HZ BOOKS

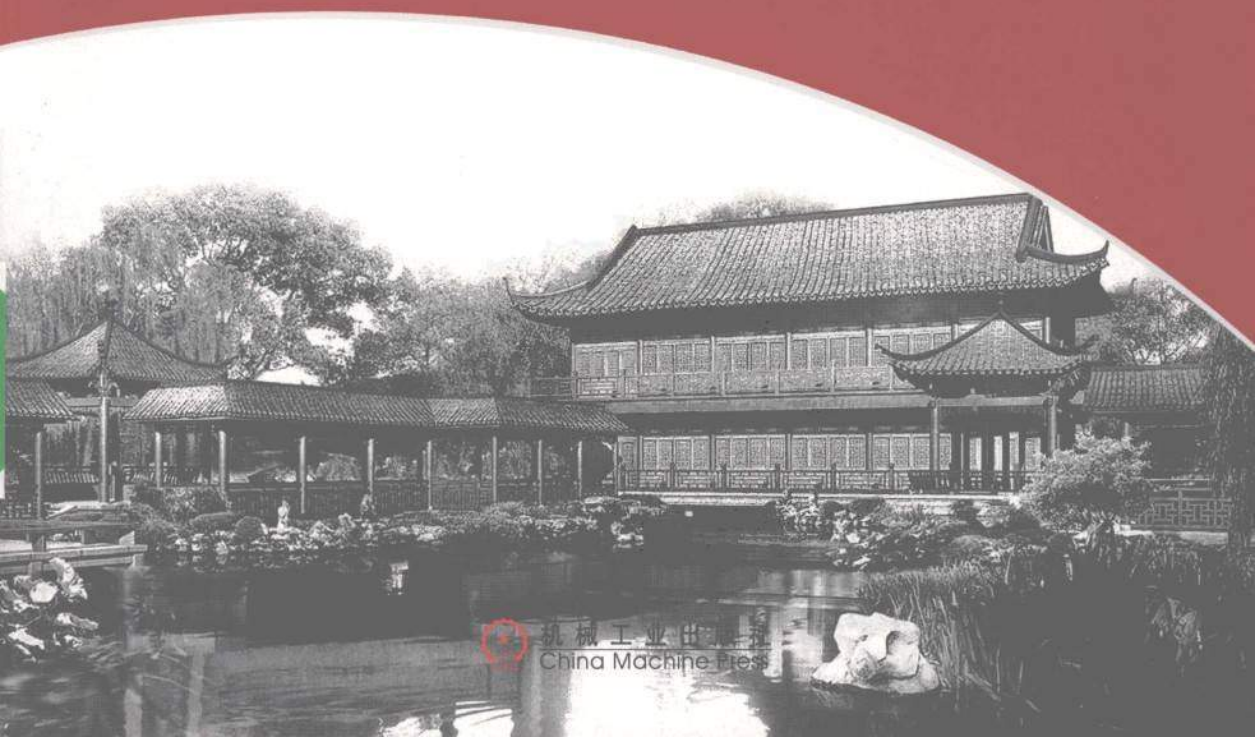
掌握算法的实用精要
写给程序员的算法书

Practical Algorithms for Programmers

程序员实用算法

（美） Andrew Binstock
John Rex 著

陈宗斌 等译



机械工业出版社
China Machine Press

程序员实用算法

如今大多数关于算法的图书都是大学教科书，或者是令人厌倦的相同算法集合改头换面后的作品。本书是给出所有算法的完整代码实现的第一本书，这些算法在开发人员的日常工作中非常有用。

本书重点关注的是实用、立即可用的代码，并且广泛讨论了可移植性和特定于实现的细节。本书作者介绍了一些有用但很少被讨论的算法，它们可用于语音查找、日期和时间例程（直到公元1年）、B树和索引文件、数据压缩、任意精度的算术、校验和与数据验证，并且全面地介绍了查找例程、排序算法和数据结构。

本书只要求读者具有C语言的初级知识以及基本代数的相关知识。源代码经过测试符合ANSI标准，可以运行在UNIX下，以及Borland、Microsoft和Watcom的编译器上。

作者简介

Andrew Binstock是《UNIX Review》的主编和《C Gazette》的创刊编辑。他是《HP LaserJet Programming》（Addison-Wesley，1991）的第一作者。

John Rex是一位计算机顾问，专攻C和C++。他是《C Gazette》的前任技术编辑，并且为许多杂志撰写文章。

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计：杨 林

PEARSON Education

Addison-Wesley
www.pearsonhighered.com

上架指导：计算机/程序设计
ISBN 978-7-111-27296-0

9 787111 272960

定价：65.00元

Practical Algorithms for Programmers

程序员实用算法

（美） Andrew Binstock
John Rex 著

陈宗斌 等译

机械工业出版社
China Machine Press

本书重点关注的是实用、立即可用的代码，并且广泛讨论了可移植性和特定于实现的细节。本书作者介绍了一些有用但很少被讨论的算法，它们可用于语音查找、日期和时间例程（直到公元1年）、B树和索引文件、数据压缩、任意精度的算术、校验和与数据验证，并且还最全面地介绍了查找例程、排序算法和数据结构。

本书结构清晰，示例丰富，可作为广大程序员的参考用书。

Authorized translation from the English language edition entitled Practical Algorithms for Programmers by Andrew Binstock and John Rex, published by Pearson Education, Inc, publishing as Addison Wesley ISBN (978-0-201-63208-X), Copyright © 1995 by Andrew Binstock and John Rex.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanic, including photocopying, recording, or by any information storage retrieval system, without permission of Pearson Education, Inc.

Chinese simplified language edition published by China Machine Press.

Copyright © 2009 by China Machine Press.

本书中文简体字版由美国 Pearson Education 培生教育出版集团授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2008-1753

图书在版编目（CIP）数据

程序员实用算法/（美）宾斯托克（Binstock，A.），（美）瑞克斯（Rex，J.）著；陈宗斌等译. —北京：机械工业出版社，2009.9

（华章程序员书库）

书名原文：Practical Algorithms for Programmers

ISBN 978-7-111-27296-0

Ⅰ. 程…　Ⅱ. ①宾…　②瑞…　③陈…　Ⅲ. 算法程序－程序设计　Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2009）第 088171 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：陈佳缓

北京京师印务有限公司印刷

2009 年 9 月第 1 版第 1 次印刷

186mm×240mm · 28 印张

标准书号：ISBN 978-7-111-27296-0

定价：65.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

前　言

本书的目的是为在应用程序中使用的算法提供一个实用的纲要。与关于算法的大多数著作不同的是，本书不是一本教材。你将不会发现实现细节(我们把它作为练习留给读者完成)；也不会发现利用较小的代码段对算法进行高度理论化的讨论，以说明如何进行实现。相反，与我们的信念(即最佳的解释是实用的程序)保持一致，你将发现完全用C实现的算法的广泛选择，以及关于如何在各种应用程序中最佳地使用它们的真正实用的讨论。本书中介绍的理论材料只用于支持程序员更改实现，以满足特定的需要或者更明智地为特定的应用选择算法。在这些情况下，本书将以一种平易的方式介绍理论。在每一章末尾提供了更抽象材料的参考文献。

关于代码

虽然C++日益普及，但是由于以下几个原因我们仍然使用C。首先，C仍是一种被广泛了解和使用的通用程序设计语言。其次，对于C代码，C++编译器的编译结果与C编译器的结果几乎完全一致。最后，从C移植到C++并不困难，但是反过来却很困难。因而，使用C代码可以兼顾尽可能多的读者。

在开发本书中的代码时主要考虑了两个目标：可读性和可移植性。本书中的全部代码都在Borland、Microsoft和Watcom for MS-DOS几种编译器下进行了测试，以确保可移植性。我们在扩展DOS下测试了Watcom编译器的编译结果，该扩展DOS使用PharLap或Rational Systems的DOS扩展器(DOS extender)。此外，除了第1章中的一个例外之外，所有的代码在UNIX下都进行了移植和重新编译。确切地讲，将其移植到SVR4的UnixWare实现。对其他UNIX平台和其他操作系统的移植工作也在进行中。

ANSI Extern

在我们的移植工作中，涉及一个很少讨论的移植问题。ANSI C标准提供了几种不同的方式，其中可以将在多个模块中声明为extern的项链接在一起。今天的许多先进编译器不要求将变量的任何局部定义声明为extern：链接器将简单地在连接的程序中定义一个extern变量并将所有的声明指向它。不过，ANSI标准不保证这种行为可以正常工作。

在ANSI下保证行为能正常工作的唯一方法是，将所有的变量声明为extern，使得至少在一个源模块中实际地定义了该变量(也就是说，没有extern指示符)。在实现这种效果时，仍然可以在一个公共的头文件中维持所有的extern变量的声明。这种技术是如下所示在头文件中声明变量，并在所有模块中包括该头文件：

```
Extern int Globalvariable1;
Extern int Globalvariable2;
```

然后在某些模块中，比如在一个包含 main( )函数的模块中，添加下面一行代码：

```
#define Extern
```

这具有取消定义 Extern 并把声明转换为定义的作用。在其他所有的模块中，将具有下面一行代码：

```
#define Extern extern
```

这意味着在所有其他的模块头都声明了 extern 变量。

这种方法具有一定的局限性。主要是，它要求在所有的模块中都适当地定义(#defined)了 Extern。一个更简单的方法是：具有一个局部模块，它利用一个明示常量(manifest constant)标识它自身。在我们的示例中，main( )模块可以通过以下明示常量通告它自身：

```
#define IN_MAIN_MODULE
```

然后，带有 Extern 声明的头文件仅需要包含以下代码：

```
#ifdef IN_MAIN_MODULE
#define Extern
#else
#define Extern extern
#endif
```

如果使用这种方法，其中一个模块将会包含 extern 变量的所有局部定义。这可能不是希望的结果。例如，你可能希望链表的定义出现在与栈不同的模块中。因此在这本书里，不会在所有场合都使用 Extern，而是为需要的栈变量使用 StkExtern，并为链表使用 ListExtern，然后在需要的模块中定义 StkExtern。有关这方面的示例出现在第 2、3 章的代码中。这样，就可以保证 ANSI 将链接 extern 变量，并以我们选择的方式把它们与定义精确匹配。

代码风格

最后，在代码可读性方面，我们已竭尽所能。为此目的，我们采用了如下约定：在定义函数和全局变量时，使用 MixedCased 形式的变量名；在声明局部变量时，使用 k_ 和_ r_ style 风格。此外，为了使代码更可读，我们使用了更多的空白(在水平方向上和垂直方向上)。唯一的例外是第 6 章(二叉树和 B 树)，该章中的代码很多，因此我们的任务是减少纯粹代码的页数。在该章中，我们采用如下约定：减少代码占据的空间，这也稍微降低了代码的可读性。

我们希望你发现本书可读性很好并且是有用的。它旨在提供立即可用的实用信息和解决方案。

Andrew Binstock 和 John Rex

致　谢

这本书最初的想法是由 Phil Pistone 于 1991 年夏天提出的，感谢他提供给我们这个为期三年的任务。Addison-Wesley 的许多优秀成员是这次延误的主要“受害者”，非常感谢他们耐心和友善。尤其是 Keith Woilman，他领导着计算机普通版图书部门并使之成为 Addison-Wesley 的御宝之一。从这个项目一开始，Claire Horne 也参与进来了；她体现了我们所联系的公司的所有和蔼以及恩典。Sarah Weaver 引导我们顺利通过了生产计划，在失败面前，她表现得比其他许多人都要优秀。还要感谢 Steve Vinoski 所做的优秀技术编辑工作和 Barbara Milligan 所做的详尽文字编辑工作。不过，无论他们怎么努力，错误在所难免，但是这完全是我们造成的。我们非常伤心地感谢 Phil Sutherland 和 Julie Stiliman，他们致力于本书的交付工作，但是他们在等待最终原稿的时候找到了新的工作。谁能拒绝他们来分享我们的谢意呢？

我们之所以能够编写这本书，这得益于许多人的重大贡献，他们启发了本书作者以及其他许多程序员：特别是，Dennis Ritchie 透彻解释了 C 语言，Donald Knuth 编写了一些关于算法的基础书籍，以及 Jon Bentley 频繁地、有说服力地提醒我们如何运用简单的原理来解决复杂的问题。

在个人层面上，我们希望感谢以下人：Bill Hunt，他编写了一本关于 PC 平台上的 C 语言程序设计的确实很好的图书(特别是其中清晰的代码)；Allen Holub，他教会我们(以及整整一代程序员)如何使用 C 语言来解决非常困难的编程任务；Mark DeSmet 和 Farley，他们开发了一款引领我们进入 C 程序设计的极佳编译器；另外还要特别感谢《*C Gazette*》一书的许多读者和对它有过贡献的人，他们提升了本书的价值。

Andrew Binstock 和 John Rex

1995 年 4 月

译 者 序

数据结构与算法是计算机专业的核心课程，是计算机软件开发和应用人员必备的专业基础。今天的大多数关于算法的图书都是大学教科书，或者是令人厌倦的相同算法集合改头换面后的作品。本书是给出所有算法的完整代码实现的第一本书，这些算法对于开发人员在其日常工作中是有用的。

本书介绍了关于算法的基础知识、基本数据结构、散列、查找、排序、树、日期和时间、任意精度的算术运算、数据压缩以及数据完整性和验证等内容。本书的目的是为在应用程序中使用的算法提供一个实用的纲要。与关于算法的大多数著作不同的是，本书不是一本教材：书中没有提供实现细节，把它作为练习留给读者完成；也没有利用较小的代码段对算法进行高度理论化的讨论，以说明如何进行实现。相反，本书完全用 C 语言实现了各种算法，并且讨论了如何在各种应用程序中最佳地使用它们。

本书只要求读者具有 C 语言的初级知识以及不超出基本代数之外的数学知识。源代码是符合 ANSI 标准的，并且对它们进行了测试，它们都可以运行在 UNIX 下以及 Borland、Microsoft 和 Watcom 的编译器上。

本书非常适合于高等院校计算机专业的学生阅读，对于从事计算机软件开发的人员，也将从本书中受益匪浅。

参加本书翻译的人员有：陈宗斌、张景友、易小丽、陈婷、管学岗、王新彦、金惠敏、张海峰、徐晔、戴锋。

由于时间紧迫，加之译者水平有限，错误在所难免，恳请广大读者批评指正。

译者

2009 年 6 月

目　录

译者序
前　言
致　谢

第1章　绪论 …… *1*
1.1　评估算法 …… *1*
1.2　修改算法 …… *4*
1.2.1　主要的优化：I/O …… *4*
1.2.2　主要的优化：函数调用 …… *7*
1.3　资源和参考资料 …… *8*

第2章　基本数据结构 …… *9*
2.1　链表 …… *9*
2.1.1　双向链表 …… *17*
2.1.2　链表的其他特征 …… *31*
2.2　栈和队列 …… *31*
2.2.1　栈的特征 …… *32*
2.2.2　队列的特征 …… *38*

第3章　散列 …… *48*
3.1　散列的概念 …… *48*
3.2　散列函数 …… *51*
3.3　冲突解决方法 …… *54*
3.3.1　线性再散列法 …… *54*
3.3.2　非线性再散列法 …… *55*
3.3.3　外部拉链法 …… *57*
3.4　性能问题 …… *69*
3.5　资源和参考资料 …… *70*

第4章　查找 …… *71*
4.1　查找的特征 …… *71*
4.1.1　准备时间 …… *72*
4.1.2　运行时间 …… *72*
4.1.3　回溯的需要 …… *72*
4.2　蛮力查找 …… *72*
4.3　Boyer-Moore 查找 …… *76*
4.3.1　启发式方法#1：跳过字符 …… *76*
4.3.2　启发式方法#2：重复模式 …… *78*
4.4　多字符串查找 …… *83*
4.5　用于正则表达式的字符串查找：grep …… *96*
4.6　近似字符串匹配技术 …… *112*
4.7　语音比较：Soundex 算法 …… *118*
4.8　Metaphone：现代的 Soundex …… *121*
4.9　选择技术 …… *128*
4.10　资源和参考资料 …… *129*
4.10.1　通用参考资料 …… *129*
4.10.2　Boyer-Moore …… *129*
4.10.3　多字符串查找 …… *130*
4.10.4　正则表达式查找 …… *130*
4.10.5　近似字符串匹配 …… *130*
4.10.6　Soundex 算法和 Metaphone 算法 …… *130*

第5章　排序 …… *131*
5.1　排序的基本特征 …… *131*
5.1.1　稳定性 …… *131*
5.1.2　对哨兵的需求 …… *131*
5.1.3　对链表进行排序的能力 …… *132*
5.1.4　输入的阶的相关性 …… *132*
5.1.5　对额外存储空间的需求 …… *132*
5.1.6　内部排序技术与外部排序技术 …… *132*
5.2　排序模型 …… *132*
5.2.1　冒泡排序 …… *137*
5.2.2　插入排序 …… *141*
5.2.3　希尔排序 …… *143*
5.2.4　快速排序 …… *146*
5.2.5　堆排序 …… *162*

5.3 对链表进行插入排序 …… 166
5.4 对链表进行快速排序 …… 171
5.5 对多个键进行排序——不稳定排序的修正方法 …… 177
5.6 网络排序 …… 178
5.7 小结：选择一种排序算法 …… 182
5.8 资源和参考资料 …… 185
第6章 树 …… 186
6.1 二叉树 …… 186
6.1.1 树查找 …… 209
6.1.2 节点插入 …… 209
6.1.3 节点删除 …… 209
6.1.4 二叉查找树的性能 …… 211
6.1.5 AVL 树 …… 212
6.2 红黑树 …… 214
6.3 伸展树 …… 218
6.4 B 树 …… 221
6.4.1 保持 B 树平衡 …… 222
6.4.2 实现 B 树算法 …… 223
6.4.3 B 树实现的代码 …… 224
6.5 可以看见森林吗 …… 276
6.6 资源和参考资料 …… 276
第7章 日期和时间 …… 278
7.1 日期例程的库 …… 279
7.2 时间例程 …… 292
7.3 用于日期和时间数据的格式 …… 293
7.4 最后的提醒 …… 300
7.5 资源和参考资料 …… 300
第8章 任意精度的算术 …… 301
8.1 构建计算器 …… 301
8.2 表示数字 …… 309
8.3 计算 …… 319
8.4 加法 …… 322
8.5 减法 …… 324
8.6 乘法 …… 329
8.7 除法 …… 335
8.8 关于计算器要注意的最后几点 …… 349
8.9 用于计算平方根的牛顿算法 …… 349
8.10 分期付款表 …… 354
8.11 资源和参考资料 …… 357
第9章 数据压缩 …… 359
9.1 行程编码 …… 360
9.2 霍夫曼压缩 …… 368
9.2.1 代码 …… 369
9.2.2 其他问题 …… 383
9.3 滑动窗口压缩 …… 384
9.4 基于字典的压缩(LZW) …… 390
9.4.1 LZW 算法的伪代码 …… 391
9.4.2 LZW 压缩的实现 …… 392
9.4.3 填满字典 …… 408
9.5 使用哪种压缩方法 …… 409
9.6 资源和参考资料 …… 409
第10章 数据完整性和验证 …… 411
10.1 简单的校验和 …… 411
10.2 加权校验和 …… 415
10.3 循环冗余校验 …… 423
10.3.1 CRC-CCITT …… 424
10.3.2 CRC-16 …… 430
10.3.3 CRC-32 …… 432
10.4 资源和参考资料 …… 437

第1章 绪 论

1.1 评估算法

除了最直观的应用之外，算法是所有程序的核心和灵魂。算法一般被设计用于以最小的代价高效地解决特定的问题。算法的价值一般取决于两方面因素：如何恰当地解决问题以及如何高效地实现解决方案。这些是算法分析的定性和定量方面。

对于许多算法，质量不是一个问题。例如，对于排序算法，必须保证每次都对所有元素正确地进行了排序。一旦出错，就必须丢弃它并且严格说来不能将其视为一种算法。在其他领域，不能基于这种简单的通过/失败测试来度量质量。例如，在第4章中介绍的Soundex算法允许检索听起来相同的单词或名字。与排序算法不同，可以调整Soundex算法，以寻找接近的匹配或者相当宽泛的匹配；这取决于实现算法的方式和开发人员的需求。在这种情况下，质量是可度量的并且是算法的重要方面，并且指导我们认真选择不同的解决方案。

算法设计的定量方面尝试确定算法所需的资源。一般来说，最重要的度量标准是时间：即算法运行得有多快？偶尔，计算机资源（比如可用的内存）也是一个重要因素。

度量性能

与基准的性能不同，算法的性能很少依据时间来加以说明。在论及排序例程时，你几乎从未听到它完成排序要花费8.62秒这样的说明。这有一个很好的理由：这种计时难以复制，并且通常依赖于正在处理的数据的具体特征。算法不依赖于计时，而是依赖于一个直观的方程，以显示输入的大小与性能之间的关系。用于显示这种关系的传统方法是使用符号 O，称为**大 O 表示法**（big-oh notation）。其工作方式是：假定你有一个算法，它简单地通读一个文本文件，从中查找单词flea。一种合理的方法是寻找字母f的每个实例（参见第4章）。当找到一个f，该算法就测试4个字母的序列，看看它是不是单词flea。在这个示例中，显然执行时间直接与文本文件的大小成正比。如果给定的文件包含 N 个字符，那么我们就说该算法的执行时间的界限是 $O(N)$。你会注意到这种表述没有考虑到可能影响性能的其他因素——例如，字母f在文本中出现的频繁程度。在查找字符串时（比如fleas rarely wear collars），字符串的长度以及其中相似字符串（比如fleas rarely wear colors）出现的频率也会影响性能。不过，严格来讲，这些因素是要处理的数据的函数，而不是算法的函数。因此，在大 O 表示法中，它们不会出现在公式中。该表示法只是简单地说明数据规模（一般用 N 表示，偶尔也用 n 表示）与算法的典型性能之间的关系。

另一个示例也许更能说明问题：在著名的冒泡排序算法中（参见第5章），对于一个给定的项目列表，通过遍历一次列表对其中的每个项目进行排序。每次遍历，都会减少一个要检查的元素。在遍历列表中的项目时将比较相邻的元素，如果它们的排序顺序不正确，就交换它们的位置。

在这种排序中，对性能影响最大的是排序的次数。对于3个项目的列表，3次遍历将产生3次比较。10个项目的列表则需要45次比较。因此，很明显，比较次数总是$N(N-1)/2$。在这个表示法中，数据规模与执行之间的关系（将这种关系称为**性能（performance）**）将记作$O((N^2-N)/2)$。除非N的值较小，否则这样的性能将慢得令人不可接受。

算法科学考虑的是通过分析和改进算法，从而稳定地增强性能方程。在设计和实现算法时，对这个算法方程有一定的感性认识是很重要的。其原因非常有说服力。如果一种算法的性能比N^2更差，我们通常认为它是无用的。如果在运算速度为1纳秒的计算机上运行算法，那么性能为$3N^3$的算法需要花费95年的时间来处理1 000 000个项目的输入，而阶为19 500 000N的算法仅需要花费5.4小时来处理同样数量的元素。很明显，造成这些巨大差异的原因是：N的阶是至关重要的，而它前面的常数则不是。在前面的示例中，即使该算法使用的是N^3而不是$3N^3$，它仍将是无用的。因此，大O表示法不使用常数；它仅使用N的阶。这样，用于冒泡排序的大O表示法是$O(N^2-N)$。

在本书中，我们对性能方程的介绍稍有不同。如以前所解释的那样使用性能方程；不过，我们将保留这些常数，因为我们觉得它们的存在也很重要。为了不给计算机科学的一些漫不经心的学生造成混淆，我们不使用大O表示法，因为对于受过训练的学生，常数的存在看起来很奇怪。

1. 平均情况与最坏情况

只根据单个方程或度量标准来讨论性能是不够的。某些算法的表现在大多数情况下相当不错，但在最坏情况下却令人无法接受。由于很少能够对通用算法将应用的数据加以限制，因此在设计算法时考虑到最坏情况是很重要的。最坏情况下的性能有时可能很差，以致于采用一种能够自如地处理最坏情况的通常慢一些的算法可能是一个更好的选择。让我们来看看著名的快速排序（quicksort）算法（参见第5章），它在C标准库中被实现为函数qsort()。它在大多数情况下都表现得非常好。其平均情况下的性能为$N\lg N$（缩写lg是指以2为底的对数。对数是指将底数（这里是2）提升为n的指数值。例如，lg8是3，因为2^3的值是8）。这意味着快速排序需要8×3个时间单位对8个项目进行排序，或者说需要32×5个时间单位对32个项目进行排序。结果是，当输入从8个项目到32个项目增加了4倍时，排序时间将从24个单位时间增加到160个时间单位。同样，1 024个项目则需要10 240个时间单位。这样的性能被认为是高效的。

不过，在将快速排序算法应用于已经排好序的输入文件时，该算法将表现出最差的性能。性能方程突然变成N^2。如表1-1所示，如果这种唯一的最坏情况完全有可能发生，那么就不能使用快速排序算法（幸运的是，对于大多数C程序员而言，编译器供应商在他们的qsort()实现中提防了这种情况，所以这个问题通常不会出现）。

表1-1 快速排序算法的平均情况和最坏情况下的性能，以针对N个项目的时间单位作为度量标准

N	平均情况：$N\lg N$	最坏情况：N^2
8	24	64
32	160	1 024
1 024	10 240	1 048 576

2. 最坏情况下的性能

当分析算法在最坏情况下的行为时，试图去简单地估计最坏情况发生的可能性是不够的。还应当考虑万一发生最坏情况时的后果。在快速排序算法的示例中，很明显，算法最主要的代价是性能惩罚；也就是说，当发生最坏情况时，排序将缓慢地继续进行。不过，对于某些算法，最坏情况下的性能惩罚可能很大，出于实用性考虑，该算法会挂起系统（例如，假定在最坏情况下数据排序需要 15 天的时间）。

某些类别的问题具有如此严重的最坏情况下的后果，以致于在编写算法时必须确保已经检测到最坏情况并进行了妥善处理。这类问题经常出现在调度领域。在本书中没有对调度算法加以介绍；不过，由于它们能够很好地用于说明该问题，我们将简要介绍其中的一个经典类别：“哲学家进餐”问题。

该问题由 E. W. Dijkstra 于 1965 年首次提出，现在已经有了很多变体。它实质上可以表述如下：5 个哲学家坐在圆餐桌旁享用一顿面条（参见图 1-1）。任意两个盘子之间只有一根筷子。吃面条的时候，每个哲学家必须使用两根筷子：从她们两边的盘子中各取一根筷子。因此，如果任何一个相邻的哲学家在吃面条，这个哲学家自己就不能吃面条，而必须等待。她可以在任何时刻尝试吃面条，方法是首先拿起一根筷子，然后再看看另一根筷子是否也可以使用。如果另一根筷子不可用，那么她必须放下刚才拿起的那一根筷子。在大多数场合（按照我们的说法，即平均情况下），所有的哲学家都能吃到面条，尽管他们中的一些需要等待轮到自己才能用餐。不过，在最坏情况下，所有的哲学家都会被饿死（如果你不熟悉这个问题，可以暂时停下来，推算一下最坏的情况应该是什么）。假定所有的哲学家都在同一时刻决定开始吃面条。他们全都拿起右边的那一根筷子，然后再转过来看左边的那根筷子，当然，这根筷子已经被别人拿走了。于是他们放下筷子，然后立刻再次开始进行下一轮尝试。刚才的情形又重复发生了。只要他们同时决定开始进餐（即最坏情况），他们将会一致地拿起再放下右边的那根筷子，直到他们被饿死为止。按照算法的术语，称程序将开始一个无限循环。这种最坏的情况杀死了哲学家以及算法本身。

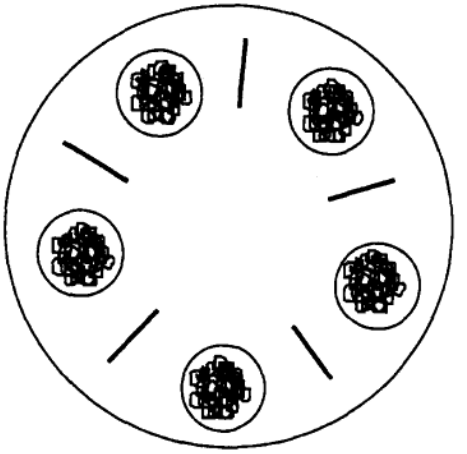


图 1-1　摆放有面条的桌子，可能会使哲学家因无法进餐而饿死

该问题的要点是：在考虑算法时，应该从其可能性和后果两方面研究最坏情况。如果发现后果是灾难性的而你又无法改变算法本身，那么就必须为最坏情况事先做好准备。

3. 最好情况下的性能

人们通常很少分析算法在最好情况下的性能。这是因为一般在最好情况下不会导致应用程序完全停止，因此，它没有什么危险。此外，与最坏情况一样，它极少发生，并且极度依赖于要处理的数据本身。但是，这并不是说对最好情况就不需要进行分析。许多算法在运行最好情况的数

据时，性能有惊人的提升。经典的示例是前面讨论过的冒泡排序算法。它的典型性能是 $O(N^2 - N)$，而在最好情况下的性能是 $O(N)$。这意味着对于大多数应用程序，冒泡排序是非常慢的，而在最好情况下这个排序例程运行得足够快。这对于应用程序开发人员具有一些隐含意义，因为在某些场合下可以预先知道要处理的数据的质量。一种典型的情况是将一个算法的输出作为另一个算法的输入。在后面将看到，在这种情况下，可以有效地使用最好情况下的性能（甚至对于冒泡排序也是如此）。

1.2 修改算法

在计算机领域，一项正在进行的工作是，通过对算法进行改进以求获得最佳的性能。这种工作通常采用以下两种策略之一：优化现有的算法或者开发新的算法。这些策略具有截然不同的目标，应当加以区别看待。在优化算法的时候，一般不会尝试使其性能方程降级。例如，我们知道冒泡排序的平均性能是 $O(N^2 - N)$。如果你必须使用冒泡排序，那么将希望确定在冒泡排序中执行的动作所耗费的时间非常短。也就是说，希望它的两个主要操作（比较列表中的元素并交换它们）执行得非常快。在应用程序的上下文中，需要付出相当大的努力来确保算法的实现是经过完全优化的。例如，需要确保在内存中而不是在磁盘上交换元素。通过处理每个数据项的处理所需的时间，可以节省大量的运行时间。通过对应用程序定制这种时间可有效地优化算法。不过，你还是无法避开以下事实：冒泡排序算法随着数据输入规模的增大，其性能开销也将呈几何级数增长。为了解决这个问题，就需要使用或设计一种新算法。

此时，开发人员的工作将经历从高度实用（优化）到抽象的过程。现在必须找到一种新方法，其性能比 $O(N^2 - N)$ 更好。如果成功地使性能方程降级，就开发出了一种新算法。这种区别是重要的，因为它将以你的方法作为条件。在几乎所有的情况下，需要分析广泛的解决方案，选择其中之一，然后根据具体情况对其进行优化。在这个过程中，需要理解这个算法是如何工作的，以便确定优化工作最好的努力方向。

首先，应该应用标准技术：使输入/输出（I/O）减到最小，减少函数调用的次数，限制计算密集型操作，比如浮点运算和频繁使用除法运算。然后，必须确定执行得最频繁的算法元素。在冒泡排序中，比较和交换应该是需要强烈关注的主题。最后，要检查可能由于疏忽而导致特别缓慢的实现。最后这一点往往与查找最坏情况相似。你正在查找可能导致性能下降的任何不寻常的情况。通常，这些情况将出现在实现细节的深处，并且是一个通常合理但偶尔代价高昂的基本假定的结果。一个已知的示例是前面讨论过的快速排序算法。

1.2.1 主要的优化：I/O

减少 I/O 和函数调用的开销是如此重要，以至于通过讨论便携式技术以降低其开销被证明具有充分的根据。I/O 通常发生在毫秒级时间范围内，而 CPU 活动一般发生在亚微秒级的范围内。因此，对于算法的性能而言，任何 I/O 的代价都非常高昂。如果不能消除 I/O 本身，那么可以通过使用智能缓冲来减小它的影响。许多程序员把它看作创建和控制他们自己的缓冲区的动机，从而花费相当多的努力和代码来管理 I/O 缓冲区。ANSI C 利用 setvbuf( ) 函数提供了一种优雅的替代方案。这个可移植的函数可以让程序员为输入流设置缓冲区的大小。问题是缓冲区应该设置成

多大才合适。对此，大多数编译器库倾向于将缓冲区大小设置为一个保守但是可以工作的大小。扩展缓冲区可以显著减少 I/O 次数。程序清单 1-1 中所示的程序 bufsize. c 允许测试你自己的系统上最佳的缓冲区大小。它接受 4 个参数：要复制的测试文件、目标文件的名称、两个缓冲区大小（一个用于输入缓冲区、另一个用于输出缓冲区）。然后，程序把输入文件复制 5 次，并在命令行指定的缓冲区大小给出复制过程的最大、最小和平均持续时间。如同所编写的那样，该程序调用 MS-DOS 的时钟并以时钟滴答次数报告计时。对于 UNIX 系统下的等价函数应该会有一些变化。作为替代，可以使用开始次数和结束次数，并报告时间（以秒为单位）。不过，以秒为单位将给出一个粗略的度量。

程序清单 1-1　在 MS-DOS 下测试最佳缓冲区大小的程序

```
/*--- BUFSIZE.C ------------------------- Listing 1-1 --------
 * Purpose:  Demonstrate use of setvbuf
 * Usage:    bufsize infile outfile [insize [outsize]]
 * Method:   Copies infile to outfile one character at a time,
 *           reporting elapsed time. Infile is set to use a
 *           buffer of insize bytes, while outfile uses an
 *           outsize bytes buffer. Default size is 512 bytes.
 * >> Should be changed per text for UNIX systems <<
 *-------------------------------------------------------------*/
#include <bios.h>   /* for timer function */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define DEF_BUF 512

#ifdef __TURBOC__                /* Borland   */
#define get_clock_ticks(x) x=biostime(0,0L)
#else                            /* Microsoft and Watcom */
#define get_clock_ticks(x) \
        _bios_timeofday(_TIME_GETCLOCK, &x)
#endif

long CopyFile ( char *infilename, char *outfilename,
            size_t insize, size_t outsize )
{
    int  c;
    long starttime, donetime;
    FILE *infile, *outfile;

    /* open input file and setup its buffer */
    if (( infile = fopen ( infilename, "rb" )) == NULL)
    {
          printf ( "Can't open %s\n", infilename );
          exit ( 1 );
    }

    if ( setvbuf ( infile, NULL, _IOFBF, insize ))
    {
         printf ( "Couldn't set infile buffer to %u bytes.\n",
                  insize );
         exit ( 1 );
```

```
    }

    /* open output file and setup its buffer */

    if (( outfile = fopen ( outfilename, "wb" )) == NULL )
    {
          printf ( "Can't open %s\n", outfilename );
          exit ( 1 );
    }

    if ( setvbuf ( outfile, NULL, _IOFBF, outsize ))
    {
         printf ( "Couldn't set outfile buffer to %u bytes.\n",
                  outsize );
        exit ( 1 );
    }

    /* do it */

    get_clock_ticks ( starttime );     /* get timer value */
    while (( c = fgetc ( infile )) != EOF )
        fputc ( c, outfile );
    get_clock_ticks ( donetime );
    fclose ( infile );
    fclose ( outfile );

    return ( donetime - starttime );
}

int main ( int argc, char *argv[] )
{
    size_t insize, outsize;
    int    i;
    long   total, average, lo, hi, elapsed;

    insize = outsize = DEF_BUF;

    if ( argc < 3 || argc > 5 )
    {
        fprintf ( stderr,
         "Usage: BUFSIZE infile outfile [insize [outsize]]\n" );
        return ( EXIT_FAILURE );
    }

    /* get buffer sizes */
    if ( argc > 3 )
         insize = (unsigned) atoi ( argv[3] );

    if ( argc > 4 )
         outsize = (unsigned) atoi ( argv[4] );

    /* now, copy the file five times */
    total = hi = 0;
    lo = LONG_MAX;
    for ( i = 1; ; i++ )
    {
```

```
        elapsed = CopyFile ( argv[1], argv[2], insize, outsize );
        if ( elapsed > hi )
            hi = elapsed;
        if ( elapsed < lo )
            lo = elapsed;
        total += elapsed;
        if ( total > 500 ||   /* Change this value depending   */
                              /*   on how long you can wait    */
             i > 4 )          /* Do 4 passes, if time limit OK */
            break;
    }

    average = total / i;

    printf ( "Average of %4ld ticks (%4ld - %4ld). "
             "Insize = %5u. Outsize = %5u.\n",
             average, lo, hi, insize, outsize );

    return ( EXIT_SUCCESS );
}
```

图 1-2 显示了在多种缓冲区大小上运行这个程序的结果。垂直轴显示了复制 64 KB 文件的时间单位；水平轴以字节为单位显示了输入和输出缓冲区的大小。这张图是在真实模式下运行的 MS-DOS 系统上生成的。它代表了在其他系统下的性能改善。当缓冲区大小为 4 096 字节时将达到最佳性能，而更大的缓冲区并没有带来明显的性能改善。事实上，对于大多数应用程序，其中最重要的需求并不是要求它们以最快的速度运行，此时 1 024 字节的缓冲区大小是足够的。

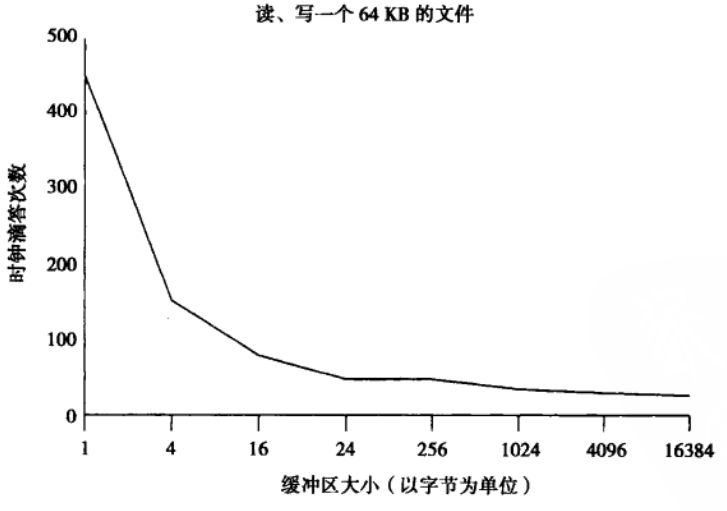


图 1-2　将 I/O 的性能作为缓冲区大小的函数，并以时钟滴答次数显示时间

1.2.2　主要的优化：函数调用

另一个重要的优化是减少函数调用的次数。虽然现在许多编译器已经针对函数调用进行了重

大优化，但是对于函数携带的重大开销还是稍存疑虑。为了验证这一点，我们可以编写一个小程序，它把一个只执行返回语句的函数调用了10 000次，你将开始感觉到函数调用的代价，如果函数传递参数或者返回值，那么花费的时间将会更长一些。常见的惯例是使用内联代码或者宏，把函数直接隐藏在算法中。研究编译器的库（如果可用，应该总是能得到源代码），并且你将会发现许多函数已经被实现为宏。要尽可能使用这些宏，并且要留心观察它们的副作用。如果你仍然必须调用函数并且不能获得所需的性能，可以考虑将供应商的源代码直接复制到你的算法中。这是一种危险的做法，仅当执行了所有其他的优化之后才应该考虑把它作为最后一种办法。除了版权问题之外，严格来讲，库的源代码也很少是可移植的，并且可能会处理隐藏在库内的变量。不过，可以轻松地引入许多函数或者把它们转换为宏。每种方法都会增加代码的大小，因此必须小心，不能过度使用这种技术。

另一种常见的算法技术是消除递归。递归是使函数调用它自身的行为。在C语言中，所有的函数都可以进行递归调用：即它们都能调用自身，至少在理论上如此。实际上，像main()和exit()这样的函数在进行递归调用时可能会引发问题。不过，几乎所有不会终止程序的函数都可以调用它们自身。在算法中经常会使用递归技术，因为它可以产生短小、优雅的解决方案。不过，它的代价高昂，包括时间和系统资源两方面，尤其体现在栈空间的消耗上。第5章给出了关于如何消除递归的一些示例。在可能的情况下，应尽量避免使用递归。

在MS-DOS环境下，还有另外几种技术可用于减少函数调用的开销。首先，启用某些高级编译器提供的优化功能，它们允许使用寄存器而不是栈来传递函数参数（例如：Borland使用-pr开关，Microsoft使用/Gr开关以及-fastcall关键字，而Watcom则会默认执行优化）。如果用到了第三方程序库，在使用这些功能时就应该小心谨慎；有关更多信息，请参考相关的编译器文档。

MS-DOS用户的第二个选择是使用pascal关键字。它将影响被调用的函数如何操作栈。如果使用这个关键字，可以同时节省时间和空间。与上述的编译开关一样，也必须小心使用pascal关键字。它不能用于参数数量可变的函数，并且一般不能把它与上述的编译开关结合起来使用。同样，请认真阅读编译器文档。如果必须从这两种方法中选择一种，就必须检查你的程序。如果程序通常只传递数量很少的参数（但不是零个参数），就使用第一个选项；如果程序具有多种类型的函数，有的函数不带参数，有的函数则带有两个或三个以上的参数，那么pascal选项将更高效。

最后，认真学习算法，然后钻研其中最频繁使用的部分。在这个过程中，拥有一种优秀的性能测量和跟踪工具（profiler）将给你带来巨大的好处。购买一种这样的工具并使用它；通过定期使用它可以加快代码的执行速度。在后面的各章中，在演示算法时我们强调了代码的清晰性。如果由于进行优化而使代码含义晦涩，我们将放弃这样做。只要有可能，我们就会给出关于在什么地方可以优化算法的提示和信息。

1.3 资源和参考资料

Rex, John. “The pascal Keyword.” *C Gazette*, Vol. 3, No. 4. 它很好地分析了使用pascal选项的好处和风险。

Tanenbaum, Andrew. *Operating Systems: Design and Implementation.* Englewood Cliffs, NJ: Prentice Hall, 1987. 尽管哲学家进餐问题已被多次介绍，但是这本经典的著作最值得一读。

第2章　基本数据结构

算法是指为了达到某种目的而操作数据的方式。算法通常适用于具体的问题，以复杂的方式来处理简单的数据。例如，编译器把简单的字符串和字符翻译成可以在特定机器上执行的二进制代码。编译器的内部工作原理是：通过相互影响的诸多算法按顺序将源代码字符转换为中间表示。这些表示都是通过其他算法将它们自身翻译成目标代码，这些代码与原始的字符和字符串（指源代码）具有极少的相似之处。尽管程序不像编译器那样有雄心壮志，但它们也同样极大地依赖于算法，以一种可预见的方式来处理数据。因此，在不了解操作数据的基本例程的情况下，不可能深入地讨论算法。那些希望通晓如何使用算法的开发人员首先要学习如何操作数据。而后，他们可以根据自己的需要应用算法技术来表示数据。

通过算法操作数据主要涉及的是在内存中表示数据的技术。怎样存储、访问数据以及怎样转换数据以便最高效地解决给定的问题？大多数问题都要求开发人员能够熟练地使用某些基本的**数据结构**（data structure）。这些数据结构是存储或处理数据的形式。最简单的数据结构包括：**数组**（array）、**链表**（linked list）、**栈**（stack）和**队列**（queue）。

出于本书的目的，假定你已经熟悉 C 语言中的数组。无论它们是字符数组（如字符串）、整数数组，还是结构体数组，操作的原理是相同的。如果您不熟悉这些操作，可以在学习下面的内容之前先参考一下关于 C 语言的初级教材。本章将详细探讨链表、栈和队列。第 3 章将再次探讨链表，其中相当深入地讨论了散列表；第 6 章将探讨错综复杂的二叉树。

2.1　链表

C 语言（以及其他许多语言）中的数组必须在编写程序时定义。也就是说，在 C 语言中定义一个数组时，还必须定义它的大小。甚至对于在全局级别（即在任何函数之外）声明或定义的数组，例如：

```
#include <stdio.h>

char BigArray[];

int main ( int argc, char *argv[] )
{
      ...
```

也必须在某个位置指定数组的具体大小，一般使用 malloc( ) 函数或者某种类似的机制来执行该任务。

无论是否使用 malloc( ) 函数，在使用数组之前都必须先指定其大小。这就带来了一个问题：当事先无法知道需要创建多少个元素时，怎样分配必要的空间。

例如，假设你需要编写一个程序，它从一个输入文件中读取城市的名称及其气温信息。最后

需要按气温对城市进行排序，并确定中间的气温（也就是说，在该气温的两边，更冷和更热的气温数量相同）。对于这个问题，使用一个简单的数组不是一个好的选择。因为不知道应该创建多大的数组才合适。或许可以声明一个比你预期所需空间大得多的数组，然后希望输入文件永远不会超过这个大小。不过，这种方法不仅浪费空间，而且如果输入文件突然超过预期的大小，则很有可能导致程序失败。

一种解决方案是读取文件两遍，第一遍用来确定数组的大小并且为它分配空间，第二遍则进行实际的数据处理。不过，这种解决方案的效率很低。因为磁盘 I/O 的速度非常慢（甚至在 RAM 磁盘上也是如此）——事实上，它几乎总是任何程序中最慢的部分，一般要慢一个数量级，因此只要有可能，就应避免读取两遍数据。

一种更巧妙的解决方案是使用链表，它按接收到数据的方式来存储它们。链表和前一种方法一样精确，但效率要高得多。链表由一连串元素（称为节点）构成。每个节点包含要存储的数据项以及一个指针，它指向链表中的下一个节点。当程序读取每个数据项时，将创建一个新的节点（使用 malloc() 函数），并将其添加在链表的尾部。在输入结束时，计算机内存中将维持一个节点列表，其中每个节点都包含数据项（例如城市名称和气温信息）以及指向下一个节点的指针。最后一个节点中的指针的值为 NULL，在 ANSI C 中将该值定义为不指向任何地方的指针。当通过遍历链表来查找中间气温时，如果遇到 NULL 指针，则说明已经到达了链表尾部。每个节点中的指针称为**链**（link），因此将这种数据结构称为**链表**（linked list）。每个链表都以一个简单的指针开始，它指向链表中的第一个数据项，这个指针称为**头指针**（head）。一个简短的链表如图 2-1 所示。

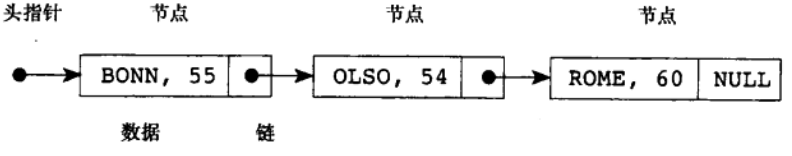


图 2-1 具有三个节点的链表

在 C 语言中，可以将图 2-1 中所示的链表表示如下：

```
struct Node {
      char *City;
      int   Temp;
      struct Node *Next;
};

typedef struct Node * Link;

Link Head;
```

创建 typedef 是为了增强代码的可读性。可以预见，“链”是指向“节点”的指针。我们定义的第一个“链”是“头指针”，当链表为空时它是 NULL，否则它指向第一个节点。

初始化一个这种类型的新链表只需要简单地初始化合适的变量：

```
Head = NULL;
NodeCount = 0;
```

整型 NodeCount 将在以后处理链表时用到。一旦添加了第一个节点，头指针将指向它。添加节点可能很容易或者是乏味的。如果在添加节点时无需考虑它们在链表（无序链表）中所占据的位置，可以简单地将节点添加在链表头部。当为每个节点都分配了空间时，它的“链”将指向当前“头指针”，然后更新“头指针”以指向新的“节点”，如图 2-2 所示。

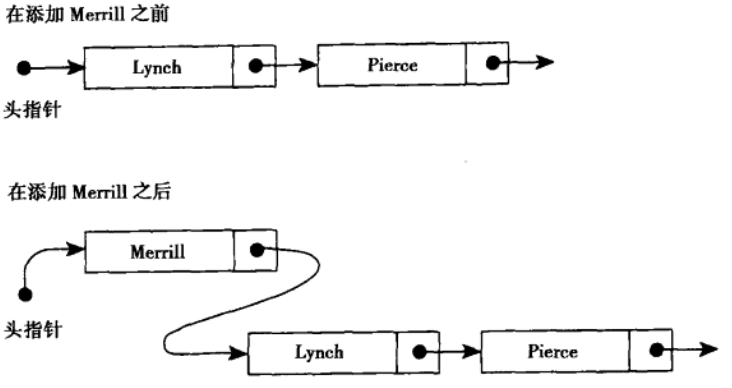


图 2-2 向链表中无序地添加节点

向其元素具有某种顺序的链表（有序链表）中添加节点需要做更多的工作。首先，需要提供一种方式来比较两个节点的数据，并且确定哪个节点在链表中处于更高或更低的位置。此外，还必须确定当新节点与现有节点重复时应该如何处理：是应该添加新节点、丢弃新节点，还是应该修改现有节点的数据？在程序清单 2-1 中，分别通过以下两个函数来执行这些任务：NodeCmp( ) 和 DuplicateNode( )。在定义了这些函数之后，就可以执行添加节点的实际工作。

在遍历链表时，仅当经过插入点或者已经到达链表尾部时，才能知道已经找到了添加节点的位置。因此，将需要保存前一个被检查的节点地址（程序清单 2-1 中的函数 AddNodeAscend( ) 中的 prev）。插入发生在当前被检查的节点（curr）和前一个节点之间。如果 pn 指向你正试图添加的节点，那么这个过程如下：

```
prev->Next = pn; /* prev must now point to pn */
pn->Next = curr; /* and pn must now point to curr */
```

需要考虑几种特殊情况：向一个空链表中添加节点、在第一个节点之前添加节点以及在链表的尾部添加节点。在函数 AddNodeAscend( ) 中显示了一种用于简洁地处理所有这些情况的方法。我们将不会测试所有的特殊情况，而是创建一个虚拟节点，并使当前链表从该节点延续下来。这样，我们就知道我们的链表永远不会为空，并且我们的实现逻辑也将大大简化。程序清单 2-1（citytemp. c）从数据文件中读取城市和气温信息，将记录插入到一个链表中（按气温和城市名称的升序进行排序），丢弃重复的记录，然后打印该有序链表，并指示位于中间的条目。数据记录是文本文件中简单的行，它以前三个字符表示气温，其后接着最多 124 个字符表示城市名称。

注意：如果不使用像链表这样的动态结构，在不读取数据两遍的情况下，将不可能确定出中间气温。最后一个循环用于打印链表中的记录，它显示了遍历链表有多简单。

程序清单2-1 打印城市和气温的有序链表

```
/*--- citytemp.c--------------------------- Listing 2-1 ---------
 *  Reads a text file of cities and temperatures in the
 *  following format:   TempCity
 *      where Temp is a number of three digits or
 *      a sign and two digits; City is a string of length < 124
 *  Examples:      -10Duluth
 *                 096Phoenix
 *  The records are read into a singly linked list by order
 *  of temperature and city; duplicates are discarded. At EOF,
 *  the whole list is printed with an indication of the median
 *  temperature. And then, the list is progressively shortened
 *  and reprinted showing the median.
 *              Usage: citytemp filename.ext
 *-----------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*--- data definitions ---*/

struct Node {            /* a node in our linked list */
    char  *City;
    int    Temp;
    struct Node *Next;
};

typedef struct Node * Link; /* Links are pointers to nodes */
Link Head;              /* head of our linked list */
int  NodeCount;         /* how many nodes in the list */

/*--- functions declarations for linked lists ---*/

int  AddNodeAscend ( Link );        /* add a node           */
void CreateList ( void );           /* initialize list      */
int  DeleteNode ( Link );           /* delete a node        */
int  DuplicateNode ( Link, Link );  /* handle duplicate inserts */
void FreeNode ( Link );             /* free a node's memory */
void ShowNodes ( void );            /* show list of nodes   */
int  NodeCmp ( Link, Link );        /* compare two nodes    */

/*--- function definitions ---*/

int AddNodeAscend ( Link to_add )
{
    Link   pn,   /* local copy of node to be added */
           prev, /* points to previous node */
           curr; /* points to node being examined */
    struct Node dummy;
    int    i;

    /* Make a copy of the input node */
    pn = ( Link ) malloc ( sizeof ( struct Node ));
    if ( pn == NULL )
        return 0;
    memcpy ( pn, to_add, sizeof ( struct Node ));
```

```
    /* set up a dummy node to simplify logic */
    dummy.Next = Head;
    prev = &dummy;
    curr = Head;

    /* insert node pn */
    for ( ;; prev = curr, curr = curr->Next )
    {
        if ( curr == NULL )
            break; /* reached the end */

        i = NodeCmp ( pn, curr );
        if ( i <= 0 )
             break; /* pn precedes curr */
    }

    if ( curr && i == 0 ) /* we have a duplicate */
        if ( DuplicateNode ( curr, pn ) == 0 )
            return ( 1 ); /* bail out if DuplicateNode says to */

    prev->Next = pn;
    pn->Next = curr;

    Head = dummy.Next;
    return ( 1 );
}

/*-------------------------------------------------------------
 * Handle the duplicate node. In this program,
 * we just delete the duplicate.
 *-------------------------------------------------------------*/
int DuplicateNode ( Link inlist, Link duplicate )
{
    FreeNode ( duplicate );
    return ( 0 );
}

int DeleteNode ( Link to_delete )
{
    Link curr,  /* the current node */
         prev;  /* the previous node */
    int  i;

    /*--- Is there anything in the list? ---*/
    if ( Head == NULL )
        return ( 0 );

    /*--- If so, step through the list looking for the node ---*/
    for ( prev = NULL, curr = Head;
        curr != NULL && ( i = NodeCmp ( to_delete, curr )) > 0;
        prev = curr, curr = curr->Next )
        /* loop around */ ;

    /*--- Found a match, so delete it ---*/
    if ( curr != NULL && i == 0 )
    {
        if ( prev )
            prev->Next = curr->Next;
        else              /* deleting Head */
```

```
            Head = curr->Next;

        FreeNode ( curr );
        NodeCount -= 1;
        return ( 1 );
    }

    return ( 0 );
}
int NodeCmp ( Link a, Link b )
{
    /* returns 1, 0, -1, depending on whether the data in
     * a is greater than, equal, or less than b.
     */

    /* if temps are unequal, return based on temp */
    if ( a->Temp != b->Temp )
        return ( a->Temp - b->Temp );

    /* else, return based on city's name */
    return strcmp ( a->City, b->City );
}

void CreateList ( void )
{
    Head = NULL;
    NodeCount = 0;
}

void FreeNode ( Link n )
{
    free ( n->City );
    free ( n );
}

void ShowNodes( void )
{
    Link pn;
    int count, median;

    /* count the nodes */
    for ( count = 0, pn = Head; pn; pn = pn->Next )
        count += 1;

    /* compute the median node */
    median = count / 2 + 1;

    /* step through the list printing cities and
     * temperatures. Announce the median temperature.
     */
    if ( count ) /* only print if there's a node */
    {
        /* initialize the needed variables */
        count = 0;      /* count of nodes we've printed */
        for ( pn = Head; pn; pn = pn->Next )
        {
            printf ( "%-20s: %3d", pn->City, pn->Temp );
            count += 1;
```

```
            if ( count == median )
                printf ( " --Median--" );
            printf ( "\n" );
        }
    }
    else
        printf ( "Empty list\n" );
}

/*--- main line ---*/
int main ( int argc, char *argv[] )
{
    FILE *fin;          /* file we'll be reading from */
    char buffer[128]; /* where we'll read the file into */

    struct Node n;     /* the node we add each time */

    if ( argc != 2 )
    {
        fprintf ( stderr, "Usage: citytemp filename.ext\n" );
        exit ( EXIT_FAILURE );
    }

    fin = fopen ( argv[1], "rt" );
    if ( fin == NULL )
    {
        fprintf ( stderr, "Cannot open/find %s\n", argv[2] );
        exit ( EXIT_FAILURE );
    }

    /* Create and initialize the linked list to empty */
    CreateList();

    /*--- main loop ---*/
    while ( ! feof ( fin ))
    {
        /* read a record consisting of a line of text */
        if ( fgets ( buffer, 127, fin ) == NULL )
            break;

        /* get rid of the trailing carriage return */
        buffer [ strlen ( buffer ) - 1 ] = '\0';

        /* copy the city name to the node to be added */
        n.City = strdup ( buffer + 3 );

        /* mark off the temperature and convert to int */
        buffer[3] = '\0';
        n.Temp = atoi ( buffer );
        /* add the node to the list */
        if ( AddNodeAscend ( &n ) == 0 )
        {
            fprintf ( stderr, "Error adding node. Aborting\n" );
            exit ( EXIT_FAILURE );
        }
    }

    ShowNodes();
```

```
    /* Now, delete something */
    printf( "\n" );
    DeleteNode ( Head );
    ShowNodes();

    while (Head && Head->Next)
    {
        printf ( "\n" );
        DeleteNode ( Head->Next );
        ShowNodes();
    }

    printf ( "\n" );
    DeleteNode ( Head );
    ShowNodes();

    fclose ( fin );
    return ( EXIT_SUCCESS );
}
```

用于该程序的一个示例数据文件可能如下所示：

```
-10Boise, ID
-05Missoula, MT
-05Missoula, MT
040Hartford, CT
060Chicago, IL
080Los Angeles, CA
080Albuquerque, NM
100Phoenix, AZ
100Yuma, AZ
030Detroit, MI
000Franchot, MN
10 Kankakee, IL
098Hilo, HI
```

来自该文件的输出将如下所示：

```
Boise, ID           : -10
Missoula, MT        :  -5
Franchot, MN        :   0
Kankakee, IL        :  10
Detroit, MI         :  30
Hartford, CT        :  40
Chicago, IL         :  60 --Median--
Albuquerque, NM     :  80
Los Angeles, CA     :  80
Hilo, HI            :  98
Phoenix, AZ         : 100
Yuma, AZ            : 100
```

程序清单2-1显示了如何创建以及遍历链表、按顺序添加节点，以及比较两个节点。它还包括有函数DeleteNode()，该函数展示了如何删除一个节点（尽管在主程序中没有用到该函数）。删除节点需要遍历链表，直至找到要删除的节点为止。一旦找到该节点，则使前一个节点中的指针指向链表中的下一个节点，这样，链表现在将完全绕过当前节点。然后程序将通过free()函数

把当前节点占用的内存返回给系统。

注意：为了使前一个节点准备好指向下一个节点，必须在遍历链表时维护一个单独的指针（称为 prev），以便跟踪哪个节点位于应该删除的节点之前。这种需求暗示了迄今为止讨论的链表的主要缺点之一：如果不做特殊处理，将不知道哪个节点指向当前节点。节点本身只能告诉你下一个节点是什么，而不能告诉你前一个节点是什么。

当无法避免对该信息的需求时，将当前指针与前一个指针作为一个指针（即 prev）来维护是可能的。我们通过 curr = prev- > Next 来加以实现，因此在用到 curr 的任何地方，我们改用 prev- > Next。

一种更巧妙的技术是：不将 prev 维护为指向链表节点的指针，而是维护为指向前一个节点的 Next 元素的指针。为此，我们将 prev 定义为 Link * prev。第 5 章中给出了使用这种技术的一个示例（程序清单 5-16（linsert. c））。当你希望处理链表的一个子集而又不允许从主链表中删除该子集时，这种技术的价值尤为突出。在第 5 章中实现链表的快速排序算法时，广泛采用了这种技术（程序清单 5-17（lquick1. c）和程序清单 5-1（lquick2. c））。拥有指向前一个节点的 Next 元素的指针的好处是：可以轻松地对链表的子集执行任何想要的修改，而又不会影响原来的那个更大的包含链表。

无论使用哪种技术，都要设法解决链表只有一个链的主要缺点：即无法说明当前节点的前一个节点是什么。解决方案是使链表中的每个节点具有两个链：一个指向前一个链，另一个指向后一个链。这样的链表称为**双向链表**（doubly linked list）。

注意：在程序清单 2-1 中，直接操纵链表的函数是通用的。在编码时，将其设计为与节点中所包含的具体数据无关。唯一的要求是：将指向下一个节点的链称为 Next。实际上，其中的两个函数必须与数据相关，这两个函数是：NodeCmp（）和 DuplicateNode（），前者用于比较两个节点中的数据以确定插入顺序，后者用于处理当插入的节点所具有的数据与现有节点相同时的情况。链表的所有通用实现都至少要求将这两个函数设计成适合于将要处理的具体数据。在程序清单 2-1 中，函数 DuplicateNode（ ）实际上并没有涉及数据，但是在其他程序中（比如第 3 章介绍的那些程序中），这个相同的函数可能需要操作数据。

在把程序清单 2-1 中的例程用于自己的目的时，可以根据需要定义 Node，并且小心保持将链命名为 Next。然后重新编写函数 DuplicateNode（）和 NodeCmp（）的代码，并且准备好运行它们。

程序清单 2-1 中展示的函数介绍了链表的基本操作。它们适合于快速而又随性的应用程序，但它们不能处理可能出现在复杂程序中的需求。例如，假设你需要维护多个具有不同数据类型的链表，这些函数将不足以处理这项任务。下面一节将详细描述怎样解决这样的问题，以及如何处理阻止快速而又随性的链表真正变得健壮的其他微妙问题。

2.1.1 双向链表

双向链表使用的节点包含数据以及指向前一个和后一个节点的链。拥有指向两个相邻节点的链带来了某些好处，特别是向前和向后遍历链表的能力。此外，在遍历链表时将不需要维护指向前一个节点的指针。因此，双向链表提供了巨大的便利。由于经常在链表上执行双向遍历，因此在双向链表的实现中建议定义一个额外的指针，用于跟踪链表中的最后一个节点。该指针称为尾

指针（tail），它具有与头指针相同的功能——但是位于链表的尾部。标准的双向链表如图 2-3 所示。

图 2-3 重要液体的短双向链表

注意：双向链表节点与单链表节点非常相似。不过实现双向链表的代码稍微复杂一些，因为每个插入和删除操作都需要操纵一个额外的指针。此外，当操作发生在链表尾部时，必须更新尾指针（通常称为 Tail）。曾经用过双向链表的人都知道，在编码的时候必须小心地使所有的指针都指向合适的位置，这是一项乏味的工作。因此，一些优秀的程序员通常都有一组通用的例程，并且总是使用它们来实现双向链表。在程序清单 2-2 到程序清单 2-4b 中给出了用于处理双向链表的一个功能全面的函数集合。这些函数称为原函数（primitive）（之所以这样命名，是因为它们的任务是做所有的底层工作），它们表现在两个方面：那些与数据无关的函数以及那些用于访问数据的函数。程序清单 2-4a 和 2-4b 给出了后一种类型的示例。

程序清单 2-2 通用双向链表的头文件

```
/*--- llgen.h ---------------------------- Listing 2-2 ---------
 *  Declarations for generic doubly linked lists.
 *  Used in conjucntion with llgen.c (Listing 2-3).
 *-------------------------------------------------------------*/
#ifndef LLGEN_H       /* make sure it's included only once */
#define LLGEN_H    1


struct Node {
    struct Node     *prev;  /* link to previous node */
    struct Node     *next;  /* link to next node */
    void            *pdata; /* generic pointer to data */
};

typedef struct Node *Link;

/* a linked list data structure */
struct List {
    Link            LHead;
    Link            LTail;
    unsigned int    LCount;
    void * ( * LCreateData )      ( void * );
    int    ( * LDeleteData )      ( void * );
    int    ( * LDuplicatedNode ) ( Link, Link );
    int    ( * LNodeDataCmp )     ( void *, void * );
};

/* The four functions specific to an individual linked list are:

   LCreateData:  is passed a pointer to an application-defined
                 object and is expected to return a pointer to
                 whatever is to be stored in the linked list.
```

```
    LDeleteData:  is passed a pointer to the object an application
                  has stored in a linked list. LDeleteData must
                  destroy the object.

    LDuplicatedNode: is passed two pointers. The first pointer is
                     to a node that you would like to add to a
                     linked list and the second is to a node that
                     is already in the list but is a duplicate of
                     the first pointer.
                     LDuplicatedNode returns:
                             0 -> do nothing to list
                             1 -> destroy duplicate
                             2 -> add duplicate to list

    LNodeDataCmp: is passed pointers to two application data
                  objects and must compare them, returning a
                  number that is < 0, zero, or > 0, depending on
                  the relationship between the first and second
                  objects.
*/

/*--- generic linked-list primitives ---*/
int  AddNodeAscend  ( struct List *, void * );
int  AddNodeAtHead  ( struct List *, void * );
struct List * CreateLList (
                void * ( * ) ( void * ),       /* create data */
                int    ( * ) ( void * ),       /* delete data */
                int    ( * ) ( Link, Link ),   /* duplicate   */
                int    ( * )  ( void *, void * )); /* compare */
Link CreateNode     ( struct List * , void * );
int  DeleteNode     ( struct List *, Link );
Link FindNode       ( struct List *, void * );
Link FindNodeAscend ( struct List *, void * );
Link GotoNext       ( struct List *, Link );
Link GotoPrev       ( struct List *, Link );
#endif
```

这个文件值得给予相当的关注。对 LLGEN_H 的初始测试确保该头文件只被包含一次。这个看似无关紧要的规定是对所有的头文件都应该养成的良好习惯。它不仅可以让你直接控制编译器正在做什么，而且可以提高编译速度。编译器中最慢的工作之一是读取源文件。这个方面的一个例子是：编译器供应商近来的趋势是为预编译的头文件提供编译选项（例如，Borland 和 Microsoft 编译器，以及 MS-DOS 和 Windows 环境下的其他编译器）。因此，编译器只读取一次相关的文件将可以加快编译速度。

注意 Node 的定义。它包含两个链：prev 和 next，然后定义了 NodeData，它是一个指向 void 的指针。指向 void 的指针是 C 语言中的通用指针。它们可以无错误地强制转换为任何数据类型。为了使双向链表的实现真正通用，我们不希望在通用函数中指定具体的数据。我们将通用函数设计成只操纵节点中的链。因此，它们不必了解具体数据是什么。这样，这些通用函数无须修改即可应用于许多不同类型的链表。当我们定义应用程序相关的函数时，需要将 void 指针强制转换为我们正在操作的数据类型。在程序清单 2-4a（llapp. h）中有两个示例数据的定义，分别为 NodeData1 和 NodeData2。

在定义了节点之后，现在将定义链表。结构 List 包含特定链表的所有数据项以及函数。前两个指针分别指向头节点和尾节点。LCount 域包含链表中节点的数量。最后，该结构中将出现 4 个指针，它们指向特定链表的函数。实际操纵链表节点中的数据的函数将只有这 4 个函数。它们与应用程序相关，并且必须为我们实现的每个链表定义它们。在程序清单 2-4b（llapp. c）中显示了这些函数的示例。

将链表定义为一种包含所有链表特有内容的结构，这意味着可以轻松地将新链表添加到程序中，甚至当这些链表存储广泛不同的数据类型时亦可如此。程序清单 2-5（lldriver. c）是一个使用了大多数链表原函数的程序。它将创建两个链表，同时在两个方向上遍历链表，添加和删除节点，以及打印链表。其输入由一个文本文件组成，该文件每行上有一个单词。这个驱动程序说明了怎样使用通用的和特定于应用程序的函数。为了创建可执行文件，需要编译 llgen. c、llapp. c 和 lldriver. c 这些文件。然后应该链接这些程序以构成可执行文件。在第 3 章中展示的一个程序（程序清单 3-4（wordlist. c））说明了怎样使用这些函数来初始化和操纵两个具有不同数据类型的链表。其中一个链表具有上百个实例，它们显示在一个表中。

程序清单 2-3 包含链表的通用原函数。这些函数只操纵节点中的链；因此它们是完全通用的，无需改变即可使用。

程序清单 2-3　双向链表的原函数

```
/*--- llgen.c ------------------------------ Listing 2-3 --------
 *  Generic primitive functions for doubly linked lists.
 *  Contains no application-specific functions.
 *  Functions are in alphabetical order.
 *-------------------------------------------------------------*/

#include <stdlib.h>
#include <string.h>

#define IN_LL_LIB   1   /* in the library of primitives */

#include "llgen.h"

/*--- Aliases to make the code more readable ---*/

#define LLHead (L->LHead)       /* The head of the current list */
#define LLTail (L->LTail)       /* The tail of the current list */
#define NodeCount (L->LCount)   /* Nodes in the current list */

#define CreateData      (*(L->LCreateData))
#define DeleteData      (*(L->LDeleteData))
#define DuplicatedNode (*(L->LDuplicatedNode))
#define NodeDataCmp     (*(L->LNodeDataCmp))

/*-------------------------------------------------
 * Add a node at head: first allocate the space for
 * the data, then allocate a node with a pointer to
 * the data, then add the node to the list.
 *-------------------------------------------------*/
int AddNodeAtHead ( struct List *L, void *nd )
{
    Link pn;
```

```
    pn = CreateNode ( L, nd );
    if ( pn == NULL )
        return ( 0 );

    /*--- Add the node ---*/
    if ( LLHead == NULL )   /* is it the first node? */
    {
        LLHead = LLTail = pn; /*--- yes ---*/
    }
    else                      /*--- no  ---*/
    {
        LLHead->prev = pn; /* first goes node before Head */
        pn->next = LLHead; /* put Head next */
        LLHead = pn;       /* then point Head to us */
    }

    NodeCount += 1;
    return ( 1 );
}

/*-------------------------------------------------
 * Add ascending. Adds a node to an ordered list.
 *-------------------------------------------------*/
int AddNodeAscend ( struct List *L, void *nd )
{
    Link        pn;         /* to node we're creating */
    Link        prev, curr; /* our current search */
    struct Node dummy;      /* a dummy node */
    int         compare;

    pn = CreateNode ( L, nd );
    if ( pn == NULL )
        return ( 0 );

    /* attach dummy node to head of list */
    dummy.next = LLHead;
    dummy.prev = NULL;
    if ( dummy.next != NULL )
        dummy.next->prev = &dummy;

    prev = &dummy;
    curr = dummy.next;
    for ( ; curr != NULL; prev = curr, curr = curr->next )
    {
        compare = NodeDataCmp ( pn->pdata, curr->pdata );
        if ( compare <= 0 )
            break; /* new node equals or precedes curr */
    }

    if ( curr != NULL && compare == 0 )
    {
        compare = DuplicatedNode ( pn, curr );
        if ( compare == 2 )
            /* do nothing -- will get inserted */;
        else
        {
```

```
            /* first, repair the linked list */
            LLHead = dummy.next;
            LLHead->prev = NULL;

            /* delete the duplicated node, if appropriate */
            if ( compare == 1 )
            {
                DeleteData( pn->pdata );
                free ( pn );
            }
            return ( 1 );
        }
    }

    prev->next = pn;
    pn->prev = prev;
    pn->next = curr;
    if ( curr != NULL )
        curr->prev = pn;
    else
        LLTail = pn; /* this node is the new tail */

    NodeCount += 1;

    /* now, unhook the dummy head node */
    LLHead = dummy.next;
    LLHead->prev = NULL;
    return ( 1 );
}

/*-------------------------------------------------------------
 * Creates a linked-list structure and returns a pointer to it.
 * On error, returns NULL. This functions accepts pointers
 * to the four list-specific functions and initializes the
 * linked-list structure with them.
 *-------------------------------------------------------------*/
struct List * CreateLList (
                void * ( * fCreateData ) ( void * ),
                int    ( * fDeleteData ) ( void * ),
                int    ( * fDuplicatedNode ) ( Link, Link ),
                int    ( * fNodeDataCmp )  ( void *, void * ))
{
    struct List * pL;

    pL = (struct List *) malloc ( sizeof ( struct List ));
    if ( pL == NULL )
        return NULL;
    pL->LHead = NULL;
    pL->LTail = NULL;
    pL->LCount = 0;

    pL->LCreateData = fCreateData;
    pL->LDeleteData = fDeleteData;
    pL->LDuplicatedNode = fDuplicatedNode;
    pL->LNodeDataCmp = fNodeDataCmp;
```

```
    return ( pL );
}

/*-------------------------------------------------------------
 * Creates a node and then calls the application-specific
 * function CreateData() to create the node's data structure.
 * Returns NULL on error.
 *-------------------------------------------------------------*/
Link CreateNode ( struct List *L, void *data )
{
    Link new_node;

    new_node = (Link) malloc ( sizeof ( struct Node ));
    if ( new_node == NULL )
        return ( NULL );

    new_node->prev = NULL;
    new_node->next = NULL;

    /*--- now call the application-specific data allocation ---*/
    new_node->pdata = CreateData( data );
    if ( new_node->pdata == NULL )
    {
        free ( new_node );
        return ( NULL );
    }
    else
        return ( new_node );
}

/*-------------------------------------------------------------
 *  Deletes the node pointed to by to_delete.
 *  Function calls list-specific function to delete data.
 *-------------------------------------------------------------*/
int DeleteNode ( struct List *L, Link to_delete )
{
    Link pn;
    if ( to_delete == NULL )        /* Double check before */
        return ( 0 );               /* deleting anything.  */

    if ( to_delete->prev == NULL )  /* we're at the head */
    {
        LLHead = to_delete->next;   /* update head */
        LLHead->prev = NULL;        /* update next node */
    }

    else if ( to_delete->next == NULL )
    {                               /* we're at the tail */
        pn = to_delete->prev;       /* get the previous node */
        pn->next = NULL;
        LLTail = pn;                /* update tail */
    }

    else                        /* we're in the list */
    {
        pn = to_delete->prev;       /* get the previous node */
```

```
        pn->next = to_delete->next; /* update previous node to */
                                    /* point to the next one. */
        pn = to_delete->next;       /* get the next node */
        pn->prev = to_delete->prev; /* update it to point to */
                                    /* the previous one. */
    }

    DeleteData ( to_delete->pdata );  /* delete the data */
    free ( to_delete );               /* free the node */

    NodeCount -= 1;

    return ( 1 );
}

/*-------------------------------------------------------------
 *  Finds node by starting at the head of the list, stepping
 *  through each node, and comparing data items with the search
 *  key. The Ascend version checks that the data in the node
 *  being examined is not larger than the search key. If it is,
 *  we know the key is not in the list. Returns pointer to node
 *  on success or NULL on failure.
 *-------------------------------------------------------------*/
Link FindNode ( struct List *L, void *nd )
{
    Link pcurr;             /* the node we're examining */

    if ( LLHead == NULL )         /* empty list */
        return ( NULL );

    for ( pcurr = LLHead; pcurr != NULL; pcurr = pcurr->next)
    {
        if ( NodeDataCmp ( nd, pcurr->pdata ) == 0 )
            return ( pcurr );
    }
    return ( NULL );              /* could not find node */
}

Link FindNodeAscend ( struct List *L, void *nd )
{
    Link    pcurr;          /* the node we're examining */
    int cmp_result;

    if ( LLHead == NULL )         /* empty list */
        return ( NULL );
    for ( pcurr = LLHead; pcurr != NULL; pcurr = pcurr->next)
    {
        cmp_result =  NodeDataCmp ( nd, pcurr->pdata );

        if ( cmp_result < 0 )
             return ( NULL );     /* too far */

        if ( cmp_result == 0 )    /* just right */
             return ( pcurr );
    }

    return ( NULL );              /* could not find node */
}
```

每个函数获取包含链表的各个数据项的结构的地址作为它的第一个参数。为了简便起见，将这个指针称为 L。你将注意到某些函数需要能够检查但不能修改节点的数据。例如，函数 AddNodeAscend( ) 以升序添加节点，需要能够将准备添加的节点数据与当前正在检查的节点数据（由 LLCurr 所指向的节点）进行比较。为此，它将调用特定于链表的比较函数，在 List 结构中由 LNodeDataCmp 指向该函数。利用出现在程序清单顶部的宏简化通过指向函数的指针来调用函数的语法：

```
#define NodeDataCmp (*(L->LNodeDataCmp))
```

LNodeDataCmp 是 List 中指向函数的指针的名称。L 是指向我们正在访问的 List 结构的指针；因此，L- > LNodeDataCmp 是实际的函数指针。通过使用 * 间接引用它来调用函数。因此，通过使用宏 NodeDataCmp（a，b），就具有调用该函数并给它传递参数 a 和 b 的作用。所有特定于应用程序的函数都采用这种方法。

注意：如果你只是希望将节点添加到链表中，而不关心它是否以升序排序，就可以使用 AddNodeAtHead( ) 函数。如果发生错误，所有这些函数以及与具体数据相关的函数（除了比较函数之外）都会返回 0；成功时则会返回 1。如果函数 AddNodeAscend( )（它按升序添加节点）遇到一个具有匹配值的节点，它首先会调用特定于应用程序的函数 DuplicatedNode( )，让你决定将如何处理。然后它返回三个可能的值之一：0，指示错误；1，指删除重复的节点；2，指向链表中添加重复的节点。

由于一个有趣的细节，删除节点需要一个特定于应用程序的函数。假定节点的数据包含一个指向字符串的指针。如果只是简单地删除节点（例如，使用 free( ) 函数）指向字符串的指针就会消失，它本应如此，但是在内存中仍会为字符串本身分配空间，从而占用内存空间。它变成了一个“孤儿”；它仍然是活动的数据，但是没有任何指针知道它的存在。因此，要彻底地处理一个节点，首先必须删除数据（使用特定于应用程序的函数 LDeleteData( )），然后可以释放该节点。

对于创建节点也是如此。不能简单地复制一个指向数据的指针。假定该指针指向一个缓冲区，一旦把节点添加到链表中就会重用这个缓冲区。通用函数无法知道传递给它的数据的持久性。因此，要创建节点，必须编写另一个特定于应用程序的函数 LCreateNode( )。这个函数将字符串复制到新的区域中（通过 strdup( ) 函数），并且将那个地址用作字符串的地址。这样，通用函数总是能够确保它们所处理的数据的完整性。这些问题在大多数链表处理中通常都会被忽略，它们称为**数据耐力**（data endurance）问题。

程序清单 2-4a 中所示的头文件声明了应用程序特有的数据和函数。在这个例子中，定义了两个链表的数据。第一个链表是一个包含 word、指向字符串的指针以及 u（无符号整数）的结构；第二个链表是一个只包含字符串的结构。

程序清单 2-4a　链表的应用程序特有方面的头文件

```
/*--- llapp.h ----------------------------- Listing 2-4a -------
 *  Application-specific data for linked list in lldriver.c (2-5)
 *  Used in conjunction with llapp.c (Listing 2-4b).
 *-------------------------------------------------------------*/
#ifndef LLAPP_H
#define LLAPP_H 1
```

```
/*
 *  Our first list's nodes consist of a pointer to
 *  a word and a count of occurrences.
 */

struct NodeData1 {
    char *word;
    unsigned int u;
};

typedef struct NodeData1 * pND1;

extern void * CreateData1 ( void * );
extern int    DeleteData1 ( void * );
extern int    DuplicatedNode1 ( Link, Link );
extern int    NodeDataCmp1 ( void *, void * );


/*
 *  Our second list's nodes consist of a
 *  pointer to a word.
 */

struct NodeData2 {
    char *word;
};

typedef struct NodeData2 * pND2;

extern void * CreateData2 ( void * );
extern int    DeleteData2 ( void * );
extern int    DuplicatedNode2 ( Link, Link );
extern int    NodeDataCmp2 ( void *, void * );

#endif
```

在第3章中，程序将使用这些结构来扫描文本。第一个链表包含唯一的单词以及它们出现的次数，第二个链表包含可疑的单词。还为每个链表声明了4个特定于应用程序的函数，用于处理数据。这些函数定义在程序清单2-4b中。

程序清单2-4b　程序清单2-5中的链表的应用程序特有方面的代码

```
/*--- llapp.c ---------------------------- Listing 2-4b --------
 *  Application-specific functions for linked-list examples.
 *  Replace these routines with your own.
 *-------------------------------------------------------------*/

#include <stdlib.h>          /* for free() */
#include <string.h>          /* for strcmp() and strdup() */

#include "llgen.h"
#include "llapp.h"

/* data is a pointer to a string */
void * CreateData1 ( void * data )
{
```

```
    struct NodeData1 * new_data;

    /*--- allocate our data structure ---*/
    if ((new_data = malloc ( sizeof ( struct NodeData1 ))) == NULL)
        return ( NULL );

    /*--- move the values into the data structure ---*/
    new_data->u    =  1;
    new_data->word =  strdup ( (char *) data );

    if ( new_data->word == NULL )   /* error copying string */
    {
        free ( new_data );
        return ( NULL );
    }
    else
        return ( new_data ); /* return a complete structure */
}

int DeleteData1 ( void * data )
{
    /*
     * In this case, NodeData1 consists of: a pointer and an int.
     * The integer will be returned to memory when the node
     * is freed. However, the string must be freed manually.
     */
     free ( ((pND1) data)->word );
     return ( 1 );
}
/*-------------------------------------------------------------
 * This function determines what to do when inserting a node
 * into a list if an existing node with the same data is found
 * in the list. In this case, since we are counting words, if a
 * duplicate word is found, we simply increment the counter.
 *
 * Note this function should return one of the following values:
 *      0       an error occurred
 *      1       delete the duplicate node
 *      2       insert the duplicate node
 * Any other processing on the duplicate should be done in this
 * function.
 *-------------------------------------------------------------*/

int DuplicatedNode1 ( Link new_node, Link list_node )
{           /* adding an occurrence to an existing word */

    pND1 pnd = list_node->pdata;
    pnd->u += 1;
    return ( 1 );
}

int NodeDataCmp1 ( void *first, void *second )
{
    return ( strcmp ( ((pND1) first)->word,
                      ((pND1) second)->word ));
}
```

```
/*=== Now the functions for the second linked list ===*/

void * CreateData2 ( void * data )
{
    struct NodeData2 * new_data;

    /*--- allocate the data structure ---*/
    if ((new_data = malloc ( sizeof ( struct NodeData2 ))) == NULL)
        return ( NULL );

    /*--- move the values into the data structure ---*/
    new_data->word =  strdup ( (char *) data );

    if ( new_data->word == NULL )   /* error copying string */
    {
        free ( new_data );
        return ( NULL );
    }
    else
        return ( new_data );
}

int DeleteData2 ( void * data )
{
    /*
     * In this case, NodeData2 consists of a pointer.
     * The string must be freed manually.
     */

     free ( ((pND2) data)->word );
     return ( 1 );
}

/* this list inserts duplicated nodes */
int DuplicatedNode2 ( Link new_node, Link list_node )
{
    return ( 2 );
}

int NodeDataCmp2 ( void *first, void *second )
{
    return ( strcmp ( ((pND2) first)->word,
                      ((pND2) second)->word ));
}
```

现在，在程序清单2-5中给出了这些函数的驱动程序。

程序清单2-5 使用链表原函数的驱动程序

```
/*--- lldriver.c ------------------------- Listing 2-5 -------
 *  Reads in text words from the file specified on the command
 *  line and places them into two linked lists. Then exercises
 *  a variety of linked-list activities, printing the results
 *  at every step.
 *  Must be linked to linked-list primitives in Listings 2-2
 *  through 2-4b.
 *-----------------------------------------------------------------*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "llgen.h"          /* Header for generic linked lists */
#include "llapp.h"          /* Header for appl.'s linked lists */

int main ( int argc, char *argv[] )
{
    char    word[64];       /* the raw word from the file */
    int     count;

    struct  List *L1, *L2;  /* two different linked lists */
    Link    w1, w2, w3;     /* cursors used to walk lists */

    FILE    *fin;           /* the input file             */

    if ( argc != 2 )
    {
        fprintf ( stderr, "Error! Usage: lldriver filename\n" );
        exit ( EXIT_FAILURE );
    }

    fin = fopen ( argv[1], "rt" );
    if ( fin == NULL )
    {
        fprintf ( stderr, "Could not find/open %s\n", argv[1] );
        exit ( EXIT_FAILURE );
    }

    /*--- set up linked-list data structures ---*/
    L1 = CreateLList ( CreateData1,      /* in llapp.c */
                       DeleteData1,      /*     "      */
                       DuplicatedNode1,  /*     "      */
                       NodeDataCmp1 );   /*     "      */

    L2 = CreateLList ( CreateData2,      /* in llapp.c */
                       DeleteData2,      /*     "      */
                       DuplicatedNode2,  /*     "      */
                       NodeDataCmp2 );   /*     "      */

    if ( L1 == NULL || L2 == NULL )
    {
        fprintf ( stderr, "Error creating linked list\n" );
        exit ( EXIT_FAILURE );
    }

    /*--- begin processing file ---*/

    while ( fgets ( word, 64, fin ) != NULL )
    {
        if ( strlen ( word ) > 0 )
            word[strlen ( word ) - 1] = 0; /* strip tail \n */
     /* now, add the word to both lists */
     if ( ! AddNodeAscend ( L1, word ))
         fprintf ( stderr,
              "Warning! Error while adding node to L1.\n" );
```

```
    if ( ! AddNodeAtHead ( L2, word ))
        fprintf ( stderr,
             "Warning! Error while adding node to L2
}
fclose ( fin );

/* now, walk the lists */

printf( "L1 contains %u items:\n", L1->LCount );
for ( w1 = L1->LHead; w1 != NULL; w1 = w1->next )
     printf("  %s occured %d times.\n",
            ((pND1) (w1->pdata))->word,
            ((pND1) (w1->pdata))->u );

printf( "L2 contains %u items:\n", L2->LCount );
for ( w1 = L2->LHead; w1 != NULL; w1 = w1->next )
    printf ( "  %s\n", ((pND2) (w1->pdata))->word );

/* both ways at once */

printf ( "L2 contains %u items:\n", L2->LCount );
w1 = L2->LHead;
w2 = L2->LTail;
for ( ; w1 != NULL && w2 != NULL;
                    w1 = w1->next, w2 = w2->prev )
    printf( "  %30s  %30s\n",
            ((pND2) (w1->pdata))->word,
            ((pND2) (w2->pdata))->word );

/* "Find" each node and delete every other one */

count = 0;
w1 = L2->LHead;
while ( w1 != NULL )
{
    w3 = FindNode ( L2, w1->pdata );
    if ( w3 != 0 )
    {
        printf ( "Found node %s",
                   ((pND2) (w3->pdata))->word );
        count += 1;
        w1 = w3->next;
        if ( count & 1 )
        {
               DeleteNode ( L2, w3 );
               printf ( " and deleted it." );
            }
            printf( "\n" );
        }
        else
            w1 = NULL;
    }
    printf ( "L2 contains %u items:\n", L2->LCount );
    w1 = L2->LHead;
    w2 = L2->LTail;
    for ( ; w1 != NULL && w2 != NULL;
                        w1 = w1->next, w2 = w2->prev )
        printf ( "  %30s  %30s\n",
```

```
                ((pND2) (w1->pdata))->word,
                ((pND2) (w2->pdata))->word );

    return ( EXIT_SUCCESS );
}
```

2.1.2 链表的其他特征

链表被称为动态数据结构，这是因为它们可以被扩展或压缩。这种特征把链表与数组及其他静态结构区分开，对于静态结构，一旦指定了其最大尺寸，以后将不能轻松地改变。无论何时需要在内存中存储数量不确定的数据项时，采用动态数据结构很可能是最佳的方法。使用链表所带来的最大不便是它们的潜在长度。搜索一个很长的链表可能很费时，而且如果不得不重复搜索很长的链表，其代价可能变得令人无法忍受。有多种技术可以用来减小开销。其中一种技术是在链表中以某种顺序放置节点，以便当你想确定链表是否包含给定的节点时，将不需要检查整个链表。另一种方法是将最近访问的节点放到链表头部。如果将要检查的数据趋向于聚集在一起，这将节省相当多的时间。也就是说，一旦操作了一个节点，很可能不久后就会再次检查它。可以在构建缓存或者在为像 C 这样的语言构建编译器时使用这种方法，在这种语言中，例如，局部变量定义在将要使用它们的函数内。因此，一旦局部变量出现在函数中，它很可能不久后就会再次出现在多个位置。之后，对它的使用将会减少，并且包含它的节点将会在链表中后移，而其他访问时间更近的节点则会向头部移动。

这些方法只有在链表不太长的情况下才有效。如果你预期会在内存中存储上百份数据，链表的效率太低，并且应当使用其他的动态数据结构。在第 3 章中介绍的散列表维护一个链表的表格，并且基于数据的值来确定应该使用哪个链表来存储节点。一种能够保证近似最优性能并且与所存储数据项的数量无关的数据结构是二叉树，第 6 章中研究了二叉树的许多变体。

我们分别仔细地介绍程序清单 2-3、2-4a 和 2-4b 中的函数，以便你可以轻松地使用它们。但是你可能想知道通常应该使用哪种类型的链表——单链表或双向链表。如果你的应用程序并不需要使用双向链表，使用单链表就是一个合适的选择。如果性能和空间是需要重点考虑的因素，就可以使用单链表；对于所有其他的情况（绝大多数情况），都应该使用双向链表。建立第二个指针的额外开销只会给性能带来几乎觉察不到的负担。可以使用一种性能测量和跟踪工具轻松地验证这一点。双向链表的优点是：如果应用程序应该不断改变并且需要更深入地访问链表，那么双向链表将比较合适。不管怎样，都必须把单链表转换为具有两个链的链表。

2.2 栈和队列

现在探讨另外两种简单的数据结构：栈（stack）和队列（queue）。这些非常有用的数据结构是应用程序中经常出现的角色。用数组或链表都可以很好地实现它们。重申一遍，链表实现允许无限地调整大小，而数组则要求在开发过程的早期就确定一个最大大小。对于栈和队列，建议在编译时而不是在运行时确定最大大小；因此，数组是实现这些数据结构的合理方式。利用链表实现栈和队列并不是一项困难的任务。在程序清单 2-3、2-4a 和 2-4b 中介绍了所有需要的原函数。

2.2.1 栈的特征

栈是一种简单的数据结构，可以最容易地表示为数组，其中添加和删除数据项是在同一端进行的。它之所以被称为栈，是因为它就像是堆叠在一起的盘子。盘子放在栈的顶部并从顶部取走。仅当取走了所有其他的盘子之后，才能接触到底下的盘子。与盘子一样，栈的重要特性是最后加入栈中的元素是第一个被删除的。这种存储数据的方法称为后进先出（last in, first out, LIFO）。

栈有许多应用。在大多数情况下，当程序员需要知道什么动作或数据位于当前正在进行的动作之前时，就可以使用栈。例如，当今的大多数编译器通过使用栈来实现高级语言。当调用函数或子例程时，就把主代码行的下一个指令的地址放到栈上。然后，当函数返回时，程序就会从栈中获取该地址，并从那一点继续向下执行。在函数调用了其他函数的情况下，将把每一个返回地址都放到栈上。这样，当函数结束时，就可以找到它们在栈中的地址。

许多其他的应用程序需要跟踪它们所在的位置或者它们怎样到达某一点，并且几乎不变的是，栈就是为此目的而选择的数据结构。在栈上放置一个项目（比如一个地址或数据项）被称为把项目**压入**（push）到栈上（图2-4）。删除一个项目被称为从栈中弹出（pop）一个项目。栈顶端的入口被作为**栈顶**（top），其他项目被认为是栈中更下面（down）的项。有时，这个术语可能会被混淆；一定要正确地使用它。

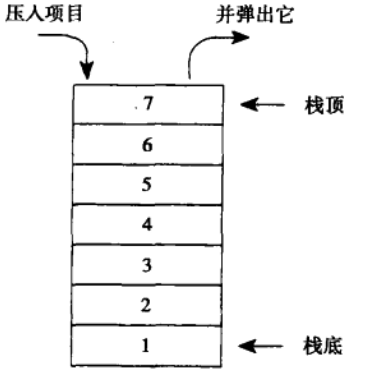


图2-4 一个典型的栈。注意项目7是压入到栈上的最后一个项目，它也将第一个弹出

用于操作栈的原函数如下：创建栈、清除栈（清除栈中的所有元素）、压入项目、弹出项目、在不弄乱栈的情况下查看栈中的任意项目。程序清单2-6（stacks.h）和2-7（stacks.c）中给出了栈的原函数。该代码利用了程序清单2-3到2-4b中展示的双向链表的原函数。程序清单2-5中特定于应用程序的内容将栈中的每个元素声明为一个结构，其中每个结构包含一个整数和一个字符。该结构将用在程序清单2-8中。

程序清单2-6 将栈的原函数实现为一个数组的头文件

```
/*--- stacks.h ---------------------------- Listing 2-6 ---------
 *  Header file for stack operations
 *-------------------------------------------------------------*/

#ifndef STACKS_H
#define STACKS_H    1

#ifdef IN_STACK_LIB
#define StkExtern
#else
#define StkExtern   extern
#endif
```

```
struct stack_struct {
    struct StkElement *base;        /* point to base of stack */
    int               stack_size;   /* number of elements */
    int               min_stack;    /* bottom-most element */
    int               max_stack;    /* last possible element */
    int               top;          /* current top */
};

typedef struct stack_struct Stack;

StkExtern void   ClearStack    ( Stack * );
StkExtern Stack* CreateStack   ( int );
StkExtern int    PopElement    ( Stack *, struct StkElement * );
StkExtern int    PushElement   ( Stack *, struct StkElement * );
StkExtern struct StkElement *
                 ViewElement   ( Stack *, int );

/*--- Application-specific material goes below ---*/

StkExtern struct StkElement  {
             int   line_no;
             char  opener;
};

#endif
```

程序清单2-7将栈实现为在程序清单2-6中定义的简单的元素数组。CreateStack( ) 函数为how_many元素的数组分配内存。然后，初始化用于保存栈的最大和最小范围的变量，它们分别是：MaxStack和MinStack。这些变量很重要，因为该函数必须确保把项目压入栈时不会超过栈的最大尺寸（如果是这样，这将会覆盖内存中的其他数据项），以及确保这些项目不会经过栈底弹出（如果是这样，它将返回实际上不在栈上的不正确的数据项）。最后，该函数将把StackTop（用于跟踪哪个栈元素位于栈顶的变量）初始化为一个非法值（-1），指示栈是空的。

程序清单2-7 栈的原函数

```
/*--- stacks.c --------------------------- Listing 2-7 ---------
 * Primitives for array-based stacks
 *-------------------------------------------------------------*/

#define IN_STACK_LIB 1

#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "stacks.h"

/*-------------------------------------------------------------
 *  clear the stack by pointing the top of the stack
 *  at an invalid item; that is, the stack is empty.
 *-------------------------------------------------------------*/
void ClearStack ( Stack *this_stack )
{
    this_stack->top = -1;
}
```

```
/*-------------------------------------------------------------
 *  allocate the stack, set the maximum and minimum bounds
 *  on the stack, and show that the stack is empty.
 *-------------------------------------------------------------*/
Stack *CreateStack ( int how_many )
{
    Stack *pstk;

    assert ( how_many > 0 );    /* make sure the size is legal */

    pstk = (Stack *) malloc ( sizeof ( Stack ));
    if ( pstk == NULL )
        return ( NULL );

    pstk->stack_size = how_many;

    pstk->base = ( struct StkElement * )
        malloc ( how_many * sizeof ( struct StkElement ));

    if ( pstk->base == NULL ) /* error in allocating stack */
        return ( NULL );

   pstk->min_stack = 0;
   pstk->max_stack = how_many - 1;

   ClearStack ( pstk );

   return ( pstk );
}
/*-------------------------------------------------------------
 *  pop an element from the stack. If stack is not already empty,
 *  copy the element, and decrement stack top.
 *-------------------------------------------------------------*/
int PopElement ( Stack *this_stack,
                 struct StkElement * destination )
{
    if ( this_stack->top == -1 ) /* stack empty, return error */
        return ( 0 );

    memmove ( destination,
              &(( this_stack->base )[this_stack->top] ),
              sizeof ( struct StkElement ));

    this_stack->top -= 1;

    return ( 1 );
}

/*-------------------------------------------------------------
 *  push an element onto the stack. If stack is not already full,
 *  point StackTop to the next slot, and copy the new element.
 *-------------------------------------------------------------*/
int PushElement ( Stack *this_stack, struct StkElement * to_push )
{
    /* is stack full? */
    if ( this_stack->top == this_stack->max_stack )
        return ( 0 );

    this_stack->top += 1;
```

```
    memmove ( &(( this_stack->base )[this_stack->top] ), to_push,
                        sizeof ( struct StkElement ));

    return ( 1 );
}

/*-------------------------------------------------------------
 *  view an element on the stack. Function is passed a value that
 *  specifies the element's position in terms of its distance
 *  from the top. 0 is the top, 1 is the element below the top,
 *  2 is the element below that. If an invalid value is passed,
 *  the function returns NULL; otherwise, it returns a pointer
 *  to the requested element.
 *-------------------------------------------------------------*/
struct StkElement * ViewElement ( Stack *this_stack,
                                  int which_element )
{
    if ( this_stack->top == -1 )
        return ( NULL );

    if ( this_stack->top - which_element < 0 )
        return ( NULL );

    return ( &(( this_stack->base )
                    [this_stack->top - which_element] ));
}
```

当通过 PushElement( ) 函数将项目压入到栈上时，将会使 StackTop 递增，以引用下一个可用的元素，然后在那里复制要压入的元素。弹出一个元素只是把顶部的元素复制到一个指定的目的地，并且使 StackTop 递减，以指向栈上的下一个项目。如果弹出的元素是栈上的最后一个项目（因此它位于 Stack［0］），那么仍然会使 StackTop 递减，并且它的值 -1 再次指示栈为空。

程序清单 2-8（braces. c）使用了栈的原函数，它读取一个 C 程序来查明其中是否有任何不匹配的圆括号、方括号或大括号。它一次一行地读取一个程序；当找到一个开始大括号或相关项目时，就把该项目及其行号压入到栈上。然后，当找到一个封闭项目时，就弹出栈顶的项目。如果弹出的项目是与当前封闭字符匹配的正确项目，则继续执行下面的处理。如果两个项目互不匹配，就会生成一条错误消息，指出开始项目和封闭项目的字符和行号。

程序清单 2-8　用于检查大括号、方括号和圆括号的可以增长见识的 C 程序

```
/*--- braces.c --------------------------- Listing 2-8 ---------
 * Checks that braces, brackets, and parentheses are properly
 * paired in a C program. If they're not, an error message
 * is printed to stderr saying at what line the unmatched
 * opening item is found. Uses a stack.
 *
 * Usage: braces filename.ext
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stacks.h"
```

```
int main ( int argc, char *argv[] )
{
    FILE *fin;                  /* file we'll be reading from */
    char buffer[128];           /* read file into this buffer */

    int line_count;             /* current line count */
    struct StkElement *stk_el;  /* scratch stack element */
    Stack *stk;                 /* the stack we will use */
    char ch;                    /* character we're examining */
    int i;                      /* for loop count */

    if ( argc != 2 )
    {
        fprintf ( stderr, "Usage: braces filename.ext\n" );
        exit ( EXIT_FAILURE );
    }

    fin = fopen ( argv[1], "rt" );
    if ( fin == NULL )
    {
        fprintf ( stderr, "Cannot open/find %s\n", argv[1] );
        exit ( EXIT_FAILURE );
    }

    /* Create and initialize the stack */

    stk = CreateStack ( 40 );   /* create a stack of 40 items */
    if ( stk == NULL )
    {
        fprintf ( stderr, "Insufficient Memory\n" );
        exit ( EXIT_FAILURE );
    }
/* Create the scratch stack element */

stk_el = (struct StkElement *)
            malloc ( sizeof ( struct StkElement ));
if ( stk_el == NULL )
{
    fprintf ( stderr, "Insufficient memory\n" );
    exit ( EXIT_FAILURE );
}

line_count = 0;

while ( ! feof ( fin ))
{
    /* read a line of a C program */
    if ( fgets ( buffer, 127, fin ) == NULL )
        break;

    line_count += 1;

    /* scan and process braces, brackets, and parentheses */
    for ( i = 0; buffer[i] != '\0'; i++ )
    {
        switch ( ch = buffer[i] )
        {
        case '(':
```

```
        case '[':
        case '{':
            stk_el->opener  = ch;
            stk_el->line_no = line_count;
            if ( ! PushElement ( stk, stk_el ))
            {
                fprintf ( stderr, "Out of stack space\n" );
                exit ( EXIT_FAILURE );
            }
            break;
        case ')':
        case ']':
        case '}':
            if ( ! PopElement ( stk, stk_el ))
                fprintf ( stderr, "Stray %c at line %d\n",
                            ch,  line_count );
            else
            if (( ch == ')'&& stk_el->opener != '(' ) ||
                ( ch == ']'&& stk_el->opener != '[' ) ||
                ( ch == '}'&& stk_el->opener != '{' ))
                fprintf ( stderr,
                  "%c at line %d not matched by %c at line %d\n",
                          ch, line_count,
                          stk_el->opener, stk_el->line_no );
                   break;
               default:
                   continue;
               }
           }
    }

    /*  We are at the end of file. Are there unmatched items? */

    if ( ViewElement ( stk, 0 ) != NULL )
        while ( PopElement ( stk, stk_el ) != 0 )
            fprintf ( stderr, "%c from line %d unmatched\n",
                        stk_el->opener, stk_el->line_no );

    fprintf ( stderr, "Error checking complete\n" );

    fclose ( fin );
    return ( EXIT_SUCCESS );
}
```

一旦读取了整个C源文件，程序就会检查是否有任何项目保留在栈上。这些是没有找到匹配的封闭项目的开始项目。然后打印出它们以及错误信息。注意：在这个程序中建议使用栈，因为当发现封闭项目时我们需要知道所处的位置。

这个程序只用于演示。它可能会受到注释以及一些字符值的欺骗。还应该记住的是，当发现一个封闭项目时，该程序总是会弹出一个元素。这在一些情况下会工作；在其他情况下，你可能想保留未匹配的开始项目。这些改进看似简单，但是对于目前讨论的问题而言，可能过于复杂以致于无法实现。不过，如果你没有使用圆括号以及字符常量和注释之类的内容（或者如果你使用了它们，并且使它们保持匹配），这个程序将会智能地通告任何错误。出于简单起见，该程序假

定每一行的长度不超过127个字符，并且每一行都以‘\n’结尾。

2.2.2 队列的特征

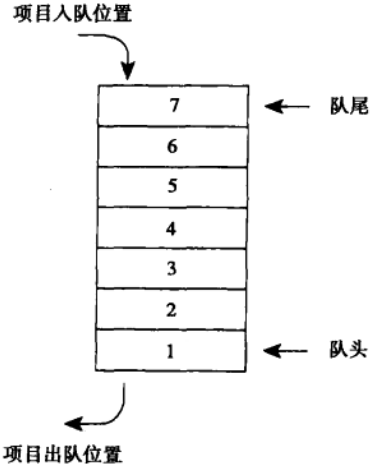


图2-5 一个典型的队列。注意项目1是第一个入队的项目，并将是第一个出队的项目

队列与栈类似。不过，与栈要求在同一端添加和删除数据项不同，队列允许你从一端添加数据项，而从另一端删除数据项。这样，第一个放置到队列中的项目将第一个被删除（如图2-5所示）。队列接近于人们在日常生活中的排队等候动作。

要将一个项目放到队列的尾部（队尾），必须采用**入队**（enqueue）动作；要从队头删除一个项目，必须采用**出队**（dequeue）动作。它们是队列的两种基本动作。

最简单形式的队列是一种容易实现的数据结构。不过，不久我们将发现，它看起来简单，但实际上隐藏了许多微妙的细节。首先是内存表示问题。一般使用数组或链表表示队列。在这两种情况下都存在同一个问题：如果保持在队尾添加元素并从队头删除元素，队列的数据结构将缓慢地在内存中迁移。当使用数组实现队列时，可以通过使数组循环来避开这个问题，这样一旦访问了数组中的最后一个元素，就可以继续访问数组中的第一个元素。这些数组通常称为**循环缓冲区**（circular buffer）。

链表实现会遇到相同的问题，只不过它具有稍微不同的形式：如果保持在队列中删除和添加节点，当分配和释放节点时，将使可用的内存碎片化。对于通过链表实现的队列（在程序清单2-9、2-10和2-11中实现了它），可以有两种选择。第一种选择是使用自己的函数来分配和返还内存。这个函数将分配一个节点池，然后使用和返回这些节点。第二种选择（在程序清单2-9中实现）是创建一个未使用节点的链表。当把项目添加到队列中时，就把节点从空闲链表移到队列中；当项目出队时，就把节点返回到空闲链表中。我们可以使用在本章前面介绍的链表函数（程序清单2-3到2-4b）。

程序清单2-9 基于链表的队列实现的头文件

```
/*--- qapp.h ------------------------------ Listing 2-9 --------
 *  Application-specific data items for linked-list queues.
 *-------------------------------------------------------------*/
#ifndef QAPP_H
#define QAPP_H 1

/*
 *  Our first list's nodes consist of a pointer to
 *  text and a priority level.
 */

#define TEXT_SIZE 60    /* the maximum size of our text string */
```

```
extern int DataCopy ( void *, void * );

/*
 * The data and functions for the queue
 */

struct NodeData1 {
    char *text;
    unsigned int priority;
};

typedef struct NodeData1 * pND1;

extern void * CreateData1 ( void * );
extern int    DeleteData1 ( void * );
extern int    DuplicatedNode1 ( Link, Link );
extern int    NodeDataCmp1 ( void *, void * );


/*
 *  The free list has identical nodes
 */

struct NodeData2 {
    char *text;
    unsigned int priority;
};
typedef struct NodeData2 * pND2;

extern void * CreateData2 ( void * );
extern int    DeleteData2 ( void * );
extern int    DuplicatedNode2 ( Link, Link );
extern int    NodeDataCmp2 ( void *, void * );

#endif
```

你应该非常熟悉程序清单2-9中所示的文件。程序清单2-10显示了用于两个链表的特定于应用程序的函数实现。

程序清单2-10　用于链表式队列的特定于应用程序的函数

```
/*--- qapp.c ----------------------------- Listing 2-10 --------
 *  Application-specific functions for queue examples.
 *  Replace these routines with your own.
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>          /* for free() */
#include <string.h>          /* for strcmp() and strdup() */

#include "llgen.h"
#include "qapp.h"

/*=== linked-list functions for queue ===*/

/*
```

```
 * our nodes come from the free list,
 * so this function is never called.
 */
void * CreateData1 ( void * data )
{
    return ( NULL );
}

int DeleteData1 ( void * data )
{
    /*
     * In this case, NodeData1 consists of a pointer and an int.
     * The integer will be returned to memory when the node
     * is freed. However, the string must be freed manually.
     */
    free ( ((pND1) data)->text );
    return ( 1 );
}
/*-------------------------------------------------------------
 * This function determines what to do when inserting a node
 * into a list if an existing node with the same data is found
 * in the list. In this case, since we are counting words, if a
 * duplicate word is found, we simply increment the counter.
 *
 * Note this function should return one of the following values:
 *      0       an error occurred
 *      1       delete the duplicate node
 *      2       insert the duplicate node
 * Any other processing on the duplicate should be done in this
 * function.
 *-------------------------------------------------------------*/

int DuplicatedNode1 ( Link new_node, Link list_node )
{
    return 2;
}

/* compare only the priority of the queue data */
int NodeDataCmp1 ( void *first, void *second )
{
    return (  ( ((pND2) first)->priority -
                ((pND2) second)->priority ));
}

/*=== Now the functions for the list of free nodes ===*/

/* data is a priority level (int) and text (string) */
void * CreateData2 ( void * data )
{
    struct NodeData1 * new_data;

    /*--- allocate our data structure ---*/

    new_data = malloc ( sizeof ( struct NodeData1 ));
    if ( new_data == NULL )
        return ( NULL );
```

```
    /*--- move the values into the data structure ---*/

    /*
     * we assign a priority of 0
     * and allocate a string of TEXT_SIZE + 1
     */
    new_data->priority =  0;
    new_data->text     =  (char *) malloc ( TEXT_SIZE + 1);

    if ( new_data->text == NULL )   /* error copying string */
    {
        free ( new_data );
        return ( NULL );
    }
    else
        return ( new_data ); /* return a complete structure */
}

int DeleteData2 ( void * data )
{
    /*
     * In this case, NodeData2 consists of a pointer.
     * The string must be freed manually.
     */

     free ( ((pND2) data)->text );
     return ( 1 );
}

/* this list inserts duplicated nodes */
int DuplicatedNode2 ( Link new_node, Link list_node )
{
    return 2;
}

/* this function is never called */
int NodeDataCmp2 ( void *first, void *second )
{
    return ( 0 );
}

/* function to copy our data */

int DataCopy ( void * dest, void * src )
{
    pND2 s, d;
    s = src;
    d = dest;

    if ( dest == NULL || src == NULL )
        return ( 0 );


    printf ( "About to copy %d - %s \n",
                s->priority, s->text );

    d->priority = s->priority;
```

```
    strncpy ( d->text, s->text, TEXT_SIZE );

    return ( 1 );
}
```

程序清单2-11（本章中的最后一个程序清单）是主驱动程序，它显示了如何利用这些函数实现队列。注意：可以给队列中的项目分配优先级。

程序清单2-11 优先级队列的实现

```
/*--- qdriver.c -------------------------- Listing 2-11 --------
 * Reads in a data file consisting of lines of text of the form
 *                          X9Message
 * where X = A for add to queue, D = delete, P = print the queue
 *       9 = priority of the queued item
 *       Message = string to enqueue
 * Note: actions D and P have no priority or message.
 * As each action is performed, a status message is printed.
 *
 * Must be linked with object files from qapp.c and llgen.c
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "llgen.h"
#include "qapp.h"

int dequeue ( struct List *, struct List *, void * );
int enqueue ( struct List *, struct List *, void * );

#define QMAX 100    /* maximum number of elements in a queue */

main ( int argc, char *argv [] )
{
    char     record[64];       /* the raw word from the file */
    int      count;
    void *   temp;             /* temporary data area */

    struct  List *queue,
                 *free_list;  /* our two queues */
FILE     *fin;              /* the input file */

if ( argc != 2 )
{
    fprintf ( stderr, "Error! Usage: qdriver filename\n" );
    exit ( EXIT_FAILURE );
}

fin = fopen ( argv[1], "rt" );
if ( fin == NULL )
{
    fprintf ( stderr, "Could not find/open %s\n", argv[1] );
    exit ( EXIT_FAILURE );
}
```

```
/*--- set up linked-list data structures for queues ---*/

queue
   = CreateLLList ( CreateData1,       /* in qapp.c */
                    DeleteData1,       /*      "    */
                    DuplicatedNode1,   /*      "    */
                    NodeDataCmp1 );    /*      "    */
free_list
   = CreateLLList ( CreateData2,       /* in qapp.c */
                    DeleteData2,       /*      "    */
                    DuplicatedNode2,   /*      "    */
                    NodeDataCmp2 );    /*      "    */

if ( queue == NULL || free_list == NULL )
{
    fprintf ( stderr, "Error creating queue\n" );
    exit ( EXIT_FAILURE );
}

/*--- allocate the free list ---*/

for ( count = 0; count < QMAX; count++ )
{
    if ( ! AddNodeAtHead ( free_list, record ))
    {
        fprintf
            ( stderr, "Could not create queue of %d\n",
                QMAX );
        exit ( EXIT_FAILURE );
    }
}

/*--- begin processing file ---*/
temp = CreateData2 ( NULL );
if ( temp == NULL )
{
    fprintf ( stderr, "Error creating temporary data area\n" );
    exit ( EXIT_FAILURE );
}

while ( fgets ( record, 64, fin ) != NULL )
{
    if ( strlen ( record ) > 0 )
        record[strlen ( record ) - 1] = 0; /* strip CR/LF */

    if ( *record == 'A' )   /* add */
    {
        ((pND2)temp)->priority = *( record + 1 ) - '0';
        ((pND2)temp)->text = record + 2;

        if ( enqueue ( queue, free_list, temp ) == 0 )
        {
            printf ( "Error enqueueing %d %s\n",
                     ((pND2)temp)->priority,
                     ((pND2)temp)->text );
            exit ( EXIT_FAILURE );
        }
```

```
    else
    {
        printf ( "Enqueued %d %s\n",
                 ((pND2)temp)->priority,
                 ((pND2)temp)->text );

        if ( queue->LCount == 0 )
            printf ( "Empty queue\n" );
        else
        {
            Link curr;
            printf ( "-------- List so far-------\n" );
            for ( curr = queue->LHead;
                  curr != NULL;
                  curr = curr->next )
              printf ( "%d %s\n",
                  ((pND2)(curr->pdata))->priority,
                  ((pND2)(curr->pdata))->text );
        }
    }
}
else
if ( *record == 'D' )   /* delete */
{
        if ( dequeue ( queue, free_list, temp ) == 0 )
        {
            printf ( "Error dequeueing %d %s\n",
                     ((pND2)temp)->priority,
                     ((pND2)temp)->text );
            return ( EXIT_FAILURE );
        }
        else
            printf ( "Dequeued %d %s\n",
                     ((pND2)temp)->priority,
                     ((pND2)temp)->text );
    }
    else
    if ( *record == 'P' )   /* print */
    {
        if ( queue->LCount == 0 )
            printf ( "Empty queue\n" );
        else
        {
            Link curr;
            printf ( "\n-------- List so far-------\n" );
            for ( curr = queue->LHead;
                  curr != NULL;
                  curr = curr->next )
              printf ( "%d %s\n",
                  ((pND2)(curr->pdata))->priority,
                  ((pND2)(curr->pdata))->text );
        }
    }
    else
        fprintf ( stderr, "Data error: %s\n", record );
}
```

```
    fclose ( fin );
    return ( EXIT_SUCCESS );
}
/*---------------------------------------------------------------
 * enqueue loads the data items in entry into the head node of
 * the free list, then adds that node to the queue based on
 * priority.
 *---------------------------------------------------------------*/

int enqueue ( struct List *lqueue, struct List *lfree,
              void *new_entry )
{
    Link curr, new_node;

    /* Are there any free nodes left? */
if ( lfree->LCount == 0 )
{
    fprintf ( stderr, "Exceeded maximum queue size\n" );
    return ( 0 );
}

/* load the data into the head of the free list */

new_node = lfree->LHead;

if ( DataCopy ( new_node->pdata, new_entry ) == 0 )
     return  ( 0 );

/* adding to an empty list? */

if ( lqueue->LCount == 0 )
{
    lfree->LHead = lfree->LHead->next;

    new_node->prev = NULL;
    new_node->next = NULL;

    lqueue->LTail = new_node;
    lqueue->LHead = new_node;

    lqueue->LCount = 1;
    lfree->LCount -= 1;

    return ( 1 );
}
else
/* Traverse the list to find the insertion position */

for ( curr = lqueue->LHead; ; curr = curr->next )
{
    if ( curr == NULL          /* at end of queue */
            ||                 /* or at insertion point */
            NodeDataCmpl ( new_entry, curr->pdata ) < 0
    {
        new_node = lfree->LHead;
        lfree->LHead = lfree->LHead->next;
```

```
        if ( curr == NULL )   /* if end of list */
        {
            new_node->prev = lqueue->LTail;
            new_node->next = NULL;
            new_node->prev->next = new_node;
            lqueue->LTail = new_node;
            }
            else
            {
                if ( curr->prev == NULL )   /* adding at head? */
                    lqueue->LHead = new_node;

                new_node->prev = curr->prev;
                new_node->next = curr;

                if ( curr->prev != NULL )
                    curr->prev->next = new_node;
                curr->prev = new_node;
            }

            lqueue->LCount += 1;

            /* update the free list */

            lfree->LCount -= 1;

            return ( 1 );
        }
        else
        {
            pND2 p1, p2;
            p1 = curr->pdata;
            p2 = new_entry;
            printf ( "searched at %d %s to insert %d %s\n",
                     p1->priority, p1->text,
                     p2->priority, p2->text );
        }
    }
}

/*-------------------------------------------------------------
 * dequeue takes a pointer that will be set to the data in the
 * node at the head of the queue. It then moves the node being
 * dequeued from the queue to the free list. Note that if you do
 * not use the dequeued data before next queue operation, the
 * data is lost, so copy it if you need to. Returns 0 on error.
 *-------------------------------------------------------------*/

int dequeue ( struct List *lqueue, struct List *lfree,
              void * our_data )
{
    Link dequeued_link;

    /* is there anything to dequeue? */
    if ( lqueue->LCount == 0 )
    {
        fprintf ( stderr, "Error dequeue from empty queue\n" );
```

```
        return ( 0 );
    }

    /* make a copy of the data being dequeued */

    if ( DataCopy ( our_data, lqueue->LHead->pdata ) == 0 )
        return ( 0 );

    /* remove the node from the queue */

    dequeued_link = lqueue->LHead;
    lqueue->LHead = lqueue->LHead->next;
    lqueue->LCount -= 1;

    /* add the node to the free list */

    dequeued_link->prev = NULL;
    dequeued_link->next = lfree->LHead;
    lfree->LHead = dequeued_link;
    lfree->LCount += 1;

    return ( 1 );
}
```

第3章 散　列

当数据项的数量事先不可知时，链表提供了一种在内存中存储数据的方法。链表的缺点是：它们的构造要求按顺序访问节点。也就是说，为了到达任意节点，都必须访问链表中它之前的所有节点。可以使用多种技术（比如对节点进行排序，或者把最近访问的节点放在靠近链表头部的位置）来减少顺序查找所需的时间；但是这些方法都无法消除查找本身是按顺序进行的要求。

为了提供对内存中存储的数据项的快速、随机访问，一种巧妙的解决方案是**散列表**（hash table）。C 语言中的常规表（比如 struct 数组）要求提前指定表中的元素数量。不过，散列表可以在表中存放数量不确定的项目，同时不会损失快速、近似随机的访问。

3.1 散列的概念

散列表的秘密是：它提供了一种方法，可以计算出特定的数据存储在表中的什么位置。这种测定是由**散列函数**（hash function）执行的，它接收一份要存储到表中的数据，并生成一个数字，指定数据将存储在表中的哪个槽（slot）中。这个数字称为**散列键**（hash key）。例如，如果你有一个表，它具有 26 个可能的槽，设计一个允许将英语单词存放到表中的散列函数不会很困难。可以使用每个单词的第一个字母作为散列键，如下所示：

```
char *word_to_hash;
hash_key = tolower (*word_to_hash) - 'a';
```

这样，表中的槽将是 table［hash_key］。虽然这个散列函数很简单，但是它使用的表很小，以致于不能存放许多单词。在实际应用中，必须使用更大的表。

一个需要立即考虑的问题是：如何解决两个表项散列到表中同一个槽的问题。这种情况称为**冲突**（collision）。如果我们使用图 3-1 所示的简单散列表，单词“scotch”和“soda”，将散列到表中的同一个槽（table［18］）。应该如何解决这一冲突呢？可以使用许多方法。所有的冲突解决方法分为两类：**再散列法**（rehashing）和**拉链法**（chaining），采用前一种方法将计算出一个新的散列值，采用后一种方法将把冲突的元素通过一个链表添加到表中。在本章后面将讨论拉链法和再散列法。

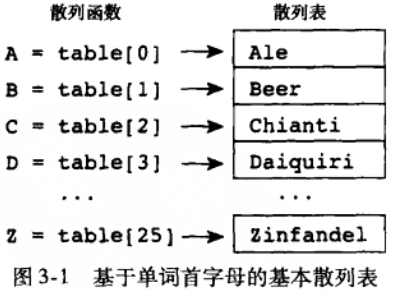


图 3-1　基于单词首字母的基本散列表

程序清单 3-1（birthday. c）显示了简单散列表的构造。程序读取一个文本文件，其中每一行包含一个生日以及对应的人员名字。当读取文件时，将名字存储在一个散列表中。当冲突发生时，程序将打印出在同一天出生的两个人的名字。

程序清单3-1　用于确定重复生日的简单散列表

```
/*--- birthday.c -------------------------- Listing 3-1 --------
 * Reads input file of birthdays and lists any duplicates.
 * Uses a simple hash table to identify the duplicates.
 *
 * Input records consists of lines of text of the form
 * MMDDName where MM = month, DD = day, Name. For example:
 *                  0212Abraham Lincoln
 * Note: for simplicity, no error checking is done on records.
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TABLE_SIZE 366  /* Maximum days in a year */

FILE *fin;              /* Birthday file */
int  NameCount,         /* Number of names read */
     DupeCount;         /* Number of dupes found */

int hash_birthday ( char * );   /* Our hash function */

#ifdef __STDC__         /* in ANSI C, there's no strdup() */
/* the source code for this function is discussed in Ch. 2 */
char *strdup ( const char * );  /* string duplication */
#endif

int main ( int argc, char *argv[] )
{
    char buffer[128];   /* where the records will be read */
    int  hash_value;    /* the hash value we will compute */
    char *name;         /* pointer to birthday name       */

    char *Table[TABLE_SIZE];   /* table of birthday folks */
    int i;              /* subscript to init the table    */

    if ( argc < 2 )
    {
        fprintf ( stderr, "Error! Expecting birthday file\n" );
        exit ( EXIT_FAILURE );
    }

    if (( fin = fopen ( argv[1], "rt" )) == NULL )
    {
        fprintf ( stderr, "Error! Cannot open %s\n", argv[1] );
        exit ( EXIT_FAILURE );
    }

    for ( i = 0; i < TABLE_SIZE; i++ )
          Table[i] = NULL;

    while ( ! feof ( fin ))
    {
        if ( fgets ( buffer, 128, fin ) == NULL )
```

```
            break;

        /* get rid of the '\n' at end of record */
        buffer [strlen ( buffer ) - 1] = '\0';

        NameCount += 1;

        hash_value = hash_birthday ( buffer );
        name = strdup ( buffer + 4 );

        if ( Table[hash_value] == NULL )  /* No duplicate, */
                                          /* so add name.  */
        {
            Table[hash_value] = name;
            continue;
        }
        else
        {                                 /* Is duplicate, */
            DupeCount += 1;               /* so tell 'em.  */
            printf ( "%s and %s have the same birthday.\n",
                     name, Table[hash_value] );
            continue;
        }
    }

    if ( DupeCount == 0 )
    {
        printf ( "No duplicate matches found among %d people.\n",
                 NameCount );
        if ( NameCount > 50 )
            printf ( "How rare!\n" );
    }
    else
        printf ( "Among %d people, %d matches were found\n",
                 NameCount, DupeCount );

    return ( EXIT_SUCCESS );
}

/*-------------------------------------------------------------
 * A simple hash algorithm. It converts the month and day
 * to the number of the day in the year. Adds the day in the
 * month to the number of days elapsed before that month began.
 *-------------------------------------------------------------*/

int hash_birthday ( char *data )
{
    const int days_elapsed[12] =
      { 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335 };
    int mm, dd;
    char record [128];  /* where we store our copy of the data */

    strcpy ( record, data );  /* make our own copy */

    record [4] = '\0';  /* Mark the spot where the name begins */
    dd = atoi ( record + 2 );   /* Convert the day to an int */
    record [2] = '\0';  /* Mark off the spot where day begins */
    mm = atoi ( record );       /* Convert the month to an int */
```

```
    /* since mm is going to be an index into days_elapsed,
     * it must be checked for the correct range.
     */

    if ( mm < 1 || mm > 12 )
    {
        fprintf ( stderr, "Error in record for %s: %02d/%02d\n",
                  (char *) data + 4, mm, dd );
        exit ( EXIT_FAILURE );
    }

    return ( days_elapsed[mm - 1] + dd );
}
```

在 main() 中，定义了名为 Table 的散列表，它由 TABLE_SIZE 个元素组成：一个元素对应于一年中可能的每一天。表的元素由指向字符的指针组成。最终，这些指针将指向那些过生日的人的名字。散列函数 hash_birthday 将把传入的月和日两个数字转换为一个小于 TABLE_SIZE 的唯一数字。它执行该任务的方式是：通过简单地将日期转换为儒略日期（Julian date）再减去 1，以适应 C 语言的基于 0 的数组。这个散列算法将 1 月 1 日转换为 0，将 2 月 1 日转换为 31，并将 12 月 31 日转换为 365。然后，该程序将检查位于 Table［hash_value］的指针。如果它是 NULL，也就是还没有谁在这一天过生日；因此将存储当前这个人的名字，并在 Table［hash_value］中插入一个指向此名字的指针。如果位于 Table［hash_value］的指针不是 NULL，就发生了冲突：两条记录被散列到表中的同一个元素上。在这种情况下，我们知道这意味着两个人的生日相同。我们打印出他们的名字，丢弃重复的记录，并继续读取下一条记录。最后，我们将汇总找到的重复记录数。

注意：这个问题非常适合于使用散列表，但是如果利用链表实现，其性能将会很差。包含所有生日的链表可能长达具有 366 个表项。对于这个长度，平均查找将需要 183 个查询。而使用散列表，即使把全部 366 个生日都加载到表中，使用一个查询就足以确定某个生日是否是重复的。

这种实现虽然好于链表，但是看起来几乎与管理一个指针数组没有多少差别。的确是这样，不过这仅仅是因为这里选择的散列表是一个简单的散列表。我们将看到，散列表很快就会变得更复杂。

3.2　散列函数

将生日转换为 0～365 之间的唯一数字的散列函数是一种特例。它被称为**完美散列函数**（perfect hash function）。它是完美的，因为对于传给它的每份数据，它都会生成唯一的散列值。在一年内没有任何两个日期具有相同的散列值。为了生成完美的散列值，必须知道散列函数的所有可能的输入，并且必须能够编写一个函数，它可以为每个输入生成唯一的值。就生日而言，我们知道日期的范围，以及如何为每个日期构造唯一的数字。我们可以编写一个非完美的散列函数，比如：

```
hash_value = dd * mm - 1;
```

它将产生 0～371 之间的键（这是大致的范围）。它是非完美的，因为它不会为每个唯一的输入生成唯一的值：3 月 10 日和 10 月 3 日都会生成散列值 29 。更糟糕的是，重复的值倾向于聚集在表的下部，而上部主要包含空槽（所有大于 31 的素数都未被使用，通过大于 12 的素数的乘积

计算得到的所有散列值也是如此)。如果选择一种非完美的算法，就意味着无法保证通过不超过一次的查找即可访问表元素。在可以设计完美散列算法的地方，它们的效率非常高。

不过，在现实情况中，几乎不可能构造完美的散列算法，花费精力寻找他们往往是白费力气。不可能编写出它们，因为通常几乎不可能提前知道要散列的完整数据集。例如，在我们马上将探讨的一个程序中，散列表用于统计文本中的重复单词。由于我们事先不知道哪些单词出现在文本中，就不能编写一个完美的算法。因此，我们将尝试编写一个良好的通用算法。

良好的散列函数具有两个令人称心如意的特点：一是快速的，并且它会把散列键均匀地分布在整个表中。二是它还必须弥补可能出现在输入数据中的聚集（聚集是具有近似元素值的数据的趋势)，对于相同的数据项，它必须总是产生相同的散列键。

你将不会搜寻完美的散列函数，而是需要设计一些散列函数，它们只用于把数据集的冲突频率降至最低。例如，让我们采用一个散列表，其中存储了取自英文文本中的单词。对于这样一个表（参见图3-1)，我们在散列函数上的初始尝试是：将每个单词的首字母用作散列键。虽然这个函数无疑是快速的，但它具有两个问题：它无法避免聚集并且它生成的唯一散列键太少。为了生成最大数量的散列键，散列函数通常会生成相当多的结果，然后用这个数量除以表的大小。

通常，散列函数具有如下形式：

$$hash\text{-}key = calculated\text{-}key \ \% \ tablesize$$

其中%是模运算符。为了提高散列键的离散程度，tablesize 应该是一个素数。生成 calculated-key 的过程是许多计算机科学论文的主题。不过，事实证明，应用程序用于计算散列键所花费的时间很少；缓慢的性能很少是由于拙劣的散列算法产生的。如果满足以下条件，散列函数一般将快速工作：

- 它们最多含有一个除法运算（一般是最终的模运算)。
- 它们生成广泛的散列键。
- 它们不依赖于将促使产生聚集的数据属性。

一种用于生成散列键的快速、通用的算法是由 AT&T 贝尔实验室的 Peter J. Weinberger 开发的(Weinberger 是 UNIX 模式匹配语言 AWK 中的“W”)，并发表在著名的“龙书”(dragon book) 中[Aho 等 1986]。程序清单3-2 中显示了一个基于 Allen Holub 改编的版本 [Holub，1990]。

程序清单3-2 一个通用的散列函数，根据 Peter Weinberger 的算法改编而成

```
/*--- HashPJW ---------------------------- Listing 3-2 ---------
 *    An adaptation of Peter Weinberger's (PJW) generic hashing
 *  algorithm based on Allen Holub's version.
 *
 *  Accepts a pointer to a datum to be hashed and returns an
 *  unsigned integer. This integer is called "calculated-key"
 *  in the text.
 *-------------------------------------------------------------*/

#include <limits.h>

#define BITS_IN_int  (sizeof (int) * CHAR_BIT)
#define THREE_QUARTERS      ((int) ((BITS_IN_int * 3) / 4))
```

```
#define ONE_EIGHTH          ((int) (BITS_IN_int / 8))
#define HIGH_BITS           ( ~((unsigned int)(~0) >> ONE_EIGHTH ))

unsigned int HashPJW ( const char * datum )
{
      unsigned int hash_value, i;

      for ( hash_value = 0; *datum; ++datum )
      {
            hash_value = ( hash_value << ONE_EIGHTH ) + *datum;
            if (( i = hash_value & HIGH_BITS ) != 0 )
                  hash_value =
                  ( hash_value ^ ( i >> THREE_QUARTERS )) &
                        ~HIGH_BITS;
      }

      return ( hash_value );
}
```

该算法之所以工作快速，是因为所有的工作都是通过位操作来完成的。出现在#define 语句中的除法运算是在编译时执行的，并且会把结果作为常量插入到代码中。该#define 语句只会测试整数中的位数（令人感到奇怪的是，这个常量不是由 ANSI 标准定义的，因为它将是一个有用的值），然后通过这个结果来获知要把最近生成的散列值移动多少位。以这种方式实现代码，以保证可移植性。对于 16 位的实现，与四分之三和八分之一对应的值分别为 12 和 2，而高位则是值为 0x3FFF 的掩码。

模运算符应该应用于从 HashPJW( ) 函数返回的值，并且结果（用返回值除以 tablesize 得到的余数）是一个散列值。与所有的实现一样，一种可能出现的情况是：像 HashPJW( ) 这样的通用算法可能不如被期待的数据集特意定制的算法那样有效。纯经验性工作是唯一的答案。因此，无论何时实现散列表，都应该编写一个函数，它将检查散列表，看看散列函数工作得有多好。然后可以借助这些统计数据对算法进行微调。

UNIX 使用了 Weinberger 的散列函数的一个不同的版本，用于在以 ELF 格式构造目标文件时执行散列（ELF 是在 1990 年在 UNIX SVR4 系统中引入的一种可执行和链接格式（Executable and Linking Format），它取代了 COFF（Common Object File Format，通用目标文件格式）。程序清单 3-3 中给出了由 UNIX Systems Laboratories（现在是 Novell 的一部分）在其官方 UNIX 文档中发布的函数。注意：这个版本期望至少 32 位的长整数。由于这个假定在所有的 MS-DOS 系统和大多数（如果不是全部）UNIX 版本上都是成立的，所以它可能不会破坏可移植性。不过，与程序清单 3-2 中所示的散列函数不同，这个函数返回一个无符号的长整型。本章末尾讨论了其性能结果。

程序清单 3-3　在 ELF 目标文件中使用的散列函数

```
/*--- ElfHash ---------------------------- Listing 3-3 ---------
 *  The published hash algorithm used in the UNIX ELF format
 *  for object files.
 *
 *  Accepts a pointer to a string to be hashed and returns an
 *  unsigned long. Algorithm is similar to that implemented
 *  in HashPJW (see Listing 3-2).
 *-------------------------------------------------------------*/
```

```
unsigned long ElfHash ( const unsigned char *name )
{
    unsigned long   h = 0, g;

    while ( *name )
    {
        h = ( h << 4 ) + *name++;
        if ( g = h & 0xF0000000 )
            h ^= g >> 24;

        h &= ~g;
    }
    return h;
}
```

程序清单3-3中的实现摘录于System V Application Binary Interface一书［UNIX Press，1990］。这个函数的最后一行有时会错误地印刷为h &= g（参见针对UnixWare的《Programming in Standard C》手册，1992）。那个遗漏的波浪号意味着这个不正确的函数将总是返回0——这几乎不是想要的散列函数！

除非你对程序清单3-2和3-3中所示的通用散列函数有特殊的要求，导致它们可能无法满足需要，否则应该可以使用这些函数。它们可以快速、合理地在表中分布键。此外，它们还具有相当好的可移植性。尽管在理论上人们崇尚完美的散列函数，但是花费时间用于优化散列函数几乎总是比花费时间用于增强程序其他某个部分的性能更好一些。利用性能测量和跟踪工具测量本章中所示程序的运行时间，总是显示性能首先主要受到I/O操作（包括极其缓慢的printf函数）的影响，然后受到频繁的分配（malloc）和释放（free）内存的影响。而仅仅从散列函数来看，它们通常只会消耗不到2%的程序时间。因此，优化的首要工作应着眼于I/O和内存管理函数。

3.3 冲突解决方法

除了可以设计出完美散列函数的情况之外，散列表必须具备合理地处理冲突的能力。当散列函数给两个不同的数据项而提供相同的散列值时，就会发生冲突。当两个不同的元素具有相同的散列值时，怎样把第二个元素添加到散列表中呢？对于这个问题广泛采用以下三种方法：（1）**线性再散列法**（linear rehashing），简单地按顺序遍历散列表，寻找下一个可用的槽；（2）**非线性再散列法**（nonlinear rehashing），计算出一个新的散列值；（3）**外部拉链法**（external chaining），将散列表中的每个槽视为具有相同散列值的数据项的链表的头部。散列表将发生冲突的项添加到该链表中。

3.3.1 线性再散列法

线性再散列法是形式最简单的冲突解决方法。刚一发现冲突，算法就会简单地遍历散列表，直至找到表中的下一个空槽，并将元素放入该槽中。此后，任何查找元素的操作都开始于散列值所指向的表槽。如果没有找到匹配，则将继续遍历散列表，直到：（1）找到相应的元素；（2）找到一个空槽（指示查找的元素不存在）；（3）检查整个散列表（指示该元素不存在并且散列表是满的）。

当遍历散列表时，不必一次检查一个槽。例如，在查找时，可以每隔两个槽检查一个槽。只要查找绕回到散列表的开始处，并且表中槽的总数不是3的倍数，则每个槽都会被检查。事实上，只要表的大小和检查槽的步长是互质的（它们没有公约数），那么表中的每个槽都会被检查。步长为1是不可取的，因为这将促使聚集产生。如果在表的同一个普通区域中发生了少数冲突，步长为1就意味着：表的这个部分将很快填满大量重复的内容，此时表的该区域中的任何活动都将涉及更大的步长。以5或更大的值作为步长，可以迅速地将查找从拥挤区域移开，并减小产生严重聚集的机会。

虽然线性再散列法很容易实现，但它具有两个缺点：不能从表中删除元素，并且当表被填满时性能下降明显。删除操作也是一个伤脑筋的问题，如果查找停止在第一个空槽上，则根本不能从表中执行删除操作。例如，假定有一个包含26个元素的散列表，我们首先将其用于保存单词，并基于它们的首字母对它们进行散列。我们读取单词“elephant”，并想将其插入到表中（参见图3-2）。我们转到用于存放“E”的槽，只会发现它已经被“Eel”占用了。我们检查下一个槽，并且发现它已经被“Elk”占用了。我们继续检查下一个槽，并且发现它是空的。因此，我们在该位置插入“Elephant”。如果下一个操作是删除“Elk”，那么，查找“Elephant”将永远也不会成功，因为在找到它之前，将先遇到通过删除“Elk”而空出的槽。因此，只有通过把使用过的槽标记为无效的才可以执行删除。这样，如果在查找“Elephant”时遇到无效的槽，就可以简单地继续执行查找。

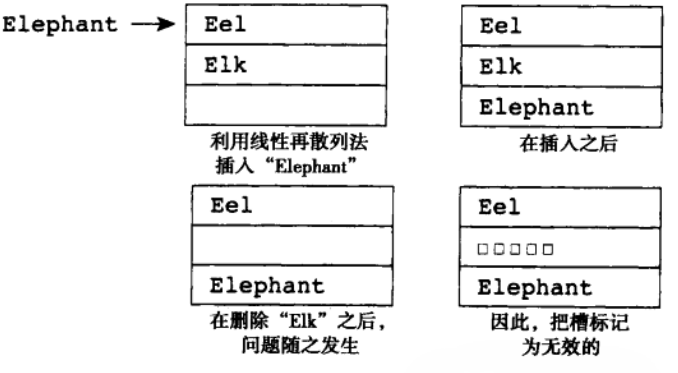


图3-2　线性再散列法及其风险

3.3.2　非线性再散列法

我们刚才探讨的方法称为线性再散列法，这是因为它从冲突位置开始，并且以顺序方式遍历散列表，来查找一个可用的槽。非线性再散列法可以避免遍历散列表。相反，它会计算一个新的散列键，并通过它跳转到表中一个完全不同的部分。其思想是：希望通过跳转到表中不同的部分，可以避免相似值的聚集（参见图3-2）。如果再散列函数跳转到的槽已经被占用了，则继续执行新一轮的再散列和跳转。如果散列表快填满了，那么在找到一个可用的槽之前，可能需要对同一个元素执行多次的非线性再散列。为了避免这个问题，仅当散列表中使用的空间不太可能超过50%时，才应该使用再散列算法。

在我们前面讨论散列函数时说明：计算散列值的一种实用方法是先计算出一个数字，然后用它除以表中槽的个数。这个除法运算的余数（模）就是散列值。另一种再散列的方法是利用除法运算产生的商，而不是余数。然后可以用这个商乘以输入字符串的长度以及第二个字符（假定它不为空）所对应的数值。也可以研究另外一些类型的乘法或位移运算。最终，将生成一个新的数字，它类似于用原始值除以槽的个数。

另一种可能稍微有点慢但非常有效的方法是使用C语言的随机数生成器。rand()函数通过一个种子生成伪随机数，并使用返回的随机数作为新的散列值。

在采用这种方法之前，需要了解rand()函数的工作方式。rand的工作机制是：首先以一个值作为种子，然后重复调用rand()函数，利用每次调用，在伪随机序列中生成下一个值。例如，下面的代码段通过用100这个值调用srand()函数，给rand机制提供种子。然后，它打印出通过5次连续调用rand()生成的值。在利用Microsoft的C编译器编译时，该程序的运行结果为：365、1 216、5 415、16 704和24 504。

```
#include <stdio.h>
#include <stdlib.h>

int main ( void )
{
    int i;

    srand ( 100u );

    for ( i = 0; i < 5; i++ )
    {
        printf ( "%6d\n", rand() );
    }

    return ( i );
}
```

注意：无论何时这个编译器的rand机制以100作为种子，总会生成这组相同的随机数。这一点很重要，它意味着rand机制可以用于再散列，因为当你查找一个元素时，散列函数将提供相同的随机散列值，它用于把该元素放在散列表中。不过，由于这一点，不应该把发生冲突的散列值用作rand()函数的种子。可以使用要散列的数据的其他某个方面，而不要使用发生冲突的散列值。

一个对散列表性能影响很大的概念称为**负载因子**（load factor）。负载因子（α）是指用插入到表中的元素个数（n）除以可用槽的总数所得到的结果。因此：

$$\alpha = n/\text{tablesize}$$

负载因子为1.00意味着插入到表中的元素个数与表中槽的个数一样多。当负载因子增大时，散列表的性能也会随之下降。在设计任何散列表时的一个重要的考虑事项是：如何处理较大的负载因子。

无论是使用线性再散列法还是非线性再散列法，只有在散列表不会接近填满的情况下，才能使用再散列。当散列表的负载因子增大时，再散列所花费的时间也会显著增加。在Kruse等人的著作［Kruse 1991］中提供了在散列表中查找元素所需探查（probe）次数的理论分析。表3-1源于该分析，该表清楚说明：当负载因子大于0.50时，再散列将不是一种切实可行的解决方案。当

表中的查找很可能不成功时尤其如此。随机再散列的最佳方案（5.0 次探查用于 0.80 的负载因子）将慢得令人无法接受。如果实现使用以前描述的 rand( ) 函数来执行再散列，每次查找都必须给 rand( ) 函数提供一个种子，然后平均要调用该函数 5 次。与执行任何其他方面的散列表操作相比，将花费更多的时间来计算随机再散列值。

表 3-1　再散列所需的理论探查次数

负载因子 >	0.10	0.50	0.80	0.90	0.99
成功的线性再散列	1.06	1.5	3.0	5.5	50.5
成功的非线性再散列	1.05	1.4	2.0	2.6	4.6
不成功的线性再散列	1.12	2.5	13.0	50.0	5 000
不成功的非线性再散列	1.10	2.0	5.0	10.0	100

只应该在快速而又随性的情况下或者在快速原型化的环境中使用再散列方法。它的主要优点是：很容易进行动态编码，在确保表负载较低并且不太可能执行删除操作的情况下它的速度足够快。对于所有其他的需求，应该使用外部拉链法。

3.3.3　外部拉链法

外部拉链法的技术是将散列表看作是一个链表数组。表中的每个槽要么为空，要么指向散列到该槽的表项的链表。可以通过把元素添加到链表中来解决冲突。同样，可以通过从链表中删除元素来执行删除操作。因此，解决冲突的代价不会超过向链表中添加一个节点；无需执行再散列。在再散列中，表项的最大数量是由表中槽的原始数量确定的，与之不同的是，外部拉链法可以容纳的元素与将在内存中存放的元素一样多。

使用我们检查过的第一个散列表，其中的单词是基于它们的首字母进行散列的，使用外部拉链法的散列表将如图 3-3 所示。假设一个输入文件包含 Eel、Elephant、Dog、Cat、Giraffe、Elk、Zebra、Goat 和 Donkey。

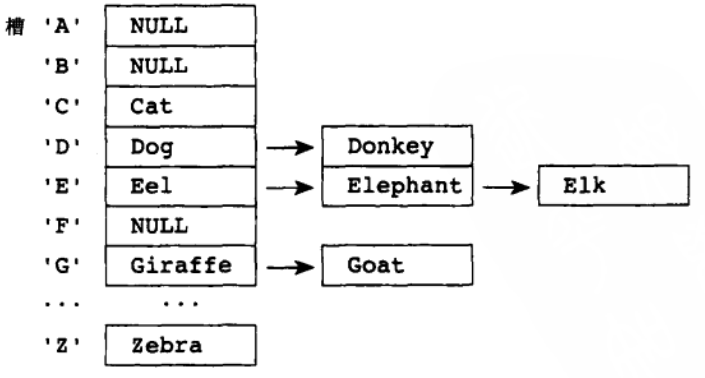


图 3-3　使用外部拉链法的散列表

你将注意到链表中的元素是按字母顺序放置的。因为不需要遍历整个链表以确定链表中是否

存在某个元素，这将加快查找的速度。因此，平均查找时间是链表长度的一半再加上1，这个“1”用于最初检查表槽。同样，无论查找是成功还是失败，平均探查次数都将与链表平均长度的一半成正比，其中可以将空链表的长度看作1。这种计算表明：在平均探查次数超过2.0之前，结合使用外部拉链法与有序链表的散列表必须有一个大于1.0的负载因子。同样有趣的是以下事实：失败的查找次数趋向于少于成功的查找次数。外部拉链法的缺点是：它需要稍微多一些的空间来实现，因为添加任何元素都需要添加指向节点的指针，并且每次探查也要花稍微多一点的时间，因为它需要间接引用指针，而不是直接访问元素。由于今天的内存成本很低并且可以使用非常快的CPU，这些缺点都是微不足道的。因此，大多数专业开发人员都使用拉链法来解决散列表冲突。

程序清单3-4（wordlist.c）接受一个输入文本文件，并统计重复单词的个数。表中元素包含一个指针和一个整数，前者指向一个单词，后者记录该单词在文本文件中出现的频率。依赖于在文本文件开始处设置的选项开关，可以把单词以及它们各自的计数打印到屏幕上。就像对所有极大地依赖于散列表的程序所应该做的那样，能够根据需要打印出与散列算法和散列表的性能有关的所有统计数据。这是监视数据的任何不同寻常的方面是否损害了散列算法的效率的最佳方式。程序清单3-4极大地依赖于第2章中介绍的链表函数。

程序清单3-4　一个用于统计文本文件中单词出现次数的程序，它使用散列表和外部拉链法来解决冲突

```
/*--- wordlist.c -------------------------- Listing 3-4 -------
 *  Lists all words in a text file by storing them in a hash
 *  table. Must be linked to the linked-list primitives of
 *  Chapter 2 and to ElfHash() of Listing 3-3.
 *-------------------------------------------------------------*

   /* uncomment the following line to print all words
    * with their associated hash value.
    */

/* #define LIST_HASH  1 */

   /* uncomment the following line to print the unique words
    * and a count of their frequency.
    */

/* #define LIST_WORDS 1 */

   /* comment out the following line if you do not want
    * an analysis of the hash function and hash table load.
    */

#define LIST_TABLE_STATS 1
   /* uncomment the following line if you want a listing
    * of all words > 10 letters. Such words are often typing
    * or scanning errors in the text or odd constructs.
    */

#define LIST_LONG_WORDS 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <ctype.h>

#include "llgen.h"          /* Header for generic linked lists */
#include "llapp.h"          /* Header for appl.'s linked lists */

extern unsigned long ElfHash ( char *);

/*--- the hash table portion ---*/

#define TABLE_SIZE  1999    /* Number of slots; a prime number */

Link   *Table;              /* Our table is an array of Links */

#if 0
int CreateTable ( Link **t )
{
    *t = calloc ( TABLE_SIZE, sizeof ( Link ));
    return ( *t == NULL ? 0 : 1 );
}
#endif

/*-------------------------------------------------------------
 * We use calloc() to allocate the table. However, on some
 * compilers the bit pattern used by calloc() for initialization
 * is not the same as NULL (which is what we want). So we check
 * for this condition and, if it occurs, intialize the table by
 * hand to NULLs. If this is not an issue, use the routine above.
 *-------------------------------------------------------------*/
int CreateTable ( Link **t )
{
    int i;

    *t = (Link *) calloc ( TABLE_SIZE, sizeof ( Link ));
    if ( *t == NULL )
        return ( 0 );

    if ( **t != NULL )      /* is the calloc'd memory == NULL? */
    {
        for ( i = 0; i < TABLE_SIZE; i++, t++ )
            **t = NULL;
    }

    return ( 1 );
}


/*-------------------------------------------------------------
 * Function to convert a string to upper case. Exists in most
 * PC C libraries but missing from many UNIX C libraries.
 *-------------------------------------------------------------*/
char *strupr ( char *str )
{
    char *s = str;

    while ( *s )
    {
        *s = toupper ( *s );
```

```
        s += 1;
    }

    return ( str );
}

/* === main line === */

int main ( int argc, char *argv[] )
{
    char     word[64];        /* the raw word from the file */
    char     *pw;             /* pointer to the word */
    int      c, i, j;
    int      chains,          /* counts how many chains     */
             chain_table[33]; /* table of chain lengths     */
                              /*    for our report.         */
    int      add_status;      /* return value from table add*/

    unsigned hash_key;

    struct List *L1,          /* list for hash table entries*/
                *long_wd;     /* list of long words         */

    struct NodeData1 nd;      /* the node we add each time  */
    struct Node      n;       /* used for scratch purposes  */

    FILE     *fin;            /* the input file             */

    if ( argc < 2 )
{
    fprintf ( stderr, "Error! Usage: wordlist filename\n" );
    exit ( EXIT_FAILURE );
}

if ( argc > 2 )
    fprintf
        ( stderr, "Warning: Usage: wordlist filename\n" );

fin = fopen ( argv[1], "rt" );
if ( fin == NULL )
{
    fprintf ( stderr, "Could not find/open %s\n", argv[1] );
    exit ( EXIT_FAILURE );
}

/*--- create the table ---*/

if ( ! CreateTable ( &Table ))
{
    fprintf ( stderr, "Error! Could not create table\n" );
    exit ( EXIT_FAILURE );
}

/*--- set up linked-list data structures ---*/

L1 = CreateLList ( CreateData1,      /* in llapp.c */
                   DeleteData1,      /*     "      */
```

```
                    DuplicatedNode1,  /*      "       */
                    NodeDataCmp1 );   /*      "       */

long_wd = CreateLLList ( CreateData2, /* in llapp.c */
                    DeleteData2,      /*      "       */
                    DuplicatedNode2,  /*      "       */
                    NodeDataCmp2 );   /*      "       */

if ( L1 == NULL || long_wd == NULL )
{
    fprintf ( stderr, "Error creating linked list\n" );
    exit ( EXIT_FAILURE );
}

/*--- begin processing file ---*/

c = ' ';

while ( ! feof ( fin ))
{
        /*--- skip white space ---*/

        while ( c != EOF && isspace ( c ))
            c = fgetc ( fin );

        /*--- pick up the word ---*/
        i = 0;
        while ( c!= EOF && !isspace ( c ))
        {
            word[i++] = c;
            c = fgetc ( fin );
        }

        if  ( c == EOF )
            break;

        word[i] = '\0';

        /*--- strip off trailing punctuation ---*/

        while ( i >= 0 && ispunct ( word[--i] ))
            word[i] = '\0';

        pw = strupr ( word );

        /*--- get the hash value ---*/

        hash_key = (unsigned int) ElfHash ( pw );
        hash_key %= TABLE_SIZE;

#ifdef LIST_HASH
        printf ( "%15s  %3d\n", pw, hash_key );
#endif

        /*--- insert into table ---*/

        L1->LHead = Table[hash_key];
```

```
        nd.word =  pw;              /* the string we're adding */
        nd.u    =  1;               /* adding one occurrence   */

        add_status =  AddNodeAscend ( L1, &nd );
        if ( add_status == 0 )      /* an error occurred */
            printf ( "Warning! Error while allocating node.\n" );

        Table[hash_key] = L1->LHead;

        /*--- handle long words ---*/
        /* if a word is longer than ten chars, put it in the
         * long word list for subsequent display.
         */

        if ( strlen ( pw ) > 10 )
            AddNodeAscend ( long_wd, &nd );

        /* if a word is longer than 20 chars, it's likely to
         * be a typo or other error; so delete it. This pro-
         * cessing is included primarily to exercise these
         * functions. It should be removed for real text
         * processing.
         */

        if ( strlen ( pw ) > 20 )
        {
            Link pl;

            pl =  FindNodeAscend ( L1, &nd );
            if ( pl == NULL )
                printf ( "processing error!\n" );
            else
                DeleteNode ( L1, pl );
        }
    }

    /*--- now dump the table ---*/

    for ( j = 0; j < 33; j++ )
          chain_table[j] = 0;

    for ( i = 0; i < TABLE_SIZE; i++ )
    {
        Link pcurr;     /* Node we're examining */

        pcurr = Table[i];           /* set to start of list */
        if ( pcurr == NULL )        /* skip empty slots */
            continue;
        else
        {
            int chain_len;

            for ( chain_len = 0; ; pcurr = pcurr->next )
            {
                memcpy ( &n, pcurr, sizeof ( struct Node ));

#ifdef LIST_WORDS
```

```
                /* Print each word and the count of occurrences */
                  printf ( "%-20s  %3u\n",
                           ( (pND1) n.pdata)->word,
                           ( (pND1) n.pdata)->u );
#endif
                  chain_len += 1;

                  if ( pcurr->next == NULL )
                      break;
              }

              if ( chain_len > 32 )
                   chain_len = 32;

              chain_table[chain_len] += 1;
          }
      }

#ifdef LIST_LONG_WORDS
      if ( long_wd->LCount < 1 )
          printf ( "No long words!\n" );
      else
      {                        /* step thru the list and print it*/
          Link pcurr;

          for ( pcurr =  long_wd->LHead;
                pcurr != NULL;
                pcurr =  GotoNext ( long_wd, pcurr))
                  printf ( "%-20s\n",
                           ((pND2)( pcurr->pdata ))->word );
      }
#endif

#ifdef LIST_TABLE_STATS

      chains = 0;
      for ( i = 32; i > 0; i-- )
      {
          if ( chain_table[i] == 0 )
              continue;
          else
          {
              printf ( "%3d chains of length %2d\n",
                       chain_table[i], i );
              chains += chain_table[i];
          }
      }
      if ( chains != 0 )
      {
          printf ( "\n%d Nodes in %u chains\n\n",
                   L1->LCount, chains );

          printf ( "Size of hash table   = %u\n",
                     (unsigned) TABLE_SIZE );

          printf ( "Average chain length = %f\n",
                    L1->LCount / (double)chains );
```

```
        printf ( "Slot Occupancy       = %f \n",
                 ( (double)chains ) / TABLE_SIZE );

        printf ( "Load Factor          = %f \n",
                 ( (double)L1->LCount ) / TABLE_SIZE );
    }
    else
        printf ( "Error! No chains found.\n" );
#endif

    return ( EXIT_SUCCESS );
}
```

有一个文本文件中包含 Arthur Conan Doyle 于 1904 年发表的经典著作“The Adventure of the Abbey Grange”，当对这个文本文件运行 wordlist. c 时，程序将会读取 9 286 个单词，其中 1 956 个单词被确定是只出现一次的。在这 1 956 个单词中，有 1 335 个单词被放在 2 个或更少表项的链中。图 3-4 中显示了由 LIST_TABLE_STATS 选项打印的统计数据。这些统计数据告诉我们相当多的关于散列过程的质量的信息。要指出的第一件事是：图 3-4 中所示的**槽占用率**（slot occupancy）。以前，把负载因子定义为表中的表项数量除以可用槽数的结果。就使用外部拉链法的表而言，这些统计数据用处不大，因为大于 0 的负载因子很常见，不值得特别关注。槽占用率（我们杜撰的术语）指示散列表中实际用于加载链表的槽数，它度量的是散列算法的质量。在这个示例中，有 59% 的槽指向链表。令人感兴趣的是各条链的平均长度。在每条链中的 1. 65 个节点处，成功的查找平均需要 1. 62 次探查（参见下一个段落中的解释）。在使用再散列的情况下，表将具有 0. 98 的负载因子，这意味着最佳的查找将需要 5. 2 次或 126 次探查，这取决于查找成功与否。此时，外部拉链法的好处是显而易见的。

```
  1 chains of length 10
  1 chains of length  9
  1 chains of length  7
  4 chains of length  6
 16 chains of length  5
 35 chains of length  4
117 chains of length  3
330 chains of length  2
675 chains of length  1

1956 Nodes in 1180 chains

Size of hash table   = 1999
Average chain length = 1.657627
Slot occupancy       = 0.590295
Load Factor          = 0.978489
```

图 3-4 对“The Adventure of the Abbey Grange”运行 wordlist. c 的结果

我们怎样才能知道成功的查找平均需要 1. 62 次探查呢？假设我们把表中的每个元素都查找一次。对于有 10 个项目的链，对第一个项目将需要 1 次探查，对第二个项目将需要 2 次探查，对第三个项目将需要 3 次探查，依此类推，直至它对链表中的最后一个元素执行了 10 次探查。具有 10 个节点的链将需要 $10+9+8+\cdots+1=55$ 次查找，才能成功地找到它的所有元素。形如 $n+n-1+$

n－2＋…＋n－n 的数字俗称三角数（triangular number）。用于查找三角数的方程是：x＝（n^2＋n）/2。在前一个示例中，n＝10，x（总探查次数）＝55。表 3-2 给出了查找表中的所有元素所需的总探查次数。

表 3-2　怎样计算探查次数

次　数	链 长 度	三 角 数	乘　积
1	10	55	55
1	9	45	45
1	7	28	28
4	6	21	84
16	5	15	240
35	4	10	350
117	3	6	702
330	2	3	990
675	1	1	675
		总探查次数	3 169
		元素总数	1 956
		平均探查次数	1.620143

你将注意到这些统计数据显示了访问表中的每个元素所需的探查次数。如果链表是有序的，平均查找将只需遍历一半链表，即可找到元素或者确定它不在表中。因此，对于成功的查找，将显著改善表 3-2 中所示的平均探查次数。

返回到程序上来，简单地读入文件，基于空白来区分单词，去除任何尾部的标点符号，并把单词转换为大写字母。然后，它计算一个散列值（即 hash_key），并利用第 2 章中介绍的 AddNodeAscend( ) 函数尝试把单词和计数 1 添加到散列表中。该函数按升序向链表中添加节点。如果出现重复的表项（意味着单词已经在表中），就会把与已经在表中的单词关联的计数递增 1。

程序清单 3-4 使用第 2 章中介绍的通用链表函数。如第 2 章中所讨论的，通用链表函数的任何使用都需要特定于应用程序的代码。该代码保存在文件 llapp.c 中，对于这个程序，程序清单 3-5显示它与第 2 章中的 llapp.c 相比变化相当小。

程序清单 3-5　针对程序清单 3-4 的链表操作的特定于应用程序的部分

```
/*--- llapp.c ---------------------------- Listing 3-5 ---------
 *  Application-specific functions for linked-lists.
 *  Here used for hash table of words and word counts.
 *-------------------------------------------------------------*/

#include <stdlib.h>          /* for free() */
#include <string.h>          /* for strcmp() and strdup() */

#include "llgen.h"
#include "llapp.h"
```

```
void * CreateData1 ( void * data )
{
    struct NodeData1 * new_data;

    /*--- allocate the data structure ---*/

    new_data = (pND1) malloc ( sizeof ( struct NodeData1 ));

    if ( new_data == NULL )
        return ( NULL );

    /*--- move the values into the data structure ---*/
    new_data->u    =  1;
    new_data->word =  strdup ( (char *) ((pND1) data)->word );

    if ( new_data->word == NULL )   /* error copying string */
    {
        free ( new_data );
        return ( NULL );
    }
    else
        return ( new_data );
}

int DeleteData1 ( void * data )
{
    /*
     * In this case, NodeData1 consists of: a pointer and an int.
     * The integer will be returned to memory when the node
     * is freed. However, the string must be freed manually.
     */

     free ( ((pND1) data)->word );
     return ( 1 );
}
/*-------------------------------------------------------------
 * This function determines what to do when inserting a node
 * into a list if an existing node with the same data is found
 * in the list. In this case, since we are counting words, if a
 * duplicate word is found, we simply increment the counter.
 *
 * Note this function should return one of the following values:
 *      0       an error occurred
 *      1       delete the duplicate node
 *      2       insert the duplicate node
 * Any other processing on the duplicate should be done in this
 * function.
 *-------------------------------------------------------------*/

int DuplicatedNode1 ( Link new_node, Link list_node )
{           /* adding an occurrence to an existing word */

    pND1 pnd = list_node->pdata;
    pnd->u += 1;
    return ( 1 );
}
```

```
int NodeDataCmp1 ( void *first, void *second )
{
    return ( strcmp ( ((pND1) first)->word,
                      ((pND1) second)->word ));
}

/*=== Now the functions for the second linked list ===*/

void * CreateData2 ( void * data )
{
    struct NodeData2 * new_data;

    /*--- allocate the data structure ---*/

    new_data = (pND2) malloc ( sizeof ( struct NodeData2 ));

    if ( new_data == NULL )
        return ( NULL );

    /*--- move the values into the data structure ---*/
    new_data->word =  strdup ( (char *) ((pND1) data)->word );

    if ( new_data->word == NULL )   /* error copying string */
    {
        free ( new_data );
        return ( NULL );
    }
    else
        return ( new_data );
}

int DeleteData2 ( void * data )
{
    /*
     * In this case, NodeData2 consists of a pointer.
     * The string must be freed manually.
     */

     free ( ((pND2) data)->word );
     return ( 1 );
}

/* do nothing on a duplicated nodes */
int DuplicatedNode2 ( Link new_node, Link list_node )
{
    return ( 1 );
}

int NodeDataCmp2 ( void *first, void *second )
{
    return ( strcmp ( ((pND2) first)->word,
                      ((pND2) second)->word ));
}
```

其中的变化包括：在 DuplicatedNode2( ) 函数中以不同的方式处理重复的表项。所有其他的链表函数都是由通用函数提供的。

一旦读取并处理了输入文本文件，程序就会创建并初始化一个小表（chain_table），并且载入达到了给定长度的链的计数。如果链表有6个节点，那么就把chain_table［6］增加1。然后，程序读取散列表，通过遍历每个链表确定所有链表的长度，并在到达链表末尾之后更新chain_table。一旦完成了这些任务，就会打印关于散列表的统计数据，并且程序会退出。

之前，我们提议对链表进行排序，以便使得查找更快速。不过，按升序或降序对节点进行排序并不总是最佳的解决方案。在某些情况下，在链表头部插入新节点更好一些。这将创建一种后进先出的顺序，类似于在第2章中讨论的栈。在分析C语言程序的情况下，这种方法很有用。编译器总是使用散列表存储程序中的标识符。在定义标识符时，就把它们存储在表中。当它们越界时，就把它们从表中删除。假设我们检查下面的程序段，它在多个位置定义了变量*i*：

```
#include <stdio.h>
int i;                  /* global i */
main ( void )
{
    int i;              /* main i */
    ...
    if ( a > 2 )
    {
            int i;      /* block-level i */
            ...
    }
}
```

一旦我们进入if块中，就只能访问在块级别定义的*i*，而不能访问其他的*i*。因此，所有指向*i*的引用都必须引用它，并且不能把指向main( ) 中的*i*或者全局级别的*i*的引用弄混淆。因为*i*的所有实例都将散列到表中的同一个槽中，所以我们要小心处理如何将它们添加到表中。事实上，它们是三个具有相同名称的不同变量。如果仅仅只是使用一个有序链表，我们将被迫做大量的工作来确定正在访问表中的哪个*i*。一种更巧妙的解决方案是：在定义这些*i*时把它们添加到链表头部。这将创建如图3-5所示的链表。

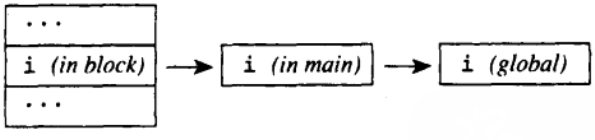


图3-5 在散列表中存储*i*的多个定义

现在，对变量*i*的任何访问都将首先查找局部块定义，从而访问*i*的正确实例。当块越界时，将从局部块中删除第一个*i*，并使所有其他指向*i*的引用指向在main( ) 顶部声明的版本。这是一种巧妙的解决方案，并且是由AddNodeAtHead( ) 函数在通用链表例程中提供的解决方案。

另一种方法是：当访问某个节点时，把它移到链表头部。这种方法利用了一些类型的数据存在着聚集在某些值周围的趋势。在程序设计中有一个良好的示例，其中只在使用局部变量之前才定义它们。当扫描一个程序并在散列表中查找标识符时，很可能将在这个区域重复引用最近访问的变量；因此把它移到链表的顶部可以缩短后续的查找时间。使用这种模型时，将把最近最少使

用的数据项移到链表尾部。

3.4 性能问题

可以通过遵循广泛的指导原则以及通过监视结果，在散列表中获得良好的性能。每个散列表都具有独特的方面，并且每个散列表都被要求处理独特的数据。如果你尽可能多地了解散列操作的工作方式，将会知道如何快速获得较好的结果。结合你自己的需要和你自己的结果检查下面的建议：

- 创建大小合理的散列表。不过，对于负载因子在 0.20 以下的散列表，如果扩展它们，将不会提供更好的性能。
- 确保每个散列表中的槽数是一个素数。
- 在有代表性的数据上测试散列算法，并且度量结果。如果不能使用有代表性的数据，就一定要测试极限数据范围。限制散列函数并只执行一个除法运算——最好是对表大小执行取模运算。
- 预先考虑冲突，并在可能的地方使用外部拉链法。

除了第一点之外，我们将详细研究其他几点。

确保每个散列表中的槽数是一个素数。常识告诉我们：当除以一个素数时，会产生最分散的余数。实践证明了这种假设。可能最糟糕的除法是除以 2 的倍数，因为这只会屏蔽被除数中的位。由于我们使用表大小对散列数字（散列函数的结果）进行模运算，如果表中的槽数是一个素数，就可以获得最佳的结果。表 3-3 列出了典型的表很可能需要的素数范围。左边的数字是你可能想要的散列表的基本大小；右边的数字是小于想要的表大小的最大素数。

表 3-3　用于散列表的素数

想要的表大小	最接近的素数	想要的表大小	最接近的素数
100	97	1500	1 499
250	241	2000	1 999
400	397	4000	3 989
500	499	5000	4 999
750	743	7500	7 499
1000	997	10000	9 973

测试散列算法。程序清单 3-4 中展示的 wordlist. c 程序说明了如何度量散列算法的性能。它说明了链的平均长度和表的利用率。它还会打印所有链尺寸的列表。如果链很长，就指示有问题。如果有许多长链，表就太小，如果只有几条链很长，数据的某个方面就会影响散列函数的结果。程序清单 3-2 和 3-3 给出了两个通用的散列函数。表 3-4 显示了当 wordlist. c 程序对 Arthur Conan Doyle 所著的 The Adventure of the Abbey Grange 一书中的文本使用这些散列函数时所得到的结果。这篇小说包含 9 176 个单词，其中有 1 901 个单词是唯一的。

表3-4 当读取 The Adventure of the Abbey Grange 的内容时两个散列算法的性能

	HashPJW( )	ElfHash( )
平均链长度	1.56	1.64
槽占用率	0.61	0.58
长度≤4 的链数	1191	1098
执行时间	1.5 秒	1.5 秒

依据这些结果，可以看到 HashPJW( ) 函数比 ElfHash( ) 函数对文本做了稍微好一点的工作。当散列数字时，ElfHash( ) 函数将比 HashPJW( ) 函数稍微好一些。表3-4 中列出的结果适用于两个散列函数的16 位的实现。算法的32 位的实现完全相同（由于在 HashPJW 中使用了预处理器别名）。

完美的散列算法将把单词分布进1 999 个槽中的1 901 个槽中，这样槽的占用率就是95%。虽然这些算法离这个目标稍微有点远，它们仍然会维持平均链长度小于2。在这个级别上，将在链表的第一个或第二个节点中找到大多数单词。这大致相当于通过随机存取而获得的好处。更传统的随机存取技术（比如二叉树或折半查找）就不能利用与这里介绍的散列算法和技术一样少的开销来检索数据。因此，散列表具有持久的吸引力。

最后，散列算法的相对质量不会影响程序的性能，注意到这一点是有趣的。散列算法的优化虽然是必要的，但是应该发生在执行了其他优化之后。告诉硬件狂热者们一个信息，运行时间是在使用 Intel 486 芯片且运行速率为 66 MHz 的 PC 上获得的。在编译程序时没有对其进行优化。大部分执行时间都花在读取文件上。

预先考虑冲突。如果使用散列表通过再散列解决冲突，就不要假定散列表大得足以避免冲突。这种观点在几乎所有的情况下都必定是错误的。考虑著名的关于生日的智力游戏。可以证明：如果随机选择 24 个人，其中两个人具有相同生日的几率超过 50%。如果随机选择 50 个人，这个几率接近于 100%。同样，在有 365 个元素的表中，在只散列了 24 个值之后，标准的散列函数有超过一半的时间会产生冲突。如果不相信这一点，可以在你的下一次大型集会上试试。程序清单3-1 中的程序（birthday. c）将使用这个过程特别轻松。

3.5 资源和参考资料

Aho、Alfred、Setli Ravi 和 Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986。

Holub, Allen. *Compiler Design in C.* Englewood Cliffs, NJ: Prentice Hall, 1990. 关于编译器实现的可用的最佳书籍。从第 482 页开始讨论了散列。

Kruse、Robert、Bruce Leung 和 Clovis Tondo. *Data Structures and Program Design in C.* Englewood Cliffs, NJ: Prentice Hall, 1991. 一本介绍数据结构的良好的教材。

UNIX Press. *System V Application Binary Interface—UNIX System.* Englewood Cliffs, NJ: UNIX Press, Prentice Hall, 1990。

第4章 查 找

比较两个字符串以及在一个字符串中查找另一个字符串是密切相关的基本编程操作。字符串比较操作要回答“两个给定的字符串相似吗，或者它们相同吗?”这样的问题。ANSI C 中的 strcmp( ) 系列函数提供了这种操作。相比而言，字符串查找操作要从头至尾查看一个大文本块，找出与目标字符串匹配的任何子串。为了解决这个问题，ANSI C 提供了 strstr( ) 函数。这些 ANSI 函数都是相关的，这是由于从本质上讲，任何查找操作都必须反复比较查找文本的多个不同的子串。不过，这些 ANSI C 函数只提供了最基本的查找和比较操作。

本章介绍了 ANSI C 的字符串查找和比较函数的多种替代方法（参见表 4-1）。这些算法在两个方面有别于 ANSI C 的字符串函数。一组算法通过利用字符串属性（Boyer-Moore 算法）或者通过同时查找多个目标字符串（Aho-Corasick 算法以及正则表达式查找）来减少查找时间。另一组算法为多个目标字符串通过近似查找（Aho-Corasick 算法，也就是正则表达式查询）来减少查询时间。第二组算法则是寻找近似的匹配。可以利用模板（grep）、通过距离指标或者甚至通过两个单词的发音（Soundex 和 Metaphone）来指定近似匹配的类型。

表 4-1 不同查找技术及其对应算法的比较

字符串查找和比较	方 法
查找一个字符串的精确副本*	蛮力算法；Boyer-Moore 算法
同时查找两个或更多字符串的精确副本*	Aho-Corasick 算法；正则表达式查找
查找给定字符串的近似匹配	正则表达式查找；字符串差别度量
语音比较	Soundex；Metaphone

* 可轻松地修改成不区分大小写。

4.1 查找的特征

所有的查找技术都具有共同的目标，但是对于给定的情况，一种技术使用的方法会影响它的适用性。虽然查找的类型可能会限制你的选择，但是在选择一种技术时需要考虑三个主要特性（另请参见表 4-2）。

表 4-2 精确字符串查找方法的比较。其中 A 是文本使用的字母表中的字符个数，TLen 是要查找的文本的长度，PatLen 是查找模式的长度

	蛮力算法	Aho-Corasick 算法	Boyer-Moore 算法
准备时间	无	O（所有目标模式的长度之和）	O（PatLen + A）
最坏情况	O（PatLen * TLen）	O（TLen）	O（TLen）
典型情况	O（TLen）	O（TLen）	O（TLen/PatLen）
是否进行回溯	是	无	是

4.1.1 准备时间

一些比较技术在可以执行查找之前要求先进行大量的计算。如果这要花费许多时间，而要查询的文本却很少，那么准备时间可能会超过通过使用具有更高理论速度的算法所节省的时间。

4.1.2 运行时间

所有的字符串查找算法都是根据 a + bn 执行的。其中 n 是要查找的文本中的字符数；b 是一个常数，表示每个字符 n 执行的比较次数，a 是开始查找所需的准备时间。本章中讨论的算法效率是由它们能在多大程度上降低 b 的值决定的。良好的查找对于查找的每个字符，只需执行少于一次的比较；也就是说，$b<1$（这种查找称为**次线性**（sublinear）查找），而效率低下的查找可能具有 $b>1.5$。除此之外，还要加上任何准备时间（一般仅当查找多个字符串时才需要）以及要查找的元素。读者习惯于标准的算法表示法，查找算法的阶全都是 $O(N)$，其中 N 是要查找的文本中的字符数。

4.1.3 回溯的需要

一些算法以线性方式查找文本，而另外一些算法则在文本中来回移动。如果查找文本以数据块的形式存在于内存中，来回移动并不困难，但是如果查找文本以数据流的形式发送给程序，来回移动的算法可能需要某种形式的缓冲。

4.2 蛮力查找

蛮力查找只是简单地缓慢通过一段文本，以寻找指定的字符串。它们差不多是最简单的算法，并且不涉及任何特殊的魔法。当你需要快速且无需太多想象力地解决简单的问题时，就很可能编写这种类型的算法。程序清单 4-1（brute1.c）显示了简单而缓慢的蛮力查找。

程序清单 4-1 一个简单而缓慢的蛮力查找

```
/*--- brute1.c --------------------------- Listing 4-1 ---------
 *  Simple brute search with no optimization.
 *  Easy to write but slow to execute.
 *
 *  #define DRIVER to have a command-line version compiled.
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define DRIVER  1   /* if #defined, runs test routine */

/*-------------------------------------------------
 * Search for string in text. Return pointer to first
 * instance of string, or NULL at end of text.
 *-------------------------------------------------*/
char * BruteSearch ( const char *text, const char *string )
{
    int len = strlen ( string );
```

```
    for ( ; *text; text++ )
        if ( strncmp ( text, string, len ) == 0 )
            return ( (char *) text );

    return ( NULL );
}

/*-------------------------------------------------
 * The main driver: accepts a string to search for and
 * a filename from the command line. It then searches
 * through the first 10,000 chars of the file and prints
 * the first 30 characters of the first match, if any,
 * and then quits.
 *-------------------------------------------------*/
#ifdef DRIVER

int main ( int argc, char *argv[] )
{
    char *search_for, *filename,
         *site;                     /* Site of match */
    FILE *fin;                      /* File to search */
    char *buffer;                   /* buffer from file */

    if ( argc < 3 )
    {
        fprintf ( stderr,
                  "Error! Usage: BRUTE1 string filename\n" );
        return ( EXIT_FAILURE );
    }
    else
    {
        search_for = argv[1];
        filename   = argv[2];
    }

    if (( fin = fopen ( filename, "rt" )) == NULL )
    {
        fprintf ( stderr, "Error: Cannot open %s\n", filename );
        return ( EXIT_FAILURE );
    }

    buffer = (char *) calloc ( 1, 10001 );
    if ( buffer == NULL )
    {
        fprintf ( stderr,
                  "Error! Cannot allocate buffer space.\n" );
        return ( EXIT_FAILURE );
    }

    fread ( buffer, 10000, 1, fin );

    site = BruteSearch ( buffer, search_for );

    if ( site == NULL )
        printf ( "%s NOT found\n", argv[1] );
    else
    {
```

```
        char solution[31];
        strncpy ( solution, site, 30 );
        solution [30] = '\0';

        printf ( "\nFound:\n%s\n", solution );
    }

    fclose ( fin );
    return ( EXIT_SUCCESS );
}
#endif
```

主要的查找函数 BruteSearch( ) 读取一段文本，并且对每个字符调用 strncmp( ) 函数，以测试字符串是否匹配。

可以对这个过程执行许多优化。首先，仅当在字符串的第一个字符上发生匹配时，才需要调用 strncmp( ) 函数，而不会为每个字符调用它。不过，即使在调用 strncmp( ) 函数之前测试是否存在这个匹配，我们仍将对每个字符执行两种测试：一种用于测试匹配，另一种用于测试文本末尾的标记。如我们前面所指出的，良好的查找性能期望对每个字符执行少于一次的比较。此时，提议的优化版本将至少执行两次比较。

一种解决方案是使用 switch 语句。许多编译器会把 switch 语句实现为一个**跳转表**（jump table），而不是一系列的 if/else 条件（跳转表是一个已知大小的数组，其元素由指向可执行代码的指针组成）。

当 switch 语句控制大量的 case 时尤其如此。换句话说，当使用跳转表时，switch 中的变量只需测试一次，即可跳转到相应 case 语句的代码上。在 Brutesearch 函数中使用以下代码将是理想的，尤其是当编译器把 switch 语句实现为跳转表时。

```
switch ( *text )

{
    case '\0':          /* test for end of text */
          return NULL;
    case *string:       /* test for match with string */
          // call to strncmp()
    default:
          text += 1; .  /* look at next character */
}
```

不幸的是，C 语言不允许在运行时评估 case 语句；因此，诸如 case * string 之类的代码是非法的。在编译时必须知道每个 case 的值。此外，这种短小的 switch 语句基本上都会被实现为一系列的 if/else 语句，这意味着查找文本中的每个字符仍然会被检查许多次。在 brute2. c（参见程序清单 4-2）中克服了这些问题。除了 BruteSearch 函数之外，其他代码都与 brute1. c 相同。

程序清单 4-2 使用查找表的蛮力查找

```
/*-------------------------------------------------
 *  Search for string in text using lookup table.
 *  Returns a pointer to the first instance of
 *  string, or NULL at end of text.
 *-------------------------------------------------*/
```

```
char * BruteSearch ( const char *text, const char *string )
{
    int len  = strlen ( string );

    /* the table. "static" assures its initialized to '\0's */
    static char lookup[UCHAR_MAX + 1];
    lookup[0]                              = 1;     /* EOT process */
    lookup[(unsigned char) (*string)] = 2;     /* a match */

    for ( ;; text++ )
    {
        switch ( lookup [(unsigned char) (*text)] )
        {
            case 0:
                break;          /* it's not EOT or a match */
            case 1:
                return ( NULL );        /* EOT */
            case 2:
                if ( strncmp
                      ( string + 1, text + 1, len - 1 ) == 0 )
                   return ( (char *) text );     /* a match */
                break;
            default:    /* good coding to include default */
                break;
        }
    }
}
```

第二个例程实现了一个查找表。这个表（lookup）包含256个值，其中每个值对应于以8位字节表示的一个可能的字符（我们假定将扫描的所有文本都使用8位字节）。使用static关键字将lookup表中的值都初始化为0。然后我们进入表中，并把特殊的值赋予我们感兴趣的字符：将表中的值2赋予字符串中的初始字符（lookup［*string］），以及将值1赋予文本末尾的字符（它的值是0）。然后，我们使用一个switch语句来确定每个输入字符的查找值：0是我们不感兴趣的字符，1表示文本末尾并且我们将返回，2则会触发对字符串比较函数的调用。

蛮力查找的这个版本运行速度比较快，这有三个原因：（1）它保持对检查的每个字符只执行一次比较；（2）表查找的执行速度比较快（单指针间接引用）；（3）用于查找的准备时间较短，因为大部分初始化工作是在启动时执行的（由于static关键字）。以这种方式初始化表意味着不能查找包含0个字节的字符串。万一我们想查找具有此值的字符串，将不得不把表显式初始化为一个非0值。

这种经过优化的蛮力方法有另外一个优点，可以在不产生更多比较的情况下来查找额外的查找字符串。例如，要执行不区分大小写的查找，可以设置查找表，使得*string的大写和小写版本都将返回2。这样，case 2将调用strncmp()的不区分大小写的版本。如果要查找多个字符串，并且它们的首字母不同，就可以把它们添加到查找进程中——同样，这也不会产生任何额外的比较。这是一种可扩展的、易于实现的、相当高效的蛮力查找算法。

注意这种算法的行为的一个方面。当在*string与*text之间找到一个匹配时，应该在text中的下一章中继续查找额外的匹配。我们可能想尝试跳过*string的长度，但是在许多情况下这样做是鲁莽的。例如，假设我们正在查找字符串ff，并且在文本fffff中找到了它。在该文本中，ff实

际上出现了4次，但是如果我们在每个匹配之后向前跳过字符串的完整长度，将只会找到两个ff：分别开始于第一个f和第三个f的ff。在蛮力查找方法中，仅当字符串不包含重复字符时，跳过字符串的整个长度才是可取的。本章中介绍了一个算法（即Boyer-Moore算法），它利用了跳过多个字符的可能性。

经过优化的蛮力方法实质上没有准备时间，并且当针对模式aa...aab查找全部内容都是as的文本时将处于其最坏情况。在这种情况下，它将在每个可能的位置尝试所有的PatLen字符（查找字符串的长度），这使得该方法的执行时间与PatLen * TextLen成正比。不过，如果运气好的话，它可能只需要把查找文本的每个字符查看一次，比如当在全部内容都是as的文本中查找b时。典型的情况介于这两种极端情况之间。

经过优化的蛮力查找算法很容易理解，并且工作得相当好。可以在大多数情况下使用它，比如需要一个简单的字符串查找，以及不必通过精心编写代码以尽可能提高其执行速度。

4.3 Boyer-Moore查找

1977年，R. S. Boyer和J. S. Moore发表了他们的经典论文“A Fast String Searching Algorithm”。Boyer-Moore技术展示了一种对字符串查找问题的新解决方案：它利用两种方法，使得无需检查文本中的所有字符即可查找文本变为可能。这两种方法称为**启发式方法**（heuristic）（即使用以前的信息来寻找解决方案的技术），涉及使用预先计算的表——这是在前面讨论蛮力查找算法时介绍过的概念，在本章剩余部分这个概念将频繁出现。

4.3.1 启发式方法#1：跳过字符

第一种启发式方法出奇简单。Boyer-Moore算法会展示要查找的文本、要搜索的模式，以及一个标记，它指向查找进程目前正在查看的位置：

```
目标模式          → corn
查找的当前标记    →    *
查找文本          → Oaks from acorns grow
```

注意：查找指针开始于查找模式的右端。由于文本中的s不与查找模式中的任何字符匹配，因此查找可以向右移动4个字符——即查找字符串的长度：

```
(原始模式位置)    → corn
移动1次之后的位置 → corn
当前标记          →    *
查找文本          → Oaks from acorns grow
```

现在，查找模式中的n与o对齐，由于查找模式中出现了o，我们再次右移，但是这一次只移动2个字符，即测试匹配所需的数量：

```
(原始位置)        → corn
移动1次之后的位置 →  corn
移动2次之后的位置 →   corn
当前标记          →    *
```

```
查找文本                          → Oaks from acorns grow
```

这一次，n 与空格对齐。由于查找模式中没有空格，我们将移动 4 个字符：

```
(原始位置)              → corn
移动 1 次之后的位置     →  corn
移动 2 次之后的位置     →   corn
移动 3 次之后的位置     →    corn
当前标记                      →    *
查找文本                      → Oaks from acorns grow
```

现在，n 与 r 对齐。由于 r 存在于目标模式中，因此只需移动 1 个字符：

```
(原始位置)              → corn
移动 1 次之后的位置     →  corn
移动 2 次之后的位置     →   corn
移动 3 次之后的位置     →    corn
移动 4 次之后的位置     →     corn
当前标记                →       *
查找文本                → Oaks from acorns grow
```

最后，找到匹配。这种技巧允许快速跳过文本：只检查 4 个字符，从而排除了前 11 个字符，不予考虑。

实现这个概念需要使用一个表（即代码中的 CharJump），它包含一个无符号的整数，用于表示在查找文本和模式字符串中使用的每个可能的字符。这个表中的每个表项告诉我们：需要把查找字符串向右移动的距离，以执行下一次比较。显然，如果我们检查一个根本没有出现在查找字符串中的字符，就可以向前移动整个字符串的长度。如果字母出现在查找字符串中，就可以只向前移动该字符到字符串右端的距离。例如，在前面的文本中，当 corn 中的 n 与 acorn 中的 r 对齐时，我们就右移 1 个字符，即 r 到字符串右端的距离。计算方法总结如下（程序清单 4-3 中给出了实现代码）：如果 t [] 是要查找的文本，p [] 是目标模式，并且 PatLen 是 p 的长度，那么对于每个可能的字母字符 c，在 p 中查找最右边出现的 c。如果没有找到 c，则 CharJump [c] = PatLen。否则，如果在位置 i 处找到 c，则 CharJump [c] = PatLen-i。

因此，对于前面的示例，CharJump 将计算如下：

```
CharJump['c'] = 3
CharJump['o'] = 2
CharJump['r'] = 1
CharJump['n'] = 0
CharJump [all others] = 4
```

这个表中的值表示当两个字符串比较失败时将把指针从其当前位置向右移动的距离。考虑下面的查找：

```
目标模式                → corn
当前标记                →  *
查找文本                → planting new crops
```

匹配 n 并移动指针：

```
目标模式          → corn
当前标记          →    *
查找文本          → planting new crops
```

这就得到一个不匹配的结果。参考 CharJump ['a'] 指示把指针向右移动 4 个空格。当指针经过查找模式的末尾时，它会“钩住”模式，并拉着它向前移动：

```
目标模式          → corn
当前标记          →    *
查找文本          → planting new crops
```

把指针移动 4 个位置将只会把查找模式移动 3 个位置，理解这一点很重要。

4.3.2 启发式方法#2：重复模式

当查找字符串中包含重复模式时，Boyer-Moore 的第二种启发式方法就派上了用场。它要稍微复杂一些。假定有以下查找：

```
目标模式          → door to door
标记              →           *
查找文本          → He went door to door
```

直到以下位置，标记才会成功地匹配：

```
目标模式          → door to door
标记              →       *
查找文本          → He went door to door
```

只使用 CharJump 表，我们可能尝试把标记向前移动 CharJump ['t'] = Patlen-5-1（或 6）个空格（记住：当标记经过查找模式的末尾时，它将钩住模式，并拉着它向前移动）。

```
目标模式          → door to door
标记              →           *
查找文本          → He went door to door
```

这将对齐 t（但是不会对齐后面更多的字符），然后在经过另外几次移动之后，最终将找到匹配。不过，甚至有可能做得更好。由于查找模式中的最后 5 个字符已经匹配，在这个例子中，查找模式的前 4 个字符将自动具有部分匹配。换句话说，为了利用查找模式自身重复的事实，可以将标记前移 13 个（而不是 6 个）空格：

```
目标模式          →     door to door
标记              →                 *
查找文本          → He went door to door
```

这里的技巧是：知道要移动多远的距离。通过使用第二个辅助表（在代码中称为 MatchJump）来计算这个距离。使用我们以前用过的相同表示法，我们希望像下面这样计算 MatchJump [k]：首先，寻找 p 中与 p [k + 1] ... p [Patlen - 1] 匹配的最右边的子串。令 s 是 s 的最大值，满足 strncmp（p + s，p + k + 1，PatLen - k - 1）= = 0，且 p [s - 1] ! = p [k]。那么，MatchJump [k]

= PatLen – s。如果没有这样的匹配子串，则寻找一个匹配前缀。如果 q 是 q 的最大值，满足 strncmp（p，p + PatLen – 1 – q，q） = =0，则 MatchJump［k］ =2 * PatLen – q – k – 1。

虽然原则上很容易理解如何计算 MatchJump［］，但是设计一种能够处理各种情况的高效算法却很棘手，甚至更难以解释。如果有兴趣，可以检查参考资料［Smit 1982］，了解关于计算 MatchJump［］的详细信息。

由于 Boyer-Moore 算法在查找给定的大文本块时工作得最好，并且它不会在‘\0’字符处停止，因此必须给程序清单 4-3 中的实现提供它要查找的文本块的长度。该代码还包括一个测试驱动程序，它查找一个文件，找出每一处出现的指定字符串。

程序清单 4-3　Boyer-Moore 字符串查找的实现

```
/*-- boyermor.c ---------------------------- Listing 4-3 --------
 * Boyer-Moore string search routine
 *
 *   Preprocessor switches: if #defined:
 *
 *      DEBUG will cause the search routine to dump its tables
 *            at various times--this is useful when trying to
 *            understand how MatchJump is generated
 *
 *      DRIVER will cause a test driver to be compiled
 *
 *-----------------------------------------------------------------*/

#define DRIVER  1
/* #define DEBUG   1 */

#if defined(DEBUG)
#define SHOWCHAR for (uT=1; uT<= PatLen; uT++)  \
                    printf(" %c ", String[uT-1])
#define SHOWJUMP for (uT=1;uT<= PatLen;uT++)    \
                   printf("%2d ", MatchJump[uT])
#define SHOWA    printf("  uA = %u  ", uA)
#define SHOWB    printf("  uB = %u", uB)
#define SHOWBACK for (uT=1;uT<= PatLen;uT++)    \
                   printf("%2d ", BackUp[uT])
#define NL       printf("\n")

unsigned uT;
#else
#define SHOWCHAR
#define SHOWJUMP
#define SHOWA
#define SHOWB
#define SHOWBACK
#define NL
#endif

#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
```

```
#define AlphabetSize (UCHAR_MAX + 1)    /* For portability */

#ifndef max
#define max(a,b) ((a) > (b) ? (a) : (b))
#endif

char *BoyerMoore ( const char * String, /* search for this */
                   const char * Text,   /* ...in this text */
                   size_t TextLen )     /* ...up to here.  */
{
            /* array of character mismatch offsets */
    unsigned CharJump[AlphabetSize];

            /* array of offsets for partial matches */
    unsigned *MatchJump;

            /* temporary array for MatchJump calculation */
    unsigned *BackUp;
    size_t   PatLen;
    unsigned u, uText, uPat, uA, uB;


    /* Set up and initialize arrays */
    PatLen = strlen ( String );
    MatchJump = (unsigned *)
         malloc ( 2 * sizeof ( unsigned ) * ( PatLen + 1 ));
    BackUp = MatchJump + PatLen + 1;

    /* Heuristic #1 -- simple char mismatch jumps ... */

    memset ( CharJump, 0, AlphabetSize * sizeof(unsigned) );
    for ( u = 0 ; u < PatLen; u++ )
        CharJump[((unsigned char) String[u])]
                    = PatLen - u - 1;

    /* Heuristic #2 -- offsets from partial matches ... */
    for ( u = 1; u <= PatLen; u++ )
    MatchJump[u] = 2 * PatLen - u;
                            /* largest possible jump */
SHOWCHAR; NL;
SHOWJUMP; NL;

u  = PatLen;
uA = PatLen + 1;
while ( u > 0 )
{
    BackUp[u] = uA;
    while ( uA <= PatLen &&
            String[u - 1] != String[uA - 1] )
    {
        if ( MatchJump[uA] > PatLen - u )
             MatchJump[uA] = PatLen - u;
        uA = BackUp[uA];
    }
    u--;
    uA--;
}
```

```
SHOWJUMP; SHOWA; SHOWBACK; NL;

for ( u = 1; u <= uA; u++ )
    if ( MatchJump[u] > PatLen + uA - u )
         MatchJump[u] = PatLen + uA - u;

uB = BackUp[uA];
SHOWJUMP; SHOWB; NL;

while ( uA <= PatLen )
{
    while ( uA <= uB )
    {
        if ( MatchJump[uA] > uB - uA + PatLen )
            MatchJump[uA] = uB - uA + PatLen;
        uA++;
    }
    uB = BackUp[uB];
}
SHOWJUMP; NL;

/* now search */
uPat =  PatLen;          /* tracks position in Pattern */
uText = PatLen - 1;      /* tracks position in Text */
while ( uText < TextLen && uPat != 0 )
{
    if ( Text[uText] == String[uPat - 1])  /* match? */
        {
            uText--;     /* back up to next */
            uPat--;
        }
        else  /* a mismatch - slide pattern forward */
        {
            uA = CharJump[((unsigned char) Text[uText])];
            uB = MatchJump[uPat];
            uText += max(uA, uB);  /* select larger jump */
            uPat = PatLen;
        }
    }

    /* return our findings */
    if ( uPat == 0 )
        return ( (char *) ( Text + ( uText + 1 )));  /* a match */
    else
        return ( NULL );                  /* no match */
}

/*-------------------------------------------------------------
 * The main driver, activated by #defining DRIVER.
 * Will print all occurrences of a match in the first
 * 10,000 characters of the target file.
 *-------------------------------------------------------------*/

#ifdef DRIVER

#define MAX_TEXT_SIZE 10000u
```

```
int main ( int argc, char *argv[] )
{
    char *SearchFor, *Filename;

    FILE *Fin;                      /* File to search */
    char *Buffer;                   /* Buffer from file */

    char *start, *p;
    int i;
    size_t   TextSize;
    unsigned count;

    if ( argc != 3 )
    {
        puts ( "Usage is: boyermor search-string filename\n" );
        return ( EXIT_FAILURE );
    }
    else
{
    SearchFor = argv[1];
    Filename  = argv[2];
}

if (( Fin = fopen ( Filename, "r" )) == NULL )
{
    fprintf ( stderr, "Can't open %s\n", Filename );
    return ( EXIT_FAILURE );
}

/* allocate search buffer and fill it with target file */
Buffer = (char*) malloc ( MAX_TEXT_SIZE + 1 );
if ( Buffer == NULL )
{
    puts ( "Error! Could not allocate buffer space\n" );
    return ( EXIT_FAILURE );
}

TextSize = fread ( Buffer, 1, MAX_TEXT_SIZE, Fin );
fclose ( Fin );

p = Buffer;
count = 0;
while ( count < TextSize )
{
    if ( *p == '\n' )
         *p = '\0';
    p++;
    count++;
}

/* now search repeatedly */

start = BoyerMoore ( SearchFor, Buffer, TextSize );
if ( start == NULL )              /* no match found */
    printf ( "\n%s Not Found.\n", SearchFor );
else                              /* match found */
    while ( start != NULL )
```

```
    {
        for ( p = start; ; p-- )  /* find start of line */
        {
            if ( *p == '\0' )
            {
                p++;
                break;
            }
            else
                if ( p == Buffer )
                    break;
            }                           /* print the match */
            printf( "Found:\n%s\n", p );
            for ( i = start - p; i > 0; i-- )
                fputc ( ' ', stdout );
            printf ( "%s\n\n", SearchFor );
            start =                     /* continue the search */
                BoyerMoore ( SearchFor, start + 1,
                             TextSize - ( start - Buffer ) - 1 );
        }
    return ( EXIT_SUCCESS );
}
#endif
```

该代码首先计算 MatchJump 和 CharJump 数组。然后遍历查找文本；如果找到一个字符不匹配，它将从合适的 MatchJump 和 CharJump 元素中选取更大的元素。

注意：可以通过删除 MatchJump 表来简化 Boyer-Moore 代码。在适当的情况下，这仍然允许 Boyer-Moore 算法保持其大部分速度，同时减少它的准备时间和空间要求。为此，需要在查找循环中用 uB = PatLen – uPat + 1 替换 uB 的计算。这个计算结果与在计算 MatchJump 时获得的最低限度的合法结果完全相同。如果没有这个值，查找循环实际上会向后移动。

Boyer-Moore 算法在计算运行时间上，稍微复杂一些。很容易推导出它的准备时间与 PatLen + AlphabetSize 成正比，但是很难确定最坏情况下的运行时间。事实证明，如果文本中不包含模式的匹配，该算法最多进行 6 * TextLen 次字符比较。其他程序设计者推测这个界限可能低到 2 * TextLen 次字符比较。这是 Boyer-Moore 算法显示其重要性的平均情况。一般来讲，它只要求时间与 TextLen/PatLen 成正比。换句话说，Boyer-Moore 算法通常只会检查一小部分文本。举例来说，记住第一个示例查找只执行了 4 次比较，而跳过了 11 个字符。这种次线性使得 Boyer-Moore 算法具有吸引力。

4.4　多字符串查找

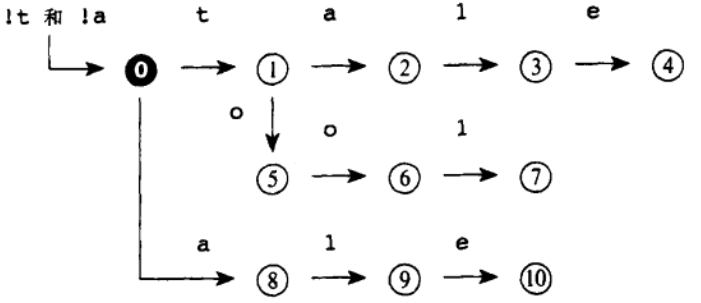
假设我们需要在一个大数据库中同时查找多个字符串。这个问题很常见；例如，我们可能想查找一个医疗卫生数据库，从中找出对 chest pain、angina pectoris 或 heart attack 的任何引用。当然，我们可以简单地从数据库中一次查找其中一个短语，但是有一种更灵活的解决方案，可以通过一次遍历来查找多个短语。

这里介绍的方法最初是由 Alfred Aho 和 M. J. Corasick 描述的［Aho 1975］。他们开发的技术主要考虑的是构造一种可以定位关键字的有限状态机（这里的**关键字**（keyword）是指要查找的字符串）。这种方法可以确定给定文本中属于关键字列表的所有子串。这些子串可能会重叠。

可以首先通过考虑的一个示例，最佳地描述该算法。假设我们想从一段文本中查找关键字

tale、tool 和 ale。对于这 3 个关键字，在三个表中描述了一种合适的状态机，如图 4-1 所示。

a）Goto 函数



所有未定义的转换都作为 FAIL_STATE 状态处理。注意：定义了从状态 0 进行的所有转换；只要作为反回状态 0 的转换即可

b）Failure 函数

state	1	2	3	4	5	6	7	8	9	10
Fail[state]	0	8	9	10	0	0	0	0	0	0

c）Output 函数

state	Output[state]
4	tale, ale
7	tool
10	ale

图 4-1 用于多个字符串查找的有限状态机

该状态机的关键是图 4-1a 中的 Goto 函数。它定义了一个状态网络，用于跟踪我们曾经看到的内容。我们从状态 0 开始，然后检查输入流中的每个字符。例如，如果在状态 0 时看到一个 a，则移动到状态 8。然后，一个 l 将把我们带到状态 9。不过，如果下一个字符不是 e，我们就参考 Fail [9]，并且改变到状态 0。值得注意的是状态 9，Goto [9，'e'] = = 10，而 Goto [9，其他任何字符] = = FAIL_STATE。如果我们利用非空的 Output 函数到达一种状态，就会找到一个匹配的字符串。以下是伪代码：

```
state = 0;
while ((c = getchar()) != EOF)  /* read text */
{
     while (Goto[state, c] == FAIL_STATE)
          state = Fail[state];

     state = Goto[state, c];
     if (Output[state] != NULL)
          report(Output[state]);
}
```

如你所看到的，一旦图 4-1 中的多个表可用，查找过程将很直观。构造这些表是这个算法的难点。我们通过两个步骤来解决这个问题。首先，构造 Goto 函数并部分地构造 Output 函数。我们从一个空的 Goto 函数开始，然后逐步从查找单词中添加每个字符。如果可能，我们将“重叠”以

相同字符开头的单词。例如，要构建图 4-1 中的 Goto 函数，首先从一个空的 Goto 函数开始（如图 4-2a 所示），然后插入单词 tale（图 4-2b）、单词 tool（图 4-2c），最后插入单词 ale（图 4-1a）。

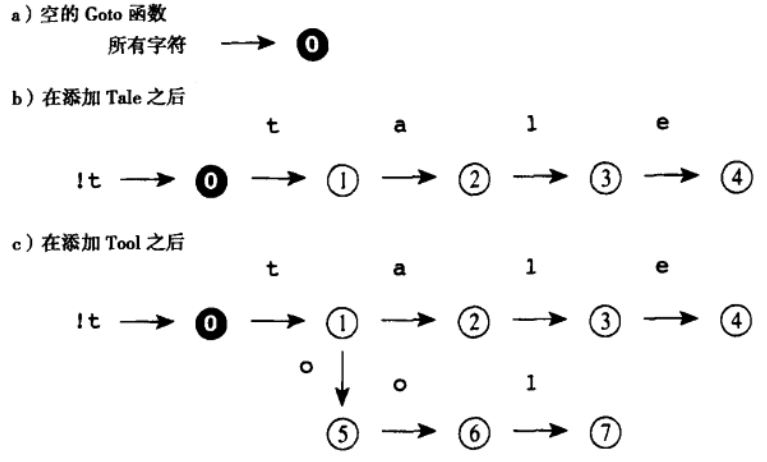


图 4-2　加载有限状态表

同时，我们还定义了 Output 函数的开头部分。这些计算的粗略伪代码如下：

```
HighState = 0;
for(word=list_of_words; word!=NULL; word=next_word)
{
     state = 0;
     j = 1;
     /* try to overlay on existing word(s) */
     while (Goto[state, word[j]) != FAIL_STATE)
     {
          state = Goto[state, word[j]];
          j += 1;
     }
     /* now create new states as needed */
     while (word[j] != '\0')  /* scan to end of string */
     {
          HighState += 1;      /* a new state */
          Goto[state, word[j]] = HighState;
          state = HighState;
     }

     /* we are at the end of word[].  Add to Output */
     Output[state] += word;  /* append word */
}

/* eliminate all FAIL_STATE transitions from state 0 */
for(i=0; i<MAXCHARS; i++)
     if (Goto[0, i] == FAIL_STATE)
          Goto[0, i] = 0;
```

计算 Fail 函数发生在构造表中的第二步即最后一步。在此，我们必须考虑：（1）重叠的查找单词；（2）给定的状态可能找到多个查找单词的可能性。我们继续逐步检查 Goto 的所有状态。如

果我们将**深度**（depth）定义为状态 x 至状态 0 的距离，那么状态 1 和 8 位于深度 1，而状态 2、5 和 9 则位于深度 2。根据直觉，Fail[depth 1 states] 必定会把我们带到状态 0。针对深度 2 的状态的 Fail 函数现在依赖于深度 0 和 1 的状态的 Fail 和 Goto 函数。归纳一下，针对深度 n 的状态的 Fail 函数依赖于更低深度的所有状态的 Fail 和 Goto 函数。

例如，考虑以前在图 4-1 中展示的 Fail[2] 的计算。将我们从状态 1 带到状态 2 的字符是 a。现在我们查找从状态 0 到状态 1 的所有转换，找出也利用了 a 的另一个转换。我们只找到如下一个 a：它将我们从状态 0 带到状态 8。因此，将 Fail[2] 设置为 8。注意 Fail 的计算如何证明了关键字 tale 和 ale 之间的重叠。

同时会更新 Output 函数。当发现重叠时，我们只需要连接两种重叠状态的 Output 函数。在下面这段伪代码中，队列是一个先进先出的链表。代码 queue = queue + x 把元素 x 添加到队尾，而 queue = queue – x 则从队头删除 x：

```
queue = empty;

/* examine state 0 first */
for (i=0; i<MAXCHARS; i++)
     if (Goto[0, i] != 0)  /* find all depth 1 states */
     {
          s = Goto[0, i];
          Fail[s] = 0;
          queue = queue + s;
     }

/* now examine states of depth 2 and greater */
while (queue != empty)
{
     r = next state from queue;
     queue = queue - r;
     for (i=0; i<MAXCHARS; i++)
     {
          if (Goto[r, i] != FAIL_STATE)
          {
               s = Goto[r, i];  /* i takes us from r to s */
               queue = queue + s;
               state = Fail[r]; /* look back one to n levels */

               /*  find a valid transition that uses i. Note
                *  that we can always find such a transition
                *  because at the very least,
                *   Goto[0, i] = 0 or some other valid state.
                */
               while (Goto[state, i] == FAIL_STATE)
                    state = Fail[state];
               Fail[s] = Goto[state, i];
               Output[s] = Output[s] + Output[Fail[s]];
          }
     }
}
```

程序清单 4-4 中展示了实现这段伪代码的实际代码。我们必须处理许多影响算法的速度和内存需求的实现问题。首先，我们必须决定状态机具有多少种可能的状态。合理的选择是将状态存

储为整数。下一个关键问题是怎样存储 Goto 函数。最简单的算法设计为每种状态使用一个跳转表。该表包含每个可能的字符的下一种状态。不幸的是，这将需要 512 个字节（一共有 256 个字符，按每个整数至少需要两个字符计算），以便每种状态都能够处理 ASCII 文本。这样，具有 100 种状态的状态机将为 Goto 函数消耗 50 KB 的空间。由于这个算法的许多应用可能只需要 20 ~ 40 种状态（也就是说，所有查找短语的总长度是 20 ~ 40 个字符），这种方法将是可接受的。实际上，如果将状态的数量限制为 256 种，则可以使用一个 8 位字节来存储每种状态，从而可以把跳转表所需的内存减半。

程序清单 4-4　查找多个字符串

```
/*--- msrch.c ---------------------------- Listing 4-4 --------
 * Purpose: search text for multiple keywords simultaneously
 * Switches: DRIVER  - will cause a test driver to be compiled
 *           MAXCHAR - maximum number of symbols recognized
 *
 * Usage: The sample driver illustrates all the key points.
 *        There are three routines:
 *
 *        (1) MsrchInit ( struct kword *) is passed list of
 *            words to search for
 *
 *        (2) MsrchGo ( int (*MsrchData) (),
 *                      void (*MsrchSignal) (char *) );
 *            does the work. It uses two pointers to functions:
 *            the first retrieves a character, the second is
 *            called when a match is found.
 *
 *        (3) MsrchEnd ( void ) cleans up the work areas
 *
 *-------------------------------------------------------------*/

#define DRIVER 1              /* compile a test driver */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct kword {                /* linked list of keywords */
    unsigned char *word;      /*   to search for.        */
    struct  kword *next;
};


#define MAXCHAR  256          /* max number of different chars
                                    we search for */
static int MaxState;          /* max number of states
                                    we have room for */

static int *MatchArray;       /*  First level of matching.
                               *  Possible values:
                               *      (1) EMPTY_SLOT  -2
                               *      (2) a character
                               *      (3) MULTI_WAY   -1
                               */
```

```
#define MULTI_WAY   -1      /* flags for match_array */
#define EMPTY_SLOT  -2

union GotoTable {       /* values in MatchArray take us here:  */
    int GotoState;      /* go here if MatchArray is a character*/
    int *BranchTable;   /* or to this MULTI_WAY branching table*/
} static *GotoArray;

#define FAIL_STATE  -1      /* in GotoState or BranchTable,
                               this means failure */

/* OutArray[] is the Output function */
                        /* list of keywords 'found' by states */
static struct kword **OutArray;

/* FailArray[] is the Fail function */
static int *FailArray;      /* failure transition array */


/* variable to track next free state */
static int HighState;

/* functions we use */
static void AddStateTrans ( int, int, int );
static void ComputeFail ( void );
static void Enter ( unsigned char * );
static void FindFail ( int state, int s, int a );
static void QueueAdd ( int *queue, int qbeg, int new );

/* set up tables needed by MsrchGo() */
void MsrchInit ( struct kword *klist )
{
    int i;
    struct kword *ktemp;

    /* compute maximum number of possible states */
    MaxState = 1;
    for ( ktemp = klist; ktemp != NULL; ktemp = ktemp->next )
          MaxState += strlen ( ktemp->word );

    /* allocate space for arrays */

    MatchArray = (int *) malloc ( sizeof(int) * MaxState );
    GotoArray  = (union GotoTable *) malloc (
                        sizeof(union GotoTable) * MaxState );
    OutArray   = (struct kword **) malloc (
                        sizeof(struct kword *) * MaxState );
    FailArray  = (int *) malloc ( sizeof(int) * MaxState );

    /* initialize state arrays */
    for ( i = 0; i < MaxState; i++ )
    {
        MatchArray[i] = EMPTY_SLOT;
        OutArray[i]   = NULL;
    }

    /* initialize state_array[0] */
    HighState = 0;
```

```
    AddStateTrans ( 0, 'a', FAIL_STATE );
         /* force a multiway table */
    AddStateTrans ( 0, 'b', FAIL_STATE );

    /* step through keywords */
    for ( ; klist != NULL; klist = klist->next )
         Enter ( klist->word );

    /* setup return to zero transitions for state[0] */
    for  ( i = 0; i < MAXCHAR; i++ )
        if ( GotoArray[0].BranchTable[i] == FAIL_STATE )
             GotoArray[0].BranchTable[i] = 0;

    /* and compute failure array */
    ComputeFail();
}
/* add transition from OldState -> NewState for MatchChar */
static void AddStateTrans ( int OldState,
                            int MatchChar,
                            int NewState )
{
    int i, *temp;

    /* is this slot empty? */
    if ( MatchArray[OldState] == EMPTY_SLOT )  /* this is easy */
    {
        MatchArray[OldState] = MatchChar;
        GotoArray[OldState].GotoState = NewState;
    }
        /* is there already a multi-way table? */
    else
    if ( MatchArray[OldState] == MULTI_WAY ) /* easy, too */
         GotoArray[OldState].BranchTable[MatchChar] = NewState;

        /* need to convert to multi-way table */
    else
    {
        temp = (int *) malloc ( sizeof(int) * MAXCHAR );
        for ( i = 0; i < MAXCHAR; i++ )
            temp[i] = FAIL_STATE;

        /* copy data from single way branch */
        temp[MatchArray[OldState]] =
            GotoArray[OldState].GotoState;

        /* and new data */
        temp[MatchChar] = NewState;

        /* and load it all into state_array */
        MatchArray[OldState] = MULTI_WAY;
        GotoArray[OldState].BranchTable = temp;
    }
}

/* add kword to list of words our machine recognizes */
static void Enter ( unsigned char *kword )
{
    int state, k;
```

```
    char *save;
    struct kword *ktemp;

    state = 0;
    save = kword;    /* keep a copy */
    /*  first, see whether we can place this word
     *  on top of an existing one
     */

    for ( ; *kword != '\0'; kword++ )
    {
        /* is this a single char slot? */

        if ( MatchArray[state] == *kword )
            state = GotoArray[state].GotoState;

        else      /* multi-way? */
        if ( MatchArray[state] == MULTI_WAY )
        {
            if (( k = GotoArray[state].BranchTable[*kword] )
                    == FAIL_STATE )
                break;
            else    /* we have a transition for this char */
                state = k;
        }
        else        /* no match for this char */
            break;
    }

    /* now add new states as needed */
    for ( ; *kword != '\0'; kword++ )
    {
        HighState += 1;
        if ( HighState >= MaxState )  /* uh-oh ... */
        {
            fputs( "INTERNAL ERROR: too many states\n", stderr );
            exit ( EXIT_FAILURE );
        }
        AddStateTrans ( state, *kword, HighState );
        state = HighState;
    }

    /* now add this keyword to output list for final state */
    ktemp = (struct kword *) malloc ( sizeof ( struct kword ));
    ktemp->word = save;
    ktemp->next = OutArray[state];
    OutArray[state] = ktemp;
}

/* build FailArray and update GotoArray */
static void ComputeFail()
{
    int *queue, qbeg, r, s;
int i;

/* allocate a queue */
queue = (int *) malloc ( sizeof ( int ) * MaxState );
```

```
    qbeg = 0;
    queue[0] = 0;

    /* scan first level and setup initial values for FailArray */
    for ( i = 0; i < MAXCHAR; i++ )
        if (( s = GotoArray[0].BranchTable[i] ) != 0 )
        {
            FailArray[s] = 0;
            QueueAdd ( queue, qbeg, s );
        }

    /* now scan lower levels */
    while ( queue[qbeg] != 0 )
    {
        /* pull off state from front of queue and advance qbeg*/

        r = queue[qbeg];
        qbeg = r;

        /* now investigate this state */
        if ( MatchArray[r] == EMPTY_SLOT )
            continue;       /* no more to do */
        else
        if ( MatchArray[r] == MULTI_WAY )
        {
            /* scan its subsidiary states */
            for ( i = 0; i < MAXCHAR; i++ )
                                /* scan BranchTable */
                if (( s = GotoArray[r].BranchTable[i] )
                        != FAIL_STATE )
                {               /* add new state to queue */
                    QueueAdd ( queue, qbeg, s );
                    FindFail ( FailArray[r], s, i );
                }
        }
        else  /* single char */
        {
            QueueAdd ( queue, qbeg, GotoArray[r].GotoState );
            FindFail ( FailArray[r], GotoArray[r].GotoState,
                        MatchArray[r] );
        }
    }

    /* tidy up */
        free ( queue );
    }

    /*-------------------------------------------------------------
     *  Actually compute failure transition.  We know that 'a'
     *  would normally cause us to go from state s1 to s2.
     *  To compute the failure value, we backtrack in search
     *  of other places 'a' might go.
     *-------------------------------------------------------------*/
    static void FindFail ( int s1, int s2, int a )
    {
        int on_fail;
        struct kword *ktemp, kdummy, *out_copy, *kscan;
```

```
    for ( ; ; s1 = FailArray[s1] )
        if ( MatchArray[s1] == a )
        {
            if (( on_fail = GotoArray[s1].GotoState )
                        != FAIL_STATE )
                break;
        }
        else
        if ( MatchArray[s1] != EMPTY_SLOT )
            if (( on_fail = GotoArray[s1].BranchTable[a] )
                        != FAIL_STATE )
                break;

    FailArray[s2] = on_fail;

    /* merge output lists */

    /* first, make a copy of OutArray[on_fail] */
    if ( OutArray[on_fail] == NULL )
        out_copy = NULL;
    else
    {
        kscan = OutArray[on_fail];
        out_copy = malloc ( sizeof ( struct kword ));
        out_copy->word = kscan->word;
        out_copy->next = NULL;
        for ( kscan = kscan->next;
              kscan != NULL;
              kscan = kscan->next )
        {
            ktemp = malloc ( sizeof ( struct kword ));
            ktemp->word = kscan->word;
            ktemp->next = out_copy->next;
            out_copy->next = ktemp;
        }
    }

    /* now merge them */
    if (( kdummy.next = OutArray[s2] ) != NULL )
    {
        ktemp = &kdummy;
        for ( ; ktemp->next->next != NULL; ktemp = ktemp->next )
                ;
        ktemp->next->next = out_copy;
    }
    else
        OutArray[s2] = out_copy;
}

/* add new to end of queue */
static void QueueAdd ( int *queue, int qbeg, int new )
{
    int q;

    q = queue[qbeg];
    if ( q == 0 )           /* is list empty? */
```

```
        queue[qbeg] = new;  /* yes */
    else                    /* no: scan to next-to-last link */
    {
        for ( ; queue[q] != 0; q = queue[q] )
            ;
        queue[q] = new; /* put this state at end of queue */
    }

    /* and terminate list */
    queue[new] = 0;
}

/* do the actual search */
void MsrchGo ( int  (*MsrchData) (),
               void (*MsrchSignal) (char *) )
{
    int state, c, g, m;
    struct kword *kscan;

    state = 0;
    while (( c = MsrchData() ) != EOF )
    {
        /* what is goto ( state, c ) ? */
        for ( ;; )
        {
            /*-----------------------------------------------
             *  We cheat slightly in the interest of
             *  speed/simplicity. The machine will spend most
             *  of its time in state==0, and this state is
             *  always a MULTI_WAY table. Since this is a
             *  simple test, we make it first and try to save
             *  the calculation of an array index
             *-----------------------------------------------*/

            if ( state == 0 ||
                 ( m = MatchArray[state] ) == MULTI_WAY )
                g = GotoArray[state].BranchTable[c];
            else
            if ( m == c )
                g = GotoArray[state].GotoState;
            else
                g = FAIL_STATE;

            if ( g != FAIL_STATE )
                break;

            state = FailArray[state];
        }
        state = g;

        /* anything to output? */
        if (( kscan = OutArray[state] ) != NULL )
            for ( ; kscan != NULL; kscan = kscan->next )
                MsrchSignal ( kscan->word );
    }
}
```

```
/* free all the arrays we created */
void MsrchEnd ( void )
{
    int i;
    struct kword *kscan;

    for ( i = 0; i < MaxState; i++ )
          if ( MatchArray[i] == MULTI_WAY )
               free ( GotoArray[i].BranchTable );

    free ( MatchArray );
    free ( GotoArray );
    free ( FailArray );

    for ( i = 0; i < MaxState; i++ )
        if ( OutArray[i] != NULL )
             for ( kscan = OutArray[i];
                   kscan!=NULL;
                   kscan = kscan->next )
                free ( kscan );
    free ( OutArray );
}

/*-------------------------------------------------------------
 * This test driver expects a command line of the form
 *    msrch file word-1 word-2 word-3 .... word-n
 *
 * It will then search file for all words on the command line.
 * The results are written to stdout. This illustrates all the
 * features of using the multisearch routines.
 *
 * This is an admittedly simple design--the search routine would
 * certainly be faster if the character fetch routine was put
 * directly into the MsrchGo() module.  However, to avoid using
 * application-specific code in the demonstration version of
 * these routines, it is coded as a separate subroutine.
 *-------------------------------------------------------------*/

#ifdef DRIVER

#define BUFSIZE 200

FILE *infile;
char inbuf[BUFSIZE];
char *inbufptr;
int  linecount;

/* declare the routines that MsrchGo() will use */
int RetrieveChar ( void );
void FoundWord();

int main ( int argc, char **argv )
{
    char infile_name[80];
    struct kword *khead, *ktemp;
    int i;
```

```
    if ( argc < 3 )
    {
        fprintf ( stderr,
            "Usage: msrch infile word-1 word-2 ... word-n\n" );
        exit ( EXIT_FAILURE );
    }

    strcpy ( infile_name, argv[1] );

    if (( infile = fopen ( infile_name, "r" )) == NULL )
    {
        fprintf ( stderr, "Cannot open %s\n", infile_name );
        exit ( EXIT_FAILURE );
    }
    linecount = 0;
    inbufptr = NULL;

    /* turn command-line parameters into a list of words */
    khead = NULL;
    for ( i = 3; i <= argc; i++ )
    {
        ktemp = (struct kword *) malloc ( sizeof ( struct kword ));
        ktemp->word = argv[i-1];
        ktemp->next = khead;
        khead = ktemp;
    }

    MsrchInit ( khead ); /* setup system; pass list of words */

    /* Now search. Note call to functions by use of pointers */
    MsrchGo ( RetrieveChar, FoundWord );

    MsrchEnd();             /* clean up */
    return ( EXIT_SUCCESS );
}

/* get next character from input stream.  Routine returns either
 *   (a) a character (as an int without its sign extended), or
 *   (b) EOF
 */
int RetrieveChar ( void )
{
    int c;

    if ( inbufptr == NULL || *(++inbufptr) == '\0' )
    {    /* read a new line of data */
        if ( fgets ( inbuf, BUFSIZE, infile ) == NULL )
        {
            fclose ( infile );
            return ( EOF );
        }
        inbufptr = inbuf;
        linecount += 1;
    }

    c = *inbufptr;
    c &= 0x00FF;    /* make sure it is not sign extended */
```

```
    return ( c );
}

/* FoundWord: called by MsrchGo() when it finds a match */
void FoundWord(char *word)
{
    int i;

    fprintf ( stdout, "Line %d\n%s", linecount, inbuf );

    i = ( inbufptr - inbuf ) - strlen ( word ) + 1;
    for ( ; i > 0; i-- )
        fputc ( ' ', stdout );

    fprintf ( stdout, "%s\n\n", word );
}
#endif
```

程序清单4-4中的代码的目标是节约内存。由于大多数状态只有一个从它们导出的有效转换（在图4-1、图4-8和图4-10中限定），我们使用两个数组实现每种状态。这两个数组实际上是Goto函数：

```
int MatchArray[];
union GotoTable {
    unsigned GotoState;
    unsigned *BranchTable;
} GotoArray[];
```

为了确定Goto［state，c］的值，可以进行以下处理：

1. 如果MatchArray［state］==c，则下一种状态是GotoArray［state］.GotoState。

2. 如果MatchArray［state］==特殊值MULTI_WAY，则下一种状态是GotoArray［state］.BranchTable［c］。

3. 否则，下一种状态就是FAIL_STATE。

使用这种结构，只需为那些通过它们导出多个可能字符的状态实现完整的跳转表。不过，很容易修改代码，删除这种特性，并为每种状态使用BranchTable。其他函数（Output和Fail函数）已经被实现为简单的查找表，因为在任何情况下它们对内存都没有过多的需求。

在编写代码时，将输入和报告函数与实际的查找代码分隔开。对于查找代码应该使用的函数，将把指向这些函数的指针传递给它。选择这种方法可以将搜索引擎保持为单独的代码段，从而将其从I/O关注的问题中隔离出来。不过，这种方法的效率相对比较低；在代码的生产版本中，最好内联执行I/O操作。

该算法的时间复杂度的分析比较直观。为了建立表，需要遍历一次每个关键字，因而所用的时间与关键字的总长度成正比。这样，查找阶段只需要与要查找的文本大小成正比的时间。

4.5 用于正则表达式的字符串查找：grep

大多数操作系统的用户都熟悉**通配符**（wildcard）的概念。通配符是指代一系列字符的特殊字符。例如，当指代所有文件时，我们熟悉的MS-DOS命令行参数中将会包括字符*.*。在这个示例中，星号意指任意0个或多个字符集——确切地讲，是其后接着一个句点（它的意义不变）的

任意字符，句点后再接着任何其他的字符集。这样，*.*的结果就指代具有名称和扩展名（比如 command.com）的所有文件。

像上一个示例中的星号那样的字符被称为**元字符**（metacharacter）（拉丁语 meta 意指“改变”）。元字符具有特殊的含义，或者它们会改变普通字符的含义。例如，C 语言中的元字符是反斜杠“\”。在上下文中，反斜杠会改变下一个字符的含义。例如，\n 不是两个字符的组合，而是一个换行符号（ASCII 0x0D）。同样，\a 表示一个报警字符（ASCII0x07）。注意：在 C 语言中，与在依赖于元字符的所有情况中一样，存在一种转义机制，指示何时应该把元字符视作没有特殊含义的普通字符。在 C 语言中，普通斜杠写作\\。

如我们稍后将看到的，多年来，整个元字符字母表在不断扩展，主要是在 UNIX 领域内演进，其中元字符的使用非常普遍。使用元字符的表达式称为**正则表达式**（regular expression），究其原因，只能推测。涉及元字符的字符串查找通常由称为 grep（general regular-expression parser（通用正则表达式解析器）的首字母缩写词）的实用程序执行（因为 UNIX 的命令行是区分大小写的，并且 UNIX 用户避免使用大写字母，甚至像 grep 这样的首字母缩写词也总是使用小写字母）。grep 读取输入文件，然后打印出其中包含有与从命令行输入的正则表达式匹配的字符串的所有行。

我们现在来研究 grep 引擎：它如何接受和理解正则表达式（这就是我们刚才提到的解析器），以及它如何查找文本以找到正则表达式的匹配。不过，在开始之前，我们应该检查将在正则表达式中允许使用的元字符。表 4-3 列出了一组通用元字符以及它们的标准含义。

表 4-3 一些通用元字符及其含义

元 字 符	含 义
^	将表达式固定在行开始位置。例如，^From 将只从行开始位置开始查找 From
$	将表达式固定在行尾。例如，tops$ 将在行尾查找 tops。注意：$ 只指定位置；换行符不是字符串的一部分。参见关于^的注释
.	匹配除换行符之外的其他任何字符
?	指示一个可选的元素。也就是说，前面的正则表达式元素可以出现 0 次或 1 次。例如，tom? e 将同时查找 toe 和 tome。多大的区别啊
*	指示前面的元素可以重复 0 次或多次。它类似于?，只不过? 被限制于最多只出现一次，而 * 可以连续出现许多次
+	其工作方式与 * 相同，只不过它将匹配一个出现一次或多次的元素。它类似于?，只不过? 被限制于元素最多出现一次
[]	表示一个字符类别。例如，to［klmn］e 将查找单词 toke、tole、tome 和 tone。可以使用连字符作为简写形式。在前面的示例中，to［k-n］e 将指示相同的查找字符串。可以通过简单地并列一些范围在一组方括号内指定多个范围。例如，如果查找一个 C 程序，可以通过 0x［0-9a-fA-F］查找十六进制的常量。如果在范围开头加上^，则意指除了范围中指定的那些字符之外的所有字符：［^0-9］将匹配除了数字之外的所有其他字符
\	转义字符，用于消除以前列出的字符的任何特殊意义。不过，它也会导致某些转义序列，比如 C 语言中的那些转义序列

另请注意表 4-4 中列出的转义序列，可以在正则表达式中自由地使用它们。

表 4-4 用于一些正则表达式的转义序列

转义序列	描 述
\b	退格符
\e	ASCII 转义字符
\f	换页符
\n	换行符
\r	回车符
\s	空格
\t	制表符
\DDD	由 1~3 个八进制数字构成的数（D 表示一位数字）
\xDDD	由 2~3 个十六进制数字构成的数（D 表示一位数字）
\xCC	控制码（C 表示一个字符），例如，ctrl-c

grep 的这种实现中的策略是：使用蛮力查找来检测正则表达式指定的字符串的所有实例。它不会减慢速度以从头开始解释比较函数内的正则表达式，而是把正则表达式简化为**模式字符串**（pattern string），然后由比较函数解释它。模式字符串是普通的 ASCII 字符（其 ASCII 值小于 0x7F）和特殊字符或者表示某类动作的**标记**（token）的混合。表 4-5 中列出了一些标记，它们的值都大于 0x80，并且都具有符号名称（每个标记中的前缀 MT_是指元字符标记）。不带标记的模式字符串由比较函数处理，就像普通的比较一样：模式中的每个字符都必须与输入字符串中对应位置上的字符匹配。

表 4-5 元字符及其对应的标记

元 字 符	含 义	标 记	标 记 值
^	行开始位置	MT_BOL	0x80 | '^'
$	行尾	MT_EOL	0x80 | '$'
[	字符类别开始处	MT_CCL	0x80 | '['
]	字符类别末尾		
^（在一个类别内）	如果是第一个字符，则对字符类别取反		
*	克林闭包（匹配出现 0 次或多次的元素）	MT_CLOSE	0x80 | '*'
+	正闭包（匹配出现 1 次或多次的元素）	MT_PCLOSE	0x80 | '+'
?	可选（匹配出现 0 次或 1 次的元素）	MT_OPT	0x80 | '?'

不过，将以不同的方式处理这些标记。例如，如果在模式字符串中找到 MT_ANY 标记，比较函数将成功地匹配输入字符串中的任何字符。当比较函数找到 MT_PCLOSE 标记时，它会向前移动模式指针，使其指向下一个标记，然后在输入字符串中寻找下一个标记的一个或多个重复项。

可以通过以下模式来表示像 a.*[0-9]x+$ 这样的正则表达式：'a' MT_CLOSE MT_ANY MT_CCL <bit_map> MT_CLOSE 'x' MT_EOL。注意：在模式字符串中，MT_CLOSE、MT_PCLOSE 和 MT_OPT 标记位于它们的操作数之前；而在输入正则表达式中，它们则位于其操作数之后。

不是非常直观的唯一标记是 MT_CCL。它用于标记一组 17 个字节，用于表示模式字符串中的字符类别。第一个字节是 MT_CCL 标记，后续的 16 个字节是位图，位图中的每一位对应一个

ASCII 字符。位 97 表示存在一个 a（ASCII 码中的 a 对应的十进制值是 97），98 表示 b，依此类推。

程序清单 4-5（minigrep. c）中展示了将正则表达式翻译成模式模板的代码。

程序清单 4-5　匹配正则表达式的 grep 引擎

```
/*--- minigrep.c ------------------------- Listing 4-5 --------
 *  Find substrings represented by regular expressions
 *  in some other string or text
 *
 *  #define DRIVER to have a command-line version compiled.
 *-------------------------------------------------------------*/

#define DRIVER 1    /* Compile the main driver */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#ifndef max
#define max(a,b) ((a) > (b) ? (a) : (b))
#endif

/*-------------------------------------------------
 * If DEBUG is defined, D(printf("hello");) expands
 * to printf("hello"). If DEBUG isn't defined, D(...)
 * is expanded to an empty string, effectively
 * removing the printf() statement from input
 *-------------------------------------------------*/
#ifdef DEBUG
#define D(x) x
#else
#define D(x)
#endif

/*--- Metacharacters in the input:  ---*/
#define BOL     '^'     /* start-of-line anchor                */
#define EOL     '$'     /* end-of-line anchor                  */
#define ANY     '.'     /* matches any character               */
#define CCL     '['     /* start a character class             */
#define CCLEND  ']'     /* end a character class               */
#define NCCL    '^'     /* negates character class if 1st char */
#define CLOSURE '*'     /* Kleene closure (matches 0 or more)  */
#define PCLOSE  '+'     /* Positive closure (1 or more)        */
#define OPT     '?'     /* Optional closure (0 or 1)           */

/*--- These are the tokens representing metacharacters -------*/

#define    MT_BOL      ( 0x80 | '^' )
#define    MT_EOL      ( 0x80 | '$' )
#define    MT_ANY      ( 0x80 | '.' )
#define    MT_CCL      ( 0x80 | '[' )
#define    MT_OPT      ( 0x80 | '?' )
#define    MT_CLOSE    ( 0x80 | '*' )
#define    MT_PCLOSE   ( 0x80 | '+' )
```

```
/*--- pattern strings are unsigned char ---*/
typedef unsigned char pattern;

/* maximum number of pattern elements. Remember that
 * character classes require 17 pattern elements.
 */
#define MAXPAT  128

/*--- need this many bytes for character-class bitmap ---*/
#define MAPSIZE 16

/*  Advance a pointer into the pattern template
 *  to the next pattern element, this is a +1 for
 *  all pattern elements but MT_CCL, where you
 *  to skip past both the MT_CCL character and the
 *  bitmap that follows that character
 */

#define ADVANCE(pat) (pat += (*pat == MT_CCL) ? (MAPSIZE+1) : 1)

/* Bitmap functions. Set bit b in the map and */
/* test bit b to see if it was set previously */
#define SETBIT(b,map) ( \
        (map)[((b) & 0x7f) >>3] |= (1<< ((b) & 0x07)) )
#define TSTBIT(b,map) ( \
        (map)[((b) & 0x7f) >>3] &  (1<< ((b) & 0x07)) )

#define ISHEXDIGIT(x) isxdigit(x)

#define ISOCTDIGIT(x) ('0'<=(x) && (x)<='7')


/*--- Return values from PatternError() and MakePattern() -----*/
#define E_NONE       0
#define E_ILLEGAL    1
#define E_NOMEM      2
#define E_PAT        3

static int Error = E_NONE;      /*--- error flag, like errno ---*/

/*--- Declare the functions we wiil create and use ------------*/
static  pattern * DoCCL      ( pattern *, unsigned char *);
static  int       DoEscapeSeq( char ** );
static  int       HexToBinary( int );
static  int       MatchOne   ( char **, pattern *, char * );
static  pattern * MakePattern( char * );
extern  char    * MatchString( char *,  pattern *, int );
static  int       OctToBinary( int );
static  char    * PatternCmp ( char *,  pattern *, char * );
extern  int       PatternErr ( void );


/*--- returns current error status ---*/
int PatternErr ( void )
{
    return ( Error );
}
```

```
/*-----------------------------------------------------------
 *  Make a pattern template from the regular-expression
 *  string pointed to by exp.
 *  Stop when '\0' or '\n' is found in exp.
 *  Return:  a pointer to the pattern template on success, NULL
 *           on failure (in which case, PatternErr()
 *           will return one of the following values:
 *
 *                E_ILLEGAL       Illegal input pattern.
 *                E_NOMEM         out of memory
 *                E_PAT           pattern too long.
 *-----------------------------------------------------------*/
static pattern *MakePattern ( char *exp )
{

    pattern *pat;  /* pattern template being assembled    */
    pattern *cur;  /* pointer to current pattern element  */
    pattern *prev; /* pointer to previous pattern element */

    pat   = NULL;
    Error = E_ILLEGAL;
    if( ! *exp || *exp == '\n' )
        return ( pat );

    if( *exp == CLOSURE || *exp == PCLOSE || *exp == OPT )
        return ( pat );

    /* get pattern buffer  */
    Error = E_NOMEM;
    if (( pat = (pattern *) malloc ( MAXPAT )) == NULL )
        return ( pat );

    /* zero the buffer if debugging */
    D( memset ( pat, 0, MAXPAT); )

    prev  = cur = pat;
    Error = E_PAT;

    while ( *exp  &&  *exp != '\n' )
    {
        if ( cur >= &pat[MAXPAT-1] )
        {
            free ( pat );
            return ( NULL );
        }

        switch ( *exp )
        {
            case ANY:
                *cur = MT_ANY;
                prev = cur++;
                ++exp;
                break;

            case BOL:
                *cur = ( cur == pat ) ? MT_BOL : *exp;
                prev = cur++;
```

```
                ++exp;
                break;

            case EOL:
            *cur = ( !exp[1] || exp[1]=='\n' ) ?
                        MT_EOL : *exp;
            prev = cur++;
            ++exp;
            break;

        case CCL:
            if ((( cur - pat ) + MAPSIZE ) >= MAXPAT )
            {
                /* need more room for bitmap */

                free ( pat );
                return ( NULL );
            }

            prev   = cur;
            *cur++ = MT_CCL;
            exp    = DoCCL ( cur, exp );
            cur   += MAPSIZE ;
            break;

        case OPT:
        case CLOSURE:
        case PCLOSE:
            switch ( *prev )
            {
                case MT_BOL:
                case MT_EOL:
                case MT_OPT:
                case MT_PCLOSE:
                case MT_CLOSE:
                    free ( pat );
                    return ( NULL );
            }

            memmove ( prev+1, prev, cur-prev );
            *prev = ( *exp == OPT ) ? MT_OPT :
                    ( *exp == PCLOSE ) ? MT_PCLOSE :
                    MT_CLOSE;
            ++cur;
            ++exp;
            break;

        default:
            prev   = cur;
            *cur++ = DoEscapeSeq ( &exp );
            break;
    }
}
    *cur  = '\0';
    Error = E_NONE;

    return ( pat );
```

```
}

/*---------------------------------------------------------------
 *  Set bits in the map corresponding to characters
 *  specified in the src character class.
 *--------------------------------------------------------------*/

static pattern *DoCCL ( pattern *map, pattern *src )
{
    int first, last, negative;
    pattern *start = src;

    ++src;                              /* skip past the [       */
    if( negative = ( *src == NCCL )) /* check for negative CCl*/
        ++src;
    start = src;                /* start of characters in class */
    memset ( map, 0, MAPSIZE );     /* bitmap initially empty */

    while ( *src && *src != CCLEND )
    {
        if ( *src != '-' )
        {
            /* Use temp variable to avoid macro side effects */

            first = DoEscapeSeq ( &src );
            SETBIT( first, map );
        }
        else
        if ( src == start  )
        {
            SETBIT( '-', map );       /* literal dash at      */
            ++src;                    /* start or end.        */
        }
        else
        {
            ++src;           /* skip to end-of-sequence char */
            if ( *src < src[-2] )
            {
                first = *src;
                last  = src[-2];
            }
            else
            {
                first = src[-2];
                 last  = *src;
             }

             while ( ++first <= last )
                 SETBIT( first, map );
             src++;
         }
     }

     if ( *src == CCLEND )
         ++src;                   /* Skip CCLEND */

     if ( negative )
```

```
        for ( first = MAPSIZE; --first >= 0 ; )
            *map++ ^= ~0;       /* invert all bits */

    return ( src );
}

/*-------------------------------------------------------------
 *  Uses PatternCmp() to look for a match of pat anywhere in
 *  str using a brute-force search. str is a character string
 *  while pat is a pattern template made by MakePattern().
 *  Returns:
 *      o  NULL if no match was found.
 *      o  A pointer the last character satisfying the match
 *           if ret_endp is true.
 *      o  A pointer to the beginning of the matched string
 *           if ret_endp is false.
 *-------------------------------------------------------------*/

char *MatchString ( char *str, pattern *pat, int ret_endp )
{
    char  *start;
    char  *end = NULL;

    /*---------------------------------------------------------
    * This test lets you do MatchString(str, MakePattern(...),?);
    * without grave consequences if MakePattern() fails.
    *---------------------------------------------------------*/

    if ( !pat )
        return NULL;

    if ( !*str )
    {
        if (( *pat == MT_EOL ) ||
            ( *pat == MT_BOL &&
                ( !pat[1] || pat[1] == MT_EOL )))
                    end = str;
    }
    else
    {
       /*-----------------------------------------------
        * Do a brute-force search for substring, comparing
        * a pattern against the input string.
        *-----------------------------------------------*/
        start = str;
        while ( *str )
        {
            if( !( end = PatternCmp ( str, pat, start )))
                str++;
            else                /* successful match */
            {
                if ( !ret_endp )
                    end = str ;
                break;
            }
        }
    }
```

```
    return ( end );
}

/*-------------------------------------------------------------
* Like strcmp, but compares str against pat. Each element of
* str is compared with the template until either a mismatch is
* found or the end of the template is reached. In the former
* case, a 0 is returned; in the latter, a pointer into str
* (pointing to the last character in the matched pattern)
* is returned. strstart points at the first character in the
* string, which might not be the same thing as line if the
* search started in the middle of the string.
-------------------------------------------------------------*/
static char *PatternCmp ( char *str, pattern *pat, char *start )
{
    char *bocl,     /* beginning of closure string.         */
         *end;      /* return value: end-of-string pointer. */

    if ( !pat )             /* make sure pattern is valid   */
        return ( NULL );

    while ( *pat )
    {
        if ( *pat == MT_OPT )
        {
           /*-------------------------------------------------
            *  Zero or one matches. It doesn't matter if MatchOne
            *  fails---it will advance str past the character on
            *  success. Always advance the pattern past both
            *  the MT_OPT and the operand.
            *-------------------------------------------------*/

            MatchOne ( &str, ++pat, start );
            ADVANCE ( pat );
        }
        else
        if ( !( *pat == MT_CLOSE || *pat == MT_PCLOSE ))
        {
           /*-------------------------------------------------
            *  Do a simple match. Note that MatchOne() fails
            *  if there's still something in pat when we are
            *  at end of string.
            *-------------------------------------------------*/

            if ( !MatchOne ( &str, pat, start ))
                return NULL;

            ADVANCE ( pat );
        }
        else         /* process a Kleene or positive closure */
        {
            if ( *pat++ == MT_PCLOSE )   /* one match req'd */
                if ( !MatchOne ( &str, pat, start ))
                        return NULL;

            /* Match as many as possible, zero is OK */
```

功的案例。

（四）纵向做透，横向做广

纵向做透是指广告在运作时间上的连续性及长度的适宜性。横向做广，指同时使用两种或多种诉求点的广告在同时期内播放，将两者结合起来构成广告运动的一个战役。广告战役是一个系统工程。消费者对广告的消化和记忆是一个反复刺激，多次积累的过程。不同的广告诉求可以使不同的消费者产生共鸣。假若仅使用一种广告，采用一个诉求点，这样往往会丢掉一部分消费者。所以为了保证最大限度地吸引消费者，最大范围的影响消费者，最大“成功率”的抓住消费者，在广告运作上，要纵向做透，横向做广。

哈药六厂的“护彤”广告战役，就是三则广告同时播放。

其一：理性诉求：“护彤，专治儿童感冒、中西药结合、剂量小、退热快、不含PPA、不含咖啡因”。“护彤，哈药六厂”。

其二：“孩子感冒发烧，当妈妈最揪心了，自从有了护彤啊，我就放心了，它对孩子感冒引起的头疼、咽喉疼、咳嗽有很好的疗效”。“护彤，哈药六厂”。

其三：“护彤，含有牛黄，它专治儿童感冒引起的头疼、发烧、连我们单位有孩子的同事，都说护彤好使”。“护彤，哈药六厂”。

从广告原理上来说，每则广告只能有一个诉求点，“护彤”的广告策略，采取的是“要做就做深、做透”的手法。创作多则广告，在同一时期进行分别的诉求，既有纯理性的功能介绍，又使用“证言广告”的手法宣传，同时又有家庭生活场景的对话渲染气氛，可谓“多管齐下”制造“气氛”。

由上面介绍的广告的四个常用的策略不难看出，“投公众之所好”、“整体统一”、“于无声处”等原理，在广告中都有相应的应用。

【范例一】

浓情魅力非凡品质

拥有一百二十年历史的立邦漆，始终以创造人类与自然的和谐为己任，一贯积极倡导环境保护，不断美化和保护着人们的生活。在技术上，立邦漆更是以卓越的防腐、耐候及环保性能傲视群雄。立邦漆的产品系列广泛，包括建筑用漆、汽车漆、汽车表面处理剂、大型结构物件的重防腐漆、海事用漆、家电用漆和电脑部件用漆等。1992 年，立邦漆来到中国，成功地领导了内外墙乳胶漆的消费潮流。

立邦漆经国家环境分析测试中心检测：不含氯化物、铅、汞等重金属，对人体无害，立邦漆经中国预防医学科学院环境卫生监测所测试证实，属实际无毒级。

【简析】

这则广告，首先表明了其“以创造人类与自然的和谐”，“倡导环保为己任”的高贵品质；接着说明介绍了其广泛的产品系列；最后，又对产品的安全性作出权威论证。层次清晰、语言精练，其广告语——“浓情魅力、非凡品质”，在突出主旨的同时，又起到了以情动人的诉求效果。

【范例二】

比萨塔会倒吗？

闻名遐迩的意大利比萨塔建立于 1350 年，建成初期它就开始倾斜。于是，600 多年来不断有人在问：它会倒吗？它什么时候倒？它怎么还不倒？

```
 *  successful match. Closure is handled one level up by
 *  PatternCmp().
 *
 *  "start" points at the character at the left edge of the
 *  line. This might not be the same thing as *strp if the
 *  search is starting in the middle of the string. Note,
 *  an end-of-line anchor matches '\n' or '\0'.
 *-------------------------------------------------------------*/
static int MatchOne ( char **strp, pattern *pat, char *start )
{
     /* amount to advance *strp, -1 == error  */
     int advance = -1;

     switch ( *pat )
     {
       case MT_BOL:                  /* First char in string? */
         if ( *strp == start )       /* Only one star here.   */
              advance = 0;
         break;

       case MT_ANY:                /* . = anything but newline */
         if ( **strp != '\n' )
              advance = 1;
         break;

       case MT_EOL:
         if ( **strp == '\n'  ||  **strp == '\0' )
              advance = 0;
         break;

       case MT_CCL:
         if ( TSTBIT ( **strp, pat + 1 ))
              advance = 1;
         break;

       default:                      /* literal match */
         if ( **strp == *pat )
              advance = 1;
         break;
     }

     if ( advance > 0 )
          *strp += advance;

     return ( advance + 1 );
}

/*-------------------------------------------------------------*/
static int HexToBinary ( int c )
{
     /* Convert the hex digit represented by 'c' to an int. 'c'
      * must be one of: 0123456789abcdefABCDEF
      */
     return (isdigit(c) ? (c)-'0': ((toupper(c))-'A')+10)  & 0xf;
}

static int OctToBinary ( int c )
{
```

```
     /* Convert the hex digit represented by 'c' to an int. 'c'
      * must be a digit in the range '0'-'7'.
      */
     return ((( c ) - '0' )  &  0x7 );
}

/*-------------------------------------------------------------
 *  Map escape sequences into their equivalent symbols.
 *  Return the equivalent ASCII character. *s is advanced
 *  past the escape sequence. If no escape sequence is
 *  present, the current character is returned and
 *  the string is advanced by one.
 *
 *  The following escape sequences are recognized:
 *
 *   \b     backspace
 *   \f     formfeed
 *   \n     newline
 *   \r     carriage return
 *   \s     space
 *   \t     tab
 *   \e     ASCII ESC character ('\033')
 *   \DDD   number formed of 1-3 octal digits
 *   \xDDD  number formed of 1-3 hex digits
 *   \^C    C = any letter. Control code
 *-------------------------------------------------------------*/
static int DoEscapeSeq ( char **s )
{
     int rval;

     if ( **s != '\\' )
          rval = * (( *s )++ );
     else
     {
          ++( *s );                    /* Skip the \ */
          switch ( toupper ( **s ))
          {
            case '\0':  rval = '\\';            break;
            case 'B':   rval = '\b' ;           break;
            case 'F':   rval = '\f' ;           break;
            case 'N':   rval = '\n' ;           break;
            case 'R':   rval = '\r' ;           break;
            case 'S':   rval = ' '  ;           break;
            case 'T':   rval = '\t' ;           break;
            case 'E':   rval = '\033';          break;

            case '^':
              rval = *++( *s ) ;
              rval = toupper ( rval ) - '@' ;
              break;

            case 'X':
              rval = 0;
              ++( *s );
              if ( ISHEXDIGIT ( **s ))
              {
                   rval  = HexToBinary ( *(*s)++ );
```

```
                }
                if ( ISHEXDIGIT(**s) )
                {
                     rval <<= 4;
                     rval  |= HexToBinary ( *(*s)++ );
                }
                if ( ISHEXDIGIT(**s) )
                {
                     rval <<= 4;
                     rval  |= HexToBinary ( *(*s)++ );
                }
                --( *s );
                break;

              default:
                if( !ISOCTDIGIT ( **s ))
                     rval = **s;
                else
                {
                     ++ ( *s );
                     rval = OctToBinary ( *(*s)++ );
                     if ( ISOCTDIGIT ( **s ))
                     {
                          rval <<= 3;
                          rval  |= OctToBinary ( *(*s)++ );
                     }
                     if ( ISOCTDIGIT ( **s ))
                     {
                          rval <<= 3;
                          rval  |= OctToBinary ( *(*s)++ );
                     }
                     --( *s );
                }
                break;
            }
            ++ ( *s );
        }
        return rval;
}
/*------------------------------------------------------------
 * Driver is compiled so as to make this program a command-line
 * utility. In its absence, you have a grep engine that can be
 * called by other applications using the format below.
 *------------------------------------------------------------*/

#ifdef DRIVER

#include <limits.h>      /* to access CHAR_BIT */

int main ( int argc, char **argv )
{
     static pattern *pat;
     static FILE    *inp;
     static char     inp_buf[ 1024 ];

     /*--------------------------------------------------------
      *  This implementation is so dependent on 8-bit bytes
```

```
     *  that they must be checked for. Since 8-bit bytes are
     *  almost universal, this should not hinder portability.
     *-----------------------------------------------------*/

    if ( CHAR_BIT != 8 )  /* ANSI-defined as bits in a char */
    {
       fprintf ( stderr, "Error: Requires 8-bit bytes\n" );
       exit ( EXIT_FAILURE );
    }

    if ( argc < 2 || argv[1][0] == '-' )
    {
         fprintf ( stderr,
                      "Usage: minigrep reg_exp filename\n" );
         fprintf ( stderr,
                      "Usage: minigrep reg_exp < filename\n" );
         exit ( EXIT_FAILURE );
    }

    if( !( pat = MakePattern ( argv[1] )))
    {
         fprintf ( stderr, "Can't make expression template\n" );
         exit ( EXIT_FAILURE );
    }

    if ( argc == 2 )
         inp = stdin;
    else
    if( !( inp = fopen ( argv[2], "r" )))
    {
         perror ( argv[2] );
         exit ( EXIT_FAILURE );
    }

    while ( fgets ( inp_buf, sizeof(inp_buf), inp ))
            if ( MatchString ( inp_buf, pat, 0 ))
                 fputs ( inp_buf, stdout );

    return ( EXIT_SUCCESS );
}
#endif
```

程序清单开头的宏定义了出现在输入正则表达式中的元字符。这样，如果你不喜欢 UNIX 风格的正则表达式语法，可以很容易地修改它们。紧接在这些宏后面定义了一些标记，它们用于表示模式字符串中的元字符。注意：利用 typedef 命令将这些模式字符串的类型定义为 unsigned char。这对于默认类型为 signed char 的编译器是必需的，因为标记将设置它们的高位。如果不执行这一步，每次查看标记时，符号扩展位都会妨碍你。MAXPAT 宏会限制模式的大小（模式是 unsigned char 类型的 MAXPAT 元素的数组）。为了简化内存分配过程，我们使用固定大小的模式。当前默认每个模式包含 128 个字符是一个良好的折衷。这种大小的模式对于使用来说是足够大的，而又不会因为存在未使用的模式元素而浪费太多的内存。

宏 ADVANCE（p）向前移动指针 p 经过当前模式元素。这个指针增量可以用于除字符类别之外的所有对象，在字符类别中，对于 MT_CCL 标记，必须通过位图的大小（MAPSIZE）加 1 来前

移指针。注意：宏 ADVANCE( ) 具有明显的副作用。它还会评估 p 的递增值，因此如果你喜欢也可以将其称为 * ADVANCE （p)。可以将其视作等价于 * + +p。通过紧接在 ADVANCE 之后定义的 SETBIT( ) 和 TSTBIT( ) 这两个宏来操纵位图。这两个宏之间的唯一区别是 SETBIT( ) 中的一个运算符 （ | =)。该宏的前半部分把保存我们感兴趣的位的数组元素隔开。可以稍微简化一下它，如下所示：

```
map[ (b & 0x7F) >>3 ]
```

&0x7F 确保请求的位数 （b) 在范围之内， > >3 等价于一个整数除以 8；每个数组元素都有 8 位 （我们的实现需要一个 8 位字节。注意在 main( ) 驱动程序中对此进行的测试)。在大多数编译器上，移位运算比除法运算更高效，因此今天的大多数优化器会将除法运算尽可能地替换为移位运算。该宏的第二部分是创建一个掩码。在该掩码中，除了其所在位置对应于我们正在寻找的位的那个 1 位之外，其他所有位都将为 0。可以利用以下代码完成这种定位：

```
1 << (b & 0x07)
```

b &0x07 （等价于 b%8，但效率更高）计算出从当前数组元素的开始处到位 b 的偏移量。通过将数字 1 左移许多位来创建掩码。通过将掩码与适当的数组元素进行“或”运算，可以将其中一个位设置为 1。通过将掩码与数组元素逐位进行“与”运算，来检测某个位是否为真 （1)。

在完成了所有的预备性工作后，我们最后分析一下实际代码。模式是由 MakePattern 函数创建的。它将返回模式模板 （它位于 malloc( ) 分配的内存中)，如果有问题，则返回 NULL。在后一种情况下，可以调用 PatternErr( ) 查明是什么出错。在 MakePattern( ) 的代码之前定义了可能的错误。

为 OPT、CLOSURE 和 PCLOSE 调用 memmove( ) 是这个函数中唯一有点技巧性的代码。问题是：在正则表达式中，*、? 和 + 标记接在它们相应的表达式之后，而在编码形式中，它们需要放在相应的表达式之前。因此，需要将前一种模式右移一个位置，从而使 MT_CLOSE、MT_PCLOSE 和 MT_OPT 标记可以位于它们的操作数之前 （这将使比较函数的工作变得更容易)。这里技巧是使用 memmove( ) 而不是 memcpy( )。在这两个字符串复制函数中，只有 memmove( ) 可以处理源字符串和目标字符串重叠的情况。DoCCL( ) 用于设置字符类别的位图。组成 MakePattern( ) 的大 switch 语句中的下一个和最后一个 case 用于处理非元字符。通过调用 DoEscapeseq( ) 来执行该任务。这个例程将同时处理普通字符和转义序列。注意：将给它传递一个指针，指向字符串指针。这允许它返回字符串指针所指向的字符或转义序列的 ASCII 值，以及将字符串指针移过该字符或转义序列。返回到实际进行模式匹配的代码中来，最高级的函数如下：

```
char *MatchString( char *str, pattern *pat, int ret_endp );
```

传递给它的参数分别是：指向由 MakePattern( ) 函数生成的模式 （pat) 的指针、要查找的字符串 （str)，以及一个标志 （ret_endp，其含义为“返回终点 （return endpoint)”)，用于指示返回的指针是应该指向匹配字符串的开头还是末尾 （如果找到匹配字符串的话)。例如：

```
MatchString ( "1234567890", MakePattern ( "4[0-9]*7" ), 0 );
```

返回一个指向 4 的指针，而

```
MatchString ( "1234567890", MakePattern ( "4[0-9]*7" ), 1 );
```

则返回一个指向7的指针。注意：当模式字符串只由BOL或EOL组成并且输入字符串为空或者只包含单个换行符时，将返回成功状态。在这种情况下，将返回指向终止符‘\n’或‘\0’（无论哪个终止符位于字符串的最左边）的指针。这种行为允许正确地处理表达式^$。如果MatchString()不能找到一个匹配，它就会返回NULL。注意：该函数使用简单的蛮力方法。

我们感兴趣的最后一个函数是*PatternCmp（char * str, pattern *pat, char * start)。它是实际的比较函数。如果有一个匹配子串，它将返回一个指向其末尾的指针（由于str必然指向子串的开头，因此无需返回该值)。起始参数仅用于处理行首锚点（^)。注意：在字符串的实际起始位置，str可以指向字符串中的任意位置，因此它不能用于此目的。

PatternCmp()函数中的一个令人好奇之处是对PatternCmp()函数的递归调用。闭包（*和+）处理需要这种调用，以便使用一种贪心算法，用于找出与正则表达式匹配的最长字符串（还有一种非贪心算法，用于找出与表达式匹配的第一个字符串，但这种算法在实践中往往用处不大)。可以借助一个例子说明该问题：在像educated turnips这样的字符串中查找与表达式[a-z]*ed匹配的子串（所有单词都以ed结尾)。非贪心算法将在第一个ed处停止；而贪心算法将获取第一个完整的单词。这里的问题是：尾随的e和d是字符类别[a-z]的合法成员。因此，闭包处理在分析此字符类别时将会“吸收”这个ed。通过调用PatternCmp()函数的循环来校正这个问题。在完成闭包处理时（当找到一个不会将操作数匹配到闭包运算符的字符时)，PatternCmp()将尝试利用一次递归调用，将输入字符串的余下部分与正则表达式的余下部分进行匹配。如果这种尝试失败，PatternCmp()将把输入字符串末尾的指针倒退一个位置并再次尝试，然后以这种方式继续，直至它倒退到与闭包匹配的子串开始处。在当前的示例中，初始闭包处理终止时，str将指向educated中的第二个d，并且模式指针（pat）指向模式中的e。PatternCmp()的递归调用失败，因此将str倒退一个位置，使之指向educated中最右边的e。由于这个调用也失败了，算法将再次后退一个位置，并最终成功执行。在这个示例中，bocl指向educated中的第一个e，并且它确保不会后退到超过子串的左边界。

在实现类似于grep的小过滤器的程序清单末尾有一个小驱动程序。它的使用方法如下：

```
minigrep  regular-expression [optional-file]
```

如果遗漏了[optional-file]，就会使用标准输入；否则，将指定的文件作为输入。仅当行中包含作为第二个参数传递的正则表达式的匹配时，才会把这样的行打印到输出。能够以与使用UNIX的grep实用程序或MS-DOS的find实用程序非常相似的方式使用这个程序。

不要被这些函数的简单性或理论上的低效率所蒙蔽。在使用正则表达式的大多数情况下，它们的表现完全令人满意。

4.6 近似字符串匹配技术

本章中迄今为止介绍的所有查找技术都适合于查找精确的字符串匹配。grep函数甚至会指定一组必须精确匹配的允许的字符串。不过，有时一个应用程序只需要执行近似的查找。之所以需要执行这种查找，往往是由于要查找的精确字符串是未知的，这个问题是本章中的余下3种算法的主题。我们将首先考虑一个字符串查找算法，它只需查找近似的匹配，并且可以指定可接受的近似程度。

一些应用程序需要近似但不精确的字符串匹配，包括将氨基酸或DNA碱基序列与其他序列作比较，以确定它们的同源性。虽然下面的实现使用的是文本短语，但是你将看到可以通过修改该算法，使之适合于任何类型的数据。

考虑一种模式P，以及要查找的文本T。P在T中的k**近似匹配**（k-approximate match）是指T的一个子串在P中最多有k个位置存在差别。这里的差别可能是以下三种差别中的任意一种：

1. P与T中对应的字符不同。
2. T中包含一个未出现在P中的字符。
3. P中包含一个未出现在T中的字符。

举例说明，假定T是compiler，P是cmpxleer。P是T的一个3近似匹配：

```
P=  c  m p x l e e r
       |   |   |
T=  c o m p i l e  r
```

这些失配说明了三种可能的差别。

可以通过计算P与T之间的最小差别的数组，找到k近似匹配。正式的表示法如下：

设P为字符串p [1] p [2] ... p [n]。

设T为字符串t [1] t [2] ... t [n]。

计算差别数组D [i, j]，作为p [1] ... p [n] 与以t [j] 结尾的T的子串之间的最少差别数。通过逐步遵循以下三条规则来计算D。这些规则考虑了以前所述的全部三种可能的失配情况。

D [i, j] 是以下三个值中的最小值：

1. D [$i-1$, $j-1$] + diff (p [i], t [j])

其中：

diff (p [i], t [j]) =0 (如果p [i] = =t [j])

=1 (其他情况)

2. D [$i-1$, j] +1
3. D [i, $j-1$] +1

对于任何j，将在t [j] 处找到k近似匹配，满足D [n, j] < =k（记住：n是P的最后一个元素）。

例如，对于P = add和T = andadd，考虑D的计算，如图4-3所示。如图中所示的那样初始化外面的行和列。顶部一行全都为0，因为长度为0的P最多有0个位置不同于任何T。给左边一列赋予逐渐增大的整数，因为长度为n的P必定在n个位置不同于长度为0的T。现在填充数组D。

		a	n	d	a	d	d
	0	0	0	0	0	0	0
a	1						
d	2						
d	3						

图4-3　初始差别数组

对于p [1] =a和t [1] =a，要计算的第一个值是D [1, 1]（在这里的讨论中，与C语言中的数组不同，字符串索引是从1（而不是从0）开始）。我们考虑三个规则，以及D [1, 1] 上方、左边和左上方的D的元素的值：

规则1：由于p [1] = = t [1]，规则1的值将是D [0, 0] + 0或0。

规则2：D [0, 1] +1 = =0 +1 = =1

规则3：D [1, 0] +1 = =1 +1 = =2

取这三个值中的最小值赋予 D [i, j]，如图 4-4 所示。

它有助于把这种计算视作基于用 x 标记的三个值，如图 4-5 所示。

该计算将继续逐列进行，直到填满了数组为止，如图 4-6 所示。

```
      a n d a d d
   0 | 0 0 0 0 0 0
-----+-------------
a  1 | 0
d  2 |
d  3 |
```

图 4-4 第一个 D[i, j] 的计算结果

```
      a n d a d d
   x | x 0 0 0 0 0
-----+-------------
a  x | 0
d  2 |
d  3 |
```

图 4-5 图 4-4 中使用的值

```
      a n d a d d
   0 | 0 0 0 0 0 0
-----+-------------
a  1 | 0 1 1 0 1 1  <-- 第1行
d  2 | 1 1 1 1 0 1
d  3 | 2 2 1 2 1 0  <-- 第3行
```

图 4-6 完成的差别数组

现在将检查该数组的第 3 行的 k 近似匹配。有两个 1 近似匹配。第一个很容易：子串 and 中有一个字母不同于 add。第 3 行中的 1 显示了匹配子串的末尾，而第 1 行左上方的 0 则显示了子串的开头。第二个 1 近似匹配更有趣。重申一遍，底下一行中的 1 显示了终点，而左上方的 0 则显示了字符串 ad 的开头。这个子串不同于 add，它删除了一个 d。

另请注意 0 近似匹配。这是 andadd 中的 add 的精确匹配。

虽然很容易确定 k 近似匹配的终点（只需检查 D 的最后一行），但是确定匹配子串的开始位置可能比较复杂。考虑图 4-7 中 P 和 T 的值。可以在第 4 行中看到单个 1 近似匹配。但是向上扫描，在第 1 行左边显示了两个 0。实际上，字符串 ffod 和 fod 都是 food 的 1 近似匹配。

找到失配子串开始位置的最短小、代价最小的方法是：搜索从底下一行到顶部一行的路径。通过将差别数组视作地形图，可以最佳地理解算法。数字越大，海拔越高。设想右下角有一颗弹球，然后轻轻地抬起数组的那个角，弹球首先会从 a 滚动 b，如图 4-8 所示。

弹球现在甚至可以看到更低的层次，因此它会滚到 c，然后滚到 d，如图 4-9 所示。既然弹球已经到达顶部一行，它会停下来，从而标记出匹配子串的开头。

```
      f f o d
   0 | 0 0 0 0
-----+---------
f  1 | 0 0 1 1  <-- 第1行
o  2 | 1 1 0 1
o  3 | 2 2 1 1
d  4 | 3 3 2 1  <-- 第4行
```

图 4-7 查找匹配的开头

```
      f f o d
   0 | 0 0 0 0
-----+---------
f  1 | 0 0 1 1  <--第1行
o  2 | 1 1 0 1
o  3 | 2 2 b 1
d  4 | 3 3 2 a  <--第4行
```

图 4-8 向后查找匹配

```
      f f o d
   0 | 0 0 0 0
-----+---------
f  1 | 0 d 1 1  <-- 第1行
o  2 | 1 1 c 1
o  3 | 2 2 b 1
d  4 | 3 3 2 a  <-- 第4行
```

图 4-9 查找匹配的开头

计算弹球路径的一种简单方式是：维护一个单独的偏移量数组。这些偏移量将添加到当前列位置，以计算子串的开头。从技术上讲，通过遵循下面这些规则为 D [i, j] 的数组 O [i, j] 计算偏移量：

1. 如果 D [i-1, j-1] <D [i, j]，向上和向左查找

则 O [i, j] =O [i-1, j-1] -1

2. 否则，如果 D [i, j-1] <D [i, j]，向上查找

则 O [i, j] =O [i, j-1]

3. 否则，如果 D [$i-1$, j] < D [i, j]，向左查找

则 O [i, j] = O [$i-1$, j] -1

4. 否则，D [$i-1$, $j-1$] 必定等于 D [i, j]，因此：

设置 O [i, j] = O [$i-1$, $j-1$] -1

然后可以像下面这样计算 k 近似匹配 D [i, j] 的开头：

起始列 = j + O [i, j]

可以通过在圆括号中放置偏移量来扩展前面的计算。我们将从图 4-10 中所示的数组开始。将顶部一行的偏移量设置为 0，因为当弹球到达这一行时它总会停下来。将左边一列设置为 +1，因为从直觉上讲在这一列中结束的任何匹配都必须开始于右边的一列（尽管实际上不可能发生这种情况）。现在开始填写偏移量。检查标记有 a 的偏移量，我们看到位于 [2, 2] 处的弹球将直接向上滚动。因此，O[2,2] = O[1,2]或者0。

		f	f	o	d
	0	0	0	0	0
f	1 (+1)	0 (0)	0 (0)	1 (0)	1 (0)
o	2 (+1)	1 (a)	1 (?)	0 (?)	1 (?)
o	3 (+1)	2 (?)	2 (?)	1 (?)	1 (?)
d	4 (+1)	3 (?)	3 (?)	2 (?)	1 (?)

图 4-10　扩展偏移量

继续进行这种计算，填写图 4-11 中所示的数组。对于每一个偏移量列，可以基于其左边的偏移量列和差别数组的两个相关列来计算它，注意到这一点很重要。

		f	f	o	d
	0	0	0	0	0
f	1 (+1)	0 (0)	0 (0)	1 (0)	1 (0)
o	2 (+1)	1 (0)	1 (-1)	0 (-1)	1 (-2)
o	3 (+1)	2 (0)	2 (-1)	1 (-1)	1 (-2)
d	4 (+1)	3 (0)	3 (-1)	2 (-1)	1 (-2)

图 4-11　完成的偏移量

approx. c（程序清单 4-6）中的代码显示了如何实现该算法。为了节省内存，代码不会计算并保存整个数组 D。作为替代，由于任何给定列中的值都只依赖于紧接在其左边的列，因此只会在内存中保存这两个列。也会使用相同的技术来维护偏移量数组中的数据。

启用代码的方式是：首先通过调用 AppInit( ) 然后重复调用 AppNext( ) 来初始化其数据区域。测试驱动程序和注释演示了这些函数的语法。

程序清单 4-6　近似字符串匹配

```
/*--- approx.c ---------------------------- Listing 4-6 -------
 *
 * Approximate string search
```

```
 *
 * Usage:
 *  approx pattern text-to-search degree-of-approximation
 *
 *  if DRIVER is #defined, a test driver is compiled
 *-------------------------------------------------------------*/

#define DRIVER 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* local, static data */

static char *Text, *Pattern;/* pointers to search strings      */
static int Textloc;         /* current search position in Text */
static int Plen;            /* length of Pattern               */
static int Degree;          /* max degree of allowed mismatch  */
static int *Ldiff, *Rdiff;  /* dynamic difference arrays       */
static int *Loff,  *Roff;   /* used to calculate start of match*/

void AppInit ( char *pattern, char *text, int degree )
{
    int i;

    /* save parameters */
    Text    = text;
    Pattern = pattern;
    Degree  = degree;

    /* initialize */

    Plen   = strlen ( pattern );
    Ldiff  = (int *) malloc ( sizeof(int) * ( Plen + 1 ) * 4 );
    Rdiff  = Ldiff + Plen + 1;
    Loff   = Rdiff + Plen + 1;
    Roff   = Loff +  Plen + 1;
    for ( i = 0; i <= Plen; i++ )
    {
        Rdiff[i] = i;   /* initial values for right column */
        Roff[i]  = 1;
    }

    Textloc = -1; /* current offset into Text */
}

void AppNext ( char **start, char **end, int *howclose )
{
    int *temp, a, b, c, i;

    *start = NULL;
    while ( *start == NULL )  /* start computing columns   */
    {
        if ( Text[++Textloc] == '\0' ) /* no more text     */
            break;

        temp     = Rdiff;  /* move right column to left    */
```

```
Rdiff    = Ldiff;  /*  so that we can compute new  */
Ldiff    = temp;   /*  right-hand column.          */
Rdiff[0] = 0;      /* top (boundary) row           */

temp     = Roff;    /* and swap offset arrays, too */
Roff     = Loff;
Loff     = temp;
 Roff[1]  = 0;

 for ( i = 0; i < Plen; i++ )  /* run thru pattern  */
 {
     /* compute a, b, & c as the 3 adjacent cells...*/
     if ( Pattern[i] == Text[Textloc] )
         a = Ldiff[i];
     else
         a = Ldiff[i] + 1;
     b = Ldiff[i+1] + 1;
     c = Rdiff[i] + 1;

     /* ... now pick minimum ... */
     if ( b < a )
         a = b;
     if ( c < a )
         a = c;

     /* ... and store */
     Rdiff[i+1] = a;
 }
/*-------------------------------------------------------
 *  Now update offset array:
 *  the values in the offset arrays are added to the
 *  current location to determine the beginning of the
 *  mismatched substring. See text for details.
 *-------------------------------------------------------*/

 if ( Plen > 1 )
     for ( i = 2; i <= Plen; i++ )
     {
         if ( Ldiff[i-1] < Rdiff[i] )
              Roff[i] = Loff[i-1] - 1;
         else
         if ( Rdiff[i-1] < Rdiff[i] )
              Roff[i] = Roff[i-1];
         else
         if ( Ldiff[i] < Rdiff[i] )
             Roff[i] = Loff[i] - 1;
         else    /* Ldiff[i-1] == Rdiff[i] */
             Roff[i] = Loff[i-1] - 1;
     }

   /* now, do we have an approximate match? */

 if ( Rdiff[Plen] <= Degree )  /* indeed so! */
 {
     *end      = Text + Textloc;
     *start    = *end + Roff[Plen];
```

```
            *howclose = Rdiff[Plen];
        }
    }

    if ( start == NULL ) /* all done: free dynamic arrays */
        free ( Ldiff );
}

#ifdef DRIVER

int main ( int argc, char *argv[] )
{
    char *begin, *end;
    int howclose;

    if ( argc != 4 )
    {
        fprintf ( stderr,
                  "Usage is: approx pattern text degree\n" );
        return ( EXIT_FAILURE );
    }

    AppInit ( argv[1], argv[2], atoi ( argv[3] ));
    AppNext ( &begin, &end, &howclose );

    while ( begin != NULL )
    {
        printf ( "Degree %d: %.*s\n",
                 howclose, end-begin + 1, begin );
        AppNext ( &begin, &end, &howclose );
    }

    return ( EXIT_SUCCESS );
}
#endif
```

可以用多种方式修改基本的算法。一种有用的改变允许它识别相邻字符转置的特殊情况。可以通过向从中选择最小值的值列表中添加第四种情况来执行该任务：

$D[i-2,j-2]+\text{diff}(p[i-1],t[j])+\text{diff}(p[i],t[j-1])+1$

另一种可能的修改将给不同的错误赋予不同的加权。例如，附加在 a 与 p 之间的差别上的加权可能很大，而用于 m 与 n 之间差别的加权可以很小。这有助于查找代表简单键盘事故的失配。

4.7 语音比较：Soundex 算法

如果使用姓氏在数据库中查找一个人的名字，仅当你准确知道名字是如何拼写的时才会成功。不过，假设你正在运行一个航班预订系统，并且需要在不知道精确的拼写时查找名字。你将需要一种方法，基于名字的发音（即要查找的语音键）对它们进行编码。然后，查找将输出具有类似读音的名字的表项；例如，你将希望相同的查找可以发现 Dickson 和 Dixon。

M. K. Odell 和 R. C. Russell 为一个系统应用了两种模式（一种是在 1918 年，另一种是在 1922 年）来编码姓氏，使得读音相似的名字具有相同的代码。该系统（称为 Soundex）被广泛使用。它基于语言学家和速记员所熟悉的一个概念：可以只基于辅音字母来区分英语单词和名字。例如，

注意当只用辅音字母表示时这个句子中的最后几个单词（the last few words of this sentence when represented by consonants alone）：th lst fw wrds f ths sntnc whn rprsntd b cnsnnts alne。其含义很清晰，即使拼写看上去有点奇怪。

Soundex 读取一个名字，并给输入的字母赋予 7 个值之一，如表 4-6 所示。将要删除的字母将会被赋予值 0。这些包括所有的元音字母以及所有不发音的辅音字母。

表 4-6 赋予不同字母的 Soundex 代码

输入的字母	值	输入的字母	值
A E I O U H W Y	0	L	4
BFPV	1	MN	5
CGJKQSXZ	2	R	6
DT	3		

在赋予所有字母各自的数字之后，将删除 0 并把重复出现的相同数字减少到一位（实际上会删除成对出现的辅音字母以及重复出现的相同辅音字母组）。代码中使用了前三个余下的数字。如果只剩下三个以下的数字，将用空格填充代码（尽管有少数实现填充的是 0）。Soundex 代码包含要编码的单词或名字的首字母，其后接着三位数字的代码。这样，将把 Lincoln 转换为 L524。

当我们使用 Soundex 时，名字 Dickson 和 Dixon 具有相同的代码：D25。Smith、Smyth 和 Smythe 也将分享共同的 Soundex 代码。Soundex 方法可以有效地用于从数据库中检索名字。它通常会选择与查找名字模糊类似的多个名字，但是它倾向于不会跳过它应该查找的名字。它还具有快速和创建简洁代码的优点；其缺点是：必须精确知道查找名字的首字母。作为一种散列算法，Soundex 最多会产生 8918 个代码，它们倾向于围绕发音进行分组。

附带的程序 soundex. c（程序清单 4-7）实现了描述的算法。主例程 Soundex( ) 接受一个指向字符串的指针，然后它会遍历该字符串并对其进行编码。第二个指针 SoundexOut 说明了应该在什么位置复制 Soundex 代码。注意：Soundex( ) 返回一个整数：如果遇到一个无效的输入字符串，它就返回一个错误代码；或者如果正确地执行编码，则返回 1。

程序清单 4-7 Soundex 语音查找的实现

```
/*--- soundex.c ---------------------------- Listing 4-7 -------
*   Will generate the soundex code for a name or word.
*
*   Usage: int Soundex ( char *name, char *soundex_code )
*
*   if DRIVER is #defined a driver routine is compiled.
*-----------------------------------------------------------*/

#include <ctype.h>
#include <string.h>

#define DRIVER 1

int Soundex ( const char *str, char *soundex_out )
{
```

```
#define SOUNDEX_ERR 0
#define SOUNDEX_OK  1

    static const char table[] =

      /* ABCDEFGHIJKLMNOPQRSTUVWXYZ */
        "01230120022455012623010202";

    int count = 0;                  /* Valid code letters done */
    char this_char, prev_char;      /* Current and previous    */
    char code [5];                  /* will hold Soundex code  */
    strcpy ( code, "    " );        /* Initialize string       */

    this_char = toupper ( *str );   /* The first character is  */
    if ( isalpha ( this_char ))     /* preserved albeit as     */
    {                               /* uppercase.              */
        code[0] = this_char;
        count = 1;
        str += 1;
    }
    else
        return ( SOUNDEX_ERR );

    prev_char = ' ';                /* previous char empty now */

    while ( count < 4 && isalpha ( *str ))
    {
        char c;

        this_char = toupper ( *str );
        c = table[this_char - 'A'];

       /*-------------------------------------------------------
        *  Remove 0's and repeated sequences from output
        *  and then insert the next character into the string.
        *-------------------------------------------------------*/

        if ( c != prev_char && c != '0' )
        {
            code[count++] = c;
            prev_char = c;
        }
        str += 1;
    }

    /* Next, if it's neither '/0' or a letter, it's an error */

    if ( *str != '\0' && ! ( isalpha( *str )))
        return ( SOUNDEX_ERR );
    else
    {
        strcpy ( soundex_out, code );
        return ( SOUNDEX_OK );
    }
}

#ifdef DRIVER          /* Driver to demo soundex results */

#include <stdio.h>
```

```
#include <stdlib.h>

int main ( int argc, char *argv[] )
{
    char code[5];

    if ( argc != 2 )
    {
        fprintf ( stderr,
                  "Error! Usage: soundex word-to-code\n" );
        return ( EXIT_FAILURE );
    }

    if ( Soundex ( argv[1], code) == 0 )
        printf ("%s is not a valid name or word\n", argv[1] );
    else
        printf ( "Soundex for %s is %s\n", argv[1], code );

    return ( EXIT_SUCCESS );
}
#endif
```

在许多数据库和许多编目系统中都使用了 Soundex 算法。这一事实应该会阻止读者改变该算法。无疑，它不是完美无缺的，但是为了维持与使用 Soundex 代码的系统的兼容性，需要不加更改地使用这里介绍的方法。不过，偶尔会出现无需维持与其他系统的兼容性的情况。可以使用以下两种方法之一：第一种方法是通过缩小或加宽可能匹配的范围，试着修补 Soundex，寄望于改进它。[Celko89] 中展示了这种方法。更系统的方法是提出一种全新的、更全面的生成 Soundex 式代码的方法。可以在 Metaphone 中找到这种系统。

4.8 Metaphone：现代的 Soundex

Metaphone 不是基于字母的近似读音简单地对它们进行编码，而是使用一系列规则将一种读音转换为另一种。与 Soundex 不同，它不会将自身限制于辅音字母，而是检查称为**双元音**（diphthong）的字母组。

Metaphone 从查找键中删除非字母的字符，并将剩下的字母转换为大写形式，如表 4-7 所示。然后它会丢弃所有的元音字母，除非单词以一个元音字母开头，在这种情况下将会保留该元音字母。剩下的辅音字母都会被映射到它们的 Metaphone 编码，只有两个例外：字母 X 代表读音“sh”，数字 0 代表读音“th”。如果除 C 之外的任何辅音字母成对出现，则会删除第二个辅音字母。表 4-7 显示了这些转换。

表 4-7 Metaphone 执行的转换

从	转 换 到
GN-、KN-、PN-	N
AE-	E
WH-	H
WR-	R

（续）

从	转换到
X -	S
B	B（除非出现在-MB 中）
C	如果在 - CIA - 、 - CH - 中，则转换为 X 否则，如果在 - CI - 、 - CE - 、 - CY - 中，则转换为 S 否则，如果在 - SCI - 、 - SCE - 、 - SCY - 中，则删除它 否则，就转换为 K
D	如果在 - DGE - 、 - DGI - 或 - DGY - 中，就转换为 J 否则，就转换为 T
G	如果在 - GH 中并且不在 B - - GH、D - - GH、 - H - - GH、 - H - - - GH 中，就转换为 F 否则，如果在 - GNED、 - GN、 - DGE - 、 - DGI - 或 - DGY - 中，就删除它 否则，如果在 - GE - 、 - GI、 - GY 中并且不在 GG 中，就转换为 J 否则，就转换为 K
H	如果出现在元音字母之前并且不在 C、G、P、S、T 之后，就转换为 H
K	如果出现在 C 之后，就删除它；否则，就转换为 K
P	如果出现在 H 之前，就转换为 F；否则，就转换为 P
Q	K
S	如果在 - SIO - 或 - SIA - 中，就转换为 X；否则，就转换为 S
T	如果在 - TIA - 或 - TIO - 中，就转换为 X 否则，如果出现在 H 之前，就转换为 O 否则，就转换为 T
V	F
W	如果出现在元音字母之后，就转换为 W；否则，就删除它
X	KS
Y	除非后接元音字母，否则转换为 Y
Z	S
F、J、L、M、N、R	永远不会转换

可以很容易看到，与 Soundex 相比，Metaphone 会对单词执行更具体的转换。你将注意到这些转换非常贴近于英语的发音方式。例如，B 将不会被转换，除非它作为-MB 组合的一部分出现在单词末尾（比如 bomb 或 comb），在该组合中，它是不发音的。序列 PH 将被压缩成 F（这意味着将不会正确地找到 haphazard——但是在不创建所有单词的字典的情况下能够多么具体地制定规则呢?）。在所有情况下，Q 都会转换为 K（回忆可知：总是位于 Q 之后的 U 已经被删除了），这恰当地导致了 cake 与 quake 或者 kick 与 quick 等之间的匹配。与 Metaphone 不同，Soundex 根本不能处理出现特殊情况的可能性。它会把 PH 转换为 P 的读音，然后把它与所有 B 和 V 的读音集中到一起。Metaphone 更高的确切性所带来的结果是：Metaphone 在其查找中比 Soundex 更精确，因为单词被更好地编码。程序清单 4-8（meraphon. c）显示了表 4-7 中的规则是如何实现的。

程序清单 4-8 Metaphone——一种更精确的语音查找算法

```
/*--- metaphon.c ------------------------ Listing 4-8 ---------
*
* Usage: the calling function must pass three arguments:
*
*    char *word      - the word to be converted to a 'metaph'
*    char *result    - a MAXMETAPH+1 byte field for the result
*    int  flag       - a flag
*
* If flag is 1, then a code will be computed for word and
* stored in result. If flag is 0, then the function will compute
* a code for word and compare it with the code passed in
* result. It will return 0 for a match, else -1. The function
* will also return -1 if word is 0 bytes long.
*-------------------------------------------------------------*/

#define DRIVER 1

#include <ctype.h>

#define MAXMETAPH  4

int Metaphone ( const char *, char *, int );

/* Character coding array */
static char codes[26] =  {
   1,16,4,16,9,2,4,16,9,2,0,2,2,2,1,4,0,2,4,4,1,0,0,0,8,0
/* A  B C  D E F G  H I J K L M N O P Q R S T U V W X Y Z*/
   };

/*--- Macros to access character coding array -------------*/
#define ISVOWEL(x)  (codes[(x) - 'A'] & 1)      /* AEIOU */

    /* Following letters are not changed */
#define NOCHANGE(x) (codes[(x) - 'A'] & 2)      /* FJLMNR */

    /* These form diphthongs when preceding H */
#define AFFECTH(x) (codes[(x) - 'A'] & 4)       /* CGPST */

    /* These make C and G soft */
#define MAKESOFT(x) (codes[(x) - 'A'] & 8)      /* EIY */

    /* These prevent GH from becoming F */
#define NOGHTOF(x)  (codes[(x) - 'A'] & 16)     /* BDH */

int Metaphone ( const char *word, char *result, int flag )
{
    char *n, *n_start, *n_end; /* pointers to string */
    char *metaph, *metaph_end; /* pointers to metaph */
    char ntrans[32];           /* word with uppercase letters */
    char newm[8];              /* new metaph for comparison */
    int  KSflag;               /* state flag for X to KS */

    /*-----------------------------------------------------
     *  Copy word to internal buffer, dropping non-alphabetic
```

```
     *-------------------------------------------------------*/

    for ( n = ntrans + 1, n_end = ntrans + 30;
         *word && n < n_end; word++ )
            if ( isalpha ( *word ))
                *n++ = toupper ( *word );

    if ( n == ntrans + 1 )    /* return if 0 bytes */
        return -1;
    n_end = n;                /* set n_end to end of string */

    /* ntrans[0] will always be == 0 */
    *n++ = 0; *n = 0;         /* pad with nulls */
    n = ntrans + 1;           /* assign pointer to start */

    /* if doing a comparison, redirect pointers */
    if ( !flag )
    {
        metaph = result;
        result = newm;
    }
/*-------------------------------------------------------
 *  check for all prefixes:
 *            PN KN GN AE WR WH and X at start.
 *-------------------------------------------------------*/

 switch ( *n )
 {
     case 'P':
     case 'K':
     case 'G':
         if ( *( n + 1 ) == 'N')
              *n++ = 0;
         break;

     case 'A':
         if ( *( n + 1 ) == 'E')
              *n++ = 0;
         break;
     case 'W':
         if ( *( n + 1 ) == 'R' )
              *n++ = 0;
         else
         if ( *(n + 1) == 'H')
         {
             *( n + 1 ) = *n;
             *n++ = 0;
         }
         break;
     case 'X':
         *n = 'S';
         break;
 }

/*-------------------------------------------------------
 *  Now, loop step through string, stopping at end of string
 *  or when the computed metaph is MAXMETAPH characters long
```

```
    *-------------------------------------------------------*/

    KSflag = 0; /* state flag for KS translation */

    for ( metaph_end = result + MAXMETAPH, n_start = n;
          n <= n_end && result < metaph_end; n++ )
    {

        if ( KSflag )
        {
            KSflag = 0;
            *result++ = *n;
        }
    else
    {
        /* drop duplicates except for CC */
        if ( *( n - 1 ) == *n && *n != 'C' )
            continue;

        /* check for F J L M N R or first letter vowel */
        if ( NOCHANGE ( *n ) ||
                ( n == n_start && ISVOWEL ( *n )))
            *result++ = *n;
        else
        switch ( *n )
        {
        case 'B':        /* check for -MB */
            if ( n < n_end || *( n - 1 ) != 'M' )
                *result++ = *n;
            break;

        case 'C':   /* C = X ("sh" sound) in CH and CIA */
                    /*   = S in CE CI and CY            */
                    /*     dropped in SCI SCE SCY       */
                    /* else K                           */
            if ( *( n - 1 ) != 'S' ||
                !MAKESOFT ( *( n + 1 )))
            {
                if ( *( n + 1 ) == 'I' && *( n + 2 ) == 'A' )
                    *result++ = 'X';
                else
                if ( MAKESOFT ( *( n + 1 )))
                    *result++ = 'S';
                else
                if ( *( n + 1 ) == 'H' )
                    *result++ = (( n == n_start &&
                                !ISVOWEL ( *( n + 2 ))) ||
                                    *( n - 1 ) == 'S' ) ?
                                    (char)'K' : (char)'X';
                else
                    *result++ = 'K';
            }
            break;

        case 'D':  /* J before DGE, DGI, DGY, else T */
            *result++ =
```

```
            ( *( n + 1 ) == 'G' &&
                MAKESOFT ( *( n + 2 ))) ?
                    (char)'J' : (char)'T';
        break;
case 'G':   /* complicated, see table in text */
    if (( *( n + 1 ) != 'H' || ISVOWEL ( *( n + 2 )))
        && (
            *( n + 1 ) != 'N' ||
            (
                (n + 1) < n_end  &&
                (
                    *( n + 2 ) != 'E' ||
                    *( n + 3 ) != 'D'
                )
            )
        )
        && (
            *( n - 1 ) != 'D' ||
            !MAKESOFT ( *( n + 1 ))
            )
    )
            *result++ =
                ( MAKESOFT ( *( n  + 1 )) &&
                *( n + 2 ) != 'G' ) ?
                        (char)'J' : (char)'K';
    else
    if( *( n + 1 ) == 'H'   &&
        !NOGHTOF( *( n - 3 )) &&
        *( n - 4 ) != 'H')
            *result++ = 'F';
    break;

case 'H':   /* H if before a vowel and not after */
            /* C, G, P, S, T */
    if ( !AFFECTH ( *( n - 1 )) &&
        ( !ISVOWEL ( *( n - 1 )) ||
        ISVOWEL ( *( n + 1 ))))
                *result++ = 'H';
    break;

case 'K':    /* K = K, except dropped after C */
    if ( *( n - 1 ) != 'C')
        *result++ = 'K';
    break;

case 'P':    /* PH = F, else P = P */
    *result++ = *( n +  1 ) == 'H'
            ? (char)'F' : (char)'P';
    break;
case 'Q':   /* Q = K (U after Q is already gone */
    *result++ = 'K';
    break;
    case 'S':   /* SH, SIO, SIA = X ("sh" sound) */
        *result++ = ( *( n + 1 ) == 'H' ||
                    ( *(n  + 1) == 'I' &&
                        ( *( n + 2 ) == 'O' ||
                        *( n + 2 ) == 'A')))  ?
```

```
                                (char)'X' : (char)'S';
            break;

        case 'T':  /* TIO, TIA = X ("sh" sound) */
                   /* TH = 0, ("th" sound ) */
            if( *( n  + 1 ) == 'I' && ( *( n + 2 ) == 'O'
                || *( n + 2 ) == 'A') )
                    *result++ = 'X';
            else
            if ( *( n + 1 ) == 'H' )
                *result++ = '0';
            else
            if ( *( n + 1) != 'C' || *( n + 2 ) != 'H')
                *result++ = 'T';
            break;

        case 'V':      /* V = F */
            *result++ = 'F';
            break;

        case 'W':      /* only exist if a vowel follows */
        case 'Y':
                if ( ISVOWEL ( *( n + 1 )))
                    *result++ = *n;
            break;

        case 'X':      /* X = KS, except at start */
            if ( n == n_start )
                *result++ = 'S';
            else
            {
                *result++ = 'K'; /* insert K, then S */
                KSflag = 1; /* this flag will cause S to be
                        inserted on next pass thru loop */
            }
            break;

        case 'Z':
            *result++ = 'S';
            break;
        }
    }
            /* compare new metaph with old */
            if ( !flag && *( result - 1 ) !=
                    metaph[( result - newm ) - 1] )
                return -1;
        }

        /* If comparing, check that metaphs were equal length */
        if ( !flag && metaph[result - newm] )
            return -1;

        *result = 0; /* null-terminate return value */
        return 0;
    }

    #ifdef DRIVER   /* compile a driver to demo routine */
```

```
#include <stdio.h>
#include <stdlib.h>

main ( int argc, char *argv[] )
{
    char coded_word [MAXMETAPH+1];

    if ( argc != 2 )
    {
        fprintf ( stderr, "Usage: METAPHON word-to-be-coded\n" );
        return ( EXIT_FAILURE );
    }

    if ( Metaphone ( argv[1], coded_word, 1 ) == -1 )
    {
        fprintf ( stderr, "Invalid word/name to be coded\n" );
        return ( EXIT_FAILURE );
    }
    else
        printf ( "Metaphone for %s is %s\n",
                        argv[1], coded_word );

    return ( EXIT_SUCCESS );
}
#endif
```

与 Soundex 一样，使用 Metaphone 的目标是：生成一个较短的代码，反映要查找的单词（查找键）的发音。该代码的长度为 MAXMETAPH 个字符（默认为 4）。这个长度应该权衡两个因素：算法能够提供多大的空间以及匹配应该具有多大的精度。代码越长，将会找到的候选匹配就越少。这可能是一个优点，也可能是一个缺点，这依赖于你想在多大的范围内执行查找。如果你用一个拼写未必正确的名字开始执行查找，这样就会发现较长的 Metaphone 代码将是一个障碍。当拼写很可能正确时，反之亦然。适用于大多数目的的可选长度似乎是 4 或 5 个字符。该算法在到达了查找键的末尾或者生成了长度为 MAXMETAPH 个字符的代码之后，才会停止工作。

辅音字母转换表决不是完整无遗的。很容易扩展和细化它，其代价是增加 Metaphone 过程的开销。在执行改进时，会小幅提高精度。在执行更改前认真考虑以下一点很重要：你可能提供了例外情况的一个小子集，但是同时意外地排除了你没有考虑到的一组多得多的单词。可以安全地进行多种扩展。例如，以 X 开头的名字（以及少数几个单词）一般是利用前导 Z 的读音进行发音的。Metaphone 会把所有的 Z 都转换为 S，因此，在所有情况下安全的转换可能需要检查前导 X，并把它转换为 S。可以无限地进行更改和调整，每一次都会稍微增加一点处理时间。

不过，还是应该提醒一下。如果你使用 Metaphone 代码作为数据库字段，那么一旦你选定了自己喜欢的模式，就不要更改 Metaphone 算法，这是至关重要的。例如，如果你具有频繁按名字查找的客户列表，就可能想为 Metaphone 代码添加一个字段。这样，查找将只需要在这个字段上寻找匹配，而不用为每个客户计算 Metaphone。即使你以后对 Metaphone 算法稍做修改，也将遗漏你以前发现的精确匹配。如果你此时必须更改 Metaphone 算法，将需要在保存它的所有地方重新计算它。

4.9 选择技术

本章综述了用于查找和比较字符串的多种广泛的技术。许多技术只适用于特定的情况；其他

技术（比如蛮力查找和 Boyer-Moore 算法）则可以在任何时候必须执行字符串查找时使用。本章中介绍的算法的一个有趣的方面是：它们或多或少都依赖于蛮力查找。这一事实证明经过优化的蛮力查找具有可接受的速度。不过，在已知精确的查找字符串的情况下，Boyer-Moore 算法几乎总是优于蛮力方法。

这并不意味着应该用 Boyer-Moore 算法替换每一种字符串查找例程。考虑以下几点：

1. Boyer-Moore 算法需要很长的准备时间。如果文本不是足够长，也许不足以弥补在准备例程中花费的时间。

2. 如果查找模式很短，Boyer-Moore 算法的优点将无法体现；最大可能的跳转受限于模式的长度（记住：如表 4-1 所示，如果 PatLen = = 1，那么三个例程的典型情况的运行时间将变得相似）。Smit（1982）说明：对于长度为 2 或 3 的模式，Boyer-Moore 可能很容易变得比其他任何一种技术缓慢［Smit 1982］。

3. 当字母表很小时，Boyer-Moore 算法的性能会降级，因为较小的字母表意味着大多数字符很可能在查找模式中至少出现一次；因此，CharJump 将不能接受如此大的跳跃距离。小字母表的一个典型示例是：用于查找 DNA 序列的字母表。其中只有 4 个“字母”，任何现实的查找模式都非常有可能包括全部 4 个字母。这个问题的一种可能的解决方案是：组合相邻的字母，产生更大的字母表。例如，可以把相邻的 DNA“字母”组合成双“字母”集合，从而产生包含 16 个字母的字母表。

4. Boyer-Moore 算法需要能够回溯文本，并且多次扫描某些字符。虽然这对于其中所有的数据都在内存中的操作是可接受的，但是用于测试数据流的额外的缓冲复杂性可能超过算法的其他优点。

5. 最后，记住：特定实现的实际性能就是真正的测试。甚至最好的算法也可能实现得很差，或者可能由于外部限制而运行得很糟糕。表 4-2 中的分析着重考虑的是“时间与 n 成正比”，但是它并没有（并且不能）给出这个比例常量。

与这些告诫的情况相反，如果你正在大量的文本中查找长长的模式，并且该文本使用的是大字母表，那么 Boyer-Moore 技术通常是最好的选择。

4.10 资源和参考资料

4.10.1 通用参考资料

Baase, Sara. *Computer Algorithms*. Reading, MA: Addison-Wesley, 1988. 这是一本关于算法的通用参考著作，其中有关于查找（特别是利用 Boyer-Moore 和近似字符串匹配技术执行查找）的良好解释。

4.10.2 Boyer-Moore

Apostolico, A. 和 R. Giancarlo. “The Boyer-Moore-Galil String Searching Strategies Revisited.” *Siam J Comput*, Vol. 15, pp. 98-105, 1986. 这篇论文和 Galil 的论文解决的是改进 Boyer-Moore 算法的最坏情况下的性能这个问题。

Boyer, R. S. 和 J. S. Moore. “A Fast String Searching Algorithm.” *Communications of the ACM*, Vol. 20, pp. 762-772, 1977. 这是原始版本。MatchJump 表的开发后来是由 Knuth 和 Rytter 改进的。

Galil, Z. "On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm." *Communications of the ACM*, Vol. 22, pp. 505-508, 1979. 参见关于 Apostolico 的注释。

Knuth, Donald E. 、J. H. Morris 和 V. R. Pratt. "Fast Pattern Matching in Strings." *Siam J Comput*, Vol. 6, pp. 323-350, 1977. 这是原始的 Knuth-Morris-Pratt 论文。它详细讨论了 Boyer-Moore 和改进的 MatchJump 算法。

Rytter, W. "A Correct Preprocessing Algorithm for Boyer-Moore String Searching." *Siam J Computing*, Vol. 9, pp. 509-512, 1980. 这篇论文解释了如何修正 MatchJump 算法中的错误。

Smit, G. de V. "A Comparison of Three String Matching Algorithms." *Software-Practice and Experience*, Vol. 12, pp. 57-66, 1982. 这篇论文展示了 KMP、Boyer-Moore 和蛮力技术的有趣的比较。它还修正了 MatchJump 算法中的另一个细微的错误。在本章中的程序清单 4-3 中展示的代码基于这篇论文中展示的伪代码。

4.10.3 多字符串查找

Aho, Alfred V. 和 M. J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search." *Communication of the ACM*, Vol. 18, pp. 333-340, 1975.

Purdum, Jack. "Pattern Matching Alternatives: Theory vs. Practice." *Computer Language*, Vol. 4, No. II, pp. 34-44, 1987.

4.10.4 正则表达式查找

Holub, Allen. "GREP: Searching for Regular Expressions." C Gazette, Vol. 5, No. 1 (Autumn) 1990. 本章中展示的代码是通过有价值地改写这篇优秀的文章而得到的。可以通过 egrep 结合使用多种查找模式，egrep 是扩展了 grep 的实用程序。这个实用程序允许针对一种通配符模式或者针对相同查找中的另一种模式执行查找。有关如何以这种形式处理模式的信息，参考 Holub 的优秀书籍 Compiler Design in C (Englewood Cliffs, NJ: Prentice Hall, 1990) 中的第 2 章。该章包含关于这种查找所需状态表的权威讨论。顺便指出，该书被许多人（包括本书的作者）认为是所编写的关于编译器构造这个主题的最精美的图书。

4.10.5 近似字符串匹配

Hall, P. 和 G. Dowling. "Approximate String Matching." *ACM Computing Surveys*, Vol. 12, pp. 381-402, 1980. 这篇论文以及 Baase 的图书极好地介绍了近似字符串匹配。

4.10.6 Soundex 算法和 Metaphone 算法

Celko, Joe. "Optimized Soundex." *C Gazette*, Vol. 4, No. 2, pp. 29-32, 1989.

Parker, Gary. "A Better Phonetic Search." *C Gazette*, Vol. 5, No. 4 (June/July), 1990. 本章中展示的代码基于这篇文章中展示的公共领域的 C 版本。该版本是通通过修改和扩展 Lawrence Philips 在 *Computer Language* (Vol. 7, No. 12 [December], 1990) 中给出的原始 Metaphone 算法而成的。Philips 的版本是在 Pick BASIC 中编写的。

第5章 排 序

排序是许多应用程序的一个重要方面。虽然 ANSI C 的 qsort( ) 是一个可工作的基本排序工具，但它具有几个潜在的问题。首先，它只能对数据项的数组进行排序。虽然 qsort( ) 的定义非常灵活，足以允许它对结构的数组或者指向结构的指针的数组进行排序，但它不能直接用于对链表进行排序。其次，当按通常的方式实现时，qsort( ) 本质上并不是一种稳定的排序（如果你不熟悉“稳定”这个术语，可参见下一节的内容）。最后，当按通常的方式实现时，qsort( ) 的性能可能极大地依赖于提供给它进行排序的数据，通过要求它对已经排好序的链表进行排序可能会严重妨碍自然实现的性能。

尽管有这些缺点，qsort( ) 通常仍然是很好的选择，因为编译器库的版本将经过良好的测试、易于应用，并且通常速度非常快。在考虑本章中讨论的其他任何替代方法之前，应该先仔细考虑 qsort( )。本章将讨论快速排序算法的潜在缺点，该算法通常作为 qsort( ) 的基础，为其提供充足的信息，以便让你决定本地实现的表现有多好，以及它是否将满足你的特定问题。不过，在可以执行该任务之前，必须先考虑所有排序算法的基本性质。

5.1 排序的基本特征

所有的排序都是基于每个记录内包含的键对这些记录（或者指向记录的指针）重新进行排序。这个键通常是记录内的单个数据项，尽管你也可以通过考虑记录内的多个数据项来创建这个键。尽管键只是在排序期间使用的记录的一部分，但是本章中的讨论在提到键时通常把它看作记录一样。不过，如果不考虑所有的记录都具有键这一事实，排序将几乎在每个细节中都有所不同。

5.1.1 稳定性

稳定的排序是指能够维持要排序的记录中预先存在的顺序。例如，如果你有一个按交易日期排序的交易列表，那么按客户编号进行的第二次稳定排序将会保持每个客户的记录按交易日期进行排序。这通常是排序的一个重要性质，但是这个性质倾向于只存在于简单的排序中，在更高级的实现中往往会失去它。有可能按客户和日期对交易列表正确地进行不稳定的排序（可以通过使用组合客户编号和交易日期创建的单个键只排序一次），但是这种方法并不总是方便的。我们将考虑稳定的排序和不稳定的排序。

5.1.2 对哨兵的需求

一些排序实现起来很麻烦，除非提供包含两个称为**哨兵**（sentinel）的特殊键的虚拟记录。这两个哨兵键有时表示为 $-\infty$ 和 $+\infty$，并且保证它们分别小于和大于待排序记录中的任何键。提供这样的哨兵值通常不方便，也许甚至是不可能的。虽然用某种形式的记录计数代替哨兵通常是可

行的，但是这会给排序带来额外的开销。不像在简单的示例中，我们将不会考虑需要哨兵的排序。

5.1.3 对链表进行排序的能力

虽然几乎所有的排序算法都设计用于对象（或者指向对象的指针）的数组，但是可以轻松地修改一些排序算法，以对链表进行排序；对其他算法则不能这样做。如果把你的数据作为链表处理最合适，那么只把它转换为数组以对其进行排序将会很麻烦。我们将考虑两种非常适合于链表的排序方式。

5.1.4 输入的阶的相关性

排序算法的阶或 $O()$ 具有显著的差别，阶的范围包括从 N 到 N^2。估计排序算法的阶的额外问题是：阶可能依赖于待排序的数据。例如，一些排序算法的阶从正序记录（记录已经处于升序中）上的 N 到逆序记录（记录处于降序中）上的 N^2 不等。其他排序算法对于随机排序的记录运行的速度最快，而对于已经有序的记录则运行得最慢。最佳情况是：排序算法对于所有类型的输入都具有接近于 N 的阶。我们将研究具有各种行为的排序的示例。

5.1.5 对额外存储空间的需求

一些排序算法需要额外的临时存储空间来存储一个或更多待排序的记录。虽然要求其大小适合于一条记录的空间很常见并且很容易提供，但是对更多空间的要求却很难满足，并且我们不会考虑任何需要较多额外存储空间的排序算法。一种可能不是立即显而易见的额外空间是以递归方式实现的排序算法对栈空间的需求。虽然我们将考虑这样一种排序算法（快速排序），但是这种算法的最终实现将利用一种技术，它将把对栈的需求显著减小到可以很容易得到满足的水平。

5.1.6 内部排序技术与外部排序技术

另外还要考虑**内部**（internal）排序与**外部**（external）排序。在内部排序中，可以同时把待排序的所有记录都加载进主内存中。外部排序与之相反：在排序期间一部分记录将总是驻留在大容量存储设备上。虽然实现细节有所不同，但是所有外部排序的基本策略都是相同的。将把待排序的数据分成块，这些块非常小，足以放入主内存中，并且利用内部排序技术一次对其中一个块进行排序。然后合并这些块，产生最终排好序的输出。从历史观点上讲，外部排序的主要问题不是出现在合并记录的技术细节上，而是更乏味的困难任务。通常在缓慢的磁带机上对数据进行排序，很难避免如下情况：花在倒带上的时间要多于读、写数据所花费的时间。随着大容量 RAM 以及可以通过虚拟内存机制提供额外内存的操作系统日益增多，应该不会经常需要外部排序。当需要对大量数据进行排序时，把数据加载进 b 树（参见下一章）中通常是一种优秀的解决方案。因此，我们在本书中将不会讨论外部排序。

5.2 排序模型

我们将在本章中开发多种类型的排序，但是它们全都具有某些基本的特性。首先，我们的大多数排序例程将把一个指针数组排序成下面的简单结构：

```
typedef struct sElement {
    char *text;
} Element;
```

该结构本身可以填充任何类型的项目，但是出于我们的目的，我们将使用单个项目 text。每个结构都将是一条待排序的记录，text 将是记录的数据以及它的键。

我们将定义这些结构的包含 100 个元素的数据，如下：

```
Element *array[100];
```

并且由于我们是在用 C 语言编程，高效的编程要求我们使用如下的 C 语言约定：从 0 到 $n-1$ 对这样一个数组的 n 个元素进行编号。如果你学习过其他参考资料中描述的排序算法，就会知道 n 个编号的数组元素通常被编号为 1 到 n。如果转换到 C 语言，就要求你仔细调整算法，考虑到这种改变。

为了进一步一般化我们的排序例程，我们总是把比较例程的地址传递给排序例程，该比较例程接受指向两个结构的指针作为其参数。如果第一个参数“小于”第二个参数，比较例程必须返回一个负值；如果两个参数相等，就返回 0；如果第一个参数“大于”第二个参数，就返回一个正值。这种行为与 strcmp( ) 函数的行为完全一样，并且在我们的测试例程实际上将使用 strcmp( ) 作为比较函数。

在 sortsub. c（程序清单 5-1）、sortsub. h（程序清单 5-2）和 sorthdr. h（程序清单 5-3）中提供了一些支持例程，它们将允许我们以一种便利的方式测试多种面向数组的排序算法的行为。sortsub. c 和 sortsub. h 共同提供了对一些例程的访问，它们可以加载和显示数据项的数组。数据是从简单的文本文件中加载的，并在一行放置一个数据项。第三个文件 sorthdr. h 为待排序的结构提供了 typedef；它还提供了必须传递给每个排序例程的比较函数的原型。这个原型有点复杂，如下所示：

```
typedef int (*CompFunc) ( void *, void * );
```

该代码指出 CompFunc 是一个指向函数的指针，该函数接受两个指向 void 参数的指针，并返回一个 int。因此，下面是一个有效的比较函数：

```
int CFunc ( Element *L, Element *R )
{
    return ( strcmp ( L->text, R->text ));
}
```

注意：真实的函数接受指向两个真实结构的指针（而不是指向 void 的指针）作为其参数。不过，由于编译器将允许我们互换指向 void 的指针与指向真实对象的指针，在 typedef 中使用指向 void 的指针将允许我们为所有的比较函数使用相同的 typedef。

程序清单 5-1　sortsub. c 的代码

```
/*--- sortsub.c ------------------------- Listing 5-1 ---------
 * I/O subroutines for sorting routines
 *
 * Note: Loads data into an array
 *-------------------------------------------------------------*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "sorthdr.h"
#include "sortsub.h"

#define MAX_ITEM_SIZE 500 /* maximum length of a text item */

/*-------------------------------------------------------------
 * Load up to MaxItems text strings from FileName into an array
 * of pointers to the text strings. Returns number of items
 * in array or -1 for failure.
 *-------------------------------------------------------------*/

int LoadArray ( char *FileName,
                int MaxItems,
                Element ***Array)
{
    FILE *infile;
    char buffer[MAX_ITEM_SIZE], *s;
    int i;

    if (( infile = fopen ( FileName, "r" )) == NULL )
    {
        fprintf ( stderr, "Can't open file %s\n", FileName );
        return  ( -1 );
    }
*Array =
    (Element **) malloc ( sizeof(Element **) * MaxItems );

if ( *Array == NULL )
{
    fprintf ( stderr, "Can't allocate array of pointers\n" );
    return ( -1 );
}

i = 0;
while ( fgets ( buffer, MAX_ITEM_SIZE, infile ))
{
    if ( i >= MaxItems )  /* limit on # of items */
    {
        printf ( "Entire data file not loaded\n" );
        break;
    }

    /* trim trailing control characters */
    s = buffer + strlen ( buffer );
    while( iscntrl ( *s ))
        *s-- = 0;

    /* make space and store it */
    ( *Array )[i] = malloc ( sizeof ( Element ));
    if (( *Array )[i] == NULL )
    {
        fprintf ( stderr, "Can't get memory for data\n" );
        return ( -1 );
    }
```

```
    ( *Array )[i]->text = malloc ( strlen ( buffer ) + 1 );
    if (( *Array )[i]->text == NULL )
    {
        fprintf ( stderr, "Can't get memory for data\n" );
        return ( -1 );
    }
    strcpy (( *Array )[i++]->text, buffer );
}

/*
 * Special case: if the array contains only 1 item, and
 * the item is an empty string, return an empty array.
 */
if ( i == 1 && *(( *Array )[0]->text ) == 0 )
    i = 0;

fclose ( infile );
    return ( i );
}

/*--- Display array of items ---*/
void ShowArray ( Element **Array, int Items, CompFunc Compare )
{
    int i, sorted = 1, column = 1;

    for ( i = 0; i < Items; i++ )
    {
        if ( column > 61 )
        {
            printf ( "\n" );
            column = 1;
        }
        else  while (( column - 1 ) % 20 )
        {
            printf ( " " );
            column += 1;
        }
        printf ( "%3d: %s", i, Array[i]->text );
        column += 5 + strlen ( Array[i]->text );

        if ( i > 0 )
        {
            if ( Compare(Array[i-1], Array[i]) > 0 )
                sorted = 0;
        }
    }

    if ( sorted )
        printf ( "\n\nThe array is sorted.\n" );
    else
        printf ( "\n\nThe array is not sorted.\n" );
}
```

程序清单 5-2　sortsub. h 的代码

```
/*--- sortsub.h -------------------------- Listing 5-2 ------
 * Prototypes of functions in sortsub.c
```

```
 *-------------------------------------------------------*/

int LoadArray ( char *FileName, int MaxItems, Element ***Array );
void ShowArray ( Element **Array, int Items, CompFunc Compare );
```

程序清单5-3 sorthdr.h的代码

```
/*--- sorthdr.h --------------------------- Listing 5-3 -------
 * General definitions for array-oriented sort routines
 *-------------------------------------------------------*/

/*-------------------------------------------------
 * We sort an array of pointers to this structure. The
 * structure could contain anything you'd like: you only
 * need to define appropriate comparison functions.
 *-------------------------------------------------*/

typedef struct sElement {
    char *text;
}
    Element;

/*-------------------------------------------------
 * A type for the comparison function: a symbol with
 * type CompFunc is a pointer to a function that takes
 * two pointers to void and returns an int.
 *-------------------------------------------------*/

typedef int (*CompFunc) ( void *, void * );
```

虽然迄今为止人们提出了许多不同的排序算法，但是我们将只会讨论表5-1中列出的5种排序算法（注意：第6种排序算法与它们差别很大，在这个表中没有列出它。将在本章末尾讨论它）。之所以选择这组排序算法，是因为它们提供了从简单到复杂的广泛技术。虽然快速排序可能是这5种排序算法中最有用、最流行的算法，但是每种排序算法都有它自己的独特风格，并且值得学习。最后，我们还将修改两种排序算法（插入排序和快速排序），以对链表进行排序。如果所有这些排序算法都不是你正好想要的，也可以把第6章中描述的树算法视作排序算法。把它们用作排序算法时，只需将数据加载进树中，然后遍历它——即可产生排好序的输出。

表5-1 基本的排序特征

名称	阶	是否稳定	注释
数组排序			
冒泡排序	N^2	是	在对逆序记录排序时性能最差，但是在对几乎有序的记录排序时则具有线性的性能
插入排序	N^2	是	在对逆序记录排序时性能最差，但是在对几乎有序的记录排序时则具有线性的性能
希尔排序	$N^{1.25}$	否	该算法的阶对于初始记录顺序基本不敏感
快速排序	$N \lg N$	否	最坏情况是N^2，但是如果认真编写程序，极少发生这种情况
堆排序	$N \lg N$	否	最坏情况与平均情况相同。尽管堆排序具有与快速排序相同的阶，但它比快速排序要慢一些

（续）

名称	阶	是否稳定	注释
链表排序			
插入排序	N^2	是	在对几乎有序的记录排序时性能最差，在对逆序记录排序时性能最好
快速排序	$N \lg N$	不定	

5.2.1 冒泡排序

冒泡排序是许多人学习的前几种排序算法之一。它很大的优点是很简单。如果出于某种原因不能使用 qsort( )，并且需要一种可以快速编程的小排序算法，就可以选择冒泡排序。它对于几乎排好序的文件执行得非常快，但是它的常规性能相对较差。因此，我们主要将把它作为其他排序算法的范型来加以研究，这并不是因为它具有使之适合于应用程序的性能特征。

冒泡排序算法非常简单：连续地扫描待排序的记录，每扫描一次，都会移动最大的记录，使之更接近于顶部，就像气泡一样缓慢上升——从而得此名。由于每次扫描都会把一条记录置于它的最终正确的位置上，因此下一次扫描将不需要重新检查这条记录。用于对 n 个记录的数组进行冒泡排序的伪代码如下所示：

```
for limit = n - 1 to 1 begin
     for i = 0 to limit begin
          if array[i] > array[i+1] then
               swap array[i] and array[i+1]
     end
end
```

该算法的阶的分析很直观：它对记录执行 n 次遍历，每遍历一次它都会执行 $n-1$ 次比较，并且可能执行同样次数的交换。因此，该算法的运行时间将与 $n(n-1)$ 成正比，或者只是仅仅与 n^2 成正比。

在 bubble. c（程序清单 5-4）中实现了冒泡排序。该代码利用了如下事实：如果任何一次给定的遍历都没有执行任何交换，记录就是有序的，并且会终止排序。这个例程还显示了以前讨论的支持例程的使用。如果常量 DRIVER 是利用#defined 定义的，则会编译测试驱动程序以及简单的比较函数。这个测试驱动程序使用 sortsub. c 中的 LoadArray( ) 读取待排序的数据项的文件，然后使用 ShowArray( ) 显示结果。实际的排序函数的原型如下：

```
void BubbleSort(Element **Array, int N, CompFunc Compare);
```

也就是说，给排序例程传递一个指向指针数组的指针、一个计数，以及一个指向比较函数的指针。我们所有的面向数组的排序例程都将利用这个接口。

程序清单 5-4　冒泡排序的代码

```
/*--- bubble.c --------------------------- Listing 5-4 ---------
 * Bubble sort an array
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to sortsub.c (Listing 5-1).
 *-------------------------------------------------------------*/
```

```
#include "sorthdr.h"

#define DRIVER 1

void BubbleSort ( Element **Array, int N, CompFunc Compare )
{
    int limit;

    /* Make steadily shorter passes ... */
    for ( limit = N - 1; limit > 0; limit-- )
    {
        int j, swapped;

        /* On each pass, sweep largest element to end of array */
        swapped = 0;
        for ( j = 0; j < limit; j++ )
        {
            if ( Compare ( Array[j], Array[j+1] ) > 0 )
            {
                Element *temp;

                temp = Array[j];
                Array[j] = Array[j+1];
                Array[j+1] = temp;
                swapped = 1;
            }
        }

        if ( !swapped )
            break; /* if no swaps, we have finished */
    }
}

#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sortsub.h"

/*
 * A comparison function
 */
int Cfunc ( Element *L, Element *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main(int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;

    if ( argc != 2 && argc != 3 )
    {
        fprintf ( stderr, "Usage: bubble infile [maxitems]\n" );
        return;
    }
```

```
    if ( argc == 3 )
        Items = atoi ( argv[2] );

    if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
        return; /* Couldn't load file */

    BubbleSort ( Array, Items, (CompFunc) Cfunc );
    ShowArray ( Array, Items, (CompFunc) Cfunc );
}
#endif
```

冒泡排序中使用的比较函数也会在本章中的其他排序算法中使用。注意：它使用 strncmp( ) 只检查每条记录的 text 字段的前 5 个字符。对数据的这种有限检查允许测试每种排序算法的稳定性。例如，假设你创建了一个输入文件，其中包含下面这些记录：

```
00002 Am
00001 Unique key
00002 I
00002 stable?
00003 Another unique key
```

稳定的排序算法将使键为 00002 的记录保持有序，而不稳定的排序算法可能会打乱它们的顺序（注意：只是可能弄乱它们。不要基于单个测试就断定排序算法是稳定的还是不稳定。在断定排序算法是稳定的之前，必须仔细分析算法或者对它进行彻底的测试。此外，也可以参考前面所示的表 5-1）。

如表 5-2 所示，该算法对提供给它进行排序的数据很敏感。如果记录已经有序，算法的执行速度将非常快，并且具有与 n 成正比的阶。不过，对于逆序记录和随机排序的记录，该算法将非常缓慢。如前所述，冒泡排序的唯一优点是很简单。

表 5-2　排序例程的性能

排序例程	执行时间[1]	比较次数	交换次数	注释
数组排序				
bubble. c（冒泡排序）				
正序[2]	14	999	0	对逆序记录执行 $N(N-1)/2$ 次比较和交换
逆序[3]	6541	499500	499500	
随机[4]	6143	500000	500000	
insert. c（插入排序）				
正序	18	99	500	将两次移动统计为一次交换，忽略移动到 temp
逆序	5938	499500	250250	
随机	2766	250000	125000	
shell. c（希尔排序）				
正序	95	4821	4821	统计交换次数的方法与 insert. c 一样
逆序	159	9894	5406	
随机	204	14000	7300	

（续）

排序例程	执行时间1	比较次数	交换次数	注释
quick1. c（基本快速排序）				栈深度
正序	5988	501497	999	1000
逆序	6044	501498	999	1000
随机	163	12500	2400	23
quick2. c（对小记录组进行的三等分快速排序和插入排序）				栈深度
正序	108	8133	754	26
逆序	175	13149	2226	20
随机	141	11000	3100	17
quick3. c（删除了尾部递归的 quick2. c）				栈深度
正序	106	8133	754	8
逆序	172	13149	2226	8
随机	139	11000	3100	6
quick4. c（具有随机枢轴和指针的 quick3. c）				栈深度
正序	133	10900	840	6
逆序	154	12500	1900	6
随机	127	11000	3100	6
heap. c				
正序	178	10379	8925	
逆序	189	10769	9791	
随机	171	10308	8929	
链表排序				
linsert. c（用于链表的插入排序）				
正序	5686	499500	1000	
逆序	15	999	1000	
随机	2700	250000	1000	
lquick1. c（用于链表的快速排序，对小记录组使用插入排序，并且删除了尾部递归）				栈深度
正序	6216	499500	499500	1
逆序	6340	499500	499500	2
随机	147	12000	12000	4
lquick2. c（像 lquick1. c 一样，但是使用随机枢轴选择）				栈深度
正序	153	11045	9853	6
逆序	134	9531	9853	6
随机	150	11000	11000	6

注：对于这些测试，所有的程序都是使用 Borland C++ 编译器在启用了优化的情况下编译的。计时数据只是在排序例程上收集的；未包括加载数据和显示结果所需的时间

① 任意时间单位。

② 对 1000 条正序记录进行排序。

③ 对 1000 条逆序记录进行排序。

④ 该数据是对 5 组 1000 条随机排序的记录进行排序所得到的平均结果。

本章中展示的每个例程的性能都是用多种方式度量的。首先，通过使用性能测量和跟踪工具来度量例程的原始执行速度。虽然只能在完全相同的机器上才会再现这些计时的绝对值，相对值的比较也可能是有用的。不过，按速度进行的相对评级在其他情况下可能稍有差别。这些示范性的例程使用相对简单的比较函数，并且它们全都是对指向结构的指针的数组进行排序。如果使用更复杂的比较函数，将会提高执行比较的时间成本。此外，虽然交换两个指针的速度非常快，但是其他应用可能需要对结构的数组进行排序，交换多字节结构所花费的时间将明显长于交换指针的时间。因此，在对 1000 条记录的测试数据集进行排序时，比较表还会报告例程所执行的比较和交换的次数。虽然更快的算法总是具有较少的比较和交换次数，但是你可能需要考虑这两个动作对于你的特定情况的相对时间成本。

5.2.2　插入排序

插入排序的工作方式类似于桥牌选手在对一副牌进行排序时所做的那样：当选取后续的每一张牌时，就相对于以前选取的牌把它插入到正确的位置。要在实际中查看这个过程，考虑对表5-3中所示的 5 个字母进行排序的问题。第一个字母是 C，很容易把它插入到一组有序的字母中。在后续的遍历中，将把 A 插入在 C 的前面，并把 B 插入在 A 和 C 之间，依此类推。

表 5-3　插入排序

遍　历	一组有序的字母	一组无序的字母
0		C A B E D
1	C	A B E D
2	A C	B E D
3	A B C	E D
4	A B C E	D
5	A B C D E	

实现插入排序几乎与实现冒泡排序一样简单。不过，要考虑一个重要的设计要点：在选取每个新记录并准备把它插入到有序记录当中属于它的位置时，你是否会从前往后或者反过来查看一遍有序记录？这个看上去似乎微不足道的决定会对性能产生很大的影响，正确的答案确实依赖于无序记录的典型排序。首先，假设无序记录的链表通常处于逆序（或者几乎逆序）状态，比如（E、D、C、B、A）。在检查每个后续的记录时，它的值通常将小于一组有序记录中的大多数记录。在这种情况下，可以通过从前往后查找一组有序的记录把我们要做的工作减至最少。不过，如果我们期望无序的记录通常是有序（或者几乎有序）的，那么就可以期望一组无序记录中的每个后续记录中的值大于大多数其他的记录。在这种情况下，如果我们从后往前查找一组有序的记录，将把我们要做的工作减至最少。如果对我们的选择没有其他的约束，通常的方法是做如下假定：如果记录根本没有任何次序，那么它们将（几乎）是有序的。因此，当算法尝试插入每个后续记录时，它应该从后往前查找一组有序的记录。下面给出了伪代码：

```
for step = 1 to N begin
     temp = array[step]
     for i = step - 1 to 0 begin
          if array[i] > temp then
               array[i+1] = array[i]
```

```
            else
                break
        end
        array[i+1] = temp
    end
```

注意：伪代码中的方法是通过把一个元素复制到一个临时存储元素中，在数组的有序部分中创建一个“空位”。然后把这个空位向后移动，经过有序的记录，直至找到它的正确位置。此时，内层循环将会终止，并把新记录插入到该空位中。

insert.c（程序清单5-5中）给出了这个算法的实际实现。它与伪代码几乎完全一样。估计该算法的阶相对比较容易。不严格地讲，插入排序将会为输入的每个记录遍历一次数据，但是在执行这 n 次遍历中的每次遍历时，我们期望向后查找一半的有序记录，以便找到新记录的位置。在最坏情况下，我们将期望向后查找所有的记录。在任何一种情况下，运行时间都与 $n^2/2$ 成正比或者仅仅与 n^2 成正比。这种二次方程式的性能与我们为冒泡排序所预测的性能相似，并且表5-2中的数据确认了这些预测，其中的两种算法可以看成是对于逆序和随机顺序的记录具有相似的运行时间。不过，插入排序一般更快一些。这是由于它使用的交换次数和比较次数比冒泡排序使用的要少。使用较少的交换次数可以更容易看到：冒泡排序将在每一步中执行完全交换，因为它会把最大的记录“冒泡”到顶部；而插入排序执行的则是所谓的“半交换”，因为它是把空位移过有序的记录。插入排序将会避开一半的比较次数，因为它在插入新记录时通常只会查看一半的有序记录。与之相反，冒泡排序在每次遍历时总会把最大的记录冒泡到顶部。因此，插入排序的比较次数少于一半，因为在插入新的记录时，它只需要浏览一半已排好序的记录。相比之下，冒泡排序要一直冒泡，使最大的元素移动到顶端。因此，虽然这两种算法在平均情况下的运行速度都不快，但是插入排序确实有优势。

程序清单5-5 插入排序的代码

```
/*--- insert.c -------------------------- Listing 5-5 ---------
 * Insertion sort of an array
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to sortsub.c (Listing 5-1)
 *-------------------------------------------------------------*/

#include "sorthdr.h"

void InsertionSort ( Element **Array, int N, CompFunc Compare )
{
    int step;

    /* Look at 2nd thru Nth elements, putting each in place */
    for (step = 1; step < N; step++)
    {
        int i;
        Element *temp;

        /* Now, look to the left and find our spot */
        temp = Array[step];
        for ( i = step - 1; i >= 0; i-- )
        {
```

```
            if ( Compare(Array[i], temp ) > 0 )
            {
                /* Not there yet, so make room */
                Array[i+1] = Array[i];
            }
            else /* Found it! */
                break;
        }
        /* Now insert original value from Array[step] */
        Array[i+1] = temp;

    }
}

#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sortsub.h"

/*
 * A comparison function
 */
int Cfunc ( Element *L, Element *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;

    if ( argc != 2 && argc != 3 )
    {
        fprintf ( stderr, "Usage: insert infile [maxitems]\n" );
        return;
    }

    if ( argc == 3 )
        Items = atoi ( argv[2] );

    if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
        return; /* Couldn't load file */

    InsertionSort ( Array, Items, (CompFunc) Cfunc );
    ShowArray ( Array, Items, (CompFunc) Cfunc );
}
#endif
```

有趣的是，插入排序像冒泡排序一样，对于几乎有序的记录，它的运行时间是线性或者接近线性的。以后当优化快速排序算法时，这个重要的事实将给我们带来好处。

5.2.3　希尔排序

希尔排序是插入排序的一个灵巧的变体。它首先观察到插入排序之所以缓慢的原因是：它一次只把一个项目称过一组有序的记录。因此，如果记录离它们的最终位置很远，则只有在经过许

多次交换之后才能正确地定位它们。我们需要的方式是：在排序的早期阶段允许记录跳转较大的距离。针对这个问题的希尔排序的解决方案如表5-4所示。

表5-4 希尔排序的实际应用

0：	待排序的项目	E	F	B	G	H	D	C	A
1：	将待排序的项目视作4对项目，每一对中包含两个项目				G				A
				B				C	
			F				D		
		E				H			
2：	对每一对项目进行排序，给出这种顺序	E	D	B	A	H	F	C	G
3：	现在把它们视作两组项目，每一组中包含4个项目		D		A		F		G
		E		B		H		C	
4：	对每一组进行排序，给出这种顺序	B	A	C	D	E	F	H	G
5：	最后，将8个项目作为一组进行排序	A	B	C	D	E	F	G	H

基本思想很简单：把记录分成几个交替的组，使用我们现在熟悉的插入排序算法对每个组进行排序。在这个示例中，有8个待排序的记录。首先，把它们分成交替的4对，并对每一对进行排序。然后把记录分成两个交替的组，每一组包含4个记录，并再次进行排序。最后，对整组记录进行排序。

要理解这种方法为什么是对普通的插入排序的改进，考虑记录A的移动。当它从完全错误的位置开始移动时，在前两次移动的每一次移动中，它将跳过相距其最终位置一半的距离，只需三次交换即可到达其目标位置。

这种每隔h个记录选择一个记录以便反复再分为交替记录集的方式称为“h排序”，并且可以根据所用的h的序列来描述希尔排序。例如，表5-4中的记录是依次为h使用值4、2和1进行h排序的。使希尔排序工作的关键是：可以使用任何稳定递减的h序列，只要h的最后一个值是1即可。实际上，当h为1时，我们所得到的只是普通的插入排序。这最后一遍插入排序会确定所有记录的最终顺序；所有以前的排序都只有一个目标：产生上一遍排序可以在线性时间内排序的“几乎”有序的文件。

选择合适的h值序列需要做大量的工作。不过，尽管要做这个工作，“最佳的”序列仍然是未知的。程序清单5-6（shell.c）中所示的希尔排序的实现使用以下模式来确定h的值序列：

设$h_1 = 1$，n = 待排序的记录数

设$h_{s+1} = 3h_s + 1$，当$h > n/9$时停止

这种简单的计算会产生表现良好的h的值序列。可以使用其他方法产生稍好一点或者更糟糕的序列（参见本章末尾的“资源和参考资料”一节的内容，了解关于确定h值的方法的更多详细信息）。

程序清单5-6 希尔排序的代码

```
/*--- shell.c ---------------------------- Listing 5-6 ---------
 * Shell's sort on an array
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to sortsub.c (Listing 5-1)
 *-------------------------------------------------------------*/
```

```
#include "sorthdr.h"

#define DRIVER 1

void ShellSort ( Element **Array, int N, CompFunc Compare )
{
    int step, h;

    /* Find starting h */
    for ( h = 1; h <= N / 9; h = 3*h + 1 )
            ;

    /* Now loop thru successively smaller h's */
    for ( ; h > 0; h /= 3 )
    {
        /* Look at hth thru Nth elements */
        for ( step = h; step < N; step++ )
        {
            int i;
            Element *temp;
            /* Now, look to the left and find our spot */
            temp = Array[step];
            for ( i = step - h; i >= 0; i -= h )
            {
                if ( Compare ( temp, Array[i] ) < 0 )
                {
                    /* Not there yet, so make room */
                    Array[i + h] = Array[i];
                }
                else /* Found it! */
                    break;
            }
            /* Now insert original value from Array[step] */
            Array[i + h] = temp;
        }
    }
}

#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sortsub.h"
/*
 * A comparison function
 */
int Cfunc ( Element *L, Element *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}
void main ( int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;

    if ( argc != 2 && argc != 3 )
{
```

```
        fprintf ( stderr, "Usage: shell infile [maxitems]\n" );
        return;
    }

    if ( argc == 3 )
        Items = atoi ( argv[2] );

    if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
        return; /* Couldn't load file */
    ShellSort ( Array, Items, (CompFunc) Cfunc );
    ShowArray ( Array, Items, (CompFunc) Cfunc );
}
#endif
```

事实证明：分析这个算法的性能有些复杂，并且依赖于所选的 h 的序列。使用以前讨论的 h 的值序列的实现似乎具有大约 $N^{1.25}$ 的阶。在表 5-2 中可以找到该算法显著优于插入排序的示范。虽然希尔排序不具有像插入排序或冒泡排序处理已经有序的记录时那样引人注目的速度，但是在处理逆序或随机顺序的记录时它也不会遭受显著的速度降级。实际上，希尔排序的运行时间对输入数据相当不敏感。希尔排序唯一的负面特性是：它不是一种稳定的排序（参见表 5-1）。不过，综合起来讲，这些特性使得希尔排序成为几乎任何情况下的一个优秀的选择，它应该是你优先考虑的几种排序算法之一。

5.2.4 快速排序

由于快速排序的阶 $N \lg N$ 很有利，它无疑是最广泛使用的高级排序算法。甚至 ANSI C 库排序函数 qsort( ) 也具有暗示快速排序的名称（注意：虽然 qsort( ) 通常基于快速排序，但是在 ANSI C 的定义中没有任何方面要求用快速排序实现它）。不过，正确地实现快速排序稍微有些复杂，在一些不太常见的情况下很容易创建表现不佳的版本。由于这些原因，我们将研究快速排序的多种变体。关于这些变体的讨论将给你提供足够多的信息，用于测试你的编辑器库的 qsort( ) 版本的实现弱点，它们可能会严重地影响应用程序的性能。

快速排序对待排序的记录采用一种“分治”法。在排序的每一步，都把记录分成两个分区。这种分区是基于称为**基准**（pivot）的记录进行的。将小于基准的所有记录划分到一个分区中，而将大于基准的所有记录划分到另一个分区中。等于基准的记录则可以划分到任何一个分区中。通过允许将等于基准的记录划分到任何一个分区中，快速排序将把由于数据集包含许多具有等号键（equal key）的记录而引起的降级行为减至最少。然后，快速排序算法将对两个分区进行快速排序。下面给出了一些不严密的伪代码：

```
Quicksort (array A) begin
     If A has only one element, return.
     Choose an element, Array[i], as the pivot.
Create two subarrays, A1 and A2. Place elements <
     Array[i] into A1, elements > Array[i] into A2,
     and those equal to Array[i] into either A1 or
     A2. At the conclusion of this step, the array A
     contains the elements of A1, Array[i], and the
     elements of A2, in that order.
```

```
Quicksort (A1).
Quicksort (A2).
end
```

虽然这看起来足够简单，但是以一种有用的方式实现它需要处理以下复杂情况：（1）适当地选择基准；（2）在分配给 A 自身的内存内构造子数组 A1 和 A2，从而避免了需要任何额外的存储空间。要查看如何执行该任务的基本示例，可以检查 quick1.c（程序清单 5-7）。从测试驱动程序开始，我们看到它把待排序的数组与比较函数一起传递给非常短的例程 Quicksort1()。这个接口例程用于把所需的栈空间减至最少。它唯一的目的是把数组和比较函数的地址存储进 static 变量 StoredArray 和 StoredCompare 中，然后调用 xQuickSort1()，它是实际的排序例程。如果没有执行该操作，就必须在执行排序期间递归地传递这些地址，从而会减慢执行速度。这种方法的唯一缺点是：这些实现是不可重入的。也就是说，如果在多线程环境中使用这段代码，其中只把这个例程及其静态数据的可执行代码的一份副本加载进主内存中，那么不同的任务同时调用这个例程将会相互干扰。如果这是一种顾虑，可以消除 static 变量，并在每次调用 xQuickSort1() 时传递数组和比较函数的地址。

程序清单 5-7　基本快速排序的代码

```
/*--- quick1.c --------------------------- Listing 5-7 ---------
 * A basic quicksort
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to sortsub.c (Listing 5-1)
 *-------------------------------------------------------------*/

#include "sorthdr.h"

#define DRIVER 1

static CompFunc StoredCompare;
static Element **StoredArray;

static void xQuickSort1 ( int L, int R )
{
    if ( R > L )
    {
        int i, j;
        Element *temp;

        /* First, partition the array using array[R] as pivot */
        i = L - 1; /* Scan up from here   */
        j = R;     /* Scan down from here */
        for ( ;; )
        {
            /*
             * Looking from left, find element >= Array[R].
             * No sentinel needed, as Array[R] will stop us.
             */
            while ( StoredCompare ( StoredArray[++i],
                                    StoredArray[R] ) < 0 )
                        ;
```

```
            /*
             * Looking from right, find element <= Array[R].
             * The loop provides boundary checking.
             */
            while ( j > 0 )
            {
                if ( StoredCompare ( StoredArray[--j],
                                     StoredArray[R]) <= 0 )
                    break;
            }

            if ( i >= j )
                break;

            /* swap ith and jth elements */
            temp = StoredArray[i];
            StoredArray[i] = StoredArray[j];
            StoredArray[j] = temp;
        }


        /* swap ith and Rth elements */
        temp = StoredArray[i];
        StoredArray[i] = StoredArray[R];
        StoredArray[R] = temp;
        xQuickSort1 ( L, i-1 );
       xQuickSort1 ( i+1, R );
    }
}

void QuickSort1 ( Element **Array, int Items, CompFunc Compare )
{
    StoredCompare = Compare;
    StoredArray = Array;
    xQuickSort1 ( 0, Items - 1 );
}

#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sortsub.h"

/*
 * A comparison function
 */
int Cfunc ( Element *L, Element *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;

    if ( argc != 2 && argc != 3 )
```

```
    {
        fprintf ( stderr, "Usage: quick1 infile [maxitems]\n" );
        return;
    }

    if ( argc == 3 )
        Items = atoi ( argv[2] );

    if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
        return; /* Couldn't load file */

    QuickSort1 ( Array, Items, (CompFunc) Cfunc );
    ShowArray ( Array, Items, (CompFunc) Cfunc );
}
#endif
```

所有实际的工作都是在xQuickSort1( ) 中完成的。给它传递两个参数：L和R，它们是在这一遍中要排序的StoredArray的子数组的第一个和最后一个元素。假如R大于L（换句话说，在待排序的子数组中有多个元素），xQuickSort1( ) 就会对子数组进行分区。它会选择Array［R］作为它的基准，它是子数组的最后一个元素。然后，它从这个子数组的两端稳定地向里检查每个元素，并对L和R之间的所有元素进行分区。它使用计数器*i*和*j*执行该任务，其中*i*用于从子数组的左边开始计数，*j*用于从子数组的右边开始计数。增加每个计数器，直至*i*找到一个大于或等于Array［R］的元素，并且*j*找到一个小于或等于Array［R］的元素。这两个元素相互之间次序颠倒（如果元素等于Array［R］，它实际上处于正确的顺序，但是如以前所讨论的那样，尝试把等于基准的记录同时放在两个分区中是有好处的）。因此，要交换元素并且允许增加计数器。当两个计数器交叉时，就用Array［*i*］交换Array［R］。现在就完成了分区。已知Array［*i*］左边的所有元素具有的键都小于或等于Array［*i*］的键，而Array［*i*］右边的所有元素都大于或等于它。此外，Array［*i*］中的元素都处于它的最终位置，从不需要再次移动或检查。然后，代码将对子数组L到*i*以及子数组*i*+1到R递归地调用它自身。

现在花一点时间向自己证明该方法实际上会工作。构造一个数组，其中包含6或7个随机排序的数据项，并逐步遵照算法执行。然后对包含6或7个已经有序的数据项的数组再次执行算法。当提供这种类型的输入时，算法的性能令人吃惊，并且对于接下来的讨论很重要。另外，还要注意算法使它自身不会越界的方式。注意递增*i*的循环利用了以下事实：当*i*等于R时，比较函数将返回0并且终止循环。与之相反，递减*j*的循环必须确保*j*不会递减到0以下。可以避免这种测试，其方法是：在Array［0］中创建一条虚拟的记录，并且确保它的键小于任何真实的键。不过，如介绍中所指出的，这个过程并非总是很方便。

如果尝试刚才建议的笔和纸的练习，将会发现当给快速排序算法提供已经有序的数据项的数组时，它的性能将会糟糕地降级。从已经有序的数组中盲目地选择Array［R］作为基准，意味着在每一次遍历中都把所有的元素只放入一个分区中，并且无论数组是正序还是逆序，都会发生这种行为。表5-2中清楚显示了这种效果，其中quick1.c对于随机排序的记录运行得很快，但是对于正序和逆序记录则像插入排序一样运行得比较缓慢。另一个副作用是：在这些情况下，对栈空间的需求会显著增大，并且对每个待排序的元素都要执行一次递归调用。减小发生这种降级行为的可能性是我们的快速排序算法的下一个版本的目标。

quick2.c（程序清单5-8）纳入了两处修改，用于改进排序性能。第二个版本的总体结构与第一个版本相同，大部分修改集中在核心例程xQuickSort2()上。

第一处修改是：xQuickSort2()拒绝对任何少于10个元素的子数组进行排序。为什么呢？大量的分析表明：如果对小的子数组使用插入排序，而不是调用对小的子数组进行快速排序所需的递归机制。“小的子数组”的定义稍微有点不严密，但是在5～25这个范围中的任何值都工作得很好。如果检查排序驱动程序QuickSort2()，将会看到在它调用了xQuickSort2()之后，还会调用InsertionSort()来完成排序工作。由于xQuickSort2()将通过把整个数组安排进局部无序记录的有序组中来创建几乎有序的数组，因此插入排序的运行时间近似于线性的。

第二处修改与选择基准的方法相关。例程不是盲目地选择第R个元素，而是执行一个称为**三平均分区法**（median-of-three）的计算。这需要检查三个元素：第L个元素、第R个元素，以及粗略位于这两个元素中间的一个元素（在代码中称为mid）。对这三个元素进行排序，然后交换第mid个元素和第R个元素。一旦完成这个过程，就可以确保以下条件成立：

```
Array[L] <= Array[R-1] <= Arrray[R]
```

此时，选择Array［R］作为基准，并对L和R－2之间的元素进行分区。我们的查找循环现在可以使用元素Array［L］和Array［R］作为内置的哨兵，从而简化了这些循环。另请注意以下事实：xQuickSort2()不会对少于10个元素的子数组进行排序，这意味着可以从例程中省略大量特殊情况的代码；我们知道数组中有足够多的元素，因此L、mid、R－1和R都将被隔开，并且是截然不同的。

程序清单5-8　利用三平均分区法进行快速排序的代码

```
/*--- quick2.c -------------------------- Listing 5-8 ---------
 * Quicksort with median-of-three partitioning
 *     and insertion sort on small subfiles.
 *
 * Uses InsertionSort() from insert.c (Listing 5-5)
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to sortsub.c (Listing 5-1)
 *-------------------------------------------------------------*/

#include "sorthdr.h"

#define DRIVER 1

static CompFunc StoredCompare;
static Element **StoredArray;

static void xQuickSort2 ( int L, int R )
{
    if ( R - L >= 9 ) /* if if there are at least 10 elements */
    {
        int i, j, mid;
        Element *temp;

        /*
         * Sort Lth, Rth, and middle element. Then swap the
```

```
 * middle element with the R-1th element. This will
 * obviate the need for bound checking.
 */
mid = ( L + R ) / 2; /* this is the middle element */
if ( StoredCompare ( StoredArray[L],
                     StoredArray[mid] ) > 0 )
{
    temp = StoredArray[L];
    StoredArray[L] = StoredArray[mid];
    StoredArray[mid] = temp;
}
if ( StoredCompare ( StoredArray[L],
                     StoredArray[R] ) > 0 )
{
    temp = StoredArray[L];
    StoredArray[L] = StoredArray[R];
    StoredArray[R] = temp;
}
if ( StoredCompare ( StoredArray[mid],
                     StoredArray[R] ) > 0 )
{
    temp = StoredArray[mid];
    StoredArray[mid] = StoredArray[R];
    StoredArray[R] = temp;
}

temp = StoredArray[mid];
StoredArray[mid] = StoredArray[R-1];
      StoredArray[R-1] = temp;

      /*
       * Now, we know that Array[L] <= Array[R-1] <= Array[R].
       * We use Array[R-1] as the pivot, so this relationship
       * gives us known sentinels. Also, we need to
       * partition only between L+1 and R-2.
       */
      i = L;      /* Scan up from here   */
      j = R - 1; /* Scan down from here */
      for ( ;; )
      {
          /* Looking from left, find element >= Array[R-1] */
          while ( StoredCompare ( StoredArray[++i],
                                  StoredArray[R - 1] ) < 0 )
                      ;

          /* Looking from right, find element <= Array[R-1] */
          while ( StoredCompare ( StoredArray[--j],
                                  StoredArray[R - 1] ) > 0 )
                      ;

          if ( i >= j )
              break;

          /* swap ith and jth elements */
          temp = StoredArray[i];
          StoredArray[i] = StoredArray[j];
          StoredArray[j] = temp;
      }
```

```
            /* swap ith and R-1'th elements */
            temp = StoredArray[i];
            StoredArray[i] = StoredArray[R - 1];
            StoredArray[R - 1] = temp;

            /* and sort the two partitions */
            xQuickSort2 ( L, i-1 );
            xQuickSort2 ( i+1, R );
        }
    }

    void QuickSort2 ( Element **Array, int Items, CompFunc Compare )
    {
        void InsertionSort ( Element **, int, CompFunc );

        /* Save some things */
    StoredCompare = Compare;
    StoredArray = Array;

    /* Quicksort to get nearly sorted file */
    xQuickSort2 ( 0, Items - 1 );

    /* Do an insertion sort on the now nearly sorted file */
    InsertionSort( Array, Items, Compare );
}

#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sortsub.h"

/*
 * A comparison function
 */
int Cfunc ( Element *L, Element *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;

    if ( argc != 2 && argc != 3 )
    {
        fprintf( stderr, "Usage: quick2 infile [maxitems]\n" );
        return;
    }

    if ( argc == 3 )
        Items = atoi ( argv[2] );

    if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
        return; /* Couldn't load file */
```

```
    QuickSort2 ( Array, Items, (CompFunc) Cfunc );
    ShowArray ( Array, Items, (CompFunc) Cfunc );
}
#endif
```

如表5-2所示，程序清单5-8中演示的修改显著改进了快速排序的性能。它现在对随机顺序、逆序和正序记录运行得同样好。不过，在栈利用率和最坏情况下的性能这些领域可以执行进一步的改进，我们的第三个版本quick3.c（程序清单5-9）处理了其中第一个改进。

程序清单5-9　删除了尾部递归的改进后的快速排序的代码

```
/*--- quick3.c -------------------------- Listing 5-9 ---------
 * Quicksort with median-of-three partitioning,
 * insertion sort on small subfiles, and removal
 * of end recursion.
 *
 * Uses InsertionSort() from insert.c (Listing 5-5)
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to sortsub.c (Listing 5-1)
 *-------------------------------------------------------------*/

#include "sorthdr.h"

#define DRIVER 1

static CompFunc StoredCompare;
static Element **StoredArray;

static void xQuickSort3 ( int L, int R )
{
    while ( R - L >= 9 )  /* if there are at least 10 elements */
    {
        int i, j, mid;
        Element *temp;

        /*
         * Sort Lth, Rth, and middle element. Then swap the
         * middle element with the R-1'th element. This will
         * obviate the need for bound checking.
         */
        mid = ( L + R ) / 2; /* this is the middle element */

        if ( StoredCompare ( StoredArray[L],
                             StoredArray[mid] ) > 0 )
        {
            temp = StoredArray[L];
            StoredArray[L] = StoredArray[mid];
    StoredArray[mid] = temp;
}
if ( StoredCompare ( StoredArray[L],
                     StoredArray[R]) > 0 )
{
    temp = StoredArray[L];
    StoredArray[L] = StoredArray[R];
    StoredArray[R] = temp;
```

```
}
if ( StoredCompare ( StoredArray[mid],
                     StoredArray[R]) > 0 )
{
    temp = StoredArray[mid];
    StoredArray[mid] = StoredArray[R];
    StoredArray[R] = temp;
}

temp = StoredArray[mid];
StoredArray[mid] = StoredArray[R-1];
StoredArray[R-1] = temp;

/*
 * Now, we know that Array[L] <= Array[R-1] <= Array[R].
 * We use Array[R-1] as the pivot, so this relationship
 * gives us known sentinels. Also, we need to partition
 * only between L+1 and R-2.
 */
i = L;      /* Scan up from here   */
j = R - 1; /* Scan down from here */
for ( ;; )
{
    /* Looking from left, find element >= Array[R-1] */
    while ( StoredCompare ( StoredArray[++i],
                            StoredArray[R - 1] ) < 0 )
                    ;

    /* Looking from right, find element <= Array[R-1] */
    while ( StoredCompare ( StoredArray[--j],
                            StoredArray[R - 1]) > 0 )
                    ;

    if ( i >= j )
        break;

    /* swap ith and jth elements */
    temp = StoredArray[i];
    StoredArray[i] = StoredArray[j];
    StoredArray[j] = temp;
       }


       /* swap ith and R-1'th elements */
       temp = StoredArray[i];
       StoredArray[i] = StoredArray[R - 1];
       StoredArray[R - 1] = temp;

       /*
        * This, and the conversion of the main loop from
        * "if (R - L >= 9)" to "while (R - L >= 9)" are the
        * only places we differ from quick2.c. These small
        * changes have a big effect: by recursing only on the
        * small half and simply looping on the large half
        * of each partition, we eliminate the possiblity
        * that worst-case input could cause us to make N
        * recursive calls. Instead, the worst case becomes
```

```
         * log2 N calls.
         */
        if ( i - L > R - i )  /* left half is larger */
        {
            xQuickSort3 ( i + 1, R ); /* recurse on small half
*/
            R = i - 1;
        }
        else  /* right half is larger */
        {
            xQuickSort3 ( L, i - 1 );
            L = i + 1;
        }
    }
}

void QuickSort3 ( Element **Array, int Items, CompFunc Compare )
{
    void InsertionSort ( Element **, int, CompFunc );

    /* Save some things */
    StoredCompare = Compare;
    StoredArray = Array;

    /* Quicksort to get nearly sorted file */
    xQuickSort3 ( 0, Items - 1 );

    /* Do an insertion sort on the now nearly sorted file */
    InsertionSort ( Array, Items, Compare );
}
#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sortsub.h"

/*
 * A comparison function
 */
int Cfunc ( Element *L, Element *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;

    if ( argc != 2 && argc != 3 )
    {
        fprintf( stderr, "Usage: quick3 infile [maxitems]\n" );
        return;
    }

    if ( argc == 3 )
        Items = atoi(argv[2]);
```

```
    if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
        return; /* Couldn't load file */

    QuickSort3 ( Array, Items, (CompFunc) Cfunc );
    ShowArray ( Array, Items, (CompFunc) Cfunc );
}
#endif
```

quick3.c 删除了**尾部递归**（tail recursion）。为了查看它如何工作，观察一下，xQuickSort2( ) 做的最后两件事是：对它自身执行两次递归调用。虽然对第一次递归调用我们可以做的工作很少，但是以递归调用它们自身结束的例程被称为尾部递归。通常可以通过把例程的参数重新设置成与那些以递归方式传递的参数匹配，然后简单地循环回例程的顶部，来删除这种尾部递归。由于它与我们对两个分区进行快速排序的顺序无关，我们也可以选择对哪一半通过递归排序，对哪一半通过循环排序。这种选择很容易；对我们有利的做法是：对较小的一半使用递归，对较大的一半使用循环。这样，万一更坏的情况变成了最坏的情况并且我们偶尔碰上了性能降级的情况，我们将总是对子数组（也就是说，至多对前一个数组的一半大小）执行递归，从而至多执行 lg *N* 次递归调用，而不是 *N* 次这样的调用。

如你所看到的，修改 quick2.c 产生 quick3.c 的过程很简单。通过把 xQuickSort2( ) 中的测试从 if（R－L > = 9）改为 while（R－L > = 9），我们提供了一种循环机制。然后，我们通过修改循环末尾的调用来删除尾部递归。我们仍然会执行调用对较小的一半进行排序，但是我们只会把 L 和 R 的值修改成我们传递的用于对较大的一半进行排序的那些值，然后执行循环以实际地执行排序。

在表 5-2 中可以看到这些修改对最大栈深度的影响。在最坏情况下，对于正序的记录数组，quick2.c 达到了最大栈深度 26，而 quick3.c 则只会执行 8 次递归调用。

现在看起来好像我们已经做得非常好。我们有一个快速的排序例程，它使用合理的栈空间，不需要额外的存储空间，并且使用同样的方式处理正序、逆序和随机顺序的记录数组。它唯一明显的缺点是：它不是稳定的排序。

不过，还有最后一个与最坏情况下的性能相关的问题。考虑一下，对于三平均分区法基准选择过程，如果把下面这种形式的数组提供给它进行排序，会发生什么事情：

A B C D E F G H G F E D C G A

也就是说，我们首先具有一个稳定增加的序列，然后具有相同的逆序序列。在第一次遍历时，三平均分区法过程对第一个、最后一个和中间的条目（A、A 和 H）进行排序。在把中间的元素（其中一个 A）与倒数第二个元素（G）进行交换之后，我们通过把 A 用作基准继续对元素进行分区。现在我们就具有一种降级情况，并且所有的元素都大于基准。更可怕的是，这种情况倾向于在后续的每个分区中持续存在。我们显然身处麻烦中，但是我们应该避免尝试设计一种甚至更复杂的模式（四平均分区法或五平均分区法?），因为不管我们选择什么固定的模式，总是可以设计一种将导致它降级的输入模式。这里唯一真正的防御措施是随机性。通过简单地修改三平均分区法以便为其分区使用第一个和最后一个元素以及一个随机选择的元素，我们几乎可以消除发生性能降级的机会。

quick4.c（程序清单 5-10）中所示的快速排序版本纳入了这些改进以及另外一处代码增强。

这个版本没有使用显式的数组索引，而是直接使用等价的指针。通过这些修改，我们不仅具有了最稳定的快速排序版本，而且具有了最快速的版本（参见表5-2）。

程序清单5-10　具有随机选择的中间基准元素的快速排序的代码

```
/*--- quick4.c -------------------------- Listing 5-10 --------
 * Quicksort with median-of-three partitioning
 * based on a randomly selected middle element,
 * insertion sort on small subfiles, remoanl of
 * end recursion, and use of pointer
 * incrementing rather than an index.
 *
 * Uses InsertionSort() from insert.c (Listing 5-5)
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to sortsub.c (Listing 5-1)
 *-------------------------------------------------------------*/

#include <stddef.h>     /* for typedef of ptrdiff_t */
#include <stdlib.h>     /* for rand() */
#include "sorthdr.h"

#define DRIVER 1

static CompFunc StoredCompare;

static void xQuickSort4 ( Element **pL, Element **pR )
{
    ptrdiff_t diff; /* ptrdiff_t is a signed type that can hold
                       the difference between two pointers */

    while (( diff = ( pR - pL )) >= 9 )  /* 10 elements a must */
    {
        int mid;
        Element *temp, **pmid, **pi, **pj, *ppivot;

        /* select a random mid element */
        mid = abs ( rand() ) % diff;
        if ( mid < 1 || mid > diff - 2 )
            mid = 1;
        pmid = pL + mid;

        /*
         * Sort Lth, Rth, and middle element. Then swap the
         * middle element with the R-1'th element. This will
         * obviate the need for bound checking.
         */
        if ( StoredCompare ( *pL, *pmid ) > 0 )
        {
            temp = *pL;
            *pL = *pmid;
    *pmid = temp;
}
if ( StoredCompare ( *pL, *pR ) > 0 )
{
    temp = *pL;
    *pL = *pR;
```

```
    *pR = temp;
}
if ( StoredCompare ( *pmid, *pR ) > 0 )
{
    temp = *pmid;
    *pmid = *pR;
    *pR = temp;
}

temp = *pmid;
*pmid = *(pR-1);
*(pR-1) = temp;

/*
 * Now, we know that Array[L] <= Array[R-1] <= Array[R].
 * We use Array[R-1] as the pivot, so this relationship
 * gives us known sentinels. Also, we need to partition
 * only between L+1 and R-2.
 */
pi = pL;        /* Scan up from here */
pj = pR - 1;    /* Scan down from here */
ppivot = *pj;
for ( ;; )
{
    /* Looking from left, find element >= Array[R-1] */
    while ( StoredCompare ( *++pi, ppivot ) < 0 )
                ;

    /* Looking from right, find element <= Array[R-1] */
    while ( StoredCompare ( *--pj, ppivot ) > 0 )
                ;

    if ( pi >= pj )
        break;

    /* swap ith and jth elements */
    temp = *pi;
    *pi = *pj;
    *pj = temp;
}
       /* swap ith and the pivot */
       *(pR - 1) = *pi;
       *pi = ppivot;

       if ( pi - pL > pR - pi )  /* left half is larger */

       {
           xQuickSort4 ( pi+1, pR ); /* recurse on smaller half */
           pR = pi - 1;
       }
       else    /* right half is larger */
       {
           xQuickSort4 ( pL, pi-1 );
           pL = pi + 1;
       }
   }
}
```

```
void QuickSort4 ( Element **Array, int Items, CompFunc Compare )
{
    void InsertionSort ( Element **, int, CompFunc );

    /* Save some things */
    StoredCompare = Compare;

    /* Quicksort to get nearly sorted file */
    xQuickSort4 ( Array, Array + Items - 1 );

    /* Do an insertion sort on the now nearly sorted file */
    InsertionSort ( Array, Items, Compare );
}

#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include "sortsub.h"

/*
 * A comparison function
 */
int Cfunc ( Element *L, Element *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;
   if ( argc != 2 && argc != 3 )
   {
       fprintf ( stderr, "Usage: quick4 infile [maxitems]\n" );
       return;
   }

   if ( argc == 3 )
       Items = atoi ( argv[2] );
   if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
       return; /* Couldn't load file */

   QuickSort4 ( Array, Items, (CompFunc) Cfunc );
   ShowArray ( Array, Items, (CompFunc) Cfunc );
}
#endif
```

为了完成所有这些代码，将快速排序的最后一个版本展示为 quick5. c（程序清单 5-11）。这个版本利用了迄今为止讨论的所有改进，但它也是你的编译器的 qsort( ) 的“插件兼容式”替换方法。这个版本没有假定待排序的数据项数组是一个指针数组，而是对任意大小的元素的数组进行排序。它还避免使用静态数据项，因此能够可重入地安全使用它。如本章末尾所讨论的，应该测试你的编译器的 qsort( ) 版本，查看它是否运行良好；如果不是这样，你可能希望代之以使用 quick5. c 中的例程。

程序清单 5-11 使用 qsort( ) 语法的经过优化的快速排序的代码

```
/*--- quick5.c --------------------------- Listing 5-11 --------
 * qsort() replacement derived from quick4.c
 * that may be used to replace your compiler's
 * version of qsort(). Just compile this routine
 * and then call xqsort() rather than qsort().
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h> /* for rand() */
#include <string.h> /* for memcpy() */

typedef int ( *CF )();
#define A(x) (a + (x) * es)

static void InsertionSort ( char *a, int n, int es,
                            CF cf, char *b )
{
    int step, i;

    /* Look at 2nd thru Nth elements, putting each in place */
    for ( step = 1; step < n; step++ )
    {
        /* Now, look to the left and find our spot */
        memcpy ( b, A(step), es );
        for ( i = step - 1; i >= 0; i-- )
        {
            if ( cf ( A(i), b ) > 0 )
            {
                /* Not there yet, so make room */
                memcpy ( A(i+1), A(i), es );
            }
            else /* Found it! */
                break;
        }
        /* Now insert original value from Array[step] */
        memcpy ( A(i+1), b, es );

    }
}

static void xQuickSort5 ( char *a, int L, int R, int es,
                          CF cf, char *b )
{
    int diff;

    while (( diff = ( R - L )) >= 9 )  /* 10 elements a must */
    {
        int mid, i, j;
        char *ppivot;

        /* select a random mid element */
        mid = abs ( rand() ) % diff;
        if ( mid < 1 || mid > diff - 2 )
            mid = 1;
        mid += L;
```

```
        /*
         * Sort Lth, Rth, and middle element. Then swap the
         * middle element with the R-1'th element. This will
         * obviate the need for bound checking.
         */
        if ( cf ( A(L), A(mid) ) > 0 )
        {
    memcpy ( b, A(L), es );
    memcpy ( A(L), A(mid), es );
    memcpy ( A(mid), b, es );
}
if ( cf ( A(L), A(R) ) > 0 )
{
    memcpy ( b, A(L), es );
    memcpy ( A(L), A(R), es );
    memcpy ( A(R), b, es );
}
if ( cf ( A(mid), A(R) ) > 0 )
{
    memcpy ( b, A(mid), es );
    memcpy ( A(mid), A(R), es );
    memcpy ( A(R), b, es );
}

memcpy ( b, A(mid), es );
memcpy ( A(mid), A(R-1), es );
memcpy ( A(R-1), b, es );

/*
 * Now, we know that Array[L] <= Array[R-1] <= Array[R].
 * We use Array[R-1] as the pivot, so this relationship
 * gives us known sentinels. Also, we need to partition
 * only between L+1 and R-2.
 */
i = L;        /* Scan up from here */
j = R - 1;    /* Scan down from here */
ppivot = A ( j );
for ( ;; )
{

    /* Looking from left, find element >= Array[R-1] */
    while ( cf ( A(++i), ppivot ) < 0 )
                ;

    /* Looking from right, find element <= Array[R-1] */
    while ( cf ( A(--j), ppivot ) > 0 )
                ;

    if ( i >= j )
        break;

    /* swap ith and jth elements */
    memcpy ( b, A(i), es );
    memcpy ( A(i), A(j), es );
    memcpy ( A(j), b, es );
}
```

```
        /* swap ith and the pivot */
        memcpy ( b, A(R-1), es );
        memcpy ( A(R-1), A(i), es );
        memcpy ( A(i), b, es );

        if ( i - L > R - i )    /* left half is larger */
        {
            xQuickSort5 ( a, i+1, R, es, cf, b );
            R = i - 1;
        }
        else  /* right half is larger */
        {
            xQuickSort5 ( a, L, i-1, es, cf, b );
            L = i + 1;
        }
    }
}

void xqsort ( char *a, int n, int es, CF cf )
{
    #define DEFAULT_BUFFER 64
    char buf[DEFAULT_BUFFER], *b;

    printf ( "sorting %d elements of size %d\n", n, es );

    /* allocate space for making a copy of an item */
    b = buf;
    if ( es > DEFAULT_BUFFER )
    {
        b = (char *) malloc ( es );
        if ( b == NULL )
        {
            qsort (a, n, es, cf );
            return;
        }
    }

    /* Quicksort to get nearly sorted file */
    xQuickSort5 ( a, 0, n - 1, es, cf, b );

    /* Do an insertion sort on the now nearly sorted file */
    InsertionSort ( a, n, es, cf, b );

    /* Cleanup */
    if ( b != buf )
        free ( b );
}
```

5.2.5 堆排序

一般来讲，正确实现的快速排序难以超越。不过，快速排序的最坏情况下的性能与 N^2 成正比，总是存在你的程序将遇到最坏条件的可能性。我们的快速排序的最终版本采取了许多步骤使它自身避免这种情况，但是有时当你的需求确定时，将不存在可能性。我们已经研究过了希尔排序，它是一种良好的通用排序算法，并且保证阶大约是 $N^{1.25}$。在本节中，我们将研究堆排序，它是阶为 $N \lg N$ 的另一种排序算法，通常比快速排序要慢一些，但是不存在最坏情况。

堆排序在工作时把待排序的数组视作一棵二叉树。例如，如果数组中有15个条目，则可以把每个条目都视作图5-1中所示的二叉树中对应的节点。可以使用下面三个简单的公式定义这棵树：

```
array[n]的父节点：        array[(n-1)/2]
array[n]的子节点：        array[n * 2 + 1]
                    和    array[n * 2 + 2]
```

也就是说，array［0］的子节点将是array［1］和array［2］，而array［5］的父节点将是array［2］。注意：这些公式适用于C语言风格的基于0的数组；如果研究堆排序上的其他工作，可以看到它们通常使用基于1的数组，其中正确的公式是：a［n］的父节点是a［$n/2$］，a［n］的子节点是a［$n*2$］和a［$n*2+1$］。

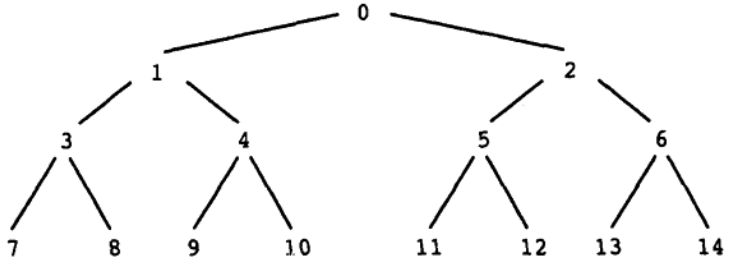


图5-1　节点的二叉树

然后，堆排序确保这棵二叉树中的条目构成一种称为**堆**（heap）的结构。要创建堆，只需确保每个节点中的值大于或等于该节点的子节点的值。这并不意味着条目是有序的，以便标准的树遍历算法（参见下一章）将产生有序的数据项。堆条件只要求节点3中的数据大于或等于节点7和8中的数据；节点4、9和10中的数据可能大于或小于节点3、7和8中的数据。但是，由于节点0中的数据必须大于或等于节点1和2中的数据——它们反过来必须大于或等于节点3、4、5和6中的数据，依此类推——节点0中的数据项将是堆中最大的数据项。因此，一旦实施堆条件，找到最大的数据项就很简单：它在array［0］中。如果删除了这个数据项，并且重新实施堆条件，次大的数据项现在就在array［0］中。通过重复这个过程，可以按顺序提取数据项。

heap.c（程序清单5-12）中显示了实现这种排序的代码。关键例程是downheap()。利用数组的开始和结束索引（节点编号）调用这个例程，它负责在这个范围内实施堆条件。这个例程极其简单，因为它假定已经部分满足了堆条件；如果存在无序情况，就把它限制于第一个节点及其子节点。因此，这个例程可以像下面的伪代码一样简单：

```
downheap(first, last) {
    parent = first;
    child = parent * 2 + 1;
    while (child <= last)  {
        /* Find larger of two children */
        if (child + 1 <= last &&
            array[child+1] > array[child])
            child++;

        /* Exit if parent > largest child */
```

```
        if (array[parent] > array[child}
            break;

        /*
         * Child is larger than parent. Swap
         * that child with parent and keep
         * looking down heap to make sure
         * heap condition is satisified.
         */
        temp = array[child];
        array[child] = array[parent];
        array[parent] = array[child];
        parent = child;
        child = parent * 2 + 1;
    }
}
```

为了实际地执行排序，为上述例程配合使用一段代码，这段代码通过从最深的父节点开始并向上处理到节点0来实施堆条件：

```
for (i = (N - 1) / 2; i >= 0; i--)
    downheap(i, N);
```

由于这个例程开始于最小、最深的堆并回溯处理，因此将在整个堆上逐步实施堆条件。如前所述，这种方法允许 downheap( ) 相当简单。

程序清单5-12 堆排序的代码

```
/*--- heap.c ----------------------------- Listing 5-12 --------
 * Heapsort on an array. Uses basic heapsort
 * algorithm, with Floyd's modification to
 * reduce order from 2N lg N to N lg N.
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to sortsub.c (Listing 5-1)
 *-------------------------------------------------------------*/

#include "sorthdr.h"

#define DRIVER 1

static CompFunc StoredCompare;
static Element **StoredArray;

/* Enforce heap condition between first and last */
static void downheap ( int first, int last )
{
    int child, parent, i;
    Element *temp;
    for ( i = first; ( child = i * 2 + 1 ) <= last; i = child )
    {
        if ( child + 1 <= last &&
            StoredCompare ( StoredArray[child + 1],
                            StoredArray[child] ) > 0 )
            child += 1;

        /* child is the larger child of i */
        temp = StoredArray[i];
```

```
        StoredArray[i] = StoredArray[child];
        StoredArray[child] = temp;
    }
    while ( 1 )
    {
        parent = ( i - 1 ) / 2;
        if ( parent < first || parent == i ||
            StoredCompare ( StoredArray[parent],
                            StoredArray[i] ) > 0 )
            break;
        temp = StoredArray[i];
        StoredArray[i] = StoredArray[parent];
        StoredArray[parent] = temp;

        i = parent;
    }
}
void HeapSort ( Element **Array, int N, CompFunc Compare )
{
    int i;
    Element *temp;

    /* Make array and compare function available to all */
    StoredCompare = Compare;
    StoredArray = Array;

    /* Make N equal to largest index */
    N -= 1;

    /* First, ensure heap property for array */
    for ( i = ( N - 1 ) / 2; i >= 0; i-- )
        downheap ( i, N );

    /*
     * Now sort by taking advantage of the fact that the
     * largest element is in Array[0]. If we remove this element
     * and move it to Array[N-1], that element is now in place.
     * We continue to sort by reenforcing the heap property on
     * Array[0 .. N-2] and then taking the next largest element
     * and moving it to Array[N-2]. By repeating this process,
     * we will ultimately sort the array.
     */
    for ( i = N; i > 0; )
    {
        temp = Array[i];
        Array[i] = Array[0];
        Array[0] = temp;
        downheap ( 0, --i );
    }
}
#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "sortsub.h"
/*
 * A comparison function
 */
```

```
int Cfunc ( Element *L, Element *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;

    if ( argc != 2 && argc != 3 )
    {
        fprintf ( stderr, "Usage: heap infile [maxitems]\n" );
        return;
    }

    if ( argc == 3 )
        Items = atoi ( argv[2] );

    if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
        return; /* Couldn't load file */

    HeapSort ( Array, Items, (CompFunc) Cfunc );
    ShowArray ( Array, Items, (CompFunc) Cfunc );
}
#endif
```

然后将执行排序，其方法是：反复获取 Array［0］中的元素（它现在是数组中最大的元素），把它移到数组末尾，并对数组的余下部分实施堆条件。

```
for (i = N; i > 0; ) {
    temp = Array[i];
    Array[0] = temp;
    downheap(0, --i);
}
```

heap. c 中的实际代码密切遵循这种模型，但是对 downheap( ) 执行了一处修改。实证研究发现：如果 downheap( ) 自始至终简单地把初始父节点中的元素推向堆的底部，然后让它浮动到其最终驻留的位置上，那么其执行速度要快两倍左右。因此，downheap( ) 的生产版本具有两个循环：第一个循环反复把父节点与其最大的子节点进行交换，直至到达树的底部为止；第二个循环反向遍历树中所处理的数据项，直至该数据项处于合适的位置为止。

从表 5-2 中可以看到，堆排序的这种实现比较和谐，但不会产生惊人的效果。即使它具有与快速排序相同的阶 $N \lg N$，它所具有的更长的内层循环也使得它的运行速度比快速排序要慢一些。不过，当你简直不能冒险由于出现最坏情况的条件而使排序变慢时，堆排序就是一个良好的、考虑周全的选择。

5.3 对链表进行插入排序

插入排序是我们将修改用于处理链表的两种排序中的第一种排序。像基于数组的排序一样，基于链表的排序利用了 lsortsub. c（程序清单 5-13）、lsortsub. h（程序清单 5-14）和 lsorthdr. h（程

序清单5-15）中给出的支持例程和定义。这些例程将对通过以下形式的结构（来自于lsorthdr.h）而构建的单链表进行排序：

```
typedef struct sElement {
    char *text;
    struct sElement *next;
} Element;
```

程序清单5-13　用于链表的插入排序的支持例程

```
/*---lsortsub.c-------------------------Listing 5-13----------
 * I/O subroutines for sorting routines
 * Loads data into an array
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "lsorthdr.h"
#include "lsortsub.h"

#define MAX_ITEM_SIZE 500 /* max length of a text item */
/*
 * Loads text strings from FileName into a linked list. Returns
 * the number of items in the list of -1 for failure.
 */
int LoadList ( char *FileName, Node **ListHead )
{
    FILE *infile;
    char buffer[MAX_ITEM_SIZE], *s;
    int i;
    Node **current;

    if (( infile = fopen ( FileName, "r" )) == NULL )
    {
        fprintf ( stderr, "Can't open file %s\n", FileName);
        return ( -1 );
    }

    i = 0;
    current = ListHead;
    while ( fgets ( buffer, MAX_ITEM_SIZE, infile ))
    {
        /* trim trailing control characters */
        s = buffer + strlen ( buffer );
        while ( iscntrl ( *s ))
            *s-- = 0;

        /* make space and store it */
        *current = malloc ( sizeof (Node) );
        if ( *current == NULL )
        {
            fprintf ( stderr, "Can't get memory for data\n" );
            return ( -1 );
        }
```

```
        (*current)->text = malloc ( strlen ( buffer ) + 1 );
        if ( (*current)->text == NULL )
        {
            fprintf ( stderr, "Can't get memory for data\n" );
            return ( -1 );
        }
        strcpy ( (*current)->text, buffer );

        current = &( (*current)->next ); /* advance */
        i++; /* keep count */
    }
    *current = NULL; /* terminate the list */

    /*
     * If the linked list contains only one element, and that
     * element is an empty string, return an empty list.
     */
    if ( *ListHead != NULL &&
        (*ListHead)->next == NULL &&
        (*ListHead)->text[0] == 0 )
    {

        *ListHead = NULL;
        i = 0;
    }

    fclose ( infile );
    return ( i );
}

/* Display array of items */
void ShowArray ( Node *ListHead, CompFunc Compare )
{
    int i = 0, sorted = 1, column = 1;

    for ( ; ListHead != NULL; ListHead = ListHead->next )
    {
        if ( column > 61 )
        {
            printf ( "\n" );
            column = 1;
        }
        else  while (( column - 1 ) % 20 )
        {
            printf ( " " );
            column += 1;
        }
        printf ( "%3d: %s", i++, ListHead->text );
        column += 5 + strlen ( ListHead->text );
        if ( ListHead->next )
        {
            if ( Compare ( ListHead, ListHead->next ) > 0 )
                sorted = 0;
        }
    }
```

```
    if ( sorted )
        printf ( "\n\nThe linked list is sorted.\n" );
    else
        printf ( "\n\nThe linked list is not sorted.\n" );
}
```

程序清单 5-14　程序清单 5-13 的原型

```
/*--- lsortsub.h ------------------------- Listing 5-14 -----
 * Prototypes of functions in lsortsub.c (Listing 5-13)
 *-------------------------------------------------------------*/

int LoadList ( char *FileName, Node **ListHead );
void ShowArray ( Node *ListHead, CompFunc Compare );
```

程序清单 5-15　lsorthdr. h 的代码

```
/*--- lsorthdr.h ------------------------- Listing 5-15 -------
 * General definitions for linked-list oriented sort routines
 *-------------------------------------------------------------*/

/*
 * We sort a linked list of these structures. The structure
 * could contain anything you'd like: you need only to define
 * appropriate comparison functions.
 */
typedef struct sNode {
    char *text;
    struct sNode *next;
} Node;

/*
 * A type for the comparison function: a symbol with type
 * CompFunc is a pointer to a function that takes two pointers
 * to void and returns int
 */
typedef int (*CompFunc) ( void *, void * );
```

可以轻松地把插入排序修改成对链表进行排序，如 linsert. c（程序清单 5-16）中所示。链表版本与数组版本的区别体现在两个方面。第一，在插入每个新记录时，必须从头至尾查找已经有序的记录。如以前所讨论的，这会导致算法在处理已经有序的记录时会变得最慢，而在处理逆序记录时则会变得最快。不幸的是，假设单链表只能从头至尾进行遍历，这将是不可避免的。

第二处修改实际上是由于第一处修改引起的，但它更微妙一点。如果检查内层循环中使用的比较函数，将会看到只要测试记录（由 walk 指向的记录）大于或等于正在检查的记录，该函数就会继续执行查找。这个小小的改变使得算法可以保持重要的稳定性，并且查看它是如何工作的很有趣。表 5-5 显示了 4 条记录，它们分别具有键 A1、A2、B1、B2。这些记录已经基于其数字部分进行了排序：序列 B1 A1 B2 A2。我们的任务是使用正向查找链表插入排序，基于它们的字母部分对它们进行排序。首先，我们将立刻执行该任务，并且使用 > = 测试。在第一遍，我们把 A1 置于 B1 之前。在第二遍，我们必须插入 B2。由于只要新记录大于或等于（> =）有序的记录我们就会继续前进，所以我们会跳过 A1 和 B1，并在 B1 后面插入 B2。最后，在第三遍，我们在 A1

后面插入A2，得到最终的正确序列：A1 A2 B1 B2。如果我们只使用>测试重复这个过程，很快就会遇到麻烦。第一遍很顺利，因为A小于B，但是当我们在第二次遍历期间插入B2时，当我们看到B1时（而不是在看到它之后）就会停止查找，从而会把B2放在B1之前。对于A2将会发生相同的事情，从而得到最终的次序：A2 A1 B2 B1。虽然这些记录就它们的字母部分而言是正确排序的，但是这种排序丢失了输入序列中存在的数字次序。

表5-5 比较的选择决定了链表的插入排序的稳定性。已经有序的记录以斜体显示

遍历次数	测试>=	测 试>
0	B1 A1 B2 A2	B1 A1 B2 A2
1	A1 B1 B2 A2	A1 B1 B2 A2
2	A1 B1 B2 A2	A1 B2 B1 A2
3	A1 A2 B2 B1	A2 A1 B2 B1

程序清单5-16 链表的插入排序的代码

```
/*--- linsert.c ------------------------- Listing 5-16 --------
 * Insertion sort of a linked list
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to lsortsub.c (Listing 5-13)
 *-------------------------------------------------------------*/

#include <stdlib.h>
#include "lsorthdr.h"

#define DRIVER 1

/* Sort from *ListHead to ListEnd */

void InsertionSortLink ( Node **ListHead, Node *ListEnd,
                         CompFunc Compare )
{
    Node *newlist;  /* we build up the new list here */
    Node *walk, *save;

    newlist = ListEnd;  /* new list will end with this pointer */
    walk = *ListHead;   /* get ready to walk the list */

    for ( ; walk != ListEnd; walk = save )
    {
        /*
         * Run thru newlist, looking for an element that is
         * greater than walk. Watch the value in pnewlink
         * carefully--it points to the 'next' element of
         * of the structure that points to the structure
         * currently under scrutiny. Thus, pnewlist is
         * the equivalent of the 'last' pointer that is
         * required by linked-list insertion algorithms.
         */
        Node **pnewlink;
        for ( pnewlink = &newlist;
```

```
            *pnewlink != ListEnd &&
              Compare(walk, *pnewlink ) >= 0;
             pnewlink = &((*pnewlink)->next) )
                   ;

        save = walk->next;      /* save this */
        walk->next = *pnewlink; /* and link in */
        *pnewlink = walk;
    }

    *ListHead = newlist;
}

#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include "lsortsub.h"

/*
 * A comparison function
 */
int Cfunc ( Node *L, Node *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}
void main ( int argc, char *argv[] )
{
    Node *ListHead;

    if ( argc != 2 )
    {
        fprintf ( stderr, "Usage: linsert infile\n" );
        return;
    }

    if ( LoadList ( argv[1], &ListHead ) == -1 )
        return; /* Couldn't load file */

    InsertionSortLink ( &ListHead, NULL, (CompFunc) Cfunc );
    ShowArray ( ListHead, (CompFunc) Cfunc );
}
#endif
```

5.4　对链表进行快速排序

我们的第二种链表排序是快速排序。可以把希尔排序修改成对链表进行排序，但是将需要大量的链表遍历。令人感兴趣的是，修改快速排序使之适合于链表的过程很直观（注意：有关这个主题的讨论将不会重复介绍在以前关于快速排序的一节中讨论过的信息。在阅读本节内容时，你可能想参考那一节中讨论的内容）。lquick1.c（程序清单 5-17）中显示了我们的第一个实现。通过使用与 linsert.c 中的 InsertionSortLink( ) 使用的相同接口，我们将以一种与 quick3.c 中非常相似的方法执行处理。由于实现三平均分区法很复杂（更不用说消耗的时间），代码将简单地获取第一个元素作为基准。分区循环遍历链表中其余的元素，并把它们放入两个新链表之一中。将两

个链表与基准链接在一起，然后通过结合递归和循环对子链表进行排序。与 quick3.c 一样，使用插入排序对小的子链表进行排序。不过，与 quick3.c 不同的是，插入排序对于正序链表的执行性能不佳（参见表 5-2），因此必须单独对小的无序子链表执行它，而不是同时对整个近似有序的链表执行它。

你可能预测到这个例程对于正序或逆序链表执行性能不佳，并且表 5-2 中的数据证实了这一点。不过，值得注意的是：消除尾部递归使得最坏情况下的最大栈深度成为唯一的要求。尽管有这个缺点，其实现确实具有非常有趣的性质：稳定性。这种惊喜来自于执行分区的方式。当把元素划分进两个子链表之一中时，总是把新元素放置在所有旧元素之后。因此，具有相同键的元素将维持相同的相对次序。要实现这种效果，还要求将小于基准的数据项放在左边的子链表中，而那些大于或等于基准的数据项则放在右边的子链表中。如前所述，在对包含大量具有相同键的记录的数据进行排序时，这种类型的分区可能会产生降级的性能。

程序清单 5-17 链表的快速排序的代码

```
/*--- lquick1.c ------------------------- Listing 5-17 --------
 * Quicksort for linked lists. Uses an insertion
 * sort on small subfiles and eliminates tail
 * recursion to minimize stack usage.
 *
 * Uses InsertionSortLink() from linsert.c (Listing 5-16)
 *
 * Adapted from an article by Jeff Taylor in
 * C Gazette, Vol 5, No. 6, 1991.
 *
 * #define DRIVER to compile a test driver
 * Driver must be linked to lsortsub.c (Listing 5-13)
 *-------------------------------------------------------------*/

#include "lsorthdr.h"

#define DRIVER 1

static CompFunc StoredCompare;
void InsertionSortLink ( Node **, Node *, CompFunc );

static void xQuickSortL1 ( Node **Head, Node *End )
{
    int left_count, right_count, count;
    Node **left_walk, *pivot, *old;
    Node **right_walk, *right;

    if ( *Head != End )
    do {
        pivot = *Head;        /* Take first element as pivot */
        left_walk = Head;     /* Set up left & right halves */
        right_walk = &right;
        left_count = right_count = 0;
/* Now, walk the list */
for ( old = (*Head)->next; old != End; old = old->next )
{
    if ( StoredCompare ( old, pivot ) < 0 )
```

```
    {
        /* Less than pivot, so goes on left */
        left_count += 1;
        *left_walk = old;
        left_walk = &(old->next);
    }
    else
    {
        /* greater than or equal, so goes on right */
        right_count += 1;
        *right_walk = old;
        right_walk = &(old->next);
    }
}

/* Now glue the halves together... */
*right_walk = End;    /* Terminate right list */
*left_walk = pivot;   /* Put pivot after things on left */
pivot->next = right; /* And right list after that */

/* Now sort the halves in more detail */
if ( left_count > right_count )
{
    /*
     * Recursively sort (smaller) right half and then
     * reset local pointers so that when we loop we
     * will see the left half as the entire list. Also,
     * if the right half has fewer than 10 elements,
     * sort it by insertion rather than by quicksort.
     */
    if ( right_count >= 9 )
        xQuickSortL1 ( &(pivot->next), End );
    else
        InsertionSortLink ( &(pivot->next), End,
                            StoredCompare );
    End = pivot;
    count = left_count;
}
else
{
    /* Converse case */
    if ( left_count >= 9 )
        xQuickSortL1 ( Head, pivot );
    else
            InsertionSortLink ( Head, pivot, StoredCompare );
        Head = &(pivot->next);
        count = right_count;
        }
    }
    while ( count > 1 );    /* end of do-while */
}

void QuickSortLink ( Node **Head, Node *End, CompFunc Compare )
{

    /* Save address of comparison function */
    StoredCompare = Compare;
```

```
    /* Quicksort the list */
    xQuickSortL1 ( Head, End );
}

#ifdef DRIVER
#include <stdio.h>
#include <string.h>
#include "lsortsub.h"

/*
 * A comparison function
 */
int Cfunc ( Node *L, Node *R )
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Node *ListHead;

    if ( argc != 2 )
    {
        fprintf ( stderr, "Usage: lquick1 infile\n" );
        return;
    }

    if ( LoadList ( argv[1], &ListHead ) == -1 )
        return; /* Couldn't load file */

    QuickSortLink ( &ListHead, NULL, (CompFunc) Cfunc );
    ShowArray ( ListHead, (CompFunc) Cfunc );
}
#endif
```

对于通常的输入类型，降级的最坏情况下的性能是不可接受的，因此 lquick2. c（程序清单 5-18）中显示了解决这种问题的一个简单的方法。在此，我们将不会使用第一个元素作为基准，而是使用一个随机选择的基准。这要求：(1) 在第一次调用 xQuickSortL2( ) 之前对链表进行计数；(2) 扫描链表，找到随机选择的基准。这些改变导致排序不稳定，但是如表 5-2 中的数据所示，新的排序对于各类输入的执行速度都很快。

程序清单 5-18　利用随机选择的基准对链表进行快速排序的代码

```
/*--- lquick2.c ------------------------- Listing 5-18 --------
 * Quicksort for linked lists. Uses an insertion
 * sort on small subfiles, eliminates tail
 * recursion to minimize stack usage, and uses
 * a randomly selected pivot to avoid problems
 * with ordered input files.
 *
 * Uses InsertionSortLink() from linsert.c (Listing 5-16)
 *
 * Adapted from an article by Jeff Taylor in
 * C Gazette, Vol 5, No. 6, 1991.
 *
```

```
 * #define DRIVER to compile a test driver
 * Driver must be linked to lsortsub.c (Listing 5-13)
 *-------------------------------------------------------------*/

#include <stdlib.h>
#include "lsorthdr.h"

#define DRIVER 1

static CompFunc StoredCompare;
void InsertionSortLink ( Node **, Node *, CompFunc );

static void xQuickSortL2 ( Node **Head, Node *End, int N )
{
    int left_count, right_count, npivot;
    Node **left_walk, *pivot, *old;
    Node **right_walk, *right;

    while ( N > 1 )
    {
        if ( N <= 9 )  /* Insertion sort small lists */
        {
            InsertionSortLink ( Head, End, StoredCompare );
            break;
        }
/* Select a pivot, but not at either end! */
npivot = abs ( rand() ) % N;
if ( npivot < 2 || npivot > N - 2 )
    npivot = 2;

/* Run thru the list to the randomly selected point */
old = *Head;
while ( npivot-- )
    old = old->next;
pivot = old->next;       /* Take as pivot */
old->next = pivot->next; /* Cut from chain */

/* Logic is now basically the same as lquick1.c */
left_walk = Head;    /* Set up left & right halves */
right_walk = &right;
left_count = right_count = 0;

/* Now walk the list */
for ( old = *Head; old != End; old = old->next )
{
    if ( StoredCompare ( old, pivot ) < 0 )
    {
        /* Less than pivot, so goes on left */
        left_count += 1;
        *left_walk = old;
        left_walk = &(old->next);
    }
    else
    {
        /* Greater than or equal to, so goes on right */
        right_count += 1;
        *right_walk = old;
```

```
            right_walk = &(old->next);
        }
    }

    /* Now glue the halves together... */
    *right_walk = End;    /* Terminate right list */
    *left_walk = pivot;   /* Put pivot after things on left */
    pivot->next = right; /* And right list after that */

    /* Now sort the halves in more detail */
    if ( left_count > right_count )
    {
        /*
         * Recursively sort (smaller) right half and then
         * reset local pointers so that when we loop, we
                * will see the left half as the entire list,
                */
               xQuickSortL2 ( &(pivot->next), End, right_count );
               End = pivot;
               N = left_count;
           }
           else
           {
               /* Converse case */
               xQuickSortL2 ( Head, pivot, left_count );
               Head = &(pivot->next);
               N = right_count;
           }
       }
    }

    void QuickSortLink ( Node **Head, Node *End, CompFunc Compare )
    {
        Node *walk;
        int count = 0;
        /* Save address of comparison function */
        StoredCompare = Compare;

        /* Count the list */
        for ( walk = *Head; walk != End; walk = walk->next )
            count += 1;

        /* Quicksort the list */
        xQuickSortL2 ( Head, End, count );
    }

    #ifdef DRIVER
    #include <stdio.h>
    #include <string.h>
    #include "lsortsub.h"

    /*
     * A comparison function
     */
    int Cfunc ( Node *L, Node *R )
```

```
{
    return ( strncmp ( L->text, R->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Node *ListHead;
    if ( argc != 2 )
    {
        fprintf ( stderr, "Usage: lquick2 infile\n" );
        return;
    }

    if ( LoadList ( argv[1], &ListHead ) == -1 )
        return; /* Couldn't load file */

    QuickSortLink ( &ListHead, NULL, (CompFunc) Cfunc );
    ShowArray ( ListHead, (CompFunc) Cfunc );
}
#endif
```

5.5　对多个键进行排序——不稳定排序的修正方法

不幸的是，我们的三个最佳的算法（快速排序、堆排序和希尔排序）都是不稳定的（参见表5-1）。虽然对这个问题没有一种通用的修正方法，但是仍然有可能对包含多个键的记录成功地使用这些排序算法。技术是使用复合键。假设我们具有一组描述了以下事务的记录：

```
struct Tr {
     long date;
     long cust_id;
     /* more stuff here */
} ;
```

如果我们具有已经按日期排好序的这样一些事务的数组，那么如果通过每个客户的记录（它们按日期排序）维护的 cust_id 进行二次排序，则是合乎需要的。为了使用希尔排序或快速排序产生按 cust_id 排序然后按 date 进行二次排序的数组，就需要使用复合键。为了实现复合键，只需更改比较函数。例如，下面这个函数将正确地对我们的事务记录进行排序：

```
int compare(struct Tr *left, struct Tr *right)
{
     long diff;
     /* look at customer id first */
     diff = right->cust_id - left->cust_id;
     if (diff != 0)
          return diff < 0 ? -1 : 1;
     /* the id's match, so now check date */
     diff = right->date - left->date;
     if (diff < 0)
          diff = diff < 0 ? -1 : 1;
     return (int) diff;
}
```

除了这种方法需要一些操作以避免可能发生的 int 和 long 转换的问题之外，这个比较例程非常简单。它首先比较 cust_id 字段。如果两个字段不同，就会返回一个非 0 的比较值。如果它们相

同，我们就具有来自同一个客户的一对事务。然后通过检查 date 字段比较这些事务。

因此，可以说我们使用了一个**逻辑**（logical）复合键。从未创建一个物理键，它包含连接有 date 的 cust_id。我们的比较函数代之以通过按顺序检查每个记录内的不同字段来创建逻辑键。虽然我们仍然没有使快速排序或希尔排序稳定，但是使用这个比较函数确实允许我们产生一个首先按 cust_id 排序然后按 date 排序的链表。

5.6 网络排序

作为以前讨论过的排序算法的一种对应算法，现在让我们研究一种特殊的排序，称为**网络排序**（network sort）。在我们迄今为止见过的所有排序中，待排序的数据会影响任何给定实现的运行时间，因为对于不同的输入将会执行不同次数的比较和交换。不过，网络排序完全是预置的。所有的比较都是预先确定的，并且它们总是会执行。为了使之清楚明了，让我们考虑用于 4 个元素的排序网络，如图 5-2 所示。

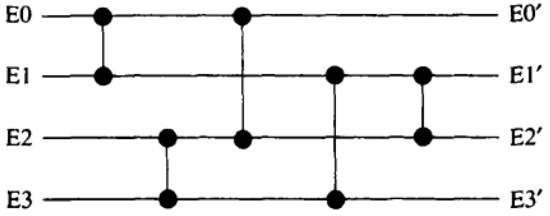


图 5-2 对 4 个元素的网络进行排序

我们为这个网络提供了 4 个待排序的元素（E0 ~ E3）。每个元素都沿着它的水平线移动，当它遇到一条垂直接线时，就会把该元素与接线另一端的元素作比较（并且可能会进行交换），并且将最终的输出显示为 E0′ ~ E3′（花点时间向自己证明它确实会在任何情况下正确地执行排序）。这种排序网络有两个迷人的特性。第一，它们总是需要相同的执行时间。在图 5-2 所示的示例中，总会执行 5 次比较 - 交换。第二，如果比较 - 交换对是合适的，就可以同时执行一些比较 - 交换。例如，在该图中，前两次比较 - 交换是完全独立的，并且可以以任意顺序或者同时执行。第二对比较交换也是如此。这意味着 4 个元素的排序网络可以运行在只执行三次比较 - 交换所需的时间内。

现在，问题变成了一个确定一组最优交换对的问题。虽然已知对 n 个元素进行排序所需的绝对最少的比较 - 交换次数是 $\log_2 n!$，并且已知使用这个最少比较 - 交换次数的排序网络适用于某些 n 值，但是对于任意的 n 值来说，没有一种常规的方式可用于导出什么是最优的比较 - 交换对的集合。用于解决这个问题的一种良好的方法是 Bose-Nelson 算法，它用于生成比较 - 排序对。我们将不会深入研究该算法为什么会工作的细节（参见 Bose 和 Nelson 所著的原始文章［Bose 1962］，以及 Knuth 关于其他排序网络的更一般的讨论［Knuth 1973］），但是我们将在程序清单 5-19 中查看该算法被实现为 bose-nel. c（基于 Hibbard 设计的经过改进的排序对生成算法［Hibbard 1963］）。

程序清单 5-19 用于生成交换对的代码

```
/*--- bose-nel.c ------------------------ Listing 5-19 --------
 * Generate swap pairs based on the Bose-Nelson
```

```
 * technique, using the algorithm described by T. N.
 * Hibbard in A Simple Sorting Algorithm, Journal of
 * the ACM 10:142-50, 1963. The code is a direct
 * translation of Hibbard's pseudocode into C, and
 * retains his variable names, line labels, and use
 * of goto statements.
 *
 * Has built-in driver
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
/* common variables */
int SwapPairs = 0; /* running total of number of swap pairs */
FILE *OutFile;      /* write swap pairs to this file */

/* Bit access macros */
#define SetBit(x,b) ((x) |=  (1 << (b)))
#define ClrBit(x,b) ((x) &= ~(1 << (b)))
#define TstBit(x,b) ((x) &   (1 << (b)))

void BoseSort ( int N )
{
    unsigned int x, xj, y, yj, j, L;

    fprintf ( OutFile,
              "{ /* Bose-Nelson sort for %d elements */\n", N );

    /* L = ceil(log2(N-1)) */
    L = sizeof(int) * CHAR_BIT - 1;
    for ( ; ! TstBit(N - 1, L); L-- )
                ;
    L += 1;

    /* starting values */
    x = 0;
    y = 1;

  /* Top of loop - x and y are a swap pair */
  A:                          /*--- goto target ---*/
    fprintf ( OutFile, "    swap(%d, %d);\n", x, y );
    SwapPairs += 1;

    j = 0;

  C:                          /*--- goto target ---*/
    xj = TstBit(x, j);
    yj = TstBit(y, j);

    if ( xj == 0 && yj == 0 )
        goto zero;
    else if ( xj && yj )
        goto one;
    else if ( xj == 0 )
        goto first_two;
```

```
    else
        goto two;

zero:                          /*--- goto target ---*/
    SetBit(x, j);
    SetBit(y, j);
    if ( y <= N - 1 )
        goto A;

one:                           /*--- goto target ---*/
    ClrBit(y, j);
    goto A;

two:                           /*--- goto target ---*/
    ClrBit(x, j);
    j += 1;
    goto C;

first_two:                     /*--- goto target ---*/
    ClrBit(x, j);
    ClrBit(y, j);
    if ( j == L )
        return;
    j += 1;
    if ( TstBit(y, j) )
        goto D;
    SetBit(x, j);
    SetBit(y, j);
    if ( y > N - 1 )
        goto first_two;
    if ( y < N - 1 )
        j = 0;

D:                             /*--- goto target ---*/
    ClrBit(x, j);
    SetBit(y, j);
    goto A;
}

int main ( int argc, char *argv[] )
{
    int i, n;
    double optimal = 0.0;

    if ( argc != 3 || ( n = atoi ( argv[1] )) <= 1 )
    {
        fprintf ( stderr,
                  "Usage: bose-nel N outfile, where N > 1\n" );
        return ( EXIT_FAILURE );
    }

    if (( OutFile = fopen ( argv[2], "w" )) == NULL)
    {
        fprintf ( stderr, "Can't open %s\n", argv[2] );
        return ( EXIT_FAILURE );
    }
```

```
    BoseSort ( n );

    fprintf ( OutFile,
                " } /* There are %d swaps */\n", SwapPairs );
    fclose ( OutFile );

    printf ( "There were %d swaps generated.\n", SwapPairs );

    /*
     * Report theoretical optimum.
     * Start by computing log10(n!)
     */
    for ( i = 1; i <= n; i++ )
        optimal += log10 ( i );

    /*
     * Convert to ceil(log2(n!)) using
     * fact that log10(2) = .30103
     */
    optimal = ceil ( optimal / 0.30103 );

    printf ( "The best theoretical sort uses %.0f swaps.\n",
           (float) optimal );

    return ( EXIT_SUCCESS );
}
```

bose-nel.c 程序实际上不会执行排序。它代之以产生一个 C 代码段，并且打算把该代码段放在另一个程序中。例如，如果我们要求 bose-nel.c 为 4 个元素生成 Bose-Nelson 排序，就会得到以下代码：

```
{ /* Bose-Nelson sort for 4 elements */
    swap(0, 1);
    swap(2, 3);
    swap(0, 2);
    swap(1, 3);
    swap(1, 2);
} /* There are 5 swaps */
```

实际的工作是由 swap( ) 完成的。它必须检查传递给它的两个数据项，如果合适就交换它们。注意这个交换对序列是怎样与以前在图 5-2 中显示的网络排序相对应的。

像 Bose-Nelson 排序这样的网络排序的主要优点是：如果可以安排同时执行一些比较-交换，网络排序可能会比你可以设计的任何其他排序更快一些。网络排序还使它们自身很好地适合于硬件实现。不过，它也有一些重大的缺点。任何给定的网络排序都只适合于某些 n 值，当比较－交换次数增加时，程序的大小也会增加；100 个数据项的 Bose-Nelson 排序将对 swap( ) 执行 1511 次调用（表 5-6）！因此，网络排序不是一种通用的排序，而最好预备用于数量较少的数据项的固定大小的排序，并且必须非常快地频繁执行。

表 5-6　对于不同 N 值，Bose-Nelson 算法生成的交换对的数量

N	交换对的数量	N	交换对的数量
10	32	100	1511
50	487	1000	57158

5.7 小结：选择一种排序算法

如我们所看到的，给定排序的所有实现都代表某种折衷。不存在单独一种最佳的排序。不过，一些算法一般优于另一些算法。从图 5-3 中可以看出，对于数量较少的记录（少于 1000 条），在希尔排序、堆排序和快速排序之间做选择确实可以通过掷硬币来决定。如果不是对数千条记录进行排序，那么堆排序和快速排序的阶 $N \lg N$ 优于希尔排序的阶 $N^{1.25}$ 的优点不会变得很明显。如果快速排序实现得很糟糕，就应该避免使用它，但是我们见过了一些用于很好地实现它的方式。虽然希尔排序和堆排序都具有极大地独立于其输入的优点——它们确实没有降级的最坏情况——但是良好实现的快速排序不太可能遇到降级情况，要比堆排序和希尔排序运行得更快，并且一般是所选的排序方法。

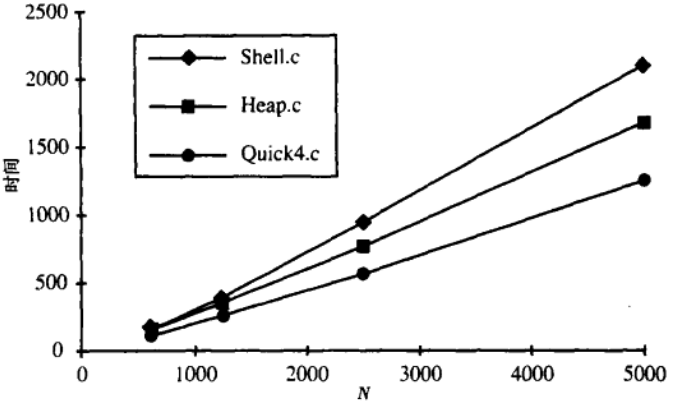


图 5-3 shell. c、heap. c 和 quick4. c 对于大小可变的随机顺序的数据集的运行时间

把这些考虑放在一边，将有可能最佳地使用 ANSI C 的 qsort( )，除非需要对链表排序。不过，明智的做法是：对少数几个数据集测试编辑器的版本，以确保它表现良好。试验正序、逆序和渐增-渐减的记录。还要试验包含许多重复键的数据集；其中所有键都是 0 或 1 的数据集提供了良好的测试。如果编译器的 qsort( ) 对全部 4 种类型的数据集都执行得很好，那么使用它就可能感觉很舒适。程序清单 5-20（qstest. c）中显示了一个用于此目的的简单例程，还有一个例程用于生成以多种方式预先排序的测试数据集（程序清单 5-21，makedata. c）。在编写本书时，有三种流行的基于 IBM PC 的 C 编译器，它们分别是由 Borland、Microsoft 和 Watcom 开发的。其中，只有 Borland C + + 的 qsort( ) 对于各种类型的数据集都有提供同样快的性能。当给定逆序的数据集或者具有许多重复键的数据集时，Microsoft 和 Watcom 提供的 qsort( ) 将会使性能降级。可供使用的编译器库的情况同样可能会发生变化，在发布任何应用程序之前一定要调查它。如果发现你的编译器的 qsort( ) 存在缺陷，可以使用表现良好的 quick5. c（参见程序清单 5-11）替代 qsort( )。在预习本章期间使用了这两个程序。

程序清单 5-20 用于测试 qsort( ) 的例程

```
/*--- qstest.c --------------------------- Listing 5-20 --------
 * Test compiler's qsort()
```

```
 * Link with sortsub.c (Listing 5-1)
 *
 * Has built-in driver
 *-------------------------------------------------------------*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sorthdr.h"
#include "sortsub.h"

/*
 * A comparison function
 */
int pCfunc ( Element **L, Element **R )
{
    return ( strncmp ( (*L)->text, (*R)->text, 5 ));
}

void main ( int argc, char *argv[] )
{
    Element **Array;
    int Items = 2000;
    time_t a, b;

    if ( argc != 2 && argc != 3 )
    {
        fprintf ( stderr, "Usage: qstest infile [maxitems]\n" );
        return;
    }

    if ( argc == 3 )
        Items = atoi ( argv[2] );

    if (( Items = LoadArray ( argv[1], Items, &Array )) == -1 )
        return; /* Couldn't load file */

    time ( &a );
    qsort( Array, Items, sizeof(Element *),
          (int (*) (const void *, const void*))pCfunc );
    time ( &b );
    printf ( "Sorted %d items in %.0lf seconds.\n",
                Items, difftime ( b, a ));
}
```

在 main() 中解释了下一个程序的使用：

程序清单 5-21　用于测试排序的数据生成器

```
/*--- makedata.c ------------------------ Listing 5-21 --------
 * Generate data for sort tests
 *
 * For usage, see main()
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
```

```
#include <ctype.h>
#include <string.h>

void main ( int argc, char *argv[] )
{
    int i, from, to, step;

    if ( argc != 3 || strchr ( "fFrRbBuU?", *argv[1] ) == NULL)
    {
        printf ( "usage: makedata [f|r|b|?] n > outfile\n"
                 "    f = forward-ordered\n"
                 "    r = reverse-ordered\n"
                 "    u = up and down-ordererd\n"
                 "    b = binary (mixed ones and zeros)\n"
                 "    ? = random-ordered\n" );

        return;
    }

    if ( *argv[1] == '?' )
    {
        to = atoi ( argv[2] );

        for ( i = 0; i != to; i++ )
            printf ( "%05d\n", abs ( rand() ));

        return;
    }

    if ( tolower ( *argv[1] ) == 'b' )
    {
        to = atoi ( argv[2] );

        for ( i = 0; i != to; i++ )
            printf ( "%05d\n", abs(rand()) % 2 );
        return;
    }

    if ( tolower ( *argv[1] ) == 'u' )
    {
        to = atoi ( argv[2] ) / 2;
        for ( i = 0; i < to; i++ )
            printf ( "%05d\n", i );
        for ( i = to; i >= 0; i-- )
            printf ( "%05d\n", i );

        return;
    }

    if ( tolower ( *argv[1]) == 'f' )
        step = 1;
    else
        step = -1;

    i = atoi ( argv[2] );

    if ( step == 1 )
```

```
    {
        from = 0;
        to = i;
    }
    else
    {
        from = i;
        to = 0;
    }

    for ( i = from; i != to; i += step )
        printf ( "%05d\n", i );
}
```

5.8 资源和参考资料

Bose、R. C. 和 R. J. Nelson. "A Sorting Problem." *Journal ACM*, Vol. 9, pp. 282-296, 1962. 本书介绍了本章中讨论的网络排序算法的基础知识。

Hibbard, T. N. "A Simple Sorting Algorithm." *Journal ACM*, Vol. 10, pp. 142-150, 1963. 这篇文章介绍了 bose-nel. c 中使用的经过改进的 Bose-Nelson 排序对生成器。

Hoare, C. A. R. "Quicksort." *Computer Journal*, Vol. 5, pp. 10-15, 1962. 这篇文章非常详细地总结和分析了作者以前关于底层算法的非常简短的描述。

Knuth, D. E. The Art of *Computer Programming*. Vol. 3. *Sorting and Searching*. Reading, MA: Addison-Wesley 1973. 本书是针对本章的两份关键参考资料之一。Knuth 对每种算法的运行时间提供了详细的数学分析，还提供了许多有趣的历史细节。他包括了大量额外的材料，如果你对以下方面感兴趣：(1) 为希尔排序计算 h 的其他方法；(2) 用于排序网络的其他方法，那么这些材料就是有用的。

Sedgewick, R. *Algorithms in C*. Reading, MA: Addison-Wesley, 1990. 本书是针对本章的另一份关键参考资料。Sedgewick 是快速排序的无可争议的大师，他的图书描述了关于其优化的细节，可以轻松获得它。他还探讨了我们在本书中没有描述的许多其他排序算法，包括外部排序。Sedgewick 的图书的唯一缺点是：其中的代码示例并没有总是正确地进行转换，以使用 C 语言的基于 0 的数组，并且如本章开头所讨论的，这样将需要修改它们，以便在任何实际的应用程序中使用。

Shell, D. L. *Communications of the ACM*. Vol. 2, pp. 30-32, 1959. 这是希尔排序的原始描述，其名称正来源于此。

Taylor, J. "Quicksorting Linked Lists." *C Gazette*, Vol. 5, No. 6, pp. 17-20, 1991. 这是用于链表的快速排序的良好讨论，并且本章中的代码遵循 Taylor 的实现。不过，注意：Taylor 的文章中发布的代码并没有正确地实现稳定的快速排序。为了修正他的代码，必须修改内层循环中的比较，以使用小于号，而不是小于或等于号。

第6章　树

许多应用程序中的关键问题是：快速定位特定排序项的能力。第3章中讨论的散列表技术代表一组解决这个问题的算法，而第4章中描述的字符串查找技术则是另一组这样的算法。第三类算法中包含树算法。虽然它们的实现比另外两种技术更复杂，但是它们提供了偏置法的优点，这使得它们在某些情况下非常有吸引力。其中特别令人感兴趣的是，可以相对容易地修改树，使得它们可以在辅助存储器中存储大量的数据。在本章中，我们将讨论4种树算法。前三种（二叉树、红黑树和伸展树）主要适用于内存中的工作，而第4种（B树）打算用于辅助存储器，比如硬盘。

6.1　二叉树

二叉树是最简单的树算法，但是它们构成了其他算法的基础。通过它们的优、缺点可以深入了解其他树算法。如图6-1所示，二叉树由节点组成，其中包含至少三个数据项：两个指向其他节点的指针以及一些用户数据。在这种情况下，树包含4个节点。节点2包含数据（它的名称）以及指向节点1和3的指针，而节点3则指向节点4。

树的描述频繁使用与家族有关的术语，比如：**子节点**（child）、**父节点**（parent）、**祖父节点**（grandparent）和**曾祖父节点**（great-grandparent）。因此，我们将把节点1称为节点2的左子节点，把节点3称为节点4的父节点，以及把节点4称为节点2的孙节点。显而易见的是，一个节点可能具有0个、1个或2个子节点。这个定义是递归式的，这是由于给定节点的任何一个子节点或者两个子节点本身就可以是二叉树。

二叉树的**根**（root）是一个没有父节点的节点，在描绘树时，通常使这个节点作为图中最上面的节点出现。任何给定节点的**高度**（height）或**深度**（depth）是将其与根节点隔开的节点数。树的高度（深度）与最高（最深）节点的高度（深度）相同。例如，在图6-1中，节点2的高度为0，节点1和3的高度为1，节点4的高度为2，并且树的总高度为2。

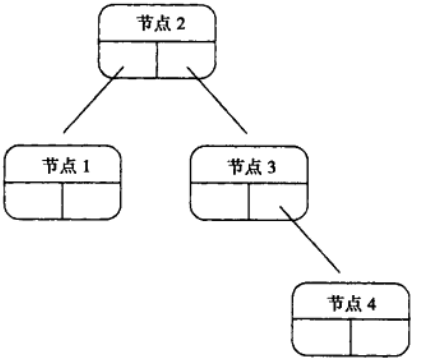


图6-1　简单的二叉树

除了它的拓扑之外，二叉树的另一个重要特征是它的**遍历顺序**（traversal order）。你往往希望访问树中的所有节点，确保以某种一致的顺序只访问每个节点一次。有三种可能的遍历顺序，它们都是递归式的：

前序（preorder）：访问（1）节点、（2）左子节点、（3）右子节点。

中序（inorder）：访问（1）左子节点、（2）节点、（3）右子节点。

后序（postorder）：访问（1）左子节点、（2）右子节点、（3）节点。

非常重要的是，访问一个子节点意味着我们将递归地访问该子节点的所有子节点。因此，即

使定义很简单，全部三种遍历顺序最终都会包括访问树中的所有节点。这三种遍历顺序可以在不同时间用于不同的目的。

考虑图 6-2。这里，解析了表达式 2 + （6 * 7），并将其存储在一棵二叉树中（删除了空指针和节点周围的方框）。如果我们希望以其自然形式打印表达式，将使用中序遍历。不过，通过后序方式遍历树，可以将表达式转换为 RPN（Reverse Polish Notation，逆波兰表示法），并且可以轻松地对其进行求值。

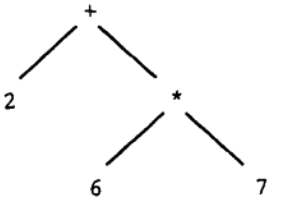


图 6-2　表达式 2 + （6 * 7）的二叉树

一定要小心，必须严格遵循树的遍历顺序，并且不应该让空的子树蒙蔽你的眼睛。图 6-3 显示了一棵高而瘦的二叉树。在右边重绘了该树，使之带有空的子树，以便可以更容易地查看顺序。中序遍历将以 1-2-3-4 的顺序访问节点，即使缺少许多子树而可能使你认为看到的是一种不同的顺序，认识到这一点很重要。添加空的子树使得能够更容易一点地查看正确的顺序。

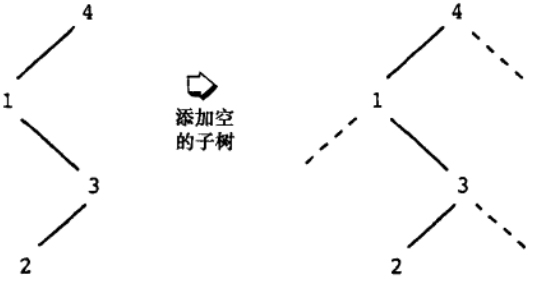


图 6-3　一棵具有许多空子树的高而瘦的树

当你想使用二叉树存储大量数据时，只需遵循一条简单的规则：当使用中序遍历时，在每个节点中存储的数据的键将具有递增的顺序。通过遵循这条规则，现在可以把二叉树视作具有图 6-4 中所示的形式。通常把这种树称为**二叉查找树**（binary search tree）。在该图中，所有的左子节点所具有的键都在其父节点的键之前；而右子节点所具有的键都至少等于父节点的键。

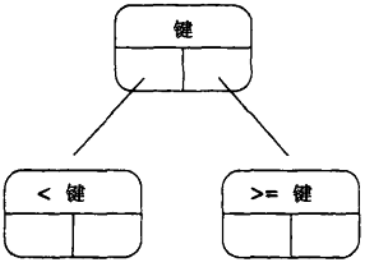


图 6-4　二叉查找树

现在让我们检查二叉树的实现。在程序清单 6-1 和程序清单 6-2 中可以找到二叉树例程的完整源代码，它们分别是 bintree. c 和 bintree. h。可以以三种不同的方式编译该代码。我们首先将检查其在未定义 REDBLACK 或 SPLAY 开关的情况下的编译版本。首先将利用以下代码在 bintree. h 中构建树节点的定义（暂时忽略 RBONLY 宏内的文本；它将不会用于我们的二叉查找树）。

```
#define BINTREE_STUFF(x) struct x *link[2] RBONLY(;int red)
typedef struct sBnode {
    BINTREE_STUFF(sBnode);
} Bnode;
```

这样，实际上，Bnode 将具有以下形式：

```
typedef struct sBnode {
    struct sBnode *link[2];
} Bnode;
```

这提供了构建树所需的两个子指针。我们实现的所有通用的树例程都将操纵 Bnode。不过，其中没有任何数据的节点将用处不大。在测试驱动程序中的 bintree.c 中可以找到用于定义带有数据的节点的方法：

```
typedef struct sMynode {
    BINTREE_STUFF(sMynode);
    char text[20];
} Mynode;
```

这将扩展成如下形式：

```
typedef struct sMynode {
    struct sMynode *link[2];
    char text[20];
} Mynode;
```

换句话说，Mynode 以与 Bnode 相同的方式开始于两个指针。如果我们使用 C++编程，将简单地把 Bnode 继续放进 Mynode 中。在 C 中，我们用手工完成 C++编译器自动完成的工作；我们在定义结构时使得它们以完全相同的形式开始，然后我们把额外的字节附加到其中一个结构的末尾。由于 ANSI C 保证数据指针具有共同的大小，自由地把指针互换为 Bnode 和 Mynode 是安全的。我们现在可以定义一个结构，它保存用于描述树的根所需的所有信息：

```
typedef struct sBintree {
    Bnode *DummyHead; /* points to node above the root */
    CompFunc Compare; /* compares two nodes */
    int DuplicatesOk; /* does tree take duplicates? */
    size_t NodeSize;  /* size (bytes) of a node */
} Bintree;
```

我们将使用 DummyHead 指向一个节点，它的右子节点实际上是树根。如果根具有父节点，那么许多树操作都会得到简化。Compare 是一个指针，它指向用户定义的函数，它用于比较两个节点，并且返回一个负值、0 或正值，这取决于它们的相对顺序（以与从 strcmp() 返回值完全相同的方式）。DuplicatesOk 告诉树例程是否允许树中的数据项具有重复的键。我们在 NodeSize 中存储用户节点的大小，因为 sizeof（Bnode）将不会考虑用户的数据区域。

程序清单 6-1　用于二叉树的基本例程

```
/*--- bintree.c -------------------------- Listing 6-1 ---------
 * Binary tree routines. Provides plain binary search,
 * red-black, and self-adjusting (splay) trees.
 *
 * Switches:        REDBLACK - Use red-black variant
 *                  SPLAY    - Use splay tree variant
 *                  TEST     - Include a test driver
 *                  ALTDRAW  - Use only standard drawing symbols
 *                             when drawing tree diagrams
 *-------------------------------------------------------------*/
```

```
#define TEST

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "bintree.h"

/* A safe malloc() */
static void * tmalloc(size_t size)
{
    void *p;
    if ((p = malloc(size)) == NULL) {
        printf("Out of memory\n");
        exit(1);
    }

    return p;
}

/*
 * Create and initialize a node for the user. 'size' both can
 * and should be greater than sizeof(Bnode) to allow for a
 * data area for the user.
 */
Bnode *InitBintreeNode(size_t size)
{
    Bnode *n;

    n = tmalloc(size);
    n -> link[LEFT] = n -> link[RIGHT]  = NULL;
    RBONLY(n -> red = 0;)

    return n;
}

/* Create an empty tree */
Bintree *NewBintree (Bnode *dummy,
    CompFunc cf,
    int dup_ok,
    size_t node_size)
{
    Bintree *t;

    t = tmalloc(sizeof(Bintree));
    t -> DummyHead = dummy;
    t -> Compare = cf;
    t -> DuplicatesOk = dup_ok;
    t -> NodeSize = node_size;

    return t;
}
#if defined(SPLAY)

/*
 * During a top-down splay, we build up the future left and right
 * sub-trees in trees whose roots are stored in the array LR[].
```

```
 * LRwalk[] retains a current pointer into each of these trees.
 * We are always interested in finding the bottom left node of
 * the right tree or the bottom right node of the left tree.
 * Thus, PUSH(LEFT) starts at LRwalk[LEFT], steps down and to
 * the right until it hits bottom, and then stores the new
 * location in LRwalk[LEFT].
 */
#define PUSH(x) {                                   \
                    Bnode *w;                       \
                    for (w = LRwalk[x];             \
                         w -> link[!(x)];           \
                         w = w -> link[!(x)]);      \
                    LRwalk[x] = w;                  \
                }

int splay(Bintree *t, Bnode *n)
{
    Bnode *s, *ch, *gch;
    Bnode LR[2], *LRwalk[2];
    int s_comp, ch_comp, dir, dir2;

    s = t -> DummyHead -> link[RIGHT];
    if (s == NULL) /* empty tree */
        return 1; /* no match */

    /*
     * Create two empty trees: we place portions of the initial
     * tree onto these two trees as we "splay down" the tree.
     */
    LR[LEFT].link[RIGHT] = NULL;
    LR[RIGHT].link[LEFT] = NULL;

    LRwalk[LEFT]  = &LR[LEFT];
    LRwalk[RIGHT] = &LR[RIGHT];

    /* not really needed */
    LR[LEFT].link[LEFT] = NULL;
    LR[RIGHT].link[RIGHT] = NULL;

    for (;;) {
        /* We are at s. Which way now? First, find s's child */
        s_comp = n ? (t -> Compare)(n, s) : 1;
        dir = s_comp < 0;
        ch = s -> link[dir];
        if (s_comp == 0 || ch == NULL)
            break;

        /* Now, find s's grandchild */
        ch_comp = n ? (t -> Compare)(n, ch) : 1;
        dir2 = ch_comp < 0;
        gch = ch -> link[dir2];

        /*
         * If we've found a match for n (ch_comp==0) or if we've
         * no further to go (gch==NULL), then we're done. ch will
         * be the root of the new tree after reconstruction is
         * complete. This case is the only exit from this loop.
```

```
         */
        if (ch_comp == 0 || gch == NULL) {
            s -> link[dir] = NULL; /* break link betw s and ch */
            LRwalk[!dir] -> link[dir] = s; /* hang s on LR */
            PUSH(!dir);  /* and push LRwalk to bottom */

            s = ch; /* advance s to ch */
            s_comp = ch_comp;
            break; /* proceed to tree reconstruction */
        }

        else { /* split up the tree as described in the text */
            if (dir == dir2) { /* zig-zig */
                s -> link[dir] = ch -> link[!dir];
                ch -> link[!dir] = s;
                ch -> link[dir] = NULL;
                LRwalk[!dir] -> link[dir] = ch;
                        PUSH(!dir);
            }

            else { /* zig-zag */
                s -> link[dir] = NULL;
                LRwalk[!dir] -> link[dir] = s;
                PUSH(!dir);
                ch -> link[dir2] = NULL;
                LRwalk[!dir2] -> link[dir2] = ch;
                PUSH(!dir2);
            }
            s = gch;
        }
    }

    /* put it all together */
    LRwalk[LEFT]  -> link[RIGHT] = s -> link[LEFT];
    LRwalk[RIGHT] -> link[LEFT]  = s -> link[RIGHT];
    s -> link[LEFT]  = LR[LEFT].link[RIGHT];
    s -> link[RIGHT] = LR[RIGHT].link[LEFT];
    t -> DummyHead -> link[RIGHT] = s;
    return s_comp;
}
#endif

/* Find node n in tree t */
Bnode *FindBintree(Bintree *t, Bnode *n)
{
    #if defined(SPLAY)
    if (splay(t, n))
        return NULL; /* exact match not found */
    else
        return t -> DummyHead -> link[RIGHT];

    #else /* plain or red-black */
    Bnode *s;
    int dir;

    s = t -> DummyHead -> link[RIGHT];
    while (s != NULL) {
```

```
        dir = (t -> Compare) (n, s);
        /*
         * If a match, we're done.
         * For Red-Black, must also be a leaf.
         */
        if (dir == 0 RBONLY(&& s -> link[RIGHT] == NULL)
            return s;
        dir = dir < 0;
        s = s -> link[dir];
    }
    return NULL; /* no match */
    #endif
}

#if defined(REDBLACK)
/*
 * Rotate child and grandchild of r along the path
 * specified by searching for n. For example, if n was
 * equal to 3, gc2, or 4, the following rotation occurs:
 *
 *            r                                  r
 *            |                                  |
 *            c                                 gc2
 *          /   \            ==>               /   \
 *        gc1    gc2                          c      4
 *       /  \    /  \                        / \
 *      1    2  3    4                     gc1   3
 *                                        /   \
 *                                       1     2
 *
 * As r may connect to c via either its left or right
 * link, there are actually four symmetric variants.
 *
 * A pointer to the top of the new rotated nodes (in the
 * case above, to gc2) is returned.
 *
 * This routine is complicated by the fact that the routine
 * uses the value of the node n to decide which direction
 * to rotate. This may or may not be the direction the caller
 * has is mind. Rather than require the caller to specify
 * the direction of the rotation, it seemed easier to allow
 * the caller to specify whether to go in the direction of n
 * or away from it. This is done by the last argument to the
 * function, flip_mode. The caller can indicate that either
 * or both of the directions to child and grandchild should
 * be reversed during the rotation.
 */

#define NO_FLIP  0
#define FLIP_GCH 1
#define FLIP_CH  2
Bnode *rotate(Bintree *t, Bnode *n, Bnode *r, int flip_mode)
{
    Bnode *ch, *gch;
    int ch_dir, gch_dir;

    /* Identify child and grandchild */
    ch_dir = (t -> Compare) (n, r) < 0;
```

```
    if (flip_mode & FLIP_CH)
        ch_dir = !ch_dir;
    if (r == t -> DummyHead) /* special condition */
        ch_dir = RIGHT;
    ch = r -> link[ch_dir];

    gch_dir = (t -> Compare) (n, ch) < 0;
    if (flip_mode) {
        if (flip_mode == FLIP_GCH)
            gch_dir = !gch_dir;
        else
            gch_dir = flip_mode & 1;
    }
    gch = ch -> link[gch_dir];

    /* rotate: now move pointers */
    ch -> link[gch_dir] = gch -> link[!gch_dir];
    gch -> link[!gch_dir] = ch;
    r -> link[ch_dir] = gch;

    return gch;
}

/*
 * Take care of colors and balance. It will color the current
 * location red, the current location's children black, and
 * then look to see if two consecutive red nodes have been
 * created. If so, a single or double rotation will be done
 * to fix the tree.
 */
void split(Bintree *t,  /* tree                    */
           Bnode *n,    /* node being inserted     */
           Bnode **c,   /* current location        */
           Bnode **p,   /* its parent              */
           Bnode *g,    /* its grandparent         */
           Bnode *gg)   /* its great-grandparent */
{
    if (t -> DummyHead -> red) {
        fputs("dummyhead was red!!\n", stdout);
        t -> DummyHead -> red = 0;
    }
    (*c) -> red = 1;
    if ((*c) -> link[LEFT])
        (*c) -> link[LEFT] -> red = 0;
    if ((*c) -> link[RIGHT])
        (*c) -> link[RIGHT] -> red = 0;

    /*
     * Check to make sure we haven't created two red
     * links in a row. If we have, we must rotate.
     */
    if ((*p) -> red) {
        g -> red = 1;
        /*
         * If the red links don't point in the same direction,
         * then will need a double rotation. The lower half
         * is around the grandparent and then the upper half
```

```
         * is around the great-grandparent.
         */
        if (((t -> Compare) (n, g) < 0) !=
            ((t -> Compare) (n, *p) < 0))
            *p = rotate(t, n, g, NO_FLIP);

        /* Same for both single and double rotations. */
        *c = rotate(t, n, gg, NO_FLIP);
        (*c) -> red = 0;
    }

    t -> DummyHead -> link[RIGHT] -> red = 0;
}
#endif

/*
 * Delete node n from tree t. Returns a pointer to the
 * deleted node -- it should then be freed or otherwise
 * destroyed. The versions for the binary tree and the
 * red-black tree are very different, due to the balancing
 * problems that the red-black version must handle.
 */
#if defined(REDBLACK)
Bnode * DelBintree (Bintree *t, Bnode *n)
{
    /*
     * The goal is to arrive at a leaf with a red parent.
     * Thus, we force this by dragging a red node with us
     * down the tree, re-arranging the tree to keep its
     * balance as we go. All the rearrangements keep the tree
     * balanced, so if we cancel the deletion or don't find
     * the specified node to delete, we can just quit.
     */

    Bnode *s, *p, *g;
    int dir, next_dir;

    g = NULL;
    p = t -> DummyHead;
    s = p -> link[RIGHT];
    dir = RIGHT;

    /*
     * First, check on the root. It must exist, have children,
     * and either it or one of its children must be red. We can
     * just paint the root red, if necessary, as this will
     * affect the black height of the entire tree equally.
     */
    if (s == NULL)
        return NULL;

    /* Check to make sure the root isn't an only child. */
    if (s -> link[LEFT] == NULL) {
        if ((t -> Compare)(n, s) == 0) {
            /* deleting the root */
```

```
        p -> link[RIGHT] = NULL;
        return s;
    }
    else
        return NULL;
}

/* Now, either the root or one of its kids must be red */
if (!s -> link[LEFT]  -> red &&
    !s -> link[RIGHT] -> red)
    s -> red = 1; /* Just color the root red */

/*
 * Now, march down the tree, always working to make sure
 * the current node is red. That way, when we do arrive
 * at a leaf, its parent will be red, making the leaf
 * very easy to delete (just drop the leaf, and replace
 * its (red) parent with its (black) sib.)
 */
for (;;) {
    /*
     * If we're at a leaf, we're done.
     */
    if (s -> link[LEFT] == NULL)
        break;

    /*
     * Where are we going next?
     */
    next_dir = (t -> Compare) (n, s) < 0;

    /*
     * If the current node or the next node
     * is red, we can advance.
     */
    if (s -> red || s -> link[next_dir] -> red)
        ;

    /*
     * (If the current node is black)
     * (and the next node is black)
     * but the next node's sib is red ...
     *
     * Then rotate from parent towards the red child. This
     * will lower the current node, and give us a new
     * grandparent (the old parent) and a new
     * parent (the sib that was red). We the paint the
     * current node red and the new parent is painted black.
     */
    else if (s -> link[!next_dir] -> red) {
        g = p;
        p = rotate(t, s -> link[next_dir], p, FLIP_GCH);
        s -> red = 1;
        p -> red = 0;
    }
```

```
/*
 * (If the current node is black)
 * (and its left child is black)
 * (and its right child is black) ...
 *
 * then (a) the current node's parent must be red (we
 * never advance unless we are leaving a red node),
 * (b) its sib must be black (because the parent is red),
 * and (c) we need to color the current node red. To
 * make this possible, we color the current node red,
 * the parent black and then check for tree imbalances.
 * Two cases exist...
 */
else {
    Bnode *sib;
    if (!p -> red)
        fprintf(stdout, "Parent not red in case 2!\n");
    sib = p -> link[!dir];
    if (sib -> red)
        fprintf(stdout, "Sib not black in case 2!\n");
    if (sib -> link[LEFT] == NULL) {
        fprintf(stdout, "Sib has no kids in case 2!\n");
        return NULL;
    }

    s -> red = 1;
    p -> red = 0;

    /*
     * First case: black sib has two black kids. Just
     * color the sib red. In effect, we are reversing
     * a simple color flip.
     */
    if (!sib -> link[LEFT]  -> red &&
        !sib -> link[RIGHT] -> red)
        sib -> red = 1;

    /*
        * Second case: black sib has at least one red kid.
        * (It makes no difference if both kids are red.)
        * We need to do either a single or double rotation
        * in order to re-balance the tree.
        */
       else {
           int redkid_dir;

           if (sib -> link[LEFT] -> red)
               redkid_dir = LEFT;
           else
               redkid_dir = RIGHT;

           if (!dir == redkid_dir) {
               sib -> red = 1;
               sib -> link[redkid_dir] -> red = 0;
               g = rotate(t, n, g, FLIP_GCH);
           }
           else {
```

```
                    rotate(t, n, p, FLIP_CH + redkid_dir);
                    g = rotate(t, n, g, FLIP_GCH);
                }
            }
        }

        /* advance pointers */
        dir = next_dir;
        g = p;
        p = s;
        s = s -> link[dir];
    }

    /* Make the root black */
    t -> DummyHead -> link[RIGHT] -> red = 0;

    /* Delete it, if a match. Parent is red. */
    if ((t -> Compare)(s, n) == 0) {
        if (!p -> red && p != t -> DummyHead)
            fprintf(stdout, "Parent not red at delete!\n");
        g -> link[(t -> Compare)(s, g) < 0] =
            p -> link[(t -> Compare)(s, p) >= 0];
        free (p); /* release internal node that we created */
        return s;
    }
    else return NULL;
}
#elif defined(SPLAY) /* Splay tree version */
Bnode *DelBintree (Bintree *t, Bnode *n)
{
    Bnode *save, *t2;

    if (splay(t, n))
        save = NULL; /* match not found */
    else {
        save = t -> DummyHead -> link[RIGHT];
        t2 = save -> link[RIGHT];
        if (t -> DummyHead -> link[RIGHT] = save -> link[LEFT])
        {   /* '=' and not '==' is correct on previous line */
            splay(t, NULL);
            t -> DummyHead -> link[RIGHT] -> link[RIGHT] = t2;
        }
        else
            t -> DummyHead -> link[RIGHT] = t2;

    }
    return save;
}

#else /* Binary tree version */
Bnode * DelBintree (Bintree *t, Bnode *n)
{

    Bnode *p, *s, *save;
    int dir, dir_old;

    p = t -> DummyHead;
```

```
    s = p -> link[RIGHT];
    dir_old = dir = RIGHT;

    /* Look for a match */
    while (s != NULL && (dir = (t->Compare)(n, s)) != 0) {
        p = s;
        dir = dir < 0;
        dir_old = dir;
        s = p -> link[dir];
    }

    if (s == NULL)
        return NULL; /* no match found */

    save = s;
    /*
     * First case: if s has no right child, then replace s
     * with s's left child.
     */
    if (s -> link[RIGHT] == NULL)
        s = s -> link[LEFT];
    /*
     * Second case: if s has a right child that lacks a left
     * child, then replace s with s's right child and
     * copy s's left child into the right child's left child.
     */
    else if (s -> link[RIGHT] -> link[LEFT] == NULL) {
        s = s -> link[RIGHT];
        s -> link[LEFT] = save -> link[LEFT];
    }
    /*
     * Final case: find leftmost (smallest) node in s's right
     * subtree. By definition, this node has an empty left
     * link. Free this node by copying its right link to
     * its parent's left link and then give it both of s's
     * links (thus replacing s).
     */
    else {
        Bnode *small;

        small = s -> link[RIGHT];
        while (small -> link[LEFT] -> link[LEFT])
            small = small -> link[LEFT];
        s = small -> link[LEFT];
        small -> link[LEFT] = s -> link[RIGHT];
        s -> link[LEFT] = save -> link[LEFT];
        s -> link[RIGHT] = save -> link[RIGHT];
    }

    p -> link[dir_old] = s;

    RBONLY(s -> red = 0;)

    return save;
}
#endif
```

```
/* Insert node n into tree t */
int InsBintree (Bintree *t, Bnode *n)
{
    #if defined(REDBLACK)
    int p_dir;
    Bnode *p, *s;
    Bnode *g   = NULL;
    Bnode *gg  = NULL;
/* Search until we find a leaf. */
p = t -> DummyHead;
p_dir = RIGHT; /* direction from p to s */
s = p -> link[RIGHT];

if (s) {
    Bnode *temp;
    int dir;

    /* Look for a leaf, splitting nodes on the way down */
    while (s -> link[RIGHT] != NULL) {
        if (s -> link[LEFT]  -> red &&
            s -> link[RIGHT] -> red)
            split(t, n, &s, &p, g, gg);
        gg = g;
        g  = p;
        p = s;
        p_dir = (t -> Compare) (n, s) < 0;
        s = s -> link[p_dir];
    }

    dir = (t -> Compare) (n, s);
    if (t -> DuplicatesOk == 0 && dir == 0)
        return TREE_FAIL; /* duplicate - not allowed */

    /*
     * Must replace s with a new internal node that has as
     * its children s and n. The new node gets the larger of
     * s and n as its key. The new node gets painted red, its
     * children are black. Coloring is done by split().
     */
    temp = tmalloc(t -> NodeSize);
    dir = dir < 0;
    memcpy(temp, dir ? s : n, t -> NodeSize);
    temp -> link[dir]  = n;
    temp -> link[!dir] = s;
    n = temp;
}

/* Add the new node */
p -> link[p_dir] = n;

/* Color this node red and check red-black balance */
split(t, n, &n, &p, g, gg);
return TREE_OK;

#elif defined(SPLAY)
int dir;
```

```
    Bnode *r;

    dir = splay(t, n);
    if (dir == 0 && t -> DuplicatesOk == 0)
        return TREE_FAIL;
    r = t -> DummyHead -> link[RIGHT];

    if (r == NULL) /* first node? */
        t -> DummyHead -> link[RIGHT] = n;
    else {
        dir = dir < 0;
        n -> link[dir] = r -> link[dir];
        r -> link[dir] = NULL;
        n -> link[!dir] = r;
        t -> DummyHead -> link[RIGHT] = n;
    }
    return TREE_OK;

    #else /* plain binary tree */
    int p_dir;
    Bnode *p, *s;

    /* Search until we find an empty arm. */
    p = t -> DummyHead;
    p_dir = RIGHT; /* direction from p to s */
    s = p -> link[RIGHT];

    while (s != NULL) {
        p = s;
        p_dir = (t -> Compare) (n, s);
        if (p_dir == 0 && t -> DuplicatesOk == 0)
            return TREE_FAIL; /* duplicate */
        p_dir = p_dir < 0;
        s = s -> link[p_dir];
    }

    /* Add the new node */
    p -> link[p_dir] = n;
    return TREE_OK;
    #endif
}

/*
 * Recursive tree walk routines. The entry point is
 * WalkBintree. It will do an inorder traversal of the
 * tree, call df() for each node and leaf.
 */
void rWalk(Bnode *n, int level, DoFunc df)
{
    if (n != NULL) {
        rWalk(n -> link[LEFT], level + 1, df);
        df(n, level);
        rWalk(n -> link[RIGHT], level + 1, df);
    }
}

int WalkBintree(Bintree *t, DoFunc df)
```

```
{
    if (t -> DummyHead -> link[RIGHT] == NULL) {
        fputs("Empty tree\n", stdout);
        return TREE_FAIL;
    }

    rWalk(t -> DummyHead -> link[RIGHT], 0, df);
    return TREE_OK;
}

#if defined(TEST)
/*
 * Test driver
 */

#define BUFLEN 100

/* Our binary tree is made up of these */
typedef struct sMynode {
    /* A copy of the items in a Bnode */
    BINTREE_STUFF(sMynode);

    /*
     * Now for the user's part of the structure. We could put
     * anything here. For these routines, a simple text area.
     */
    char text[20];
} Mynode;

int LoadString(Bintree *t, char *string)
{
    Mynode *m;

    m = (Mynode *) InitBintreeNode(sizeof(Mynode));
    strncpy(m->text, string, sizeof(m->text));
    m->text[sizeof(m->text) - 1] = 0;

    return InsBintree(t, (Bnode *) m);
}
void FindString(Bintree *t, char *string)
{
    Mynode m, *r;
    strncpy(m.text, string, sizeof(m.text));
    m.text[sizeof(m.text) - 1] = 0;
    if ((r = (Mynode *) FindBintree(t, (Bnode *) &m)) == NULL)
        puts(" Not found.\n");
    else
        printf(" Found '%s'.\n", r -> text);
}

void DeleteString(Bintree *t, char *string)
{
    Mynode m, *n;
    strncpy(m.text, string, sizeof(m.text));
    m.text[sizeof(m.text) - 1] = 0;
    n = (Mynode *) DelBintree(t, (Bnode *) &m);
    if (n)
```

```
        free (n);
    else
        fprintf(stdout, " Did not find '%s'.\n", string);
}

void LoadFile(Bintree *t, char *fname)
{
    FILE *infile;
    char buffer[BUFLEN], *s;
    int i = 0, j = 0;

    if ((infile = fopen(fname, "r")) == NULL) {
        fputs(" Couldn't open the file.\n", stdout);
        return;
    }

    while (fgets(buffer, BUFLEN, infile)) {
        s = buffer + strlen(buffer);
        while(iscntrl(*s))
            *s-- = 0;
        if (buffer[0] == ';') /* a comment */
            ;
        else if (buffer[0] == '-' && buffer[1] != 0) {
            DeleteString(t, buffer+1);
            j++;
        }
        else {
            LoadString(t, buffer);
            i++;
        }
    }

    fclose(infile);
    printf("Loaded %d items and deleted %d from %s.\n",
        i, j, fname);
}

/*
 * A sample action function: it prints out the data
 * at each node along with the node's level in the tree
 */
int ShowFunc(void *m, int level)
{
    RBONLY(if (((Mynode *)m) -> link[LEFT] == NULL))
        fprintf(stdout, "%s (%d)\n",
        ((Mynode *)m) -> text, level);

    return TREE_OK;
}


/*
 * A pair of functions to print the tree as a diagram.
 */

#if !defined(ALTDRAW)
```

```
  #define TOP '+'
  #define BOT '+'
  #define HOR '-'
  #define VRT '|'
#else
  #define TOP '/'
  #define BOT '\\'
  #define HOR '-'
  #define VRT '|'
#endif

#if defined(REDBLACK)
  #if !defined(ALTDRAW)
    #define RTOP '+'
    #define RBOT '+'
    #define RHOR '-'
    #define RVRT '|'
  #else
    #define RTOP '*'
    #define RBOT '*'
    #define RHOR '#'
    #define RVRT '#'
  #endif
#endif

#define DRAWBUF 100
char draw[DRAWBUF];
char work[DRAWBUF * 2];
int maxdepth;
RBONLY(int blackheight;)
RBONLY(int maxblack;)
FILE *outfile;

void xrWalk(Bnode *n, int level)
{
    int i;

    if (n != NULL) {
        /* Monitor */
        if (level > maxdepth)
            maxdepth = level;
        RBONLY(if (!n -> red) blackheight++;)

        /*
         * Go right
         */
        draw[level * 2] = TOP;
        #if defined(REDBLACK)
        if (n -> link[RIGHT] && n -> link[RIGHT] -> red)
            draw[level * 2] = RTOP;
        #endif
        draw[level * 2 + 1] = ' ';
        xrWalk(n -> link[RIGHT], level + 1);

        /*
         * Show current node
         */
```

```
        strncpy(work, draw, level * 2);
        if (level > 0) {
            int c;

            c = work[0];
            for (i = 2; i < level * 2; i += 2)
                #if !defined(REDBLACK)
                if (work[i] == c)
                #else
                if (((c == TOP || c == RTOP) &&
                    (work[i] == TOP || work[i] == RTOP)) ||
                    ((c == BOT || c == RBOT) &&
                    (work[i] == BOT || work[i] == RBOT)))
            #endif
                work[i - 2] = ' ';
            else
                c = work[i];

        work[level * 2 - 1] =
            RBONLY(((Mynode *)n) -> red ? RHOR :)
            HOR;

        for (i = 0; i < level * 2 - 2; i += 2)
            if (work[i] != ' ') {
                #if !defined(REDBLACK)
                work[i] = VRT;
                #else
                if (work[i] == TOP || work[i] == BOT)
                    work[i] = VRT;
                else
                    work[i] = RVRT;
                #endif
            }
    }

    sprintf(work + level * 2, "%s (%d)",
                        ((Mynode *)n)->text, level);
    fputs(work, outfile);

    #if defined(REDBLACK)
    if (n -> link[LEFT] == NULL) { /* leaf */
        if (maxblack < 0)
            maxblack = blackheight;
        else if (maxblack != blackheight)
            fprintf(outfile, "  Leaf has black height %d!",
            blackheight - 1);
    }
    #endif
    fputs("\n", outfile);

    /*
     * Go left
     */
    draw[level * 2] = BOT;
    #if defined(REDBLACK)
    if (n -> link[LEFT] && n -> link[LEFT] -> red)
        draw[level * 2] = RBOT;
```

```
    #endif
    draw[level * 2 + 1] = ' ';
    xrWalk(n -> link[LEFT], level + 1);
        RBONLY(if (!n -> red) blackheight--;)
    }
}

int xWalkBintree(Bintree *t, char *name, char *mode)
{
    if (t -> DummyHead -> link[RIGHT] == NULL) {
        fputs("Empty tree\n", stdout);
        return TREE_FAIL;
    }

    maxdepth = -1;
    RBONLY(blackheight = 0;)
    RBONLY(maxblack = -1;)

    outfile = stdout;
    if (name) {
        outfile = fopen(name, mode);
        if (outfile == NULL) {
            fprintf(stdout, "Can't open %s.\n", name);
            name = NULL;
            outfile = stdout;
        }
    }

    xrWalk(t -> DummyHead -> link[RIGHT], 0);
    #if defined(REDBLACK)
    fprintf(outfile, "Max depth %d, black height %d.\n",
        maxdepth, maxblack - 1);
    #else
    fprintf(outfile, "Max depth %d.\n", maxdepth);
    #endif

    if (name)
        fclose(outfile); /* a real file */
    else
        fflush(outfile); /* stdout */

    return TREE_OK;
}

int compare_length = 0;
int CompareFunc(void *n1, void *n2)
{
    if (compare_length)
        return strncmp(((Mynode *)n1)->text,
        ((Mynode *)n2)->text,
        compare_length);
    else
        return strcmp(((Mynode *)n1)->text,
        ((Mynode *)n2)->text);
}
```

```
main(int argc, char **argv)
{
    char inbuf[BUFLEN], *s;
    Bintree *tree;
    Mynode *dummy;
    FILE *logfile = NULL;

    #if defined(REDBLACK)
    printf("Red-black binary tree test driver.\n"
        "  Lines to red nodes are drawn with %c%c%c lines and\n"
        "  lines to black nodes are drawn with %c%c%c lines.\n",
          RHOR, RHOR, RHOR, HOR, HOR, HOR);
    #elif defined(SPLAY)
    printf("Splay tree test driver.\n");
    #else
    printf("Plain binary tree test driver.\n");
    #endif

    /* create a dummy node for the tree algorithms */
    dummy = (Mynode *) InitBintreeNode(sizeof(Mynode));
    dummy->text[0] = 0; /* must contain valid data */

    /* create a tree */
    tree = NewBintree((Bnode *) dummy,
                      CompareFunc, 1, sizeof(Mynode));

    for (;;) {
        fputs("Action (? for help): ", stdout);
        fflush(stdout);
        fgets(inbuf, BUFLEN, stdin);
        s = inbuf + strlen(inbuf);
        while(iscntrl(*s))
            *s-- = 0;

        if (logfile)
            fprintf(logfile, "%s\n", inbuf);

        switch (inbuf[0]) {
            case '?':
                fputs(
                    "@file     - Load strings in file to tree\n"
                    "a string  - Add string to tree\n"
                    "c nn      - Compare only first nn chars\n"
                     "d string  - Delete string from tree\n"
                     "dup [0|1] - Disallow/allow duplicates\n"
                     "f string  - Find string in tree\n"
                     "l file    - Log actions to file\n"
                     "l         - Turn off action logging\n"
                     "s [file]  - Display tree (overwrite file)\n"
                     "S [file]  - Display tree (append to file)\n"
                     "w         - Walk tree, running ShowFunc()\n"
                     "q         - Quit\n"
                     , stdout);
                fflush(stdout);
                break;

            case '@':
```

```
        LoadFile(tree, inbuf + 1);
        break;

    case 'a':
        if (inbuf[1] != ' ' || inbuf[2] == 0)
            fputs(" Not a valid command\n", stdout);
        else
            if (LoadString(tree, inbuf + 2)==TREE_FAIL)
                fputs(" ** Insertion failed\n", stdout);
        break;

    case 'c':
        if (inbuf[1] == ' ' && inbuf[2] != 0) {
            compare_length = atoi(inbuf+2);
            if (compare_length < 0)
                compare_length  = 0;
        }
        if (compare_length)
            printf("Comparing first %d chars.\n",
            compare_length);
        else
            printf("Comparing entire text.\n");
        break;

    case 'd':
        if (inbuf[1] == 'u' && inbuf[2] == 'p') {
            if (inbuf[3] == ' ' &&
                (inbuf[4] == '0' || inbuf[4] == '1'))
                tree -> DuplicatesOk =
                inbuf[4] == '0' ? 0 : 1;
            fputs("Duplicates are ", stdout);
            if (tree -> DuplicatesOk == 0)
                fputs("not ", stdout);
            fputs("allowed.\n", stdout);
            break;
        }

        if (inbuf[1] != ' ' || inbuf[2] == 0)
            fputs(" Not a valid command\n", stdout);


        else
            DeleteString(tree, inbuf + 2);
        break;

    case 'f':
        if (inbuf[1] != ' ' || inbuf[2] == 0)
            fputs(" Not a valid command\n", stdout);
        else
            FindString(tree, inbuf + 2);
        break;

    case 'l':
        if (inbuf[1] != ' ' || inbuf[2] == 0) {
            if (logfile) {
                fclose(logfile);
                logfile = NULL;
```

```
                    }
                    else
                        fputs(" Logfile not open\n", stdout);
                }
                else {
                    logfile = fopen(inbuf + 2, "w");
                    if (logfile == NULL)
                        printf("Can't open %s\n", inbuf + 2);
                }
                break;

            case 's': case 'S':
                if (inbuf[1] == ' ' && inbuf[2] != 0)
                    xWalkBintree(tree, inbuf + 2,
                    inbuf[0] == 's' ? "w" : "a");
                else
                    xWalkBintree(tree, NULL, NULL);
                break;

            case 'w':
                WalkBintree(tree, ShowFunc);
                break;

            case 'q':
                if (logfile)
                   fclose(logfile);
               return;

           case ';':
               break;  /* comment */

           default:
               fputs(" Not a valid command\n", stdout);
               break;
        }
    }
}
#endif
```

如下面的代码所示，我们定义了5种树基本操作（树创建、树查找、节点插入、节点删除和树遍历）的原型。

```
/* Prototypes */
Bintree *NewBintree (Bnode *dummy, CompFunc cf,
                     int dup_ok, size_t node_size);
Bnode   *FindBintree(Bintree *t, Bnode *n);
int      InsBintree (Bintree *t, Bnode *n);
Bnode   *DelBintree (Bintree *t, Bnode *n);
int      WalkBintree(Bintree *t, DoFunc df);
```

用于这些例程的代码在 bintree.c 中。还有一个测试驱动程序，它演示了使用这些基本操作，而不是创建和遍历树的简单过程。让我们检查三个函数。

6.1.1 树查找

查找一棵二叉查找树（程序清单6-1，FindBinTree）很直观。看看下面的伪代码：

```
search(node, find) {
      while (node != NULL) {
            if (node->data == find->data)
                  return node;
            else if (find->data < node->data)
                  node = node -> left;
            else
                  node = node -> right;
      }
      return NULL; /* search failed */
}
```

从任何给定的节点开始，我们首先询问节点的数据是否与查找项匹配。如果是，我们就退出。如果不是，我们就查看左子节点，看看查找项是否小于当前节点；或者查看右子节点，看看它是否大于当前节点。在实际代码中，我们利用了以下事实：把指向两个子节点的指针定义为两个元素的数组，而不是两个截然不同的名为 left 和 right 的元素。在 bintree. h 中，我们随意把 RIGHT 定义为0，并把 LEFT 定义为1。在代码中，把来自比较函数的返回值存储在变量 dir 中。然后，不用编写以下代码：

```
if (dir < 0)
   branch to link[1]
else
   branch to link[0]
```

我们利用了表达式 dir <0 将返回0或1的事实。因此，我们简单地编写以下代码：

```
examine link[dir < 0]
```

注意如何处理失败的查找。如果 node 曾经变为 NULL，代码就知道想要的数据项遗失了并且返回 NULL。

6.1.2 节点插入

节点插入也很简单（程序清单6-1，InsBinTree）。我们首先向下遍历树，就好像我们在查找想插入的节点一样。当我们到达一个没有子节点的节点时（也就是说，当当前节点为 NULL 时），就插入新节点作为最后一个节点的子节点。这种操作是定义树的结构具有指向节点的指针的一个原因，它指向根。

6.1.3 节点删除

删除是三种基本操作中最难以实现的操作（程序清单6-1，DelBinTree），这主要是由于我们必须确信在正确的相对位置维持节点，从而使得将维持在图6-4中定义的关键次序。同样，我们向下遍历树，查找我们想删除的节点。一旦找到它，就会有三种可能的情况。在这里的讨论中，我们把要删除的节点称为 s。

情况1：如果s缺少右子节点，就用s的左子节点替换s，如图6-5a所示。这是最简单的情况，它还可以处理s没有子节点的可能情况。

情况2：如果s的右子节点没有左子节点，那么就使s的左子节点转变为其右子节点的左子节点，并用右子节点替换s，如图6-5b所示。

情况3：如果s的右子节点具有它自己的两个子节点，就会发生最糟糕的问题，如图6-5a所示。我们继续查找s的右子树，找到它最左边的节点。其结果是我们找到了s之后的下一个最大的节点。根据定义这个节点（在图中称为“leftmost”）缺少左子节点，因此我们可以通过用其右子节点替换它，从而轻松地释放它。然后，把s的两个子节点提供给这个新的节点，从而释放s。

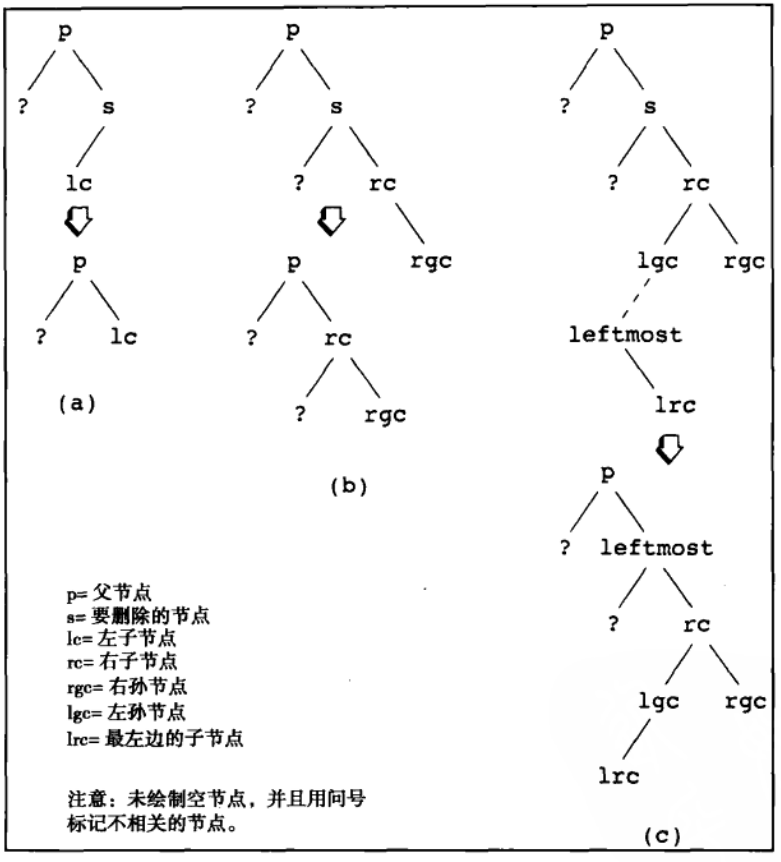


图6-5 从二叉树中删除节点

程序清单6-2 二叉树例程的定义

```
/*--- bintree.h -------------------------- Listing 6-2 ---------
 * Binary-tree definitions
 *
 *---------------------------------------------------------------*/
```

```
#define TREE_OK    (0)
#define TREE_FAIL (-1)
#define LEFT        1
#define RIGHT       0

#if defined(REDBLACK)
    #define RBONLY(x) x
#else
    #define RBONLY(x)
#endif


/*
 * Basic node structure. The actual size of a node is unknown as
 * the user will have appended data bytes on to the end of
 * this structure. The BINTREE_STUFF macro is a convenient way
 * to summarize the items the tree algorithm requires in the
 * node. Its argument is the tag of the structure being defined.
 */
#define BINTREE_STUFF(x)    struct x *link[2] \
                            RBONLY(;int red)
typedef struct sBnode {
    BINTREE_STUFF(sBnode);
} Bnode;

/* Control structure for a binary tree */
typedef int (*CompFunc) (void *node1, void *node2);
typedef int (*DoFunc) (void *node, int level);

typedef struct sBintree {
    Bnode *DummyHead;
    CompFunc Compare;
    int DuplicatesOk;
    size_t NodeSize;
} Bintree;

/* Prototypes */
Bintree *NewBintree (Bnode *dummy, CompFunc cf,
                     int dup_ok, size_t node_size);
Bnode   *FindBintree(Bintree *t, Bnode *n);
int      InsBintree (Bintree *t, Bnode *n);
Bnode   *DelBintree (Bintree *t, Bnode *n);
int      WalkBintree(Bintree *t, DoFunc df);
Bnode   *InitBintreeNode(size_t size);
```

6.1.4 二叉查找树的性能

二叉查找树的平均性能非常好。如果一棵树具有 N 个随机分布的节点，那么它的平均高度应该为 $\lg N$ 个节点。例如，如果我们在二叉树中插入单词“Marry in haste, repent at leisure”，就会得到如图 6-6 所示的树。得到的树的高度为三层，并且是合理平衡的。我们能够做得更好吗？当然，因为这棵树只需要两层那么高。不过，通过加载一组数据项而得到的二叉查找树的形状不仅依赖于数据项，而且依赖于加载它们的顺序。这种类型的树没有内在的机制用于阻止子树之间的失衡。

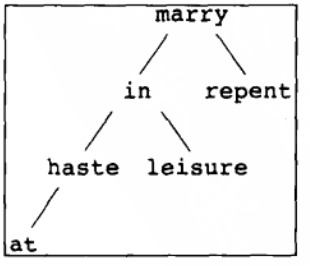


图 6-6 在加载短语“Marry in haste, repent at leisure”之后得到的树

当二叉树的输入数据已经有序时，将会发生最坏情况。例如，如果按顺序把同样6个单词加载进二叉树中时，得到的树将完全失衡，如图6-7所示。在这种情况下，树的高度将不是 lg N，而是 N。这种不佳的最坏情况下的性能使得折半查找是不可接受的，除非你可以：(a) 保证随机顺序的输入；(b) 在创建树之前研究输入。如果你提前知道要加载的数据项，就有可能通过使用直观的动态编程技术来构造一棵最优的二叉树。在 Knuth 的著作［Knuth 1973］中描述了这种技术，在这里将不会讨论它。我们将代之以将注意力转向更复杂的算法，它们不论什么时候都强制树是平衡的。

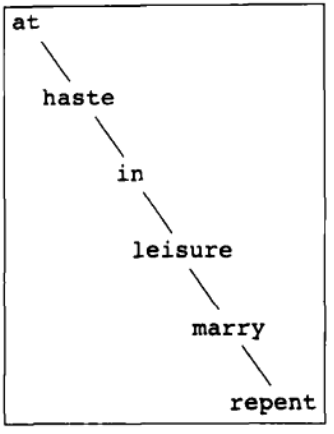


图6-7 在加载短语“at haste in leisure marry repent”之后得到的长而瘦的树

6.1.5 AVL树

有多种方法可用于创建平衡树。例如，如果你知道树中的数据项多久使用一次，就可以把更常用的数据项放在树中更高的位置，从而创建**加权的平衡树**（weight-balanced tree）。不过，一般更有用的是假定所有数据项的使用频率相同的树。在这种情况下，将强制树是高度平衡的（height-balanced）；也就是说，任何给定的子树都不允许比其兄弟子树高得太多。这种思想的具体表述最初是由 G. M. Adel'son-Vel'skii 和 E. M. Landis 提出的［Adel'son 1962］。遵循它们的实现的树通常称为**AVL树**（AVL tree）。AVL树是一种遵守以下规则的二叉查找树：任何给定节点的子树的高度最多相差1。

这个规则要求我们把树视作子树的森林，并且在整棵树中实施该规则，如图6-8所示。注意：该规则适用于小子树（a和b），以及根的两棵子树（c和d）。虽然该规则听起来可能有些麻烦，但是事实证明可以通过只执行树的局部操作来维护一棵平衡树。在两种情况下，对于用于二叉查找树的算法，节点插入和删除以完全相同的方式开始。在插入或删除之后，如果引入了失衡的情况，就必须重新平衡树。

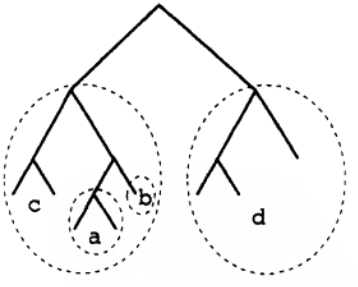


图6-8 AVL树

为了查看它如何工作，考虑图6-9中的树。在这里，插入或删除操作导致子树a变得太高。对于以b为根的子树则没有这个问题，这棵子树是平衡的。相反，失衡发生在整棵树内：右子树的高度为h，而左子树的高度为h+2（回忆可知：我们称树的根的高度为0，因此左子树的高度为h+2，而不是h+3）。为了修正这个问题，对树应用**单旋转**（single rotation）。实际上，我们使树以节点b和d为轴；节点b上移，而节点d下移。注意：这个单旋转有一个对称的变体，其中右子树将变得太高。在这种交换之后，左子树（以a为根）和右子树（以d为根）的高度为h+1。仔细研究这幅图，因为这个基本的操作将在本章余下部分中反复出现。

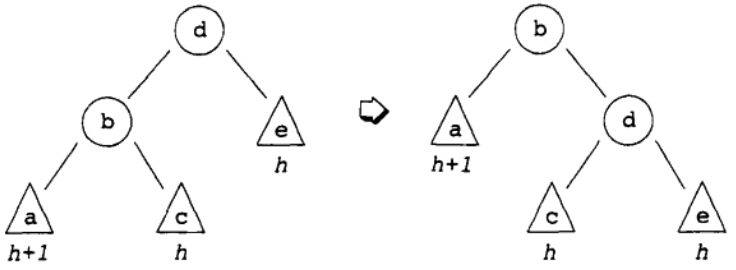


图 6-9　稍微有点抽象的 AVL 树，其中将实际的节点绘制为圆圈，而三角形代表给定高度的子树

当太高的树是“外部”子树时，单旋转就是有用的。不过，当太高的子树是“内部”子树时，单旋转将不会修正这个问题。考虑图 6-10 中展示的情形。开始时，树是失衡的：左子树的高度为 $h+2$（沿着从 f 到 b 到 d 再到 c 的路径），而右子树的高度为 $h+1$。为了校正这种情形，我们将执行所谓的**双旋转**（double rotation）。实际上，节点 d 绘制在节点 b 和 f 之前，并在这个过程中将其子节点移交给这些节点。子树 c 和 e 的相对高度可以颠倒，但是其中一棵树的高度必须为 h，另一棵树的高度必须为 $h-1$。它有一个对称的变体，其中太高的子树是节点 f 的右子节点的子树。

在定义了这两种旋转之后，我们现在能够修复我们遇到的任何类型的树失衡情况。由于这两种操作对于后续的讨论很重要，你自己应该花一些时间绘制这些旋转的草图，并且确保你完全理解了它们。还要记住：每种配置都有一个镜像变体，其中失衡的子树位于树的另一边。

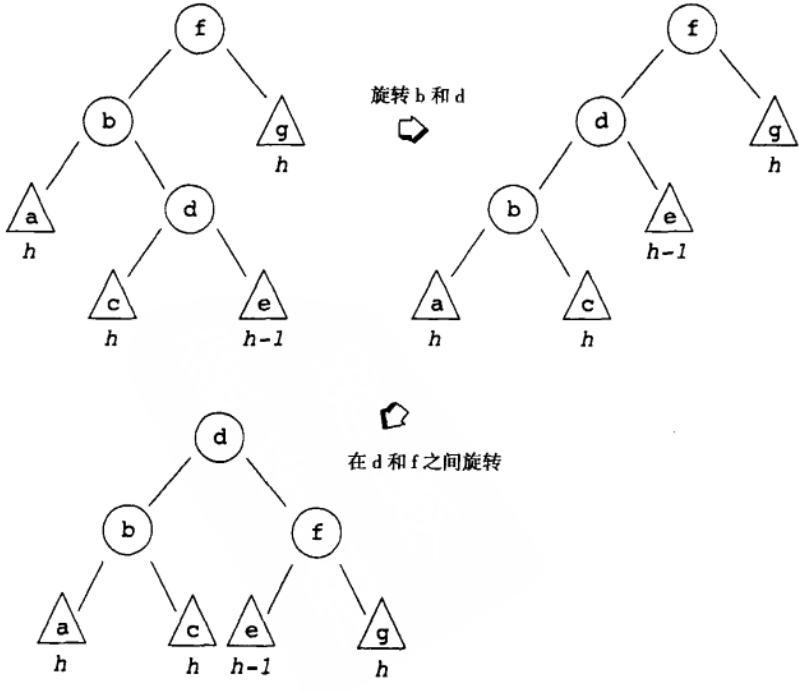


图 6-10　双旋转

AVL 规则的作用是：确保树永远不会在实质上变为失衡状态，并且可以证明具有 N 个节点的 AVL 树的高度将与 $\lg N$ 成正比。此外，由于我们可以在对树进行一次从上到下的遍历期间执行插入和删除操作，因此可以在与 $\lg N$ 成正比的时间内执行这些操作。不过，实际上实现 AVL 树比较麻烦，这是由于必须考虑许多特殊情况，并且存在以下可能性：树重构可能需要进行多次旋转，以修复由于以前的旋转而产生的新的失衡。这通常意味着必须对插入或删除期间采用的访问路径执行向后遍历。在下一节中将介绍一种好得多的解决方案，它允许旋转发生在向下查找遍历期间。

6.2 红黑树

实现 AVL 树的一种更容易的方式是：使用称为**红黑树**（red-black tree）的概念。当定义了宏 REDBLACK 时，bintree. c 和 bintree. h（程序清单 6-1 和程序清单 6-2）中的代码就实现了红黑树。事实上，红黑树是二叉查找树。不过，它利用了两个新概念。

第一，在红黑树中，数据只存储在树的**叶节点**（leaf）中。也就是说，只有不带子节点的节点才能包含实际的数据。内部节点只用于引用。

第二，将每个节点都视作带有红色或黑色。节点的颜色由下面这些规则确定：

- 所有叶节点都是黑色的。
- 在沿着从根出发的任何路径上都不允许出现两个连续的红色节点。
- 树的所有叶节点都必须具有相同的**黑色深度**（black depth），它被定义为叶节点与根之间的黑色节点的数量减 1。

图 6-11 中显示了一棵示例红黑树。这棵树的黑色高度为 2，因为在从根到任何叶节点的路径上都有 3 个黑色节点。为了突出黑色高度，在绘制树时，使所有黑色节点都位于相同层上。这棵树在它的叶节点中包含数字 1 ~ 9。一定要注意，只有“真实的”数据位于叶节点中。内部节点中的值只用于引用。当我们查找红黑树时，直至我们找到也是叶节点的匹配节点时，查找才会完成。

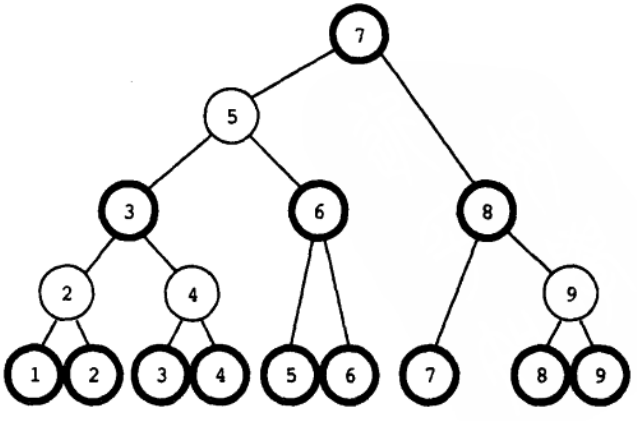


图 6-11 红黑树，其中黑色节点是用粗边框绘制的

红色和黑色的思想只是一种标记工具，它使得很容易维护平衡树，理解这一点很重要。稍后将讨论这样做的方式。节点的红色或黑色没有暗示任何其他的事情，并且树在其他方面的表现完全像是二叉树。为了允许例程保持节点颜色，必须向每个节点添加新的数据项。原则上，这个新数据项可以像单个位一样简单。实际上，通用例程将把它存储为 int。在 bintree.h 中演示了这一点，其中把变量 red 添加到宏 BINTREE_STUFF 中。因此，我们的基本节点现在具有以下形式：

```
typedef struct sBnode {
    struct sBnode *link[2];
    int red;
} Bnode;
```

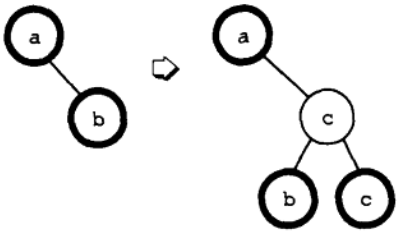


图 6-12　向红黑树中插入“c”

图 6-12 显示了向红黑树中插入节点。如果找到一个具有黑色父节点的叶节点，这个过程就很直观。关键是这个操作不会改变树的黑色高度；就黑色高度而言，引入红色节点是一种中性操作。这种插入技术是仅仅执行树的局部操作即可使红黑树维持其平衡的原因。最终结果是：在最坏情况下，我们的树将由交替的红色和黑色节点组成。这样，这种具有 N 个节点且黑色高度为 k 的树将具有实际高度 2k，并且这个高度仍然与 lg N 成正比。

为了确保我们到达具有黑色父节点的叶节点，我们将在树的向下遍历期间执行一种称为**颜色翻转**（color flip）的操作。图 6-13 中显示了它，其中节点 2 可以是其父节点的左子节点或右子节点。每次在向下遍历期间遇到具有两个红色子节点的黑色节点时，就会执行颜色翻转。同样，注意：这种操作不会改变树的黑色高度，因此无需参考树的其他部分即可执行它。

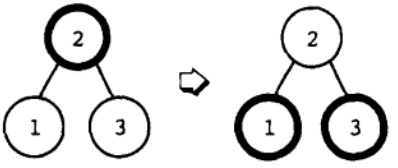


图 6-13　基本的颜色翻转

不过，颜色翻转有可能在一排产生两个红色节点。如图 6-14 所示，这可能以两种可能的方式之一发生。在每一种情况下，就黑色高度而言，树仍然是平衡的；问题是要分解成对的红色节点。可以通过执行单旋转或双旋转来完成该任务。如果红色节点处于相同的方向，单旋转就足够了。如果红色节点呈锯齿形，那么将需要两次旋转。每个位置都有一个对称的变体。

在代码中，所有的重新平衡工作都是由 split( ) 处理的。这个例程和一个辅助例程 rotate( ) 用于处理使红色节点数加倍的各种可能的情况。

对于记录来说，在插入期间并不总是可能到达具有黑色父节点的叶节点。如果最终的叶节点具有一个带有黑色兄弟节点的红色父节点，将不能通过颜色翻转修正问题。可以代之以像图 6-12 所示的那样插入节点，从而在一排创建两个红色节点。然后必须重新平衡树，如图 6-14 所示。

图 6-15 显示了这个问题的一个示例。在这种情况下，节点 4 是红色的，并且我们想插入节点 3。为此，我们将执行图 6-12 中所示的标准插入过程。这将创建一棵树，就黑色高度而言它仍然是平衡的，但它在一排具有两个红色节点。然后通过（在这种情况下）双旋转校正这种失衡状况。

在节点插入期间发生的所有以前的操作的一个重要特征是：自始至终在每一步都会保持树的黑色高度和平衡。因此，程序可以在任何时间停止，即使它已经对树进行了一次或多次重新排序。

从红黑树中删除节点就像从二叉树中删除节点一样，是所有操作中最复杂的操作。在理论上，

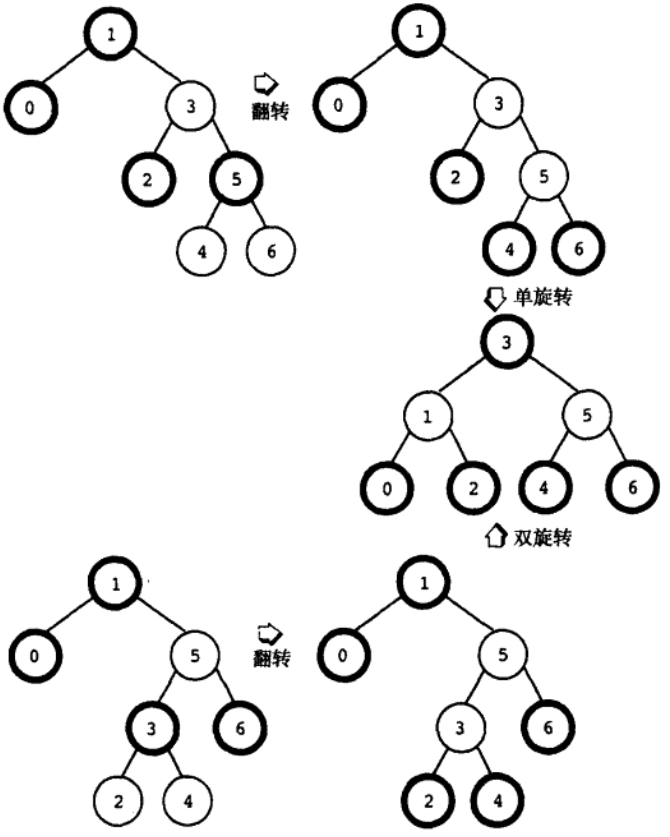


图 6-14 通过颜色翻转构成相邻红色节点的两种模式

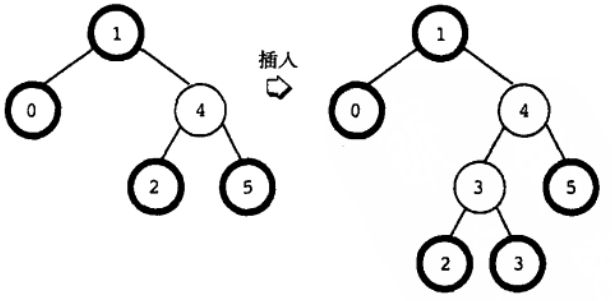


图 6-15 当最终的叶节点具有一个带有黑色兄弟节点的红色父节点时的节点插入

删除应该如同图 6-12 中所示的颠倒动作一样简单。换句话说，我们希望发现要删除的节点具有红色父节点。遵循插入节点的思想，我们只是简单地在树中把红色节点向下拖动。这开始于根；如果它的任何子节点都不是红色的，我们就把根涂成红色。然后，在通往要删除节点的路径上的每个节点中，通过执行以下过程之一强制节点变为红色：

- 如果当前节点是红色的，或者如果沿着遍历路径的下一个节点是红色的，我们就继续前进。
- 如果当前节点是黑色的并且下一个节点也是黑色的，当前节点就必须具有一个红色父节点，我们将不会继续前进。如果下一个节点具有红色兄弟节点，就可以通过执行图 6-16 中所示的操作来避免这种情况。
- 当前节点是黑色的，下一个节点是黑色的，当前节点的兄弟节点也是黑色的，但是当前节点的父节点是红色的。如果当前节点的兄弟节点具有两个黑色子节点，那么解决方案就很简单：只把兄弟节点涂成红色。实际上，这是一种相反的颜色翻转，如图 6-17 所示。

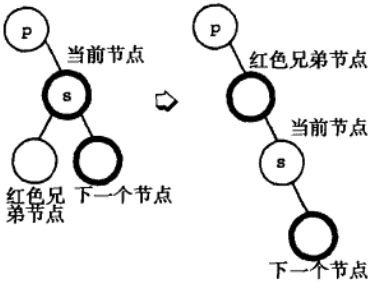


图 6-16　如果当前节点和下一个节点是黑色的但是下一个节点的兄弟节点是红色时所执行的操作

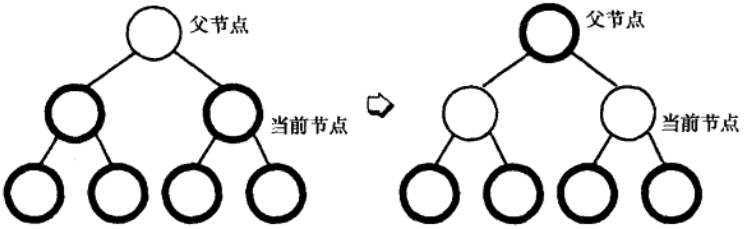


图 6-17　相反的颜色翻转把当前节点涂成红色

- 这是最复杂的情况。其条件与前一种情况相同，只不过当前节点的兄弟节点具有一个或多个红色子节点。如果简单地把当前节点涂成红色并把它的父节点涂成黑色，执行单旋转或双旋转即可平衡树。就单旋转而言，将需要少量的重新涂色，如图 6-18 所示。

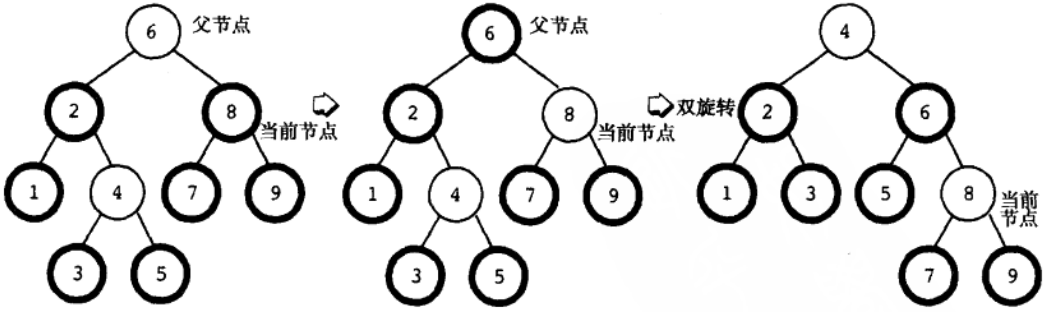


图 6-18　把当前节点涂成红色，并把父节点涂成黑色，然后利用单旋转或双旋转平衡树

像上一节中描述的 AVL 树一样，可以看到红黑树的黑色高度与 lg N 成正比。如前所述，红黑规则的作用是：保证黑色高度为 h 的红黑树的总高度至多为 2h，因此树的总高度仍然与 lg N 成正比。与 AVL 树一样，可以在单次向下遍历期间执行所有的操作，因此查找、删除和插入操作的执行时间都与 lg N 成正比。实际上，这个算法执行得非常好。

6.3 伸展树

伸展树是第三种二叉树变体。它们使用的结构与本章开头讨论的简单二叉查找树相同。在节点中没有存储平衡信息或节点颜色信息。作为替代，伸展树是链表维护的“上移”方法（在第2章和第3章讨论）与二叉树之间的交叉。无需像我们对红黑树所做的那样尝试保持树处于平衡状态，我们将实现一种恒定重排的形式：每次访问树时，都使用双旋转和单旋转重排树，使得访问过的节点（或者用于它的空间）位于树的根部。伸展树会在使用时“学习”，并且最近使用的数据项将比未使用的数据项更快地被访问。

所有的伸展树操作都与称为“伸展”的特殊操作相关。Splay（tree，node）将重排tree，使得node是树的根。如果node在树中不存在，那么根将是位于node之前或之后的节点（如果它在树中存在的话）。

为了查看这个操作如何使得查找、插入和删除更容易，在讨论如何实现Splay()之前，让我们考虑这些动作。查找很普通。在Splay(tree，node)之后，我们只检查树的根。如果它与node完全相同，那么我们就在树中找到了node。否则，node不在树中，如图6-19所示。

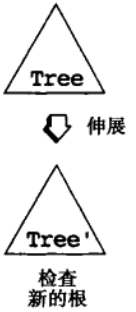


图6-19 在伸展之后完成查找

插入几乎一样简单：在伸展之后，把树切成两半，使得新节点成为根，并使得两棵树把新的根的右子节点和左子节点分成两半。图6-20演示了这个过程，其中在伸展之后重绘了树，使得可以看见树的前两层。把新节点放在节点a的左边或右边，这依赖于两个节点的相对次序。在通过伸展定位想要的节点之后，从伸展树中删除节点只是按相反顺序执行这些步骤的简单事情。

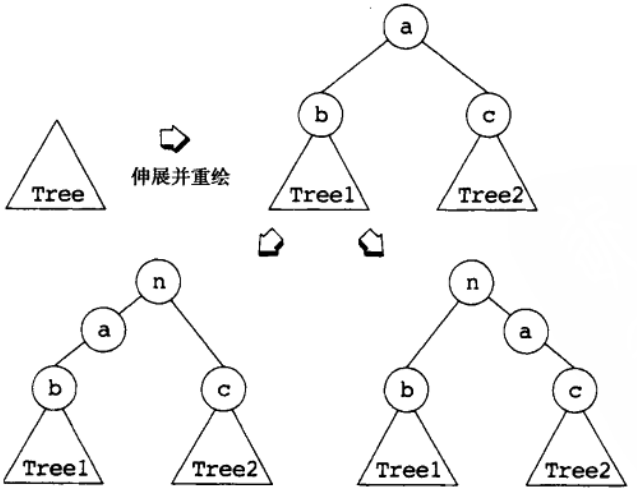


图6-20 在伸展之后插入

实现伸展极其直观。首先将描述自底向上的伸展，因为它更容易理解。不过，以自顶向下的

方式可以更高效地实现伸展，并且将在代码中采用这种方法。

为了实现自底向上的伸展，我们首先描绘从根到想要节点的路径。然后，通过应用图6-21中所示的三种情况之一，逐渐把这个节点旋转到根。在每一种情况下，标记为1的节点都是目标节点，并且三角形代表未指定的子树，它们可能是任意大小，也可能为空。显示的树通常是子树，附加到树的下一层被显示为一条垂直虚线，它可能是左链接或右链接。操作a是终端情况，其中目标节点变成树的根。操作b是“之字形”情况，并被实现为两次连续的单旋转：第一次在节点3和2之间，第二次在节点2和1之间。操作c也是“锯齿形”情况，并被实现为双旋转。每种情况都有对称的变体。在该图中，我们应用操作b或c，直到目标节点变为根，在这种情况下我们就完成了任务，或者目标节点只是从根中删除的一层，在这种情况下将应用操作a并且完成了任务。

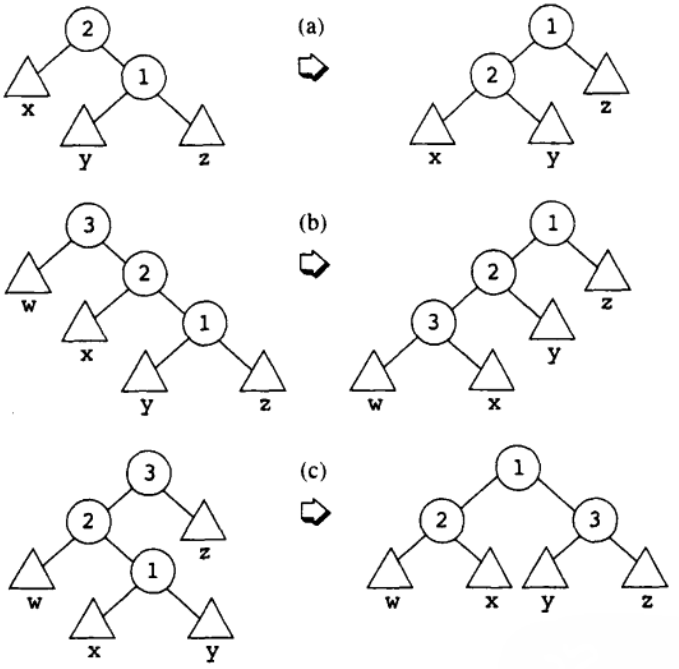


图6-21 自底向上伸展的三个操作

实现自顶向下伸展的代码如程序清单6-1（bintree.c）中所示，并且是由常量SPLAY控制的。该代码是图6-22中所示效果的直观实现。

在图中所示的各种情况下，顶部的节点是当前树的根，并且查找路径正把我们带往节点1。前三个操作与图6-21中的那些操作完全相同；操作a是终端情况，操作b和c是“之字形”情况和“锯齿形”情况。为了执行伸展，我们如所示的那样把树分解开，并把各个片断悬挂到左存储树和右存储树上，标记为L和R。中心片断变为新树，并根据相同的规则重新处理。在左存储树内，总是把片断放在右边尽可能远的地方；而在右存储树内，则将其放在左边尽可能远的地方。当树缩减成想要的节点时，就把各个片断重新装配起来，如操作d中所示。

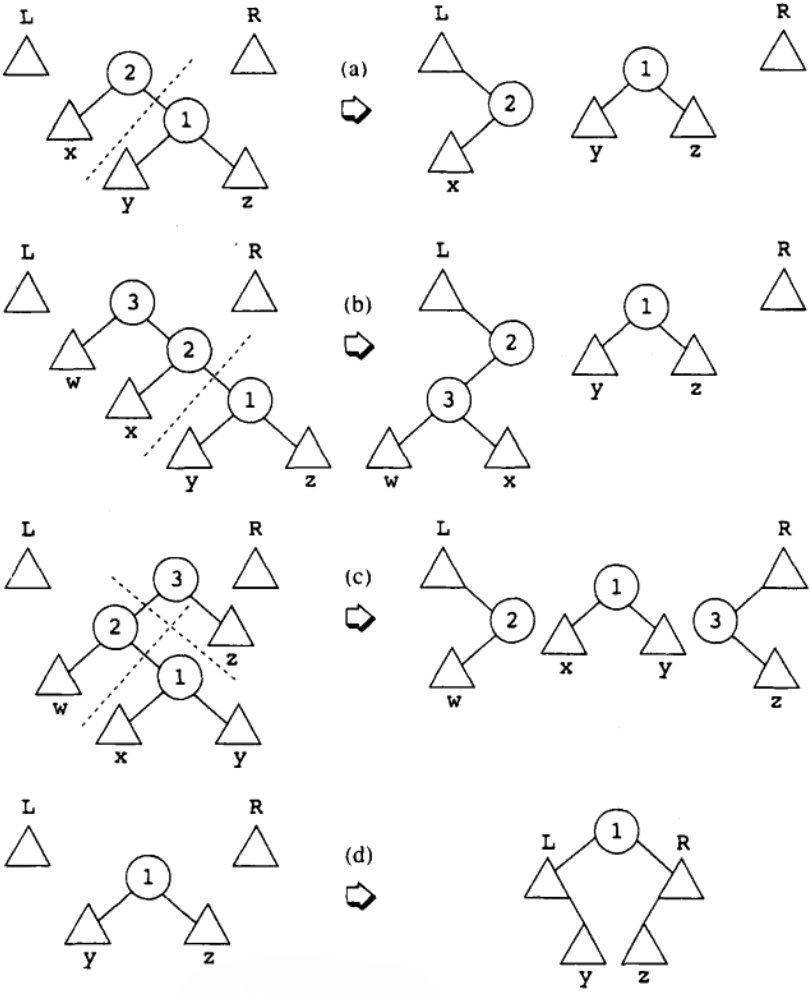


图 6-22 执行自顶向下伸展的过程

与其他树一样，通过把左、右子指针定义为两个元素的数组，将镜像情况作为常规情况的一部分进行处理。代码中唯一复杂的方面是维护左、右存储树的方式。这些树的根存储在数组 LR 中，而数组 LRwalk 则保存一个指针，它指向右子树左下角的节点和左子树右下角的节点。在每次向存储树中添加节点之后，都会调用宏 PUSH 把 LRwalk 推进到存储树的底部。

用于伸展树的查找、插入和删除操作的代码极其简单，注意到这一点很有趣。当把该代码与二叉树和红黑树变体使用的代码作比较时，这种简单性尤其明显。

图 6-23 中显示了通过自顶向下伸展执行的树重排的类型。排列 a 是在按顺序插入了数字 1～10之后的伸展树；b 是在查找节点 1 之后的树；c 是在查找节点 5 之后的树；d 是在查找节点 4 之后的树。

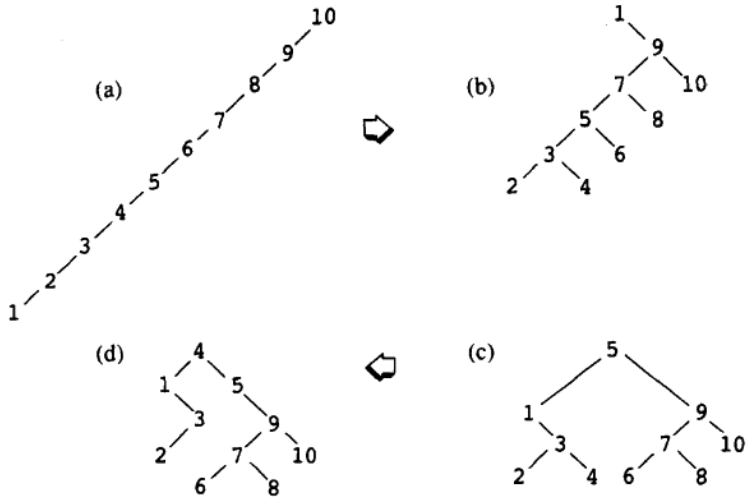


图 6-23 由自顶向下伸展执行的树排列的类型

注意到关于树排列的这些类型的几个事实很重要。首先，如果元素是按顺序插入的，那么伸展树将像二叉树一样变得长而瘦。不过，对较低元素的任何访问都将迅速具有打乱树的效果。混乱的树更短一些，可以改进对所有元素的访问。其次，最近使用的元素将保持在树顶部附件。这在某些情况下具有明显的优点。最后，自顶向下和自底向上伸展操作不会产生完全相同的树（尽管差别极小）。不过，每种操作产生的效果总体上是相同的，而自顶向下的实现更高效，并且更容易实现。

分析伸展树的性能很复杂，因为每个操作都会影响总体结构以及树中许多其他节点的深度。例如，很容易看到：访问给定的节点将粗略地把沿着查找路径的每个节点的深度减半。尽管分析很复杂，Sleator 和 Tarjan[Sleator 1985] 证明对具有 n 个节点的伸展树进行 m 次访问所需的时间与 $(m+n)\lg(m+n)$ 成正比。由于伸展树从使用中“学习”的方式，必须通过多次访问来分析伸展树的性能。不过，对于任何次数足够多的树访问，伸展树的性能与任何其他类型的平衡树的性能一样好。

6.4 B 树

本章中迄今为止描述的树算法对于完全可以在内存中维护的数据工作得很好。如果不是这样，就需要一种适用于磁盘存储器的树算法。利用磁盘存储器具有它自己的一组特殊的限制：

- 与访问内存中的数据所需的时间相比，访问磁盘上的数据非常慢。
- 最好对相当大的数据块执行读/写操作。实际上，一次读入的数据越多，就会做得越好（在限制范围内）。使用 2048 ~ 4096 字节的数据块一点也不罕见（参见第 1 章，了解关于磁盘 I/O 优化的讨论）。

磁盘存储器的这些限制意味着必须尝试最高效地使用大块磁盘存储器。应该围绕着这些块组织我们的算法，并且应该计划将多份数据放入每个块中。解决这个问题的算法统称为 **B 树**

(B-tree)算法。几乎每一位作者都以稍微有点不同的方式实现 B 树，但是底层的概念是相同的。

所有的 B 树都利用了两种截然不同的块：索引块和数据块。数据块是 B 树的叶节点，并且所有的数据都存储在其中。索引块是上层块，它们只包含允许程序描绘从根到数据块中的想要记录的路径所需的足够信息。例如，考虑图 6-24，它描绘了一棵具有两层索引块的 B 树。B 树的关键特性如下：

- 所有数据块都位于相同层——在这里是向下第三层。
- 所有的索引块和数据块都包含某个最低限度的数据量。可能依据键的数量或者使用的块空间的数量来定义这个数据量。
- 与我们前面介绍的树一样，索引块中的数据项只是键。为了查找键为 Harrison 的数据记录，我们从顶部（根）块开始，到达第二层的中间块，然后到达这个索引块的最右边的子块。在此将会找到实际的记录。

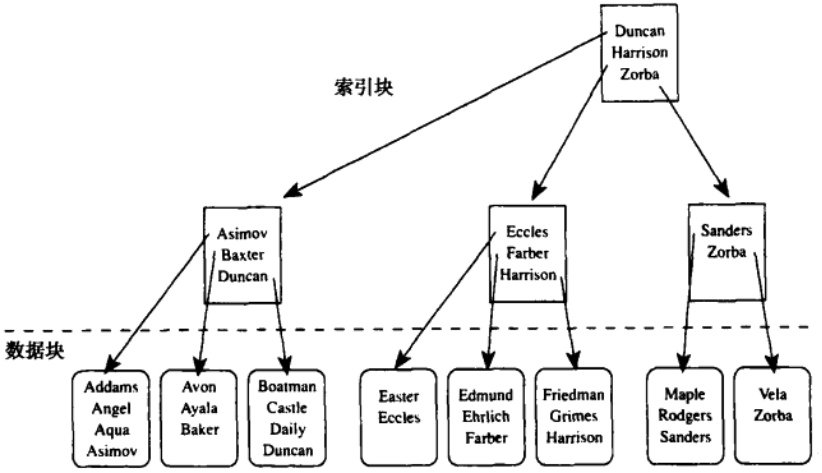


图 6-24 具有两个索引层的 B 树

这些特性综合在一起意味着至少在某种意义上 B 树是平衡的。由于所有的数据都处于树中的相同深度，对于树的各个部分将具有同样快的访问速度。在更严格的意义上，通常把 B 树描述为阶（order）d，其中树中的每个节点都包含 d ~ 2d 个键或记录。这样，具有 N 个条目的树的高度将是 $\log_d^N$。注意：虽然阶会影响树的高度，但是阶和高度是两个差别很大的概念。不过，树的这种基于其阶的定义确实只对 B 树有用，其中所有的键都具有固定的长度。如果使用可变长度的键，那么要求所有块都包含某种数据量将更合适。换句话说，我们将不会要求每个块中包含 d ~ 2d 个键，而是要求（例如）每个块中有 15% ~ 85% 的字节被使用。

6.4.1 保持 B 树平衡

在纸上可以很容易地描述 B 树。当块变得太满或太空时，就会发生问题。当它太满时，必须把块分割成两半，并在块的父块中插入一个额外的键。这很容易做到，当然除非父块也是满的。在这种情况下，可能需要分割父块。最终，可能需要分割每个索引块，直到根块。如果根块也需

要分割，树的高度必定会增加。与之相反，当发生删除操作并且块变得太空时，就会检查同一层上的相邻块。如果可能，应该把太空的块与其左边或右边的兄弟块合并起来。可以在树中串联这个过程，导致在许多上层索引块之间进行合并。如果没有合并上层索引块，那么它们也可能会受到影响。

再次考虑图 6-24。假设我们想检测键 Zorba。这样，Vela/Zorba 块相对较空，必须与 Maple/Rodgers/Sanders 块合并。为此需要改变 Sanders/Zorba 索引块，使得它只具有键 Vela，并且必须把根块调整为具有指向 Duncan/Harrison/Vela 的指针。最终结果如图 6-25 所示。这假定 Eccles/Parker/Harrison 索引块并不是足够大，从而也能够吸收 Vela。如果它能够这样，那么树将进一步折叠，并且具有一个根块、两个第二层索引块和 7 个数据块。

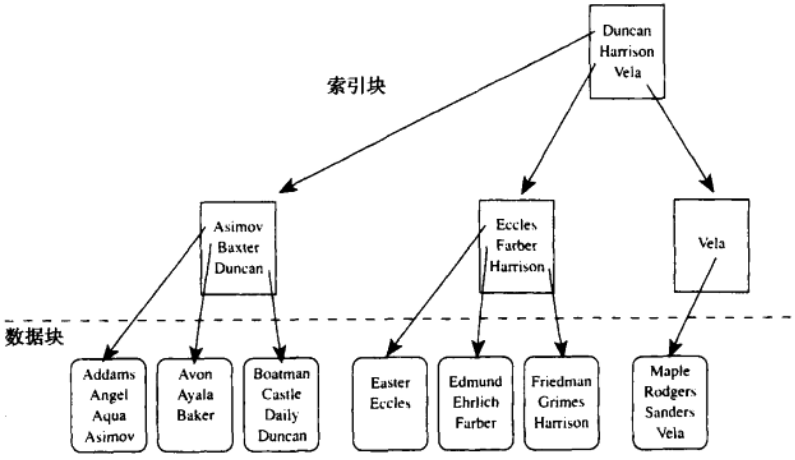


图 6-25　对于图 6-24 所示的 B 树，在删除了 Zorba、合并了两个数据块并且向上传播新键之后得到的结果

6.4.2　实现 B 树算法

实现 B 树是主要的任务。分割和合并数据块与索引块需要做的工作很复杂，并且充斥着一些特殊情况。不过，通过进行几个简化假设，可以相当大地减小复杂性，同时实现一个完全令人满意的 B 树例程。对于这种 B 树实现，设计目标如下：

- 允许可变长度的键。
- 允许可变长度的记录。
- 把索引和数据保存在单独的文件中。如果你喜欢，这些文件可能使用单独的基本块大小，并且具有单独的分割和合并断点。如果索引损坏，那么使用单独的索引文件允许对数据文件重建索引。

为了实现这些特性，同时轻松应对特殊情况，应该做出以下几点假设：

- 在创建数据集时，必须提供具有最大键的单个记录。不能删除这个记录。这个记录的存在意味着树永远不会为空。

- 树将具有固定的高度，并且将在创建 B 树时定义这个高度。

这两条假设的实际效果是：空树将包含多个索引块层以及一个最大的数据记录。例如，图6-26中显示了具有三个索引层的空 B 树。不能删除键为∞的记录。

这些假设实际上并不是非常具有限制性。对固定高度的树的要求是相对的，而不是绝对的；也就是说，可以轻松地修改代码，允许树在高度上增高或缩短。这里选择了固定高度的模式，因为正如我们将在以后看到的，高度为 4 的树甚至对于最苛刻的应用程序也绰绰有余。

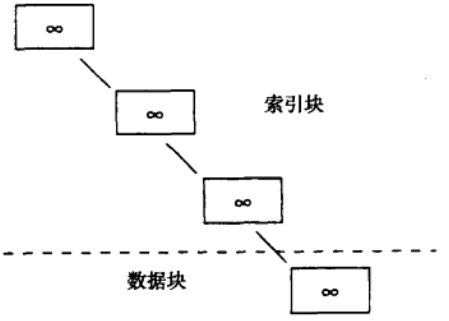


图 6-26 具有三个索引层的空 B 树

6.4.3 B 树实现的代码

编码 B 树模块是应用理论和语用学中的一个练习。除了刚才介绍的概念之外，这里将讨论很少的理论；代码及其局限性的讨论提供了对设计 B 树模块的充分见解。用于我们的实现的代码开始于程序清单6-3（bt-hdr. h）。由于这是一个如此大且复杂的程序，在表6-1 中提供了对源代码模块的少量指导。

表 6-1 源代码模块

文 件	页 数	用 途	主要入口点
bt-hdr. h（程序清单6-3）	225	B 树头文件	
bt-new. c（程序清单6-4）	228	创建 B 树	bt_ new ()
bt-open. c（程序清单6-5）	233	打开现有的数据集	bt_open() bt_close()
bt-data. c（程序清单6-6）	235	操作数据集	bt_walk() bt_find() bt_add() bt_delete()
bt-disk. c（程序清单6-7）	255	阻塞缓冲	bt_getblock4read() bt_getblock4write() bt_flush()
user. h（程序清单6-8）	262	驱动程序的头部	
user. c（程序清单6-9）	262	示例驱动程序	main()

最好从 user. c 中的 main() 开始查看代码。这是一个简单的驱动程序，与 bintree. c 中的驱动程序非常相似。它允许创建、打开、关闭、添加、删除和显示数据集。它主要由 switch 语句组成；要查看如何使用特定类型的特性，只需检查相关的 case 语句即可。

1. 布置数据集：bt_new. c

我们的数据集将包含两个文件，分别命名为 file. ndx 和 file. dat（参见程序清单6-4）。. ndx 文

件包含索引；.dat 文件包含实际的数据。两个文件具有相似的布局：

块 0：文件标识和布局数据

块 1 ~ n：索引或数据记录

程序清单 6-3　B 树例程的定义和宏

```
/*--- bt_hdr.h ---------------------------- Listing 6-3 ---------
 * B-tree header file, to be included by user code
 *
 *-------------------------------------------------------------*/

#if !defined(BT_HDR_H)
#define BT_HDR_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 * Some defines
 */
#define MAX_PATH_AND_FILE 100  /* max length of full file name */
#define TREE_OK   (0)
#define TREE_FAIL (-1)

/*
 * Some general typedefs
 */
typedef unsigned BLKSIZE; /* Holds size of disk blocks */
typedef long DISKOFFSET;  /* File offset for fseek */

/*
 * Structure of index files:
 *
 * First block (offset 0L, size is determined by size_struct):
 *   Signature string: SIGNATURE SIG_NDX
 *
 *   Sizes:            Sizes size_struct;
 *
 *      Root block:     DISKOFFSET root_block;
 *      Free block chain: DISKOFFSET free_block;
 *
 *
 * Data blocks
 *   Block control:  BlockControl blockdata;
 *                     If blockstate == sNode, then the
 *                     DISKOFFSETS in this block are pointers
 *                     to additional index blocks.
 *                     If blockstate == sLeaf, then DISKOFFSET
 *                     is a pointer to a data block in the
 *                     .dat file
 *
 *                   key/DISKOFFSET pairs
 *
 *                     The user-supplied function get_key_size()
 *                     is used to step through the block and the
```

```
 *                    user-supplied functions key2keycmp() and
 *                    key2reccmp() are used to compare keys.
 */

/*
 * Structure of data files:
 *
 * First block (offset 0L, size is determined by size_struct):
 *   Signature string: SIGNATURE SIG_DAT
 *
 *   Sizes:            Sizes size_struct;
 *
 *      Root block:     DISKOFFSET root_block;
 *      Free block chain: DISKOFFSET free_block;
 *
 *
 * Data blocks
 *   Block control: BlockControl blockdata;
 *   Data items:    The data records are variable length and are
 *                  just packed in one after another. The user-
 *                  supplied function get_rec_size() is employed
 *                  to step through the records.
 */

/* Signature strings */
#define SIGNATURE "B-Tree File/V1.0/"

/* SIG_DATA and SIG_INDEX must be same length */
#define SIG_DATA  "Dat"
#define SIG_INDEX "Ndx"

typedef struct sSize {
       /* Block sizes and share/split breakpoints */
    BLKSIZE block_size; /* size of blocks */
    BLKSIZE split;      /* split blocks when this # bytes used */
    BLKSIZE merge;      /* merge blocks when this # bytes used */
    unsigned levels;    /* # of index levels (.ndx files only) */
} Sizes;

/* Control structures for disk blocks */
typedef enum eBlockState { sAvail, sNode, sLeaf } BlockState;
typedef struct sBlockControl {
    size_t     bfree;   /* Free data area begins here */
    BlockState blockstate;
    DISKOFFSET parent;  /* parent of this block */
    DISKOFFSET next;    /* used when block belongs */
                        /* ... to a free chain */
} BlockControl;

/* Control structures for I/O buffer blocks */
typedef enum eBufState { sFree, sClean, sDirty } BufState;
typedef struct sBufferList {
    BufState state;
    DISKOFFSET offset;
    char *buffer;
    struct sBufferList *next;
} BufferList;

/*
 * Master control structure for access to a pair of database
```

```
 * files. See code in bt_open.c for example initialization.
 */
#define NDX 0
#define DAT 1
typedef struct sBtree {
    /* Files, file names, and buffers */
    struct {
        char filename[MAX_PATH_AND_FILE];
        int modified;
        FILE *file;
        BufferList *bufferlist;
        Sizes sizes;
        DISKOFFSET root_block;
        DISKOFFSET free_block;
    } fdata[2];

    /*--- Data objects and procedures to manipulate them ---*/

    /* get size of a key */
    unsigned (*getkeysize)(void *);

    /* size of key in a record */
    unsigned (*getkeyNrecsize)(void *);

    /* get size of a record */
    unsigned (*getrecsize)(void *);

    /* compare key to key */
    int (*key2keycmp)(void *, void *);

    /* compare key to record */
    int (*key2reccmp)(void *, void *);
    /* copy a key from a record */
    void (*rec2keycpy)(void *, void *);

    /* Miscellaneous */
    int error_code;     /* Index into ErrorCode[] */
    int duplicatesOK;   /* Are duplicate keys ok? */

    /* used by insert & delete */
    DISKOFFSET CurrentDataBlock;
    void *SearchKey;
    void *FoundRec;
} Btree;
/* some defines to simplify the code */
#define GETKEYSIZE     (*(bt->getkeysize))
#define GETKEYNRECSIZE (*(bt->getkeyNrecsize))
#define GETRECSIZE     (*(bt->getrecsize))
#define KEY2KEYCMP     (*(bt->key2keycmp))
#define KEY2RECCMP     (*(bt->key2reccmp))
#define REC2KEYCPY     (*(bt->rec2keycpy))
#define FDATA(x,y)     (bt->fdata[x].y)
#define SIZES(x,y)     (bt->fdata[x].sizes.y)

/* error messages */
extern char *ErrorText[]; /* defined in bt_disk.c */
```

```
/* Prototypes */

/* action function used during tree walk */
typedef int (*DoFunc) (Btree *bt, void *rec);


/* bt_new.c */
int  bt_new(Btree *bt, void *maxkey, void *maxrec);

/* bt_disk.c */
char *bt_getblock4read(Btree *bt, int f, DISKOFFSET dof);
char *bt_getblock4write(Btree *bt, int f, DISKOFFSET dof);
BufferList *bt_getnew4write(Btree *bt, int f);
int  bt_flush(Btree *bt);
int  bt_bufinit(Btree *bt, int f);
void bt_bufrelease(Btree *bt, int f);
int  bt_releaseblock(Btree *bt, int f, DISKOFFSET dof);

/* bt_open.c */
int  bt_open(Btree *bt);
int  bt_close(Btree *bt);

/* bt_data.c */
int  bt_add(Btree *bt, void *rec, void *key);
int  bt_delete(Btree *bt, void *key);
int  bt_find(Btree *bt, void *rec, void *key);
int  bt_walk(Btree *bt, DoFunc df);
#endif

The data in block 0 permits the files to be self-identifying, and the
layout of this data is shown in bt_hdr.h. The block contains the
following:
"A signature string"        // identifies the file as legitmate
struct sSize {
    unsigned block_size;    // size of blocks in bytes
    unsigned split;         // split blocks using more than this
    unsigned merge;         // merge blocks using less than this
    unsigned levels;        // height of tree (.ndx only)
} sizes;
long root_block;                    // first index block (.ndx only)
long free_block;                    // first block in free-block chain
The code in bt_new.c is reached by calling bt_new() and creates two empty
files in a straightforward fashion. For .ndx files, the appropriate number
of index levels are created, while .dat files are created with one data
block that contains the aforementioned record with a maximal key.
```

程序清单6-4 创建新的B树数据集的代码

```
/*--- bt_new.c ---------------------------- Listing 6-4 ---------
 * Create a new, empty B-Tree dataset
 *
 * See user.c (Listing 6-9) for a test driver.
 *-------------------------------------------------------------*/

#include "bt_hdr.h"
```

```
/*
 * Create new dataset files. Files are not left open
 *
 * Dataset has one record in it, a record with the maximum key
 * that will ever be used. We populate each index level with
 * this one key and create one data block containing this record.
 */
int bt_new(Btree *bt, void *maxkey, void *maxrec)
{
    char *buffer, *s;
    int i, ret;
    unsigned length, j;
    DISKOFFSET *dof;
    Sizes *sizes;
BLKSIZE blksize;
BlockControl *bc;

/* Check values, allocate buffers */
ret = TREE_OK;
for (i=0; i < 2; i++) {
    if (SIZES(i,block_size) > 10000) { /* seems unlikely */
        bt -> error_code = 2;
        ret = TREE_FAIL;
        break;
    }

    if (SIZES(i,split) > SIZES(i,block_size) ||
        SIZES(i,merge) > SIZES(i,block_size)) {
        bt -> error_code = 3;
        ret = TREE_FAIL;
        break;
    }

    ret = bt_bufinit(bt, i);
    if (ret)
        break;

    /* Open the file */
    FDATA(i, file) = fopen(FDATA(i, filename), "wb");
    if (FDATA(i, file) == NULL) {
        bt -> error_code = 1;
        ret = TREE_FAIL;
        break;
    }

    /* Steal access to a buffer & initialize file */
    buffer = s = FDATA(i, bufferlist) -> buffer;
    blksize = SIZES(i, block_size);
    memset(buffer, 0, blksize);

    /* Signature string */
    strcpy(s, SIGNATURE);
    strcat(s, i == NDX ? SIG_INDEX : SIG_DATA);
    s += strlen(s) + 1;

    /* Buffer size information */
```

```
    sizes = (Sizes *) s;
    *sizes = bt -> fdata[i].sizes;

    /* Offsets to root block and some free blocks */
    s += sizeof(Sizes);
    dof = (DISKOFFSET *) s;
    *dof = blksize;   /* root is first block */
*(dof+1) = 0;     /* and the empty block list */

if (fwrite(buffer, blksize, 1, FDATA(i, file)) != 1) {
    bt -> error_code = 4;
    ret = TREE_FAIL;
    break;
}

memset(buffer, 0, blksize);
bc = (BlockControl *) buffer;
bc -> bfree = sizeof(BlockControl);
bc -> next       = 0L;

if (i == NDX) {
    /* create a block of maxkey/offset */
    bc -> blockstate = sNode;
    s = buffer + bc -> bfree;
    length = GETKEYSIZE(maxkey);
    memcpy(s, maxkey, length);
    s += length;
    bc -> bfree += length + sizeof(DISKOFFSET);

    /* write out levels blocks,
                            each pointing to the next */
    for (j = 1; j <= SIZES(i, levels); j++) {
        bc -> parent = (DISKOFFSET) blksize * (j - 1);
        *((DISKOFFSET *) s) =
                        (DISKOFFSET) blksize * (j + 1);
        /* last block points to data file */
        if (j == SIZES(i, levels)) {
            bc -> blockstate = sLeaf;
            *((DISKOFFSET *) s) =
                        SIZES(DAT, block_size);
        }
        if (fwrite(buffer, blksize, 1,
                        FDATA(i, file)) != 1) {
            bt -> error_code = 4;
            ret = TREE_FAIL;
            break;
        }
    }
}
else /* i == DAT */ {
    /* create a block with one entry: maxrec */
    bc -> blockstate = sLeaf;
    bc -> parent = (DISKOFFSET) SIZES(NDX, block_size) *
                                SIZES(NDX, levels);
    s = buffer + bc -> bfree;
    length = GETRECSIZE(maxrec);
```

```
            memcpy(s, maxrec, length);
            bc -> bfree += length;
            if (fwrite(buffer, blksize, 1,
                              FDATA(i, file)) != 1) {
                bt -> error_code = 4;
                ret = TREE_FAIL;
                break;
            }
        }
        fclose(FDATA(i, file));
        FDATA(i, file) = NULL;

        if (ret)
            break;
    }

    bt_bufrelease(bt, NDX);
    bt_bufrelease(bt, DAT);
    return ret;
}
```

块 0 中的数据允许文件是自标识的，bt_hdr. h 中显示了这个数据的布局。块中包含以下内容：

```
"A signature string"    // identifies the file as legitimate
struct sSize {
    unsigned block_size;// size of blocks in bytes
    unsigned split;     // split blocks using more than this
    unsigned merge;     // merge blocks using less than this
    unsigned levels;    // height of tree (.ndx only)
} sizes;
long root_block;        // first index block (.ndx only)
long free_block;        // first block in free-block chain
```

当调用 bt_new( ) 时，就会触及 bt_new. c 中的代码，并且这段代码将以一种直观的方式创建两个空文件。对于 . ndx 文件，将会创建适当数量的索引层，而 . dat 文件则是利用一个数据块创建的，该数据块中包含前面提到的具有最大键的记录。

bt_new( ) 的参数包括一个指向 Btree 结构的指针。B 树模块中的每个例程都将使用这个结构。它包含访问构成数据集的 . ndx 和 . dat 文件所需的所有信息。在 bt_hdr. h 中的 sBtree 中可以找到其布局。

B 树开始于两个子结构的数组：其中一个子结构用于关联的 . ndx 文件，另一个用于 . dat 文件。也许除了 BufferList 对象之外，这些结构的其他内部数据元素都很直观。给定数据集的索引和数据文件具有一组独立的缓冲区，它们是在一个链表中维护的，并且这个指针是链表的头部。缓冲区是由 bt_disk. c 中的代码管理的。有关磁盘块的大小、分割指针与合并指针等的值是在调用 bt_new( ) 时定义的。此后，无论何时通过 bt_open( ) 打开数据集，都会从数据文件自身当中加载这些值。

可以通过一对宏（#define NDX 0 和#define DAT 1）来简化对这两个结构的访问，这两个宏提供了符号下标。为了进一步简化代码，定义了另一对宏，用于减少访问结构的成员所需的代码：

```
#define FDATA(x,y)      (bt->fdata[x].y)
#define SIZES(x,y)      (bt->fdata[x].sizes.y)
```

假定 bt 是指向 Btree 的指针，这两个宏使得很容易访问每个文件的相关数据。例如，FDATA

(NDX, root_back) 访问根索引块的磁盘偏移量。所有模块都遵循如下约定：把 bt 用作指向本地 Btree 结构的指针的名称，因此可以自由地使用这些宏。

在这一对结构后面有一组 6 个指针，它们指向必须由用户提供的例程。这些指针是允许 B 树模块容纳广泛数据类型的关键。这些例程允许 B 树模块操纵记录以及它们的键，而无需实际地知道其中包含什么内容。为了查看它如何工作，检查 user. c 中的示例驱动程序使用的记录和键结构：

```
typedef struct UserRecord {
    char key[11];
    char name[25];
    char addr[25];
} UR;
```

这个数据结构包含一个键以及两个数据项。为了简单起见，它们被实现为固定长度的对象，但是正如我们将看到的，这不是 B 树模块的一项要求。示例驱动程序接受 x；y 形式的输入，把分号前面的字符串放入 name[ ] 中，并把分号后面的字符串放入 addr[ ] 中。把 name[ ] 的前 10 个字符取作键。因此，要求 user. c 插入 “Holmes；Sherlock” 将创建具有 7 个字节的键的记录 (“Holmes” 以及一个 null 终止符)。

现在，作为一个实现 6 个例程的示例，考虑确定记录中键的大小的代码：

```
unsigned UserGetKeyNRecSize(void *k) {
    return strlen( ((UR *) k) -> key ) + 1;
}
```

给这个例程传递一个 void 指针，可以假定它指向一个有效的数据记录。代码把它强制转换为一个指向用户记录的指针，对键调用 strlen( )，加上 1 (用于 null 终止符)，并返回值。

另请注意：这种技术不要求键是记录中的第一个数据项，甚至不要求键存在于单个位置中。例如，键可以是连接多个不同数据项的结果，并且这种连接可以在任何需要键的时间动态完成。

实际的数据块和索引块自身很简单。每个数据块都开始于一个如下形式的 BlockControl 结构 (来自于 bt_hdr. h)：

```
typedef struct sBlockControl {
    size_t     bfree;   /* Free data area begins here */
    BlockState blockstate;
    DISKOFFSET parent;  /* parent of this block */
    DISKOFFSET next;    /* when block belongs to a free chain */
} BlockControl;
```

字段 bfree 是从块的开始位置到下一个空闲字节的偏移量。块中的所有数据都存在于紧接在 BlockControl 结构之后的连续字节中，因此 bfree 的初始值是 sizeof (BlockControl)。BlockControl (它是 enum) 的可能的值只与索引块相关，并且由 sAvail、sNode 和 sLeaf 组成。sAvail 把块标识为空闲块。块中标记 sNode 的指针指向其他索引块，而 sLeaf 中的那些指针则指向数据块。最后，parent 是一个向上的指针，允许从数据块遍历回根，而 next 则用于存储空闲块的链。索引块中的数据由连续的键/DISKOFFSET 对组成。如前所述，键的长度是可变的，而 DISKOFFSET 被 typedef 为一个长整型，它是 fpos_t 的标准数据类型，用于保存磁盘地址。数据块中的数据只是简单地由一个接一个写入的记录组成。对于这两种块类型，B 树模块使用用户提供的函数来确定键和记录的大小，从而遍历数据。

程序清单6-5 用于打开B树数据集的代码

```
/*--- bt_open.c ---------------------------- Listing 6-5 ---------
 * Open a B-Tree dataset
 *
 * See user.c (Listing 6-9) for a test driver.
 *-------------------------------------------------------------*/

#include "bt_hdr.h"
#if defined(HELP1)
#define SHOW(x) x
#else
#define SHOW(x)
#endif

/* Open files, allocate buffers */
int bt_open(Btree *bt)
{
    int i, ret;
    char buffer[512], *s;

    /* Check values, allocate buffers */
    ret = TREE_OK;
    for (i=0; i < 2; i++) {
        FDATA(i,file) = fopen(FDATA(i,filename), "r+b");
        if (FDATA(i,file) == NULL) {
            bt -> error_code = 6;
            ret = TREE_FAIL;
            break;
        }
        FDATA(i, modified) = 0;
        SHOW(printf("bt_open file: %s, file: %Fp\n",
                        FDATA(i, filename), FDATA(i, file));)

        if (fread(buffer, 512, 1, FDATA(i,file)) != 1) {
            bt -> error_code = 7;
            ret = TREE_FAIL;
            break;
        }
        s = buffer;
        if (strcmp(s, i == NDX ?
                  SIGNATURE SIG_INDEX : SIGNATURE SIG_DATA)) {
            bt -> error_code = 8;
            ret = TREE_FAIL;
            break;
        }

        s += strlen(s) + 1;
        bt -> fdata[i].sizes = *((Sizes *) s);
        s += sizeof(Sizes);
        bt -> fdata[i].root_block = *((DISKOFFSET *) s);
        s += sizeof(DISKOFFSET);
        bt -> fdata[i].free_block = *((DISKOFFSET *) s);

        ret = bt_bufinit(bt, i);
        if (ret)
```

```
            break;
        }

        return ret;
    }

    /* Flush buffers and shut down */
    int bt_close(Btree *bt)
    {
        int i, ret;

        ret = bt_flush(bt);
        for (i=0; i<2; i++) {
            if (FDATA(i,file)) {
                if (fclose(FDATA(i,file))) {
                    bt -> error_code = 9;
                    ret = TREE_FAIL;
                }
                FDATA(i,file) = NULL;
            }
            bt_bufrelease(bt, i);
        }

        return ret;
    }
```

2. 使用数据集：bt_data. c

bt_data. c 模块（程序清单 6-6）包含用于执行树遍历、查找、添加和删除操作的代码。用于穷尽地遍历树的代码是一个良好的起点。入口点是 bt_walk( )。给例程传递两个参数：一个指向要遍历的树的指针和一个要对树的每个记录执行的函数。可以在 user. c 中找到这样一个函数的例子，即 DisplayFunc( )。该函数通过 user. c 中的示例驱动程序的“w”选项使用，并且显示当前记录。在执行少量的初始化任务之后，bt_walk( ) 将调用 bt_walk_down( ) 执行实际的工作。该函数接受多个参数：

```
btint bt_walk_down(Btree *bt,       /* the tree                */
                   int f,           /* NDX or DAT?             */
                   DISKOFFSET block, /* the block              */
                   DoFunc df,       /* an action function      */
                   int level,       /* depth of search         */
                   int mode)        /* SELECTIVE or EXHAUSTIVE */
```

这个函数检查通过 f 和 block 的组合指定的块。如果 f 是 NDX，那么就假定 block 是索引块的偏移量。否则，block 就是数据块。函数 df 是用户提供的函数，为数据集中的每个记录调用它，而 level 是一个指示递归深度的计数器。mode 标志确定是将检查树的所有可能的分支（EXHAUSTIVE），还是只将检查沿着通往查找 bt- > SearchKey 中存储的键的路径上的那些分支（SELECTIVE）。提供这两种模式使得相同的子例程可以用于遍历整棵树，或者只是查找它的一部分。

程序清单6-6 用于添加、删除和查找记录的代码

```
/*--- bt_data.c -------------------------- Listing 6-6 ---------
 * Add, remove, and find records
 *
 * See test driver is in user.c  (Listing 6-9)
 *-------------------------------------------------------------*/

#include "bt_hdr.h"

#if defined(HELP1)
#define SHOW(x) x
#else
#define SHOW(x)
#endif

/*
 * Support routines for find, search, walk
 */

int FindFunc(Btree *bt, void *rec) {
    int compare = KEY2RECCMP(bt->SearchKey, rec);
    if (compare > 0)
        return TREE_OK;
    else if (compare == 0) {
        if (bt->FoundRec)
            /*
             * find calls us with FoundRec = a
             * data area where the round record is
             * to be copied
             */
            memcpy(bt->FoundRec, rec, GETRECSIZE(rec));
        else
            /*
             * delete and add call with FoundRec == NULL. When
             * we find the record, we set FoundRec non-NULL
             * to indicate our success.
             */
            bt->FoundRec = rec; /* irrelevant, non-NULL value */
    }
    else
        bt->FoundRec = NULL;
    return TREE_FAIL;
}

int LocateFunc(Btree *bt, void *rec) {
    int compare = KEY2RECCMP(bt->SearchKey, rec);
    if (compare > 0)
        return TREE_OK;
    else if (compare == 0)
        bt->FoundRec = rec;
    else
        bt->FoundRec = NULL;
    return TREE_FAIL;
}

/*
```

```
 * Walk the tree, calling user function df for each data record
 */
#define SELECTIVE  0  /* modes for bt_walk_down */
#define EXHAUSTIVE 1
int bt_walk_down(Btree *bt,          /* the tree                */
                 int f,              /* NDX or DAT?             */
                 DISKOFFSET block,   /* the block               */
                 DoFunc df,          /* an action function      */
                 int level,          /* depth of search         */
                 int mode)           /* SELECTIVE or EXHAUSTIVE */
{
    char *s;
    BlockControl *bc;
    unsigned offset, keysize;
    int retval;
    DISKOFFSET datablock;

    s = bt_getblock4read(bt, f, block);
    if (!s) {
        bt->error_code = 7;
        return TREE_FAIL;
    }

    bc = (BlockControl *) s;
    retval = TREE_OK;
    if (f == NDX) {
        offset = sizeof(BlockControl);
        while (offset < bc->bfree) {
            keysize = GETKEYSIZE(s + offset);
            datablock = *((DISKOFFSET *) (s + offset + keysize));
            if (mode == SELECTIVE) {
                if (KEY2KEYCMP(bt->SearchKey, s+offset) > 0) {
                    offset += keysize + sizeof(DISKOFFSET);
                    continue;
                }
            }
            retval =  bt_walk_down(bt,
                      bc->blockstate == sLeaf ? DAT : NDX,
                      datablock, df, level + 1, mode);
            if (retval)
                break;

            /* recursive calls may invalidate our buffer */
            s = bt_getblock4read(bt, f, block);
            bc = (BlockControl *) s;
            offset += keysize + sizeof(DISKOFFSET);
        }
    }
    else { /* f == DAT */
        bt->CurrentDataBlock = block;
        offset = sizeof(BlockControl);
        while (offset < bc->bfree) {
            retval = df(bt, s + offset); /* do user's bidding */
            if (retval)
                break;
            offset += GETRECSIZE(s + offset);
        }
```

```
    }

    return retval;
}

/*
 * Systematically walk the tree
 */
int bt_walk(Btree *bt, DoFunc df) {
    bt->CurrentDataBlock = 0L;
    return bt_walk_down(
        bt, NDX, FDATA(NDX, root_block), df, 0, EXHAUSTIVE);
}

/*
 * Search for a record
 */
int bt_find(Btree *bt, void *rec, void *key) {
    int retval;

    bt->SearchKey = key;
    bt->FoundRec = rec;
    retval = bt_walk_down(
        bt, NDX, FDATA(NDX, root_block), FindFunc, 0, SELECTIVE);
    if (retval && bt->FoundRec)
        return TREE_OK; /* found it */
    else
        return TREE_FAIL;
}

/*
 * update index blocks
 */
DISKOFFSET bt_fix_index (Btree *bt,
                         void *replacekey,
                         DISKOFFSET fixblock,
                         DISKOFFSET oldblock,
                         void *newkey,
                         DISKOFFSET newblock)
{
    /*
     * This is a very heavily used routine that performs
     * major updates to index blocks. It can insert a new
     * index pointer, replace a pointer, or delete a pointer.
     * It will split the block if it overflows.
     *
     * If replacekey == NULL: (inserting a new key)
     *   In index block fixblock, find the key/DISKOFFSET pair
     *   that has oldblock for its DISKOFFSET. Then,
     *    1) update this key/DOF pair so that DOF is newblock
     *    2) insert newkey/oldblock as a new pair just in front
     *    3) split index block if now too big
     *
     * If replacekey != NULL (replacing an existing key)
     *   In index block fixblock, find the key/DISKOFFSET pair
     *   that has oldblock for its DISKOFFSET This pair will
     *   actually be replacekey/oldblock. The value of replacekey
```

```
     *   is used both as a flag and to allow the routine to see
     *   quickly if it will need to split the block. Then,
     *    1) update this key/DOF pair so that key is newkey (**)
     *    2) split index block if now too big
     *    3) promulgate key replacement upwards if key/DOF pair
     *       is the last key/DOF pair in the block
     *
 *   (**) If newkey is NULL, then the key/DISKOFFSET pair is
 *        simply deleted and not replaced.
 *
 * Note that we do not actually do these operations in this
 * order. We proceed by first determining if we need to split
 * the index block. If so, we allocate a new block and then
 * recursively call ourself to place the new entry into the
 * next level up. Only if this succeeds do we actually commit
 * a change to the database.
 *
 * We return the DISKOFFSET of oldblock and newblock's new
 * parent.
 */

char *pblock,           /* current index block (fixblock) */
     *pblock1 = NULL;   /* added index block, if we need one */
char *keycopy;
BlockControl *bc, *bc1 = NULL;
DISKOFFSET newindexblock = 0, newparent;
BufferList *bl = NULL;
unsigned keysize, newsize, previous;
unsigned offset, lastoffset, replacesize;
int delta;
int bt_merger(Btree *, int, DISKOFFSET);

SHOW(printf("bt_fix_index/a: %Fp, %d, %ld\n",
                                        bt, NDX, fixblock);)
pblock = bt_getblock4write(bt, NDX, fixblock);
if (!pblock)
    return 0L;
bc = (BlockControl *) pblock;
newsize = -sizeof(DISKOFFSET);
if (newkey)
    newsize = GETKEYSIZE(newkey);
replacesize = 0;
if (replacekey)
    replacesize = GETKEYSIZE(replacekey) + sizeof(DISKOFFSET);
delta = (int) newsize + (int) sizeof(DISKOFFSET) -
                                        (int) replacesize;
if (bc->bfree + delta > SIZES(NDX,split)) {
    int oldblockseen = 0;

    if (bc->parent == 0) {
        /*
         * We're trying to split the root block!
         * Complain & abort.
         */
        bt->error_code = 17;
     return 0L;
```

```
}

/* allocate a new index block */
bl = bt_getnew4write(bt,NDX);
if (!bl)
    return 0L;
pblock1 = bl -> buffer;
newindexblock = bl -> offset;
bc1 = (BlockControl *) pblock1;
bc1 -> blockstate = sNode;

/* locate the break point */
offset = sizeof(BlockControl);
previous = 0;
while (offset < bc->bfree/2) {
    previous = offset;
    keysize = GETKEYSIZE(pblock + offset);
    if (*((DISKOFFSET *)(pblock + offset + keysize)) ==
                                            oldblock)
        oldblockseen = 1;
    offset += keysize + sizeof(DISKOFFSET);
}
if (previous == 0) {
    /*
     * Major problem. there is one key that it so large
     * we can't divide the block. This should never
     * happen, as keys should be small relative to the
     * size of the block. Complain and abort. The index
     * may now be corrupt.
     */
    bt->error_code=18;
    return 0L;
}
keysize = GETKEYSIZE(pblock + previous);
keycopy = malloc(keysize);
memcpy(keycopy, pblock + previous, keysize);

/* make sure we can alter our parent */
newparent =
    bt_fix_index(bt,
                 NULL,
                 bc->parent,
                 fixblock,
                 keycopy,
                 newindexblock);
free (keycopy);
if (!newparent)
    return 0L;

/* refresh our pointers */
SHOW(printf("bt_fix_index/b: %Fp, %d, %ld\n",
                            bt, NDX, fixblock);)
pblock  = bt_getblock4write(bt,NDX,fixblock);
SHOW(printf("bt_fix_index/c: %Fp, %d, %ld\n",
                            bt, NDX, newindexblock);)
pblock1 = bt_getblock4write(bt,NDX,newindexblock);
if (!pblock || !pblock1)
```

```
    return 0L;
bc  = (BlockControl *) pblock;
bc1 = (BlockControl *) pblock1;
bc -> parent = bc1 -> parent = newparent;

/* divide'em up */
memcpy(pblock1 + bc1->bfree, pblock + offset,
                                bc->bfree - offset);
bc1 -> bfree += bc -> bfree - offset;
bc  -> bfree = offset;

/* common parent */
bc1 -> parent = bc -> parent = newparent;

/* same level, same use */
bc1 -> blockstate = bc -> blockstate;

/* tell children of new index block */
/* ... about their new parent */
offset = sizeof(BlockControl);
while(offset < bc1 -> bfree) {
    char *pblock2;
    BlockControl *bc2;
    DISKOFFSET goal;

    keysize = GETKEYSIZE(pblock1 + offset);
    goal = *((DISKOFFSET *)(pblock1 + offset + keysize));
    SHOW(printf("bt_fix_index/d: %Fp, %d, %ld\n",
         bt, bc1->blockstate == sLeaf ? DAT : NDX,
         goal);)
    pblock2 = bt_getblock4write(bt,
              bc1->blockstate == sLeaf ? DAT : NDX,
              goal);
    if (!pblock2)
        return TREE_FAIL;
    bc2 = (BlockControl *) pblock2;
    bc2 -> parent = newindexblock;
        /* freshen our pointers */
        SHOW(printf("bt_fix_index/e: %Fp, %d, %ld\n",
             bt, NDX, newindexblock);)
        pblock1 = bt_getblock4write(bt, NDX, newindexblock);
        bc1 = (BlockControl *) pblock1;
        offset += keysize + sizeof(DISKOFFSET);
    }

    /* transfer our attention to new index block */
    if (!oldblockseen)
        fixblock = newindexblock;
}

/*
 * Now, after many struggles, it is safe to insert our new
 * pointer into the block pointed to by pblock. We start by
 * locating the key/DISKOFFSET pair that has oldblock for
 * its DISKOFFSET...
 */
pblock =
```

```
    bt_getblock4write(bt, NDX, fixblock); /* fresh pointer */
if (!pblock)
    return 0L;
bc = (BlockControl *) pblock;

offset = sizeof(BlockControl);
lastoffset = 0;
keysize = GETKEYSIZE(pblock + offset);
while (offset < bc ->bfree) {
    keysize = GETKEYSIZE(pblock + offset);
    if (*((DISKOFFSET *)(pblock + offset + keysize)) ==
                                                oldblock)
        break;

    lastoffset = offset;
    offset += keysize + sizeof(DISKOFFSET);
}
if (*((DISKOFFSET *)(pblock + offset + keysize)) !=
                                                oldblock) {
    /* something is very wrong */
    bt -> error_code = 19;
    return 0L;
}

if (replacekey) { /* change keys but not DISKOFFSETs */
    DISKOFFSET saveblock;

    /* move remaining data up */
memmove(pblock + offset + replacesize + delta,
        pblock + offset + replacesize,
        bc->bfree - offset - replacesize);

/* insert newkey/oldblock */
if (newkey) { /* if NULL, we are actually deleting */
    memmove(pblock + offset, newkey, newsize);
    *((DISKOFFSET *)(pblock + offset + newsize)) =
                                            oldblock;
}
bc->bfree += delta;

/*
 * Is this the last pointer in the block? If so, must
 * update our parent so that it knows about the change
 */
saveblock = fixblock;
if (offset + newsize + sizeof(DISKOFFSET) >=
                                            bc->bfree) {
    int delete_newkey = 0;
    if (!newkey && lastoffset) {
        /*
         * We are deleting the last key of a non-empty
         * block. We must promote a new last key.
         */
        newkey =
            malloc(GETKEYNRECSIZE(pblock + lastoffset));
        if (!newkey) {
```

```
                bt->error_code=16;
                return 0L;
            }
            REC2KEYCPY(pblock + lastoffset, newkey);
            delete_newkey = 1;
        }
        fixblock = bt_fix_index(bt, replacekey, bc->parent,
                                fixblock, newkey, 0L);

        if (delete_newkey)
            free(newkey);

        /* is the block empty? if so, delete it. */
        if (bc->bfree == sizeof(BlockControl)) {
            if (bt_releaseblock(bt, NDX, saveblock))
                fixblock = 0L;
        }
        else { /* should we merge the block? */
            if (bc->bfree < SIZES(NDX, merge))
                if (bt_merger(bt, NDX, saveblock))
                        fixblock = 0L;
               }
           }

           return fixblock;
       }
       else {
           /* update existing key/DISKOFFSET to point to newblock */
           *((DISKOFFSET *)(pblock + offset + keysize)) = newblock;

           /* insert newkey/oldblock */
           memmove(pblock + offset + newsize + sizeof(DISKOFFSET),
                   pblock + offset,
                   bc->bfree - offset);
           memmove(pblock + offset, newkey, newsize);
           *((DISKOFFSET *)(pblock + offset + newsize)) = oldblock;
           bc -> bfree += newsize + sizeof(DISKOFFSET);

           return fixblock;
       }
}

/*
 * Add a new record
 *
 * Our general add strategy is:
 *
 *  1) Locate the spot for the insertion
 *  2) Split the data block if necessary (bt_split)
 *  3) Split index blocks if necessary (bt_fix_index)
 *  4) Re-locate the spot for the insertion
 *  5) Insert can now proceed without further splits
 *
 * Note that we actually commit step 3 to the database
 * before step 2--this allows us to abort if step 3
 * fails.
 */
```

```
/* split bottom data block in half */
int bt_split(Btree *bt)
{
    char *pblock, *pblock1, *keycopy;
    BufferList *bl;
    DISKOFFSET datablock, datablock1, newparent;
    BlockControl *bc, *bc1;
    unsigned previous, offset;

    datablock = bt->CurrentDataBlock;
SHOW(printf("bt_split/a: %Fp, %d, %ld\n",
                        bt, DAT, datablock);)
pblock = bt_getblock4write(bt, DAT, datablock);
if (!pblock) {
    bt->error_code = 7;
    return TREE_FAIL;
}

bc = (BlockControl *) pblock;
SHOW(printf("bt_split: parent of %ld is %ld\n",
                        datablock, bc->parent);)

offset = sizeof(BlockControl);
previous= 0;
while (offset < bc->bfree/2) {
    previous = offset;
    offset += GETRECSIZE(pblock + offset);
}

if (previous == 0) { /* major problem - can't divide block */
    bt -> error_code = 15;
    return TREE_FAIL;
}

/* get a copy of the break record's key */
keycopy = malloc(GETKEYNRECSIZE(pblock + previous));
if (!keycopy) {
    bt -> error_code = 16;
    return TREE_FAIL;
}
REC2KEYCPY(pblock + previous, keycopy);

/* get a new block */
bl = bt_getnew4write(bt, DAT);
if (!bl) {
    bt->error_code = 7;
    return TREE_FAIL;
}

pblock1 = bl -> buffer;
datablock1 = bl -> offset;
bc1 = (BlockControl *) pblock1;
bc1 -> blockstate = sLeaf;

/*
 * do insert into upper level blocks first. If this fails,
 * we have not committed any changes to the database and we
 * can just abort.
```

```
     */
    SHOW(printf("calling bt_fix_index(%Fp,%s,%ld,%ld,%ld\n",
                    bt,keycopy,bc->parent,datablock,datablock1);)
    newparent =
        bt_fix_index(bt, NULL, bc->parent, datablock,
                     keycopy, datablock1);
    free(keycopy);
    if (!newparent)
        return TREE_FAIL;

    /* Now divide the data */
    SHOW(printf("bt_split/b: %Fp, %d, %ld\n",
                                bt, DAT, datablock);)
    pblock  = bt_getblock4write(bt, DAT, datablock);

    SHOW(printf("bt_split/c: %Fp, %d, %ld\n",
                                bt, DAT, datablock1);)
    pblock1 = bt_getblock4write(bt, DAT, datablock1);

    bc  = (BlockControl *) pblock;
    bc1 = (BlockControl *) pblock1;
    memcpy(pblock1 + bc1->bfree, pblock + offset,
                                bc->bfree - offset);
    bc1 -> bfree += bc -> bfree - offset;
    bc  -> bfree = offset;
    bc1 -> parent = bc -> parent = newparent; /* common parent */

    return TREE_OK;
}

int bt_add_record(Btree *bt, void *rec, void *key)
{
    char *s;
    BlockControl *bc;
    unsigned offset, datasize;
    DISKOFFSET datablock;
    int retval;

  top:
    datablock = bt->CurrentDataBlock;
    SHOW(printf("bt_add_record: %Fp, %d, %ld\n",
                                bt, DAT, datablock);)
    s = bt_getblock4write(bt, DAT, datablock);
    if (!s) {
        bt->error_code = 7;
        return TREE_FAIL;
    }
   bc = (BlockControl *) s;
   datasize = GETRECSIZE(rec);

   if (bc->bfree + datasize > SIZES(DAT,split)) {
       if (bt_split(bt))
           return TREE_FAIL; /* couldn't do it */
       else {
           /* re-locate position for block to insert */
           retval = bt_walk_down(
               bt, NDX, FDATA(NDX, root_block),
```

```
                LocateFunc, 0, SELECTIVE);
            if (!retval)
                return retval;
            goto top; /* try again */
        }
    }

    /* looks good. make a space */
    offset = sizeof(BlockControl);
    while (offset < bc->bfree) {
        if (KEY2RECCMP(key, s + offset) < 0)
            break; /* goes before this record */
        offset += GETRECSIZE(s + offset);
    }
    /* make our moves */
    memmove(s + offset + datasize, s + offset,
                            bc->bfree - offset);
    memmove(s + offset, rec, datasize);
    bc->bfree += datasize;
    return TREE_OK;
}

int bt_add(Btree *bt, void *rec, void *key)
{
    int retval = TREE_OK;

    bt->SearchKey = key;
    bt->FoundRec = NULL;
    retval = bt_walk_down(
        bt, NDX, FDATA(NDX, root_block),
        LocateFunc, 0, SELECTIVE);
    if (!retval) {
        bt->error_code = 14;
        return TREE_FAIL; /* should never occur */
    }

    /*
     * Now, if FoundRec == NULL, then the current data record is
     * the right place for the new record and the new record
     * does not exist.
     *
     * If FoundRec != NULL, then FoundRec points to a data record
     * with the same key as the record to be inserted
     */

    retval = TREE_OK;
    if (bt->FoundRec) {
        if (!bt->duplicatesOK)
            retval = TREE_FAIL;
    }

    if (!retval)
        retval = bt_add_record(bt, rec, key);

    return retval;
}
```

```
/*
 * Delete a record
 *
 * Our general delete strategy starts in
 * bt_delete_record() and is:
 *
 *  1) Locate & delete offending record
 *
 *  2) If record was the rightmost record in its block,
 *     update key in parent index block(s). Note that
 *     this process involves removing an old key and
 *     inserting a new one: if the new key is larger
 *     than the old one, the block could potentially
 *     exceed the split limit. This could in turn split
 *     the parent, etc. Alternatively, the parent
 *     could need to merged. All of this is handled by
 *     bt_fix_index().
 *
 *  3) If datablock that held the deleted record is now
 *     smaller than merge, locate and examine the
 *     left and then the right sisters of this block.
 *     If the current block can be combined with
 *     either, then we will merge them. To merge,
 *     we first shift the tree so that the target
 *     datablocks have the same index block as their
 *     parent, we combine the blocks, and then drop
 *     one entry from the parent index block.
 */
/*
 * Routines to merge blocks
 */
#define LEFT 0
#define RIGHT 1

/* find sister of child in parent at level==0 */
DISKOFFSET bt_find_sister(Btree *bt,
                          DISKOFFSET parent,
                          DISKOFFSET child,
                          int direction,
                          int level) {

    char *pblock;
    BlockControl *bc;
    unsigned offset, lastoffset, keysize;
    DISKOFFSET dof;

    pblock = bt_getblock4read(bt, NDX, parent);
    if (!pblock)
        return 0L;
    bc = (BlockControl *) pblock;

    /* find key/diskoffset with child for its diskoffset */
    lastoffset = 0;
    offset = sizeof(BlockControl);
    dof = 0;
    while (offset < bc -> bfree) {
        keysize = GETKEYSIZE(pblock + offset);
```

```
        dof = *((DISKOFFSET *)(pblock + offset + keysize));
        if (dof == child)
            break;
        lastoffset = offset;
        offset += keysize + sizeof(DISKOFFSET);
    }

    if (dof != child) {
        bt->error_code = 22;
        return 0L;
    }

    if (direction == LEFT) {
        if (lastoffset) { /* it is in this block */
            keysize = GETKEYSIZE(pblock + lastoffset);
            dof =
                *((DISKOFFSET *)(pblock + lastoffset + keysize));
            while (level > 0) { /* unwind the chain */
                level--;
            pblock = bt_getblock4read(bt, NDX, dof);
            if (!pblock)
                return 0L;
            bc = (BlockControl *) pblock;
            offset = sizeof(BlockControl);

            /* find rightmost entry */
            while (offset < bc -> bfree) {
                keysize = GETKEYSIZE(pblock + offset);
                dof = *((DISKOFFSET *)(pblock + offset +
                                                keysize));
                offset += keysize + sizeof(DISKOFFSET);
            }
        }
        return dof;
    }
    else if (bc->parent)
        return bt_find_sister(bt, bc->parent, parent,
                              LEFT, level + 1);
    else
        return 0L;
}
else { /* direction == RIGHT */
    keysize = GETKEYSIZE(pblock + offset);
    offset += keysize + sizeof(DISKOFFSET);

    if (offset < bc->bfree) { /* it is in this block */
        keysize = GETKEYSIZE(pblock + offset);
        dof = *((DISKOFFSET *)(pblock + offset + keysize));
        while (level > 0) { /* unwind the chain */
            level--;
            pblock = bt_getblock4read(bt, NDX, dof);
            if (!pblock)
                return 0L;
            bc = (BlockControl *) pblock;
            offset = sizeof(BlockControl);
            keysize = GETKEYSIZE(pblock + offset);
            dof =
```

```
                *((DISKOFFSET *)(pblock + offset + keysize));
        }
        return dof;
    }
    else
    if (bc->parent)
        return bt_find_sister(bt, bc->parent, parent,
                              RIGHT, level + 1);
    else
        return 0L;
    }
}

/* merge left into right, if possible */
int bt_do_merge(Btree *bt,
                int f,  /* DAT or NDX */
                DISKOFFSET left,
                DISKOFFSET right) {
    char *pleft, *pright;
    BlockControl *bcleft, *bcright;
    void *oldkey;
    unsigned offset, recsize;

    pleft = bt_getblock4read(bt, f, left);
    if (!pleft)
        return TREE_FAIL;
    pright = bt_getblock4read(bt, f, right);
    if (!pright)
        return TREE_FAIL;

    bcleft  = (BlockControl *) pleft;
    bcright = (BlockControl *) pright;

    if (bcleft->bfree + bcright->bfree - sizeof(BlockControl) <
            SIZES(f, split)) { /* let's do it! */
        unsigned moving;

        /* get modifiable copies */
        pleft = bt_getblock4write(bt, f, left);
        if (!pleft)
            return TREE_FAIL;
        pright = bt_getblock4write(bt, f, right);
        if (!pright)
            return TREE_FAIL;

        bcleft  = (BlockControl *) pleft;
        bcright = (BlockControl *) pright;

        /* this many new bytes */
        moving = bcleft->bfree - sizeof(BlockControl);

        /* make room & copy */
        memmove(pright + sizeof(BlockControl) + moving,
                pright + sizeof(BlockControl),
                bcright->bfree);
        memmove(pright + sizeof(BlockControl),
                pleft  + sizeof(BlockControl),
                moving);
```

```
bcright->bfree += moving;

/* now, to discard the left block ... */

/* ... we first find & copy the last record's key */
offset = sizeof(BlockControl);
if (f == NDX)
    recsize = GETKEYSIZE(pleft + offset) +
                                sizeof(DISKOFFSET);
else
    recsize = GETRECSIZE(pleft + offset);
while (offset + recsize < bcleft->bfree) {
    offset += recsize;
    if (f == NDX)
        recsize = GETKEYSIZE(pleft + offset) +
                                sizeof(DISKOFFSET);
    else
        recsize = GETRECSIZE(pleft + offset);
}

if (f == NDX) {
    unsigned keysize;

    keysize = GETKEYSIZE(pleft + offset);
    oldkey = malloc(keysize);
    if (!oldkey) {
            bt->error_code = 16;
            return TREE_FAIL;
    }
    memmove(oldkey, pleft + offset, keysize);
}
else {
    oldkey = malloc(GETKEYNRECSIZE(pleft + offset));
    if (!oldkey) {
        bt->error_code = 16;
        return TREE_FAIL;
    }
    REC2KEYCPY(pleft + offset, oldkey);
}

/* ... and then run up the tree, deleting oldkey */
if (!bt_fix_index(bt, oldkey, bcleft->parent,
                  left, 0L, 0L))
    return TREE_FAIL;
free (oldkey);

/* ... and finally free the block */
if (bt_releaseblock(bt, f, left))
           return TREE_FAIL;

       /* tell children about their new parent */
       if (f == NDX) {
           pright = bt_getblock4write(bt, f, right);
           if (!pright)
               return TREE_FAIL;

           bcright = (BlockControl *) pright;
```

```
            moving += sizeof(BlockControl);
            offset  = sizeof(BlockControl);
            while(offset < moving) {
                char *pblock2;
                BlockControl *bc2;
                DISKOFFSET goal;
                unsigned keysize;

                keysize = GETKEYSIZE(pright + offset);
                goal = *((DISKOFFSET *)(pright + offset +
                                                keysize));
                SHOW(printf("bt_do_merge: %Fp, %d, %ld\n",
                       bt, bcright->blockstate == sLeaf ? DAT :
                       NDX, goal);)
                pblock2 = bt_getblock4write(bt,
                          bcright->blockstate == sLeaf ? DAT :
                          NDX, goal);
                if (!pblock2)
                    return TREE_FAIL;
                bc2 = (BlockControl *) pblock2;
                bc2 -> parent = right;

                /* freshen our pointers */
                pright = bt_getblock4write(bt, NDX, right);
                bcright = (BlockControl *) pright;
                offset += keysize + sizeof(DISKOFFSET);
            }
        }

        return TREE_OK;
    }
    else
        return TREE_FAIL; /* didn't fit */
}

/* main merge routine */
int bt_merger(Btree *bt, int f, DISKOFFSET block)
{
    char *pblock;
    BlockControl *bc;
    DISKOFFSET sister;

    pblock = bt_getblock4read(bt, f, block);
    if (!pblock)
        return TREE_FAIL;
    bc = (BlockControl *) pblock;

    if (bc->bfree < SIZES(f,merge)) { /* try to do a merge! */
        /* look left */
        sister = bt_find_sister(bt, bc->parent, block, LEFT, 0);
        if (sister && !bt_do_merge(bt, f, sister, block))
            return TREE_OK;

        /* look right */
        sister = bt_find_sister(bt, bc->parent, block, RIGHT, 0);
        if (sister && !bt_do_merge(bt, f, block, sister))
```

```
            return TREE_OK;
    }

    if (!bt->error_code)
        return TREE_OK;
    else
        return TREE_FAIL;
}

int bt_delete_record(Btree *bt)
{
    DISKOFFSET block;
    char *pblock;
    BlockControl *bc;
    unsigned offset, recsize, lastoffset;
    int compare, retval = TREE_OK;

    /* target record should be in this block */
    block = bt->CurrentDataBlock;
    pblock = bt_getblock4write(bt, DAT, block);
    if (!pblock)
        return TREE_FAIL;
    bc = (BlockControl *) pblock;

    /* find the record */
    offset = sizeof(BlockControl);
    lastoffset = 0;
    while (offset < bc->bfree) {
        recsize = GETRECSIZE(pblock + offset);
        compare = KEY2RECCMP(bt->SearchKey, pblock + offset);
        if (compare < 0) {
        bt -> error_code = 20;
        return TREE_FAIL; /* should not happen */
    }
    else if (compare == 0)
        break;
    lastoffset = offset;
    offset += recsize;
}

if (offset >= bc->bfree) {
    bt -> error_code = 20;
    return TREE_FAIL;
}

/* expunge the record */
bc->bfree -= recsize;
if (bc->bfree > offset) { /* not the last record */
    memmove(pblock + offset,
            pblock + offset + recsize,
            bc->bfree - offset);
    retval = bt_merger(bt, DAT, block);
}
else {
    /* deleting the last record in a block. must promulgate
       the key of the new last record upward.
     */
```

```
    void *oldkey, *newkey = NULL;
    oldkey = malloc(GETKEYNRECSIZE(pblock + offset));
    if (!oldkey) { bt->error_code=16; return TREE_FAIL; }
    REC2KEYCPY(pblock + offset, oldkey);

    /* is there a new last record? */
    if (lastoffset) {
        newkey = malloc(GETKEYNRECSIZE(pblock + lastoffset));
        if (!newkey) {
            bt->error_code = 16;
            return TREE_FAIL;
        }
        REC2KEYCPY(pblock + lastoffset, newkey);
    }

    /* run up the tree, replacing oldkey with newkey */
    if (!bt_fix_index(bt, oldkey, bc->parent, block,
                      newkey, 0L))
        retval = TREE_FAIL;

    if (!retval) {
        if (lastoffset) /* merge if needed */
                retval = bt_merger(bt, DAT, block);

            else /* this block is empty */
                retval = bt_releaseblock(bt, DAT, block);
        }

        free(oldkey);
        if (newkey)
            free(newkey);
    }

    return retval;
}

int bt_delete(Btree *bt, void *key)
{
    int retval;

    bt->SearchKey = key;
    bt->FoundRec = NULL;
    retval = bt_walk_down(
        bt, NDX, FDATA(NDX, root_block), FindFunc, 0, SELECTIVE);
    if (retval && bt->FoundRec)
        return bt_delete_record(bt);
    else
        return TREE_FAIL;
}
```

现在应该很容易理解 bt_walk_down( )。在检索了指定块的副本之后，bt_walk_down( ) 将递归地调用它自身以处理下一个索引块层，或者调用用户提供的动作函数来遍历数据记录。用于遍历索引块的代码演示了在模块中别的位置使用的许多习惯语法。

```
offset = sizeof(BlockControl);
while (offset < bc->bfree) {
    keysize = GETKEYSIZE(s + offset);
    datablock = *((DISKOFFSET *) (s + offset + keysize));
    if (mode == SELECTIVE) {
        if (KEY2KEYCMP(bt->SearchKey, s+offset) > 0) {
            offset += keysize + sizeof(DISKOFFSET);
            continue;
        }
    }
    retval =  bt_walk_down(bt,
              bc->blockstate == sLeaf ? DAT : NDX,
              datablock, df, level + 1, mode);
    if (retval)
        break;

    /* recursive calls may invalidate our buffer */
    s = bt_getblock4read(bt, f, block);
    bc = (BlockControl *) s;
    offset += keysize + sizeof(DISKOFFSET);
}
```

计数器 offset 被初始化为指向第一个数据字节的指针。由于数据紧接在块的 BlockControl 结构之后开始，只需要把 offset 设置为 sizeof（BlockControl）。然后，只要 offset 小于或等于 bt- > bfree，代码就会在通过 offset 指定的位置检查键。注意 offset 确定是偏移量：它指定一个相对于 s 的位置，其中 s 是一个通过调用 bt_getblock4read( ) 返回的指针，它最初用于检索块并将其存储在内存中。这种分隔很重要，因为如你稍后将看到的，s 的值可能在这个循环的连续迭代期间发生变化。可以通过使用宏 GETKEYSIZE 在 s + offset 处确定键的大小。这个宏扩展进通过 bt- > getkeysize 指定的函数的调用中。然后可以在 s + offset + keysize 处找到对应的 DISKOFFSET。最后，假定 mode 不是 SELECTIVE，bt_walk_down( ) 将再次调用它自身。一旦这个递归调用返回，循环就会通过再次调用 bt_getblock4read( ) 来刷新其指向磁盘块的指针。最后这一步很重要。尤其是在更高的索引块中，对 bt_walk_down( ) 的每次递归调用可能需要进行十多次其他的递归调用，其中每次递归调用都将加载一个索引块。磁盘块缓存将保存少量块，它将基于最近最少使用的原则丢弃它们。如果生成其他递归树访问，总是必须假定指向磁盘块的指针是无效的。

检查数据块的代码在结构上与用于扫描索引块的代码类似。额外的唯一创新是：这段代码将在 bt- > CurrentDataBlock 中保存当前数据块的偏移量。在对 bt_walk_down( ) 执行了 SELECTIVE 调用之后，调用者可以使用这个值来确定包含感兴趣的记录的数据块的位置。

3. 查找记录：bt_find( )

用于定位特定记录的代码构建于遍历整棵树的代码之上，并且开始于 bt_find( )。在此，将指向目标键和输出区域的指针加载进 Btree 结构中，并且调用 bt_walk_down( )。不过，对于这个调用，mode 是 SELECTIVE，并且动作函数是 FindFunc( )，这是由 B 树代码在 bt_data. c（程序清单 6-6）的开始处提供的一个函数。由于 mode 是 SELECTIVE，在每一层上只会检查一个索引块。然后，一旦找到正确的数据块，就会为它的每个数据记录调用 FindFunc( )。当找到正确的数据记录时，就把它复制到输出区域，并且发信号通知成功完成任务。注意：像这个模块中的许多例程一

样，FindFunc( ) 实际上具有两种操作模式：它可以从给定的记录查找并复制数据，或者它可以确定具有给定键的记录是否存在于数据集中。

4. 插入新记录：bt_add( )

添加和删除是两种最复杂的操作。其中，添加要容易一些，我们将从它开始介绍。添加的入口点是 bt_data. c 中的 bt_add( ) （程序清单 6-6）。这个例程首先使用熟悉的 bt_walk_down( ) 例程定位将用于存放新记录的数据块。如果合适的话，这种查找还允许例程拒绝插入具有重复键的记录。一旦确定正确的位置，就会调用 bt_add_record( ) 做实际的工作。如果新记录合适，就会简单地把它复制到适当的位置。如果不合适，就会调用 bt_split( ) 处理问题。对于 bt_split( )，我们现在就会遇到第一个重要的树操作子例程。

在找到粗略地把当前数据块分为两半的记录之后，bt_split( ) 就会调用 bt_fix_index( ) 使指向新块的指针插入到父索引块中。这个例程是整个 B 树模块真正的核心。它可以插入新的键和指针的组合、更新指针的键，或者删除键和指针。为任何一个索引块执行这些任务很简单；例如，当插入新的键和指针导致块溢出时就会出现问题。例程以一种直观的方式处理这些问题：它朝着根块递归，直到它可以看到操作将成功为止。然后，当释放递归栈时就会执行所有必要的更改。实际的块操作也很直观。唯一真正复杂的原因是：例程同时用于插入、修改和删除。不过，执行这些操作的代码段彼此独立，并且可以单独研究。

在插入期间，bt_fix_index( ) 面临的唯一问题是：插入新指针可能导致当前索引块包含比 split 更多的字节，从而需要进行块分割。如前所述，这种断点是由块的“充满度”决定的，而不是由块中记录的绝对数量决定的。这种方法允许数据块包含许多小记录，或者只是包含几个大记录。如果块溢出，就必须获得新的索引块，并且必须在上一级索引块中插入一个指向它的指针。这当然可能导致上一级索引块分割，依此类推。如果根块溢出，那么树的高度必须增加。当前代码不允许这样做（将代之以返回一个错误代码），但它很容易实现：只需检索一个新块，使之成为根块，并插入一个指向当前索引块的指针作为唯一的条目。后面的代码然后将把分割传播进这个新的根块中。

5. 删除记录：bt_delete( )

删除是 B 树模块中最具挑战性的部分。在确定感兴趣的记录存在后，bt_delete( ) 就会调用 bt_delete_record( ) 执行实际的工作。这个例程首先从数据块中删除想要的记录。这总是安全的，即使它把块变为空。不过，如果删除操作在块中留下了一些数据，就必须处理一种特殊情况：如果删除的记录是块中的最后一个记录，用于该块的索引块的键（即索引块中指向这个块的键）现在就出现了错误，并且必须校正它。在前面的图 6-25 中显示了用于生成新的向上终端键的这个过程的一个示例。这个过程可能要求修改多个层上的索引块，注意到这一点很重要。此外，由于我们允许可变长度的键，新的键可能小于或大于它所替换的键。因此，更改键实际上可能导致给定的索引块被分割或者被合并。同样，bt_fix_index( ) 会处理所有这些情况。

虽然用于分割两个索引块的代码嵌入在 bt_fix_index( ) 中，但是用于合并两个块的代码则位于单独的例程 bt_merger( ) 中。如果一个块需要被合并，这个例程首先会定位左边的兄弟块，然

后定位右边的兄弟块，并确定是否有可能进行合并。如果是，bt_merger( ) 就会调用 bt_do_merge ( ) 执行以下任务：

1. 把左边的块中的数据拖入右边的块中。
2. 从父索引块中删除左边的块的最右边的条目的键。
3. 把左边的块放到空闲链上。

查找左边或右边的兄弟块是 bt_find_sister( ) 的任务。你可能想到，这个过程可以像查找一个索引层以找到前一个或后一个条目一样简单。如果我们需要块中最左边的条目的左兄弟条目，可能需要在树中向上寻找若干层以便找到想要的条目。当我们需要查找块中最右边的条目的右兄弟条目时，也可能会发生相同的情形。这个过程被实现为递归查找。

注意：bt_fix_index( ) 在删除过程中起着主导作用。与以前一样，删除索引键可能导致索引块收缩，使得它需要与其兄弟块合并。这将导致对 bt_merger( ) 的嵌套的递归调用，依此类推。为了查看这个过程的图形表示，考虑以 bt_delete( ) 为根的调用图，如图 6-27 所示。

```
                                        递归地调用
bt_delete()
└=>-bt_walk_down()                      [itself]
└=>-bt_delete_record()
   └=>-bt_merger()
   |    └=>-bt_find_sister()            [itself]
   |    └=>-bt_do_merge()
   └=>-----└=>-bt_fix_index()           [itself] [bt_merger()]
```

图 6-27　以 bt_delete( ) 为根的调用图

6. 访问磁盘：bt_disk. c

通过模块 bt_disk. c（程序清单 6-7）来实现缓冲的磁盘访问。为每个 . ndx 和 . dat 文件都会维护一份磁盘缓冲区列表。主要通过 bt_getblock4read( ) 和 bt_getblock4write( ) 访问磁盘块。这些例程之间的唯一区别是：在 bt_getblock4write( ) 之后，假定块是“脏”的，并将在丢弃它之前把它写到磁盘。

当请求磁盘块时，就会查找缓冲区列表。如果块不存在，就会丢弃列表中的最后一个块，并读入新块。最近使用的块总是保存在列表的开头。在初始化期间，将分配足够多的索引块缓冲区，以确保在完全高度的树的每一层中至少有一个块可以在内存中同时存在。

程序清单 6-7　用于 B 树的磁盘例程和错误消息

```
/*--- bt_disk.c --------------------------- Listing 6-7 ---------
 * Disk-access routines and array of error messages
 *
 *-------------------------------------------------------------*/
#include "bt_hdr.h"

#if defined(HELP1)
#define SHOW(x) x
#else
#define SHOW(x)
```

```
#endif

char *ErrorText[] = {
  /*  0 */ "Not an error",
  /*  1 */ "bt_new: Could not create new data file",
  /*  2 */ "bt_new: Bad block_size value",
  /*  3 */ "bt_new: Bad split or merge values",
  /*  4 */ "bt_new: Could not write to new data file",
  /*  5 */ "bt_disk: Couldn't allocate buffers",
  /*  6 */ "bt_open: Couldn't open data file",
  /*  7 */ "bt_disk: Couldn't read data file",
  /*  8 */ "bt_open: Signature string missing",
  /*  9 */ "bt_close: Couldn't close file",
  /* 10 */ "bt_disk: Couldn't read/write file",
  /* 11 */ "bt_add: Attempted to add duplicate key",
  /* 12 */ "bt_split: Attempted to split block w/ NULL parent",
  /* 13 */ "bt_split: Couldn't find child in parent",
  /* 14 */ "bt_data: Locate failed",
  /* 15 */ "bt_data: couldn't split a block--record too large",
  /* 16 */ "bt: out of memory",
  /* 17 */ "bt_fix_index: trying to split root index block!",
  /* 18 */ "bt_fix_index: couldn't split a block--key too large",
  /* 19 */ "bt_fix_index: lost oldblock--critical error",
  /* 20 */ "bt_delete_record: couldn't find target record",
  /* 21 */ "bt_replacekey: couldn't find target key",
  /* 22 */ "bt_find_sister: couldn't find target key"
};

static int bt_blwrite(Btree *bt, int f, BufferList *bl)
{
    int ret = TREE_OK;

    if (fseek(FDATA(f,file), bl->offset, SEEK_SET))
        ret = TREE_FAIL;
    if (!ret &&
        fwrite(bl->buffer,
        SIZES(f,block_size), 1, FDATA(f,file)) != 1)
          ret = TREE_FAIL;

    if (ret) {
        bt -> error_code = 10;
        SHOW(printf("failing in " __FILE__
                    " at line %d\n", __LINE__);)
   }
   else
       bl->state = sClean;

   return ret;
}

static int bt_blread(Btree *bt, int f, BufferList *bl)
{
    int ret = TREE_OK;

    SHOW(printf("bt_blread: bt = %Fp, f = %d, bl = %Fp\n",
                        bt, f, bl);)
```

```
    SHOW(printf("filename: %s, file: %Fp\n",
                        FDATA(f, filename), FDATA(f, file));)
    SHOW(printf("buffer state=%d, offset=%lx,"
                "buffer=%Fp, next=%Fp\n",
                        bl->state, bl->offset,
                        bl->buffer, bl->next);)
    SHOW(printf("seeking to %ld\n", bl-> offset);)
    SHOW(fflush(stdout);)
    if (fseek(FDATA(f,file), bl->offset, SEEK_SET)) {
        ret = TREE_FAIL;
        SHOW(printf("failing in " __FILE__
                    " at line %d\n", __LINE__);)
    }
    SHOW (else printf(" now at %ld\n", ftell(FDATA(f, file)));)


    if (!ret &&
        fread(bl->buffer,
        SIZES(f,block_size), 1, FDATA(f,file)) != 1) {
          ret = TREE_FAIL;
        SHOW(printf("failing in " __FILE__
                    " at line %d\n", __LINE__);)
    }

    if (ret) {
        bl -> state = sFree;
        bt -> error_code = 10;
        SHOW(printf("failing in " __FILE__
                    " at line %d\n", __LINE__);)
    }
    else
        bl->state = sClean;

    return ret;
}

/* Load a disk block into a free buffer */
static BufferList *bt_getbuf(Btree *bt, int f, DISKOFFSET dof)
{
    BufferList *bl, *free_b, *free_p, *parent, *gparent;

    /* Already loaded? */
    free_b = free_p = parent = gparent = NULL;
    for (bl = FDATA(f,bufferlist); bl; ) {
        if (bl->state == sFree) {
            free_p = parent;
            free_b = bl;
        }
        else if (bl->offset == dof) {
            if (parent) { /* move to top of list */
                parent->next = bl -> next;
                bl -> next = FDATA(f,bufferlist);
                FDATA(f, bufferlist) = bl;
            }
            return bl;
        }
```

```
        gparent = parent;
        parent = bl;
        bl=bl->next;
    }

    if (!free_b) { /* Must free up an active buffer */
        /*
         * parent points to last buffer in list, and
         * gparent is the parent of this buffer. Because we
         * always move the most recently used buffer to the head
         * of the list, the last buffer is the least-recently
         * used buffer and gets tossed out.
         */
        free_b = parent;
        parent = gparent;
        if (free_b->state == sDirty && bt_blwrite(bt, f, free_b))
           free_b = NULL;
    }
    else
        parent = free_p;

    if (free_b) { /* A free buffer */
        SHOW(printf("bt_getbuf: Assigning bl at %Fp"
                    "to offset %ld\n", free_b, dof);)
        free_b -> offset = dof;
        if (bt_blread(bt, f, free_b)) {
            free_b -> state = sFree;
            free_b = NULL;
        }
        else {
            free_b -> state = sClean;
        }
    }

    if (free_b && parent) { /* Move new buffer to head of list */
        parent -> next = free_b -> next;
        free_b -> next = FDATA(f,bufferlist);
        FDATA(f,bufferlist) = free_b;
    }

    return free_b;
}

/* Get a block for read only */
char *bt_getblock4read(Btree *bt, int f, DISKOFFSET dof)
{
    BufferList *bl;
    SHOW(printf("bt_getblock4read(%Fp, %d, %ld)\n",bt,f,dof);)
    bl = bt_getbuf(bt, f, dof);
    if (bl)
        return bl -> buffer;
    else
        return NULL;
}

/* Get a block for read & write */
char *bt_getblock4write(Btree *bt, int f, DISKOFFSET dof)
```

```
{
    BufferList *bl;

    SHOW(printf("bt_getblock4write(%Fp, %d, %ld)\n",bt,f,dof);)
    bl = bt_getbuf(bt, f, dof);
    if (bl) {
        bl -> state = sDirty;
        return bl -> buffer;
    }
    else
        return NULL;
}

/* Get a new block for read & write */
BufferList *bt_getnew4write(Btree *bt, int f)
{
BufferList *bl;
BlockControl *bc;
DISKOFFSET dof;

/* first, are then any empty blocks already available? */
if ((dof = FDATA(f, free_block)) != 0) {
    SHOW(printf("bt_getnew4write(%Fp, %d, %ld)\n",bt,f,dof);)
    bl = bt_getbuf(bt, f, dof);
    if (!bl)
        return bl;
    bc = (BlockControl *) bl->buffer;
    FDATA(f, free_block) = bc->next;
    FDATA(f, modified) = 1;
}
else {
    /* extend the file */
    if (fseek(FDATA(f,file), 0L, SEEK_END))
        return NULL;
    dof = ftell(FDATA(f, file));

    /* just write something */
    if (fwrite(FDATA(f,bufferlist),
                    SIZES(f,block_size),
                    1,
                    FDATA(f, file)) != 1) {
        bt -> error_code = 10;
        SHOW(printf("failing in " __FILE__
                    " at line %d\n", __LINE__);)
        return NULL;
    }
    SHOW(printf("bt_getnew4write #2(%Fp, %d, %ld)\n",
                    bt,f,dof);)
    bl = bt_getbuf(bt, f, dof);
    if (!bl)
        return bl;
}

/* initialize block */
bc = (BlockControl *) bl->buffer;
bc -> bfree = sizeof(BlockControl);
bc -> blockstate = sAvail;
```

```
    bc -> parent = 0;
    bc -> next = 0;

    /* and ensure that all is written out */
    bt_flush(bt);

    bl -> state = sDirty;
        return bl;
    }

    int bt_flush(Btree *bt) {
        int i, ret;

        ret = TREE_OK;
        for (i=0; i<2; i++) {
            if (FDATA(i,file) && FDATA(i,bufferlist)) {
                char *p;
                DISKOFFSET *d;
                BufferList *bl;

                /* root block data changed? */
                if (FDATA(i, modified)) {

                    /* get 1st block */
                    p = bt_getblock4write(bt, i, 0L);
                    p += sizeof(SIGNATURE) + sizeof(SIG_INDEX) - 1;
                    p += sizeof(Sizes);
                    d = (DISKOFFSET *) p;
                    *d = FDATA(i, root_block);
                    d++;
                    *d = FDATA(i, free_block);
                    FDATA(i, modified) = 0;
                }

                for (bl = FDATA(i,bufferlist); bl; bl = bl->next) {
                    if (bl->state == sDirty)
                        ret = bt_blwrite(bt, i, bl);
                }

                fflush(FDATA(i, file));
            }
        }

        return ret;
    }

    int bt_bufinit(Btree *bt, int f)
    {
        int i, cnt;
        BufferList dummy, *bl;

        if (FDATA(f,bufferlist) == NULL) {
            bl = &dummy;
            if (f == DAT)
                cnt = 5;
            else
                cnt = (int) SIZES(f,levels) * 3 / 2;
            if (cnt < 3)
                cnt = 3; /* minimum number */
```

```
        for (i = 0; i < cnt; i++) {
            bl -> next = malloc(sizeof(BufferList));
            if (bl -> next == NULL) {
                bt -> error_code = 5;
                return TREE_FAIL;
            }
            bl = bl -> next;
            bl -> buffer =
                malloc(SIZES(f,block_size));
            if (bl -> buffer == NULL) {
                bt -> error_code = 5;
                return TREE_FAIL;
            }
            bl -> state = sFree;
            bl -> offset = 0;
            bl -> next = NULL;
        }

        FDATA(f,bufferlist) = dummy.next;
    }
    return TREE_OK;
}

/* Release memory allocated to buffers */
void bt_bufrelease(Btree *bt, int f)
{
    BufferList *bl, *bl2;

    bl = FDATA(f,bufferlist);
    if (bl) {
        while(bl) {
            free(bl -> buffer);
            bl2 = bl -> next;
            free(bl);
            bl = bl2;
        }
        FDATA(f,bufferlist) = NULL;
    }
}

/* put a block on the free chain */
int  bt_releaseblock(Btree *bt, int f, DISKOFFSET dof)
{
    char *pblock;
    BlockControl *bc;

    pblock = bt_getblock4write(bt, f, dof);
    if (!pblock)
        return TREE_FAIL;
    bc = (BlockControl *) pblock;

    bc -> next = FDATA(f, free_block);
    bc -> blockstate = sAvail;
    FDATA(f, free_block) = dof;
    FDATA(f, modified) = 1;

    return TREE_OK;
}
```

7. 使用模块：main( )

我们对执行上述所有工作的代码充分加了注释，可以通过直接检查代码并且运行示例应用程序来获取关于其性能的进一步的细节。如我们已经讨论过的，这个应用程序将把一对字符串加载进一个结构中，并且基于第一个字符串的值计算键。示例驱动程序可用于插入、删除和查找记录。可以随时转储树的内容及其内部结构。此外，还可以生成磁盘块使用的映像。

为了进一步简化测试，通常利用少量的 split 和 merge 编译驱动程序。这导致数据块和索引块迅速填充，从而允许利用最少的数据详细检查树的性能。

程序清单 6-8　用于 user. c 的示例数据的定义

```
/*--- user.h ------------------------------ Listing 6-8 ---------
 * Sample data record definition
 *-------------------------------------------------------------*/

/* example data records */
typedef struct UserRecord {
    char key[11];
    char name[25];
    char addr[25];
} UR;

/* a record with a maximal key. initialized in user.c */
extern UR MaxRec;


/* The routines */
unsigned UserGetKeySize ( void *r );
unsigned UserGetRecSize ( void *r );
int UserKey2KeyCmp ( void *k1, void *k2 );
int UserKey2RecCmp ( void *k, void *r );
void UserRec2KeyCpy ( void *k, void *r );
```

程序清单 6-9　用于 B 树代码的支持例程和测试驱动程序

```
/*--- user.c ------------------------------ Listing 6-9 ---------
 *Sample support routines and a driver
 *-------------------------------------------------------------*/

#include "bt_hdr.h"
#include <ctype.h>
#include "user.h"

#define TESTSIZES /* use small block limits to force splits */

/*
 * A record with a maximal key
 */
UR MaxRec = { "\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff",
              "Maximal", "record" };

/*
 * The support routines that the btree routines call
 */
```

```
unsigned UserGetKeySize(void *k) {
    return strlen( (char *) k ) + 1;
}

unsigned UserGetKeyNRecSize(void *k) {
    return strlen( ((UR *) k) -> key ) + 1;
}

unsigned UserGetRecSize(void *r) {
    return sizeof(UR);
}

int UserKey2KeyCmp(void *k1, void *k2) {
    return strcmp((char *) k1, (char *) k2);
}

int UserKey2RecCmp(void *k, void *r) {
    return strcmp((char *) k, ((UR *) r)->key);
}
void UserRec2KeyCpy(void *rec, void *key) {
      strcpy((char *) key, ((UR *) rec) -> key);
}

/*
 * Some tree display routines
 */
char *BlockStates[] = { "sAvail", "sNode", "sLeaf" };
BLKSIZE index_blksize;
FILE *outfile;

void show_basic(Btree *bt, int f) {
    DISKOFFSET length, blkcount;
    BLKSIZE blksize;

    fprintf(outfile, "File: %s\n", FDATA(f,filename));

    blksize = SIZES(f, block_size);
    fprintf(outfile, "  Block size: %d, split: %d, merge: %d\n",
           blksize,
           SIZES(f, split),
           SIZES(f, merge));

    if (f == NDX)
        fprintf(outfile, "  Index has %d levels\n",
                SIZES(f, levels));

    fseek(FDATA(f, file), 0L, SEEK_END);
    length = ftell(FDATA(f,file));
    blkcount = length/blksize;
    fprintf(outfile, "  File length %ld (%ld blocks)\n",
                length, blkcount);

    if (f == NDX)
        fprintf(outfile, "  Root index block at %ld"
                         " (block no. %ld)\n",
               FDATA(f, root_block),
               FDATA(f, root_block)/blksize);
```

```
    fprintf(outfile,
            "  First free block at %ld (block no. %ld)\n",
           FDATA(f, free_block),
           FDATA(f, free_block)/blksize);
}

/*
 * Given index block is at given level. Display contents
 * and then recursively examine the children.
 */
void prefix(int level) {
    for (; level; level--) fprintf(outfile, " * ");
}

void walk_down(Btree *bt, int f, DISKOFFSET block,
               int level, int mode) {
    char *s, *key;
    BlockControl *bc;
    unsigned offset, keysize, keycount = 0;
    DISKOFFSET *pDof;

    prefix(level);
    if (f == NDX)
        fprintf(outfile,
                "Level %d index block %ld at offset %ld.",
                level, block/index_blksize, block);
    else
        fprintf(outfile, "Data block %ld at offset %ld.",
                         block/SIZES(DAT, block_size), block);

    if (mode)
        fprintf(outfile, "\n");

    s = bt_getblock4read(bt, f, block);
    if (!s) {
        bt->error_code = 7;
        return;
    }

    bc = (BlockControl *) s;

    if (mode) {
        prefix(level);
        fprintf(outfile,
           "blockstate: %s, free data: "
           "%d, parent: %ld, next: %ld\n",
           BlockStates[bc->blockstate],
           bc->bfree,
           bc->parent,
           bc->next);
    }

    if (f == NDX) {
        offset = sizeof(BlockControl);
        while (offset < bc->bfree) {
            keysize = GETKEYSIZE(s + offset);
            key = s + offset;
            pDof = (DISKOFFSET *) (key + keysize);
```

```
            if (mode) {
                prefix(level);
                fprintf(outfile,
                        "Offset %d: key '%s', DISKOFFSET %ld\n",
                        offset, key, *pDof);
            }
            offset += keysize + sizeof(DISKOFFSET);
            keycount++;
        }
        if (!mode)
            fprintf(outfile, " %u entries\n", keycount);

        offset = sizeof(BlockControl);
        while (offset < bc->bfree) {
            keysize = GETKEYSIZE(s + offset);
            walk_down(bt,
                      bc->blockstate == sLeaf ? DAT : NDX,
                      *((DISKOFFSET *) (s + offset + keysize)),
                      level + 1, mode);

            /* recursive calls may invalidate our buffer */
            s = bt_getblock4read(bt, f, block);
            bc = (BlockControl *) s;
            offset += keysize + sizeof(DISKOFFSET);
        }
    }
    else { /* f == DAT */
        offset = sizeof(BlockControl);
        while (offset < bc->bfree) {
            keysize = GETRECSIZE(s + offset);
            if (mode) {
                prefix(level);
                fprintf(outfile,
                    "Offset %d: key '%s', name: %s, addr: %s\n",
                    offset,
                       ((UR *) (s + offset))->key,
                       ((UR *) (s + offset))->name,
                       ((UR *) (s + offset))->addr);
            }
            offset += keysize;
            keycount++;
        }
        if (!mode)
            fprintf(outfile, " %u entries\n", keycount);
    }
}

/*
 *display contents of data files
 *
 * mode = 0: just display number of pointers at each level
 * mode = 1: display all data
 */
void show_btree(Btree *bt, FILE *f, int mode) {
    outfile = f;
    if (mode) {
        show_basic(bt, NDX);
```

```
        show_basic(bt, DAT);
    }

    index_blksize = SIZES(NDX, block_size);
    walk_down(bt, NDX, FDATA(NDX, root_block), 0, mode);
}

/*
 * Block mapping routines
 *
 * Map is built up in map[] with these codes:
 *  C = control block (always block 0)
 *  N = index block
 *  D = data block
 *  . = lost block
 */
struct MapData {
    long length;
    unsigned blkcount;
    char *map;
} mapdata[2];

void show_btree_mapr(Btree *bt, DISKOFFSET iblock)
{
    char *pblock;
    BlockControl *bc;
    unsigned offset;
    DISKOFFSET dof;

    /* say we've been here */
    mapdata[NDX].map[iblock/SIZES(NDX,block_size)] = 'N';

    /* get the block */
    pblock = bt_getblock4read(bt, NDX, iblock);
    if (!pblock)
        return;
    bc = (BlockControl *)pblock;

    /* scan it */
    offset = sizeof(BlockControl);
    while (offset < bc->bfree) {
        offset += GETKEYSIZE(pblock + offset);
        dof = *((DISKOFFSET *) (pblock + offset));
        offset += sizeof(DISKOFFSET);
        if (bc->blockstate == sNode) {
            show_btree_mapr(bt, dof);
            pblock = bt_getblock4read(bt, NDX, iblock);
            if (!pblock)
                return;
            bc = (BlockControl *)pblock;
        }
        else
            mapdata[DAT].map[dof/SIZES(DAT,block_size)] = 'D';
    }
}

void show_btree_map(Btree *bt, FILE *file) {
```

```
    int i, j;

    /* setup maps */
    for (i=0; i < 2; i++) {
        fseek(FDATA(i, file), 0L, SEEK_END);
        mapdata[i].length = ftell(FDATA(i, file));
        mapdata[i].blkcount =
                    mapdata[i].length/SIZES(i, block_size);
        mapdata[i].map = malloc(mapdata[i].blkcount);
        for (j=0; j < mapdata[i].blkcount; j++)
            mapdata[i].map[j] = '.';
        mapdata[i].map[0] = 'C';
    }

    /* walk the tree */
    show_btree_mapr(bt, FDATA(NDX, root_block));

    /* trace the free blocks */
    for (i=0; i < 2; i++) {
        DISKOFFSET block;
        char *pblock;
        BlockControl *bc;

        block = FDATA(i, free_block);
        while(block) {
            pblock = bt_getblock4read(bt, i, block);
            if (!pblock)
                break;
            mapdata[i].map[block/SIZES(i, block_size)] = 'f';
            bc = (BlockControl *) pblock;
            block = bc -> next;
        }
    }
    /* display the map */
    for (i=0; i < 2; i++) {
        fprintf(file, "%s Block Map:\n      ",
                    i == 0 ? "Index" : "Data");
        for (j=0; j < 10; j++)
            fprintf(file, " %d", j);

        for (j=0; j < mapdata[i].blkcount; j++) {
            if (j % 10 == 0) {
                fprintf(file,"\n%4d: ", j);
            }
            fprintf(file, " %c", mapdata[i].map[j]);
        }
        fprintf(file, "\n");
        free(mapdata[i].map);
    }

}

/*
 * Sample display function
 */
int DataCount = 0;
int DisplayFunc(Btree *bt, void *rec) {
```

```
    UR *ur;
    ur = (UR *) rec;
    fprintf(outfile, "k:'%-10s', n:'%-24s', a:'%-24s'\n",
                     ur->key, ur->name, ur->addr);

    DataCount++;
    return TREE_OK;
}

/*
 * Make a new dataset
 */
int make_dataset(char *name)
{
    Btree bt;
    int retval;

    /*
     * Fill in parts of the structure. For the sizes, we
     * give what we want. The only real restriction is that
     * block_size - split must leave enough room to insert
     * the largest possible record or key.
     */
    strcpy(bt.fdata[NDX].filename, name);
    strcat(bt.fdata[NDX].filename, ".ndx");
    bt.fdata[NDX].modified = 0;
    bt.fdata[NDX].bufferlist = NULL;
    bt.fdata[NDX].sizes.block_size = 2048;
    #if defined(TESTSIZES)
    bt.fdata[NDX].sizes.split      = 80;
    bt.fdata[NDX].sizes.merge      = 45;
    #else
    bt.fdata[NDX].sizes.split      = 2000; /* room for one key */
    bt.fdata[NDX].sizes.merge      = 1024;
    #endif
    bt.fdata[NDX].sizes.levels     = 4;

    strcpy(bt.fdata[DAT].filename, name);
    strcat(bt.fdata[DAT].filename, ".dat");
    bt.fdata[DAT].modified = 0;
    bt.fdata[DAT].bufferlist = NULL;
    bt.fdata[DAT].sizes.block_size  = 4096;
    #if defined(TESTSIZES)
    bt.fdata[DAT].sizes.split       = 500;
    bt.fdata[DAT].sizes.merge       = 300;
    #else
    bt.fdata[DAT].sizes.split       = 3950; /* room for a rec */
    bt.fdata[DAT].sizes.merge       = 2048;
    #endif
    bt.fdata[DAT].sizes.levels      = 0; /* any value is OK */

    bt.getkeysize     = UserGetKeySize; /* user.c */
    bt.getkeyNrecsize = UserGetKeyNRecSize;
    bt.getrecsize     = UserGetRecSize;
    bt.key2keycmp     = UserKey2KeyCmp;
    bt.key2reccmp     = UserKey2RecCmp;
    bt.rec2keycpy     = UserRec2KeyCpy;
```

```
    bt.error_code   = 0;
    bt.duplicatesOK = 1;

    retval = bt_new(&bt, MaxRec.key, &MaxRec);
    if(retval == TREE_OK)
        printf("Data files created.\n");
    else
        printf("Create failed: %s.\n",
                ErrorText[bt.error_code]);

    return retval;
}

/*
 * Open a dataset
 */
Btree *open_dataset(char *name)
{
       Btree *bt;

       bt = malloc(sizeof(Btree));
       if (!bt)
              return NULL;

       strcpy(bt->fdata[NDX].filename, name);
       strcat(bt->fdata[NDX].filename, ".ndx");
       bt->fdata[NDX].bufferlist = NULL;

       strcpy(bt->fdata[DAT].filename, name);
       strcat(bt->fdata[DAT].filename, ".dat");
       bt->fdata[DAT].bufferlist = NULL;

    bt->getkeysize     = UserGetKeySize; /* user.c */
    bt->getkeyNrecsize = UserGetKeyNRecSize;
    bt->getrecsize     = UserGetRecSize;
    bt->key2keycmp     = UserKey2KeyCmp;
    bt->key2reccmp     = UserKey2RecCmp;
    bt->rec2keycpy     = UserRec2KeyCpy;

       bt->error_code = 0;
       bt->duplicatesOK = 1;

       if (bt_open(bt)) {
        printf("Open failed: %s.\n",
                              ErrorText[bt->error_code]);
              free(bt);
              bt = NULL;
       }
    else
        fprintf(stdout, "Opened\n");
       return bt;
}

/*
 * A sample driver
 */
```

```
int DoData(Btree *bt, char *buffer, int mode) {
    UR ur;
    char *s;
    s = strchr(buffer, ';');
    if (!s) {
        if (mode == 0)
            s = ""; /* don't need address for deletes */
        else
            return TREE_FAIL; /* trying to add partial data */
    }
    else
        *s++ = '\0';

    strncpy(ur.key, buffer, 10); /* key is first 10 characters */
    ur.key[10] = '\0';
    strncpy(ur.name, buffer, 24);
    ur.name[25] = '\0';
    strncpy(ur.addr, s, 24);
    ur.addr[25] = '\0';

    if (mode == 0)
        return bt_delete(bt, ur.key);
    else
        return bt_add(bt, &ur, ur.key);
}

#define BUFLEN 100
void LoadFile(Btree *bt, char *fname)
{
    FILE *infile;
    char buffer[BUFLEN], *s;
    int i = 0, j = 0, retval = TREE_OK;

    if ((infile = fopen(fname, "r")) == NULL) {

        fputs(" Couldn't open the file.\n", stdout);
        return;
    }

    while (fgets(buffer, BUFLEN, infile)) {
        s = buffer + strlen(buffer);
        while(iscntrl(*s))
            *s-- = 0;

        printf("Loading %s\n", buffer);
        if (buffer[0] == ';') /* a comment */
            ;
        else if (buffer[0] == '-' && buffer[1] != 0) {
            retval = DoData(bt, buffer+1, 0);
            if (retval) {
                printf("  --delete failed\n");
                retval = 0;
            }
            else
                j++;
        }
        else {
```

```
            retval = DoData(bt, buffer, 1);
            if (retval)
                break;
            i++;
        }
    }

    if (retval)
        printf("Failed at line %s\n", buffer);

    fclose(infile);
    printf("Loaded %d items and deleted %d from %s.\n",
        i, j, fname);
}


main(int argc, char **argv)
{
    char inbuf[BUFLEN], *s;
    Btree *bt = NULL;
    FILE *logfile = NULL;

    for (;;) {
        fflush(stdout);
        fputs("Action (? for help): ", stdout);
        fflush(stdout);
        fgets(inbuf, BUFLEN, stdin);
        s = inbuf + strlen(inbuf);
        while(iscntrl(*s))
            *s-- = 0;

        if (logfile)
            fprintf(logfile, "%s\n", inbuf);

        if (!bt && strchr("@adfkKmMsSwW", inbuf[0])) {
            fputs(" **no open dataset\n", stdout);
            continue;
        }

        switch (inbuf[0]) {
            case '?':
                fputs(
        "@file      - load strings in file into tree\n"
        "a string   - add name;addr to tree\n"
        "d string   - delete name;addr from tree\n"
        "dup [0|1]  - disallow/allow duplicates\n"
        "f string   - find name;addr in tree\n"
        "k/K [file] - display key counts (K = overwrite file)\n"
        "l file     - log actions to file\n"
        "l          - turn off action logging\n"
        "m/M [file] - display block usage map\n"
        "n file     - make a new dataset\n"
        "o file     - open an existing dataset\n"
        "s/S [file] - display tree (S = overwrite file)\n"
        "w/W [file] - walk tree, (W = overwrite file)\n"
        "q          - quit\n"
```

```
, stdout);
        fflush(stdout);
        break;

    case '@':
        LoadFile(bt, inbuf + 1);
        break;

    case 'a':
        if (inbuf[1] != ' ' ||
            !inbuf[2]       ||
            !strchr(inbuf,';'))
            fputs(" Not a valid command\n", stdout);
        else
            if (DoData(bt, inbuf + 2, 1) == TREE_FAIL)
                fputs(" ** Insertion failed\n", stdout);
        break;

    case 'd':
        if (inbuf[1] == 'u' && inbuf[2] == 'p') {
            if (inbuf[3] == ' ' &&
                (inbuf[4] == '0' || inbuf[4] == '1'))
                bt -> duplicatesOK =
                inbuf[4] == '0' ? 0 : 1;
            fputs("duplicates are ", stdout);
            if (bt -> duplicatesOK == 0)
                fputs("not ", stdout);
            fputs("allowed.\n", stdout);
            break;
        }

        if (inbuf[1] != ' ' || inbuf[2] == 0)
            fputs(" Not a valid command\n", stdout);


        else {
            if (DoData(bt, inbuf + 2, 0) == TREE FAIL)
                    fputs(" ** Delete failed\n", stdout);
            }
            break;

        case 'f':
            if (inbuf[1] != ' ' || inbuf[2] == 0)
                fputs(" Not a valid command\n", stdout);
            else {
                UR record;
                inbuf[12] = '\0';
                if (bt_find(bt, &record, inbuf+2) ==
                                                TREE_FAIL)
                    fputs(" ** Find failed\n", stdout);
                else
                    fprintf(stdout, "found %s;%s\n",
                            record.name, record.addr);
            }
            break;

        case 'k': case 'K': {
                FILE *out;
```

```
            if (inbuf[1] == ' ' && inbuf[2] != 0) {
                out = fopen(inbuf+2,
                            inbuf[0] == 'k' ? "w" : "a");
                if (!out)
                    printf("Can't open %s\n", inbuf + 2);
                else {
                    show_btree(bt, out, 0);
                    fclose(out);
                }
            }
            else
                show_btree(bt, stdout, 0);
        }
        break;

    case 'l':
        if (inbuf[1] != ' ' || inbuf[2] == 0) {
            if (logfile) {
                fclose(logfile);
                logfile = NULL;
            }
            else
                fputs(" Logfile not open\n", stdout);
        }
        else {
            logfile = fopen(inbuf + 2, "w");
            if (logfile == NULL)
                printf("Can't open %s\n", inbuf + 2);
        }
        break;

    case 'm': case 'M': {
            FILE *out;

            if (inbuf[1] == ' ' && inbuf[2] != 0) {
                out = fopen(inbuf+2, inbuf[0] ==
                                        'm' ? "w" : "a");
                if (!out)
                    printf("Can't open %s\n", inbuf + 2);
                else {
                    show_btree_map(bt, out);
                    fclose(out);
                }
            }
            else
                show_btree_map(bt, stdout);
        }
        break;

    case 'n':
        if (inbuf[1] != ' ' || inbuf[2] == 0)
            fputs(" Not a valid command\n", stdout);
        else
            make_dataset(inbuf+2);
        break;
```

```
case 'o':
                if (inbuf[1] != ' ' || inbuf[2] == 0)
        fputs(" Not a valid command\n", stdout);
    else {
        if (bt) {
            if (bt_close(bt))
                printf("\nClose failed: %s\n",
                        ErrorText[bt->error_code]);
            else
                printf("\nData files closed.\n");

            bt = NULL;
        }
                    bt = open_dataset(inbuf+2);
    }
    break;

case 'q':
    if (logfile)
        fclose(logfile);

    if (bt) {
        if (bt_close(bt))
            printf("\nClose failed: %s\n",
                    ErrorText[bt->error_code]);
         else
             printf("\nData files closed.\n");
     }
     return;

 case 's': case 'S': {
         FILE *out;

         if (inbuf[1] == ' ' && inbuf[2] != 0) {
             out = fopen(inbuf+2, inbuf[0] ==
                                 's' ? "w" : "a");

             if (!out)
                 printf("Can't open %s\n", inbuf + 2);
             else {
                 show_btree(bt, out, 1);
                 fclose(out);
             }
         }
         else
             show_btree(bt, stdout, 1);
     }
     break;

 case 'w': case 'W': {
         if (inbuf[1] == ' ' && inbuf[2] != 0) {
             outfile = fopen(inbuf+2, inbuf[0] ==
                                 'w' ? "w" : "a");
             if (!outfile)
                 printf("Can't open %s\n", inbuf + 2);
             else {
                 DataCount = 0;
```

```
                                bt_walk(bt, DisplayFunc);
                                fprintf(outfile, "%d items\n",
                                        DataCount);
                                fclose(outfile);
                                outfile = NULL;
                            }
                        }
                        else {
                            DataCount = 0;
                            outfile = stdout;
                        bt_walk(bt, DisplayFunc);
                        fprintf(outfile, "%d items\n",
                                DataCount);
                        outfile = NULL;
                    }
                }
                break;

            case ';':
                break;  /* comment */

            default:
                fputs(" Not a valid command\n", stdout);
                break;
        }
        if (bt && bt->error_code) {
            printf("ERROR: %s\n", ErrorText[bt->error_code]);
            bt->error_code = 0;
        }
    }
}
```

8. 确定树的高度

树的容量是由它的高度及其块大小确定的。查看表 6-2 中所示的示例。假设我们决定使用 2048 字节的索引块，并且估计平均键大小是 10 字节。如果 diskoffset 是 4 字节，那么键加上 diskoffset 就是 14 字节。其中 BlockControl 结构将使用 12 字节，这为每个索引块留下了（2048 - 12）/ 14 = 145 个键的空间。我们进一步决定使用 4096 字节的数据块，并且估计我们的平均记录大小是 100 字节。这样，每个数据块将可以存储（4096 - 12）/100 = 40 个记录。这样，树的容量将是 $145^{height-1} \times 40$。

表 6-2　通过树的高度和块大小确定树容量的示例

高度	最大记录数	高度	最大记录数
1	5 800	4	17 682 025 000
2	841 000	3	121 945 000

虽然实际的容量要稍微小一些，因为我们从不允许块完全填满，但是高度为 4 的树看起来就像它将存储许多数据一样。另请注意：创建固定高度为 4 的树具有极少的性能影响。由于块缓冲系统，对于较小的树，相关的上层索引块很可能在内存中总是可用。因此，访问它们不需要进行

磁盘访问，并且引入的开销很小。由于这些原因，所以选择固定高度的模式。

9. 额外的修改

程序清单6-3～6-9中的B树代码确实只是一个开头，商业B树程序包提供了比这里实现的多得多的特性。不过，这个模块当然可以起作用，它执行得很好，并且良好地演示了各种必须解决的问题。可以添加的特性如下：

- 可以允许树的高度增加或缩短。这涉及在bt_fix_index()找到必须分割的根块时添加新的根块。相反，将需要把用于删除根块的代码添加到删除例程中。不过，在现实中，这将对性能具有极少的影响。
- 具有next()和previous()函数将是有用的，给定一个记录，它们将分别返回下一个和前一个记录。示例代码没有直接提供这种能力。相反，树遍历例程将调用用户定义的函数，接连操作每个记录。不过，可以使用bt_find_sister()中的代码轻松地构建这些函数。
- 可以提供改进的灾难保护措施。例如，如果用户在树更新期间终止会话，数据基础很可能损坏。可以通过保存文件的“前像”，最好地处理这种问题。每次修改块时，都将“前”副本写到“前像”文件中。然后，在用户可定义的时间间隔内，将所有文件冲洗到磁盘上，最后一步是清理“前像”文件。这种方法允许从许多类型的灾难中恢复。
- 可以创建额外的维护工具——尤其是用于对.dat文件重建索引的工具和用于修复损坏的索引的工具。

6.5 可以看见森林吗

本章介绍了许多基础知识。从简单的二叉树开始，我们研究了高度平衡的树（红黑树）和使用平衡的树（伸展树）。其中每种树都可以轻松地用于内存中的树。然后，为了解决更大数据集的问题，我们创建了一个磁盘上的B树模块。不过，所有这些树的共同之处是键和记录稳定递增的思想。虽然用于树遍历的算法在细节上有所不同，但是它们在概念上是相似的：从顶部开始，下降到左边，转到中间，然后转到右边。甚至插入和删除从头至尾也具有类似的感觉。

这些是本书中的一些最复杂的例程，只能通过运行实际的代码来查看它们都是如何工作的，而没有其他替代方法。每个例程都具有一个广泛的测试驱动程序用于简化这个过程，每种树都能够以图形形式打印它自身，以便于研究。

6.6 资源和参考资料

Adel'son-Vel'skii、G. M. 和E. M. Landis. “An Algorithm for the Organization of Information.” *Soviet Math*, Vol. 3, pp. 1259-1263, 1962. 这是原始的AVL论文。Knuth和Melhom所做的讨论更容易理解。

Bayer, R. 和E. McCreight. “Organization and Maintenance of Large Ordered Indexes.” *Acta Informatica*, Vol. 1, pp. 173-189, 1972. This is the basic B-tree paper.

Comer, Douglas. “The Ubiquitous B-tree.” *Computing Surveys*, Vol. II, pp. 121-137, 1979. 这篇文章包含一节关于B树历史的有趣内容，包括有关术语“B树”的来龙去脉的讨论。

Knuth, Donald E. *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Reading, MA:

Addison-Wesley, 1973. 这是本章的两个关键的一般性资源之一，并且具有百科全书式的格式。特别要查看 Section 6.22，在标题“The Performance of Binary Trees”下面查阅本章中描述的材料。

Melhorn, K. *Data Structures and Algorithms* 1: *Sorting and Searching*. Berlin: Springer-Verlag, 1984. 本书是用于本章的另一个关键的一般性资源。它讨论了 Knuth 忽略了的许多有趣的次级主题。特别是，Melhorn 关于伸展树的讨论极其出色，尽管感兴趣的读者实际上应该参考 Sleator 和 Tarjan 所著的原始文章。

Sleator, D. D. 和 R. E. Tarjan. “Self-Adjusting Binary Search Trees.” *Journal ACM*, Vol. 32, pp. 652-686, 1985. 这是关于伸展树的原始论文。在 bintree.c 中演示的自顶向下的“伸展”操作就取自于这篇文章。

Tarjan、Robert E. 和 Christopher J. Van Wyk. “An O（n log log n）-Time Algorithm for Triangulating a Simple Polygon.” *Siam J. Comput*, Vol. 17, pp. 143-178, 1988. 这篇文章包含关于红黑树的有帮助的讨论。

第7章　日期和时间

日期和时间例程提出的挑战并不是传统算法中的典型情况。相反，这些挑战重点关注的是处理人类历史的变幻莫测，并把它们反映在历法中。为了理解日期和时间例程如何工作，需要理解历法是如何演进为它们目前的状态的。

儒略历最初是由儒略·恺撒（从而得此历法名）于公元前46年实现的，它把一年的平均长度定义为365 $\frac{1}{4}$天。额外的一天（称为**闰日**（leap day））每4年出现一次，通过确定每4年中的第4年（**闰年**（leap year））将包含366天来识别它，而其余3年（**平年**（common year））将包含365天。这种安排导致了不精确。通过把一年设置为365 $\frac{1}{4}$天，就把日历设置为地球围绕太阳旋转一圈的持续时间，即365天加6个小时。事实上，旋转一圈的时间为365天5小时49分12秒。定义的历年是10分48秒（或10.8分钟），这太长了。经过400年后，这种差别将导致历法自身提前3天。

随着时间的流逝，问题变得更糟糕，到16世纪时（在制定儒略历之后大约1600年），历法就提前了12天。这对于罗马天主教教堂具有严重的影响，它负责规定复活节的时间。教会发现复活节失去了冬末的感觉，神圣的一年中所有重大的神圣节日（圣诞节除外）都被定义为与复活节相关。

1582年，圣格列高利十三世教皇决定去掉儒略历增加的额外天数，并且重新定义哪些年是闰年，以便它们更好地反映天文学的现实。他从10月份中去掉了10天：1582年10月4日后面接着1582年10月15日。此外，他还命令：从那时起，除了那些可以被100整除但不能被400整除的年份之外，其他所有可以被4整除的年份都将是闰年。例如，在1599年到1999年这段时间里，除了1700年、1800年和1900年之外，其他所有可以被4整除的年份都是闰年。这种更改的效果是：在每400年里，有3个百年不是闰年。这种措施校正了儒略历中每400年要多出3天的错误。新历法按教皇的名字被命名为格列高利历（即阳历），今天几乎在全世界都使用它。

格列高利教皇宣告所做的修改只适用于罗马天主教教堂统治的领地。基督教国家（比如英格兰）原则上拒绝附和罗马教皇的任何法令，因此是在不同的时间转而接受格列高利历。虽然大多数基督教国家是在1699～1701年的时间期限内采纳新历法的，但是英格兰（乃至美洲殖民地）直到1752年才转变过来。到它们做出这种转变时，儒略历中另一个不合乎逻辑的日子在这两种历法之间开始显现出来。因此，1752年9月，英格兰从历法中去掉了11天：1752年9月2日后面紧接着1752年9月14日。在1752年之后，大多数主要的西方国家都已经开始施行格列高利历。不过，较小的国家由于各种政治原因而坚持了长得多的时间。罗马尼亚直到1919年才转变过来；土耳其则直到1927年才这样做。甚至直到今天，处于东正教统治之下的一些国家仍在使用格列高利历的一种变体来制定它们自己的日历系统。因此，在讨论历史日期时，仅仅知道事件发生于何时是不

够的，还要知道它发生于何处。

在本章介绍的例程中，我们使用 1582 年作为重大转变的日期。我们可以使用英国/美国的 1752 年，但是，在 1582 年到 1752 年这个时间段内的重要日期与欧洲大陆发生的事件（而不是在英格兰或美洲殖民地发生的事件）联系非常紧密。我们的选择是一种常见的选择，尽管 UNIX 选择美国日期 1752 年作为其转变的时间点。因此，当你使用 1752 年之前的任何日历时，都必须确定你是在使用儒略历，还是在使用格列高利历。

格列高利历还进行了另外一处改变：它把新年的起始日期设置为 1 月 1 日。以前，各个国家在不同的日子庆祝新年——有些国家早在 12 月 25 日就开始庆祝，另外一些国家则迟至 3 月 25 日。在 1752 年以前，英格兰和美洲殖民地使用 3 月 25 日作为新年的节日。也就是说，1701 年 3 月 24 日后面接着 1702 年 3 月 25 日。由于这种改变，乔治·华盛顿的生日（今天被确定为 1732 年 2 月 22 日）实际上被记录在 1731 年 2 月 11 日这一天。因此，如果你需要准确讨论 1752 年之前的任何日期，都必须知道事件记录于何处，以及该日期是否已转换为格列高利历。现代以前的日期是一件难以处理的事情。

大多数日历系统都使用本章中介绍的例程采用的约定：把 1582 年用作重大转变的时间点，并且当前日历约定在 1 月 1 日开始新的一年，可以依据这种约定向后推测前几年的时间。人们通常认为这种推测意味着一些日期偶尔会与在给定的日子记录它们的日期不一致。如果你需要知道为 1582 年的某个事件记录的准确日期，这些例程将不适合你，并且几乎所有的例程都不能做到这一点。今天的计算机上使用的大多数日历都寻求做一件事：给每一天提供一个唯一的标识序列码，它允许用户相对于当前时间段计算日期。除此之外，一切都是徒劳无益的。这些例程将告诉你在它们的模式中公元 848 年 6 月 1 日是否是星期六，但是并不保证答案将与现实一致（假如你生活在那一天的话）。像这样依据当前日历模式向后推测的日历被称为**预期的**（proleptic）日历。对于重大转变日期之前的日期，应该小心使用它们。

7.1 日期例程的库

如果人们不需要知道他们生活在哪一年，计算所有日子的一种简单方式是选择一天并称之为第 1 天，然后从那一天起连续地对每一天进行编号。这种模式最初是由一个修道士提出的，他按自己父亲的名字 Julius 把这些编号的日子命名为**儒略日**（Julian day）（不要把儒略日与儒略历弄混淆）。由于这位修道士相信历法的周期通常汇集于公元前 4713 年 1 月 1 日，所以他把这个日期用作他的起点（这一天与任何可辨别的事件以任何方式联系紧密的可能性极低）。尽管如此，还是把从那一天起向前编号的日子称为儒略日。根据惯例，一般把 1582 年 10 月 15 日这个重大的日子（格列高利历的第一天）接受为儒略日期 2 299 161。

在稍后将介绍的例程中，我们从公元 1 年 1 月 1 日（基督教纪元理论上的第一天）起统计日子（根据惯例，没有公元 0 年的说法）。由于我们的日期使用不同于儒略统计法的起点，所以不能把它们称为儒略日期；因此，在本章中，我们只是简单地把它们称为日期编号。

我们所有的例程都在形如 struct tm 的结构中保存日期（和时间），在 ANSI C 头文件 time. h 中定义了该结构，如图 7-1 所示。这个结构可用于存储年、月、日、时、分、秒以及许多其他的数据。

```
struct tm {
        int tm_sec;         /* seconds, 0-based */
        int tm_min;         /* minutes, 0-based */
        int tm_hour;        /* hours,   0-based */
        int tm_mday;        /* day of the month, 1-based */
        int tm_mon;         /* month, note: 0-based! */
        int tm_year;        /* year since 1900 */
        int tm_wday;        /* weekday: Sun = 0, Sat = 6 */
        int tm_yday;        /* day of the year, 0-based */
        int tm_isdst;       /* daylight savings time flag */
};
```

图 7-1 time. h 中的 ANSI tm 结构

这个结构具有几个不适宜的方面：月份基于 0，这意味着 1 月被违反直觉地编号为 0；此外，使用这个结构的大多数函数都期望年份在 0 ~ 99 之间。这些例程把 1900 添加到年份中。在我们的工作中，我们将尊重基于 0 的月份这种用法；不过，我们的年份将一直回溯到公元 1 年。因此，在我们的例程中，当计算日期时，总是必须使用完整的年份。否则，将把 94 解释为公元 94 年，而不是 1994 年。

本章中解释的例程将只使用 3 个日期字段和 3 个时间字段。其他字段将被忽略或者单独计算。

为了把日期移入 tm 结构中，可以使用函数 LoadDateStruct( )，如图 7-2 所示。该函数应该用作这个库中的所有日期例程的入口点。

```
/*-------------------------------------------------------------
 * Loads a date passed as three integers into a tm structure.
 * Returns 0 on success, -1 on error.
 *-------------------------------------------------------------*/
int LoadDateStruct ( struct tm *date, int yy, int mm, int dd )
{
    date->tm_year = yy;
    date->tm_mon  = mm - 1; /* for compatibility w/ ANSI C */
    date->tm_mday = dd;

    if ( IsDateValid ( date ) == NO )
        return ( -1 );
    else
        return ( 0 );
}
```

图 7-2 LoadDateStruct( ) 函数把日期加载进 tm 结构中

函数 LoadDateStruct( ) 期望传递给它一个指向 tm 结构的指针和三个要加载的整数：它们分别用于年、月、日。这些应该是实际的值（函数将使月份基于 0）。由于 LoadDateStruct( ) 一般在使用其他日期函数之前调用，它还将通过调用 IsDateValid( ) 来检查日期的有效性，IsDateValid( ) 函数如图 7-3 所示。

```
/*-------------------------------------------------------------
 * Returns YES if date is valid, otherwise returns NO.
 *------------------------------------------------------------*/
int IsDateValid ( struct tm *date )
{
    if ( date->tm_mon < 0 || date->tm_mon > 11 )   /* per ANSI */
        return ( NO );

    if ( date->tm_year < 1 )
        return ( NO );

    if ( IsYearLeap ( date->tm_year ) == NO )
        if ( date->tm_mday < 1 ||
             date->tm_mday > Days_in_month[ date->tm_mon + 1] )
            return ( NO );
        else
        if ( date->tm_mday < 1 ||
             date->tm_mday >
                       Days_in_month_leap[ date->tm_mon + 1] )
            return ( NO );

    return ( YES );
}
```

图 7-3 IsDateValid( ) 确定给定的日期是否有效

IsDateValid( ) 检查所有的参数都在范围内。例如，我们最早的日期是公元 1 年，因此所有的日期都必须具有大于 0 的年份。它还会通过查找表中的天数，来检查月份中的天数。这个表是由日期函数访问的多个数据表之一。因此，为日期例程在全局级将这些表声明为静态数据元素。这意味着所有的日期例程都可以访问这些表，但是其他代码则不能这样做。这些表如图 7-4所示。

IsDateValid( ) 访问整数 Days_in_month 和 Days_in_month_leap 的数组。除了第 2 个元素（February）中的值之外，这些数组完全相同，可以预测它的值分别为 28 和 29。如果必要，可以把这两个表合并为一个表，并且每次都可以计算 February 的条目的值。不过，这需要跟踪元素的值，或者在开始计算每个日期时频繁重置它。更改小常量表的值不是一个良好的实践，它会为一些不易察觉的错误大开方便之门。由于内存一般价格便宜并且非常充裕，使用多个表是一种更好的选择。此外，它还允许将表定义为 const，这可以充当另一种安全措施。

为了让 IsDateValid( ) 知道使用哪个表，它首先必须确定当前年份是否为闰年。为此，它将调用 IsYearLeap( )，如图 7-5 所示。它给该函数传递一个整数，其中包含要检查的年份，并且返回 YES 或 NO，这取决于年份是否为闰年。

```
static const int Days_in_month [13] = {
    0,
    31, 28, 31,      /* Jan, Feb, Mar */
    30, 31, 30,      /* Apr, May, Jun */
    31, 31, 30,      /* Jul, Aug, Sep */
    31, 30, 31       /* Oct, Nov, Dec */
};

/* same as above but for a leap year */

static const int Days_in_month_leap [13] = {
    0,
    31, 29, 31,      /* Jan, Feb, Mar */
    30, 31, 30,      /* Apr, May, Jun */
    31, 31, 30,      /* Jul, Aug, Sep */
    31, 30, 31       /* Oct, Nov, Dec */
};

static const char *Day_name[7] = {
    "Sunday",    "Monday",
    "Tuesday",   "Wednesday",
    "Thursday",  "Friday",
    "Saturday"
};

static const char *Day_name_abbrev[7] = {
    "Sun",   "Mon",
    "Tue",   "Wed",
    "Thu",   "Fri",
    "Sat"
};

static const char *Month_name[13] = {
    "",
    "January",   "February",  "March",
    "April",     "May",       "June",
    "July",      "August",    "September",
    "October",   "November",  "December"
};

static const char *Month_name_abbrev[13] = {
    "",
    "Jan",   "Feb",   "Mar",
    "Apr",   "May",   "Jun",
    "Jul",   "Aug",   "Sep",
    "Oct",   "Nov",   "Dec"
};
```

图 7-4 由日期例程访问的全局静态数据

```
/*-------------------------------------------------------------
 * Returns YES or NO depending on whether a year is leap or not
 *-------------------------------------------------------------*/
int IsYearLeap ( int year )
{
    if ( year % 4 != 0 )     /* if year not divisible by 4... */
        return ( NO );       /* it's not leap */

    if ( year < 1582 )       /* all years divisible by 4 were */
        return ( YES );      /* leap prior to 1582            */

    if ( year % 100 != 0 )   /* if year divisible by 4, */
        return ( YES );      /* but not by 100, it's leap */

    if ( year % 400 != 0 )   /* if year divisible by 100, */
        return ( NO );       /* but not by 400, it's not leap */
    else
        return ( YES );      /* if divisible by 400, it's leap */
}
```

图 7-5 IsYearLeap( ) 函数确定给定的年份是否为闰年

注意解释如何确定闰年的注释。依据以前的解释，在格列高利历中（也就是说，从 1582 年起），能够被 4 整除的所有年份都是闰年，除非它们能够被 100 整除但是不能被 400 整除。因此，1900 年就不是闰年（能够被 100 整除但是不能被 400 整除），而 2000 年就是闰年（能够同时被 100 和 400 整除）。在 1582 年以前，能够被 4 整除的所有年份都是闰年（至于最初应用这个规则的精确时间还有一些争议。可能公元 4 年和公元 8 年不是闰年，这只是由于还没有遵守这个规则。不过，由于没有权威的证据存在，可以方便地假定这两个年份都是闰年）。

知道如何计算日期还允许我们把日期转换为日编号。日编号开始于某个选定的日期，并给该日期赋予值 1。给此后的每一天赋予一个连续的整数；也就是说，下一天是第 2 天，其后接着第 3 天，依此类推。以前讨论的儒略日编号使用公元前 4713 年 1 月 1 日作为起始日期。这里展示的例程使用公元 1 年 1 月 1 日（即基督教纪元的第一天）作为起始日期。图 7-6 中的函数 DateToDayNumber( ) 接受一个指向 tm 日期结构的指针，并把日期转换为日编号。注意：返回的日编号是一个长整数。如果发现日期是无效的，该函数就会返回一个值为 -1L 的长整数。

```
/*-------------------------------------------------------------
 * Returns a long which contains the number of days since
 * Jan 1, AD 1; returns -1L on error. See text for observations
 * on the use of this function.
 *-------------------------------------------------------------*/
long DateToDayNumber ( struct tm * date )
{
    long day_num;
    int  mm, yy;
```

图 7-6 DateToDayNumber( ) 函数把日期转换为日编号

```
if ( IsDateValid ( date ) == NO )
    return ( -1L );

yy = date->tm_year;
mm = date->tm_mon + 1;

/* get all the regular days in preceding years */

day_num =  ( yy - 1 ) * 365L;

/* now get the leap years */

day_num += ( yy - 1 ) / 4L;

/*
 * now back out the century years that are not leap:
 * this would be all century years that are
 * not evenly divisible by 400: 1700, 1800, 1900, 2100...
 */

day_num -= ( yy - 1 ) / 100L;
day_num += ( yy - 1 ) / 400L;

/*
 * before 1582 all century years were leap, so adjust for
 * this. If year is > 1582, then just add 12 days for years
 * 100, 200, 300, 500, 600, 700, 900, 1000, 1100, 1300, 1400
 * and 1500. Otherwise, calculate it.
 */

if ( yy - 1 > 1582L )
    day_num += 12L;
else
{
    day_num += ( yy - 1 ) / 100L;
    day_num -= ( yy - 1 ) / 400L;
}

/* now, add the days elapsed in the year so far */

if ( IsYearLeap ( date->tm_year ) == NO )
    while ( --mm )
        day_num += Days_in_month[ mm ];
else
    while ( --mm )
        day_num += Days_in_month_leap[ mm ];

/* add days in current month for the year being evaluated */

day_num += date->tm_mday;
```

图7-6 （续）

```
    /*
     * now adjust for the 10 days cut out of the calendar when
     * the change was made to the Gregorian calendar. This change
     * reflects the jump from October 4 to October 15, 1582, a
     * deletion of 10 days.
     */

    if ( day_num > 577737L )
        day_num -= 10L;

    return ( day_num );
}
```

图 7-6 （续）

该函数的代码充分加了注释，使得它可以是自解释的。唯一确实不同寻常的代码出现在函数的末尾，一旦计算了日编号，就会对其进行处理。该代码如下：

```
if ( day_num > 577737L )
      day_num -= 10L;
```

直到函数中的这个位置，日编号计算还没有考虑1582 年 10 月 4 日后面接着 10 月 15 日。这种测试确定计算的日期是否位于 1582 年 10 月 4 日（从公元 1 年 1 月 1 日起，它将是第 577 737 天）之后。如果计算的日期在这之后，函数就会从日编号中减去 10 天，这 10 天标记着格列高利历的开始。

日编号是所有日期和日历函数的基石。例如，为了确定一周中的某一天，可以获取日编号，然后使用模函数。任何给定的日编号对 7 求模都将返回一个 0 ~ 6 之间的唯一值，它可用于指示一周中的某一天。例如，图 7-7 中的 DayOfWeek( ) 接受一个 tm 结构，并返回一个 0 ~ 6 之间的整数，其中 0 表示星期天。

```
/*-------------------------------------------------------------
 * Returns an integer from 0-6 signifying the day of the week,
 * where 0 = Sunday and 6 = Saturday. Returns -1 on error.
 *-------------------------------------------------------------*/
int DayOfWeek ( struct tm *date )
{
    long day_num = DateToDayNumber ( date );

    if ( day_num < 0 )
        return ( -1 );

    return ( (int) (( day_num % 7 ) + 5 ) % 7 );
}
```

图 7-7 DayOfWeek( ) 函数计算给定的一周中的某一天

根据惯例，在西方日历中，每个星期都开始于星期天，因此一般给星期天赋予值 0，并给星期六赋予值 6。一旦 DayOfWeek( ) 接受一个包含日期的 tm 结构，它就会计算日编号。然后用这

个日编号除以7，并获得余数（使用求模运算符）。由于我们的计数系统不是开始于星期天，必须用求模的结果加上5，以便获得一致的结果，即星期天的值为0并且星期六的值为6。不过，把求模的结果加上5可能生成一个大于6的结果；因此，将第二次执行对7求模的运算。然后返回最终的值，作为一周中的某一天。

一份类似的有用信息（我们稍后将需要它）表示它是一年中的哪一天——例如，2月25日是一年中的第56天。图7-8中所示的函数DayOfYear()接受一个日期结构，并把它转换为日编号。然后，它计算那一年1月1日的日编号，并返回两个日编号之间的差值，作为一年中的某一天。

```
/*-------------------------------------------------------------
 * Returns an integer from 1-366 signifying the day of the year.
 * where 1 is Jan 1, 365 or 366 is Dec 31, and -1 is an error.
 *-------------------------------------------------------------*/
int DayOfYear ( struct tm *date )
{
    struct tm jan1;

    if ( LoadDateStruct ( &jan1, date->tm_year, 1, 1 ) == -1 )
        return ( -1 );

    return ((int) ( DateDiff ( date, &jan1 ) + 1 ));
}
```

图7-8 DayOfYear()函数确定给定日期是一年中的哪一天，如果出错就返回-1

日编号的另一个有用的应用是告诉你两个日期相差多远的函数。这个函数DateDiff()（如图7-9所示）只计算两个单独日期的日编号，然后从第一个日编号中减去第二个日编号。

```
/*-------------------------------------------------------------
 * Accepts pointers to two date structures and returns the
 * difference in days between them as a long int. Negative if
 * the first date is earlier, positive if it is later. No error
 * return is possible, so programmers must make sure the dates
 * are valid.
 *-------------------------------------------------------------*/
long DateDiff ( struct tm *d1, struct tm *d2 )
{
    return ( DateToDayNumber ( d1 ) - DateToDayNumber ( d2 ));
}
```

图7-9 DateDiff()函数计算两个给定日期之间的天数，不返回错误代码

由于这个函数可以返回任何长整型值，因此它不能返回错误代码。在调用该函数之前，必须确保日期是有效的；否则，结果可能不可靠。

在调用了上述任何函数之后，显然需要在某个时间把日编号转换为有效的日期。图7-10中所示的函数DayNumberToDate()接受一个长整数和一个指针，前者包含日编号，后者指向一个tm结构，函数将把得到的日期放入其中。

```
/*-----------------------------------------------------------
 * Converts a day number into an actual date.
 *-----------------------------------------------------------*/
int DayNumberToDate ( long int day_num, struct tm *date )
{
    int     dd, mm, yy, i;
    long    days_left;

    if ( day_num > 577737L )
        day_num += 10L;

    yy = (int) ( day_num / 365 );
    days_left = day_num % 365L;

    /* prior to 1700, all years evenly divisible by 4 are leap */

    if ( yy < 1700 )
        days_left -= ( yy / 4 );
    else
    {
        days_left -= ( yy / 4 );     /* deduct leap years */
        days_left += ( yy / 100 );   /* add in century years */
        days_left -= ( yy / 400 );   /* deduct years / 400 */
        days_left -= 12;             /* deduct century years that
                                        were leap before 1700 */
    }

    /* make sure days left is > 0 */

    while ( days_left <= 0 )
    {
        if ( IsYearLeap ( yy ) == YES )
        {
            yy -= 1;
            days_left += 366;
        }
        else
        {
            yy -= 1;
            days_left += 365;
        }
    }

    /*
     * yy holds the number of elapsed years.
     * So, add 1 for the current year.
     */
```

图 7-10　DayNumberToDate( ) 把编号转换回有效的日期。如果出现错误，就返回 -1

```
    yy += 1;

    /*
     * now deduct the days in each month, starting
     * from January to find month and day of month.
     * adjust for leap year, of course.
     */

    dd = (int) days_left;
    mm = 0;

    if ( IsYearLeap ( yy ) == YES )
        for ( i = 1; i <= 12; i++ )
        {
            mm = i;
            if ( dd <= Days_in_month_leap[i] )
                break;
            else
                dd -= Days_in_month_leap[i];
        }
    else
        for ( i = 1; i <= 12; i++ )
        {
            mm = i;
            if ( dd <= Days_in_month[i] )
                break;
            else
                dd -= Days_in_month[i];
        }

    if ( LoadDateStruct ( date, yy, mm, dd ) == -1 )
        return ( -1 );
    else
        return ( 0 );
}
```

图7-10 （续）

与把日期转换为日编号的例程相比，上面的代码要稍微复杂一点。该代码首先调整从格列高利历开始日期之后的天数。然后，它用日编号除以365，并把商存储在yy中，并试着猜测它是哪一年。将通过这个除法得到的余数存储在days_left中。然后，例程尝试从days_left中减去出现在yy之前的所有闰日。如果days_left下降到0以下，就从yy中减去一年，并把days_left加上365或366，直到它大于0为止。此时，yy将保存正确的完整年数。days_left的值将告诉我们月和日。由于月和日将出现在完整的年数之后的一年中，在开始计算月和日之前将递增yy。这些最后的计算易于执行，并在代码中做了充分解释。

下面给出一个如何使用本章中迄今为止所展示例程的例子，如程序清单7-1（lilcal.c）所示。这个程序执行UNIX的cal实用程序所做工作的一个子集。当不带参数地调用lilcal时，它将打印当前月份的日历。如果利用月份和年份调用它，它就会打印那个月的日历。后一个命令行的示例

是 lilcal 7 1776。这个命令将打印 1776 年 7 月这个重大月份的日历。

程序清单 7-1　lilcal.c 是一个执行 UNIX 的 cal 实用程序的大量日历打印能力的程序

```
/*--- lilcal.c ------------------------------------------------
 * lilcal works similarly to the UNIX cal program.
 * If you type lilcal with no arguments, it prints the calendar
 * for the current month; otherwise, provide it with two argu-
 * ments: the month and year. Hence, lilcal 9 56 will print the
 * calendar for September 1956. October 1582 is hardwired as
 * a special case. UNIX cal allows you to print the calendar for
 * a whole year. Since this code is cosmetic rather than algo-
 * rithmic, it is not included here.
 *-----------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "datelib.h"

void PrintCalendar ( int, int );
void PrintOct1582 ( void );

int main ( int argc, char *argv[] )
{
    int start_yy, start_mm;

    time_t timer;
    struct tm *tms;

    if ( argc != 1 && argc != 3 )
    {
        fprintf ( stderr, "lilcal requires 0 or 2 arguments\n" );
        return ( EXIT_FAILURE );
    }

    if ( argc == 1 )    /* current month */
    {
        timer = time ( NULL );
        tms   = localtime ( &timer );
        tms->tm_mon += 1;

        start_mm = tms->tm_mon;
        start_yy = tms->tm_year;
        start_yy += 1900;
    }
    else
    if ( argc == 3 )
    {
        start_mm = atoi ( argv[1] );
        start_yy = atoi ( argv[2] );
    }

    /*
     * due to the calendar change, October 1582 is a unique
     * month and must be provided for specially.
     */
```

```
    if ( start_yy == 1582 && start_mm == 10 )
        PrintOct1582();
    else
        PrintCalendar ( start_yy, start_mm );

    return ( EXIT_SUCCESS );
}

void PrintCalendar ( int cal_yy, int cal_mm )
{
    char      buffer [40];
    struct tm date;
    int       day_of_week;
    int       days_in_month;
    int       days_to_print[42];
    int       i, j, rows;

    if ( LoadDateStruct ( &date, cal_yy, cal_mm, 1 ) == -1 )
    {
        fprintf ( stderr, "Illegal month: %02d-%02d\n",
                    cal_yy, cal_mm );
        return;
    }

    /* get the day of the week for the 1st of month */

    day_of_week = DayOfWeek ( &date );

    /* find out how many days in the month */

    switch ( cal_mm )
    {
        case  1:
        case  3:
        case  5:
        case  7:
        case  8:
        case 10:
        case 12:
            days_in_month = 31;
            break;

        case  4:
        case  6:
        case  9:
        case 11:
            days_in_month = 30;
            break;

        case  2:
            days_in_month = 28;
            if ( IsYearLeap ( cal_yy ) == YES )
                days_in_month += 1;
            break;
    }

    /* load up the days to print */

    for ( i = 0; i < 42; i++ )
```

```
        days_to_print[i] = 0;

    for ( i = day_of_week, j = 1; j <= days_in_month; i++, j++ )
        days_to_print[i] = j;

    /* now print the calendar */

    TimeSprintf ( buffer, 40, "    %B %Y", &date );
    printf ( "%s\n", buffer );
    printf ( "  S  M Tu  W Th  F  S\n" );

    for ( rows = 0; rows < 6; rows++ )  /* print up to 6 rows */
    {                                   /* of 7 days.         */
        for ( i = 0; i < 7; i++ )
            if ( days_to_print[(rows * 7) + i] == 0 )
                printf ( "   " );
            else
                printf ( " %2d",
                         days_to_print[(rows * 7) + i] );
        printf ( "\n" );
    }
}
void PrintOct1582 ( void )
{
    printf ( "    October 1582\n" );
    printf ( "  S  M Tu  W Th  F  S\n" );
    printf ( "     1  2  3  4 15 16\n" );
    printf ( " 17 18 19 20 21 22 23\n" );
    printf ( " 24 25 26 27 28 29 30\n" );
    printf ( " 31\n" );
}
```

该程序把月份和年份加载进 tm 结构中，并指定那个月的第一天。然后，它调用 DayOfWeek ( ) 查明那个月开始于哪一天。接下来，它验证那个月中有多少天。然后，它把那个月中的每一天移入合适的日历时段中。最后，它一次打印 7 个条目的数组（即一个日历行）。注意：对 1582 年 10 月进行了明确的测试，并为这个月提供了一个硬编码的解决方案。这个程序还会调用 TimeSprintf( )，在后面的程序清单 7-14 中展示了这个日期格式化例程。

lilcal. c 中的代码调用了 datelib. h，后者包含库函数的原型。这个文件出现在程序清单 7-2 中。

程序清单 7-2　datelib. h 中包含日期和时间例程的原型

```
/*-----------------------------------------------------------
 * Prototypes for datelib.c
 *-----------------------------------------------------------*/

long    DateDiff ( struct tm *, struct tm * );
long    DateToDayNumber ( struct tm * );
int     DayNumberToDate ( long int, struct tm * );
int     DayOfWeek ( struct tm * );
int     DayOfYear ( struct tm * );
int     IsDateValid ( struct tm * );
int     IsTimeValid ( struct tm * );
int     IsYearLeap ( int );
int     LoadDateStruct ( struct tm *, int, int, int );
int     LoadTimeStruct ( struct tm *, int, int, int );
double  TimeDiff ( struct tm *, struct tm * );
```

```
int     TimeSprintf ( char *, int, char *, struct tm * );

#ifndef YES
#define YES     1
#define NO      0
#endif
```

注意：这些原型包括时间函数。下一节将讨论这些函数，还将讨论以多种格式打印日期和时间的重要例程。

7.2 时间例程

前面关于日历的讨论表明：从历史上讲，日历确实不足以用一种可预测、顺序的方式记录日和年。今天，我们使用的日历系统实质上是通用的和可预测的。只要我们使用取自于格列高利历的日期，就可以完全精确地计算最近和遥远未来的日期。不过，考虑格列高利历以前的日期给日期操作注入了怀疑和混淆的元素。

时间度量也遭受了一些同样令人烦恼的问题。回忆可知：通过参照天文事件来度量时间，比如地球的旋转（日）或者地球围绕太阳的旋转（年）。因此，对日的细分（秒、分钟和小时）与一天的长度精确相关——也就是说，每分钟60秒，每小时60分，每天24小时。不过，地球在其旋转过程中会产生非常微小的偏差，从而要求定期增加几秒钟的时间（称为**闰秒**）来补偿这些不规则性。闰秒（其数量从0到2不等）一般由全世界的标准组织每个季度公布一次，并把它加到该季度的最后一天上去。在ANSI C标准中明确考虑了闰秒：tm结构可以在tm_sec中保存秒数值0~61。由于无法预测何时添加闰秒或者要添加多少闰秒，因此不可能精确度量具体的未来事件的时间间隔。下面的例程没有考虑闰秒，并且假定每分钟60秒。

大多数时间例程都从系统中提取当前时间，并操纵以打印出它，或者把它用作计时事件的基础。ANSI C库提供了众多可以很好地执行这些任务的函数。这里将不会重复介绍这些函数。相反，我们将介绍一些函数，用于把时间加载进tm结构中、验证时间、以秒为单位度量两个事件之间的距离，以及以各种广泛的格式打印时间。

把时间加载进tm结构中与加载日期并无二致。因此，该函数的代码（如图7-11所示）非常直观。

```
/*-------------------------------------------------------
 * Loads time into an existing tm structure.
 * Returns 0 on success, -1 on error
 *-------------------------------------------------------*/
int LoadTimeStruct ( struct tm *date, int hh, int mm, int ss )
{
    date->tm_hour = hh;
    date->tm_min  = mm;
    date->tm_sec  = ss;

    if ( IsTimeValid ( date ) == NO )
        return ( -1 );
    else
        return ( 0 );
}
```

图7-11 LoadTimeStruct() 函数接受一个指向tm结构的指针，它把传递用于小时、分钟和秒的整数加载进该结构中

注意：该函数没有在 tm 结构中利用日期字段。因此，如果日期对你很重要，就可以在加载时间之前或之后，自己把它加载进结构中（使用 LoadDateStruct( )）。

LoadTimeStruct( ) 通过调用 IsTimeValid( ) 并给它传递一个指向 tm 结构的指针来验证时间，如图 7-12 所示。如果时间的各个组成部分都在正确的范围内（使用 24 时制并且每分钟 60 秒），函数就会返回 YES；否则，它会返回 NO。

```
/*-----------------------------------------------------------
 * Returns YES if time is valid; otherwise returns NO.
 *----------------------------------------------------------*/
int IsTimeValid ( struct tm * date )
{
    if ( date->tm_hour < 0 || date->tm_hour > 23 )
        return ( NO );

    if ( date->tm_min < 0 || date->tm_min > 59 )
        return ( NO );

    /*
     * Technically speaking, ANSI allows for minutes with
     * 61 or 62 seconds for leap seconds that are added
     * with little warning by astronomical societies to
     * reflect slight changes in the earth's rotation.
     * For reasons discussed in the text, however, we
     * observe only the 60-second minute.
     */

    if ( date->tm_sec < 0 || date->tm_sec > 59 )
        return ( NO );

    return ( YES );
}
```

图 7-12 IsTimeValid( ) 函数验证 tm 结构中的时间字段

函数 TimeDiff( ) 类似于 ANSI C 函数 difftime( )。它接受两个 tm 结构，其中分别加载了时间和日期信息，如图 7-13 所示。然后，它从第二个时间中减去第一个时间，并返回一个双精度型的秒数。由于任何双精度值（正值或负值）都可能是正确的，所以不会返回错误代码。因此，要确保提交给该函数的数据是有效的。

TimeDiff( ) 超过 ANSI difftime( ) 的好处在于 difftime( ) 具有以下局限性：所有的时间都必须发生在 1900 年或以后的某个日期。而我们的函数则可以接受从公元 1 年 1 月 1 日起的所有日期和时间。

7.3 用于日期和时间数据的格式

前面研究过的几乎所有的日期和时间例程都会出现在需要以一种程序员选择的格式打印或显示日期或时间的应用程序中。由于这些格式在区域（有许多种不兼容的国际格式）和用户偏好方面相差甚远，ANSI C 库提供了一个类似于 sprintf( ) 的函数，它允许设计信息的格式。这个函数 strftime( ) 像 sprintf( ) 一样，接受一个输出缓冲区和一个格式字符串。它还接受一个变量，指示输出字符串的最大值，并接受一个指向 tm 结构的指针，该结构中包含要格式化的数据。

```
/*-------------------------------------------------------------
 * TimeDiff() operates like difftime(); however, it can accept
 * times before Jan 1, 1900. No error return.
 *-------------------------------------------------------------*/
double TimeDiff ( struct tm *t1, struct tm *t2 )
{
    long day_num;
    double d1, d2;

    day_num = DateToDayNumber ( t1 );
    d1 = day_num * 86400.0 + t1->tm_hour * 3600.0 +
         t1->tm_min * 60.0 + t1->tm_sec;

    day_num = DateToDayNumber ( t2 );
    d2 = day_num * 86400.0 + t2->tm_hour * 3600.0 +
         t2->tm_min * 60.0 + t2->tm_sec;

    return ( d1 - d2 );
}
```

图 7-13 给 TimeDiff( ) 函数传递两个加载了有效日期和时间数据的 tm 结构，然后返回两个事件之间的时间（以秒为单位）。注意返回类型是双精度型

格式字符串使用一组格式字符，它们不同于 sprintf( ) 的那些格式字符。表 7-1 中列出了用于 strftime( ) 的格式字符。

表 7-1 ANSI C 的 strftime( ) 函数中使用的格式字符

格式字符	描 述	示例输出
%a	星期几的名称（3 个字符）	Mon
%A	星期几的名称（完整名称）	Monday
%b	月份名称（3 个字符）	Oct
%B	月份名称（完整名称）	October
%c	日期和时间（UNIX 风格）	Oct 3 10：11：12 1994
%d	一月中的某一天，以 2 位数字表示（01～31）	03
%H	小时，以 2 位数字表示（01～23）	22
%I	小时，以 2 位数字表示（01～12）	10
%j	一年中的某一天，以 3 位数字表示（001～366）	309
%m	月份，以 2 位数字表示（01～12）	10
%M	分钟，以 2 位数字表示（00～59）	11
%p	上午或下午	AM
%S	秒，以 2 位数字表示（00～59）	12
%U	以 2 位数字表示的星期编号，其中每个星期从星期天开始，并且第 1 个星期是一年中第一个完整的星期（00～53）	37
%w	星期几（0～6，0 = 星期天）	1
%W	以 2 位数字表示的星期编号，其中每个星期从星期一开始，并且第 1 个星期是一年中第一个完整的星期（00～53）	7
%x	日期字符串（11 个字符）	Oct 3 1994
%y	不带百年的年份，以 2 位数字表示	94
%Y	带有百年的年份，以 4 位数字表示	1994
%Z	时区字符，如果没有使用时区，就不会有这些字符	EST
%%	类似于 printf( ) 的% 符号	%

在 strftime( ) 中使用格式化选项，可以用广泛的格式表示时间和日期，包括国际格式。不过，strftime( ) 只适用于从 1900 年开始的年份。由于这种局限性，我们提供了 TimeSprintf( )，如图 7-14 所示。这个函数的工作方式与 strftime( ) 相同——接受相同的参数并且生成相同的输出——只不过如果发现无效的格式字符，TimeSprintf( ) 将输出一条错误消息。TimeSprintf( ) 中的代码大量使用了本章前面展示的日期和时间例程，并且很好地演示了如何使用这些函数。要稍微注意一下：像%c（打印日期和时间）这样的参数将会递归地调用 TimeSprintf( )，因为它们的输出确实是 TimeSprintf( ) 可以单独生成的一系列格式化字段。

```
/*-------------------------------------------------------------
 * Operates like strftime() except:
 *  a) it returns -1 on error;
 *  b) strftime() expects tm_year to be 2 digits and so always
 *     adds 1900 to the year, while TimeSprintf() believes the
 *     year to be the actual year: 94 is AD 94 and so does not
 *     add 1900.
 *-------------------------------------------------------------*/
int TimeSprintf ( char *string, int max_size,
                  char *format, struct tm *ptm )
{
    char *pf;
    int  i, j;

    int  yy;        /* scratch variable for year */
    int  mon;       /* our internal use of the month is 1-12 */
    char *buffer;   /* where the output string goes */

    mon = ptm->tm_mon + 1;

    buffer = malloc ( 1024 );   /* largest likely output string */
    if ( buffer == NULL )
        return ( -1 );

    i = 0;    /* where we are in the output string */


    for ( pf = format; *pf != '\0' && i < 1024; pf++ )
    {
        if ( *pf != '%' )
            buffer[i++] = *pf;
        else
            switch ( *(++pf) )
            {
                /*--- day of the week: Sunday, Monday...---*/

            case 'a':               /* 3-letter weekday */
                j = DayOfWeek ( ptm );
                strncpy ( buffer + i,
                          Day_name_abbrev[j], 3 );
                i += 3;
                break;
```

图 7-14 TimeSprintf( ) 函数以多种格式打印日期和时间

```
case 'A':                   /* full weekday */
    j = DayOfWeek ( ptm );
    strcpy ( buffer + i, Day_name[j] );
    while ( *(buffer + i) != '\0' )
        i++;
    break;

case 'w':                   /* weekday as a digit */
    j = DayOfWeek ( ptm );
    buffer[i++] = j + '0';
    break;

    /*--- day of the month: 1, 2, 3...---*/

case 'd':                   /* day as a number */
    buffer[i++] = ptm->tm_mday / 10 + '0';
    buffer[i++] = ptm->tm_mday % 10 + '0';
    break;

case 'm':                   /* month as a number */
    buffer[i++] =  mon  / 10 + '0';
    buffer[i++] =  mon  % 10 + '0';
    break;

    /*--- month ---*/

case 'b':                   /* 3-letter month */
    strncpy ( buffer + i,
              Month_name_abbrev[mon], 3 );
    i += 3;
    break;

case 'B':                   /* full month name */
    strcpy ( buffer + i, Month_name[mon] );
    while ( *(buffer + i) != '\0' )
        i++;
    break;

    /*--- year ---*/

case 'Y':                   /* year with century: 1994 */
    yy = ptm->tm_year;

    if ( yy < 0 )
    {
        buffer[i++] = '-';
        yy = -yy;
    }

    if ( yy > 9999 )
    {
        buffer[i++] = yy / 10000 + '0';
        yy %= 10000;
    }
```

图7-14 （续）

```
    if ( yy > 999 )
    {
        buffer[i++] = yy / 1000 + '0';
        yy %= 1000;
    }

    if ( yy > 99 )
    {
        buffer[i++] = yy / 100 + '0';
        yy %= 100;
    }

    buffer[i++] = yy / 10 + '0';
    buffer[i++] = yy % 10 + '0';
    break;

case 'y':                   /* year without century: 94 */
    yy = ptm->tm_year;

    if ( yy < 0 )
    {
        buffer[i++] = '-';
        yy = -yy;
    }

    if ( yy > 100 )
        yy %= 100;

    buffer[i++] = yy / 10 + '0';
    buffer[i++] = yy % 10 + '0';
    break;

case 'x':                   /* locale-specific date */
{
    char s[18];

    TimeSprintf ( s, 18, "%a, %b %d, %Y", ptm );
    strcpy ( buffer + i, s );
    i += strlen ( s );
}
    break;
    /*--- time in numbers ---*/

case 'I':                   /* hour on 12-hr clock */
{
    int hour;

    if ( ptm->tm_hour > 12 )
        hour = ptm->tm_hour - 12;
    else
        hour = ptm->tm_hour;

    buffer[i++] = hour / 10 + '0';
    buffer[i++] = hour % 10 + '0';
}
    break;
```

图 7-14 （续）

```
case 'H':                   /* hour on 24-hr clock */
    buffer[i++] = ptm->tm_hour / 10 + '0';
    buffer[i++] = ptm->tm_hour % 10 + '0';
    break;

case 'M':                   /* minute as a number */
    buffer[i++] = ptm->tm_min / 10 + '0';
    buffer[i++] = ptm->tm_min % 10 + '0';
    break;

case 'S':                   /* seconds as a number */
    buffer[i++] = ptm->tm_sec / 10 + '0';
    buffer[i++] = ptm->tm_sec % 10 + '0';
    break;

case 'X':                   /* time string: hh:mm:ss */
{
    char s[9];

    TimeSprintf ( s, 9, "%H:%M:%S", ptm );
    strcpy ( buffer + i, s );
    i += 8;
}
    break;

    /*--- miscellaneous ---*/

case 'c':                   /* date and time */
{
    char s[28];

    TimeSprintf ( s, 28, "%b %d %H:%M:%S %Y", ptm );
    strcpy ( buffer + i, s );
    i += strlen ( s );
}
    break;

case 'j':                   /* day of the year: 1-366 */
{
    int day_of_year;

    day_of_year = DayOfYear ( ptm );

    if ( day_of_year < 10 )
        buffer[i++] = day_of_year + '0';
    else
    {
        if ( day_of_year > 99 )
        {
            buffer[i++] = day_of_year / 100 + '0';
            day_of_year %= 100;
        }

        buffer[i++] = day_of_year / 10 + '0';
        buffer[i++] = day_of_year % 10 + '0';
```

图7-14 （续）

```
    }
}
    break;

case 'p':                   /* AM or PM */
    if ( ptm->tm_hour < 12 )
        buffer[i++] = 'A';
    else
        buffer[i++] = 'P';
    buffer[i++] = 'M';
    break;

case 'U':                   /* Sunday week of year */
case 'W':                   /* Monday week of year */
{
    int day_of_year,
        day_of_week,
        sunday_week;

    struct tm jan1;

    int jan1_dow;           /* jan 1 day of week */

    day_of_year = DayOfYear ( ptm );
    day_of_week = DayOfWeek ( ptm );

    LoadDateStruct ( &jan1, ptm->tm_year, 1, 1 );
    jan1_dow = DayOfWeek ( &jan1 );

    if ( jan1_dow != 0 )
        day_of_year -= ( 7 - jan1_dow );

    if ( day_of_year < 1 )
        sunday_week = 0;
    else
    {
        sunday_week = 1;
        sunday_week += ( day_of_year - 1 ) / 7;
    }

    if ( *pf == 'W' )   /* if Monday week */
        if ( day_of_week < 1 )
            sunday_week -= 1;

    if ( sunday_week > 9 )
    {
        buffer[i++] = ( sunday_week / 10 ) + '0';
        sunday_week %= 10;
    }

        buffer[i++] = sunday_week + '0';
    }
        break;

   case 'Z':                /* the time zone */
```

图 7-14 （续）

```
{
    char *temp;

    temp = setlocale ( LC_TIME, NULL );
    if ( temp == NULL )
        buffer[i++] = 'C';
    else
    {
        strcpy ( buffer + i, temp );
        i += strlen ( temp );
    }
}
    break;

case '%':                /* the percent sign */
    buffer[i++] = '%';
    break;

default:
                strcpy ( buffer + i, "Error in TimeSprintf() " );
                i += 23;
            }
    }
    buffer[i] = '\0';

    strncpy ( string, buffer, max_size );
    free ( buffer );

    if ( i <= max_size )
        return ( i );
    else
        return ( 0 );
}
```

图 7-14 （续）

7.4 最后的提醒

本章中展示的例程提供了在执行几乎所有的日历编程时所需的所有基本功能。只需确保：如果使用1753年以前的日期，就要小心决定你认可在哪一年转换到格列高利历（1752年，美国日期；或1582年，本章中或者大多数欧洲国家使用的日期）。注意：重大年份之前的所有日期都有相当大的可能性是虚假的。一般更多的是根据惯例给它们赋值，而不是直接反映日历对于任何给定的日子是从哪一天开始的。

7.5 资源和参考资料

Meyer, Peter. “Julian and Gregorian Calendars.” *Dr. Dobb's Journal*, March 1993（#198）. 这篇文章是最近关于计算日历的最佳讨论。其中的解释清晰易懂，但是它们的代码实现却难以理解。其中的代码还包含一些怪异的方面。

西方日历史已经编写了许多次。在几乎每一本历书中都出现了一份良好的大纲。其他参考书籍不同程度地详细讨论了这个迷人的主题。

第8章　任意精度的算术

任意精度的算术（arbitrary-precision arithmetic）允许用户利用在许多位置都很准确的结果来执行算术运算。本章中展示的例程在某个精度级别执行加法、减法、乘法和除法运算，它在实际中受限于用户的栈空间，并且绝对受限于短整型的大小。32 768 位的绝对限制应该可以满足几乎所有可想得到的应用程序的需要。

使用任意精度的算术有两个主要目的。第一个目的是操纵那些比本机平台直接支持的数字更大的数字——一般在尾数中大约有 18 位，指数中的数量级大约为 300。第二个目的是在所有数字中无论大小都要避免舍入错误。

在许多情况下，使用的数字会超过在 C 语言及其他编程语言中可用的双精度数和浮点数的范围，这样做是有益的。数论、随机数生成及其他类型的数学研究通常要求能够处理非常大的数字，它们超出了标准硬件处理器的作用范围。不过，任意精度的算术主要用于解决麻烦的舍入错误，它们是由在内存中存储数字的方式引起的。使用二进制数字模拟十进制数字引入了一系列复杂性，特别是当需要表示分数时。为了查看这项任务有多困难，只需要知道以二进制形式存储普通分数的难度。当考虑将怎样以传统的二进制项存储像 1/10 这样的数字时，可以清楚看出把十进制分数映射到二进制形式的过程充斥着在表示中出错的可能性。

这些错误具有两种形式：浮点数的总和不准确，以及相邻的非常小的数字倾向于变成相等。这两种错误形式的累积使得浮点数不适用于会计。由于标准的 32 位芯片上可用的最大整数允许达到 $\pm 2 \times 10^9$ 的整数，用于超过 2000 万美元的会计软件将不能记录美分——这是一种不可接受的限制。

出于科学的目的，不能区分非常小的数字是一种严重的折中。Nakamura［Nakamura 1993］说明了一个数字必须变得多少才能使它不可区分。在表 8-1 中可以看到，对这个问题的调查研究表明：当把像 Cray 这样的超级计算机排除在外时，数字（一般称为计算机 ε）变化很小。

表 8-1　多个流行平台的计算机 ε

精　度	IBM PC	IBM 370	VAX 11	Cray XMP
单精度	1.19E-7	9.53E-7	1.19E-7	3.55E-15
双精度	2.77E-17	2.22E-16	2.77E-17	1.26E-29

在不能处理小于 10^{-17} 的数字或者大于 2100 万美元的总额之间，有一些令人信服的理由让我们放弃浮点计算，以支持任意精度的算术。本章中使用的方法是：把整数存储为一些数字的数组，其中每个数字都占据一个字节。

8.1　构建计算器

可以以独立的方式使用在本章中展示的任意精度的例程。不过，为了演示它们的使用，给它们绑定了三个前端：计算器、使用牛顿的方法对平方根进行的高精度计算以及分期付款表。计算

器将在本节中讨论；另外两种应用将在本章末尾讨论。

这里展示的计算器可以交互式地使用或者以批处理方式使用。如果不带参数地从命令行运行计算器，它将要求用户输入两个项，并要求提供对它们执行的运算。然后，它会在屏幕上显示答案并退出。如果提供一个参数，计算器将期望参数是一个脚本，其中列出了项和要执行的运算。程序把这些项写到名为 longmath. out 的文件中，除非提供了第二个命令行参数，在这种情况下，将把该参数用作输出文件名。

输入文件包含一个普通的文本文件，其中每一行中具有一个条目。列出这些条目的方式与将在正常计算器上输入它们的方式相同：第一个项，其后接着一个运算符，再接着第二个项。然后把计算的结果转为下一个运算的第一个项。为了开始一个新的运算，用户指定一个 C 或 c 运算符（用于执行清理），它会把所有项重置为 0。图 8-1 显示了一个示例脚本文件，其后接着图 8-2 中的结果输出文件。在答案文件中，把精度设置为 20 位数字。

```
12
/
7
-
1.0
*
2
+
16
/
8
C
123456.789012
/
3.141592653589793
```

图 8-1 用于任意精度的计算器的示例脚本文件

```
12
/
7
=
1.7142857142857142857
-
1.0
=
0.7142857142857142857
*
2
=
1.4285714285714285714
+
16
```

图 8-2 从图 8-1 中的脚本得到的答案文件

```
=
17.428571428571428571
/
8
=
2.1785714285714285714
C
123456.789012
/
3.141592653589793
=
39297.516459025981773
```

图 8-2 （续）

用于计算器的代码（如程序清单 8-1 所示）首先创建三个数据结构：其中两个用于保存将输入的项，第三个用于保存计算的结果。接下来，程序将检查它是否是交互式调用的，然后从用户或者输入脚本获取输入。它把计算的项转换为一种内部形式，把它们加载进上述的数据结构中，执行运算，并在屏幕上显示答案或者把它写到输出文件。如果程序正在运行脚本，它就会重复这个过程，直到脚本末尾。

程序清单 8-1 calcmain.c 为计算器处理输入和输出，并执行指定的计算

```
/*--- calcmain.c ------------------------- Listing 8-1 --------
 * Calculator main line for longmath routines.
 *
 * If the program is invoked without any arguments, it responds
 * as an interactive command-line calculator, with all necessary
 * prompts on the screen. Otherwise:
 *
 * If one argument is supplied, it is taken to be a filename,
 * for a calculator script that may contain the following:
 * terms in the prescribed form,
 * the arithemetic operators: +, -, *, /
 * the directive 'C' which clears and resets the calculator to 0
 *
 * In the absence of a clear command, the result of the previous
 * operation is carried over into the first term.
 *
 * If a second file is specified, all terms and operations and
 * their results are written to it; otherwise, the output file
 * is longmath.out
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "longmath.h"

int main ( int argc, char *argv[] )
{
    struct TermData *term1,     /* two terms and the solution */
                    *term2,
```

```
                *solution;

int operator;
int format  = NORMAL;       /* default to NORMAL notation */

char buffer [2*MAX_SIZE];   /* buffer for user input */

/*--- allocate and initialize the terms ---*/

term1    = TermCreate();
term2    = TermCreate();
solution = TermCreate();

if ( term1 == NULL || term2 == NULL || solution == NULL )
{
    fprintf ( stderr, "Cannot allocate memory\n" );
    return ( EXIT_FAILURE );
}

/*--- the interactive calculator ---*/

if ( argc < 2 )     /* no input file specified, hence  */
{                   /*  it must be interactive.        */

    printf ( "Arbitrary-Precision Math. Press Q to Quit\n" );

    /* get the first term */

    printf ( "Type first term and press <Enter>\n" );
gets ( buffer );
if ( *buffer == 'q' || *buffer == 'Q' )
    return ( EXIT_SUCCESS );
else
{
    format = AsciiToTerm ( buffer, term1 );
    if ( ! format )
    {
        printf ( "Invalid term; aborting!\n" );
        exit ( EXIT_FAILURE );
    }
}

/* get the second term */

printf ( "Type second term and press <Enter>\n" );
gets ( buffer );
if ( *buffer == 'q' || *buffer == 'Q' )
    return ( EXIT_SUCCESS );
else
{
    format = AsciiToTerm ( buffer, term2 );
    if ( ! format )
    {
        printf ( "Invalid term; aborting!\n" );
        exit ( EXIT_FAILURE );
    }
}
```

```
    /* get the desired operation */

    printf (
      "Type operation ( + - * / ) and press <Enter>\n" );

    operator = getchar();
    if ( operator == 'q' || operator == 'Q' )
        return ( EXIT_SUCCESS );

    if ( operator != '+' && operator != '-' &&
         operator != '*' && operator != '/' )
    {
        printf ( "Invalid operator; aborting!\n" );
        exit ( EXIT_FAILURE );
    }

    /*--- compute the result --- */
    if ( ! ComputeResult (
              term1, operator, term2, solution ))
     return ( EXIT_FAILURE );

    /*--- print the answer ---*/

    TermToAscii ( solution, buffer, format );

    printf ( "%s\n", buffer );

    return ( EXIT_SUCCESS );
}
else    /* calculator operations in a file */
{
    FILE *fin, * fout;            /* file handles */
    char fin_name[64],            /* file names */
         fout_name[64];

    strcpy ( fin_name, argv[1] );
    if ( argc > 2 )
        strcpy ( fout_name, argv[2] );
    else
        strcpy ( fout_name, "longmath.out" );

    fin = fopen ( fin_name, "rt" );
    if ( fin == NULL )
    {
        fprintf ( stderr, "Error opening: %s\n", fin_name );
        return ( EXIT_FAILURE );
    }

    fout = fopen ( fout_name, "wt" );
    if ( fout == NULL )
    {
        fprintf ( stderr, "Error opening: %s\n", fout_name );
        return ( EXIT_FAILURE );
    }

    while ( ! feof ( fin ))
    {
```

```
        /* get the first term */

        format =  GetFileTerm ( term1, fin, fout );
        if ( ! format )
            break;

        /* get the operator */
if ( ! GetFileOperator ( &operator, fin, fout ))
    break;

continuing:  /* we'll jump to here if further ops */

/* get next term */

format =  GetFileTerm ( term2, fin, fout );
if ( ! format )
    break;

/*
 * if an error occurs in computing, abort *unless*
 * the next operation is a 'clear,' which starts
 * a new calculation.
 */

if ( ! ComputeResult (
            term1, operator, term2, solution ))
{
    fprintf ( stderr, "Error in Computing.\n" );
    fprintf ( fout, "Error in Computing.\n" );

    if ( ! GetFileOperator ( &operator, fin, fout ))
    {
        fprintf ( fout, "Aborting...\n" );
        return ( EXIT_FAILURE );
    }
    else
    if ( operator != 'C')
    {
        fprintf ( fout, "Aborting...\n" );
        return ( EXIT_FAILURE );
    }
    else    /* next operation was 'clear' */
    {
        TermInit ( term1 );
        TermInit ( term2 );
        TermInit ( solution );
        continue;
    }
}
else
{
    /* if no error, print the solution */

    TermToAscii ( solution, buffer, format );
    fprintf ( fout, "=\n%s\n", buffer );
```

```
                /* get the next operation */

                if ( ! GetFileOperator ( &operator, fin, fout ))
                    break;

                if ( operator == 'C' )  /* clear = reinit */
                {
                    TermInit ( term1 );
                    TermInit ( term2 );
                    TermInit ( solution );
                    continue;
                }
                else                    /* keep going */
                {
                    TermCopy ( term1, solution );
                    TermInit ( term2 );
                    TermInit ( solution );
                    goto continuing;
                }
            }
        }
        fclose ( fin );
        fclose ( fout );
    }
    return ( EXIT_SUCCESS );
}

/*-------------------------------------------------------------
 * Get an operator from the input file and check it for errors;
 * write a copy of it to outfile. Returns 0 on error, else 1.
 *-------------------------------------------------------------*/
int GetFileOperator ( int *operator,
                      FILE * infile, FILE *outfile )
{
    char buffer [ MAX_SIZE + 4 ];
    char *p;

    if ( fgets ( buffer, MAX_SIZE + 4, infile ) == NULL )
        return ( 0 );

    /* replace the CR/LF with a null; check length */

    p = strchr ( buffer, '\n' );
    if ( p == NULL )
    {
        printf ( "Error: invalid operator :\n%s\n", buffer );
        fprintf ( outfile,
                  "Error: invalid operator :\n%s\n", buffer );
        return ( 0 );
    }
    else
        *p = '\0';

    *operator = *buffer;

    if ( *( buffer + 1 ) != '\0' ||
            ( *operator != '+' &&
```

```
            *operator != '-' &&
            *operator != '*' &&
            *operator != '/' &&
            *operator != 'c' &&
            *operator != 'C' ))
    {
        printf ( "Error: Invalid operator %c\n", *operator );
        fprintf ( outfile,
                  "Error: invalid operator :\n%s\n", buffer );
        return ( 0 );
    }

    if ( *operator == 'c' )
        *operator = 'C';

    fprintf ( outfile, "%c\n", *operator );

    return ( 1 );
}

/*-------------------------------------------------------------
 * Get a term from the input file and check it for errors;
 * write a copy of it to outfile. Returns 0 on error, else
 * returns SCIENTIFIC or NORMAL, indicating the number format.
 *-------------------------------------------------------------*/
int GetFileTerm ( struct TermData * t,
                  FILE * infile, FILE *outfile )
{
    char buffer [ MAX_SIZE + 4 ];
    char *p;
    int  format = NORMAL;

    if ( fgets ( buffer, MAX_SIZE + 4, infile ) == NULL )
        return ( 0 );   /* EOF */

    /* replace the CR/LF with a null; check length */

    p = strchr ( buffer, '\n' );
    if ( p == NULL )
    {
        printf ( "Error: term too long:\n%s\n", buffer );
        fprintf ( outfile,
                  "Error: term too long:\n%s\n", buffer );
        return ( 0 );
    }
    else
        *p = '\0';

    /* convert to term and check length and errors */

    format = AsciiToTerm ( buffer, t );
    if ( t->sign == 0 ||
         ( t->places_before + t->places_after > MAX_SIZE ))
    {
        printf ( "Error: term too long or invalid:\n%s\n",
                  buffer );
```

```
        fprintf ( outfile,
                  "Error: term too long or invalid:\n%s\n",
                   buffer );
        return ( 0 );
    }

    fprintf ( outfile, "%s\n", buffer );
    return ( format );
}
```

8.2 表示数字

需要以某种方式存储算术运算中使用的数字（称为项），使得可以被对它们执行计算的所有例程访问。在本章中，将在如下形式的结构中表示它们：

```
struct TermData {
       char *term;
       int  sign;
       int  digits_before;
       int  digits_after;
}
```

字段 term 是一个字符数组，其中每个字符都包含一位数字。数组的大小是 2 * MAX_SIZE + 1，其中 MAX_SIZE 是一个预处理器常量，用于规定项的最大值。换句话说，MAX_SIZE 表示将在算术运算中使用的精度。如果想更改这个精度，可以更改这个常量的值，并重新编译例程（可以在程序清单 8-20 中的 longmath. h 中找到 TermData 和 MAX_SIZE）。

建立 term 的内部表示，使得小数点后面的第一位数字（如果有的话）位于数组的中点（即位于 term［MAX_SIZE］中）。这允许小数点两边存在 MAX_SIZE 位数字。这种格式只用于数字的内部表示，注意到这一点很重要。任何计算结果的外部表示都受限于总位数 MAX_SIZE。如果必要，可截去低阶位（在小数点后）。如果结果仍然超过最大位数，就会发生溢出。如果溢出的是有效位（小数点左边的位），溢出就会导致错误。如果溢出发生在小数点右边的位中，就会发生舍入，并且截去多余的位。丢弃右边超过最大指定范围（通过 MAX_SIZE 设定）的位看起来似乎有些浪费，但是保留它们将导致不精确的结果。例如，在一些操作中，有可能保留 2 * MAX_SIZE 个位，而在其他计算中可能连一个位也不保留。而且，这些例程基于大多数计算器提供的模型，它们具有固定大小的窗口。当右边发生溢出时，它们会显示一个溢出错误；当左边发生溢出时，它们会执行舍入并截去一些位。

我们选择这种表示项的方法，使得可以利用最大的精度存储中间结果。在小数点两边各有 MAX_SIZE 个位的内部表示允许我们随时方便地知道小数点位于什么位置。

在结构 TermData 中，将整数 sign 设置为 +1 或 -1，这取决于项的符号。字段 digits_before 和 digits_after 分别用于指定小数点前面和后面的位数。当 MAX_SIZE 等于 5 时，包含值 12.34 的项将存储为图 8-3 所示的形式。在该图中，DEC_LOC 指向小数点后面的第一位。

为了以 TermData 使用的格式放置一个数字，可以调用函数 AsciiToTerm()，传递给它一个字符串，其中包含要转换的数字。函数 AsciiToTerm() 期望传递给它的字符串将使用以下两种格式之一：正常格式和科学记数法。正常格式是大多数人通常书写数字的方式。在正常格式中，项具有

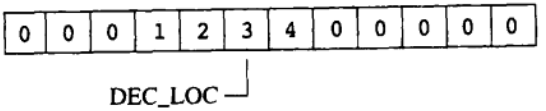


图 8-3 在 MAX_SIZE = 5 的项中存储的 12.34 的规范化视图

一个可选的前导符号（不认可尾随符号）。如果数字在小数点左边没有位，可以选择前置一个前导 0。在用于指定正则表达式的语法中，项必须使用以下格式，以便被函数 AsciiToTerm( ) 接受：

```
[ + | - ]?[0-9] * [. [0-9] + ]
```

其中 * 指示前面加括号的表达式出现了 0 次或多次；+ 指示出现了一次或多次，? 则指示出现了 0 次或 1 次。

大多数 C 程序员都熟悉科学记数法，它是用于表示浮点数的格式。这些数字包含三个部分：**尾数**（mantissa）、字母“e”或“E”（它充当分隔符）和**指数**（exponent）。尾数和指数由一位或多位数字组成，可以选择前置 + 或 - 。只有尾数可以使用小数点。在我们的实现中，如果尾数中有小数点，那么它前面最多只能有一位数字。提供给计算器的数字可以使用科学记数法或正常的记数法。输入的最后一项使用的记数法确定了结构的格式。

如果 AsciiToTerm( ) 遇到一个以科学记数法表示的数字，它就会返回明示常量 SCIENTIFIC；否则，它将会返回 NORMAL。如果出现错误，它就会返回 0，并且把 TermData 中的 sign 变量设置为 0。由于所有的函数都会处理 TermData 数据结构，常识建议我们在结构自身中包括一个错误代码，使得函数可以独立地确定项的有效性，而无需依靠在函数之间传递的标志或错误代码。由于 sign 的合法值只有 +1 或 -1，因此 0 将产生有效的错误标志。在程序清单 8-2 中，AsciiToTerm( ) 将为算术函数对项进行规范化。

程序清单 8-2 函数 AsciiToTerm( ) 为算术函数对项进行规范化

```
/*-------------------------------------------------------------
 * Converts a null-terminated ASCII number in either normal or
 * scientific notation to a term.
 *
 * If the first character is a sign or a decimal point, we
 * process it and replace it with a 0, so as to enable
 * processing of a string of digits. However, to maintain the
 * integrity of the ASCII string, we save the original first
 * character in first_char and restore it before returning.
 * This is of use only when the calculator is reading input
 * from a file.
 *
 * Returns NORMAL, SCIENTIFIC depending on format of string,
 *  or returns 0 on error.
 *-------------------------------------------------------------*/
int AsciiToTerm ( char *buffer, struct TermData *t )
{
    char * dec_pt,
         * p;
    int    i, exponent, len, notation;
    char * new_term;
    char   first_char;
```

```
    char * exp;          /* where the exponent flag is */

    new_term = t->term;   /* point new_term to where it'll go */

    /* is it scientific notation? */

    exp = strpbrk ( buffer, "eE" );
    notation = ( exp == NULL ? NORMAL : SCIENTIFIC );

    first_char = *buffer;

    if ( isdigit ( *buffer ))        /* get the sign */
        t->sign = +1;
    else
    {
        if ( *buffer == '-' )
            t->sign = -1;
        else
        if ( *buffer == '+' )
            t->sign = +1;
        else
        if ( *buffer != '.' )
        {
            t->sign = 0;     /* flag error */
            return ( 0 );
        }

        /* replace any leading sign by a zero */

        if ( *buffer != '.' )
            *buffer = '0';
    }
dec_pt = strchr ( buffer, '.' );
len = strlen ( buffer );

/*
 * load the digits after the decimal point. The first
 * digit goes at term[DEC_LOC], subsequent digits
 * go to the right up to MAX_SIZE digits.
 */

if ( dec_pt != NULL )  /* Only if there's a decimal point */
{
    p = dec_pt + 1;
    for ( i = DEC_LOC; *p && i != 2*MAX_SIZE; i++, p++ )
    {
        if ( ! isdigit ( *p ))  /* check that it's a digit */
        {
            if ( notation == NORMAL )
            {
                t->sign = 0;     /* if not, show an error */
                break;
            }
            else   /* if it's scientific notation... */
            {
                if ( *p == 'e' || *p == 'E' )
                    break;
```

```
                else
                {
                    t->sign = 0;    /* show an error */
                    break;
                }
            }
        }
        else
            new_term[i] = *p - '0';
    }
}

if ( notation == SCIENTIFIC )
{
    p = exp + 1;
    exponent = atoi ( p );
}

if ( t->sign == 0 )     /* any error so far ? */
    return ( 0 );

/*
 * load the digits before the decimal point. You load at
 * the first place right of buffer[DEC_LOC] and add digits
 * to the left up to MAX_SIZE digits.
 */

if ( dec_pt == NULL )
{
    if ( notation == NORMAL )
        p = buffer + len - 1;
    else
        p = exp - 1;
}
else
    p = dec_pt - 1;


for ( i = DEC_LOC - 1; p >= buffer && i >= 0; i--, p-- )
{
    if ( ! isdigit ( *p ))  /* check that it's a digit */
    {
        t->sign = 0;        /* if not, indicate an error */
        break;
    }
    else
    {
        new_term[i] = *p - '0';
        if ( p == buffer ) /* this test for pointer wrap- */
            break;          /* around on Intel segments   */
    }
}

/*
 * if it's scientific notation, shift the term right or left
 * depending on the exponent. If the exponent is > 0, shift
 * left exponent number of places; if it's < 0, shift right.
```

```
 */
if ( notation == SCIENTIFIC )
{
    if ( exponent > 0 )
    {
        while ( exponent-- )    /* shift left */
        {
            if ( *new_term > 0 )
            {
                printf ( "Error: %s too large\n", buffer );
                return ( 0 );
            }
            memmove ( new_term, new_term + 1,
                                        2*MAX_SIZE - 1 );
            new_term[2*MAX_SIZE - 1] = 0;
        }
    }
    else
    if ( exponent < 0 )         /* shift right */
    {
        int warning = 0;

        while ( exponent++ )
        {
            if ( new_term[2*MAX_SIZE - 1] > 0 )
                warning += 1;

            memmove ( new_term + 1, new_term,
                                        2*MAX_SIZE - 1 );
            *new_term = 0;
        }

        if ( warning )
            printf ( "Low order truncation of % digits.\n",
                        warning );
    }
}       /* note: if exponent = 0, no shift occurs */

/*
 * find out how many places before decimal point:
 * start at new_term[0] and move left until you encounter
 * the first non-zero digit or the decimal point.
 * Minimum places_before = 1.
 */

for ( p = new_term;
      *p == '\0' &&  p < new_term + DEC_LOC; p++ )
          ;       /* just loop to the first non-zero digit */

t->places_before = ( new_term + DEC_LOC ) - p;

/*
 * find out how many places after decimal point:
 * if there was a decimal point, then start at right end
 * of new_term and go left until you encounter the first
 * non-zero digit or the decimal point. If there was no
 * decimal point, then places_after = 0.
 */
```

```
    if ( dec_pt == NULL )
        t->places_after = 0;
    else
    {
        for ( p = new_term + 2*MAX_SIZE;
              *p == '\0' && p >= new_term + DEC_LOC;
              p-- )
                ;   /* just loop to the first non-zero digit */

        t->places_after = ( p - ( new_term + DEC_LOC ) + 1 );
    }

    *buffer = first_char;

    return ( notation );
}
```

你将注意到：在程序清单8-2中，数字存储为它们的实际值，而不是存储为字符。例如，数字0被存储为0x00，而不是“0”（在ASCII字母表中，它将是0x30）。这意味着对数字数组执行的操作不能使用C语言的字符串函数，因为这些函数把0x00视作特殊字符。

你还将注意到：数组中小数点的位置被称为term［DEC_LOC］（用于十进制数位）。以前，我们指出第一个十进制数字位于term［MAX_SIZE］。预处理器常量DEC_LOC被定义为MAX_SIZE的同义字。只是出于可读性的目的才使用DEC_LOC。最后要注意的一点是：数组是在前导数字的左边填充0，而不是在最后一位数字的右边填充0。

一个免费赠送的函数TermToAscii()把规范化的项转换为ASCII字符串，如程序清单8-3所示。它用于把解答打印到屏幕上或者打印到一个文件；前者是通过简单地使用printf()实现的。该函数通过使用与用于输入的相同格式生成规范化的项：如果为负数，就带有前导符号；如果小数点左边没有数字，就带有前导0。它将打印MAX_SIZE的有效数字的最大值。

程序清单8-3 函数TermToAscii()把规范化的项转换为可打印的字符串

```
/*-------------------------------------------------------------
 * Converts a normalized term into an ASCII string
 *-------------------------------------------------------------*/
void TermToAscii ( struct TermData * t, char *ascii,
                   int notation )
{
    char *first,        /* first printing digit */
         *last,         /* last printing digit */
         *output;       /* where the ascii string is built */

    first = t->term;
    output = ascii;

    /* skip leading zeros */

    while ( *first == '\0' && first < t->term + 2*MAX_SIZE )
        first++;

    /* if at end, print answer of 0 and exit */
```

```
    if ( first == t->term + 2*MAX_SIZE )
    {
        *output++ = '0';
        *output = '\0';
        return;
    }
    else    /* if past decimal point, start at dec. pt. */
    {
        if ( first >= t->term + DEC_LOC )
             first  = t->term + DEC_LOC;
    }

    /* find last non-zero digit up to decimal point */

    last = t->term + 2*MAX_SIZE;

    while ( last > first &&
            last >= t->term + DEC_LOC &&
           *last == '\0' )
                last--;

    /* before beginning printing, check the sign */

    if ( t->sign == -1 )
        *output++ = '-';

    /* if we start at the decimal point, print leading 0 */
    if ( first == t->term + DEC_LOC )
        *output++ = '0';

    while ( first <= last )
    {
        if ( first == t->term + DEC_LOC )
            *output++ = '.';

        *output++ = *first + '0';
        first += 1;
    }

    *output = '\0';

   if ( notation == SCIENTIFIC )
        AsciiToScientific ( ascii );

}
```

给该函数传递一个标志，指示将用于显示结果的格式。如果想要的格式是科学记数法，那么就会把字符串传递给 AsciiToScientific( )，它执行最终的处理，把字符串转换成科学记数法（参见程序清单 8-4）。注意：这个函数将会去除尾随的 0，比如 1.6000000e4 将显示为 1.6e4。喜欢保留尾随 0 的用户可以注释掉指定的代码段。

程序清单 8-4　函数 AsciiToScientific( ) 把规范格式的数字转换为科学记数法

```
/*-----------------------------------------------------------
 * Takes a string in normal notation and converts it to
```

```
 * scientific notation of the form:
 *                  [-](0-9).(0-9)*e[+|-](0-9)*
 * The final string overlays the original string. This could be
 * dangerous since the new form is conceivably longer than the
 * original form. However, we know the input string is twice
 * MAX_SIZE, so there should always be enough room to fit. This
 * routine would have to modify this aspect if it were to be
 * used as a generic format conversion function.
 *-------------------------------------------------------------
*/
int AsciiToScientific ( char *ascii )
{
    char * buffer,
         * dec_pt,      /* location of the decimal pt, if any */
         * first_digit; /* where the first non-zero digit is */

    char str_exp[8];    /* will hold exponent string */


    int i, j, ascii_len, exponent;
i = j = exponent = 0;

ascii_len = strlen ( ascii );

buffer = malloc ( ascii_len + 6 );
if ( ascii_len < 1 || buffer == NULL )
    return ( 0 );

/* process the sign */

if ( ! isdigit ( *ascii ))
{
    if ( *ascii == '-' )
        buffer[i++] = ascii[j++];
    else
    if ( *ascii != '+' )
    {
        printf (
            "Invalid number for scientific format\n" );
        return ( 0 );
    }
}

/* now process the digits. First check for decimal point. */

dec_pt = strchr ( ascii, '.' );

/* skip to the first non-zero digit */

while ( ascii[j] == '.' || ascii[j] == '0' )
{
    j +=1;
    if ( ascii[j] == '\0' )    /* end of string */
    {
        printf (
            "Invalid number for scientific format\n" );
        return ( 0 );
```

```
    }
}
first_digit = &ascii[j];

buffer[i++] = ascii[j++];
buffer[i++] = '.';
if ( ! ascii[j] )       /* e.g. 6 = 6.0e1 */
    buffer[i++] = '0';
else
    while ( ascii[j] )
        if ( ascii[j] == '.' )
            j += 1;
        else
            buffer[i++] = ascii[j++];

/* do we have trailing zeros? Trim 1.600e4 to 1.6e4 */

j = i - 1;
while ( j > 1 && buffer[j - 1] != '.' )
{
    if ( buffer[j--] == '0' )
        i -= 1;
    else
        break;
}

/* compute the exponent */

if ( dec_pt )
{
    if ( first_digit > dec_pt ) /* e.g., 0.0065  */
        exponent = ( first_digit - dec_pt ) * -1;
    else                        /* e.g., 650.2   */
        exponent = dec_pt - first_digit - 1;
}
else  /* no decimal point */
{
    exponent = ascii_len - 1;
    if ( ! isdigit ( *ascii ))  /* was there a + or - ? */
        exponent -= 1;
}

/* output the exponent */

buffer[i++] = 'e';

memset ( str_exp, '\0', 8 );
sprintf ( str_exp, "%d", exponent );
strcpy ( buffer + i, str_exp );

/* overlay the original string */

strcpy ( ascii, buffer );
free ( buffer );
return ( 1 );
}
```

要创建项，可以使用 TermCreate( ) 和 TermInit( )，前者用于分配数据结构，后者用于把项初

始化为0（参见程序清单8-5和程序清单8-6）。你将注意到TermInit()把符号初始化为+1。不能使用符号0，因为以前讨论过这个值是一个错误标志。可以在创建时调用TermInit()初始化项，或者用于清理它们，以及把它们重置为0。

程序清单8-5 在程序中，TermCreate()分配项并调用TermInit()

```
/*-------------------------------------------------------------
 * Creates a TermData structure and initializes it to
 * a value of zero. Returns NULL on error.
 *-------------------------------------------------------------*/
struct TermData * TermCreate ( void )
{
    struct TermData *t;

    t = malloc ( sizeof ( struct TermData ));
    if ( t == NULL )
        return ( NULL );

    t->term = (char *) malloc ( 2*MAX_SIZE + 1 );
    TermInit ( t );

    return ( t );
}
```

程序清单8-6 TermInit()把项初始化为0

```
/*-------------------------------------------------------------
 * Initialize a term to zero.
 *-------------------------------------------------------------*/
void TermInit ( struct TermData *t )
{
    t->places_before = 0;
    t->places_after  = 0;
    t->sign          = 1;
    memset ( t->term, '\0', 2*MAX_SIZE + 1 );
}
```

用于处理项的最后一个函数是TermCopy()，它简单地把项从一个数据结构复制到另一个数据结构中，如程序清单8-7所示。

程序清单8-7 函数TermCopy()用于复制项

```
/*-------------------------------------------------------------
 * Copy a term from the second argument to the first.
 * Returns a pointer to the copied string.
 *-------------------------------------------------------------*/
struct TermData * TermCopy ( struct TermData * dest,
                             struct TermData * src )
{
    dest->sign          = src->sign;
    dest->places_before = src->places_before;
    dest->places_after  = src->places_after;

    memcpy ( dest->term, src->term, 2*MAX_SIZE + 1 );
```

```
    return ( dest );
}
```

8.3 计算

函数 ComputeResult( ) 用于监督算术运算，如程序清单 8-8 所示。给它传递 4 个参数：一个包含应该执行的运算的字符（有效的选择有：+、-、* 和/，分别用于加法、减法、乘法和除法），以及三个指向 TermData 结构的指针。前两个结构中包含将对其执行算术运算的项，第三个结构将保存规范化的结果。

程序清单 8-8　函数 ComputeResult( ) 用于监督算术运算

```
/*-------------------------------------------------------------
 * This function handles the signs and invokes the correct
 * arithmetic operation.
 *-------------------------------------------------------------*/
int ComputeResult ( struct TermData *t1, int operator,
              struct TermData *t2, struct TermData *sol )
{
    int cmp;

    TermInit ( sol );   /* just good practice */

    switch ( operator )
    {
        case '+':
      /*
       * addition really occurs only when the signs
       * are the same. If signs differ, the operation
       * is really subtraction. In that case, we call
       * subtraction routines. Before calling
       * NormalSubtract(), we make sure the terms are
       * in the right order. See the comments in the
       * subtraction section below for more info on this.
       */

      if ( t1->sign == t2->sign ) /* a + b or -a + -b */
      {
          if ( ! NormalAdd ( t1, t2, sol ))
          {
              fprintf ( stderr,
                  "Overflow on addition\n" );
              return ( 0 );
          }
          sol->sign = t1->sign;
      }
      else
      if ( t1->sign == -1 )   /* -a + b */
      {
          sol->sign = -1;

          cmp = NormAbsCmp ( t1, t2 );
          if ( cmp < 0 ) /* t2 larger than t1 */
          {
```

```
            sol->sign = -sol->sign;
            NormalSubtract ( t2, t1, sol );
        }
        else
        if ( cmp > 0 ) /* t1 larger than t2 */
            NormalSubtract ( t1, t2, sol );
        else
        {               /*  t1 = t2, so sol = 0 */
            TermInit ( sol );
            return ( 1 );
        }
    }
    else                    /* a + -b */
    {
        sol->sign = +1;

        cmp = NormAbsCmp ( t1, t2 );
        if ( cmp < 0 ) /* t2 larger than t1 */
        {
            sol->sign = -sol->sign;
          NormalSubtract ( t2, t1, sol );
     }
     else
     if ( cmp > 0 ) /* t1 larger than t2 */
         NormalSubtract ( t1, t2, sol );
     else
     {              /*  t1 = t2, so sol = 0 */
         TermInit ( sol );
         return ( 1 );
     }
  }
  break;

case '-':
  /*
   * there are four possible cases for subtraction,
   * and each is treated differently:
   *  a - b       subtract a from b
   *  a - -b      add a and b
   * -a - b       add a and b, multiply by -1
   * -a - -b      subtract a from b, multiply by -1
   *
   * multiplying by -1 is accomplished simply
   * by flipping the sign of the result.
   *
   * Moreover, subtraction is set up so that we always
   * subtract the smaller term from the larger. If t2
   * is larger than t1, however, then we flip the
   * sign of the solution and reverse the terms. This
   * approach works because x - y = -1( y - x ). That
   * is, you can flip the terms in subtraction if you
   * flip the sign of the difference:
   *              3 - 7 = -1 * ( 7 - 3 )
   */

  if ( t1->sign == +1 )
  {
```

```
    if ( t2->sign == +1 )  /* a - b */
    {
        sol->sign = +1;

        cmp = NormAbsCmp ( t1, t2 );
        if ( cmp < 0 ) /* t2 larger than t1 */
        {
            sol->sign = -sol->sign;
            NormalSubtract ( t2, t1, sol );
        }
        else
        if ( cmp > 0 ) /* t1 larger than t2 */
            NormalSubtract ( t1, t2, sol );
        else
        {               /*  t1 = t2, so sol = 0 */
            TermInit ( sol );
            return ( 1 );
        }
    }
    else
    {                       /* a - -b */
        if ( ! NormalAdd ( t1, t2, sol ))
        {
            fprintf ( stderr,
                "Overflow on addition\n" );
            return ( 0 );
        }
        sol->sign = +1;
    }
}
else
{                           /* -a - b */
    if ( t2->sign == +1 )
    {
        if ( ! NormalAdd ( t1, t2, sol ))
        {
            fprintf ( stderr,
                "Overflow on addition\n" );
            return ( 0 );
        }
        sol->sign = -1;
    }
    else                    /* -a - -b */
    {
        sol->sign = -1;

        cmp = NormAbsCmp ( t1, t2 );
        if ( cmp < 0 ) /* t2 larger than t1 */
        {
            sol->sign = -sol->sign;
            NormalSubtract ( t2, t1, sol );
        }
        else
        if ( cmp > 0 ) /* t1 larger than t2 */
            NormalSubtract ( t1, t2, sol );
        else
        {               /*  t1 = t2, so sol = 0 */
            TermInit ( sol );
            return ( 1 );
```

```
                }
            }
        }
        break;

    case '*':
        /*
         * multiplication sign issues are straightforward.
         * just multiply and set the sign, depending on
         * whether the terms have the same sign.
         */

        if ( ! NormalMultiply ( t1, t2, sol ))
        {
            fprintf ( stderr, "Overflow on mulitply\n" );
            return ( 0 );
        }
        if ( t1->sign == t2->sign )
            sol->sign = +1;
        else
            sol->sign = -1;
        break;


    case '/':
        /* likewise for division */

        if ( ! NormalDivide ( t1, t2, sol ))
            return ( 0 );

        if ( t1->sign == t2->sign )
            sol->sign = +1;
        else
            sol->sign = -1;
        break;

    default:
        fprintf ( stderr, "Unsupported operation %c\n",
                  operator );
        return ( 0 );
    }

    return ( 1 );
}
```

如以前所解释的，函数 ComputeResult( ) 规范化两个算术项，并通过检查项的数据结构中的符号字段来检查项中的错误。然后，该函数将初始化将要保存解答的项。最后，将处理字符 operation，它指示应该执行算术函数。注意：这个过程需要对项的符号执行一些操作。不久就会清楚为什么需要这样做。

8.4 加法

加法通常被认为是那些在数学上可以通过加号表示的运算。例如，其中 a 和 b 是正数，a + b 显然就是加法运算。不过，对于 a + (- b)，就不太容易看出这是一个加法运算。从技术上讲，

它仍然是加法运算，即使它是通过使用减法（a－b）实现的。加法和减法之间的这种转换是在ComputeResult( ) 中处理的。当该函数检测到要求执行加法运算时，它会检查符号，并且仅当两个项具有相同的符号时才会调用加法函数 NormalAdd( )（以对规范化的项执行加法运算而得名）；否则，它会把运算传递给减法函数 NormalSubtract( )。函数 ComputeResult( ) 依赖于相加的项全为正或者全为负，来设置结果（称为和（sum））的符号。

如程序清单 8-9 中所示，NormalAdd( ) 中的加法运算很直观。给该函数传递两个规范化并且小数点对齐的项。函数从最右边的有效位开始把两个数相加在一起。如果小数点右边没有有效位，加法过程将从小数点左边的第一位开始。

程序清单 8-9　函数 NormalAdd( ) 把两个规范化的项相加

```
/*-------------------------------------------------------------
 * We start at the rightmost digit of the terms, but no farther
 * left than the decimal point. We work our way left, adding as
 * we go until we reach the leftmost digit, and then go one digit
 * farther, in case of any carry.
 *-------------------------------------------------------------*/
int NormalAdd ( struct TermData *t1, struct TermData *t2,
                struct TermData *sum )
{
    int i, j, start, stop;

    start = DEC_LOC +
            max ( t1->places_after, t2->places_after );
    stop  = DEC_LOC - 1 -
            max ( t1->places_before, t2->places_before );
for ( i = start; i >= stop; i-- )
{
    sum->term[i] += ( t1->term[i] + t2->term[i] );
    if ( sum->term[i] > 9 )
    {
        sum->term[i] -= 10;
        sum->term[i-1] += 1;
    }
}

sum->places_after = start - DEC_LOC;

for ( i = 0; i < DEC_LOC; i++ )
      if ( (sum->term)[i] != 0 )
        break;
sum->places_before = DEC_LOC - i;

/* make sure that the sum is within MAX_SIZE digits */

i = sum->places_before + sum->places_after - MAX_SIZE;

if ( i > 0 )    /* sum is larger than MAX_SIZE */
{
    int carry = 0;

    if ( i > sum->places_after )
        return ( 0 );   /* high-order truncation will occur*/
```

```
    sum->places_after -= i;     /* adjust sum */

    j = DEC_LOC + sum->places_after;     /* do rounding */
    if ( (sum->term)[j] > 4 )
        carry = 1;

    if ( carry )
        while ( 1 )
        {
            j--;

            if ( j == 0 && (sum->term)[j] > 8 )
                return ( 0 );

            (sum->term)[j] += carry;

            if ( (sum->term)[j] > 9 )
                (sum->term)[j] -= 10;
            else
                break;
        }
        while ( i-- )        /* do low-order truncation */
        {
            (sum->term)[DEC_LOC +
                    sum->places_after + i ] = 0;
        }
    }

    return ( 1 );
}
```

8.5 减法

与加法一样，无论何时需要执行减法运算，函数 ComputeResult( ) 都会检查项的符号，以确保确实需要它。像 a - (- b) 这样的运算将提交给 NormalAdd( ) 执行加法运算。在减法中，仅当项具有相似的符号（a - b 和 - a - (- b)）时，才会由函数 NormalSubtract( ) 处理。

函数 NormalSubtract( ) 受益于简单的代数等式：a - b = - 1 * (b - a)。这个规则指示我们可以交换减法中项的次序，只要把结果乘以 - 1 即可。由于乘以 - 1 就等同于切换符号，可以将该规则简述如下：可以交换减法中的项，只要交换结果的符号即可（减法的结果称为差（difference））。可以按手动执行减法的方式直观地理解这个规则：如果要求从 11 中减去 25，我们实际上会从 25 中减去 11，然后交换差的符号，这将会得到 - 14。由于从绝对值较大的数中减去绝对值较小的数要容易得多，NormalSubtract( ) 也会应用这个相同的原理。在调用 NormalSubtract( ) 之前，ComputeResult( ) 将会检查两个项的绝对值，以查看是否需要交换它们。它通过调用 NormAbsCmp( ) 并给该函数传递指向两个 TermData 结构的指针来执行该任务，如程序清单 8-10 所示。NormAbsCmp( ) 返回与标准 ANSI C 库中的 strcmp( ) 相同的值。

程序清单 8-10 函数 NormAbsCmp( ) 比较两个项的绝对值，并以与 strcmp( ) 相同的方式返回值

```
/*-------------------------------------------------------------
 * NormAbsCmp() works similar to strcmp(). The return value is
```

```
 * < 0, = 0, > 0 depending on whether the *absolute value* of
 * the first term is less than, equal to, or greater than the
 * second term, respectively.
 *-------------------------------------------------------------*/
int NormAbsCmp ( struct TermData * t1, struct TermData * t2 )
{
    int memcmp_result;

    /* first check the digits before the decimal point */

    if ( t1->places_before > t2->places_before )
        return ( 1 );

    if ( t2->places_before > t1->places_before )
        return ( -1 );

    /*
     * same number of digits before decimal point,
     * so compare character by character
     */

    memcmp_result =
         memcmp ( t1->term + DEC_LOC - t1->places_before,
                  t2->term + DEC_LOC - t2->places_before,
                  t1->places_before +
                    max ( t1->places_after, t2->places_after ));

    if ( memcmp_result > 0 )
        return ( 1 );
    else
    if ( memcmp_result < 0 )
        return ( -1 );
    else
        return ( 0 );
}
```

函数 NormAbsCmp( ) 通过检查每个项有多少位前导数字来确定哪个项更大（前导位出现在小数点左边）。如果两个项具有相同数量的前导位，函数就会逐位比较这两个项，直至它找到一个更大的位。

一旦按大小正确地建立了项，NormalSubtract( ) 中的循环从最右边的位开始遍历项，并执行减法运算。与加法一样，如果小数点右边没有数字位，减法就从小数点左边的第一位开始。

一个有趣的细节是如何执行进位。传统上讲，在美国，如果必须从较少的数字中减去较大的数字，可以按下面这种方式进行：

```
  234     变为     1 12 14
- 89              -  8  9
  145               1  4  5
```

从上面一项（**被减数**（minuend））的 4 中减去底下一项（从技术上讲，这个项被称为**减数**（subtrahend））最右边的 9，将 4 扩展为 14，并递减被减数中左边的下一位（3），指示将 10 加到 4 上。重复这个过程以减去 8：将 2（被减数中过去 3 所在的位置）扩展为 12，并将前导 2 减为 1。这将生成示例右边所示的减法。

NormalSubtract( ) 中使用的方法是在世界上的其他地区使用的方法。当需要时，这种系统不会从被减数左边的位中减去 1，而是给减数添加 1 位。上面的示例将如下所示：

```
  234     变为      2 13 14
= 89              = 1  9  9
  145               1  4  5
```

当最初从 4 中减去 9 时，通常将 4 扩展为 14。但是，我们将不会递减被减数中的 3，而是递增减数中的 8，这将得到 9。然后，我们首先把 3 扩展为 13，并从中减去这个 9。将 3 扩展为 13 导致我们会重复以前的步骤：递增减数中的下一位。在这里，将百位从 0 递增为 1。从被减数中的 2 中减去这个 1，这是减法中的最后一步。注意：这两种方法都会得到完全相同的结果。程序清单 8-11 显示了一个示例程序，其中 NormalSubtract( ) 使用第二种方法减去两个规范化的项。

程序清单 8-11　在程序中，NormalSubtract( ) 减去两个规范化的项

```
/*-------------------------------------------------------------
 * subtraction: We first determine where to start the process
 * of subtraction. It is the rightmost digit of the two terms,
 * that is to the right of the decimal point, else it is
 * at the first digit left of the decimal point. We then
 * proceed from that digit (the least significant digit) and
 * move to the left until we reach the leftmost digit of the
 * two terms. Because of the possibility of a carry, we go one
 * digit more to the left. This duplicates manual subtraction.
 *-------------------------------------------------------------*/
int NormalSubtract ( struct TermData * t1, struct TermData * t2,
                     struct TermData * diff )
{
    /*
     * The result of a subtraction is called a difference,
 * hence we named the variable containing the answer diff.
 */
int carry, i, j;
int max_after, max_before;
char *p1, *p2, *pd;

/*
 * we'll copy the terms to scratch since we'll be altering
 * them with any carries and borrows that we do.
 */
char scratch1 [2*MAX_SIZE + 1],
     scratch2 [2*MAX_SIZE + 2];

memcpy ( scratch1, t1->term, 2*MAX_SIZE + 1 );
memcpy ( scratch2, t2->term, 2*MAX_SIZE + 1 );

/* where to start subtracting? */

max_before = max( t1->places_before, t2->places_before );
max_after  = max( t1->places_after,  t2->places_after );

j =  max_before + max_after;

/*
```

```
 * at worst, the rightmost digit is the last digit left of
 * the decimal point. So, start no further left than there.
 */

p1 = scratch1   + DEC_LOC + max_after - 1;
p2 = scratch2   + DEC_LOC + max_after - 1;
pd = diff->term + DEC_LOC + max_after - 1;

while ( j >= 0 )
{
    /*
     * if there is a carry, borrow 10 and
     * add 1 to the next higher digit in t2
     */

    if ( *p2 > *p1 )
    {
        *p1  += 10;
        *( p2 - 1 ) += 1;
    }

    *pd = *p1 - *p2;  /* the actual subtraction */
    pd--; p1--; p2--; /* move to the next higher digit */
    j--;
}

for ( j = 0, pd = diff->term;
      pd < diff->term + DEC_LOC; j++, pd++ )
        if ( *pd != 0 )
            break;

diff->places_before = DEC_LOC - j;

/*
 * to get the number of places after, start at max_after,
 * the maximum number of places after, and move left
 * until the first nonzero digit right of the decimal pt.
 */

pd = diff->term + DEC_LOC + max_after - 1;
while ( pd >= diff->term + DEC_LOC )
    if ( *pd == 0 )
    {
        max_after -= 1;
        pd -= 1;
    }
    else
        break;

diff->places_after  = max_after;

/*
 * There can be too many digits in the result: subtract
 * MAX_SIZE digits right of the decimal pt from MAX_SIZE
 * digits left of the decimal point. If this occurs, we
 * round and truncate.
 */
```

```
i = diff->places_before + diff->places_after - MAX_SIZE;

if ( i > 0 )    /* overflow */
{
    diff->places_after = MAX_SIZE - diff->places_before;
    i = DEC_LOC + diff->places_after;
    if ( (diff->term)[i] > 4 )  /* round up if > 4 */
        carry = 1;
    else
        carry = 0;

    j = i;
       i -= 1;

       /* add carry (the rounding) */

       while ( carry )
       {
           if ( i == 0 && (diff->term)[i] > 8 )
           {
               printf ( "Overflow on subtraction\n" );
               return ( 0 );
           }

           (diff->term)[i] += carry;

           if ( (diff->term)[i] > 9 )
           {
               (diff->term)[i] -= 10;
               carry = 1;
               i--;
           }
           else
               carry = 0;
       }

       /* now zero out the digits we have truncated */

       while ( j < 2*MAX_SIZE )
           (diff->term)[j++] = 0;
   }

   return ( 1 );
}
```

人们特别喜爱执行减法运算的第二种方法，因为它不需要特殊的处理。第一种方法确实具有一种特殊情况：当被减数中左边的下一位是0时，就要减去进位。例如，假设你想从201中减去99。当把1扩展为11时，你将发现不能简单地递减0。作为替代，必须测试并用9替换0，并且必须把进位传播给左边的下一位。如果下一位是0，就必须重复这个过程，并再次把进位传播给左边的下一位。如果左边的位包含一串0，那么在可以执行任何进一步的减法之前，必须为其中每个0执行进位和传播。使用第二种方法就不会发生这种问题。通过递增减数，把进位的处理限制到左边的下一位。甚至在最坏情况下，其中通过进位把9递增为10，这个10仍然可以等待，直至轮到减去它，而无需立即在数字中一直向上传播进位。两种方法之间的这个差别使得第二种方法

在C语言的代码中要容易得多。

8.6 乘法

像加法和减法一样，乘法也是一种直观的运算。在计算实际开始之前，ComputeResult( ) 将会检查两个项的符号。如果它们相同，乘法的结果（称为积）就为正；否则，结果就为负。然后把两个项传递给 NormalMultiply( )，并在其中执行乘法运算。

为了查看如何执行乘法运算，我们需要检查乘法问题，就像手动执行它一样。下面给出了一个示范性的例子，我们希望它看起来很熟悉：

```
     3 1 8 7   被乘数
   x   4 6 9   乘数
   2 8 6 8 3   中间积
 1 9 1 2 2 ·
1 2 7 4 8 · ·
-------------
1 4 9 4 7 0 3   积
```

对于乘数中的每一位，将生成单独的中间积。为了把执行乘法和加法的次数减至最少（将为乘数中的每一位遍历一次被乘数），我们将使两个项中较短的项作为乘数。

然后，乘法将为乘数中的每一位遍历被乘数中的每一位。该运算是手动执行乘法运算的相当好的副本，只有两个细节例外：将在生成中间积时求它们的总和，并且各个位的乘法是在一个查找表中执行的。

出于速度的缘故而执行了后一种修改。如果我们要模拟手动计算，将不得不把乘数中的每一位与被乘数相乘，并且将不得不对结果应用求模函数（它是作为除法执行的），以便确定将哪一位放在中间积中。由于只有一千种可能的位组合，它们的积将存储在一个整数表中，并且可以简单地进行查找而不是计算。注意表（在代码中经过深思熟虑将其命名为 table [ ]）的格式；例如，4×7 的结果不是 28，而是 0x0208。实质上，表项是利用两个字节加载的：一个字节包含低阶位（8）；另一个字节包含高阶位（2）。如果在表中将结果输入为 28，就需要执行单独的操作将 8（它被加到中间结果上）与 2（或 20）隔开，后者会保留下去。

表自身的结构需要做一点解释，因为这个表被设计成在访问它时不需要执行乘法运算（毕竟，这是开始创建这个表的目的）。最实用的方法是构建一个二维表。然后，我们将访问 table [4，7] 来确定 4×7 的积。不过，通常把二维表的下标转变为可以被乘法访问的指针地址。为了避免这个乘法，将表设计为一个一维表。其下标的计算方法是：10 * 第一个数字再加上第二个数字。因此，以前的访问将被计算为 10 * 4 + 7 或 table [47]。这里的值与 table [74] 的值相同，即 0x0208。你可能注意到这里是使用乘法来确定这个下标。但是，事实上，这个下标的第一部分的计算（4 * 10）是通过查找包含 10 个条目的小表（名为 mults_of_10）来完成的。因此，利用这两个表（它们占据 110 个整数的空间），消除了为相乘的每一位执行乘法（和除法）运算的需要。

关于乘法要注意的一点是：与其他函数不同，NormalMultiply( ) 使用 4 * MAX_SIZE（而不是 2 * MAX_SIZE）个缓冲区。这允许小数点后面有 2 * MAX_SIZE 个位。这种变化是由像 0.001 * 0.999 这样的乘法引起的。假设 MAX_SIZE 是 3，如果内部小数点后面没有 2 * MAX_SIZE 个位，就不能执行这个乘法运算，因为结果的最右边的位将放在小数点右边的 6 个位置中。虽然出现在积中的可能最大位数只是 MAX_SIZE，但是我们需要计算积的完整长度，以便在截去多余的

位之前正确地进行舍入。在我们的示例中，0.001 × 0.999 给出了 0.000999 的内部表示。由于 MAX_SIZE 是 3，舍入和截去操作将把它转变为 0.001。注意：如果小数点后面没有 2 * MAX_SIZE 个位，舍入将会不正确，并且结果将为 0。在程序清单 8-12 中，NormalMultiply( ) 执行两个规范化的项的乘法运算。

程序清单 8-12　函数 NormalMultiply( ) 执行两个规范化的项的乘法运算

```
/*-------------------------------------------------------------
 *  Multiplication of normalized terms.
 *  In the expression c = a * b, a is called the multiplicand,
 *  b the multiplier, and c the product.
 *-------------------------------------------------------------*/
int NormalMultiply ( struct TermData * t1,
            struct TermData * t2, struct TermData * prod )
{
    /*
     * Each digit of the multiplier will require the generation
     * of an intermediate result, which is added to previous
     * intermediate results, to produce the product. Hence,
     * we'll make the multiplier the shorter of the two terms.
     */

    char *mcand,     /* the multiplicand */
         *mier,      /* the multiplier   */
         *temp;      /* temporary hold area for product */

    int  mcand_curr, /* where we are in the muliplicand */
         mier_curr,  /* where we are in the multiplier */
         temp_curr,  /* where we are in the temp product */
         temp_here;

    int  mcand_len,  /* number of digits in multiplicand */
         mier_len;   /* number of digits in multiplier */

    int  carry;      /* the carry digit when adding the
                        intermediate results */
    int  i, j, from, to;

    static int table [100] =
    {
     /*         0*0    0*1    0*2    0*3    0*4  */
     /* 0 */ 0x0000,0x0000,0x0000,0x0000,0x0000,
     /*         0*5    0*6    0*7    0*8    0*9  */
            0x0000,0x0000,0x0000,0x0000,0x0000,
     /*         1*0    1*1    1*2    1*3    1*4  */
     /* 1 */ 0x0000,0x0001,0x0002,0x0003,0x0004,
     /*         1*5    1*6    1*7    1*8    1*9  */
            0x0005,0x0006,0x0007,0x0008,0x0009,
     /*         2*0    2*1    2*2    2*3    2*4  */
     /* 2 */ 0x0000,0x0002,0x0004,0x0006,0x0008,
     /*         2*5    2*6    2*7    2*8    2*9  */
            0x0100,0x0102,0x0104,0x0106,0x0108,
     /*         3*0    3*1    3*2    3*3    3*4  */
     /* 3 */ 0x0000,0x0003,0x0006,0x0009,0x0102,
     /*         3*5    3*6    3*7    3*8    3*9  */
            0x0105,0x0108,0x0201,0x0204,0x0207,
```

```
    /*          4*0    4*1    4*2    4*3    4*4  */
    /* 4 */ 0x0000,0x0004,0x0008,0x0102,0x0106,
    /*          4*5    4*6    4*7    4*8    4*9  */
            0x0200,0x0204,0x0208,0x0302,0x0306,
 /*         5*0    5*1    5*2    5*3    5*4  */
 /* 5 */ 0x0000,0x0005,0x0100,0x0105,0x0200,
 /*         5*5    5*6    5*7    5*8    5*9  */
        0x0205,0x0300,0x0305,0x0400,0x0405,
 /*         6*0    6*1    6*2    6*3    6*4  */
 /* 6 */ 0x0000,0x0006,0x0102,0x0108,0x0204,
 /*         6*5    6*6    6*7    6*8    6*9  */
        0x0300,0x0306,0x0402,0x0408,0x0504,
 /*         7*0    7*1    7*2    7*3    7*4  */
 /* 7 */ 0x0000,0x0007,0x0104,0x0201,0x0208,
 /*         7*5    7*6    7*7    7*8    7*9  */
        0x0305,0x0402,0x0409,0x0506,0x0603,
 /*         8*0    8*1    8*2    8*3    8*4  */
 /* 8 */ 0x0000,0x0008,0x0106,0x0204,0x0208,
 /*         8*5    8*6    8*7    8*8    8*9  */
        0x0400,0x0408,0x0506,0x0604,0x0702,
 /*         9*0    9*1    9*2    9*3    9*4  */
 /* 9 */ 0x0000,0x0009,0x0108,0x0207,0x0306,
 /*         9*5    9*6    9*7    9*8    9*9  */
        0x0405,0x0504,0x0603,0x0702,0x0801
};

static int mults_of_ten [10] =
    { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90 };

mcand_len = mier_len = 0;

/*
 * products can overflow a term, since the number of
 * digits left of the decimal pt in the product is the
 * sum of the digits left of the decimal pt in the
 * multiplier and multiplicand - 1. Likewise, the number
 * of digits right of the decimal point is the sum of
 * these digits in the multiplier and multiplicand.
 *
 * Hence, our internal representation of the product,
 * temp, has 2*MAX_SIZE digits on either side of the
 * decimal point. We will round and truncate as necessary
 * when we load the digits into prod.
 */

temp = calloc ( 1, 4*MAX_SIZE + 1 );
if ( temp == NULL )
{
    printf ( "Out of memory in multiplication.\n" );
    return ( 0 );
}
/*
 * the following test avoids guaranteed high-order
 * truncation and saves having to do the multiplication
 * only to discover the overflow at end.
 */
```

```
if ( t1->places_before + t2->places_before - 1 >
     MAX_SIZE )
{
    free ( temp );
    return ( 0 );
}

if (( t1->places_before + t1->places_after ) >
( t2->places_before + t2->places_after ))
{
    mcand = t1->term;
    mier  = t2->term;

    mcand_len = t1->places_before + t1->places_after;
    mier_len  = t2->places_before + t2->places_after;

    mcand_curr = DEC_LOC + t1->places_after - 1;
    mier_curr  = DEC_LOC + t2->places_after - 1;
}
else
{
    mcand = t2->term;
    mier  = t1->term;

    mcand_len = t2->places_before + t2->places_after;
    mier_len  = t1->places_before + t1->places_after;

    mcand_curr = DEC_LOC + t2->places_after - 1;
    mier_curr  = DEC_LOC + t1->places_after - 1;
}

/*
 * The number of digits after the decimal points in a
 * product is the sum of the number of decimal digits
 * in  each term: 12.6 * 1.2 = 15.12.
 * Hence, we start putting digits into prod, using the
 * following formulation:
 */

temp_curr = 2*DEC_LOC +
            t1->places_after + t2->places_after - 1;
carry = 0;
while ( mier_len > 0 )  /* the multiplication loop */
{                       /* for each digit of multiplier */
    int j, a, b, val;

    i = mcand_len;
    j = mcand_curr;
    temp_here = temp_curr;

    while ( i >= 0 )     /* process the whole multiplicand */
    {
        if ( mier [ mier_curr ] == 0 )
            break;

        a = mier [ mier_curr ];
        b = mcand [ j ];
```

```
        a += mults_of_ten [ b ];

        val = table [ a ];
        temp[ temp_here ] += carry + val & 0x00FF;
        carry = val >> 8;
        if ( temp[temp_here] > 9 )
        {
            carry += temp[temp_here] / 10;
            temp[ temp_here ] %= 10;
        }
        j--;          /* move up the multiplicand */
        i--;          /* one less iteration to do */
        temp_here--; /* move one product digit to the left */
    }
    mier_curr--;    /* move up the multiplier */
    mier_len--;     /* one less multiplier digit */
    temp_curr--;    /* move up the solution by one digit */
}

if ( carry > 0 )
    temp[temp_curr] = carry;

for ( i = 0; i < 2*DEC_LOC; i++ )
      if ( temp[i] != 0 )
        break;

/* did we overflow anyway? */

if (( 2*DEC_LOC - i ) > MAX_SIZE )
{
    free ( temp );
    return ( 0 );
}
else
    prod->places_before = 2*DEC_LOC - i;

/* copy the digits before the dec pt from temp to prod */

from = 2*DEC_LOC - 1;
to   = DEC_LOC - 1;

i = prod->places_before;
while ( i-- )
    (prod->term)[to--] = temp[from--];

/*
 * now examine the digits after the decimal point
 * and perform rounding and truncation as necessary
 */

for ( i = 4*MAX_SIZE - 1; i >= 2*DEC_LOC; i-- )
      if ( temp[i] != 0 )
        break;

prod->places_after = i - 2*DEC_LOC + 1;

/* j = maximum places after */
```

```
j = MAX_SIZE - prod->places_before;

/* do we have to round and truncate? */

if ( j < prod->places_after )
{
    prod->places_after = j;
    if ( temp[2*DEC_LOC + j] > 4 )  /* we round up if > 4 */
        carry = 1;
    else
        carry = 0;

    /* copy the digits over */

    from = 2*DEC_LOC + j - 1;
    to   = DEC_LOC + j - 1;

    while ( j-- )
        (prod->term)[to--] = temp[from--];

    /* now do the rounding */
       if ( carry )
       {
           i = DEC_LOC + prod->places_after - 1;
           while ( 1 )
           {
               /*
                * we now add carry (the rounding). If
                * the current digit is 9, the carry will
                * generate a 10, meaning that we have to
                * carry to the next digit left. If this
                * occurs at the leftmost digit, it can
                * cause an overflow, so this possibility
                * is checked for first.
                */

               if ( i == 0 &&              /* overflow */
                    (prod->term)[i] > 8 )
               {
                   free ( temp );
                   return ( 0 );
               }

               (prod->term)[i] += carry;

               if ( (prod->term)[i] > 9 )
               {
                   (prod->term)[i] -= 10;
                   i--;
               }
               else
                   break;
           }
       }
    }
    else    /* no truncation so just copy the digits */
    {
```

```
        j = prod->places_after;

        from = 2*DEC_LOC + j - 1;
        to   = DEC_LOC + j - 1;

        while ( j-- )
            (prod->term)[to--] = temp[from--];
    }

    free ( temp );
    return ( 1 );
}
```

8.7 除法

除法是4种基本的算术运算中最困难的运算。它需要做比其他运算多得多的工作，这些工作本质上要复杂得多。在这里展示的计算器中，用于除法的代码是用于乘法的代码的三倍多，而后者又是用于减法的代码的两倍。在检查这些复杂性之前，我们需要检查除法自身的过程。我们使用现代美国学校讲授的方法，如图8-4所示。

```
          3 1 7
      ┌─────────
2 8 1 │ 8 9 0 7 7
        8 4 3
        ─────
          4 7 7
          2 8 1
          ─────
          1 9 6 7
          1 9 6 7
          ───────
                0
```

图8-4 美国学校中讲授的手动除法运算

在该图中，281是**除数**（divisor），89077是**被除数**（dividend），317是**商**（quotient）。843、477和1967这些数字都是中间被除数。这种图可能看起来很熟悉，或者你至少可以向其中移入一种不同的系统。

多种因素造成了除法的复杂性：

- 与其他运算不同，除法需要猜测。首先猜测商的每一位，然后测试它是太大还是太小。
- 被除法在不断变化。当我们执行除法运算并把数字向下转移时，被除数将变为一系列不同的中间被除数。
- 对于较大的数，不能完整地处理数字。必须代之以取除数的前几位和被除数的前几位，以便猜测商的下一位。

除法的这三个方面使得它是一种动态运算；也就是说，项的值一直在变化。因此，我们将花费相当多的工作来记录在除法运算过程中所处的位置。为了最小化记录各个项、它们的长度及其值的难度，除法函数NormalDivide()不会把项存储为一系列单独的位，而是将其存储为真正的字符串，例如，将0存储为‘0’（0x30），而不是0（0x00）。这种方法虽然会带来开销，但它允许C语言的字符串函数轻松地操纵项。它还使得代码要简单得多。参考程序清单8-13。

程序清单8-13 函数NormalDivide()执行两个规范化的项的除法运算

```
/*-------------------------------------------------------------
 * Division is accomplished by turning the term data into an
 * array of characters rather than bytes holding the numeric
 * value of each digit--the latter format is used in the other
 * arithmetic operations. However, in division, it becomes a
 * cumbersome format. (See text.)
 *-------------------------------------------------------------*/
int NormalDivide  ( struct TermData * dvend_arg,
                    struct TermData * dvsor_arg,
```

```
                    struct TermData * quot )
{
#define ASCII_VAL ( '0' ) /* the difference between 0 and '0' */

    unsigned int
         leading_zeros = 0,
         divend_len,
         divsor_len,
         idivend_len;

    int  strcmp_return,
         strncmp_return;

    char *answer,
         *dividend,
         *divisor,
         *interim,
         *new_interim,
         *product;

    unsigned int
         quo_guess,

         next_dvend_digit = 0,

         answer_idx = 0,

         dvsor_len,
         idvend_len;
int  dvend_order, /* order of magnitude of divisor and   */
     dvsor_order; /* dividend. Used for locating decimal */
                  /* point in quotient.                  */

char *pc;   /* generic variables */
int  i, j;


dividend    = calloc ( 1, 2*MAX_SIZE + 1 );
divisor     = calloc ( 1, MAX_SIZE + 1 );
interim     = calloc ( 1, 2*MAX_SIZE + 1 );
new_interim = calloc ( 1, 2*MAX_SIZE + 1 );
product     = calloc ( 2, 2*MAX_SIZE + 1 );
answer      = calloc ( 2, MAX_SIZE + 1 );

if ( dividend == NULL || divisor == NULL  ||
     interim  == NULL || new_interim == NULL ||
     product  == NULL || answer == NULL )
{
    printf ( "Error allocating memory in division\n" );
    return  ( 0 );
}

/* load dividend with the digits as character values */
i = 0;
for ( pc  = dvend_arg->term +
            DEC_LOC - dvend_arg->places_before;
      pc <= dvend_arg->term +
```

```
            DEC_LOC + dvend_arg->places_after - 1;
      pc++, i++ )
        dividend[i] = *pc + ASCII_VAL;

/* remove leading zeros */

while ( *dividend == '0' )
    DivShiftArrayLeft ( dividend );


/* likewise, for the divisor */
i = 0;
for ( pc  = dvsor_arg->term +
            DEC_LOC - dvsor_arg->places_before;
      pc <= dvsor_arg->term +
            DEC_LOC + dvsor_arg->places_after - 1;
      pc++, i++ )
        divisor[i] = *pc + ASCII_VAL;

while ( *divisor == '0' )
    DivShiftSmallArrayLeft ( divisor );
/* If divisor is zero, signal error and abort */

if ( DivCheckZeroOnly ( divisor ))
{
    printf ( "Error: Division by Zero\n" );
    return ( 0 );
}

/* If dividend is zero, set quotient to zero and exit */

if ( DivCheckZeroOnly ( dividend ))
{
    TermInit ( quot );
    return ( 1 );
}

divend_len = strlen ( dividend );
divsor_len = strlen ( divisor );

/*
 * If dividend is < divisor,
 * add zeros to end of dividend
 * an add leading zeros to quotient
 */

while ( divend_len < divsor_len )
{
    *( dividend + divend_len ) = '0';
    leading_zeros += 1;
    divend_len += 1;
}

if ( divend_len == divsor_len )
{
    strcmp_return = strcmp ( dividend, divisor );
    if ( strcmp_return < 0 )    /* dividend is lesser */
```

```
    {
        *( dividend + divend_len ) = '0';
        leading_zeros += 1;
        divend_len += 1;
    }
    else
    if ( strcmp_return == 0 )    /* they're the same */
    {
        answer[answer_idx++] = '1';
        goto wrapup;
    };                    /* otherwise, divisor is lesser */
}
/* load the dividend into interim */

strcpy ( interim, dividend );
idivend_len = strlen ( interim );

loop:                    /*--- main loop ---*/

if ( divsor_len > 3 )
    dvsor_len = idvend_len = 4;
else
if ( divsor_len < idivend_len )
{
    if ( strcmp ( divisor, interim ) > 0 )
    {
        dvsor_len = divsor_len;
        idvend_len = divsor_len + 1;
    }
    else
        dvsor_len = idvend_len = divsor_len;
}
else /* can only be the terms are equal length */
{
    dvsor_len  = divsor_len;
    idvend_len = idivend_len;
}

if ( dvsor_len == idvend_len && dvsor_len > 1 )
    if ( DivAtoin ( divisor, dvsor_len ) >
         DivAtoin ( interim, dvsor_len ))
            dvsor_len -= 1;

quo_guess = DivAtoin ( interim, idvend_len ) /
            DivAtoin ( divisor, dvsor_len );

if ( quo_guess > 9 )
     quo_guess /= 10;

try_quo_guess:      /*--- try_quo_guess goto ---*/

DivQuickMult ( divisor, quo_guess, product );

strncmp_return =
    strncmp ( product, interim, strlen ( product ));

if ( strncmp_return > 0 )   /* if product > interim */
```

```
{
    if ( quo_guess == 1 )
    {
        /*
         * a quo_guess of 1 can be a special case:
         * try 9 and bring down another digit
         */

        if ( DivSpecialCase ( divisor, interim ))
        {
            quo_guess = 9;

            /*
             * did we already pad the dividend ?
             * if so, then add a zero to interim dividend,
             * if not, then bring down another digit.
             */

            if ( leading_zeros )
                interim[idvend_len++] = '0';
            else
            if ( next_dvend_digit < divend_len )
                interim[idvend_len++] =
                    dividend[next_dvend_digit++];

            /* and try again */

            goto try_quo_guess;
        }
        else            /* not special case */
            DivShiftArrayRight ( product );
    }
    else

    /* quo_guess != 1, so check whether array needs shift */

    if ( strlen ( product ) < idivend_len )
        DivShiftArrayRight ( product );
    else
    {           /* guess was just too high so try again */
        quo_guess -= 1;
        goto try_quo_guess;
    }
}

/* load the correct digit */

answer[answer_idx++] = quo_guess + ASCII_VAL;

/*
 * if no next digit to bring down has been ascertained,
 * the next operation sets which digit to start bringing
 * down. Only done the first time through here.
 */

if ( ! next_dvend_digit )
    next_dvend_digit = strlen ( product );
```

```
/* new_interim = interim - product */

DivQuickSub ( interim, product, new_interim );

if (( DivCheckZeroOnly ( new_interim )
    && next_dvend_digit  >= divend_len )
    || answer_idx >= MAX_SIZE )        /* are we done? */
        goto wrapup;

while ( *new_interim == '0' )
    DivShiftArrayLeft ( new_interim );

memset ( interim, '\0', 2*MAX_SIZE + 1 );
strcpy ( interim, new_interim );
idivend_len = strlen ( interim );

get_next_digit:     /*--- loop for get next digit ---*/

if ( next_dvend_digit < divend_len )
    interim[idivend_len++] = dividend[next_dvend_digit++];
else
{                   /* if beyond EO dividend, bring down 0 */
    interim [ idivend_len++] = '0';
}

if ( idivend_len < divsor_len )    /* if interim < divisor */
{
    answer[answer_idx++] = '0';
    if ( answer_idx >= MAX_SIZE )
        goto wrapup;
    goto get_next_digit;
}
else
if ( idivend_len == divsor_len )    /* same length */
{
    if ( strcmp ( divisor, interim ) > 0 )
    {                           /* but divisor is greater */
        answer[answer_idx++] = '0';
        if ( answer_idx >= MAX_SIZE )
            goto wrapup;
        goto get_next_digit;
    }
}

goto loop;

wrapup:

/*
 * Now take result from answer and place it in quot. We
 * compare the order of magnitudes of the dividend and the
 * divisor to determine where the decimal point goes.
 * The rule is:
 *
 *   Places  =      Order        -  Order       +  1
 *                      (dvend)         (divisor)
 *
```

```
 *      where strcmp ( dividend, divisor ) > 0
 *      otherwise, don't add 1.
 *
 * A positive number is the number of places left of
 * the decimal point. From this, we subtract any leading
 * zeros.
 */

/* get the order for the dividend */

if ( dvend_arg->places_before != 0 )
    dvend_order = dvend_arg->places_before;
else
{
    i = 0;
    pc = dvend_arg->term + DEC_LOC;
    while ( pc < dvend_arg->term + 2*MAX_SIZE )
    {
        if ( *pc != 0 )
            break;
        else
        {
            i  += 1;
            pc += 1;
        }
    }
    dvend_order = -i;
}

/* get the order for the divisor */
if ( dvsor_arg->places_before != 0 )
    dvsor_order = dvsor_arg->places_before;
else
{
    i = 0;
    pc = dvsor_arg->term + DEC_LOC;
    while ( pc < dvsor_arg->term + 2*MAX_SIZE )
    {
        if ( *pc != 0 )
            break;
        else
        {
            i  += 1;
            pc += 1;
        }
    }
    dvsor_order = -i;
}

i = dvend_order - dvsor_order;

if  ( strcmp ( dividend, divisor ) >= 0 )
    i += 1;

j = DEC_LOC - i;

 /*
```

```
  * quot has already been intialized to zeros,
  * so we can start moving in the digits.
  */

for ( i = 0; i < answer_idx && j < 2*MAX_SIZE; i++, j++ )
     (quot->term)[j] = answer[i] - ASCII_VAL;

/* compute the number of places before and after */

for ( i = 0; i < DEC_LOC; i++ )
      if ( (quot->term)[i] != 0 )
        break;

quot->places_before = DEC_LOC - i;

for ( i = 2*MAX_SIZE - 1; i >= DEC_LOC; i-- )
      if ( (quot->term)[i] != 0 )
        break;

quot->places_after = i - DEC_LOC + 1;
    /* free the terms we created on the heap */

    free ( dividend );
    free ( divisor );
    free ( interim );
    free ( new_interim );
    free ( product );
    free ( answer );

    return ( 1 );
}
```

这个函数中的第一项是#define ASCII_VAL（'0'），它定义了'0'与0之间的差别。它是ASCII字符值（从此就是ASCII_VAL），把一个值转换为它的对应字符。这用于把项转换为字符。代码接下来给需要用于保存由除法生成的内部值的变量分配空间。然后把传递给函数的项加载进字符数组dividend和divisor中。任何前导0（比如当一个项为0.009时可能出现的那些前导0）都会被函数DivShiftArrayLeft()和DivShiftSmallArrayLeft()删除，如程序清单8-14所示。这两个函数之间的区别只是它们移动的数组的大小不同（注意：只由除法例程调用的函数将具有前缀Div）。

程序清单8-14 函数DivShiftArrayLeft()和DivShiftSmallArrayLeft()把字符数组左移一个字节，并丢弃第一个字符。在除法中用于截去前导0

```
/*-------------------------------------------------------------
 * Shifts an array of chars left by one character, truncating
 * the leftmost char. Called by division only when the leading
 * character is a '0'. The small version works on MAX_SIZE
 * arrays, the regular version on 2*MAX_SIZE.
 *-------------------------------------------------------------*/
void DivShiftArrayLeft ( char *array )
{
    char buffer [ 2*MAX_SIZE + 1 ];
    memset ( buffer, '\0', 2*MAX_SIZE + 1);
    strcpy ( buffer, array );
    strcpy ( array, buffer + 1 );
}
```

```
void DivShiftSmallArrayLeft ( char *array )
{
    char buffer [ MAX_SIZE + 1 ];
    memset ( buffer, '\0', MAX_SIZE + 1);
    strcpy ( buffer, array );
    strcpy ( array, buffer + 1 );
}
```

可以毫无损失地截去前导0，因为不需要它们。用于定位小数点的后一个计算将不需要保留该信息。如程序清单8-15所示，然后函数DivCheckZeroOnly( ) 将检查数组，查看任何一个数组是否包含0值。如果传递给该函数的数组只包含0，那么它将返回1；否则，它将返回0。

程序清单8-15　函数DivCheckZeroOnly( ) 检查字符数组是否具有非0值

```
/*-------------------------------------------------------------
 * This function checks a division term for a zero value.
 * Called only by division operation.
 *-------------------------------------------------------------*/
int DivCheckZeroOnly ( const char *array )
{
    while ( *array )
    {
        if ( *array != '0' )
            return ( 0 );
        array += 1;
    }

    return ( 1 );
}
```

如果除数为0，就会发生错误，在屏幕上显示一条消息（即使从脚本运行程序也会如此），并且利用错误代码0终止除法运算。如果被除数为0，NormalDivide( ) 就会把商设置为0并返回（0除以任何合法数字都为0）。

如果除数比被除数长（由于删除了前导0，更长的数字意味着它具有更大的值），就会用0填充被除数，直至它具有与除数相同的长度。因此，6除以15将转换为60除以15。这个操作将影响小数点的位置，因此会在变量leading_zeros（之所以这样命名该变量，是因为它表示必须在最后的商中添加多少个前导0）中记录添加到被除数中的0的个数。在前一个示例中，将8转换为60，并且把leading_zeros设置为1。往后，当我们格式化商时，将会知道正确的答案不是4，而是具有前导0的4，即0.4。

通过用0填充被除数，我们现在知道被除数比除数长，或者它们具有相同的长度。如果被除数和除数长度相同，就比较它们。如果除数更大（比如50除以60），就再次用0填充被除数（使之变成500除以60），并且增大leading_zeros。由于除第一个除法之外的所有其他除法都是使用中间被除数执行的（参见图8-4），就简单地把初始被除数加载进中间被除数interim中，现在就可以为所有除法使用相同的代码。

我们现在准备好进入除法的循环。它开始于goto的目的地，即loop。第一项任务是算出我们可以使用除数和被除数的多少位，以对商的数字进行第一次猜测。我们尝试获取尽可能多的位，由于我们使用的位越多，猜测的位就越有可能正确。我们可以使用的最大位数是4，这是由于16

位整数的大小限制。它们的最大值是65 535——这意味着4位数字将具有最大的大小，其中所有这样的数字都可以保存在整数中。如果除数和被除数比4位数字长，就使用它们的前4位数字。然后比较这两个值。如果除数值大于被除数值，就只取除数的前三位。例如，如果用676774（被除数）除以98985（除数），将从4位“残余数字”开始：即6767和9898。由于9898大于6767，就把除数的残余数字缩减为三位，即989，并用6767除以它，从而得到对商的数字的第一次猜测。

把残余数字转换为整数是由函数DivAtoin()执行的，如程序清单8-16所示。这个函数的工作方式就像是ANSI C函数atoi()，但会给它传递第二个参数，指示要从传递的字符串转换的最大位数。

如果除数和被除数都没有4位或更多的位，就会执行每一次尝试，以确保残余数字具有相同的位数（其中除数的残余数字要小一些），或者除数的残余数字具有较少的位。

最后，一旦确定了残余数字，就会用被除数的残余数字除以除数的残余数字，从而得到对商的数字的猜测。把这个位存储在quo_guess中。在上面的示例中，将用6767除以989，得到一个（正确的）猜测，即6。

程序清单8-16 函数DivAtoin()把传递的字符串转换为一个整数，并且会限制要使用的字符的最大数量

```
/*-------------------------------------------------------------
 * Performs atoi for length number of characters. Called by
 * division to establish the stubs of the divisor and dividend
 * we will use to try guessing the next quotient digit.
 *-------------------------------------------------------------*/
int DivAtoin ( const char *string, int length )
{           /* usual tests for sign and white space omitted */
    int i, n;
    n = 0;

    for ( i = 0;
          i < length && ( string[i] >= '0' && string[i] <= '9'
);
          i++ )
            n = 10 * n + string[i] - '0';

    return ( n );
}
```

我们现在确定猜测是否正确。首先，如果它大于10（比如用残余数字49除以2，得到24），就用它除以10（得到猜测2）。接下来，用猜测的数字乘以除数，并把结果存储在product中。从程序清单8-17中可以看出，通过使用DivQuickMult()执行这个乘法运算，该函数是乘法的简写形式，因为它总是只用一位数字乘以字符串。

程序清单8-17 函数DivQuickMult()用一位数字乘以一个字符数组

```
/*-------------------------------------------------------------
 * This function multiplies an array of digits as characters
 * by a single-digit integer. Called by division only.
 *-------------------------------------------------------------*/
void DivQuickMult ( const char *long_term, int digit,
                    char *result )
{
```

```
    int from, to; /* array subscripts */
    int new_carry, old_carry, hold;

    new_carry = old_carry = hold = to = 0;

    memset ( result, '\0', 2*MAX_SIZE + 1 );
    for ( from = strlen ( long_term ) - 1;
          from >= 0; from--, to++ )
    {
        hold = ( long_term[from] - ASCII_VAL ) * digit;
        new_carry = hold / 10;
        result[to] = hold % 10 + old_carry;
        if ( result[to] > 9 )
        {
            new_carry += 1;
            result[to] -= 10;
        }
        result[to] += ASCII_VAL;
        old_carry = new_carry;
    }

    if ( old_carry )        /* if any left over */
        result[to] = old_carry + ASCII_VAL;

    strrev ( result );
}
```

如果积大于（通过 strcmp() 确定）中间的被除数，我们的猜测就太大。在采取措施之前，需要探讨几种可能性。第一种是：strcmp() 函数提供了不准确的比较。例如，如果我们用 12 除以 7，正确的猜测将是 1，并且积将是 7。不过，当字符串 7 和 12 之间的比较发生时，7 似乎要大于 12，因为 7 大于 12 中的初始 1。在这种情况下，我们将检查积的长度。如果它小于中间被除数的长度，我们就知道具有一个较小的数，而不管 strcmp() 函数的结果是什么。不过，为了规范化积以便用于以后的减法，我们把积向右移动，并前置一个（或多个）前导 0，直至它具有与中间被除数相同的长度。这个操作是由 DivShiftArrayRight() 函数执行的，其操作是自解释的。参考程序清单 8-18。

程序清单 8-18　函数 DivShiftArrayRight() 右移字符数组并前置 0

```
/*-------------------------------------------------------------
 * Shifts an array of characters right one character and
 * prepends a zero. Called only by division.
 *-------------------------------------------------------------*/
void DivShiftArrayRight ( char *array )
{
    memmove ( array + 1, array, strlen ( array ));
    array[0] = '0';
}
```

另一种情况是：猜测的数字太大。如果它具有不同于 1 的任何值，我们就把它递减 1，并尝试再次验证猜测。为此，我们跳转到 try_quo_guess。

不过，如果猜测是 1，就不能轻率地递减猜测，因为猜测 0 总是一个错误。我们将检查是否

应该把猜测设置为9。为此，调用函数 DivSpecialCase()，如程序清单8-19所示。该函数将检查9是否会工作。

程序清单8-19 函数 DivSpecialCase() 用9乘以除数，并把积与中间被除数作比较，以便测试猜测9是否会工作

```
/*-------------------------------------------------------------
 * If the division guesses a quotient digit that is too high,
 * we normally would decrement the guess by 1 and try again.
 * However, if the guessed digit is a 1, we have to be careful,
 * because the decrement should give us a 9, not a 0. This
 * function tests whether the correct digit is in fact a 9, or
 * whether we have misapprehended the dividend on our guess.
 * Called only by division.
 *-------------------------------------------------------------*/
int DivSpecialCase ( const char *divisor,
                     const char *curr_dividend )
{
    char test_result [2*MAX_SIZE + 1];

    DivQuickMult ( divisor, 9, test_result );

    if ( strcmp ( curr_dividend, test_result ) > 0 )
        return ( 1 );
    else
        return ( 0 );
}
```

在以下情况中将需要这种测试。如果用888 867除以88 888，两个残余数字都是8888，则会得到猜测1。在指出9是否会工作之前，利用这个函数测试它。另一种可能性是以前明确指出的一种可能性（12/7 = 1，strcmp() 将会错误地执行比较）。不管怎样，我们都会递减到9，或者保留目前的猜测。

不过，如果我们递减到9，就不能简单地重新尝试猜测，因为如果猜测1太大，那么9总是会失败。我们必须从被除数中转到下一位数字。从以前所做的工作可知：如果 leading_zeros 大于0，那么就已经不得不填充被除数，因为被除数本来就比除数短。在这种情况下，我们只是简单地给中间被除数添加另一个0，然后试试猜测9。

在确定了一位正确的猜测数字之后，就把它存储在数组 answer 中。然后要注明我们在被除数中所处的位置，以便当我们转到被除数中的下一位数字时，就可以知道我们在被除数中移动了多远的位置。这个位置是由 next_dvend_digit 跟踪的。

接下来，将使用 DivQuickSub() 从中间被除数中减去所猜测数字与除数的积，如程序清单8-20所示。结果是新的中间被除数，称为 new_interim。

程序清单8-20 函数 DivQuickSub() 从一个字符数组中减去另一个字符数组

```
/*-------------------------------------------------------------
 * Subtracts one array of chars (the subtrahend) from another
 * array (the minuend), generating a difference. Called only
 * by division.
 *-------------------------------------------------------------*/
void DivQuickSub ( char *minuend, char *subtrahend,
```

```
                    char *diff )
{
    int sub, to;    /* indices to various arrays */

    sub = to = strlen ( subtrahend ) - 1; /* start at right   */
    diff[to + 1] = '\0';        /* after setting end of string */

    while ( sub >= 0 )
    {
        if ( minuend[sub] < subtrahend[sub] )
        {
            minuend[sub] += 10;
            subtrahend[sub - 1] += 1;
        }
        diff[to--] = minuend[sub] - subtrahend[sub] + CV;
        sub -= 1;
    }
}
```

我们将检查新的中间被除数，查看它是否为0。如果它为0，那么当且仅当我们转到了被除数中的每一位数字或者我们的答案超过了最大位数时，我们就完成了任务。如果我们完成了任务，就跳转到 wrapup，其中将适当地对答案进行格式化。

如果我们没有完成任务，就删除新的中间被除数中的任何前导0，并从原始被除数中转到另一位数字。如果我们已经越过了被除数的末尾，那么就转到0，只通过把它追加到新的中间被除数末尾即可。

在我们转到这一位数字之后，就把新的中间被除数与除数作比较。如果它小于除数，猜测商的数字就是没有意义的；我们知道商将是0。因此，我们只是简单地把0添加到答案中，转到下一位数字，并再次尝试比较。仅当中间被除数大于除数时，才会循环回到除法过程的开头，并再次尝试。

最后，我们就具有一个完整的解决方案。剩下的唯一任务是确定小数点的位置。针对它的规则是：比较除数和被除数的数量级。答案中的第一位数字是距离小数点的位置数 x，其中 x 为正表示位于小数点的左边，x 为负则表示位于小数点的右边。公式如下：

$$x = \text{order}(\text{dividend}) - \text{order}(\text{divisor}) + y$$

其中：当 strcmp（dividend，divisor）>0 时，y=1；否则，y=0。例如，12/5=2.4。order（dividend）=2，order（divisor）=1，y=0。因此，x=1，并且小数点左边有1位数字。把这种情况与75/5=15作比较；数量级与前一个示例相同，但是 y=1，因此 x=2，并且小数点左边有两位数字。

在知道了小数点的位置以后，就把数字加载进 TermData 结构中，小心地减去 ASCII_VALUE 值，并从除法函数返回。

当然还有其他方法用于除法运算。一些作者使用的方法是：把数字转换为以256为基数，然后应用快速傅里叶变换，帮助确定下一个商猜测。

我们之所以选择上述方法，是因为它比较直观并且易于理解。它允许读者对其进行优化。可能的优化包括：不使用字符串，而代之以使用二进制值，就像在其他运算中所做的那样。也可以

对快速的乘法和减法函数进行优化，使用详细介绍这些运算时所示的处理方法即可。可以把数组的右移和左移操作移入内联代码中，而不是每次都要调用函数。在这些优化当中，最具有挑战性的当然是字符串的转换。如果保留字符串，简单的优化不会每次都加上和减去 ASCII_VALUE，而是把这些值放入一个查找表中，这与我们在乘法一节中所做的相似。

程序清单 8-21 中的代码显示了函数声明和定义的常量。

程序清单 8-21 头文件 longmath. h，它显示了用于计算器的函数声明和全局变量

```
/*--- longmath.h --------------------------- Listing 8-20 ------
 *  Header file for arbitrary-precision arithmetic routines
 *  in longmath.c
 *-------------------------------------------------------------*/

#ifndef LONGMATH_H              /* avoid multiple inclusions */
#define LONGMATH_H  1

struct TermData {               /* the data for each term */
        char   *term;               /* the term */
        int     sign;               /* pos or neg; 0 = error */
        int     places_before;      /* before decimal point */
        int     places_after;       /* after decimal point */
};

#ifndef MAX_SIZE
#define MAX_SIZE    20          /* maximum length of a term. */
#endif                          /* Can be changed freely.    */

#define NORMAL      1           /* formats of terms */
#define SCIENTIFIC  2

#ifndef min
#define min(a,b) ((a) < (b) ? (a) : (b))
#define max(c,d) ((c) > (d) ? (c) : (d))
#endif


int  AsciiToTerm    ( char *, struct TermData * );
int  AsciiToScientific ( char * );
int  ComputeResult  ( struct TermData *, int,
                        struct TermData *, struct TermData * );
int  DivAtoin       ( const char *, int );
int  DivCheckZeroOnly   ( const char * );
void DivQuickMult   ( const char *, int, char * );
void DivQuickSub    ( char *, char *, char * );
void DivShiftArrayLeft      ( char * );
void DivShiftSmallArrayLeft ( char * );
void DivShiftArrayRight     ( char * );
int  DivSpecialCase ( const char *, const char * );
int  GetFileOperator( int *, FILE *, FILE * );
int  GetFileTerm    ( struct TermData * , FILE *, FILE * );
int  NormalAdd      ( struct TermData *, struct TermData *,
                        struct TermData * );
int  NormAbsCmp     ( struct TermData *, struct TermData * );
```

```
int  NormalDivide   ( struct TermData *, struct TermData *,
                      struct TermData * );
int  NormalMultiply ( struct TermData *, struct TermData *,
                      struct TermData * );
int  NormalSubtract ( struct TermData *, struct TermData *,
                      struct TermData * );
char * strrev       ( char * ); /* for portability */
void TermInit       ( struct TermData * );
struct TermData *
     TermCopy       ( struct TermData *, struct TermData * );
struct TermData *
     TermCreate     ( void );
void TermToAscii    ( struct TermData *, char *, int );


#define DEC_LOC     MAX_SIZE    /* location of decimal point */

#endif
```

8.8 关于计算器要注意的最后几点

本章中展示的计算器是旨在演示使用任意精度的算术的应用程序。它并不打算作为一种生产计算器。为了使之变成完全合格的生产计算器，甚至只对于算术函数，也应该考虑几种增强措施：

- 应该检测任何溢出。目前，这种实现要求用户知道所生成结果的最大宽度。它不会检查用户指定的宽度是否足够。考虑一下，如果用数量级为 10^{-20} 的数字除以数量级为 10^{20} 的数字，得到的商将至少具有 40 位数字。避免这个问题的一种安全的方式是：在编译例程时，把 MAX_SIZE 设置为比所需的大得多的某个值，比如 100 位数字。
- 一些例程把项放在栈上。如果你预期将使用超过 250 位数字，这些例程就应该把项放在堆上，而不是放在会使栈溢出的磁盘上。其折中仍然是性能。
- 脚本工具应该允许变量和注释。
- 目前实现的脚本工具把错误消息显示到屏幕上，并在输出脚本中简单指出发生的错误。
- 良好的计算器将通过添加记忆保存/恢复、倒数和指数来扩展这些例程。它还将在结果中插入逗号来改进可读性，并将以可用的最大精度提供内置的常量，比如 π 和 e。
- 具有某种方式来指出小数点后面所需的位数是有帮助的。在涉及金钱的计算中尤其需要这种特性。

8.9 用于计算平方根的牛顿算法

如本章开头所提到的，使用任意精度的例程的一个非常有说服力的理由是：可以获得比大多数 C 编译器及其标准库所提供的浮点支持更准确的答案。这里演示了上述的例程，以显示它们可匹敌的准确性。

本节中展示的程序通过使用艾萨克·牛顿在 18 世纪发明的著名算法来计算平方根。该程序还将通过调用标准 C 库中的 sqrt( ) 来计算生成的平方根——并比较两个结果。

牛顿的算法源于他在求解多项式方程时所做的工作［Knuth 1981］。他提议的方程（取自 Robert Seeley 的书籍 Calculus of One Variable［Seeley 1972］）用于常量 c，其中 $c>0$：

$$x_{n+1} = \frac{c + x_n^2}{2x_n}$$

该方程执行 c 的平方根的一系列近似。每个近似都构建在前一个之上。一旦计算了 $x_n + 1$，它就在下一次迭代中变为 x。x 的第一个值是对平方根的猜测。在我们的实现中，第一个猜测总是 $c/3$。可以根据需要执行许多次近似；无论何时用户终止迭代，$x_n + 1$ 都是然后要计算的 c 的平方根。

在实际中，当发生以下两种情况之一时，应该停止近似。第一种情况是：当两次连续的迭代产生相同的 $x_n + 1$ 值时。这个值是该方法可以近似的最佳平方根。停止近似的第二种情况是：当牛顿的方法开始在两个值之间来回往复时。对于某些数字——该方法可以获得非常接近的值（一般稍小一点），然后它会再次迭代，并获得一个稍大的值，在下一次迭代中，它会获得前一个太小的数字。此时，它就进入了一个将总是生成相同值对的循环中。在我们的实现中，将通过不允许超过 50 次近似来捕获这种情况。由于在 50 次近似之前汇聚在单个平方根上的几乎所有的值变得如此之长，我们任意设置这种限制，作为进一步计算的中断点。程序清单 8-22 显示了用于牛顿的平方根近似算法的驱动程序。

程序清单 8-22　用于牛顿的平方根近似算法的驱动程序

```
/*--- sqrtmain.c ------------------------- Listing 8-22 -------
 * Uses the longmath routines to perform Newton's algorithm for
 * finding square roots. Compares the result to the C floating-
 * point function sqrt(). This implementation keeps approximating
 * until Newton's method no longer generates a difference
 * or until 50 iterations have been performed. Beyond 50
 * iterations, it is assumed that the algorithm is ping-ponging
 * around the final number. Square root of 525 with MAX_SIZE
 * = 20 is an example of this problem.
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "longmath.h"

main ( int argc, char *argv[] )
{

    int i;
    char buffer[2*MAX_SIZE];


    struct TermData *square;     /* the original number */

    struct TermData *xn,         /* variables in equation */
                    *xn1,
                    *product;

    struct TermData *two,        /* constants */
                    *three;

    struct TermData *temp1,      /* intermediate results */
                    *temp2,
```

```
                    *temp3,
                    *z;

    float fsquare, fsqrt;       /* the floats we'll compare to */

    square  = TermCreate();
    xn      = TermCreate();
    xn1     = TermCreate();
    product = TermCreate();
two     = TermCreate();
three   = TermCreate();

temp1   = TermCreate();
temp2   = TermCreate();
temp3   = TermCreate();
z       = TermCreate();


if ( argc < 2 )
{
    printf ( "Usage: sqrtmain number\n" );
    return;
}

AsciiToTerm ( argv[1], square );

/* no imaginary numbers in this version */

if ( square->sign == -1 )
{
    printf ( "Square cannot be negative\n" );
    return;
}

/* square root of zero is zero */

if ( square->places_before + square->places_after == 0 )
{
    printf ( "0\n" );
    return;
}

/*
 * We start with a guess of dividing square by 3, hence
 *           xn = square / 3.0
 */

AsciiToTerm ( "3.0", three );

if ( ! ComputeResult ( square, '/', three, xn ))
{
    printf ( "error 1\n" );
    return;
}


printf ( "Square root of %s\n", argv[1] );
```

```
TermToAscii ( xn, buffer, NORMAL );
printf ( "First guess is %s\n", buffer );

AsciiToTerm ( "2.0", two );

for ( i = 0; ; i++ ) /* repeat until root or max 50 loops */
{

    /* Newton's approach keeps approximating using
     * this formula:
     *  xn1 = ( square + ( xn * xn )) / ( 2.0 * xn )
     */

    TermCopy ( z, xn );
    if ( ! ComputeResult ( z, '*', xn, temp2 ))
    {
        printf ( "error 2\n" );
        return;
    }

    if ( ! ComputeResult ( square, '+', temp2, temp1 ))
    {
        printf ( "error 3\n" );
        return;
    }


    if ( ! ComputeResult ( two, '*', xn, temp3 ))
    {
        printf ( "error 4\n" );
        return;
    }

    if ( ! ComputeResult ( temp1, '/', temp3, xn1 ))
    {
        printf ( "error 5\n" );
        return;
    }

    TermToAscii ( xn1, buffer, NORMAL );
    printf ( "%2d %s\n", i, buffer );

    /* Are we done ? */

    if ( ! NormAbsCmp ( xn, xn1 ))
        break;

    if ( i > 49 )
    {
            printf ( "\nLast entry is the closest approxima"
                     "tion. Algorithm no longer converging\n" );
            break;
        }

        /* The current xn1 becomes the next xn */

        TermCopy ( xn, xn1 );
```

```
    }

    /* print the square root */

    TermToAscii ( xn1, buffer, NORMAL );
    printf ( "\nSquare root: %s\n", buffer );

    /* print the square of the square root */

    TermCopy ( xn, xn1 );
    if ( ! ComputeResult ( xn, '*', xn1, product ))
    {
        printf ( "error 6\n" );
        return;
    }

    TermToAscii ( product, buffer, NORMAL );
    printf ( "Computed square: %s\n", buffer );

    /* print the difference */

    if ( ! ComputeResult ( square, '-', product, xn ))
    {
        printf ( "error 7\n" );
        return;
    }
    TermToAscii ( xn, buffer, NORMAL );

    printf ( "Delta: %s\n", buffer );

    /* now test the built-in floating-point routines */

    fsquare = atof ( argv[1] );
    fsqrt = sqrt ( fsquare );

    printf ( "\n\nMath lib root: %4.12f, square: %4.12f\n",
                fsqrt, fsqrt * fsqrt );

    return;
}
```

对于要查找其平方根的数字，程序期望作为命令行参数传递它。例如，要查找 12 的平方根，可以输入：

```
sqrtmain 12
```

图 8-5 显示了输出到屏幕上的结果。

算法要花费 16 次近似才能确定 12 的平方根。在列出这些近似值后，程序将打印所计算的平方根，并把计算的平方根与原始平方根之间的差值显示为 Delta。在最后一行，显示通过使用标准 C 库中的 sqrt（12）获得的平方根和根的平方值。

这个示例显示通过牛顿算法获得的结果要比标准数学库准确得多。上面的结果是使用 Borland 的 C/C + + 编译器版本 4.0 中的库生成的。通过其他库生成的值显示在准确性方面的差距相当大。完全根据经验，我们发现标准数学库例程对于大于 1000 万的值更准确。在这个阈值以下，牛顿的

```
Square root of 12
First guess is 4
 0 3.5
 1 3.464285714285714285714
```

```
 * The program output consists of a table that shows the
 * outstanding principal, the amount of interest, and the
 * principal paid for each month of the loan.
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "longmath.h"

void ShowUsage ( void );
int main ( int argc, char *argv[] )
{
    struct TermData *principal, *rate, *payment;
    struct TermData *interest, *temp, *temp2;

    char buffer [7*MAX_SIZE];   /* buffer for output */

    int i, month;

    month      = 0;
    principal  = TermCreate();
    rate       = TermCreate();
    payment    = TermCreate();
    interest   = TermCreate();
    temp       = TermCreate();
    temp2      = TermCreate();

    /* try out various values */

    if ( argc != 4 )
    {
        ShowUsage();
        return ( EXIT_FAILURE );
    }

    for ( i = 1; i < argc; i++ )
    {
        char *p;
        if ( *argv[i] != '-' )
        {
            ShowUsage();
            return ( EXIT_FAILURE );
        }
        else
        {
            p = &argv[i][1];
            switch ( *p )
            {
                case '$':
                    AsciiToTerm ( p + 1, principal );
                    break;
                case 'p':
                    AsciiToTerm ( p + 1, payment );
                    break;
                case 'r':
```

```
                    AsciiToTerm ( p + 1, rate );
                    break;
                default:
                fprintf ( stderr,
                            "Invalid switch: -%c\n", *p );
                ShowUsage();
                return ( EXIT_FAILURE );
        }
    }
}

/*
 * divide rate by 12 to get monthly rate,
 * then copy back to rate.
 */

AsciiToTerm ( "12", temp );
if ( ! ComputeResult ( rate, '/', temp, temp2 ))
{
    printf ( "error 1\n" );
    return ( EXIT_FAILURE );
}
TermCopy ( rate, temp2 );

/* print principal and monthly rate */

memset ( buffer, ' ', MAX_SIZE * 7 );
TermToAscii ( principal, buffer, NORMAL );

buffer [strlen ( buffer )] = ' ';
TermToAscii ( rate, buffer + MAX_SIZE * 2 + 3, NORMAL );
buffer[strlen ( buffer )] = ' ';
TermToAscii ( payment, buffer + MAX_SIZE * 4 + 3, NORMAL );

printf (
    "Original Principal     Monthly Rate        Payment\n" );
printf ( "%s\n\n", buffer );

printf ( "Mnth Principal Left      Interest Paid" );
printf ( "       Principal Paid\n" );

/* print payments while principal > 0 */

while ( principal->sign > 0 )
{

    /* interest = principal * rate */

    if ( ! ComputeResult ( principal, '*', rate, interest ))
    {
        printf ( "error 2\n" );
            return ( EXIT_FAILURE );
        }

        /* temp = payment - interest */

        if ( ! ComputeResult ( payment, '-', interest, temp ))
```

```
        {
            printf ( "error 3\n" );
            return ( EXIT_FAILURE );
        }

        if ( temp->sign < 0 )
        {
            printf (
                "Monthly interest is > than monthly payment"
                "\nLoan will never be paid off\n" );
            return ( EXIT_FAILURE );
        }

        /* print month (same as payment number) */

        if (( month % 12 ) == 0 )
            printf ( "\n" );
        printf ( "%4d ", ++month );

        /* print: principal  interest  temp */

        memset ( buffer, ' ', MAX_SIZE * 7 );
        TermToAscii ( principal, buffer, NORMAL );
        buffer[strlen ( buffer )] = ' ';
        TermToAscii ( interest, buffer + MAX_SIZE * 2, NORMAL );
        buffer[strlen ( buffer )] = ' ';
        TermToAscii ( temp, buffer + MAX_SIZE * 4, NORMAL );
        printf ( "%s\n", buffer );

        /* principal = principal - temp */

        if ( ! ComputeResult ( principal, '-', temp, temp2 ))
        {
            printf ( "error 4\n" );
            return ( EXIT_FAILURE );
        }
        TermCopy ( principal, temp2 );
    }

    return ( EXIT_SUCCESS );
}
void ShowUsage ( void )
{
    printf (

   "Loan amortization calculator. Enter the principal,\n"
   "interest rate, and loan payment on the command line,\n"
   "where -$ = principal, -r = rate, -p =payment. For example,\n"
   "a loan of $250,000 at 7.5%% with payments of $2900 enter:\n\n"
   "   amrtmain -r0.075  -$250000 -p2900\n\n"
   "Note: UNIX users should enter the principal as -\\$250000\n" );
}
```

8.11 资源和参考资料

Bentley, Jon. *More Programming Pearls: Confessions of a Coder.* Reading, MA: Addison-Wesley,

1988. 这是算法文献中真正的经典。它是 Bentley 对其在 *Communications of the ACM* 中的“Programming Pearls”专栏的重述的第二卷。

Knuth, Donald E. *The Art of Computer Programming*. Vol. 2: Semi-Numerical Algorithms, 2d ed. Reading, MA: Addison-Wesley, 1981. 参见第 250 ~ 312 页，了解关于任意精度的算术的讨论。自从 Knuth 论述这个主题起，已经出现了多种其他的方法。

Nakamura, Shoichiro. *Applied Numerical Methods in C*. Englewood Cliffs, NJ: Prentice Hall, 1993. 这是一本关于数值方法的教材，其中包括 C 语言中的良好解释和多种解决方案。

Seeley, Robert. *Calculus of One Variable*. Glenview, IL: Scott, Foresman & Co., 1972. 这是一本学院级微积分学教材，不建议用于教学目的。

第9章 数据压缩

数据压缩尝试转换数据，使得它们占据更少的空间。今天存在两种类型的数据压缩：**无损**（lossless）压缩和**有损**（lossy）压缩，在前一种压缩方式中，在压缩期间不会发生数据损失；在后一种压缩方式中，将会丢失一些数据，并且不会把它们纳入到压缩的数据中。如果采用有损压缩，在对压缩文件进行解压缩时，将不会产生原始文件的所有数据。有损压缩主要用于压缩某些类型的图形文件，其中对极高压缩程度的要求超过了对保留所有原始数据的要求。例如，对较高分辨率的图像进行压缩以便于传送到站点上，当以较低的分辨率显示时可能会因为压缩程度非常高而牺牲一些细节。音频文件可以进行类似的折中。在本章中，只将介绍无损算法。也就是说，算法保证解压缩的文件将具有与原始文件相同的数据。

所有的数据压缩技术都依赖于数据冗余的存在。它们都会检查要压缩的数据，以便找出可以用更短符号表示的重复出现的字符或字符模式。把重复出现的字符或模式转换为较短符号（称为**代码**（code））的过程称为**编码**（encoding）；把代码转换为原始字符或模式的过程称为**解码**（decoding）。文本编辑器使用简单形式的编码把一系列空格编码为制表符。制表符是对许多必要的空格进行编码的符号，以达到下一个制表位。

如果压缩算法找不到任何要编码的模式，就不会发生数据压缩。在这种情况下，输出文件最多具有与原始文件相同的大小；在最坏情况下（并且这种情况更常见），压缩文件将比原始文件更大。这个原则是不可违反的：对于所有的压缩算法和每个输入文件，都会存在一个不能进一步压缩的压缩版本。如果不是这样，就可以反复压缩文件，直至它收缩到单个字符。

其他一些因素可能使输出文件变得比原始文件更大。一个主要原因是内务处理开销。为了使压缩程序正确地工作，它通常需要具有关于如何编码文件的信息。这种信息一般是在输出文件开头的头部区域中传递的。在上一个把空格转换为制表符的示例中，解压缩程序必须知道制表位出现在什么位置（每4列、每8列等）。如果源文件中缺少这种信息，那么当目标机器使用不同于原始机器的制表位时，这通常会导致程序员手动重新排列代码。如果将空格转换为制表符的压缩是真实的数据压缩程序包的一部分，压缩文件将具有制表位大小的开销。

还有其他一些扩展开销。继续使用上一个示例，我们将在不应该将制表符扩展为空格的文件中陷入困境。例如，UNIX makefiles 中的某些条目需要制表符。为了说明不应该扩展它，制表符必须具有某种标志。必须把这个标志的存在和含义传达给解压缩程序，这会导致额外的开销。

由于数据压缩的效力（表示为编码字节与解压缩的数据字节之间的比率）与原始数据中的模式紧密相关，它会遵循以下原则：对数据本身知道得越多，对它的压缩就越高效。因此，可以开发一些即席数据压缩模式，它们对于预期的数据集极其高效，即使在不同的环境中应用它们时可能不令人满意。本章中展示的算法都是通用的压缩模式。尽管它们具有通用性，但是可以明显看出：对于某些类型的数据，某些算法将远远好于其他算法。

9.1 行程编码

行程编码（run-length encoding，RLE）有时称为**重现编码**（recurrence coding）。这两个名称都暗示了这种算法将采用的方法。它寻找一系列重复出现的字符，并把它们压缩成单个字符，其后接着一个计数，说明该字符在顺串（run）中的出现次数。

例如，可以把字符串 AAAAAABBBBCCCCC 压缩成字符串 A6B4C5，它显示了每个字符，其后紧接着该字符的出现次数。这个压缩字符串包含三个代码（在这里是字母-计数的组合），可以轻松地将其解码为原始字符串。编码的字符串占据 6 个字节，而原始字符串则占据 15 个字节。这个示例的压缩效率是 40%。也就是说，压缩数据的大小与原始数据的大小之间的比率 6/15 是 0.4。熵同样也是 0.40；它采用 48 位来编码 120 位的数据。

这个示例的局限性在于：它不能很好地压缩字符的较短顺串。由于代码需要两个字节，在压缩两个字节的顺串时不会节省空间，并且在编码单个字母的顺串时还要进行扩展。因此，字符串 AABCD 将变为 A2B1C1D1。编码字符串的长度为 8 个字节，而原始字符串的长度只有 5 个字节。压缩是无效的。

为了修正这个问题，仅当序列中包含三个或更多的相同字符时，才对它们进行编码。所有其他的字符将不会进行压缩。为了在压缩文件中确定序列的开头，我们使用一个标志字节，将下两个字节标识为压缩代码。采用这种方式，整个代码将占据三个字节：一个标志字节、组成序列的字符和重复出现的计数。例如，AAAAABC 将变为［标志字节］A5BC。在后面的实现中使用了这种模式，并且它是通常实现 RLE 的方式。

标志字节（也称为**哨兵**（sentinel））需要特殊考虑。如果哨兵自然地出现在要压缩的文本中，就必须特殊地对待它。处理它的一种简单方式是：编码它，并赋予它计数 1。因此，如果我们使用哨兵 0xFF，就将获得以下可能性：

- 用于序列 AAAAA 的代码：

  0xFF, 'A', 5　　（常规压缩代码）

- 用于 0xFF 的自然出现的字节：

  0xFF, 0xFF, 1　　（输入文本中的哨兵）

利用这种方法，出现在输入文件中的单个哨兵字节将扩展得到的输出。因此，应该小心选择哨兵值，使得它不太可能在输入文件中频繁出现（注意：输入文件中的每个哨兵字节都不必生成 3 字节的代码。如果出现两个连续的哨兵，则只需把代码中的计数改为 2 即可；对于 3 个或更多的连续哨兵也是如此。只有输入文件中的 1 个或 2 个哨兵字节的顺串才会扩展输出）。这里暗示了对它自身进行优化。由于只有哨兵字节可以具有计数 1 或 2，可以只使用两个字节来编码哨兵顺串 1 或 2：哨兵字节后接计数 1 或 2。无需添加要重复的字节的值，因为从计数可知重复字节只能是哨兵。通过这种模式，处理输入文件中单个哨兵字节的开销将减小到 2 个字节。

在程序清单 9-1（rle1.c）中所示的 RLE 的实现中，以稍微不同的方式排列代码：哨兵字节、计数和重复字节。这种顺序是前面刚刚讨论过的优化所必需的。注意：最大计数是最大字节值，即 255。大于 255 字节的重现序列将分解为一系列代码，它们每个都表示一个 255 字节的分段。

程序清单9-1 用于行程编码压缩的编码器

```
/*--- rle1.c ---------------------------- Listing 9-1 ---------
 *  Run-Length Encoding for files of all types
 *
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>

#define Sentinel    0xF0   /* the sentinel flag */
#define BUFFER_SIZE 30000

#define WriteCode(a,b,c) \
            fprintf( outfile, "%c%c%c", a, b, c )

FILE *infile,
     *outfile;

int main ( int argc, char *argv[] )
{
    char *buffer;
    char prev_char;
    int  bytes_read,
         count,
         eof,
         first_time,
         i;

    if ( argc != 3 )
    {
        fprintf ( stderr, "Performs standard RLE compression\n"
                          "Usage: rle1 infile outfile\n" );
        return ( EXIT_FAILURE );
    }

    if (( infile = fopen ( argv[1], "rb" )) == NULL )
{
    fprintf ( stderr, "Error opening %s\n", argv[1] );
    return ( EXIT_FAILURE );
}

if (( outfile = fopen ( argv[2], "wb" )) == NULL )
{
    fprintf ( stderr, "Error opening %s\n", argv[2] );
    return ( EXIT_FAILURE );
}

/* write the header to the output file */

fprintf ( outfile, "%c%c%c%c", 'R', 'L', '1', Sentinel );

/* initialize the necessary variables */

eof        = 0;
first_time = 1;
```

```
buffer = (char *) malloc ( BUFFER_SIZE );
if ( buffer == NULL )
{
    fprintf ( stderr, "Unable to allocate %d bytes\n",
                BUFFER_SIZE );
    return ( EXIT_FAILURE );
}

/* process the input file */

while ( ! eof )
{
    bytes_read = fread ( buffer, 1, BUFFER_SIZE, infile );
    if ( bytes_read == 0 )
    {
        eof = 1;
        break;
    }

    for ( i = 0; i < bytes_read; i++ )
    {
        /* first time through is a special case */

        if ( first_time )
        {
            prev_char = buffer[i];
            count = 1;
            first_time = 0;
            i++;
        }

        if ( buffer[i] == prev_char )   /* repeated char */
        {
            count += 1;
            if ( count == 255 )
            {
                WriteCode ( Sentinel, count, prev_char );
                count = 0;
            }
            continue;
        }
        else    /* a new char, so write out all known data */
        {
            if ( count < 3 )
            {
                if ( prev_char == Sentinel )
                {
                    fprintf ( outfile, "%c%c",
                                    Sentinel, count );
                }
                else
                    do
                    {
                        fputc ( prev_char, outfile );
                    }
                        while ( --count );
            }
```

```
            else
                WriteCode ( Sentinel, count, prev_char );

            prev_char = buffer[i];
            count = 1;
        }
    }

    /* we're at end of bytes_read, is it EOF? */

    if ( bytes_read < BUFFER_SIZE )
        eof = 1;
}

/* at EOF, so flush out any remaining bytes to be written */

if ( count < 3 )
{
    if ( prev_char == Sentinel )
        {
            fprintf ( outfile, "%c%c",
                            Sentinel, count );
        }
        else
            do
            {
                fputc ( prev_char, outfile );
            }
                while ( --count );
    }
    else
        WriteCode ( Sentinel, count, prev_char );

    fclose ( infile );
    fclose ( outfile );

    return ( EXIT_SUCCESS );
}
```

程序清单9-1中的程序非常直观。它逐字节读入数据，如果当前字节与前一字节不同，就把前一字节写到输出文件，并保存当前字节。如果当前字节与前一字节相同，就递增计数器。如果计数器超过3（它是使编码高效的最小长度），那么下一个不同字节将强制用公式表示压缩代码，并把它写到输出文件。还会如以前所讨论的那样监视和处理在文本中自然出现的哨兵（这里相当随意地把它选作0xF0）。

要注意的一个细节是：在写入任何压缩数据之前，程序将把头部信息写到输出文件。这个头部由4个字节组成：前三个字节包含字母RL1，最后一个字节包含哨兵值。至少，需要一个字节的头部来标识哨兵字节的值。字符串RL1用于确认对正确类型的压缩文件应用解压缩程序。关于字母RL1没有任何神秘之处；任何充分的信息值都将会工作。

就哨兵值的选择而言，可以增强RLE实现。以前的讨论表明：可以在输出文件中扩展自然出现在输入文件中的单个哨兵字节。因此，监视输入文件中自然出现的哨兵的频率就是有意义的。

如果它们超过预定的阈值，更改哨兵就是有意义的。可以使用未定义的代码值来执行该任务：代码包含一个哨兵字节，其后接着计数0。因此，要把哨兵从0xF0改为0xF4，可以发出后面的代码：0xF0，0，0xF4。一旦在哨兵代码后面检测到计数0的字节，解压缩程序就知道下一个字节表示新哨兵。

数据压缩算法能够更改它处理数据的方式，以便获得更好的效率，这使得算法是**自适应的**(adaptive)。用于数据压缩的许多技术都是自适应的，尤其是在压缩效力比压缩速度更重要的地方。例如，霍夫曼算法（Huffman algorithm，在本章后面介绍）读取输入文件两次：一次用于确定输入文件中每个字符出现的频率，另一次则使用字符频率来获得最大的压缩程度。

在RLE的这个版本（并且在大多数版本）中没有实现这里建议的自适应程度，因为RLE被设计成快速、简单。开销越小越好。此外，RLE几乎总是用在已知的数据集上（比如数据库），其中选择有效哨兵的可能性很高。

用于RLE的解压缩例程（参见程序清单9-2，unrle1.c）读取压缩文件，检查头部中的RL1签名，并使用下一个字节作为哨兵值。然后，程序从头至尾读取文件，把任何普通字符传递给输出文件，并扩展通过哨兵值通告的任何代码。压缩例程和解压缩例程都使用较大的输入缓冲区(30 000字节)来加快处理速度。解压缩例程中的大部分代码都用于处理撞到缓冲区边界的压缩代码的问题。除了这个内务处理问题之外，代码是相当直观的。

程序清单9-2　用于行程编码解压缩的解码器

```
/*--- unrle1.c ---------------------------- Listing 9-2 --------
 *  Uncompresses rle1 files
 *
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 30000

FILE *infile,
     *outfile;

int main ( int argc, char *argv[] )
{
    char *buffer,
         sentinel;
int  bytes_read,
     count,
     i;

/*--- process files open ---*/

if ( argc != 3 )
{
    fprintf ( stderr, "Uncompresses RLE1-compressed file\n"
                      "Usage: unrle1 infile outfile\n" );
    return ( EXIT_FAILURE );
}
```

```
if (( infile = fopen ( argv[1], "rb" )) == NULL )
{
    fprintf ( stderr, "Error opening %s\n", argv[1] );
    return ( EXIT_FAILURE );
}

if (( outfile = fopen ( argv[2], "wb" )) == NULL )
{
    fprintf ( stderr, "Error opening %s\n", argv[2] );
    return ( EXIT_FAILURE );
}

/*--- allocate input buffer ---*/

buffer = (char *) malloc ( BUFFER_SIZE );
if ( buffer == NULL )
{
    fprintf ( stderr, "Unable to allocate %d bytes\n",
                BUFFER_SIZE );
    return ( EXIT_FAILURE );
}

/*--- check the file header ---*/

bytes_read = fread ( buffer, 1, 4, infile );
if ( bytes_read != 4 )
{
    fprintf ( stderr, "Unable to read %s\n", argv[1] );
    return ( EXIT_FAILURE );
}

if ( buffer[0] == 'R' && buffer[1] == 'L' &&
     buffer[2] == '1' )
        sentinel = buffer[3];
else
{
    fprintf ( stderr, "%s is not an RLE 1 file\n", argv[1] );
    return ( EXIT_FAILURE );
}

/*--- process the file ---*/

while ( 1 )     /* loop until break occurs */
{
    bytes_read = fread ( buffer, 1, BUFFER_SIZE, infile );
    if ( bytes_read == 0 )
        break;

    for ( i = 0; i < bytes_read; i++ )
    {
        if ( buffer[i] != sentinel )
            fputc ( buffer[i], outfile );
        else
        {       /* process sentinel */

            if ( i > bytes_read - 3 ) /* near buffer end */
            {
```

```
if ( i > bytes_read - 2 ) /* no bytes left */
{
    bytes_read = fread ( buffer, 1,
                BUFFER_SIZE, infile );
    if ( bytes_read < 2 )
    {
        fprintf ( stderr,
                  "error in %s\n", argv[1] );
        fclose ( infile );
        unlink ( argv[2] );
        return ( EXIT_FAILURE );
    }
    else
    {
        i = -1;
        goto process_count;
    }
}
else                        /* one byte left */
{
    if ( buffer[i + 1] < 3 )
        goto process_count;

    count = buffer[i + 1];

    bytes_read = fread ( buffer, 1,
                    BUFFER_SIZE, infile );
         if ( bytes_read < 1 )
         {
             fprintf ( stderr,
                       "error in %s\n", argv[1] );
             fclose ( infile );
             unlink ( argv[2] );
             return ( EXIT_FAILURE );
         }

         i = 0;
         do
         {
             fputc ( buffer[i], outfile );
         }
             while ( --count );
     }

 }
 else                       /* not end of buffer */
 {
   process_count:

     count = buffer[++i];

     switch ( count )    /* what's the count? */
     {
         case 0:
             fprintf ( stderr,
                       "error in %s\n", argv[1] );
             fclose ( infile );
```

```
                        unlink ( argv[2] );
                        return ( EXIT_FAILURE );
                    case 1:
                        fputc ( sentinel, outfile );
                        break;
                    case 2:
                        fputc ( sentinel, outfile );
                        fputc ( sentinel, outfile );
                        break;
                    default:
                        i += 1;
                        do
                        {
                            fputc ( buffer[i], outfile );
                        }
                            while ( --count );
                        break;
                }
            }
        }
    }

    if ( bytes_read < BUFFER_SIZE )
        break;
}

fclose ( infile );
fclose ( outfile );

return ( EXIT_SUCCESS );
}
```

就像目前所实现的那样，用于压缩和解压缩的代码极其快速。不过，可以使它变得更快。输出例程必须一次输出一个字符（它使用 fputc()），这导致了缓慢的输出。我们可以加快速度，其方法是：把字节写到内存中的缓冲区中，当缓冲区填满并且到达 EOF 时，就把它写到磁盘。这样，就可以在一个操作中把多个字符写到磁盘，而不是一次写一个字符。事实上，来自 fputc() 的写入将会被编译器库缓冲，但是这些缓冲区倾向于比较小——一般为 512 字节或 1 024 字节。

进一步的改进是：把某种类型的校验和放入压缩文件的头部中。解压缩例程将使用这种校验和确保解压缩文件具有与原始压缩文件相同的内容。在第 10 章中将介绍关于校验和的更多信息。

迄今为止研究的行程编码类型广泛用在业务环境中，因为大多数数据库都分配固定长度的数据字段，比如客户名字、客户地址等。在这种数据库中，将在字段末尾未使用的字节中填充空格。因此，数据库记录为行程压缩提供了理想的环境：在数据库记录中会出现重复空格的序列，并且良好选择的哨兵与实际数据字节发生冲突的可能性很小。当把行程编码应用于数据库时，它通常可以获得优于 50% 的重大压缩效率，并且可以快速、容易地实现它。出于这些原因，对于王安系统（Wang system）上的 ISAM 文件、在 IBM 的 VM/370 copyfile 实用程序的打包选项中以及在用于 BBS 的 ARC 文件压缩实用程序中，它都是作为默认的压缩模式出现的。

有许多其他的方式用于实现行程编码。Microsoft 在具有 .RLE 扩展的 Windows 文件中使用行程编码。在 Microsoft 的实现中，压缩文件（图形文件）包含一些记录，用于指定将显示颜色值的

行，其后接着压缩代码，用于指定要显示的颜色以及应该用多少像素显示它。虽然这种方法抛弃了传统的哨兵代码方法，但它会保留使用代码值，以指示相同值的重复模式。有关 Microsoft 的 RLE 文件的更多信息，可以参考 Tom Swan 所著的关于 Windows 文件格式的书籍［Swan 1994］。

从行程编码可以自然延伸出一种压缩模式，它使用代码来指示重复出现的模式，其中每种模式都大于一个字节。一个例子是为在文件中可能多次出现的整个字符串使用行程编码。9.3 节将讨论这个算法系列。

9.2 霍夫曼压缩

霍夫曼压缩（得名于 D. A. Huffman 的一篇论文［Huffman 1952］）通过检查输入文件并查看哪些字符出现得最频繁来处理数据压缩的问题。霍夫曼压缩把位代码分配给各个字符，为最常见的字符使用较短的位代码，并为不太常见的字符使用较长的位代码。在输出文件的前端，编写一个头部，给每个字符提供对应的位。然后把用于每个输入字符的位代码写到输出文件。解压缩算法读取编码的文件并执行相反的过程，根据头部中的转换信息把位代码转换回原始字符。

霍夫曼的方法使用二叉树把各个字符转换为代码。其思想很简单。考虑一棵二叉树，其中数据项只驻留在叶节点（或外部节点）中。可以选取从树的根到给定叶节点的路径作为用于那个叶节点中的数据的二进制代码，其方法是把左子节点记为 0，并把右子节点记为 1。

例如，图 9-1 中的树为三个字母（h、a 和 t）提供了一种长度可变的编码模式。用于 h 的代码是 00，用于 a 的代码是 01，用于 t 的代码是 1。为了获得这些值，可以从树的根朝着想要的字母行进，并且记录在树中从根向下行进时经过的每条路径。如果向左行进，就添代码中添加一个 0；如果向右行进，就向代码中添加一个 1。因此，h 需要向左边下行两次，从而得到 00。t 要求向右边下行一次，从而得到 1。

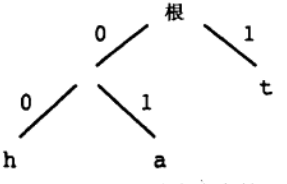


图 9-1 只在叶节点中存储数据的简单二叉树

使用这种无歧义的模式编码较长的消息很直观。例如，可以把 that 编码为 100011（来自于用于字母的 4 个代码——1-00-01-1）。为了解码这些位，只需执行相反的过程即可。从 1 开始，转到右边，并且发现我们到达了一个外部节点，编写其数据 t。同样，00 将把我们带到 h，01 把我们带到 a，最后的 1 将把我们带到终点 t。注意：由于这种模式构建于二叉树上，它可以保证用于每个字母的代码是互斥的。

具有用于输入文件中的每个可能字符的外部节点的任何二叉树都可以用于这种压缩模式，虽然这是显而易见的，但是一些二叉树要好于另外一些二叉树。特别是，我们喜欢使常用字母具有较短的位序列（也就是说，接近于树的顶部），而很少使用的字母应该接近于底部。在图 9-1 中就是这样做的，把 1 放在接近于顶部的位置，并赋予它 1 位的序列，而 h 和 a 由具有 2 位的序列。霍夫曼的算法采用了二叉树的构造，并把它用于创建代码。该算法涉及以下步骤：

1. 扫描输入文件，并统计每个输入字符的出现次数。

2. 将这些符号/计数对视作开始形成的二叉树的外部节点。找出具有最小计数的两个节点，并创建连接这两个节点的父节点。把这个父节点的计数设置为子节点的计数的和。

3. 重复第 2 步，直至所有节点都成为一棵二叉树的一部分。

为了具体解释这个概念，考虑 11 个字符的串 abracadabra。首先，统计每个字母的出现频率。

字母	出现次数
a	5
b	2
c	1
d	1
r	2

接下来，如图 9-2 中所示的那样构造树。在第 1 步中，按顺序放置节点。当两个节点具有相同的计数时，可以先转到任何一个节点。然后，在第 2 步中，开始构造父节点的过程。选取两个具有最小计数的节点（c 和 d），构造一个新的父节点，并给它提供计数 2（c 和 d 的计数之和）。在第 3 步中，我们利用 r 和 b 重复这个过程。此时，有三个空闲的父节点：a，具有计数 5；r 和 b 的父节点，具有计数 4；以及 c 和 d 的父节点，具有计数 2。在第 4 步，为这三个父节点中最小的父节点构造一个父节点，最后构造单个父节点把整棵树绑定在一起。

编码现在很容易：用于 a 的位串是 0，用于 b 的位串是 101，等等。最终的结果如下：

a	b	r	a	c	a	d	a	b	r	a
0	101	100	0	110	0	111	0	101	100	0

或者，把这些位分组进块中，其中每个块包含 4 个位，并把位转换为十六进制值，从而产生如下结果：

0101	1000	1100	1110	1011	000
5	8	C	E	B	0

利用这种方法，就把 11 个字节压缩成 3 个字节。虽然这种节省似乎会给人留下相当深刻的印象，但要记住输出文件还需要包含树结构以便允许解码，并且这些额外的开销字节可能会消耗掉我们节省的一大部分空间。

注意：这棵树将把最常见的字符（a）放在靠近树的顶部，并赋予它尽可能短的位串。所有其他代码的长度都是 3 位。

9.2.1 代码

实现霍夫曼编码所需的代码很直观，分别如程序清单 9-3 ~ 9-5 中的 huffenc. c、huffdec. c 和 huffman. h 所示。每个程序都包含一个测试驱动程序，使得可以清楚看出其用法。当定义了适当的变量时，测试驱动程序还会显示多种内部统计信息和表。

查看压缩例程，如程序清单 9-3 所示，主例程是 HuffEncode( )。这个例程利用其中保存有计数的数组，然后调用 HuffScan( ) 累积计数，调用 HuffBuild( ) 构建树，并且调用 HuffCompress( ) 生成输出文件。霍夫曼树构建在数组 HuffTree [ ] 中，它实际上是一个结构数组。有趣的是，我们可以事先确定这个数组的最大大小。稍加思考，我们就可以清楚知道：具有 N 个外部节点（或叶节点）的树必须具有 $N-1$ 个内部节点。例如，具有两个叶节点的树需要一个父节点。因此，在 huffman. h 中，把 MAXSYMBOLS 定义为 256（用于每个可能的字符），并把 MAXNODES 定义为 MAXSYMBOLS * 2 - 1。然后，在我们的数组中，把前 256 个结构用作树的叶节点，并且第 i 个

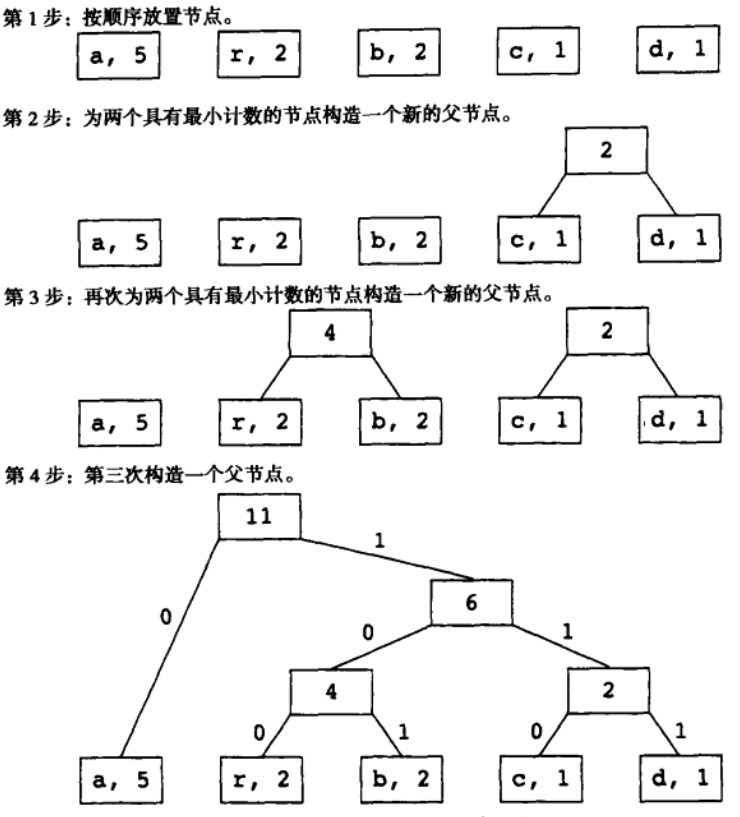


图 9-2 为 abracadabra 构造霍夫曼树

条目对应于其 ASCII 码等于 *i* 的字符。然后使用数组中余下的结构为父（内部）节点构建结构。最后可以简单地把节点之间的链接索引进数组中。

程序清单 9-3 用于霍夫曼编码的例程，包括示例驱动程序

```
/*--- huffenc.c --------------------------- Listing 9-3 ---------
 * Purpose:          Compress an input file, using the Huffman
 *                   encoding technique
 *
 * Entry point:      int HuffEncode (FILE *infile, FILE *outfile)
 *
 *                   Will compress infile into outfile. Both files
 *                   should already be open in binary mode. They
 *                   will be closed by HuffEncode().
 *
 *                   HuffEncode() returns:
 *                       0: success
 *                       1: error while reading in
 *                       2: no data in input file
 *                       3: malloc() failed
 *                       4: error while writing out
```

```
 *
 * Switches:        DRIVER - compiles a test driver
 *                  DUMP - dumps Huffman tree at various points
 *                  SHOWSTATS - provides compression statistics
 *--------------------------------------------------------------*/
#define DRIVER
#define SHOWSTATS
#define DUMP

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "huffman.h"

#if defined(SHOWSTATS)
#define STATS(x) x
CountType HuffBytesHdr; /* no. header bytes in output file */
CountType HuffBytesOut; /* no. data bytes in output file */
#else
#define STATS(x)
#endif

CountType HuffBytesIn; /* count of bytes in input file */

/*
 * The Huffman tree is built in this array. The ith entry
 * coresponds to the symbol i, so we do not explicitly store
 * the corresponding symbol in the array. The child and parent
 * links are also indexes into the array.
 */
struct _HuffTree {
    CountType count;
    LinkType child[2];
    LinkType parent;
    char *hcode;        /* points to bit code */
    int bitcount;       /* number of bits in bit code */
} HuffTree[MAXNODES];

LinkType HuffRoot;      /* tree's root */
LinkType HuffCount;     /* number of internal nodes */
char *HuffTable = NULL; /* table of bit codes */
char *HuffTablePtr;     /* points to free space in table */
unsigned HuffBytes;     /* size of bit code table */

#ifdef DUMP
static void HuffDump ( LinkType n )
{
    LinkType i;

    printf ( "root is %d\n", HuffRoot );
    printf ( "No.  Sym   Count  Parent   "
             "Left  Right  BitNo  Bits\n" );
    for ( i = 0; i <= n; i++ )
    {
        if ( HuffTree[i].count )
        {
            printf( "%3d. ", i );
            if ( i < MAXSYMBOLS )
```

```
            {
                if ( isprint ( i ))
                    printf ( " %c ", i );
                else
                    printf ( "x%.2X", i );
            }
            else
                printf ( "n/a" );
            printf ( " %7ld   %5d  %5d  %5d",
                    HuffTree[i].count,
                    HuffTree[i].parent,
                    HuffTree[i].child[LEFT],
                    HuffTree[i].child[RIGHT] );

            if ( HuffTree[i].hcode )
            {
                int j, k;
                int byte, bit;

                j = HuffTree[i].bitcount;
                printf ( "  %5d  ", j );
                for ( k = 0; k < j; k++ )
                {
                    byte = k / 8;
                    bit = k - byte * 8;
                    if ( HuffTree[i].hcode[byte] &
                                        ( 1 << ( 7-bit )))
                        printf ( "1" );
                    else
                        printf ( "0" );
                }
            }
            printf ( "\n" );
        }
    }
    printf( "\n" );
}

#define DUMPTREE(x) HuffDump(x)
#else
#define DUMPTREE(x)
#endif

/* scan the input file and acquire statistics */

static int HuffScan ( FILE *infile )
{
    int c;
    HuffBytesIn =  0;
    while (( c = fgetc ( infile )) != EOF )
    {
        HuffTree[c].count++;
        HuffBytesIn++;
    }

    if ( ferror ( infile ))
        return ( 1 );
```

```
    else
    {
        rewind ( infile );
        return ( 0 );
    }
}

/* walk the tree, either to get statistics or build bit table */
static void HuffWalk ( LinkType k, int depth )
{
    int dir, bitno, set, byte, bit;
    LinkType w, trace;

    for ( dir = 0; dir < 2; dir++ )
    {
        if ( HuffTree[k].child[dir] == -1 )
        {
            if (dir == LEFT)  /* Left and right are same */
            {
                if ( ! HuffTablePtr )
                    HuffBytes += ( depth + 7 ) / 8;
                else
                {
                    /* second pass, build the codes */
                    HuffTree[k].bitcount = depth;
                    HuffTree[k].hcode = HuffTablePtr;
                    bitno = depth - 1;

                    /* run up the parent links */
                    for ( trace = k, w = HuffTree[k].parent;
                          w;
                          trace = w, w = HuffTree[w].parent )
                    {

                        /* which way did we come? */
                        if ( HuffTree[w].child[LEFT] == trace )
                            set = LEFT;
                        else
                            set = RIGHT;
                        /* record the bit */
                        if ( set )
                        {
                            byte = bitno / 8;
                            bit = bitno - byte * 8;
                            HuffTablePtr[byte] |=
                                        1 << ( 7 - bit );
                        }
                        bitno--;
                    }
                    HuffTablePtr += ( depth + 7 ) / 8;
                }
            }
        }
        else
            HuffWalk ( HuffTree[k].child[dir], depth + 1 );
    }
}
```

```
/* build the internal nodes */
static int HuffBuild ( void )
{

    LinkType i, k, lo1, lo2;

    /* count active nodes */
    for ( i = 0; i < MAXSYMBOLS; i++ )
        if ( HuffTree[i].count )
            HuffCount++;

    /* ensure we have at least two active nodes */
    if ( HuffCount < 1 )
        return ( 2 );
    if ( HuffCount == 1 )
    {
        for ( i = 0; i < MAXSYMBOLS; i++ )
            if ( HuffTree[i].count == 0 )
            {
                HuffTree[i].count++;
                HuffCount++;
                break;
            }
    }

    HuffRoot = MAXSYMBOLS + HuffCount - 2;

    /* build the internal nodes */
    for ( i = MAXSYMBOLS; i <= HuffRoot; i++ )
{
    /* first, find two smallest nodes */
    for ( k = 0; k < i; k++ )
    {
        if ( HuffTree[k].count && !HuffTree[k].parent )
        {
            lo1 = k;
            k++;
            break;
        }
    }

    for ( ; k < i; k++ )
    {
        if ( HuffTree[k].count && !HuffTree[k].parent )
        {
            lo2 = k;
            k++;
            break;
        }
    }

    for ( ; k < i; k++ )
    {
        if ( HuffTree[k].count && !HuffTree[k].parent )
        {
            if ( HuffTree[k].count < HuffTree[lo1].count )
                lo1 = k;
```

```
            else
            if ( HuffTree[k].count < HuffTree[lo2].count )
                lo2 = k;
        }
    }

    /* now, build the new node and update the tree */
    HuffTree[i].count = HuffTree[lo1].count +
                        HuffTree[lo2].count;
    HuffTree[i].child[LEFT]  = lo1;
    HuffTree[i].child[RIGHT] = lo2;
    HuffTree[lo1].parent = HuffTree[lo2].parent = i;
}

/* build the bit codes */
HuffBytes = 0;
HuffTablePtr = NULL;
HuffWalk ( HuffRoot, 0 );

HuffTable = (char *) malloc ( HuffBytes );
    memset ( HuffTable, 0, HuffBytes );
    if (! HuffTable )
        return ( 3 );
    HuffTablePtr = HuffTable;
    HuffWalk ( HuffRoot, 0 );

    return ( 0 );
}

/* create the output file */
static int HuffCompress ( FILE *infile, FILE *outfile )
{
    LinkType i;
    struct _Header header;
    int outchar, outbit, c;
    char sig[] = SIGNATURE;

    STATS ( HuffBytesHdr = 0 );
    STATS ( HuffBytesOut = 0 );

    /* signature bytes */
    fwrite ( sig, strlen ( sig ) + 1, 1, outfile );
    STATS ( HuffBytesHdr += strlen ( sig ) + 1 );

    /* the root pointer */
    fwrite ( &HuffRoot, sizeof ( HuffRoot ), 1, outfile );
    STATS ( HuffBytesHdr += sizeof ( HuffRoot ));

    /* the number of internal nodes */
    fwrite ( &HuffCount, sizeof ( HuffCount ), 1, outfile );
    STATS ( HuffBytesHdr += sizeof ( HuffCount ));

    /* the character count */
    fwrite ( &HuffBytesIn, sizeof ( HuffBytesIn ), 1, outfile );
    STATS ( HuffBytesHdr += sizeof ( HuffBytesIn ));

    /* the active nodes */
```

```
    for ( i = 0; i <= HuffRoot; i++ )
    {
        if ( HuffTree[i].count )
        {
            header.index = i;
            header.child[0] = HuffTree[i].child[0];
            header.child[1] = HuffTree[i].child[1];
            fwrite ( &header, sizeof ( header ), 1, outfile );
            STATS( HuffBytesHdr += sizeof ( header ));
        }
    }
    /* now, compress the input file */
    outchar = 0; /* build up output bytes here */
    outbit = 0;
    while (( c = fgetc ( infile )) != EOF )
    {
        char *s;
        int k, count, byte, bit, set;

        s = HuffTree[c].hcode;
        count = HuffTree[c].bitcount;

        /* translate character into bit codes */
        for ( k = 0; k < count; k++ )
        {
            byte = k / 8;
            bit = k - byte * 8;
            if ( s[byte] & ( 1 << ( 7-bit )))
                set = 1;
            else
                set = 0;

            if ( set )
                outchar |= 1 << ( 7 - outbit );
            outbit++;
            if ( outbit == 8 )
            {
                fputc ( outchar, outfile );
                outchar = 0;
                outbit = 0;
                STATS ( HuffBytesOut++ );
            }
        }
    }

    /* do the last byte, if necessary */
    if ( outbit )
    {
        fputc ( outchar, outfile );
        STATS ( HuffBytesOut++ );
    }

    if ( ferror ( infile ))
        return ( 1 );
    if ( ferror ( outfile ))
        return ( 4 );

    return ( 0 );
}
```

```
int HuffEncode ( FILE *infile, FILE *outfile )
{
    int retval;
    LinkType i;

    /* set all counts to zero */

    HuffRoot = 0;
    HuffCount = 0;
    memset ( HuffTree, 0, sizeof ( HuffTree ));
    for ( i = 0; i < MAXSYMBOLS; i++)
    {
        HuffTree[i].child[LEFT]  = -1;
        HuffTree[i].child[RIGHT] = -1;
    }

    /* do frequency counts */
    if ( retval = HuffScan ( infile ))
        goto done;

    /* build the tree */
    if ( retval = HuffBuild() )
        goto done;

    DUMPTREE ( HuffRoot );

    /* compress the data file */
    retval = HuffCompress ( infile, outfile );

    #if defined(SHOWSTATS)
    if ( ! retval )
    {
        printf ( "The input file contained %lu bytes\n",
                    HuffBytesIn );
        printf ( "The output file contained %lu header bytes "
                 "and %lu data bytes\n",
                    HuffBytesHdr, HuffBytesOut );
        printf ( "Output file %lu%% the size of input file\n",
                    (( HuffBytesHdr + HuffBytesOut ) * 100 ) /
                       HuffBytesIn );
    }
    #endif

  done:
    fclose ( infile );
    fclose ( outfile );
    return ( retval );
}
/*-------------------------------------------------------------
 * The following driver for the previous routines is active
 * if DRIVER is defined.
 *-------------------------------------------------------------*/
#ifdef DRIVER
int main ( int argc, char *argv[] )
{
    FILE *infile, *outfile;
    int retval;
```

```
    if ( argc != 3 )
    {
        fprintf( stderr, "Usage: huffenc infile outfile\n" );
        return ( EXIT_FAILURE );
    }

    infile = fopen ( argv[1], "r+b" );
    if ( infile == NULL )
    {
        fprintf ( stderr, "can't open %s for input\n", argv[1] );
        return ( EXIT_FAILURE );
    }

    outfile = fopen ( argv[2], "w+b" );
    if ( outfile == NULL )
    {
        fprintf ( stderr, "can't open %s for output\n", argv[2]);
        return ( EXIT_FAILURE );
    }

    if ( retval = HuffEncode ( infile, outfile ))
    {
        printf( "compression failed: " );
        if ( retval == 1 )
            printf ( "input error\n" );
        else
        if ( retval == 2 )
            printf ( "empty tree\n" );
        else
        if ( retval == 3 )
            printf ( "malloc failed\n" );
        else
        if ( retval == 4 )
            printf ( "output error\n" );
        else
            printf("\n");
        return  ( retval );
    }
    else
    {
        printf ( "%s was compressed into %s\n",
                                        argv[1], argv[2] );
        return ( EXIT_SUCCESS );
    }
}
#endif
```

现在看看做主要工作的三个例程，第一个例程 HuffScan( ) 读取输入文件，并为它读取的每个字符递增计数。它执行该操作是为了确定每个字符在输入文件中出现的频率，以便具有用于构建代码树的计数。

第二个例程 HuffBuild( ) 稍微复杂一点。它首先统计其计数不是 0 的节点数。一旦确定了这个数量，就可以直接计算所需的父节点的数量。然后，该例程将通过多次遍历节点并在每次遍历过程中找出最小的计数对，来构建父节点。这种扫描总是开始于外部节点，然后转到内部（父）

节点。

下一个问题是为每个字符计算位代码。我们首先递归地遍历树（参见 HuffWalk()），并统计每个外部节点的深度。这个深度将舍入到最接近的 8 的倍数，从而提供方便地对每个字符进行编码所需的字节数，它开始于一个字节界限。然后，再次遍历树，这一次将计算用于每个字符的代码，并将其存储在缓冲区中。这第二次遍历针对的是树中的父链接；它使我们很容易解开遍历路径，并计算所需的位串。在大小合适的缓冲区中为每个字符构建代码的方法是必需的，因为我们不能事先预测最长的代码序列的长度。这种方法尝试使用 16 位无符号整数存储代码，但是如果树给代码分配了 17 位的序列，那么它就会失败。同样，使用无符号长整型也可能会失败，尽管它不太容易失败。最后一个例程 HuffCompress( ) 产生一个输出文件，它首先写出所需的信息以重构树，然后写出与每个输入字符对应的位序列。

解压缩是在 huffdec. c 中实现的，如程序清单 9-4 所示。在读取了头部信息之后，例程将读取位，并追溯它们到其对应的外部节点。注意：要转换的字节数是作为头部信息的一部分发送的；如果没有该信息，解压缩例程将不知道何时停止。文件末尾不能用作停止命令，因为最后一个字节可能只包含少数几个信息位。

程序清单 9-4　带有示例驱动程序的霍夫曼解码例程

```
/*--- huffdec.c ------------------------- Listing 9-4 ---------
 * Purpose:          Uncompress a Huffman-compressed file
 *
 * Entry point:      int HuffDecode(FILE *infile, FILE *outfile)
 *
 *                   Will uncompress infile into outfile. Both
 *                   files should already be open in binary mode.
 *                   They will be closed by HuffDecode().
 *
 *                   HuffDecode() returns:
 *                       0: success
 *                       1: invalid signature byte
 *                       2: invalid header bytes
 *                       3: invalid data bytes
 *
 *
 * Switches:         DRIVER - compiles a test driver
 *                   DUMP - dump the tree after loading
 *-------------------------------------------------------------*/
#define DRIVER

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "huffman.h"

/* a simple tree */
struct _HuffTree {
    LinkType child[2];
} HuffTree[MAXNODES];

CountType HuffBytesIn;  /* number of bytes to decode */
```

```
LinkType HuffRoot;      /* the tree's root */
LinkType HuffCount;     /* the number of internal nodes */

#ifdef DUMP
static void HuffDump ( LinkType n )
{
    LinkType i;

    printf ( "root is %d\n", HuffRoot );
    printf ( "No.  Sym  Left  Right\n" );
    for ( i = 0; i <= n; i++ )
    {
        printf( "%3d. ", i );
        if ( i < MAXSYMBOLS )
        {
            if ( isprint ( i ))
                printf( " %c ", i );
            else
                printf ( "x%.2X", i );
        }
        else
            printf ( "n/a" );

        printf ( " %5d  %5d",
                HuffTree[i].child[LEFT],
                HuffTree[i].child[RIGHT] );

        printf ( "\n" );
    }
    printf ( "\n" );
}
#define DUMPTREE(x) HuffDump(x)
#else
#define DUMPTREE(x)
#endif

/* the actual decompression routine */
int HuffDecode ( FILE *infile, FILE *outfile )
{
    int retval = 0;
    LinkType i;
    char buffer[10];
    struct _Header header;
    CountType bytesout;
    int inbyte, bitno, mask, set;

    /* set all counts to zero and initialize symbols */
    memset ( HuffTree, 0, sizeof ( HuffTree ));

    /* check the signature */
    fgets ( buffer, strlen ( SIGNATURE ) + 2, infile );
    if ( strcmp ( buffer, SIGNATURE ))
    {
        retval = 1;
        goto done;
    }
```

```
    /* read in header bytes */

    fread ( &HuffRoot, sizeof ( HuffRoot), 1, infile );
    fread ( &HuffCount, sizeof ( HuffCount), 1, infile );
fread ( &HuffBytesIn, sizeof ( HuffBytesIn), 1, infile );

for ( i = 0;
        i < HuffCount + ( HuffRoot - MAXSYMBOLS + 1 ); i++)
{
    fread ( &header, sizeof ( header ), 1, infile );
    HuffTree[header.index].child[0] = header.child[0];
    HuffTree[header.index].child[1] = header.child[1];
}

if ( ferror ( infile ))
{
    retval = 2;
    goto done;
}
DUMPTREE ( HuffRoot );

/* now, translate the data bytes */
bitno = 8;
for ( bytesout = 0; bytesout < HuffBytesIn; bytesout++ )
{
    /* walk down the tree */
    i = HuffRoot;
    for ( ;; )
    {
        /* do we need a new input byte? */
        if ( bitno > 7 )
        {
            inbyte = fgetc ( infile );
            if ( inbyte == EOF )
            {
                retval = 3;
                goto done;
            }
            bitno = 0;
            mask = 0x80;
        }

        /* test the current bit */
        if ( inbyte & mask )
            set = 1;
        else
            set = 0;
        bitno++;
        mask >>= 1;

        /* walk down the tree */
        i = HuffTree[i].child[set];
            /* are we there yet? */
            if ( HuffTree[i].child[0] == -1 )
            {
                /* at the bottom -- write the character */
                fputc ( i, outfile );
```

```
                break;
            }
        }
    }

  done:
    fclose ( infile );
    fclose ( outfile );
    return ( retval );
}

#ifdef DRIVER
int main ( int argc, char *argv[] )
{
    FILE *infile, *outfile;
    int retval;

    if ( argc != 3 )
    {
        fprintf ( stderr, "Usage: huffdec infile outfile\n" );
        return ( EXIT_FAILURE );
    }

    infile = fopen ( argv[1], "r+b" );
    if ( infile == NULL )
    {
        fprintf ( stderr, "can't open %s for input\n", argv[1] );
        return ( EXIT_FAILURE );
    }

    outfile = fopen ( argv[2], "w+b" );
    if ( outfile == NULL )
    {
        fprintf ( stderr, "can't open %s for output\n", argv[2]);
        return ( EXIT_FAILURE );
    }

    if ( retval = HuffDecode ( infile, outfile ))
    {
        printf ( "decompression failed: " );
        if ( retval == 1 )
            printf ( "invalid signature byte\n" );
        if ( retval == 2 )
            printf ( "read error in header\n" );
        if ( retval == 3 )
            printf ( "read error in data\n" );
        else
            printf( "\n" );
        return ( retval );
    }
    else
    {
        printf ( "%s was expanded into %s\n",
                                        argv[1], argv[2]);
        return ( 0 );
    }
}
#endif
```

明示常量和数据结构定义在头文件 huffman.h 中，程序清单 9-5 中显示了它。

程序清单 9-5 用于 huffenc.c 和 huffdec.c 的头文件

```
/*--- Listing 9-5 ------------------------ huffman.h ---------*
 * Header file for Huffman compression routines                *
 *-------------------------------------------------------------*/

/* Max number of unique symbols or codes */
#define MAXSYMBOLS 256

/* Total number of external and internal nodes */
#define MAXNODES   (MAXSYMBOLS*2-1)

/* Child directions */
#define LEFT  0
#define RIGHT 1

typedef unsigned long CountType;
typedef int LinkType;

/* Header byte structure */
struct _Header {
    LinkType index;
    LinkType child[2];
};

#define SIGNATURE "Huff1"
```

该代码的一个麻烦的方面是：将重复出现操纵位的需求。如果你能仔细研究代码，确保你熟悉 C 语言的位操作。

9.2.2 其他问题

霍夫曼编码极其有效。在它最初出现之后的许多年里，霍夫曼压缩被认为是最有效的通用压缩算法。今天，滑动窗口压缩技术（比如在下一节中讨论的那些技术）享有这种荣誉，尽管在许多情况下霍夫曼编码仍然被证明是出众的算法。霍夫曼编码的两个方面不利地影响了它的结果。就压缩效率而言，同时包括树和文件的要求意味着霍夫曼编码对于较小的文件不是非常有效；树以及相关的头部信息所消耗的空间远远超过了压缩节省的空间。就性能而言，霍夫曼编码必须读取输入文件两次。对于较小的文件，这几乎不会引起什么问题。不过，当压缩大于 1 MB 的文件时，差别可能是显而易见的。

人们为什么更喜欢选择霍夫曼编码（而不是行程编码）作为常规的压缩算法呢？这是由于它不需要相同字符的顺串以便压缩文件。它只需要输入文件中字符的不均匀分布。由于情况几乎总是这样，霍夫曼编码对于几乎所有的文件都工作得很好。

可以优化以前的程序清单中展示的霍夫曼编码的实现，以便获取更好的性能。有两项任务花费了它的大部分时间：读取输入文件两次和构建树。为了改进读取性能，使用缓冲区更可取。无需从磁盘读取输入文件两次，可以一次把它读入缓冲区中，并在那里处理它。特别是在具有相当多的内存并且几乎或者完全没有内存分页损失的系统上，这种方法工作得非常好。例如，扩展的

DOS 程序可以在无需进行内存分页的情况下分配若干兆字节的缓冲区。这种程序应该将文件中尽可能多的内容读入缓冲区中。

可以用多种方式来节省在遍历计数以创建位代码的树时所做的工作。一种方式是具有单独的步骤（比如排序），用于在构造树之前对计数进行排序。Bentley［Bentley 1995］讨论了用于解决该问题的一些方法。

迄今为止，我们讨论了两种压缩算法，它们查看单个字节，并使用它们的重现次数（RLE）或者它们的出现频率（霍夫曼编码）来压缩文件。下一个算法系列（即滑动窗口技术）使用像字符串这样的系列字节作为压缩单元。

9.3 滑动窗口压缩

在本章的第一节中，我们解释了行程编码（RLE）的基础知识，它是一种简单的压缩技术，用于检测多次连续出现的相同字符，并把重复的字符编码为单个代码。不过，RLE 将不会对像 invitees were invited with involved invitations 这样的文本提供压缩。这个示例包含出现了三次的字符串 invit。这三个字符串应该是可以压缩的，但是它们将会被 RLE 忽略。

由 Jacob Ziv 和 Abraham Lempel 提出的算法系列［Ziv 1977］依靠**基于字典**（dictionary-based）的技术来处理这种问题。将包含输入文件中的字符串的字典构建成输入文件，并读入它以进行压缩。如果输入文件中的字符串与字典中的条目匹配（意味着字符串已经出现在输入文件中），就会把指向前一个条目的简短代码——一般由 2 个字节组成——写到压缩文件中。在解压缩期间，解码引擎将构建一个类似的字符串的字典，并且无论何时发现压缩代码，都会把它们写到解压缩文件。

暂且不讨论构造字典的方式，让我们检查压缩代码。在所有的压缩机制中，都要花费相当多的工作使压缩代码尽可能短。压缩代码越长，原始数据就必须越长，以便实现数据压缩。LZ 算法一般使用 2 个字节来编码必要的数据。2 个字节提供了 16 位的有用数据。典型的 LZ 编码模式将使用 12 位来指示将引用字典中的哪个字符串，并使用 4 位来指示字符串在多大程度上是重复的。对字典的 12 位引用意味着字典可以具有多达 4 096 个条目（编号为 0 ~ 4095，因为 4095 是可以在 12 位中编码的最大数字）。4 位的字符串长度指示在字典的字符串中有多少字节与当前文本匹配。例如，如果字典条目 33 保存字符串 to be or not to，并且当前输入字符串是 to be orderly at work，那么两个字符串的前 8 个字符将匹配。因此，LZ 代码将使用前 12 位来包含字典条目（33），并使用 4 位来指示匹配的长度（8）。通过这种方法，2 个字节的 LZ 代码将表示 8 个输入字符。

理论上讲，在这种实现下，字符串不能匹配 15 个以上的字符，因为这是可以在 4 位中表示的最大数量。我们稍后将研究的技术允许把这种限制扩展到 18 个字符。

将把没有出现在字典中的字符串添加到字典中，并传递给输出文件进行解压缩。解码机制需要某种方式来获知它正在检查的字节是 LZ 代码，还是未压缩的数据。用于处理这个问题的传统方法是让编码器在 8 个字符或压缩代码的每个序列的开头发出单个字节（称为**标志字节**（flag byte））。这个字节包含 8 位，如果对应的元素是 LZ 代码，就把一位设置为 0；如果对应的元素是未压缩的字符，就把一位设置为 1。因此，如果 8 个项目的序列包含：LZ 代码，e，l，e，p，h，LZ 代码，!，其中逗号之间的斜体字符表示未压缩的字母，对应的字节将具有二进制值 01111101（参

见图9-3，它演示了标志字节与数据元素之间的对应关系的另一个示例）。

这种信息给我们提供了标准LZ压缩模式下可用的最大和最小压缩。如果由于没有字符串重复而使得不可能进行压缩，输出文件将需要为每个字符使用9位（8位用于字符本身，并在标志字节中把1位设置为1）。因此，对于每8位的数据，解压缩文件将增大1位——实际上相当于原始文件大小的112.5%。在最好情况下，其中每18个字节的输入与字典中的字符串匹配，它需要用17位（16位用于LZ代码，1位用于标志字节）编码18字节（144位）的字符串。因此，利用每144位的数据需要17位编码的比率，LZ技术可以获得的最大压缩是其大小为原始文件大小的11.8%的文件。这两个数字都是极限值，在实际中极少发生。注意：最佳情况并不像所说明的那样好，因为在第一遍读入时，18字节的字符串将原封不动地写到输出文件，这是由于它在字典中还不存在。本章的下一节将介绍基于LZ的字典压缩。

自从它最初被提出起，LZ字典的实现已经经历了巨大的变化。在原始著作中，它基于二叉树，在尝试查找匹配时，每次都会从根开始查找它。LZ以后的版本利用了散列表。甚至最近的实现也没有使用像这样的固定字典。它们代之以在最后4 096个字节内寻找字符串的匹配。如果在前4 096个字节内找到了当前字符串的匹配，代码中12位的字段将保存距离字符串的偏移量，而不是指向表项的引用。这种方法简化了解码机制，因为它不必构建或维护字典。它代之以只是简单地在内存中保存解压缩数据的最后4 096个字节，并在解压缩文件时参考该数据。在解压缩了每个字节之后，将向前移动4 096个字节的窗口。这就是为什么把这个算法系列称为**滑动窗口压缩**（sliding-window compression）的原因。

在Microsoft的分发磁盘上经常可以见到利用滑动窗口压缩技术压缩的文件。这些文件通常具有由两个字符其后接着一个下划线组成的扩展名（这样一个文件可能是filename.ex_）。为了压缩这些文件，可以使用Microsoft的compress.exe实用程序。然后，可以使用Microsoft的expand.exe程序或者程序清单9-6中所示的程序（mslzunc.c）解压缩它们。

程序清单9-6　用于解压缩Microsoft的LZ压缩文件的程序

```
/*--- mslzunc.c -------------------------- Listing 9-6 --------
 *  Uncompress a file compressed with Microsoft's compress.exe
 *  program. compress.exe uses a Microsoft variant of the LZ
 *  family of sliding-window compresssion algorithms.
 *
 *  if DRIVER is #defined a driver mainline is compiled.
 *-------------------------------------------------------------*/

#define DRIVER

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mslzunc.h"

/* 4KB Uncompression window */
char Window[WINSIZE];

/*-------------------------------------------------------------
 * Principal routine that uncompresses the input data
 * using Microsoft's own implementation of LZ algorithms.
 *-------------------------------------------------------------*/
```

```
void UncompressData ( FILE *infile, FILE *outfile,
                      unsigned long uncomp_size )
{

    unsigned char   bit_map, byte1, byte2;
    int             length, counter;
    long            location, curr_pos = 0L;

    /* Init our window to spaces */
    memset ( Window, ' ', WINSIZE );

    /* Go through until done */
    while ( curr_pos < uncomp_size )
    {

        /* Get bit_map of flag indicating codes and bytes */
        bit_map = fgetc ( infile );
        if ( feof ( infile ))
            return;

        /* Go through and decode data */
        for ( counter = 0; counter < 8; counter++ )
        {

            /* It's a code, so decode it and copy the data */
            if ( ! BITSET ( bit_map, counter ))
            {
                byte1 = fgetc ( infile );

                /* Shouldn't be EOF, but just in case... */

                if ( feof ( infile ))
                    return;

                byte2 = fgetc ( infile );
                length = LENGTH ( byte2 );
                location = OFFSET ( byte1, byte2 );

                /* Copy data from 'window' */
                while ( length != 0 )
                {
                    byte1 = Window[WRAPFIX(location)];
                    Window[WRAPFIX(curr_pos)] = byte1;
                    fputc ( byte1, outfile );
                    curr_pos++;
                    location++;
                    length--;
                }
            }
            else    /* it's just a data byte */
            {
                byte1 = fgetc ( infile );
                Window[WRAPFIX(curr_pos)] = byte1;
                fputc ( byte1, outfile );
                curr_pos++;
            }
```

```
            if ( feof ( infile ))
                return;

        }
    }
}
/*-------------------------------------------------------------
 * Verifies the header of the compressed file looking for the
 * Microsoft signature and the size of the uncompressed file.
 * Returns the latter upon success or 0L upon failure.
 *-------------------------------------------------------------*/
unsigned long VerifyHeader ( FILE *infile )
{

    COMPHEADER header;
    unsigned long comp_size;

    fseek ( infile, 0, SEEK_END );
    comp_size = ftell ( infile );
    fseek ( infile, 0, SEEK_SET );
    fread ( &header, sizeof ( header ), 1, infile );
    if (( header.Magic1 != MAGIC1 ) ||
        ( header.Magic2 != MAGIC2 ))
    {
        fprintf ( stderr, "input file is not a valid "
                          "compressed file!\n" );
        return ( 0L );
    }

    printf ( "Uncompressing file from %lu bytes to %lu bytes\n",
             comp_size, header.UncompSize );

    return ( header.UncompSize );
}

#ifdef DRIVER
/*-------------------------------------------------------------
 * Driver to exercise previous decompression routines. Accepts
 * exactly two command-line arguments: the name of the compressed
 * file and the name to use for the output compressed file.
 *-------------------------------------------------------------*/
int main ( int argc, char *argv[] )
{

    FILE *infile, *outfile;
    unsigned long uncompressed_size;

    if ( argc != 3 )
    {
        fprintf ( stderr, "Usage: mslzunc"
                  "compressed-file uncompressed-file\n" );
        return ( EXIT_FAILURE );
    }
    if (( infile = fopen ( argv[1], "rb" )) == NULL)
    {
        fprintf( stderr, "Can't open %s for input\n", argv[1] );
```

```
        return ( EXIT_FAILURE );
    }

    if (( outfile = fopen ( argv[2], "wb" )) == NULL )
    {
        fprintf ( stderr, "Can't open %s for output\n", argv[2]);
        return ( EXIT_FAILURE );
    }

    uncompressed_size =  VerifyHeader ( infile );
    if ( uncompressed_size != 0L )   /* no error occurred */
        UncompressData ( infile, outfile, uncompressed_size );

    fclose ( infile );
    fclose ( outfile );

    if ( uncompressed_size == 0L )  /* previous error occurred */
        return ( EXIT_FAILURE );
    else
        return ( EXIT_SUCCESS );
}
#endif
```

这个程序（基于 Davis 的著作［Davis 1994］）显示了滑动窗口压缩的一些细节，以及 Microsoft 自己的一些扩展名。这个程序的代码需要做一点解释。驱动程序例程（main( )）接受两个参数：压缩文件的名称和要生成的解压缩文件的名称。在打开文件之后，程序将使用 VerifyHeader ( ) 验证压缩文件中的头部信息。该头部定义在结构 COMPHEADER 中，它是在程序清单 9-7（mslzunc. h）中声明的。这个头部包含一个签名，它包含两个长整数（32 位整数），其值出现在 MAGIC1 和 MAGIC2 中，其后接着值为 0x41 的单个字节。接下来是一个字节，其中包含原始文件名中被压缩程序用_替换的字符。为了重构原始文件名，可以使用这个字符（称为 FileFix）替换输入文件名中最后的下划线。头部中的下一个（即最后一个）字段是 UncompSize，它是原始文件的长度（以字节为单位），并且它应该等于生成的解压缩文件的长度。在验证了头部中的这些神奇的值并且注明了输出文件的大小之后，就调用 UncompressData( ) 开始处理文件本身。

函数首先读取标志字节，它的 8 位对应于文件中后 8 个元素的状态。在 Microsoft 的实现中，把位设置为 1 就意味着对应的元素是一个未压缩的字节，其输出应该保持不变。如果把位设置为 0，对应的元素就是 2 字节的压缩代码，稍后将讨论它的准确内容。在标志字节中，最低有效位（位于最右边）给出了接下来的一个元素的状态。因此，值为 0x93 的标志字节将是 8 个元素（10010011）的标志，这意味着（从最右边的标志位开始）接下来的两个元素将是未压缩的字节，其后接着两个压缩代码、一个未压缩的字节、一个压缩代码和最后一个未压缩的字节。图 9-3 演示了这个示例。

对这些值进行的测试是通过头文件中的 BITSET 宏完成的，如程序清单 9-7 所示。

如以前所解释的，压缩代码占据 16 位。在这 16 位中，第二个字节中的最低的 4 位表示替换字符串的长度，而较高的 12 位表示距离解压缩的输出文件中的当前位置的偏移量。

4 个长度位可以保存 0 ~ 15 之间的值。由于压缩代码本身占据 2 字节，长度的切合实际的最小值是 3 字节，因为小于 3 字节的字符串将不会被压缩。因此，0 字节的编码长度表示切合实际的最小长度，即 3 字节。把长度值加上 3 意味着现在可以表示的最长字符串是 18 字节。程序清单 9-7

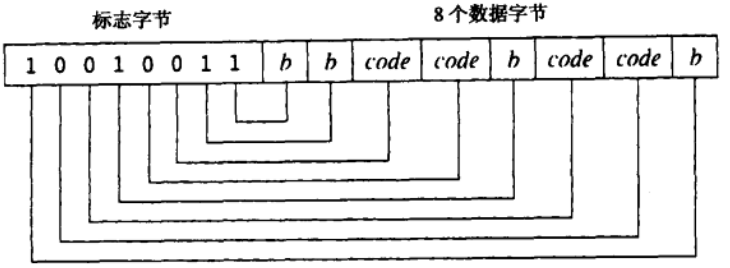


图 9-3 标志字节与数据元素之间的对应关系（b = 未压缩的字节；code = 2 字节的压缩代码）

中的宏 LENGTH 显示了将从代码字节中提取并且会相应地进行调整的长度。

指向替换字符串的偏移量包含在压缩代码的剩余 12 位中。程序清单 9-7 中的宏 OFFSET 用于提取数据。一旦确定了位置值，就把它加上 16。这最后一个值显示了必须在最后 4096 个字节的输出窗口中行进多远，以选取替换字符串。

程序清单 9-7 mslzunc.c 的位运算符和头部定义

```
/*--- mslzunc.h ---------------------------- Listing 9-7 --------
 * Values and bit-manipulation macros used in mslzunc.c
 *-----------------------------------------------------------------*/

typedef struct tagCOMPHEADER {
   long    Magic1;
   long    Magic2;
    char    Is41;     /* 0x41 */
    char    FileFix; /* character saved for -r option */
    long    UncompSize;
} COMPHEADER;

/* Microsoft's magic numbers in header. Magic1 = "SZDD" */

#define MAGIC1 0x44445A53
#define MAGIC2 0x3327F088

/* Constants and macros for uncompression */

#define WINSIZE  4096
#define LENGTH(x) ((((x) & 0x0F)) + 3)
#define OFFSET(x1, x2) \
    ((((x2 & 0xF0) << 4) + x1 + 0x0010) & 0x0FFF)
#define WRAPFIX(x)      ((x) & (WINSIZE - 1))
#define BITSET(byte, bit)  (((byte) & (1<<bit)) > 0)
```

Microsoft 的实现期望将窗口初始化为空格。这很重要，因为窗口被感知为长度为 4096 字节的循环队列。例如，如果出现在字节 0 ~ 10 中的字符串将会重复出现，就在其前面放置 4 个空格，偏移量将开始于字节 4092。通过指向 4092，偏移量将指向字节 4092 ~ 4095 中的 4 个空格，以及位置 0 ~ 10 中的字符串。程序清单 9-4 中显示了另一个示例，其中刚刚把字符串“Who is”放在窗口中。读取的下一个字符串是“Who is”——字节 0 ~ 5 中存在的相同字符串，但前置有一个空格。由于 0 以前的字节包含空格，并且引用可以环绕它（4095 之后的下一个字节是字节 0），引用

就指向字节 4 095 中的空格。这种环绕是由程序清单 9-7 中的宏 WRAPFIX 处理的。

（那些使用 UNIX 或者不能访问 Microsoft 的压缩程序的读者将在可以为本书获得的代码磁盘上找到源代码实现）。

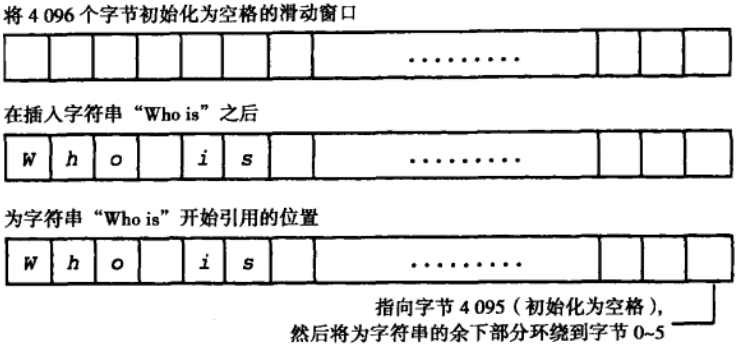


图 9-4 Microsoft 的滑动窗口如何环绕

9.4 基于字典的压缩（LZW）

关于滑动窗口压缩的上一节内容介绍了基于字典的压缩的概念，特别是 Lempel 和 Ziv 在 LZ 算法系列中清楚解释的那些概念。在继续阅读下面的内容之前，应该先阅读这段介绍。

Terry Welch（在优利系统公司任职的计算机科学家）对 LZ 基于字典的算法添加了一个重大的改进。他观察到：如果预先加载字典，它为字符集中的每个字符都具有一个条目，就会带来多个优点。第一，由压缩例程生成的输出可以只包含与压缩字典中的条目对应的代码。第二，不需要把字典本身作为压缩文件的一部分进行传送。作为替代，编码和解码例程开始于相同的 256 个字符的基本字典，并且会在它们处理任务时填满字典。我们稍后将看到这些优点为什么很重要。

通过把 LZ 算法命名为 LZW 来表达对 Welch 所做的增强工作的敬意，本章余下部分中将讨论该技术。不过，注意：在你自己的工作中使用该算法之前，需要获得优利系统公司的准许。LZW 受到优利系统公司的多项专利的保护，该公司一再保护这些专利免受侵犯。要小心使用 LZW。

编码和解码机制所做的第一件事是：占据表中的前 256 个槽，其中每个表项分别用于机器的字符集中的 256 个字符中的一个字符。如前所述，预先加载字典意味着没有不能立即转换为字典代码的输入字符。最初，这意味着利用输出代码 0 替换输入字符 0，这不会给人留下非常深的印象。不过，压缩例程将基于看到的字符序列把条目添加到字典中，并且这些条目很快就会允许把多个字符的输入序列压缩成单个代码输出序列。注意：使用字典条目时要求使用整个条目。与滑动窗口不同，没有使用部分条目的规定。

为了查看这个过程如何工作，让我们压缩短语 AT AN ATAN（图 9-5）。第一个字符的处理很平常：它只是作为字典中预先存在的对应代码的输出。同样，第二个字符是作为它自身的输出。不过，即使在输出方面没有任何有趣的事情发生（输入两个字符，输出两个代码），字典也正在开始成形。现在看到了两个字符的序列“AT”之后，就把新条目制作成字典：代码 256 现在对应两个字符的短语 AT。后面的空格、A 以及 N 的处理同样令人索然无味，但是每个输入字符现在

都会驱动向字典中添加新的短语。这个短语总是包含最后输出的短语以及连接到它的新字符。

输入	输出	字典动作
A	65	无动作。代码在字典中
T	84	为“AT”添加代码256
空格	32	为“T”添加代码257
A	65	为“A”添加代码258
N	78	为“AN”添加代码259
空格	参见下面的内容	
A	258	“A”是条目258。为“N”添加代码260
T	84	为“AT”添加代码261
A	参见下面的内容	
N	259	“AN”是条目259。为“TA”添加代码262

图9-5 “AT AN ATAN”的压缩

当我们第二次到达“A”时，就会发生一些有趣的事情，因为我们具有的输出代码将不是预定义的256种代码之一。代码258以前被定义为“A”，并且我们现在使用它来替换输入字符串中的“A”。余下的压缩过程是相似的，并且使用代码259替换输入字符串中最后的“AN”。

在这个示例中，我们把10个字符压缩成8个代码。由于代码将需要大于8位的输入字符（常见的代码大小是12位和16位），这个压缩过程实际上很可能增加这个非常短的输入文件的大小。不过，对于更大的输入文件，LZW算法将有机会在字典中构建大得多的短语，并且开始产生非常好的压缩。

9.4.1 LZW算法的伪代码

LZW算法从概念上讲很简单。在伪代码中，压缩算法看起来如下所示：

```
OldString = GetFirstCharacter();
while (input exists)
{
     NewChar = GetNewCharacter()
     NewString = OldString + NewChar;
     if (InDictionary(NewString))
          OldString = NewString;
     else
     {
          OutputCode(OldString);
          AddToDictionary(NewString);
          OldString = NewChar;
     }
}
OutputCode(OldString); /* last data code */
OutputCode(EndOfFileCode);
```

解压缩同样很简单：

```
OldString = GetACode(); /* first code is itself */
OutputString(OldString);
while ((NewCode = GetACode()) != EndOfFileCode) {
     NewString = Lookup(Newcode);
     OutputString(NewString);
     Append 1st character of NewString to OldString;
```

```
        AddToDictionary(OldString);
        OldString = NewString;
    }
```

虽然这看起来很简单，但是在解压缩期间可能逐渐显示出一个隐性的问题。设 c 是一个字符，并且设 S 是任何长度 >0 的字符串。如果字典包含一个 cS 形式的条目作为代码 k，那么 cScSc 形式的输入序列将导致压缩例程输出代码 k 以响应初始的 cS，然后在内部为 cSc 创建代码 k'。并且，由于下一个序列是 cSc，在为解压缩例程明确定义它之前将使用代码 k'（图 9-6）！

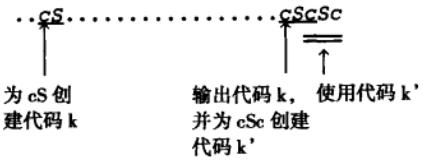


图 9-6 在某种情况下，LZW 压缩例程可能输出不为解压缩例程所知的代码

不过，幸运的是，只在这种情况下才可能发生这种事情，并且解压缩例程可以轻松地处理这个问题：如果看到未知的代码，可以轻松地把它导出为与前一个代码对应的字符串的值，并且把那个字符串的第一个字符追加到字符串中。虽然这个问题听起来有些深奥，但它实际上相当普遍，并且可以利用像“AB ABABA”这样的输入字符串产生（图 9-7）。不过，在确定了这些问题之后，我们就准备好实际地实现 LZW 算法。

输入	输出	字典动作
A	65	无动作。代码在字典中
B	66	为“AB”添加代码 256
空格	32	为“B”添加代码 257
A		为“A”添加代码 258
B	256	为“ABA”添加代码 259
A		无动作
B		无动作
A	259	为“ABAx”添加代码 260
x	…	…

图 9-7 “AB ABABAx”的压缩

9.4.2 LZW 压缩的实现

LZW 压缩的实际实现必须处理几个棘手的问题。第一，应该使用多大的代码？这个答案确定了可以使用的字典的大小：如果使用 12 位的代码，那么将需要一个可以保存 212（4096）个条目的字典。第二，我们怎样实际地存储字典呢？在压缩期间，我们需要能够快速确定给定的代码是否存在于字典中；而在解压缩期间，我们需要能够快速扩展给定的代码。第三，当字典填满时，我们要做什么（显然，它将具有更大的输入文件）？

在程序清单 9-8（头文件 lzw. h）、程序清单 9-9（压缩例程 lzcomp. c）和程序清单 9-10（解压缩例程 lzwunc. c）中展示了一种处理这些问题的 LZW 的实现。虽然这些例程提供了两种可能的代码大小（12 位或 16 位），但是也许只能在编译时选择代码大小，并且它还受到底层编译器和操作系统的限制。甚至在使用较大的内存模型时，在 MS-DOS 下也没有足够的内存让 16 位的编译器使用 16 位的代码。实际上，12 位的代码可以用于 16 位的编译器，而 16 位的代码则只能被 32

位的编译器使用。不过，有趣的是，从12位转到16位得到的收获可能比期望的要少一些。这是由于在16位的代码字典中存储较大短语的潜力被在每个代码中发送4个以上的位的要求抵消了。在任何一种情况下，唯一真正的改变发生在输入/输出例程中。本书将重点关注12位代码的情况，感兴趣的读者可以检查程序清单，了解关于16位代码所需的实现改变的详细信息。

头文件lzw.h（程序清单9-8）定义了许多关键的常量。首先，我们需要保留多个代码以供内部使用：END_OF_INPUT被保留为终止代码；当我们处理字典填满的问题时，将需要NEW_DICTIONARY；STARTING_CODE和MAX_CODE将分别告诉我们第一个和最后一个空闲的字典代码。另请注意：虽然我们的字典最多将包含4 096个代码，但是我们将把实际的字典大小（DICTIONARY_SIZE）定义得稍大一点。如稍后将说明的，这有利于进行字典查找。

程序清单9-8 用于12位和16位的LZW实现的明示常量

```
/*--- lzw.h ------------------------------ Listing 9-8 --------
 * Header file for LZW compression routines                     *
 *-------------------------------------------------------------*/

/* select dictionary size */
#if !defined(BITS)
#define BITS 12
#endif

#if BITS == 12
/* Critical sizes for LZW */
#define PRESET_CODE_MAX 256     /* codes like this are preset */
#define END_OF_INPUT    256     /* this code terminates input */
#define NEW_DICTIONARY  257     /* reinitialize the dictionary */
#define STARTING_CODE   258     /* first code we can use */
#define MAX_CODE        4096    /* 2 ^ BITS */
#define UNUSED_CODE     4096    /* an invalid code that is never
                                   output. may be >= MAX_CODE if
                                   CodeType can hold it. The
                                   other option is to define
                                   after the fashion of the
                                   code for END_OF_INPUT */
#define DICTIONARY_SIZE 5021    /* a prime # > MAX_CODE * 1.2 */

/* all of these should be unsigned */
typedef unsigned short CodeType;  /* can hold MAX_CODE */
typedef unsigned short IndexType; /* can hold DICTIONARY_SIZE */
typedef unsigned long  CountType; /* used for statistics only */

#define SIGNATURE "LZW12"

#elif BITS == 16 /* requires a 32-bit context */
/* Critical sizes for LZW */
#define PRESET_CODE_MAX 256     /* codes like this are preset */
#define END_OF_INPUT    256     /* this code terminates input */
#define NEW_DICTIONARY  257     /* reinitialize the dictionary */
#define UNUSED_CODE     258     /* an invalid code */
#define STARTING_CODE   259     /* first code we can use */
#define MAX_CODE        65536   /* 2 ^ BITS */
#define DICTIONARY_SIZE 81901L /* a prime # > MAX_CODE * 1.2 */
```

```
typedef unsigned short CodeType; /* can hold MAX_CODE */
typedef unsigned long IndexType; /* can hold DICTIONARY_SIZE */
typedef unsigned long CountType; /* used for statistics only */

#define SIGNATURE "LZW16"

#else
#error Undefined BITS value!
#endif
```

在压缩和解压缩之间，用于存储压缩字典所需的数据稍有区别。在压缩例程中，我们使用如下形式的结构：

```
struct Dictionary {
    unsigned char c;
    CodeType code;
    CodeType parent;
} dictionary[DICTIONARY_SIZE];
```

其中 CodeType 是一个无符号的数量，可以保存 MAX_CODE。表中的每个条目都是一些字符口的最终节点，并且通过回溯利用 parent 中的索引定义的链表来构建实际的字符串。这个结构中的其他项还有：c 和 code，前者是在这个节点上终止的字符串的最后一个字符，后者是在这个节点上终止的条目的输出代码。注意：用于给定字符串的代码编号并不对应于它在表中的位置，而是被指定为表项本身的一部分。由于我们执行表查找的方式，所以选择这样做。虽然有可能构建一棵树来保存字典，但是在树的每个节点上需要多达 256 个分支（其中一个分支用于每个可能的后续字母）。这种结构查找缓慢并且浪费内存。我们代之以利用关于树的访问方式的知识，并把字典表视作散列表与链表之间的交集。

为了理解它如何工作，注意：用于输入字典中的项的代码被存储为条目的一部分，并且不对应于输入字典中的物理位置。最后这一点很重要。你可能发现把字典中的条目的位置视作其物理索引并把它对应的代码视作其逻辑索引是有用的。无论何时需要查找一个字符串，将总是具有用于短语的（输出或逻辑）代码，它与字符串的前 $n-1$ 个字符以及目标字符串的第 n 个字符匹配。我们在 LZWFind( ) 中结合这两个值，在输入字典中产生一个散列式物理索引。也就是说，假设我们已经扫描了字符串 xy，把它记作字典代码 400，并把它存储在物理索引 500 处。要查明字典是否包含字符串 xyz，我们现在把字典代码 400 与用于 z 的代码结合起来，产生新的散列式物理索引——也许是 1200 个结果的值。我们现在检查字典中的物理条目 1200。如果这个条目（a）具有代码字母 z 并且（b）具有父代码 400，那么就找到了我们所需要的。这个散列过程通常会直接把我们引导至想要的节点。如果没有找到匹配的节点（或者利用代码 UNUSED_CODE 标记的空节点），那么就简单地使用预先指定的步长值反复遍历表，直至找到匹配节点或空节点（参见第 3 章，复习散列的概念）。像这样使用散列查找表是输入字典包含比将使用的更多槽的原因：保留一些空槽，以允许表更高效地工作。为这个表选择的散列函数可能对其性能具有重大影响，并且代码中使用的函数是在经过多次试验之后选择的。可以通过定义 HASHHITS 和 HASHSTATS 变量并且检查输出的结果，轻松地监视散列函数的效率。

程序清单9-9 LZW压缩

```
/*--- lzwcomp.c -------------------------- Listing 9-9 --------------
 * Compress an input file, using the LZW encoding technique
 *
 * Entry point:     int LZWEncode (FILE *infile, FILE *outfile)
 *
 *                  Will compress infile into outfile. Both
 *                  files should already be open in binary mode.
 *                  They will be closed by LZWEncode().
 *
 *                  LZWEncode() returns:
 *                      0: success
 *                      1: error while reading in
 *                      2: malloc error
 *                      4: error while writing out
 *
 * Switches:        DRIVER    - compiles a test driver
 *                  HASHSTATS - reports on hashing efficiency
 *                  HASHHITS  - show table of hash hits
 *                  SHOWSTATS - provides compression statistics
 *-----------------------------------------------------------------*/

#define DRIVER
#define SHOWSTATS

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lzw.h"

CountType LZWTotBytesIn;   /* no. data bytes in input file */
CountType LZWTotBytesOut;  /* no. data bytes in output file */
CountType LZWCurrBytesIn;  /* no. data bytes in current frame */
CountType LZWCurrBytesOut; /* no. data bytes in current frame */
unsigned  LZWBestRatio;    /* best ratio we've had */
unsigned  LZWLastRatio;    /* last ratio we saw */

#if defined( HASHHITS ) && !defined( HASHSTATS )
#  define HASHSTATS
#endif

#if defined( HASHSTATS )
#  define SHOWHASH(x) x
CountType HashCount;
CountType HashCollide;
IndexType FreeMax;
#  if defined( HASHHITS )
unsigned long HashHits[ DICTIONARY_SIZE ];
#  endif
#else
#  define SHOWHASH(x)
#endif

/* structure for the data dictionary */
struct Dictionary {
    unsigned char c;
```

```
    CodeType code;
    CodeType parent;
};

/*--- the data storage and output routines ---*/

#if BITS == 12

struct Dictionary dictionary[ DICTIONARY_SIZE ];

/* These are the 12-bit routines */

static int havenibble;
static int olddata;
static void LZWOutInit ( FILE *outfile )
{
    char sig[] = SIGNATURE;
    fwrite ( sig, strlen ( sig ) + 1, 1, outfile );
    havenibble = 0;
    LZWLastRatio = LZWBestRatio = 100;
    SHOWHASH( HashCount = HashCollide = 0; )
#   if defined( HASHHITS )
    SHOWHASH( memset ( HashHits, 0,
              DICTIONARY_SIZE * sizeof ( unsigned long )); )
#   endif
    SHOWHASH( FreeMax = 0; )
}

static int LZWOutFlush ( FILE *outfile )
{
    if ( havenibble )
    {
        if ( fputc ( olddata, outfile ) == EOF )
            return ( 4 );
        LZWTotBytesOut += 1;
        LZWCurrBytesOut += 1;
    }
    return ( 0 );
}

static int LZWOut ( FILE *outfile, CodeType code )
{
    if ( havenibble )
    {
        olddata |= ( code >> 8 ) & 0xF;
        if ( fputc ( olddata, outfile ) == EOF )
            return ( 4 );
        if ( fputc ( code, outfile ) == EOF )
            return ( 4 );
        LZWTotBytesOut += 2;
        LZWCurrBytesOut += 2;

        havenibble = 0;
    }
    else
    {
```

```
        olddata = ( code >> 4 ) & 0xFF;
        if ( fputc ( olddata, outfile ) == EOF )
            return ( 4 );
        LZWTotBytesOut += 1;
        LZWCurrBytesOut += 1;
        olddata = ( code << 4 ) & 0xF0;
        havenibble = 1;
    }
    return ( 0 );
}
#elif BITS == 16
struct Dictionary *dictionary;

/* These are the 16-bit routines */
static void LZWOutInit( FILE *outfile )
{
    SHOWHASH( CountType i; )
    char sig[] = SIGNATURE;

    fwrite ( sig, strlen ( sig ) + 1, 1, outfile );
    SHOWHASH( HashCount = HashCollide = 0; )
#  if defined( HASHHITS )
    SHOWHASH( for ( i = 0; i < DICTIONARY_SIZE; i++ )
                HashHits[ i ] = 0; )
#  endif

    dictionary = malloc ( DICTIONARY_SIZE *
                  sizeof( struct Dictionary ));
}
static int LZWOutFlush ( FILE *outfile )
{
    return ( 0 );
}

static int LZWOut ( FILE *outfile, CodeType code )
{
    if ( fwrite ( &code, sizeof( CodeType ), 1, outfile ) != 1 )
        return ( 4 );
    LZWTotBytesOut += 2;
    LZWCurrBytesOut += 2;
    return ( 0 );
}
#else
#  error Undefined value for BITS!
#endif

/* our hashing lookup function */
static IndexType LZWFind ( CodeType currcode, int in )
{
    IndexType ndx;
    int step = 11, pastzero = 0;

#if BITS == 12
    ndx = ( currcode << 7 ) ^ in;
#elif BITS == 16
    ndx = ( ( IndexType ) currcode << 8 ) | in;
#endif
```

```
    ndx = ndx % DICTIONARY_SIZE;
    SHOWHASH( HashCount++; )
#if defined( HASHHITS )
    SHOWHASH( HashHits[ ndx ] ++; )
#endif
    for ( ;; )
    {
        if ( dictionary[ ndx ].code == UNUSED_CODE )
            break;
        if ( dictionary[ ndx ].parent == currcode &&
             dictionary[ ndx ].c == in )
            break;
        ndx += step;
        if ( ndx >= DICTIONARY_SIZE)
        {
            ndx -= DICTIONARY_SIZE;
            pastzero += 1;

            /*
             * Next is a safety check. If the step
             * value and the dictionary size are
             * relatively prime, there should never
             * be a problem. However, let's not loop
             * too many times.
             */

            if ( pastzero > 5 )
                step = 1;
        }
        SHOWHASH( HashCollide++; )

    }
    return ( ndx );
}

int LZWEncode ( FILE *infile, FILE *outfile )
{
    int retval = 0, in;
    CountType i;
    IndexType freecode = STARTING_CODE;
    CodeType currcode;
    IndexType idx;

    LZWTotBytesIn = 0;
    LZWTotBytesOut = 0;
    LZWCurrBytesIn = 0;
    LZWCurrBytesOut = 0;
    LZWOutInit( outfile );
#if BITS == 16
    if ( ! dictionary )
        return ( 2 );
#endif
    for ( i = 0; i < DICTIONARY_SIZE; i++ )
        dictionary[ i ].code = UNUSED_CODE;

    if (( currcode = fgetc ( infile )) == (CodeType) EOF )
        currcode = END_OF_INPUT;
```

```
    else
    {
        LZWTotBytesIn += 1;
        LZWCurrBytesIn += 1;
        currcode &= 0xFF; /* make sure we don't sign extend */
        while (( in = fgetc ( infile )) != EOF )
        {
            LZWTotBytesIn += 1;
            LZWCurrBytesIn += 1;
            idx = LZWFind ( currcode, in );
            if ( dictionary[ idx ].code == UNUSED_CODE )
            {
                /* not a match */
                retval = LZWOut ( outfile, currcode );

                /* now, update the dictionary */
                if ( freecode < MAX_CODE )
                {
                    dictionary[ idx ].c = in;
                    dictionary[ idx ].code = freecode++;
                    dictionary[ idx ].parent = currcode;
                }
                currcode = in;

                /* Had a miss; check compression efficiency */
                if ( LZWCurrBytesIn >= 10000 )
                {
                    unsigned ratio;
                    ratio = ( LZWCurrBytesOut * 100 ) /
                            LZWCurrBytesIn;
#if defined( HASHSTATS )
                    printf( "Input: %lu, "
                            "Output: %lu, "
                            "Overall Ratio; %lu%%, ",
                            LZWTotBytesIn,
                            LZWTotBytesOut,
                            ( LZWTotBytesOut * 100 ) /
                            LZWTotBytesIn );
                    printf( "This frame: %u%%\n", ratio );
#endif
                    LZWCurrBytesIn = 0;
                    LZWCurrBytesOut = 0;
                    if ( ratio > LZWBestRatio )
                    {
                        if ( ratio > 50 &&
                             ( ratio > 90 ||
                             ratio > LZWLastRatio + 10 ))
                        {
#if defined( HASHSTATS )
                            printf ( "Dictionary reset\n" );
#endif
                            LZWOut ( outfile, NEW_DICTIONARY );
                            for ( i = 0;
                                  i < DICTIONARY_SIZE; i++ )
                                dictionary[ i ].code =
                                    UNUSED_CODE;
                            SHOWHASH( if (freecode > FreeMax)
                                FreeMax = freecode; )
                            freecode = STARTING_CODE;
```

```
                    }
                }
                else
                    LZWBestRatio = ratio;
                LZWLastRatio = ratio;
            }
        }
        else /* we match so far--keep going */
            currcode = dictionary[ idx ].code;

        if ( retval ) /* make sure no problems so far */
            break;
    }
}

retval = LZWOut ( outfile, currcode );
if ( ! retval )
    retval = LZWOut ( outfile, END_OF_INPUT );
if ( ! retval )
    retval = LZWOutFlush ( outfile );

#if defined( SHOWSTATS )
    if ( ! retval )
    {
        printf ( "The input file contained %lu bytes\n",
                 LZWTotBytesIn );
        printf ( "The output file contained %lu data bytes\n",
                 LZWTotBytesOut );
        if ( LZWTotBytesIn != 0 )
            printf ( "Output file is %lu%% "
                     "the size of input file\n",
                     ( LZWTotBytesOut * 100 ) / LZWTotBytesIn );
    }
#endif
#if defined( HASHSTATS )
    {
        float ratio;
        if ( freecode > FreeMax )
             FreeMax = freecode;
        printf( "The highest code used was %u\n", FreeMax );
        printf( "There were %ld hashes and %ld collisions,\n",
            HashCount, HashCollide );
        ratio = ( float ) HashCollide / ( float ) HashCount;
        printf( "for a ratio of %f\n", ratio );
#   if defined( HASHHITS )
        for ( i = 0; i < min( DICTIONARY_SIZE, 4096 ); i += 8 )
        {
            int j;
            printf( "%6ld: ", i );
            for ( j = 0; j < 8; j++ )
            {
                if ( i + j >= DICTIONARY_SIZE )
                    break;
                printf( "%10ld  ", HashHits[ i + j ] );
            }
            printf( "\n" );
        }
```

```
#  endif
    }
#endif

    if ( ferror ( infile ))
        retval = 1;
    if ( ferror ( outfile ))
        retval = 4;
    fclose ( infile );
    fclose ( outfile );
    return ( retval );
}

/*-------------------------------------------------------------
 * The following driver for the previous routines is active
 * if DRIVER is defined.
 *-------------------------------------------------------------*/
#if defined( DRIVER )
int main( int argc, char *argv[] )
{
    FILE * infile, *outfile;
    int retval;

    if ( argc != 3 )
    {
        printf( "Usage: lzwenc infile outfile\n" );
        return ( EXIT_FAILURE );
    }

    infile = fopen( argv[ 1 ], "r+b" );
    if ( infile == NULL )
    {
        fprintf( stderr, "can't open %s for input\n", argv[1] );
        return ( EXIT_FAILURE );
    }

    outfile = fopen( argv[ 2 ], "w+b" );
    if ( outfile == NULL )
    {
    fprintf( stderr, "can't open %s for output\n", argv[2] );
    return ( EXIT_FAILURE );
}
setvbuf ( outfile, NULL, _IOFBF, 8192 );

retval = LZWEncode ( infile, outfile );

if ( retval )
{
    printf( "compression failed: " );
    if ( retval == 1 )
        printf ( "input error\n" );
    else
    if ( retval == 2 )
        printf ( "malloc error\n" );
    else
    if ( retval == 4 )
```

```
            printf ( "output error\n" );
        else
            printf ( "\n" );
        return  ( retval );
    }
    else
    {
        printf( "%s was compressed into %s\n",
                                    argv[ 1 ], argv[ 2 ] );
        return ( EXIT_SUCCESS );
    }
}
#endif
```

在解压缩期间，可以不需要 code 条目，并且只是简单地把节点存储在表中的合适位置中。这可以工作，因为在解压缩期间我们从不需要查找表中的短语：我们总是具有正确的代码，并且可以直接把它用作表的索引。作为替代，我们现在的问题是：在最终节点上开始，并解开相应的短语。很容易完成这项任务，其方法是：向上追溯 parent 值，直至到达代码值 <256 的节点。此时，字符串必须终止。这个过程是在 lzwunc. c（程序清单 9-10）中由 LZWLoadBuffer( ) 执行的。这个例程使用一个简单的字符串缓冲区以逆序构建字符串。注意：解压缩例程知道值 <256 的节点用于表示它们自身，并且从不需要实际地进行初始化。

程序清单 9-10 LZW 解压缩

```
/*--- lzwunc.c --------------------------- Listing 9-10 -------
 * Decompress an LZW-compressed file
 *
 * Entry point:     int LZWDecode(FILE *infile, FILE *outfile)
 *
 *                  Will decompress infile into outfile. Both
 *                  files should already be open in binary mode.
 *                  They will be closed by LZWDecode().
 *
 *                  LZWDecode() returns:
 *                      0: success
 *                      1: invalid signature byte
 *                      2: bad malloc
 *                      3: read error
 *                      4: write error
 *
 *
 * Switches:        DRIVER - compiles a test driver
 *                  CODES  - displays input codes
 *-------------------------------------------------------------*/
#define DRIVER

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "lzw.h"

#if defined( CODES )
unsigned long outcount;
#endif
```

```
/* structure for data dictionary */
struct Dictionary {
    unsigned char c;
    CodeType parent;
};

/* decode buffer */
static unsigned char *DecodeBuffer;
static unsigned DecodeBufferSize = 0;
static unsigned LZWLoadBuffer( unsigned, CodeType );

/* the input routines and data storage routines */
#if BITS == 12
/* These are the 12-bit input routines */
/* data dictionary */
struct Dictionary dictionary[ MAX_CODE ];

static int havenibble;
static int olddata;
static CodeType incode;
static int LZWInInit ( void )
{
    havenibble = 0;
    if ( ! DecodeBufferSize )
    {
        DecodeBufferSize = 1000;
        DecodeBuffer = malloc( DecodeBufferSize );
        if ( DecodeBuffer == NULL )
            return ( 2 );
    }
#   if defined( CODES )
    outcount = 0;
#   endif
    return 0;
}

static int LZWIn ( FILE *infile )
{
    int data;
    if ( havenibble )
    {
        incode = olddata << 8;
        if (( data = fgetc ( infile )) == EOF )
            return ( 4 );
        incode |= ( data & 0xFF );
        havenibble = 0;
#   if defined( CODES )
        {
            unsigned count;
            printf ( "%6ld (1): %4x, \"", outcount, incode );
            count = LZWLoadBuffer ( 0, incode );
            while ( count )
                if ( DecodeBuffer[ --count ] < 32 )
                    printf ( "'0x%X'", DecodeBuffer[ count ] );
                else
                    fputc ( DecodeBuffer[ count ], stdout );
            printf ( "\"\n" );
```

```
        }
#   endif
        return ( 0 );
    }
    else
    {
        if (( data = fgetc ( infile )) == EOF )
            return ( 4 );
        incode = ( data & 0xFF ) << 4;
        if (( data = fgetc ( infile )) == EOF )
            return ( 4 );
        incode |= ( data >> 4 ) & 0xF;
#   if defined( CODES )
        {
            unsigned count;
            printf( "%6ld (0): %4x, \"", outcount, incode );
            count = LZWLoadBuffer ( 0, incode );
            while ( count )
                if ( DecodeBuffer[ --count ] < 32 )
                    printf ( "'0x%X'", DecodeBuffer[ count ] );
                else
                    fputc ( DecodeBuffer[ count ], stdout );
            printf ( "\"\n" );
        }
#   endif
        olddata = data & 0xF;
        havenibble = 1;
        return ( 0 );
    }
}

#elif BITS == 16
/* These are the 16-bit routines */
/* data dictionary */
struct Dictionary *dictionary;

static CodeType incode;
static int LZWInInit( void )
{
#   if defined( CODES )
    outcount = 0;
#   endif
    if (!DecodeBufferSize)
    {
        DecodeBufferSize = 1000;
        DecodeBuffer = malloc ( DecodeBufferSize );
        if ( DecodeBuffer == NULL )
            return ( 2 );
    }
    dictionary = malloc( DICTIONARY_SIZE *
                    sizeof( struct Dictionary ) );
    if ( dictionary == NULL )
        return ( 2 );
    return ( 0 );
}

static int LZWIn ( FILE *infile )
```

```
{
    if ( fread ( &incode, sizeof( CodeType ), 1, infile ) != 1 )
        return ( 4 );
#  if defined( CODES )
    {
        unsigned count;
        printf ( "%6ld (0): %4x, \"", outcount, incode );
        count = LZWLoadBuffer ( 0, incode );
        while ( count )
            if ( DecodeBuffer[ --count ] < 32 )
                printf ( "'0x%X'", DecodeBuffer[ count ] );
            else
                fputc( DecodeBuffer[ count ], stdout );
        printf( "\"\n" );
    }
#  endif
    return ( 0 );
}

#else
#  error Undefined value for BITS!
#endif

/* the actual decompression routine */
static IndexType freecode;
static unsigned LZWLoadBuffer ( unsigned count, CodeType code )
{
    if ( code >= freecode )
    {
        printf( "LZWLoad: code %u out of range!", code );
        return ( 0 );
    }
    while ( code >= PRESET_CODE_MAX )
    {
        DecodeBuffer[ count++ ] = dictionary[ code ].c;
        if ( count == DecodeBufferSize )
        {
            DecodeBuffer =
               realloc ( DecodeBuffer, DecodeBufferSize + 1000 );
            if ( ! DecodeBuffer )
            {
                /* out of memory */
                DecodeBufferSize = 0;
                return ( 0 );
            }
            else
                DecodeBufferSize += 1000;
        }
        code = dictionary[ code ].parent;
    }
    DecodeBuffer[ count++ ] = code;
    return ( count );
}

int LZWDecode ( FILE *infile, FILE *outfile )
{
    char buffer[ 10 ];
```

```
    int retval = 0;
    unsigned int inchar;
    unsigned count;
    CodeType oldcode;

    /* check the signature */
    fgets ( buffer, strlen ( SIGNATURE ) + 2, infile );
    if ( strcmp ( buffer, SIGNATURE ))
    {
        retval = 1;
        goto done;
    }

    /* prime the pump */
    if ( retval = LZWInInit() )
        goto done;
priming:
    freecode = STARTING_CODE;
    if ( retval = LZWIn ( infile ))
        goto done;
    if ( incode == END_OF_INPUT )
        goto done;

    /* the first character always is itself */
    oldcode = incode;
    inchar = incode;
    if ( fputc( incode, outfile ) == EOF )
    {
        retval = 4;
        goto done;
    }
#if defined( CODES )
    outcount += 1;
#endif
    while ( ! ( retval = LZWIn ( infile )))
    {
        if ( incode == END_OF_INPUT )
            break;
        if ( incode == NEW_DICTIONARY )
            goto priming;
        if ( incode >= freecode )
        {
            /* We have a code that's not in our dictionary! */
            /* This can happen only one way--see text */

            count = LZWLoadBuffer ( 1, oldcode );

            /* Make last char same as first. Can use either */
            /* inchar or the DecodeBuffer[count-1] */

            DecodeBuffer[ 0 ] = inchar;
        }
        else
            count = LZWLoadBuffer ( 0, incode );

        if ( count == 0 )
            return ( 2 ); /* had a memory problem */
```

```
        inchar = DecodeBuffer[ count - 1 ];
        while ( count )
        {
            if ( fputc ( DecodeBuffer[--count], outfile) == EOF )
            {
                retval = 4;
                goto done;
            }
#if defined( CODES )
            outcount += 1;
#endif
        }

        /* now, update the dictionary */
        if ( freecode < MAX_CODE )
        {
            dictionary[ freecode ].parent = oldcode;
            dictionary[ freecode ].c = inchar;
            freecode += 1;

#if defined( CODES )
            {
                unsigned cnt;
                printf( "    just added code %5u: \"",
                                freecode - 1 );
                cnt = LZWLoadBuffer ( 0, freecode - 1 );
                while ( cnt )
                    if ( DecodeBuffer[ --cnt ] < 32 )
                        printf ( "'0x%X'", DecodeBuffer[ cnt ] );
                    else
                        fputc ( DecodeBuffer[ cnt ], stdout );
                printf( "\"\n" );
            }
#endif
        }
        oldcode = incode;
    }

 done:
    fclose ( infile );
    fclose ( outfile );
    return ( retval );
}

#if defined( DRIVER )
int main ( int argc, char *argv[] )
{
    FILE * infile, *outfile;
    int retval;

    if ( argc != 3 )
    {
        fprintf ( stderr, "Usage: lzwdec infile outfile\n" );
        return ( EXIT_FAILURE );
    }

    infile = fopen ( argv[ 1 ], "r+b" );
```

```
    if ( infile == NULL )
    {
        fprintf ( stderr, "can't open %s for input\n", argv[1] );
        return ( EXIT_FAILURE );
    }

    outfile = fopen ( argv[ 2 ], "w+b" );
    if ( outfile == NULL )
    {
        fprintf( stderr, "can't open %s for output\n", argv[2] );
        return ( EXIT_FAILURE );
    }

    if ( retval = LZWDecode( infile, outfile ))
    {
       printf( "uncompression failed: " );
       if ( retval == 1 )
           printf ( "invalid signature bytes\n" );
       else
       if ( retval == 2 )
           printf ( "malloc failed\n" );
       else
       if ( retval == 3 )
           printf ( "read error in data\n" );
       else
       if ( retval == 4 )
           printf ( "write error on output\n" );
       else
           printf ( "\n" );
       return ( retval );
    }
    else
    {
        printf ( "%s was expanded into %s\n",
                                        argv[ 1 ], argv[ 2 ] );
        return ( 0 );
    }
}
#endif
```

9.4.3 填满字典

任何相对大的输入文件都会很快填满字典，并且需要考虑这种情况。一种方法是什么事也不做：只需停止记录新的代码即可。压缩将继续进行到可能的程度。这种方法可能导致性能降级。算法有效地执行压缩的能力强烈依赖于它继续看见在其字典中具有代码的数据。如果输入流中的数据类型发生重大改变，那么压缩可能急剧降级。示范例程通过监视压缩的效率来处理这种问题。大约对于每 10 000 个输入字节，LZWEncode( ) 将会检查它的压缩统计信息。如果压缩的能力看起来似乎在降级，那么就会清理字典，并重新开始添加短语。通过在压缩文件中检测保留代码 NEW_DICTIONARY 向解压缩例程通知这一事实。

比较这些例程的 12 位版本和 16 位版本是有趣的。虽然 16 位版本有时对非常大的文件（> 1 MB）具有优势，但是 12 位版本通常似乎对更典型的较小文件更为出色。为了在启用 16 位处

理的情况下编译程序，可以在 lzw.h 中把 BITS 定义为 16。在下面一节中详细比较了两个版本的性能。

9.5 使用哪种压缩方法

如这一整章中所指出的，多种不同的技术适用于某些类型的数据。下面的表 9-1 显示了本章中展示的算法的压缩效率。在所有情况下，用于压缩的文件都很大。对较小文件进行的压缩将显示不同的结果。

表 9-1 本章中讨论的算法的压缩性能将输出文件的大小显示为输入文件的百分比

文档	RLE	霍夫曼压缩	滑动窗口压缩①	LZW 12 位	LZW 16 位
文本：460 KB	76.6%	48.2%	42.4%	36.9%	32.8%
二进制：1.9 MB	98.3%	79.7%	54.8%	74.4%	69.8%
数据库：5.3 MB	46.2%	36.2%	32.0%	30.1%	23.6%

① 使用 Microsoft 的压缩算法。

这个表显示了一些有趣的模式。对于较大的文件，16 位的 LZW 始终优于 12 位的 LZW。LZW 一般优于滑动窗口（纯二进制文件例外）；滑动窗口技术优于霍夫曼压缩，并且它们都优于 RLE。

在选择算法时，除了压缩效率之外，还有其他一些因素需要考虑。例如，在所有这些实现中，滑动窗口压缩是最慢的——通常比另外几种算法要慢得多，而 RLE 则是最快的。RLE 只能对数据库进行充分压缩（如以前所解释的）。不过，它确实具有独特的优点：它是唯一一种可以从损坏中恢复的方法。其他方法把字节编码为较短的位模式。如果任何位损坏，所有的下游字节都必定会损坏。而在 RLE 中，损坏只限制于已损坏的字节。如果单个位损坏，则只会影响那个数据字节或者编码符号。经过损坏点的解压缩将会正确地工作。

霍夫曼算法是不能动态执行压缩的唯一方法。为了让霍夫曼算法执行其工作，它必须读取整个文件两次，因此在压缩可以开始之前，必须知道完整的数据集。

在大多数情况下，LZW 算法似乎应该更受欢迎，不幸的是，由于优利系统公司对其享有专利，从而束缚了它的应用。不能免费使用它。无疑，可以创建你自己的版本（添加 Welch 的改进包括预先加载字典）。不过，甚至在使用这些粗制滥造的版本之前，可能也需要得到许可。

这意味着：对于大多数用户而言，唯一可以免费使用的有效的压缩例程是霍夫曼编码和滑动窗口压缩算法。幸运的是，它们对各类文件都可以做出令人满意的良好工作。

9.6 资源和参考资料

Bentley, Jon. "Squzng Engl Txt." *UNIX Review*, Vol. 13, No. 2, pp. 61. February 1995.

Davis, Pete. "Microsoft's Compression File Format." *Windows/Dos Developer's Journal*, July 1994, pp. 59-64.

Huffman, D. A. "A Method for the Construction of Minimum-Redundancy Codes." *Proceedings of the IRE*, Vol. 40, No. 9, pp 1098-1101, September 1952.

Nelson, Mark. *The Data Compression Book*. Redwood City, CA: M&T Books, 1991. 如果你想更

深入地研究数据压缩，这本书是关于这个主题的最佳书籍，它具有广泛的回旋余地。在该书中，利用经过深思熟虑的插图和相当多的C源代码透彻解释了所有重要的算法。

Swan，Tom. *Inside Windows File Formats*. Indianapolis，IN：Howard Sams & Co.，1994.

Ziv、Jacob 和 Abraham Lempel. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory*，Vol. 23，No. 3，pp. 337-343，May 1977. 尽管 Ziv 是第一作者，出于某些不为人所知的原因，这个算法一般还是称为 Lempel-Ziv 或 LZ。

第 10 章　数据完整性和验证

向初级程序员介绍的第一个首字母缩写词是 GIGO（garbage in，garbage out，“垃圾进，垃圾出”，类似于“种瓜得瓜，种豆得豆”的意思）。它简单地提醒：如果数据不准确，那么基于该数据的任何结果也将不准确。数据验证是在计算的几乎所有方面都可以发现的一种活动，但是它更频繁地发生在数据处理和远程通信中。这两个领域都具有对数据验证的相同需求，尽管它们以非常不同的方式来诉诸这种需求。

在数据处理中，是在把数据输入到应用程序中时对其进行验证的。通常，验证过程发生在把数据键入到应用程序中时。例如，知道用户已经键入了引擎零件的序列号、催询单或者个人记录是很重要的。在输入时进行的验证可以采用非常简单的形式。一个例子是：确保日期在年份中具有 4 位数字，并在月和日字段中各具有两位数字。进一步的完整性检查可能确保数字对应于有效的日期。其他字段的验证可以使用多种广泛的方式。例如，校验和（将在下一节中讨论）通常用于确保在序列号内没有颠倒数字。

远程通信的字段使用数据验证来确保接收数据的站点所接收到的数据与发起站点发送的数据相同。远程通信协议在传送期间使用各种技术来确保数据完整性。这些技术包括为每个字符使用奇偶校验位，以及为传送数据的每个分组使用校验和。远程通信中使用的技术将在本章后面讨论。

除了远程通信和数据处理之外，在用户需要知道文件是否已被修改的任何地方都会使用数据验证技术（比如校验和）。例如，防病毒方案通常的工作方式是：比较来自当前文件的校验和与用于那些特定文件的记录上的校验和。通过这样做，可以检测到病毒执行的任何篡改。设置高级安全性的 Internet 站点使用类似的校验和方案，准确查看在系统上上传或者修改了什么内容。数据验证通常用在备份软件中，用于确保备份文件与原始文件完全相同。无论何时需要确保数据或二进制文件的完整性，都可以在本章中展示的算法当中找到强大的解决方案。

10.1　简单的校验和

验证数据的最简单的方式是使用**校验和**（checksum）。校验和是一个计算的值，当添加从数据自身中提取的一些值时，它会生成可以确保其完整性的第三个值。

简单的校验和（称为**奇偶校验**）用在计算机的硬件内，用于验证内存中保存的数据的完整性。奇偶校验也广泛用在远程通信中，该技术最初在其中获得了广泛应用。

当把一个字节存储在内存中时，它通常会占据 8 位。不过，对所有内存操作透明的是，对于每个 8 位的字节，都将在内存中设置第 9 位（称为**奇偶校验位**），除了最低级的硬件之外，其他所有硬件都不能访问它。这个位包含值 1 或 0，这依赖于与之关联的字节的值。在使用**奇校验**（odd parity）的方案中，将把奇偶校验位的值设置成使得值为 1 的位（包括原始字节中的位和奇偶校验位）的总数是奇数。例如，如果一个字节的值为 0（所有位都是 0），那么就奇校验来说，奇偶校

验位的值将是1。这样，在原始字节与奇偶校验位之间，将具有奇数个值为1的位；在这种情况下，总数就是1。同样，如果一个字节的值是14（二进制值00001110），由于设置为1的位的总数已经是奇数，所以奇偶校验位的值将是0。在使用**偶校验**（even parity）的方案中，设置为1的位的总数将是偶数，而不是奇数。使用奇偶校验位，计算机内存管理硬件可以粗略地检查内存中的数据是正确写入的，并且没有损坏。

奇偶校验检查的好处是：可以非常快地执行。其缺点是：可能有许多错误无法检测到。单个位错误总会被检测到，但是如果两个位改变值，奇偶校验将不会注意到这一点。考虑下面的示例：

1111 0111　　奇偶校验位（偶数）：1

1111 1011　　奇偶校验位（偶数）：1

在这里，调换了字节中两个位的位置，但它生成了相同的奇偶校验位。从这个示例中可以看出：奇偶校验位可以检测到奇数个位的换位，但是不能检测到偶数个位的换位。

远程通信协议也使用奇偶校验。确切地讲，异步远程通信（最频繁使用的远程通信类型，IBM的大型机除外）在每个字符的末尾使用奇偶校验位，以确保正确地传送了它。任何使用过调制解调器的人都有为拨入公告板（BBS）或其他站点而设置所需协议的经历。必须正确完成的一种设置是目标站点使用的奇偶校验类型。如果奇偶校验不匹配，通信协议将错误地认为发生了传输错误。奇偶校验的一般选择是：奇校验、偶校验或者不使用奇偶校验（没有奇偶校验位）。

例如，CompuServe默认使用偶校验。不过，今天的大多数拨号服务都不使用奇偶校验，这主要是由于奇偶校验检查的可靠性问题，以及由于发送奇偶校验位将用光远程通信带宽。在典型的异步远程通信中，每个字节都会消耗以下位数：

1个起始位（宣告其后的数据字节）

8个数据位（字节本身）

0个或1个奇偶校验位

1个停止位（宣告字节传输结束）

因此，使用奇偶校验位意味着每个字节都会从10位增加到11位，从而导致吞吐量下降了10%。这足以（特别是从奇偶校验检查的局限性的角度看）促使人们极少在异步通信中使用奇偶校验位。其他验证数据的方式经常被使用。本章在后面讨论它们。

仅当单个位出错时，奇偶校验检查才是有效的。一旦两个或更多的位损坏，正确地执行奇偶校验检查的概率平均来讲只有50%。对于重要数据的传输，这种有效性太低。奇偶校验检查仍然保留在计算机硬件中，因为奇偶校验位的处理是在固件（firmware）中完成的，因此执行速度非常快。此外，由于奇偶校验位只能被内存管理硬件访问，所以奇偶校验位不会阻碍数据吞吐量。不过，任何为计算机购买过内存的人都确实会注意到RAM芯片9个一组（其中8个芯片用于保存数据位，1个芯片用于保存奇偶校验位），这意味着需要付出一定的成本来维持奇偶校验检查内存的传统。

其他形式的校验和在许多应用程序中使用得更有效一些。例如，用于Intel处理器的目标文件中的记录都具有校验和。目标文件是由编译器或汇编器生成的。它们包含一些记录，其中保存有与合适的库链接之前的程序的可执行代码。目标文件的格式因处理器而异。Intel处理的目标文件格式指定每个记录的最后一个字节包含1字节的校验和，它是按Intel文档［Intel 1981］中指定的方式计算的："每个记录中的最后一个字段是校验和，它包含记录中所有其他字节之和的2的补码

(以256为模)。因此，记录中所有字节之和（以256为模）等于0。”

Intel的例子是一个经典的校验和——它是从要检查的数据生成的值，以便提供可以确定其完整性的第三个值（在这里是0）。可以轻松地把用于该校验和的代码组合在一起。把数据中的所有字节相加进单个char字段（丢弃多余的位），并取那个和的负值，如程序清单10-1所示。

程序清单10-1 计算Intel目标文件的校验和

```
char IntelCheckSum ( char *data, int length )
{
   char sum;
   int i;

   for ( i = 0, sum = 0; i < length; i++ )
         sum += data[i];

   return (-sum );
}
```

注意：通过把sum设置为char类型（而不是int类型），把sum的值限制为最大8位（或者最大值255）。任何大于255的和都将占据8个以上的位，并且会截去溢出的位。对小于256的值进行这种截去就是Intel规范指定对256求模的含义。代码中的最后一步把最终的和乘以-1，从而生成校验和，它与sum相加将得到0。注意：我们必须知道数据的长度，以生成校验和。与C语言中的字符串不同，校验和过程不能结束于第一个null字节，因为这个字节可以包含有效的数据。

这种校验和比以前讨论的奇偶校验检查有效得多。它远远不止能够发现单个位损坏。同样，在可以欺骗这种校验和报告无效的数据是正确的之前，必须有两个位出错。如果单个位自身出错，就应该在最终的校验和中显示出来。不过，如果两个补充位出错导致第二个位的错误抵消了第一个位的错误，那么校验和方法将不能检测到这种错误。注意：发生这种位组合的几率只有1/256。因此，这种校验和远比奇偶校验检查要准确得多。

另请注意这种校验和具有相当大的空间效率：它可以通过只使用单个字节，为较大的数据块提供某种数据验证。不过，当这种校验和验证的数据量增加到超过256字节时，校验和的错误率将变得更加麻烦。让我们通过使用一个8字节的字符串来研究这一点，该字符串由以下二进制值组成（以十进制形式显示）：

128，128，128，128，128，128，128，14

我们获得了校验和142——用于最后两个字节的校验和。这是由于前6个字节成对出现，并且每一对的和是256，从而导致前6个字节的校验和为0。最后两个字节（128，14）中的任何改变都将准确地反映在最终的校验和中，根据以前提到的附加条件，最后一个字节中的改变不会在前一个字节中相应地抵消（1/256的几率）。在前一个示例中，任何生成校验和0的6字节的字符串都可以替换6个存储128的字节，而不会被校验和检测到。继续使用这种方法，我们很快就会发现：任何2个字节的组合出错并且不会被校验和检测到的几率为1/256。这导致的结果是：要执行校验和检查的数据越长，出错的几率越大。可以使用一个字节的校验和并且不会冒过大风险的一般可接受的最大长度是256字节。

为了提高校验和方法的可靠性，可以使用两个字节的校验和，它执行相同的计算。为此，将

更改程序清单10-1中的代码，使得函数返回一个int，并且把sum定义为int。

一个有趣的历史记录是在Intel记录中使用一个字节的校验和。在第一次发布Intel目标标准之后不长时间，编译器供应商承认他们能够生成正确的Intel目标记录，而无需利用校验和验证它们。并且由于磁盘驱动器平均每写入10^{12}个位才会出现1个错误，虽然仍然会生成校验和字段，但是从来也不会使用它。在某个时间，编译器编写者选择完全忽略这个字段，并且简单地在校验和字节中插入一个0。当Intel于1993年重新发布标准时［Intel 1993］，对校验和字段的描述相应地进行了更新："一些编译器写入一个0字节，而不是计算校验和，这两种形式都应该可以被处理目标模块的程序所接受。"

校验和的选择是由要验证的信息的性质决定的。例如，20世纪80年代早期，美国邮政服务（United States Postal Service）设计了一种条形码方案，它允许扫描仪从信封中读取邮政编码，然后对邮政编码进行编码并沿着信封的底部边缘打印它。然后，后续的邮件扫描仪可以识别这些条形码，并利用它们自动递送信件。这个条形码系统［USPS 1984］使用奇偶校验方案来编码邮政编码中的每个数字，并且使用校验和来验证整个数字序列。条形码由短线条和长线条组成，其中长线条表示1，短线条表示0。图10-1中显示了邮政条形码的示例。

图10-1 美国邮政服务的条形码显示了9位数字的邮政编码（12345-6789），其后接着一位校验和数字（5）

邮政条形码方案的工作方式如下：校验和是必须与前9位数字的总和相加以得到10的倍数的数字。在图10-1中，前9位数字的总和是45，因此校验和是5。这些数字中每个数字本身都包括5根线条，并且每根线条表示值为0或1的位。前4根线条包含指示数字的位，而最后一根线条则用于获得偶校验。线条的值（从左到右）是7-4-2-1。之所以为最左边的线条选择7而不是8，其原因将在下一个段落中解释。我们看到1被表示为00011，或者更具体地讲，1被表示为0001和奇偶校验位1。同样，6被表示为01100，或者表示为0110，其后接着奇偶校验位0；8被表示为1001，其后接着奇偶校验位0。

通过使用7（而不是8）作为最左边的线条，美国邮政服务对它的方案添加了一种改进：每个数字中至少都有两个1（这不同于偶校验。回想一下，对于偶校验，一个数字中不能有设置为1的位）。这种改进意味着必须赋予数字0一个特殊值：1100，并且带有奇偶校验位0。

最后，条形码具有初始和尾随的1，它们指示条形码的开始和结束。使用这种方案，美国邮政服务可以验证每个数字都正确地进行了编码（通过使用奇偶校验位），并且整个代码是正确的（通过使用校验和）。最后的检查包括统计1的个数：由于每个数字都必须具有至少两个1，并且在代码的开头和末尾各有一个1，因此每个条形码都应该具有至少22个1。

迄今为止介绍的校验和可以有效地用于验证现有的数据被正确地进行了编码。例如，给定一个正确的邮政编码，邮政条形码系统可以快速验证已经正确地读取并打印了条形码。确切地讲，这些校验和只能告诉你数据是以有效的形式表示的。它们不能告诉你原始数据是否正确以便从它

开始执行处理。在前面的示例中，校验和不能指出原始的邮政编码是否正确，而只能指出在条形码中编码了有效的数字序列。这些校验和处理的是形式，而不是内容。这种局限性意味着这些校验和不能用于验证由键盘穿孔机操作员输入的数据。对于手动输入的数据，需要同时检查数据具有有效的形式，并且它与正在键入的原始数据匹配。为此，需要使用加权校验和。

10.2 加权校验和

加权校验和（weighted checksum）尝试克服简单校验和的重大缺点：检测换位。假设一位键盘穿孔机操作员正在输入9位的邮政编码，并且我们依据美国邮政服务的方法计算校验和。我们计算第10位数字，当把它与9位邮政编码数字相加时，将产生一个为10的倍数的总和。这个校验和数字将告诉我们邮政编码是否具有有效的形式，但是它不会告诉我们数字是否进行了换位。在图10-1中，邮政编码12345-6789生成的校验和是5。如果数据输入操作员调换了前两位数字的位置（21345-6789），我们将获得相同的校验和，并且不会注意到错误。

为了解决换位的问题（数据输入中的主要问题），我们必须给正在验证的每一位数字的位置赋予不同的值。通过给每个位置赋予不同的加权来执行该任务。然后用数字本身乘以加权，并使用这些加权数字的和作为校验和的基础。简单的方案是给每个位置赋予7的递增倍数，如表10-1中所示。

表10-1 基于7的幂的位置加权

位　置	加　权	位　置	加　权
1	1	6	16807
2	7	7	117649
3	49	8	823543
4	343	9	5764801
5	2401	10	40353607

依据这个表，将把像12345这样的5位数字编码为1 * 2401，2 * 343，3 * 49，4 * 7，5 * 1。它们的和将是3267。然后，我们将取这个数字对11求模（把值0赋予余数10）当把模11应用于3267时，结果为0。这是校验和数字。数据处理中通常使用的另一种方案是使用9的幂对11求模。稍后将比较这些方法的准确度。

使用模11而不是和的最后一位数字（它对应于模10）基于通过把一个数字除以一个质数（11）而产生的排列。在第3章“散列”中首先介绍了这个主题。把任何数字除以11，它将生成一个新的余数（除非数字正好是11的倍数）。对于数字10则不是这样，因为用于10的模函数将永远不会改变最终的余数数字。选择11而不是更大的质数（比如13）允许最少的值映射。在模11中，将余数10映射到0；就13而言，将不得不把三个余数（10、11、12）映射到单个数字，而不会在余数的分布中取得重大的收获。程序清单10-3（cksmtest.c，将在本章后面介绍）显示了选择模10而不是模11将对校验和数字产生多么重大的负面影响。

程序清单10-2中显示了用于生成上述加权校验和的代码，以及允许在命令行指定要检查的数字的小驱动程序。这段代码把表10-1中的值放在一个查找表中，并以一种直观的方式计算校验和。

程序清单10-2 如何生成7对11求模的校验和

```
/*--- ck7mod11.c ------------------------ Listing 10-2 ------
 * Demonstrates weighted checksum using powers of 7 and
```

```
 * modulo 11
 *-----------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define DRIVER   1 /* generate a main line when #defined */
#define VERBOSE  1 /* print information about computation */

int Get7Mod11Chksum ( const char *data, int length )
{
    int  i, j;
    int  position;
    unsigned long int sum;

    static long int powers [10] = { 1, 7, 49, 343, 2401,
            16807, 117649, 823543, 5764801, 40353607 };

    /* make sure all digits can be looked up */

    if ( length > 10 )
        return ( -1 );

    sum = 0L;
    position = 0; /* what digit in the data are we at? */

    for ( i = length; i > 0; i--, position++ )
    {
        /* do we have a digit ? */

        if ( ! isdigit ( data[i-1] ))
            return ( -2 );

        /* convert digit character to a number */

        j = data[i-1] - '0';

        /* look up power, multiply by j, add to sum */

        sum += powers [position] * j;
    }
    /* get the sum modulus 11 */

    i = (int) ( sum % 11L );


    if ( i == 10 )
        i = 0;

#ifdef VERBOSE
    printf ( "Sum is %ld, checksum is %d\n", sum, i );
#endif

    return ( i );
}
```

```
#ifdef DRIVER
main ( int argc, char *argv[] )
{
    int i;

    if ( argc < 2 )
    {
        fprintf ( stderr, "Generates checksum using 7 mod 11.\n"
                "Requires a command line # to validate.\n" );
        return ( EXIT_FAILURE );
    }

#ifdef VERBOSE
    printf ( "Computing 7 mod 11 checksum for: %s\n", argv[1] );
#endif

    i = Get7Mod11Chksum ( argv[1], strlen ( argv[1] ));

    if ( i == -1 )
        fprintf ( stderr,
                  "Number must be ten digits or less\n" );

    if ( i == -2 )
        fprintf ( stderr, "Number is invalid.\n" );

    return ( EXIT_SUCCESS );
}
#endif
```

从表 10-2 中可以看到，12345 有 4 种可能的换位方式。注意：所有这些换位都不会生成与原始数字相同的加权校验和。

表 10-2　12345 的可能的换位方式以及相应的校验和

换　位	和	校 验 和
21345	5325	1
13245	3561	8
12435	3309	9
12354	3273	6

捕获两种换位是一项困难得多的任务，并且这种错误可以轻松地逃脱检查。不过，在现实中，这种情况应该不会频繁发生。如果键盘穿孔机操作员定期产生两种换位错误，就有需要处理的质量问题。当然，这种质量很快就会被检测到，因为校验和将报告太多的错误需要处理，以便让工作能够继续下去，除非操作员变得更准确。

在选定一种校验和方法之前，需要测试该方法，以便查看它能够多好地避免为换位生成与原始数字相同的校验和，你将发现这样做是有用的。程序清单 10-3（cksmtest. c）执行这种测试。它创建一个表 table，用于保存 100 个随机的 10 位数字。然后，它遍历该表，并为其中的每个数字以及该数字的所有可能的换位计算校验和。它将记录（在 collisions 中）换位的数字生成与原始数字相同校验和的次数。然后，它会生成包含 100 个 10 位数字的新表，但是只使用每个数字的前 9 位来测试 9 位数字上的换位。它将以这种方式继续，直至它测试了 100 个各种大小（从 10 位到 2

位）的随机数字。

程序清单 10-3　这个程序使用各种可能的换位，对 100 个随机数字测试多种加权校验和方法

```
/*--- cksmtest.c ------------------------ Listing 10-3 ------
 *  Test checksum methods to see the distribution of checksums
 *  when transpositions occur. Performs the testing on 100
 *  random 10-digit numbers all the way to 100 2-digit
 *  numbers. Benign transpositions (see text) are not included.
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

char table [100] [10]; /* the table of random digits */
                       /* max: 100 numbers of 10 digits each */

void DisplayUsage ( void );
void LoadTableRandom ( char * );
int  GetChecksum ( char *, int, int );

int main( int argc, char *argv[] )
{
    int  i, j, size, method;
    char *p, *pmax;
    char buffer[10];

    p = (char *) table;     /* start of table */
    pmax = p + 1000;        /* end of table */

    if ( argc != 2 )
    {
        DisplayUsage();
        return ( EXIT_FAILURE );
    }
    else
        method = atoi ( argv[1] );

    switch ( method )
    {
        case 1: printf ( "Using 7 mod 11\n\n" );    break;
        case 2: printf ( "Using 9 mod 11\n\n" );    break;
        case 3: printf ( "Using 7 mod 26\n\n" );    break;
        case 4: printf ( "Using 7 mod 10\n\n" );    break;
        case 5: printf ( "Using 9 mod 10\n\n" );    break;
        default:
                DisplayUsage();
                return ( EXIT_FAILURE );
    }

    /* we loop through for every size from 10 to 2.
     * We don't handle 1-digit numbers since they
     * offer no possibility of transpositions.
     */
```

```
 for ( size = 10; size > 1; size-- )
 {
     int collisions = 0;
/* Load the table with random digits */

LoadTableRandom ( pmax );

/* main test loop: we process 100 numbers
 * of length size. For each number, we do every
 * possible transposition and get its checksum.
 */

for ( i = 0, p = (char *)table; i < 100;
                                    i++, p += size )
{
    int k, l, orig_cksum;

    /* get the original number's checksum */

    orig_cksum = GetChecksum ( p, size, method );

    /* now copy the number to buffer, where we'll
     * do the transpositions.
     */

    memcpy ( buffer, p, size );

    /* do the transpositions and checksums */

    for ( k = 0; k < size - 1; k++ )
    {
        /* is there a transposition possible?
         * if both digits are the same, they cannot
         * be transposed; so skip these, otherwise
         * they will generate bogus collisions.
         */

        if ( buffer[k] == buffer[k + 1] )
            continue;

        /* otherwise, do the transposition */

        l = buffer[k + 1];
        buffer[k + 1] = buffer[k];
        buffer[k] = (char) l;

        /* get the checksum */

        j = GetChecksum ( buffer, size, method );
            if ( j == orig_cksum )
                collisions += 1;

            /* now undo this transposition */

            l = buffer[k + 1];
            buffer[k + 1] = buffer[k];
            buffer[k] = (char) l;
```

```
            }

        }   /* end of loop processing numbers of one size */

        printf ( "%2d digits avg. %1.2f collisions\n",
                    size, (float) ( collisions / 100.0 ));

        collisions = 0;
    }

    return ( EXIT_SUCCESS );
}

void LoadTableRandom ( char *table_end )
{
    int i;
    unsigned short int j;
    char *p, *p2;

    /* 200 random ints will get us close to 1000 digits
     * we presume it gets us at least 500 digits, which is
     * a safe bet, but will not always be the case.
     */

    p = (char *) table;

    for ( i = 0; i < 200; i++ )
    {
        j = (short int) rand();
        sprintf ( p, "%u", j );
        p += strlen ( p );
    }

    /* how many digits left to fill out? */

    i = table_end - p;

    p2 = p - 1;

    /* go backwards from p and copy digits into the
     * remaining table space. This will copy i digits
     * prior to p and place them in reverse order after p.
     */
    while ( i )
    {
        *p++ = *p2--;
        i--;
    }
}

int GetChecksum ( char *data, int length, int which_test )
{
    int  i, j;
    int  position;
    unsigned long int sum;
    long int *powers;
```

```
    static long int powers_of_7 [10] = { 1, 7, 49, 343, 2401,
            16807, 117649, 823543, 5764801, 40353607 };

    static long int powers_of_9 [10] = { 1, 9, 81, 729, 6561,
            59049, 531441, 4782969, 43046721, 387420489 };

    if ( which_test == 2 || which_test == 5 )
        powers = powers_of_9;
    else
        powers = powers_of_7;

    /* make sure all digits can be looked up */

    if ( length > 10 )
        return ( -1 );

    sum = 0L;
    position = 0; /* what digit in the data are we at? */

    for ( i = length; i > 0; i--, position++ )
    {
        /* do we have a digit ? */

        if ( ! isdigit ( data[i-1] ))
            return ( -2 );

        /* convert digit character to a number */

        j = data[i-1] - '0';

        /* look up power, multiply by j, add to sum */
        sum += powers [position] * j;
    }

    /* get the sum modulus depending on method */

    switch ( which_test )
    {
        case 1:
        case 2:
            i = (int) ( sum % 11L );
            if ( i == 10 )
                i = 0;
            break;
        case 3:
            i = (int) ( sum % 26L );    break;
        case 4:
        case 5:
            i = (int) ( sum % 10L );    break;
        default:
            DisplayUsage();
            exit ( EXIT_FAILURE );
    }

    return ( i );
}
```

```
void DisplayUsage()
{
    fprintf ( stderr,
        "Error: need to specify checksum method to test\n"
        "1. Powers of 7 Mod 11\n"
        "2. Powers of 9 Mod 11\n"
        "3. Powers of 7 Mod 26\n"
        "4. Powers of 7 Mod 10\n"
        "5. Powers of 9 Mod 10\n" );
}
```

该程序使用了以前提到的两种方法——使用7的幂对11求模（7 mod 11）和使用9的幂对11求模（9 mod 11）——以及一个额外的方法：7 mod 26。表10-3中显示了这个程序使用7 mod 11和9 mod 11的结果。它还像以前所提到的那样测试了7 mod 10和9 mod 10。

表10-3 两种不同的校验和方法的结果，其中显示了具有与原始数字相同校验和的换位数量（对于100个随机数字）

数 字	7 mod 11	9 mod 11	数 字	7 mod 11	9 mod 11
10	9	15	5	4	7
9	11	13	4	6	7
8	12	12	4	6	7
7	10	7	2	3	2
6	5	6			

如果运行程序清单10-3中的程序，将很可能获得与表10-3中有所不同的结果。这是由于编译器库的rand()函数生成的数字序列中的变化。即使这个表基于各种大小的100个随机数字实例，结果也极大地依赖于生成的测试数字。表10-3显示了在Borland、Microsoft和Novell的UnixWare的编译器上运行该程序的平均结果。作为变化的演示，利用Borland的编译器，使用9 mod 11方法为10位的数字运行测试将产生16次冲突，而利用UNIX产品，则只会产生10次冲突。

从该表中可以看出：在发生换位时，为了避免校验和冲突，使用7 mod 11方法更好一些。表中的数字显示：当发生任何换位时，对于7位或更长的数字，加权校验和一般可以捕获的数字刚好在90%以下，而当数字在7位以下时，它捕获的数字将超过90%。

不过，这些结果具有欺骗性。在表10-3中，我们看到在100个10位的随机数字中，当我们使用7 mod 11方法时，将有9种不引人注意的换位。在事实上，百分比非常接近于100%的准确性，因为每个10位的数字都有9种可能的换位。因此，在900种可能的换位中，将只有9种不会被检测到。而且，在900种可能的换位中，有10%是良性的，因为它们是相同数字的换位。例如，在数字1234567890中，可能的换位（从左到右）如下：

2-1、3-2、4-3、5-4、6-5、7-6、8-7、9-8、0-9（9种换位和9种可能的错误）

不过，在1223445678中，将有如下9种可能的换位：

2-1、2-2、3-2、4-3、4-4、5-4、6-5、7-6、8-7（9种换位和7种可能的错误）

由于相同数字的换位（上一个例子中的2-2和4-4）不构成错误，就可以从校验和需要捕获的可能错误池中去除它们。由于任何数字后面接着另一个匹配数字的几率为10%，因此在900种可能的换位中，只有810种是错误。在这810种错误中，7 mod 11校验和将捕获除9种错误以外的

其他所有错误。因此，它的错误捕获率将达到99%，即使平均只能保证91%的10位数字是正确的。

如果这些结果不充分，还有几种可能的改进方法。其中相当重要的改进是使数字的某些部分有意义，使得其他数据可以验证其正确性。不过，如果必须使用较长的无意义数字，就要使校验和是一个字母而不是一个数字。这将提供26种可能的校验和，而不是单个数字允许的10种校验和。例如，查看程序清单10-3中的校验和方法3。它使用7的幂对26求模。这种改变的所得到的结果是戏剧性的：除了最大的数字之外，其他所有数字中的冲突都降至0。在长度多达10位的数字中，这种方法几乎会捕获由单个换位组成的100%的数据输入错误。

有趣的是，最后这种方法没有得到广泛使用。其主要原因是由于速度问题。如果数据输入专业人员不必把他们的手从键盘上移开以便输入单个字母，那么他们可以快得多地使用10个键的数字键盘。在一串数字中输入单个字母通常需要把两只手都放到键盘上，以便为单次按键定位手指，或者需要操作员低头看键盘以定位字母。这必定会减慢输入速度。与以前一样，这是速度与准确度之间的折衷。注意：如果选择使用字母校验和，就应该避免使用像O和I这样的字母，因为很可能把它们与数字0和1弄混淆。在程序清单10-3所示的代码中，应该显式测试校验和O和I，并把它们映射到其他字母。

提高校验和准确度的另一种解决方案是使用两位数字的校验和（利用模101）。这种方法可以确保几乎100%的准确度，同时使得操作员的手不必离开数字键盘。不过，由于输入两个额外的数字（而不是一个数字）需要花费更多的时间并且由于额外的字节将会占据的额外磁盘空间，大多数数据处理方法都回避了这种方法。

在涉及较长数字的实际情况下，一般的趋势是使用条形码来确保速度和准确度。条形码现在是标准的，其中必须快速输入较长的数字。考虑联邦快递公司或其他运输公司的空运订单，或者大型供货商的配件清单。这些公司使用条形码作为数据输入的主要手段，同时仍然在要编制条形码的数字中保留校验数字。这样，无论是扫描还是键入数字，都可以保持较高的准确度。

10.3 循环冗余校验

在前面关于校验和的讨论中，我们发现简单的校验和在标记数据错误方面表现得比较好，而加权校验和则对发现数据输入中的错误非常有效。不过，这两种形式的校验和都受限于它们可以验证的数据的大小。如以前所讨论的，简单的校验和对于超过256字节的数据不准确；加权校验和对于长度在12～15位以上的数字不实用，因为7或9的幂变得太大，以至于不能在C语言的整数数据类型内执行处理。由于这种限制，就需要生成非常准确地验证大数据块的校验和。而且，由于涉及大量的数据，算法必须快速工作。在远程通信这个领域中，往往需要这种验证。在远程通信中，把大数据块从一个站点发送到另一个站点。接收站点必须能够确定它接收到的数据是否与发送给它的数据完全相同。为了完成这个任务，现代远程通信协议使用称为**循环冗余校验**（cyclic redundancy check，CRC）的校验和。这种校验和并不像其名称听起来那样使人畏缩不前。这个名称本身揭示了该算法的起源，并且没有特殊的意义——不像散列算法中的“散列”（参见第3章）和红黑树中的分类法（参见第6章）。

CRC并不是起源于远程通信中，而是起源于计算机硬件中。最初，CRC用于验证计算机内的

数据传输。直到今天，CRC 仍然被磁盘驱动器用于验证数据块的读取和写入。

CRC 计算一个数字，它反映了数据块中各个位的值和位置。它通过以多项式的形式构造各个位的指数来执行该任务：

$$bit_{n-1} * x^{n-1} + bit_{n-2} * x^{n-2} + \ldots + bit_0 * x^0$$

（x 的值比较特殊，稍后将进行讨论）。在我们研究的 *CRC* 的实现中，这个多项式一次确定 16 位。也就是说，$n=16$。然后用这个多项式除以两个多项式之一，这依赖于我们决定使用的 CRC 的形式。如果我们使用 CRC-16（磁盘驱动器的控制器中使用的 CRC 算法），那么用作除数的多项式如下：

$$x^{16} + x^{15} + x^2 + 1$$

如果使用 CRC-CCITT（CCITT 是世界范围内的通信标准组织），那么用作除数的多项式如下：

$$x^{16} + x^{12} + x^5 + 1$$

CRC-CCITT 用在 Xmodem-CRC 协议中（普通的 Xmodem 使用 1 字节的校验和）和 IBM 的 SDLC/HDLC 远程通信协议的早期版本中。之所以使用这两个特定的除数，是因为相当多的数学资料指示这样做，Joseph Campbell 和 David Schwaderer 解释了其中大部分资料［Campbell 1987；Schwaderer 1988］，无需广泛的数学知识即可理解它们。这使我们能够明确地说：这些多项式的除法可以给出 CRC 值的宽泛分布，并且可以使用移位的方法快速完成。

10.3.1 CRC-CCITT

实际的 CRC 值是上述除法的余数。例如，如果我们想要获得用于 ASCII 字母 g 的 CRC-CCITT 值，则将执行以下计算。字母 g 是十六进制的 67。因此，它的二进制表示是 01100111。把它构造成以前描述的多项式形式，将获得以下多项式：

$$0x^7 + 1x^6 + 1x^5 + 0x^4 + 0x^3 + 1x^2 + 1x^1 + 1x^0$$

删除系数为 0 的项并简化余下的项，当我们除以 CRC-CCITT 多项式时，将得到以下方程：

$$\text{CRC-CCITT（g）} = \frac{x^6 + x^5 + x^2 + x + 1}{x^{16} + x^{12} + x^5 + 1} \text{的余数}$$

初看上去，它看上去像是一个几乎不可能的方程，并且为了快速呈现一个值，它必须会涉及非常多的困难计算（指数和除法）。不过，如前所述，要小心选择 CRC 多项式和 x 的神奇值，以便只使用位操作即可执行这个除法运算。程序清单 10-4（ccittotf.c，动态用于 CCITT）通过一次一个字节地计算 CRC，显示了如何对一个文件计算 CRC-CCITT。第一个函数 GetCCITT() 执行实际的计算，并返回一个字节的 CRC。然后把返回的值传递回函数，以便验证下一个字节。当检查了文件中的所有字节时，将会显示最终的 CRC 值。这个值就是用于整个文件的 CRC。

程序清单 10-4　为整个文件一次一个字节地动态计算 CRC-CCITT

```
/*--- CCITTOTF.C  ------------------------ Listing 10-4 -----
 * Compute CCITT-CRC on-the-fly.
 * Usage:   ccittotf filename
 *
 * Based on a similar program by Nigel Cort in C Gazette 5.1
 * (Autumn 1990).
 *-----------------------------------------------------------*/
```

```
#include <stdio.h>
#include <stdlib.h>

#define DRIVER 1 /* DRIVER includes the file processing */


/*  Compute single-byte CRC-CCITT on-the-fly */

unsigned short GetCCITT ( unsigned short crc, unsigned short ch )
{
    static unsigned int i;
    ch <<= 8;                              /* Move to MSB       */
    for ( i = 8; i > 0; i-- )            /* Go through 8 bits */
    {
        if ( (ch ^ crc) & 0X8000 )       /* Perform CRC calc. */
             crc = ( crc << 1 ) ^ 0x1021;
        else
             crc <<= 1;
        ch <<= 1;
    }
    return ( crc );
}

#ifdef DRIVER
int main ( int argc, char * argv[] )
{
   FILE *fin;             /* file we're reading into buffer */
   char *buffer;          /* buffer we're working on        */
   size_t i, j;           /* counters of bytes in buff      */
   unsigned short crc;    /* the CRC value being computed   */

   if ( argc < 2 )
   {
      fprintf ( stderr, "Error! Must specify filename.\n" );
      return ( EXIT_FAILURE );
   }

   if (( fin = fopen ( argv[1], "rb" )) == NULL )
   {
      fprintf ( stderr, "Cannot open %s", argv[1] );
      return ( EXIT_FAILURE );
   }

   /*-------------------------------------------------
    * Set up a very large input buffer of 32K bytes.
    * This program does no good if it doesn't fly!
    *-------------------------------------------------*/

   if (( buffer = (char *) malloc ( 32766 )) == NULL )
   {
      fprintf ( stderr, "Out of memory\n" );
      return ( EXIT_FAILURE );
   }

   crc = 0;

   for (;;)
```

```
    {
        i = fread ( buffer, 1, 32766, fin );
        if ( i == 0 )
        {
            if ( feof ( fin ))  /* we're done, so show results */
            {
                printf ( "CCITT CRC for %s is %04X\n",
                                                argv[1], crc );
                return ( EXIT_SUCCESS );
            }
            else          /* read another 32K of file */
                continue;
        }

        for ( j = 0 ; j < i; j ++ )  /* loop through the buffer */
              crc = GetCCITT ( crc, buffer [j] );
    }
}
#endif
```

在程序清单10-4所示的实现中，在CRC值的计算中0x1021这个值看起来很显眼。这是移位操作中使用的神奇值，用于镜像以前提到的 x 的神奇值。如果你想知道如何把计算移入一系列移位操作中以及这个神奇的值为什么是必要的，将需要深入学习关于CRC的数学知识。以前提及的Schwaderer和Campbell所著的书籍就是良好的起点。

显然，CCITT多项式中使用的多项式具有用于其分母的常量值。而且，由于一个字节中最多有256种可能的位组合，因此多项式的除法中将只有256种可能的分子。在知道这一点后，就可以创建一个CRC-CCITT值的表，它可以用于查找某个字节的单个CRC，而不用动态计算CRC。通过使用CRC的查找表，可以把程序清单10-4中的程序构造成更高效地运行，虽然其代价是额外消耗内存中1024个字节的空间。图10-2中显示了这些值的表，就像它在C程序中看起来那样。

```
/* CCITT Lookup Table */
unsigned short ccitt_table[256] =
{

/*   0 -- */  0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
/*   8 -- */  0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
/*  16 -- */  0x1231, 0x0210, 0x3273, 0x2252, 0x52B5, 0x4294, 0x72F7, 0x62D6,
/*  24 -- */  0x9339, 0x8318, 0xB37B, 0xA35A, 0xD3BD, 0xC39C, 0xF3FF, 0xE3DE,
/*  32 -- */  0x2462, 0x3443, 0x0420, 0x1401, 0x64E6, 0x74C7, 0x44A4, 0x5485,
/*  40 -- */  0xA56A, 0xB54B, 0x8528, 0x9509, 0xE5EE, 0xF5CF, 0xC5AC, 0xD58D,
/*  48 -- */  0x3653, 0x2672, 0x1611, 0x0630, 0x76D7, 0x66F6, 0x5695, 0x46B4,
/*  56 -- */  0xB75B, 0xA77A, 0x9719, 0x8738, 0xF7DF, 0xE7FE, 0xD79D, 0xC7BC,
/*  64 -- */  0x48C4, 0x58E5, 0x6886, 0x78A7, 0x0840, 0x1861, 0x2802, 0x3823,
/*  72 -- */  0xC9CC, 0xD9ED, 0xE98E, 0xF9AF, 0x8948, 0x9969, 0xA90A, 0xB92B,
/*  80 -- */  0x5AF5, 0x4AD4, 0x7AB7, 0x6A96, 0x1A71, 0x0A50, 0x3A33, 0x2A12,
/*  88 -- */  0xDBFD, 0xCBDC, 0xFBBF, 0xEB9E, 0x9B79, 0x8B58, 0xBB3B, 0xAB1A,
/*  96 -- */  0x6CA6, 0x7C87, 0x4CE4, 0x5CC5, 0x2C22, 0x3C03, 0x0C60, 0x1C41,
/* 104 -- */  0xEDAE, 0xFD8F, 0xCDEC, 0xDDCD, 0xAD2A, 0xBD0B, 0x8D68, 0x9D49,
```

图10-2 用于每个可能字节的CRC-CCITT值的源代码表

```
/* 112 -- */  0x7E97, 0x6EB6, 0x5ED5, 0x4EF4, 0x3E13, 0x2E32, 0x1E51, 0x0E70,
/* 120 -- */  0xFF9F, 0xEFBE, 0xDFDD, 0xCFFC, 0xBF1B, 0xAF3A, 0x9F59, 0x8F78,
/* 128 -- */  0x9188, 0x81A9, 0xB1CA, 0xA1EB, 0xD10C, 0xC12D, 0xF14E, 0xE16F,
/* 136 -- */  0x1080, 0x00A1, 0x30C2, 0x20E3, 0x5004, 0x4025, 0x7046, 0x6067,
/* 144 -- */  0x83B9, 0x9398, 0xA3FB, 0xB3DA, 0xC33D, 0xD31C, 0xE37F, 0xF35E,
/* 152 -- */  0x02B1, 0x1290, 0x22F3, 0x32D2, 0x4235, 0x5214, 0x6277, 0x7256,
/* 160 -- */  0xB5EA, 0xA5CB, 0x95A8, 0x8589, 0xF56E, 0xE54F, 0xD52C, 0xC50D,
/* 168 -- */  0x34E2, 0x24C3, 0x14A0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
/* 176 -- */  0xA7DB, 0xB7FA, 0x8799, 0x97B8, 0xE75F, 0xF77E, 0xC71D, 0xD73C,
/* 184 -- */  0x26D3, 0x36F2, 0x0691, 0x16B0, 0x6657, 0x7676, 0x4615, 0x5634,
/* 192 -- */  0xD94C, 0xC96D, 0xF90E, 0xE92F, 0x99C8, 0x89E9, 0xB98A, 0xA9AB,
/* 200 -- */  0x5844, 0x4865, 0x7806, 0x6827, 0x18C0, 0x08E1, 0x3882, 0x28A3,
/* 208 -- */  0xCB7D, 0xDB5C, 0xEB3F, 0xFB1E, 0x8BF9, 0x9BD8, 0xABBB, 0xBB9A,
/* 216 -- */  0x4A75, 0x5A54, 0x6A37, 0x7A16, 0x0AF1, 0x1AD0, 0x2AB3, 0x3A92,
/* 224 -- */  0xFD2E, 0xED0F, 0xDD6C, 0xCD4D, 0xBDAA, 0xAD8B, 0x9DE8, 0x8DC9,
/* 232 -- */  0x7C26, 0x6C07, 0x5C64, 0x4C45, 0x3CA2, 0x2C83, 0x1CE0, 0x0CC1,
/* 240 -- */  0xEF1F, 0xFF3E, 0xCF5D, 0xDF7C, 0xAF9B, 0xBFBA, 0x8FD9, 0x9FF8,
/* 248 -- */  0x6E17, 0x7E36, 0x4E55, 0x5E74, 0x2E93, 0x3EB2, 0x0ED1, 0x1EF0
};
```

图 10-2 （续）

尝试把这个表手动输入到程序中是愚蠢的。这样做将导致出现一个错误，并且用于文件的最终的 CRC 值将是不正确的。一种更简单的方法是让计算机生成这个表。程序清单 10-5 是一个短小的程序，它用于生成与图 10-2 中所示的完全相同的表。

程序清单 10-5 用于创建 CRC-CCITT 查找表的程序

```
/*---mkccitt.c ------------------------- Listing 10-5 -------
 *  Makes a CRC-CCITT table for lookups in C-usable
 *  format. Outputs to screen; Redirect to disk as needed.
 *  For example:
 *                  mkccitt > ccitt.tbl
 *
 *  Based on code from Nigel Cort in C Gazette 5.1 (Autumn 1990)
 ------------------------------------------------------------*/

#include <stdio.h>

unsigned short CalculateCCITT ( unsigned int );

unsigned short table [256];

void main ( void )
{
   unsigned int i;

   /* first calculate the CRC values */

   for ( i = 0; i < 256; i++ )
      table [i] = CalculateCCITT ( i );

   /* then print out the table */
```

```
    printf ( "/* CCITT Lookup Table */ \n" );
    printf ( "unsigned short ccitt_table[256] =\n{" );

    for ( i = 0; i < 256; i++ )
    {
       /* start a new row every eight CRCs */

       if (( i % 8 ) == 0 )
          printf ( "\n  /* %3u -- */  ", i );

       /* print the CRC */

       printf ( "0x%04X", table [i] );
       if ( i != 255 )
          printf ( ", " );
    }

    /* print closing brace for table */

    printf ( "\n};\n" );
}

unsigned short CalculateCCITT ( unsigned int index )
{
    unsigned short a, i;
    a = 0;          /* serves as an accumulator */

    index <<= 8;

    /* The heart of the CRC-CCITT */

    for ( i = 8; i > 0; i-- )
    {
       if (( index ^ a ) & 0x8000 )
          a = ( a << 1 ) ^ 0x1021;
       else
          a <<= 1;
       index <<= 1;
    }

    return ( a );
}
```

为了使用图 10-2 中所示的表，必须稍加修改程序清单 10-4 中所示的 CRC-CCITT 程序。当然，删除了用于生成 CRC 的函数；然后把表查找操作插入到主处理循环中。程序清单 10-6（crcccitt. c）中显示了用于使用这个表的完整程序。注意：这个程序清单与程序清单 10-4 中的 main( ) 函数几乎完全相同。只改变了执行查找的最后两行代码。另请注意利用#include 包括了图 10-2 中的表。

程序清单 10-6　利用图 10-2 中的查找表计算 CRC-CCITT 的程序

```
/*--- CRCCCITT.C  ------------------------ Listing 10-6 --------
 * Compute CCITT-CRC using a lookup table
 * Usage:   crcccitt filename
 *
 * Based on a similar program by Nigel Cort in C Gazette 5.1
```

```
 * (Autumn 1990).
 *-------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>

#include "ccitt.tbl"  /* generated by mkccitt.c (Listing 10-5) */

int main ( int argc, char * argv[] )
{
   FILE *fin;             /* file we're reading into buffer */
   char *buffer;          /* buffer we're working on        */
   size_t i, j;           /* counters of bytes in buff      */
   unsigned short crc;    /* the CRC value being computed   */

   if ( argc < 2 )
   {
      fprintf ( stderr, "Error! Must specify filename.\n" );
      return ( EXIT_FAILURE );
   }

   if (( fin = fopen ( argv[1], "rb" )) == NULL )
   {
      fprintf ( stderr, "Cannot open %s\n", argv[1] );
      return ( EXIT_FAILURE );
   }

   /*-------------------------------------------------
    * Set up a very large input buffer of 32K bytes.
    * This program does no good if it doesn't fly!
    *-------------------------------------------------*/

   if (( buffer = (char *) malloc ( 32766 )) == NULL )
   {
      fprintf ( stderr, "Out of memory\n" );
      return ( EXIT_FAILURE );
   }

   crc = 0;

   for (;;)
   {
      i = fread ( buffer, 1, 32766, fin );
      if ( i == 0 )
      {
         if ( feof ( fin ))  /* we're done, so show results */
         {
            printf ( "CRC-CCITT for %s is %04X\n",
                                          argv[1], crc );
            return ( EXIT_SUCCESS );
         }
         else          /* read another 32K of file */
            continue;
      }

      for ( j = 0 ; j < i; j ++ )  /* loop through the buffer */
        crc =
```

```
            (crc << 8) ^ ccitt_table [ (crc >> 8) ^ buffer [j] ];
    }
}
```

10.3.2 CRC-16

如前所述，CRC 算法的另一个版本是 CRC-16。它使用不同的多项式，以前展示了它。程序清单 10-7（crc16.c）中展示的 CRC-16 的实现显示了一种稍微不同的方法，用于使用 CRC 的查找表。这个程序清单（基于 William James Hunt 的著作）使用只有 16 个条目的非常短的表，并且会查找每半个字节（即 4 位）。这种方法提供了比动态计算更快的速度，但它会节省上一个程序清单中的表所消耗的一些空间。由于这种方法很好地协调了速度和大小，因此它非常适合于嵌入式程序，在这种程序中，内存非常珍贵，但是速度仍然是极其重要的。

程序清单 10-7　为文件生成 CRC-16 值的程序

```
/*--- crc16.c ---------------------------- Listing 10-7 ------
*   Performs CRC-16 check on a file.
*   Usage: crc16 finename
*
*   Modeled on William Hunt's work and Nigel Cort's port.
*-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>

#define DRIVER 1    /* brings in the main line */


/* Compute a CRC-16 value */

unsigned short int GetCRC16 (
            int start,      /* starting value */
            const char *p,  /* points to chars to process */
            int n )         /* how many chars to process */
{

    static unsigned int crc16_table[16] =   /* CRC-16s */
    {
          0x0000, 0xCC01, 0xD801, 0x1400,
          0xF001, 0x3C00, 0x2800, 0xE401,
          0xA001, 0x6C00, 0x7800, 0xB401,
          0x5000, 0x9C01, 0x8801, 0x4400
    } ;

    unsigned short int total;  /* the CRC-16 value we compute */
    int r1;

    total = start;

    /* process each byte */

    while ( n-- > 0 )
    {
```

```
        /* do the lower four bits */

        rl = crc16_table[ total & 0xF ];
        total = ( total >> 4 ) & 0x0FFF;
        total = total ^ rl ^ crc16_table[ *p & 0xF ];

        /* do the upper four bits */

        rl = crc16_table[ total & 0xF ];
        total = ( total >> 4 ) & 0x0FFF;
        total = total ^ rl ^ crc16_table[ ( *p >> 4 ) & 0xF ];

        /* advance to next byte */
        p++;
    }
    return ( total ) ;
}


#ifdef DRIVER
#define BUFSIZE 8192

main ( int argc, char *argv[] )
{
    FILE *fin;
    int  n;         /* number of bytes actually read */
    unsigned short int  crc;       /* CRC value */
    char *buffer;  /* buffer into which we read file */

    if ( argc < 2 )
    {
        fprintf ( stderr, "Error: must provide filename\n" );
        return ( EXIT_FAILURE );
    }

    if (( fin = fopen ( argv[1], "rb" )) == NULL )
    {
        fprintf ( stderr, "Cannot open %s \n", argv[1] );
        return ( EXIT_FAILURE );
    }

    buffer = (char *) malloc ( BUFSIZE );
    if ( buffer == NULL )
    {
        fprintf ( stderr, "Error allocating memory\n" );
        return ( EXIT_FAILURE );
    }

    crc = 0;

    /* principal processing loop */

    do
    {
        /* read file in 8KB blocks */

        n = fread ( buffer, 1, BUFSIZE, fin );
```

```
        /* Fold in this block's CRC-16 */

        crc = GetCRC16 ( crc, buffer, n );
    }
    while ( n == BUFSIZE );    /* Stop when done */

    fclose ( fin ) ;

    /* check for read error */

    if ( ferror ( fin ))
        fprintf ( stderr, "Error in Processing %s\n", argv[1] );

    /* No error, so report CRC-16 */

    else
        printf ( "CRC-16 for  %s = %04X\n", argv[1], crc );

    return ( EXIT_SUCCESS );
}
#endif
```

除了极少见的例外之外（将在下一节中讨论），CRC-16 和 CRC-CCITT 在捕获小数据块上的错误方面非常有效。Josepn Campbell 引用了以下统计数据，其中显示了数据块的大小以及捕获的错误率［Campbell 1987］：

≤16 位	100%
17 位	99.9969%
>17 位	99.984%

后两个数字是平均数字。显然，涉及的位越多，忽略错误的可能性就越大。迄今为止介绍的两种 CRC 方法都是 16 位的方法；也就是说，它们会生成 16 位的 CRC。如果要验证的数据块超过了 4 KB，16 位 CRC 的错误率将显著恶化，使得不再能够确保几乎无缺陷的数据完整性。对于更大的数据块，需要 32 位的 CRC：即 CRC-32。

10.3.3 CRC-32

本章前面介绍的 16 位 CRC 是用于 16 位校验和的算法的经典实现。例如，CRC-CCITT 是在 Xmodem-CRC 中发现的相同 CRC。不过，16 位 CRC 具有一个严重的缺点：它们不能捕获丢弃了数据块的前导 0 位的情况。关于它的解释是合乎逻辑的。CRC 的初始值被设置为 0。你将注意到在 16 位 CRC 使用的表中（参见图 10-2），用于 0 字节的值也是 0。因此，由两个 0 字节组成的数据块所具有的 CRC 将与由 5 个 0 字节组成的数据块相同。因此，如果发送 5 个字节的块，但是只有两个字节到达，那么 CRC 将不会检测到这种错误。

Xmodem-CRC 解决了这种问题，因为它使用 128 字节的块。在检查每个块末尾的 CRC 值之前，Xmodem-CRC 会检查它是否接收到了完整的 128 字节。这样，将会在执行 CRC 计算之前检测到任何丢弃的 0 字节。同样，如果使用本章中介绍的 16 位 CRC 比较两个文件，在可以检查 CRC 值看看它们是否匹配之前，必须首先检查两个文件的大小看看它们是否相等。

CRC-32 通过把 CRC 的值初始化为 0xFFFFFFFF（而不是 0）来解决这种问题。使用非 0 的

初始值被称为**预处理**（preconditioning）。CRC-32 也通过改变最终的 CRC 值实现了**后处理**（post-conditioning）。它交换了最终的 CRC 值中的位。后处理似乎不会解决特定的完整性所关心的问题。它只是用于 CRC-32 的 CCITT 规范的一部分。CRC 的其他变体使用不同的预处理和后处理方法。

用于 CRC-32 的除法多项式如下：

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x^{1}+1$$

与 CRC-CCITT 一样，这里使用的方法是创建一个表，并且在这个表中查找各个字节的 CRC。这个表如图 10-3 所示。

```
/* CRC-32 Lookup Table */
unsigned long crc32_table[256] =
{
  /*   0 -- */  0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA,
  /*   4 -- */  0x076DC419, 0x706AF48F, 0xE963A535, 0x9E6495A3,
  /*   8 -- */  0x0EDB8832, 0x79DCB8A4, 0xE0D5E91E, 0x97D2D988,
  /*  12 -- */  0x09B64C2B, 0x7EB17CBD, 0xE7B82D07, 0x90BF1D91,
  /*  16 -- */  0x1DB71064, 0x6AB020F2, 0xF3B97148, 0x84BE41DE,
  /*  20 -- */  0x1ADAD47D, 0x6DDDE4EB, 0xF4D4B551, 0x83D385C7,
  /*  24 -- */  0x136C9856, 0x646BA8C0, 0xFD62F97A, 0x8A65C9EC,
  /*  28 -- */  0x14015C4F, 0x63066CD9, 0xFA0F3D63, 0x8D080DF5,
  /*  32 -- */  0x3B6E20C8, 0x4C69105E, 0xD56041E4, 0xA2677172,
  /*  36 -- */  0x3C03E4D1, 0x4B04D447, 0xD20D85FD, 0xA50AB56B,
  /*  40 -- */  0x35B5A8FA, 0x42B2986C, 0xDBBBC9D6, 0xACBCF940,
  /*  44 -- */  0x32D86CE3, 0x45DF5C75, 0xDCD60DCF, 0xABD13D59,
  /*  48 -- */  0x26D930AC, 0x51DE003A, 0xC8D75180, 0xBFD06116,
  /*  52 -- */  0x21B4F4B5, 0x56B3C423, 0xCFBA9599, 0xB8BDA50F,
  /*  56 -- */  0x2802B89E, 0x5F058808, 0xC60CD9B2, 0xB10BE924,
  /*  60 -- */  0x2F6F7C87, 0x58684C11, 0xC1611DAB, 0xB6662D3D,
  /*  64 -- */  0x76DC4190, 0x01DB7106, 0x98D220BC, 0xEFD5102A,
  /*  68 -- */  0x71B18589, 0x06B6B51F, 0x9FBFE4A5, 0xE8B8D433,
  /*  72 -- */  0x7807C9A2, 0x0F00F934, 0x9609A88E, 0xE10E9818,
  /*  76 -- */  0x7F6A0DBB, 0x086D3D2D, 0x91646C97, 0xE6635C01,
  /*  80 -- */  0x6B6B51F4, 0x1C6C6162, 0x856530D8, 0xF262004E,
  /*  84 -- */  0x6C0695ED, 0x1B01A57B, 0x8208F4C1, 0xF50FC457,
  /*  88 -- */  0x65B0D9C6, 0x12B7E950, 0x8BBEB8EA, 0xFCB9887C,
  /*  92 -- */  0x62DD1DDF, 0x15DA2D49, 0x8CD37CF3, 0xFBD44C65,
  /*  96 -- */  0x4DB26158, 0x3AB551CE, 0xA3BC0074, 0xD4BB30E2,
  /* 100 -- */  0x4ADFA541, 0x3DD895D7, 0xA4D1C46D, 0xD3D6F4FB,
  /* 104 -- */  0x4369E96A, 0x346ED9FC, 0xAD678846, 0xDA60B8D0,
  /* 108 -- */  0x44042D73, 0x33031DE5, 0xAA0A4C5F, 0xDD0D7CC9,
  /* 112 -- */  0x5005713C, 0x270241AA, 0xBE0B1010, 0xC90C2086,
  /* 116 -- */  0x5768B525, 0x206F85B3, 0xB966D409, 0xCE61E49F,
  /* 120 -- */  0x5EDEF90E, 0x29D9C998, 0xB0D09822, 0xC7D7A8B4,
  /* 124 -- */  0x59B33D17, 0x2EB40D81, 0xB7BD5C3B, 0xC0BA6CAD,
  /* 128 -- */  0xEDB88320, 0x9ABFB3B6, 0x03B6E20C, 0x74B1D29A,
  /* 132 -- */  0xEAD54739, 0x9DD277AF, 0x04DB2615, 0x73DC1683,
  /* 136 -- */  0xE3630B12, 0x94643B84, 0x0D6D6A3E, 0x7A6A5AA8,
  /* 140 -- */  0xE40ECF0B, 0x9309FF9D, 0x0A00AE27, 0x7D079EB1,
```

图 10-3　CRC-32 的查找表

```
  /* 144 -- */  0xF00F9344, 0x8708A3D2, 0x1E01F268, 0x6906C2FE,
  /* 148 -- */  0xF762575D, 0x806567CB, 0x196C3671, 0x6E6B06E7,
  /* 152 -- */  0xFED41B76, 0x89D32BE0, 0x10DA7A5A, 0x67DD4ACC,
  /* 156 -- */  0xF9B9DF6F, 0x8EBEEFF9, 0x17B7BE43, 0x60B08ED5,
  /* 160 -- */  0xD6D6A3E8, 0xA1D1937E, 0x38D8C2C4, 0x4FDFF252,
  /* 164 -- */  0xD1BB67F1, 0xA6BC5767, 0x3FB506DD, 0x48B2364B,
  /* 168 -- */  0xD80D2BDA, 0xAF0A1B4C, 0x36034AF6, 0x41047A60,
  /* 172 -- */  0xDF60EFC3, 0xA867DF55, 0x316E8EEF, 0x4669BE79,
  /* 176 -- */  0xCB61B38C, 0xBC66831A, 0x256FD2A0, 0x5268E236,
  /* 180 -- */  0xCC0C7795, 0xBB0B4703, 0x220216B9, 0x5505262F,
  /* 184 -- */  0xC5BA3BBE, 0xB2BD0B28, 0x2BB45A92, 0x5CB36A04,
  /* 188 -- */  0xC2D7FFA7, 0xB5D0CF31, 0x2CD99E8B, 0x5BDEAE1D,
  /* 192 -- */  0x9B64C2B0, 0xEC63F226, 0x756AA39C, 0x026D930A,
  /* 196 -- */  0x9C0906A9, 0xEB0E363F, 0x72076785, 0x05005713,
  /* 200 -- */  0x95BF4A82, 0xE2B87A14, 0x7BB12BAE, 0x0CB61B38,
  /* 204 -- */  0x92D28E9B, 0xE5D5BE0D, 0x7CDCEFB7, 0x0BDBDF21,
  /* 208 -- */  0x86D3D2D4, 0xF1D4E242, 0x68DDB3F8, 0x1FDA836E,
  /* 212 -- */  0x81BE16CD, 0xF6B9265B, 0x6FB077E1, 0x18B74777,
  /* 216 -- */  0x88085AE6, 0xFF0F6A70, 0x66063BCA, 0x11010B5C,
  /* 220 -- */  0x8F659EFF, 0xF862AE69, 0x616BFFD3, 0x166CCF45,
  /* 224 -- */  0xA00AE278, 0xD70DD2EE, 0x4E048354, 0x3903B3C2,
  /* 228 -- */  0xA7672661, 0xD06016F7, 0x4969474D, 0x3E6E77DB,
  /* 232 -- */  0xAED16A4A, 0xD9D65ADC, 0x40DF0B66, 0x37D83BF0,
  /* 236 -- */  0xA9BCAE53, 0xDEBB9EC5, 0x47B2CF7F, 0x30B5FFE9,
  /* 240 -- */  0xBDBDF21C, 0xCABAC28A, 0x53B39330, 0x24B4A3A6,
  /* 244 -- */  0xBAD03605, 0xCDD70693, 0x54DE5729, 0x23D967BF,
  /* 248 -- */  0xB3667A2E, 0xC4614AB8, 0x5D681B02, 0x2A6F2B94,
  /* 252 -- */  0xB40BBE37, 0xC30C8EA1, 0x5A05DF1B, 0x2D02EF8D
};
```

图 10-3 （续）

程序清单 10-8（mkcrc32. c）中展示了将生成这个表的程序。除了一些计算以及显示中的微小变化之外，这个程序看起来与程序清单 10-5（mkccitt. c）中所示的程序非常相似。程序清单 10-8 中使用的方法基于 Mark R. Nelson 所著的文章［Nelson 1992］。在 Schwaderer 的著作中可以找到一种用于构造这个表的差别很大的方法［Schwaderer 1988］。

程序清单 10-8 创建 CRC-32 查找表的程序

```
/*--- mkcrc32.c ------------------------- Listing 10-8 ------
 *  Makes a CRC-32 table for 32-bit lookups in C-usable
 *  format. Outputs to screen; Redirect to disk as needed.
 *  For example:
 *                  mkcrc32 > crc32.tbl
 *
 *  Adapted from code by Mark Nelson (DDJ, May 1992)
 ------------------------------------------------------------*/

#include <stdio.h>

unsigned long table[ 256 ];
```

```
unsigned long CalculateCRC32 ( int );

void main ( void )
{
   unsigned int i;

   /* first calculate the CRC-32 values */

   for ( i = 0; i < 256; i++ )
      table [i] = CalculateCRC32 ( i );

   /* then print out the table */

   printf ( "/* CRC-32 Lookup Table */ \n" );
   printf ( "unsigned long crc32_table[256] =\n{" );

   for ( i = 0; i < 256; i++ )
   {
      /* start a new row every four CRCs */

      if (( i % 4 ) == 0 )
         printf ( "\n  /* %3u -- */  ", i );

      /* print the CRC */

      printf ( "0x%08lX", (unsigned long) table [i] );
      if ( i != 255 )
          printf ( ", " );
   }

   /* print closing brace for table */

   printf ( "\n};\n" );
}

unsigned long CalculateCRC32 ( int i )
{
    int j;
    unsigned long crc;
    crc = i;

    for ( j = 8 ; j > 0; j-- )
    {
        if ( crc & 1 )
            crc = ( crc >> 1 ) ^ 0xEDB88320L;
        else
            crc >>= 1;
    }
    return ( crc );
}
```

对文件的实际 CRC 处理发生在程序清单 10-9（crc32. c）中。这个程序看起来与程序清单 10-6 中所示的程序非常相似，因为它们使用的机制非常相似：读取文件，并使用图 10-3 中所示的查找表逐字节计算 CRC-32，在程序中利用#include 命令包括了该表。注意在返回值之前变量 crc 的预处理及其后处理。

程序清单 10-9 通过使用查找表计算 CRC-32 的程序

```
/*--- crc32.c ---------------------------- Listing 10-9 -------
 * Compute CRC-32 using a lookup table
 * Usage:  crc32 filename
 *-------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>

#include "crc32.tbl"  /* generated by mkcrc32.c (Listing 10-8) */

int main ( int argc, char * argv[] )
{
   FILE *fin;              /* file we're reading into buffer */
   unsigned char *buffer; /* buffer we're working on        */
   size_t i, j;           /* counters of bytes in buff      */
   int k;                 /* generic integer                */
   unsigned long crc;     /* the CRC value being computed   */

   if ( argc < 2 )
   {
      fprintf ( stderr, "Error! Must specify filename.\n" );
      return ( EXIT_FAILURE );
   }

   if (( fin = fopen ( argv[1], "rb" )) == NULL )
   {
      fprintf ( stderr, "Cannot open %s\n", argv[1] );
      return ( EXIT_FAILURE );
   }

   /*-----------------------------------------------
    * Set up a very large input buffer of 32K bytes.
    * This program does no good if it doesn't fly!
    *-----------------------------------------------*/

   if (( buffer = (unsigned char *) malloc ( 32766 )) == NULL )
   {
      fprintf ( stderr, "Out of memory\n" );
      return ( EXIT_FAILURE );
   }

   /* preconditioning sets crc to an initial nonzero value */

   crc = 0xFFFFFFFF;

   for (;;)
   {
      i = fread ( buffer, 1, 32766, fin );
      if ( i == 0 )
      {
         if ( feof ( fin ))  /* we're done, so show results */
         {
            /* postconditioning inverts the bits in CRC */
```

```
            crc = ~crc;

            /* now print the result */

            printf ( "CRC-32 for %s is %08lX\n",
                                          argv[1], crc );
            return ( EXIT_SUCCESS );
         }
         else          /* read another 32K of file */
            continue;
      }

      for ( j = 0; j < i; j ++ )  /* loop through the buffer */
      {
         k = ( crc ^ buffer[j] ) & 0x000000FFL;
         crc = (( crc >> 8 ) & 0x00FFFFFFL ) ^ crc32_table[k];
      }
   }
}
```

应该使用哪一种 CRC 呢？如果正在远程通信装置中发送数据分组，机会就是协议将选择合适的 CRC。不过，如果分组大于 4 KB，就一定要使用 CRC-32，而不是 16 位。如果使用 CRC 来验证两个二进制文件具有相同的内容（比如在测试数据压缩/解压缩方法中），则 CRC-32 方法更可取。它适合于较大的文件，并且由于它使用预处理，所以它消除了首先验证文件大小的需要。出于这种原因，数据压缩实用程序（比如 PKZIP 的当前版本）使用 CRC-32 进行文件验证（使用 PKZIP 的-v 选项查看每个压缩文件的 CRC-32 结果）。在第 9 章“数据压缩”中非常详细地讨论了数据压缩算法与 CRC 之间的关系。

10.4 资源和参考资料

Campbell, Joseph. *C Programmer's Guide to Serial Communications*, Indianapolis, IN: Howard Sams & Co., 1987. 这是针对 PC 程序员的关于远程通信的权威书籍。尽管自从本书最初问世起由于硬件有了相当大的改进而使本书显得有些过时，但是该书中关于调制解调器编程原理的解释以及 CRC 中位操作的细节介绍特别清晰易懂。

Intel Corporation. *8086 Relocatable Object Module Formats*, order #121748-001. Santa Clara, CA: Intel Corporation, 1981. 尽管当 PC 初次亮相时这篇文档已经有些过时，但它是许多年来操作系统、编译器、汇编器和连接器的编写者使用的唯一权威性文档。它后来被［Intel 1993］（参见该文档）所取代。

Intel Corporation. *Tool Interface Standards: Portable Formats Specification*, order # 241597-002. Santa Clara, CA: Intel Corporation, 1993. 这篇文档（可以从 Intel 免费获取）指定了用于 MS-DOS、UNIX 和 MS Windows 的目标文件格式。它还包含关于调试记录的格式的信息。

Nelson, Mark R. “File Verification Using CRC: 32-bit Cyclic Redundancy Check.” *Dr. Dobb's Journal*, May 1992, p. 64.

Schwaderer, David. *C Programmer's Guide to NetBIOS*. Indianapolis, IN: Howard Sams & Co., 1988. 本书包含关于 CRC 的良好技术性概述，并且带有用于编码它们的一些不同寻常的实现。

U. S. Postal Service. *Preparing Business and Courtesy Reply Mail*. Publication 12, September 1984. 可以从大多数美国邮局获得它。

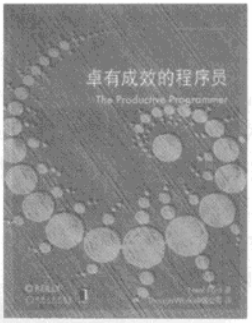
HZ BOOKS
华章图书

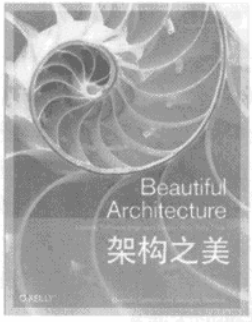
华章经典

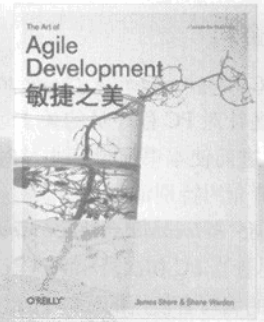
为每一个团队提供最有价值的阅读服务

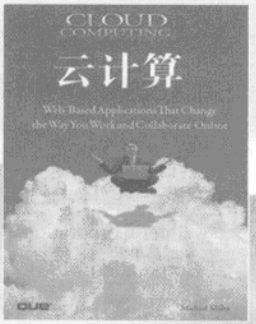
每一本书都值得您和您的团队一起阅读

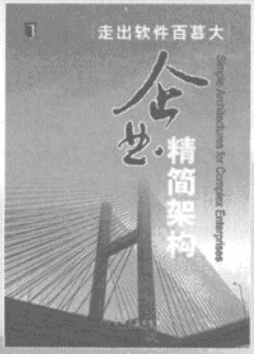
分享阅读分享成功

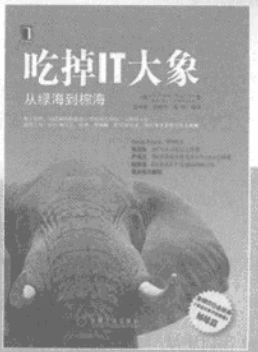




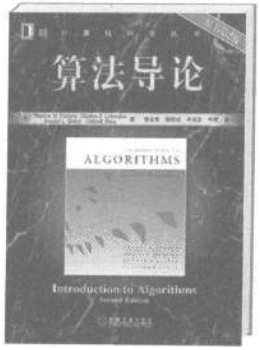








经典推荐

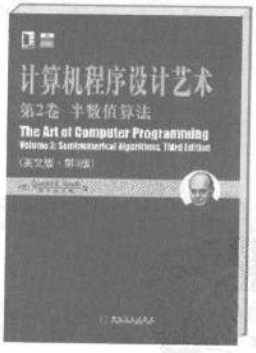


算法导论 (原书第2版)
作 者：Thomas H.Cormen 等
译 者：潘金贵 顾铁成 等
书 号：7-111-18777-6
定 价：85.00元

计算机算法的设计与分析
作 者：Alfred V. Aho;
John E.Hopcroft;
Jeffrey D. Ullman
译 者：黄林鹏 王德俊 张仕
书 号：978-7-111-21543-1
定 价：49.00元

算法概论注释版
作 者：Sanjoy Dasgupta;
Christos Papadimitriou;
Umesh Vazirani
译 者：钱枫 邹恒明
书 号：978-7-111-25361-7
定 价：55.00元

计算机程序设计艺术
第1卷 基本算法（英文版·第3版）
作 者：Donald E. Knuth
书 号：978-7-111-22709-0
定 价：95.00元



计算机程序设计艺术
第2卷 半数值算法（英文版·第3版）
作 者：Donald E. Knuth
书 号：978-7-111-22718-2
定 价：109.00元

计算机程序设计艺术
第3卷 排序和查找（英文版·第2版）
作 者：Donald E. Knuth
书 号：978-7-111-22717-5
定 价：95.00元

www.hzbook.com

填写读者调查表　加入华章书友会
获赠精彩技术书　参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：　　书号：7-111-(　　)

姓名：	性别：☐男　☐女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐朋友推荐　☐书店　☐图书目录　☐杂志、报纸、网络等　☐其他

2. 您从哪里购买本书：

☐新华书店　☐计算机专业书店　☐网上书店　☐其他

3. 您对本书的评价是：

技术内容	☐很好	☐一般	☐较差	☐理由_____
文字质量	☐很好	☐一般	☐较差	☐理由_____
版式封面	☐很好	☐一般	☐较差	☐理由_____
印装质量	☐很好	☐一般	☐较差	☐理由_____
图书定价	☐太高	☐合适	☐较低	☐理由_____

4. 您希望我们的图书在哪些方面进行改进？

_____

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

_____

6. 您有没有写作或翻译技术图书的想法？

☐是，我的计划是_____☐否

7. 您希望获取图书信息的形式：

☐邮件　☐信函　☐短信　☐其他_____

请寄：北京市西城区百万庄南街1号　机械工业出版社　华章公司　计算机图书策划部收
邮编：100037　电话：（010）88379512　传真：（010）68311602　E-mail: hzjsj@hzbook.com