留每一个节点的多个副本。即使是我们去测试每一个将到栈上的节点是否已经被访问过，若被访问过就避免把节点放到栈上，也是如此。要避免这个问题，我们可以使用一个栈实现，该实现通过使用一个“忘记旧项”策略来避免重复，因为最接近栈顶端的副本常常是最先被访问的，所以其他的就很容易被弹出。

在图 5.34 中说明了用于深度优先搜索的栈动态依靠于每一个邻接表上的节点，该邻接表就按它们于表中出现的同一顺序在栈上结束。当一次推入一个节点时，对于已经给定的邻接表要得到这个顺序，我们将不得不首先推入最后的节点，然后是次后的节点，以此类推。更进一步，要限制栈的大小为顶点的数目，而同时按与深度优先搜索中相同的顺序访问节点，我们需要用“忘记旧项”策略以使用一个栈原则。如果按与深度优先搜索中相同的顺序访问节点对我们来说并不重要，我们可以避免以上的复杂方法，并直接用公式表达一个非递归的基于栈的图遍历方法：利用初始化到开始节点的栈，我们进入一个循环，在这循环当中，我们访问栈中顶端的节点，然后继续通过它的邻接表，把每一个节点推到栈的顶端（如果该节点还未被访问），用“忽略新项”策略使用一个不允许重复的栈实现。该算法以一种类似于深度优先的方式访问图中所有的节点，但它却不是递归的。

**程序 5.22　宽度优先搜索**

要访问图中与节点 $k$ 相连的节点，我们把 $k$ 放到一个 FIFO 队列中，然后进入一个我们从这个队列中得到下一节点的循环，并且，如果它未被访问，那么就访问它并把所有的未访问点压入到它的邻接表上，如此继续，直到队列为空。

```
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);
        }
  }
```

前几段中的算法是值得我们注意的，原因是我们可以使用任意的一般化的队列 ADT，

并且访问图中的每一个节点（并产生一棵生成树）。例如，如果我们使用队列而不是栈，那么我们就有宽度优先搜索，该搜索可与树中的层次顺序遍历相类比。程序 5.22 就是这个方法的实现（假设我们使用类似于程序 4.12 的一个队列实现）；操作中的一个算法的例子在图 5.35 中描述。在第六部分，我们将考察大量的基于更复杂的一般化队列 ADT 的图算法。

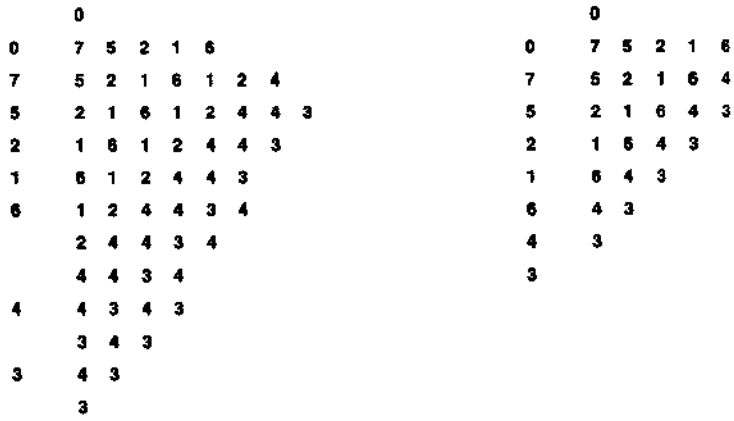



**图 5.35 宽度优先搜索队列动态**

我们从队列上的 0 开始，然后得到 0，访问它，并且把节点按照它们放入队列中的顺序放到邻接表 7 5 2 1 6 上。然后我们得到 7，访问它，并且把节点放到它的邻接表上，如此继续。用一个“忽略新项”策略（右边）避免重复，我们得到同一结果，而不需要任何无关的队列项。

宽度优先搜索和深度优先搜索都访问图中所有的节点，但是它们的实现方式大相径庭，如图 5.36 演示的那样。宽度优先搜索类似于一支搜索军队铺开来去覆盖领土；深度优先搜索类似于一个搜索者尽可能深入地去调查未知的地域，只有在碰到死胡同时才回头。这些都是有意义的解决问题的基本范型，应用于许多计算科学领域，而不仅仅是图搜索。

## 练习

5.92 通过画出与图 5.33（左边）与图 5.34（右边）对应的图，展示递归深度优先搜索如何访问图中的节点，其中该图为边序列 0-2，1-4，2-5，3-6，0-4，6-0 和 1-3（参见练习 3.70）而构造。

5.93 通过画出图 5.33（左边）与图 5.34（右边）对应的图，展示基于栈的深度优先搜索如何访问图中的节点，其中该图为边序列 0-2，1-4，2-5，3-6，0-4，6-0 和 1-3 而构造。

5.94 通过画出图 5.33（右边）与图 5.35（左边）对应的图，展示（基于队列的）宽度优先搜索如何访问图中的节点，其中该图为边序列 0-2，1-4，2-5，3-6，0-4，6-0 和 1-3 而构造。

○ 5.95 为什么在性质 5.10 中运行时间被引作 V+E，而不是简单的一个 E？

5.96 通过画出与图 5.33（左边）与图 5.35（右边）对应的图，展示当实施一个“忘记旧项”策略时，基于栈的深度优先搜索如何访问图 3.15 中的文本的样例图的节点。

5.97 通过画出图 5.33（左边）与图 5.35（右边）对应的图，展示当实施一个“忽略新项”策略时，基于栈的深度优先搜索如何访问图 3.15 中的文本的样例图的节点。

▷ 5.98　为以邻接表表示的图实现一个基于栈的深度优先搜索。

○ 5.99　为以邻接表表示的图实现一个递归的深度优先搜索。

## 5.9　前景展望

递归位于早些时候的对计算本质的理论研究的核心。研究如何去把可用计算机解决的问题从不可用计算机解决的问题分离出来，递归函数与程序就在这种数学研究中起到了一个中心作用。

当然，要在如此简短的讨论中达到如同对树与递归的讨论一样深远的程度，这是不可能公平处理到的。许多最佳的递归程序的例子都将是我们整本书中的焦点所在——分治法算法与递归数据结构，它们都已经被成功运用于解决广泛的问题。对于许多的应用来说，没有理由要去超出一个简单直接的递归实现；对于其他的应用，我们将考察候选的非递归实现和由底向上实现的来源出处。

在本书中，我们的兴趣在于递归程序与数据结构的实际运用方面。我们的目标是开发出一种能产生精致高效实现方案的递归。要实现这种目标，我们需要对某些简单程序的危险特别关心，这种简单程序能导致函数调用指数级数目或是不可能的深度嵌套。除去这些缺陷，递归程序与数据结构都是很有吸引力的，因为它们常常为我们准备了归纳的参数，这些参数能够使我们相信我们的程序是正确且高效的。

我们在整本书中使用树，既能帮助我们理解程序的动态性质，又能帮助我们理解动态数据结构。第十二章到第十五章，集中了关于显式树结构的控制内容。在本章中描述的性质为我们提供了基础的信息，如果我们要高效使用显式树结构就需要这些信息。

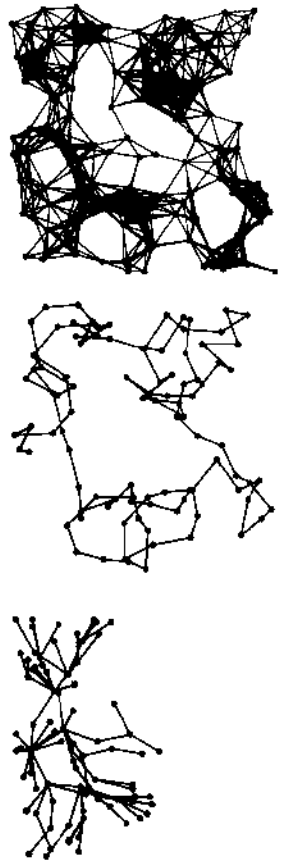


**图 5.36　图遍历树**

该图展示了深度优先搜索（中间图）和宽度优先搜索（底部图），在一个大图的搜索的运行途中（上部图）。深度优先搜索从一个节点蜿蜒到另一个节点，所以大部分的节点都与另外两点相连。与之相对，宽度优先搜索扫过整个图，访问与一个给定节点相连的所有节点，再移动，所以若干节点都与多个节点相连。

除了它在算法设计的中心作用，递归并非一种万能药。就如我们在树遍历和图遍历中所发现的那样，当我们有多重的计算任务要完成时，基于栈的（固有地递归）算法不是惟一选择。一种用于许多问题的有效的算法设计技巧就是使用一般化队列实现而不是栈来给我们根据主观标准而非就近选择来挑拣下一个任务的自由。有效支持这类操作的数据结构和算法是第九章的首要主题，当我们考察第七部分的图算法时，我们将遇到许多关于它们应用的例子。

## 第二部分参考资料

有大量的关于数据结构的入门性教科书。比如，Standish 所著的书以一种比本书更为从容的风格介绍了链接结构、数据抽象、栈和队列、存储分配以及软件工程概念。当然，Kernighan-Ritchie 和 Stroustrup 经典著作分别是关于 C 语言实现和 C++语言实现的详尽信息的无价的来源。由 Meyers 所著的书同样提供了关于 C++语言实现的有用信息。

PostScript 的设计者也许并未预想到他们的语言成为学习基础算法与数据结构的人员的极大的兴趣。但是，该语言并不难学，而且参考手册是完整的和易得的。

在 Hanson 所著的书中，详尽地描述了客户接口实现范型，而且伴有大量的例子。要是编程者想为大型系统写出既没有 bug 又可移植的代码，这本书是不可忽视的。

Knuth 的书，特别是第一卷到第三卷，保持了关于基础数据结构的性质的权威来源。Baeza-Yates 和 Gonnet 有更新的信息，有广泛的参考书目作支持。Sedgewick 和 Flajolet 详尽地介绍了树的数学性质。

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 Compurer 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. Flajoler, *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.

# 第三部分

# 排 序 算 法

# 第六章

# 基本排序方法

在深入研究排序算法之前，我们先学习几种基本排序方法，这些排序方式适用于一些数据较少或者其数据具有特殊结构的文件。要详细学习这些基本排序算法的原因有几点。第一，通过学习这些基本排序方式，我们可以学习排序算法的相关术语和基本机制，从而为学习更复杂的算法奠定基础。第二，在许多实际应用中，这些简单的排序方式的效率比那些更完善更普遍适用的排序方法要有效。第三，一些简单的排序方式能扩展成更普遍适用的排序方式，或者有利于提高更复杂的算法的效率。

本章的目的不只是简单地介绍这些基本方式，还要为将在后面章节学习的排序算法构建框架。我们将学习排序算法应用中的各种重要因素，观察不同类型的输入文件，比较各种排序方式及学习它们各自的特性。

我们首先用一个简单的驱动程序来检查各种排序方式，以了解所要遵循的约定。我们还要讨论各种排序方式的基本特性，这对于评价特定应用程序中所选的算法是很重要的。我们将先观察以下三种排序方式的实现：选择排序、插入排序和冒泡排序。我们会深入讨论这些算法的特性。接着，我们观察 Shell 排序，这个排序法可能不算是基本排序方式，但它很容易实现，并且与插入排序有很大关联。在讲完 Shell 排序法的数学特性后，我们将沿着第三、四章的讨论思路，深入研究开发数据类型接口和实现的主题，通过对练习中的数据文件进行排序来扩展基本算法。然后，我们讨论不与数据和链表排序直接相关的排序方法。本章还会讨论一种特殊的排序方法，这种方法适用于键值限制在一个很小范围的时候。

在许多排序应用程序中，都会选择一个简单的算法。首先，一个排序程序通常只使用一次或几次。一旦我们对一组数据进行了排序，我们就不再需要在应用程序中使用相同的排序程序来处理这些数据了。如果一个基本的排序方法不比其他数据处理（如读入数据和读出数据）慢，就没必要寻找一个更快的方法。如果需要排序的数据不是很多（比如，只有几百个元素），我们就宁愿实现并运行一个简单的方法，而不是将精力耗费在系统排序方法的接口或实现和调试复杂的方法上。其次，基本方法通常适用于小文件（比如，几十个元素），复杂的方法会导致很高的开销，以至于比简单的方法更慢。因此除非我们想对大量

的小文件进行排序，否则这个问题是不值得考虑的，不过有这种需求的应用程序也是常见的。另一种相对容易排序的文件是那些已经基本排序好的（或者已经排序好了！）或者那些包含大量重复键的文件。我们将会看到，在对这些结构严密的文件进行排序时，有几种基本方法特别有效。

通常，在这里讨论的基本排序方法，对 $N$ 个随机组合的数据进行排序的运行时间与 $N^2$ 成正比。如果 $N$ 比较小，运行时间就会比较适中。正如刚才所说的，对于小文件和其他一些特殊情况，基本方法的效率要比复杂方法的高。不过本章讨论的方法不适用于大的随机组合的文件，因为即使使用最快的计算机，运行时间仍然会很巨大。例外的是 Shell 排序法（参见 6.6 节），在对 $N$ 个数据进行排序时，如果 $N$ 较大，则这种排序法只需花费比 $N^2$ 少得多的步骤，并且可以证明它适用于中等大小的文件和其他一些特殊情况。

## 6.1 游戏规则

在讨论各个排序方法之前，先了解排序算法的通用术语和基本假设是很重要的。我们将讨论文件（file）的排序方法，这些文件的数据项（item）包含键（key）。这些概念都是从现代编程环境中抽象出来的。键是数据项的一个部分（通常是一个小部分），用于控制排序。排序方法的目的是重新组合数据项，使其键根据一些定义良好的排序规则（通常是数字或字母顺序）整齐排列。不同应用程序的键和数据项会有不同的特性，不过排序程序只需关心如何将键和相关信息排列整齐。

如果被排序的文件适合放在内存中，则排序方法被称为“内部（internal）排序”。从磁带或磁盘上对文件进行排序被称为“外部（external）排序”。这两种方法的主要区别在于，内部排序可以很容易地访问任何数据项，而外部排序必须顺序访问数据项，或至少在大的数据块中是如此。我们将在第十一章中介绍几种外部排序方法，但是我们所讨论的大部分算法都属于内部排序方法。

我们将讨论数组和链表。数组排序和链表排序同样重要：在讨论算法时，我们一定会遇到一些适合顺序存储的基本任务，同样也会遇到适合链式存储的任务。一些典型排序方法在处理数组和链表时同样有效，而一些就只适用于其中一种。其他一些存储上的限制有时也是很重要的。

我们首先讨论数组排序。程序 6.1 举例说明了我们在执行时所要使用的一些约定。包括一个驱动程序，它通过读取标准输入数据或产生随机整数列（由某一整数生成规则指定）来建立数组；然后调用一个排序函数，将数组中的数据排好顺序；接着输出排好序的结果。

**程序 6.1 使用驱动程序的数组排序例子**

本程序举例说明了在执行基本数组排序时的约定。Main 函数中初始化了一个整型数组（使用随机数或读取标准输入），接着调用 sort 函数来对数组进行排序，最后输出已排好序的结果。

模板函数可对任何类型的已声明了比较关系的数据进行排序。这里使用的 sort 函数是一种插入排

序法（参看 6.3 节的详细描述、例子和改进算法）。该算法使用了一个模板函数，来比较两个数据项，如果第二项小于第一项，就将两项位置交换。

我们可以通过对“<”操作符进行声明来实现对不同类型的数据进行排序，而根本不需要修改 sort 函数（参看 6.7 节）。

```
#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;
  }
```

通过第三章、第四章我们知道，有很多方法可用于调整排序程序，使之适用于其他类型的数据。我们将在 6.7 节中详细讨论如何使用这些方法。程序 6.1 所用的 sort 函数是一个模版式程序，它只是通过第一个参数进行排序，对数据进行了一些简单的操作。通常，通过这种方法，我们可以使用相同的代码对其他类型的数据进行排序。例如，如果程序 6.1 的 main 函数中产生、存储、打印随机数列的代码改为处理浮点数而不是整数，我们根本不需要修改 sort 函数。为了提供这种适应性（同时要明确地声明那些存储数据项的变量），我们的 sort 实现将对 Item 数据类型进行处理。在这里，我们可以将 Item 当作整数 int 或者浮点数 float；在 6.7 节，我们将详细讨论数据类型的处理机制，使用在第三章、第四章讨论

过的方法来使排序程序可适用于任意类型的数据项，包括浮点数、字符串和其他不同类型的键。

我们可以用本章不同的数组排序函数及第七章到第十章的数组排序函数取代 sort 函数。这些函数中都假定存储的数据项是 Item 类型，都包括三个参数：数组、数组的左界和右界。它们都根据“<”操作符，来比较数据项中的键，并调用 exch 或 compexch 函数来交换数据项。为了区分排序算法，我们会对不同的排序算法起不同的名字。对它们进行改名，修改驱动程序或者在像程序 6.1 那样的客户端程度中用函数指针调用排序算法都很简单，无需改变排序算法中的任何代码。

这些约定使我们可以检验多种简练的数组数据排序算法。在 6.7 节和 6.8 节中，我们将讨论一个关于如何对更普通数据实现排序和对各种不同类型数据实现排序的示例程序。虽然我们要讨论许多方面内容，但我们的焦点将放在算法结果上，我们现在就开始讲述。

程序 6.1 中所使用的排序方法是一种插入排序，这种方法将在 6.3 节中详细讨论。由于它只使用了比较—交换操作，所以它是一种非适应性排序，即它所执行的操作的次序与数据的顺序无关。相反地，适应性排序是指，执行时调用各函数的不同次序依据于比较所得的结果（调用<操作符）。非适应性排序很重要，因为它们适合应用于硬件实现中（参看第二章）。不过，我们讨论的排序方法大部分是适应性的。

通常，排序算法的执行时间是最重要的参数。在 6.2~6.4 节中讨论的选择排序、插入排序和冒泡排序对 $N$ 个数据项进行排序时，执行时间均与 $N^2$ 成正比，这点会在 6.5 节中讨论。而在第七章到第十章讨论的进一步排序方法的执行时间与 $N \log N$ 成正比，不过在 $N$ 较小或数据结构较特殊的情况下，这些排序算法的效率通常比不上我们这里讨论的基本方法。在 6.6 节中，我们将介绍一个更先进的方法（Shell 排序），它的执行时间可与 $N^{3/2}$ 成正比或者更低。在 6.10 节中，将介绍一种特别的方法（索引排序），对某些类型的键，这种方法的执行时间可达到与 $N$ 成正比。

上一段的分析结果是通过列举算法所执行的基本操作（比较和交换）而得到的。如 2.2 节所讨论的，我们还需要考虑这些操作的花费，一般集中在执行最频繁的操作上（算法的内部循环）。我们的目的是执行有效合理的算法。为了达到此目的，我们不仅要避免内部循环中不必要的花费，还要设法减少内部循环中的指令。一般，减少应用程序花费的最好方法是改用一种更有效的算法；第二种是减缩内部循环。对这两种方法，我们在排序算法中都会详细讨论。

排序算法所要消耗的额外内存空间是我们需要讨论的第二个重要特征。基本上，有三种类型的算法：一种是除了可能使用一个小堆栈或表外，不需使用任何额外内存空间的排序方法；一种是使用链表或指针、数组索引来代表数据，因此存储这 $N$ 个指针或索引需要额外的内存空间；第三种是需要额外的空间存储要排序的数据的副本。

我们经常会对一些有重键的数据进行排序——我们还可能对同一组数按不同键进行多次排序。在这种情况下，我们需要知道所使用的排序方法是否具有以下特性：

**定义 6.1**　如果排序后文件中拥有相同键的项的相对位置不变，这种排序方式是稳定的。

例如，将一个按字母顺序排列的包含毕业年份的学生名册重新按年份进行排序，稳定的排序方法得到的列表中，在同一个班级的学生仍然按字母顺序排列，而一个不稳定的排序方法所得到的列表可能没有丝毫原来按字母顺序排列的痕迹。图 6.1 就是这样一个例子。经常，不熟悉稳定性的人第一次碰见这种情况时，会很惊讶于不稳定的排序算法似乎把数据弄得很乱。

本章所讨论的基本排序方法有几种（不过不是所有）是稳定的。相反，我们在以后章节讨论的较复杂的排序方法中大部分（不是所有的）是不稳定的。如果稳定性是必需的，我们可通过在排序前为各键添加小索引或使用其他方法扩展排序的键，以此强制排序方法稳定。这样做相当于结合图 6.1 的两个键进行排序；改用一个稳定的算法会比较好。我们很容易会把算法的稳定性当作是必然的，事实上，如果不使用额外的事件或空间，我们后面章节讨论的排序方法中很少可以达到稳定。

上面提过的，排序程序通常使用两种方法之一来访问数据项：访问键来进行比较；或者访问整个数据项来移动。如果需要访问的数据项比较大，通过使用间接的排序方法避免所使用的访问空间太大是比较明智的：我们不是直接对数据项进行重新安排，而是使用一组指针（或索引），使第一个指针指向最小的数据项，第二个指针指向次小的数据项，以此类推。我们可以将键保存在数据项中（如果键比较大）或者保存在指针中（如果键比较小）。我们可以在排序后重新安排数据项，不过通常不需要这样，因为我们已经可以按顺序访问这些数据（间接地）了。我们会在 6.8 节中讨论间接排序。

| | |
|---|---|
| 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 |

**图 6.1　稳定排序方法的例子**

数据可以任意键进行排列。假设，开始时，数据按第一个键排序（顶部列表）。然后按第二个键排序，不稳定的排序方法不会保持第二个键相同的记录的相对位置（中间列表）。而稳定的排序方法保持它们的相对位置（底部列表）。

## 练习

▷ 6.1　有一种儿童排序玩具，包含 $i$ 张卡，每张卡可一一对应挂在某个钉子上，$i$ 从 1~5。写一个算法将卡片挂到合适的钉子上，假设不可以从卡片上看出它与哪个钉子对应（必须通过尝试是否适合判断）。

6.2　有一种纸牌游戏，要求将桌子上的纸牌按花色排序（按黑桃、红心、梅花、方块这样的顺序），每种花色中又按大小排序。叫几个朋友来玩这个游戏（相互洗牌），记下他们所使用的方法。

6.3　思考如何将牌排序，条件是所有牌正面朝下排成一行，所能做的只是随意地检查两张牌的值，需要的话就将它们交换位置。

○ 6.4 思考如何将牌排序，条件是所有牌叠成一堆，只能察看最上面两张牌的值，必要时进行交换，然后将最上面一张牌移到最底。

6.5 列出对三个数据进行排序时所进行的三个比较-交换操作序列（参看程序 6.1）。

○ 6.6 列出对四个数据进行排序时所进行的五个比较-交换操作序列。

● 6.7 写一个客户端程序来检查所使用的排序方法是否稳定。

6.8 在 sort 操作后检查数据是否排了序并不能保证排序算法正确，为什么？

● 6.9 写一个本地的程序对不同大小的文件进行 sort 操作，计算每次运行所使用的时间，然后将平均运行时间打印出来或图示出来。

● 6.10 写一个检验程序对可能在现实中出现的复杂或特殊的数据执行 sort 程序。例如，原来已排好序的文件、倒序的文件、所有键都是相同的文件、只有两种不同值的文件和大小为 0 或 1 的文件。

## 6.2 选择排序

以下是其中一种简单排序算法的执行过程。首先，选出数组中最小的项，将它与数组第一个成员交换位置。然后选出次小的项，将它与数组第二个成员交换位置。按这种算法一直做下去，直到整个数组排完序。这种排序方法叫做选择排序，因为它是通过不断选出剩余数据项中最小项来实现的。图 6.2 显示了该算法的一个示例。

程序 6.2 是依照先前约定的一个选择排序程序。内部循环只是执行了将当前数据项与先前找到的最小项比较的操作（加上其他必需的代码，包括将当前数据项索引递增和检查它是否超出了数组的界限）；要更简单是很困难的了。移动数据项的操作是在内部循环以外执行的；每次交换都将一个成员放在它最终的位置上，所以交换的次数是 $N$-1（最后一个成员不需要交换）。因此，执行的时间由比较操作的数目决定。在 6.5 节中，我们会得到这个数目与 $N^2$ 成正比，并且会进一步讨论如何预知整个运行时间和如何将选择排序与其他基本排序方法进行比较。

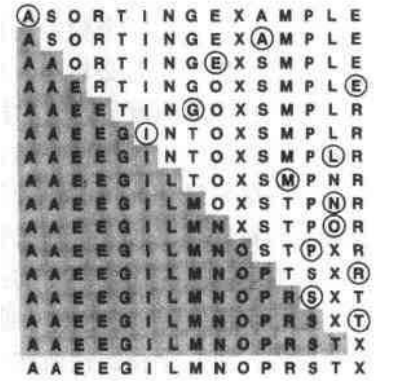



**图 6.2 选择排序示例**

本例的第一步没有执行任何操作，因为数组中没有一个项比在左边的 A 小。第二步中，另一个 A 是剩余数据项中最小的项，所以将它与在第一位的 S 交换位置，接着，第三步中接近中间的 E 与第二位的 O 交换位置；然后，第四步，另一个 E 与在第四位的 R 交换位置，依此类推。

**程序 6.2 选择排序**

i 的取值范围为 l-r-1，将 a[l]与 a[i]，…，a[r]中最小的项交换位置。随着 i 的取值范围为从左到右遍历，在它左边的数据项均已处在数组正确的位置中（而且不会再被访问），所以当 i 到达了最右边时数组就排好序了。

```
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;
      exch(a[i], a[min]);
    }
}
```

选择排序有一个缺点，它的运行时间和文件中已有的排序的关系很少。从文件中选出最小元素的每步操作过程，并没有给出下一步要找的最小项的位置的相关消息。例如，使用该排序方法的用户可能会很奇怪地发现，该程序对已排好序的文件或者各数据项都相同的文件排序所花的时间与对随机排列的文件排序所花的时间基本相同。我们以后会看到，其他方法会比较好地利用输入文件数据项原有序列。

虽然选择排序比较简单并且执行时间上具有强迫性，但它对某一类重要文件的排序效率要比其他方法好：对于数据项比较大，键又比较小的文件，应该选择该方法。对这种文件进行排序，移动数据所花费的时间要比比较数据的时间大很多，而其他算法移动数据的步数都比选择排序要多（参看 6.5 节的性质 6.5）。

**练习**

▷ 6.11　按图 6.2 的方法，显示选择排序对下面例子的排序过程：

E A S Y Q U E S T I O N

6.12　选择排序中，最多需要执行几次交换操作，包括所有特别的数据。平均次数又是多少？

6.13　给出一个含有 $N$ 个元素文件的例子，使执行插入排序时，使 a[j]<a[min]的检测结果为 false 的次数最大化（因此 min 要更新）。

○ 6.14　选择排序是否稳定？

## 6.3　插入排序

在桥牌中人们通常使用的排序方法是，每次只考虑一张牌，将牌插入已经排了序的牌的适当位置中（保持排序）。在计算机应用中，为了插入新数据，先将较大的数据项一个个向右移动，然后将新数据插入空位中。程序 6.1 中的 sort 程序就是使用了这种方法，这种排序方法称为插入排序。

和选择排序一样，排序过程中当前数据项左边的数据项都是排好序的，不过它们不是处于最终的位置上，因为之后它们可能还需要进行移动来为更小的数据腾出空间。不过当索引移到最右边时，数组就完全排好序了。图 6.3 显示了该

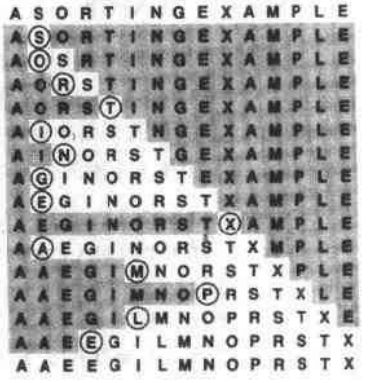



**图 6.3　插入排序示例**

在插入排序的第一步中，在第二位的 S 比 A 大，所以它不需要移动。第二步，第三位的 O 与 S 交换位置，AOS 成为排好序的序列，以此类推。图中，既没有打阴影也没打圈的元素都向右移了一位。

操作的一个示例。

程序 6.1 中执行的插入排序是很直接的。我们现在考虑三种方法来对它进行改善，以举例说明在执行时常会碰见的问题：我们希望代码既简洁又有效，不过有时是很难兼顾两者的，所以我们必须尽量得出一个平衡。我们首先编辑一个普通的程序，然后通过一系列修改来尝试着改进它，检测每次修改的效率（和正确性）。

首先，当我们碰到一个不比当前键大的项时，就停止执行 compexch 操作，因为左边子序列是已排好序的。对应地，当条件 a[j-1]<a[j]为真时，我们可以中断程序 6.1 中的 sort 函数的内部循环过程。这个修改使插入排序成为可适应性排序方法，并使随机排列文件的排序时间减少了大概 1/2（参看性质 6.2）。

根据前一段所描述的改进方法，我们在两种情况下可终止内部循环——我们可以使用 while 循环改写代码，更清晰地反映该特性。一个更细致的考虑是，j>1 判断通常是无用的：事实上，它只有当所要插入的元素目前是最小的且要插入数组最前端时才为真。通常的改进方法是将要排序的数据放在 a[1]到 a[N]中，a[0]中存放一个附加键，该键和数组中最小的元素一样大或者更小。这样，就可以同时对不同情况进行监测，使内部循环更小，程序运行得更快。

增加附加键有时是比较难于实现的：可能最小的可能值难于确定，也可能调用的程序没有空间包含一个附加键。程序 6.3 显示了一个围绕这两个问题的插入选择示例：第一步直接将键最小的数据项放在第一位。接着，对剩下的数组进行排序，将数组中第一个且最小的项当作了附加项。我们应该尽量避免在代码中使用标志，因为直接的解题比较容易理解，不过当使用标志可以使程序更简单和效率更高时，我们也要注意。

**程序 6.3 插入排序**

这段代码是对程序 6.1 的一种改进，包括：（1）首先将数组中最小的元素放在第一位，这项可以当作是一个标志；（2）在内部循环中，它只是做一个简单的分配，而不是执行交换操作；（3）当元素被插入正确的位置后就中断内部循环。对每个 i，通过将 a[1]，…，a[i-1]中比 a[i]项要大的项向右移动一位，将 a[i]插入正确的位置，这样来实现对 a[1]，…，a[i]的排序。

```
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;
      }
  }
```

第三个改进方法是考虑将内部循环中无关的操作去掉。这点中注意的是对相同元素的多次交换是低效的。如果有两个或以上的交换，如：

```
t = a[j]; a[j] = a[j-1]; a[j-1] = t;
```

接着是

```
t = a[j-1]; a[j-1] = a[j-2]; a[j-2] = t;
```

等等。t 的值在两次操作序列间并没改变，我们将时间浪费在了将它存储起来，又在接下的交换中将它读出。程序 6.3 将较大的元素一个个向右移一位而不是进行交换，这样就避免了在这点上浪费时间。

程序 6.3 的插入排序法比程序 6.1 中的效率要高（在 6.5 节中，我们会了解到几乎是两倍的效率）。本书中，我们既关心细致高效的算法，也关心细致高效的执行过程。在这点上，基本算法差别不大——我们可将程序 6.1 的 sort 过程当作非适应性插入排序。对算法特性的清楚了解是对应用程序执行有效算法的最佳指导。

和选择排序不同的是，插入排序的运行时间和输入文件数据的原始排列顺序密切相关。例如，如果文件较大，并且键已经排好序（或几乎排好序），插入排序要比选择排序快。我们会在 6.5 节中更详细地比较这两种算法。

**练习**

▷ 6.15　参照图 6.3，显示使用插入排序对下列文件进行排序的过程：

E A S Y Q U E S T I O N

6.16　编写插入排序执行程序，其中内部循环使用 while 语句，并且可像上面提过的那样在两种条件之一成立时中断循环。

6.17　对练习 6.16 中 while 循环的每种条件，描述一个有 $N$ 个元素的文件，使当循环结束时条件总为假。

○ 6.18　插入排序是否稳定？

6.19　显示一个非适应性插入排序算法的执行过程，使用程序 6.3 中第一个 for 循环语句来寻找最小的元素。

## 6.4　冒泡排序

许多人首先学习的是冒泡排序，因为它非常简单：遍历文件，如果邻近的两个元素的顺序不对，就将两者交换，重复这样的操作直到整个文件排好序。冒泡排序最重要的特点是容易实现，不过它是否比插入排序或选择排序更容易实现还没有定论。冒泡排序的执行速度比另外两种排序方法的要慢，不过为了完备性我们还是对它进行简要的讨论。

假设我们都是将文件元素从右移到左的。第一步中，当遇到最小的元素时，将它与左边元素逐个交换，直到将最小的元素移到队列的最左边。然后第二步中，将第二小的元素

放到队列左边第二位中，以此类推。因此，一共需要 $N$ 步，冒泡排序实际上是一种选择排序，但需花费更多工作将每个元素放到合适的位置。程序 6.4 是一个示例，图 6.4 显示了一个例子的执行过程。

**程序 6.4 冒泡排序**

i 的取值范围为 l~r-1，内部循环（j）通过从右到左遍历元素，对连续的元素进行比较-交换操作，实现将 a[i]，…，a[r]中最小的元素放到 a[i]中。在所有的比较操作中，最小的元素都要移动，所以它就像是泡泡那样“冒”到最前端。和插入排序相似，随着索引 i 从左移到右，在 i 左边的元素都已处在最后的正确位置上。

```
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]);
  }
```

我们可以像在 6.3 节中对插入排序那样（参看练习 6.25），通过对内部循环进行改进来加快程序 6.4 的执行速度。实际上，通过比较代码，可以发现程序 6.4 和程序 6.1 中的非适应性插入排序方法很像。两者不同的是，内部 for 循环中，插入排序是从左边（已排好序的序列）向右移动，冒泡排序是从右边（未排好序的序列）向左移动。

程序 6.4 只使用了 compexch 操作，因此它是非适应性的。不过我们可以对它进行改进使它运行得更有效率：通过监测判断文件是否已排好序，当其中一步中已没有进行任何交换操作，即文件已经排好序，就可以中断外部循环。做了这项改进后，就会提高冒泡排序对某类数据的运行效率，不过通常它效率的提高比不上能中断内部循环的插入排序，如 6.5 节中详细讨论的那样。

**图 6.4 冒泡排序示例**

在冒泡排序中，键值小的元素移到了左边。随着从右到左排序，每个键和左边的进行交换直到遇见一个更小的。第一步中，E 和 L，P，M 交换，直到遇见 A；接着，A 移到文件的开始处，在已经在第一位的 A 前停住。冒泡排序中，第 i 步后，大小排行 i 的元素到达了它最终的位置，这点和插入排序一样，而其他的元素也不断地向它们最终的位置移近。

## 练习

▷ 6.20 参照图 6.4，显示运用冒泡排序对下列数据进行排序的运行过程：

E A S Y Q U E S T I O N

○ 6.21 给出一个例子，使执行冒泡排序时，所需要执行的交换操作最多。

○ 6.22 冒泡排序是否稳定？

6.23　解释冒泡排序为什么比练习 6.19 中描述的非适应性选择排序更优越。

● 6.24　做实验检验，当对冒泡排序加入了检测文件是否已排好序的操作时，对随机的 $N$ 个数据进行排序可以节省多少步。

6.25　对冒泡排序进行改进，使内部循环所要执行的操作最少。要确保所进行的“改进”不会减慢程序的执行效率。

## 6.5　基本排序方法的执行特性

选择排序、插入排序和冒泡排序在最坏和普通情况下都是二次算法，而且都不需要额外的内存空间。因此，它们的执行时间只相差一个常数因素，不过它们执行上有很大差别，如图 6.5~图 6.7 所举的例子。

通常，排序算法的运行时间是和算法所执行的比较次数、数据项移动或交换的次数成比例的。对于随机输入数据，对排序方法的比较包括对进行的比较操作和交换操作数目常数因子差异的比较及内部循环长度常数因子差异的比较。对于特殊的输入数据，排序方法运行时间的差异可能会超过一个常数因子。本节中，我们重点观察各项分析结果。

**性质 6.1**　选择排序使用大概 $N^2/2$ 次比较操作和 N 次交换操作。

通过观察图 6.2 中执行的例子可以很容易地得到这个结论。该例子是一个 $N*N$ 的表格，没有打阴影的字母需要进行比较操作。表格中大概有一半的字母没有打阴影——对角线以上的字母。对角线上 $N$-1 个字母（不包括最后一个字母）均需要执行一次交换操作。更精确地，通过检查程序代码可以知道，$i$ 从 1 到 $N$-1，每步都需要执行一次交换操作和 $N$-$i$ 次比较操作，因此一共执行 $N$-1 次交换操作和 $(N-1)+(N-2)+\cdots+2+1=N(N-1)/2$ 比较操作。这个结论与输入数据的类型无关；选择排序中惟一与输入数据有关的是更新 min 的次数。在最坏的情况下，该数目也是成二次方的；但在一般情况下，它只是 $O(N \log N)$（参看附录部分），所以我们可以认为执行时间和输入数据无关。■

**性质 6.2**　在一般情况，插入排序执行大概 $N^2/4$ 次比较操作和 $N^2/4$ 次半交换（移动）操作，而在最坏的情况下需要两倍的数量。

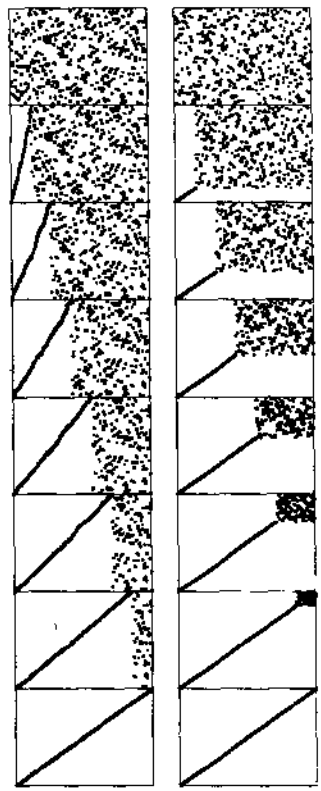


**图 6.5　插入排序和选择排序的动态特性**

上面是对某一随机数列进行排序执行过程的示例图，其中左边的是插入排序，右边的是选择排序。这里用小点表示要排序的 a[i]数据项。在排序前，小点是随便分布的；排序后，形成了一条从左下方到右上方的对角线。插入排序不能预知各点在数组中的最终位置；而选择排序则不会改变已操作了的点的位置。

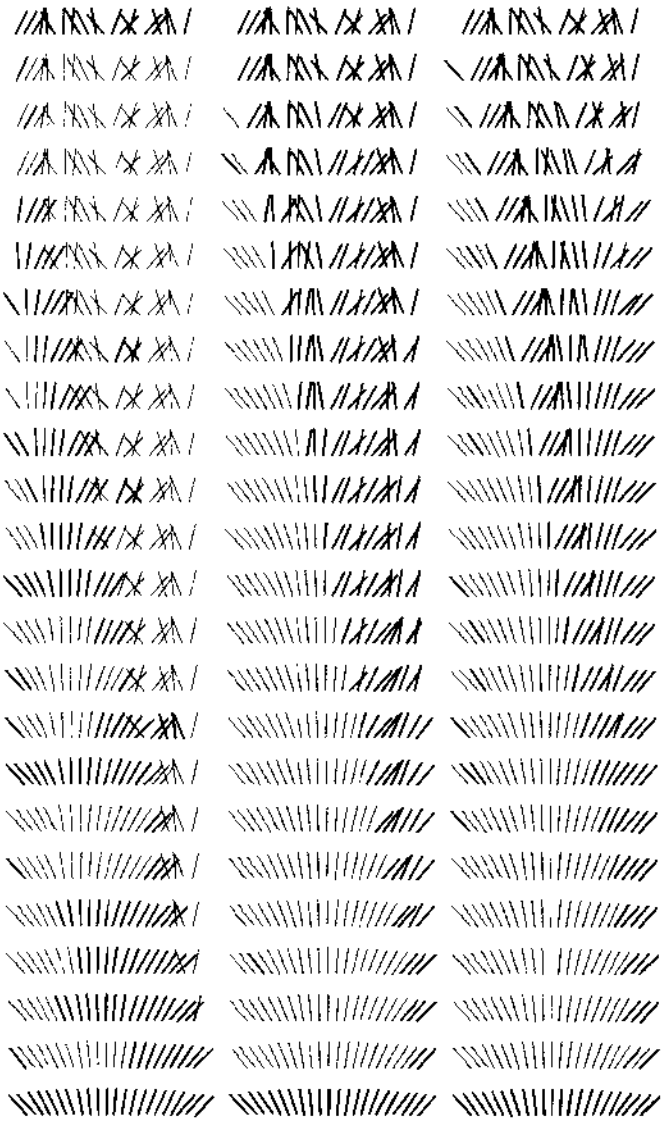


**图 6.6 基本排序方式的比较和交换操作**

该图显示了插入排序、选择排序和冒泡排序在对文件进行排序的过程中的差别。用线段集合表示要排序的文件，根据线段角度进行排序。每步中黑线表示需要访问的数据项，灰线表示不需要访问的数据项。对于插入排序（左边），每步中要插入的数据项平均要向前移动已排好序的数列的一半长度。选择排序（中间）和冒泡排序（右边）每步中都要访问所有尚未排序的数据来找出下一个最小的数据；两者不同的地方是冒泡排序将所遇见的相邻的顺序不对的数据项进行交换，而选择排序只是将最小的数据项交换到相应位置。这点不同使冒泡排序中尚未排序的数列逐渐趋向排好序。

如程序 6.3 执行过程，比较操作的数目和移动操作数目是一样的。和性质 6.1 一样，这

个结论可以很容易地通过图 6.3 的 $N*N$ 表格中算法的执行过程得出。这里，对角线以下的元素时要计算的——所有的，是最坏的情况。对于随机输入数据，平均情况下，每个元素要向后移动一半路程，所以对角线以下一半的元素要计算。■

**性质 6.3**　在一般情况和最坏的情况中，冒泡排序执行大概 $N^2/2$ 次比较操作和 $N^2/2$ 次交换操作。

冒泡排序中，第 $i$ 步中需要执行 $N-i$ 次比较-交换操作，证明的过程和选择排序一样。当算法被修改成可以在发现文件已排好序时终止，运行的时间就要根据输入数据而决定。如果文件是已经排好序的，就只需要一步；但如果文件是反序的，第 $i$ 步就需要执行 $N-i$ 次比较和交换操作。最坏情况下的效率和一般情况下的效率差别不大，虽然要证实这个结论的分析很复杂（参见“第三部分参考资料”）。■

虽然需要在对已基本排好序的文件进行操作时才能体现插入排序和冒泡排序的效率，但这两种排序方式对实际情况下经常会碰见的某些类型文件是很有效的。而一般的排序方式则不适用于这些文件。例如，使用插入排序对一个已排好序的文件进行操作。每个元素都能立刻确定在文件中的正确位置，整个执行时间是线性的。使用冒泡排序的情况一样，但使用选择排序的运行时间依然是二次的。

**定义 6.2**　文件中一对顺序不对的项称为一个倒置（inversion）。

要统计文件中的倒置数，我们可以对每个数据左边比它大的元素的数目进行累加（该数目表示了该数据项所对应的倒置数）。而在插入排序中，这个累加数目就是数据项要插入文件时所要向前移动的步数。有一点顺序的文件的倒置数要比完全混乱的文件的少。

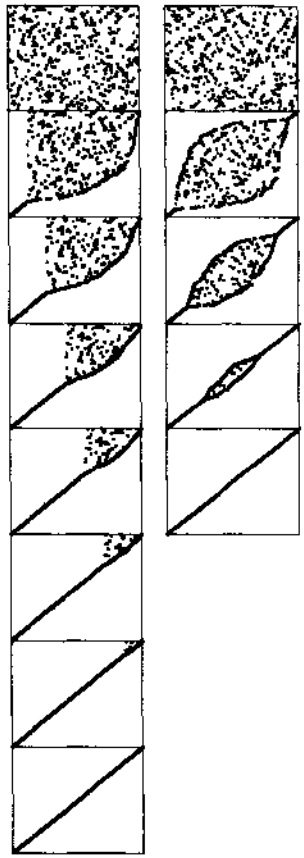


**图 6.7　两种冒泡排序的动态特性**

标准的冒泡排序（左边）执行过程和选择排序相似，每步中都将一个元素放到合适的位置中，不过它还对数组的其他部分进行某种程度的不均匀的排序。如果对数组不断替换从左到右和从右到左的扫描方向，这种冒泡排序称为晃动排序（右边），这种方法的执行效率更高（参看练习 6.30）。

某类文件中，每个数据项都接近于它排序后的最终位置。例如，某些人在打牌时，先按花色组织好牌，使牌接近于最终的位置，然后对牌一张张地进行考虑。我们会讨论一些使用同样方法的排序算法——这些算法在最先的步骤中产生一个部分排序的文件，使各个元素都接近于它的最终位置。插入排序和冒泡排序（不包括选择排序）是对该类型文件非常适合的方法。

**性质 6.4**　对于倒置数是常数级别的文件，插入排序和冒泡排序所要进行的比较和交换操作数目是一个线性数。

如刚才所讨论的，插入排序的运行时间与文件中的倒置数直接相关。对于冒泡排序（这

里，我们指的是程序 6.4，进行了一点修改，当文件已排好序就中断操作），证明的方法更加复杂（参看练习 6.29）。冒泡排序的每步中，每个元素的在右边比它小的元素个数均减一（除了那些已经为 0 的），因此对于该种文件，冒泡排序的执行步骤最多是一个常数级别，也因此，所执行的比较和交换操作是一个线性数目。■

对于另外一种类型的部分排序了的文件，可能是对已排好序的数列添加一些元素或者对已排好序的数列的一些元素进行了修改。该类型文件在应用中是很常见的。对这种类型的文件，使用插入排序的效率比较好；而选用冒泡排序和选择排序的效率就不高。

**表 6.1 基本排序算法的实验性研究**

对小文件进行操作时，插入排序和选择排序的效率是冒泡排序的两倍，但运行时间增长的速度以 4 为基数（当文件的大小以 2 倍增长时，运行时间就以 4 倍增长）。对于巨大的随机排列的文件，这些方法的效率都不高（例如，使用 6.6 节的 Shell 排序法，表中的相应数目将小于 2。当比较操作的花费较大时）如，比较的键是字符串类型——这时，插入排序比其他两种方法的效率都高，因为它使用的比较操作要少得多。这里没有包括的是交换操作花费较大的情形，在该情况下，选择排序的效率最好。

| | 32 位整数类型键 | | | | | 字符串类型键 | | |
|---|---|---|---|---|---|---|---|---|
| *N* | S | I* | I | B | B* | S | I | B |
| 1000 | 5 | 7 | 4 | 11 | 8 | 12 | 8 | 19 |
| 2000 | 21 | 29 | 15 | 45 | 34 | 56 | 31 | 78 |
| 4000 | 85 | 119 | 62 | 182 | 138 | 228 | 126 | 321 |

其中：

S 选择排序（程序 6.2）

I* 插入排序，基于交换（程序 6.1）

I 插入排序（程序 6.3）

B 冒泡排序（程序 6.4）

B* 晃动排序（练习 6.30）

**性质 6.5** 如果文件中有一部分元素的倒置数超过常数级别，插入排序所运行的比较和交换操作仍是一线性数。

插入排序的运行时间是由文件的总倒置数决定的，而与倒置数的分布情况无关。■

要根据性质 6.1~6.5 对算法的运行时间进行总结，我们需要分析比较操作和交换操作的相对花费，该特性与数据项和键的大小有关（参看表 6.1）。例如，如果数据项是单字键，这样，交换操作（访问四个数组）的花费是比较操作的两倍。在这种情况下，选择排序和插入排序的运行时间差不多，而冒泡排序要比较慢。不过，在相对于键来说数据项比较大的情况下，选择排序是效率最高的。

**性质 6.6** 对数据项较大键较小的文件，选择排序的运行时间是一个线性数。

$M$ 表示数据项大小相对于键大小的比例。假设比较操作的花费为 1 单位时间，则交换操作的操作为 $M$ 单位时间。选择排序在比较操作上的花费大概是 $N^2/2$ 单位时间，而交换操作上的花费则是 $NM$ 单位时间。如果 $M$ 比 $N$ 的常数倍要大，则 $NM$ 超过 $N^2$，这种情况下运行时间与 $NM$ 成比例，即和移动所有数据所需要的时间总量成比例。■

例如，对 1000 个包含单字键和 1000 字其他元素的数据项进行排序，而且必须对数据项进行重新安排，这时，选择排序的效率最高，因为运行时间由移动所有 1 百万数据所要消耗的时间所决定。在 6.8 节中，我们会讨论重新安排数据的可选方案。

**练习**

▷ 6.26　对所有键都相同的文件，哪个基本排序方法（选择排序、插入排序或者冒泡排序）运行得最快？

6.27　对倒序的文件，哪个基本排序方法运行得最快？

6.28　给出一个包含 10 个元素的文件（键由 A 到 J），使冒泡排序算法所使用的比较操作要比插入排序算法所使用的少，或者证明不存在这样的文件。

● 6.29　证明：冒泡排序的每步中，都使每个元素的在左边比自己大的元素数减 1（除了已为 0 的）。

6.30　执行这种冒泡排序：执行过程中不断地更改从左到右和从右到左的数据扫描顺序。该种方法称为晃动排序（更快但更复杂，参看图 6.7）。

● 6.31　证明性质 6.5 对晃动排序不适合（参看练习 6.30）。

●●6.32　在 PostScript（参看 4.3 节）中执行选择排序，参照图 6.5~图 6.7 的方法画出执行过程的图示。你可以使用递归法，或者阅读 PostScript 的手册学习关于循环和数组的相关语法。

## 6.6　Shell 排序法

插入排序运行效率较低的原因是它所执行的交换操作涉及邻近的数据项，使得数据项每次只能移动一位。例如，如果键最小的数据项刚好在数组的尾端，就需要 $N$ 步来将该项放到数组最前端。Shell 排序法是插入排序的扩展，它通过允许非相邻的数据项进行交换来提高执行效率。

该算法将文件重新安排，使文件具有这样的特性，每第 $h$ 个文件（从任何地方开始）组成一个排好序的文件。这样的文件称为已 $h$-排序。另一种说法就是，已 $h$-排序的文件是 $h$ 个独立的已排好序的文件，相互交叉在一起。对 $h$ 值较大的 $h$-排序文件，可以通过移动相距较远的元素使文件比较容易地进行 $h$ 值较小的 $h$-排序。通过按照任意以 1 结束的步长序列进行操作，就产生了一个排好序的文件：这就是 Shell 排序法的本质。

Shell 排序法的一种执行方式是，对每个 $h$，单独使用插入排序对 $h$ 子串进行操作。虽然这个过程看起来已较简单，但因为子串的独立性，我们可以使用一个更简单方法。对文件进行 $h$ 排序时，通过将较大的元素移到右边，我们将之插入到它的 $h$ 子串中（参看图 6.8）。我们使用插入排序的代码来完成该任务，但将移动后增加或减少的数目变成了 $h$ 而不是原

来的 1。该资料表明 Shell 排序只是一种与插入排序方法类似的算法，如程序 6.5。该程序的执行过程在图 6.9 中表示。

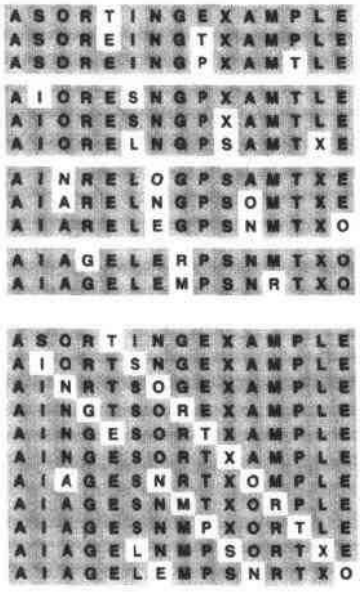


图 6.8 交叉 4-排序

该图的上部份表示了对包含 15 个元素的文件进行 4-排序的执行过程。首先对 0，4，8，12 位置的子串进行插入排序，接着对 1，5，9，13 位置的子串进行插入排序，接着对 2，6，10，14，最后对 3，6，11 位置的子串进行插入排序。不过这四个子串是独立的，所以我们可以通过将每个元素插入到它所在的子串中获得相同的结果，每次向后退 4 步（下部份）。将上部分每节中的第一行取出，接着取出每节的第二行，以此类推，就得出了下部分的列表。

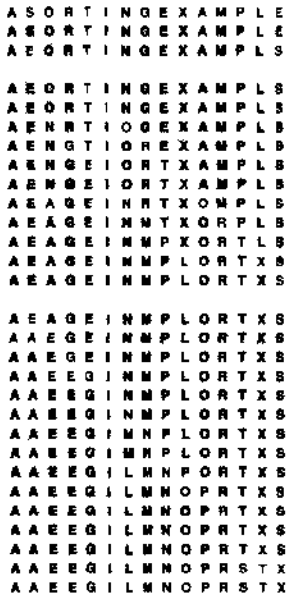


图 6.9 Shell 排序法的例子

对一个文件依次进行 13-排序（顶部）、4-排序（中部）和 1-排序（底部），执行中所进行的比较操作不多（没有打阴影的元素）。最后一步只是单纯的插入排序，不过没有元素需要移动很大距离，因为通过开始两步文件已基本顺好序。

那么如何决定所选用的步长序列呢？一般很难回答这个问题。步长序列的不同特征在文献中已有讨论，有一些在实际中也运用得很好，不过还没证明出最好的序列。在实际中，我们通常使用以几何级别减少的序列，这样，不同步长的数目是文件大小的对数级。例如，如果每次的步长是上一次的一半，那么对一个包含 1 百万数据的文件进行排序，大概只需 20 种步长；如果该比例是 1/4，就只需 10 种步长。尽量使用较少数量的步长是很明显的——我们还需要考虑步长间的相互作用，如它们约数的大小和其他一些特性。

**程序 6.5 Shell 排序法**

如果在插入排序中，我们不使用标志，并且用“h”代替“1”，插入排序的程序就对文件进行了 h-排序。增加一个外部循环来改变步长就实现了 Shell 排序方法，这里使用的步长序列是 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;
        }
  }
```

得到一个好的步长序列的实际效果最多大概能提高 25%的效率。不过该问题是简单问题存在内在复杂性的很好例子，这点是很吸引人的。

程序 6.5 使用的步长序列 1 4 13 40 121 364 1093 3280 9841...，步长间的比例大概是 1/3，是由 Knuth 在 1969 年提出的（参见“第三部分参考资料”）。该方法很容易实现（从 1 开始，通过乘 3 加 1 得到下一个步长），而且效率相对还可以，即使是对中等大小的文件，如图 6.10 所示。

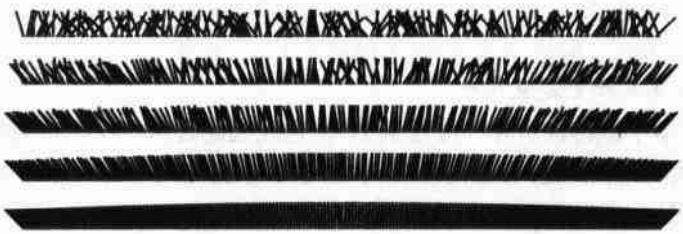


**图 6.10　对一个随机的数列进行 Shell 排序**

Shell 排序中每步都使整个文件更接近于排好序。该文件中首先进行 40-排序，然后进行 13-排序、4-排序，最后进行 1-排序。每步都使文件更接近于排好序。

其他的步长序列可以得到效率更高的排序，不过相对程序 6.5 中的序列很难将效率提高 20%，即使是对相对较大的 $N$。其中一种可以提高效率的步长序列是 1 8 23 77 281 1073 4193 16577...$4^{i+1}+3*2^{i}+1$，$i>0$，可以证明在最坏情况下使用该序列运行得更快（参看性质 6.10）。图 6.12 显示该序列及 Knuth 的序列——和其他一些序列——对大文件有类似的动态特性。事实上的确可能存在更好的步长序列。练习中将会提出几种对步长序列进行改进的观点。

相反地，有一些不好的步长序列，如 1 2 4 8 16 32 64 128 256 512 1024 2048...（1959 年当 Shell 提出 Shell 算法时给出的序列，参见“第三部分参考资料”），由于直到最后一步在奇数位置的元素才会与偶数位置的元素进行比较，这样使用这个序列的效率将很低。对

于随机文件，该特性是显著的，而对最坏的情况，简直是灾难：该方法将退化到需要二次的运行时间，如，奇数位置上保存的是较大的一半元素，而偶数位置中保存的是较小的一半元素（参看练习 6.36）。

在初始化确保使用同一序列后，程序 6.5 通过对步长除以 3 得到下一个步长。另一种方法是 h=N/3 或者其他 $N$ 函数开始。最好避免这种做法，因为在上段所举的不好的序列就是从 $N$ 的某种函数中得到。

我们对 Shell 排序法的效率的讨论是不精确的，因为还没有人能对该算法进行精确的分析。由于这点，我们不仅很难对不同的步长序列进行比较，也不能将 Shell 排序法和其他排序法进行分析性的比较。Shell 排序法的运行时间公式也还未得出（该公式与步长序列有关）。Knuth 发现公式 $N(\log N)^2$ 和公式 $N^{1.25}$ 都很适合，而晚些的研究又发现，对某些序列，一个更复杂的公式同样适用：$N^{1+1/\sqrt{\lg N}}$。

我们通过对一些 Shell 排序法已知的特性进行讨论来对本节进行总结。这样做的基本目的是要表明表面简单的算法也会有复杂的特性的，并且对算法进行分析不仅是实际运用中的重要工作，还是一种智力上的挑战。有兴趣寻找一个新的效率更好的 Shell 排序步长序列的读者会觉得以下信息很有用，而另外一些读者可能希望直接跳到 6.7 节。

**性质 6.7** *对已 $k$-排序的文件进行 $h$-排序所得到的结果是一个既 $h$-排序又 $k$-排序的文件。*

该特性似乎显而易见，但证明起来很微妙（参看练习 6.47）。■

**性质 6.8** *对既已 $h$-排序又 $k$-排序的文件进行 $g$-排序时所执行的比较操作数小于 $N(h-1)(k-1)/g$，假设 $h$ 和 $k$ 是互质的。*

图 6.11 中图示了该特性。对任意元素 $x$，在它左边超过 $(h-1)(k-1)$ 位置的数没有一个比 $x$ 大，如果 $h$ 和 $k$ 是互质的（参看练习 6.43）。当进行 $g$-排序时，对每 $g$ 个数据项我们最多只检查其中一个数据。■

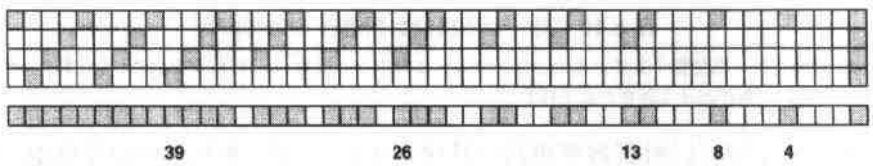



**图 6.11 已 4-和 13-排序的文件**

假设图下部分的数组已 4-排序和 13-排序了，那些打阴影的格表示比在右边的某个数据项要小或者相等。上部分的四行表示了该数列的原始状态。假设右边数据项所在位置为 $i$，数组进行了 4-排序后就意味着在 $i$-4，$i$-8，$i$-12...位置上的数据项比它要小或相等（最上面一行）；因此再进行了 13-排序，再加上原来进行的 4-排序，在 $i$-13，$i$-17，$i$-21，$i$-25，...位置的数据项也较小或相等（第二行）；同样，在 $i$-26，$i$-30，$i$-34，$i$-38，...位置的数据项也较小或相等（第三行）；以此类推。而没有打阴影的项表示可能比在它左边的数要小；最多有 18 个这样的数据项（其中最远的是在 $i$-36 位置的项）。因此，对大小为 $N$，已进行了 13-排序和 4-排序的文件进行插入排序时最多需要执行 $18N$ 次比较操作。

**性质 6.9**　对步长序列 1 4 13 40 121 364 1093 3280 9841... Shell 排序法使用少于 $O(N^{3/2})$ 的比较操作。

对于大的步长，有 $h$ 个大小约为 $N/h$ 的子串，最坏情况下的消耗大概为 $N^3/h$。对于小的步长，性质 6.8 表明了该消耗为 $Nh$。如果我们更好地对每个步长利用这些限制，得到的结果是一样的。它对任意以指数倍增长的基本序列都适用。■

**性质 6.10**　对步长序列 1 8 23 77 281 1073 4193 16577... Shell 排序法使用少于 $O(N^{3/3})$ 的比较操作。

要证明这个特性可沿用证明性质 6.9 的思路。和性质 6.8 类似，该特性表明了对小的步长，消耗大约为 $Nh^{1/2}$。要精确地证明这点，需要一系列的数学理论，这些已超出了本书范围（参见“第三部分参考资料”）。■

我们刚才所讨论的步长数列中各步长是互质的。另一些有效的步长数列的步长不一定是互质的。特别是，性质 6.8 表明，如果一个文件已进行了 2-排序和 3-排序，则在最后的插入排序中，每个元素至多移动一位。也就是说，这样的文件可通过一步的冒泡排序算法排好序（不需要运行插入排序中的额外循环）。这里，如果一个文件已 4-排序和 6-排序，那么如果我们对它进行 2-排序，则在最后的插入排序中，文件中每个元素也最多只需移动一位（因为每个子串都是已 2-排序和 3-排序的）；同样，如果一个文件已 9-排序和 6-排序，对它进行 3-排序后，则在最后的插入排序中，文件中每个元素也最多只需移动一位。延续这样的思路，我们就可以得到以下观点，是由 Pratt 在 1971 年提出的（参见“第三部分参考资料”）。

Pratt 的方法是使用以下三角形数列作为步长序列，其中三角形中每个数据是右上方数据的两倍，同时也是左上方数据的三倍。

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

如果我们按照从下到上、从右到左的顺序读取这些数据作为一个 Shell 排序法的步长序列，那么进行步长为 $x$ 排序前已进行了 $2x$ 和 $3x$ 排序，因此，每个子串都是已 2-排序和 3-排序的，即最后排序中没有一个数据项移动超过一个位置。

**表 6.2　对 Shell 排序的步长序列的实验性研究**

即使只是使用 2 的幂数作为步长序列，Shell 排序法也比其他基本排序方式要快很多，但某些步长序列可使排序效率提高 5 倍或以上。本表中列出的三个最好的序列在设计中是完全不同的。即使对大文件，

Shell 排序法仍适用，特别是和选择排序、插入排序和冒泡排序相比（参看表格 6.1）。

| $N$ | O | K | G | S | P | I |
|---|---|---|---|---|---|---|
| 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 |

其中：

O 1 2 4 8 16 32 64 128 256 512 1024 2048...

K 1 4 13 40 121 364 1093 3280 9841...（性质 6.9）

G 1 2 4 10 23 51 113 249 548 1207 2655 5843...（练习 6.40）

S 1 8 23 77 281 1073 4193 16577 ...（性质 6.10）

P 1 7 8 49 56 64 343 392 448 512 2401 2744 ...（练习 6.44）

I 1 5 19 41 109 209 505 929 2161 3905 ...（练习 6.45）

**性质 6.11** 对步长序列 1 2 3 4 6 9 8 12 18 27 16 24 36 54 81，Shell 排序法使用少于 $O(N(\log N)^2)$次的比较操作。

三角形中的小于 $N$ 的步长数目肯定少于$(\log_2 N)^2$。■

由于数量太多，实际运用中 Pratt 的步长序列的效率比不上其他序列。我们可以使用同样规则，根据两个互质的数字 $h$ 和 $k$ 来构造一个步长序列。在实际运用中，这样的数列的效果较好，因为，在最坏情况下的消耗相等于根据性质 6.11 对随机文件花费的过高估计。

为 Shell 排序法设计一个好的步长序列这个问题，是简单算法也会有复杂特性的一个很好的例子。我们当然不能对所有遇到的问题都进行这种程度的详细讨论（我们不仅是没有这样的环境，而且，如我们在对 Shell 排序法进行讨论时那样，我们会碰见一些超出本书范围的数学分析或研究问题）。然而，本书的许多算法是过去几十年里由许多研究人员进行了广泛的分析和实验后得到的成果，我们可以从中获益。这点表明了改进算法的研究既是一项智力挑战又是可以得到实际运用成果回报的工作，即使只是对很简单的算法进行研究。表 6.2 的研究结论表明，对步长序列进行设计的一系列方法在实际运用中是有效的；其中，序列 1 8 23 77 281 1073 4193 16577...是 Shell 排序法中使用的最简单步长序列之一。

图 6.13 表示了 Shell 排序法对不同类型的文件进行操作时效率都不错，而不只适用于随机文件。事实上，对一给定的步长序列，如何构造一个文件、使 Shell 排序算法的执行效率差是一项有挑战性的练习（参看练习 6.42）。如我们已经提过的，有一些不好的步长序列在最坏的情况下会执行二次性的比较操作（参看练习 6.36），不过因为限制很少，所以可以

构造大量合适的序列。

由于即使对大文件，Shell 排序法都具有较高的运行效率以及代码简单容易执行，很多排序应用程序都选用了 Shell 排序法。在以下几章中，我们会讨论一些更高效的算法，但除非 $N$ 很大，它们也最多也只是提高两倍的效率，而且要复杂得多。简而言之，如果你需要快速地解决一个排序问题，而且不会涉及系统排序，你可以使用 Shell 排序法，然后迟点时间再考虑是否需要用一个更完备的算法代替它。

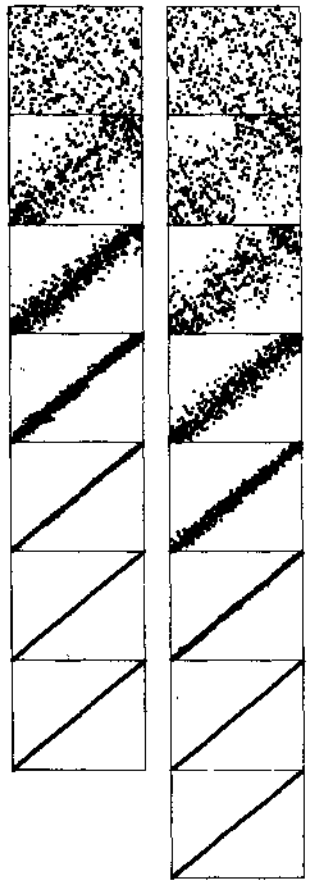


图 6.12　Shell 排序法的动态特性（两种不同的步长序列）

在本例中，Shell 排序法就好像是将一个粘附在四角的橡皮带，拉向对角线的两端。所使用的两个步长序列分别为：121 40 13 4 1（左边）和 209 109 41 19 5 1（右边）。第二种方法执行的步骤比第一种方法多一步，但它更快，因为每步的执行效率更高。

## 练习

▷ 6.33　Shell 排序是否稳定？

6.34　显示使用步长序列 1 8 23 77 281 1073 4193 16577 ...的 Shell 排序法的执行过程，参照 Knuth 序列的代码，使用直接的计算公式得出连续的步长序列。

▷ 6.35　模仿图 6.8 和 6.9 给出对 E A S Y Q U E S T I O N 进行排序的图表。

6.36　计算对奇数位置数据为 1，2，...，$N$，偶数位置数据为 $N$+1，$N$+2，...，2$N$ 的文件使用序列 1 2 4 8 16 32 64 128 256 512 1024 2048... 进行 Shell 排序时所要花费的运行时间。

6.37　写一个驱动程序来对 Shell 排序法的步长序列进行比较。从标准输入中读入数列，每行一个序列，然后用它们来对 10 个大小为 $N$ 的随机文件进行排序，$N$=100，1000 和 10000。统计比较操作的数目和计算实际的运行次数。

● 6.38　做实验证明对 $N$=10000 的文件，使用步长序列 1 8 23 77 281 1073 4193 16577 ...是否可以通过增加一个步长或减少一个步长来提高运行效率。

● 6.39　做实验求出，对 $N$=10000 的随机文件，使用步长序列 1 4 13 40 121 364 1093 3280 9841 ...进行排序，如果用 $x$ 代替 13，即 $x$ 为多少时运行时间最短？

6.40　做实验求出，对 $N$=10000 的随机文件，使用步长序列 1，$\lfloor\alpha\rfloor$，$\lfloor\alpha^2\rfloor$，$\lfloor\alpha^3\rfloor$，$\lfloor\alpha^4\rfloor$，...进行排序时，$\alpha$为多少时运行时间最短？

● 6.41　对包含 1000 个元素的随机文件，寻找一个包含三个步长的序列，使所需要进行的比较操作数量最少。

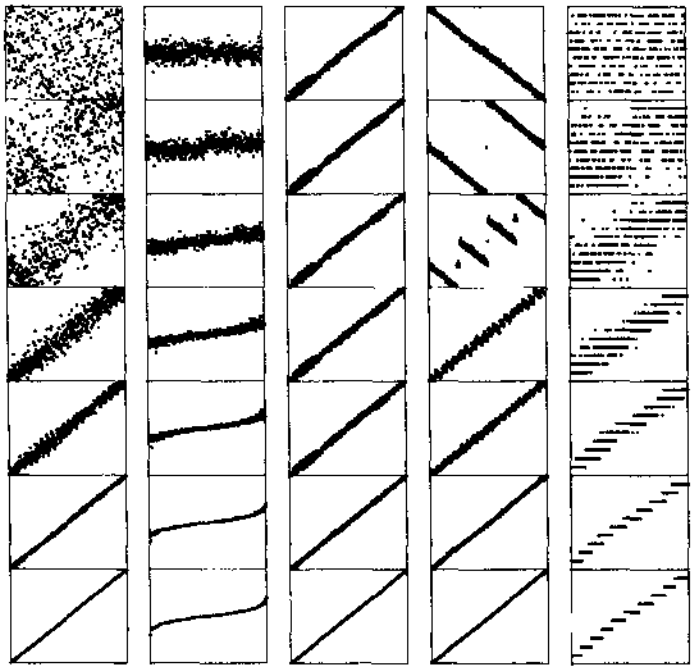


**图 6.13 Shell 排序法对不同类型文件的动态特性**

本图显示了使用序列 209 109 41 19 5 1，对随机文件、高斯序列、已基本排序的序列、接近倒序序列和分成 10 组的随机文件（从左到右）进行排序的过程。开始时，每步的运行时间决定于文件的顺序性。执行了几步后，这些文件都基本排序了，运行时间就与原有输入文件没有很显著的关系了。

●●6.42 构造一个包含 100 个元素的文件，使得使用步长序列 1 8 23 77 进行 Shell 排序时，所要执行的比较操作数量最多。

● 6.43 证明：假设 $h$ 和 $k$ 互质，则任意比 $(h-1)(k-1)$ 大或相等的数可以表示为 $h$ 和 $k$ 的线性组合（系数非负）。提示：如果最早的 $h$-1 个 $k$ 的倍数中有任意两个数除以 $h$ 的余数相等，则 $h$ 和 $k$ 必然有公约数。

6.44 做实验求出，对包含 10000 个数据的随机文件，使用以 $h$ 和 $k$ 为底的 Pratt 序列进行 Shell 排序时，$h$ 和 $k$ 为何值，运行时间最短。

6.45 步长序列 1 5 19 41 109 209 505 929 2161 3905 …是基于将序列 $9*4^i-9*2^i+1$ 和序列 $4^i-3*2^i+1$，$i>0$，结合得出的。对包含 10000 个数据的文件进行排序，比较单独使用这些序列和使用结合序列的结果。

6.46 序列 1 3 7 21 48 112 336 861 1968 4592 13776 …起源于一个基本的互质的序列，如 1 3 7 16 41 101，接着构造一个三角形，如 Pratt 序列中那样，这里，通过将第 $i$-1 行的第一个数据与基本序列第 $i$ 个元素相乘，及将第 $i$-1 行中所有元素分别与基本序列第 $i$+1 个元素相乘，得出三角形中第 $i$ 行数据。做实验求出一个基本序列，使对 10000 个数据进行排序时效率有所提高。

● 6.47 完成对性质 6.7 和 6.8 的证明。

● 6.48 执行一个基于练习 6.30 中的晃动排序方法的 Shell 排序法，与标准算法进行比较。注意：所使用的步长序列要和标准算法中所使用的完全不同。

## 6.7　对其他类型的文件进行排序

虽然在学习排序算法时，只简单地考虑将数字按大小排列或将名称按字母顺序排列，这也是合理的，但认识到排序算法其实与数据类型没有太大关系也是很重要的，而且做出更通用的设定并不困难。我们已经详细地讨论过了如何将程序分成独立的部分来对数据类型进行实现和抽象（参看第三、第四章）；本节中，我们将考虑如何利用已得出的结论，对排序算法构造实现、接口和客户程序。特别是，我们对接口问题进行以下讨论：

- 数据项（item），或者其他要排序的普遍对象；
- 存放数据项的数组(array)；

**程序 6.6　数组的排序驱动程序**

这个对基本数组排序的驱动程序使用了三个显式接口：第一个是数据类型，将普遍数据项的操作过程进行封装；第二个是较高层次的在排序过程中执行的 exch 和 compexch 操作；第三个是初始化和打印（及排序）数组的函数。通过这种方法将驱动程序分成模块，我们可以对不同类型的数据进行排序，而不需要进行任何修改、共享交换及比较-交换过程以及独立地对数组功能进行编译（可能是为其他驱动程序所使用）。

```
#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);
  }
```

item 数据类型使我们在设定一定基本操作后，可以对任何类型的数据使用相同的排序程序。该过程对简单的数据类型和抽象的数据类型同样适用，而我们需要考虑众多的执行过程。数组方面的接口没那么重要；我们调用它的目的是给出一个使用多样数据类型的多模程序例子。我们只考虑一种数组执行方式（只向前）。

程序 6.6 是一个客户端程序，它和程序 6.1 的 main 程序的功能大致相同，但加入了将数组和数据项封装到独立模块的代码。特别地，通过替换不同的模块，我们可以不必修改客户端程序，而对不同类型的数据、不同类型的排序程序进行检验。程序 6.6 还提到了在排序过程中所要执行的 exch、compexch 操作的接口。我们可能会坚持将这些包含在 Item.h 接口中，不过因为当定义了操作符<后，程序 6.1 已满足我们的需要，而它的语义很容易理

解，所以更简便的方法是将它们放在一个单独的模块中，任意的排序算法都可以用来对所有数据类型进行操作。要完成该执行过程，我们需要精确地定义 array 和 item 数据类型的接口，然后提供可执行的程序。

**程序 6.7 数组数据类型接口**

该 Array.h 接口为包含抽象数据项的数组定义了高层次的函数：初始化随机值，通过读取标准输入进行值初始化，输出数据，及对数据进行排序。

```
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);
```

程序 6.7 中的接口定义了一系列可能会对数组进行的高层次操作的例子。我们希望能够用键值对数组进行初始化，或者是随机的或者是从标准输入中得到；我们希望能对数据进行排序（这是当然的！）；而且我们还希望输出结果。这只是其中一些例子；在某些特定的应用程序中，我们可能希望定义其他一些操作（标准模块库中的 Vector 类就是其中一种提供该类型普通接口的方法）。使用程序 6.7，我们可以用不同的执行过程来替换这些操作，而不需要修改客户端程序，在这里，即指程序 6.6 中使用 main 接口的程序。我们正在研究的各种排序方法都可以当作 sort 函数的执行过程。程序 6.8 是一个简单的执行例子。模块机制允许我们根据需要替换其他执行过程。例如，我们可以使用一个 show 函数的实现，当数组比较巨大的时候，只打印出数组的某些部分。

**程序 6.8 数组数据类型的执行情况**

本代码是对程序 6.7 中所定义的函数的实现过程，使用 item 类型和在独立的接口中定义的基本功能（参看程序 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;
    }
```

使用类似的方法，要使用某些特别类型的数据项或关键字，我们就对该类型进行定义，并且在一个外部的接口中对所有相关的操作进行声明，然后给出在数据项接口定义的操作的执行过程。例如，假设有一个会计应用程序，其中我们有一个对应于顾客账号的关键字和一个对应于顾客账目余额的浮点数。程序 6.9 是为该应用应于数据类型的例子。接口代码对<操作符进行了声明，我们需要使用它来对关键字进行比较；同样还声明了产生随机关键字的操作、读取关键字值的操作及输出关键字值的操作。程序 6.10 给出了这个简单例子中这些函数的具体过程。明显的，我们可以使用同样的方法对特殊的应用程序进行操作。例如，Item 可以是一个类似 C++类的 ADT，而关键字可以是类对象的一个成员函数，而不只是某结构中的数据成员。我们会在第十二章中讨论 ADT。

**程序 6.9　数据项的示例接口**

程序 6.6 中调用的文件 Item.h 定义了所要进行排序的数据项的数据类型。本例中，数据项是包含整数键和相关浮点数信息的小记录。重载操作符<将会独立执行，其他三个函数 scan（读取一个 Item 数据作为它的参数）、rand（在参数中存储一个随机的 Item 数据）、和 show（输出一个 Item 数据）同样也会独立执行。

```
typedef struct record { int key; float info; } Item;
 int operator<(const Item&, const Item&);
 int scan(Item&);
void rand(Item&);
void show(const Item&);
```

程序 6.6~程序 6.10 再加上我们在 6.2 节~6.6 节讨论的任一种排序算法，就构成了一个对小数据进行排序的测试程序。通过为其他类型的数据提供类似的接口和执行程序，我们可以实现对不同类型的数据进行排序，如复数（参看练习 6.50）、矢量（参看练习 6.55）或者多项式（参看练习 6.56），而不需要对排序代码进行任何修改。对于类型更复杂的数据项，接口和执行过程会更复杂，不过这个执行工作和我们一直讨论的算法设计问题是完全分离

的。我们可以用相同的方法使用本章中所讨论的大部分排序算法，对我们将学习的第七章到第九章的排序算法同样适用。在第 6.10 节中，我们会详细讨论一个重要的例外，它将引出一个重要的需要单独讨论的排序算法体系，这也是第十章中的主题。

本节中所讨论的方法，是程序 6.1 和包括错误检查、内存管理和其他更普遍能力的工业性实现的完整集合的一个过渡。这种类型的封装在一些现代编程和应用环境中越来越重要。对某些问题，我们不去解答。我们的主要目的是去论证，通过我们已经检验过的相对简单的机制，我们学习的排序算法是广泛适用的。

**程序 6.10 数据项的执行过程示例**

本代码中给出程序 6.9 中声明的重载操作符<及 scan、rand 和 show 函数的执行过程。由于记录的结构较小，exch 可以使用内置的赋值操作符，而无须担忧复制数据项时的消耗。

```
#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; }
```

## 练习

6.49 对程序 6.9 和程序 6.10 进行改写，使用重载操作符<<和操作符>>代替 scan 和 show，并且使用自己的接口来修改程序 6.8。

6.50 编写接口程序和执行程序，使之可对复数类型数据 $x+iy$ 进行排序，使用量级 $\sqrt{x^2+y^2}$ 作为键值。注意：忽略平方根会更高效。

○ 6.51 写一个接口程序，声明一个一级抽象数据类型（参看 4.8 节），并且像前一题那样，编写一个数据项为复数的执行程序。使用程序 6.3 和程序 6.6 对所编写的程序进行检验。

▷ 6.52 在程序 6.8 和程序 6.7 中，增加一个 check 函数来检验数组是否已排好序。

● 6.53 在程序 6.8 和程序 6.7 中增加一个 testinit 函数，根据与图 6.13 相似的分布产生测试数据。为客户端程序提供一个整型参数来指定该分布。

● 6.54 修改程序 6.7 和程序 6.8，对 abstract 数据类型进行操作。（执行过程必须产生和保存数组，如第三章中对栈和队列执行的那样。）

○ 6.55　编写接口和执行程序，对多维变量进行排序，维数为 $d$，首先跟据第一维数据进行排序，第一维相等的就根据第二维数据进行排序，第二维也相等的数据就根据第三维排序，以此类推。

6.56　编写接口和执行程序，对多项式进行排序（参看第 4.9 节）。其中需要对合适的排序规则进行定义。

## 6.8　索引和指针排序

对字符串类型数据进行排序的过程与程序 6.9 和程序 6.10 相似，对这点进行研究是很重要的，因为排序应用中有很多键使用特殊的字符串。此外，由于字符串的长度是不定的，它可以很长，所以对字符串进行创立、删除和比较操作时的消耗可能会很大，因此，我们需要特别注意执行过程中额外的不必的消耗。

最后，我们使用一个包含指针的数据表示法（指向一个字符数组），典型的 C 语言字符串表示法。接着，我们可以将程序 6.9 的第一行修改成：

```
typedef struct { char *str; } Item;
```

从而将程序 6.9 变成适合字符串的接口程序。我们将指针放在 struct 中是因为 C++不允许重载操作符<对内部数据类型（如指针）进行操作。这种情况在 C++中是普遍的；用于调整接口程序，使之适用于另一种数据类型的类（或结构，struct）称为包装类。虽然这里对包装类的要求很低，但在某些情况中，就需要更复杂的执行过程。不久我们就会讨论到另一个例子。

程序 6.11 是对字符串类型数据项的一个执行过程。重载操作符<的执行过程很容易根据 C 语言库中的字符串比较函数得出，不过 scan 和 rand 函数的执行过程比较复杂，因为我们首先要知道字符串的存储方式。程序 6.11 使用我们在第三章（程序 3.17）中讨论过的方法，使用一个缓冲器对数据类型进行存储。我们还可以对每个字符串进行动态的内存分配，或者使用像 String 类型那样的标准模块类，或者将缓冲器放置在客户端程序中。我们可以使用以上任一种存储方法（使用相应的接口程序）和我们讨论过的任一种排序方法来对字符串进行排序。

当我们对程序进行模块化设计时，都会面临这种选择内存管理方法的问题。选择内存管理方法时应该根据哪些对象做出决定：是客户端程序、数据类型，还是系统？这个问题很难很快得出结论（某些程序语言的设计者对这个问题也含糊其辞）。某些现代编程系统（包括某些 C++执行过程）包含自动管理内存的机构。我们会在第九章中再次讨论这个问题，那时我们会讨论一个更复杂的抽象数据类型。

程序 6.11 是一个使用指针类型的例子，我们会简要地进行讨论。另一个使数据项在排序时不需要（中间）移动的方法是构造一个索引数组，该数组中添加了由比较操作进行修改的键。假设要排序的数据项存储在数组 data[0]，…，data[N-1]，因为某些原因（如数据项比较巨大），我们不希望将它们进行移动。要进行排序，我们需要使用第二个数组 a 来存

储数据项的索引。开始时，我们对 a[i]进行初始化，使 a[i]=i，i=0，…,N-1。也就是说，开始时，a[0]保存第一个数据项的索引，a[1]保存第二个数据项的索引，以此类推。而排序的目的是对索引数组 a 进行重新安排，使 a[0]保存最小数据项的索引，a[1]保存第二小数据项的索引，以此类推。通过读取索引键，我们可以知道排序的结果——例如，我们可以用这种方法输出排好序的数组。

**程序 6.11 字符串数据项的执行过程**

该执行方法允许我们使用自己的排序程序对 C 字符串进行操作。我们使用一个包含指向字符串的指针的结构来代表数据项（参看正文），因此排序时会对指向字符串的指针数组进行操作，对它们进行重新安排使字符串按字母顺序排列。为了明确地阐明内存管理系统的执行过程，在本模块中，定义了一个固定大小的存储缓冲器来保存字符串；也许动态地分配内存空间可能更适当。Rand 的执行过程在这里可以忽略。

```
#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;
  }
```

需要声明的是我们的排序算法适用于对数组进行操作，而不只是对通常的整数序列。我们的目的是要声明一个 Index 类型，使我们可以对操作符<进行重载，例如：

```
int operator<(const Index& i, const Index& j)
  { return data[i] < data[j]; }
```

假设有一个存储 Index 类型对象的数组 a，任意的排序算法会对 a 中的索引重新安排，使 a[i]中存放比 data[i]小的键的号码（在已排好序的数组中的 a[i]的索引）。（简单来说，该方法是假设数据是键值，而忽略整个数据项——对更巨大更复杂的数据项，我们可以使用相同的原理，通过修改操作符<来访问数据项的键，或者使用一个通用的类成员函数对键进行操作。）我们使用以下包装类来定义 Index：

```
struct intWrappper
  {
    int item;
    intWrapper(int i = 0)
      { item = i; }
    operator int() const
      { return item; }
  };
typedef intWrapper Index;
```

该结构中的构造函数将 int 转换成 Index，函数 operator int()将 Index 变回 int，因此我们可以在任何可以使用内在数据类型 int 对象的地方使用 Index 类型对象。

图 6.14 中显示了对相同数据项的两个不同关键字进行索引排序的例子。一个客户端程序可通过对操作符<进行字义来使用其中一个关键字，而另一个客户端程序则对操作符<进行另一项定义使之适用于另一个关键字，不过两个程序都可以使用相同的排序程序来产生索引数组，使之可根据各自的关键字进行排序访问。

| | | | | |
|---|---|---|---|---|
| 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 |

**图 6.14　索引排序示例**

通过对索引而不是记录本身进行操作，排序时我们可以同时对几个键进行操作。本例中，索引包括学生的姓名和级别，第二列是对姓名进行索引排序的结果，第三列是对级别进行索应排序的结果。例如，Wilson 按字母顺序排在最后一位，而他在第 10 个年级中；Adams 按字母顺序排在第一位，而他在第 6 个年级中。

对 $N$ 个比 $N$ 小的正整数进行重新排列在数学上称为置换；一个索引排序进行的是一个置换过程。在数学上，置换通常声明为对 1~$N$ 进行重新整理；我们则使用 0~$N$-1 来强调置换和 C++数组索引的直接关系。

这种使用索引数组进行间接排序的方法可通过任何支持数组的编程语言实现。另一种方法是使用指针，如我们刚才对字符串类型所做的那样（程序 6.11）。对数据项大小固定的数组进行排序时，使用指针排序和使用索引排序完全相同，不过指针排序比索引排序更具有普遍性，因为指针可以指向任意地方，这样要排序的数据项不需要固定大小。和索引排序中一样，假设 a 是指向关键字的指针数组，调用 sort 操作的结果是对指针进行重新调整，使连续地访问 a 数组数据时可以按顺序地访问到各键。通过指针指向执行比较操作；通过交换指针执行交换操作。

标准 C 库中的排序函数 qsort 是一个指针排序程序（参看程序 3.17），它将自己的比较函数作为一个参数（而不是像我们刚才那样依赖于重载操作符<）。该函数包括四个参数：数组；要排序的数据项个数；数据项的大小；指向比较两个数据项的函数的指针，该函数根据指向这两个数据项的指针执行。例如，假设 Item 是 char*，以下执行的字符串排序代码遵循了该约定：

```
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); }
```

**程序 6.12 记录数据项的接口程序**

该记录包含两个关键字：第一个是字符串关键字（例如，姓名），第二个是整数关键字（例如，级别）。这些数据都比较大，不容易复制，所以我们定义 Item 为包含指向记录的指针的结构。

```
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&);
```

以下算法没有在接口程序中详细说明，但在快速排序（参看第七章）中广泛使用。在第七章中，我们会讨论这点正确的原因。我们还会从该章到第十一章，逐步了解为什么对某些特别类型的应用程序使用其他算法会更适合，而且我们会寻求提高计算执行效率的方法，当排序时间是应用程序中重要的因素时，这点更重要。

在典型的情况下，指针用来对包含几个可能的关键字的记录进行访问。例如，包含学生姓名和级别或者人们的姓名和年龄的记录，可以声明如下：

```
struct record { char[30] name; int num; }
```

我们会希望使用任一项作为关键字对数据进行排序。程序 6.12 和 6.13 提供了一个关于指针排序的接口程序和执行程序，这些程序使我们能实现这点。我们使用一个指向记录的指针数组，并且为不同的排序应用操作提供不同的操作符<执行过程。例如，如果我们将程序 6.13 和包含以下语句的文件一起编译，排序执行过程将对整数类型关键字进行指针排序操作：

```
#include "Item.h"
int operator<(const Item &a, const Item &b)
  { return a.r->num < b.r->num; }
```

**程序 6.13 记录数据项的执行程序**

这里，scan 和 show 函数的执行过程和程序 6.11 中对字符串类型数据进行的执行过程类似，因此，对记录进行了内存分配和保存。我们在单独的文件中对操作符<的执行过程进行声明，这样我们就可以使用不同的执行程序，因此，可以改变排序的关键字，而不需要修改任何其他的代码。

```
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;
  }
```

而如果我们将程序 6.13 和包含以下语句的文件一起编译，排序执行过程将对字符串类型关键字进行指针排序操作：

```
#include "Item.h"
#include <string.h>
int operator<(const Item &a, const Item &b)
  { return strcmp(a.r->name, b.r->name) < 0; }
```

使用索引和指针的主要原因是避免对要排序的数据进行强行访问。即使文件只可读，我们也可以对文件进行“排序”。此外，我们还可以通过多重的索引或指针数组，对文件进行多重关键字的排序（参看图 6.14）。这种不需要对数据进行修改的适应性在很多应用中都有用。

使用索引的第二个原因是可以无需移动整个记录项。这样，由于比较操作只需访问记录的一个小部分，而记录的大部分在整个排序过程中都不会触及到，因此对数据项很大（而键较小）的文件进行操作时，可以节省很大部分的消耗。在普通的情况下，甚至对较大的文件，交换操作的间接实现使它的花费和比较操作的花费差不多（包括为设置索引或指针时的额外内存消耗）。事实上，如果关键字比较长，交换操作的消耗会比比较操作的少。当我们估计对整数类型文件进行排序的消耗时，经常会假设比较操作和交换操作的消耗是差不多的。当执行指针或索引操作时，很多情况下都使用了这种假设。

很多应用程序中，对数据项都无需进行物理上的重新调整，来使之与索引的顺序相匹配，因此我们只需简单地根据索引数组给出的顺序访问数据项。如果由于某些原因，这样的操作不能满足要求，我们就要考虑一个典型的问题：如何重新调整已进行索引排序的文件？代码

```
for (i = 0; i < N; i++) datasorted[i] = data[a[i]];
```

很简单，但需要额外的内存空间存储数组的另一个拷贝。如果没有足够的空间放置文件的拷贝，那怎么办呢？我们不可以盲目地让 data[i]=data[a[i]]，因为这样会过早地改写了 data[i]的值。

图 6.15 示范了解决这个问题的方法，也只是使用了一个简单的步骤。为了将第一个元素移到正确的位置上，首先将

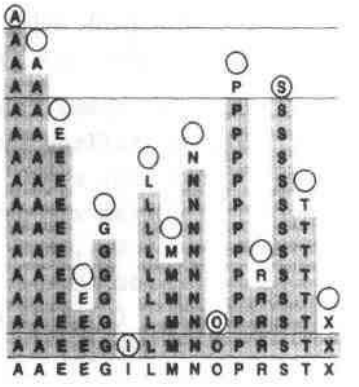



**图 6.15　适当位置的排序**

为了重新整理数组，我们从左到右访问数组，在循环操作中移动需要操作的元素。这里，包含四个循环：第一个和最后一个循环是单个元素的情况。第二个循环从 1 开始，S 保存在一个临时变量中，位置 1 上变为空，将第二个 A 移到那里，位置 10 上变为空，位置 10 由 P 填充，位置 12 变为空，接着在位置 1 的数据要填充到位置 12，于是将 S 重新写到该位置上，就完成了一个 1 10 12 的循环，将这些位置上的数据放到了合适的位置上。类似地，循环 2 8 6 13 4 7 11 3 14 9 完成了排序。

该位置的元素移到它的正确位置中，以此类推。按着这种方法，直到找到应放在第一位的元素，这样我们就做了一次将元素放到合适位置的循环。接着，我们移动第二个元素，用相同的方法继续做循环操作，以此类推（我们已经访问过的元素，已经在正确的位置上（a[i]=i），即不需再移动）。

特别地，对每个 i，保存 data[i]的值，然后初始化一个索引变量 k 为 i。接着，我们将数组位置 i 上的数据视为空，寻找一个元素来补充。该元素就是 data[a[k]]，也就是说，使 data[k]=data[a[k]]。这样，data[a[k]]的值又为空了，将 k 值变为 a[k]。重复这样的操作，直到需要补充 data[i]，即我们刚才保存的那个数据。当我们将一个元素放到合适的位置时，就对数组 a 进行更新。任意在合适位置的元素都有 a[i]等于 i。继续访问数组，当遇到还未放到合适位置的元素时，就做一次新的循环操作，对每个元素至多移动一次。程序 6.14 是该方法的执行过程。

**程序 6.14 适当位置的排序**

根据索引数组 a[0]，…，a[N-1]，对数组 data[0]，…，data[N-1]进行重新整理，将各元素放到合适位置。任何 a[i]= =i 的元素表示已放到了合适的位置，不需要再次修改。否则，将 data[i]保存到 v，接着做循环操作，a[i]，a[a[i]]，a[a[[i]]]，以此类推，直到再次遇到索引 i。对第二个还未放到合适位置的元素，继续进行这样的操作，按照这样的方法，最后对整个文件进行了重新整理，每个记录只需移动一次。

```
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;
      }
  }
```

这个过程称为合适置换（in situ permutation），或者对文件的适当位置的排序（in-place rearrange）。虽然这个算法很有意思，但大部分情况下都不需要使用，因为间接的访问数据已经可以达到要求了。同样地，如果相对于数量来说，数据项的大小比较大，那么最有效的方法是简单地对它们进行一次传统的选择排序（参看性质 6.5）。

间接排序需要额外的内存空间存放索引或指针数组，还需要额外的时间进行间接的比较操作。在许多应用中，这些消耗是为了不需移动数据所做的小花费。对于包含大数据项的文件，我们通常会选择使用非直接排序，而对许多应用，根本不需要移动数据。本书中，我们通常会直接访问数据。不过，在某些情况下，为了刚才所说的原因，我们会使用指针或索引数组来避免移动数据。

**练习**

6.57　给出一个例子，数据项直接保存记录，而不是保存记录的指针。这样的安排，对小的记录，适用于程序 6.12 和程序 6.13（记住 C++支持结构赋值）。

○ 6.58　显示如何使用 qsort 来解决程序 6.12 和程序 6.13 中的提到的排序问题。

▷ 6.59　对键 EASYQUESTION 进行索引排序，列出得到的索引数组。

▷ 6.60　对 EASYQUESTION 进行索引排序（参见练习 6.59），然后列出对数据进行重新整理时所需要的数据移动操作序列。

6.61　给出一个大小为 $N$ 的置换（数组 a 的一系列值），该置换使程序 6.14 中 a[i] ! = i 的次数最大。

6.62　证明：程序 6.14 中，通过不断移动关键字和清空项的操作，必然会返回到一开始访问的数据项。

6.63　设计一个与程序 6.14 类似的与指针排序对应的程序。假设：指针指向一个有 $N$ 个记录的数组，记录类型为 Item。

## 6.9　链表排序

根据第三章所说的，数组和链表是组织数据的两种基本方法，而在 3.4 节中，作为链表处理的例子，我们讨论了对链表进行的插入排序执行过程（程序 3.11）。至今，我们所讨论的排序程序中，都是假设排序数据存放在数组中的，而如果我们使用链表来组织数据，这些排序程序不能被直接使用。在某些情况下，这种算法是有用的，但只是当以基本的序列化方法对数据进行处理时，我们能有效地支持链表。

**程序 6.15　定义链表类型接口程序**

可以将链表的接口程序与程序 6.7 中的数组接口程序进行对比。randlist 函数创建了一系列随机数据项，包括进行内存分配。showlist 函数输出列表中的关键字值。排序程序使用重载操作符<对数据项进行比较，接着通过指针重新整理数据项。使用平常的方法来表示节点的数据（参看第三章），还调用了一个节点的构造函数，根据给定的值和空的链接来建立一个新的节点。

```
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);
```

程序 6.15 是一个与程序 6.7 类似的接口程序，使用了链表类型数据。和程序 6.15 一致，与程序 6.6 对应的驱动程序只有一行代码：

```
main(int argc, char *argv[])
  { showlist(sortlist(scanlist(atoi(argv[1])))); }
```

大部分的工作（包括内存分配）留给了链表和 sort 的执行过程。和数组驱动程序一样，我们希望对列表进行初始化（或者通过标准输出或者使用随机值），来显示列表的内容，并且，当然，要对它进行排序。通常，我们使用 Item 作为要排序的数据的类型，与在第 6.7 节做的一样。链表接口程序的具体执行代码和我们在第三章详细讨论的相似，我们将把这个留作练习。

该接口程序比较低级，它没有区分链接（指向节点的指针）和链表（指针或者是 0 或者指向一个包含列表指针的节点）。换句话说，我们可以为链表和执行过程选择使用一个一级 ADT，对虚拟节点协定进行精确声明，等等。使用低级方法能让我们更容易关注链接的操作，这点决定了算法和数据结构的特征，也是本书主要关心的地方。

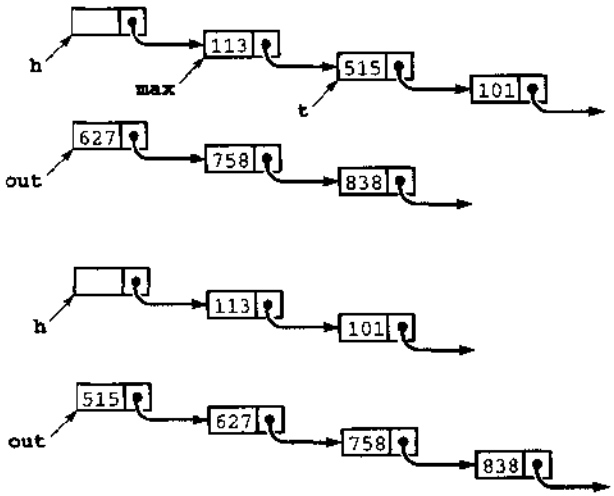



图 6.16 链表选择排序

该图表描述了对链表进行选择排序的其中一步。假设有一个输入列表，通过指针 h→next 指向，及一个输出列表，通过 out（最顶）指向。遍历该输入链表，使 max 指针指向包含最大数据项节点（t 指针指向）的前一个节点。将 t 所指向的节点从输入列表中移去（长度减一），将它加到输出列表的最前端（长度加一），使输出列表保持按顺序排列（最底）。重复这样的操作，直到输入列表清空、节点在输出列表中按顺序排列。

操作链接结构有一个基本的定律，在很多应用中是很重要的，但在我们的代码中不一定这么显著。在更复杂的环境中，有可能所操作的列表节点是由应用系统中其他部分控制的（如多重链表）。由于节点能被排序过程以外的指针所指向，我们的程序只能改变节点间的链接，而不能改变键值或其他内容。例如，当我们进行交换操作，最简单的似乎是直接交换数据项（如我们在数组中所做的那样）。但其他指向该节点的指针会发现值被改变了，

而可能导致不希望的效果。我们对链表进行修改，是希望通过指定的指针能按顺序地访问各节点，而通过其他指针访问的顺序没有受到影响。这样做会使实现过程更困难，但通常是必须的。

我们可以将插入排序、选择排序和冒泡排序进行修改，使它们适用于链表，而每项修改都是很有趣的挑战。选择排序很简单：设定一个输入列表（初始时存放数据）、一个输出列表（存放排好序的结果），只是对输入列表进行扫描，找出最大的元素，将它从输入列表中移走，加入到输出列表的最前端（参看图 6.16）。该执行过程是对列表操作的一个简单练习，对短列表进行排序是很有效。程序 6.16 给出了执行过程。其他排序方法的实现留作练习。

**程序 6.16　链表选择排序**

对链表进行选择排序很简单，但由于在链表开头插入数据很方便，它的执行过程和数组类型的有点不同。我们设定了一个输入列表（由 h->next 指向）、一个输出列表（由 out 指向）。当输入列表非空时，对它进行扫描，找出剩余元素中最大的那项，然后将该数据项从输入列表中移去，将它插入到输出列表的最前端。该执行过程使用了一个辅助过程 findmax，该操作返回一个链接指针指向最大元素的节点的指针（参看练习 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;
  }
```

在一些列表操作中，我们可能根本不需要直接执行排序操作。例如，我们可以选择在任何时候都保持列表按顺序排列，按插入排序中的方法插入新的节点。如果进行的插入操作相对少或者列表较小，以及其他一些情况下，该过程的额外花费会很少。例如，由于某些原因（可能是需要检查是否有重复数据），在插入新节点前，我们可能需要遍历整个列表。在第十四章中，我们会讨论一个用在已排好序的列表上的算法，而在第十二章和第十四章中，我们会看到许多通过数据顺序获得效率的数据结构。

## 练习

▷ 6.64　列出使用程序 6.16 对 A S O R T I N G E X A M P L E 进行操作时，输入列表和输出列表的内容。

6.65　给出程序 6.15 中链表接口程序的执行程序。

6.66 编写链表排序的执行-驱动客户程序（参看练习 6.9）。

● 6.67 编写链表的一级 ADT（参看第 4.8 节），包括初始化随机数的构造函数、根据重载操作符<<初始化的构造函数、通过重载操作符>>进行的输出操作、析构函数、复制构造函数和排序函数。使用选择排序实现 sort，findmax 作为一个私有的成员函数。

6.68 对链表执行冒泡排序。小心：在链表中交换两个相邻的元素要比乍看起来要难。

▷ 6.69 对程序 3.11 中的代码进行封装，使它和程序 6.16 有同样的功能。

6.70 在程序 3.11 中使用的插入排序法，使链表插入排序在某些输入文件中要比数组类型的慢。描述一个这样的文件，并解释原因。

● 6.71 设计一个用于链表的 Shell 排序程序，使得与数组类型的比较起来，所花费的时间或空间没有明显的增多。提示：使用冒泡排序。

●●6.72 对序列进行 ADT，允许使用单个客户程序对链表和数组排序程序进行调试。也就是说，客户程序可以产生一个包含 $N$ 个数据项的序列（或者产生随机序列或者从标准输入中读入），对序列进行排序，或者显示它的内容。例如，在文件 SEQ.cxx 中的 ADT，必须使用以下代码：

```
#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();
  }
```

编写一个使用数组表示和另一个用链表表示的执行过程。使用选择排序。

●●6.73 扩展练习 6.72 的执行程序，使它成为一级 ADT。提示：在标准模板库中寻找解决方法。

## 6.10 关键字索引统计

有些排序方法可以根据关键字的特性来提高效率。例如，考虑下面问题：对包含 $N$ 个不同数据的文件进行排序，这些数据是从 0~$N$-1 的整数。使用一个临时数组 b，可以很快地解决这个问题，该数组声明如下：

```
for (i = 0; i < N; i++) b[key(a[i])] = a[i];
```

也就是说，排序时将关键字作为索引，而不只用于比较。本节中，我们会讨论一个使用这种关键字索引方法的排序算法，该方法在关键字值是一个小范围整数时比较有效。

如果所有的关键字都是 0，排序极其容易完成，不过

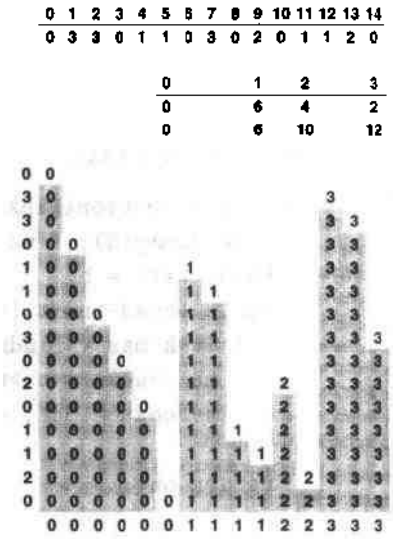



图 6.17 根据键索引统计排序

首先，我们测定文件中不同值的键的个数分别是多少：本例中有 6 个 0、4 个 1、2 个 2 和 3 个 3。接着，进行局部统计，得出比其他关键字小的关键字的数目：0 个关键字比 0 小，6 个关键字比 1 小，10 个关键字比 2 小，12 个关键字比 3 小（中间的表格）。接着，我们将这些统计数作为索引将关键字放到合适的位置：在文件开头的 0 放到位置 0；根据 0，增加指针值，使之指向下一个 0 应该到的位置。接着，文件左边第二位的 3 放到位置 12（因为有 12 个关键字比 3 要小）；它相应的数增加，以此类推。

现在要考虑关键字值是 1 和 0 的情况。这种排序问题会发生在投票表决事件中，0 表示同意，1 表示反对。一种方法是首先统计 0 值的数目，接着对输入数组 a 进行第二次遍历，使用包含两个统计数的数组将数据项分配到临时数组 b 中。Cnt[0]的初始值是 0，值为 0 的关键字的数目保存在 cnt[1]中，这说明没有关键字的值小于 0，有 cnt[1]个关键字的值小于 1。因此，数组 b 中，开始时是填 0 值（从 b[[cnt[0]]，或 b[0]开始），而从 b[cnt[1]]开始填 1 值，代码如下：

```
for (i = 0; i < N; i++) b[cnt[a[i]]++] = a[i];
```

就将数组 a 的数据项分配到了数组 b 中了。这样，通过将关键字作为索引（在 cnt[0]、cnt[1]中挑选）就可以很快完成排序。

这个问题很快可以解决。一个具相同性质的更实际的问题是：对 $N$ 个键值在 0~$M$-1 之间的数据项进行排序。我们可以将前一章所讨论的算法扩展成一个称为关键字索引统计的算法，当 $M$ 不是很大时，该算法可以有效地解决这种问题。和只包含两种关键字值的情况类似，思路是先统计各种不同值的个数，然后根据统计数将数据项分配到各位置中。首先，统计各关键字值的数目；接着，统计等于及小于各值的键的数目。最后，和只有两个值的情况一样，根据这些统计数分配数据项。

**程序 6.17　关键字索引统计**

第一个 for 循环将总量初始化为 0；第二个 for 循环将第二个统计数设为 0 的数目，将第三个统计数设为 1 的数目，以此类推。接着，第三个 for 循环只是将这些数目相加，根据统计，分别计算比某关键字要小或相等的关键字的总数。这些数目显示了关键字在文件中所属的结尾处的索引。第四个 for 循环根据这些索引将关键字移到了辅助数组 b 中，而最后一个循环将已排好序的文件写回到 a 中。该代码中，关键字必须是小于 M 的整数，我们可以对它进行修改，使它可以从更复杂的数据项中提取这样的关键字（参看练习 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-l];
  }
```

**性质 6.12**　关键字索引统计是一个线性时间的排序算法，关键字值的范围是文件大小的一个常数因子。

每个数据项移动两次：一次是为了对数据项进行分配；一次是为了将数据项移回到原

来的数组中，而对每个关键字会访问两次：一次是进行统计；一次是进行分配。算法中另外两个 for 循环与构造统计数有关，而除非统计操作的数目比文件的大小要大得多，这两个循环不会影响到整个运行时间。■

如果进行排序的文件很大，使用辅助数组 b 会导致内存分配问题。我们可以通过修改程序 6.17 来使用恰当的空间完成排序操作（避免使用额外的数组），使用的方法和程序 6.14 中的方法相似。该操作和在第七章和第十章中讨论的基本排序方法密切相关，因此我们推迟到 12.3 节的练习 12.16 和练习 12.17 中进行讨论。在第十二章中我们会发现，节省空间的操作牺牲了算法的稳定性，这样就限制了算法的效用，因为包含大量重复关键字的文件经常还带有其他相关的关键字，这些关键字的相对顺序不应改变。在第十章中，我们会观察一个非常重要的例子。

## 练习

○ 6.74 编写一个特别的关键字索引排序，对数据项只具有三种值（*a*、*b* 或 *c*）的文件进行排序。

6.75 假设对一个随机排列的文件进行插入排序，其中该文件数据只能具有三种值之一。该运行时间是线性的、二次的，还是两者之间的？

▷ 6.76 对文件 A B R A C A D A B R A 进行关键字索引统计排序，显示其执行过程。

6.77 对数据项较大、包含范围较小的整数关键字的文件执行关键字索引统计。

6.78 按指针排序实现关键字索引统计。

# 第七章

# 快速排序

这一章介绍排序算法中应用最广泛的快速排序算法。最基本的快速排序算法是在1960年由C. A. R. Hoare提出的，从那时起这一算法就由许多人进一步研究（参见“第三部分参考资料”）。快速排序算法之所以这样通用是因为它易于实现，它可以处理多种不同的输入数据，许多情况下它所需要的资源也比其他排序算法少。

快速排序算法的优点是在空间上只使用一个小的辅助栈，时间上对 $N$ 个数据项排序平均只需要等价于 $N \log N$ 的时间，而且它内部的循环很小。它的缺点是不稳定，在最坏情况下要执行 $N^2$ 次操作，而且如果实现上有一个不注意的小错误就会使得执行情况很糟糕。

快速排序算法的性能很好理解，它已经经过完整的数学化分析，因此我们可以简略地讨论它的性能。算法的分析也经过实验验证，快速排序经过多次提炼，它已经是许多实际排序领域所采用的算法，所以在如何更有效地实现快速排序算法方面仔细研究是值得的，相似的实现还可以用于其他算法。我们可以放心地使用快速排序算法，因为我们可以肯定它对性能的影响。

改进快速排序算法很重要：一个更快速的排序算法是计算机科学领域“更好的捕鼠器”，快速排序算法就是可选的方法。几乎从Hoare第一次发表这个算法的一刻起，它的改进版本就不断地出现。其中很多方法都经过实验和分析，人们很容易被这些改进版本所蒙蔽，因为这些算法将程序的某一部分性能提高的同时常常将另一部分性能降低。我们将详细地讨论三个改进版本。

一个好的快速排序算法应该在大多数计算机上运行得都比其他排序算法快，而且快速排序算法已经作为一个库排序实用程序用于其他排序领域。实际上，标准C++类库中的排序程序被称为qsort，因为快速排序是实现中的最基本的算法。然而，快速排序的运行时间取决于输入，从输入数据项的线性关系到平方关系不等，人们常常对一些输入数据的意想不到的结果感到吃惊，尤其是在一些改进版本中。如果一个应用程序不能确定使用快速排序算法是否正确，那么它最好使用Shell排序，因为Shell排序在实现时不需要太多分析调查。然而对于大文件，快速排序算法的性能是Shell排序的5倍到10倍，它还可以更有效地处理在实际问题中可能遇到的其他类型的文件。

## 7.1 基本算法

快速排序算法是一种分治（divide-and-conquer）的排序方法。它将数组分为两个部分，然后分别对两个部分进行排序。我们将看到，划分的位置决定于输入文件中元素的初始位置。关键在于划分的过程，它遵循以下三条性质对数组进行重组：

- a[i]应该位于它最终在数组中的某个位置上；
- a[i]，…，a[i-1]中元素都比 a[i]小；
- a[i+1]，…，a[r]中元素都比 a[i]大；

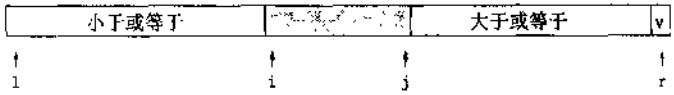
我们通过划分来完成排序，然后递归处理子文件，如图 7-1 所示。因为划分的过程至少将一个元素放到它最终应该在的位置，不难证明排序组成的递归方法是一种归纳证明。程序 7.1 是实现这一思路的递归程序。

**程序 7.1 快速排序**

如果数组中有一个或者更少的元素，就什么都不做。否则，数组先经过一个划分的预处理（见程序 7.2），预处理过程将 a[i]放在 l 和 r 之间的它应该在的某个位置上，然后对数组中元素重排，递归调用该过程完成排序。

```
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);
  }
```

我们使用一般的策略来实现划分。首先，我们选择 a[r]作为划分元素——划分后位于最终应该在的位置上的元素。然后，我们从数组的最左边向中间扫描数据，直到找到一个比划分元素大的元素；同时，从最右边向中间扫描数据，直到找到一个比划分元素小的元素。显然，使扫描停止的两个元素的位置与它们的大小关系相反，于是交换这两个元素。继续这样的过程，我们就可以保证数组中位于左侧指针左侧的元素都比划分元素小，位于右侧指针右侧的元素都比划分元素大，如下图所示。




图中，v 表示划分元素的值，i 表示左侧指针，j 表示右侧指针。从图中不难看出，左侧指针 i 在遇到大于等于划分元素的元素时停止扫描，右侧指针 j 在遇到小于等于划分元素的元

素时停止扫描，这一策略看似产生了许多对等于划分元素的不必要交换（我们将在这一节的后面解释这一策略的原因）。当扫描指针相遇时，我们需要做的就是完成划分过程，将 a[r]与右侧子文件的最左端元素交换（左指针所指向的元素）。程序 7.2 是这样一个过程的实现，图 7.2 和图 7.3 都描述了这个例子。

快速排序算法的内部循环增加了一个指针并将数组中元素与某个固定值比较。这一简化操作加速了快速排序算法：在排序算法中很难找到一个更小的内部循环。

程序 7.2 中增加了对划分元素是数组中最小元素的测试。其实可以使用一个标志来避免这样的检测：快速排序算法的内部循环很小，这样一个多余的检测可能会对性能带来影响。在这一实现过程中当划分元素是数组中最大元素时标志没有用，因为划分元素本身在停止扫描时就位于数组的最右边。本节的后面还会介绍划分过程的另一个实现。在本章的其他地方我们在遇到与划分元素相等的元素时并不停止扫描，我们可能在这样的实现中增加一个是否到达数组最右边的测试。另一方面，在 7.5 节讨论的快速排序算法的改进方案中既没必要检测也不需要在两端增加标志。

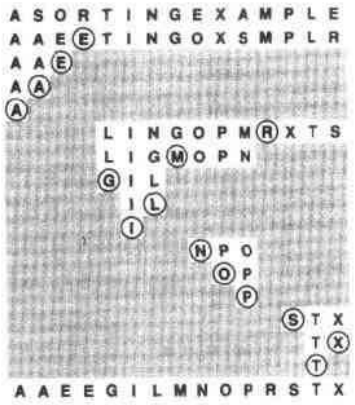



**图 7.1 快速排序的示例**

快速排序是一个递归划分过程，我们将一个文件划分，划分的原则是将一些元素放在它们最终应该在的位置上，对数组进行重组使比划分元素小的元素都在划分元素的左侧，比划分元素大的元素都在划分元素的右侧。然后分别对左右两部分分别递归处理。图中的每一行表示划分后的子文件中的内容。最后结果是一个全排序的文件。

划分过程是不稳定的，因为每个元素都有可能在交换时被移动到大量与它相等的元素（未经检测）后面。至今还没有使基于数组的快速排序算法稳定的方法。

划分过程的实现很重要。保证递归程序结束的最直接的方法就是：

（1）不对大小小于等于 1 的文件进行递归调用；

（2）只对比给定输入小的文件递归调用。

这两个策略很明显，但它可以保证给定输入时程序执行终止。例如，快速排序算法实现中有小错误——不能保证一个元素最终会被放到它应该在的位置，那么对元素划分时就可能恰好返回文件中的最大元素或最小元素，从而使得递归调用陷入死循环。

**程序 7.2 划分**

变量 r 保存了划分元素 a[r]所在的位置，i 和 j 分别是左扫描指针和右扫描指针。划分循环使得 i 增加 j 减小，while 循环保持一个不变的性质——i 左侧没有元素比 v 大，j 右侧没有元素比 v 小。一旦两个指针相遇，我们就交换 a[i]和 a[r]，即将 v 赋值给 a[i]，这样 v 左侧的元素都小于等于 v，v 右侧的元素都大于等于 v，结束了划分过程。

划分循环是一个不确定的循环，当两个指针相遇时，就通过 break 语句结束。j == l 判断用来防止划分元素是文件中的最小元素。

```
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 == l) break;
        if (i >= j) break;
        exch(a[i], a[j]);
      }
    exch(a[i], a[r]);
    return i;
  }
```

当文件中有相等的元素时，指针相遇时的处理就复杂些。我们在 i<j 结束扫描后，使用 j 代替 i-1 来确定第一次递归调用的左侧子文件的右边，从而改进划分过程。这种情况使循环叠代增加了一次，但改进了程序，因为无论扫描循环以 j 结束还是以 i 结束都指向同一元素，我们使得两个元素都位于它们最终的位置上：结束扫描的元素，即与划分元素相等的元素，以及划分元素本身。这种情况可能发生，例如在图 7.2 中如果 *R* 是 *E*。这一改变可能是值得的，因为，在特殊情况下，程序中给出的输入文件中最右边的元素是最小的，这样第一步递归调用 quicksort(a,i+1,r) 就相当于不执行（退化）。程序 7.2 中的划分实现更容易理解，然而，我们认为它是一种基本的快速排序划分过程。如果输入文件中有大量的重复元素，就会有一些其他因素影响算法。我们接下来将考虑这些因素。

**图 7.2 快速排序划分示例**

快速排序划分以选择划分元素开始。程序 7.2 选择最右边的元素 E 作为划分元素。然后从左向右扫描那些小的元素，从右向左扫描那些大的元素，交换那些使扫描停止的元素，反复进行这样的过程直至两个指针相遇。首先我们从左侧开始扫描，在 S 处停止，然后从右侧开始扫描，在 A 处停止，然后交换 S 和 A。然后，我们继续从左侧扫描，在 O 处停止，再从右扫描，在 E 处停止，然后交换 O 和 E。然后我们的扫描指针相遇：我们继续从左向右扫描，在 R 处停止，再从右继续扫描（经过 R），直至到达元素 E。我们交换 R 和划分元素 E（最右边的），至此，划分过程结束。

我们对与划分元素相等的元素可以采用三种策略来处理：两个扫描指针遇到它时都停止（如程序 7.2），只有一个指针停止而另一个跳过，两个扫描指针都跳过它。哪个策略是最好的这一问题已经在数学上仔细地研究过了，结果表明两个扫描指针都停止的策略是最好的，主要是因为这一策略尽量使重复元素多的情况下划分更均衡，而另外两种策略都使得一些输入文件的划分更不均衡。我们也将在 7.6 节讨论一个更复杂更有效的方法来处理重复元素问题。

根本上说，排序算法的效率依赖于分割文件的位置的选取，而这一位置的选取又由划分元素决定。图 7.2 表明将一个大的顺序随机的文件分割成两个较小的顺序随机的文件，但分

割位置可能在文件中的任何位置。我们更希望分割位置位于文件的中间，但没有足够的信息保证这样的性质。如果文件中元素的顺序是随机的，选择 a[r]作为划分元素和选择其他元素作为划分元素是一样的，二者在平均情况下都会在接近中间的位置分割。在 7.4 节我们将对这个算法进行分析，我们会看到这一选择与理想选择之间的比较。在 7.5 节我们会根据分析的结果讨论如何选择划分元素才能使算法更有效。

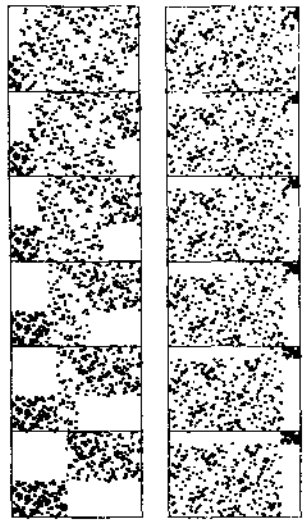


**图 7-3 快速排序算法的划分过程的动态特性**

划分过程将一个文件分成两个子文件，这两个子文件可以分别独立地排序。扫描指针左侧没有元素比划分元素大，所以左上方没有点；因为扫描指针右侧没有元素小，所以右下方没有点。从这两个例子中可以看出，划分过程将一个随机顺序的数组划分为两个小的随机顺序数组，同时一个元素（划分元素）位于对角线上。

### 练习

▷ 7.1 按前面例子的风格，给出快速排序算法应用于文件内容为 E A S Y Q U E S T I O N 每一步的排序结果。

7.2 给出文件 1 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0 是如何划分的，分别使用程序 7.2 和前面文字中建议的修改。

7.3 实现一个不含 break 语句和 goto 语句的划分算法。

● 7.4 为链表实现一个稳定的快速排序算法。

○ 7.5 对于 $N$ 个元素的文件，给出快速排序算法执行过程中最大元素被移动的最大次数。

## 7.2 快速排序算法的性能特性

尽管基本的快速排序算法已经很有用处，但它的不可靠性使得它在实际应用中处理某些特定文件时效率很低。例如，如果文件的大小为 $N$，而且文件已排序好，那么所有的划分都将退化，程序会调用它自身 $N$ 次，每一次调用只移动一个元素。

**性质 7.1** *快速排序算法在最坏情况下要进行 $N^2/2$ 次比较操作。*

根据上面的讨论，对于一个已经排好序的文件的比较次数为：

$$N+(N-1)+(N-2)+\cdots+2+1=(N+1)N/2$$

如果文件中的元素是逆序的，或者实际应用中很少出现的其他类型的文件，所有的划分也将退化（见练习 7.6）。■

这一现象不仅意味着执行时间将为 $N^2/2$，而且处理递归调用所需要的空间也将是 $N$（见 7.3 节），对于大文件来说这是不可接受的。幸运的是，有一个相对简单的办法可以减少这种在实际应用程序中典型发生的最坏情况。

快速排序算法的最好情况是每次划分过程都恰好将文件分割成两个大小完全相等的部分。这一情况使得快速排序算法的比较次数满足分治算法：

$$C_N=2C_{N/2}+N$$

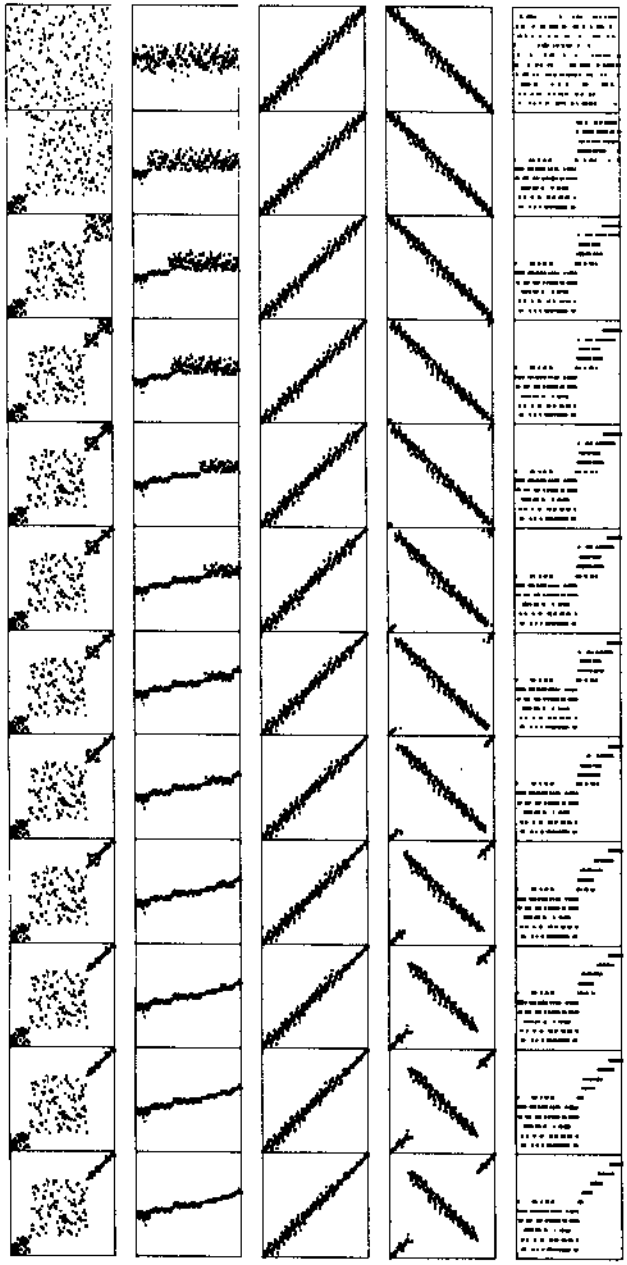


图 7.4 处理不同类型文件时快速排序算法的动态特性

在快速排序算法中随机选择划分元素对于不同的文件结果截然不同。这些图说明元素的初始位置分别是随机的、满足高斯分布的、接近顺序的、几乎逆序的和只有 10 个不同的值的随机顺序的文件的结果，对于小的子文件使用相对较大的值来中止。不参与划分的元素接近对角线，剩下的数组可以用插入排序来解决。几乎排好序的文件需要过多的划分操作。

$2C_{N/2}$ 是对两个子文件排序所需要的时间，$N$ 是使用分割指针时检查所有元素需要的时间。从第五章的内容我们可以知道这个递推方程有如下解：

$$C_N \approx N \lg N$$

尽管情况不一定总是这样，但如果每次划分的位置都在中间，平均情况就是这样。如果考虑各种可能的划分位置，那么递推方程会更加复杂，更难求解，但最终结果是类似的。

**性质 7.2**　快速排序算法平均进行 $2N\ln N$ 次比较。

对于顺序随机的 $N$ 个元素使用快速排序算法，比较次数的递推方程为：

$$C_N = N+1+\frac{1}{N}\sum_{1\le k\le N}(C_{k-1}+C_{N-k}) \qquad N \ge 2$$

并且，$C_1=C_0=0$。第 $N+1$ 项中包含了划分元素与其他元素比较的时间（在扫描指针相遇时会多比较一次）；其他来自对于每个元素 $k$ 作为划分元素的概率为 $1/k$，经过分割后剩下的两个大小分别为 $k-1$ 和 $N-k$ 的子文件。

尽管它看起来很复杂，但这个递推方程很容易解，解的过程分三步。首先，$C_0+C_1+\cdots+C_{N-1}$ 是与 $C_{N-1}+C_{N-2}+\cdots+C_0$ 相同的，所以我们可以得到这样的递推方程：

$$C_N = N+1+\frac{2}{N}\sum_{1\le k\le N} C_{k-1}$$

然后，我们可以将等式两边分别乘以 $N$ 和 $N-1$，再两两相减，来化简这个求和公式：

$$NC_N-(N-1)C_{N-1} = N(N+1)-(N-1)N+2C_{N-1}$$

这个公式就简化为递推方程：

$$NC_N = (N+1)C_{N-1}+2N$$

第三步，用 $N(N+1)$ 除以最后的方程两边，再将方程逐层展开：

$$\begin{aligned}\frac{C_N}{N+1} &= \frac{C_{N-1}}{N}+\frac{2}{N+1}\\ &= \frac{C_{N-2}}{N-1}+\frac{2}{N}+\frac{2}{N+1}\\ &= \vdots\\ &= \frac{C_2}{3}+\sum_{3\le k\le N}\frac{2}{k+1}\end{aligned}$$

确切的答案几乎等于一个级数求和（见 2.3 节）：

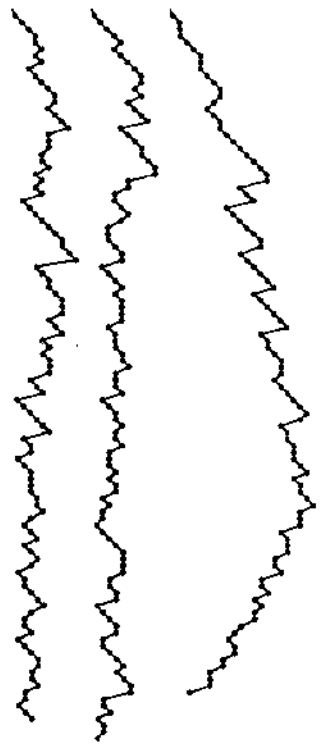


**图 7.5　快速排序算法中栈的大小**

在输入文件中元素顺序随机的情况下，快速排序算法的递归栈增长的不快，但在退化情况出现时它需要额外的空间，对于两个随机文件（左边和中间）和一个排序好的文件（右边）的栈大小如图显示。

$$\frac{C_N}{N+1} \approx 2\sum_{1\le k<N}\frac{1}{k} \approx 2\int_1^N \frac{1}{x}dx = 2\ln N$$

又因为 $2N\ln N \approx 1.39N\lg N$，所以平均情况下的比较次数比最好情况高 39%。■

上面的分析是在假定待排序的文件由键值不同且顺序随机的记录组成的情况下进行的，但程序 7.1 和程序 7.2 的实现在元素值不是完全不同以及顺序不完全随机的一些情况下，执行的速度变慢，如图 7.4 所示。如果排序算法被大量使用，或用于处理大规模文件（或者用做排序库中的通用排序算法，处理不同特性的排序文件），我们会在 7.5 节和 7.6 节考虑改进算法，使得它在实际应用中减少最坏情况的出现概率，而且还会减少 20%的平均运行时间。

**练习**

7.6 给出六个由 10 个元素组成的文件，使得快速排序算法（程序 7.1）在应用时所需要比较的次数和最坏情况一样（即所有元素都已排好序）。

7.7 写一个程序来计算 $C_N$ 的值，在 $N=10^3$，$10^4$，$10^5$ 和 $10^6$ 时，比较它的确切值和近似值 $2N\ln N$。

○ 7.8 如果待排序文件由 $N$ 个相等的元素组成，那么快速排序算法（程序 7.1）将进行多少次比较操作。

7.9 如果待排序文件有 $N$ 个元素组成，但 $N$ 个元素中只有两个不同的值（$k$ 个元素是同一个值，另外 $N-k$ 个元素是另一个值，那么快速排序算法（程序 7.1）将进行多少次比较操作。

● 7.10 写一个程序来为快速排序算法产生最好情况时的文件：即一个由 $N$ 个完全不同的元素组成的文件，而且每次划分操作时会产生两个大小最多差 1 的子文件。

## 7.3 栈大小

如同我们在第三章做的那样，我们在快速排序算法中使用一个下推栈，假设栈中包含了待排序的子文件的内容。每当我们需要对一个子文件排序时，我们就从栈中弹出一个子文件。在划分结束时，我们会产生两个待排序的子文件，此时就将这两个子文件压入栈中。在程序 7.1 中递归实现部分，系统用栈来保存这类信息。

对于一个顺序随机的文件，栈的大小最大与 $\log N$（参见“第三部分参考资料”），但在特定情况下栈的大小可能增长到等价于 $N$，如图 7-5 所示。事实上，最坏情况是文件已经是排好序的。栈增长的长度和输入文件长度成比例很少见，但快速排序的递归实现的真正困难在于存在一个基础栈，在文件很大时退化情况可能使程序因为缺少内存空间而非正常结束，这是一个库中排序程序所不希望见到的（事实上，我们通常会在运行空间溢出之前运行时间溢出）。对这种行为很难有任何保证，但我们将会在 7.5 节中看到提供这样的保证机制使得退化现象几乎不出现并不难。

程序 7.3 是一个快速排序算法的一个非递归实现，它会比较两个子文件的大小，将大的那个先压入栈。图 7.6 显示了这一策略。将这个例子与图 7.1 比较，我们可以看到使用这种策略子文件并没有发生改变，只是它们的执行顺序发生了变化。这样我们就在不影响时

间的情况下节省了空间。

将大的子文件压入栈的策略保证了栈中每一个入口都比它下面的入口的一半小，所以栈大概需要 $\lg N$ 个入口。当每次划分都在文件的中部时，栈的长度才会达到最大。对于顺序随机的文件，实际栈的最大长度都比较小；对于退化的文件来说会更小。

**程序 7.3 非递归的快速排序**

这个快速排序算法的非递归实现（见第五章）使用了一个显式的下推栈，用将参数压栈代替了递归调用，过程将会循环地从栈中弹出参数，直至栈中没有任何参数。我们将两个子文件中较大的一个压入栈，以保证在排序 $N$ 个元素时栈的最大长度是 $\lg N$（见性质 7.3）。

```
#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);
        if (i-l > r-i)
          { push2(s, l, i-1); push2(s, i+1, r); }
        else
          { push2(s, i+1, r); push2(s, l, i-1); }
      }
  }
```

**性质 7.3** 采用快速排序算法对 $N$ 个元素排序，如果两个子文件中较小的一个先被排序，那么栈的大小不会超过 $\lg N$。

栈大小的最坏情况一定小于 $T_N$，$T_N$ 满足递推方程 $T_N=T_{\lfloor N/2\rfloor}+1$，其中 $T_1=T_0=0$。这个递推方程是第五章中常见的一个（见练习 7.13）。■

这一个技术没必要像递推方程那样有效，因为它依赖于尾递归删除（tail-recursion removal）操作。如果过程的最后一个动作是调用另外一个过程，一些程序环境就会安排一些诸如先清理栈中局部变量之类的事，而不是调用后再清理。没有尾递归删除，我们就不能保证快速排序算法中的栈的大小。例如，对一个大小为 $N$ 的已经排好序的文件调用快速排序算法，递归调用大小为 $N$-1 的排好序的文件，既而又调用大小为 $N$-2 的排好序的文件，如此下去，最终使栈的深度达到了 $N$ 的常数倍。这一现象暗示了要使用一个非递归的实现来保证栈的额外增长。另一方面，一些 C++的编译器会自动地减少尾递归操作，一些机器

直接用硬件来支持函数调用——在这样的环境下，程序 7.3 中的非递归实现会比程序 7.1 中的递归实现慢。

图 7.7 进一步说明了非递归方法处理子文件（未排序好的）和递归方法处理其他文件一样。它用树结构来说明，划分元素是树中的根节点，左边的子树中元素代表划分元素左侧的子文件，右边的子树代表划分元素右侧的子文件。使用快速排序算法的递归实现会按前序来周游这棵树，而非递归的实现会按小子树优先的策略访问树。

当我们使用一个显式栈时，如我们在程序 7.3 中所做的，我们要避免一些在递归实现中的固有花费，尽管现代的程序系统在这样简单的程序中不会带来很多花费。程序 7.3 还可以进一步的改进。例如它可以将两个子文件都压入栈，栈顶的子文件随即被弹出，我们也可以仅将变量 l 和 r 压入栈中。而且 r<=l 的判断在从栈中弹出的子文件中进行即可，尽管不将这些子文件压入栈中效率会更高（见练习 7.14）。这种情况看似意义不大，但快速排序算法的递归特性确实使得在排序过程中一部分子文件的大小为 0 或 1。下面我们会对快速排序算法做更大的改进，尽量扩大上面的想法，尽可能有效地处理子文件，以获得更高的效率。

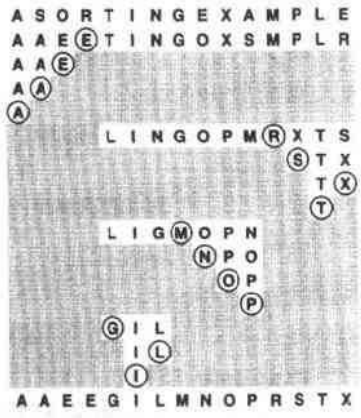



图 7.6 快速排序算法的示例（先排序小的子文件）

先排序哪个子文件并不影响快速排序算法的正确操作以及运行时间，它只会影响递归结构下的下栈的大小，图中显示的是划分后先排序小子文件的情况。

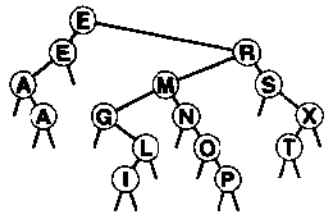



图 7.7 快速排序算法的分割树

如果我们修改图 7.1 和图 7.6 中的划分图，将一个划分元素与它两个子文件的划分元素相连，我们就得到了划分过程的静态表示。在这棵二叉树中，每一个子文件由它的划分元素代表（或是它本身，如果子文件的大小为 1），每个接点的子树是划分后两个代表子文件的树。这里没有显示空的子文件，尽管我们的递归算法中在如果划分元素是文件中的最大元素或最小元素时，在 r<l 的情况下仍递归调用。树本身并不依赖于哪个子文件先被划分。快速算法的递归实现按前序来遍历这棵树，而非递归实现以小子树优先的策略访问这棵树。

## 练习

▷ 7.11 如图 5.5 所示，当程序 7.3 用于对内容为 E A S Y Q U E S T I O N 的文件排序时，给出每一次压入和弹出栈时栈中的内容。

▷ 7.12 回答练习 7.11 中的问题，但我们每次都将右边的子文件压入栈，然后再将左边的子文件压入栈（递归实现中的情况）。

▷ 7.13 完成性质 7.3 的证明，用归纳法。

7.14 修改程序 7.3，使得它不将 r<=l 的子文件压入栈中。

▷ 7.15 给出当 $N=2^n$ 时，程序 7.3 所需要的栈的最大长度。

7.16 给出当 $N=2^n-1$ 和 $N=2^n+1$ 时，程序 7.3 所需要的栈的最大长度。

○ 7.17 在快速排序算法的实现中使用一个队列代替栈是否可行？解释你的答案。

7.18 确定并汇报你的编程环境是否实现了尾递归操作。

● 7.19 实际运行最基本的递归快速排序算法，以确定对于顺序随

机的 $N$ 个元素组成的文件需要的栈的平均长度，$N=10^3$，$N=10^4$，$N=10^5$，$N=10^6$。

●●7.20 使用快速排序算法对一个顺序随机的 $N$ 个元素组成的文件排序时，给出长度分别为 0，1，2 的子文件的平均数目。

## 7.4 小的子文件

在快速排序算法的递归实现中程序多次因为小的子文件而调用自身，这使得我们萌生了一个对快速排序算法改进的想法。我们可以在对小的子文件计数的同时尽可能地使用一个好的方法。一个明显的方法就是将递归调用过程前面的 return 语句改为一个插入排序，如下：

```
if (r-l <= M) insertion(a, l, r);
```

这里，$M$ 是一些依赖实现的参数。我们可以通过数学分析或实践研究确定 $M$ 的最佳值。在这种研究中最常用的方法就是使 $M$ 取 5~25 之间的值运行程序，$M$ 在此范围内要比原先的 $M$=1 至少快 10%（见图 7-8）。

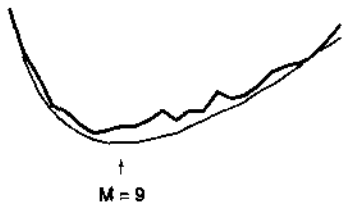



**图 7.8 小的子文件的中止值 $M$**

选择子文件的中止值可以将平均执行时间减少 10%。选择中止值并不难，将一个范围的值（5~25）应用到大多数实践中都有效。粗线（上面的）是通过实验得出的，细线（下面的）是通过分析得出的。

一种比调用插入排序更有效的解决这类小的子文件的方法，就是将前面的判断改为：

```
if (r-l <= M) return;
```

就是我们忽略那些在划分过程中的小子文件。在一个非递归的实现中就是不将大小大于等于 $M$ 的任何文件压入栈中。在划分之后，得到的就是几乎排好序的文件。如我们在 6.5 节中讨论的，这种文件适合用插入排序来处理。插入排序运行它几乎和直接将这些小文件合并差不多。这一方法使用时应该小心，因为快速排序有错误时可能会导致插入排序再不起任何作用。如果出现错误情况，就只能多做一些额外的工作。

图 7.9 说明对大文件执行这一过程的情况。即使在子文件的中止值相对较大时，快速排序部分运行得也很快，因为只有极少数文件执行了划分操作。插入排序部分执行得也很快，因为它操作的对象是几乎排好序的文件。

无论何时我们处理一个递推算法，这一技术都会带来很多好处。因为它很接近本质，我们可以保证所有递归算法在执行小的问题时效率很高；我们通常对算法中那些小的部分采取一些激进的算法以保证整个算法效率的提高。

### 练习

7.21 在快速排序中调用的插入排序是否需要标志？

7.22 改进程序 7.1，给出在被划分文件大小分别为 10，100，1000 时，比较操作在其中的比例，并输出当你排序一个大小为 $N$ 顺序随机的文件时的比例，其中 $N=10^3$，$10^4$，$10^5$ 和 $10^6$。

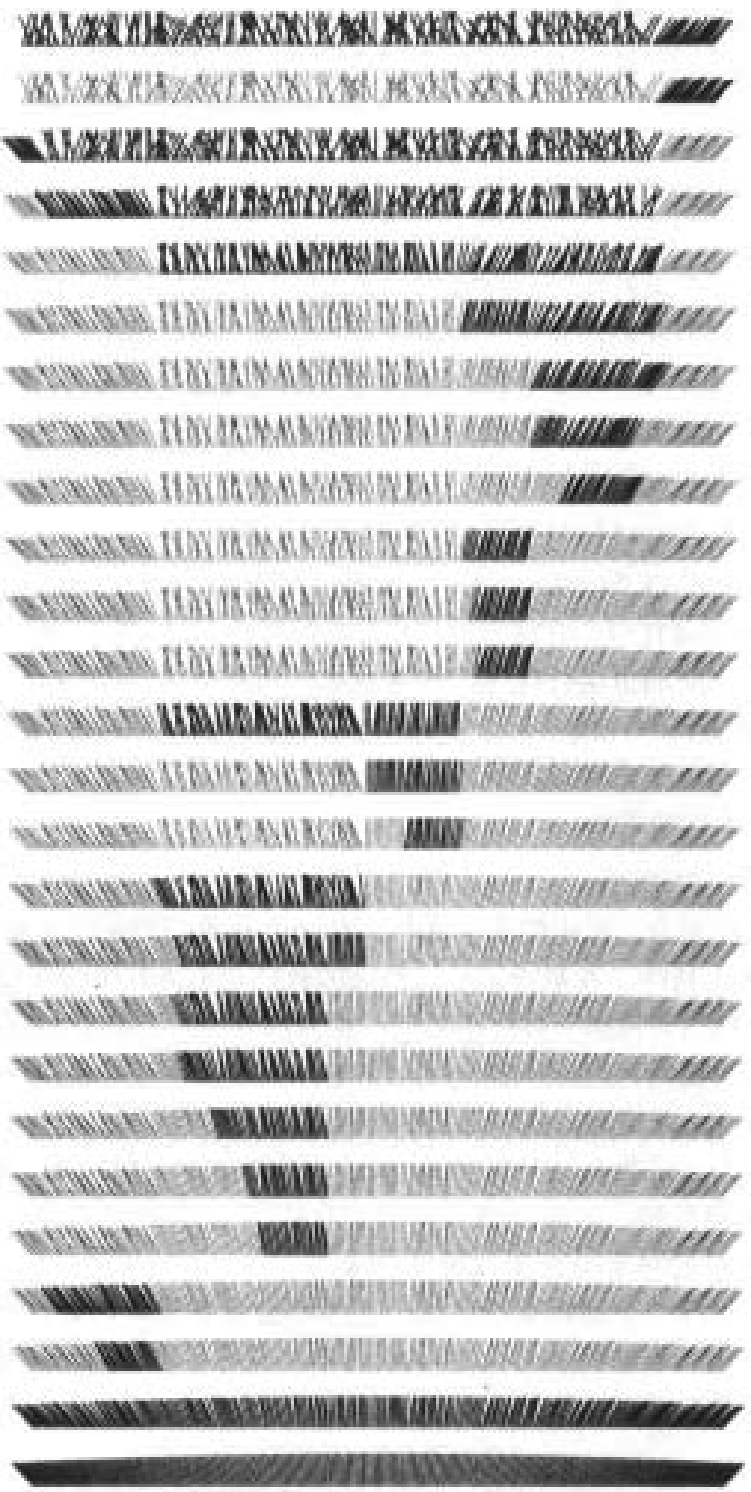


图 7.9 快速排序算法中的比较操作

快速排序子文件是独立的。这张图给出了在中止值为 15 或更小时对 200 个元素的文件排序时，划分得到的子文件的情况。我们数出那些深色的线，就可以得出比较操作的次数。在这个例子中，数组中每个位置大概参与了 6–7 次子文件的排序。

○ 7.23 实现一个以下要求的递归的快速排序算法：处理小的子文件时使用插入排序算法，假设中止值为 $M$，如果文件由 $N$ 个元素组成，元素顺序随机，在你的计算环境中用实验数据说明使程序 7.4 中运行最快的 $M$ 的值，$N=10^3$，$10^4$，$10^5$ 和 $10^6$。

7.24 与练习 7.23 类似，设计一个非递归的快速排序算法。

7.25 与练习 7.23 类似，假设待排序的记录中包含了一个主键和一个指向其他信息的指针 b（但我们这里不使用指针排序）。

● 7.26 写一个输出直方图的程序：假设文件大小为 $N$，中止值为 $M$，运行快速排序算法时进行插入排序的子文件的大小为纵坐标。在 $M=10$，100，1000 和 $N=10^3$，$10^4$，$10^5$ 和 $10^6$ 时运行你的程序。

7.27 在中止值为 $M$ 时，对一个由 N 个元素组成的顺序随机的文件，通过实验运行快速排序算法来确定栈的平均长度，其中 $M=10$，100，1000 和 $N=10^3$，$10^4$，$10^5$ 和 $10^6$。

## 7.5 利用三个元素的中间元素来划分

对快速排序算法的另一个改进措施就是使用一个尽可能将文件中分的划分元素。这有几种可能性。一个安全的选择就是避免最坏情况，使用数组中的一个随机元素作为划分元素。这样最坏情况发生的可能性就相对很小。这种方法是概率算法的一个简单例子，概率算法利用随机性来取得较大可能性的性能。我们将在本书后面看到大量应用概率的算法设计，这些算法时常依赖输入。例如快速排序算法，在实际应用中你不可能用随机数字产生器来产生数据供它使用；简单的随意选择可能也是有效的。

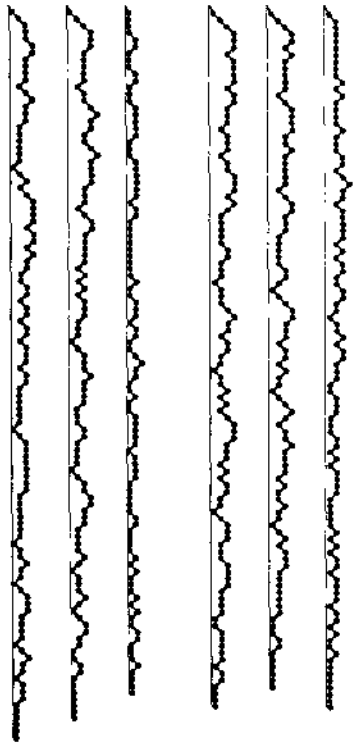


图 7.10 快速排序算法的改进版本中栈的大小

先排序小的子文件保证了栈的长度是最坏情况的对数。这里显示的栈大小对应的文件与图 7.5 是相同的，在先排序小的子文件的基础上增加了使用三个元素的中间元素。该图没有给出运行时间，运行时间依赖于栈中的文件的大小，而不是文件的数目。例如，第三个文件（已经排好序）不需要太多的栈空间，但运行时间却很长，因为待排序的子文件通常很大。

另一个选择划分元素的方法，就是从文件中取出三个元素，使用三个元素的中间元素作为划分元素。取数组中的左边元素、中间元素和右边元素，对这三个数排序（可以利用第六章中的三交换方法来排序），用 a[r-1]与中间元素交换，然后对 a[l+1]，…，a[r-2]进行划分。这一改进被称为三个元素的中间元素法（median-of-three method）。

三个元素的中间元素法在以下几个方面改进了快速排序算法。首先，它使得最坏情况在实际排序中几乎不可能发生。如果排序需要 $N^2$ 时间，被检测的三个元素中有两个必然在大的元素所在子文件中，或在小的元素所在的子文件中，这一情况在大多数划分中都会出现。其次，它减少了划分对标志的依赖，因为这个函数用划分之前被检测的三个元素中的一个。最后，它使总的平均运行时间大约减少了 6%。

三个元素的中间元素法和小的子文件的中止值结合起来可以将原始的递归实现的快速算法提高 20%~25%。程序 7.4 是中和了这些改进方法后的实现。

我们可能还会考虑继续改进程序、消除递归、用内联代码代替函数调用、使用标志等。然而，在现代的机器上，这样的过程调用一般都很有效，它们并不在循环内部。更重要的是，对小的子文件使用中止值已经顶替了那些可能造成的额外开销（在内部循环的外部）。使用带栈的非递归实现主要原因是在限制栈大小方面给予保证（见图 7.10）。

进一步的提高程序的性能也是可能的（例如，我们可以使用 5 个或者更多元素的中间元素），但获得的时间对于随机文件来说微乎其微。我们会意识到用汇编语言或机器语言对内部循环的编码（或者是整个程序的编码）可以更节省时间。这些观察已经被专家证明在大多数情况下对重要的排序应用程序都有效（参见“第三部分参考资料”）。

对于顺序随机的文件，程序 7.4 的第一个交换操作是多余的。我们使用这一交换操作不仅仅是因为它将对排序好的文件优化划分，还因为它保证在实际应用中不会出现异常情况（例如，见练习 7.33）。图 7.11 说明了对于大量不同类型的文件，在划分时选择中间元素时的情况。

**程序 7.4 改进的快速排序算法**

选择第一个、中间和最后一个元素的中间元素作为划分元素，去掉小的子文件的递归调用，这一系列操作改进了快速算法。实现中取数组中第一个、中间和最后一个元素的中间元素作为划分元素（否则将这些元素排除在划分过程之外）。大小为 11 或更小的文件在划分过程中被忽略，然后执行第六章中的插入排序 insertion 来完成排序。

```
static const int M = 10;
template <class Item>
void quicksort(Item a[], int l, int r)
  {
    if (r-l <= M) return;
    exch(a[(l+r)/2], a[r-1]);
    compexch(a[l], a[r-1]);
      compexch(a[l], a[r]);
        compexch(a[r-1], a[r]);
    int i = partition(a, l+1, r-1);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
  }
template <class Item>
void hybridsort(Item a[], int l, int r)
  { quicksort(a, l, r); insertion(a, l, r); }
```

使用三个元素的中间元素法是在这样的前提下的一种特殊情况：我们取样一个未知的文

件，用样本元素的性质来估计整个文件性质。对于快速排序算法，我们希望通过估计一个中间值来使划分更平衡。算法的这一特性使得我们不必要有一个很好的估计（如果这样的估计需要计算很长时间，我们甚至不需要进行估计），我们只想避免太糟糕的估计。如果我们随机地选取一个样本，文件中只有一个元素，那么无论输入是什么我们都会得到一个实际上肯定运行很快的随机算法。如果我们随机地从文件中选取 3 个或 5 个元素，然后使用样本的中间值来划分，我们就会得到一个较好的划分，但改进的时间被取样本所需要的时间抵消了。

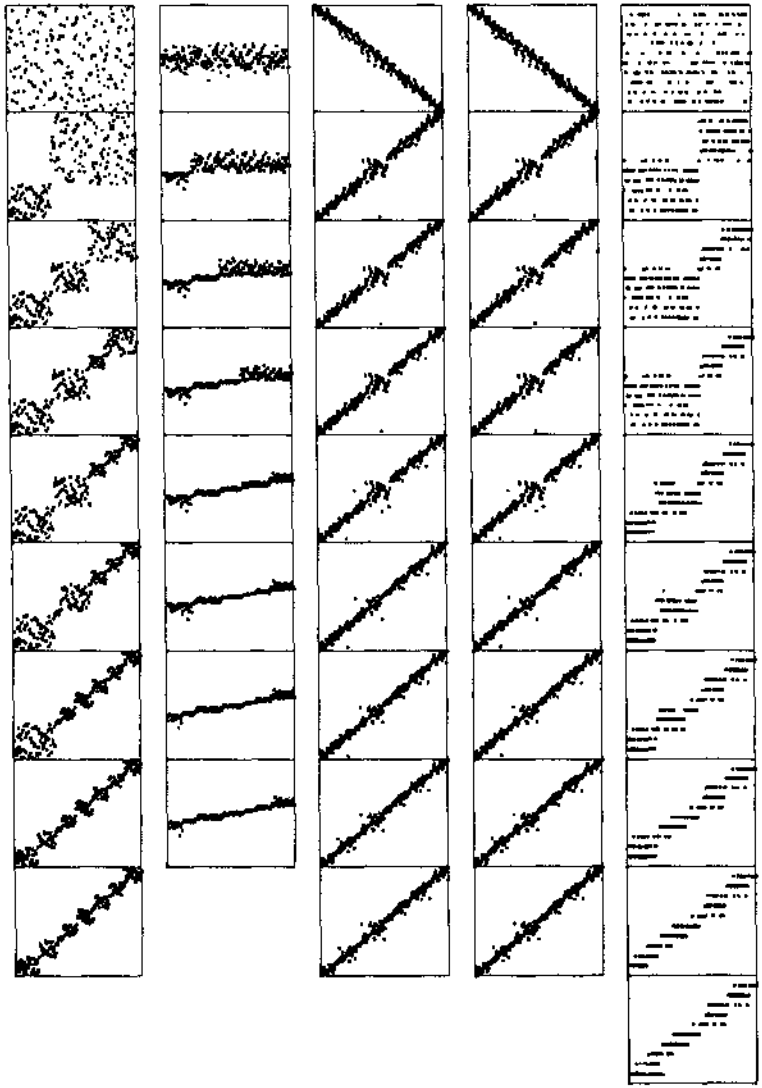


**图 7.11 对于不同类型的文件采用三个元素的中间元素的快速排序算法的动态特性**

将快速排序算法修改为使用三个元素的中间元素使划分过程更加坚固。图 7-4 所示的退化类型的文件在这里运行得很好。另一个达到这一目标的方法就是使用随机的划分元素。

快速排序算法之所以得到了广泛应用是因为它在很多情况下都能运行得很好。其他方法可能更适合解决某些特定情况，而快速排序算法可以比其他方法处理更多的排序问题，而且它通常会比其他可选择的方案更快。表 7.1 给出了一些支持这一结论的实验数据。

**表 7.1 基本排序算法的实验研究**

对于大的顺序随机的文件，快速排序（程序 7.1）比 Shell 排序快两倍（程序 6.6）。使用对小的子文件使用中止值和取三个元素的中间元素的改进算法都分别使运行时间减少了 10%（程序 7.4）。

| | | 基本快速排序 | | | 采用三个元素的中间元素的快速排序算法 | | |
|---|---|---|---|---|---|---|---|
| $N$ | shellsort | $M$=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 |

## 练习

7.28 我们采用的取三个元素的中间元素的方法不能完全保证样本元素不参加划分过程。一个理由就是它们可能是标志。请给出另一个原因。

7.29 实现一个快速排序算法，算法在划分前随机地从文件中取五个元素的中间元素做划分元素。保证所取的样本元素不参加划分（见练习 7.28）。用大量的随机文件来比较你的算法和取三个元素的中间元素的快速排序算法的性能。

7.30 在大量的不随机的文件上运行练习 7.29 中你的程序，例如排好序的文件、逆序的文件以及所有元素值都相等的文件。它运行这些文件的性能与运行顺序随机的文件的性能有何区别？

●●7.31 实现一个使用 $2^k-1$ 个样本的快速排序算法。首先对样本进行排序，然后在样本中的中间元素基础上进行划分，接着将剩余的两个样本元素放到每个子文件中，这样它们可以在子文件中使用，而不会再被排序。这个算法被成为样本排序（samplesort）。

●●7.32 运行实验数据，确定在样本排序中样本个数的最佳值，令 $N=10^3$，$10^4$，$10^5$ 和 $10^6$。使用快速排序算法和样本排序算法处理，对这样的样本排序有什么区别？

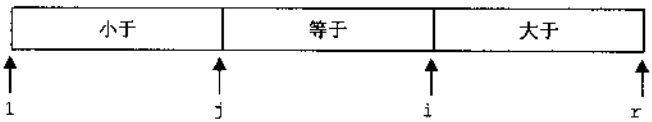
● 7.33 改变程序 7.4，忽略第一次交换操作，扫描时忽略与划分元素相等的元素，说明这样的程序处理一个逆序的文件时运行的时间是平方量级。

## 7.6 重复值

带有大量重复值的文件排序问题在实际应用中经常出现。例如，我们可能希望按出生年来排序一个人员信息文件，甚至用排序来分辨性别。

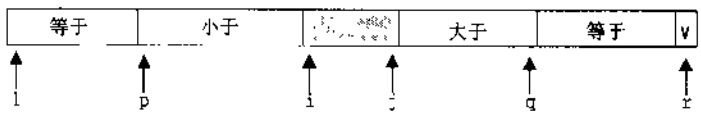
待排序的文件中有大量的重复值时，我们前面所考虑的快速排序算法的效率就会变得不可思议的低，但它们仍有改进的可能。例如，如果一个文件完全由相等的值组成，它就不需要再进行任何排序，但我们前面的算法仍然进行划分直至得到小的子文件，无论文件有多大（见练习 7.8）。在输入文件中有大量重复值时，快速排序算法的递归特性使得子文件经常仅由重复值组成，这是可以改进的地方。

一个直接的想法是将文件分成三个部分：一个是比划分元素小的；一部分是和划分元素一样大的；另一部分比划分元素大：




完成这样的划分过程比完成两路划分过程更复杂，目前已经有很多方法用于解决这一问题。Dijkstra 的这个程序作为“荷兰国旗问题”很流行，因为三种可能的范围对应着旗子上的三种颜色（参见“第三部分参考资料”）。对于快速排序算法，我们增加一个限制——扫描一次文件就完成这项工作，而不是两次扫描完成，两次扫描会使快速排序算法的速度慢 2 倍，即使完全没有重复键。

Bentley 和 McIlroy 于 1993 年发明了一个很好的三路划分方法，它仅仅将标准的划分过程做如下改动：扫描时将遇到的左子文件中与划分元素相等的元素放到文件的最左边，将遇到的右边子文件中与划分元素相等的元素放到文件的最右边。在划分过程中，我们保证了这样一种情形：




然后，当两个扫描指针相遇时相等的值的确切位置就知道了，我们只要将所有键与划分元素相等的元素交换即可。这一方法不是十分满足三路划分方法中要求的一遍扫描的要求，但额外的工作量只等价于重复值的个数。这一事实说明两点：第一，即使在没有重复值的情况下，该方法也会运行得很好，因为没有额外的工作。第二，如果只有值个数不变，那么这个方法几乎是线性的：每个划分阶段都将与划分元素值相等的值去掉，所以每个值只参加有限次划分。

图 7.12 说明了三路划分算法的一个例子，程序 7.5 是基于这一方法快速排序算法的实现。这个实现只需要在交换循环中增加两个 if 语句以及两个 for 循环，for 循环主要是将与

划分元素相等的元素放到应该在的位置。看起来在保证三路划分上它似乎比其他方法用了更少的代码。更重要的是，它不仅尽可能有效地处理重复值问题，而且在没有重复值时它的额外工作也很少。

## 练习

▷ 7.34 解释当一个顺序随机的文件具有以下两个特性时运行程序 7.5 会出现什么情况：（1）只有两个不同的值；（3）有三个不同的值。

7.35 修改程序 7.1，使得它在遇到子文件中所有值都相等时返回。在有 $t$ 个不同的值的顺序随机的大文件上运行你的程序，比较你的程序和程序 7.1 的性能，$t$=2，5 和 10。

7.36 假设我们在程序 7.2 中跳过那些与划分元素相等的元素，而不是在遇到它们时就停止扫描，证明程序 7.1 的运行时间会是这种情况的平方。

● 7.37 有 $O(1)$个不同值的文件，证明练习 7.36 中程序的运行时间是平方量级。

7.38 写一个确定在文件中不同的值的个数的程序。使用你的程序来统计由 $N$ 个 0~$M$−1 之间的整数组成的顺序随机的文件中不同的值的个数，其中 $M$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$。

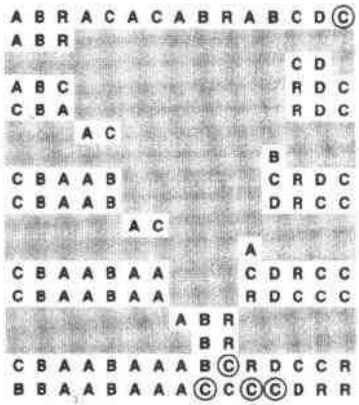


**图 7.12 三路划分**

该图显示了将所有与划分元素相等的元素放到应该在的位置的过程。如图 7-12 所示，我们从左开始扫描直至找到一个不小于划分元素的元素，从右扫描直至找到一个不大于划分元素小的元素，然后将它们交换。如果左边的元素与划分元素相等，那么我们将它交换到数组的最左边，对右边做类似的处理。当两个指针相遇时，我们和前面一样将划分元素放到它应该在的位置，然后交换两侧所有与它值相等的元素，使得它们在应该在的位置。

**程序 7.5 三路划分的快速排序算法**

这个程序基于将数组划分为三个部分：比划分元素小的元素（在 a[l]，…，a[j]中），与划分元素相等的元素（在 a[j+1]，…，a[i-1]中），和比划分元素大的元素（在 a[i]，…，a[r]中）。然后排序算法以两个递归调用结束。

因为程序中 l 和 p 是左侧与划分元素相等的元素所在的位置范围，q 和 r 是右侧与划分元素相等的元素所在的位置范围。在划分循环过程中，当扫描指针停止时，i 和 j 所指向的元素交换，它会检查这两项是否与划分元素相等。如果左边的元素与划分元素相等，那么就将它交换到数组的左边，如果右边的元素与划分元素相等，那么就将它交换到数组的右边。

两个指针相遇后，所有与划分元素相等的元素都被交换到它们应该在的位置，剩下的两个子文件分别进行递归调用。

```
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);
  }
```

## 7.7 字符串和向量

如果被排序的元素是字符串，我们在这一章可以使用类似程序 6.11 中的数据类型来实现快速排序算法。尽管这一方法提供了一个正确而有效的实现（对于大文件，它比我们目前看到的其他任何方法都快），但仍有一个隐藏的代价值得考虑。

问题出在 strcmp 函数上，strcmp 函数从左到右地依次比较两个字符串，它所需要的时间等价于两个字符串前面相等的字符的个数。对于快速排序算法中后面的划分过程，当那些值很接近时，这个比较的时间就相对较长。又因为快速排序算法的递归特性，算法的开销几乎都花在后半段上，所有改进字符串匹配过程是值得的。

例如，我们考虑一个大小为 5 内容为 discreet、discredit、discrete、discrepancy 和 discretion 的文件。对这些关键字排序时每次比较的字符至少是 7，如果我们知道前六个字符相等的额外信息，我们完全可以从第 7 个字符开始比较。

我们在 7.6 节讨论的三路划分过程提供了利用这一现象的好方法。在每次划分时，我们只检查一个字符（假设在位置 d），假设待排序的两个元素从 0 到 d-1 都是相等的。我们在三路划分时只要把第 d 个字符小于划分元素的关键字放在左侧，把第 d 个字符等于划分元素的关键字放在中间，把第 d 个字符大于划分元素的关键字放在右边。然后我们像前面一样继续，只是我们对中间的子文件进行排序，从 d+1 个字符开始。这一方法很适用于字符串排序，运行效率很高（见表 7.2）。我们在这里有一个令人信服的例子。

表 7-2 不同的快速排序算法的实验研究

这个表中给出了在对 Moby Dick 中的前 $N$ 个单词排序时，几个不同版本的快速排序算法的相对开销。对小的子文件直接使用插入排序，或者暂时忽略它们在文件排好序后再调用插入排序，是两种等价的策略，但对整数键排序时开销更小，这是因为字符串的比较更费时。如果我们在划分过程中遇到重复键时不停止，那么对一个所有键都相等的文件排序时间是平方量级；这一结果在这个例子中很明显，因为数据中有大量出现频繁的单词。因为同样的原因，三路划分很有效，它比系统的排序算法快 30%~35%。

| $N$ | V | I | M | Q | X | T |
|---|---|---|---|---|---|---|
| 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 |

其中：

V 快速排序（程序 7.1）

I 对小的子文件的插入排序

M 忽略小的子文件，然后插入排序

Q 系统 qsort

X 扫描重复值（当键值全部相等时，复杂度为平方级的）

T 三路划分（程序 7.5）

为了实现排序算法，我们需要一个更一般化的允许访问字符键的抽象类型。C++中实现的方法很简单。然而我们到第十章才会考虑细节问题，第十章中我们还会考虑大量排序中用到的技术，这些技术很简单地将排序的键分解成小的部分。

这个解决方案可以处理多维的排序问题，也就是说被排序的关键字可能是向量，按照向量中的第一个部分来排序，如果第一部分相等再比较第二部分，如此下去。如果每个部分没有重复的值，问题就变成了对第一部分的排序；然而，在一个典型的应用程序中，每一个部分可能只有几个不同的值，三路划分（在中间划分时比较下一个部分）。这一情况由 Hoare 在他的原始论文中就有讨论，而且是一个重要的应用。

## 练习

7.39 讨论改进选择排序、插入排序、冒泡排序和 Shell 排序等算法对字符串排序的可能性。

○ 7.40 标准的快速排序程序（程序 7.1，使用程序 6.11 中的字符串类型）在对一个由长度为 $t$ 的 $N$ 个完全相同的字符串组成的文件时，需要比较几个字符？根据文中建议的修改回答相同的问题。

## 7.8 选择

寻找一些数字的中间数是与排序有关但又不是排序的重要应用。这一操作是许多数据处理应用程序或统计计算应用程序中经常出现的。一个解决办法就是将这些数字排序，再找到中间数，但使用快速排序划分过程可以做得更好。

寻找中间元素是选择操作的一个特例，所谓选择操作就是寻找一个数字集合中第 $k$ 小的数字。对于一个数据项，算法只有在确定有 $k$-1 个元素比它小有 $N$-$k$ 个元素比它大之后，才能保证这个数据是第 $k$ 小的，大多数选择算法都将文件中所有第 $k$ 小的元素返回，而没有进行过多的处理。

选择有很多实际应用程序。使用中间元素和其他排序统计（order statistics）来将文件分成小的组是很普通的。通常，一个大文件中只有小部分被保存并做进一步处理；在这样的例子中，一个程序可以选择文件中 10%的元素，而不需要对整个文件排序。另一个重要的例子是将使用关于中间元素的划分过程作为许多分治算法的第一步。

我们已经看到了一个适合直接用于选择的算法。如果 k 很小，那么选择排序（见第六章）就可以工作得很好，它的运行时间等价于 $Nk$：首先寻找最小的元素，然后通过找余下元素的最小元素来找第二小的元素，如此下去。对于大一点的 $k$，我们将在第九章中看到一些运行时间等价于 $N\log k$ 的方法。

对于所有 $k$，快速排序算法中的划分过程可是使选择方法的平均运行时间达到线性。回想一下，快速排序算法中将数组 a[l]，...，a[r]重组，返回整数 i，a[l]到 a[i -1]都小于等于 a[i]，从 a[i+1]到 a[r]都大于等于 a[i]，如果 k 等于 i，那么我们的工作就完成了。否则，如果 k<i，那么我们需要进一步对左边的子文件进行处理，如果 k>i，那么我们需要进一步对右边的子文件进行处理。这个方案对应的递归程序是 7.6。对一个小文件应用这一过程的例子如图 7.13 所示。

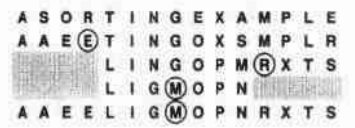



**图 7.13 中间元素的选择**

对于排序示例中的关键字，基于划分的选择只用三次递归调用就找到了中间元素。第一次调用，我们要找到大小为 15 的文件中第 8 小的元素，划分过程为我们找到了第 4 小的元素（E）；第二次递归调用，我们要找到大小为 11 的文件中第 4 小的元素，划分过程为我们找到第 8 小的元素（R）；第三次递归调用，我们要找到大小为 7 的文件中的第四小的元素，划分过程找到了它（M）。经过这三次递归调用后，文件中元素被重组了，中间元素都位于适当的位置，小的元素位于左边，大的元素位于右边（相等的元素可以在任何一边），但不是全排序好的。

**程序 7.6 选择**

这个划分过程目的在于寻找数组中第(k-1)个最小元素(在 a[k]中)：它将数组重组，使得 a[1]，...，a[k-1]都小于等于 a[k]，a[k+1]，...，a[r]都大于等于 a[k]。

例如，我们可以调用 selection(a,0,N-1,N/2)来根据中间元素对数组进行划分，中间元素位于 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);
  }
```

程序 7.7 的非递归版本来自程序 7.6 的递归版本。既然程序常常以调用自己而结束，我们可以简单地重新设置参数，从头开始调用。我们在不用栈的情况下去掉了递归，也就减少了关于数组索引 k 的计算。

**程序 7.7 非递归的选择**

选择的非递归实现只是完成划分，如果划分的位置落在指定位置的左边，那么就将左边的指针 l 向右移动；如果划分的位置落在指定位置的右边，那么将右边的指针 r 向左移动。

```
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;
      }
  }
```

**性质 7.4** 基于快速排序的选择平均情况下是线性的。

如同我们在快速排序算法那部分一样，我们只是粗略地讨论这个问题，在文件很大时，每一次划分大约恰好将数组等分，所以整个过程所需要的时间大概是 $N+N/2+N/4+N/8+\cdots=2N$ 次比较。当然这一粗略的讨论与事实相距甚远。在 7.2 节的快速排序算法部分（参见“第三部分参考资料”），我们曾经给出一个类似但更复杂的讨论，结果是平均比较次数大约为

$$2N+2k\ln(N/k)+2(N-k)\ln(N/(N-k))$$

也就是对于任意一个 $k$，比较次数是线性的。例如，$k=N/2$，这个公式就表明大概需要 $(2+2\ln 2)N$ 次比较来寻找中间元素。■

图 7-14 给出了一个寻找大文件的中间元素的方法的示例。图中只有一个子文件，每次调用时它都被去掉相同比例的部分，所以这一过程最多进行 $O(\log N)$步。我们可以用采样的办法来加速程序，但这样做需要仔细的练习（见练习 7.45）。

最坏情况和快速排序差不多——使用这一方法寻找已排好序的文件中的最小元素，所需要的运行时间是平方量级。我们还可以修改这个基于快速排序的选择过程，使它的运行时间是线性的。这些修改在理论上都很重要，但很复杂，在实际应用中用处不大。

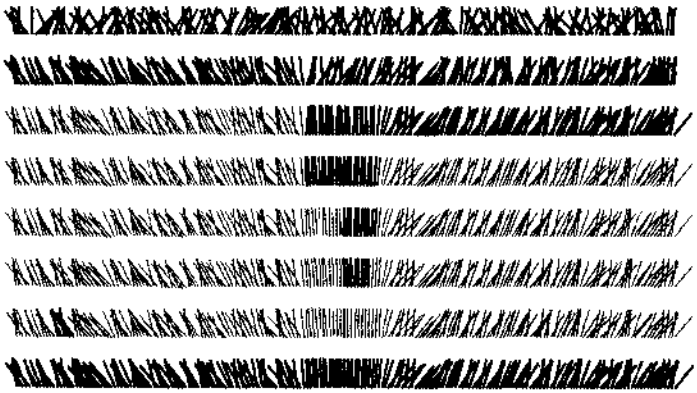


**图 7.14　用划分来选择中间元素**

利用对被寻找元素所在的子文件的划分来完成选择过程．划分的位置决定了将左指针向右移动，还是将右指针向左移动。

## 练习

7.41　使用选择排序来寻找第 $N$ 小的元素平均需要进行多少比较。

7.42　使用选择排序来寻找第 $\alpha N$ 小的元素平均需要进行多少比较，$\alpha$=0.1，0.2，…，0.9?

7.43　使用选择排序来寻找第 $N$ 小的元素，最坏情况下需要进行多少比较?

7.44　写一个高效的程序来对数组进行重组，使得所有与中间元素值相等的元素都在它们应该在的位置上，小的元素在它左边，大的元素在它右边。

●●7.45　研究使用取样来改进选择过程的想法。提示：使用中间元素不一定总是有用。

● 7.46　实现一个基于三路划分的选择算法，用它来对顺序随机的文件进行排序，组成文件的元素有 $t$ 个不同值，$t$=2，5 和 10。

# 第八章

# 归并及归并排序

我们在第七章研究的快速排序算法系列是以选择操作为基础的：找出文件中第 $k$ 小的数据。我们发现进行选择操作类似于将文件分成两部分，$k$ 个小的数据项和 $N-k$ 个大的数据项。本章中，我们研究一组基于补充算法（归并）的排序算法：将两个排好序的文件合并构成一个更大的排好序的文件。归并是一个直接分治（参看第 5.2 节）的排序算法的基础操作，类似地，两种操作都可以很容易实现。

某种意义上说，选择操作和归并操作是互补的，因为选择操作将一个文件拆成两个独立的文件，而归并则将两独立的文件合并成一个文件。当我们应用分治法范例来产生排序方法时，这两个操作的对比更强烈。我们可以对文件进行重新整理，以便当文件两部分排好序后，文件就排好序了；另一种方法是，可以将文件拆成两个部分，对两个部分进行排序，然后归并两个部分组成一个排好序的文件。我们已经讨论过第一种方法：那就是快速排序，包括一个选择操作过程，紧跟的是两个递归调用。本章中，我们将研究归并排序，它的执行过程和快速排序相似，两个递归调用之前进行的是归并操作。

归并排序最吸引人的特性之一是对 $N$ 个元素的文件进行排序，它的运行时间与 $N\log N$ 成比例，与输入的数据无关。在第九章中，我们会看到另一个运行时间与 $N\log N$ 成比例的算法，称为堆排序。归并排序最主要的缺点是直接执行时需要与 $N$ 成比例的额外内存空间。我们可以克服这个障碍，但是做法很复杂而且花费很大，因此在实际中一般不值得这样做，特别是当可以选择堆排序时。归并排序和堆排序的复杂度一样，它内部循环的长度在快速排序和堆排序的之间，因此当运行速度很重要，不能有最坏的运行情况，又有足够的空间可以使用时，可以考虑使用归并排序。

运行时间一定是 $N\log N$ 可能是一个缺点。因为，例如，在第六章中，我们讨论了一些基本的排序方法，这些方法在某些特殊的情况下运行时间可以为线性的。如文件一开始已基本排好序的情况下，或者只有一些特殊的键值的时候。相反，归并排序的运行时间只与输入文件的关键字的数目有关，而与它们的顺序没有太大关系。

归并排序是一种稳定的排序，当算法的稳定性很重要的时候，该特性使归并排序可作为一种选择。与归并排序竞争的算法，如快速排序和堆排序，是不稳定的。使这些算法变

成稳定有很多的方法，但需要额外的内存空间；因此当稳定性是基本的要求时，归并排序中需要额外内存空间这个缺点就变得没那么重要了。

归并排序另一个特性是它的执行过程中基本是按顺序访问数据的（一个数据项接着另一个数据项），这点在某些情况下是重要的。例如，归并排序是链表排序选择的方法之一，而链表排序中只可以按顺序地访问数据项。类似地，我们在第十一章中会看到，在特殊目的和高效的机器中通常选用归并操作作为排序的基本操作，因为在该种情况下，按顺序存储数据运行得最快。

## 8.1 二路归并

对两个已排好序的文件，我们可以简单地将它们合并成一个排好序的输出文件，分别取出两个输入文件中最小的数据，将这两个数据中较小的数放到输出文件中，循环操作，直到两个输入文件的数据项都已输出。在本节和下一节中，我们会讨论该抽象操作的几个执行过程。该操作的运行时间是输出元素数目的线性函数，因为在已排好序的文件中找出最小元素的执行时间必然是一个常数，其中文件数据项使用的数据结构支持常数时间的顺序访问，如数组或链表。该过程是对两个文件进行归并（二路归并）；在第十一章中，我们会详细讨论多个文件的归并（多路归并），该过程是对多个文件进行操作的。多个文件归并操作是外部排序中的重要操作，该算法也会在第十一章中详细讨论。

**程序 8.1 归并**

为了要将两个已排好序的数组 a 和 b 合并成一个排好序的数组 c，使用一个 for 循环，该循环每步中将一个元素放到 c 中。如果 a 已为空，就从 b 中读取元素；如果 b 已为空，就从 a 中读取元素；如果两个数组中都还包含数据，就将 a、b 剩余数据中最小的那项放到 c。除了固有的 a、b 数组都是已排好序的假设外，该执行程序还假设了 c 是独立于 a、b 的（即，彼此间没有覆盖或分享内存空间）。

```
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++];
      }
  }
```

开始时，我们假设有两个独立的排好序的整数数组 a[0] ,...,a[N-1]及 b[0] ,...,b[M-1],

要将它们合并成第三个数组 c[0],...,c[N+M-1]。很明显，如程序 8.1 所提到的，执行过程是要从 a、b 剩余的元素中提取最小的那项填入 c。该执行过程很简单，不过我们现在要讨论它的两个重要特性。

首先，该执行过程假设数组是独立的。特别是，如果 a 和 b 是巨大的数组，那么就需要第三个同样巨大的数组 c 存放输出文件。由于需要与进行归并的文件大小成比例的额外内存空间，所以我们希望有另一种合适的方法可以不需要额外的巨大内存空间，例如对排好序的数组 a[l]，...，a[m]和数组 a[m+1]，...，a[r]，通过在 a[l]，...，a[r]中移动各元素，将它们合并成一个排好序的数组。这个问题很值得我们立刻停下来想想如何实现。它看起来很容易解决；但事实上，我们现在所知的相关算法都很复杂，特别是和程序 8.1 比起来。要想出一个更好的合适归并算法真的很困难，我们会在第 8.2 节中再讨论这个问题。

归并算法有自己独特的运行特点。例如，在一个典型的数据处理环境中，可能包含一个巨大的（已排好序）数据文件，需要经常地添加一些新的数据项。一种方法是将每组输入数据批量地加入到主文件中，然后对整个文件重新进行排序。但更好的方法是，将新输入的数据项（较小）进行排序，然后将得到的排好序的小文件与大的主文件合并。还有一些其他的应用特例，使归并算法更有研究的价值。本章中主要讨论基于归并算法的排序方法。

**练习**

8.1 假设要将一个大小为 $N$ 的排好序的文件与大小为 $M$ 的排好序的文件合并，其中 $M$ 比 $N$ 要小得多。$N=10^3$，$10^6$ 和 $10^9$ 的情况下，和重新排序操作相比，基于归并算法的方法可以快多少倍？结果用 $M$ 的函数表示。假设，对大小为 $N$ 的文件，排序程序的运行时间是 $c_1N\lg N$ 秒，对大小为 $N$ 和 $M$ 的文件，归并程序的运行时间是 $c_2$（$N+M$）秒，其中 $c_1 \approx c_2$。

8.2 和练习 8.1 的两种算法相比，使用插入排序的结果又如何？（假设小文件是随机排列的，因此在大文件中每个插入操作要移动一半路程，运行时间大概是 $c_3MN/2$，$c_3$ 约等于其他两个常数。）

8.3 使用程序 8.1 进行恰当的归并操作，描述对序列 A E Q S U Y E I N O S T 调用 merge（a，a，N/2，a+N/2，N-N/2）的运行情况。

○ 8.4 按练习 8.3 的方法调用程序 8.1 时，是否当且仅当两个输入的子序列是已排好序的，所产生的输出才是正确的？证明你的答案，并给出一个例子。

## 8.2 抽象的合适归并算法

虽然执行归并算法需要额外的内存空间，我们仍会在这里讨论用于排序操作的抽象的合适归并算法。在下一个归并执行程序中，将使用接口程序 merge(al,m,r)来声明 merge 子过程，将数组 a[l]，...，a[m]和 a[m+1]，...，a[r]合并成一个排好序的文件，结果保存在 a[l]，...，a[r]中。执行中，我们可以首先将所有数据复制到一个辅助数组中，接着使用程序 8.1 中的基本方法；我们还需考虑对该过程进行改进。尽管辅助数组需要的额外空间是固定的开销，

但 8.4 节的改进，会避免复制数组需要的额外时间。

基本归并算法中第二个值得注意的特性是内部循环中包括了两个测试操作，这两个操作判断对两个输入数组访问是否已到了结尾。当然，这两个测试通常得到的是失败的信息，这种情况下，就可以改用标志关键字代替测试操作。也就是说，如果在数组 a 和辅助数组的最后加上一个包含比其他关键字都要大的关键字的数据项时，就可除去测试操作，因为当数组 a（b）访问完后，该标志使下一个要加入数组 c 的元素一定是从数组 b（a）中取出，直到完成了整个合并操作。

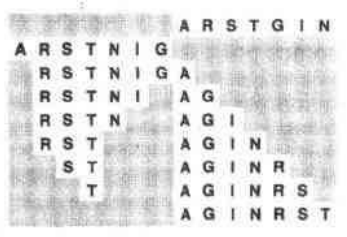



**图 8.1　不使用标志关键字进行归并**

对两个按七升顺序排列的文件，将它们复制到一个辅助的数组中，将第二个文件按倒序方式紧跟在第一个文件后面。接着，按以下简单的规则操作：将左边或右边键较小的项移到输出文件中。最大的关键字被当作了标志关键字，而不管它在哪个数组中。本图显示了文件 A R S T 和 G I N 归并的过程。

不过，通过第六章和第七章可以知道，有时是很难使用标志关键字的，因为可能很难确定标志关键字的值或者不够存储空间使用标志关键字。对归并操作，有一个简单的补救方法，如图 8.1 所示。该方法使用了以下基本思路：要复制数组来执行合适的归并排序，只需简单地将第二个数组变成倒序（不需要额外的消耗），因此相关的指针从右移向左。这样的安排使最大的元素成为了标志关键字——不管实际在哪个数组。程序 8.2 是基于该思路的抽象合适归并的有效执行程序；它会作为本章迟些会讨论的排序算法的基础。它仍会使用一个大小与归并输出文件大小成比例的额外数组，不过效率比直接执行程序的要高，因为避免了检测数组是否终止。

**程序 8.2　抽象的合适归并算法**

本程序没有使用标志关键字来进行归并，而是将第二个数组按倒序方式复制到辅助数组中，紧跟在第一个数组后面（使辅助数组变成 bitonic 顺序）。第一个 for 循环移动第一个数组，并使 i 指向 l，为开始归并做好准备。第二个 for 循环移动第二个数组，让 j 指向 r。接着，在归并操作中（第三个 for 循环），将最大的元素当作标志关键字，不管它在哪个数组。程序的内部循环都较小（移到辅助数组中，比较，移回到数组 a 中，增加 i 或 j 的值，增加 k 的值并检测）。

```
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++];
  }
```

一个关键字序列先是不断增大，然后不断减小（或者先是减少，然后增大），被称为bitonic序列。对bitonic序列进行排序等价于归并操作，有时很容易将归并问题转化为bitonic排序问题。不使用标志关键字的方法很简单。

归并算法的稳定性是程序8.1中一个重要的特性。它保持了重复关键字中的相对顺序，该特性很容易验证，而当执行抽象归并时，保持算法的稳定性是很重要的，因为稳定的归并立刻可以导出稳定的排序方法，我们会在第8.3节中看到这点。要保持稳定性并不容易，例如，程序8.2就不稳定（参看练习8.6）。对这点的考虑使构造一个真正合适的归并算法的问题更复杂。

**练习**

▷ 8.5 显示如何使用程序8.2对序列A E Q S U Y E I N O S T进行归并，使用图8.1风格的图表。

○ 8.6 解释为什么程序8.2是不稳定的，并编写一个稳定的版本。

8.7 写出使用程序8.2对序列E A S Y Q U E S T I O N进行操作的结果。

○ 8.8 是否当且仅当两个输入子数组已排好序，程序8.2所产生的输出才是正确的？证明你的答案，或者给出一个相反的例子。

## 8.3 自顶向下的归并排序

得出了归并算法后，就不难使用该算法作为递归排序过程的基础。要对某个文件进行排序，先将该文件平分成两部分，对两部分进行递归排序，接着将它们进行归并。程序8.3是其中一种执行方法，图8.2图示了一个例子。如第五章所说的，该算法是高效算法设计中分治法的一个著名示例。

自顶向下的归并排序算法具有自顶向下的管理风格，将一个大的任务分成几个可以独立解决的部分，将它们分配给各下属人员解决。如果只是简单地将任务平分成两部分，然后将下属得到的结果合并，将它传送给上级，该过程就与归并排序类似。没有太多实际工作要做，直到没有下属的人员获得了任务（本情况下，是对两个大小为1的文件进行归并）；不过，管理机构做了大部分的工作，将各结果合并。

归并排序很重要，因为它是最理想的直接排序方式（运行时间与$N \log N$成比例），执行过程可以具有稳定性。这些特性可以比较容易地证明。

如我们在第五章所说的（加上第七章所说的快速排序），可以使用树型结构来使递归算法的递归调用过程形象化，帮助我们理解算法的变量，加快对算法的分析操作。对于归并

**图8.2 自顶向下的归并排序示例**

图中每一行显示了自顶向下归并算法中一次调用merge操作的结果。首先，将A和S归并得到AS，接着归并O和R得到OR；接着，将OR和AS合并，得到AORS。随后，将IT和GN合并，得到GINT，然后将该结果与AORS合并，得到AGINORST，以此类推。该方法递归地将小记录文件合并成大文件。

排序，递归调用结构只与输入文件的大小有关。对任何给定的 N 值，我们可以声明一棵树，称为分治树，它描述了程序 8.3 执行过程中处理的子文件的大小（参看练习 5.73）：如果 $N$ 为 1，该树仅包含一个标志为 1 的节点；否则，该树中包括一个标志为文件大小 N 的根节点，一个$\lfloor N/2 \rfloor$树作为左子树，一个$\lceil N/2 \rceil$树作为右子树。每个树节点对应归并排序中一个递归调用。当 $N$ 是 2 的幂时，该构造方法会产生一个完全平衡的树，所有的节点都是 2 的幂，而外节点为 1。当 $N$ 不是 2 的幂数时，树的结构更复杂。图 8.3 中对两种情况都举了例子。在 5.2 节，当讨论与归并算法使用同一种递归调用过程的算法时，我们已碰见过这样结构的树。

**程序 8.3　自顶向下的归并排序**

基本归并排序方法是一个基本的分治法的递归程序，通过将数组 a[l],...,a[r]分成两个部分 a[l],...,a[m]和 a[m+1],...,a[r]来进行排序，对它们独立地进行排序（利用递归调用），然后将得到的排好序的文件合并，得到最终的排好序的文件。Merge 函数需要一个辅助数组，该数组必须够大以保存输入文件的副本，不过可以将该抽象操作看作是合适的归并算法（参看正文）。

```
template <class Item>
void mergesort(Item a[], int l, int r)
  { if (r <= l) return;
    int m = (r+l)/2;
    mergesort(a, l, m);
    mergesort(a, m+1, r);
    merge(a, l, m, r);
  }
```

上面所说的树的结构特性是直接与归并排序的分析结果相关的。例如，算法中使用的比较操作数目与节点标志的总量相等。

**性质 8.1**　*对任何包含 $N$ 个元素的文件进行排序，归并排序需要执行大约 $N \lg N$ 次比较操作。*

在 8.1 节和 8.2 节的执行程序中，将两个 $N/2$ 和 $N/2$ 文件进行归并时，需要执行 $N$ 次比较操作（该数量会偏差 1 到 2 次，由标志位的使用方法决定的）。因此整个排序中所使用的比较操作数目为：$M_N=M_{\lfloor N/2 \rfloor}+M_{\lceil N/2 \rceil}+N$，其中 $M_1=0$。该递归表达式还描述了节点标志的总数和包含 $N$ 个节点的树的外部路径长度（参看练习 5.73）。如果 $N$ 是 2 的幂时，上述结论很容易得到验证，并且可以归纳证明出普通的 $N$ 值情况。练习 8.12 到 8.14 描述了一个直接的证明过程。■

**性质 8.2**　*归并排序使用与 $N$ 成正比的额外内存空间。*

这个结论很容易根据 8.2 节的讨论得出。通过执行一些步骤，使算法变得复杂些，可以减少所需的额外空间数（参看练习 8.21）。在 8.7 节，我们会看到对链表结构的文件，归并排序同样有效，但需要额外的空间存放指针。对于数组，如 8.2 节和将在 8.4 节讨论的那

样，可以不需要额外的空间进行归并，但在效率上，并不值当实际应用该策略。■

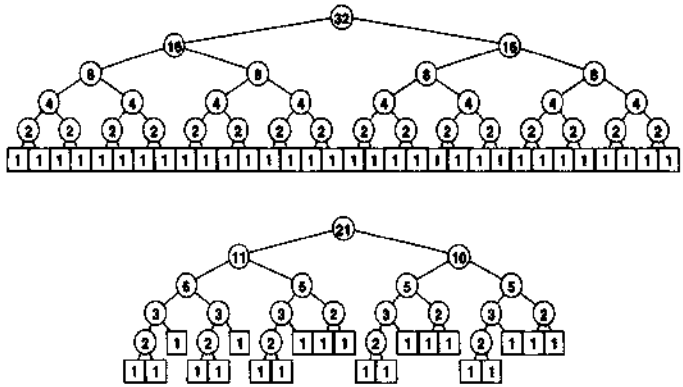


图 8.3 树结构

这些树的图描述了自顶向下归并排序产生的子问题的大小。这些树与快速排序对应的树不同，例如，树的模式只与原始文件的大小有关。上图显示了包含 32 个元素的文件的排序过程。对两个包含 16 个元素的文件进行递归排序，然后将它们进行归并。通过对 8 个元素的文件进行递归排序来实现对包含 16 个元素的文件进行排序，以此类推。如果文件的大小不是 2 的幂，树的结构更复杂，如底部图所示。

**性质 8.3** *如果所使用的归并算法是稳定的，那么归并排序算法是稳定的。*

经过归纳，该特性很容易得到证明。程序 8.1 中的归并执行过程显示了重复关键字的相对位置没有改变。然而越复杂的算法，越容易丧失稳定性（参看练习 8.6）。■

**性质 8.4** *归并排序算法所需的资源和输入文件的原始顺序无关。*

在执行过程中，输入文件只是决定了归并操作中元素的处理顺序。由于要将数据移动到辅助数组，每步中需要额外空间和数目与子文件大小成比例的一系列步骤。代码中两个 if 语句的分支过程的运行时间可能会有所差别，这点会使运行时间表达式中包含一个与输入文件有关的小变量，但对文件使用的比较操作及其他操作的数目与数据项的排列情况无关。要注意的是，这点和算法是非适应性的说法是不同的（参看第 6.1 节）——比较操作的序列与输入数据的顺序无关。■

**练习**

▷ 8.9 显示运用程序 8.3 对序列 E A S Y Q U E S T I O N 进行的归并操作。

8.10 对 $N$=16，24，31，32，33 和 39，画出相应的树结构。

8.11 对数组执行递归归并排序，使用三路归并，而不是二路归并。

○ 8.12 证明分治树中所有标志为 1 的节点都在最低两层中。

○ 8.13 证明大小为 $N$ 的分治树中每层节点标签大小的总和为 N，除了最后一层可能有所例外。

○ 8.14 使用练习 8.12 和 8.13，证明归并排序中所需执行的比较操作数目介于 $N \lg N$ 和 $N \lg N+N$ 之间。

8.15 找出并证明归并排序执行的比较操作数目和小于 $N$ 的$\lceil \lg N \rceil$位正数的总位数的关系。

## 8.4 对基本排序方法进行改进

如快速排序中那样，通过对小文件处理上进行一些修改可提高大部分递归算法的效率。由于递归算法经常在小文件中使用，因此对小文件的处理进行改进可以达到提高整个算法的效率。因此，和使用快速排序时一样，对小的子文件转换使用插入排序，可以使某类型的归并排序执行程序的运行效率提高 10%~15%。

对归并排序的第二个改进方法是节省将数组复制到辅助数组所花费的时间。为了达到这个目的，修改递归调用，将输入数组和辅助数组的作用调换。一种方法是执行两个版本的例程——一个将输入文件存放在数组 aux 中，输出文件存放在数组 a 中，另一个是将输入文件存放在数组 a 中，输出文件存放在数组 aux 中——然后让两个版本的例程互相调用。另一种方法在程序 8.4 中显示，在一开始时为数组做一个备份，接着使用程序 8.1，通过转换递归调用中的变量，消除外部的数组复制操作。事实上，我们是在将排好序的输出文件放到辅助数组和输入数组之间进行转换。

消除数组复制操作的代价是重新要在内部循环中执行测试操作，测试数组是否已到了末尾。（回忆一下如何通过复制操作构造 bitonic 数组，消除程序 8.2 中的测试操作。）通过运用相同思路的递归执行程序，可以重新补回该损失：在归并和归并排序中都执行两项操作，一项使数组正序排列，一项使数组倒序排列。使用这种方法，可以重新生成 bitonic 数组，因此不需要在内部循环中使用标志关键字。

由于该方法使用了四个基本例程的副本和一些相互转换的变量，它只被一些专家（或学生）所使用，不过它确实可以明显提高归并排序的效率。在 8.6 节讨论的实验结果表明，结合这些改进方法，归并排序的效率可以提高 40%，不过归并排序还是要比快速排序慢 25%。这个结论是和执行程序及运行的机器有关的，但在不同的情况下都会得到类似的结果。

**程序 8.4 不进行复制操作的归并排序**

归并程序中对 b 进行排序，将结果放在 a 中。因此，在调用时，要将结果放在 b 来进行修改，并且使用程序 8.1 将 b 中的文件归并到 a 中。这样，通过归并过程，就完成了所有移动元素的操作。

```
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+l, b+l, 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);
}
```

另一个归并执行方法，包括一个检测文件是否到结尾的操作，运行时间会根据输入文件有相对较大的差别，但差别也不是很大。对随机文件，当一个子文件已读取完毕时，其他子文件的大小也已很小，而移动到辅助数组的花费依然和子文件的大小成比例。当文件中已基本排好序时，可以这样对归并算法进行改进，如果文件已经完全排好序了，就跳过 merge 调用操作，但对大部分类型的文件来说，该操作并没太大效用。

**练习**

8.16 执行一个抽象就地归并程序，使用与两个要进行归并的数组中较小那个数组的大小成比例的额外内存空间（该方法为归并排序节省一半的空间）。

8.17 对大的随机文件进行归并该序，当某个子文件访问完毕后，将其他子文件未访问数据项的平均数表示为 $N$（归并中两个子文件大小的总和）的函数。

8.18 假设程序 8.3 中，当 a[m]<a[m+1]时，跳过 merge 调用。这样修改后，对已排好序的文件再次进行归并排序时，可以节省多少比较操作？

8.19 对大的随机文件运行练习 8.18 中提到的修改过的算法。将节省的 merge 操作的平均数表示成 $N$（排序文件的初始大小）的函数。

8.20 假设归并排序对 h-排序文件进行操作，$h$ 值较小。如何利用输入文件的特性，对 merge 过程进行改进？写出将 Shell 排序和归并排序结合的算法。

8.21 编写一个归并算法，使所使用的额外内存空间减少到 $\max(M, N/M)$，该算法使用了以下思路：将数组分成 $N/M$ 个大小为 $M$ 的块（为了简便，假设 $N$ 是 $M$ 的倍数）。接着使用选择排序对各块的记录进行排序，然后将数组的第一块归并到第二块中，接着将第二块归并到第三块中，以此类推。

8.22 证明练习 8.21 中的方法的运行时间是线性的.

8.23 执行没有复制操作的 bitonic 归并排序。

## 8.5 自底向上的归并排序

如我们在第 5 章所讨论的，每个递归程序都有一个等价的非递归程序与之相对应，而计算的顺序相反。非递归的归并排序很值得详细研究。

考虑递归算法中归并算法的执行序列。在图 8.2 的例子中，对大小为 15 的文件进行排序时，所执行的归并序列如下：

1-by-1 1-by-1 2-by-2 1-by-1 1-by-1 2-by-2 4-by-4
1-by-1 1-by-1 2-by-2 1-by-1 2-by-1 4-by-3 8-by-7

**程序 8.5　自底向上的归并排序**

自底向上归并排序对整个文件进行 m-m 的归并操作，每次 m 的值都变成两倍。只有当文件的大小是 m 的偶数倍数时，最后的子文件的大小为 m，因此最后一个归并算法是 m-x 归并，x 比 m 要小或者和 m 一样大。

```
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-l; m = m+m)
      for (int i = l; i <= r-m; i += m+m)
        merge(a, i, i+m-1, min(i+m+m-1, r));
  }
```

归并算法的执行序列是由算法的递归结构决定的。然而，子文件是独立处理的，归并操作可以以不同的序列进行。对同一个例子，图 8.4 显示了自底向上的方法，所使用的归并序列如下：

1-by-1　1-by-1　1-by-1　1-by-1　1-by-1　1-by-1　1-by-1
2-by-2　2-by-2　2-by-2　2-by-1　4-by-4　4-by-3　8-by-7

每种情况中，都包括了 7 个 1-1 归并、三个 2-2 归并，一个 2-1 归并、一个 4-4 归并、一个 4-3 归并及一个 8-7 归并，不过归并操作的执行顺序不同。自底向上算法将所剩余文件中最小的两项归并，从数组的左边向右边进行访问。

**图 8.4　自底向上归并排序示例**

每行显示了自底向上归并过程中调用 merge 的结果。首先进行 1-1 归并：A 和 S 归并成 A S；接着 O 和 R 归并成 O R；以此类推，因为文件的大小是奇数，最后的 E 没有包括在归并中。第二步中，进行 2-2 归并：A S 和 O R 进行归并得到 A O R S，以此类推。接着进行 4-4 归并、4-3 归并，最后是 8-7 归并。

递归算法中的归并操作序列是由图 8.3 中的树结构决定的：按前序顺序访问整棵树。如第 3 章所说的，非递归算法可以通过使用栈来得到相同的归并操作序列。不过并不需要严格要求使用前序序列：任何在访问某个节点前先访问它的子树的遍历方法都可以产生正确的算法。惟一的限制是，进行归并的文件必须是已排好序的。对于归并排序，先完成所有 1-1 归并，接着完成所有 2-2 归并，然后完成所有 4-4 归并，以此类推，按这样的顺序执行是很方便的。从递归树的底部开始访问的序列与按水平顺序访问相对应。

第五章中很多例子表明，当按自底向上思路思考时，最好是重新使用联合解决的方法考虑，即解决小的子问题，然后将它们结合得到大问题的答案。特别是在程序 8.5 中我们使用合治的非递归归并排序，方法如下：将文件中所有元素看作是一个大小为 1 的已排序子文件，遍历该文件列表，进行 1-1 归并，产生大小为 2 的顺序子文件；接着遍历新的文

件列表，进行 2-2 归并，产生大小为 4 的子文件；然后进行 4-4 的归并来产生大小为 8 的子文件；以此类推，直到整个文件排好序。最后的子文件的大小和其他的不同，除非文件的大小是 2 的幂数，但我们也可以将它归并进来。

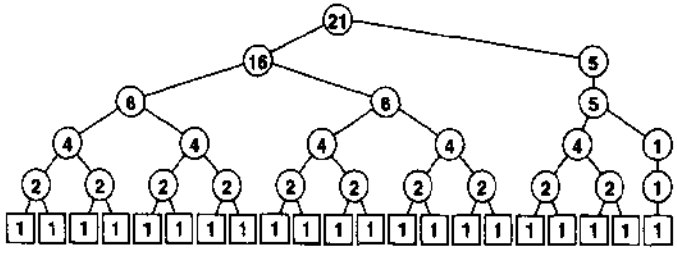



**图 8.5 自底向上归并排序文件的大小**

当文件大小不是 2 的幂数时，自底向上归并排序的结构和自顶向下归并排序的完全不同（参看图 8.3）。对自底向上归并排序，所有子文件的大小，除了最后一个，都是 2 的幂数。这些区别有利于理解算法的基本结构，但对实际应用没什么影响。

如果文件的大小是 2 的幂数，自底向上归并排序中执行的归并操作和递归归并操作中的一样，不过归并操作的执行顺序会有所不同。自底向上归并排序按水平的顺序遍历分治树，从树的底部已直到顶部。相反，递归算法称为自顶向下归并排序，因为前序遍历对树是进行从上到下的访问。

如果文件的大小不是 2 的幂数，从图 8.5 可以看出，自底向上算法所使用的归并算法有所不同。自底向上算法和合治树对应（参看练习 5.75），和自顶向下算法中的分治树有所差别。也可以使用它来产生递归算法的归并序列，但没有原因需要这样做，因为对比整个算法的消费，两种方法消费上的差异很小。

性质 8.1 至 8.4 同样适用于自底向上的归并排序，再加上以下一些特性：

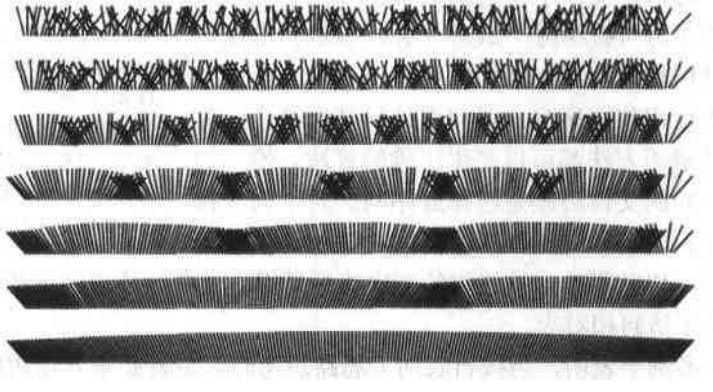


**图 8.6 自底向上归并排序**

对包含 200 个元素的文件进行自底向上归并排序，只需运行 7 步。每步都使已排好序的文件数目减少一半，而子文件的大小变成两倍（最后一个子文件例外）。

**性质 8.5**　自底向上归并排序中，每一步中进行归并排序的子文件的大小是 2 的幂数，最后一个子文件的大小是一个例外。

推过归纳很容易得到这个结论。■

**性质 8.6**　对包含 $N$ 个元素的文件进行自底向上归并排序时，所要执行的步数等于 $N$ 用二进制表示时的位数（忽略 0 位）。

自底向上归并排序中每一步都使已排好序的子文件的大小变成两倍，因此 $k$ 之后的子文件大小为 $2^k$。所以，对包含 $N$ 个元素的文件进行排序的步骤数是满足 $2^k \geq N$ 的最小 $k$ 值，也就是 $\lceil \lg N \rceil$，N 用二进制表示时的位数。我们还可以通过归并或对合治树的结构进行分析得到这个结论。■

图 8.6 中示例了对大文件进行自底向上合并操作的情况。我们可以通过 20 步对 1 百万的元素进行排序，通过 30 步对 10 亿数据进行排序，以此类推。

根本上，自底向上和自顶向下归并排序都基于将两个排好序的子文件合并成一个排好序的子文件的操作。两个算法非常相似，当文件大小为 2 的幂数时，使用相同的归并运算，不过两者是有所差别的。图 8.7 显示了两者对大文件进行操作时的动态差别。当内存空间不是很大，又不希望有最坏的情况产生时，两种方法都可以选择。两个算法都是分治和合治算法设计的重要范例。

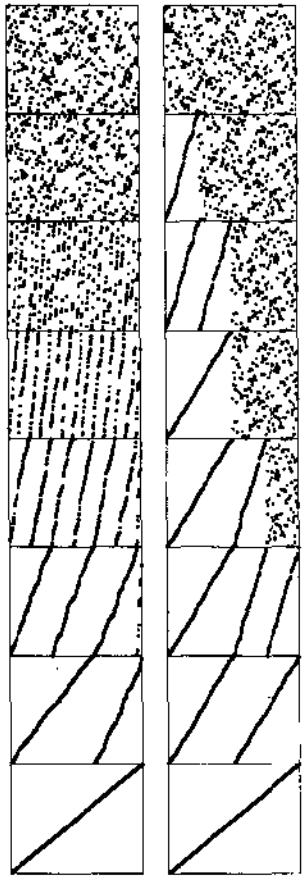


**图 8.7　自底向上和自顶向下归并算法的差别**

自底向上归并排序（左边）包括了一系列将排好序的子文件进行归并的操作，直到只剩下一个为止。文件中每个元素，除了在最后的一些外，每步操作都会访问到。与之相反，自顶向下归并排序（右边）首先对文件的第一部分进行排序，接着对另一半进行排序（递归地），所以它执行过程的结构完全不同。

## 练习

8.24　显示对关键字序列 EASYQUESTION 进行自底向上归并排序（程序 8.5）所执行的归并操作。

8.25　执行一个自底向上的归并排序程序，开始时使用插入排序对包含 $M$ 个数据项的块进行操作。对包含 $N$ 个元素的随机文件进行排序，当 $N=10^3$，$10^4$，$10^5$ 和 $10^6$ 时，判断 $M$ 值为多少时程序的运行效率最高。

8.26　画出与程序 8.5 归并过程对应的树结构，$N$=16，24，31，32，33 和 39。

8.27　编写一个归并算法，使其所使用的归并操作和自底向上归并排序算法所使用的一样。

8.28　编写一个自底向上的归并算法，使其使用的归并操作和自顶向下归并算法所使用的一样。（该练习比练习 8.27 要困难的多）。

8.29　假设文件大小是 2 的幂。消除掉自顶向下归并排序中的递归操作，得到一个使用相同归并操作序列的非递归归并排序程序。

8.30 证明自顶向下归并排序的执行步骤数也是 $N$ 用二进制表示时的位数（参看性质 8.6）。

## 8.6 执行典型的归并排序算法

表 8.1 显示了各种改进方式的相关效应。如通常情况一样，这些研究表明了如果对算法的内部循环进行一定的改进，算法的必要运行时间可以减少一半或以上。

除了在 8.2 节中提到的改进方法外，确保两个数组中最小的元素存放在简单的变量或寄存器中，也可以避免不必的数组存放操作。因此，归并排序的内部循环可以简化为一个比较语句（包含条件判断）、两个指针增加语句（k 及 i 或 j）和一个判断循环是否已完成的语句。内部循环中指令的总数比快速排序的略高，不过这些指令只需操作 $N\lg N$ 次，而快速排序运行的次数要多出 39%（或者 29%，进行了修改后）。要更精确地对特殊情况下算法的执行效果进行比较，需要更认真的执行过程和详细的分析操作；不过，我们知道归并排序的内部循环操作要比快速排序的稍长。

**表 8.1 归并排序算法的研究**

这里对随机的浮点数文件进行排序，使用了不同值的 $N$，表明标准的快速排序比标准的归并排序快两倍；对小文件增加的中断操作使自底向上和自顶向下归并算法的执行时间均减少了 15%；对这些大小的文件，自顶向下归并排序要比自底向上归并排序快 10%；但对随机文件，即使减少了归并算法中的文件复制操作，归并排序还是比简单的快速排序算法慢 50%~60%（参看表 7.1）。

| | | 自顶向下 | | | 自底向上 | |
|---|---|---|---|---|---|---|
| $N$ | Q | 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 |

其中：

Q 快速排序，标准（程序 7.1）

T 自顶向下归并排序，标准（程序 8.1）

T* 自顶向下归并排序，增加对小文件的中断程序

O 自顶向下归并排序，增加中断程序和不进行数组复制

B 自底向上归并排序，标准（程序 8.5）

B* 自底向上给并排序，增加对小文件的中断程序

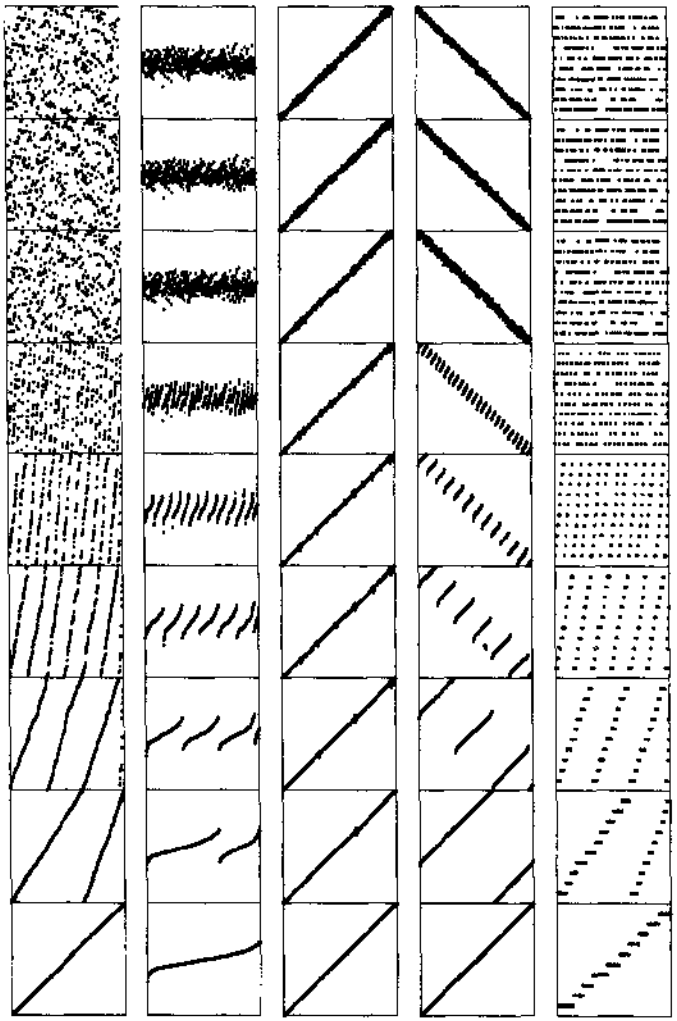


**图 8.8　使用自底向上归并排序对各种类型的文件进行操作**

归并排序的运行时间和输入文件无关。本图显示了对随机文件、高斯序列、基本排好序的文件、倒序的文件及随机按 10 个关键字排列的文件进行自底向上归并排序的操作步骤，显示了操作只与文件的大小有关，而与文件的输入值无关。这个特性与快速排序及其他很多算法都不同。

通常，虽然在很多程序中对算法进行的改进不可抗拒，但有时候这也会带来一些问题，所以执行前应更认真地进行考虑。这里，归并排序比快速排序优先之处是归并排序是稳定的算法，而且对任何输入文件执行的速度都很快，而它的不足之处是它需要使用和数组大小成比例的额外内存空间。当倾向使用归并排序（速度是很重要的）时，就需要考虑提到过的改进方法，还要认真考虑所使用的代码、机器结构的特性等等。

另一方面，需要认真注意执行过程，避免不必的消耗。所有的编程人员都有过因忽略了某个重要的特性而导致的尴尬。当对程序进行认真检查、消除这种毛病后，有时运

行效率可以提高两倍。不断进行检测操作是最有效的改进方法。

在第 5 章中，我们对这些观点进行了详细的讨论。不过对程序进行最优化非常重要，因此值得在每次讨论改进方法时都提出来。对归并排序，根据性质 8.1 到 8.4，算法的最优化是令人满意的，对我们所讨论过的任何情况都适合：运行时间和 $N \log N$ 成比例，与输入文件无关（参看图 8.8）；使用额外运行空间；执行过程是稳定的。在改进运行效率时，保持这些特性并不困难。

**练习**

8.31 不使用数组复制执行自底向上归并排序操作。

8.32 编写一个三层混合排序程序，使用快速排序、归并排序和插入排序算法，使它的运行效率和快速排序的最好效率相同（即使对小的文件），而在最坏的情况下，它的运行效率要较好。

## 8.7 使用链表执行归并排序

归并排序中需要额外的内存空间，所以我们可以考虑使用链表来执行。也就是说，不使用额外的辅助数组，而改用指针。首先我们需要解决如何对链表进行排序（参看第 6.9 节）。事实上，归并排序是很适合用于链表的。程序 8.6 给出了一个对链表进行归并操作的完整执行过程。注意所使用的归并操作代码和基于数组的归并操作代码一样简单（程序 8.2）。

**程序 8.6 链表归并操作**

该程序将由 a 指向的链表与由 b 指向的链表进行归并，并使用了一个辅助指针 c。merge 操作中进行的键比较操作包括了等于的情况，因此如果 b 链表跟在 a 链表后，该归并程序是稳定的。为了简单起见，我们约定所有的链表都以 0 结尾。另一种对链表结尾的约定同样适用（参看表 3.1）。更重要的是，没有使用头节点，以避免相对应的消耗。

```
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;
  }
```

根据这个归并算法，很容易得出一个自顶向下的递归链表归并排序程序。程序 8.7 是

一个直接的递归执行过程，输入指向未排序链表的指针，输出指向包含相同元素的已排好序的链表的指针。通过重新整理链表的节点来完成工作：不需要设置临时的节点或链表。程序 8.7 使用了一个巧妙的方法找出链表的中部：将链表的长度作为参数传递给递归程序，或者保存链表的长度。通过递归公式很容易了解这个问题，虽然它是较复杂的算法。

**程序 8.7　自顶向下的链表归并排序**

该程序将由 c 指向的链表平均分割成两部分，由 a 和 b 指向，对两部分进行递归排序，然后使用 merge 程序（程序 8.6）产生最后的结果。该输入链表结尾必须为 0（因此 b 链表也是这样），割分程序中 c->next=0 将 a 链表的结尾设为 0。

```
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));
  }
```

对链表归并排序，也可以使用自底向上的合治方法，虽然链表元素位置的确定要比它看起来要更复杂。如在 8.3 节对自底向上数组排序的方法进行的讨论那样，当进行自底向上的链表归并排序时，没有必要一定要将归并操作变成递归的或基于数组的。

有一个简单的自顶向上链表归并排序版本：将数据项放到循环的链表中，对链表进行处理，将已排好序的子文件进行归并，直到整个文件排好序。在概念上，这个方法很简单，不过（使用低效的链表相关处理程序）在实现上是很微妙的（练习 8.36）。另一个自底向上链表归并排序程序是基于与程序 8.8 一样的思路：将所有要归并的链表放到一个 ADT 中。在概念上，该方法也很简单，不过（使用 ADT 中的高效程序）执行过程也是很微妙的，其中一个重要特性是，该方法可以利用文件中已有的顺序。事实上，链表的执行过程数目不是$\lceil \lg N \rceil$，而是$\lceil \lg S \rceil$，$S$ 是原有数组中已排好序的子文件的数目。该方法有时称为自然归并排序。对随机文件，它没有什么显著的好处，因为只能节省一到两步（事实上，这个方法可能会比自上而下的方法要慢，因为需要额外的花费查出文件中已排好序的子文件），不过如果文件中包含几块已排好序的子文件，该方法是很有效的。

**程序 8.8　自底向上链表归并排序**

该程序使用队列 ADT 来执行一个自底向上的归并排序。队列元素是排好序的链表。首先将队列初始为长度 1 的列表，该程序只是简单地将两个链表从队列中移走，对它们进行归并，然后将结果放回到队列中，继续这样的操作，直到只剩下一个列表。该操作对应一系列对所有元素进行的操作，每步都将已排好序的文件的长度增加一倍，和前面提到的自底向上的归并算法一样。

```
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;
  }
```

**练习**

● 8.33 编写一个自顶向下的链表归并排序程序，将文件的长度作为递归过程的参数，用它来决定划分链表的方式。

● 8.34 编写一个自顶向下的链表归并排序程序，将文件长度保存在头节点中，用它来决定划分列表的方式。

8.35 在程序 8.7 中增加一个小文件中断程序。判断中断值的正确选择来提高程序的效率。

○ 8.36 执行一个自底向上的归并排序程序，使用正文中描述的循环链表。

8.37 在练习 8.36 的自底向上循环链表归并排序程序中加入一个小文件中断程序。判断如何决定进行中断的标志来提高程序的效率。

8.38 为程序 8.8 增加一个小文件中断程序。判断如何决定进行中断的标志来提高程序的效率。

○ 8.39 画出与程序 8.8 执行过程对应的合治树，$N$=16，24，31，32，33 和 39。

8.40 画出与循环链表归并排序（练习 8.38）执行过程对应的树结构，$N$=16，24，31，32，33 和 39。

8.41 对包含 $N$ 个随机的 32 位整数文件中已排好序的子文件数目进行研究。

● 8.42 对随机的包含 64 位键的文件进行自然归并排序，当 $N=10^3$，$10^4$，$10^5$ 和 $10^6$ 时，判断所要运行的步骤数目。提示：完成该练习不需要执行排序算法（甚至不需要完整的生成 64 位键）。

● 8.43 初始化时将输入文件中已排序的子文件组成序列，接着将程序 8.8 转换成自然归并排序。

○ 8.44 执行一个基于数组的自然归并排序。

## 8.8 再次讨论递归过程

本章中的程序及上一章所讨论的快速排序，是分治算法的典型执行程序。在以后的章节中，我们还会碰到几个有类似结构的算法，因此对这些执行过程的基本特性进行更详细的研究很有价值。

将快速排序称为分治算法可能更准确：在递归执行程序中，大部分的工作都是在递归调用之前完成的。与之相反，递归归并算法更遵循分而治之的思路：首先，将文件分割成两

个部分；接着，对每个部分分别进行处理。归并排序首先处理的是小子文件；最后处理最大的子文件。快速排序首先对最大的子文件进行处理，最后处理小文件。按照本章一开始所使用的方式，对两个算法进行管理分析是很有趣的：快速排序中各个管理者集中讨论如何将问题进行细分，当完成了子任务时也就完成了整项工作；而归并排序中每个管理者快速地将任务平分成两部分，因此完成了子任务后，需要将结果进行合并处理。

在非递归的执行过程中，这两种方法的差别是很明显的。快速排序必须使用栈，因为它保存大的子问题，然后根据数据的情况将它进行拆分。归并排序的非递归程序仍然简单，因为它拆分文件的操作与数据无关，所以，可以通过重新整理对子问题的处理顺序来产生更简单的程序。

对于快速排序，更自然的是将它看作一个自顶向下的算法，因为它先是对递归树的顶端进行操作，接着向下处理直到完成整个排序。我们可以构造一个非递归的快速排序程序，对递归树按水平顺序从上到下进行处理。因此，排序算法中包含多重的步骤，将文件拆分成小的子文件。对于数组，该方法并不实用，因为需要一定的消耗来跟踪子文件；但对链表，它类似于自顶向上的归并排序。

我们已经知道在稳定性上，归并排序和快速排序不同。对于归并排序，如果假设子文件已经稳定地排好序了，只要确保所使用的归并操作是稳定的，排序就是稳定的，这点很容易做到。根据算法的递归结构立刻可以归纳证明该算法的稳定性。而对基于数组的快速排序执行过程，没有简单的方法可以稳定地将数组进行划分，因此在执行递归程序前，稳定性已被破坏。然而对链表进行的直接快速排序程序是稳定的（参看练习 7.4）。

如我们在第五章所见的，包含一个递归调用的算法通常简化成一个循环操作，不过包含两个递归调用的算法，如归并排序和快速排序，使用分治的算法和树型结构，很多最好的算法都采取这样的方式。归并排序和快速排序需要认真的研究，不仅因为它们重要的实用价值，还因为它们本身具有的递归特性，使我们可以更好地研究和理解其他递归算法。

### 练习

● 8.45　假设归并排序将文件按随机方式进行拆分，而不是恰好进行平分。使用这样的方法对包含 $N$ 个元素的文件进行排序，平均需要使用多少次比较操作？

● 8.46　研究使用归并排序算法对字符串进行操作的情况。对大文件进行排序时，平均需要进行多少次比较操作？

● 8.47　比较对链表进行快速排序（参看练习 7.4）和对链表进行自顶向下归并排序（程序 8.7）的情况。

# 第九章

# 优先队列与堆排序

许多应用程序要求我们根据关键字处理记录，但不需要任何时候都对记录完全排序。通常，我们管理一个集合中的记录，然后处理关键字最大的一个，接着可能加入更多的记录，然后处理当前的关键字最大的记录……在这样的一个环境下，有一个合适的数据结构支持插入新记录和删除最后到的记录。这个数据结构就是“优先队列”（priority queue）。使用优先队列类似于使用队列（先进先出）和栈（后进先出），但要高效率地实现它们是很有挑战性的。优先队列是我们在 4.6 节讨论过的队列 ADT 的一个最重要的推广。实际上，优先队列是适当地把队列与栈一般化，因为我们能够通过恰当的优先级设置来用优先队列实现队列与栈的数据结构（详细见练习 9.3 和 9.4）。

**定义 9.1** *优先队列是一个数据结构，队列中的记录是有关键字的，它支持两个基本操作：插入新记录和删除关键字最大的记录。*

使用优先队列的应用程序包括：模拟系统，系统中记录的关键字对应于事件的时间，需要处理的记录按照年代顺序排列；计算机系统中的工作时间表，记录的关键字对应于用户的优先级；数值计算，记录的关键字对应于计算错误，最大的错误应该优先处理。

我们可以使用任何优先队列作为排序算法的基础。先插入所有的元素，然后不断地把最大的元素移出就可以得到从大到小排列的元素序列了。在本书的后续部分，我们将看到如何使用优先队列作为一个更高级算法的构件。在第五部分，我们将会发展一个文件压缩算法，其中使用了本章的程序；而在第七部分，我们将会看到优先队列如何成为一个适当的抽象概念来帮助我们理解几个基本“图搜索”算法之间的关系。这些，仅仅是优先队列作为基本工具在算法设计中担当重要角色的几个例子而已。

实际上，优先队列比上文给出的定义要复杂得多，因为要增加其他的一些操作来满足我们在使用中出现的需求。当然，许多优先队列的构件是相当实用的一个主要原因是——它们对客户程序的适应性，这些客户程序通常需要对带关键字的记录集合有不同的操作。我们希望建立和实现一个数据结构，包括含有数值关键字（优先级）的记录，而且支持以下的操作：

- 由 $N$ 个给定的记录构建一个优先队列；
- 插入一个新记录；

- 移出最大的记录；
- 改变任意一个记录的优先级；
- 移出任意的一个记录；
- 联结两个优先队列。

如果允许记录有相同的关键字，我们把“最大”理解为“所有包含最大键值的记录”。就像许多数据结构一样，我们也需要加入标准操作，如“创建一个新队列”、“该对列是否为空”，和可能的“析构整个队列”、“复制队列”操作。

这些操作之间有部分操作是相互叠加的，有时候定义其他的相似操作也是很方便的。例如，一些客户可能需要频繁地在优先队列中检索最大记录，但不需要删除它。又或者，我们可以有一个用新记录替代最大记录的操作。我们能够用两个基本操作作为构件实现以上这些操作：“搜索最大记录”可以是“删除最大操作”加上“插入”操作，而“替代最大记录”操作可以是“插入”操作后“删除最大记录”或者“删除最大记录”接着“插入”操作。我们通常会有更高效率的代码，但除非它们是必要的，而且精确地说明了，我们才会直接实现这些操作。“精确的说明”不是永远都那么简单明了。例如，“替代最大记录”的两个实现方法实际上是很不同的：前一个方法永远都会使优先队列暂时增加一个记录，而后一个方法永远只是往一个刚刚少了一个记录的队列里加入一个记录。类似地，“改变记录优先级”操作能够实现为“删除”操作然后“插入”操作，而“创建”操作可以实现为不断的“插入”操作。

**程序 9.1　基本的优先队列 ADT**

这个接口定义了最简单的优先队列的操作：初始化、队列是否为空、加入新记录、删除最大记录。这些功能的初步实现采用数组和链表，在最坏情况下会需要线性时间。但我们将看到在本章里，所有功能的实现都会被认为是与队列内记录个数的对数成比例的时间复杂度。如通常一样，传给构造函数的变量指定队列中可能出现的最大记录，但这个变量对一些操作不是必需的。

```
template <class Item>
class PQ
  {
    private:
      // 随实现的变动而变动的代码
    public:
      PQ(int);
      int empty() const;
      void insert(Item);
      Item getmax();
  };
```

对一些应用程序而言，围绕“最小值”而不是“最大值”来进行操作可能会方便一些。我们集中研究面向最大值的优先队列。当我们需要讲到其他类的优先队列时，我们会专门

讲明它是一个面向最小值的优先队列（一个允许我们删除最小记录的优先队列）。

优先队列是一个原型式的抽象数据类型（ADT）（参见第四章）：它提出了一套设计良好的数据操作，也提供了一个简便的抽象，从而允许我们把应用程序（客户）和本章中不同的实现方式分离开来。程序 9.1 给出的接口定义了最基本的优先队列操作；我们将在 9.5 节考虑一个更完整的接口。严格地讲，我们想包括的各种操作的不同子集导致不同的抽象数据类型，但优先队列的本质特性是由“删除最大记录”操作和“插入”操作确定的，所以我们应该把精力集中在这两个操作上面。

优先队列的不同实现方式提供各种操作的不同性能特性，而不同的应用程序要求对不同的操作集合有高效率的表现。当然，性能的差别是，原则上在抽象数据类型概念中能够出现的惟一的差别。这种情形导致成本权衡。在本章里，我们将会考虑一系列的方法来逼近这些成本权衡，几乎达到理想中的在对数时间 $O(\log N)$里执行“删除最大记录”操作、在常数时间 $O(1)$里执行所有的其他操作。

首先，我们在 9.1 节中通过讨论一些基本的优先队列数据结构来举例说明这一点。然后，在 9.2 节到 9.4 节，我们集中阐述一个经典的数据结构——堆，它允许除“联结”操作外的所有操作的高效实现。在 9.4 节，我们也会看到从这些实现自然而来的一个重要的排序算法。接着，在 9.5 节和 9.6 节中，我们会看到某些在开发完整优先队列抽象数据类型时遇到问题的细节。最后，在 9.7 节，我们会观察一个更加高级的数据结构，叫做“二项式队列”，我们用来在最坏运行时间里实现所有操作（包括“联结”操作）。

在我们学习所有这些不同的数据结构的时候，我们应该记住由链表和连续内存储存方式（在第 3 章中介绍过）决定的权衡以涉及编写应用程序需要的包的问题。特别是，本书中迟些出现的某些高级算法是使用优先队列的客户程序。

**练习**

▷ 9.1 在序列中字母代表“插入”操作，星号代表“删除最大记录”操作

PRIO*R**I*T*Y***QUE***U*E

给出“删除最大记录”所返回的序列。

▷ 9.2 在练习 9.1 中的约定中加入一个“+”号表示“联结”，圆括号划定优先队列的界限。给出下列处理之后优先队列的内容。

（（（PRIO*）+（R*IT*Y*））***）+（QUE***U*E）

○ 9.3 解释如何用一个优先队列抽象数据类型实现一个栈抽象数据类型。

○ 9.4 解释如何用一个优先队列抽象数据类型实现一个队列抽象数据类型。

## 9.1 基本的实现方法

我们在第三章讨论的基本数据结构为我们提供了许多实现优先队列的选择。程序 9.2 是一个使用无序数组作为基础数据结构的实现。“搜索最大记录”操作的实现通过扫描数组

找到最大记录，然后把最大记录和最后一个记录互换，并且减小队列的长度。图 9.1 显示数组在一系列操作中的内容。这个基本实现对应于我们在第四章中看到的一些相似的栈和队列的实现（详见 4.7 节和 4.15 节），而且对小队列很有用。重要的区别在于性能。对于栈和队列来说，我们能够开发出对所有操作都是常数时间内完成的实现；对优先队列来说，很容易找到“插入”操作或者“删除最大记录”操作在常数时间完成的实现方法。但找到一个令得“插入”操作和“删除最大记录”操作都能在常数时间完成的实现方法会提高高负荷任务的速度，而这也是本章的主题。

**程序 9.2 一个优先队列的数组实现**

这个实现可以与我们在第四章谈过的栈和队列的数组实现相比较（详见 4.7 节和 4.15 节），让记录保存在一个无序数组里。从数组的末尾加入和删除记录，就如栈一样。

```
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];
        }
  };
```

我们能够使用由链表或数组实现的无序或有序序列。让记录无序和让它们保持有序的基本折中是：维持一个序列有序能够实现“删除最大记录”和“查找最大记录”操作在常数时间完成，但可能意味着“插入”操作需要扫描整个链表；然而一个无序序列能够实现常数时间的“查找”操作，但需要扫描整个序列来完成“删除最大记录”操作。无序序列是对这个问题的“懒惰”逼近原型，因为我们不到非得做时就拖延（去找最大记录）；有序序列是这个问题的“热心”逼近原型，因为我们提前做尽量多的事情（去保持列表在插入操作时有序）

从而让后继的操作高效率。我们能够在任意情况下使用数组或链表，折中方法是使用双向链表实现常数时间的删除操作（和无序情况下的“联结”操作），但需要为链接付出更多的空间。

**表 9.1 最坏情况下优先队列操作的运行时间**

优先队列抽象数据类型的不同实现在性能上有很大的不同，就如不同的方法在这张最坏情况运行时间表里所显示的（在一个很大的常数 N 范围内）。基本的方法（头四行）的一些操作需要常数时间而其他的操作需要线性时间；更高级的方法确保在大多数或所有操作中都只需要对数时间或常数时间的性能。

| | 插入 | 删除最大记录 | 删除 | 查找最大记录 | 修改优先级 | 联结 |
|---|---|---|---|---|---|---|
| 有序数组 | $N$ | 1 | $N$ | 1 | $N$ | $N$ |
| 有序链表 | $N$ | 1 | 1 | 1 | $N$ | $N$ |
| 无序数组 | 1 | $N$ | 1 | $N$ | 1 | $N$ |
| 无序链表 | 1 | $N$ | 1 | $N$ | 1 | 1 |
| 堆 | $\lg N$ | $\lg N$ | $\lg N$ | 1 | $\lg N$ | $N$ |
| 二项式队列 | $\lg N$ | $\lg N$ | $\lg N$ | $\lg N$ | $\lg N$ | $\lg N$ |
| 理论最好情况 | 1 | $\lg N$ | $\lg N$ | 1 | 1 | 1 |

在一个长度为 $N$ 的优先队列数组里，不同实现方式的不同操作（在一个常量范围内）在最坏情况下的运行时间概括如表 9.1。

开发一个完整的实现需要小心注意接口——特别是客户程序在执行“删除”和“修改优先级”操作时如何访问节点，以及作为一个数据类型它们自己在执行“联结”操作时如何访问优先队列。这些话题都会在 9.4 节和 9.7 节中讨论，那里会给出两个完整的实现：一个使用双向无序链表，另一个使用二项式队列。

使用优先队列的客户程序的运行时间不但由队列中记录的关键字决定，而且由不同操作的组合决定。记住那些简单的实现方式是明智之举，因为它们通常能够在许多特定情况下比复杂方法做得要好。例如，无序链表实现可以很适合一个“删除最大记录”操作很少而“插入”操作很多的应用程序；一个有序链表可能适合一个需要做大量“查找最大记录”操作，或者加入到序列中的记录要比优先队列中原有的记录还要多的情况。

**图 9.1 优先队列的例子（用无序数组表示）**

该序列是对左列（从上到下）中的操作的结果序列，其中字母表示插入操作，星号表示删除最大值的操作。每行表示一个操作，删除最大值操作所删除的字母，以及操作之后的数组内容。

## 练习

▷ 9.5 评价下列想法：要实现常数时间的“查找最大记录”操作，为

什么不一直记录最大元素插入的位置，然后直接返回最大元素的值？

▷ 9.6 给出在执行完图 9.1 中操作序列后数组的内容。

9.7 提供一个基本优先队列接口的实现，使用一个有序数组作为底层数据结构。

9.8 提供一个基本优先队列接口的实现，使用无序链表作为底层数据结构。提示：参考程序 4.8 和 4.14。

9.9 提供一个基本优先队列接口的实现，使用一个有序链表作为底层数据结构。提示：参考程序 3.11。

○ 9.10 考虑一个懒惰实现，链表只有在执行“删除最大元素”和“查找最大值”操作时有序。插入操作从上一次排序开始就被保存在另外一个链表里，然后在需要的时候排序与联结。讨论这样一个实现方式在基于无序和有序链表实现的优点。

● 9.11 编写一个性能驱动客户程序，用“insert”操作填满优先队列，接着用“getmax”操作删除半数的记录，然后用“insert”操作再次填满队列，用“getmax”操作删除所有记录，对长短不一的随机数序列如此做数次；测量每次运行的时间；然后输出平均运行时间。

● 9.12 编写一个性能驱动客户程序，用“insert”操作填满优先队列，然后在一秒钟内执行尽量多的“insert”和“getmax”操作，对长短不一的随机数序列如此做数次；输出平均能够做“getmax”操作的次数。

9.13 用你在 9.12 完成的客户程序比较程序 9.2 中的无序数组实现和在练习 9.8 中得到的无序链表实现。

9.14 用你在 9.12 完成的客户程序比较 你在练习 9.7 和 9.9 中得到的有序数组实现和有序链表实现。

● 9.15 编写一个练习驱动客户程序，在实际应用程序可能遇到的困难和病态情况下使用我们程序 9.1 中优先队列接口的函数。简单的例子包括关键字已经有序、逆序、所有关键字相同和序列里的关键字只有两个值。

9.16 （本练习看上去是 24 个练习）证明表 9.1 中的四种基本实现的最坏运行时间的边界，参考程序 9.2 和练习 9.7 到 9.9 中的“插入”和“删除最大记录”操作；非正式地描述其他几个操作的方法。对于“删除”、“改变优先级”和“联结”操作，假设你有办法直接知道记录的位置。

## 9.2 堆的数据结构

本章的中心话题是一个简单的叫做“堆”的数据结构，它能够高效地支持基本的优先队列操作。在一个堆里，记录是被存储在一个数组里，使每个关键字都比另外两个指定位置上的关键字的值要大。以此类推，那些指定位置上的关键字又各自比另外两个指定位置上的关键字的值要大。如果我们把这些关键字放在一个二叉树的结构里，每个关键字都有两条边指向比它小的关键字，堆的排列就容易看清楚了。

**定义 9.2** *如果一棵树中每个节点中的关键字都大于或等于所有子节点中的关键字（如果有子节点的话），就称它是按堆排序的。同样，每个堆排序树节点中的关键字都小于或等于父亲节点中的关键字（如果有父节点的话）。*

**性质 9.1** *没有任何一个堆排序树的节点关键字大于根节点的关键字。*

我们能够把堆排序的限制应用于任何树。它在使用“完全二叉树”时特别方便。记得第三章我们能够先画好根节点，然后继续把节点从左到右放置，不断地把上一层节点下方

的两个节点与父节点联结起来，直到所有的 $N$ 个节点都已经放好位置。我们能够把完全二叉树放在数组里面表示出来，根节点放在位置 1，它的孩子放在位置 2 和 3，然后下一层的节点放在位置 4、5、6、7，再下一层的节点放在 8、9、10……就好像图 9.2 所示。

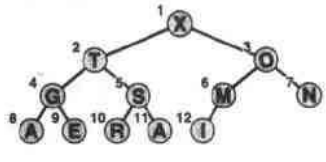



**图 9.2 用数组表示一个堆顺序的完全二叉树**

可以看到在数组$\lfloor i/2 \rfloor$位置的元素是在 $i$ 位置的元素的父亲，$2 \le i \le N$（同样，第 $i$ 个元素是第 $2i$ 个元素和第$(2i+1)$个元素的父亲）。很方便地对应表示出树状分布的元素。这里的对应关系就等同于为一棵完全二叉树的节点从左到右按层编号。一棵树是堆顺序的，如果任意节点的关键字都是大于或等于它的子节点的关键字的话。一个堆是一个数组表示的堆顺序完全二叉树。第 $i$ 个元素在堆中大于或等于第 $2i$ 个元素和第$(2i+1)$个元素。

**定义 9.3** 堆是一个节点集合，其中的关键字排放在一个堆排序的完全二叉树中，表示为一个数组。

我们能够用一个链表式的表示方法来表示堆顺序的树，但完全二叉树给了我们用一个压缩数组表示堆的机会，我们能够很容易由一个节点到它的父亲节点和子节点而不需要额外的链接。在位置 $i$ 的节点的父节点的位置是$\lfloor i/2 \rfloor$，与之相反，两个孩子节点的位置是 $2i$ 和 $2i+1$。这种安排使树的遍历比在对以链接表示时更加容易，因为在一个链接表示中，每一个节点都有三个链接来上下遍历一棵树（每个元素有一个指针指向父节点、一个指针指向一个子接点）。完全二叉树用数组表示是刚性的结构，但它们有足够的柔韧性来支持我们实现高效的优先队列算法。

我们将会在9.3节看到我们能够用堆来实现所有优先队列操作（除了“联结”操作之外），而且这些操作在最坏情况下只需要对数时间。所有实现的操作都是顺着堆中的路径进行（向着堆底把父节点移向字节点或者向着堆顶把子节点移向父节点，但堆中上下的方向不变）。正如我们在第三章中讨论的，在一棵有 $N$ 个节点的完全二叉树中，所有的路径有大约 $\lg N$ 个节点：大约有 $N/2$ 个节点在底部，$N/4$ 个节点有位于底部的子节点，$N/8$ 个节点有孙节点在底部，以此类推。每一代节点都约是下一代节点数量的一半，最多有 $\lg N$ 代节点。

我们也能够用额外的链接来表示树结构，从而开发出高效的优先队列的实现。例子包括三指针的堆顺序完全树（详见 9.5 节）、联赛（详见 5.19 节）和二项式队列（详见 9.7 节）。对于简单的栈和队列，考虑链接方式实现的一个重要理由是它们能够把我们从不得不提前知道堆列的长度中释放出来，而这在数组表示中是必不可少的。我们也能够利用链接结构提供的柔韧性来发展特定情况下的高效算法。在处理在本章练习和 9.7 节中讨论过的树的链接表示之前，我们鼓励对树的链接表示缺乏经验的读者们去读一读第十二章，学习那些对于更重要的符号表抽象数据类型来说是基本方法的知识。可是，对链接结构的仔细考虑可以放到第二遍阅读时做，因为我们这一章的首要话题是堆（堆顺序完全树的非链接数组表示）。

## 练习

▷ 9.17 一个按递减排好序的数组是不是一个堆？

○ 9.18 堆中的最大元素必须位于第一位，次大的元素必须排在第 2 或第 3 位。在一个大小为 15 的堆

中给出位置列表，当第 $k$ 大的元素

（1）可以出现

（2）不可以出现

$k=2$，3，4

（假设所有的元素的值都是不同的）

- 9.19　对任意的 $k$，堆的大小为 $N$，回答练习 9.18。
- 9.20　对第 $k$ 小的元素，回答练习 9.18 和 9.19。

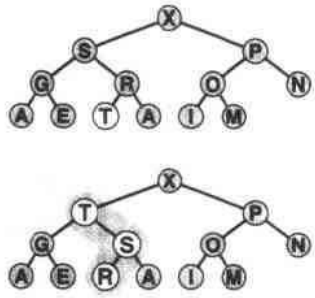



图 9.3　自底向上堆化

上方的树是堆顺序的，除了底部的节点 T 之外。如果我们把节点 T 和它的父节点交换，整棵树就是堆顺序的了，除了节点 T 可能比它的新的父节点大。继续把 T 节点和父节点互换，直到我们遇到根节点或是从节点 T 到根节点的路径上有一个节点大于 T 节点，这样我们就能够确定整棵树都是符合堆的条件的。我们能够用这个过程作为堆的“插入”操作的基础，去重新确定堆中加入一个新节点后（加入到堆的最右下方，如果需要就新建一层）仍然符合堆的条件。

## 9.3　基于堆的算法

所有基于堆的优先队列算法都会首先简单地破坏堆的条件，然后遍历堆，按需要重新构造堆，以确保完全符合堆的条件。这个过程有时候叫做“堆化”，或者仅仅是修复堆。这里有两种情况。当一些节点的优先级提高了（或者一个新的节点被加入到堆的底部），我们不得不向上遍历堆来重构堆。当一些节点的优先级下降了（例如，如果我们用一个新节点替换了根节点），我们也不得不向下遍历堆来重构堆。首先，我们考虑如何实现这两个基本功能；然后，我们看看如何利用它们来实现不同的优先队列操作。

如果堆的属性是因为一个节点的关键字变得比父节点的大，那么我们就要把该节点和它的父节点互换来修复被破坏的部分。在互换之后，该节点大于它的两个孩子（一个是原来的父节点，另一个原来就小于旧的父节点），但可能仍然大于它现在的父节点。我们能够用同样的方法来修复，把节点不断地往堆顶移动，直到遇到一个更大的父节点或是到达堆顶。这个处理过程的一个例子会出现在 9.3 节。那些代码都是直截了当的，基于堆中位置 k 上的节点的父节点的位置是 k/2 的想法。程序 9.3 是一个修复功能的实现，该功能通过把节点在堆中往上移，修复了一个可能因为提高某个堆中节点的优先级而导致的堆被破坏。

**程序 9.3　自底向上堆化**

当一个节点的优先级提高了，为了重构堆，我们从堆的下方不断往上移动，交换节点 k 和它的父节点（在位置 k/2）。如果需要，继续交换直到 a[k/2]<a[k]或者直到我们到达堆的顶部。

```
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; }
  }
```

如果堆的属性是因为一个节点的关键字变得至少小于其中一个子节点的关键字而被破坏，那么我们能够通过把该节点与它的两个子节点中较大的一个互换来修复被破坏的堆。这样的改变可能会在孩子节点中造成堆属性的破坏；我们用同样的方法修复，把节点不断下移，直到遇到节点比它的两个子节点都大，或者已经到达堆的底部为止。这样一个过程的例子在图 9.4 可以看到。代码又一次直接遵循"在位置 k 的节点的孩子在位置 2k 和 2k+1"的事实。程序 9.4 是一个功能的实现，该功能通过不断把节点往堆的底部方向移动，修复了由于指定节点的优先级提高而可能导致的堆破坏。为了能够检测是否到达堆的底部，这个功能需要知道堆的大小（N）。

**程序 9.4 自顶向下堆化**

当一个节点的优先级下降后，为了重构堆，我们从堆的上部向下移动，如果必要的话，把节点 k 和它的较大子节点互换，直到在位置 k 的节点的子节点都小于 k 节点或者 k 节点已经在堆的底部。注意，如果 N 是偶数而 k=N/2，那么在位置 k 的节点只有一个子节点——这种情况必须适当处理。

在这个程序里的内部循环有两个特殊出口：一个是到达了堆的底部，另一个是在堆的内部已经符合堆的条件了。这是一个"break"结构的原型例子。

```
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;
      }
  }
```

这两个操作是独立于树结构的表示的，只要我们能够访问任何节点的父节点（在自底向上方法中）和子节点（在自顶向下方法中）就行了。在自底向上方法中，我们从树的下方不断往上移动，交换指定节点和它的父节点的关键字，直到我们到达根节点或者有一个父节点有更大（或相等）的关键字。在自顶向下方法中，我们从树的上部向下移动，交换指定节点和它较大子节点的关键字，直到我们到达树的底部或者没有子节点有更大的关键字。以此方式推广，这些操作不但可以应用于二叉树，而且可以应用于任何树结构。高级的优先队列算法通常使用更多的树结构，但依赖这些相同的基本操作来维持对堆顶最大元素的访问。

如果我们想像用堆来表示一个社团的结构，每一个子节点表示一个下属（而每一个父节点表示直接上司），然后这些操作就有了有趣的解释。自底向上方法对应于一个刚提升的新经理走马上任，发号施令的链也要改变了（通过与低级的老板交换工作），直到新人遇到一个更高级的老板。自顶向下方法的情况是类似的，当公司的总经理被一个较低级的人取代。

如果总经理的最有力下属比新人强，他们交换工作，而我们从支配链向下，把新人降级同时提升其他人，直到新人到达相应能力的水平，再没有更高资历的下属了（这种理想情节在现实生活中的确存在）。利用这种类似，我们常常把一个堆中的上移称为“晋升”（promotion）。

这两个基本操作允许基本优先队列抽象数据类型的高效实现，就如程序 9.5 给出的那样。把优先队列表示成一个堆顺序的数组，“插入”操作相当于在数组末尾加入新元素，然后把该元素由堆的底部往上移，从而重构堆；“删除最大元素”操作相当于把最大元素从堆顶移走，然后把最后的元素移到堆顶，再把它从堆顶往下移，从而重构堆。

**程序 9.5　基于堆的优先队列**

为了实现 insert 操作，我们把 N 增加 1，把最新的元素加入到堆的末尾，然后用 fixUp 操作来重构堆。对于 getmax 操作，堆的大小要减 1，然后我们得到 pq[1]的值，再把 pq[N]移到 pq[1]来减少堆的大小，接着用 fixDown 操作重构堆。构造函数的实现和 empty 操作是不值一提的。数组的第一个位置 pq[0]是不用的，但可以作为一些实现的标志。

```
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--];
        }
  };
```

**性质 9.2**　优先队列抽象数据类型的“插入”和“删除最大元素”操作能够用堆顺序树实现，当处理一个有 $N$ 个元素的队列时，“插入”操作需要少于 $\lg N$ 次比较，而“删除最大元素”操作需要少于 $2\lg N$ 次比较。

这两个操作包括遍历根节点到堆底的路径，但没有一条大小为 $N$ 的堆中的路径包含多于 $\lg N$ 个元素（参看性质 5.8 和练习 5.77）。“删除最大元素”操作对每个节点需要 2 次比

较：一次是找到关键字更大的子节点，另一次是决定子节点是否需要被晋升。■

图 9.5 和 9.6 给出了一个例子，我们通过往一个空堆一个接一个插入元素构造了一个堆。在我们一直使用的数组表示中，这个处理对应于通过在数组中移动元素用堆对数组排序，注意，每次我们移动到一个新元素，堆的大小就增加 1，然后要重构堆。整个堆排序的过程最坏情况下耗时 $N$ 到 $\lg N$（如果每个新元素都是当前最大的元素，它要遍历到堆顶的整个路径），但这个过程在平均情况下只需要线性时间（一个随机的新元素一般只需要遍历几层而已）。在 9.4 节中我们将会看到一个在最坏情况下用线性时间构造堆的方法（对一个数组进行堆排序）。

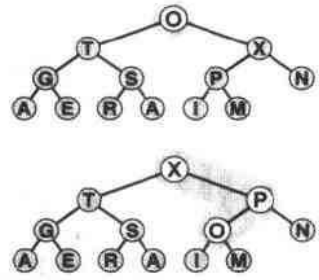



**图 9.4 自顶向下堆化**

上方的树是堆顺序的，除了根节点之外。如果我们把节点 O 和它的两个子节点中较大的子节点（X）交换，整棵树就是堆顺序的了，除了子树的根节点 O 之外。继续把 O 节点和它的两个子节点中较大的子节点互换，直到我们到达堆的底部或是在某一点节点 O 比它的两个子节点都大。这样我们就能够确定整棵树都是符合堆的条件的。我们能够用这个过程作为堆的“删除最大元素”操作的基础，去重新确定堆中根节点被堆底层的最右下方节点取代后，仍然符合堆的条件。

程序 9.3 和 9.4 中基本的 fixUp 和 fixDown 操作也允许直接实现“改变优先级”和“删除元素”的操作。为了改变在堆中某处的元素的优先级，如果优先级是提高了，我们用 fixup 来重构堆，如果优先级下降了，用“fixdown”来重构堆。这些操作的完全实现，涉及特定的数据项，只有在数据结构中有指向每个数据项位置的指针才有意义。我们应该仔细考虑 9.5 节和 9.7 节中的详细实现。

**性质 9.3** 优先队列抽象数据类型的“改变优先级”、“删除元素”和“替换最大元素”操作能够用堆顺序树实现，对任何含有 $N$ 个元素的队列，都只需要不多于 $2\lg N$ 次比较。

由于这些操作需要指向数据项的指针，我们把支持这些操作的实现推迟到 9.6 节（见程序 9.12 和图 9.14）。这些操作都包括沿着一条堆中路径移动节点，在最坏情况下可能是从顶到底或者是从底到顶。■

小心在意“联结”操作并没有被包括在内。高效率地联结两个优先队列看上去需要一个“世故”得多的数据结构。在 9.7 节中我们将会考虑这样一个数据结构。另外，这里给出的基于堆的方法对于广泛的应用程序都是足够的了。它使用最小的额外空间，而且除了在频繁而量大的“联结”操作的情况下，都能保证高效运作。

就如我们提到过的，我们能够用任何优先队列开发一个排序方法，就如程序 9.6 给出的。我们只是把要排序的关键字插入到优先队列中，然后不断用“删除最大元素”操作来把它们从堆中删除并按递减排序。如果用一个无序链表实现优先队列，同样的方法就成了选择排序；用一个有序链表就对应于插入排序。

**程序 9.6 用优先队列排序**

用一个优先队列抽象数据类型对一个子数组 a[1]，…，a[r]排序，我们仅仅是用 insert 操作把所有元素都放进优先队列中，然后用 getmax 操作把它们移出来，按递减排列。这个排序算法运行时间与 $N\lg N$ 成比

例，但会使用与排序元素个数成比例的额外空间。

```
#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();
  }
```

图 9.5 和图 9.6 给出了当使用一个基于堆的优先队列实现的时候，第一阶段的一个例子（构造堆的过程）。图 9.7 和图 9.8 给出了堆实现的第二阶段（我们称为“向下排序”的过程）。从实用的目的来看，这个方法相对不够优美，因为它不必要地造成了一份待排序元素（在优先队列里）的额外拷贝。另外，用 $N$ 次插入操作也不是构造包含 $N$ 个元素的堆最高效率的做法。在下一节里，当我们考虑经典堆排序算法的一个实现的时候，我们会讲到以上这两点。

**练习**

▷ 9.21 在往一个空堆插入关键字 E A S Y Q U E S T I O N 后，给出堆的内容。

▷ 9.22 根据练习 9.1 的约定，堆初始为空，给出下列操作结束后堆的内容：

PRIO*R**I*T*Y***QUE***U*E

9.23 exch 互换，操作是堆中很基本的操作，每次都要读取和保存两次。给出一个更高效率的方法来避免这个问题。

9.24 为什么在“fixDown”操作中我们不用一个标志来避免 j<N 的测试？

○ 9.25 为基于堆的优先队列实现的程序 9.5 加上一个“替换最大元素”的操作。确定考虑了用来作替换的元素比堆中原来所有元素都大的情况。注意：使用 pq[0]能够得出一个好算法。

9.26 在堆的“删除最大元素”操作中必须被移动的关键字最小的一个元素是哪一个？给出一个大小为 15 的堆的被移动的最小关键字元素例子。

9.27 在连续三个堆的“删除最大元素”操作后，必须被移动的关键字最小的元素是哪一个？

## 9.4 堆排序

我们能够用程序 9.6 的基本思想对一个数组排序而不需要任何额外的空间，只要在待排序的数组里不断维持一个堆就可以了。那就是说，焦点集中在排序任务上，我们放弃了把优先队列的实现藏起来的想法，不被优先队列抽象数据类型接口所约束，我们直接用 fixUp 和 fixDown 操作。

把程序 9.5 直接用于程序 9.6 对应于从数组的左边向右边处理，用 fixUp 保证扫描指针的左边的元素形成一个堆顺序的完全树。然后，在向下排序过程，我们把最大的元素放到

由于堆减少而腾出的空间里。那就是说，向下排序过程就好像选择排序，但它使用更高效率的方法从树组中的未排序部分找出最大的元素。

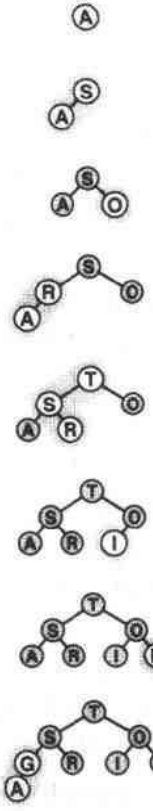


图 9.5 自顶向下的堆构造

这个序列描述了把 A S O R T I N G 插入到一个初始为空的堆中。新的元素被加入到堆底，从左向右排列。每个插入操作只影响插入点到根的路径上的节点，所以最坏情况下的时间耗费是堆的大小的对数时间。

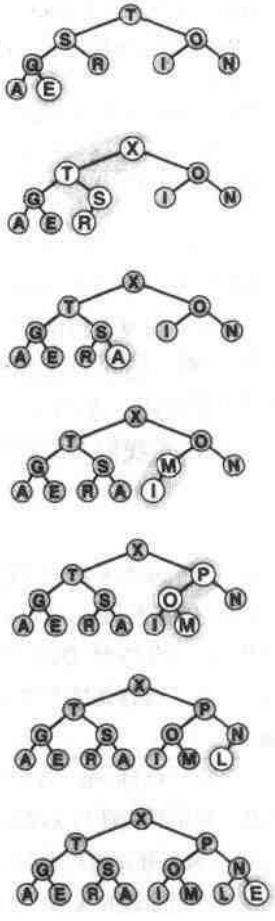


图 9.6 自顶向下构造堆

这个序列描述了向图 9.5 中的堆插入 E X A M P L E 节点的操作。构造一个大小为 N 的堆的总时间是少于

$$\lg 1+\lg 2+\cdots+\lg N,$$

即少于 $N\lg N$。

相对于图 9.5 和图 9.6 中通过在堆中不断执行插入操作来构造堆，通过与之相反的方向来构造堆是一个更高效率的方法，即在堆底不断地构造子堆，就如图 9.9 所示。那就是说，我们把数组中的每个位置都看作是一个小的子堆的堆顶，利用 fixDown 操作对这些子堆的处理和它处理大堆一样好的优点。如果某个节点的两个子节点是堆，那么调用 fixDown 操作能够使该子树成为一个堆。通过不断从数组末尾往回处理，对每一个节点调用 fixDown 操作，我们能够逐步保证数组符合堆的属性。该扫描操作能够从数组的中间开始，因为我

们可以跳过大小为 1 的子堆不处理。

**程序 9.7 堆排序**

在经典堆排序算法中直接用 fixDown 操作。for 循环构造堆，然后 while 循环把最大元素和最末尾的元素互换并且修复堆，直到堆为空为止。指向 a[l-1]的指针 pq 允许代码把下标为 1 的元素到指针所指的元素构成的子数组作为一个完全树的数组处理（详见图 9.2）。一些程序开发环境也许不允许这种用法。

```
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); }
  }
```

一个完整的实现参见程序 9.7，一个经典的堆排序算法。尽管该程序的循环部分看上去做了不同的工作（首先构造堆，然后为向下排序破坏堆），这些代码都基于相同的基本过程，用数组实现一棵完全树，除了根节点外，恢复树的堆顺序。图 9.10 给出了对应于图 9.7 和 9.9 的数组的内容。

**性质 9.4** *自底向上构造堆需要线性时间。*

这个事实是根据观察到大多数被处理的堆都不大。例如，要构造一个有 127 个元素的堆，我们处理 32 个大小为 3 的堆、16 个大小为 7 的堆、8 个大小为 15 的堆、4 个大小为 31 的堆、2 个大小为 63 的堆和一个大小为 127 的堆，那么在最坏情况下共需要 32*1+16*2+8*3+4*4+2*5+1*6=120 次晋升操作（比较操作的两倍）。对 $N=2^n-1$，晋升操作次数的上限是：

$$\sum_{1\le k<n} k2^{n-k-1} = 2^n - n - 1 < N$$

当 $N$+1 不是 2 的幂的时候类似的结论也成立。■

这个性质对于堆排序并不是特别重要，因为它的时间仍然是比不上向下排序耗费的 $N \lg N$ 时间，但它对其他的优先队列应用程序很重要，一个线性时间的构造操作能够生成线性时间的算法。正如图 9.6 所说的，连续用 $N$ 次插入操作构造一个堆在最坏情况下需要 $N \lg N$ 时间（尽管在平均情况下只需要线性时间）。

**性质 9.5** *用堆排序对 $N$ 个元素排序的比较次数少于 $2N \lg N$。*

由性质 9.2 得出的是稍微高一点的 $3N \lg N$ 的结果。这里给出的上界是基于性质 9.4 的一个更加仔细的计算。■

性质 9.5 和就地排序（in-place）的性质是堆排序有实用意义的两个主要原因。无论输

入是什么，它都保证以与 $N \lg N$ 成比例的时间根据位置把 $N$ 个元素排好序。没有什么最坏运行时间的输入使得堆排序的运行明显变慢（不像快速排序），堆排序也不用任何额外的空间（不像归并排序）。保证在最坏情况下的性能要付出代价：例如，堆排序算法的内部循环（每一次比较的耗费）比快速排序有更多的基本操作，对随机文件它比快速排序有更多的比较次数，所以堆排序在典型和随机情况下比快速排序要慢。

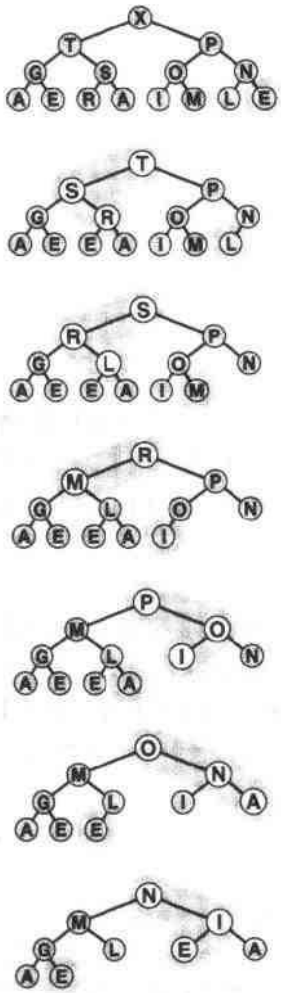


**图 9.7 用堆排序**

用最右下方的元素替换了堆中最大的元素后，我们能够通过详细审查从堆顶到堆底的路径重构堆。

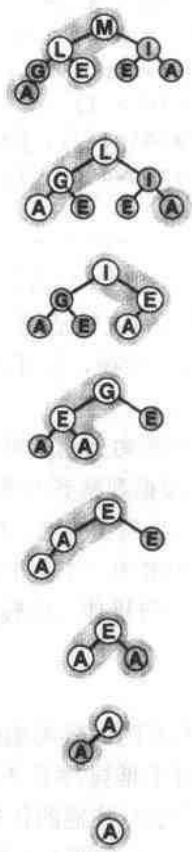


**图 9.8 用堆排序**

在图 9.7 中描述了把剩下的元素从堆中移出。即使所有的元素都由堆顶移回到堆底，排序的总时间耗费仍少于 $\lg 1+\lg 2+\cdots+\lg N$，即少于 $N \lg N$。

堆对解决在 $N$ 个元素中找出第 $k$ 大的元素也有用（参看第七章），特别是 $k$ 比较小的情况。在堆排序算法中，我们在把第 $k$ 个元素从堆顶移出的时候停止算法。

**性质 9.6**　基于堆的选择操作允许我们在 $k$ 比较小的时候以和 $N$ 成比例的时间找到第 $k$ 大元素，如果 $k$ 接近 $N$，则需要与 $N \lg N$ 成比例的时间。

一个方法是构造一个堆，用少于 $2N$ 次的比较操作（由性质 9.4），然后移出最大的 $k$ 个元素，需要用 $2k \lg N$ 或者更少的比较（由性质 9.2），共 $2N+2k \lg N$ 的时间。另一个方法是构造一个最小的大小为 $k$ 的堆，然后用剩下的元素进行 $k$ 次“替换最小元素”的操作（删除最小元素，然后是插入操作），共需要最多 $2k+2(N-k)\lg k$ 次比较（参看练习 9.35）。这个方法只要使用与 $k$ 成比例的空间，所以当 $k$ 比较小而 $N$ 很大（或不能提前知道大小）的时候，对检索最大的 $k$ 个元素是很有吸引力的。在随机或典型情况下，当 $k$ 相对于 $N$ 来说比较小时，第二种方法中 $\lg k$ 的上界可以近似地看成是 $O(1)$。■

有许多提高和改进堆排序的方法被发掘出来。由 Floyd 提出的一个想法是，注意到在向下排序过程中一个被再次插入到堆中的元素通常要遍历直到堆底的整条路径，所以我们能够通过避免检测该元素是否到达它的位置而节省时间，只需要简单地把两个子节点中较大的一个晋升直到到达堆底，然后往堆顶回溯直到合适的位置。这个想法减少了比较的次数，使得比较次数的图形是类似渐近线的——接近 $\lg N! \approx N \lg N - N/\ln 2$，这是任何比较算法所需要比较次数的绝对最小值（参看第八部分）。这个方法需要额外的簿记（bookkeeping），在实践中它仅仅用于比较操作耗时相对很高的情况（例如，我们需要对关键字是字符串的记录或者其他长关键字的数据类型进行排序）。

另外一个想法是构造基于数组表示的完全堆顺序的三叉树的堆，位于 $k$ 的节点大于或等于位于位置 $3k-1$，$3k$ 和 $3k+1$ 的节点，同时小于或等于 $\lfloor (k+1)/3 \rfloor$ 位置的节点，$k$ 是 1 到 $N$ 之间的 $N$ 个元素的数组的某个位置。在减少树的层数和增加检索每个节点的三个子节点的耗费两者之间有一个折中。这个折中由具体实现的细节决定（详见练习 9.30）。进一步增加每个节点的子节点数并不见得有建设性。

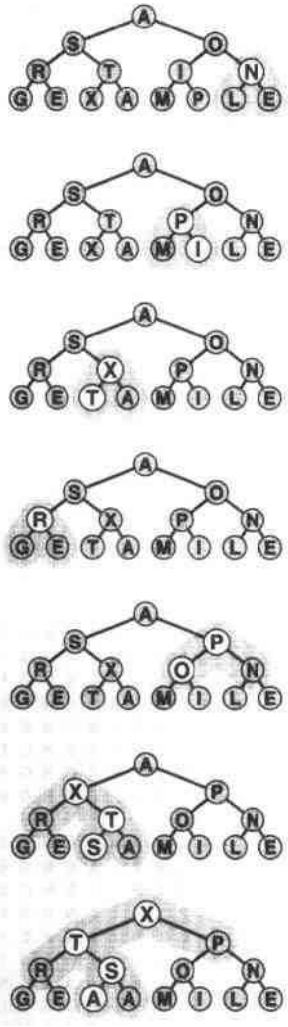


**图 9.9　自底向上构造堆**

从右往左并自下而上处理，我们通过确定“当前节点下方的子树是堆顺序的”构造一个堆。在最坏情况下总的时间耗费是线性的，因为大多数节点都在堆底附近。

图 9.11 给出了堆排序对一个随机顺序文件的操作。首先，过程看上去只是排序，因为在构造堆的时候大量的元素被移动到文件的开头。但然后就如预期的一样，该方法看上去更像一个选择排序的镜像。图 9.12 显示了不同类型的输入文件能够产生特殊属性的堆，但从排序的进程来看它们更像是随机的堆。

很自然地，我们对如何为某个应用程序在堆排序、快速排序和归并排序之间取舍有兴趣。堆排序和归并排序的选择主要归结于不稳定的排序（参看练习 9.28）和需要额外内存的排序之间的选择；堆排序和快速排序的选择归结于平均情况下速度和最坏情况下速度之间的选择。人们已经广泛地改进了快速排序和归并排序内部循环，我们把同样的改进工作放到本章的练习中。使得堆排序比快速排序快是不大可能的——就如表 9.2 中由研究经验指出的那样——但那些对自己机器中高速排序感兴趣的人会发现这个练习有启发性。就如通常那样，不同机器的特性和编程环境会担当重要角色。例如，快速排序和归并排序有位置特性，所以在特定的机器上会给它们带来更多的好处。当比较操作是极端费时的时候，Floyd 的方法是很理想的，因为这样的情况下它是时间和空间折中的最佳选择。

**图 9.10 堆排序示例**

堆排序是一个高效率的基于选择的算法。首先，我们在适当位置自底向上构造一个堆。图中最上面的 8 行对应于图 9.9。然后，我们不断地把最大元素从堆中移出、底部无阴影部分对应于图 9.7 和图 9.8；有阴影部分包含增长的已排序文件。

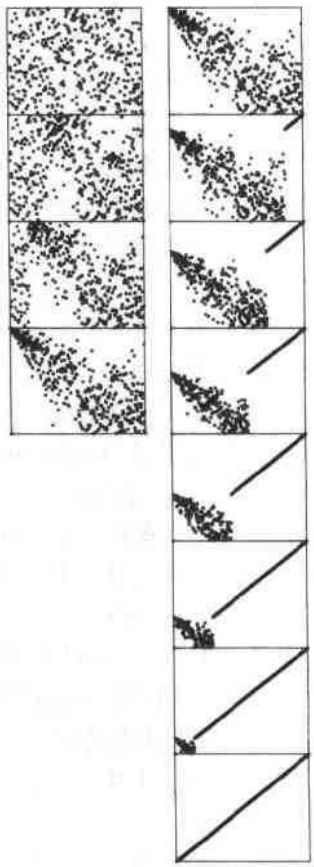


**图 9.11 堆排序的动态特性**

构造堆的过程（左边）看上去没有对文件排序，只是把大的元素放在起始位置的附近。然后，向下排序的过程（右边）像选择排序那样工作，在起始位置保持一个堆并且在文件的末尾建立一个排好序的数组。

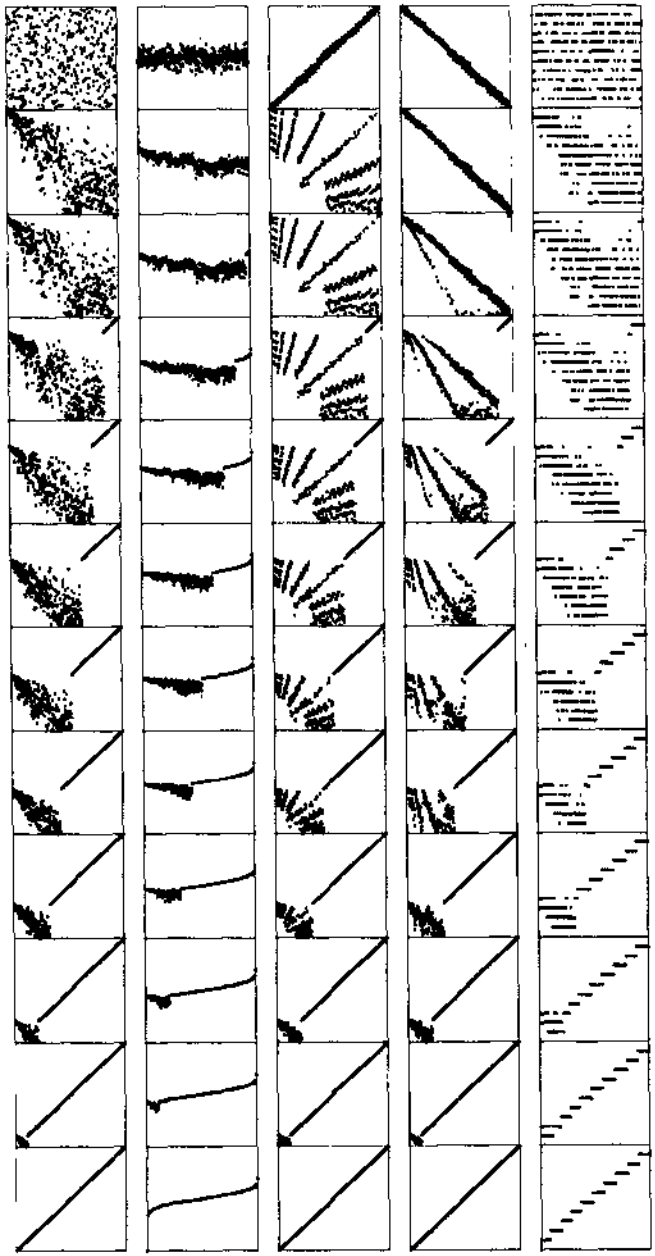


**图 9.12　堆排序在不同类型的文件中的动态特性**

堆排序的运行时间对输入并不是特别敏感。无论什么输入，最大的元素都会在少于 $\lg N$ 步内找到。这些图表显示文件是随机的、高斯序列、几乎有序的、几乎逆排序的和带有 10 个不同关键字的随机排序（在最上方，从左到右）。第 2 个图从顶部开始显示自底向上算法构造的堆，其余的图显示对每个文件的向下排序过程。堆有时候是把初始文件作一个镜像，但随着排序过程的进行，随机文件都变得更像堆了。

表 9.2 根据经验对堆排序算法的研究

表左边显示的对随机整数文件应用各种排序方法的时间比较确认了我们的期望，堆排序比快速排序要慢但比归并排序要快。表右边最前面的 $N$ 个单词的时间显示，Floyd 的方法在比较操作很费时的情况下是对堆排序的一个有效改进。

| | 32 位整数关键字 | | | | | 字符串关键字 | | |
|---|---|---|---|---|---|---|---|---|
| $N$ | 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 | | | |

其中：

Q 快速排序，标准实现（程序 7.1）

M 归并排序，标准实现（程序 8.1）

PQ 基于优先队列的快速排序（程序 9.5）

H 堆排序，标准实现（程序 9.6）

F Floyd 的堆排序改进

## 练习

9.28 显示出堆排序是不稳定的。

● 9.29 根据经验测定堆排序用于构造操作的时间百分比是多少，$N=10^3$，$10^4$，$10^5$ 和 $10^6$。

● 9.30 实现一个基于完全堆顺序的三叉树的堆排序，就如课文中所描述的。根据经验比较你的程序和标准实现的比较操作次数，当 $N=10^3$，$10^4$，$10^5$ 和 $10^6$。

● 9.31 继续练习 9.30，根据经验测定 Floyd 的方法对三叉堆是否有效。

○ 9.32 只是考虑比较的耗费，假设需要 $t$ 次比较来找出最大的 $t$ 个元素，当堆排序使用 $t$ 叉堆的时候，找出使得比较操作次数 $N \log N$ 的系数最小的 $t$。首先，假设是程序 9.7 的一个直接一般化；然后，假设 Floyd 的方法能够在内部循环中减少一次比较。

○ 9.33 当 $N=32$ 时，给出一个关键字的排列，使得堆排序执行尽量多的比较操作。

●● 9.34 当 $N=32$ 时，给出一个关键字的排列，使得堆排序执行尽量少的比较操作。

9.35 证明构造一个大小为 $k$ 的优先队列，然后执行 $N-k$ 次“替换最小元素”（“插入”操作，接着“删除最小元素”操作）操作，会余下最大的 $k$ 个元素留在堆中。

9.36 实现在性质 9.6 的讨论中提到的基于堆排序的选择算法的两个版本，使用练习 9.25 中描述的方

法。凭经验比较它们与第七章的基于快速排序方法所执行的比较操作次数，$N=10^6$，$k=10$，100，$10^3$，$10^4$，$10^5$和$10^6$

● 9.37 实现基于表示前序堆顺序树而不是层顺序思想的堆排序。凭经验比较这个版本和标准实现的比较操作次数，关键字随机排列，$N=10^3$，$10^4$，$10^5$和$10^6$。

## 9.5 优先队列抽象数据类型

对大多数应用程序而言，我们希望把优先队列例程化，而不是执行“删除最大元素”操作后返回一个值，告诉我们哪个记录有最大的关键字，而且在其他操作中也一样。也就是说，我们指定优先级和使用优先队列仅仅是为了在一个适当的顺序中访问其他信息。这个安排相似于在第六章描述过的间接排序和指针排序的概念。特别是，这个方法需要诸如“改变优先级”和“删除”的操作有意义。在这里我们仔细检查这种思想的一个实现细节，不但因为我们将要在本书后面这样使用优先队列，而且因为这种情况是我们设计优先队列抽象数据类型的接口的时候要面对的问题原型。

当我们希望从一个优先队列中删除一个记录，我们如何确定是哪一个记录？当希望维持多个优先队列的时候，我们如何组织实现方法使得我们处理其他类型的数据时能够用统一的方法控制优先队列？像这样的问题是第四章的主题。程序 9.8 接着我们在 4.8 节讨论的话题给出了一个优先队列的普遍接口。它支持的情况是，一个客户有关键字和相关的信息，而主要对与最高关键字相关的信息感兴趣，他可以对对象有许多其他的数据处理操作，就如我们在这一章开头讨论的那样。所有的操作引用特定优先队列的时候都要通过一个句柄（一个不确定的指向一个结构的指针）。“插入”操作返回客户程序加入到队列中的对象的句柄。对象句柄不同于优先队列句柄。在这种安排下，客户程序为跟踪句柄负责，它们在后面可以确定哪些对象会被“删除”和“改变优先级”操作所影响，以及哪些优先队列会被所有的操作影响。

**程序 9.8 完整的优先队列抽象数据类型**

这个优先队列抽象数据类型的接口允许客户程序删除记录、改变优先级（用实现提供的句柄）和联结优先队列。

```
template <class Item>
class PQ
  {
    private:
      // 决定于实现的代码
    public:
      // 决定于实现的句柄定义
      PQ(int);
      int empty() const;
```

```
    handle insert(Item);
    Item getmax();
    void change(handle, Item);
    void remove(handle);
    void join(PQ<Item>&);
};
```

这种安排给客户程序和实现都带来了约束。客户程序除了通过这个接口外，没有其他方法通过句柄访问信息。它有责任适当使用句柄：例如，对一个实现来说没有好的方法来检查诸如一个客户使用某个已经被删除记录的句柄是否合法。在这一部分，实现不能随意移动信息，因为客户程序有它们要在稍候使用的句柄。当我们检查实现的细节时，这一点会变得更加清晰。就像往常一样，无论我们在自己的实现里检查什么程度的细节，一个如程序 9.8 的抽象接口都是一个有用的出发点，这个出发点在应用程序和实现的需求之间取得一个折中。

在程序 9.9 中给出了直接实现基本优先队列的操作，它使用无序双向链表的表示方式。这些代码显示出接口的本性：由它很容易开发出其他的、相似的、直接的、使用其他基本表示方式的实现。

**程序 9.9 无序双向链表的优先队列**

这个实现包括程序 9.8 中的“构造”、“测试是否为空”和“插入”操作（参看程序 9.10 其他 4 个操作的实现）。它维护一个简单的无序链表，该链表带有头尾节点。我们确定结构节点是双向链表节点（包括一个记录和两个链接）。私有数据表示构成链表的头尾链接。

```
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>&);
};
```

正如我们在 9.1 节讨论的，程序 9.9 和 9.10 给出的实现对优先队列比较小而且“删除最大元素”和“查找最大元素”操作都不频繁的应用程序是很适合的；另外，基于堆的实现是完美的。用额外的链接为堆顺序树实现 fixUP 和 fixDown 操作，而维持句柄完整是我们留在练习里的挑战，因为在 9.6 和 9.7 节我们将会详细考虑两个可供替换的方法。

一个如程序 9.8 那样的完全的抽象数据类型有很多好处，但有时候考虑一个其他的安排也是有好处的，对客户程序和实现有不同的约束。在 9.6 节我们考虑一个例子，客户程序负起维护记录和关键字的责任，优先队列例程直接引用它们。

接口上的轻微改动也是合适的。例如，我们可能希望有一个函数返回优先队列最高优先级的值，而不是仅仅一个方法来访问该关键字和相关信息。我们在 4.8 节考虑过的关于内存管理和复制语义的话题开始活动了。我们不考虑“析构”和真实的“复制”操作，而且选择了“联结”操作的几种可能之一（详见练习 9.43 和 9.44）。

把这些程序加入到程序 9.8 的接口中很容易，但开发一个所有操作都保证在对数时间里完成的实现就有挑战性得多了。在优先队列不会变得太大或者“插入”和“删除最大元素”操作的混合有一些特定的性质的应用程序里，灵活的接口是需要的。在另一方面，一个队列会变得巨大而且是成十倍或成百倍地增大，性能方面就应该注意和详细考虑了，值得约束确保高效性能的操作集。很多研究是进行混合不同操作的优先队列算法的设计的，在 9.7 节讲述的二项式队列是一个很重要的例子。

## 练习

9.38 你会用哪一个优先队列实现去检索 $10^6$ 个随机元素的集合中最小的 100 个元素？证明你的回答是正确的。

9.39 提供类似于程序 9.9 和 9.10 的实现，使用有序双向链表。注意：因为客户程序在数据结构中有

句柄，你的程序只能够修改节点的链接（而不是关键字）。

**程序 9.10 双向链表优先队列（续）**

用这些实现替换了程序 9.9 中相应的声明，产生了一个完整的优先队列的实现。“删除最大元素”操作需要扫描整个列表，但维护双向链表的耗费是由“改变优先级”、“删除”和“联结”操作都是在常数时间实现的事实决定的，只需要对链表进行基本操作（参看第三章关于双向链表的细节）。

如果想的话，我们能够增加析构函数、复制构造函数和重载赋值操作符来更进一步把这个实现发展为一级抽象数据类型（详见 4.8 节）。注意“联结”操作的实现从包括在结果中的自变量访问链表节点，但不会对节点进行拷贝。

```
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;
  }
```

9.40 提供“插入”和“删除最大元素”（程序 9.1 的优先队列接口）操作的实现，使用由节点和链接清楚表示的完全堆顺序树。注意：因为客户程序没有数据结构中的句柄，你能够利用在节点间交换信息比交换接点更容易的事实。

● 9.41 提供“插入”、“删除最大元素”、“改变优先级”和“删除”操作（程序 9.8 的优先队列接口）的实现，使用含有显式链接的堆顺序树。注意：因为客户程序在数据结构中有句柄，这个练习比练习 9.40

要难，不仅因为节点有三个链接，而且因为你的程序只能够改变节点的链接（而不是关键字）。

9.42　为你练习 9.41 中的实现加入一个“联结”操作。

○ 9.43　往程序 9.8 中加入析构函数、复制构造函数和重载赋值操作符的声明，使它变成一个一级抽象数据类型，往程序 9.9 和 9.10 加入相应的实现，编写一个驱动程序来测试你的接口和实现。

● 9.44　改变程序 9.9 和 9.10 中“联结”操作的接口和实现，使得它返回一个 PQ（联结变量的结果）并有释放该变量的效果。

9.45　提供一个支持“构造”和“删除最大元素”操作的优先队列接口和实现，使用联赛（详见 5.7 节）。程序 5.19 会为你提供基本的“构造”操作。

● 9.46　把你关于练习 9.45 的解决方案转换成一个一级抽象数据类型。

● 9.47　在你关于练习 9.45 的解决方案中加入一个“插入”操作。

## 9.6　索引元素的优先队列

设想在优先队列中处理的记录是在一个已有的数组里。在这种情况下，使优先队列通过数组的下标来引用队列中的元素就变得有意义了。此外，我们能够使用数组的下标作为句柄来实现所有的优先队列操作。在程序 9.11 和图 9.13 中给出的接口展示了如何把这种方法应用到我们在第六章讲过的根据下标排序的例子中。没有对记录的复制和修改，我们能够用一个优先队列来保存一个记录的子集。

在一个数组中使用下标来访问记录是很自然的安排，但这种方法导致在实现上与程序 9.8 对立。现在客户程序不能够自由地移动信息了，因为优先队列通过管理下标管理了原来由客户程序管理的数据。在这部分，优先队列的实现在客户程序不给出下标的情况下就不能使用下标。

为了开发一个实现，我们精确地使用了与我们在 6.8 节利用下标排序一样的方法。我们控制下标和重载的操作符<，使得比较操作与客户程序的数组相关。这里复杂性增加了，因为优先队列程序需跟踪对象，这样它才能够在客户程序通过句柄（数组下标）应用对象的时候找到它们。基于这一点，我们增加一个下标数组来记录对优先队列中关键字的位置。为了本地化这个数组的维护，我们仅仅在“exch”操作中移动数据，然后适当定义“exch”操作。

**程序 9.11　索引元素的优先队列抽象数据类型接口**

这个接口提供在客户程序中用下标构造优先队列的方法，而不是用数据结构本身构造优先队列。“插入”、“删除最大元素”、“修改优先级”和“删除”操作程序都使用包含一个数组下标的句柄，客户程序重载操作符<来比较两个数组的入口。例如，客户程序可能定义操作符<使得 i<j 是 data[i].grade 和 data[j].grade 比较的结果。

```
template <class Index>
class PQ
  {
```

```
    private:
      // 与实现相关的代码
    public:
      PQ(int);
      int empty() const;
      void insert(Index);
      Index getmax();
      void change(Index);
      void remove(Index);
};
```

这种方法的一个利用堆的完整实现在程序 9.12 中给出。这个程序只是和程序 9.5 有稍微不同，但它很值得学习，因为它很有实际应用价值。我们把这个程序建立的数据结构称为“索引堆”。我们将会使用这个程序作为其他从第五部分到第七部分的算法的构件。就像通常一样，我们不做任何的错误检测，而我们假设（例如）下标是永远在有效范围内，而且用户也不会尝试往一个满的队列中加入记录或从一个空队列中删除元素。增加这样的检测代码也是很容易的事情。

我们能够对任何使用数组表示的优先队列（如练习 9.50 和练习 9.51）应用同样的方法。这种间接访问的方法最主要的缺点是要用到额外的空间。下标数组的元素必须和数据数组的元素一样多，而优先队列空间的最大值可以是小得多。另一个在数组中建立优先队列的方法是让客户程序生成包含关键字和数组下标的相关信息，或者使用一个下标关键字和程序支持的重载操作符<。那么，如果实现使用一个如程序 9.9 和 9.10 或练习 9.41 的连续表示方式，那么在任何时候优先队列使用的空间就会和它所能容纳的最大元素个数成比例。这样我们的实现方法在空间资源紧张和优先队列只是包含数组数据一小部分的情况下就显得相当完美了。

| 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 |

**图 9.13 索引堆数据结构**

通过控制索引，而不是记录本身，我们能够在一个数组的记录子集上建立一个优先队列。这里，一个在数组中含有 5 个元素的堆 pq 含有最高分的 5 名学生的索引。这样，data[pq[1]].name 包含“Smith”，最高分学生的名字，以此类推。一个逆向数组 pq 允许优先队列的程序把数组下标看作是句柄。例如，如果我们需要把“Smith”的分数改为 85，我们改变 data[3].grade 的入口，然后调用 PQchange(3)。优先队列的实现访问在 pq[qp[3]]的记录（或者 pq[1]，因为 qp[3]=1）和在 data[pq[1]].name 的新关键字（或者 data[3].name，因为 pq[1]=3）。

把这个方法与在 9.5 节提供的优先队列的完整实现相比较，就可以看出它们在抽象数据类型设计上的根本不同。在第一种情况里（例如程序 9.8），分配和回收关键字的内存、改变关键字的值等等的工作是优先队列的责任。这个抽象数据类型支持客户程序带有对队列中元素的句柄，客户程序只通过调用优先队列的例程来访问队列中的元素，句柄是调用例程的参数。在第二种情况里（例如程序 9.12），

客户程序为关键字和记录负责，而且优先队列的例程只通过由用户（在程序 9.12 中是数组下标）提供的句柄访问关键字和记录的信息。两种情况都需要客户程序和抽象数据类型实现之间的合作。

**程序 9.12　基于索引堆的优先队列**

程序 9.11 的实现包括在客户程序中维持一个索引数组中的 pq。例如，如果客户程序为 Index 类型的参数定义了操作符<，就如程序 9.11 前在注释中指出的一样，然后，当 fixUp 操作比较 pq[j]和 pq[k]，实际上是比较 data.grade[pq[j]]和 data.grade[pq[k]]。我们假设 Index 是一个包装类，它的对象能够指向数组，使得我们能够保持堆的位置对应于 pq[k]中的 k 值，也允许我们实现“改变优先级”和“删除”操作（参看练习 9.49）。对堆中的所有 k 值，我们保证 pq[qp[k]] = qp[pq[k]] = k（参看图 9.13）。

```
template <class Index>
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(pq, N); }
      Index getmax()
        {
          exch(pq[1], pq[N]);
          fixDown(pq, 1, N-1);
          return pq[N--];
        }
      void change(Index k)
        { fixUp(pq, qp[k]); fixDown(pq, qp[k], N); }
  };
```

注意，在这本书里，通常我们对协作的兴趣超过鼓励使用编程语言所支持的机制。特别是，我们希望实现的性能特性适应客户所需要操作的动态组合。确定这种适应性的一个方法是寻找有确定最坏情况性能极限的实现，但我们能够更简单地通过用简单的实现匹配客户对性能的需求来解决很多问题。

### 练习

9.48 假设一个数组已经包含了关键字 EASYQUESTION。在这些关键字都用程序 9.12 插入到一个初始为空的堆里面之后，给出 pq 和 qp 数组的内容。

○ 9.49 为程序 9.12 加入一个“删除”操作。

9.50 用一个有序数组表示一个优先队列，实现一个索引元素的优先队列抽象数据类型（参看程序 9.11）。

9.51 用一个无序数组表示一个优先队列，实现一个索引元素的优先队列抽象数据类型（参看程序 9.11）。

○ 9.52 已有一个含有 $N$ 个元素的数组，考虑一棵含有 $2N$ 个节点、包括数组的索引的二叉树（表示为一个数组 pq），它有以下属性：

（i）对 $i$ 从 0 到 $N-1$，我们有 pq[N+i]=i；

（ii）对 i 从 0 到 N-1，如果 a[pq[2*i]]>a[pq[2*i+1]]，我们有 pq[i]=pq[2*i]，另外，我们有 pq[i]=pq[2*i+1]。这样一个结构叫做一个“索引堆联赛”，因为它结合了索引堆和联赛（参看程序 5.19）的特点。给出与 EASYQUESTION 相对应的索引堆联赛。

○ 9.53 实现索引元素的优先队列抽象数据类型（参看程序 9.11），使用索引堆联赛（参看程序 9.45）。

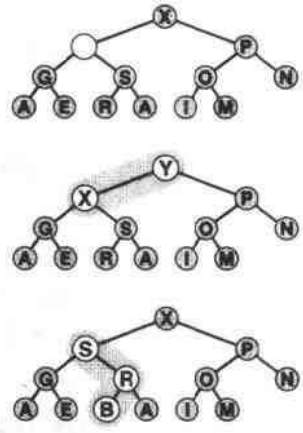



**图 9.14 改变一个堆中节点的优先级**

除了某一个给出的节点之外，最上方的图描述了一个被称为堆顺序的堆，如果该节点是大于它的父节点，那么它必须上移，就如图 9.3 所说的。这种情况显示在中间的图中，Y 被上移（一般情况下，在移到树根之前上移操作就结束了）。如果该节点是小于两个子节点中较大的一个，那么它就必须下移，就如图 9.3 所说的。这种情况显示在底部的图中，B 被下移（一般情况下，在移到树底部之前上移操作就结束了）。我们能够用这个过程作为改变优先级的基本操作，在改变节点的关键字后恢复堆；或者作为堆的“删除”操作的基础，在用最底层最右边的关键字代替了某个节点中的关键字后，对堆进行重构。

## 9.7 二项式队列

没有一种我们考虑过的实现方法允许我们实现“联结”、“删除最大元素”和“插入”操作都在最坏情况下有高效性能。无序链表有快速的“联结”和“插入”操作，但“删除最大元素”操作就很慢；有序链表有快速的“删除最大元素”操作，但“联结”和“插入”操作就很慢；堆有快速的“插入”和“删除最大元素”操作，但“联结”操作就很慢。（详见表 9.1）在频繁或数据量很大的联结操作担任重要角色的应用程序里，我们需要考虑更多高级数据结构。

在上文，我们用“高效”来表达操作在最坏情况下不应该用超过对数时间。这个约束看上去排除了数组表示方式，因为我们显然只能够用线性时间把其中小的一个数组的元素加入到大的数组中。程序 9.9 的无序双向链表表示方式的“联结”操作需要常数时间，但需要我

们遍历整个链表来执行“删除最大元素”操作。使用一个有序双向链表（练习 9.39）给出了一个常数时间的“删除最大元素”操作，但需要线性时间归并链表实现“联结”操作。

许多数据结构都发展出能够支持所有优先队列操作的高效实现。它们中的大部分都是基于堆顺序树的直接链表表示。在树中执行向下移动操作需要两个链接（或者是对二叉树的两个子节点，或者是在用一般树表示二叉树的时候指向第一个子节点和兄弟节点的链接），而在向上移动操作时需要与父节点的一个链接。开发堆顺序操作的实现，而且这些操作为任何（堆顺序）包括节点和链接的树结构或者其他表示方式工作通常是件容易的事情。难点在于动态操作，例如“插入”、“删除”和“联结”，它们都需要我们对树的结构进行调整。不同的数据结构基于不同的修改树结构并且要维持树平衡的策略。通常，算法使用的树比完全树更灵活，但保证树被高效率地保持平衡，从而确保在最坏时间情况下以对数时间为上界。

管理一个三叉树结构的“费用”可能是一个繁重的负担——确保一个特定实现在所有环境下正确地维持那三个指针是一个巨大的挑战（参看练习 9.40）。不但如此，在很多实际情况下，证明高效实现所需要的所有操作是困难的，所以我们应该在提出这样一个实现前要先暂停一下。在另一方面，证明高效实现是不需要的也是困难的，但保证所有的优先队列操作都很快的投资是值得的。不理会这些考虑，下一步从堆到一个允许“联结”、“插入”和“删除”操作的高效实现的数据结构很迷人，而且值得认真学习。

即使有一个树结构的链接表达方式，堆和堆顺序二叉树的条件都太强了，以至于不能实现“联结”操作的高效实现。给出两棵堆顺序的树，我们如何把它们归并在一起？例如，如果其中一个棵树有 1023 个节点而另一棵有 255 个节点，我们如何把它们归并为一棵 1278 个节点的树而不处理超过 10 到 20 个节点？如果树是堆顺序和完全的，一般看上去不可能归并堆顺序树，但不同的高级数据结构已经被设计出来，以减弱堆顺序和平衡条件来得到设计高效“联结”操作的灵活性。下一步，我们考虑这个问题的一个有独创性的解决方法，叫做二项式队列，由 Vuillemin 在 1978 年发展出来。

首先，我们要注意“联结”操作对于一个特定类型而且对堆顺序要求不严格的树没有太高的价值。

**定义 9.4** *一棵包含有关键字节点的二叉树被称为“左堆序”（left heap order），如果每个节点的关键字都是大于或等于该节点左子树的所有节点的关键字（如果有的话）的话。*

**定义 9.5** *一个“2 次方堆”（power-of-2 heap）是一棵左堆序树，根节点有空的右子树和完全左子树。一棵二项式树的左孩子对应一个二次方堆，而它的右孩子放在它的右兄弟的位置上。*

二项式树和二次方堆是等价的。我们同时使用这两种表达方式是因为二项式树想像起来稍微容易一些，而二次方堆的简单表达方式通向简单的实现。特别是，我们依靠以下由上文定义导出的事实。

- 二次方堆的节点数是 2 的幂；
- 根节点的关键字最大；
- 二项式树是堆顺序的。

**程序 9.13 联结两个同样大小的二次方堆**

我们只需要修改一些链接来把两个等大的二次方堆联结成一个增大了一倍的二次方堆。我们定义这个函数作为实现中的一个私有成员，它是二项式队列算法高效实现的关键。

```
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 p; }
  }
```

二项式队列算法的基本操作是联结两个节点数相等的二次方堆。生成一个两倍节点数的二次方堆很容易，就如图 9.16 说明的一样。两个根节点有较大关键字的一个成为结果堆的根节点（较小的根节点所在的堆称为结果堆根节点的左子树），而它的左子树成为较小根节点的右子树。给出一个树的链接表示，“联结”操作是一个常数时间的操作：我们简单地调整顶部的两个链接。程序 9.13 中给出了一个实现方法。这个基本操作是 Vuillemin 解决“实现无慢操作的优先队列”一般方法的核心。

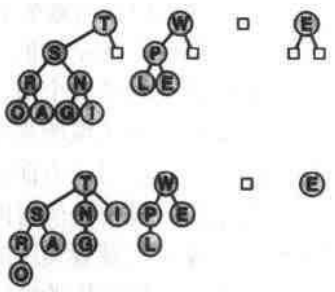


**图 9.15 一个大小为 13 的二项式队列**

一个大小为 N 的二项式队列是一列左堆序的二次方堆，每个二次方堆在 N 的二进制表示中代表一个位。这样，个大小为 $13=(1101)_2$ 的二项式队列包括一个 8-堆、一个 4-堆和一个 1-堆。这里给出了同一个二项式队列的左堆序的二次方堆（上方）和堆顺序的二项式树（下方）的表示。

**定义 9.6** 一个二项式队列是一个二次方堆的集合，而且其中没有两个二次方堆的大小是一样的。二项式队列决定于队列节点的数目，对应于整数的二进制表示法。

一个 N 个元素的二项式队列有一个二次方堆代表在 $N$ 的二进制表示中有一位是 1。例如，一个 13 个节点的二项式队列包括一个 8-堆、一个 4-堆、一个 1-堆，如图 9.15 所示。在一个大小为 N 的二项式队列中最多有 $\lg N$ 个二次方堆，而且这些堆的高度都不到 $\lg N$。

相应于定义 9.5 和 9.6，我们用包含关键字和两个链接的节点（就好像图 5.10 的树表示方式一样）来表示二次方堆（和指向元素的句柄）；通过把以下的代码包括进程序 9.8，我们把二项式队列表示成二次方堆的数组。

```
struct node
  { Item item; node *l, *r;
    node(Item v)
      { item = v; l = 0; r = 0; }
  };
typedef node *link;
link* bq;
```

数组不大而且树也不高；就如我们将要看到的，这个表示方式高效而且适应性强，允许所有优先队列操作在少于 lgN 的步骤内完成。

首先，让我们考虑“插入”操作。往一个二项式队列中插入一个新的元素的过程精确地反映了一个二进制数的增加。为了把一个二进制数加一，我们从右往左移，改变其中的 1 和 0；由于 1+1 等于二进制的 10，我们直到找到最右方的 0，然后把它改为 1。用相似的方法，在一个二项式队列中加入一个元素的时候，我们从右往左移，把对应于 1 的堆和进位堆联结，直到找到最右边的空位置插入进位堆。

特别是，把一个新元素插入到二项式队列中，我们把新元素作为一个 1-堆。然后，如果 $N$ 是偶数（最右的一位是 0），我们只要把这个 1-堆放在二项式队列最右边的空位置上。如果 $N$ 是奇数（最右的一位是 1），我们把对应新元素的 1-堆和二项式队列最右边的 1-堆联结成一个进位 2-堆。如果二项式队列中对应 2 的位置是空的，我们就把进位堆放在这里；否则，我们把进位 2-堆和二项式队列中的 2-堆联结成一个进位 4-堆，以此类推，直到我们在二项式队列中找到一个空位置。整个过程在图 9.17 中有所描述。程序 9.14 是一个实现。

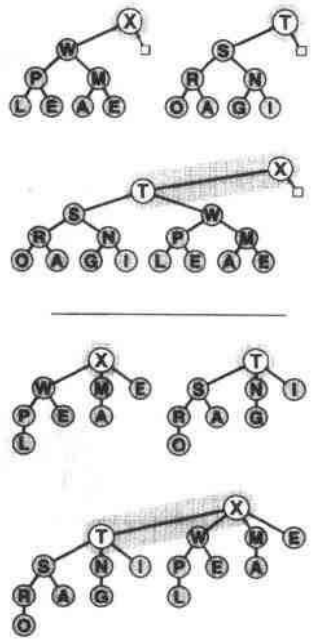


**图 9.16　联结两个同样大小的二次方堆**

我们把两个二次方堆（上方）中较大的根节点作为联结后的根节点，这个根节点的左子树作为原先另外一个根节点的右子树。如果原来每个堆有 $2^n$ 个节点，联结结果就有 $2^{n+1}$ 个节点。如果原来是左堆序的，联结结果也是左堆序的，最大的元素作为根节点。堆顺序二项式树表示方法的等价操作会在下文给出。

其他二项式队列操作用二进制算法来解释都很容易理解。就如我们将要看到的，实现“联结”对应于二进制数的加法。

目前，假设我们有一个“联结”操作的（高效率的）函数是把两个优先队列的联结结果放置在第一个优先队列中。用这个函数，我们就能够通过调用“联结”函数来实现“插入”操作，只不过其中一个被联结的二项式队列只含有一个元素（参看练习 9.63）。

我们也能够调用“联结”操作实现“删除最大元素”操作。为了找出一个二项式队列中的最大元素，我们扫描队列中的二次方堆。这些堆的每一个都是左堆序的，所以它的根节点是最大元素。二项式队列中的最大元素是所有根节点中最大的一个。因为在二项式队列中只有不超过 $\lg N$ 个堆，寻找最大元素的时间不会超过 $\lg N$。

**程序 9.14　二项式队列中的插入操作**

要在一个二项式队列中插入一个节点，我们先用这个节点生成一个 1-堆，并标志为一个进位 1-堆，然后从 $i$=1 开始重复以下过程。如果二项式队列没有 $2^i$-堆，我们把进位 $2^i$-堆放进队列中。如果二项式队列有 $2^i$-堆，我们把两个 $2^i$-堆联结（调用程序 9.13 中的 pair 函数）生成一个 $2^{i+1}$-堆，$i$ 加 1，重复这些操作直到在二项式队列中找到一个空堆位置。

```
handle insert(Item v)
  { 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;
  }
```

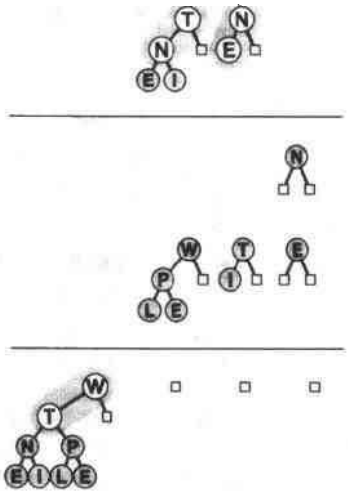


图 9.17 向一个二项式队列中插入一个新元素

在一个有 7 个节点的二项式队列中插入一个新元素与二进制加法 $(111)_2+1=(1000)_2$ 相类似，在每一位都有进位。二项式队列依插入结果是在底部的图中，有一个 8-堆和空的 4、2、1 堆。

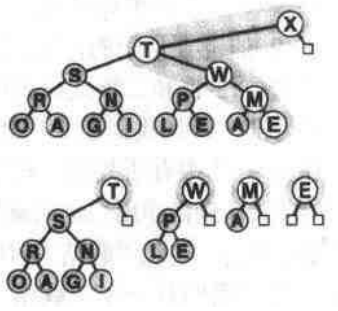


图 9.18 在一个二次方堆中删除最大元素

删除根节点产生一个二次方堆的森林，全都是左堆序的，根节点都是原来树最右沿上的节点。这个操作提供了二项式队列删除最大元素的方法：删除二次方堆的根节点，然后用“联结”操作把原来二项式队列剩下的二次方堆归并成新的二项式队列。

要执行“删除最大元素”操作，我们注意到删除一个 $2^k$-堆的根节点会留下 $k$ 个左堆序的二次方堆——一个 $2^{k-1}$-堆，一个 $2^{k-2}$-堆……我们能够很容易地重构一个大小为 $2^k$-1 的二项式队列，如图 9.18 所示。然后，我们能够用“联结”操作来把这个大小为 $2^k$-1 的二项式队列和剩下的其他队列联结，从而完成“删除最大元素”操作。这个实现在程序 9.15 中给出。

**程序 9.15 在一个二次方堆中删除最大元素**

我们先扫描所有根节点找出最大元素，然后从二项式队列中删除包含最大元素的二次方堆。我们接着从该堆中删除最大元素节点，并临时构造一个包含该二次方堆余下组成部分的二项式队列。最后，我们用“联

结”操作把这个二项式队列重新归并到原来的二项式队列中。

```
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;
  }
```

我们如何联结两个二项式队列呢？首先我们注意到这个操作是很简单的，如果它们不是两个相同大小的二次方堆，如图 9.19 所示：我们简单地把两个二项式队列中的堆归并成一个二项式队列。一个含有 10 个元素的队列（包括一个 8-堆和一个 2-堆）和一个含有 5 个元素的队列（包括一个 4-堆和一个 1-堆）简单地归并成一个含有 15 个元素的队列（包括一个 8-堆、一个 4-堆、一个 2-堆和一个 1-堆）。更一般的情况就像二进制加法一样，完成进位，如图 9.20 所示。

例如，当我们把一个大小为 7 的队列（包括一个 4-堆、一个 2-堆和一个 1-堆）加入到一个大小为 3 的队列（包括一个 2-堆和一个 1-堆）中，我们得到一个大小为 10 的队列（包括一个 8-堆和一个 2-堆）：要完成这个加法，我们需要归并 1-堆得到进位 2-堆，然后归并 2-堆得到进位 4-堆，接着归并 4-堆得到一个 8-堆的结果，在某种意义上精确地相似于二进制加法$(011)_2+(111)_2=(1010)_2$。图 9.19 的例子比图 9.20 的要简单，因为它相似于$(1010)_2+(0101)_2=(1111)_2$，没有进位。

这种用二进制算法的直接类推给了我们一个“联结”操作自然的实现（参看程序 9.16）。对每一个位，要考虑 8 种情况，基于涉及到的 3 位数字（进位和两个操作数中的位）

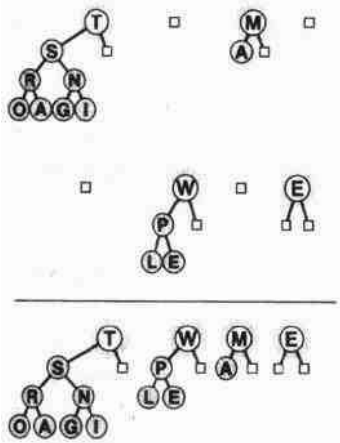



**图 9.19　联结两个二项式队列（无进位）**

当两个要联结的二项式队列没有任何相同大小的二次方堆时，联结只不过是简单的归并。执行这个操作就相似于两个二进制数相加没有遇到 1+1 的情况（没有进位）。这里，一个大小为 10 的二项式队列和一个大小为 5 的二项式队列联结成一个大小为 15 的二项式队列，相应于$(1010)_2+(0101)_2=(1111)_2$。

的所有不同取值。代码比简单的加法要复杂，因为我们要处理不同的堆，而不是相同位数，但每一种情况都是简单的。例如，如果 3 个位都是 1，我们需要在结果的二项式队列中保留一个堆，而把其他两个堆联结成一个进位到下一个位置。实际上，这个操作执行了我们抽象数据类型中的所有操作：我们（仅仅）抵抗住把程序 9.16 发布成一个纯抽象二进制加法过程的诱惑，二项式队列只不过是一个客户程序使用比程序 9.13 更为复杂的位加法过程的实现。

**性质 9.7** 所有的优先队列抽象数据类型的操作都能够用二项式队列实现，从而在 $O(\lg N)$ 的步骤内完成含有 $N$ 个元素的队列的任何操作。

**程序 9.16 联结（归并）两个二项式队列**

这部分代码模仿两个二进制数的加法操作。从右往左处理，初始进位为 0，我们简单明了地处理 8 种可能的情况（操作数和进位的所有可能的值）。例如，情况 3 对应操作数都是 1 而进位是 0。然后，结果是 0，但进位是 1（操作数相加的结果）。

就好像 pair，这个函数是实现的一个私有成员函数，被 getmax 和 join 函数调用。抽象数据类型函数 join(PQ<Ite m>& p)是由调用 BQjoin(bq,p.bq)实现的。

```
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;
        }
  }
```

这些性能界限是数据结构的设计目标。它们是所有操作的实现都是包括二项式队列中树的根节点的一个或两个循环的直接结果。

简单起见，我们的实现在所有的树间循环，从而它们的运行时间是与二项式队列的最大元素个数的对数成比例的。在队列实际大小比队列最大元素个数要小的时候，通过检测队列的大小或者使用哨兵指针来记录循环停止的位置，我们能够使它们符合我们声称的性

能（参看练习 9.61 和练习 9.62）。这样的修改在很多情况下都是不值得的，因为队列的最大元素个数是重复循环所需最多时间的指数倍数。例如，如果我们把最大元素个数设为 $2^{16}$ 而队列通常只有数千个元素，那么我们的简单实现方法就要循环 15 次，而更复杂一些的实现方法很可能仍然需要循环 11 到 12 次，而且这种方法导致额外的维护元素数目或哨兵的耗费。在另一方面，盲目地设置一个巨大的最大元素数目会导致我们的程序在处理小队列时运行得比期望中要慢。■

**性质 9.8**　用 $N$ 个插入操作构造一个大小为 $N$ 的二项式队列在最坏情况下需要 $O(N)$ 次比较。

其中一半的插入操作（当队列的大小为偶数而且没有 1-堆）不需要任何比较操作；剩下的一半插入操作（这时没有 2-堆）只需要 1 次比较操作；当没有 4-堆的时候，只需要 2 次比较操作，以此类推。因而，比较操作的总次数少于 $0*N/2+1*N/4+2*N/8+\ldots<N$。至于性质 9.7，我们也需要用到练习 9.61 和 9.62 中讨论到的改变来保证在最坏情况下线性时间的上限。■

正如在 4.8 节讨论的，我们没有在程序 9.16 的“联结”操作的实现中考虑内存分配，所以它有一个内存泄漏，因而可能在某些情况下失效。为了修正这个缺陷，我们需要对参数的内存分配和实现“联结”操作的函数的返回值给予恰当的注意（参看练习 9.65）。

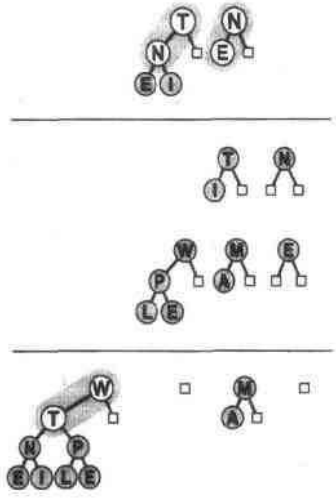


**图 9.20　联结两个二项式队列**

通过一个模仿二进制数加法 $(011)_2+(111)_2=(1010)_2$ 的过程把一个 3 个元素的二项式队列加入到一个 7 个元素的二项式队列中。添加 N 到 E 在结果堆中产生了一个空的 1-堆和一个包括 N 和 E 的进位 2-堆。然后 3 个 2-堆相加在结果堆中产生了一个 2-堆和一个包括 T、N、E、I 的进位 4-堆。这个 4-堆被加入到其他的 4-堆中，产生图的底部的二项式队列。在整个过程里只有少数几个节点涉及操作。

二项式队列提供了快速性能的保证，但数据结构已经被设计得在理论上有更好的性能特性，对一些操作提供常数时间性能的保证。这个问题是数据结构设计中一个有趣而活跃的领域。在另一方面，许多这些深奥的结构的实际效用是可疑的，在我们钻进一些复杂的数据结构的解决方案之前，我们需要搞清楚性能瓶颈存在而且只能够通过减少一些优先队列的操作来解决问题。事实上，对于实际应用程序，我们应该倾向于使用一个基本的结构来排错（debugging）和处理小队列；然后，除非需要高速的“联结”操作，我们应该使用堆来提高操作速度；最后，我们应该使用二项式队列来保证所有操作的对数时间性能。考虑过了所有的事情，无论如何，一个基于二项式队列的优先队列包是软件库的一个有价值的添加。

## 练习

▷ 9.54　画出一个 29 个元素的二项式队列，使用二项式树的表示方式。

● 9.55　写一个程序画出一个二项式队列的二项式树表示，队列的大小为 $N$（只需要画出节点和链接，

不需要关键字）。

9.56 给出一个空二项式队列在插入EASYQUESTION后的结果。

9.57 给出一个空二项式队列在插入EASY后的结果，再给出一个空二项式队列在插入QUESTION后的结果。然后给出在执行“删除最大元素”操作后每个队列的结果。最后，给出两个队列联结后的结果。

9.58 使用练习9.1的约定，给出下列操作后一个初始为空的二项式队列的结果。

PRIO*R**I*T*Y***QUE***U*E

9.59 用练习9.2的约定，给出下列操作后一个初始为空的二项式队列的结果。

(((PRIO*)+(R*IT*Y*))***)+(QUE***U*E)

9.60 证明有$2^n$个节点的二项式树在第$i$层有$\binom{n}{i}$个节点，$0\le i\le n$。（这个事实是二项式树名称的来由。）

○ 9.61 实现性质9.7的二项式队列，通过修改二项式队列数据类型来包含队列的元素个数，然后用队列的元素个数来控制循环。

○ 9.62 实现性质9.7的二项式队列，通过维护标志指针来标记循环停止的位置。

● 9.63 显式使用已经给出的“联结”操作实现二项式队列的插入操作。

●●9.64 实现二项式队列的“改变优先级”和“删除”操作。注意：你将要为节点加上第三个链接。

● 9.65 在课文中的二项式队列的实现（程序9.13到9.16）中加入一个析构函数、一个复制构造函数和一个重载赋值操作符，从而发展练习9.43中一级抽象数据类型的实现，并且编写一个驱动程序来测试你的接口和实现。

● 9.66 凭经验比较作为排序基础的二项式队列和堆，就好像程序9.6一样，关键字是$N=10^3$，$10^4$，$10^5$和$10^6$中的随机数。

● 9.67 好像堆排序那样开发一个根据位置的排序方法，但基于二项式队列。提示：参看练习9.37。

# 第十章

# 基数排序

很多排序应用中，用于声明文件记录顺序的关键字可以是复杂的。例如，电话本或图书馆目录中使用的重关键字。为了将这种文件和我们已经研究过的排序方法的基本特性分离，我们只是对两个关键字进行比较和交换两个数据项（隐藏了这些操作中对关键字进行处理的细节），在第六章到第九章运行的大部分方法中，都将它们作为排序算法和应用程序间的抽象接口程序。本章中，我们会讨论对排序关键字进行的不同处理。例如，通常不需要在每步中都对完整的关键字进行处理，例如要在电话本中寻找某人的电话号码，只需检查名字开头的几个字母，找到包含该号码的页数。要在排序算法中获取类似的效果，我们需要将对关键字进行的比较操作变成将关键字进行分解成一系列固定大小的块或字节。二进制数是一系列比特流，字符串是一系列字母，十进制数是一系列的阿拉伯数字，还有其他类型（不过不是所有的）的关键字可以按这种方法进行分解。每次只对关键字的一个部分进行处理的排序算法称为基数排序。这些算法不仅是对关键字进行比较，它们还对关键字的不同部分进行处理和比较。

基数排序算法将关键字看作是基于-$R$ 的数字系统中表示的数字，$R$（基数）有不同的值，算法将处理这些数中的独立数字。例如，邮局中有一部机器对包裹序列进行处理，包裹上有一个 5 位的十进制数字，机器将包裹分放到 10 个管道中：一个保存数字以 0 开头的包裹，一个保存数字以 1 开头的包裹，一个保存数字以 2 开头的包裹，以此类推。如果必要的话，这些管道可以独立地进行处理，通过对第二个数字进行同样的操作，或者如果只有一些包裹使用另一种更简单的方法进行操作。如果我们按从 0 到 9 的顺序从管道中挑拣包裹，并且在处理后每个管道中的包裹都排好序，我们就可以按顺序提取包裹。该过程是 $R$=10 的基数排序的一个简单例子，而在很多关键字是 5 位到 10 位十进制数的排序应用中可以选择这种方法，如邮政编码、电话号码或者社会保险号。我们会在 10.3 节中详细讨论这个方法。

基数 $R$ 的不同值适用于不同的应用程序。本章中，主要讨论整数类型和字符串类型的关键字，它们可以广泛地使用基数排序。对整数，由于在计算机中它们是用二进制数表示的，我们通常会使用 $R$=2 或其他 2 的幂，因为我们可以将关键字拆分成独立的部分。对于字符串类型的关键字，使用 $R$=128 或者 $R$=256，$R$ 的值根据字节的长度决定。除了这些直

接的声明外，事实上，所有可以在数字计算机中表示的物体都可以表示为二进制数。并且我们可以对许多排序程序进行重新构造，使用另外类型的关键字使基数排序可以对二进制数字进行操作。

基数排序算法基于一个抽象操作“将关键字中第 $i$ 个数字提取出来”。幸运的是，C++提供了低层次的操作函数，可以用直接而高效的方法执行这样的操作。该特性是很显著的，因为其他大部分语言（如 Pascal）鼓励我们编写与机器无关的程序，故意使编写与机器表示数字的方法有关的程序很难实现。在这种语言中，很难执行许多按位操纵技术，而在实际中，这些技术对大部分计算机都是适用的。基数排序一度无法得到实现。但 C 和 C++的设计者意识到直接按位进行操作经常很有用，因此我们可以利用低级语言的工具执行基数排序。

好的硬件设备也是需要的，而且不能将此视为理所当然，一些机器（包括旧的和新的）提供了有效的方法对小数据进行操作，但另外一些机器（包括旧的和新的），如果使用这样的操作，运行效率明显降低。尽管基数排序可以简单地称作抽取数字的操作，但要使基数排序算法达到最好的效率，它和硬件及软件环境是密切相关的。

基数排列中有两种基本的实现方法，这两种方法差别很大。第一种方法从左到右检查关键字的位数，首先处理最重要的数字。这些方法通常被称为 MSD（最高位优先）基数排序。MSD 基数排序是很吸引人的，因为完成工作时，它所需检查的信息量最少（参看图 10.1）。MSD 索引排序和快速排序类似，因为它们都根据关键字开始的几位将文件进行拆分，然后递归地对各子文件进行相同的操作。事实上，当基数为 2 时，MSD 基数排列的执行情况和快速排序类似。第二种基数排序方法就不同：它们按从右到左的顺序检查关键字的位数，首先处理最不重要的数字。这些方法通常被称为 LSD（最低位优先）基数排序。LSD 基数排序有些违反常规，因为它们花费了时间来检查一些不会影响结果的信息，不过很容易对这个问题进行改进，并且在很多排序应用中可以选择这种方法。

.396465048 .015583409 .0
.353336658 .159072306 .1590
.318693642 .159369371 .1593
.015583409 .269971047 .2
.159369371 .318693642 .31
.691004685 .353336658 .35
.899854354 .396465048 .39
.159072306 .538069659 .5
.604144269 .604144269 .60
.269971047 .691004685 .69
.538069659 .899854354 .8

图 10.1 MSD 基数排序

虽然这 11 个 9 位小数（左边）中一共有 99 个数字，但只需检查其中 22 个数字（右边），就可以将它们排好序（中间）。

## 10.1 比特、字节、字

要理解基数排列，就要了解：（$i$）计算机是对一组组比特流进行处理的，称为机器字，通常由一些小块组合而成，这些小块称为字节；（$ii$）排序关键字通常组织成字节序列；（$iii$）小的字节序列同样可以充当数组的基数或者物理地址。因此，使用以下定义进行操作是很简便的。

**定义 10.1** 字节是一个固定长度的比特序列；字符串是一个可变长度的字节序列；字是一个固定长度的字节序列。

在基数排序中，根据实际情况，关键字可以是字或字符串。本章中讨论的一些基数排

列算法是基于固定长度关键字的(字),另一些则是用于关键字的长度不定的情况(字符串)。

通常机器的字节长度是 8 位,而字的长度是 32 位或 64 位(实际位数可以在头文件<limits.h>中查出),不过也可以很方便地将字节和字的位数设为其他值(通常是内在机器位数大小的几倍或几分之一)。我们使用与机器及应用相关的常数来声明每个字的位数和每个字节的位数,例如:

```
const int bitsword = 32;
const int bitsbyte = 8;
const int bytesword = bitsword/bitsbyte;
const int R = 1 << bitsbyte;
```

上述声明中还包括了基数排序中所使用的常数 R,它表示字节的位数,可以有不同的值。使用这些声明时,通常假设 bitsword 是 bitsbyte 的倍数,每个机器字的位数不小于(典型的是等于) bitsword,并且字节可以单独地设定地址。不同的机器有不同的方法查阅它们的比特和字节;我们的讨论中假设字中各位是编了号的,从左到右,从 0 到 bitsword-1,字中各字节也是编了号的,从左到右,从 0 到 bytesword-1。两种情况下,我们都假设编号是按从最重要到最次要的顺序。

大部分机器中都有对位进行的 and 和 shift 操作的功能,我们可利用这些操作从字中提取某些字节。C++中,我们可以直接通过以下操作从二进制字 A 中提取第 B 个字节:

```
inline int digit(long A, int B)
  { return (A >> bitsbyte*(bytesword-B-1)) & (R-1); }
```

例如,通过向右移动 32-3*8 位,可以将 32 位数字中字节 2(第三个字节)提取出来,接着使用掩码 00000000000000000000000011111111 将其他位清为 0,只留下希望读取的字节,右边的八位。

很多机器中另一种选择的方法是进行整理,使基数和字节大小关联,这样只需一个步骤就可以很快地获得所需位数。C++的 C 风格字符串直接支持这种操作:

```
inline int digit(char* A, int B)
  { return A[B]; }
```

如果我们使用了 6.8 节中的结构封装,可以这样编写:

```
inline int digit(Item& A, int B)
  { return A.str[B]; }
```

这种方法也可以用于数字,虽然不同的数字表达方式使得该代码变得复杂。不管是哪种情况,我们都要知道,在一些情况下,这种访问字节的操作可以使用基本的“移动掩盖”操作完成,与前一章中所提过的类似。

我们可以将关键字看作数字,而字节是各阿拉伯数字。对一个数字,基数排序中所需的基本操作是从这个数字中提取出一位。当我们设定基数是 2 的幂时,这些数字由比特组成。就可以容易地使用刚才所提过的语句直接访问。事实上,我们使用 2 的幂数作为基数的根本

原因就是可以容易地读取比特流。在某些硬件环境中，我们也可以使用其他基数。例如，假设 a 是一正数，对基数 R，a 第 b 位数字为：

$$\lfloor a/R^b \rfloor \bmod R$$

适用于高效数字计算操作的计算机中，这种计算对普通 $R$ 的效率和 $R$=2 时的一样。

对于左边包含一个十进制小数点的小数，如图 10.1 所示，就要用另一种方法表示。在这里，$a$ 的第 $b$ 位是

$$\lfloor aR^b \rfloor \bmod R$$

如果使用的计算机可以高效地运行这些操作，我们就可以将它们作为基数排序的基础。该模型可以应用于变长的关键字中，如字符串。

因此，本章将关键字看作是基于 $R$ 的数字（$R$ 没有指定），并使用抽象 digit 操作来读取关键字的各位，并确保在某些计算机中可以快速地执行 digit。

**定义 10.2** *关键字是基数-R 数字，各位数字从左边开始编码（由 0 开始）。*

根据我们刚才讨论的例子，最好假设这种提取操作可以让许多程序在大部分机器上高效的实现，不过我们还是要注意在一个给定的硬件和软件环境中，某一特定类型的程序是比较高效的。

假设关键字的长度都不短，因此值得将各位数字分离。如果关键字较短，我们可以使用第六章中的关键字索引统计方法。该方法可以在线性时间中对 $N$ 个介于 0 和 $R$-1 的关键字进行排序，需要使用一个大小为 $R$ 的辅助表来存放统计数，和另一个大小为 $N$ 的表来重新整理记录。因此，如果我们有足够的空间存放 $2^w$ 大小的表格，就可以容易地在线性时间内对 $w$ 位关键字进行排序。事实上，索引统计排序使用的基本思路和基本 MSD 及 LSD 基数排序方法一样。当关键字相当长（例如 $w$=64），没有足够的空间使用大小为 $2^w$ 的表格时，基数排序就很有效了。

## 练习

▷ 10.1 将 32 位内容看作是基数-256 数字时，要使用多少位？描述如何提取出各位。同样对基数 216 也进行考虑。

▷ 10.2 $N$=$10^3$，$10^6$ 和 $10^9$，能对介于 0 和 $N$ 的所有数字用 4 字节字表示时，所需的最小字节长度是什么？

○ 10.3 重载操作符<进行 digit 提取操作（这样，我们就可以将第六章和第九章的算法和本章的算法进行比较）。

○ 10.4 设计并执行一个程序，比较使用位移动和算术函数进行提取位数操作时的花费。对两种算法，每秒可以提取多少位数字？注意：编译器可能会将算术操作转换成位移动操作，或者相反！

● 10.5 编写一个程序，对 $N$ 个十进制小数（$R$=10）进行排序，统计排序过程中所要进行的位比较操作的总数，按照图 10.1 示例的那样。对 $N$=$10^3$，$10^5$，$10^6$ 进行操作。

● 10.6 如果 $R$=2，使用 32 位随机数，对练习 10.5 所得的程序进行修改。

● 10.7 当数字按高斯序列分布时，对练习 10.5 的程序进行修改。

## 10.2 二进制快速排序

假设，对文件的记录我们可以进行重新整理，使所有以 0 开头的关键字比以 1 开头的关键字要小。这样，我们可以使用一种递归排序方法，是快速排序（参看第七章）的另一个版本：将文件拆分，接着对两个子文件单独进行排序。为了重新整理文件，从左边开始寻找以 1 开头的关键字，从右边开始寻找以 0 开头的关键字，交换，直到两个扫描指针交错了。这种算法在文献中（包括本书较早的版本）通常称为基数交换排序；这里，我们使用二进制快速排序这个名字来强调它只是 Hoare 提出的算法的一个简单修改版本，虽然它发现的比快速排序还要早（参见“第三部分参考资料”）。

程序 10.1 是这种方法的完整执行过程。划分文件的操作和程序 7.2 的完全相同，除了使用了数字 $2^b$ 作为划分元素，而不是使用文件中的一些关键字。因为文件中可能不包含 $2^b$，因此在拆分过程中不能保证某个元素已被放置在合适的位置。该算法和普通快速排序的不同之处还包括，它是对位数超过 1 的关键字进行递归调用的。这点不同在实际应用中是很重要的。例如，当对一个包含 $N$ 个元素的文件进行拆分时，对位数大于 1 的关键字，就会进行对大小为 $N$ 的子文件进行的递归调用。因此，这种调用操作的数目受关键字的位数限制。相反，如果在标准快速排序中也使用不在文件中的值作为割分值，就会导致无穷的递归循环。

和标准的快速排序一样，有很多可以选择的内部循环执行语句。在程序 10.1 中，两个内部循环中都包括了判断指针是否交错的测试语句。这样做，当 $i=j$ 时，需要进行额外的交换操作，这个可以通过 break 语句避免，如程序 7.2 中所作的，虽然这里将 a[i]和自己进行交换并没什么影响。另一个可选择的方法是使用标志关键字。

**程序 10.1 二进制快速排序**

本程序按关键字开始的位数对文件进行拆分，接着递归地对子文件进行排序。变量 d 跟踪正被检查的位数，从 0 开始（最左边）。当 j=i 时，该拆分工作结束，这时，a[i]右边所有数据项第 d 位的值是 1，而 a[i]左边所有数据项第 d 位的值是 0。数据项 a[i]第 d 位的值是 1，除非文件中所有关键字第 d 位的值都是 0。这里，拆分循环后面紧跟着一个测试操作。

```
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); }
```

图 10.2 描述了程序 10.1 对小文件的执行情况，跟图 7.1 的快速排序进行比较。该图显示了数据的移动，但没有显示不同的移动进行的原因——这点和关键字的二进制表示法有关。图 10.3 将对同一个例子进行更详细的讨论。这个例子假设字母是由一个 5 位代码表示的，字母表中第 $i$ 个字母表示为二进制数字 $i$。这种编码是实际字符编码的一个简化版，在实际中，使用更多的位数（7、8，甚至 16）表示更多的字符（大小写字母、数字和特殊的符号）。

对包含随机位数的字键，程序 10.1 的开始指针必须指向字的最左位，或者位 0。通常，开始指针是根据实际情况设定的，与硬件中每个字的位数及整数和负数的表示方法有关。对于图 10.2 和 10.3 中的 5 位关键字，在 32 位机器中，开始指针将是位 27。

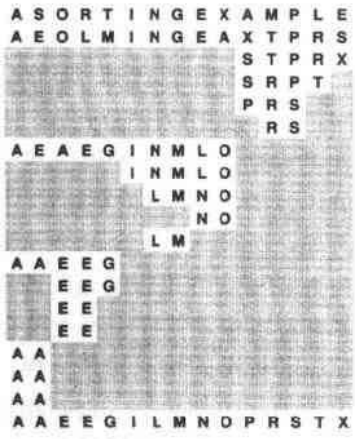



**图 10.2 二进制快速排序例子**

按最开始位数进行的拆分操作不能保证任何数据项被放到了合适位置中；只能保证所有关键字以 0 开头的数据项排在所有关键字以 1 开头的数据项之前。我们可以将这个图与快速排序中的图 7.1 进行比较，虽然不知道关键字的二进制表示法时拆分操作也是不清晰的。图 10.3 更详细地解释了进行拆分操作的位置。

这个例子提出了二进制快速排序在实际应用中的一个潜在问题：退化的划分（划分的所有关键字中，正在讨论的所有位的值都相等）操作可能会频繁发生。在关键字的构成字符间也会产生同样的问题，例如，将四个字符进行 8 位编码，接着将它们放在一起就组成了 32 位关键字。这样，在每个字符的开始处都可能会产生退化的划分，因为，例如，在大部分字符编码中，所有小写字母都使用同样的开始位值。这个问题使我们在对编码数据进行排序时要进行地址编码，在其他基数排序中也有类似的问题。

当一个关键字通过左边几位可以和其他关键字进行区分后，就不需要在检查更多位数的值了。该特性在某些情况下是优点，而在其他情况则是缺点。当关键字是完全随机数时，对每个关键字只需检查 $\lg N$ 位值，这个可以比关键字中的位数要小的多。在 10.6 节中会讨论这个观点，也可以参看练习 10.5 和图 10.1。例如，对一个具有 1000 个数据项的随机文件进行排序，可能只需对每个关键字检查 10 或 11 位的值（即使这些是 64 位的关键字）。另一方面，相同的关键字间要检查所有位的值。基数排序不适用于那些包含许多重复的长关键字的文件。对关键字完全由随机值组成的文件，二进制快速排序和标准快速排序的运

行效率都较高（两者的区别由抽取位的操作和比较操作的效率决定），但对非随机的关键字值，标准的快速排序算法的效率要较高，当由大量重复关键字时适合使用二路快速排序。

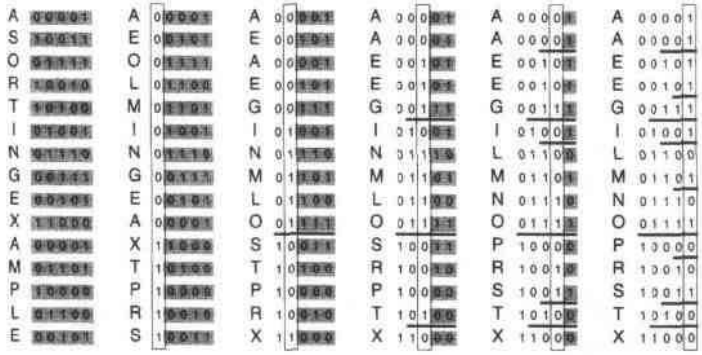


**图 10.3　二进制快速排序例子（显示关键字的各位）**

将关键字转换成二进制编码，就可以将图 10.2 改画成图 10.3，对表格进行压缩以显示各子文件的独立排序操作，并将行和列调换。第一步中，将文件拆分成两个子文件，一个所有的关键字以 0 开头，另一个以 1 开头。接着第一个子序列拆分成一个以 00 开头的子文件和另一个以 01 开头的子文件；对应的，运行的某个时候，将另一个子文件拆分成一个以 10 开头的子文件和另一个以 11 开头的子文件。当所有的位数都处理后（如对重复关键字），或者到子文件的大小为 1，就停止该过程。

和快速排序一样，可以很方便地用二进制树描述划分操作（如图 10.4 所示）：根节点对应于一个要进行排序的子文件，它的两个子树对应于划分后的两个子文件。在标准快速排序中，我们知道划分过程中至少将一个记录放到了合适的位置中，因此将那个关键字作为根节点；在二进制快速排序中，只有当子文件的大小为 1 或者已经遍历了关键字中各位的值后，才能知道关键字放在了合适位置，所以将关键字放在树的底部。这样的结构称为二进制树——第十五章中将详细讨论这些树的特性。其中一个重要的特性是，树的结构完全由关键字的值决定，而不是它们的顺序。

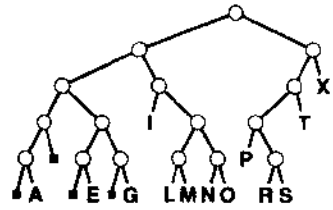



**图 10.4　二进制快速排序划分树**

该树描述了与图 10.2 和图 10.3 对应的二进制快速排序划分结构。因为没有数据项一定在合适的位置，关键字与树的末节点对应。该结构有以下特性：从根节点开始访问到关键字，0 代表左分支，1 代表右分支，给出了关键字的前导位。这是排序过程中该关键字与其他关键字进行区分的位。黑色的小方块表示空的划分部分（如果所有关键字的前导位都相同，就会出现在树的同一边）。本例中，这种情况只发生在接近树的底部，不过可能发生在树的较高位置；例如，如果关键字中不包含 I 或者 X，它们的节点就会被一个空节点代替。注意，重复的关键字（A 和 E）不可以被划分（当检查了所有的位后，排序操作将它们放到同一个子文件中）。

二进制快速排序的划分分界线由数据项的范围及数据项的数目的二进制表示法决定。例如，如果文件是小于 $171=10101011_2$ 的随机整数序列，第一位的划分大概就是 128，因此子文件是不相等的（一个大小为 128，一个大小为 43）。图 10.5 中的关键字是随机的 8 位值，所以不会有这样的情况发生，但我们要知道在实际应用中有这样的情况。

通过消除递归和一些特殊处理，可以提高程序 10.1 的基本递归执行效率，跟我们在第七章对标准快速排序进行的处理类似。

**练习**

△ 10.8 对序列E A S Y Q U E S T I O N进行基数快速排序，按图 10.2 的方法，画出相应于划分步骤的线索。

10.9 对 3 位二进制数字文件：001，011，101，110，000，001，010，111，110，010，比较二进制快速排序中使用的交换操作数目和普通快速排序中使用的数目。

○ 10.10 为什么在二进制快速排序中不像普通的快速排序那样，强调先对两个子文件中较小的那个进行操作？

○ 10.11 使用二进制快速排序法对一个随机的数据项小于 171 的非负整数文件进行操作，描述第二层的划分操作的情况（当对左边子文件进行划分时和对右边子文件进行划分时）。

10.12 编写一个程序，在预处理过程中，鉴别直到哪位所有关键字的位值都相等的。接着调用一个忽略那些位的二进制快速排序操作。设输入数据是 32 位字，结构如下：最右边 16 位是随机的，最左边 16 位全都是 0，除了第 $i$ 位为 $i$，$i$ 是右边部分的 1 的数目，对 $N=10^3$，$10^4$，$10^5$和 $10^6$，将该程序的运行时间和使用标准二进制快速排序程序的运行时间进行比较。

10.13 修改二进制快速排序，检查是否所有的关键字都相等。使用练习 10.12 中的数据类型，对 $N=10^3$，$10^4$，$10^5$和 $10^6$，将该程序的运行时间和使用标准二进制快速排序程序的运行时间进行比较。

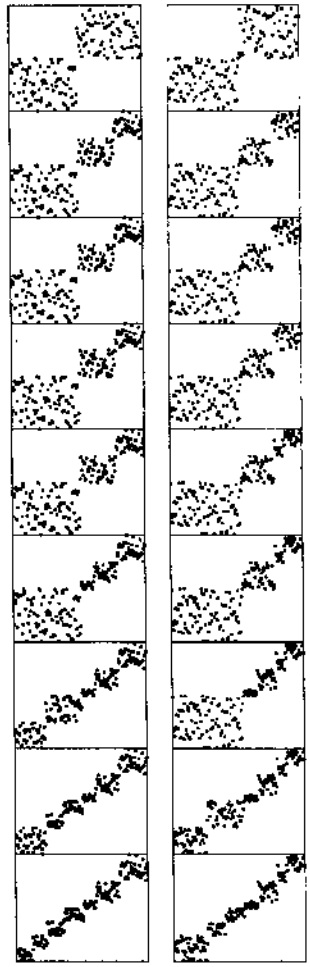
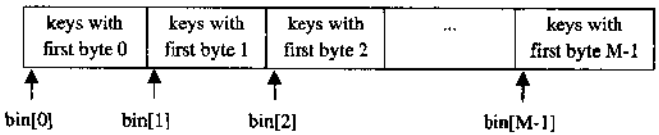


**图 10.5 二进制快速排序对大文件进行操作时的动态特性**

在二进制快速排序中进行的划分操作和关键字的顺序的关系，没有标准快速排序中的那么强烈。这里，两个不同的随机 8 位文件的划分情况几乎一样。

## 10.3 MSD 基数排序

将关键字当作是基数-2（二进制）数字，在基数排序中只使用 1 位，并且先考虑最重要的位。概括来说，就是通过先比较最重要的位来完成对基数-$R$ 数字的排序。这样做，需要将数组拆分成 $R$ 个不同部分，而不只是两部分。传统上，使用接收器（bin）或者盒（bucket）表示划分的部分，并将运算看作使用一组 $R$ 接收器，每个包含了第一位的一种可能值，如下面图所示的：



我们对关键字进行访问，将它们分配到相应的接收器中，接着递归地对接收器中的内容进行排序。

图 10.6 显示了一个对随机整数序列进行 MSD 基数排列的例子。和二进制快速排列不同，该算法可以很快地将一个基本有序的文件排好序，甚至是在第一个划分中，如果基数够大的话。

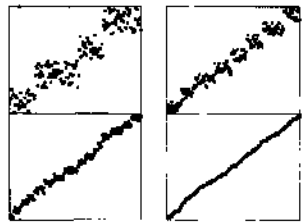


图 10.6 MSD 基数排序的动态特性

MSD 基数排序可能只需一步就能基本完成整个排序任务，如这里显示的随机 8 位整数的例子。MSD 排序的第一步，考虑最先的 2 位（左边），将文件划分成 4 个子文件。接下的一步又将每个子文件划分成 4 个子文件。3 位的 MSD 排序（右边）在一步中就将文件划分成 8 个子文件。第二步中，每个子文件又被划分成 8 部分，每个只包含几个数据。

如 10.2 节所提到的，基数排序最有用的特性之一是，可以对字符串类型的关键字进行快速的操作。这点可以在 C++ 或其他支持字符串操作的编程环境中实现。对 MSD 基数排序，只需简单地使用一个与字节大小相关的基数。要提取某一位，就读入字节；要移到下一位，就将指针加一。这里，我们对固定长度的关键字进行讨论；以后我们会知道对变长的字符串关键字，可以很容易地使用同一个基本算法进行处理。

图 10.7 显示了对包含 3 个字母的单词进行 MSD 基数排序的例子。为了简便，该图假设基数为 26，而在大部分应用程序中会根据字符的编码，使用一个更大的基数。首先，对单词进行划分，使以 a 开头的单词在以 b 开头的单词前，等等。接着，对以 a 开头的单词进行递归排序，接着对以 b 开头的单词进行排序，以此类推。如本例所表明的，排序中大部分工作依赖于对第一个字母进行的划分；从第一个划分得到的子文件已经很小。

如在第七章和 10.2 节提到的快速排序，及在第 8 章提到的归并排序那样，我们可以通过使用一些小算法对递归程序进行改进。在基数排序中对小的子文件（包含小数目元素的接收器）使用另一种不同的排序方法，是非常有效的，因为有很多这样的小文件。另一方面，通过调整 R 的值也可以对算法进行改进，因为：如果 $R$ 太大，初始化和检查接收器的消耗会很大；如果 $R$ 太小，就不能很好地利用将文件分得越细越精确的潜在效率。在本节最后和 10.6 节中我们会接着讨论这个问题。

要执行 MSD 基数排序，需要对数组的划分方式进行归纳，与第七章的快速排序相联系。这些方法是基于指针的，指针从数组的两端开始，在数组中间相遇；在只有两个或三个划分的情况下，这些方法很适用，但不能直接归纳出来。幸好，第六章的关键字索引统计方法适用于关键字值在一个小范围的文件，它能很好地满足我们的要求。我们使用了一个统计表格和一个辅助数组；对数组进行的第一次遍历中，对每个最主要的位值进行统计。这些统计数告诉我们在哪里进行划分。接着，对数组的第二次遍历中，使用这些统计数将数据项移到辅助数组中的合适位置中。

程序 10.2 执行了这个方法。它的递归结构和快速排序的相同。因此在 7.3 节讨论的问题也需要在这里进行讨论。为了避免过分的递归深度，是否需要最后处理最大的子文件？当然是不需要的，因为递归的深度由关键字的长度限制。是否应使用简单的方法如插入排序对小的子文件进行排序？当然需要，因为有大量这样的小文件。

为了进行拆分，程序 10.2 使用了一个和待排序数组的大小一样的辅助数组。同样，我们可以选择使用合适的关键字索引统计（参看练习 10.17 和 10.18）。我们要特别注意空间问题，因为递归调用可能会使用过多的空间存放局部变量。在程序 10.2 中，为了移动关键字（aux）所使用的临时缓冲区可能会很大，但保存统计数和拆分位置（count）的数组必须是局部的。

**程序 10.2 MSD 基数排序**

这个程序是从程序 8.17（关键字索引统计排序）中派生的，将对关键字引用操作改为对关键字各位进行操作，在最后添加一个 loop 循环，确保每个子文件中的关键字在递归调用中都从同一位开始。对变长的以 0 结束的关键字（如 C 风格的字符串），省略第一个 if 语句和第一个递归调用。该执行程序使用了一个足够大的可以保存输入文件副本的辅助数组（aux）。

```
#define bin(A) l+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[count[digit(a[i], d)]++] = a[i];
    for (i = l; i <= r; i++) a[i] = aux[i-l];
    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);
  }
```

在很多实际应用中，包含较长的关键字和记录的基数排序所使用的辅助数组的额外内存空间并不是主要问题，因为对这样的数据，通常会使用指针排序。因此，额外的空间是用于调整指针的，和关键字及记录项的大小相比指针较小（但这个问题也不是毫无意义的）。如果空间上允许，而且运行效率很重要（在使用基数排序时的通常情况），我们也可以通过递归参数转换来消除数组复制的开销，和 10.4 节中对合并排序的处理类似。

对于随机关键字，第一步后，每个接收器中的关键字的数目（子文件的大小）平均为 $N/R$ 个。实际中，关键字可能不是随机的（如，当关键字是英语单词字符串时，我们知道只有一些是 x 开头的，而没有单词是 xx 开头的），因此，有很多接收器是空的，而非空的接收器中，有些的关键字的数目比另外一些的要多得多（参看图 10.8）。虽然有这样的影响，

多步拆分操作还是能有效地将大文件分成多个小文件进行排序。

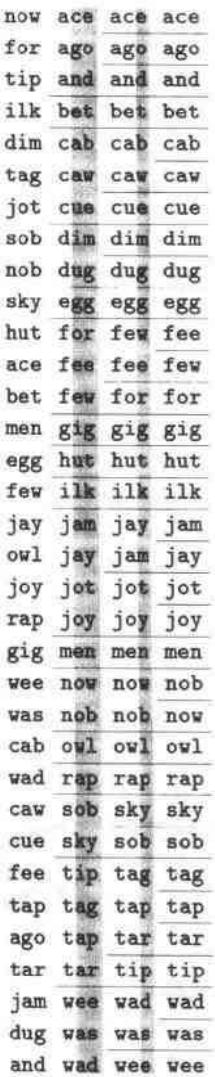



**图 10.7 MSD 基数排序例子**

根据第一个字母，将单词划分到 26 个接收器中。接着，用同样的方法对所有的接收器内容进行排序，从第二个字母开始。

**图 10.8 MSD 基数排序例子（包含空接收器）**

对小的文件，即使是在第一步，也会遇到太多的空接收器。

另一个执行 MSD 基数排序的方法是使用链表。对每个接收器使用一个指针：在对数据项进行排序的第一步中，根据开始位的值将每个数据项插入到适当的链表中。接着，对子序列进行排序，最后将所有的链表组合在一起得到完整的排好序的序列。这个方法带来了一个有挑战性的编程练习（参看练习 10.36）。要将所有的链表组合在一起，需要知道链

表的开始位置和结束位置，而且，当然许多链表都是空的。

要使基数排序在某个特定程序中有好的效率，需要通过挑选合适的基数和对小文件的中断数，来减少空接收器的数目。例如，假设对 $2^{24}$（大概 1600 万）个 64 位整数进行排序。要使统计表相对于文件较小，可以选择基数 $R=2^{16}$，相应地对关键字的 16 个位进行检查。但在第一次划分后，子文件的平均大小仅是 $2^8$，对这样的文件使用 $2^{16}$ 基数太过度了。最坏的是，可能有很多这样的文件，本例中大概有 $2^{16}$ 个。对应于这 $2^{16}$ 个文件中的每个文件，排序过程会产生 $2^{16}$ 个空的统计数，而只有 $2^8$ 个是非空的，等等，因此至少需要消耗 $2^{32}$ 次算术操作。程序 10.2 执行时假设大部分接收器是非空的，因此对每个空接收器执行了许多算法操作（如，对所有的空接收器都执行递归调用），所以在该例中，运行时间会很巨大。对第一级来说，更合适的基数可以是 $2^8$ 或 $2^4$。简而言之，在 MSD 基数排序中，我们要确保不对小文件使用大基数。在 10.6 节，当认真检查不同方法的特性的时候，我们会详细讨论这个观点。

如果我们设置 $R=256$，并消除对空接收器进行的递归调用，那么程序 10.2 就能有效地对 C 风格字符串进行排序。如果我们知道所有字符串的长度比某个固定长度要小，我们可以将变量 bytesword 设为该长度，或者可以消除对标准变长的字符串进行排序时对 bytesword 的检测操作。对于字符串，我们通常可以执行位提取操作，像单个数组那样，如我们在 10.1 节讨论的。通过调整 $R$ 和 bytesword（并对它们的值进行检测），我们可以修改程序 10.2，使它可用于处理包含非标准字母或格式不标准及有限制长度或其他约定的字符串。

字符串排序再次说明了正确处理空接收器的重要性。图 10.8 显示了与图 10.7 类似的划分过程，不过清晰地显示了两个字母的单词和空的接收器。本例中，使用基数 26 对两个字母的单词进行基数排序，因此每步中有 26 个接收器。在第一个接收器中，没有太多的空接收器，然而，在第二步中，大部分接收器是空的。

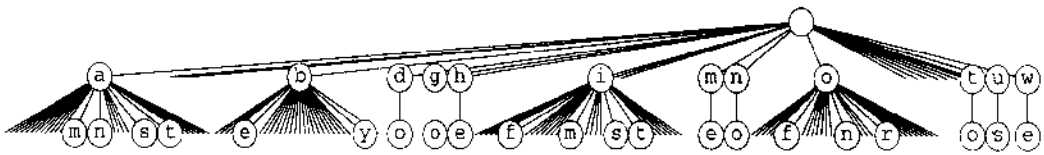



**图 10.9 MSD 基数排序的递归结构**

本对对应于使用程序 10.2 的方法对图 10.8 的例子进行 MSD 基数排序的递归操作过程。如果文件的大小是 0 或 1，就没有任何递归调用。否则，有 26 个调用：每个对应于当前字节的一个可能的值。

MSD 基数排序函数按关键字的第一位对文件进行划分，接着根据各个值对子文件进行递归调用。图 10.9 显示了图 10.8 中的例子的 MSD 基数排序的递归调用结构。该调用结构和一个多路树对应，和图 10.4 中二进制快速排序的树结构类似。每个节点对应于 MSD 排序中某个子文件的递归调用。例如，节点标志为 o 的根子树对应于对包含 of、on 和 or 三个关键字的子文件进行的排序操作。

这些图表明：在对字符串进行 MSD 排序时，会产生大量的空接收器。在 10.4 节中，我们会研究一种解决方法；在第十五章，我们会仔细研究字符串处理程序中的树结构的用

途。通常，我们都对数结构进行简化，不包括与空接收器对应的节点，将标志从边缘移到下端的节点，如图 10.10 所示，与图 10.7 中三个字母的 MSD 基数排序的递归调用结构（忽视空的接收器）相对应。例如，节点标志为 j 的根子树对应于对包含四个关键字 jam、jay、jot、joy 的接收器进行的排序操作。我们会在第十五章中详细讨论这种树结构的特性。

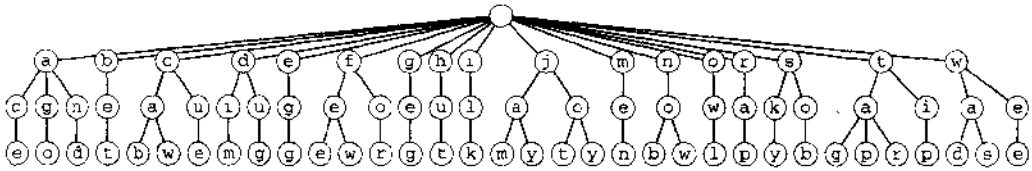


**图 10.10 MSD 基数排序的递归结构（忽略空的子文件）**

这里的 MSD 基数排序递归结构比图 10.9 的更复杂。树中的每个节点标志了某个关键字第 $i$-1 位的值，$i$ 是从节点到根的距离。树中每个从根到底部的路径代表了一个关键字；将路径中节点的标志组合起来就形成了关键字。这棵树和图 10.7 中的三个的字母 MSD 排序示例对应。

如何提高对非随机的长字符串数据进行 MSD 基数排序的效率，是一个重要问题。特别是，关键字中包含长的相等的或不需要的数据，或者它们中的一部分局限在某个小范围中。例如，在一个学生数据信息处理程序中，会包含与毕业年份（4 个字节，但是四个不同值之一）、州的名字（可能是 10 个子节，不过是五十个不同值之一）、性别（1 个字节，二个给定值之一）以及姓名（比较像随机字符串，但可能比较长，不规则的字母分布，在固定长度会有空格）相对应的关键字。所有这些不同的限制使进行 MSD 基数排列时有大量的空接收器（参看练习 10.23）。

解决这个问题的一种方法是，编写一个更复杂的提取操作程序，将所有字符串排序情况的信息都考虑进去。另一个容易实现的方法是，在统计过程中，了解非空接收器范围的最高值和最低值，然后只使用在此范围的接收器（可能还包括一些特殊的情况，如 0 或空值），该方法称为接收器-范围法（bin-span heuristic）。这样的整理对以前提到的例子很适用。如，对基数-256 的文字数字数据，关键字中有一部分是数字，与各位对应的非空接收器只有 10 个，对关键字另一部分的大写字母，只有 26 个非空接收器与它们对应。

执行接收器-范围法有多种可选择的方法（参见“第三部分参考资料”）。例如，我们可以将非空的接收器保存在一个附加的数据结构中，只对这些接收器进行统计及递归调用。然而，这样做（甚至包括接收器-范围法）在这种情况下可能是太过度了，因为除非基数很巨大或者文件很小，否则所节省的消耗是很少的，而对这两种情况，我们可以通过使用一个较小的基数或使用另一种排序方法来提高效率。我们所能节省的消耗和通过调整基数或对小文件转换排序方法的操作所节省的消耗是一样的，但实现起来没有这两种方法简单。在 10.4 节中，我们会讨论另一个版本的快速排序，该方法能有效地解决空接收器的问题。

## 练习

▷ 10.14 画出与图 10.9 对应的简洁树结构（没有空的接收器，节点中显示关键字，如图 10.10）。

▷ 10.15 和图 10.10 对应的树中共有多少个节点？

▷ 10.16 显示使用 MSD 基数排序对序列 now is the time for all good people to come the aid of their party 进行操作时的划分情况。

● 10.17 编写一个程序，执行四路划分，对每个关键字的出现频率进行统计，如关键字索引统计中所作的那样，接着使用与程序 6.14 类似的方法移动关键字。

●●10.18 编写一个程序解决 $R$ 路划分程序中的一般问题，使用练习 10.17 中提出的方法。

○ 10.19 编写一个程序随机产生 80 字节的关键字。使用该程序产生 N 个随机关键字，接着使用 MSD 基数排序对它们进行排序。$N=10^3$，$10^4$，$10^5$ 和 $10^6$。输出每次排序中所要检查的关键字总字节数。

○ 10.20 练习 10.19 中，对每个给定的 $N$ 值，在程序中所会访问到的最右的位数是多少？如果已经作了该练习，修改程序对这个数目进行记录，将理论上的结果和实际的结果进行比较。

10.21 编写一个程序，通过将随机产生的 80 字节序列打乱来产生关键字。使用该程序产生 $N$ 个随机关键字，接着使用 MSD 基数排序对它们进行排序。$N=10^3$，$10^4$，$10^5$ 和 $10^6$。将得到的结果和随机情况下的进行比较（参看 10.19）。

10.22 练习 10.21 中，对每个给定的 $N$ 值，在程序中所会访问到的最右的位数是多少？如果已经作了该练习，将理论上的结果和实际的结果进行比较。

10.23 编写一个程序产生 30 字节的随机字符串，这些字符串由三部分组成：一个 4 字节的域，值是 10 个给定字符串之一；一个 10 字节的域，值是 50 个给定字符串之一；一个单字节的域，值是 2 个给定值之一；而剩下的 15 个字节保存随机的左对称字符串，可以看作是第 4 个域。使用该程序产生 N 个随机关键字，接着使用 MSD 基数排序对它们进行排序。$N=10^3$，$10^4$，$10^5$ 和 $10^6$。输出所检查的总的字节数。将得到的结果和使用随机数的结果进行比较（参看练习 10.19）。

10.24 将程序 10.2 进行修改，执行接收器-范围算法。用练习 10.23 中的数据对程序进行检测。

```
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 caw
jot and and cue
sob fee egg egg
nob cue cue dug
sky caw caw 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
wee wee now owl
was was nob nob
cab men men now
wad wad rap
caw 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 wee wee
jam rap wad wad
```

**图 10.11 三路基数快速排序**

将文件划分成 3 个部分：开始字母是从 a 到 I 的单词、从 j 开始的单词及从 k 到 z 的单词。接着递归地执行排序操作。

## 10.4 三路基数快速排序

另一种用快速排序进行 MSD 基数排序的方法是，对关键字前导位使用三路划分，只在中间子文件移动到下一个字节（关键字的前导位与划分元素的相同）。这种方法很容易实

现（一句声明语句加上程序 7.5 中的三路划分代码就可以实现了），而它适用于不同的环境。程序 10.3 是该方法的一个完整执行过程。

**程序 10.3　三路基数快速排序**

这个 MSD 基数排序的代码和三路划分的快速排序的基本相同（程序 9.5），不过有以下变化：（*i*）对关键字进行操作变成对关键字的各位进行操作；（*ii*）当前字节作为参数加入到递归序列中；（*iii*）对中间子文件的递归调用移到下一个字节。在进行移到下一个字节的递归调用之前，通过检测划分值是否为 0 来避免移动越过了字符串的结尾。当划分值为 0 时，左子文件为空，中间子文件对应于程序已找到的相同关键字的个数，右边子文件对应于那些较长的需要进一步处理的字符串。

```
#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);
  }
```

根本上，三路基数快速排序包括根据关键字的开始字符对文件进行排序（适用快速排序），接着对剩下的关键字递归地执行本方法。对字符串进行排序时，该方法的效率类似于普通的快速排序和 MSD 基数排序。事实上，它可以看作是两个算法的混合物。

将三路基数快速排序和标准的 MSD 基数排序进行比较，该方法只将文件分成三部分，因此它不具备快速多路划分的益处，特别是在排序开始几步中。相反，在迟一点的步骤中，MSD 基数排序包含了大量的空接收器，而三路基数快速排序很适用于重复关键字、小范围关键字、小文件及其他 MSD 基数排序运行较慢的情况。更重要的是，划分操作适用于非随机关键字中。而且不需使用辅助数组。但缺点是，当子文件数目较大时，需要额外的交换操作来通过三路划分获得多路划分的效果。

图 10.11 显示了使用该方法对图 10.7 的 3 字母单词排序问题进行操作的示例。图 10.12 显示了递归调用结构。每个节点对应于三个递归调用：第一位较小的关键字（左子树），第一位相等的关键字（中间子树），第一位较大的关键字（右子树）。

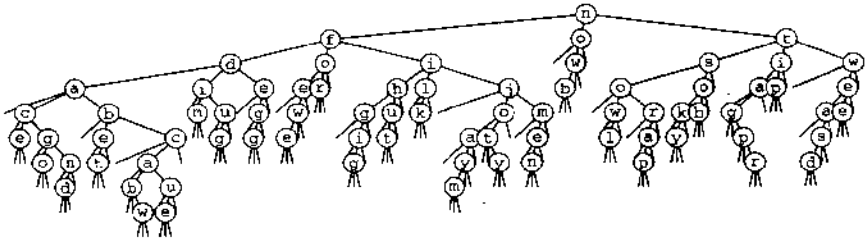


**图 10.12 三路基数快速排序的递归结构**

本树联合了图 10.10 树的 26 路节点及二重二进制搜查树，如图 10.13 所示。从树的根部开始到树的底端并以中间链接结束的路径对应于文件的一个关键字。图 10.10 包含 1035 个没有描述的空连接；而本树中显示了所有的 155 个空连接。每个空指针对应一个空的接收器，因此两图的差别表明了三路划分算法如何大幅度地减少 MSD 基数排序中的空接收器数目。

当排序关键字与 10.2 中的提取数符合时，标准的快速排序（及第六章到第九章的其他排序程序）可以看作是 MSD 基数排序，因为比较操作首先访问关键字的最重要部分（参看练习 10.3）。

例如，如果关键字是字符串，关键字不同时，基本快速排序比较操作只需访问前面几位，如果第一位相等就判断第二位是否相等，以此类推。因此，标准的算法可以自动获得一些与 MSD 基数排序相同的益处（参看第 7.7 节）。最重要的区别是，当头几位相等时，标准算法不能采取一些特殊的相应措施。事实上，程序 10.3 可以看作是某种快速排序算法，在进行多重划分后，保存数据项前导位数的相关资料。在小的子文件中，在执行了排序中的大部分比较操作后，关键字可能有很多相等的前导位值。标准算法每个比较操作中，都必须访问所有的字节；三路算法则不需这样做。

对关键字较长（简化起见，假设定长）但大部分主要的字节完全相等的情况进行考虑。对这种情况，普通快速排序的运行时间和字长度乘以 $2N\ln N$ 的积成比例，然而基数排序的运行时间和字长乘以 $N$（发现所有重要的相等字节）加上 $2N\ln N$（对剩余的短关键字进行排序）的总和成正比。也就是，这种方法可以比普通的快速排序快 $\ln N$，正是比较操作的总和。实际操作中，也有很多例子具有相似的特点（参看练习 10.25）。

三路基数快速排序的另一个有趣特性是它和基数大小没有直接关系。对其他基数排序

方法来说，需要一个大小由基数值决定的辅助数组，而且我们需要确保数组的大小不会远远大于文件大小。对于本方法，不需要额外的表格，采用非常大的基数（大于字的大小）使该方法退化成普通的快速排序，而采用基数 2 使它退化成二进制快速排序，而中间大小的基数可以有效地处理具有重复关键字的情况。

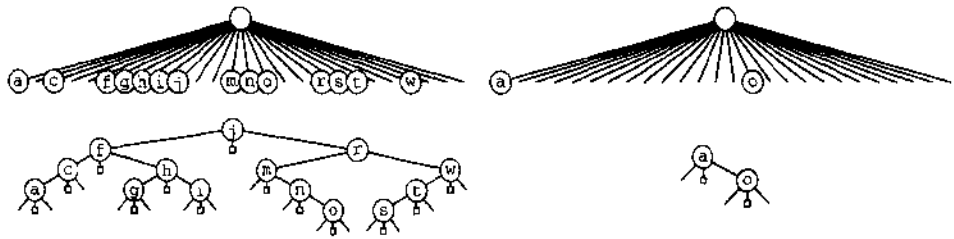


**图 10.13　三路基数排序的树节点示例**

通过三路划分减少一个字节和递归地对其他位进行操作，来解决 MSD 基数排列的空接收器问题。将每个描述 MSD 基数排序递归结构的 M-路节点（参看图 10.9）相应地替换成每个非空接收器对应有一个内部节点的三重树。对于完整的树（左边），这样的改变不能节省空间却会消耗时间；但对空树（右边），减少了消耗的时间和所用的空间。

在许多应用中，我们可以执行一个混合的方法以获得较高的效率，如对大文件，利用多路划分的优点，使用标准的 MSD 基数排序，对较小的文件，为了避免大量空接收器的不良影响，使用三路基数快速排序。

三路基数快速排序同样适用于向量关键字中（不管是数学上的还是 C++标准库中的）。也就是说，如果关键字由独立的部分（每个是一个抽象关键字）构成，我们希望关键字先是根据第一个部分排序，如果第一部分相等就根据关键字的第二部分排序，如此类推。我们可以将矢量排序看作是基数排序的概括，R 值任意大。当对程序 10.3 进行修改以适应这种情况的时候，这个程序被称为多重关键字快速排序。

### 练习

10.25　对 $d>4$，假设关键字包含 $d$ 个字节，最后四个字节是随机值，其他的值都为 0。对大小为 $N$ 的大文件进行三路基数快速排序（程序 10.3）和普通的快速排序（程序 7.1），估计要检查的字节总数，并计算运行时间的比率。

10.26　对随机的 64 位关键字且 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，进行三路基数快速排序，字节的大小为什么时，运行时间最短？

● 10.27　编写对链表进行的三路基数快速排序的执行程序。

10.28　设关键字是 t 浮点数向量，编写一个多重快速排序的执行程序，使用练习 4.6 描述的浮点数检测操作。

10.29　使用练习 10.19 的程序产生关键字，对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，运行三路基数快速排序。将它的执行情况和 MSD 基数排序的进行比较。

10.30　使用练习 10.21 的程序产生关键字，对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，运行三路基数快速排序。将它的执行情况和 MSD 基数排序的进行比较。

10.31 使用练习 10.23 的程序产生关键字，对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，运行三路基数快速排序。将它的执行情况和 MSD 基数排序的进行比较。

## 10.5 LSD 基数排序

另一种基数排序方法是从右到左检查字节。图 10.14 显示了如何在 3 步中对 3 个字母的单词进行排序。根据最后的字母对文件进行排序（使用关键字索引统计），接着根据中间字母进行排序，最后根据第一个字母进行排序。

一开始，不容易确信这个方法可行；事实上，只有当使用的排序方法稳定的时候（参看定义 6.1），这个方法才有效。一旦稳定性得到确保，就可以很容易地证明出 LSD 基数排序有效：在将关键字的 $i$ 个尾字节排好序后（以一种稳定方式），我们知道文件中的任何两个关键字都处于正确的顺序，因为第 $i$ 个尾字节的第一位是不同的，对该位进行排序就能将它们放到合适的位置；或者是因为第 $i$ 个尾字节的第一位相同，根据稳定性，这两个关键字的顺序正确。用另一种方法解释，如果一对关键字中 $w-i$ 位未检查的值相等，关键字间的差别就由 $i$ 个已检查的位决定。相反，如果 $w-i$ 位未检查的值不相等，那么已经检查的 $i$ 位没有影响，迟一点的步骤会根据更明显的差别修改两者的顺序。

稳定性的必要性意味着，例如，二进制快速排序的划分方法不能用于这种从右到左排序的二进制版本中。相反，关键字索引统计操作是稳定的，可以用它来编写典型有效的算法。程序 10.4 是该方法的一个执行程序。似乎需要额外的数组进行分配操作——运用练习 10.17 和 10.18 中的技术可以不使用辅助数组，但牺牲了稳定性。

LSD 基数排序是老式卡片排序机器所使用的方法。那些机器可以根据卡上选定列的孔的式样，将一桌子的卡分到 10 个接收器中。如果卡片中列有标志数字，可以通过对卡片最右一位进行检查，接着收集卡片按顺序堆放，进行位数左移，以此类推。直到访问到第一位。对卡片进行的堆放操作是稳定的，关键字索引统计就是模仿这个操作。这个版本的 LSD 基数排序不仅在 20 世纪 50 年代和 60 年代的商业应用中很重要，它还被许多谨慎的程序员所使用，在程序卡片组最后几列中标志序列号码，这样当机器意外关闭时，可以将卡片组按顺序复原。

| | | | |
|---|---|---|---|
| now | sob | cab | ace |
| for | nob | wad | ago |
| tip | cab | tag | and |
| ilk | wad | jam | bet |
| dim | and | rap | cab |
| tag | ace | tap | cav |
| 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 |

**图 10.14 LSD 基数排序例子**

应用 LSD 基数排序，通过三步（从左到右）将 3 字母单词排好序。

**程序 10.4　LSD 基数排序**

该程序对字中的字节进行关键字索引统计，从右移向左。关键字索引统计操作必须是稳定的。如果 R=2（这样 bytesword 和 bitsword 相等），该程序是直接基数排序——一个从右到左，一位接一位的基数排序（参看图 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 = 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[count[digit(a[i], d)]++] = a[i];
        for (i = l; i <= r; i++) a[i] = aux[i-l];
      }
  }
```

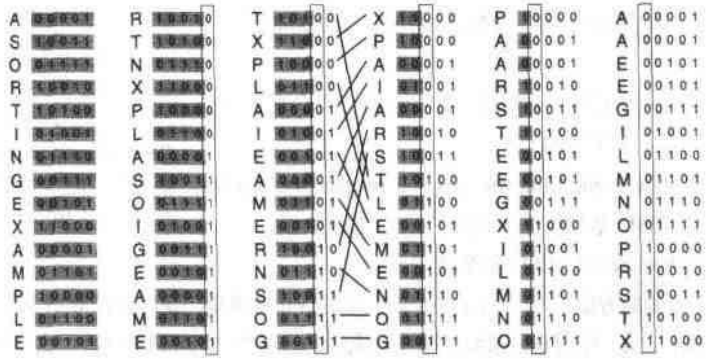


**图 10.15　LSD（二进制）基数排序例子（揭露关键字的各位）**

该图表显示了对示例文件进行从右到左、一位接一位的基数排序操作。通过对第（$i$-1）列进行提取操作（稳定的）将所有第 $i$ 位为 0 的关键字调到所有第 $i$ 位为 1 的关键字前，这样就产生了第 $i$ 列。如果操作前第（$i$-1）列是按第（$i$-1）位的值进行排序的，那么在操作后，第 $i$ 列是按尾 $i$ 位的值进行排序的。第二步中明确地显示了关键字的移动操作。

图 10.15 描述了对示例关键字进行二进制 LSD 基数排序操作，与图 10.3 进行比较。对这些 5 位关键字，排序算法用 5 步完成了，从右到左访问关键字的各位。对单位关键字记录进行排序相当于将文件进行划分，使所有 0 关键字的记录出现在 1 关键字记录之前。如刚才所

说的，我们不可以使用本章一开始讨论的程序 10.1 的划分方法，虽然它能解决同一个问题，但它是不稳定的。基数 2 排序很重要，因为它经常适用于高性能机器和有特殊目的的硬件中（参看练习 10.38）。软件中，我们使用尽可能多的位数来减少操作的步骤数，只受统计中所用的数组的大小限制（参看图 10.16）。

由于有变长的关键字，很难将这个 LSD 过程应用到字符串排序中。对于 MSD 排序，可以根据关键字的开始位进行区分；但 LSD 排序是基于固定长度的关键字的，开始位要在最后一步中才处理。即使对于（长的）固定长度的关键字，LSD 基数排序仍然对关键字的右部分做了许多不必要的工作，因为，如我们所看到的，排序中只有关键字的左边部分是真正有用的。在 10.7 节中，我们在详细地讨论了基数排序的特性后，会提出一个解决的方法。

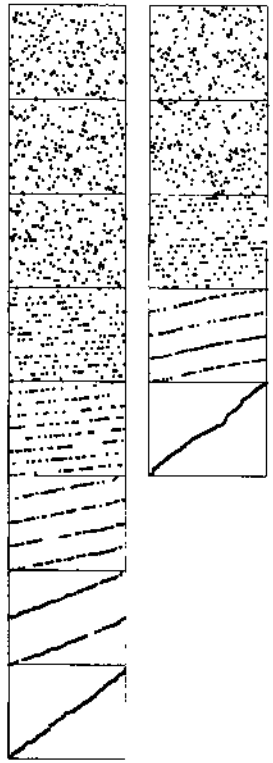


**图 10.16 LSD 基数排序动态特性**

本图显示了对随机 8 位键进行 LSD 基数排序的过程，基数为 2（左边）和 4（右边），基数为 4 的排序中包含了基数 2 中的所有步骤。例如，当剩下两位时（左边倒数第 3 个，右边倒数第 2 个），该文件包含了以 00、01、10、11 开头的四个子文件。

### 练习

10.32 使用练习 10.19 产生的关键字，对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，进行 LSD 基数排序。将它的运行情况和 MSD 基数排序的进行比较。

10.33 使用练习 10.21、10.23 产生的关键字，对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，进行 LSD 基数排序。将它的运行情况和 MSD 基数排序的进行比较。

10.34 使用图 10.15 例子的二进制快速排序划分方法，进行 LSD 基数排序，显示其结果（非排序的）。

△ 10.35 对关键字 now is the time for all good people to come the aid of their party，使用 LSD 基数排序，对最先两位进行操作。

● 10.36 编写一个对链表进行 LSD 基数排序的程序。

● 10.37 寻找一个有效的方法：（*i*）对文件中的记录进行重新整理，使所有以 0 开头的关键字排在所有以 1 开头的关键字的前面；（*ii*）使用与记录数目平方根成比例的额外空间（或者更少）；（*iii*）是稳定的。

● 10.38 对 32 位字数组进行排序，只能使用以下抽象操作：给定某位的位置 *i*，和指向数组 a[k]的指针，对 a[k]，a[k+1]，…，a[k+63]进行重新整理，操作必须是稳定的，使所有第 *i* 位是 0 的字排在第 *i* 位是 1 的字之前。

## 10.6 基数排序的特性

对 $N$ 个 $w$ 字节关键字进行 LSD 基数排序的运行时间和 $Nw$ 成比例，因为该算法中对 $N$ 个关键字一共访问了 $w$ 次。这个分析结果和输入数据无关，如图 10.17 所示。

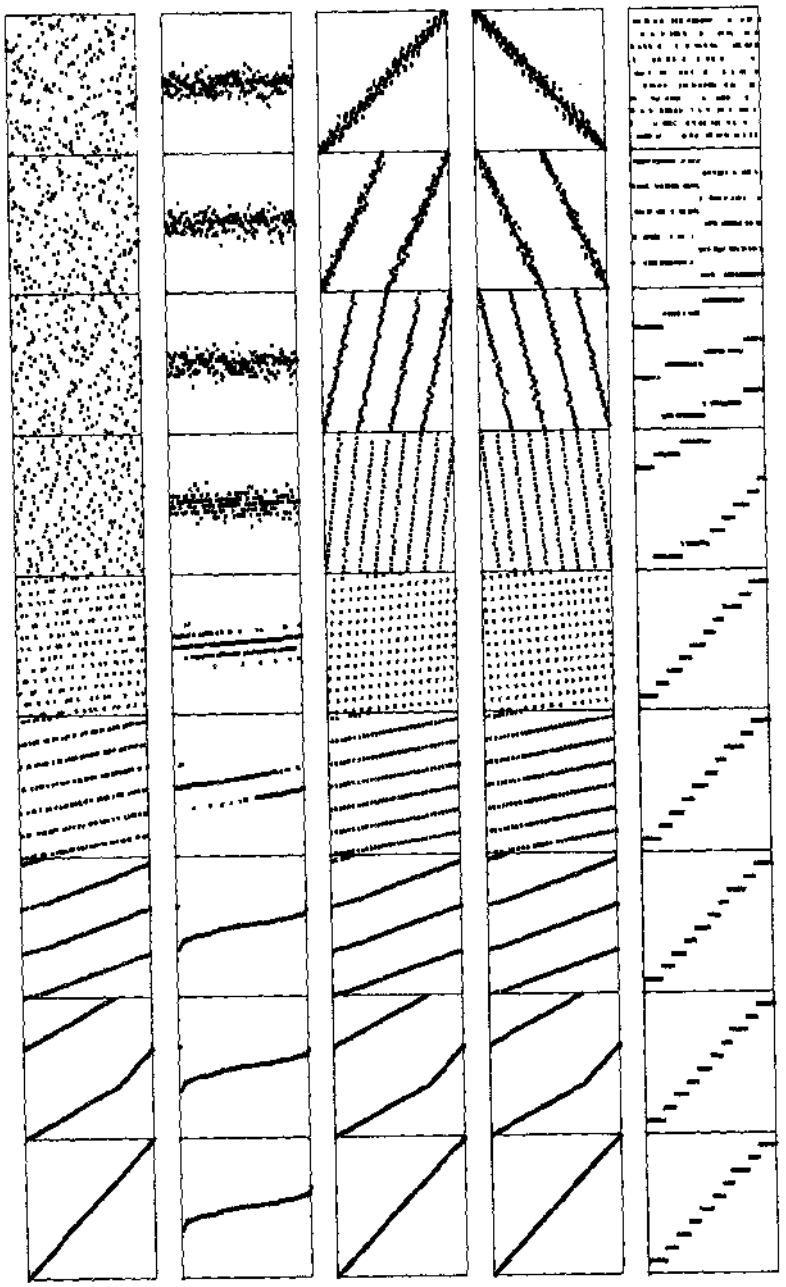


图 10.17 对不同类型文件执行 LSD 基数排序的动态特性

本图描述了使用 LSD 基数排序对包含 200 个数据项的随机文件、高斯序列、接近有序、接近倒序、和随机只有 10 个关键字的文件进行操作的情况（从左到右）。运行时间和输入数据的初始化顺序没有太大关系。包含相同关键字的三个文件（第一、第三和第四个，都是从 1 到 200 的整数序列），在接近完成操作的时候，排序情况几乎相同。

在关键字长和字节短的情况下，运行时间可近似于和 $N \lg N$ 成比例：例如，使用二进制 LSD 基数排序对 10 亿个 32 位关键字进行操作时，$w$ 和 $\lg N$ 都差不多是 32。关键字较短和字节较长的情况下，运行时间可近似于和 $N$ 成比例：例如，对 64 位关键字进行 16 位基数排序，即 W 是 4，一个很小的常数。

要精确地比较基数排序和其他排序方法的特性，要仔细地统计关键字中的字节数，而不只是统计关键字的总数。

**性质 10.1** *基数排序中最坏的情况是要检查所有关键字中的所有字节。*

也就是说，基数排序的运行时间是线性的，因为它最坏的情况就是运行时间和输入的总位数成比例。这个结论直接由对程序的观察得出：没有一位是要检查超过一次的。使用任何讨论过的程序，都可以发生最坏的情况，就是当所有的关键字相等的时候。■

如我们所见的，对随机关键字和其他许多情况，MSD 基数排序的运行时间可以是低于线性值的，因为并不需要对整个关键字进行检查。以下典型结论适用于关键字非常长的情况：

**性质 10.2** *当排序的关键字是随机数时，二进制快速排序平均对大概 $N \lg N$ 位进行检查。*

如果文件大小是 2 的幂方且位的值是随机的，那么大概一半的前导位是 0，一半是 1，因此循环 $C_N = 2C_{N/2}+N$ 能描述程序的性能，如我们在第七章所讨论的。并且，这种描述并不完全准确，因为对半进行划分只是平均的情况（还由于关键字中位数是有限的）。不过，二进制快速排序进行的划分要比标准快速排序的更接近中心，因此，运行时间大概等于最好的情况。对这个证明的详细分析是算法分析的典型事例，由 Knuth 在 1973 年前首次提出（参见“第三部分参考资料”）。■

这个结果可归纳运用到 MSD 基数排序中。不过，我们重点关心整个运行时间，而不只是关键字字符检验操作的运行时间，因此，我们必须注意，MSD 基数排序部分运行时间是和基数 R 的大小成比例，而与关键字无关。

**性质 10.3** *对大小为 $N$ 的文件进行基数 $R$ 的 MSD 基数排序至少需要 $2N+2R$ 步。*

MSD 基数排序包括至少一个关键字索引统计步骤，而关键字索引统计包括至少两个访问该记录的操作（一个是为了统计，一个是为了分配），总共至少 $2N$ 步，还有两个对统计数的访问操作（一个为了在开始时将它们初始化为 0，一个为了判定子文件在哪里结束），总共至少 $2R$ 步。■

该特性非常明显，但对于了解 MSD 基数排序很重要。特别是，它说明不能因为 $N$ 值较小就断定运行时间较短，因为 $R$ 可能比 $N$ 要大得多。简而言之，对小文件应使用其他方法。这个观察结果是我们在 10.3 节最后讨论的空接收器问题的解决方法。例如，如果 $R$ 是 256，$N$ 是 2，基本方法仅在比较操作上就比 MSD 基数排序要快 128 倍。MSD 基数排序的递归结构确定了递归程序会在很多小文件中对自己进行调用。因此，在本例中，如果忽略空接收器问题，会使整个基数排序的效率减低 128 倍。对于中间情况（如，假设 $R$=256，

$N$=64)。消耗情况没有这么严重，不过依然很巨大。插入排序并不适用，因为它大概要进行 $N^2/4$ 次比较操作，这也是很大的花费；忽略空接收器也不明智，因为数量很巨大。解决这个问题最简单的方法是使用小于文件大小的基数。

**性质 10.4** *如果基数总是小于文件大小，平均情况下，MSD 基数排序所执行的步骤数在 $N\log_R N$ 的一个小常数因子之内（关键字由随机数组成），而在最坏的情况下，步骤数在关键字总字节数一个常数因子之内。*

最坏情况下的结果直接可从之前的讨论得出，而性质 10.2 的分析可归纳出平均情况下的结果。对大 R，因数 $\log_R N$ 较小，因此总的运行时间和 $N$ 成比例。例如，如果 $R=2^{16}$，对所有 $N<2^{48}$，$\log_R N$ 小于 3，几乎所有实际的文件的大小都包含了。■

如从性质 10.2 得到的，从性质 10.4 我们可以推出一个重要的特性：对较长的随机关键字，MSD 基数排序的运行时间小于总位数的线性函数。例如，对 100 万个 64 位随机关键字进行排序，大概只需检查前导的 20 到 30 位，或者少于一半的数据。

**性质 10.5** *对 $N$ 个关键字（任意长），三路基数快速排序平均要进行 $2N\ln N$ 次字节比较操作。*

了解这个结论有两种方式。第一种，假设对前导位进行的划分操作和快速排序的相同，接着（递归地）对子文件进行相同的操作，因此整个操作的运行时间和普通快速排序的相同——不过进行的是对单个位值的比较，而不是对整个关键字。第二种，假设使用图 10.13 中提到的方法，由于快速排序中对 $R$ 位进行排序需 $2R\ln R$ 步，因此性质 10.3 中提出的 $N\log_R N$ 运行时间必须乘以 $2\ln R$ 因子。我们不进行完整的证明（参见“第三部分参考资料”）。■

**性质 10.6** *LSD 基数排序可以在 $w/\lg R$ 步中对 $N$ 个 $w$ 位关键字的记录进行排序，使用额外的空间存放 $R$ 统计数（和重新整理文件的缓冲器）。*

直接通过执行程序可以证明这个观点。特别是，如果 $R=2^{w/4}$，要进行 4 步线性排序。■

## 练习

10.39 假设输入文件中 1 到 1000 的数字每个有 1000 个，都是 32 位字。描述如何利用文件的特点编写一个高效的基数排序程序。

10.40 建设输入文件中，有 1000 个不同的 32 位数字，每个数字有 1000 个。描述如何利用文件的特点编写一个高效的基数排序程序。

10.41 使用三路基数快速排序对定长的字节字符串进行排序，在最坏的情况中，最多需要检查多少字节数？

10.42 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，对同一长字符串文件，使用三路基数快速排序和标准的快速排序，比较两者所检查的总字节数。

○ 10.43 对包含 $N$ 个关键字的文件 A，AA，AAA，AAAA，AAAAA，AAAAAA，...进行 MSD 基数排序和三路基数快速排序，求所要检查的总字节数。

## 10.7 运行时间低于线性的排序

根据 10.6 节的分析结果，我们可以得到一个重要结论：基数排序的运行时间可以比关键字总位数的线性函数要低。本节中，我们会讨论这个观点的实际应用。

在 10.5 节中的 LSD 基数排序执行程序在文件中使用了 bytesword。使用大的 $R$，只要 $N$ 也大，而且有足够的空间存放 R 统计数，排序方法就是有效的。如性质 10.5 证明过程中提到的，较合理的设置是 $\lg R$（每字节的位数）等于字大小的四分之一，因此，基数排序是 4 关键字索引统计操作。每个关键字的每个字节都经过检查，但每个关键字只有四个域。这个例子直接和许多计算机的结构对应：一个典型的结构是有 32 位字，每个字包含 4 个 8 位字节。我们从字中提取字节，而不是比特，因为在许多计算机中这样做更有效。现在，每个关键字索引统计步骤是线性的，而且因为只有四个，整个排序操作是线性的——也是我们在排序操作中所能期待的最好效率。

事实上，使用两个关键字索引统计步骤就可以了。可以这样做的原因是，当对 w 位关键字最先的 w/2 位信息进行了处理后，文件已基本排好序了。如快速排序中那样，我们可以通过使用插入排序来高效地完成任务。可以通过对程序 10.14 进行细微的改写来实现该方法。要使用最先的一半位进行从右到左排列，我们只需将外部的 for 循环改写成从 bytesword/2-1 开始，而不是 bytesword-1。接着，对得到的接近排好序的文件进行插入排序。图 10.3 和 10.18 很好地证明了一个前导位排好序的文件基本有序。图 10.3 第 4 列中的文件进行插入排序只需 6 次交换操作，而图 10.18 显示了一个较大的文件，如果前面的一半位排好了序，同样可以用插入排序对它进行有效的操作。

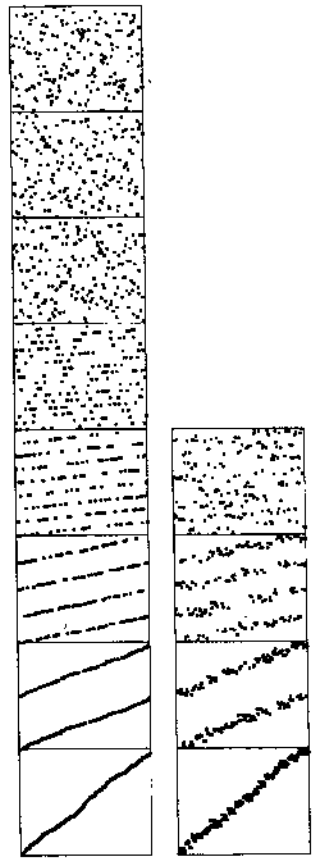


图 10.18 对 MSD 比特流进行 LSD 基数排序的动态特征

如果关键字是随机的位，根据关键字的前导位对文件进行排序可以使整个文件接近于有序。对包含 6 位关键字的随机文件，该图将一个 6 步 LSD 基数排序（左边）和一个 3 步 LSD 基数排序（右边）进行比较。第二种策略几乎快两倍。

对某些大小的文件，需要使用原来辅助数组使用的额外空间进行一次关键字索引统计步骤，用于对文件进行重新整理。例如，对 1 百万个 32 位随机关键字进行排序，可以使用一次对前导的 20 位进行的关键字索引统计排序和一次插入排序来完成操作。这样做，需要空间存放（1 百万）统计数——比辅助数组所需的要少得多。使用这个方法和使用基数 $2^{20}$ 的 MSD 基数排序相同，虽然本质上对小的文件应使用小基数（参看性质 10.4 之后的讨论）。

基数排序中的LSD步骤被广泛使用，因为它的控制机构非常简单，而且它的基本操作适合机器语言执行，这点可以直接改编到有特殊功能的高效硬件中。在这种环境中，运行完整的LSD基数排序可能是最快的。如果使用指针，需要空间存放N个指针（及R统计数）来执行LSD基数排序，这点投资可以产生一个对随机文件进行排序只需3到4步的方法。

在常规编程环境中，关键字索引统计程序内部循环中的操作数目比快速排序或归并排序内部循环的要多很多。该特性表明很多情况下，刚才提出的线性运行时间可能不比快速排序的运行时间少很多。

普通目的的算法如快速排序比基数排序要应用得广泛，因为它们适用于大量不同的情况。这点的主要原因是，基数排列基于的关键字抽取算法没有我们在第六章到第九章使用的算法（比较操作）那么普遍。例如，为排序实用程序安排接口程序的一个基本方法是让客户端提供比较操作功能。这是C++库中qsort使用的接口程序。这样的安排不仅让客户端可以使用关于重复关键字的特定信息来执行一个更快的比较操作，还可以使用不包含关键字的顺序关系来进行排序。在第二十一章中，我们会讨论一个这样的算法。

当可以使用任意算法时，是使用快速排序还是使用本章所提到的各种基数排序算法（及相关版本的快速排序），这点就不仅和应用程序的特点有关（如关键字、记录和文件大小），还和编程及机器环境的特性有关，硬件方面的特性决定了存储的效率和比特及字节的使用。表格10.1和10.2的试验结果验证了我们提出的各种基数排序的线性或低于线性运行时间的理论，这点使这些排序方法适用于多种合适的情况。

**表10.1 基数排序的研究（整数关键字）**

对N个元素的32位随机整数文件，这些基数排序（都包含一个当$N$小于16时改为执行插入排序的中断）的相关运行时间表明基数排序可以是最快的排序程序，如果小心使用的话。当我们对小文件使用大基数时，MSD基数排序的效率很低，不过对基数进行修改，使之小于文件大小就可以解决这个问题。对整数关键字操作最快的方法是对一般前导位进行排序的LSD基数排序，通过仔细处理内部循环还可以进一步提高效率（参看练习10.45）。

| | | 4位字节 | | 8位字节 | | | 16位字节 | | |
|---|---|---|---|---|---|---|---|---|---|
| $N$ | 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 |

其中：

Q 快速排序，标准（程序 7.1）

M MSD 基数排序，标准（程序 10.2）

L LSD 基数排序（程序 10.4）

M* MSD 基数排序，基数根据文件大小设置

L* 对 MSD 位进行 LSD 基数排序

**表 10.2 基数排序的研究（字符串）**

对 Moby Dick 的前导 $N$ 字进行各种排序（除了堆排序，都包含一个当 $N$ 小于 16 时改为执行插入排序的中断），相关的运行时间表明 MSD 的第一步对字符串数据有效。三路基数排序中对小子文件进行中断的效果比不上其他方法，而且除非对插入排序进行修改，避免访问关键字的最先部分，否则一点效用也没有（参看练习 10.46）。

| $N$ | Q | T | M | F | R | X | X* |
|---|---|---|---|---|---|---|---|
| 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 |

其中：

Q 快速排序，标准（程序 7.1）

T 三路划分快速排序（程序 7.5）

M 归并排序（程序 8.2）

F 使用 Floyd 改进的堆排序（参看 9.4 节）

R MSD 基数排序（程序 10.2）

X 三路基数快速排序（程序 10.3）

X* 三路基数快速排序（包含中断程序）

## 练习

▷ 10.44 使用 LSD 基数排序对前导位进行操作，然后使用插入排序完成任务的主要障碍是什么？

● 10.45 编写一个程序，对 32 位关键字进行 LSD 基数排序，内部循环中的操作要尽量的少。

10.46 执行三路基数快速排序，对小文件进行的插入排序不使用那些已知是相等的前导位。

10.47 给出 1 百万个随机 32 位关键字，确定字节大小，使总的运行时间最短，其中对头两个子节进行 LSD 基数排序，接着使用插入排序来完成任务。

10.48 如果使用，10 亿个 64 位关键字，回答练习 10.47。

10.49 如果使用，3 步 LSD 基数排序，回答练习 10.48。

# 第十一章

# 特殊用途的排序方法

排序方法是很多应用系统的关键组件；为了使一次排序的速度尽可能地提高，或使其具备处理巨大文件的能力，即使采取特别措施也不是什么异乎寻常的事。我们也许会遇到一个计算机系统的高性能扩展，或者特别针对排序而设计的专门硬件，又或者仅仅一个基于新型体系设计的计算机系统。在这些情况中，对于被排序数据的执行操作，我们过去对其相对开销所作出的默认假设可能不再有效了。在本章中我们将分析一些排序方法的范例，它们被设计成可以在多种不同类型的平台上高效率地运行。我们还要考虑影响高性能硬件的约束条件的几个不同例子，以及一些有助于实现高性能排序的方法。

任何新型的计算机体系最终都必须支持一种高效的排序方法。事实上，因为它是如此重要和易于理解，排序作为评价新体系结构的测试工具由来已久。我们不单希望知道哪些已知算法在一台新机器上运行得最好及其原因，还想知道一台新机器的特有性能是否可以在某些新算法中得到利用。为了开发新算法，我们定义了一台虚拟机器，它封装了实际机器的核心属性；并设计、分析对应虚拟机器的算法；然后对最佳算法和模型进行实现、测试和改进。我们利用以往的经验——包括适用于我们在第六章到第十章中探讨过的通用机器的许多方法，但虚拟机器所施加的限制帮助我们把注意力集中到系统真正的开销上，并清楚表明不同的算法适合不同的机器。

在讨论范围的一个极端，我们将考虑低层模型——惟一允许的操作是“比较-交换”操作；在讨论范围的另一个极端，我们将考虑高层模型——把大型数据块读出或写入到一个速度较慢的外部存储介质，或在独立的并行处理器之间执行大型数据块读出或写入。

首先我们分析一种归并排序版本，它被称为“巴彻尔奇偶归并排序”（Batcher's odd-even mergesort）。它基于一种分治策略的归并算法；该算法只使用“比较-交换”操作，以及对于数据移动执行的完全混洗（perfect-shuffle）和完全逆混洗（perfect-unshuffle）操作。它们不但在自己的范畴中具有重要意义，还适用于排序以外的很多其他问题。接下来我们把巴彻尔方法作为一种排序网络进行分析。排序网络（sorting network）是低层排序硬件的一个简单抽象。网络由互连的比较器（comparator）——能够执行“比较-交换”操作的模块——组成。

另一种主要的抽象排序问题是外部排序（external sorting），这是指排序的文件过大以至于无法全部加载入内存中。访问单个记录所付出的开销甚至可能是不可接受的，因此我们将使用一个抽象模型——记录在内存和外部设备之间以大数据块方式传输。我们将考虑用于外部排序的两个算法，并通过模型对它们进行比较。

最后我们将讨论并行排序（parallel sorting），这种情况下被排序的文件分布于独立的并行处理器之间。我们定义一个简单的并行机器模型，然后分析巴彻尔方法如何提供一种高效的解决方案。我们使用同一个基本算法分别解决高层和低层问题，这本身就是一个展示抽象方法的巨大威力的极具说服力的例子。

这一章中不同的虚拟机器比较简单，但值得加以研究，因为它们封装的具体约束条件在特定的排序应用中可能是至关重要的。低层排序硬件必须由简单组件组成；而外部排序一般需要以块为单位访问巨大的数据文件，其中顺序访问的效率要比随机访问高得多；并行排序涉及到处理器之间的通信限制。一方面，我们不能赞同那些完全对应于某些实际机器的具体机器模型；另一方面，我们讨论的抽象方法不仅使我们接触到提供了有关机器性能的实质限制信息的理论方程，还有具有直接实用价值的有趣算法。

## 11.1 巴彻尔奇偶归并排序

开始我们先来讨论一种排序方法，它只基于两种抽象操作："比较-交换"（compare-exchange）操作和完全混洗操作（以及其逆操作——完全逆混洗。这个算法在1968年首先由巴彻尔提出，因此称为巴彻尔奇偶归并排序。用混洗、"比较-交换"和双重递归实现这个算法是件很简单的事，但要明白为什么算法成功运作、分解混洗和递归的过程以便在低层次上分析其运作的细节却是很有挑战性的。

我们在第六章里曾简要地接触过"比较-交换"操作，当时我们提到：一些在那里讨论的基本排序方法可根据这个抽象操作，更简练地表示出来。现在我们感兴趣的是单独使用"比较-交换"操作分析数据的方法。标准的比较将被排除在外："比较-交换"操作不返回结果，因此一个程序无法采取依赖于具体数据值的行动。

**定义 11.1** *非适应性排序算法是一种执行操作的序列只依赖于输入的个数，而不是关键字数值的排序算法。*

在这一节中，我们的确允许单向重新组织数据的操作，例如交换和完全混洗；但它们不是最本质的，我们将在11.2节看到这一点。非适应性方法是排序的直线程序（straight-line program）的等价物：它们可以被直接表示为一个将要执行的"比较-交换"操作的列表。例如以下序列是一个对三个元素排序的直线程序：

```
compexch(a[0], a[1])
compexch(a[1], a[2])
compexch(a[0], a[1])
```

我们使用循环、混洗和其他高级操作，以求方便、简洁地表达算法；但我们开发一个

算法的目标是：定义一个 compexch 操作的固定序列，使其可以为每一个 $N$ 值，对任何由 $N$ 个关键字的组合进行排序。为了不失一般性，我们可以假设关键字的值是 1 到 $N$ 之间的整数（参阅练习 11.4）；要知道一个直线程序的正确性，我们必须证明它可以排序所有这些值的每种可能存在的排列（例如参阅练习 11.5）。

**程序 11.1　完全混洗和完全逆混洗**

程序 11.1 给出了我们将要使用的另一种抽象操作——完全混洗和完全逆混洗的一个实现方法。而图 11.1 则对两者分别给出了范例。完全混洗重新排列数组，其方式和一位专家洗一副纸牌时可能采用的方式相对应：纸牌被准确地分成两半，交替地从每一叠中抽出纸牌达到洗牌的目的。我们总是从纸牌的上半叠取出第一张纸牌。如果纸牌数为偶，则分成的两半有相等数量的牌；如果纸牌数为奇，则多出的一张牌放在上半部分。而完全逆混洗的作用则恰恰相反：我们通过交替地把牌放入上半部分和下半部分，达到使这叠纸牌逆混洗的目的。

```
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];
  }
```

我们在第六章到第十章中讨论的排序算法很少是非适应性的——它们都使用运算符 < 或者以其他方式分析关键字，然后根据关键字的值执行不同的操作。一个例外是仅仅使用“比较-交换”操作的冒泡算法（bubble sort，参阅 6.4 节）。Pratt 的 shell 排序（SHELLSORT 参阅 6.6 节）版本是另一个非适应性方法。

巴彻尔排序正是 8.3 节中的自上而下归并排序；差别在于它使用了巴彻尔奇偶归并——这是一种非适应性的自上而下递归归并方法，而不是第八章的适应性归并实现方法。程序 8.3 本身根本不访问数据，所以我们使用一种非适应性归并就意味着整个排序都是非适应性的。

在这一节和 11.2 节的正文中，如果不加以特别说明，我们都假设被排序的元素数量是 2

的次幂。那么我们就可以一直引用“$N/2$”，而无须对 $N$ 为奇数等情况作出特别说明。当然，这个假设是不切合实际的——我们的程序和例子涉及其他的文件大小数值，但它大大简化了讨论过程。我们将在 11.2 节末尾再回到这个问题进行讨论。

巴彻尔归并自身就是一个分治策略的递归方法。要进行 1-1 的归并，我们使用一次“比较-交换”即可。否则，要进行 $N$-$N$ 归并的话，我们执行逆混洗操作，得到两个 $N/2$-$N/2$ 的归并问题，然后用递归方式解决它们，得到两个有序的文件。再混洗这些文件，我们就获得一个几乎完成排序的文件。所剩余的步骤只是执行一遍 $N/2-1$ 次独立的“比较-交换”组成的处理：对于 $i$ 从 1 到 $N/2-1$，操作在元素 $2i$ 和 $2i+1$ 之间进行。图 11.2 描述了一个范例。从这个描述中我们马上可以得到程序 11.2 的具体实现。

为什么这个方法能够对所有可能的输入序列排序呢？这个问题的答案根本不能用“显而易见”来形容。经典的证明是一种间接方法，它基于非适应性排序程序的一个一般特性。

**性质 11.1（0-1 法则）** *若输入全部为 0 或 1 时，一个非适应性程序可产生排序完毕的输出，那么当输入是任意关键字时，它也可产生排序完毕的输出。*

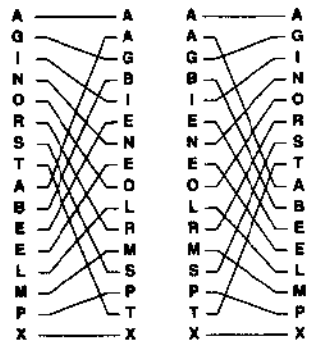



**图 11.1 完全混洗和完全逆混洗**

要执行一遍完全混洗（左图），我们取出文件的第一个元素，接着是下半部分的第一个元素、文件的第二个元素、下半部分的第二个元素……我们认为元素是从上到下由 0 开始编号。那么上半部分的元素进入偶数位置，下半部分的元素进入奇数位置。要执行一遍完全逆混洗（右图），我们的操作刚好相反：偶数位置的元素进入上半部分，奇数位置的元素进入下半部分。

**程序 11.2 巴彻尔奇偶归并（递归版本）**

本递归程序使用程序 11.1 的 shuffle 和 unshuffle 操作，实现了一个抽象内部归并。然而这些操作并不是不可或缺的——程序 11.3 是本程序的一个自下而上的非递归版本，并除去了混洗操作。在这里我们主要关心的是：当文件大小是 2 的次幂时，本实现方法提供了巴彻尔算法的一个紧凑描述。

```
template <class Item>
void merge(Item a[], int l, int m, int r)
  {
    if (r == l+1) compexch(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)
      compexch(a[i], a[i+1]);
  }
```

关于这个性质可参阅练习 11.7。■

**性质 11.2**　巴彻尔奇偶归并（程序 11.2）是一种有效的归并方法。

利用 0-1 法则，我们检查的只是：当输入全部为 0 或 1 时，该方法正确地进行归并。假设在第一个子文件中存在 $i$ 个 0，第二个子文件中存在 $j$ 个 0。这个性质的证明涉及到对四种情况的检验，取决于 $i$ 和 $j$ 是偶数或是奇数。若两者都是偶数，那么两个归并子问题分别都涉及到有 $i/2$ 个 0 的文件和有 $j/2$ 个 0 的文件，因此两个结果都有 $(i+j)/2$ 个 0。混洗后我们得到一排序完毕的 0-1 文件。在 $i$ 奇 $j$ 偶或 $i$ 偶 $j$ 奇的情况下，混洗后的 0-1 文件也是排序完毕的。但 $i$ 奇 $j$ 奇时，我们最后要把一个有 $(i+j)/2+1$ 个 0 的文件和一个有 $(i+j)/2-1$ 个 0 的文件一起混洗，因此混洗后的 0-1 文件具有 $i+j-1$ 个 0、1 个 1、1 个 0，然后是 $N-i-j-1$ 个 1（如图 11.3 所示）。由最后阶段的其中一个比较器完成排序。■

实际上我们并不需要混洗数据。的确，我们可以结合使用程序 11.2 和程序 8.3，产生一个适用于任何 N 值的直线排序程序；具体方法是修改 compexch 和 shuffle 的实现，改为保持索引并间接地引用数据（参阅练习 11.12）。又或者我们可以让程序输出“比较-交换”指令，以此利用原本的输入（参阅练习 11.13）。我们可把这些技术应用于任何通过交换、混洗或其他类似操作重新排列数据的非适应性排序方法。对于巴彻尔归并算法，其结构是如此简单，至于我们能够直接开发出一种自下而上的实现方法。在 11.2 节我们将看到这一点。

**图 11.2　自上而下的巴彻尔奇偶归并排序范例**

要使 A G I N O R S T 和 A E E L M P X Y 归并，我们从一个逆混洗操作开始，建立两个独立的、只有大约原先大小一半的归并问题（如第二行所示）。现在我们必须分别归并：A I O S 和 A E M X（数组的前半部分），G N R T 和 E L P Y（数组的后半部分）。在以递归方式解决这些子问题后，我们把它们的解答混洗（如倒数第二行所示），并通过以下“比较-交换”操作完成排序：E-A、G-E、L-I、N-M、P-O、R-S 和 T-X。

## 练习

▷ 11.1　给出混洗和逆混洗以下关键字的结果：E A S Y Q U E S T I O N。

11.2　推广程序 11.1，实现 h 路混洗和逆混洗。对于文件大小不是 $h$ 倍数的情况，解释你的策略。

● 11.3　实现混洗和逆混洗操作，其中不能使用辅助数组。

● 11.4　对一个可排序 N 个不同关键字的直线程序，说明它也可以排序 $N$ 个不一定不同的关键字。

▷ 11.5　说明正文给出的直线程序怎样对 1、2、3 构成的 6 种排列顺序进行排序。

○ 11.6　给出一个可排序四个元素的程序。

● 11.7　证明性质 11.1。提示：说明如果程序不能对某些带有任意关键字的输入序列排序，那么也存在一些它不能排序的 0-1 序列。

▷ 11.8 说明使用程序 11.2 时，以下关键字是如何被归并的：A E Q S U Y E I N O S T。使用图 11.2 图解的范例风格。

▷ 11.9 对关键字：A E S Y E I N O Q S T U，解答练习 11.8。

○ 11.10 对关键字：1 0 0 1 1 1 0 0 0 0 1 0 1 0 0，解答练习 11.8。

11.11 对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，通过试验比较巴彻尔归并排序和标准的自上而下归并排序（程序 8.3 和程序 8.2）的运行时间。

11.12 给出 compexch、shuffle 和 unshuffle 的具体实现，使程序 11.2 和程序 8.3 作为一个间接排序运作（参阅 6.8 节）。

○ 11.13 给出 compexch、shuffle 和 unshuffle 的具体实现，使程序 11.2 和程序 8.3 对应给定的 $N$ 值，打印出一个排序 $N$ 个元素的直线程序。你可以使用一个辅助全局数组记录索引。

11.14 如果我们把归并的第二个文件颠倒顺序，就得到一个正如 8.2 节所定义那样的 bitonic 序列。改变程序 11.2 的最后一个循环：从 L 而不是 L+1 开始，就可使程序变为可以对 bitonic 序列排序。说明使用这个方法时，以下关键字是如何被归并的：A E S Q U Y T S O N I E。使用图 11.2 图解的范例风格。

● 11.15 证明练习 11.14 中描述的经过修改的程序 11.2 可对任何 bitonic 序列排序。

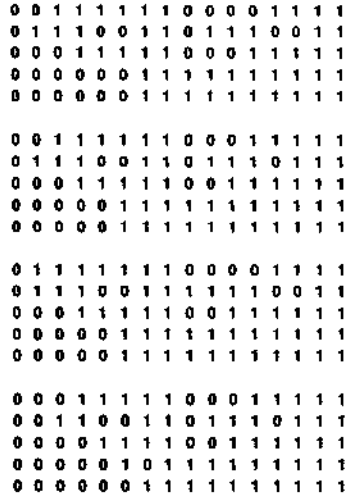


图 11.3 0-1 归并的四种情况

这四个范例各由五行组成，分别是：一个 0-1 归并问题；一次逆混洗操作的结果，它给出了两个归并问题；递归完成归并的结果；一次混洗的结果；最后奇偶比较的结果。仅当两个输入文件中 0 的个数都是奇数时，最后阶段执行一次交换。

## 11.2 排序网络

研究非适用性排序算法的最简单模型是一个仅能通过“比较-交换”操作访问数据的抽象机器。这样的机器称为排序网络（sorting network）。一个排序网络由原子部件——比较-交换模块（compare-exchange module），又称比较器（comparator）——组成。它们被连接在一起，以便实现执行完全一般化排序的能力。

图 11.4 展示了一个简单的排序网络，它对应四个关键字。通常我们把一个 $N$ 个元素的排序网络画成 $N$ 条水平线的序列，一对水平线之间由比较器连接。我们假想将被排序的关键字在网络中从右向左传递；每当遇到一个比较器时，若需要把较小的关键字放到上部，就互换这对数字。

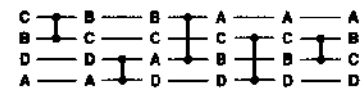


图 11.4 一个排序网络

关键字在网络中的水平线上从左向右移动。如果必须把较小的一个放到更高的线上，它们遇到的比较器就把关键字互换位置。在这个例子中，B 和 C 在顶部的两条线上互换，接着是 A 和 D 在底部的两条线上互换，再接着是 A 和 B，以此类推。结束时使关键字从上到下依次排列。本例中，除第四个比较器以外的所有比较器都执行互换，这个网络可排序四个关键字组成的任何序列。

在我们可以建立一个基于这个方案的实际排序机器之前，需要解决许多细节问题。比方说，我们还没有制订对输入数据编码的方法。一种方法是：把图 11.4 的每条连线看作一组直线，其中每条拥有 1 比特数据，使得一个关键字的所有比特同时流过一条线。另一种方法是：让比较器读入它们的输入，每次顺着单独一条线读入 1 比特（最高

位优先）。还有计时机制也没有制订：必须包括一种机制，以确保没有比较器在它的输入就绪以前就执行操作。排序网络是一种优良的抽象，因为它允许我们把这些实现的考虑事项和高层设计的考虑事项（例如将比较器的数量减到最少）分隔开来。而且我们还会在 11.5 节中看到，排序网络抽象也有助于直接电路设计以外的应用。

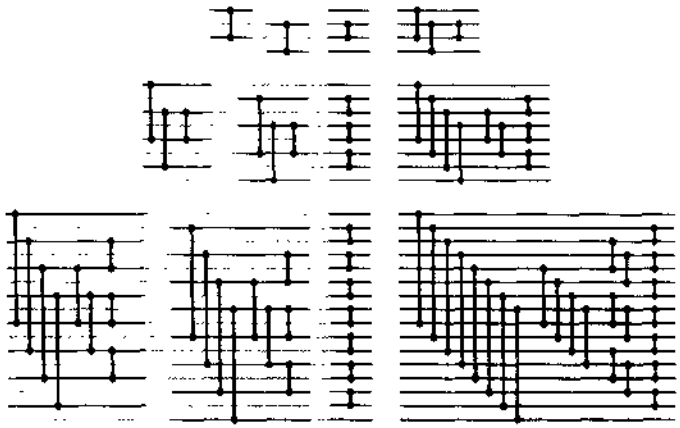


**图 11.5　巴彻尔奇偶归并网络**

本图包含了四条水平线（上图）、八条水平线（中图）和十六条水平线（下图）的网络，这些不同的表示方法展现了网络的基本递归结构。左边是大小为 $N$ 的网络结构的直接表示，其中带有两份大小为 $N/2$ 的网络副本：一份用于偶数编号的线，一份用于奇数编号的线；并附加了 1-2 线之间的比较器、3-4 线之间的比较器、5-6 线之间的比较器……右边是源自左边网络的、更为简单的网络，它们通过组合相同长度的比较器得出。因为我们可以移动奇数线上的比较器，使它们越过偶数线上的比较器而不产生干扰，所以这样的组合是完全可能的。

排序网络的另一个主要应用是作为并行计算的模型。如果两个比较器不使用同一条输入线，我们就认为它们可以同时运作。譬如图 11.4 的网络就显示了四个元素可在三步并行步骤中被排序的情况。0-1 和 2-3 比较器可以在步骤一中同时操作，接着 0-2 和 1-3 比较器可以在步骤二中同时操作，最后 2-3 比较器在步骤三中完成这次排序。给出任意的网络后，要完成以下的工作并不困难：把比较器分类，组成一系列的并行阶段（parallel stage），每个并行阶段由多组可以同时操作的比较器构成（参阅练习 11.17）。对于高效的并行计算，我们面对的挑战是设计出具有尽可能少的并行阶段的网络。

程序 11.2 和一个适用于所有 N 值的归并网络直接相应，但考虑一种直接的自下而上结构（如图 11.5 所示）对我们也很有启发意义。为了构造一个大小为 $N$ 的归并网络，我们使用两份大小为 $N/2$ 的网络副本：一份用于偶数编号的线，一份用于奇数编号的线。因为两组比较器互不干扰，我们可以重新组织它们，使两个网络交错。那么最后我们就完成了在 1 和 2、3 和 4 等直线之间带有比较器的网络；如果线的数量更多，以此类推。这个奇偶交错处理取代了程序 11.2 里的完全混洗。至于这些网络准确地归并的证明和为性质 11.1、11.2 给出的是一样的，它们都利用了 0-1 法则。图 11.6 显示了一个执行中的归并操作的范例。

**程序 11.3 巴彻尔奇偶归并（非递归版本）**

这个巴彻尔奇偶归并（假设文件大小是 2 的次幂）的具体实现简洁而神秘。我们可以通过分析它如何与递归版本（参阅程序 11.2 和图 11.5）对应，了解到它完成归并的方式。程序在 $\lg N$ 遍执行内完成归并，这些执行由统一和独立的“比较-交换”指令组成。

```
template <class Item>
void merge(Item a[], int l, int m, int r)
  { int N = r-l+1;  // assuming N/2 is 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]);
  }
```

程序 11.3 是一个巴彻尔归并的自下而上实现，并且不带混洗操作。它对应于图 11.5 的网络。这个程序是个紧凑而雅洁的内部归并函数，也许理解它的最佳途径就是把它看作网络的一种备选表示方案。然而关于它成功地完成归并任务的直接证据并不容易得到，对这个问题的思考也是相当有趣的。我们将在这一节的末尾分析一个这样的证明。

图 11.7 显示的巴彻尔奇偶排序网络由图 11.5 的归并网络创建，其中利用了标准的递归归并排序结构。这个结构是双重递归的：一次用于归并网络，一次用于排序网络。虽然它们不是最优的——我们不久将讨论最优网络——这些网络也具有很高的效率。

**性质 11.3** 巴彻尔奇偶排序网络具有大约 $N(\lg N)^2/4$ 个比较器，可用 $(\lg N)^2/2$ 步并行步骤运行。

归并网络大约需要 $\lg N$ 个并行步骤；而排序网络需要 $1 + 2 + \cdots + \lg N$ 个，或者说大约 $(\lg N)^2/2$ 个并行步骤。这里将把比较器的计数留作练习（参阅练习 11.23）。■

在程序 8.3 里的标准递归归并排序中使用程序 11.3 的归并函数，将给出一个紧凑的内部排序方法。它是非适应性的，调用 $O(N(\lg N)^2)$ 次“比较-交换”操作。作为另一种选择，我们可以从归并排序中消除递归，直接实现整个排序的一个自下而上版本（如程序 11.4 所示）。和程序 11.3 一样，也许理解它的最佳途径就是把它看作图 11.7 网络的一种备选表示方案。这个实现涉及到向程序 11.3 加入一个

**图 11.6 自下而上的巴彻尔归并范例**

消除了所有混洗后，我们例子中的巴彻尔归并总计需要 25 次“比较-交换”操作，这里已描述出这些操作。它们划分为四个独立的“比较-交换”操作阶段，每阶段的操作都有固定的偏移量。

循环和一项测试，因为归并和排序具有相似的递归结构。为执行自下而上的一遍操作，把长 $2^k$ 的有序文件的序列归并成长 $2^{k+1}$ 的有序文件的序列，我们要使用完整的归并网络。但只需要引入那些完全处于子文件中的比较器。这个程序也许可以赢得一个奖项——它是我们所见的最简洁的重要排序实现方法。当我们希望充分利用高性能的结构特性，为小文件开发一种高速的排序（或建立一个排序网络）时，它很可能是最佳选择。如果我们没有预先了解这些正被讨论的递归实现和网络结构的话，要理解这个程序排序的方式及其原因的困难程度将是令人生畏的。

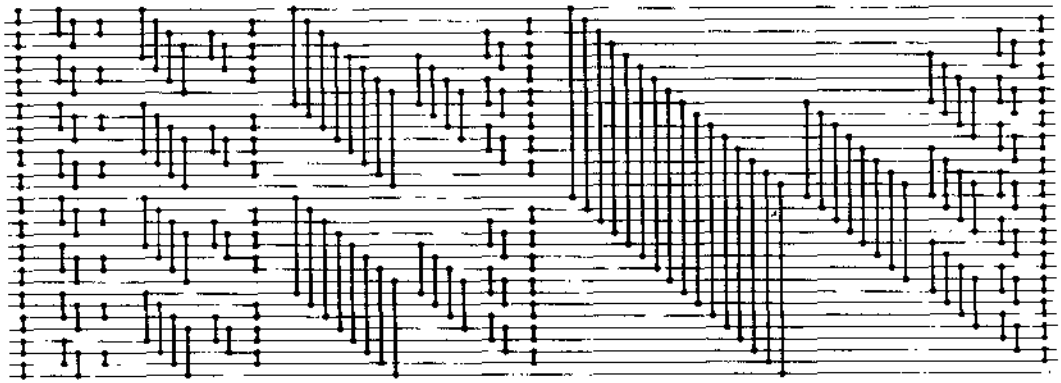


**图 11.7　巴彻尔奇偶排序网络**

这个 32 线的排序网络包括两个 16 线网络的副本、四个 8 线网络的副本……从右向左看，我们就以自上而下的方式了解其结构：一个 32 线的排序网络由两个 16 线排序网络副本（一个用于上半部分，一个用于下半部分）及它们之后的一个 16×16 归并网络组成。每个 16 线网络由两个 8 线排序网络副本及它们之后的一个 8×8 归并网络组成，如此类推。从左向右看，我们就以自下而上的方式了解其结构：首列比较器创建出大小为 2 的排序子文件；接着我们得到 2×2 归并网络，它创建出大小为 4 的排序子文件；然后 4×4 归并网络创建出大小为 8 的排序子文件……以此类推。

和平常的分治策略方法一样，当 $N$ 不是 2 的次幂时我们有两个基本选择（参阅练习 11.24 和练习 11.21）。我们可以从中间划分（自上而下），或在小于 $N$ 的 2 最大次幂处划分（自下而上）。对于排序网络来说后者稍稍简单一些，因为它等价于以下处理：为大于等于 $N$ 的 2 最小次幂建立一个完全网络，然后仅仅使用前 $N$ 条线以及两端都和这些线相连的比较器。要证明该结构的有效性相当简单。假定没被使用的线有这样的标志关键字：它们大于网络中的其他任意关键字，那么那些线上的比较器永不交换，所以消除它们没有影响。当然，我们可以使用来自较大网络的任何 $N$ 条相邻线的集合——我们认为顶部的忽略不用的线有较小的标志值，而底部的忽略不用的线则有较大的标志值。所有这些网络都有大约 $N(\lg N)^2/4$ 个比较器。

排序网络的理论拥有一段充满趣味的历史（参见“第三部分参考资料”）。Bose 在 1960 年之前首先提出以下问题——找出比较器数量尽可能少的网络，因此它也被称为 Bose-Nelson 问题。巴彻尔网络是第一个解决该问题的正确方案，而且在一段时期内人们猜测它是最优的。巴彻尔归并网络的确是最优解，因此任何实际拥有更少比较器的排序网络都必须用

递归归并排序以外的途径建造。研究人员一直无法解决寻找最优排序网络的问题，直到1983年，Ajtai、Komlos和Szemeredi才证明：存在具有$O(N \log N)$个比较器的网络。然而AKS网络是一个没有任何实用价值的数学模型，而巴彻尔网络至今依然是可实际应用的最佳方案之一。

**程序11.4 巴彻尔奇偶排序（非递归版本）**

巴彻尔奇偶排序的实现直接对应于图11.7的网络表示。它划分成多个阶段，它们分别由变量p标出。最后一个阶段（p等于N时）是巴彻尔奇偶归并。倒数第二阶段（p等于N/2时）是奇偶归并，它消除了第一阶段和所有跨越N/2值的比较器；倒数第三阶段（p等于N/4时）是奇偶归并，它消除了头两个阶段和所有跨越N/4值的比较器；以此类推。

```
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))
              compexch(a[l+j+i], a[l+j+i+k]);
  }
```

完全混洗和巴彻尔网络之间的联系使我们以下的课题变得很有趣：通过考虑另一版本算法来完成对排序网络的研究。如果我们混洗巴彻尔奇偶归并中的直线，将得到所有比较器都联系着相邻直线的网络。图11.8展示了一个网络，它对应于程序11.2的相应混洗实现。这种互相连接的模式有时也称为蝴蝶网络（butterfly network）。图中显示的还有同一直线程序的另一表示方式，它提供了一种更为一致的模式，而且只涉及满混洗。

对于阐明其基础结构的方法，图11.9给出了又一种解释方式。首先我们把一个文件写在另一个下面，然后比较那些垂直相邻的元素，如果必须把较大者置于较小者下方，则进行互换。接下来我们平分每一行，使分割出的每一半（共四部分）交错，再对第二行、第三行的元素执行同样的“比较-交换”操作。因为先前已执行了排序，所以不必进行涉及其他行对之间的比较。“拆分-交错”操作可让表的行、列保持有序状态。这个性质是通过同一的操作来保持不变的：每

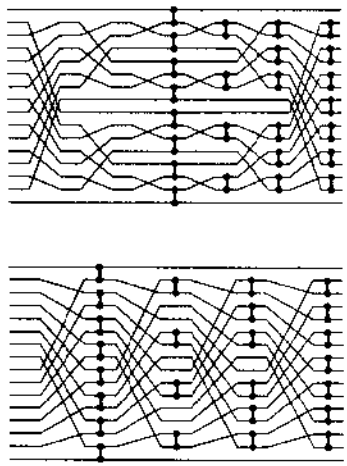


**图11.8 巴彻尔奇偶归并中的混洗**

程序11.2的一个作为排序网络的直接实现，给出了一个充满递归逆混洗和混洗的网络（上图）。而一种等价的实现只涉及到满混洗（下图）。

步骤使行数加倍，列数减半，却仍然使行、列保持有序状态；最后我们以 N 行组成的 1 列结束，因此这就是完全排序后的结果。图 11.9 中的图形和图 11.8 底部的网络之间的联系是：当我们以列序为主序（首列的元素后跟随的是第二列的元素，如此类推）写下这些表时，我们会发现从其中某一步到下一步所需的置换正是完全混洗。

现在，在一个内建了完全混洗互连功能的虚拟并行机器（如图 11.10 所示）的帮助下，我们将可以直接实现类似图 11.8 底部那样的网络。正如算法指出的那样，每个步骤中机器都在某些相邻的处理器对之间执行“比较-交换”操作，然后对数据进行一次完全混洗。为该机器编程的工作，需要指定在每次循环中哪些处理器对应该进行“比较-交换”操作。

对于自下而上方法和这个巴彻尔奇偶归并的全混洗版本，图 11.11 都显示了它们的动态特性。

混洗是一种描述在分治策略算法中数据移动的重要抽象，它起源于排序以外的许多问题。譬如，若一个 $2^n \times 2^n$ 平方矩阵以行序为主序储存，那么 $n$ 次完全混洗将会使矩阵转置（把矩阵转换成以列序为主序储存）。更重要的例子包括快速 Fourier 变换和多项式求值（参阅第八部分）。通过一个类似于图 11.10 示例的循环完全混洗机器，我们可以解决所有这些问题，但需要更多强大的处理器。我们甚至可以构想拥有能够执行完全混洗和完全逆混洗的通用处理器（已经建造出一些这种类型的真实机器）。在 11.5 节我们将回到对这些并行机器的讨论上来。

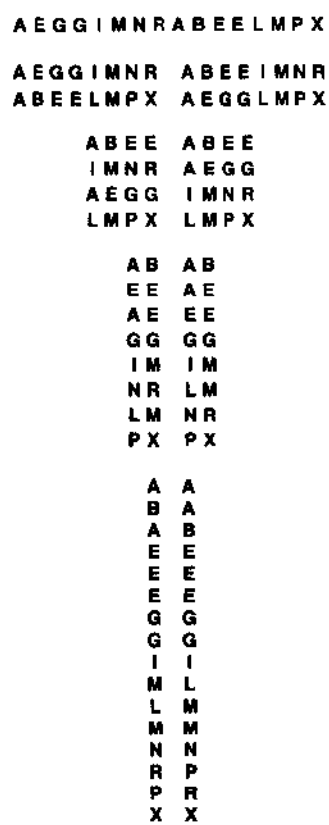



**图 11.9　“拆分-交错”归并**

从两个排列于同一行的有序文件开始，我们重复以下操作来实现归并：将每行平分成两半，使它们交叉排列（左图），在目前垂直相邻、来自不同行的元素之间执行“比较-交换”操作（右图）。开始时我们有 16 列-1 行，然后 8 列-2 行、4 列-4 行、2 列-8 行，最后有 1 列-16 行——这就是排序结果。

## 练习

11.16　分别给出对应 4（参阅练习 11.6）、5、6 个元素的排序网络，要求使用尽可能少的比较器。

○ 11.17　编写一个程序，计算任意给定直线程序所需执行的并行步骤数。提示：使用以下的标记策略。把输入线标记为属于 0 阶段，然后对每个比较器作以下处理：如果两条输入线之一标号为 $i$ 而且另一条标号不大于 $i$，把两条输出线标记为 $i+1$ 阶段的输入。

11.18　当 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$ 时，对随机次序的关键字比较程序 11.4 和程序 8.3 的运行时间。

▷ 11.19　为执行一个 10 × 11 归并画出巴彻尔网络。

11.20　证明图 11.8 提出的递归逆混洗和混洗之间的关系。

○ 11.21　根据正文的参数，在图 11.7 中隐藏着 11 个排序 21 元素的网络。画出其中具有最少比较器的一个。

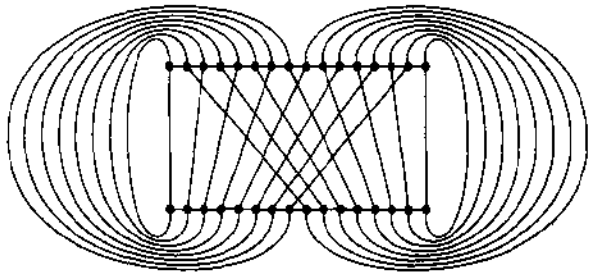


**图 11.10 一个完全混洗机器**

这里画出了一个具有互连性质的机器，它能够有效地执行巴彻尔算法（和很多其他算法）。一些并行计算机也有类似的连接。

11.22 对 $2 \le N \le 32$，给出巴彻尔奇偶排序网络中比较器数量。当 $N$ 不是 2 的次幂时，网络来源于以下网络的前 $N$ 条线：它对应其次最大的 2 次幂。

○ 11.23 对 $N = 2^n$，导出巴彻尔奇偶排序网络中使用的比较器数量的精确表达式。注意：对照图 11.7 检查你的答案，该图分别显示了 $N$ 等于 2、4、8、16 或 32 时，拥有 1、3、9、25 和 65 个比较器的网络。

○ 11.24 构造一个排序网络，使用自上而下的递归风格排序 21 个元素，其中大小为 $N$ 的网络是以下网络的合成：两个大小分别为 $\lfloor N/2 \rfloor$、$\lceil N/2 \rceil$ 的网络，以及其后的一个归并网络。（把你对练习 11.19 的答案作为该网络的最终部分。）

11.25 对根据练习 11.24 中的描述构建的排序网络（$2 \le N \le 32$），利用递归关系计算其中比较器的数目，将所得结果和练习 11.22 的结果相比较。

● 11.26 找出一个 16 线排序网络，要求比巴彻尔网络使用更少的比较器。

11.27 利用练习 11.14 描述的方案，画出对应图 11.8 的适用于 bitonic 序列的归并网络。

11.28 对 $N = 32$，画出对应具有 Pratt 增量的希尔排序（参阅 6.6 节）的排序网络。

11.29 对 $N = 16$，32，64，128 和 256，给出一张表，其中包含了练习 11.28 描述的网络中比较器的数量，以及巴彻尔网络中比较器的数量。

11.30 设计一个排序网络，对含有 N 个元素的文件排序，这些元素是“3-排序”和“4-排序”的。

● 11.31 利用练习 11.30 所得的网络，设计一个基于 3 和 4 乘积的

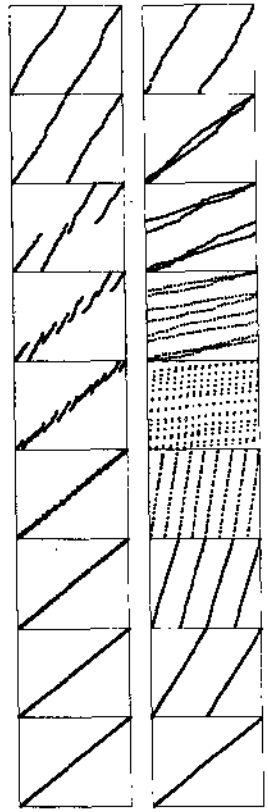


**图 11.11 奇偶归并的动态特性**

奇偶归并的这个自下而上版本（左图）涉及到一系列阶段——我们对一个排序子文件较大的半部分和下一个文件较小的半部分执行“比较-交换”。使用完全混洗（右图）后，该算法就有一种完全不同的外观。

类 Pratt 方案。对 $N=32$ 画出你的网络，并根据它回答练习 11.29。

● 11.32　对 $N=16$，画出巴彻尔奇偶排序网络的一个版本，使连接相邻直线的独立比较器阶段之间存在完全混洗。该网络的最后四个阶段应该来自图 11.8 底部的归并网络。

○ 11.33　为图 11.10 的机器编写一个归并程序，并使用以下约定。指令是 15 比特的序列：对于 $1\leq i\leq 15$，第 $i$ 比特如果为 1，则表明处理器 $i$ 和处理器 $i$-1 应该执行一次“比较-交换”操作。一个程序是指令的序列，机器在每条指令之间执行一遍完全混洗。

○ 11.34　使用练习 11.33 描述的约定，为图 11.10 中的机器编写一个排序程序。

## 11.3　外部排序

我们接下来转到另一种抽象排序问题，它适用于以下情况：被排序文件过大而不能全部加载到计算机的随机访问存储器。我们使用术语外部排序（external sorting）来描述这样的情况。存在许多不同类型的外部排序设备，它们也许会对用于实现排序的原子操作加以大量的不同限制。但是，对使用下面两种基本原子操作的排序方法进行的讨论依然是有益的：读取，把数据从外部存储器读取到主内存中；写入，把数据从主内存写入到外部存储器中。我们假设这两种操作的开销远远大于基本计算操作的开销，以至于我们可以完全忽略后者。例如在这个抽象模型中，我们就忽略内存排序的开销！对于巨大的内存或糟糕的排序方法，本假设也许是不恰当的。但如果有必要的话，一般可以对实际情况中的真实开销作出估计。

外部存储设备的类型和成本各有不同，种类繁多，因此使得外部排序方法的开发高度依赖于现有的技术。这些技术可能很复杂，很多参数会影响它们的性能。因此一种聪明的方法也许因为技术上一个简单的更改就不被人们欣赏或使用了——在对外部排序的研究中这是完全可能的。由于这个原因，我们在这一节中将集中回顾一般的方法，而不会讨论具体的实现。

鉴于外部设备高昂的读写开销，通常对访问操作设有严格的限制，具体情况视特定设备而定。例如对于绝大多数类型的设备，主内存和外部存储器之间的读写操作一般都以大型连续数据块的形式进行，这是最高效率的方法。同样，高容量的外部设备常常被专门设计——当我们以顺序方式访问数据块时，设备达到性能最高点。例如，如果不首先从磁带开端的内容一直向前扫描，我们便无法访问末端的内容。基于实用目的，我们访问的磁带内容局限于那些接近最近被访问过的内容的部分。几种现代技术也有同样的性质。相应地，在这一节里我们专注于讨论顺序读写大型数据块的方法，并默认以下假设：对于我们所关注的机器和设备，可以达到这类数据访问的快速实现。

在读取或写入许多不同文件的过程中，我们假设它们处于不同的外部存储设备上。对于陈旧的机器，文件被存储在外部挂接的磁带上，所以这个假设是绝对的必要条件。当涉及到磁盘时，仅使用单一外部设备实现我们讨论的算法也是可能的，但一般来说使用多个设备将高效得多。

如果有人计划实现一个高效程序来对大型文件排序，第一步也许是实现一个高效程序，以便为该文件生成一个副本。第二步也许是实现一个程序，使文件的顺序逆转。在实现一次外部排序的过程中，面对在解决上述问题时所产生的任何困难都必须得到解决（排序也许必须做这两步的其中之一）。使用一个抽象模型的目的在于：它允许我们把实现问题从算法设计问题中分离出来。

我们分析的排序算法被组织成为很多次对全部数据的处理过程；而我们常常直接通过计算这些过程的次数，来评估一种外部排序方法的开销。在典型情况下，我们所需的处理过程的次数相对较少——也许是十次，或者更少。这个事实意味着：即使消除的仅仅是一次过程，也可以极大地改善性能。我们的基本假设是：外部排序方法的运行时间由输入和输出决定；因此我们可以通过把它使用过程的次数乘以读写整个文件所需的时间，估算出一个外部排序方法的运行时间。

总而言之，我们将对外部排序使用的抽象模型涉及到一个基本假设：被排序文件过大而不能一次加载到计算机的主内存中。它还取决于两种其他资源：运行时间（处理数据的过程次数）和可供使用的外部设备数。我们假设拥有：

- $N$ 条保存于一外部设备上等待排序的记录，
- 内存中容纳 $M$ 条记录的空间，
- $2P$ 个排序期间使用的外部设备。

我们为包含输入数据的外部设备分配标号 0，为其他的分配标号 1、2、…、$2P-1$。排序的目标是把这些记录以排序后的顺序重新存入设备 0。我们将会看到，在 P 和总计运行时间之间必须取得一个折中方案——我们关注的是把折中方案量化，以便于比较不同的可供选用的策略。

这个理想化模型可能是不现实的，其原因有很多。尽管如此，它和任何优秀的抽象模型一样，的确抓住了这种情况的核心方面；它提供了一个精确的框架，使我们可以在其中探索算法思想——许多思想在实际环境中都具有直接的利用价值。

大多数外部排序方法都使用以下的一般策略。对被排序的文件进行第一遍处理，把它分解成大小等于内存容量的多个数据块，对这些块排序。然后把排序后的块归并到一起，有必要的话多执行几次这样的从头到尾处理文件的过程，从而建立连续的、较大的有序数据块，直到整个文件都完成排序为止。这种方法称为“排序-归并”（sort-merge）。自从 20 世纪 50 年代计算机首次在商业应用中得到广泛使用以来，该方法一直得到有效的应用。

最简单的“排序-归并”策略称为“平衡多路归并”（balanced multiway merging），图 11.12 阐明了这种策略。它由一个“初始分布”（initial distribution）处理和接着的几个“多路归并”（multiway merging）处理组成。

在“初始分布”处理中，我们把输入均匀地分配到外部设备 $P$、$P+1$、…、$2P-1$ 之间，每个设备得到的都是 $M$ 条记录组成的有序的数据块（最后的数据块可能例外，如果 $N$ 不是 $M$ 的倍数的话，它将小于前面的块）。我们可以轻易地执行这个分配——从输入读取前 $M$

条记录，排序后把排序块写入设备 $P$；再从输入读取下 $M$ 条记录，排序后把排序块写入设备 $P+1$……如此类推。万一到达设备 $2P-1$ 后还有更多的输入（也即 $N > PM$），我们就把下一个排序块写入设备 $P$，把下一个排序块写入设备 $P+1$……以此类推。我们不断地重复这种处理，直到没有更多的输入为止。完成分布后，每个设备上的排序块数量是 $N/M$ 的舍入值或仅次于它的下一个整数。若 $N$ 为 $M$ 的倍数，则所有排序块的大小均为 $N/M$（否则，除了最后一块外所有排序块的大小均为 $N/M$）。对于比较小的 $N$，也许数据块数量小于 $P$，那么一个或更多的设备为空。

```
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 · $

A O S · D M N · A E X · F H T · E R V · $
I R T · E G R · L M P · O R T · C E O · $
A G N · G I N · E I W · F I Y · O R S · $

A A G I N O R S T · F F H I O R T T Y · $
D E G G I M N N R · C D E E O R R S V · $
A E E I L M P W X · $

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 · $
C D E E F F H I O O R R R S T T V Y · $
· $

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 · $
```

**图 11.12　平衡 3 路归并范例**

在“初始分布”处理中，我们从输入取出元素 A S O，对它们排序，把有序序列 A O S 保存到第一个输出设备；接着，从输入取出元素 R T I，对它们排序，把有序序列 I R T 保存到第二个输出设备。在输出设备之间不断循环执行这个步骤，我们最终得到 15 个序列，每个输出设备上有 5 个。在第一阶段的归并中，我们归并 A O S、I R T 和 A G N，得到 A A G I O R S T，保存在第一个输出设备；然后我们归并输入设备上的第二组序列，得到 D E G G I M N N R，保存在第二个输出设备……以此类推，并再一次使数据均衡地分布在三个设备上。我们再执行 2 遍归并过程即可完成排序。

在第一遍多路归并的过程中，我们把设备 $P$ 到设备 $2P-1$ 看作输入设备，把设备 0 到设备 $P-1$ 看作输出设备。我们执行 $P$ 路归并，把输入设备上大小为 $M$ 的排序块归并成大小为 $PM$ 的排序块，然后以尽可能均衡的方式把它们分配到输出设备上。首先，我们把每个输入设备的第一块归并起来，把结果保存在设备 0 上；然后，我们把每个输入设备的第二块归并起来，把结果保存在设备 1 上……以此类推。到达设备 $P-1$ 后，我们把下一个排序块写入设备 0，把下一个排序块写入设备 1……以此类推。我们不断地重复这个过程，直到处理完全部输入为止。完成这个分配后，每个设备上的排序块数量是 $N/(PM)$的舍入值或仅次于它的下一个整数。若 $N$ 为 $PM$ 的倍数，则所有排序块的大小均为 $PM$（否则，最后一块会较小）。如果 $N$ 不大于 $PM$，最后将只剩下一个排序数据块（位于设备 0）。整个过程也就完成了。

否则，我们重复上述过程，进行第二次多路归并：把设备 0、1、…、$P-1$ 看作输入设备，把设备 $P$、$P+1$、…、$2P-1$ 看作输出设备。我们执行 $P$ 路归并，使输入设备上大小为 $PM$ 的有序块归并成为大小为 $P^2M$ 的有序块，然后把它们分配到输出设备上。若 $N$ 不大于

$P^2M$，第二遍处理以后我们就完成排序了（结果保存在设备 $P$）。

在设备 0 到 $P-1$、设备 $P$ 到 $2P-1$ 之间来回不断重复上述过程，我们就通过 $P$ 路归并使数据块大小每次增加 $P$ 倍，直到我们最终得到仅仅一个数据块为止——结果位于设备 0 或设备 $P$。每次处理的最后归并不一定是完全 $P$ 路归并，否则这个过程是彻底平衡的。图 11.13 仅使用数字和序列的相对大小描述了该过程。我们通过以下操作估算归并的开销：执行本表标明的乘法运算，对结果求和（不包括底部一行的项），再除以序列的初始数量。这个计算以对数据进行的处理次数的形式，给出了所需开销。

| 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 | | | | | |

**图 11.13 平衡三路归并的序列分布**

对于一个 15 倍于内存大小的文件的平衡三路归并，在其“初始分布”处理中我们分别把相对大小为 1 的 5 个序列放入设备 3、4、5，使设备 0、1、2 为空。在归并的第一阶段我们分别把相对大小为 3 的 2 个序列放入设备 0、1，把相对大小为 3 的 1 个序列放入设备 2，使设备 3、4、5 为空。然后我们归并设备 0、1、2 上的序列，把它们重新分配到设备 3、4、5……以此类推，直到只剩下一个序列（位于设备 0）为止。被处理的记录总数是 60，每次过程处理 15 条记录。

为了实现 $P$ 路归并，我们可以使用一大小为 $P$ 的优先队列。我们希望反复地输出最小的元素，但不是从每个将被归并的 $P$ 排序数据块中输出，然后用提供该元素的块中其次最小的元素取代原先的元素输出。为了完成这个操作，我们把设备索引保存在优先队列中，并有一个操作符 < 函数，它从指明的设备中读取将被读取的下一条记录的关键字值（它还提供了一个大于记录中所有关键字的标志值，以表明已到达数据块的末尾）。那么归并就成为一个相当简单的循环了：它从拥有最小关键字的设备读取下一条记录，把最小记录写入到输出，然后在优先队列中用同一设备的下一条记录代替该记录。如此不断重复本过程，直到一个标志关键字是优先队列中最小值为止。我们可以使用一个堆的实现方法，使优先队列所需时间正比于 $\log P$；但通常 $P$ 是如此的小，以至于这部分开销和写入外部存储器的开销相比显得微不足道。在我们的抽象模型中，我们忽略优先队列的开销，并假设能够高效地对外部设备数据进行顺序访问，从而我们可以通过计算处理数据过程的次数来衡量运算时间。在具体实践中我们可以用一个基本的优先队列实现，把我们的编程焦点集中到以下一点：保证外部设备以最高效率运行。

**性质 11.4** *当外部设备数量为 $2P$ 且内存空间足以容纳 $M$ 条记录时，一个基于 $P$ 路平衡归并的“排序-归并”过程需要执行的处理次数为：$1+\lceil \log_P(N/M) \rceil$。*

初始分布需要一次处理过程。若 $N = MP^k$，则在第一次归并后所有块的大小都是 $MP$，第二次后是 $MP^2$，第三次后是 $MP^3$……以此类推。在 $k = \log_P(N/M)$ 次过程后，排序完成。则，若 $MP^{k-1} < N < MP^k$，由于受未完成块和空块的影响，处理过程接近结束时块的大小将改变，但我们依然可在 $k = \lceil \log_P(N/M) \rceil$ 次处理后完成排序。■

例如，如果我们打算用 6 个设备和足够容纳 1 百万条记录的内存对 10 亿条记录进行排序，我们可以通过一个三路“排序-归并”达到目的。总共需要 8 次对全部数据的处理过程：一次分配过程，和 $\lceil \log_3 1000 \rceil = 7$ 次归并过程。我们在分配处理后将有 1 百万条记录组成的有

序序列，在第一遍归并处理后将有 3 百万条记录组成的有序序列，在第二遍归并处理后将有 9 百万条记录组成的有序序列，在第三遍归并处理后将有 27 百万条记录组成的有序序列……以此类推。我们可以估算到：排序文件所需的时间大约是复制文件所需的时间的 9 倍。

在实际的“排序-归并”中，我们要做出的最重要的决定是对 $P$ 值（也就是归并的阶数）的选择。在我们的抽象模型中我们只限于顺序访问，这意味着 $P$ 必须是可供使用的外部设备数量的一半。对于许多外部存储设备来说，本模型是一个接近现实的模型。然而对于其他很多设备而言，非顺序访问是可能的——只不过它要比顺序访问付出更昂贵的代价。只要一些设备可用于这种排序，非顺序访问也许是不可避免的。在这样的情况下，我们依旧可以使用多路排序，但我们将必须考虑到基本的性能权衡问题——增加 $P$ 将减少处理过程的次数，但会使非顺序访问（访问速度较慢）的数量增加。

### 练习

▷ 11.35　以图 11.12 所示图例的风格，说明如何通过平衡三路归并，完成对以下关键字的排序：EASY QUESTIONWITHPLENTYOFKEYS。

▷ 11.36　假设我们使使用的外部设备数量加倍，那么将对多路归并中处理过程的次数产生怎样的影响？

▷ 11.37　假设我们使可用的内存容量增长 10 倍，那么将对多路归并中处理过程的次数产生怎样的影响？

● 11.38　为外部输入输出开发一个接口，该接口涉及到的顺序传输的数据块来自非同步操作的外部设备（或者可从你的系统中一个现存设备获取其中细节）。用这个接口实现 P 路归并，要求在把 P 路输入文件和原输入文件安排到不同的输出设备的同时，使得 $P$ 值尽可能大。将你的程序的运行时间和逐个把文件复制到输出所需的时间进行对比。

● 11.39　使用练习 11.38 所得接口编写一个程序，使得对于你的系统中可能的最大文件，该程序将这个文件的顺序逆转。

● 11.40　你如何对一个外部设备上的所有记录执行一遍完全混洗？

● 11.41　对于多路归并，以下列算法为核心开发一个开销模型：算法能够在同一设备上的一个文件和另一个文件之间切换，所需开销是固定的，且远远高于一次顺序读取的开销。

●● 11.42　对根据快速排序或 MSD 基数排序方式进行的划分操作，开发一个以它为基础的外部排序方法，进行分析并把它和多路归并作比较。你可以像我们在这一节对“排序-归并”的描述中所做的那样，使用一个高层抽象；但你应该力争根据一给定数量的设备和一给定数量的内存，预测运行时间。

11.43　你如何对一个外部设备的内容排序，若（除内存外）没有其他可供使用的设备？

11.44　你如何对一个外部设备的内容排序，若只有一个附加设备（以及内存）可供使用？

## 11.4　“排序-归并”的实现

11.3 节中概述的一般“排序-归并”策略在实践中是有效的。在本节中，我们将讨论两种能够减低开销的改善方法。第一种技术是“替换选择”（replacement selection），它对于运行时间的作用等同于增加我们使用的内存容量；第二种技术是“多阶段归并”（polyphase

merging），它对于运行时间的作用等同于增加我们使用的设备数量。

在 11.3 节我们讨论了对 $P$ 路归并使用优先队列，但说明过一点：$P$ 值太小以至于快速的算法改良是无足轻重的。然而，在“初始分布”的处理中，我们可以充分利用快速优先队列，以生成比内存能够容纳的长度更长的有序序列。这个思想是：把（未排序的）输入传递通过一个巨大的优先队列，像前面提到的那样，总是向优先队列写入最小的元素，而且总是用输入中的下一个元素取代它。但有一个附带条件——如果这个新元素比最近输出的一个小，那么我们标记它为下一个数据块的一个元素，并把它视为大于当前数据块的所有元素，这是因为它不可能成为当前排序块的一部分。当一个已标记元素到达优先队列的顶端时，我们就开始处理一个新数据块。图 11.4 描述了操作中的方法。

**性质 11.5** 对于随机关键字，替换选择产生的序列大约是使用的堆的大小的两倍。

如果我们打算采用堆排序产生初始序列，我们将向内存填入记录，然后逐个把它们写出，不断这样做直到堆变空为止。然后我们向内存填入另一批记录，并再三重复这个过程。平均计算，在这个过程中堆只占用内存的一半。相比之下，替换选择使内存保持充满同样数据结构的状态，因此它的性能达到前者的两倍也就一点都不奇怪了。这个性质的详尽证明需要一个复杂的分析（参见“第三部分参考资料”），然而这个性质却很容易通过实验得到证明（参阅练习 11.47）。■

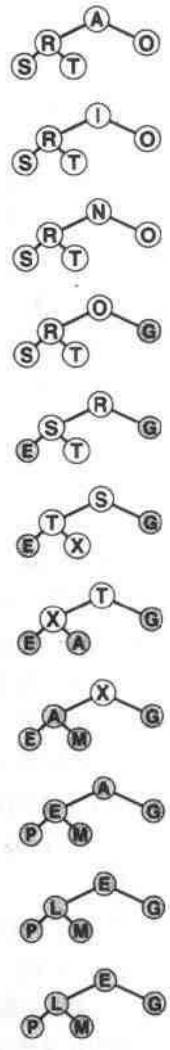


**图 11.14 替换选择**

本系列图例说明了我们怎样通过一个大小为 5 的堆，从序列 A S O R T I N G E X A M P L E 分别生成以下两个长度为 8 和 7 的序列：A I N O R S T X 和 A E E G L M P。

对于随机文件，替换选择的实际作用是或许能节省一次归并处理过程。我们不必从大小约等于内存容量的有序序列开始，然后进行一次归并处理以产生较长序列；我们可以立即从大小约等于内存容量两倍的序列开始。对于 $P=2$，本策略将恰好节省一次归并处理过程；对于更大的 P，其作用就没有这么显著。然而我们知道：实际排序极少涉及随机文件；而如果关键字之间存在某些次序，那么使用替换选择可以产生巨大的序列。例如，如果文件中没有“在它之前存在 $M$ 个以上较大的关键字”的关键字，那这个文件便可以通过替换选择处理被完全排序，而且根本不需要归并！这种可能性正是使用替换选择的最主要的实际原因。

平衡多路归并的主要缺点是只有大约一半设备在归并过程中得到充分应用，它们包括：$P$

个输入设备和任何一个汇集输出的设备。一种备选方案是始终进行（2*P*-1）路归并，把所有输出送到设备 0 上，然后在每次归并处理结束时把数据重新分配到其他磁带上。但这种途径并不能取得更高效率，因为实际上它为了进行分配，使处理的次数增加一倍。平衡多路归并似乎需要以下两者之一：过多的磁带设备或过多的复制操作。人们已经创造出几种智能算法，保持所有外部设备处于工作状态，方法是改变小排序块被归并到一起的途径。这些方法中最简单的一种称为“多阶段归并”。

多阶段归并背后的基本思想是：把替换选择产生的排序块稍稍不平均地分配到可用的磁带设备之间（留空其中一个），然后采用一种“归并直到空”（merge-until-empty）策略。这种策略是：既然被归并的磁带长度不一，必然有一个比其他的更早完成处理，那么就可以把它用作输出。也就是说，我们切换输出磁带（当前有一些排序块在其上）和当前为空的输入磁带所担当的角色，不断继续这个过程直到只剩下一块为止。图 11.15 描述了一个范例。

```
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 • $

A O S • D M N • A E X • F H T • $
I R T • E G R • L M P • O R T • E R V • D R S • $
A G N • G I N • E I W • F I Y • C E O • • • $

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 • $
E R V • D R S • $
C E O • • • $

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 • $
A E E I L M P W X • F F H I O R T T Y • $
• $

D D E G G I M N N R R S • $
F F H I O R T T Y • $
A A A C E E E E G I I L M N O O P R R S T V W X • $

A A A C D D E 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 • $
```

**图 11.15　多阶段归并范例**

在初始分布阶段，我们依据一个预先安排好的计划，把不同数量的序列安置到各个磁带上，而不是像我们在图 11.12 所做的那样——保持序列数量的平均。然后我们在每个阶段进行三路归并，直到完成排序为止。这样会比平衡归并有更多阶段，但这些阶段并不涉及到全部数据。

“归并直到空”策略适用于任意数量的磁带，如图 11.16 所示。归并被分解为许多阶段（phase），其中并不是所有阶段都涉及到全部数据。而且无需额外的复制操作。图 11.16 显示了如何计算初始的序列分布。我们通过回溯处理，计算每个设备的序列数量。

对于图 11.16 描述的例子，我们的推论如下：我们希望完成归并、得到一个位于设备 0 的序列。因此，就在最后一次归并之前我们希望设备 0 为空，而在设备 1、2 和 3 上各有 1 个序列。接下来，我们推断出倒数第二次归并之前我们所需的序列分布，正是这种分布情况使归并操作产生当前的这个序列分布。设备 1、2 和 3 中的一个必须是空的（使得它可以成为倒数第二次归并的输出设备），我们任意地选出设备 3。也就是说，倒数第二次归并把来自设备 0、1 和 2 的各 1 个序列归并起来，把结果保存到设备 3。因为倒数第二次归并完成后

在设备 0 留下 0 个序列，在设备 1 和 2 留下各 1 个序列，所以它开始时的情况必然是：在设备 0 中有 1 个序列，在设备 1 和 2 中各有 2 个序列。类似的推理告诉我们：在前一个归并处理开始时，在设备 3、0 和 1 中分别有 2、3 和 4 个序列。以这种方式继续推理下去，我们便可以建立序列分布表了：取出每一行中最大的数字，置零，并把它加入到每个其他数字中，得到先前一行。这个约定对应于为前一行定义出能够给出当前行的最高次序归并。本方法适用于任何数量的磁带（至少 3 个）。由此产生的数字组成了广义斐波纳契数列（generalized Fibonacci numbers），它具有许多有趣的性质。若序列数目不是斐波纳契数，我们可假设存在哑元序列，使初始序列数目严格等于该表格要求的数。而实现一个多阶段归并的主要挑战就是确定如何分布初始序列（参阅练习 11.54）。

给定了序列分布后，我们就可以通过回溯处理（跟踪记录归并产生的序列长度），计算序列的相对长度。譬如，图 11.16 的范例中的第一次归并产生 4 个位于设备 0、相对大小为 3 的序列，在设备 2 留下 2 个相对大小为 1 的序列，在设备 3 留下 1 个相对大小为 1 的序列，以此类推。正如我们在平衡多路归并中所做的那样，我们可以执行已指明的乘法运算，对结果求和（不包括底部的一行），再除以初始序列的数量；由此可得到一个开销估算，它相当于对所有数据执行一遍完整处理的开销的乘积。为简单起见，我们在开销计算中包括了哑元序列——它为我们给出了关于真实开销的一个上界。

**性质 11.6** *当外部设备数量为 3 且内存空间足以容纳 $M$ 条记录时，若一个“排序-归并”过程基于替换选择及其后的一个二路多阶段归并，则所需执行的有效处理次数的平均值约为：* $1+\lceil \log_\phi (N/2M) \rceil / \phi$。

Knuth 和其他研究人员在 20 世纪 60 年代和 70 年代完成了关于多阶段归并的全面分析，这个分析复杂而广泛，也超出了这本书的范围。对于 $P=3$，该算法涉及到斐波纳契数，因此$\phi$也出现了。更大的 P 将对应产生其他常数。因子 $1/\phi$说明了这样一个事实：每个阶段仅涉及数据的一部分。我们计算“有效处理”次数的公式是：读入的数据量除以总数据量。一些对这方面的全面研究结果

| 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 | | | |

**图 11.16 多阶段三路归并的序列分布**

本图例是对一个 17 倍于内存容量的文件进行的多阶段三路归并。在它的初始分布阶段，我们把 7 个序列置于设备 0，4 个序列置于设备 2，6 个序列置于设备 3。然后在第一阶段，我们进行归并直到设备 2 为空，并保留 3 个大小为 1 的序列在设备 0 上，2 个大小为 1 的序列在设备 3 上，并在设备 1 上创建 4 个大小为 3 的序列。对一个 15 倍于内存容量的文件，我们在开始时把 2 个哑元序列置于设备 0（参阅图 11.15）。整个归并处理的数据块总数是 59，这比我们的平衡归并范例（参阅图 11.13）少 1，但我们可少用 2 个设备（参阅练习 11.50）。

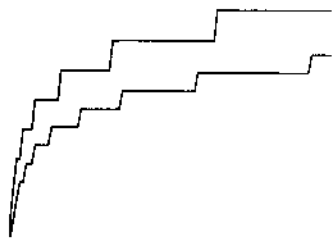


**图 11.17 平衡归并和多阶段归并的开销比较**

在具有 4 个磁带设备的平衡归并中，所需的处理次数（上一条线）总是大于在具有 3 个磁带设备的多阶段归并所需的处理次数（下一条线）。这些曲线根据性质 11.4 和 11.6 的函数画出，其中 N/M 的范围从 1 到 100。因为要考虑哑元序列，多阶段归并的真实性能比这里的阶梯曲线所指出的要更加复杂。

是相当惊人的。例如，在磁带之间分配哑元序列的最优方法涉及使用额外的阶段和更多的哑元设备，而且其数量似乎大于我们先前认为所需的数量。其原因是某些序列的使用在归并中要比其他序列更为频繁（参见“第三部分参考资料”）。■

例如外部设备数量为 3 且内存空间足以容纳 1 百万条记录时，如果我们要排序 10 亿条记录，我们可以通过二路多阶段归并，以$\lceil \log_\phi 500 \rceil / \phi = 8$次处理达到目的。把分布处理也计算在内的话，我们需要付出的开销仅比一次平衡归并多一点点（一次处理），但后者使用的设备多一倍。也就是说，我们可以这样想：多阶段归并允许我们以一半数量的设备完成同样的工作。对于给定数量的设备，多阶段归并始终比平衡归并更高效（如图 11.17 所示）。

正如我们在 11.3 节开始部分讨论的那样，我们把注意力集中到一个具有顺序访问外部设备能力的抽象机器上，因而能够分离算法问题和实际使用问题。开发实际应用时，我们必须检验我们的基础假设并保证它们依然有效。例如，我们依赖于输入输出函数的高效实现（它们在处理器和外部设备之间传输数据）和其他系统软件。现代计算机系统一般都具备了对这些软件经过悉心调试的具体实现。

让我们把这个观点延伸到一个极端。要知道很多现代计算机系统都提供了一个容量巨大的虚拟内存（virtual memory）——这是一个比我们一直以来使用的访问外部存储器的模型更广义的抽象模型。在虚拟内存中我们能够对极大数量的记录进行寻址，而无需负责确保被寻址数据在必要时从外部存储器传送到内存中，这个任务交给了系统；我们对数据的访问看起来就和直接访问内存一样方便。但这个幻象并非完美的：只要程序访问的内存地址相对接近最近被访问的地址，那么就很少需要进行外部存储器到内存的传输，这样虚拟内存的性能是不错的（例如顺序访问数据的程序就属于这一类）。然而，如果程序的内存访问比较分散，那么虚拟内存系统的性能可能将急速下滑（把所有时间都花费在访问外部存储器上），产生的结果也是灾难性的。

虚拟内存可作为排序大型文件的一种备选方案，我们不应忽略这一点。我们可以直接实现“排序-归并”，或者使用一种诸如快速排序或归并排序的内部排序方法，这样更为简单。在一个优良的虚拟内存环境中，这些内部排序方法是值得我们认真考虑的。而访问地址分散于内存各处的方法——例如堆排序或基数排序——则似乎不太适合，因为系统性能将急速下滑。

另一方面，使用虚拟内存可能牵涉过多的系统管理开销：这样我们必须改为依靠自己进行处理了——要最大限度地利用高性能外部设备，明确的方法（例如我们已经讨论过的那些）也许是最好的。我们一直以来分析的方法的一个特征是：它们被设计成可以令计算机系统中尽可能多的独立部件以最大效率运转，不让任何部件空闲。当我们把独立部件看作处理器本身时，我们就进入了 11.5 节的主题——并行计算。

### 练习

▷ 11.45　对于关键字 E A S Y Q U E S T I O N，给出由替换选择产生的序列。优先队列大小为 4。

○ 11.46 对一个给定文件使用替换选择而生成的文件，若再次使用替换选择，将产生什么影响？

● 11.47 对大小分别为 $N=10^3$，$10^4$，$10^5$ 和 $10^6$ 的随机文件，通过实验确定由优先队列大小为 1000 的替换选择产生的序列平均数。

11.48 当你在一个含有 N 条记录的文件中使用替换选择生成初始序列，而采用的优先队列大小为 $M$（$M<N$）时，在最差情况下序列数量是多少？

▷ 11.49 说明关键字 EASYQUESTIONWITHPLENTYOFKEYS 怎样通过多阶段归并进行排序，采用图 11.15 图解范例的风格。

○ 11.50 在图 11.15 的多阶段归并例子中，我们向 7 个序列的磁带设备加入 2 个哑元序列。考虑向磁带设备分配哑元序列的其他方案，并找出归并开销最小的一个。

11.51 画出一张对应图 11.13 的表，以确定对全部数据进行 5 次处理（使用 6 个设备）时，可由平衡二路归并方法归并的序列的最大数量。

11.52 画出一张对应图 11.16 的表，以确定其开销等同于对全部数据进行 5 次处理（使用 6 个设备）时，可由多阶段归并方法归并的序列的最大数量。

○ 11.53 编写一个程序，对于一给定数量的设备和一给定数量的初始块，分别计算多路归并的处理次数和多阶段归并的有效处理次数。对 $P=3$，4，5，10 和 100，$N=10^3$，$10^4$，$10^5$ 和 $10^6$，使程序打印一张关于每种方法所需开销的表。

●●11.54 编写一个程序，为 $P$ 路多阶段归并把初始序列顺序地分配到各设备上。每当序列数是一个广义斐波纳契数时，应按照算法的要求把序列分配到各设备上。你的任务是找出一个便捷的方法，每次分配一个序列。

● 11.55 使用练习 11.38 定义的接口实现替换选择。

●●11.56 结合你对练习 11.38 和练习 11.55 的解决方案，产生一个“排序-归并”的具体实现。利用多阶段归并，用你的程序对系统中尽可能大的文件进行排序。可能的话，确定增加设备数量对运行时间的影响。

11.57 对于一个在虚拟内存环境中处理大型文件的快速排序，怎样用它处理较小型的文件？

● 11.58 如果你的计算机有一个合适的虚拟内存系统，对于大型文件通过实验比较快速排序、LSD 基数排序、MSD 基数排序和堆排序。使用尽可能大的文件。

● 11.59 为基于 $k$ 路归并的递归多路归并排序开发一个实现方案，使它适用于在一个虚拟内存环境中排序大型文件（参阅练习 8.11）。

● 11.60 如果你的计算机有一个合适的虚拟内存系统，通过实验确定 $k$ 值，使你对练习 11.59 的实现方案有最小的运行时间。使用尽可能大的文件。

## 11.5 并行“排序-归并”

我们怎样可以让几个独立的处理器共同协作，处理同一排序问题呢？“到底是这些处理器，还是整个计算机系统控制外部存储设备？”这个问题是为高性能计算机系统进行算法设计的核心。最近的一些年里，并行计算的主题得到了十分广泛的研究。人们已经发明了许多不同类型的并行计算机，并提出了许多不同的并行计算模型。而排序问题就是检验两者有效性的一种测试实例。

在 11.2 节对排序网络的讨论中，我们已经分析过低层次的并行操作，当时我们考虑的是在同一时间执行多个“比较-交换”操作。现在我们将讨论高层次的并行模型——将有大量独立的通用处理器（而不仅仅是比较器）可访问同样的数据。我们将再一次忽略很多实际问题，但可以在这个环境中分析算法问题。

我们对并行处理所使用的抽象模型涉及一个基本假设：被排序的文件分布于 $P$ 个独立处理器之间。假设我们拥有：

- $N$ 条等待排序的记录。
- $P$ 个处理器，每个能够控制 $N/P$ 条记录。

我们为处理器分配标号 0、1、…、$P$-1，并假设被输入的文件位于处理器的局部存储器上（也就是说，每个处理器有这些记录的 N/P）。排序的目标是重新排列这些记录，使最小的 $N/P$ 条记录位于处理器 0 的存储器，次小的 $N/P$ 条记录位于处理器 1 的存储器……以此类推，且记录都有序。我们将看到，$P$ 和总计运行时间之间相互制约——我们感兴趣的是把这个权衡问题量化，使我们可以比较可供选择的不同策略。

这个模型是针对并行机制的很多可能方案之一，至于实际应用上，它有很多和我们对外部排序使用的模型（参阅 11.3 节）一样的不足之处。的确，它没有涉及其中一个在并行计算中要面对的最重要的问题：处理器间通信的制约。

我们将假设这些通信所需开销远远高于对局部存储器的访问开销，以至于最高效的处理形式是以大数据块的格式顺序进行处理。在某种意义上，处理器把其他处理器的局部存储器看作为外部存储设备。这里必须再一次说明：从实际观点出发，可以认为这个高层次抽象模型是不尽如人意的，因为它过分简化了情况；从理论观点出发，也可以认为它是不尽如人意的，因为它没有作出完全的规定。但它依然提供了一个框架，使我们可以在其中发展有用的算法。

这个问题（和这些假设）的确为抽象的威力提供了一个令人信服的范例。因为我们可以通过修改“比较-交换”操作，使它作用于大型数据块，而使用和 11.2 节中讨论的相同的排序网络。

**定义 11.2**　*归并比较器接收两个大小为 $M$ 的有序文件作为输入，并产生两个有序文件作为输出：一个含有 $2M$ 个输入中最小的 $M$ 个，而另一个含有 $2M$ 个输入中最大的 $M$ 个。*

这样的操作很容易实现：归并两个输入文件，并输出归并结果的前半部分和后半部分。

**性质 11.7**　*我们可以由以下步骤对一个大小为 $N$ 的文件排序：把它划分为 $N/M$ 份大小为 $M$ 的文件块，对每一块排序，然后使用一个内建归并比较器的排序网络。*

由 0-1 法则确定这个性质需要很高的技巧（参阅练习 11.61），但追踪分析一个像图 11.18 这样的例子却是一次很有说服力的练习过程。■

我们把性质 11.7 描述的方法称为块排序（block sorting）。在我们在一个特定的并行机器上使用这个方法以前，我们有很多设计参数要考虑。我们对该方法的关注涉及以下性能特性：

**性质 11.8** 结合使用巴彻尔排序和归并比较器时，在 $P$ 个处理器上执行的块排序，可以在大约 $(\lg P)^2/2$ 个并行步骤中排序 N 条记录。

对于这段文字中的并行步骤（parallel step），我们的意思是一组分离归并比较器的集合。性质 11.8 就是性质 11.3 和 11.7 的直接推论。■

要在两个处理器上实现一个归并比较器，我们可以用以下方法：让它们交换各自数据块的副本，并且都进行归并（以并行方式），然后其中之一保存关键字较小的半部分，另一个保存关键字较大的半部分。如果数据块传输速度和单个处理器速度相比较慢，我们可以把每次数据块传输的开销乘以 $(\lg P)^2/2$，从而估算排序所需的时间总和。这个估算包含了大量假设：例如，它假定多个数据块传输能够同步完成且没有性能损失——这是一个在现实并行计算机中极少能够实现的目标。但对于理解我们在实际实现中所预期的目标，它仍提供了一个很好的出发点。

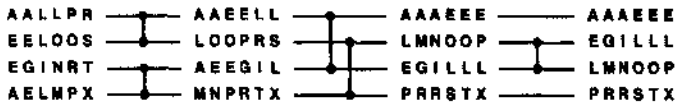



**图 11.18 块排序范例**

本图说明了我们可以怎样利用图 11.4 的网络对数据块进行排序。对位于两条输入线的元素，比较器把较小的半部分输出到顶部线上，把较大的半部分输出到底部线上。三个并行步骤就足以完成这个任务了。

若块传输的开销和单个处理器速度处于可比较的水平（这是另一个在实际机器中只能尽力接近、未能实现的理想化目标），那么我们就必须考虑执行初始排序的时间。每个处理器（并行地）做大约 $(N/P)\lg(N/P)$ 次比较，以此对 $N/P$ 个块初始排序；对 $(N/P)\times(N/P)$ 的归并也需要大约 $P^2(\lg P)/2$ 个阶段。如果一次比较的开销是 $\alpha$，且一次归并的每条记录的开销是 $\beta$，那么合计的运行时间大概是：

$$\alpha(N/P)\lg(N/P)+\beta(N/P)P^2(\lg P)/2$$

对极大的 $N$、极小的 $P$，这将是我们在任何基于比较操作的并行排序方法中可以预期得到的最好性能；因为这种情况的开销大约是 $\alpha(N\lg N)/P$，这已是最佳结果了。任何排序都需要 $N\lg N$ 次比较，我们所能实现的最佳情况是一次处理其中的 $P$ 个。对于较大的 $P$，公式的第二项起决定作用，而开销大约是 $\beta N(P\lg P)/2$，这是次优的但也许相当有竞争力的方案。例如在 64 个处理器上排序 10 亿个元素，第二项在所需开销中大约占了 $256\beta N/P$，相比之下第一项占了 $32\alpha N/P$。

当 $P$ 比较大时，某些机器上所有处理器之间的通信也许会产生一个瓶颈。如果事实的确如此的话，则使用一个图 11.8 里的完全混洗也许能提供一种控制这些开销的方法。一些并行计算机有内建的低层互连机制，可允许我们高效地实现混洗，也正是由于这个原因。

这个例子说明了在一定环境下，我们可以使大量处理器高效地处理一巨大的排序问题。要找到这种实现的最佳途径，我们必定需要考虑有关这类并行机器的许多其他算法，需要学习有关一台实际并行机器的许多其他特性，还需要考虑我们使用的机器模型的许多演化

情况。而且，我们也许需要采用一种完全不同的途径来实现并行机制。尽管如此，以下思想仍是并行计算的基础：增加处理器数量将增加它们之间的通信开销。而巴彻尔网络提供了一种控制这些开销的有效途径，我们在 11.2 节中从一个低层次的角度看到了这点，在本节中从一个高层次的角度也看到了这点。

在这一节和本章其他部分描述的排序方法具有一种特别的风格，它不同于我们在第六章到第十章讨论过的方法所具有的风格，因为它们涉及到应付我们在普通编程中不会考虑的制约。在第六章到第十章中，关于我们的数据性质所作的假设，已足以允许我们比较对应同一基本问题的大量不同方法了。相比之下，本章中我们把焦点集中到清晰说明多种问题，并为每个问题讨论一些解决方案。这些例子说明了现实世界制约的改变可以为算法解决方案提供新的机遇，而这个过程的一个关键所在是开发出这些问题的有益的抽象模式。

排序在很多实际应用中是必需的，而在新计算机结构和新编程环境中，一个高效排序的设计常常是首先需要解决的问题之一。就新的发展奠基于过往经验而言，则对我们在这里和第六章讨论过的一系列技术的了解十分重要；就创造出剧烈演化的新分支而言，如果我们要在新型机器上开发出快速排序程序的话，则这里讨论的这类抽象思维也是必需的。

## 练习

○ 11.61　利用 0–1 法则（性质 11.1）证明性质 11.7。

● 11.62　为利用巴彻尔奇偶归并的块排序实现一个顺序版本：（1）利用标准归并排序（程序 8.3 和 8.2）对块进行排序；（2）利用标准抽象就地归并（程序 8.2）实现归并比较器；（3）利用自下而上巴彻尔奇偶归并（程序 11.3）实现块排序。

11.63 把练习 11.62 描述的程序看作一个关于 $N$、$M$ 的函数，对于较大的 $N$ 估算其运行时间。

● 11.64　重做练习 11.62 和 11.63，但在两个实例中都用自下而上巴彻尔奇偶归并（程序 11.3）替代程序 8.2。

11.65　对 $N = 10^3$，$10^6$，$10^9$ 和 $10^{12}$，为等式 $(N/P)\lg N = N P \lg P$ 给出 $P$ 的值。

11.66　对 $P = 1$，4，16，64 和 256，为一个并行巴彻尔块排序使用的数据元素之间的比较次数，给出形如 $c_1 N \lg N + c_2 N$ 的近似表达式。

11.67　要利用 100 个处理器对分布在 1000 个磁盘上的 $10^{15}$ 条记录排序，需要多少个并行步骤？

# 第三部分参考资料

本部分的主要参考书是 Knuth 系列的第三卷，也就是讲排序和检索的那一本。事实上，我们讨论过的每一个问题的详尽的资料都可以在这本书中找到。特别是，本书所讨论的多种算法的性能特征的结论，在这本书中有完整的数学上的分析过程为依托。

关于排序方面的著作非常多。Knuth 和 Rivest 在 1973 年列出的参考书目中包括了几百篇论文，这些都描述了我们所考察的很多经典方法的发展情况。Baeza-Yates 和 Gonnet 的著作则包含了更新的参考书目，里面有涵盖了最近的研究的更多论文。在 Sedgewick 的 1996 年的论文中，我们可以看到 Shell 排序在当前阶段的进展。

对于快速排序来说，最好的参考书籍是 Hoare 在 1962 年的原始论文，其中提出了所有重要的改进方法，包括了第七章中选择来讨论的那些的用途。在 Sedgewick 的 1978 年的论文中，可以看到关于数学上的分析的更多细节，以及在该算法被广泛应用以后提出的许多修正和改进的特定效果。Bentley 和 McIlroy 则给出了这个论题在当前的发展。第七章中的三路分割和第十章的三路基数快速排序则是基于 Bentley 和 Sedgewick 在 1997 年写的论文的。最早的分割形式算法（二分快速排序或基数交换排序）出现在 Hildebrandt 和 Isbitz 在 1959 年所写的论文当中。

由 Brown 实现和分析的 Vuillemin 的二项式队列数据结构，精致而且高效地支持了所有的高级队列操作。由 Fredman、Sedgewick、Sleator 和 Tarjan 描述的堆则是一个特别有趣的改进。

McIlroy、Bostic 和 McIlroy 在 1993 年所写的论文体现了基数排序在实现方面的艺术水平。

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. Sedgewick, "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. Sedgewick, 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. Sedgewick, "Implementing quicksort programs," *Communications of the ACM* 21,10 (October 1978).

R. Sedgewick, "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).

# 第四部分

# 搜索算法

# 第十二章

# 符号表和二叉搜索树

从大量先前存储的数据中重新取出特定的一段或几段信息是一项称为搜索（search）的基本操作，它是大量计算任务中本质性的操作。正如对从第六章到第十一章的排序算法和第九章中特定的优先队列所作处理那样，我们处理的数据划分为记录或数据项（item），每一数据项有一个用于搜索的关键字（key）。搜索的目的是找出带有和某给定搜索关键字（search key）相吻合的关键字的数据项。搜索的意图通常是访问数据项中的信息（而不仅仅是关键字），以进行处理。

搜索的应用范围很广泛，并涉及到许多不同操作。举个例子，想想一家银行，它需要保存所有客户的账户信息，并搜索这些记录以检查账户结余和执行交易。另一个例子是一家航空公司，它需要保存所有航班的预定机位信息，并搜索这些记录以找出空闲的座位，或取消或修改这些预定机位信息。第三个例子是一个网络软件接口的搜索引擎，它在网络中寻找包含某个给定关键字的所有文档。这些应用的要求在某些方面是相似的（银行和航空公司都要求精确性和可靠性），在另一些方面是不同的（银行的数据和其他个体的数据相比有很长的使用寿命），而它们全都需要优良的搜索算法。

**定义 12.1** *符号表是一种由带有关键字的数据项构成的数据结构，它支持以下两种基本操作：插入一个新数据项，返回一具有某给定关键字的数据项。*

符号表有时候也称为字典（dictionary），因为它类似于一个由来已久的系统——通过在一本参考书籍中按字母顺序列出单词，为它们提供定义。在一本英语字典中，“关键字”就是单词，“数据项”就是和单词相关的词条，它们含有定义、读音和其他信息。人们在一本字典中使用搜索算法搜索信息，通常依靠的是“词条以字母顺序出现”这个事实。电话本、百科全书和其他参考书籍本质上都是以同样的方式组织起来的，我们将要讨论的一些搜索方法也依赖于按顺序排列的数据项。

基于计算机的符号表的一个优点是，它们可以具有远远超出一本字典或电话本的动态性，所以我们将讨论的算法大部分都构建这样的数据结构：它们不但允许高效的搜索算法，还支持高效的操作实现，这些操作包括添加新数据项、删除或修改数据项、把两个符号表合并为一个等等。在本章中我们将重新讨论很多问题，它们和我们在第九章讨论的对应优

先队列的操作有关。在计算机科学中，开发支持搜索的动态数据结构是其中一个最古老、研究得最广泛的问题；它将是本章和第十三章到第十六章的主要焦点所在。我们会看到，很多创造性的算法已经（并且正在）被发掘出来，以解决符号表的实现问题。

除了刚刚提到的基本应用类型以外，计算机科学家和程序员们还深入地研究了符号表，因为它们对在计算机系统中组织软件具有不可或缺的辅助作用。一个符号表是一个程序的字典：关键字是程序中使用的符号名称，数据项包含描述被命名对象的信息。从科学计算的早期（当时符号表允许程序员从使用机器代码中的数字地址转向使用汇编语言的符号名称）到新千年的现代化应用（符号名称具有跨越全世界计算机网络的含义），快速搜索算法已经并仍将在计算中扮演一个核心角色。

在低层次抽象中频繁出现的符号表，偶尔也在硬件层次上出现。有时候使用术语相关存储器（associative memory）描述这个概念。我们将集中考虑软件实现，但我们考虑的一些方法也适用于硬件实现。

和我们在第六章中研究的排序方法一样，我们将通过着眼于一些基本方法开始本章对搜索方法的研究。这些基本方法对小型表和其他特殊情况很有用，并展现了更高级方法利用的基础技术。然后，在本章的许多剩余部分中我们将把焦点集中到BST（binary search tree，二叉搜索树）上，这是一个支持快速搜索算法的基本的、得到广泛应用的数据结构。

我们在2.6节讨论了两个搜索算法，以显示数学分析在协助我们开发高效算法的过程中的有效作用。为了本章的完整性，我们会重复部分在2.6节中考虑过的信息，然而为了某些证明我们将往回引用那一节的内容。在本章的稍后部分我们也会引用在5.4和5.5节讨论的二叉树基本属性。

## 12.1 符号表抽象数据类型（ADT）

和优先队列一样，我们把搜索算法看作属于声明许多一般操作的接口，这些操作可以从特定具体实现中分离出来，因此我们能够轻易地用备选的实现方案进行替换。我们关心的操作包括：

- 插入一个新数据项；
- 搜索一个（或多个）具有某给定关键字的数据项；
- 删除一个指定数据项；
- 在一符号表中选择第k大的数据项；
- 排序符号表（按它们的关键字顺序显示所有数据项）；
- 联结两个符号表。

正如我们对待很多其他数据结构那样，我们也许需要向这个集合加入标准的构造（construct）操作、测试是否为空（test if empty）操作，或许还有销毁（destroy）和复制（copy）操作。此外，我们也许还打算考虑有关这个基本接口的其他不同实际修改。例如一个搜索-插入（search-and-insert）操作常常是很有吸引力的，因为对很多实现而言，对一个关键字

的搜索即使不成功，也为插入一个带有该关键字的新数据项给出了所需的信息。

我们一般使用术语“搜索算法”来表示“符号表 ADT 实现”，尽管后者更恰当地意味着为符号表定义和建立一个基本的数据结构，并实现搜索以外的其他 ADT 操作。符号表对于很多计算机应用十分重要，以至于在很多编程环境中它们都作为高层次抽象可直接使用。C 标准库有 bsearch——12.4 节中二叉搜索算法的一种实现，而 C++标准模板库（Standard Template Library）提供了大量符号表，称为相关容器（associative containers）。和往常一样，一种通用实现方法很难符合多种应用苛刻的性能要求。现在已经开发出许多用于实现符号表抽象的创造性方法；我们对它们的研究将设定一个环境，以帮助我们了解预先封装的实现方法的特性，并确定什么时候应该开发一种专用于某种特定应用的实现方案。

和我们对排序的处理一样，我们将考虑具体方法而不指定被处理的数据项类型。我们讨论的实现采用了一种接口，它定义了 Item 和对之执行的基本抽象操作，这和 6.8 节中详细讨论的方式是一致的。我们讨论基于比较的方法和基于基数的方法，后者使用关键字或多个关键字作为索引。为了强调搜索过程中数据项和关键字扮演的个别角色，我们扩展第六章到第十一章中使用的 Item 概念，使得数据项包含有类型 Key 的关键字。因为我们现在要求使用的数据项比讨论排序算法时还要多（一点），我们将假设它们被封装成用 C++类实现的 ADT——程序 12.1 展示了这一点。我们利用成员函数 key()从数据项中提取关键字，并重载操作符==检验两个关键字是否相等。在本章和第十三章中，我们还要重载操作符< 来比较两个关键字的值，以便在搜索中指引我们；在第十四章和第十五章中，我们的搜索算法基于以下方法：用第十章使用的基本基数操作提取多个关键字。我们还假设数据项被初始化为空（null），客户程序可访问一个函数 null()，它能够检验一个数据项是否为空。我们使用空数据项的原因是：当出现符号表中没有数据项具有搜索关键字的情况时，可提供一个返回值。

**程序 12.1　数据项 ADT 的实现范例**

这个类展示了我们对符号表数据项的基本常规，它定义了数据项 item——一由整数关键字和相关的浮点信息组成的小记录。我们的符号表实现是利用操作符==和<比较关键字的客户程序，并分别使用成员函数 key()和 null()访问关键字和检验数据项是否为空。

我们也在类型定义中引入了函数 scan（读取一个 item）、rand（生成一个随机 item）和 show（打印一个 item），以便驱动程序调用。这些安排使我们可以灵活地实现和测试对应不同类型数据项的不同符号表实现方案。

```
#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; }
```

为了对浮点数、字符串和来自 6.8 节、6.9 节的更复杂的数据项使用这些接口和实现方法进行搜索，我们需要做的只是：保证对 Key、key()、null()、操作符==和操作符<有正确的定义可供使用，并把 rand、scan 和 show 修改成为类的成员函数，并正确地引用关键字。

程序 12.2 是一个定义基本符号表操作（联结除外）的接口。在本章和接下来的几章中，我们将在客户程序和所有搜索实现之间利用这个接口。从 4.8 节的意义上来说，我们使用的并非一级 ADT，因为复制构造函数、重载赋值操作符和析构函数的加入会分散我们对算法核心特性的注意力，尽管它对大多数实现方案都是一项直截了当的任务。我们也可以在程序 12.2 中定义该接口的另一版本，以类似于程序 9.8 的方式控制对数据项的处理（见练习 12.7）。但在它足以通过关键字控制数据项的典型情况中，这种方案不必要地使程序复杂化。这个接口没有指出我们如何确定删除哪一个数据项。大多数实现使用了这个解释："删除一个具有和指定数据项相等关键字的数据项"，这隐含地使用了一次搜索操作。其他提供句柄并能够测试数据项一致性的实现方法避免了在删除之前进行搜索的需要，因此可以支持更快的算法。同样，当我们为联结操作考虑算法时（这意味着我们有处理多个符号表的应用），我们也许希望使用一级符号表 ADT 实现，它们被十分谨慎地实现以避免浪费时间或空间（见 12.9 节）。

某些算法不假定关键字之间存在任何隐含的顺序，因此只使用操作符==（不使用操作符<）比较关键字。但是很多符号表实现都利用关键字之间隐含的顺序关系，使用操作符<结构化数据并为搜索提供指引。抽象操作 select 和 sort 也显式地引用关键字顺序。sort 函数

被封装为一个把所有数据项按顺序输送到输出流而无须对它们重新排列的函数。我们可以很轻易地推广 sort，产生一个依照关键字顺序访问数据项的函数，这也许是通过对每数据项应用一个作为参数传递的过程。我们把符号表的 sort 函数命名为 show，这是因为我们提供的实现方案依照排序后的顺序显示符号表的内容。不使用操作符<的算法并不要求关键字可与另一个关键字作比较，也不一定支持 select 和 sort。

**程序 12.2　符号表抽象数据类型**

这个接口为一个简单的符号表定义以下操作：初始化、返回数据项计数结果、寻找一个带有某给定关键字的数据项、添加一个新数据项、删除数据项、选择第 $k$ 个最小的数据项，以及依照其关键字顺序显示数据项（到一个指定输出流）。

```
template <class Item, class Key>
class ST
  {
    private:
      // 与实现相关的代码
    public:
      ST(int);
      int count();
      Item search(Key) ;
      void insert(Item);
      void remove(Item);
      Item select(int);
      void show(ostream&);
  };
```

在一个符号表的实现中，几个数据项具有相同的关键字这个可能性应该得到特别的关注。一些应用禁止重复的关键字，所以关键字可被用作句柄。这种情况的一个例子是在人事档案中把社会保障号码用作关键字。其他应用也许牵涉到许多有重复关键字的数据项，例如，文档数据库里的关键字搜索典型地将产生多个搜索命中（hit）。

我们可以用几种方法之一处理有重复关键字的数据项。一种途径是坚持使主要搜索数据结构只包含特有关键字的数据项，并为每个关键字保持一个链接，它指向一个由具有重复关键字的应用数据项组成的列表。也就是说，我们在主要数据结构中使用含有一个关键字和一个链接的数据项，而没有具有重复关键字的数据项。这种安排在某些应用中很方便，因为一次搜索操作即可返回全部含有某个给定搜索关键字的数据项，或者用一次删除操作即可删除它们。从实现的这个角度来看，这种安排相当于把重复关键字的处理交给客户程序。第二种可能性是在主要搜索数据结构中保留有相同关键字的数据项，对于一次搜索操作返回任何对应某给定关键字的数据项。对于每次只处理一个数据项的应用来讲，这样更为简单，因为这时处理关键字重复的数据项的顺序并不重要。从算法设计的角度来看这也

许不太方便，因为很可能需要对该接口进行扩展，引入一种机制来提取所有含某给定关键字的数据项，或对每个含有该给定关键字的数据项调用一个指定的函数。第三种可能性是假设每个数据项（除了本身的关键字以外）有一个独特的标识号，并且要求一次搜索操作对给定关键字找出含有给定标识号的数据项。又或者我们会需要一些更复杂的机制。当存在重复关键字时，这些考虑适用于所有符号表操作。如果我们希望删除所有含某给定关键字的数据项、含该关键字的任一数据项或者一个指定数据项（这需要一个实现提供对数据项的处理句柄）呢？当描述符号表的实现时，我们非正式地指出怎样可以最简便地处理关键字重复的数据项，但不一定为每种实现都讨论各自的机制。

程序 12.3 是个简单的客户程序，它展示了符号表实现的常用做法。它利用一张符号表找出一串（随机生成或从标准输入设备读入的）关键字里面不同的值，然后按排序后的顺序打印出来。

**程序 12.3 符号表客户程序范例**

本程序利用一张符号表找出一串随机生成或从标准输入设备读入的关键字中不同的值。它对每个关键字都使用 search 来检查之前是否已看到过它。若这个关键字之前没有被看到过，它就把含有该关键字的数据项插入到符号表。在 Item.cxx 中已指定了关键字和数据项的类型，以及对它们执行的抽象操作（在程序 12.1 可以看到相关例子）。

```
#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;
  }
```

和往常一样，我们必须意识到一点：符号表操作的不同实现方案有不同的性能，这也许和操作的混合有关。一个应用也许相对地比较少使用插入操作（或者用于建立一张表），但却需要执行大量的搜索操作；而另一个应用也许极其频繁地对相对较小的表执行插入和

删除操作，并夹杂着一些搜索操作。并非所有实现都支持所有操作，某些实现也许会为部分功能提供高效的支持，而牺牲其他功能；这样做是基于一个隐含的假设：功能开销大，执行次数少。在符号表接口中每个基本操作都有很重要的应用，因而人们已经提出了许多基础的组织方案，以支持有效地利用这些操作的不同组合。在本章和接下来的几章里面，我们将集中讨论基本功能构建、插入和搜索的实现，适当时也对删除、选择、排序和联结作一些评论。要考虑的算法种类繁多，它们是由和基本操作的不同组合对应的不同性能，或者对关键字值的制约，又或者数据项大小等因素产生的。

在本章中我们将看到这些实现方法：对于随机关键字，搜索、插入、删除和选择所需时间平均和字典中数据项中数量的对数成正比，而排序则线性正比。我们在第十三章中分析的方法可保证得到这个水平的性能。而我们将看到的 12.2 节的一个实现和第十四、十五章的几个实现，可以在某些情况下使性能达到常数时间。

我们已研究过很多其他的关于符号表的操作。其中例子包括：手动搜索（finger search），其中一次搜索可以从前一次搜索结束的那一点开始；范围搜索（range search），其中我们希望计数或显示落入某一区间的所有节点；还有我们认为关键字之间存在距离（distance）的近邻搜索（near-neighbor search），其中我们希望找出其关键字最接近一个给定关键字的所有数据项。在第六部分，我们将在几何算法环境中考虑这样的操作。

## 练习

▷ 12.1 编写一个 Item 类实现（类似于程序 12.1），支持这个符号表实现处理仅由整数关键字组成的数据项。

12.2 编写一个 Item 类实现（类似于程序 12.1），支持这个符号表处理仅由 C 类型字符串关键字组成的数据项，像程序 6.11 那样为字符串建立一个缓冲区。

▷ 12.3 利用符号表 ADT 程序 12.2，实现堆栈和队列的 ADT。

▷ 12.4 利用由程序 12.2 接口定义的符号表 ADT，实现一个优先队列 ADT，支持“删除最大者”和“删除最小者”的操作。

12.5 利用由程序 12.2 接口定义的符号表 ADT，实现一个数组排序，使它兼容于第六章到第十章的相应排序方法。

▷ 12.6 向程序 12.2 加入对应以下部件的声明：一个析构函数、一个复制构造函数和一个重载赋值操作符，使它转变成一个一级 ADT（参阅 4.8 节和 9.5 节）。

12.7 为一个允许客户程序通过句柄删除特定数据项和改变关键字的符号表 ADT，定义一个接口（参阅 4.8 节和 9.5 节）。

▷ 12.8 为具有以下两个域的数据项定义数据项类型接口和实现方案：一个 16 比特整数关键字和一个含有和该关键字相关信息的 C 字符串。

● 12.9 对 $N$ 个小于 1000 的随机正整数，给出我们的驱动程序例子（程序 12.3）将在其中找出的不同关键字个数的平均值，$N=10$，$10^2$，$10^3$，$10^4$ 和 $10^5$。通过实验或理论分析，或两者的结合确定你的答案。

## 12.2 关键字索引搜索

假设关键字值都是小数字且两两不同。这种情况下，最简单的搜索算法是基于把数据项存储在一个用关键字作索引的数组中，就像程序 12.4 给出的实现方法那样。其代码很简单：操作符 new[ ]用 nullItem 初始化所有数组项，然后我们直接通过把关键字值为 k 的一个数据项存储在 st[k]实现插入操作，而通过检查 st[k]实现搜索关键字值为 k 的数据项。要删除关键字值为 k 的数据项的话，我们就把 st[k]置为 nullItem。程序 12.4 里的选择、排序和计数实现方法利用了对整个数组的线性扫描，扫描中跳过空的数据项。这种实现把以下任务留给客户程序解决：处理关键字重复的数据项和检查非法操作的情况，例如对一个不在表内的关键字指定删除操作。

**程序 12.4 基于关键字索引数组的符号表**

本代码假设关键字值为小于标记值 M 的正整数，并利用它们作为数组的索引。我们使用以下约定：Item 构造函数创建数据项，其关键字值等于标记值，使得 ST 构造函数能够找出一个空数据项中的 M 值。当标记值很大时，这种监视的主要开销是所需的空间；当 N 相对 M 较小时，是 ST 构造函数所需的时间。

```
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); }
};
```

对于我们在本章和第十三章到第十五章中考虑的符号表实现方案来说，这个实现是一个分支点。它可用于多个不同的客户程序和不同的数据项类型。编译器将校验并确定：接口、实现和客户程序都遵守同一个已定义的约定。

我们在第六章里分析过关键字索引计数排序方法，而它的基本操作和关键字索引搜索所基于的编制索引操作是一样的。当关键字索引搜索可用时，它是首选的方法，因为几乎不可能比它更高效地实现搜索和插入操作了。

如果根本没有任何数据项（只有关键字），我们可以用一张由位构成的表。这种情况下的符号表称为存在表（existence table），因为我们可把第 $k$ 个位看作表明 $k$ 是否存在于表的关键字集合中。例如，我们可以利用这个方法快速确定：电话交换机里一个给定的 4 位数字是否已被分配（在一台 32 位计算机上使用一张 313 字的表，见练习 12.14）。

**性质 12.1** *若关键字值为小于 $M$ 的正整数且数据项都具有独特的关键字，则符号表数据类型可由数据项的关键字索引数组实现，使得对一个 $N$ 数据项的表执行任何操作时，插入、搜索和删除操作所需时间为常数，初始化、选择和排序所需时间和 $M$ 成正比。*

审视一下代码即可马上得出以上结论。要注意的是，对关键字的约束条件默认为 $N \le M$。■

程序 12.4 不处理重复关键字，它假设关键字值在 0~M-1 之间。我们可以利用链表或者 12.1 节中提到的其他方法之一保存任何关键字重复的数据项，而在使用关键字作为索引之前我们可以对它们进行简单的改造（见练习 12.13）。但我们将推迟到第十四章才详细考虑这些情况，那时我们要讨论散列（也称为哈希函数，hashing）：它使用同一种方法对广义关键字实现符号表：改造来自于一个潜在的较大值域的关键字，使得它们落入一个比较小的值域，然后对关键字重复的数据项执行恰当的动作。我们暂时假设可以悄悄地忽略以下情况（像程序 12.4 那样）：一个旧数据项的关键字值等于一个即将插入数据项的关键字值；或者把它作为一种出错情况处理（见练习 12.10）。

程序 12.4 中的 count 实现是一种懒惰的方法，仅当函数 count 被调用时我们才执行动作。一种（积极的）备选方案是：在一个局域变量中保存表的非空位置的计数值，如果对一个包含 nullItem 的表位置进行插入操作则使变量增加，如果对一个不包含 nullItem 的表位置进行删除操作则使变量减少（见练习 12.11）。如果 count 操作极少使用（或根本不用）

且可能存在的关键字的数量较少，那么懒惰方法较佳；如果 count 操作经常使用且可能存在的关键字的数量很大，那么积极方法较佳。对一个通用函数库程序来说，积极方法更为可取，因为在最差情况下它可以提供最佳性能，付出的代价只是插入和删除操作少量数值恒定的性能损失；对一个大量使用插入和删除操作、却很少使用 count 操作的应用程序的内层循环来说，懒惰方法更为可取，因为它给出了一般操作的最快速实现方案。正如我们在几个场合中所见那样，在必须支持多种操作混合的 ADT 的设计过程中，这类进退两难的困境常常出现。

在一个通用接口的开发过程中，我们也需要做出很多其他设计上的抉择。譬如，应该使所有对象的关键字范围一致还是使不同对象具有不同的关键字范围呢？如果选择了后者，也许必须向构造函数添加参数，并使其具有让客户访问关键字范围的功能。

关键字索引数组对许多应用都很有用，但如果关键字不落入一个小范围内它们就不适用了。实际上，我们可以认为本章和接下来的几章所关注的是为以下情况设计解决方案：其关键字来自一个极大的值域范围，以至于使用一张为每个关键字保留一个位置的索引表是不可行的。

### 练习

12.10 利用动态分配的关键字索引数组实现一个一级符号表 ADT（见练习 12.6）。

▷ 12.11 修改程序 12.4 的实现，提供一个积极的 count 实现（通过记录非空数组项的数目）。

▷ 12.12 修改你对程序 12.10 的实现，提供一个积极的 count 实现（见练习 12.11）。

12.13 开发程序 12.4 的一个使用函数 h(Key)的版本，该函数把关键字转换为小于 M 的非负整数，且没有两个关键字映射到同一整数上。（这个改良使得在关键字处于一个小范围内——不一定从 0 开始——或其他简单情况下，该实现方案都相当有用。）

12.14 对于数据项是小于 M 的正整数的关键字（再没有相关的信息）的情况，开发程序 12.4 的版本。在实现中使用一个大小约为 M/bitsword 字的动态分配数组，其中 bitsword 是你的计算机系统中每个字的位数。

12.15 使用你对程序 12.14 的实现，对 $N$ 个小于 $N$ 的非负整数组成的一个随机序列，通过实验确定其中不同整数的数量平均值和标准偏差。$N$ 接近于你的计算机上一个程序可使用的、表示为位数的内存数量（见程序 12.3）。

## 12.3 顺序搜索

对来自于一个极大的值域因而不能被用作索引的关键字，一个简单的符号表实现方法是把这些数据项依次连续地存储在一个数组里面。要插入一个新数据项时，我们通过使较大的元素向后移动一个位置把它加入到数组中，就像我们对插入排序所做的那样；要执行一次搜索时，我们按顺序在数组中查看。因为数组有序，当我们遇到一个大于搜索关键字

的关键字时我们就可以报告一次搜索失败。而且因为数组有序，选择和排序都可以直接实现。程序 12.5 就是一个基于这种途径的符号表实现。

**程序 12.5 基于数组的符号表（有序）**

类似程序 12.4，本实现使用了一个数据项数组，但它并不要求关键字是小整数。我们用和插入排序同样的方式保持数组有序：插入一新数据项时，通过移动较大的元素腾出空间。那么,函数 search 即可对数组从头到尾扫描，搜索带有指定关键字的数据项，若遇到一个带有更大关键字的数据项就返回 nullItem。函数 select 和 sort 都很简单，而 remove 的实现则留作练习（见练习 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); }
  };
```

我们可以用以下方法对程序 12.5 中搜索实现的内层循环稍作改良：设立一个标记，免

除在表中数据项没有指定搜索关键字的情况下数组访问越界。更具体地说，我们可将数组结尾后面的位置保留作为标记，然后在一次搜索之前把搜索关键字填入其关键字域。那么搜索操作永远以一个含有搜索关键字的数据项结束，而我们则可以检查该数据项是否是标记，以此确定关键字是否在符号表中。

作为另一种选择，我们可以开发一个并不坚持要数组中的数据项保持有序的实现方法。要插入一个新数据项时，我们把它放到数组末尾；要执行一次搜索时，我们按顺序在数组中查看。这种途径的特有性质是：插入操作很快，但选择和排序操作需要花更多的工夫（它们各自需要第七章到第十章讨论的方法之一）。我们可以通过以下步骤删除带有某指定关键字的一个数据项：首先对它执行一次搜索，然后把数组中最后的数据项移动到它的位置上，并把数组大小减 1；重复这个操作，我们便可删除带有某指定关键字的全部数据项。如果有一个给出数组中数据项索引的处理句柄可供利用，我们就不需要进行搜索操作，而删除操作可在恒定时间内完成。

符号表实现的另一种直观方法是利用链表。前面提到的两种选择再一次摆在我们面前，我们可以选择使链表有序，以便轻松地支持排序操作；也可以使它停留在无序状态，以便快速完成插入操作。程序 12.6 是后者的一个实现。和往常一样，相对于使用数组，链表的优点在于，我们不必预先知道表占用的最大空间；而缺点则是我们需要额外的空间（用于链接），而且不能有效地支持选择操作。

**程序 12.6 基于链表的符号表（无序的）**

这里实现的 construct、count、search 和 insert 函数使用了一个链表，其中每个节点包含一个数据项，该数据项由一个关键字和一个链接。insert 函数在链表开头插入一个新数据项，所花费的时间是常数的。成员函数 search 调用了私有递归函数 searchR 来扫描链表。

```
#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++; }
};
```

无序数组和有序链表方法留作练习（见练习 12.20 和 12.21）。在应用中可以交替使用这四种实现方法（数组或链表，有序或无序），其区别仅仅在于（我们所期望的）时间和空间的要求。在本章和接下来的几章中，我们将分析许多对应符号表实现问题的不同方法。

保持数据项有序是对以下概括思想的一种描述：符号表实现通常以某种方式利用关键字来结构化数据，以此提供快速的搜索。这种结构也许能使其他一些操作快速地实现，但这种开销节省必须和维护该结构的开销达到平衡，而后者可能减慢其他操作的速度。我们将看到这种现象的许多例子。例如，在一个经常需要调用排序函数的应用中，我们会选择一种有序（数组或链表）表示方法，因为这种选定的符号表结构使排序函数很简单，反之则需要详尽的排序操作实现。在一个我们已知可能会经常执行选择操作的应用中，我们将选择一种有序数组表示方法，因为这种符号表结构使选择操作在常数时间内完成。相比之下，在一个链表、甚至是有序链表中，选择操作需要的时间呈线性增长。

要更详细地分析对随机关键字的顺序搜索的性能，我们先来考虑插入新关键字的开销以及成功搜索和不成功搜索各自的开销。我们常常把前者称为“搜索命中”（search hit），而把后者称为“搜索失败”（search miss）。我们关注的是在平均水平和最坏情况下，命中和失败两种结果的开销。严格来讲，我们的有序数组实现（见程序 12.5）对每个被分析的数据项使用了两次比较（分别是==和<）。为了便于分析，在第十二章到第十六章中我们把这样一对数据项比较看作一次单独的比较过程，因为我们通常可以进行低层次的优化以有效地把它们结合起来。

**性质 12.2** *对于在一个含有 $N$ 个数据项的符号表中进行的顺序搜索，搜索命中（平均）大约需要 $N/2$ 次比较。*

参见性质 2.1。参数 N 适用于有序或无序的数组或链表。■

**性质 12.3** *对于在一个含有 $N$ 个无序数据项的符号表中进行的顺序搜索，搜索失败需要一恒定次数的插入操作和 $N$ 次比较。*

这些事实对于数组和链表表示都成立，而且可以从实现方法直接得出（见练习 12.20 和程序 12.6）。■

**性质 12.4** *对于在一个含有 $N$ 个有序数据项的符号表中进行的顺序搜索，插入、搜索命中和搜索失败（平均）都需要 $N/2$ 次比较。*

参见性质 2.2。再次，这些事实对于数组和链表表示都成立，而且可以从实现方法直接得出（见练习 12.21 和程序 12.5）。■

通过连续插入建立一张有序表实质上等于运行 6.2 节的“插入-排序”算法。建表的总计运行时间是二次方的，所以我们不会把这种方法用于大型的表。如果我们在一个比较小的表里有非常多的搜索操作，那么保持数据项有序是值得的，因为从性质 12.3 和 12.4 我们可知这种策略能够使搜索失败的时间减少一个 2 的因子。若表中不保存关键字重复的数据项，维持表有序的额外开销并非表面看上去那样繁重，这是因为一次插入操作仅在一次搜索失败后发生，因此插入时间和搜索时间成正比。另一方面，如果表可以保存关键字重复的数据项，对一个无序表我们能够得到时间恒定的插入实现。如果应用中我们有极大数量的插入操作和相对较少的搜索操作，则推荐使用无序表。

除了这些分别，前面讨论的实现方法之间还有以下的标准性能权衡：链表实现把空间用于链接，而数组实现则要求预先知道表的最大使用空间，否则表的增长将受到阻延（见 14.5 节）。同样，正如 12.9 节讨论的那样，一个链表实现能够灵活地支持其他操作（例如联结和删除）的有效实现。

表 12.1 总结了这些结论，并把它们和稍后在本章和第十三、第十四章里讨论的其他搜索算法放到一起作对比。在 12.4 节我们讨论二叉搜索（binary search），它把搜索时间缩减到 lg N，因此在静态表（相对来说，插入操作较少）中得到了广泛应用。

在 12.5 节到 12.9 节中，我们将讨论二叉搜索树（binary search tree），它具有支持在正比于 lg N 的时间内完成搜索和插入的灵活性，但这仅仅是平均情况。我们在第十三章中将讨论红黑树（red-black tree）和随机二叉搜索树（randomized binary search tree），它们分别能保证使算法性能达到对数水平或非常近似的水平。在第十四章中将讨论散列（也称为哈希函数，hashing），在平均水平上它支持常数时间内的搜索和插入，但不能有效地支持排序和一些其他操作。在第十五章中将讨论基数搜索方法，它类似第十章的基数排序方法。在第十五章中我们将讨论适用于存储在外部设备上的文件的方法。

## 练习

▷ 12.16 向我们基于有序数组的符号表实现方案（程序 12.5）加入一个对删除操作的实现。

▷ 12.17 为我们基于链表（程序 12.6）和基于数组（程序 12.5）的符号表实现方案实现 searchinsert 函数。它们应该在符号表中搜索和一给定数据项具有相同关键字的数据项，如果没有则插入该数据项。

12.18 为我们基于链表的符号表实现方案（程序 12.6）实现一个选择操作。

12.19 对一个最初为空的表，给出把关键字 E A S Y Q U E S T I O N 输入到该表所需的比较次数。表使用的 ADT 分别用以下四种基本方法之一实现：有序或无序的数组或链表。假设对每个关键字执行一次

搜索，若搜索失败则进行一次插入，就像练习 12.17 那样。

12.20　为程序 12.2 的符号表接口实现构造（construct）、搜索和插入操作，用一个无序数组表示该符号表。你的程序应该符合表 12.1 提出的性能特点。

○ 12.21　为程序 12.2 的符号表接口实现构造、搜索、排序和插入操作，用一个有序链表表示该符号表。你的程序应该符合表 12.1 提出的性能特点。

○ 12.22　改变我们基于链表的符号表实现方案（程序 12.6），支持客户数据项处理（见练习 12.7）；添加对一个析构函数、一个复制构造函数和一个重载赋值操作符的实现（见练习 12.6）；添加删除和联结操作；编写一个驱动程序测试你的一流符号表 ADT 接口以及实现。

**表 12.1　符号表中插入和搜索的开销**

本表中的数据项是作为 $N$ 和 $M$ 的一个函数、在一个常数因子范围内的运行时间，其中 $N$ 代表符号表里面数据项的数量，$M$ 代表符号表的大小（如果它不等于 $N$）。在这里考虑的实现中，我们可以插入新数据项而无须理会符号表是否存在关键字重复的数据项。基本方法（开始 4 行）对某些操作需要常数时间，对其他则需要线性时间；更高级的方法对大多数甚至全部操作都能保证得到对数或常数时间。对应选择操作一列的 $N\lg N$ 项表示了排序数据项的开销——在理论上对数据项的一个无序集合进行线性时间内的选择是有可能的，但实际上很难实现（见 7.8 节）。标了星号的项表明几乎不可能出现最坏情况。

| | 最坏情况 | | | 平均情况 | | |
|---|---|---|---|---|---|---|
| | 插入 | 搜索 | 选择 | 插入 | 搜索命中 | 搜索失败 |
| 关键字索引数组 | 1 | 1 | $M$ | 1 | 1 | 1 |
| 有序数组 | $N$ | $N$ | 1 | $N/2$ | $N/2$ | $N/2$ |
| 有序链表 | $N$ | $N$ | $N$ | $N/2$ | $N/2$ | $N/2$ |
| 无序数组 | 1 | $N$ | $N\lg N$ | 1 | $N/2$ | $N$ |
| 无序链表 | 1 | $N$ | $N\lg N$ | 1 | $N/2$ | $N$ |
| 二叉搜索 | $N$ | $\lg N$ | 1 | $N/2$ | $\lg N$ | $\lg N$ |
| 二叉搜索树 | $N$ | $N$ | $N$ | $\lg N$ | $\lg N$ | $\lg N$ |
| 红黑树 | $\lg N$ | $\lg N$ | $\lg N$ | $\lg N$ | $\lg N$ | $\lg N$ |
| 随机树 | $N^*$ | $N^*$ | $N^*$ | $\lg N$ | $\lg N$ | $\lg N$ |
| 散列 | 1 | $N^*$ | $N\lg N$ | 1 | 1 | 1 |

12.23　编写一个性能驱动程序，它利用 insert 填充一个符号表，然后利用 select 和 remove 清空该表。对各种长度（从小到大）的关键字随机序列执行几遍这样的操作，测量每次运行的时间并打印或绘制平均运行时间。

12.24　编写一个性能驱动程序，它利用 insert 填充一个符号表，然后利用 search，使得表中每个数据项平均被命中 10 次，而失败次数也大约是这个数字。对各种长度（从小到大）的关键字随机序列执行几遍这样的操作，测量每次运行的时间并打印或绘制平均运行时间。

12.25　编写一个练习驱动程序，它对实际应用中可能出现的困难或病态的情况使用程序 12.2 中符号

表接口的函数。简单的例子包括：已经有序的文件、逆序的文件、所有关键字相同的文件和仅由两个不同值组成的文件。

○ 12.26 对一个进行 $10^2$ 次插入操作、$10^3$ 次搜索操作和 $10^4$ 次选择操作且这些操作随机混杂的应用，你会采用哪一种符号表实现方案？证明你的答案。

○ 12.27 对其他五种操作和使用频率的可能组合回答练习中的 12.26 问题。（本练习实际上是五个练习的集合。）

12.28 自组织（self-organizing）搜索算法是这样一个算法：它重新排列所有数据项，使得在搜索中那些经常被访问的数据项很可能及早被找到。修改你对练习 12.20 的搜索实现，对每次搜索命中执行以下操作：把找到的数据项移到链表开头，并把链表开头和空出位置之间的所有数据项向右移动一个位置。这个处理称为“前端迁移”（move-to-front）启发算法。

▷ 12.29 给出经过以下处理后关键字的顺序：把关键字为 E A S Y Q U E S T I O N 的数据项输入到一个最初为空的表，输入方法是用搜索和跟随搜索失败进行的插入操作，其中利用了“前端迁移”自组织启发式搜索（见练习 12.28）。

12.30 为自组织搜索方法编写一个驱动程序，它利用 insert 把 $N$ 个关键字填入一个符号表，然后做 10N 次搜索，根据预定义的可能分布情况命中数据项。

12.31 分别对 $N = 10, 100$ 和 $1000$，利用你对练习 12.30 的解决方案，比较你对练习 12.20 和练习 12.28 的实现的运行时间，其中使用以下可能分布：搜索第 $i$ 个最大关键字的可能性是 $1/2i$，$1 \le i \le N$。

12.32 对以下可能分布重做练习 12.31：搜索第 $i$ 个最大关键字的可能性是 $H_N/i$，$1 \le i \le N$。这个分布称为 Zipf 定律。

12.33 比较“前端迁移”启发算法和对应练习 12.31、12.32 中分布的最优排列。后者保持关键字处于递增顺序（它们的预期访问频率的递减顺序）。也就是说，在练习 12.31 中使用程序 12.5 而不是你对练习 12.20 的解决方案。

## 12.4 二叉搜索

在顺序搜索的数组实现中，我们可以利用一个搜索处理使对于大型数据项集合的搜索时间大大减少。这种处理的基础是应用标准分治策略（见 5.2 节）：把数据项集合分成两部分，确定搜索关键字属于哪一部分，然后集中处理那部分数据项。划分数据项集合的一种合理方法是保持数据项有序，然后利用指向有序数组的索引界定正在进行处理的那一部分。这种搜索技术称为二叉搜索（binary search）。程序 12.7 是该基本策略的一个递归实现。程序 2.2 是这种方法的一个非递归版本——它不需要堆栈，因为程序 12.7 的递归函数以递归调用为结果。

图 12.1 显示了搜索一个小型表时二叉搜索分析的子文件；图 12.2 则描述了一个更大的范例。每次迭代消除的部分略微多于表的一半，因此所需的迭代次数并不多。

**性质 12.5** *对于一次搜索（命中或失败），二叉搜索使用的比较次数永远不超过* $\lfloor \lg N \rfloor + 1$。

参阅性质 2.3。有个很有趣的性质值得注意：在一个大小为 $N$ 的表中，一次二叉搜索使用的比较次数恰恰就是 $N$ 的二进制表示的比特位数，这是因为向右移 1 位的操作把 N 的二进制表示转换为 $\lfloor N/2 \rfloor$ 的二进制表示（参阅图 2.6）。■

**程序 12.7　二叉搜索（对应基于数组的符号表）**

这个 search 实现使用了一个递归二叉搜索过程。要找出一个给定关键字 v 是否存在于一有序数组中，该实现先比较 v 和中央位置上的元素。如果 v 较小，那么它一定处于数组的前半部分；如果 v 较大，那么它一定处于数组的后半部分。

数组必须有序。这个函数可以取代程序 12.5 中的 search，后者在插入期间动态地维持这个顺序。我们也可以引入一个符号表构造函数，它接收一个数组作为参数，然后用参数数组中的数据项建立一个符号表，并利用一标准排序例程对它进行处理，为搜索作好准备。

```
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); }
```

像我们在插入排序所做的那样保持符号表有序——这样运行时间将是插入操作次数的二次函数。但如果搜索操作次数极大，这个开销是可以接受的，甚至是可忽略不计的。在典型情况下，搜索开始之前所有数据项（即使其中一大部分）都可利用，则我们可以通过以下的构造函数构造符号表：它接收一个数组作为参数，并利用第六章到第十章中讨论的标准排序方法之一，在初始化期间对符号表排序。这样做以后，我们有许多方法进行更新。例如，我们可以像程序 12.5 那样在插入时保持有序（也可参阅练习 12.21）；或者把它们分批，再排序并归并（正如练习 8.1 讨论的那样）。任何更新都可能牵涉到这样一个数据项：它的关键字比表中任何数据项的关键字都要小，因此为了腾出空间必须移动表中的每一个数据项。这种由更新操作导致高开销的可能性就是使用二叉搜索的最大弱点。另一方面，有非常多这样的应用：其中可以预先排序一个静态表，而由程序 12.7 这样的实现提供的快速访问使得二叉搜索成为我们的最佳选择。

**图 12.1　二叉搜索**

在范例文件中，二叉搜索仅仅使用了三次迭代就找出了搜索关键字 L。第一次调用时，算法把 L 和文件中间的关键字 G 作比较。因为 L 较大，下一次迭代就在文件的右半部分进行。然后，因为 L 小于 M——右半部分中间的关键字，第三次迭代涉及的子文件包括 H、I 和 L，大小为 3。再进行一次迭代后，子文件大小为 1，本算法找到了关键字 L。

如果我们必须动态地插入新数据项，似乎我们需要一个链表；但一个单向链表不能产生

高效的实现方法，因为二叉搜索的效率有赖于我们能够通过索引迅速访问任何子数组的中间位置，而访问单向链表的中间位置的惟一途径是跟随链接向前遍历。我们需要更复杂的数据结构来把二叉搜索的高效和链接结构的灵活结合起来，我们不久就开始分析这个问题。

若表中存在重复关键字，则我们可以扩展二叉搜索以支持以下符号表操作：计算带有给定关键字的数据项数量或把它们作为一个组返回。如果符号表里多个数据项都带有和搜索关键字相等的关键字，它们就构成了表中一个连续块（因为表有序），而程序 12.7 中一次成功的搜索将在这一块中某处结束。如果一个应用需要访问所有这些数据项，我们可以加入从搜索终止的一点向两个方向进行扫描的代码，并返回两个索引，指出其关键字和搜索关键字相等的数据项。这种情况下，搜索的运行时间正比于 $\lg N$ 以及找到的数据项数量的和。对于更为一般的范围搜索问题：寻找其关键字落入某指定值域的所有数据项，也可用一种类似方法解决。在第六部分我们将考虑这些对符号表操作的基本集合的扩展。

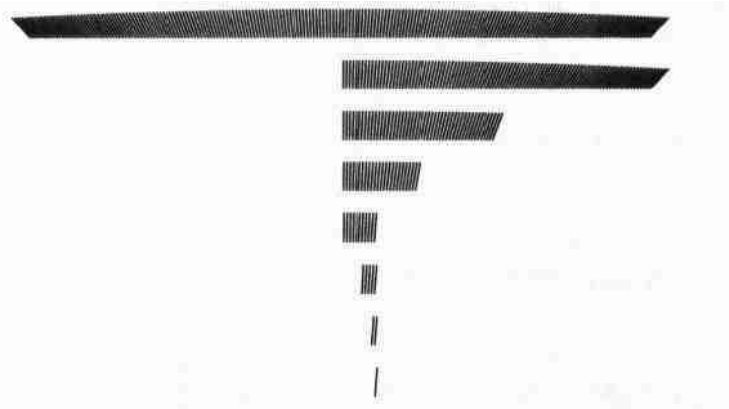


**图 12.2 二叉搜索**

通过二叉搜索，我们只需要 7 次迭代即可在含有 200 个元素的文件中定位一条记录。子文件大小遵循以下顺序：200，99，49，24，11，5，2，1，每一个都略少于前一个的一半。

二叉搜索算法进行的比较操作的顺序是预先确定的：使用的特定序列依赖于搜索关键字的值和 $N$ 值。可用一个二叉树结构（诸如图 12.3 所展示的一个）描述该比较结构。这棵树类似于我们在第八章里用于描述归并排序的子文件大小的那一棵（图 8.3）。对于二叉搜索，我们走其中一条穿过树的路径；对于归并排序，我们遍历所有穿过树的路径。这棵树既是静态的也是隐含的；在 12.5 节中，我们将看到的算法使用一种动态的、显式构造的二叉树结构来指引搜索过程。

二叉搜索一个可能的改善方法是：更精确地推测搜索关键字落入当前所关注区间的位置（而不是盲目地在每一步都把它和中间元素作对比测试）。这种策略模拟我们在电话目录中寻找一个名字或在字典中寻找一个单词时采用的方法：若我们正在寻找的词开头的字母接近字母表的开始，我们就在那本书的开始搜索；但如果它开头的字母接近字母表的末尾，我们就在那本书的末尾搜索。为实现这个称为插值搜索（interpolation search）的方法，我

们对程序 12.7 作如下修改：把语句

```
m = (l+r)/2;
```

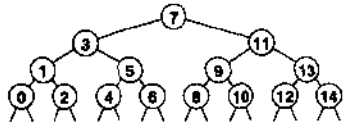
替换为

```
m = l+(v-a[l].key())*(r-l)/(a[r].key()-a[l].key());
```

为了证明这个修改的正确性，特作以下说明：$(l+r)/2$ 是 $l+1/2\ (r-l)$ 的简写——我们把区间大小的一半和左端点相加，算出区间的中间位置。使用插值搜索实际等于把方程里的 1/2 替换成对关键字可能处于的位置的估计——明确来说就是 $(v-k_l)/(k_r-k_l)$，其中 $k_l$ 和 $k_r$ 分别表示 a[l].key( ) 和 a[r].key( )的值。这个计算基于以下假设：关键字值是数值型且均匀分布。

对随机关键字构成的文件，以下情况是可能的：对一次搜索（命中或失败），插值搜索使用的比较次数小于 $\lg\lg N+1$。其证明远远超出了本书的范围。这个函数增长得很慢，在实用中可以看作为一个常数：如果 $N$ 等于 10 亿，$\lg\lg N<5$。因此我们只用几次访问即可找到任何数据项（平均来说），对于二叉搜索这是一个实质性的改善。若关键字分布得比随机情况更规则，插值搜索的性能甚至会更好。实际上，极限情况是 12.2 节提到的关键字索引搜索方法。

然而，插值搜索的确非常依赖于“关键字在区间上均匀分布”这个假设——分布极不均匀的关键字可以严重地影响它，而在实践中这种情况的确经常发生。而且该算法还要求额外的运算。对较小的 $N$，直接二叉搜索的开销 $\lg N$ 足够地接近 $\lg\lg N$，使得插值搜索的开销不大可能达到值得关注的水平。另一方面，对于大型文件、比较操作开销特别大的应用、涉及到高昂的访问开销的外部方法，我们的确应该考虑使用插值搜索。



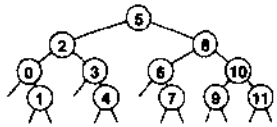




**图 12.3　二叉搜索的比较次序**

这些分治策略的树形图描述了二叉搜索中比较操作的索引次序。该模式依赖于初始文件大小，而不是文件里关键字的值。它们和对应归并排序和相近算法的树（图 5.6 和图 8.3）略有不同，因为根所在的元素不被包括到子树中。

上图说明了如何搜索一个由 15 个元素组成、索引值从 0 到 14 的文件。我们先考虑中间的元素（索引 7），然后（递归地）做以下步骤：若所求元素较小，考虑左子树；若所求元素较大，考虑右子树。每次搜索对应这棵树里面一条从上到下的路径：例如，对位于元素 10 和 11 之间的元素的一次搜索涉及的序列是 7，11，9，10。对于大小不等于 2 的次幂减 1 的文件，这个模式就不太规则，正如下图（对应 12 个元素）表明的那样。

## 练习

▷ 12.34　实现一个非递归的二叉搜索函数（参阅程序 12.7）。

12.35　对 $N=17$ 和 $N=24$，画出对应图 12.3 的树。

○ 12.36　找出符合下面条件的 N 值：在一个大小为 N 的符号表中，二叉搜索分别比顺序搜索快 10，100 和 1000 倍。根据分析预测这些值，并通过实验验证。

12.37　假设对一个大小为 N 的动态符号表的插入操作以插入排序形式实现，但我们用二叉搜索进行搜索。假定搜索操作的使用频率比插入操作高 1000 倍。对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，估算总计时间中投

入到插入操作的百分比。

12.38 利用二叉搜索和懒惰插入开发一个符号表实现方案，支持构造、计数、搜索、插入和排序操作。使用以下策略：为主符号表维持一个大型有序数组，为最近插入数据项维持一个无序数组；当 search 被调用时，对最近插入数据项（如果有的话）排序，把它们归并到主表中，然后使用二叉搜索。

12.39 向你对练习 12.38 的实现加入懒惰删除操作。

12.40 为你对练习 12.38 的实现回答练习 12.37。

○ 12.41 实现一个类似于二叉搜索（程序 12.7）的函数，返回符号表中其关键字等于给定关键字的数据项数量。

12.42 编写一个程序，为一给出的 $N$ 值生成 $N$ 条宏指令的序列，指令的索引从 0~$N$-1，形式为 compare(l, h)，其中列表的第 $i$ 条指令代表“比较搜索关键字和表中索引 $i$ 项的数值，若相等则报告搜索命中，若小于则继续执行第 l 条指令，若大于则继续执行第 $h$ 条指令”（保留索引 0 表示搜索失败）。该序列应该有以下性质：任何操作对于同样的数据都执行和二叉搜索同样的比较操作。

12.43 为练习 12.42 的宏开发一个扩展，使得你的程序产生以下机器代码：在一个大小为 $N$ 的表中进行二叉搜索，对每次比较使用尽可能少的机器指令。

● 12.44 假设对于 1~N 之间的 i，a[i] ==10*i。在一次对 $2k$-1 的失败搜索中，插值搜索需要检查多少个表位置？

● 12.45 确定 $N$ 值，使得在一个大小为 $N$ 的符号表中，插值搜索比二叉搜索快 1、2 和 10 倍，这里假设关键字是随机的。根据分析预测这些值，并通过实验验证。

## 12.5 二叉搜索树

为了解决插入开销过高的问题，我们将使用一个显式的树结构作为符号表实现的基础。这种基本的数据结构为搜索、插入、选择和排序等符号表操作开发在平均情况下具有最快性能的算法。对于很多应用它是最佳选择，并且成为了计算机科学中其中一种最基本算法。

我们在第五章比较详细地讨论了树，但回顾一下这个术语是有益的。一棵树（tree）的定义性质是：每个节点仅被另一个称为双亲的节点指向。一棵二叉树（binary tree）的定义性质是：每个节点具有左、右链接。链接可以指向其他二叉树或外部节点（external node），后者没有链接。有两个链接的节点也称为内部节点（internal node）。对于搜索，每个内部节点还有一个带有关键字值的数据项，我们把指向外部节点的链接称为空链接（null link）。内部节点的关键字值和搜索关键字作比较，并控制搜索的过程。

**定义 12.2 二叉搜索树（BST）是一棵二叉树，其中它的每个内部节点都有一个相关的关键字，并有以下附加性质：任意节点中的关键字大于（或等于）该节点左子树中所有节点含有的关键字，并小于（或等于）该节点右子树中所有节点含有的关键字。**

程序 12.8 利用 BST 实现符号表的搜索、插入、构造和计数操作。实现的第一部分把 BST 节点定义为：每个节点包含一个数据项（和一个关键字）、一个左链接和一个右链接。这些

代码还维护着一个容纳有树中节点数量的域，用于支持计数操作的一种积极的实现。左链接指向一棵 BST 以指明具有较小（或相等）关键字的数据项，右链接指向一棵 BST 以指明具有较大（或相等）关键字的数据项。

**程序 12.8　基于 BST 的符号表**

这个实现中的 search 和 insert 函数使用了私有递归函数 searchR 和 insertR，这两个私有递归函数直接反映出 BST 的递归定义。请注意 insertR 中对一个引用参数的使用。链接 head 指向树的根。

```
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); }
  };
```

给出这个结构以后，马上可得出在一棵 BST 中搜索关键字的一个递归算法：如果树为空，我们得到一次搜索失败；如果搜索关键字等于树根的关键字，我们得到一次搜索命中。否则我们在恰当的子树中（递归地）搜索。程序 12.8 中的 searchR 函数直接实现了这个算法。我们调用一个递归程序，它接收一棵树作为首个参数、一个关键字作为第二个参数，以该树的根和该搜索关键字开始。在每一步中我们可得到保证：除了当前子树外，这棵树没有其他部分能包含带有该搜索关键字的数据项。正如二叉搜索中，区间大小在每次迭代时都缩小（略多于）一半那样，在二叉搜索树中当前子树都比上一棵子树小（理论上来说，大约小一半）。这个过程要不在找到一个带有该搜索关键字的数据项（搜索命中）时结束，要不在当前子树变空（搜索失败）时结束。

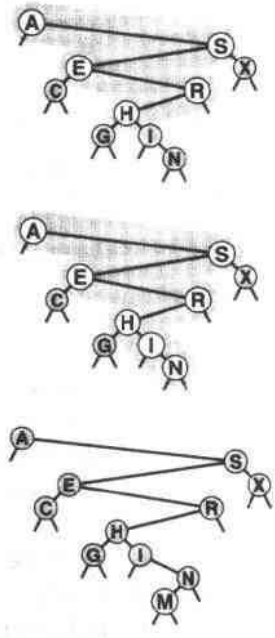



**图 12.4 BST 的搜索和插入**

在这棵样本树对 H 的一次成功搜索（上图）中，我们在树根处向右搜索（因为 H 大于 A），然后在根的右子树向左出发（因为 H 小于 S）……如此类推，不断地在树里往下寻找直到遇到 H 为止。在这棵样本树对 M 的一次失败搜索（中图）中，我们在树根处向右搜索（因为 M 大于 A），然后在根的右子树向左出发（因为 M 小于 S）……如此类推，不断地在树里往下寻找直到我们在树的底部、N 的左链接遇到一个外部链接为止。要在搜索失败后插入 M 的话，我们直接用一个指向 M 的链接代替使刚才搜索终止的链接（下图）。

图 12.4 中顶部的图例显示了对一棵作为范例的树的搜索过程。搜索过程从顶端出发，在每一个节点都进行对该节点的孩子之一的一次递归调用，所以搜索就定义出一条贯穿这棵树的路径。在搜索命中的情况下，路径在包含该关键字的外部节点终止。在搜索失败的情况下，路径在一个外部节点终止，正如图 12.4 中间的图例所显示的那样。

程序 12.8 用链接 0 表示外部节点，还有一个私有数据成员 head 作为指向树根的一个链接。我们把 head 设为 0，即可构造一棵空 BST。在类似于我们在表 3.1 对链表讨论的多种组合中，我们还可以在树根使用一个哑元节点，并使用另一个哑元节点表示所有外部节点（见练习 12.53）。

程序 12.8 的搜索函数就和二叉搜索一样简单；BST 的一个核心特色是插入的实现和搜索同样简单。一个递归函数 insertR 把一个新数据项插入到一棵 BST 中，它遵循的逻辑类似于我们用于开发 searchR 的逻辑，并使用了一个引用参数 h 建立该树：如果树为空，我们把 h 设置成为指向一个包含该数据项的新节点的链接；如果搜索关键字小于根的关键字，我们把它插入到左子树；否则就把它插入到右子树。也就是说，只有在最后的递归调用中插入新数据项时，引用参数才被改变。在 12.8 节和第十三章里我们将研究更高级的树结构，它们也用这个同样的递归方案很自然地表示，但更频繁地改变引用参数。

图 12.5 和图 12.6 展示了我们怎样通过向一棵原先为空的树插入一系列关键字，从而构造出一棵样本 BST。新节点挂接到树底部的空链接上，树的结构不会因此改变。因为每个节

点有两个链接，树倾向于向外、而不是向下生长。

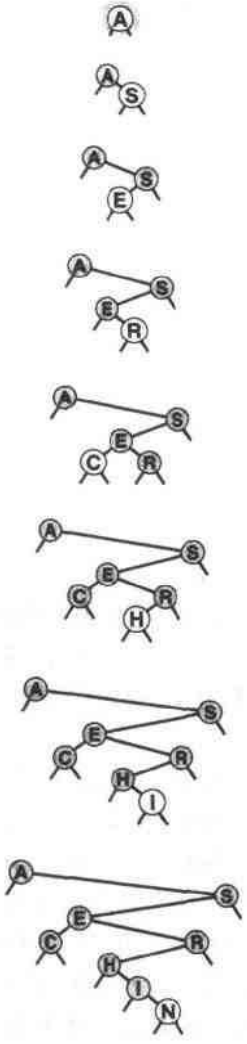



**图 12.5　BST 的构造过程**

这个系列的图描述了向一棵原先为空的树插入关键字 A S E R C H I N 的结果。每次插入都跟随在树底部的一次搜索失败之后。

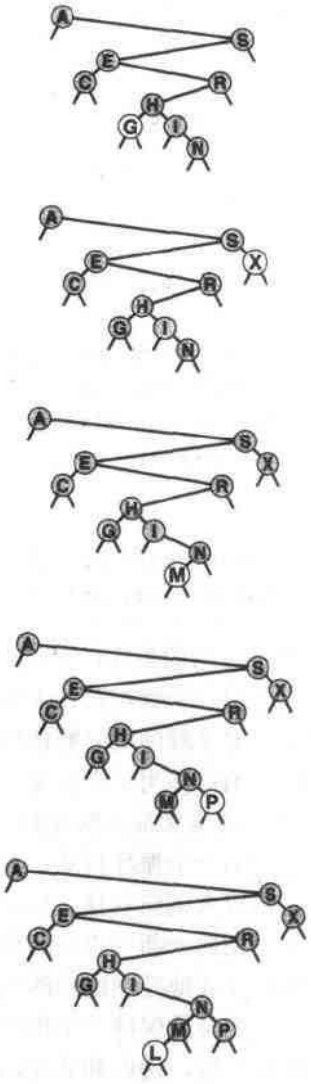



**图 12.6　BST 的构造过程（接图 12.5）**

这个系列的图描述了向图 12.5 中开始的 BST 插入关键字 G X M P L 的结果。

使用 BST 时，不需要太多工夫即可得到符号表的排序函数。构造一棵二叉搜索树实质等于排序数据项，因为当我们以正确的方式观察它时，一棵二叉搜索树代表了一个经过排序的文件。在我们的图中，如果在页面上从左向右读（忽略它们的高度和链接），关键字就按顺序出现了。一个程序仅有用于处理的链接，但依据定义，一个简单的中序遍历即可完成

这项任务：这正如程序 12.9 的递归实现 showR 所显示的那样。为了依照其关键字的顺序显示一棵 BST 的全部数据项，我们先（递归地）依照关键字的顺序显示左子树中的数据项，再显示树根，再（递归地）依照关键字的顺序显示右子树中的数据项。

**程序 12.9 对 BST 排序**

对一棵 BST 的一次中序遍历依照其关键字的顺序访问数据项。在这个实现中，我们利用 item 的成员函数 show 依照关键字顺序打印出数据项。

```
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); }
```

正如 12.1 节讨论的那样，当我们希望以一种系统的方式访问符号表中的每个数据项时，我们将不时地引用一个针对符号表的一般访问（visit）操作。对于 BST，我们可以通过以下方法依照其关键字顺序访问数据项：在刚才给出的描述中用“访问”取代“显示”，或者把访问数据项的函数作为一个参变量传递（参阅 5.6 节）。

构思 BST 的搜索和插入实现时，采用非递归的思路也是很有益处的。在一个非递归实现中，搜索过程由一个循环构成：我们对比搜索关键字和树根的关键字，若搜索关键字较小则向左移，若较大则向右移。插入的组成部分有：一次搜索失败（在一个空链接结束）和随之进行的用指向一新节点的指针替代该空链接的操作。这个处理过程对应了沿着树中一条向下的路径显式地控制链接的过程（参阅图 12.4）。特别是，为了能够在树的底部插入一个新节点，我们必须保持一个指向当前节点的双亲的链接，正如程序 12.10 的实现所做的那样。和往常一样，递归和非递归版本本质上是等价的，但对这两种观点的了解可以提高我们对算法以及数据结构的认识。

程序 12.8 的 BST 函数没有明确地检验关键字重复的数据项。当插入一个新节点而且它的关键字等于树中的某个已经存在的关键字时，它将落入树中现存节点的右边。这种约定做法的一个副作用是关键字重复的节点在树里显得不连续（参阅图 12.7）。然而，我们可以找到它们，方法是：从 search 找到第一个匹配的那点继续搜索，直到遇到一个 0 链接为止。正如 9.1 节中提到的，有几种其他方法可处理关键字重复的数据项。

BST 和快速排序是对称的。树根的节点对应快速排序中的分割元素（它左边的关键字都不比它大，右边的关键字都不比它小）。在 12.6 节中，我们将看到这个观察结论怎样和

对树的属性的分析发生关联。

**程序 12.10　BST 中的插入（非递归）**

向一棵 BST 插入一数据项等价于先对它进行一次失败搜索，然后在搜索终止的地方为该数据项拼接一个新节点，代替原来的空连接。拼接这个新节点要求我们在树中向下处理的过程中，记录当前节点 q 的双亲节点 p。当我们到达树的底部，对 p 指向的节点，我们必须把它的链接改为指向被插入的新节点。

```
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;
    if (v < p->item.key()) p->l = new node(x);
                      else p->r = new node(x);
  }
```

## 练习

▷ 12.46　当你向一棵原先为空的树按以下顺序插入关键字 E A S Y Q U T I O N 时，画出所得的 BST。

▷ 12.47　当你向一棵原先为空的树按以下顺序插入关键字 E A S Y Q U E S T I O N 时，画出所得的 BST。

▷ 12.48　向一个原先为空、使用一 BST 的符号表放入关键字 E A S Y Q U E S T I O N，给出所需的比较次数。假设对每个关键字执行一次搜索，每次搜索失败后执行一次插入，就像程序 12.3 那样。

○ 12.49　向一棵原先为空的树依照 A S E R H I N G C 的顺序插入这些关键字，也可得出图 12.6 顶部的树。给出这些关键字的 10 种其他排列顺序，使其产生同样的结果。

12.50　为二叉搜索树（程序 12.8）实现一个 searchinsert 函数。它应该在符号表中搜索一个和给定数据项具有相同关键字的数据项，如果一个也没有就插入该数据项。

▷ 12.51　编写一个函数，返回一个 BST 中带有和给定关键字相等的关键字的数据项个数。

12.52　假设我们预先对一棵二叉树中关键字被访问的频率进行了估算，那么把这些关键字插入到树时，应该以可能的访问频率的递增顺序还是递减顺序插入？解释你的答案。

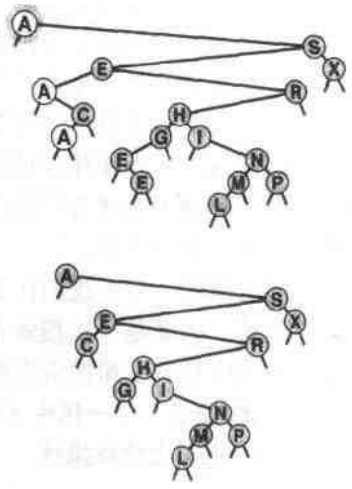


**图 12.7　BST 中的重复关键字**

当一棵 BST 有关键字重复的记录时（上图），它们似乎分散在树的各处，从三个突出显示的 A 可看到这点。重复关键字的确出现在该关键字的搜索路径上——从根到一个外部节点。所以可以很容易地访问它们。然而，为了避免像“C 下面的 A”这样易于混淆的用法，我们在范例中使用各不相同的关键字（下图）。

○ 12.53 对程序 12.8 的 BST 实现中的搜索和插入代码，用以下两个哑元节点进行简化：一节点 head 包含一个数据项，它具有一个小于所有其他关键字的标记关键字，且右链接指向树根；一节点 z 包含一个数据项，它具有一个大于所有其他关键字的标记关键字，且左、右链接都指向自己，它代表所有外部节点（外部节点都链接到 z）。（参阅表 3.1。）

12.54 修改程序 12.8 的 BST 实现，使链表中关键字重复的数据项不会挂接成为树的节点。更改这个接口使搜索操作类似排序操作（对于所有含有搜索关键字的数据项）。

12.55 程序 12.10 的非递归插入程序使用了过多的比较次数来确定 p 的哪个链接应被新节点代替。给出一个实现，使用链接指针来避免这个比较操作。

## 12.6 BST 的性能特性

二叉搜索树上算法的运行时间由树的形状决定。在最好的情况下，树达到完美的平衡状态——在根和每个外部节点之间有大约 $\lg N$ 个节点；但在最坏的情况下，在搜索路径上可能有 $N$ 个节点。

我们也许希望搜索时间也能在平均情况下达到对数水平，因为插入的第一个元素成为这棵树的根：若准备随机插入 $N$ 个关键字，那么该元素将把关键字分为两半（平均情况下），这样就可以产生对数水平的搜索时间（在子树中采用同样的论点）。实际上，一棵 BST 引起的比较次数正好和二叉搜索一样——这种情况有可能发生（参阅练习 12.58）。对于这个算法这将是最好的结果，对所有搜索都可以保证对数水平的运行时间。在一个真正随机的环境中，根等于任意一个关键字的可能性都相等，因此这样一棵完美的平衡树是极为罕见的，而在每次插入操作后使树保持完全平衡状态并不容易。然而，对于随机关键字来说高度不平衡的树也是极为罕见的，因此平均情况下树的平衡状态相当好。在这一节里，我们将量化这个观察结果。

更明确地，对于我们在 5.5 节讨论的二叉树，其路径长度和高度直接关系到 BST 中搜索的开销。高度是一次搜索在最坏情况下的开销，内部路径长度和搜索命中的开销直接相关，外部路径长度和搜索失败的开销直接相关。

**性质 12.6** *在一棵由 $N$ 个随机关键字建立的 BST 中，平均情况下搜索命中需要大约 $2\ln N \approx 1.39\lg N$ 次比较操作。*

正如在 12.3 节讨论的那样，我们认为连续的==和<是单独一次比较。一次搜索命中在一个给定节点结束，它使用的比较次数等于该节点到树根的距离加 1。把所有节点的这些距离相加，我们就得到树的内部路径长度。因此，理想的数量是 1 加上 BST 的平均内部路径长度，我们可以用一个熟悉的论证来分析它：如果 $C_N$ 表示一棵含有 $N$ 个节点的二叉搜索树的平均内部路径长度，我们有下面的递推公式：

$$C_N = N - 1 + \frac{1}{N}\sum_{1\le k\le N}(C_{k-1} + C_{N-k})$$

其中 $C_1 = 1$。其中一项 $N$-1 考虑的是：对于树里的其他 $N$-1 个节点，它们每一个的路

径长度中都有 1 个来自于根节点。表达式的其余部分来源于以下分析：根（第一个被插入）的关键字作为第 $k$ 个最大关键字的可能性都相等，此时得出的随机子树大小分别是 $k$-1 和 $N$-$k$。这个递推公式几乎等于我们在第七章为快速排序解答的那一个公式，我们可以用同样的方式解答它以得出以下陈述的结论。■

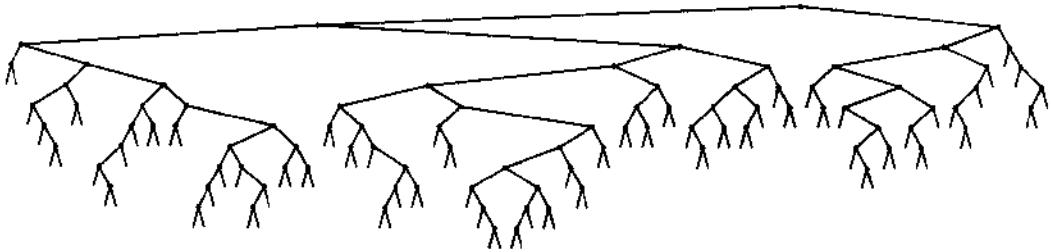


**图 12.8　一棵二叉搜索树的范例**

这棵 BST 通过向一棵原先为空的树插入大约 200 个随机关键字建立起来，在其中进行的搜索使用的比较都不超过 12 次。一次搜索命中的平均开销约为 10。

**性质 12.7**　在一棵由 $N$ 个随机关键字建立的 BST 中，平均情况下插入和搜索失败需要大约 $2\ln N \approx 1.39\lg N$ 次比较操作。

在一棵有 $N$ 个节点的树里搜索一个随机关键字，在失败的情况下，该搜索结束于 $N$+1 个外部节点中任意一个的可能性都相等。这个性质结合以下事实：任何树中外部路径长度和内部路径长度的差仅仅为 $2N$（参阅性质 5.7），即可确定该陈述的结论。任意 BST 中，一次插入或搜索失败的平均比较次数大约比一次搜索命中的平均比较次数大 1。■

性质 12.6 表示：对于随机关键字，我们应该预期 BST 的搜索开销比二叉搜索开销高大约 39%；但性质 12.7 表示这个额外开销非常值得，因为可用大约同样的开销插入一个新关键字——这种灵活性是二叉搜索不能提供的。图 12.8 显示了一棵由长随机排列建立的 BST。尽管它具有一些短路径和一些长路径，我们可认为它有良好的平衡性：任何搜索操作所需的比较次数少于 12，而一次随机搜索命中的平均比较次数是 7.00，相比之下二叉搜索是 5.74。

性质 12.6 和 12.7 是基于平均情况下的性能所得的结果，该性能依赖于随机排序的关键字。如果关键字并非随机排序，这个算法的执行效果可能很差。

**性质 12.8**　在最坏情况下，在一棵含有 $N$ 个关键字的二叉搜索树中进行的一次搜索需要 $N$ 次比较。

图 12.9 和图 12.10 描述了最坏情况下的 BST 的两个例子。对于这些树，二叉树搜索并不比使用单向链表的顺序搜索优胜。■

因此，对符号表的基本 BST 实现方案的优良性能有赖于以下条件：其关键字足够地类似于随机关键字，使这棵树不会含有许多长路径。此外，这种最坏情况的状态在实践中并非不可能——当我们使用标准算法向一棵原先为空的树顺序或逆序地插入关键字时，它就会出现。而这一系列操作是我们当然会尝试采用的——如果没有明确地警告我们避免这种做法的话。在第十三章中，我们将分析使得这种最坏情况几乎不可能出现的技术，以及完全消

除它的技术，从而令所有树都显得更类似于最佳情况下的树——保证所有路径长度达到对数水平。

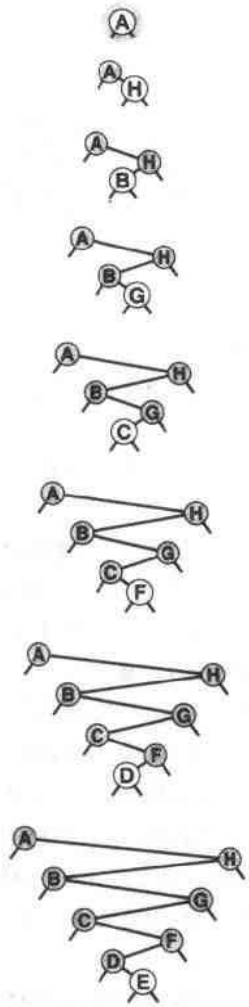



图 12.9 一棵最坏情况下的 BST

如果关键字以递增顺序进入 BST，它会退化成一种等价于单向链表的形式，导致二次的构造树时间和线性的搜索时间。

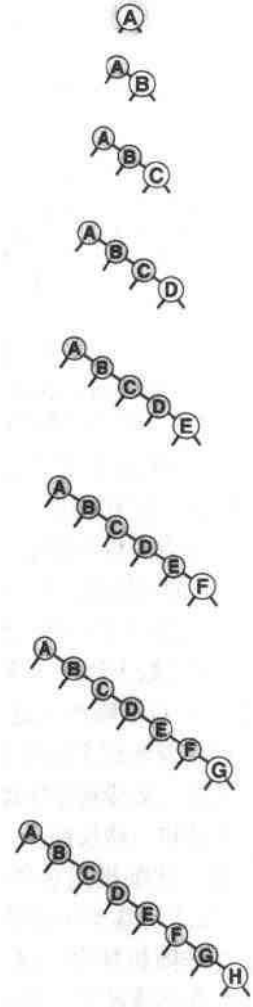



图 12.10 另一棵最坏情况下的 BST

很多其他的关键字插入顺序引致退化的 BST，例如这里的一种。尽管如此，一棵由随机排序关键字建造的树还是很可能达到良好的平衡状态。

在我们业已讨论的其他符号表实现方法中，没有任何一种适用于以下任务：把数目巨大的随机关键字插入到一张表中，然后搜索其中的每一个——对这个任务，我们从 12.2 节到 12.4 节讨论的每种方法的运行时间都达到二次水平。而且，分析结果告诉我们：在一棵

二叉树中到达一个节点的平均距离和该树的节点数量的对数成正比。这给予了我们有效处理混合的搜索、插入和其他符号表 ADT 操作的灵活性，我们不久就会看到这一点。

**练习**

▷ 12.56　编写一个递归程序，计算一给定 BST 中任意搜索所需的最大比较次数（树的高度）。

▷ 12.57　编写一个递归程序，计算一给定 BST 中一次搜索命中所需的平均比较次数（树的内部路径长度除以 N）。

12.58　给出向一棵原先为空的 BST 插入关键字 E A S Y Q U E S T I O N 的一种插入序列，使得在以下意义上，产生的树等价于二叉搜索：对于同一关键字集合，在 BST 中对任意关键字所作的搜索和二叉搜索使用相同的比较操作序列。

○ 12.59　编写一个程序，把一关键字集合插入到一棵原先为空的 BST，使得在练习 12.58 描述的意义上，产生的树等价于二叉搜索。

12.60　设 $2 \le N \le 5$，把 $N$ 个关键字插入到一棵原先为空的树，画出所有可能产生的结构不同的 BST。

● 12.61　确定以下情况的概率：练习 12.60 中每棵树都是把 $N$ 个不同的随机元素插入到一棵原先为空的树所得的结果。

● 12.62　存在多少棵 $N$ 个节点构成且高度为 $N$ 的二叉树？存在多少种不同方法，把 $N$ 个不同关键字插入到一棵原先为空的树，得出高度为 $N$ 的 BST？

○ 12.63　通过归纳法证明：任何二叉树中，外部路径长度和内部路径长度之间的差是 $2N$（参阅性质 5.7）。

12.64　对于 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$，把 $N$ 个随机关键字插入到一棵原先为空的树，建立一棵二叉搜索树。通过实验研究，分别计算其中的搜索命中和搜索失败所使用的比较次数的平均值和标准差。

12.65　对于 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$ 以及 $t = 10$，100 和 1000，编写一个程序，通过把 $N$ 个随机关键字插入到一棵原先为空的树建立 $t$ 棵 BST，并计算最大树高度（对于在 $t$ 棵树中任意一棵进行的任意一次搜索失败，所涉及的最大比较次数）。

## 12.7　符号表的索引实现

对于许多应用，我们希望一种有数据结构仅仅帮助我们找到数据项，而无须把它们四处移动。例如，我们也许有一个带有关键字的数据项组成的数组，而我们可能希望搜索方法给出指向数组中和某一个关键字吻合的数据项的索引。又或者，我们可能希望从搜索结构中删除带有某个索引的数据项，但仍然把它保留在数组里以作其他用途。在 9.6 节中，我们考虑了在优先队列中处理索引数据项、间接地访问一个客户数组中的数据的优点。对于符号表，这同一概念引致熟悉的索引（index）：一个处于数据项集合外部的搜索结构，它支持对给定关键字的快速访问。在第十六章中，我们将考虑数据项、也许甚至包括索引都处于外部存储器的情况；这一节中我们简要地考虑数据项和索引都可以加载到内存的情况。

我们可以改编二叉搜索树，用我们为 6.8 节的排序、9.6 节的堆提供间接实现的恰恰一样的方式建立索引：用一个 Index 封装为 BST 定义数据项，并安排通过 key 成员函数从数

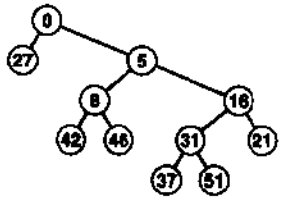
据项中提取出关键字，这都和往常一样。而且，正如我们在第三章对链表所做的那样，我们可对链接使用并行数组。我们利用三个数组，分别用于数据项、左链接和右链接。链接就是数组索引（整数），而我们则把所有代码中诸如下面的链接引用

```
x = x->l
```

用诸如下面的数组引用取代：

```
x = l[x]
```

这个方法避免了每个节点的动态内存分配的开销——不管搜索函数是什么，这些数据项都占用一个数组，而我们为每个数据项预先分配两个整数来容纳树的链接，因为已经知道：当所有数据项都在搜索结构中时，我们至少需要这个数量的空间。用于链接的空间并非一直都处在使用状态，但其存在目的是：让搜索程序无需任何用于分配的时间管理开销就可以使用它。这种方法的另外一个主要特色是它允许加入额外的数组，而完全不需要改变控制树的代码。当搜索程序返回对应一个数据项的索引时，它给出了一种立即访问该数据项的相关信息的途径，这是通过那个索引访问一个适当的数组得到的。




```
 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...
   ...
```

**图 12.11 文本字符串索引**

在这个关于一字符串索引的例子中，我们把字符串关键字定义为：从一段文字中的每个词语开始；然后我们用它们的字符串索引访问关键字，建立一个 BST。原则上关键字长度是任意的，但在具体实践中一般只检查开头的一些字符。例如，为了找出短语 never mind 是否在本文中出现，我们在树根和 call……（字符串索引 0）比较，然后在根的右孩子和 me……（索引 5）比较，然后在该节点的右孩子和 some……（索引 16）比较，最后我们在该节点的左孩子（索引 31）发现了 never mind。

实现 BST 旨在对庞大的数据项数组的搜索提供援助，这种途径有时候很有用。因为它避免了把数据项复制到 ADT 的内部表示中的额外花费，以及经由 new 分配和构造的管理开销。在下列情况下使用数组是不恰当的：空间非常宝贵且符号表显著地增大和缩小，特别是难以提前估计符号表的最大容量。如果不可能对使用空间作出准确预测，未使用的链接很可能浪费数据项数组的空间。

索引概念的一种重要应用是：在一个文本字符串中提供关键字搜索操作。程序 12.11 是这一应用的一个例子。它从一外部文件读入一个文本字符串。然后，为文本字符串中的每个位置定义一个字符串关键字（从该位置开始直到字符串的末尾），它利用字符串指针把所有关键字插入到一个符号表。字符串关键字的这种用法和一个字符串数据项的类型定义（例如练习 12.2 的那一个）不同，因为没有牵涉到存储分配。我们使用的字符串关键字可有任意长度，但我们只保持指向它们的指针，并且只着眼于足够的字符，以确定两个字符串中哪一个应该首先出现。没有任何两个字符串相等（例如它们的长度各不相同），但如果我们修改==，使得若两个字符串的一个是另一个的前缀就认为两者相等，那么我们就可以利用符号表找出一个给定的查询字符串是

否在文本中——直接调用 search 即可。

程序 12.11 从标准输入设备读取一系列查询，利用 search 确定每个查询是否在文本中，并打印文本中该查询首次出现的位置。如果符号表通过 BST 实现，那么我们根据性质 12.6 可预期：搜索将涉及到大约 $2N\ln N$ 次比较。譬如，一旦建立了索引，我们就可以在一个由大约 1 百万字符（例如 Moby Dick）构成的文本中，以大概 30 次字符串比较找到任何短语。这种应用和编制索引是一样的，因为 C 字符串指针就是指向一个字符数组的索引：如果 x 指向 text[i]，那么两个指针的差 x-text 就等于 i。

**程序 12.11　索引化一个文本字符串的范例**

这个程序假设 Item.cxx 定义了以下内容：对应数据项中字符串关键字的一个 char*数据表示，一个使用 strcmp 的重载操作符<，一个使用 strncmp 的重载操作符==，以及一个从 Item 到 char*（参阅正文）的转换操作符。主程序从一特定文件读入一个文本字符串，并利用一个符号表建立来自于各字符串的一个索引，这些字符串通过从文本字符串的每个字符出发进行定义。然后，它从标准输入设备读取查询字符串，并打印文本中找到该查询的位置（或打印 not found）。由于采用 BST 符号表实现，这个搜索即使对于巨大的字符串也能快速完成。

```
#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;
  }
```

当我们在实际应用中建立索引时，还有许多其他问题需要我们加以考虑；还有许多方法，让我们可以对字符串关键字的属性加以特别利用，以加速我们的算法。关于字符串搜索以及提供具有对字符串关键字有益功能的索引，对应这两个问题的更完善的方法将是第五部分的主要议题。

表 12.2 给出了经过实验得出的一些结果，它们为我们业已检讨的分析结果提供了支持。该表也展示了 BST 对于含有随机关键字的动态符号表的用处。

## 练习

12.66 把我们的 BST 实现（程序 12.8）修改为使用数据项的索引数组，而不是已分配的内存。利用练习 12.23 或 12.24 的驱动程序，比较你的程序和标准实现方法的性能。

12.67 修改我们的 BST 实现（程序 12.8），利用并行数组支持一个含有客户数据项处理句柄的符号表 ADT（参阅练习 12.7）。利用练习 12.23 或 12.24 的驱动程序，比较你的程序和标准实现方法的性能。

12.68 把我们的 BST 实现（程序 12.8）修改为利用以下思想表示 BST：在树节点中保持一个由含关键字的数据项组成的数组和一个由链接（每一个都和一个数据项相关）组成的数组。BST 中一个左链接对应着移动到树节点中的数组的下一个位置，而一个右链接对应着移动到另一个树节点。

○ 12.69 给出一个文本字符串的例子，使得其中对应程序 12.11 的索引构造部分的字符比较次数是字符串长度的二次函数。

12.70 修改我们的字符串索引实现（程序 12.11）——仅使用从单词边界开始的关键字建立索引（参阅图 12.11）。（对于 Moby Dick，这种改变使索引大小以原索引大小的五分之一还少）

○ 12.71 利用练习 12.38 描述的实现方法，实现程序 12.11 的一个在字符串指针数组上使用二叉搜索的版本。

12.72 对下列操作比较你从练习 12.71 所得的实现和程序 12.11 的运行时间：分别对 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$，为 $N$ 个字符构成的一个随机文本构造一索引，并为每个索引中的随机关键字进行 1000 次（失败）搜索。

**表 12.2 对符号表实现的实验研究**

这个表为构造一个符号表，然后在表中搜索每一个关键字给出了相对时间。BST 提供了搜索和插入的快速实现；所有其他方法对于这两个任务其中之一都需要二次的时间。二叉搜索一般比 BST 搜索稍快一点，但不适用于巨大的文件，除非可以对表进行预排序。标准的 BST 实现为每个树节点分配内存，而索引实现为整棵树预先分配内存（加速构造过程），并利用数组索引代替指针（减慢了搜索速度）。

| | 构造 | | | | | 搜索命中 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | 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 |

其中：

A　无序数组（练习 12.20）
L　有序链表（练习 12.21）
B　二叉搜索（程序 12.7）
T　标准二叉搜索树（程序 12.8）
T*　索引二叉搜索树（练习 12.67）

## 12.8　在 BST 的根进行的插入

在 BST 的标准实现中，每个插入的新节点都进入树底部的某处，代替某个外部节点。这种事务状态并不是一个绝对的要求：它只不过是自然递归插入算法的一个人为的要求。在这一节中，我们将考虑另外一种插入方法：我们坚持每个新节点插入到树根，因此最近插入的节点就在树的顶部。通过这种途径构造的树有一些很有趣的特性，但我们考虑这种方法的主要原因是：在两种我们在第十三章讨论的改良 BST 算法中，它扮演着至关紧要的角色。

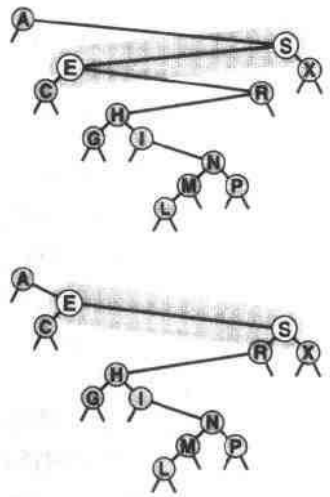



**图 12.12　一棵 BST 中的右旋转**

本图例显示了在一棵 BST 样本（上图）中的节点 S 进行一次右旋转的结果（下图）。包含 S 的节点在树中向下移动，成为它以前的左孩子的右孩子。

我们用以下步骤完成旋转：从 S 的左链接得到指向新根 E 的链接，通过复制 E 的右链接设置 S 的左链接，把 E 的右链接设置为指向 S，把从 A 指向 S 的链接改为指向 E。

旋转的作用是把 E 及其左子树向上移动一层，把 S 及其右子树向下移动一层。树的其余部分完全不受影响。

假设被插入数据项的关键字大于树根的关键字。我们可以这样着手建造一棵新树：把新数据项放入一个新的根节点，把原有根作为左子树，原有根的右子树作为新树的右子树。然而右子树也许包含一些比较小的关键字，所以我们必须做更多工作才能完成这次插入。类似地，如果被插入数据项的关键字小于树根的关键字，但大于树根左子树的所有关键字，我们依旧可建造一棵新数据项处于树根的新树；但如果左子树包含一些较大的关键字，也必须做更多工作。要把所有含较小关键字的节点移到左子树、把所有含较大关键字的节点移到右子树，大体上看这似乎是一次非常复杂的转换过程。

幸运的是，对这个问题存在一种简单的递归解决方案：它基于旋转（rotation）——对于树的一种基本转换。实质上，一次旋转允许我们使一棵树中根和其中一个根的子孙互换角色，但仍然保持节点关键字之间的次序。一次右旋转（right rotation）涉及到根和左孩子（见图 12.12）。这个旋转把根放到右子树中，实际上逆转了根的左链接的方向：在旋转之前它从根指向左孩子；进行旋转之后，它从原来的左孩子（新的根）指向原来的根（现在根的右孩子）。使得旋转起作用的巧妙的地方是：把左孩子的右链接复制成为原有根的左链接。对于旋转所牵涉到的两个节点，

这个链接指向的所有节点的关键字都介于两者之间。最后，指向原有根的链接必须改为指向新的根。一次左旋转（left rotation）的描述和刚才给出的描述一样，但要把每一处的“右”和“左”互换（见图 12.13）。

**程序 12.12 BST 中的旋转**

这对双胞胎例程在一棵 BST 上执行旋转操作。右旋转使得原来的根成为新根（根原来的左子树）的右子树；左旋转使得原来的根成为新根（根原来的右子树）的左子树。对于在节点中保存着一个计数域的实现（例如为了支持选择，我们将在 14.9 节看到这一点），我们还必须交换旋转中涉及的节点的计数域（参阅练习 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; }
```

一次旋转是涉及到三个链接和两个节点的局部改变，它允许我们在树里到处移动节点，而不影响 BST 有助于搜索的全局排序性质（参阅程序 12.12）。我们利用旋转在树里移动特定节点，并保持树的平衡状态。在 12.9 节中，我们用旋转实现删除、联结和其他 ADT 操作；在第十三章中，我们利用它们帮助构建能够提供接近最佳性能的树。

旋转操作提供了一种对根插入的简单直接的递归实现方案：递归地向适当子树插入新数据项（当完成递归操作时把它留在该子树的根），然后旋转使它成为主干树的根。图 12.14 描述了一个例子，而程序 12.13 则是这种方法的一个直接实现。这个程序是证明递归威力的极具说服力的例子——我们鼓励任何不那么信服的读者试一试练习 12.76。

图 12.15 和 12.16 显示了我们如何通过向一棵原先为空的树插入一系列节点（利用根插入方法），构造出一棵 BST。若关键字序列是随机的，一棵经由这种途径建立的 BST 和通过标准方法建立的 BST 拥有恰恰一样的随机性质。例如，性质 12.6 和 12.7 对于由根插入建立的 BST 也成立。

在实践中，根插入方法的一个优点是新近插入的关键字接近树顶部。而对新近插入关键字的搜索命中的开销也因此很可能低于标准方法的开销。这个性质十分重要，因为许多应用在它们的搜索和插入操作中恰好有这类动态混合。一个符号表可能包含大量数据项，但一大部分的搜索

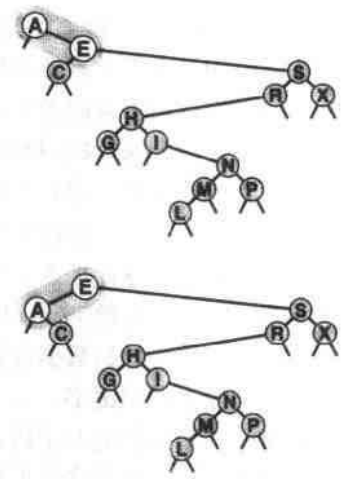



**图 12.13 一棵 BST 中的左旋转**

本图例显示了在一棵 BST 样本（上图）中的节点 A 进行一次左旋转的结果（下图）。包含 A 的节点在树中向下移动，成为它以前的右孩子的左孩子。

我们用以下步骤完成旋转：从 A 的右链接得到指向新根 E 的链接，通过复制 E 的左链接设置 A 的右链接，把 E 的左链接设置为指向 A，把指向 A 的链接（这棵树的头链接）改为指向 E。

操作也许只访问最近插入的数据项。例如在一个商业事务处理系统中，当前活动的事务可以逗留在顶部附近并可迅速处理，而无须访问已遗失的旧事务。根插入方法自动给予数据结构这种性质以及类似性质。

**程序 12.13　BST 中的根插入**

有了程序 12.12 的旋转函数，马上可以得出在 BST 的根插入一个新节点的递归函数：向适当子树的根插入新数据项，然后执行适当的旋转把它带到主干树的根。

```
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); }
```

如果我们也改造搜索函数——当出现搜索命中时把找到的节点带到树根，那么我们就得到一种自组织的搜索方法（参阅练习 12.28）了，它能使频繁访问的节点保持在树的顶部。在第十三章中，我们将看到这种思想的一个系统实现，它提供的一个符号表实现可保证拥有快速的性能特性。

正如对于我们在本章提到的几种其他方法那样，很难对实际应用中根插入方法和标准插入方法的性能比较做出一个精确的定论；这是因为具体性能依赖于符号表操作的混合，其依赖程度难以细加分析。如果已知大部分搜索操作用于最近插入的数据，则我们不能分析该算法的事实不应该成为劝阻我们使用根插入的理由，但我们始终寻求精确的性能保障——我们在第十三章的主要焦点就是构造 BST 从而能够提供这些保障的方法。

## 练习

▷ 12.73　利用根插入方法向一棵原先为空的树插入数据项，其关键字为 E A S Y Q U E S T I O N，画出所得的 BST。

12.74　给出 10 个关键字（字母 A 到 J）构成的一序列，使得：通过根插入方法把它们插入到一棵原先为空的树时，需要最大数量的比较操作以建立该树。给出使用的比较操作次数。

12.75　向程序 12.12 加入必要的代码，使它能正确地修改在旋转操作后必须改变的计数域。

○ 12.76　实现一个非递归的 BST 根插入函数（参阅程序 12.13）。

12.77　对于 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，把 $N$ 个随机关键字插入到一棵原先为空的树，建立一棵二叉搜索树，然后对 $N/10$ 个最近插入的关键字执行由 $N$ 次随机搜索构成的序列。通过实验研究，分别计算其中

搜索命中和搜索失败所使用的比较次数的平均值和标准差。对标准插入方法和根插入方法都进行这个实验，然后比较其结果。

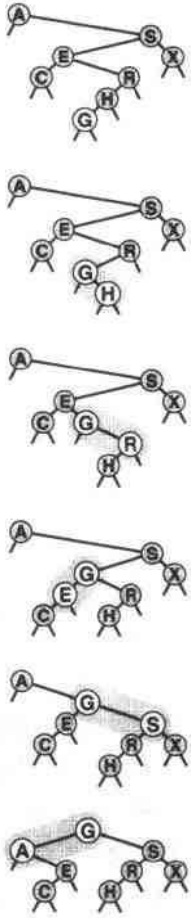


图 12.14 BST 根插入

本系列图例说明了以下处理的结果：把 G 插入到顶部的 BST，插入后用（递归）旋转把新近插入节点 G 带到树根处。这个过程等价于插入 G，然后执行一系列旋转操作把 G 带到树根处。

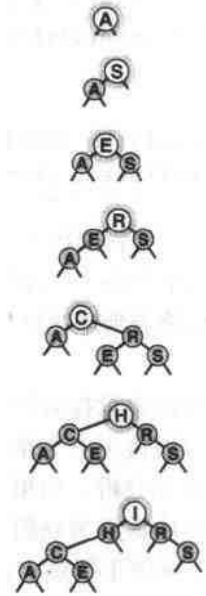


图 12.15 使用根插入的 BST 构造

本系列图例描述了向一棵原先为空的 BST 插入关键字 A S E R C H I 的结果，其中利用了根插入方法。每一个新节点都插入到树根处，且顺着它的搜索路径的链接也要修改，以得到一棵正确的 BST。

## 12.9 其他 ADT 函数的 BST 实现

对于基本的利用二叉树结构的搜索、插入和排序函数，12.5 节中给出的递归实现方法是简单直接的。在本节中我们考虑选择、联结和删除的实现。选择也有一种自然的递归实现，但其他的实现可能十分麻烦，甚至可能引起性能问题。对选择的考虑很重要，因为对

于许多应用来说，有效地支持选择和排序的能力正是 BST 比其他作为竞争对手的数据结构更可取的一个原因。一些程序员避免使用 BST，以此避免必须和删除操作打交道；在本节中我们将看到一个紧凑的实现方法，它把这些操作绑合起来，并使用了 12.8 节的“旋转到根”（rotation-to-the-root）技术。

一般地，这些操作都涉及到顺着树里的一条路径向下移动；所以对于随机 BST，我们可预期它的开销是对数水平。然而，我们不能想当然地认为：当在树上执行多个操作时，BST 仍将保持随机状态。在这一节的末尾我们将回到这个问题上来。

为了实现选择操作，我们可以使用一种递归过程，它类似于 7.8 节里描述过的基于快速排序的选择方法。为了找出一棵 BST 中含有第 $k$ 个最小关键字的数据项，我们检查左子树的节点数量。若那里存在 $k$ 个节点，那么我们返回树根处的数据项。否则，若左子树拥有 $k$ 个以上节点，我们就（递归地）在那里寻找第 $k$ 个最小节点。如果这两个条件都不成立，则左子树拥有 $t$ 个元素且 $t<k$，BST 中第 $k$ 个最小元素就是右子树的第（$k-t-1$）个最小元素。程序 12.14 是这种方法的一个直接实现。和往常一样，因为函数的每次执行最多以一次递归调用结束，我们马上可得出一个非递归版本（参阅练习 12.78）。

在 BST 节点中引入计数域的主要算法原因是为了支持选择的实现。它还允许我们支持对计数（count）操作的一个琐细实现（返回根的计数域），在第十三章我们将看到另一个用途。使用计数域的缺点是：它在每个节点都占用额外的空间，以及每个改变这棵树的函数必须更新此域。在一些以搜索和插入为主要操作的应用中，也许不值得费心来保持计数域，但如果在一个动态符号表中对选择操作的支持十分重要，为此付出的也许只是一个小小的代价。

我们可以把这个选择操作的实现改为划分操作，后者重新编排树以便把第 k 个最小元素放入树根，它使用的正是我们在 12.8 节对根插入使用的同一种递归技术：如果我们（递归地）把目标节点放到一棵子树的树根，我们就可以用一次旋转操作使它成为整棵树的根。程序 12.15 给出了这种方法的一个实现。如同旋转，分区不是一个 ADT 操作，因为它是一个转换特定符号表表示方式的函数，对客户程序来说应该是透明的。更正确地，它是一个辅助程序，我们可利用它实现 ADT 操作或者使它们更高效地运行。图 12.17 描述了这样一个例子：它以和图 12.14 同样的途径，显示了这个过程如何等价于顺着树里面的根到目标节点的

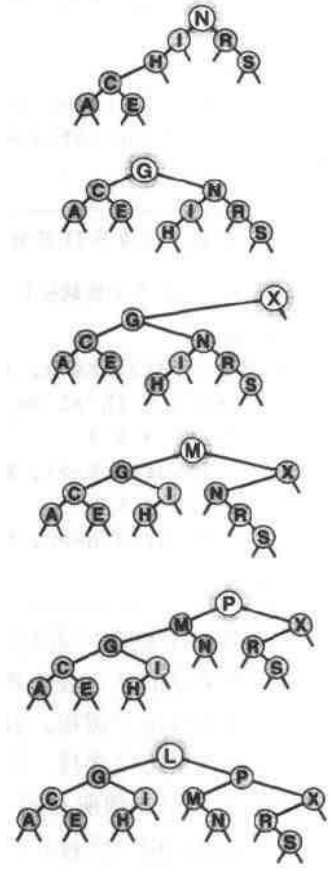


**图 12.16　使用根插入的 BST 构造（接图 12.15）**

本系列图例描述了向图 12.15 开始的 BST 插入关键字 N G X M P L 的过程。

路径向下处理，然后向上折回，执行旋转操作以把该节点向上带到树根。

**程序 12.14 对一棵 BST 的选择操作**

这个过程假设对于每个树节点都保持着子树大小。比较本程序和在一个数组中的基于快速排序的选择操作（程序 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); }
```

**程序 12.15 一棵 BST 的分区**

在递归调用后添加旋转操作就把程序 12.14 的选择函数转换成一个把 BST 中第 $k$ 个最小节点放入树根的函数。

```
void partR(link& h, int k)
  { int t = (h->l == 0) ? 0: h->l->N;
    if (t > k )
      { partR(h->l, k); rotR(h); }
    if (t < k )
      { partR(h->r, k-t-1); rotL(h); }
  }
```

为了从 BST 删除一带有给定关键字的节点，我们首先检查该节点是否处于其中一棵子树之中。如果是的话，我们用从该子树（递归）删除该节点的结果取代原来的子树。如果将被删除的节点位于树根，我们用两棵子树结合成一棵树的结果取代原树。对于完成这个结合有几种方法可供选择。图 12.18 展示了其中一种，而程序 12.16 给出了一个实现。若已知两棵 BST 中第二棵所有的关键字都大于第一棵所有的关键字，为了对它们进行结合，我们对第二棵树应用分区操作，以便把其中最小的元素带到树根。此时，这个根的左子树必须为空（否则会存在一个比根还小的元素，这是自相矛盾的），而我们可通过用一个指向第一棵树的链接取代那个链接，完成这项工作。图 12.19 显示了在一棵样本树中一系列的删除操作，它阐明了一些可能产生的情况。

这种方法是不对称的，而且在某种意义上是特别的：为什么用第二棵树中最小的关键

字作为新树的根，而不用第二棵树中最大的关键字呢？也就是说，对我们正准备删除的节点，为什么选择在树的中序遍历中用它的下一个节点取代它，而不用前一个节点呢？我们也许还希望考虑其他方法。例如，如果被删除的节点有一个空的左链接，为什么不直接用它的右孩子作为新树根，而使用右子树中带有最小关键字的节点呢？对这个基本删除程序的许多类似修正已经被提出过。不幸的是，它们都具有一个相似的缺陷：即使先前这棵树是随机的，删除以后留下的树也不是随机的。而且，事实已经表明：如果对这棵树进行大量随机的“删除-插入”操作对（参阅练习 12.84），程序 12.16 倾向于得出一棵轻微失去平衡的树（平均高度正比于 $\sqrt{N}$ ）。

在实际应用中也许不会注意到这些分别，除非 $N$ 十分巨大。但是，一种不优美的算法和不符合需要的性能特性结合起来，也是不能令人满意的。在第十二章我们将分析两种不同的方法以解决这种情况。

搜索算法要求的删除实现大大复杂于搜索实现，这种情况是很典型的。这些关键字值在对该结构的塑造中扮演了一个中心角色，因此删除一个关键字可能导致很复杂的修补工作。一种选择是使用懒惰的删除策略，把删除的节点留在数据结构中但把它们标注为“已删除”，从而在搜索中可以忽略它们。在程序 12.8 的搜索实现中，我们可跳过对这些节点的等同性测试，从而实现这种策略。我们必须确保大量被标注的节点不会引起时间或空间的过度浪费，但是如果删除操作并不频繁，其额外开销也许是微不足道的。方便的时候（例如，对于树底部的节点，这样做十分容易），我们可以把被标注节点重用于未来的插入。或者，我们可周期性地重建整个数据结构，省去被标注节点。这些考虑对任何牵涉插入和删除的数据结构都适用——它们不是符号表的专利。

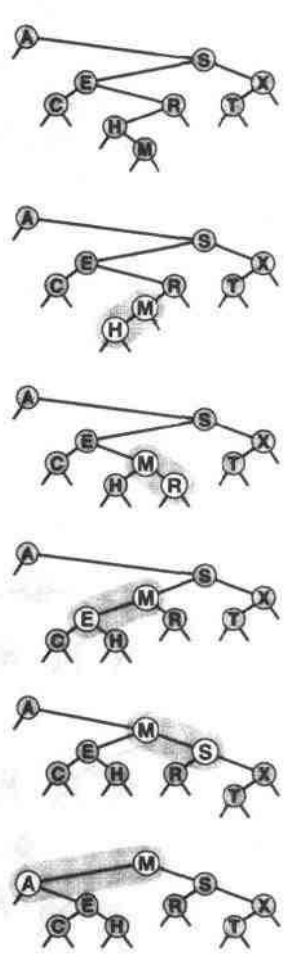


**图 12.17　一棵 BST 的分区**

本系列图例描述了对于一棵样本 BST（上图），以和根插入同样的方式使用（递归）旋转，进行关于关键字中值的分区结果（下图）。

我们通过为使用 BST 的符号表 ADT 的实现，考虑带有处理句柄的删除和联结操作的实现，结束对本章的讨论。我们假设这些句柄是链接，且省略对封装问题的进一步讨论，从而我们可以把精力集中到两个基本算法上来。

在一个删除带有给定句柄（链接）的节点的函数实现中，主要的挑战和链表中遇到的一样：我们需要更改结构中指向正被删除节点的指针。至少存在四种解决这个问题的方法。首先，我们可向每个树节点加入第三个链接，它指向节点的双亲。这种安排的缺点是维护额外的链接是一件很麻烦的工作，我们在之前的几个场合中已经提到了这一点。其次，我

们可以利用数据项中的关键字在树里搜索，当我们找到一个匹配指针就停下来。这个方法的缺陷是：一个节点的平均位置接近树的底部，因而这种方法需要在树中走一段不必要的旅程。第三种方法，我们可以使用用一个引用或者一个指向该节点的指针的指针作为句柄。这种方法是一种利用 C++和 C 的解决方案，但其他很多语言不支持这种做法。第四种方法，我们可采用一种懒惰方法，标注出已删除的节点并周期性地重建数据结构，正如刚刚描述的那样。

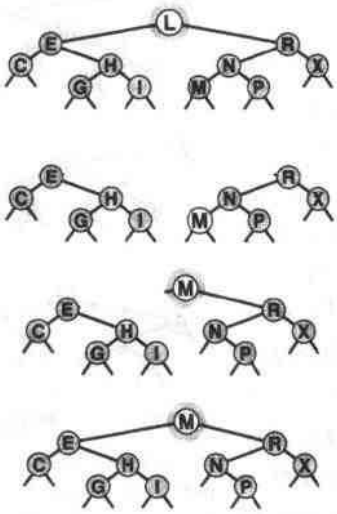



图 12.18 在一棵 BST 中删除根

本图例显示了删除一棵样本 BST（顶部图）的根所得结果（底部图）。首先我们删除该节点，留下两棵子树（从上往下数第二个图）。然后我们对右子树分区，以使最小的元素处于树根（从上往下数第三个图），使左链接指向一棵空子树。最后，我们用一个指向原树的左子树的链接代替该链接（底部图）。

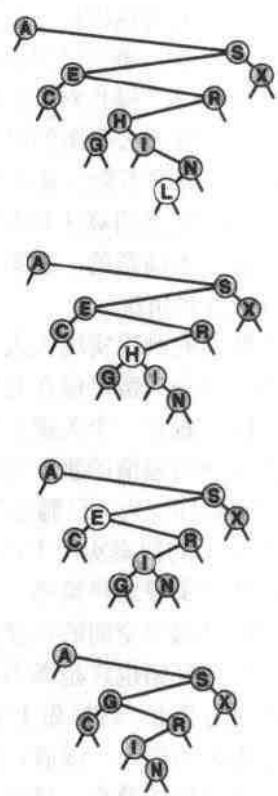



图 12.19 BST 节点的删除

本系列图例描述了从顶部的 BST 删除带有给定关键字 L、H 和 E 的节点的结果。首先，L 被直接删除，因为它在树的底部。接着，H 被它的右孩子 I 取代，因为 I 的左孩子为空。最后，E 被它在树中的后继节点——G 所取代。

我们必须对一级符号表 ADT 考虑的最后一个操作是联结操作。在一个 BST 实现中，这等于归并两棵树。我们怎样把两棵 BST 联结成一棵呢？有各种各样的算法可进行这项工作，但每种都有特定缺陷。例如，我们可遍历第一棵 BST，把它的每个节点插入到第二棵

BST（这个算法是单程过程：利用对第二棵 BST 的插入操作作为一个对第一棵 BST 的遍历的函数参变量）。这个解决方案不具有线性运行时间，因为每个插入操作可能需要线性时间。另一个想法是遍历两棵 BST，把数据项输入到一个数组，对它们执行归并，然后建立一棵新的 BST。这种操作能够在线性时间内完成，但它还要使用一个可能很大的数组。

**程序 12.16　在一棵 BST 中删除带有给定关键字的节点**

这个删除操作的实现删除在 BST 中遇到的第一个带有给定关键字 v 的节点。它从上向下处理，为适当的子树作递归调用，直到要删除的节点位于树根处。然后，它用两棵子树结合的结果取代该节点——右子树中最小的节点成为树根，接着把它的左链接设置成指向左子树。

```
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()); }
```

程序 12.17 是联结操作的一种紧凑的线性时间的递归实现。首先，我们利用根插入，把第一棵 BST 的根插入到第二棵 BST。这个操作使我们得到两棵已知其关键字小于这个根的子树，两棵已知其关键字大于这个根的子树，因此我们通过以下处理得到结果：（递归）把前面一对结合到这个根的左子树，把后面一对结合到这个根的右子树（!）。在一次递归调用中，每个节点最多只可以成为根节点一次，所以总计时间是线性的。

**程序 12.17　两棵 BST 的联结**

如果任一 BST 为空，另外一棵就是结果。否则，我们通过以下步骤结合两棵 BST：（随意地）挑选第一棵树的根作为树根，用根插入方法把它插入到第二棵，然后（递归地）结合左子树对和右子树对。

```
private:
  link joinR(link a, link b)
    {
      if (b == 0) return a;
      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); }
```

图 12.20 显示了一个例子。像删除操作一样，这个过程是不对称的而且可以产生很不平衡的树；但随机化提供了一个简单的修正方法，我们将在第十三章看到这一点。要注意的是：对联结操作使用的比较操作次数即使在最坏情况下也必须至少达到线性水平；否则我们可以开发一种使用少于 NlgN 次比较的排序算法，它利用一种诸如自下而上归并排序的方法（参阅练习 12.88）。

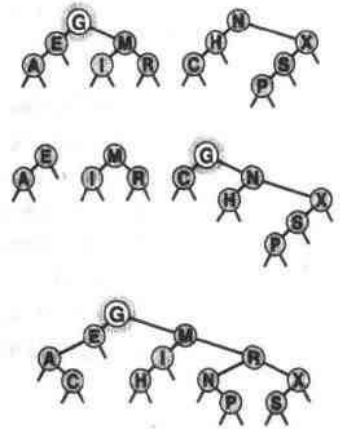


**图 12.20 两棵 BST 的联结**

本图例显示了结合两棵样本 BST（上图）的结果（下图）。首先，我们利用根插入方法，把第一棵树的根 G 插入到第二棵树（从上往下数第二个图）。我们就得到两棵关键字小于 G 的子树和两棵关键字大于 G 的子树。（递归地）结合这两对子树即可得出结果（下图）。

在对于联结和删除操作的转换期间，我们还没有引入维护 BST 节点中的计数域所必需的代码，它对于我们希望还能支持选择操作的应用（程序 12.14）也是必需的。在概念上来说这个任务很简单，但仍要谨慎面对。处理的一种系统方法是实现一个小型实用程序，它用一个比该节点所有孩子的计数域总和大 1 的值设置该节点的计数域，然后对每一个链接被更改的节点调用这个程序。更具体地说，我们可对程序 12.12 的 rotL 和 rotR 中的节点都这样处理；这样已能满足程序 12.13 和程序 12.15 的转换需要了，因为它们单独地使用旋转操作转换树。对于程序 12.16 的 joinLR、removeR 以及程序 12.17 的 join，在 return 语句之前对被返回的节点调用“节点计数”更新程序就足以完成任务了。

BST 的基本搜索、插入和排序操作都易于实现，而且即使在操作序列中只存在少量的随机性，它也可以良好地执行。因此 BST 广泛地用于动态符号表。它们还可接纳简单的递归解决方案，以支持其他类型操作；我们在本章中的选择、删除和联结操作中已经看到过这一点，而且也将在本书的稍后部分看到许多例子。

尽管它们很有用，在应用中使用 BST 仍然存在两个主要缺点。第一个是它们需要把大量空间用于链接。我们常常认为链接和记录具有大约相等的大小（比方说一个机器字）——如果的确如此的话，那么一个 BST 实现把它分配得到的内存空间的三分之二用于链接，只有三分之一用于关键字。在使用大型记录的应用中这个作用不那么重要，但在使用大型指针的环境中就有很大的影响。如果内存资源十分宝贵，我们也许宁愿用第十四章的其中一种开放寻址混编方法，而不使用 BST 了。

使用 BST 的第二个缺点是树的平衡性可能变得很差，并引起性能急剧降低。在第十三章中，我们将分析几种方法以提供性能上的保障。如果用于链接的内存空间足够的话，这些算法使 BST 成为用作符号表 ADT 实现基础的一个极具吸引力的选择，因为对于一个有用 ADT 操作的大型集合，它们可保证得到快速性能。

## 练习

▷ 12.78　实现一个非递归 BST 选择函数（参阅程序 12.14）。

▷ 12.79　向一棵原先为空的树插入带有关键字 E A S Y Q U T I O N 的数据项，然后删除 Q，画出得出的 BST。

▷ 12.80　向一棵原先为空的树插入带有关键字 E A S Y 的数据项，再向另一棵原先为空的树插入带有关键字 Q U E S T I O N 的数据项，然后结合两个结果。画出得出的 BST。

12.81　实现一个非递归 BST 删除函数（参阅程序 12.16）。

12.82　实现对应 BST 的删除（程序 12.16）的一个版本，它删除树中带有和给定关键字相等关键字的所有节点。

○ 12.83　改变我们基于 BST 的符号表实现，以支持客户数据项处理（参阅练习 12.7）；加入对一个复制构造函数、一个重载赋值操作符和一个析构函数的实现（参阅练习 12.6）；加入删除和联结操作；并利用你从练习 12.22 得到的驱动程序测试你的一级符号表 ADT 接口以及实现。

12.84　在一棵 $N$ 个节点构成的随机树中进行一个长系列的交错随机插入和删除操作，运行实验以确定一棵 BST 的高度如何随之增长。其中 $N = 10$，100 和 1000，对于每个 $N$ 执行次数达到 $N^2$ 的“插入-删除”操作组合。

12.85　实现删除操作（参阅程序 12.16）的一个版本，它随机决定：用于取代被删除节点的是它在树里的前驱，还是它在树里的后继。对于这个版本，运行练习 12.84 所描述的实验。

○ 12.86　实现删除操作的一个版本，它以根插入的方式（程序 12.13），利用一个递归函数，通过旋转操作把被删除节点移动到树的底部。当你的程序从一棵 31 个节点的完全树中删除根时，画出产生的树。

○ 12.87　运行实验以确定一棵 BST 的高度如何随着以下过程增长：重复地把树根处的数据项重新插入到树中，这棵树是由在一棵 $N$ 个节点构成的随机树中结合这个根的子树而产生的，其中 $N = 10$、100 和 1000。

○ 12.88　实现基于联结操作的自下而上归并排序的一个版本：先把关键字放入 $N$ 棵单节点的树中；然后成对结合单节点的树，得到 $N/2$ 棵双节点的树；然后成对结合双节点的树，得到 $N/4$ 棵四节点的树，以此类推。

12.89　实现联结操作（参阅程序 12.17）的一个版本，它随机决定：用第一棵树的根还是第二棵树的根作为所得树的根。对于这个版本，运行练习 12.87 所描述的实验。

# 第十三章

# 平　衡　树

前一章提到的 BST 算法对于许多种类的应用都工作得很好，但它们的确都有在最坏情况下性能较差的问题。而且，一个颇为尴尬的事实是：标准 BST 算法性能大大降低的最坏情况。就像对于快速排序算法那样，如果算法的使用者不加以注意就很容易在实践中发生。已经有序的文件、含有大量重复关键字的文件、顺序倒转的文件、大小关键字交替出现的文件或者其中任何大的片段具有简单结构的文件，它们都可引起二次的 BST 构造次数和线性的搜索次数。

在理想情况下，我们可以使树保持完美的平衡状态，就像图 13.1 描述的那棵树那样。这个结构和二叉搜索树相对应，因此我们可保证所有搜索能够在少于 $\lg N+1$ 次比较内完成；但是对于动态插入和删除，它的维护需要很大的开销。对于全部外部节点都位于树底部的一层或至多两层的任何 BST，这个搜索性能的保障都成立；因为存在许多这样的 BST，我们在调整树以达到平衡状态的过程中有相当大的灵活性。如果我们满足于接近最优的树，那么甚至可以得到更大的灵活性。例如，存在大量高度少于 $2\lg N$ 的 BST。如果我们放松标准但可以保证我们的算法只建立这样的 BST，那么就可以针对性能急剧下降的最坏情况提供保障——这正是我们在实际应用中希望一个动态数据结构所能达到的。作为一个附带收益，我们还可获得更佳的平均情况下的性能。

在 BST 中产生更好的平衡的一种途径是周期性地直接使它们重新平衡。的确，我们可以通过程序 13.1 所示的递归方法（参阅练习 13.4），在线性时间内完全平衡大多数 BST。这样的重新平衡操作很可能可以对随机关键字改善性能，但并不能针对一个动态符号表在最坏情况下的二次性能提供保障。一方面，在重新平衡操作之间插入一系列关键字，其插入时间可能以该系列的长度的二次函数递增；另一方面，我们不希望频繁地重新平衡巨大的树，因为每次重新平衡操作至少要花费该树大小的线性时间。这种性能上的折中使得我们难以利用全局重新平衡来保障动态 BST 中的快速性能。所有我们即将考虑的算法，都在它们在树里游历的过程中进行递增的、局部的操作，这些操作集中地改善整棵树的平衡性，然而它们从来不必以程序 13.1 的方式遍历树里的所有节点。

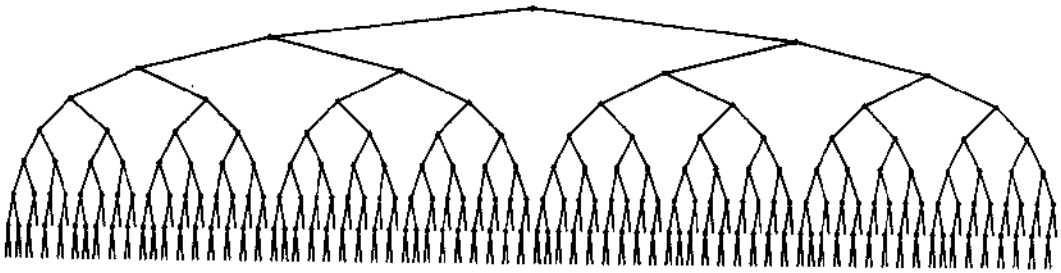


图 13.1　一棵完美平衡的大型 BST

这棵 BST 的外部节点全都落入底部两层的其中一层，而任何搜索的比较次数都和对于同一关键字的二叉搜索所使用的比较次数相等（如果这些数据项都位于一个有序数组中）。一个平衡树算法的目的是：使一棵 BST 保持处于尽可能地接近像本树这样的良好平衡状态，同时依然可支持高效的动态插入、删除和其他的字典 ADT 操作。

为基于 BST 的符号表实现提供性能保障的问题，对于以下主题的分析给予我们一个极好的讨论机会：当我们要求性能保障时，我们到底想获得什么。我们将看到对这个问题的解决方案，它们分别是算法设计中提供性能保障的三种一般方法的精华例子：我们可以随机化（randomize）、分期化（amortize）或最优化（optimize）。我们现在依次考虑这些方法。

随机化算法向算法本身引入随机决策的产生，以大大减少一种最坏情况出现的机会（无论输入内容是什么）。我们已经看到过这种方案的一个主要例子，当时我们利用一个随机元素作为快速排序中的分区元素。在 13.1 和 13.5 节中，我们将分析随机化 BST 和跳跃表（skip list）——两种在符号表实现中利用随机化操作、为所有符号表 ADT 操作都给出高效实现的简单方法。这些算法简单而且适用性广，却在数十年中都没被人们所发现（参见“第四部分参考资料”）。证明这些算法有效的分析不属于基础内容，但这些算法易于理解、实现和投入实际应用。

**程序 13.1　平衡一棵 BST**

本递归函数利用来自程序 12.15 的分区函数 partR，在线性时间内使一棵 BST 达到完美平衡状态。我们先划分以便把中值节点放入树根，然后（递归地）对子树执行同样的操作。

```
void balanceR(link& h)
  {
    if ((h == 0) || (h->N == 1)) return;
    partR(h, h->N/2);
    balanceR(h->l);
    balanceR(h->r);
  }
```

分期化方法是每次都做额外的工作，以求避免稍后更多的工作，它能对每次操作的平均开销（所有操作的开销总和除以操作次数）提供得到保障的上界。在 13.2 节里，我们将分析（splay BST），这是一种我们用于为符号表实现提供这样的性能保障的 BST 变形。这种方法的开发对分期化概念的发展是一个有力的推动（参见“第四部分参考资料”）。该算法是我们在第十二章讨论的根插入方法的一个直接扩展，但证明其性能界限的分析十分复杂。

最优化方法不怕麻烦地为每个操作提供性能保障。业已发展出多种采用这个途径的方法，其中一些可追溯到 20 世纪 60 年代。这些方法要求我们在树里面保持一些结构信息，而典型的情况是：程序员发现这些算法的实现是件十分令人烦恼的事情。在本章中，我们将分析两个简单的抽象，它们不但使得实现过程简单直接，还可令开销达到接近最优性能的上界。

在分别利用这三种方法，分析带有得到保障的快速性能的符号表 ADT 之后，我们以对它们的性能特性的比较结束本章的讨论。除了由每种算法提供的性能保障的不同本性所表明的差别外，这些方法各自附带了时间或空间上的（相对轻微的）开销，以便提供这些保障；而开发一个真正最佳的平衡树 ADT 依然是一个研究目标。但我们在本章考虑的算法也是十分重要，对于各种各样的应用，它们能够提供动态符号表中搜索和插入（和几个其他的符号表 ADT 操作）的快速实现。

## 练习

○ 13.1 实现一个高效的函数，重新平衡在其节点中没有一个计数域的 BST。

13.2 修改程序 12.8 的标准 BST 插入函数——每当符号表的数据项数量达到 2 的次幂，就利用程序 13.1 来重新平衡树。$N = 10^3$，$10^4$，$10^5$ 和 $10^6$，对于以下任务比较你的程序和程序 12.8 的运行时间：（1）从 $N$ 个随机关键字建立一棵树；（2）在所得的树中搜索 $N$ 个随机关键字。

13.3 当向一个符号表插入 $N$ 个关键字构成的一个递增序列时，估算你从练习 13.2 所得程序使用的比较次数。

●●13.4 说明程序 13.1 对一棵退化树的运行时间和 $N\lg N$ 成正比。然后对该树给出一个尽可能弱的限制条件，它意味着程序在线性时间内运行。

13.5 修改程序 12.8 的标准 BST 插入函数——对于任何遇到的节点，若在它其中一棵子树中有少于四份之一它的节点，关于中值进行分区。$N = 10^3$，$10^4$，$10^5$ 和 $10^6$，对于以下任务比较你的程序和程序 12.8 的运行时间：（1）从 $N$ 个随机关键字建立一棵树；（2）在所得的树中搜索 $N$ 个随机关键字。

13.6 当向一个符号表插入 $N$ 个关键字构成的一个递增序列时，估算你从练习 13.5 所得程序使用的比较次数。

● 13.7 扩展你对练习 13.5 的实现，以便在执行删除操作时用同样方式重新平衡。在一棵 $N$ 个节点构成的随机树中执行一个由交错随机插入和删除构成的操作长序列，运行实验以确定一棵 BST 的高度如何随之增长。其中 $N = 10$，100 和 1000，对于每个 $N$ 执行次数达到 $N^2$ 的“插入-删除”操作组合。

## 13.1　随机化 BST

为了分析二叉搜索树的平均情况下的性能开销，我们做出以下假设：按随机次序插入数据项（参阅 12.6 节）。在 BST 算法的环境中，该假设的主要推论是树里的每个节点成为树根的可能性都相等，且这个性质也对子树成立。值得注意的是，有可能对这个算法引入随机性，使得无须对节点插入的次序作任何假设，性质也成立。这个思想很简单：当我们向一棵 $N$ 个节点构成的树插入一个新节点时，新节点应该以概率 $1/(N+1)$ 出现在树根，因此我们就用这个概率做出进行根插入的随机决定。否则，我们递归地使用该方法插入新记录：如果该记录的关键字小于树根的关键字，就插入到左子树；如果该记录的关键字大于树根的关键字，就插入到右子树。程序 13.2 是这个方法的一个实现。

从非递归的角度上看，进行随机插入相当于对该关键字执行一次标准搜索，在每一步都随机决定应该继续搜索还是终止搜索并进行根插入。因此，新节点可能被插入到它的搜索路径上的任何地方，如图 13.2 所示。从概率的意义上来说，标准 BST 算法和根插入方法的简单概率结合提供了性能保障。

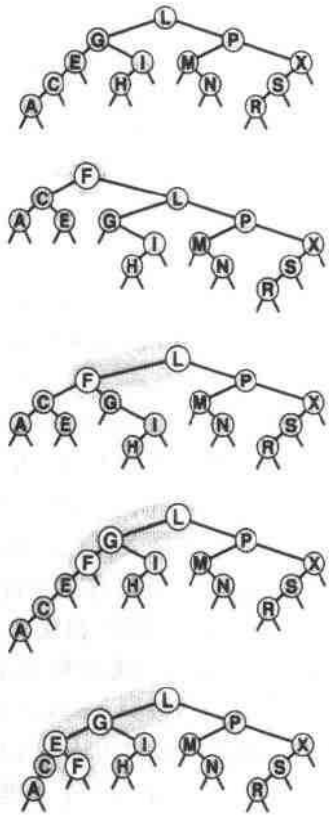


**图 13.2　对一棵随机化 BST 的插入**

一个新记录在一棵随机化 BST 中的最后位置可以是该记录的搜索路径上的任何地方，这依赖于搜索期间随机化决定的结果。本图显示了把一个关键字为 F 的记录插入到一棵样本树（顶部图）时，它的每个可能最终位置。

**性质 13.1**　*建立一棵随机化 BST 等价于从关键字的一个随机初始分布建立一棵标准 BST。我们使用大约 $2N\ln N$ 次比较构造一棵 $N$ 个数据项的随机化 BST（无论数据项对于插入操作以何种顺序出现），而使用大约 $2\ln N$ 次比较在这样一棵树中进行搜索。*

每个元素成为树根的可能性都相等，且这个性质对于它的两棵子树也相等。根据构造过程，可知这个命题的第一部分成立，但我们还需要一个谨慎的概率论证，以此表明根插入方法在子树中保留随机性（参见“第四部分参考资料”）。■

**程序 13.2　随机化 BST 插入**

本函数随机做出决定：是用程序 12.13 的根插入方法，还是用程序 12.8 的标准插入方法。在一棵随机 BST 中，每个节点位于树根的可能性都相等；因此我们通过以概率 $1/(N+1)$ 把一个新节点放入一棵大小为 N 的树的根，得到随机树。

```
  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); }
```

随机化 BST 和标准 BST 在平均情况下的性能差别很细微，但却是本质上的差别。平均开销是相同的（尽管随机化树的比例常数稍稍高一点），但对于标准树，其结果依赖于以下假设：对于插入操作，数据项以它们的关键字的一种随机排序出现（所有排序的可能性相等）。在许多实际应用中这个假设无效，因此随机化算法的意义就在于它允许我们撤除该假设，并改为依靠概率定律和随机数生成器中的随机性。如果数据项被插入时它们的关键字有序、逆序或者依照其他任何顺序，BST 仍然是随机的。

图 13.3 描述了为一个关键字样本集合构造一棵随机化树的过程。因为算法作出的决定是随机化的，每次我们运行这个算法，得出的树序列都可能不同。图 13.4 显示了一棵随机化树，它由一个其关键字按递增顺序排列的数据项集合构造，但显示出和一棵由随机排序数据项构造的标准 BST 拥有同样的性质（比较图 12.8）。

但依然存在随机数生成器在每个机会都产生错误的决定、并因此得出极不平衡的树的可能性。但我们可以用数学分析这个可能性，并证明它微小得几乎不可能出现。

**性质 13.2** 一棵随机化 BST 的构造开销大于平均水平 $\alpha$ 倍的概率小于 $e^{-\alpha}$。

Karp 在 1995 年开发出对概率递归关系的一个一般解决方案，它蕴涵着这个结果和其他具有相同特性的类似结果（参见“第四部分参考资料”）。■

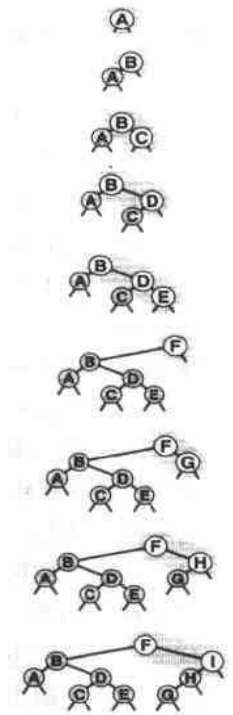


**图 13.3 一棵随机化 BST 的构造**

本系列图例描述了利用随机化插入、向一棵原先为空的 BST 插入关键字 A B C D E F G H I 的结果。底部的树看起来是通过标准 BST 算法、按随机顺序插入同样的关键字而建立的。

例如，建立一棵含 100000 个节点的随机化 BST 需要大约 2.3 百万次比较操作，但比较次数超过 23 百万的概率远远小于 0.01%。对于满足处理这样大小的真实数据集合的实际要求，这样一个性能保障是绰绰有余了。当对这样的任务使用标准 BST 时，我们不能提供如此一个保障：例如，如果数据中存在突出的顺序关系，我们将要面对性能上的问题；这种情况在随机数据中不大可能出现，但由于许多原因，它在真实数据中的出现绝对不是不寻常的事情。

根据同样的论证，一个类似于性质 13.2 的结果对于快速排序的运行时间也成立。但该结果在这里更重要，因为它还意味着树里面的搜索操作的开销接近平均水平。不考虑构造这棵树的任何额外开销，我们可以利用标准 BST 实现执行搜索操作，所需开销只依赖于树的形状，而且平衡处理根本不需要额外开销。在典型的应用中，搜索操作的使用次数远远高于其他任何操作，则以上性质就十分重要了。例如，上一段提到的 100000 个节点的 BST 可能保存着一本电话目录，而且可能用于数百万次搜索。我们几乎可以肯定：每次搜索的开销将在平均开销——大约 23 次比较的一个很小的常数因子范围内；而且对于实际用途，我们不必担忧大量搜索操作将耗费接近 100000 次比较的可能性。而用标准 BST 的话，我们就必须考虑这个问题了。

**程序 13.3　随机化 BST 结合**

本函数利用了和程序 12.17 同样的方法，不同点是：关于哪一个节点用作结合树的根，它利用了确保每个节点成为树根的可能性都相等的概率，做出一个随机化决定，而不是武断的决定。私有成员函数 fixN 把 b->N 更新为子树中对应域的总和加 1（对于空树则加 0）。

```
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); }
```

随机化插入的其中一个主要缺点是：每次插入期间在每个节点生成随机数的开销。一

个高质量的系统支持随机数生成器，也许会尽力使产生的伪随机数具有比随机化 BST 所要求的更好的随机性。所以在某些实际环境中（例如，若数据项次序随机这个假设有效），构造一棵随机化 BST 也许比构造一棵标准 BST 费时更多。正如我们对快速排序所做的那样，我们可以通过使用以下的数字来削减开销：数字未达到完美随机，但它们生成的成本低而且足够地类似于随机数字，使得对于在实际应用中可能出现的关键字插入序列，它们能够避免 BST 在最坏情况下出现糟糕的性能（参阅练习 13.14）。

随机化 BST 的另一个潜在缺点是：它们需要在每个节点有一个域用于记录该节点的子树的节点数量。这个域占用的额外空间可能成为大型树的一个缺陷。另一方面，正如我们在 12.9 节讨论的那样，这个域存在的必要性也许是出于其他原因——比方说支持选择操作，或者提供对数据结构完整性的一个校验。在这些情况下，随机化 BST 不会导致额外的空间开销，是一个极具吸引力的选择方案。

**程序 13.4 随机化 BST 中的删除**

我们使用对标准 BST 所用的同样的 remove 函数（参阅程序 12.16），但用这里显示的一个函数取代原来的 joinR 函数。关于用被删除节点的前驱节点还是后继节点取代它，这里的 joinR 函数利用了确保每个节点成为所得树的根的可能性都相等的概率，做出一个随机化决定，而不是武断的决定。为了正确地保持节点的计数值，我们还需要引入一个对应 h 的 fixN 调用（参阅程序 13.3），作为 removeR 的最后语句。

```
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; }
  }
```

在树中保留随机性的基本指导原则还导至另一个结果：在依然能生成随机树的同时，高效地实现删除、联结和其他符号表 ADT 操作。

为了联结一棵 $N$ 节点的树和一棵 $M$ 节点的树，我们使用来自第十二章的基本方法，不同的地方是：我们基于推论“结合树的根必须以概率 $N/(N+M)$来自于 $N$ 节点的树，以概率 $M/(N+M)$来自于 $M$ 节点的树”，对选择树根作出随机化决定。程序 13.3 是这个操作的一个实现。

以同样的途径，我们用一个随机化决定取代删除算法中的武断决定，如程序 13.4 所示。这个方法对应着一个我们没有在删除标准 BST 的节点考虑使用的选择，因为若缺少随机化操作，它似乎将产生不平衡的树（参阅练习 13.21）。

**性质 13.3** 用随机化插入、删除和联结操作组成的一个任意序列生成一棵树，等价于从

该树关键字的一个随机排列建立一棵标准 BST。

正如对性质 13.1 那样，需要一个谨慎的概率论证来确立这个结论（参见“第四部分参考资料”）。

证明关于概率算法的事实要求对概率论有充分的了解，但对于使用该算法的程序员，理解这些证明不一定是个必要条件。谨慎的程序员将检查像性质 13.3 这样的要求（例如检查随机数生成器的质量，或该实现的其他性质）——无论它们怎样被证明，因此可以充满自信地使用这些方法。对于支持一个完整的、具有接近最佳的性能保障的符号表 ADT，随机化 BST 也许是最容易的方法；它们也因此适用于许多实际应用。

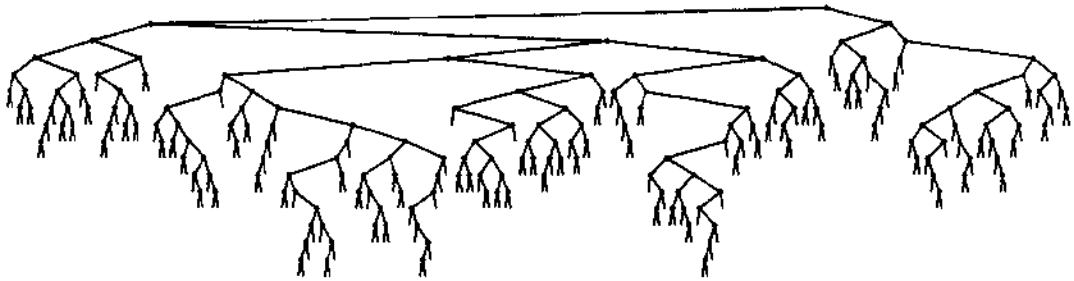


**图 13.4　一棵大型随机化 BST**

这棵 BST 是利用随机化插入、向一棵原先为空的树按递增顺序插入 200 个关键字的结果。这棵树看起来是由随机次序的关键字建立的（参阅图 12.8）。

## 练习

▷ 13.8　画出由以下处理所得的随机化 BST：向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项，并采用一个不良随机化函数——每当树大小为奇数时，它采取根插入方案。

13.9　对 $N$ = 10 和 100，编写一个驱动程序，它执行以下的实验 1000 次：利用程序 13.2，向一棵原先为空的随机化 BST 插入关键字从 0 到 $N$-1（依照这个顺序）的数据项。然后对于每个 $N$，打印出对以下假设的 $x^2$ 统计：每个关键字以概率 $1/N$ 落入树根（参阅练习 14.5）。

○ 13.10　给出 F 落入每一个图 13.2 描述的位置的概率。

13.11　编写一个程序，对于搜索路径上的每个节点，计算一次随机化插入在一棵给定树的其中一个内部节点结束的概率。

13.12　编写一个程序，计算一次随机化插入在一棵给定树的其中一个外部节点结束的概率。

○ 13.13　实现程序 13.2 中随机化插入函数的一个非递归版本。

13.14　画出由以下处理所得的随机化 BST：向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项，其中利用了程序 13.2 的一个版本——用决定是否切换到根插入的测试（111 % h - >N）== 3 代替涉及 rand()的表达式。

13.15　利用程序 13.2 的一个版本——用决定是否切换到根插入的测试（111 % $h$ - >$N$）== 3 代替涉及 rand()的表达式，重做练习 13.9。

13.16 说明导致关键字 E A S Y Q U T I O N 被建立为一棵退化树（关键字有序，左链接为空）的随机化决定序列。这个结果发生的概率是多少？

13.17 向一棵原先为空的树依照顺序 E A S Y Q U T I O N 插入这些关键字，有可能每棵包含这些关键字的 BST 都由某个随机化决定序列构造吗？解释你的答案。

13.18 对于 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$，把 $N$ 个随机关键字插入到一棵原先为空的树，建立一棵随机化 BST。通过实验研究，分别计算其中的搜索命中和搜索失败所使用的比较次数的平均值和标准差。

○ 13.19 画出由以下处理所得的 BST：用程序 13.4 从你在练习 13.14 中的树中删除 Q，其中利用测试（111 %（a ->N + b ->N））< a ->N 来决定是否在树根处和 a 联结。

13.20 画出由以下处理所得的 BST：向一棵原先为空的树插入带有这些关键字 E A S Y 的数据项，然后向一棵原先为空的树插入带有这些关键字 Q U E S T I O N 的数据项，再结合两个结果。其中利用程序 13.3 和练习 13.19 描述的测试。

13.21 画出由以下处理所得的 BST：向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项，然后利用程序 13.4 删除 Q，采用一个永远返回 0 的不良随机化函数。

13.22 在一棵 $N$ 个节点构成的随机树中，进行一个利用程序 13.2 和 13.3 的交替随机插入和删除操作长序列，通过实验确定一棵 BST 的高度如何随之增长。其中 $N$ = 10，100 和 1000，对于每个 $N$ 执行次数达到 $N^2$ 的“插入-删除”操作组合。

○ 13.23 比较你从练习 13.22 所得结果和以下结果：利用程序 13.2 和 13.3 对一棵 $N$ 个节点构成的随机树删除并重新插入最大关键字，其中 $N$ = 10，100 和 1000，对于每个 $N$ 执行次数达到 $N^2$ 的“插入-删除”操作组合。

13.24 以你从练习 13.22 所得程序为工具，确定它对于每个被删除的数据项，对 rand()的平均调用次数。

## 13.2 发散 BST

在 12.8 节的根插入方法中，我们通过利用左右旋转，达到了把新近插入节点带到树根的主要目标。在这一节中，我们分析的是：我们可以如何修改根插入，使得旋转操作也可在某种意义上使树平衡。

我们不考虑（递归地）使用把新近插入节点带到树顶部的单一旋转操作，而考虑把节点从作为树根的孙子之一的一个位置带到树顶部的两个旋转操作。首先，我们执行一次旋转，使节点成为树根的一个孩子。然后，执行另一次旋转，使它成为树根。根据从根到正被插入的节点的两个链接是否

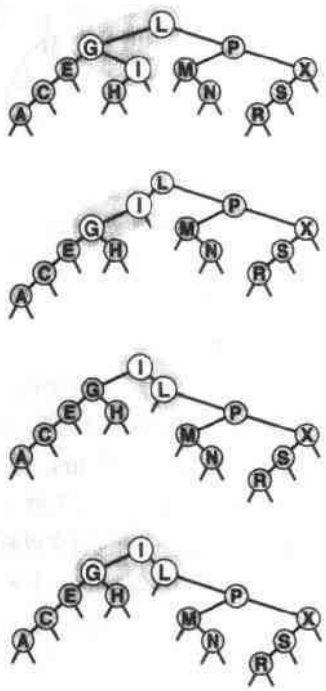



**图 13.5 一棵 BST 中的双重旋转（不同定向）**

在这棵样本树（顶部图）中，在 G 的一次左旋转及紧接着的在 L 的一次右旋转，把 I 带到树根（底部图）。这些旋转操作可以完成一个标准或发散 BST 的根插入过程。

以相同方式定向，存在两种本质不同的情况。图 13.5 显示了这两个方向不同的情况；图 13.6 的左边部分显示了两个方向相同的情况。当从根到正被插入的节点的两个链接以相同方式定向时，存在另一种可供选择的处理方案：直接在树根执行两次旋转操作，正如图 13.6 的右边部分所显示的那样。而发散 BST 正是基于这个结论。

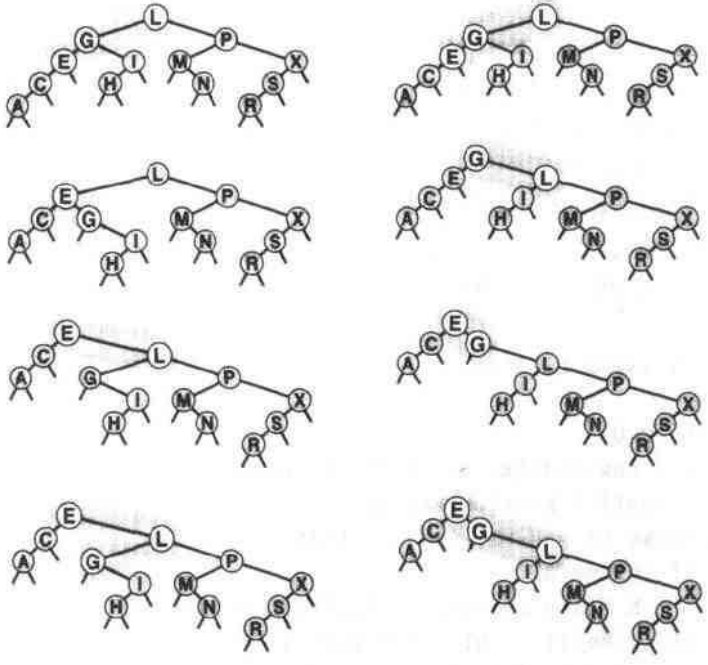


**图 13.6　一棵 BST 中的双重旋转（相同定向）**

当一个双重旋转操作的两个链接以相同方向定向时，我们有两种选择。通过标准的根插入方法，我们首先执行较低的旋转（左图）；若通过发散插入方法，我们首先执行较高的旋转（右图）。

发散插入（splay insertion）利用图 13.5（从根到搜索路径上的孙子的链接具有不同定向时，用标准根插入方法）和图 13.6 右边部分（从根到搜索路径上的孙子的链接具有相同定向时，在树根执行两次旋转）显示的转换，把新近插入节点带到树根。用这种方法建立的 BST 就是发散 BST（splay BST）。程序 13.5 是发散插入的一个递归实现；图 13.7 描述了单独一次插入的例子，而图 13.8 显示了一棵样本树的构造过程。发散插入和标准根插入之间的区别也许显得不太合理，但它非常重要：发散操作消除了最坏情况下的二次水平性能，而这正是标准 BST 的主要缺陷。

**性质 13.4**　当向一棵原先为空的树插入 $N$ 个节点、建立一棵发散 BST 时，所使用的比较次数是 $O(N\lg N)$。

这个界限是一个我们很快就要讨论的更强有力的性质——性质 13.5 的推论。■

这个 O 表示式所蕴涵的常数是 3。例如，要利用发散插入建立一棵含 100 000 节点的

BST，需要的比较次数永远少于 5 百万。这个结果不保证得出的搜索树将良好地平衡，也不保证每种操作都将高效地完成；但它对于总计运行时间的潜在保证意义重大，而且我们在实践中观察到的实际运行时间很可能更少。

**程序 13.5 BST 中的发散插入**

本函数和程序 12.13 的根插入算法的不同之处仅在于一个实质细节：如果搜索路径的方向是“左-左”或“右-右”，则该节点可用一次双重旋转从树的顶部而不是树的底部带到树根（参阅图 13.6）。

对于源自树根的搜索路径，本程序检查其中的两个步骤的四种可能性，并执行适当的旋转操作：

左-左：在树根右旋转两次；

左-右：在左孩子左旋转，然后在树根右旋转；

右-右：在树根左旋转两次；

右-左：在右孩子右旋转，然后在树根左旋转。

```
private:
  void splay(link& h, Item x)
    {
      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); }
```

当我们利用发散插入向一棵 BST 插入一节点时，我们不但把该节点带到树根，而且使得（在搜索路径上）遇到的其他节点距离树根更近。准确地说，我们执行的旋转操作把树根到任何我们遇到的节点的距离削减一半。如果我们实现搜索操作的方式使得它在搜索期间执行发散转换，上述性质还是成立的。树里面的一些路径的确变长了：若我们并不访问那些路径上的节点，这个影响是无关重要的。若我们的确访问位于一条长路径上的节点，在执行以上处理后，它到树根的距离就减少一半；因此，没有任何一条路径能累积产生高额开销。

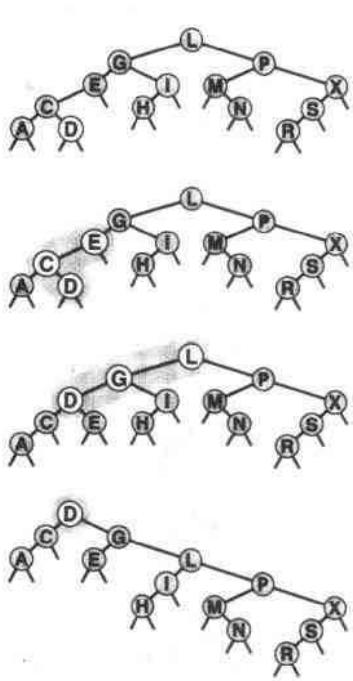


**图 13.7　发散插入**

本图例描述了利用发散根插入方法，对图中顶部的样本树插入关键字为 D 的一条记录的结果（底部图）。在这种情况中，这个插入过程由以下处理组成：一次“左-右”双重旋转和随后的“右-左”双重旋转（从顶部开始）。

**性质 13.5**　*在一棵 $N$ 节点的发散 BST 中执行 $M$ 个插入或搜索操作构成的任意序列，所需的比较次数是 $O((N+M)\lg(N+M))$。*

对这个结果的证明由 Sleator 和 Tarjan 在 1985 年提出，它是算法的分期化分析的一个经典例子（参见“第四部分参考资料”）。在第八部分我们将详细地对它进行分析。■

性质 13.5 是一个分期化的性能保障：我们保障的不是每种操作的高效执行，而是所有执行的操作的平均开销是高效的。这个平均值不是概率上的；相反，我们声明的是可保障总计开销处于低水平。对于许多应用，这种保障已经绰绰有余，但对于其他一些应用它也许还不足够。例如，在使用发散 BST 时我们不能为每一种操作都支持得到保证的响应时间，因为某些操作可能需要线性的时间。如果一种操作的确需要线性的时间，那么我们可以保证其他操作将快得多；但对于必须等待的用户，这也许起不了任何的安慰作用。

性质 13.5 给出的界限是对于所有操作的一种最坏情况下的界限：它也许比实际开销高出许多，这正是最坏情况界限的典型特征。发散操作使最近访问的元素更接近树的顶部，因此，对于具有不一致的访问模式的搜索应用——特别是具有一个相对较小（即使更改速度很慢）的访问数据项处理集的应用，这种方法极具吸引力。

图 13.9 给出的两个范例显示了“发散-旋转”操作在使树平衡的过程中的有效性。在这些图中，通过少量的搜索操作，一棵退化树（通过把数据项依照其关键字次序插入而建立的树）就拥有了相对较好的平衡状态。

如果在树中保存了重复的关键字，那么对于给定节点中的关键字，发散操作可以使含有相等关键字的数据项分散落入该节点的两边（参阅练习 13.38）。这个结论告诉我们：我们不能像对标准二叉搜索树那样轻易地找到带有给定关键字的全部数据项。我们必须

在两棵子树中都检查重复情况，或者像第十二章讨论的那样，利用一些其他方法处理重复关键字。

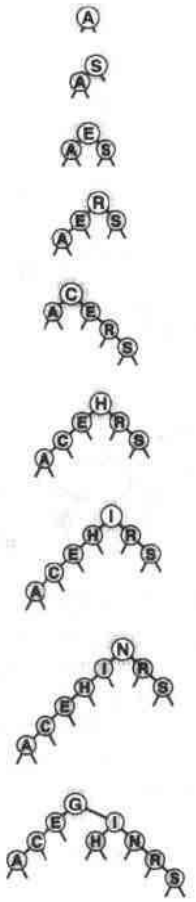


图 13.8 发散 BST 构造过程

本系列图例描述了利用发散插入方法，向一棵原先为空的树插入带有关键字 A S E R C H I N G 的记录的过程。

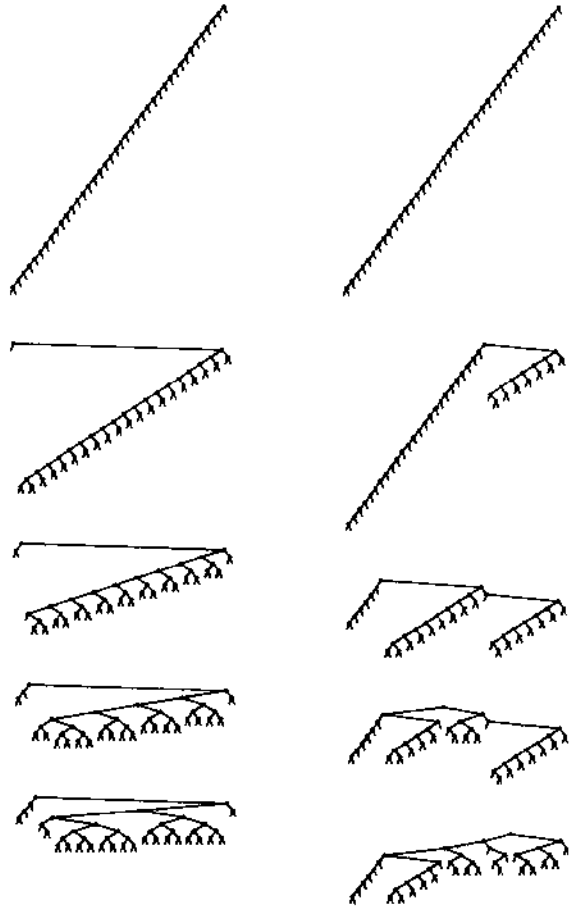


图 13.9 用搜索平衡一棵最坏情况的发散树

利用发散插入，向一棵原先为空的树依照排序后的顺序插入关键字，则每次插入只需要一个常数数量的步骤，但会生成一棵不平衡的树，如左、右两图顶部所示。左边的图例序列显示了（用发散方法）搜索树里面最小、次小、第三小和第四小关键字的结果。每次搜索使通向该搜索关键字（以及树中大多数其他关键字）的路径长度减半。右边的图例序列显示了用一系列随机搜索命中，使同一棵最坏情况下的起始树达到平衡的过程。每次搜索使位于其路径上的节点数量减半，并削减了树中许多其他节点的搜索路径长度。综合来看，少量的搜索操作大大改善了树的平衡状态。

## 练习

▷ 13.25 画出由以下处理所得的发散 BST：利用发散插入方法，向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项。

▷ 13.26　对于一次双重旋转，必须改变多少个树链接？在程序 13.5 的每一次双重旋转中，实际改变了多少个树链接？

13.27　利用发散操作，向程序 13.5 加入一个搜索实现。

○ 13.28　实现程序 13.5 的发散插入函数的一个非递归版本。

13.29　利用你从练习 12.30 得到的驱动程序，通过以下操作确定发散 BST 作为自组织搜索结构的有效性：对练习 12.31 和 12.32 所定义的搜索查询分布，将发散 BST 和标准 BST 进行比较。

○ 13.30　利用发散插入向一棵原先为空的树插入 $N$ 个关键字，其中 $2 \le N \le 7$。画出所有可能得出的结构不同的 BST。

● 13.31　确定以下概率：练习 13.30 的每棵树都是向一棵原先为空的树插入 $N$ 个随机且不相同的关键字的结果。

○ 13.32　对于 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$，通过发散插入把 $N$ 个随机关键字插入到一棵原先为空的树，建立一棵 BST。通过实验研究，分别计算在其中的搜索命中和搜索失败所使用的比较次数的平均值和标准差。你不必进行任何搜索，只要建立树并计算它们的路径长度即可。和随机化 BST 相比，发散 BST 更平衡，还是更不平衡，还是平衡程度相等？

13.33　扩展你对练习 13.32 所得的程序，在每棵构造好的树中执行 N 次带有发散操作的随机比较（它们很可能失败）。发散操作怎样影响对应一次搜索失败的平均比较次数？

13.34　利用你从练习 13.32 和 13.33 所得程序测量运行时间，而不只是对比较操作计数。进行同样的实验。解释你从实验结果所得结论中的任何变化。

13.35　对以下任务比较发散 BST 和标准 BST：从一份至少有 1 百万字符的现实世界文本中建立一索引。测量建立该索引所需时间以及 BST 中平均路径长度。

13.36　通过插入 N 个随机关键字建立一棵发散 BST，其中 $N = 10^3$，$10^4$，$10^5$ 和 $10^6$。通过实验确定在该树中的搜索命中的平均比较次数。

13.37　通过实验研究测试以下的意念：对于随机化 BST，用发散插入代替标准根插入。

▷ 13.38　画出由以下处理所得的发散 BST：向一棵原先为空的树依照关键字顺序 0 0 0 0 0 0 0 0 0 0 0 0 1 插入带有这些关键字的数据项。

## 13.3　自上而下 2-3-4 树

尽管我们可以通过随机化 BST 和发散 BST 提供良好的性能保障，它们两者依然存在以下可能性：一个特定搜索操作需要线性的时间。它们也因此不能帮助我们回答关于平衡树的基本问题：是否存在一种 BST，使我们能够保证每个插入和搜索操作的开销将是该树大小的对数？在本节和 13.4 节，我们考虑一种 BST 抽象归纳，以及这些树作为一种 BST 类型的一个抽象表示，它将使我们能够对这个问题作出肯定的回答。

为了保证我们的 BST 将是平衡的，我们需要在所使用的树结构中存在一定的灵活性。为了获得这种灵活性，让我们假设树里面的节点能容纳一个以上关键字。更明确地说，我们将使用三节点和四节点，它们各自能够容纳两个和三个关键字。一个三节点具有三个从

它出发的链接：一个用于所有其关键字比该节点本身的两个关键字都要小的数据项；一个用于所有其关键字介乎于该节点本身的两个关键字之间的数据项；一个用于所有其关键字比该节点本身的两个关键字都要大的数据项。类似地，四节点具有四个从它出发的链接：每一个都用于由它的三个关键字定义的四个值域区间之一。一棵标准 BST 中的节点因此可被称为二节点：一个关键字，两个链接。稍后，我们将看到一些有效途径，它们定义并实现对这些扩展节点的基本操作；现在，让我们假设我们可以便捷地控制它们，并看看如何可以把它们放到一起以形成树。

**定义 13.1** 2-3-4 搜索树是一棵或者为空、或者由以下三类节点组成的树：二节点，具有一个关键字、一个左链接指向带有较小关键字的一棵树、一个右链接指向带有较大关键字的一棵树；三节点，具有两个关键字、一个左链接指向带有较小关键字的一棵树、一个中链接指向带有介乎于该节点本身的两个关键字值之间的关键字的一棵树、以及一个右链接指向带有较大关键字的一棵树；四节点，具有三个关键字和四个链接，对于该节点的关键字所内含的四个值域区间，这些链接分别指向带有区间定义的关键字值的一棵树。

**定义 13.2** 平衡 2-3-4 搜索树是一棵 2-3-4 搜索树，其中所有指向空树的链接和树根的距离都相同。

在本章中，我们将用术语 2-3-4 树指代平衡搜索 2-3-4 树（在其他环境中它表示一种更广义的结构）。图 13.10 描述了一棵 2-3-4 树的例子。在这样一棵树里对关键字的搜索算法，是对 BST 的搜索算法的概括。为了确定一个关键字是否在树中，我们把它和树根的关键字作对比：若等于其中任何一个，我们就得到一次搜索命中；否则的话，我们顺着链接，从树根进入对应包含该搜索关键字的关键字数值集合的子树，并在该树递归地搜索。对于表示二节点、三节点和四节点以及组织寻找正确链接的机制，存在许多方法；我们把对这些解决方案的讨论推迟到 13.4 节，到时我们将讨论一种特别方便的方案。

为了向一棵 2-3-4 树插入一个新节点，我们可以执行一次失败搜索，然后挂接该节点，正如对 BST 的处理那样，但新树将变得不平衡。2-3-4 树之所以重要，其主要

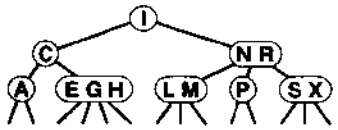



图 13.10 一棵 2-3-4 树

本图例描述了一棵含有关键字 A S R C H I N G E X M P L 的 2-3-4 树。我们可以通过以下方法在这一棵树中找到一关键字：利用树根节点的关键字找到指向一子树的一个链接，然后递归地重复该处理。例如，为了在本树中搜索 P，我们将跟随从树根出发的右链接（因为 P 大于 I），然后跟随从树根的右孩子出发的中链接（因为 P 处于 N 和 R 之间），然后在包含 P 的二节点成功地终止这次搜索。

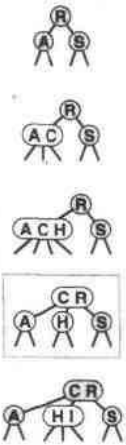



图 13.11 对一棵 2-3-4 树的插入

一棵仅由二节点构成的 2-3-4 树相当于一棵 BST（顶部图）。我们可以通过把对 C 的搜索终止处的二节点转变为一三节点，达到插入 C 的目的（从上往下数第二个图）。类似地，我们可以通过把对 H 的搜索终止处的三节点转变为一四节点，实现对 H 的插入（从上往下数第三个图）。为插入 I 我们必须做更多的工作，因为对它的搜索在一个四节点终止。首先，我们分裂该四节点，把它的中间关键字向上传递给双亲，并把该节点转变成三节点（从上往下数第四个图，突出显示）。这个转换给出了包含以上关键字的一棵有效的 2-3-4 树，它的底部有容纳 I 的空间。最后，我们把 I 插入到当前终止搜索的二节点，并把该节点转变成三节点（底部图）。

原因是：在每种情况中，我们都可以进行插入并仍然能在树里保持完美平衡状态。例如，若搜索终止处的节点是一个二节点，很容易看出我们所作的处理就是把该节点转变成一个三节点。类似地，若搜索在一个三节点终止，我们就把该节点转变成一个四节点。但如果搜索在一个四节点终止，我们应该怎么办呢？答案是我们可以在为新关键字腾出空间的同时保持树里的平衡，具体方法是：首先把四节点分裂成两个二节点，把中间关键字向上传递给该节点的双亲。图 13.11 展示了这三种情况。

现在，如果我们必须分裂一个四节点，而它的双亲也是一个四节点，我们应如何处理呢？一种方法是也把双亲分裂，但是它的祖父母节点还可能是一个四节点，祖父母节点的双亲也可能是，以此类推——我们可以在树里一直向上不断分裂节点。一种更容易的方法是通过在树里向下搜索的过程中分裂任何遇到的四节点，从而保证搜索路径不会在一个四节点终结。

更具体地说，正如图 13.12 显示的那样，每次我们遇到和一四节点相连接的一个二节点，我们把这个节点对转换成和两个二节点相连接的一个三节点；而每次我们遇到和一四节点相连接的一个三节点，我们把这个节点对转换成和两个二节点相连接的一个四节点。不仅因为关键字可被到处移动的方式，还因为链接可被到处移动的方式，对四节点的分裂是可能的。两个二节点拥有和一个四节点相同数量（4 个）的链接，所以我们可以执行分裂，而不必在分裂节点以下（或以上）产生任何连锁改变。一个三节点并非仅仅通过加入另一个关键字就转变成四节点，它还需要另一个指针（在这种情况下，由分裂提供这额外的指针）。关键一点是：这些转换纯粹是局部的——除了图 13.12 所示部分，再没有树的其他部分需要检查或修改。这些转换的每一个都把四节点的关键字之一向上传递到该节点在树中的双亲，并相应地重新组织链接。

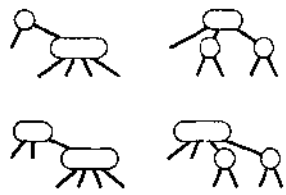


**图 13.12　在一棵 2-3-4 树中分裂四节点**

在一棵 2-3-4 树中，我们可把任何其双亲不也是四节点的四节点分裂成两个二节点，并把它的中间记录向上传递给它的双亲。连接到一四节点的一个二节点（顶部左图）变成连接到两个二节点的一个三节点（顶部右图），而连接到一四节点的一个三节点（底部左图）变成连接到两个二节点的一个四节点（底部右图）。

当我们在树中向下处理时，我们不必担心当前节点的双亲是一个四节点，因为我们的转换处理已经保证了：当我们通过树里的每个节点，向下继续处理的出发点决不是一个四节点。特别是，当我们到达树的底部时，我们并非位于一个四节点上，我们可以直接插入新节点：把一个二节点转换成三节点，或把一个三节点转换成四节点。我们可将该插入操作看作对树底部的一个假想四节点的分裂，它向上传递这个新关键字。

最后一个细节问题是：只要树根变成一个四节点，我们就把它分裂成一个由三个二节点组成的三角形，正如我们在前一个例子中对第一个被分裂节点所做的那样。在一次插入后马上分裂树根的做法和留待下一次插入才进行分裂的选择相比，稍稍方便一点，因为我们永远不必为树根的双亲节点担忧。分裂树根（也只有这种操作）使得树再增高一层。

图 13.13 描述了对一个样本关键字集合构造 2-3-4 树的过程。和从上向下生长的标准 BST 不同，这些树从下向上生长。因为四节点在自上而下的过程中分裂，这些树就被称为自上而下 2-3-4 树。这个算法很重要，因为它生成几乎完美地平衡的搜索树，然而在遍历树的过程中它只需作几个局部转换。

**性质 13.6** 在 $N$ 个节点的 2-3-4 树中的搜索最多访问 $\lg N+1$ 个节点。

每个外部节点到树根的距离都相等：我们执行的转换操作对于任何节点到树根的距离都没有影响，除了当我们分裂树根时（在这种情况下所有节点到树根的距离增加 1）。若所有节点都是二节点，上面陈述的结果成立，因为这棵树就类似于一棵完全二叉树；若存在三节点和四节点，这个高度只可能更低。■

**性质 13.7** 在 $N$ 个节点的 2-3-4 树中的插入，在最坏情况下需要对少于 $\lg N+1$ 个节点进行分裂；而在平均情况下，所需的节点分裂次数可能少于 1。

可能发生的最坏情况是：在通向插入点的路径上的所有节点都是四节点，它们全部都将被分裂。但在一棵构造于 $N$ 个元素的随机排列的树中，不但最坏情况不太可能出现，而且在平均情况下所需的节点分裂次数也极少，这是因为树里不存在许多四节点。例如，在图 13.14 描述的大型树中，所有四节点中只有两个不处于最底层。专家们至今还没有得出对 2-3-4 树在平均情况下的性能的精确分析结果，但实验研究清楚表明：非常少的分裂操作用于使树保持平衡。最坏情况也只是 $\lg N$——在实际情况中不会达到接近这个数值。■

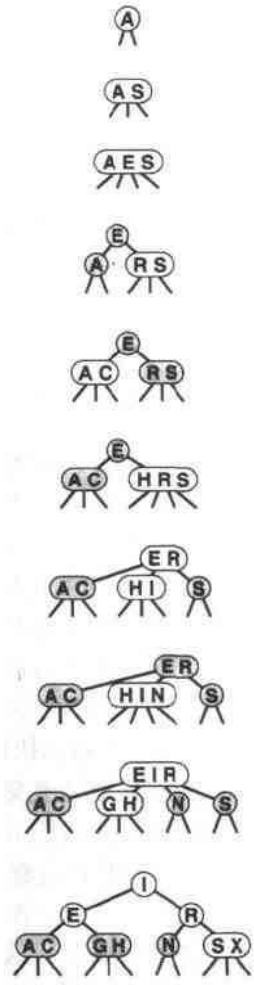



**图 13.13 2-3-4 搜索树的构造过程**

本系列图例描述了向一棵原先为空的 2-3-4 树插入带有关键字 A S E R C H I N G X 的数据项的结果。我们分裂在搜索路径上遇到的每个四节点，因此确保在树底部为新数据项留有空间。

对于利用 2-3-4 树的搜索操作，先前的描述已经足以定义一个能在最坏情况下保证取得良好性能的算法了。然而，这仅仅达到了给出一个实现的目标的一半。尽管我们可能编写出这样的算法：它们在实际上对分别代表二、三和四节点的不同数据类型执行转换操作，但其中所涉及到的大多数任务都难以在这个直接表示中加以实现。正如发散 BST 那样，为了控制更复杂的节点结构，由此产生的管理开销可导致这些算法比标准 BST 搜索还要慢。平衡的主要目的是针对一个最坏情况提供保障，但我们当然希望对于这保障的管理开销越低越好，而且我们也希望能避免在这个算法的每一次运行过程都要付出这些开销。幸运的是，正如我们在 13.4 节将要看到的那样，存在一种相对简单的关于二、三和四节点的表示

方法，它允许用一种统一的途径进行转换，而且除了标准二叉树产生的开销外，它几乎不需要管理开销。

**图 13.14　一棵大型 2-3-4 树**

这棵 2-3-4 树是向一棵原先为空的树作 200 次随机插入的结果。树里面所有搜索路径都只有 6 个或更少的节点。

我们业已描述的算法只是保持 2-3-4 搜索树的平衡的其中一种可能途径。到目前为止已开发出几种其他方法，它们都可以达到同一目的。

譬如，我们可自下而上进行平衡。首先，我们在树中进行一次搜索，以找到被插入数据项所属的底部节点。若那个节点是二节点或三节点，我们就像先前那样，使它增长为三节点或四节点。若它是四节点，我们就像先前那样把它分裂（把新数据项插入到在树底部得出的二节点之一），并把中间数据项插入到其双亲节点——如果其双亲节点是二节点或三节点的话。若双亲节点也是四节点，我们把它分裂（把底部的中间节点插入到适当的二节点中），并把中间数据项插入到其双亲节点——如果其双亲节点是二节点或三节点的话。若祖父母节点还是四节点的话，我们以同样方式继续在树里向上处理——不断分裂四节点，直到我们在搜索路径上遇到一个二节点或三节点为止。

在仅有二节点或三节点（没有四节点）的树中，我们可以进行这种自下而上的平衡操作。这种方法在算法执行期间会导致更多的节点分裂，但它更易于编写程序代码，因为需要考虑的情况较少。在另一种方法中，我们寻求通过以下处理减少节点分裂的数目：当我们准备好分裂一个四节点时，就寻找它的不是四节点的兄弟节点。

所有这些方法的实现都涉及到同一个基本递归模式，我们将在 13.4 节看到这一点。我们还将在第十六章讨论它的概括实现。相对于其他方法，我们考虑的自上而下插入方法的主要优点是：它能够在树里面的一遍自上而下处理中取得必要的平衡效果。

### 练习

▷ 13.39　画出由以下处理所得的平衡 2-3-4 搜索树：利用自上而下插入方法，向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项。

▷ 13.40　画出由以下处理所得的平衡 2-3-4 搜索树：利用自下而上插入方法，向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项。

○ 13.41　对于含 N 个节点的平衡 2-3-4 搜索树，可能的最小和最大高度是什么？

○ 13.42　对于含 N 个关键字的平衡 2-3-4 BST，可能的最小和最大高度是什么？

○ 13.43　$2 \le N \le 12$，画出所有结构不同的含 N 个关键字的平衡 2-3-4 BST。

● 13.44　确定以下概率：练习 13.43 的每棵树都是向一棵原先为空的树插入 $N$ 个随机且不相同的关键字

的结果。

13.45 制作一张表格，显示对于练习 13.43 的每个 $N$ 值的同构树数量。同构是指这些树可通过交换节点中的子树，实现互相转换。

▷ 13.46 描述出对应平衡 2-3-4-5-6 树中搜索和插入操作的算法。

▷ 13.47 画出由以下处理所得的不平衡 2-3-4 搜索树：向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项，其中利用了下列方法。若搜索终止于一个二节点或三节点，就像平衡算法那样把它转变成二节点或四节点；若搜索终止于一个四节点，用一个新二节点取代该四节点中的适当链接。

## 13.4 红黑树

前一节描述的自上而下 2-3-4 插入算法很容易理解，但由于所有可能出现的不同情况，对它的直接实现十分麻烦。我们需要维护三种不同类型的节点、比较搜索关键字和节点中的每个关键字、把链接以及其他信息从一种类型的节点复制到另外一种类型的节点中、创建及消除节点，等等。在本节中，我们分析 2-3-4 树的一种简单抽象表示方法，它能把我们引向符号表算法的一个自然实现，该实现在最坏情况下具有接近最佳的性能保障。

其基本意念是把 2-3-4 树表示为标准 BST（仅有二节点），但对每个节点加入额外的一位信息，以此对三节点和四节点编码。我们把这些链接看作为两种不同类型：把由三节点和四节点构成的小型二叉树绑合起来的红链接，以及绑合 2-3-4 树的黑链接。具体地说，如图 13.15 所示，我们把四节点表示为由红链接连接起来的三个二节点，把三节点表示为由单一红链接连接起来的两个二节点。三节点中的红链接可以是一个左链接或一个右链接，因此存在两种表示每个三节点的方法。

在任何树中，每个节点都被一个链接所指向，所以对链接染色相当于对节点染色。相应地，我们对每个节点利用额外的一位存储指向该节点的链接颜色。我们把这种方式表示的 2-3-4 树称为红黑 BST（red-black BST）。每个三节点的方向由我们即将描述的算法的动态性决定。实施一条规则、使得三节点都向同一方向倾斜是可以做到的，但没有这样做的理由。图 13.16 显示了一个红黑树的范例。如果我们删除红链接，并把它们连接的节点压缩到一起，结果就得到图

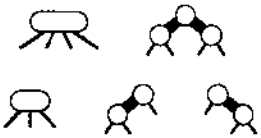


图 13.15 红黑树中的三节点和四节点

对于表示 2-3-4 树中的二节点和四节点，运用两种链接类型向我们提供了一个有效的途径。我们把红链接（我们的图例中的粗线）用于节点间的内部连接，把黑链接（我们的图例中的细线）用于 2-3-4 树链接。一个四节点（顶部左图）通过一棵平衡子树表示，它由被红链接相连的三个二节点组成（顶部右图）。两者都具有三个关键字和四个黑链接。一个三节点（底部左图）通过由单一红链接相连的两个二节点表示（左边或右边）。它们都具有两个关键字和三个黑链接。

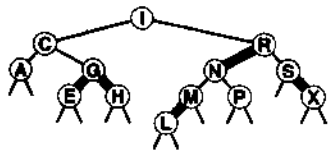



图 13.16 一棵红黑树

本图例描述了一棵包含关键字 A S E R C H I N G E X M P L 的红黑树。我们可用标准 BST 搜索在这样一棵树中寻找一个关键字。在这棵树中，任何从树根到一外部节点的路径都有三个黑链接。如果我们把这棵树中通过红链接相连的节点压缩到一起，即可得到图 13.10 的 2-3-4 树。

13.10 的 2-3-4 树。

红黑树有两个核心性质：（1）对 BST 的标准搜索方法无须修改即可运作；（2）它们直接对应 2-3-4 树。因此我们可以通过保持这种对应关系，实现平衡 2-3-4 树算法。我们同时得到了两种结构的最佳点：来自标准 BST 的简单搜索方法，以及来自 2-3-4 搜索树的简单的插入-平衡方法。

搜索方法永远不检查代表节点颜色的域，所以平衡机制不会令基本搜索处理所需的时间增加额外开销。因为在典型应用中每个关键字只插入一次，却可能被搜索许多次，最终结果是我们以相对较低的开销（因为在搜索期间不进行任何平衡工作）获取了得到改善的搜索次数（因为树得到了平衡）。而且，用于插入的开销较少：仅当我们遇到四节点，才需要执行平衡操作；而由于我们总是对它们进行分解，在树中存在的四节点不多。插入处理程序的内层循环就是在树中向下推进的代码（和标准 BST 中的搜索操作或“搜索-插入”操作一样），其中加入了一个额外的测试：如果一节点有两个红孩子，它就是一个四节点的一部分。这个低开销正是红黑树的高效率的主要原因。

现在，让我们来考虑关于两个转换的红黑表示——当我们的确遇到一个四节点时，也许需要执行这两个转换：如果我们有连接到一个四节点的一个二节点，那么我们应该把这对节点转换为连接到两个二节点的一个三节点；如果我们有连接到一个四节点的一个三节点，那么我们应该把这对节点转换为连接到两个二节点的一个四节点。当一个新节点插入到树的底部时，我们把它想像为一个必须分裂的四节点，而它的中间节点被向上传递、插入到搜索终止的底部节点（自上而下的过程可保证它是一个二节点或三节点）中。当我们遇到连接到一个四节点的一个二节点，所需的转换很简单；如果我们遇到其“右”边连接到一个四节点的一个三节点，同样的转换处理也可奏效，正如图 13.17 的前两种情况所显示的那样。

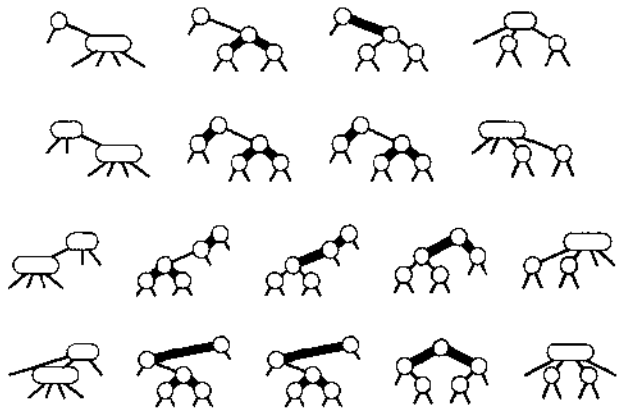


**图 13.17　在一棵红黑树中分裂四节点**

对一棵红黑树中其双亲不是四节点的一个四节点，我们通过以下处理实现分裂它的操作：改变构成四节点的三个节点中的节点颜色，然后可能进行一次或两次旋转操作。如果双亲节点是一个二节点（顶部图），或是具有一个便利方向的三节点（从上往下数第二个图），就不需要旋转操作。如果该四节点位于三节点的中央链接上，需要一次双重旋转；否则，一次单一旋转就足够了（从上往下数第三个图）。

图 13.17 的后两种情况显示了：如果我们遇到连接到一个四节点的一个三节点，其余两种可能出现的情况。（实际上存在四种情况，因为这两者的镜像还可以对另一方向的三节点发生）。在这些场合中，基本的四节点分裂将留下两个连续的红链接——所产生的树并不能表示出一棵和我们的约定相一致的 2-3-4 树。这种情况不算太差，因为我们拥有三个有红链接相连的节点：我们需要做的全部工作就是转换这棵树，使得红链接从同一节点向下指。

幸运的是，我们一直都在使用的旋转操作正是取得期望的效果所需要的工具。让我们从两种剩余情况中比较简单的一种开始讨论：图 13.17 的第三种情况，其中连接到一三节点的一个四节点已经分裂，留下两个连续的、指向同一方向的红链接。如果三节点指向另一个方向，那么这种情况就不会发生：相应地，我们可重新组织树的结构，以此转换三节点的方向，并因此把这种情况简化到和第二种情况一样，这时直接的四节点分裂就足以完成所需处理了。重新组织树结构、把三节点重定向是一次单一旋转操作，其附加条件是两个节点的颜色必须进行转换。

**程序 13.6 红黑 BST 中的插入**

本函数利用红黑表示法实现 2-3-4 树中的插入操作。我们对类型 node 加入一个颜色位 red（并相应地扩展它的改造函数），1 表明节点为红色，而 0 表明节点为黑色。在树中向下推进时（在递归调用以前），我们搜索四节点，并通过切换全部三个节点中的颜色位实现对四节点的分裂。当我们到达树的底部时，为将被插入的数据项创建一个新的红节点，并返回一个指向它的指针。在树中向上返回的过程中（在递归调用以后）我们检查是否需要执行一次旋转操作。如果搜索路径上有两个方向相同的红链接，我们从链接上方的节点执行一次单一旋转，然后切换颜色位，以生成一个正确的四节点。如果搜索路径上有两个方向不同的红链接，我们从链接下方的节点执行一次单一旋转，以此简化成另一种情况，留待向上的下一步再处理。

```
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; }
```

最后，如果连接到一三节点的一个四节点已经分裂，并留下两个连续的、指向不同方向的红链接，我们对这种情况作如下处理：执行旋转操作，马上把它简化为链接方向相同的情况，然后就可以像先前那样进行处理。这个转换相当于我们在 13.2 节中对发散 BST 所使用的“左-右”和“右-左”双重旋转操作，但是我们还必须多费一点工夫，保持节点颜色正确。图 13.18 和 13.19 描述了红黑插入操作的例子。

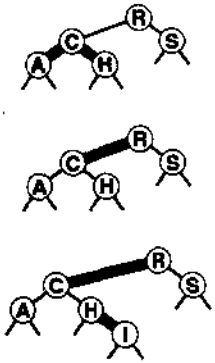



**图 13.18　对一棵红黑树中的插入**

本图例描述了向顶部的红黑树样本插入带有关键字 I 的一个数据项所得的结果（底部图）。在这种情况下，插入过程由以下处理组成：用一次颜色切换分裂 C 处的四节点（中间图），然后把新节点加入树底部，把包含 H 的节点从一个二节点转化为一个三节点。

程序 13.6 是对红黑树的插入操作的一个实现，它执行图 13.17 总结出的转换处理。该递归实现使我们可能执行以下处理：在树中向下推进的过程中，执行四节点的颜色切换（在递归调用以前），然后在树中向上返回的过程中，执行旋转（在递归调用以后）。如果没有我们已开发出的实现这个程序的两层抽象，它会变得难以理解。我们可检查到：这个巧妙的递归方法实现了图 13.17 描述的旋转操作；然后，我们可检查到：该程序实现了我们关于 2-3-4 树的算法——在树中向下推进时分裂四节点，然后把新数据项插入到搜索路径在树底部终止处的二节点或三节点。

图 13.20（我们可以把它看作为图 13.13 的一个更详细的版本）显示了程序 13.6 如何构造出红黑树，该树表示了插入一个关键字集合范例时所得的平衡 2-3-4 树。图 13.21 显示了一棵树，它建造于我们一直使用的较大的例子；在这棵树对一个随机关键字的搜索中，访问节点的平均个数是 5.81，相比之下在十二章中用同样关键字构造的树是 7.00，而一棵完美平衡树可能的最佳性能是 5.74。对于同样的关键字，我们仅仅付出了一些旋转操作的代价，就得到一棵平衡程度远远高于我们在本章所见的其他任何树种类的树。对于利用一个二叉树结构的插入操作，程序 13.6 是一个有效的、相对紧凑的算法，它保证对所有搜索和插入操作都仅需对数水平的步骤数。它是为数不多的具有这个性质的符号表实现方法之一；在一个库实现

中，如果不能准确界定被处理的关键字序列的性质，则它的用处更可得到肯定。

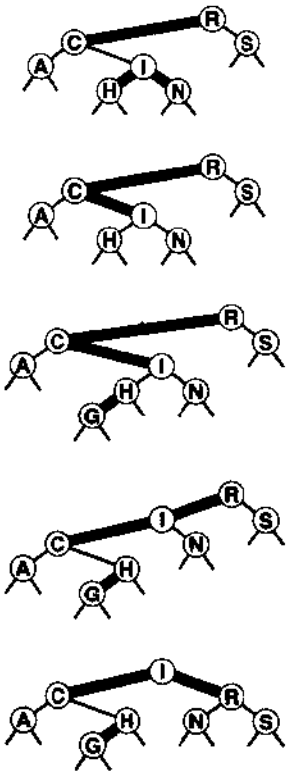


**图 13.19 对一棵红黑树中的插入（带旋转操作）**

本图例描述了向顶部的红黑树样本插入带有关键字 G 的一个数据项所得的结果（底部图）。在这种情况下，插入过程由以下处理组成：用一次颜色切换分裂 I 处的四节点（从上往下数第二个图），然后把新节点加入树底部（从上往下数第三个图），然后（在递归函数调用后，返回到代码中搜索路径的每个节点上）在 C 点执行一次左旋转、在 R 点执行一次右旋转，以完成分裂四节点的过程。

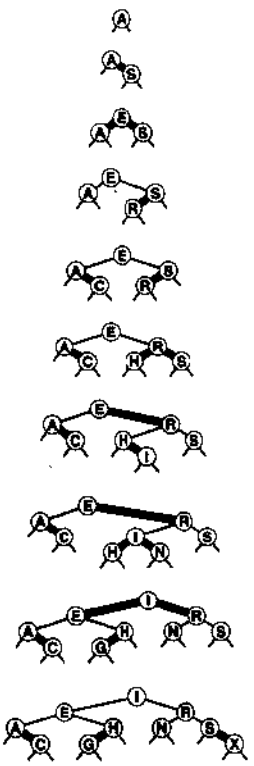


**图 13.20 一棵红黑树的构造过程**

本系列图例描述了向一棵原先为空的红黑树，插入带有关键字 A S E R C H I N G X 的记录所得结果。

**性质 13.8** 在 $N$ 个节点的红黑树中的一次搜索需要少于 $2\lg N+2$ 次比较操作。

只有对应于一棵 2-3-4 树中连接到一个四节点的一个三节点的分裂，才需要在红黑树执行一次旋转，所以本性质由性质 13.2 得来。最坏的情况是：通向插入点的路径由交替的三节点和四节点组成。■

而且，程序 13.6 导致的平衡操作的开销非常小，而它生成的树已接近最佳，因此把它作为一种快速的通用搜索方法来考虑，也是极具吸引力的。

**性质 13.9** 平均情况下，在一棵由随机关键字建造的 $N$ 个节点的红黑树中，一次搜索需要大约 $1.002\lg N$ 次比较。

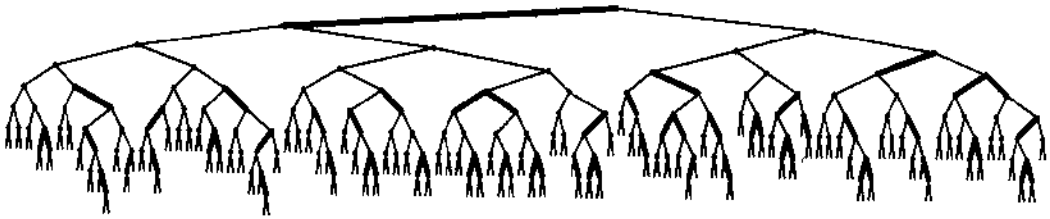


图 13.21　一棵大型红黑树

这棵红黑树是向一棵原先为空的树插入 200 个随机排序关键字的所得结果。这棵树中全部的搜索失败都使用了 6~12 次比较操作。

常数 1.002 已经通过局部分析和模拟得到了确认（参见“第四部分参考资料”）。它已足够地低，以至于我们可以认为红黑树在实际应用方面是最佳的，但以下问题仍没有得到解答：红黑树是否真正地收敛于最佳结果。这个常数等于 1 吗？■

因为程序 13.6 的递归实现在递归调用前进行一些处理，在递归调用后也要进行一些处理，所以它在顺着搜索路径向下处理时对树进行一些修改，在向上返回处理时也对树进行一些修改。因此，它不具有在一遍自上而下处理中完成平衡操作的性质。这个事实对大多数应用都几乎没有影响，因为算法可保证递归深度并不大。对于一些涉及到多个访问同一棵树的独立过程的应用，我们也许需要一个非递归的实现，它在任意给定时刻都积极地对常数个节点执行操作（参阅练习 13.66）。

对于一个在树中携带了其他信息的应用，旋转操作的开销也许相当昂贵，它可能使得我们要更新旋转操作中涉及到的子树中的所有节点内含信息。对于这样一个应用，我们可以用以下方法保证每个插入操作最多只涉及一次旋转：用红黑树实现在 13.3 节末尾描述的自下而上 2-3-4 搜索树。在这些树中，一次插入涉及到沿着搜索路径对四节点进行的分裂，这个分裂在红黑树表示中只需要做颜色变换，而无须进行旋转。随后的操作是：如果沿着搜索路径向上返回时，遇到第一个二节点或一个三节点（参阅练习 13.59），则执行一次单一旋转或双重旋转（图 13.17 的其中一种情况）。

如果要在树中保留重复的关键字，那么我们必须允许其关键字等于给定节点的数据项落入该节点的两边，正如我们对发散 BST 所做的处理那样。否则的话，长串的重复关键字可以产生严重的失衡。这个结论再次告诉我们：找出所有含给定关键字的数据项需要专门的代码。

13.3 节的末尾已提到过，在业已提出的用于实现平衡二叉树的几种类似策略中，2-3-4 树的红黑表示法是其中一种（参见“第四部分参考资料”）。正如我们所看到的，是旋转操作使树变得平衡：我们已经从这些树的一种特定视角进行考虑，它让我们容易决定什么时候进行旋转。从这些树的其他视角出发则产生其他算法，在这里我们将简要地讨论其中的几个。

用于平衡树的最古老、最著名的数据结构是高度平衡树（height-balanced tree，或称为 AVL tree），它在 1962 年由 Adel'sonVel'skii 和 Landis 发现。这些树拥有一个性质：每个节

点的子树高度最多相差 1。如果一次插入导致某节点的其中一棵子树在高度上增加 1，那么平衡状态就可能被打破。然而，在每种情况中，一次单一旋转或双重旋转即可使节点恢复平衡状态。基于这个结论的算法与自下而上平衡 2-3-4 树的方法相近：先对该节点执行一次递归搜索，然后在递归调用以后检查不平衡的情况，有必要的话执行一次单一旋转或双重旋转来修正问题（参阅练习 13.61）。我们必须知道以下情况，才能做出执行哪种旋转（如果需要旋转的话）的决定：每个节点和它的兄弟节点的高度相比，其高度小 1、相等还是大 1。在一种直接的途径中，每个节点需要两位来记录这个信息；不过若利用红黑抽象的话，也有可能无需任何额外存储空间即可达到目的（参阅练习 13.62 和 13.65）。

因为四节点在自下而上 2-3-4 算法中并不扮演特殊的角色，所以完全可能只使用二节点和三节点、以实质上相同的途径建立平衡树。以这种方法建立的树称为 2-3 树，它在 1970 年由 Hopcroft 提出。在 2-3 树中不存在足够的灵活性，以实现一种方便的自上而下插入算法。再一次，红黑架构能够简化该实现方法；但自下而上 2-3 树和自下而上 2-3-4 树相比并没有特别的优点，因为仍旧需要单一旋转和双重旋转保持平衡状态。自下而上 2-3-4 树的平衡性稍稍好一点，并且拥有每次插入最多使用一次旋转操作的优点。

在第十六章中，我们将研究平衡树的另一种重要类型：这是 2-3-4 树的一个扩展，称为 B 树（B-tree）。对于较大的 M 值，B 树允许在每个节点存在最多 $M$ 个关键字，也因此广泛应用于涉及到超大型文件的搜索应用。

我们已经通过和 2-3-4 树的对应关系定义了红黑树。而进行直接的结构定义也是相当有趣的。

**定义 13.3** *红黑树是一棵二叉搜索树，其中每个节点都被标记为红或黑，附加的限制条件是：在任何从一个外部链接通向树根的路径上，不存在连续出现的两个红节点。*

**定义 13.4** *平衡红黑树是一棵红黑 BST，其中所有从外部链接通向树根的路径都拥有相同数量的黑节点。*

现在，开发一个平衡树算法的另一种可供选择的方法是完全忽略 2-3-4 树抽象，并制定一个插入算法，该算法通过旋转操作保存平衡红黑 BST 的定义性质。例如，使用自下而上算法对应于：用一个红链接在搜索路径底部挂接新节点，然后顺着搜索路径向上处理，在处理过程中进行旋转或颜色变换（正如图 13.17 所示的每种情况那样），以此分解在路径上遇到的任何连续的红链接对。我们执行的基本操作和程序 13.6 及其自下而上对应版本都一样，但也存在细微的差别。这是因为三节点可指向两个方向之一，具体操作也可按不同顺序执行，也可以成功地使用各种各样不同的旋转决策。

下面我们进行总结：利用红黑树实现平衡 2-3-4 树，我们可以开发出一个符号表；其中若一个搜索操作在一文件（比方说共有 1 百万数据项）中搜索一个关键字，则它只需要把该关键字和大约 20 个其他关键字作比较，即可完成任务。在最坏情况下，使用的比较次数也不超过 40。此外，和每次比较操作相关的开销极少，因此即使在一个巨大文件中也可确保进行快速搜索。

## 练习

▷ 13.48　画出由以下处理所得的红黑 BST：利用自上而下插入方法，向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项。

▷ 13.49　画出由以下处理所得的红黑 BST：利用自下而上插入方法，向一棵原先为空的树依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项。

○ 13.50　画出由以下处理所得的红黑树：向一棵原先为空的树依照次序插入从 A 到 K 的字母。然后大体上描述一下：当通过按递增次序插入关键字以建立树时，发生了什么情况。

13.51　给出一系列插入操作，它们将构造出图 13.16 所示的红黑树。

13.52　生成两棵随机的 32 节点的红黑树。（通过人手或某个程序）把它们绘制出来。把它们和建立于相同关键字的（不平衡）BST 作比较。

13.53　若一棵 2-3-4 树有 $t$ 个三节点，有多少棵不同的红黑树与之对应？

○ 13.54　$2 \le N \le 12$，画出所有结构不同的含 $N$ 个关键字的红黑搜索树。

● 13.55　确定以下概率：练习 13.43 的每棵树都是向一棵原先为空的树插入 $N$ 个随机且不相同的关键字的结果。

13.56　制作一张表格，显示对于练习 13.54 的每个 $N$ 值的同构树数量。同构是指这些树可通过交换节点中的子树，实现互相转换。

●●13.57　说明以下结论：在最坏情况下，一棵 $N$ 个节点的红黑树中，几乎所有从树根到一个外部节点的路径都具有长度 $2 \lg N$。

13.58　在最坏情况下，对一棵 $N$ 个节点的红黑树的一次插入需要多少次旋转操作？

○ 13.59　利用红黑表示以及和程序 13.6 一样的递归方法，为以自下而上平衡 2-3-4 树作为基本数据结构的符号表实现构造、搜索以及插入操作。提示：你的代码可以类似于程序 13.6，但应该以一种不同的顺序执行操作。

13.60　利用红黑表示以及和程序 13.6 一样的递归方法，为以自下而上平衡 2-3 树作为基本数据结构的符号表实现构造、搜索以及插入操作。

13.61　利用和程序 13.6 一样的递归方法，为以高度平衡（AVL）树作为基本数据结构的符号表实现构造、搜索以及插入操作。

● 13.62　把你从练习 13.61 所得实现修改为利用红黑树（每节点 1 位）来对平衡信息编码。

● 13.63　利用一个红黑树表示（其中三节点总是向右倾斜），实现平衡 2-3-4 树。注意：这个改动允许你从插入的内层循环中删除其中一项位测试。

● 13.64　程序 13.6 执行旋转操作以保持四节点处于平衡状态。利用一个红黑树表示为平衡 2-3-4 树开发一个实现方案，其中四节点可以被表示成任何三个节点，它们由两个红链接相连（无论是否完美平衡）。

○ 13.65　不对颜色位使用任何额外存储空间，为红黑树实现*构造*、*搜索*以及*插入*操作，其根据是以下的技巧。要把一节点变成红色的话，就交换它的两个链接。然后，若要测试一节点是否为红色的话，就测试它的左孩子是否大于它的右孩子。你必须修改比较操作，以适应可能的指针交换；这个技巧用以取代位

比较操作的是关键字比较，这很可能需要更大的开销，但它也说明了若有必要，可以消除节点中的颜色位。

● 13.66 实现一个非递归的红黑 BST 插入函数（参阅程序 13.6），它对应于使用一次自上而下处理过程的平衡 2-3-4 树插入。提示：保存链接 gg、g 和 p，它们分别指向当前节点的曾祖父母、祖父母和父母节点。双重旋转可能需要所有这些链接。

13.67 编写一个程序，计算一棵给定红黑 BST 中黑节点所占百分比。通过向一棵原先为空的树插入 $N$ 个随机关键字来测试你的程序，$N = 10^3$，$10^4$，$10^5$ 和 $10^6$。

13.68 编写一个程序，计算一棵给定 2-3-4 搜索树中，位于三节点和四节点的数据项所占百分比。通过向一棵原先为空的树插入 $N$ 个随机关键字来测试你的程序，$N = 10^3$，$10^4$，$10^5$ 和 $10^6$。

13.69 对每个节点用 1 位表示颜色，我们可以表示二、三和四节点。那么为了表示一棵二叉树的五、六、七和八节点，我们将需要对每个节点用多少位？

13.70 对于 $N = 10^3$、$10^4$、$10^5$ 和 $10^6$，把 $N$ 个随机关键字插入到一棵原先为空的树，建立一棵红黑树。通过实验研究，分别计算其中的搜索命中和搜索失败所使用的比较次数的平均值和标准差。

13.71 利用你为练习 13.70 实现的程序，计算用于建立这些树的旋转和节点分裂次数。并讨论这些结果。

13.72 利用你从练习 12.30 所得的驱动程序，对于练习 12.31、12.32 定义的搜索询问分布，把发散 BST 的自组织搜索方面分别和红黑树、以及标准 BST 在最坏情况的性能保障作比较（参阅练习 13.29）。

● 13.73 为红黑树实现一个搜索函数，它在树中向下推进的过程中执行旋转并更改节点颜色，以确保位于搜索路径底部的节点不是一个二节点。

● 13.74 利用你对练习 13.73 的解决方案为红黑树实现一个删除函数。它找到要被删除的节点，再继续搜索，直至该路径底部的一个三节点或四节点，并从底部移动其后续节点以取代被删除的节点。

## 13.5 跳跃表

在本节中，我们考虑一种用于开发符号表操作的快速实现的方法，起初它似乎和我们一直以来考虑过的基于树的方法完全不同，但实际上它和它们的关系十分密切。它以一种随机化数据结构为基础，而且几乎肯定能够为我们一直以来考虑过的、对应符号表 ADT 的所有基本操作提供接近最佳的性能。这个基本的数据结构由 Pugh 在 1990 年研究得出（参见“第四部分参考资料”），称为跳跃表（skip list）。它利用一个链表中节点的额外链接，在搜索过程中每次跳过链表的一大部分。

图 13.22 给出了一个简单例子，其中一个有序链表的每隔第三个节点就包含一额外链接，该链接允许我们跳过链表的三个节点。我们可以利用这个额外链接来加快搜索的速度：我们在顶部链表扫描，直到我们找到该关键字，或遇到一个含有较小关键字的节点且它含有一个链接指向含有较大关键字的节点为止，然后就利用底部的链接检查介于其间的两个节点。这种方法使搜索速度提高 3 倍，因为在该链表对第 $k$ 个节点的一次成功搜索中，我们只检查大约 $k/3$ 个节点。

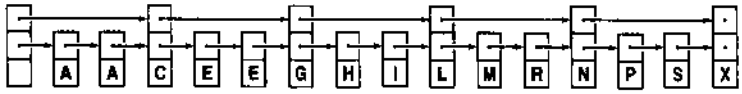



**图 13.22　一个两层链表**

这个链表中每隔第一个节点就有第二链接，因此我们能够以几乎三倍于我们顺着第一链接前进的速度，在表中跳跃扫描。例如，我们从表的开始追踪仅仅 5 个链接，即可到达第十二个节点 P：顺着第二链接依次到达 C、G、L 和 N，然后由 N 的第一链接到达 P。

我们可重复使用这个结构，提供第二个额外链接，以便能够在带有额外链接的节点中更快速地扫描，等等推。而且，我们可以使该结构一般化——每个链接跳过不定数量的节点。

**定义 13.5**　跳跃表是一个有序链表，其中每个节点包含不定数量的链接，节点中的第 i 个链接构成的单向链表跳过含有少于 i 个链接的节点。

图 13.23 描述了一个跳跃表范例，并显示了一个搜索并插入新节点的例子。为了进行搜索，我们在顶层链表扫描，直到我们找到该搜索关键字或一个含有较小关键字的节点、且它含有一个链接指向含有较大关键字的节点为止；然后，我们向下移到第二层链表并重复这个过程，如此继续下去，直到找到搜索关键字或在底层发生一次搜索失败为止。为了进行插入，我们先搜索，如果新节点有至少 k 个额外链接，则在从第 k 层移动至第 k-1 层时把它链接到表中。

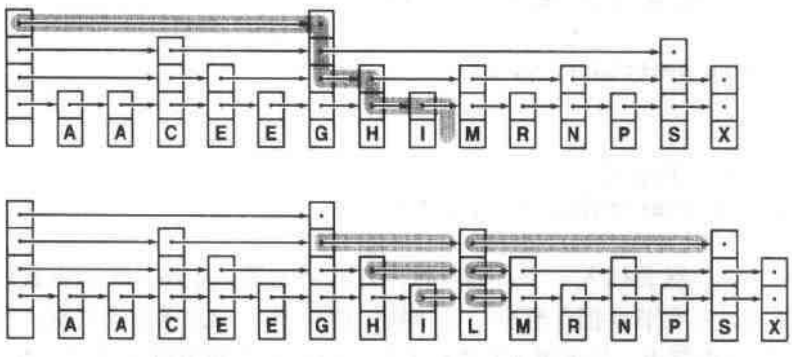



**图 13.23　跳跃表中的搜索和插入**

通过向图 13.22 的结构加入更多层次，并容许链接跳过不定数量的节点，我们就得到一个一般跳跃表的范例。为了在表中搜索一个关键字，我们从最高层开始，每当遇到一个不小于搜索关键字的关键字就向下移动。这里（上图），我们搜索 L 的过程是：从第 3 层开始，移动通过第一个链接，然后在 G 向下移（把空链接视为指向标志值的一个链接）；然后通过到达 1，再向下移到第 2 层，因为 S 大于 L；然后向下移到第 1 层，因为 M 大于 L。要插入带有三个链接的节点 L，我们把它链接到三重链表中，其位置恰恰就是搜索过程中我们找到指向更大关键字的链接处

这些节点的内部表示直接易懂。我们用一个链接数组取代一个单向链表中的单链接，并设立一个整数来记录该节点的链接数目。内存管理也许是跳跃表最复杂的一面——我们

不久后（到时我们将考虑插入操作）将分析类型声明和分配新节点的代码。暂时来说，注意到以下一点就足够了：我们可以通过访问 t->next[k]，来访问跳跃表中第 $k$+1 层二跟随节点 t 的节点。程序 13.7 的递归实现说明了跳跃表中的搜索不仅是单向链表中搜索的一个直接推广，而且和二叉搜索或者 BST 里的搜索类似。我们测试当前节点是否具有搜索关键字。然后，如果它没有，我们把当前节点的关键字和搜索关键字作比较。若前者较大，我们再进行一次递归调用；若较小，则进行一次不同的递归调用。

**程序 13.7 跳跃表中的搜索**

对于单向链表中的搜索，若 k 等于 0，本代码就等价于程序 12.6。对于一般的 k，如果链表中第 k 层的下一个节点的关键字小于搜索关键字，则我们移动到该节点；若它不小于搜索关键字，则我们向下移动到第 k-1 层。

```
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); }
```

当我们希望向跳跃表插入一个新节点时，要面对的第一个任务是确定我们希望该节点拥有多少个链接。所有节点拥有至少一个链接；遵从图 13.22 描述的直觉，我们可以在第二层上每次跳过 $t$ 个节点，如果每 $t$ 个节点中的一个有至少两个链接的话；不断地迭代，我们可得出结论：我们希望每 $t^j$ 个节点中的一个有至少 $j$+1 个链接。

为生成具有这个性质的节点，我们利用一个以概率 $1/t^j$ 返回 $j$+1 的函数进行随机化。给出 j 后，我们创建一个带有 $j$ 个链接的新节点，并利用和我们对搜索操作所采用的同一递归方案（如图 13.23 所示）把它插入到跳跃表。我们到达第 $j$ 层以后，每次我们移动到下一层就把新节点链接到表中。至此为止，我们已确定：当前节点的数据项小于搜索关键字，并（在第 $j$ 层上）链接到一个不小于搜索关键字的节点。

为初始化一个跳跃表，我们建立一个头节点，它拥有我们在该表中允许的最大数量的层，且所有层都有空链接。程序 13.8 和 13.9 实现了跳跃表的初始化和插入操作。

**程序 13.8 跳跃表数据结构和构造函数**

跳跃表中的节点有一链接数组，因此 node 的构造函数必须分配该数组并将所有链接设为 0。常数 lgNmax 是我们在该表中允许的最大层数：对于小型表可设为 5，而对于大型表则可设为 30。和往常一样，变量 N 保存表中数据项的数量，而 lgN 则是层的数量。一个空表是这样一个头节点：具有 lgNmax 个全部设置为 0 的链接，N 和 lgN 也被设置为 0。

```
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; }
```

**程序 13.9 跳跃表中的插入**

我们以概率 $1/2^j$ 产生一个新的 j 链接节点，然后跟随和程序 13.7 完全一致的搜索路径，不过链接新节点的进行是在我们向下移动到底部 j 层中的每一层的时候。

```
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); }
```

图 13.24 显示了以随机次序插入一个关键字集合样本时，一个跳跃表的构造过程。图 13.25 是一个较大的范例；图 13.26 则显示了对于图 13.24 的同一关键字集合，一个跳跃表的构造过程，但插入次序改成了递增顺序。正如随机化 BST 那样，跳跃表的随机性质并不依赖于关键字插入的顺序。

**性质 13.10** 平均情况下，一个具有参数 $t$ 的随机化跳跃表中的搜索和插入操作需要大约 $(t \log_t N)/2=(t/(2 \lg t))\lg N$ 次比较。

我们期望跳跃表有大约 $\log_t N$ 层，因为 $\log_t N$ 大于符合 $t^j = N$ 的最小 $j$。在每一层上，我们预期：在前一层上跳过大约 $t$ 个节点，而且在平均情况下，我们在下降到下一层之前应该要检查这些节点的大约一半。层的数量比较少，从图 13.25 的例子可清楚地看到这一点；但确立这一点的精确分析不属于基础内容（参见“第四部分参考资料”）。

**性质 13.11** 跳跃表平均拥有 $(t/(t-1))N$ 个链接。

在底部存在 $N$ 个链接，第一层有 $N/t$ 个，第二层有大约 $N/t^2$ 个……则整个表中链接的总数大概是

$N(1+1/t+1/t^2+1/t^3\cdots)=N/(1-1/t)$■

挑选一个适当的 $t$ 值——这个问题马上把我们引入到一个时间-空间的权衡问题中。当 $t=2$ 时，跳跃表平均需要大约 $\lg N$ 次比较以及 $2N$ 个链接，其性能足可与我们用 BST 得到的最佳性能媲美。对于较大的 $t$，搜索和插入的时间会较长，但用于链接的额外空间则较少。如果对性质 13.10 的表达式求微分，我们将发现选择 $t=e$ 使得一个跳跃表中搜索操作所需的预期比较次数减到最小。下表以构造一个 $N$ 数据项的表所需的比较次数，给出了 $N\lg N$ 的系数值。

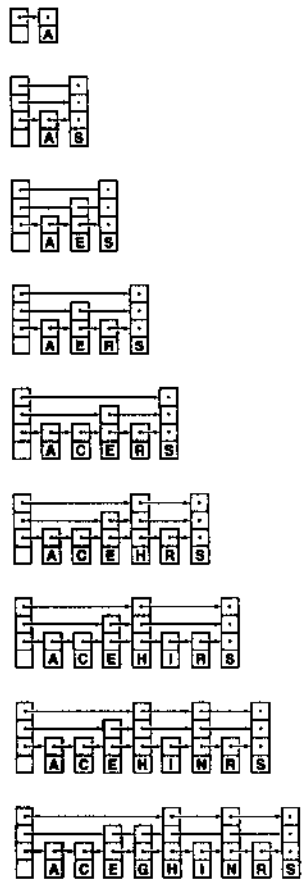



**图 13.24 跳跃表的构造过程**

本系列图例描述了向一个原先为空的跳跃表插入带有关键字 A S E R C H I N G 的数据项的所得结果。节点具有 $j$ 个链接的概率是 $1/2^j$。

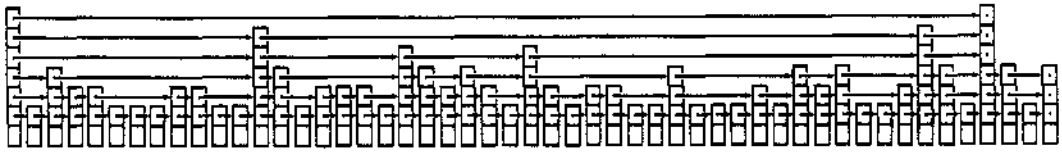


**图 13.25　一个大型跳跃表**

本跳跃表是向一个原先为空的表插入 50 个随机排序关键字的结果。我们可通过跟踪 7 个或更少的链接访问任何节点。

| $t$ | 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 |

如果进行比较、跟踪链接和递归地向下移动这些操作会引来差异极大的开销水平，则我们可以按这些数据行进行更精细的计算（参阅练习 13.83）。

因为搜索时间是对数的，我们可通过增加 t，把空间开销削减到和单向链表（若紧密利用空间的话）相差无几的水平。对运行时间的精确估算依赖于以下两方面的评估：跟踪链接跨越跳跃表的相对开销，以及向下移动到下一层的递归调用。在第十六章我们将重新回到这类时间-空间的权衡问题上来，到时我们会着眼于为大型文件编制索引的问题。

用跳跃表可直截了当地实现其他符号表函数。例如，程序 13.10 利用我们在程序 13.9 对插入函数所用的同一个递归方案，给出了删除函数的一个实现。要实现节点的删除，我们解除该节点在每一层的链接（在我们在插入操作中为它进行链接的位置），解除它在底层链表的链接后我们就释放该节点（和遍历该表以执行插入之前的创建操作相对）。要实现联结的话，我们归并跳跃表（参阅练习 13.78）；要实现选择的话，我们对每个节点添加一个域，它给出指向该节点的最高层链接所跳过的节点数量（参阅练习 13.77）。

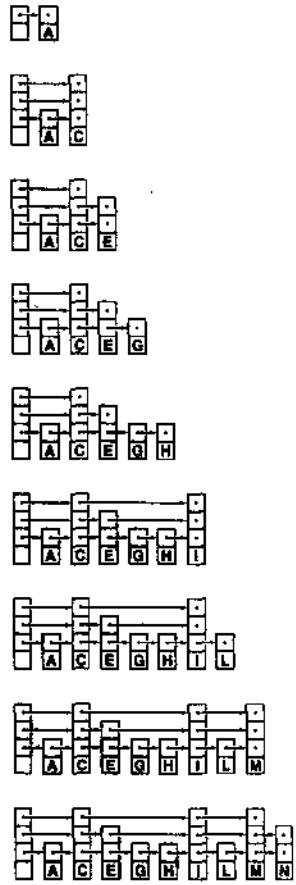


**图 13.26　用有序关键字构造跳跃表的过程**

本系列图例描述了向一个原先为空的跳跃表插入带有关键字 A C E G H I N R S 的数据项的所得结果。本表的随机性质并不依赖于关键字插入的顺序。

尽管跳跃表作为快速移动通过一个链表的一种系统方法，很容易概念化，但明白以下一点也是十分重要的：跳跃表的核心数据结构只不过是平衡树的另一种表示方案而已。譬如，图 13.27 显示了图 13.10 的平衡 2-3-4 树的跳跃表表示。我们可以利用跳跃表抽象而不是 13.4 节的红黑树抽象，来实现 13.3 节的平衡 2-3-4 树算法。所得的代码比我们已讨论过的实现方法要复杂一点（参阅练习 13.80）。在第十六章我们将重新探讨跳跃表和平衡树之间的关系。

**程序 13.10 跳跃表中的删除**

为了从一个跳跃表删除带有给定关键字的一个节点，若我们在某一层找到一个指向它的链接，就解除该节点链接，当我们到达最底层时就删除它。

```
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); }
```

图 13.22 展示的理想化的跳跃表是一个僵化的结构；当我们插入一个新节点时，很难对该结构进行维护，就正如有序数组对于二叉搜索那样，因为插入操作需要在插入节点后修改所有节点的所有链接。软化该结构的一种途径是使得在建立的链表中，每个链接跳过下一层的 1 个、2 个或 3 个链接：这种安排对应于 2-3-4 树，如图 13.27 所示。本节讨论的随机化算法是软化这个结构的另一种有效方法；我们将在第十六章考虑其他可选方案。

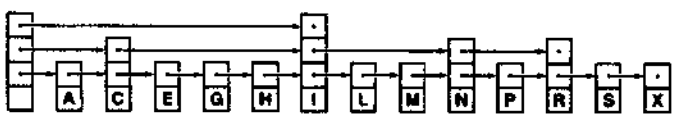



**图 13.27 一棵 2-3-4 树的跳跃表表示**

本跳跃表是图 13.10 的 2-3-4 树的一种表示方法。一般而言，跳跃表对应于每节点有一个或更多链接（也允许一节点，即只有 1 个链接而没有关键字的节点）的多路平衡树。为了建立对应一棵树的跳跃表，我们对每个节点赋予和它在树中的高度相等数量的链接，然后水平地把节点链接起来。为了建立对应一跳跃表的树，我们组合被跳过的节点，并递归地把它们链接到下一层的节点。

## 练习

13.75 画出由以下处理所得的跳跃表：向一个原先为空的表依照关键字顺序 E A S Y Q U T I O N 插入带有这些关键字的数据项，假设 randX 返回的值序列是 1、3、1、1、2、2、1、4、1、1。

▷ 13.76 画出由以下处理所得的跳跃表：向一个原先为空的表依照关键字顺序 A E I N O Q S T U Y 插

入带有这些关键字的数据项，假设 randX 返回的值序列和练习 13.75 相同。

13.77 为一个基于跳跃表的符号表实现选择操作。

● 13.78 为一个基于跳跃表的符号表实现联结操作。

▷ 13.79 修改程序 13.7 和程序 13.9 给出的搜索和插入实现，用一个标志节点而不是 0 结束跳跃表。

○ 13.80 对于使用平衡 2-3-4 树抽象的符号表，利用跳跃表实现构造、搜索和插入操作。

○ 13.81 利用程序 13.9 的 randX()函数建立一个具有参数 $t$ 的跳跃表，平均需要多少个随机数？

○ 13.82 对于 $t=2$，修改程序 13.9，消除 randX 的 for 循环。提示：数值 $t$ 的二进制表示中的最后 $j$ 位，以概率 $1/2^j$ 采用任何特定的 $j$ 位的数值。

13.83 对以下情况选择适当的 $t$ 值，使搜索开销最小：和执行一次比较操作的开销相比，跟踪一个链接的开销是它的 $\alpha$ 倍，移动到下一层递归是它的 $\beta$ 倍。

○ 13.84 开发一个跳跃表实现：它在节点中拥有节点本身，而不是像我们在程序 13.11 到 13.10 那样，拥有指向一个指针数组的指针。提示：把数组放到 node 的末尾。

## 13.6 性能特性

对于一个特定的应用，我们怎样在随机化 BST、发散 BST、红黑 BST 和跳跃表之中进行选择呢？我们已经把注意力集中于这些算法的性能保障的不同性质上。时间和空间永远是主要的考虑因素，但我们还必须考虑到一些其他因素。在本节中，我们将简要讨论实现问题、实验研究、运行时间的估算以及空间要求。

所有基于树的算法都依赖旋转：顺着搜索路径的旋转实现是大多数平衡树算法的一个核心要素。我们已经使用过递归实现方式，它们隐式地把指向搜索路径上的节点的指针保存在递归堆栈上的局部变量中；但是每种算法都可以用一种非递归方式实现——在一遍自上而下通过树的处理过程中，操作常数个节点，对每个节点执行常数次链接操作。

随机化 BST 是三种基于树的算法中最容易实现的。主要的要求是对随机数生成器有信心，并避免花费太多时间生成随机位。发散 BST 稍稍复杂一点，却是标准根插入算法的一种直接扩展。红黑 BST 还需要涉及到稍多一点的代码，以便检查并控制颜色位。红黑树相对其他两者的一个优点，是颜色位可用于调试中的一致性检测，并可作为在树的生存期内任何时刻的一次快速搜索的一个保障。从对一棵发散 BST 的检查无法知道产生该树的代码是否完成了所有正确的转换工作：一个错误（bug）可能（也只能！）导致性能问题。类似地，随机化 BST 或跳跃表的随机数生成器的一个错误也可能引致性能问题，不加以注意的话将无法发现这些问题。

跳跃表易于实现，而且如果要支持全部符号表操作，它是特别有吸引力的方案；因为搜索、插入、删除、联结、选择和排序都具有易于编制的自然实现方法。跳跃表中搜索的内层循环比树结构的要长（它涉及到指向指针数组的一个额外索引，或者向下移动到下一层的一个额外递归调用），因此搜索和插入所需时间也较长。跳跃表也使得程序员受到随机数生成器的支配——调试一个行为随机化的程序的确是一个很大的挑战，而一些

程序员会发现：和拥有随机数量链接的节点打交道尤其使人不安。

对于我们在本章中讨论过的四种方法以及来自第十二章的基本 BST 实现方法，表 13.1 给出了关于它们的性能的实验数据，其中所针对的关键字是随机 32 位整数。该表的信息证实了我们从 13.2 节、13.4 节和 13.5 节中的分析结果所得的预期。对于随机关键字，红黑 BST 比其他方法快得多。红黑 BST 中的路径比随机化 BST 或发散 BST 短 35%，而且在内层循环中进行的处理也较少。随机化树和跳跃表要求我们对每次插入操作生成至少一个新的随机数，而发散 BST 对每次插入和每次搜索都需要在每个节点进行一次旋转操作。相比之下，红黑 BST 的开销则是：我们在插入期间必须在每个节点检查 2 位的数值，并偶尔需要执行一次旋转。对于不均匀的访问，发散 BST 也许使用较短的路径，但这些开销节省很可能被以下的事实抵消：在内层循环中，搜索和插入操作都涉及到每个节点处的旋转操作；只有极端的情况例外。

**表 13.1 平衡树实现的实验研究**

对于多个不同 $N$ 值，用 $N$ 个 32 位整数的随机序列生成 BST，而建立并搜索 BST 的相对时间表明：所有方法都具有良好性能——即使对巨大的表，但红黑树明显要比其他方法快得多。所有方法都利用标准 BST 搜索，只有两种方法例外：在发散 BST 中，我们在搜索时进行发散操作，把经常访问的关键字带到接近树顶部的地方；而跳跃表使用了本质上相同的算法，只不过采用一种不同的基本数据结构。

| | 构造 | | | | | | 搜索命中 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | 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 |
| 10000 | 159 | 277 | 447 | 282 | 177 | 310 | 118 | 106 | 132 | 112 | 84 | 229 |
| 20000 | 347 | 621 | 996 | 636 | 411 | 670 | 235 | 234 | 294 | 247 | 193 | 523 |

其中：

B 标准 BST（程序 12.8）
T 用根插入构造的 BST（程序 12.13）
R 随机化 BST（程序 13.2）
S 发散 BST（程序 13.5 以及练习 13.33）
C 红-黑 BST（程序 13.6）
L 跳跃表（程序 13.7 以及程序 13.9）

发散 BST 无须对平衡信息使用额外空间，红黑 BST 需要额外的 1 位，而随机化 BST 则需要一个计数域。对于许多应用来说，这个计数域是出于其他原因而保持的，所以对于

随机化 BST 这也许并不代表着额外的开销。实际上，如果使用发散 BST、红黑 BST 或跳跃表的话，我们可能需要加入这个域。若有必要，我们可以通过消除颜色位，使红黑 BST 像发散 BST 那样高效地利用空间（参阅练习 13.65）。空间曾经一度是编程的首要问题；在现代应用中其重要程度已大大降低，但细心的程序员依然必须对资源浪费保持警觉。例如，我们必须清楚地知道：某些系统可能对一个小小的计数域或节点中的 1 位颜色域使用一整个 32 位的存储字，而另外一些系统也许在内存中压缩这些域，以至于解压过程需要大量额外的时间。如果紧密利用空间，具有较大参数 $t$ 的跳跃表可以把用于链接的空间削减将近一半；付出的代价是搜索速度减慢，但仍处于对数水平。通过某些规划，也可以把基于树的方法实现为每节点一个链接（参阅练习 12.68）。

总体上来讲，我们在本章讨论过的所有方法都将为典型应用提供良好的性能；对于有兴趣开发出一种高性能的符号表实现方案的人士，它们各有优点。发散 BST 作为一种自组织搜索方法将提供良好的性能，尤其是频繁访问一个小型关键字集合成为应用的典型模式时；而对于一个全功能的符号表 BST，随机化 BST 可能更快、更容易实现；跳跃表易于理解，而且能够以少于其他方法的空间提供对数水平的搜索性能；红黑 BST 对于符号表库实现极具吸引力，因为它们在最坏情况下可提供得到保障的性能界限，还能提供对应随机数据的最快的搜索和插入算法。

除了在许多应用中的特定用途外，这个对应“为符号表 ADT 开发出高效实现方案”的问题的全套解决方案还有十分重要的意义。因为它展示了在我们考虑其他问题的解决方案时，可供选择的算法设计基本途径。在我们对简单、最优算法的不断追求中，我们常常遇到接近最优的有用算法，例如这里讨论的这些。而且，正如我们对排序所见那样，像这些基于比较的算法仅仅是个开始——通过转向一个更低层次的抽象（其中我们可以逐个处理关键字），我们可开发出甚至比本章讨论的更快速的实现方法，在第十四章和第十五章我们将看到这一点。

## 练习

13.85 利用随机化 BST 开发一个符号表实现，它包括一个析构函数、一个复制构造函数和一个重载赋值操作符，并支持和具有客户数据项处理句柄的一流符号表 ADT 相对应的构造、计数、搜索、插入、删除、联结、选择和排序操作（参阅练习 12.6 和 12.7）。

13.86 利用跳跃表开发一个符号表实现，它包括一个析构函数、一个复制构造函数和一个重载赋值操作符，并支持和具有客户数据项处理句柄的一流符号表 ADT 相对应的构造、计数、搜索、插入、删除、联结、选择和排序操作（参阅练习 12.6 和 12.7）。

# 第十四章

# 散　　列

我们前面介绍的搜索算法是基于比较操作的。与之不同的是 12.2 节中介绍的关键字索引的搜索方法，该方法是把关键字为 $i$ 的元素存储在表的第 $i$ 个位置，使其可以直接访问。关键字索引的搜索算法使用关键字值作为数组下标，而不是对它们进行比较。而且关键字值必须是落在表的地址范围内的互不相等的整数。在这一章，我们将介绍一种关键字值索引搜索算法的扩展——散列。它可以处理很多具有不规则特性关键字的典型搜索应用，那些关键字是我们无法预先获得的。与词典数据结构中通过比较待查关键字和元素关键字来实现搜索所不同的是，我们试图通过算术运算将关键字转换成表的地址来标记表中的元素。最终我们得到一种与基于比较的搜索所完全不同的搜索算法。

散列搜索算法由两部分组成。首先是能够将查询关键字转换成表地址的散列函数。理想的情况是不同的关键字应映射到不同的地址，但经常出现两个或多个关键字对应同一表地址的情况。处理这种关键字的过程就是散列搜索算法的第二部分——冲突调节。我们将要研究的冲突调节方法之一是使用链表，它对待查关键字数目无法预知的动态环境有直接的帮助。我们将要考察的另外两种冲突调节方法可以实现在固定数组上的快速搜索。另外，我们还将研究一种对这些方法的改进，以处理我们无法预知表长的情况。

散列方法对搜索时间和存储空间的使用达到较好的折中。在关键字索引的搜索算法中，如果没有存储量的限制，我们用关键字作为存储地址，只用一次存储器访问就可以完成任何查询操作。但是，长关键字使得所需存储量过大无法满足，所以上述理想情况通常是不能实现的。另一方面，如果没有搜索时间的限制，我们可以通过采用顺序搜索算法，从而实现最小量的存储空间。散列提供了一种使用合理的存储量和搜索时间的方法来达到两者的平衡。而且，我们仅仅通过调节散列表的大小而不用重写代码或者更换算法就可以取得我们所选择的平衡。

散列是一个经典的计算科学的问题，各种算法被深入研究和广泛应用。在较为宽松的假设条件下，我们应当看到下面的期望是合理的：搜索和插入符号表的时间独立于表的大小而成为常数。

这个期望是任何对符号表操作的理论最佳性能，但出于下面两个原因，散列搜索并不

是万能的：第一，运算时间依赖于关键字的长度，这对实际的长关键字搜索十分不利；第二，对符号表的其他操作，如选择和排序等，散列并未提供有效的实现方法。在本章中，我们将详细讨论上述以及其他问题。

## 14.1 散列函数

能将关键字转换成表地址的函数称为散列函数。我们要提的第一个问题是如何构造散列函数。通常，算术运算容易实现，但我们也必须谨慎以避免小的疏漏。如果我们有一个长度为 $M$ 的表，那么我们需要的是能将关键字转化成[0, $M$−1]范围内整数的函数。理想的散列函数要易于计算并且近似为随机函数。在这里，随机是指对每一个输入，相应的输出应在某种程度上是等概率的。

散列函数随关键字的不同而不同。严格来说，对每一种可能用到的关键字，我们需要不同的散列函数。为了提高效率，我们通常希望避免显式类型转换，而争取得到基于如下思想的转换：将关键字表示为二进制机器码里可以用来做算术计算的整数。在早期的计算机中，散列先于高级语言把一个值首先看作是一个字符串型数据，其次才看作是一个整型数据，这种做法在一段时期内是比较通常的做法。某些程序依赖于关键字在特定机器上的表示方法，由于这些程序对机器的依赖性，它们很难移植到另外的机器上，所以一些高级语言编写这种程序就显得很困难。散列函数通常依赖于将关键字转换为整数的过程。因此，机器独立性和高效率有时很难同时实现。我们可以使用一条机器指令来散列简单的整型或浮点型关键字，但对字符串或其他复合类型的关键字，我们需要更多的注意效率问题。

```
.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
```

**图 14.1 应用于浮点型关键字的乘法散列函数**

我们把关键字乘以 97 就可以把介于 0、1 之间的浮点数转换成大小为 97 的表的索引。在这个例子中，出现了三处冲突，分别在地址 17、53 和 67 处。散列值是由关键字的若干最高位决定的，而最低几位不起作用。设计散列函数的一个目的就是要避免这种不平衡，使得每位数在运算中均起作用。

最简单的情况可能是已知的固定范围内的浮点数作为关键字。例如，关键字是 0 到 1 之间的数，我们可以把每一个数乘以 $M$，然后取最接近的整数，这样就得到 0 到 $M$−1 之间的整数地址。图 14.1 给出了一个例子。对于固定的数 $s$ 和 $t$，如果关键字大于 $s$ 小于 $t$，我们可以通过减 $s$ 再除以（$t$−$s$）便得到 0 到 1 之间的数，然后乘以 $M$ 就是表地址。

如果关键字是 $w$ 位整数，我们可以通过除以 $2^w$ 得到 0、1 之间的浮点数，然后按照前面介绍的那样进行运算（乘以 $M$）。如果浮点运算开销太大并且数字不会大到溢出的话，我们可以通过整数的算术运算得到同样的结果：把关键字乘以 $M$，再右移 $w$ 位来实现除以 $2^w$（或者，如果乘法会导致溢出，可以先移位再相乘）。只有当关键字在一定范围内平均分布

的时候，这种散列函数才是有效的，因为散列值只由关键字的若干高位决定。

下面介绍一种简单有效而且十分常用的散列函数——模散列函数。构造该函数的方法是选择表长 $M$ 为素数，对任何整型关键字 $k$，计算 $k$ 除以 $M$ 的余数，即：

$$h(k)=k \bmod M$$

$h(k)$就是模散列函数。这种运算易于实现（在 C++中为 k%M），而且可以高效地将小于 $M$ 的整型关键字均匀分散开。图 14.2 给出了一个小例子。

对浮点关键字也可以使用模散列函数。如果关键字在一个小的范围内，我们可以按比例将它们转换为 0 到 1 之间，再乘以 $2^w$ 以得到 $w$ 位结果，然后使用模散列函数。另一种方法是使用关键字的二进制表示（如果可能的话）作为模散列函数的操作数。

无论关键字是表示成机器码的整数，还是压缩成一个机器字的一串字符，或者是其他极多种可能情况，只要我们能得到组成关键字的每一位数字，我们就可以应用模散列方法。由于有些数位是用来编码的，故一个压缩成机器字的字符串与一个随机整型关键字有很大不同，但我们可以将上述两者（连同其他类型的机器码关键字）看成是索引到一个小型表中的随机序列。

我们选择模散列表长度 M 为素数，图 14.3 解释了这样做的首要原因。在这个 7 位编码字符数据的例子中，我们把关键字当作基数为 128 的数据——在关键字中，一位数代表一个字符。比如，与 1816567 对应的数也可以表示为：

$$110*128^2+111*128^1+119*128^0$$

由于字母 n、o 和 w 的 ASCII 码为 $156_8$=110，$157_8$= 111，$168_8$=119。这样，选择表长 $M$=64 不适合这种类型的关键字。因为 $x$ mod 64 的值不受 64（或 128）的倍数与 $x$ 之和的影响——该散列函数对任何关键字的返回值是关键字的后 6 位。一个好的散列函数应该考虑关键字的全部位，特别是由字符组成的位。当 $M$ 的因子含有 2 的幂的时候，类似情况也会发生。因此，避免这种现象的最简单方法是取 $M$ 为素数。

| | | | |
|---|---|---|---|
| 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 |

**图 14.2 应用于整型关键字的模散列函数**

图中最右三行依次列出了采用三个函数对图中最左边一行的 16 位数进行散列的结果，这三个函数为：

v%97(对左边一行)

v%100（对中间一行）

(int)(a*v)%100（对右边一行）

这里，a 取 0.618033。对三个不同函数，相应的表长分别为 97、100 和 100。散列结果呈随机分布（因为关键字是随机的）。中间的函数（即 v%100）仅仅使用关键字的最高两位数，因此，对非随机关键字其性能会变差。

除了要求表长为素数之外，模散列没有什么难于实现之处。对某些应用而言，用已知的小素数就可以满足要求，或者我们还可以从已知的素数表中找一个与表长接近的素数来

实现。例如，当 $t$=2、3、5、7、13、17、19 和 31 时（再无其他 $t$<31 满足此条件），形如 $2^t-1$ 的数是素数，这就是著名的默森尼（Mersenne）素数。为了对某一个固定长度的表实现动态分配，我们需要计算与表长度接近的素数，这种计算就不是那么容易实现了（但我们将在第五部分介绍一种巧妙的算法）。因此，实际应用中，我们通常采用一个预先计算好的表（见图 14.4）。使用模散列函数不是要求表长为素数的惟一原因，我们将在第 14.4 节考虑另外一个原因。

对于整数关键字，另一种可供选择的方法是乘法与取模两种算法的结合：将关键字乘以 0 到 1 之间的一个常数，然后做模 $M$ 运算。相当于采用形如下式的散列函数

$$h(k)=\lfloor k\alpha \rfloor \bmod M$$

理论上，$\alpha$ 和 $M$ 的值之间存在互相影响，而且关键字的有效基数可能导致异常结果，但实际应用中，如果我们采用随机的 $\alpha$ 值，将不会出现问题。一个常用的 $\alpha$ 值是 0.618033……（即黄金分割比）。关于 $\alpha$ 的选择变化有很多研究，比较突出的有对高效机器指令（如移位和掩码）实现的散列函数（参见“第四部分参考资料”）。

在很多符号表应用中，关键字不是整数且并不短，而是较长的字母序列。我们如何计算字符串“averylongkey？”的散列值呢？在 7 位 ASCII 码中，该关键字对应于如下 84 位整数：

$$97*128^{11}+118*128^{10}+101*128^{9}+114*128^{8}+121*128^{7}$$
$$+108*128^{6}+111*128^{5}+110*128^{4}+103*128^{3}+107*128^{2}$$
$$+101*128^{1}+121*128^{0}$$

该数太大，无法在大多数计算机中表示为普通算术函数。况且我们还会遇到更长的关键字，那情形将更为复杂。

```
now 6733767 1816567 55 29
for 6333762 1685490 50 20
tip 7232360 1914096 48  1
ilk 6473153 1734251 43 18
dim 6232355 1651949 45 21
tag 7230347 1913063 39 22
jot 6533764 1751028 52 24
sob 7173742 1898466 34 26
nob 6733742 1816546 34  8
sky 7172771 1897977 57  2
hut 6435364 1719028 52 16
ace 6070745 1602021 37  3
bet 6131364 1618676 52 11
men 6671356 1798894 46 26
egg 6271747 1668071 39 23
few 6331367 1684215 55 16
jay 6530371 1749241 57  4
owl 6775754 1833964 44  4
joy 6533771 1751033 57 29
rap 7130360 1880304 48 30
gig 6372347 1701095 39  1
wee 7371345 1962725 37 22
was 7370363 1962227 51 20
cab 6170342 1634530 34 24
wad 7370344 1962212 36  5
```

**图 14.3　应用于编码字符的模散列函数**

图中每行依次列出一个 3 字符的字、该字的八进制和十六进制的 21 位编码以及表长分别为 64 和 31 的标准模函数（在最右两列）。表长 64 的散列结果是我们所不希望的，因为只有最右边几位数影响了散列值，而且自然语汇字符的散列值没有达到均匀分布。例如，所有以 y 结尾语汇的散列值均为 57，与之相比，表长为素数 31 的散列表产生的冲突数小于其长度的一半。

**程序 14.1　字符串关键字的散列函数**

本算法实现字符串关键字的散列函数，对关键字中的每一个字符进行加、乘运算各一次。如果我们把常数 127 换成 128，程序将采用霍纳（Horner）算法，只计算 7 位 ASCII 码对应的数被表长除所得的余数。如果表长为 2 的倍数或 2 的幂，采用基数为 127 的素数可以使我们避免异常情况。

```
int hash(char *v, int M)
  { int h = 0, a = 127;
```

```
    for (; *v != 0; v++)
      h = (a*h + *v) % M;
    return h;
  }
```

我们可以通过一段一段地将关键字转化来计算长关键字的模散列函数值。而且，我们可以利用模函数的算术特点并使用霍纳（Horner）算法（见 4.9 节）。这种算法是基于另外一种求关键字对应的散列值的方法。例如，我们写出如下的表达式：

(((((((((97*128+118)*128+101)*128+114)*128+121)*128
+108)*128+111)*128+110)*128+103)*128+107)*128
+101)*128+121

即我们可以按字符串从左到右的顺序计算出每一个字符编码对应的十进制数，然后乘以 128，再加下一个字符的编码值。该计算的结果大于我们机器所能表示的最大整数，但我们的兴趣不在于计算该数，我们需要的仅仅是它除以 $M$ 以后的余数，当然这个余数应该是相对较小的。另外，我们也不用多次累加到一个相当大的数就可以得到结果，我们可以在运算的每一步都进行一次模 $M$ 运算——每次加乘运算之后我们只需保存模 $M$ 的值——这样得到的结果与每次累加（如果我们的机器字长够大的话）最后除以 $M$ 得到的结果是一致的（见练习 14.10）。该程序采用了一点巧妙的手法：采用素数 127 代替 128 作基数。其原理将在下一段介绍。

| $n$ | $\delta_n$ | $2^n-\delta_n$ |
|---|---|---|
| 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 |

**图 14.4 可供散列表采用的素数**

对于 $8\le n\le 32$，图中的最大素数小于 $2^n$。这些素数可用于动态分配且表长为素数的散列表。对任意给定上述范围内的正数，我们可以用该表得到一个不大于 2 的该数次幂的素数。

与采用霍纳（Horner）算法的模散列相比，在相同计算开销（即对关键字中每个字符进行一到两次算术运算）的前提下，有多种计算散列函数的方法。就随机关键字而言，这几种方法没有什么不同，但实际应用中关键字一般不是随机的。为了使实际关键字随机化，我们不得不考虑散列中的随机化算法——因为对于任何类型的关键字，我们都希望散列函数产生随机的表地址。我们不必死抠模散列函数定义的字面意思——即我们仅希望把关键字中的每个字符转化成一个小于 $M$ 的整数，所以这里的随机化并不难实现。程序 14.1 说明了一种实现方法：采用素数而不是采用定义中要求的采用 2 的幂作为相应于 ASCII 字符的基数。图 14.5 解释了这种改变如何避免了典型字符串关键字的离散问题。理论上讲，由程序 14.1 得到的散列值对表长为 127 倍数的表是不利的（但实际应用中这种影响极小）。我们可以选择乘数为随机值的算法。另一种更有效的方法是在计算中使用随机系数，并且对关键字中每位数采用不同的随机值。这种算法称为通用散列算法（universal hashing）。

**程序 14.2　用于字符串关键字的通用散列函数**

本程序与程序 14.1 做相同的运算，但它采用伪随机系数代替固定系数。这样近似达到了理想的情况，即对两个不相等关键字发生冲突的可能性为 $1/M$。在计算散列函数的时候，我们采用一种较简易的随机数产生算法以免浪费过多的时间。

```
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;
  }
```

理论上讲，一个理想的通用散列函数使得两个不同关键字产生冲突的几率为表长的倒数，即 $1/M$。我们可以证明，采用一个每个值不同的随机序列而非固定值作为程序 14.1 中的系数 $\alpha$ 可使得模散列函数成为一种通用散列函数。然而，我们不能为关键字中的每一个字符耗费时间去生成相应的随机数。程序 14.2 给出了一个可行的折中方法：我们可以产生一个伪随机序列来改变系数 $\alpha$。

总之，对一个抽象符号表采用散列表实现，第一步要采用一种把关键字映射到一个小于表长 $M$ 的非负整数的散列函数将抽象类型接口进行扩展。下列程序对介于 $s$ 到 $t$ 范围内的浮点数实现了上述操作：

```
inline int hash(Key v, int M)
  { return (int) M*(v-s)/(t-s); }
```

对整数关键字，我们可以直接使用返回值 v % M。如果 $M$ 不是素数，可以用下面的散列函数返回值：

```
(int) (.616161 * (float) v) % M
```

或者采用如下整数运算：

```
(16161 * (unsigned) v) % M
```

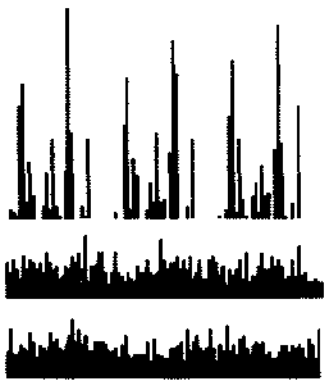


**图 14.5　字符串关键字的散列函数**

图中上、中、下三行分别显示了程序 14.1 采用下列参数：

M=96 和 a=128（上）

M=97 和 a=128（中）

M=96 和 a=127（下）

对一组英文单词（取自 Melville 的《Moby Dick》一书中的前 1000 个不同单词）散列所得的离散值。第一个例子中，字母的不均匀使用和表长与乘数有 32 这一公因子联合导致了较差的离散结果，而且使得结果分布不均匀。另外两个例子的结果是均匀分布的，因为其表长与乘数互质。

包括应用于字符串关键字的程序 14.1 在内的所有这些散列函数，都是多年来程序员们普遍采用的古老而经典的算法。程序 14.2 的通用算法由于仅用很小的额外开销就提供了随机的散列结果，故对字符串关键字的散列有明显改进。我们还可以类比得到对整数关键字的相似算法（见练习 14.1）。

在实际应用中，通用散列比简单的散列算法慢许多，因为关键字的每个字符都要做两

次算术运算对于长关键字而言是相当耗时的。为了改善这一缺陷，我们可以将关键字分成较大的段。事实上，我们可以像在基本模散列中那样把分段扩大至不超出机器字的限度。正如我们前面详细讨论的，这类特殊的操作难于实现，而且在某些特别强调数据类型的高级语言中容易导致循环错误或者死循环。但在 C++中，采用适当的数据类型转化而无需额外的工作即可实现。在很多情况下，这些因素要认真考虑，因为散列函数的计算很可能在内循环中，因此提高散列的运算速度也就提高了整个运算的速度。

除了介绍这些算法的好处之外，在实现这些算法的时候也要十分小心。因为，第一，我们在进行数据类型转化和用算术函数作用于在不同机器上表示的关键字的时候，要十分小心地避免程序出错。特别是一个程序由旧机器移植到不同字长的新机器上的时候，这种移植是典型的错误源。第二，散列函数极有可能包含在程序的内层循环中，因此它的运算时间基本上决定了程序的运算时间。在这种情况下，很有必要搞清楚它（指散列函数——译者注）是否降低了机器代码的效率。这些操作也是明显导致代码效率低的原因。例如，简单模散列与先乘系数 0.61616 的模散列对浮点操作数的运算在一台较慢的机器上运算时间的差别是惊人的。对多数机器而言，最快的算法是使 M 为 2 的幂且使用下面的散列函数

```
inline int hash(Key v, int M)
  { return v & (M-1); }
```

该函数只使用关键字的最低位，但是逐位与运算比整数除法快很多，从而以效率的提高部分地抵消了差的离散结果带来的负面影响。

在实现散列时，散列函数总是返回同一值是一个较为典型的错误。其产生原因可能是一次数据类型转化没有正确完成。这种错误被称为执行（performance）错误，因为使用这种散列函数的程序可能会正确执行，但会非常慢（因为它被设计成仅对离散情况良好的值达到高效率）。这类函数很容易用一行程序来实现，因此我们应该对任何遇到的特殊符号表关键字类型都检验一下它的执行情况。

我们可以用 $x^2$ 统计量来检验散列函数产生的序列是否随机（见练习 14.5），但或许这个要求过于苛刻。事实上，只要一个散列函数对某个输入值每次的输出结果相同，也就相当于 $x^2$ 统计量等于 0，我们就可以满足了。我们仍然要考虑大量 $x^2$ 统计的情况。在实际应用中，采用一组分布范围广从而没有哪一数据处于支配地位的测试数据就足够了（见练习 14.15）。出于这种考虑，一个较好的基于通用散列算法的符号表实现有时需要检查散列值是否均匀地离散开。客户也需要被告知是否发生过小概率事件或者散列函数中存在程序错误。在实际的随机算法中加入这类检查是明智的。

## 练习

▷ 14.1 采用第十章的 digit 抽象，将一个机器字看作一个字节序列，对以机器字中的位的形式表示的关键字实现一个随机散列函数。

14.2 在你的编程环境下检查将 4 字节关键字转化为 32 位整数是否存在执行时间开销。

○ 14.3 按下述思路设计一个散列函数：将字符串关键字每 4 字节取一段，然后对每 32 位进行算术运算。最后，将运算时间与程序 14.1 对 4、8、16 和 32 字节的运算时间进行比较。

14.4 编写一个程序，寻找$\alpha$和$M$的值，使$M$尽可能小，而且用散列函数$\alpha$*x % M 对图 14.2 提供的数据产生不同的结果（即不发生冲突）。所得结果即可作为一个理想散列函数的例子。

○ 14.5 对$N$个关键字和表长为 M 的散列表，编写一个程序，计算$x^2$统计量，该统计量由下式给出定义：

$$\chi^2 = \frac{M}{N}\sum_{0\le i<M}(f_i - \frac{N}{M})^2$$

其中，$f_i$是散列值$i$对应的关键字的数目。如果散列值是随机的，由于$N>cM$，统计量应该以 $1-1/c$ 的概率等于$M \pm \sqrt{M}$。

14.6 使用 14.5 的程序评价散列函数 618033*x % 10000 对大于 $10^6$ 的随机正数的计算结果。

14.7 从你的系统中的某个大文件（比如一个词典）中取出不同的字符串作为关键字，并用练习 14.5 的程序来评价程序 14.1 中的散列函数。

● 14.8 假设采用$t$位整数作关键字，请证明素数$M$的模散列函数的下述特征：对于只有关键位不同的两个关键字，散列得到不同的散列值。

14.9 考虑整数关键字的采用任意固定素系数α的模散列函数(α*x) % M，问：采用非素数 M 能取得好的散列效果吗？

14.10 假设 a、b、x 和 M 是非负整数，请证明：

$$(((ax) \bmod M) + b) \bmod M = (ax + b)$$

▷ 14.11 如果从一个文本文件，如练习 14.7 的一本词典中，取词作为关键字，那么无法得到好的$x^2$统计结果。请解释这个结论。

14.12 使用练习 14.15 的程序评价散列函数 97*x % M，表长介于 100 到 200 之间，关键字采用小于 $10^6$ 的 $10^3$ 个随机正整数。

14.13 使用练习 14.15 的程序评价散列函数 97*x % M，表长介于 100 到 200 之间，关键字采用 $10^2$ 到 $10^3$ 之间的整数。

14.14 使用练习 14.15 的程序评价散列函数 100*x % M，表长介于 100 到 200 之间，关键字采用小于 $10^6$ 的 $10^3$ 个随机正整数。

14.15 引入一个更简单的判据，即产生同一散列值次数大于 3N/M 次就否定改方法，采用本判据完成练习 14.12 和 14.14。

## 14.2 链地址法

我们在 14.1 节中介绍的散列函数将关键字转化成散列表地址，这是散列算法的第一部分；其第二部分就是如何处理冲突问题，即两个关键字散列到同一地址的问题。最简单的方法是对每个散列地址建立一个链表，将散列到同一地址的不同关键字存入相应的链表中。这需要用到程序 14.3 中给出的基本链表搜索算法（见第十二章）。不同的是在这里不是建

一个表，而是建 M 个表。

**程序 14.3 采用链地址法的散列**

本符号表的实现是基于程序 12.6 的代码，但是把其中的 ST 构造函数和 search、insert 函数分别用这里给出的代码来代替，而且将程序 12.6 中的链地址头用这里的链地址头数组来代替。用一个散列函数生成散列值从而选择对应的链表．构造函数按每个链表长度期望值为 5 对 M 进行赋值，因此其他操作只需几次探测就可以完成。

```
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++; }
```

传统上这种方法被成为链地址法，因为发生冲突的元素通过分离的 $M$ 个链表连接起来。图 14.6 显示了一个链地址法的例子。由于采用基本的顺序搜索，我们可以使冲突元素在各自的链表中有序或无序存储。正如 12.3 节中讨论的那样，对链地址法也可采用基本的折中方法，但这样节省的时间开销并不明显（因为链表短），而空间开销更显著（因为链表数目多）。

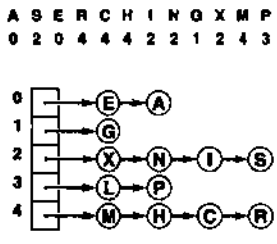



**图 14.6 采用链地址的散列**

图 14.6 给出了将关键字 A S E R C H I N G X M P L 采用链地址法（链表元素为无序存储）插入到空散列表的结果，使用的散列值在图的上端给出。其中，A 插入表位置 0，S 插入 2，E 插入 0（为了保持插入操作的时间为常数，故在表头插入），R 插入 4，以此类推。

在单链表中，我们可以采用一个头节点使得有序表的插入操作的代码更简洁，但我们不会对相互独立的 $M$ 个地址链使用 $M$ 个头节点。事实上，我们可以用每个链表的第一个节点组成散列表，从而删去散列表到链地址的 $M$ 个链接（见练习 14.20）。

对于一次搜索失败，我们能够认为该散列函数有效地散列了关键字的值，使得 $M$ 个地址链中每一个都被等概率地搜索到。对每一个地址链，我们在 12.3 节讨论的性能均适用。

**性质 14.1**　链地址法使用 $M$ 个链表的额外空间，将顺序搜索的比较次数减少的倍数与 $M$ 成比例（从平均意义上讲）。

如第十二章介绍的，链表的平均长度是 $N/M$，搜索成功所需比较次数的期望值只是表长的一半。对于无序链表，不成功的搜索将从头至尾比较整个链表，而有序链表只需比较一半长度。■

通常，我们在链地址法中采用无序链表，因为它既易于实现又高效：插入操作所需时间为常数，搜索操作耗时与 $N/M$ 成比例。如果考虑可能产生大量数目搜索失败的情况，我们可以采用有序链表来使搜索速度加快一倍，但代价是插入操作减慢。

上述性质 14.1 并不重要，因为无论元素以何种方式分配到地址链表中，平均链表长均为 $N/M$。例如，全部元素都落在第一个地址链中，而平均链表长为$(N+0+0+\ldots+0)/M=N/M$。散列在实际中有用的真正原因在于每个地址链以极大的概率包含 $N/M$ 个元素。

**性质 14.2**　在一个采用链地址法的散列表中，有 $M$ 个地址链和 $N$ 个关键字，则每个地址链中元素数目在 $N/M$ 的一个小的因子范围内的概率非常接近 1。

对于熟悉基本概率知识的读者，我们简单地分析一下这一经典的结论。一个地址链含有 $k$ 个元素的概率为：

$$\binom{N}{k}\left(\frac{1}{M}\right)^k\left(1-\frac{1}{M}\right)^{N-k}$$

这是一个基本根据。我们在 $N$ 个元素中选择 $k$ 个：所有 $k$ 对应的元素概率为 $1/M$，剩下 $N-k$ 个元素未散列到已知地址链中的概率为 $1-1/M$，考虑到$\alpha = N/M$，我们可以重写上式为

$$\binom{N}{k}\left(\frac{\alpha}{N}\right)^k\left(1-\frac{\alpha}{N}\right)^{N-k}$$

根据经典的泊松（Poisson）近似公式，上式小于

$$\frac{\alpha^k e^{-\alpha}}{k!}$$

根据这个结果，可得一个地址链含有多于 $t\alpha$ 个元素的概率小于

$$\left(\frac{\alpha e}{t}\right)^t e^{-\alpha}$$

就实际参数变化范围而言，这个概率是相当小的。例如，平均地址链长度为 20，我们散列到某一地址链的元素个数大于 40 的概率小于 $(40e/2)^2 e^{-20} \approx 0.0000016$。■

上述分析是一个经典的“占有问题”的例子，即将 $N$ 个球随机投到 $M$ 个缸里，并分析球在不同缸里的分配情况。对这些问题的经典数学分析告诉我们许多跟研究散列算法相关的结论。让我们看一个例子，泊松近似公式告诉我们空地址链的个数为 $e^{-\alpha}$。一个更为有趣的结论是在第一次冲突发生前已经插入的元素平均为 $\sqrt{\pi M/2} \approx 1.25\sqrt{M}$ 个。这个结论就是经典的“生日问题”的解答。例如，类似的分析告诉我们取 $M$=365，我们要找两个同日出生的人，那么第一次找到这样两个人之前我们共要找 24 人。另一个经典结论告诉我们在

每个地址链中都至少含有一个元素之前，共有 $MH_M$ 个元素已被插入表中。这个结论是经典的“期票收藏问题”的解答。让我们再看一个例子，取 $M$=1280，在收藏到联赛 32 支球队每队 40 名球员的全部球星卡之前，我们要收集 9898 张球星卡。这些结论体现了前面所分析的散列的特征。实际上，如果散列函数产生近似随机的散列值，我们便可以放心地使用链地址法（参见“第四部分参考资料”）。

在实现链地址法的时候，我们既要取 $M$ 尽量小以避免空地址链浪费大量连续的内存空间，又要让 $M$ 足够大以使顺序搜索效率最高。混合法（比如采用二叉树代替链表）的使用可能会得不偿失。根据经验，我们应该选择 $M$ 大约为链表中关键字数目的 1/15 或者 1/10，这样每个地址链所包含元素数目的期望值为 5 个或 10 个。链地址法的一个好处是 $M$ 的选择并非决定性的：如果更多的关键字不期而至，搜索仅仅比提前选择相对稍大的链表长的情况多消耗一小段时间；如果更小的关键字出现在链表中，那么我们将获得更快速的搜索，但同时可能会浪费少量空间。当空间不是关键性资源的时候，不妨选择足够大的 $M$ 以使搜索时间为常数；当空间至关重要的时候，仍可以选择 $M$ 不超过我们所能承受的范围，并使性能得到随 $M$ 成一定比例的提高。

上一段的内容是出于对搜索时间的考虑。在实际应用中，由于两方面的原因通常采用无序的地址链。第一，正如我们已经提到的，插入操作非常快：即计算散列函数，为节点分配内存，把节点链接到相应的地址链的头部。在很多应用中，并不需要分配内存这一步（因为插入到符号表的元素可能已经是含有可用链接字段的记录），因而我们可能只需 3 到 4 个机器指令就可以完成插入操作。在程序 14.3 中，采用无序地址链的第二个好处就是每个链表均像堆栈那样工作，因此我们可以轻易地删除最近插入的元素，这种元素在每个地址链的开始位置（见练习 14.21）。当我们实现一个有嵌套作用域的符号表时，比如在一个编译器中，这种操作就显得格外重要。

在前面的几种符号表实现中，我们实际上无意中给了类成员一个处理重复关键字的选择。一个如程序 12.11 的类可以在插入前通过搜索来检查是否存在重复的关键字。另外一个类可以不搜索链表中是否有重复元素从而获得快速的插入操作。

一般地，当应用中需要排序与选择等符号表操作时，散列并不十分合适。然而，散列常用在如下的典型情况下：我们需要对一个符号表进行大量的搜索、插入和删除操作，并在最后按关键字的顺序一次打印输出。编译器中的符号表可作为这种情形的一个例子；另一个例子是删除重复关键字的程序，如程序 12.11。若对无序地址链完成上述操作，我们不得不采用第六章至第十章介绍的某种排序方法；对有序链表的实现，我们可以采用归并排序达到排序时间与 $N\lg M$ 成比例（见练习 14.23）。

## 练习

▷ 14.16 分别采用有序和无序的地址链，请计算出最坏情况下插入 $N$ 个关键字到空的散列表中需要多少时间。

▷ 14.17 用散列函数 $11k \bmod M$ 将第 $k$ 个字母转化为表地址，采用无序地址链，将关键字 E A S Y Q U T I O N 按顺序插入 $M$=5 的空散列表中，请给出插入后的结果。

▷ 14.18 采用有序地址链，完成练习 14.17，并请回答：你的结果是否因你插入元素的顺序改变而改变？

○ 14.19 编写一个程序，对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，采用链地址法将 $N$ 个随机整数插入表长为 $N/100$ 的散列表中，然后给出最长和最短的地址链的长度。

14.20 请删去程序 14.3 中的地址链头，把符号表表示为一个节点的数组（每个地址链的入口就是它的第一个节点）。

14.21 按下列要求修改程序 14.3：对每个元素增加一个整型字段，然后用一个函数实现删除该字段大于给定整数 $N$ 的所有元素。

14.22 请修改程序 14.3 的搜索算法，显示出关键字等于某一给定关键字的所有元素，显示方式与程序 14.3 的方式相同。

14.23 采用有序地址链，实现一个符号表（固定表长为 97），其中包括一个析构函数、一个复制构造函数和一个重载赋值操作符，要求该算法支持对第一类符号表的构造、统计、搜索、插入、删除、联接、选择和排序等操作，并且支持类成员编号（见练习 12.6）。

## 14.3 线性探测

如果我们能预知将要填入散列表的元素数目并且我们有足够的内存空间以容纳所有的关键字，那么我们在散列表中采用链地址法是不值得的。人们设计了几种方法，对表长 $M$ 大于元素数目 $N$ 的情况，依靠空的存储空间来解决冲突问题。这类方法被称为开放定址法。

最简单的开放定址法是线性探测：当冲突发生时（即我们准备把元素插入的位置已经被另一个不同的元素所占据），我们检查表中的下一个位置。按惯例将这种检查（以决定给定的表位置上是否存在一个于搜索关键字不同的元素）称为探测。线性探测的特点在于每次探测有三种可能的输出结果：如果表位置上含有与搜索关键字匹配的元素，那么搜索命中；如果表位置上为空，那么搜索失败；否则（如果表位置上的元素与搜索关键字不匹配）我们继续探测表的更高地址直到出现上述两种结果之一（如果搜索到表尾则要回到表头继续搜索）。如果在搜索失败后要插入一个含有搜索关键字的元素，我们将其放入使搜索结束的空位置上。程序 14.4 就是用线性探测实现的符号表这种抽象数据类型的。采用线性探测建散列表的过程如图 14.7 所示。

与链地址法一样，开放定址法的性能依赖于负载因子 $\alpha=N/M$（$\alpha$ 表示散列表的填满程度——译者注），但我们将有不同的解释。对链地址法来说，$\alpha$ 是每个地址链的平均元素个数而且大于 1。而对开放定址法，$\alpha$ 表示表中位置被占据的百分比，它一定是小于 1 的。因此，我们通常把 $\alpha$ 称为负载因子。

对稀疏表（$\alpha$ 很小），我们期望大多数搜索只需几次探测就能找到表的空位。而对于一个接近满的表（$\alpha$ 接近 1），一次搜索将需要相当多次的探测，而且当表完全填满时探测将

变成死循环。有代表性的做法是在线性探测中，为了避免过长的搜索时间，我们不允许表达到半满状态。也就是说额外空间与其用来构造地址链，不如用来缩短探测序列。采用线性探测的散列表长大于采用链地址法的散列表，因此，必须使 $M>N$。但由于未使用链表，总的存储空间会有所减少。在 14.5 节我们将详细比较两者空间使用的情况。现在让我们先来研究线性探测的时间开销作为一个 $\alpha$ 的函数的性质。

在线性探测中，多个元素聚合在一段连续的非空表空间中的现象称为“聚集”(cluster)。线性探测法的时间消耗依赖于元素被插入时的聚集方式。试考虑对半满表线性探测的两种极端情况：最佳情况是偶数地址的表位置是空的，奇数地址的表位置已有元素；最坏的情况是表的后半段填有元素而前半段是空的。两种情况下聚集的平均长度均为 $N/2N=1/2$，但两者对搜索失败的平均探测次数是不同的。前者为 1 加

$$(0+1+0+1+\ldots)/(2N)=1/2$$

**程序 14.4 线性探测**

将符号表元素存储在表长为元素数目期望值两倍的链表中，这样来实现符号表。将元素数目的期望值初始化为 nullItem。链表保存了全部元素，如果元素过大（超过机器字长——译者注），我们可以适当转化其类型使得可以保存对应于元素的链。

要插入一个新元素，我们散列得到一个表地址并探测其右边是否为空。与我们在关键字索引的搜索算法（程序 12.4）中所做的完全一样，用一个空元素作为空位置的标记。如果要搜索一个给定关键字的元素，我们可以把链表指针移到该关键字散列的位置，比较是否匹配，当命中空位置时就停止搜索。

我们总是希望链表不要达到半满状态，并据此为 M 赋值。如果散列函数产生近似随机的散列值，那么其他操作只需要几次探测就可以完成。

```
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;
  }
```

后者为 1 加

$$(N+(N-1)+(N-2)+\ldots)/(2N)=N/4$$

（我们认为所有搜索至少进行一次探测）。

概括地说，搜索失败所需平均探测次数与聚集长度的平方成比例。我们一次从链表中的一个位置开始搜索，共进行 $M$ 次（即起始位置每次不同——译者注）失败的搜索，将每次搜索的探测次数相加再除以总数 $M$，这样就计算出平均值。每次搜索失败至少进行一次探测，因此我们在第一次后统计探测次数。如果聚集长度为 t，则表达式

$$(t+(t-1)+\ldots+2+1)/M=t(t+1)/(2M)$$

表示了长度为 $t$ 的聚集对最终结果的贡献。若聚集长度总和是 $N$，则将表中所有聚集对于的探测数目相加，可得一次搜索失败平均时间开销为 $1+N/(2M)$加所有聚集长度的平方和再除以 $2M$。给定一个链表，我们就能很快地计算该链表失败搜索的平均开销（见练习 14.28），但是在线性探测算法中，聚集是一个复杂的难以解析的动态过程。

**性质 14.3**　*采用线性探测解决散列冲突的时候，在链表长为 $M$ 且链表元素个数为 $N=\alpha M$ 的散列表中，搜索命中和失败所需平均探测次数分别为*

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \text{ 和 } \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

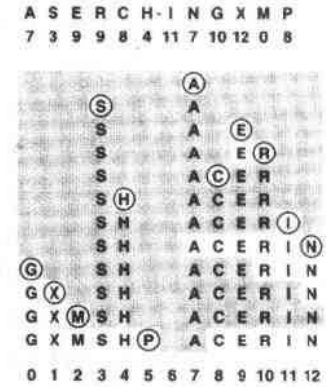



**图 14.7　采用线性探测的散列算法**

图中显示了将关键字 A S E R C H I N G X M P 插入长度为 13 的空散列表的过程。采用的是开放定址法，用线性探测处理冲突。首先，A 插入位置 7，然后 S 插到 3，E 插到 9，R 在 9 出发生冲突后插到 10，以此类推，探测到表右端时将返回到左端；例如，最后插入的关键字 P，散列到位置 8，而 8 到 12 都会冲突，故返回左端 0，才继续探测到 5 才结束。图中所有没有被探测的位置用圆圈表示。

尽管结果的形式相当简单，但对线性探测的精确分析确是一个极具挑战性的工作。Knuth 在 1962 年完成的工作是算法分析发展史上的一个里程碑（参见“第四部分参考资料”）。■

$\alpha$ 越接近 1，上面的估计越不准确，但由于我们通常不会对接近满的表也采用线性探测法，所以根本不用考虑$\alpha$ 接近 1 的情况。下面的表格概括了线性探测的散列表中搜索命中和失败所对应探测次数的期望值。

| 负载因子（$\alpha$） | 1/2 | 2/3 | 3/4 | 9/10 |
|---|---|---|---|---|
| 搜索命中 | 1.5 | 2.0 | 3.0 | 5.5 |
| 搜索失败 | 2.5 | 5.0 | 8.5 | 55.5 |

搜索失败的开销往往大于搜索命中，但在尚未达到半满的散列表中，两者的平均值都仅是几次探测而已。

正如我们在链地址法中做的那样，我们让客户程序来选择表中是否保留重复的关键字。在线性探测散列表中，重复元素不一定出现在连续的位置上——其他有相同散列值的元素可以出现在重复元素中间。

由建表过程的性质决定了关键字在线性探测散列表中时是随机排列的。抽象数据表的排序和选择操作需要从 0 点开始并采用第六章至第十章里介绍的某种方法。所以在排序和选择操作频繁时，应用线性探测法并不合适。

我们如何在线性探测散列表中删除一个关键字呢？仅仅移走它是不行的，因为后面将插入的元素会跳过被移走的元素留下的空位，因此插入操作所进行的探测应该在被删除元素的左边空位停止。这种问题的一个解决办法是把被删除元素所在位置到其右边下一个空位置之间的全部元素重新散列。图 14.8 说明了这一过程。程序 14.5 则是这种方法的一种实现。在稀疏表中，这只需要几次重新散列。另一种实现元素删除操作的方法是把被删关键字替换为一个标志关键字，这种标志可为搜索占据该位置而且可以被插入操作识别和使用。

**图 14.8 线性探测散列表的删除操作**

图示为把图 14.7 散列表中的元素 X 删除的过程。图中第二行为仅仅将 X 移走的结果，由于 M 和 P 的散列位置被 X 移走后的空位隔断了，故这种方法是不能采用的。因此，我们要重新按顺序插入 M、S、H 和 P（即与 X 在同一聚集内的关键字）。采用图中上部给出的散列值，并用线性探测法解决了冲突。这样，M 填上了 X 的空位，S 和 H 无冲突地插入表中，然后 P 插入位置 2。

## 练习

> 14.24 在最坏情况下，采用线性探测法把 $N$ 个关键字插入到一个空散列表中，需要多少时间开销？

> 14.25 将关键字 E A S Y Q U T I O N 按顺序插入表长 $M$=16 的空表中，采用线性探测法，用散列函数 $11k \bmod M$ 将第 $k$ 个关键字转化成表地址，请给出散列后的结果。

14.26 令 $M$=10 重做练习 14.25。

○ 14.27 请编写一个程序，采用线性探测，将 $10^5$ 个小于 $10^6$ 的随机非负整数插入表长为 $10^5$ 的散列表中，并绘图显示每 $10^3$ 个练习插入操作所用的总探测次数。

14.28 对 $N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 分别编写程序，采用线性探测，将 $N/2$ 个随机整数插入到表长为 $N$ 的散列表中，然后计算一次搜索失败所需的平均探测次数。

14.29 对 $N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 分别编写程序，采用线性探测，将 $N/2$ 个随机整数插入到表长为 $N$ 的散列表中，然后计算一次搜索命中所需的平均探测次数。请不要在表尾搜索所有的关键字（注意建表的开销）。

● 14.30　请做实验确定：在一个表长为 $2N$，有 $N$ 个关键字（分别令 $N$=10，100 和 1000）的散列表中，采用程序 14.4 和 14.5 的算法来进行插入和删除随机交替的多次操作，随着操作次数由 $N$ 升至 $N^2$，搜索命中与失败的平均开销是否会变化？

**程序 14.5　线性探测散列表的删除操作**

删除一个与给定关键字相匹配的表中元素，我们需要搜索该元素并用 nullItem 来代替之。然后，我们要判定删除后的空位右边的元素应散列在原位还是会左移，这是因为删除留下的空位会导致搜索探测的中止。因此，我们需要被删除元素所在聚集中的其他所有元素。由于散列表未达到半满状态，平均来说需要重新散列的元素数目应该较小。

```
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); }
  }
```

## 14.4　双重散列

对线性探测算法（事实上对任何散列算法）的一个重要原则是当我们搜索某一特定关键字的时候，我们要保证探测散列到同一地址的所有关键字（特别是，当该关键字在表中时就探测到它本身）。在开放定址散列算法中，特别是当表快要填满的时候，其他关键字也要被检查。在图 14.7 描述的例子中，搜索 N 需要探测 C、E、R 和 I，但它们并没有相同的散列值。还有更坏的情况，插入一个具有某一散列值的关键字会大大增加其他散列值关键字的搜索时间，例如在图 14.7 中，插入 M 导致搜索位置 7~12 和 0~1 的时间增加。由于跟聚集形成的过程有关，这种现象被称为聚类（clustering）。它使得对接近满的散列表的线性探测操作运行速度很慢。

幸运的是有一种简单的方法可以实际上消除聚类问题，这就是双重散列算法。其基本策略与线性探测算法一样，惟一不同在于它不是检查冲突点后面每一个表位置，而是采用第二个散列函数得到一个固定增量用于探测序列。程序 14.6 给出了实现方法。

第二个散列函数必须精心选择，否则程序可能会根本不起作用。首先，我们必须排除第二个散列函数产生散列值 0 的情况，因为 0 散列值将导致每次冲突时出现死循环。其次，第二个散列函数产生的散列值必须与表长互素，否则某些探测序列将会非常短（不妨以表长为第二个散列值的两倍为例）。考虑到这些情况，一种实现方法是选择 M 为素数，

再选择第二个散列函数使其返回值小于 M。在实际应用中，可采用下面的一个简单的散列函数：

```
inline int hashtwo(Key v) { return (v % 97) + 1; }
```

当表长并不小的时候，该散列函数可以满足多种散列要求。而且在应用中，这种简化带来的效率下降并不十分明显。如果散列表大而且稀疏，则表长不必为素数，因为每次搜索只需少数几次探测（要走这条捷径，就要对过长时间的搜索进行检测以防止死循环，见练习 14.38）。

**程序 14.6 双重散列**

我们采用第二散列函数以决定冲突发生后探测步长的增量，除此以外，双重散列与线性探测算法是相同的。这个步长不能为 0 而且要与表长互素。线性探测算法的删除函数（见程序 14.5）不能用于双重散列，因为任一关键字会落到不同的探测序列中。

```
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;
  }
```

图 14.9 给出了采用双重散列法建立一个小的散列表的过程。图 14.10 显示出双重散列算法的结果是比线性探测更少的聚集（因而聚集的长度也更短）。

**性质 14.4** *采用双重散列算法处理冲突的时候，对表长为 $M$，元素个数为 $N=\alpha M$ 的散列表，搜索命中与失败所需平均探测次数分别为*

$$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right) \quad 和 \quad \frac{1}{1-\alpha}$$

这两个公式是由 Guibas 和 Szemeredi 通过深入的数学分析得出的结论（参见“第四部分参考资料”）。这个结论证明了当我们采用这样的探测序列——即各个关键字不相关且每次探测等概率地命中每一个表位置，那么双重散列算法与复杂的随机散列算法是等效的。上述算法其实仅仅是双重散列的一个近似。例如，我们采用双重散列算法时尽量保证在每个表位置上只比较一次，但随机散列对每个表位置会比较多次。当然，对稀疏表而言，

两种方法产生冲突的概率是一样的。由于双重散列算法易于实现而随机散列算法便于分析，所以我们对两种方法都感兴趣。

随机散列算法中一次搜索失败的平均开销由下式给出

$$1+\frac{N}{M}+\left(\frac{N}{M}\right)^2+\left(\frac{N}{M}\right)^3+\ldots=\frac{1}{1-(N/M)}=\frac{1}{1-\alpha}$$

上式左边的表达式代表一次搜索失败所需探测次数大于 $k$（$k$=0, 1, 2,...）的概率之和（根据基本概率理论，它与平均值是相等的）。一次搜索总要用到至少一次探测，再需一次探测的概率为 $N/M$，需要第三次探测的概率为$(N/M)^2$，以此类推。我们可以用该公式近似计算 $N$ 个关键字的散列表中平均一次搜索命中的概率

$$\frac{1}{N}\left(1+\frac{1}{1-(1/M)}+\frac{1}{1-(2/M)}+\cdots+\frac{1}{1-((N-1)/M)}\right)$$

每个关键字等概率地被命中；搜索与插入一个关键字的开销是相同的；将第 $j$ 个关键字插入表中与从 $j$-1 个关键字中搜索失败的开销也是相同的。因此，这个公式代表平均开销。我们可以化简该公式，将分子与分母同乘以 $M$：

$$\frac{1}{N}\left(1+\frac{M}{M-1}+\frac{M}{M-2}+\cdots+\frac{M}{M-N+1}\right)$$

并根据 $H_M\approx\ln M$，可得

$$\frac{M}{N}(H_M-H_{M-N})\approx\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)\ \blacksquare$$

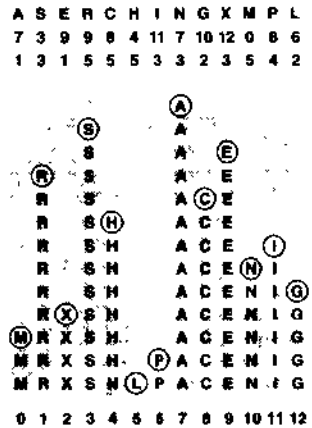



**图 14.9　双重散列**

图中显示了采用双重散列处理冲突的算法，用开放定址法把 A S E R C H I N G X M P L 插入空散列表的过程，其中散列值在图的顶端给出。图中，第一和第二散列函数的输出散列值分别列在关键字下面两行。与图 14.7 一样，被探测的表位置没有用圆圈表示。A 填入位置 7，S 填入 3，E 填入 9，这些均与图 14.7 相同，但当 R 在位置 9 发生冲突后被填入位置 1，这是因为采用第二散列函数算出的步长增量为 5。类似地，P 在 8，12，3，7，11 和 2 处发生冲突后最终被填入 6，其原因是采用第二散列值 4 作为步长增量。

Guibas 和 Szemeredi 证明的双重散列算法与理想的随机散列算法两者性能之间的关系是一种渐进关系，并且与实际的表长无关，其结论是建立在散列函数的返回值为随机的假设之上的。在实际应用中，即使我们使用极易计算的第二散列函数，如（v % 97）+1，性质 14.5 中的渐进公式仍可作为双重散列算法性能的精确估计。与线性探测算法中的相关公式一样，当$\alpha$趋向 1 时，公式的结果以慢于$\alpha$变化的速度趋向无穷。

图 14.11 清楚地对比了线性探测法和双重散列法。对稀疏表，两者性能相当，随着表不断填满，线性探测法的性能下降比双重散列法快很多。下面的表格给出了双重散列法搜索命中与失败所需探测数目的期望值：

| 负载因子（$\alpha$） | 1/2 | 2/3 | 3/4 | 9/10 |
|---|---|---|---|---|
| 搜索命中 | 1.4 | 1.6 | 1.8 | 2.6 |
| 搜索失败 | 1.5 | 2.0 | 3.0 | 5.5 |

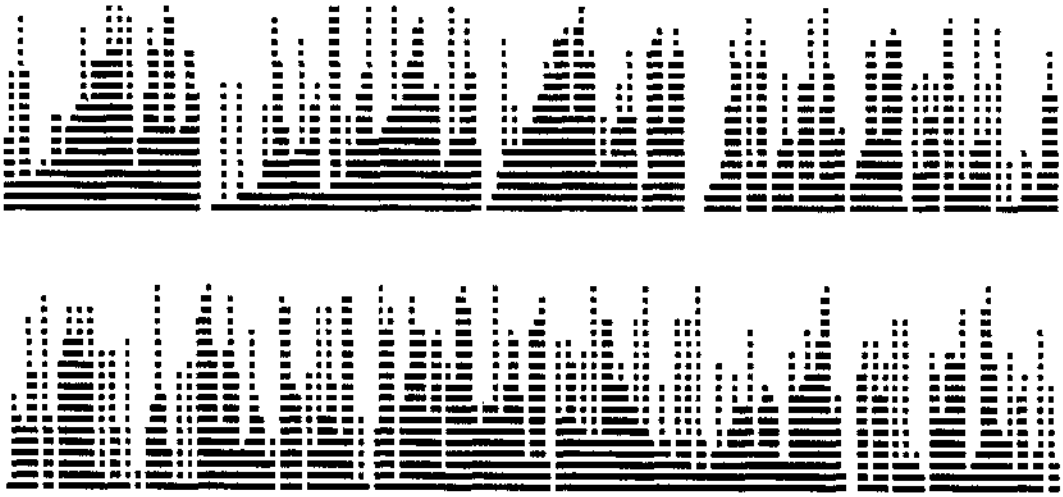


图 14.10 聚类

图中顶部列出了使用的关键字的散列值，中间与底部为分别采用线性探测法和双重散列法将关键字插入散列表后它们的分布情况。每行显示插入 10 个记录的结果。随着插入的进行，各记录聚合成被空位分隔开的聚类。由于搜索开销与聚类长度成比例，长的聚类是我们所不希望的。在线性探测算法中，随着记录的不断插入，长的聚类趋于更长，故少数几个长聚类将其决定作用。在双重散列算法中，这种作用并不明显，而且聚类相对较短。

搜索失败往往比搜索命中开销大，然而即使当表达到 9/10 满的时候，两者的平均开销也仅为几次探测。

让我们换个角度来讨论这一结果，即相同的平均搜索时间条件下，双重散列算法的表长小于线性探测算法的表长。

性质 14.5 *若我们使负载因子对线性探测算法和双重散列算法分别小于* $1-1/\sqrt{t}$ *和* $1-1/t$，*则可以保证平均搜索开销小于* $t$ *次探测。*

设性质 14.4 和 14.5 中的搜索失败的开销为 $t$，即可求出 $\alpha$。■

例如，为了保证一次搜索平均探测数目小于 10 次，若采用双重散列算法就要求保持表 10%为空，采用线性探测算法则要求表 32%为空。如果我们需要处理 $10^5$ 个元素，为了保证搜索失败的探测数目在 10 次以内，则需要另外 $10^4$ 个空间。与之相比，链地址法需要超过 $10^5$ 个链，BST 法更需两倍之多。

程序 14.5 实现删除操作的算法（即把被删除元素所在的聚集上的其他元素都重新散列）影响了双重散列算法的性能，因为被删除关键字可能出现在整个表的任何位置上。因此，我们需要求助于 12.3 节最后讨论的一种方法，即把被删除元素用一个标志来取代，表示该位置被占用但不与任何关键字相匹配（见练习 14.33）。

与线性探测算法类似，对于需要排序与选择操作的全功能的符号表，双重散列算法也不是合适的方案。

**练习**

▷ 14.31 采用双重散列算法，将关键字 E A S Y Q U T I O N 按顺序插入表长为 16 的空散列表中，请给出散列结果。初始散列函数为 $11k \bmod M$，第二散列函数为 $(k \bmod 3)+1$（其中 $k$ 表示第 $k$ 个字母）。

▷ 14.32 令 $M$=10，完成练习 14.31。

14.33 采用置空标志的方法实现双重散列算法的删除操作。

14.34 用双重散列算法完成练习 14.27。

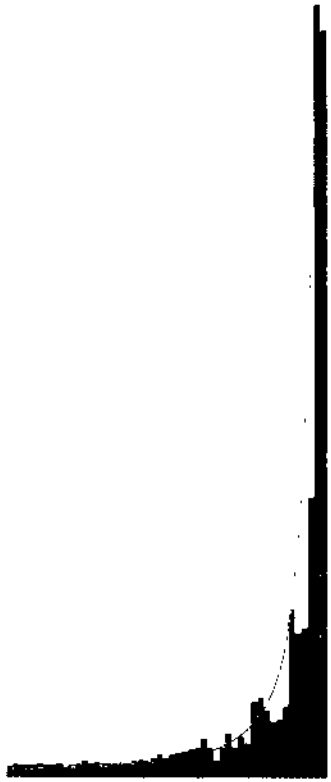
14.35 用双重散列算法完成练习 14.28。

14.36 用双重散列算法完成练习 14.29。

○ 14.37 采用嵌入式随机整数发生器（见程序 14.2）产生的随机种子作关键字，实现一个近似的随机散列算法。

14.38 设一个长度为 $10^6$ 的半满表，且元素在表中的位置是随机选择的，请估计表地址中可被 100 整除的位置被占据的概率。

▷ 14.39 假设你的双重散列算法代码中有一个错误，使得一个或两个散列函数总是返回同一个值（非零）。请回答在下列情况下各是出现了什么问题：（1）第一个散列函数出错；（2）第二个散列函数出错；（3）两个都出错。



## 14.5 动态散列表

随着散列表中关键字数目增多，搜索的性能不断下降。采用链地址法时，搜索时间逐步增大——当关键字数目加倍时，搜索时间也加倍。采用线性探测或双重散列等开放地址法时，结论也一样。但当表接近满时开销迅速增大，而且将达到一个任何关键字都不能再被插入的临界点。这种情况与树状搜索相反，当树节点数目加倍时，搜索开销仅有少量增长（对一次比较而言）。

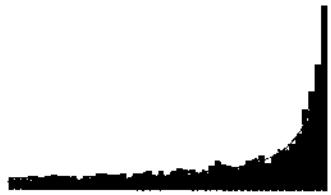


**图 14.11 开放定址算法的开销**

上下两图分别显示了采用线性探测算法和双重散列算法将关键字插入一个表长为 1000 的空散列表的开销。每条柱代表插入 20 个关键字的开销，灰色的曲线表示理论值（见性质 14.4 和 14.5）。

随着表的填满，加倍表长的方法可以使表元素继续增加。由于所有元素都要重新插入表中，故加倍表长的操作开销很大，但这种操作的频率并不高。程序 14.7 是加倍操作的一种实现，采用的是线性探测算法。图 14.12 给出了一个例子。该方法对双重散列算法同样适用。若应用于链地址法，其基本思想也是一致的（见练习 14.46）。每当散列表接近半满时，我们使表长加倍。第一次扩展后，散列表介于 1/4 满于半满两状态之间，因此平

均搜索开销小于 3 次探测。而且，尽管加倍表长后的重新散列开销大，但由于其使用频度低，故这部分开销只占建表总开销的很小一部分。

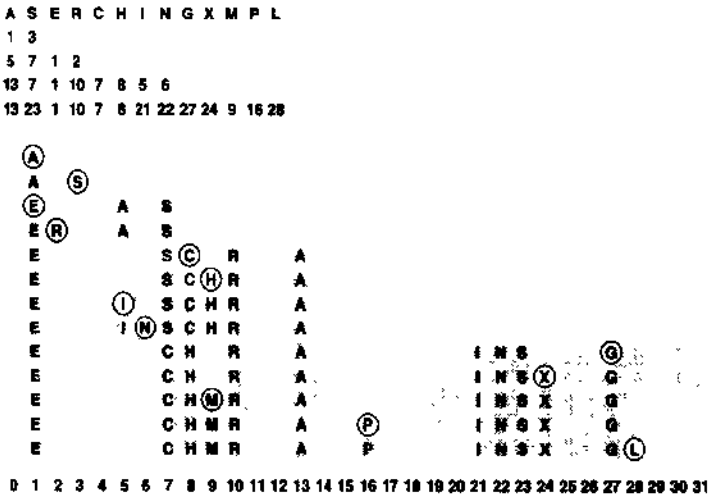



图 14.12 动态散列表的扩张

图上第一行给出了关键字值，采用线性探测算法处理冲突，本图显示了将关键字 A S E R C H I N G X M P L 插入表长动态加倍的散列表的过程。关键字值下面 4 行分别给出了表长为 4、8、16 和 32 时关键字的散列值。表长从 4 开始，当插入 E 时表长加倍，插入 C 和 G 时表长分别再加倍，最终表长为 32。每次表长加倍，所有关键字均重新散列和插入。这样，所有的插入操作都是在表处于稀疏状态下进行的，因此冲突很少（我们可以把表的稀疏定义为表不超过 1/4 满的状态）。

**程序 14.7 动态散列表的插入（应用于线性探测算法）**

这里将任意个数的关键字采用线性探测算法插入散列表的思路是：每当表接近半满时，便把表长加倍（该思路同样可用于双重散列算法和链地址法）。表长加倍的操作需要下面三步：为新表分配内存，将全部关键字重新散列到新表中，释放旧表所占内存。我们使用子函数 init()来建立或重建一个固定长度的表，其中的元素均为空。实现方法与程序 14.4 中的 ST 构造函数一样，故这里我们将这段代码省略了。

```
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();
  }
```

从另一角度看，每次插入操作的平均时间开销小于 4 次探测。这并不代表平均每次插入操作只需小于 4 次的探测；事实上，正如我们所知的，那些导致表长加倍的插入操作需要的探测次数相当多。这个结论也可作为“缓冲分析”的一个例证，即我们无法保证这个算法对每一种操作均是快速的，但我们可以保证在平均意义上每个操作的开销都是低的。

尽管总开销低，插入操作的性能曲线仍是不稳定的：多数操作非常快，但极少数几个操作耗时几乎相当于前面那些插入操作开销的总和。在表元素由一千增到一百万的过程中，这种情形（即开销大的插入操作——译者注）大约会出现 10 次。在很多应用中，这种现象时允许的，但如果对性能要求苛刻或要保证绝对性能的情况下，就不允许出现这种现象了。我们不妨以例子来说明。或许一个银行或航空公司愿意接受并承担让一个客户在百万次交易中等待大约 10 次的后果，但他们决不会接受在线资金交易或航班控制系统中的长时间等待，因为那将带来灾难性的后果。

如果我们支持抽象数据表的删除操作，随着表元素的减少，将表长减半就可以达到使表缩短的目的（见练习 14.44）。但有一个附加条件，表长减半与加倍所对应的元素个数的门限值是不同的。否则，即使很大的表，几次连续的插入与删除操作也会导致多次复杂的表长加倍和减半的操作。

**性质 14.6** *对符号表进行一系列 t 次搜索、插入和删除操作，所用时间与 t 成比例，且空间开销在元素个数的某个常数因子的范围内。*

当插入操作使散列表达到半满状态时，我们采用线性探测算法使表长加倍，当删除操作使表降为 1/8 满时，我们将表长减半以使表收缩。在上述两种情况下，若重建后的表长为 $N$，则表含有 $N/4$ 个元素，那么在表长再加倍（可通过 $N/2$ 个元素重新插入长度为 $2N$ 的表中来完成）之前可以插入 $N/4$ 个元素；同样，在表长再减半（可以通过 $N/8$ 个元素重新插入长度为 $N/2$ 的表中来完成）之前，可以删除 $N/8$ 个元素。在两种情况下，需要重新插入的元素个数均不超过使表达到重建点所做的操作（指插入和删除——译者注）数目的两倍，因此，总开销是线性变化的。而且，散列表总是介于 1/8 与 1/4 满的状态之间（见图 14.13），这也说明了性质 14.4 中的每个操作所需的探测次数小于 3。■

在使用模式无法预知的通用库符号表应用中，上述方法是适用的，因为它能合理地处理各种表长的情况。其基本缺点在于表扩张和缩减时重新散列和内存分配的开销问题。当搜索操作居于支配地位时，采用稀疏表将获得极好的结果。在第十六章，我们会考虑另外一种不需要重新散列的方法，而且那种方法适用于大型表的外部搜索。

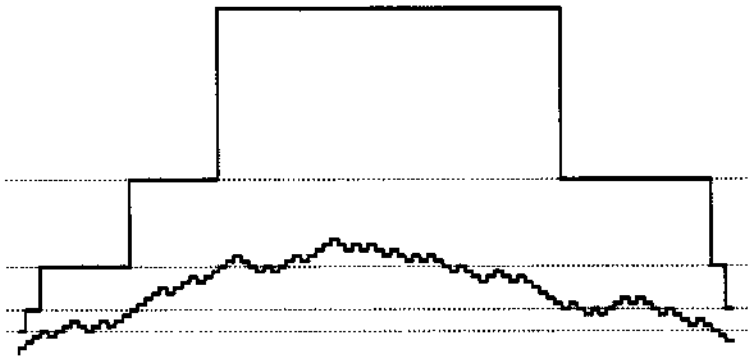


图 14.13 动态散列

图中最下一行表示表中元素个数，上面一行表示每次进行插入和删除操作时对应的表长，采用构造动态表的算法为：当插入至半满时表长加倍，当删除至 1/8 满时表长减半。初始表长为 4，且总保持为 2 的幂（图中虚线即 2 的幂）。按关键字数目变化的曲线轨迹每穿过虚线一次，表长相应改变一次（从下向上穿过为加倍，从上向下穿过为减半——译者注）。这样表的状态总是介于 1/8 满和半满之间。

### 练习

▷ 14.40 请用动态散列法（即表达到半满时把表长加倍）将关键字 E A S Y Q U T I O N 按顺序插入到空散列表中，初始表长为 $M$=4，用线性探测算法处理冲突，采用散列函数 $11k \bmod M$，其中 $k$ 表示第 $k$ 个字母。

14.41 请回答当表达到半满时将表长扩大为原来的 3 倍是否比加倍更经济？

14.42 请回答当表达到 1/3 满时将表长扩大为原来的 3 倍是否比达到半满时加倍更经济？

14.43 请回答当表达到 3/4 满时将表长加倍是否比达到半满时加倍更经济？

14.44 像程序 14.4 中那样，在程序 14.7 中加入一个删除程序，并在删除至表 1/8 满时把表长减半。

○ 14.45 请针对链地址法实现程序 14.7，要求每当地址链平均长度达到 10 时把表长增加 1/10。

14.46 采用简单删除操作（见练习 14.33）的双重散列算法来修改程序 14.7 和你完成的练习 14.44，在决定何时扩张和缩减表长时，请务必考虑你在删除时所用到的空元素和空位置。

14.47 采用线性探测算法实现一个动态符号表，要求包含一个析构函数、一个复制构造函数和一个重载赋值操作符，并支持建表、统计、搜索、插入、删除和联结等一级符号表 ADT 操作，并且支持客户句柄（见练习 12.6 和 12.7）。

## 14.6 综述

正如我们在前面介绍各种散列算法时所讨论的，针对某一特定应用选择最佳的散列算法要考虑多方面的因素。前面提到的所有散列算法都可以把符号表的插入和搜索操作的开

销减少至某一时间常数，而且这些算法对相当多的应用都是十分有效的。我们可以将三种主要算法（线性探测法、双重散列法和链地址法）粗略地总结如下：线性探测法是三者中最快的（前提是内存足够大以保证表是稀疏的），双重散列法使用内存最为有效（但需要额外的时间开销来计算第二散列函数），链地址法最易于实现（假设存在一种好的内存分配方案）。表 14.1 给出了关于几种算法性能的经验数据及其解释。

线性探测算法和双重散列算法的选择主要取决于计算散列函数的开销和表的负载因子。对稀疏表（$\alpha$ 小），两种方法都仅用几次探测，但双重散列算法遇到长关键字时的两次散列函数计算开销大。另一方面，如图 14.11 所示，随着$\alpha$ 接近于 1，双重散列算法性能大大优于线性探测算法。

由于需要精确计算内存使用情况，线性探测算法和双重散列算法与链地址法的比较更为复杂。链地址法需要额外的空间存储链表；开放定址法（*包括线性探测算法和双重散列算法——译者注*）需要在表内使用额外的空间来终止探测。下面以具体例子来说明之：假设我们使用链地址法建立了一个具有 $M$ 个地址链的散列表，链的平均长度为 4，并设每个表元素和链占一个机器字的空间。由于我们用相应的链来代替一定数量的元素，故认为元素与链占同样大空间的假设是合理的。在上述假设下，散列表共使用 $9M$ 个机器字的内存空间（$4M$ 用于元素，$5M$ 用于地址链），而且平均每次搜索需 2 次探测。然而，对 $4M$ 个元素且表长为 $9M$ 的散列表，线性探测算法完成一次搜索命中平均只需（1+1/(1−4/9)）/2=1.4 次探测，即在使用同样多内存的条件下比链地址法快 30%；另一方面，对 $4M$ 个元素且表长为 $6M$ 的散列表，线性探测算法完成一次搜索命中平均需要 2 次探测，即在同样时间开销条件下比链地址法节省 33%的存储空间。而且，我们还可以使用如程序 14.7 中的动态散列算法以确保随着元素数目的增多，表始终处以稀疏状态。

前面的讨论说明，基于性能考虑我们没有理由放弃开放定址法而选择链地址法。但实际应用中对固定表长的情况我们通常选用链地址法，理由有许多：易于实现（特别是删除操作）；对符号表或其他抽象数据类型需要使用的事先分配好链接字段的元素，实现起来只需很少的额外空间；尽管随着表的填满它的性能会下降，但在某种意义上这种下降是允许的，而且由于它仍比顺序搜索快，故通常不会影响应用的效果。

**表 14.1 实现散列表的实验研究**

对 32 位整数随机序列所做的符号表建表和搜索操作得到的数据，证明了对易于散列的关键字采用散列搜索比树状搜索算法搜索速度快。在稀疏散列表中，双重散列算法慢于线性探测算法和链地址法（这是由于计算第二散列函数的缘故），但随着表的填满它变得快于线性探测算法。而且双重散列算法是惟一可以仅使用很少的额外内存开销就可以实现快速搜索的算法。由于内存的重新分配和元素的重新散列，通过线性探测算法和表长加倍建立的动态散列表开销最大，但能达到最快的搜索速度。因此，它很适用于搜索起决定作用且关键字数目无法准确预知的应用。

| | 构造 | | | | | 搜索失败 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | 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 | 161 | 30 |
| 200000 | 407 | 84 | | | 159 | 186 | 156 | | | 33 |

其中：R 红黑 BST（程序 12.8 和 13.6）

H 链地址法（程序 14.3，采用表长 20000）

P 线性探测算法（程序 14.4，采用表长 200000）

D 双重散列算法（程序 14.6，采用表长 200000）

P* 表长加倍扩张的线性探测算法（程序 14.7）

针对某些特殊应用，人们提出了很多种散列算法，我们不能一一详述，在这里简要介绍三种特殊散列算法以试图说明其本质。

一类方法是在双重散列算法的插入操作时把元素来回移动，这样使得搜索命中率更高。Brent 研究出一种方法使得搜索命中的平均时间开销限制在一个时间常数之内（参见“第四部分参考资料”）。在搜索命中起决定作用的应用中，这种方法会相当有用。

另有一种被称为有序散列的算法，在线性探测算法中引入排序，使得搜索失败的开销接近于搜索成功。在使用标志位的线性探测算法中，当遇到一个空位置或与搜索关键字匹配的位置时，我们便停止搜索；在有序散列算法中，当我们遇到大于或等于搜索关键字的元素时就停止搜索（使用这种方法必须建立有序表）（参见“第四部分参考资料”）。引入排序所带来的改进与链地址法中对地址链排序所获得的效果是一样的。这种方法是为搜索失败处于决定地位的应用所设计的。

一个失败搜索速度快而命中搜索稍慢的符号表可以用来实现异常词典。例如，一个文本处理系统的断字算法对大多数单词正常工作，但遇到异常情况（如遇到单词“bizarre”）时不能正常工作。在一个大文档中只有很少几个单词有可能出现在异常词典中，即几乎所有的搜索都应该是失败的。

上述例子仅是多种对散列算法改进中的几种，很多算法是有趣的并且有重要的应用。要提醒大家注意的是：除非要求很高或者对性能/复杂度进行了认真的分析，否则不要轻易使用高级的算法，因为我们所认真讨论过的链地址法、线性探测法和双重散列法已经足够简单、高效并适用于很多应用了。

前面异常词典的例子告诉我们针对某一常用操作对算法做少量修改就可以提高其性能——在这里是针对搜索失败进行修改。再看一个例子，假设我们有一个 1000 个元素的异常词典，要搜索 1 百万个元素是否在其中，事实上几乎所有元素的搜索都是失败的。若元素为奇异（bizzarre）英文单词或 32 位随机整数，就会出现这种情况。一种方法是把所有词散列成 15 位的散列值（表长约 $2^{16}$）。1000 种异常值占表的 1/64 并使得绝大多数搜索立即以失败结束，即第一次探测就发现空的表位置。但如果表中含有 32 位的字，我们可以将它转化成奇异常表并采用 20 位的散列值。这样只需比较一位就可以完成搜索；而搜索命中则需要在一个小的表中多次进行探测。异常占据表的 1/1000；搜索失败仍占绝大多数；而且我们通过 1 百万次位比较操作就可以完成任务。该方法采用了这样一个基本思想：一个散列函数对每个关键字产生一个标志（certificate）——这一思想在符号表之外的应用中极为有用。

在符号表实现的许多应用中，散列比二叉树（见第十二、十三章）应用更为广泛，因为它简单并能提供最佳的搜索时间（为常数），而且如果关键字是标准类型或足够简单，我们定可以找到好的散列函数。二叉树优于散列的地方是它基于简单的抽象接口（无需设计散列函数）；二叉树是动态的（不需关于搜索次数等其他信息）；二叉树可以在最坏情形下保证性能（即使最好的散列算法也有可能将全部关键字散列到同一位置）；二叉树支持更多种操作（最重要的是排序与选择）。当上述因素不重要时散列算法当然是理想的选择，但还有一种例外：若关键字是长字符串，我们可以构造一种搜索起来比散列更快的数据结构，这将在第十五章中介绍。

## 练习

▷ 14.48 对 1 百万个整数关键字，若要求每个表的平均搜索失败的探测数目相同，请计算采用三种散列算法（线性探测算法、双重散列算法和链地址法）建立的散列表的表长，然后统计比较散列函数的计算结果。

▷ 14.49 对 1 百万个整数关键字，若要求每个散列表可使用 3 百万字的内存空间，请计算三种散列算法（线性探测算法、双重散列算法和链地址法）对一次搜索失败平均所需的探测数目。

14.50 用课文中介绍的适用于多数搜索为失败的快速算法实现一个抽象类型符号表，要求：对第二次比较采用链地址法。

# 第十五章

# 基 数 搜 索

第十章中我们介绍了基数排序。采用类似的方法我们可以得到一些搜索算法，称为基数搜索算法，即每一步不是比较整个关键字，而是比较关键字的一段或一部分。当关键字易于分段时这些算法便很有用，而且能有效地完成多种搜索任务。

这里我们采用与第十章相同的抽象模型，按内容不同可将关键字分为两种：一种是字，即固定长度的字节序列；另一种是字符串，或简称串，即变长的字节序列。对字类型的关键字，我们用基 $R$ 的数的形式表示，并按数位进行各种操作。我们可以把 C 语言的字符串看作终止于一个特殊符号的变长数，这样无论是定长关键字还是变长关键字我们都可以把算法基于“提取关键字的第 i 位”这一抽象的操作之上，当然要考虑到关键字小于 i 位的情况。

基数搜索算法最基本的优点在于：在最坏情况下仍能提供较好的搜索性能而无需平衡树那样的算法复杂度；而且，它对变长关键字易于实现；某些基数搜索算法在搜索数据结构内对部分关键字进行排序从而节省空间开销；与二叉树和散列算法相比，基数搜索可以快速访问数据。与基数排序一样，基数搜索也存在一定缺陷，比如空间使用效率低，若不能高效地访问关键字中的每一字节，搜索性能也会受影响。

在本章中，首先，我们介绍每次用关键字的一位来遍历二叉树的几种搜索算法。我们将连续讨论一系列方法，而且每一种都能克服前面一种的某个毛病，最终我们将得到一种适用于多种搜索应用的巧妙算法。

然后，我们将概括 R 叉树。同样，我们会介绍一系列算法并最终得到一种支持基本符号表及其扩展的灵活高效的算法。

在基数搜索中，我们通常先比较关键字的最高位（MSD）。很多算法与基于最高位的基数排序算法思路是一致的，正如基于 BST 的搜索算法与快速排序算法的思路一致。特别是，我们还要介绍与第十章中线性时间排序算法类似的方法——基于同一思想的常数时间搜索算法。

在本章最后，我们会介绍采用基数搜索算法的大文本索引应用。这是一种很自然的解决方案，而且为我们在第五部分介绍高级字符串处理做了必要的准备。

## 15.1　数字搜索树

最简单的基数搜索算法基于数字搜索树（DST）。其搜索和插入算法与二叉树搜索只有一点不同：我们给树分支不是根据整个关键字的比较结果，而是根据关键字某些被选位的比较结果。在第一层使用最重要的位，第二层使用次重要的位，以此类推，直到遇到外部节点。程序 15.1 给出了搜索算法，其对应的插入算法也是类似的。由于我们有位函数操作可以访问关键字每一独立的位，故不用“小于”进行关键字比较。事实上，这段代码与程序 12.8 中二叉树搜索算法的代码是一样的，但其性能截然不同，这些将在下面介绍。

如第十章所述，基数排序算法需要我们注意相等的关键字的情况，在基数搜索中亦然。在本章中，我们假设符号表中所有关键字值均不相同。由于我们可采用 12.1 节中的方法处理相同关键字的情况，故上述假设并不失一般性。在基数搜索中集中精力研究不等值关键字是非常重要的，因为关键字值是我们将要研究的几种数据结构中的共有元素。

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

**图 15.1　单一字符关键字的二进制表示**

与第十章一样，我们用 i 的 5 位二进制数表示字母表中第 i 个字母，图中给出了一组字母的表示，在本章中我们常用这组字母作例子，而且按从左到右的顺序将各位称为第 0 位至第 4 位。

**程序 15.1　二进制数字搜索树**

本程序的搜索算法显示了如何通过修改标准 BST 的搜索和插入算法（见程序 12.8）来实现采用 DST 的符号表。在算法中我们不是比较整个关键字以决定在树上左移还是右移，而是比较关键字的某一位（如前导位）。递归调用的第三个参数是在搜索过程中随着节点的下移用来移动关键字比较位的。采用 10.1 节的方法，对比较位进行位操作。同样，我们也可以修改程序 12.8 以得到插入操作的算法。

```
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); }
```

图 15.1 给出了本章常用的一些单字母关键字的二进制表示。图 15.2 给出了一个 DST 树插入操作的例子。图 15.3 显示了在空树中插入关键字的过程。插入和搜索操作是由关键

字的各位决定的，但任意 DST 不具有 BST 的有序性，也就是说 DST 中一个给定节点不一定大于其左边的节点且小于其右边的节点，这种有序性在每个关键字不等的 BST 中是一定存在的。然而，对一个给定节点，其左边的关键字一定小于其右边的关键字（假设该节点在第 $k$ 层，那么其前 $k$ 位是一样的，但下一位则左边的为 0，右边的为 1），而且该节点的关键字可能是其子树中所有关键字中的最大值、最小值或其中任意值。

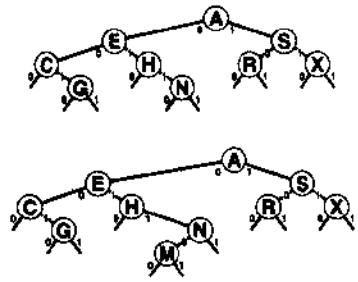



**图 15.2 DST 和插入操作**

图中上半部分以关键字 M=01101 的失败搜索为例。（由于关键字的二进制表示的第一位是 0）从根节点出发左移。（由于第二位是 1）然后右移，然后再右移、左移，最后到达 N 点下面的空链接。图中下半部分给出插入 M 的过程。我们在搜索的结束点将空链接接替换为一个新节点的链接，这与在 BST 中的插入方法是一致的。

DST 的特点是每个关键字是一条由关键字的位按从左到右顺序决定的路径。该性质保证了程序 15.1 可以正确实现搜索和插入操作。

假设关键字是定长（$w$ 位）的字，要使每个关键字不同则要求 $N \le 2^w$，不妨令 $N$ 远远小于 $2^w$（不满足此情况的话可采用 12.2 节中关键字索引的搜索算法）。事实上，很多实际搜索应用是满足该要求的。例如，对具有 $10^5$ 条记录且关键字长为 32 位但可能没有 $10^6$ 那么多记录的符号表或者任意多条记录数的 64 位关键字的符号表，DST 都是适用的。DST 同样适用于变长关键字。对于变长关键字的情况我们将在 15.2 节介绍，并介绍一些相关算法的变形。

若关键字数目很大而字长相对较小，则建立 DST 表的最坏情况比 BST 的要好很多。对很多应用而言（例如关键字是由随机的位组成的），最长路径的长度也是相对较短的。特别是，最长路径是由最长关键字的位数决定的；而且，若关键字是定长的，搜索时间也由关键字的位数决定，如图 15.4 所示。

**性质 15.1** *在一个 $N$ 个随机关键字的数字搜索树（DST）中，一次搜索或插入操作平均需要 $\lg N$ 次比较，最坏情况为大约 $2\lg N$ 次比较。比较次数不大于关键字的位数。*

下一节我们将介绍一个很自然的问题，现在先用其中的参数，对随机关键字给出最坏情况和平均情况的结果，将此留作练习（见练习 15.30）。出于简单的直觉，我们可以将随机关键字分为以 0 开头和以 1 开头两部分，且两部分各分到某节点的一侧。在树上每下移一层，需要比较关键字的一位，因此搜索所需的比较次数不会超过关键字的位数。对于 $w$ 位关键字且总数 $N$ 远远小于 $2^w$ 的典型情况，路径长度接近 $N$，所以比较次数也远远小于随机关键字的位数。■

图 15.5 给出了 7 位关键字的较大型的数字搜索树，它接近了完美的平衡。DST 的吸引人之处就在于对许多应用（甚至很大的数据量）都能提供近乎最佳的性能，且易于实现。例如，即使关键字数目达到数十亿，32 位关键字的 DST（或 4 个字母，每个字母用 8 位二进制表示）也可使比较次数小于 32，64 位关键字的 DST 可使比较次数小于 64。对于大的 $N$，DST 可提供与红黑树相当的比较次数，并且不超过 BST 的复杂度（BST 只提供与 $N^2$

成比例的复杂度）。所以，如果关键字的每一位都易于访问的话，那么数字搜索树（DST）是对搜索和插入等符号表操作使用平衡树的一种不错的选择。

### 练习

▷ 15.1　使用图 15.1 给出的二进制表示，请画出将关键字 E A S Y Q U T I O N 按顺序插入一棵空 DST 的结果。

15.2　请给出将关键字 A B C D E F G 插入平衡 DST 的过程（所得到的结果也是一个有效的 BST）。

15.3　将关键字 A B C D E F G 插入到平衡 DST 中，使每个节点小于其子树上的所有节点。

▷ 15.4　构造 DST，请画出结果。从一棵空树开始，所用关键字为：01010011，00000111，00100001，01010001，11101100，00100001，10010101，01001010。

15.5　请问我们能否像 BST 中那样在 DST 中保存相同的关键字，为什么？

15.6　对 $N=10^3,10^4,10^5$ 和 $10^6$，用 $N$ 个随机的 32 位关键字构造 DST，并与同样关键字的 BST 和红黑树进行树高和内部路径长度的比较。

○ 15.7　对 $N$ 个不同的 $w$ 位关键字的情况，请完整描述 DST 最坏情况下内部路径的长度。

● 15.8　对基于 DST 的符号表实现删除操作。

● 15.9　对基于 DST 的符号表实现选择操作。

○ 15.10　不建立 DST，请设计程序，计算给定关键字序列的 DST 高度，要求算法复杂度与序列长度成线性关系。

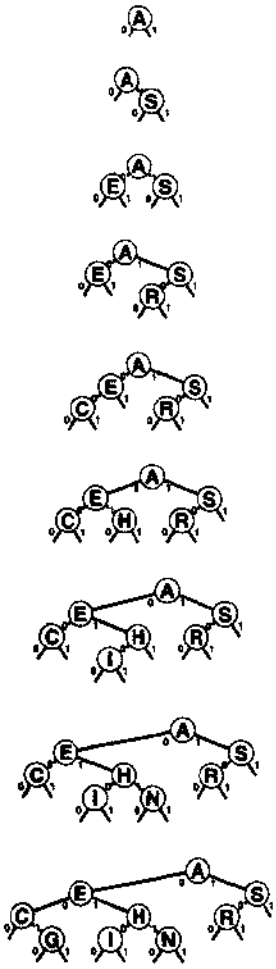


**图 15.3　DST 的建立**

本图给出了将关键字 A S E R C H I N G 插入一个空 DST 的过程和结果。

## 15.2　线索（trie）

在本节我们介绍与 DST 类似的、采用关键字的位比较来完成搜索的一种搜索树，但不同之处在于树中的关键字是有序排列的，并因此该树支持如 BST 中的排序等符号表操作。其基本思路是把关键字只存储在树的底端——即树叶节点上。这种数据结构有很多有用的性质，而且成为了很多有影响的搜索算法的基础。它是由 de la Briandais 在 1959 年首先发现的，而且由于它对检索（re*trie*val）操作非常有用，故在 1960 年被 Fredkin 命名为线索（trie）。有趣的是平时我们不得不把它读成“try-ee”或者干脆读“try”以区别树“tree”。为了命名的一致性，我们应该用术语“二叉搜索线索”来命名线索，但事实上线索这一名称已经被广泛接受了。我们在本节介绍基本的二叉搜索线索，在 15.3 节介绍

它的一种重要的变形，在 15.4 和 15.5 节介绍基本的多叉搜索线索和一系列变形。

线索对于定长与变长关键字的情况都适用。为了简化讨论，我们假定任一搜索关键字都不是其他关键字的前缀。例如，定长且各不相等的关键字满足此条件。

在一个线索中，我们把所有关键字保存在树叶节点上。大家可以参见 5.4 节，树叶是指一棵树上没有孩子的节点，树叶与外部节点是不同的，我们把外部节点看作是空孩子。在二叉树中，树叶是左右链接都为空的内部节点。把所有关键字保存在树叶而不是内部节点使得我们可以像在 DST 中那样（见 15.1 节）比较关键字的位来完成搜索。而且两者基本的一致性在于：若当前比较位是 0，则向下进入左子树比较；若当前比较位是 1，则向下进入右子树比较。

**定义 15.1** *线索是所有关键字存储在树叶的一种二叉树数据结构，其递归定义这样给出：没有任何关键字的线索是一个空链接；单关键字的线索是一个含有该关键字的树叶；多关键字的线索起始于一个内部节点，其左链接指向以 0 位开始的关键字，右链接指向以 1 位开始的关键字，然后对关键字的后续位依此规则继续构造子树，直至关键字最后一位。*

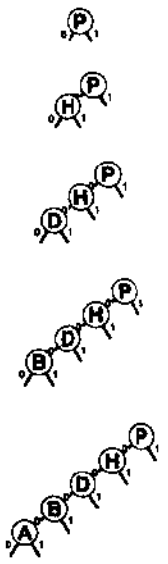



**图 15.4 最坏情况下的 DST**

图中给出了插入下列关键字到空 DST 中的结果，关键字为 P=10000，H=01000，D=00100，B=00010 和 A=00001。该序列很像退化树，但其路径长度由关键字的二进制表示的长度所决定。除了 00000，其他任何 5 位关键字都无法使树的高度增加。

所有关键字都被存储在树叶上，且其路径顺序是由关键字中前导位的排列模式决定的。相反，每一树叶存储的是由树根到该叶的路径所惟一确定的关键字。非树叶的空链接对应的是不属于线索中关键字的位序列。因此，要在线索中搜索一个关键字，我们只需像在 DST 中那样由树根起、根据关键字的不同位向下分支，但不需要在内部节点进行比较。搜索过程是这样的：我们从线索的顶端出发，先比较关键字的最左边一位，若它是 0 则进入左子树，若是 1 则进入右子树，然后将所用关键字向右移一位继续搜索。若搜索结束于一个空链接则表示搜索失败；由于只能有一个树叶含有与搜索关键字相同的元素，因此若搜索结束于一个树叶则只需一次关键字比较就可以完成搜索。程序 15.2 实现了上述过程。

**图 15.5 一个 DST 的例子**

这个通过插入大约 200 个随机关键字构成的 DST 是很平衡的。

**程序 15.2　线索的搜索**

与程序 15.1 中 DST 的搜索一样，该函数使用关键字的各位控制搜索的进程。有三种可能的结果：如果遇到一个树叶（其左右链接都是空链接），它将是线索中惟一对应关键字 v 的记录，因此我们比较该节点是包含 v（即搜索命中）还是其关键字的前面若干位与 v 匹配（即搜索失败）。如果遇到一个空链接，则其双亲的其他链接不能为空，说明线索中含有与搜索关键字的相应位不同的关键字，同时搜索失败。这里假设所有关键字各不相同，而且（如果关键字长度不同）任意关键字不会成为其他关键字的前缀。同时，item 成员没有用在非树叶节点上。

```
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); }
```

跟通常一样，要在一个线索中插入一个关键字，首先要进行搜索。如果搜索在一个空链接处结束，我们便将该空链接替换为指向含有被插关键字的树叶的链接。如果搜索在一个树叶处结束，我们需要增加内部节点，直到遇到与增加的节点的所有孩子的关键字有一位不同。图 15.6 给出了在线索搜索和插入的例子；图 15.7 给出了将关键字插入空线索构造线索的过程。程序 15.3 是线索插入算法的完整实现。

**程序 15.3　线索的插入**

将新节点插入到线索中，首先要进行搜索，并区分搜索失败中可能出现的两种情况，分别处理。

如果搜索失败不是出现在一个树叶上（即在一个空链接上），我们将遇到的空链接替换为指向新节点的链接（*该节点就是被插入元素——译者注*）。

如果搜索失败出现在一个树叶上，我们采用 split 函数插入新的内部节点，该节点表示与插入关键字第一个不同位的左边所有位。split 函数的 switch 语句用来处理两位转换后遇到的四种情况：如果两位相同（即 $00_2$=0 或者 $11_2$=3），则继续运行 split 函数；如果两位不同（即 $01_2$=1 或者 $10_2$=2），则跳出 split 函数。

```
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); }
```

我们不用访问树叶的空链接，也不把元素存储在非树叶节点中，故我们可以通过采用 union 或另外两种定义节点的类（见练习 15.20 和 15.21）节省空间。当前，我们仍为了简便起见采用 BST、DST 和其他二叉树数据结构中的单节点的定义方法，这种方法的显著特点为内部节点不含关键字而树叶节点的左右链接为空。当然，这种定义的简化导致了一定空间的浪费。在 15.3 节中，我们将介绍针对多节点类型对算法的改进，并且在第十六章中我们将介绍用 union 实现的搜索。

根据前面的定义和几个例子，我们可以得出几个线索的基本性质。

**性质 15.2** *一个线索的结构与其关键字插入的顺序是无关的：对任一关键字序列存在一个惟一的线索。*

这是线索区别于我们前面介绍的其他搜索树的一个特性：其他搜索树的建立既依赖于我们所使用的关键字，又依赖于关键字的插入顺序。■

一个线索的左子树的全部关键字都是以 0 开头，而右子树的关键字都是以 1 开头。线索的这个性质与基数排序有一定相似性：二叉线索搜索与二进制快速排序（见 10.2 节）对关键字的分类方式是完全一致的。通过比较图 15.6 的线索与图 10.4 的快速排序（注意：所

有关键字都不同）也可以得出两者的关系。我们在第十二章也介绍了两者的相似性。

线索的特别（与 DST 不同）之处在于其关键字在树中是有序的，因此我们可以直接实现符号表的排序和选择操作（见练习 15.17 和 15.18）。而且，线索与 DST 一样都是平衡树。

**性质 15.3** *对由 $N$ 个随机位组成的随机关键字进行一次插入或搜索操作平均需要 $\lg N$ 次位比较。最坏情况下的位比较次数为 $N$。*

在分析线索时要注意我们所做的假设，即所有关键字均不同，且没有一个关键字是其他关键字的前缀。满足这些假设的一个简单模型是我们从无穷的随机位构成的序列中取出我们建立线索所需的位。

下面我们通过概率分析得出平均情况下的结果。在一个随机线索中，$N$ 个关键字与一个搜索关键字的前 $t$ 位中至少有一位不同的概率为：

$$(1-\frac{1}{2^t})^N$$

用 1 减上式即得线索中存在一个关键字与搜索关键字前 $t$ 位匹配的概率：

$$1-(1-\frac{1}{2^t})^N$$

从另一角度解释，上式就是搜索所需比较位数大于 $t$ 位的概率。根据基本概率知识，对于 $t\geq 0$，随机变量大于 $t$ 的概率的和是随机变量的平均值，所以平均搜索开销为：

$$\sum_{t\geq 0}\left(1-(1-\frac{1}{2^t})^N\right)$$

根据近似式 $(1-1/x)^x\sim e^{-1}$，可得搜索开销近似等于：

$$\sum_{t\geq 0}\left(1-e^{-N/2^t}\right)$$

上面的求和式中，$\lg N$ 个 $2^t$ 充分小于 N 的项非常接近 1；所有 $2^t$ 充分大于 $N$ 的项非常接近 0；另外少数 $2^t\approx N$ 的项介于 0 和 1 之间。因此，求和结果大约为 $\lg N$。对上式的精确分析与估计需要高深的数学理论（参见“第四部分参考资料”）。我们分析的前提是假设 $w$ 足够大，使得我们在搜索时不会出现搜索关键字的所有位都用完的情况，但实际的 $w$ 值会使开销减小。

在最坏情况下，我们可能会遇到两个相当多位相同的关键字，当然这种情况的概率非

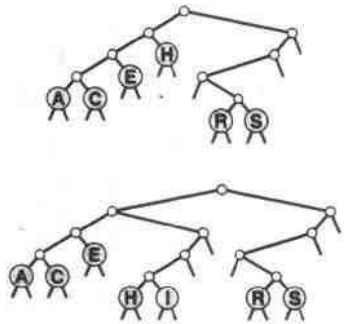



**图 15.6　线索的搜索和插入**

线索中的关键字都存储在树叶中（树叶定义为左右链接都为空的节点）；非树叶节点的空链接对应的关键字也不在线索的全部关键字中。

图中上半部分给出了一个线索的例子，在该线索中若要完成对关键字 H=01000 的成功搜索，我们由树根出发并且先左移（因为关键字 H 的二进制表示的第一位是 0），再右移（因为第二位是 1），即搜索到 H，显然 H 是该线索中惟一一个前两位是 01 的关键字。且该树中没有关键字以 101 或 11 开头，以这两种形式开头的关键字在本例中对应非树叶节点的空链接。

图中下半部分给出了在线索中插入 I 的过程：我们需要加入 3 个非树叶节点，一个对应 01 且含有一个对应 011 的空链接；一个对应 010 且含一个对应 0101 的空链接；还有一个对应 0100 且其左孩子为 H=01000，其右孩子就是要插入的 I=01001。

常小。性质 15.3 中提到的最坏情况与这个概率呈指数关系(见练习 15.29)，即更小。■

我们也可以将分析 BST 的一种方法（见性质 12.6）推广来分析线索的性质。即以 0 开头的关键字为 $k$ 个，另外以 1 开头的关键字为 $N-k$ 个的概率为 $\binom{N}{k}/2^N$，由此我们可以得出外部路径长度的递推公式：

$$C_N = N + \frac{1}{2^N}\sum_k \left(\binom{N}{k}(C_k + C_{N-k})\right)$$

这个递推公式与 7.2 节中快速排序的递推公式是类似的，但推导难度更大。而且这个结果是我们在性质 15.3 中介绍的平均搜索开销的表达式的 $N$ 倍（见练习 15.26）。仔细研究这个递推公式可以加深我们对线索比 BST 更平衡的理解，即从中间将关键字分类的概率更大。因此，这个递推公式与归并排序和快速排序的递推公式两者（前者大约等于 $N\lg N$，后者大约等于 $2N\lg N$）相比跟前者更相像。

我们已经考察了一些线索区别于其他搜索树的性质，但还有一个不好的性质是当关键字多数位相同时需要进行单分支遍历。如图 15.8 所示，无论线索中有几个关键字，只有最后一位不同的关键字就需要与关键字长相等的路径长度，此时内部节点的数目甚至比关键字的数目还多。

**性质 15.4** *一个由 $N$ 个 $w$ 位随机关键字构成的线索平均含有 $N/\ln 2 \approx 1.44N$ 个节点。*

通过整理性质 15.3 的公式，我们可以得到 $N$ 个关键字的一个线索中平均节点数目的公式如下（见练习 15.27）：

$$\sum_{t\geq 0}(2^t(1-(1-\frac{1}{2^t})^N) - N(1-\frac{1}{2^t})^{N-1})$$

因为这一公式中很多项的值都不像性质 15.3 中的公式中那样趋向 0 或 1，所以其数学推导比前者更为复杂（参见“第四部分参考资料”）。■

我们可以通过实际经验来检验上述结果。图 15.9 就给出了一个大的线索，它的节点数目比相同关键字构造的 BST 和 DST 多 44%，但更均衡并且达到近似最佳的搜索开销。我们起初会担心额外的节点会增加搜索操作的平均开销，但事实正相反——例如，即使我们把平衡线索的关键字数目加倍，

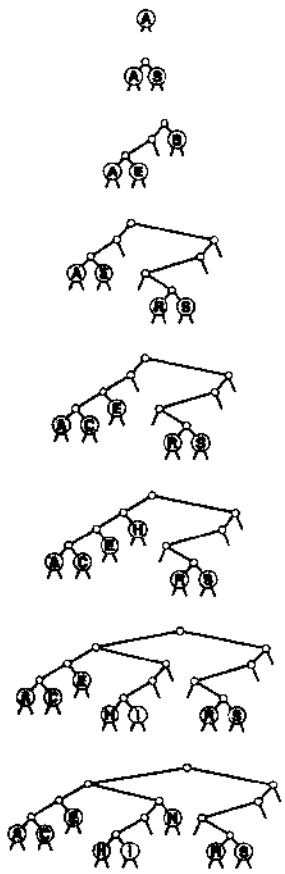


**图 15.7 线索的建立**

图中描述了将关键字 A、S、E、R、C、H、I、N 插入空线索的过程。

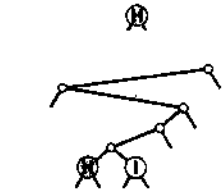


**图 15.8 二叉线索的最坏情况**

图中给出了将关键字 H=01000 和 I=01001 插入空二叉线索的结果。比较 DST 的类似情况（见图 15.4），路径的长度等于关键字的二进制表示的字长；由图可见，即使线索中只有两个关键字，路径的长度也比较长。

搜索操作增加的开销也仅仅为一次比较操作。

**图 15.9　一个线索的例子**

这里给出了一个由 200 个随机关键字构成的线索，它是平衡的。但由于单路分支的影响，其中不必要的节点占 44%（图中树叶上的空链接没画出）。

考虑到程序 15.2 和 15.3 实现起来的简便，我们假设关键字是定长且各不相同的，因此程序每次可以比较一位而且并不会用尽关键字的全部位。同样为了简便，在性质 15.2 和 15.3 中，我们假设每个关键字的位数是任意的，因此各关键字相同的概率（呈指数衰减）很小。这些假设直接适用于变长字符串关键字的情况，当然仍有细节值得注意。

如果对变长关键字的情况使用前面的程序，除了每个关键字都不同这一限制，还要增加一个条件，即每个关键字都不能成为任一关键字的前缀。很多应用会自动满足这个条件，我们将在 15.5 节介绍这部分内容。另外，因为我们对每个前缀的处理都与一定的内部节点相联系，所以我们可以在内部节点中存入一定信息以处理某些关键字成为其他关键字前缀的情况（见练习 15.31）。

对于由足够多的随机位组成的长关键字，性质 15.2 和 15.3 描述的平均情况仍成立。在最坏情况下，线索树的高度仍取决于最长关键字的位数。如果关键字非常长或者比较相似，则搜索的开销就会过大，比如这种情况会出现在字符编码的数据中。在接下来的两章中，我们将介绍对长关键字的应用能够减小开销的方法。一种方法是通过把单树枝压缩成单链接以缩短路径长度——我们将在 15.3 节介绍如何简洁高效地实现该方法。另一种方法是使每个节点包含多于 2 个链接以达到缩短路径长度的目的——这将在 15.4 节介绍。

## 练习

▷ 15.11　将关键字 E A S Y Q U T I O N 按顺序插入空线索中，请画出结果。

15.12　请回答：如果用程序 15.3 将一个线索中已经存在的元素再插入线索中会出现什么结果？

15.13　将关键字 01010011，00000111，00100001，01010001，11101100，00100001，10010101，01001010 按顺序插入空线索中，请画出结果。

15.14　对 $N=10^3,10^4,10^5$ 和 $10^6$，用 $N$ 个随机的 32 位关键字构造一个线索，并与同样关键字的 BST 和红黑树进行树高、节点数目和内部路径长度的比较（参见练习 15.6）。

15.15　对 $N$ 个不同的 $w$ 位关键字的情况，请完整描述线索在最坏情况下内部路径的长度。

● 15.16　对基于线索数据结构的符号表实现删除操作。

○ 15.17　对基于线索数据结构的符号表实现选择操作。

15.18 对基于线索数据结构的符号表实现排序操作。

▷ 15.19 请编写程序输出与给定搜索关键字前 t 位相同的线索中的所有元素。

○ 15.20 采用 C++的 union 在满足下列要求的线索中实现搜索和插入操作，要求线索包含两种节点：一种节点为含有链接但无元素的非树叶，另一种为含有元素但无链接的树叶。

○ 15.21 采用一对导出类在满足下列要求的线索中实现搜索和插入操作，要求线索包含两种节点：一种节点含有链接但无元素的非树叶，另一种为树叶含有元素但无链接。

15.22 修改程序 15.3 和 15.2 使之满足下列要求：搜索关键字保存在机器的寄存器里，每次沿树下移时采用移位的方法得到关键字的下一位。

15.23 对于一个固定的 $r$，请修改程序 15.3 和 15.2 实现一个 $2^r$ 的线索表，使用前 $r$ 位来索引这些表并采用标准的取模的算法访问相应线索上的关键字。除非表中有非常多的空记录，否则这种方法可以使搜索操作节省 $r$ 步。

15.24 如果随机关键字（假设关键字足够长并且各不相同）数目为 N，练习 15.23 中的 $r$ 应该取多少？

15.25 请编写一个程序，通过将关键字排序并比较排序后的邻接关键字值来计算一系列定长的不同关键字构造的线索的节点数目。

● 15.26 请用归纳法证明性质 15.3 后面给出的随机线索中外部路径长度的递推公式为 $N\Sigma_{t\geq 0}(1-(1-2^{-t})^N)$，该公式与快速排序的公式是类似的。

● 15.27 请推导性质 15.4 给出的随机线索中平均节点数目的公式。

● 15.28 对 $N=10^3,10^4,10^5$ 和 $10^6$，请编程计算 $N$ 个元素的随机线索中的平均节点数目，并以 $10^{-3}$ 的精度输出计算结果。

●● 15.29 请证明 $N$ 个随机 0、1 序列关键字构造的线索的高度为 $2\lg N$。提示：考虑生日问题（见性质 14.2）。

● 15.30 请证明一个随机关键字构成的 DST 的平均搜索开销渐进于 $\lg N$（见性质 15.1 和 15.2）。

15.31 请针对下列情况修改程序 15.2 和 15.3，要求适用变长 0、1 序列的关键字且允许数据结构中存在重复元素。并请考虑 d 大于长度 v 的特殊情况以及函数 bit(v，d)的返回值。

15.32 请用线索构造一个支持某种抽象数据表的数据结构，使用 $w$ 位整数关键字。要求支持构造、插入和搜索操作，其中搜索和插入操作采用整数作参数，若搜索失败则返回 nullItem.key()，搜索命中则返回搜索所使用的参数。

## 15.3 帕氏线索

我们在 15.2 节中介绍的基于线索的搜索有两个缺点。其一是单路分支导致线索中增加不必要的节点。其二是线索中有两种不同类型的节点导致复杂度增加（见练习 15.20 和 15.21）。1968 年，Morrison 发现了解决上述两个问题的方法，提出了帕氏线索（他是根据其算法的英文名首字母的缩写将该算法称为 Patricia 线索，即用于字母数字编码信息搜索的实际算法，我们在这里称为帕氏线索算法）。Morrison 是针对字符串索引的应用提出了他的算法，这种类型的应用我们会在 15.5 节介绍，但他的算法对符号表应用也是有效的。与 DST 类似，帕氏线索可以实现在 $N$ 个节点的树中完成对 $N$ 个关键字的搜索；与普通线索类

似，每次搜索开销仅为 $\lg N$ 次位比较和一次整个关键字的比较，并且支持其他符号表的操作。另外，其性能与关键字长度无关，该数据结构同时适用于变长关键字的情况。

由标准的线索数据结构出发，我们通过一种简单的方法来避免单路分支：在每个节点中存储要测试的位的序号，并由此来决定比较完该节点之后要选取的路径。这样，我们就可以跳过子树上所有对应的位相同的关键字，从而进行直接的比较。而且，我们还采用另一种简单的方法来避开外部节点：我们将所有数据存储在内部节点，并将指向外部节点的链接替代为指向相应线索中内部节点的链接。上述两种方法的采用使得我们可以用由包含关键字和两个链接（还有一个保存序号的附加字段）的二叉树构成线索。这样的数据结构就称为帕氏线索。在帕氏线索中，我们像 DST 那样把关键字存储在节点中，并根据搜索关键字的位来遍历该树，但不采用沿树下移的方法进行遍历，只是到达树底的时候将关键字存储在那里以方便以后访问。

正如上段介绍的，如果把标准线索和帕氏线索看作线索这一抽象数据结构的两种不同表示方法，我们就容易理解帕氏线索的算法机理了。例如图 15.10 和图 15.11 上半部给出的两个例子分别解释了帕氏线索的搜索和插入操作，其实两者的抽象数据结构与图 15.6 给出的线索是一样的。这里的搜索和插入算法是抽象线索数据结构的一个具体实现，当然与 15.2 节介绍的搜索和插入算法有所不同，但两者的实质是一样的。

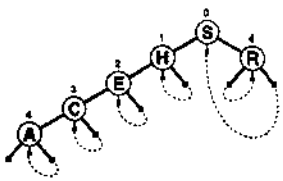



**图 15.10　帕氏线索的搜索**

图中给出了对关键字 R=10010 在帕氏线索中搜索命中的过程。首先向右移（因为第 0 位是 1），然后左移（因为第 4 位是 0），这样指向 R（该树中惟一关键字为 1***0 的元素）。向下遍历的过程中，我们只需比较节点上的序号（而不用比较节点中的关键字）。当我们第一次遇到向上的链接，就将该链接指向的元素与搜索关键字进行比较，因为它就是该树中可能满足搜索条件的惟一关键字。

我们还可以以失败搜索 I=01001 为例，我们从树根出发首先左移（因为第 0 位是 0），然后右移（因为第 1 位是 1），从而找到 H（线索中惟一以 01 开头的关键字），但发现 H 不等于 I，故搜索失败。

程序 15.4 给出了帕氏线索搜索算法的实现。该方法与普通线索的搜索有三点不同：不存在空链接，我们直接比较关键字中既定的一位而不是比较相邻的下一位，并且在指向上部节点处进行一次整个关键字的比较就完成了一次搜索。检验某链接是否指向上部很容易实现，因为（根据定义）随着在树上下移，这类节点中存储的位的序号会增加。搜索过程是这样的：我们从树根出发，沿着树下移，使用节点中的位序号比较搜索关键字中的各位——若当前位是 1 就右移，若是 0 就左移。在遍历过程中节点中的关键字完全不用进行比较。最后，我们遇到向上的链接：每个向上的链接指向线索中最终完成搜索的惟一元素。如果向上的链接指向的元素等于搜索关键字，则搜索命中；否则，搜索失败，从而完成搜索。

**程序 15.4　帕氏线索的搜索**

程序中，递推函数 searchR 的返回值就是线索中惟一含有关键字 v 的记录对应的节点。程序遍历整个线索树，但每遇到一个节点只比较一位——即字段中位序号表明的那一位。当遇到外部节点时，即遇到向上的链接时搜索结束，并比较整个关键字以决定搜索命中还是失败。上述搜索的操作就是由搜索函数 searchR 完成的。

```
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;
    }
```

图 15.10 解释帕氏线索的插入过程。帕氏线索的插入与标准线索的插入有些不同，遍历过程中相应于单分支遍历的位都没有被比较，而标准线索的搜索失败是取决于线索中的空链接。对两者来说每次搜索都是在线索中的树叶节点结束，而且两者都比较搜索关键字，但帕氏线索不比较对应于单分支的那些位。

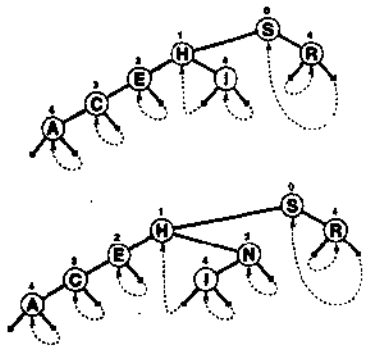


**图 15.11 帕氏线索的插入**

图中给出了将关键字 I 插入图 15.10 中的帕氏线索的过程。通过比较得到 H=01000 和 I=01001 只有第 4 位不同（见图中上半部），我们插入一个新节点。还要完成在插入的新节点后搜索的链接，即如果搜索关键字的第 4 位是 1，我们要搜索 H（左链接），若第 4 位是 0 我们要搜索 I（右链接）。

图中下半部分给出了再插入关键字 N=01110 的示意图。通过比较第 2 位我们在 H 和 I 之间插入节点 N。

如图 15.11 所示，帕氏线索的插入操作恰好体现了线索插入的两种情况。通常，我们可以由搜索失败得到新关键字属于什么地方。对于线索来说，这种失败可能是由空链接引起的，也可能是由叶节点的关键匹配错误引起的。对帕氏线索而言，我们需要额外注意插入操作的类型，因为插入过程中我们没考虑决定分支的位。帕氏线索的一次搜索操作总是结束于一次整个关键字的比较，而且这个关键字包含了全部我们所需的信息。比较过程中一旦我们发现搜索关键字与结束搜索的关键字最左一位不同，就要重新遍历线索，即在搜索路径上与各节点进行位比较。如果在搜索过程中我们遇到一个节点，其确定的位序号高于该节点的关键字与搜索关键字不相等位的序号，此时帕氏线索的搜索对应于标准线索中结束于空链接的搜索，我们就插入一个用来比较该位的新节点。如果没遇到上述情况（即没遇到一个节点其确定的位序号高于该节点的关键字与搜索关键字不相等位的序号），此时帕氏线索的搜索对应于标准线索中在树叶节点结束的搜索，我们就插入一个新节点以区分搜索关键字和最后遇到的关键字（也就是使得搜索结束的关键字——译者注）。我们每次只需插入一个新节点以区别从左边起第一个不同的位，而在标准线索的插入过程中，每次分支都要插入一个节点直到到达

比较位为止。新插入的节点除了提供我们所需要的位辨别外，还用来存放将要插入的关键字。图 15.12 给出了一个帕氏线索建立过程的例子。

程序 15.5 给出的就是帕氏线索的插入算法。这段代码直接源于上一段的论述，并且我们所使用的链接指向的节点其包含的位序号不会大于当前指向外部节点的位序号。故这段代码只需检测链接的这一性质，而不用来回移动某些关键字和对应的链接。起初帕氏线索中指向上的链接显得很奇妙，但在插入每个新节点时所作的采用哪个链接的选择是好理解的。最终结果是采用单一类型的节点比采用两种节点类型使得代码更简洁。

帕氏线索建立起来后，位序号为 k 的节点下面所有的外部节点前 k 位都相同（否则，我们在序号小于 k 的位置就可以插入一个节点以区分序号小于 k 的位置上就有不同的两个关键字）。因此，我们可以采用下述方法将帕氏线索转化为标准线索：在跳过的位之间插入合适的内部节点并且把指向上面的链接替换为指向外部节点的链接（见练习 15.48）。由于关键字在内部节点中的分配依赖于关键字插入的顺序，所以性质 15.2 对帕氏线索并不完全成立。内部节点的结构不依赖关键字插入的顺序，但外部链接的结构和关键字的分配却相反。

由帕氏线索根源于标准线索的事实可以得出一个重要结论：我们可以采用一种递归的无序遍历来有序地访问标准线索数据结构中的节点，程序 15.6 给出了实现方法。我们只访问了外部节点，它们是通过测试非递增的位索引来识别的。

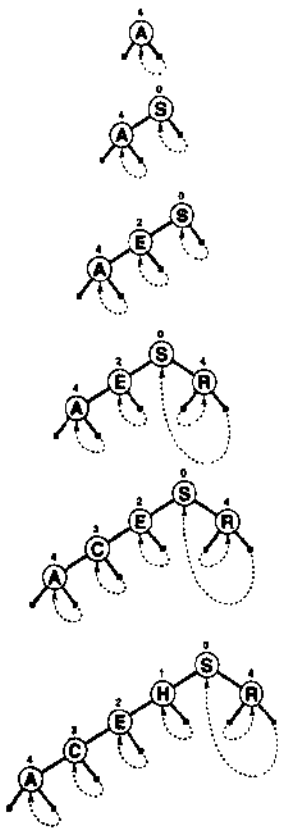



**图 15.12　帕氏线索的建立**

图中给出了将关键字 A S E R C H 插入空的帕氏线索的结果。图 15.11 的下半部分给出了在帕氏线索中插入关键字 I 和 N 的结果。

帕氏线索体现了基数搜索的精髓，即设法标识出关键字中不相同的位并把它们存入一定的数据结构（且无额外的节点）中，使得我们可以尽量快地在其中找到与搜索关键字相当的关键字。图 15.13 给出了用与图 15.9 中相同的关键字构造的帕氏线索——它不仅比标准线索少 44%的节点，而且近乎完全的平衡。

**图 15.13　帕氏线索示例**

通过插入约 200 个关键字建立的帕氏线索与图 15.9 的线索删除单路分支后相同。生成的树近乎完全平衡。

**性质 15.5** *N* 个 0、1 序列关键字的帕氏线索其插入和搜索操作平均需要大约 lg*N* 次位比较，最坏情况下需要大约 2lg*N* 次比较。而且位比较次数不大于关键字的长度。

由于帕氏线索的路径长度不长于相应的标准线索，故该结论是性质 15.3 的一个直接推论。帕氏线索搜索复杂度的精确分析同样是十分困难的；其结果是平均而言，帕氏线索比标准线索的比较次数少一次（参见“第四部分参考资料”）。■

考虑到我们前面提到的各种直接的权衡，表 15.1 的实验数据证明了对整数关键字的符号表应用，DST、标准二叉线索和帕氏线索的性能是可比的（而且这些数据结构与第十三章介绍的平衡树的搜索时间也是可比的，甚至小于平衡树），即使对以 0、1 随机序列做关键字的符号表实现也是如此。

**程序 15.5 帕氏线索的插入**

帕氏线索的插入操作也要首先完成相应的搜索。程序 15.4 中的搜索函数 searchR 返回给我们一个与将要插入的关键字不同的关键字。我们需要判断返回关键字与插入关键字的最左位是否不同，若不同则用这里给出的递归函数 insertR 在树上下移至那点并插入一个包含关键字 v 的新节点。

在函数 insertR 中会出现如图 15.11 所示的两种情况。如果搜索关键字与找到的关键字不同的位被跳过，则插入的新节点将替换一个内部链接；如果搜索关键字与找到的关键字不同的位不用再用来区分线索上的其他关键字，则插入的新节点将替换一个外部链接。

```
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;
        }
      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; }
```

**程序 15.6　帕氏线索的排序**

这里递归的过程显示了帕氏线索中的记录按关键字排序的情况。我们假设帕氏线索中的元素都在外部节点上，而且这些外部节点可以通过比较其上的位序号小于其双亲节点上的位序号来识别。若不然，该程序则是典型的无序遍历。

```
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); }
```

由性质 15.5 不难发现搜索开销不随关键字长度增加。与之相比，标准线索中搜索开销依赖于关键字的长度——两个关键字的第一个不同位可能落在关键字的任意位上。我们所介绍的所有基于比较的搜索算法都依赖于关键字的长度——如果两个关键字仅有最后一位不同，则需要与关键字长度成比例的搜索开销。而且，由于需要计算散列函数，散列算法的搜索开销总是与关键字的长度成比例的。而帕氏线索直接比较关键字第一个不同的位，只需少于 $\lg N$ 次的位比较。这种性能使得帕氏线索（或者不用单一分支遍历的标准线索）成为长关键字搜索应用的首选。

例如，假设我们的计算机为 8 位字长（即一个字节的长度——译者注）且我们要在上百万个 1000 位的关键字中进行搜索。此时，帕氏线索只需访问搜索关键字中的大约 20 个字节（即 160 位——译者注）和 125 字节的比较。而散列算法则需要访问全部 125 个字节（即 1000 位——译者注）以计算散列函数再加几次比较。另外，基于比较的算法需要大约 20 到 30 次整个关键字的比较。对基于关键字比较的算法而言，在一次搜索初期确实只需几次的字比较，但后面的比较次数大大增加。在 15.5 节中，我们将对长关键字的搜索性能再次比较这些算法。

表 15.1 线索实现的实验研究

这里，由 32 位随机整数序列构成的符号表的建立和搜索所需的相对时间证明了数字方法与平衡树方法是相当的，而且即使对随机的 1、0 序列作关键字的情况也是如此。当关键字越长并且越不随机（见表 15.2），或者使得关键位访问的代码更高效时（见练习 15.22），性能差距越显著。

| | 构造 | | | | 搜索命中 | | | |
|---|---|---|---|---|---|---|---|---|
| $N$ | 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 |

其中：

B 红黑 BST（见程序 12.8 和 13.6）

D DST（见程序 15.1）

T 线索（见程序 15.2 和 15.3）

P 帕氏线索（见程序 15.4 和 15.5）

对帕氏线索，完全没有必要限制搜索关键字的长度。我们将在 15.5 节看到，帕氏线索对可能很大的变长关键字十分有效。即使关键字很长，我们仍可以期望，对 $N$ 个记录的帕氏线索的搜索，需要比较的位数大约正比于 $\lg N$。

## 练习

15.33 如果用程序 15.5 将一个已存在于树中的元素再插入其中，将会出现什么结果？

▷ 15.34 将关键字 E A S Y Q U T I O N 按顺序插入空的帕氏线索中，请画出结果。

▷ 15.35 将关键字 01010011、00000111、00100001、01010001、11101100、00100001、10010101、01001010 按顺序插入空的帕氏线索中，请画出结果。

○ 15.36 将关键字 01001010、10010101、0010001、11101100、01010001、00100001、00000111、01010011 按顺序插入空的帕氏线索中，请画出结果。

15.37 对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，用 N 个随机的 32 位关键字构造一个帕氏线索，并与同样关键字的 BST 和红黑树进行树高、节点数目和内部路径长度的比较。（参见练习 15.6 和 15.14）

15.38 对 $N$ 个不同的 $w$ 位关键字的情况，请完整描述帕氏线索在最坏情况下内部路径的长度。

▷ 15.39 对基于帕氏线索数据结构的符号表实现选择操作。

● 15.40 对基于帕氏线索数据结构的符号表实现删除操作。

● 15.41 对基于帕氏线索数据结构的符号表实现联接操作。

○ 15.42 请编写程序，输出与给定搜索关键字前 t 位相同的帕氏线索中的所有元素。

15.43 请修改标准线索的搜索和插入算法（见程序 15.2 和 15.3），用帕氏线索中的办法代替单分支的遍历。如果你完成了练习 15.20，请由那里的程序开始。

15.44 按练习 15.23 的要求，修改帕氏线索搜索和插入算法（见程序 15.4 和 15.5），实现 $2^r$ 个线索的表。

15.45 请说明帕氏线索的每个关键字都在其搜索路径上，并且在搜索操作时遍历到树底才遇到。

15.46 请修改帕氏线索的搜索算法（见程序 15.4），在向下遍历过程中对关键字进行比较以提高搜索命中的性能。并通过实验数据评价修改后性能的提高（见练习 15.45）。

15.47 采用帕氏线索对 $w$ 位整数实现支持某种抽象数据表的数据结构（见练习 15.32）。

● 15.48 请编写程序将帕氏线索和标准线索互相转化。

## 15.4 多叉线索和 TST

对基数排序，如果每次比较多于一位就可以明显地提高搜索算法速度。对基数搜索亦然：每次比较 $r$ 位，我们可以使搜索快 $r$ 倍。但在基数搜索中采用基数排序的这种思想需要多考虑一个问题，即每次比较 $r$ 位要求每个节点上链接的数目 $R=2^r$，那些未被使用的空链接就导致了相当数量的空间浪费。

在 15.2 节中介绍的（二叉）线索中，每个关键位对应的节点有两个链接：一个对应于该位为 0 的情况，一个对应该位是 1。推广到 $R$ 叉线索，每个关键位对应的节点应该有 $R=2^r$ 个链接，每个对应该位的一种可能值。关键字在树叶上有序排列（树叶为所有链接都是空链接的节点）。R 叉线索中的搜索过程是这样的：我们从树根出发先比较最左边的位，根据关键字确定遍历的路径。如果关键位的值是 $i$，就沿第 $i$ 个链接下移（同时将比较位移到下一个关键位）。当到达某个树叶的时候，就得到了这个线索中与遍历路径对应的含有所比较的关键位的惟一关键字，这样我们就可以将该关键字与搜索关键字整个比较以判断搜索命中还是失败。如果搜索到一个空链接，说明搜索关键字的关键位组合不存在于该线索中，所以搜索失败。图 15.14 给出了一个十进制数关键字的 10 叉线索。正如第十章中介绍的，实际应用中常见的数字是由较少的线索节点判别。更多常用类型关键字具备的这种效应是一些高效的搜索算法的基础。

在介绍多类型节点的完整符号表实现之前，我们先要研究存在表（existence-table）问题，即只考虑关键字（而没有相应的记录或其他相关信息）和将关键字插入某数据结构并在其中搜索该关键字是否被正确插入的算法。为了能够使用前面介绍的几种符号表实现的接口，我们采用前面的约定，即搜索失败的返回值是 nullItem，搜索命中的返回值是一个虚元素。这些约定简化了程序代码并且清楚地揭示了多叉线索的结构。在 15.5 节中，我们将

讨论包括字符索引在内的多种符号表应用。

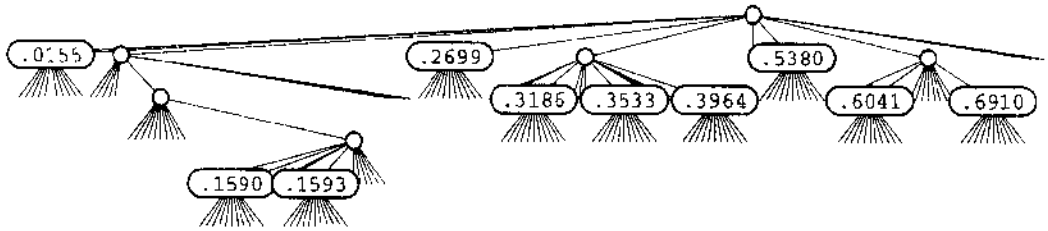



**图 15.14 十进制数字做关键字的 R 叉线索**

图中描述了下面一组关键字（见图 12.1）的线索

.396465048
.353336658
.318693642
.015583409
.159369371
.691004885
.899854354
.159072306
.604144269
.269971047
.538069659

每个节点有 10 个链接（每个指向该位的一个可能值）。在树根节点，0 链接指向第一位为 0 的关键字（仅一个）；1 链接指向第一位为 1 的关键字（有两个），以此类推。由于该组中没有以 4、7、8 和 9 开头的关键字，故这些链接是空链接。而且以 0、2 和 5 开头的关键字在该组中只有一个，故在各自的位置上都有一个树叶节点。这样，每次右移一位，由递归程序构造了这种数据结构。

**定义 15.2** *存在线索（existence trie）是一组递归定义的关键字：无关键字的存在线索是一个空链接；非空的一组关键字对应的存在线索是一个内部节点，它含有指向关键位每个可能值的链接，在构造子树时移去当前关键字的最左边一位不予考虑，即比较下一个关键位，以此类推来建立这种数据结构。*

为了简单起见，我们假设没有任何关键字是其他关键字的前缀。一般地，我们认为关键字是定长的或每个关键字都有以特定位结束的字符（该字符仅仅作为结束符），其值为 NULLdigit 而且关键字各不相同。这个定义的意义在于我们可以通过存在线索来实现存在表而不用对任何信息进行排序；这些信息完全包含在线索这一数据结构的定义中。每个节点有 R+1 个链接（每位的可能取值各对应一个链接再加结束符对应的链接），再无其他信息。搜索时，使用关键字中的位确定遍历的路径。如果关键字的位用尽时恰好到达某个结束符的链接，则搜索命中；否则搜索失败。插入时，首先要搜索到一个空链接，然后为关键字中每一位增加一个节点。图 15.5 给出了一个 27 叉线索插入和搜索的例子；程序 15.7 给出了基本的（多义）存在线索的搜索和插入算法。

**程序 15.7 存在线索的搜索和插入**

这里的多叉线索的搜索和插入实现将关键字保存于线索这种数据结构中。每个节点含有 $R$ 个指向线索下一层的指针。若当前关键字的第 $t$ 个字符是 $i$，则遍历时就沿着第 $i$ 个链接下移。如果给定的参数存在于线索中，搜索函数的返回值就是一个虚元素，否则就返回 nullItem。我们可以修改接口以实现只使用 Key 类型的关键字或者在元素类中实现 Item 到 Key 的类型转换。

```
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); }
```

对于定长且各不相同的关键字，我们可以省掉分配给结束符的链接，并且可以在搜索达到关键字长度的时候结束搜索（见练习 15.55）。在前面采用线索介绍对定长关键字最高位排序的时候（见图 10.10），我们已经给出了一个这种线索的例子。

从某种意义上说，线索这种抽象数据结构是最优的，因为在最坏情况下其搜索的时间开销与关键字长度成比例，而空间开销与关键字中字符个数成比例。但总的空间使用对每

个字符都高达大约 R 个链接，因此我们要设法改进这种算法。正如在二叉树中介绍的，我们可以考虑将纯线索结构看作一种基本的定义良好的一组关键字这种抽象结构的特殊表示，然后再考虑这种抽象结构的其他表示是否有更好的性能。

**定义 15.3** 多叉线索是一种每个树叶对应一个关键字的多叉树，下面给出其递归定义：无关键字的多叉线索是一个空链接；单关键字的多叉线索是一个含有该关键字的树叶；关键字基数大于 1 的线索的某个节点有指向相应关键位的每个可能取值的链接，构造该节点的子树时移去当前关键位不予考虑，比较下一关键位，以此类推建立的数据结构就是多叉线索。

我们假设关键字各不相同且任意一个都不是其他的前缀。在标准多叉线索中搜索，我们使用关键字的各位来确定遍历的路径，可能的结果有 3 种：若到达一个空链接，则搜索失败；若到达另一个包含搜索关键字的树叶，则搜索命中；若到达一个不含搜索关键字的树叶，则搜索失败。每个树叶都含有 $R$ 个空链接，故如 15.2 节介绍的那样，需要区分表示树叶节点和非树叶节点。我们在本章介绍一种实现方法，并将在第十六章给出另一种方法。通过 15.3 节的解析过程我们也可以给出标准多叉线索的性质。

**性质 15.6** 由 $N$ 个随机字串构成的标准 R 叉线索的搜索和插入操作平均需要大约 $\log_R N$ 次字比较。由 $N$ 个随机关键字构成的 $R$ 叉线索的链接的数目大约是 $RN/\ln R$。同时，搜索和插入所需的字比较不超过关键字中的字数。

这些结论是性质 15.3 和 15.4 的结论的推广。我们可以将前面证明中的 2 换成 $R$ 来证明这些结论，这些同样需要高深的数学知识。■

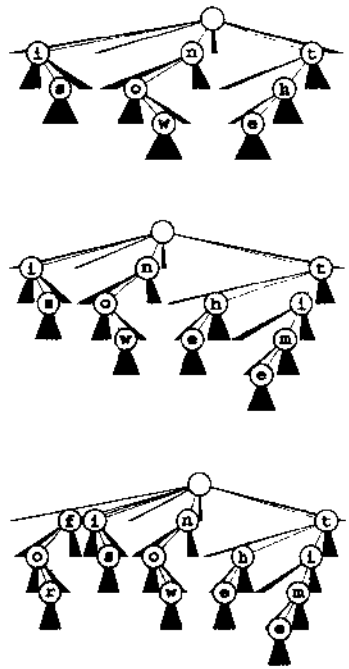


**图 15.15 R 叉存在线索的搜索和插入操作**

对关键字 now、is 和 the，一个 26 叉线索有 9 个节点：一个树根节点加每个字母对应的一个节点（见上图）。图中都用字母标出了相应的节点，但在该数据结构中我们没有用显式的标记，因为每个标记都可以由双亲节点的链接数组中其链接的位置推出。

要将关键字 time 插入其中，我们对已存在节点分支以保存 t 并插入新节点保存 i、m 和 e（见中图）；要插入关键字 for，我们要在树根处分支并加入新节点 f、o 和 r。

性质 15.6 给出的结论体现了算法性能中时间开销和空间开销的折中。一方面，存在大量未使用的空链接——只有接近树根的节点使用相对较多的链接。另一方面，树的高度并不高。我们给出一个典型的例子，$R$=256 并且有 $N$ 个 64 位随机关键字。应用性质 15.6，一次搜索最多需要 $(\lg N)/8$ 次字符比较（最多 8 次），而且我们只使用少于 $47N$ 个链接。如果空间开销不受限制，我们可以采用 $R$=65536，这样需要多于 5900 个链接但只需 4 次字符比较操作，故此时这种算法将非常高效。

在 15.5 节，我们仍将研究标准多叉线索；在这里，我们将介绍程序 15.7 实现的一种线索的表示：三叉搜索线索（TST），图 15.16 同时给出了示意图。在一个 TST 中，每个节点

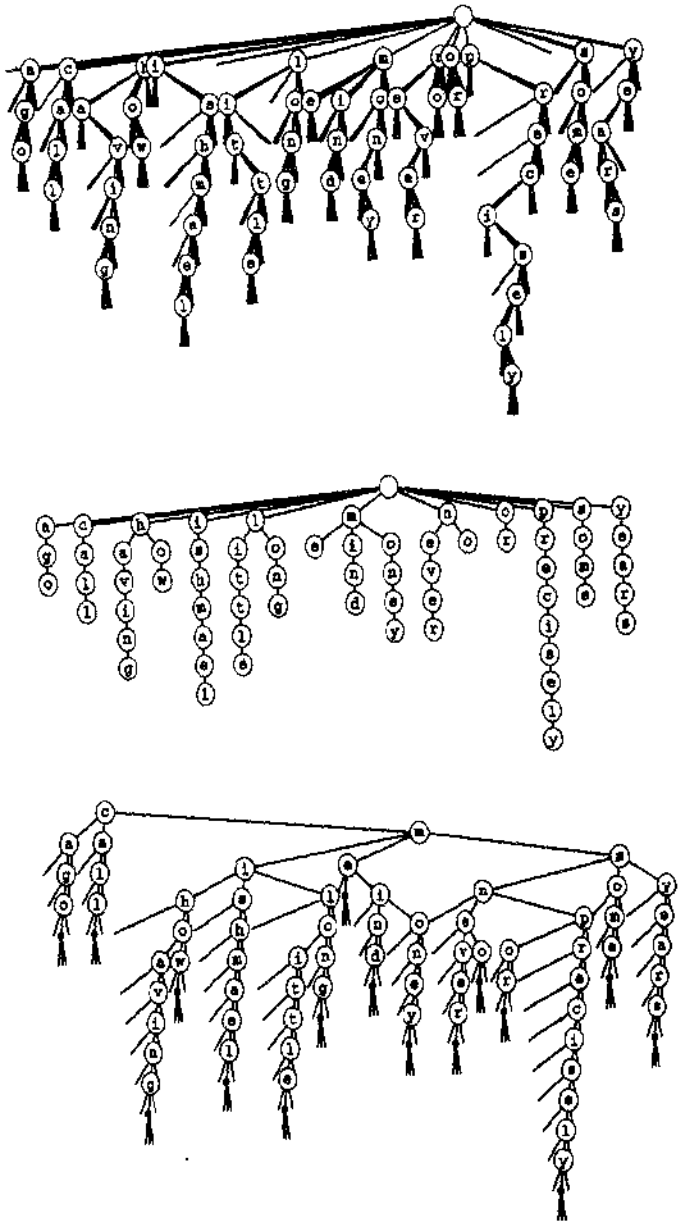


**图 15.16　存在线索的结构**

三幅图给出了 16 个词做关键字的存在线索的不同表示。这 16 个词为：call me ishmael some years ago never mind how long precisely having little or no money。上图是 26 叉存在线索的表示；中间是不带空链接的抽象线索的表示；下图是其 TST 表示。26 叉线索的关键字数目过多，而 TST 是抽象线索的一种有效的实现方法。

上面的两个线索假设不存在一个关键字是另一个的前缀。例如，添加关键字 not 将使关键字 no 消失。我们可以在每个关键字的结尾添加一个 null 字符来改正这个问题，如底部的 TST 所示。

含有一个字符和三个链接，三者分别对应于相应位小于、等于和大于节点上的关键字的情况。三叉线索的这种结构与二叉搜索树中非空链接对应的字符关键字的排列方式是等价的。在程序 15.7 实现的存在线索中，节点表现为 $R+1$ 个链接，而且我们可以通过每个非空链接的关键字序号推出其关键字字符。相应地，TST 中的非空链接对应的所有字符也都表示在每个节点中——只有在遍历中间的链接时才能找到相应于关键字的字符。

这里所介绍的存在三叉搜索树的搜索算法是好理解的；其插入算法相对复杂，但由于其直接对应于标准存在线索的插入算法，故也不难理解。搜索时，我们首先在树根节点比较关键字的第一个字符。如果搜索关键字的第一个字符小于树根的该字符，我们沿左链接下移；如果大于树根的第一个字符，再沿右链接下移；若两者相等，则沿中链接下移，然后继续按上述算法递归搜索。如果我们遇到一个空链接或者遇到一个空节点，则搜索以失败而告终，如果在中链接遍历并且遇到空节点，则搜索命中。要插入一个新关键字，像在标准线索中那样，首先要完成搜索，然后在搜索路径的最后加入新节点。程序 15.8 给出了搜索和插入算法的详细实现，而图 15.17 则给出了与图 15.15 中的线索相应的 TST。

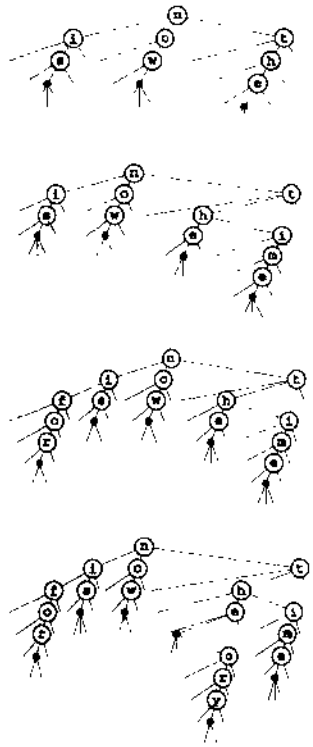


**图 15.17 存在三叉搜索树**

这里的三叉搜索树每个节点含有一个字母，每个节点只有 3 个孩子而不是 26 个。最上面三棵树是相应于图 15.15 的 RST 的插入的例子，但对其进行了修改，即加入了结束符。因此，我们可以去掉每个关键字不能是其他关键字的前缀这一限制，这样，我们便可以如最下面的图所示将关键字 theory 插入其中。

前面我们已经介绍了搜索树和排序算法之间的某些一致性，我们发现 TST 同三路基数排序的关系与 BST 同快速排序的关系一致，同线索和二分快速排序的关系一致。也跟 M 叉线索同 M 路基数排序的关系是一致的。图 10.13 的三路基数排序同样可以看作一组相同关键字 TST。类似地，线索中的空链接问题也对应于基数排序中的空箱问题；同时，三叉分支对上述两种问题都提供了有效的解决办法。

**程序 15.8 存在三叉搜索树的搜索和插入**

这段代码实现了与程序 15.7 同样的抽象线索的算法，但每个节点含有一个关键字和三个链接：三个链接分别对应于小于、等于和大于搜索关键字的相应位的情况。

```
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); }
```

我们可以通过把关键字存储在相应位不同的树叶节点中以更有效地使用空间，还可以像在帕氏线索中那样不采用单分支的内部节点达到提高空间利用率的效果。在本节最后，我们将介绍前一种方法。

**性质 15.7**　完整 TST 的搜索和插入操作的时间开销与关键字的长度成比例。TST 中链接的数目最多为所有关键字中字母个数的三倍。

最坏情况下，每个对应全部 $R$ 个链接的不平衡节点伸展开像一个单链接表。但在随机关键字的情况下这种情况基本不会出现。更典型的是，在第一层，字节比较次数的期望值小于等于 $\ln R$（由于树根节点对 $R$ 个不同类型的值与 BST 很相似），在下面几层也可能如此（前提是：关键字有共同的前缀且其后是 $R$ 个不同的字节值），而且对大多数字母而言，只

需几次字节比较(这是由于多数节点被非空链接分隔得很稀疏)。搜索失败只需几次字节比较，即搜索在线索树的较高处就遇到空链接而结束。搜索命中对每个搜索关键字只需一次字节比较，因为大多数这类关键字都在线索底部单分支的节点上。■

使用 TST 最根本的优点在于它很好地适用于实际应用中关键字不规则的情况。主要有两种主要的效果。首先，实际的关键字来自很大的字符集，并且特殊字符的使用并不统一——例如，一个特定的字符串集可能只取可用字符中的一小部分。采用 TST，我们可以使用 128 或 256 个字符编码而无需担心 128 或 256 叉分支所浪费的空间，而且不用考虑哪些字符集是相关的。非罗马字母的字符集有数千个字母——TST 就特别适用于这类字符串所作的关键字。其次，实际应用中的关键字通常具有一定的结构，其结构随不同应用而不同，可能一时使用关键字某部分的一个字母或另一部分的一位数字甚至是某特殊字符作为分界符(见练习 15.72)。让我们看看图 15.18 的例子，图中给出了一个在线图书馆数据库的部分索书号列表。对这类关键字，某些线索中的节点应表示成 TST 中的一元节点(对有分界符的关键字)；某些应表示成 10 节点 BST 中的 10 节点(对所有位都有数字的关键字)；另外的应被表示成 26 节点 BST 中的 26 节点(对所有位都是字母的关键字)。这种结构自动生成，并不用特别分析这些关键字。

```
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
```

**图 15.18 字符串关键字的例子(图书馆索书号)**

这里采用在线图书馆数据库的例子是要说明实际应用中字符串关键字有多种结构。某些关键字可以看作随机字母序列，某些可看作随机数字序列，还有一些是含有固定值或固定结构的元素。

**表 15.2 字符串关键字搜索的实验研究**

对图 15.18 中图书馆索书号作为字符串关键字构成的线索，其建立和搜索的时间开销证明了 TST 对字符串关键字搜索失败的时间开销最小，这是因为搜索不需要比较关键字中全部字符。当然，这类线索的建立开销稍微大一些。

| | 构造 | | | | 搜索失败 | | | |
|---|---|---|---|---|---|---|---|---|
| *N* | B | H | T | T* | B | H | T | T* |
| 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 |

其中：

B　标准 BST（程序 12.8）

H　采用链接地址法的散列算法（$M=N/5$）（程序 14.3）

T　TST（程序 15.8）

T*　树根为 $R^2$ 路分支的 TST（程序 15.11 和 15.12）

基于 TST 的搜索算法与其他算法相比，另一个实际的优势在于对搜索失败的运行效率非常高，即使关键字很长也是如此。通常只需几次字节的比较（和寻找几个指针）就可以完成一次搜索失败。像我们在 15.3 节讨论的，在一个 $N$ 个关键字的散列表中一次搜索失败所需比较次数与关键字长度成比例（需要计算散列函数），而在搜索树中需要至少 $\lg N$ 次关键字的比较，在帕氏线索中对随机的搜索失败也需要 $\lg N$ 次位比较。

**程序 15.9　TST 中部分匹配的搜索**

我们明智地采用多递归调用，可以在 TST 结构中实现部分匹配的搜索。如程序所示，我们搜索输出某些位不确定（不定的位用星号标出）的字符串的所有部分匹配结果。这里，我们没有采用抽象数据类型的函数或者抽象的元素，而是使用 C 语言处理字符串的语句。

```
private:
  char word[maxW];
  void matchR(link h, char *v, int i)
    {
      if (h == 0) return;
      if ((*v == 0) && (h->d == 0))
        { word[i] = 0; cout << word << " "; }
      if ((*v == '*') || (*v == h->d))
        { word[i] = h->d; matchR(h->m, v+1, i+1); }
      if ((*v == '*') || (*v < h->d))
        matchR(h->l, v, i);
      if ((*v == '*') || (*v > h->d))
        matchR(h->r, v, i);
    }
public:
  void match(char *v)
    { matchR(head, v, 0); }
```

表 15.2 给出了实验数据以证明前面两段的结论。

TST 吸引人的第三个原因是它支持比符号表操作更广泛的操作。例如，程序 15.9 给出了部分匹配搜索的例子，它的输出结果是数据结构中与搜索关键字在某些确定位匹配（而其他位不匹配）的所有关键字。图 15.19 也给出了一个例子。显然，对程序稍加修改，就可以像排序那样访问所有匹配的关键字而不只是输出它们（见练习 15.58）。

其他类似的任务用 TST 同样易于实现。例如，我们可以访问与搜索关键字最多只有一位不同的所有关键字（见练习 15.59）。这类操作开销较大，或者对其他符号表的实现很重要。在本书第五部分，我们将详细介绍字符串搜索中非完全匹配的类似问题。

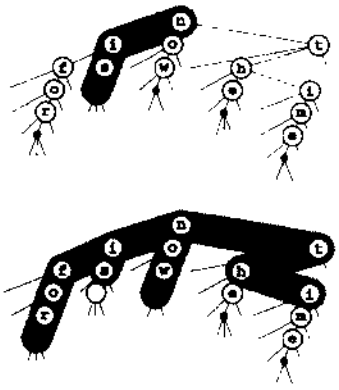


图 15.19 基于 TST 的部分匹配搜索

（如上图所示）要搜索所有符合 i* 模式的关键字，我们在 BST 中搜索所有第一个字符为 i 的元素。在本例中，经过两次单路分支我们找到了 is（惟一符合 i*模式的关键字）。对一个限制更小的模式，如*o*(如下图所示)，我们首先对第一个字符遍历 BST 中全部节点，对第二个字符只访问为 o 的节点，最后得到关键字 for 和 now。

帕氏线索也具有类似的优点：TST 比帕氏线索最大的好处在于它访问的是字节而不是位。原因之一是很多机器提供这种字节操作，而且 C++对 C 语言的字符串数据类型提供了直接访问字节的操作。另一个原因是在某些应用中，字节操作直接反映了数据结构中的数据本身的字节定位——我们可以以上文介绍的部分匹配的搜索为例（尽管我们在第十八章中会介绍一种采用位访问的巧妙算法，但该算法可以使部分匹配搜索的运行速度更快）。

要去掉 TST 中的单路分支，我们需要注意大多数单路分支出现在关键字的末端，而且如果我们改进为多叉线索就不会出现单路分支（在多叉线索中，我们将保存记录的节点置于线索的最高层）。也可以采用类似帕氏线索中保存字节的序号的做法（见练习 15.65），但为简单起见我们不考虑这种方法。多路分支和 TST 表示的结合在很多应用中相当有效，而对关键字很长一段都匹配的情况，用类似帕氏线索去除单路分支的方法更能提高算法的性能（见练习 15.72）。

**程序 15.10 TST 混合节点类型的定义**

这段代码定义与程序 15.11 和程序 15.12 相关的数据结构，应用于采用 TST 的符号表。在树根节点采用 $R$ 路分支：树根是一个 $R$ 个链接的数组的头，它根据关键字的高位排序。每个链接指向一个由含有相应位的关键字构成的 TST。这种混合节点定义综合了线索（树根节点的有序使得搜索更快）和 TST（除了树根节点，每个字符使用一个节点，空间利用率高）的优点。

```
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;
```

基于 TST 的搜索算法的另一种改进是在树根直接使用多叉节点。最简单的方法是实现一个含有 $R$ 个 TST 的表：每个 TST 对应关键字第一位的每个可能取值。若 R 不太大，我们还可以使用关键字的前两位的可能取值（此时表长为 $R^2$）。这种方法达到高效的前提是最高几位的分布尽可能均匀。我们不妨类比人们在电话本中搜索名字的过程。第一步是做一个多选一（“找找以‘A’开头的”），之后可能是二选一（“它在‘Andrews’之前，‘Aitken’之后”），然后应该是一串匹配字符（“‘Algonquin’……不，没有‘Algorithm’，因为没有任何关键字以‘Algor’开头”）。

程序 15.10~15.12 采用一定折中实现了基于 TST 的符号表的搜索和插入操作，它在树根节点采用 R 路分支而且把元素保存在树叶节点（因此只要所有关键字都不同，便不会出现单路分支）。这几段代码是对字符串关键字搜索比较快的算法。TST 这种数据结构同样支持其他一些操作。

在一个很大的符号表中，我们可以令分支因子等于表长。在第十六章我们将介绍建立多叉线索搜索的系统方法，因此我们可以采用多路基数搜索应用于任意大小的文件。

**程序 15.11 混合 TST 实现的符号表 ADT 的插入操作**

这里采用 TST 实现的插入操作是将元素保存在树叶节点中，广义化了程序 15.3。我们对第一个字符采用用 $R$ 路分支，然后对每个字符打头的子分别采用分立的 TST。如果搜索结束于一个空链接，就增加一个新节点。如果搜索结束于一个树叶节点，我们要增加一个内部节点使该树叶与被插入关键字相区别。

```
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); }
```

**程序 15.12 混合 TST 实现的符号表 ADT 的搜索操作**

用（由程序 15.11 实现）TST 实现的搜索操作类似于多叉线索的搜索，对每个节点我们采用三个链接而不是 $R$ 个（除了树根节点以外）。我们根据关键字的位确定遍历路径，搜索结束于空链接（搜索失败）或结束于关键字匹配的树叶节点（搜索命中）或结束于关键字不匹配的树叶节点（搜索失败）。

```
  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); }
```

**性质 15.8** *$N$ 个随机字符序列关键字构成的 TST（元素存储在树叶中且底部没有单路分支），其搜索或插入和树根上 $R^t$ 路分支遍历的操作大约需要 $\ln N - t\ln R$ 次字节访问。而且所需链接的数目为 $R^t$（属于树根节点）加上 $N$ 的常数倍。*

这些粗略的估计可以直接由性质 15.6 推出。就时间开销而言，对 $R$ 个关键字字符的

可能值，我们假设除了某常数个节点外所有搜索路径上的节点都等价于随机 BST，因此我们可以将 BST 的时间开销乘以 $\ln R$。就空间开销而言，我们假设在树的上面几层节点均被关键字字符的 $R$ 个可能值填满，而在底部几层节点中字符值的数目只是某常数。■

我们举个例子来看，如果我们有 10 亿个随机字符串关键字且 $R$=256，在顶端采用一个长度为 $R^2 = 65536$ 的表，则典型的一次搜索需要大约 $\ln 10^9 - 2\ln 256 \approx 20.7 - 11.1 = 9.6$ 次字节的比较。这种方法将搜索开销减少二分之一。如果我们使用完全随机的关键字，则采用直接比较关键字最高字符的存在表算法就可以达到上述性能，这在 14.6 节已经介绍过。采用 TST，对不完全随机的关键字也可以达到上述性能。

对随机关键字，我们需要将树根节点无多路分支的 TST 与标准 BST 进行一下比较。性质 15.8 告诉我们 TST 的搜索需要大约 $\ln N$ 次字节比较，而标准 BST 需要大约 $\ln N$ 次关键字比较。在 BST 顶部，关键字的比较只要一次字节比较，但在树底部则需要多次比较。两者性能的差异并不太大。TST 优于标准 BST 之处在于它对搜索失败运行更快；它适合树根节点的多路分支；（最重要的是）它适合字符串关键字并不随机的情况，因此 TST 中一次搜索的开销不会大于一个关键字的长度。

树根节点 $R$ 路分支的算法并不是对任何应用都有效——例如图 15.18 提到的在线图书馆索书号的数据库，那里所有关键字都是以 $L$ 或 $W$ 开头的。还有一些应用需要更大的树根分支数——例如前面提到的随机整数关键字的例子，这时我们要用尽可能大的表。我们对不同应用要做相应的修改以达到最高性能，但使用 TST 使得我们无需过多考虑对不同应用的修改就可以达到好的性能。

对记录保存在树叶中的线索或 TST，它最突出的特性应该是性能与关键字长度无关。这样我们可以将其用于任意长的关键字。在 15.5 节中，我们将介绍这类应用的一种非常高效的实现。

## 练习

▷ 15.49　采用二十七叉分支，将关键字：now is the time for all good people to come the aid of their party 插入空的存在线索中，请画出结果。

▷ 15.50　将关键字：now is the time for all good people to come the aid of their party 插入空的存在 TST 中，请画出结果。

▷ 15.51　字节长度为两位，将关键字：01010011、00000111、00100001、01010001、11101100、00100001、10010101、01001010 插入空的四叉线索中，请画出结果。

▷ 15.52　字节长度为两位，将关键字：01010011、00000111、00100001、01010001、11101100、00100001、10010101、01001010 插入空的 TST 中，请画出结果。

▷ 15.53　字节长度为四位，将关键字：01010011、00000111、00100001、01010001、11101100、00100001、10010101、01001010 插入空的 TST 中，请画出结果。

○ 15.54　请将图 15.18 给出的图书馆索书号作关键字，插入空的 TST 中并画出结果。

○ 15.55 假设（定长）关键字为 $w$ 字节的词（因此无需结束符），请修改多叉线索搜索和插入的程序（程序 15.7）。

○ 15.56 假设（定长）关键字为 $w$ 字节的词（因此无需结束符），请修改 TST 搜索和插入的程序（程序 15.8）。

15.57 对 $N=10^3,10^4,10^5$ 和 $10^6$，请实验比较下面三种数据结构的时间和空间开销：每字节三位的随机整数关键字构成的八叉线索、每字节二位的随机整数关键字构成的四叉线索和同样关键字的二叉树（参见练习 15.14）。

15.58 请修改程序 15.9，使得它像排序算法那样访问与搜索关键字匹配的所有节点。

○ 15.59 给定一整数 k，请编写一个函数，输出 TST 中与搜索关键字最多有 k 位不同的所有元素。

● 15.60 对 $N$ 个不同的 $w$ 位关键字的情况，请完整描述 $R$ 叉线索在最坏情况下内部路径的长度。

● 15.61 采用多叉线索算法实现一个符号表，要求包含一个析构函数、一个复制构造函数和一个重载赋值操作符，并支持构造、统计、搜索、插入、删除和联接等一级抽象符号表 ADT 操作，并且支持类成员编号（见练习 12.6 和 12.7）。

● 15.62 采用 TST 实现一个符号表，要求包含一个析构函数、一个复制构造函数和一个重载赋值操作符，并支持构造、统计、搜索、插入、删除和联接等第一类抽象符号表 ADT 操作，并且支持类成员编号（见练习 12.6 和 12.7）。

▷ 15.63 请编程输出 R 叉线索中与搜索关键字前 t 字节相等的所有关键字。

● 15.64 采用我们在帕氏线索中相同的做法，请修改多叉线索的搜索和插入算法以去掉其单路分支。

● 15.65 采用我们在帕氏线索中相同的做法，请修改 TST 的搜索和插入算法以去掉其单路分支。

15.66 请编程将一个代表 TST 内部节点的 BST 平衡化（重新分配位置使得所有外部节点都在一层或两层上）。

15.67 请编写 TST 的另一版本的插入算法，实现用平衡树表示所有的内部节点（见练习 15.66）。

● 15.68 对 N 个不同的 w 位关键字的情况，请完整描述 TST 在最坏情况下内部路径的长度。

15.69 首先，编写一个程序生成随机的字长为 80 字节的字符串关键字（见练习 10.19）。对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，采用 $N$ 个前面生成的随机关键字构造一个 256 叉线索，即完成搜索并在搜索失败时插入。要求对每个 $N$，输出线索中节点的总数和建立线索的时间开销。

15.70 对 TST 重新完成练习 15.69。并请比较两者的性能。

15.71 请编写一个关键字发生器程序，通过打乱一个随机的字长为 80 字节的字序列生成关键字（见练习 10.21）。对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，使用该关键字发生器，构造一个 $N$ 个元素的 256 叉线索，即完成搜索并在搜索失败时插入。并请比较该程序与练习 15.69 中对应的随机关键字的程序的性能。

○ 15.72 请编写一个关键字发生器，要求生成字长为 30 字节的随机字符串，且该字符串由下述 4 个字段组成：一个 4 字节字段取自 10 个给定字符串中的一个；一个 10 字节字段取自 50 个给定字符串中的一个；一个 1 字节字段取自 2 个给定字符串中的一个；一个 15 字节字段含有随机的左对齐的字母串，字母串长度介于 4~15 之间（见练习 10.23）。对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，使用该关键字发生器，构造一个 $N$ 个元素的 256 叉线索，即完成搜索并在搜索失败时插入。要求对每个 $N$ 输出线索中节点的总数和建立线索的时间开销。并请比较该程序与练习 15.69 中对应的随机关键字的程序的性能。

15.73　对 TST 重新完成练习 15.72。并请比较该程序与对应与线索的程序的性能。

15.74　采用多路数字搜索树，请实现对字节串关键字的搜索和插入算法。

▷ 15.75　请将关键字：now is the time for all good people to come the aid of their party 插入空的 27 叉 DST 中，并画出结果。

● 15.76　采用链接表表示线索的节点，请实现多叉线索的搜索和插入算法（与用 BST 表示 TST 相反）。并实验研究有序表和无序表两者哪个更有效，并将结果与基于 TST 的算法相比较。

## 15.5　文本字符索引算法

在 12.7 节中我们介绍了建立字符串索引的方法，在一大段文本中，我们采用带字符串指针的二叉搜索树来判断某个给定关键字是否在其中。在本节中，我们从相同的出发点介绍采用多叉线索的更高级的算法。我们将文本中每个位置到文本末看作一个字符串，该位置为字符串关键字的起始点，并由这些关键字构造带字符串指针的符号表。所有的这些关键字都不相同（例如，长度不同），而且多数都很长。在这里，搜索的目的是判定给定关键字是否为索引中的某个关键字的前缀，这就相当于搜索关键字是否出现在该文本中。

由指向文本中字符串指针确定的字符串关键字构成的搜索树被称为下标树（suffix tree）。我们可以使用任何允许变长关键字的算法。基于线索的搜索算法特别适用，因为其时间开销与关键字长度无关（除了在关键字尾部采用单路分支的算法），但与各关键字相区别的位的个数有关。这种特性与运行时间正比于关键字长度的算法（比如散列算法）形成了鲜明的对比。

图 15.20 给出了分别由 BST、帕氏线索和（带树叶的）TST 构造的文本字符索引的例子。这里的索引都是采用起始于字边界的字符串做关键字；而起始于字符边界的字符串做关键字会使得索引更复杂，并且使用更多的空间。

严格来讲，即使随机的字符串文本也不会导致索引中相应关键字为随机的（因为这些关键字不是独立的）。我们在实际应用中很少遇到随机文本的情况，而且分析方法的不同也不会影响我们采用基数的方法来达到快速索引算法。我们不对每一种实现字符串索引的方法一一详细介绍其性能特点，因为我们前面所讨论的字符串关键字的符号表的某些折中对字符串索引算法也适用。

对一个典型文本，因为标准 BST 易于实现，所以成为我们的首选（见程序 12.10），而且在典型应用中，可以提供好的性能。关键字的相互依赖（特别是对每个字符位置建立一个索引的时候）导致 BST 应用于大文本时常出现最坏情况，尽管非平衡 BST 只有在建造的很差时才出现。

帕氏线索最初就是为字符串索引的应用而设计的。要使用程序 15.5 和 15.4，我们只需写一个实现位操作的函数，该函数对给定的字符串指针和整数 i 能返回字符串中的第 i 位（见练习 15.82）。实际应用中，实现文本字符串索引的帕氏线索高度是对数的。而且，由于不用比较关键字的全部字节，所以帕氏线索对搜索失败特别有效。

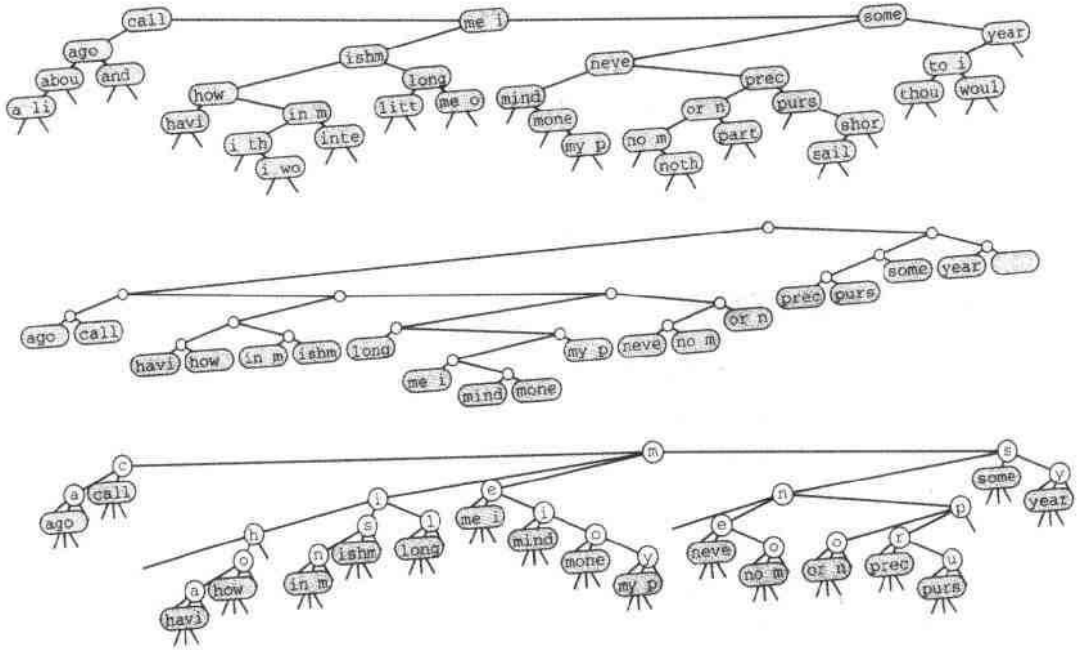



**图 15.20 文本字符串索引的例子**

这里给出了由文本：call me ishmael some years ago never mind now long precisely……构成的文本字符串索引。上图采用 BST，中间的图采用帕氏线索，下图采用 TST。在指针指向的点开始每四个字符构成一个节点。

采用 TST 同样具有帕氏线索的一些优点，易于实现，而且目前的机器人都具有字节访问的操作对 TST 更为有利。对于非完全匹配的简单应用，如程序 15.9，它也能实现。使用 TST 构造字符串索引，由于不会出现某一关键字是另一关键字前缀的情况，我们要删除其代码中处理字符串尾的部分，而且因此我们不用比较到字符串尾。由于我们要从文本某个位置开始，将一个（短的）搜索关键字与一个（长的）关键字相比较，于是就需要修改元素类型接口中操作符==的定义，使得如果一个字符串是另一个的前缀就认为两者相等，这与 12.7 节中一样。另一个必要的修改是我们在节点中保存的是字符串指针而不是字符，这样使得每个节点指向文本中的一个位置（该位置出现在树中由树根到该节点所确定的字符串在文本中第一次出现的地方）。实现上述修改就可以灵活高效地实现文本字符串索引（见练习 15.81）。

除了上面介绍的好处，我们在使用 BST、帕氏线索和 TST 应用于典型文本字符串索引时还有一个很重要的方面：文本一般是固定的，故我们无需支持动态插入操作。也就是说我们只需建立一次索引并用它来完成非常多次搜索而无需改变它。因此我们不需要像 BST、帕氏线索或 TST 那样的动态数据结构。最基本的算法是字符串关键字的二叉搜索（见第 12.4 节）。建立的索引是一系列字符串指针；索引的建立是一个字符串指针排序的过程。采用二叉搜索而不用动态数据结构的优势在于节省空间。对具有 N 个位置的文本建立索引，采用二叉搜索需要 N 个字符串指针；而采用基于树的算法就需要至少 3N 个指针（一个指针指向字符串，另外两个为链接）。要索引的文本通常很大，由于二叉搜索算法只使用基于树的

算法三分之一的空间开销就可以实现呈对数的搜索时间开销，故二叉算法可能更常用。如果有足够的空间，由于 TST 在关键字上单向移动而不用折回（二叉搜索无法做到），所以对很多应用 TST 可以提供快速的搜索。

如果我们有很大的文本并打算做少量的搜索，就无需建立完整的索引。在第五部分，我们将考虑字符串搜索问题，即快速判断一段文本中是否含有给定关键字且无需预处理。我们还要介绍其他字符串搜索的问题，它们介于无需预处理和建立完整的索引这两个极端之间。

## 练习

▷ 15.77　请对下列词：now is the time for all good people to come to aid of their party 使用 26 叉 DST 建立文本字符串索引，并画出结果。

▷ 15.78　请对下列词：now is the time for all good people to come to aid of their party 使用 26 叉线索建立文本字符串索引，并画出结果。

▷ 15.79　请对下列词：now is the time for all good people to come to aid of their party 使用图 15.20 所示的 TST 建立文本字符串索引，并画出结果。

▷ 15.80　请对下列词：now is the time for all good people to come to aid of their party 使用课文中介绍的每个节点含有字符串指针的方法建立文本字符串索引，并画出结果。

○ 15.81　请修改程序 15.11 和 15.12 的 TST 搜索和插入算法，实现基于 TST 的字符串索引。

○ 15.82　请实现帕氏线索将 C 的字符串类型（即字符数组）作为位串（即 0、1 序列——译者注）来处理的程序接口。

○ 15.83　对字母表中的第 i 个字母采用 i 的 5 位二进制编码表示，并对下列词：now is the time for all good people to come to aid of their party 使用帕氏线索建立文本字符串索引，请画出结果。

15.84　采用基于 TST 的思想（即比较字符而不是整个字符串）来改进二叉搜索是无效的，为什么？

15.85　在你的系统中找一个大文本文件（不小于 $10^6$ 字节），分别采用标准 BST、帕氏线索和 TST 算法对该文本进行字符串索引，请给出三者的高度和内部路径长度。

15.86　对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$，请分别实验研究对 $N$ 个 32 字符的字符表中的随机字符串文本建立索引，依次采用标准 BST、帕氏线索和 TST 算法，请比较三者的高度和内部路径长度。

○ 15.87　请编写一个高效的程序找出一段大文本中最长的重复字符串。

○ 15.88　请编写一个高效的程序找出一段大文本中出现频率最高的长度为 10 的字符串。

● 15.89　请建立一个支持下述操作的字符串索引：可以返回给定参数在文本中出现的次数，并像排序和搜索那样对所有与搜索关键字匹配的位置都访问。

○ 15.90　请给出一段长为 $N$ 的字符串，使得对基于 TST 的字符串索引的性能很差。并估计采用 BST 算法对该文本索引需要的开销。

15.91　对 $N=10^3$，$10^4$，$10^5$ 和 $10^6$ 分别对随机的 $N$ 位字符串建立索引，且每位都是 16 的倍数。请实验研究字节长度取多少（1、2、4、8 还是 16）才使得基于 TST 的索引时间开销最小？

# 第十六章

# 外 部 排 序

用于超大文件中的元素搜索算法有非常广泛的实际应用。对巨大的数据集，搜索是一种基本的操作，而且在很多应用环境中占据相当大的资源。随着因特网的出现，我们具备了搜索与某一工作相关的所有信息的能力——同时我们要更有效地去完成搜索任务。在本章中，我们讨论对我们可以想像的任意大的符号表进行高效搜索的算法机理。

类似于第十一章，这里要讨论的算法与不同应用的软硬件环境有关。因此，我们将要介绍的算法比前几章中给出的 C++算法更抽象。当然，这些算法也都是我们熟悉的搜索算法的直接实现，而且用 C++语言方便地表示了出来并适用于很多情况。在这里，我们采用的方式跟第十一章有所不同：首先详细介绍特定的算法实现，再介绍其性能特点，然后讨论具体算法如何应用于实际情况才更有效。咬文嚼字地讲，本章的标题用词不当，因为我们将用 C++程序来描述这些算法，而这些程序与我们在第十二章到第十五章介绍的其他符号表应用算法是可以互换的。同样，它们根本不是“外部”方法。然而，那些算法都是基于一个简单的抽象模型，这样使得我们可以对特定的外部设备给出精确的实现搜索的方法。

详细的抽象模型对搜索不像对排序那样有用，这是因为对很多重要应用而言，模型所涉及的搜索开销很低。我们主要考虑的是存储在外部设备上的大型文件的搜索，我们可以快速访问这些外部设备上任意大的数据块，比如一个磁盘。对只可以顺序存取的磁带设备（其模型我们在第十一章讨论过），搜索操作便退化为从头至尾有顺序地读操作。对磁盘设备，我们可以更有效地搜索：最为显著的是，我们将要介绍的算法只需 3 到 4 个磁盘数据段的引用，就可以支持数亿甚至数万亿个记录的符号表的搜索和插入操作。系统参数，如数据段长度和访问新段的开销与访问段内某元素的开销之比等，会影响算法的性能，但对参数的一定取值范围内这种影响相对而言并不明显（而这恰是实际应用中常遇到的情况）。而且，我们要使得这些算法适用于特定的实际应用所要采取的重要步骤是显而易见的。

对磁盘设备而言，搜索是一种重要的操作。根据磁盘的特定性能，人们组织文件以尽可能快速地存取磁盘上的信息。简言之，我们可以认为在磁盘这种被人们设计用来存储大

量信息的设备上可以简单高效地实现搜索操作。在本章中，我们介绍的算法是基于比基本磁盘操作稍微高一点的抽象层之上，这些算法还支持插入等其他动态符号表操作。这些算法优于直接算法之处等价于 BST 和散列算法优于二分搜索和顺序搜索之处。

在很多应用环境中，我们可以对一个巨大的虚拟内存进行直接寻址，并依靠操作系统来有效地处理程序对数据的需要。我们将讨论的问题在这类环境下的符号表应用中也很有价值。

一个用计算机处理的信息集合称为一个数据库。很多研究都给出了建立、维护和使用数据库的方法。这其中很多工作是对抽象模型的研究和搜索操作的实现，当然，这里的搜索的标准比我们前面考虑的“单一关键字匹配”的判据更为复杂。在一个数据库中，由于含有多重关键字，所以搜索是基于部分匹配的，而且返回结果可能是很多记录。我们在第五部分和第六部分讲解了这类算法。通常的搜索需要是足够复杂的，因为它需要我们在整个数据库中顺序搜索，对每一个记录检查其是否满足搜索标准。目前，对数据库系统而言，在大的数据文件中非完全匹配的快速搜索性能是很重要的，而且很多现代的数据库的理论基础正是本章所介绍的算法。

## 16.1 游戏规则

如第十一章中那样，我们认为顺序存取数据比非顺序存取经济很多。我们所采用的模型都把任何用来实现符号表的存储设备分为若干页面，即可以被磁盘高效存取的连续的信息段。每个页面含有很多记录；我们的任务就是有序地组织页面内的记录，使得我们只访问几个页面就可以搜索到任一个记录。我们假设读一个页面的 I/O 时间决定着访问一个记录或者在该页面内进行其他运算所需的时间。这个模型在很多方面都过于简化，但它体现了外部存储设备的基本特性，使得我们便于研究基本的算法。

**定义 16.1** *页面是一个连续的数据段。对一个页面的第一次访问称为探测（probe）。*

我们对探测次数少的符号表实现感兴趣。我们不去精确地估计页面大小和探测时间与段内记录访问所需时间之比。我们期望这些值大约为 100 或 1000；这里无需很精确地估计是因为，我们将介绍的算法对这些值并不十分敏感。

这个模型与下面这种文件操作系统是相关的，在这种操作系统中，很多文件组成了具有惟一标识符的段，并使得访问、插入和删除操作的效率很高。一定数目的记录组成一个数据段，而且相对于读一个段的开销而言，在段内处理记录的开销小得多。

这个模型与虚拟内存系统也是相关的，这种虚拟内存系统使得我们可以访问非常大一段地址空间，通过它把常用数据存储在快速设备（如内存）而把不常用数据存储在速度慢的设备（如磁盘）。很多计算机系统都具备复杂精确的分页寻址机制，通过把常用页面保存在缓存中以快速存取来实现虚拟内存。分页寻址系统也是基于我们所考虑的抽象模型：把磁盘分段，并认为段寻址的时间开销远远大于段内存取数据的时间开销。

所以，我们提出的页面的概念与文件操作系统中的段和虚拟内存系统中的页面是一致

的。简单起见，在考虑算法实现的时候我们通常认定这种一致性。出于某些特殊应用的要求不同，我们可能会考虑每段的多页面或者每页面的多段；这些细节不会降低算法的效率，从而不会影响其在某一抽象层面上的实用性。

我们生成页面、页面引用和带关键字的记录。对一个巨大的数据库，最重要的问题是如何维护记录索引。在 12.7 节中，我们认为记录按某一固定的表的形式组成符号表，我们的任务就是构造一个带有关键字和记录索引的数据结构，使得我们可以快速找到一个给定记录。例如，一个电话公司把客户信息保存在一个静态数据库中，该库有几个索引字段（可能使用不同关键字），公司用这个数据库完成每月收费、日常事务处理、周期性的请求回应等事务。对巨大的数据集，数据索引非常重要：我们通常不复制基本数据的复本，一来是出于那样需要相当多的额外空间，二来是为了避免多复本带来的数据不统一问题。

因此，我们通常认为每个记录是实际数据的一个引用，这个记录可能是一个页地址或数据库的一个复杂接口。简单起见，我们在数据结构中不保存数据的复本，但我们要保存关键字的复本——这是实际需要。同样为了简单起见，我们对记录和页地址不用抽象的接口——而是使用指针。这样，我们可以在一个虚拟内存系统中直接使用这里的算法，但需要将指针和指针访问转化成适合外部排序算法的复杂格式。

对前面提到的两个参数（段的大小和相对访问时间）在相当广范围内的取值，我们来研究一个完整动态符号表的搜索、插入等操作如何用几次探测就完成的算法。如果我们要完成非常多次操作，则需要做一些认真的改进。例如，如果我们能把搜索开销由 3 次探测减为 2 次，则可以将系统性能提升 50%！但在这里我们暂不考虑这种改进；这里的效率是随着系统和应用变化的。

在早期的计算机中，外部存储设备往往体积大、速度慢而且存储量小。因此，很有必要通过一定方法克服这些缺点的限制。早期的编程学中尽是介绍如何在旋转的磁盘、磁鼓等外部设备上快速存取数据的程序，或者是介绍如何使得存取数据所需物理位移最小的程序，而且还有一些小错误带来的算法的大失败。相比而言，当今的存储设备不仅体积小而且速度快、存储量大；所以基本不用考虑前面提到的问题。事实上在当今的编程环境中，我们都可以避开各种设备的物理特性——即我们在各种计算机（甚至在未来的机器）上都达到高效率比在某一特定机器上达到最高效率更重要。

对长期存在的数据库，围绕着数据的完整性和灵活可靠的访问提出了很多问题。在这里我们不予考虑。对于这类应用，我们将要介绍的方法便是一些好的算法的根本，并成为系统设计的出发点。

## 16.2 索引顺序存取

建立索引的一个直接方法是使用一个包含关键字和记录引用的数组，按照关键字的顺序保存，并用二分搜索（见 12.4 节）来实现搜索操作。若有 $N$ 条记录，则这种方法需要 $\lg N$ 次探测——即使对很大的文件亦然。我们这里所采用的基本模型使得我们立刻会想到两点

对这个简单方法的改进。第一，通常索引本身会很大并超出一个页面的范围。由于我们可以通过页面引用访问页面，故我们可以建立完全平衡的二叉树，该树的内部节点包含关键字和页面引用，外部节点包含关键字和记录指针。第二，访问 $M$ 表记录项的开销与访问二叉表的开销一样，故我们可以采用 $M$ 叉树使得每个节点的开销与二叉树一样。这一改进将探测数目减小为与 $\log_M N$ 成比例。正如在第十章和第十五章中介绍的，我们在实际实现中可以把这个量看作常量。例如当 $M$ 取 1000 时，若 $N$ 小于 $10^{12}$，则 $\log_M N$ 小于 5。

图 16.1 给出了一个关键字集合的例子，图 16.2 给出了相应关键字构成的树的例子。为了讲述的方便，我们只能采用相对较小的 $M$ 和 $N$ 值，但它仍表明对大的 $M$ 值树是平衡的。

图 16.2 给出的树是一种独立于设备的索引表的抽象表示，它与我们已经讨论过的其他很多种数据结构是相似的。另外，我们注意到，这种数据结构与底层的磁盘存取软件中依赖设备的索引相差并不太远。例如，在早期的操作系统中采用一种两层模式，其底层对应于特定磁盘设备页面上的记录，而第二层则对应于一个指向各个设备的主索引。在这种操作系统中，主索引保存在主存储器中，因此访问一个记录需要两次磁盘访问：一次访问得到索引地址，一次访问得到含有该记录的页面地址。随着磁盘容量的增大，索引的大小也增加，使得一个索引要保存在几个页面中，最后导致图 16.2 所示的分级模式的出现。我们要继续介绍这种数据结构的抽象表示，因为它可以由系统底层的软硬件来直接实现。

| | |
|---|---|
| 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 |

**图 16.1 八进制关键字的二进制表示**

我们在本章多次使用图中左边一栏给出的 3 位八进制关键字，其右边给出了相应的 9 位二进制表示。

当今很多操作系统也使用类似这种树的数据结构在一个磁盘页面序列上组织大型文件。这类树不含关键字，但它们能有效地支持顺序存取文件的一些常用操作，而且如果每个节点含有它对应的树的大小，则它们还支持搜索文件中第 $k$ 个元素所在页面地址的操作。

由于这里将顺序关键字组织和索引存取结合起来，故人们习惯地将图 16.2 给出的索引算法称为索引顺序存取。该算法适用于数据库极少变动的情况。我们有时也将这里的索引称为目录。采用索引顺序存取的一个缺点是修改目录的开销很大。例如，增加一个记录就需要重建整个数据库，因为很多关键字要分配新位置而且要更新索引值。为了克服这一缺点，早期的操作系统提供了磁盘上的溢出页面和页面上的溢出空间，但这种方法对动态环境并不十分有效（见练习 16.3）。我们在 16.3 节和 16.4 节介绍的方法对这类特定情况提供了系统而高效的解决方案。

**性质 16.1** 在一个索引顺序文件中一次搜索需要常数次探测，但一次插入需要重建整个索引。

在这里（而且在本章全文中）我们用“常数”一词并不十分严格，用它引用与 $\log_M N$ 成比例的量。我们以前讨论过，对实际的文件大小，“常数”一词是适用的。图 16.3 给出了更多的例子。即使我们使用一个 128 位的搜索关键字，并具备 $2^{128}$ 个不同的记录（这几乎大到无法实现了），我们仍可以使用 1000 叉分支，只需 13 次探测就可以完成数据项的搜索。■

我们在这里不考虑这种索引的建立和搜索，因为它是我们在 16.3 节中讨论的基本算法的一个特例（见练习 16.17 和程序 16.2）。

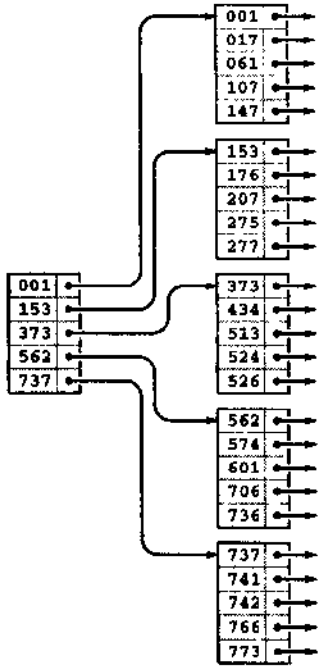



**图 16.2 索引顺序文件结构**

在一个顺序索引中，在整个页面我们按顺序保存关键字（图右半部分），每个页面的索引就是该页面最小的关键字值（图左半部分）。要增加一个关键字，需要重新建立这种数据结构。

### 练习

> 16.1 对 $M$=10，100，1000 和 $N$=$10^3$，$10^4$，$10^5$ 和 $10^6$，请列表表示 $\log_M N$ 的值。

▷ 16.2 对 $M$=5 和 $M$=6，分别画出给定的关键字的索引顺序文件的结构。关键字为：516，177，143，632，572，161，774，470，411，706，461，612，761，474，774，635，343，461，351，430，664，127，345，171 和 357。

○ 16.3 假设我们在容量为 $M$ 的页面上建立了 $N$ 个记录的索引顺序文件，而且每个页面保留了 $k$ 个空位置供数据库扩张使用。请给出完成一次搜索所需探测次数的函数公式，它是关于 $N$，$M$ 和 $k$ 的函数。并使用这个公式对 $M$=10，100 和 1000，$N$=$10^3$，$10^4$ 和 $10^5$ 和 $10^6$，$k$=$M$/10，计算出一次搜索所需探测次数。

○ 16.4 假设一次探测的开销是 $\alpha$ 个时间单位，在页面内搜索一个记录的平均开销是 $\beta M$ 个时间单位。对 $\alpha/\beta$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$，请计算 $M$ 取何值时，索引顺序文件中一次搜索的时间开销最小。

## 16.3 B 树

要构造在动态情况下同样有效的搜索数据结构，我们需要建立多叉树，但每个节点不一定都必须含有 $M$ 个记录。我们只需要求每个节点的记录数不超过 $M$ 个就可以了，这样节点仍将充满整个页面，但每个节点的记录数变少了。为了保持遍历搜索时的搜索路径足够短，我们要弄清每个节点至少要包含的记录数，我们认为每个节点要含有（或者说，至少含有）$M$/2 个节点，而树根节点例外，它至少含有一个记录（即两个链接）就可以。后面我们详细介绍这种数据结构的实现的时候就会清楚地看到这种例外的原因了。Bayer 和

McCreight 首先研究了将多叉平衡树应用于外部搜索，并在1970年将这种数据结构命名为B树。很多人仅仅用B树这一术语代表Bayer和McCreight在算法中提出的那种数据结构；在这里，我们将B树这一术语一般化，用来代表一类相关的数据结构。

我们已经给出了一个B树的实现：通过定义13.1和13.2，我们看到了阶数为4的B树正是第十三章中介绍的平衡2-3-4树，其节点的链的数目最多为4最少为2。事实上，两者基本的抽象是一致的，因此我们可以通过构造自上而下的2-3-4树（见13.4节）来实现B树。但是，内部搜索和外部搜索的不同（见16.1节）使我们不得不考虑实现中的一些具体问题。在这一节我们研究实现具有下述特性的搜索树：

- 将2-3-4树推广为节点数为 $M/2$ 到 $M$ 的树；
- 将多叉节点用记录和链的数组表示；
- 实现一个索引结构而不是含有记录的搜索结构；
- 对节点从树底向上拆分；
- 将索引与记录分离。

上述特性中后两个并不重要，但易于实现，并且在一些B树应用中很常见。

| | |
|---|---|
| $10^5$ | 词典中词条的数目 |
| $10^6$ | 《Moby Dick》一书的字数 |
| $10^9$ | 社会保障号码的数目 |
| $10^{12}$ | 全世界电话号码数目 |
| $10^{15}$ | 地球上生活过的总人数 |
| $10^{20}$ | Coney 岛沙滩上沙子的颗粒数 |
| $10^{25}$ | 人类所生产的所有存储设备的比特数 |
| $10^{79}$ | 宇宙中的电子数 |

**图16.3 数据集大小的几个例子**

这里的数据告诉我们任意符号表都不会超过 $10^{30}$ 条记录。即使在这样一种不可能的巨大数据库中，如果我们采用1000叉分支，我们只需少于10次探测就可以完成对关键字的搜索。甚至如果我们有本事存储宇宙中每个电子的信息，采用1000叉分支的搜索使得我们只需不足27次的比较。

图16.4给出了一个4-5-6-7-8树，它是将13.3节中的2-3-4树加以推广实现的。其推广是简单明了的：四节点含有3个关键字和4个链，五节点含有4个关键字和5个链，以此类推，这里每个链用于关键字之间的任意可能间隔。搜索时从树根出发，比较当前关键字与搜索关键字的间隔，然后经由相应的链到达下一个节点。如果在某一节点中找到搜索关键字则搜索命中；如果遍历到树底都没有找到搜索关键字则搜索失败。与自上而下的2-3-4树中一样，如果我们把一个满的节点拆分，则在向下遍历搜索完成后我们可以在树底插入一个新关键字：如果树根是八节点，我们将它拆分成连接两个四节点的二节点；或者，如果遇到连在一个八节点上的k节点，我们就把它替换为连在两个四节点上的 $(k+1)$ 节点。这一规定使得我们在到达树底时有插入新节点的空间。

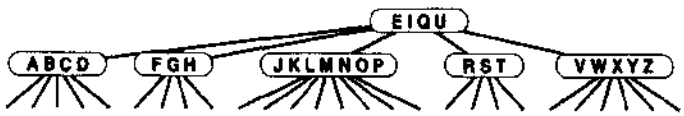



**图16.4 一个4-5-6-7-8树**

该图给出了一种2-3-4树的实现，这棵树是由含有4至8个链接的节点（相应地含有4至7个关键字）构成的。在建立2-3-4树时，可采用自上而下或自下而上的插入算法，当遇到八节点（含有8个链接的节点——译者注）时便将它拆开，这样使得树高为常数。例如，要将另外一个J插入这棵树中，首先要将八节点拆开成两个四节点，然后将M插入树根，使其变成一个六节点。如果遇到树根被拆分的情况，我们不得不重新建立一个二节点的树根。这就是前面讲到的树根节点链的数目不能是4（即M/2，此处M=8——译者注）的原因。

另外，正如我们在 13.3 节讨论的 2-3-4 树的特性，我们也可以自下而上进行拆分，其过程是这样的：先搜索，再将关键字保存在树底的节点上，若该节点是八节点，则将它拆分为两个四节点，将它的关键字中间一个插入其双亲节点并建立指向两个新的四节点的链，按上述步骤沿树自下而上直到遇到非八节点为止。

将上面两段中的 4 和 8 分别用 $M/2$ 和 $M$ 来代替，就是对 $M/2$-...-$M$ 树的描述，这里 $M$ 可以取任意（包括 2 在内的）正偶数（见练习 16.9）。

**定义 16.2** *$M$ 阶 B 树或者为空，或者由 $k$ 节点构成，每个节点含有 $k$-1 个关键字和 $k$ 个指向树的链接，它们代表不同关键字确定的 $k$ 个关键字间的间隔。这种数据结构有下述特点：在树根节点 $k$ 必须介于 2 和 $M$ 之间，在其他节点 $k$ 必须介于 $M/2$ 和 $M$ 之间；指向空树的链到树根的距离必须相等。*

B 树算法就是基于上述基本的抽象。与第十三章一样，在选择这类树的具体实现时我们有相当大的自由。例如，我们可以采用扩展红黑树来实现（见练习 13.69）。对外部搜索，假设 $M$ 足够大使得 $M$ 节点可以充满一个页面，我们可以用更简单的有序数组来实现。分支因子至少为 $M/2$，因此一次搜索或插入所需的探测次数就为常数，这将在后面的性质 16.2 中介绍。

除了实现前面我们提出的方法，我们还要考虑采用 16.1 节介绍的标准索引能否实现的问题。我们将索引指向外部页面的关键字和索引指向内部页面的关键字的副本保存在树底。在树底插入新关键字，并使用基本的 $M/2$-...-$M$ 树抽象模型。当一个页面含有 $M$ 个记录时，我们将它拆分成两个 $M/2$ 个记录的页面，并在其双亲中插入一个指向新页面的索引。当树根需要拆分时，我们生成一个含有两个孩子的新树根，这样使得树高增加了 1。

图 16.5 到 16.7 给出了将图 16.1 中的关键字（按给定顺序）插入空树中构造 B 树的过程，这里 B 树的阶数 $M$=5。插入操作简单地说就是在页面中增加一个新记录，但我们可以通过观察最后的树的结构发现并不简单，其中有几个重要的步骤。它共有 7 个外部页面，因此需要 6 次内部节点的拆分，而且该树高为 3，所以树根必须拆分 2 次。这些在图的注解中已有说明。

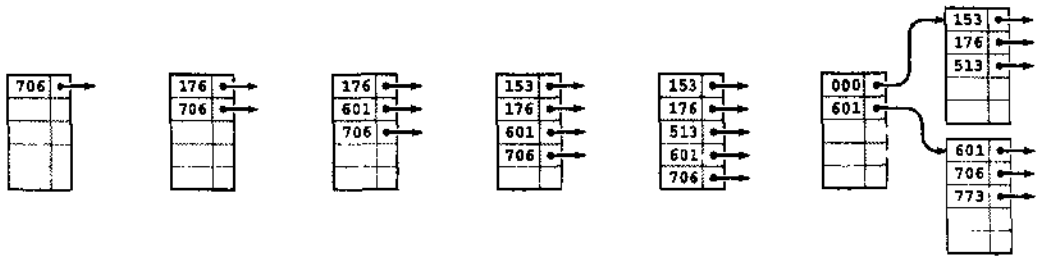



**图 16.5 B 树的建立，第一部分**

本例给出了 3 位八进制（即 9 位二进制）关键字插入空的 B 树的过程，共 6 次插入，所在页面可容纳 5 个关键字和链。在页面内我们按顺序存储关键字。第 6 次插入操作导致了一次拆分，将一个节点拆成两个各含 3 个关键字的外部节点，并产生一个作为索引的内部节点：它的第一个指针指向含有大于等于 000 且小于 601 的关键字的页面，第二个指针指向含有大于等于 601 的关键字的页面。

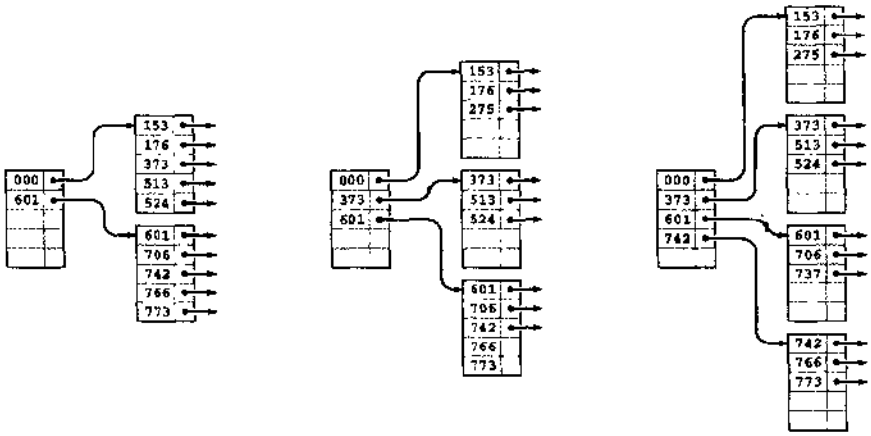



图 16.6　B 树的建立，第二部分

在图 16.5 最右边一副的树上插入关键字 742，373，524 和 766 之后，所有页面均满了（本图左边一副）。再插入关键字 275，就需要拆分第一页，而且要在索引中增加指向新页的链（这里使用新页面最小的关键字 373）（本图中间一副）；当插入 737 时，下面的页面需要拆分，同样需要在索引中增加指向新页的链（本图右边一副）。

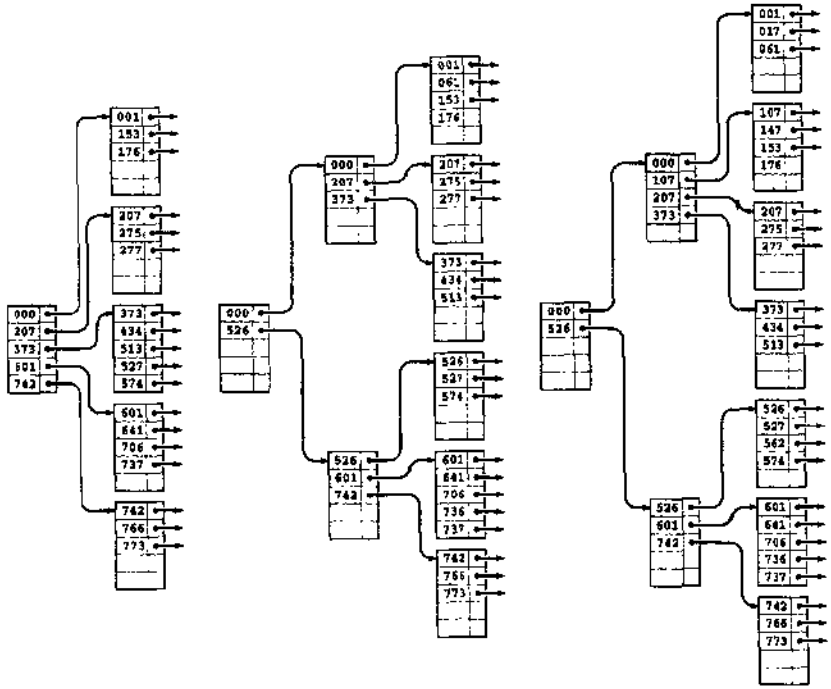



图 16.7　B 树的建立，第三部分

继续前面的例子，我们在图 16.6 的基础上插入 13 个关键字：574，434，641，207，001，227，061，736，526，562，017，107 和 147。在插入关键字，227、526 和 107 的时候，需要进行节点拆分，本图左、中、右三部分分别给出了相应的情况。关键字 526 的插入同时导致了索引页面的拆分，这样使得树高增加 1。

程序 16.1 给出了 B 树节点类型的定义。我们没有具体规定实际应用中可能要求的节点结构，因为实际中可能需要指向特定磁盘页面的索引。为了清楚起见，我们只使用一种节点类型，每个节点由记录的数组构成，每个记录含有一个关键字、一个元素和一个链接。每个节点还含有一个统计有效记录总数的字段。我们不引用内部节点的元素和外部节点的链接；我们在树上元素的范围内不引用其关键字。在具体应用中，我们可以选择使用 union 或者派生类等数据结构实现节省空间的目的。另一方面，我们可以在树中使用指向元素的链来代替关键字以实现时间和空间这对矛盾的折中。这些都要在实际应用中根据不同关键字、元素和链接的情况才能作出决定，并对我们的代码进行相应的修改。

**程序 16.1 B 树节点类型的定义**

B 树中每个节点都含有一个数组和统计数组中有效记录总数的字段。每个数组记录是一个关键字、一个元素和一个指向某节点的链接。对内部节点我们只使用关键字和链接；对外部节点我们只使用关键字和元素。我们通过将记录总数字段赋值为 0 来把新节点初始化为空。

```
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;
```

程序 16.3 给出了 B 树的插入操作的实现；这里也采用了前面第十三章和第十五章中大量采用的递归算法。但这里是自下而上的实现，因为我们在每次递归调用后再检查是否需要拆分节点，因此第一个要拆分的节点是外部节点。拆分需要我们在节点的双亲中加入一个指向新节点的链，这可能导致双亲节点的拆分且继续向上插入新节点，以此类推，直到树根节点（若树根节点需要拆分，我们构造一个新树根，它含有两个孩子）。相反，程序 13.6 给出的 2-3-4 树在递归调用之前检查是否需要节点的拆分，因此它是自上而下的实现。当然，我们对 B 树同样可以实现自上而下算法（见练习 16.10）。在许多 B 树的应用中采用自上而下算法还是自下而上算法差别并不大，因为这类树是非常平的。

程序 16.2 给出了 B 数的搜索操作代码，而且使用前面给出的 B 树节点类型的定义。对外部节点，我们从节点数组中寻找与搜索关键字相匹配的关键字，如果找到就返回相应元素，若搜索失败就返回一个空元素。对内部节点，我们从节点数组中寻找指向可能含有搜索关键字的惟一子树的链接。

**程序 16.2 B 树的搜索**

这里 B 树的搜索跟我们前面常用的一样，也是基于递归函数。对内部节点（树高为正数），我们通过遍历首先找到比搜索关键字大的第一个关键字，然后在其子树上按前面的链接的指向进行递归调用。对外部节点（树高为 0），我们直接遍历比较是否存在与搜索关键字相等的元素。

```
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); }
```

程序 16.4 给出了节点拆分的代码。我们使用的变量 $M$ 取偶数值，而且每个节点对多只有 $M$-1 个元素。这一规定使得我们在对一个节点进行拆分前可以插入第 $M$ 个元素，这样不用太多的开销就大大简化了代码（见练习 16.20 和 16.21）。简便起见，我们只把每个节点中 $M$ 个元素的上半部分用于本节后面将要提到的分析结果；而且这一简化带来的实际差别也是很小的。在自上而下的实现中我们不用再采用这种方法，因为每个节点总是有空间让你插入新节点。

**性质 16.2** *在阶数为 $M$、元素数目为 $N$ 的 B 树中，一次搜索或插入操作需要的探测次数介于 $\log_M N$ 和 $\log_{M/2} N$ 之间——且在实际应用中为常数。*

**程序 16.3 B 树的插入**

与插入排序一样，在插入新元素前先将大于它的元素右移一位。如果插入填满了节点，则调用 split 函数将节点拆分成两半，并在其双亲节点中插入指向新节点的链，这个插入同样可能引发拆分操作，并一直向上传递到树根节点。

```
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++].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++;
      }
```

**程序 16.4 B 树节点的拆分**

在 B 树中拆分一个节点，我们要生成一个新节点，将旧节点中大的一半关键字移到新节点中，然后调整计数字段的值并对节点中央的标志位赋值。这里我们认为 M 是偶数，并在每个节点中使用一个额外的位置保存导致拆分发生的元素。也就是说，每个节点的关键字数目最大为 M-1，而且当一个节点含有 M 个元素时就要把它拆分成两半，每个新节点含有 M/2 个关键字。

```
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;
  }
```

这一性质的根据是 B 树内部的节点（非树根节点和非外部节点）含有链的数目介于 $M/2$ 和 $M$ 之间，因为这类节点都是由 $M$ 个关键字的节点拆分而成的，而且它们只能在上述幅度范围内增加关键字的数目（即某一节点下方的节点发生拆分）。在最好情况下，这些节点构成了一棵度数为 $M$ 的完全树，并达到性质 16.2 所给出的下界。在最坏情况下，我们得到的则是度数为 $M/2$ 的树。■

当 $M$ 取 1000 时，若 $N$ 小于 125 百万，则树高不超过 3。在一些典型情况下，我们可以把树根节点保存在内存中以使得开销减为 2 次探测。对于某些在磁盘上搜索的应用，在进行非常多次搜索之前首先要完成这一步；在带有缓存的虚拟内存系统中，由于树根节点被访问的次数最多，所以要把它保存在最快的存储器中。

对一个搜索应用来说，在一个大文件中进行搜索和插入操作所需探测数目为 2 次几乎是我们的极限了，B 树被广泛应用的原因就在于它可以达到这一理想情况。当然，获得速度和灵活性的同时也要付出一定代价，那就是节点中未被占用的空间，这往往是由大文件造成的。

**性质 16.3**　*阶数为 $M$，由 $N$ 个随机关键字组成的 B 树，含有页面数的期望值为 $1.44N/M$。*

Yao 采用数学分析的方法在 1979 年得出这一结论（参见“第四部分参考资料”），当然这部分数学内容是超出本书范围的。其推导是基于一个描述树增长的统计模型。在前 $M/2$ 个节点被插入之后，在任意给定时间，共有 $t_i$ 个外部页面并含有 $i$ 个元素，并且对 $M/2 \le i \le M$，有 $t_{M/2}+\ldots+t_M=N$。由于任意两个节点间隔以近似相等的概率接受随机关键字的插入，所以一个节点含有 $i$ 个元素的概率是 $t_i/N$。下面对这一概率分几种情况深入分析：若 $i<M$，则它等于含有 $i$ 个元素的外部页面数目减少一半同时含有（$i+1$）个元素的外部页面数目增加一倍的概率；若 $i=2M$，它等于含有 $2M$ 个元素的外部页面数目减少一半同时含有 $M$ 个元素的外部页面数目增加两倍的概率。这种随机过程被称为马尔可夫链（Markov chain）。Yao 的结论就是基于对马尔可夫链的性质的分析。■

我们可以写一段程序仿真这一随机过程来验证性质 16.3（见练习 16.11 和图 16.8、图 16.9）。当然，我们也可以只建立一棵随机的 B 树并考察它的结构性质。用随机过程仿真比数学分析和完全实现这种数据结构都来得简单，而且仿真这种手段对我们的学习和算法比较都是一种重要的工具（练习 16.16 给出了一个例子）。

其他符号表操作与我们前面介绍的基于树的实现是类似的，我们留作练习（见练习 16.22 到 16.25）。其中，选择和排序操作较简单，而删除操作更具挑战性。不妨将插入与删除比较一下。多数删除操作都是简单的外部页面上的元素并使元素数目字段减 1，但我们如何在一个含有 $M/2$ 个元素的节点中进行删除呢？一种自然的方法是用它的兄弟节点的元素来填充删除后的空间（可能使得节点数目减半），但实现起来很复杂，因为我们要相应地

寻找与我们所移动的元素相关联的关键字。在实际情况下，我们可以采用很简单的办法使得节点保持非满状态，且对算法性能的影响很小（见练习 16.25）。

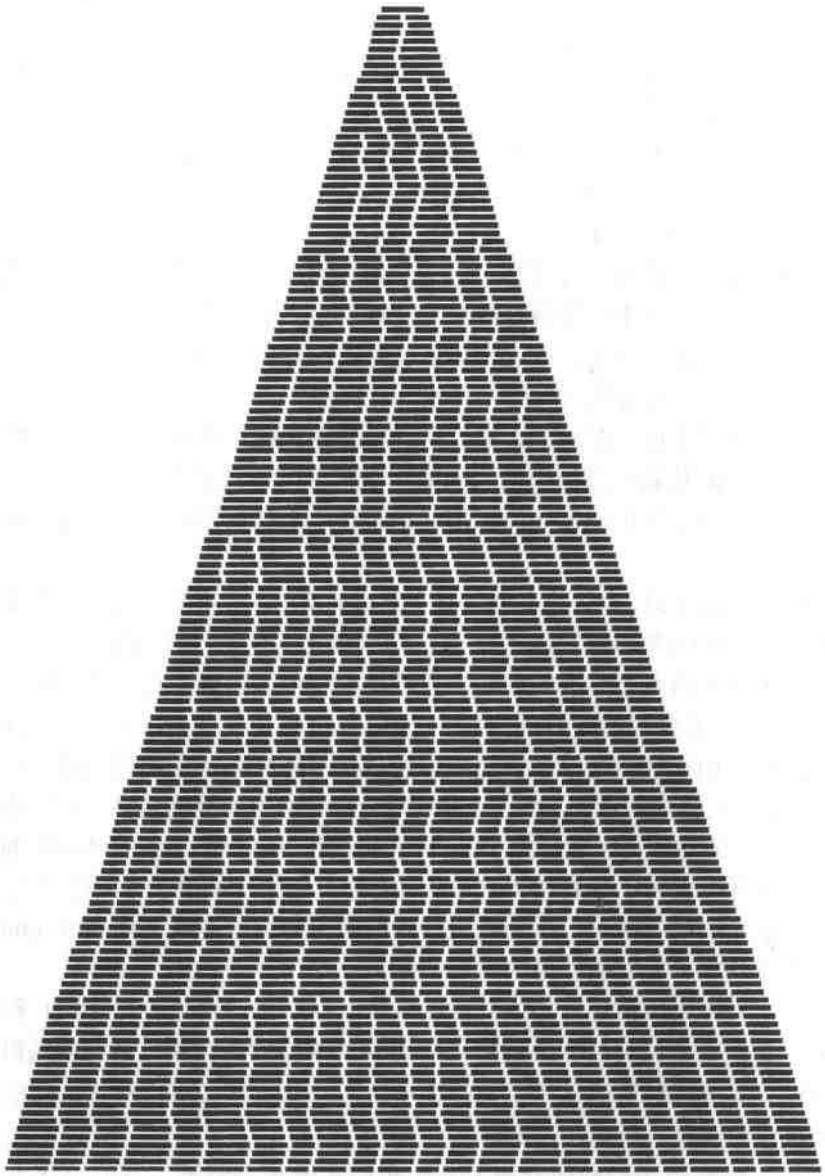


**图 16.8 一棵大型 B 树的生长过程**

在仿真中，我们将含有随机关键字的元素插入原本空的 B 树中，这里每个页面可容纳 9 个关键字和链。图中的行代表外部节点，一个小线段代表一个外部节点，而且线段长度与节点含有的元素数目成比例。大多数插入操作发生在不满的节点上，并使得节点的大小加倍。当一次插入发生在一个满的外部节点上时，该节点拆分为两半

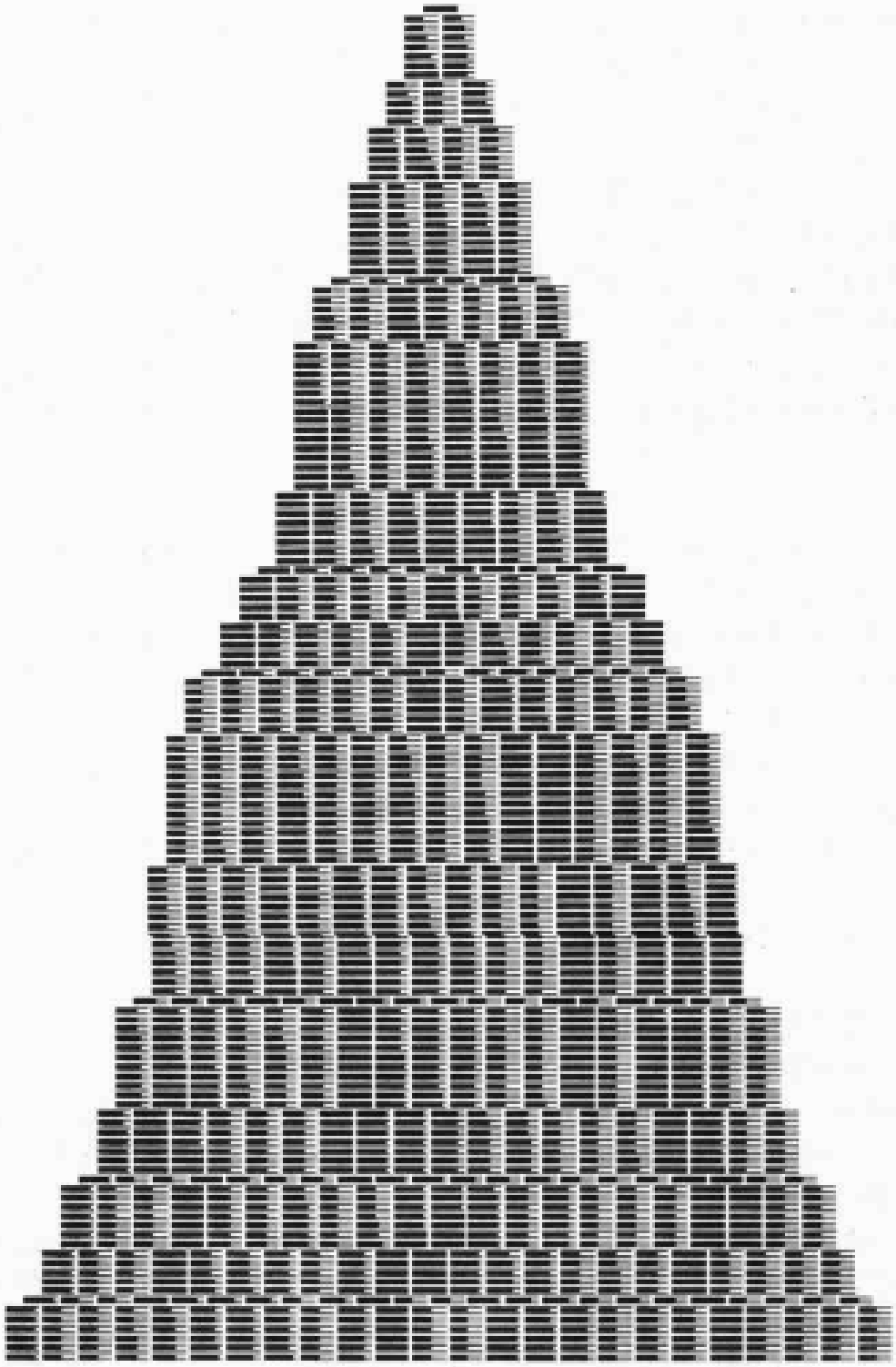


图 16.9　一棵大型 B 树的生长过程，页面占用情况

本图是图 16.8 的另一版本，它显示了在 B 树生长过程中页面是如何逐渐填满的。同样，插入多发生在非满的页面，并使得页面的占用度加倍。当插入发生在一个满的页面时，该页面便拆分成两个半满的页面。

基于基本的B树有很多变形。其中一类是把尽可能多的页面引用保存在一个内部节点中，这样节省了时间开销，但分支因子增大，也使得树更平。正如我们已经讨论的，这些改进在现代的操作系统中带来的好处并不明显，因为我们采用标准的参数值就可以实现一次搜索或插入只需 2 次探测——这几乎是我们的极限了。另一类变形可以节省存储空间，它是在拆分之前把兄弟节点组合起来。练习 16.13 到 16.16 就是基于这种思路，对于随机关键字节省了 23%~44%的存储空间。与前面几章一样，在选择不同算法的时候，要针对不用应用的特点做出相应的分析。B 树适用于很多情况，但这里我们不详细介绍了。同时，由于要考虑到不同软硬件平台的情况，故其具体的实现也不在我们讨论范围之列。深入钻研到具体的实现并不符合本书的主旨，而且我们给出的基本算法已经具备较好的性能，所以我们也不去讨论各种操作系统中如何实现 B 树的代码，而且这些代码往往是不兼容的。

## 练习

▷ 16.5 将给定的关键字按顺序插入空的 3-4-5-6 树中，请给出结果。关键字为：EASYQUESTION WITHPLENTYOFKEYS。

○ 16.6 对 $M$=5，将关键字 516，177，143，632，572，161，774，470，411，706，461，612，761，474，774，635，343，461，351，430，664，127，345，171 和 357 按顺序插入空的树中，请根据图 16.5 到 16.7 画图给出插入的过程。

○ 16.7 对每个 $M$>2 的值（$M$ 取 3、4、5 ——译者注），请给出在练习 16.6 中插入关键字后树高的值。

16.8 对 $M$=4，请画出将 16 个相等的关键字插入空的 B 树后的结果。

● 16.9 将关键字 EASYQUESTION 按顺序插入空的 1-2 树中，请画出结果。并解释 1-2 树作为平衡树并不常用的原因。

16.10 请采用类似于 2-3-4 树插入的算法（程序 13.6）修改 B 树的插入程序，即程序 16.3，使其沿单路向下拆分。

● 16.11 请编写程序将 $N$ 个随机关键字插入阶数为 $M$ 的空 B 树中，要求使用性质 16.3 后面介绍的随机过程。对 $M$=10，100 和 1000 和 $N$=$10^3$，$10^4$ 和 $10^5$，$10^6$，运行你的程序。

○ 16.12 假设有这样的三层树：我们可以在内存中保存 $a$ 个链，页面中数目介于 $b$ 到 $2b$ 之间的链表示内部节点，数目介于 $c$ 到 $2c$ 之间的元素表示外部节点。请问：作为 $a$、$b$ 和 $c$ 的函数，该树可容纳元素数目的最大值是多少？

○ 16.13 这里我们考虑 B 树的一种改进，即兄弟拆分法（或叫做 $B^*$树）。当节点中记录数达到 $M$ 要进行拆分时，我们把该节点与其兄弟节点组合。如果其兄弟节点含有 $k$ 个记录且 $k$<$M$−1，我们将两者的记录重新分配，每个得到$(M+k)/2$ 个。否则（即 $k$=$M$ − 1——译者注），我们就构造一个新节点，并把全部记录分配到三个节点中，每个分到大约 $2M/3$ 个记录。同样，我们可以允许树根节点含有 $4M/3$ 个记录，当达到这个数目时就构造一个新树根以把原来的拆分。请讨论对 B*树进行一次插入或插入操作所需的探测次数。并对 $M$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 的不同取值，把你的结果与 B 树的结果（见性质 16.2）进行比较。

●●16.14　请实现 B*树的插入操作（基于前面介绍的兄弟拆分法）。

● 16.15　仿照图 16.8，请画出兄弟拆分法的图。

● 16.16　对 $M$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 的不同取值，采用兄弟拆分法，通过往空树中插入随机关键字构造阶数为 $M$ 的 B*树，请使用随机过程的仿真程序（见练习 16.11）来求出所使用的平均页面数。

● 16.17　请编写一个建立自下而上的 B 树索引的程序，索引起始于指向页面的数组，这里的页面含有 $M$ 到 $2M$ 个元素，且排列是有序的。

● 16.18　我们在课文中给出的 B 树插入算法（程序 16.3）可以对所有页面均满的情况（比如图 16.2 的情况）建立索引吗？请对你的回答给以解释。

16.19　假设很多不同的计算机可以访问同一索引，因此可能出现同一时刻几个程序同时往一棵 B 树中插入新节点的情况。我们选择自上而下的算法而不用自下而上的算法，请解释其原因。假定每个程序在读取某个节点时可以屏蔽其他程序对该节点的修改，并将此修改延迟进行。

● 16.20　请修改程序 16.1 到 16.3，使得每个节点可以容纳 M 个元素。

▷ 16.21　对 $N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 的不同取值，请列表表示出 $\log_{999}N$ 和 $\log_{1000}N$ 的差别。

▷ 16.22　对基于 B 树的符号表实现排序操作。

○ 16.23　对基于 B 树的符号表实现选择操作。

●●16.24　对基于 B 树的符号表实现删除操作。

○ 16.25　对基于 B 树的符号表实现删除操作，并采用下述简单的方法：在外部页面上删除一个给定元素（允许页面容纳元素数目少于 $M/2$），除了被删除元素为这一页最小元素的情况外，不改变该节点上面的树的状态。

● 16.26　使用节点内的二分搜索，请修改程序 16.2 和 16.3。对 $N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 的不同取值，将 $N$ 个随机关键字插入空表中以构造符号表，请使得时间开销达到最小值的 $M$ 分别是多少？

## 16.4　可扩充散列

将数字搜索算法应用于外部搜索除了使用 B 树还有另外一种算法，它是由 Fagin、Nievergelt、Pippenger 和 Strong 在 1978 年提出的，称为可扩充散列。这种方法只需一到两次探测就可以完成搜索。同样，在插入操作中也只需一到两次探测。

可扩展散列综合了散列、多叉线索和顺序访问等算法的特点。与第十四章中的散列一样，可扩充散列是一种随机化算法——其第一步是求出能将关键字转化为整数的散列函数（见 14.1 节）。在本节中为了简单起见，我们只考虑关键字为随机的定长数字串的情况。可扩充散列与多叉线索（见第十五章）类似之处在于搜索开始时，先使用关键字的前几位索引到一个长度为 2 的幂的表中。与 B 树相似之处是可扩充散列当页面填满时要拆分成两个新页面。同时，可扩充散列建立一个目录，使得我们可以找到与搜索关键字相匹配的元素所在的页面，这是它与顺序访问算法的相同点。可扩充散列将几种我们熟悉的算法糅合在一起，在我们对搜索算法的研究中是一种较好的结果。

假设我们可用的磁盘空间是 2 的某次幂，不妨设为 $2^d$。这样，我们可以建立一个指向 $2^d$ 个不同页面的索引目录，并使用关键字的 $d$ 位作为目录中的索引。图 16.10 给出了这一过程。与 B 树中一样，我们保持页面中的元素有序，并且在与搜索关键字相对应的页面中进行顺序搜索。

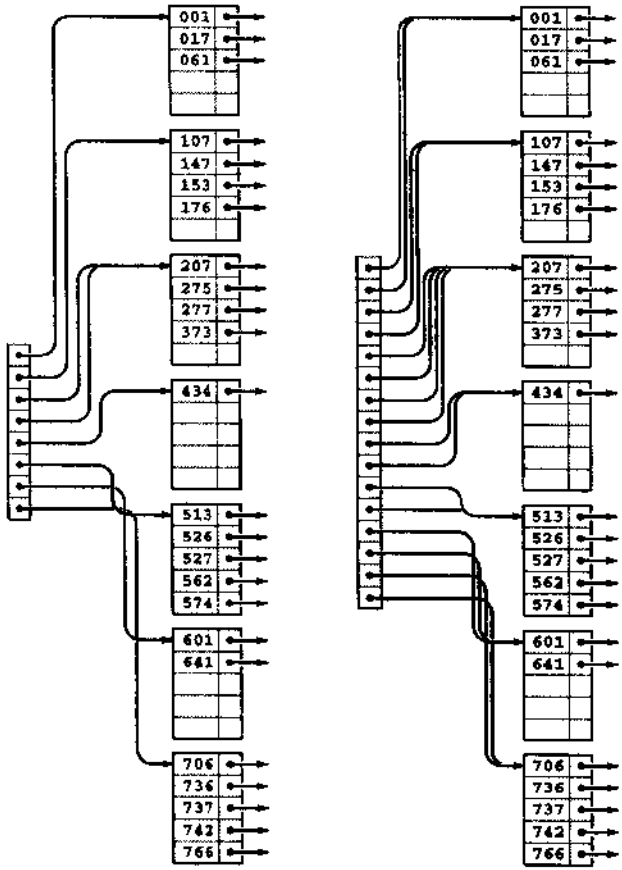



图 16.10 目录页索引

这里的目录含有 8 条记录，通过将前三位与相应页面匹配的关键字存储在页面中，我们总共可以保存 40 个关键字，并且可以通过目录中的指针来访问各页面（见左图）。目录中的 0 记录含有指向所有以 000 开头的关键字所在页面的指针；1 记录则含有指向所有以 001 开头的关键字所在页面的指针；2 记录含有指向所有以 010 开头的关键字所在页面的指针，以此类推。如果某些页面元素数目少，我们可以减少一些页面，并采用多个指针指向一个页面。在本例中（见左图），373 在以 2 开拓的关键字所在的页面上；同时，该页就被定义为含有所有以 01 开头的关键字。

如果我们使目录长度加倍并复制每一个指针，我们就得到可以用关键字的前 4 位作索引的数据结构（见右图）。例如，图中最后一页定义为含有所有以 111 开头的元素，而且如果搜索关键字是 1110 或 1111，我们通过目录指针都可以访问到该页面。这个大的目录可以容纳更长的表。

图 16.10 解释了可扩充散列的两个基本概念。第一，我们无需保存 $2^d$ 个页面。也就是说，我们可以使多个指针指向同一页面，同时不会影响搜索的速度。我们把具有不同前 $d$ 位值的

关键字保存在同一页面，对给定关键字仍然可以通过目录中的指针访问到其相应元素所在的页面。第二，随着表长的不断增大，我们可以采用加倍目录长度的方法来容纳更多的元素。

**程序 16.5　可扩充散列的数据结构**

一个可扩充散列表是一个指向页面的指针的目录，这里的页面（类似于 B 树中的外部节点）最多含有 $2M$ 个元素。每个页面包含一个统计本页面记录个数的字段（$m$）和一个用来区别本页内所有关键字所需位数的整数（$k$）。与前面一样，N 用来表示整个表中元素的个数。变量 $d$ 代表我们用来索引的关键字的位数，而 $D$ 则是目录中记录的个数，从而 $D=2^d$。表初始化为长度是 1 的目录，其指针指向一个空页面。

```
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;
        }
    };
```

我们用于可扩充散列的数据结构比 B 树的简单很多。该结构包括元素数目不超过 $M$ 的页面和一个含有 $2^d$ 个指针的目录（见程序 16.5）。目录中第 $x$ 条记录的指针指向含有关键字前 $d$ 位的值等于 $x$ 的所有元素的页面。我们取的 $d$ 足够大以使得每个页面的元素数目都不超过 $M$。可扩充散列的搜索很容易实现：用关键字的前 $d$ 位在目录中索引，便找到含有与搜索关键字相匹配的元素的页面，然后在该页面中进行顺序搜索（见程序 16.6）。

**程序 16.6　可扩充散列的搜索**

在可扩充散列表中的搜索相当于简单地使用关键字的前几位在目录中索引，然后再与搜索关键字对应的页面中完成顺序搜索。惟一的要求是：目录的每条记录指向一个页面，该页面一定包含以相应的确定位开头的全部关键字。

```
  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); }
```

实现插入操作所需要的数据结构稍微复杂一点，但所需的搜索完全可以用上面的程序而无需任何修改。要实现插入，我们首先要提出下列问题：

- 如果某页的元素数目超过该页的容量，我们该怎么办？
- 我们该选择多大的目录长度才合适？

让我们举个例子来看，在图 16.10 中我们不能取 $d$=2，因为会出现某些页溢出（即页面的元素数目超过其容量——译者注）的情况；我们也不取 $d$=5，因为那样有太多空页面。跟往常一样，我们对符号表这一抽象数据类型感兴趣，因此在建表时搜索和插入操作是交替进行的。考虑到这一点，我们不妨提炼一下前面提出的两个问题：

- 如果需要往一个满页面中插入元素我们该怎么办？

比如图 16.10 中，我们不能再插入以 5 和 7 开头的元素，因为那些相应页面已经满了。

**定义 16.3** 阶数为 $d$ 的可扩充散列表是一个含有 $2^d$ 个页面引用的目录，这里的页面含有的元素数目不超过 $M$。每个页面上的元素以其关键字前 k 位相区别，目录含有 $2^{d-k}$ 个指向页面的指针，它起始于页面上关键字的前 $k$ 位决定的位置。

某些 $d$ 位组合在关键字中根本不会出现，尽管我们有简单的将指针指向空页面的方法，但在定义 16.3 中，我们不规定目录中的相应记录；我们仅仅简单讨论一下这个问题。

随着表中元素增多，我们需要完成两种基本操作：一是页面拆分，即把满的页面上的一些关键字分配到其他页面；二是目录拆分，就是使目录长度加倍并使 $d$ 加 1。特别是，当一个页面满了时，我们将它拆分成两个页面，并根据关键字最左边不同的位决定哪些要转到新页面中去。在拆分页面的同时，我们要调整目录中的指针，同时要在需要的时候使目录长度加倍。

与前几章一样，理解可扩充散列算法的最好方法是考察将一组关键字插入空表的过程。如果我们所提到每种情况在插入过程都会出现，那么很快我们就会理解这一算法的基本原理了。图 16.11 到图 16.13 给出了建立可扩充散列表的过程，关键字是本章中我们一直在用的 25 个八进制关键字。如在 B 树中一样，多数插入操作都是平凡的：仅仅是简单地在页面中加入一个新关键字。由于我们最初只有一个页面而最后结果是八个页面，可以得出有七次插入操作导致了页面拆分；另一方面，由于我们起始于目录长度为 1，而最后结果是

目录长度为 16，可得 4 次插入操作导致了目录拆分。

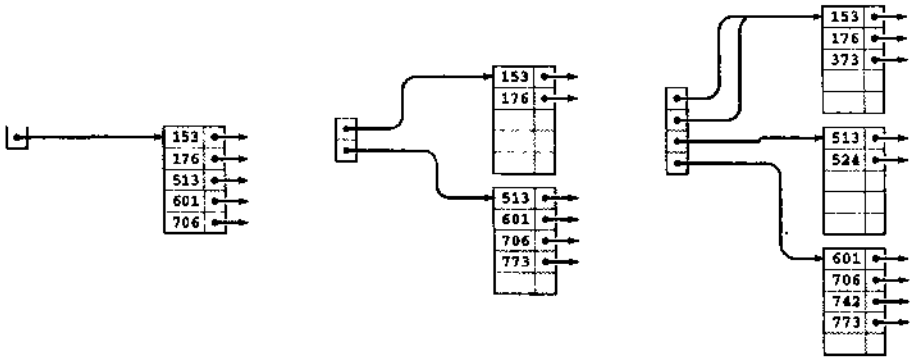



**图 16.11　可扩充散列表的建立，第一部分**

类似 B 树，前五次都是插入到一个页面（见左图），然后当我们插入 737 时，需要页面拆分（一个页面含有所有以 0 开头的关键字，另一个页面含有所有以 1 开头的关键字），同时要使目录长度加倍以使得有新指针指向两个页面（见中图）。然后插入 742 到下面一页（因为它以 1 开头），再插入 373 到上面一页（因为它以 0 开头），当插入 524 时，我们需要对下面一页进行拆分。这次拆分，我们把所有以 10 开头的关键字和以 11 开头的关键字分别保存在两个页面，同时要再次使目录长度加倍以容纳更多的指针（见右图）。此时目录中有两个指针指向同一页面，即以 0 开头的关键字所在页面，这两个指针中一个对应于所有以 00 开头的关键字，另一个对应于所有以 01 开头的关键字。

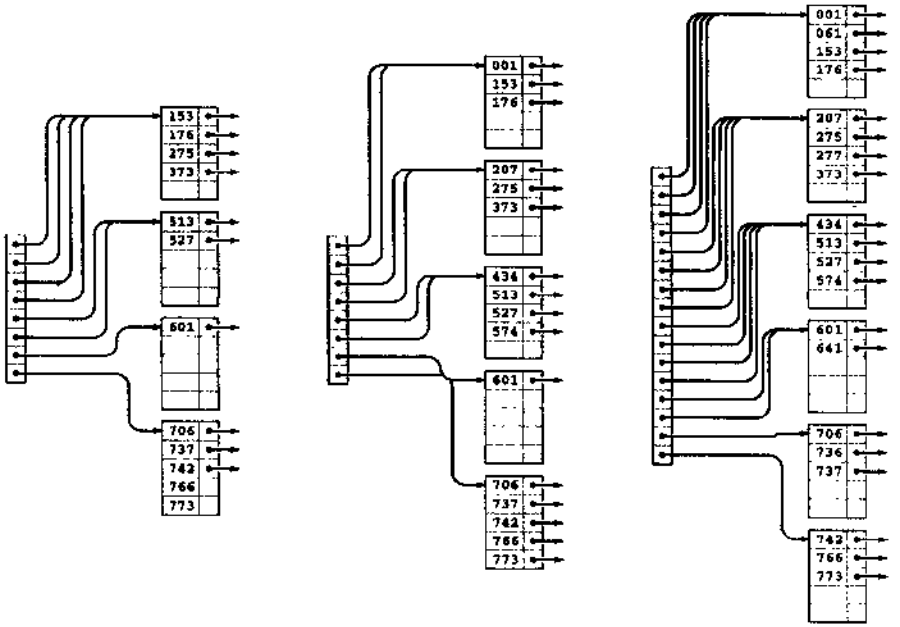



**图 16.12　可扩充散列表的建立，第二部分**

我们在图 16.11 最右边的树中插入关键字 766 和 275 时不出现拆分的情况（见左图）。然后当我们需要插入 737 时，最下部的页面需要拆分，而且由于这一页只有一个指针，故目录也需要拆分（见中图）。然后我们插入关键字 574，434，641 和 207，最后插入 001 时又导致顶部的页面拆分（见右图）。

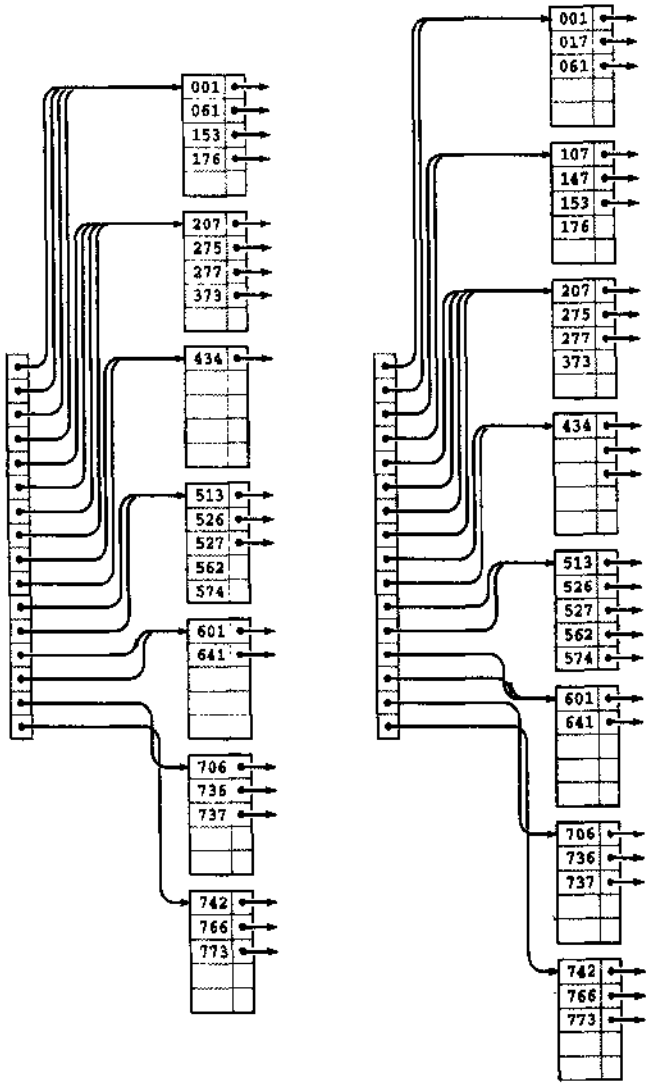



**图 16.13 可扩充散列表的建立，第三部分**

接着图 16.11 和图 16.12 的例子，我们再插入 5 个关键字：526，562，017，107 和 147。当我们插入 526 和 107 时分别出现了节点拆分的情况（对应于左图和右图）。

**性质 16.4** 可扩充散列表的结构仅与构成该表的关键字有关，而与关键字的插入顺序无关。

不妨回头看一下与关键字有关的线索（见性质 15.2），它的每个内部节点都有一个保存其子树所含元素的个数的标志。当且仅当线索的内部节点标志小于 $M$ 而其双亲节点的标志大于等于 $M$ 时，线索的内部节点与可扩充散列表的页面是相对应的。即节点下面所有的元

素都在对应的页面中。如果某个节点在第 $k$ 层，它对应于从树根到该节点的路径决定的 $k$ 位数字的一个组合，同时可扩充散列表中对应于这一 $k$ 位组合的指针正是指向与该节点所对应的页面。因此，我们不用考虑关键字插入的顺序就可以把一个线索转化成一个可扩充散列表，这一性质可以看作是性质 15.2 的一个推论。■

程序 16.7 给出了可扩充散列表的插入操作的实现。第一步，我们通过惟一的页面索引找到可能含有插入关键字的页面。然后，我们在该页面上插入关键字，这与我们在 B 树中对外部节点的操作一样（见练习 16.2）。如果插入导致页面中元素数目达到 $M$，则调用拆分函数，这也与 B 树一样，但这里的拆分函数复杂很多。每个页面都保存有该页面所有关键字相同的位数 $k$，而且由于我们从 0 开始计数，故第 $k$ 位也是拆分时需要比较以划分这些元素的那一位。

在拆分时，我们先建一个新页面，然后将比较位是 0 的元素保留在旧页面而将该位是 1 的元素移到新页面，然后将两个页面的计数字段加 1，即保存 $k$+1。这种方法在应用中会出现所有关键字第 $k$ 位都相同的情况，我们无法达到拆分的目的。如果这样，我们就继续比较下一位，直到可以使得每个页面至少含有一个元素。除非对同一关键字我们有 $M$ 个值，否则这一过程最后一定会终止。我们简要讨论这一情况。

在 B 树中，我们在每一页都留出一个记录空间使得拆分操作可以在插入后进行，这样简化了程序代码。在这里，我们同样采取这种技巧，同时在分析时忽略其影响。

**程序 16.7　可扩充散列表的插入操作**

在可扩充散列表中插入新元素首先要进行搜索；然后在搜索所确定的页面进行插入；如果插入会导致溢出，我们还要拆分页面。插入的基本方法与 B 树是一样的，但搜索和拆分算法有所不同。拆分函数生成一个新节点，然后比较被拆页面所有关键字的第 $k$ 位（从左边数）：如果这一位是 0，则元素留在原页面；如果是 1，则元素移到新页面。拆分完成后，$k$+1 的值被保存到“关键字第一个不同位的位置”字段。我们要重新拆分直到使得每个节点至少含有一个元素为止。最后，在目录中添加指向新页面的指针。

```
  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); }
```

**程序 16.8 可扩充散列表的目录的插入操作**

这段简单的代码是可扩充散列进程的核心。给定链接 t，指向携带前 k 位匹配的数据项的节点，该数据项将被插入目录。最简单的情况是 d 和 k 相等，则只把 t 赋于 d[x]，x 是 t->b[0]（和页面中的其他新数据项）的前 d 位的值。如果 k>d，则需使目录大小加倍，直到 d=k。如果 k<d，则需要增加指针，第一个 for 循环计算需要被设置为（$2^{d-k}$）的指针数，第二个 for 循环设置指针。

```
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;
  }
```

当创建一个新页面时，我们要在目录中插入一个指针指向新页面。程序 16.8 给出了这段代码。最简单的情况是在插入前，有两个指针指向被拆分的页面。此时，我们只需把第二个指针指向新页面即可。如果在页面上我们用来区分不同关键字的位数 $k$ 大于访问目录所需的关键字位数，我们就不得不增大目录长度以容纳更多的记录。然后还要相应的更新目录中的指针。

如果多于 $M$ 个元素有重复元素，这个表将会溢出，而且程序 16.7 在寻找关键字不相同的位的时候会进入死循环。如果关键字有过多的位相同，会出现一个相关的问题，就是目录会变得很大。我们可以拿这种情形与 MSD 基数排序做比较，当文件中有很多重复关键字或者关键字中相同位较多时，排序需要相当多的时间。这两种数据结构这种情形下是一致的。我们可以通过散列函数的随机化作用来避免此类问题（见练习 16.43）。即使在散列算法中，由于散列函数对相同的输入给出相同的散列值输出，所以如果出现大量重复关键字的情况，也需要额外的措施；而且如果相等的关键字多于一个页面的容量，整个算法将会崩溃。所以，我们在使用前面这些的代码前要增加检测措施以避免上述情况的出现（见练习 16.35）。

体现可扩充散列算法性能的基本参数是页面数（这一点与 B 树一致）和目录长度。算法所需的随机化由散列函数提供，故对 $N$ 次不同关键字的插入，我们可以得到平均情况下的算法性能。

**性质 16.5** *对 $N$ 个元素的文件，可扩展散列算法平均需要 $1.44(N/M)$ 个页面，这里 $M$ 是每个页面所能容纳的元素个数。目录中记录数目的期望值为 $3.92(N^{1/M})(N/M)$。*

这个（深奥的）结论是对前一章我们所简要讨论的线索算法性能的推广（参见“第四部分参考资料”）。结论中使用的常数分别为：$\lg e=1/\ln 2$ 和 $e\lg e=e/\ln 2$，而且这些值的精确结果是在其平均值附近摆动的。我们不应对这一结果感到惊讶，因为例如目录长度是 2 的次幂的这一事实我们已经考虑在结果中了。■

注意到目录长度增加的速度比 $N$ 增加的线性速度要快，特别是对 $M$ 小的情况更是如此。对实际应用中的 $M$ 和 $N$ 的取值，$N^{1/M}$ 非常接近 1，因此实际应用中我们可以期望目录长度为 $4(N/M)$。

我们把目录组织成指针的一维数组。我们可以把目录保存在内存中，或者如果它太大，可以把树根节点保存在内存，该节点指向目录所在的页面，当然也是采用相同的索引算法。如果不采用这种方法，我们可以增加一层数据，对第一层数据的前 10 位（比如说）索引，然后对第二层数据剩下的位进行索引（见练习 16.36）。

跟前面 B 树那节一样，我们把其他符号表操作留作练习（见练习 16.38 到 16.41）。同样，这里的删除操作也相对复杂，但允许页面非满在很多应用中都会起作用的，这一点也可以参考 B 树中论述的部分。

## 练习

▷ 16.27　如果我们在图 16.10 的数据结构中使用长度为 32 的目录，会有多少页面是空的？

16.28　对 $M$=5，将关键字 562，221，240，771，274，233，401，273 和 201 按顺序插入空的树中，请根据图 16.11 到 16.13 画图给出插入的过程。

○ 16.29　对 $M$=5，将关键字 562，221，240，771，274，233，401，273 和 201 按顺序插入空的树中，请根据图 16.11 到 16.13 画图给出插入的过程。

○ 16.30 假设给你一个元素有序的数组，对这组元素，请叙述如何确定可扩充散列表所用的目录长度。

● 16.31 请编写程序，用一组给定的有序数组建立一个可扩充散列表，要求完成下列两步：一是确定目录长度（见练习 16.30），二是在页面中分配元素并填充目录。

○ 16.32 请给出一组关键字，使得它对应的可扩充散列表的目录长度为 16，且指向一页的指针有 8 个。

●●16.33 请根据图 16.8 画出可扩充散列的图。

● 16.34 请编程计算将 $N$ 个随机关键字插入空树中以构造可扩充散列表所需的平均外部页面总数和平均目录长度，这里每个页面的容量为 $M$ 个元素。对 $M$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 的不同取值，再计算出空位置所占的百分比。

16.35 为了防止出现重复关键字过多或者关键字相同的位过多带来的问题，请对程序 16.7 加以修改，加入适当的检测措施。

● 16.36 请采用两层目录的手段对程序 16.5 到 16.7 进行修改，且每个目录节点的指针数目不能超过 $M$。要求特别注意当目录由一层变为两层的时需采取的措施。

● 16.37 请修改程序 16.5 到 16.7，使得在该数据结构中实现每个页面可容纳 $M$ 个元素。

○ 16.38 对可扩充散列表实现排序操作。

○ 16.39 对可扩充散列表实现选择操作。

●●16.40 对可扩充散列表实现删除操作。

○ 16.41 使用练习 16.25 介绍的方法，对可扩充散列表实现删除操作。

●●16.42 请实现另一版本的可扩充散列算法，要求在拆分目录的同时拆分页面，这样可以使得每个目录指针指向惟一的页面。并请做实验比较你的算法与课文中标准的算法。

○ 16.43 对 $M$=10、100 和 1000，且 $1 \le d \le 20$，请做实验研究在我们找到多于 $M$ 个数有相同的前 $d$ 位之前，我们需要生成多少个随机数。

● 16.44 使用表长为 $2M$ 的散列表，请修改采用链地址法的散列算法（见程序 14.3），将元素保存在长度为 $2M$ 的页面中。即当一个页面满了时，把它链接到一个新页面，因此每个记录指向一个连接在一起的页面链。对 $M$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 的不同取值，请做实验确定用 $N$ 个随机关键字建立的这种表完成一次搜索所需的平均探测次数。

○ 16.45 取 $2M$ 作为页面长度，请按照把访问满的页面作为“冲突”的思路修改双重散列算法（见练习 14.6）。对 $M$=10，100 和 1000，$N$=$10^3$，$10^4$，$10^5$ 和 $10^6$ 的不同取值，请做实验确定用 $N$ 个随机关键字建立的这种表完成一次搜索所需的平均探测次数，初始表长取 $3N/2M$。

## 16.5 综述

本章中所介绍的算法的最重要的应用是为保存在外部存储设备上（例如，一个磁盘文件中）的大型数据库建立索引。尽管我们所介绍的基本的算法是强大的，但基于 B 树或者可扩充散列算法来开发一个文件系统是一件相当复杂的工作。首先，我们不能直接使用本章中的 C++程序——这些代码必须加以修改才能读写不同硬件系统中的磁盘文件。第二，

我们必须保证我们的程序所使用的参数（如页面大小、目录长度等）针对我们使用的特定硬件做好适当地调整。第三，我们要考虑代码的可靠性，而且要加入出错检测和纠正机制。例如，我们要有能力检查某一数据结构是否处于一致的状态，而且要考虑如何应付突然出现的错误。对这类问题进行系统的考虑更为重要，但这超出了本书的范围。

另一方面，如果我们的编程环境支持虚拟内存，我们便可以直接使用本章给出的 C++ 程序来实现很多种大型的符号表应用。粗略地说，每次我们访问一个页面，系统就把该页面放在一个缓存中，而且缓存中的数据访问非常高效。如果我们要访问不在缓存中的页面，则系统需要访问外部存储设备，因此在某种意义上，缓存访问失败与探测次数开销的度量是等价的。

对 B 树而言，每次搜索和插入操作都要访问树根节点，因此树根节点一定在缓存中。否则，对足够大的 $M$，典型的搜索和插入操作要有最多两次缓存访问失败。若缓存空间大，搜索所需的第一页（树根的孩子节点）就在缓存中，因此平均的搜索开销将明显小于两次探测。

对可扩充散列而言，整个目录都保存在缓存中的可能性不大，因此我们期望目录访问和页面访问都有一次缓存访问失败（这是最坏情况）。也就是说，在大型表中一次搜索需要两次探测，一次是访问目录的相应部分，一次是访问相应的页面。

这些算法恰好组成了我们用来结束搜索算法的一个专题，因为要用好这些算法，我们需要掌握二分搜索、BST、平衡树、散列和线索等算法（这些是我们从第十二章到第十五章所讨论的基本搜索算法）的基本性质。作为一个专题，这些算法为解决实际的一大类符号表应用提供了多种方案，而且它们构成了算法科学具有强大力量的极有力的证明。

## 练习

● 16.46　使用 B 树，实现一个符号表，其中包括一个析构函数、一个复制构造函数和一个重载赋值操作符，要求该算法支持对一级符号表 ADT 的构造、统计、搜索、插入、删除和联结等操作，并且支持类成员编号（见练习 12.6 和 12.7）。

● 16.47　使用可扩充散列算法，实现一个符号表，其中包括一个析构函数、一个复制构造函数和一个过载分配算子，要求该算法支持对一级符号表 ADT 的构造、统计、搜索、插入、删除和联结等操作，并且支持类成员编号（见练习 12.6 和 12.7）。

16.48　使用一种抽象数据类型作为页面索引，修改 16.3 节中 B 树的程序（程序 16.1 到 16.3）。

16.49　使用一种抽象数据类型作为页面索引，修改 16.4 节中可扩充散列算法的程序（程序 16.5 到 16.8）。

16.50　对一个典型的缓存系统（即最近访问的 $T$ 个页面保存在内存中），在一棵 B 树中进行 $S$ 次随机搜索，请估算平均每次搜索所需的探测次数。假设 $S$ 远远大于 $T$。

16.51　应用练习 16.50 提供的缓存模型，请估算在一个可扩充散列表中每次搜索所需的探测次数。

○ 16.52 如果你的系统支持虚拟内存，请设计一个实验，在大型符号表中的随机搜索应用中，比较 B 树和二分搜索的性能，并完成你设计的实验。

16.53 请实现一个优先级队列，要求支持大量元素的建立、插入和删除最大值操作（见第 9 章）。

16.54 请基于用跳跃表表示的 B 树实现一个外部符号表（见练习 13.80）。

● 16.55 如果你的系统支持虚拟内存，请做实验研究对用 B 树实现的大型符号表中的随机搜索操作，M 取何值时达到最快搜索。（虽然作这个实验前需要花精力了解你的系统的性能，但这是值得的。）

●●16.56 对表驻留于外部存储设备的应用环境，请修改 16.3 节中 B 树的代码（程序 16.1 到 16.3）。如果你的系统支持非顺序文件访问，将整个表保存在一个（大）文件中，并使用文件中的位移量来代替指针。如果你的系统支持外部文件的直接访问，使用页面地址代替指针。如果你的系统支持上述两种情况，请选择一种最适合大型符号表的方法。

●●16.57 对表驻留于外部存储设备的应用环境，请修改 16.4 节中可扩充散列算法的代码（程序 16.5 到 16.8）。选择一种往文件中分配目录的页面的方法，并请解释你选择该方法的原因（见练习 16.56）。

# 第四部分参考资料

这一部分最基本的参考书作者如下：Knuth；Baeza、Yates 和 Gonnet；Mehlhorn；还有 Cormen、Leiserson 和 Rivest。我们所讨论的很多算法在上述作者的书中都有详细的论述，而且还有数学分析和对实际应用的建议。Knuth 的书中介绍了很多经典的方法，其他书则侧重最近涌现的新方法，当然我们后面还列出了很多文献。这一部分中我们所说的所有“超出本书范围”的内容在我们的参考书和 Sedgewick Flajolet 的书中都有论述。

第十三章中的材料来自下列论文：Roura 和 Martinez 在 1996 年的论文，Sleator 和 Tarjan 在 1985 年的论文，Guibas 和 Sedgewick 在 1978 年的论文。从这些论文发表的年份可以看出，平衡树仍是人们正在研究的课题。本书引用的内容包括红黑树等类似数据结构的性质的详细证明，而且后面还有近期的文献作为参考。

我们在第十五章对线索的处理是很经典的（尽管其 C++实现在文献中很少见）。关于 TST 的材料来自于 Bentley 和 Sedgewick 在 1997 年的论文。

Bayer 和 McCreight 在 1972 年的论文中提出了 B 树，Fagin、Nievergelt、Pippenger 和 Strong 在 1979 年的论文中提出了可扩充散列算法。而 Flajolet 在 1983 年的论文中对可扩充散列给出了分析结果。如果读者希望获得更多的关于外部搜索算法的知识，请务必读一读这些论文。这些算法的实际应用出现在数据库系统中。Date 的书可以作为这一领域（指数据库——译者注）的入门教材。

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. Sedgewick, “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. Sedgewick, "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. Sedgewick 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.

L. Guibas and R. Sedgewick, "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. Sedgewick 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.