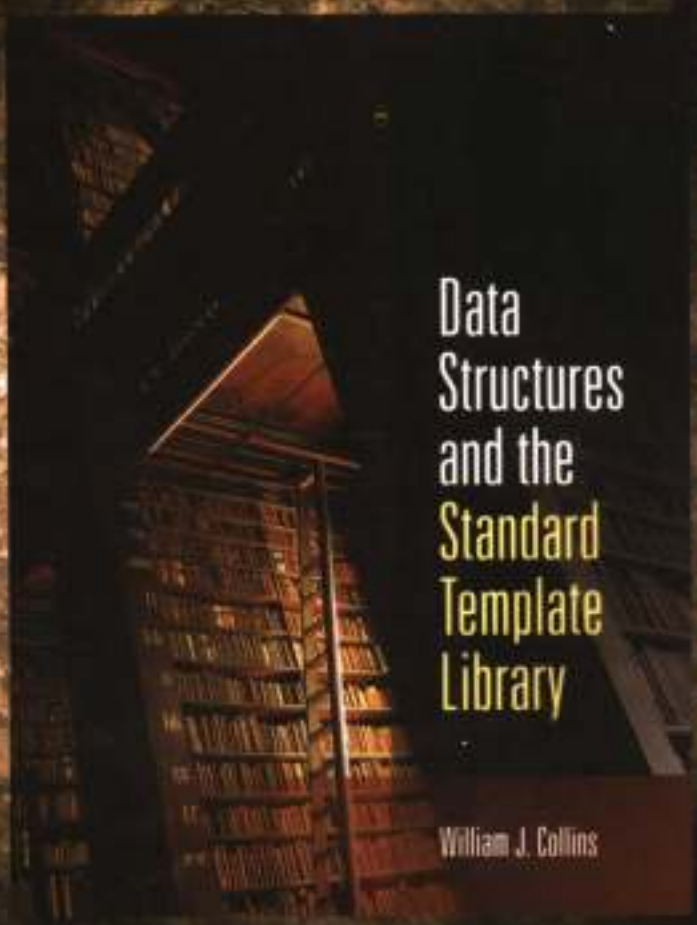
Education 计 算 机 科 学 丛 书

数据结构与STL

(美) William J. Collins 著　　周翔 译



Data Structures and the Standard Template Library

机械工业出版社
China Machine Press

本书讲述了数据结构的基本原理及其实现，并使用了C++作为教学语言。通过方法接口，示例和应用的学习，引导学生逐渐理解和掌握如何高效地使用数据结构。大部分数据结构是在标准模板库（STL）中提供的。本书还详细研究了这些STL数据结构的规范实现，展示了这些实现的高效和简洁性。为了深入理解实现的要点，还对其中几个数据结构的不同实现进行了测试。

贯穿全书的宗旨是鼓励结合实践的学习。每章末尾的编程项目让学生可以开发并实现自己的数据结构，或者是扩展、应用这一章中介绍的数据结构。可选的实验帮助学生通过编程巩固所学知识。

本书特点：

- 本书配套网站上包含了实验、课件、习题解答等等，网站地址是 www.mhhe.com/collins。
- 每个实验都要求学生进行仔细的观察、推测和检测才能得出结论，能够鼓励学生积极主动地学习。
- 书中还精心设计了许多教学提示和习题。

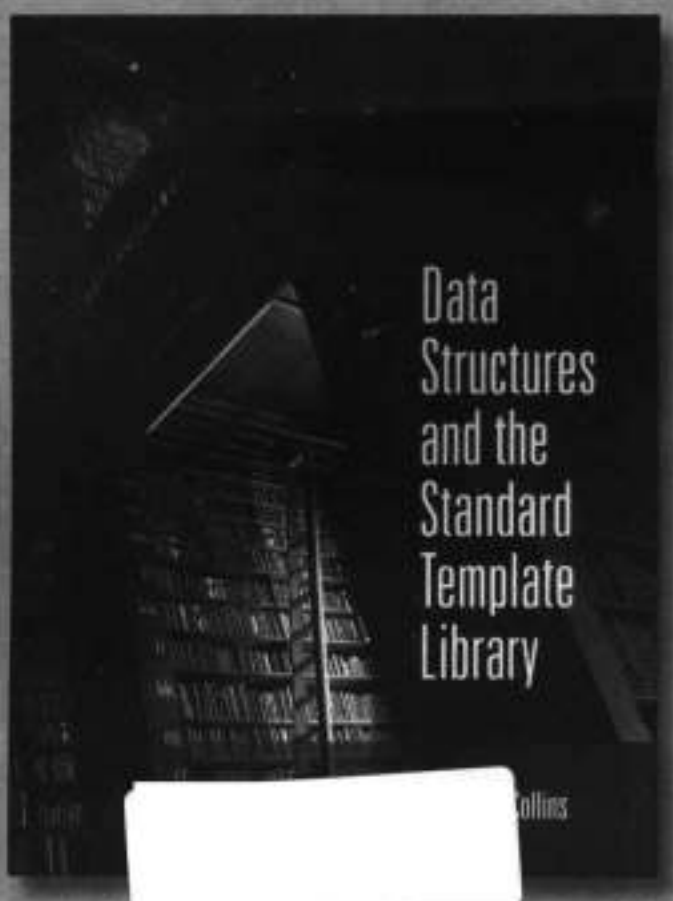
ISBN 7-111-13962-3
9 787111 139621
华章图书
网上书店：www.china-pub.com
北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259，68995264
读者服务信箱：hzedu@hzbook.com
http://www.hzbook.com
ISBN 7-111-13962-3/TP · 3461
定价：49.00 元

TP311.12
60

计 算 机 丛 书

数据结构与STL

（美）William J. Collins 著　　周翔 译



Data Structures and the Standard Template Library

机械工业出版社
China Machine Press

数据结构一直是计算机科学专业课程的核心内容，它是信息的组织方式。对于相同的算法，用不同的数据结构表示其中的抽象数据类型会造成不同的执行效率。

本书从面向对象程序设计的角度，具体使用C++语言，讲述了数据结构及其算法。通过对方法接口、示例和应用的学习，引导学生逐渐理解和掌握如何高效地使用数据结构。

本书与传统数据结构教材相比，除了保留系统、全面的风格之外，还具有重视与实际编程结合、侧重标准模板库的实现描述等特点；并配有丰富的习题及实验，是一本优秀的课堂和自学参考用书。

William J. Collins: Data Structures and the Standard Template Library (ISBN 0-07-236935-5)

Copyright © 2003 by the McGraw-Hill Companies, Inc.

Original English edition published by The McGraw-Hill Companies, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition jointly published by McGraw-Hill Education (Asia) Co. and China Machine Press.

本书中文简体字翻译版由机械工业出版社和美国麦格劳-希尔教育(亚洲)出版公司合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有McGraw-Hill公司防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书版权登记号：图字：01-2003-1004

图书在版编目（CIP）数据

数据结构与STL /（美）柯林斯（Collins, W. J.）著；周翔译. -北京：机械工业出版社，2004.4

（计算机科学丛书）

书名原文：Data Structures and the Standard Template Library

ISBN 7-111-13962-3

Ⅰ.数… Ⅱ.①柯… ②周… Ⅲ.①数据结构 ②C语言-程序设计 Ⅳ.①TP311.12 ②TP312

中国版本图书馆CIP数据核字（2004）第014412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：刘立卿

北京瑞德印刷有限公司印刷 · 新华书店北京发行所发行

2004年4月第1版第1次印刷

787mm × 1092mm 1/16 · 34.25 印张

印数：0 001 - 4 000 册

定价：49.00 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

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

秘　书　组

武卫东　　温莉芳　　刘　江　　杨海玲

译 者 序

数据结构一直是计算机科学专业课程的核心内容。它是信息的组织方式。对于相同的算法，用不同的数据结构表示其中的抽象数据类型会造成不同的执行效率。因此，对数据结构的研究一直是计算机科技工作者努力的方向。本书从面向对象程序设计语言的角度讲述数据结构及其算法，使用了惠普公司提供的标准模板库（STL）作为基础，代码具有简洁性和高效率的特点，有很重要的实践意义。

在教材内容的安排上，本书仍按照各种不同的数据结构分类进行系统的介绍。在给出了面向对象程序设计及容器类的一般概念之后，通过堆栈、队列、树等常用数据结构剖析了方法接口及其简单的应用。随着C++的广泛应用，容器和模板也越来越受到编程者的重视。以往的教材虽然也详尽地介绍了各种数据结构，但总局限于理论上的探讨；与此不同的是，本书更强调标准模板库的使用，而不仅仅是一般的结构和算法。在介绍每种结构时，都尽可能地融入了它在STL中的表现形式和接口，这样就较好地解决了数据结构和实际编程应用脱节的问题。

本书与传统数据结构教材相比，除了保留系统、全面的风格之外，还增加了以下的新特点：

- 重视数据结构与实际编程的结合。书中的源代码都使用C++语言编写，并进行了验证，更方便有编程需要的读者理解和使用。
- 侧重STL。书中提供了大量的方法接口及实例分析。
- 使用了大量的文本和图表进行辅助说明。
- 每章都包含了章节目标及丰富的习题，对了解学习要求和进一步深入学习提供了很好的帮助。

对于学习和了解数据结构的本、专科学生及研究生而言，本书可作为他们的教材和教学参考书；而对于编程人员、技术服务人员以及程序使用者而言，本书也是一本很好的参考读物。

本书由周翔翻译，在翻译过程中得到了隋立恒、王勇等的帮助。由于本书的内容新，涉及面广，加之译者水平有限，书中难免会存在一些问题，恳请读者提出批评意见。

译 者

2003年9月

前　言

本书讲述了数据结构及算法。实现语言选用了C++，这适用于已经学习过相关基础课程的学生。这些课程并不一定是面向对象的，但应当覆盖基本语句和数据类型，比如数组和文件处理的基础。

标准模板库

本书的显著特点是其依据为标准模板库（STL）——这个库由惠普公司提供。使用这种方法有几个优点。首先，这些代码是被详尽检测过的；其次，读者通过本书有机会学习到以前没有接触过的专业代码，它是相当高效率和简洁的；第三，这个库在今后的课程中也是非常有用的。

大多数情况下库不会描述数据结构的实现。这是有好处的，我们可以将注意力放在功能而不是实现细节上。关于这些类的定义，可以参阅惠普研究实验室的Stepanov等人的原始实现（参见Stepanov and Lee，1994）。这个惠普实现是作者所知的全部实现的基础。

考虑过的其他实现

与标准模板库的惠普实现同样重要的是，本书不是只关注于数据结构和算法的基础课程。和惠普实现不同的方法也是值得考虑的。例如，**list**类的实现使用了有头节点的双向链表，它和单链表以及有头尾域的双向链表是不同的。另外本书还比较了不同实现的差异。当然，还有一些数据结构（像图）和算法（像回溯）是不在标准模板库里的。

本书也可以满足数据结构和算法课程的基本需要：让学生练习开发他们自己的数据结构。书中有许多编程项目，它们的数据结构要么是从头创建的，要么是从章中的范例扩展而来的。另外还有一些项目，是开发或扩展那些使用标准模板库的应用程序。

标准C++

所有的代码都是基于ANSI/ISO标准C++的，并且在Windows平台（C++ Builder和Visual C++）和Unix平台(G++)上测试过。标准模板库遵循的规范（不是指具体实现）是ANSI/ISO C++的一部分。

教学方法的特点

本书有几个特点，这些可以改善教学者的教学环境以及学生的学习环境。每章开头都给出了目标，末尾至少给出一个主要编程任务。每个数据结构都描述得很详细，每个方法都有一个前置条件和后置条件。另外，大部分方法都给出了调用示例以及调用结果。

对细节问题，特别是标准模板库的惠普实现，进行了认真的研究，并在29个实验中得到补充。参阅前言的“实验的组织”部分可以了解更多关于这些实验的信息。每章后都有多种

习题，教师可以使用这些习题的答案。

辅助材料

所有的辅助材料都放在网站上：www.mhhe.com/collins。

网站为学生提供了下列信息的链接：

- 实验的观察以及入门方法。
- 本文开发的所有项目的源代码。
- 项目的Applet，它有一个很强的可视化组件。

另外，教师可以从网站获取下列信息：

- 教师关于实验的选择。
- 每章的PowerPoint幻灯片（大约1500张幻灯片）。
- 每章习题的答案，PowerPoint里展示的习题以及实验的答案。

章节纲要

第1章介绍了作为后续章节基础的C++的特点。大部分材料表现的都是面向对象技术：类，继承，构造器，析构器以及运算符的重载。通过实验来回顾类、继承和运算符的重载。

第2章介绍了容器类以及和容器类存储相关的问题。顺序存储和链式存储都需要使用指针。为了说明链式存储，创建了一个单链表类。这个过分简单的Linked类为描述标准模板库的几个关键特色（如模板、迭代器和通用型算法）提供了背景。相关的实验是基于指针、迭代器、运算符重载和通用型算法的。

第3章是软件工程简介，概述了开发软件的四个步骤：分析，设计，实现和维护。使用统一建模语言（UML）作为设计工具来描述继承、复合和聚合。贯穿后续章节的大O表示法可用于脱离环境来评估方法的时间代价。本章还探讨了带驱动器的运行时评估和计时，而且每个主题都对应一个实验。

第4章讲述了递归，它将重点暂时从数据结构转移到算法上。介绍了回溯，把回溯作为解决问题的一般技术。并采用了相同的BackTrack类，用于搜索迷宫、在棋盘上放置八个皇后（使她们不能互相攻击）等应用，并阐述了一个马可以遍历棋盘上的每个空格，而且每个空格只经过一次。其他的递归调用，如汉诺塔游戏和生成置换，更进一步地突出了递归的优雅，尤其是将它和对应的迭代实现相比较时。在后面的章节里也会遇到递归，特别是快速排序法的实现和二叉树的定义中。此外，对每个专业程序员而言，递归是一个必不可缺的工具。

在第5章里，开始使用vector和deque类学习标准模板库。向量是一个灵巧的数组：自动调整大小，并配有方法处理任意位置的插入和删除操作。而且，向量是模板化的，因此将**int**类型元素插入**int**类型向量的方法和将**string**类型元素插入**string**类型向量的方法是完全相同的。设计过程从vector类中最常用方法的**方法接口**（前置条件，后置条件和方法头）开始，然后是惠普的大致实现和实验中的进一步的细节。vector类的应用、高精度的算法是公钥加密算法的基础。这个应用在实验和编程项目里进行了更深层次的研究。队列和向量很相似，至少从数据结构的角度来说是这样的。但是实现细节上仍有较大区别，这些细微差异将在实验里探讨。

第6章描述了list数据结构和类，它们的特征是，方法花费线性时间进行随机插入、删除和

检索。这个属性迫使它们使用**链表迭代器**：遍历list对象，并使用常数时间的方法在当前位置上插入、删除和检索对象。本章也介绍了双链式的环型实现，并在一个实验中介绍了其他细节。其应用是一个小的行编辑器，这很适合链表迭代器。在编程项目中对该应用做了进一步的扩展。还有关于迭代器种类，以及向量、双端队列和链表的运行时间比较的实验。

queue和stack类是第7章的主题。这两个类都是**容器配接器**：它们采用了其他一些类的方法接口。对queue和stack类而言，缺省的“其他”类都是deque类。因而得到的stack和queue类的方法定义都是单行的。队列的具体应用——计算洗车处的平均等待时间，属于计算机仿真的范畴。stack类有两个应用：递归的实现，以及中缀到后缀的转换。后一个应用在实验中进行了扩展，并构成了“求一个条件的值”编程项目的基础。

第8章的重点是一般的二叉树，特别是折半查找树。介绍了二叉树的基本特点；这些对理解AVL树、红黑树、堆、霍夫曼树和决策树是很重要的。折半查找树类是红黑树的惠普实现的单色版本。

第9章中考察了AVL树。作为实现重新平衡的机制，介绍了旋转。借助于斐波纳契树，确定了AVL树的高度总是和树中项的数量成对数关系。AVLTree类不是标准模板库的组成部分，但是包含了几个重要的特色，如函数对象；还对该难题提供了后续的实验研究。除了erase方法（编程项目9.1）之外，整个类都得到了实现。AVL树的应用是一个简单的拼写检查器。

红黑树在第10章中进行了研究。仔细研究了红黑树中的插入和删除算法，并且提供了相关的实验。红黑树不在标准模板库中，但是它们是标准模板库中四个关联容器类（set、map、multiset和multimap）大部分实现的基础。在一个集合里，每个项只由一个键组成，是不允许重复键的。多集合允许重复键。在一个映射里，每个项只由一个惟一的键部分和另一部分组成。多映射允许重复键。本章还介绍了一个应用（用来计算文件中每个单词出现的频率），以及有关四个关联容器类的实验。

第11章介绍了priority_queue类，即另一个容器配接器。缺省使用的是vector类，但幕后还有一个堆，使得以常数平均时间进行插入，而且即使在最坏情况下，也能以对数时间删除最高优先级的元素。可以考虑基于list或基于set的实现。它的应用是在数据压缩领域，特别是霍夫曼编码：给定一个文本文件，生成一个最小的无前缀编码。编程项目的任务是将编码转换回原来的文本文件。实验将公平性融进了优先队列，这样即便是同为最高优先级的项，也总是先处理在优先队列中滞留时间最长的。

排序是第12章的主题。开发了基于比较的排序的最小上界的估算。研究了四种“快速”排序：树排序（多集合），堆排序（随机访问容器），归并排序（列表）和快速排序（随机访问容器）。本章的实验在随机产生的数值上比较了这些排序方法。编程项目是排序姓名和社会保障号码文件。

第13章先开始复习了顺序和折半查找，然后研究了散列。通常，在标准C++或标准模板库的惠普实现中都不支持散列类。本章开发了一个hash_map类。这个类的方法接口和map类的是相同的，除了插入，删除或查找的平均时间花费是常数而不是对数时间！应用包含了字符表的创建和维护，以及对第9章中拼写检查器应用的修正。还比较了链式散列和开放地址散列；在编程项目中进一步探索了它们的不同。hash_map类的速度是实验的主题。

第14章介绍了最常用的数据结构——图、树和网络。给出了基本算法的框架：广度优先迭代，深度优先迭代，连通性，寻找最小生成树以及查找两个顶点间的最短路径。开发的惟一

的类是使用邻接表实现的network类。其他类（如undirected_graph和undirected_network）可以直接定义为network类的子类。在实验中研究了货郎担问题，并且提出了一个编程项目，要求完成network类的邻接矩阵版本。本章还提出了另一个回溯应用，使用的仍然是第4章所介绍的BackTrack类。

每一章都对应一个相关的网页，其中包括了该章中开发的所有程序以及适于阐释概念的Applet。

附录

附录1包含了有助于学生理解书中数学概念的背景信息。累加符号和对数的初步性质是最基本的，而数学归纳原理使得我们能更深地分析二叉树和开放地址散列。

string类是附录2的主题。这个功能强大的类是标准模板库的一个重要组成部分，并使得学生可以绕开乏味的字符数组。

多态性是一个指针引用对象层次中不同对象的能力，在附录3中进行了说明。多态性是面向对象编程的一个基本特性，但被归入了附录，因为它并不是介绍数据结构和算法所必需的论题。

实验的组织

本书中共涉及到29个网络实验。学生和教师可以访问的URL是:

www.mhhe.com/collins

实验不只包含了基本素材，还提供了对文字材料的补充。例如，在研究了vector、deque和list类之后，用一个实验对这三个类进行了一些时间测试。

实验是独立的，因此教师可以很灵活地指定实验。可以将它们指定为:

1) 闭卷实验

2) 开卷实验

3) 不计分作业

除了能明显地提高学习积极性，这些实验还能鼓励学生运用科学的方法。学生们观察到一些现象，如标准模板库的list类的组织。然后阐明并提交一个关于该现象的假设——以及他们自己的代码。测试之后可能需要修正他们的假设，提交根据实验得到的最终结论。

Bill Collins

目　录

出版者的话
专家指导委员会
译者序
前言
第1章　C++中的类 ……1
1.1　类 ……1
1.1.1　方法接口 ……1
1.1.2　对象 ……2
1.1.3　数据抽象 ……4
1.1.4　构造器 ……6
1.1.5　一个Employee类 ……7
1.1.6　Employee类的定义 ……11
实验1：Company项目 ……13
1.1.7　继承 ……13
1.1.8　受保护的访问 ……14
1.1.9　HourlyEmployee类 ……15
实验2：关于继承的更多的细节 ……18
1.1.10　运算符的重载 ……19
1.1.11　友元 ……20
实验3：重载运算符“=”和运算符“>>” …21
1.1.12　信息隐藏 ……21
总结 ……22
习题 ……22
编程项目1.1：一个Sequence类 ……25
第2章　容器类的存储结构 ……27
2.1　指针 ……27
2.1.1　堆和堆栈的对比 ……29
2.1.2　引用参数 ……29
2.1.3　指针字段 ……30
2.1.4　数组和指针 ……30
实验4：指针变量赋值与动态变量赋值的对比 ……31
2.1.5　动态变量的存储空间释放 ……31
2.2　数组 ……32
2.3　容器类 ……33
2.3.1　容器类的存储结构 ……33
2.3.2　链式结构 ……34
2.3.3　迭代器 ……37
2.3.4　Iterator类的设计和实现 ……39
实验5：定义其他的迭代器运算符 ……40
2.3.5　pop_front方法 ……40
2.3.6　析构器 ……41
实验6：重载运算符operator = ……42
2.3.7　通用型算法 ……42
实验7：更多关于通用型算法的知识 ……46
2.3.8　数据结构和标准模板库 ……46
总结 ……46
习题 ……47
编程项目2.1：扩展Linked类 ……47
第3章　软件工程简介 ……49
3.1　软件开发生命周期 ……49
3.2　问题分析 ……49
3.3　程序设计 ……51
3.3.1　方法接口和字段 ……51
3.3.2　依赖关系图 ……53
3.4　程序实现 ……54
3.4.1　方法验证 ……55
实验8：驱动器 ……56
3.4.2　正确性实现的可行性 ……56
3.4.3　方法效率评估 ……56
3.4.4　大O表示法 ……57
3.4.5　快速获取大O估算 ……59
3.4.6　平衡折中 ……62
3.4.7　运行时间分析 ……63
3.4.8　随机性 ……65
实验9：计时和随机性 ……66
3.4.9　类型转换 ……66
3.5　程序维护 ……67
总结 ……67
习题 ……67
编程项目3.1：Linked类的进一步扩充 ……69
第4章　递归 ……71

4.1 简介 ……71
4.2 阶乘 ……71
4.3 十进制到二进制的转换 ……75
实验10：斐波纳契数 ……78
4.4 汉诺塔 ……78
4.5 回溯 ……86
4.6 折半查找 ……97
实验11：迭代折半查找 ……106
4.7 生成置换 ……106
4.8 间接递归 ……114
4.9 递归的代价 ……115
总结 ……116
习题 ……116
编程项目4.1：汉诺塔的迭代版本 ……121
编程项目4.2：八皇后问题 ……122
编程项目4.3：马的遍历问题 ……123
第5章 向量和双端队列 ……127
5.1 标准模板库 ……127
5.2 向量 ……128
5.2.1 vector类的方法接口 ……129
5.2.2 向量迭代器 ……134
5.2.3 向量和其他容器的对比 ……136
5.2.4 vector类可能的字段 ……137
5.2.5 vector类的一个实现 ……137
实验12：vector类的更多的实现细节 ……142
5.3 向量的一个应用：高精度算法 ……142
5.3.1 very_long_int类的设计 ……143
5.3.2 very_long_int类的一个实现 ……144
实验13：扩展very_long_int类 ……147
5.4 双端队列 ……147
实验14：惠普的deque类实现的更多细节 ……154
5.5 双端队列的一个应用：非常长的整数 ……154
总结 ……154
习题 ……155
编程项目5.1：扩展very_long_int类 ……157
编程项目5.2：deque类的另一种实现 ……157
第6章 表 ……159
6.1 表 ……159
6.1.1 list类的方法接口 ……160
6.1.2 迭代器接口 ……163
6.1.3 链表方法和向量或双端队列方法的差别 ……165
6.1.4 list类的字段和实现 ……166
6.1.5 list节点的存储 ……171
实验15：更多list类的实现细节 ……174
实验16：计时顺序容器 ……174
实验17：迭代器，第二部分 ……174
6.1.6 list类的其他实现 ……175
6.2 链表应用：一个行编辑器 ……177
6.2.1 Editor类的设计 ……180
6.2.2 Editor类的实现 ……182
总结 ……187
习题 ……187
编程项目6.1：扩展Editor类 ……189
编程项目6.2：list类的另一种设计和实现 ……195
第7章 队列和堆栈 ……197
7.1 队列 ……197
7.1.1 queue类的方法接口 ……197
7.1.2 使用queue类 ……200
7.1.3 容器配接器 ……201
7.1.4 一个接近的设计 ……202
7.2 计算机仿真 ……204
7.3 队列应用：洗车仿真 ……205
7.3.1 程序设计 ……207
7.3.2 CarWash类的实现 ……208
7.3.3 CarWash方法的分析 ……212
7.3.4 随机化到达时间 ……212
实验18：随机化到达时间 ……214
7.4 堆栈 ……214
7.4.1 Stack类的方法接口 ……214
7.4.2 使用stack类 ……215
7.4.3 stack类是一个容器配接器 ……216
7.5 堆栈应用1：递归是如何实现的 ……216
7.6 堆栈应用2：将中缀转换成后缀 ……222
7.6.1 后缀表示法 ……224
7.6.2 转换矩阵 ……226
7.6.3 记号 ……227
实验19：将中缀转化成后缀 ……228
7.6.4 前缀表示法 ……228
总结 ……230
习题 ……231
编程项目7.1：扩展洗车仿真 ……232
编程项目7.2：求一个条件的值 ……233
编程项目7.3：一个迭代的迷宫搜索 ……237

编程项目7.4: queue类的另一个设计 ……237
第8章 二叉树和折半查找树 ……239
8.1 定义和属性 ……239
8.1.1 二叉树定理 ……245
8.1.2 外部路径长度 ……247
8.1.3 二叉树的遍历 ……248
8.2 折半查找树 ……253
8.2.1 BinSearchTree类 ……254
8.2.2 BinSearchTree类的Iterator类 ……255
8.2.3 BinSearchTree类的字段和实现 ……257
8.2.4 递归方法 ……261
8.2.5 BinSearchTree迭代器 ……269
实验20: BinSearchTree的平均高度 ……270
总结 ……270
习题 ……271
编程项目8.1: BinSearchTree类的另一种实现 ……274
第9章 AVL树 ……277
9.1 平衡的折半查找树 ……277
9.2 旋转 ……277
9.3 AVL树 ……281
9.3.1 AVL树的高度 ……282
9.3.2 函数对象 ……284
实验21: 更多的函数对象的细节 ……286
9.3.3 AVLTree类 ……286
9.3.4 fixAfterInsertion方法 ……289
9.3.5 insert方法的正确性 ……297
9.4 AVL树的应用: 一个简单的拼写检查器 ……299
总结 ……302
习题 ……302
编程项目9.1: AVLTree类的erase方法 ……305
编程项目9.2: 改进的SpellChecker项目 ……305
第10章 红黑树 ……307
10.1 红黑树 ……307
10.1.1 红黑树的高度 ……310
10.1.2 惠普的rb_tree类 ……313
10.1.3 rb_tree类中的insert方法 ……315
实验22: 使用全部三种情况的红黑树插入 ……320
10.1.4 erase方法 ……320
实验23: erase的调用，其中应用了全部四种情况 ……331
10.2 标准模板库的关联容器 ……331
10.3 集合应用: 再次讨论拼写检查器 ……334
10.3.1 multiset类 ……335
实验24: 更多与set和multiset类相关的知识 ……336
10.3.2 map类 ……336
10.3.3 multimap类 ……339
实验25: 更多与map和multimap类相关的知识 ……340
总结 ……340
习题 ……340
编程项目10.1: 一个简单的辞典 ……343
编程项目10.2: 创建一个词汇索引 ……343
第11章 优先队列和堆 ……347
11.1 介绍 ……347
11.1.1 priority_queue类 ……348
11.1.2 priority_queue类的字段和实现 ……350
11.1.3 堆 ……351
实验26: 优先队列中的公平性 ……359
11.1.4 priority_queue类的另一种设计及实现 ……359
11.2 优先队列的应用: 霍夫曼编码 ……360
11.2.1 huffman类的设计 ……364
11.2.2 huffman类的实现 ……366
总结 ……371
习题 ……372
编程项目11.1: 解码一个消息 ……374
第12章 排序 ……377
12.1 介绍 ……377
12.2 排序能有多快 ……380
12.3 快速排序 ……382
12.3.1 树排序 ……382
12.3.2 堆排序 ……383
12.3.3 归并排序 ……385
12.3.4 快速排序 ……390
12.3.5 分治法算法 ……395
实验27: 排序算法的运行时间 ……396
总结 ……396
习题 ……397
编程项目12.1: 排序一个文件 ……402

第13章 查找和散列类 ……405
13.1 分析查找的框架 ……405
13.2 查找方式复习 ……405
13.2.1 顺序查找 ……405
13.2.2 折半查找 ……406
13.2.3 红黑树查找 ……408
13.3 hash_map类 ……408
13.3.1 hash_map类中的字段 ……409
13.3.2 散列 ……409
13.3.3 链式 ……412
13.3.4 iterator类的字段和实现 ……419
13.3.5 hash_map类的实现 ……420
13.3.6 链式散列分析 ……423
13.3.7 value_type类 ……425
13.3.8 应用 ……425
实验28: hash_map计时 ……427
13.4 hash_set类 ……427
13.5 开放地址散列 ……427
13.5.1 erase方法 ……430
13.5.2 主聚类 ……434
13.5.3 双散列 ……435
13.5.4 开放地址散列分析 ……439
总结 ……441
习题 ……442
编程项目13.1: 使用链式和双散列构造一个符号表的运行时间比较 ……444
第14章 图、树和网络 ……445
14.1 无向图 ……445
14.2 有向图 ……447
14.3 树 ……448
14.4 网络 ……450
14.5 图算法 ……451
14.5.1 迭代器 ……451
14.5.2 连通性 ……457
14.5.3 产生最小生成树 ……458
14.5.4 寻找网络中的最短路径 ……462
14.6 开发一个网络类 ……465
14.7 network类 ……465
14.7.1 network类中的字段 ……467
14.7.2 network类的实现 ……469
14.7.3 与边相关的方法的实现 ……470
14.7.4 全局方法的实现 ……472
14.7.5 get_minimum_spanning_tree方法 ……475
14.7.6 get_shortest_path方法 ……476
14.7.7 网络方法的时间花费估算 ……478
实验29: 货郎担问题 ……479
14.7.8 network类的另一种设计和实现 ……479
14.8 回溯通过网络 ……481
总结 ……483
习题 ……484
编程项目14.1: 完成邻接矩阵的实现 ……486
编程项目14.2: 回溯通过一个网络 ……486
附录1 数学背景 ……489
附录2 string类 ……501
附录3 多态性 ……511
参考文献 ……515
索引 ……517

第1章 C++中的类

这是一本关于编程的书：重点是对数据结构及算法的理解和使用。C++标准模板库中聚集了大量的数据结构和算法。第2~12章将侧重于解释什么是库以及如何使用库来简化编程。要想使用这些信息，必须先熟悉本章提到的类的情况。有些类是读者已了解的；有些则可能是完全陌生的。所有这些，不论是对库自身还是对如何在编程中使用库而言，都是必须的。

目标

1) 理解类、对象和消息的基本原理。
2) 比较程序开发人员和用户对类的看法。
3) 能够灵活运用数据抽象的原理，开放—封闭原理，以及子类替换规则。

1.1 类

类将变量和操作这些变量的函数连接了起来。

类是一个用户自定义类型，它由一些变量（称作**字段**⊖）和操作这些字段的函数（称作**方法**）组成。类将静态组件（字段）和动态组件（方法）封装在一个实体中。这个封装技术改善了程序的模块性：通过隔离类和程序的其他部分，可以更容易理解和修改程序。

假设现在想解决一些问题，需要使用日历上的数据。如果没有现成的可用类，那么就创建一个类Date。这个Date类将由一个或更多的字段组成，用来存放日期，还需要方法操作这些字段。开始时不必过多考虑如何选择表示日期的字段，也不要考虑操作这些字段的方法。因为我们的目的是使用Date，所以首先需要确定Date类的**职责**。也就是，这个类希望提供给用户什么？假设职责是：

1) 给定年、月、日，构造一个日期。
2) 从日期中读出年、月、日。
3) 判断给定的一个日期是否有效。
4) 返回给定日期之后的下一个日期。
5) 返回给定日期的前一个日期。
6) 返回给定日期是星期几（如星期二）。
7) 判断给定日期是否在某些日期之前。
8) 以年、月、日的形式输出一个日期，比如：2004，5，10。

1.1.1 方法接口

方法接口提供给用户有关方法的全部信息。

⊖ 此边栏的页码为该书原书页码，与书末索引中的页码相呼应。

⊖ 在其他参考书中，“字段”也可称为数据成员、成员变量、实例变量或属性；“方法”也称为成员函数、服务或操作。

类的职责都被精练在**方法接口**中：用户调用方法时需要的明确信息。每个方法接口应包含三个部分：一个前置条件，一个后置条件，一个以分号结束的方法头。**前置条件**是在方法运行前假设的程序状态，**后置条件**是在保证前置条件为真时方法运行后的程序状态。

例如，下面给出了isValid方法的方法接口：

```
//后置条件：如果这个Date是合法的就返回真：year应当是1800到2200之间的整数；
2 //          month必须是1到12之间的整数；day必须是1到给定年给定月的天数之
//          间的整数；以上三条必须同时成立。否则，将返回假。
bool isValid();
```

这里没有给出前置条件，因为调用这个方法前程序状态无须特别限定。从技术性上而言，前置条件就是**真（ture）**。但是在书写时可以省略。每个方法都应完成一定的功能，因此总是需要明确地给出后置条件。为了阐明后置条件里的“这个Date”短语，需要先定义术语“对象”，它是面向对象编程技术的基础。

1.1.2 对象

给定一个类，**对象**（有时称作类的一个**实例**）是一个变量，它拥有这个类的字段并可以调用这个类的方法。例如，如果定义：

```
Date myDate;
```

那么myDate是一个Date类型的对象。如果在程序后面编写：

```
if (myDate.isValid( ))
    cout << "The date is valid.";
else
    cout << "The date is not valid.";
```

这表示对象myDate调用它的isValid方法。因此，myDate也被看作是一种**调用对象**——调用方法的对象。

在isValid方法的后置条件里，短语“这个Date”指的是调用对象。因此根据调用对象的当前值返回一个**布尔**（bool）值。例如，假若对象myDate的字段的值分别是5，17，2003。那么调用

```
myDate.isValid()
```

将返回**真**。但是如果myDate的字段是4，31，2003，那么调用

```
myDate.isValid()
```

将返回**假**（false）。

通常情况下，方法调用句的组成是一个对象、后跟一个点，然后是方法标识符，最后是圆括号括起的一串变元。在面向对象中，**消息**是对象对一个方法的调用。例如，myDate.next()返回调用对象日期的下一个日期。在这个消息里，对象myDate调用了Date类的next方法。术语“消息”是指程序的某一部分和另外部分的通信。例如，消息thisDate.next()
3 也可能是从除了Date类之外的其他类的方法发送的。

精练Date方法的职责后可以得到下面的方法接口：

```
//后置条件：这个Date由month、day和year构成
Date (int month, int day, int year);
```

注意 1.1.4节中将给出这个方法的说明和一个示范消息。

```
//后置条件: 用读入的month、day和year设置这个Date的日期。
void readinto();
```

示例 假设有下列程序:

```
Date lastDate;
lastDate.readinto();
```

如果输入行包含

```
12  1  2003
```

那么lastDate的字段将分别描述日期2003年12月1日。

```
//后置条件: 如果这个Date是合法的就返回真: year应当是1800到2200之间的
//整数; month必须是1到12之间的整数; day必须是1到给定年给定月的天数之
//间的整数; 以上三条必须同时成立。否则, 将返回假。
bool isValid();
```

示例 假设currentDate是类Date的一个对象。如果这个对象的字段的值分别是2，29，2004，那么currentDate.isValid()将返回真。但是如果currentDate的字段值分别是2，29，2005，那么currentDate.isValid()将返回假。

```
//前置条件: 这个Date是有效的。
//后置条件: 返回这个Date之后的Date。
Date next();
```

4

示例 假设date是Date类的一个对象。如果对象的字段值分别是2000,2,29，那么消息date.next()将返回值2000,3,1。如果date中是一个无效值，然后调用date.next()会怎样呢？因为无效日期不符合前置条件，所以无法定义结果。也就是，可能不返回日期，也可能返回一个无意义的日期，或是程序错误，等等。类的用户有义务保证在调用方法前前置条件成立。例如，用户可以按如下方式进行:

```
if (date.isValid( ))
      . . . date.next( ) . . .
else
      cout << "Invalid date" << endl;
```

```
//前置条件: 这个Date是有效的。
//后置条件: 返回这个Date之前的Date。
Date previous();
```

```
//前置条件: 这个Date是有效的。
//后置条件: 返回这个Date是星期几——如“星期天”、“星期一”等。
String dayOfWeek();
```

示例 假设meetingDate是Date类的一个对象，而且如果对象的字段值代表日期2003年6月12日，那么消息

meetingDate.dayOfWeek()

将返回“星期四”。

注意 string类是ANSI标准C++的一部分，因此在正文和实验中都可以自如地使用。附录2详细地解释了这个类；如果读者不熟悉这个非常重要的类，请现在阅读附录。

```
//前置条件：这个Date和otherDate是有效的。
//后置条件：如果Date在otherDate之前就返回真。否则，返回假。
bool isPriorTo (Date otherDate);
```

示例 假设currentDate是Date类的一个对象，它的字段值代表2003年3月27日，startDate也是Date类的一个对象，它的字段值代表2004年1月1日，那么

currentDate.isPriorTo (startDate)

将返回真。

```
//前置条件：这个Date是有效的。
//后置条件：将Date按照月、日、年的形式输出。
void print();
```

示例 假设newtDate是Date类的一个对象，它的字段值代表2003年1月28日，运行
newDate.print();

则输出

January 28, 2003

类的这个观点是从用户的角度出发，聚焦于类用户需要的信息上。1.1.3节将着眼于开发者的角度，并比较这两种方式。

1.1.3 数据抽象

数据抽象是将类为用户提供的东西和类的定义分离开。

迄今为止讨论的都是方法接口，也就是类为编程者提供了什么，而不是类的字段和方法定义——即类是如何定义的。这种分离称作**数据抽象**，分离中的“什么”和“如何”是面向对象编程的基本特点。使用Date类的编程者不会去关心数据如何表示或者方法如何定义。字段，也就是日期的表示形式，可能采用下面的一个形式，也可能采用完全不同的形式：

int字段描述month、day和year：比如使用值“2”代表month，“28”代表day，“2002”代表year

一个七位或八位的**long**字段描述theDate：比如使用值“1042005”（代表2005年1月4日）和“10042005”（代表2005年10月4日）

用一个长度为8的字符串字段描述theDate：比如使用值“02282002”

用一个长度为10的字符串字段描述theDate：比如使用值“02-28-2002”

如果用长度为6的字符串字段表示theDate——比如“022802”，将会有什么错误？

同样，某些方法可能需要选择方法的定义。例如，isValid()方法的定义可能使用一个**switch**语句或是一系列的**else-if**语句：

```
if (month == 4 || month == 6 || month == 9 || month == 11)
    return 30; // 30 days hath September, April, June and November
else if (month == 1 || month == 3 || month == 5 || month == 7 ||
    month == 8 || month == 10 || month == 12)
    return 31; // January, March, May, July, August, October, December
else if (year % 4 != 0 || (year % 100 == 0 && year % 400 != 0))
    return  28; // one year is slightly less than 365.25 days
return 29;
```

一个完整的面向Date类的项目（包括Date类的具体实现），可以从本书网站的源代码链接上找到。

字段和方法定义的实现细节让人在开发一个使用Date类的类时分心。可能其他一些人已经完成了Date类的定义，那么只需使用那个Date类即可，而不必再做额外的工作。但即便必须独立定义Date类，也可以把它推迟到使用Date类的类开发工作完成之后再进行。通过使用Date的方法接口，可以增强这些类的独立性：只要不修改Date的方法接口，那么对Date类的任何修改都不会影响这些类的有效性。

当编程者关注类提供的服务而不是类的实现细节时，就已经应用了数据抽象原理：

数据抽象原理

用户的代码不应访问他所使用的类的实现细节。

数据抽象原理的一个要点是如果类A使用类B，那么类A的方法不应访问类B的字段。实际上，类B的字段只应被类B的方法访问。例如，假设下面的定义是在Date类之外进行的：

```
Date currentDate;
```

那么类似下列形式的表达式：

```
currentDate.month
```

是对数据抽象原理的违法应用，因为Date类是否有month字段是一个实现细节问题。即使Date类有一个month字段，开发者可以自由地修改Date类，而不能改变方法接口。例如，开发者可以将Date类修改为只包含一个字段：

```
string theDate;
```

数据抽象原理有利于类的用户，因为这使他们摆脱了对类的实现细节的依赖性。当然，这么说是基于类的方法接口已经提供了用户所需要的所有类的信息。一个类的开发者应当创建功能足够强大的方法，从而使用户不需要借助任何实现细节。这个功能应在方法接口中给出清晰、明确的说明。

方法的前置条件和后置条件是在开发者和用户之间隐式契约的一部分。契约的规则如下：

如果方法的用户在调用该方法前保证前置条件为真，那么开发者将保证在方法执行完毕后后置条件为真。

综上所述，从开发者角度出发的有关类的讨论，类是由字段和操作这些字段的方法定义组成的。从用户角度来说可以抽象为：类是由方法接口组成的。

基本上，标准模板库是被完全测试过的类的集合，这些类在多种应用中都是有实用价值的。应用程序和大部分指定的项目将使用标准模板库，因此它们只依赖于方法接口，而不依

7

赖库中方法的定义。

1.1.4 构造器

C++允许在对象的定义中包含对象的初始化，这减轻了编程者必须初始化对象的负担。这个定义加上初始化的机制就是**构造器**：它是类中的一个特殊方法，和类的名字相同。每次定义类的对象，都将自动调用类的构造器。例如，可以用如下方式构造一个Date对象：

```
Date startDate (7,1,2003);
```

这个语法有点不寻常：类标识符后面跟着对象标识符，再后面是一对圆括号括起的变元列表。对象startDate中的字段现在表示日期“2003年7月1日”。这个构造器的方法定义依赖于Date类中的字段。例如，如果Date类有**int**型字段month、day和year，那么可以进行下面的定义：

```
Date (int monthIn, int dayIn, int yearIn)
{
    month = monthIn;
    day = dayIn;
    year = yearIn;
}//3个参数的构造器
```

8

另外，如果Date类只有一个字段：

```
long theDate;
```

那么构造器的定义就只有一条（晦涩的）赋值语句：

```
Date (int monthIn, int dayIn, int yearIn)
{
    theDate= ( (monthIn*100)+dayIn)*10000+yearIn;
}//3个参数的构造器
```

该讨论产生了一个问题：如果不用这个构造器定义一个对象会发生什么？那么Date类也必须定义一个0-参数构造器。例如，假设有程序段：

```
Date today; // 注意，在today后面没有圆括号
today.readInto();
```

在这个示例中，调用了一个0-参数构造器——即使today的定义后面没有圆括号。例如，如果Date类有**int**型字段month、day和year，可以做如下定义：

```
// 后置条件：这个Date对象被初始化为1800年1月1日
Date() // 注意，这里是需要圆括号的
{
    month=1;
    day=1;
    year=1800;
}//0-参数构造器
```

也就是说，这个构造器将把today的日期初始化为1800年1月1日。这个初始化值将在调用readInto方法之后被覆盖。

缺省构造器就是不带参数的构造器。

如果一个类没有显式地定义任何构造器怎么办？那么将由编译器自动生成一个0-参数构造器。因此，0-参数构造器也称为**缺省构造器**。但是如果类定义了任意一个构造器，编译器都不再生成缺省构造器。这样，除了一个构造器是缺省构造器之外，类的每个实例都必须用至少有一个参数的构造器进行定义。从安全的角度出发，当定义包含任何构造器的类时，应当显式地定义一个缺省构造器。1.1.9节阐述了使用这个预防措施的另一个原因。

一个构造器，不管有没有参数，都不会自动初始化类的字段，因此必须显式地写出代码初始化字段，例如**int**型字段。例外的情况是**对象字段**，即字段的类型自身就是一个类的时候。对象字段将根据那个类相应的构造器进行初始化。例如，假设一个Calendar类有两个字段：

```
Date startDate,
     endDate (12,31,2200);
```

9

如果声明

```
Calendar calendar;
```

那么将调用Calendar类的缺省构造器。在构造器运行之初，Calendar的对象calendar的startDate字段将被初始化为日期1800年1月1日，而endDate字段将被初始化为日期2200年12月31日。

1.1.5节使用其他范例继续讨论方法接口、方法定义和构造器。

1.1.5 一个Employee类

在这个范例中，创建一个名为Employee的类，用来表示一个公司的雇员。每个雇员的信息由雇员姓名和薪水总额组成。Employee类的职责是：

1) 将雇员的姓名初始化为空字符串，薪水总额初始化为0.00。
2) 读入一个雇员的姓名和薪水总额。
3) 判断是否到达了输入结束标记（姓名是“*”，薪水总额是-1.00）。
4) 判断一个雇员的薪水总额是否大于其他雇员的薪水总额。
5) 获取一个雇员的姓名和薪水总额的拷贝。
6) 输出一个雇员的姓名和薪水总额。

根据这些职责，开发方法接口：

```
//后置条件：这个雇员的姓名被设置成空字符串，薪水总额被设置成0.00。
Employee();

//后置条件：读入雇员的姓名和薪水总额。
void readInto();

//后置条件：如果Employee包含结束标记就返回真。否则，返回假。
bool isSentinel() const;

//后置条件：如果Employee的薪水总额高于otherEmployee的薪水总额就返回
//真。否则，返回假。
bool makesMoreThan (const Employee&otherEmployee) const;

//后置条件：这个Employee包含了对otherEmployee的拷贝。
```

10 **void** getCopyOf (**const** Employee&otherEmployee);

注意 返回的是otherEmployee的另一个副本。例如，假设输入包含

Simth, John 100000.00
Siddiqi, Amena 120000.00

然后运行下面的语句:

```
Employee oldEmployee,
         newEmployee;

newEmployee.readInto( );
oldEmployee.getCopyOf (newEmployee);
newEmployee.readInto( );
```

然后在第一次调用readInto之后，得到:

newEmployee

Smith,John	100000.00

oldEmployee

	0.00

调用getCopyOf方法之后，得到:

newEmployee

Smith,John	100000.00

oldEmployee

Smith,John	100000.00

最后，readInto的第二次调用给出:

newEmployee

Siddiqi,Amena	120000.00

oldEmployee

Smith,John	100000.00

```
//后置条件: 输出这个Employee的姓名和薪水总额。
```

11 **void** printOut() **const**;

Employee类的方法接口包含了使用该类的编程者所需要的所有信息。另一方面，类的开发者必须决定需要哪些字段，然后定义方法。例如，开发者可能决定需要两个字段: 雇员的name（string型）和grossPay（**double**型)。方法头和字段（以及字段的类型）组成了一个**类的声明**。

包含类的声明的文件称作**头文件**。下面是头文件employee1.h，它包含了Employee类的声明:

```
#ifndef EMPLOYEE
#define EMPLOYEE
```

```
#include <string>

using namespace std;

class Employee
{
    public:
        //后置条件: 这个雇员的姓名被设置成空字符串，薪水总额被设置成0.00。
        Employee();

        //后置条件: 读入雇员的姓名和薪水总额。
        void readInto();

        //后置条件: 如果Employee包含结束标记就返回真。否则，返回假。
        bool isSentinel() const;

        //后置条件: 输出这个Employee的姓名和薪水总额。
        void printOut() const;

        //后置条件: 这个Employee包含了对otherEmployee的拷贝。
        void getCopyOf (const Employee&otherEmployee);

        //后置条件: 如果Employee的薪水总额高于otherEmployee的薪水总额就返回
        //真。否则，返回假。
        bool makesMoreThan (const Employee&otherEmployee) const;

    private:
        string name;
        double grossPay;

        const static string EMPTY_STRING;
        const static string NAME_SENTINEL;
        const static double GROSS_PAY_SENTINEL;
}; // Employee
#endif
```

12

现在仔细地看一下这个文件。每个以符号“#”打头的行代表了一条**编译器指令**：一条编译器消息。如前两行

```
#ifndef EMPLOYEE
#define EMPLOYEE
```

通知编译器，如果标识符EMPLOYEE还没有在这个项目中定义过（**if not defined**），那么就定义EMPLOYEE。使用这些指令的原因是为了避免重复声明一个标识符——重声明标识符是错误的。因此，如果这个文件——特别是标识符EMPLOYEE——还没有被编译器遇到，那么将编译这个文件。否则，整个文件中所有通向

```
#endif
```

的路径都将被编译器忽略。因此即使项目的多个文件里包含

```
#include "employee1.h"
```

也不会发生重复声明的情况。下一条编译指令，

```
#include <string>
```

通知编译器在这个对象中包含了标准模板库的string类（附录2详尽地论述了这个类）。现在需要了解的是我们可以定义一个string对象，然后使用这个对象进行读、写、赋值和比较。例如，

```
string s;
cin >> s;
cout << s;
s = "yes";
if (s < "wow")
while (s != "end")
if (s[0]>'m')//如果s的0位置的字符>'m'
...
```

在讨论Employee类之前尚需讨论它的多个特性。标准模板库里的标识符——如string——
13 被归类到称作std的名字空间中。**使用指令**由关键字**using**和**namespace**以及一个给定名字空间组成。特别是，

```
using namespace std;
```

通知编译器正在使用std名字空间，因此即使在这个项目的另一个文件中定义了它自己的string标识符，string仍然代表的是标准模板库中定义的类。

现在可以讨论Employee类了。Employee被声明成一个类——**class**是一个关键字，它有六个方法、两个字段和三个常量标识符。注意一个**class**（或**struct**）的关闭声明之后必须跟一个分号。

方法和字段（以及其他标识符）的区别是圆括号，因此即使方法中没有形参也不能省略圆括号。

类的声明还有一些显著特点:

• 构造器

```
Employee();
```

正如1.1.4节所述，构造器帮助用户摆脱了进行对象的显式初始化的负担。初始化工作在定义对象时自动完成，如

```
Employee bestPaid;
```

构造器和类名字相同，并且没有返回类型。一个没有参数的构造器称作缺省构造器，因为如果不定义任何构造器的话，C++编译器将自动提供一个0-参数构造器（这只是为了使类似于bestPaid这样的定义合法化）。

常量引用参数既保证了效率，又保证了安全性。

• 在getCopyOf方法中，otherEmployee是一个**常量引用参数**。作为一个由&说明的引用参数，当调用方法时只传送相应变元的地址，因此对这个变元不做任何复制工作。这既节省了时间又节省了空间。同样，由于有关键字**const**，所以在getCopyOf方法中修改otherEmployee是非法的。这保证了安全性——相应的变元将不会被修改。

• 关键字**const**还出现在isSentinel、printOut和makesMoreThan的方法头的最后，它保证

了这些方法不会修改调用对象。在makesMoreThan方法中，参数otherEmployee又是一个常量引用参数。

- 注意在Employee类的声明里，关键字**public**后面跟着一个冒号。在C++中，一个类的不同成员（即字段、常量和方法）可以拥有不同的保护级别，它指明了什么样的代码可以访问那些特定的成员。另一个保护级是用**private**：指示的（不要忘记冒号！）。这个保护级只允许类的方法访问成员。如果不指定，那么缺省保护级就是**private**。 14
- name和grossPay字段都是**private**，因此在Employee类的方法之外不能访问这些字段。
- EMPTY_STRING、NAME_SENTINEL和GROSS_PAY_SENTINEL是**类常量**标识符：它们应用在Employee类的所有实例中，而不只是某个单独的实例上。这说明应该只有一个GROSS_PAY_SENTINEL的拷贝，而不是Employee类的每个实例都有一个拷贝。为了说明这是“类相关”而不是“对象相关”的，关键字**static**也是声明的一部分。常量的值在类的定义里提供，这在1.1.6节中有进一步的叙述。

1.1.6 Employee类的定义

包含类的定义的文件称作**源文件**。要定义类，必须提供它的方法和类常量的定义。方法的定义是一个完整的函数：头和体。下面是源文件employee1.cpp，它包含了Employee的类定义：

```
#include <iostream>
#include <iomanip>//声明输出格式化的对象
#include "employee1.h"//声明Employee类
Employee::Employee( )
{
      name = EMPTY_STRING;
      grossPay = 0.00;
} // default constructor
void Employee::readInto( )
{
      const string NAME_AND_PAY_PROMPT =
            "Please enter a name and gross pay, to quit, enter ";
      cout << NAME_AND_PAY_PROMPT << NAME_SENTINEL << " "
           < GROSS_PAY_SENTINEL;
      cin > > name > > grossPay;
} // readInto

bool Employee::isSentinel( ) const
{
      if (name == NAME_SENTINEL && grossPay ==
                                                          15
                  GROSS_PAY_SENTINEL)
            return true;
      return false;
} // isSentinel
```

```
void Employee::printOut( ) const
{
    cout << name << " $" << setiosflags (ios::fixed) << setprecision (2)
        << grossPay << endl;
} // printOut

void Employee::getCopyOf (const Employee& otherEmployee)
{
    name = otherEmployee.name;
    grossPay = otherEmployee.grossPay;
} // getCopyOf

bool Employee::makesMoreThan (const Employee& otherEmployee) const
{
    return grossPay > otherEmployee.grossPay;
} // makesMoreThan

const string Employee::EMPTY_STRING = ""; // 这里没有使用static
const string Employee::NAME_SENTINEL = "*";
const double Employee::GROSS_PAY_SENTINEL = -1.0;
```

public保护级允许在代码的任意部分进行访问。一个类的方法通常都使用**public**，因为这些方法往往是在使用类时我们希望其他代码部分调用的东西。对Employee对象而言，任意代码段都可以调用构造器和其他方法：isSentinel、readInto、writeOut、getCopyOf和makesMoreThan。

为了将类的方法和其他函数区分开，需要一个限定词。限定词由类的名字及随后的两个（不是一个）冒号组成。因此，例如readInto方法的定义开始是

```
void Employee::readInto()
```

这个头指出readInto是Employee类的一个方法。这两个连续的冒号被称作**作用域解析运算符**：左边的操作数是类，右边的操作数是这个类的成员。作用域解析运算符消除了readInto标识符中所有可能的不确定性，即使一个非成员函数也叫做readInto，或者文件里的另一个类也有一个readInto标识符。

注意，一个类常量标识符的定义提供了它自身的值。作用域解析运算符必须被包含进去，但关键字static被省略了。

16

在一个方法定义中，调用对象的字段和方法不用类标识符也可以被访问。因此，如果在方法定义中一个字段总是以它自身出现，那么这个字段被假定成调用这个方法的对象的一个字段。当然，没有调用对象时类常量也会出现，但那是因为它们是和类关联在一起，而不是和类的一个实例相关联。

Employee类的用户对这些实现细节不感兴趣。例如，假设要寻找公司中薪水最高的雇员。在这个应用里，使用Employee来开发一个Company类。Company类的职责是构造一个公司，并寻找输出公司中薪水最高的雇员，也就是薪水总额最大的雇员的姓名和薪水总额。现在将这些职责精炼为一个缺省构造器——findBestPaid和printBestPaid方法的方法接口：

```
//后置条件：这个Company被初始化。
Company();
```

```
//后置条件：找出这个Company中薪水最高的雇员。
void findBestPaid();

//后置条件：输出这个Company的薪水最高的雇员。
void printBestPaid() const;
```

这些Company方法的细节以及开始的main函数见实验1。

实验1：Company项目　　（所有实验都是可选的）

正如1.1.3节中发现的一样，应当尽可能地使用已有的类。如果一个类拥有这个应用所需的最多但不是全部的方法怎么办？简单的方法是抛弃这个类再开发一个自己的类，但是这会浪费时间，降低效率。另一个选择是拷贝类中需要的部分并将这些部分合并到我们开发的一个新的类中。这种选择存在的危险是这些部分可能是不正确的或者是效率低的。即使源类的开发者替换了这些不正确的或效率低的代码，我们的类仍然是错误的或效率低的。比较好的选择是使用1.1.7节中讲述的继承。

1.1.7 继承

编程者应努力编写可重用软件组件。例如，定义一个函数计算10个数值的平均数，不如定义一个函数计算任意个数值的平均数。通过编写可重用代码，不仅节省了时间，而且避免了错误地修改已有代码的潜在危险。

17

继承是定义一个新的类，且该类包含已有类的全部字段以及部分或全部方法的能力。

可重用性可以应用在类上，一种方式是通过类的一个特殊、强大的特性——继承。继承是定义一个新的类，且该类包含已有类的全部字段以及部分或全部方法的能力。早期已有的类称作**超类**、**基类**或者**祖先类**。新的类（可能声明了新的字段和方法），称作**子类**、**派生类**或者**子孙类**。一个子类也可以通过给出不同于超类的方法定义覆盖超类中已有的方法。

为了解释继承是如何发挥作用的，还是从类Employee开始。假设有若干应用使用了Employee。一个新的应用涉及到查找计时工资最多的雇员。为了实现这个应用，输入应当由雇员姓名、工作的小时数（**int**型）和小时工资（**double**型）组成。薪水总额是“小时数×小时工资”。

现在可以改变Employee，给它添加hoursWorked和payRate字段并修改readInto和isSentinel方法。但是，为了一个新的应用修改在现有应用中已成功使用的类是非常冒险的。这背后的概念是众所周知的开放—封闭原理：

开放—封闭原理

每个类都应当被打开——也就是能够通过继承扩展；并封闭——也就是在现有应用中保持稳定性。

具体地说，Employee类不应该为了一个新的应用而被修改。不用重写Employee，可以开发HourlyEmployee——Employee的一个子类。HourlyEmployee的每个对象将保存Employee的信息——姓名和薪水总额，以及工作的小时数和小时工资。makesMoreThan、getCopyOf和

printOut方法不必修改，因为HourlyEmployee类的对象可以像Employee的对象一样调用那些方法。毕竟，一个计时雇员还是一个雇员！下面是HourlyEmployee版本的readInto和isSentinel方法的接口。

```
//后置条件：读入这个HourlyEmployee的姓名、工作的小时数和小时工资，
//并计算出薪水总额。
void readInto();

//后置条件：如果这个HourlyEmployee包含了结束标记就返回真。否则，
//返回假。
bool isSentinel() const;
```

在开始HourlyEmployee类的完整的声明和定义之前，需要先了解一下子类方法是如何访问超类方法的。

1.1.8 受保护的访问

前面提到过类声明中的缺省保护级是**private**，而且可以访问这些**private**字段或者方法的只有类方法自身的代码。具体来说，这暗示了子类是不能访问这些**private**字段或方法的。可以令这些字段或者方法是**public**，但是这样一来任何代码——甚至其他类的代码——都可以访问到这些部分，而这可能是我们所不希望的。为了处理子类的代码访问情况，可以使用**protected**保护级。

18

下面说明了**protected**是如何工作的。假设x是一个成员——也就是说是类A的一个字段、常量或方法。如果在x的声明前写上

```
protected:
```

那么A的任一个方法都可以访问x，而且A的任意子类的任一个方法也都能访问到x，但是其他的代码不能访问x。

由于希望在子类中提高可重用性，所以通常将类的字段和类常量设置成**protected**而不是**private**访问。因此，我们修改了Employee的类声明，将字段和类常量设置为**protected**而不是**private**状态：

```
#ifndef EMPLOYEE
#define EMPLOYEE

#include <string>

using namespace std;

class Employee
{
    public:

    //后置条件：这个Employee的姓名被设置成空字符串，薪水总额被设置成0.00。
    Employee();

    //后置条件：如果这个Employee包含结束标记就返回真。否则，返回假。
    bool isSentinel() const;
```

```
        //后置条件：读入这个Employee的姓名和薪水总额。
        void readInto();

        //后置条件：输出这个Employee的姓名和薪水总额。
        void printOut() const;

        //后置条件：这个Employee包含了对otherEmployee的拷贝。
        void getCopyOf (const Employee&otherEmployee);

        //后置条件：如果Employee的薪水总额高于otherEmployee的薪水总额就返回
        //          真。否则，返回假。
        bool makesMoreThan (const Employee&otherEmployee) const;

    protected:

        string name;
        double grossPay;

        const static string EMPTY_STRING;
        const static string NAME_SENTINEL;
        const static double GROSS_PAY_SENTINEL;

    }; // Employee
    #endif
```

19

顺带说一下，对Employee类的这个改变不止没有违背开放—封闭原理，而且正是遵从了这个原理，因此Employee类可以合法地派生子类。

一个类的**protected**成员只能被该类和它的子类的方法访问。

综上所述，限制最严格的保护级是**private:**，它只能被类的方法访问。稍微弱一点的保护级是**protected:**，它只能被类和它的子类的方法访问。限制最松的保护级是**public:**，它可以被任何代码访问。

现在再转回去考虑计时雇员的开发问题。

1.1.9 HourlyEmployee类

通常情况下，子类的声明都以如下形式开始：

class <subclass identifier>: **public** <superclass identifier>

这里用到了保留字**public**，这说明每个超类成员的保护级决定了它在子类方法中的可访问性。也就是说，超类的**public**和**protected**成员可以被任意子类方法访问，超类的**private**成员在子类方法中是不能被访问的。

除了这个特点，HourlyEmployee类声明的其他部分和正常类声明的形式是相同的，并且只是声明或定义了子类中新的字段、方法或覆盖的方法。为了HourlyEmployee的潜在子类，将字段和类常量都设置成**protected**（而不是**private**）状态：

```
#ifndef HOURLY_EMPLOYEE
#define HOURLY_EMPLOYEE

#include "employee1.h"
```

```
class HourlyEmployee : public Employee
{
    public:
20      //后置条件: 这个HourlyEmployee被初始化。
        HourlyEmployee();

        //后置条件: 读入这个HourlyEmployee的姓名、工作的小时数和小时工资。
        void readInto();

        //后置条件: 如果读入了结束标记就返回真。否则，返回假。
        bool isSentinel() const;

        protected:
            int hoursWorked;
            double payRate;

            const static int HOURS_WORKED_SENTINEL;
            const static double PAY_RATE_SENTINEL;
}; // HourlyEmployee

#endif
```

Employee类的name和grossPay字段会怎么样呢？它们将被Employee类的缺省构造器初始化。无论何时调用任何子类的构造器，都将自动调用超类的缺省构造器[⊖]。这保证了至少所有超类对象的字段将被正确的初始化。

下面是源文件hourlyEmployee2.cpp:

```
#include <iostream>

#include "hourlyEmployee2.h"

HourlyEmployee::HourlyEmployee( ) { }

void HourlyEmployee::readInto( )
{
    const string NAME_HOURS_RATE_PROMPT =
        "Please enter a name, hours worked and pay rate. The sentinels
            are ";

    cout << NAME_HOURS_RATE_PROMPT << NAME_SENTINEL <<
        " "
      << HOURS_WORKED_SENTINEL << " " <<
            PAY_RATE_SENTINEL << ": ";

    cin > > name > > hoursWorked > > payRate;

21  grossPay = hoursWorked * payRate;

} // readInto

bool HourlyEmployee::isSentinel( ) const
{
    if (name == NAME_SENTINEL
```

⊖ 如果超类至少定义了一个构造器但没有缺省构造器，将生成一个编译时错误信息。

```
                && hoursWorked == HOURS_WORKED_SENTINEL
                && payRate == PAY_RATE_SENTINEL)
            return true;
        return false;
    } // isSentinel

    const int HourlyEmployee::HOURS_WORKED_SENTINEL = -1;
    const double HourlyEmployee::PAY_RATE_SENTINEL = -1.00;
```

我们希望找到并输出公司中薪水最高的雇员的姓名。正如前面创建了Employee的一个子类HourlyEmployee一样，现在需要创建Company的一个子类Company2。为什么？这是因为Company类中没有提及HourlyEmployee。可以很容易地在HourlyEmployee的用户Company类中补救这个问题。对Company2来说，所有的问题就在于Hourly Employee的方法接口上。Company2和Company只有一点不同：findBestPaid方法被覆盖，因为这个方法的employee对象定义为

```
HourlyEmployee employee;
```

而不是

```
Employee employee;
```

下面是Company2的声明：

```
#ifndef COMPANY2
#define COMPANY2

#include "company1.h"

class Company2 : public Company
{
    public:
        // 后置条件：求出薪水总额最高的计时雇员。数值相同的
        //           将忽略。
        void findBestPaid( );
}; // Company2

#endif
```

再就是Company2的定义：

```
#include "company2.h"
#include "hourlyEmployee2.h"
void Company2::findBestPaid( )
{
    HourlyEmployee employee;

    employee.readInto( );
    if (!employee.isSentinel( ))
    {
        atLeastOneEmployee = true;
        while (!employee.isSentinel( ))
        {
            if (employee.makesMoreThan (bestPaid))
                bestPaid.getCopyOf (employee);
            employee.readInto( );
        }
```

```
        } // 输入没有以结束标记开始
    } // findBestPaid
```

这个方法的新奇之处在于行

```
bestPaid.getCopyOf (employee);
```

Employee类的getCopyOf方法没有被覆盖，它指定的形参类型是Employee。但是在Company2的findBestPaid定义中调用getCopyOf方法时，变元employee是HourlyEmployee类型的，而不是Employee类型的。因为HourlyEmployee是Employee的一个子类，所以无论何时在一个表达式中调用Employee对象，都可以用HourlyEmployee对象替代。这是子类替换规则的一个应用:

子类替换规则

无论何时在一个表达式中调用一个超类对象，都可以用一个子类对象替换。

子类替换规则说明一个子类对象也是一个超类对象。例如，一个HourlyEmployee也是一个Employee。但是下面的语句是非法的:

```
employee = bestPaid;//非法的
```

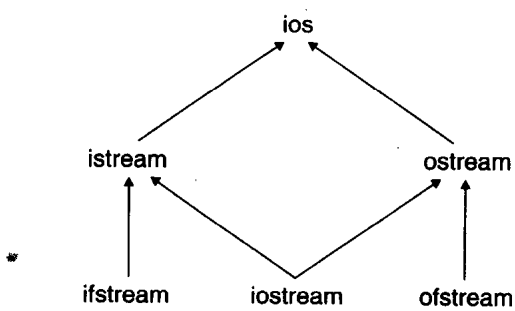
这个赋值语句是非法的，这是因为一个Employee不必是一个HourlyEmployee。这里不能应用子类替换规则，因为赋值语句的左边必须是一个变量，而不是一个任意的表达式。

实验2包含了关于继承的更多的细节。

23

实验2：关于继承的更多的细节 （所有实验都是可选的）

在前面的示例中，超类Employee只有一个子类HourlyEmployee，并且Company也只有一个子类Company2。某些情况下，可能得到类的完整的层次关系。例如，一个“**流**”是从源传送到目的地的一系列信息。下面是C++的流类的部分层次:



简单地说，ios类处理低层的输入－输出，像状态位和相关方法：eof()、clear()等等。istream类在ios中添加了提取运算符**operator>>**，ostream添加了插入运算符**operator<<**。ifstream类通过增加open和close方法扩展了istream类。iostream类通过为istream和ostream各添加一个专用构造器扩展了它们。这里最引人注目的是iostream，它是**多重继承**的一个范例:iostream是两个基类——istream和ostream的子类。

读者可能经常会遇到下面的情况。在开发一个类B时，发现其他一些类A的方法是非常有

用的。一种可能性是令B从A继承；也就是，B是A的一个子类。那么B可以使用A的所有方法。另一种方案是在类B中定义一个字段，它的类型是A。那么可以通过这个字段调用A的方法。这里重要的是要领会这两种访问类A的方式之间的区别。

继承描述了“**是一个**”关系。一个HourlyEmployee是一个Employee。一个iostream是一个istream。下面的说法也是正确的：一个istream是一个ios，因此一个iostream是一个ios。从数学的角度来说，是一个关系是传递的。

另一方面，类中的字段组成了类的“**有一个**”关系。例如，Employee类的name字段是string类型的，因此可以说一个Employee有一个string。同样，Company类的bestPaid字段是Employee类型的，因此可以说一个Company有一个Employee。

一般说来，如果类B分享了A的全部功能，那么B从A继承是更好的选择。但是如果B中只有一个组件能从A的方法中受益，那么比较好的选择是将类A的一个对象作为类B的一个字段，这个对象可以调用类A的有关方法。通常，选择不一定是非常清晰明确的，这时经验将是最好的向导。

使用面向对象方法时，并不强调程序的整体开发，而是强调开发模块化的程序部件，即 24
类。这些类不仅使程序更容易理解和维护，而且在其他程序中也可重用。这种方法更深一层的优点在于一个类的决策很容易修改。首先决定需要什么样的类；且因为每个类和其他类之间通过方法接口互相影响，所以可以随心所欲地修改类的字段和方法的定义——只要方法接口保持不变就可以。

1.1.10节中探索了C++的一个便利的特征，即这样一种能力：定义容易记忆的运算符，而不是容易忘记或易拼错的方法标识符。

1.1.10 运算符的重载

方法标识符带来了一个小麻烦，就是和通用运算符相比，它们需要特殊的名字。例如，在Employee类中，使用方法标识符readInto代替了插入运算符**operator>>**。C++允许用户使用运算符取代方法标识符，也就是说，可以在某一类中将运算符的意义扩展到一个方法。技术性的术语称之为**运算符重载**：给一个现有的运算符赋予另外的意义。实际上，运算符重载的例子是很常见的。比如，运算符+当操作数都是**int**型时就进行整数加法，当都是**float**型时就进行浮点加法，如果操作数都是string对象就进行连接操作。

为了说明类中运算符的重载，先看一个简单的例子。这里是Employee类中重载**operator>**的声明：

```
//后置条件：如果这个Employee的薪水总额高于otherEmployee的薪水总额就返回
//真。否则，返回假。
bool operator> (const Employee& otherEmployee) const;
```

这个声明和1.1.5节中makesMoreThan方法的声明是完全相同的，除了方法标识符makesMoreThan被替换成了

```
operator>
```

注意**operator**是关键字。

从这个声明中不难推断定义如下：

```
bool Employee::operator> (const Employee& otherEmployee) const
```

```
{
    return grossPay>otherEmployee.grossPay;
}//重载>
```

现在**operator>**代替了makesMoreThan方法，因此表达式

```
employee.makesMoreThan (bestPaid)
```

25 将被替换为

```
employee>bestPaid
```

通常，当一个方法被一个运算符取代时，调用对象（如employee）成为运算符左边的操作数。方法的变元（如bestPaid）成为运算符右边的操作数。运算符=的重载和运算符>的重载非常相似。细节问题参阅实验3。

1.1.11 友元

现在来处理运算符<<的重载。但是这个运算符是在ostream类中定义的，而不是在Employee类中定义的。因此难点是必须允许运算符<<访问Employee类中的字段。实现这个目标的一种方法是将这些字段设置成**public**来代替**protected**。但是这样将使每个用户都能访问到这些字段，那么用户和Employee类的具体实现之间就连接得过于紧密了。

我们所希望的是一个折中的办法，既保持Employee字段的**protected**状态，又令它可以被ostream类中的对象（如cout）访问到。C++提供了一个解决方案[⊖]，称作友元声明。在Employee的声明中，关键字**friend**指示**operator**<<是Employee的一个“友元”。给定两个类A和B，如果类A的方法m被允许访问类B的所有成员（public、private或protected），那么方法m是类B的**友元**。为了满足这个条件，类B也必须声明为类A的方法m的友元。下面是文件employee3.h中将**operator**<<声明成Employee类的友元的代码：

```
friend ostream& operator<< (ostream& stream,
                            const Employee& employee);
```

这个重载版本的运算符<<返回了对一个ostream（也就是输出流）对象的引用，因此employee3.cpp中运算符<<的定义是：

```
ostream& operator<< (ostream& stream, const Employee& employee)
{
    cout << employee.name << " $" << setiosflags(ios::fixed)
         << setprecision (2) << employee.grossPay << endl;

    return stream;
}//重载<<
```

因为运算符<<被声明成Employee类的友元，所以该运算符的定义允许访问Employee的

26 name和grossPay字段。

现在**operator**<<已经被重载了，可以重新编写findBestPaid()，将

```
bestPaid.printOut();
```

⊖ 可以说，C++的主要问题就是它为每个问题都提供了一种解决方案！这使得它是一个功能强大，但是很难掌握的语言。

替换成

```
cout<<bestPaid;
```

注意在findBestPaid()中书写语句

```
cout<<bestPaid.name<< "$" <<bestPaid.grossPay;
```

是不合法的，因为并没有允许findBestPaid()访问Employee的字段。换句话说，findBestPaid()不是Employee的友元。如果想使用这个语句，必须把findBestPaid()设置成Employee的友元。实际上，可以通过在employee.h中进行声明将整个Company设置成Employee的友元。

```
friend class Company; //需要关键字"class"说明友元是一个类
```

那么Company类的findBestPaid()和printBestPaid()方法将可以访问Employee类的所有成员。注意友元关系不是一个对称的关系：这个声明使得Company成为Employee的友元，但没有令Employee成为Company的友元。因此Employee类不能访问Company类的非公有（没有设为**public**）成员。

除了成为友元，在运算符<<的声明中还有一个有趣的地方：返回值是一个ostream对象的引用。这是值得关注的，因为cout自身就是一个ostream对象。因此可以在同一个语句中多次使用运算符<<。例如，可以编写

```
cout<<BEST_PAID_MESSAGE<<bestPaid;
```

这个语句的第一部分——cout<<BEST_PAID_MESSAGE，返回对一个ostream对象的引用，然后这个对象在bestPaid上应用它的**operator**<<。通常，当一个运算符返回与调用它的对象类型相同的引用时，就可以把几个这种运算符的调用连接在一起。

实验3探讨了**operator**=的重载，它和**operator**>的重载很相似；还讨论了**operator**>>的重载，它和**operator**<<的重载很相似。

实验3：重载运算符“=”和运算符“>>”　　（所有实验都是可选的）

1.1.3节介绍了数据抽象原理，由此使用某个类的代码不应当访问该类的实现细节。这样会不会给用户造成负担？**protected**修饰符适合用在什么地方？**friend**怎么样呢？1.1.12节描述了C++禁止用户代码访问所使用的类的实现细节的限制程度问题。 27

1.1.12 信息隐藏

数据抽象原理说明了用户的代码不应当访问所使用类的实现细节。根据这个原理，可以保护用户代码不受那些实现细节的修改的干扰，比如像字段的修改。如果用户代码不能访问所使用类的实现细节，那么保护将进一步增强。在下面的原理中说明了这一限制：

信息隐藏原理

一个语言应当允许类的开发者禁止用户代码访问类的实现细节。

数据抽象原理的遵守是用户的责任，反之，信息隐藏原理为类开发者提供了一个语言特

性的支持。这两个原理的目的是相同的：保护类用户不受类的内部改动的影响；内部改动只影响类的定义，但不影响类的公有方法接口。

前面已经看到C++通过对方法和字段使用**private**和**protected**修饰符来支持信息隐藏。**private**修饰符获得完全的隐藏：只有类的方法才可以访问**private**的非局部变量和方法。**protected**修饰符对一般用户提供了完全的隐藏，但是它允许类的任意子类进行访问。

可见性修饰符**public**允许任何类进行访问。最后，C++提供一个“漏洞”来避免信息隐藏，即**friend**修饰符。正如在1.1.11节所见，如果类A用**friend**修饰符声明类B（或是类B中的一个方法），那么所有类A的成员，即使是**private**成员，都可以被类B中的成员（或是类B中声明为友元的方法）访问。

总结

在这一章以及实验1到实验3中，收集了C++的一些主题，这有助于读者理解和使用标准模板库。章中大部分内容都探索了C++的面向对象特性。例如，C++支持面向对象语言的这些基本特点：

封装：通过类、头文件和源文件以及作用域解析运算符。

继承：通过子类和**protected**成员。

面向对象语言的第三个基本特点——多态，在附录3中有详细的解释。

同时读者也看到了面向对象编程的三个相关原理：

28 *数据抽象原理*：用户的代码不应当访问所使用类的实现细节。

开放—封闭原理：每个类都应当被打开——也就是能够通过继承扩展，并封闭——也就是在现有应用中保持稳定性。

信息隐藏原理：一个语言应当允许类的开发者禁止用户代码访问类的实现细节。

习题

1.1 在dateMain项目中重新实现Date类的isValid方法——在本书网站的源代码链接中，假设Date类只有一个字段：

```
long theDate;
```

日期的格式是*(m)mddyyyy*，也就是说，月使用一位或两位，每月中的日使用两位，年使用4位。例如，值1042005代表2005年1月4日；值10042005代表2005年10月4日。

1.2 a. 在Date类中，为daysLeftInMonth方法开发一个方法接口。例如，如果myDate是一个Date对象，它的日期是2003年2月13日，那么myDate.daysLeftInMonth()将返回15。

b. 根据习题1.2a开发的方法接口定义daysLeftInMonth()方法。假设Date类使用**int**型字段day、month和year。

提示 *使用daysInMonth方法。*

1.3 在Employee类中，有一个equalTo方法，它的方法接口是：

```
//后置条件：如果这个Employee和otherEmployee的薪水总额相等就返回真。
//否则，返回假。
```

```
bool equalTo (const Employee& otherEmployee) const;
```

a. 定义这个方法。

b. 将equalTo方法替换成运算符 = =的重载版本。

提示 所有需要修改的就是头。

1.4 下面是一个简单类的头文件: 29

```
#ifndef AGE
#define AGE

class Age
{
        public:
                //后置条件: 这个Age被初始化为0。
                Age();

                //后置条件: 这个Age被初始化为newAge。
                Age ( int newAge);

                //后置条件: 返回这个Age。
                int getAge();

                //后置条件: 将这个Age设置成newAge;
                void setAge (int newAge);
        protected:
                int age;
};//类Age
#endif
```

a. 为Age类创建源文件 (Age.cpp)。

b. 创建一个文件以及main函数，在这个函数中定义Age对象并且调用每个方法至少两次。

1.5 根据习题1.4a中的Age类，开发一个Salary类读输入的薪水，直到到达结束标记(-1.00)；并输出超过平均值的薪水数量。薪水平均值是薪水的总和除以薪水的份数。除了一个缺省构造器，还有两个方法，它们的接口如下:

```
//后置条件: 求出输入 (结束标记 = -1.0) 的所有薪水的平均值。
void findAverageSalary();

//前置条件: 至少输入一个薪水值 (在结束标记之前)。
//后置条件: 输出在输入中超过平均值的薪水值。
void printAboveAverageSalaries();
```

假设除了最后一行之外，输入的每一行中包含一个合法的薪水值，并且输入中有至少1个至多100个这样的薪水。用一个main函数测试Salary类。

1.6 下面是一个简单类的头文件SimpleClass.h:

```
#ifdef SIMPLE_CLASS
```

30

```
#define SIMPLE_CLASS
class SimpleClass
{
        public:
                //后置条件: 这个SimpleClass对象被初始化成最小的GPA并输出。
                SimpleClass();

                //后置条件: 这个SimpleClass的最小GPA被初始化成gpa_in并输出。
                SimpleClass (float gpa_in);

        protected:
                const static float MIN_GPA;
                float gpa;
        }//SimpleClass
        #endif
```

开发相应的源文件SimpleClass.cpp。下面给出了一个main函数来测试SimpleClass类:

```
#include <iostream>
#include <string>
#include "SimpleClass.h"

using namespace std;

int main( )
{
      const string CLOSE_WINDOW_PROMPT =
           "Please press the Enter key to close this output window.";

      SimpleClass sc1;

      SimpleClass sc2 (3.2);

      cout << endl << endl << CLOSE_WINDOW_PROMPT;
      cin.get( );

      return 0;
} // main
```

输出将是

```
2
3.2
```

31

```
请按下“回车”键关闭这个输出窗口。
```

1.7 在习题1.6的SimpleClass.h中，将行

```
float gpa;
```

替换成行

```
float gpa = MIN_GPA;
```

当重新生成并返回项目时会出现什么情况?

1.8 在习题1.6的SimpleClass.cpp中，注释掉头文件中的缺省构造器，但不注释掉源文件

中的。重新生成并返回项目时会出现什么情况？如果注释掉源文件中的但不注释掉头文件中的缺省构造器会怎样？如果将源文件和头文件中的缺省构造器都注释掉又会怎样？

1.9 在习题1.6的SimpleClass.cpp中，使用下面的定义作为缺省构造器：

```
SimpleClass :: SimpleClass() { }
```

当重新生成并返回项目时会出现什么情况？

1.10 将方法定义加入到头文件是合法的。那么将方法定义放在源文件，和头文件分离开，这样会获得什么样的功效？ 32

编程项目1.1：一个Sequence类

在这个项目里，读者首先是一个类的开发者，然后又成为这个类的用户。开始时，给出Sequence类的一些方法接口，它们当中保持了一系列string类型的项。

```
//后置条件：这是一个空Sequence。
Sequence();

//后置条件：如果这个Sequence不能再储存任何项就返回真。否则，返回假。
bool full() const;

//后置条件：返回这个Sequence中项的当前数量。
int size() const;

//前置条件：这个Sequence中至少保有一个项。
//后置条件：在这个Sequence里添加s（插入到最后）。
void push_back (const string& s);

//前置条件：这个Sequence里至少有k个项。
//后置条件：返回对这个Sequence里位置k上的项的引用。
string& operator[ ] (int k );
```

第一部分：定义Sequence类的方法。

提示 使用下面的字段：

```
string data [ MAX_SIZE]      //保存项的数组
int count;                   //Sequence中项的当前数量
```

和

```
const static int MAX_SIZE = 10;     //整型常量是合法的
```

第二部分：创建一个main函数来测试Sequence类。每行读入一个单词，当到达结束标记（开发者自己选取的结束标记）时，求出Sequence中按字母顺序最小的和最大的单词，并将每个包含“no”的单词中的这部分换成“yes”，然后求出Sequence中介于“aardvark”和“panda”之间的单词数量。 33

第2章　容器类的存储结构

这一章仍然是为通用的数据结构和具体的标准模板库的学习做准备。从大多数人公认的最难而且最容易出错的C++的特点——即指针的概念——开始；但是指针同时也是功能非常强大而且应用非常广泛的，因此迟早都必须掌握它。本章中还介绍了容器类：数据结构的面向对象版本。**容器类**是指它的每个实例都收集了很多项的类。这里特别有意思的是容器类的存储结构，主要是数组和链式结构。本章的最后讨论了标准模板库的一个重要的组件——通用型算法，它是预定义的、模板函数，它增大了容器类中方法的收集。

目标

1) 理解指针和动态变量。
2) 探索顺序结构和链式结构的优缺点。
3) 学习Linked类中的字段和方法。
4) 能够创建并使用模板类。
5) 定义和使用迭代器。
6) 学习如何找到和使用通用型算法。

2.1　指针

现代编程语言允许编程者在程序运行过程中显式地创建和撤消变量。这些变量称作**动态变量**，它的存储区域在需要时进行分配，而当不再使用时释放。**堆**就是为动态变量保留的一大块内存区域。

指针变量包含了另一个变量的地址。

与普通变量不同，动态变量是决不能直接访问的。实际上，一个动态变量根本就没有一个标识符。相反，动态变量总是通过一个指针变量来间接访问。**指针变量**是一个变量，它包含了另一个变量，通常是动态变量的地址。指针类型由一个类型和随后的一个星号组成。举一个简单的例子，可以声明：

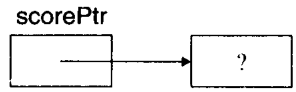
```
int* scorePtr;
```

这将变量scorePtr声明为一个指针变量。最终，scorePtr将包含一个**int**型变量的地址。要创建一个动态变量，并令scorePtr指向它，可以使用**new**运算符：

```
scorePtr = new int;
```

new运算符为一个动态变量分配了存储空间。

在这个例子中，**new**运算符的操作数是类型**int**，将为这个类型分配存储空间，并返回一个指针指向这个空间。换句话说，当执行赋值语句scorePtr=**new int**;时，将创建一个**int**类型的动态变量，并将这个动态变量的地址存放在scorePtr里。用图示方法，可以从指针变量到动态变量之间画一个箭头：



问号表示这个动态变量尚未赋值。如果想访问或修改动态变量必须通过指针变量，在这个例子中，就是scorePtr。这可以通过在scorePtr的**前面**放上一个星号来实现，例如，

```
*scorePtr=7; // 在动态变量中存储了7
cin>>*scorePtr; // 从输入中读入数据，放进动态变量
if(*scorePtr==0) // 测试动态变量中是否包含0
```

上面的例子里，动态变量是**int**类型的。在很多应用中，动态变量的类型是类。下面的例子中将创建一个动态对象，即一个动态变量，它的类型是一个类。回忆第1章中的Employee类。首先声明一个指针变量：

36

```
Employee* employeePtr;
```

就像对scorePtr所做的工作一样，现在通过调用**new**运算符创建动态Employee变量：

```
employeePtr = new Employee;
```

为了访问Employee类型的动态对象的一个成员（也就是一个字段或者方法），可以应用**脱引用**（即解除引用）运算符——星号：

```
(*employeePtr).readInto();
```

外面的圆括号是必须的，因为**成员选择**运算符——点——比脱引用运算符的优先级高。但是对一个指向动态对象的指针脱引用，然后选择这个对象的成员是一个非常常见的操作，C++中有一个关于这个应用的特殊符号，即**脱引用和选择**运算符——->，它可以按如下方式使用：

```
employeePtr->readInto();
```

在定义指针变量时必须小心，因为C++编译器隐式地将*和变量而不是类型联系在一起。例如，下面的语句

```
Employee*  emp1Ptr;
           emp2Ptr;
```

声明了一个指针变量emp1Ptr。变量emp2Ptr——尽管它的名字听起来像指针——只是一个普通的Employee对象。如果想声明同一类型的多个指针变量，可以在声明前面加上一个**typedef**，为指针类型给出一个名字。例如，

```
typedef Employee* EmployeePtr;
```

这条语句使得EmployeePtr成为类型Employee*的另一个名字。在**typedef**里声明的名字可以用来声明一个变量。例如，

```
EmployeePtr emp1Ptr,
            emp2Ptr;
```

就声明了两个指向Employee对象的指针变量。注意这里没有使用*，**typedef**只不过为一个类型声明了另一个名字。

另一种声明同一类型的两个指针变量的方式是使用两个声明语句：

```
Employee* emp1Ptr;
Employee* emp2Ptr;
```

2.1.1 堆和堆栈的对比

读者可能感到奇怪，为什么声明指向雇员的变量，而不直接声明雇员。一个原因是为了节约**堆栈**——存储局部变量的部分内存空间。假设在Company类的findBestPaid()方法中声明 37
emp1Ptr和emp2Ptr。因为emp1Ptr和emp2Ptr都是局部的，所以只需要分配指针自身的堆栈空间即可，而保存两个雇员需要的堆栈空间要多很多。当然，使用动态变量，那么两个雇员将存放在堆中，而堆比堆栈要大多了。如果一个类的每个对象都由很多项组成，那么使用堆比使用堆栈，其优势是不言而喻的。

2.1.2 引用参数

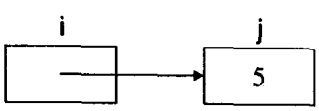
引用参数代表对指针的一个灵活运用。**引用参数**是一个不可修改的指针，它是自动脱引用的。当调用函数时，指针获取相应变元的地址；在函数的整个执行中，指针保持了相同的地址。函数执行过程中，指针自动脱引用；也就是说，如果x是一个引用参数，那么函数里每次出现的x都被解释为

```
*x
```

例如，假设有下面的程序段：

```
void sample (int& i)
{
      i = 3;
}// sample
int main( )
{
      int j = 5;
      sample (j);
      cout << j;
      return 0;
} // main
```

正如定义所指出的，j是一个**int**类型的变量，它的初始值是5。参数i的类型不是**int**，而是指向**int**的指针。当从main函数中调用sample函数时，变量j的地址被拷贝到i中。下面的图显示了调用时内存相关部分的内容：



因为引用参数是自动脱引用的，所以赋值语句

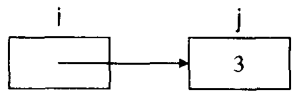
```
i=3;
```

被解释成

```
*i=3;
```

38

当执行这个语句时，*i，也就是j的值将发生改变：



因此程序的输出将是3，而不是5。

2.1.3 指针字段

类可以包含指针类型的字段，包括指向它自身类型的指针，例如，

```
class Student
{
    public:
        . . . // 方法声明

    protected:
        string name;
        float gpa;
        Student* next;
}; // Student
```

那么一个Student对象中的一个字段是指向另一个Student对象的指针，然后这个对象的一个字段又是一个指向另一个Student对象的指针，等等。这样的类可以用来将任意数量的学生记录“链接”在一起。指针的这个非常重要的应用将在2.3.2节叙述，它是标准模板库中的几个类的最通用实现的基础：它们是list、map和set。

指针的另一个应用是对多态性（这是附录3的主题）的支持。

2.1.4 数组和指针

指向数组的指针是很特别的。指向数组元素的指针是元素指针类型的。例如，指向一个**int**类型的数组的指针可以声明成：

```
int* scores;
```

注意这里没有说明数组的大小，而实际上，scores既可以指向单个**int**类型变量，也可以指向一个**int**类型的数组。

为了在堆中分配一个数组，需要将元素数量填入一对方括号中。例如，

```
cin>>n;
scores = new int[n];
```

39

为包含n个整数的数组分配了空间，并在scores里存储了数组第一个元素的地址。注意数组的大小是n中存储的值，并且这个值直到运行时才知道。通常情况下，从堆中为数组分配空间时，必须在编译时就清楚数组的大小；而它可以在运行时再确定！

如何访问数组元素呢？下标运算符**operator**[]，自动脱引用一个指向数组的指针，并访问对应的内存位置。在前面关于C++的学习中，读者必定已经注意到数组第一个元素的下标是0。例如，scores[0]将访问数组的第一个元素，而scores[2]将访问数组的第三个元素。注意在这个表达式中没有星号。这是因为数组变量被看作是一个自动脱引用的指针，它指向指定数组的第一个元素。

通用数组指针定理

通常，对任意数组a和非负整数i，a[i]和*(a+i)是等价的。

例如，使用前面定义的scores，scores[3]访问的元素与*(scores+3)访问的元素是相同的。

如果在一个函数中简单地声明了：

```
float a[100];
```

那么不是从堆，而是从堆栈中为数组分配空间。任何存储在堆栈中的数组的大小必须给定为一个常数。即便这样，数组指针定理仍然成立。

实验4增强了指针变量赋值和动态变量赋值之间的区别。

实验4：指针变量赋值与动态变量赋值的对比　　（所有实验都是可选的）

2.1.5 动态变量的存储空间释放

动态变量的内存空间是在调用**new**运算符时分配的，但是怎样释放呢？不再访问的动态变量称作是**无用单元**或是一个**内存泄漏**。如果程序中产生了过多的无用单元，那将造成内存溢出，这是一种错误的情况。那么开发者有没有责任进行无用单元收集，即释放不再访问的动态变量的空间呢？

delete运算符释放一个动态变量占据的存储空间。

不幸的是，开发者必须处理自己的无用单元。C++提供了一个**delete**运算符用来释放一个动态变量占据的存储空间。这个运算符只有一个操作数——即指向即将被释放的动态变量的指针。因此可以编写下面的程序：

40

```
Node* nodePtr;
nodePtr=new Node;
//使用*nodePtr
.....
delete nodePtr;
```

最后一条语句释放了nodePtr指向的动态变量占据的存储空间；现在脱引用nodePtr是非法的。并且，不论是否调用**delete**运算符，都可以通过为指针变量赋以常量标识符NULL，来表示这个指针变量不再指向任何空间了：

```
nodePtr=NULL;
```

要释放一个动态数组的空间，可以在关键字**delete**的后面使用空下标，例如，

```
string* names;
names=new string[500];
//使用数组names
.....
delete[] names;
```

2.3.5节将看到使用**delete**运算符释放大量动态变量的存储空间。

2.2 数组

回忆一下，2.1.4节提到，一个数组变量实际上是一个指针变量，它包含了数组第一个条目的地址。例如，运行下面的程序：

```
string* names;                  //将names声明为字符串指针类型
names=new string[5];            //将names定义成（一个指针，指向）包含5个
                                //字符串的数组
names[0]= "Cromer";             //在names数组的第一个条目里存储"Cromer"
names[3]= "Panchenko";          //在names数组的第四个条目里存储"Panchenko"
```

根据数组指针定理，前一条将"Cromer"赋给names[0]的赋值语句等价于

```
*names="Cromer";
```

将"Panchenko"赋给names[3]的赋值语句等价于

41

```
*(names+3)="Panchenko";
```

一旦创建了数组，那么它的大小就是固定的，而新的数组（由同一个指针变量指向的）可以迟些创建。例如，可以编写

```
string* names;
int n;

cin>>n;
names=new string[n];
.....
delete[] names;//避免内存泄漏
names=new string[2*n];
```

数组中相邻的元素是**连续**存储的，也就是存储在相邻的位置上。这种邻接性的一个重要结果就是在访问数组中一个单独的元素时，不需要先访问任何其他的元素。例如，可以直接访问names[2]，而不需要先访问names[0]和names[1]再到达names[2]。数组的这种**随机访问**属性在第5、11、13章中很容易看到。在任何情况下都需要一个存储结构，在这个结构中给定相对位置就能快速地访问到元素，因此数组在任何情况下都是适用的。

数组也有几个缺点。首先是因为元素是连续的，数组的大小是固定的；所以整个数组的空间必须在元素存入数组之前分配。如果太大，就会有很多空间不能使用；如果太小，就必须再分配一个大的数组，然后将小数组的内容转移到大数组里。

数组的另一个缺点是它的插入和删除需要移动太多元素。例如，假设一个数组的下标从0到999，并且现在从0到755的下标上都存储了元素。如果一个新的元素要放到下标300上，那么在这个元素被插入之前，必须先将下标在300至755之间的元素依次移动到301至756之间的下标处。图2-1显示了这样插入的效率。

到目前为止的编程工作中，读者可能还没有正确的评价数组的随机访问特性。那是因为现在仍没有看到除了数组之外，内存中还有存储元素集的其他方式。2.3节里介绍了这种方式，它在动态对象上也是非常通用的。

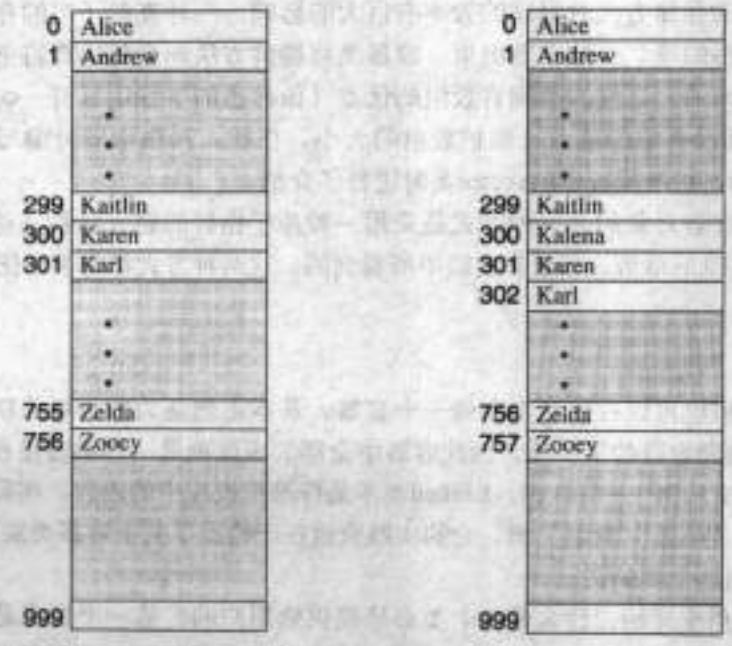


图2-1 在数组中进行插入：将"Kalena"插入到左边数组的下标300处，必须先将300，301，…，755下标处的元素相应的移动到下标301，302，…，756的位置上

2.3 容器类

容器类是一个类，它的每个实例都包含了很多项。

容器是一个变量，它由很多项的集合组成。迄今为止接触到的惟一的容器就是数组，但是以后几乎全部的工作都专注于其他的容器，即容器类的实例。**容器类**是一个类，它的对象都是容器。第5~14章将学习到许多应用广泛的容器类。通常，这些容器类都由相似的方法接口。例如，每个容器类都有一个empty方法，它的接口是：

```
//后置条件：如果容器类中没有项就返回真，否则，返回假。
bool empty() const;
```

假设myList是容器类list的一个对象，并且有四个项，执行

```
cout<<myList.empty();
```

将输出

```
0
```

因为在C++中，常数0和**false**是同义的（并且它们和NULL也都是同义的）。

当然，方法接口无法说明方法的任务是如何实现的。在第5~14章中，将探讨学到的每个容器类的细节。但是现在根据项的存储方式，可以将容器类简单的分类。

2.3.1 容器类的存储结构

容器对象，也就是容器类的实例对象，通常占据与容器中项的数量成比例的内存空间。

因此容器在内存中的存储方式对程序的效率有巨大的影响。一种直截了当的存放容器对象的方法是连续存储单独的项：存储在数组里。容器类将提供方法和重载运算符来创建并维护数组。从用户的角度来看，这样的类拥有数组的优点（像熟悉的下标运算符—**operator**[]，和随机访问性质），而没有它的缺点（维护数组的大小，在插入和删除项时编写代码的缺陷）。这种方法在第5章学习标准模板库的vector类时进行了介绍。

在内存中存储容器对象的另一种方式是采用一种基于指针的链式结构（这是2.32节的主题）。正如读者将在以后章节、项目和实验中所看到的，这两种方式都有很广泛的应用。

2.3.2 链式结构

现在不使用数组也可以在内存中存储一个容器。基本思想是为每个项关联一个**链接**（也就是一个指针），指向容器的下一项。因此容器中全部的项像项链一样都链接在一起。现在创建一个简单的容器类示例——Linked。Linked类不是标准模板库中的内容。实际上，它只有三个方法（初始时），是一个“玩具”类，它的实践价值在于给出了关于容器类实现的一些观点。实验5、6和7是使用这个类的试验。

照常从用户的角度开始。什么是这个类必须提供给用户的？第一个要点是需要指明项的类型是什么。是需要int型，Employee对象或其他类型的对象的Linked容器吗？无论选择哪一个都将彻底限制容器类的实用性。最好的选择是让用户自己选择！也就是说，让Linked类的用户在定义Linked容器对象时决定项的类型。创建这个灵活的Linked类依赖于C++的一个强大功能——模板。

当一个容器类被模板化时，类的每个实例都包含一个模板变元：单独项的类型。

不用定义单个容器类，而是定义一个**模板**（或者说模子），它允许在编译时创建某些固定类型的容器类。这个固定类型好像是容器类的一个“变元类型”。这里模板类的名字是Linked，用户跟在带**模板变元**[⊖]的类标识符后，模板变元在类名的后面以角括号括起。例如，这里给
44 出了一个**int**类型和一个Employee对象类型的Linked容器的定义：

```
Linked<int> intList;
Linked<Employee> employeeList;
```

从这个观点来说，除了**int**型项，向intList中插入任何项都是非法的；除了Employee对象，向employeeList中插入任何项也都是非法的。在补充Linked类的方法接口之后将给出一些示例。

现在看一下在Linked类的定义中是如何处理模板的。开始是关键字**template**，后面跟着角括号，及其里面的关键字**class**和一个标识符：**模板参数**。这个标识符将在编译时当类被实例化时，由用户提供的类型替代——像**int**或是Employee。下面是定义的开头：

```
template<class T>
class Linked
{
```

当模板类被实例化时，模板变元替换类的定义中出现的每个模板参数。

在下一段中将会看到，无论何时需要在Linked类的定义中提供一个项的类型，都简单地

⊖ 为了简单起见，假设这里是单个的模板变元。后面还将遇到有多个模板变元的类。

使用T。读者可能会感到奇怪，这样一个强大的特性只需要做这么少的工作。最难的部分，也是完全可以忽略的，就是编译器正确高效地实现模板。

现在，Linked类将只有三个职责：创建空的Linked对象，返回Linked对象中项的数量，在Linked对象的前面插入一个新的项。下面是方法接口：

```
//后置条件：这个Linked对象是空的，也就是，没有任何元素。
Linked();

//后置条件：返回Linked对象中项的数量。
long size() const;

//后置条件：在这个Linked对象的前面插入newItem
void push_front(constT& newItem);
```

注意push_front的参数是类型T，是模板参数。

现在可以修改那两个Linked对象，intList和employeeList：

```
intList.push_front (27);
intList.push_front (51);
intList.push_front (12);

Employee employee;

employee.readInto( );
while (!employee.isSentinel( ))
{
      employeeList.push_front (employee);
      employee.readInto( );
} // while
```

45

每个项都被插入在它的容器的前面，例如，intList现在包含下列顺序的项：

12，51，27

从用户的角度来看，对Linked类没什么可说的，所以将注意力转移到这个类的字段和实现上。一种可能性是使用一个数组字段并将项存储在数组里。习题2.3探讨了这种选择。

另一个策略是将容器的每一项存储在一个名为Node的**结构（struct）**[⊖]里。Node将包含两个字段：一个item的字段是类型T，next字段的类型是Node*，也就是指向Node的指针：

```
struct Node
{
      T item;
      Node* next;
}; //结构Node
```

Node将是Linked中的一个**protected**类。因为Node的所有成员都是**public**的，所以Linked中的任何方法都能访问到Node的item和next字段。Linked中仅有的字段是：

```
Node* head;//指向第一个节点
```

⊖ **结构**是一个类，它的每个成员都是**public**的。在一个**结构**中定义方法是合法的，不过通常情况下，一个**结构**对象只包含字段。

```
long length;//容器中项的数量
```

缺省构造器的定义非常直接:

```
Linked()
{
    head=NULL;
    length=0;
}//缺省构造器
```

可以看到作用域解析运算符::没有出现在定义中。那是因为这个定义是放在头文件linked.h里的。到目前为止，每个类都有一个包括声明的头文件以及包含方法定义的源文件。一般说来，分配存储空间的代码都不应该出现在头文件里；那么可以预编译头文件，而不用在每个项目中重新编译。但是使用模板时，在定义类中的对象前不会进行存储空间的分配。

46 因此，为了简化起见，就把声明和方法定义都放在头文件linked.h中。

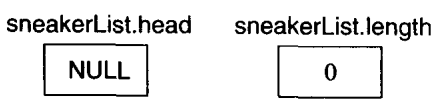
size()方法的定义甚至比缺省构造器的定义还要简单:

```
long size() const
{
    return length;
}//方法size
```

在讨论push_front方法的定义之前，先看一下调用这个方法会出现什么情况。假设已经在main函数里定义了一个Linked对象，例如:

```
Linked<string> sneakerList;
```

这时调用了缺省构造器，因此得到:



现在在sneakerList的前面添加一个项:

```
sneakerList.push_front("Nike");
```

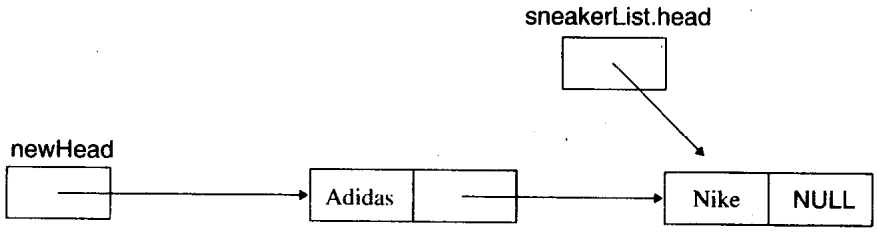
这个消息的功效是创建一个sneakerList.head指向的Node对象。这个节点的item字段将包含值“Nike”。节点的next字段不指向任何地方，因此它的值是NULL。当然，sneakerList.length也必须相应增加。这得到了:



再一次，用户所要做的一切就是在sneakerList的前面添加一个项:

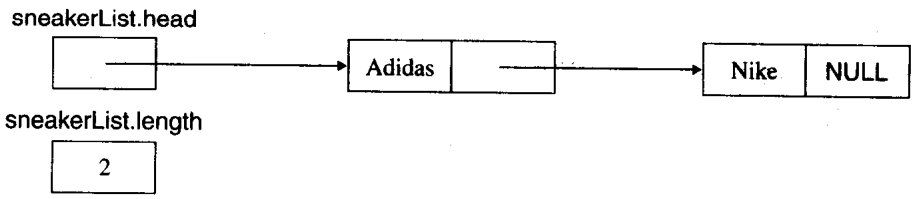
```
sneakerList.push_front("Adidas");
```

首先，将创建一个新的Node对象。当然，这里并不希望sneakerList.head指向这个新节点，因为这样将导致带有“Nike”的节点不可访问了。而采用的方法是在创建新的Node对象时，令一个Node指针newHead指向它。新节点的item字段得到值“Adidas”。新节点的next字段指向最初的节点（item字段中值是“Nike”的节点）。现在得到:



47

最后，在sneakerList.head中存储地址，这也是newHead里的地址：



根据迄今为止所学到的，应当可以解决所有容器项的存储问题：将每个项输入到一个Node对象，再把它插入到Linked对象的前面。这里是push_front的完整的定义：

```
void push_front(const T& newItem)
{
    Node* newHead=new Node;
    newHead->item=newItem;
    newHead->next=head;
    head=newHead;
    length++;
}//方法push_front
```

这个例子过于简单，因为Linked类只有很少的功能。在实际的应用中，可能需要一个方法删除Linked对象中前面的项，并且可能希望用户能够访问列表中的每个项。但是还需不需要一个方法来输出项，另一个方法来查找最大的项，等等？2.3.3节将介绍迭代器，它是循环通过一个容器全部项的问题的完美解答。

Linked实现中的要点是链式结构存储容器项和数组结构有几个关键不同：

1) 不需要提前知道容器的大小，可以随意地添加项。因此不必像在数组中一样，担心分配的空间过多或过少。但是应当注意到，在每个Node对象里，next字段自身都占据了额外的空间，因为它所容纳的是程序信息而不是问题的信息。

2) 随机访问是不能实现的。要创建方法找到一个项，应当从访问head项（即，head指向的Node对象中的item字段）开始，然后访问头项的下一项，依次类推。

2.3.3 迭代器

从2.3.2节可以看出，为了使Linked类有应用价值，希望用户能够循环通过一个Linked容器。在不违背数据抽象原理，也就是说不允许用户代码访问Linked类的实现细节的前提下，这是怎样实现的呢？答案是用户的迭代器。迭代器是一个对象，它允许容器的用户循环通过容器且不违背数据抽象原理。

48

迭代器允许用户代码循环通过一个容器且不访问容器类的实现细节。

几乎每一个容器类中都嵌入了迭代器类，它允许用户访问容器的每一项。容器类有begin()和end()方法。begin()方法返回了位于容器对象第一项的迭代器，end()方法返回位于容器对象最后一项之外的迭代器。

迭代器类还必须提供什么其他的方法来循环通过一个容器呢？还需要一个迭代器运算符——**operator++**，它将调用的迭代器对象前进到容器的下一项。如果迭代器位于最后一项，**operator++**将把迭代器定位到最后一项之后的位置上。另外，也应该有一个迭代器的**operator***用于对迭代器当前位置上的项脱引用。最后，还要有**operator==**和**operator!=**运算符方法测试迭代器的相等和不等情形。

下面的main函数阐述了薪水的Linked容器上迭代器的使用。程序读入一列薪水，然后输出高于平均薪水的每个薪水值。薪水值使用push_front方法存放在Linked容器中，然后一个迭代器循环通过Linked容器并输出大于平均薪水的每个薪水值。

下面是主函数：

```
int main( )
{
    const string PROMPT =
        "Please enter a salary; the sentinel is ";
    const string RESULTS =
        "Here are the above-average salaries:";
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";
    const float SENTINEL = -1.00;
    Linked<float> salaryList;
    float salary,
        total = 0.00;
49  cout << PROMPT << SENTINEL << ": ";
    cin >> salary;
    while (salary != SENTINEL)
    {
        salaryList.push_front (salary);
        total += salary;
        cout << PROMPT << SENTINEL << ": ";
        cin >> salary;
    } // 读入并合计薪水
    float average;
    if (salaryList.size( ) > 0)
        average = total/salaryList.size( );
    Linked<float>::Iterator itr;
    cout << RESULTS << endl;
    for (itr = salaryList.begin( ); itr != salaryList.end( ); itr++)
        if (*itr > average)
            cout << *itr << endl;
```

```
    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );
    cin.get( );

    return 0;
} // main
```

特别需要注意的是itr的声明:

```
Linked<float>::Iterator itr;
```

一般情况下，对每个容器类而言，它的迭代器类被嵌进了容器类中；这是之所以需要作用域解析运算符的原因。迭代器类提供了遍历容器类的方法。换句话说，迭代器类是幕后工作者，它使得容器类的用户能循环通过一个容器。

2.3.4 Iterator类的设计和实现

现在开发支持Linked类的Iterator类。由于C++的效率难题，所以将结构Node对象和Iterator类嵌入Linked类。因此只要简单地指定字段，Linked的方法就可以访问Node或者Iterator的任意字段。在创建Iterator类时，也需要扩展Linked类（例如，begin()和end()方法）。

下面是Linked类的大体轮廓:

```
template<class T>
class Linked {

    protected:
        struct Node
        {
            T item;
            Node* next;
        }; // 结构Node
        Node* head;
        long length;

    public:
        class Iterator {

            // 公有、私有和受保护的迭代器成员
            . . .
        } // 类Iterator
        // Linked类的方法的定义
        . . .
} // 类Linked
```

50

简化的Iterator类有一个字段:

```
Node* nodePtr;
```

用一个Iterator构造器初始化nodePtr:

```
//后置条件: 通过newPtr初始化迭代器。
Iterator(Node* newPtr)
{
```

```
    nodePtr=newPtr;
}//构造器
```

这个构造器不应当是**公有的**，因为用户不应访问Node类。但是在第1章中曾提到，只要一个类有任何的构造器，就不会自动定义缺省构造器。因此将显式地定义一个**公有**缺省构造器，这样用户可以构造一个Iterator对象。Iterator类的定义部分是

```
public:
    //后置条件: 构造这个Iterator对象。
    Iterator(){}//缺省构造器
```

后加运算符——**operator++(int)**的定义分三步。首先，为一个临时Iterator对象temp赋一个调用对象的值；然后增加nodePtr；最后，返回temp。调用对象的**赋值**反映了C++的一个灵巧的特点，**this**指针。在一个方法中，关键字**this**指向调用对象，因此***this**就是调用对象自身。

怎样“增加”指针nodePtr呢？答案是令nodePtr指向下一个Node对象，也就是，利用赋值语句:

```
nodePtr=(*nodePtr).next;
```

下面是代码:

```
//前置条件: 这个Iterator对象定位于某一项。
51 //后置条件: 这个Iterator对象在Linked对象中前进，并返回调用前迭代器
//位置上的项。
Iterator operator++(int)
{
    Iterator temp=*this;
    nodePtr=(*nodePtr).next;
    return temp;
}//后加++
```

虚参数类型**int**，用来表示运算符++的后加版本。实验5包含了后加运算符的定义和Iterator类的其他几个运算符。

实验5：定义其他的迭代器运算符　　（所有实验都是可选的）

2.3.5 pop_front方法

Linked类中另一个有用的方法是pop_front，它删除容器前面的项。这里是方法接口:

```
//前置条件: 这个Linked容器非空。
//后置条件: 这个Linked容器前面的项已被删除。
void pop_front();
```

pop_front的一种可能的定义如下:

```
void pop_front()
{
    head=(*head).next;
    --length;
```

```
}//方法pop_front
```

这个定义很精致，但有一个危险的缺陷：进行调用之前的前面的节点现在成为无用单元——它仍占据内存，但不再是可访问的。这个无用单元可以不断堆积，并最终导致程序运行的内存溢出。解决的方法是调用**delete**运算符释放不使用的空间。下面是修正后的pop_front版本：

```
void pop_front()
{
    Node* oldHead=head;
    head=(*head).next;
    delete oldHead;//释放*oldHead
    --length;
}//pop_front
```

52

oldHead指针保证调用开始时head指向的调用的开始位置的空间不被释放，直到调整head之后。而且oldHead自身的空间将在pop_front调用结束后自动被释放。

2.3.6 析构器

如果想删除所有Linked容器的项怎么办？一种方法是由用户直接去做，迭代通过容器并在每次迭代中调用pop_front。更好点的主意是定义一个destroy方法，由它迭代通过容器并在每次迭代中调用pop_front。这个方法的问题是用户可能忘记调用destroy，特别是对于不再访问的容器而言。例如，如果在一些方法中定义了一个Linked对象，在这个方法执行完之后，对象就**超出作用域**了，也就是，这个对象再也不能被访问了。

当一个类的实例超出作用域，也就是不再能被访问到时，就自动调用析构器方法。

回忆前面讨论初始化对象时说过，由于用户的健忘性促使C++的开发者提供了构造器：即当定义一个对象时自动调用的初始化方法。为了删除超出作用域的对象，用户的健忘性又促使C++的开发者提供了一个**析构器**：它是一个方法，当对象超出作用域时，自动调用它来释放对象的空间。

析构器的头以“~”开始，后面跟着类标识符，然后是圆括号。例如，这里是Linked析构器的方法接口：

```
//后置条件：释放Linked对象占据的空间。
~Linked()
```

注意 *析构器是没有返回类型也没有参数的！*

下面是完整的定义：

```
~Linked()
{
    while(head!=NULL)
        pop_front();
}//析构器
```

回忆在前面的叙述中讲过，如果没有为类创建构造器，那么编译器将自动生成一个。这个缺省构造器没有参数，并只是为类的每个对象字段而调用它。但是小心些！如果创建了任

一的构造器，编译器将不再创建缺省构造器；那么如果需要就必须显式地定义一个缺省构造器。而且当创建给定类的一个子类时将需要它，因为子类构造器开始时会自动调用它们的超
53 类缺省构造器。

同理，如果没有为类创建析构器，编译器将自动生成一个。这个缺省析构器只是为类的每个对象字段而调用。但因为Linked类中没有对象字段，所以这样的一个析构器没什么作用。这也是必须再定义我们自己的析构器的原因。

最后，需要说一下常量标识符。例如，假设在Linked类的声明/定义中声明了下面的语句：

```
protected:
    const static string HEADING;
```

在类声明之后和#endif之前不会出现这个标识符的定义。定义必须模板化，因为Linked就是一个模板类：

```
template<class T>
const string Linked<T>::HEADING="This is a linked list.";
```

只有在Linked类或它的子类的方法里才可以访问HEADING。

实验6处理了Linked类的另一个扩充：重载运算符**operator=**。

实验6：重载运算符operator=　　（所有实验都是可选的）

2.3.7节说明了对很多容器类而言有大量的预定义函数可以实现一些通用的工作。

2.3.7 通用型算法

正如不应重复地创建已经存在的类，在创建方法时也不应定义已经定义过的方法。标准模板库不仅提供了大量有用的容器类，而且还提供了大量的函数——像排序、查找、拷贝、累加等等——它们可以应用在容器对象上。作为额外收获，这些函数（称作**通用型算法**）可以应用在数组上！并且为了更灵活，所有的通用型算法都是模板函数。

模板函数的主要思想和模板类是相同的。一个函数的模板是函数定义的框架；模板包括一个或多个未指定的类型。当调用这个函数时指定类型，这样编译器可以为函数生成相应的代码。例如，调用模板函数swap可以交换两个未指定类型的变量的值；这个类型可以是容器（通过重载运算符**operator=**）或者甚至可以是**int**类型的。下面是直接从标准模板库的惠普实
54 现的文件algorith.h中得到的函数定义：

```
template <class T>
inline void swap (T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

前面曾讲过**template**是一个关键字，并且总是跟着**<class...>**。这次关键字**class**后面的类型标识符T是未定义的。当函数调用被编译器翻译成机器语言时，变元的类型就取代了未指定类型。类型标识符可以是一个类，像string、Employee或Linked；也可以是一个简单类型，

像**int**或**double**。无论何时调用这个函数时，关键字**inline**总是告知编译器直接生成机器代码，而不是函数调用。例如，假设有：

```
float x,
      y;
...
swap (x, y);
```

它生成的机器代码将和下面代码的机器代码相同：

```
float x,
      y;
...
float tmp = x;
x = y;
y = tmp;
```

处理嵌入函数的效率要比处理函数调用的高很多，因为它不需要将变元传递给参数，将控制权交给被调用的函数并在调用完成后返还控制。嵌入函数不能包含循环，而且必须是无递归的（递归将在第4章介绍）。

下面的程序swap.cpp，先交换两个string的值，然后又交换了两个**int**的值：

```
#include <iostream>
#include <algorithm> // 定义了大多数通用型算法
#include <string>
using namespace std;
int main( )
{
      const string CLOSE_WINDOW_PROMPT =
            "Please press the Enter key to close this output window.";
      string s1 = "yes",
            s2 = "no";
      swap (s1, s2);
      cout << s1 << " " << s2 << endl; // 输出 :: no yes
      int i1 = 58,
         i2 = 902;
      swap (i1, i2);
      cout << i1 << " " << i2 << endl; // 输出 :: 902 58
      cout << endl << endl << CLOSE_WINDOW_PROMPT;
      cin.get( );
      return 0;
} // main
```

55

另一个模板函数的例子是合计数组中元素的值。在这个程序中定义了函数add_up并调用它两次，一次是合计**double**类型的数组，另一次是合计**int**类型的数组。

```
#include <iostream>
#include <string> // 声明string类
using namespace std;

// 后置条件: 返回a[0...n-1]中元素的总和。
template <class T >
T add_up (T a[ ], int n)
{
      T sum = 0;
      for (int i = 0; i < n; i++)
            sum = sum + a [i];
      return sum;
} // add_up

int main( )
{
      const string DOUBLE_MESSAGE =
            "The sum of the doubles is";
      const string INT_MESSAGE =
            "The sum of the ints is ";
      const string CLOSE_WINDOW_PROMPT =
            "Please press the Enter key to close this output window.";
56    const int SIZE1 = 5;
      const int SIZE2 = 20;
      double weight [SIZE1] = { 3.2, 3.1, 2.9, 3.1, 3.0 };
      int count [SIZE2];
      cout << DOUBLE_MESSAGE << add_up (weight, SIZE1) << endl
            << endl;
          // 输出:: double型的总和是15.3
      for (int i = 0; i < SIZE2; i++)
          count [i] = i;
      cout << INT_MESSAGE << add_up (count, SIZE2) << endl
            << endl;
          // 输出:: int型的总和是190
      cout << endl << CLOSE_WINDOW_PROMPT;
      cin.get( );
      return 0;
} // main
```

模板函数add_up有一个缺点：它只能用于数组。虽然迄今为止只介绍了一个容器类(Linked类)，但后面还有关于容器类的大量描述。因此最好是有一个模板函数，它不仅能计算数组中各项的合计，而且能计算容器对象中各项的合计。

这样的函数是存在的。它就是numeric.h文件中定义的accumulate函数。下面列出了完整的定义：

```
//后置条件: 返回init和容器里从first位置（包括first）到last位置
//          （不包括last）之间的所有项的总和
template <class InputIterator, class T>;
T accumulate (InputIterator first, InputIterator last, T init)
{
    while (first != last)
        init = init + *first++;
    return init;
} // accumulate
```

这个函数有两个模板类型：InputIterator和T。T是至今尚未定义的类型，初始值、返回值以及迭代器first和last位置上的值都是这个类型的。参数first和last的类型是InputIterator，但是一个输入迭代器有多大容量呢？一个输入迭代器必须能够：

访问（不需要修改）容器的每一项；

前进到容器的下一项；

判断是否到达了容器的末端。

输入迭代器这样命名是因为它们反映了读一个输出流的行为。输入迭代器必须支持的运算符只有：

迭代器脱引用，即**operator***；

迭代器自加，即**operator++**；

迭代器相等性测试，即**operator==**和**operator!=**。

注意这些正是accumulate用在first和last迭代器上的运算符！因为Linked类中嵌入的Iterator类包含了这些运算符，所以Iterator类就是刚才定义的InputIterator的广义范畴。也就是说，可以用accumulate合计Linked容器中的项。前面提到的Linked方法begin()和end()返回Iterator，它和刚才看到的输入迭代器的作用是一致的。下面是代码：

```
Linked<int> list;
...
int sum = accumulate (list.begin( ), list.end( ), 0);
```

通用型算法是一个模板函数，它可以通过迭代器应用在容器对象上，也可以通过指针应用在数组上。

模板函数accumulate是通用型算法的一个例子。**通用型算法是一个模板函数，它可以应用在容器对象和数组上。要累加数组中的项，需要的全部变元是指向开头的指针和指向最后一项之后位置的指针，以及一个初始值。前面讲过，其实一个数组变量自身就是指向数组开头的指针。因此数组变量加上数组的大小就是指向数组最后一项之后位置的指针。**可以重写程序来计算**double**类型的数组和**int**类型的数组的和，只要去掉add_up的定义，并将两个add_up调用替换成：

```
cout << DOUBLE_MESSAGE << accumulate (weight, weight + SIZE1,
            0.0) << endl << endl;
```

和

```
cout << INT_MESSAGE << accumulate (count, count + SIZE2, 0) << endl
```

57

```
<< endl;
```

实验7进一步探讨了一般的通用型算法和具体的accumulate函数的多功能性。

实验7：更多关于通用型算法的知识 （所有实验都是可选的）

这一章的最后将从用户角度简单看一下容器。

2.3.8 数据结构和标准模板库

数据结构就是用户眼中的容器。

数据结构就是用户眼中的容器。除了数组，所有C++里的容器都是某些容器类的实例，因
58 此对数据结构的兴趣其实就是对容器类的兴趣。具体说来，容器类的用户可以：

1) 创建该类的一个实例。

2) 调用该类的**public**方法。

开发者通过提供字段和方法定义完成了这个描述，那么可以说开发者**实现**了数据结构。例如，如果关注Linked类的方法接口，那么可以将该类看作一个数据结构；但是如果关注具体字段的选择和方法定义，这就是考虑该数据结构的实现。

标准模板库包含了具有广泛应用价值的数据结构。在标准C++里，标准模板库只是一部分，是未指定数据结构的实现细节。编译器的编写者可以自由地提供满足给定容器类方法接口的任何实现。从第5章开始，将学习标准模板库的数据结构以及每个数据结构可能的实现。顺便说一下，几乎所有的容器类都是模板化而且包含一个关联迭代器类的，因此这些话题在后面的章节中是非常有用的。

总结

指针和动态变量的概念对深入理解C++是非常重要的。一个相关的概念是通用数组指针定理。

通用数组指针定理

通常情况下，对任意数组a和非负整数i，a[i]和*(a+i)是等价的。

容器类是一个类，它的每个对象，即该类的一个实例，都由多项的集合组成。**容器**对象，即容器类的一个实例，可以连续存储在一个数组或一个链式结构里。在链式结构存储中，每个项都存入一个称作**节点的结构**，它包含了一个指向另一节点的指针。

几乎和每个容器类都相关的是迭代器类。**迭代器**是允许用户在不违背数据抽象原理的前提下，循环通过一个容器的对象。大部分迭代器类拥有下面的运算符：

```
operator!=      //将一个迭代器和另一个迭代器进行比较
operator++      //将迭代器前进到容器的下一个位置
operator*       //返回迭代器位置上的项
```

而且大部分容器类有一个begin()方法和一个end()方法，begin()方法可以返回位于容器开始处的迭代器，end()方法返回位于恰好是容器末尾的下一个位置的迭代器。

Linked类是容器类的一个非常简单的示例。毫无疑问，Linked类有一个链式结构。

标准模板库的大部分都由各种容器类的方法接口组成。编译器的编写者可以在满足方法接口的前提下随意地实现这些容器类。 59

习题

2.1 a. 创建一个**int**类型的Linked容器intList。向intList中插入五个整数。

b. 创建itr，迭代通过一个**int**类型的Linked容器。

c. 输出intList。

d. 创建两个**int**类型的Linked容器——oddList和evenList。

e. 对intList中的每一项，根据它是奇数或是偶数，分别将它的拷贝插入到oddList或是evenList里。

f. 输出oddList和evenList。

2.2 a. 创建employeeList，它是Employee对象的一个Linked容器。

b. 从输入中读入五个雇员，并将每个雇员插到employeeList的前面。读入第6个雇员，他的姓名是“**ZZZ**”，薪水总额是$100 000.00。

c. 从employeeList中输出超过第六个雇员薪水总额的每个雇员的姓名和薪水总额。

提示 使用迭代器。

2.3 用一个数组实现原始的（三方法）Linked类。开始时，数组的缺省容量是100，每当溢出当前容量时就将容量加倍。

提示 除了一个数组字段之外，还要加入一个front字段，它保存前一个元素的下标。每插入一个元素就为front加1。从什么意义上说，这种方式比从下标0插入一个新的元素好？

2.4 解释下列push_front方法定义错误的原因：

```
void push_front (const T& newItem)
{
      head = new Node;
      head -> item = newItem;
      head -> next = head;
      length++;
} // 方法 push_front
```

2.5 从本章或实验4到实验6中举出两个关于Linked类方法的例子，它们不使用Iterator对象循环通过容器。 60

2.6 在Linked类中，嵌入的Iterator类没有自减运算符——**operator- -(int)**。实际上，使用Iterator的当前设计和实现，不可能定义这样的运算符，解释原因。修改Iterator类，使它可以定义后减运算符。 61

编程项目2.1：扩展Linked类

在实验6中，通过声明和定义**operator=**扩充了Linked类。这里是另外一些方法的方法

接口：

```
//后置条件：如果这个Linked对象中没有项就返回真。否则，返回假。
bool empty() const;

//后置条件：如果这个Linked对象和otherLinked包含了相同顺序的相同项就
//返回真，否则返回假。
bool operator==(const Linked& otherLinked) const;

//后置条件：如果这个Linked对象和otherLinked没有包含相同顺序的相同项
//就返回真，否则返回假。
bool operator!=(const Linked& otherLinked) const;
```

1) 定义这些方法。

2) 使用一个main函数，它定义并操作两个**int**类型的Linked对象，以此来测试你的定义：

```
Linked<int> intList1,
            intList2;
```

测试应包括对第一部分里定义的每个方法的若干次调用；对每个返回bool值的方法，应包括一个返回值为真的调用和一个返回值为假的调用。还需要足够的输出来证明方法的正确性。

3) 重新用main函数测试定义，这里定义并操作两个Employee对象的Linked对象（需要重载Employee类的**operator==**）：

```
Linked<Employee> empList1,
                 empList2;
```

从键盘读入直到到达结束标记，这样来创建这两个Linked对象。多次调用第一部分里的每个方法。

62

第3章　软件工程简介

今天的计算机比起我们出生年代的计算机功能强大了很多。如果用芯片上晶体管的数量作为衡量计算机能力的标准，那么自1967年以来，计算机的能力几乎每18个月就增强一倍。这个惊人的统计就是著名的摩尔定理，是Intel公司的主席——Gordon Moore于1965年提出的。由于硬件能力的稳步增长，计算机可以在相对短的时间内解决非常复杂的问题。但是相对大型的程序开发则需要系统的方法。一般，开发一个10 000行的程序比开发一个5000行的程序不止难两倍。

目标

1) 理解软件开发生命周期的四个阶段。
2) 使用统一建模语言开发依赖关系图。
3) 创建方法、类和项目的测试用例。
4) 进行方法的大O分析和运行时间分析。

3.1　软件开发生命周期

为了使编程者能够应付大型程序的复杂性，出现了软件工程这门学科。**软件工程**是原理、技术和工具在软件生产上的应用。一些相关的概念起源于数学（形式化的大O分析），一些起源于物理科学（科学方法），还有一些起源于工程（项目生命周期）。大部分概念将在本章进行介绍，另外在实验以及后续章节中也会进行阐述。

即将学习的模型称作**软件开发生命周期**，它是构成一个编程项目的四个顺序阶段。其中一些阶段还可以分成若干子阶段。这里先给出一个大体的描述，按照时间先后这几个阶段为：

1) *问题分析*：仔细地阐明将要解决的问题。

2) *程序设计*：决定解决问题需要的类，它们是怎样建立关系的，以及在没有现成可用类时，确定类的方法接口和字段。

3) *程序实现*：为那些不是现成的类定义方法，然后验证、分析并集成类。

4) *程序维护*：对前面阶段的成果进行修改。

生命周期很少是顺序的过程。更确切地说，它是**迭代的**：当处在后面的阶段时，常常需要重做前面阶段的部分或全部工作。

3.2　问题分析

假设现在给出了问题的描述。这个描述可能很简单，甚至有一些模糊不清。但是在构造一个程序解决问题之前，必须清晰地理解该问题。问题分析阶段的目标就是要清楚地了解到需要做什么。这里故意省略了如何解决问题。通过这样的方式实践了广义的数据抽象原理，这个广义原理就是著名的**抽象原理**：

抽象原理

*当试图解决问题时，应当把**做什么**和**如何**做区分开来。*

把做什么从如何做中抽象出来，可以避免陷入一些细节的困扰，这些细节可能是后面的
64 阶段或者是由其他人所解决的问题。

问题分析阶段的大部分工作就是提供**功能规格说明**：根据输入和输出，用详细、明确的语句描述了程序应该做什么。规格说明应该回答下面这样的问题：

1) 输入的格式是什么样的？输入值的类型和范围是什么？

2) 输出的格式是什么样的？输出值的类型和范围是什么？

3) 即将执行什么样的任务？

4) 程序会不会是交互式的，也就是输入能否响应输出？或者在程序运行前是否创建输入文件？如果程序是交互的，那么输入结束的标志是什么？

5) 怎样处理输入错误，即，将执行多少输入编辑？交互程序的一个优点是在输入错误时能输出一个错误消息，因此用户可以立即更正错误。例如，下面的规格说明可以应用在输入一个年份上：

年份应当是1800到2200之间的整数（包括两端的数）。年份的输入应当响应提示“Please enter the year: ”。

a. 如果输入的值不是一个四位的整数，将输出错误消息“Error —— the year entered is not a four-digit integer.”

b. 如果输入的值是一个四位整数，但是不在1800~2200的范围内，输出的错误消息是“Error —— the year entered is not in the range 1800—2200.”

c. 每次输入一个错误的年份值时，应当输出相应的错误消息和提示。

规格说明不接受所有的错误输入吗？从程序开发者的观点来看，安全的回答是“Yes”。如果程序失败了，不管是意外的终止或是生成了错误的输出，编程者总是最大的嫌疑对象。编程者喜欢构造能够经受不可容忍的箭头符号之类的输入的程序。也就是说，编程者宁愿构造**健壮的程序**：在无效输入时程序不会意外终止。一个健壮的程序允许用户从输入错误中恢复：当出现错误的输入时，通报用户错误信息和提示进行正确的输入——通常是通过输出信息实现。

但是关于输入编辑的最终决定取决于客户，他是程序的购买者。在某些情况下，比如国防和病人的监护，意外的终止或错误的输出将酿成重大的灾难。在商务环境里，基于错误输出做出的决定也将产生损失惨重的后果。但是通常情况下，一些输入错误可以安全地忽略。例如，假设要为一个医院的病人制作账单。客户（医院管理人员）可能察觉费用过于昂贵了，这时可以根据编程者的时间，检查病人的医院记录的每个字段进行更正。

65

系统测试

根据给出的规格说明，创建样本输入值并手工求出相应的输出。在这个阶段，样本输入和样本输出的主要目的是确认对问题的理解以及输入输出格式。稍后，在书写程序之后，它们还可以作为测试用例，以对程序运行符合规格说明增强信心。比较好的是在书写程序之前而不是之后生成这些系统测试，否则程序书写的方式将不知不觉地受到测试的影响。

举一个简单的例子，对计算测验成绩平均值的程序进行下面的系统测试。样本输入用粗

体显式，以区分输入和输出。

系统测试 这个程序计算一系列测验成绩的平均值。每个成绩必须是0和100之间的整数，包括0和100。-1用来作为结束标志。

Please enter a test score:**80**

Please enter a test score:**90**

Please enter a test score:**700**

Error:The score must be an integer between 0 and 100, inclusive.

Please enter a test score:**7o**

Error:The score must be an integer between 0 and 100,inclusive.

Please enter a test score:**70**

Please enter a test score:**-1**

The mean is 80.0.

开发全面的系统测试是一项艰难、费时的工作。如果疏漏了一个临界的测试，程序可能会包含错误，根据摩尔定理，这个错误可能在最不适宜、最昂贵的时刻显露出来。

开发规格说明和系统测试的人是**系统分析员**。系统分析员有几个职责。首先，系统分析员必须理解终端用户（即最终运行程序的人）的需要。其次，系统分析员和客户必须在将要解决的问题上形成共识。最后，系统分析员必须能够为编程者提供一个明确、详细的问题描述。

所有的这些交流都可以利用文档进行。明确的文档，像功能规格说明，趋向于排除后面的分歧，像所说过的观点和谁赞成这个观点。文档提供了项目中除程序源代码之外的惟一可见的证据。在问题解决过程的后面阶段还会看到其他类型的文档。

创建规格说明和系统测试之后，下一个阶段是设计解决问题的程序。 66

3.3 程序设计

在这个阶段要决定需要什么样的类来解决问题，以及这些类之间是如何建立联系的。不夸张地说，编程者在这个阶段花费的努力将最大程度地影响整体项目的成功。一个公司可以有数千个可用类，如果其中一个类适用于项目，而开发者没有发现这个类，那么可能会浪费数百个小时来重新创建这个类。用几小时浏览软件库，就可以找到准时、低预算项目和延期、超预算项目之间的差别。不要做重复的工作!

3.3.1 方法接口和字段

使用那些已在应用的类是简单而令人高兴的。在剩余章中遇到的大多数项目中，标准模板库提供了一些必要的类。事实上，对所有的项目而言，至少需要创建一个类。对每个这样的类，先列举出类的职责：即类必须提供给用户的服务。将这些职责精炼成方法接口。还需要决定类的字段，但是将方法定义推迟到程序实现阶段。换句话说，在设计时开发头文件，在实现阶段开发源文件。

例如，在第1章的最高薪水雇员的问题上，指出了Employee类的下列职责：

把一个雇员的姓名初始化为空，薪水总额初始化成0.00。

读入一个雇员的姓名和薪水总额。

判断是否到达了结束标志。

判断一个雇员的薪水总额是否高于另一些雇员的薪水总额。

获取一个雇员的姓名和薪水总额的拷贝。

输出一个雇员的姓名和薪水总额。

然后将这些职责精炼成方法接口：

```
//后置条件：这个Employee的姓名被设置成一个空字符串，薪水总额被设置
//          成0.00。
Employee();

//后置条件：读入这个Employee的姓名和薪水总额。
void readInto();

//后置条件：如果这个Employee是结束标志就返回真。否则，返回假。
bool isSentinel() const;

//后置条件：如果这个Employee的薪水总额比otherEmployee的高就返回真。
//否则，返回假。
bool makesMoreThan(const Employee& otherEmployee) const;

//后置条件：这个Employee包含了otherEmployee的一个拷贝。
void getCopyOf(const Employee& otherEmployee);

//后置条件：输出这个Employee的姓名和薪水总额。
void printOut() const;
```

67

方法的开发者和使用者之间暗含了一种合约。

一个方法的前置条件和后置条件构成了方法的开发者和使用方法的编程者之间的**合约**。如果用户在调用方法之前确认满足了前置条件，那么开发者就能保证方法将最终结束并且结束时后置条件为真。如果前置条件为假并且用户仍不管不顾地调用了方法，开发者就没有责任保证结果：错误的答案，程序崩溃，失败……

Employee类有两个字段：

```
string name;
double grossPay;
```

需要的另一个类是Company，它的职责是寻找薪水最高的雇员，并输出这个雇员的姓名和薪水总额。方法接口是

```
//后置条件：找到这个Company里薪水最高的雇员。
void findBestPaid();

//后置条件：输出这个Company中薪水最高的雇员的姓名和薪水总额。
void printBestPaid();
```

在Company类中，惟一的字段是：

```
Employee bestPaid;
```

方法接口是文档化设计决策时方法层次上的工具。3.3.2节介绍类层次上的文档工具。

3.3.2 依赖关系图

类的关系文档的重要工具是依赖关系图。**依赖关系图**是一个表格，它体现了项目中类之间、字段之间和调用对象之间的依赖关系。说明这些关系的符号来自**统一建模语言**，它是一 68
个工业标准化语言，集中了当前软件工程中处理系统建模的实践。例如，如果类A是类B的一个子类，就从A到B画一个实心的箭头。图3-1包含了最高计时薪水雇员的问题设计的部分依赖关系图。

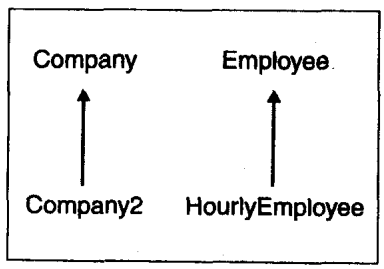


图3-1 最高计时薪水雇员的问题设计的部分依赖关系图

类的对象字段和该类的调用对象之间的依赖关系要复杂些。假设有下面的代码:

```
class X
{
    Y y;
    ...
}; // 类X
;
```

这里要考虑两种情况:

1) 复合。当回收（或者说重新分配）X类型的对象的空间时，也同样回收对象y的空间。换句话说，y的存在依赖于X的对象。

2) 聚合。当回收X类型的对象的空间时，不回收对象y的空间。换句话说，y的存在不依赖于X的对象。

例如，在Employee类里，只有调用Employee的对象存在时，name对象才存在。回想第2章中，如果一个类没有显式的析构器，就由编译器给出一个隐式析构器。这个隐式析构器只是（显式或隐式地）为类的每个字段调用析构器。因此当一个Employee对象超出作用域时，它的name字段的空间就被回收。即，name对象依赖于调用Employee的对象，这就是复合情况的例子。 69

为了描述统一建模语言中的复合，从封装类向字段画一个箭头，箭头的开端是一个实心的菱形。图3-2说明了name对象对Employee类的依赖性和bestPaid对Company类的依赖性（也是复合关系）。

当封装类的一个方法返回了指向字段对象的指针时就发生聚合。例如，使用下面的代码:

```
class X
{
    protected:
        Y y;
```

```
    public:
        Y* sendIt( )
        {
            ...
            return &y;
        } // 方法sendIt
    ...
}; // 类X
```

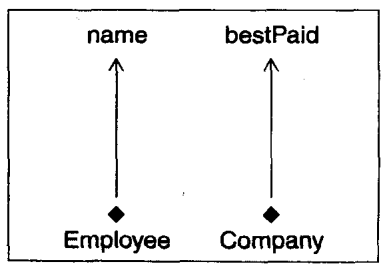


图3-2 统一建模语言中**复合**的依赖关系图。name对象依赖于调用的Employee对象，bestPaid对象依赖于调用的Company对象

在这种情况下，当回收调用的X对象时并不希望回收y的空间，因为sendIt返回了一个指向y的指针。当调用的X对象的存储空间被回收时，那个返回的指针可能仍存在。为了保证回收X的空间时还能够访问到y对象，封装类必须有一个显式析构器，它不回收y的空间。

在统一建模语言里，聚合用封装类指向字段对象的箭头描述；箭头的开端是一个空心的菱形。图3-3说明了刚才讨论的例子中的聚合。

70

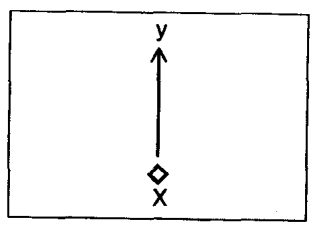


图3-3 对统一建模语言中**聚合**的说明。当调用的对象X的存储空间被回收时，对象y的空间不会被回收

在C++中，由开发者决定是应用复合还是应用聚合（例如类X）。这个决定确定了类是否应该有一个显式的析构器，如果需要，那么写出它的定义。

一旦完成了设计，就可以开始实现阶段的工作。3.4节讨论了这个阶段。程序设计阶段的最后部分——确定每个类的字段——有时被看作是程序实现阶段的一部分。这个区别并没有什么实际意义，因为从事设计和实现的通常是同一个开发小组。

3.4 程序实现

在这个阶段，首先，通过提供方法的定义，完成每个新类的定义。然后判断类的每个方法的正确性和效率。当我们很有把握相信一个方法在局部运行很好时，就可以考虑它和类的

其他方法的适应程度以及该类和项目的其他类的适应程度。之所以先考虑方法的正确性，是因为一个不正确的方法是毫无价值的，而一个效率低的方法可能是可以使用的。

3.4.1节讨论如何能对方法的正确性有信心。

3.4.1 方法验证

想增强对方法正确性的信心，最常用的技术是**验证**方法，也就是说，用很多字段及参数（如果方法包含任何输入语句的话，还有输入）的样本值测试方法。然后可以比较实际结果和方法后置条件的预期结果。

例如，假设想测试第1章中描述的Date类的next()方法。应当特别谨慎地确保方法在**边界值**，比如一个月的最后一天、一年的最后一个月等处运行正确。如表3-1所示，这里给出了一些测试数据:

71

表3-1 测试数据

调用对象的样本值			next()的预期结果		
日	月	年	日	月	年
17	11	2003	18	11	2003
30	11	2003	1	12	2003
31	12	2003	1	1	2004
28	2	2000	29	2	2000
28	2	2003	1	3	2003
28	2	2004	29	2	2004
28	2	2100	1	3	2100

表的最后四行反映了地球围绕太阳公转近似为365.2425天这个事实。**闰年**是指可以被4整除，且不能被100整除，或者能被400整除的年份。

如果next()方法通过了这些测试，就会增强对这个方法正确性的信心。但是还不能肯定方法的正确性，因为并没有尝试所有可能的测试。通常，运行所有可能的测试几乎是不可能的，因此可以只根据测试来推断正确性。正如E.W.Dijkstra曾说的，*测试可以揭示错误的存在但不能揭示错误的不存在*。

测试的另一个问题是测试者——构造测试数据的人——的客观性。编程者往往都以欣赏甚至热爱的眼光来看待自己的工作（例如“一个杰作”，“一个永远美丽、令人高兴的东西”，“世界的第八大奇迹”）。正因如此，编程者不适合测试他们自己的方法，因为测试的目的是揭露错误。实际上构造测试数据的人应当希望方法不能通过测试。在一个班级中，教师是比较满足这个标准的人!

如果一个类只有一个方法，可以设立一个测试主体来孤立地处理这个方法。更常见的情况是一个类有多个方法，需要对它们进行一致的测试。例如，需要测试方法m1()之后再测试方法m2()，还要先测试m2()再测试m1()。**驱动器**是一个程序，创建它就是为了专门测试一个类中的方法。成品强度驱动器可以更广阔地测试方法，它比普通的、特别是测试单个方法的驱动器要复杂得多。关于驱动器的知识将在实验8中继续讨论。

实验8：驱动器 （所有实验都是可选的）

正如类通常不止有一个方法一样，一个项目往往也不止有一个类。对一个多类的项目，应当先测试哪个类？在面向对象的环境中，常用的是自底向上的测试。使用**自底向上测试**，一个项目的低层次的类（就是可以被其他类使用但不使用其他类的类）首先被测试，然后再结
72 合高层次的类，依次类推。在项目的每个类都满足这个测试之后，可以运行**系统测试**，也就是将项目作为一个整体进行测试。前几个系统测试的输入作为问题分析阶段的一部分来提供。如果项目通过了这些测试，就进行另外的测试，直到确信项目是正确的，即它满足规格说明。

测试的目的是检测程序中的错误（或者是增强对程序中不存在错误的信心）。当测试显示出程序中存在错误，必须马上判断是什么导致的错误。这可能是一些严谨的探测工作所必须的。探测的目的是修正。整个验证阶段——测试、探测和修正——是迭代的。一旦修正了一个错误，应当重新开始测试，因为这个“修正”可能产生了新的错误。

3.4.2 正确性实现的可行性

正如本章开头所述，近年来的发展趋势是软件功能日益强大。（近期有个广告将大型程序定义为“有超过百万行代码”的程序。）但是在开发中一直缺少资深的系统分析员和编程员，而且在项目的管理上常常表现得急燥冒进。因此，目前“完成”的系统中都不可避免地存在错误。

从专业软件人员的观点来看，这种情况是难以忍受的。那么能够做什么呢？首先，也是一个主要的转变，就是管理必须对软件持有长远的眼光。正在为当前项目开发的类还不确定能否直接或通过继承被重新使用，因此在保证这些类的正确性上花费的额外的时间精力实际上是对未来的一种投资。而且，软件开发团队的每个成员都必须有质量的压力。系统分析员必须努力工作，以明确地指明将要做什么。程序员需要在正确性方面继续努力。整体的目标是创建一个环境来抵制错误，因此，将来软件的担保将像硬件的担保一样通用。

从管理的观点来看，完美总是和利润矛盾的。如果开发团队对系统正确性有99%的自信，那么这个系统可能就发售了，然后利润滚滚而来。要是延迟到有100%的自信才发售，那么将花费非常多的额外的时间，项目的利润也会急剧下降。当然发售一个有很多错误的系统会破坏公司的信用，因此必须在两者之间取得平衡。在面向对象的环境里，与其说是正确性和利润的矛盾，不如说是短期利润和长期利润的矛盾。

73 现在已经看到了方法、类和项目的验证，下面将注意力转移到它们的效率的评估上。

3.4.3 方法效率评估

方法的正确性只是依赖于方法是否做了预想的工作。但是一个方法的效率更大程度上依赖于方法是如何定义的。怎样衡量效率？可以在这个特别针对该任务创建的程序中反复地编译和执行方法。但那样的分析将依赖于编译器、操作系统以及所使用的计算机。在这个阶段，希望有一个更抽象的分析，它可以直接调查方法的定义。那么现在的问题就是如何通过方法定义估算一个方法需要的运行时间？

可以使用方法的跟踪中运行的语句数量作为衡量这个方法需要的运行时间的标准。这个标准可以表示成问题“大小”的函数。例如，对一个排序问题而言，就是被排序的值的数量。一般说来，一个有n个输入记录的问题称作是“大小为n的……”。

给定大小为n的某个问题的方法，令worstTime(n)是方法跟踪中执行的语句的最大数量（遍及所有可能的参数和输入数值）。有时我们也会对方法在平均情况下的性能感兴趣。定义averageTime(n)为方法跟踪中执行的语句的平均数量。这个平均接收了方法的所有调用，而且假设n个问题值的每种可能的安排都是相似的。对某些应用，最后这个假设是不现实的，因此averageTime(n)可能不是很准确。

有时候，特别第4章和第12章，还要估算方法的存储空间需求。为此，把worstSpace(n)作为方法跟踪中访问的变量的最大数量，averageSpace(n)是方法跟踪中访问的变量的平均数量。

3.4.4 大O表示法

人们并不需要精确地计算worstTime(n)和averageTime(n)，或者worstSpace(n)和averageSpace(n)，因为它们只是相应方法的时间、空间需要的近似值。这里用**大O**表示法近似这些函数。因为是单独地看这个方法，所以这个“近似的近似”能很好地说明方法将有多快的速度。

大O表示法的基本观点是，我们经常希望确定一个函数行为的**上界**，也就是确定函数可能运行得多么糟。例如，假设给定一个函数f。如果某些函数g不精确地说是f的一个上界，那么就称f是g的大O。把“不精确地说”换成“详细解释”，就得到如下定义：

74

> g是一个函数，它有非负整数变元，并对所有的变元返回一个非负值。将O(g)定义成函数f的集合，使得对某些非负常数C和K，满足
>
> 对所有的$n \geqslant K$，$f(n) \leqslant Cg(n)$

如果f是在$O(g)$中，就称f是“g的O”或者“f是g量级的”。

大O表示法的主要思想是：如果f是$O(g)$，那么最终f就限制在g的常量倍数上，因此可以使用$O(g)$作为函数f的估算上界。

通过符号的标准“妄用”，通常可以将函数和它所计算的数值关联起来。例如，令g是下面定义的函数：

$g(n)=n^3 \qquad n=0,1,2,...$

那么可以将$O(g)$替换成$O(n^3)$。

例3.1

令f是如下定义的函数：

$f(n) = n(n+3)+4 \qquad n=0,1,2,...$

证明f是$O(n^2)$。

■ 解

必须找到非负常数C和K，使得对所有的$n \geqslant K$，$f(n) \leqslant Cn^2$。首先重新书写函数定义：

$f(n)=n^2+3n+4 \qquad n=0,1,2,...$

然后说明，对$n \geqslant$某些非负整数，这个定义中的每一项都小于等于常数倍的n^2。马上可知：

$$n^2 \leqslant 1n^2 \qquad n \geqslant 0$$
$$3n \leqslant 3n^2 \qquad n \geqslant 0$$
$$4 \leqslant 4n^2 \qquad n \geqslant 1$$

因此，对任意$n \geqslant 1$，

$f(n) \leqslant n^2 + 3n^2 + 4n^2 = 8n^2$

也就是说，当C=8和K=1时，对所有的$n \geqslant K$，都有$f(n) \leqslant Cn^2$。这说明f是$O(n^2)$的。

75 通常情况下，如果f是一个多项式形式：

$$a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$$

那么可以通过选择K=1，C=$|a_k|+|a_{k-1}|+\ldots+|a_1|+|a_0|$，进行如例3.1所示的计算，得到$f$是$O(n^k)$的。例3.2说明在求一个函数的级时可以忽略对数的底数。

例3.2

令a和b是正的常数。证明如果f是$O(\log_a n)$，那么f也是$O(\log_b n)$。

■ 解

假设f是$O(\log_a n)$，那么存在非负常数C和K，使得对所有的$n \geqslant K$，

$$f(n) \leqslant C\log_a n$$

根据对数的基本属性（参见附录1），

$\log_a n = (\log_a b)\ (\log_b n)$ $\qquad n>0$

令$C_1 = C\log_a b$，

那么对所有的$n \geqslant K$，有

$$f(n) \leqslant C\log_a n = C \quad \log_a b \quad \log_b n = C_1 \log_b n$$

因此f就是$O(\log_b n)$。

注意大O表示法只是给出了一个函数的上界。例如，如果f是$O(n^2)$，那么f也是$O(n^2+5n+2)$、$O(n^3)$和$O(n^{10}+3)$。应当尽可能地从这个量级层次中选择最小的元素，最常用的量级层次如图3-4所示。称$O(g)$是f的**最小上界**意味着f是$O(g)$，而且对任意函数h，如果f是$O(h)$，那么$O(g) \subset O(h)$。

$$O(1) \subset O(\log n) \subset O(n^{1/2}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \cdots \subset O(2^n) \cdots$$

图3-4 量级层次中的一些元素，符号“$\subset$”表示“包含于”。例如，每个$O(1)$的函数也是在$O(\log n)$中的

例如，对n=0,1,2,...，如果$f(n)$=n+7，那么最好的说法是f是$O(n)$——即使f也是$O(n\log n)$和$O(n^3)$。图3-5显示了另外一些函数范例以及它们对应的量级。

不同量级的函数结果是不同的，下面给出具体的例子。假设$n=10^6$，那么

$$\log_2 n \approx 20$$

$$n = 10^6$$

$$n\log_2 n \approx 20 \times 10^6$$

$$n^2 = 10^{12}$$

注意$\log_2 n$与n之间以及$n\log_2 n$与n^2之间的巨大差异。在第8章中，我们将研究数据结构——折半查找树，其插入、删除和查找方法的averageTime(n)是$O(\log n)$，但这些方法的worstTime(n)是$O(n)$。同样，在第12章，通过对比$n\log_2 n$与n^2，说明了简单排序与快速排序的不同，前者的averageTime(n)为$O(n^2)$，后者的averageTime(n)为$O(n\log n)$。

3.4.5节表明，在大O表示法的帮助下，近似worstTime(n)或者averageTime(n)是很容易的。

量级	函数范例
$O(1)$	$f(n) = 3000$
$O(\log n)$	$f(n) = [n \log_2(n + 1) + 2]/(n + 1)$
$O(n)$	$f(n) = 5 \log_2 n + n$
$O(n \log n)$	$f(n) = \log_2 n^n$ (See Appendix 1)
$O(n^2)$	$f(n) = n(n + 1)/2$

图3-5 量级层次中的一些函数范例

3.4.5 快速获取大O估算

使用大O表示法可快速粗略地估算worstTime(*n*)和averageTime(*n*)的最小上界。

通过估算方法中的循环迭代次数，往往可以马上得到worstTime(*n*)在量级层次中的最小上界。令*S*代表任意语句序列，它的执行不包含循环语句，循环语句的迭代次数是依赖于*n*的。下面的方法框架提供了在量级层次中求worstTime(*n*)的最小上界的范例。

1) worstTime(*n*)是$O(1)$:

S

注意*S*的执行可能是百万条语句的执行！例如:

```
double sum = 0;
for (int i = 0; i < 10000000; i++)
    sum += sqrt (i);
```

worstTime(*n*)是$O(1)$的原因是因为循环迭代的数量是常数，所以不依赖于*n*。实际上在这样的情况下，通常会绕过大O表示法，称worstTime(*n*)是“常数”。

2) worstTime(*n*)是$O(\log n)$:

```
while (n > 1)
{
    n = n / 2;
    S
} // while
```

77

令*t*(*n*)是*S*执行的次数；那么*t*(*n*)等于*n*可以不断被2除直到*n*=1的次数。根据附录1中例A1.3，*t*(*n*)是$\leq \log_2 n$的最大的整数。也就是说，*t*(*n*)=floor($\log_2 n$)[⊖]。所以*t*(*n*)是$O(\log n)$，因此worstTime(*n*)也是$O(\log n)$。

这种反复把一个容器分裂成两个的现象将在第4章和第8~12章中反复出现。根据对分裂的观察：它标志着worstTime(*n*)是$O(\log n)$。

分裂法则

如果在每次循环迭代过程中，用一个大于1的常数除*n*，那么迭代的次数将是$O(\log n)$。

当$O(\log n)$是worstTime(*n*)的最小上界时，就说worstTime(*n*)是和*n*成“对数关系”的。

⊖ floor(*x*)返回小于等于*x*的最大整数。

3) worstTime(n)是$O(n)$:

```
S
for (i = 0; i < n; i++)
{
    S
} // for
S
```

worstTime(n)是$O(n)$的原因就是因为**for**循环执行了n次，而不管在每次**for**循环的迭代中执行了多少条语句。

当n是worstTime(n)的最小上界时，就说worstTime(n)是和n成“线性关系”的。

4) worstTime(n)是$O(n\log n)$:

```
for (i = 0; i < n; i++)
{
    m = n;
    while (m > 1)
    {
        m = m / 2;
        S

    } // while
} // for
```

78

for循环共执行了n次。在**for**循环的每次迭代中，又执行floor($\log_2 n$)次的**while**循环。所以worstTime(n)是$O(n\log n)$。

5) worstTime(n)是$O(n^2)$:

a.
```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
        S
    } // for j
```

S执行的次数是n^2。

b.
```
for (i = 0; i < n; i++)
    for (j = i; j < n; j++)
    {
        S
    } // for j
```

S执行的次数是

$$n+(n-1)+(n-2)+\cdots+3+2+1=\sum_{k=1}^{n}k$$

正如附录1中例A1.1所示，这个累加之和等于

$$n\ (n+1)/2$$

也就是$O(n^2)$。即worstTime(n)是$O(n^2)$。

c.
```
for (i = 0; i < n; i++)
{
    S
} // for i
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
        S
    } // for j
```

对第一段，worstTime(n)是$O(n)$，第二段的worstTime(n)是$O(n^2)$，因此，把这两段合起来，worstTime(n)就是$O(n^2)$。通常情况下，对序列

A

B

如果*A*的worstTime(n)是$O(f)$而且*B*的worstTime(n)是$O(g)$，那么序列*A*、*B*的worstTime(n)就是$O(f+g)$。 79

当n^2是worstTime(n)的最小上界时，就说worstTime(n)是和n成“平方关系”的。

我们喜欢尽可能用简单的语言（“常数”，“对数”，“线性”，“平方”）进行描述。但是在3.4.6节中将会看到，仍然有很多场合，所有可以给出的就是一些上界的大O估算，而不必是量级层次上的最小上界。

假设有一个方法，它的worstTime(n)和n成线性关系。那么对某些常数C，可以写成:

$$\text{worstTime}(n) \approx Cn$$

问题的大小加倍，即n加倍会有什么影响呢?

$$\begin{aligned}\text{worstTime}(2n) &\approx C2n\\ &= 2Cn\\ &\approx 2\text{worstTime}(n)\end{aligned}$$

换句话说，如果n加倍，那么最坏时间花费的估算也会加倍。

同理，如果一个方法的worstTime(n)和n成平方关系，那么对某些常数C，可以写成:

$$\text{worstTime}(n) \approx Cn^2$$

那么

$$\begin{aligned}\text{worstTime}(2n) &\approx C(2n)^2\\ &= C4n^2\\ &= 4Cn^2\\ &\approx 4\text{worstTime}(n)\end{aligned}$$

换句话说，如果n加倍，那么最坏时间花费的估算将变成4倍。习题3.7、习题12.5和实验16以及实验27探究了这种关系的一些其他例子。

图3-6显示了量级层次中6种量级的相对增长速度。这个图说明了大O的差别将最终支配函数行为估算中的其他因素的原因。例如，即使当变元小于100 000时，$T_1(n)=n^2/100$比$T_2(n)=100n\log_2 n$小；但当n足够大时，T_1将比T_2大很多。

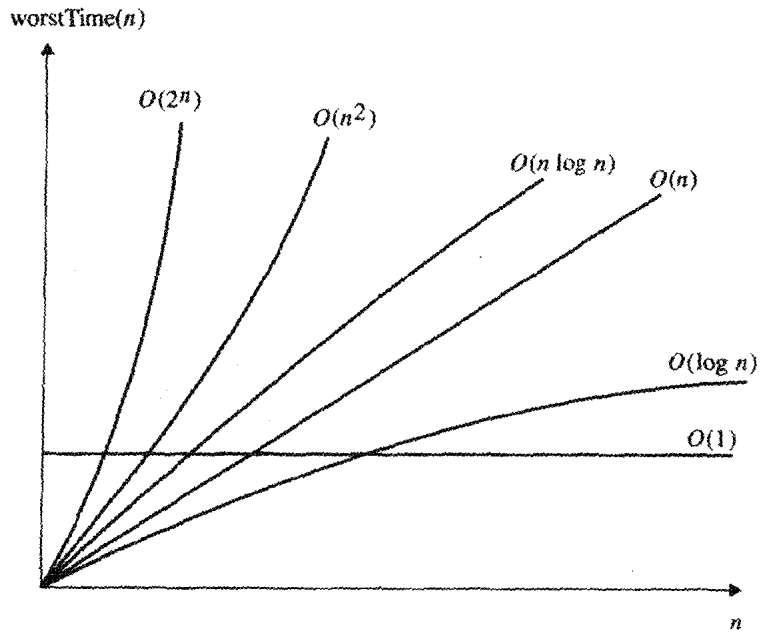


图3-6 几种函数量级的worstTime(n)示意图

多项式时间方法是这样一种方法：对某些正整数k，它的worstTime(n)是$O(n^k)$。例如，一个方法的worstTime(n)是$O(n^2)$，这是一个多项式时间方法。同样，worstTime(n)是$O(n\log n)$的方法也是一个多项式时间方法，因为$O(n\log n) \subset O(n^2)$。当试图开发一个方法去解决给定问题时，应尽可能地使用多项式时间方法；否则，对n的值很大的情况，方法的运行是很困难的。如图3-6所示的worstTime(n)是$O(2^n)$的方法由于该方法的worstTime(n)增长速度过快，以致于当n的值很大时这个方法是不可用的。这样的方法，它们不是多项式时间，这类方法统称为**幂时间**方法。**棘手问题**是指任何可以解决问题的方法都是幂时间方法的情况。例如，有一个需要输出2^n个数值的问题就很棘手。在第4章里将看到两个棘手问题的例子。实验29探讨了货郎担问题，解决这个问题的已知的方法都是幂时间方法。货郎担问题是否棘手还是个悬而未决的问题，因为也可能存在多项式时间方法用以解决这个问题。

80

如果只使用一个方法，那么averageTime(n)和worstTime(n)的优化，即优化运行时间是比较容易的。但是管理整个项目时，通常需要平衡考虑。3.4.6节探索了其他因素，像空间利用和项目最终期限之间的协调。

3.4.6 平衡折中

在3.4.5节中，看到了如何分析方法的运行时间需求。同样的大O表示法还可以估算方法的空间需求。理想状态下，应当能开发出既快又小的方法。但是在现实世界中这个目标很难实现。更多时候，在编程中会遇到下面的问题：

1) 程序的运行时间估算比性能规格说明上规定的可接受时间长。**性能规格说明**规定了整个或部分程序的时间和空间上界。

2) 程序的空间需求估算比性能规格说明上规定的可接受空间大。

3) 程序可能需要一种编程者不太熟悉的技术。这会给整个项目带来难以接受的延迟。

通常需要做出平衡：程序克服了某个问题可能又突出了另外两个问题。现实的编程总让人很难决断。仅仅使开发的程序能运行往往是不够的。适应这类的约束可以增强编程的灵活性，成为一个优秀的编程人员。

直到现在，我们还是把正确性和效率分开考虑。根据数据抽象原理，使用类的代码的正确性应该不依赖该类的实现细节。但是代码的效率更依赖于这些细节。换句话说，为了效率，类的开发者可以自由地选择任何字段和方法定义，提供正确的不依赖这些选择的方法。例如，假设一个类的开发者创建类的三个不同版本：

A. 正确的，效率低的，不允许用户访问字段。

B. 正确的，有点效率的，不允许用户访问字段。

C. 正确的，高效率的，允许用户访问字段。

大多数情况下，比较好的选择是B。选择C将违背数据抽象原理，因为使用C的程序的正确性依赖于C的字段。

通过在方法后置条件中加入性能规格说明，把效率估算和方法正确性结合起来。例如，第12章的通用型算法sort的一部分后置条件是

worstTime(n)是$O(n^2)$

那么如果方法的定义正确，它的最坏时间花费就不会超过n的平方。回想大O估算，它仅仅提供了**上界**。但是方法开发者可以在不违背合约的前提下，自由地改进最坏时间花费的上界。例如，开发者可能提供不同定义的sort方法，它的worstTime(n)是$O(n\log n)$。

下面是方法后置条件中三条关于大O估算的规格说明的约定：

1) 变量n指容器中项的数量。

2) 对很多方法，worstTime(n)是$O(1)$。如果不给出worstTime(n)的估算，可以假设worstTime(n)是$O(1)$。

3) 通常情况下，averageTime(n)和worstTime(n)有相同的大O估算。当它们不同时，将对两者都做详细的说明。

82

这里有必要强调的一点是，在所有阶段做的所有工作都有一个文档组件，例如，问题分析阶段的规格说明，程序设计阶段的方法接口和依赖关系图，以及程序实现阶段的大O分析。一般说来，每个阶段的正式文档可用于减少模糊并突出职责。

大O分析提供了估计方法效率的跨平台估算。3.4.7节探讨了处理效率的运行时间工具。

3.4.7 运行时间分析

以流逝的时间估算运行时间是很不精确的。

在前面已经看到大O表示法可以不依赖任何具体的计算环境估算方法的效率。在实践中，我们也希望可以估算某些确定环境下的效率。为什么要进行估算？首先，*在多道程序设计环境（比如Windows）中，很难判断单个任务将执行多长时间。*为什么？因为有很多东西是在后台运行的，像维护桌面时钟，循环等待直到单击鼠标，更新来自邮箱和浏览器的信息。在任何给定时刻，都会有很多这样的进程被Windows管理器控制运行。并且每个进程将获得一个几毫秒的时间片。一个任务执行后流逝的时间很少会是对任务工作时间的精确的测量。

寻求效率的精确测量的另一个问题是它可能耗费很长的时间：O(永远)。例如，假设比较两个排序方法，我们希望能确定每个方法排序一些容器所花费的平均时间。这个时间将很大

程度上依赖于所选项的具体顺序。因为n项共有$n!$种不同的顺序，生成所有可能的顺序，并对每个顺序运行方法并计算平均时间，这是不可能的。

取而代之的是，可以生成一个样本顺序，它是“没有特别顺序”的。“没有特别顺序”所对应的统计学概念是随机。可以使用排序随机样本的时间估算平均排序时间。

C++提供了time和rand函数来帮助用户解决计时问题。

time函数：C++的time函数采用一个指向**struct**对象的指针保存当前时间的信息（像小时、天和年）。当使用一个NULL变元进行调用时，返回值是**long**类型的，代表自1970年1月1日0点起所经过的时间（国际标准时间，以秒为单位）。

为了求出一个任务耗费的时间，可以计算任务代码运行之前和刚运行结束之后的时间，然后用“之后的”时间减去“之前的”时间。例如，进行运行时间测试的程序通常应包括下面的部分：

83

```
// 判断一个任务耗费的时间。
#include <iostream>
#include <string>
#include <time.h> // 声明time函数
int main ( )
{
    const string TIME_MESSAGE_1 = "The elapsed time was ";
    const string TIME_MESSAGE_2 = " seconds.";
    const string CLOSE_WINDOW _PROMPT =
        "Please press the Enter key to close this output window:";
    long start_time,
        finish_time,
        elapsed_time;
    start_time = time (NULL);
    // 运行任务:
    ...
    // 计算任务执行后流逝的时间:
    finish_time = time (NULL);
    elapsed_time = finish_time - start_time;
    cout << TIME_MESSAGE_1 << elapsed_time <<
            TIME_MESSAGE_2
        << endl;
    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );
    return 0;
} // main
```

以整个时间测量运行时间妨碍了精确性。但是正如曾提到过的，在多道程序环境下很难孤立一个任务的运行时间。来自其他因素的影响，像高速缓冲存储器的利用和早先程序运行中存在的磁盘缓冲区，都会导致即便是同一任务连续运行的运行时间上的极大不同。因此应

该把一个任务共用的时间（即执行任务后流逝的时间）看作是实际运行时间的估算——它甚至比大O分析还粗糙。

计时是实验9的主题。这个实验还增加了3.4.8节介绍的随机性的素材。

3.4.8 随机性

给出一系列数值，从这些数值中**随机**地选择一个数值，也就是每个数值都有相同的被选中的几率。这样选择的数值称作**随机数**，而返回随机数的函数称作**随机数生成器**。C++有一 84
个系统函数rand，它提供了一个随机数生成器。严格说来，rand函数是一个伪随机数生成器，因为这些数值是计算出的而不是真正随机的——它们是由函数求出的。不考虑如何计算时，这些数值看起来是随机的。在系统头文件stdlib.h里声明了rand函数。

rand函数没有参数，并返回一个0到RAND_MAX之间的伪随机数（stdlib.h中定义常量标识符RAND_MAX为0x7ff，这是十六进制形式，表示32 767）。

rand函数计算的数值依赖于给定的种子。种子是一个预声明的**unsigned int**（无符号整型），它的初始值是用srand函数设置的。srand函数接收一个**unsigned int**类型的变元并将种子设置成这个数。在任何srand调用前调用rand生成的序列是和用1作为种子传递调用srand生成的序列相同。每次调用rand函数时，就用种子的当前值求种子的下一个值。种子的这个新值确定了rand的返回值。

例如，假设两个程序有：

```
#include <stdlib.h> // 声明srand和rand函数

srand (100);
for (int i = 0; i < 5; i++)
      cout << rand ( ) << endl;
```

这两个程序的输出恰好是相同的[⊖]：

```
1862
11548
3973
4846
9095
```

当想要比较程序行为时，比如在后面的实验以及第4~14章中，这个重复性是很有帮助的。一般来说，重复性是科学方法的一个基本特点。

如果不想重复，开始时就使用time(NULL)作为变元调用函数srand。这暗示着除非在同一秒内运行两遍程序（这基本是不可能的），否则是绝不可能得到相同的伪随机数序列的。

例如，编写代码：

```
#include <stdlib.h> // 声明srand和rand函数

srand (time (NULL));
for (int i = 0; i < 5; i++)
      cout << rand ( ) << endl;
```

85

每次运行程序时，都会得到0和RAND_MAX之间的不同的5个随机整数。

⊖ 实际产生的序列依赖于计算平台。

实验9着重讲述了函数time和rand，以及它们在计时实验中的使用。

实验9：计时和随机性 （所有实验都是可选的）

在程序实现介绍中最后描述一下类型转换。类型转换的基本思想是改变表达式的求值，将表达式的类型临时解释成一个不同的类型。

3.4.9 类型转换

为了显式地改变一个表达式的类型，C++提供了类型转换。**类型转换**的组成是一对圆括号及其中的一个类型，后面跟着将被转换的表达式。例如，可以把一个表达式按如下方式从**long**转换成**float**：

```
long i = 3,
     j = 5;

cout << i / j << endl; // 输出将是0

cout << (float) i / j << endl; // 输出将是.6
```

类型转换运算符——圆括号——比除法运算符的优先级高，因此在除以j之前i就被转换成浮点型了。另一种表示法是将要类型转换的表达式放进圆括号：

```
cout<<float(i)/j<<endl;
```

由多个标识符组成的类型必须用圆括号括起来，由多个标识符或文字符号组成的表达式也必须用圆括号括起来。有时候类型和表达式都需要括号。例如，可能有：

```
(Node*)(head->next)
```

通常，使用类型转换时，表达式的数值的大小和类型的大小所占据的字节是相同的。这是因为通常类型转换并不修改值的实际位数，而只是解释这些位。但是对于数值，C++定义了不同数值类型的转换，因此即使它们大小不同也可以进行转换。例如，可以将**short**转换成**float**。在另一个例子中，我们将**int**转换成**char**：

```
int i=65;
cout<<(char)i<<endl;
```

86 这些将输出语句中i的类型转换成**char**，因此输出是

```
A
```

因为ASCII排序序列中第65个字符就是‘A’。

也可以将**char**转换成**int**。例如，假设有：

```
char new_char='D';
cout<<(int)new_char;
```

输出是68，因为‘D’是ASCII排序序列中第68个字符。

所有的指针都是相同类型的，一般是4个字节，因此它们之间的转换是合法的。在第4章探索**void**指针的奥秘时，应用这一点会有很好的效果。但转换一定要谨慎！如果将某个确定大小的表达式转换成小一些的表达式，就会丢失一些重要的信息。例如，如果将**long**转换成

int，或将**double**转换成**float**，就会丢失精度。

程序开发生命周期的最后阶段是程序的维护。

3.5 程序维护

程序实现阶段结束之后，该程序就可以给终端用户了。某些程序，称作**成品**程序，以几年作为运行周期。随着时间的流逝，每个程序几乎都不可避免地经历一些改变。**程序维护**指的是对已经部署的程序进行修改。这个维护工作可以由原先开发程序的分析人员、设计人员和编程人员完成，但更多的时候是由有新观点的新团队来完成。

软件维护和硬件维护有根本的不同，因为代码是不会变坏的。磁盘驱动器有一个平均失败时间，但是在延期使用之后**switch**语句也不会开始丢失case。那么软件维护需要做什么？维护团队有责任更正发现的错误，并避免在创建程序中出现错误造成将来的失败。不过大部分维护团队所做的工作都是去增强原系统。

增强一个已有的程序是一项具有挑战性的任务。如果用可维护性的观点开发源项目，像进行全面系统测试，对类高度模块化，那么维护就容易得多了。

例如，考虑计税软件。税表是每年都会改变的，因此它们应该是一个单独的类。同样，可能会扩充纳税人的身份识别以允许数字签名，另外还有常用的数据：姓名，地址，社会保障号码等等。那么必须扩展包含这些字段的类，以增加额外的功能。因为纳税代码逐年越变越复杂（可能只是看上去如此），所以对模块化的要求也在相应增强。 87

程序维护决不是一个次要的工作。在某些公司中，编程者平均有超过50%的时间用于程序维护。这强调了软件开发生命周期的所有阶段中对文档的需要。没有文档，即使是维护自己的程序也是非常耗费时间且困难重重的。

总结

本章介绍了**软件工程**的基本概念，它是原理、技术和工具在软件生产上的应用。程序生产“年表”——软件开发生命周期——由四个阶段组成。每个阶段都有一个文档部分，而且整个过程通常是迭代而不是线性的。

在问题分析阶段，开发详细的规格说明和系统测试。

在程序设计阶段，按照需要设计新的类。为每个这样的类确定方法接口和字段。依赖关系图阐述了类之间的继承关系以及调用对象的对象字段的依赖性或独立性。

在程序实现阶段，为程序设计阶段引入的类定义方法。大O表示法可以快速地估算方法的时间-空间效率。项目验证是自底向上进行的。用一个驱动器测试低层次的类，然后使用这些类测试高层次的类。最后使用问题分析阶段开发的系统测试以及其他的测试来验证项目。运行时间分析经常使用time和random函数。

程序验证完毕之后，可以在程序维护阶段进行问题或程序的改造。

习题

3.1 创建一个方法，

```
void sample(int n);
```

使得它的worstTime(n)是$O(n)$，但worstTime(n)不和n成线性关系。

提示 $O(n)$提供了一个上界，但是“和n成线性关系”指的是最小上界。

3.2 研究下面的代码：

```
//令i是满足a[i]=item的0...n-1之间的最小的下标。
i=0;
while(a[i]!=item)
88          i++;
```

假设a是n元素数组并且在0...$n-1$之间至少有一个下标k，满足a[k]=item。寻找一个函数g使得worstTime(n)是$O(g)$，而且$O(g)$是worstTime(n)的最小上界。

3.3 研究下面的关于字符串数组a的代码：

```
// (按照增序) 排序a[0...n-1]:
for (i = 0; i < n-1; i++)
{
    // 排序a[0...i]并令a[0...i]<=a[i+1...n-1]:
    position = i;
    for (j = i+1; j < n; j++)
        if (a [j] < a [position])
            position = j;

    string temp = a [i];
    a [i] = a [position];
    a [position] = temp;

} // 外部for
```

a. 当i=0时，内部**for**循环进行$n-1$次迭代。当i=1时进行多少次迭代？当i=2时呢？

b. 求一个n的函数，当i取0到$n-2$之间的值时内部**for**循环的总迭代次数。

c. 找到一个函数g，使得外部**for**语句的worstTime(n)是$O(g)$，并且$O(g)$是worstTime(n)的最小上界。

3.4 对下面的每个函数f，当n=0,1,2,3,...时，寻找函数g使得$O(g)$是f的最小上界：

a. $f(n) = (2 + n)(3 + \log_2 n)$

b. $f(n) = 11 \log_2 n + n/2 - 3452$

c. $f(n) = 1 + 2 + 3 + \ldots + n$

d. $f(n) = n(3 + n) - 7n$

e. $f(n) = 7n + (n - 1)\log_{10}(n - 4)$

f. $f(n) = \log_2(n^2) + n$

g. $f(n) = \dfrac{(n + 1)\log_2(n + 1) - (n + 1) + 1}{n}$

h. $f(n) = n + n/2 + n/4 + n/8 + n/16 + \ldots$

3.5 在量级层次中，有...，$O(\log n) \subset O(n^{1/2})$，...，证明：对任意整数$n$>16，$\log_2 n < n^{1/2}$。

从微积分得到的提示 证明对所有的实数x>16，函数$\log_2 x$的斜率小于函数$x^{1/2}$的
89 斜率。由$\log_2(16)=16^{1/2}$，推断对所有实数x>16，$\log_2 x < x^{1/2}$。

3.6 对下列代码段，寻找一个函数g，使得$O(g)$是worstTime(n)的最小上界。在每段程序里，S代表了一些不依赖于n的循环语句。

a.
```
for (i = 0; i * i < n; i++)
        S
```

b.
```
for (i = 0; sqrt (i) < n; i++)
        S
```

c.
```
k = 1;
for (i = 0; i < n; i++)
        k *= 2;
for (i = 0; i < k; i++)
        S
```

提示 在每种情况里，2都是答案的一部分。

3.7 a. 假设有一个方法，它的worstTime(n)和n成线性关系。计算当n变成三倍时对最坏时间花费估算的影响。也就是说，根据worstTime(n)估算worstTime($3n$)。

b. 假设有一个方法，它的worstTime(n)和n成平方关系。计算当n变成三倍时对最坏时间花费估算的影响。也就是说，根据worstTime(n)估算worstTime($3n$)。

c. 假设有一个方法，它的worstTime(n)是常数。计算当n变成三倍时对最坏时间花费估算的影响。也就是说，根据worstTime(n)估算worstTime($3n$)。

3.8 针对下述问题开发功能规格说明：给出一系列测验成绩，求出平均成绩之下的成绩的数量。

3.9 为习题3.8描述的问题创建系统测试。

3.10 用一个Linked对象，设计并实现程序解决习题3.8和习题3.9描述的问题。

3.11 证明$O(n) = O(n+7)$。

提示 使用大O的定义。

3.12 下面的哪个表达式包含一个在1和6之间（包括1和6）的随机整数？

a. rand()%6

b. rand()%5+1

c. rand()%7

d. rand()%6+1

90

编程项目3.1：Linked类的进一步扩充

1) 扩充实验8的Linked类，添加一个pop_back方法。下面是方法接口：

```
//前置条件：这个Linked对象非空。
//后置条件：调用前在这个Linked对象后面的项已被删除。worstTime(n)是O(n)。
void pop_back();
```

2) 扩展实验8的LinkedDriver类以验证pop_back方法。

3) 假设从一个空的int类型的Linked容器开始：

```
Linked <int> intList;
```

比较调用n次push_fronts之后再调用n次pop_fronts需要的时间和调用n次push_backs之后再调用n次pop_backs需要的时间。选择足够大的n，使得差别至少是2秒。这些时间和你的大O分析一致吗？

91

第4章 递 归

初学者和有经验的编程人员之间的重要区别就在于对递归的理解。本章的目的是引导读者了解递归函数适用的场合，然后就可以看到递归的精巧以及它的强大功能，当然还有使用不当时的潜在危险。

标准模板库中的大部分常用实现都使用了递归：tree（红-黑树）类中的copy函数和sort算法。但是递归的价值远远不止这两个实例。例如，第7章中的stack类的一个应用，就是递归函数翻译成机器代码。以及在第8章中，大部分与二叉树相关的定义都是或者可以是递归的。越早接触递归，就越有可能掌握它所适用的环境并使用它!

目标

1) 总结适合用递归解决的问题的特点。
2) 比较递归和迭代函数的时间空间代价，以及它们的开发难易程度。
3) 通过执行结构框架跟踪一个递归函数的执行。
4) 掌握求解问题时使用的回溯策略。

4.1 简介

简单地说，当一个函数包含着对它自身的调用时就称这个函数是递归的[⊖]。这个描述令人产生一个可怕的想法，就是递归函数的执行将导致无穷多的递归调用。但是在正常情况下，这是不会发生的，调用最终可以停止。为了说明递归是如何实现的，以下给出了一个典型的递归函数：

```
if (最简单的情况)
    直接处理
else
    递归调用一个较简单的情况
```

这部分说明，适合采用递归处理的问题具有以下两个特点：

1) 问题的复杂情况可以简化成和它形式相同的较简单的情况。
2) 最简单的情况可以直接处理。

综上所述，如果读者熟悉数学归纳法（参见附录1），就会发现这两个特点恰好对应着归纳部分和基础部分。

随着示例的深入，不要被旧的编程思想禁锢。尽可能地尝试将每一个问题都转化成形式相同但较为简单的问题，递归地去考虑。

4.2 阶乘

给定一个正整数n，n的阶乘记作$n!$，它是所有1和n（包括1和n）之间的整数的乘积。例如，

⊖ 递归的正式定义参见4.8节。

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

和

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

另一种计算阶乘的方法如下:

$$4! = 4 \times 3!$$

这个公式没什么用处，除非能知道3! 是什么。但是可以根据小一点的数的阶乘来继续计算阶乘:

$$3! = 3 \times 2!$$
$$2! = 2 \times 1!$$

94 注意1! 可以直接计算出来: 它的值是1。现在再回过头来计算4! :

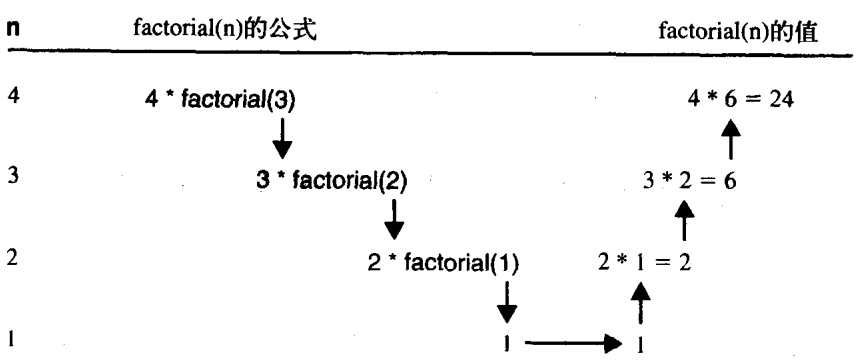
$$2! = 2 \times 1! = 2 \times 1 = 2$$
$$3! = 3 \times 2! = 3 \times 2 = 6$$
$$4! = 4 \times 3! = 4 \times 6 = 24$$

对$n>1$，我们将计算$n!$的问题简化成计算$(n-1)!$。当到达1! （也就是1）时停止简化。以上的这些观察产生了下面的函数; 为了能正常地结束[⊖]，将0! 定义成1。

```
// 前置条件: n>=0。
// 后置条件: 返回的数值是n!，也就是1和n（包括1和n）之间所有整数的乘积。
//           worstTime(n)是O(n)。
long factorial (int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial (n - 1);
} // factorial
```

在factorial中，有一个对factorial的调用，因此它是递归的。在本书网站提供的源代码中有这个函数的驱动程序。

下面的图表显示了当初始变元值为4时，factorial的跟踪执行情况:



⊖ 从n项中抽取k项的组合数是$n!/[k! \cdot (n-k)!]$。当$n=k$时，得到$n!/(n! \cdot 0!)$，因为0!=1，所以这个组合值是1。

函数的每次递归调用，形参n的值都会随之减1。但是在n=1即最后一个调用之后，就需要和前面的n的值相乘。例如，当n=4时，n*factorial(n-1)的计算必须延迟到对factorial(n-1)的调用结束之后。当最终值6（也就是factorial(3)）被返回时，n的值4就可用了。 95

这样，在调用factorial(n-1)时必须保存n的数值。而当factorial(n-1)调用结束之后又必须恢复这个值来计算n* factorial(n-1)。递归的好处就在于编程者不需要明确地处理这些存储和恢复；计算机会做这个工作。

factorial函数的前置条件是：n是一个非负整数。如果一个用户调用factorial(-1)将产生什么结果呢？因为n既不是0也不是1，所以执行**else**部分，它包含了对factorial(-2)的调用。在这个调用中，n仍然是既不为0也不为1，因此继续调用factorial(-3)，然后是factorial(-4)，factorial(-5)，等等。最后，所有这些被存储的n的拷贝将使得**堆栈**（即一块内存区域）溢出。这个现象称作**无穷递归**。

如前置条件所述，在上面的factorial函数里，如果变量值大于等于1就可以避免无穷递归。通常情况下，如果每个递归调用都可以向“最简单的情况”推进，那么就可以避免无穷递归。注意如果将

```
(n==0||n==1)
```

替换成

```
(n<=1)
```

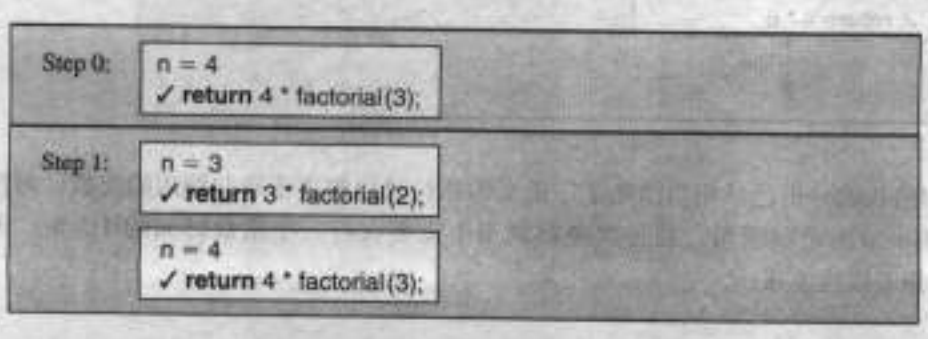
那么就算输入一个负数也不会出现无穷递归。这时将返回1。不过这个版本对用户来说更危险，因为即使变元违背前置条件也无法察觉。通常来说，得到一个错误的答案还不如没有答案。

执行结构框架

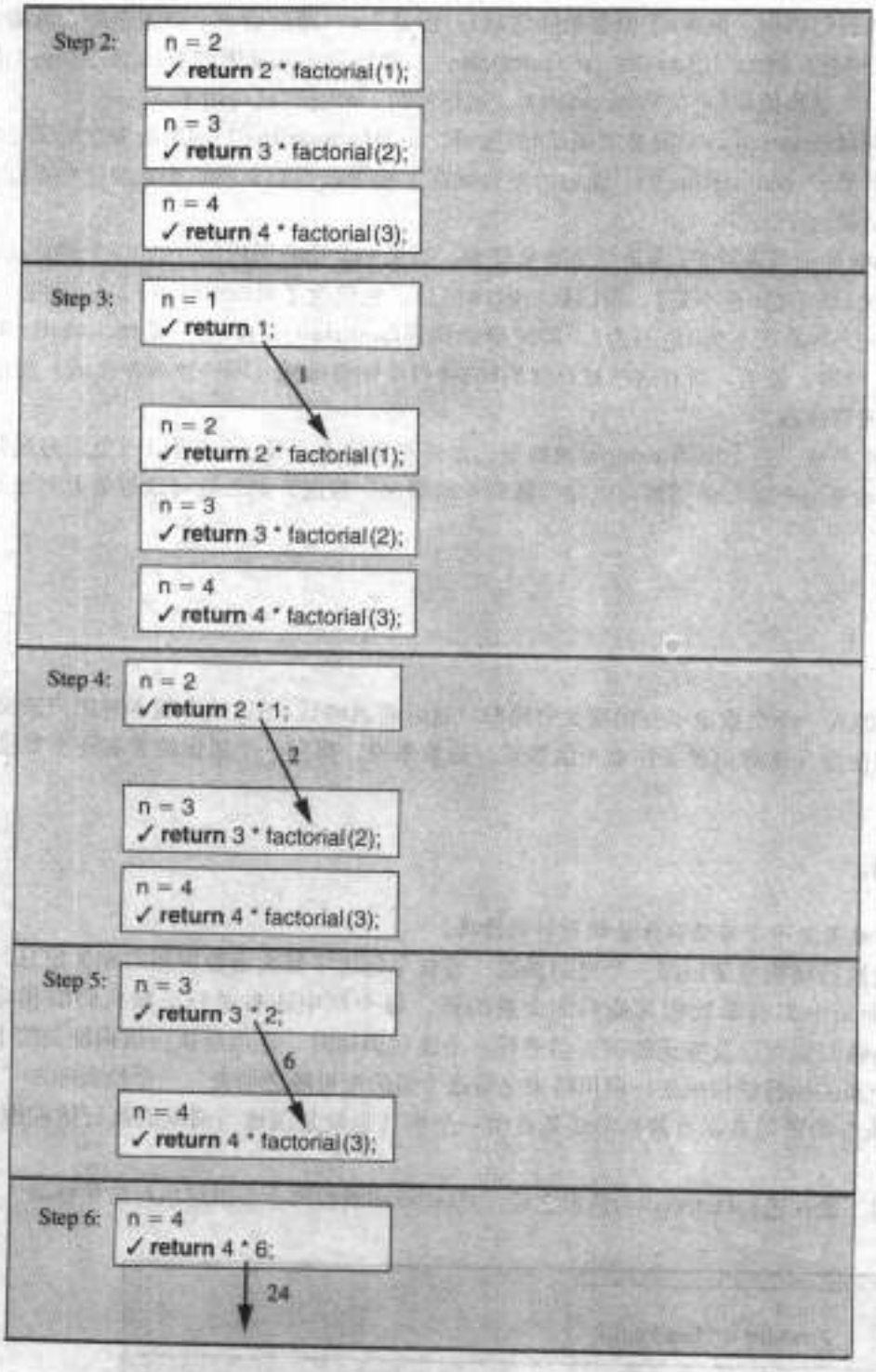
执行结构框架显示了当递归方法执行时的情况。

可以通过**执行结构框架**跟踪一个递归函数：方框中包含了每次函数调用的相关信息。每个执行结构框架中都有参数和其他局部变量的值。每个框中还有递归函数代码的相应部分——特别是递归调用以及变元的值。当进行一个递归调用时，在当前执行结构框架的上面就会构造一个新的执行结构框架：调用结束之后这个新的框也随之消失。一个检查标志“✓”指示了现在执行的语句是在当前框中还是在前一个框（也就是创建当前框的执行结构框架）中执行。

例如，以下是在进行factorial(4)调用之后，对factorial函数的单步跟踪执行结构框架：



96



97

对factorial函数的分析已经相当清晰了。需要的执行时间取决于递归调用的次数。对任何参数n，就会有n-1次递归调用。在每次递归调用中需要执行一个常数时间的**if**语句。因此worstTime(*n*)与*n*成线性关系。

递归函数通常都需要一些额外的存储代价。例如，每次进行一个factorial递归调用时，都需要保存返回地址和变元的拷贝。因此，worstSpace(*n*)也和*n*成线性关系。

递归可以使问题的解决更加容易。但是任何可以用递归来处理的问题也同样可以用迭代处理。**迭代**函数使用循环语句取代了递归。例如，以下使用了一个迭代函数计算阶乘:

```
// 前置条件: n>=0
// 后置条件: 返回的数值是n!，也就是1和n之间（包括1和n）所有整数的乘积。
//           worstTime(n)是O(n)。
long factorial (int n)
{
    int product = n;

    if (n == 0)
        return 1;
    for (int i = n - 1; i > 1; i--)
        product = product * i;

    return product;
} // factorial函数
```

在这个迭代的factorial版本中，worstTime(*n*)和*n*成线性关系，这和递归是相同的。但是不论*n*取值如何，迭代函数的跟踪始终只需要使用三个变量，因此没有太大的存储需要，这和递归不同。读者可能马上想到迭代是根据递归的定义进行的，从这个意义上讲迭代比递归要好。这是个不错的观点。最后，这两个函数都是相当容易理解的。迭代多两个变量，但是递归函数展示了一个新的解决问题的技术，当然它需要一些额外的开销。 98

综上所述，在这个例子里，factorial函数的迭代版本比递归版本要好一些。这里的目的是说明了在一个简单的情况下，递归是值得考虑的，尽管最终仍然是迭代更好。但下面4.3节给出的例子中，迭代版本就不那么吸引人了。

4.3 十进制到二进制的转换

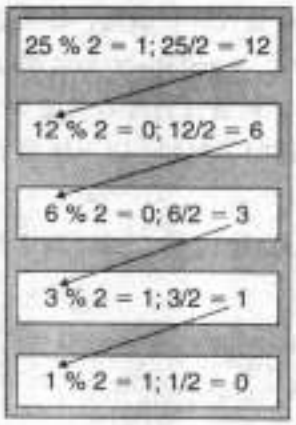
人们习惯以10作为基数计算，可能是因为我们生来都有10个手指。计算机自然以2作为基数计算，这是因为电子元件的二态性。计算机需要执行的一个任务是将十进制（以10为基数）转化成二进制（以2为基数）。现在来开发一个简单的函数来解决这个问题:

给定一个非负整数*n*，输出和它相等的二进制数。

例如，如果*n*是25，那么和它相等的二进制数就是11001。有好几种方法可以解决这个问题，其中之一是基于以下观察:

最右边的位是*n*%2；其他位和*n*/2的二进制等值数相同。

例如，如果*n*是25，那么25的二进制等值数的最右边位是25%2，即1；剩余的位是25/2也就是12的二进制等值数。因此可以得到所有的位如下:

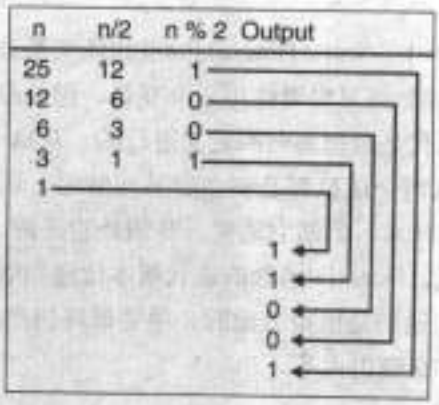


自底向上可以写出剩下的位，这样最右边的位就写在最后。于是输出就是

11001

计算二进制等值数的函数称作writeBinary。下面的图表阐述了调用writeBinary(25)的执行过程：

99



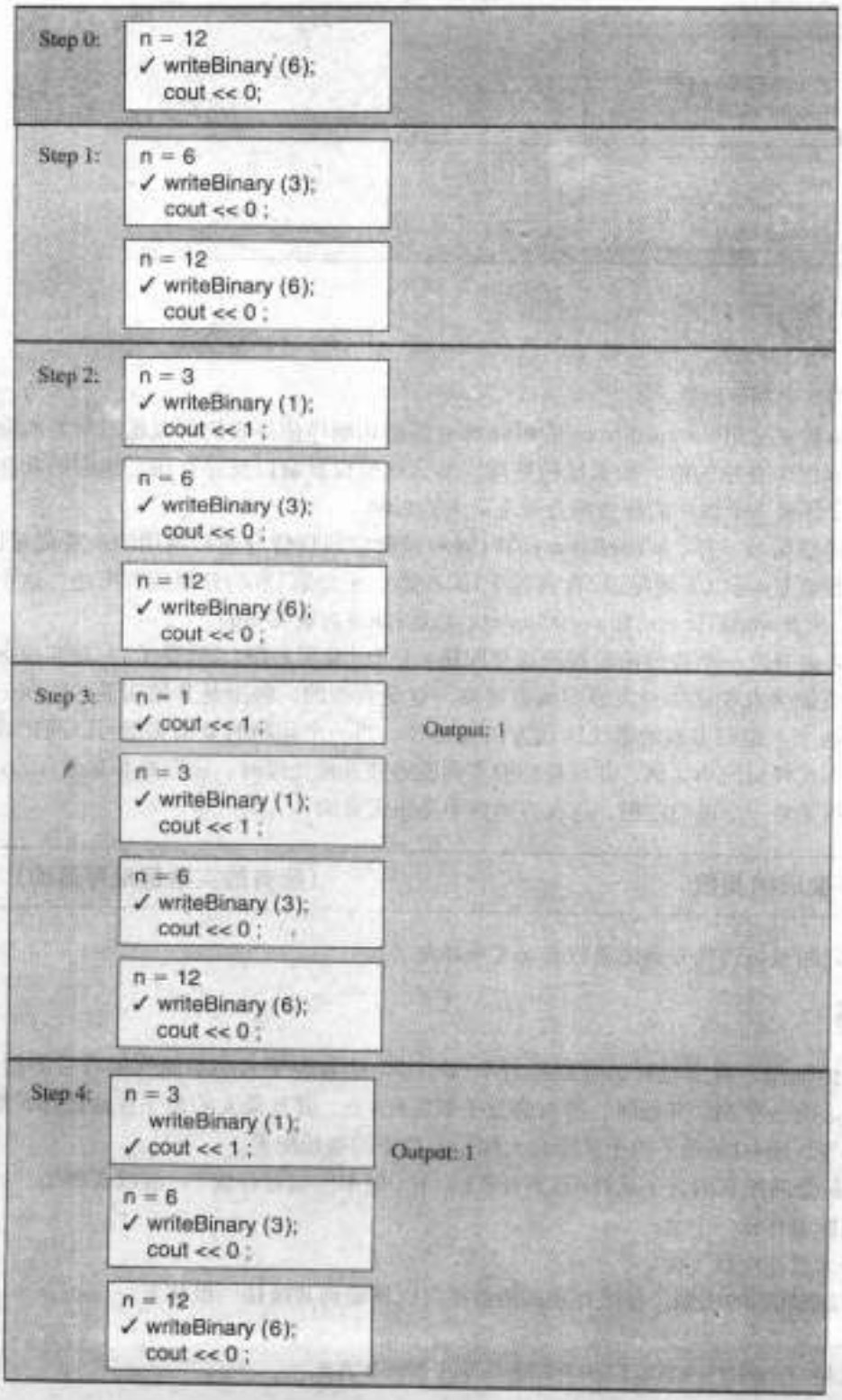
这些探讨说明在任何输出之前必须执行全部的计算。从递归来说，就是在输出n%2之前需要写出n/2的二进制等值数。换句话说，writeBinary函数里应当在输出语句之前放上递归调用。当n/2是0，即n小于等于1时停止。

函数如下：

```
// 前置条件: n>=0
// 后置条件: 输出n的二进制等值数。worstTime(n)
//           是O(log n)。
void writeBinary (int n)
{
    if (n == 0 || n == 1)
        cout << n;
    else
    {
        writeBinary (n / 2);
        cout << n % 2;
    } // else
```

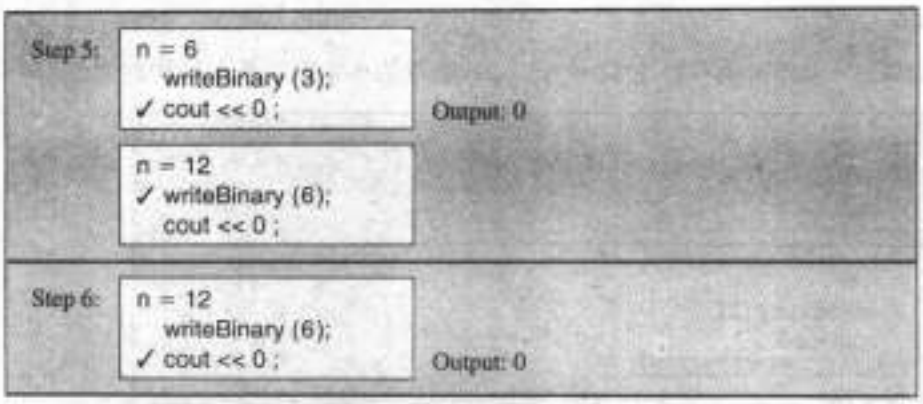
} // writeBinary

下面是初始调用writeBinary(12) 之后writeBinary函数的单步跟踪执行结构框架:



100

101



完整的输出是：

```
1100
```

这就是12的二进制等值数。

在前面的函数定义里，writeBinary的**else**部分的语句顺序使得我们可以推迟所有的输出操作，直到计算出全部位的值。如果反转顺序，那么这些位就会以反序打印。递归的功能如此强大，以至于任何一个微小的改变都会带来巨大的影响。

和通常的递归函数一样，writeBinary的时间和存储空间代价与递归调用的次数成正比。递归调用的次数就是n可以不断除以2直到等于1的次数。正如第3章的分裂规则所述，这个值是floor($\log_2 n$)，因此worstTime(n)和worstSpace(n)都是和n成对数关系的。

现在由读者来开发一个迭代函数解决这个问题。（可以参照习题4.2的提示。）在完成这个迭代函数之后可能会发现这比开发递归函数要难。这是典型的，甚至是非常明显的：在一些适合递归的问题上，递归方案通常比迭代方法流畅些。当一个问题的复杂实例可以简化成一些和复杂实例形式相同的小实例，并且最小的实例能够被直接处理时，递归是非常适合的。

实验10介绍了另一个递归应用，这在自然界中是很常见的。

实验10：斐波纳契数　　（所有的实验都是可选的）

[102] 在下一个问题里，迭代方案比递归方案更难开发了。

4.4 汉诺塔

在汉诺塔游戏里，有三根杆，分别标着A、B、C，还有若干大小不同的编着号的盘子，每个盘子的中心有一个洞。开始时，所有的盘子都在杆A上，并且最大的盘子在最底部，然后是次大的，等等。图4-1显示了四个从小到大编号的盘子的初始配置。

游戏的目标是将所有的盘子从杆A移到杆B上；杆C用来作临时存放[⊖]。游戏规则是

1) 每次只能够移动一个盘。

2) 不允许大盘在小盘上面。

3) 除了第2条规则的限制，每根杆顶部的盘子可以移动到其他任一根杆上。

[⊖] 有些版本中游戏的目标是将盘子从杆A移到杆C，而杆B作临时存放。

现在试着从图4-1的初始配置开始玩这个游戏。马上就面临一个进退两难的问题：将盘1移动到杆*B*上还是杆*C*上呢？如果移错了就可能导致最终四个盘子都在杆*C*上而不是杆*B*上。

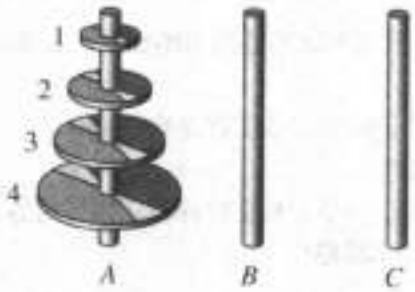


图4-1　四个盘的汉诺塔游戏的初始位置

不去考虑盘1开始时应当移到哪儿，再来集中精力考虑盘4——最底部的盘。当然不能马上把盘4移走，但是最终盘4必须从杆*A*移到杆*B*。根据游戏的规则，将盘4移走之前的配置应当如图4-2所示。

上面的观察有助于解决如何将四个盘从*A*移到*B*吗？确实有几分帮助。但仍然需要决定如何将三个盘（每次只移动一个）从杆*A*移动到杆*C*。然后将盘4从*A*移动到*B*。最后再决定如何将三个盘（每次只移动一个）从杆*C*移动到杆*B*。 103

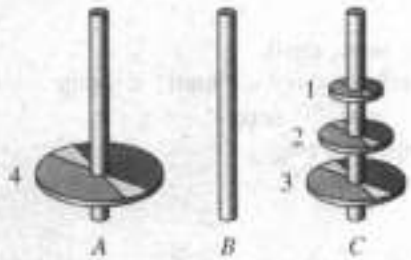


图4-2　将盘4从杆*A*移到杆*B*之前的汉诺塔游戏配置

这个策略说明：把如何移动四个盘子的问题可以简化成如何移动三个盘子的问题。再就是需要决定如何将三个盘子从一根杆移到另一根杆。

但是前面的策略可以重复应用！为了移动三个盘子，即，从杆*A*移到杆*C*，需要先将两个盘子从*A*移到*B*，然后将盘3从*A*移到*C*，最后再将两个盘子从*B*移到*C*。持续地简化，最终任务将是把盘1从一根杆移到另一根上。

在这个问题上4这个数没什么特别。对任何一个正整数*n*都能描述如何将*n*个盘子从杆*A*移动到杆*B*：如果*n*=1，就是将盘1从杆*A*移动到杆*B*。如果*n*>1，

1) 首先，将*n*−1个盘子从杆*A*移动到杆*C*，使用杆*B*临时存放盘子。

2) 然后将盘*n*从杆*A*移动到杆*B*。

3) 最后，将*n*−1个盘子从杆*C*移动到杆*B*，使用杆*A*临时存放盘子。

这还没有彻底解决问题，比如还没有描述如何将*n*−1个盘子从*A*移到*C*。但是这个策略可以很容易地推广，只需要将常量*A*、*B*和*C*替换成变量源、目的地和临时杆。例如，将变量初始化：

源=*A*

目的地=*B*

临时杆=*C*

那么把*n*个盘子从源移动到目的地的通用策略如下。如果*n*是1，把盘1从源移动到目的地。否则，

1) 将*n*−1个盘子（每次移动一个）从源移动到临时杆。

2) 将盘*n*从源移动到目的地。

3) 将*n*−1个盘子（每次移动一个）从临时杆移动到目的地。

以下的递归函数实现了这个策略：

```
// 前置条件: n>0。
// 后置条件: 需要输出将n个盘子从杆orig（源）移动到杆dest（目的地）
//           的步骤。杆temp（临时杆）用于临时存放。
//           worstTime(n)是O(2^n)。
void move (int n, char orig, char dest, char temp)
{
    if (n == 1)
        cout << "Move disk 1 from " << orig << " to "
             << dest << endl;
    else
    {
        move (n - 1, orig, temp, dest);
        cout << "Move disk " << n << " from " << orig
             << " to " << dest << endl;
        move (n - 1, temp, dest, orig);
    } // else
} // move
```

104

跟踪这个函数的执行是非常困难的，因为参数和变元值的相互干扰，使得很难跟踪当前哪个杆是源，哪个是目的地，以及哪个是临时杆。在下面的执行结构框架里，参数的值是调用中的变元的值，后继调用的变元值取自函数代码和当前参数的值。例如，假设初始调用是

```
move(3, 'A', 'B', 'C');
```

那么第0步的参数值将会是那些变元值，因此可以得到

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
```

因为n不等于1，所以执行move函数的**else**部分：

```
move (n - 1, orig, temp, dest);
cout << "Move disk" << n << " from" << orig << "to" << dest << endl;
move (n - 1, temp, dest, orig);
```

那些变元的值又从参数的值得到，因此就是

```
move (2, 'A', 'C', 'B');
```

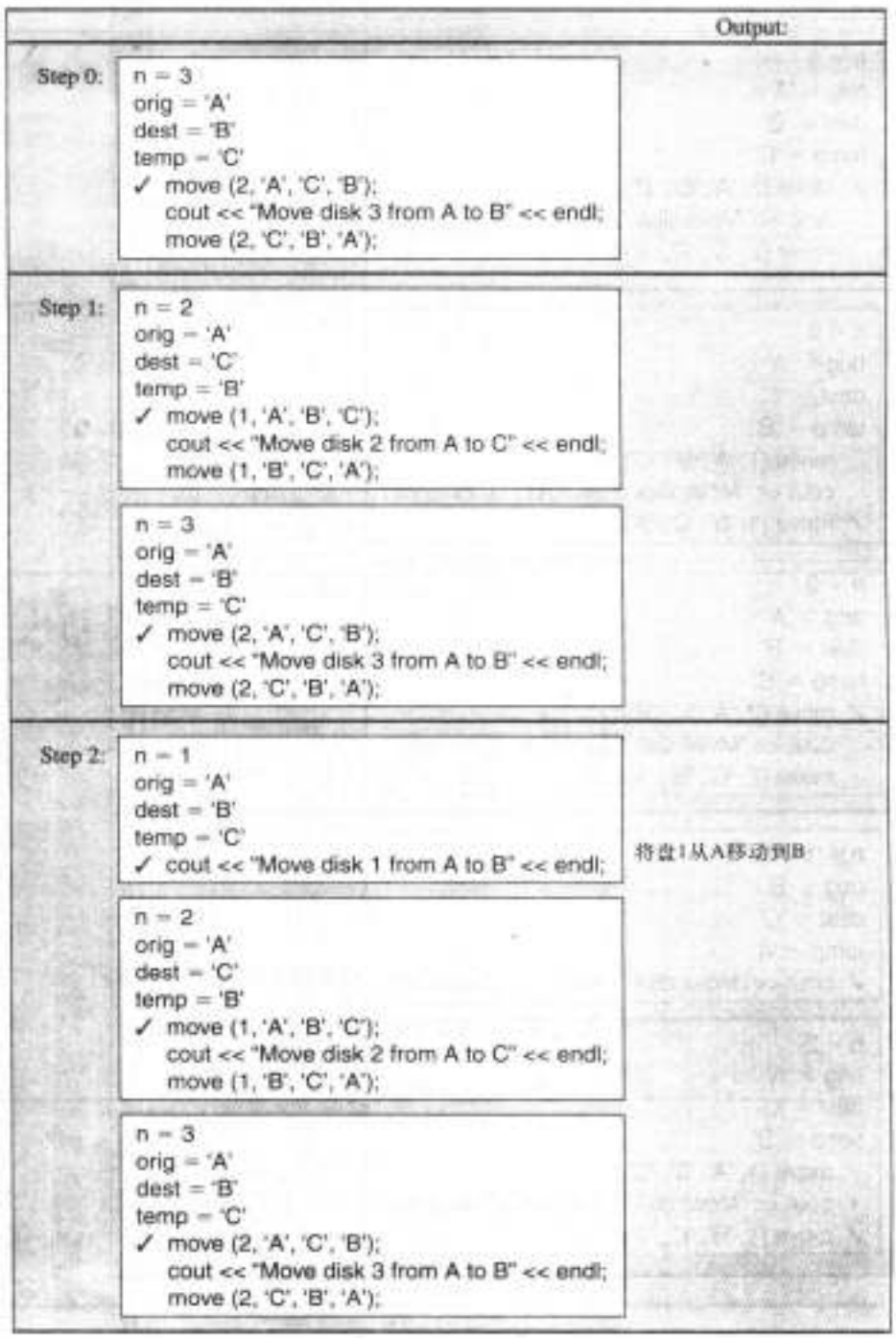
```
cout << "Move disk 3 from A to B" << endl;
move (2, 'C', 'B', 'A');
```

下面是当初始调用是

```
move(3, 'A', 'B', 'C');
```

105

时move函数的单步跟踪执行结构框架。在开始进行跟踪之前要确保了解如何获取参数值和变元值。



106

Step 3:

```
n = 2
orig = 'A'
dest = 'C'
temp = 'B'
    move (1, 'A', 'B', 'C');
✓ cout << "Move disk 2 from A to C" << endl;
    move (1, 'B', 'C', 'A');
```

将盘2从A移动到C

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
✓ move (2, 'A', 'C', 'B');
    cout << "Move disk 3 from A to B" << endl;
    move (2, 'C', 'B', 'A');
```

Step 4:

```
n = 2
orig = 'A'
dest = 'C'
temp = 'B'
    move (1, 'A', 'B', 'C');
    cout << "Move disk 2 from A to C" << endl;
✓ move (1, 'B', 'C', 'A');
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
✓ move (2, 'A', 'C', 'B');
    cout << "Move disk 3 from A to B" << endl;
    move (2, 'C', 'B', 'A');
```

Step 5:

```
n = 1
orig = 'B'
dest = 'C'
temp = 'A'
✓ cout << "Move disk 1 from B to C" << endl;
```

将盘1从B移动到C

```
n = 2
orig = 'A'
dest = 'C'
temp = 'B'
    move (1, 'A', 'B', 'C');
    cout << "Move disk 2 from A to C" << endl;
✓ move (1, 'B', 'C', 'A');
```

Step 5:

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
✓ move (2, 'A', 'C', 'B');
   cout << "Move disk 3 from A to B" << endl;
   move (2, 'C', 'B', 'A');
```

Step 6:

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
   move (2, 'A', 'C', 'B');
✓ cout << "Move disk 3 from A to B" << endl;
   move (2, 'C', 'B', 'A');
```

将盘3从A移动到B

Step 7:

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
   move (2, 'A', 'C', 'B');
   cout << "Move disk 3 from A to B" << endl;
✓ move (2, 'C', 'B', 'A');
```

Step 8:

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'
✓ move (1, 'C', 'A', 'B');
   cout << "Move disk 2 from C to B" << endl;
   move (1, 'A', 'B', 'C');
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
   move (2, 'A', 'C', 'B');
   cout << "Move disk 3 from A to B" << endl;
✓ move (2, 'C', 'B', 'A');
```

Step 9:

```
n = 1
orig = 'C'
dest = 'A'
temp = 'B'
✓ cout << "Move disk 1 from C to A" << endl;
```

将盘1从C移动到A

Step 9:

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'
✓ move (1, 'C', 'A', 'B');
   cout << "Move disk 2 from C to B" << endl;
   move (1, 'A', 'B', 'C');
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
   move (2, 'A', 'C', 'B');
   cout << "Move disk 3 from A to B" << endl;
✓ move (2, 'C', 'B', 'A');
```

Step 10:

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'
   move (1, 'C', 'A', 'B');
✓ cout << "Move disk 2 from C to B" << endl;
   move (1, 'A', 'B', 'C');
```

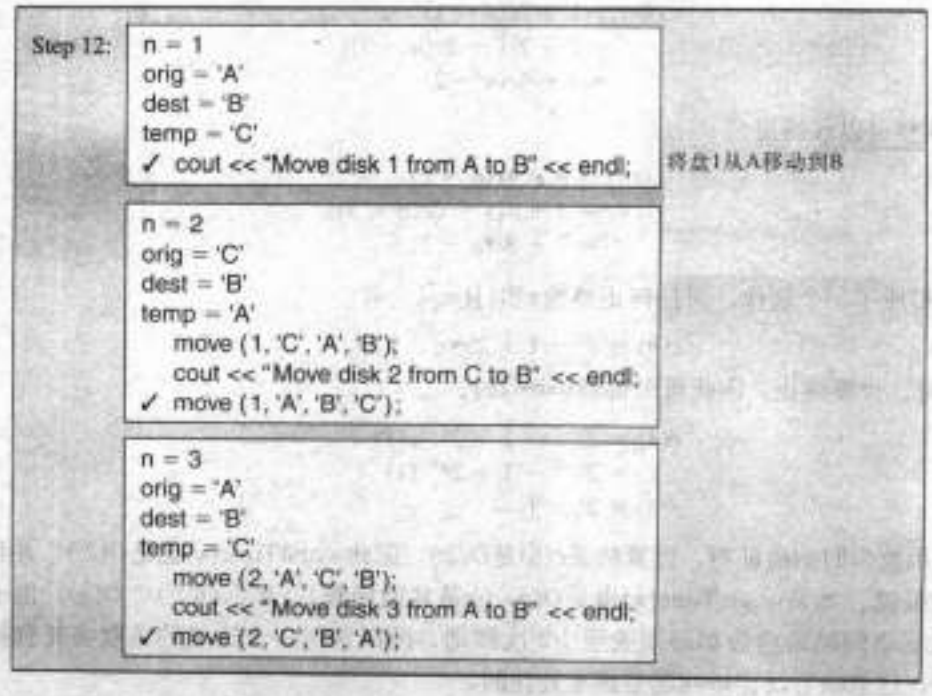
将盘2从C移动到B

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
   move (2, 'A', 'C', 'B');
   cout << "Move disk 3 from A to B" << endl;
✓ move (2, 'C', 'B', 'A');
```

Step 11:

```
n = 2
orig = 'C'
dest = 'B'
temp = 'A'
   move (1, 'C', 'A', 'B');
   cout << "Move disk 2 from C to B" << endl;
✓ move (1, 'A', 'B', 'C');
```

```
n = 3
orig = 'A'
dest = 'B'
temp = 'C'
   move (2, 'A', 'C', 'B');
   cout << "Move disk 3 from A to B" << endl;
✓ move (2, 'C', 'B', 'A');
```



注意上面显示出的不一致性：即开发递归函数的相对容易和跟踪它的执行的相对困难。设想一下跟踪move(15, 'A', 'B', 'C')的执行将会怎样！幸运的是无须忍受这种痛苦。计算机会很好地处理这种乏味的琐事。开发者"只"需要编制正确的程序而由计算机来执行。对move函数以及这一章中的其他递归函数，都可以证明函数的正确性。见习题4.17。

递归函数不能明确地描述执行中涉及的大量细节。因此，递归有时被看作是"懒惰的编程者的问题解决工具"。如果想显示递归的价值，不妨试着去开发move函数的迭代实现。编程项目4.1给出了一些提示。

一个递归关系

当n是盘子的数量时，worstTime(n)是多少？在确定递归函数的时间需求时，对函数的调用次数是极为重要的。让$c(n)$等于一个给定n值的move函数的调用次数，那么对任何一个正整数n，worstTime(n) = $c(n)$。因此现在需要做的就是求$c(n)$的量级，再根据它来求worstTime(n)。当n=1时，只用了一个move调用，因此$c(1)$=1。对n>1，用n作为第一个变元进行move的初始调用，在这个调用中又使用了两个以n−1作为第一个变元的move调用。即，对n>1，

110

$$c(n) = 1 + 2c(n - 1)$$

这个等式称作一个**递归关系**，因为$c(n)$的定义是来自于它前面的值的。对n>2，可以按照以下方式计算$c(n-1)$：开始调用move，在这个调用中又调用了两个以$n-2$作为第一个变元的move函数。即，对n>2，

$$c(n - 1) = 1 + 2c(n - 2)$$

如果将这个等式代换进第一个c的等式，对n>2的情况，就可以得到

$$\begin{aligned} c(n) &= 1 + 2c(n-1) \\ &= 1 + 2[1 + 2c(n-2)] \\ &= 3 + 4c(n-2) \end{aligned}$$

对$n>3$，同样可以代换得到

$$\begin{aligned} c(n) &= 3 + 4c(n-2) \\ &= 3 + 4[1 + 2c(n-3)] \\ &= 7 + 8c(n-3) \end{aligned}$$

这看起来形成了一个规律。对任何正整数n和k且$n>k$，有：

$$c(n) = 2^k - 1 + 2^k c(n-k)$$

当$n-k=1$时，代换终止。因此可以得到$k=n-1$时，

$$\begin{aligned} c(n) &= 2^{n-1} - 1 + 2^{n-1}c(1) \\ &= 2^{n-1} - 1 + 2^{n-1}(1) \\ &= 2^n - 1 \end{aligned}$$

这些可以用数学归纳法证明。计算结果$c(n)$是$O(2^n)$，因此worstTime(n)也是$O(2^n)$，并且是最小的。也就是说，如果worstTime(n)也是$O(g)$（g是其他函数），那么$O(2^n) \subset O(g)$。由于任何用来解决汉诺塔问题的函数都必须做至少2^n次移动，所以任何一个这类的函数将耗费指数级的时间代价，这意味着汉诺塔问题是很难处理的。

move的存储需求是比较适中的，因为尽管调用move时会占用空间，但当调用结束后就会释放空间。因此move需要的额外存储数量不仅仅依赖于move的调用次数，而且依赖于开始但是尚未结束的调用的最大数量。可以利用执行结构框架计算这个数字。每次进行一个递归调用，就会构造另一个执行结构框架，并且每次调用返回，都会删除一个执行结构框架。例如，如果用$n=3$进行第一个move调用，那么执行结构框架的最大数量就是3。通常情况下，执行结构框架的最大数都是n。因此worstSpace(n)和n是成线性关系的。

4.5节提及了解决问题的一个通用策略——回溯，这种策略就是为了某些目的需要按原先步骤折回的方法。从某个回溯应用的组成中抽象出所有回溯应用都需要的组件，这是有很重
111 要的意义的。

4.5 回溯

回溯的基本思想是：如何从一个给定的起始位置到达目的地。重复地选择，也可能是猜测，下一个位置是什么。如果假设的选择是正确的——也就是说，新位置可能在通往目的地的路上，那就前进到这个新的位置上继续。如果这个选择进入了死胡同，就返回前一个位置并进行另一个选择。回溯就是通过一系列的位置选择并从不能到达目的地的位置逆序折返，从而最终到达目的地的策略。

例如，观察图4-3。从P0位置想找到一条路径通往目的地P16，只允许向两个方向移动——北和西。策略是：从任何一个位置上出发，首先试着向北；如果不能向北，再试着向西；如果不能向西，就返回前面选择向北的最近的一个位置并转而向西。在每次移动后，检查结果，即是否到达目的地。根据该策略所确定的移动的顺序，我们对图4-3里的位置编了号。

图4-3中可以看到“逆序折返”。当从位置P4不能向北或向西时，首先返回位置P3。它没有向西的选择，因此返回P2。从位置P2，可以选择向西，于是到达P5，它没有向北的选择，

但是可以向西，于是移动到P6，再向北移动到P7，这是一个死胡同。然后折回P1，并向西移动到P8。从P8不会再向北移动到P5，因为已经发觉了P5将通向一个死胡同。因此从P8向西并最终到达目的地。

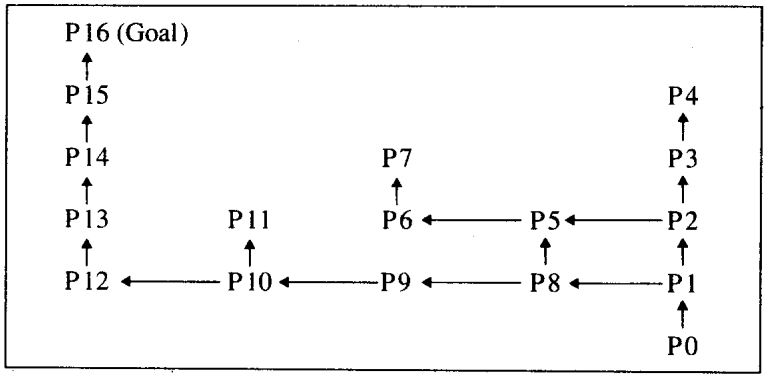


图4-3　回溯获取通往目的地的路径。得到的结果是P0、P1、P8、P9、P10、P12、P13、P14、P15、P16

通往目的地的路径不应该包含任何死胡同，这正说明了回溯的一个微妙之处。访问一个位置时，将把它记录在通往目的地的路径上。但是如果这个位置只不过到达一个死胡同，那将取消这个记录。 112

现在可以使用Wirth(1976, p.138)发现的类回溯算法，而无需为每个具体应用开发回溯方法。下面将展示一个具体应用中的算法。即将讨论的BackTrack类基于Noonan(2000)中的一个类。Application类的实现细节在BackTrack类中是无法访问到的，它要通过头文件Application.h访问。相应的源文件将在具体应用里实现。

BackTrack类的用户提供了：

- 一个源文件实现Application.h
- 一个Position类，它定义了在这个应用里“位置”的意义。

任何回溯应用都使用相同的main函数、BackTrack类和Application头文件。

Application方法对图4-3之前和之后的讨论都是通用的。例如，其中一个方法测试给定的位置是否有效，即是否在通往目的地的一条路上。下面是头文件Application.h，它还声明了一个Iterator类，用来从一个给定位置上进行迭代：

```
#ifndef APPLICATION
#define APPLICATION

#include <iostream>
#include "Position.h"

using namespace std;

class Application
{
    friend ostream& operator<< (ostream& stream,
                                Application& app);
    public:
```

```
        // 后置条件: 返回这个利用输入或赋值生成的
        //           Application的初始状态以及
        //           起始位置。
        Position generateInitialState( );

        // 后置条件: 如果pos在通往目的地的路上就返回真。
        //           否则将返回假。
        bool valid (const Position& pos);

        // 前置条件: pos代表了一个有效位置。
        // 后置条件: pos被记录成一个有效位置。
113     void record (const Position& pos);

        // 后置条件: 如果pos是这个应用的最后一个
        //           位置就返回真，否则
        //           返回假。
        bool done (const Position& pos);

        // 后置条件: pos被标记为不在通往目的
        //           地的路径上。
        void undo (const Position& pos);

    class Iterator
    {
        public:

        // 后置条件: 这个Iterator被初始化。
        Iterator ( );

        // 后置条件: 用pos进行这个Iterator的初始化。
        Iterator (const Position& pos);

        // 前置条件: 这个Iterator可以从这个位置前进。
        // 后置条件: 返回这个Iterator的当前位置，并将
        //           这个Iterator前进到
        //           下一个位置。
        Position operator++ (int);

        //后置条件: 这个Iterator再也不能前进了。
        bool atEnd( );
        protected:
        void* fieldPtr;     // 以后解释
    };// 类Iterator
};// 类Application

#endif
```

Application类不需要任何字段，因为每个问题都是Application的一个实例。也就是说，每个应用，即这个类的每个实现中的任何字段都可以改变。取而代之的是，对每个实现将在与

Application.h对应的源文件中分别定义一些专用的变量。

另一方面，对一个具体应用，嵌入的Iterator类将有若干实例，因此需要字段来区分不同的迭代器对象。但是不同应用的Iterator类的字段也应各不相同。通过定义一个虚字段fieldPtr，使得Application类的每个实现可以为具体应用指定实际的字段。对应一个具体的应用，下面小节中给出了详细的实现细节。 114

BackTrack类是通用的：没有任何特定于应用的信息。头文件如下：

```
#ifndef BACKTRACK
#define BACKTRACK

#include "Application.h";
#include "Position.h";

class BackTrack
{
    public:

        // 后置条件：通过app初始化这个BackTrack对象。
        BackTrack (const Application& app);

        // 后置条件：尝试一条通过pos的路径。
        //           如果尝试成功就返
        //           回真；否则
        //           返回假。
        bool tryToSolve (Position pos);
    protected:
        Application app;
}; // 类BackTrack

#endif
```

现在关注一下tryToSolve方法，它是回溯的基础。对任意pos值，创建一个从那个位置开始的迭代器，并循环直到成功或再也无法进行下去。在每次循环迭代过程中需要检测由迭代器产生的下一个移动。下面列出了三种可能：

1) 那些选择之一是目的地。那么循环终止并返回真，表示成功。

2) 那些选择之一是有效的，但不是目的地。那么从这个有效选择开始进行一个递归调用tryToSolve。

3) 所有的选择都不是有效的。那么循环结束并返回假，表示从当前位置不能到达目的地。

下面是源文件BackTrack.cpp：

```
#include "BackTrack.h"

using namespace std;

BackTrack::BackTrack (const Application& app)
{
    this -> app = app;
} // 构造器
                                                          115
bool BackTrack:::tryToSolve (Position pos)
```

```
{
    bool success = false;

    Position::Iterator itr = pos.begin( );
    while (!success && itr!= pos.end( ))
    {
        pos = itr++;
        if (app.valid (pos))
        {
            app.record (pos);
            if (app.done (pos))
                success = true;
            else
            {
                success = tryToSolve (pos);
                if (!success)
                    app.undo (pos);
            } // 取消
        } // 一个有效位置
    } // while
    return success;
} // 方法tryToSolve
```

tryToSolve的变元代表一个有效并被记录的位置。无论何时从tryToSolve返回，就恢复pos前一个调用的值，如果它通向死胡同就取消它的标记。

main函数初始化应用并生成初始状态。generateInitialState方法可能读入或指定初始位置。在这两种情况下，该方法都将返回初始位置并记录；初始位置是不能被回溯的。对tryToSolve的初始调用最终将返回成功或失败：如果成功，就输出应用的最终状态。

下面是BacktrackMain.cpp:

```
#include <iostream>
#include <string>
#include "BackTrack.h"
#include "Application.h"
#include "Position.h"

using namespace std;

int main( )
{
    const string INITIAL_STATE =
        "The initial state is as follows:\n";
    const string SUCCESS =
        "\n\nA solution has been found:";
    const string FAILURE =
        "\n\nThere is no solution:";
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    Application app;
```

```
    BackTrack b (app);

    cout << INITIAL_STATE;
    Position start = app.generateInitialState( );
    cout << app;
    if (!app.valid (start))
        cout << FAILURE << endl;
    else
    {
        app.record (start);
        if (app.done (start) || b.tryToSolve (start))
            cout << SUCCESS << endl << app;
        else
        {
            app.undo (start);
            cout << FAILURE << endl;
        } // 失败
    } // start有效

    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );

    return 0;
} // main
```

现在来开发一个框架来实现回溯，利用这个框架可以很容易地解决很多问题。下面小节中描述了一个这样的问题，编程项目4.2和编程4.3中还有两个，第14章中还将遇到一个。

令人惊奇的应用

为了了解回溯的应用，让我们来开发一个程序——在迷宫里寻找路径。例如，图4-4表示了一个7 × 13的迷宫，其中1代表通道，0代表墙。在迷宫里只允许水平和垂直移动；禁止斜向 [117]
的移动。起始位置必须是1，在左上角，目的地在右下角。

```
1 1 1 0 1 1 0 0 0 1 1 1 1
1 0 1 1 1 0 1 1 1 1 1 0 1
1 0 0 0 1 0 1 0 1 0 1 0 1
1 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 1 1 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1
```

图4-4 一个迷宫：1代表通道，0代表墙。假设起始位置在左上角，目的地在右下角

对这个迷宫，一个成功的遍历将会产生一条从起始位置到目的地的路径，然后把每个这样的位置标记上9。因为在这个迷宫里有两条可行的路径，所以最后选择哪条路径取决于迭代器如何排序各个选择。为了具体描述，先假设这些选择按照北、东、南和西排序。例如，从坐标（5，8）的位置开始，第一个选择应当是（4，8），然后依次是（5，9）、（6，8）和（5，7）。

以（0，0）作为初始位置，下面的这些位置可能出现在最后的路径上，因此被记录:

(0,1)//向东移动
(0,2)//向东移动
(1,2)//向南移动
(1,3)//向东移动
(1,4)//向东移动
(0,4)//向北移动
(0,5)//向东移动

这里最后一个位置是个死胡同，因此取消（0，5）和（0，4）的标记，回溯到（1，4），然后记录:

(2,4)//向南移动
(3,4)//向南移动
(3,5)//向东移动

最后又到达一个死胡同。取消（3，5）的标记之后，折回（3，4），然后前进——不再需要任何回溯，到达目的地。图4-5显示了穿越图4-4所示迷宫的相应路径，在路径上的位置用9表示，死胡同用2表示。

```
9 9 9 0 2 2 0 0 0 2 2 2 2
1 0 9 9 9 0 2 2 2 2 2 0 2
1 0 0 0 9 0 2 0 2 0 2 0 2
1 0 0 0 9 2 2 0 2 0 2 2 2
1 1 1 1 9 0 0 0 0 1 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 0
0 0 0 0 9 9 9 9 9 9 9 9 9
```

图4-5 穿越图4-4的迷宫的一条路径。在路径上的点用9表示，死胡同用2表示

在这个应用里，一个位置只是一对坐标：行，列。Position类是很容易开发的，下面是
118 Position.h:

```
#ifndef POSITION
#define POSITION

class Position
{
    protected:
        int row,
            column;

    public:
        Position( );
        Position (int row, int column);
        void setPosition (int row, int column);
        int getRow( );
        int getColumn( );
}; // 类Position

#endif
```

然后是Position.cpp:

```
#include "Position.h"

Position::Position( )
{
    row = 0;
    column = 0;
} // 缺省构造器

Position::Position (int row, int column)
{
    this -> row = row;
    this -> column = column;
} // 构造器

void Position::setPosition (int row, int column)
{
    this -> row = row;
    this -> column = column;
} // 方法setPosition

int Position::getRow( )
{
    return row;
} // 方法getRow()

int Position::getColumn( )
{
    return column;
} // 方法getColumn()
```

119

头文件Application.h在Maze.cpp中得到实现。下面除了嵌入的Iterator类以外，其余的就是Maze.cpp:

```
#include <iostream>
#include "Application.h"

const short WALL = 0;

const short CORRIDOR = 1;

const short PATH = 9;

const short TRIED = 2;

const short ROWS = 7;

const short COLUMNS = 13;

short grid[ROWS][COLUMNS] =
{
    {1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1},
    {1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1},
```

```
        {1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1},
        {1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1},
        {1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1}
};// grid

Position start,
         finish;
```

120

```
using namespace std;

Position Application::generateInitialState( )
{
      const string START_PROMPT =
            "Please enter the start row and start column: ";
      const string FINISH_PROMPT =
            "Please enter the finish row and finish column: ";

      int row,
          column;

      cout << START_PROMPT;
      cin  >> row >> column;
      start.setPosition (row, column);
      cout << FINISH_PROMPT;
      cin >> row >> column;
      cin.get( );
      finish.setPosition (row, column);
      return start;
} // 方法generateInitialState

bool Application::valid (const Position& pos)
{
      if (pos.getRow( ) >= 0 && pos.getRow( ) < ROWS &&
            pos.getColumn( ) >= 0 && pos.getColumn( ) < COLUMNS &&
            grid [pos.getRow( )][pos.getColumn( )] == CORRIDOR)
                  return true;
      return false;
} // 方法valid

void Application::record (const Position& pos)
{
      grid [pos.getRow( )][pos.getColumn( )] = PATH;
} // 方法record

bool Application::done (const Position& pos)
{
      return pos.getRow( ) == finish.getRow( ) &&
            pos.getColumn( ) == finish.getColumn( );
} // 方法done
```

```
void Application::undo (const Position& pos)
{
    grid [pos.getRow( )][pos.getColumn( )] = TRIED;
} // 方法undo

ostream& operator<< (ostream& stream, Application& app)
{
    cout << endl;

    for (int row = 0; row < ROWS; row++)
    {
        for (int column = 0; column < COLUMNS; column++)
            cout << grid [row][column] << ' ';
        cout << endl;
    } // 外部 for
    return stream;
} // 运算符 <<
```

121

在Maze.cpp中，常量和变量标识符（如**WALL**、**CORRIDOR**、**grid**、**start**和**finish**）都不是字段，因为它们只有在Application的迷宫实现里才有意义。

Maze.cpp的其余部分是嵌入类Iterator的开发。这个类有三个**int**字段：

```
row         //这个迭代器的当前行
column      //这个迭代器的当前列
direction   //这个迭代器的方向：0代表北，1代表东，2代表南，3代表西
```

但是一个类的字段必须在头文件里而不是源文件里定义。现在面临一个困难的选择：头文件Application.h是通用的，因此它不能包含任何专门的迷宫应用信息。利用一个**void**指针可以让我们走出困境：

```
void *fieldPtr;
```

这给人的第一印象是**void**指针指向空。但实际上恰恰相反：任何类型的指针都可以分配给一个**void**指针！例如，可以使用下列程序：

```
void* ptr;
int* intPtr = new int;
string* stringPtr = new string;

*intPtr = 50;
ptr = intPtr;
cout << *(int*)ptr << endl;

*stringPtr = "yes";
ptr = stringPtr;
cout << *(string*)ptr << endl;
```

输出将是：

122

```
50
yes
```

注意在这个代码里，**void**指针被脱引用之前必须将**void**指针明确转换为一个具体的指针类型。

头文件Application.h里将fieldPtr定义成一个**void**指针，而在源文件Maze.cpp里声明了一个**结构**（所有成员都是**公有**的类），它有三个字段：

```
struct itrFields
{
    int row,
        column,
        direction;
}; // itrFields
```

Iterator的构造器为fieldPtr分配一个类型是这个**struct**指针的变量，并且++运算符和atEnd()方法利用（通过类型转换）*fieldPtr的结果值。下面给出了Iterator方法的实现：

```
Application::Iterator::Iterator (Position pos)
{
    itrFields* itrPtr = new itrFields;
    itrPtr -> row = pos.getRow( );
    itrPtr -> column = pos.getColumn( );
    itrPtr -> direction = 0;
    fieldPtr = itrPtr;
} // 构造器

Position Application::Iterator::operator++(int)
{
    itrFields* itrPtr = (itrFields*)fieldPtr;
    int nextRow = itrPtr -> row,
        nextColumn = itrPtr -> column;
    switch (itrPtr -> direction++)
    {
        case 0: nextRow = itrPtr -> row -1; // 北
                break;
        case 1: nextColumn = itrPtr -> column + 1; // 东
                break;
        case 2: nextRow = itrPtr -> row + 1; // 南
                break;
        case 3: nextColumn = itrPtr -> column - 1; // 西
    } // switch;
123 Position next (nextRow, nextColumn);
    return next;
} // 运算符 ++

bool Application::Iterator::atEnd( )
{
    return ((itrFields*)fieldPtr) -> direction >= 3;
} // 方法atEnd
```

因为fieldPtr是Iterator里的一个字段，所以对Iterator的每个实例都有一个fieldPtr的拷贝。因此即使在对tryToSolve的不同递归调用过程中创建了各自的迭代器，每个迭代器也都有它自

己的fieldPtr。可以避免**void**指针指向grid字段的原因是由于在应用里只有一个grid字段，因此不需担心多个grid实例的交叉影响。

不是采用单步跟踪，而是列出初始调用tryToSolve(pos)((其中pos=(0,0))之后的前几个有效选择:

(0,1)//向东移动
(0,2)//向东移动
(1,2)//向南移动
(1,3)//向东移动
(1,4)//向东移动
(0,4)//向北移动
(0,5)//向东移动；死胡同；折回(1,4)并重新选择
(2,4)//向南移动
(3,4)//向南移动
(3,5)//向东移动；开始通向死胡同
……

tryToSolve方法在这个应用里耗费了多长的时间？假设迷宫有n个位置。在最坏情况下，如图4-6所示，每个位置都被考虑到，因此worstTime(n)和n成线性关系。而且有一大半的位置是有效的，所以有$O(n)$次tryToSolve的递归调用，因此worstSpace(n)也和n成线性关系。

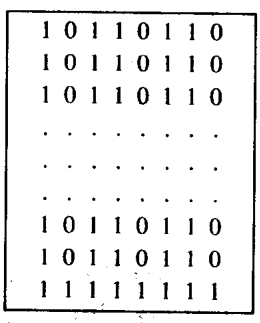


图4-6　最坏情况下的迷宫：列1，4，7，……中除了最后一行全是0；而在迷宫的其他位置都是1

124

编程项目4.2和编程4.3包含了回溯的两个应用。由于刚才的例子把回溯和迷宫遍历分开讨论，所以BackTrack类和Application头文件在新项目里都没有改变，主函数也是一样！实际上，在编程项目4.2和编程4.3里，Position类也没有变化。为了完成编程项目4.2和编程4.3，所有要做的只是去实现Application.h。

现在回过来关注一下查找技术——折半查找。在数组里开发一个递归函数执行折半查找。实验11使用一个普遍适用的迭代折半查找算法进一步探究了这项技术。

4.6　折半查找

假设要在数组中查找一项。最简单的方法是顺序查找：从第一个位置开始，持续地向后查找，直到找到这个项或是到达数组的尾部。这个查找策略称作**顺序查找**，是algorith.h里的通用型算法find的基础:

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, // 定位于容器中的第1项
                    InputIterator last, // 定位于容器中最后一项之后
                    const T& value)
{
    while (first != last && *first != value)
        ++first;
    return first;
}
```

这个算法应用在容器对象（使用迭代器）上和应用在数组（使用指针）上一样成功。例如，可以顺序查找一个employees的Linked容器或是由20个salaries组成的数组：

```
Linked<Employee>::Iterator itr = find (employees.begin( ),
        employees. end( ), newEmployee);
double* salaryPtr = find (salaries, salaries + 20, newSalary);
if (itr != employees.end( ))
    cout << "newEmployee found!" << endl;
if (salaryPtr != salaries + 20)
    cout << "newSalary found at index" << (salaryPtr - salaries);
```

在数组的顺序查找里，如果查找不成功就出现了最坏时间花费。如果那样的话，必须扫描整个数组，因此worstTime(n)和n是成线性关系的。在平均情况下，假设每个位置都以相同的几率存放查找的元素，那么大约查找$n/2$个元素，因此averageTime(n)也和n成线性关系。

可以改进时间代价吗？当然可以！在这一节里将讲述一个数组查找技术，这时
125 worstTime(n)及averageTime(n)和n只是成对数关系。

给出一个即将被查找的数组和需要查找的数值，然后开发一个**折半查找**——之所以这样叫是因为在每一步上都把查找区域分成两个，直到查找结束。一个重要的规定是：

折半查找的数组应当是有序的。

假设这个数组的元素中已经定义了**operator<**。为了具有普遍性，希望查找已经定义**operator<**的任意元素类型的排序数组：string、**int**、用户定义的类，等等。因此函数将元素类型T作为一个模板：

```
template <classT>
```

出于简单性上的考虑，函数将返回**true**或**false**，这依赖于探索的值是否被找到。实验11将这个观点延伸到函数，该函数指示探索的值在一个排序数组中的位置：也就是说，在不破坏数组顺序的前置条件下，将数值插入到什么地方。尽管实验11提出了函数的迭代版本（这是标准模板库的典型实现），但下面开发的函数是递归的。

函数共有三个参数：

1) 一个指针指向正在搜索区域的第一个位置。

2) 一个指针指向正在搜索区域之后的第一个位置。

3) 被搜索的值。

下面是完整的方法接口：

```
// 前置条件：数组从头到尾根据
//           运算符< 排序。
```

```
// 后置条件: 如果搜索的值出现在数组的
//           元素序列里就返回真。否则，
//           返回假。
template<class T>
bool binary_search (T* first, T* last, const T& value);
```

注意T*被用作指针指向数组的一个位置。如果myArray是一个五元素数组，那么myArray是指向数组第一个位置的指针，而myArray+5是指向数组第五个位置后面的一个指针。因此binary_search调用里的第二个变元不是指向搜索区域的最后一个位置，而是指向搜索区域之后的第一个位置。如果读者能长久地牢记这种观点，那么对标准模板库的学习将更容易些。

下面的文件中有一个main函数，它示范了标准模板库中对binary_search函数的几次调用。

```
#include <iostream>
#include <string>
#include <algorithm> // 定义binary_search算法
using namespace std;

int main( )
{
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    const string FOUND_MESSAGE = " was found.";

    const string NOT_FOUND_MESSAGE = " was not found.";

    const int INT1 = 111;

    const int INT2 = 702;

    const string STRING1 = "Ken";

    const string STRING2 = "Ed";

    const string STRING3 = "Abe";

    int scores [9] = {7, 22, 84, 106, 117, 200, 494, 555, 702};

    string names [10] = {"Ada", "Ben", "Carol", "Dave", "Ed", "Frank",
                         "Gerri", "Helen", "Iggy", "Joan"};

    if (binary_search (scores, scores + 9, INT1))
        cout << INT1 << FOUND_MESSAGE << endl;
    else
        cout << INT1 << NOT_FOUND_MESSAGE << endl;

    if (binary_search (scores, scores + 9, INT2))
        cout << INT2 << FOUND_MESSAGE << endl;
    else
        cout << INT2 << NOT_FOUND_MESSAGE << endl;

    if (binary_search (names, names + 10, STRING1))
        cout << STRING1 << FOUND_MESSAGE << endl;
    else
```

126

```
        cout << STRING1 << NOT_FOUND_MESSAGE << endl;

    if (binary_search (names, names + 10, STRING2))
        cout << STRING2 << FOUND_MESSAGE << endl;
    else
        cout << STRING2 << NOT_FOUND_MESSAGE << endl;

    if (binary_search (names, names + 10, STRING3))
        cout << STRING3 << FOUND_MESSAGE << endl;
    else
127 cout << STRING3 << NOT_FOUND_MESSAGE << endl;

    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );
    return 0;
} // main
```

程序的输出是:

```
111 was not found.
702 was found.
Ken was not found.
Ed was found.
Abe was not found.
```

按下“回车”键可以关闭输出窗口。

递归binary_search算法的基本思想是先找到从first到last区域间的中间元素。中间元素的值是*middle，middle的定义是:

```
T* middle = first + (last - first) / 2;
```

这个赋值语句右边的指针运算是非常与众不同的：两个指针相减可以求出它们之间的（整数）距离，并且一个指针可以和一个偏移量相加。

如果中间元素比value（搜索的数值）小，就从first的新值（即middle+1）到last之间的新区域中执行折半查找，也就是

```
if (*middle < value)
    binary_search (middle + 1, last, value);
```

否则，如果value小于中间元素，那就从first到last的新值（即middle）之间的新区域中执行折半查找。也就是

```
else if (value < *middle)
    binary_search (first, middle, value);
```

否则，返回**true**（因为中间元素等于value）。

例如，根据这个策略，在图4-7所示的数组names里搜索“Ed”。这个图显示了调用binary_search函数查找“Ed”的程序的状态。赋值语句

```
middle = first + (last - first) / 2;
```

使得middle指向“Frank”项。

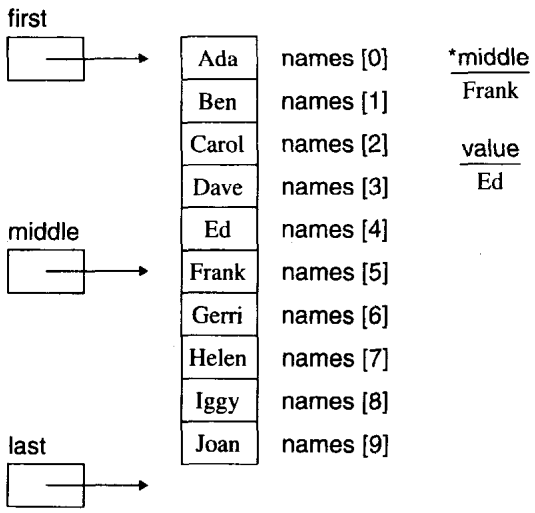


图4-7　在调用binary_search（names,names+10,"Ed"）之初的程序状态。这时将会从下标0到下标9之间查找"Ed"

中间元素"Frank"不小于"Ed"，而"Ed"小于"Frank"，因此执行从first到middle之间区域的折半查找。调用是

```
binary_search(first,middle,value);
```

128

形参last获得了变元middle的值。在binary_search的执行中，赋值语句

```
middle=first+(last-first)/2;
```

令middle指向"Carol"所在的位置，如图4-8所示。

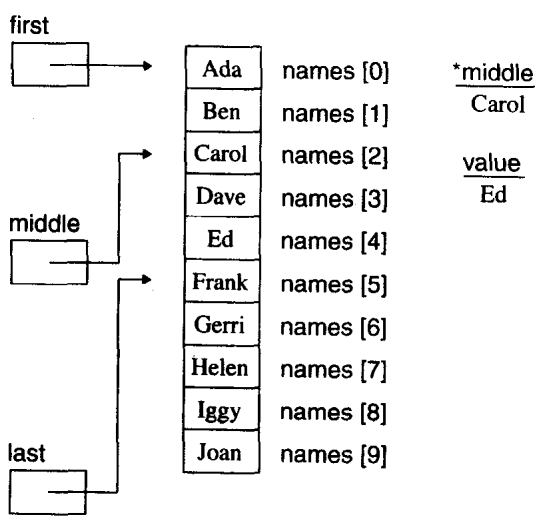


图4-8　当在下标0到下标4之间的区域折半查找"Ed"之初的程序状态

中间元素"Carol"比"Ed"小，因此从下标3（"Carol"之后的下标）到下标5（last指向的下标）之间执行折半查找。调用是

```
binary_search(middle+1,last,value);
```

形参first获得了变元middle+1的值。在binary_search的执行中，赋值语句

```
middle=first+(last-first)/2;
```

令middle指向“Ed”项的位置，参见图4-9。成功！中间元素等于value，于是返回**true**。

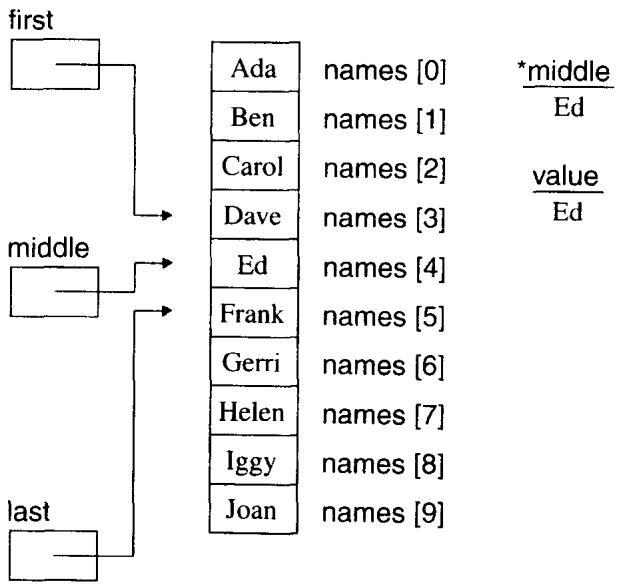


图4-9 当在下标3到下标5之间的区域折半查找“Ed”之初的程序状态

惟一尚未解决的问题就是如果数组中没有元素和value相等会怎样，这时用户希望返回**false**。对任何区域的搜索都应当使first<last，因此当first>=last时停止搜索。

下面给出了完整的定义：

129
~
130

```
template<class T>
bool binary_search (T* first, T* last, const T& value)
{
    if (first >= last)
        return false;
    T* middle = first + (last - first) / 2;
    if (*middle < value)
        return binary_search (middle + 1, last, value);
    else if (value < *middle)
        return binary_search (first, middle, value);
    return true;
} // binary_search
```

下面是初始调用

```
binary_search(names,names+10, "Dan");
```

之后的这个函数的执行跟踪，其中names数组的定义和前面相同。注意“Dan”并不在names数组里。

Step 0:

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]
first = names
last = names + 10
value = "Dan"
middle = names + 5
*middle = "Frank"
return binary_search (names, names + 5, "Dan");
```

Step 1:

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]
first = names
last = names + 5
value = "Dan"
middle = names + 2
*middle = "Carol"
return binary_search (names + 3, names + 5, "Dan");
```

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]
first = names
last = names + 10
value = "Dan"
middle = names + 5
*middle = "Frank"
return binary_search (names, names + 5, "Dan");
```

131

Step 2:

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]
first = names + 3
last = names + 5
value = "Dan"
middle = names + 4
*middle = "Ed"
return binary_search (names + 3, names + 4, "Dan");
```

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]
first = names
last = names + 5
value = "Dan"
middle = names + 2
*middle = "Carol"
return binary_search (names + 3, names + 5, "Dan");
```

Step 2:

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]

first = names
last = names + 10
value = "Dan"
middle = names + 5
*middle = "Frank"

return binary_search (names, names + 5, "Dan");
```

Step 3:

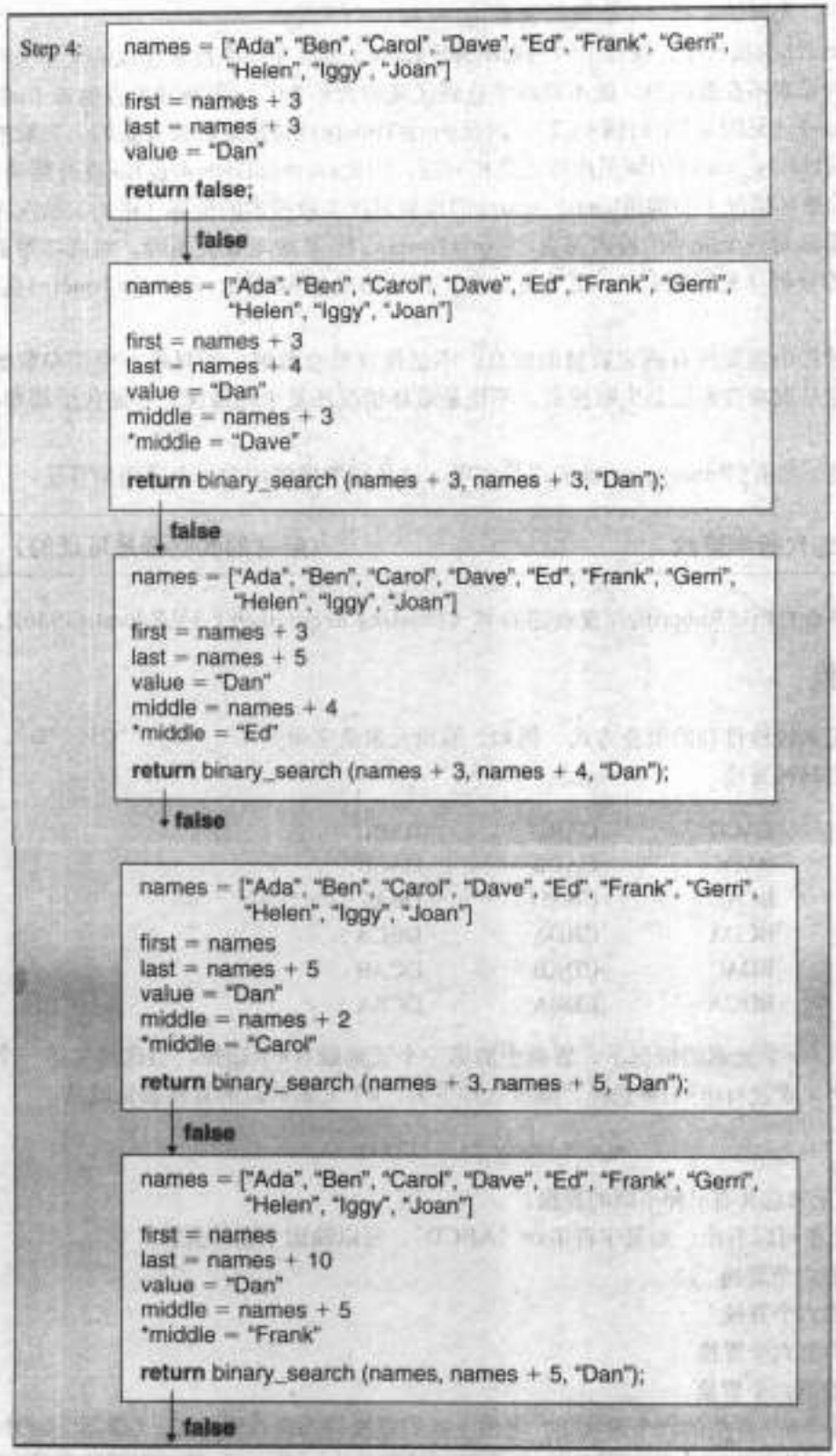
```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]

first = names + 3
last = names + 4
value = "Dan"
middle = names + 3
*middle = "Dave"

return binary_search (names + 3, names + 3, "Dan");
```

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]

first = names + 3
last = names + 5
value = "Dan"
middle = names + 4
*middle = "Ed"

return binary_search (names + 3, names + 4, "Dan");
```

132

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]

first = names
last = names + 5
value = "Dan"
middle = names + 2
*middle = "Carol"

return binary_search (names + 3, names + 5, "Dan");
```

```
names = ["Ada", "Ben", "Carol", "Dave", "Ed", "Frank", "Gerri",
         "Helen", "Iggy", "Joan"]

first = names
last = names + 10
value = "Dan"
middle = names + 5
*middle = "Frank"

return binary_search (names, names + 5, "Dan");
```



133

binary_search函数会耗费多少时间呢？这要分为两种情况：找不到元素的失败搜索和找到

元素的成功搜索。先开始分析一个失败的搜索。

在binary_search的执行中，假设中间元素不等于value。那么下一个搜索区域的大小大约减半。如果搜索的元素不在数组里，就不断减半直到区域的大小为0。n不断减半直到等于0的次数是floor($\log_2 n$)+1（见附录1中的例A1.2）。因此worstTime(n)和n是成对数关系的。失败搜索在平均情况下对binary_search的调用次数也是相同的，因此averageTime(n)也和n成对数关系。

成功搜索的最坏情况下的调用binary_search的次数只比失败搜索的最坏（平均）情况下的调用多一次。所以对一个成功的搜索而言，worstTime(n)和n是成对数关系的。对成功搜索的平均情况，它的分析（见习题4.13）更为复杂些，但是结果是相同的：averageTime(n)也和n成对数关系。

每次调用过程中需要保存固定数量的信息：不是保存整个数组，而只是一个指向数组的指针。所以不论是成功搜索还是失败搜索，不论是最坏情况还是平均情况，存储代价都和n成对数关系。

在实验11里，将看到binary_search的迭代实现，这是标准模板库的一个通用型算法。

134

实验11：迭代折半查找　　（所有的实验都是可选的）

下一个例子取自Eric Robert的深受欢迎的书《*Thinking Recursively*》[见Roberts(1986)]。

4.7 生成置换

置换是将元素线性排列的组合方式。例如，假设元素是字母‘A’、‘B’、‘C’、‘D’，那么可以得到下面24种置换：

ABCD	BACD	CABD	DABC
ABDC	BADC	CADB	DACB
ACBD	BCAD	CBAD	DBAC
ACDB	BCDA	CBDA	DBCA
ADBC	BDAC	CDAB	DCAB
ADCB	BDCA	CDBA	DCBA

一般来说，在n个元素的情况下，置换中的第一个元素就有n个选择。当选择完第一个元素之后，第二个元素就有(n−1)种选择。继续这样下去，n个元素的总的置换数量就是

$$n(n-1)(n-2)\ldots(2)(1)$$

也就是说，n个元素总共有$n!$种不同的置换。

从这个例子中可以看出，如果字符串s=“ABCD”，可以输出下面的置换：

‘A’打头的六个置换

‘B’打头的六个置换

‘C’打头的第六个置换

‘D’打头的第六个置换

如何得到从‘A’打头的六个置换呢？考察上面的置换列表将找到答案。（提示：6=3! ）

观察列表发现的关键问题是：每个以‘A’打头的置换中，在‘A’后面跟着的是‘BCD’

的不同的置换。这就给出了一个递归的解决方法。根据“BCD”的六个不同的置换写出完整的s的字符串，于是就得到了‘A’打头的“ABCD”的六种不同的置换。

对接下来的六个置换，可以先交换‘A’和‘B’，那么s=“BACD”。然后重复前面的过程——这时应求出“ACD”的置换，再输出s的每个置换。

再下一组置换，首先交换‘B’和‘C’，也就是 s[0]和s[2]，那么s=“CABD”。接着置换“ABD”（即，s[1...3]），然后输出s的每个置换。

最后六个置换，先交换s[0]和s[3]（即，‘C’和‘D’），这样s=“DABC”，然后在每次s[1...3]（即，“ABC”）置换后输出s。

正如使用高效的递归斐波纳契函数做过的实验9一样，让permute成为一个包装函数。那么置换的每一层次的起始位置都可以是递归函数的一个变量。这个实现细节隐藏在包装函数的定义中，该函数包含了一个常量引用参数s，并把s传递给递归函数。那么permute函数执行后原始的字符串s将不会改变。 135

以下是包装函数：

```
// 后置条件：打印输出s的全部置换。
void permute (const string& s)
{
    rec_permute (s, 0);
} // permute
```

递归函数的函数接口是：

```
// 后置条件：每个置换s[k...s.length()-1]。
//           后输出s。
void rec_permute (string s, unsigned k);
```

参数k是**unsigned**（无符号）类型，这是因为它需要和s.length()比较，而s.length()返回类型是**unsigned int**。

交换和置换可以在一个**for**循环里完成：

```
for (unsigned i = k; i < s.length( ); i++)
{
    swap (s [i], s [k]); // swap是一个通用型算法
    rec_permute (s, k + 1);
} // for
```

注意，当第一次执行循环时，s[i]和它自身交换。这使得在第一次循环里s[i]保持不变。例如，在置换“ABCD”里，开始保留‘A’不变并置换“BCD”。

当k等于s.length()-1时，递归调用序列停止。这时可以输出s。完整的函数是相当简单的：

```
// 后置条件：每个置换s[k...s.length()-1]。
//           后输出s。
void rec_permute (string s, unsigned k)
{
    if (k == s.length( ) - 1)
        cout << s << endl;
    else
```

```
        for (unsigned i = k; i < s.length( ); i++)
        {
            swap (s [i], s [k]); // swap是一个通用型算法
            rec_permute (s, k + 1);
        } // for
136 } // rec_permute
```

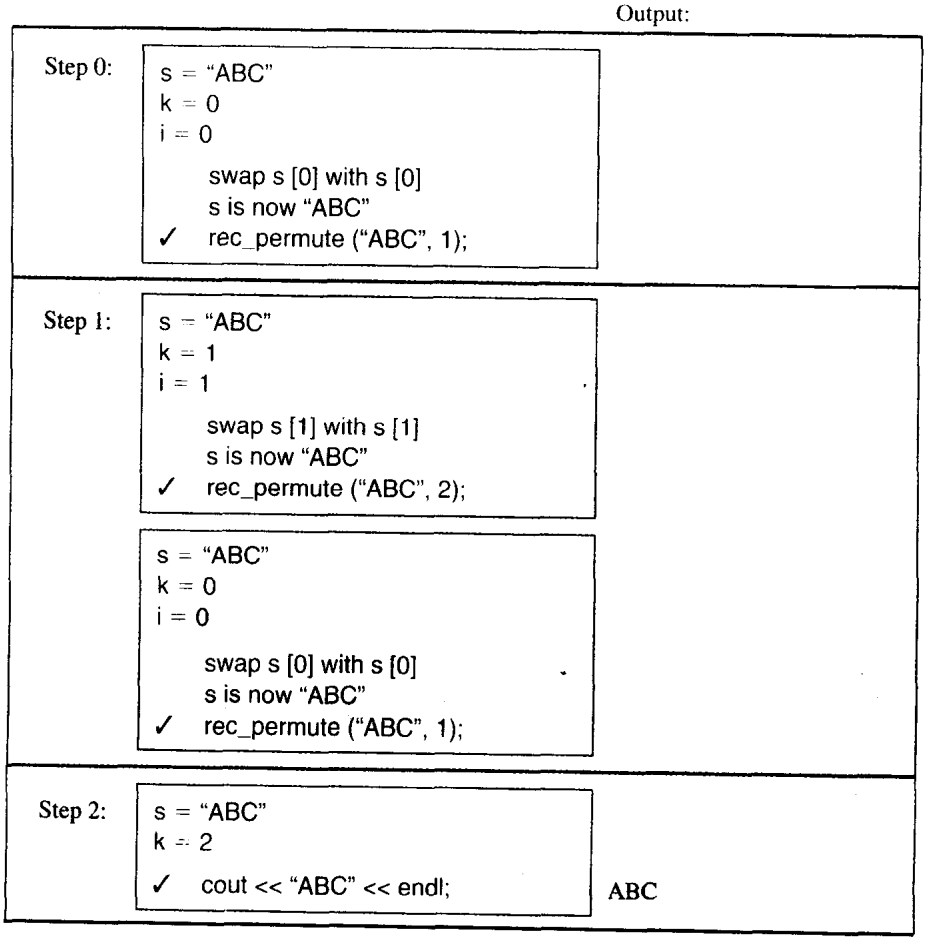
紧记s的值在递归调用过程中是不会改变的：s是rec_permute的一个值参。但是交换会改变s的值。例如，当调用

```
rec_permute ("BAC", 1);
```

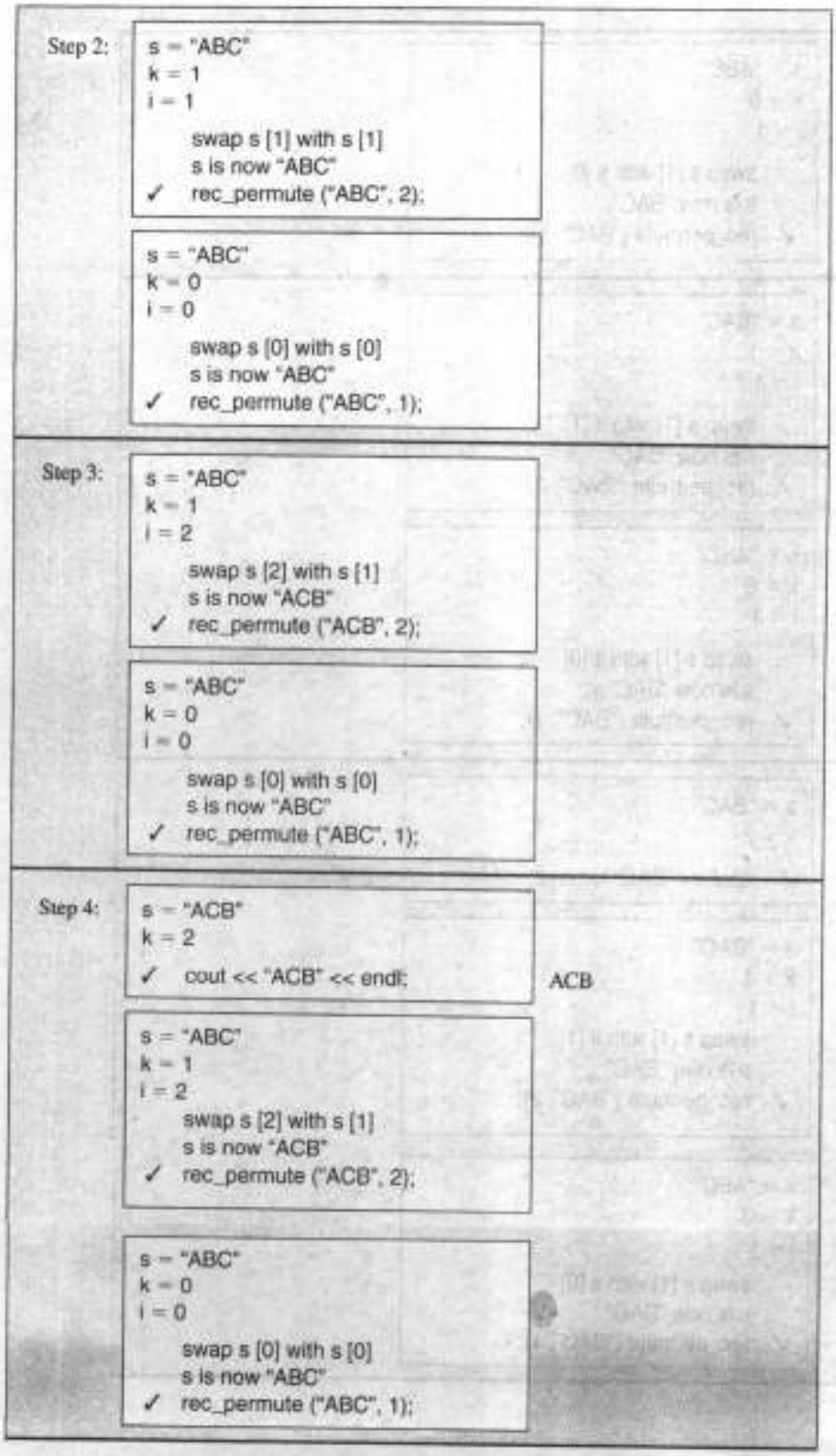
时，在这个调用过程里**for**语句的每次交换调用之后s的值都会改变。s的值的序列是：

"BAC"　　（**for**循环的第一次重复之前）
"BAC"　　（交换s[1]和s[1]之后）
"BCA"　　（交换s[1]和s[2]之后）

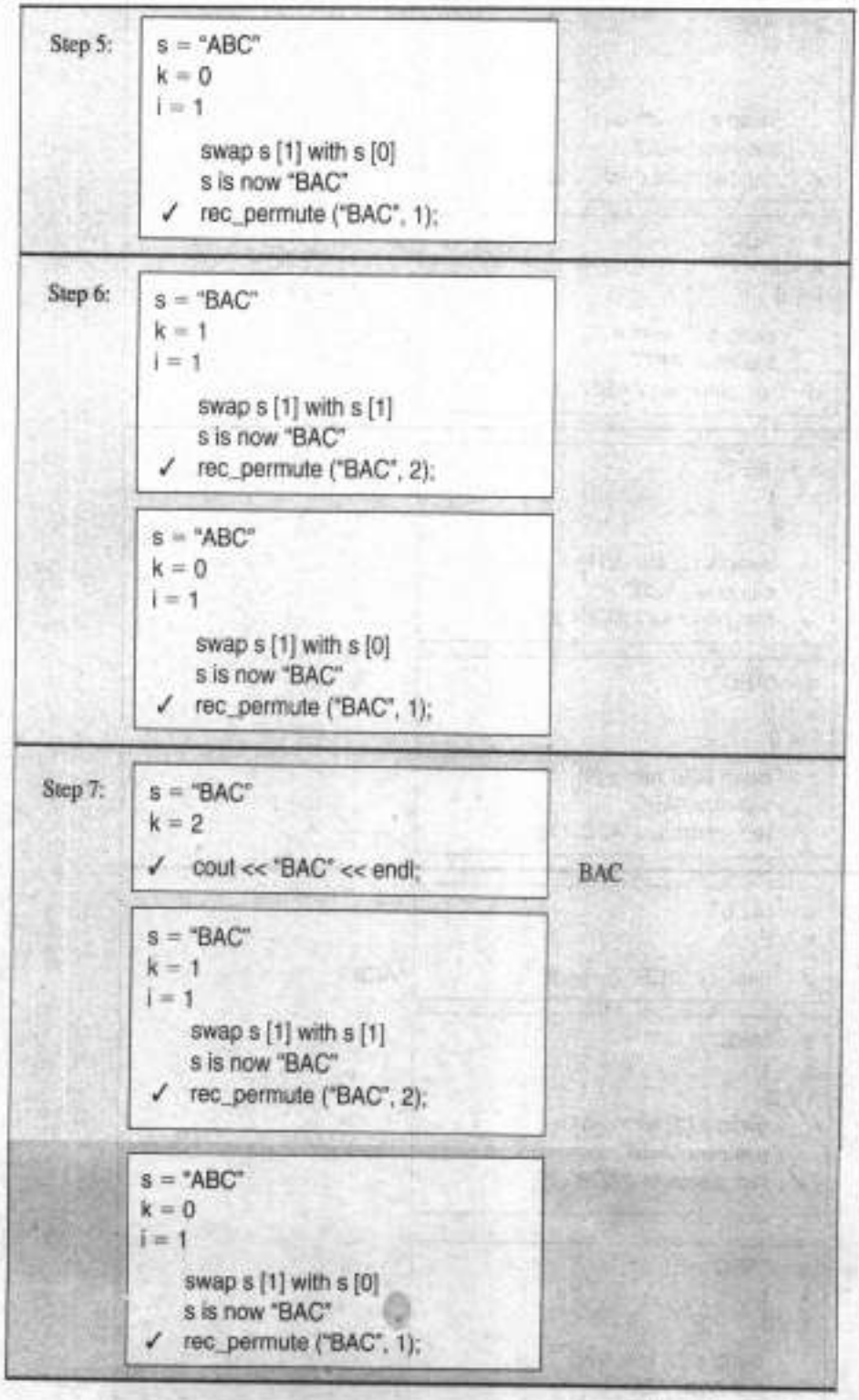
下面列出了初始调用permute（"ABC"）后再调用rec_permute（"ABC"，0)之后的执行结构框架跟踪：



137


Step 2:
s = "ABC"
k = 1
i = 1
swap s [1] with s [1]
s is now "ABC"
✓ rec_permute ("ABC", 2);
s = "ABC"
k = 0
i = 0
swap s [0] with s [0]
s is now "ABC"
✓ rec_permute ("ABC", 1);
Step 3:
s = "ABC"
k = 1
i = 2
swap s [2] with s [1]
s is now "ACB"
✓ rec_permute ("ACB", 2);
s = "ABC"
k = 0
i = 0
swap s [0] with s [0]
s is now "ABC"
✓ rec_permute ("ABC", 1);
Step 4:
s = "ACB"
k = 2
✓ cout << "ACB" << endl;
ACB
s = "ABC"
k = 1
i = 2
swap s [2] with s [1]
s is now "ACB"
✓ rec_permute ("ACB", 2);
s = "ABC"
k = 0
i = 0
swap s [0] with s [0]
s is now "ABC"
✓ rec_permute ("ABC", 1);

138


Step 5:
s = "ABC"
k = 0
i = 1
swap s [1] with s [0]
s is now "BAC"
✓ rec_permute ("BAC", 1);
Step 6:
s = "BAC"
k = 1
i = 1
swap s [1] with s [1]
s is now "BAC"
✓ rec_permute ("BAC", 2);
s = "ABC"
k = 0
i = 1
swap s [1] with s [0]
s is now "BAC"
✓ rec_permute ("BAC", 1);
Step 7:
s = "BAC"
k = 2
✓ cout << "BAC" << endl;
BAC
s = "BAC"
k = 1
i = 1
swap s [1] with s [1]
s is now "BAC"
✓ rec_permute ("BAC", 2);
s = "ABC"
k = 0
i = 1
swap s [1] with s [0]
s is now "BAC"
✓ rec_permute ("BAC", 1);

Step 8:

```
s = "BAC"
k = 1
i = 2
    swap s [2] with s [1]
    s is now "BCA"
✓ rec_permute ("BCA", 2);
```

```
s = "ABC"
k = 0
i = 1
    swap s [1] with s [0]
    s is now "BAC"
✓ rec_permute ("BAC", 1);
```

Step 9:

```
s = "BCA"
k = 2
✓ cout << "BCA" << endl;
```

BCA

```
s = "BAC"
k = 1
i = 2
    swap s [2] with s [1]
    s is now "BCA"
✓ rec_permute ("BCA", 2);
```

```
s = "ABC"
k = 0
i = 1
    swap s [1] with s [0]
    s is now "BAC"
✓ rec_permute ("BAC", 1);
```

Step 10:

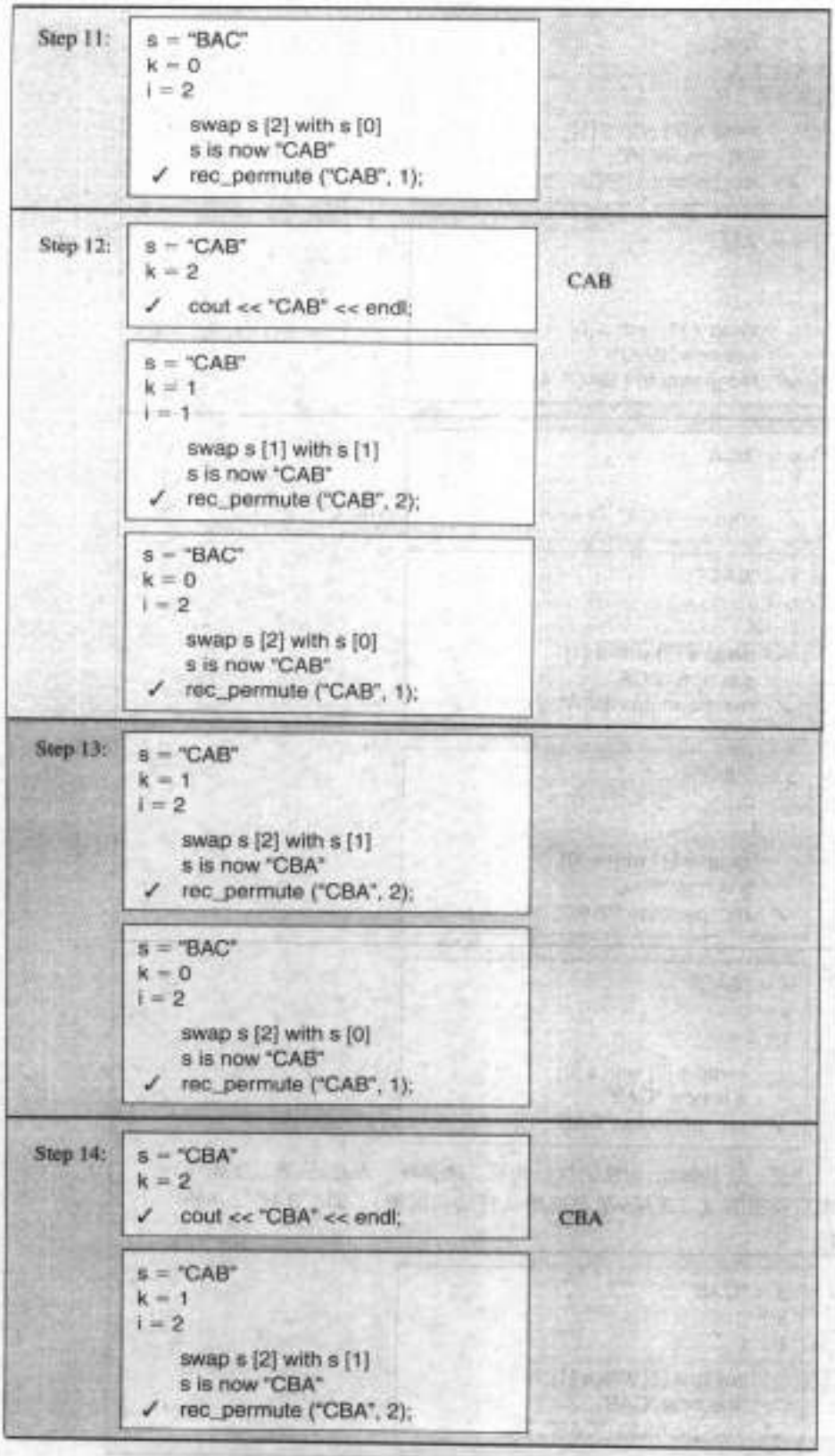
```
s = "BAC"
k = 0
i = 2
    swap s [2] with s [0]
    s is now "CAB"
✓ rec_permute ("CAB", 1);
```

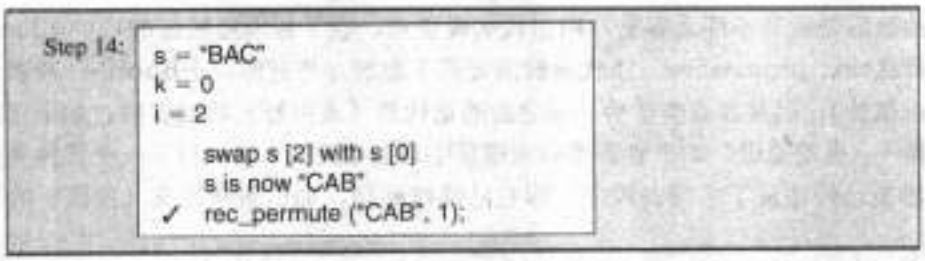
注意：这是**for**语句原始执行的第三次循环，因此s从第二次循环结束后获得的值（见第9步底部的执行结构框架），即“BAC”开始执行。

Step 11:

```
s = "CAB"
k = 1
i = 1
    swap s [1] with s [1]
    s is now "CAB"
✓ rec_permute ("CAB", 2);
```

140


Step 11:
s = "BAC"
k = 0
i = 2
swap s [2] with s [0]
s is now "CAB"
✓ rec_permute ("CAB", 1);
Step 12:
s = "CAB"
k = 2
✓ cout << "CAB" << endl;
CAB
s = "CAB"
k = 1
i = 1
swap s [1] with s [1]
s is now "CAB"
✓ rec_permute ("CAB", 2);
s = "BAC"
k = 0
i = 2
swap s [2] with s [0]
s is now "CAB"
✓ rec_permute ("CAB", 1);
Step 13:
s = "CAB"
k = 1
i = 2
swap s [2] with s [1]
s is now "CBA"
✓ rec_permute ("CBA", 2);
s = "BAC"
k = 0
i = 2
swap s [2] with s [0]
s is now "CAB"
✓ rec_permute ("CAB", 1);
Step 14:
s = "CBA"
k = 2
✓ cout << "CBA" << endl;
CBA
s = "CAB"
k = 1
i = 2
swap s [2] with s [1]
s is now "CBA"
✓ rec_permute ("CBA", 2);



估算时间和存储需求

对时间需求，假设k从0开始，并令n代表s的长度。然后**for**循环重复n次，也就是执行n次置换递归调用。在**for**循环里的每次重复中，都会用k=1再次调用rec_permute，然后进行rec_permute的$n-1$次（额外的）递归调用。因此递归调用次数之和是$n+n(n-1)$。

这个过程继续下去直到$k=n-1$，这时打印输出s。所以递归调用的总次数是$n+n(n-1)+n(n-1)(n-2)+...+n(n-1)(n-2)...3+n!$。 142

由于$n>1$，所以这个累加和的每一项至少都是下一项的一半。从这个和的最右边和次右边项开始，得到（参见A1.3中连乘符号的解释）：

$$\prod_{i=3}^{n} i < (1/2)n!$$

对右边的第二个和第三个元素，

$$\prod_{i=4}^{n} i < (1/2)\prod_{i=3}^{n} i < 1/4\, n!$$

因此得到

$$\prod_{i=4}^{n} i < (1/2^2)\, n!$$

公式右边的指数比左边i的开始索引小2。继续上述模式，可以得到最左边项的公式：

$$n = \prod_{i=n}^{n} i < (1/2^{n-2})\, n!$$

于是

$$n + n(n-1) + n(n-1)(n-2) + \cdots + n(n-1)(n-2)\cdots 3 + n! < (1/2^{n-2})n! + (1/2^{n-3})n! + \cdots + (1/2^2)n! + (1/2^1)n! + n!$$

最后的累加和小于$2n!$；也就是说，递归调用的次数小于$2n!$，所以worstTime(n)是$O(n!)$，并且这是最小值，因为必须输出$n!$个置换。因为在$n>4$时$2^n<n!$，所以任何输出$n!$个数值的函数都一定是指数级的函数，由此推断输出$n!$个数值是非常困难的。

那么存储需求如何呢？不论任何时候，至少都有n个启动的rec_permute。因为s是一个值参，s的n个字符在每个启动中都保存一遍，所以存储需求是n的平方。实际上说，存储需求和时间需求没什么联系。例如，如果n=13，n^2只不过是169，而$n!$却超过了60亿。

这个递归函数的开发并不那么容易。用迭代实现更难，除了标准模板库的<algorithm>文件里的通用型算法next_permutation。迭代函数需要两个参数并将返回一个**bool**值：容器类开头的迭代器（或指针），以及容器类最后一项之后的迭代器（或指针）将被置换。如果可以执行任何一个置换——也就是说，如果容器类尚未违反字母表顺序——就执行下一个置换并返回**true**。如果容器类已经违反了字母表顺序，即它已经被颠倒，那么服从原先（前面）的顺序
143 并返回**false**。

注意string在标准模板库中是一个容器类，因此字符串对象定义了begin和end方法。表4-1显示了s="123"时next_permutation的调用序列，语句如下：

```
while (next_permutation(s.begin(),s.end())
```

为了输出s的全部置换，s必须是**遵守顺序的**：ASCII码的顺序。幸运的是，这时sort通用型算法出现了。下面是使用next_permutation的permute函数一个实现：

```
// 后置条件：输出s的每个置换。
void permute (string s)
{
    sort (s.begin( ), s.end( ));
    cout << endl << s << endl;
    while (next_permutation (s.begin( ), s.end( ))
        cout << s << endl;
} // 函数permute
```

输出了每个置换，共$n!$个，所以**while**循环执行了n!-1次。next_permutation函数的worstTime(n)是一个常量，因此，对permute的这个迭代实现，worstTime(n)是$O(n!)$，这是一个最小值。

4.1节里粗略地给出了递归函数的定义，就是一个函数调用它自身。4.8节将解释这个定义是不充分的。

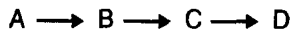
表4-1

调 用 前	调 用 后	返 回 值
123	132	**true**
132	213	**true**
213	231	**true**
231	312	**true**
312	321	**true**
321	123	**false**

4.8 间接递归

C++允许函数间接递归。例如，如果函数A调用函数B，函数B调用函数A，那么A和B都是递归的。

因为间接递归是合法的，所以不能简单地把递归定义成一个函数调用它自身。为了说明"递归"的正式定义，先来定义"**活化**"。若一个函数正被执行或是调用了一个活动的函数，
144 那么它就被称作是"活化"的。例如，考虑下面的函数调用链：



也就是A调用B，B调用C，C调用D。当D执行时，活化的函数是：

D，因为它正被执行

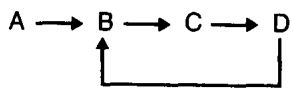
C，因为它调用了D，而D是活化的

B，因为它调用了C，而C是活化的

A，因为它调用了B，而B是活化的

如果一个方法在活化状态下可以被调用，就称该方法是递归的。

现在再来定义“递归”。**递归**函数是指这个函数在活化状态下可以被调用。例如，假设有下面的调用链：



那么B、C和D都是递归的，因为它们都在活化状态下被调用了。

当一个递归函数被调用时，必须保存一些信息，这样在递归调用的执行过程中，这些信息不会被覆盖。当递归调用结束后，这些信息将被恢复。这种保存和恢复以及其他和递归执行相关的工作会耗费一些执行时间和存储空间。4.9节估算了递归的代价，并推测了这个代价的合理性。

4.9　递归的代价

综上所述，每当一个函数调用它自身时都要保存一些信息。这些信息全部都涉及到一个**活动记录**，因为它们都归属于当前活化函数的执行状态，实际上，任何一个函数被调用时都创建一个活动记录；它可以在判断一个给定函数是否为递归（直接或间接）方面，减轻编译器的负担。

基本上，一个活动记录就是一个没有语句的执行结构框架。每个活动记录包含：

1) 返回地址，也就是调用结束后即将被执行的语句地址。

2) 每个值参、形参的值：对应变元的拷贝。

3) 每个引用形参的地址（7.5节有一个示例，它是一个带有引用形参的递归函数）。

4) 函数的其他局部变量的值。

调用结束后，前一个活动记录的信息被恢复并重新执行调用的函数。保存和恢复这些记录将耗费一些执行时间代价，记录本身又占据空间。但是这些代价可以忽略不计，因为编程者开发一个迭代函数将耗费巨大的精力，而递归函数则是更容易接受的。递归方法，像move和rec_permute，相对于它们的迭代版本来说要简洁优美得多。 145

怎样决定是采用递归还是采用迭代呢？如果读者能轻松地开发一个迭代方案，那就用它吧！如果不行，就需要考虑递归能否适用。也就是说，如果问题的复杂情况可以简化成和原先形式相同的简单情况，并且最简单的情况可以直接解决，那么应该试着使用递归函数。

如果迭代函数不容易开发，又适合用递归，那么递归和迭代相比如何呢？最坏情况下，递归版本将会和迭代版本有相同的时间代价和存储代价。在最好情况下，开发递归函数比开发迭代函数耗费的时间要少得多，而且时间和空间性能都差不多。比如，这一章的move、

tryToSolve和rec_permute函数。当然，开发的递归函数的性能也可能很差，像实验9中fib函数的原始版本，它的性能就像迭代函数一样差。

这一章中主要关注于递归到底是什么。后面的第7章里还将讨论这个机制，称作**堆栈**，编译器正是借助它来保存和恢复活动记录的。正如在第3章中看到的，解决问题时，做什么和怎样去做还是有很大分别的。

总结

这一章的目的是熟悉递归的基本思想，这会有助于读者理解第8章和第12章中的递归函数，并有助于在需要时设计自己的递归函数。

递归函数是指这个函数在活化状态下可以被调用。**活化**函数是一个正在执行或是调用活化函数的函数。

递归应当适用于具有下列特性的任何问题:

1) 问题的复杂情况可以简化成和原先形式相同的简单情况。

2) 最简单的情况可以直接解决。

遇到这样的问题，通常可以直接开发一个递归函数。但是可能一个**迭代**函数——由循环组成的——将耗费更少的时间和存储空间。对某些问题，开发迭代函数比递归函数容易些。但
146 一般都是相反的（参阅move、tryToSolve和rec_permute函数）。

无论何时，任何函数（递归的或非递归的）被调用，都将创建一个活动记录，它提供了函数执行的参考结构框架。每个活动记录包含:

1) 返回地址，也就是调用结束后即将被执行的语句地址。

2) 每个值参、形参的值：对应变元的拷贝。

3) 每个引用形参的地址（需要两次存储访问才能访问到相应的变量）。

4) 函数的其他局部变量的值。

活动记录保存了信息以保证递归的可行性，如若不然，函数调用它自身时将会破坏这些信息。当前函数执行结束时，将返回当前活动记录指定的地址。

习题

4.1 下面计算阶乘的函数有什么错误?

```
// 前置条件: n>=0。
// 后置条件: 返回n!
long fact (int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return fact (n+1) / (n+1);
} // fact
```

4.2 开发writeBinary函数的迭代版本。输入一个非负十进制整数给main函数，然后调用这个迭代writeBinary函数，测试函数的性能。

提示　用一个**while**循环生成位的值，并用一个数组保存这些值，然后是包含一个反序步骤的**for**循环输出那些值。

4.3 在初始（不正确的）调用move(0，'A'，'B'，'C')之后，显示汉诺塔move函数的执行结构框架跟踪的前三步。

4.4 实施执行结构框架跟踪，求出下面调用的输出，它是rec_permute函数的**错误**版本：

147

```
rec_permute ("ABC", 0);

// 后置条件: 每次进行s[k...s.length()-1]。
//           置换后输出s。
void rec_permute (string s, unsigned k)
{
    if (k == s.length( ) - 1)
        cout << s << endl;
    else
        for (unsigned i = k; i < s.length( ) ; i++)
        {
            swap (s [i], s [k + 1]);
            rec_permute (s, k + 1);
        } // for
} // rec_permute
```

把这个方法替换成置换项目中的rec_permute方法（见本书网站的源代码）检测输出。

4.5 实施执行结构框架跟踪，求出初始rec_permute("ABC", 0)；调用后的rec_permute函数**错误**版本的输出：

```
// 后置条件: 每次进行s[k...s.length()-1]。
//           置换后输出s。
void rec_permute (string s, unsigned k)
{
    if (k == s.length( ) - 1)
        cout << s << endl;
    else
        for (unsigned i = k; i < s.length( ) ; i++)
        {
            rec_permute (s, k + 1);
            swap (s [i], s [k]);
        } // for
} // rec_permute
```

把这个方法替换成置换项目中的rec_permute方法（见本书网站的源代码）来检测输出。

4.6 研究<algorithm>中的通用型算法next_permutation。用简练的语言描述当容器中包含7，1，4，6，5，3，2时算法的工作流程。

4.7 给定两个正整数i和j，则它们的最大公约数记作

148

$$\gcd(i, j)$$

它是满足($i\%k$=0)且($j\%k$=0)的最大整数k。

例如，gcd(35,21)=7而gcd(8,15)=1。编制一个递归函数返回*i*和*j*的最大公约数。检测函数：向一个main函数输入两个正整数并输出最大公约数。下面是函数接口：

```
//前置条件：i>0,j>0。
//后置条件：返回i和j的最大公约数。
int gcd (int i, int j);
```

提示 根据欧几里得的算法，如果*i*%*j*=0，那么*i*和*j*的最大公约数就是*j*。否则，*i*和*j*的最大公约数就是*j*和(*i*%*j*)的最大公约数。

4.8 **回文**是指这样的字符串，即不管从左向右或是从右向左都是相同的。例如，下面的字符串都是回文：

ABADABA
RADAR
OTTO
MADAMIMADAM
EVE

在这个练习中规定每个字符串都只是由大写字母组成。(在习题4.9里将去掉这个限制。)开发一个递归函数测试回文。函数接口是：

```
//后置条件：如果s[i...j]是一个回文就返回真。否则返回假。
bool isPalindrome(string s, int i, int j);
```

检测函数：向main函数输入一个字符串并判断它是不是回文，并输出结果。

提示 如果i>=j，则s[i...j]是一个（无足轻重的）回文。否则，当且仅当s[i]=s[j]且s[i+1...j-1]是回文，s[i...j]才是回文。

4.9 扩展习题4.8开发的递归函数，使得在测试s是否是回文的过程中，可以忽略不是字母
149 的字符，并且不区分大小写。例如，下面列出的都是回文：

Madam,I'm Adam.
Able was I'ere I saw Elba.
A man.A plan.A canal.Panama!

提示 可以用ctype.h里的toupper函数，将一个小写字母转换成大写字母。这个函数使用了一个参数：一个**char**类型的变量ch。如果ch在'a' ... 'z'之间，那么就返回ch的大写形式的ASCII码值。否则就返回ch自身的ASCII码值。例如，因为'B'的ASCII码值是66，所以

```
cout << toupper ('b') << endl
     << (char)toupper ('b') << endl
     << (char)toupper ('D') << endl
     << (char)toupper ('?') << endl;
```

将输出

```
66
```

```
B
D
?
```

4.10 在第2章的Linked类里，使用递归开发了一个reversePrint方法反序输出一个Linked对象。例如，如果执行

```
Linked<string> myList;

myList.pushFront ("yes");
myList.pushFront ("no");
myList.pushFront ("maybe");
myList.pushFront ("but");
myList.reversePrint( );
```

将输出

```
yes
no
maybe
but
```

4.11 a. 开发一个递归函数power，返回整数指数的数值，接口如下：

```
//前置条件：n>=0。
//后置条件：返回i^n的值。
long power(int i, int n);
```

150

提示 定义$0^0=1$，因此对任何整数i，$i^0=1$。对任意整数i和任意的$n>0$，

$$i^n = i\ (i^{n-1})$$

b. 编制power的迭代实现。

c. 编制power的递归实现，它的worstTime(n)是$O(\log n)$。

提示 如果n是偶数，则power(i,n)=power(i*i,n/2)；如果n是奇数，power(i,n)=i*i^{n-1} = i*power(i*i, n/2)。

在这三个例子中，分别用main函数读入i和n的数值，再输出i^n以测试power函数。

4.12 编写一个递归函数，分析把一定数量的钱变成两角五分的辅币、一角硬币、五分硬币和分币的不同变换方法的数量。例如，如果有17分，那么共有6种转换方式：

1个一角硬币，1个五分硬币和2个分币

1个一角硬币和7个分币

3个五分硬币和2个分币

2个五分硬币和7个分币

1个五分硬币和12个分币

17个分币

下面给出了函数接口：

```
//前置条件：denomination=1（表示分币），2（表示五分硬币），3（表示一角
```

```
//              硬币）或 4（表示两角五分辅币）。
//后置条件: 如果amount<0，那就返回0。否则，返回这些钱换成硬币的不同变换
//              方法的数量。
int ways(int amount, int denomination);
```

为了简化ways函数，使用一个coins函数返回每个硬币的币值。因此，coins(1)返回1，coins(2)返回5，coins(3)返回10，coins(4)返回25。

测试ways和coins函数：输入一个钱数，再输出将它变成两角五分的辅币、一角硬币、五分硬币和分币的不同变换方法的数量。

提示 将amount换成不大于两角五分辅币的变换方法的数量等于将amount−25变换成
151 不大于两角五分辅币的方法数，加上将amount变换成不大于一角硬币的方法数。

4.13 证明：递归binary_search函数在查找成功的情况下，averageTime(n)和n是成对数关系的。

提示 令n代表将搜索的数组的长度。因为平均调用次数是n的非递减函数，所以只要证明对等于2的幂的n论证结论为真，那么小于它的也同样成立。因此假设：

对某正整数k，$n=2^{k-1}$

在一个成功的查找中，

如果查找的元素在搜索区域的中间，调用一次就结束。

如果查找的元素在搜索区域的四分之一或四分之三处，调用两次结束。

如果查找的元素在搜索区域的八分之一、八分之三、八分之五或八分之七处，调用三次结束。

依次类推。

所有成功查找的调用总数是：

$$(1 \cdot 1) + (2 \cdot 2) + (3 \cdot 4) + (4 \cdot 8) + (5 \cdot 16) + \cdots + (k \cdot 2^{k-1})$$

平均调用次数以及averageTime(n)可以估算为这个和除以n。现在使用习题A1.3的结果和前提$k=\log_2(n+1)$可以论证。

4.14 怎样修改递归binary_search函数，可以使它在每次调用中只做一次比较？

提示 见实验11。

4.15 修改Maze应用，使得终端用户可以输入一个文件名保存迷宫。

4.16 修改Maze应用，使它允许斜向移动。

152 4.17 利用数学归纳法原理（附录1），证明汉诺塔范例中的move函数是正确的。

提示 对n=1,2,3,...，令S_n为语句move(n,orig,dest,temp)，输出n个盘子从任意的杆orig移动到另一根杆dest上的步骤。

a. 基本情况。证明S_1为真。

b. 归纳分析。令n是大于1的整数并假设S_{n-1}为真。然后证明S_n为真。根据move函数的代码，调用move(n,orig,dest,temp)时会发生什么？

4.18 汉诺塔应用的move函数的执行跟踪中，步骤数等于move递归调用次数加上输出语

句的数量。因为每次调用move(包含了n=1时的调用)都有一个输出语句，所以move递归调用的次数总是比输出语句数少1，而输出语句的数量是2^n-1。例如，在本章所示的执行跟踪里，n=3，所以步骤数是6+7（回忆一下，从第0步开始，最后是第12步）。当n=4时，执行跟踪共有多少步呢？通常情况下，步骤数是n的函数，那么总共多少？ 153

编程项目4.1：汉诺塔的迭代版本

编制汉诺塔游戏中move函数的迭代实现。输入盘子数量到main函数，然后调用move函数，测试该方法。

提示 *如果能回答下面的三个问题，就可以得到每一步上正确的移动：*

1) *即将移动哪个盘子？* 为了回答这个问题，设立一个n位的计数器。令n是盘子的数量，并将计数器全部清零。例如，如果n=5，就从00000开始。

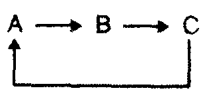
每一位对应一个盘子：最右边的位对应盘1，次右边的位对应盘2，依次类推。在每一步中，移动最右边的0位对应的盘子，因此第一个移动的盘子应当是盘1。

移动一个盘子之后，应按如下方式增加计数：自右向左反转位（0变成1，1变成0），直到遇到一个0。例如，前几次增加计数和移动如下：

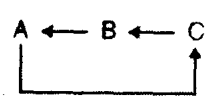
00000//移动盘1
00001//移动盘2
00010//移动盘1
00011//移动盘3
00100//移动盘1
00101//移动盘2

在做了31次移动之后，计数器全部位都是1，因此不再需要也不可能做任何移动。通常需要2^{n-1}次移动和计数。 154

2) *盘子将向哪儿移动？* 如果n是奇数，就将奇数号的盘子按顺时针移动：



而将偶数号的盘子按逆时针移动：



如果n是偶数，就将偶数号的盘子按顺时针移动，而将奇数号的盘子按逆时针移动。

如果用0、1、2取代杆的编号A、B、C，那么可以很容易地用模运算实现移动。也就是，如果当前位置是杆k，那么

$$k = (k+1)\%3$$

实现了一个顺时针移动，而

$$k=(k+2)\%3$$

实现了一个逆时针移动。输出时再将它转换回字符:

```
cout<<char(k+'A');
```

3) *现在盘子在哪里？*跟踪盘1的去向。如果计数器指示盘1即将被移动，就用问题2的答案来移动盘子。如果计数器指示即将移动的盘子不是盘1，那么问题2的答案将解答盘子的当前位置。为什么会这样？因为这个盘子不能移动到盘1的上面并且现在不能从盘1所在的杆上
155 移走。

编程项目4.2：八皇后问题

将八个皇后放在棋盘上，并保证每个皇后都不会被其他皇后攻击到，编写并验证这个程序。

分析

棋盘有八行八列。在象棋游戏里，皇后是功能最强的：她可以攻击和她同行、和她同列或者在她的对角线上的任何棋子。见图4-10。

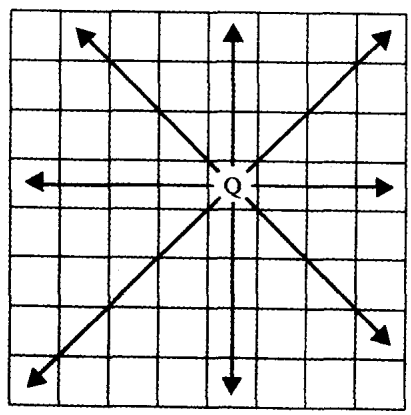


图4-10 象棋中可以被皇后攻击到的位置。箭头指示了
156 棋盘中心的皇后（Q）可以攻击到的位置

这个问题不需要输入。输出是放置好八个皇后之后的棋盘状态。例如:

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

输出时不需要显示线条。

提示 每一行、每一列都应恰好有一个皇后。从（0，0）开始放置一个皇后，即第0行第0列；然后在每列上放置一个皇后。一个有效的位置应当和前面放置的皇后不在同行、同列或同一对角线上。QueensIterator构造器将前进到第0行的下一列上。**operator++（int）**将前进到同一列的下一行。因此第一次调用tryToSolve时，选择是：

（0，1）//无效：和（0，0）处的皇后在同一行
（1，1）//无效：和（0，0）处的皇后在同一对角线
（2，1）//有效
再次调用tryToSolve时，选择是：
（0，2）//无效：和（0，0）处的皇后在同一行
（1，2）//无效：和（2，1）处的皇后在同一对角线
（2，2）//无效：和（2，1）处的皇后在同一行
（3，2）//无效：和（2，1）处的皇后在同一对角线
（4，2）//有效

157

编程项目4.3：马的遍历问题

编写程序实现马对棋盘的遍历，并验证该程序。

分析

一个棋盘有八行八列。从当前位置出发，一个马的移动必须跨越两行一列或是两列一行。例如，图4-11显示了（5，3）位置上（即第5行第3列）的马的所有的合法移动。

	0	1	2	3	4	5	6	7
0								
1								
2								
3			K7		K0			
4		K6				K1		
5				K				
6		K5				K2		
7			K4		K3			

图4-11 对坐标（5，3）上的马而言，合法的移动就是K0到K7标记的位置

简化一下问题，假定马从（0，0）开始移动。按照顺序尝试图4-11所示的全部移动。也就是从位置（行，列）出发，依次尝试：

（row－2,column+1）
（row－1,column+2）

(row+1,column+2)
(row+2,column+1)
(row+2,column-1)
(row+1,column-2)
(row-1,column-2)
(row-2,column-1)

158 图4-12显示了前几个移动。

	0	1	2	3	4	5	6	7
0	1				3			
1			2				4	
2								10
3								5
4							9	
5							6	
6						8		
7								7

图4-12 马从坐标（0，0）开始的前几个以及重复图4-11所示顺序的有效移动。方框里的整数代表移动的次序

图4-12里从（0，0）开始的9个移动都没有发生回溯。实际上前36次移动都不需要回溯。但是发生回溯的次数是非常惊人的：超过300万。根据这种顺序，迭代得到的答案是

	0	1	2	3	4	5	6	7
0	1	38	55	34	3	36	19	22
1	54	47	2	37	20	23	4	17
2	39	56	33	46	35	18	21	10
3	48	53	40	57	24	11	16	5
4	59	32	45	52	41	26	9	12
5	44	49	58	25	62	15	6	27
6	31	60	51	42	29	8	13	64
7	50	43	30	61	14	63	28	7

注意第37次移动，位置（1，3）没有选择第一种移动方式——到位置（3，2），也没有选择第二种。这两个选择都通向死胡同，将发生回溯。第三种选择——到（0，1），是正确的答案。

系统测试1 线条是不需要输出的。请输入开始的行和列：

159

0 0

答案是:

	0	1	2	3	4	5	6	7
0	1	38	55	34	3	36	19	22
1	54	47	2	37	20	23	4	17
2	39	56	33	46	35	18	21	10
3	48	53	40	57	24	11	16	5
4	59	32	45	52	41	26	9	12
5	44	49	58	25	62	15	6	27
6	31	60	51	42	29	8	13	64
7	50	43	30	61	14	63	28	7

系统测试2 请输入开始的行和列:

3 5 160

答案是:

	0	1	2	3	4	5	6	7
0	33	42	35	38	31	40	19	10
1	36	57	32	41	20	9	2	17
2	43	34	37	30	39	18	11	8
3	56	51	58	21	28	1	16	3
4	59	44	29	52	47	22	7	12
5	50	55	46	27	62	15	4	23
6	45	60	53	48	25	6	13	64
7	54	49	26	61	14	63	24	5

获得这个答案需要1100万次回溯。从某些位置出发，比如（0，1），需要超过600万次的回溯。但从任意一个位置出发，都可以找到答案；参阅

http://www.wealth4freedom.com/WORLDNEWSSTAND/knightstour.htm

尽管看起来几乎是立刻得到了答案，但这也并不表示解答过程是没有回溯的。对每个起始位置而言，通过回溯获得的解答都保存在一个文件里。 161

第5章　向量和双端队列

本章将开始标准模板库的数据结构的学习。正如在第2章中提出的，标准模板库中提供的每个数据结构都是某些容器类的方法接口的集合。和用户的观点相适应的是，标准模板库没有指定任何实现细节；5.2.5节和5.4.1节中将考虑可能的实现。

在顺序容器中，元素从第1个到最后一个是连续存储的。

这里介绍的容器类有vector和deque类。这些类以及第6章中的list类称作**顺序**容器类。在顺序对象中，可以将项看作是从第1个到最后一个连续存储的。例如，一个顺序对象pets可能有按如下方式安排的项："dog"，"cat"，"iguana"，"gerbil"。这里"dog"是第一项，而且"gerbil"是最后一项。在这个例子中，项不是按照字母顺序排序的。

向量是一维数组的类版本。和数组相似，向量中的项是连续存储的。但是和数组不同的是，向量的大小是在程序运行过程中根据需要自动增加的。在了解向量的简便和强大功能后，用户可能再也不会为数组费心了！双端队列也和数组相似，它允许常数时间内对任意项的随机访问。但是从双端队列的前面进行插入或删除时，averageTime(n)是常数，而在数组或向量里是和n成线性关系的。本章中讲述的向量和双端队列的应用是在公钥加密算法领域。

目标

1) 理解标准模板库的主要组件之间的关系：容器类，迭代器和通用型算法。
2) 比较用户和开发者对vector类以及deque类的看法。
3) 能够判断什么时候使用向量或双端队列更好，什么时候使用数组更好。
4) 比较用户和开发者对VeryLongInt类的观点。

5.1　标准模板库

面向对象编程的一个主要目标是代码的重用，例如通过继承。一般来说，人们更愿意使用已经开发好的类而不是白手起家开发一个项目。在容器类中有一个特殊情况，即验证过的库、高效率的容器类可以显著降低项目开发时间。标准模板库（Stepanov and Lee,1994）正是提供了一个这样的库。标准模板库的三个主要的组件是：

1) 模板容器类的集合。
2) 通用型算法，也就是模板函数的集合。
3) 迭代器种类也就是系列迭代器类的集合。

在标准模板库中，通用型算法通过迭代器操作容器。

正如在实验7中所看到的，通用型算法通过迭代器操作容器。例如，通用型算法find在容器中搜索一个给定项。但是这个搜索并不基于任何容器类自身的具体细节，因为如果那样，find的使用就被局限在这个容器类中了。而且，在另一个代码重用的范例中，任何关联迭代器类属于InputIterator类的容器中都可以使用find算法搜索。实际上，迭代器从它的容器中抽象出访问该容器所有项时所必需的信息。

这里没有任何对标准模板库实现的规定。

标准模板库是美国国家标准化组织认可的官方C++语言的一部分。但这只意味着规定了方法和函数的接口。这里没有任何对标准模板库实现的规定：只要满足接口，开发者可以自由地以任何方式实现类和通用型算法。在惠普研究实验室中由Stepanov、Lee和其他人开发的原始实现是现今所用的很多实现的基础：Microsoft的Visual C++、Inprise的C++Builder和Metrowork的CodeWarrior等。下面将学习该原始实现，另外还提出了另一个可行的实现。

即将学习到的标准模板库中的第一个容器类是vector类，它基本上就是数组的类版本。首先定义向量是什么，观察少许数量众多的vector方法的接口，并对比向量和数组，然后给出
164 vector类的标准实现的概观，最后开发一个使用向量的应用。对deque类也采用相同的方法。

5.2 向量

向量是一个项的有限序列，满足：

1) 给出序列中任何项的下标，就可以在常数时间内访问或修改该下标对应的项。

2) 在序列尾部进行的插入平均说来只耗费常数时间，但是worstTime(n)是$O(n)$，其中n代表序列中项的数量。

3) 对序列尾部进行的删除，worstTime(n)是常数。

4) 对任意的插入和删除，worstTime(n)和averageTime(n)都是$O(n)$。

vector类以及标准模板库中其他所有的类都是模板化的。也就是说，项的类型可以是一个基本类型，比如**int**或**double**；也可以是一个类，如string类或第1章中介绍的Employee类。例如，可以按如下方式定义字符串的一个空向量：

```
vector<string>fruits;
```

在向量中允许重复项。因此，如果把“oranges”、“apples”、“grapes”和“apples”插入fruits，那么向量将拥有四个项。项“oranges”位于下标0处，“apples”位于下标1处，“grapes”位于下标2处，“apples”位于下标3处。这些项不是按照字母顺序排序的。实际上，一个向量中的项不一定是可以比较的。例如，假设向量是一个**文本**，即行序列；那么称一行“小于”另一行是没有意义的。当然，仍然可以比较行的下标，并称当前行的下标小于其他某些行的下标。

vector类有两个模板参数：

```
template<class T, class Allocator=allocator>
```

模板参数T代表项的类型。Allocator参数涉及了内存分配模型（例如，一个指向T的指针是否缺省地定义成T*，或定义成特殊些的，像T_far*）。处理多种分配模型所需要的灵活性是很多标准模板库实现比较复杂的根源。为简单起见，采取allocator类给出并在<defalloc>中定义的缺省分配模型。根据参考标准（Musser and Saini, 1996, p.274）中的建议，我们“省略Allocator参数的进一步探讨”。实际上，

> 今后，所有的声明和定义都将采取缺省分配模型。

5.2.1节从用户的角度开始vector类的设计，即方法接口的设计。在vector类中有50多个方

法，因此应该集中精力处理在应用中最有可能用到的方法。方法接口并不包括任何具体的字 165
段或方法定义。这种分离是数据抽象的基本原理。

每个方法的时间需要用大O表示法指定，因为只需要确定一个上界：方法的特定实现可能降低这个上界。如果没有给出方法的时间估算，可以假设worstTime(*n*)是常数，其中*n*代表向量中项的数量。如果方法的平均时间花费估算和最坏时间花费估算相同，就只给出最坏时间估算。

5.2.1 vector类的方法接口

下面列出了vector中少数几个最常用方法（参见表5-1）的接口。

表5-1 一些vector方法的简单描述（假定定义为：vector<**double**>::iterator itr;）

方 法	功 能
vector<**double**>weights	weights是一个空的向量
weights.push_back(107.2)	在向量weights的尾部插入107.2
weights.insert(itr,125.0)	在itr所在位置插入125.0；weights中插入点之后的项依次向后移动一个位置；返回一个位于新插入项上的迭代器
weights.pop_back()	删除weights中尾部的项
weights.erase(itr)	删除itr所在位置上的项；高位的项依次向前移动；令所有指向擦除位置后面位置的迭代器失效
weights.size()	返回weights中项的数量
weights.empty()	如果weights中没有项就返回真；否则返回假
weights[3]=110.5	将weights中下标3处的项替换成110.5
itr=weights.begin()	将itr置于weights开头项的位置
itr==weights.end()	如果itr恰好位于weights最后一项之后就返回真；否则返回假
weights.front()=105.0	将weights中下标0处的项替换成105.0

166

1. //后置条件：这个向量为空；也就是说，它当中不包含任何项。
vector();

例 下面是创建空的fruits向量的定义以及测试数据：

```
vector<string>fruits;
vector<Employee>employees;
vector<int>scores;
```

注意 *这也是一个构造器，称作**拷贝构造器**，它将一个向量初始化成其他向量的拷贝。方法头是：*

```
vector(const vector<T>& x);
```

例如，如果已经在示例中构造了向量fruits，可以编写：

```
vector<string>newFruits(fruits);
```

那么就定义了newFruits，同时把它也初始化成了fruits的拷贝。

2. //后置条件：在这个向量的尾部插入x的拷贝。averageTime(n)是常数，worstTime(n)是O(n)，
//　　但是对n次连续的push_back调用，worstTime(n)只是O(n)。
void push_back(const T& x);

例 下面是创建四个项的向量的代码:

```
vector<string> fruits;

fruits.push_back("oranges");
fruits.push_back("apples");
fruits.push_back("grapes");
fruits.push_back("apples");
```

向量fruits现在包含了下列顺序的项:

"oranges", "apples", "grapes", "apples"

注意 接口没有规定任何实现细节。在一个有代表性的实现中，第一次调用push_back时为向量分配一个存储块（比如，1K字节）。随后的插入将填满存储向量的这个块。如果存储块已满不能进行插入，就重新分配，并将整个向量拷贝到大小是当前块两倍的新块中。那么指向前面块的所有的迭代器和引用就都失效了。

167

3. //前置条件: 迭代器位于向量头和向量尾后的下一个位置之间。
   //后置条件: x的拷贝放入迭代器位置所在的位置。调用前，每个大于等于该位置下标的位置依次向后移动。返回位于新插入元素位置的迭代器。worstTime(n)是O(n)。
   iterator insert(iterator position, **const** T& x);

例 假设fruits是push_back示例中的向量，它有如下顺序的项:

"oranges", "apples", "grapes", "apples"

如果迭代器itr位于下标2的 "grapes" 项的位置上，那么

```
vector<string>::iterator new_itr=fruits.insert(itr,"kiwi");
```

将使fruits变成

"oranges", "apples", "kiwi", "grapes", "apples"

并且迭代器new_itr位于下标2的 "kiwi" 项的位置上。itr失效；也就是说，不能确认itr的位置，甚至不能确认itr是否还在fruits中某项的位置上。

注意1 如果一个插入导致重新分配（参见push_back方法的注意），旧的迭代器和引用都将失效。如果不发生重新分配，那么只有插入点上及之后的迭代器和引用失效。

注意2 push_back方法是insert的特例。

4. //前置条件: 向量非空。
   //后置条件: 这次调用前向量尾部的项被删除。
   void pop_back();

例 假设fruits是insert方法示例中的向量，它有如下顺序的项:

"oranges", "apples", "kiwi", "grapes", "apples"

如果消息是

168

```
fruits.pop_back();
```

那么fruits将有下列顺序的项:

“oranges”，“apples”，“kiwi”，“grapes”

5. //前置条件：迭代器位置位于向量中某一项的位置上。
//后置条件：这次调用前位于迭代器位置上的项被删除。调用前每个下标大于迭代器位置
// 下标的项依次向前移动。worstTime(n)是O(n)。
void erase(iterator position);

例 假设fruits是pop_back方法示例中的向量，它有如下顺序的项：

“oranges”，“apples”，“kiwi”，“grapes”

如果迭代器itr位于项“apples”的位置上并且发送消息

fruits.erase(itr);

那么fruits将包含下列顺序的项：

“oranges”，“kiwi”，“grapes”

注意1 erase方法使擦除点之后的所有迭代器和引用都失效。

注意2 pop_back方法是erase的特例。

注意3 这里有一个erase方法的版本，它有两个迭代器参数——first和last。所有在first（包括first）和last（不包括last）之间的项将被擦除。例如，假设fruits是有如下顺序项的向量：

“oranges”，“apples”，“kiwi”，“grapes”

如果迭代器itr1位于项“apples”位置上，itr2位于项“grapes”位置上，并发送消息

fruits.erase(itr1,itr2);

那么fruits将包含如下顺序的项：

“oranges”，“grapes”

169

项“grapes”未被擦除，这是因为erase调用中的第二个变元itr2是即将被擦除的最后一项之后的项。这个erase方法的时间代价和向量中last之后的项的数量成正比，因为那些都是必须移动的项。

6. //后置条件：返回向量中项的数量。
unsigned size() **const**;

例 假设fruits是有如下顺序项的向量：

“oranges”，“kiwi”，“grapes”

如果有

cout<<fruits.size();

那么输出将是

3

注意1 在ANSI标准C++中，返回类型是size_type。在stddef.h中有

```
typedef unsigned size_t;
```

最后，通过缺省分配器，

```
typedef size_t size_type;
```

因此返回类型是**unsigned**。

注意2 为了求出在重新分配发生前可以插入多少项，配合使用size方法和capacity方法。capacity方法的接口在编程项目5.2中给出了，它返回重新分配发生前向量中可以存储的项的数量。例如，如果vec是一个向量对象，

```
cout<<vec.capacity()-vec.size();
```

将输出重新分配发生之前还可以插入的项的数量。

7. //后置条件：如果这个向量中不包含任何项就返回真。否则，返回假。
170 **bool** empty() **const**;

例 假设fruits是有如下顺序项的向量：

“oranges”，“kiwi”，“grapes”

如果有

```
while(!fruits.empty())
    fruits.pop_back();
```

循环将执行3次，然后fruits为空。

8. //前置条件：0<=n<向量中项的数量。
//后置条件：返回对向量的从开头算起的第n项的引用。
T& **operator**[](**unsigned** n);

例1 假设fruits是有如下顺序项的向量：

“oranges”，“kiwi”，“grapes”

如果有

```
cout<<fruits[1];
```

那么输出将是

```
kiwi
```

例2 假设fruits是有如下顺序项的向量：

“oranges”，“kiwi”，“grapes”

如果有

```
fruits[1]="limes";
```

那么fruits将包含如下顺序的项：

“oranges”，“limes”，“grapes”

例3 可以使用下标运算符迭代通过一个向量。例如，假设fruits是有如下顺序项的向量：

“oranges”，“limes”，“grapes”

如果有

171

```
for(int i=0;i<fruits.size();i++)
    cout<<fruits[i]<<endl;
```

那么输出是

```
oranges
limes
grapes
```

注意1 后置条件没有指出当变元值小于0或大于等于向量中项的数量时会怎样。在调用方法前保证前置条件为真是vector类的**用户**的职责。

注意2 这个方法返回一个**引用**，也就是一个地址。因此可以修改那个地址的内容，正如在这个方法接口的例2中所做的一样。

注意3 使用这个运算符不会改变向量的大小。

9. //后置条件：返回位于向量开头的迭代器。
   iterator begin();

例 假设fruits是有如下顺序项的向量：

“oranges”，“kiwi”，“grapes”

172

如果有

```
vector<string>::iterator itr=fruits.begin();
```

那么itr就位于“oranges”项的位置上。

10. //后置条件：返回恰好位于向量最后一项之后位置的迭代器。
    iterator end();

例 假设fruits是有如下顺序项的一个向量：

“oranges”，“kiwi”，“grapes”

如果有

```
vector<string>::iterator itr=fruits.end();
```

那么itr就恰好位于“grapes”项之后。因此消息

```
fruits.insert(itr, "lemons");
```

将和下面的消息

```
fruits.push_back("lemons");
```

作用相同，也就是，fruits将包含如下顺序的项：

“oranges”，“kiwi”，“grapes”，“lemons”

注意 如果向量为空，那么begin()返回的迭代器就等于end()返回的迭代器。

```
11. //前置条件: 这个向量非空。
    //后置条件: 返回对这个向量开头项的引用。
    T& front();
```

例 假设fruits是有如下顺序项的向量:

"oranges", "kiwi", "lemons", "grapes"

如果有

```
cout<<fruits.front();
```

输出就是

```
oranges
```

注意1 这个方法可以用来替换向量中开头的项。例如，假设写出

```
fruits.front()="pears";
```

fruits中开头的项就变成了"pears"。这条语句和

```
fruits[0]="pears";
```

是等价的。

注意2 这里有一个相似的方法返回对向量尾部项的引用:

173

```
T& back();
```

在方法接口8的例3中，列举了使用下标运算符**operator**[]迭代通过向量的方法。5.2.2节描述了vector类的iterator类，它提供了另一种（但不是等价的）迭代工具。

5.2.2 向量迭代器

与vector类关联的迭代器实际上是指针，因此任何在数组中可以用指针实现的问题也可以在向量中用迭代器实现。特别是下面的向量－迭代器（即指针）运算符: ++, +, *, ! =和==。例如，假设fruits是有如下顺序项的向量:

"oranges", "kiwi", "grapes", "lemons"

可以按下面的方式输出fruits里的全部的项:

```
vector<string>::iterator itr;
for (itr = fruits.begin( ); itr != fruits.end( ); itr++)
      cout << *itr << endl;
```

在方法接口8的例3中，可以看到另一种使用下标运算符**operator**[]解决同一任务的方法:

```
for (unsigned i = 0; i < fruits.size( ); i++)
      cout << fruits [i] << endl;
```

一般来说，如果itr是向量vec中某一项位置上的向量迭代器，那么*itr引用了和vec[itr-vec.begin()]相同的项。同理，如果vec是一个向量，那么可以得到数组指针定理的向量-迭代器推论。

向量-迭代器推论

vec[n]等价于*(vec.begin()+n).

因为一个向量-迭代器可以直接访问向量中的任一项，所以向量-迭代器是属于随机访问迭代器类的。

下面通过一个小程序阐述了比较关心的几个向量方法。通过调用push_back生成一个随机薪水向量，用通用型算法accumulate（见第3章）计算这些薪水的总和，然后输出两次超出平均值的薪水，一次使用基于下标的循环，一次使用基于迭代器的循环。

```
#include <vector>
#include <iostream>
#include <string>
#include <stdlib>
#include <numeric>                                                    174

using namespace std;

int main( )
{
    const string PROMPT = "Please enter the number of salaries: ";
    const string ERROR_MESSAGE =
        "The number of salaries should be > 0.";
    const double SALARY_FACTOR = 5.00; // 让薪水为现实的
    const string AVERAGE = "The average salary is ";
    const string ABOVE = "The above-average salaries are: ";
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";
    vector<double> salaries;
    vector<double>::iterator itr;
    int n; // 薪水数量
    cout << PROMPT;
    cin >> n;
    if (n <= 0)
        cout << ERROR_MESSAGE << endl;
    else
    {
        for (int i = 0; i < n; i++)
            salaries.push_back (rand( ) * SALARY_FACTOR);
        double salarySum = accumulate (salaries.begin( ),
                                       salaries.end( ), 0.00);
        double averageSalary = salarySum / n;
        cout << endl << AVERAGE << averageSalary << endl;
```

```
            cout << endl << endl << ABOVE << endl;
            for (int i = 0; i < n; i++)
                if (salaries [i] > averageSalary)
                    cout << salaries [i] << endl;

            cout << endl << endl << ABOVE << endl;
            for (itr = salaries.begin( ); itr != salaries.end( ); itr++)
                if (*itr > averageSalary)
                    cout << *itr << endl;
175     } // n > 0
        cout << endl << endl << CLOSE_WINDOW_PROMPT;
        cin.get( );
        cin.get( );

        return 0;
    } // main
```

到了这里，读者可能认为向量是一个带有方法的、自动调整大小的数组。大概就是这样！在5.2.3节中还会稍许扩展一下这个观点，同时比较向量和第2章的Linked类。

5.2.3 向量和其他容器的对比

向量容器和数组相比较如何呢？向量相对数组的最大优势是，vector方法是已经开发好的，而数组的用户必须创建维护数组时所必需的代码。例如，为了在数组的任意位置插入或擦除，必需编写代码打开或关闭空间。vector类的insert和erase方法可以自动地进行处理，而push_back和insert方法可以在向量空间不足时自动调整向量的大小。

向量和数组都能调用标准模板库的通用型算法，向量和数组也都允许通过下标运算符访问或修改项。数组惟一比向量好的地方是它可以快速初始化。例如，可以定义一个数组并马上进行初始化：

```
string[] words={"yes","no","maybe"};
```

到目前为止所接触到的另一个容器类是第2章中非常简单的Linked类。对于在Linked容器开头进行的插入，worstTime(*n*)是常数，而向量中的插入只有在平均情况下是常数。在最坏情况下，当大小增加时，必须把所有的项拷贝到一个更大的向量中，但这是极少发生的。向量的一个优点是它带有方法可以在容器的任意位置插入和删除项；而Linked类中没有这样的方法。

最后，向量-迭代器是随机访问迭代器；也就是说，一个向量-迭代器能够以常数时间访问向量的任意项。链式迭代器的功能没有这么强大；访问Linked容器的任意项，它的worstTime(*n*)是与距容器头的距离成线性关系的。链式迭代器属于**前向迭代器**类别：这一类别的迭代器只可以通过加1操作前进通过一个Linked容器。其中没有减运算符，也没有标量加法。

惠普公司提供的vector类的原始实现细节是很容易令人迷失的。这个实现不仅像预期的一
176 样效率高，而且很简练。它还有超出想像的一般性：不局限于单个内存分配模型。5.2.4和5.2.5节给出了一个路线图来理解它的实现，其版权声明为：

```
Copyright(c)1994
Hewlett Packard Company
Permission to use, copy, modify, distribute, and sell this software and its documentation for
```

any purpose is hereby granted without fee, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

5.2.4 vector类可能的字段

没有标准模板库的单个实现。标准参考见《*STL Tutorial and Reference Guide*》(Musser and Saini, 1996)，其中提供了方法接口和注意事项以及如何使用标准模板库的示例。只要满足方法接口，实现者有极大的自由去选择字段和方法接口。这一节的概要是基于第一个这样的实现的，它来自于惠普研究实验室。

对vector类，需要回答的第一个问题是，“项将存在哪里？”需要一个连续的存储结构来支持随机访问。一个数组！因此得到一个指针——start，它存储数组第一个位置的地址。

还有另外两个指针：

finish，它指向紧随向量最后一项之后的位置。

end_of_storage，它指向紧随数组占据的最后一个空间之后的位置。

编译器可能为这些字段使用不同的标识符，或使用不同涵义的令人信服的字段。

5.2.5 vector类的一个实现

现在已经指定了字段，可以直接写出几个方法的定义。缺省构造器不为数组分配存储块，而是将全部三个指针字段设置成NULL，也就是0。顺便说一下，begin()方法只是返回start，end()方法只是返回finish，而size()方法返回finish-start。下面的代码将产生预期的输出0:

```
vector<double>weights;
cout<<weights.size();
```

177

当第一个项插入向量时（使用push_back方法或insert方法），将分配堆中的一个存储块。这个块的大小随编译器不同而异。具体地说，假设分配了1024个字节，并且一个**double**占据8个字节。如果第一次插入消息是

```
weights.push_back(7.3);
```

那么就分配128个double型数组，如图5-1所示。

如图5-1所示，start和finish字段分别指向数组的第一个和第二个单元，并且end_of_storage字段指向紧随数组之后的第一个单元。消息

```
weights.size()
```

将返回值1，即finish-start。并且消息

```
weights.begin()
```

和

```
weights.end()
```

将分别返回位于下标0和下标1单元的迭代器（即指针）。

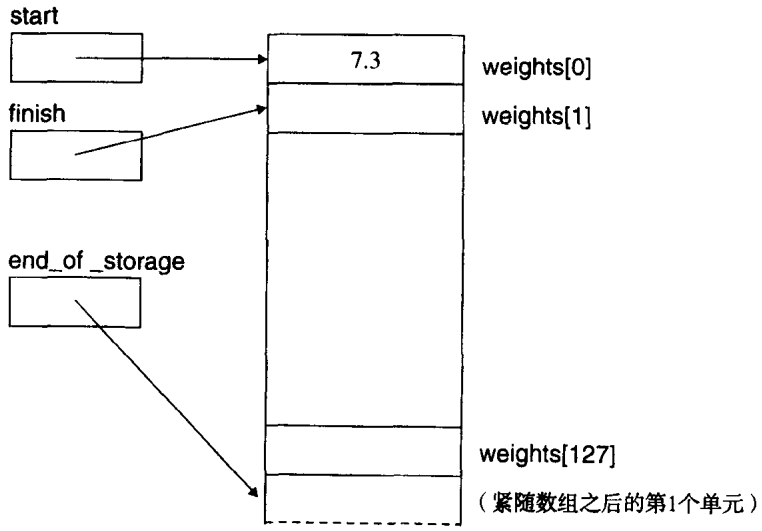


图5-1 使用push_back方法将7.3插入之后的vector<double>weights

end_of_storage在这个实现中起什么作用呢？假设将另外的127个**double**型插入weights。图5-2显示了对内存的影响。

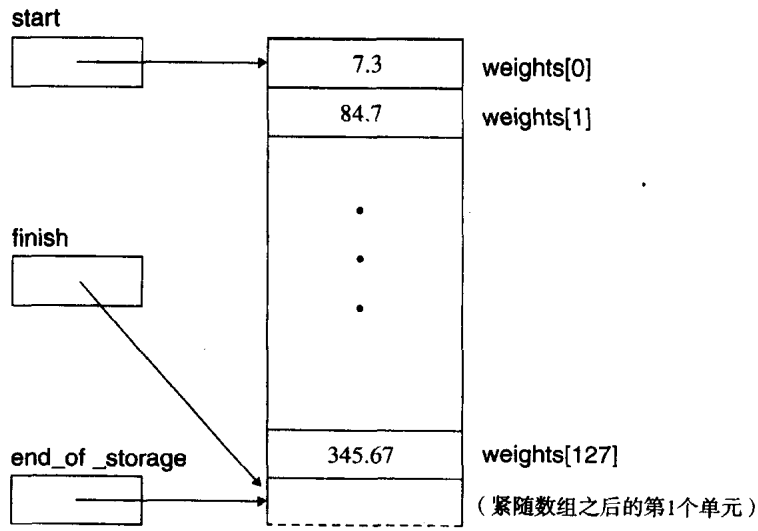


图5-2 有128项的vector<**double**>weights

现在假设发送下面的消息:

178 weights.push_back(15.5);

问题是finish=end_of_storage。项15.5不能存入分配给weights的当前块，也不能把它存到end_of_storage指向的单元，因为那个单元可能包含其他一些程序变量（或其他的重要信息）。

如果一个向量对象对应的数组已满且又尝试新的插入，那么这个数组的容量将加倍。

那就必须增加原先的数组大小适应新的项，这表示分配一个新的堆存储块。那么旧的项将被拷贝到新的数组，旧数组的存储空间被回收并且消除它的项（通常马上进行），然后将新的项插入这个新数组。为了避免频繁地重复拷贝，分配的块是当前块大小的两倍。图5-3显示

了分配这个新块之后内存的相应部分，旧项被拷贝进新的块，插入新的项，并回收旧存储块的空间。

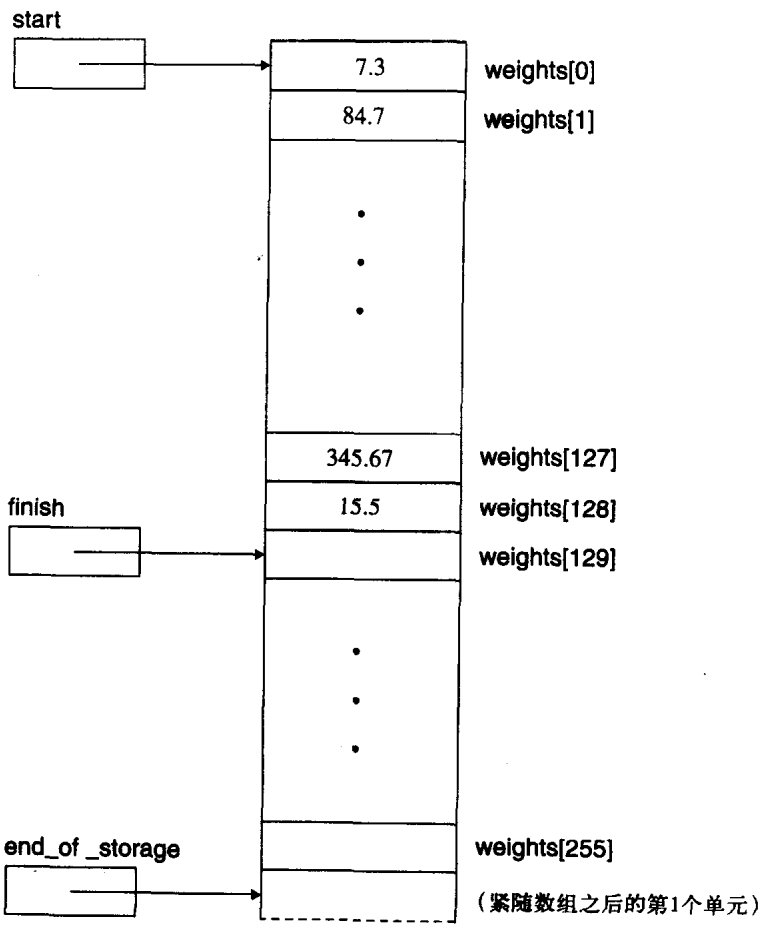


图5-3 调整大小后的vector<double>weights

这个调整大小的策略说明了为什么push_back方法的averageTime(n)是常数。假设向量的当前容量是n项，而且向量已满。如果现在调用push_back方法n次以插入n个项，那么将进行多少次数据移动呢？为了插入第一个新的项，分配的存储块必须能存储$2n$个项，把n个旧项移动到这个新块中，然后添加新项。下面的$n-1$次push_back调用每次只需要移动一下，因此push_back的n次调用中项被移动的总次数是$2n$，也就是每次push_back调用平均需要2次移动。

worstTime(n)和n是成线性关系的，而且这是当向量调整大小时出现的。但是就像在上一段中看到的，下$n-1$次push_back调用只花费常数时间。这个现象（一次方法调用和n次方法调用有相同的worstTime(n)估算）是经常出现的，因此可以将它归纳成一个函数：***amortizedTime(n)***。这个观点是指，如果补偿（即展开）一个长序列的方法调用代价，方法调用的总代价除以方法调用的次数可能很小。这里的“代价”指的是执行的语句的数量。对push_back方法而言，amortizedTime(n)是常数。在应用中，amortizedTime(n)计算出一个比worstTime(n)更现实的估算结果。而且amortizedTime(n)的计算没有做averageTime(n)计算所需要的假设——每次调用都和其他任何调用有着相同的几率。

179

对n次push_back的连续调用，整个调用序列的worstTime(n)和n是成线性关系的，因此amortizedTime(n)是常数。

向除了向量尾部之外的单元进行插入需要将插入点之后的每一项依次移到当前单元的下一单元中。例如，假设从图5-3所示的向量开始，消息是：

180
```
weights.insert(weights.begin()+1,19.94);
```

回忆前面讲过的，begin方法返回start，因此weights.begin()+1将指向下标1的单元上。从下标128至下标1的每一项必须移动到它们的下一个下标单元上。项15.5移动到下标129的单元，然后345.67移动到下标128的单元……再然后84.7移动到下标2的单元，最后19.94移动到下标1的单元，如图5-4所示。注意，插入的结果是必须修改finish，并且如果当前块已经满了，那么插入就需要调整大小。

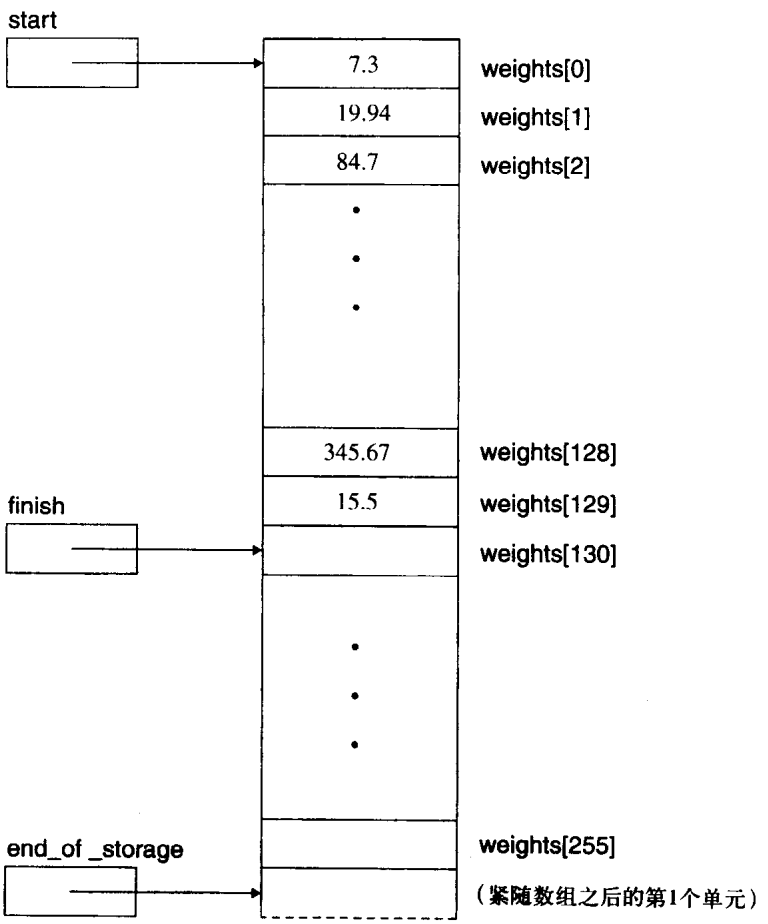


图5-4 向图5-3的vector<double>weights中的下标1单元插入19.94之后的情形

为了说明数据移动的美妙，这里是简单情况下的insert方法的代码，也就是不需要调整大小，并且position!=end()：

```
if (finish != end_of_storage)
```
181
```
{
    construct (finish, *(finish - 1));
```

```
        copy_backward(position, finish - 1, finish);
        *position = x;
        ++finish;
    }
```

调用construct把向量的最后一项（finish-1指向的）拷贝到下一个单元中。那么通用型算法copy_backward把*(finish-2)拷贝到*(finish-1)，把*(finish-3)拷贝到*(finish-2)……把*position拷贝到*(position+1)。最后，将x存入position并增加finish。

这里给出insert的完整定义，并且紧接着对其进行了解释：

```
iterator insert(iterator position, const T& x)
{
    size_type n = position - begin( );
    if (finish != end_of_storage)
        if (position == end( ))
        {
            construct(finish, x);
            finish++;
        }
        else
        {
            construct (finish, *(finish - 1));
            copy_backward(position, finish - 1, finish);
            *position = x;
            ++finish;
        }
    else
    {
        size_type len = size( ) ? 2 * size( ): static_allocator.init_page_size( );
        iterator tmp = static_allocator.allocate(len);
        uninitialized_copy(begin( ), position, tmp);
        construct(tmp + (position - begin( )), x);
        uninitialized_copy(position, end( ), tmp + (position - begin( )) + 1);
        destroy(begin( ), end( ));
        static_allocator.deallocate(begin( ));
        end_of_storage = tmp + len;
        finish = tmp + size( ) + 1;
        start = tmp;
    }
    return begin( ) + n;
}
```

182

这个定义中最令人迷惑不解的行是赋值语句：

```
size_type len=size()?2*size():static_allocator.init_page_size();
```

因为假设缺省分配器size_type代表和**unsigned int**相同的意义。赋值语句右边的是**条件运算符**的一个应用。这是什么？条件运算符提供了常用的**if/else**语句的简写形式。例如，取代下面的语句：

```
if (first > second)
    big = first;
else
    big = second;
```

可以简单地写成

```
big=(first>second)?first:second;
```

条件表达式的语法是:

```
condition? expression_t:expression_f
```

它的意思是: 如果condition为**true**，条件表达式的值就是expression_t的值。否则，条件表达式的值就是expression_f的值。

但是在这个令人迷惑的赋值语句中，简单地调用了size方法作为条件:

```
size()
```

在C++里，**false**和0有相同的含义（都和NULL意义相同）。**true**等价于任意非零整数。称条件size()为**true**意味着size()非零。因此这个令人迷惑的语句的意义是: 如果这个向量的大小不是0，就将len赋值成这个向量大小的两倍；否则，就将len赋值成初始块的大小。

insert方法剩余的部分不难理解，但是读者可能会奇怪为什么进行

```
destroy(begin(),end());
```

的调用，这是为向量的每一项调用析构器。这是为了预防那些项占据比数组单元多的空间。例如，向量中的每一项自身都是一个Linked对象。Linked对象在数组里占据的空间是最小的: head和length字段。但是Linked对象中的节点将不再能访问了。如果没有回收这个空间，那么内存泄漏将导致程序内存溢出。如果项比单个数组单元占据的空间少，那么析构器就什么也不做。

与push_back和insert相比，pop_back方法的实现是轻而易举的。决没有任何大小的调整，
183 因此所有的就是减小指针finish，另外，同样是为了预防的目的，需要调用弹出项的析构器。通用目的的erase方法比pop_back方法更复杂些，这只是因为必须移动项来填补擦除项留下的空间。

实验12包括了vector类的更多的实现细节。

实验12: vector类的更多的实现细节 （所有实验都是可选的）

了解所有这些vector类的底层工作之后，很高兴可以转移到高层次的工作，也就是类的应用上。应用处理了任意的高精度算法，它是公钥加密算法的主要内容。

5.3 向量的一个应用: 高精度算法

现在介绍高精度算法作为vector类的一个应用。马上将讨论细节问题，但是有必要先回想一下: 类的使用是独立（除了效率）于类是如何实现的。因此幸运的是，不需要禁锢在vector类的任何具体实现中。

在公钥加密算法中，使用超过100位长的整数编码和解码。这些**非常长的整数**的基本情

况是:

1) 用相对少的时间——$O(n^3)$——生成一个n位的非常长的整数，它是一个素数[⊖]。例如，假设想生成一个500位的素数，那么需要的循环迭代的次数大约是$500^3 = 125\ 000\ 000$。

公钥加密算法依赖于在只给出乘积时，将乘积分解成两个非常大的素数的乘积的指数级难度。

2) 用非常长的时间——粗略估计约为$10^{n/2}$次循环迭代——求出一个不是素数的n位的非常长的整数的素数因子。例如，假设求一个500位非素数的因子，那么需要的循环迭代次数近似为$10^{500/2}=10^{250}$。

3) 假设生成了两个非常大的素数p和q。可以快速地计算乘积$(p-1)(q-1)$，并可以把这个乘积提供给任何一个想发送消息给你的人。发送者使用这个乘积编码消息——详情请参阅Simmons(1992)。乘积和编码消息是**公开**的，也就是说是通过不安全的信道（像电话、邮政服务或计算机网络）进行传送的。

4) 但是解码消息需要了解p和q的值。因为求p和q因子需要特别长的时间，所以只有你才能解码消息。

非常长的整数比编程语言中直接可用的整数需要多得多的精度。现在将定义、设计和实 184
现very_long_int类的一个简单的版本。习题5.5要求增强这个版本。实验13涉及了这个增强版本的驱动器的开发，编程项目5.1进一步扩展了very_long_int类。

5.3.1 very_long_int类的设计

very_long_int类的每个对象将包含一个大小不确定的非负整数。它只有三个方法：一个很长的整数可以读入，输出，或是和另一个很长的整数相加。下面是方法接口:

```
1. //前置条件: 输入是一系列位，后面跟着一个'X'，忽略无效的位和字符以及空格、
   //            行尾标志。开头处没有0，除非是0自身用单个0表示。
   //后置条件: very_long包含了非常长的整数，它的位来自instream，并返回一个对instream
   //            的引用。worstTime(n)是O(n)，其中n是输入中位字符的数量。
   friend istream& operator>>(istream& instream,very_long_int& very_long);
```

例 假设输入包含:

```
473A53
81X
```

输入语句是:

```
cin>>very_long;
```

那么very_long中将包含4735381。

```
2. //后置条件: 将very_long的数值写入outstream。worstTime(n)是O(n)，
   //            其中n是very_long的大小。
   friend ostream& operator<<(ostream& outstream,
                              const very_long_int very_long);

3. //后置条件: 返回调用对象（左边的操作数）和other_very_long（右边的操作数）
```

⊖ 当p的正整数因子只有1和p自身时，那么大于1的整数p就是素数。

```
//          的总和。worstTime(n)是O(n)，其中n是调用对象和other_very_long中
//          位数的最大值。
very_long_int operator+(const very_long_int& other_very_long);
```

例 假设new_int和old_int是非常长的整数，它们的值分别是1234567890123456789O和15。
185 如果发送的消息是

```
new_int + old_int
```

那么返回的数值将是12345678901234567905。

very_long_int应当包含哪些字段呢？可能需要某种容器，使得每一位是容器的一项。但是需要什么样的容器呢，是数组、vector对象，还是Linked对象。在这个应用中，向量的自动调整大小特性使得它们比数组更适合，而且使用向量，我们早已开发了大量的方法。至于向量和Linked结构之间的选择，Linked类的缺点是它的迭代器只是个前向迭代器。也就是说，只能从前往后地迭代通过一个Linked容器，这会大大降低加法的效率，因此在这个应用中选择vector类。

very_long_int和vector之间的相应关系是什么：是“是一个”（继承）关系还是“有一个”（复合/聚合）关系？也就是说，very_long_int是vector的一个子类，还是令very_long_int包含一个vector类型的字段？very_long_int类的主要目的是执行算法，因此它极少使用vector类的函数。更好的说法是“very_long_int有一个向量”而不是“very_long_int是一个向量”。

所以very_long_int类的惟一的字段是digits，一个vector对象。图5-5给出了very_long_int类的依赖关系图。由于调用very_long_int的隐式析构器时将为向量digits自动调用析构器，所以采用复合。

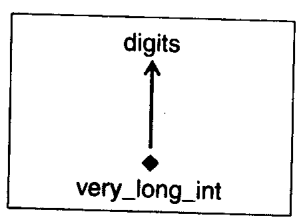


图5-5 very_long_int类的依赖关系图

向量digits的每一项都是一个位。对位应当使用什么整数类型呢？为了节约空间，我们选择了**char**，因为实际上在所有的编译器上，**char**类型的变量只占据一个字节。字段的定义是：

```
vector<char>digits;
```

这些位将按照通常的顺序自前向后地存储在向量digits中。例如，如果一个very_long_int的值是758，那么7将存储在下标0，5存储在下标1，而8存储在下标2。

186 现在已经了解了方法接口和字段，再来关注一下类的实现。

5.3.2 very_long_int类的一个实现

这一节给出了重载运算符的实现：**operator>>**、**operator<<**和**operator+**。谨记vector类的优势（快速随机访问和尾部插入）和缺点（在除尾部之外的单元插入是很慢的）。当然，

除了效率之外，所有的这些都不依赖于vector自身的实现细节：只依赖于方法接口。

1. istream& **operator>>**(istream& instream,very_long_int& very_long)

开始时，先彻底清除very_long，得到一个空容器。然后不断读入字符，直到到达‘X’。对每位的字符，通过调用push_back将相应位的值增加进digits。

下面是方法接口：

```
istream& operator>> (istream& instream, very_long_int& very_long)
{
    const char LOWEST_DIGIT_CHAR = '0';

    const char HIGHEST_DIGIT_CHAR = '9';

    const char SENTINEL = 'X';

    char digit_char;

    very_long.digits.erase (very_long.digits.begin( ), very_long.digits.end( ));
    do
    {
        // 读入至此已加到digits的每一位。
        instream >> digit_char;
        If ((LOWEST_DIGIT_CHAR <= digit_char) &&
                (digit_char <= HIGHEST_DIGIT_CHAR))
            very_long.digits.push_back (digit_char -
                    LOWEST_DIGIT_CHAR);
    } // do
    while (digit_char != SENTINEL);

    return instream;
} // 重载>>
```

这个运算符花费多长时间？对输入的每位字符进行一次循环迭代，每次循环迭代的平均时间依赖于push_back的平均时间，它是常数。因此**operator>>**的averageTime(n)和n成线性关系，其中n是输入的字符位的数量。

worstTime(n)是多少呢？这个时间分析和vector类的push_back方法的分析相关。回忆前面的介绍，n次push_back调用的worstTime(n)和n成线性关系；也就是说，**operator>>**的worstTime(n)和n成线性关系。

187

2. ostream& **operator<<**(ostream& outstream,very_long_int very_long)

从下标0开始遍历向量digits并输出获得的每一位。因为每一位声明的类型都是**char**，所以需要将它们转换成**int**。否则，比如位的数值是7，然后输出将是第7个ASCII字符；这时没有输出任何字符，而是听到一个铃声。

下面是代码：

```
ostream& operator<< (ostream& outstream, const very_long_int very_long)
{
    for (unsigned i = 0; i < very_long.digits.size( ); i++)
        outstream << (int)very_long.digits [i];
```

```
    return outstream;
} //   重载   <<
```

这个运算符的worstTime(*n*)和*n*成线性关系，因为循环迭代的数量是digits.size，而且下标运算符花费常数时间。

可以重新编写**for**语句，用一个迭代器替代下标:

```
vector<char>::iterator itr;
for (itr = very_long.digits.begin( ); itr != very_long.digits.end( ); itr++)
    cout << int (*itr);
```

3. very_long_int very_long_int::**operator**+(**const** very_long_int& other_very_long)

从调用对象和other_very_long对象的最低有效位开始，将它们按位相加。把除以10后的部分和添加进very_long_int对象sum。为very_long_int类自动调用的缺省构造器除了为sum的digits字段调用缺省构造器之外，什么也不做。这个调用使得digits变成一个空向量。

如果部分和大于10，将生成一个进位。出于效率的考虑，将对部分和使用push_back，所以必须在加法之后反转向量digits，这样最高有效位将结束在下标0。例如，假设newInt是一个very_long_int对象，数值是328，oldInt也是一个very_long_int对象，数值是47。如果消息是

```
newInt+oldInt
```

那么在按位相加并压入后，sum中包含数值573。反转后sum中将包含正确的数值375。因此在返回sum之前调用通用型算法reverse。

这里是重载**operator**+的代码:

```
very_long_int very_long_int::operator+ (const very_long_int&
                other_very_long)
{
188     unsigned carry = 0,
            larger_size,
            partial_sum;
        very_long_int sum;
        if (digits.size( ) > other_very_long.digits.size( ))
            larger_size = digits.size( );
        else
            larger_size = other_very_long.digits.size( );
        for (unsigned i = 0; i < larger_size; i++)
        {
            partial_sum = least (i) + other_very_long.least (i) + carry;
            carry = partial_sum / 10;
            sum.digits.push_back (partial_sum % 10);
        } // for
        if (carry == 1)
            sum.digits.push_back (carry);
        reverse (sum.digits.begin( ), sum.digits.end( ));
        return sum;
```

```
} // 重载+
```

least方法是非公有**辅助**方法的一个示例：创建它是为了简化另一方法的实现。在这种情况中，least(i)返回给定位向量中的第*i*个最低有效位。个位（最右边）看作是第0个最低有效位，十位看作是第一个最低有效位，依此类推。例如，假设向量digits的值是3284971，*i*是2。那么返回的位是9，因为9是digits向量中第二个最低有效位；第0个最低有效位是1，而第1个最低有效位是7。方法定义是：

```
//后置条件：如果i>=digits.size()，就返回0；否则返回digits中的第i个
//          最低有效位。最低有效位是第0个最低有效位。
char very_long_int::least (unsigned i) const
{
    if (i >= digits.size( ))
        return 0;
    else
        return digits [digits.size( ) - i - 1] ;
} // least
```

为了简化，假设调用对象和other_very_long对象是大小为*n*的非常长的整数。对least方法，averageTime(*n*)是常数。在一个vector中进行添加平均只耗费常数时间，因此**operator**+定义中**for**语句的averageTime(*n*)和*n*成线性关系。对reverse通用型算法而言，averageTime(*n*)和*n*成 189
线性关系，因此**operator**+的averageTime(*n*)和*n*成线性关系。通过对**operator**<<进行相同的分析，可以说明**operator**+的worstTime(*n*)和*n*成线性关系。

注意，如果以反序存储这些位，那么**operator**+的定义可以稍微简单些，而时间估算不会改变。但是读入一个非常长的整数的时间将和*n*成平方关系，因为每一位将被插入到向量digits的前面。

习题5.5扩展了very_long_int类，实验13涉及到这个扩展的实现。

实验13：扩展very_long_int类　　（所有实验都是可选的）

5.4 双端队列

即将讨论的下一个顺序类是deque类。deque（双端队列）是“double ended queue”的缩写，但它的发音是“deck”。**双端队列**是有如下特征的项的有限序列：

1) 给定序列中任意项的下标，就可以花费常数时间访问或修改这个下标上的项。

2) 平均情况下，在序列头或尾的插入只耗费常数时间，但是worstTime(*n*)是$O(n)$，这里*n*代表序列中项的数量。

3) 在序列尾进行的插入和删除，它们的worstTime(*n*)是常数。

4) 对任意的插入和删除，worstTime(*n*)是$O(n)$，averageTime(*n*)也一样。

在双端队列的首尾插入和删除都是很快的，而向量只有在尾部进行插入和删除才比较快。

从概念上来看，向量和双端队列仅有的区别是：双端队列对象可以在它自身的首尾快速地插入和删除，而向量对象只有在尾部才能快速地插入和删除。

deque类没有（或不需要）vector类中的capacity和reserve方法。除此之外，deque类和它的关联iterator类拥有vector类及其关联iterator类所拥有的所有的方法接口，而且还多了两个双端队列方法：

```
//后置条件: 在这个双端队列的开头插入x的一个拷贝。averageTime(n)是常数，
//          worstTime(n)是O(n)，而对n次连续插入，worstTime(n)也只是O(n)。
//          也就是说，amortizedTime(n)是常数。
void push_front(const T& x);

//后置条件: 这个双端队列开头的项被删除。
void pop_front();
```

190

注意 在pop_front()的后置条件中没有给出时间估算，这暗示着它的worstTime(*n*)是常数。

无疑，对于容器开头进行的插入和删除，双端队列要比向量快很多。下面是它们的类似之处：

1) 在向量和双端队列中，给定下标或迭代器都可以检索或替换任意项，并且这些操作的worstTime(*n*)都是常数。

2) 在向量和双端队列中，从尾部插入一个项的averageTime(*n*)都是常数，worstTime(*n*)都是$O(n)$，但是对*n*次连续的尾部插入，worstTime(*n*)也只是$O(n)$。也就是说，amortizedTime(*n*)都是常数。

3) 在向量和队列中，删除尾部项的worstTime(*n*)都是常数。

根据push_front和pop_front的估算，给人的印象是双端队列有时比向量快，而且从不比向量慢。在接下来的小节中，当看到deque类的典型的字段和实现时，就可以洞察到，除了开头或接近开头的位置，在其他位置进行的插入和删除中，双端队列比向量稍慢一些的原因。但是这里首先给出一个简单的程序解释双端队列：

```
#include <deque>
#include <iostream>
#include <istring>

using namespace std;

int main( )
{
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    deque<string> words;
    deque<string>::iterator itr;

    words.push_back ("yes");
    words.push_back ("no");
    words.push_front ("maybe");
    words.push_front ("wow");

    cout << endl << "the deque after 4 insertions:" << endl;
    for (unsigned i = 0; i < words.size( ); i++)
```

```
        cout << words [i] << endl;                                    191
    words.pop_front( );
    words.pop_back( );

    cout << endl << "the deque after deleting the front and back items"
         << endl;
    for (itr = words.begin( ); itr != words.end( ); itr++)
         cout << (*itr) << endl;

    words.front( ) = "now";
    words.back( ) = "but";

    cout << endl << "the deque after replacing \"maybe\" with \"now\" "
         << "and \"yes\" with \"but\"" << endl << *(words.begin( )) << endl
         << *(words.end( ) - 1);

    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );

    return 0;
} // main
```

注意用了三种不同的方式访问了words的开头项:

```
words[0]
words.front()
*words.begin()
```

同样，也用了三种不同的方式访问了尾部的项。程序的输出是

```
the deque after 4 insertions:
wow
maybe
yes
no

the deque after deleting the front and back items
maybe
yes

the deque after replacing "maybe" with "now" and "yes" with "but"
now
but

please press the Enter key to close this output window.
```

deque类的字段和实现

照例，现在将关注deque类的惠普的设计和实现。编程项目5.2继续开发了一个更简单（但效率不同）的版本。在deque类的惠普的设计中，主要的字段是一个指针数组，指向保存项的连续存储块。所有这些块都是相同大小的。一个块能够保存的项的数量是1KB/项的大小。指 192

针数组，称作**映射数组**，初始时数组从头到尾都是未使用的单元。中间单元将指向当前保存双端队列项的存储块。字段start和finish是迭代器，它们分别指向队列的第一个项和最后一项之后的位置。

在一个简化的范例中，假设每个块保存五个项，并且deque对象pets包含了11个这样的项，依次是："dog"，"cat"，"pig"，"gerbil"，"canary"，"duck"，"cow"，"horse"，"parrot"，"fox"，"rabbit"。图5-6显示了如何表示deque对象pets，使用问号指示未使用单元。

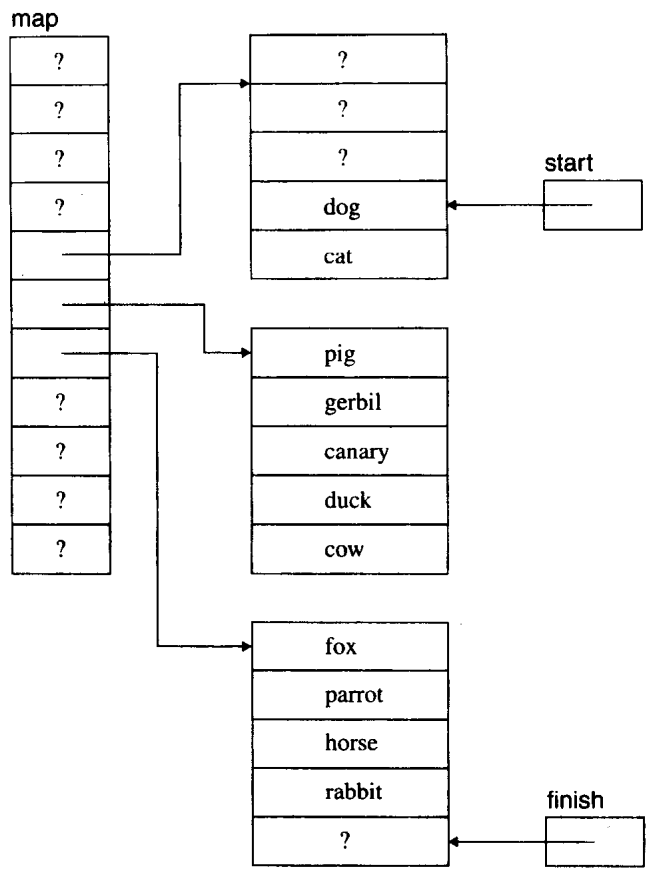


图5-6 包含11个项的双端队列

像下面的消息

```
pets.pop_front();
```

很容易处理：start现在将指向"cat"。但是在双端队列上运行下面的消息将产生什么样的结果呢？

```
pets.push_back("mouse");
pets.push_back("iguana");
```

193

项"mouse"被添加到第三块（块2）的尾部，而迭代器finish将指向第三块尾部之后的位置。当尝试添加"iguana"时，迭代器finish将超出块尾，因此必须分配一个新的块。图5-7显示了执行pop_front和两次push_back之后的deque对象pets的情形。

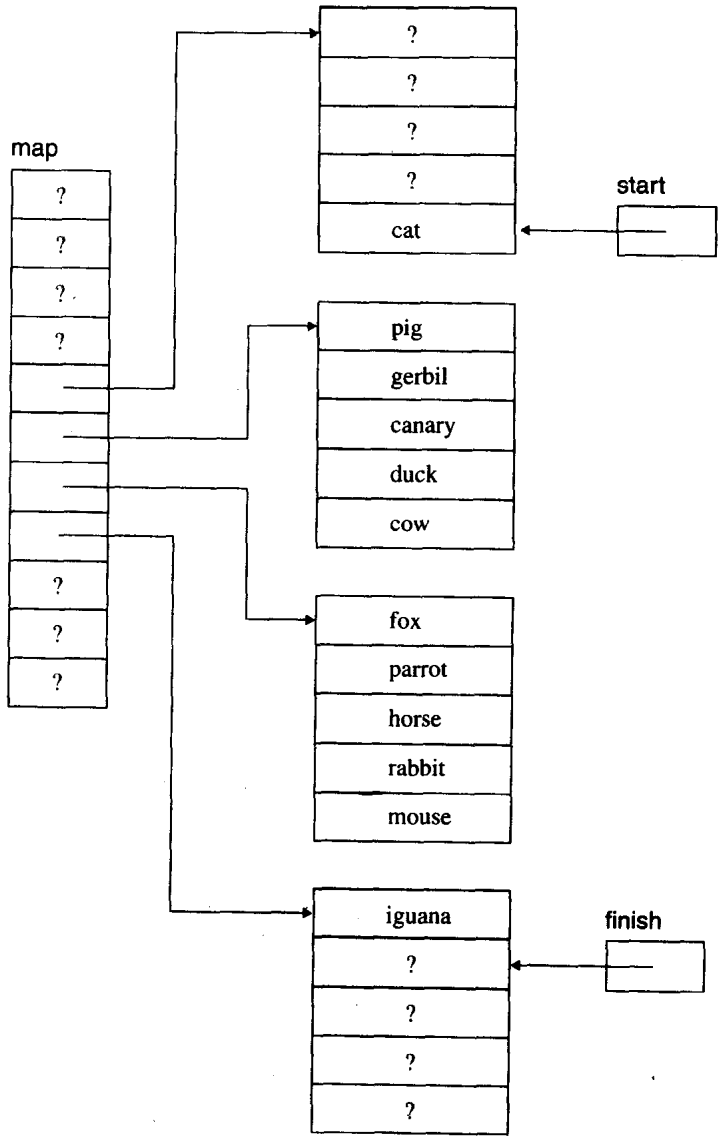


图5-7 对图5-6的deque进行一次pop_front和两次push_back调用后的情况 194

令人奇怪的是，在push_back方法中，如何决定必须分配一个新的块。这些还有iterator字段的其他的神秘之处的答案嵌入在deque类里。每个迭代器有四个字段。当轻松地称“一个迭代器位于项x的位置时”，其实有四个关于项x的字段和它们的含义：

1) first，指向包含项x的块的第一个项。

2) current，指向项x的指针。

3) last，指向包含项x的块的尾部的下一个单元的指针。

4) node，指向map中单元的指针，其中map指向包含项x的块首。

例如，假设双端队列迭代器itr位于图5-7的双端队列中的项“duck”。图5-8显示了迭代器itr、start和finish的字段的值。在这个图中，画有反斜线的框表示紧随块尾之后的单元。请用点时间认真学习图5-8。

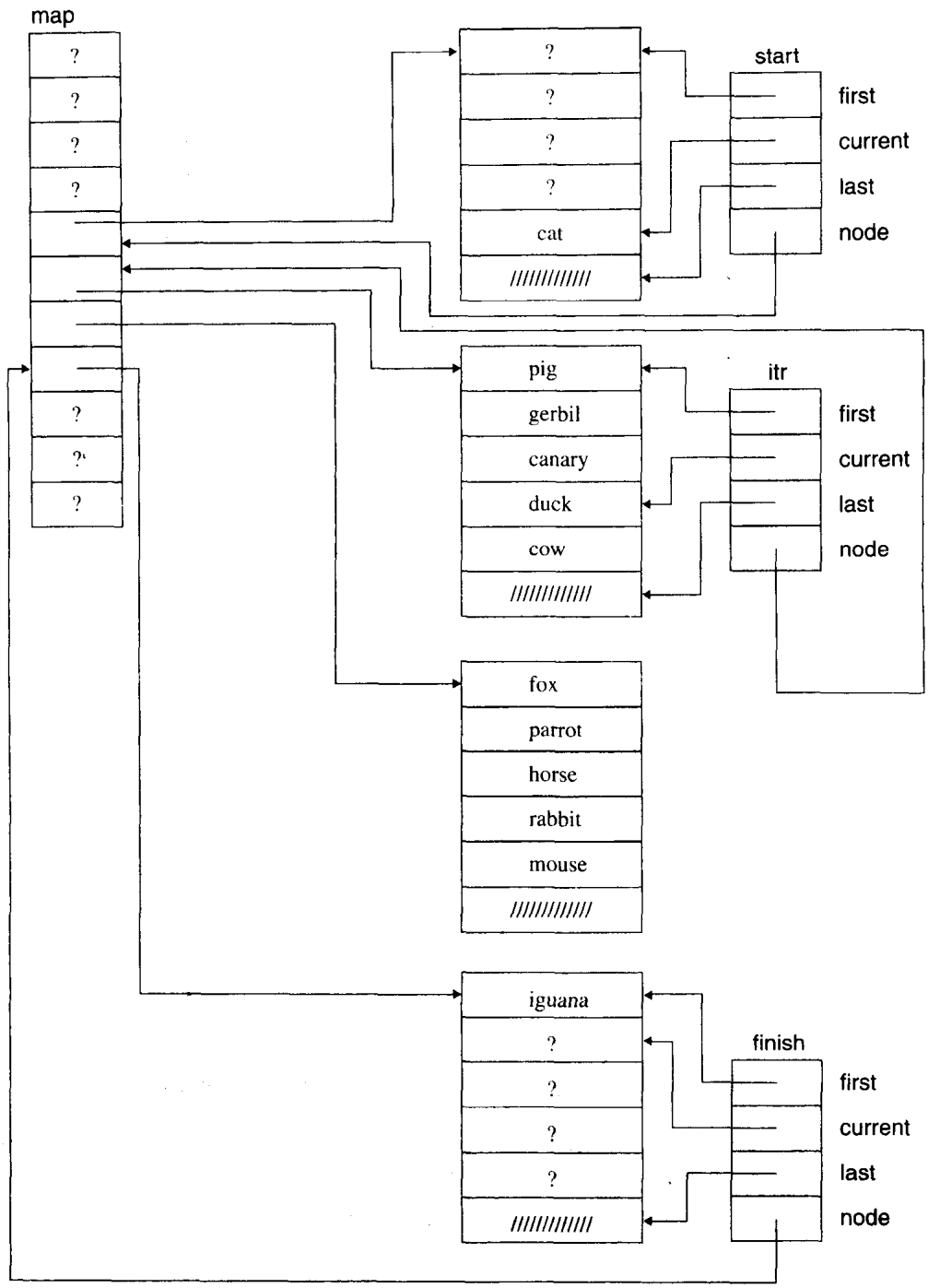


图5-8 图5-7的deque及其迭代器细节。注意finish.current指向紧随双端队列最后一项的下一个单元

这些功能强大的迭代器允许双端队列方法使用简单的算法实现头尾的快速插入和删除，以及在给定下标时检索或替换任何项。例如，假设有：

```
pets[9]="goose";
```

为了返回下标9的引用，必须了解项所在的块号，还有它距该块开头的偏移量。首先需要了解从哪儿开始，即到块0中第一个项有多远？用start.current减去start.first，得到4，这意味着pets中第一个项——项0——在第一个块中的偏移量是4。然后将4和给定下标9相加得到13。因为块的大小是5，13/5给出了块号，即2。同理，块的偏移量是13%5，即3。

因此返回对块2中偏移量为3的单元的引用。这个位置上的项“rabbit”被替换成“goose”。现在“goose”在双端队列的下标9——注意第一个项“cat”在下标0。

如果向deque对象pets发送另一个pop_front消息，第一个块将是未使用的，因此可以回收它，然后相应地调整迭代器start。下面是pop_front()的代码：

```
void pop_front( )
{
    destroy(start.current); // 为start.current指向的项调用析构器
    ++start.current;
    --length;
    if (empty( ) || begin( ).current == begin( ).last)
        deallocate_at_begin( ); // 回收这个块
}
```

195 ~ 196

push_front方法首先判断是否需要在队列的开头分配一个新的块。然后减少start.current，令它指向新块开头的下一个可用单元。下面是代码：

```
void push_front(const T& x)
{
    if (empty( ) || begin( ).current == begin( ).first)
        allocate_at_begin( );
    --start.current;
    construct(start.current, x); // 将x的对象拷贝进start.current
                                 // 指向的单元。
    ++length;
}
```

是否需要调整大小？是的。当所有映射指针指向的块都被使用而又需要另一个块时，map大小将加倍，并且旧的指针将位于新的map数组的中间。因此双端队列中插入的大O时间估算和向量相同：averageTime(n)是常数，worstTime(n)和n成线性关系，并且amortizedTime(n)也是常数。

当调整大小时，只影响map数组：正在使用的块是不会有任何改变的！回想在 vector类中，当前容量的增加将导致所有的向量项被拷贝到一个新的数组中。只有在双端队列中在除头尾以外的单元进行插入或删除时才需要重新安排项。例如，在图5-8所示的deque对象pets中，如果消息是：

```
itr.erase();
```

那么这将导致双端队列中“duck”前面的四个项向下移动（迭代器start必须相应调整）。移动项的代价是很昂贵的，特别是当有很多项或每一项都很大时。但是即便在这里，deque类也提供了一个好办法。正如前面看到的，如果即将删除的项接近双端队列的开头，那么前面的项将向下移动。但是如果即将删除的项接近双端队列的尾部，那么后面的项将向上移动。因此

一个双端队列内部的插入或删除将平均移动1/4的项。

通过前面的介绍说明了双端队列类幕后的复杂性，但是对它的能力还没什么了解。双端队列比向量好到什么程度？现在列出各种情况：

1) 调整大小时没有项的移动。当在双端队列的头尾插入时需要一个额外的块，不需要移动双端队列的任何一项。偶然情况下才会因为map自身太小了，而创建一个双倍大小的map，然后将旧的map拷贝到新map的中间，但这也没有移动项。在向量中，调整大小需要移动所
197 有的项。

2) 对deque类的pop_front方法，worstTime(*n*)是常数。vector类甚至都没有pop_front方法，因为它的worstTime(*n*)和*n*成线性关系，但是一些vector类的用户可能天真地假设它的worstTime(*n*)像pop_back一样是常数。为了从向量vec中删除开头的项，应当执行下面的语句：

```
vec.erase(vec.begin());
```

并且worstTime(*n*)和*n*是成线性关系的。

3) 在双端队列里，push_front平均只耗费常数时间。vector类甚至没有一个push_front方法。在双端队列里，push_front可能需要调整map的大小，因此worstTime(*n*)是$O(n)$，因为map的大小和双端队列中项的数量是成正比的。但是amortizedTime(*n*)仍是常数。

4) 在双端队列里，未使用的块将被回收。如果一个双端队列收缩到外部两个块（位于start或finish的块）之一不再使用，那么将回收该块。回忆一下，向量只会成长，但从不收缩。

5) 对内部的插入和删除，双端队列比向量要少几次数据移动。例如，如果即将移动的项接近双端队列的开头，那么只有删除点之前的项被移动。否则，只有删除点之后的项被移动。平均情况下，只有25%的项将被移动，而向量是50%。

双端队列的重大缺陷是需要模算法将下标转换成块地址。实际上，除非大多数操作都位于或接近容器的开头，否则向量是比双端队列快的。实验16尝试着比较了向量、双端队列和链表（list类将在第6章中介绍）。其间，实验14包含了deque类的惠普实现的更多细节。

实验14：惠普的deque类实现的更多细节　　（所有实验都是可选的）

5.5 双端队列的一个应用：非常长的整数

用deque类处理非常长的整数会怎样？可以主要使用very_long_int类的方法定义。但是现在digits将是一个deque对象，替代了vector对象。出于效率而不是正确性上的考虑，惟一的改变是重载**operator+**。这里将使用push_front替换push_back来求部分和与进位。那么digits将不
198 再是反序的，因此可以避免reverse的调用。这些改变不影响大O时间，但是可以稍微加速实际运行。

总结

本章介绍了两个顺序容器类：vector类和deque类。向量比数组要功能强大得多。比如向量是自动调整大小的。当向量不断成长超出当前容量时，就创建两倍大小的数组，并把向量拷贝到这个数组里。这和寄居蟹成长出壳相仿。向量对比数组的一个更大的优势在于插入和删除，用户不用编写代码为新的条目腾出空间或是填补被删除条目的空间。

除了向量有capacity和reserve方法（见编程项目5.2）而双端队列有push_front和pop_front方法不同以外，双端队列和向量有相同的方法。push_front方法平均只花费常数时间，而pop_front方法总是花费常数时间的。

向量和双端队列的应用是高精度算法，它是公钥加密算法的重要组成。

习题

5.1 a. 假设有如下定义：

```
vector<char>letters;
vector<char>::iterator itr;
```

下面的语句序列将输出什么字？

```
letters.push_back ('f');
letters.push_back ('i');
letters.push_back ('e');
letters.push_back ('r');
letters.push_back ('c');
letters.push_back ('e');

itr = letters.begin( );
cout << *itr;
itr++;
cout << *itr;

cout << letters [3];
itr += 4;
cout << *itr;
```

5.2 假设vector_plus是vector类的一个子类，并假设vector_plus没有新的字段。定义下列的每一个vector_plus方法：

a.
```
//前置条件: 调用对象中有一个项等于item。
//后置条件: 从调用对象中删除等于item的项。worstTime(n)是O(n)。
void erase_item (const T& item);
```

199

提示 从调用通用型算法find开始。例如，假设有

```
item* ptr=find(fruits.begin(),fruits.end(),"bananas");
```

那么ptr将位于fruits中“bananas”出现的位置上，或者，如果“bananas”没有出现在fruits中，就位于紧随fruits最后一项之后的单元上。

b.
```
//后置条件: 调用对象包含全部其原来的项以及随后的vec中的全部的项。如果一个项
//          在原调用对象中出现一次，在vec中出现一次，那么那个项将在合并
//          的对象中出现两次。
void merge(const vector_plus<T>& vec);
```

提示 这可以通过重复调用push_back实现。

c.
```
//后置条件: 返回的是调用对象中惟一项的数量。项是惟一的是指它在容器
//          中只出现一次。worstTime(n)是O(n*n)。
```

```
int unique_count()const;
```

提示 使用通用型算法count。例如，假设有

```
int n=0;
count(fruits.begin(),fruits.end(),"bananas",n);
```

那么n将包含“bananas”在fruits中出现的次数。

5.3 在设计very_long_int类的过程中，决定使用（“有一个”关系）而不用继承（“是一个”关系）vector类。为什么？

5.4 按如下要求修改very_long_int类的设计：digits中的每一项由一个5位整数组成。这对大O时间将有什么影响？运行时间呢？

5.5 扩充very_long_int类，增加方法进行初始化、比较和计算斐波纳契数。方法接口如下：

```
a. //后置条件：这个very_long_int为空。
   very_long_int();
b. //前置条件：n是一个非负整数（不是very_long_int类中的对象）。
   //后置条件：这个very_long_int被初始化成n。
   void initialize(unsigned n);
c. //前置条件：n是一个非负整数（不是very_long_int类中的对象）。
   //后置条件：用n的初始值构造very_long_int。
   very_long_int(unsigned n);
```

200

例 假设消息是

```
very_long_int temp_int(1);
```

那么结果是令very_long_int对象temp_int的值是1。

提示 n%10返回n中最右边的位。

```
d. //后置条件：如果这个very_long_int的值比other_very_long的值小
   //就返回真；否则，返回假。WorstTime(n)是O(n)。
   bool operator<(const very_long_int& other_very_long) const;
```

例 假设定义

```
very_long_int new_int(154),
              old_int(215);
```

如果发送消息

```
new_int<old_int
```

那么将返回**true**。

提示 如果这两个very_long_int的大小不同，那么小一些的very_long_int的值一定小于大一些的very_long_int的值[⊖]。如果大小相同，就从最高有效位开始；逐位比较两个very_long_int，直到（除非）两个数的对应位不同。

```
e. //后置条件：如果这个very_long_int的值大于other_very_long_int的
```

⊖ 回想一下，very_long_int的开头是没有0的。

```
//                值就返回真。否则，返回假。
bool operator>(const very_long_int& other_very_long) const;
```

201

```
f. //后置条件：如果这个very_long_int的值等于other_very_long_int
  //                的值就返回真；否则，返回假。worstTime(n)是O(n)。
bool operator==(const very_long_int& other_very_long) const;

g. //前置条件：n是一个正整数（不是very_long_int类中的对象）。
   //后置条件：返回第n个斐波纳契数。worstTime(n)是O(n*n)。
very_long_int fibonacci (int n)const;
```

例 假设发送下面的消息

```
temp_int.fibonacci(100);
```

返回的very_long_int的值将是354224848179261915075——第100个斐波纳契数。

提示 模仿实验10中斐波纳契函数的迭代设计。i和n都是普通整数，但是previous、current和temp将是very_long_int。

5.6 假设开发very_long_int类的过程中，决定使向量digit包含反序的整数。例如，如果输入包含“386X”，当读入3时，它将被存储在位置0。然后读入8并存入位置0，将3移到位置1。最后，读入6并存入位置0，因此，得到6、8、3分别位于位置0到位置2。重新定义相应的重载运算符>>、<<和+。求>>、<<和+的大O时间。

5.7 vector类既没有push_front方法，也没有pop_front方法。以你的观点看，为什么省略了这两个方法？

202

编程项目5.1：扩展very_long_int类

在very_long_int类中，为多个应用开发一个重载的运算符以及一个阶乘方法。下面是接口：

```
//后置条件：返回值是这个very_long_int和otherVeryLong的乘积。
//worstTime(n)是O(n*n)，其中n是调用之前调用对象的位数和
//other_very_long位数之中较大的一个。
very_long_int operator*(const very_long_int& otherVeryLong);

//前置条件：n>=0。
//后置条件：返回n的阶乘。worstTime(n)是O(nlog(n!))，即n次乘法，
//每个乘积的位数少于log(n!)即n!的位数
very_long_int factorial(int n);
```

用实验13的驱动程序验证这些方法。

203

编程项目5.2：deque类的另一种实现

使用vector类实现deque类。不要使用标准模板库中惠普的实现中的deque类，开发一个vector类的子类来简单地实现。它有助于读者熟悉另外的几个vector方法：

```
1. //后置条件：构造大小为n的向量。每个项的值由T的缺省构造器给出。
   vector(unsigned n);
2. //后置条件：返回无需调整大小就可以存储在向量中的项的数量。
```

```
    unsigned capacity() const;
3. //后置条件：如果在这次调用前，向量的当前容量小于n，那么向量大小就会调整成
   //          一个>=n的容量。
   void reserve(unsigned n);
```

在deque类的这个实现中，除了向量中的字段，还将（至少）有两个字段：

```
unsigned front=START_SIZE/2,
         back=START_SIZE/2-1;
```

front和back字段分别是双端队列头尾的下标。构造器将双端队列初始化成START_SIZE（即100）个项，每个项的值都由T的构造器给出。那么开始时front是50，back是49。有几个方法被覆盖。例如，begin()返回start+front，而end()返回start+back+1。

基本上，在项x上应用push_front方法，得到：

```
front++;
(*this)[0]=x;
```

在项x上应用push_back方法，得到：

```
back++;
(*this)[back-front]=x
```

任何时候双端队列的大小都是back-front+1。这里要说明一个复杂的情况，当调用push_front之前front值为0，或是调用push_back之前back的值是capacity()-1。在任一种情况下，向量的大小都将加倍：

```
reserve(2*capacity());
```

双端队列原先的内容现在放在调整大小后的双端队列的下半部分，因此需要重新居中，将这些项向高下标方向移动。如果设置

```
unsigned n=capacity();
```

那么下标$n/2$位置的项将被移动到下标$3n/4$，下标$n/2-1$位置的项将被移动到下标$3n/4-1$，依次类推。现在可以按照描述进行插入。

第6章 表

本章通过介绍另一个顺序容器类继续学习标准模板库的数据结构，这个类为list类。在表和向量（或者双端队列）之间，性能上存在重大的差别。例如，表缺乏向量的随机访问特征：访问表中某个下标处的项需要从表头或表尾（接近该下标的一端）开始循环。但是一旦定位了进行插入或删除的位置，表就能够以常数时间进行插入和删除。这使得迭代器成为几乎所有表应用的基本要素，而且，list类还缺少下标运算符**operator**[]。

在定义表是什么之后，就列举list类和它的关联iterator类的一些方法接口。这个用户的视角全部是由标准模板库指定的。然后提供惠普的设计和实现的一个大体轮廓，并提出简单些（但效率低些）的设计。表的应用描述了一个简单的行编辑器，利用表的能力在任意位置快速地连续插入和删除。

目标

1) 从用户角度和开发者角度全面地理解list类。
2) 给出一个需要顺序容器类的应用，并能判断表、向量或双端队列中哪一个更适合这个应用。
3) 比较list类的惠普的设计和单表设计以及带有头尾字段的双向表设计。

6.1 表

日常生活中常常为了排序而构造表：杂货铺的杂货清单，登记表，电话目录，班级花名册，电视节目表等等。因此，表中经常表述的问题如：

给出一系列的测验成绩，将它们按升序排序。

输出所有欠缴费用的俱乐部成员的名单。

表——有时称作**链表**，是项的有限序列，它具有下列特征：

1) 访问或修改序列中的任意项需要花费线性时间。

2) 给出序列中某一位置的迭代器，在这个位置上插入或删除一个项需要花费常数时间。

从6.1.1节开始，我们将设计并实现符合这个数据结构的list类。那么表的这两个属性与向量对象的行为相比如何呢？回忆一下，访问或修改向量vec中位置k上的项，可以采用下标运算符：

```
vec[k]
```

下标运算符也可以用于双端队列。在表中，必须使用迭代器。假设lis是list类的一个实例，而我们想访问从lis头开始算起的第k个位置上的项。可以从lis开头顺序前进直到位置k，或是从lis的末尾后退到位置k，从这两者中选择一条较短的路径：

```
if (k < lis.size( ) / 2)
{
```

```
        // 从lis开头循环前进:
        itr = lis.begin( );
        for (int i = 0; i < k; i++)
            itr++;
    } // if
    else
    {
        // 从lis末尾向后循环:
        itr = lis.end( );
        for (int i = lis.size( ); i > k; i--)
            itr--;
    } // else
```

这个访问的时间和k成正比。令n表示链表中项的数量。在最坏情况下，当k=n时，循环迭代的数量是$n/2$，和n成线性关系。从k到链表开头或末尾的平均距离是$n/4$，也和n成线性关系。

不能像向量和双端队列那样以常数时间进行访问的原因是链表迭代器不是随机访问迭代器，而只是双向迭代器。这意味着从链表的一个给定位置上，迭代器可以前进或后退一个位置。对比前面章节中，随机访问迭代器可以直接前进或后退任意个位置。

206

但是一旦准确地定位了一个迭代器，在链表中这个位置的插入或删除就只需要常数时间，而在向量或双端队列中则需要线性时间。这说明了使用链表取代向量（或双端队列）的主要动机：当应用需要在除了容器尾部（对双端队列而言是头尾两端）的位置进行多次插入或删除时，适合使用链表。

多个链表的合并只需要常数时间。举一个例子解释“合并”的意思，假设list1包含项

“television”，“radio”，“stereo”，“CD player”

而迭代器itr位于“radio”。如果list2包含项

“camcorder”，“VCR”，“laser disk player”

可以发送下面的消息：

```
list1.splice(itr,list2);
```

结果是list2的项从list2中移走并插入到list1中项“radio”的前面。因此list1将包含

“television”，“camcorder”，“VCR”，“laser disk player”，
“radio”，“stereo”，“CD player”

而list2为空。至此，读者大概能领会到向量或双端队列的合并需要和提供合并项的容器的大小成正比的线性时间的原因。

6.1.1 list类的方法接口

list类和它的关联iterator类的方法接口和前面在向量及双端队列中看到的很相似。先从list类中应用最广泛的方法的接口开始。表6-1简略地概括了这些方法。list类是模板类，有表示链表项类型的模板参数T。

表6-1 一些list方法的简要描述（假设定义为：list<double>::iterator itr;）

方 法	功 能
list<**double**> x	x成为一个空链表
list<**double**> weights(x)	list对象weights包含了对list对象x的拷贝
weights.push_front(8.3)	在weights的开头插入8.3
weights.push_back(107.2)	在weights的尾部插入107.2
weights.insert(itr,125.0)	在itr所在的位置插入125.0；将插入点到weights尾部之间的项向上移动；返回指向刚插入项的迭代器
weights.pop_front()	删除weights开头的项
weights.pop_back()	删除weights尾部的项
weights.erase(itr)	删除itr所在位置的项；只有那些被删除项位置上的迭代器和引用失效
weights.erase(itr1,itr2)	删除weights中itr1所在位置（包括itr1）到itr2所在位置（不包括itr2）之间的项
weights.size()	返回weights中项的数量
weights.empty()	如果weights中没有项就返回true；否则，返回false
itr=weights.begin()	令itr位于weights开头的项
itr==weights.end()	如果itr恰好位于紧随weights最后一项之后的位置就返回true；否则，返回false
new_weights=weights	先前定义的list对象new_weights中包含weights的一个拷贝
weights.splice(itr,old_weights)	把old_weights中的所有项放在weights里itr所在位置的前面。不论weights或old_weights里原先有多少项，这个方法的时间总是常数
weights.sort()	根据operator<排序weights中的项

1. //后置条件：这个链表为空。
   list();

注意 通常都是隐式地调用这个缺省构造器，例如，

list<Employee> employees;

令employees成为一个空链表，它的项是Employee类型的。

2. //后置条件：构造链表并将其初始化为x的拷贝。
   // worstTime(n)是O(n)，其中n是x的大小。
   list(**const** list<T>& x);

例 假设前面定义了一个字符串链表old_words。如果写成

list<string> new_words(old_words);

那么new_words被构造并包含了old_words的拷贝。

207 ~ 208

注意 回忆第5章，这种类型的构造器称作**拷贝构造器**。

3. //后置条件：将x插入到这个链表的开头。
   void push_front(**const** T& x);

注意 vector类没有push_front方法。这样方法可能给人的印象是在vector对象开头的插入很快。

4. //后置条件：在这个链表尾部插入x。
   void push_back(**const** T& x);

5. //后置条件：将x插入到调用前position所在位置的项的前面。返回位于x位置
   //　　上的迭代器。
   iterator insert(iterator position, **const** T& x);

注意 worstTime(n)是常数。对vector类的insert方法，worstTime(n)是$O(n)$。

6. //前置条件：这个链表非空。
   //后置条件：将调用前这个链表开头的项从链表中删除。
   void pop_front();

7. //前置条件：这个链表非空。
   //后置条件：将调用前这个链表尾部的项从链表中删除。
   void pop_back();

8. //前置条件：position位于链表中某项上。
   //后置条件：将调用前position位置上的项从链表中删除。
   void erase(iterator position);

注意 worstTime(n)是常数。vector类中erase方法的worstTime(n)是$O(n)$。

209

9. //前置条件：first位于链表的某一项上，而last位于该项之后的某项上。
   //后置条件：将调用前所有位于first（包括first）和last（不包括last）
   //　　之间的项从链表中删除。
   void erase(iterator first,iterator last);

注意 这个方法的时间和删除的项的数量成正比。回忆在对应的vector方法中，这个时间是和删除的最后一项**之后**的项的数量成正比的（因为需要移动后面的那些项来填满清空的位置）。

10. //后置条件：返回这个链表中项的数量。
    unsigned size() **const**;

11. //后置条件：如果链表为空就返回真。否则，返回假。
    bool empty() **const**;

12. //后置条件：返回位于这个链表开头的迭代器。
    iterator begin();

13. //后置条件：返回位于这个链表末尾后的迭代器。
    iterator end();

注意 如果调用对象链表为空，那么begin方法返回的iterator就等于end方法返回的迭代器。

14. //后置条件：这个链表包含了x的拷贝，并返回对这个链表的引用。
    list<T>& **operator=**(**const** list<T>& x);

注意 这个赋值运算符与拷贝构造器（方法2）的不同之处在于，拷贝构造器的调用对象在被初始化为参数x的同时还进行了定义。

15. //后置条件：从position位置开始将x的内容插入这个链表，然后x为空。
    void splice(iterator position,list<T>& x);

注意 不论x多大，这个方法总是花费常数时间。

16. //前置条件：为类型T定义了运算符>。
    //后置条件：这个链表中的项按照升序排列。worstTime(n)是O(nlogn)。
    void sort();

注意 将在第12章中学习这个方法。 210

链表中也有front和back方法，它们的方法接口与向量中的相同。

6.1.2 迭代器接口

list类支持双向迭代器，而不是随机访问迭代器。没有运算符+就是一个很好的说明。下面是接口：

1. //后置条件：这个迭代器位于链表的下一个位置，并返回对这个迭代器的引用。
   iterator& **operator**++();

注意 这是一个**前加**运算符；也就是说，迭代器前进并返回对新位置迭代器的引用。例如，假设cities是一个list对象，它包含下面的城市：

"Boston"，"College Station"，"Lansing"，"Pasadena"

如果itr是位于"College Station"上的链表迭代器，并编写了

list<string>::iterator new_itr=++itr;

那么itr和new_itr就都位于"Lansing"上了。

2. //后置条件：这个迭代器位于链表的下一个位置，并返回对迭代器前一个值
   //　　的拷贝。
   iterator **operator**++(**int**)

注意 这是一个**后加**运算符；也就是说，迭代器前进但返回前进之前迭代器的值。后加运算符有一个**int**类型的参数，使用它的惟一目的是为了与前加运算符相区别。实际上并没有变元对应这个**int**参数。例如，假设cities是一个list对象，它包含下面的城市：

"Boston"，"College Station"，"Lansing"，"Pasadena"

如果itr是位于"College Station"的链表迭代器并编写了

list<string>::iterator old_itr=itr++;

那么itr就位于"Lansing"，但是old_itr仍位于"College Station"。

3. //后置条件：这个迭代器位于链表的前一个位置，并返回对这个迭代器的引用。
   iterator& **operator**--();//前减

4. //后置条件：这个迭代器位于链表的前一个位置，并返回对这个迭代器前一个值 211
   //　　的拷贝。
   iterator **operator**--(int);//后减

5. //前置条件：这个迭代器位于链表的某一项上。

```
//后置条件: 返回对这个迭代器位置上的项的引用。
T& operator*();
```

例 假设itr位于项“Lansing”上。如果编写了

```
cout<<(*itr);
```

输出将是

```
Lansing
```

注意 因为返回了一个引用，所以可以使用这个运算符来改变链表中一个项的值。例如，

```
*itr="Detroit";
```

将把itr位置上的项的值改成“Detroit”。

```
6. //后置条件: 如果这个迭代器和x位于链表相同的地方，就返回真。否则，返回假。
   bool operator==(const iterator& x);
```

注意 还有**operator**！=。

这里用一个小程序来解释几个链表方法和链表迭代器运算符。

```
#include <list>
#include <iostream>
#include <string>

using namespace std;

int main( )
{
      list<string> words;
      list<string>::iterator itr;

      words.push_back ("yes");
      words.push_back ("no");
      words.push_front ("maybe");
212   words.push_front ("wow");
      cout << "size = " << words.size( ) << endl;

      cout << endl << "the list after 4 insertions:" << endl;
      for (itr = words.begin( ); itr != words.end( ); itr++)
            cout << (*itr) << endl;

      words.pop_front( );
      words.pop_back( );

      cout << endl << "the list after 2 deletions:" << endl;
      for (itr = words.begin( ); itr != words.end( ); itr++)
            cout << (*itr) << endl;
      cin.get( );

      return 0;
} // main
```

这个程序的输出是:

```
size = 4

the list after 4 insertions:
wow
maybe
yes
no

the list after 2 deletions:
maybe
yes
```

在开始学习list类可能的实现之前，应当先从用户的观点，即标准模板库的数据结构出发，比较链表和向量（或双端队列）。6.1.3节探讨了链表和向量作为数据结构的区别。

6.1.3 链表方法和向量或双端队列方法的差别

如果应用需要访问或修改顺序容器中位置迥异的项，就选择向量（或双端队列）。

list方法和vector或deque方法的最显著的差别是list类中没有下标运算符——**operator**[]。这意味着链表缺乏随机访问属性。在6.1节中曾看到，通过从链表头部或尾部开始循环，可以模拟下标运算符的作用，但是这样花费的时间和从开头（或末尾）项到给定下标项之间的项的数量成线性关系。

因此当应用需要访问或修改顺序容器中位置迥异的项时，使用向量或双端队列将比使用链表快很多。

213

如果应用需要迭代通过一个顺序容器并在迭代中进行插入或删除，就选择链表。

另一方面，在链表中插入或删除一个迭代器所在位置上的项只需要常数时间，而在向量或双端队列中却需要线性时间。

如果应用的大部分工作都是迭代通过一个顺序容器以及在迭代中进行插入或删除，那么使用链表要比使用向量或双端队列要快很多。

list类和vector或deque类之间的另一个差别是插入和删除将导致迭代器失效的程度。通常，链表中的插入和删除只会使直接相关的迭代器失效。例如，假设lis是一个链表对象，itr1和itr2是迭代器。如果itr1位于item1而itr2位于稍微后面点儿的项，那么消息

```
lis.erase(itr1);
```

将令itr1失效。也就是说，正如人们所期望的，itr1将不再依赖于指向item1的指针。但是itr2将仍指向发送erase消息前它所指向的相同的项。

现在假设对向量vec有相同的情况，发送消息

```
vec.erase(itr1);
```

那么itr2将不再指向发送erase消息前它所指向的项。原因是向量中的删除会导致删除点之后的项的重新布置。从双端队列中删除也会出现相同的问题。并且itr1也会失效，因为erase方法为itr1所在的项调用了析构器。

插入之后的迭代器状态是相似的。对链表来说，没有项的移动，因此迭代器仍然位于插

入前它们所在的位置。对向量来说，插入必定要移动项，因此迭代器可能失效。特别是当新的大小超过原有容量时，会发生扩展，这时所有的迭代器和引用都将失效。否则，只有在插入点之后的迭代器和引用才会失效。双端队列的情形和向量很相似，不同的是双端队列中失效的范围是从插入点到队列头尾中接近插入点的一端。

6.1.4 list类的字段和实现

在这一节，概括描述了标准模板库的list类的一个实现（即惠普的版本）。由于C++对效率的关注，使得所有广泛应用的实现，包括惠普的实现，都多少有些复杂。6.1.6节中将考察一些简单但效率较低的设计。

和惠普的实现中所有其他的模板类一样，list类的声明和方法定义在相同的文件中。最基
214 本的字段是length和node，定义如下：

```
template<class T>
class list {
    protected:
        unsigned length;

        struct list_node
        {
            list_node* next;
            list_node* prev;
            T data; // 保存一个项
        }; // list_node

        list_node* node;
```

图6-1显示了串在一起的链表节点，就像项链上的珠子一样。

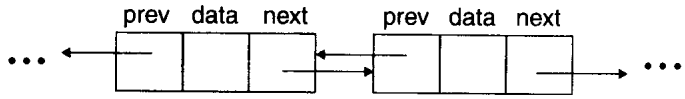


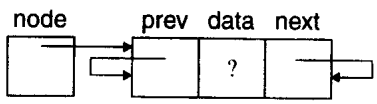
图6-1 在一个list中，节点中的每一项还包括指向前一个或后一个节点的指针

这里有些不可思议的是，指向一个节点的指针包含标识符node，而不是node_ptr，但是这是所有实现所共有的特点（可能是因为其他的实现都是基于惠普的实现）。

node指向的list_node称作**头节点**。在头节点中，data字段是未使用的，而且最初，prev和next字段[⊖]都是指回头节点自身的。也就是说，缺省构造器包含下面的代码：

```
(*node).next=node;
(*node).prev=node;
```

因此在调用缺省构造器之后，有：



⊖ 在惠普的实现中，给定prev和next的类型为void*，因为list_node*只有对缺省构造器才是正确的。而因为采用了缺省构造器，所以可以代入list_node*。

在一个非空链表中，头节点的next字段指向链表的第一项，prev字段指向链表的最后一项。因此链表的存储就像一个环，双向链表。例如，图6-2是有两个string项的链表容器。 215

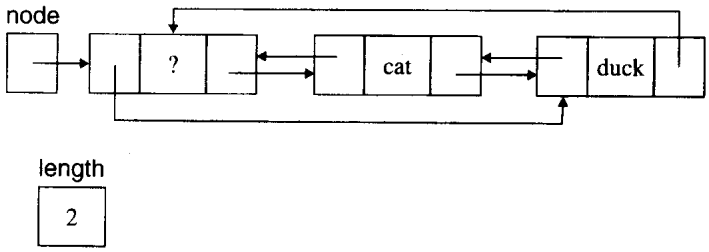


图6-2 有两个项“cat”和“duck”的链表。每个项存储在一个也有prev和next字段的结构中

使用链来连接链表节点是第2章中Linked类所包含的概念，但是这里的每个节点除了next指针还有prev指针，并且还包含一个头节点。

在学习一些list方法的定义之前，需要说一下嵌入的iterator类。这个类有两个受保护成员：一个字段和一个构造器：

```
protected:
    list_node* node;
    iterator(list_node* x):node(x){}
```

这个构造器头有一个**构造器初始化部分**：一个冒号后面跟着由逗号分隔的许多字段的初始化。每个**字段的初始化**由字段标识符和圆括号中的初始值组成。这样做的结果是node字段被初始化成x。实际上，不用初始化部分也可以实现这个结果，可以在构造器定义里将x赋值给node[⊖]。注意这是iterator类的node字段，而不是list类的node字段。

构造器是**protected**的原因是，普通用户对list_node将一无所知，因此没有理由调用这个构造器。这里有一个缺省构造器，它是**public**的。iterator类中有**public**方法的定义也并不奇怪。例如， 216

```
public:
    iterator( ) { }

    T& operator*( ) const { return(*node).data;}

    iterator& operator++ ( )
    {
        node = (*node).next; return *this;
    }
```

⊖ 如果在新类的构造器执行之前必须为对象字段调用构造器，那么构造器初始化部分就是必须的。例如，可能有：

```
class D{
public:
        D(int i): v(i){cout<<v<<endl;}
protected:
        very_long_int v;
```

初始化任何有类作用域的非静态变量也需要构造器初始化部分。

```
iterator operator++ (int)
{
    iterator tmp = *this; ++*this; return tmp;
}
```

实验15有更多关于iterator类的细节。

现在可以考虑list的方法。将定义8个方法：begin、end、insert、push_front、push_back、erase（有一个参数）、pop_front和pop_back。正如在图6-2中所看到的，最后的list_node的next字段指向头节点。也就是说，头节点是链表中最后的list_node的下一个节点。由此可知，list类中的end方法返回位于头节点的迭代器。下面是定义：

```
iterator end(){return node;}
```

后面的事情就有些奇怪了。返回值是node，即指向头节点的指针。但是end方法的返回类型是iterator！迭代器有一个指针字段（也称作node），但是迭代器是一个对象而不是一个指针。在C++中，如果遇到一个类型不匹配的表达式，那么可能的情况下，编译器将执行一个匹配类型的**自动类型转换**。在这个情况下，类型list_node*需要转换成iterator类型。并且由于有**protected**构造器，类型强制转换可自动执行，在前面的iterator类中可以看出这一点：

```
iterator(list_node* x):node(x){}
```

因此end方法真正返回的不是list类的node字段的拷贝，而是从list类node字段构造的一个迭代器。

正如图6-2所示，begin方法应当返回一个迭代器，它位于头节点的next字段指向的list_node，即包含链表中第一个项的list_node。begin方法的定义中也使用了自动类型转换：

```
iterator begin(){return(*node).next;}
```

现在来处理insert方法：

217

```
iterator insert(iterator position, const T& x);
```

这个方法在list_node中存储了项x，并调整了一些next和prev指针字段，使得这个list_node位于迭代器position所在的list_node的前面，然后返回位于新插入节点上的迭代器。

图6-4显示了在图6-3的链表pets上运行下述消息的作用：

```
pets.insert(itr,"dog");
```

通过图6-4得出的重要认识是：

> 在一个list容器中执行插入时，没有项被重新安排。

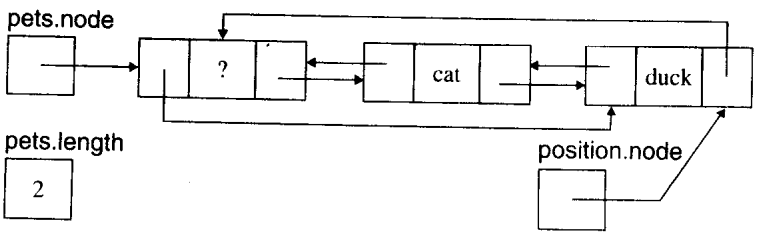


图6-3 图6-2的链表中迭代器position所在的list_node包含“duck”

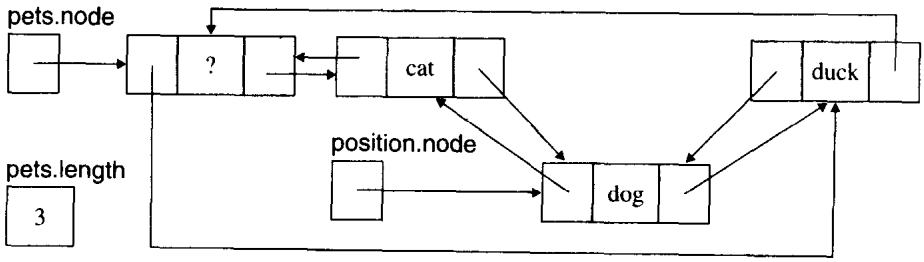


图6-4 插入“dog”之后图6-3的链表情形

通过图6-4，可以推断实现insert方法的步骤如下:

1) 为tmp所指向的list_node分配空间。

2) 在tmp的数据字段中存储项x。

3) 为tmp的next字段（更专业地说，是tmp指向的link_node的next字段）赋值position.node。

4) 为tmp的prev字段赋值position.node指向的link_node的prev字段的值。 218

5) 将position.node指向的list_node的prev字段指向的link_node的next字段赋值为tmp。

6) 将position.node指向的link_node的prev字段赋值为tmp。这个赋值必须在第4步和第5步之后进行。

7) 增加长度。

8) 返回tmp。

根据这个观点，可以提供insert方法的大部分定义。例如，第4步可以写成:

```
(*tmp).prev=(*position.node).prev;
```

但是定义也可能用下面语句实现步骤1:

```
list_node *tmp=new list_node;
```

这是有效的，但效率低。首先，每个list_node有相同的大小，但是通用堆管理器不能利用这种一致性。依赖于计算机系统，每次调用**new**运算符将产生一个中断，而且这些重复的中断将大大降低项目的运行速度。

惠普的实现采用的方法是令list类开发它自己的内存管理例程: get_node分配链表节点，put_node回收空间。因而insert的定义为:

```
iterator insert (iterator position, const T& x)
{
    list_node* tmp = get_node( );
    construct(value_allocator.address((*tmp).data), x);
    (*tmp).next = position.node;
    (*tmp).prev = (*position.node).prev;
    (* ((*position.node).prev)).next = tmp;
    (*position.node).prev = tmp;
    ++length;
    return tmp;
}
```

实验15中讲述了get_node方法的细节，6.1.5节探索了链表存储的基本要素。为了对list类的实现有一个完整的评价，应当细读编译器的list类的实现中的实际代码。它和这里显示的实现可能很相似。

push_front和push_back方法的定义是单行的:

```
void push_front(const T& x) { insert(begin( ), x); }
219 void push_back(const T& x) { insert(end( ), x); }
```

因为有头节点，所以每个链表节点有一个前向和后向节点，并且它简化了插入和删除。

这两个定义这么简单说明了使用头节点的美妙之处: insert方法可以处理前向插入，也可以处理后向插入! insert方法总是将新的项（在一个链表节点中）插入到两个链表节点之间。对于push_front方法，项被插入到头节点和首节点之间。对于push_back方法，项被插入到尾节点和头节点之间。

还需要定义三个方法: erase、pop_front和pop_back。但是一旦定义了erase方法，pop_front和pop_back方法的定义很容易就可以得到（这仍是受益于头节点）。使用insert方法可以将一个新的项（在link_node中）添加进链表。erase方法从链表中去除一个项。图6-5显示了有三个项的链表，并且有一个迭代器位于其中一项上。

为了从pets中删除项“dog”，基本需要两个步骤:

1) 将cat的list_node的next字段改为指向duck的list_node。

2) 将duck的list_node的prev字段改为指向cat的list_node。

执行这两个步骤的情形如图6-6所示。

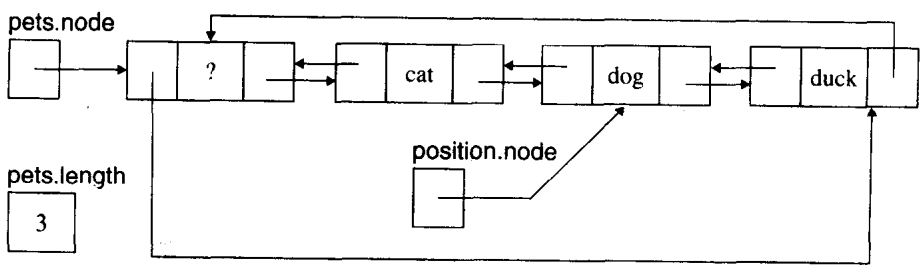


图6-5 将图6-4中有三个项的链表整理后的情形

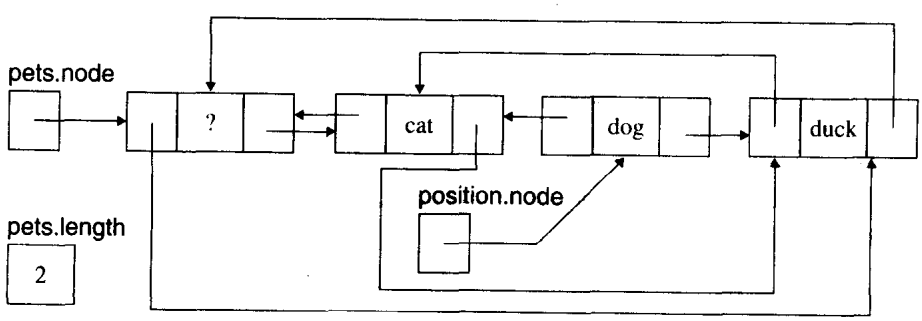


图6-6 在图6-5的链表中去除“dog”之后

还需要考虑一些内务处理: 必须为string对象“dog”调用析构器，回收包含“dog”的
220 link_node的空间，并削减length字段。下面是定义:

```
{
    (*((*position.node).prev)).next = (*position.node).next;
    (*((*position.node).next)).prev = (*position.node).prev;
    destroy(value_allocator.address((*position.node).data));
    put_node(position.node);
    --length;
}
```

6.1.5节将解释put_node方法是如何回收被删除节点的空间的。

pop_front方法删除begin()位置上的项，pop_back方法删除end()前一个位置的项:

```
void pop_front( ) { erase(begin( )); }
void pop_back( )
{
    iterator tmp = end( );
    erase(--tmp);
}
```

在list类的实现中，我们最后所要学习的是链表节点的分配和回收。

6.1.5 list节点的存储

6.1.4节介绍了insert方法调用get_node方法返回指向保存插入项的节点的指针的情况。实验15讨论了get_node方法的定义，但是为了使这个讨论有意义，需要学习链表节点是如何存储的。

在链表中进行第一次插入时，分配一大块内存——通常是1KB。这个块，称作**缓冲区**，用来进行连续的插入，直到填满缓冲区（这里将忽略缓冲区满的情况以及删除如何适合这种表示）。为了判断何时填满缓冲区，list类包含了一个next_avail字段——指向下一次插入将使用的节点，以及一个last字段——指向缓冲区末尾的下一个节点:

```
list_node* next_avail;
list_node* last;
```

图6-7说明了包含四个项——“cat”、“dog”、“duck”、“lion”——的一个list对象，pets。

无论何时分配一个新节点，也不管这个分配是针对insert、push_front、push_back中的哪一个执行的，都要使用next_avail字段。例如，图6-7中push_back的调用将调整next_avail的链表节点的prev和next字段，使得该链表节点的prev字段将指向“lion”链表节点，而它的next字段将指向头节点。然后将增加next_avail自身。例如，图6-8显示了在图6-7所示配置中下面 221
语句的作用:

```
pets.push_back("monkey");
```

这里还需要再考虑两个细节：当缓冲区耗尽时怎么办？删除节点后会怎么样？当缓冲区已满，也就是当next_avail=last时，就分配一个相同大小的新缓冲区。为了跟踪全部的链表缓冲区（以便在废弃链表时可以回收缓冲区），在list类中包含了一个buffer_list字段。这个字段包含了一个指向单链表的指针，它的节点类型是list_node_buffer。每个list_node_buffer有两个字段：一个指向缓冲区的指针，以及一个指向下一个list_node_buffer的指针。

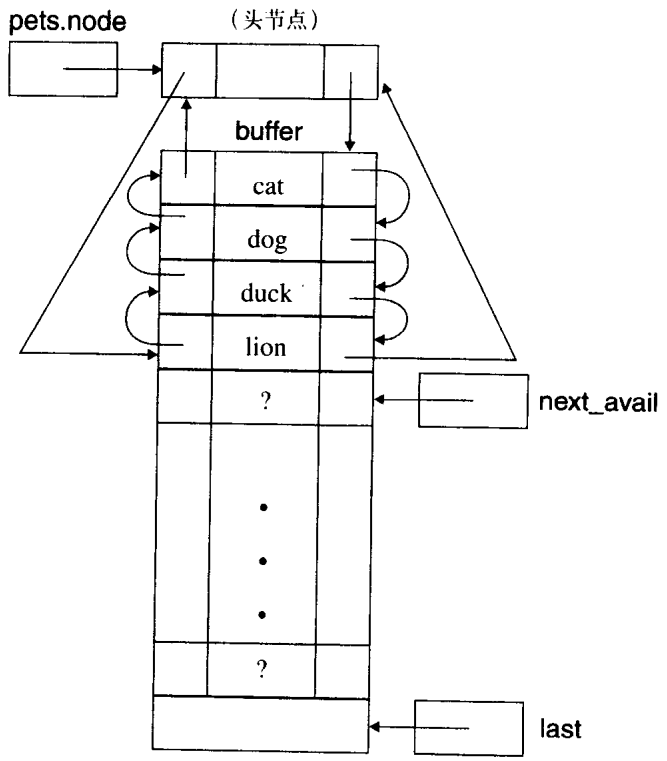


图6-7 buffer中存储的四个宠物的链表

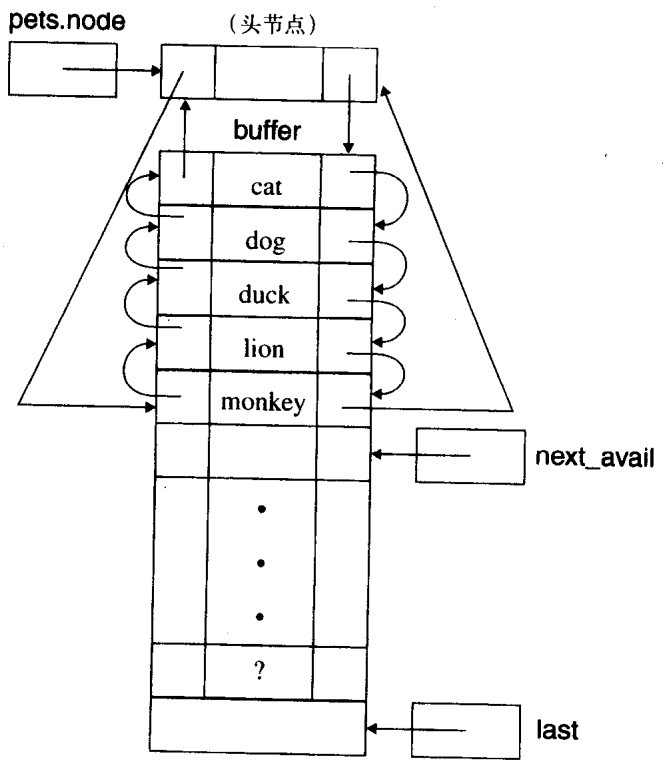


图6-8 在图6-7中调用pets.push_back（“monkey”）之后链表的情况

```
struct list_node_buffer
{
    list_node_buffer* next_buffer;
    list_node* buffer;
};
list_node_buffer* buffer_list;
```

例如，图6-9显示了一个分配了三个缓冲区的链表。

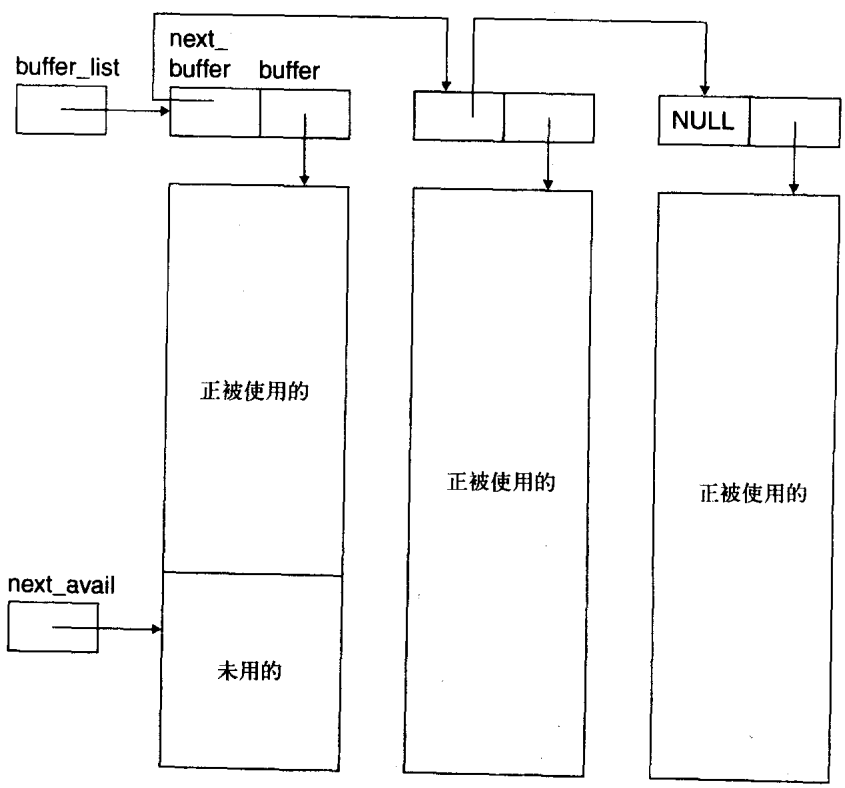


图6-9 为一个链表分配的缓冲区。分配了三个缓冲区，并且最近分配的缓冲区中仍有尚未存储链表节点的部分。数组变量buffer指向数组中的第一个链表节点

最后需要讲一下删除。如果只是把被删除的节点遗弃并任其失效将是很浪费的。相反，采用一个节点链表保存曾经在链表中但后来被删除的节点。这些可反复利用的节点被组织在一个单链表中。这个list类的free_list字段保存了指向最近被删除节点的指针：

```
list_node* free_list;
```

忽略这个节点的prev字段，它的next字段指向下一个最近被删除的节点，那个节点的next字段指向第三个最近被删除的节点，依次类推，空闲链表曲折穿过所有分配的缓冲区。在put_node方法中将一个被删除的节点添加进空闲链表：

```
void put_node(link_type p)
{
    p->next = free_list;
```

```
        free_list = p;
    }
```

例如，从图6-8所示的五个宠物的链表中删除“duck”，删除之后对链表的影响如图6-10所示。

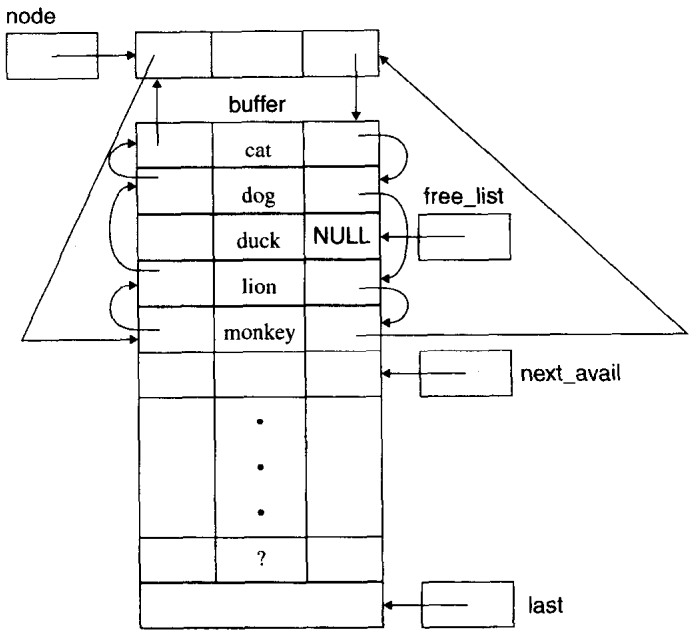


图6-10 从图6-8中删除“duck”之后的链表情形。“duck”节点现在位于空闲链表的首节点，也是惟一的节点

无论何时发生插入（包括push_front或者push_back），都将检测空闲链表。如果空闲链表非空，那么将使用它的首节点进行插入操作，并将它从空闲链表中删除。如果空闲链表为空，
222 ~ 223 就使用next_avail的节点，除非next_avail=last。如果是那样，将分配一个新的缓冲区并连接到缓冲区链表的开头，然后把这个新缓冲区的第一个节点分配出去。

这些各种各样的缓冲区和链表的空间不会被回收，除非清除链表。因此，如果应用创建一个大的链表，然后删除几乎全部的项，那么所有的链表空间仍被占用着。

实验15还有很多惠普的实现的细节，实验16用一个实验比较了向量、双端队列和链表。现在已经看到了几种类型的迭代器——随机访问的和双向的，下面将学习标准模板库的迭代器的组织。

实验15：更多list类的实现细节 **（所有实验都是可选的）**

224 **实验16：计时顺序容器** **（所有实验都是可选的）**

实验17：迭代器，第二部分 **（所有实验都是可选的）**

像在6.1.4节中所说的，惠普的实现中主要的也是微妙的特点之一就是头节点。这个节点

使我们不需要用专门的手段进行头尾的插入或删除。对每个给定的节点，总是有一个节点在它的前面，另一个节点在它的后面。特别是，头节点总是在链表第一个节点的前面，而且也总是在链表最后一个节点的后面。

6.1.6节中考察list类的其他实现时将更清晰地说明头节点的优点。这些实现比list类的惠普实现要简单，但是效率低。在编程项目6.2里给读者提供了机会去完成6.1.6节概述的一个实现。

6.1.6 list类的其他实现

225

现在开发list类的其他几个实现。为了简单起见，将依靠new和delete运算符的堆管理器实现分配和回收链表节点。第2章的单链接Linked类怎么样呢？这里是Linked类中的字段：

```
protected:
    struct Node
    {
        T item;
        Node* next;
    }; // 结构Node

    Node* head;
    Node* tail; // 这个字段是在实验8中加入的
    long length;
```

能否扩充Linked类，使它满足list类所有的方法接口呢？这个问题还伴随着一些list方法的后置条件——特别是时间估算。

例如，list类的pop_back方法的后置条件没有显式地包含时间估算。根据约定，这意味着该方法的任何实现，worstTime(*n*)必须是常数。那么pop_back方法用于图6-11的Linked容器会怎样？

pop_back方法将必须令尾节点之前节点的next字段为NULL。为此需要一个循环，因此worstTime(*n*)将和*n*成线性关系。这将违背list类中pop_back方法接口的常数时间需求；因此必须放弃list类的单链表实现。

如果修改Linked类，令它成为双向链接的呢？图6-12显示了从图6-11所示的三个项的链表改造过来的结果。



图6-11 有三个项的Linked容器。在这个类中将怎样定义pop_back方法呢

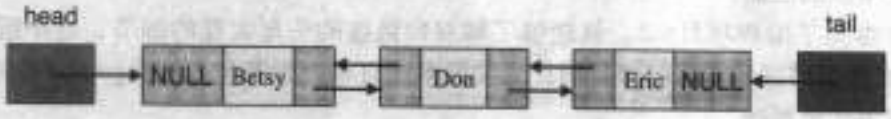


图6-12 有head和tail字段的双向链式容器

这里是字段定义：

```
protected:
    struct Node
    {
        T item;
        Node* next;
        Node* prev;
    }; // 结构Node

    Node* head;
    Node* tail;
226 long length;
```

缺省构造器将简单地把head和tail设置成NULL，length设置成0。但是现在要密切注意NULL引用。例如，push_front方法将有一个空容器的特殊情况。

```
void push_front (const T& x)
{
    Node* temp_head = new Node;
    (*temp_head).item = x;
    (*temp_head).next = head;
    (*temp_head).prev = NULL;
    length++;
    if (head == NULL)
    {
        head = temp_head;
        tail = head;
    } // if
    else
    {
        (*head).prev = temp_head;
        head = temp_head;
    } // else
} // 方法push_front
```

需要对push_back做相似的测试。对pop_front和pop_back，将需要用特殊的手段删除容器中惟一的项。同样，insert方法将首先测试：

```
227 if(head==NULL||position.node==head)
```

这个实现的最后一行就是在链表头尾插入和删除的特殊情况所遭遇的情形。惠普的实现通过一个虚节点——头节点——避免了特殊情况，这个头节点既在链表第一个节点的前面，又在最后一个节点的后面。

如果你参与了编程项目6.2，就能够了解双向链接的头尾实现的细节。这个版本使用了new和delete运算符进行空间管理，因此它的效率比自带内存管理方法（get_node和put_node方法）的惠普实现要低。

6.2节中说明了实现细节的本质，并观察了list类的一个应用：一个简单的文本编辑器。

6.2 链表应用：一个行编辑器

开发一个行编辑器来说明list类。**行编辑器**是一个程序，它逐行地操作文本。假设每行中至多有75个字符长，文本的第一行看作行0，选定的某一行称作**当前行**。每个编辑命令以一个$标志开始，而且只有编辑命令才以$开始，共有八个编辑命令。这里给出了其中四个命令；剩余的四个将在编程项目6.1中介绍。

1. $Insert

随后的直到下一条编辑命令之前的每一行都将被插入到当前行之后的文本中。插入的最后一行成为当前行。如果调用$Insert时文本为空，那么插入的就是文本中仅有的行。例如，假设文本为空而且有如下行：

```
$Insert
Water, water every where,
And all the boards did shrink;
Water, water every where,
Nor any drop to drink.
```

那么插入后文本将变成如下形式，其中“>”指示了当前行：

```
  Water, water every where,
  And all the boards did shrink;
  Water, water every where,
>Nor any drop to drink.
```

再举一个例子，假设文本是：

```
  Now is the
>time for
  citizens to come to
  the
  aid of their country.
```

228

那么序列

```
$Insert
all
good
```

将导致文本变成

```
  Now is the
  time for
  all
>good
  citizens to come to
  the
  aid of their country.
```

2. $Delete k m

删除文本中行k和m之间的每一行，包括行k和行m。如果当前行在这个范围内，那么新的当前行将是第$k-1$行；否则，当前行和命令执行前是相同的。例如，假设文本是

```
  Now is the
  time for
  all
>good
  citizens to come to
  the
  aid of their country.
```

命令

```
$Delete 3 5
```

将使文本变成

```
  Now is the
  time for
>all
  aid of their country.
```

如果k的值是0并且删除的行中包括当前行，那么说明删除之后，当前行在所有文本行的前面。因此，如果紧随着出现$Insert命令，就将在文本第一行的前面进行插入。例如，假设文本是

```
  a
  s
>p
  a
  r
  k
```

229 而命令是

```
$Delete 0 2
$Insert
q
u
```

那么文本将变成

```
  q
>u
  a
  r
  k
```

在删除之后插入之前，当前行在所有文本行的前面。插入在文本的开头插入了两行，因此当前行现在是“u”。适当情况下将输出下面的错误消息:

```
*** Error: The first line number > the second.
*** Error: The first line number < 0.
*** Error: The second line number > last line number.
*** Error: The command is not followed by two integers.
```

3. $Line m

行号是m的行成为当前行。例如，如果文本是:

```
  Mairzy doats
  an dozy doats
>an liddle lamsy divy.
```

那么命令

```
$Line 0
```

将使行0成为当前行:

```
>Mairzy doats
  and dozy doats
  and liddle lamsy divy.
```

命令

```
$Line -1
```

后面跟着$Insert方法时，可以在文本开头插入行。如果*m*小于-1或是大于文本最后一行的行号将输出错误信息。参阅命令2。

4. $Done

这终止了文本编辑器的运行。为了方便起见，我们将输出最终的文本。任何非法的命令将输出一个错误消息，比如“$End”、“$insert”或“?Insert”。 230

系统测试1（输入用黑体表示）

```
Please enter a line:
$Insert
Please enter a line:
This is line zero.
Please enter a line:
This is line one.
Please enter a line:
This is line two.
Please enter a line:
$Line 1
Please enter a line:
$Insert
Please enter a line:
This is line 1.5.
Please enter a line:
This is line 1.6.
Please enter a line:
This is line 1.7.
Please enter a line:
This is line 1.8.
Please enter a line:
$Delete 1 3
Please enter a line:
$Done
```

```
Here is the final text:
  This is line zero.
  This is line 1.7.
>This is line 1.8.
  This is line two.

Please press the Enter key to close this output window.
```

系统测试2（输入用黑体表示）

```
Please enter a line:
Insert
*** Error: Not one of the given commands
Please enter a line:
$Insert
Please enter a line:
a
Please enter a line:
b
Please enter a line:
$line
*** Error: Not one of the given commands
Please enter a line:
$Line 2
*** Error: The line number must be less than the text size.
Please enter a line:
$Done
Here is the final text:
  a
>b

Please press the Enter key to close this output window.
```

231

6.2.1 Editor类的设计

为了决定Editor类应当包含哪些方法，首先要解决的就是一个编辑器必须做什么？在给出的编辑器命令中，很明显有一些职责：

- 解析行，判断它是否是一个合法的命令。
- 检测命令中的错误。
- 管理文本。

这样就可以开始了。parse方法将解释读入的行。这个方法将行作为它惟一的参数。parse应当返回什么？对某些命令而言，如$Insert、$Delete和$Line，如果不能执行，应当返回一个错误消息。而对有些命令来说，如$Done，应返回完整的文本。在编程项目6.1中，有一个命令$Print，它要么返回一个错误消息，要么返回一些文本。如何区分错误消息和文本呢？根据问题的规格说明，文本行不能以“$”开始，因此通过在任意错误消息前放置该符号可以断定

谁是什么。parse方法将返回一个字符串，如果第一个字符是“$”就代表错误消息，对$Done命令返回的则是文本（对$Insert、$Delete和$Line命令，在没有错误时将插入一行或返回一个空行）。

command_check方法将检测每条命令中的错误，并且对这四条命令中的任意一条都有一个单独的方法。下面是方法接口：

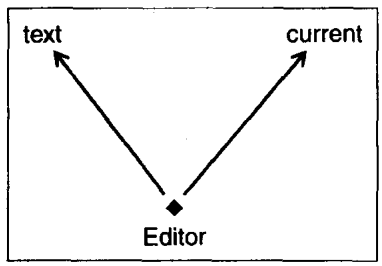
```
//后置条件: 这个Editor为空。
void Editor();                                                    232

//后置条件: 如果line是一个合法的命令，那么执行该命令并返回运行结果。
//          如果line是将被插入的行，那么就试着插入并返回结果。否则，返回命令
//          非法错误消息。
string parse(const string& line);

//后置条件: 检测line中的错误。如果没有找到错误，就处理命令并返回结果。
//          否则，返回一个错误消息。
string command_check(const string&line);

//后置条件: 如果line不是太长，就将它插入这个Editor并返回一个空行。
//          否则，返回一个错误消息。
string insert_command(const string&line);

//后置条件: 如果可能就删除行k到行m之间的文本，并返回一个空行。
//          否则，返回一个错误消息。
string delete_command(int k, int m);

//后置条件: 如果可能，将索引号为m的行设置成为文本的当前行，并返回一个空行。
//          否则，返回一个错误消息。
string line_command(int m);

//后置条件: 完成编辑器的运行并返回文本。
string done_command();
```

在开始定义这6个方法之前，必须决定将使用哪些字段。其中一个字段将保存文本，因此称它为text。文本将是一个序列，而且通常需要在文本内部进行插入和/或删除，因此text应当是list类中的一个对象。（令人惊异！）

为了确定当前行，可以使用一个整数字段currentLineNumber，或是一个迭代器字段current。某些指令（如$Delete和$Line）操作行号，但是链表的插入和删除需要迭代器，因此很难说哪种更好。这两种情形如何呢？我们将试图找出答案，即使每次插入和删除都必须修改这两个字段。bool字段inserting将判断输入行是被插入编辑器还是被看作一个命令。所有 233
的字段都是受保护的，允许子类使用它们：

```
protected:
    list<string> text;
    list<string>::iterator current;
    int currentLineNumber;
    bool inserting;
```

下面是两个复合示例的依赖关系图：



既然了解了方法接口和字段，就可以定义这些方法来实现Editor类了。

6.2.2 Editor类的实现

缺省构造器显式地初始化除text之外的字段，text是用它的缺省构造器初始化的。

```
Editor::Editor( )
{
    current = text.begin( );
    currentLineNumber = -1;
    inserting = false;
} // 缺省构造器
```

parse方法决定了行代表的是一个命令还是将被插入的文本。如果都不是，那么命令开始字符（'$'）将加到错误消息前。

```
string Editor::parse (const string& line)
{
    if (line.substr (0, 1) != COMMAND_START)
        if (inserting)
            return insert_command (line);
        else
            return COMMAND_START +
                MISSING_COMMAND_ERROR +
                COMMAND_START;
    return command_check (line);
} // parse
```

234

command_check消息将命令和命令参数分隔开并调用相应的命令。对$Delete和$Line命令，必须从命令行中提取行号。例如，假设行包含

$Line 173

用string方法find确定空格的位置。子字符串"173"从第一个空格位置+1开始，并一直到字符串的末尾。字符串方法substr返回这个子字符串，它由string方法c_str转换成了一个字符数组。最后，atoi函数将数字字符数组转换成一个整数。

```
string Editor::command_check (const string& line)
{
    string command;
```

```
int blank_pos1 = line.find (BLANK),
    blank_pos2;
if (blank_pos1 >= 0) // line至少有一个变元
      command = line.substr (0, blank_pos1);
else
      command = line;
if (command == INSERT_COMMAND)
{
      inserting = true;
      return BLANK;
} // $Insert
else
{
      int k,
          m;

      inserting = false;

      if (command == DELETE_COMMAND)
      {
            // 查找k和m
            if (blank_pos1 >= 0)
            {
                  blank_pos2 = line.find (BLANK, blank_pos1 + 1);
                  if (blank_pos2 >= 0)
                  {
                        k = atoi (line.substr (blank_pos1 + 1,
                              blank_pos2 - blank_pos1 - 1).c_str( ));
                        m = atoi (line.substr (blank_pos2 + 1).c_str( ));
                        return delete_command (k, m);
                  } // 给定了2个行号
                  return COMMAND_START +
                        MISSING_NUMBER_ERROR;
            } // 至少给定了1个行号
            return COMMAND_START +
                  TWO_NUMBERS_ERROR;
            } // $Delete
      else if (command == LINE_COMMAND)
      {
            // 查找m
            if (blank_pos1 >= 0)
            {
                  m = atoi (line.substr (blank_pos1 + 1).c_str( ));
                  return line_command (m);
            } // 给定行号
            return COMMAND_START + MISSING_NUMBER_ERROR;
      } // $Line
      else if (command == DONE_COMMAND)
            return done_command( );
```

235

```
            return COMMAND_START + ILLEGAL_COMMAND_ERROR;
        } // 不是一个插入命令
    } // command_check
```

最后，开始实质性的工作——管理list对象text。花费常数时间的insert_command方法检测一行是否太长，如果不是，就将line插入到当前行之后的文本中：

```
string Editor::insert_command (const string& line)
{
    if (line.length( ) > MAX_LINE_LENGTH)
        return COMMAND_START + LINE_TOO_LONG_ERROR;
    current = text.insert (++current, line);
    currentLineNumber++;
    return BLANK;
} // insert
```

delete_command首先检测像k<0这样的错误。如果k≥0，从文本开头的迭代器first开始，然后k次增加first。然后删除文本中的后m-k个项。这个循环之后，需要修改current和currentLineNumber。注意，如果没有currentLineNumber字段，那么决定是否修改current将是有点困难的。下面是delete_command方法的定义：

```
string Editor::delete_command (int k, int m)
{
if (k < 0)
    return COMMAND_START + FIRST_TOO_SMALL_ERROR;
if (m >= (int)text.size( ))
    return COMMAND_START + SECOND_TOO_LARGE_ERROR;
if (k > m)
    return COMMAND_START +
        FIRST_GREATER_THAN_SECOND_ERROR;
list<string>::iterator first = text.begin( );
for (int i = 0; i < k; i++)
    first++;
for (int i = k; i <= m; i++)
    text.erase (first++);
if (currentLineNumber >= k && currentLineNumber <= m)
{
    currentLineNumber = k - 1;
    current = --first;
} // if
else if (currentLineNumber > m)
    currentLineNumber -= m + 1 - k ; // current未改变
return BLANK;
} // delete_command
```

236

delete_command将花费多长时间？用n代表text中的行数。最坏情况下，k=0且m=n-1，也就是说将删除每一行，每次erase调用将花费常数时间，因此worstTime(n)和n成线性关系。平

均情况下，m的数值大约是$n/2$，因此迭代次数以及averageTime(n)也和n成线性关系。

line_command是增加还是减小当前行号，这依赖于currentLineNumber和n的关系：

```
string Editor::line_command (int m)
{
    if (m < -1)
        return COMMAND_START + FIRST_TOO_SMALL_ERROR;
    if (m >= (int)text.size( ))
        return COMMAND_START + FIRST_TOO_LARGE_ERROR;
    if (currentLineNumber < m)
    {
        for (int i = currentLineNumber; i < m; i++)
            current++;
        currentLineNumber = m;
    } // if
    else
    {
        for (int i = currentLineNumber; i > m; i--)
            current--;
        currentLineNumber = m;
    } // else
    return BLANK;
} // line_command
```

237

令n代表文本的行数。在最坏情况下，currentLineNumber=-1且m=$n-1$，current将迭代通过text的每一行，因此worstTime(n)和n成线性关系。平均情况下，currentLineNumber和m之间的距离大约是$n/2$，因此平均迭代次数以及averageTime(n)仍和n成线性关系。

done_command方法将返回文本，包括当前行标志：

```
string Editor::done_command( )
{
    const string FINAL_MESSAGE = "Here is the final text: \n"; string
            text_string = FINAL_MESSAGE;
    if (currentLineNumber == -1)
        text_string += " >\n";
    for (list<string>::iterator itr = text.begin( ); itr != text.end( ); itr++)
        if (itr == current)
            text_string += " >" + *itr + "\n";
        else
            text_string += " " + *itr + "\n";
    return text_string;
} // done_command
```

这个方法迭代通过文本的每一行，因此worstTime(n)、averageTime(n)都和n成线性关系。

main函数处理行编辑器应用中所有的输入和输出。

main函数定义了一个Editor对象editor，然后为输入的每一行调用editor.parse(line)。返回的结果中如果有打头的“$”就把它去掉，然后输出。输入一行表示一个问题。提取运算符>>

可以用来读入一个命令，但是我们并不清楚这时的行是否还包含行号，比如$Delete和$Line命令中就有行号。它是由没有输入语句的Editor类的方法决定的。因此需要main函数中读入一整行。C++提供了getline函数实现这个目的。函数接口是：

```
//后置条件：从isStream中去掉开头的空白字符，从当前字符到'\n'都被存入一行。
238 istream& getline(istream& inStream,string& line);
```

然后Editor方法可以再细分此行。

下面是定义，使用了一个do循环来保证在循环终止前输出$Done命令的结果。

```
int main( )
{
    const string PROMPT = "Please enter a line: ";
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    Editor editor;

    string result;

    string line;

    do
    {
        cout << PROMPT<< endl;
        getline (cin, line);
        result = editor.parse (line);
        if (result.substr (0, 1) != COMMAND_START)
            cout << result << endl << endl;
        else
            cout << result.substr (1) << endl << endl;
    } // do
    while (line != DONE_COMMAND);

    cout << CLOSE_WINDOW_PROMPT;
    cin.get( );
    return 0;
} // main
```

这里为文本的每一项进行一次do循环迭代（还有其他的循环迭代）。因此worstTime(n)至少是和n成线性关系。要获得更好的估算，需要了解命令序列。

不改变Editor类，也可以将程序修改为接收文件输入并向文件发送输出。下面是修改后的main函数。

```
int main( )
{
    const string IN_PROMPT = "Please enter the path for the input file: ";

    const string OUT_PROMPT =
        "Please enter the path for the output file: ";

    const string ECHO = "The line was: ";
```

```
    const string CLOSE_WINDOW_PROMPT =
          "Please press the Enter key to close this output window.";

    Editor editor;

    string result;

    string inFileName,
           outFileName,
           line;

    fstream inFile,
            outFile;

    cout << IN_PROMPT;
    cin >> inFileName;
    cout << OUT_PROMPT;
    cin >> outFileName;
    inFile.open (inFileName.c_str( ), ios::in);
    outFile.open (outFileName.c_str( ), ios::out);

    do
    {
          getline (inFile, line);
          outFile << ECHO << line << endl;
          result = editor.parse (line);
          if (result.substr (0, 1) != COMMAND_START)
                outFile << result << endl << endl;
          else
                outFile << result.substr (1) << endl << endl;
    } // do
    while (line != DONE_COMMAND);
    outFile.close( );

    cout << CLOSE_WINDOW_PROMPT;
    cin.get( );
    return 0;
} // main
```

239

所有的相关文件参阅本书网站的源代码链接。

总结

本章的焦点是list类。链表是顺序容器，它缺乏像向量和双端队列那样的随机访问能力。但是它的内部插入和删除只花费常数时间——而在向量和双端队列中是线性时间。保持这个常数时间特性是因为，insert和erase方法需要一个位于插入或删除位置的迭代器参数。

240

一个简单的行编辑器应用程序利用了list类的能力，因此可以快速地在链表的任何位置进行多个插入和删除。

习题

6.1 a. 假设有如下定义：

```
list<char> letters;
list<char>::iterator itr;
```

给出发送下面的每条消息之后链表中的字符序列:

```
itr = letters.begin( );
letters.insert (itr, 'f');
letters.insert (itr, 'e');
itr++;
letters.insert (itr, 'r');
itr++;
itr++;
letters.insert (itr, 't'); // Hint: we now have e,r,f,t
letters.insert (itr, 'e');
letters.erase (letters.begin( ));
itr--;
itr--;
letters.insert (itr, 'p');
itr++;
letters.insert (itr, 'e');
itr = letters.end( );
itr--;
letters.insert (itr, 'c');
```

b. 编写代码输出a部分中letters的最终内容。

c. 重做a部分，letters用字符数组取代一串字符。

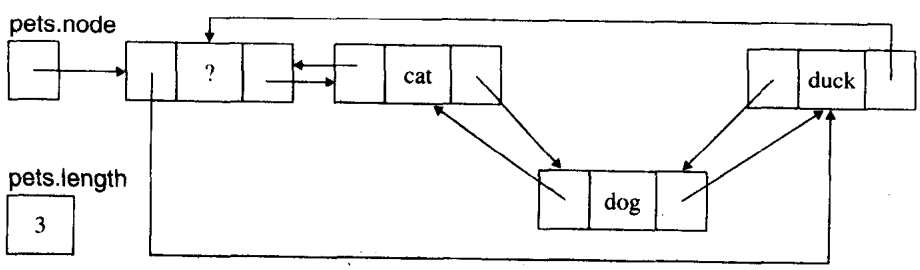
d. 重做a部分，letters用字符串取代一串字符。

e. 重做a部分，letters用字符向量取代字符串。

f. 重做a部分，letters用字符双端队列取代字符串。

6.2 根据访问、插入和删除的大O时间比较链表和向量以及双端队列。

241 6.3 下面再看一下图6-4中的链表，求在该表上执行下列消息的影响:




```
list<string>::iterator position = pets.begin( );
pets.push_front("bunny");
pets.erase (position);
pets.push_back ("frog");
```

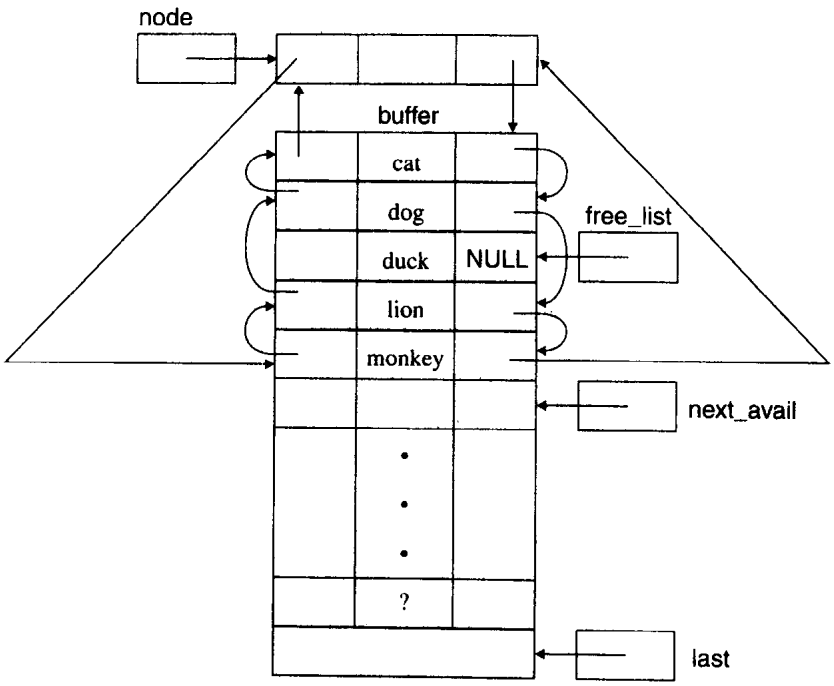
6.4 向量、双端队列和链表中，哪一个能够提供更快速的very_long_int类实现？为什么？

6.5 假设list类利用了堆管理器——使用new和delete运算符进行内存分配和回收。定义insert方法和有一个参数的erase方法。

6.6 在Editor类的delete_command方法中，变量k和m是**int**类型的。为什么当k类型是**unsigned**时会发生错误？

6.7 假设myList是一个list对象，其项的类型是**double**。编写代码以反序输出项。

6.8 下面再看一下图6-10:



242

如果从这个链表中删除“lion”，下面节点的next字段将指向哪里:

a. 包含“cat”项的节点?

b. 包含“dog”项的节点?

c. 包含“duck”项的节点?

d. 头节点?

提示 *被删除的节点成为空闲链表的第一个节点。*

6.9 如果list类的设计中包含head和tail字段，那么通过把链表连成环，可以避免每个链表节点的prev和next字段出现NULL值。也就是说，在一个非空链表中，头节点的prev字段将指向尾节点，而尾节点的next字段将指向头节点。将这个设计应用在begin、end和push_front方法的实现上有什么影响?

243

编程项目6.1：扩展Editor类

扩展Editor类，使它包含有下面接口的方法（6.2节已介绍了4个命令）:

5. **$Change**

%X%Y%

在当前行中，出现给定字符串X的每个地方都将被替换成给定字符串Y。例如，假设当前行是

bear ruin'd choirs,wear late the sweet birds sang

那么命令

$Change
%ear%are%

将使当前行变成

bare ruin'd choirs,ware late the sweet birds sang

如果接着发出命令

$Change
%wa%whe%

将得到

bare ruin'd choirs,where late the sweet birds sang

注意

1) 如果X或Y包含一个百分号，终端用户应选择另一个分隔符。例如，

$Change
#0.16#16%#

2) Y给定的字符串可能是一个空串。例如，如果当前行是

aid of their country.

那么命令

$Change
%of%%

将当前行修改成

aid their country.

3) 如果分隔符出现次数小于3，将输出如下的错误消息:

244 ***Error:Delimiter must occur three times.

6. **$Last**

将输出文本最后一行的行号。例如，如果文本是

I heard a bird sing
>in the dark of December.
A magical thing
and a joy to remember.

那么命令

$Last

将输出3。文本和当前行位置保持不变。

7. **$Print k m**

输出文本中从行k到行m之间（包括行k和行m）的每一行的行号和内容。例如，如果文本是

Winston Churchill once said that
>democracy is the worst
form of government
except for all the others.

那么命令

$Print 0 2

将输出

0Winston Churchill once said that
1democracy is the worst
2form of government

文本和当前行位置保持不变。正如命令2一样，如果k比m大；或是k小于0或m大于文本最后一行的行号，那么就输出一个错误消息。

系统测试1（样本输入用黑体表示）

```
Please enter a line:
$Insert
Please enter a line:
You can fool
Please enter a line:
some of the people
Please enter a line:
some of the times,
Please enter a line:
but you cannot foul
Please enter a line:
all of the peeple
Please enter a line:
all of the time.
Please enter a line:
$Line 2
Please enter a line:
$Print 2 1
*** Error: The first line number > the second.
Please enter a line:
$Print 2 2
2 some of the times,
Please enter a line:
$Change %s%%
Please enter a line:
$Print 2 2
2  ome of the time,
Please enter a line:
```

$Change %o%so
*** Error: Delimiter must occur three times.

Please enter a line:
$Change %o%so%

Please enter a line:
$Print 2 2

2 some sof the time,

Please enter a line:
Change
*** Error: Command must begin with $.

Please enter a line:
$Change %sof%of%

Please enter a line:
$Print 2 2

2 some of the time,

Please enter a line:
$Line -1

Please enter a line:
$Insert

Please enter a line:
Lincoln once said that

Please enter a line:
you can fool

Please enter a line:
some of the people

Please enter a line:
all the time and

Please enter a line:
all of the time and

Please enter a line:
$Last

10

Please enter a line:
$Print 0 10

0 Lincoln once said that
1 you can fool
2 some of the people
3 all the time and
4 all of the time and
5 You can fool
6 some of the people
7 some of the time,
8 but you cannot foul
9 all of the people
10 all of the time.

Please enter a line:
$Line 5

Please enter a line:
$Change %Y%y%

Please enter a line:
$Print 5 5

5 you can fool

Please enter a line:
$Line 6

Please enter a line:
$Change %some%all%

Please enter a line:
$Print 6 6

6 all of the people

Please enter a line:
$Line 8

Please enter a line:
$Change %ul%ol%

Please enter a line:
$Print 8 8

8 but you cannot fool

Please enter a line:
$Line 9

Please enter a line:
$Change %ee%eo%

Please enter a line:
$Print 9 9

9 all of the people

Please enter a line:
$Delete 3 3

Please enter a line:
$Print 0 10
*** Error: The second line number is greater than the number of lines in the text.

Please enter a line:
$Last
9

Please enter a line:
$Print 0 9

0 Lincoln once said that
1 you can fool
2 some of the people
3 all of the time and
4 you can fool
5 all of the people
6 some of the time,
7 but you cannot fool
8 all of the people
9 all of the time.

Please enter a line:
$Done

Here is the final text:
Lincoln once said that
you can fool
some of the people
all of the time and
you can fool
all of the people
some of the time,
but you cannot fool
>all of the people
all of the time.

Please press the Enter key to close this output window.

系统测试2（样本输入用黑体表示）

Please enter a line:
$Insert

Please enter a line:
Life is full of

Please enter a line:
successes and lessons.

246
~
249

Please enter a line:
$Delete 1 1

Please enter a line:
$Insert

Please enter a line:
wondrous oppurtunities disguised as

Please enter a line:
hopeless situations.

Please enter a line:
$Last

2

Please enter a line:
$Print 0 2

0 Life is full of
1 wondrous oppurtunities disguised as
2 hopeless situations.

Please enter a line:
$Line 1

Please enter a line:
$Change %ur%or%

Please enter a line:
$Print 0 2

0 Life is full of
1 wondrous opportunities disguised as

```
2 hopeless situations.

Please enter a line:
$Done

  Here is the final text:
  Life is full of
>wondrous opportunities disguised as
  hopeless situations.

Please press the Enter key to close this output window.
```

250

编程项目6.2：list类的另一种设计和实现

实现6.1.6节中描述的双向链表，即带头尾字段的list类的设计。这个项目中仅需要实现6.1.1节中给出的前13个方法。设置一个驱动程序测试这个实现。还需要实现至少几个迭代器运算符：*、!+和++（前加或后加都可以）。

251

第7章　队列和堆栈

本章介绍了标准模板库中另外两个数据结构：queue类和stack类。由于只能以有限的方式访问或修改队列和堆栈，因此它们每个只有很少的（少于10个）方法接口，而vector、deque和list类都至少包含40个方法接口；而且这些类的实现是非常直截了当的。实际上，这些类“配接”一些基础容器类的实现。例如，任何有push_back、pop_back、back、empty和size方法的容器类都可以作为stack类的基础类。队列和堆栈有广泛的应用。我们将从queue类开始，因为它的大部分应用都是通用的，而堆栈主要是用在计算机系统中。

目标

1) 能够定义队列和堆栈的特性。
2) 理解queue和stack类之所以被称作“容器配接器”（container adaptor）的原因。
3) 考察计算机仿真中队列的作用。
4) 考察递归的实现中以及中缀表示法向后缀表示法转换的实现中堆栈的作用。

253

7.1　队列

队列是项的有限序列，满足：

1) 插入只允许从尾部进行。
2) 删除、检索和修改只允许从头部进行。

队列中的项是按照时间排序的：先入，先出。

队列中的项是按照时间排序的：插入的第一项（在尾部）最终将是第一个（从头部）被删除、检索或修改的项。第二个插入的项将是第二个被删除、检索或修改的项，依次类推。队列的这个定义属性有时被称为“先到，先服务”，“先入，先出”，或简称FIFO。图7-1显示了经过几次插入和删除的队列。

队列的示例有很多，如：

展示窗中排成一行的汽车。

排队等待购买球赛入场券的球迷。

在超市中等待付款的顾客。

在机场等待起飞的飞机。

254

我们可以继续举出许许多多队列的例子。7.3节将介绍计算机仿真领域中队列的一个应用。

7.1.1　queue类的方法接口

和标准模板库中其他的容器类一样，queue类是一个模板类：

```
template<class T, class Container=deque<T>>
```

除了项的类型T之外，队列的每个实例还包含一个容器模板，缺省为deque<T>。这暗示

着deque可以平均以常数时间处理队列的属性定义：尾部插入，头部删除和头部访问。考虑实现时我们将观察队列的这个性质。

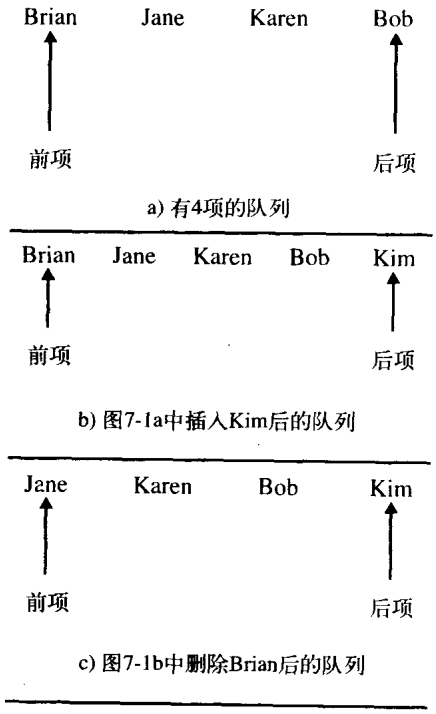


图7-1 经过几次插入和删除的队列

与第5章和第6章的顺序容器类不同，queue容器类有一个最小接口。下面是所有方法的接口：

```
1. //前置条件：用这个Container的拷贝初始化队列。
   explicit queue(const Container&=Container());
```

注意 保留字**explicit**只能和构造器一起使用，它指示编译器不应当在构造器语句中执行自动类型转换[⊖]。也就是说，如果提供了构造器的变元，那么这个变元必须是第二个模板变元指定的Container类的一个容器对象。例如，下面是一些合法的队列定义：

```
a. queue<string> q1;
b. queue<string,deque<string>> q2(q1);
c. queue<string> q3(q1);
d. queue<double,list<double>> q4;
e. queue<double,list<double>> q5(q4);
```

q1、q2和q3的定义是等价的，q4和q5的定义也是等价的。而下面的两个定义是不合法的：

```
f. queue<string,list<string>> q6(q1);
                //构造器变元应当是一个list
```

⊖ 回忆在第6章的list类的iterator类中，自动类型转换将指针转换成了迭代器。

g. queue<**double**>>q7(q4);
//构造器变元应当是一个deque

255

2. //后置条件：如果这个队列为空就返回真。否则，返回假。
bool empty() **const**;
3. //后置条件：返回这个队列对象的项的数量。
unsigned size() **const**;
4. //后置条件：将项x插入到这个队列的尾部。averageTime(n)是常数。
// worstTime(n)是O(n)，但是对n次连续插入，全部n次插入的worstTime(n)
// 只是O(n)。也就是说，amortizedTime(n)是常数。
void push (**const** value_type& x);

注意1 这个方法经常被称作“enqueue”（入列）。

注意2 **typedef**将T的含义声明为value_type。

5. //前置条件：这个队列非空。
//后置条件：返回对这个队列开头项的引用。
T& front();

注意 因为返回值是一个引用，所以这个方法可以用来修改队列开头的项。例如，如果my_queue是一个字符串非空队列，

my_queue.front()="Courtney";

把my_queue中存储的第一项替换成“Courtney”。

6. //前置条件：这个队列非空。
//后置条件：返回对这个队列开头项的一个常量引用。
const T& front();

注意 因为返回值是一个常量引用，所以这个方法不能用来修改队列的开头项。但是这个方法可以获取开头项。例如，如果my_queue是一个非空队列，

cout<<my_queue.front();

将输出my_queue的第一个项。

256

7. //前置条件：这个队列非空。
//后置条件：调用前位于这个队列开头的项将从这个队列中删除。
void pop();

注意1 pop方法不返回弹出的项。为了获取这个项，在调用pop()之前调用front()。

注意2 这个方法经常被称作“dequeue”（出列），注意不是“deque”（双端队列）。

8. //前置条件：这个队列非空。
//后置条件：返回对这个队列尾部项的引用。
T& back();

注意1 队列的定义不需要一个back方法，但是标准模板库中包括了这个方法。

注意2 这个方法可以用来修改队列中插入的最后一项。

```
9. //前置条件：这个队列非空。
   //后置条件：返回对这个队列尾部项的一个常量引用。
   const T& back();
```

回忆前面的约定：当没有给出时间估算时，方法的worstTime(n)就是常数。因此对除了push之外的所有方法，worstTime(n)都是常数。push方法的worstTime(n)是O(n)，但是amortizedTime(n)是常数。对queue类的所有方法而言，averageTime(n)都是常数。

queue类没有一个关联容器类。为什么没有呢？因为根据队列的定义，queue对象中惟一能被访问的项就是queue开头的项。因此，如果能获取任意一项将违背queue的定义。(实际上，back方法就违背了队列的定义)。

在7.1.2节中将看到使用queue类是很容易的。

7.1.2 使用queue类

queue只有很少的方法，而且没有迭代器。但这并不意味着不能输出队列，只是需要做些额
257 外的工作。例如，这里有一个程序生成图7-1所示的队列。把队列作为一个值形参，printQueue函数就可以在不破坏队列的前提下，输出队列的拷贝。

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

void printQueue (queue<string> names)
{
      cout << endl << endl << "Here is the current queue:" << endl;
      while (!names.empty( ))
      {
            cout << names.front( ) << endl;
            names.pop( );
      } // while
} // 函数printQueue

int main( )
{
      const string CLOSE_WINDOW_PROMPT =
            "Please press the Enter key to close this output window.";

      queue<string> names;

      names.push ("Brian");
      names.push ("Jane");
      names.push ("Karen");
      names.push ("Bob");
      printQueue (names);

      names.push ("Kim");
      printQueue (names);

      names.pop( );
```

```
        printQueue (names);

        cout << endl << endl << CLOSE_WINDOW_PROMPT;
        cin.get( );
        return 0;
    } // main
```

在主函数main中，printQueue调用中的变元names是不会改变的，但是printQueue方法必须对这个变元做一个拷贝。

正如读者可能猜想到的，queue类的实现是相当简单的，简单到令人惊讶的地步！7.1.3节显示了这些早已做过的工作。 258

7.1.3 容器配接器

根据已经了解的标准模板库知识可以推想，在满足给定方法接口的前提下，编译器的编写者能自由地以任何方式设计和实现queue类。情况并不是这样。实际上，queue类的设计和实现是标准模板库的一部分，而不是编译器编写者可选的。

容器配接器C使用一些基础的容器对象定义C的方法。

queue类是容器配接器的一个示例。**容器配接器C**将一些基础容器转换成类C的容器。容器配接器——queue、stack和priority_queue——与标准模板库的其他部分的处理是截然不同的。它们的方法定义必须调用基础容器类的方法。

在queue类情况下，所有需要基础类Container所做的是它应当支持empty、size、front、push_back和pop_front方法（以及back方法）。例如，下面给出了标准模板库中queue类的一部分：

```
template <class T, class Container = deque<T>>
class queue
{
    protected:
        Container c;
    public:
        void pop( ) { c.pop_front( ); }
        const T& front( ) const { return c.front( ); }
```

作为queue类的使用者或实现者，所做的惟一选择就是用什么作为基础容器。list类可以作为基础类来使用。回想一下，list类中有size、empty、push_back、pop_front、front和back方法。

deque类也可以作为基础类；实际上，deque类就是缺省基础类。在deque类的惠普实现中，size、empty、pop_front、front和back方法花费常数时间。push_back方法的averageTime(n)是常数。worstTime(n)发生在必须调整映射数组大小时，它和n成线性关系，但是随后的n次尾部插入，每次只用常数时间。也就是说，对deque类中的push_back方法而言，amoritizedTime(n)是常数。

vector类不能满足基础类的要求：它没有pop_front方法。这不是标准模板库创建者的疏忽。而且，这个有意的疏漏意味着对用户的警告，就是在向量中删除开头的项时，worstTime(n)和n成线性关系（指所有基于惠普实现的实现）。

如果有一些其他的包含这些方法的容器类，可以根据它们定义一个queue对象。例如，只
259 需要稍做一点工作，就能扩展Linked类，使它满足配接类的要求，详情参阅习题7.7。7.1.4节为队列容器配接器开发了另一个配接类。

7.1.4 一个接近的设计

前面已经提到vector类不能作为队列的基础类，因为vector类缺乏pop_front方法。另一种选择也是基于数组的，它直接操作数组字段，使得平均情况下的push_back和pop_front调用只花费常数时间。在这个基础类——queueArray——的设计中有五个字段：

```
T[ ] data;
unsigned size;
int head,
    tail,
    max_size;     // data中可以存储项的最大数量
```

data数组保存项，size保存项的数量，max_size保存项的最大数量（在必须调整大小之前的）。开头的项总是在下标head处，但是head并非总是值0。基本的，发生push_back和pop_front调用时，队列就沿着数组data“滑下”。head字段包含了data的下标，即队列开头项的下标；而tail字段包含了队列尾部项的下标。将tail初始化为-1表示队列为空。图7-2显示了在这些字段上调用push_back两次及调用pop_front一次的作用。问号表示了一个未知或不相关的数值。

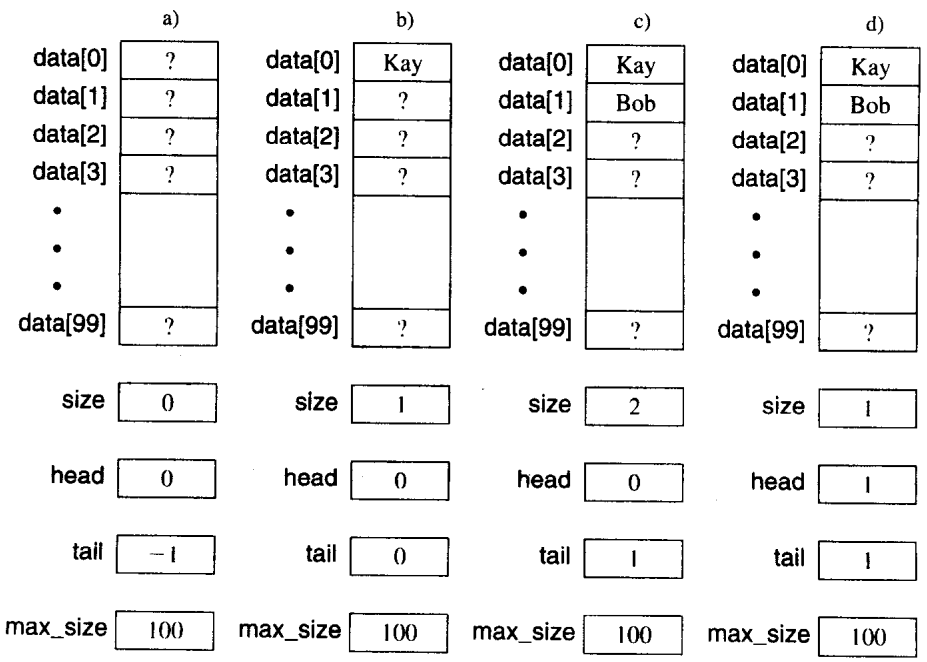


图7-2 队列的一个基于数组的基础容器：a）初始化，b）调用push_back("Kay")之后，c）调用push_back("Bob")之后，d）调用pop_front()之后

图7-2有几个吸引人的特点。首先，注意在每次push_back调用中都要增加tail。在pop_front调用中，项并没有做物理的（真正的）移动。在图7-2d里，“Kay”仍然处在data[0]。所有pop_front所做的就是增加head，表示队列开头的项在次高下标上了。但是如果在每次push_back调用中增加tail，并且在每次pop_front调用中增加head，最终将得到图7-3所示的四个项的队列。

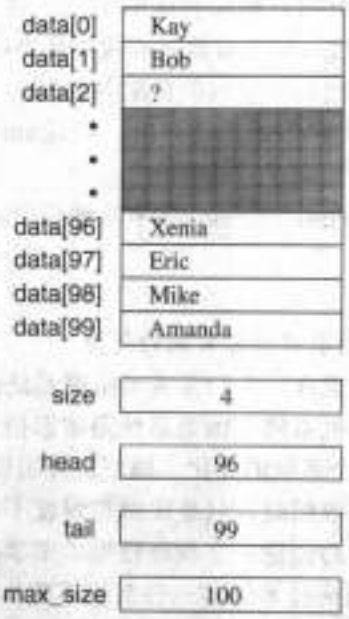


图7-3 队列中下标96~99之间的4项

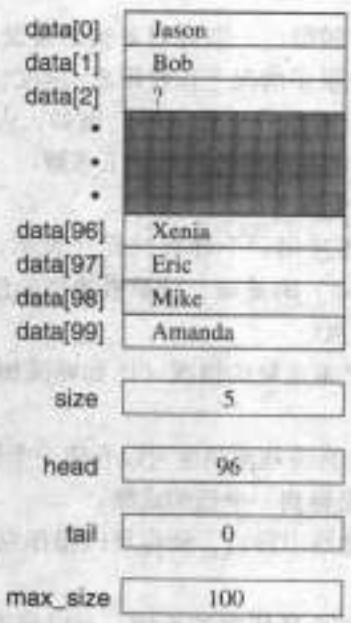


图7-4 容器对象queueArray中有5项存在数组data里：前4项在下标96~99，最后一项在下标0

如果现在使用push_back方法将“Jason”添加进队列会怎样呢？容器对象queueArray的大

小恰好是4，因此不需要增加数组data的长度。可以将四个项移动至下标0到下标3，然后把“Jason”插入到下标4处。但是应当尽可能避免移动项。有没有留心另一种选择？下标0的位置是可用的，所以就把“Jason”放在那里。图7-4显示了结果。

图7-4所示的容器对象queueArray是**环形的**：数组data中下标0位置的项跟随着下标99位置的项。这种安排需要习惯一下，因为有可能得到tail<head的结果。不过它的美妙之处在于，这样的push_back调用只花费常数时间。使用相同的观点，并允许head从0前进到99，这样
260 pop_front调用也只花费常数时间。惟一的非常数时间操作是当size=max_size且调用push_back时数组的扩充。在这种情况下要创建两倍空间的数组并将旧的项拷贝到新的数组里。这花费线性时间，但只是每n次push_back调用才发生一次，因此amortizedTime(n)是常数，所以averageTime(n)是常数。

实现细节留到编程项目7.4。现在看一个包含多个队列应用的领域——计算机仿真。

7.2 计算机仿真

模型——*即系统的简化，使我们能够研究系统的行为。*

系统是若干互相作用的部件的集合。人们常常对研究系统的行为感兴趣，例如，经济系统、政治系统、生态系统甚至计算机系统。因为系统通常都很复杂，所以可能需要使用模型以便于管理任务。**模型**，也就是一个系统的简化，模型设计的目的是为了研究系统的行为。

一个**物理模型**和它代表的系统很相似，只是比例和强度不同。例如，可能为切萨皮克海
261 湾中的潮汐运动或计划中的购物中心创建一个物理模型。军事演习、春季训练和混战也是物
262 理模型的例子。不幸的是，使用当前技术仍有一些系统不能作出物理模型——至今还有一些物理本质不能预期的行为，像天气。通常，就像在飞行训练中，物理模型可能太昂贵、太危险，或者是很麻烦的。

有时系统可以表示成一个**数学模型**：一组针对系统的假设、变量、常量和等式。数学模型当然要比物理模型易于处理。在很多情况下，像距离=速度*时间和勾股定理，在合理的时间内就能够分析求解等式。但是有时候并不能这样。例如，大部分微分等式不能分析求解，以及有数千个等式的经济学模型不能在合理时间内手工求解。

计算机模型使得能有效地模拟复杂系统。

在这样的情况下，数学模型通常用一个计算机程序表示。计算机模型是研究复杂系统（像天气预报、太空飞行和城市规划）的基础。计算机模型的使用称作“计算机仿真”。使用计算机模型比使用原系统有几个优点：

1）*安全*。飞行仿真器可以用严重危险的情况（比如飓风和劫机等）训练飞行学员，但没有人会受伤[⊖]。

2）*经济*。营业方针课程中的仿真游戏使学生可以管理一个假想的公司，与其他学生竞争。如果公司“破产”，惟一的代价就是得到一个低的成绩。

3）*高速*。计算机通常能很快地给出预言，使你可以操作它们。这个特点几乎是每个仿真（从股票市场到国防）的基础。

4）*灵活*。如果得到的结果不符合所研究的系统，可以修改这个模型。这是**反馈**的例子：

⊖ 只有一次，有一个学员在一个大风雪条件下引擎失灵的仿真中，他惊慌失措，从仿真驾驶座舱中“跳伞”出来，撞上真实的地面而扭伤了脚踝。

各种因素产生结果的过程自身也正受着结果的影响。参阅图7-5。

这些好处是很引人注目的，因此计算机仿真成为研究复杂系统的公认的方法。这并不是说计算机仿真是所有系统问题的万能药。模型化一个系统需要简化，因此有必要提出模型和系统的区别。例如，假设30年前开发了一个地球生态系统的计算机仿真，你可能忽略了CFC（氯氟碳）的影响，而实际上目前所有的环境科学家都相信CFC正严重地破坏着臭氧层。 263

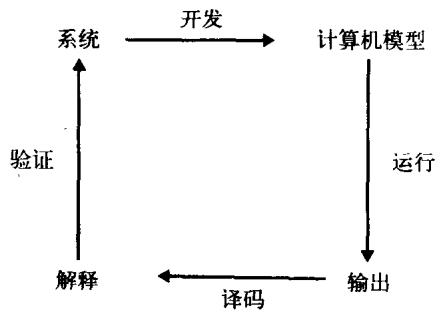


图7-5 计算机仿真的反馈

计算机仿真的缺点是它的结果往往是预言式的解释，而预言总是一件冒险的交易。因此，在计算机仿真结果的前面总是有如下的声明："如果变量之间的关系与初始条件和描述一致，那么结果将可能如下……"

7.3 队列应用：洗车仿真

队列可以用于多种仿真。例如，下面将阐述在Speedo洗车处的交通流动仿真中队列的使用。

问题

给出洗车处的到达时间，计算每辆车的平均等待时间。

分析

假设洗车处有一个工作站；即一个"服务器"。每个车需洗10分钟，队列中每次至多有五辆车等待洗车。当正在清洗一辆车并且队列中有五辆车时，如果有一辆车到达，那么它将作为"溢出"不准入内且不计算在内。

平均等待时间是将每辆车的等待时间加起来再除以车的数量。结束标记是999。下面是关于到达和离开的详细情况：

1) 如果在同一分钟中既有到达的又有离开的，就先处理离开的车。

2) 如果当队列为空且没有车被清洗时，到达了一辆车，那么就马上开始清洗这辆车；它没有进入队列。

3) 每当一辆车开始通过十分钟的清洗周期，它就离开队列并停止等待。 264

系统测试1（输入用黑体表示）

```
Please enter the next arrival time. The sentinel is 999.
1
Please enter the next arrival time. The sentinel is 999.
3
Please enter the next arrival time. The sentinel is 999.
```

5
Please enter the next arrival time. The sentinel is 999
8
Please enter the next arrival time. The sentinel is 999
8
Please enter the next arrival time. The sentinel is 999
15
departure time = 11
Please enter the next arrival time. The sentinel is 999
999
departure time = 21
departure time = 31
departure time = 41
departure time = 51
departure time = 61
The average waiting time, in minutes, was 19.3333.
Please press the Enter key to close this output window.

系统测试2（输入用黑体表示）

Please enter the next arrival time. The sentinel is 999.
5
Please enter the next arrival time. The sentinel is 999.
5
Please enter the next arrival time. The sentinel is 999.
7
Please enter the next arrival time. The sentinel is 999.
12
Please enter the next arrival time. The sentinel is 999.
12
Please enter the next arrival time. The sentinel is 999.
13
Please enter the next arrival time. The sentinel is 999.
265
14
Overflow
Please enter the next arrival time. The sentinel is 999.
18
departure time = 15
Please enter the next arrival time. The sentinel is 999.
19
overflow
Please enter the next arrival time. The sentinel is 999.
25
departure time = 25
Please enter the next arrival time. The sentinel is 999.
999

```
departure time = 35
departure time = 45
departure time = 55
departure time = 65
departure time = 75
departure time = 85
The average waiting time, in minutes, was 27.875.
Please press the Enter key to close this output window.
```

7.3.1 程序设计

我们主要使用的类是CarWash。仿真是**事件驱动**的，也就是说在处理中，关键是判断下一事件是到达事件还是离开事件。

现在可以指定三个方法，它们的接口如下：

```
//后置条件：洗车处为空。
CarWash();

//后置条件：处理所有的到达和离开事件。
void runSimulation();

//后置条件：输出平均等待时间或一个错误消息。
void printResult();
```

CarWash类的设计中的下一步是选择字段。从解决问题需要的一些变量开始，然后从这些变量中选择字段。等待清洗的车应当按照事件顺序排序，因此需要的变量之一是队列carQueue。carQueue中的每一项是一个Car对象，因此，为了决定Car类应当包含的方法，暂时推迟CarWash类的开发。

266

当一辆车离开队列进入工作站时，就可以用当前时间减去车的到达时间，计算出车的等待时间。因此Car类至少要提供一个getArrivalTime()方法，返回刚才弹出（即出列）的车的到达时间。

现在继续决定CarWash类中需要的变量。正如前一段所指示的，应当有waitingTime和currentTime变量。为了计算平均等待时间，需要变量numberOfCars和sumOfWaitingTimes。如何决定下一个事件是到达事件还是离开事件呢？可以根据变量nextArrivalTime（这是读入的）和 nextDepartureTime来决定。当没有车被清洗时，我们希望处理到达事件，因此这时将nextDepartureTime变量设置成一个非常大的数——比方说10 000。

这时，我们已经有了7个变量。哪些应当是字段呢？一个简单的启发是（根据经验），类的大部分公有方法应当使用类的大多数字段，详情参阅Riel(1996)。无疑，printResult方法只使用了变量sumOfWaitingTimes和numberOfCars，而runSimulation方法使用了全部变量。因此对字段的决定可归结为：哪些变量必须在构造器中初始化？

因为carQueue是一个对象，在它定义时将自动初始化，所以它不需要是一个字段。必须被初始化的非对象变量是sumOfWaitingTimes、numberOfCars、currentTime和nextDepartureTime；这些将是字段。不需要初始化waitingTime（currentTime和getArrival Time()返回值之间的差）

和nextArrivalTime（它的值是读入的）。

下面是目前所得到的CarWash.h:

```
#ifndef CAR_WASH
#define CAR_WASH

#include <string>
#include <queue>
#include <iostream>

#include "car.h"

using namespace std;

class CarWash
{
    public:
        // 后置条件: 初始化这个CarWash。
        CarWash( );

        // 后置条件: 处理这个CarWash全部的到达和
        //           离开事件。
267     void runSimulation( );

        // 后置条件: 输出平均等待时间或一个
        //           错误消息。
        void printResult( );

    protected:

        const static int INFINITY; // 指示没有车正在清洗

        const static int MAX_SIZE; // carQueue中允许的车的
                                   // 最大数量

        const static int WASH_TIME; // 每辆车清洗的分钟数

        int currentTime,
            nextDepartureTime,
            numberOfCars,
            sumOfWaitingTimes;
}; // 类CarWash
#endif
```

因为CarWash类中没有为对象的字段，所以也没有依赖关系图。

7.3.2 CarWash类的实现

我们已经完成了（至少目前是这样）字段的选择，因此可以开始方法的定义了。构造器的定义是直截了当的:

```
CarWash::CarWash( )
{
```

```
        currentTime = 0;
        numberOfCars = 0;
        sumOfWaitingTimes = 0;
        nextDepartureTime = INFINITY;
    } // 缺省构造器
```

在继续往下进行之前，先给出常量定义：

```
const int CarWash::INFINITY = 10000;
const int CarWash::MAX_SIZE = 5;
const int CarWash::WASH_TIME = 10;
```

runSimulation方法的定义说明了这是一个基于事件的仿真。对读入的每个nextArrivalTime值，如果这个时间小于nextDepartureTime值，就处理一个到达时间并读入另一个nextArrivalTime值。否则，就处理一个离开事件。当到达结束标记时，需要洗清洗站中的以及仍然在队列中的所有的车。

方法定义是相当简单的，因为推迟了到达和离开事件的处理：

268

```
void CarWash::runSimulation( )
{
    const string PROMPT =
        "\nPlease enter the next arrival time. The sentinel is ";
    const int SENTINEL = 999;
    queue <Car> carQueue;
    int nextArrivalTime;
    cout << PROMPT << SENTINEL << endl;
    cin >> nextArrivalTime;
    while (nextArrivalTime != SENTINEL)
    {
        if (nextArrivalTime < nextDepartureTime)
        {
            processArrival (nextArrivalTime, carQueue);
            cout << PROMPT << SENTINEL << endl;
            cin >> nextArrivalTime;
        } // if
        else
            processDeparture (carQueue);
    } // 当没有达到SENTINEL时
    // 清洗carQueue中余下的车。
    while (nextDepartureTime < INFINITY)
        processDeparture (carQueue);
} // runSimulation
```

下面是方法processArrival和processDeparture（它们是**protected**方法）的接口：

```
//后置条件: 对nextArrivalTime时刻到达的车，要么不准进入（如果发送这个消息之前
//          carQueue已满），要么就让它进入这个CarWash。
```

```
void processArrival (int nextArrivalTime,queue<Car>& carQueue);

//后置条件: 一辆车结束清洗并且当carQueue非空时弹出这辆车。
void processDeparture(queue<Car>& carQueue);
```

为了处理一个到达事件，首先更新currentTime字段并检测溢出。如果这个到达事件没有溢出，就增加numberOfCars字段，而且这辆到达的车要么开始清洗（服务器为空时），要么插入carQueue对象的队列。下面是代码:

```
void processArrival (int nextArrivalTime, queue <Car>& carQueue)
269 {
    const string OVERFLOW = "Overflow";
    currentTime = nextArrivalTime;
    if (carQueue.size( ) == MAX_SIZE)
        cout << OVERFLOW << endl;
    else
    {
        numberOfCars++;
        if (nextDepartureTime == INFINITY)
            nextDepartureTime = currentTime + WASH_TIME;
        else
            carQueue.push (Car (nextArrivalTime));
    } // 不是溢出
} // 方法processArrival
```

这个方法表露了是如何和Car类建立关系的: 有一个构造器使用nextArrivalTime作为变元。下面是Car类的头文件和源文件:

```
// Car.h
#ifndef CAR
#define CAR
class Car
{
    public:
        // 后置条件: 这个Car被初始化。
        Car ( );
        // 后置条件: 用nextArrivalTime初始化这个Car。
        Car (int nextArrivalTime);

        // 后置条件: 返回这个Car的到达时间。
        int getArrivalTime( );

    protected:
        int arrivalTime;
}; // 类Car
#endif

// Car.cpp
```

```
#include "car.h"
Car::Car( ) { }
Car::Car (int nextArrivalTime)
{
    arrivalTime = nextArrivalTime;
} // 构造器

int Car::getArrivalTime( )
{
    return arrivalTime;
} // 方法getArrivalTime
```

270

在这个项目中，很容易就可以废弃Car类，但在该项目的后续扩展中，将需要更多关于车辆的信息——周长，是否可改变，车轴数量等等。

为了处理一个离开事件，首先更新currentTime字段，然后检查在carQueue对象中是否有车。如果有就取出（出列）第一个车，计算它的等待时间并加到sumOfWaitingTimes上，然后开始清洗这辆车。否则，就将nextDepartureTime字段设置成一个很大的数字，指示现在没有车正在清洗中。下面是定义：

```
void CarWash::processDeparture (queue <Car >& carQueue)
{
    int waitingTime;

    cout << "departure time = " << nextDepartureTime << endl;
    currentTime = nextDepartureTime;
    if (!carQueue.empty( ))
    {
        Car car = carQueue.front( );
        carQueue.pop( );
        waitingTime = currentTime - car.getArrivalTime( );
        sumOfWaitingTimes += waitingTime;
        nextDepartureTime = currentTime + WASH_TIME;
    } // carQueue 非空
    else
        nextDepartureTime = INFINITY;
} // 方法processDeparture
```

最后也是最简单的CarWash方法是printResult方法：

```
void CarWash::printResult( )
{
    const string NO_CARS_MESSAGE = "There were no cars in the car
            wash.";
    const string AVERAGE_WAITING_TIME_MESSAGE =
        "\nThe average waiting time, in minutes, was ";
    if (numberOfCars == 0)
        cout << NO_CARS_MESSAGE << endl;
    else
```

```
        cout << AVERAGE_WAITING_TIME_MESSAGE
             << ((double)sumOfWaitingTimes / numberOfCars)
             << endl;
} // 方法printResult
```

271 main函数只不过是调用CarWash类中的三个公有方法:

```
#include <iostream>
#include <string>

#include "CarWash.h"

using namespace std;
int main( )
{
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    CarWash carWash;

    carWash.runSimulation( );
    carWash.printResult( );
    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );
    cin.get( );

    return 0;
} // main
```

7.3.3 CarWash方法的分析

每个CarWash方法将耗费多少时间？使用缺省的基础类deque，在最坏情况下所有方法花费的都是常数时间，除了push方法，它在平均情况下是常数时间，但是worstTime(n)和n是成线性关系的。即使在最坏情况下，n次连续的插入也只花费线性时间，因此所有需要考虑的只有runSimulation方法中的循环，特别是“读-和-处理”循环以及“处理-剩余-车辆”循环。如果到达了n辆车，“读-和-处理”循环就将执行n次，而“处理-剩余-车辆”循环将执行至多6次：一辆车正在清洗中，五辆车在队列中。可以断定，对runSimulation方法而言，worstTime(n)和n成线性关系。其他所有方法的worstTime(n)是常数。

7.3.4 随机化到达时间

其实没必要读入到达时间。可以由仿真程序产生到达时间，程序应提供包含**平均到达时间**（即全体到达时间的平均值）的输入。为了根据平均到达时间生成到达时间序列，需要了解到达时间的分布。现在定义一个函数计算到达时间分布状况，即著名的**泊松分布**。下面讨论的数学依据超出了本书范围，感兴趣的读者可以参阅数学统计学方面的文献。

令x是到达间隔的任意时间。那么$F(x)$，也就是从现在到下一辆车到达的间隔至少为x分钟
272 的几率，它由下式给出:

$$F(x)=\exp(-x/\text{meanArrivalTime})$$

例如，$F(0)=\exp(0)=1$；也就是说，从现在至少过0分钟才会有下一辆车到达的几率。同样，$F(\text{meanArrivalTime})=\exp(-1)\approx 0.4$。$F(10\ 000*\text{meanArrivalTime})$近似为0。$F$函数的曲线如图7-6所示。

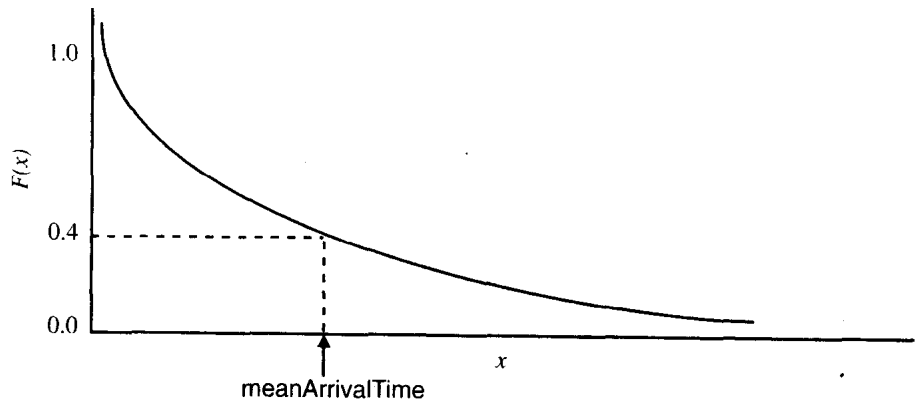


图7-6 到达间隔时间的泊松分布图

为了产生随机到达时间，引入一个整数变量，称作timeTillNext，它包含了从当前时间直到下一辆车到达之间的分钟数。按照如下方式确定timeTillNext的数值。根据分布函数F，经过至少timeTillNext分钟才有下一辆车到达的几率由下式给出：

```
exp(-timeTillNext/meanArrivalTime)
```

这个表达式代表一个几率，明确地讲，就是一个大于0且小于等于1的浮点数。为了随机化这个几率，将表达式和一个相同范围内的随机数值randomDouble相关联。调用rand()函数返回0和RAND_MAX之间的整数（包括0和RAND_MAX），其中RAND_MAX是在实现中定义的变量，它的值至少是32 767。因此设置

```
randomDouble=rand()/double(RAND_MAX+1);
```

类型转换保证了商是**double**类型。那么randomDouble变量包含一个大于等于0.0并小于1.0的**double**型值。因此1-randomDouble将包含大于0.0并小于等于1.0的数值。这正是我们需要的，因此令1-randomDouble等同于exp(-timeTillNext/ meanArrivalTime)。解下面的等式可以求出timeTillNext的值：

273

```
timeTillNext=-meanArrivalTime*log(1-randomDouble);
```

log函数返回其变元的自然对数。最后，将0.5加到这个表达式的右边，以便四舍五入到最接近的整数：

```
timeTillNext=-meanArrivalTime*log(1-randomDouble)+0.5;
```

为了说明数值是如何计算出来的，假设平均到达时间是3分钟，而且1-randomDouble的数值序列从0.71582、0.280151和0.409576开始。那么前三个timeTillNext的数值将是

1，即-3*log(0.71582)+0.5
4，即-3*log(0.280151)+0.5
3，即-3*log(0.409576)+0.5

因此第一辆车将在洗车处开门后1分钟时到达，第二辆车将再过4分钟之后即第5分钟时到

达，第三辆车将再过3分钟即第8分钟时到达。

实验18：随机化到达时间 （所有实验都是可选的）

7.4节介绍了另一个容器配接类——stack类，它主要应用在计算机系统内部。

7.4 堆栈

堆栈是项的有限序列，并满足序列中被删除、检索或修改的项只能是最近插入序列的项。这个项称作是堆栈“顶”的项。

堆栈中的项是反时间顺序存储的：后入，先出。

例如，自助餐厅的盘子架上放了一堆盘子，只能在顶部进行插入和删除。换一种方式讲，最近放上架子的盘子将是下一个被取走的盘子。堆栈的这个定义属性有时简称为“后入，先出”，或LIFO。与这个观察一致，插入也称作“推入”，删除也称作“弹出”。图7-7a显示了三个项的堆栈，图7-7b、c和d显示了两次弹出然后一次推入对堆栈的影响。

7.4.1节考察stack类的方法接口。stack类的接口数甚至比queue类还要少，因为在堆栈中，只有一个位置才能进行插入、删除、检索或修改操作。

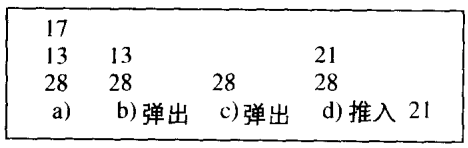


图7-7 进行若干弹出和推入操作后的堆栈：先弹出17和13，然后推入21

7.4.1 Stack类的方法接口

stack类是模板化的，使用deque作为缺省容器类：

```
template<class T, class Container=deque<T>>
class stack
{
```

274

下面是stack类中全部方法的接口。注意它们和queue类的相似性。

1. //后置条件：这个堆栈为空，也就是，它不包含任何项。
   `explicit stack (const Container&=Container());`

2. //后置条件：如果这个堆栈为空就返回真。否则，返回假。
   `bool empty();`

3. //后置条件：返回这个堆栈中项的数量。
   `unsigned size();`

4. //后置条件：x被插入到堆栈的顶部。averageTime(n)是常数。worstTime(n)
   //　　是O(n)，但是对n次连续的推入操作，全部n次推入的worstTime(n)也只是
   //　　O(n)。也就是说，amortizedTime(n)是常数。
   `void push (const T& x);`

5. //前置条件：这个堆栈非空。

```
//后置条件: 返回对这个栈顶项的引用。
T& top();
```

6.
```
//前置条件: 这个堆栈非空。
//后置条件: 返回对这个栈顶项的一个常量引用。
const T& top() const;
```

7.
```
//前置条件: 这个堆栈非空。
//后置条件: 移走本方法调用前位于栈顶的项。
void pop();
```

注意 这个方法并不返回弹出的项。如果想访问弹出的项，应在调用pop()之前先调用top()方法。

7.4.2 使用stack类

迭代器不能在堆栈中使用，因为只有堆栈顶部的项才是可以访问的。这并不代表不能够输出堆栈。但是输出操作不能使用任何除方法接口之外的堆栈细节信息。在7.1.2节中试图输出queue容器内容时也有相似的情况。这里有一个小程序可以产生图7-7所示的堆栈: 275

```
#include <iostream>
#include <vector>
#include <string>
#include <stack>

using namespace std;

void printStack (stack< int, vector<int> > ages)
{
    cout << endl << endl << "Here is the current stack:" << endl;
    while (!ages.empty( ))
    {
        cout << ages.top( ) << endl;
        ages.pop( );
    } // while
} // 函数printStack

int main( )
{
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";
    stack<int, vector< int> > ages;
    ages.push (28);
    ages.push (13);
    ages.push (17);
    printStack (ages);

    ages.pop ( );
    printStack (ages);

    ages.pop( );
```

```
    printStack (ages);
    ages.push (21);
    printStack (ages);
    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );
    return 0;
} // main
```

7.4.3 stack类是一个容器配接器

与queue类一样，stack类也是一个容器配接器：也就是说，stack类的方法是根据一些基础容器类的方法定义的。配接的容器类必须包含下列方法：empty，size，push_back，pop_back
276 和back。注意这暗示着：*堆栈的顶部是容器类的尾部！*

对向量、链表和双端队列而言，在平均情况下，push_back和pop_back只花费常数时间，而且amortizedTime(n)也是常数。因此不管是vector类、deque类或是list类，都能配接stack类，其中缺省的是双端队列类。

对标准模板库中所有的容器配接器而言，它的设计只有一个字段：

```
Container c;
```

而且正如queue类的实现一样，所有的stack方法定义都是单行的，其中c调用了相应的方法。例如，下面是stack类的两个方法定义：

```
void pop( )
{
    c.pop_back( );
} //  方法  pop

T& top
{
    return c.back( );
} //  方法  top
```

7.1.4节中曾考虑为queue类配接一个用户开发的容器。queueArray类满足所要配接的容器的需求，它有一个很吸引人的特点：它是一个环形的数组，只要队列中项的数量不大于数组的大小，就不需要调整它的大小。可以开发一个stackArray类，使它满足配接stack类的容器的需求。但是这个类没有可取之处：它只是简陋地模仿vector类。因为任何C++编译器都需要提供一个vector类，所以这可以绕过stackArray类的开发。

现在集中精力看几个重要的应用。

7.5 堆栈应用1：递归是如何实现的

在第4章中已经看到递归方法的几个范例。依照抽象原理，我们只关注递归做什么而忽略了递归是如何通过编译器或解释器实现的问题。这说明可视化帮助——运行结构框架——和这个实现是紧密相关的。现在略述一下堆栈是如何应用在递归实现中的，以及在这个应用中函

数（特别是递归函数）的时间空间含义。

每当发生一个函数调用，不论它是否是递归的都要保存调用函数的返回地址。保存这个 277
信息是因为这样计算机将清楚函数执行结束后，应当在哪里恢复调用函数的运行。同理，还必须保存大量有关函数局部变量的信息。这是为了防止当函数直接或间接递归时信息被破坏。在4.9节中提到过，编译器不止为递归，而是为所有的函数保存这个信息。这个信息统称为活动记录或堆栈帧。

每个活动记录包括:

1) 一个包含调用函数返回地址的变量。

2) 对每个引用形参，有一个包含相应变元地址的变量。

3) 对每个值形参，有一个变量，它开始时包含相应变元值的拷贝。

4) 在函数块中声明的每个变量。

可以用一个运行时堆栈实现递归。

主存的一部分——**堆栈**——被分配为运行时堆栈，当函数被调用时将一个活动记录推入其中，并且当函数执行结束时再弹出一个活动记录。在函数执行过程中，顶部的活动记录包含了函数的当前状态。

举一个简单的例子，跟踪一个包括第4章的writeBinary函数的小程序的运行。返回地址被注释成RA1和RA2。

```
#include <iostream>
#include <string>
int main( )
{
    const string PROMPT = "Please enter a nonnegative integer";

    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    int n;
    cout << PROMPT;
    cin >> n;
    if (n < 0)
        cout << "Error: You entered a negative integer.";
    else
        writeBinary (n); // RA1

    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );

    return 0;
} // 函数main                                                    278

// 前置条件: n是十进制表示法中的一个非负整数。
// 后置条件: 输出n的二进制表示。worstTime(n)
//              是O(log n)。
void writeBinary (int n)
{
    if (n == 0 || n == 1)
        cout << n;
```

```
    else
    {
        writeBinary (n / 2); // RA2
        cout (n % 2);
    } // else
} // writeBinary
```

writeBinary函数用值形参n作为它惟一的局部变量，因此每个活动记录将包含两个字段：

1) 一个字段表示返回地址。

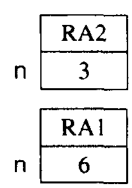
2) 一个字段表示值形参n的值。

假设读入整数6。当从main函数中调用writeBinary时，将创建一个活动记录并将其推入堆栈，如图7-8所示。因为n>1，所以用3（即6/2）作为实参值递归调用writeBinary。于是创建第二个活动记录并将其推入堆栈，如图7-9所示。



活动堆栈

图7-8 第一次激活writeBinary方法之前的活动堆栈。RA1是返回地址

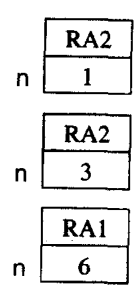


活动堆栈
（两个记录）

279

图7-9 第二次激活writeBinary方法之前的活动堆栈

因为n的值仍然大于1，所以再次调用writeBinary，这次使用1（即3/2）作为实参值。然后创建第三个活动记录并将其推入，如图7-10所示。



活动堆栈
（三个记录）

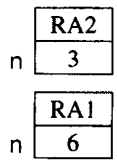
图7-10 第三次激活writeBinary方法之前的活动堆栈

因为n=1，所以输出n的值。输出是

1

这完成了writeBinary方法的第三次激活，因此堆栈弹出并返回到地址RA2，堆栈如图7-11所示。执行writeBinary方法中RA2地址处的输出语句，而且输出是3%2，即，

1



活动堆栈

（两个记录）

图7-11 完成writeBinary方法的第三次激活之后的活动堆栈

堆栈再次被弹出并再一次返回到RA2，如图7-12所示。RA2地址处的输出语句的输出是6%2的值，即，

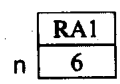
0

280

这就完成了writeBinary方法的最初的激活。堆栈再一次被弹出，变成空，并返回到RA1——在main函数的尾部。完整的输出是：

110

它是输入值6的二进制等值形式。



活动堆栈

图7-12 完成writeBinary方法的第二次激活之后的活动堆栈

对一个引用形参而言，相应变元的地址被推入堆栈。当生成机器代码时，编译器将引用形参的每次出现都看作是指向相应变元的指针。例如，考虑下面的main函数：

```
int main( )
{
      string s = "maybe";
      int i = 3;

      sample (s, i); // RA1
      cout << s << " " << i;

      return 0;
} // 函数main
```

sample函数的定义是：

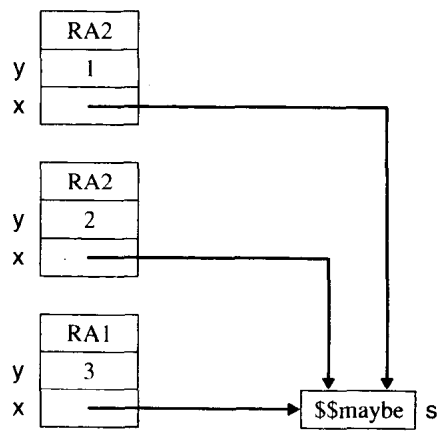
```
void sample (string& x, int y)
{
      x.insert (0, "$");
```

```
        y--;
        if (y > 0)
            sample (x, y); // RA2
    } // sample
```

它生成的代码与下面函数体生成的代码是一致的:

```
x->insert (0, "$");
y--;
if (y > 0)
281     sample (*x, y);
```

在开始sample的第二次递归调用时，堆栈到达了最大高度3:



这样，另一个“$”被增加到x，并减少y。因为y不再大于0，因此不再进行递归调用。三次弹出之后，输出是

```
$$$maybe 3
```

变元s受sample函数调用的影响，因为相应形参x是一个引用参数。另一方面，变元i不受sample函数调用的影响，因为对应i的形参，即y，它是一个值参。

这个讨论应当已说明了编译器是如何实现递归的大体观点。有一个活动记录堆栈处理所有的函数调用，因此每个记录必须包含一个保存记录大小的字段。那样就可以弹出正确数量的字节。为了简化，忽略讨论中的细节。另一个技术性问题涉及到如何操作一个（非void）函数返回的数值，细节问题随编译器不同而不同。一个公用技术是在执行返回前将数值推入栈顶。然后调用函数的工作就是采取适当的行为，它至少将包含堆栈的弹出操作。

编译器必须为运行时的活动堆栈的创建和维护产生代码。每次进行一个调用——甚至是一个非递归调用，必须保存完整的局部环境。这可能效率很低，不管是在时间方面还是在空间方面，例如，把一个庞大的容器对象传递给值形参时。在这种情况下，无论何时进行调用都将把整个容器的拷贝推入堆栈。

如果读者已经设计了递归函数，应当评估一下程序中递归的时间－空间开销的潜在影响。
282 如果发现开销过大，就需要将递归函数转换成迭代函数。这总是可以实现的。

如果提出迭代函数很困难，可以用迭代函数模拟递归函数，它创建并维护自身的堆栈，其中保存了需要的信息。例如，编程项目7.3需要第4章的回溯应用中的（递归）tryToSolve方

法的迭代版本。当创建自己的堆栈时需要决定每个项的组成。例如，如果函数的递归版本包含单个递归调用，那就不需要保存返回地址。下面是writeBinary函数的迭代，基于堆栈的版本（writeBinary函数的不基于堆栈的迭代版本参阅习题4.2）。

```
// 前置条件: n是十进制表示法中的一个非负整数。
// 后置条件: 输出n的二进制表示。worstTime(n)
//              是O(logn)。
void writeBinary (int n)
{
      stack<int> myStack;

      myStack.push (n);
      while (n > 1)
      {
            n = n/2;
            myStack.push (n);
      } // 推入
      while (!myStack.empty( ))
      {

            n = myStack.top( );
            myStack.pop( );
            cout << (n % 2);
      } // 弹出
      cout << endl << endl;

} // 方法writeBinary
```

不要忽略了将递归函数转换成迭代函数所付出的时间代价。一些递归函数，像factorial和Fibonacci函数，很容易就可以转换成迭代函数。而有些函数的转换就没那么简单了，像第4章的move、tryToSolve和permute函数。而且，迭代版本可能缺乏递归版本的简洁优美，这使得它的验证和维护很复杂。

如同第4章中提到的，如果已经准备好一个迭代函数，并且它的效率还可以接受的话，那么就使用它。如果不行，那么在环境允许的情况下应当考虑递归函数。也就是说，只要问题是如下的情况就可以尝试递归：问题的复杂实例可以简化成与原问题形式相同的简单实例，并且最简单的实例可以直接解决。有关活动堆栈的这个讨论有助于做出全面、平衡的决定。 283

7.6节描述了另一个和编译器相关的应用：将表达式转换成机器代码。转换的一个重要方面是确保括号的匹配。作为准备，这里给出一小段程序测试括号是否匹配。输入一串左括号和右括号，然后输出指示括号是否匹配。例如，下面的字符串由匹配的括号组成：

```
(() ())
() ((()))
```

而下面的两个字符串包含了不匹配的括号：

```
()) (
((()))
```

基本策略是：当遇到一个‘(’时，将它推入堆栈；当遇到‘)’时，就弹出堆栈，除非它已空。当到达输入字符串尾部时，如果堆栈为空就说明括号是匹配的。

```
#include <vector>
#include <stack>
#include <iostream>
#include <string>

using namespace std;

int main( )
{
    const string PROMPT = "Please enter a string of parentheses: ";

    const char LEFT = '(';

    const char RIGHT = ')';

    const string SUCCESS = "The parentheses are matching.";

    const string FAILURE = "The parentheses are NOT matching.";

    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    stack< char, vector<char> > parenStack;

    string parens;

    bool matching = true;

    cout << PROMPT;
    cin >> parens;
    for (unsigned i = 0; i < parens.length( ) && matching; i++)
        if (parens [i] == LEFT)
284             parenStack.push (LEFT);
        else if (parens [i] == RIGHT)
            if (parenStack.empty( ))
                matching = false;
            else
                parenStack.pop( );

    if (matching && parenStack.empty( ))
        cout << endl << SUCCESS << endl;
    else
        cout << endl << FAILURE << endl;
    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );

    return 0;
} // main
```

注意，如果堆栈已空而又遇到一个右括号时就退出循环。忽略除左、右括号之外的所有字符。这个程序可以在本书网站的源代码链接中找到。

7.6 堆栈应用2：将中缀转换成后缀

7.5节中说明了一个编译器或解释器是如何实现递归的。本节将介绍另一个“内部”应用：

算术表达式从中缀到后缀表示法的转化。这是编译器在创建机器层次的代码时或者解释器求算术表达式值时的关键任务。

中缀表示法中，二元运算符放在它的操作数之间。例如，图7-13显示了中缀表示法中的几个算术表达式。为了简化起见，一开始先将注意力放在单字母标识符、圆括号以及二元运算符+、-、*和/上。

```
a + b
b - c * d
(b - c) * d
a - c - h / b * c
a - (c - h) / (b * c)
```

图7-13 采用中缀表示法的几个算术表达式

算术的常用规则是:

1) 运算通常都是从左向右执行的。例如，如果有

a + b - c

那么将先执行加法。

285

2) 如果当前运算符是+或-，并且下一个运算符是*或/，那么将在当前运算符之前先应用下一个运算符。例如，如果有

b + c*d

那么在加法之前先运行乘法。对

a - b + c*d

应先执行减法，然后是乘法，最后是加法。可以将这个规则解释为乘法和除法比加法和减法有“更高的优先级”。

3) 可以用括号改变规则1和规则2所指定的顺序。例如，如果有

a - (b + c)

那么先执行加法。同理，对

(a - b)*c

就先执行减法。

图7-14显示了图7-13中最后两个表达式的求值顺序。

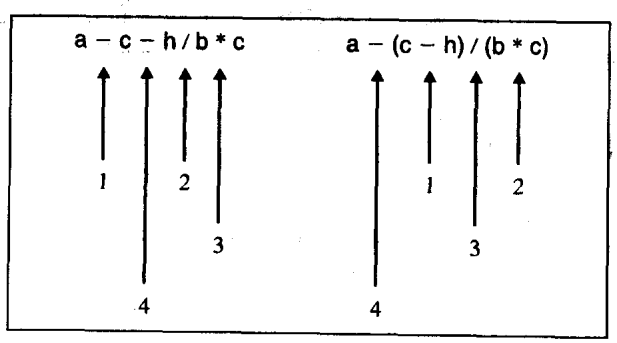


图7-14 图7-13中最后两个表达式的求值顺序

第一个广泛使用的编程语言是FORTRAN（FORmula TRANslator），如此命名是因为它的编译器可以将算术公式转化成机器层的代码。在早期（1960年之前）的编译器中，这个转化是直接进行的。但是直接转化是很难使用的，因为在了解一个运算符的两个操作数之前，不能生成它的机器层的代码。这个要求使得当每个操作数都是一个括号括起来的子表达式时，非常难于处理。

7.6.1 后缀表示法

在后缀表示法中，运算符紧随在它的操作数的后面。

现代编译器不会直接将算术表达式转化成机器层的代码。而是使用一个中间形式，称为**后缀表示法**。在后缀表示法中，运算符紧随在它的操作数的后面。例如，给出中缀表达式a + b，
286 它的后缀形式就是ab+。对a + b*c，后缀形式是abc*+，因为+的操作数是a以及b和c的乘积。对

(a+b)*c

其后缀形式是

ab+c*

因为在后缀表示法中运算符就紧随着它的操作数，不需要括号，因而没有使用括号。图7-15显示了几个算术表达式的中缀和后缀表示法。

中缀	后缀
a − b + c * d	ab − cd *+
a + c − h / b * r	ac + hb / r * −
a + (c − h) / (b * r)	ach − br */+

图7-15 采用中缀和后缀表示法的几个算术表达式

怎么样才能将中缀表示法的算术表达式转化成后缀表示法呢？先将中缀表示法看作一个字符串，然后再试着提出相应的后缀字符串。后缀字符串中的标识符将和中缀字符串中的一致，因此，一旦遇到一个标识符，就可以将它添加进后缀字符串。但是在后缀表示法中，运算符必须放在它们的操作数之后。因此，在中缀字符串中遇到一个运算符时，必须先临时地保存它。

例如，假设要将中缀字符串

a - b + c*d

转化成后缀字符串。（空格只是为了便于阅读，现在还不需要将它们考虑成中缀表达式的一部分。）采用下面的步骤：

'a' 被添加进后缀，后缀现在是 "a"

'-' 被暂时保存

'b' 被添加进后缀，后缀现在是 "ab"

当遇到 '+' 时，注意因为它的优先级和 '-' 相同，所以根据自左向右规则（算术规则1），应当先执行减法。因此将 '-' 加入后缀字符串并暂时保存 '+'，后缀字符串现在是 "ab-"。然后将 'c' 加入后缀，后缀变成 "ab-c"。 287

乘法比加法的优先级别高，因此 '*' 应当在 '+' 之前加入后缀。但是也必须先暂时保存 '*'，因为它的一个操作数（即 'd'）还没有加入后缀。

当 'd' 加入后缀时，后缀字符串变成 "ab-cd"。那么加入 '*'，后缀字符串变成 "ab-cd*"。最后加入 '+'，最终的后缀表示形式是 "ab-cd*+"。

临时存储工具用堆栈是很方便的。管理这个operatorStack对象的规则是：

1) 初始状态下，operatorStack为空。

2) 对中缀字符串中的每个运算符，执行循环，直到运算符被推入operatorStack对象：如果operatorStack对象为空或者中缀运算符的优先级比operatorStack对象顶部的运算符高，那么将运算符推入operatorStack对象；否则，弹出operatorStack对象，并将弹出的运算符加入后缀字符串。

3) 一旦到达输入字符串的尾部，则执行循环，直到operatorStack对象为空：弹出operatorStack对象并将弹出的运算符添加进后缀字符串。

这些规则的基本事实可以总结如下：

中缀优先级高，推入

例如，图7-16显示了将表达式

a + c - h/b*r

中缀表达式: a + c - h / b * r		
中缀	*operatorStack*	后缀
a	(空)	a
+	+	a
c	+	ac
-	-	ac+
h	-	ac+h
/	-/	ac+h
b	-/	ac+hb
*	-*	ac+hb/
r	-*	ac+hb/r
	-	ac+hb/r*
	(空)	ac+hb/r*-

图7-16 将a + c - h/b*r转化成后缀表示。在每一步，operatorStack对象的栈顶就是它最右边的项

转化成它的后缀形式的过程中，运算符堆栈的历史记录。怎样处理括号呢？在中缀字符串中遇到一个左括号时，立即将它推入operatorStack对象，但是把它的优先级定义得比任何二元运算符都低。在中缀字符串中遇到右括号时，operatorStack对象就不断地弹出，并将弹出的项加入后缀字符串，直到栈顶的“运算符”是一个左括号。然后弹出左括号，但不将它加入后缀，并重新扫描中缀字符串。

例如，将a*(b + c)转化成后缀时，项*、(和+将被推入，然后当遇到右括号时将弹出‘+’和‘(’(后入，先出)。最后的后缀形式是

a b c + *

288 图7-17展示了一个更复杂的例子，将x − (y*a/b − (z + d*e) + c)/f转化成后缀形式。

中缀表达式:	x − (y * a / b − (z + d * e) + c) / f
中缀	*operatorStack* 后缀
x	(空) x
−	− x
(	−(x
y	−(xy
*	−(* xy
a	−(* xya
/	−(/ xya*
b	−(/ xya*b
−	−(− xya*b/
(	−(−(xya*b/
z	−(−(xya*b/z
+	−(−(+ xya*b/z
d	−(−(+ xya*b/zd
*	−(−(+* xya*b/zd
e	−(−(+* xya*b/zde
)	−(−(+ xya*b/zde*
	−(−(xya*b/zde*+
	−(− xya*b/zde*+
+	−(+ xya*b/zde*+−
c	−(+ xya*b/zde*+−c
)	−(xya*b/zde*+−c+
	− xya*b/zde*+−c+
/	−/ xya*b/zde*+−c+
f	−/ xya*b/zde*+−c+f
	− xya*b/zde*+−c+f/
	(空) xya*b/zde*+−c+f/−
后缀表达式:	x y a * b / z d e * + − c + f / −

图7-17 将x − (y*a/b − (z + d*e) + c)/f由中缀形式转化成后缀形式。每一步中operatorStack对象的栈顶就是它最右边的项

7.6.2 转换矩阵

在转化处理的每一步，只要知道中缀字符串的当前字符和运算符堆栈的栈顶字符，就能确定采取什么样的行为。所以可以创建一个矩阵来概述转化过程。行下标代表当前中缀字符的可能值；列下标代表运算符堆栈的栈顶字符的当前值；矩阵的条目代表即将采取的行为。

这样的矩阵称作**转换矩阵**，因为它指出了从一个形式转换到另一个形式的转换信息。图7-18显示了将一个简单表达式从中缀表示法转化成后缀表示法的转换矩阵。

图7-18中的转换矩阵的图形化表现使我们马上就能看出如何将一个简单的表达式从中缀转化成后缀。现在可以设计和实现程序来完成这个转化。出于可扩展性上的考虑，程序可以与图7-18中的转换矩阵合起来考虑：扩展矩阵以适应更复杂的表达式。 289

采取的行为		运算符堆栈顶字符			
		(	+,−	*,/	(空)
中缀字符	标识符	Append to postfix	Append to postfix	Append to postfix	Append to postfix
	)	Pop; pitch '('	Pop to postfix	Pop to postfix	Error
	(	Push	Push	Push	Push
	+,−	Push	Pop to postfix	Pop to postfix	Push
	*,/	Push	Push	Pop to postfix	Push
	(空)	Error	Pop to postfix	Pop to postfix	Done

图7-18 将简单表达式从中缀表示法转化成后缀表示法的转换矩阵

7.6.3 记号

使用转换矩阵的程序可能不能操作字符本身，因为每个字符都有太多可能的（合法）值。例如，为每个合法的中缀字符使用一行的转换矩阵中，只一个标识符就需要52行。并且如果改变规则，即允许使用多个字符的标识符，那么将需要数百万行！

当一个程序被**记号化**时，它被分成小的有意义的单元。

取而代之的是，合法的字符通常被组合成“记号”。**记号**是程序中最小的有意义的单元。每个记号都有两部分：通用部分和具体部分。通用部分保存它的类别，具体部分使我们能取回记号化的字符。为了将简单的中缀表达式转化成后缀，将记号分类为：identifier、rightPar、 290
leftPar、addOp（表示+和−）、multOp（表示*和/）以及empty（表示虚值）。具体部分将包含中缀字符串中记号化字符的索引。例如，给出中缀字符串

```
a + b*c
```

要记号化‘b’，将设置它的类别为identifier，索引是2。

记号的结构随编译器的不同有很大的分别。一般说来，变量标识符的记号的具体部分包含了在表中的地址，这个表称作是**符号表**。该地址上将存储标识符、变量标识符指示、它的类型、初始值、声明它的块以及对编译器有用的其他信息。

实验19中开发了一个完整的“中缀到后缀”项目，使用了记号和大量的输入编辑。

291

实验19：将中缀转化成后缀 （所有实验都是可选的）

7.6.4 前缀表示法

在前缀表示法中，运算符直接放在操作数的前面。

7.6.1节中描述了如何将一个中缀表达式转化成后缀表示。另一种可能是将中缀转化成前缀表示，前缀表示法[⊖]中运算符直接放在它的操作数的前面。图7-19显示了几个算术表达式的中缀和后缀表示。

Infix	Prefix
a - b	- a b
a - b * c	- a * b c
(a - b) * c	* - a b c
a - b + c * d	+ - a b * c d
a + c - h / b * d	- + a c * / h b d
a + (c - h) / (b * d)	+ a / - c h * b d

图7-19 几个算术表达式的中缀和后缀表示

怎样才能将一个简单的算术表达式从中缀转化成前缀形式呢？就像在中缀到后缀表示的转换中一样，需要保存每个运算符，直到获得它的全部两个操作数。但是不能只是遇到一个标识符，就将它加入前缀字符串。取而代之的是，这里将需要保存每个标识符，实际上是每个操作数，直到获得它的运算符。

操作数和运算符的保存利用两个堆栈对象——operandStack和operatorStack——是很容易实现的。最初两个堆栈都为空。在中缀字符串中遇到一个标识符时，就将它推入operandStack对象。管理operatorStack对象的规则就像中缀到后缀的转化中一样。

operandStack对象怎么样呢？假设刚刚从operatorStack对象中弹出了栈顶运算符——opt，然后也从operandStack对象的栈顶弹出了两个操作数——opnd1和opnd2。连接（结合在一起）opt、opnd2和opnd1，并将结果推入operandStack对象。重要的是：在连接中，opnd2应在opnd1之前，因为opnd2是在opnd1之前推入operandStack对象的。

持续这个过程直到到达中缀表达式的尾部。然后重复下面的过程直到operatorStack为空：

从operatorStack对象中弹出opt。

292 从operandStack对象中弹出opnd1。

从operandStack对象中弹出opnd2。

将opt、opnd2和opnd1连接在一起，然后将结果推入operandStack对象。

当operatorStack对象最终为空时，operatorStack对象顶部的操作数（也是惟一的）将是对

⊖ 前缀表示法是由一个波兰逻辑学家，Jan Lucasiewica创造的。有时会称它为波兰表示法，而后缀又称作反波兰表示法。

应原中缀表达式的前缀字符串。

例如，如果开始的表达式是

a + b*c

那么两个堆栈的历史记录如下:

	infix	operandStack	operatorStack
1.	a	a	
2.	+	a	+
3.	b	b a	+
4.	*	b a	* +
5.	c	c b a	* +
6.		*bc a	+
7.		+a*bc	

相应于原字符串的前缀字符串是

+a*b c

293

举一个更复杂的例子，假设中缀字符串是

a + (c - h)/(b*d)

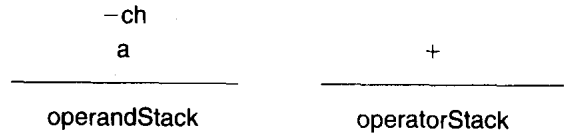
那么在第一个右括号的处理过程中，两个堆栈的项如下:

	infix	operandStack	operatorStack
1.	)	h c a	- (+

2.

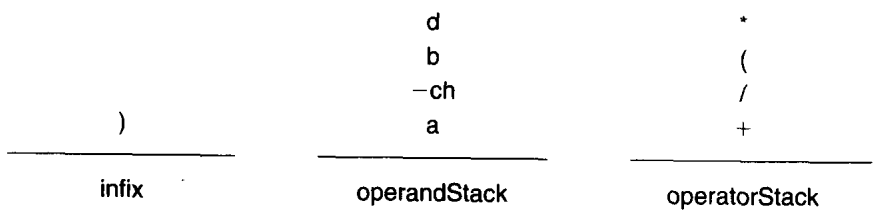


3.

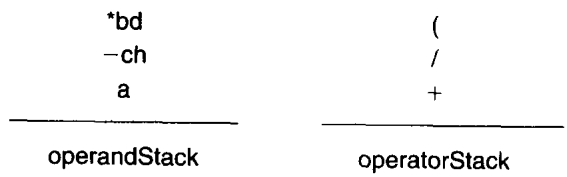


在中缀字符串的第二个右括号的处理过程中，将得到

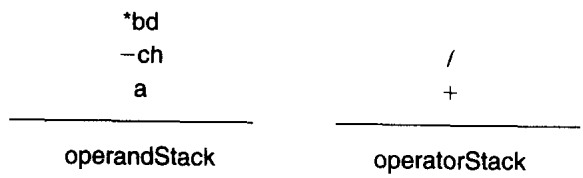
1.



2.

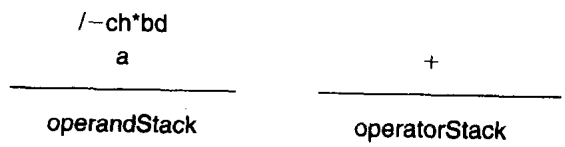


3.

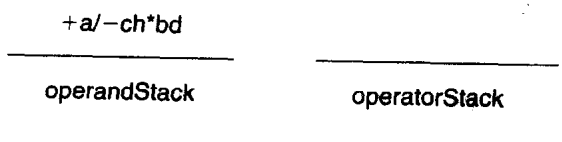


294 已到达了中缀表达式的末尾，因此operatorStack对象重复弹出操作。

4.



5.



前缀字符串是+a/−ch*bd。

总结

队列是项的有限序列，项的检索、删除和修改只能在头部进行，而插入只能在尾部进行。这个公平的*先来先服务*限制使得queue类成为很多系统的一个重要组件。特别是，queue类在研究这些系统行为的计算机模型的开发中扮演了关键的角色。

堆栈是项的有限序列，其中被删除、检索或修改的项只能是最近插入序列的项。该项称作位于堆栈**顶**的项。编译器实现递归正是通过生成创建和维护**活动堆栈**的代码来实现的：活动堆栈是一个运行时堆栈，它保存了每个激活函数的状态。另一个堆栈应用是将中缀表达式转化成机器代码。通过运算符堆栈，可以很容易地把一个中缀表达式转化成后缀表示，也就是中缀表示法和机器语言的中间形式。

习题

7.1 假设定义

```
queue<int> x;
```

说明发送下面每条消息之后队列的情况：

```
a. x.push(2000);
b. x.push(1215);
c. x.push(1035);
d. x.push(2117);
e. x.pop();
f. x.push(1999);
g. x.pop();
```

295

7.2 当x定义如下时，重复做习题7.1：

```
stack<int> x;
```

7.3 扩展实验7的Linked类，使得它可以作为queue容器配接器的基础类。

7.4 习题7.3的扩展Linked类能不能作为stack容器配接器的基础类？解释原因。

7.5 回想前面，“deque”代表“双端队列”，也就是说，一个既允许在头部又允许在尾部进行插入、删除和检索操作（且仅需要常数时间）的队列。双端队列还有什么显著特点？如果在定义deque类之前定义queue类，为什么它会和标准模板库是矛盾的？

7.6 假设依次将a、b、c、d、e推入一个最初为空的堆栈。然后这个堆栈弹出4次，每当一个项从堆栈中弹出，就将其插入一个最初为空的队列。如果接着从队列中删除一项，那么下一个即将被删除的是哪一项？

7.7 queue类能否作为stack类的基础类？请解释原因。

提示 *基础类必须提供什么方法？*

7.8 使用一个活动记录堆栈，在初始调用factorial(4)之后跟踪递归阶乘方法的运行。

7.9 将下列表达式转化成后缀表示：

```
a. x + y*z
b. (x + y)*z
c. x - y - z*(a+b)
d. (a + b)*c - (d + e*f/((g/h + i - j)*k))/r
```

7.10 将习题7.9的表达式分别转化成前缀表示。

7.11 一个后缀表示的表达式可以依靠堆栈在运行时求值。为简化起见，假设后缀表达式只由整数值和二元运算符组成。例如，可能有下面的后缀表达式：

8 5 4 + * 7 -

求值过程如下：当遇到一个数值时就将它推入堆栈。当遇到一个运算符时，取出堆栈的第一个和第二个项，应用运算符（第二个项是左操作数，第一个项是右操作数），并将结果推入堆栈。后缀表达式处理完后，这个表达式的值就是栈顶的项（也是惟一的）。

296

例如，对前面给出的表达式，堆栈的内容变化如下：

			4
		5	5
	8	8	8
9		7	
8	72	72	65

将下述表达式转化成后缀表示，并使用堆栈对表达式求值：

5 + 2*(30 - 10/5)

297 7.12 queue类和stack类都没有定义析构器，缺少这些会不会导致内存泄漏？解释原因。

编程项目7.1：扩展洗车仿真

令Speedo的洗车仿真更接近现实。

分析

根据泊松分布，到达时间应当由平均到达时间随机产生。Speedo增加了一个新的特点：服务时间不必是10分钟，但依赖于顾客的需要，像只清洗、清洗加打蜡或是清洗并吸尘。一辆车的服务时间应当在开始清洗它之前计算出来，也就是从顾客了解将花费多少时间直到他离开洗车处为止。服务时间也是一个泊松分布，应当由平均服务时间随机产生。

平均等待时间和平均队列长度都计算到一个小数位。平均等待时间是等待时间的总和除以顾客的数量。

平均队列长度是仿真中每分钟队列长度的总和除以直到最后一个顾客离开所经过的分钟数。为了计算队列长度的总和，对仿真的每一分钟，累加这一分钟内队列中顾客的数量。也可以用另一种方式计算这个总和：对每个顾客累加他在队列中的分钟数总和。但是这是等待时间的总和！因此可以将平均队列长度计算为等待时间的总和除以直到最后一个顾客离开时仿真经过的分钟总数。而且已经在平均等待时间中计算了等待时间的总和。

同理可以计算溢出的数量。使用500作为随机数生成器的种子。

系统测试1（输入用黑体表示）

Please enter the mean arrival time:**3**
Please enter the mean service time:**5**
298 Please enter the maximum arrival time:**25**

时　间	事　件	等待时间
1	到达	
4	到达	

（续）

时　间	事　件	等待时间
6	到达	
7	离开	0
7	到达	
7	到达	
8	离开	3
11	离开	2
14	离开	4
15	到达	
15	到达	
17	离开	7
20	到达	
24	离开	2
24	到达	
25	离开	9
27	离开	5
36	离开	3

平均等待时间是3.9分钟。

平均队列长度是1.0辆车。

溢出数量是0。

系统测试2（输入用黑体表示）

Please enter the mean arrival time:**8**
Please enter the mean service time:**5**
Please enter the maximum arrival time:**20**

时　间	事　件	等待时间
3	到达	
9	离开	0
10	到达	
12	到达	
13	离开	0
13	到达	
14	离开	1
17	离开	1

平均等待时间是0.5分钟。

平均队列长度是0.1辆车。

溢出数量是0。

299

编程项目7.2：求一个条件的值

开发一个程序求条件的值。

分析

输入将由一个条件（即一个布尔表达式）和后面每行一个的数值组成，数值分别代表条

件中各个变量的值。例如:

```
b*a > a + c
6
2
7
```

变量b得到值6，a得到2，且c得到7。运算符*优先级比>高，而且+的优先级也比>高，因此表达式的值为**true**（12大于9）。

每个变量都以仅由小写字母组成的标识符形式给出。所有的变量必须都是整数值。这里没有直接常量。合法的运算符和优先级从高至低分别是

```
*, /, %
+, -（即整数加法和减法）
>, >=, <=, <
==, ! =
&&
||
```

括号括起来的子表达式是合法的。不需要任何输入编辑。

系统测试1

```
Please enter a condition, or $ to quit.
b * a > a + c
Please enter a value.
6
Please enter a value.
2
Please enter a value.
7
The value of the condition is true.
Please enter a condition, or $ to quit.
b * a < a + c
Please enter a value.
6
Please enter a value.
2
Please enter a value.
7
The value of the condition is false.
Please enter a condition, or $ to quit.
first + last * next == current * (next - previous)
Please enter a value.
6
Please enter a value.
2
Please enter a value.
7
Please enter a value.
5
Please enter a value.
3
The value of the condition is true.
Please enter a condition, or $ to quit.
```

```
first + last * next != current * (next - previous)
Please enter a value.
6
Please enter a value.
2
Please enter a value.
7
Please enter a value.
5
Please enter a value.
3
The value of the condition is false.
Please enter a condition, or $ to quit.
a * (b + c / (d - b) * e) >= a + b + c + d + e
Please enter a value.
6
Please enter a value.
2
Please enter a value.
7
Please enter a value.
5
Please enter a value.
3
The value of the condition is true.
Please enter a condition, or $ to quit.
a * (b + c / (d - b) * e) <= a + b + c + d + e
Please enter a value.
6
Please enter a value.
2
Please enter a value.
7
Please enter a value.
5
Please enter a value.
3
The value of the condition is false.
Please enter a condition, or $ to quit.
$
```

301

系统测试2

```
Please enter a condition, or $ to quit.
b < c && c < a
Please enter a value.
10
Please enter a value.
20
Please enter a value.
30
The value of the condition is true.
Please enter a condition, or $ to quit.
b < c && a < c
Please enter a value.
```

```
10
Please enter a value.
20
Please enter a value.
30
The value of the condition is false.
Please enter a condition, or $ to quit.
b < c || a < c
Please enter a value.
10
Please enter a value.
20
Please enter a value.
30
The value of the condition is true.
Please enter a condition, or $ to quit.
c < b || c > a
Please enter a value.
10
Please enter a value.
20
Please enter a value.
30
The value of the condition is true.
Please enter a condition, or $ to quit.
b != a || b <= c && a >= c
Please enter a value.
10
Please enter a value.
20
Please enter a value.
30
The value of the condition is true.
Please enter a condition, or $ to quit.
(b != a || b <= c) && a >= c
Please enter a value.
10
Please enter a value.
20
Please enter a value.
30
The value of the condition is false.
Please enter a condition, or $ to quit.
a / b * b + a % b == a
Please enter a value.
17
Please enter a value.
5
The value of the condition is true.
Please enter a condition, or $ to quit.
$
```

提示 参阅实验19和习题7.11里中缀到后缀的转化。构造后缀队列之后，创建一个int项的向量values对应标识符向量symbolTable，还要创建int项的堆栈runtimeStack，

推入和弹出**int**和**bool**类型的数值（回想一下，**false**和0是同义的，而**true**和1是同义的）。 303

编程项目7.3：一个迭代的迷宫搜索

重做第4章的迷宫搜索项目，将tryToSolve方法替换成一个模拟递归的迭代方法。

提示 使用堆栈模拟对tryToSolve的递归调用。

项目的原始版本可参阅本书网站上的源代码链接。 304

编程项目7.4：queue类的另一个设计

7.1.4节中描述了queueArray类的实现。包含了一个驱动器来验证类。

提示 惟一的复杂情况出现在调用push_back方法且queueArray占据了全部的data数组，即size=max_size时，或者(tail+1)%max_size=head时也是一样的。然后应当创建一个两倍data大小的数组，并将旧数组拷贝到这个新数组里。首先，拷贝下标head和旧数组尾部之间的项，从下标0开始。如果head=0，那么再没有什么可拷贝的。否则，将旧数组中剩下的项（下标0...tail）拷贝到新的数组，从前一个拷贝结束位置之后开始。

通用型算法copy可以简化工作。回忆一下，通用型算法既能操作数组又能操作容器对象。例如，为了将oldData[head...max_size-1]中的每一项拷贝到data[0...max_size-head]:

```
copy(oldData + head,oldData + max_size,data);
```

对push_back和pop_front方法，使用模算法。例如，将

```
if(head==max_size)
    head=0;
else
    head++;
```

替换成等价的

```
head=(head+1)%max_size;
```

305

第8章　二叉树和折半查找树

本章从第2、5、6、7章中的线性结构中延伸出了“分枝”，介绍一个二维的概念：二叉树。在对二叉树的定义和性质有一定了解之后，将去认识一个特殊类型的二叉树——折半查找树，它的项是遵守某种顺序的。

折半查找树是很吸引人的数据结构，因为它们在平均情况下的插入、删除和查找都只需要对数时间（但是在最坏情况下需要线性时间）花费。这个性能要远胜于数组、向量或列表结构中在平均情况下的插入、删除和查找的线性时间花费。例如，当n=1 000 000时，$\log_2 n$<20。

本章使用了一个不在标准模板库类中的BinSearchTree类来实现折半查找树的数据结构。学习BinSearchTree类的主要原因是它是第9、10章的AVLTree和rb_tree类的简化版本。它们是“平衡”的折半查找树，其高度总是和n成对数关系。这表示对于插入、删除和查找，worstTime(n)和n是成对数关系的。在第11章中探讨了另外两种类型的二叉树——堆和霍夫曼树，在第12章中将学习决策树。本章的素材将有助于读者更好地理解第9~12章中的树。

目标

1) 理解二叉树的概念和重要属性，像二叉树定理和外部路径长度定理。
2) 能够在一个二叉树中进行各种遍历。
3) 比较BinSearchTree类的insert、erase方法和vector、deque以及list类的相应方法的时间效率。
4) 讨论BinSearchTree类的find方法和通用型算法find以及binary_search的相同点和不同点。

8.1　定义和属性

一个递归的定义。

下面的定义确定了整章的基调：

> **二叉树**t要么为空，要么由一项（称作**根项**）和两个不相交的二叉树（称作t的**左子树**和**右子树**）组成。

我们将这些子树分别表示成leftTree(t)和rightTree(t)。采用函数表示法leftTree(t)代替对象表示法t.leftTree()，是因为没有二叉树这个数据结构。为什么没有呢？因为对像插入和删除之类的操作，有众多不同的方法（甚至是不同的参数列表）用于不同类型的二叉树。注意二叉树的定义是递归的，并且大多数和二叉树关联的定义天生就是递归的。

在描绘一个二叉树时，根项习惯上画在顶部。为了说明根项和左、右子树的关系，画一条从根项到左子树的西南向的线以及一条从根项向右子树的东南向的线。图8-1显示了几个二叉树。

图8-1a中的二叉树和图8-1b中的二叉树是不同的，因为B是图8-1a中的左子树，而不是图8-1b的左子树。本书第14章中将会讲到，当把这两个树看作一般的树时，它们是等价的。

二叉树的一个子树自身也是二叉树，因此图8-1a中有7个二叉树：整个的二叉树，以B为根的二叉树，以C为根的二叉树以及四个空的二叉树。试着计算一下图8-1d中的子树的总数量。

植物学术语：根，树枝，树叶。

从根到子树的线称作是**树枝**。一个相关的左子树和右子树均为空的项称作**树叶**。树叶上没有向下连接的树枝。图8-1e所示的二叉树中有四个树叶：15，28，36和68。可以递归求出一个二叉树中树叶的数量。令*t*是一个二叉树，*t*中树叶的数量记为leaves(t)，可以做如下递归定义：

如果*t*为空

leaves(*t*)=0

否则如果*t*只由一个根项组成

leaves(*t*)=1

否则

308 leaves(*t*)=leaves(leftTree(*t*))+leaves(rightTree(*t*))

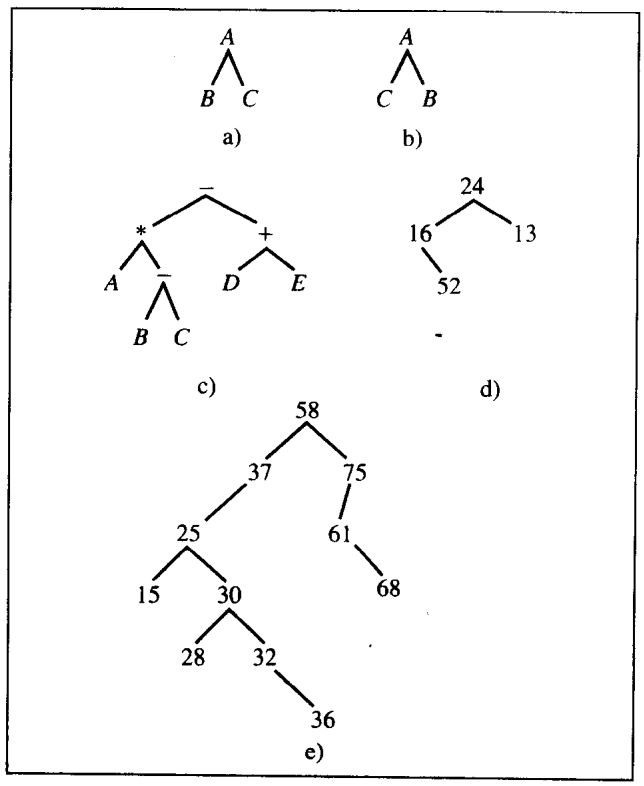


图8-1 几个二叉树

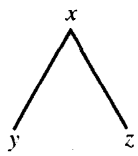
这是一个数学上的定义，而不是C++的方法。定义的最后一行说明*t*中树叶的数量等于*t*的左子树的树叶数量加上*t*的右子树的树叶数量。练习用这个定义计算图8-1a的树叶数量。当然，只要看看整个树就能计算出树叶的数量，不过这个定义是原子的而不是整体的。

二叉树的每一项都可以由它在树中的位置惟一地确定。例如，令*t*是图8-1c中的二叉树，它有两项的值是‘-’。要区分它们，可以称其中一个是“值为‘-’而位置在*t*的根的项”，而

另一个是“值为‘-’而位置在t的左子树的右子树的根的项”。不严格地讲，可以说是二叉树中的“某项”，而严格来讲，要称位于某某位置的“项”。

家族术语：父亲，子女，兄弟。

令t是如下的二叉树：



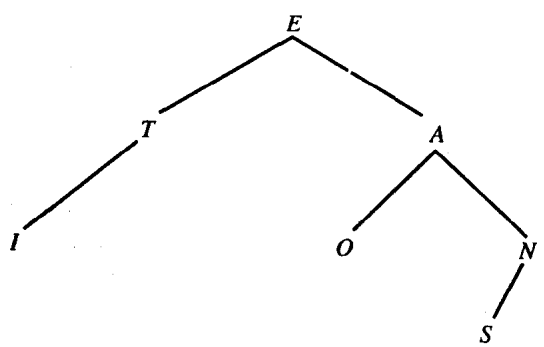
309

那么称x是y的**父亲**而y是x的**左子女**。相似地，称x是z的**父亲**而z是x的右子女。在一个二叉树中，每个项都有0个、1个或2个子女。例如，在图8-1d中，24有两个子女，16和13；16仅有一个子女52；13和52都没有子女，也就是说，它们是树叶。对树中的任意项w，将w的父亲记为parent(w)，w的左子女记为left(w)，w的右子女记为right(w)。

二叉树中的根项没有父亲，而其他的每个项都有一个父亲。继续采用家族术语可以定义兄弟、祖父、孙子、第一代堂兄弟、祖先和后代。例如，如果B是根项为A的子树中的一项，那么项A就是项B的**祖先**。递归地说，如果parent(B)=A或者A是parent(B)的祖先，那么A就是B的祖先。

如果A是B的一个祖先，那么从A到B的**路径**就是从A开始到B的项的序列，序列中的每一项（除了最后一个）都是下一项的父亲。例如，在图8-1e中，序列37，25，30和32是37到32的路径。

非形式化地说，二叉树的高度是根和最远的树叶（即祖先最多的树叶）之间的树枝数量。例如，下面是一个高度为3的二叉树：



这个树的高度为3，因为从E到S的路径上有三个树枝。假设某些二叉树，它的左子树高度为12，而右子树高度为20，那么整个树的高度是多少？答案是21。

空二叉树的高度是-1。

一般来讲，二叉树的高度比左子树和右子树的最大高度大1。这可以给出一个二叉树高度的递归定义。但是首先需要知道基本情况是什么，即空树的高度。我们希望单个项的树的高度是0：没有从根项出发的树枝；也就是左右子树均为空。但是如果0是比空子树高度多1的数，那就需要将一个空子树的高度定义成更奇怪的-1。

令t是一个二叉树，将t的**高度**height(t)定义如下：

如果t为空，

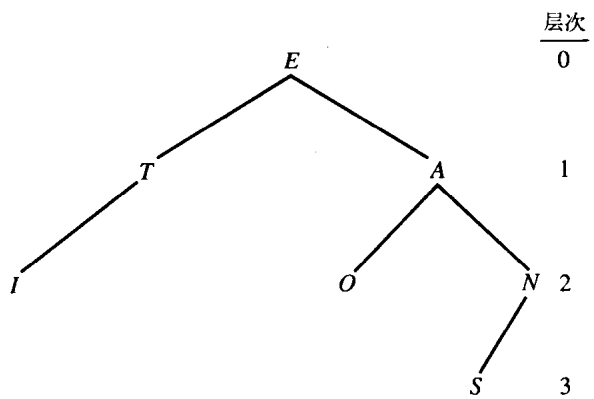
height(t)=−1

否则

310 height(t)=1+maximum of{height(leftTree(t)),height(rightTree(t))}

这个定义表明了只有一个项的二叉树高度为0，因为它的每个空子树的高度是−1。同理，图8-1a中二叉树的高度是1，图8-1e中二叉树的高度是5。

高度是整个二叉树的属性。二叉树的每一项可以定义一个相似的概念：项的层次。非形式化地说：给定项的层次是根项和它之间的树枝数量。例如，下面是一个二叉树，它的层次如图所示：



注意根项的层次是0，而树的高度等于该树中的最高层次。现在给出一个形式化的定义。令t是一个非空二叉树，对t中的任意项x，定义level(x)如下：

如果x是根项，

level(x)=0

否则

level(x)=1+level(parent(x))

项的层次也称作是项的**深度**。一个非空二叉树的高度是最远的树叶的深度！

二-树是这样一种二叉树：它要么为空，要么每个非叶节点连接两条向下延伸的树枝。例如，图8-2a是一个二-树，而图8-2b就不是二-树。递归地说，二叉树t是一个**二-树**，当

t为空

或者

t的两个子树为空或t的两个子树都是非空二-树。

二叉树t为**满树**：是指t是所有树叶都在同一层上的二-树。例如，图8-3a是满树，而图8-3b则不是满树。递归地说，二叉树t是**满树**，当

t为空

或者

311 t的左右子树高度相同而且都是满树。

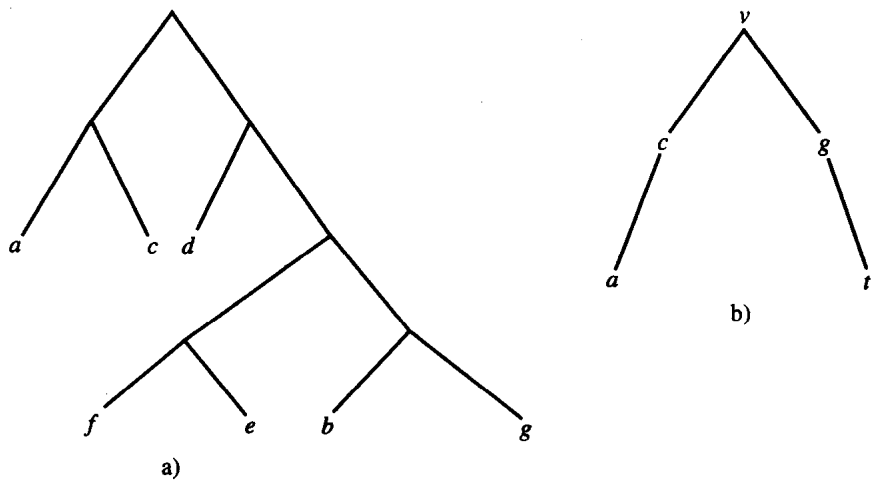


图8-2 a)一个二-树；b)一个不是二-树的二叉树

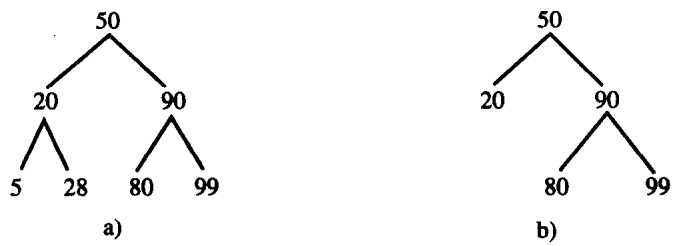
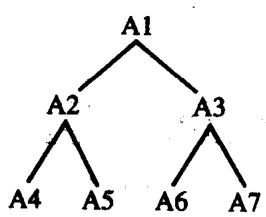


图8-3 a)一个满二叉树；b)一个不满的二叉树

当然，每个满二叉树都是二-树，但是反过来就未必成立。例如，图8-3b是一个二-树，但不是满树。对满二叉树来说，在树的高度和项的数量之间有一个关系。例如，如果满二叉树高度为2，那么该树必定恰好有7项：



高度为3的满二叉树中有多少项？高度为4的呢？对一个满二叉树t，能不能为t中项的数量推测出一个和height(t)相关的公式呢？

比“满”稍微弱一些的表示是“完全”。当二叉树t在height(t)-1层次上为满且最低层的全部树叶尽可能向左排列时，就称它是完全的。“最低层”是指距离根最远的层次。

任何满二叉树都是完全的，但并非所有完全二叉树都是满的。图8-4显示了几个二叉树。例如，图8-4a是一个完全二叉树但不是满二叉树。图8-4b中的树不是完全的，因为C只有一个子女。图8-4c中的树也不是完全的，因为树叶I和J并没有尽可能地向左排列。

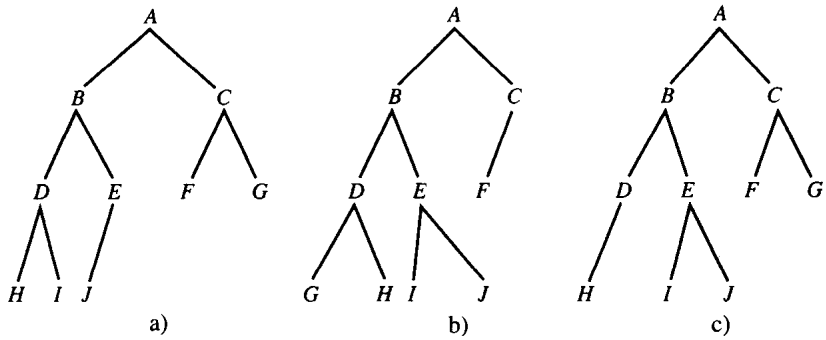
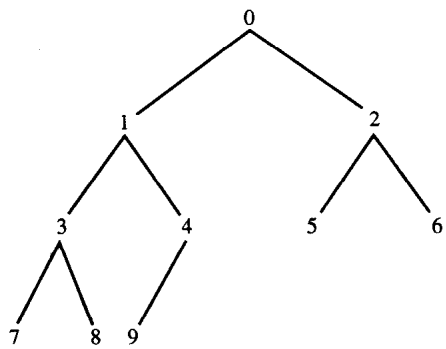


图8-4 三个二叉树，其中只有a)是完全的

在一个完全二叉树中可以为每一项关联一个索引。根项分配的索引是0。对任意正整数i，如果索引i上的项有子女，那么它的左子女索引是$2i+1$，右子女的索引是$2i+2$。例如，如果完
312 全二叉树有10项，那么这些项的索引如下：



索引8上的项的父亲位于索引3，而索引5上的项的父亲位于索引2。通常来说，如果i是一
313 个正整数，那么索引i上的项的父亲位于索引$(i-1)/2$。

项的索引是很重要的，因为据此我们可以将一个完全二叉树的项存入一个数组。具体地讲，就是把在树中索引i上的项就存到数组中索引为i的位置。例如，下面是保存了图8-4a中项的数组：

A	B	C	D	E	F	G	H	I	J

实际上，所要做的（在第11章中）就是把项存入数组，然后访问这些项，就好像它们仍然在一个完全二叉树中一样。因此完全二叉树是可以用数组实现的一种抽象形式。大部分访问将是从父亲的索引到子女的索引，或是从子女的索引到父亲的索引。不仅可以快速计算出这些索引[⊖]，而且由于数组的随机访问属性，也能够快速地读取相应的项。

前面已经说明了如何递归地计算二叉树t的树叶数量leaves(t)和它的高度height(t)。同理，也可以递归计算t中项的数量$n(t)$：

如果t为空

$n(t)=0$

⊖ 在位级别上，比如，对i进行一个左移位然后加1就能得到$2i+1$的数值。这是很快的，是机器层的操作。

否则

$n(t)=1+n$ (leftTree(t)+n(rightTree(t))

8.1.1 二叉树定理

对任意二叉树t，leaves(t) ≤ $n(t)$，且leaves(t)=$n(t)$当且仅当t只由一项组成时。下面的定理刻画了leaves(t)、height(t)和$n(t)$之间的关系。

二叉树定理 对任意非空二叉树t，

1) $\text{leaves}(t) \leqslant \dfrac{n(t)+1}{2.0}$

2) $\dfrac{n(t)+1}{2.0} \leqslant 2^{\text{height}(t)}$

3) 如果t是一个二-树，那么 $\text{leaves}(t)=\dfrac{n(t)+1}{2.0}$

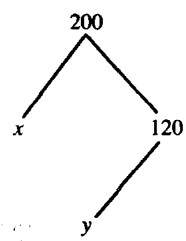
4) 如果 $\text{leaves}(t)=\dfrac{n(t)+1}{2.0}$，那么$t$是一个二-树。

5) 如果t是满树，那么 $\dfrac{n(t)+1}{2.0}=2^{\text{height}(t)}$

6) 如果 $\dfrac{n(t)+1}{2.0}=2^{\text{height}(t)}$，那么$t$是满树。

314

注意 因为二叉树定理中等式的分母是2.0，所以商是一个浮点数。例如，7/2.0=3.5。不能使用整数除法是因为定理的第4部分。令t是下面的二叉树：



在这个树中，leaves(t)=($n(t)$+1)/2，但是t不是二-树。

定理的这6个部分都可以用t的高度进行归纳证明。附录1给出了一些详细情况。和这个证明一致，大部分有关二叉树的定理都可以用树的高度进行归纳证明。原因是如果t是一个二叉树，那么leftTree(t)和rightTree(t)的高度都小于height(t)，因此往往就可以采用数学归纳法。例如，下面给出了二叉树定理的第2部分的证明（附录1的例5是第1部分的证明）：

证明 对k=0,1,2,...，令S_k是语句：

如果t是高度为k的二叉树，

那么 $\dfrac{n(t)+1}{2.0} \leqslant 2^{\text{height}(t)}$

1) 基本情况。如果k=0，那么t只有一个项，因此

$$\frac{n(t)+1}{2.0}=\frac{1+1}{2.0}=1=2^0=2^{\text{height}(t)}$$

即S_0为真。

2) *归纳情况*。令k是任意非负整数，假设$S_0,S_1,\ldots,S_k$为真。令t是高度为$k+1$的二叉树。t中的任意一项要么是根，要么是左子树或右子树。也就是说，

$$n(t) = 1+n(\text{leftTree}(t))+n(\text{rightTree}(t))$$

因此有

$$\frac{n(t)+1}{2.0} = \frac{1+n(\text{leftTree}(t))+n(\text{rightTree}(t))+1}{2.0}$$
$$= \frac{n(\text{leftTree}(t))+1}{2.0} + \frac{n(\text{rightTree}(t))+1}{2.0}$$

leftTree(t)和rightTree(t)这两个高度都小于height(t)，因此leftTree(t)和rightTree(t)都是$\leqslant k$的，应用归纳假设，即，

$$\frac{n(\text{leftTree}(t))+1}{2.0} + \frac{n(\text{rightTree}(t))+1}{2.0} \leqslant 2^{\text{height}(\text{leftTree}(t))} + 2^{\text{height}(\text{rightTree}(t))}$$

315

设hmax为height(leftTree(t))和height(rightTree(t))的最大值。那么

$$2^{\text{height}(\text{leftTree}(t))} + 2^{\text{height}(\text{rightTree}(t))} \leqslant 2^{h\max} + 2^{h\max} = 2^{1+h\max} = 2^{\text{height}(t)}$$

根据前面的等式和不等式可以得到

$$\frac{n(t)+1}{2.0} \leqslant 2^{\text{height}(t)}$$

这就验证了归纳情况是正确的。综上所述，根据数学归纳原理，对任意非空二叉树t，二叉树定理的第2部分都是成立的。

满二叉树的高度和树中项的数量n成对数关系。

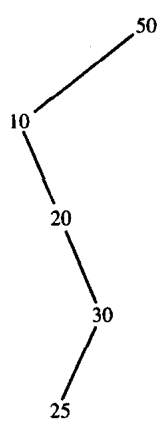
如果t是一个满二叉树，那么根据二叉树定理以及任意空树高度为-1，可以推断

$$\text{height}(t) = \log_2((n(t)+1)/2.0)$$
$$= \log_2(n(t)+1)-1$$

链的高度和树中项的数量n成线性关系。

因此可以说一个满树的高度是和n成对数关系的，其中n是树中项的数量（当所指的树很明确时，通常可以用n代替$n(t)$）。即使t只是完全树，它的高度也是和n成对数关系的。参见习题8.7。另一方面，t可能是一个链。**链**是每个非叶节点都恰好有一个子女的二叉树。例如下面就是一个链：

316



如果t是一个链，那么height(t)=n(t)−1，因此链的高度和n是成线性关系的。第9和第10章中有关树的工作都是围绕着如何保持对数高度以及避免线性高度展开的。基本上，在一个高度和n成对数关系的二叉树中进行插入和删除的worstTime(n)和n也是成对数关系的。这也是之所以在很多应用中树更适宜于顺序容器的原因。例如，假设要将项按顺序存储在容器中，那么使用向量、双端队列和链表插入或删除指定项的worstTime(n)和n是成线性关系的。

8.1.2 外部路径长度

读者可能会奇怪我们为什么对从根到树叶的路径长度的总和感兴趣，但是下面的定义是有很大实践意义的。令t是一个非空二叉树。t的**外部路径长度**$E(t)$是t中所有树叶的深度总和。例如在图8-5中，树叶的深度总和是2+4+4+4+4+5+5+1=25。

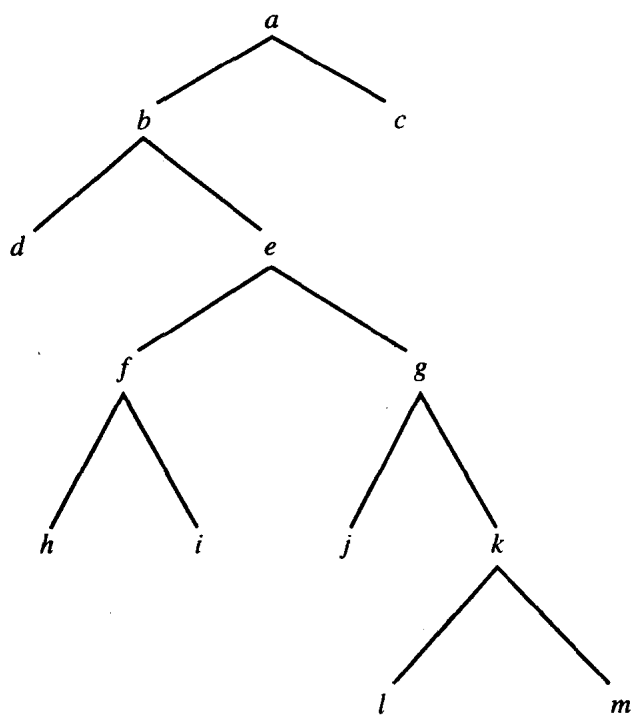


图8-5 外部路径长度是25的二叉树

下面的外部路径长度的下界将得出排序算法中的一个重要结论（参见第12章）。

外部路径长度

令t是有k（k>0）个树叶的二叉树，那么

$E(t) \geq (k/2)\text{floor}(\log_2 k)$

317

证明 令t是有k个树叶的二叉树，k>0。对任意层次L，在L层上树叶的最大数量是2^L（这很容易用数学归纳法证明），而且只有当t满时树叶数量才是2^L。

因此，所有$<L$的层次上树叶的最大数量仍然是2^L，因为在小于L的层次上的任何树叶都不能构成一个满树。在使用floor($\log_2 k$)−1替换L时，由于$2^{\log_2 k}=k$，故所有小于等于floor($\log_2 k$)−1的层次上的树叶的最大数量是

$$2^{\text{floor}(\log_2 k)-1} < 2^{\log_2 k-1} = k/2$$

所以大于floor($\log_2 k$)−1的全部层次上的树叶的最小数量至少是$k/2$。也就是说，全部大于等于floor($\log_2 k$)的层次上的树叶的最小数量至少为$k/2$。而每个这样的树叶至少为外部路径长度贡献floor($\log_2 k$)，因此有

$$E(t) \geq (k/2)\text{floor}(\log_2 k)$$

注意 在第12章中还需要用到这个结论，但是通过一个更复杂的证明过程，可以说明：对任何包含k个树叶的非空二−树，$E(t) \geq k\log_2 k$。详情可参阅Kruse(1987, pp. 171-178)。

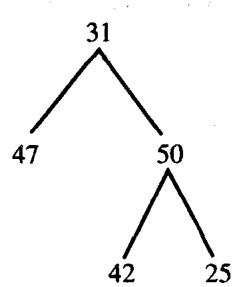
8.1.3 二叉树的遍历

二叉树t的遍历是一个算法，它访问t中的每一项且只访问一次。这里没有BinaryTree类：它不够灵活，不能支持标准模板库里和二叉树相关的数据结构中的各种插入和删除方法。因此下面的算法不是方法。下面指出了四种不同的遍历。

遍历1。中序（inOrder）遍历：左−根−右 假设t是一个二叉树，以下是算法：

```
inOrder(t)
{
    if(t非空)
    {
            inOrder(leftTree(t));
            访问t的根项;
            inOrder(rightTree(t));
    }//if
}//中序遍历
```

在每次递归调用中访问一项。如果n代表树中项的数量，那么worstTime(n)就是和n成线性
318 关系的。可以使用这个递归描述列出下面二叉树t在中序遍历中的项：



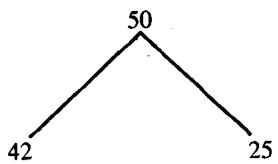
树t非空，因此可以开始执行leftTree(t)的中序遍历，即

47

这只包含一个项的树成为*t*的当前值。因为它的左子树为空，所以访问这个*t*的根，即47；这样就完成了这个*t*的遍历，因为rightTree(*t*)也是空。现在*t*再一次指向原先的树。下一个要访问的是*t*的根项，即

31

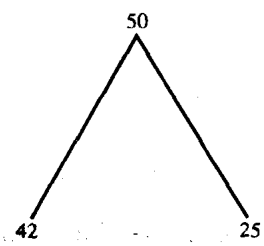
然后，执行rightTree(*t*)的中序遍历，即



这成为*t*的当前值。那么先执行leftTree(*t*)的中序遍历，即，

42

现在这个只包含一个项的树成为*t*的当前值。因为它的左子树为空，所以访问*t*的根项42。这个*t*的右子树也是空。因此对这个树的中序遍历就是访问这个惟一的项42，现在*t*再一次地指向包含三个项的二叉树：



下一个将访问这个*t*的根项，即，

50 319

最后，执行rightTree(*t*)的中序遍历，即，

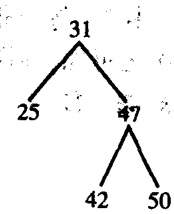
25

因为这个只包含一个项的树*t*的左子树为空，所以访问它的根项25。由于*t*的右子树也是空，所以至此就全部完成了。

完整的清单是：

47 31 42 50 25

中序遍历的命名来自于对一种特殊的二叉树——折半查找树——的访问，中序遍历可以按照顺序访问项。例如，下面是一个折半查找树：

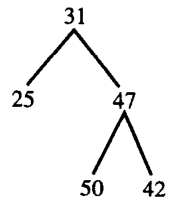


中序遍历将按如下顺序访问项:

25 31 42 47 50

在一个折半查找树中，左子树中全部的项都小于等于根项，而根项又小于等于全部的右子树的项。读者认为折半查找树还必须具备什么属性，才能使得中序遍历可以按顺序访问它的项?

提示 下面的树不是折半查找树:



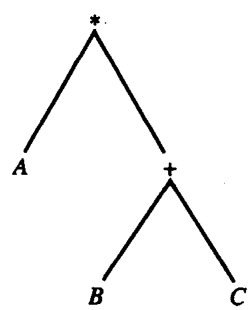
本章剩余的大部分和第9章、第10章的全部内容都是讨论折半查找树的。

遍历2。后序（postOrder）遍历：左-右-根 在二叉树t上的算法是:

```
postOrder(t)
{
    if(t非空)
    {
            postOrder(leftTree(t));
            postOrder(rightTree(t));
            访问t的根项;
    }//if
}//后序遍历
```

因为在每次递归调用中访问一项，所以worstTime(n)和n是成线性关系的。

假设在下面的树中进行后序遍历:



将按照以下顺序访问项:

*A B C + **

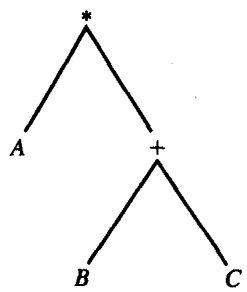
这个二叉树可以看作是一个“表达式树”：每个非叶项都是一个二元运算符，它的操作数正是对应的左子树和右子树。根据这个阐述可知，后序遍历将得到表达式的后缀表示!

遍历3。前序（preOrder）遍历：根-左-右 在二叉树t上的算法是:

```
preOrder(t)
{
```

```
    if(t非空)
    {
            访问t的根项;
            preOrder(leftTree(t));
            preOrder(rightTree(t));
    }//if
}//前序遍历
```

和中序、后序算法一样，它的worstTime(n)也是和n成线性关系的。



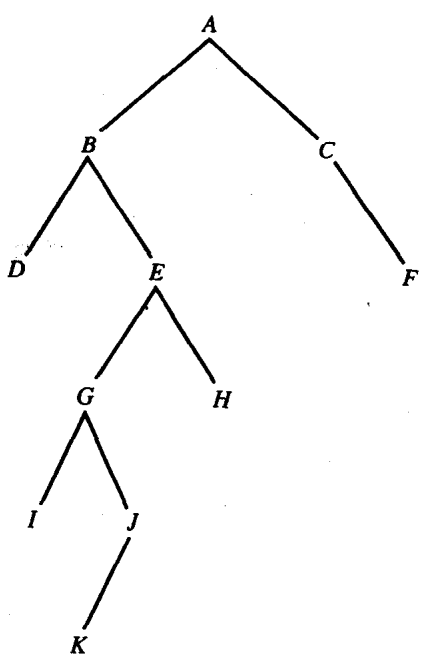
321

上图的前序遍历将按照如下顺序访问项:

$$* A + B C$$

对一个表达式树来说，前序遍历将得到表达式的前缀表示。

使用前序遍历的二叉树搜索称作**深度优先搜索**，因为它总是先从左边尽可能地深入，然后再搜索右边。例如，假设对下面的树进行深度优先搜索:



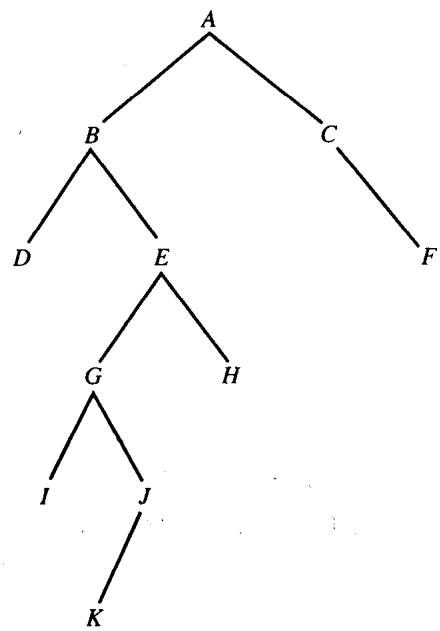
如果搜索的目标是H，那么项的访问次序是:

$$A\ B\ D\ E\ G\ I\ J\ K\ H$$

第4章的**回溯**方法就包含了深度优先搜索，但是它的每一步中都可能有多于两个的选择。例如，在迷宫搜索中，可以选择向北、东、南和西移动。因为向北移动是首选，所以将不断重复这个选择，直到到达目的地或是不能再向北移动了。然后可能的话将向东移动，然后再尽可能向北移动。在第14章里，将在广义二叉树中再次遇到回溯。

遍历4。广度优先（breadthFirst）遍历：逐层遍历 在一个非空二叉树t中执行广度优先遍历，首先访问根项；接着自左至右访问根的子女；然后自左至右访问根的孙子；依次类推。

322 例如，假设在下面的树中执行广度优先遍历：



那么项被访问的次序是

A B C D E F G H I J K

实现这个遍历的一个方法是逐层地生成非空子树（指针的）链表。我们需要检索符合产生的顺序的这些子树，这样才能够逐层地访问项。什么样的容器的检索顺序和插入顺序相同呢？是队列！下面是二叉树t上的算法：

```
breadFirst(t)
{
    //my_queue是二叉树的队列
    //tree是一个二叉树
    if(t非空)
    {
            my_queue.push(t);
            while(my_queue非空)
            {
                    tree=my_queue.front();
                    my_queue.pop();
                    访问tree的根;
                    if(leftTree(t)非空)
```

```
                my_queue.push(leftTree(t));
            if(rightTree(t)非空)
                my_queue.push(rightTree(t));
        }//while
    }//如果t非空
}//广度优先遍历
```

323

在每次循环迭代中访问一个项，因此worstTime(n)和n是成线性关系的。

使用队列进行广度优先遍历是因为需要按照它们保存的顺序检索这些子树（先入，先出）。在中序、后序和前序遍历里，子树按照和它们保存顺序相反的顺序检索（后入，先出）。这三种方法都使用了递归，正如在第7章中看到的，这等价于一个迭代的、基于堆栈的算法。

在第14章中学习广度优先遍历的数据结构时还会遇到这种遍历，它的数据结构的限制不如二叉树那么严格。顺便提一下，如果希望更严格，像一个完全二叉树，那么可以将树存储在一个数组里，而广度优先遍历只不过是迭代地通过数组。根在索引0，根的左子女在索引1，根的右子女在索引2，根的最左面的孙子在索引3，依次类推。

在完成二叉树的全部理论介绍之后，读者可能已经准备好学习一个现实的类了。在8.2节中开发了BinSearchTree类，从用户的角度来看，它首先是一个数据结构。在BinSearchTree里插入和删除的averageTime(n)和n是成对数关系的。这对插入和删除的averageTime(n)和n成线性关系的有序向量、双端队列和链表而言是一个显著的改进。但是在最坏情况下，BinSearchTree并不比顺序容器类强：BinSearchTree中的插入和删除的worstTime(n)和n是成线性关系的。

如果想找到插入和删除的worstTime(n)和n是成对数关系的类，那就必须等到第9章学习了AVL树之后。AVLTree类和第10章的rb_tree（红黑树）的插入和删除在最坏情况下的时间花费是对数关系的。但是它们的方法定义——例如惠普的rb_tree类的实现——比折半查找树的要复杂许多。从实际角度来说，学习BinSearchTree类的主要目的是为学习AVLTree和rb_tree类做准备。

8.2 折半查找树

另一个递归定义。

先从折半查找树的递归定义开始：

> **折半查找树**t是一个二叉树，它满足t要么为空，要么为
> 1) leftTree(t)中的每一项都小于等于t的根项。
> 2) rightTree(t)中的每一项都大于等于t的根项。
> 3) leftTree(t)和rightTree(t)都是折半查找树。

图8-6显示了一个折半查找树。

324

折半查找树的中序遍历将按照增序访问项。例如，对图8-6中的折半查找树，中序遍历访问的次序是

15 25 28 30 32 36 37 50 55 59 61 68 75

正如在折半查找树中曾经定义过的，树中是允许重复项的。有些作者在折半查找树的定

义中使用了“小于”和“大于”。为了和随后的第10章保持一致，本书选用了“小于等于”和“大于等于”。

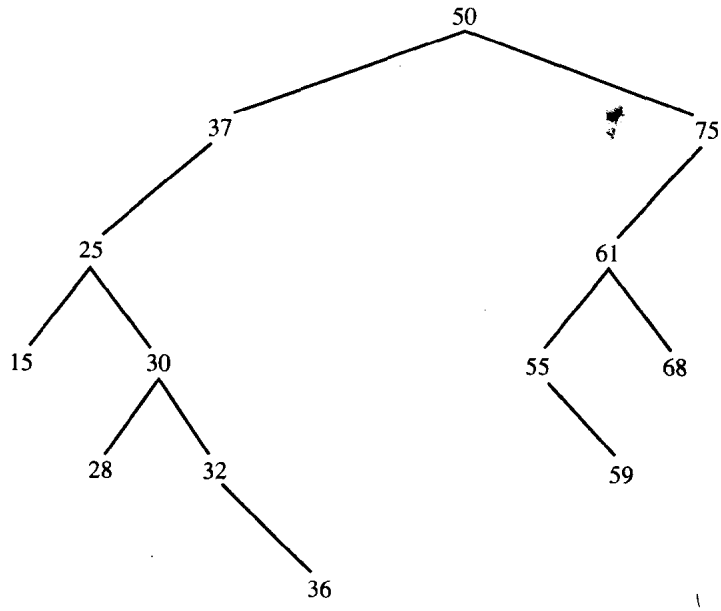


图8-6 一个折半查找树

折半查找树是关联容器的一个范例。在一个**关联容器**中，项通过和其他项的比较确定它在容器中的位置。每个项有一个**键**：它是项的一部分，用来进行比较。在第9章和第10章中将学习其他几个关联容器。

8.2.1节从BinSearchTree类的用户角度描述了折半查找树的数据结构。

8.2.1 BinSearchTree类

下面通过BinSearchTree类的方法接口学习折半查找树的数据结构。对BinSearchTree类，
325 假设每个用户将提供一个项的类T的实例化版本，以及用**operator**<来比较项。在和T对应的模板变元中无需显式地定义**operator**==，因为a==b就等价于!(a<b)&&!(b<a)。

*对应于模板参数T的项的类必须定义**operator**<来比较项。*

现在用BinSearchTree类来简单地介绍一下折半查找树。对反复运行的程序，成品工作建议采用标准模板库的关联容器（参阅第10章）。因此，为BinSearchTree类赋予大量的方法还不如采用相反的途径。下面是最小的折半查找树类中的8个方法接口（为了进行验证，实验20中添加了一个height方法）：

```
1. //后置条件: 这个BinSearchTree为空。
   BinSearchTree();
2. //后置条件: 返回这个BinSearchTree中项的数量。
   unsigned size() const;
3. //后置条件: 如果在BinSearchTree中有一个项等于item，那么返回的
   //          将是指向该项的迭代器。否则，返回值将是和end()方法返回值相同的迭代器。
   //          averageTime(n)是O(logn)，worstTime(n)是O(n)。
```

```
Iterator find (const T& item) const;
```

注意 这是BinSearchTree类中的一个方法，而不是通用型算法find，后者的averageTime(n)是$O(n)$而不是$O(\log n)$。

```
4. //后置条件：将item插入这个BinSearchTree。返回位于新插入项上的迭代器。
   //          averageTime(n)是O(logn)，worstTime(n)是O(n)。
   Iterator insert (const T& item);
```

注意1 没有参数指定从**哪里**插入项。这是因为必须根据顺序将项插入到属于它的位置上。如果允许用户指定项插入的位置，那么这个插入将破坏顺序，也就不再是一个折半查找树了。

注意2 用户可以排除在折半查找树中插入重复项的情况。假设dictionary是BinSearchTree类的一个对象，那么只有当item不在dictionary中时才将其插入：

```
if (dictionary.find(item)==dictionary.end())
    dictionary.insert(item);
```

```
5. //前置条件：itr位于这个BinSearchTree的某一项上。
   //后置条件：从BinSearchTree中删除itr位置上的项。本次调用前*itr之后的
   //          所有的迭代器将失效。worstTime(n)是O(n)。amoritizedTime(n)是常数，
   //          因此averageTime(n)是常数。
void erase(iterator itr);
```

326

注意 为了在折半查找树中删除任意一项，考虑结合find和erase方法。例如，为了从BinSearchTree对象dictionary中删除word的一个拷贝：

```
dictionary.erase(dictionary.find(word));
```

可以使用循环在BinSearchTree对象中删除某一项的全部拷贝。

```
6. //后置条件：如果这个BinSearchTree非空，那么将返回树中位于最小的项上的
   //          一个迭代器。否则，将返回和end()方法返回值相同的迭代器。
   Iterator begin();
7. //后置条件：返回值是一个可以用来判断这个BinSearchTree遍历结束的迭代器。
   //          如果这个BinSearchTree非空，那么在返回的迭代器的前一个位置上就是最大
   //          的项。
   Iterator end();
8. //后置条件：释放为这个BinSearchTree分配的空间。worstTime(n)是O(n)。
   ~BinSearchTree();
```

这些方法接口中缺少了什么？折半查找树缺少顺序容器类的支柱方法：push_back，pop_back，push_front和pop_front方法。推入操作在这里是不合法的，因为就像在insert方法（方法4）的“注意1”中所解释的，不允许用户指定项插入的位置。弹出是合法的，但是这里没有包含它们，因为用户很少会希望删除最小或最大的项，除非知道这个项上将发生什么。而在这些情况中可以使用erase方法；在指定Iterator类的方法接口之后将说明如何完成这项工作。

8.2.2 BinSearchTree类的Iterator类

折半查找树的迭代器是双向的，因此这个迭代器的方法和list类的迭代器方法是相同的：

++和--的前缀、后缀版本，*（脱引用），==以及！=。因此如果itr是一个Iterator对象，那么可以采用下面的循环迭代通过一个BinSearchTree对象my_tree:

```
for(itr=my_tree.begin();itr!=my_tree.end();itr++)
327     *itr...//访问itr位置上的项
```

和预期的相同，BinSearchTree迭代器的增量或减量代码比向量、链表甚至是双端队列的迭代器都要复杂很多。

这里用一个小程序说明BinSearchTree类和它的关联Iterator类的使用:

```
#include <string>
#include <iostream>
#include "bst.h" // 定义BinSearchTree类

// 这个程序验证一个折半查找树中的查找、插入和删除。
// search tree.

using namespace std;

int main ( )
{
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    BinSearchTree<int> tree;
    BinSearchTree<int>:Iterator itr;

    tree.insert (85);
    tree.insert (70);
    tree.insert (91);
    tree.insert (70);

    cout << "Here is the tree:" << endl;
    for (itr = tree.begin( ); itr != tree.end( ); itr++)
        cout << *itr << " ";

    if (tree.find (72) == tree.end( ))
        cout << endl << endl << "72 was not found in the tree"
            << endl;;

    itr = tree.find (85);
    if (itr != tree.end( ))
        cout << endl << endl << "85 was found in the tree" << endl;
    tree.erase (itr);
    cout << endl << endl << "Here is the tree after 85 was deleted:"
        << endl;
    for (itr = tree.begin( ); itr != tree.end( ); itr++)
        cout << *itr << " ";

    cout << endl << endl << "The largest item in the tree is "
                << *--tree.end( );
328 cout << endl << endl << "size = " << tree.size ( );

    cout << endl << endl << CLOSE_WINDOW_PROMPT;
```

```
    cin.get( );
} // binsearchtreeexample
```

输出如下:

```
Here is the tree:
70 70 85 91

72 was not found in the tree

85 was found in the tree

Here is the tree after 85 was deleted:
70 70 91

The largest item in the tree is 91

size = 3

Please press the Enter key to close this output window.
```

注意树中最大的项是如何被访问的: 因为tree.end()返回紧随最大项之后位置上的迭代器，所以--tree.end()将返回一个位于最大项位置上的迭代器。那么如何访问树中的最小项呢?

现在将注意力转去关注BinSearchTree类的一种可能的实现。编程项目8.1探讨了另一种实现。

8.2.3 BinSearchTree类的字段和实现

BinSearchTree类中的字段和第6章中list类的字段很相似。主要的字段是header，是指向tree_node结构的指针:

```
struct tree_node
{
    T item;
    tree_node* parent,
             * left,
             * right;
    bool isHeader;   // 指示这个节点是不是头节点
}; // tree_node

tree_node* header;
```

BinSearchTree类中惟一不同的字段是:

```
unsigned node_count;
```

329

node_count字段记录了树的大小。当创建一个树时，node_count的值为0，而在header的tree_node中，parent链接值为NULL，left和right链接指回header，并且isHeader值为**true**。item字段是未定义的；实际上，header的tree_node的item字段将始终保持未定义状态，就像list类的data字段一样。图8-7显示了一个空BinSearchTree对象的配置。

当向一个空树中添加项时，该项的值被拷贝到根节点的item字段中，根节点的parent字段将指回header节点，而根节点的left和right字段值为**NULL**。header中所有的指针字段现在都指

向这个根节点。isHeader字段的目的是区分header节点和根节点。例如，在将项17插入一个空树之后，得到如图8-8所示的树。

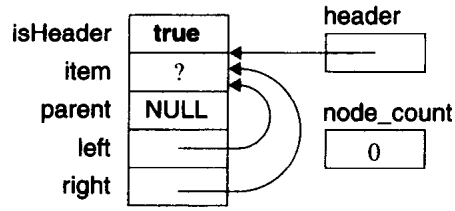


图8-7 一个空BinSearchTree对象的表示

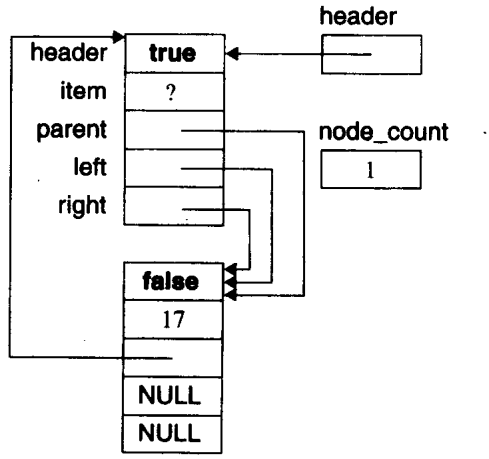


图8-8 包含一个项（17）的BinSearchTree对象的内部表示

注意这种表示法的一个奇怪的特点：根节点的父亲是头节点，而头节点的父亲是根节点！因此根节点是它自身的祖父节点。曾经有过一首畅销歌曲是“I'm My Own Grandpa”⊖。

330

在头节点中，left和right字段分别指向树中最小和最大的项。

当项被添加进这个树中时，将像预期的那样调整根节点的left和right字段。header引用的节点的left和right字段也进行了调整，这可能是出乎意料的。它们指向的节点分别包含最小和最大的项。例如，图8-9显示了包含5个项的BinSearchTree对象的内部表示。

令header的left字段指向最小的项，这可以使begin()方法的设计变得快速简洁：直接返回位于header的left字段指向的节点上的迭代器。那么最大项怎么办？end()方法返回位于header的节点（也就是最后一个节点之后的位置）上的迭代器。因此用户可以像8.2.2节中的程序一样，从末尾退出之后访问BinSearchTree的最大项。结果是访问header的right字段指向的项。

在完成BinSearchTree类之后将实现Iterator类，但是现在所有需要了解的有关Iterator的字段和实现，就是它应该有指针字段和一个参数的构造器：

```
typedef tree_node* Link;
```

⊖ *M*和有一个已成年的女儿*D*的寡妇*W*结婚。*D*和*M*的爸爸结婚并有了一个孩子*S*。这样*S*就是*M*的兄弟（因为它们有相同的父亲），同时也是*M*的“孙子”（因为*M*的妻子是*S*的祖母）。但是*S*的“祖父”也是*S*的兄弟的祖父，因此*M*是他自己的祖父。

```
protected:
    Link link;
    Iterator(Link new_link):link(new_link){}
```

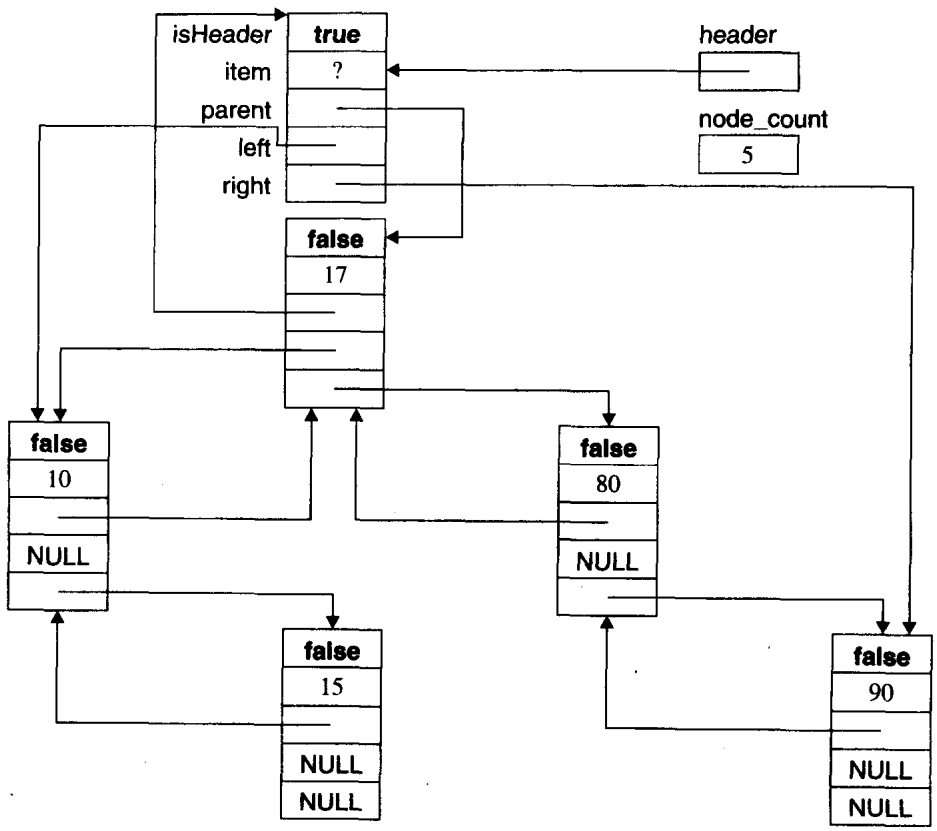


图8-9 包含5个项的BinSearchTree对象的内部表示 331

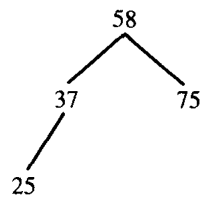
回忆一下，list类的关联iterator类也有相似的指针字段和一个参数的构造器，并且它们也都是**protected**的。

接下来看BinSearchTree类的三个基本方法——find、insert和erase——的定义。对find方法，从一个指向根的child开始在树中下降，根据child->item和所搜索项的比较来决定用child的左子女还是右子女替换child。方法定义是：

```
Iterator find( const T& item)
{
    Link parent = header;
    Link child = header -> parent;
    while (child != NULL)
    {
        if (! (child -> item < item) )
        {
            parent = child;
            child = child -> left;
```

```
            } // if <
            else
                child = child -> right;
        } // while
        if (parent == header || item < parent -> item)
            return end( );
        else
            return parent;
    } // find
```

例如，假设从下面的折半查找树开始：



如果在这个树中查找30，可以得到下面的parent和child的值序列（当指针非空时就给出它脱引用的项）：

parent	child
NULL	58
58	37
37	25
	NULL

332

现在循环终止，因为child的值为NULL。并且由于item小于parent->item，所以返回值是位于树中最后一项之后的迭代器。

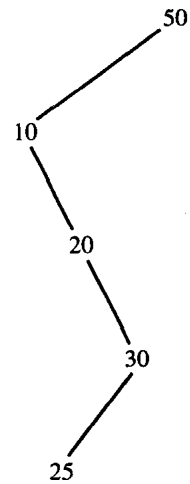
这个方法的一个与众不同的特点是它并非访问了树中搜索的项就停止！（这和实验11中binary_search算法很相似。）假设，比如说在树中搜索37，那么将得到下面的parent和child的值序列（指针非空时就脱引用）：

parent	child
NULL	58
58	37
37	25
	NULL

在第二次循环迭代中，child->item等于搜索的项，即37。那么将child保存在parent中并用child->left替换child。从这往后，child项将不再大于搜索的项。也就是说，从那时起的child项将小于等于搜索的项。在循环结束时，parent不等于header，而且item不小于parent->item，因此将返回parent（位于parent的迭代器）。为什么在这个循环的设计中找到项了也不能停止呢？这是出于效率的考虑：每次循环迭代只进行一次而不是两次比较。

这个方法将花费多长时间？对这个方法，实际是BinSearchTree类的所有不容易的方法，估算worstTime(n)或averageTime(n)的基本要素是树的高度。假设搜索成功（搜索失败的情况

也可以采用相似的分析），最坏情况下是在一个链上寻找树叶。例如，假设在下面的折半查找树上寻找25:



333

在这种情况中，循环迭代的次数就等于树的高度。一般来说，如果树中项的数量是n，树的高度是$n-1$，那么worstTime(n)就是和n成线性关系的。

现在求一个成功查找的averageTime(n)。关键仍然是树的高度。对通过随机插入和删除构造的折半查找树而言，平均高度H是和n成对数关系的（参见Cormen，1992）。实验20里测试了这个结论。find方法先在树的第0层搜索，而且每次循环迭代都下降至树中的下一层。因为averageTime(n)需要不超过H次迭代，所以马上可以得出结论：averageTime(n)是$O(\log n)$。

为了证实averageTime(n)和n成对数关系，需要说明$O(\log n)$是averageTime(n)的最小上界。所有折半查找树的平均迭代次数是大于等于一个完全折半查找树的平均迭代次数的。在一个完全二叉树t中，至少有一半的项是树叶（见习题8.13）。因此find()方法的平均迭代次数必定至少是(height(t)−1)/2，即(ceil($\log_2(n(t)+1)$)−2)/2，见习题8.7。也就是说find方法的平均迭代次数大于等于log$n$的某一函数。因此averageTime(n)不可能再比$O(\log n)$好了。

对BinSearchTree类中的find方法，averageTime(n)是和n成对数关系的。

通过前两段可以断定averageTime(n)是和n成对数关系的。顺带说一下，这就是之所以定义find方法而不使用通用型算法find的原因。后者的averageTime(n)是和n成线性关系的。

如果折半查找树是满的，那么树的高度就和n成对数关系。正是由于这种情况折半查找树才得以命名。这时应用find方法和在数组中应用binary_search通用型算法访问的就是相同顺序的相同项。例如，满树中的根项对应着数组中间的项。

8.2.4 递归方法

find方法定义的一个新奇之处在于它不是递归的。本章之前的大部分概念（包括折半查找树自身）都是递归定义的。但是当它成为方法定义时，毫无例外的都需要循环。为什么这样呢？一个圆滑的解答是left和right都是tree_node*类型，而不是BinSearchTree类型的，因此不能调用

```
left.find(item)//不合法的
```

但是可以使用一个递归的findItem方法，使find成为它的一个包装方法：

```
bool find (const T& item)
{
    return findItem (header -> parent, item);
} //  方法  find
```

findItem方法是相当简单的：

```
Iterator findItem (Link link, const T& item)
{
    if (link == NULL)
        return end( );
    if (link -> item < item)
        return findItem (link -> right, item);
    if (item < link -> item)
        return findItem (link -> left, item);
    return link; // 自动类型转换成Iterator
}
```

334

不论在时间还是存储上，这个递归版本的效率比迭代版本都要稍微差一些（尽管大O时间估算是相同的）。正是这个差别降低了这个递归版本的价值。迭代版本实际上和即将在第10章中看到的rb_tree类的惠普的实现中的方法是相同的，它在优雅的基础上实现了高效率。另外，对包装方法的需要也减少了递归的一些光彩。

insert方法和find方法很相似，因为都使用循环在树中下降并保存调整过的节点指针的父亲。当插入项成为树最左边或最右边项时要特别小心。下面是它的定义：

```
Iterator insert (const T& item)
{
    if (header -> parent == NULL)
    {
        insertLeaf (item, header, header -> parent);
        header -> left = header -> parent;
        header -> right = header -> parent;
        return header -> parent;
    } // 在树的根处插入
    else
    {
        Link parent = header,
             child = header -> parent;
        while (child != NULL)
        {
            parent = child;
            if (item < child -> item)
                child = child -> left;
            else
                child = child -> right;
```

```
        } // while
        if (item < parent -> item)
        {
            insertLeaf (item, parent, parent -> left);
            if (header -> left == parent) // parent -> item 是最小
                                          // 的项
                header -> left = parent -> left;
            return parent -> left;
        } // 在父亲的左边插入
        else
        {
                insertLeaf (item, parent, parent -> right);
                if (header -> right == parent) // parent -> item 是最大
                                               // 的项
                header -> right = parent -> right;
            return parent -> right;
        } // 在父亲的右边插入
    } // 树非空
} // insert
```

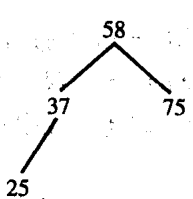
335

insert方法将一项作为树叶插入到BinSearchTree对象中。

insertLeaf方法实际上是将项作为树叶插入，调整链接并令node_count加1:

```
void insertLeaf (const T& item, Link parent, Link& child)
{
    child = new tree_node;
    child -> item = item;
    child -> parent = parent;
    child -> left = NULL;
    child -> right = NULL;
    child -> isHeader = false;
    node_count++;
} // insertLeaf
```

例如，假设在下面的折半查找树中插入30:



将得到下列parent和child的值序列（指针非空时就脱引用）:

parent	child
NULL	58
58	37
37	25
25	NULL

336

现在**while**语句的执行结束，因为child值为NULL。然后，30>=parent->item，所以30将作为一个树叶插入到parent的右边。实际上，*折半查找树中插入的每一项都成为一个树叶*。

insert的分析和find的相同：averageTime(n)和n成对数关系，但是worstTime(n)和n成线性关系。insert的递归版本比迭代版本要优美些（也稍微慢些）。就像处理递归find方法一样，还是从一个包装方法开始：

```
Iterator insert (const T& item)
{
    if (header -> parent == NULL)
    {
        insertLeaf (item, header, header -> parent);
        header -> left = header -> parent;
        header -> right = header -> parent;
        return header -> parent;
    } // 在树的根处插入
    return insertItem (item, header, header -> parent);
} // insert
```

递归方法insertItem如下：

```
Iterator insertItem (const T& item, Link parent, Link& child)
{
    if (child == NULL)
    {
        insertLeaf (item, parent, child);
        if (item < header -> left -> item)
            header -> left = child;
        if (item > header -> right -> item)
            header -> right = child;
        return child;
    }
    if (item < child -> item)
        return insertItem (item, child, child -> left);
    return insertItem (item, child, child -> right);
} // 方法insertItem
```

现在处理erase(Iterator itr)方法。给出Iterator参数itr，需要访问指向itr节点的itr父节点中的字段。如果itr位于根节点，那么这就是parent字段，否则就是itr的父亲的left或right字段。然后改变该指针字段完成对itr节点的删除。因此实现删除的方法（deleteLink）需要一个指针作为引用参数。见习题8.15。

前面已经提到过Iterator类有一个指针字段link，它指向迭代器位于的节点。这是定义erase
337 方法时需要知道的对Iterator类的全部。下面是erase的定义：

```
void erase (Iterator itr)
{
    if (itr.link -> parent -> parent == itr.link) // itr位于根节点
        deleteLink (itr.link -> parent -> parent);
```

```
        else if (itr.link -> parent -> left == itr.link) // itr位于一个左子女处
            deleteLink (itr.link -> parent -> left);
        else
            deleteLink (itr.link -> parent -> right);
    } // erase
```

deleteLink方法删除了参数link指向的节点。如果该节点的左子树或右子树为空，那么只要简单地将这个节点剪除掉即可。也就是说，用它的右子树的节点替换它（如果左子树为空）或者用它的左子树的节点替换它。图8-10显示了这个删除示例发生之前和之后的情况。如果两个子树都非空时删除要困难一些。举例解释这种情况，比如要删除图8-11中的树根。

对有两个子女的项，erase方法将用项的直接后任取代它，然后从树中剪除这个后任。

我们希望在不重构树的情况下完成删除。因此在删除28之前，需要先找到一项来替换28。在这一点上惟一适合的项是直接前任26和直接后任37。例如，如果在图8-11中用37替换28，然后在树中删除以“旧”的37为根的树的根项，将得到如图8-12所示的树。由于即将删除的项的直接后任是它的右子树中最左边的项，那个项没有左子女，因此该项的删除就换成了剪除过程，参见图8-10。

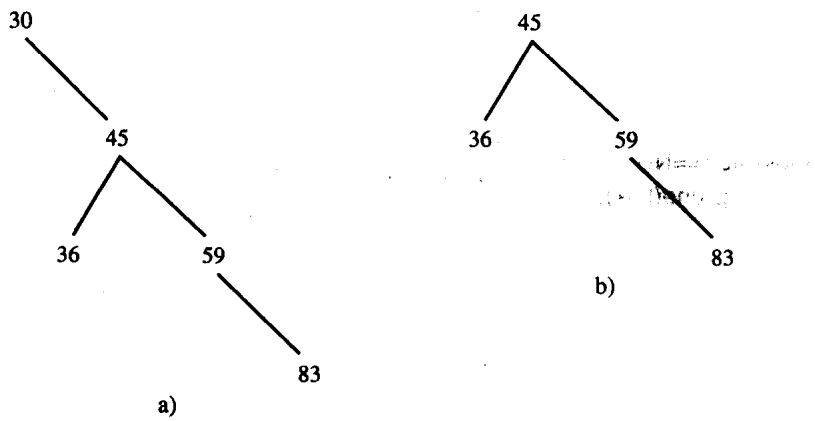


图8-10 a)一个折半查找树，在其中删除左子树为空的节点30；b)30被删除后的折半查找树

338

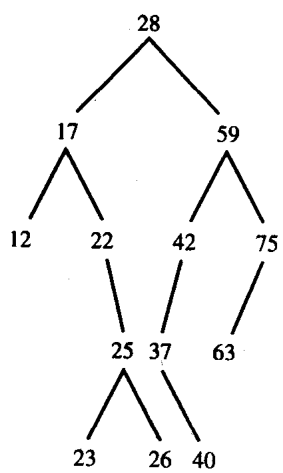


图8-11 即将删除28的折半查找树

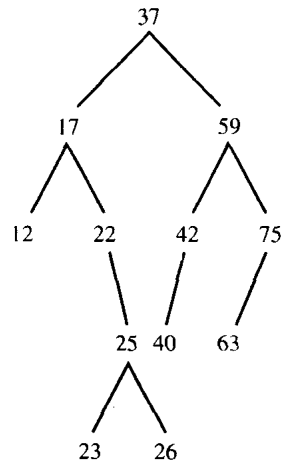
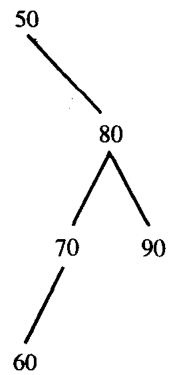


图8-12 在图8-11中的折半查找树中删除28，用28的直接后任37替换28，然后删除“旧”的37之后的折半查找树

至今还没有给出deleteLink方法。prune方法用link的左子树或右子树中适当的一个替换link。查找和删除右子树中后任的工作比较麻烦，把这项工作推迟。

```
//后置条件：删除link指向的项。
339 void deleteLink(Link& link){
    if(link->left==NULL||link->right==NULL)//项没有子女
            prune(link);
    else
            //从link的右子树中删除link的后任。
}//deleteLink
```

prune方法的定义是比较简单的，但被删除的节点包含最小或最大的项时除外。例如，假设要剪除下面树的根：



在删除之前，50是最小的项。为了在删除后获得最小的项，需要从新的根，项80开始尽可能地向左移动，直到包含项60的节点。然后在header的left字段中存储指向这个节点的指针；回想一下，header的left字段总是指向包含最小项的节点。

下面是prune方法的定义：

```
// 前置条件: link指向的子树至少有一个子女。
// 后置条件: 从这个BinSearchTree中删除link指向的项。
// BinSearchTree.
void prune (Link& link)
{
    Link linkCopy = link,
         newLink;

    node_count--;
    if ((link -> left == NULL) && (link -> right == NULL))
    {
        if (link == header -> left)
            header -> left = link -> parent; // 新最左项
        if (link == header -> right)
            header -> right = link -> parent; // 新最右项
        link = NULL;
    } // link的项是树叶
    else if (link -> left == NULL)
    {
        link = link -> right;
        link -> parent = linkCopy -> parent;
        if (linkCopy == header -> left)
        {
            newLink = link;
            while ((newLink -> left) != NULL)
                newLink = newLink -> left;
            header -> left = newLink; // 新最左项
        } // 重新计算最左项
    } // link -> left 非空
    else
    {
        link = link -> left;
        link -> parent = linkCopy -> parent;
        if (linkCopy == header -> right)
        {
            newLink = link;
            while ((newLink -> right) != NULL)
                newLink = newLink -> right;
            header -> right = newLink; // 新最右项
        } // 重新计算最右项
    } // root -> right 非空
    delete linkCopy;
} // prune
```

340

最后，开发代码以在link的右子树中删除它的后任。基本思想是找到这个右子树中最左边的节点。当最后到达一个左子树为空的节点，那么节点中的项就是被删除项的后任。然后用link的后任项替换link的项，再从树中剪除那个后任节点。

下面是deleteLink的定义：

```
// 后置条件: 删除原先在link中的项。
void deleteLink (Link& link)
{
    if (link -> left == NULL || link -> right == NULL)
        prune (link);
    else if (link -> right -> left == NULL)
    {
        link -> item = link -> right -> item;
341     prune (link -> right);
    } // link的右子树的空左子树
    else
    {
        Link temp = link -> right -> left;
        while (temp -> left != NULL)
            temp = temp -> left;
        link -> item = temp -> item;
        prune (temp -> parent -> left);
    } // link的右子树的非空左子树
} // deleteLink
```

现在开始分析最坏情况下的erase方法和它的附属方法: Iterator位于树根，并且树是一个链，链的惟一的树叶是根项的后任。那么deleteLink中循环迭代的次数将是$n-1$。因此worstTime(n)和n成线性关系。平均情况下，查找后任的循环迭代次数就等于Iterator类中**operator**++的迭代次数。在8.2.5节中将证明**operator**++的averageTime(n)是常数；实际上amortizedTime(n)也是常数。因此erase方法的averageTime(n)是常数。

最后开发析构器。使用包装器，递归版本是小巧易用的。包装器调用的destroy方法递归地消除节点的左子树和右子树，然后释放节点的空间:

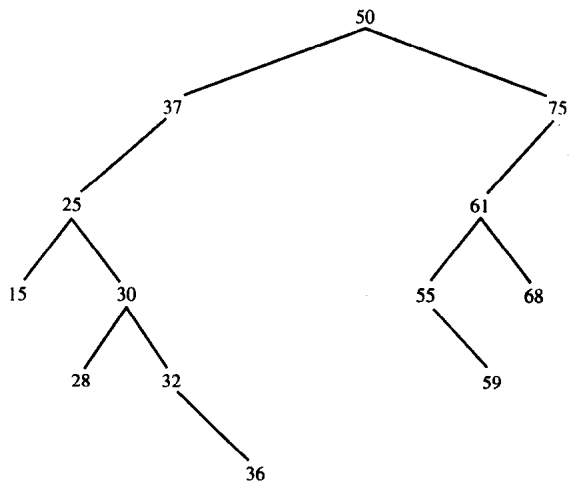
```
~BinSearchTree( )
{
    destroy (header -> parent);
} // 析构器

void destroy (Link link)
{
    if (link != NULL)
    {
        destroy (link -> left);
        destroy (link -> right);
        delete (link);
    } // if
} // destroy 方法
```

在每一对递归调用中释放一个节点，因此总的递归调用次数是$2n$，且worstTime(n)和n成线性关系。析构器的迭代版本有点笨拙，因为一旦节点被释放，那么它的父亲就不再能访问到了。开发迭代版本的提示参阅习题8.8。

8.2.5 BinSearchTree迭代器

在结束对折半查找树的讨论之前，需要对关联Iterator类的设计说几句。正如前面看到的， 342
Iterator类只有一个字段link，它指向某一节点，以及通过一个节点指针创建迭代器的构造器。

脱引用运算符（即**operator***）及**operator**==都用一行定义。但是++和--的定义就不太直接了。解释一下原因，假设有一个迭代器位于下面的树中项50所在的节点，如何增加itr呢？



50的后任是55。为了从50到达这个后任，先向右移动（到75），然后尽可能地向左移动。总是可以这样进行吗？只有对那些有非空右子女的节点才是这样。如果右子女是空的（比如包含项36的节点）会怎样呢？如果一个节点link的右子女为NULL，就在树中向上移动，尽可能地向左找到后任；link的后任是link的最左边的祖先的父亲。例如，36的后任是37。相似地，68的后任是75。同理，28的后任是30；因为28是一个左子女，所以向上向左0次——仍然在28，然后得到该节点的父亲，它的项是30。最后，75的后任是NULL，因为它最左边的祖先是50，它是没有父亲的。

下面是++的前加版本的定义：

```
Iterator& Iterator:operator++ ( )
{
    Link tempLink;

    if ((link -> right) != NULL)
    {
        link = link -> right;
        while ((link -> left) != NULL)
            link = link -> left;                                    343
    } // 节点有右子女
    else
    {
        tempLink = link -> parent;
        while (link == tempLink -> right)
        {
```

```
                link = tempLink;
                tempLink = tempLink -> parent;
            } // 向上移动并尽可能向左
            if ((link -> right) != tempLink)
                link = tempLink;
        } // 节点没有右子女
        return *this;
    } // 前缀 ++
```

operator++()的时间花费不一定是常数。例如，如果将序列n,1,2,3,...,$n-1$插入进一个初始为空的树，那么从$n-1$到n将需要$O(n)$次迭代。因此worstTime(n)是和n成线性关系的。但是在树的遍历中，每个节点至少被指向一次，至多被指向三次：一次到达它的左子树，一次是它自身，一次到达它的右子树的后任。那么遍历树中的n项需要n到$3n$之间次迭代。这暗示着不仅averageTime(n)是常数，而且对**operator**++ ( ) 的n次连续调用，worstTime(n)也只是和n成线性关系。换句话说，**operator**++()的amortizedTime(n)是常数。

惟一需要isHeader字段的地方是减运算符中，它的开头如下：

```
Iterator& operator--()
{
    if (link -> isHeader)
        link = link -> right; // 返回最右边的
    ...
```

这个**if**语句使得反序迭代通过BinSearchTree容器成为可能。不能将条件换成

```
link==header
```

因为BinSearchTree字段，如header，不能用在Iterator类中，除非该字段和一个具体的BinSearchTree对象关联（例如，myTree.header）。并且也不能将条件换成

```
link->parent->parent==link
```

因为这个条件对根也是成立的。包含嵌入Iterator类的完整的BinSearchTree类可以查阅本书网站的源代码链接。

344

实验20提供了对前面观点（BinSearchTree对象的平均高度是和n成对数关系）的运行时支持。

实验20：BinSearchTree的平均高度　　（所有实验都是可选的）

这里没有包含任何BinSearchTree类的应用，因为只要重新定义第9章和第10章中的类（AVLTree、tree、set、multiset、map或multimap）之一的树实例就可以取代任何应用。对这些类中的每一个，即使在最坏情况下，插入、删除和查找都花费对数时间，而所有其他的方法的性能都和BinSearchTree中的相同。

总结

二叉树t要么为空，要么由一项称作**根项**的和两个不相交的称作t的**左子树**和**右子树**的二叉

树组成。这是一个递归的定义，并且还有很多相关术语的递归定义：高度，树叶数量，项的数量，二-树，满树，等等。这些术语之间的相互关系可由下面的定理表示：

二叉树定理 对任意非空二叉树t，

$$\text{leaves}(t) \leqslant \frac{n(t)+1}{2.0} \leqslant 2^{\text{height}(t)}$$

当且仅当t是一个二-树时第一个相等关系成立。

当且仅当t是一个满树时第二个相等关系成立。

对一个二叉树t，t的**外部路径长度**$E(t)$是从根到树叶所有距离的总和。基于比较的排序算法的下界可以由下面的定理获得：

外部路径长度定理 令t是有k个（k>0）树叶的二叉树，那么

$E(t) \geqslant (k/2)\text{floor}(\log_2 k)$

折半查找树t是一个二叉树，它满足t为空，或者

1) leftTree(t)中的每一项都小于等于t的根项。

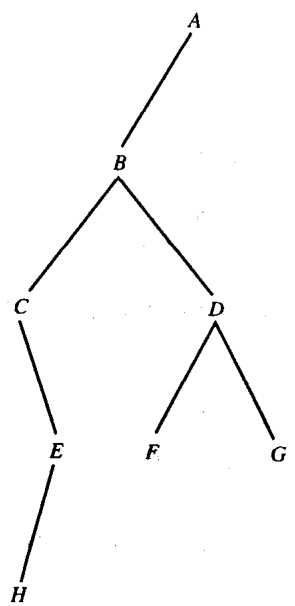
2) rightTree(t)中的每一项都大于等于t的根项。

3) leftTree(t)和rightTree(t)都是折半查找树。

BinSearchTree类保存了一个折半查找树的项。对find、insert和erase方法，worstTime(n)是$O(n)$，其中n是树中项的数量。但是这三个方法的averageTime(n)都是$O(\log n)$。 345

习题

8.1 解答有关下面二叉树的问题：



a. 哪一个是根项?

b. 树中有多少项?

c. 树中有多少树叶?

d. 树的高度是多少?

e. 左子树的高度是多少?

f. 右子树的高度是多少?

g. *F*的层次是多少?

h. *C*的深度是多少?

i. *C*有几个子女?

j. *F*的父亲是哪一个?

k. *B*的子孙有哪些?

l. *F*的祖先有哪些?

m. 在中序遍历过程中输出项的结果是什么?

n. 在后序遍历过程中输出项的结果是什么?

o. 在前序遍历过程中输出项的结果是什么?

346 p. 在广度优先遍历过程中输出项的结果是什么?

8.2 a. 构造一个高度为3、包含8个项的二叉树。

b. 能否构造一个高度为2、包含8个项的二叉树?

c. 当*n*分别取1到20之间的整数时，求*n*个项的二叉树可能的最小高度。

d. 根据在c小题的计算结果，尝试用公式表示*n*个项的二叉树可能的最小高度，其中*n*是任意的正整数。

e. 令*n*是任意正整数。用归纳法验证*n*个项的二叉树可能的最小高度是floor($\log_2 n$)。

8.3 a. 10个项的二叉树中树叶可能的最大数量是多少? 构造一个这样的树。

b. 10个项的二叉树中树叶可能的最小数量是多少? 构造一个这样的树。

8.4 a. 构造一个不是完全树的二-树。

b. 构造一个不是二-树的完全树。

c. 构造一个不满的完全二-树。

d. 在包含17个项的二-树中有多少树叶?

e. 在包含731个项的二-树中有多少树叶?

f. 二-树必须总是包含奇数个项。为什么?

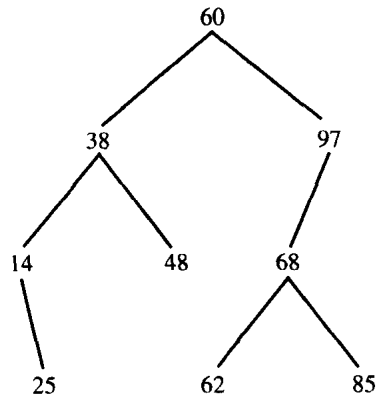
提示 使用二叉树定理以及树叶数量必须是整数这一事实。

g. 高度为4的满二叉树中共有多少项?

h. 高度为12的满二叉树中共有多少项?

i. 使用二叉树定理和每个满树都是二-树这一事实，求包含63个项的满二叉树的树叶数量。

8.5 说明在下面的二叉树中分别采用中序、后序、前序和广度优先遍历时，项的访问次序。



347

8.6 证明一个包含n个项的二叉树有$2n+1$个子树（包括整个树在内）。这些子树有多少是空的？

8.7 证明：如果t是一个完全二叉树，那么

$$\text{height}(t)=\text{ceil}(\log_2(n(t)+1))-1$$

ceil函数返回大于等于它的变量的最小的整数。例如，ceil(35.3)=36。

提示 令t是一个完全二叉树，t1是一个高度比t小1的满二叉树。令t2是一个和t高度相同的满二叉树。因此$n(t1)<n(t)\leqslant n(t2)$。因为log函数是严格递增的，所以：

$$\log_2(n(t1)+1)-1<\log_2(n(t)+1)-1\leqslant\log_2(n(t2)+1)-1$$

第一个不等式左边的值是一个整数，（为什么？）它比第二个不等式右边的值小1。（为什么？）因此

$$\text{ceil}(\log_2(n(t)+1))-1=\log_2(n(t2)+1)-1$$

同理，

$$\log_2(n(t2)+1)-1=\text{height}(t2)\qquad\text{（为什么？）}$$

8.8 二叉树定理是针对非空二叉树表述的。定理的6个部分中哪一个对空树是不成立的？

提示 (0+1)/2.0!=0。

8.9 给出一个不是二-树但$\text{leaves}(t)=(n(t)+1)/2$的非空二叉树的示例。

8.10 证明二叉树定理的第3部分。

提示 对树的高度使用数学归纳法（通用形式）。

8.11 a. 说明将下列项插入一个初始为空的BinSearchTree对象后的结果：

30 40 20 90 10 50 70 60 80

b. 寻找插入后能够和a小题产生相同的BinSearchTree对象的另一个项序列。

8.12 用语言描述如何从折半查找树中删除下面的每个项：

348

a. 没有子女的项。

b. 有一个子女的项。

c. 有两个子女的项。

8.13 证明在任意完全二叉树t中，至少有一半的项是树叶。

提示 如果t为空就没有项，因此结论显然为真。如果位于最高索引的树叶是一个右子女，那么t是一个二-树，结论可以遵循二叉树定理的第3部分。否则就给t增加一个左子女，使它变成一个有$n(t)-1$个项的完全二-树。

8.14 开发BinSearchTree类的deleteLink方法的递归版本。

提示 令deleteLink调用递归方法deleteSuccessor，它的方法接口如下：

```
//后置条件：在这次调用中后任项被删除前，successor中包含了树的link项的后任项。
void deleteSuccessor(T& successor, Link& link);
```

8.15 erase方法体的前两行是：

```
itr(itr.link->parent->parent==itr.link)//itr位于根节点
      deleteLink(itr.link->parent->parent);
```

即使itr.link->parent->parent就等于itr.link，下面的deleteLink调用

```
deleteLink(itr.link);
```

也是不正确的，解释原因。

提示 哪个树节点的什么字段指向根节点（因此如果根节点即将被删除，它也必须随之修改）？

8.16 开发BinSearchTree类的析构器的迭代版本。

提示 实现8.1.2节中的广度优先遍历算法。这里用一个链接队列代替二叉树队列。先向队列中插入根，也就是header->parent。当在队列中删除每个link时，就将它的左右
349 子树的链接插入队列，除非这些链接为NULL。最后，释放link指向的节点空间。

编程项目8.1：BinSearchTree类的另一种实现

这个项目说明了折半查找树的数据结构有多个实现。读者可以使用下面描述的方法将一个折半查找树（实际上可以是任意二叉树）存入磁盘，这样以后可以重新得到它的原始结构。

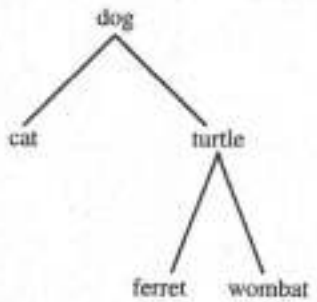
开发折半查找树的数据结构的基于数组的设计和实现。开发的这个类——BinSearchTreeArray——和BinSearchTree类有相同的方法描述，但是使用索引模拟了父亲、左链接和右链接。例如，tree_node可以声明如下：

```
struct tree_node
{
      T item;
      int parent,
          left,
          right;
};//tree_node
```

同理，BinSearchTreeArray类可能包含下面的三个字段：

```
tree_node[] tree;
int header;
unsigned nodeCount;
```

头节点存储在tree[0]中，根节点存储在tree[1]中，因此在tree_node里不需要isHeader字段。NULL指针用-1表示。例如，假设按顺序输入“dog”、“turtle”、“womba-t”、“cat”和“ferret”创建一个折半查找树。那么树的形式如下：



项将按照插入的顺序从索引1开始存放进树的数组。数组的表示是： 350

	item	parent	left	right
0	?	1	4	3
1	dog	0	4	2
2	turtle	1	5	3
3	wombat	2	−1	−1
4	cat	1	−1	−1
5	ferret	2	−1	−1

方法定义和BinSearchTree类中的非常相近，除了如下面形式的表达式

```
header->left
```

被替换成

```
tree[HEADER].left//const int HEADER=0;
```

例如，find的方法可能定义如下：

```
Iterator find (const T& item)
{
    int parent = HEADER;  // const int HEADER = 0;
    int child = tree [HEADER].parent;
    while (child != -1)
    {
        if (!(tree [child].item < item))
        {
            parent = child;
            child = tree [child].left;
        } // item <= tree [child].item
        else
            child = tree [child].right;
```

```
    } // while
    if (parent == HEADER || item < tree [parent].item)
        return end( );
    else
        return parent;
} // find
```

351

这里也将需要一个嵌入的Iterator类。

第9章 AVL 树

正如在第8章中提到的，折半查找树存在的一个很严重的问题是它们可能会严重不平衡。特别是BinSearchTree类中find、insert和erase方法的worstTime(n)和n成线性关系。我们希望避免这些方法的worstTime与n的线性时间花费。大体策略是保持树的高度和n成对数关系。本章介绍了对折半查找树改进后的一种数据结构——AVL树，AVL树是高度总和n成对数关系的折半查找树。

在标准模板库中并没有包括AVL树。但是AVL树中的插入和删除算法比即将在第10章中学习到的红黑树的算法要简单些。标准模板库有四个关联容器类——set、multiset、map和multimap，它们通常是用红黑树实现的。

目标

1) 了解平衡的折半查找树对普通的折半查找树有什么改进。
2) 说明AVL树是平衡的折半查找树。
3) 解释什么是函数对象以及如何使用函数对象。
4) 能够使用方法在AVL树中插入。

9.1 平衡的折半查找树

根据第8章中对find、insert和erase方法的分析，很明显折半查找树在平均情况下是有效率的，但是在最坏情况下它是一个链，和向量、双端队列或链表差不多。有几个数据结构是基于折半查找树且总是平衡的。如果一个折半查找树的高度总是和树中项的数量n成对数关系，就称它是**平衡的**。

三种广为人知的平衡折半查找树的数据结构是AVL树、红黑树和扩展树。9.2节中研究了AVL树，并将在编程项目9.1中做进一步的探讨。第10章将研究红黑树。有关扩展树的信息，有兴趣的读者可以参考Sahni(2000)。这些数据结构都不是标准模板库中的内容。但是在标准模板库的惠普实现中，在定义标准模板库的四个关联容器类中的一个**private**字段时使用了rb_tree（红黑树）类。

9.2节中描述了如何获得高度与n成对数关系的折半查找树。

9.2 旋转

保持一个折半查找树平衡的基本机制是**旋转**：将树绕着某一项进行调整，并保证项需要的顺序。这里将探究旋转中的**螺母**和**螺钉**；在9.3.4节、10.1.3节、10.1.4节和编程项目9.1中将看到它们是如何使用的。

有两种基本的旋转类型。在**左旋转**中，将项移动到它的左子女的位置上，而将项的右子女移动到它的位置上。例如，图9-1显示了绕着项50进行的左旋转。在旋转之前和之后，树都是一个折半查找树。

图9-2显示了另一个围绕项80进行左旋转的例子，它将树的高度从3降低到2。图9-2中一个有趣的特点是，旋转之前90的左子女85最后成了80的右子女。这是所有围绕x进行的左旋转的共同现象：x的右子女的左子树成为x的右子树。在图9-1中也发生了相同的现象，但是50的右子女的左子树为空。

图9-3将图9-2中的旋转推广到一个更广泛的情形：旋转所围绕的项不是树的根项。图9-3说明了所有旋转的另一个方面：所有不在旋转项子树中的项是不受旋转影响的。也就是说，在这两个树中都有：

354

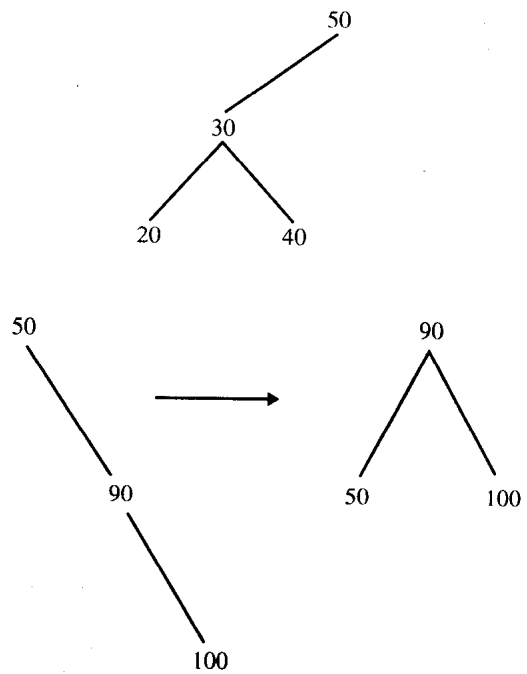


图9-1 围绕50进行左旋转

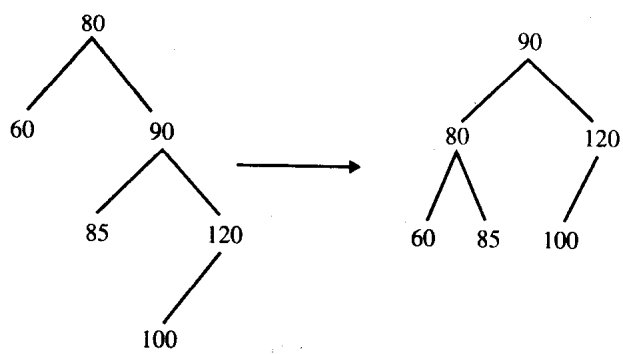


图9-2 围绕80进行左旋转

旋转的代码并不涉及任何项的移动；只是操作指向节点的指针。假设x是指向一个节点的指针，y是指向x的右子女的指针。围绕x的左旋转基本可以通过两个步骤实现：

```
x->right=y->left;//例如，图9-2中的85
```

y->left=x;

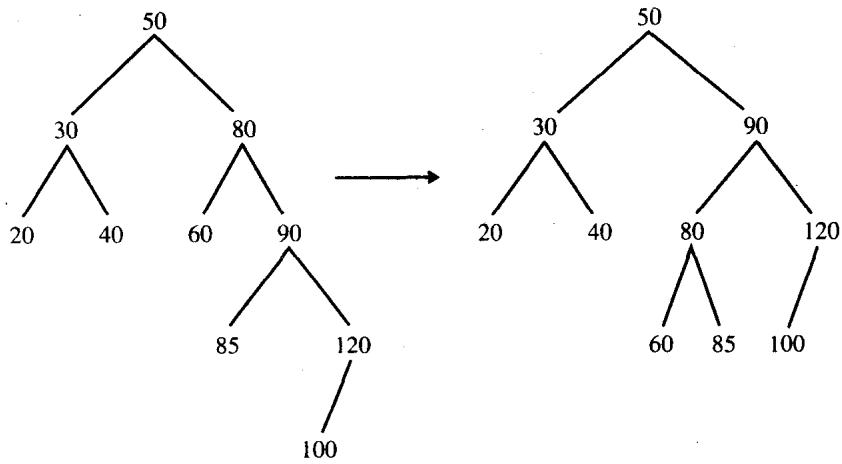


图9-3 围绕图9-2中的80进行左旋转，但是这里的80不是根项

旋转的大部分代码都需要调整旋转所围绕项的父亲。

不幸的是也必须调整parent字段并添加相当多的代码。这里是BinSearchTree类中rotate_left方法的完整定义——回忆第8章中header节点的parent字段指向根节点：

```
// 来自 Cormen (1990)
// 后置条件：执行围绕x的左旋转。
void rotateLeft (tree_node* x)
{
    tree_node* y = x -> right;
    x -> right = y -> left;
    if (y -> left != NULL)
        y -> left -> parent = x;
    y -> parent = x -> parent;
    if (x == header -> parent) // 如果x是根
        header -> parent = y;
    else if (x == x -> parent -> left) // 如果x是一个左子女
        x -> parent -> left = y;
    else
        x -> parent -> right = y;
    y -> left = x;
    x -> parent = y;
}
```

这说明将“惊动”多少个父节点！但是从乐观的角度来说，没有需要移动的项，时间是常数。

右旋转又怎么样呢？图9-4显示了一个简单的例子：围绕100进行右旋转。令x是指向树节点的指针，y是指向x的左子女的树节点的指针。那么围绕x进行的右旋转基本可由两个步骤完成：

x->left=y->right;

```
y->right=x;
```

当然，一旦包括了父节点的调整，那么方法长度将会大量增加，但时间仍然是常数。实际上，如果将rotate_left方法定义中的left和right进行交换，就可以得到rotate_right的定义！并且如同左旋转一样，右旋转之后的树仍然是一个折半查找树。

355
~
356

在迄今看到的所有旋转中，树的高度都是减少1。这没什么可奇怪的；实际上，减少高度正是旋转的动机。但是并非每个旋转都必须减少树的高度。例如，图9-5显示了围绕项50的树节点进行的左旋转，它并没有影响树的高度。

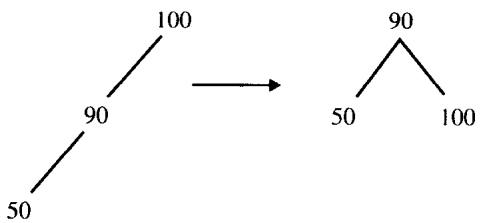


图9-4 围绕100进行的右旋转

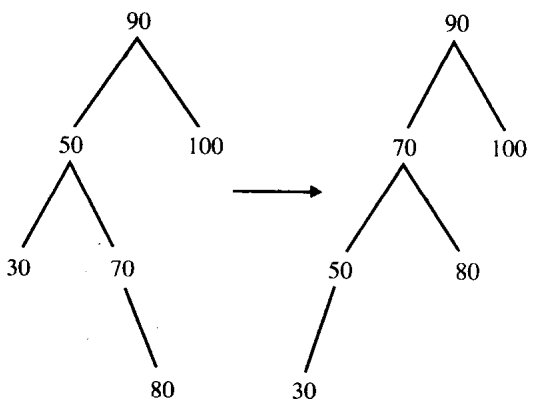


图9-5 围绕50进行的左旋转。旋转后，树的高度仍然是3

图9-5中的左旋转确实没有减少树的高度。但是通过几分钟的测试就可以证明单个旋转是不能减少图9-5左边树的高度的。现在看一下图9-5中右边的树，能否找到一个旋转可减少该树的高度呢？不能围绕70进行右旋转，那只会让我们退回出发点。那么围绕90进行右旋转呢？看！图9-6显示了结果。

图9-5和图9-6中的旋转可以看作一组：围绕90的左子女进行左旋转，随后围绕90进行右旋转，这称作**双旋转**。图9-7显示了另一种双旋转：围绕50的右子女进行右旋转，随后再围绕50进行左旋转。

这里是旋转的主要属性。

在继续深入讨论之前，我们先来看一下旋转的主要特点：

1) 有四种旋转——

a. 左旋转；

b. 右旋转；

357

c. 围绕某项的左子女进行左旋转，随后再围绕该项自身进行右旋转；

d. 围绕某项的右子女进行右旋转，随后再围绕该项自身进行左旋转。

2) 不在旋转所围绕项的子树中的节点是不受旋转影响的。

3) 旋转耗费常数时间。

4) 旋转之前和之后的树都是折半查找树。

5) 左旋转的代码和右旋转的代码是对称的（反之亦然），只需要简单地交换left和right。

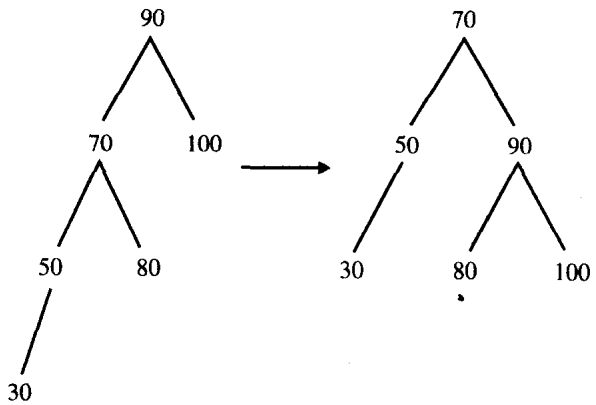


图9-6 围绕90进行右旋转。树的高度从3降到了2

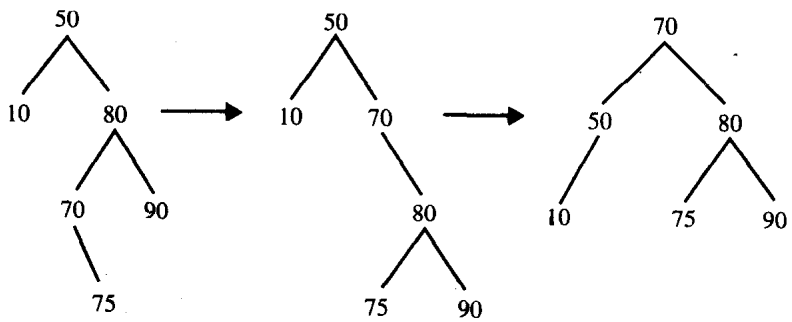


图9-7 双旋转：围绕50的右子女进行右旋转，随后再围绕50进行左旋转

9.3节介绍了一种基于折半查找树的数据结构——AVL树，不过它使用了旋转保持平衡。

9.3 AVL树

AVL树的一个递归定义。

AVL树是一个折半查找树，它或者为空，或者具有下面的两个属性： 358

1) 左子树和右子树的高度之差至多为1。

2) 左右子树都是AVL树。

AVL树是根据它们的开发者——两个俄国数学家，Adel' son-Vel' skii和Landis——命名的（见Adel' son-Vel' skii and Landis, 1962）。图9-8显示了三个AVL树，图9-9显示了三个不是AVL树的折半查找树。

图9-9中第一个树不是AVL树，因为它的左子树高度是1而右子树高度是-1。第二个树不是AVL树，因为它的左子树和右子树都不是AVL树。第三个树也不是AVL树，因为它的左子

树高度是1，而右子树高度是3。

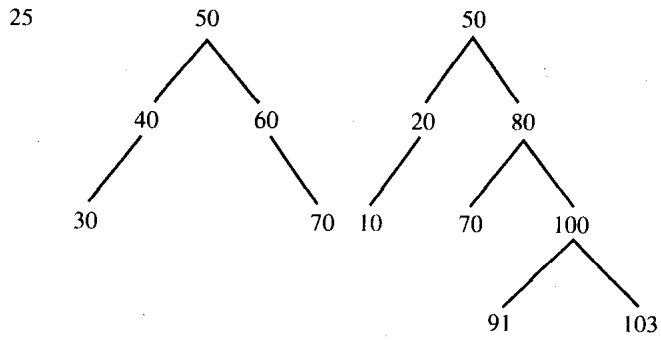


图9-8 三个AVL树

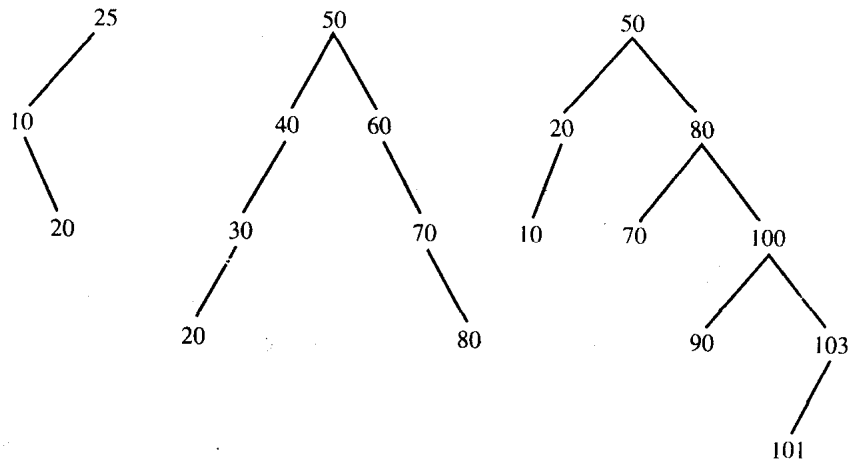


图9-9 三个不是AVL树的折半查找树

9.3.1节中将说明AVL树是一个平衡的折半查找树，也就是说，AVL树的高度总是和n成对数关系的。这和一般的折半查找树——最坏情况下高度和n成线性关系（比如链）——形成鲜明的对比。线性和对数之间的差别是非常大的，例如，假设n=1 000 000 000 000，那么$\log_2 n$
359 小于40。

9.3.1 AVL树的高度

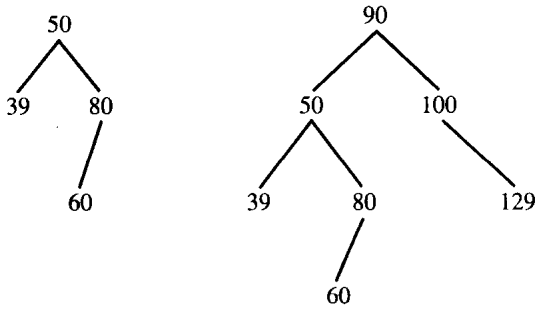
可以证明AVL树的高度和n成对数关系，而且该证明过程中又涉及到了斐波纳契数。

AVL树的高度和树中项的数量n成对数关系。

声明 如果t是一个非空AVL树，那么height(t)和n成对数关系，其中n是t中项的数量。

证明 这里将证明即便是一个AVL树t的n个项的最大高度，也仍然和n成对数关系。如何求出n个项的AVL树可能的最大高度呢？正如Kruse(1987)提出的，重新分析问题有助于得出答案。给定高度h，那么任意一个该高度的AVL树中项的最小数量是多少？

对h=0,1,2,...，令min_h是高度为h的AVL树中项的最小数量。很明显，min_0=1而min_1=2。min_2和min_3的值可以参考下面的AVL树：



一般来说，如果$h1>h2$，那么min_{h1}大于构造高度为$h2$的AVL树需要的项的数量。也就是说，如果$h1>h2$，那么$min_{h1}>min_{h2}$。换句话说，min_h是一个递增函数。

假设t是高度为h的AVL树且包含min_h个项，$h>1$。那么怎样描述t的左、右子树的高度呢？根据高度的定义，这两个子树之一必定有一个的高度是$h-1$。同样根据AVL树的定义，另一个子树的高度必定是$h-1$或$h-2$。实际上，因为t包含了该高度的树中最少的项，所以它的一个子树的高度必定是$h-1$并包含min_{h-1}个项，而另一个子树的高度必定是$h-2$并包含min_{h-2}个项。

树中的项总是比它的左子树和右子树中项的数量多1。因此有

$$min_h = min_{h-1} + min_{h-2} + 1 \quad h是任意>1的整数$$

这个递推关系看起来非常像生成斐波纳契数的公式。术语**斐波纳契树**就是指对应高度上项数量最少的AVL树。根据递推关系以及min_0和min_1的数值，可以对h进行归纳证明： 360

$$min_h = fib(h+3)-1 \quad h是任意非负整数$$

例如，因为min_6=33而且min_7=54，所以包含50个项的AVL树的最大高度是6。

还可以进一步对h进行归纳证明（参阅习题9.5）：

$$fib(h+3)-1 \geqslant (3/2)^h \quad h是任意非负整数$$

将这两个结果合并得到

$$min_h \geqslant (3/2)^h \quad h是任意非负整数$$

以2为底数（可以使用任何底数）取对数，将得到

$$\log_2 min_h \geqslant h\log_2(3/2) \quad h是任意非负整数$$

将这个公式重写成适合大O声明的形式，由于$1/\log_2(1.5)<1.75$，故：

$$h \leqslant 1.75\log_2 min_h \quad h是任意非负整数$$

如果t是高度为h的包含n个项的AVL树，那么一定有$min_h \leqslant n$，因此对任意这样的AVL树，

$$h \leqslant 1.75\log_2 n$$

也就是说，即使在最坏情况下，任何包含n个项的AVL树的高度都是$O(\log n)$。

这意味着任意AVL树的高度都是$O(\log n)$。那么$O(\log n)$是最小上界吗？是的，下面说明原因。根据二叉树定理的第二部分，对任意高度为h包含n个项的二叉树来说，

$$h \geqslant \log_2(n+1)-1$$

因此可以断定，即使在最坏情况下，任何一个包含n个项的AVL树的高度也是和n成对数关系的。

为了更好地说明AVL树和折半查找树的关系，我们将在9.3.3节和9.3.4节中设计并实现AVLTree类。

9.3.2 函数对象

函数对象是类中的一个对象，其中重载了函数调用运算符， **operator()**。

在开始设计和实现AVLTree类之前，还需要介绍一个重要的概念——函数对象。**函数对象**——也称作**函子**，是类中的一个对象，其中重载了函数调用运算符**operator()**。被重载的**operator()**所在的类称作**函数类**或**函子类**。例如，假设函数类MyClass按如下方式重载**operator()**:

361

```
int operator( ) (int i)
{
    return 5 * i;
} // operator( )
```

那么可以编写

```
MyClass d;
cout << d (15);
```

在输出语句中，对象d看起来像一个函数，这也就是术语“函数对象”的由来。

但是函数对象的值是什么呢？回想一下模板的重要性。前面已经看到过一些例子，这些例子表明，模板提供了项类型的灵活性，因此可以在容器类中使用各种各样的项类型。模板也提供了运算符类型的灵活性，这样就可以在容器类中使用各种各样的运算符。下面几段将说明函数对象是如何工作的，以及模板在其中扮演的角色。

BinSearchTree类的一个缺点是项与项之间的比较必须使用**operator<**。这在很多应用中是很好的，但并非对所有的应用都是如此。例如，假设树中的项是考试成绩，并希望将它们按降序存储。或者也可能树中的每一项由雇员的姓名、薪水和部门组成。一个应用可能希望分部门、每个部门内按照字母顺序将雇员存储到树中；而另一个应用又可能希望按照薪水的降序将雇员存储在树中。那么在项的类中就不能重载**operator<**同时为这两个应用服务。

C++允许类的用户为某一具体应用裁剪设计比较关系。这个灵活的机制是模板参数；AVLTree类的定义的开头是:

```
template<class T, class Compare>
class AVLTree
{
    protected:
        Compare compare;
```

compare字段就是一个函数对象的例子。当用户定义一个AVLTree容器时，在定义中会包

含重载**operator**()的类的一个模板变元。例如，下面是前一段中两个雇员应用所对应的定义：

```
AVLTree< Employee, ByDivision< Employee > > avl1;
AVLTree< Employee, ByDecreasingSalary< Employee > > avl2;
```

362

对avl1对象，雇员将根据部门进行比较；而在部门内部将按照字母顺序进行比较。下面是ByDivision类的定义：

```
template<class T>
class ByDivision
{
    bool operator( ) (const T& x, const T& y) const
    {
        if (x.division < y.division)
            return true;
        if (x.division == y.division && x.name < y.name)
            return true;
        return false;
    } // 重载()
} // 类ByDivision
```

这是一个奇怪的类：没有字段，而且只有一个方法！当然，编译器将提供一个缺省构造器，但是这个构造器什么也不做，因为类中没有字段。在AVLTree类中，函数对象compare将替换**operator**<。例如，在BinSearchTree类的insert方法中有

```
if(item<child->item)
```

AVLTree类中相应的行是

```
if(compare(item,child->item))
```

这里的函数对象compare正调用它的函数调用运算符**operator**()。结果依赖于调用insert方法的AVLTree对象。如果它是avl1，那么当item的部门小于child->item的部门时；或者部门相同，而item的姓名按照字母顺序排在child->item的姓名之后时就返回**true**。

现在已经知道了ByDivision类，毫无疑问也可以得到ByDecreasingSalary类：

```
template<class T>
class ByDecreasingSalary
{
    bool operator( ) (const T& x, const T& y) const
    {
        return x.salary > y.salary;
    } // 重载()
} // 类ByDecreasingSalary
```

这时读者可能勉强同意有时函数对象还是值得的。但是如果需要的只不过是一个简单的比较，就像**operator**<所提供的，这会怎样呢？没有问题。在文件<function>中，有几个预定义的类，其中重载了**operator**()进行简单的比较。例如，函数类less按如下方式重载了**operator**():

363

```
bool operator( ) (const T& x, const T& y) const { return x < y; }
```

因此，如果只是想在AVLTree对象中用**operator**<比较**int**项，可以定义

```
AVLTree< int, less<int> > myTree;
```

函数对象是标准模板库的关联容器类所必须的，而且不仅仅能用在专门的比较上。实验21提供了更多函数对象的细节，特别是说明了它们比单纯的函数有更强大的功能。

实验21：更多的函数对象的细节　　（所有实验都是可选的）

9.3.3 AVLTree类

除了9.3.2节中提到的函数对象特点之外，AVLTree类和BinSearchTree类是非常相似的。基本的方法——size、find、insert和erase，它们的方法头是相同的。方法接口中惟一的不同是find、insert和erase方法的worstTime(*n*)和*n*成对数关系。这是因为AVL树的高度总是和*n*成对数关系。

AVLTree类将包括和BinSearchTree类相同的字段以及函数对象compare:

```
protected:
        Compare compare;
        tree_node* header;
        unsigned node_count;
```

tree_node类是被扩展过的。除了BinSearchTree类的tree_node**结构**的五个字段（item，parent，left，right，isHeader）之外，AVLTree类的tree_node**结构**又增加了一个额外的字段:

```
char balanceFactor;
```

如果一个节点的balanceFactor字段值为‘L’，那么它的左子树高度比右子树高度大1。

如果一个节点的balanceFactor字段值为‘=’，那么它的左右子树高度相同。如果balanceFactor字段值为‘L’，那么它的左子树高度比右子树高度大1。如果balanceFactor字段值为‘R’，就表示它的右子树高度比左子树高度大1。

除了find、insert和erase之外，AVLTree类的方法定义和BinSearchTree类中的方法定义是完全相同的。AVLTree类的find方法使用了函数对象compare取代了**operator**<。下面是它的
364 定义:

```
Iterator find (const T& item)
{
    Link parent = header;
    Link child = header -> parent;
    while (child != NULL)
    {
        if (!compare (child -> item, item))
        {
            parent = child;
```

```
                child = child -> left;
            } // item "<=" child -> item
            else
                child = child -> right;
        } // while

        if (parent == header || compare (item, parent -> item))
            return end( );
        else
            return parent;
    } // find
```

如果AVLTree类实例的第2个模板变元中有less，那么find的这个定义就是和BinSearchTree类中的定义等价的。例如，

```
AVLTree<string,less <string> >words;
```

insert方法的定义比BinSearchTree类中的定义要稍微复杂些。大量的详细资料说明保持对数高度需要很高的代价。erase方法的定义留到编程项目9.1中。

下面是insert方法的方法接口：

```
//后置条件: item被插入到这个AVLTree中，并返回位于这个新插入项的迭代器。
//          worstTime(n)是O(logn)。
Iterator insert(const T& item);
```

当进行插入时，变量ancestor保存了平衡因子是‘L’或‘R’的、被插入项的最近的祖先(的指针)。

insert方法的实现是基于Horowitz等的研究的（1965）。AVLTree类中的insert方法很像BinSearchTree类中的insert方法。但是当从根到插入点沿着树下行时，需要明了balanceFactor值为‘L’或‘R’的插入节点的最近的祖先。将这样的节点称作ancestor。例如，如果将60插入到图9-10中的AVLTree中，那么ancestor就是项为80的节点。 365

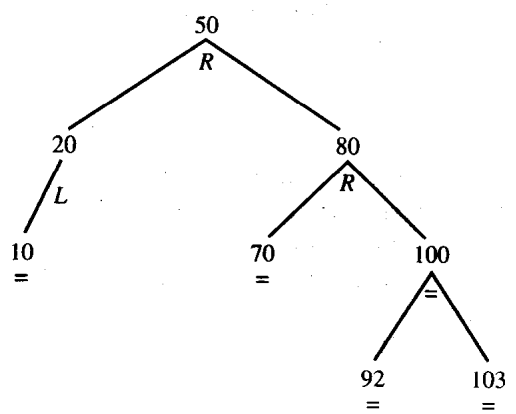


图9-10 包含每个节点的相关的平衡因子的AVL树

在项按照BinSearchTree模式被插入到AVLTree对象中之后，调用一个修正方法处理旋转以及balanceFactor字段的调整。下面是insert方法的定义：

```
Iterator insert (const T& item)
{
    if (header -> parent == NULL)
    {
        insertItem (item, header, header -> parent);
        header -> left = header -> parent;
        header -> right = header -> parent;
        return header -> parent;
    } // 在树的根部进行插入
    else
    {
        Link parent = header,
             child = header -> parent,
             ancestor = NULL;

        while (child != NULL)
        {
            parent = child;
            if (child -> balanceFactor != '=')
                ancestor = child;
            if (compare (item, child -> item))
                child = child -> left;
            else
                child = child -> right;

        } // while
        if (compare (item, parent -> item))
        {
            insertItem (item, parent, parent -> left);
            fixAfterInsertion (ancestor, parent -> left);
            if (header -> left == parent)
                header -> left = parent -> left; // 最左边的子女
            return parent -> left;
        } // 在parent的左边进行插入
        else
        {
            insertItem (item, parent, parent -> right);
            fixAfterInsertion (ancestor, parent -> right);
            if (header -> right == parent)
                header -> right = parent -> right; // 最右边的子女
            return parent -> right;
        } // 在parent的右边进行插入
    } // 树非空
} // insert
```

366

insertItem方法的定义和BinSearchTree类中insertItem方法的定义惟一的区别就是将balanceFactor字段的值设置成‘=’。fixAfterInsertion方法十分复杂，下面专门用一节讲述这个方法。

9.3.4 fixAfterInsertion方法

下面是fixAfterInsertion方法的方法接口（回忆一下，Link只是tree_node*的简化）：

```
//后置条件：如果需要的话，就恢复AVL属性，可以通过在被插入的tree_node和离它
//          最近的balanceFactor是'L'或'R'的祖先之间进行旋转以及平衡因子的
//          调整来实现。
void fixAfterInsertion(Link ancestor, Link inserted);
```

fixAfterInsertion方法的定义的开头是root和item的定义：

```
Link root=header->parent;
T item=inserted->item;
```

方法的剩余部分由六种情况组成。选择哪一个情况完成方法依赖于ancestor节点的balanceFator值以及项被插入的位置。 367

情况1

ancestor为NULL；也就是说，被插入节点的每个祖先的balanceFactor值为‘=’。调整这些祖先的balanceFactor值然后结束。例如，图9-11显示了这个情况之前和之后的树。在图9-11以及其他情况的所有图中，都假定函数对象compare是less类的一个实例，因此compare(a, b)可以解释成a<b。

它的结果是这六种情况的每一个都包含了从插入节点的父亲直到插入点的某些祖先（但不包括它）的路径上的平衡因子调整。因此在这种情况里采取的行动是调整根的平衡因子，然后调用adjustPath(root, inserted)。下面是方法的接口：

```
//后置条件：如果需要，就调整所有从inserted节点（不包括它）到to节点
//          （不包括它）之间的节点的平衡因子。
void adjustPath(Link to,Link inserted);
```

adjustPath方法在需要时调整两个给定节点（不包括本身）之间路径上的每个节点的平衡因子。

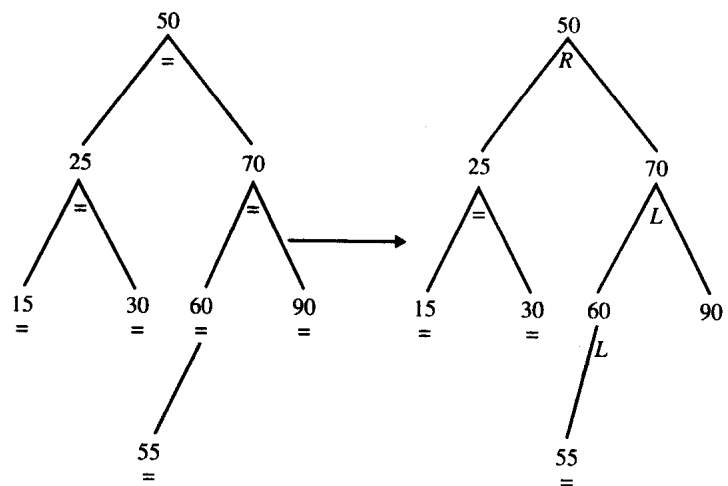


图9-11 左边是调用fixAfterInsertion之前的AVLTree对象；插入的项是55，它的全部祖先的balanceFactor值都是‘=’。右边是调整平衡因子之后的AVLTree对象

adjustPath方法从inserted节点沿着树向后倒退。路径上的每个平衡因子当前值都是‘=’。通过比较插入项和给定节点的项，可以求出路径上任意给定节点的平衡因子。具体地说，如
368 果插入项小于给定项，那么新的平衡因子应该是‘L’，否则就是‘R’。下面是定义：

```
void adjustPath (Link to, Link inserted)
{
    T item = inserted -> item;

    Link temp = inserted -> parent;
    while (temp != to)
    {
        if (compare (item, temp -> item))
            temp -> balanceFactor = 'L';
        else
            temp -> balanceFactor = 'R';
        temp = temp -> parent;
    } // while
} //  方法  adjustPath
```

因为一个AVL树的高度总是和n成对数关系，所以这个方法的循环迭代次数，以及由此得到的worstTime(n)都和n成对数关系。

情况1的全部操作只是调整根的平衡因子，然后调用adjustPath方法：

```
if (ancestor == NULL)
{
    if (compare (item, root -> item))
        root -> balanceFactor = 'L';
    else
        root -> balanceFactor = 'R';
    adjustPath (root, inserted);
} // 情况1：所有祖先的平衡因子都是 '='
```

情况2

ancestor->balanceFactor的值是‘L’，并且在ancestor节点的右子树中进行插入；或者ancestor->balanceFactor的值是‘R’，并且在ancestor节点的左子树中进行插入。举例说明这个情况的应用，在图9-11右边的AVL树中插入28。在这个情况，也就是第一种情况的if语句之后，将ancestor节点的平衡因子设置成‘=’，然后对inserted和ancestor之间的节点进行通常的调整：

```
else if ((ancestor -> balanceFactor == 'L' &&
          !compare (item, ancestor -> item)) ||
          (ancestor -> balanceFactor == 'R' &&
          compare (item, ancestor -> item)))
369 {
    ancestor -> balanceFactor = '=';
    adjustPath (ancestor, inserted);
} // 情况2：在和ancestor的平衡因子相反的子树中进行插入
```

图9-12显示了这种情况之前和之后的树。

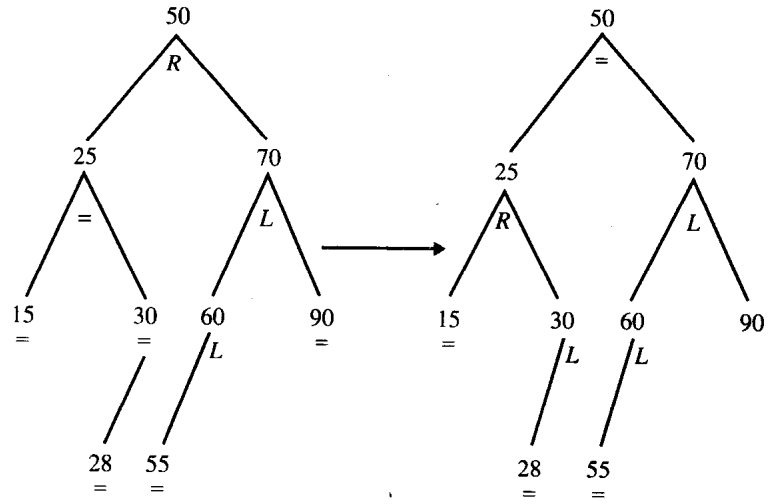


图9-12 左边是刚插入28后的AVL树。其他节点的平衡因子都是插入前的。右边是调整平衡因子之后的同一个AVL树。调整的只是从28到50之间的路径上的平衡因子

剩余的四种情况都需要通过旋转重新平衡树：左旋转，右旋转，左-右旋转以及右-左旋转。在每次这样的旋转之后，主要是通过调用adjustPath方法调整平衡因子。

情况3

ancestor->balanceFactor的值是‘R’，并且被插入节点是在ancestor节点的右子树的右子树之中。在这个情况中，围绕ancestor节点进行了一个左旋转。下面是代码：

```
else if (ancestor -> balanceFactor == 'R' &&
      !compare (item, ancestor -> right -> item))
{
      ancestor -> balanceFactor = '=';
      rotate_left (ancestor);
      adjustPath (ancestor -> parent, inserted);
} // 情况3：在ancestor的右子树的右子树中插入
```

图9-13说明了这种情况。

370

情况4

ancestor->balanceFactor的值是‘L’，并且在ancestor节点的左子树的左子树中进行插入。在这个情况中，围绕ancestor节点进行了一个右旋转：

```
else if (ancestor -> balanceFactor == 'L' &&
      compare (item, ancestor -> left -> item))
{
      ancestor -> balanceFactor = '=';
      rotate_right (ancestor);
      adjustPath (ancestor -> parent, inserted);
} // 在ancestor的左子树的左子树中进行插入
```

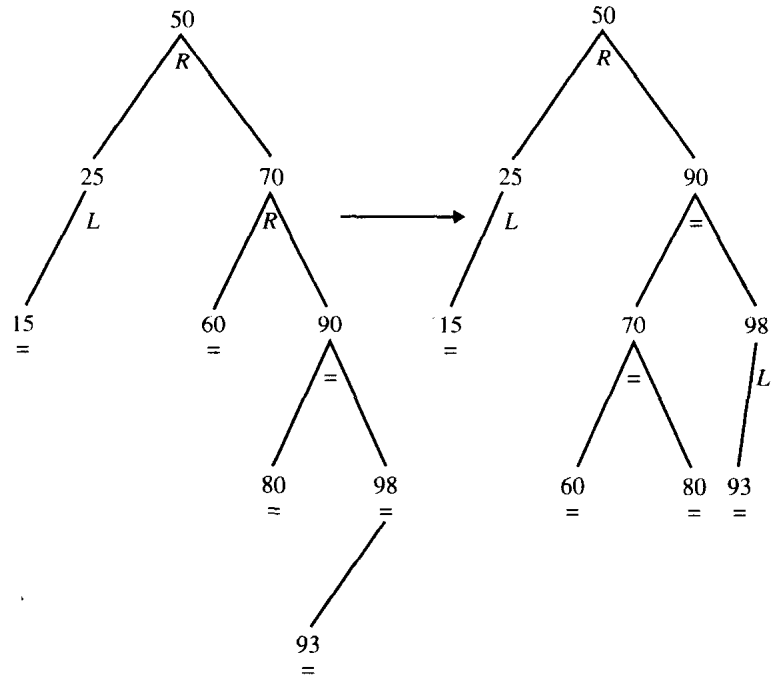


图9-13 左边是插入93后变得不平衡的AVL树。其他节点的平衡因子都是插入前的。在这个情况里，ancestor是包含项70的节点。需要围绕70进行一个左旋转。右边是调整平衡因子后重构的AVL树

371 图9-14说明了这种情况。

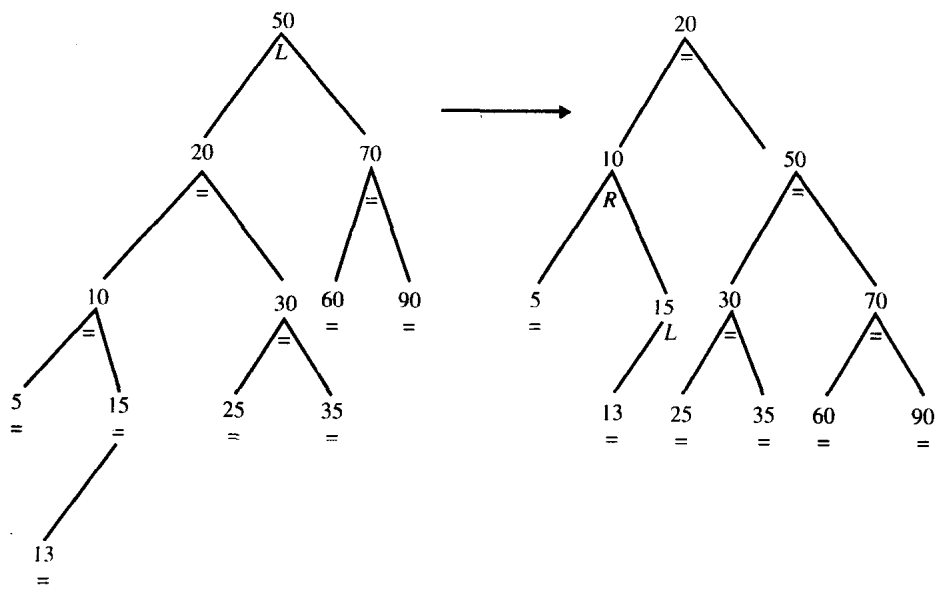


图9-14 左边是插入13后变得不平衡的AVL树。其他节点的平衡因子都是插入前的。在这个情况里，ancestor是包含项50的节点。需要围绕50进行一个右旋转。右边是调整平衡因子后重构的AVL树

情况5

ancestor->balanceFactor的值是‘L’，并且在ancestor节点的左子树的右子树中进行插入。在这个情况中，围绕ancestor节点的左子女进行一个左旋转，然后围绕ancestor节点进行一个右旋转。在adjustLeftRight方法中处理balanceFactor的调整，在情况5的代码之后将讨论该方法。

下面是情况5的代码：

```
else if (ancestor -> balanceFactor == 'L' &&
         !compare (item, ancestor -> left -> item))
{
    rotate_left (ancestor -> left);
    rotate_right (ancestor);
    adjustLeftRight (ancestor, inserted);
} // 在ancestor的左子树的右子树中进行插入
```

adjustLeftRight的方法接口如下：

```
// 后置条件: 在进行左-右旋转之后，调整从inserted（不包括在内）
//           到ancestor的兄弟（包括在内）的路径上所有的
//           平衡因子。
void adjustLeftRight (Link ancestor, Link inserted);
```

372

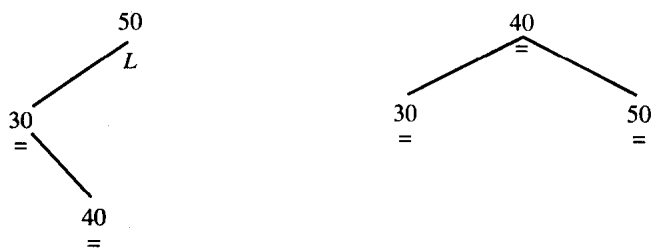


图9-15 左边是插入40后变得不平衡的AVL树。其他节点的平衡因子都是插入前的。在这个情况里，ancestor是包含项50的节点。需要围绕30进行一个左旋转，随后再围绕50进行一个右旋转。右边是调整平衡因子后重构的AVL树

这个方法需要情况5的三个子情况。最简单的子情况（情况5a）是如图9-15所示的最简单的左-右旋转。在这个子情况中，惟一的修改是将ancestor节点的balanceFactor字段设置成‘=’。

另外两个子情况的确定是通过在插入项和ancestor节点旋转后的父亲之间进行比较来进行。图9-16说明了这两个子情况中的前者（情况5b）。因为35小于ancestor节点的父亲（40），35最终在左子树而不是右子树中，因此ancestor节点的balanceFactor字段的值是‘R’。重新平衡的路径是从插入项直到ancestor节点的兄弟（不包括在内）：

373

```
else if (compare (item, ancestor -> parent -> item))
{
    ancestor -> balanceFactor = 'R';
    adjustPath (ancestor -> parent -> left, inserted);
} // 情况5b: item"<"ancestor的父亲的项
```

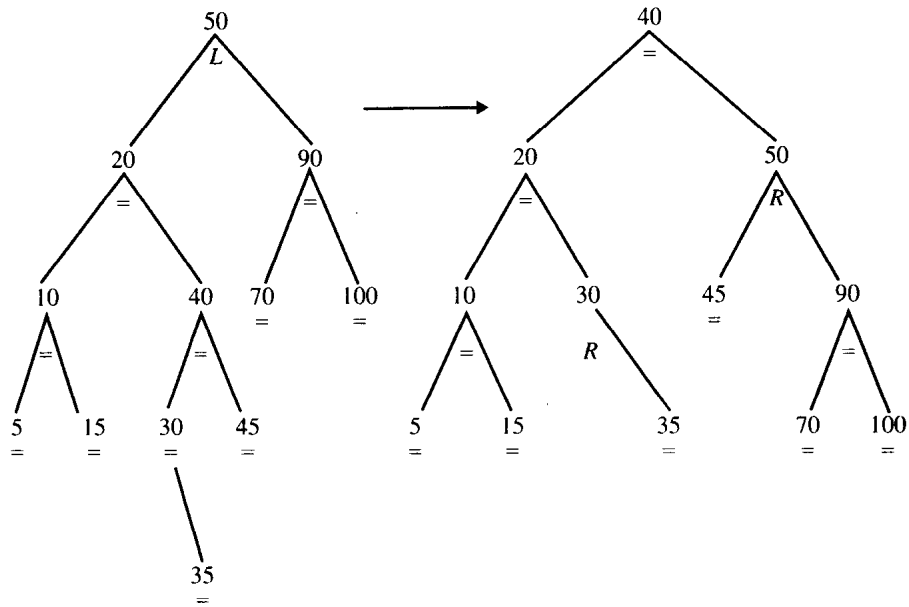


图9-16 左边是插入35后变得不平衡的AVL树。其他节点的平衡因子都是插入前的。在这个情况里，ancestor是包含项50的节点。需要围绕20进行一个左旋转，随后再围绕50进行一个右旋转。右边是调整平衡因子后重构的AVL树

图9-17说明了最后一个子情况（情况5c），当插入项最终位于ancestor节点的父亲的右子树中时。在这个子情况里，42最终在ancestor节点的父亲（40）的右子树中，因此ancestor节点的balanceFactor值是‘=’，并且ancestor节点的兄弟（20）的平衡因子是‘L’。

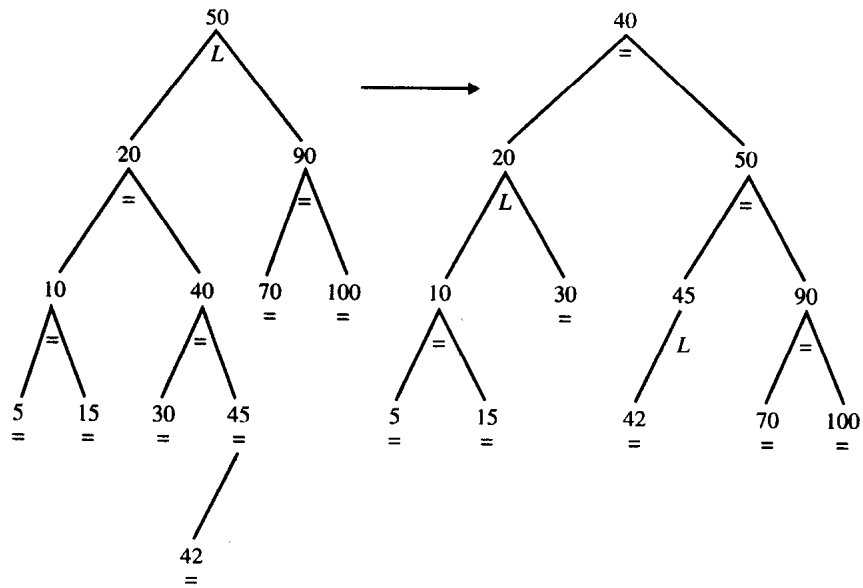


图9-17 左边是插入42后变得不平衡的AVL树。其他节点的平衡因子都是插入前的。在这个情况里，ancestor是包含项50的节点。需要围绕20进行一个左旋转，随后再围绕50进行一个右旋转。右边是调整平衡因子后重构的AVL树

这个子情况的代码是:

```
else
{
    ancestor -> balanceFactor = '=';
    ancestor -> parent -> left -> balanceFactor = 'L';
    adjustPath (ancestor, inserted);
} // 情况5c: item">"ancestor的父亲的项
```

374

情况6

ancestor->balanceFactor的值是‘R’，并且插入的节点是在ancestor节点的右子树的左子树中。在这个情况里，围绕ancestor节点的右子女执行一个右旋转，然后再围绕ancestor节点进行一个左旋转。和情况5一样，情况6也有三个子情况。幸运的是，它们和情况5的三个子情况是对称的。例如，图9-18说明了情况6b，其中插入项最终在ancestor节点的（旋转后的）父亲的右子树中。

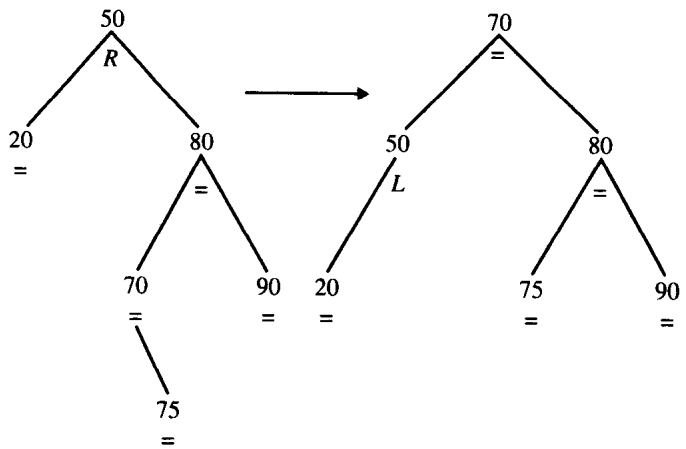


图9-18 左边是插入75后变得不平衡的AVL树。其他节点的平衡因子都是插入前的。在这个情况里，ancestor是包含项50的节点。需要围绕80进行一个右旋转，随后围绕50进行一个左旋转。右边是调整平衡因子后重构的AVL树

因为75最终在70的右子树中，所以50（也就是ancestor节点）的balanceFactor字段的值被设置成‘L’。下面是adjustRightLeft方法的完整定义:

```
void adjustRightLeft (Link ancestor, Link inserted)
{
    T item = inserted -> item;

    if (ancestor -> parent == inserted) // 情况6a
        ancestor -> balanceFactor = '=';
    else if (!compare (item, ancestor -> parent -> item))
    {
        ancestor -> balanceFactor = 'L';
        adjustPath (ancestor -> parent -> right, inserted);
    } // 情况6b: item">="ancestor的父亲的项
```

375

```
            else
            {
                ancestor -> balanceFactor = '=';
                ancestor -> parent -> right -> balanceFactor = 'R';
                adjustPath (ancestor, inserted);
            } // 情况6c: item"<"ancestor的父亲的项
} //  方法  adjustRightLeft
```

最后是包括全部六种情况的fixAfterInsertion方法:

```
void fixAfterInsertion (Link ancestor, Link inserted)
{
    Link root = header -> parent;
    T item = inserted -> item;
    if (ancestor == NULL)
    {
        if (compare (item, root -> item))
            root -> balanceFactor = 'L';
        else
            root -> balanceFactor = 'R';
        adjustPath (root, inserted);
    } // 情况1: 所有祖先的平衡因子都是'='
    else if ((ancestor -> balanceFactor == 'L' &&
            !compare (item, ancestor -> item)) ||
            (ancestor -> balanceFactor == 'R' &&
            compare (item, ancestor -> item)))
    {
        ancestor -> balanceFactor = '=';
        adjustPath (ancestor, inserted);
    } // 情况2: 在和ancestor的平衡因子相反的子树中进行插入
    else if (ancestor -> balanceFactor == 'R' &&
            !compare (item, ancestor -> right -> item))
    {
        ancestor -> balanceFactor = '=';
        rotate_left (ancestor);
        adjustPath (ancestor -> parent, inserted);
    } // 情况3: 在ancestor的右子树的右子树中插入
376 else if (ancestor -> balanceFactor == 'L' &&
            compare (item, ancestor -> left -> item))
    {
        ancestor -> balanceFactor = '=';
        rotate_right (ancestor);
        adjustPath (ancestor -> parent, inserted);
    } // 情况4: 在ancestor的左子树的左子树中进行插入
    else if (ancestor -> balanceFactor == 'L' &&
            !compare (item, ancestor -> left -> item))
    {
        rotate_left (ancestor -> left);
```

```
            rotate_right (ancestor);
            adjustLeftRight (ancestor, inserted);
        } // 情况5: 在ancestor的左子树的右子树中进行插入
        else
        {
            rotate_right (ancestor -> right);
            rotate_left (ancestor);
            adjustRightLeft (ancestor, inserted);
        } // 情况6: 在ancestor的右子树的左子树中进行插入
    } // 方法 fixAfterInsertion
```

还需要证实insert方法的正确性和效率。首先处理正确性。

9.3.5 insert方法的正确性

我们需要证明：如果在调用insert方法之前AVLTree对象是一个AVL树，那么在调用之后调用对象仍然是一个AVL树。旋转保持了折半查找树的属性，这样剩下的工作只是去证实左子树和右子树的高度之差至多是1，而且这两个子树也都是AVL树。insert方法使用了平衡因子，而不是直接去计算高度。需要证明平衡因子能真实地反映高度；也就是说，需要证明如果在调用之前平衡因子是正确的，那么在调用之后平衡因子也仍然是正确的。

实际上，在fixAfterInsertion方法的六种情况中，惟一会增加整个树的高度的就是情况1，即ancestor的值为NULL的情况。在这个情况中，高度会增加的只有那些位于插入项（不包括在内）到根项（包括在内）的路径上的子树。在调用之前，所有这些子树的平衡因子都是‘=’，随后将根据插入项是“小于”或是“大于等于”子树的根项，而把该子树的平衡因子调整成‘L’或‘R’。

在情况2中，假设ancestor节点的平衡因子是‘L’，并且在ancestor节点的右子树中进行插入。那么高度会增加的只有那些位于插入项（不包括在内）到ancestor的项（不包括在内）的路径 377
上的子树，它们的平衡因子也会相应地调整。而且ancestor节点的平衡因子也被设置成‘=’。

图9-19说明了情况3——左旋转情况——的结果。在调用insert之前以ancestor节点为根的子树的高度是不受旋转影响的。除了这个子树之外，所有的高度和平衡因子都是不受影响的。情况4的正确性证明和情况3是对称的。

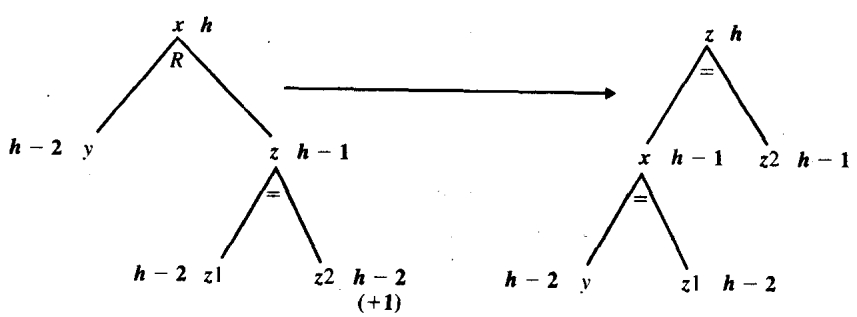


图9-19 在AVLTree容器中进行左旋转。项x代表ancestor节点，在$z2$的子树中进行插入，相应的高度用黑体表示。高度h可能是1，因此y、$z1$和（插入之前）$z2$可能是NULL。如图所示，子树的高度——以及由此推出整个树的高度——是不受旋转影响的

情况5c的一般情形如图9-20所示。由于树的简单性，所以情况5a的正确性是很容易证明的。情况5b的正确性可以用一个和图9-20相似的图证明，情况6的正确性证明和情况5是对称的。

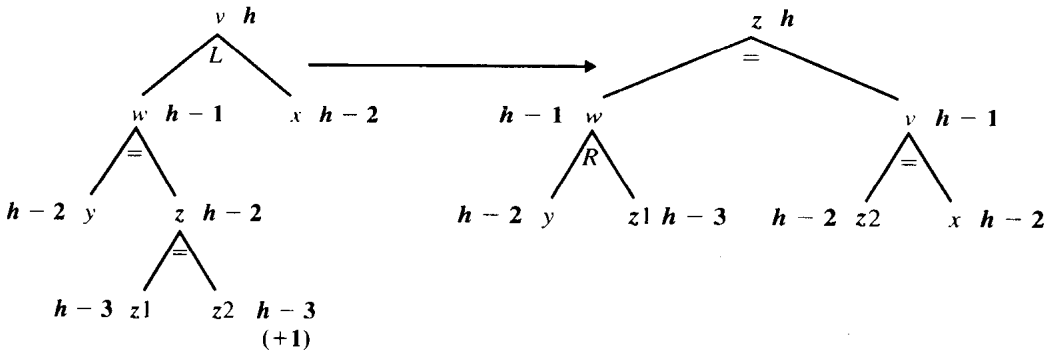


图9-20 情况5c的一般情形：在z的右子树中进行插入。左边的ancestor节点的项是v。先围绕w进行一个左旋转，随后再围绕v进行一个右旋转。高度h可能只是2，或者$z1$和$z2$可能是NULL。如图所示，子树的高度——以及由此推出整个树——是不受旋转影响的

378

运行时测试可以进一步增加对insert方法正确性的信任。构造一个大小为n的AVL树，其中n的值是由终端用户输入的。树中的每一项都是随机产生的，而且是**int**类型。首先，这里用一个包装方法确保折半查找树是一个AVL树：

```
// 后置条件: 如果这个树确实是一个AVLTree就返回真。
//           否则，返回假。
bool isAVLTree( )
{
    return checkAVLTree( header -> parent );
} // 方法isAVLTree
```

下面是包装的方法：

```
// 后置条件: 检验这个树，确保左右子树（如果存在的话）的高度差在1之内，
//           而且这些子树都是AVL树。
bool checkAVLTree (Link root)
{
    if (root == NULL)
        return true;
    else
        if ((abs (height (root -> left) − height (root -> right)) <= 1)
                && (checkAVLTree (root -> left))
                && (checkAVLTree (root -> right)))
            return true;
    return false;
} // checkAVLTree
```

最后是主函数的基本代码：

```
for (i = 0; i < n; i++)
    tree.insert (rand( ));
if (tree.isAVLTree( ))
    cout << "success" << endl;
else
    cout << "failure" << endl;
cout << "height = " << tree.height( ) << endl;
```

当输入10 000作为n时，结果是成功的，也就是说该树的确是一个AVL树。

这个测试有助于分析insert方法的性能。这个树的高度是15，和10 000个项的AVL树的最小高度12.29（$=\log_2 10001-1$）相差不远。而且15小于随机产生的10 000个项的随机产生的BinSearchTree对象的平均高度的一半（参阅实验20）。但是在平均情况下，BinSearchTree插入只比AVLTree插入多花费一点点时间！为什么？因为BinSearchTree对象是低维护容器。对 379
每个AVLTree对象，有一个保险策略能保证树的高度总是和n成对数关系。这个策略能让你平稳地进行插入、删除和查找，使它们的worstTime(n)和n成对数关系。该策略的代价是在插入和删除中付出额外的时间来维护AVL树的属性。

包含各种情况和子情况的erase方法的定义是编程项目9.1的目标。9.4节提供了AVL树的一个应用：检查一个文档中的单词是否出现在字典中。

9.4 AVL树的应用：一个简单的拼写检查器

现代字处理器最有用的特点之一就是拼写检查，就是在文档中扫描可能的拼写错误。这里说“可能的”拼写错误是因为文档中可能包含合法但不在字典中的单词。例如，在键入本章中使用的单词“iterator”和“postorder”时，这些单词就被看作是字处理器所找不到的。

完整的问题是：给出dictionaryFile中的一个字典，以及由用户提供名称的文件中的一个文档，输出在文档中而在字典里找不到的所有的单词。下面进行一些简单化的假设：

1) 字典只由小写单词组成。

2) 文档中的每个单词只由字母组成——其中可能有一些或全部是大写的。

3) 文档中的每个单词后面跟着0个或更多的标点符号，随后是任意数量的空白和行尾标志。

4) 字典文件是按照字母顺序排列的并且将被装入内存。文档文件不必按照字母顺序排列，如果不进行复制的话，它也将被装入内存。

下面是一个小字典文件、小文档文件的内容，以及出现在小文档文件中而小字典文件中却没有的单词：

```
// 字典文件:
a
all
and
be
done
is
more
said
```

```
than
when
where
// 文档文件:
When all is sed and done,
more is said than done.
// 可能拼写错的单词:
sed
```

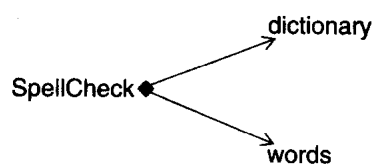
为了解决这个问题，创建一个包含下列方法的SpellCheck类:

```
//后置条件: 读入字典文件中的单词。worstTime(n)是O(n log n)，其中n是
//          字典文件中单词的数量。
void readDictionaryFile();

//后置条件: 读入文档文件中的单词。worstTime(k)是O(k log k)，其中k是
//          文档文件中单词的数量。
void readDocumentFile();

//后置条件: 输出在文档中而不在字典中的每个单词。worstTime(k,n)是
//          O(klogn)，其中k是文档文件中的单词数量，n是字典文件中的单词数量。
void compare();
```

这里仅有的字段是用来保存字典文件中单词的dictionary，以及保存文档文件中特有的单词的words——因为存储一个单词的多个拷贝是毫无意义的。这两个字段都是AVLTree对象，其中每个项都是string并按照字母顺序进行string的比较:

```
protected:
    AVLTree<string, less<string> > dictionary,
                                   words;
```

依赖关系图如下:



下面是readDictionaryFile()方法的直接定义:

```
void readDictionaryFile( ) {
    const string DICTIONARY_FILE = "dictfile.dat";
    fstream dictionaryFile;
    string word;
    dictionaryFile.open (DICTIONARY_FILE.c_str( ), ios::in);
    while (dictionaryFile >> word)
        dictionary.insert (word);
} // readDictionaryFile
```

这个方法将耗费多长时间呢？假设在dictionaryFile中有n个单词，那么**while**循环将被执行n次。每次迭代向dictionary进行一次插入。对AVLTree对象dictionary中的每次插入，worstTime(n)都和n成对数关系，因此readDictionaryFile方法的worstTime(n)是$O(n\log n)$，并且这就是最小上界。

readDocumentFile的定义只是稍微复杂些。读入文档文件的名字，然后再读入文件。将读入的每个单词转换成小写，去掉尾部的标点符号（如果有的话），并将其插入words（除非该单词已经在words中）。下面是完整的定义：

```
void readDocumentFile( )
{
    const string DOCUMENT_FILE_PROMPT =
        "Please enter the name of the document file: ";
    fstream documentFile;
    string documentFileName,
        word;

    cout << endl << DOCUMENT_FILE_PROMPT;
    cin >> documentFileName;
    documentFile.open (documentFileName.c_str( ), ios::in);
    while (documentFile >> word)
    {
        // 将单词转换成小写:
        string temp;
        for (unsigned i = 0; i < word.length( ); i++)
            temp += (char)tolower(word [i]);
        word = temp;

        // 去掉单词尾部的标点符号:
        while (!isalpha (word [word.length( ) -1]))
            word.erase(word.length( ) - 1);

        // 将单词插入words除非它已经在words中:
        if (words.find (word) == words.end( ))
            words.insert (word);
    } // 当documentFile中有更多的单词
} // readDocumentFile
```

382

完成readDocumentFile方法需要花费的时间是很容易估算的。从文件中读入k个单词的worstTime(k)和k成线性关系。向AVLTree对象words中插入每一个单词的worstTime(k)是和k成对数关系的。因此，对于readDocumentFile方法，worstTime(k)是$O(k\log k)$，而且这也是最小上界。

最后，也是最容易的，是compare方法的定义：

```
void compare( )
{
    const string MISSPELLED =
        "Here are the possibly misspelled words:";

    AVLTree< string, less< string > >::Iterator itr;
```

```
        cout << endl << MISSPELLED << endl;
        for (itr = words.begin( ); itr != words.end( ); itr++)
            if (dictionary.find (*itr) == dictionary.end( ))
                cout << *itr << endl;
    } // compare
```

迭代通过words中的k个单词的worstTime(k)和k成线性关系，而用find方法在dictionary的n个单词中进行查找的worstTime(n)和n成对数关系。因此compare的worstTime(k,n)是$O(k\log n)$，而且这是最小上界。编程项目9.2改进了这个应用。

在第13章中将遇到另一个容器类——hash_set类，它也不属于标准模板库。在这个类里，插入、删除和查找的平均时间基本上是常数。因此可以令dictionary和words字段是hash_set对象，然后重做这个问题。不需要再做任何其他的改动！对该版本的拼写检查项目，readDictionaryFile方法的averageTime(n)将和n成线性关系，readDocumentFile方法的averageTime(k)将和k成线性关系。而compare方法的averageTime(k,n)将和k成线性关系。例如，在hash_set版本的拼写检查中，compare方法的worstTime(k,n)将是$O(kn)$。

总结

本章着眼于改进BinSearchTree类，确保树在每次插入或删除之后是**平衡的**，也就是说，它的高度和n成对数关系。采用旋转进行重新平衡。**旋转**是围绕某一项进行的树的调整，它使项需要的顺序保持不变。

383 **AVL树**是一个折半查找树，它或者为空，或者具备下面两个属性：

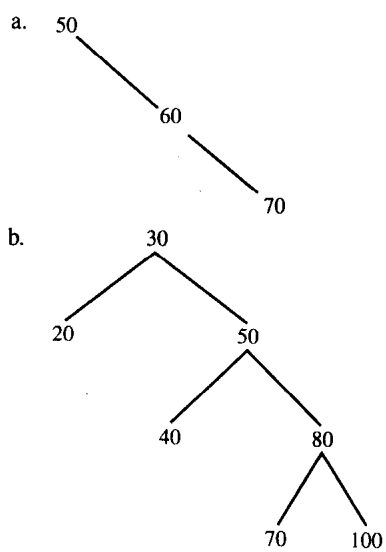
1) 左、右子树的高度之差至多为1。

2) 左、右子树都是AVL树。

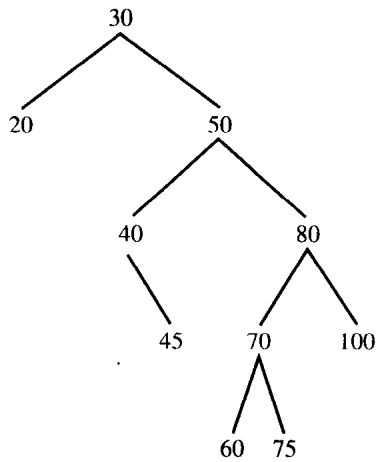
AVL树的高度总是和树中项的数量n成对数关系。

习题

9.1 在下面的每个折半查找树中围绕50进行左旋转。



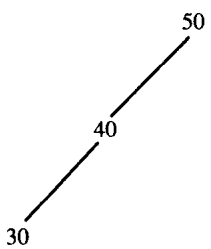
c.



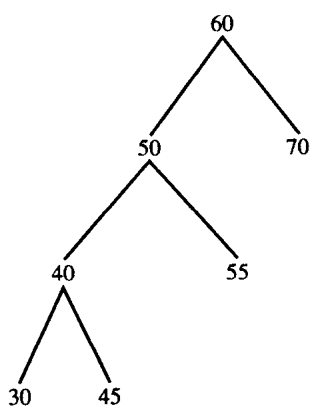
384

9.2 在下面的每个折半查找树中围绕50进行右旋转。

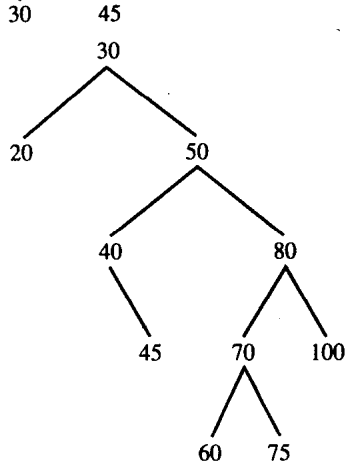
a.



b.

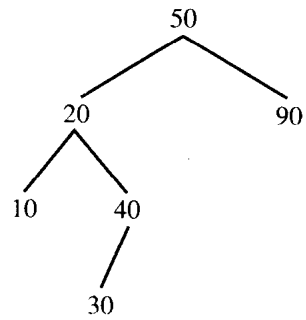


c.

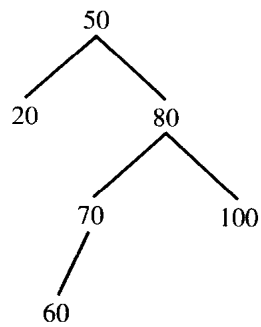


385

9.3 在下面的折半查找树中执行双旋转（围绕20进行左旋转，然后再围绕50进行右旋转），将树的高度降低成2。



9.4 在下面的折半查找树中执行双旋转，将树的高度降低成2。



9.5 证明对任意非负整数h，$fib(h+3)-1 \geq (3/2)^h$

提示 对$h>1$，

$(3/2)^{h-1}+(3/2)^{h-2} = (3/2)^{h-2}(3/2+1) > (3/2)^{h-2}(9/4)$

9.6 假设将max_h定义为一个高度为h的AVL树中项的最大数量。

a. 计算max_3。

b. 对任意$h \geq 0$，求出max_h的公式。

提示 使用第8章的二叉树定理的第2部分。

c. 包含100个项的AVL树的最大高度是多少?

9.7 证明包含32个项的任意AVL树的高度都恰好是5。

386 **提示** 计算max_4和min_6。

9.8 在AVLTree类中，开发一个worstTime(n)是$O(\log n)$的height()方法。

提示 树节点的平衡因子指示了它的哪一个子树高度更大一些。

9.9 AVLTree类中的find方法的worstTime(n)和n成对数关系。BinSearchTree类中的find方法的worstTime(n)和n成线性关系。但是这两个方法的定义是相同的（除了用函数对象compare替换了**operator<**）！解释原因。

387

编程项目9.1：AVLTree类的erase方法

定义AVLTree类中的erase方法。下面是方法接口：

```
//前置条件：itr位于这个AVLTree的某一项上。
//后置条件：从这个AVLTree中删除itr位置上的项。worstTime是O(logn)。
//          （在这次调用之前位于*itr之后的所有迭代器都将失效）amortizedTime(n)
//          是常数，因此averageTime(n)是常数。
void erase(Iterator itr)
```

提示 在执行一个BinSearchTree方式的删除之后，令ancestor为实际上被删除节点的父亲。（如果*itr有两个子女，那么实际上被删除的节点中保存了*itr的后任，因此ancestor节点将是这个后任的父亲。）循环直到这个树成为包含适当平衡因子的AVL树。在循环中，假设删除的项位于ancestor节点的右子树中（对称的分析可以处理左子树）。那么根据ancestor节点的平衡因子是‘=’，‘L’或‘R’可分三个子情况。在全部这三个情况中都必须调整ancestor节点的平衡因子。对‘=’子情况，循环终止。对‘R’子情况，用(*ancestor).parent节点替换ancestor节点并继续循环。对‘L’子情况，根据(*ancestor).left节点的平衡因子是‘=’，‘L’，‘R’又分成三种子情况。而且‘R’子-子情况又包含了三个子-子-子情况！ 388

编程项目9.2：改进的SpellChecker项目

修改拼写检查项目。如果文档单词x不在字典中而单词y在字典中，且x和y的差别或者是相邻字母的调换，或者是单个字母的差别，那么y应当作为x的一个选择。例如，假设文档单词是“asteriks”而字典包含了“asterisk”。通过调换“asteriks”中相邻的字母“s”和“k”，可以得到“asterisk”。因此“asterisk”应当看作是一种选择。同理，如果文档单词是“seperate”或“seprate”，而字典单词是“separate”，那么“separate”也将作为这两种情况的选择。

下面是两个系统测试的字典单词：

```
a
algorithms
asterisk
coat
equals
he
pied
pile
plus
programs
separate
structures
wore
```

下面是文档文件doc1.dat:

She woar a pide coat.

下面是另一文档文件doc2.dat:

Alogrithms plus Data Structures equals Pograms

系统测试1（输入用黑体表示）

In the input line, please enter the name of the document file.
doc1.dat

Possible misspellings	Possible alternatives
pide	pied, pile
she	he
woar	

389

系统测试2（输入用黑体表示）

In the input line, please enter the name of the document file.
doc2.dat

Possible misspellings	Possible alternatives
alogrithms	algorithms
data	
pograms	programs

390

第10章 红 黑 树

本章介绍了另一种平衡折半查找树——红黑树。红黑树的高度限制不像AVL树那么严格，但也仍然是和n成对数关系的。在红黑树中进行插入和删除的算法比在AVL树中进行插入和删除的算法要稍微难一些。研究红黑树的主要优点在于它们是惠普实现中标准模板库的关联容器类——set、multiset、map和multimap——的基础。本章中也将探讨这些关联容器类。

目标

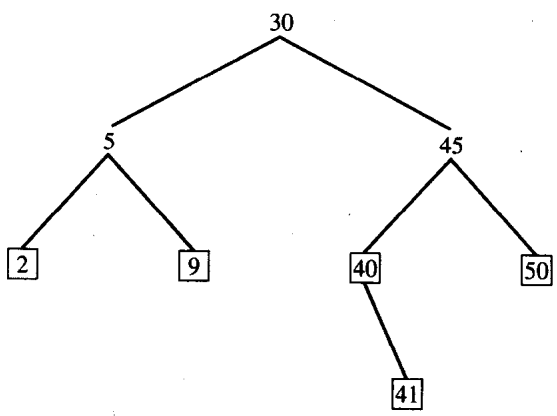
1) 解释红黑树是平衡折半查找树的原因。
2) 能够理解红黑树中的插入或删除方法。
3) 理解集合与多集合、映射与多映射之间的区别。
4) 比较关联数组和普通数组。

391

10.1 红黑树

基本上红黑树就是一个折半查找树，树中的每个节点都采用了一种彩色约定。具体地说，根据下面马上要给出的规则为树的每一项关联一个红颜色或一个黑颜色。规则之一涉及到路径。回忆在第8章中，如果项A是项B的一个祖先，那么A到B的**路径**是从A开始到B结束的项的序列，其中序列中的每一项都是下一项的父亲。

我们将重点关注从根到没有子女或只有1个子女的项之间的路径[⊖]。例如，在下面的树中有五条从根到没有子女或一个子女的项（方框中的）之间的路径。



注意一条路径是到有一个子女的项40的。因此刚才描述的路径不一定必须要到达树叶。

每个红黑树都必须满足红色规则和路径规则。

⊖ 可以等价地考虑从根项到一个空子树的路径，因为一个树叶有两个空子树，而有一个子女的项也有一个空子树。当采取这种方法时，就扩展折半查找树，为每个这样的空子树加入一种特殊的项——占位树叶。

红黑树是一个折半查找树，它或者为空，或者根项着黑色，而其他的每个项着红色或黑色，并满足下面的属性：

红色规则：如果某项着红色，那么它的父亲必须是黑色的。

路径规则：从根项到没有子女或**有一个子女**的项的所有路径上的黑色项的数量必须是相同的。

例如，图10-1显示了一个红黑树，其中的项是整数。观察一下，这是一个有着黑色根的折半查找树。图中红色的项均没有红色的父节点，这满足了红色规则。同样，在根到没有子女或有一个子女的项的五条路径上各有两个黑色项，因此也满足了路径规则。换句话说，该
392 树是一个红黑树。

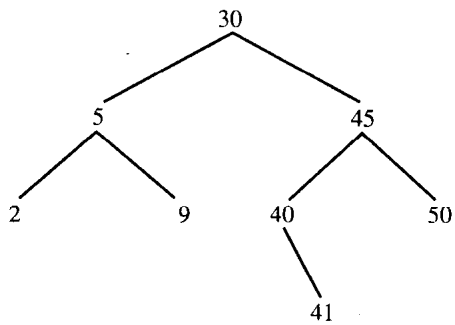


图10-1 包含8个项的红黑树

即使从根到树叶的每条路径上包含相同数量的黑色项，也可能不满足路径规则！

图10-2中的树就不是一个红黑树，即使它满足了红色规则而且从根到树叶的每条路径上都包含相同数量的黑色项。它违背了路径规则，因为从50到80（仅有一个子女的项）的路径上只有一个黑色项，而从50到20的路径上有两个黑色项，从50到100的路径也是如此。

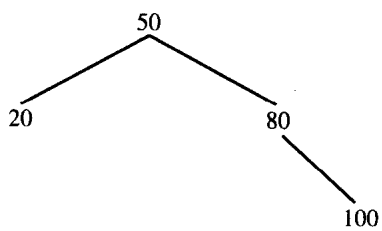


图10-2 一个不是红黑树的折半查找树

同样，图10-3中的树也不是一个红黑树。因为它违背了路径规则，例如，从70到40（仅有一个子女的项）的路径上有三个黑色项，但是从70到110的路径上有四个黑色项。该树是严重不平衡的：任何只有两个树叶的树的高度必定和n成线性关系。

图10-1中的红黑树是相当均匀平衡的，但并非每个红黑树都具备这样的特性。例如，图10-4显示了一个左边下垂的红黑树。很容易证明它是黑色根的折半查找树，并且满足了红色定理。对路径定理，在从根到没有子女或有一个子女的项的每条路径上都恰好有两个黑色项。也就是说，这个树是一个红黑树。但是一个红黑树不平衡的程度是有限的。例如，不能在图10-4的项10下再悬挂任何项。如果添加一个红色项，那么将不再满足红色定理。如果添加一个黑色项，那么路径定理又将不成立。

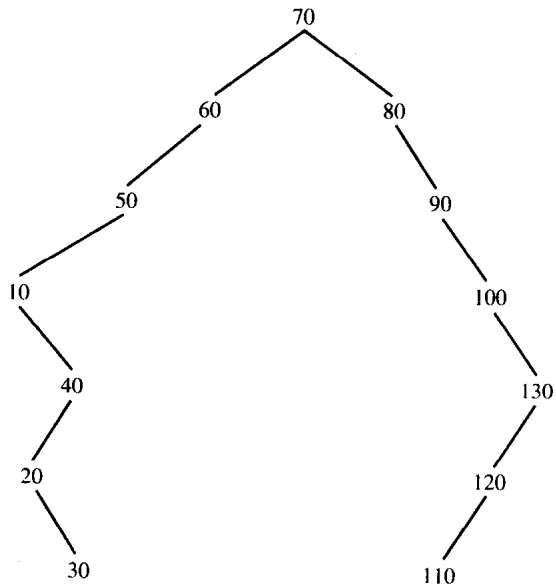


图10-3 一个不是红黑树的折半查找树

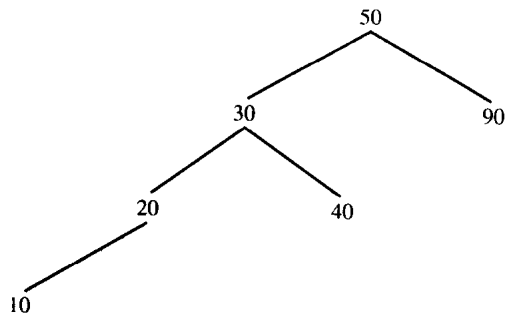


图10-4 一个不是“均匀”平衡的红黑树

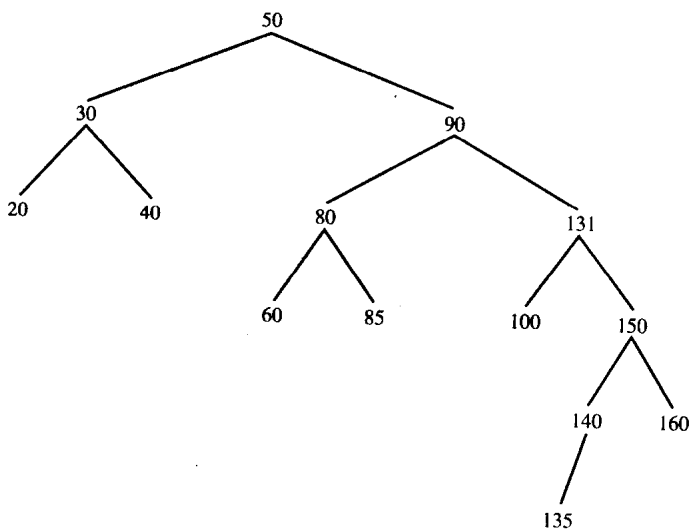


图10-5 包含14个项且具备最大高度5的红黑树

如果一个红黑树是完全的，除了最低层是红色的树叶而其他所有的项都是黑色的，那么
393 树的高度将是最小的，近似为$\log_2 n$。为了求出对应于给定n的最大高度，应当在某条路径上包含尽可能多的红色项，而其他所有项都是黑色的。例如，图10-4显示了一个这样的树，图10-5包含了另外一个。包含全部红色项的路径长度将是没有红色项的路径长度的两倍。这些树使我们假设：红黑树的最大高度小于$2\log_2 n$。在10.1.1节中将证明这个假设。

10.1.1 红黑树的高度

从红黑树的几乎所有非叶节点都有两个子女来说，它是相当浓密的。实际上，如果某个
394 项只有一个子女，那么这个项必定是黑色的，而它的子女一定是一个红色的树叶。这种浓密性使我们相信红黑树是平衡的，也就是说，即使在最坏情况下，它的高度也是和n成对数关系的。对比一下折半查找树，它在最坏情况下的高度和n成线性关系！为了证明红黑树的高度总是和n成对数关系，需要几个初步结论。

1. *声明1*

令y是红黑树的一个子树的根，从y到它的没有子女或有一个子女的子孙间的路径上的黑色项的数量是相同的。

要证明这个声明，先假设x是一个红黑树的根，y是一个子树的根。令b0是x（包括在内）到y（不包括在内）之间的黑色项的数量，令b1是y（包括在内）到它的任意一个没有子女或有一个子女的子孙（包括在内）之间的黑色项的数量，并令b2为y（包括在内）到它的任意另一个没有子女或有一个子女的子孙（包括在内）之间的黑色项的数量。图10-6描述了这个状况。

例如，返回图10-5，假设x是50，y是131，而y的两个子孙分别是100和135。那么b0=1，代表从50到90的路径上黑色项的数量；131是不计算在b0之内的。b1和b2的值都是2。

一般来说，根据整个树上的路径规则，一定有b0+b1=b0+b2。这就暗示着b1=b2。换句话
395 说，从y到它的任意没有子女或有一个子女的子孙的路径上的黑色项数量都是相同的。

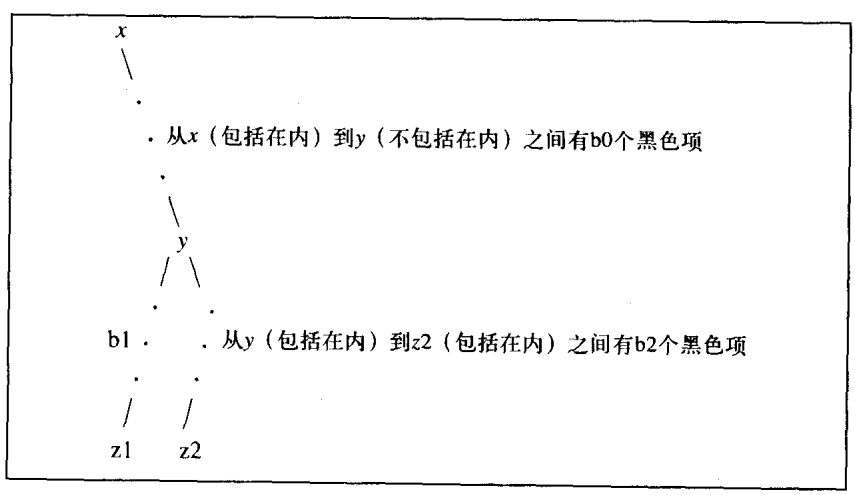
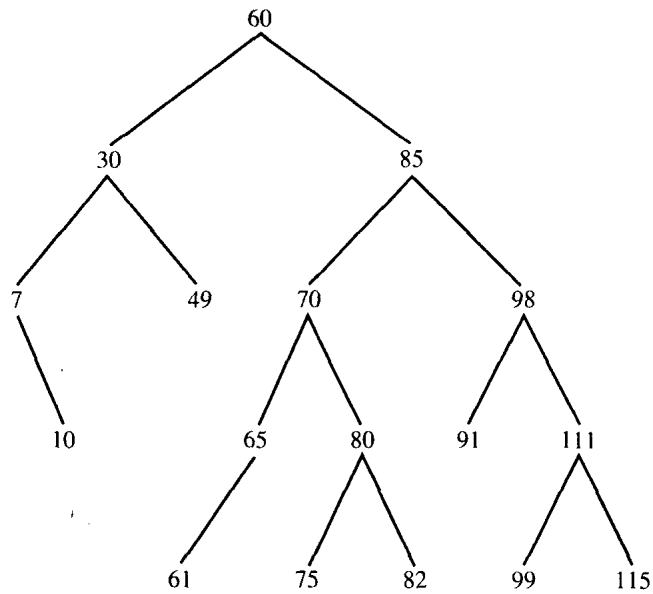


图10-6 以x为根的红黑树的一部分；y是x的一个子孙，z1和z2是任选的y的两个没有子女或有一个子女的子孙。那么b0代表了从x（包括在内）到y（不包括在内）的路径上的黑色项数量；b1和b2分别代表从y（包括在内）到z1和z2（包括在内）的路径上的黑色项数量



396

图10-7 一个根的黑色高度为3的红黑树

现在已经证明了声明1，可以进行下面的定义。令y是红黑树中的一项，那么定义y的**黑色高度**（记作bh(y)）为:

bh(y)=从y到y的任意一个没有子女或有一个子女的子孙的路径上的黑色项数量

根据声明1可知，从某一项到它的没有子女或有一个子女的任何子孙的路径上的黑色项数量必定都是相同的，因此可以定义黑色高度。例如，在图10-4中，50的黑色高度是2；20、30、40和90的黑色高度是1；10的黑色高度是0。图10-7显示了一个红黑树，其中60的黑色高度是3，而85的黑色高度是2。

2. 声明2

对一个红黑树的任意非空子树t，

$$n(t) \geqslant 2^{\mathrm{bh}(\mathrm{root}(t))} - 1$$

（在这个声明中，$n(t)$代表t中项的数量，root(t)代表t的根项。）

这个声明的证明照例是对t的高度进行数学归纳。

基本情况

假设height(t)=0。那么$n(t)$=1，如果t子树的根是黑色的那么bh(root(t))就等于1，如果根是红色的那么bh(root(t))就等于0。无论哪种情况都有1 ≥ bh(root(t))。因此

$$n(t)=1=2^1-1 \geqslant 2^{\mathrm{bh}(\mathrm{root}(t))} - 1$$

这就证明了声明2中的基本情况。

归纳情况

令k是任意非负整数，并假设对高度小于等于k的任意子树，声明2都成立。令t是一个高度为k+1的子树。如果t的根有一个子女，那么一定有bh(root(t))=1，因此

$$n(t)=1=2^1-1=2^{\mathrm{bh}(\mathrm{root}(t))} - 1$$

这就完成了当t的根只有一个子女时的归纳情况的证明。

否则，t的根一定有一个左子女v1和一个右子女v2。如果t的根是红色的，那么bh(root(t))=bh(v1)=bh(v2)。如果t的根是黑色的，那么bh(root(t))=bh(v1)+1 =bh(v2)+1。在两种情况下，都有

$$bh(v1) \geqslant bh(root(t))-1$$

及

$$bh(v2) \geqslant bh(root(t))-1$$

397 t的左右子树的高度都小于等于k，因此应用归纳假设可以得到

$$n(leftTree(t)) \geqslant 2^{bh(v1)}-1$$

及

$$n(rightTree(t)) \geqslant 2^{bh(v2)}-1$$

t中项的数量比leftTree(t)和rightTree(t)中项的数量多1。

整理所有这些公式，可以得到

$$\begin{aligned} n(t) &= n(leftTree(t))+n(rightTree(t))+1 \\ &\geqslant 2^{bh(v1)}-1+2^{bh(v2)}-1+1 \\ &\geqslant 2^{bh(root(t))-1}-1+2^{bh(root(t))-1}-1+1 \\ &=2*2^{bh(root(t))-1}-1 \\ &=2^{bh(root(t))}-1 \end{aligned}$$

这就完成了当t的根有两个子女时的归纳情况证明。

综上所述，根据数学归纳原理，声明2对所有红黑树的非空子树都是成立的。

最后将证明一个重要的结论：红黑树的高度和n成对数关系，其中n代表树中项的数量。

3. *声明3*

即使在最坏情况下，红黑树的高度也和n成对数关系。

对任何包含n个项的红黑树，height(t)和n成对数关系。

为了证明声明3，令t是一个红黑树。根据红色规则，从根到最远树叶的路径上至多有一半项可以是红色的，因此至少有一半的项是黑色的。也就是说，

$$bh(root(t)) \geqslant height(t)/2$$

根据声明2:

$$\begin{aligned} n(t) &\geqslant 2^{bh(root(t))}-1 \\ &\geqslant 2^{height(t)/2}-1 \end{aligned}$$

据此可以得到:

$$height(t) \leqslant 2\log_2(n(t)+1)$$

这就暗示了height(t)是$O(\log n)$。根据二叉树定理的第2部分，

$$height(t) \geqslant \log_2(n(t)+1)-1$$

398 所以可推断$O(\log n)$是height(t)的最小上界。换句话说，height(t)和n成对数关系。

声明3说明红黑树不会距离平衡太远。而另一方面，折半查找树在最坏情况下，比如树是一个链的时候，高度是成线性关系的。

10.1.2节描述了惠普的rb_tree类。结果是find、insert和erase方法即便在最坏情况下也只花费$O(\log n)$时间。在标准模板库中没有任何红黑树类，但是在该库的任意一个实现中都极可能包含一个以rb_tree类为原型的红黑树实例，它是set、multiset、map和multimap类定义中的一个字段。

10.1.2 惠普的rb_tree类

rb_tree类中的基本概念是**键**。回忆在第8章中，一个项的键由项与项之间用来比较的部分组成。例如，社会保障号码可以作为雇员项的一个键，而学生证号码也可以作为学生项的一个键。

下面是rb_tree类定义的开头:

```
template <class Key, class Value, class KeyOfValue, class Compare>
class rb_tree
{
```

再花些时间理解一下指定模板参数的行。**class** Key代表了键的类型。当实例化一个rb_tree对象时，就用一个实际的类替换虚类型Key。例如，如果键是string类型的，那么实例的开头将是

```
rb_tree<string,...> my_tree;
```

与Value模板参数对应的模板变元是树中插入的项的类型。通常情况下，键和值(value)的类型是相同的。例如，每个值可能是一个汽车厂商，像“Ford”，而每个键也是相同的。因此在树中将存储“Ford”，而当该项和其他项进行比较时，将比较键“Ford”和其他的键。

下一个模板参数KeyOfValue是函数类类型的。函数类和函数对象在第9章介绍过，并在实验21中进行了研究。KeyOfValue参数将被一个函数类取代，在该类中**operator**()从值中返回键。假设key是那个函数类中的一个函数对象，而项v是Value类型的。只有两种情况是需要注意的:

1) 键和值是相同的: 在这种情况里，key(v)只是返回v。这个情况适用于标准模板库中的set和multiset类。

399

2) v是一个对[⊖]，它的第一个组件是键: 这种情况下key(v)返回v的第一个组件。这种情况适用于标准模板库中的map和multimap类。例如，假设每个值都由下面的对组成: 汽车厂商和总销售额（以十亿美圆计）。那么键是汽车厂商，而每个对的第二个组件都包含了该厂商的总销售额。例如，可能有下面的两个对

“Ford”，14

“Honda”，22

⊖ 文件<utility>包含了一个模板**结构**pair，以及有两个参数的构造器:

```
template<class T1, class T2>
struct pair{
        T1 first;
        T2 second;
        pair (cont T1& x, const T2& y);
        ...
```

最后一个模板参数是Compare，这是另一个函数类类型的参数。一个rb_tree对象经常用内置函数类less作为第四个模板变元进行实例化。为什么？键通常都是通过**operator<**进行比较的。回想一下第9章中，在<function>中定义的函数类less采用如下方式重载了**operator()**:

```
bool operator()(const T& x, const T& y) const{return x<y;}
```

换句话说，如果less是实例化rb_tree对象中的第四个模板变元，那么键将根据**operator<**进行比较。为了在一个rb_tree对象中进行比较，定义了一个函数对象:

```
Compare key_compare;
```

因此如果less是实例化rb_tree对象中的第四个模板变元，那么在rb_tree对象中的消息

```
key_compare(x,y)
```

将被解释成

```
x<y
```

rb_tree中的字段和list类中的字段是相当相似的。

rb_tree类的基本组成和list类非常相似：有节点以及header、free_list、buffer_list、next_avail和last字段。还有，rb_tree类用get_node和put_node方法（取代了**new**和**delete**）处理它自身的内存管理。每个节点的结构都反映了一个红黑树的特点。下面是rb_tree_node的定义:

```
enum color_type = {red, black};
struct rb_tree_node
{
    color_type color_field;
    rb_tree_node* parent_link;
    rb_tree_node* left_link;
    rb_tree_node* right_link;
    Value value_field;
};
```

400

下面是两个着色约定:

1) header节点着色red。

2) 当一个节点最初被插入时着色red；重新着色时必须满足红色规则。

在大多数情况下，字段都是通过如parent、color和left之类的函数而不是直接进行访问的。因此用parent(header)替换了(*header).parent_link来返回对header的parent_link字段的引用。使用这些辅助函数的一个原因是它们的定义封装了所有分配模型的细节。另一个简化是允许将NULL指针看作是一个指向普通rb_tree_node的指针：它是一个特殊的节点——NIL，它的颜色是black，它的left和right字段是NULL，而它的parent字段至少在初始情况下为NULL。这说明了使用辅助函数的另一个优点：可以测试除了y为NULL的特殊情况外，color(y)是否返回颜色black。

与AVLTree类中的find方法相比，rb_tree类中的find方法的定义要稍微抽象些（用一个函数对象从值中获取键），也更深奥些（使用了逗号运算符和条件运算符）。但是基本的算法是相同的，并且worstTime(*n*)仍然和*n*成对数关系:

```
iterator find(const Key& k)
{
    rb_tree_node* y = header;
    rb_tree_node* x = root( );
    while (x != NIL)
        if (!key_compare(key(x), k))
            y = x, x = left(x);
        else
            x = right(x);
    iterator j = iterator(y);
    return (j == end( ) || key_compare(k, key(j.node))) ? end( ) : j;
}
```

insert和erase的方法定义比AVLTree中的相应方法稍稍短些，但更为复杂。在继续深入之前，需要了解insert和erase方法的定义并不是很直观的。红黑树最初是在R.Bayer(1972)的论文 401
“Symmetric binary B-trees: Data structure and maintenance algorithms”中提出的。这些被称作是“2-3-4树”中的插入和删除算法很冗长，但是整个的插入和删除策略是很容易理解的。在L.Guibas和R.Sedgewick(1978)的论文“A diochromatic framework for balanced trees”中对这些结构应用红黑着色时，所提供的方法更简短，但是更难理解。

10.1.3 rb_tree类中的insert方法

标准模板库的关联容器类的惠普实现是基于红黑树类的。

insert的方法头是：

```
pair<iterator, bool> insert(const value_type& v)
```

在进入插入的详细讨论之前，有必要解释一下返回类型：pair<iterator, **bool**>。在标准模板库的惠普实现中，rb_tree类成为四个关联容器类——set、multiset、map和multimap的基础。在set和map类里，插入一个值，它的键和容器中某些已有值的键相同，这是非法的。但是在multiset和multimap里允许出现重复的键。为了区别这些情况，rb_tree类中包含了一个字段：

```
bool insert_always;
```

每个rb_tree构造器都有一个**bool**参数always，它初始化了insert_always字段。因此set和map类构造它们的rb_tree字段时，将always参数对应的变元设置成**false**，这样就禁止了重复键。而在multiset和multimap类里，该变元设为**true**，这样就允许重复键。

当调用rb_tree类的insert方法时，如果insert_always字段的值是**true**，或者v的键没有重复树中已有的键，就插入v。那么返回的对就由一个位于插入值的迭代器和值**true**组成。但是如果insert_always字段的值是**false**，而且v的键重复了树中已有的键，就不能插入v，这时返回的对由位于给定键原始值上的迭代器和值**false**组成。

现在考虑一下如何执行插入。共需要五个步骤完成v的插入：

1) 创建一个由x指向的节点。

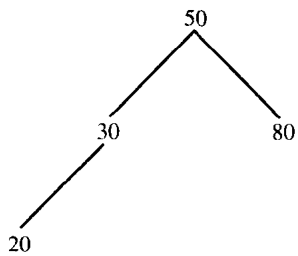
2) 在x（指向的）节点的value_field字段中存储项v。

3) 用BinSearchTree风格将节点作为一个树叶插入，并将x的颜色设置成red。

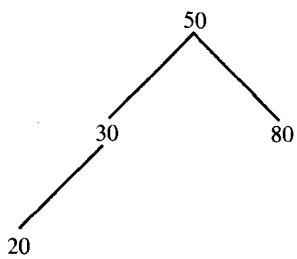
4) 需要时可通过重新着色或调整结构（重构）调整树。

402 5) 将根的颜色设置成black。

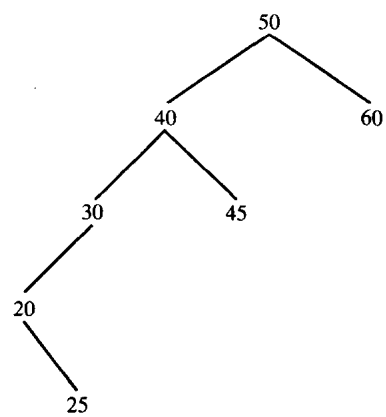
在调用真正的骨干部分_insert（它包含了变元x、x的父亲以及即将插入的项）之前，实际上只有步骤1和步骤2实现了insert方法。惟一需要仔细思考的是步骤4。首先想了解的是为什么需要步骤4。假设将20插入一个红黑树，完成步骤3之后树的状态如下:



它违背了红色规则，因此必须做些调整。在这种情况下，可以用一个简单的解决方案:将30节点和80节点的颜色都从red修改成black。这就得到如下的红黑树:



有时单纯调整颜色（重新着色）是不够的。例如，假设在一个红黑树中插入25，树的状态如下:



403

该树违背了红色规则，因此必须进行调整。只进行重新着色能否将它恢复成一个红黑树呢？读者可以自己实验一下。

实际上，只通过重新着色是不可能将上面的树恢复成红黑树的，下面解释一下原因。因为这个树的黑色高度是2，那么项50、40、30、20和25的路径上必须恰好包含两个黑色节点。但是因为根50必须是黑色的，所以项40、30、20、25的路径上必须恰好有一个黑色项。而红

色规则要求四个项的路径上至多包含两个红色节点。但是在一条四个项的路径上不可能只包含一个黑色项和至多两个红色项。

在这样的情况下，必须调整树的结构，就是对树的某些部分进行旋转。不是孤立地考虑这个例子，让我们开发一个通用框架来调整结构。

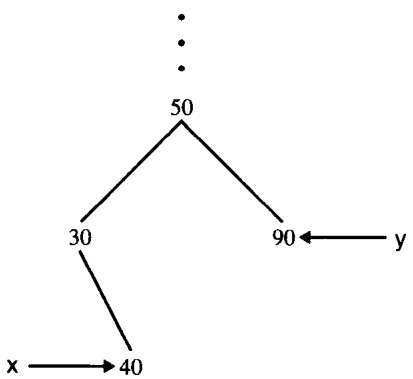
假设刚刚执行了x节点的插入。为了简化后面的讨论，就把这个节点称作x而不是x指向的节点。需要旋转吗？如果x是根，那么答案是“不需要”，因为根的颜色将在插入的第5步中设置成black。相似地，如果x的父亲是black，那么也不需要调整结构，因为这样并不违背红色规则。因此继续循环，直到满足下面的条件：

```
while(x!=root() && color(parent(x))==red)
```

实际上，临界因素是x的父亲的兄弟的颜色。下面大概描述一下当x的父亲有一个左子女时的情况——将“左”和“右”翻过来就可以得到x的父亲有一个右子女时的情况。令y指向x的父亲的（右）兄弟。就把这个兄弟节点称作y而不是y指向的节点。需要考虑三种情况。

情况1 color(y)=red

例如，假设在一个红黑树中插入40，相关部分如下：

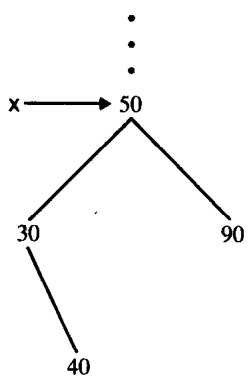


这时的行动如下：

```
color(y)=black;
color(parent(x))=black;
color(parent(parent(x)))=red;
x=parent(parent(x));
```

404

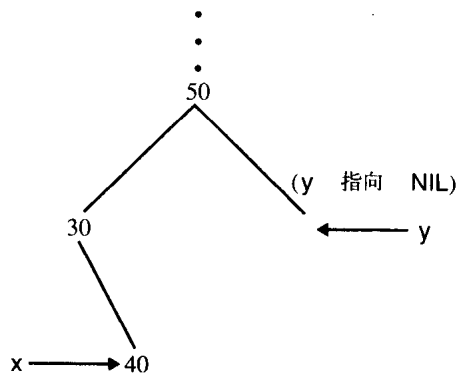
这样得到的局部树是：



在这个情况也只能在这个情况之后将继续循环。但是现在x比树高两层，因此循环迭代的最大数量将是树的高度的一半。这也就说明了之所以rb_tree类中的insert方法的worstTime(n)和树中项的数量n成对数关系的原因。

情况2 color(y)=black且x是一个右子女

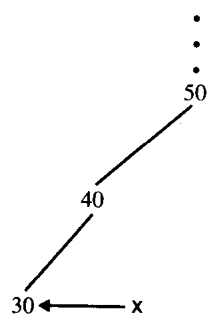
例如，假设将40插入一个红黑树，因而产生的树如下：



注意当y为NULL时，y就成为一个颜色为black的指向NIL的指针。这时的行动是：

```
x=parent(x);
rotate_left(x);
```

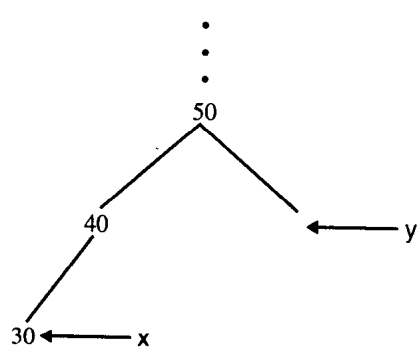
405 在这个左旋转之后的树是：



这样仍没有结束，因为x和它的父亲都是red，所以产生了情况3。

情况3 color(y)=black且x是一个左子女

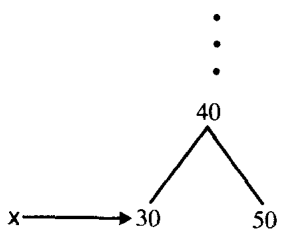
例如，假设将30插入一个红黑树，树的相关部分如下：



这时的行动是:

```
color(parent(x))=black;
color(parent(parent(x)))=red;
rotate_right(parent(parent(x)));
```

在这个右旋转之后的树的局部如下:



406

注意在采取情况3规定的行动之后，x的父亲的颜色将总是black的，因此将终止**while**循环的运行。

基本的思想是：如果情况1不适用，那么就先检测情况2，然后不论情况2是否适用都要应用情况3。注意在情况2之后总是要应用情况3的。下面是全部的组成（当x的父亲是一个左子女时）:

```
if(y是red){//情况 1
    ...
}
else{//y必须是black
    if(x是一个右子女){//情况 2
            ...
    }
    //情况 3
    ...
}
```

如果在任意一次循环迭代中不能应用情况1，那么必定应用情况3（之前可能应用情况2），并且x的父亲的颜色将被设置成black。因此在该迭代之后循环的执行将终止。

下面是完整的**while**循环:

```
while (x != root( ) && color(parent(x)) == red)
    if (parent(x) == left(parent(parent(x)))) // 如果parent(x)是一个左子女
    {
        y = right(parent(parent(x)));
        if (color(y) == red) // 情况 1
        {
            color(parent(x)) = black;
            color(y) = black;
            color(parent(parent(x))) = red;
            x = parent(parent(x));
        }
        else // y的颜色必须是black
        {
```

```
            if (x == right(parent(x))) // 情况 2
            {
                x = parent(x); rotate_left(x);
            }
            // 情况 3
            color(parent(x)) = black;
407         color(parent(parent(x))) = red;
            rotate_right(parent(parent(x)));
        }
    }
    else // parent(x)是一个右子女
    {
        y = left(parent(parent(x)));
        if (color(y) == RED) // 情况 1
        {
            color(parent(x)) = black;
            color(y) = black;
            color(parent(parent(x))) = red;
            x = parent(parent(x));
        }
        else // y的颜色必须是black
        {
            if (x == left(parent(x))) // 情况 2
            {
                x = parent(x); rotate_right(x);
            }
        // 情况 3
        color(parent(x)) = black;
        color(parent(parent(x))) = red;
        rotate_left(parent(parent(x)));
    }
}
```

紧随这个循环之后是:

```
color(root())=black;
```

实验22是一个示例，其中包含了请求插入一个项时的全部三种情况。

实验22：使用全部三种情况的红黑树插入　　（所有实验都是可选的）

正如在10.1.1节中所证明的，任何红黑树的高度都和树中项的数量n成对数关系。因此在将某项作为树叶插入时，worstTime(n)和n成对数关系。那么执行**while**循环时的worstTime(n)也和n成对数关系。综上所述，对整个insert方法而言，worstTime(n)和n成对数关系。

10.1.4 erase方法

408 在rb_tree类的研究的最后，考察一下erase方法的细节。下面是它的头:

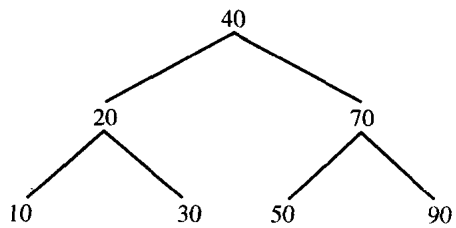
void erase(iterator position)

这个方法从树中删除position位于的节点。它也像插入一样有几种情况，不过这里是四种而不是三种情况，在认真讨论这些情况之前，先做一些准备工作。

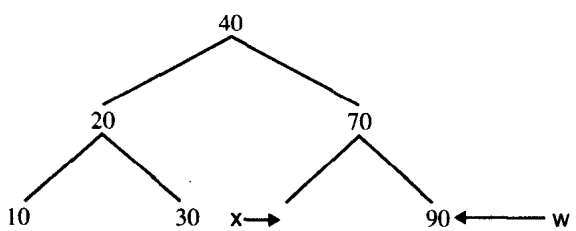
代码的开头是相当简单的。节点的删除就像BinSearchTree版本的erase方法删除节点一样。读者要确保理解了BinSearchTree类的erase方法是如何工作的。例如，如果被删除的节点只有一个子女，那么用该子女替换被删除的节点。如果被删除的节点有两个子女，那么用被删除节点的直接后任替换它[⊖]。一个红黑树的高度是$O(\log n)$，所以rb_tree类的erase方法的这个部分在最坏情况下将耗费对数时间。在删除之后不能直接停止，因为这时有可能违背红色规则和路径规则。

如果被删除节点只有一个子女，那么只需要再做一点工作。这是因为，就像在10.1.1节中提到的，被删除节点必定是black，而替换它的节点必定是red。因此将替换它的节点的颜色设置成black，得到的就仍然是一个红黑树。

如果被删除节点是一个树叶 只有当被删除节点是一个树叶或是有两个子女时才需要做更多的工作。假设被删除节点是一个树叶。如果该树叶是根或是red，那就不需要再做什么。否则，令x为（指向）NIL节点（的指针），用它替换被删除的black树叶，并令w是（指向）x的兄弟节点（的指针）。例如，假设要在下面的红黑树中删除50:



那么用一个NIL节点替换50，得到:



409

根据路径规则，因为被删除的节点不是NIL，所以w不能是（指向）NIL（的指针）。那么将继续循环通过四种情况，直到开始是black的x成为根或是red。

如果被删除节点有两个子女 现在考虑当被删除节点有两个子女的情况。如果被删除节点是red并且替换的节点也是red，那么得到的仍旧是一个红黑树。例如，如果从图10-8所示的红黑树中删除80，将得到图10-9所示的红黑树。

410

此外，如果被删除节点是black，而替换的节点是red，那么只需要将替换节点的颜色改成black，得到的就仍然是一个红黑树。例如，如果从图10-9所示的红黑树中删除40，将得到图10-10所示的红黑树。因此如果被删除节点有两个子女并且替换的节点是red，那么只需要一点

⊖ 因为可能有另一个迭代器位于该后任上，所以比只是替换被删除节点的代码要复杂些。

工作就能够保证树满足红色规则和路径规则。

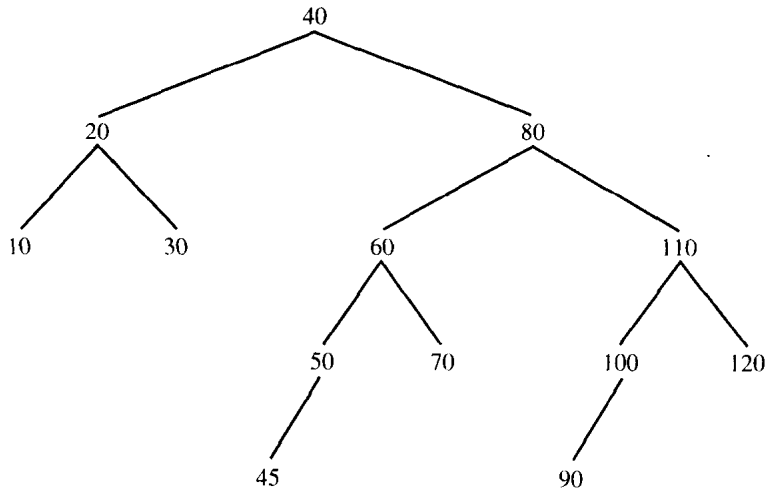


图10-8 即将从中删除80的红黑树

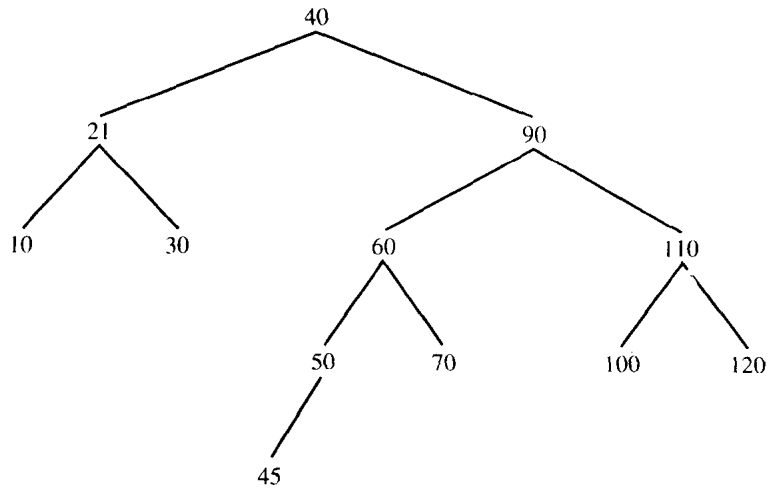


图10-9 从图10-8中删除80之后的红黑树

现在假设被删除节点有两个子女并且替换的节点是black。开始先将被删除节点的颜色赋予替换的节点。如果替换节点有一个非空右子女，那么这个子女必定是red的，因为替换节点不可能有左子女。删除时，替换节点的右子女将取代替换节点。然后将这个右子女着色black，得到一个红黑树。

假设替换节点有一个空的右子女。令x为（指向）NIL节点（的指针），它取代了替换节点。令w为（指向）x的兄弟（的指针）。也就是说，w是替换节点的兄弟。例如，从图10-11所示的红黑树中删除40之后，将得到图10-12所示的树。注意，由于替换节点是black的，所以路径规则要求w不能是NIL。

如果x开始是red，那么所有需要做的就是将x设成black，然后就满足了这两个规则。类似地，如果x开始是根，那么将x设成black就完成了。惟一需要做较多工作的情况是当x是一个

black的非根节点时，将不断循环直到x为red或者x是根节点。

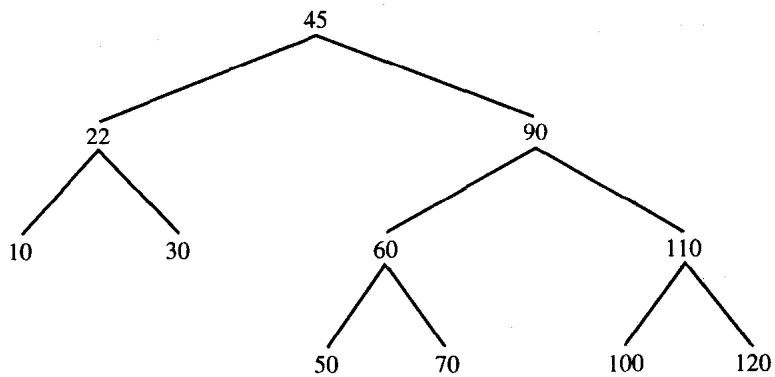


图10-10 在图10-9中删除40之后的红黑树

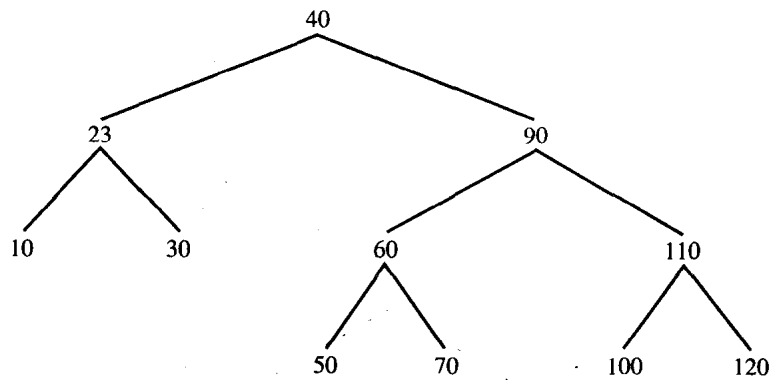


图10-11 即将从中删除40的红黑树。替换的项是50

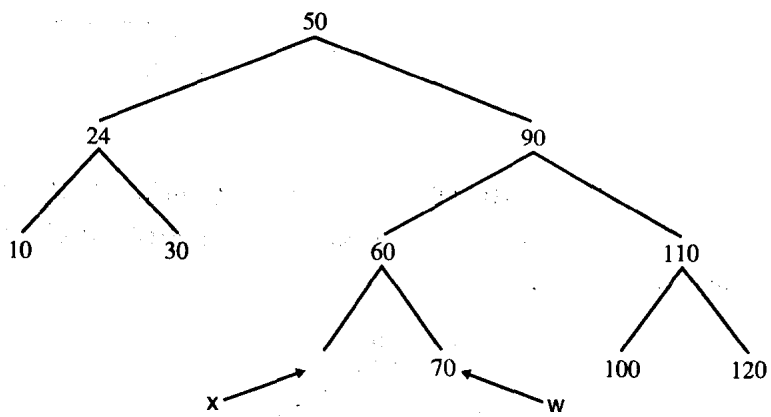


图10-12 从图10-11中删除40之后的树

如果被删除节点是一个树叶或者有两个子女 如果被删除节点是一个树叶，那么x是（指向）NIL节点（的指针），它取代了替换节点。如果被删除节点有两个子女，那么x是（指向）替换节点的右子女（的指针）。在这两种情况中，去掉被删除的节点之后，w都是x 的兄弟。然后将继续循环，直到x是red或者x是根。当x是左子女时共有四种情况，而当x是右子女时也 411

有四个对称的情况（只是交换“左”和“右”）。和讨论插入时一样，这里只介绍x是一个左子女的情况。本节的最后将说明全部的八种情况。

无论被删除节点是一个树叶或是有两个子女，都适用下面的情况。

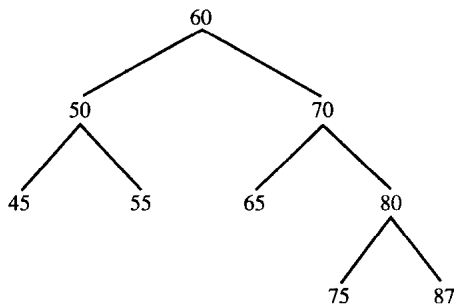
情况1 color(w)=red

这种情况下采取的行动是：

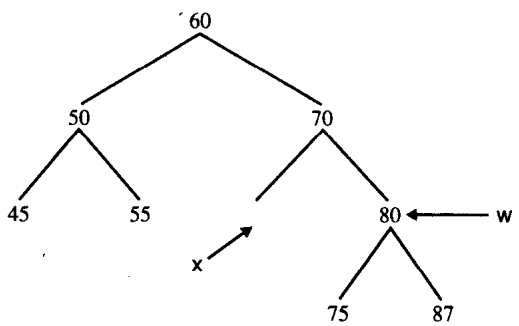
```
color (w) = black;
color (parent (x)) = red;
rotate_left( parent (x));
w = right (parent (x));
```

412

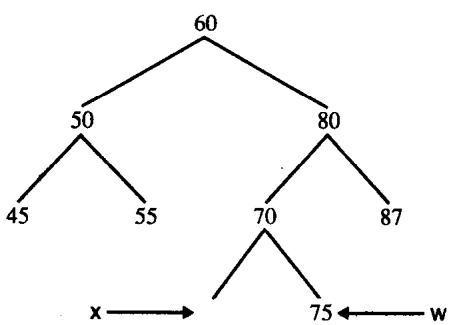
例如，假设从下面的树中删除65：



删除65之后，x和w如下所示：



在这个树中，x是（指向）NIL（的指针）。因为w是red的而且x是一个左子女，所以采用情况1。因此将w的颜色设置成black，x的父亲的颜色设置成red，在x的父节点处进行左旋转，并将w设置成x的父亲的右子女：



这时仍然不满足路径规则，但是有了一个新的着色为black的w!

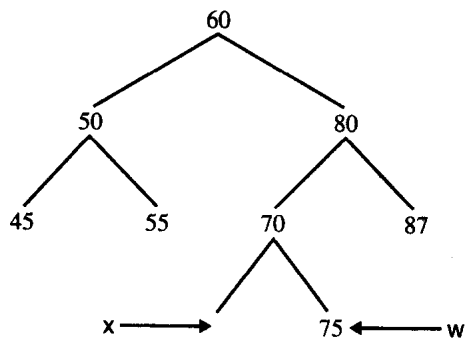
这样情况1就转换成了其他三种情况之一。 413

情况2 w的子女是black

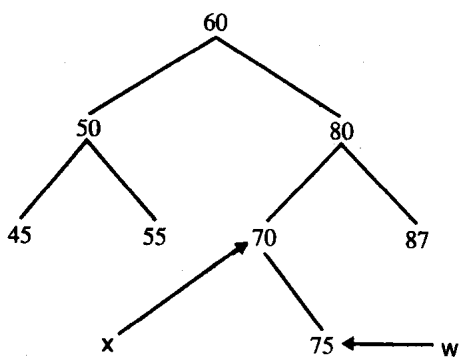
回想一下，w不能是NIL，而w的一个或两个子女可能是NIL，但是NIL的颜色是black，因此不需要专门区分w的子女为NIL的情况。无论x是一个右子女或左子女，情况2中采取的行动是:

```
color(w)=red;
x=parent(x);
```

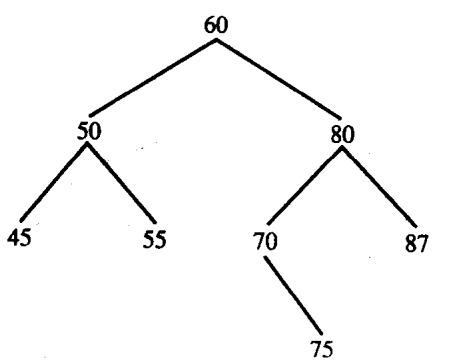
例如，可以在情况1的范例最终得到的折半查找树上应用情况2。在该例子的最后得到了:



因为w的两个子女都是black（回忆NIL节点都是black的），所以可以应用情况2得到:



现在x是red，因此循环失败，将x的颜色设置成black，最终得到如下红黑树:



414

迄今为止看到的erase方法中循环的大体轮廓是:

```
while (x不是根且x的颜色为black)
{
    if (x == x的父亲的左子女)
    {
        w = x的父亲的右子女;
        if (w的颜色是red)
        {
            ... 情况1 ...
        }
        if (w的子女都是black)
        {
            ... 情况2 ...
        }
        else
        {
            ... 情况3和情况4 ...
        }
    else // x是一个右子女
    {
        ...
    }
} // while
 color (x) = black;
```

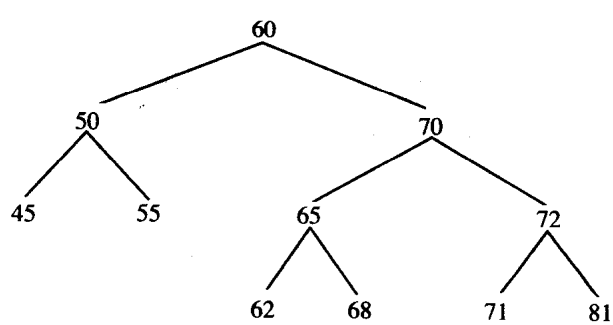
幸运的是，循环总会在情况3或情况4之后终止。如果适用情况1，那么循环总会在单次迭代之后终止，除非处理完情况1之后又应用了情况2。情况2是惟一需要另外的循环迭代的情况，而且当应用它时x将更接近根，因此**while**循环至多执行$O(\log n)$次。

情况3 w的右子女是black

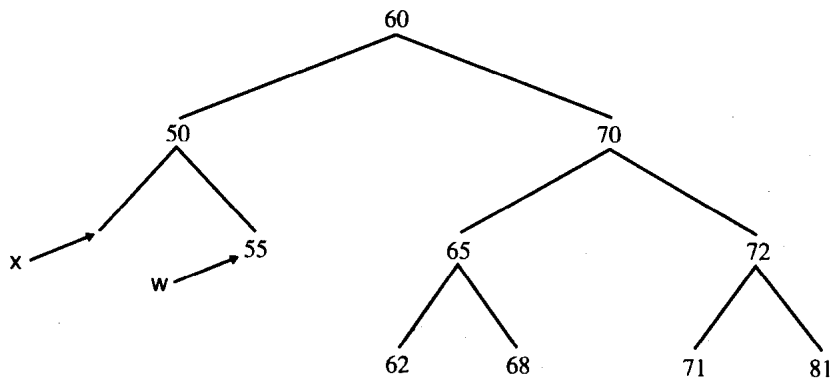
根据刚刚给出的erase方法的轮廓，很显然在这种情况下，w的两个子女不可能都是black。因此w的左子女必定是red的。这时采取的行动如下:

```
color(left(w))=black;
color(w)=red;
rotate_right(w);
w=right(parent(x));
```

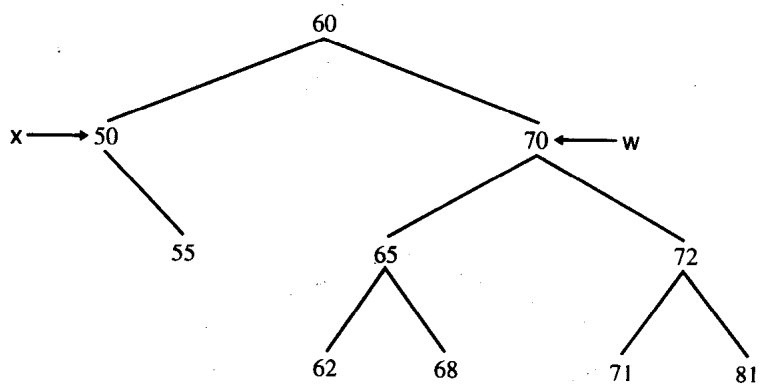
415 例如，假设从下面的红黑树中删除45:



在**while**循环的开头有:

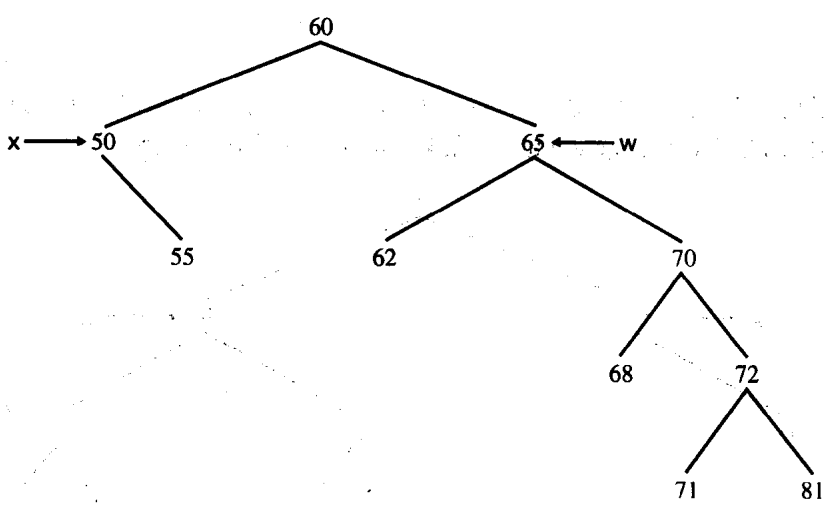


因为w是black而且w的两个子女都是black（NIL节点着色black），所以采用情况2。应用情况2之后，在下一个循环迭代里将得到:



这时适用情况3，因为w是black的且w的左子女是red而右子女是black。重新着色，右旋转并重新设置w之后得到:

416



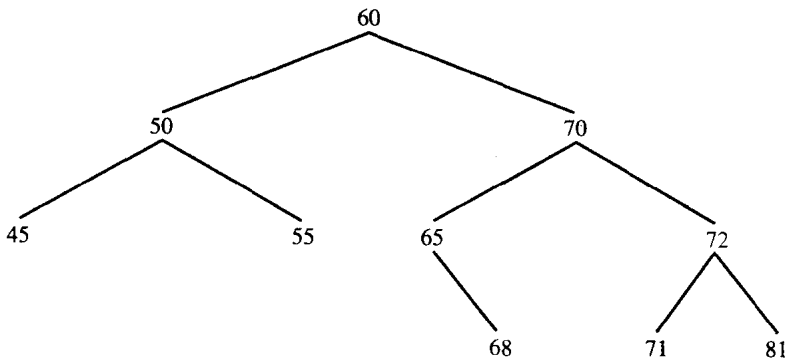
乍看起来这好像没什么改进。但是注意w的右子女现在是red的，这将带我们进入情况4。

情况4 w的右子女是red

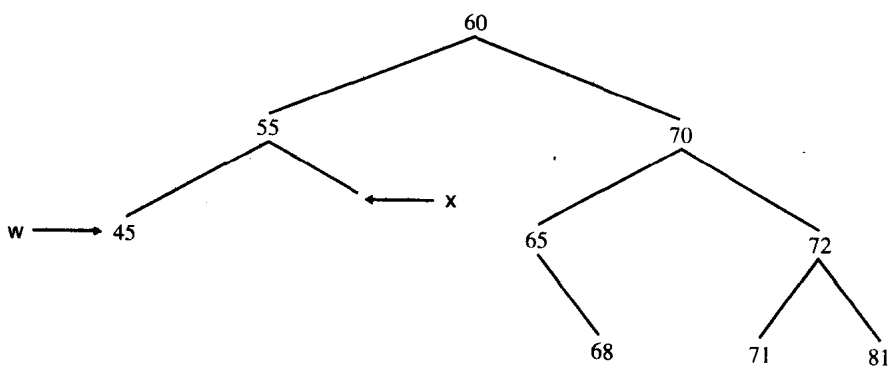
下面是将采取的行动:

```
color (w) = color (parent (x));
color (parent (x)) = black;
color (right (w)) = black;
left_rotate (parent (x));
break; // 终止循环
```

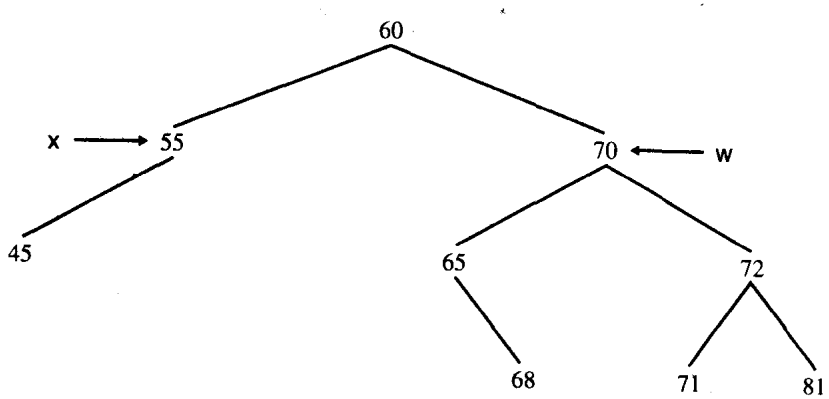
例如，假设从下面的红黑树开始:



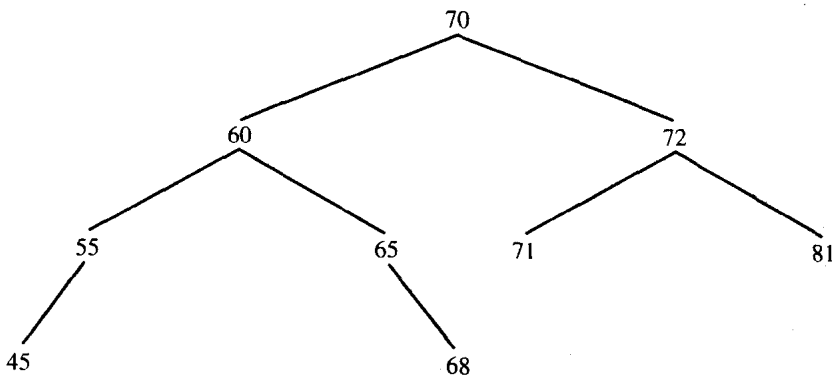
417 删除50之后有:



这时适用情况2，而且当x是一个右子女时和x是左子女时采取的行动相同。因此将color(w)设置成red并将x设置成parent(x)。在循环的下一次迭代中得到:

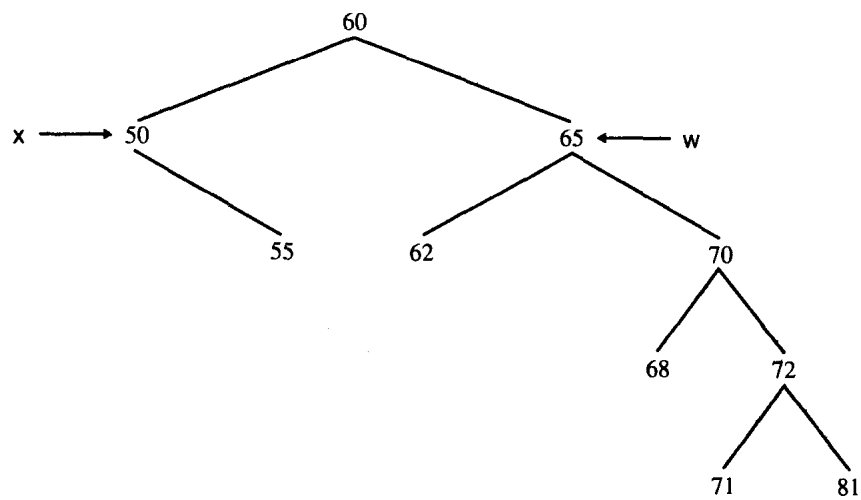


现在适用情况4，因此重新着色并围绕x的父亲进行左旋转:

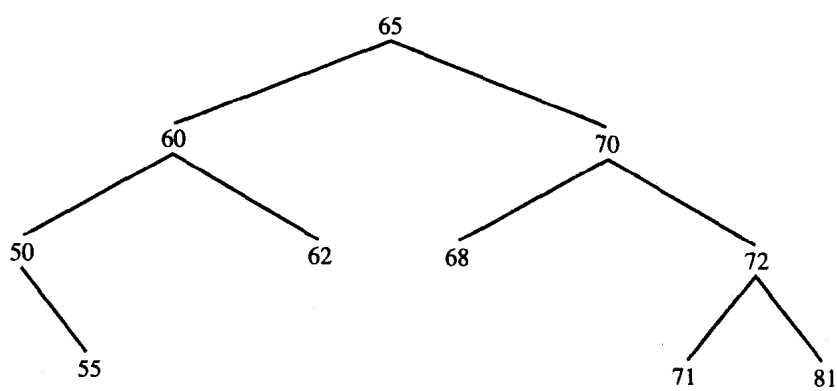


这个树同时满足了红色规则和路径规则，因此结束。 418

下面这个例子是情况4前面是情况3的情形，假设从情况3示例末尾的树开始:



w的右子女是red，因此适用情况4。重新着色并围绕x的父亲左旋转之后得到:



现在既满足了红色规则又满足了路径规则，因此结束。

只要应用情况3，那么w的右子女将变成red，因此在情况3之后总是应用情况4。比复制情况4的代码更好的方法是将该代码放在情况3代码的后面。当x是一个左子女时，情况3和情况4

的大致结构是:

```
if(w的右子女是black)
{
...情况3...
}
...情况4...
```

在本节开头提到过，erase方法的第一部分——使用了BinSearchTree方式的删除——在最坏情况下花费的时间与n成对数关系。在**while**循环中组成了erase方法的第二部分，只有在应用情况2时循环继续，然后x向根移动，迭代的最大次数是$O(\log n)$。将这两部分合并在一起，erase方法在最坏情况下花费的时间和n成对数关系。平均情况下，这两个部分各自只花费常数时间，因此averageTime(n)是常数。实际上amortizedTime(n)也是常数。

下面是包含了x是左子女时四种情况的完整**while**循环，以及x是右子女时的四个对称情况。

```
while (x != root( ) && color(x) == black)
    if (x == left(parent(x)))
    {
        link_type w = right(parent(x));
        if (color(w) == red) // 情况1
        {
            color(w) = black;
            color(parent(x)) = red;
            rotate_left(parent(x));
            w = right(parent(x));
        }
            if (color(left(w)) == black && color(right(w))
                    == black)// 情况2
            {
            color(w) = red;
            x = parent(x);
            }
            else
            {
            if (color(right(w)) = black) // 情况3
            {
                color(left(w)) = black;
                color(w) = red;
                rotate_right(w);
                w = right(parent(x));
            }
            // 情况4
            color(w) = color(parent(x));
            color(parent(x)) = black;
            color(right(w)) = black;
            rotate_left(parent(x));
            break;
            }
```

```
        }
        else
        {
            //
            // 与前面的对称; 交换"左"和"右"
            //
            link_type w = left(parent(x));
            if (color(w) == red) // 情况1
            {
                color(w) = black;
                color(parent(x)) = red;
                rotate_right(parent(x));
                w = left(parent(x));
            }
            if (color(right(w)) == black && color(left(w)) ==black)// 情况2
            {
                color(w) = red; x = parent(x);
            }
            else
            {
            if (color(left(w)) == black) // 情况3
            {
                color(right(w)) = black;
                color(w) = red;
                rotate_left(w);
                w = left(parent(x));
            }
            // 情况4
            color(w) = color(parent(x));
            color(parent(x)) = black;
            color(left(w)) = black;
            rotate_right(parent(x));
            break;
        }
    } // while循环结束
    color(x) = black;
```

实验23说明了在单次erase方法调用中是如何应用全部四种情况的。

实验23：erase的调用，其中应用了全部四种情况 （所有实验都是可选的）

421

10.2 标准模板库的关联容器

下面即将讨论标准模板库中的四个关联容器类。回忆在第8章中，**关联容器**是通过项之间键的比较来确定项位置的容器。

*rb_tree*类主要目的不是直接的应用，而是作为标准模板库中四个关联容器类的典型实现的基础。根据对下面两个问题的四种可能的答案，可以分出四种相应的数据结构:

一个项只由一个键组成吗?

容器允许多个项使用相同的键吗?

下面的表说明了这些问题的答案是如何决定了四种数据结构的。

只由键组成吗?	允许重复项吗?	
	是	否
是	多集合	集合
否	多映射	映射

例如，集合的定义是一个关联容器类，其中每个项仅由一个键组成，而且不允许重复项；它的插入、删除和查找的worstTime(n)与n成对数关系。自此以后，我们主要关心的是这四种数据结构的可能的实现。先从set和multiset类开始，然后再研究map和multimap类。

set类

在标准C++中，set类声明的开头如下:

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key>>
class set
{
```

和往常一样，先忽略分配器的工作。在set类里，键就是整个的项，而且键是独一无二的。例如，下面是set对象的定义，其中包含了按词典顺序的字符串项:

```
set <string> names;
```

422 因为集合是一个关联容器，它的项是根据和其他项之间的比较进行存储的。而比较使用的是模板参数Compare对应的模板变元，这个模板变元在缺省情况下是函数类less，它在标准模板库中的<function>里的定义如下:

```
template <class T>
struct less : binary_function<T, T, bool> {
      bool operator( )(const T& x, const T& y) const { return x < y; }
};
```

因此如果函数对象comp是函数类less的一个实例，那么comp(x,y)将返回表达式x<y的数值。当然，不使用缺省情况，也可以指定除less之外的函数类——甚至可以指定用户声明的函数类。例如，可以定义一个set对象，其中包含降序存储的**double**类型的项:

```
set<double, greater <double> > salaries;
```

在惠普的实现中，有一个rb_tree类型的**private**字段:

```
      typedef Key key_type;
      typedef Key value_type;
      typedef Compare key_compare;
      typedef Compare value_compare;
private:
```

```
typedef rb_tree<key_type, value_type,
        ident<value_type, key_type>, key_compare> rep_type;
rep_type t; // 红黑树表示集合
```

在ident函数类中，**operator**()获取一个value_type类型的变元并简单的返回该变元。因此t是一个rb_tree的实例，其中键就是整个值。

set类包含了通常的各色容器方法：多个构造器，一个析构器，begin，end，size，empty，find，insert和erase。大部分的方法定义只不过是让t调用相应的rb_tree方法。例如，find和erase的定义如下：

```
iterator find(const key_type& x) const{return t.find(x);}
void erase(iterator position){t.erase((rep_type::iterator&)position);}
```

insert方法是不同的，因为如果一个项和set容器中已经存在的项相同将不能被插入。因此，就像在10.1.3节中所解释的，rb_tree类中的insert方法必须返回一个对：一个迭代器和一个**bool**值。下面是它的定义，还有方法接口：

```
//后置条件：如果项x已经出现在集合中，那么返回的对就由一个位于此插入项前
//          的迭代器和false组成。否则，返回的对就由一个位于新插入项的
//          迭代器和true组成。
pair<iterator, bool> insert(const value_type& x)
{
    pair<iterator, bool> p = t.insert(x);
    return pair<iterator, bool>(p.first, p.second);
}
```

423

例如，如果my_company是一个雇员的set容器，那么可以有：

```
pair<iterator, bool> p = my_company.insert (employee);
if (!p.second)
    cout << "duplicate item; insertion not made." << endl;
```

由于find、erase和insert方法只是调用它们的rb_tree的对应部分，所以这三个set方法的worstTime(n)都是和n成对数关系。

下面的程序将整数按照降序插入set中；并拒绝重复的整数：

```
#include <iostream>
#include <string>
#include <set>

using namespace std;

int main( )
{
    typedef set< int, greater< int > > my_set;

    const int SIZE = 8;

    const string HEADING = "Here are the items in the set:";

    const string REPEAT = " is already in the set. Insertion rejected.";
```

```
    const string CLOSE_WINDOW_PROMPT =
          "Please press the Enter key to close this output window.";

    my_set s;
    my_set::iterator itr;
    int data [SIZE] = { 5, 3, 9, 3, 7, 2, 9, 3 };

    for( int i = 0; i < SIZE; i++ )
    {
          pair< my_set::iterator, bool > p = s.insert (data [i]);
          if( !p.second)
                cout << data[ i ] << REPEAT << endl;
    } // for

    cout << endl << HEADING << endl;
    for( itr = s.begin( ); itr != s.end( ); itr++ )
          cout << *itr << endl;
    cout << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );
    return 0;
} // main
```

424

输出是:

```
3 is already in the set. Insertion rejected.
9 is already in the set. Insertion rejected.
3 is already in the set. Insertion rejected.

Here are the items in the set:

9
7
5
3
2
Please press the Enter key to close this output window.
```

当然，在这个程序中如果确信数组data中所有的项都是独一无二的，那么可以简单地忽略返回值并写为:

```
s.insert(data[i]);
```

10.3 集合应用：再次讨论拼写检查器

9.4节中开发了一个拼写检查器作为AVLTree类的应用。做很小的改动，就可以把拼写检查器程序改造成set类的应用。下面是需要做的工作:

1) 在spellcheck.h中，将#include "avl.h" 替换成#include <set>。

2) 在spellcheck.h中，把dictionary和words的定义里的AVLTree替换成set。

3) 在spellcheck.cpp中，把compare()方法里的itr定义中的AVLTree和Iterator分别替换成set和iterator。

作为一个set类应用，其时间花费与AVLTree应用中的是一样的。

10.3.1 multiset类

在multiset容器里，每项只由一个键组成而且允许重复项。为了允许重复，每个multiset构造器都在调用rb_tree构造器时将**true**传递给always。那么rb_tree对象中的insert_always字段将获得值**true**。 425

由于在一个multiset对象里允许重复项，所以insert方法只是返回一个位于新插入项的迭代器。又因为一个项可能有多个拷贝，所以有一个multiset方法可以返回位于第一个不破坏多集合顺序且能够插入的位置上的迭代器。

```
iterator lower_bound(const T& x) const;
```

同样，upper_bound方法返回位于最后一个不破坏多集合顺序且能够插入的位置上的迭代器。equal_range方法返回一对分别位于给定项的第一个和最后一个这样位置上的迭代器。例如，下面的程序创建一个整数的multiset容器并允许重复：

```
#include <iostream>
#include <string>
#include <set> // 声明集合和多集合
using namespace std;
int main( )
{
    typedef multiset< int, less< int > > my_set;
    typedef pair<my_set::iterator, my_set::iterator > Range;
    const int SIZE = 8;
    const string HEADING = "Here are the items in the multiset:";
    const string THREES = "Here are the threes: " ;
    const string CLOSE_PROMPT =
        "Please press the Enter key to close this output window.";
    my_set s;
    int data [SIZE] = { 5, 3, 9, 3, 7, 2, 9, 3 };
    my_set::iterator itr;
    for(int i = 0; i < SIZE; i++ )
        s.insert (data [i]);
    cout << endl << HEADING << endl;
    for (itr = s.begin( ); itr != s.end( ); itr++)
        cout << *itr << endl;
    cout << endl << THREES << endl;
    Range result = s.equal_range (3);
    for ( itr = result.first; itr != result.second; itr++ )
        cout << *itr << endl;
    cout << endl << CLOSE_PROMPT;
```

426

```
    cin.get( );
    return 0;
} // main
```

result变量包含了一对迭代器：result.first位于多集合中3的第一个实例；result. last位于多集合中3的最后一个实例随后的位置。

实验24提供了用set和multiset类进行实验的机会。

实验24：更多与set和multiset类相关的知识　　（所有实验都是可选的）

本章最后将考察map和multimap类。其中尤为重要的是map类的关联数组运算符——**operator**[]。它们的特别之处在于索引不必是整数：索引可以是一个string，一个Employee，或任何东西！

10.3.2 map类

在标准C++中，map类声明的开头是：

```
template<class Key, class T, class Compare = less<Key>,
    < Allocator = allocator<T> >
class map
{
```

在map类中，每个值都是一个对——<Key,T>，并且不允许重复的键。例如，定义一个称作students的map容器，其中键是学生的姓名（一个string），T代表学生的当前年级平均分的类型，并且姓名按照字母顺序存储，那么可以写为：

```
map<string,float,less<string>> students;
```

因为一个学生在某个给定时间只能有一个年级平均分，所以map容器students在一个学生和他的年级平均分之间定义了惟一的关联。实际上，students将每个学生“映射”到了他的年级平均分上。在map容器students内部将没有键相同的对，也就是说，没有相同学生姓名的值。

和其他的关联类相同，很多方法是可用的：各种构造器，size，empty，insert，erase，find，begin，end…… insert方法采用一个值——即一个对——作为它的变元；回忆一下，pair类的构造器可以用来初始化一个“对”对象。因此可以写为：

```
pair<string,float>student("Stamp,Lisa",3.96);
students.insert(student);
```

427

map类有一个非常优越的特色：关联数组。在普通数组中，索引是一个整数。而在**关联数组**中，索引是一个键，并且键可以是任意类型的，可以是string、**double**、**int**类型，甚至可以是一些用户定义的类。如下赋值语句

```
a[x]=m;
```

将对<x, m>插入进映射a。如果映射中已经有一个键为x的对会怎样？那么赋值语句的作用是把那个对的第二个组件替换成m。例如，前面向map容器students中进行的插入可以被替换成

```
students["Stamp,Lisa"]=3.6;
```

为了更好地了解关联数组的方便性，考虑文本文件中每个单词出现的频率的计算问题。每个单词后面可能跟着标点符号，它们不是单词的一部分。我们将创建一个映射，其中的每个项都是一个对：一个惟一的单词——键，以及文件中单词出现的次数。下面是程序：

```
#include <string>
#include <ctype.h> //  声明  tolower, isalpha
#include <fstream>
#include <map>
using namespace std;
typedef map< string, int, less< string >>  FrequencyMap;

int main( )
{
    const string INPUT_PROMPT =
        "Please enter the name of the input file:";
    const string OUTPUT_PROMPT =
        "Please enter the name of the output file:";
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";
    FrequencyMap frequencies;
    FrequencyMap::iterator itr;
    string in_file_name,
        out_file_name,
        word;
    ifstream in_file;
    ofstream out_file;
    cout << INPUT_PROMPT << endl;
    cin >> in_file_name;
    in_file.open (in_file_name.c_str( ), ios::in);
    cout << endl << OUTPUT_PROMPT << endl;
    cin >> out_file_name;
    out_file.open (out_file_name.c_str( ), ios::out);
    while (in_file >> word)
    {
        // 使单词为小写:
        string temp;
        for (unsigned i = 0; i < word.length( ); i++)
            temp += (char)tolower (word [i]);
        word = temp;
        while (!isalpha (word [word.length( ) - 1]) )
            word.erase (word.length( ) - 1);
```

428

```
            if (frequencies.find (word ) == frequencies.end( ))
                frequencies [word] = 1;
            else
                frequencies [word]++;
        } // while

        for (itr = frequencies.begin( ); itr != frequencies.end( ); itr++)
            out_file << (*itr).first << ", " << (*itr).second << endl;

        cout << endl << endl << CLOSE_WINDOW_PROMPT;
        cin.get( );
        return 0;
    } // main
```

输出文件由<单词，频率>形式的对组成，并且单词是按照字母顺序排列的。例如，假设输入文件是:

This program counts the
number of occurrences of words in a text.
The text may have many words
in it, including big words.

那么输出文件将是:

a, 1
big, 1
counts, 1
have, 1
in, 2
429 including, 1
it, 1
many, 1
may, 1
number, 1
occurrences, 1
of, 2
program, 1
text, 2
the, 2
this, 1
words, 3

map类的任何实现都很可能是基于平衡折半查找树的。例如，下面给出的是来自惠普的实现的:

```
private:

    typedef rb_tree<key_type, value_type,
        select1st<value_type, key_type>, key_compare> rep_type;

    rep_type t; // 用红黑树表示映射
```

在select1st函数类中，函数调用运算符**operator**()采用了一个value_type类型的参数，也就是对<**const** Key,T>。对中的第一个组件——键——被返回。因此t是一个rb_tree对象，其中每个值都是一个对，而且对的比较是根据每个对的第一个组件进行的。

从这点讲，map类中的所有方法定义都是单行的，其中t调用对应的rb_tree方法。甚至关联数组运算符**operator**[]的定义也是单行的:

```
T& operator[ ](const key_type& k)
{
    return (*((insert(value_type(k, T( )))).first)).second;
}
```

需要用一段来解释这一行。因为value_type是pair类型的，所以对参数k以及T的缺省构造器应用pair构造器；构造一个pair对象。这个pair对象，称作p，被插入映射中。实际上，映射的insert方法调用t.insert(p)，它将返回一个迭代器-**bool**对。如果在树中已经有一个值的键是k，那么迭代器就位于这个值上。否则，迭代器将位于新插入的<k,T()>对上。无论哪种情况，迭代器都将脱引用，得到一个值，也就是一个<Key,T>类型的对。对这个对中第二个组件（T类型）的引用被返回。这个**return**语句难以理解的一个原因是因为first代表的是迭代器-**bool**对的第一个组件，而second代表的是Key-T对的第二个组件。 430

编程项目10.1是map类的一个应用，在第13章和第14章中还将有几个关联数组的例子。为了完整性，这里也将介绍multimap类，但它的应用非常少。

10.3.3 multimap类

正如你所料，multimap允许有多个键值相同的项，而且每个项都是一个对<Key,T>。为了避免多义性，这里禁止使用关联数组：如果一个multimap容器myMulti中有对<"Mark", 3>和<"Mark", 5>，那么表达式

```
myMulti["Mark"]
```

可以引用任一对的第二个组件。

下面是一个multimap定义的例子:

```
multimap<int,string,greater<int>> most_wins;
```

multimap容器most_wins将由过去100年中美国棒球协会中历年的最大胜利次数和得到这些胜利的投手组成。需要一个multimap类，因为胜利的比赛次数可能存在重复。multimap将按照胜利次数的降序排列。下面是一个插入示例:

```
most_wins.insert(pair<int,string>(31,"Denny McLain"));
```

multimap类的一个与众不同的特点是它的insert方法包含一个**提示**迭代器:

```
iterator insert(iterator position, const value_type& x);
```

和insert的其他版本一样，方法将在树中查找x归属的位置。迭代器position将提示从哪里开始查找。例如，假设要在多映射multi中插入数组data里的n个项，并假设data中的那些项已经按正确顺序——升序——排列了。可以按如下方式进行下去:

```
itr=multi.begin();
```

```
for(int i=0;i<n;i++)
    itr=multi.insert(itr,data[i]);
```

在**for**循环的每次迭代中都调用insert方法，itr提供了查找的开始点。因为项都是按照顺序插入的，所以itr总是位于最大的项上。那么新的项将以常数时间而不是对数时间插入。插入n个项的总时间将只是$O(n)$，而不是不使用提示迭代器时的$O(n\log n)$。

实际上，带提示的插入方法也可以用于set、multiset和map类，但是它们还具有其他的特点。

431

实验25：更多与map和multimap类相关的知识　　（所有实验都是可选的）

总结

红黑树是一个折半查找树，它或者为空，或者根项着黑色，而其他的每个项着红色或黑色，并满足下面的属性：

红色规则：如果某项着红色，那么它的父亲必须是黑色的。

路径规则：从根项到没有子女或**有一个子女**的项的所有路径上的黑色项的数量必须是相同的。

红黑树的高度总是和n成对数关系。

本章还介绍了标准模板库中的四个容器类，它们比较有代表性的实现都是基于红黑树的。这四个类是

1) set:　　value_type=key_type; 不允许重复。
2) multiset:　　value_type=key_type; 允许重复。
3) map:　　value_type=pair<key_type,T>; 不允许重复。
4) multimap:　　value_type=pair<key_type,T>; 允许重复。

习题

10.1 写出将下面的数值插入一个初始为空的红黑树后的结果：

30,40,20,90,10,50,70,60,80

10.2 从习题10.1的红黑树中删除20和40。说明每次删除之后整个树的情况。

10.3 构造一个大小是20而且没有red项的红黑树是不可能的。解释原因。

10.4 任选整数$h \geq 1$，并按照如下方式创建一个红黑树：插入1，2，3，…，2^{h+1}。删除2^{h+1}，$2^{h+1}-1$，$2^{h+1}-2$，…，2^h。用h=1，2和3分别进行试验。最终得到的红黑树有什么不寻常之处？

Alexandru Balan有助于开发这个公式。

10.5 假设v是一个红黑树中具有一个树叶的项。解释一下v为什么必须是black，而v的子女必须是一个red树叶。

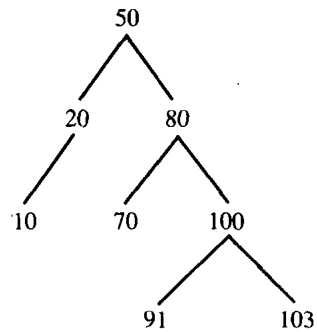
10.6 构造一个红黑树（忽略颜色时），使它不是一个AVL树。

10.7 Guibas和Sedgewick(1978)提供了将任意AVL树着色成红黑树的一个简单的算法：对AVL树中的每一项，如果以该项为根的子树高度是一个偶整数并且以它的父亲为根

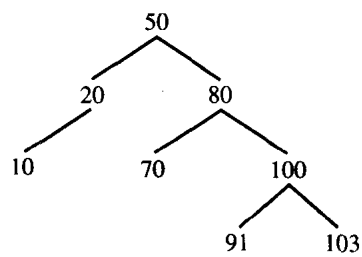
的子树高度是奇数，那么将该项着色red；否则，着色black。

例如，考虑下面的AVL树：

432



在这个树中，20的子树高度是1，80的子树高度是2，10的子树高度是0，70的子树高度是0。注意因为整个树的根没有父亲，所以这个算法保证了根将着色black。下面是这个AVL树着色后得到的红黑树：



用最小的4个项创建一个高度为4的AVL树，然后将该树着色成一个红黑树。

10.8 假设在红黑树定义中把路径规则替换成如下形式：

路径规则2：所有从根项到树叶的路径上的黑色项数量必须相同。

a. 找出一个包含n个项的（采用这个新定义的）红黑树，它的高度和n成线性关系。

b. 找出一个折半查找树，它不能着色成红黑树（即使使用这个新定义）。

10.9 在BinSearchTree类中，Iterator类的自减方法**operator**--()需要tree_node结构包含一个isHeader字段：

```
Iterator& operator-- ( )
{
    If (link -> isHeader)
        link = link -> right; // 返回最右边的
    ...
```

433

在rb_tree类中，rb_tree_node结构没有isHeader字段。那么用什么代码代替条件link->isHeader呢？

10.10 假设已知一个公司里每个雇员的姓名和部门号。没有重复的姓名，并希望根据姓名的字母顺序存储这些信息。例如，部分输入可能如下所示：

Misino,John	8
Nguyen,Viet	14
Panchenko,Eric	6

Dunn,Michael 6
Deusenbery,Amanda 14
Taoubina,Xenia 6

而我们希望按照下面的顺序存储这些项:

Deusenbery,Amanda 14
Dunn,Michael 6
Misino,John 8
Nguyen,Viet 14
Panchenko,Eric 6
Taoubina,Xenia 6

解决这个问题应当使用set、multiset、map或multimap?

10.11 重做习题10.10，不过现在要求按照部门号码升序排列，而且在每个部门号内部按照姓名的字母顺序排列。例如，部分输入可能如下:

Misino,John 8
Nguyen,Viet 14
Panchenko,Eric 6
Dunn,Michael 6
Deusenbery,Amanda 14
Taoubina,Xenia 6

而我们希望按照下面的顺序存储这些项:

Dunn,Michael 6
Panchenko,Eric 6
Taoubina,Xenia 6
Misino,John 8
Deusenbery,Amanda 14
Nguyen,Viet 14

解决这个问题应当使用set、multiset、map或multimap?

10.12 假设有

434

```
map<string, int> my_map;
```

解释

```
my_map.insert("Ford",20000);
```

和

```
my_map["Ford"]=20000;
```

之间的区别。

提示 如果“Ford”已经作为my_map中某个值的键出现过会怎样?

10.13 对下面的每个类型，构造一个适用它的对象并解释为什么该类型适合这个对象。

```
set< double, greater< double> >
multiset<string>
map< double, string>
multimap< int, double, greater<int>>
```

435

编程项目10.1：一个简单的辞典

辞典是一个同义词字典。例如，这里是一个小辞典，每个单词后面跟着它的同义词：

close near confined

confined cramped

correct true

cramped confined

near close

one singular unique

singular one unique

true correct

unique singular one

需要解决的问题是：给定一个辞典文件以及从键盘输入的单词，输出键入的每个单词的同义词。

分析

辞典文件将是按照字母顺序排列的。对键入的每个单词，如果该单词的同义词在辞典文件中就显示它的同义词。否则，将显示一条错误消息。

系统测试（输入用黑体表示） 辞典文件同上。

```
Please enter a word; the sentinel is ***
one
Here are the synonyms:
singular unique

In the input line, please enter a word; the sentinel is ***
two
The word is not in the thesaurus.

Please enter a word; the sentinel is ***
close
Here are the synonyms:
near
confined

Please enter a word; the sentinel is ***
***
```

436

编程项目10.2：创建一个词汇索引

给出一个文本，为文本中的每个单词开发一个词汇索引。**词汇索引**由文本中的每个单词和该单词每次出现的行号组成。包括对所有方法的大O时间估算。

分析

1) 输入的第一行将包含到文本文件的路径，输入的第二行包含到输出文件的路径。

2) 文本中的每个单词只由字母组成——部分或全部单词可以是大写的。

3) 文本中的每个单词后面跟着0个或更多标点符号，随后是任意数量的空格和行尾标志。

4) 输出由小写且按字母顺序排列的单词组成；每个单词后面跟着它每次出现的行号。行

号之间应当以逗号分隔。

5) 文本将装入内存。

6) 文本中的行号从1开始。

7) 这个项目必须使用映射来构造词汇索引。在试验25中使用了多映射。

假设doc1.in包含下面的文本:

This program counts the
number of words in a text.
The text may have many words
in it,including big words.

另外，假设doc2.in包含下面的文本:

Fuzzy Wuzzy was a bear.
Fuzzy Wuzzy had no hair.
Fuzzy Wuzzy was not fuzzy.
Was he?

系统测试1

In the Input line, please enter the path to the text file.
doc1.in
In the Input line, please enter the path to the output file.
437 **doc1.out**

下面是程序完成后doc1.out的内容。

```
Here is the concordance:
a: 2
big: 4
counts: 1
have: 3
in: 2, 4
including: 4
it: 4
many: 3
may: 3
number: 2
of: 2
program: 1
text: 2, 3
the: 1, 3
this: 1
words: 2, 3, 4
```

系统测试2

In the Input line, please enter the path to the text file.
doc2.in
In the Input line, please enter the path to the output file.
doc2.out

下面是程序完成后doc2.out的内容。

```
Here is the concordance:
a: 1
bear: 1
fuzzy: 1, 2, 3
had: 2
hair: 2
he: 4
no: 2
not: 3
was: 1, 3, 4
wuzzy: 1, 2, 3
```

438

第11章　优先队列和堆

本章的中心是优先队列数据结构。**优先队列**是一个容器，根据一定方式为项分配优先级，其中只有优先级最高的项才能被访问和删除。有趣的是，优先队列的这个限制使得在平均情况下插入只花费常数时间，即使在最坏情况下，删除也只花费对数时间。在提供priority_queue类的方法接口之后将描述标准C++的实现。

因为priority_queue类是一个容器配接器，所以类的实现看起来可能是相当简单的，就像第7章中的queue和stack类的实现一样。但并非如此，我们将需要两个通用型算法——push_heap和pop_heap，这就带来了有关堆的探讨。**堆**是一个完全二叉树，其中的每个项都大于等于它的子女。最后将开发一个优先队列在数据压缩领域内的应用。特别是，**霍夫曼树**将信息编码成压缩的形式以节约信息传送的时间。

目标

1) 定义优先队列。
2) 理解push_heap、pop_heap和make_heap的堆操作。
3) 比较各种优先队列数据结构的不同实现的平衡折中。
4) 能够用霍夫曼算法进行数据压缩。
5) 确定贪心算法的特性。

11.1　介绍

队列的变体——优先队列，是一个普通的结构，它的基本思想是让项排队等候服务。选择的依据并非严格遵循先来先服务机制。例如，病房中的病人是根据受伤严重程度而不是他们的到达时间来对待的。同理，在航运管理中，往往有一队飞机等待着陆，但是如果某架飞机缺油或是载有生病的乘客，那么管理者可以将它移到队列的前面。

网络上的共享打印机是适合使用优先队列的另一个例子。正常情况下，各个任务按照到达时间进行打印，但是一旦某一任务正在打印，那么其他的几个任务将进入服务队列。最高优先级将被赋予打印页数最少的任务，这将优化完成任务的平均时间。同样，按优先级别服务的观点适用于任何共享资源：中央处理器，家用汽车，下一学期提供的课程，等等。

下面是它的定义：

> **优先队列**是一个容器，根据一定方式为项分配优先级，其中只有优先级最高的项才能被访问和删除。

例如，假设项是整数，而且当$i>j$时，整数i比j具有更高的优先级；那么最大的项就具备最高的优先级。这个定义没有明确说明在**哪里**进行插入。明了项的插入位置是优先队列开发者的职责，而不是优先队列用户的职责。

优先队列是**公平的**，如果在优先队列中有两项的优先级别相同，那么在队列中滞留时间长的项将先从队列中删除。

如果有两个或更多的项都是最高优先级会怎样？为了公平起见，这个约束应当有利于在优先队列中滞留时间最长的项。这个公平性的请求不是定义的一部分，也不是priority_queue类的一部分。实际上，就像即将在试验26中看到的，在priority_queue类的标准实现中这些约束并没有被公平地处理。试验26提供了对这个问题的解答。

本章主要致力于优先队列的一个首要的应用：霍夫曼编码。在第14章中，优先队列被用于两个图算法中：Prim的最小生成树算法和Dijkstra的最短路算法。要进一步探讨优先队列的多功能性，请参阅Dale(1990)。

11.1.1节中通过提供priority_queue类中方法的方法接口来定义该类。

11.1.1 priority_queue类

priority_queue类声明的开头如下：

```
template<class T, class Container = vector<T>,
        class Compare = less<Container::value_type> >
class priority_queue
{
```

440 priority_queue类是一个容器配接器。

第一个模板参数T是优先队列中存储的项的类型。Container模板参数代表的是基础容器类，它的方法接口与priority_queue类**相配接**。相应模板变元类的要求是要包含方法front、push_back和pop_back，并且支持随机访问迭代器。vector或deque类满足了这些要求，而且缺省类是vector类。（为什么不能采用list类？）和Compare参数对应的模板变元必须是一个函数类，它的**operator**()对value_type进行比较，这和作为T几乎完全相同。缺省变元是内置函数类less，因此值**最大**的项将放在priority_queue对象的前面。

缺省情况下优先级最高的项就是值最大的项。

在priority_queue类中只有七个方法。下面是方法接口：

1. //后置条件：用x和一个容器对象的拷贝初始化这个优先队列。
```
explicit priority_queue(const Compare& x=Compare(),
                        const Container&=Container());
```

例 可以定义两个优先队列如下：

```
pq<employee> e_pq1();
pq<employee> e_pq2(e_pq1);//e_pq2包含了e_pq1的一个拷贝。
```

2. //后置条件：通过Compare对象x和基础容器对象y，这个优先队列被初始化为
//　　从位置first（包括在内）到last（不包括在内）之间的项的拷贝。
```
template<class InputIterator>
priority_queue(InputIterator first, InputIterator last,
               const Compare& x=Compare(),
               const Container& y=Container());
```

3. //后置条件：返回这个优先队列中项的数量。
```
size_type size() const;
```

4. //后置条件：如果这个优先队列中没有项，那么返回真。否则返回假。
```
bool empty() const;
```

5. //后置条件：x被插入到这个优先队列中。averageTime(n)是常数，而
   //　　worstTime(n)是O(n)。
   void push(**const** value_type& x);

6. //前置条件：这个优先队列非空。
   //后置条件：发送这个消息之前队列中优先级别最高的项被删除。worstTime(n)是
   //　　O(logn)。
   void pop();

7. //前置条件：这个优先队列非空。
   //后置条件：返回对这个队列中优先级别最高的项的常量引用。
   const value_type& top() **const**;

441

注意　因为**const**是返回类型规格说明的一部分，所以修改priority_queue对象的顶部项是非法的。这样的修改可以改变顶部项的优先级。

下面的小程序创建并维护了两个优先队列，一个是字母顺序的string项，另一个是递减顺序的**int**项。

```
#include <vector>
#include <queue> //  定义  priority_queue 类
#include <iostream>
#include <string>

using namespace std;

int main( )
{
    const string WORDS = "Here are the words in alphabetical order:";

    const string SCORES = "Here are the scores in descending order:";

    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    priority_queue<string, vector<string>, greater<string> > words;
    priority_queue<int> scores; // 使用缺省的第二个和第三个
                                // 模板变元

    words.push ("yes");
    words.push ("no");
    words.push ("maybe");
    words.push ("wow");

    cout <<WORDS <<endl;
    while (!words.empty( ))
    {
        cout <<words.top( ) <<endl;
        words.pop( );
    } // 显示并弹出单词

    scores.push (50);
    scores.push (71);
    scores.push (65);
```

442

```
        scores.push (57);
        scores.push (60);

        cout <<endl <<endl <<SCORES <<endl;
        while (!scores.empty( ))
        {
            cout <<scores.top( ) <<endl;
            scores.pop( );
        } // 显示并弹出分数

        cout <<endl <<endl <<CLOSE_WINDOW_PROMPT;
        cin.get( );

        return 0;
    } // main
```

输出是:

```
Here are the words in alphabetical order:
maybe
no
wow
yes

Here are the scores in descending order:
71
65
60
57
50

Please press the Enter key to close this output window.
```

现在将注意力重新转到priority_queue类的字段和实现上。通过配接另一个容器类的方法接口可以大大简化工作。但是困难的工作是将一种新型的树结构，称作堆，用于像向量或双端队列之类的基于数组的容器。

11.1.2 priority_queue类的字段和实现

就像queue和stack类一样，priority_queue类的方法定义是标准C++的一部分。根据标准C++，priority_queue类的定义是在<queue>（所指示的文件）中，但是对于惠普的实现，该定义是在<stack>（所指示的文件）中。

443 priority_queue类里的两个字段分别是一个容器对象c和一个函数对象comp:

```
protected:
    Container c;
    Compare comp;
```

缺省情况下，容器c将是vector类的一个实例。函数对象comp将实例化某些内置或用户定义的函数类，缺省情况下是less类。

和读者所想的一致，top()方法将返回c.front()所返回的引用，但是让人困惑的是pop()方法

调用了c.pop_back()来删除优先级最高的项。那么该项是在前面还是后面呢？当看到pop()方法定义[⊖]时更加深了这个谜团：

```
void pop( )
{
    pop_heap (c.begin( ), c.end( ), comp);
    c.pop_back( );
} // pop
```

因此实际上是一个pop_heap通用型算法[⊜]在幕后控制着容器c。随后对pop_back方法的调用再完成最后的任务。在pop_heap方法里完成了pop方法的比较困难的工作。同样，push方法的代码是：

```
void push (const value_type& x)
{
    c.push_back(x);
    push_heap(c.begin( ), c.end( ), comp);
} // push
```

这里读者可能会猜测，通用型算法push_heap和pop_heap使用了堆，但是什么是堆呢？11.1.3节中将揭示这些内容。

11.1.3 堆

堆的递归定义。

堆t是这样一个完全二叉树，它或者为空，或者满足：

1) 根据项之间的某些比较方法，根项是t中最大的项。

2) leftTree(t)和rightTree(t)都是堆。

图11-1显示了一个包含10个**int**项的堆。

堆中的顺序是自顶向下的，但不是自左向右的。

堆中的顺序是自顶向下的，但不是自左向右的：每个根项都大于等于它的每个子女，但是有些左兄弟大于它们的右兄弟，而有些则小于它们的右兄弟。例如在图11-1中，85比它的右兄弟48大，但是30就小于它的右兄弟36。 444

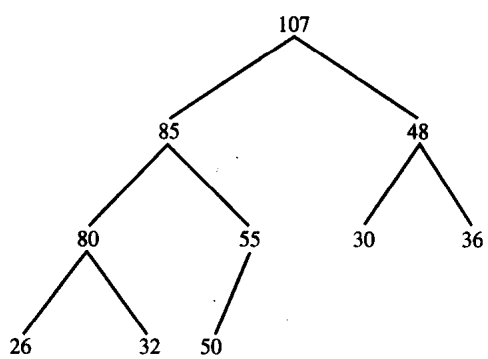


图11-1 包含10个**int**项的堆

⊖ 从技术上说，标准只是规定了定义得到的结果必须和定义中说明的结果一致。

⊜ 在惠普实现的<heap>里对这个算法进行了定义，但是在标准C++中是在<algorithm>里定义的。

堆是一个奇怪的结构，但是它在多种应用中却很有用。当然，其中一种就是priority_queue类的实现。我们将关注两个主要的堆操作——push_heap和pop_heap。另外还将提及另一个操作——make_heap，它从头创建一个堆，在调用priority_queue构造器时将调用该操作。在第12章中还将看到第四个也是最后一个堆操作——sort_heap，它的主要作用是将一个堆创建成一个排序容器。

堆是一个完全二叉树。正如在第8章中看到的，对于完全二叉树中的每一项，都可以为它关联一个0...$n(t)-1$之间的位置。如果将这些位置看作索引，那么堆中的项就可以存储在一个数组中。例如，下面是图11-1所示的堆的数组表示：

107	85	48	80	55	30	36	26	32	50	...

任何支持随机访问迭代器的容器——如数组、向量或双端队列都可以充当一个堆。

数组的随机访问特点便于堆的处理：给定某一项的索引，就可以快速地访问该项的子女。例如，索引i处项的子女位于索引$2i+1$和$2i+2$，而索引j处项的父亲位于索引$(j-1)/2$。不久将看到，堆可以快速地从父亲移动到它的子女的位置上，反之亦然，这个能力使得堆成为实现优先队列的一个有效的存储结构。push_heap方法的接口是：

```
//前置条件：从first（包括在内）到last-1（不包括在内）之间的项是一个堆。
//后置条件：位于last-1上的值被插入到堆中。
template<class RandomAccessIterator, class Compare>
inline void push_heap(RandomAccessIterator first,
                      RandomAccessIterator last, Compare comp);
```

445

即将插入的项，也就是last-1位置上的项，被存储在临时变量value中，而腾出的位置（其中仍然包含着即将插入的项）被称作是**陷阱**。从技术的角度讲，使用了变量holeIndex；它包含了当前陷阱在容器中的索引。例如，图11-2显示了在调用push_heap方法插入90时图11-1中堆的变化。

要了解当90存储在陷阱中时是否需要维护堆，将90和陷阱父亲——索引为(hole_index-1)/2上的项55进行比较。因为90>55，所以将55移动到陷阱处并把holeIndex移动到55原先的位置上，参阅图11-3。继续循环，直到到达堆的顶部或是找到一个位置使得90 < 陷阱父亲上的项。然后将90插入陷阱所在的位置，最后得到的堆如图11-4所示。

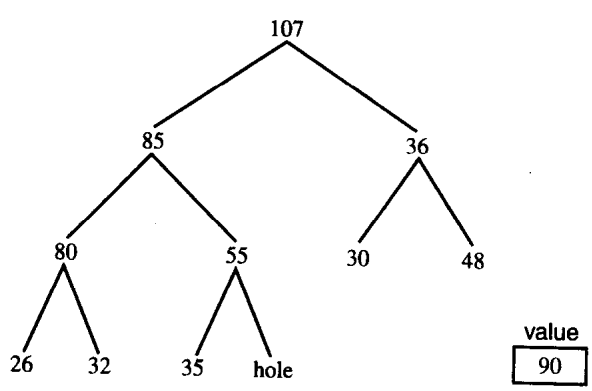


图11-2 push_heap方法开始时图11-1中的堆。即将被插入的项是90

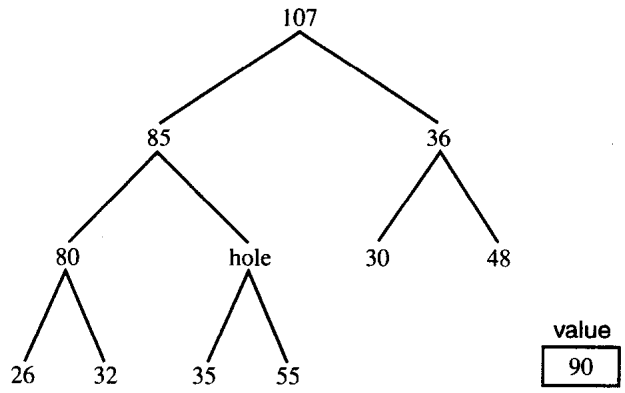


图11-3 对图11-2中使用push_heap方法将90和55比较之后的堆 446

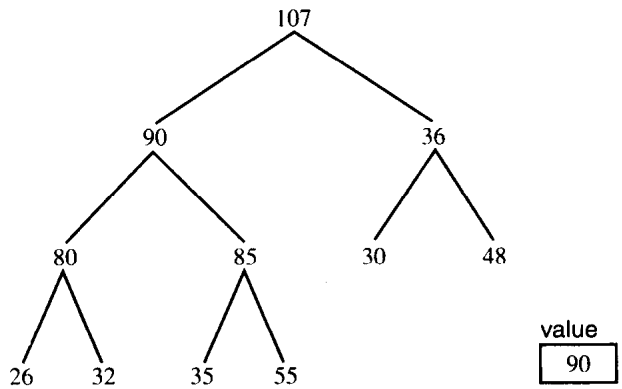


图11-4 在图11-1中调用push_heap方法插入90之后的堆

下面是__push_heap方法中最基本的代码（把Distance看作是**int**的同义字）：

```
Distance parent = (holeIndex - 1) / 2;
while (holeIndex > topIndex && comp (*(first + parent), value))
{
      *(first + holeIndex) = *(first + parent);
      holeIndex = parent;
      parent = (holeIndex - 1) / 2;
} // while
*(first + holeIndex) = value;
```

在最坏情况下，将要被插入的项大于first位置上的项。这时**while**循环的迭代次数与树的高度成正比。而完全二叉树的高度和n成对数关系（见习题8.7）。也就是说，worstTime(n)和n成对数关系。

在平均情况下，堆中大约有一半的项比即将插入的项小，另一半则大于它。但是堆是非常浓密的：至少有一半的项是树叶。又因为堆的性质，大部分比插入项小的项将位于树叶层或接近树叶层的位置。实际上，循环迭代的平均数量小于3（参见Schaffer和Sedgewick，1993），这就满足了后置条件的要求：averageTime(n)是常数。

针对priority_queue类中的push方法的定义，averageTime(n)是常数。

现在可以分析push方法了。当堆满时就进入了最坏情况。这时采取的行动依赖于基础容器类——例如，向量或双端队列。无论如何将需要n的线性关系次拷贝，因此worstTime(n)和n成线性关系。但是这个拷贝并不是频繁发生的，每n次插入发生一次，因此averageTime(n)是
447 常数（实际上，amortizedTime(n)是常数），这和push_heap方法是相同的。

pop_heap方法的接口如下：

```
//前置条件：堆非空。
//后置条件：从堆中删除last-1位置上的项。
template<class RandomAccessIterator, class Compare>
inline void pop_heap(RandomAccessIterator first,
                     RandomAccessIterator last, Compare comp);
```

迭代器last位于紧随堆底之后的位置。pop_heap算法恰好与push_heap方法相反：pop_heap自堆顶向下运行。顶部的项是即将被删除的项，它将被存储到索引是last-1的位置上。（将保存该项是为了简化sort_heap方法。）因此将last-1位置上的项暂时移到变量value里。最初时陷阱是堆的顶部位置。图11-5显示了将pop_heap方法应用到图11-4所示的堆上时的初始设置。

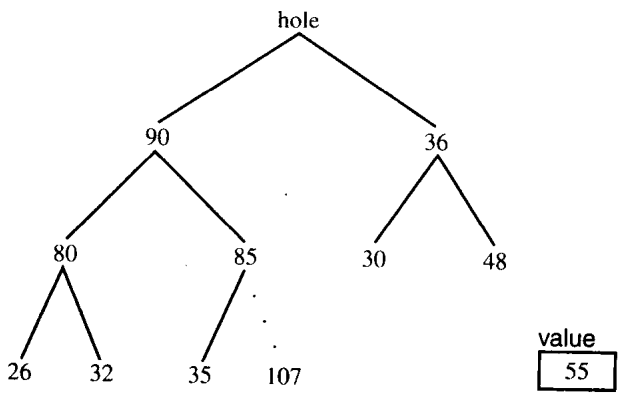


图11-5 将pop_heap方法应用到图11-4所示的堆上时的初始情况

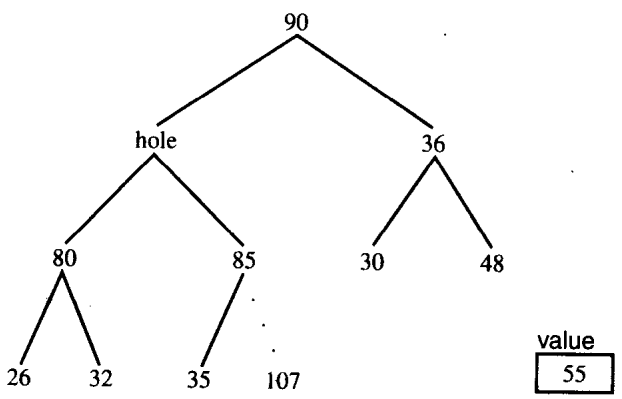


图11-6 将陷阱移动到90占据的位置之后图11-5中所示的堆的情况

项107被存储在超过堆底部的位置，但是仍需要找出55（现在存在value里）可以存放的位置。不用在55和其他项之间比较，可以先通过陷阱项子女间的相互比较将陷阱下移。较大的子女被移动到陷阱占据的位置，而陷阱则被设置成这个保存较大子女的位置。图11-6显示了

将陷阱下移到90原先占据的位置之后的堆。再进行两次循环迭代之后，陷阱下沉到树叶层，如图11-7所示。注意107不在比较之列，因为107不再被看作是堆的一部分。

这样还没有彻底完成，因为55不能被插入到陷阱的位置。因此现在调用push_heap方法插入55。图11-8显示了最终得到的树，不过弹出的值107不是堆的一部分。 448

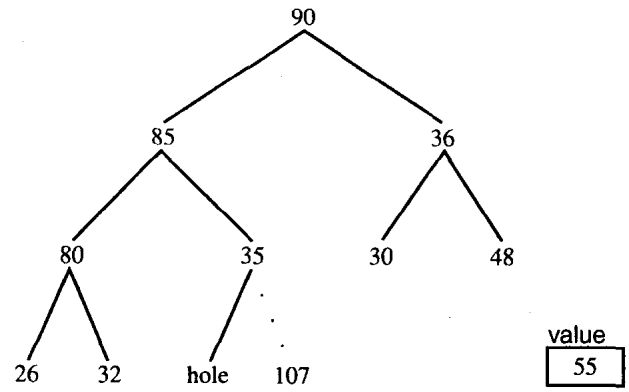


图11-7 将陷阱移动到35占据的位置之后图11-6中所示的堆的情况

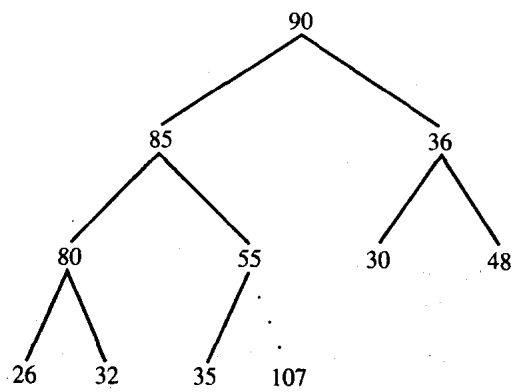


图11-8 对图11-4中的堆调用pop_heap方法之后的情况。注意107不是堆的一部分

pop_heap方法的真正的工作是在辅助方法——__adjust_heap——里完成的。下面是__adjust_heap方法的定义：

```
template <class RandomAccessIterator, class Distance, class T,
          class Compare>
void __adjust_heap (RandomAccessIterator first, Distance holeIndex,
                    Distance len, T value, Compare comp)
{
    Distance topIndex = holeIndex;
    Distance secondChild = 2 * holeIndex + 2;
    while (secondChild < len)
    {
        if (comp(*(first + secondChild), *(first + (secondChild - 1))))
            secondChild--;
        *(first + holeIndex) = *(first + secondChild);
```

```
            holeIndex = secondChild;
            secondChild = 2 * (secondChild + 1);
        }
        if (secondChild == len)
        {
            *(first + holeIndex) = *(first + (secondChild - 1));
            holeIndex = secondChild - 1;
        }
        __push_heap(first, holeIndex, topIndex, value, comp);
    }
```

pop_heap要花费多长时间？循环将陷阱下移到树叶层需要的时间和树的高度是成比例的。堆是一个完全二叉树，因此它的高度总是和n成对数关系。另外的push_heap调用将花费常数时间（平均情况下）到对数时间（最坏情况下，因为不需要调整大小）。因此对pop_heap方法而言，worstTime(n)和n成对数关系，averageTime(n)也是如此。

我们需要在父亲和子女之间便捷地移动。使用随机访问迭代器（等价于数组索引）可以很容易地实现这一点。红黑树也可以很容易地在父亲和子女之间移动，但是它的代价是需要额外的空间：存储在节点中的每一项都需要parent、color、left和right字段。相比之下堆是很出色的：每个项都不需要额外的空间。

最后考虑make_heap算法，它通过支持随机访问迭代器的容器来创建一个堆。方法接口是：

```
//后置条件：first（包括在内）迭代器到last（不包括在内）迭代器之间
//的项形成一个堆。
template<class RandomAccessIterator, class Compare>
inline void make_heap(RandomAccessIterator first,
                      RandomAccessIterator last, Compare comp);
```

449 ~ 450

first项就是堆自身，因此可以简单地循环通过容器的其余部分并在每次迭代中调用push_heap：

```
itr=first++;
while(itr!=last)
    push_heap(first,itr++,comp);
```

但是在一个堆中有一半的项是树叶，并且每个树叶能自动成为一个堆。因此对push_heap的调用有一半都是浪费。

可以代替的方法是从项不是树叶的最高索引处开始构造堆。例如在图11-9中，从项30的索引处开始。除了陷阱索引处的项，即30将被临时存放在变量value里不同外，这个调整与对pop_heap进行的调整很相似。在35向上移动到陷阱并且陷阱下移到35原先的位置之后，调用push_heap将30插入陷阱。图11-10显示了成功插入30之后树的外观。接着移动到容器中较低的索引，下一个要调整的是以40为根的子树，然后调整以70为根的子树。这时得到的树如图11-11所示。

451

为了使以50为根的子树满足堆属性，陷阱选择在50所占据位置的索引，并将50存入变量value。将55移进陷阱之后，陷阱被移动到55原先占据的位置上。用push_heap方法将50插入这个新陷阱开始的路径之后，从整个树的根开始进行最后的堆调整。调整之后得到的堆如图

11-12所示。

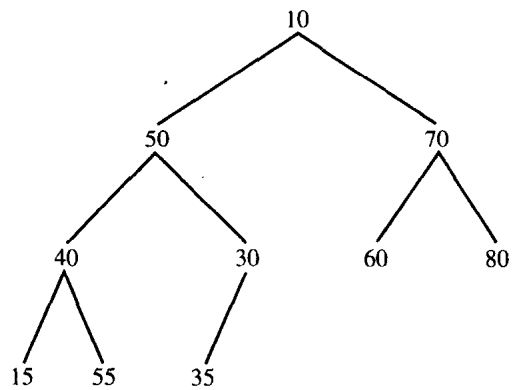


图11-9 调用make_heap方法之后将成为堆的一个完全二叉树

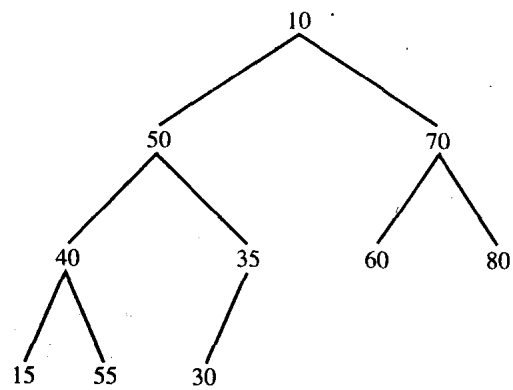


图11-10 对图11-9的完全二叉树进行调整，使以30为根的子树满足堆属性之后的情况

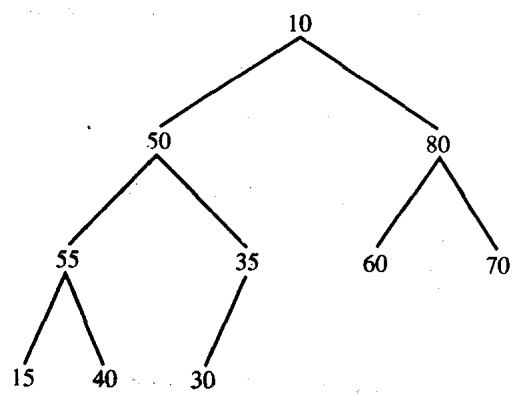


图11-11 对图11-10的完全二叉树进行调整，使以40和70为根的子树都满足堆属性之后的情况

下面是基本的代码（__adjust_heap被pop_heap调用；__adjust_heap的第二个变元是最初的陷阱索引）：

```
if (last - first < 2)
    return;
```

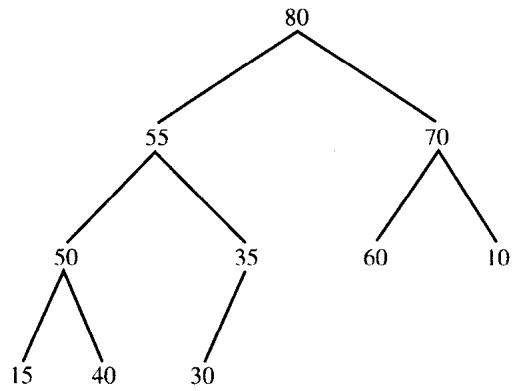


图11-12 对图11-11中的完全二叉树进行调整，使根为50和10的子树满足堆属性之后的情况

```
Distance len = last - first;
Distance parent = (len - 2) / 2;

while (true)
{
    __adjust_heap (first, parent, len, T(*(first + parent)), comp);
    if (parent == 0)
        return;
    parent--;
} // while
```

make_heap方法的分析略微有些困难，但结果是，不论最初的容器中项是如何排列的，worstTime(n)总是和n成线性关系。下面是一些详细情况。大约要调整$n/2$个堆，而且每次调整花费的时间和从堆的根到它最远树叶的距离成比例。当根的索引i从$n/2$下降到0时，距离近似为$\log_2(n/(i+1))$。很荒谬的是，最坏情况出现在容器已经是一个堆时。因为这时的每次调整都需要沿着树的一条路径下降寻找树叶陷阱，随后当调用push_heap方法时再沿着相同的路径上升。因此最坏情况下的总迭代次数近似为:

$$\sum_{i=1}^{n/2} 2\log_2(n/i)$$

令$m=n/2$。回忆一下，商的对数等于对数的差，因此总和等于:

$$2(m\log_2 n - \sum_{i=1}^{m}\log_2 i)$$

也就等于

$$2(m\log_2 n - \log_2(\prod_{i=1}^{m} i)) = 2(m\log_2 n - \log_2(m!))$$

对数的总和是乘积的对数。(参阅附录A1.3中累乘表示法的说明。)

根据Stirling的阶乘近似的对数形式，

$$\begin{aligned}\log_2(m!) &\approx \log_2((m/e)^m)\\ &= m\log_2 m - m\log_2 e\\ &\approx m\log_2 m - 1.5m\end{aligned}$$

因为$\log_2 m=\log_2(n/2) = \log_2 n - \log_2 2 = \log_2 n-1$，

$$\begin{aligned}2(m\log_2 n - \log_2 m!) &\approx 2(m\log_2 n - (m\log_2 m - 1.5m))\\ &= 2(m\log_2 n - m(\log_2 n-1) +1.5m)\\ &= 2(m+1.5m)\\ &= 5m\\ &= 2.5n\end{aligned}$$

453

所以worstTime(n)和n成线性关系。平均情况下需要进行约一半次数的迭代，因此averageTime(n)也和n成线性关系。

现在通过迂回的方法揭示了priority_queue类的实现细节，然后可以更详细地研究“公平性”问题。回想一下，优先队列是公平的，当两个项具有相同的优先级，那么插入早的项将先被访问或删除。

实验26：优先队列中的公平性 （所有实验都是可选的）

11.1.4 priority_queue类的另一种设计及实现

在结束对priority_queue类的设计和实现之前，再看一下能够代替堆的其他方案。正如前面讨论过的，标准C++需要用堆来实现priority_queue类；但也很容易得到一些其他的实现，并且它们可以提供和堆相当的作用。

一种实现是基于列表的。惟一的字段是：

```
list<value_type> c;
```

基本思想是将优先队列中的项按照降序存储到容器c中，因此c的前面保存了高优先级的项。top和pop方法只不过是分别调用了c.front()和c.pop_front()，因此这些方法的worstTime(n)是常数。这点比堆实现强，在堆版本中它们分别花费常数和对数时间。

push方法也是单行的，但更复杂些。回忆在实验11中，upper_bound通用型算法返回一个迭代器，它位于容器中能够插入且不违背顺序关系的最后一个位置。这说明了项应当被插入c里的位置：push方法根据项的相对值将它插入进c中对应的最后一个位置。

```
void push (const value_type& x)
{
    c.insert (upper_bound (c.begin( ), c.end( ), x, greater<value_type>), x);
} // 方法push
```

当upper_bound方法被应用到只支持双向（不是随机访问）迭代器的类上时，它花费n的线性时间。list类的insert方法只花费常数时间，因此push的worstTime(n)和n成线性关系，这和push方法的堆版本是相同的。但是这个push版本的averageTime(n)也和n成线性关系，而堆版本则花费常数时间。又因为插入项总是存储在和它优先级相同的项之后，因此这个实现是公平的。 454

另一种有意义的实现使用了set容器，其中的项按照降序存储：

```
set<value_type, greater<value_type>> c;
```

同样，top、push和pop方法都是单行的：

```
const value_type& top ( ) const
{
    return *(c.begin( ));
} // 方法top

void push (const value_type& x)
{
    c.insert (x);
} // 方法push

void pop( )
{
    c.erase (c.begin( ));
} // 方法pop
```

根据第10章中对set方法的分析，priority_queue类的这个实现中的top方法需要常数时间。push和pop方法的worstTime(n)则和n成对数关系。

表11-1总结了时间估算。当然，标准C++中只有基于堆的实现才是合法的，但是其他的实现也各有各的优点。

表11-1 对priority_queue类的三种实现，分别比较它们的top、push和pop方法的averageTime(n)和worstTime(n)（用分号分隔）

实现	top	push	pop
堆	$O(1);O(1)$	$O(1);O(n)$	$O(\log n);O(\log n)$
列表	$O(1);O(1)$	$O(n);O(n)$	$O(1);O(1)$
集合	$O(1);O(1)$	$O(\log n);O(\log n)$	$O(\log n);O(\log n)$

455 11.2节介绍了优先队列的一个重要的应用：数据压缩。

11.2 优先队列的应用：霍夫曼编码

假设有一个大的信息文件，如果能够在不丢失任何信息的前提下将文件压缩，节约空间，这将是非常有利的。更重要的是如果用压缩的信息代替原始信息，那么传送信息花费的时间也将大大减少。

为了简明性和具体性，假设消息文件M包含100 000个字符，并且每个字符都是“a”，“b”，“c”，“d”，“e”，“f”或者“g”。因为共有七个字符，所以如果希望为每个字符使用相同数量的位，那么可以用ceil($\log_2 7$)即3个位对每个字符进行惟一的编码[⊖]。例如，可以为“a”采用000，为“b”采用001，为“c”采用010，依次类推。像“cad”这样的单词可以编码成010000011。那么编码文件E只需要300 000个位，再加上额外的几个表示编码信息自身的位：“a”=000，等等。

还可以减少一些字符的位的数量以节约空间。例如，可以使用下面的编码：

⊖ ceil(x)返回大于等于x的最小整数。例如，ceil(17.2)=18。

a=0

b=1

c=00

d=01

e=10

f=11

g=000

这大约能把编码文件E的大小减少三分之一（除非字符“g”出现得非常频繁）。但是这个编码可能导致二义性。例如，位序列001可以被解释成“ad”或者“cb”或者“aab”，这取决于是把前两位作为一组，还是后两位作为一组，或是单独地对待每一位。

这个编码方案产生二义性的原因是有些编码是其他编码的前缀。例如，0是00的前缀，因此不可能确定00究竟表示“aa”还是“c”。通过令编码**无前缀**可以避免二义性，也就是说，任何编码都不是其他编码的前缀。

无前缀编码是没有二义性的。

一种保证编码无前缀位的方法是创建一个二叉树，其中左边的树枝解释为0，右边的树枝解释为1。如果每个编码的字符是树中的一个树叶，那么该字符的编码就不会是任何其他字符编码的前缀。换句话说，到每个字符的路径就提供了无前缀编码。例如，图11-13里有一个二叉树，说明了从字符“a”到字符“g”的无前缀编码。 456

为了获取一个字符的编码，二叉树的根从空编码开始，然后继续，直到到达要被编码的树叶。在循环中，当转向左边时就在编码中添加0，转向右边时就在编码中添加1。例如，“b”被编码成01，“f”被编码成1110。由于每个被编码的字符是一个树叶，所以该编码是无前缀的，因此也无二义性。但是还不能确定它能否节约空间或传送时间。这完全依赖于每个字符出现的频率。因为有三个编码占2位，四个编码占4位，所以这个编码机制实际上比简单的每个字符3位的编码方式要占据更多的空间。

这暗示着如果已知每个字符的频率并根据这些频率构造编码树，将可能节约大量的空间。使用字符频率确定编码是**霍夫曼编码**（Huffman，1952）的基本思想。霍夫曼编码是一种无前缀编码策略，它保证在无前缀编码中是最优的。霍夫曼编码是Unix的compress实用程序的基础，也是联合图像专家组（Joint Photographic Experts Group，JPEG）编码处理的一部分。

霍夫曼编码利用了消息中每个字符出现的频率。

从计算一个给定消息M中每个字符的频率开始。注意这些计算的时间和M的长度成线性关系。例如，假设M中的字符是字母“a”到“g”，并且它们的频率如下：

a: 5000

b: 2000

c: 10,000

d: 8000

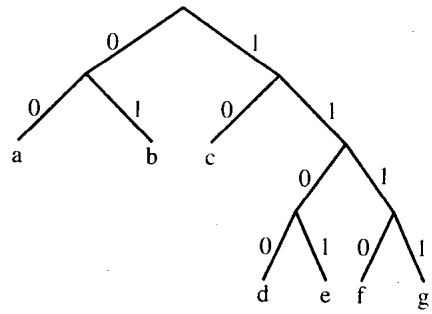
e: 22,000

f: 49,000

g: 4000

M的大小是100 000个字符。如果忽略频率，并将每个字符编码成惟一的三位序列，那么

共需要300 000个位来编码消息M。马上我们将看到这和最优编码相差多远。



457

图11-13 产生从字母“a”到字母“g”的无前缀编码的二叉树

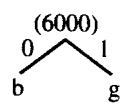
按照频率的升序把“字符-频率”对一个接一个地插入到优先队列中。

一旦计算出每个字符的频率，就将按照频率的升序将“字符-频率”对一个接一个地插入到优先队列中。因此优先队列中顶部的“字符-频率”对将包含出现频率最低的字符。最初得到的优先队列是:

(b:2000)(g:4000)(a:5000)(d:8000)(c:10000)(e:22000)(f:49000)

事实上，我们对这个优先队列的全部认识是: 访问和删除操作的对象是(b:2000)对。作为priority_queue类的用户，不应当假定采取堆的实现形式（即使这是标准C++中的当前实现）。所显示的对按照升序排列只是为了简化。

下面将自底向上地创建一个二叉树，称作**霍夫曼树**。先从优先队列中进行两次弹出操作，获得两个频率最低的字符（也就是优先级最高的）。第一个弹出的字符“b”成为二叉树最左边的树叶，“g”成为最右边的树叶。它们俩的频率之和记作树的根，并插入到优先队列中。现在得到霍夫曼树:

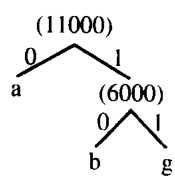


优先队列现在包含:

(a:5000)(:6000)(d:8000)(c:10000)(e:22000)(f:49000)

从技术上说，优先队列和霍夫曼树由“字符-频率”对组成; 但是当两个频率求和时字符可以省略，而且在显示霍夫曼树的树叶时可以省略频率。实际上，算法操作的是对的引用而不是对自身。这使得可以用引用表示典型的二叉树结构——左，右，根，父亲; 这是遍历通过霍夫曼树所需要的。

当从优先队列中弹出对(a:5000)和(:6000)时，它们成为扩展得到的树的左子树和右子树，该树的根是它俩的频率之和。这个和被插入到优先队列里。现在得到的霍夫曼树是:



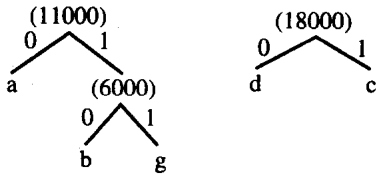
优先队列现在包含:

(d:8000)(c:10000)(:11000)(e:22000)(f:49000)

注意在每次迭代中弹出出现频率最低的字符的意义。它们最终被放在霍夫曼树的底部，使用最长的编码。出现频率最高的字符，如“f”，最终将放在接近霍夫曼树根的位置，并使用最短的编码。这就是霍夫曼编码最小的原因。 458

出现频率最低的字符最终将远离霍夫曼树的根。

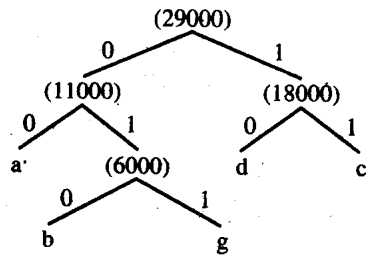
当弹出“d”和“c”时，它们不能连接到主干树上，因此它们就成为另一个树的左分枝和右分枝，而该树的根——它们的总和——被插入到优先队列中。这样得到了两个临时的霍夫曼树:



优先队列现在包含:

(:11000)(:18000)(e:22000)(f:49000)

当弹出对(:11000)时，它成为二叉树左边的树枝，而右边的树枝是下一个弹出的对(:18000)。它俩的和成为这个二叉树的根，并被插入到优先队列中，因此得到霍夫曼树:



优先队列现在包含:

(e:22000)(:29000)(f:49000)

当弹出下两个对时，“e”成为霍夫曼树左边的树枝，而(:29000)成为右边的树枝，根(:51000)被插入到优先队列中。最后弹出两个对(f:49000)和(:51000)分别成为最后的霍夫曼树的左右树枝。这两个频率的和是根的频率(:100000)，它将作为惟一的项插入到优先队列中。最终得到的霍夫曼树如图11-14所示。霍夫曼编码是:

a:1100

b:11010

c:1111

d:1110

e:10

f:0

g:11011 459

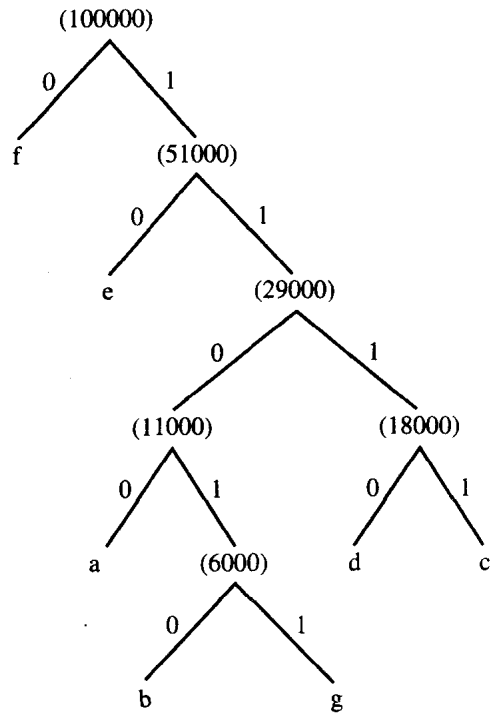


图11-14 最后的霍夫曼树

现在把消息M翻译成编码消息E是很容易的。接收端怎么样呢？能不能很容易地将E解码成原始消息M？从树根和E的开头起，为E中的0选取左树枝，为1选取右树枝，继续这样直到到达一个树叶，这就是M中的第一个字符。再返回树的顶部并继续读E。例如，如果M是“cede”，那么E就是111110111010。从树根开始，E开头的四个1将通向树叶“c”。然后返回树根并继续读E，读入第5个1。为这个1选取右树枝，然后为0选取左树枝，到达“e”。依次类推。

消息E的大小等于M中所有字符对应的字符编码位数与该字符频率的乘积之和。因此要得到这个例子中E的大小，需要将“a”编码中的位数4和“a”出现的频率5000相乘，加上“b”编码中的位数5和“b”出现的频率2000相乘，等等。得到

$$(4 \times 5000) + (5 \times 2000) + (4 \times 10\,000) + (4 \times 8000) + (2 \times 21000)$$
$$+ (1 \times 49\,000) + (5 \times 4000) = 213\,000$$

这大约比固定长度的每个字符3位的编码方式少百分之三十，因此显著地节约了空间和传送时间。但是也应当注意到，固定长度的编码一般要比霍夫曼编码解码快很多：例如，编码位可以解释成数组索引——该索引上的条目就是编码对应的字符。

460

11.2.1 huffman类的设计

要了解霍夫曼树的本质，现在来开发一个huffman类处理消息的编码。消息将放在一个输入文件中，而编码消息将发送给一个输出文件。构造器初始化huffman对象（包括输入和输出文件），给定路径名：

```
//后置条件：根据in_file_name和out_file_name初始化这个huffman对象。
huffman(string in_file_name, string out_file_name);
```

另一个要执行的任务是很容易确定的：需要创建优先队列和霍夫曼树，计算霍夫曼编码，最后将霍夫曼编码和编码后的消息保存到输出文件中。方法接口是：

```
//后置条件：创建优先队列。worstTime(n)是O(n)。
void create_pq();

//后置条件：创建霍夫曼树。
void create_huffman_tree();

//后置条件：计算霍夫曼编码。
void calculate_huffman_codes();

//后置条件：将霍夫曼编码和编码后的消息存储到输出文件中。worstTime(n)
//          是O(n)。
void save_to_file();
```

霍夫曼树中的每个节点都包含一些编码信息（字符、它的频率和它的编码）以及一些二叉树信息（指向左子树、右子树和父亲的指针）。huffman_node类不会嵌入到huffman类中，这样其他的类，比如解码一个编码消息的类，也可以访问huffman_node类。

下面是huffman_node类，它的字段全部是公有的并且没有方法：

```
struct huffman_node;

typedef huffman_node* node_ptr;

struct huffman_node
{
    char id;
    int freq;
    string code;
    node_ptr left,
             right,
             parent;
}; // huffman_node
```

461

huffman类有一个嵌入的函数类——compare。当然，compare类没有字段，并且只有一个方法——**operator**()。

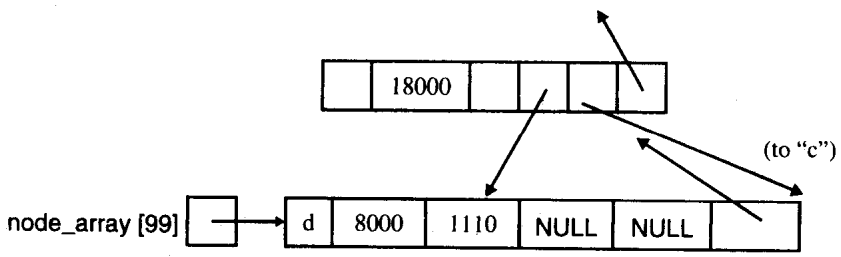
```
class compare
{
        public:
            bool operator( ) (const node_ptr& c1,
                              const node_ptr& c2) const
            {
                return (*c1).freq > (*c2).freq;
            } // 重载运算符()
} // 类compare
```

由于在比较中使用了**operator>**，所以在优先队列里频率最低的项将具有最高的优先级。

在huffman类中还应当有哪些字段？给定输入中的一个字符，我们希望能快速地访问它的huffman_node。通过创建一个索引是字符自身的数组，就可以利用数组的随机访问性质得到常数时间的访问：

```
static const int MAX_SIZE=256; //只有对静态整数常量才合法
node_ptr node_array[MAX_SIZE];
```

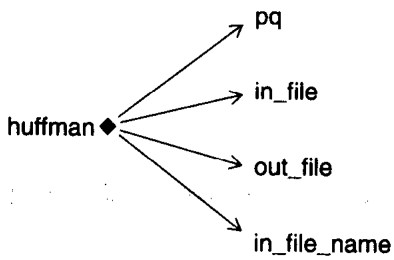
这就为ASCII字符集中的每个字符提供了一个数组位置。例如，在霍夫曼编码的例子中，“d”的频率是8000，而“d”的编码是“1110”。该字符的信息就存储在位于索引(**int**) 'd'，也就是99上的指针指向的结构**struct**中。因此node_array中会有部分如下：



还需要一个优先队列，一个输入文件，一个输出文件和一个输入文件名。输入文件名使我们可以打开输入文件两次：一次是计算频率，后一次是编码消息。因此在huffman中有如下的字段：

462
```
priority_queue< node_ptr, vector<node_ptr>, compare > pq;
fstream in_file,
        out_file;
string in_file_name;
```

在依赖关系图中，



所有的字段，即使是in_file和out_file，都表示复合，因为在huffman类之外没有对这些文件的引用。

这就完成了huffman类的设计，粗略地说是头文件huffman.h的设计。在11.2.2节中将关注源文件huffman.cpp。

11.2.2 huffman类的实现

构造器像计划地那样打开文件：

```
huffman::huffman (string in_file_name, string out_file_name)
{
    (*this).in_file_name = in_file_name;
    in_file.open (in_file_name.c_str( ), ios::in);
    out_file.open (out_file_name.c_str( ), ios::out);
} // 构造器
```

create_pq方法首先填充node_array，然后据此填充优先队列pq:

```
void huffman::create_pq( )
{
    const string DEFAULT_TOTAL =
    "With a fixed number of bits per character, the message size in bits is ";
    node_ptr entry;
    int sum = 0,
        n_chars = 0,
        n_bits_per_char;
    for (int i = 0; i < MAX_SIZE; i++)
    {
        node_array [i] = new huffman_node;
        (*node_array [i]).freq = 0;
    } // 初始化数组
    create_node_array( );
    for (int i = 0; i < MAX_SIZE; i++)
    {
        entry = node_array [i];
        if ((*entry).freq > 0)
        {
            pq.push (entry);
            sum += (*entry).freq;
            n_chars++;
        } // if
    } // 计算字符和频率
    n_bits_per_char = ceil(log (n_chars) / log(2));
    cout <<endl <<DEFAULT_TOTAL
            <<(sum * n_bits_per_char)<<endl;
} // create_pq
```

463

n_bits_per_char的计算需要解释一下。在C++中，log函数返回它的变量的自然对数（也就是以e作为底数）。应用附录1中对数的性质将底数e转换成底数2。

create_node_array方法处理了一个很有趣的问题。每次从名为in_file的输入文件中读入一行之后，就在node_array中为新行标记创建一个条目。因此输入文件中的空白行不会被忽略，但是应当如何读入一行呢？肯定不能使用

```
in_file>>line;
```

提取运算符**operator>>**开始时跳过所有的空白（包括空格和新行标记），然后不断读入，

直到遇到下一个空白。因此空行将不会被计算在内——空格也是如此。然而函数getline读入一整行并且不跳过最初的空白。下面是create_node_array方法:

```
void huffman::create_node_array( )
{
    node_ptr entry;
    string line;
    while (getline (in_file, line))
    {
        for (unsigned j = 0; j < line.length( ); j++)
        {
464         entry = node_array [(int)(line [j])];
            (*entry).freq++;
            if ((*entry).freq == 1)
            {
                (*entry).left = NULL;
                (*entry).right = NULL;
                (*entry).parent = NULL;
            } // 字符第1次出现
        } // for
        entry = node_array [(int)'\n'];
        (*entry).freq++;
        (*entry).left = NULL;
        (*entry).right = NULL;
        (*entry).parent = NULL;
    } // while
} // create_node_array
```

create_huffman_tree方法和11.2节描述的一样:

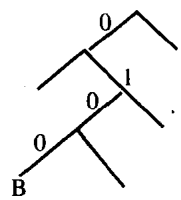
```
void huffman::create_huffman_tree( )
{
    node_ptr left,
             right,
             sum;
    while( pq.size( ) > 1 )
    {
        left = pq.top( );
        pq.pop( );
        (*left).code = string ("0");
        right = pq.top( );
        pq.pop( );
        (*right).code = string ("1");
        sum = new huffman_node;
        (*sum).parent = NULL;
        (*sum).freq = (*left).freq + (*right).freq ;
        (*sum).left = left;
```

```
            (*sum).right = right;
            (*left).parent = sum;
            (*right).parent = sum;

            pq.push( sum );
        } // while
    } // create_huffman_tree
```

465

calculate_huffman_codes方法迭代通过node_array。对每个字符频率非零的条目，按下列方式为它创建编码：从一个空string变量code开始，将条目的编码字段添加进code，然后用条目的父亲替换条目。code的最终值将作为该条目的code字段插入node_array。

例如，假设部分霍夫曼树如下：



那么‘B’的编码应当在4次迭代中计算，每次迭代后得到的编码值如下：

0

00

100

0100

这个字符串，“0100”，存储在nodeArray的索引66处的节点的code字段中，回想一下，在ASCII码序列中(**int**)'B'=66。

下面是方法定义：

```
void huffman::calculate_huffman_codes( )
{
    const string HUFFMAN_CODES = "Here are the Huffman codes: ";
    const string ENCODED_SIZE_MESSAGE =
        "\n\nThe size of the encoded message, in bits, is ";
    int total = 0;
    string code;
    node_ptr entry;
    cout <<endl <<HUFFMAN_CODES <<endl;
    for (int i = 0; i < MAX_SIZE; i++)
    {
        code = "";
        entry = node_array [i];
        if ((*entry).freq > 0)
        {
            cout <<(char)i <<" ";
            do
```

466

```
                {
                    code = (*entry).code + code;
                    entry = (*entry).parent;
                } // do
                while ((*entry).parent != NULL);

                cout <<code <<endl;
                (*node_array [i]).code = code;
                total += code.length( ) * (*node_array [i]).freq ;
            } // if
        } // for
        cout <<ENCODED_SIZE_MESSAGE <<total <<endl;
    } // calculate_huffman_codes
```

现在node_array包含了每个字符的霍夫曼编码，然后可以创建输出文件。这个文件既包含编码——例如‘a’ 0100，又包含编码后的消息。因此首先将霍夫曼编码写到输出文件里，然后重读输入文件，编码每个字符，而且将编码后的消息发送到输出文件中。下面是定义：

```
void huffman::save_to_file( )
{
    node_ptr entry;

    string line;

    in_file.close( );
    in_file.open (in_file_name.c_str( ), ios::in);

    for (int i = 0; i < MAX_SIZE; i++)
    {
        entry = node_array [i];
        if ((*entry).freq > 0)
            out_file <<(char)i <<" " <<(*entry).code <<endl;
    } // for
    out_file <<"***" <<endl;

    while (getline (in_file, line))
    {
        for (unsigned j = 0; j < line.length( ); j++)
        {
            entry = node_array [(int)(line [j])];
            out_file <<(*entry).code;
        } // for
        entry = node_array [(int)'\n'];
        out_file <<(*entry).code;
    } // while
    out_file.close( );
} // save_to_file
```


这总共将花费多长时间？令n为输入消息中的字符数量。create_node_array和save_to_file方法都迭代通过消息，因此这两个方法的worstTime(n)和n成线性关系。create_pq方法只花费

线性时间是因为它调用了create_node_array来计算每个字符的频率：create_pq的其余部分只用常数时间，因为至多能有256个项插入优先队列中。同理，create_huffman_tree和calculate_huffman_codes方法只花费常数时间。save_to_file方法读取输入文件，因此worstTime(*n*)也和*n*成线性关系。

希望编码一个消息的用户可以调用这些方法。例如，如果输入文件包含：

```
more money needed
```

那么输出文件将是：

```
 0000
  100
d 1011
e 11
m 001
n 011
o 010
r 0001
y 1010
**
00101000011110000101001111101010100011111111011111010110000
```

在这个编码中，新行字符是非打印字符，因此在显示它时是空白。但是打印该字符会导致显示一个新行，因此得到一个空行并且在下一行上有一个空格和编码0000。

随后另一个用户可能想解码这个消息。这要分两步完成：首先，必须读入编码重新构造霍夫曼树；其次，读入编码的消息并输出解码后的消息，这应当和原消息一致。编程项目11.1要求解码被编码的消息。所有的相关文件可以参阅本书的源代码链接。

在寻找全局最优的解决方案时，贪心算法做出了局部最优的选择。

霍夫曼编码是**贪心算法**的一个例子：当打算做出一个选择时，就选择最经济的选项。在霍夫曼编码的上下文中，加入霍夫曼树的下一项是频率最低的“字符-频率”对。这样的对总是位于优先队列的顶部。在霍夫曼编码的情况中，贪心成功了：得到的编码使用了数量最少的位。在第14章里还有两个贪心算法的例子，它们也使用了优先队列，习题14.6中则说明了并非总能成功的贪心。

468

总结

本章介绍了**优先队列**：它是一个容器，根据一定方式为项分配优先级，其中只有优先级最高的项才能被访问和删除。访问的worstTime(*n*)是常数，但是删除的worstTime(*n*)是$O(\log n)$。项在优先队列中插入的位置没有任何约束，但是它的worstTime(*n*)必定是$O(n)$，averageTime(*n*)必定是常数。

在priority_queue类的设计中，项被存储在一个堆中。**堆**t是一个完全二叉树，*t*或者为空，或者满足：

1) 根据项之间的某些比较方法，根项是t中最大的项。

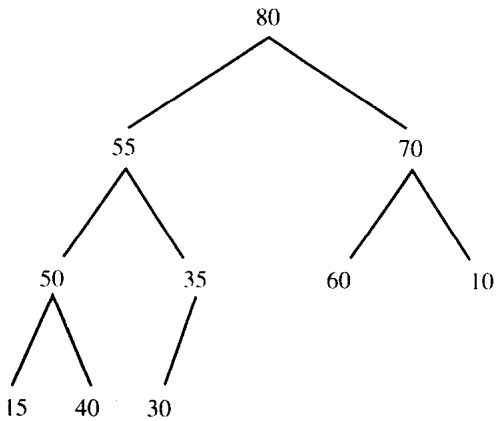
2) leftTree(*t*)和rightTree(*t*)都是堆。

因为完全二叉树可以通过子女的索引计算父亲的索引，反之亦然，所以堆可以用一个数组表示。这样就可以利用数组的随机访问能力访问一个给定索引上的项。

优先队列可以应用在数据压缩领域。给定一个消息，它可以将其中的每个字符无二义性地编码成最少数量的位。一种获得最小编码的方法是使用霍夫曼树。**霍夫曼树**是一个二-树，其中每个树叶代表原信息中一个不同的字符，每个左树枝用0标记，而每个右树枝用1标记。通过跟踪从每个字符树叶返回根的路径，并将这个路径上每个树枝的标记添加进去，就可以得到该字符的霍夫曼编码。

习题

11.1 再次观察图11-12，说明对图中的堆连续进行下面的变动的相关步骤：



469

a. push(83);

b. push(61);

c. pop();

11.2 阐述优先队列的惠普的实现的不公平性。也就是说，举出优先队列中的优先级最高的两项，但是先插入的项并不是先被删除的项。

11.3 假设一个vector容器c1包含下面的项的序列：10，20，30，40，50，60，70，80，90，100。说明在调用下面的方法时会发生什么情况：

```
make_heap(c1.begin(), c1.end(), less<int>);
```

11.4 假设一个vector容器c2包含和习题11.3中c1相同的项，不过顺序相反。说明在调用下面的方法时会发生什么情况：

```
make_heap(c2.begin(), c2.end(), less<int>);
```

11.5 估算习题11.3中make_heap调用的时间花费相对于习题11.4中make_heap调用的时间花费。

11.6 为下面的字符频率创建“字符-频率”对的堆（最高优先级=最低频率）：

a:5,000

b:2,000

c:10;000

d:8,000

e:22,000

f:49,000

g:4,000

11.7 利用下面的霍夫曼编码将单词“faced”翻译成一个位序列:

a:1100

b:11010

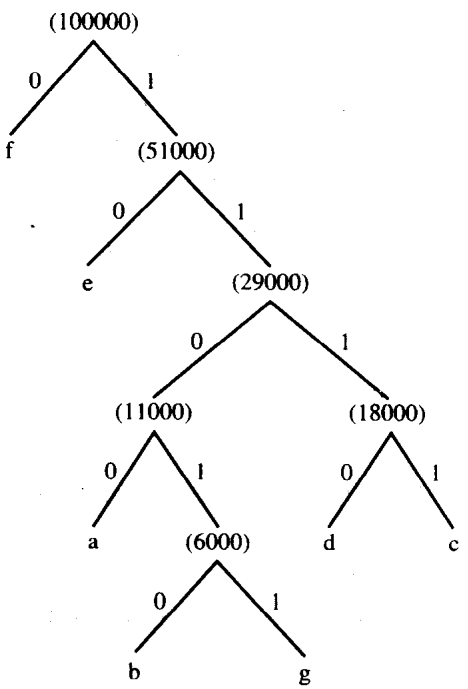
c:1111

d:1110

e:10

f:0

11.8 使用下面的霍夫曼树将位序列11101011111100111010翻译成从‘a’到‘g’的字母: 470



11.9 对下面的霍夫曼编码，举例说明一个5位且不能是任何编码消息开头的序列:

a:1100

b:11010

c:1111

d:1110

e:10

f:0

能否找到另一个这样的5位序列？解释原因。

11.10 霍夫曼树必须是二-树吗？解释原因。

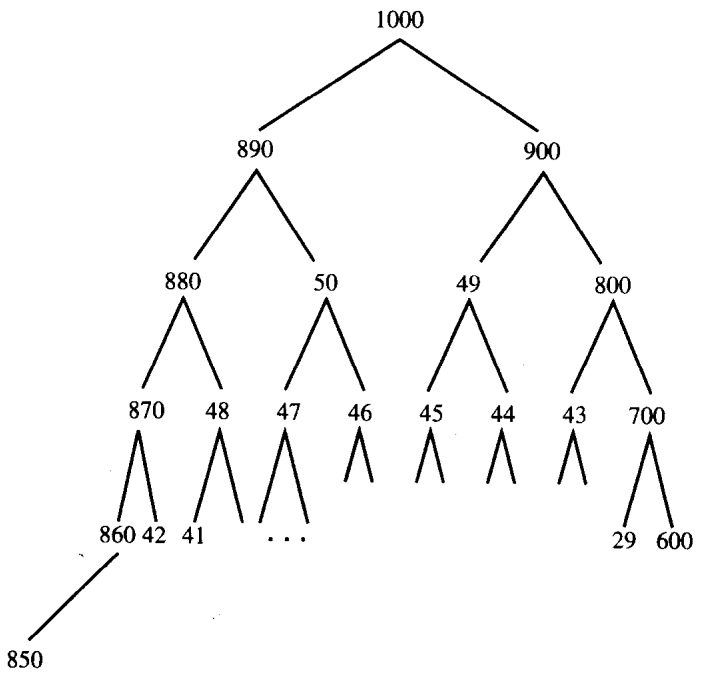
11.11 描述用从‘a’到‘h’的字母创建一个消息的过程，其中两个字母的霍夫曼编码是

7位。再用从‘a’到‘h’的字母创建一个消息，其中所有的字母的霍夫曼编码都是3位。

11.12 在习题11.8的霍夫曼树中，所有非叶节点的频率总和是215 000。这也是编码消息E的大小。说明为什么在任何霍夫曼树中，所有非叶节点的频率之和总是等于编码消息的大小。

471

11.13 为下面的堆调用pop_heap方法时需要多少次循环迭代？能不能找到一个项的数量相同但pop_heap方法需要更多迭代次数的堆？



472

11.14 在huffman类中，并没有用huffman_node类的id字段。为什么遗漏了这个字段？

编程项目11.1：解码一个消息

假设一个消息使用霍夫曼编码进行了编制。开发一个项目解码这个编制后的消息，从而获得原始消息。

分析

正如本章的霍夫曼应用中的输出文件所示，输入文件将由两部分组成：

每个字符和它的编码

编码后的消息

假设文件huffman.ou1包含了下面的内容（由于页面所限，编码后的消息被分成三行）：

```
 0010
  0111
a 000
b 1
c 0011
d 010
e 0110
```

```
**
1000010011100110110010011001110011000101110100001001001001100000100111010
0000100111001100001000101111111111111111111111111111110010
```

系统测试1

```
In the Input line, please enter the name of the input file.
huffman.ou1
In the Input line, please enter the name of the output file.
decode.ou1
Please press the Enter key to close this output window.
```

文件decode.ou1将包含:

```
bad cede cab dab
dead dad cad
bbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

假设文件huffman.ou2包含了下面的内容（消息来自Coleridge的“Rubiyat of Omar Khayam”）:

```
10000
 101
, 001110
. 0011010
A 001100
D 0010100
I 0010101
K 001111
T 0010010
W 0010110
X 0010111
a 1111
b 011000
c 00000
d 11011
e 010
g 0010011
h 11100
i 100011
l 0001
m 00001
n 0111
o 11010
p 011001
r 1001
s 1100
t 01101
u 11101
v 100010
w 001000
y 0011011
**
```

```
00101010111101001011111110111111111011111011011101110
00111101110100111111110101100000011111100011101001111
11001111011110000001100101110001101111101101010000100
11011101011001000101011111100111011001010101110111101
00000101010111011010000001001010010001110100000010110
11100010100101010100110000010110011110010101101111000
10101110011110000010010101101110110011000111000100101
00110110011111011110000001001011100100111010111010010
01111100101000001111100010010100101111100101011111101
00001011000010100100010101100110010101101110101010000
11111011100111010000001010011010001000011110101101110
10101111110111001110101110001010110011001011100010111
1001101010000
```

系统测试2

```
In the input line, please enter the name of the input file.
huffman.ou2
In the input line, please enter the name of the output file.
decode.ou2
Please press the Enter key to close this output window.
```

文件decode.ou2将包含:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree,
Where Alph the sacred river ran
Through caverns numberless to man,
Down to a sunless sea.
```

474
|
475

第12章 排 序

排序是最重要的常用计算机操作之一，也就是将容器中的项按照顺序排列。从早期简单的小容器排序到高效的频繁使用的邮件列表和字典排序，在各种各样的排序算法中进行选择始终是每个编程者的一项重要技术。本章所讨论的大多数排序算法都囊括在标准模板库中，不过它们的实现留给开发者自己完成。更明确地说是我们将研究惠普的实现。另外的几种排序算法，像选择排序、冒泡排序和基数排序，这些将在习题中进行研究。

目标

1) 能够判断究竟哪种排序算法适合某个具体的应用。
2) 了解每个排序算法的缺陷。
3) 解释分治法算法的标准。

12.1 介绍

本章只考虑基于比较的排序；也就是需要将项和其他项进行比较的排序。如果预先知道每个项的最终位置，那么比较就不是必需的。例如，如果开始时是未排序的100个不同的从0到99的整数，那么不需要比较也知道整数0最后必定在位置0，依次类推。最广为流传的不基于比较的排序算法是基数排序，见习题12.14。

每个排序算法的用户可以选择缺省的比较运算符——**operator<**，或是提供一个函数对象用于项的比较。如果选择缺省情况，那么排序的结果将是按照升序排列的集合。为了简化，在本章中将使用缺省的**operator<**。

在分析一个排序方法的过程中，averageTime(n)和worstTime(n)都是非常重要的。在某些应用中，像国防和生命支持系统，排序算法在最坏情况下的性能是很关键的。我们将了解到几种排序算法，它们提供了几种保险策略，以避免出现难以接受的非常差的最坏情况。

我们通过下面的10个整数说明每个方法的结果：

59 46 32 80 46 55 87 43 70 81

排序算法的空间需要是值得一提的。大部分算法的空间需要都是很小的：少许迭代器和索引，以及用来保存一个项的临时变量。快速排序的averageSpace(n)和n成对数关系，worstSpace(n)和n成线性关系。而树排序的averageSpace(n)和worstSpace(n)都和n成线性关系。

另一个用来评估排序算法的标准是稳定性。一个**稳定的**排序方法保持了相等项的相对顺序。例如，假设有一个关于学生的vector容器，其中每个条目由学生的姓和该学生的总成绩数组成，并希望根据总成绩数排序。如果排序方法是稳定的，而（“Balan”，28）出现在比（“Wang”，28）之前的索引上，那么在排序之后（“Balan”，28）将仍然出现在（“Wang”，28）之前。稳定性可以简化项目的开发。例如，假设上面的vector容器已经按照姓名排序，又要求应用程序根据成绩数排序；而成绩数相同的学生应当按照字母表顺序排列。稳定的排序不需

要做任何额外的工作，就可以确保相同成绩数的学生按照字母表顺序排列。

下面将开始讨论一种排序算法，它的独立性能很差，不过和其他排序算法相结合时将得到最快的（平均情况下）运行时间。

插入排序

478 假设即将排序的项是在一个支持**随机访问**迭代器的容器中。用迭代器i循环通过容器。在每次循环迭代中，使用另一个循环根据**小于**i的位置上的项将*i插入到其对应的位置。例如，假设i位于下面项中最右边的项:

32 45 59 80 91 46

那么在这个迭代之后，将有

32 45 46 59 80 91

__insertion_sort算法是

```
template <class RandomAccessIterator>
void __insertion_sort(RandomAccessIterator first,
                      RandomAccessIterator last)
{
    if (first == last)
        return;
    for (RandomAccessIterator i = first + 1; i != last; ++i)
        __linear_insert(first, i, value_type(first));
}
```

在__linear_insert调用中，第三个变元使我们可以确定项的类型。为了最高效地插入*i，__linear_insert的代码比预想得要复杂得多。和变元first和i相对应的分别是参数first和last。将*last保存在一个局部变量value中。如果value<*first，就把first到last-1之间的项（向后）拷贝到first+1和last之间；然后将value存储到first中。例如，假设i位于下列项中索引4的位置:

46 59 85 91 32

将32保存在value里，把91移动到32原先的位置，85移动到91原先的位置，59移动到85原先的位置，再把46移动到59原先的位置。最后将value移动到46原先的位置上:

32 46 59 85 91

否则，value不小于*first，然后调用__unguarded_linear_insert方法。这个方法从last-1位置开始，并一直将项向上移动，直到到达value对应的位置。下面是__linear_insert和__unguarded_linear_insert方法的代码:

```
template <class RandomAccessIterator, class T>
inline void __linear_insert(RandomAccessIterator first,
                            RandomAccessIterator last, T*)
{
    T value = *last;
    if (value < *first) {
        copy_backward(first, last, last + 1);
479     *first = value;
```

```
        }
        else
            __unguarded_linear_insert(last, value);
}
template <class RandomAccessIterator, class T>
void __unguarded_linear_insert(RandomAccessIterator last, T value)
{
        RandomAccessIterator next = last;
        --next;
        while (value < *next)
        {
            *last = *next;
            last = next--;
        }
        *last = value;
}
```

“无防护”（unguarded）意味着不需要尝试就能保证**while**循环最终会结束。这个结束保证来自于__linear_insert中的**if**语句。因此这个**if**语句的目的是使得**while**条件避免额外的next!=first-1测试。这种复杂却极有效的代码是惠普实现的特点，而且其他的实现都是以此为基础的。

插入排序举例　下面是对__insertion_sort中**for**循环的五次迭代的跟踪，包括**while**循环迭代。i位置上的项用黑体表示，而且每次迭代后从first到i之间的项都用下划线表示。

for循环，迭代1
59 **46** 32 80 46 55 87 43 70 81
59 59 32 80 46 55 87 43 70 81
46 59 32 80 46 55 87 43 70 81

for循环，迭代2
46 59 **32** 80 46 55 87 43 70 81
46 59 59 80 46 55 87 43 70 81
46 46 59 80 46 55 87 43 70 81
32 46 59 80 46 55 87 43 70 81

for循环，迭代3
32 46 59 **80** 46 55 87 43 70 81
32 46 59 80 46 55 87 43 70 81

480

for循环，迭代4
32 46 59 80 **46** 55 87 43 70 81
32 46 59 80 80 55 87 43 70 81
32 46 59 59 80 55 87 43 70 81
32 46 46 59 80 55 87 43 70 81

for循环，迭代5
32 46 46 59 80 **55** 87 43 70 81

32 46 46 59 80 80 87 43 70 81
32 46 46 59 59 80 87 43 70 81
32 46 46 55 59 80 87 43 70 81

插入排序分析 令n为从first（包括在内）到last（不包括在内）之间的项的数量。__insertion_sort中的**for**循环总是执行恰好n-1次。如果容器开始时是递增顺序的，那么__linear_insert中**while**循环的迭代次数将是0。因此总的迭代次数将是n-1，即和n成线性关系。宽松点儿说，可以说如果容器开始时“几乎”是递增顺序的，将不会有很多的内部循环迭代；那么总的迭代次数仍和n成线性关系。

另一方面，假设容器开始时是递减顺序的，像

92 85 71 55 42 23

那么在**for**循环的第一次迭代中将有**while**循环的一次迭代——打开85的位置。在**for**循环的第二次迭代中，将有**while**循环的两次迭代——打开71的位置。在**for**循环的第三次、第四次和第五次迭代中，将分别有**while**循环的三次、四次和五次迭代。一般而言，如果容器是递减顺序的，那么**while**循环的迭代次数将是：

$$1+2+3+...+(n-2)+(n-1)=\sum_{k=1}^{n-1}k=n(n-1)/2$$

也就是说，worstTime(n)和n成平方关系。在平均情况下（见习题12.7）将有约$n^2/4$次**while**循环迭代，因此averageTime(n)也和n成平方关系。

插入排序的averageTime(n)和n成平方关系。

插入排序是稳定的，因为如果value=*next，那么**while**循环停止，然后value被插入到next+1位置。通过从i开始“向下审查”而不是从first开始“向上审查”，在容器已经有序时将获得最好的时间性能，当容器近似有序时也将获得很好的时间性能。在12.3.4节中将利用这个性能。

12.2 排序能有多快

481 使用插入排序法排序一个支持随机访问迭代器的容器，它的worstTime(n)和n成平方关系。在考虑一些更快的排序方法之前，让我们花少许时间先看一下还能做多少改进。用来进行这个分析的工具是决策树。**决策树**是一个二叉树，其中每个非叶节点代表两项之间的比较，而每个树叶则表示这些项的一个排序序列。例如，图12-1显示了应用插入排序的一个决策树，其中即将被排序的项存储在变量a_1、a_2和a_3中。

对即将排序的项的每一种置换，决策树必须有一个树叶与之对应[⊖]。n个项的置换总数量是$n!$，因此如果排序n个项，那么相应的决策树必须有$n!$个树叶。根据二叉树定理，任何非空二叉树t中的树叶数量都$\leq 2^{height(t)}$。因此，对排序n个项的决策树t，

$$n! \leq 2^{height(t)}$$

两边取对数，得到

$$height(t) \geq \log_2 n!$$

⊖ 为了简化，假设排序方法没有做任何冗余的比较。

换句话说，对任何基于比较的排序而言，必定有一个树叶的深度至少为$\log_2 n!$。在决策树的上下文中，这意味着必定存在一种项的排列，对它的排序需要至少$\log_2 n!$次比较。因此对基于比较的排序，worstTime(n) ≥ $\log_2 n!$。根据习题12.8，$O(\log n!) = O(n\log n)$。这说明，不严格地讲：

> 基于比较的排序的worstTime(n)不小于$O(n\log n)$。

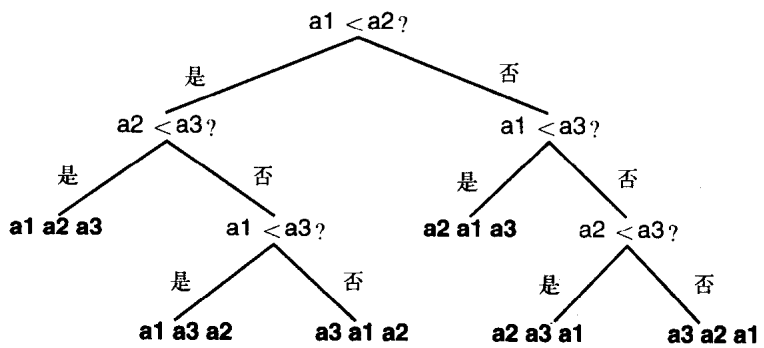


图12-1　通过插入排序法排序三个项的决策树

还可以进一步深入：任何基于比较的排序方法的averageTime(n)不小于$O(n\log n)$。为了得到这个结果，假设t是一个排序n个项的决策树。那么t有$n!$个树叶。所有$n!$种置换中，排序n项 [482]
的平均比较次数是总比较次数除以$n!$，也就是$E(t)/n!$。根据第8章的外部路径定理，$E(t) \geq (n!/2)\text{floor}(\log_2 n!)$。因此得到：

averageTime(n) ≥ 平均比较次数
=E(t)/n!
≥ (n!/2)floor($\log_2$n!)/n!
=(1/2)floor($\log_2$n!)

因为floor($\log_2 n!$) ≤ $\log_2 n!$，所以floor($\log_2 n!$)是$O(\log n!)$。根据习题12.8，$O(\log n!)=O(n\log n)$。可以断定floor($\log_2 n!$)是$O(n\log n)$。将这个结果代入averageTime中，得到

> 基于比较的排序的averageTime(n)不小于$O(n\log n)$。

对任何基于比较的排序，如果worstTime(n)是$O(n\log n)$，那么averageTime(n)也是$O(n\log n)$。

幸运的是，这个目标并不是遥不可及的。本章中剩余的每个算法的averageTime(n)都是$O(n\log n)$。实际上，下面三个排序算法的worstTime(n)也只是$O(n\log n)$。可以利用这个事实证明averageTime(n)必定也是$O(n\log n)$。为什么？假设有一个排序算法，它的worstTime(n)是$O(n\log n)$；那么averageTime(n)肯定不会比$O(n\log n)$差。但是根据平均时间估算，averageTime(n)也不能比$O(n\log n)$快。因此可以得出结论，如果一个排序算法的worstTime(n)是$O(n\log n)$，那么它的averageTime(n)必定也是$O(n\log n)$。

12.3 快速排序

在12.3.1到12.3.4节中，将考察四个averageTime(n)仅为$O(n\log n)$的排序算法。其中三个算法的worstTime(n)也是$O(n\log n)$，而另一个的worstTime(n)是$O(n^2)$。最奇怪的是最后一个算法，称作快速排序，它往往被看作是综合效率最高的排序！快速排序在最坏情况下的性能很差，但是在平均情况下的运行时间和速度性能是很多方法中最好的。在实验27中将证明这一性能。先从基于第10章的multiset类的排序算法开始。

12.3.1 树排序

即将被排序的项是一个接一个地插入到一个初始为空的multiset容器中的。这就是树排序，因为最困难的工作是在multiset类的insert方法中完成的，或者说是rb_tree类中的insert方法更为合适。通用型算法tree_sort不是标准模板库的一部分，它是完成实际工作的函数的包装器。

```
template<class ForwardIterator>
void tree_sort (ForwardIterator first, ForwardIterator last)
{
    if (first != last)
        tree_sort_aux (first, last, *first);
} // tree_sort算法
```

483

辅助函数tree_sort_aux的第三个参数是将被排序的项的类型。这个类型是构造set容器所必需的。下面是tree_sort_aux的定义：

```
template<class ForwardIterator, class T>
void tree_sort_aux (ForwardIterator first, ForwardIterator last, T)
{
    multiset< T, less< T > > tree_set;
    ForwardIterator itr;
    for (itr = first; itr != last; itr++)
        tree_set.insert (*itr);
    copy (tree_set.begin( ), tree_set.end( ), first);
}
```

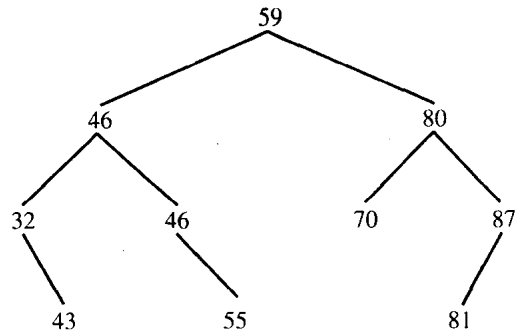
例如，要在一个vector容器v中应用树排序，应使用

```
tree_sort(v.begin(), v.end());
```

树排序举例 先给出项的初始集合，然后通过逐个将这些项插入一个初始为空的multiset容器，构造红黑树（假设multiset类是用一个红黑树实现的）。原始顺序的项是：

59 46 32 80 46 55 87 43 70 81

得到的红黑树是：



然后将项拷贝到原来的容器中。 484

树排序分析　对i=1, 2, ... , n−1，插入第i项需要至多$\log_2 i$次迭代。如果插入第i项之后需要重构红黑树，那么至多需要$(\log_2 i)/2$次迭代（回忆第10章红黑树插入的情况2中，用x的祖父替换了x）。这个最坏情况下的总迭代次数是:

$$1.5\sum_{i=1}^{n}\log_2 i \leqslant 1.5\sum_{i=1}^{n}\log_2 n = 1.5n\log_2 n$$

树排序是快速而稳定的，但空间需求与n成线性关系。

因此树排序的worstTime(n)是$O(n\log n)$，而且根据12.2节里的讨论，$O(n\log n)$一定是worstTime(n)的最小上界。那么averageTime(n)又怎么样呢？由于averageTime(n) ⩽ worstTime(n)，所以averageTime(n)是$O(n\log n)$。根据12.2节的结论，同理可得$O(n\log n)$也一定是averageTime(n)的最小上界。

树排序需要创建n项的多集合，因此worstSpace(n)和n成线性关系。树排序是稳定的，因为如果即将插入的一项y等于红黑树中已经存在的一项x，那么y将被插入到x的右子树中。从那时起的任何旋转，y都将被看作好像是x小于y，因此在所有前向迭代中x将在y之前出现。

12.3.2　堆排序

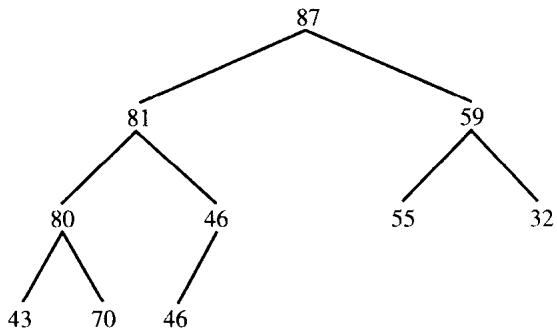
堆排序方法是J.W.J.Williams（1964）发明的。给定一个支持随机访问迭代器的容器c，堆排序将进行以下两步处理:

```
make_heap(c.begin(), c.end()); //使用运算符<进行比较
sort_heap(c.begin(), c.end());
```

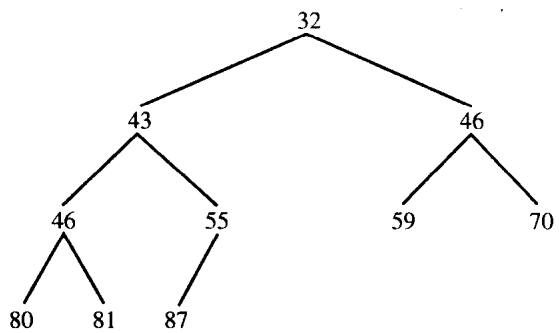
我们在第11章中已经了解了make_heap算法。sort_heap算法是相当简单的，因为困难的工作都由pop_heap方法完成了:

```
//前置条件: 容器中从first（包括在内）到last（不包括在内）之间的项构成
//          了一个堆。
//后置条件: 容器中从first（包括在内）到last（不包括在内）之间的项以
//          升序排列。worstTime(n)是O(nlogn)。
void sort_heap(RandomAccessIterator first, RandomAccessIterator last)
{
    while(last - first >1)
          pop_heap(first, last--);//使用运算符<进行比较
}//sort_heap
```

堆排序举例 如果照例使用10个项的集合，那么执行完make_heap算法之后，将得到下面
485 的堆:



在sort_heap调用中，反复弹出根项并将它插入容器尾，因此在10次迭代之后得到下面的完全二叉树，而不再是一个堆:



在容器中，项的序列将是按升序排列的。

堆排序分析 堆排序分析依赖于make_heap和pop_heap算法的分析。从第11章最后我们了解到make_heap的worstTime(n)只是$O(n)$。在sort_heap中，调用了pop_heap算法n次。第一次这样的调用需要不超过$2\log_2 n$次迭代。使用2这个因子是因为必须沿着所有到树叶的路径找到一个空间，随后将沿着所有的路径返回根，将最后一项插入。第二个调用需要不超过$2\log_2(n-1)$次迭代…… 倒数第二个调用需要不超过两次的迭代。最后一次pop_heap调用不需要迭代。也就是说，在最坏情况下，迭代的总次数是:

$$2\sum_{i=1}^{n}\log_2 i$$

剩下的分析和树排序相同: worstTime(n)是$O(n\log n)$，averageTime(n)也是$O(n\log n)$，并且这些都是最小上界。由于堆排序是一个本地（in-place）排序，所以worstSpace(n)是常数。

堆排序是快速的不稳定的本地排序。

堆排序不是一个稳定的排序方法。例如，假设vector对象vec由下面的姓名和测验成绩组
486 成，测验成绩高的优先级别高:

"Jones", 85 "Smith", 85

make_heap调用将保留vec不变，因为Smith的优先级不大于Jone的优先级。那么sort_heap将把"Jones"存储在索引1，而"Smith"存储在索引0:

"Smith", 85　"Jones", 85

12.3.3 归并排序

归并排序算法是作为list类中的sort方法提出的。下面是它的方法接口:

```
//后置条件: 这个列表根据运算符<按照升序排列。worstTime(n)是O(nlogn)。
void sort();
```

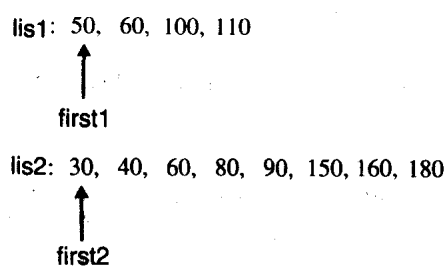
回忆在list类的惠普实现中，列表中的每一项都是作为节点的一个字段存储的，在这个节点中还有prev和next指针字段。几乎所有的排序工作都可以通过操作这些指针字段完成；项本身是不需要移动的。在研究这个排序方法的设计之前，需要开发一个辅助方法——merge。**protected**型方法merge的头如下:

```
//前置条件: 这个列表和x都是根据运算符<排序的列表。
//后置条件: 这个列表是一个排序列表，它包含所有在这次调用前属于它自身或是
//          属于x的项。worstTime(n)是O(n)，其中n是（调用前）这个调用列表对象
//          的大小。
void merge(list<T>& x);
```

注意　例如，如果一个项在调用对象中出现一次，在x中出现两次，那么在调用merge之后，该项将在调用对象中出现三次。

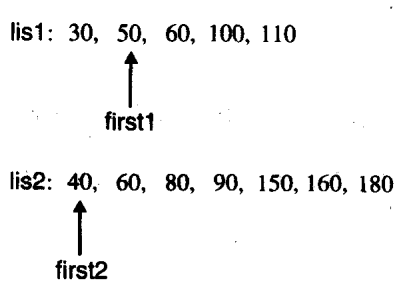
merge方法的惠普的定义使用了一对迭代器——first1和first2。最初，first1位于调用对象的第一个节点，first2位于x中的第一个节点。循环直到耗尽一个列表。在每次迭代中，如果first2的项小于first1的项，就通过调整指针，将first2的节点转移到恰好位于first1的节点之前并改变first2。否则，就改变first1。循环之后如果x尚未耗尽，就把所有x中的项转移到调用对象的尾部。最后为调用对象的长度增添上x的长度并把x的长度设置为0。

例如，假设调用是lis1.merge(lis2)，这些列表包含下面的项:

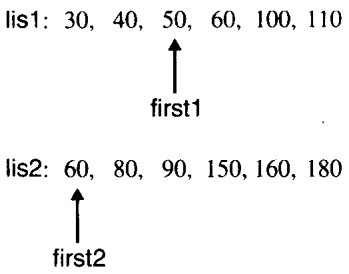


487

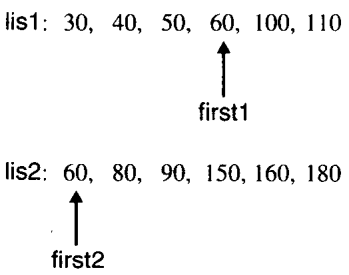
因为first2的项30小于first1的项50，所以将first2的节点转移（通过指针的调整）到恰好位于first1的节点之前并修改first2。现在有:



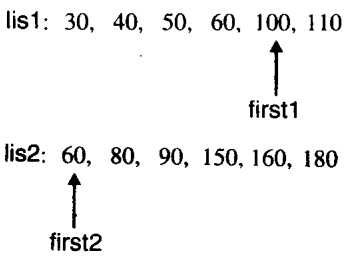
因为first2的项仍小于first1的项，所以转移first2的节点并修改first2：



现在first2的项不小于first1的项，因此只需要修改first1：

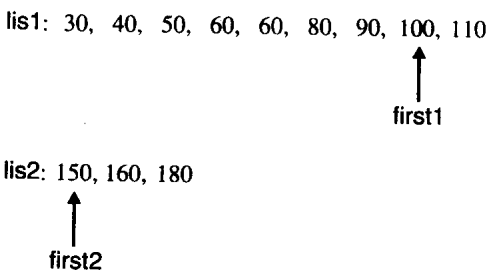


再次修改first1，因为first2的项60不小于first1的项（也是60）：

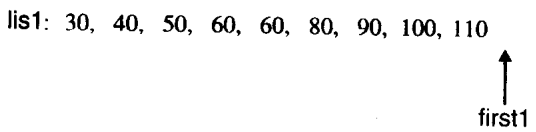


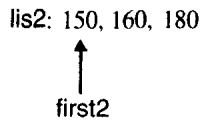
488

在接下来的三次循环迭代中，first2的项都小于first1的项，因此转移first2的项并修改first2：

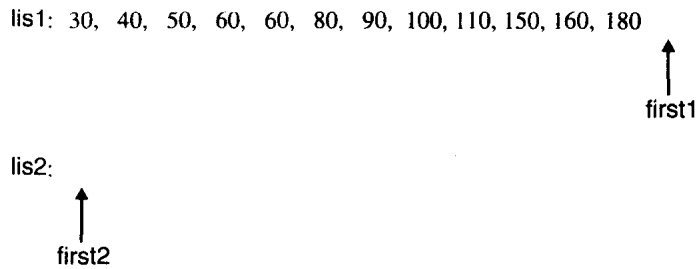


在下面的两次迭代中，first2的项都不小于first1的项，因此修改first1：





现在lis1被耗尽，于是将list对象lis2中所有剩余的节点转移（通过指针的调整）到list对象lis1的尾部：



下面是惠普的定义：

```
template <class T>
void list<T>::merge(list<T>& x)
{
    iterator first1 = begin( );
    iterator last1 = end( );
    iterator first2 = x.begin( );
    iterator last2 = x.end( );
    while (first1 != last1 && first2 != last2)
        if (*first2 < *first1) {
            iterator next = first2;
            transfer(first1, first2, ++next);
            first2 = next;
        }
        else
            ++first1;
    if (first2 != last2) transfer(last1, first2, last2);
        length += x.length;
    x.length = 0;
}
```

489

为了今后的引用，现在了解一下当两个列表包含相同数量的项时merge的情况，在这种情况下，merge是很糟糕的。假设这两个列表的项是交错的，例如，

lis1: 10, 30, 50, 70
lis2: 20, 40, 60, 80

在这种情况下需要七次循环迭代：(10 20)，(20 30)，(30 40)，(40 50)，(50 60)，(60 70)，(70 80)。一般而言，如果每个列表大小为k，那么最坏情况下的总循环迭代次数是$2k$-1。

使用merge方法归并子列表，列表类的sort方法从调用对象的头开始，并重复地将子列表合并成双倍大小的子列表。大小为1的子列表（它是自动排序的）被合并成大小为2的排序子

列表；大小为2的排序子列表被合并成大小为4的排序子列表，依次类推。归并后的子列表存储在一个称作counter的列表的数组中。对每个从0到63的i值，counter[i]开始为空，随后可能包含0、2^i或2^{i+1}个项。只要counter[i]包含了2^{i+1}个项，如果counter[i+1]为空就将它们转移给counter[i+1]，否则就和counter[i+1]合并。

归并排序举例 假设消息是lis.sort()并且lis包含下面的项:

```
lis: 30, 10, 20, 50, 40, 90, 25, 35, 15, 85, 60
```

lis中的第一个节点被转移到一个名为carry的临时表中，然后和counter[0]交换。现在有:

```
lis: 10, 50, 20, 40, 90, 25, 35, 15, 85, 60
carry:
counter [0]: 30
```

490 当lis中的下一个项转移到carry中时，该列表和counter[0]归并，然后转移到counter[1]:

```
lis: 50, 20, 40, 90, 25, 35, 15, 85, 60
carry:
counter [0]:
counter [1]: 10, 30
```

当50转移到carry中时，该项被转移到counter[0]，因为counter[0]为空。当20转移到carry中时，该项和50进行归并，使得counter[0]满。因此counter[0]和counter[1]进行归并:

```
lis: 40, 90, 25, 35, 15, 85, 60
carry:
counter [0]:
counter [1]: 10, 20, 30, 50
```

但是现在counter[1]已满，所以它的项被转移到counter[2]:

```
lis: 40, 90, 25, 35, 15, 85, 60
carry:
counter [0]:
counter [1]:
counter [2]: 10, 20, 30, 50
```

在归并了lis中的下面四项之后，counter[3]包含了原表中的前八项且以升序排列。当归并lis中的最后三项时，得到:

```
lis:
carry:
counter [0]: 60
counter [1]: 15, 85
counter [2]:
counter [3]: 10, 20, 25, 30, 35, 40, 50, 90
```

最后，从counter[2]下至counter[0]，所有的counter列表被归并到counter[3]，然后和lis进行交换，因此最终结果是:

lis: 10,15,20,25,30,35,40,50,60,85,90

carry列表在算法中扮演了一个重要的角色，尽管在每次迭代的开头和结尾处它都是空。列表中的每一项在和某一个counter列表归并或交换之前都存储在carry中。而且对某些i，当counter[i]已满时，就把counter[i]和一个空的carry交换。然后如果counter[i+1]为空，就只是将carry和counter[i+1]交换；否则就把carry和counter[i+1]归并。在这两种情况下，carry和counter[i]都是空。

下面是惠普的定义：

491

```
template <class T>
void list<T>::sort( )
{
    if (size( ) < 2)
        return;
    list<T> carry;
    list<T> counter[64];
    int fill = 0;
    while (!empty( ))
    {
        carry.splice(carry.begin( ), *this, begin( ));
        int i = 0;
        while(i < fill && !counter[i].empty( ))
        {
            counter[i].merge(carry);
            carry.swap(counter[i++]);
        }
        carry.swap(counter[i]);
        if (i == fill)
            ++fill;
    }
    for (int i = 1; i < fill; ++i)
        counter[i].merge(counter[i-1]);
    swap(counter[fill-1]);
}
```

归并排序分析　list类的sort方法将花费多长时间？和树排序以及堆排序一样，这里还是做一个最坏情况下的分析来了解worstTime(n)和averageTime(n)。对列表中每对连续的项，counter[0]中对的第二项和对的第一项归并。这样的$n/2$次归并的每一个都需要merge方法中的一次循环迭代，因此对归并的总迭代数量是$n/2$。对列表中每四个连续的项，counter[1]中每四项的第二对和第一对归并，而且这样的$n/4$次归并的每一个至多需要merge方法中的三次循环迭代。继续这样下去可得到下面的表，其中k=floor($\log_2 n$):

从carry转移到	归并次数	每次归并需要的迭代
counter[0]	$n/2$	1
counter[1]	$n/4$	3
counter[2]	$n/8$	7

（续）

从carry转移到	归并次数	每次归并需要的迭代
counter[3]	$n/16$	15
...	...	...
counter[k-1]	n/k	2^k-1

492

在表的每一行上，总迭代次数小于n。行的数量是k，也就是floor($\log_2 n$)。因此所有行的总迭代数量小于$n$floor($\log_2 n$)，也就是$O(n\log n)$。但是任何基于比较的排序必定需要至少$O(n\log n)$次迭代。由此可以断定list类中sort方法的worstTime(n)是$O(n\log n)$，所以averageTime(n)也是$O(n\log n)$，并且这些都是最小上界。

空间需求怎么样呢？从有n项的列表开始，并且这些项是*从不移动的*——所有的“转移”都使用指针操作而不是拷贝。因此可以说worstSpace(n)是常数。严格地讲，讨论归并排序的大O性质是不合适的，因为对很大的n来说，该方法将失效。具体地说，counter数组至多保存64个列表和$2^{64}-1$个项。而从实际的角度来说，这个缺陷不太可能产生什么不好的结果，因为2^{64}是一个比百万的四次方还大的数。

归并排序是一个快速的本地排序，它是稳定的。

要证明归并排序是稳定的，首先需要考察merge方法。假设有下面形式的消息：

```
lis1.merge(lis2)
```

其中lis1中的x1和lis2中的x2相等。因为x2不小于x1，所以x2和x1的比较将导致在merge方法中增加first1迭代器。这就保证了当归并结束时x1将位于x2之前。

现在假设在原来的列表中，x比y更接近列表的开头，并且x等于y。当y被转移到临时表carry里时，x已经在counter[k]里，其中$k \geqslant 0$。如果k=0，那么当调用

```
counter[0].merge(carry);
```

时将比较x和y，而且根据对merge的分析可知x将位于y之前。否则，当调用

```
counter[k].merge(counter[k-1]);
```

时将比较x和y，同样根据对merge的分析可知x将位于y之前。综上所述归并排序是稳定的。

12.3.4 快速排序

一种最有效且应用最广泛的排序算法是快速排序，它是由C.A.R.Hoare（1962）开发的。标准模板库里的<algorithm>中的通用型算法sort有如下接口：

```
//后置条件: 容器中从first到last-1之间的项是以升序排列的。
//          averageTime(n)是O(n log n)，worstTime(n)是O(n*n)。
template<class RandomAccessIterator>
void sort(RandomAccessIterator first,RandomAccessIterator last);
```

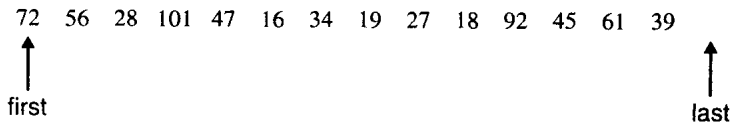
这个sort版本假定项之间的比较使用的是**operator**<。还有另一个版本，它的第三个参数
493 是一个用来比较项的函数对象。

容器可以是数组、向量、双端队列或是某些支持随机访问迭代器的用户定义的容器。基本上，惠普的sort方法首先将从first到last-1之间的项分隔成一个左子分区和一个右子分区，使

得左子分区中的每一项都小于等于右子分区中的每一项。最后，使用快速排序对左、右子分区排序。最后的这个步骤很容易用两个递归调用完成，因此这里将着重关注分隔阶段。

首先，确定一个**枢轴**（pivot）项：为了将容器分成两段，用来与容器中其他项进行比较的项。枢轴应当选择三个项——*first、*(first+(last-first)/2)和*(last-1)——的中值[⊖]。

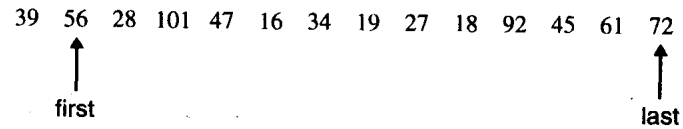
快速排序的分隔举例　假设试图分隔下面的容器：



注意，因为last-first是14，所以项19就是*(first+(last-first)/2)。枢轴是39；也就是72、19和39三个值的中值。现在想把所有小于等于39的项移动到左子分区，把所有大于等于39的项移动到右子分区。值等于39的项可以放在任一分区里，并且这两个子分区的大小不一定相同。

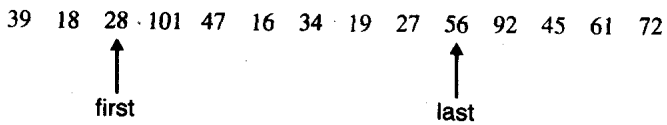
为了完成这个分隔，使用了一个**while**循环，直到产生**return**。在这个循环中有两个**while**循环。第一个内部循环增加first直到*first ≥ 枢轴。然后last减1，并且第二个内部循环将减少last直到*last ≤ 枢轴。如果first ≥ last，就返回first。否则就交换first和last上的项，增加first并再次执行外部循环。

在这个例子中，因为72是first中的项，而72 ≥ 39，所以第一个内部循环立即停止了。然后last减1并进入第二个内部循环。39是last中的项，因为39 ≤ 39，所以循环也立即停止。当交换72和39并增加first后，有：

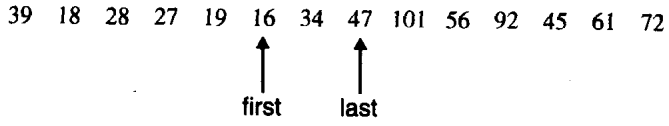


494

由于56 ≥ 39，所以第一个内部循环又一次马上停止。然后last减1，进入第二个内部循环。这个循环迭代三次，最后*last=18。交换并增加first之后得到：

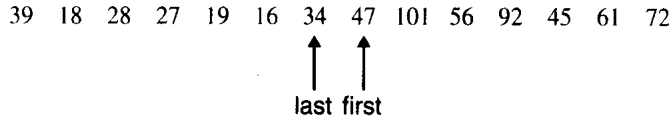


在接下来的两次外部循环迭代中，101和27交换，47和19交换。现在有：

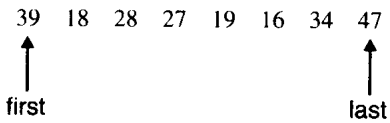


在外部循环的下一次迭代中，第一个内部循环迭代两次，last减少，第二个内部循环马上停止。因为first ≥ last，所以外部循环结束并返回first。分隔如下：

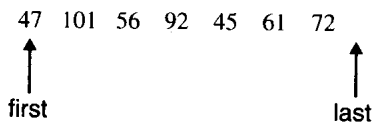
⊖ 三个值的**中值**是一个中间的值，也就是当排序这三个值时位于中间位置的数值。



左子分区由项39, 18, ... , 34组成；右子分区由项47, 101, ... , 72组成。注意在first之前的每一项都≤39，而位于first上及其后的项都≥39。使用快速排序对左子分区进行排序，将得到新的first和last值：



注意last位于子分区中最后一项的下一项。枢轴是34，即（39，27，34）的中值。原容器的右子分区的设置是：



这里枢轴是72，即（47，92，72）的中值。在习题12.10中要求完成这个第二层次的分隔。

495 刚才描述的算法称作__unguarded_partition。下面是__unguarded_partition的定义：

```
//前置条件: first<last。从first到last-1之间至少有项x和项y，满足
//          (pivot<x) 为假且(y<pivot)为假。
//后置条件: 对于任意满足first的原始值<=itr1且itr2<=last的原始值的
//          迭代器itr1和itr2，如果itr1<first的最后值，那么(pivot<*itr1)
//          为假，如果itr2>=first的最后值，那么(*itr2<pivot)为假。worstTime(n)
//          是O(n)。

template <class RandomAccessIterator, class T>
RandomAccessIterator __unguarded_partition(RandomAccessIterator first,
                                           RandomAccessIterator last,
                                               T pivot)
{
    while (1)
    {
        while (*first < pivot)
            ++first;
        --last;
        while (pivot < *last)
            --last;
        if (!(first < last))
            return first;
        iter_swap(first, last);
            ++first;
    }
}
```

现在已经了解了分隔是如何完成的，下面将开发高层次的快速排序算法。__quick_sort_loop_aux函数首先求出枢轴，然后调用__unguarded_partition。__unguarded_partition的返回值存储在RandomAccessIterator对象cut中。先用快速排序对两段（first到cut-1，cut到last-1）中较小的一段排序，然后再排序较大的一段。先选择较小一段的原因是它能够被分隔的次数少于较大的段，因此任何时候堆栈中活动记录的数量都在减少。

枢轴的选择、__unguarded_partition的调用以及较小段的快速排序是在一个**while**循环中发生的，循环将不断运行直到last-first小于等于某一阈值。采用阈值的目的将在下面诠释。现在假设阈值是1，因此只要从first到last-1之间的段中有至少两项，循环就继续下去。下面是完整的__quick_sort_loop_aux算法：

```
template <class RandomAccessIterator, class T>
void __quick_sort_loop_aux(RandomAccessIterator first,                    496
                           RandomAccessIterator last, T*)
{
    while (last - first > __stl_threshold)
    {
        RandomAccessIterator cut = __unguarded_partition (first, last,
                T(__median(*first, *(first + (last - first)/2), *(last - 1))));
        if (cut - first >= last - cut)
        {
            __quick_sort_loop(cut, last);
            last = cut;
        }
        else
        {
            __quick_sort_loop(first, cut);
            first = cut;
        }
    }
}
```

带有参数first和last的__quick_sort_loop算法只不过是获得项*first的类型T，然后调用以T为模板的__quick_sort_loop_aux。下面是__quick_sort_loop的定义：

```
template <class RandomAccessIterator>
inline void __quick_sort_loop(RandomAccessIterator first,
                              RandomAccessIterator last) {
    __quick_sort_loop_aux(first, last, value_type(first));
}
```

快速排序分析　分析快速排序的性能，先从一个简单的假设开始，假定常数_stl_threshold的值为1。也就是说，每个段都将被分隔，除非该段的大小是0（即，如果first=last）或者1（如果first=last-1）。但是这意味着排序大小为n的容器将需要约n个分区。当把一个段分隔成两部分时，由于每个项都和枢轴进行比较，所以最里面的两个**while**循环的迭代次数近似等于段的大小。因此总的迭代次数依赖于被分隔的段的大小。

观察快速排序的效果可知，这就好像创建了一个假想的折半查找树，其中根项是枢轴，而它的左右子树分别是左段和右段。例如，图12-2显示了当快速排序效率最高时导出完全折半查找树的示例，也就是每次分隔都将段分成两个同样大小的子段的情况。如果项原来就是按顺序排列或按逆序排列的，那么将得到这样的树，因为这时*first、*(first+(last-first)/2)和
497 *(last-1)的枢轴将总是*(first+(last-first)/2)。

因为每次分隔都将段分成两个相等的部分，所以图12-2的树中分隔层次的数量就等于它的高度floor($\log_2 7$)，即2。迭代的总数量约是14（也就是n*floor($\log_2 n$)，其中n=7）。对大小是n的容器，如果每次分隔都得到大小相同的子分区，那么分隔层次的数量近似为$\log_2 n$，因此总的迭代次数就近似是$n\log_2 n$。

对比图12-2中的树与图12-3中的树。图12-3中的树表示了下面的项序列产生的分隔：

20 30 40 10 80 50 60 70

每次迭代中，枢轴是次小项或者次大项时将出现最坏的情况。也就是上面的序列产生的图12-3所示的树。

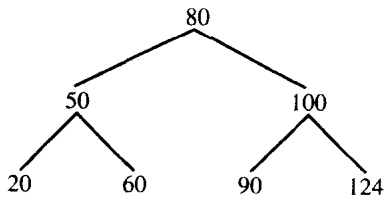


图12-2 通过重复分隔包含七个项的随机访问容器，得到同等大小的段，由此创建的假想的折半查找树

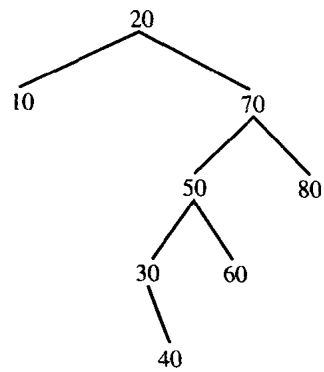


图12-3 最坏情况的分隔：每次分隔只能将快速排序的段的大小减1，相应的（假想的）折半查找树在每个非根层次上只有一个树叶

在最坏情况中，第一个分隔需要约n次迭代并产生一个大小为$n-1$的段（另一个是大小为1
498 的段）。当分隔这个大小为$n-1$的段时需要约$n-1$次迭代，并产生一个大小为$n-2$的段。持续这个过程，直到最后产生的较大的子分区大小为1。总的迭代次数近似为$n+(n-1)+(n-2)+...+1$，共计$n(n+1)/2$次。由此可以推断worstTime(n)和n成平方关系。顺带提一下，这个最坏情况并不是在每天运行中都可能遇到的。在习题12.5中将要求开发一个算法以产生这种最坏情况。

现在估算快速排序的平均时间。平均时间花费由容器中项的全部n!个初始排列控制。假想折半查找树的每个层次都代表着分隔，大约需要n次迭代。层的数量是这个折半查找树的平

均高度。因为折半查找树的平均高度和n成对数关系，由此可推断总的迭代次数是$O(n\log n)$。也就是说averageTime(n)是$O(n\log n)$。根据12.2节中关于排序究竟能有多快的结论可知，这一定也是最小上界。

阈值　分隔有两个或三个项的段比直接对它们排序更费时。正因如此，只有当一个段的大小大于某常数__str_threshold时才在其中执行快速排序。在惠普的实现里，这个常数的值是16，不过最适当的值依赖于机器。

当__quick_sort_loop_aux执行结束时，容器中从first到last-1之间的段不会完全有序，而是“半有序的”。也就是说，从first到last−1之间的项将由k个段组成，k是正整数：

first到itr_1-1，itr_1到itr_2-1，itr_2到itr_3-1，…，itr_{k-1}到last−1

对任何1和k-1之间的j而言，所有段j中的项都小于等于段j+1中的项。每个段的大小至多是__str_threshold。

平均情况下，获得这些半有序的段的时间花费是$O(n\log n)$，因为大约有$\log_2(n/$__stl_threshold)个分隔层次，每个层次有n次迭代。

“不过”，读者可能质疑，“在平均情况下的时间花费仍为$O(n\log n)$而且甚至没有得到一个有序的列表？目的是什么？”当考察运行时间时将了解到，目的就是使用阈值比不使用它要快一些，因为快速排序不适合小的段。

要完成排序，将插入排序应用到从first到last-1之间的所有的项上。回忆一下，当容器有序或近似有序时，该算法的时间花费和n成线性关系。特别是因为每个半有序的段的大小小于等于__stl_threshold，所以排序这样的段时插入排序执行的循环迭代次数近似为__stl_threshold$^2/4$。这里最多有n个段，因此采用这样的方式排序first到last-1之间的项最多需要n*__stl_threshold$^2/4$次循环迭代，又因为__stl_threshold是一个常数，所以它和n是成线性关系的。 499

因而这个改进版本的快速排序的averageTime(n)仍旧是$O(n\log n)$，但倘若把__stl_threshold设置为1，那么实际的运行时间性能要快些。注意worstTime(n)将仍和n成平方关系。

快速排序的空间需求依赖于递归调用的数量。在最坏情况下，递归调用的数量和n成线性关系，因此worstSpace(n)和n成线性关系。在平均情况下，递归调用的数量和n成对数关系，因此averageSpace(n)和n成对数关系。

快速排序是不稳定的。比如，假设开始时的序列如下：

0 10 10 2 3 4 5 6 7 8 9 10 11 12 13 14 5 6 15

枢轴是8。在第一次分隔中，位于索引1上的10和索引17上的6交换，索引2上的10和索引16上的5交换。现在这两个10的相对位置就和原先的不同，并且这样的结果将保持下去（回忆一下，插入排序是稳定的）。

快速排序在平均情况下是非常快的，而在最坏情况下则很慢，它是不稳定的，并且也不是本地的排序方法。

总之，快速排序最坏情况下的时间花费是非常糟糕的，它的空间需要也不是常数，并且不是一个稳定的排序。快速排序的吸引人之处就在于平均情况下的快速性；在实验27中将证实这一点。

12.3.5　分治法算法

快速排序是一个分治法算法。每个分治法算法都具备下列特性：

- 函数由至少两个函数自身的递归调用组成。
- 递归调用是独立且可以并行执行的。
- 通过结合递归调用的结果可以实现原先的任务。

快速排序是一个分治法算法。

在快速排序里，左右子分区的分隔可以分别完成，然后用快速排序对左右子分区排序。当所有子分区都用快速排序排列好顺序之后，调用插入排序将项按顺序插入原容器。因此满足分治法算法的需要。

分治法算法和一个任意的包含两个递归调用的递归算法有什么不同呢？区别是一个任意的递归算法中的递归调用不需要是独立的。例如，在汉诺塔问题中，$n-1$个盘子必须在这些同样的n-1个盘子从临时杆移动到目的地之前，先从源移动到临时杆上。注意实验9中原先的斐波纳契函数就是一个分治法算法，但是这两个独立的函数调用暗示了函数总体的无效率性。

实验27对本章中所有快的排序算法进行了运行时间实验。

500

实验27：排序算法的运行时间　　（所有实验都是可选的）

总结

本章包含了五个基于比较的排序方法。插入排序只能用于支持随机访问迭代器的容器（像数组、向量和双端队列），并且把它应用在升序或接近升序排列的容器上只花费n的线性时间。插入排序并非直接应用（它的averageTime(n)和n成平方关系），它是通用型算法sort的一部分。

对基于比较的排序而言，worstTime(n)不会快于$O(n\log n)$，并且averageTime(n)也不快于$O(n\log n)$。

树排序算法反复地将项插入一个multiset容器；worstTime(n)是$O(n\log n)$。sort_heap算法的worstTime(n)也是$O(n\log n)$。归并排序是list类的排序方法，它在最坏情况下的时间花费也是$O(n\log n)$。因此，所有这三种排序算法的averageTime(n)都是$O(n\log n)$，并且所有这些上界都是最小的。

快速排序——通用型sort算法——只能用于支持随机访问迭代器的容器，像数组、向量和双端队列。算法开始时将从first到last-1之间的段分成左子段和右子段，使得左子段中的每一项小于等于右子段中的每一项。左右子段自身再进行分隔，只要有一个段的大小大于某一阈值（一般是16）就持续这个过程。最后，调用一种插入排序完成整体的排序。尽管sort在最坏情况下的时间花费是平方关系的，但是averageTime(n)也只是$O(n\log n)$。

表12-1对本章中提到的排序算法进行了简单的归纳总结。

表12-1　排序算法的重要特点

排序算法	averageTime(n)	worstTime(n)	运行时间排名	稳定吗？
插入排序	$O(n^2)$	$O(n^2)$	5	是
树排序	$O(n\log n)$	$O(n\log n)$	3	是
堆排序	$O(n\log n)$	$O(n\log n)$	2	否
归并排序	$O(n\log n)$	$O(n\log n)$	4	是
快速排序	$O(n\log n)$	$O(n^2)$	1	否

* 运行时间排名是根据排序n个随机整数所花费的时间确定的。详情参阅实验27。

习题

12.1 使用下面的数值序列跟踪表12-1中给出的每种排序方法的执行情况:

10 90 45 82 71 96 82 50 33 43 67

用1作为快速排序的阈值。 501

12.2 a. 针对每种排序算法重新安排习题12.1中的数值序列，使得用最少的比较次数排序容器。

b. 针对每种排序算法重新安排习题12.1中的数值序列，使得用最多的比较次数排序容器。

12.3 假设通过将项插入一个初始为空的BinSearchTree容器来进行排序，估算worstTime(n)和averageTime(n)。

12.4 本章的哪些排序算法的averageTime(n)和worstTime(n)的大O估算相同?

12.5 开发一个算法排列容器中0...$n-1$的整数，使得快速排序需要平方时间才能排序容器。

提示 为了获得最坏的时间花费，每次分隔都应当产生一个只包含一个项的段，段中或者包含最小的项，或者包含最大的项。一种方法是将项按顺序排列，但是中间的两项是最小的和最大的项。例如，如果数值是0...9，那么就从

1，2，3，4，0，9，5，6，7，8

开始。注意，执行每次分隔时，枢轴或者是次小项，或者是次大项。因此每次分隔都将产生一个大小是1的段。

12.6 a. 假设有一个排序算法的averageTime(n)是$O(n\log n)$，并假设这就是最小上界。例如，本章中的任一快的排序都将满足上面的条件。回忆在实验16中，runTime(n)代表在一个特定计算机系统中算法实现排序n个随机整数需要的时间。因此可以写成:

$$\text{runTime}(n) \approx k(c)n\log_c n \text{ 秒}$$

其中c是一个整数变量，而k是一个函数，它的值依赖于c。证明runTime(cn) ≈ runTime(n)(c+c/log$_c n$)。

b. 使用习题12.6a中的方法估算当runTime(100 000)=10秒时，runTime(200 000)的值。

12.7 证明插入排序中需要的**while**循环平均迭代次数是

$$n(n-1)/4$$ 502

提示 首先证明，对[1...n]中的k，插入第k项需要的**while**循环平均迭代次数是(k−1)/2。(第一个插入的项对应着k=1；没有“第0”项。)

12.8 证明$O(\log n!)=O(n\log n)$。

提示 $n!=\prod_{i=1}^{n} i \leq \prod_{i=1}^{n} n = n^n$

(累乘表示法将在附录1的A1.3中介绍。) 同样，

$$n!=\prod_{i=1}^{n} i \geq \prod_{i=1}^{n/2} (n/2) = (n/2)^{n/2}$$

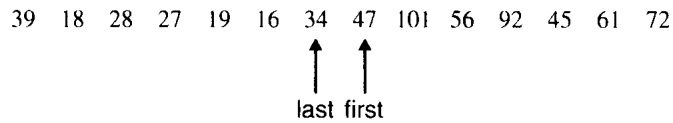
因此$(n/2)^{n/2} \leqslant n! \leqslant n^n$

然后取对数。

12.9 证明用七次比较就足以排序任何五个项的序列。

提示 比较第一个和第二个项；比较第三个和第四个项；再比较前两次比较中较大的项。通过三次比较，得到三个项的有序链，第四个项小于（或等于）链中的某一项。现在将第五个项和链的中间项比较。再用三次比较完成排序。注意$\log_2(5!)>6$，因此使用六次比较是不足以排序任何五个项的序列的。

12.10 使用快速排序在下面的容器中完成第二层的分隔；第一层分隔后有：



假设阈值为1。

12.11 给出一个支持随机访问迭代器的容器（如数组、向量或双端队列），通用型算法nth_element将分隔从迭代器first到last-1之间的项。算法的接口是：

```
//后置条件：对任何满足first<=itr1、itr2<last的迭代器itr1和itr2，
//          如果itr1<=position，那么!(*position<*itr1)，如果position
//          <=itr2，那么!(*itr2<*position)。
template<class RandomAccessIterator>
void nth_element (RandomAccessIterator first,
                  RandomAccessIterator position,
                  RandomAccessIterator last);
```

503

例如，假设v是一个由**int**项组成的vector容器。可以按如下方式输出v中所有项的中值：

```
vector<int>::iterator mid_itr = v.begin( ) + (v.end( ) - v.begin( )) / 2;
nth_element (v.begin( ), mid_itr, v.end( ));
cout << *mid_itr << endl;
```

惠普的实现是使用一个包装函数，它开始时调用一个辅助算法确定项*first的类型。下面是nth_element和辅助算法的定义：

```
template <class RandomAccessIterator>
inline void nth_element(RandomAccessIterator first,
                        RandomAccessIterator nth,
                        RandomAccessIterator last)
{
    __nth_element(first, nth, last, value_type(first));
}

template <class RandomAccessIterator, class T>
void __nth_element(RandomAccessIterator first,
                   RandomAccessIterator nth,
                   RandomAccessIterator last, T*)
{
```

```
    while (last - first > 3)
    {
        RandomAccessIterator cut = __unguarded_partition
            (first, last,
             T(__median(*first, *(first + (last - first)/2), *(last - 1))));
        if (cut <= nth)
            first = cut;
        else
            last = cut;
    }
    __insertion_sort(first, last);
}
```

估算averageTime(n)和worstTime(n)。开发一个运行时间测试来验证这个估算。 504

12.12 在所有排序算法中，最简单的是选择排序。假设容器支持随机访问迭代器——也可以勉强用支持双向迭代器的容器。为了排序从first位置（包括在内）到last位置（不包括在内）的项，需要将最小的项和first位置上的项交换，次小项和first+1位置上的项交换，依次类推。下面是方法接口：

```
// 后置条件：从first（包括在内）到last（不包括在内）的项按升序排列。
//           worstTime(n)是O(n*n)。
template<class RandomAccessIterator>
void selection_sort (RandomAccessIterator first, RandomAccessIterator last);
```

例 假设容器由下列值组成：

59 46 32 80 46 55 87 43 70 81

首先，循环通过first到last-1，找到最小项32位于first+2的位置。交换32和59；容器现在是：

32 46 59 80 46 55 87 43 70 81

其次，循环通过first+1到last-1，找到次小项43位于first+7的位置。交换43和*(first+1)。容器现在是：

32 43 59 80 46 55 87 46 70 81

在下一个循环中，将first+2位置上的项（即59）和first+4位置上的项（即46）交换。容器现在是：

32 43 46 80 59 55 87 46 70 81

依次类推。

实现选择排序法，并提供worstTime(n)和averageTime(n)的估算。

12.13 一种著名的但是效率非常差的排序算法是冒泡排序。从一个支持随机访问迭代器的容器开始——也可以勉强使用支持双向迭代器的容器。在第一次循环迭代中，有一个嵌套的循环将每个项和高一个位置上的项比较，需要时进行交换。第二次（外部循环）迭代返回开始处，然后内部循环比较并交换项。继续这个过程直到容器有序。为了避免不需要的比较，内部循环只进行到外部循环前面的迭代中最后一个交换为止。下面是方法接口： 505

```
//后置条件: 从first（包括在内）到last（不包括在内）的项按升序排列。
//          worstTime(n)是O(n*n)。
template<class RandomAccessIterator>
void bubble_sort(RandomAccessIterator first, RandomAccessIterator last);
```

例 假设容器由下列项组成:

59 46 32 80 46 55 87 43 70 81

外部循环的第一次迭代之后有:

46 32 59 46 55 80 43 70 81 87

这时惟一能保证的就是最大的项位于容器中最后的索引处。外部循环的第二次迭代中，不断进行比较和交换，直到80和70交换了为止:

32 46 46 55 59 43 70 80 81 87

外部循环的第三次迭代中仅有的一次交换是交换43和59，并且当发现59小于70时内部循环停止。

32 46 46 55 43 59 70 80 81 87

进行3次以上的外部迭代后，数组排序为:

32 43 46 46 55 59 70 80 81 87

实现冒泡排序。顺带提一下，内部循环迭代的平均数量（正如读者所猜想的）是:

$$\frac{n^2-n}{2}-\frac{(n+1)\ln(n+1)}{2}+\frac{n+1}{2}\left(\ln 2+\lim_{k\to\infty}\left(\sum_{i=1}^{k}\frac{1}{i}-\ln k\right)\right)$$

$$+(2/3)\sqrt{2\pi(n+1)}+31/36+O(n^{-1/2})\text{中的某些项}$$

正如Knuth(1973)所说，“简而言之，冒泡排序没什么吸引人的，除了一个容易记的名字和它所带来的一些有趣的理论问题”。

12.14 基数排序和本章描述的其他排序方法有着“根本的”不同。这个排序方法基于项的内部表示，而不是基于项之间的比较。因此，worstTime(n)不会好于$O(n\log n)$的限制在这里并不适用。而且实际上，基数排序是worstTime(n)为$O(n\log n)$的算法的基础（见Andersson, 1995）。

基数排序广泛应用于电动机械穿孔卡片排序程序中。感兴趣的读者可以参考Shaffer(1998)。

506

假设想排序一个从first迭代器到last-1之间的支持随机访问迭代器的容器。为了简化，首先把每个项都设置成一个至少包含两个十进制位的非负整数。表示法是基于10的，也可以说是以10为基数的，这正是基数排序得名的原因。除了即将排序的容器之外，还有一个包含10个list容器的lists数组，即十个可能的数字位中的每一个分别对应一个链表。

在第一次外部循环迭代中，容器中的每一项被添加到项的个位（最低有效位）所对应的链表中。然后，从每个链表的头开始，lists[0]、lists[1]等等中的项被存回原容器里。这就重写了原容器。在第二次外部循环迭代中，容器中的每一项被添加到项的十位所对应的链表中。然后，从每个链表的头开始，lists[0]、lists[1]等等中

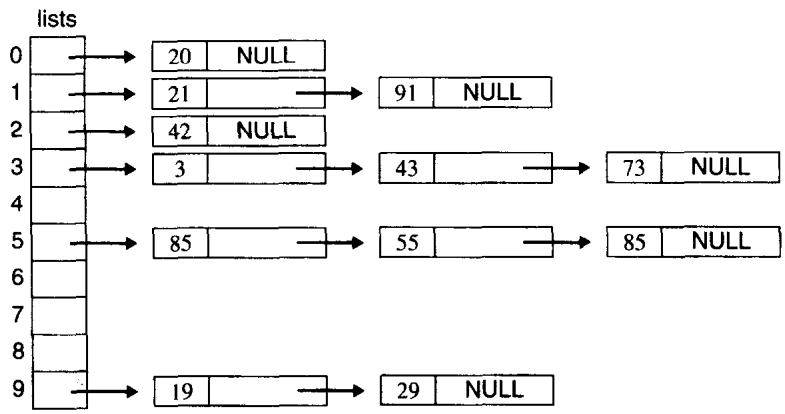
的项被存回原容器里。下面是方法接口:

```
//前置条件: 项是整数。
//后置条件: 从first（包括在内）到last（不包括在内）之间的项按升序排列。
//          worstTime(n)见下面的注释。
template<class RandomAccessIterator>
void radix_sort(RandomAccessIterator first, RandomAccessIterator last);
```

例 假设从下面的容器开始:

85 3 19 43 20 55 42 21 91 85 73 29

其中的每一项都被添加到它的最右位所对应的链表中。(从概念上说，更简单的办法是把每个链表看作一个以NULL终结的单链表，而不是带有头的双链表。）链表的数组如下:

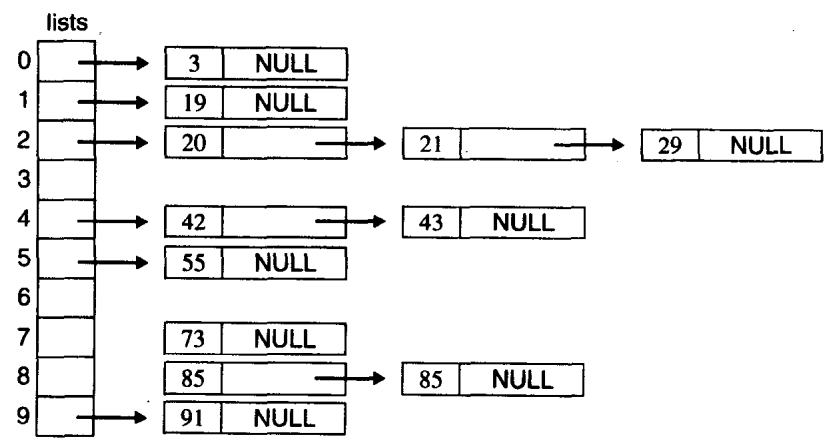


507

然后，从每个链表的头开始，lists[0]、lists[1]等等中的项被存回原容器里:

20 21 91 42 3 43 73 85 55 85 19 29

在下一个外部循环迭代中，容器中的每一项被添加到它们十位所对应的链表中:



最后（由于这些整数至多有两位），从每个链表的头开始，lists[0]、lists[1]等等中的项被存回原容器里，容器现在是有序的:

3 19 20 21 29 42 43 55 73 85 85 91

对任意的非负int项组成的容器实现基数排序。假设N是容器中最大的整数。外部循环迭代的数量至少是ceil($\log_{10}$N)。因此worstTime(n,N)是O(nlogN)。

注意 基数排序中的项是整数，但是稍微做一下改动，项的类型就可以换成string之类的类型。在此情况下string类中的每个字符都应当对应一个链表。

508

编程项目12.1：排序一个文件

将一个文件按升序排序。

分析

输入行将包含即将被排序的文件路径。文件中的每一项将由姓名（名后面是一个空格，随后是姓，然后又是空格，之后是中间名）以及社会保障号码组成。根据姓名排序文件；姓名相同的根据社会保障号码排序。例如，排序之后的文件部分可能如下：

Jones Jennifer Mary 222222222
Jones Jennifer Mary 644644644

为了方便起见，可以假设每个名字都有一个中间名。假设文件persons.dat由以下项组成：

Kiriyeva Marina Alice	333333333
Johnson Kevin Michael	555555555
Misino John Michael	444444444
Panchenko Eric Sam	888888888
Taoubina Xenia Barbara	111111111
Johnson Kevin Michael	222222222
Deusenbery Amanda May	777777777
Dunn Michael Holmes	999999999
Reiley Timothy Patrick	666666666

系统测试1

```
Please enter the path for the file to be sorted.
persons.dat
The file persons.dat has been sorted. Please close this output window when you are
        ready.
```

文件persons.dat现在的组成将是：

Deusenbery Amanda May	777777777
Dunn Michael Holmes	999999999
Johnson Kevin Michael	222222222
Johnson Kevin Michael	555555555
Kiriyeva Marina Alice	333333333
Misino John Michael	444444444
Panchenko Eric Sam	888888888
Reiley Timothy Patrick	666666666
Taoubina Xenia Barbara	111111111

509

使用相同的人名进行随机生成的大型的系统测试。社会保障号码将是随机产生的0~32 767的int项。例如，部分文件可能包含：

aaa 9736
aaa 5917
aaa 8476

提示 如果确信整个文件能够放入内存，那么这将是一个相当简单的问题。但是并非如此。假设想排序类Person中的对象组成的一个大文件。具体地讲，假设Person类中的一个对象占据50个字节，并且数组的最大存储量是500 000字节。因此Person数组的最大容量是10 000。

先将人的文件读入一个数组（或向量），每个块由10 000个人组成。每个块用堆排序方法排序并以交替方式存入两个临时文件之一：leftTop和leftBottom。图12-4说明了该文件排序第一阶段的结果。

然后不断通过交替过程，直到所有项都有序并存储在单个文件中。使用的临时文件是leftTop，leftBottom和rightBottom；personsFile自身扮演了rightTop的角色。在每个阶段，我们将文件顶部和底部的文件对中的两个块归并。结果是两倍大小的块交替地存储在其他的顶部和底部文件对中。把两个文件中的有序块归并成另一文件中的有序、双倍大小块的代码在merge方法中就基本完成了——使用列表而不是文件块。图12-5说明了第一个归并过程。

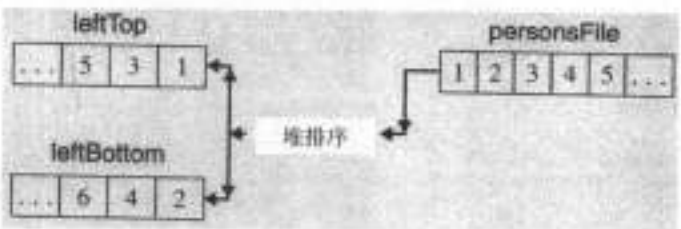


图12-4 文件排序的第一阶段：将personsFile中的每个未排序块插入一个数组，通过堆排序方法排序并存入leftTop或leftBottom

510

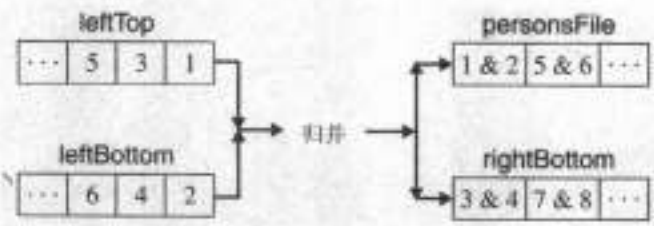


图12-5 文件排序中第一次归并的过程，文件leftTop和leftBottom里包含了有序块，而personsFile和rightBottom里包含了双倍大小的有序块

如果在一次自左至右归并之后rightBottom仍旧是空，那么排序结束并且在personsFile里保存了排序的文件。否则就执行自右至左的归并，之后将检查leftBottom是否仍旧为空。如果是，那么将leftTop拷贝到personsFile中，排序结束。

这将花费多长时间呢？假设在n/k块中有n项，每个块的大小是k。在堆排序阶段，创建n/k块中的每个有序块平均需要大约$k\log_2 k$的时间。每个归并阶段只进行n次迭代，并且有大约$\log_2(n/k)$个归并阶段。总时间就是所有阶段的时间之和，粗略计算是，

$$
\begin{aligned}
(n/k)k\log_2 k + n\log_2(n/k) &= n\log_2 k + n\log_2(n/k) \\
&= n\log_2 k + n\log_2 n - n\log_2 k \\
&= n\log_2 n
\end{aligned}
$$

因为averageTime(n)是最优的，即$O(n\log n)$，所以这样的排序算法经常被用于系统排序实
511 用程序中。

第13章　查找和散列类

在第12章里设计了几个排序算法。现在介绍一个同等重要的查找。先总结几个简单的查找算法揭开介绍新类——hash_map——的序幕。这个类尚未包括在标准模板库中。但是散列映射在某些应用中是非常有用的，比如创建一个符号表。基本上，它的insert、find和erase方法的平均时间花费都是常数！这个显著的性能应归功于一个特殊的称作散列的技术。

目标

1) 理解散列工作的原理，了解什么时候应当使用它，而什么时候不应当使用它。
2) 解释均匀散列设想的意义。
3) 比较不同的冲突处理程序：链式，1-偏移，商-偏移。

13.1　分析查找的框架

在开始考虑查找方法之前，我们需要一个框架来分析它们。因为查找可能成功，也可能失败，所以查找算法的分析应当包含这两种可能性。为每个查找方法估算$averageTime_s(n)$，即成功查找集合中全部n个项的平均时间。可以简单地假设容器中的每一项被查找的几率是相同的。

我们对$worstTime_s(n)$也比较感兴趣，就是成功查找一项所需要的语句最大数量。也就是说，给定一个n值，需要考察n个项的全体置换以及成功搜寻项的所有可能的选择。对每个置换和项，确定查找该项所需要的迭代（或递归调用）次数。那么与最大迭代次数相对应的就是$worstTime_s(n)$。

也可以估算$averageTime_u(n)$，即失败查找的平均时间以及$worstTime_u(n)$。对失败的查找，假设在平均情况下，每个可能的失败都是等几率的。例如，在n项的有序容器里，一个失败查找根据给定项的位置共有$n+1$种可能性：

在容器中第一项之前
在第一项和第二项之间
在第二项和第三项之间
……
在第$(n-1)$项和第n项之间
在第n项之后

13.2节复习了至今使用过的查找方法。

13.2　查找方式复习

迄今为止已经了解到三种不同的查找方式：顺序查找，折半查找和面向树的查找。下面将依次看一下这些查找方式。

13.2.1　顺序查找

通用型算法find有如下的接口：

```
//后置条件: 如果从first（包括在内）到 last（不包括在内）之间的迭代器中
//          有一项等于value，那么返回的迭代器就是满足*i=value的范围内的第一个
//          迭代器。否则就返回last。worstTime(n)是O(n)。
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

514

下面是惠普的实现:

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    while (first != last && *first != value)
        ++first;
    return first;
}
```

find算法表现了在first（包括在内）到last（不包括在内）的迭代器中的**顺序查找**。项*first、*++first等等分别和搜寻值进行比较，直到找到这个值或在规定区域内没有迭代器位于这个值上。由于项是被线性访问的，所以find可以和任意支持前向迭代器的容器类一起使用，也就是可以和迄今为止我们了解的所有除容器配接器之外的容器类一起使用。

顺序查找不论成功与否，平均和最坏时间花费都和n成线性关系。

在一个容器中查找时，假设每项被搜寻到的几率是相等的。因此循环迭代（或递归调用）的平均数量近似为$n/2$。因此$averageTime_s(n)$和n成线性关系。最坏情况下，容器中的最后一项就是要搜寻的项，因此进行了n次迭代，$worstTime_s(n)$仍和n成线性关系。

在失败查找中，断定给定项不在容器里之前，find必须访问全部的n项，因此$average_U(n)$和$worstTime_U(n)$都和n成线性关系。

13.2.2 折半查找

有时我们会预先知道容器是有序的。可以调用通用型算法binary_search来排序项，算法得名是因为搜索段的大小是不停被分成两半的。重复实验10的内容，这里给出该算法的惠普的实现（项根据**operator<**排序）:

```
template <class ForwardIterator, class T>
inline bool binary_search (ForwardIterator first,
                           ForwardIterator last,
                           const T& value)
{
    ForwardIterator i = lower_bound(first, last, value);
    return i != last && !(value < *i);
}
```

回想通用型算法lower_bound将返回位于容器中能够存储item且不会破坏序列顺序的第一个位置的迭代器。lower_bound算法调用了两个都名为__lower_bound的辅助算法之一，但是这两个算法的参数根据迭代器是随机访问迭代器还是前向迭代器而不同。下面是使用随机访问迭代器的版本:

515

```
template <class RandomAccessIterator, class T, class Distance>
RandomAccessIterator __lower_bound (RandomAccessIterator first,
                                    RandomAccessIterator last,
                                    const T& value,
                                    Distance*,
                                    random_access_iterator_tag)
{
    Distance len = last - first;
    Distance half;
    RandomAccessIterator middle;

    while (len > 0)
    {
        half = len / 2;
        middle = first + half;
        if (*middle < value)
        {
            first = middle + 1;
            len = len - half - 1;
        }
        else
            len = half;
    }
    return first;
}
```

在分析binary_search算法之前，注意它返回一个**bool**值，而不是迭代器，因此如果只希望了解项是否在序列中，那么应当使用binary_search。如果想了解项在哪里或者可以插入它的位置，那么应当直接调用lower_bound。

对折半查找而言，无论成功与否，平均和最坏时间花费都和n成对数关系。

对一个支持随机访问迭代器的容器（如数组、向量或双端队列）而言，binary_search比通用型算法find快很多。无论查找是否成功，__lower_bound中的**while**循环将继续下去直到len=0。令n表示first和last之间的距离。因为len开始时使用n的值，所以循环迭代的次数将是n可以除以2直到n=0的次数，并且该次数近似为$\log_2 n$。因此得到$\text{averageTime}_S(n)=\text{worstTime}_S(n)=\text{averageTime}_U(n)=\text{worstTime}_U(n)$，它们都和$n$成对数关系。

回忆在实验10中，为什么找到一个匹配时没有中止__lower_bound中的**while**循环，这是出于效率的考虑。在测试匹配时加入一个额外的比较将耗费更长的时间，除非查找成功并且在较早的迭代中就找到了搜索项。

仅使用前向迭代器的__lower_bound算法版本与使用随机访问迭代器的版本基本一致，除了行

516

```
middle = first + half;
```

被替换为

```
middle = first;
advance(middle, half);
```

iterator.h中的advance算法里包含了一个执行half次的循环，每次循环迭代中都将增加middle（增量为1）。这个__lower_bound版本不论在成功还是失败时，平均和最坏情况下都将花费线性时间。

13.2.3 红黑树查找

在第10章中介绍过红黑树。rb_tree类中的find方法（与find通用型算法相反）和__lower_bound通用型算法相当相似。下面是find方法：

```
iterator find(const Key& k)
{
    link_type y = header; /*最后一个不小于k的节点。*/
    link_type x = root( ); /*当前节点*/

    while (x != NIL)
        if (!key_compare(key(x), k))
            y = x, x = left(x); // 逗号运算符是合法的
        else
            x = right(x);
    iterator j = iterator(y);
    return (j == end( ) || key_compare(k, key(j.node))) ? end( ) : j;
}
```

对红黑树查找而言，无论成功还是失败，平均和最坏情况下的时间都和n成对数关系。

正如__lower_bound通用型算法中一样，不要专门去测试匹配更有效率。红黑树t的高度和n成对数关系，并且find方法总是从根迭代到一个树叶。而从根到任何一个树叶的距离至少是height(t)/2。因此对rb_tree类中的find方法而言，$averageTime_S(n)=worstTime_S(n)=averageTime_U(n)=worstTime_U(n)$，它们都和$n$成对数关系。

本章的其余部分将致力于hash_map类的研究。除了前置条件中的时间估算，hash_map类的方法接口和第10章中的map类没什么实质的分别。

13.3 hash_map类

本节将概述hash_map类的用户接口，该类并不是标准模板库的一部分。回忆前面，键是项的一部分，而且它是访问项的依据。hash_map类是模板化的，而且和map类一样，它的两
517 个模板参数是Key和T。每个值都是一个对，它的第一个组成部分是Key类型，第二个组成部分是T类型的。hash_map容器中任意两个值都不能使用相同的键，并且键是根据相等运算符**operator==**进行比较的。

这里不再提供现成的hash_map类，而是开发一个准系统（只能通过关联数组运算符，运算符[]进行扩张）。这允许我们可以进行更多有关基本性质的讨论。从用户的角度来说，hash_map和map除了时间估算，并没有什么不同。基本上，hash_map中的查找，插入和删除的averageTime(n)都是常数！

下面是方法接口：

```
1. //后置条件：这个hash_map是空。
   hash_map();
```

2. //后置条件：返回这个hash_map中项的数量。
   int size();

3. //后置条件：如果一个包含x的键的项已经被插入到这个hash_map中，那么返回的对就由位于
   //　　早先被插入项的迭代器和false组成。否则，返回的对就由位于新插入项
   //　　的迭代器和true组成。时间估算将在第13.3.6节中探讨。
   pair<iterator,**bool**> insert(**const** value_type<**const** key_type, T>& x);

4. //后置条件：如果这个hash_map里已经包含了键部分是key的值，那么就返回对该值第二个
   //　　组成部分的引用。否则，就将一个新的值<key T()>插入这个映射。时间
   //　　估算将在13.3.6节中探讨。
   T& **operator**[](**const** key_type& key);

5. //后置条件：如果这个hash_map里已经包含了键组件是key的值，就返回位于该
   //　　值的迭代器。否则，将返回和end()返回值相同的迭代器。时间估算将在13.3.6
   //　　节中探讨。
   iterator find(**const** key_type& key);

6. //前置条件：itr位于这个hash_map中的某个值上。
   //后置条件：从这个hash_map里删除itr位置上的值。时间估算将在13.3.6节
   //　　中探讨。
   void erase(iterator itr);

7. //后置条件：返回位于这个hash_map开头的迭代器。时间估算将在13.3.6节中
   //　　探讨。
   iterator begin();

8. //后置条件：返回一个迭代器，它可以用在比较中用来判断迭代通过这个hash_map的
   //　　过程是否应当结束。
   iterator end();

9. //后置条件：这个hash_map对象的空间被回收。
   ~hash_map();

518

为了简单起见，关联迭代器选择的是前向迭代器。仅有的运算符是（后加）**operator++(int)**、**operator***、**operator==**和**operator!=**。这些是迭代通过一个容器所需要的全部运算符。

13.3.1 hash_map类中的字段

在确定hash_map类的字段的过程中，首先回忆一下前一章中的链式或连续表示法。但是如果项是无序的，那么就需要顺序查找，这在平均情况下将花费线性时间。即使项是有序的，并且可以使用这个顺序，最理想的查找也只能获得对数平均时间花费。

本章的剩余部分将致力于说明如何通过散列获得查找、插入和删除的常数平均时间花费。一旦定义了散列是什么以及它是如何工作的，hash_map方法的定义就变得相对简单了。

13.3.2 散列

一开始，让我们先考察使用下面两个字段的连续实现：

buckets：一个值数组

count: hash_map容器中值的数量

首先介绍一个极简单的例子，然后再继续介绍一些更现实的例子。假设数组初始化为保存1000个项——即值，并且每个项由一个三位的键组件——它保存了标识符（ID）号码和一个
519 用来保存姓名的字符串组件。当调用这个hash_map类的构造器时得到的状态如图13-1所示。

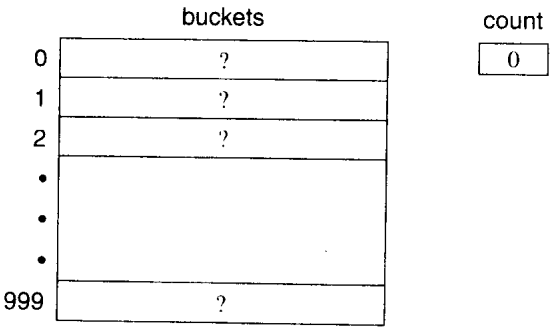


图13-1 一个空hash_map容器的表示法设计

应当在哪个索引处存储键组件是251的项？一个显而易见的选择是索引251。它有几个优点：

1) 值可以直接插入数组中，而不需要访问任何其他的值。

2) 以后对该值的查找只需要访问位置251。

3) 能够直接删除该值而不需要先访问其他值。

图13-2显示了插入三个键为251、118和335的值之后hash_map容器的可能状态；现在省略了字符串组件，因为它们和这个讨论是无关的。

迄今为止还没有做什么大的处理。现在做一个稍微不同的应用，假设数组buckets仍旧保存了1000个项，但是每个项使用了社会保障号码作为键。需要把这个键转换成数组中的下标，
520 并且希望这个转化能够快速地完成，也就是说使用少量的循环迭代。

为了能进行快速的访问，选取社会保障号码最右边的三位并将项存储在索引是该三位的位置上。例如，键为214-30-3261（连字号只是为了增强可读性）的项将被存储在索引261上。同理，键值是033-51-8000的项将被存储在单元0上。图13-3显示了这两次插入操作之后的hash_map容器。

当两个不同的键生成相同的索引时就发生冲突。

读者可能已经注意到这个方案的潜在缺陷：两个不同的键可能有相同的最右边的三位，例如，214-30-3261和814-02-9261。当两个不同的键产生相同的表索引时，就称作**冲突**，并且冲突的键称作**同义词**。不久将处理冲突问题。现在先简单地认识一下：当键空间的大小（也就是合法键值的数量）大于索引空间（即可用单元的数量）时，冲突的可能性就总是存在的。

散列是将一个键转换成一个表索引的过程。

散列是将一个键转换成一个表索引的过程。转换从一个**散列函数**开始：在键上执行一些容易计算的操作并返回一个**unsigned long**（无符号长整型）值，然后将它转化成数组buckets里的一个索引。散列中的其他部分是冲突处理程序。

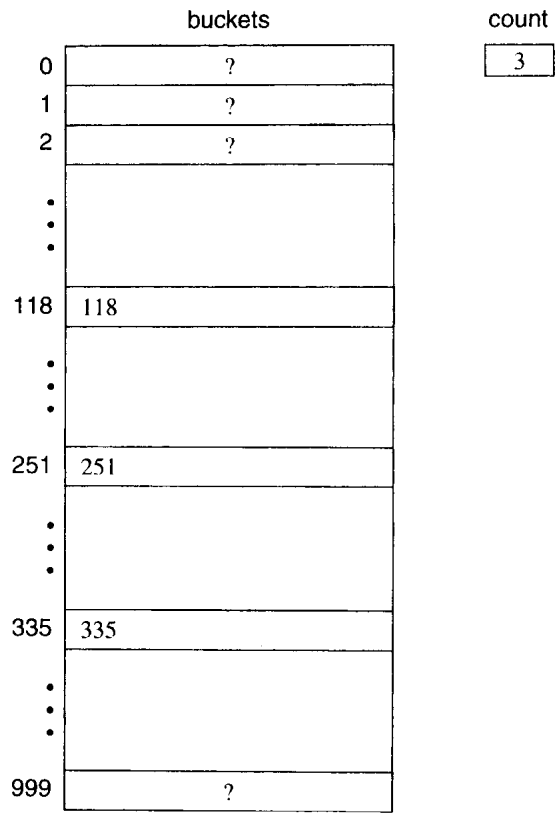


图13-2 在图13-1中插入三个项之后的hash_map容器。只显示了键字段

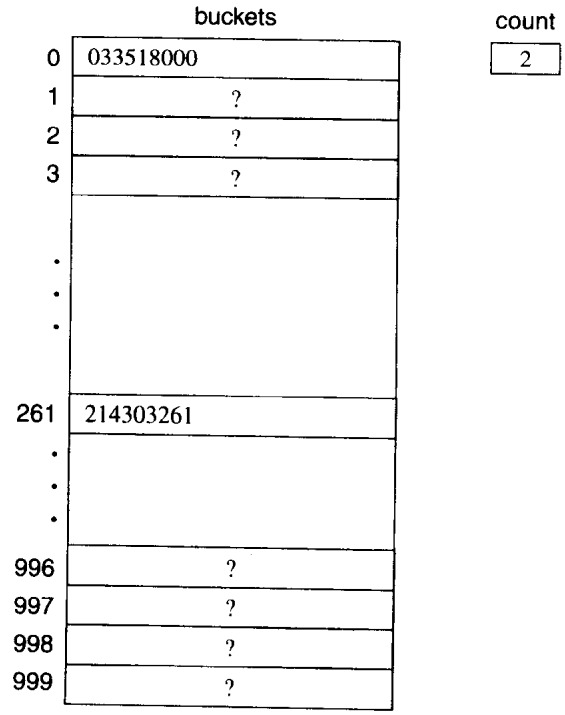


图13-3 包含两个项的hash_map容器。键是社会保障号码。允许用字符串表示一个姓名

冲突处理是散列算法的一个重要部分。

术语**散列**暗示着按某种方式扰乱键；换句话说，用键进行散列。不论是否成功地散列这些键，都有可能发生冲突，因此必须将冲突处理作为散列算法的一部分。下面将研究一个简单有效且十分高效的冲突处理程序：链式。

13.3.3 链式

当使用链式处理冲突时，每个表单元都包含了键被散列到该单元的索引上的全部项的链表。

一种解决冲突的方法是在每个桶条目上存储一个链表，它包含了所有键被散列到该buckets数组中该索引上的项。

```
template<class Key, class T, class HashFunc>
class hash_map
{
    typedef Key key_type;
    typedef HashFunc hash_func;
```

第三个模板参数对应的模板变元是一个函数类，即函数调用运算符——**operator()**——被重载的类。这个被重载的运算符的头是：

```
unsigned long operator()(const key_type& key)
```

例如，如果每个键都是一个**int**，那么可以定义如下的简单的函数类：

```
class hash_func
{
    public:
        unsigned long operator( ) (const int& key)
        {
            return (unsigned long)key;
        } // 重载运算符()
} //  类  hash_func
```

在继续深入之前，先看一个简单的程序，它用了一个hash_map容器来存储一些学生的六位ID号码和名。

```
#include <iostream>
#include <string>
#include "hash_map1.h"

class hash_func
{
    public:

        unsigned long operator( ) (const int& key)
522     {
            return (unsigned long)key;
        } // 运算符()
}; //  类  hash_func

int main( ) {
```

```
    typedef hash_map <int, string, hash_func> hash_map_class;

    const string CLOSE_WINDOW_PROMPT =
          "Please press the Enter key to close this output window.";

    hash_map_class students;
    hash_map_class::iterator itr;

    value_type<const int, string> student(555555, "Mike");
    students.insert (student);

    students.insert (value_type<const int, string> (333333, "Alan"));

    students [111111] = "Bob";

    cout << "size = " << students.size( ) << endl;
    for (itr = students.begin( ); itr != students.end( ); itr++)
          cout << (*itr).first << " " << (*itr).second << endl;
    cout << "looking for " << 333333 << endl;
    itr = students.find (333333);
    if (itr == students.end( ))
          cout << "not found " << endl;
    else
          cout << "found " << (*itr).first << endl;

    cout << "looking for " << 222222 << endl;
    itr = students.find (222222);
    if (itr == students.end( ))
          cout << "not found " << endl;
    else
          cout << "found " << (*itr).first << endl;

    for (int i = 0; i < 500; i++) // ids: 000000, 000001, 000002, . . .
          students [i] = "";

    cout << "size = " << students.size( ) << endl;

    cout << "removing 111111" << endl;
    students.erase (students.find (111111));

    cout << "size = " << students.size( ) << endl;
    if (students.find (111111) == students.end( ))
          cout << "111111 not in map" << endl;
    else
          cout << "oops, 111111 found in map" << endl;                    523

    if (students.find (444444) == students.end( ))
          cout << "444444 not in map" << endl;
    else
          cout << "oops, 444444 found in map" << endl;
    cout << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );

    return 0;
}
```

这个程序的输出是:

```
size = 3
555555 Mike
111111 Bob
333333 Alan
looking for 333333
found 333333 Alan
looking for 222222
not found
size = 503
removing 111111
size = 502
111111 not in map
444444 not in map
Please press the Enter key to close this output window.
```

现在继续hash_map类的设计。在buckets数组的每个索引上将存储一个列表，它当中是所有键被散列到该索引上的项。这个设计（即**链式散列**）的字段是:

```
list<value_type<const key_type, T>>* buckets; // 在buckets数组的每个索引上
                                               // 将存储所有其键被散列到该
                                               // 索引上的项组成的列表。
int count,          //这个hash_map中项的数量
    length;         //这个hash_map中桶的数量
hash_func hash;     //hash是一个函数对象
```

每个表中的项构成了一个链，这也正是术语**链式**的由来。每个list对象被看作一个**桶**，而且每个项都被存储在一个list类的节点中。回忆在第6章中，每个节点都有data（也就是项）、prev和next字段。prev字段在hash_map类里是没有用的，因此假设每个节点将由一个item字段和一个next字段组成。

524 为了解释链式散列是如何操作的，考虑使用社会保障号码作为键来存储1000个项。每个项——即值——由键和姓名组成。因为每个键都是**int**型的，消息hash(key)只不过是返回键自身，因此索引是key%1000。最初，每个单元中将包含一个空列表。图13-4显示了应用下列语句

```
buckets[hash(key)%1000].push_back(value);
```

插入有如下键的值之后的状况:

214-30-3261
033-51-8000
214-19-9528
819-02-9528
819-02-9261
033-30-8262
215-09-1766
214-17-0261

为了避免图13-4散乱，每个桶的内容都被简化了。严格地说，每个桶包含一个list对象，因此每个桶包含list类中的字段，如__node、__last和__next_avail。在图13-4的每个桶中显示的只有其中的__node->next，它指向列表中的第一个节点。并且假设每个列表是以NULL终结的。

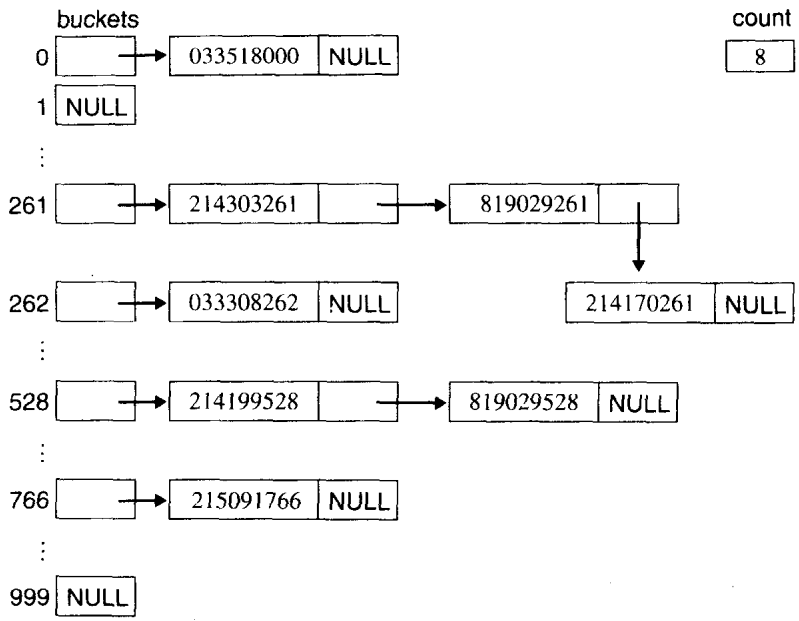


图13-4 这八个项所发送到的链式hash_map容器，每个项是一个<键，姓名>对。在本图中省略了姓名字段

525

为了更好地洞察一些实现的问题，考虑当key_type不是整数类型时的情况。例如，假设每个键是一个至多20个字符的string对象。下面是在hash_func类上的第一次尝试：

```
// 警告：这个函数类只是为了起说明作用。不要使用它！！
class hash_func
{
    public:
        unsigned long operator( ) (const string& key)
        {
            unsigned long total = 0;
            for (unsigned i = 0; i < key.length( ); i++)
                total += key [i];
            return total;
        } // 运算符()
}; // 类hash_func
```

这个类满足了将string键转换成**unsigned long**类型的最小需要，但是它有几个缺点。首先，包含相同字母但是顺序不同的字符串将生成相同的数字。例如，“sewn”和“news”返回的数值都是445（因为**int**('n')=110，**int**('e')=101，**int**('w')=119，**int**('s')=115）。

一个更严重的问题在于这个类可能严重地限制了可用桶的数量。例如，假设每个键仅由

小写字母组成。那么产生的最大总和小于2500（20个z），而很多较小的数将不会产生（例如，123到193）。如果操作很多字符串，将有很多冲突，并且表的查找是顺序的。

一般而言，我们希望每个键产生一个不同的数字。这对至多20个小写字母的字符串是不可能的，因为大约有26^{21}个这样的字符串。这个数远远大于**unsigned long**类型中最大的数值（如果**long**类型的存储占用4个字节，就是2^{32}）。希望满足的条件是：如果L是**unsigned long**的数量，那么两个键产生相同**unsigned long**值的几率是$1/L$。

下面的程序里有一个hash_func类，它也对字符串中的字符进行了合计，但是每部分的总和都乘以了13。最后的总和再乘上一个很大的素数，确保当key.length()很小时也能产生一个大的数字：

```
#include <iostream>
#include <string>
#include "hash_map1.h"

class hash_func
{
526
    public:

        unsigned long operator( ) (const string& key)
        {
            const unsigned long BIG_PRIME = 4294967291;
            unsigned long total = 0;

            for (unsigned i = 0; i < key.length( ); i++)
                total = total * 13 + key [i];
            return total * BIG_PRIME;
        } // 运算符()
}; // 类hash_func
```

下面的main函数创建了一个由学生的姓名和年级平均成绩组成的hash_map容器。每个学生的姓名被散列到hash_map容器里的一个索引上。

```
int main( ) {

    typedef hash_map <string, float, hash_func> hash_map_class;

    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";

    hash_map_class students;
    hash_map_class::iterator itr;

    value_type<const string, float> student("Mike", 3.2);
    value_type<hash_map_class::iterator, bool>p = students.insert (student);

    students.insert (value_type<const string, float> ("Alan", 3.6));
    students ["Bob"] = 3.1;
    for (itr = students.begin( ); itr != students.end( ); itr++)
        cout << (*itr).first << " " << (*itr).second << endl;

    (*(p.first)).second = 1.7;
```

```
    students ["Bob"] = 3.04;
    cout << "size = " << students.size( ) << endl;
    for (itr = students.begin( ); itr != students.end( ); itr++)
        cout << (*itr).first << " " << (*itr).second << endl;

    cout << "looking for " << "Alan" << endl;
    itr = students.find ("Alan");
    if (itr == students.end( ))
        cout << "not found " << endl;
    else
        cout << "found " << (*itr).first << endl;

    cout << "looking for " << "Jack" << endl;
    itr = students.find ("Jack");                                    527
    if (itr == students.end( ))
        cout << "not found " << endl;
    else
        cout << "found " << (*itr).first << endl;

    for (int i = 0; i < 500; i++)
    {
        string temp = "";
        students [temp + char (i)] = 2.0;
    } // for

    cout << "size = " << students.size( ) << endl;

    cout << "removing Bob" << endl;
    students.erase (students.find ("Bob"));

    cout << "size = " << students.size( ) << endl;
    if (students.find ("Bob") == students.end( ))
        cout << "no Bob" << endl;
    else
        cout << "oops" << endl;
    if (students.find ("Mike") == students.end( ))
        cout << "no Mike--oops" << endl;
    else
        cout << "Mike found" << endl;

        cout << endl << CLOSE_WINDOW_PROMPT;
        cin.get( );

        return 0;
} // main
```

输出是:

```
Bob 3.1
Mike 3.2
Alan 3.6
size = 3
Bob 3.04
```

```
Mike 1.7
Alan 3.6
looking for Alan
found Alan
looking for Jack
not found
size = 259
528 removing Bob
size = 258
no Bob
Mike found

Please press the Enter key to close this output window.
```

(能够指出为什么输出的从next到last的大小只是259的道理吗？**提示**：总共只有256个不同的ASCII字符。)

均匀散列设想指的是散列键的索引是独立于其他散列键的索引的。

极为重要的一点是，hash_func类要成功地进行散列，产生**int**，使得到的索引分布在整个表中。这个观点称作**均匀散列设想**。大概地说，均匀散列设想指出所有可能的键的集合是均匀分布在所有表索引的集合中的。也就是说，每个键是以同等可能性散列到数组的任意一个索引上的。

对所有的应用来说，全都没有满足均匀散列设想的hash_func类。甚至函数调用运算符返回**unsigned long**参数的hash_func类也不是一个好的选择。例如，假设键是一个9位的雇员编号，其中最右边的四位保存了雇员工作的部门。如果表的大小是10 000，并且当前应用中的所有雇员都在相同部门工作，那么所有的键将被散列到相同的索引上!

hash_func类由hash_map的用户决定，而不是由hash_map类的开发者决定。

重要的是，hash_func类是由hash_map类的用户决定的，而不是由hash_map类的开发者决定的。用户知道键的类型，并能大概了解将键转换成**unsigned long**的最好的方式。在hash_map类中，这个**unsigned long**按照如下方式转换成buckets数组里的一个索引（回忆前面，length字段保存了桶的数量):

```
int index=hash_func(key)%length;
```

现在讨论一下buckets数组应当有多大。如果hash_map类开始时有很多的桶，那将浪费大量的空间。可以采用一个保守的方法来代替：最初桶的数量是相当小的——即211，并可以根据情形增加。什么时候增加呢？因为单独的列表可以是任意大的，所以能在列表数组buckets里存储多于count(=size())的项，因此即便count接近length，也不需要重新调整大小。但是如果总是不调整大小，那么每个列表的平均大小将越来越大。由于列表的查找是顺序的，所以averageTime(*n*)将和*n*成线性关系，而我们的目标是获得平均情况下的常数时间花费。

只要**加载因子**——也就是count和length的比值——超过某些固定值，就将调整大小。这个固定值的定义是

```
const static float MAX_RATIO;
```

529 也就是说，MAX_RATIO是扩充buckets数组之前count和length的最大比值。MAX_RATIO

的值是0.75，因此如果(**float**)count/length>0.75，就将进行扩充。这确保了列表的平均大小将是很小的，事实上就是小于1.0的。

当调整大小时，数组中的每一项必须被散列到新数组中；新数组中项的索引将几乎总是和它在旧数组中的下标不同。顺带提一下，数组没有选择向量的原因是为了进行不用拷贝的扩充。使用向量，扩充总是需要将旧的数组拷贝到新数组的相同索引中，而该额外的、无用的工作将降低散列的速度。

hash_map方法的定义被简化了，因为很多的工作在list类的方法定义过程中就已经完成了。例如，hash_map类中insert方法的定义包含了对list类的push_back方法的调用。但是即使经过了简化，insert方法的定义还是必须处理错综复杂的调整大小问题。

hash_map迭代器允许我们访问list迭代器，这是几个hash_map方法定义所需要的。因此先从hash_map类的嵌入式iterator类的字段和实现开始。

13.3.4 iterator类的字段和实现

为了迭代通过一个hash_map容器中的全部的项，hash_map迭代器必须能迭代通过每一个列表。因此一个hash_map迭代器必须包含一个list迭代器才能迭代通过一个列表。但是hash_map迭代器不能只是一个list迭代器。为什么呢？因为当hash_map迭代器到达一个列表尾部时，必须能到达下一个列表的开头。为了克服这个问题，hash_map类中将包括当前list迭代器的信息，还有该列表的位置——也就是它在buckets数组中的下标。因此iterator类有这样的两个字段：

```
unsigned index;
list<value_type<const key_type, T>>::iterator list_itr;
```

还需要少量的字段，因为一个迭代器需要一些办法来确定当前迭代通过的是哪一个hash_map容器，特别是哪一个buckets数组。因此每个迭代器将有一个buckets字段指向与当前hash_map容器的buckets字段相同的数组：

```
list<value_type<const key_type, T>>* buckets;
```

最后还需要判断迭代器何时到达buckets数组的尾部，因此迭代器类将包含：

```
unsigned length;
```

所有的这四个字段将在hash_map容器的begin方法中获得初始值，详情参阅13.3.5节。 530

除了**operator++**之外，所有迭代器的方法定义都是很直截了当的。例如：

```
bool operator== (const iterator& other) const
{
    return (index == other.index) &&
           (list_itr == other.list_itr) &&
           (buckets == other.buckets) &&
           (length = other.length)
} // ==

value_type<const key_type, T>& operator* ( )
{
    return *list_itr;
} // 运算符
```

后加运算符从调用对象*this的保存开始，然后增加list_itr字段。如果没有位于这个桶的尾部，就返回保存的迭代器。否则就继续前进到剩余的桶，直到（除非）找到一个非空桶。如果确实找到了一个非空桶，就将list_itr字段设置成该桶的开头并返回保存的迭代器。否则，将list_itr字段设置成最后一个桶的尾部并返回保存的迭代器。

下面是它的定义：

```
iterator operator++ (int)
{
    iterator old_itr = *this;
    if (++list_itr != buckets [index].end( ))
        return old_itr;
    while (index < length - 1)
    {
        index++;
        if (!buckets [index].empty( ))
        {
            list_itr = buckets [index].begin( );
            return old_itr;
        } // 桶的条目非空
    }// while
    list_itr = buckets [index].end( );
    return old_itr;
} // 后加运算符++
```

531 这个方法所需要的时间花费将在实现hash_map类之后进行估算。

13.3.5 hash_map类的实现

现在可以完成hash_map类的方法定义，从构造器开始：

```
hash_map( )
{
    count = 0;
    length = DEFAULT_SIZE; // = 211
    buckets = new list<value_type<const key_type, T> > [DEFAULT_SIZE];
} // 缺省构造器
```

begin方法查找整个buckets数组，寻找一个非空的位置：

```
iterator begin( )
{
    int i = 0;
    iterator itr;

    itr.buckets = buckets;
    while (i < length - 1)
    {
        if (!buckets [i].empty( ))
        {
            itr.list_itr = buckets [i].begin( );
```

```
                itr.index = i;
                itr.length = length;
                return itr;
            } // buckets [i] 非空
            i++;
        } // while
        itr.list_itr = buckets [i].end( );
        itr.index = i;
        itr.length = length;
        return itr;
    } // begin
```

end方法的定义要简单些——返回位于最后一个桶中最后一个位置的下一位置的迭代器：

```
iterator end( )
{
    int i = length - 1;
    iterator itr;

    itr.buckets = buckets;
    itr.list_itr = buckets [i].end( );
    itr.index = i;
    itr.length = length;
    return itr;
} // end
```

532

现在考虑一个困难的方法：insert。在正常环境下，插入是很直接的。要插入value_type< **const** key_type, T> x，首先查明x是否已经在hash_map里了：

```
iterator old_itr;
key_type key = x.first;
old_itr = find (key);
if (old_itr != end( ))
```

如果那样将只是返回<old_itr, **false**>对。否则，将key散列到一个buckets索引上，把x添加进该列表并增加count。剩下的工作只是检测是否必须调整buckets数组的大小，也就是说，如果count>**int**(MAX_RATIO*length)。出于模块化的考虑，这个子任务将在辅助方法check_for_expansion中处理。

下面是insert的定义：

```
pair<iterator, bool> insert (const value_type<const key_type, T>& x)
{
    iterator old_itr;
    key_type key = x.first;

    old_itr = find (key);
    if (old_itr != end( ))
        return pair <iterator, bool> (old_itr, false);
    int index = hash (key) % length;
```

```
        buckets [index].push_back (x);
        count++;
        check_for_expansion( );
        return pair<iterator, bool> (find (key), true);
    } // insert
```

如果需要扩充，那就创建一个长度是当前数组长度的两倍加一的新数组。当前数组的每一项都被重新进行散列，然后添加进新数组中的一个链表里，这个新数组随后将取代旧的数组。最后，删除旧数组中每个链表里的每一项。

下面是check_for_expansion的定义：

```
void check_for_expansion( )
{
533         list<value_type <const key_type, T> >::iterator list_itr;
    int index;
    if (count > int (MAX_RATIO * length))
    {
        list< value_type<const key_type, T> >* temp_buckets = buckets;
        length = 2 * length + 1;
        buckets = new list< value_type <const key_type, T> > [length];
        Key new_key;
        for (int i = 0; i < length / 2; i++) // 重新散列旧的数值
            if (!temp_buckets [i].empty( ))
                for (list_itr = temp_buckets [i].begin( );
                        list_itr != temp_buckets [i].end( );
                        list_itr++)
                {
                    new_key = (*list_itr).first;
                    index = hash (new_key) % length;
                    buckets [index].push_back (*list_itr);
                } // 将temp_buckets[i]存回桶中
        // 删除旧的链表
        for (int i = 0; i < length / 2; i++)
            if (!temp_buckets [i].empty( ))
                temp_buckets [i].erase (temp_buckets [i].begin( ),
                                        temp_buckets [i].end( ));
    delete[ ] temp_buckets;
    } // 将桶的大小加倍
} //  方法  check_for_expansion
```

如果进行了扩充，那么buckets数组的长度将被改变，因此必须重新计算每个键所对应的索引。结果是调整大小前在同一个链表里的项在调整之后可能位于不同的链表里。例如，假设数组的长度在调整大小之前是48。那么键为63和111的项将在桶[15]的链表里。调整大小之后，大小变为97；键为63的项将在桶[63]的链表里，而键为111的项将在桶[14]的链表里。

我们将把对insert方法（以及check_for_expansion方法）的分析推迟到开发find和erase方法之后再进行。

与insert方法相比，find方法更容易些：利用通用型算法find查找列表（即桶），找到键散列的索引。如果返回的列表迭代器与end()方法返回的迭代器不同，就将buckets_ptr和index字段填充进一个新创建的迭代器里并返回该迭代器。

下面是代码：

```
iterator find (const key_type& key)
{
    int index = hash (key) % length;                                    534
    iterator itr;
    itr.list_itr = std::find (buckets [index].begin( ), buckets [index].end( ),
                    value_type <const key_type, T> (key, T( )));
    if (itr.list_itr == buckets [index].end( ))
        return end( );
    itr.buckets = buckets;
    itr.index = index;
    itr.length = length;
    return itr;
} // find
```

这里指定了std::find，它强调了调用的是通用型算法find，而不是hash_map的find方法。

erase方法的定义就更简单了：

```
void erase (iterator itr)
{
    buckets [itr.index].erase (itr.list_itr);
    count--;
} // erase
```

hash_map类中的关联数组运算符**operator[]**的定义和第10章中map类中同类运算符使用了同样的单行语句（除了value_type不是<const key_type, T>对的缩写外）：

```
T& operator[ ] (const key_type& key)
{
    return (*((insert(value_type<const key_type, T> (key, T( )))).first)).second;
} // 运算符[ ]
```

完整的hash_map类可以参阅本书网站的源代码链接。

13.3.6 链式散列分析

现在进行均匀散列设想。也就是说，假设每个键独立于其他键的散列，以同等几率散列到数组的任意一个索引上。

如果使用链式散列并满足了均匀散列设想，那么成功或失败查找的平均时间花费都是常数。

令n=count，m=length=桶的数量=列表数量。如果均匀散列设想成立，那么n个项将相当均匀地分布到m个列表里，列表的平均大小是n/m。那么时间估算将依赖于n和m。如果MAX_RATIO=0.75，将总是有：

$$n/m < 0.75$$

每个列表的平均大小至多是0.75。那么平均情况下查找一个列表将需要至多0.75次迭代。
535 也就是说find方法的$\text{averageTime}_S(n, m)$和$\text{averageTime}_U(n, m)$都是常数。

图13-5总结了链式散列的查找时间估算。在每次近似中，时间不是依赖于n，而是依赖于n/m的比值。图13-6估算了使用不同n/m比值的成功和失败查找的循环迭代次数。在图13-5和图13-6的两个表里都假定均匀散列设想成立。erase方法的分析也是相同的，不论在成功或失败的情况下，删除每个项的平均时间花费都是常数。

insert方法怎么样呢？对插入而言，可能需要对付扩充。只要扩充buckets数组，它的长度将加倍（再加1），因此桶将不会被扩充，直到n的大小再次加倍。也就是说，每$2n$次插入只进行一次扩充，因此averageTime(n,m)仍是常数，事实上amortizedTime(n,m)也是常数。实际说来，当发生扩充时，该常数将可能非常大。其他的技术，像渐进式扩充和延伸式扩充，每次只是为数组增加一个小的量并避免大范围的重新散列和重新插入。延伸式散列的讨论见Heileman(1996,pp.232-233)。

对iterator类中的**operator**++，首先计算迭代的总次数并将这个总和除以n。为了遍历整个buckets数组，访问每个项共需要n次迭代，访问m个链表共需要m次迭代，总共是$n+m$次迭代。假设MAX_RATIO=0.75，那么当$n \geq 3m/4$时将调整buckets的大小，并且在调整之后，m的新值将会是它的旧值的两倍（近似值）。等价地讲，在调整之后，m的旧值将是新值的一半。
536 那么在第一次调整大小之后，总是有$3(m/2)/4 < n$。因此$n+m < n+8n/3<4n$。迭代的平均次数是总次数除以n，因此**operator**++的平均迭代次数小于4。也就是说，averageTime(n,m)是常数。

链式：

$\text{averageTime}_S(n, m) \approx \frac{n}{2m}$ 次迭代

$\text{averageTime}_U(n, m) \approx \frac{n}{m}$ 次迭代

图13-5 使用链式散列的成功和失败查找的平均时间估算汇总。采用了均匀散列设想。在图中，n=插入值的数量；m=length=桶的数量=列表数量

$\frac{n}{m}$	0.25	0.5	0.75	0.90	0.99
Successful	0.13	0.25	0.38	0.45	0.50
Unsuccessful	0.25	0.50	0.75	0.90	0.99

图13-6 链式散列下，不同n/m比值对应的查找时间（近似的循环迭代次数）估算。这个图没有假定MAX_RATIO=0.75。事实上，MAX_RATIO甚至可以大于1

最坏时间花费 至今我们一直忽略了对最坏时间花费的讨论。这是散列的致命弱点，就好像最坏时间花费是快速排序中最容易受攻击的方面一样。不论均匀散列设想满足与否，都仍然会有一个数据集合，其中有许多键散列到相同的索引上，导致查找的迭代次数和n成线性关系。因此成功或失败查找的worstTime(n,m)都和n成线性关系。根据相同的原因，可以证明insert和erase方法的worstTime(n,m)与find方法的一样，都和n成线性关系。

如果采用链式散列，不论均匀散列满足与否，成功或失败查找的最坏时间花费都和n成线性关系。

operator++在最坏情况中，buckets[0]里将包含$n-1$项，buckets[$m-1$]里将包含1项。前进

到这最后一项将需要**while**循环的$m-1$次迭代，因此**operator++(int)**的worstTime(n,m)和m成线性关系。

底线是：除非确信均匀散列设想对应用中的键空间是合理的，并且我们主要关心的是平均情况下的性能，否则就使用map容器，以保证最坏情况下能获得对数时间。

13.3.7 value_type类

每个列表中的项都是value_type类型的，并且到目前为止可能都假定value_type的定义是：

```
typedef pair<const key_type, T> value_type;
```

这正是第10章中value_type在map类里的定义。那么在hash_map类中应当有什么差别呢？有差别的原因是hash_map类调用了list类的方法。例如，hash_map类的find方法中的一条语句是：

```
itr.list_itr=std::find(buckets[index].begin(), buckets[index].end(),
                       value_type<const key_type,T>(key,T()));
```

这个语句搜寻了一个列表，但是pair类没有用**operator==**作为列表中项比较的根据。注意比较应当只使用第一个组件，也就是键。

解决这个问题的一种方案是令value_type成为一个模板化的结构；value_type将重载**operator==**。下面是value_type结构：

```
template<class key, class T>
struct value_type
{
      public:
            value_type (const value_type& p): first (p.first)
            {
                  second = p.second;
            }
            value_type (const key& key, const T& t): first(key)
            {
                  second = t;
            }
            bool operator == (const value_type& x)
            {
                  return first == x.first;
            } // 运算符 ==
            key first;
            T second;
}; // value_type结构
```

537

在构造器中，构造器初始化部件（参阅6.1.4节）避免了为**const**字段赋值的问题。**const**字段必须在进入构造器体之前初始化，这也正是需要构造器初始化部件的原因。

13.3.8 应用

毫不费力就可以找到散列的应用。在第10章中开发了一个程序，计算文本中每个单词的

频率。程序借助一个map容器保存<单词，频率>形式的对。通过用一个hash_map容器替换map容器，可以充分减少平均时间花费：

```
#include "hash_map1.h"
typedef hash_map< string, int, hash_func > FrequencyMap;
FrequencyMap frequencies;
```

并且第三个模板变元现在是一个散列函数类。由于使用了一个hash_map容器，所以输出文件将不会是按照字母顺序的：

```
program 1
of 2
a 1
text 2
words 3
big 1
many 1
counts 1
this 1
in 2
occurrences 1
number 1
the 2
it 1
may 1
have 1
538 including 1
```

这个顺序基于单词散列的数组下标。为了使输出文件按照单词的字母顺序排列，迭代通过frequencies并将每个<string, int>对插入map容器：

```
map<string, int>s_frequencies;
map<string, int>::iterator s_itr;

for (itr = frequencies.begin( ); itr != frequencies.end( ); itr++)
    s_frequencies.insert (pair <string, int> ((*itr).first, (*itr).second));
for( s_itr = s_frequencies.begin( ); s_itr != s_frequencies.end( ); s_itr++ )
    out_file << (*s_itr).first << " " << (*s_itr).second << endl;
```

很奇怪的是，将单词-频率对按照字母顺序输出（averageTime(n)是$O(n\log n)$）将比先创建这些对（averageTime(n)只是$O(n)$）花费更长的时间。

举一个非常重要的例子，让我们重新考虑实验19中开发的前缀转换成后缀的程序。在Compiler类里定义了：

```
vector<string> symbolTable;
```

稍后在Token类（Compiler的友元）中有：

```
index = find (symbolTable.begin( ), symbolTable.end( ), referent)
        - symbolTable.begin( );
```

```
if (index == symbolTable.size( ))
    symbolTable.push_back (referent);
```

这个symbolTable的查找在最坏和平均情况下都花费$O(n)$时间，因为symbolTable是一个vector容器。通过用一个map容器替换vector容器，可以将平均和最坏情况的时间花费减少到$O(\log n)$。如果用一个hash_map容器替换vector容器，平均时间花费将减少到$O(1)$，但是最坏时间花费是$O(n)$。这显示了在map容器和hash_map容器之间的一个平衡。在这个应用里，平均时间花费是至关紧要的，因此hash_map容器总是不变的选择。同样，平均时间和最坏时间的平衡考虑使得快速排序成为系统排序应用程序中所有排序方法的首选。

散列对磁盘文件的直接访问也是很有用的。当一个键被散列到一个文件位置上时，只需要一次柱面访问就可以将整个桶引入内存。保持最少的柱面访问是文件处理的主要目的，因为每次柱面访问都需要机电臂在磁盘上的移动。

在实验28中将对比hash_map容器和map容器的运行时间速度。

实验28：hash_map计时　　（所有实验都是可选的）

539

13.4 hash_set类

使用散列，所有的工作都涉及到键-索引的关系。如果值只由键或者还有另一个组件组成是没什么问题的。在前一种情况里，有hash_set；而在后一种情况里则有hash_map。开发hash_set类是相当直截了当的。事实上，可以将hash_set看作一个hash_map，其中的非键字段被忽略。根据这个观点，hash_set类中有一个hash_map字段。下面是简单的实现：

```
template<class Key, class HashFunc >
class hash_set
{
    class T{ };
    hash_map<Key, T, HashFunc> my_hash_map;
    . . .
    iterator find(const key_type& x) const { return my_hash_map.find(x); }
    . . .
} // 类hash_set
```

现在可以重做第9章的拼写检查应用程序，但是用hash_set容器替换AVLTree容器。假设均匀散列设想成立，则令n代表字典文件的大小，并令k代表文档文件的大小。每次在字典文件中查找一个给定单词平均只花费常数时间。因此hash_set版本的compare方法的averageTime(k, n)和k成线性关系；回想一下，这个方法的AVLTree版本的averageTime(k, n)是$O(k\log n)$。综合结果是hash_set版本的compare方法的worstTime(k, n)是$O(kn)$，而AVLTree版本的该时间花费则是$O(k\log n)$。

13.5节探讨了另一个避免使用链表的冲突处理程序。

13.5 开放地址散列

使用链式处理冲突的基本思想是：当一个键被散列到buckets数组中的一个指定索引上时，

该键的值将被插入到buckets[index]上链表的尾部。

使用开放寻址处理冲突是通过在表中查找一个空（也就是“开放”）的单元。

开放寻址提供了另一种方法来处理冲突。使用**开放寻址**，每个表单元只包含一个值；没有链表。要插入一个值，如果该值的键被散列到包含了不同键的索引上，那么将系统地查找表的剩余部分，直到找到一个空的也就是“开放的”单元。

最简单的开放寻址策略是使用1作为偏移量。也就是说，插入一个键被散列到索引j上的值，如果buckets[j]为空，那么该值就被插入到这里。否则就调查索引j+1，然后是索引j+2，等等，直到找到一个开放的位置。图13-7显示了插入下列值时创建的表：

214-30-3261

033-51-8000

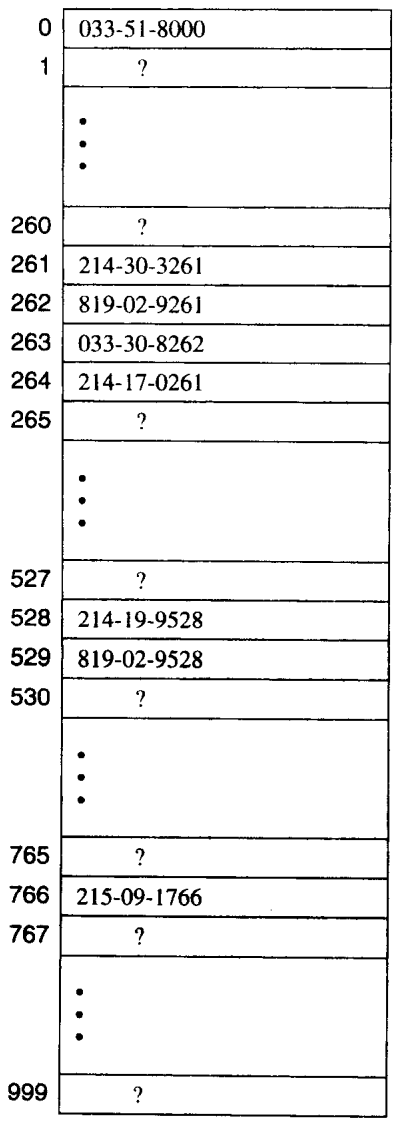


图13-7 插入八个值的表。使用偏移量为1的开放寻址方式来处理冲突

214-19-9528
819-02-9528
819-02-9261
033-30-8262
215-09-1766
214-17-0261

540 ~ 541

为了判断一个单元是否被占用，value_type类将使用一个**bool**字段occupied。当构造一个散列映射时，表中每个值的occupied字段将被设置成**false**。当insert方法将一个值插入表中时，该值的occupied字段就被设置成**true**。图13-8显示了在图13-7的表上加入这个字段后的结果。

	key	occupied
0	033-51-8000	**true**
1	?	**false**
	• • •	
260	?	**false**
261	214-30-3261	**true**
262	819-02-9261	**true**
263	033-30-8262	**true**
264	214-17-0261	**true**
265	?	**false**
	• • •	
527	?	**false**
528	214-19-9528	**true**
529	819-02-9528	**true**
530	?	**false**
	• • •	
766	215-09-1766	**true**
	• • •	
999	?	**false**

图13-8 在表中插入几个值后的结果：插入的每个值对应的occupied字段被设置成**true**

下面是开放寻址的两个小细节：

542

1) 为了确保在有开放单元可用时能找到它，表必须是环绕式的：如果索引length-1上的单

元不是开放的，那么下一个尝试的索引是0。

2) 条目的数量不能超过数组长度，因此加载因子不能超过1.0。当数组总是有至少一个开放（即空的）单元时，它将简化find、insert和erase方法的实现并提高它们的效率；因此需要严格限制加载因子小于1.0。回忆一下，链式索引的加载因子是可以超过1.0的。

现在考虑使用开放寻址且偏移量为1的hash_map类的设计和实现所涉及的内容。我们将引用链式散列设计的全部字段：buckets、count、length和hash，但是buckets将是一个值数组，而不是值列表的数组。同样，嵌入的iterator类也将包含和链式散列相同的first和second字段，还有occupied字段。find、insert和erase方法必须重新定义，因为这些方法的链式版本都访问了链表。

13.5.1 erase方法

在定义find和insert方法之前，需要先定义erase方法，因为删除值的细节对查找和删除会有微妙的影响。这个意思是说，假设希望从图13-8的表中删除键为214-30-3261的值，如果只是将该值的occupied字段设置成**false**，那么将得到图13-9所示的表。

	key	occupied
0	033-51-8000	**true**
1	?	**false**
	• • •	
260	?	**false**
261	214-30-3261	**false**
262	819-02-9261	**true**
263	033-30-8262	**true**
264	214-17-0261	**true**
265	?	**false**
	• • •	
527	?	**false**
528	214-19-9528	**true**
529	819-02-9528	**true**
530	?	**false**
	• • •	
766	215-09-1766	**true**
	• • •	
999	?	**false**

图13-9 将图13-8的表中键为214-30-3261的值的occupied字段设置成**false**，删除该值之后的结果

发现这个删除策略的缺陷了吗？214303261的同义词所取的路径被阻塞了。查找键为819029261的值将失败，因为find方法一旦遇到一个occupied=**false**的值就会停止。为了避免这个问题，在value_type类里添加另一个字段：

```
bool marked_for_removal;
```

当一个值插入表的桶中时，该字段被初始化为**false**。erase方法将该字段设置成**true**。当marked_for_removal字段为**true**，说明它的值不再是散列映射的一部分，但是允许1-偏移的冲突处理程序沿着它的路径继续下去。图13-10显示了进行过八次插入之后的表，图13-11显示了“删除”键为214303261的项之后的结果。

	key	occupied	marked_for_removal
0	033-51-8000	**true**	**false**
1	?	**false**	**false**
	• • •		
260	?	**false**	**false**
261	214-30-3261	**true**	**false**
262	819-02-9261	**true**	**false**
263	033-30-8262	**true**	**false**
264	214-17-0261	**true**	**false**
265	?	**false**	**false**
	• • •		
527	?	**false**	**false**
528	214-19-9528	**true**	**false**
529	819-02-9528	**true**	**false**
530	?	**false**	**false**
	• • •		
766	215-09-1766	**true**	**false**
	• • •		
999	?	**false**	**false**

图13-10 插入八个值后的表。使用1-偏移的开放寻址方式来处理冲突。在每个值中，只显示了key（也就是first）、occupied和marked_for_removal字段

现在键为819029261的值的查找将是成功的。因为erase方法的参数是一个迭代器，该迭代

器的索引字段提供了对即将删除的值的访问。定义如下:

```
void erase (iterator itr)
{
    buckets [itr.index].occupied = false;
    buckets [itr.index].marked_for_removal = true;
    count--;
} // erase
```

	key	occupied	marked_for_removal
0	033-51-8000	true	false
1	?	false	false
	• • •		
260	?	false	false
261	214-30-3261	false	true
262	819-02-9261	true	false
263	033-30-8262	true	false
264	214-17-0261	true	false
265	?	false	false
	• • •		
527	?	false	false
528	214-19-9528	true	false
529	819-02-9528	true	false
530	?	false	false
	• • •		
766	215-09-1766	true	false
	• • •		
999	?	false	false

图13-11 发送消息erase(214303261)之后图13-10中表的内容

543 ~ 544

开发完find和insert方法之后再来分析erase方法。

find方法不断进行循环，直到找到一个未占用的单元或匹配的键。只有当一个值没有删除标记时才检测该值的键。定义如下:

```
iterator find (const key_type& key)
```

```
{
    unsigned long hash_int = hash (key);
    int index = hash_int % length;                                    545
    iterator itr;

    while (buckets [index].marked_for_removal
            || (buckets [index].occupied && buckets [index].first != key))
        index = (index + 1) % length;
    if (!buckets [index].occupied && !buckets [index].marked_for_removal)
        return end( );
    itr.buckets = buckets;                                            546
    itr.index = index;
    itr.length = length;
    return itr;
} // find
```

不破坏1-偏移路径而删除值的marked_for_removal方法又对check_for_expansion方法提出了一个问题。问题是确定重新散列的适当条件。先从链式散列使用的条件开始:

```
count>int(MAX_RATIO*length)
```

使用这个条件决定是否用开放寻址重新散列是错误的，假设length字段是1000并进行大量的交叉插入和删除。例如，假定有980次插入和780次删除。那么count字段值将是200，显然表中还将剩余大量的空间。但将只有20个未占用的单元。那么使用find方法的平均情况下的失败查找将需要多于15次循环迭代。这是难以接受的。

解决的方法是，只要count的值加上最近删除的次数超过MAX_RATIO*length，就重新散列。“最近”的删除代表着自length值最后一次改变之后所发生的删除。这个解决方法需要在hash_map类里添加一个count_plus字段:

```
int count_plus;//count值+自length最后一次改变后删除的次数
```

当count_plus的值超过MAX_RATIO*length的阈值时将重新进行散列。我们不需要将marked_for_removal值插入新表，因为将为插入的值创建新的路径；也就是说，将重新开始。因此在重新散列之后，count的值将被赋给count_plus。从这时直到下一次重新散列，每插入一个值就增加count和count_plus字段，不过每删除一个值就减少count字段。(习题13.7考虑了一种可行的改进：当count的值超过MAX_RATIO*length时重新散列并且表的大小加倍；当count_plus的值超过MAX_RATIO*length时重新散列但不改变表的大小。)

下面是check_for_expansion方法的定义:

```
void check_for_expansion( )
{
    unsigned long hash_int;
    int index,
        offset,
        old_length;

    if (count_plus > int (MAX_RATIO * length))                        547
    {
```

```
            value_type<key_type, T>*temp_buckets = buckets;
            old_length = length;
            length = next_prime (old_length);
            buckets = new value_type<key_type, T> [length];
            for (int i = 0; i < old_length; i++)
                if (temp_buckets [i].occupied)
                {
                    hash_int = hash (temp_buckets [i].first);
                    index = hash_int % length;
                    while (buckets [index].occupied)
                        index = (index + 1) % length;
                    buckets [index] = temp_buckets [i];
                } // 将temp_buckets[i]送回桶中
                delete[ ] temp_buckets;
            count_plus = count;
        } // 桶的大小加倍
    } // 方法check_for_expansion
```

下面是insert方法的定义：

```
pair<iterator, bool> insert (const value_type<const key_type, T>& x)
{
    key_type key = x.first;
    unsigned long hash_int = hash (key);
    int index = hash_int % length;
    while ((buckets [index].marked_for_removal)||
            buckets [index].occupied && key != buckets [index].first)
        index = (index + 1) % length;
    if (buckets [index].occupied && key == buckets [index].first)
        return pair <iterator, bool> (find (key), false);
    buckets [index].first = x.first;
    buckets [index].second = x.second;
    buckets [index].occupied = true;
    buckets [index].marked_for_removal = false;
    count++;
    count_plus++;
    check_for_expansion( );
    return pair<iterator, bool> (find (key), true);
} // insert
```

由于在count_plus的值超过MAX_RATIO*length时将重新散列，并且需要MAX_RATIO的值严格限制在1.0之内，所以在表中至少总有一个位置是未占用且没有删除标记的。结果是，find方法中的**while**循环保证了最后的终止。

548

13.5.2 主聚类

1-偏移冲突处理程序仍旧有一个恼人的特点：所有散列到给定index上的键将探查相同的路径：index，index+1，index+2，等等。更糟的是，所有散列到该路径上任一索引的键也将

沿着该索引之后的相同路径。例如，图13-12显示了图13-10中的部分表。在图13-12中，散列到261的键的路径是261，262，263，264…… 散列到262的键的路径是262，263，264，265……**簇**是非空单元的序列。使用1-偏移冲突处理程序，簇是由同义词构成的，包括不同冲突的同义词。在图13-12中，索引261、262、263和264上的单元组成了一个簇。每当一个条目添加进簇，该簇不仅变大，而且成长得更快，因为任何散列到新索引上的键将沿着簇中已存储的键的相同的路径。**主聚类**是当冲突处理程序允许加速簇成长时发生的现象。

主聚类出现在开放寻址冲突处理程序允许加速簇成长时。

显然，1-偏移冲突处理程序是易受主聚类影响的。主聚类存在的问题是在查找、插入和删除的过程中顺序遍历的是不断增长的路径。长的顺序遍历是散列的祸根，因此应设法避免这个问题。

如果选择20作为偏移量而不使用1作为偏移量会怎么样呢？我们仍然将得到主聚类：散列到index上的键将重叠散列到index+20、index+40等索引处的键的路径上。事实上，这会引起比主聚类更大的问题！例如，假设表的大小是100并且偏移量是20。如果一个键被散列到33，那么只有下列索引的单元才允许出现在该簇中：

33 53 73 93 13

一旦填满了这些单元，那么就不能再插入任何散列到这些索引上的键对应的值。出现这额外问题的原因是偏移和表的大小有一个公因子。令表的大小为素数可以避免这个问题，但是仍有主聚类的问题。

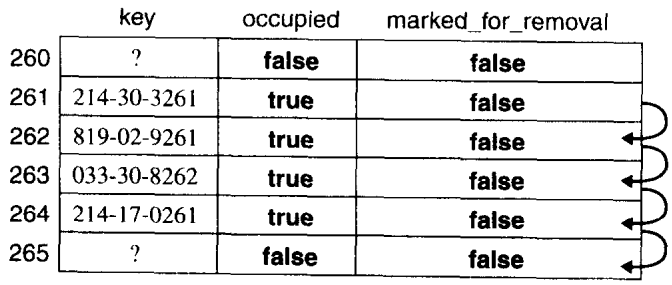


图13-12 散列到261的键的路径重叠了散列到262、263、264的键的路径

549

1-偏移（或任何线性偏移）冲突处理程序导致主聚类问题的原因是由于任何键的偏移都是相同的。13.5.3节中解答了主聚类的问题，令每个键不仅确定索引，而且还确定偏移量。

13.5.3 双散列

如果使偏移量依赖于键而不是所有的键使用相同的偏移，那么主聚类的问题是可以避免的。例如，可以设置

```
offset=hash_int/length;
```

要了解在简单的环境中这是如何工作的，向大小为19的表中插入下列键：

33

72

71

55
112
109

这些键的创建不是随机的，不过可以说明，不同冲突甚至是相同冲突的键不会遵循相同的路径。下面是相应的余数和商：

键	键%19	键/19
33	14	1
72	15	3
71	14	3
112	17	5
55	17	2
109	14	5

第一个键33被存储在索引14，而第二个键72被存储在索引15。第三个键71散列到14，但是该单元已被占用，因此索引14加上偏移3得到索引17，71就被存储到这个单元里。第四个键112散列到17（已占用），索引17加上偏移5；因为22已经超出了表的范围，所以用22%19，即3，它是一个未被占用的单元。键112就被存储到索引3。第五个键55散列到17（已占用），然后散列到(17+2)%19，即空单元0。第六个键109散列到14（已占用），然后散列到(14+5) %19，即0（已占用），然后是（0+5）%19，即一个未占用的单元5。图13-3显示了这些插入之后的结果。

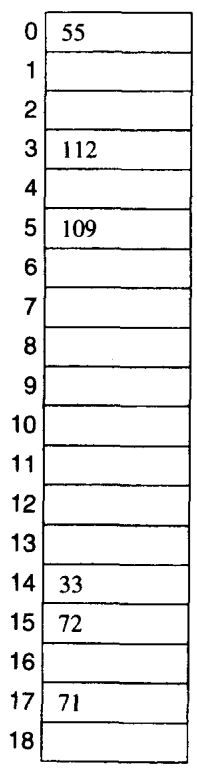


图13-13 向表中插入六个值的结果；冲突处理程序使用散列值和表大小的商作为偏移

这个冲突处理程序是著名的**双散列**（因为索引和偏移都是通过散列键获得的），或者称作**商-偏移**冲突处理程序。在接触代码和分析之前还需要处理最后一个问题：如果偏移是表大小 550
的倍数会怎样？例如，假设要将键为736的条目添加进图13-13的表中，有：

736%19=14

736/19=38

因为单元14已被占用了，所以下一个尝试的单元是（14+38）%19，得到的还是14！为了避免陷入这种僵局，只要key/length是length的倍数就使用1作为偏移量。这是一个很罕见的现象：平均每m个键中才发生一次，其中m=length。习题13.8证明了如果使用这个冲突处理程序并且表的大小是一个素数，那么任何键的偏移序列将会覆盖整个表。

下面对1-偏移版本稍做改进，得到双散列的insert方法：

```
pair<iterator, bool> insert (const value_type<const key_type, T>& x)
{                                                                    551
    key_type key = x.first;
    unsigned long hash_int = hash (key);
    int index = hash_int % length,
        offset = hash_int / length;
    if (offset % length == 0)
            offset = 1;
    while ((buckets [index].marked_for_removal)||
            buckets [index].occupied && key != buckets [index].first)
        index = (index + offset) % length;
    if (buckets [index].occupied && key == buckets [index].first)
            return pair <iterator, bool> (find (key), false);
    buckets [index].first = x.first;
    buckets [index].second = x.second;
    buckets [index].occupied = true;
    buckets [index].marked_for_removal = false;
    count++;
    count_plus++;

    check_for_expansion( );
    return pair<iterator, bool> (find (key), true);
} // insert
```

check_for_expansion方法结合考虑了商-偏移和对素数大小的表的需要：

```
void check_for_expansion( )
{
    unsigned long hash_int;
    int index,
        offset,
        old_length;

    if (count_plus > int (MAX_RATIO * length))
    {
        value_type<key_type, T> * temp_buckets = buckets;
```

```
            old_length = length;
            length = next_prime (old_length);
            buckets = new value_type<key_type, T> [length];
            for (int i = 0; i < old_length; i++)
                if (temp_buckets [i].occupied)
                {
                    hash_int = hash (temp_buckets [i].first);
                    index = hash_int % length;
                    offset = hash_int / length;
552                 if (offset % length == 0)
                        offset = 1;
                    while (buckets [index].occupied)
                        index = (index + offset) % length;
                    buckets [index] = temp_buckets [i];
                } // 将temp_buckets[i]送回桶中
            delete[ ] temp_buckets;
            count_plus = count;
        } // 桶大小加倍
    } // 方法check_for_expansion
```

next_prime方法返回至少是变元两倍的最小素数。

```
//后置条件: 返回至少是p的两倍的最小的素数。
int next_prime (int p) {

    int i;

    bool factorFound;

    p = 2 * p + 1;
    while (true)
    {
        i = 3;
        factorFound = false;

        // 检测3,5,7,9,...,sqrt(p)是否是p的因子
        while (i * i < p && !factorFound)
            if (p % i == 0)
                factorFound = true;
            else
                i += 2;
        if (!factorFound)
            return p;
        p += 2; // 获取下一个备选素数
    } // 当没有找到素数时
} // 方法next_prime
```

对1-偏移版本的find方法所做的惟一改动是通过hash(key)/length计算偏移，并且在**while**循环中，index是和偏移量而不是1相加:

```
iterator find (const key_type& key)
```

```
{
    unsigned long hash_int = hash (key);
    int index = hash_int % length,                                    553
     offset = hash_int / length;
  if (offset % length == 0)
      offset = 1;
  iterator itr;
  while (buckets [index].marked_for_removal
          || (buckets [index].occupied
           && buckets [index].first != key))
      index = (index + offset) % length;
  if (!buckets [index].occupied && !buckets [index].marked_for_removal)
      return end( );
  itr.buckets = buckets;
  itr.index = index;
  itr.length = length;
  return itr;
} // find
```

13.5.4节说明了使用商偏移和素数大小的表的开放寻址方式所表现出的预期性能。如果采用均匀散列设想，那么插入、删除和查找的平均时间花费都是常数。

13.5.4 开放地址散列分析

现在估算使用开放地址散列方式调用find方法的成功和失败查找的时间。特别是，假设使用商偏移，表的大小是素数，并且满足均匀散列设想。

正如在链式散列分析中所做的，还是使用m作为length的值，n作为count的值。采用循环迭代次数作为成功和失败查找的平均和最坏时间花费的估算。当表中已经有$k>0$个值时，失败查找需要的迭代次数恰好和插入第$(k+1)$个值需要的迭代次数相同。并且这也正是成功查找第$(k+1)$个值所需要的迭代次数。

对任何满足$0 \leq k<m$的任意k，定义

$$E(k,\mathrm{m})$$

为插入第$(k+1)$个值时所需要的预期迭代次数。

根据刚才提出的第$(k+1)$个值的插入和第k个值的失败查找之间的关系，find方法应满足下面的估算：

$$\mathrm{averageTime}_U(n,m) \approx E(n,m)\text{次迭代}$$

因此我们将计算$E(n,m)$来估算find方法的$\mathrm{averageTime}_U(n,m)$。无疑，

$$E(0,m)=1 \qquad \text{对任意的} m>1$$ 554

对任意$k>0$，如果第$(k+1)$个值最初散列到一个开放地址，那么$E(k,m)=1$；根据均匀散列设想，这个几率是$(m-k)/m$。否则，根据几率k/m，第$(k+1)$个值最初将散列到一个已占用的地址，因此需要的迭代次数是1加上表的其余部分所需要的迭代次数。但是在表的其余部分所需要的迭代次数恰好是在大小为$m-1$的表中插入第k个值需要的迭代次数，即$E(k-1,m-1)$。

将上述最后一段论述编写成一个等式，得到

$$E(k,m)=\frac{m-k}{m}*1+\frac{k}{m}(1+E(k-1,m-1)) \qquad \text{其中} 1\leqslant k<m$$

把这个递推关系和初始条件（即对任意$m\geqslant 1$，$E(0,m)=1$）相结合，可以得到函数E的递归定义。化简递推关系，得到

$$E(k,m)=1+\frac{k}{m}E(k-1,m-1) \qquad \text{其中} 1\leqslant k<m$$

在例A1.6中，开发了这个等式的闭合形式；也就是说，结果的计算既不需要循环（像求和）也不需要递归调用。结果是:

$$E(k,m)=(m+1)/(m+1-k) \qquad \text{所有满足} 0\leqslant k<m \text{的} m \text{和} k$$

这个猜想可以用数学归纳法证明（对k或m进行归纳）。

现在再将它和find方法的$\text{averageTime}_U(n,m)$联系起来:

$$\begin{aligned}\text{averageTime}_U(n,m)&=E(n,m)=(m+1)/(m+1-n)\\&\approx 1/(1-n/m)\end{aligned}$$

剩下的工作就是估算$\text{averageTime}_S(n,m)$。这近似为向表中插入$n$个值时每次插入需要的平均迭代次数。也就是，

$$\begin{aligned}\text{averageTime}_S(n,m)&\approx\frac{E(0,m)+E(1,m)+\cdots+E(n-1,m)}{n}\\&=\frac{1}{n}\frac{m+1}{m+1}+\frac{m+1}{m}+\frac{m+1}{m-1}+\cdots+\frac{m+1}{m-n+2}\\&=\frac{m+1}{n}\left(\frac{1}{m+1}+\frac{1}{m}+\frac{1}{m-1}+\cdots+\frac{1}{m-n+2}\right)\end{aligned}$$

一个重要的自然对数的基于微积分的属性是:

$$\sum_{j=1}^{k}1/j\approx\ln k \qquad \text{对} k>1 \text{的任意整数}$$

555 当k越大，这个近似值就越准确。那么

$$\frac{1}{m+1}+\frac{1}{m}+\frac{1}{m-1}+\cdots+\frac{1}{m-n+2}\approx\sum_{j=1}^{m+1}1/j-\sum_{j=1}^{m-n+1}1/j\approx\ln(m+1)-\ln(m-n+1)$$

对find方法，现在可以推断

$$\begin{aligned}\text{averageTime}_S(n,m)&\approx\frac{m+1}{n}(\ln(m+1)-\ln(m-n+1))\\&=\frac{m+1}{n}\ln\left(\frac{m+1}{m-n+1}\right)\approx\frac{m}{n}\ln\left(\frac{1}{1-n/m}\right)\end{aligned}$$

这些估算的意义在于说明了成功和失败查找的时间花费只依赖于n/m的比值。例如，如果n/m是0.5，

$$\text{averageTime}_S(n,m)\approx 2\ln\left(1-\frac{1}{2}\right)=2\ln 2\approx 1.39 \text{ 次迭代}$$

$$\text{averageTime}_U(n,m)\approx\left(1/\left(1-\frac{1}{2}\right)\right)=2 \text{ 次迭代}$$

图13-14总结了成功和失败查找（也就是find方法调用）的时间花费估算。为了进行比较，这里还包括了图13-5中有关链式散列的信息。图13-15提供了一些细节：各种n和m的比值所对应的预期循环迭代次数。为了进行比较，也包括了图13-6中有关链式散列的信息。

粗略地观察图13-15，感觉上链式散列比双散列快很多。不过图中给出的只是循环迭代次数估算。运行时间测试则可能是（也可能不是）另一种的情形。运行时间比较是编程项目13.1的主题。

链式:

$\text{averageTime}_S(n, m) \approx \frac{n}{2m}$ 次迭代

$\text{averageTime}_U(n, m) \approx \frac{n}{m}$ 次迭代

双散列:

$\text{averageTime}_S(n, m) \approx \frac{m}{n} \ln\left(\frac{1}{1 - \frac{n}{m}}\right)$ 次迭代

$\text{averageTime}_U(n, m) \approx \frac{1}{1 - \frac{n}{m}}$ 次迭代

图13-14 链式和双散列下，调用find进行成功和失败查找的平均时间估算。
图中，n=插入值的数量；m=表的长度=桶的数量

556

$\frac{n}{m}$	0.25	0.50	0.75	0.90	0.99
链式散列:					
成功	0.13	0.25	0.38	0.45	0.50
失败	0.25	0.50	0.75	0.90	0.99
双散列:					
成功	1.14	1.39	1.85	2.56	4.65
失败	1.33	2.00	4.00	10.00	100.00

图13-15 链式散列和双散列下，调用find进行成功和失败查找的平均循环迭代次数估算。
图中，n=插入值的数量；m=表的长度=桶的数量

即使均匀散列设想适用，在最坏情况下，仍旧能将每个值散列到相同的索引上并产生相同的偏移量。因此双散列下的find方法的$\text{worstTime}_S(n,m)$和$\text{worstTime}_U(n,m)$都和n成线性关系。

链式散列和双散列的所有相关文件可以参阅本书网站的源代码链接。

总结

本章中开发了hash_map类，它在平均情况下的插入、查找和删除只花费常数时间。这个异常的性能是由于**散列**（即把键转换到表索引的过程）。散列算法必须包括一个冲突处理程序，以处理两个键可能散列到相同的索引上的情形。一个广泛应用的冲突处理程序是链式的。在**链式**方法中，hash_map容器被表示成一个列表的数组。每个列表包含了散列到数组中该索引

上的键所对应的项。另一种冲突处理程序是**开放寻址**：当键散列到的索引所对应的表单元包含了另一个键的项时，就查找表的剩余部分，直到找到一个未占用单元或是匹配的键。

均匀散列设想指的是任意键都是以同等几率散列到表的任意索引的情况。关于链式散列，均匀散列设想暗示着表中每个列表的平均大小都小于某些常数。而关于开放地址散列，这一设想暗示着表中项和表的长度的比值被限定在某些常数之上。

557 **加载因子**是表中项的数量和表大小的比值。如果均匀散列设想成立，那么成功和失败查找的时间花费只依赖于加载因子。这对于插入和删除也同样成立。只要加载因子达到某些固定比值，表的大小就加倍，那么插入、删除或查找的平均时间花费是常数。

习题

13.1 用Moo大学中25 000个学生来构造一个hash_map容器。每个学生的键是学生的惟一的六位ID号码。学生对中的第二个组件是学生的年级平均成绩。在hash_map容器中插入几个项。

13.2 用Moo大学中25 000个学生来构造一个hash_map容器。每个学生的键是学生的惟一的六位ID号码。学生对中的第二个组件是学生的年级平均成绩和在班上的名次。在hash_map容器中插入几个值。注意第二个组件不是由单个数据组成的。

13.3 假设已有一个hash_map容器，希望插入一个项（除非它已经在容器中了）。如何实现？

13.4 作为一名使用hash_map类的编程者，开发一个print_sorted函数：

```
//前置条件：键的类中包含运算符<。
//后置条件：按照键的升序输出从first（包括在内）到last（不包括在内）
//          之间的项。
template<class ForwardIterator>
void print_sorted(ForwardIterator first,ForwardIterator last);
```

提示 基本思想是将散列映射拷贝到一个映射中，然后输出映射。但是为了构造一个映射对象，定义中必须包括前两个模板变元（前置条件使得函数类less成为缺省的第三个模板变元）。在第12章的tree_sort方法定义之前已经看到了这个状况。为了确定映射定义的模板变元，print_sorted调用

```
print_sorted_aux(first,last,(*first).first,(*first).second);
```

print_sorted_aux定义的开头是：

```
template <class ForwardIterator, class Key, class T>
void print_sorted_aux (ForwardIterator first, ForwardIterator last, Key, T)
{
    map<Key, T> my_map;
```

558 13.5 对下面的每个方法寻找函数g和h，使得averageTime(n)是$O(g)$，worst-Time(n)是$O(h)$，并且这些都是最小上界。某些方法的g和h可能是相同的。

a. 成功调用（也就是找到项）通用型算法find查找顺序容器（如数组、向量、双端队列或列表）中的项。

b. 调用通用型算法binary_search；假设容器中的项是按顺序排列的，并且容器支持

随机访问迭代器。

c. 成功调用BinSearchTree类中的find方法。

d. 成功调用map类中的find方法。

e. 成功调用hash_map类中的find方法——应当采用均匀散列设想。

13.6 在链式散列的实现中，为什么使用数组取代了向量？

13.7 使用本章中讨论的开放地址策略，只要count_plus的值达到阈值就重新散列并将表大小加倍，尽管count的值可能远远低于阈值。假设用如下行为取代上述做法：只要count的值达到阈值就重新散列并将表大小加倍，而当count_plus达到阈值时就重新散列并且不会将表大小加倍。在什么情形下这种“改进”是低效率的？

提示 如果count的值接近count_plus值会怎样？

13.8 假设p是一个素数。使用模代数证明，对任意正整数*index*和*offset*（*offset*不是p的倍数），下面的容器恰好有p个元素：

$$(index + k*offset)\%p \quad \text{其中} k=0,1,2,...p-1$$

13.9 给定下面的程序段，添加代码输出所有的键，然后添加代码输出所有的值：

```
hash_map<string, int, hash_func>age_map;

age_map.insert (value_type<const string, int> ("dog", 15));
age_map.insert (value_type<const string, int> ("cat", 20));
age_map.insert (value_type<const string, int> ("turtle", 15));
age_map.insert (value_type<const string, int> ("human", 15));
```

13.10 比较链式散列和使用商偏移的开放地址散列的空间需求。假设指针或**int**值占用4个字节，**bool**值占用1个字节。特别是，假定table.length=100003，loadFactor是0.75，且count=count_plus=75000。 559

13.11 在第10章中提到过术语**字典**，可以把它看作一个任意的键-值对集合，它是**映射**的同义词。如果要创建一个真正的字典，那么你是更喜欢将元素存储在一个map容器还是一个hash_map容器中？

13.12 使用采用商偏移冲突处理程序的开放寻址方法，将下列键插入一个大小为13的表中：

20

33

49

22

26

202

508

38

9

下面是相关的余数和商：

键	键%13	键/13
20	7	1
33	7	2
49	10	3
22	9	1
26	0	2
202	7	15
508	1	39
38	12	2
9	9	0

560

编程项目13.1：使用链式和双散列构造一个符号表的运行时间比较

进行运行时间实验，比较链式散列、使用1-偏移的开放寻址和使用商偏移的开放寻址。在每种情况中，使用0.75作为最大加载因子并采用素数表大小（next_prime方法可以生成一个足够大的素数来避免表的重新散列）。hash_map1.h（链式）、hash_map2.h（1-偏移）和hash_map3.h（商偏移）文件可以参阅本书网站的源代码链接。

实验将模拟编译器的一部分，具体说是将标识符散列到一个符号表中。符号表曾经在第7章的中缀转换成后缀的应用中讨论过，并在实验17里实现了它。为了简化，每个元素的值部分将采用空字符串。

561

第14章　图、树和网络

很多情况下，人们往往希望研究对象之间的关系。例如，在一个课程表中，对象是课程，关系就基于是否必修这一先决条件。在航空旅行中，对象是城市；如果两个城市间有班机，那么它们就是相关的。从视觉的角度要求用图的形式描述这样的情形，使用点（称作**顶点**）代表对象，而线（称作**边**）代表关系。本章将介绍几个基于顶点和边的容器。最后将在一个类中定义、设计并实现其中一种容器，其他的结构可以定义为这个类的子类。类和子类目前都不是标准模板库的一部分。

目标

1) 定义有向容器和无向容器情况下的术语**图**、**树**和**网络**。
2) 比较广度优先迭代和深度优先迭代。
3) 理解Prim的查找最小生成树的贪心算法以及Dijkstra的查找顶点间最短路径的贪心算法。
4) 在network类中，比较邻接表设计和邻接矩阵设计。
5) 能够解决包括回溯通过图、树或网络情形的问题。

14.1　无向图

无向图由顶点和称作边的无序顶点对组成。

无向图由称作**顶点**的项和不同的称作**边**的无序顶点对组成。下面是一个无向图：

顶点：*A*，*B*，*C*，*D*，*E*

边：*(A,B)*，*(A,C)*，*(B,D)*，*(C,D)*，*(C,E)*

边中的顶点对被封闭在圆括号里，代表顶点对是无序的。例如，称从*A*到*B*有一条边和从*B*到*A*有一条边是完全相同的。这也是之所以使用“无向”这个词的原因。图14-1描绘了这个无向图，其中每条边用连接它的顶点对的线表示。

根据图14-1可以得到作为顶点和边的容器的无向图的原始公式。而且，图14-1比原始公式更好地领会了无向图。因此从现在开始将用如图14-1所示的形式代替公式进行阐述。

图14-2包含了另外几个无向图。注意边的数量可以小于顶点的数量（如图14-2a和b所示）、等于顶点的数量（如图14-1所示）或大于顶点的数量（如图14-2c所示）。

如果一个无向图包含了所有可能的边，就称它是**完全的**。在一个完全无向图中，边的数量是多少呢？令*n*代表顶点数量。图14-3显示了当*n*=6时边的最大数量是15。

能不能求出一个公式来计算*n*个顶点的完全无向图中边的数量呢（*n*是任意正整数）？一般而言，从*n*个顶点中的任意一个开始，并和其余*n*-1个顶点各构造一条边。然后从这*n*-1个顶点中任意一个开始，和其余*n*-2个顶点各构造一条边（到第一个顶点的边在前一步中已经构造了）。再从这*n*-2个顶点中任意一个开始，和其余*n*-3个顶点各构造一条边。持续这个过程，直到在第*n*-1步中构造最后一条边。构造的边的总数是：

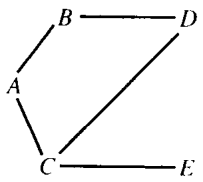


图14-1 无向图的可视表示

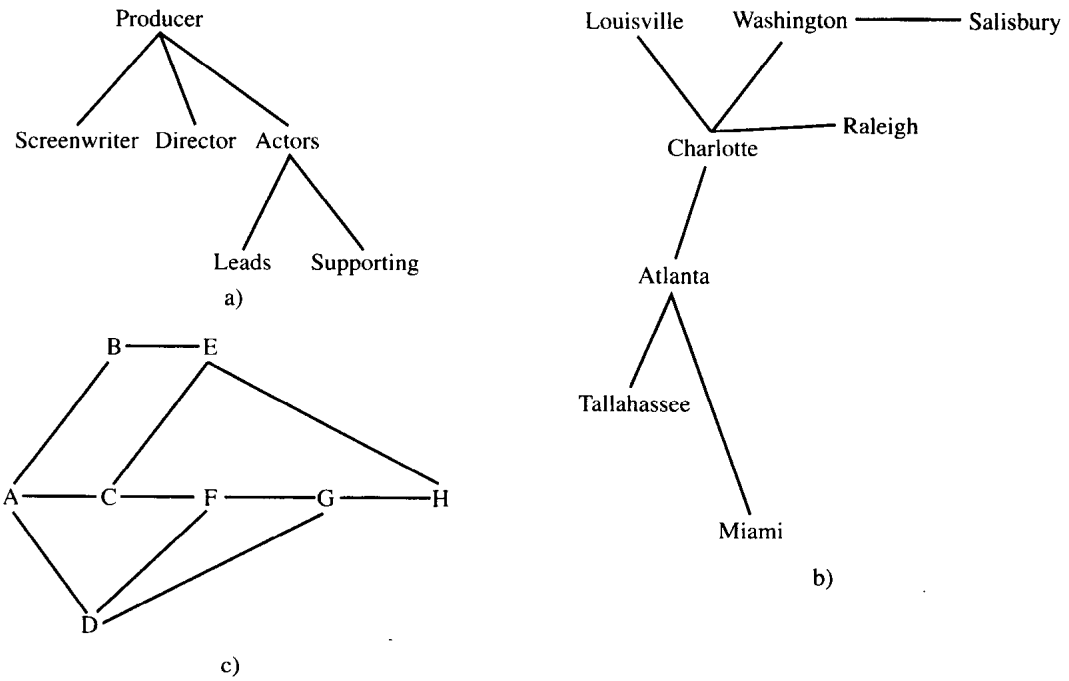


图14-2 a) 有6个顶点和5条边的无向图。b) 有8个顶点和7条边的无向图。c) 有8个顶点和11条边的无向图

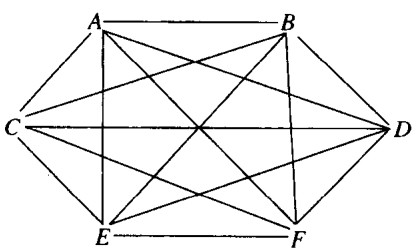


图14-3 6个顶点且具有最大数量（15）的边的无向图

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\sum_{i=1}^{n-1} i = n(n-1)/2$$

这最终的等式可以通过直接归纳n进行证明，也可以从例A1.1中的证明得来——前n个正整数的总和等于$n(n+1)/2$。

564

如果在两个顶点之间有一条边，那么它们就是**邻接的**。例如，在图14-2b中，Charlotte和Atlanta是邻接的，而Atlanta和Raleigh就不是邻接的。邻接的顶点称作**邻居**。

路径是一个顶点序列，其中每对连续的顶点都是一条边。

路径是一个顶点序列，其中每对连续的顶点都是一条边。例如，在图14-2c中，

A，*B*，*E*，*H*

565

是从*A*到*H*的路径，因为(*A*,*B*)、(*B*,*E*)和(*E*,*H*)是边。另一条从*A*到*H*的路径是：

A，*C*，*F*，*D*，*G*，*H*

对*k*个顶点的路径而言，路径的**长度**是*k*−1。换句话说，路径长度是路径上边的数量。例如，在图14-2c中，从*C*到*A*的路径长度是3：

C，*F*，*D*，*A*

事实上还有一条更短的从*C*到*A*的路径，即，

C，*A*

一般来说，在两个顶点之间可能有几条最短路径。例如图14-2c中，

A，*B*，*E*

和

A，*C*，*E*

都是从*A*到*E*的最短路径。

回路是一条路径，它的第一个和最后一个顶点是相同的并且没有重复的边。例如在图14-2b中，

Atlanta, Tallahassee, Miami, Atlanta

是一条回路。在图14-2c中，

B，*E*，*C*，*A*，*B*

是一条回路。同样，

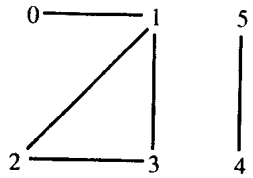
E，*C*，*A*，*B*，*E*

也是一条回路。图14-2a中的无向图是**无环的**，也就是说，它不含任何回路。在该无向图中，

Producer, Director, Producer

不是一个回路，因为边(Producer, Director)是重复的——无向图中的边是无序对。

如果无向图中任两个不同顶点间都有一条路径，那么就说该图是**连通的**。非正式地说，当无向图是“一整块”时就是连通的。例如，图14-1到14-3中所有的图都是连通的。下面的无向图包含6个顶点和5条边，它不是连通的：



566

14.2 有向图

到目前为止还没有涉及到边的方向。如果能够从顶点*V*到达顶点*W*，那么就假设也可以

从顶点W到顶点V。在很多情况下，这个假设可能是不现实的。例如，假设边代表街道，顶点代表十字路口。如果连接顶点V到顶点W的街道是一条从V到W的路，而从W到V可能就没有边相连。

在有向图中，每条边都是顶点的有序对。

有向图由顶点和边组成，其中边是顶点的有序对。例如，下面是一个有向图，它的边用角括号来代表有序对:

顶点：A，T，V，W，Z

边：<A,T>，<A,V>，<T,A>，<V,A>，<V,W>，<W,Z>，<Z,W>，<Z,T>

绘图时，这些边用箭头表示，箭头的方向从有序对的第一个顶点指向第二个顶点。例如，图14-4包含了刚才定义的有向图。

有向图中的路径必须遵循箭头的方向。有向图中**路径**的正式定义是k个顶点（k>1）V_0，V_1，…，V_{k-1}组成的序列，并满足<V_0, V_1>，<V_1, V_2>，…，<V_{k-2}, V_{k-1}>是有向图中的边。例如图14-4中

A，V，W，Z

是从A到Z的路径，因为<A, V>，<V, W>和<W, Z>都是有向图中的边。但是

A，T，Z，W

不是一条路径，因为从T到Z没有边。使用几分钟就可以证明，对图14-4中的任意两个顶点，从第一个顶点到第二个顶点间都有一条路径。

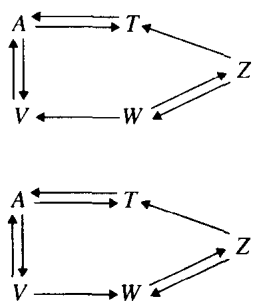


图14-4 一个有向图

有向图D是**连通的**是指对任意一对不同的顶点x和y，从x到y都有一条路径。图14-4是一个连通有向图，但是下面的有向图则不是连通的（试着说明原因）:

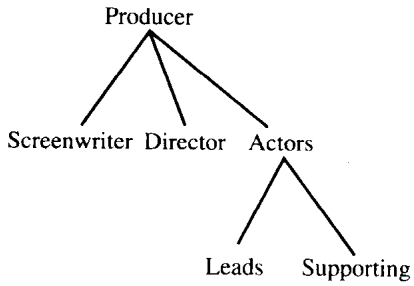
根据这些例子可以看出，实际上可以用“有向图”来定义术语“无向图”：**无向图**是这样一种有向图，其中任意两个顶点V和W，如果从V到W有一条边，那么从W到V也有一条边。这个观察结果将在开发C++类中派上用场。

在14.3和14.4节中将看到图的特殊形式：树和网络。当术语“图”、“树”或“网络”直接出现时，“有向”是一个隐含的前缀。无向的结构将明确地称为“无向图”、“无向树”或“无向网络”。

14.3 树

无向树是一个连通、无环的无向图，其中一项被指定为**根项**。例如，下面是从图14-2a得

到的无向树；“Producer”是根项。



在大多数情况下，人们感兴趣的往往是有向树，也就是一个包含从父亲指向子女的箭头的树。**树**——有时称作**有向树**——是一个有向图，它或者为空，或者包含一个称作**根项**的项，它具有以下特点：

1) 没有边进入根项。

2) 每个非根项上恰好有一条边进入它。

3) 从根项到每个其他项之间有一条路径。

例如，图14-5表明，很容易就可以把图14-2a中的无向树重绘成有向树：通常不需为画树中的箭头担忧，因为方向总是自顶向下的（除了在绘制类和子类的层次结构时）。

第8章中的很多二叉树方面的术语——像“子女”、“树叶”和“分支”——也可以扩展应用到任意的树上。例如，图14-5中的树有四个树叶，高度为2。但是“满”的概念一般不能应用到树上，因为对一个父亲可能的子女数量没有限制。事实上，不能只是将二叉树定义成每个项至多有两个子女的树。为什么不能呢？图14-6有两个不同的二叉树，而它们作为树是等价的。

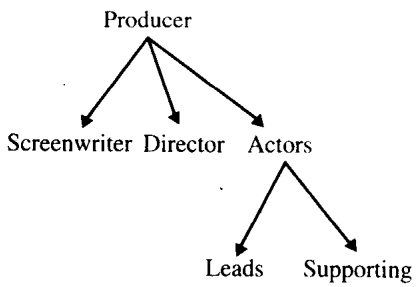


图14-5 一个（有向）树

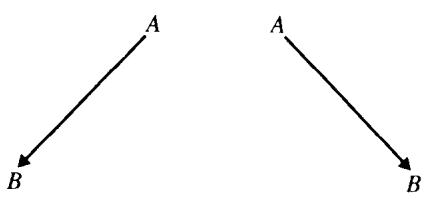


图14-6 两个不同的二叉树，一个包含了空的右子树，而另一个包含了空的左子树

可以将二叉树定义为一个（有向）树，其中每个顶点至多有两个子女，分别标记为“左”子女和“右”子女。树使得我们可以研究如父亲-子女和管理者-管理对象这样的层次关系。 568
并且由于树是任意的，所以不必受二叉树至多是两个子女的限制。

14.4 网络

网络（或**带权图**）是一个图，它的每条边都有一个相关的非负整数，称作是边的**权值**。网络可以是有向的或无向的。

有时会将一个非负整数和图（可以是有向的或无向的）中的每条边相关联。这些非负整数称作**权值**，这样得到的结构称作**网络**或**带权图**。例如，图14-7中是一个无向网络，其中每个权值代表图14-2b的图中城市间的距离。

（有向）网络的价值是什么？也就是说为什么带权值的边的方向是有意义的？即使能够沿着一条边的两个方向行进，但沿着某一方向的权值也可能和另一方向的权值不同。例如，假设权值代表飞机在两个城市间飞行的时间。由于主要是西风，所以从纽约飞往洛杉矶的时间通常要比从洛杉矶飞往纽约的时间长。图14-8显示了一个网络，其中从顶点D到顶点F的边的权值和该边上另一方向的权值不同。

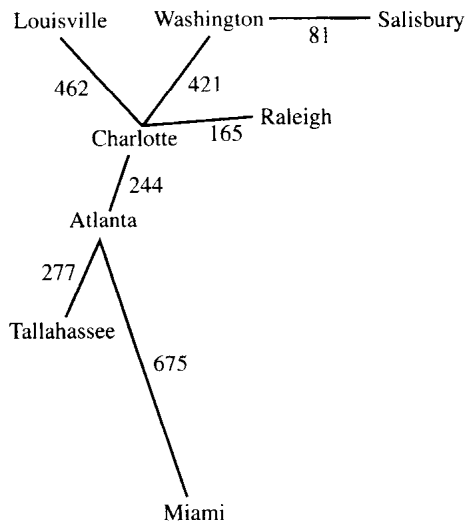


图14-7 一个无向网络，它的顶点代表城市，每条边上的权值代表边中两个城市之间的距离

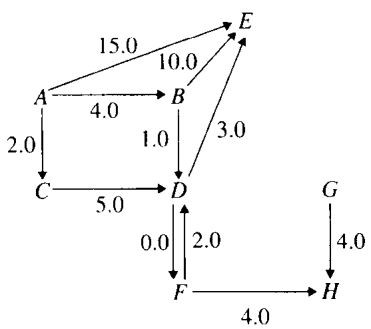


图14-8 8个顶点和11条边的网络

569 对网络中两个顶点间的每条路径，可以计算路径上的权值之和。例如，在图14-8中，路径A、C、D、E的总权值是10.0。能否找到一条从A到E的更短的路径，即总权值更小的路径[⊖]？

⊖ 这和图形意义上的“更短”（也就是路径上边的数量更少）的含义不同。

从*A*到*E*的最短路径是*A*、*B*、*D*、*E*，总权值为8.0。14.5.4节中将开发一个算法，以此寻找网络中两个顶点之间的最短路径（可能会有多条）。

图14-8中的网络不是连通的，因为，比如从*B*到*C*就没有路径。回忆一下，在一个有向网络中的路径必须要遵循箭头的方向。

现在已经了解了如何定义一个图、树或网络，接着将概略描述一些著名的算法。这些算法的实现将在14.7.4节到14.7.6节中开发。 570

14.5 图算法

两个常见的图算法是广度优先迭代和深度优先迭代。

学习其他图算法的先决条件是要能迭代通过一个图，因此先从迭代器开始考虑。我们将关注两种类型的迭代器：广度优先和深度优先。这些术语使我们想起在第8章中曾经学习过二叉树的广度优先和深度优先（也称作前序）遍历。

14.5.1 迭代器

和有向图或无向图关联的迭代器有好几种，它们也可以用于树和网络（有向的或无向的）。首先，可以只是迭代通过图中全部的顶点。迭代不需要遵循什么特别的顺序。例如，下面是迭代通过图14-8所示的网络中的顶点的过程：

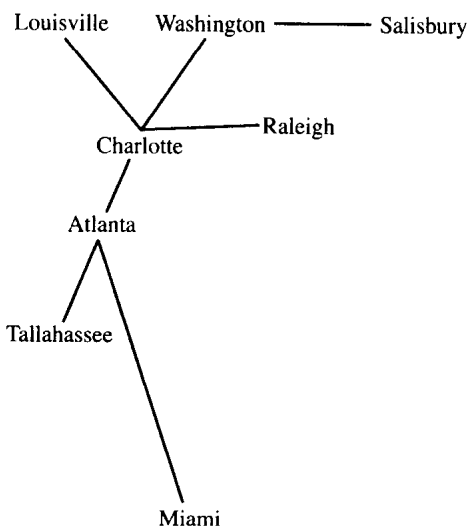
A, *B*, *D*, *F*, *G*, *C*, *E*, *H*

除了迭代通过图中全部顶点之外，有时还要关注迭代通过从给定顶点可达的所有顶点的情况。例如，在图14-8的网络中，可能想迭代通过从*A***可达**的所有顶点，也就是位于从*A*开始的路径上的所有顶点。比如其中一个迭代是：

A, *B*, *C*, *D*, *E*, *F*, *H*

顶点*G*从*A*是不可达的，因此*G*将不在任何从*A*开始的迭代中。

广度优先迭代器 *广度优先迭代器*和第8章中的广度优先遍历相似，先访问出发的顶点，然后是出发点尚未到过的邻居，然后是这些邻居的尚未到达过的邻居，等等。例如，假设下面的图（图14-2b）中所有顶点是按照字母顺序输入的：



571

下面将从Atlanta开始执行广度优先迭代，并按照字母顺序访问邻居。因此开始是:

Atlanta

现在可以到达Atlanta的邻居。当按照字母顺序访问这些邻居时，得到:

Charlotte, Miami, Tallahassee

然后按照字母顺访问Charlotte的尚未到达的邻居，再按照字母顺序访问Miami和Tallahassee的尚未到达的邻居。Charlotte的第一个邻居Atlanta被忽略，因为它已经被访问过了。Charlotte的三个尚未到达过的邻居是:

Louisville, Raleigh, Washington

现在已经访问了Charlotte的所有的邻居，再前进到下一个城市Miami。访问Miami尚未访问过的邻居（没有），以及Tallahassee的（没有）、Louisville的（没有）、Raleigh的（没有）尚未访问过的邻居，然后是Washington尚未访问过的邻居，它们是:

Salisbury

现在就全部完成了，因为Salisbury没有尚未到达过的邻居。换句话说，已经迭代通过了从Atlanta可以到达的所有城市，从Atlanta开始的广度优先迭代是:

Atlanta, Charlotte, Miami, Tallahassee, Louisville, Raleigh, Washington, Salisbury

如果从Louisville开始广度优先迭代，那么城市的访问顺序将是不同的:

Louisville, Charlotte, Atlanta, Raleigh, Washington, Miami, Tallahassee, Salisbury

试着确定从Charlotte开始的广度优先遍历中的城市访问顺序。

在设计breadth_first_iterator类之前先来做一些准备工作。需要了解已经到达过的顶点。因此，将这些到达过的顶点存储在某种类型的容器中。明确地说是想按照顺序访问当前顶点的邻居，而这些邻居最初是被储存在容器里的。由于要按照这些顶点添加进容器的顺序从容器中删除它们，所以队列是一个合适的选择。并且还希望能记住当前顶点。

现在可以为breadth_first_iterator开发一个高级算法——细节将推迟到创建像network这样的可以嵌入breadth_first_iterator类的类时再进行探讨。构造器将开始顶点插入队列，然后把所有顶点都标记为尚未到达的:

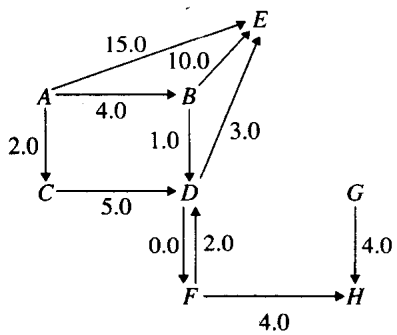
```
breadth_first_iterator (start顶点)
{
    对网络中的每个顶点:
            将它标记为尚未到达的。
572 把start顶点标记为可达的。
    vertex_queue.push(start);
}//构造器的算法
```

后加运算符**operator++(int)**将vertex_queue中前面的顶点取出队列，然后迭代通过该顶点的邻居。每个尚未到达过的邻居将被标记成可达，然后插入队列。

```
breadth_first_iterator operator++(int)
{
    breadth_first_iterator temp=*this;
    current=vertex_queue.front();
```

```
        vertex_queue.pop();
        对current的每个邻居顶点:
                如果该顶点是尚未到达的
                {
                        将该顶点标记为可达的;
                        将该顶点插入vertex_queue队列
                }//if
        return temp;
    }//运算符++(int)的算法
```

举例说明广度优先迭代是如何进行的，假设通过输入图14-9所示的边和权值序列创建一个网络，创建的网络将和图14-8中的网络相同:




```
A B  4.0
A C  2.0
A E 15.0
B D  1.0
B E 10.0
C D  5.0
D E  3.0
D F  0.0
F D  2.0
F H  4.0
G H  4.0
```

图14-9 生成图14-8所示网络的边和权值序列

573

假定处理从*A*开始的广度优先迭代，首先在构造器中将*A*插入队列。第一次调用++将*A*从队列中取出；把*B*、*C*、*E*插入队列；返回位于*A*的breadth_first_iterator。第二次调用++将*B*从队列中取出，把*D*插入队列，并返回位于*B*的breadth_first_ iterator。图14-10显示了从*A*开始的广度优先迭代生成的完整队列。

注意在图14-10中缺少了顶点*G*。这是因为*G*从*A*是不可达的。如果从任何其他顶点开始执行广度优先迭代，访问到的顶点将比图14-10更少。例如，从顶点*B*开始广度优先迭代将依次访问

B, *D*, *E*, *F*, *H*

广度优先迭代器特别适用于树的迭代。开始顶点是根，正如第8章所述，这种迭代器将逐

层访问顶点：根，根的子女，根的孙子，等等。

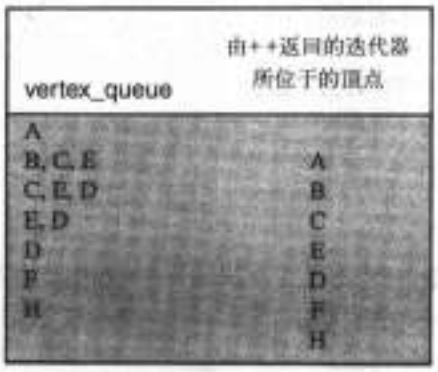
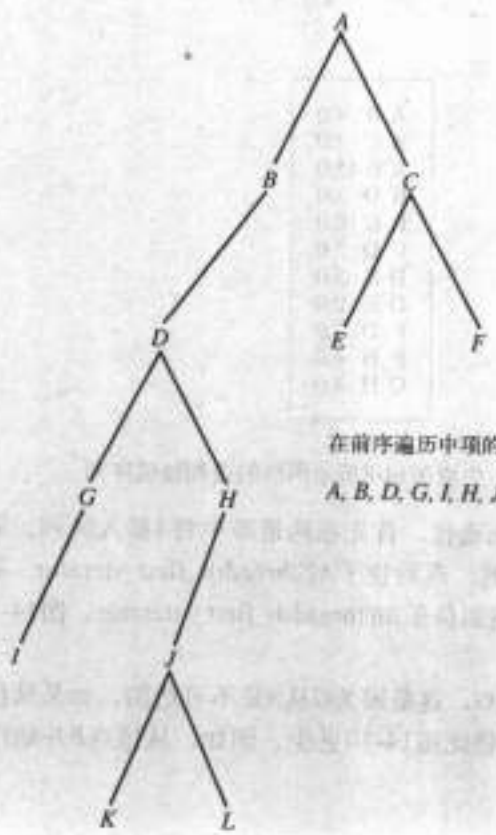


图14-10 从A开始的广度优先迭代的顶点。顶点按照图14-9中输入的顺序被插入队列，然后从队列中取出

深度优先迭代器 另一种专门的迭代器是*深度优先迭代器*。深度优先迭代器是第8章的前序遍历的推广。回忆在第8章里，前序遍历的算法是：

```
preOrder(t)
{
```



在前序遍历中项的访问顺序：

A, B, D, G, I, H, J, K, L, C, E, F

图14-11 一个二叉树以及前序遍历中项将被访问的顺序

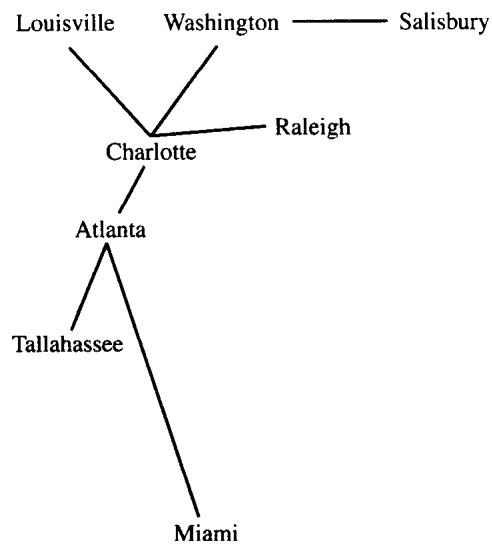
```
    if(t非空)
    {
            访问t的根项;
            preOrder(leftTree(t));
            preOrder(rightTree(t));
    }//if
}//前序遍历
```

574

为了帮助大家回忆前序遍历是如何工作的，图14-11显示了一个二叉树和它的项的前序遍历。可以将一个二叉树的前序遍历描述如下：从根的左侧路径开始。一旦到达该路径的末尾，算法回溯到一个有尚未到达的右子女的项。然后从该右子女开始下一条左侧路径。

图的深度优先迭代要稍微复杂些，因为在图中没有“左”和“右”的概念。首先，将第一个顶点标记为可达的，其他每个顶点都是不可达的。然后，只要还有尚未访问的可达顶点，就访问最近可达的未访问的顶点并将它的不可达的邻居标记为可达的。 575

例如，在下面的图中，从Atlanta开始执行深度优先迭代，假设这些顶点最初是按照字母顺序输入的：



首先访问开始顶点：

Atlanta

然后将Atlanta的邻居Charlotte、Miami、Tallahassee标记为可达的，然后访问最近的可达顶点，即Tallahassee（而不是Charlotte）。Tallahassee没有尚未到达的邻居，因此访问下一个最近可达的顶点Miami。Miami也没有尚未到达的邻居，因此访问Charlotte，并将Charlotte的不可达的邻居Louisville、Raleigh、Washington标记为可达的。

访问最近可达顶点Washington，而且它仅有的不可达邻居Salisbury被标记为可达的。现在Salisbury是最近可达的顶点，因此访问Salisbury。最后访问Raleigh，接着是Louisville。顶点的访问次序如下：

Atlanta, Tallahassee, Miami, Charlotte, Washington, Salisbury, Raleigh, Louisville

从Charlotte开始的深度优先迭代的顶点的访问顺序是：

576 Charlotte, Washington, Salisbury, Raleigh, Louisville, Atlanta, Tallahassee, Miami

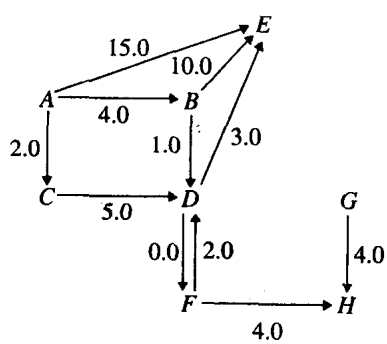
在广度优先迭代中，我们将顶点存储在队列中，使得能够按照顶点存储的顺序访问它们。而在深度优先迭代中，下一个被访问的顶点是最近可达的顶点；因此顶点将使用堆栈而不是队列进行存储。除此之外，深度优先迭代器的基本策略和广度优先迭代器的基本策略是一致的。下面是**operator++(int)**的高级算法：

```
depth_first_iterator operator++(int)
{
    depth_first_iterator temp=*this;
    current=vertex_stack.top();
    vertex_stack.pop();
    对current的每个邻居顶点:
            如果该顶点还不是可达的
            {
                    将它标记为可达的;
                    把该顶点推入vertex_stack
            }//if
    return temp;
}//深度优先迭代中的运算符++(int)的算法
```

正如在广度优先迭代中所做的，假设根据下列输入按照给定次序创建一个网络：

A B 4.0
A C 2.0
A E 15.0
B D 1.0
B E 10.0
C D 5.0
D E 3.0
D F 0.0
F D 2.0
F H 4.0
G H 4.0

创建的网络和图14-8所示的相同：



577

图14-12显示了在根据图14-9的输入生成的图14-8所示的网络中进行深度优先迭代的堆栈状态序列。也可以开发深度优先迭代的回溯版本。回溯版本是等价的，但是可能会利用递归

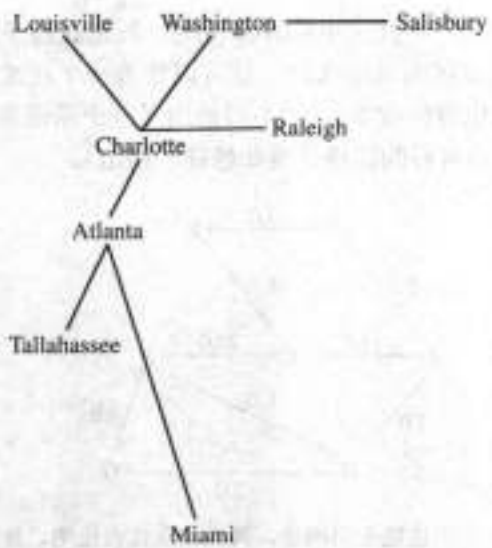
替代显式的堆栈。

Stack (top vertex is shown leftmost)	Vertex where iterator returned by ++ is positioned
A	
E, C, B	A
C, B	E
D, B	C
F, B	D
H, B	F
B	H
	B

图14-12 从A可达的顶点的深度优先迭代。假设顶点是按照图14-9的顺序输入的

14.5.2 连通性

14.1节中定义了**连通的**无向图是指图中任意两个顶点之间都有一条路径。例如，下面就是一个连通无向图：



578

只有当网络是连通的时候，才能执行广度优先或深度优先迭代通过网络（或无向网络）中的全部顶点。事实上，可以借助任意两个顶点之间的迭代来测试连通性。假设itr是通过网络中全部顶点的迭代器。令vertex是一个类，顶点是其中的项。对每个itr++返回的迭代器位于的顶点v而言，令b_itr是从v开始的一个广度优先迭代器。进行检测，确保从v可达的顶点数量（包含v自身）等于网络中的顶点数量。

下面是判断网络连通性的高级算法：

```
//后置条件：如果这个网络是连通的就返回真，否则，返回假。
bool is_connected()
{
    令itr是网络中的迭代器；
    //对每个顶点v，考察从v可达的顶点数量是否等于这个网络中顶点的总数量。
```

```
for(itr=网络的起点; itr!=网络的终点; itr++)
{
        vertex v=*itr;
        //计数从v可达的顶点数量。
        unsigned int count=0;
        breadth_first_iterator b_itr;
        for(b_itr=breadth_first_begin(v);b_itr!=breadth_first_end();
              b_itr++)
            count++;
        if(count<网络大小)
            return false;
}//当itr没有停止迭代时
return true;
}//is_connected方法的算法
```

现在还不能估算这个算法的时间和空间需求，因为它们依赖于迭代器方法等的分析。

在14.5.3节和14.5.4节中，概略描述了两个重要的网络算法的开发。每个算法都是十分复杂的，它们都是根据算法的发明人（Prim，Dijkstra）命名的。

14.5.3 产生最小生成树

假设一个电缆公司要连接一个社区中的许多住宅，假定在房子下面铺设电缆的成本是数百美圆，求住宅连接到电缆系统的最小代价。这可以作为一个连通无向网络问题来处理，每个权值代表两个邻居间铺设电缆的成本。图14-13给出了一个示范设计，某些门户之间的距离
579 没有给出，因为它们代表了不可行的连接（例如越过一座山）。

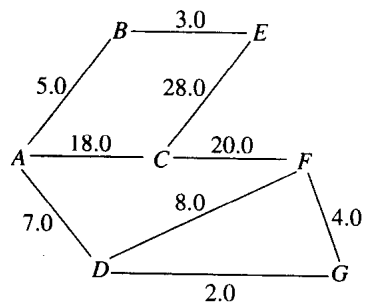


图14-13 一个连通无向网络，其中顶点代表住宅，权值代表连接两个住宅的成本（单位是百美圆）

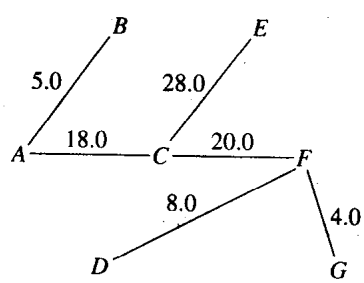


图14-14 图14-13中无向网络的一个生成树

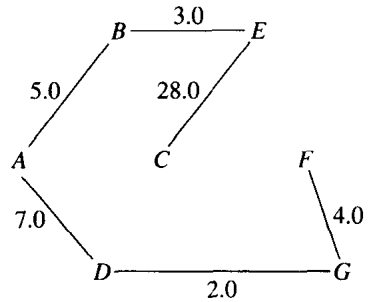


图14-15 图14-13中无向网络的另一个生成树

网络的**生成树**由所有的顶点和一些边以及它们的权值组成。生成树**最小**是指在网络中其他的生成树的权值之和不会更小。

在一个连通无向网络中，**生成树**是由网络所有顶点和一些边（以及它们的权值）组成的树。例如，图14-14和图14-15显示了图14-13中网络的两个生成树。为了简单起见，将顶点*A*指定为每个树的根。

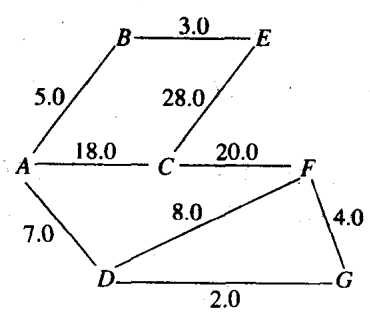
最小生成树是指该树的所有权值之和不大于其他生成树的权值之和。最初的铺设电缆问题可以重申为在一个连通无向网络中构造最小生成树。为了说明解决这个问题的困难程度，现在试着为图14-13中的网络构造一个最小生成树。当解决有1000所住宅的社区问题时，“扩充”你的解决方案将是多么困难？ 580

Prim的算法构造了一个最小生成树。

构造最小生成树的算法应归功于R.C.Prim（1957），这里给出它的基本策略。从一个空的树T以及在网络中选择的任意顶点*v*开始。将*v*添加进T。对每个顶点*w*，(*v*, *w*)是权值为*wweight*的边，将有序三元组<*v*, *w*, *wweight*>保存在一个容器里——很快将会了解是什么样的容器。然后循环，直到T中包含了和原始网络相同数量的顶点。在每次循环迭代中，从容器中删除三元组<*x*, *y*, *yweight*>，*yweight*是容器中所有三元组里最小的权值。如果*y*还不在T里，就将*y*和边(*x*, *y*)添加进T，并把所有满足*z*不在T中且(*y*, *z*)为权值是*zweight*的边的三元组<*y*, *z*, *zweight*>保存进容器。

应当使用什么样的容器呢？容器应按照权值排序；还需要能向容器中添加一个项——即一个三元组，以及从容器中删除权值最小的三元组。**优先队列**将快速完成这些任务：通过基于堆的实现，push方法的averageTime(*n*)是常数，pop的averageTime(*n*)和*n*成对数关系。任何时候，权值最小的三元组都将位于优先队列的顶部。

为了了解Prim的算法的工作原理，从图14-13中的无向网络开始，该图重复如下：



最初，树T和priority_queue对象pq都为空。选取A（也可以选取其他任何顶点），将A添加进T，然后将形如<A, w, $wweight$>的每个三元组添加进pq，其中(A, w)是权值为$wweight$的边。图14-16显示了T的内容和此时的pq。为了更易读，pq中的三元组按照权值的升序显示；严格地说，我们能确认的就是top方法将返回权值最小的三元组，而pop方法将从pq中删除该三元组。

当最小权值的三元组<A, B, 5.0>从pq中删除时，顶点B和边(A, B)以及它的权值被添加进T，而三元组<B, E, 3.0>添加到pq（的顶部）。如图14-17所示。

在下一次迭代中，从pq里删除三元组<B, E, 3.0>，顶点E和边(B, E)添加进T，三元组<E,
581 C, 28.0>被添加进pq。如图14-18所示。

在下一次迭代中，从pq里删除三元组<A, D, 7.0>，顶点D和边(A, D)以及它的权值被添加进T，而三元组<D, F, 8.0>和<D, G, 2.0>则被添加进pq。如图14-19所示。

在下一次迭代中，从pq里删除三元组<D, G, 2.0>，顶点G和边(D, G)以及它的权值被添加进T，而三元组<G, F, 4.0>则被添加进pq。如图14-20所示。

在下一次迭代中，从pq里删除三元组<G, F, 4.0>，顶点F和边(G, F)以及它的权值被添加进T，而三元组<F, C, 20.0>则被添加进pq。如图14-21所示。

T	pq
A	<A, B, 5.0>
	<A, D, 7.0>
	<A, C, 18.0>

图14-16 在图14-13所示的无向网络上应用Prim算法时开始时的T和pq的内容

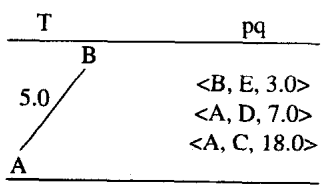
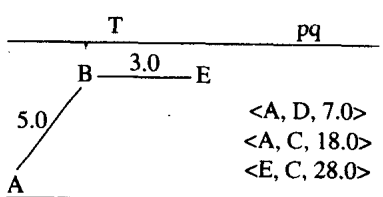


图14-17 在图14-13所示的无向网络上应用Prim算法的过程中T和pq的内容



582 图14-18 在图14-13所示的无向网络上应用Prim算法的过程中T和pq的内容

在下一次迭代中，从pq里删除三元组<D, F, 8.0>。但是没有项加入T或pq，因为F已经在T中了！

在下一次迭代中，从pq里删除三元组<A, C, 18.0>，顶点C和边(A, C)以及它的权值被添加进T；没有项加入pq。没有项加入pq的原因是由于所有C上的边，即(C, A)、(C, E)和(C, F)对中的第二项都已经在T中了。如图14-22所示。即使pq非空，也已经完成了，因为原始网络中的每个顶点都已经在T中了。

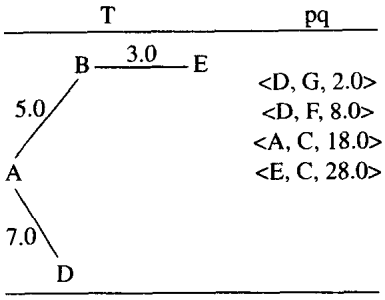


图14-19 在图14-13所示的无向网络上应用Prim算法的过程中T和pq的内容

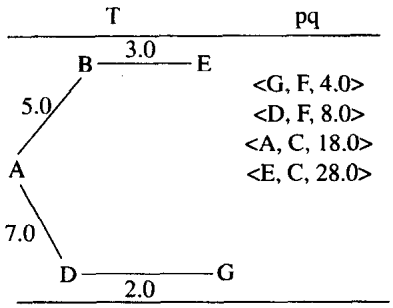


图14-20 在图14-13所示的无向网络上应用Prim算法的过程中T和pq的内容

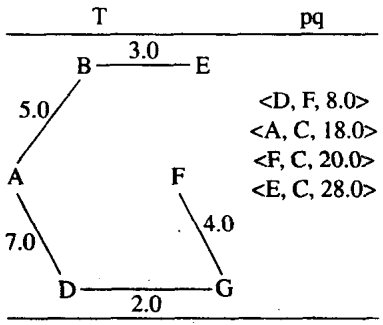
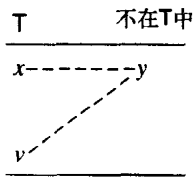


图14-21 在图14-13所示的无向网络上应用Prim算法的过程中T和pq的内容

T的构造方式说明了T是一个生成树。用反证法能够证明T是一个最小生成树。假设T不是一个最小生成树。在某些迭代中，三元组<*x*, *y*, *yweight*>从pq中删除而边(*x*, *y*)加入T，但是在T中还有其他顶点，它的边(*v*, *y*)的权值小于边(*x*, *y*)的权值。可以用图形方式证明: 583



注意由于*v*已经在T中了，所以以<*v*, *y*,?>开头的三元组必定早就添加进pq了。但是边(*v*, *y*)的权值不能小于边(*x*, *y*)的权值，因为从pq中删除的是三元组<*x*, *y*, *yweight*>，不是以<*v*, *y*,?>开头的三元组。这同(*v*, *y*)的权值小于(*x*, *y*)的权值的声明矛盾。因此添加进边(*x*, *y*)的T必定仍是最小的。在14.7.5节中定义了get_minimum_spanning_tree方法。

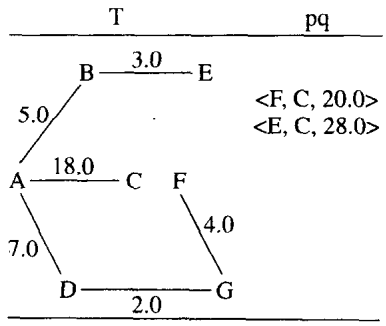


图14-22 在图14-13所示的无向网络上应用Prim算法的最后一次迭代后T和pq的内容

Prim的方法是另一个贪心算法的例子（第11章介绍的霍夫曼编码算法也是一个贪心算法）。在每次循环迭代中，做了局部最优选择：权值最小的边添加进T。这个局部最优（也就是贪心）选择的序列得到了整体最优的解答：T是一个最小生成树。

14.5.4节介绍了另一个贪心算法。习题14.6和实验29证明了贪心并非总是成功的。

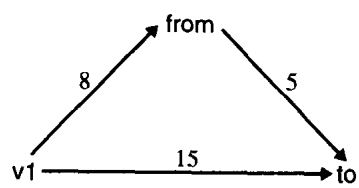
14.5.4 寻找网络中的最短路径

构造最小生成树的策略和下面的寻找网络（或无向网络）中顶点v1到其他某些顶点v2间最短路径的策略很相似。这里的“最短”意味着总权值最小。这两个算法都是贪心的，而且都使用了优先队列。Edsgar Dijkstra（1959）开发的最短路径算法在本质上说是：自v1开始，并当v2的对从优先队列pq中删除时就停止的广度优先迭代。每个对由顶点w和迄今为止从v1到w的最短路径上所有边的权值之和组成。

给定网络中的两个顶点，Dijkstra的算法构造了一条最短路径，即总权值最小的路径。

优先队列是根据最小总权值排序的。我们使用了一个map容器weight_sum来记录总的权值，其中的每个键都是一个顶点w，它映射到迄今为止从v1到w的最短路径上所有边的权值总和。为了能在完成时重新构造最短路径，这里还有另一个map容器predecessor，它的每个键是一个
584 顶点w，而其映射的对象是迄今为止从v1到w的最短路径上的w的直接前任。

基本上，weight_sum记录了迄今为止从v1到其他每个顶点间路径的最小权值。最初，pq由和v1邻接的顶点（以及它们的权值）组成。每次迭代中都贪心地选择了pq中的“顶点-权值”对<from, weight>，它的总权值是pq中所有顶点-权值对中最小的。如果from有一个邻居to，通过路径<v1, ... , from, to>可以减小它的总权值，那么就改变to的路径和最小权值，并将to（以及它的新的总权值）添加进pq。例如，可能有：

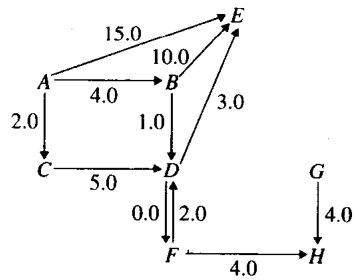


那么从v1到to之间的总权值就减小到13，对<to, 13>加入到pq，而to的前任变成from。如果在v1和v2之间有路径的话，这将最终产生它们之间的最短路径。

开始时，weight_sum为每个顶点关联一个非常大的总权值（如10 000.0），而predecessor

为每个顶点关联一个空顶点。然后由迭代通过v1的邻居来求精这些初始化。对每个邻居w，它的边(v1, w)的权值是weight，weight_sum将w映射到weight，predecessor将w映射到v1，并将顶点-权值对<w, weight>添加进pq。对v1自身而言，weight_sum将v1映射到0.0，predecessor将v1映射到v1。初始化阶段就完成了。

假设想查找图14-8的网络中从*A*到*E*的最短路径，图形重复如下：



初始化的结果如图14-23所示。注意忽略了顶点*G*，因为*G*从*A*是不可达的。

现在开始寻找从*A*到*E*的最短路径（如果有的话）。我们将一直循环，直到找到该路径，或pq为空。图14-23中所示的初始化之后，这个外部循环将第一次执行：从pq中删除对<*C*, 2.0>并迭代（内部循环）通过*C*的邻居。从*C*出发的边上，惟一的顶点是*D*，并且该边的权值是5.0。这个权值加上2.0（*C*的权值之和）是7.0，小于*D*的总权值。因此在weight_sum中，*D*的权值总和改成7.0。图14-24显示了weight_sum、predecessor和pq的结果。 585

图14-24指出目前从*A*到*D*的权值最小的路径的总权值是7.0。在外部循环的第二次迭代中，从pq里删除<*B*, 4.0>并迭代通过*B*的邻居，即*D*和*E*。结果如图14-25所示。

此时，到*D*的权值最小的路径的总权值是5.0，到*E*的权值最小的路径的总权值是14.0。外部循环的第三次迭代中，从pq里删除<*D*, 5.0>并迭代通过*D*的邻居，即*F*和*E*。本次迭代的结果如图14-26所示。 586

weight_sum	predecessor	pq
A, 0.0	A	<C, 2.0>
B, 4.0	A	<B, 4.0>
C, 2.0	A	<E, 15.0>
D, 10000.0		
E, 15.0	A	
F 10000.0		
H, 10000.0		

图14-23 Dijkstra的最短路径算法的初始化阶段

weight_sum	predecessor	pq
A, 0.0	A	<B, 4.0>
B, 4.0	A	<D, 7.0>
C, 2.0	A	<E, 15.0>
D, 7.0	C	
E, 15.0	A	
F, 10000.0		
H, 10000.0		

图14-24 外部循环的第一次迭代之后，Dijkstra的最短路径算法应用的状态

weight_sum	predecessor	pq
A, 0.0	A	<D, 5.0>
B, 4.0	A	<D, 7.0>
C, 2.0	A	<E, 14.0>
D, 5.0	B	<E, 15.0>
E, 14.0	B	
F, 10000.0		
H, 10000.0		

图14-25 外部循环的第二次迭代之后，Dijkstra的最短路径算法应用的状态

weight_sum	predecessor	pq
A, 0.0	A	<F, 5.0>
B, 4.0	A	<D, 7.0>
C, 2.0	A	<E, 8.0>
D, 5.0	B	<E, 14.0>
E, 8.0	D	<E, 15.0>
F, 5.0	D	
H, 10000.0		

图14-26 外部循环的第三次迭代之后，Dijkstra的最短路径算法应用的状态

外部循环的第四次迭代中，从pq里删除<*F*, 5.0>；检查*F*的邻居，即*D*和*H*；并修改容器。如图14-27所示。

weight_sum	predecessor	pq
A, 0.0	A	<D, 7.0>
B, 4.0	A	<E, 8.0>
C, 2.0	A	<H, 9.0>
D, 5.0	B	<E, 14.0>
E, 8.0	D	<E, 15.0>
F, 5.0	D	
H, 9.0	F	

图14-27 外部循环的第四次迭代之后，Dijkstra的最短路径算法应用的状态

外部循环的第五次迭代开始时，从pq里删除<*D*, 7.0>。目前，weight_sum中从*A*到*D*的最小总权值就记录成5.0。因此在内部循环迭代中，weight_sum、predecessor或是pq都没有改变。

外部循环的第六次迭代中，从pq里删除<*E*, 8.0>。因为我们想寻找的是到*E*的最短路径，所以至此就完成了。如何确认没有到*E*的更短的路径了呢？如果还有另一条到*E*的路径，它的总权值*t*小于8.0，那么对<*E*, *t*>将比对<*E*, 8.0>更早从pq中删除。

通过添加*E*、*E*的前任*D*、*D*的前任*B*和*B*的前任*A*，从predecessor来构造最短路径，即一个顶点列表。那么列表的顺序将是正确无误的:

A, *B*, *D*, *E*

对Dijkstra的算法的描述遗漏了少许细节。例如，顶点、边和邻居将如何存储？14.6节中开发了一个类，该节提供了许多漏掉的细节，这些细节不仅涉及到Dijkstra的算法，而且涉及到所有与图相关的工作。

14.6 开发一个网络类

本章介绍了6个数据结构：无向图，（有向）图，无向树，（有向）树，无向网络和（有向）网络。我们希望为这些结构开发一些类，它们的代码可最大共享。面向对象的解决方案是利用继承，但是精确地说该怎么做呢？如果令directed_graph类是undirected_graph的一个子类，那么实际上，所有和边相关的代码必须被覆盖。这是因为在一个无向图里，每条边(*A*, *B*)代表了两个连接：从*A*到*B*和从*B*到*A*。同样，为图编写的代码用于网络时也必须重新编写。

为了代码的重用，我们定义了一个（有向）network类，并将其他的图类定义为network的子类。

一个更好的方法是定义（有向）network类，并令其他类成为该类的子类。例如，可以把一个undirected_network容器看作是directed_network容器，它的每条边都是双向的。因此将边(*A*, *B*)添加进一个undirected_network容器的方法定义是：

```
//后置条件：如果边<v1,v2>已经在这个无向网络中，那么返回假。否则，将这条
//          边以及给定的weight插入这个无向网络并返回真。
bool insert_edge (const vertex& v1, const vertex v2, double& weight)
{
    if (contains_edge (v1, v2))
        return false;
    network::insert_edge (v1, v2, weight);
    network::insert_edge (v2, v1, weight);
    return true;
//network的子类undirected_network类中的insert_edge方法
```

（在这个方法定义里，必须使用作用域解析运算符——即**operator**::——指明所调用的方法是network的insert_edge方法，而不是undirected_network类的insert_edge方法。）

此外，还可以将一个有向图看作一个网络，其中所有的权值都是1.0。directed_graph类将包含下列方法：

```
//后置条件：如果在本次调用前，<v1,v2>不是这个directed_graph中的边，
//          那就将它加入这个directed_graph并返回真。否则就返回假。
boolean insert_edge(const vertex& v1, const vertex& v2)
{
    network::insert_edge(v1,v2,1.0);
}//network的子类directed_graph类中的insertEdge方法
```

14.7节中开发了一个（有向）network类。我们把子类undirected_graph、directed_graph、undirected_tree、directed_tree和undirected_network的开发留作习题。 588

14.7 network类

开发类的首要问题是确定该类中应当包含的公有方法：这些组成了用户角度的类。对网络类而言，需要有与顶点相关的方法，以及与边相关方法和网络整体的方法。先从与顶点相关的方法开始（14.7.7节中给出了时间估算）：

```
//后置条件：如果这个网络包含v就返回真；否则，返回假。
bool contains_vertex(const vertex& v);
```

```
//后置条件：如果v已经在这个网络中就返回假。否则，将v加入这个网络并返回真。
bool insert_vertex(const vertex& v);

//后置条件：如果v是这个网络里的一个顶点，那么就从网络中删除v以及所有
//          和它相连的边，并返回真。否则，返回假。
bool erase_vertex(const vertex& v);
```

下面是与边相关的方法的接口：

```
//后置条件：返回这个网络中边的数量。
unsigned int get_edge_count();

//后置条件：如果<v1,v2>组成了这个网络中的一条边，那么就返回该边的权值。
//          否则，返回-1。
double get_edge_weight(const vertex& v1, const vertex& v2);

//后置条件：如果这个网络包含了边<v1,v2>就返回真。否则，返回假。
bool contains_edge(const vertex& v1, const vertex& v2);

//后置条件：如果边<v1,v2>已经在这个网络中，就返回假。否则，将它以及
//          给定的权值插入这个网络并返回真。
bool insert_edge(const vertex& v1, const vertex& v2, const double& weight);

//后置条件：如果<v1,v2>是这个网络中的一条边，就删除这条边并返回真。
//          否则，返回假。
bool erase_edge(const vertex& v1, const vertex& v2);
```

589

最后是应用于整体网络的方法的接口，包含了三个辅助迭代器：

```
//后置条件：这个网络为空。
network();

//后置条件：这个网络包含了对other的拷贝。
network(const network& other);

//后置条件：返回这个网络中的顶点数量。
unsigned int size();

//后置条件：如果这个网络中没有顶点就返回真。否则，返回假。
bool empty();

//前置条件：v在这个网络里。
//后置条件：返回v的邻居列表。
list<vertex> get_neighbor_list(const vertex& v);

//后置条件：如果这个网络是连通的就返回真。否则，返回假。
bool is_connected();

//前置条件：这个网络是连通的。
```

```
//后置条件：返回这个网络的最小生成树。
network<vertex> get_minimum_spanning_tree();

//后置条件：返回从v1到v2的最短路径以及它的总权值。
pair<list<vertex>, double> get_shortest_path(const vertex& v1, const vertex& v2);

//后置条件：返回位于这个网络开头的迭代器。
iterator begin();

//后置条件：返回的迭代器可以用在比较关系中，以终止这个网络里的迭代。
iterator end();

//前置条件：顶点v在这个网络里。
//后置条件：返回一个广度优先迭代器（breadth_first_iterator），它通过从v可达的所有顶点。
breadth_first_iterator breadth_first_begin(const vertex& v);

//后置条件：返回breadth_first_iterator，它可以用在比较关系中，以终止这个
//          网络里的迭代。
breadth_first_iterator breadth_first_end();

//前置条件：顶点v在这个网络里。
//后置条件：返回一个深度优先迭代器（depth_first_iterator），它通过从v可达的所有顶点。
depth_first_iterator depth_first_begin(const vertex& v);

//后置条件：返回depth_first_iterator，它可以用在比较关系中，以终止这个
//          网络里的迭代。
depth_first_iterator depth_first_end();
```

590

在get_minimum_spanning_tree的方法头中，返回类型是network，而不是tree。这是因为tree是directed_graph的子类，它当中的每条边的权值是1.0。因此get_minimum_spanning_tree方法返回一个network，其中有一个特别指定的项，称作根，并且：

1) 没有边进入根项。

2) 每个非根项上恰好有一条边进入它。

3) 从根项到每个其他项之间有一条路径。

4) 每条边有一个非负权值。

14.7.1 network类中的字段

照例，要设计一个类，基本决策涉及到字段的选择。在（有向）network类中，将每个顶点v与所有使得<v,w>组成一条边的顶点w相关联。由于这是一个网络，所以如果<v,m>组成了一条边，那么就把每个这样的顶点w以及边<v,w>的权值一起包含进来。

可以将整体组织改述如下：把网络中的每个顶点v和所有下列形式的对相关联：

```
w,weight
```

其中<v,w>组成了给定weight的边。现在仍存在两个问题：将使用什么样的容器保存关联，以及将使用什么样的容器保存和给定顶点相关联的“全部的对”？要回答第二个问题，预先并

不知道将有多少对，并且对的顺序也不重要。我们可能希望迭代通过和一个给定顶点关联的全部对，因此选择了一个list容器来保存对。

最后，需要一个容器将每个顶点v和对<w,weight>的列表容器相关联，其中<v,w>是权值为weight的一条边。这个关联的重要特点是，给定一个顶点v，希望能快速地访问关联list容器。
591 术语“关联”暗示着将把每个顶点“映射”到它的list容器。为了加速平均访问速度，需要使用一个hash_map容器。回忆在第13章中，插入、删除或查找一个hash_map容器的平均时间花费是常数——假定均匀散列设想成立。

不幸的是，hash_map目前仍不是标准模板库的一部分，因此，为了可移植性，将使用map类来替换。回忆在第10章中，插入、删除或查找一个map容器的最坏时间花费和n成对数关系。虽然没有hash_map类的平均时间快，但已经相当好了，并且比hash_map类的最坏时间花费（和n成线性关系）少很多。

network类的头是：

```
template<class vertex, class Compare=less<vertex>>
class network;
```

map类需要第二个模板参数。也就是说，network类的用户提供了一个模板变元，指明如何比较顶点。这些比较决定了顶点-列表对将在map容器（可能是一个红黑树）中储存的位置。

network类中惟一的字段是：

```
protected:
    map_class adjacency_map;
```

其中map_class是通过如下方式定义的：

```
typedef map<vertex,list<vertex_weight_pair>,Compare> map_class;
```

在network类的**邻接表**的设计中，惟一的字段是一个map对象adjacency_map，其中每个键是一个顶点并且每个值都是键的邻居（以及权值）的链表。

回想一下，map类里的每个值都是一个对。在adjacency_map容器中，每个值将由一个顶点-列表对组成，其中顶点是键，而列表包含了和键邻接的每个顶点，以及相应边的权值。这个设计被称作是**邻接表**设计。

如果itr是map_class::iterator类型的，那么*itr返回一个对。明确地说，(*itr).first是一个顶点，并且(*itr).second是该顶点的邻接表。邻接表中的项是vertex_weight_pair类型的，即包含to和weight字段（分别是vertex和double类型）的**结构**。这些字段比“对”**结构**中的first和second字段容易理解，因此读者应该不会混淆vertex_weight_pair项和脱引用映射迭代器返回的对。使用vertex_weight_pair**结构**[⊖]的另一优势是可以重载**operator>**用于优先队列的比较。

下面是vertex_weight_pair的定义：

```
struct vertex_weight_pair
{
592     vertex to;
        double weight;
        //后置条件：通过x和y初始化这个vertex_weight_pair。
```

⊖ 回忆一下，**结构**是一个类，它的所有成员都是**public**的。

```
    vertex_weight_pair(const vertex& x, const double& y)
    {
        to=x;
        weight=y;
    }//两个参数的构造器

    //后置条件: 如果这个vertex_weight_pair小于x就返回真。否则,
    //          返回假。
    bool operator>(const vertex_weight_pair& p) const
    {
        return weight>p.weight;//权值最小的具有最高的优先级。
    }//运算符>
};//vertex_weight_pair类
```

14.7.2 network类的实现

由于类中惟一的字段是一个map，所以几个方法的定义都很简单，因为很多工作都已经在相应的map方法中完成了：

```
network( ) { }

network (const network& other)
{
    adjacency_map = other.adjacency_map;
} // 拷贝构造器

unsigned size( )
{
    return adjacency_map.size( );
} // size方法

bool empty( )
{
    return size( ) == 0;
} // empty方法

bool contains_vertex (const vertex& v)
{
    return adjacency_map.find (v) != adjacency_map.end( );
} // contains_vertex方法
```

network类中这些以及其他方法的时间花费将在14.7.7节中探讨。 593

向network容器插入一个顶点v是很直接的。如果v不在网络中，就使用关联数组运算符将对<v, empty list>插入adjacency_map容器。下面是insert_vertex的定义：

```
bool insert_vertex (const vertex& v)
{
    if (adjacency_map.find (v) != adjacency_map.end( ))
```

```
            return false;
        adjacency_map [v] = list< vertex_weight_pair >( );
        return true;
    } // insert_vertex方法
```

erase_vertex的定义需要费一番工夫。从adjacency_map中删除一个顶点v是很简单的，删除从v出发的边的关联列表也是如此。但是还必须删除每条进入v的边。因为边的信息是以顶点-权值对的形式存储在一个列表里，所以必须迭代通过network容器中的全部顶点。对每个顶点都要迭代通过它的对列表来寻找v。只要发现v在某个对中，就从列表里删除该对。下面是erase_vertex的定义：

```
bool erase_vertex (const vertex& v)
{
    map_class::iterator itr = adjacency_map.find (v);
    if (itr == adjacency_map.end( ))
        return false;
    adjacency_map.erase (itr);
    list<vertex_weight_pair>::iterator list_itr;
    for (itr = adjacency_map.begin( ); itr != adjacency_map.end( ); itr++)
        // 在(*itr).second的列表中，删除所有的<v, ?>对
        for (list_itr = (*itr).second.begin( );list_itr != (*itr).second.end( );
                list_itr++)
            if ((*list_itr).to == v)
            {
                (*itr).second.erase (list_itr);
                break; // 退出内部for循环
            } // 在(*itr).second列表中找到的v
    return true;
} // erase_vertex
```

14.7.3 与边相关的方法的实现

现在继续考虑与边相关的方法。为了计算network容器中边的数量，可以利用任何顶点的关联list容器的大小代表了从该顶点出发的边的数量这一事实。因此，可以迭代通过网络中的
594 全部顶点，并累计关联list容器的大小。定义如下：

```
unsigned get_edge_count( )
{
    int count = 0;

    map_class::iterator itr;

    for (itr = adjacency_map.begin( ); itr != adjacency_map.end( ); itr++)
        count += (*itr).second.size( );
    return count;
} // get_edge_count方法
```

边的权值计算是相似的。要求<v1, v2>的权值，可迭代通过v1的关联列表，寻找顶点是v2的对。如果找到，就返回该对的权值；否则就返回-1.0，说明<v1, v2>不是网络中的边。定义

如下:

```
double get_edge_weight (const vertex& v1, const vertex& v2)
{
    map_class::iterator itr = adjacency_map.find (v1);
    if (itr == adjacency_map.end( )
            || adjacency_map.find (v2) == adjacency_map.end( ))
        return -1.0;
    // 迭代通过v1的邻居列表:
    list<vertex_weight_pair >::iterator list_itr;
    for (list_itr = ((*itr).second).begin( ); list_itr != ((*itr).second).end( );
            list_itr++)
        if ((*list_itr).to == v2)
            return (*list_itr).weight; // 返回<v1,v2>的权值
    return -1.0; // 网络中没有<v1,v2>边
} // get_edge_weight
```

相似的迭代可以用来判断网络是否包含一条给定的边:

```
bool contains_edge (const vertex& v1, const vertex& v2)
{
    map_class::iterator itr = adjacency_map.find (v1);
    if (itr == adjacency_map.end( )
            || adjacency_map.find (v2) == adjacency_map.end( ))
        return false;
    // 考察v2是否在v1的邻接顶点列表里:
    list<vertex_weight_pair >::iterator list_itr;
    for (list_itr = ((*itr).second).begin( ); list_itr != ((*itr).second).end( );
            list_itr++)
        if ((*list_itr).to == v2)
            return true;
    return false;
} // contains_edge方法
```

595

删除一条边的代码也是相似的:

```
bool erase_edge (const vertex& v1, const vertex& v2)
{
    map_class::iterator itr = adjacency_map.find (v1);
    if (itr == adjacency_map.end( )
            || adjacency_map.find (v2) == adjacency_map.end( ))
        return false;
    // 如果<v1, v2>组成了一条边，就从v1的邻接边列表里
    // 删除边<v1, v2>以及该边的权值:
    list<vertex_weight_pair >::iterator list_itr;
    for (list_itr = (*itr).second.begin( ); list_itr != (*itr).second.end( );
            list_itr++)
        if ((*list_itr).to == v2)
        {
            (*itr).second.erase (list_itr);
```

```
                return true;
            } // if
        return false;
    } // erase_edge方法
```

添加边<v1, v2>需要将vertex_weight_pair<v2,weight>加入v2的关联邻接表中:

```
bool insert_edge (const vertex& v1, const vertex& v2, double weight)
{
    if (contains_edge (v1, v2))
        return false;
    insert_vertex (v1);
    insert_vertex (v2);
    (*(adjacency_map.find(v1))).second.push_back (vertex_weight_pair
            (v2, weight));
     return true;
} // insert_edge方法
```

14.7.4 全局方法的实现

最后讨论应用于整体网络的方法。迭代通过一个顶点关联的邻接表可以得到该顶点的邻居列表。邻接表中每个vertex_weight_pair项里的to顶点被添加进一个初始为空的列表。定义如下:

```
list<vertex > get_neighbor_list (const vertex& v)
{
596     list<vertex_weight_pair>::iterator list_itr;
    list<vertex> vertex_list;
    for (list_itr = adjacency_map [v].begin( ); list_itr !=
            adjacency_map[v].end( ); list_itr++)
        vertex_list.push_back (list_itr -> to);
    return vertex_list;
} // get_neighbor_list方法
```

要判断网络是否连通，可以迭代通过网络中的所有顶点。对每个顶点v，使用广度优先迭代器迭代通过从v可达的顶点。对任何顶点v，如果从v可达的顶点数量小于网络中的顶点数量，那么网络不是连通的；否则，网络就是连通的。下面是它的定义:

```
bool is_connected( )
{
    map_class::iterator itr;
    // 对每个顶点v，检查从v可达的顶点数量是否等于
    // 这个网络中顶点的总数量。
    for (itr = adjacency_map.begin( ); itr != adjacency_map.end( );itr++)
    {
        vertex v = (*itr).first;
```

```
            // 计数从v可达的顶点数量。
            unsigned count = 0;
            breadth_first_iterator b_itr;
            for (b_itr = breadth_first_begin (v); b_itr != breadth_first_end( );
                        b_itr++)
                count++;
            if (count < adjacency_map.size( ))
                return false;
        } // 迭代通过网络中的全部顶点。
        return true;
    } // is_connected方法
```

为了从一个start顶点开始执行网络的广度优先迭代，需要了解的不仅仅是start顶点，而且还有网络。邻接映射将为第二个参数服务，但我们并不需要该映射的单独的拷贝。breadth_first_begin方法把start顶点和adjacency_map的地址发送给breadth_first_iterator类中的构造器：

```
// 前置条件: 顶点v在这个网络里。
// 后置条件: 返回一个breadth_first_iterator，它通过
//           所有从v可达的顶点。
breadth_first_iterator breadth_first_begin (const vertex& v)
{
    breadth_first_iterator b_itr (v, &adjacency_map);
    return b_itr;
} // breadth_first_begin方法
```

597

breadth_first_iterator类遵循了14.5.1节中给出的框架。为了能快速判断一个给定顶点是否可达，令reached为<vertex, **bool**>形式的对的映射（指向映射的指针）。这是另一种对！头和字段是：

```
class breadth_first_iterator
{
    friend class network;
    protected:
        queue<vertex>* vertex_queue;
        map<vertex, bool, Compare>* reached;
        map_class* map_ptr;
```

通过将这些字段定义成指针，可以很大程度地简化广度优先迭代器的相等性测试：比较指针，而不需迭代通过容器。breadth_first_iterator的两个参数的构造器的参数是一个start顶点和一个映射指针。映射指针参数被赋给map_ptr字段，然后初始化reached和vertex_queue字段。对每个顶点v，将对<v,false>插入*reached。然后把对<start,true>插入*reached。再将start推入*vertex_queue。代码如下：

```
// 后置条件: 在start顶点上初始化这个breadth_first_iterator。
breadth_first_iterator (const vertex& start, map_class* ptr)
```

```
{
    map_ptr = ptr;
    reached = new map<vertex, bool, Compare>( );
    vertex_queue = new queue <vertex>( );
    // 将每个顶点标记为不可达的:
    map_class::iterator itr;
    for (itr = (*map_ptr).begin( ); itr != (*map_ptr).end( ); itr++)
        (*reached)[(*itr).first] = false;
    (*reached)[start] = true;
    (*vertex_queue).push (start);
} // 两个参数的构造器
```

后加运算符**operator**++从*vertex_queue队列的开头取出顶点，将和该顶点邻接的所有尚未到达的顶点插入队列，并且当*vertex_queue为空时清空字段（和end()方法比较）。定义如下：

598

```
//前置条件: 这个breadth_first_iterator尚未到达这个网络的所有可达顶点。
//后置条件: 这个breadth_first_iterator前进到这个网络的下一个可达顶点;
//          返回前进之前的迭代器。
breadth_first_iterator operator++ (int)
{
    breadth_first_iterator temp = *this;
    vertex current = (*vertex_queue).front( );
    (*vertex_queue).pop( );
    map_class::iterator itr = (*map_ptr).find (current);
    list<vertex_weight_pair>::iterator list_itr;
    // 迭代通过current的邻居:
    for (list_itr = (*itr).second.begin( ); list_itr != (*itr).second.end( );
            list_itr++)
    {
        vertex to = (*list_itr).to;
        // 到达过顶点to吗?
        if ((*reached) [to] == false)
        {
            (*reached) [to] = true;
            (*vertex_queue).push (to);
        } // if
    } // for
    if ((*vertex_queue).empty( ))
    {
        vertex_queue = NULL;
        reached = NULL;
        map_ptr = NULL;
    } // 如果队列为空
    return temp;
} // 运算符++
```

operator==测试字段是否相等，而脱引用运算符**operator***返回*vertex_queue开头的顶点。depth_first_iterator类和breadth_first_iterator类之间惟一的重要差别是使用vertex_stack替代了vertex_queue。

14.7.5 get_minimum_spanning_tree方法

get_minimum_spanning_tree方法的定义遵循了14.5.3节中给出的框架。从某些作为根的顶点开始，并获取和该根关联的vertex_weight对列表。将每个这样的边三元组<root, vertex, weight>推入一个优先队列。然后从优先队列中弹出权值最小的三元组<x, y, weight>，直到生成树的大小等于网络的大小。如果y不在生成树中，就把y和边<x, y>以及权值加入生成树，并迭代通过y的邻居；如果邻居z不在生成树里，那么就把边三元组<y, z, <y, z>边的权值>推入优先队列。 599

为了更方便，创建了edge_triple**结构**，这使得我们可以很容易地访问一条边以及它的权值，并定义了三元组优先队列需要的**operator>**。get_minimum_spanning_tree的定义是：

```
network<vertex> get_minimum_spanning_tree( ) {
    network min_spanning_tree; // 最小生成树是一个网络，这样可以
                               // 在其中包含权值。
    priority_queue<edge_triple, vector<edge_triple>,
            greater<edge_triple> > pq;
    vertex root,
           x,
           y,
           z;
    iterator itr;
    list< vertex_weight_pair > adjacency_list;
    list< vertex_weight_pair >::iterator list_itr;
    double weight;
    if (empty( ))
        return min_spanning_tree;
    itr = begin( );
    root = (*itr).first;
    min_spanning_tree.insert_vertex (root);
    adjacency_list = adjacency_map [root];
    for (list_itr = adjacency_list.begin( ); list_itr != adjacency_list.end( );
            list_itr++)
        pq.push (edge_triple (root, list_itr -> to, list_itr -> weight));
    while (min_spanning_tree.size( ) < size( ))
    {
        x = pq.top( ).from;
        y = pq.top( ).to;
        weight = pq.top( ).weight;
        pq.pop( );
        if (!min_spanning_tree.contains_vertex (y))
```

```
            {
                min_spanning_tree.insert_vertex (y);
                min_spanning_tree.insert_edge (x, y, weight);
600             adjacency_list = adjacency_map [y];
                for (list_itr = adjacency_list.begin( );
                        list_itr != adjacency_list.end( );
                        list_itr++)
                {
                    z = list_itr -> to;
                    if (!min_spanning_tree.contains_vertex (z))
                    {
                        weight = list_itr -> weight;
                        pq.push (edge_triple (y, z, weight));
                    } // z还不在树中
                } // 迭代通过y的邻居
            } // y还不在树中
        } // 树的顶点数少于这个网络的顶点数量
        return min_spanning_tree;
    } // get_minimum_spanningTree方法
```

14.7.6 get_shortest_path方法

最后来定义get_shortest_path方法，它返回从顶点v1到顶点v2的总权值最小的路径。正如14.5.4节给出的框架一样，Dijkstra的算法是从v2开始的广度优先迭代。priority_queue容器pq由<w, total_weight>形式的对组成，其中total_weight是迄今得到的从v1到w的最短路径上所有边的权值总和。优先队列按照最小总权值排序。为了记录从v1出发的所有局部路径的总权值，使用了一个映射weight_sum，它将每个顶点w和迄今得到的从v1到w的最短路径上所有边的权值总和相关联。为了能在通过时重新构造最短路径，还使用了另一个map容器predecessor，它将每个顶点w和迄今得到的从v1到w的最短路径上w的直接前任相关联。

最初，list_itr迭代通过所有vertex_weight对<w, wweight>的邻接表，其中<v1, w>是权值为wweight的边。对每个对*list_itr:

```
weight_sum[list_itr->to]=list_itr->weight;
predecessor[list_itr->to]=v1;
pq.insert(*list_itr);
```

其他所有顶点的初始权值为MAX_PATH_WEIGHT。

然后从pq中再三弹出最小总权值的顶点from。接着list_itr迭代通过所有<to, weight>对的邻接表，其中<from, to>是权值为weight的边。每个对<to, weight>:

```
if (weight_sum [from] + weight < weight_sum [to])
{
601     weight_sum [to] = weight_sum [from] + weight;
    predecessor [to] = from;
    pq.push (vertex_weight_pair (to, weight_sum [to]));
}
```

从pq中弹出v2时，将前任顶点添加到一个链表中并和最短路径的总权值一起返回。定义如下：

```
pair<list<vertex>, double> get_shortest_path (const vertex& v1,
                                              const vertex& v2)
{
    const double MAX_PATH_WEIGHT = 1000000.0;

    map<vertex, vertex, Compare> predecessor;
    map<vertex, double, Compare> weight_sum;
    priority_queue<vertex_weight_pair,
                   vector<vertex_weight_pair>,
                   greater<vertex_weight_pair> > pq;

    list<vertex_weight_pair >::iterator list_itr;

    breadth_first_iterator b_itr;

    vertex to,
           from;

    double weight;

    if (adjacency_map.find (v1) == adjacency_map.end( ) ||
                adjacency_map.find (v2) == adjacency_map.end( ))
        return pair<list<vertex>, double> (list<vertex>( ), -1.0);

    bool found_v2 = false;
    for (b_itr = breadth_first_begin (v1); b_itr != breadth_first_end( );
                b_itr++)
        if (*b_itr == v2)
        {
            found_v2 = true;
            break;
        } // if
    if (!found_v2)
        return pair<list<vertex>, double> (list<vertex>( ), -1.0);

    weight_sum [v1] = 0.0;
    predecessor [v1] = v1;
    for (b_itr = breadth_first_begin (v1); b_itr != breadth_first_end( );
                b_itr++)
    {
        weight_sum [*b_itr] = MAX_PATH_WEIGHT;
        predecessor [*b_itr] = vertex( );
    } // 初始化weight_sum以及前任
    for (list_itr = adjacency_map [v1].begin( ); list_itr !=
                adjacency_map [v1].end( ); list_itr++)
    {
        weight_sum [list_itr -> to] = list_itr -> weight;
        predecessor [list_itr -> to] = v1;
        pq.push (vertex_weight_pair (*list_itr));
```

602

```
    } // 调整v1邻接顶点对应的weight_sum、predecessor、pq
    bool path_found = false;
    while (!path_found)
    {
        from = pq.top( ).to; // 获取vertex_weight_pair中权值
                             // 总和最小的顶点
        pq.pop( );
        if (from == v2)
            path_found = true;
        else
        {
            for (list_itr = adjacency_map [from].begin( );
                 list_itr != adjacency_map [from].end( ); list_itr++)
            {
                to = list_itr -> to;
                weight = list_itr -> weight:
                if (weight_sum [from] + weight < weight_sum [to])
                {
                    weight_sum [to] = weight_sum [from] + weight;
                    predecessor [to] = from;
                    pq.push (vertex_weight_pair (to,
                                                 weight_sum [to]));
                } // 如果from_weight_sum+weight>to_weight_sum
            } // 迭代通过from的列表
        } // 否则就是没有找到路径
    } // 当没有找到路径时
    list<vertex> path;
    vertex current = v2;
    while (current != v1)
    {
        path.push_front (current);
        current = predecessor [current];
    } // 当尚未返回v1时
    path.push_front (v1);
    return pair<list<vertex>, double> (path, weight_sum [v2]);
603 } // get_shortest_path方法
```

14.7.7 网络方法的时间花费估算

令V是网络中的顶点数量，E是边的数量。为了简单起见，这里只考虑平均时间花费。基本上，如果一个方法需要迭代通过所有顶点而不通过关联列表，那么averageTime(V, E)和V成线性关系。这是因为adjacency_map和顶点键一起存储在一个红黑树里，并且迭代通过顶点需要V的线性时间。迭代通过单个顶点的邻接表的averageTime(V, E)是$O(\log V+E/V)$，因为在红黑树里访问一个顶点（即一个键）花费$\log V$时间，而V个列表中共有E条边。

get_edge_count方法迭代通过所有顶点，因此averageTime(V, E)和V成线性关系。另一方

面，get_edge_weight方法迭代通过它的第一个顶点参数所关联的列表，因此averageTime(*V*, *E*)是$O(\log V+E/V)$，并且这也正是最小上界。

erase_vertex方法首先从网络中查找并删除它的变元，因此averageTime(*V*,*E*)和*V*成对数关系。然后方法迭代通过所有顶点，并且，对每个顶点，又迭代通过与该顶点关联的列表，因此averageTime(*V*, *E*)和$V*E/V$（等于*E*）成线性关系。所以erase_vertex的averageTime(*V*,*E*)是$O(\log V+E)$，并且这就是最小的。

在广度优先或深度优先迭代的分析中，首先简单假设网络是连通的。然后广度优先迭代通过所有的*V*顶点，而且对每个顶点，都迭代通过它的关联列表。因此averageTime(*V*,*E*)和*E*成线性关系。深度优先迭代同上。is_connnected方法迭代通过所有顶点并为每个顶点执行一个广度优先迭代，因此averageTime(*V*, *E*)和*VE*成线性关系。

在get_minimum_spanning_tree方法中，外部循环的迭代次数和*V*成线性关系。每次外部循环迭代中，优先队列弹出（需要log*E*次迭代），并由内部循环迭代通过邻居列表（需要*E*/*V*次迭代），因此迭代的总次数是$O(V(\log E+E/V))$。那么averageTime(*V*, *E*)就是$O(V\log E+E)$。这可能不是最小上界，因为只是建立了一个优先队列弹出所需要的迭代次数的上界。

为了简化get_shortest_path方法的分析，假设从v1到v2间的路径距离至少是*V*/2。那么对路径中的每个顶点，内部循环将迭代通过它的邻居。因此，和getMinimumSpanningTree方法一样，它的averageTime(*V*, *E*)也是$O(V\log E+E)$。

完整的network类以及驱动器程序可以参阅本书网站的源代码链接。实验29介绍了一个最著名的网络问题，进一步探索了贪心算法，并涉及到一些难题讨论。

实验29：货郎担问题　　（所有实验都是可选的）

604

14.7.8 network类的另一种设计和实现

在14.7.1节的network类的设计中，顶点是作为键存储在一个map容器里的。每个值的另一个组成部分是顶点-权值对的**邻接表**，也就是顶点键的邻居（以及边的权值）链表。这个结构很适合稀疏网络，也就是边的数量不大于顶点数量的网络。在任何连通网络里，一定有：

$$V \leq E \leq V(V+1)/2$$

*V*个邻接表里有*E*条边，因此邻接表的平均大小是*E*/*V*。

contains_edge方法需要花费log*V*的时间访问相应的邻接表，以及需要*E*/*V*的时间查找该列表。因此contains_edge方法的averageTime(*V*, *E*)是$O(\log V+E/V)$。如果已知*E*和*V*的关系，就能够求精这个估算。特别是，如果边的数量和*V*成线性关系，那么每个邻接表的平均大小将是常数，并且可以快速地迭代通过邻接表，以判断网络中是否包含一条给定边。那么contains_edge方法的averageTime(*V*, *E*)和*V*成对数关系。

另一方面，如果边的数量和*V*成平方关系，那么邻接表的平均大小将和*V*成线性关系。这暗示着由于每次迭代都需要顺序地穿过一个邻接表，所以averageTime(*V*, *E*)和*V*成线性关系。那么contains_edge方法的averageTime(*V*, *E*)就和*V*成线性关系。

network类的**邻接矩阵**设计中，使用了一个*V*行*V*列的矩阵保存每条边的权值。

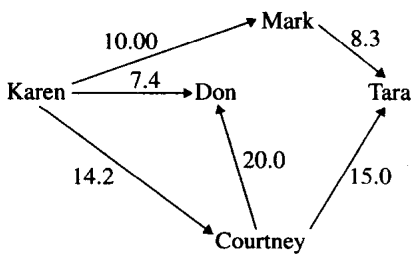
另一种常用的网络表示法是使用邻接矩阵而非邻接表。**邻接矩阵**是**double**项组成*V*行*V*列

的二维向量（或数组）。adjacency_matrix[*i*][*j*]保存了从第*i*个顶点到第*j*个顶点的边的权值。我们需要能快速地将顶点关联到一个数组下标，并且还能快速地将下标关联到它的顶点。为了这最后的任务，创建一个顶点向量或数组：vertexs[*i*]包含了与adjacency_matrix[*i*]（矩阵中的第*i*行）所对应的顶点。

为顶点和下标间的关联创建vertex_map——一个map容器，其中每个键是一个顶点，而每个值中的另一个组成部分是与顶点键对应的索引。图14-28显示了一个网络以及相应的邻接矩阵表示形式。

图14-28假设了顶点是按照如下顺序加入网络的："Karen"，"Mark"，"Don"，"Courtney"，"Tara"。每当一个顶点加入网络时，该顶点就被存储在vertices向量的下一个未占用索引上，并且将该顶点-索引对加入vertex_map容器。下面是这个设计中的四个字段：

```
vector<vector<double>> adjacency_matrix;   //保存每条边的权值
605 vector<vertex> vertices;                //保存顶点
map<vertex, int> vertex_map;                //将每个顶点和它的索引关联
int next;                                   //下一个顶点在vertices中的索引
```



	0	1	2	3	4
0	−1.0	**10.0**	**7.4**	**14.2**	−1.0
1	−1.0	−1.0	−1.0	−1.0	**8.3**
2	−1.0	−1.0	−1.0	−1.0	−1.0
3	−1.0	−1.0	**20.0**	−1.0	**15.0**
4	−1.0	−1.0	−1.0	−1.0	−1.0

adjacency_matrix

0	Karen
1	Mark
2	Don
3	Courtney
4	Tara

vertices

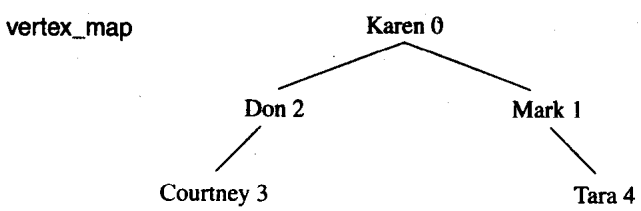


图14-28 网络的邻接矩阵表示。条目是−1.0说明没有边。为了增强可视性，边的权值都用黑体表示

在缺省构造器中，定义了顶点初始数量的缺省值。然后adjacency_matrix字段全部被初始化成−1.0；vertices字段被分别初始化成一个空向量；next被初始化成0。

如果vertex已经在网络里，insert_vertex方法就返回**false**。否则，调整这四个字段并返回**true**。averageTime(V, E)和V成对数关系（除非发生扩充），因为必须查询vertex_map才能求出给定顶点对应的索引。定义如下：

```
//后置条件: 如果顶点已经在这个网络里或者网络已满，那么返回假。否则，                    606
//          将顶点加入这个网络并返回真。
bool insert_vertex (vertex v)
{
    if (vertex_map.find (v) != vertex_map.end( ) || next ==
            vertices.size( ) - 1)
        return false;
    check_for_expansion( );
    vertices [next] = v;
    for (unsigned i = 0; i < vertices.size( ); i++)
        adjacency_matrix[next][i] = -1.0;
    vertex_map.insert (pair <vertex, int> (v, next));
    next++;
    return true;
} // insertVertex方法
```

network类的这个实现的剩余部分将在编程项目14.1中探讨。

本章最后的论题是回溯。第4章中已经了解了如何使用回溯来解决多种应用，现在我们将应用领域扩展到网络，当然还有图和树。

14.8 回溯通过网络

在第4章提到回溯时，了解了三个应用，其中的基本结构都没有改变。特别是使用了相同的BackTrack类和Application头在以下应用中：

1) 搜索一个迷宫。

2) 在棋盘上放置八个皇后——使得每一个都不会被其他皇后攻击到。

3) 阐述了一个马可以遍历棋盘中的每个格而且不会重复到达任意格。

第4章介绍的回溯结构也可以用于回溯通过网络。

网络（或图、树）也适合回溯。例如，假设有一个城市网络，每条边的权值代表两个城市之间的距离，以英里为单位。给定一个出发城市和一个终点城市，寻找一条路径，其中每条边的距离都小于前一条边的距离。图14-29中给出了一些样本数据，先给出出发点和终点城市，随后给出了每条边。图14-30描述了根据图14-29的数据产生的网络。

对这个问题，一种方案是下面的路径：

$$\text{Boston} \xrightarrow{214} \text{NewYork} \xrightarrow{168} \text{Harrisburg} \xrightarrow{123} \text{Washington}$$

更短的（也就是总权值更小的）路径是：

607

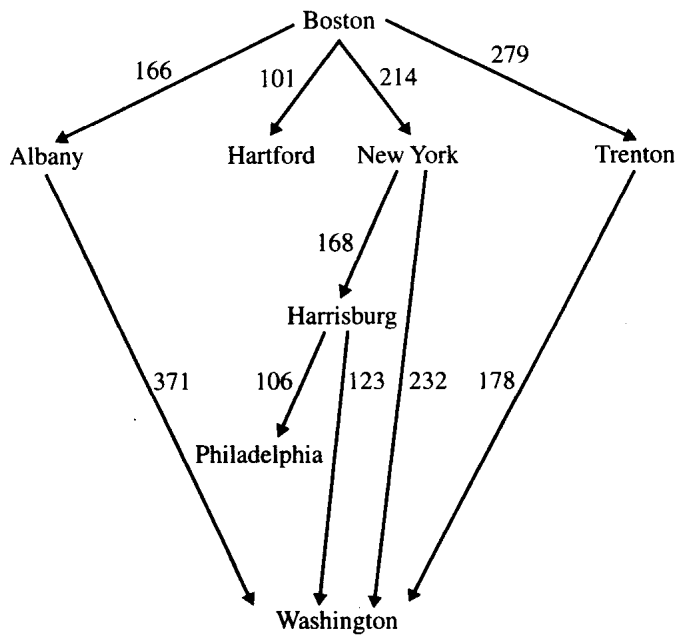
$$\text{Boston} \xrightarrow{279} \text{Trenton} \xrightarrow{178} \text{Washington}$$

最短路径在这个问题上是不合法的，因为它的距离是递增的（从214到232）：

$$\text{Boston} \xrightarrow{214} \text{NewYork} \xrightarrow{232} \text{Washington}$$

Boston	Washington	
Albany	Washington	371
Boston	Albany	166
Boston	Hartford	101
Boston	New York	214
Boston	Trenton	279
Harrisburg	Philadelphia	106
Harrisburg	Washington	123
New York	Harrisburg	168
New York	Washington	232
Trenton	Washington	178

图14-29 一个网络：第一行包含了出发和终点城市；其他每一行包含了两个城市以及从第一个城市到第二个城市间的距离

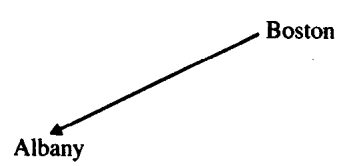


608

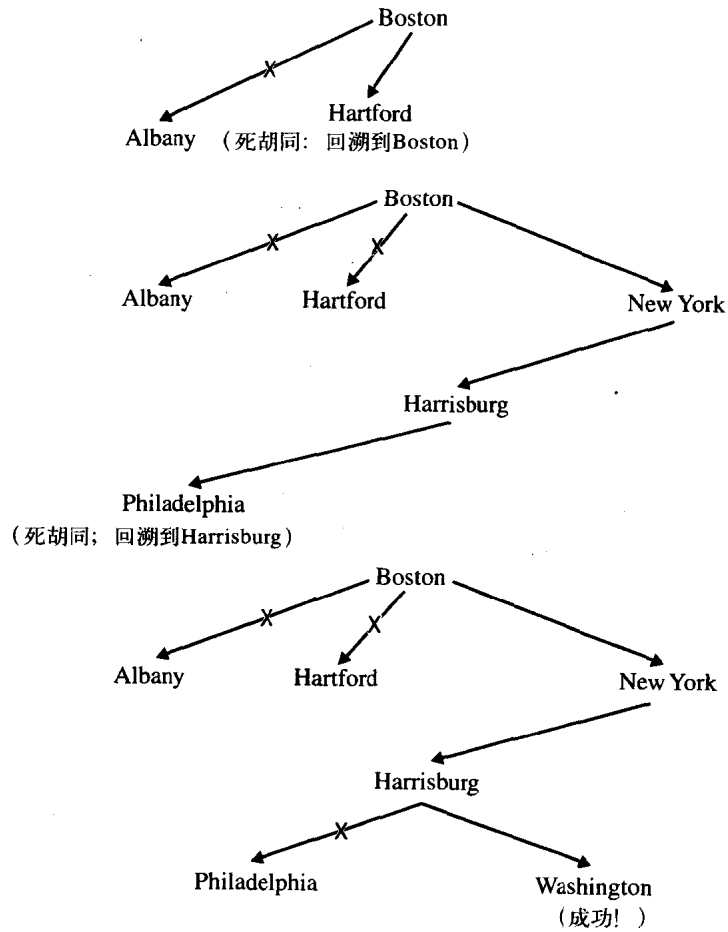
图14-30 一个城市网络：每条边的权值代表这条边上的城市间的距离

当到达一个死胡同时，可以回溯通过网络。回溯的基本策略是从出发点开始利用深度优先查找。在每个位置，迭代通过该位置的邻居。访问的邻居位置的次序就是相应边最初插入网络中的顺序。因此可以保证，只要存在一条路径方案，就一定可以找到，但是它不一定是最短的路径方案。

下面是回溯产生答案的步骤序列：



（死胡同；回溯到Boston）



609

第4章中介绍的框架提供了BackTrack类以及Application头。Position类又怎样呢？这个类必须进行修改：“行”和“列”这样的术语在网络里是没有意义的。细节在编程项目14.2中讨论。

总结

无向图由称作**顶点**的项和不同的称作**边**的无序顶点对组成。如果对是有序的，那么就得到一个**有向图**。**树**，有时称作**有向树**，是一个有向图，它或者为空或者包含一项，称作**根项**，使得：

1) 没有边进入根项。

2) 每个非根项上恰好有一条边进入它。

3) 从根项到每个其他项之间有一条路径。

网络（或无向网络）是一个有向图（或无向图），其中每条边有一个关联非负整数，这些非负整数称作这条边的**权值**。网络也被称作是**带权图**。

一些重要的图和网络算法有：

1) 从一个给定顶点可达的所有顶点的广度优先迭代。

2) 从一个给定顶点可达的所有顶点的深度优先迭代。

3) 判断一个给定图是否连通，也就是对任意两个顶点，从第一个顶点到第二个顶点间是否有一条路径。

4) 寻找网络的最小生成树。

5) 寻找网络中两个顶点间的最短路径。

network类的一种可行的设计和实现是在map容器里将每个顶点和它的邻居相关联。具体地说，adjacency_map是一个map容器，其中每个键是一个顶点v，它被映射到顶点-权值对<w, weight>的链表，其中weight是边<v, w>的权值。

另一种设计和实现是将权值存储在V行V列的二维数组里，其中V是顶点数量。比如，连接顶点i和j的边的权值被存储在adjacency_matrix[i][j]中。

一些网络问题可以通过回溯解决。围绕一个给定位置的迭代就是迭代通过顶点-权值对链表（由给定顶点的相邻边组成）。

习题

14.1 a. 画出下面的无向图：

顶点：A，B，C，D，E

610 边：(A,B)，(C,D)，(D,A)，(B,D)，(B,E)

14.2 a. 画出包含四个顶点以及尽可能多的边的无向图。图中将包含多少条边？

b. 画出包含五个顶点以及尽可能多的边的无向图。图中将包含多少条边？

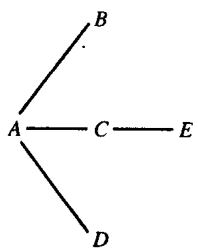
c. V个顶点的无向图中边的最大数量是多少？

d. 证明(c)小题中得出的结论。

提示 对V进行数学归纳。

e. V个顶点的有向图中边的最大数量是多少？

14.3 假设有下面的无向图：

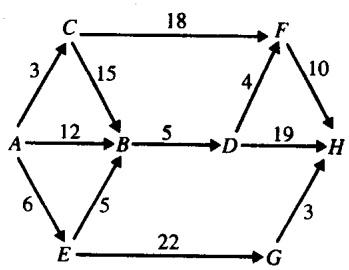


假设顶点是按照字母顺序插入图中的。

a. 从A开始执行无向图的广度优先迭代。

b. 从A开始执行无向图的深度优先迭代。

14.4 对给定网络，使用最笨的方法求出从A到H的最短路径；也就是说，列出所有的路径，然后检测哪一条的权值最小。



14.5 对习题14.4给出的网络，使用Dijkstra的算法（在get_shortest_path方法中）寻找从*A*到*H*的最短路径。

14.6 Prim的算法（get_minimum_spanning_tree）和Dijkstra的算法（get_shortest_path）是**贪心的**：局部最优选择的优先级最高。在这些情况下，如果局部最优选择产生全局最优选择，那么贪心成功。是否所有的贪心算法在所有输入下都是成功的呢？本题中将探讨一个换硬币算法。在一种情况下，贪心算法对所有输入都是成功的。在另一种情况下，贪心算法对某些输入是成功的，而对其他一些输入则是失败的。 611

假设想使用尽可能少的硬币兑换金额少于一美圆的零钱。因为越少越好，所以在每一步上的贪心（即局部最优）选择都是增加后不会超过原始金额的数值最大的硬币。下面是贪心算法：

```
//前置条件: 0<=amount<=99
//后置条件: 输出amount兑换的零钱，要求硬币尽可能少。
void print_fewest (int amount)
{
    int coin[ ] = {25, 10, 5, 1};

    const string RESULT =
        "With as few coins as possible, here is the change for ";

    cout << RESULT << amount << ":";
    for (int i = 0; i< 4; i++>
        while (coin [i] <= amount)
        {
            cout << coin [i] << endl;
            amount -= coin [i];
        } // while
} // print_fewest
```

例如，假设amount的值是62。那么输出将是

```
25
25
10
1
1
```

5就是将62分转换成25分、5分、一角和一分所需要的最少硬币数。

a. 证明这个算法对任何0到99分（包括99）之间的金额都是最优的。

b. 举例说明，如果不能使用5分，那么贪心算法就并非对所有输入都是最优的。也就是，如果有

```
int coins[]={25,10,1};
...
for(int i=0;i<3;i++)
...
```

612

那么算法对某些输入而言将不是最优的。

14.7 忽略习题14.4的图中箭头的方向，那么该图就描述了一个无向网络。使用Prim的算

法寻找这个无向网络的最小生成树。

14.8 忽略习题14.4的图中箭头的方向，并假设图中所有的权值都是1.0。使用Dijkstra的算法寻找从A到H的最短路径。

14.9 修改Dijkstra的算法，寻找一个给定顶点v到网络中其他所有顶点的最短路径。

14.10 从network类的邻接表设计着手，定义undirected_network类中的erase_edge方法。

14.11 如果使用邻接矩阵设计了undirected_network类，那么该矩阵将具有什么有趣的特点？

14.12 给出方法的具体示例，说明当E和V成线性关系时，network的邻接表设计要快于邻接矩阵设计。

14.13 给出方法的具体示例，说明当E和V成平方关系时，network的邻接表设计要慢于邻接矩阵设计。

14.14 如果V很大，比如大于10 000，而且E和V成线性关系，那么network类的哪种设计更好？

14.15 network类的邻接矩阵设计的主要缺陷是什么？

14.16 重新安排图14-29中的边，使得回溯产生的路径与提供的路径不同。

14.17 在网络类的邻接表设计中不需要定义一个析构器。为什么？邻接矩阵设计是否需要
613 定义一个析构器？解释原因。

编程项目14.1：完成邻接矩阵的实现

完成使用邻接矩阵设计的network类的实现。下面是缺省构造器的定义：

```
network( )
{
    const unsigned v = 100;
    vertices.resize (v);
    adjacency_matrix.resize (v);
    for (unsigned i = 0; i < v; i++)
    {
        adjacency_matrix[i].resize(v);
        for (unsigned j = 0; j < v; j++)
            adjacency_matrix[i][j] = -1.0;
    }//初始化行i
    next = 0;
}//缺省构造器
```

根据你的实现和习题14.12到14.15，对network类的邻接表设计和adjacency_matrix设计作
614 出全面的比较。

编程项目14.2：回溯通过一个网络

给出一个网络，其中每个顶点是一个城市，每条边代表两个城市间的距离，求出从一个出发点到终点城市的路径，其中它的每条边的距离都小于前一条边的距离。

分析

每个城市都将以至多14个字符的字符串表示，且没有嵌入的空格。输入的第一行将包含出发城市和终点城市。后面的每一行到终止符号“***”前，将由两个城市和这两个中第一个城市到第二个城市的距离（以英里为单位）组成。

不需要输入编辑工作。最早的输出将是网络。如果没有答案，那么最后的输出将是

There is no solution.

否则，最后的输出将是

There is solution:

后面跟着答案中的相应边（出发城市，到达城市，距离）。

系统测试1（输入用黑体表示）

In the Input line, please enter the start and finish cities, separated by a blank. Each city name should have no blanks and be at most 14 characters in length.
Boston Washington

In the Input line, please enter two cities and their distance; the sentinel is ***
Boston NewYork 214

In the Input line, please enter two cities and their distance; the sentinel is ***
Boston Trenton 279

In the Input line, please enter two cities and their distance; the sentinel is ***
Harrisburg Washington 123

In the Input line, please enter two cities and their distance; the sentinel is ***
NewYork Harrisburg 168

In the Input line, please enter two cities and their distance; the sentinel is ***
NewYork Washington 232

In the Input line, please enter two cities and their distance; the sentinel is ***
Trenton Washington 178

In the Input line, please enter two cities and their distance; the sentinel is ***
***

615

The initial state is as follows:
Trenton Washington 178.0
NewYork Harrisburg 168.0
NewYork Washington 232.0
Harrisburg Washington 123.0
Boston NewYork 214.0
Boston Trenton 279.0

A solution has been found:

FROM CITY	TO CITY	DISTANCE
Boston	NewYork	214.0
NewYork	Harrisburg	168.0
Harrisburg	Washington	123.0

Please close this window when you are ready.

系统测试2（输入用黑体表示）

In the Input line, please enter the start and finish cities, separated by a blank. Each city name should have no blanks and be at most 14 characters in length.
Boston Washington

In the Input line, please enter two cities and their distance; the sentinel is ***
Boston Trenton 279

In the Input line, please enter two cities and their distance; the sentinel is ***
Boston NewYork 214

In the Input line, please enter two cities and their distance; the sentinel is ***
616 **Harrisburg Washington 123**

In the Input line, please enter two cities and their distance; the sentinel is ***
NewYork Harrisburg 168

In the Input line, please enter two cities and their distance; the sentinel is ***
NewYork Washington 232

In the Input line, please enter two cities and their distance; the sentinel is ***
Trenton Washington 178

In the Input line, please enter two cities and a weight; the sentinel is ***
***
The initial state is as follows:
Trenton Washington 178.0
NewYork Harrisburg 168.0
NewYork Washington 232.0
Harrisburg Washington 123.0
Boston Trenton 279.0
Boston NewYork 214.0

A solution has been found:

FROM CITY	TO CITY	DISTANCE
Boston	Trenton	279.0
Trenton	Washington	178.0

Please close this window when you are ready.

注意 系统测试2的答案和系统测试1的答案不同，这是因为在系统测试2中，Boston-
617 Trenton边是在Boston-New York边之前输入的。

附录1 数学背景

A1.1 简介

数学是人类脑力劳动的突出成就之一。它对客观现象提供的抽象模型推动了科学和工程技术在每一个领域的发展。大部分计算机科学都是基于数学的，本书也毫不例外。本附录介绍了书中涉及到的数学概念。末尾给出了一些习题，读者可以通过练习巩固学到的知识。

A1.2 函数和序列

Whitehead和Russell（1910）首先揭示了数学上的一个令人惊异的特征，即它只需要两个基本概念。其他的每个数学术语都可以建立在简单的**集合**和**元素**上。例如，有序对<a, b>可以定义为包含两个元素的集合:

$$<a, b>=\{a, \{a, b\}\}$$

元素a被称作是有序对的**第一成分**，而b被称作是**第二成分**。

给定两个集合A和B，可以定义一个从A到B的**函数**f，写作:

$$f: A \rightarrow B$$

它作为有序对<a, b>的集合，其中a在A中，b在B中，而A中的每个元素恰好是f里某一个有序对的第一成分。因此在一个函数中，没有两个有序对的第一成分是相同的。集合A和B被分别称作**定义域**和**值域**。

例如，

$$f=\{<-2, 4>, <-1, 1>, <0, 0>, <1, 1>, <2, 4>\}$$

定义了定义域为{−2, −1, 0, 1, 2}及值域为{0,1,2,4}的“平方”函数。函数中没有两个有序对的第一成分相同，但是两个有序对可以拥有相同的第二成分。例如，对

$$<-1, 1>\text{和}<1, 1>$$

具有相同的第二成分，即1。

如果<a, b>在f中，可以写作$f(a)=b$。这给出了一个更为熟悉的关于函数f的描述。它的定义是:

$$f(i)=i^2 \quad i\text{在}-2\text{到}2\text{之间}$$

函数的另一个名字是**映射**。这是第10章中用到的术语，用来描述一个容器，其中的每个值由一个惟一的键成分以及第二成分组成。实际上就是从键到第二成分的函数，这也正是键必须是惟一的原因。

有限序列t是这样一个函数，对某些正整数k（称作序列**长度**），t的定义域是集合{0, 1, 2, ... , k-1}。例如，下面定义了一个长度为4的有限序列:

$t(0)$=“Karen”

$t(1)$="Don"

$t(2)$="Mark"

$t(3)$="Courtney"

因为每个有限序列的定义域都是从0开始的，所以通常定义域可以是隐式的，写作:

t="Karen", "Don", "Mark", "Courtney"

A1.3 累加和累乘

数学需要相当多的符号处理。因此，简洁是一个很重要的因素。总和的表示就是一个很好的符号简化的例子。将

$$x_0 + x_1 + x_2 + \cdots + x_{n-1}$$

写成

$$\sum_{i=0}^{n-1} x_i$$

这个表达式可以理解成“当i取从0到$n-1$的值时，x_i的总和”。我们称i为“计数下标”。一个计数下标对应着**for**语句中的一个循环控制变量。例如，下面的代码将在sum中储存数组x里从0到n-1成员的总和:

620

```
sum=0.0;
for(int i=0;i<n;i++)
    sum+=x[i];
```

当然，字母i没有任何特别的意义。例如，也可以将

$$\sum_{j=1}^{10}(1/j)$$

作为

$$1+1/2+1/3+\cdots+1/10$$

的简写形式。同样，如果$n \geqslant m$，

$$\sum_{k=m}^{n}(k2^{-k})$$

就是

$$m2^{-m} + (m+1)2^{-(m+1)} + \cdots + n2^{-n}$$

的简写形式。

另一种简写形式是累乘符号，它比累加符号出现的频率低。例如，

$$\prod_{k=0}^{4} a[k]$$

是

$$a[0]*a[1]*a[2]*a[3]*a[4]$$

的简写形式。

A1.4 对数

John Napier是一位苏格兰男爵及兼职数学家，他在1614年发表的论文中首先描述了对数。从那时起直到计算机的发明，对数的主要价值就是在于处理算术的难题：它们使得很大数字的乘法（和除法）可以仅通过加法（和减法）实现。

现今的对数只有很少的计算应用——例如，测量地震的里氏震级。但是正如第3~14章所述，对数提供了进行算法分析的工具。

我们根据幂来定义对数，就好像可以根据减法定义加法，而除法又可以根据乘法定义一样。给定一个实数$b>1$，称b为**基数**。以b为基数的任意大于0的实数x的**对数**可以写成 621

$$\log_b x$$

它的定义是该实数y满足

$$b^y=x$$

例如，$\log_2 16=4$，因为$2^4=16$。同理，$\log_{10}100=2$，因为$10^2=100$。那么$\log_2 64$是多少？估算$\log_{10}64$。

通过给出的对数定义以及相应的幂性质可以验证下面的关系：对任意实数值$b>1$以及任意正实数x和y，有

1) $\log_b 1=0$

2) $\log_b b=1$

3) $\log_b(xy)=\log_b x+\log_b y$

4) $\log_b(x/y)=\log_b x-\log_b y$

5) $\log_b b^x=x$

6) $b^{\log_b x}=x$

7) $\log_b x^y=y\log_b x$

根据这些等式，可以获得基数转换的公式。对任意大于1的基数a和b以及任意$x>0$，

$$\begin{aligned}\log_b x &= \log_b a^{\log_a x} && \text{（根据性质5）}\\ &= (\log_a x)(\log_b a) && \text{（根据性质7）}\end{aligned}$$

基数e（≈ 2.718）在微积分学中有重要的意义；因此，以e为基数的对数被称作是**自然对数**，并用ln替代了$\log_e$。

要将一个自然对数转换成以2为基数的对数，可以应用基数转换公式。对任意$x>0$，

$$\ln x=(\log_2 x)(\ln 2)$$

两边同除以ln2，得到

$$\log_2 x=\ln x/\ln 2$$

假设预先确定了函数ln，那么这个等式就可以用于$\log_2 x$的近似。

函数ln和它的逆exp使得我们可以执行幂运算。例如，假设想计算x^y，其中x和y都是实数且$x>0$。首先重写x^y：

$$\begin{aligned}x^y &= e^{\ln(x^y)} && \text{（根据性质6）}\\ &= e^{y\ln x} && \text{（根据性质7）}\end{aligned}$$

在C++中，这最后一个表达式可以重写为：

```
exp(y*log(x))
```

622 exp和log在<cmath>中进行了定义。

A1.5 数学归纳

对数分析中的很多声明也可以解释成整数的性质。例如，对任意正整数n，

$$\sum_{i=1}^{n} i = n(n+1)/2$$

在这样的情况下，可以使用数学归纳法证明结论。

> **数学归纳原理** 令$S_1, S_2, \ldots$是一系列命题。如果同时满足下面两个条件，
> 1) S_1为真。
> 2) 对任意正整数n，若S_n为真，则S_{n+1}为真。
> 那么对任意正整数n，命题S_n为真。

为了帮助读者理解这个原理的意义，假设$S_1, S_2, \ldots$是一系列命题，并满足条件1和条件2。根据条件2，因为S_1为真，所以S_2必定为真。再次应用条件2，因为S_2为真，所以S_3必定为真。从此不断使用条件2，可以断定S_4为真，然后S_5为真，等等。这说明原理中的结论是合理的。

要使用数学归纳来证明一个声明，首先需要将声明描述成一系列命题$S_1, S_2, \ldots$的形式。然后证明S_1为真——这称作“基础情况”。最后，需要证明条件2——“归纳情况”。这个证明的框架如下：令n是任意正整数并假设S_n为真。要证明S_{n+1}为真，需要将S_{n+1}代回S_n，也就是假设条件中。证明的剩余部分通常会利用算术或代数。

例A1.1

我们将使用数学归纳原理证明下面的声明：对任意正整数n，

$$\sum_{i=1}^{n} i = n(n+1)/2$$

证明 首先将声明描述成一系列命题。当n=1, 2, ...时，令S_n是命题

$$\sum_{i=1}^{n} i = n(n+1)/2$$

623 1) 基础情况。

$$\sum_{i=1}^{1} i = 1 = 1(2)/2$$

所以S_1为真。

2) 归纳情况。令n为任意正整数并假设S_n为真。也就是，

$$\sum_{i=1}^{n} i = n(n+1)/2$$

需要证明S_{n+1}为真：

$$\sum_{i=1}^{n+1} i = (n+1)(n+2)/2$$

根据如下观察可以将S_{n+1}代回S_n：前(n+1)个整数的总和是前n个整数的总和加上n+1。即

$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1)$$

$$= n(n+1)/2 + (n+1) \qquad \text{因为假设} S_n \text{为真}$$
$$= n(n+1)/2 + 2(n+1)/2$$
$$= (n(n+1) + 2(n+1))/2$$
$$= (n+2)(n+1)/2$$

于是推断S_{n+1}为真（只要S_n为真）。因此，根据数学归纳原理，对任意正整数n，命题S_n为真。

数学归纳原理的一个重要变形是下列形式：

数学归纳原理——增强形式 令$S_1, S_2, \ldots$是一个命题序列。如果满足下列条件，

1) S_1为真。

2) 对任意正整数n，若$S_1, S_2, \ldots, S_n$为真，则S_{n+1}为真。

那么对任意正整数n，S_n为真。

这个版本和前一版本的区别在于归纳情况。当希望证明S_{n+1}为真时，可以假设$S_1, S_2, \ldots, S_n$为真。

数学归纳原理和数学归纳原理的增强形式是等价的。

在继续深入之前，先要说服自己确信这个原理是合理的。乍看起来这个增强形式比原版本的功能要强大得多，而实际上，它们是等价的。

现在应用数学归纳原理来得出一个简单而重要的结论。 624

例A1.2

对任意正整数n，它可以不断除以2直到n=1的次数是floor($\log_2 n$)。

证明对任意正整数n，下面的循环语句的迭代次数是floor($\log_2 n$)：

```
while(n>1)
n=n/2;
```

（回忆一下，函数floor(x)返回≤x的最大整数。例如，floor(18)返回18。）

证明 当n=1, 2, ...时，令t(n)表示循环迭代次数。当n=1, 2, ...时，令S_n表示命题：

$$t(n) = \text{floor}(\log_2 n)$$

1) 基本情况。当n=1时，循环根本就不执行，因此t(n)=0=floor($\log_2 n$)，也就是说S_1为真。

2) 归纳情况。令n为任意正整数，并假设$S_1, S_2, \ldots, S_n$都是真。需要证明S_{n+1}为真。这分两种情况：

一种情况是n+1是偶数。那么第一次迭代之后的迭代次数等于t((n+1)/2)。所以，有

$$t(n+1) = 1 + t((n+1)/2)$$
$$= 1 + \text{floor}(\log_2((n+1)/2)) \qquad \text{(根据归纳假设)}$$

= 1+floor($\log_2$(n+1)−$\log_2$(2)) (因为商的对数就等于对数的差)
= 1+floor($\log_2$(n+1)−1)
= 1+floor($\log_2$(n+1))−1
= floor($\log_2$(n+1))

因此S_{n+1}为真。

另一种情况是n+1是奇数。那么第一次迭代之后的迭代次数就等于$t(n/2)$。所以，有

t(n+1) = 1+t(n/2)
= 1+floor($\log_2$(n/2)) (根据归纳假设)
= 1+floor($\log_2$n−$\log_2$2)
= 1+floor($\log_2$n−1)
= 1+floor($\log_2$n)−1
= floor($\log_2$n)
= floor($\log_2$(n+1)) (因为$\log_2$(n+1)不会是一个整数)

因此S_{n+1}为真。

625 综上所述，根据数学归纳原理的增强形式可知，对任意正整数n，S_n为真。

看过这个范例之后，注意几乎可以用相同的形式证明折半查找的最坏情况下，迭代次数是

$$\text{floor}\ (\log_2 n)+1$$

在数学归纳原理的原形式和增强形式中，基本情况都是证明S_1为真。而对某些情形，可能需要从除1之外的其他整数开始证明。例如，假设想证明对任意$n \geqslant 4$，

$$n!>2^n$$

（注意对n=1，2，3，这个命题为假）。那么命题序列就是S_4，S_5，…。基本情况需要证明S_4为真。

还有一些情形下可能有几个基本情况。例如，假设想证明对任意正整数n，fib(n)<2^n（实验10中定义的fib方法，计算了斐波纳契数。）基本情况是

$$\text{fib}(1)<2^1 \text{和} \text{fib}(2)<2^2$$

由此观察可以得到下面的原理：

数学归纳原理——通用形式 令K和L是任意正整数并满足$K \leqslant L$，令S_K，S_{K+1}，…是一个命题序列。如果满足下面的两个条件，

1) $S_K, S_{K+1}, \ldots, S_L$为真。

2) 对任意正整数$n \geqslant L$，若$S_K, S_{K+1}, \ldots, S_n$为真，则$S_{n+1}$为真。

那么对任意正整数$n \geqslant K$，命题S_n为真。

通用形式是对增强形式的扩展，它使得命题序列可以从任意整数(K)开始并可以有任意数量的基本情况（$S_K, S_{K+1}, \ldots, S_L$）。如果$K$=$L$=1，那么通用形式就还原成了增强形式。

例A1.3和A1.4使用了数学归纳原理的通用形式来证明有关斐波纳契数的结论。

例A1.3

证明对任意正整数n，

626 $$\text{fib}(n)<2^n$$

证明 对$n=1, 2, \ldots$，令S_n代表命题

$$\text{fib}(n)<2^n$$

在数学归纳原理的通用形式里，$K=1$表示序列从1开始；$L=2$表示有两个基本情况。

1) $\text{fib}(1)=1<2=2^1$，因此S_1为真。$\text{fib}(2)=1<4=2^2$，因此S_2为真。

2) 令n为任意$\geqslant 2$的整数，并假设S_1，S_2，…，S_n为真。需要证明S_{n+1}为真（也就是，$\text{fib}(n+1)<2^{n+1}$）。根据斐波纳契数的定义，

$$\text{fib}(n+1)=\text{fib}(n)+\text{fib}(n-1) \qquad n\geqslant 2$$

因为$S_1, S_2, \ldots, S_n$为真，所以一定有S_{n-1}和S_n为真。因此

$$\text{fib}(n-1)<2^{n-1} \quad \text{且} \quad \text{fib}(n)<2^n$$

然后得到

$$\begin{aligned}\text{fib}(n+1) &= \text{fib}(n)+\text{fib}(n-1)\\ &< 2^n+2^{n-1}\\ &< 2^n+2^n\\ &= 2^{n+1}\end{aligned}$$

因此$\text{fib}(n+1)$为真。

综上所述，根据数学归纳原理的通用形式，可以断定对任意正整数n，$\text{fib}(n)<2^n$。

现在可以用相似的方法证明下面的斐波纳契数的下界：

$$\text{fib}(n)>(6/5)^n \qquad n\geqslant 3$$

提示 利用数学归纳原理的通用形式，令$K=3$，$L=4$。

现在已经确定了斐波纳契数的下界和上界，读者可能会好奇这些界限是否能再改进。还可以做得更好！在例A1.4中检验了第n个斐波纳契数的精确的、封闭的公式。“封闭”的公式是指既不是递归又不迭代的公式。 627

例A1.4

证明对任意正整数n，

$$\text{fib}(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

在考虑证明之前，先计算几个数值，确信公式能提供正确的值。

证明 对$n=1, 2, \ldots$，令S_n是命题

$$\text{fib}(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

令$x=(1+\sqrt{5})/2$，$y=(1-\sqrt{5})/2$。注意

$$x^2=\frac{(1+\sqrt{5})^2}{4}=\frac{1+2\sqrt{5}+5}{4}=\frac{3+\sqrt{5}}{2}=x+1$$

同理，$y^2=y+1$。

现在进行证明。

1) $\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}-\frac{1-\sqrt{5}}{2}\right)=1$，因此$S_1$为真。要证明$S_2$为真，可以按如下方式进行:

$$\begin{aligned}\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^2-\left(\frac{1-\sqrt{5}}{2}\right)^2\right)&=(1/\sqrt{5})(x^2-y^2)\\&=(1/\sqrt{5})(x+1-(y+1))\\&=(1/\sqrt{5})(x-y)\\&=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}-\frac{1-\sqrt{5}}{2}\right)\\&=1=\text{fib}(2)\qquad\text{（根据定义）}\end{aligned}$$

因此S_2也为真。

2) 令n为任意大于1的正整数，并假设$S_1, S_2, \ldots, S_n$为真。需要证明S_{n+1}为真；也就是，

$$\text{fib}(n+1)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

根据斐波纳契数的定义，

$$\text{fib}(n+1)=\text{fib}(n)+\text{fib}(n-1)$$

628 因为S_n和S_{n-1}为真，所以有（使用x和y）

$$\text{fib}(n)=(1/\sqrt{5})(x^n-y^n)$$

以及

$$\text{fib}(n-1)=(1/\sqrt{5})(x^{n-1}-y^{n-1})$$

代换进前面的公式可得:

$$\begin{aligned}\text{fib}(n+1)&=(1/\sqrt{5})(x^n+x^{n-1}-y^n-y^{n-1})\\&=(1/\sqrt{5})(x^{n-1}(x+1)-y^{n-1}(y+1))\\&=(1/\sqrt{5})(x^{n-1}x^2-y^{n-1}y^2)\\&=(1/\sqrt{5})(x^{n+1}-y^{n+1})\end{aligned}$$

所以S_{n+1}为真。

综上所述，根据数学归纳原理的通用形式可知，对任意正整数n，S_n为真。

例A1.5提出了非空二叉树的一个结论：树叶数量至多是树中项的数量加1再除以2。归纳是基于树的高度的，因此基本情况是针对单项的树，也就是高度为0的树进行证明的。

例A1.5

令t是一非空二叉树，它有leaves(t)个树叶和$n(t)$个项。证明

$$\text{leaves}(t)\leq\frac{n(t)+1}{2.0}$$

证明 对k=0,1,2,...，令S_k是命题：对任意高度为k的非空二叉树t，

$$\text{leaves}(t) \leqslant \frac{n(t)+1}{2.0}$$

1) 如果t的高度是0，那么leaves(t)=n(t)=1，因此

$$1 = \text{leaves}(t) \leqslant \frac{n(t)+1}{2.0}=1$$

所以S_0为真。

2) 令k是任意$\geqslant 0$的整数，并假设S_0, S_1, ... , S_k为真。需要证明S_{k+1}为真。令t是高度为k+1的非空二叉树。leftTree(t)和rightTree(t)的高度都$\leqslant k$，因此它们都满足了归纳假设。即， 629

$$\text{leaves}(\text{leftTree}(t)) \leqslant \frac{n(\text{leftTree}(t))+1}{2.0}$$

且

$$\text{leaves}(\text{rightTree}(t)) \leqslant \frac{n(\text{rightTree}(t))+1}{2.0}$$

而t中的每个树叶不是在leftTree(t)中就是在rightTree(t)中。也就是说，

$$\text{leaves}(t) = \text{leaves}(\text{leftTree}(t)) + \text{leaves}(\text{rightTree}(t))$$

那么就有

$$\begin{aligned}\text{leaves}(t) &\leqslant \frac{n(\text{leftTree}(t))+1}{2.0}+\frac{n(\text{rightTree}(t))+1}{2.0}\\ &=\frac{n(\text{leftTree}(t))+n(\text{rightTree}(t))+1+1}{2.0}\end{aligned}$$

除了t的根项，t中的每个项不是在leftTree(t)就是在rightTree(t)里，因此

$$n(t)=n(\text{leftTree}(t))+n(\text{rightTree}(t))+1$$

用这个等式的右边代换左边的内容，前面的不等式就变成了

$$\text{leaves}(t) \leqslant \frac{n(t)+1}{2.0}$$

也就是说，S_{k+1}为真。

综上所述，根据数学归纳原理的通用形式，对任意非负整数k，S_k为真。证明结束。

A1.6 归纳和递归

归纳和递归很相似，但方向是不同的。

归纳和递归很相似，两者都有一些基本情况，同样，也都有一般的情况，它们可以简化为更简单的情况，并最终得到基本情况。但是方向是不同的。使用递归，是从一般情况开始并最终简化得到基本情况。使用归纳，是从基本情况开始，并利用它来开发一般情况。

例A1.6涉及了开放地址散列（第13章）的分析，它是从一个递归定义开始，然后猜测出一个封闭的形式。

例A1.6

开发函数E的封闭形式，对任意k和m，$0 \leqslant k<m$:

630

$$E(0,m)=1 \quad \text{其中m>1}$$

$$E(k,m)=1+\frac{k}{m}E(k-1,m-1) \quad \text{其中}1\leqslant k<m$$

解 首先注意到

$$\begin{aligned}E(1,m)&=1+\frac{1}{m}E(0,m-1)\\&=1+\frac{1}{m}*1\\&=\frac{m+1}{m}\end{aligned} \quad \text{对所有}m\geqslant 1$$

同理，

$$\begin{aligned}E(2,m)&=1+\frac{2}{m}E(1,m-1)\\&=1+\frac{2}{m}\frac{m}{m-1}\\&=\frac{m+1}{m-1}\end{aligned} \quad \text{对所有}m>2$$

而且，

$$\begin{aligned}E(3,m)&=1+\frac{3}{m}E(2,m-1)\\&=1+\frac{3}{m}\frac{m}{m-2}\\&=\frac{m+1}{m-2}\end{aligned} \quad \text{对所有}m>3$$

因此可以推测

$$E(k,m)=\frac{m+1}{m+1-k} \quad \text{对所有}m\text{和}k,\ 0\leqslant k<m$$

这个推测可以归纳（对k或m）证明。例如，如果对k进行归纳，那么k=0,1,2,...时，命题S_k的序列是

$$E(k,m)=\frac{m+1}{m+1-k} \quad \text{对所有}m,\ m>k$$

习题

A1.1 使用数学归纳证明，在第4章的汉诺塔游戏中，对任意正整数n，从一个杆移动n个盘子到另一个杆共需要2^n-1次移动。

A1.2 使用数学归纳证明，对任意正整数n，

631

$$\sum_{i=1}^{n}Af(i)=A\sum_{i=1}^{n}f(i)$$

其中A是一个常数，f是一个函数。

A1.3 使用数学归纳证明，对任意正整数n，

$$\sum_{i=1}^{n}(i*2^{i-1})=(n-1)*2^{n}+1$$

A1.4 令n_0是满足下列条件的最小正整数

$$\text{fib}(n_0)>n_0^2$$

a. 求出n_0。

b. 使用数学归纳证明，对任意$n \geqslant n_0$，

$$\text{fib}(n)>n^2$$

A1.5 证明fib是$O((1+\sqrt{5})^n/2)$。

提示 参阅例A1.4中的公式。注意

$$\text{abs}((1-\sqrt{5})/2)<1$$

因此，当n“非常大”时，可以忽略$((1-\sqrt{5})/2)^n$。

A1.6 证明对任意非负整数n，

$$\sum_{i=0}^{n}2^{i}=2^{n+1}-1$$

632

附录2 string类

A2.1 简介

本附录的目的是熟悉标准C++中的string类。一旦了解了这个类的功能和简洁性，相信读者将很乐意学习它——特别是学习过C中的字符串之后。

在一个文件中使用string类必须包含下面的两行:

```
#include <string>
using namespace std;
```

这两行都是发送给编译器的指令。第一行请求了文件（它是标准模板库的一部分），它声明在当前文件中可以访问string类。第二行说明了标准模板库的标识符将不使用限定词std出现在当前文件中。

字符串里的单个字符可以通过它的索引访问，就像数组一样。例如，如果一个字符串对象s包含“nevermore”，那么s[0]就包含了n，s[1]包含e，等等。不要忘记索引是从0开始的。

string类有100多个方法，因此不要指望能全部记住（甚至是认识）。附录A2.2节中包括了最广泛使用的方法的接口和示例。然后将介绍一个简单的字符串处理程序，随后是string类的典型实现的框架。

A2.2 string类的声明

本节分成四部分讲述：构造器，运算符，非成员函数以及其他方法。为了增强易读性，区分下面的方法接口与标准C++规范，这里使用**char**代替了**charT**，并用**unsigned int**代替了size_type。

A2.2.1 构造器

```
1. //后置条件: 该字符串为空。
   string();
```

例 按如下方式将一个字符串对象初始化为空:

```
string s1;
```

```
2. //后置条件: 该字符串被初始化为字符串s的拷贝。
   string(const char* s);
```

例 可以在一条语句中定义并初始化一个字符串:

```
string s2("bed");
```

现在s2中顺序包含了字符b、e和d。

注意 本构造器有一个常见的等价形式，

```
string s2="bed";
```

```
3. //前置条件: pos<=str中的字符数量。
   //后置条件: 这个字符串被初始化为从pos索引开始，并且长度为n和
   //          str.size()-pos之间较小者的str的子字符串的拷贝。如果省略第三个变元，
   //          那么将假定第三个变量是最大的无符号整数。如果也省略了第二个变元，那么第
   //          二个变元就取0。
string(const string str, unsigned int pos=0, unsigned int n=-1);
```

例 假设定义如下:

```
string s3 = "jabberwocky";
string s4 (s3),
       s5 (s3, 5, 3);
```

那么s4包含了“jabberwocky”，s5包含了“rwo”；也就是从索引5开始，长度为3的s3的子字符串。

注意 -1的二进制表示全部是由1组成的。当这被解释成一个**unsigned int**时，它就表示了可能的最大的**unsigned int**。因此当头中有

634

```
unsigned int n=-1
```

时，这实际是一个有效（但易混淆）的方式，将n的缺省值设置成最大的**unsigned int**。

A2.2.2 运算符

```
4. //后置条件: 这个字符串中包含了str的拷贝。
   string& operator=(const string& str);
```

例 假设s2采取和构造器3的示例相同的定义。然后可以定义

```
string s3=s2;
```

现在s2和s3中包含了相同的字符串。改变s2的值将不会影响s3的值（反之亦然）。例如，假设随后进行赋值

```
s2="flower"; // 赋值运算符
```

现在s2中包含了“flower”，而s3中包含的仍旧是“bed”。

注意 字符串赋值语句的右边可以由一个字符串常量或一个字符、一个字符串组成。例如，

```
s2="start-up";
s3='?';
```

```
5. //前置条件: pos<这个字符串中字符的数量。
   //后置条件: 返回对这个字符串中索引pos上字符的引用。
   char& operator[](unsigned int pos);
```

例 假设字符串对象s2中包含“flower”，并编写

```
s2[0]='g';
```

因为这个运算符（即索引运算符）将返回一个引用，所以这条赋值语句用‘g’替代了s3的索引0上的‘f’。因此现在s2中将包含“glower”。

A2.2.3 非成员函数

这些函数可能是其他类中的方法。 635

6. //后置条件：str被插入到os中，并返回os。
   ostream& **operator**<<(ostream& os, **const** string& str);

例 假设有

```
string s6="yes",
s7="no";
cout<<s6<<"or"<<s7<<endl;
```

输出将是

```
yes or no
```

7. //后置条件：跳过is中所有的空白字符（空格和行尾标志），然后从is中取出
   //　　字符序列直到下一个空白字符（但不包括），并将它存入str。返回is。
   istream& **operator**>>(istream& is, string& str);

例 假设有

```
string s,
       t;
cin>>s;
cin>>t;
```

如果输入字符流包含了一个空行，随后是

```
This was the next line.
```

那么s中将包含“this”而t中将包含“was”。

8. //后置条件：从is中取出直到行尾的字符（除了分隔符‘\n’以外），并将其插入str，
   //　　然后返回is。
   ifstream getline(Ifstream& is, string& str);

例 假设有

```
string line;
getline(cin,line);
```

那么输入中下一行的内容（不包含分隔符‘\n’）将存储在line中。

注意 这个函数返回了一个ifstream对象。当到达文件尾时，返回值是NULL，它和0以及**false**是等价的。因此可以不断读入文件my_file，直到到达文件末尾： 636

```
while(getline(my_file, line))
{
   ...
```

9. //后置条件：返回lhs和rhs结合（即连接）组成的字符串。
   string **operator**+(**const** string& lhs, **const** string& rhs);

例 假设有

```
string s6="first",
       s7="last",
       s8=s6+s7;
```

那么s8将包含“firstlast”，也就是字符串“first”和“last”连接组成的字符串。如果希望在s8中用空格分隔s6和s7，可以编写

```
string s6="first",
       s7="last",
       s8=s6+" "+s7;
```

10. //后置条件：如果lhs和rhs包含相同序列的字符，就返回真。否则，返回假。
    bool operator==(const string& lhs, **const** string& rhs);

例 假设有

```
string sa="nevermore";
       sb="nevermore";
cout<<(sa==sb)<<endl;
```

输出将是0——也就是假，因为sb比sa多一个字符；在sb尾部的空格是一个字符。

11. //后置条件：如果lhs按词典顺序排在rhs之后，就返回真。否则，返回假。
    bool operator==(const string& lhs, **const** string& rhs);

例 假设有

```
string s1 = "elephant",
       s2 = "mouse";
```

637

```
if (s1 < s2)
       cout << "\"elephant\" is less than \"mouse\".";
else
       cout << "\"elephant\" is not less than \"mouse\".";
```

因为在ASCII码序列中，e位于m之前，所以在词典顺序中“elephant”就排在“mouse”之前。因此输出将是

```
"elephant" is less than "mouse".
```

A2.2.4 既不是构造器也不是运算符的方法

下面的方法按照字母顺序排列。

12. //后置条件：返回值是指向有size()+1个项的数组的第一个项的指针，其中前
    //　　size()个项就等于这个数组的相应项，而最后一项是一个空字符。
    const char* c_str();

例 在ofstream（或ifstream）类的open方法中，第一个变元必须是代表文件名的字符数组。因此可以将文件名作为一个字符串读入，并通过c_str打开文件：

```
cout<<"Enter the name of the file you want to create:";
string out_file_name;
```

```
cin>>out_file_name;
ofstream out_file;
out_file.open(out_file_name.c_str(),ios::out);
```

13. //前置条件：pos<=这个字符串的字符数量。
    //后置条件：在这个字符串中删除一个子字符串——从pos索引开始、长度为n和
    //　　str.size()-n中较小者的子字符串。返回对这个字符串（删除子字符串之后）
    //　　的引用。
    string& erase(**unsigned int** pos=0, **unsigned int** n=-1);

例 假设有

```
string s="jabberwocky";
s.erase(5,3);
cout<<s<<endl;
```

那么就从s中删除从索引5开始长度为3的子字符串。因此输出将是 638

```
"jabbecky"
```

注意 *关于头中的赋值语句*

unsigned int n=-1

请参阅构造器3中的“注意”部分。

14. //前置条件：迭代器位于这个字符串的某一项上。
    //后置条件：在这次调用之前位于position位置上的项从这个字符串里删除。
    //　　调用前索引>position的位置上的每个项分别移动到低一位索引位置。
    //　　worstTime(n)是O(n)。
    void erase(iterator position);

注意 *这其实和vector类中一个参数的erase方法的接口相同。事实上，string类可以看作是vector<**char**>的一个扩展版本。并且，就像vector类中一样，string类的erase方法也会使删除点之后的迭代器失效。*

15. //后置条件：从索引pos开始，如果str作为这个字符串的一个子串出现，那么就
    //　　返回str在这个字符串中第一次出现的起始位置的索引。否则，返回-1。
    int find(**const** string& str, **unsigned int** pos=0) **const**;

例 假设有

```
string message="The snow is now on the ground.",
       code="now";
cout<<message.find(code);
```

“now”在字符串message中的第一次出现是在索引5（记着是从索引0开始），因此输出将是

```
5
```

假设将语句换成

```
cout<<message.find(code,6);
```

寻找的字符串仍旧是“now”，但是搜索将从索引6开始。从该索引开始，字符串message中“now”的第一次出现是在索引12上的子字符串，因此输出将是

639

12

注意，尽管搜索是在索引6开始，但是这个索引是从字符串开头算起的。

最后，假设换成

```
cout<<message.find(code,14);
```

因为字符串message从索引14之后就没有“now”出现，所以输出将解释成**unsigned int**项的-1（参阅构造器3的“注意”部分）。例如，如果**unsigned int**项占据32位，那么输出将是

```
4294967295
```

这并非预期的结果！为了补救这个状况，将find方法的返回结果强制转换成一个**int**项：

```
cout<<int(message.fin(code,14));
```

输出是

```
-1
```

```
16. //前置条件: pos1<=size()
    //后置条件: 字符串str被插入这个字符串的索引pos1之前，并返回对这个字符
    //          串的引用。
    string& insert(unsigned int pos1, const string& str);
```

例 假设有

```
string s="bed",
       t="and";
cout<<s.insert(1,t)<<endl;
```

那么字符串t将插入字符串s中b和e之间，s将包含“banded”。

```
17. //前置条件: 迭代器位于这个字符串开头和尾部之间的位置。
    //后置条件: x的拷贝放入迭代器位于的位置。调用前index>=position的索引
    //          位置上的每个字符分别被移动到高一位的索引上。返回位于新插入字符上的迭代
    //          器。worstTime(n)是O(n)。
    iterator insert(iterator position, const char x);
```

注意 这个方法其实和vector类中两个参数的insert方法是相同的。所有插入点之后的迭代器将失效。

640

```
18. //后置条件: 返回这个字符串中的字符数量。
    unsigned int length() const;
```

例 假设string对象s2和s3的值同方法16的示例中一样，并编写

```
cout<<s2.length()<<""<<s3.length()<<endl;
```

输出将是

```
6 3
```

注意 string类还有一个size()方法，它和length()方法是等价的。

```
19. //后置条件: 从索引pos后退搜索，如果str作为这个字符串的子串出现，那么
    //          将返回str在这个字符串中最后一次出现的起始位置的索引。否则，返回-1。
```

```
int rfind(const string& str, unsigned int pos=-1) const;
```

例 假设有

```
string code="The snow is now on the ground, I know",
       match="now";
cout<<code.rfind(match)<<"**"
    <<code.rfind(match,22)<<"**"
    <<code.rfind(match,10)<<"**"
    <<int(code.rfind(match,3))<<endl;
```

如果从string对象code尾部开始，“now”最后一次出现在索引34。如果从索引22开始，并向code开头搜索，那么“now”最后一次出现在索引12。如果从索引10开始并向开头搜索，那么“now”最后一次出现在索引5。如果从索引3开始并向code开头搜索，那么“now”则没有出现。因此输出是

```
34**12**5**-1
```

这个例子中代码的最后一行强制转换成**int**的原因已经在方法15（find方法）的“注意”部分讨论过了。

```
20. //前置条件: pos<=size();
    //后置条件: 返回值是从pos索引开始、长度为n和size()-pos中较小者的
    //          字符串的子串。
    string substr(unsigned int pos=0, unsigned int n=-1) const;
```

641

例 假设有

```
string s="fruits and vegetables";
cout<<s.substr()<<endl
    <<s.substr(7)<<endl
    <<s.substr(7,3);
```

输出将是

```
fuits and vegetables
and vegetables
and
```

注意 本方法头中的

```
unsigned int n=-1
```

请参阅构造器3的“注意”部分。

A2.2.5 一个字符串处理程序

下面的程序举例一致说明了几个字符串方法。程序计算了文件中某些目标字符串的出现次数。输入由文件名和目标字符串组成。

```
#include <string>
#include <fstream>

using namespace std;
```

```
int main( )
{
    const string INPUT_PROMPT =
        "Please enter the name of the input file: ";
    const string TARGET_PROMPT =
        "Please enter the target string: ";
    const string RESULT =
        "The number of occurrences of the target string is ";
    const string CLOSE_WINDOW_PROMPT =
        "Please press the Enter key to close this output window.";
    string in_file_name,
        target,
642     line;
    ifstream in_file;
    int count = 0,
      pos;
    cout << INPUT_PROMPT << endl;
    cin >>in_file_name;
    in_file.open( in_file_name.c_str( ), ios::in );
    cout << endl << TARGET_PROMPT << endl;
    cin >> target;
    while (getline (in_file, line))
    {
        pos = line.find (target);
        while (pos != -1)
        {
            count ++;
            line = line.substr (pos + target.length( ));
            pos = line.find (target);
        } // 当line中仍旧保存target的拷贝时
    } // 当文件还保存更多行时
    cout << endl << RESULT << count << endl;
    cout << endl << endl << CLOSE_WINDOW_PROMPT;
    cin.get( );
    return 0;
} // main
```

注意1 在这个程序中，不需要将find方法的返回值强制转换成**int**项，因为该值直接被赋给了一个**int**变量，pos。

注意2 能否使用>>读入每个单词，然后将该单词和变量target的内容比较？

```
string word;
...
```

```
while(in_file>>word)
    if(word==target)
        count++;
```

这个方法不会检测一个单词内部目标的出现。例如，如果target是“count”，将不会在“uncountable”或“accountant”中检测该字符串。 643

A2.3 string类的字段和实现

string类的基础类是basic_string。实际情况如下：

```
typedef basic_string<char> string;
```

string类和很多标准模板库里的容器类很相似。这里是它们共有的方法：

empty, size, insert, erase, find, begin, end, =, ==, !=

string类和vector类还有一个索引运算符**operator**[]。事实上，string类的典型（简化）设计中包含了三个和vector字段相对应的字段：

```
char* data;//对应vector字段start
unsigned nchars;//对应vector字段finish
unsigned capacity;//对应vector字段end_of_storage
```

data字段是指向项类型为**char**的数组的指针。nchars字段保存了字符串中当前的字符数量。capacity字段保存了data（指向的）数组的当前大小。

例如，假设开始时是

```
string s;
```

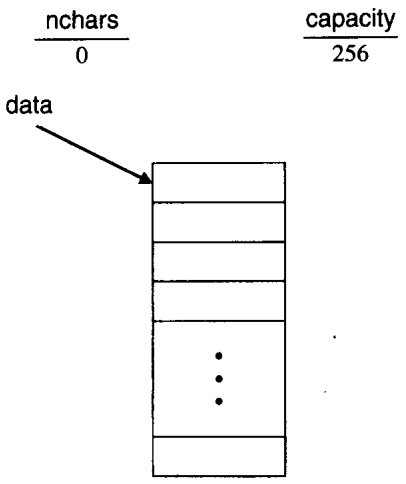
那么就分配了某个固定大小（比如256）的数组。图A2-1显示了内存的相关特性。如果接着赋值

```
s="yes";
```

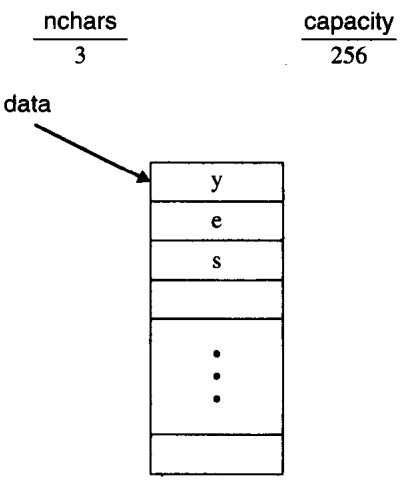
那么相关内存单元的结果如图A2-2所示。很容易看出大多数string类的方法是如何使用这些字段实现的。例如，下面是substr方法的定义，假设一个构造器返回一个字符串指针（指向一个字符数组）以及一个**unsigned int**：

```
string substr (unsigned pos = 0, unsigned n = -1) const
{
    unsigned rlen = n < (length( ) - pos) ? n : (length( ) -pos);
    return string (data + pos, rlen);
} // substr方法
```

644



图A2-1 一个空字符串的设计表示



图A2-2 三个字符的字符串的设计表示

附录3 多 态 性

A3.1 简介

面向对象编程有三个基本特色：字段和方法封装进一个实体，子类对类的字段和方法的继承，以及多态性。**多态性**——来自希腊单词的“多”和“形态”，是根据消息发送者对象的运行时类型而对消息进行不同解释的能力。例如，在第1章中，创建了一个使用readInto方法的Employee类，然后创建了一个子类HourlyEmployee，它使用了新版本的readInto方法。C++支持如下的程序：

```
Employee* employeePtr;
employeePtr = new Employee;
// employeePtr指向Employee类中的一个对象。
...
employeePtr -> readInto( );
...
employeePtr = new HourlyEmployee;
// 现在employeePtr指向HourlyEmployee类的一个对象……
employeePtr -> readInto( );
...
```

最初，employeePtr指向Employee类的一个对象，因此第一条消息employeePtr->readInto()将读入一个姓名和薪水总额。随后，employeePtr指向HourlyEmployee类的一个对象，因此第二条消息employeePtr->readInto()将读入一个姓名、工作时数和薪水额。

这个简单的例子说明了多态性的一个重要方面：

> 当发送一个消息时，调用的方法的版本依赖于对象的类型，而不是指针的类型。

明确地说，在这个例子中，调用的readInto的版本依赖于对象的类型——Employee或HourlyEmployee，而不是指针的类型（即指向Employee的指针）。

A3.2 多态性的重要性

目前，读者可能还不是很信服。很容易就可以重新编写A3.1节中给出的例子以避免多态性，因此可能会置疑多态性是否有实际价值。答案无疑是肯定的。现在修改例子，用以说明多态性是如何使得有关继承的方法能够代码重用。我们将修改Company类中的findBestPaid()方法，使得用户可以选择是求薪水最高的雇员还是薪水最高的计时雇员。特别是将提示用户输入“Employee”或“Hourly”，指示下一个输入行的组成。

```
//前置条件: 确定薪水总额最高的雇员或计时雇员。忽略重复值。
void Company::findBestPaid( )
{
      string CODE_PROMPT =
"Enter Employee for Employee input or Hourly for hourly employee: ";

      string EMPLOYEE_INPUT = "Employee";

      string HOURLY_EMPLOYEE_INPUT = "Hourly";

      Employee* employeePtr;

      string code;

      cout << endl << CODE_PROMPT;
      cin >> code;
      if (code == EMPLOYEE_INPUT)
            employeePtr = new Employee;
      else if (code == HOURLY_EMPLOYEE_INPUT)
            employeePtr = new HourlyEmployee;

      while (employeePtr -> readInto( ))
            if (employeePtr -> makesMoreThan (bestPaid))
                  bestPaid.getCopyOf (*employeePtr);
} // findBestPaid
```

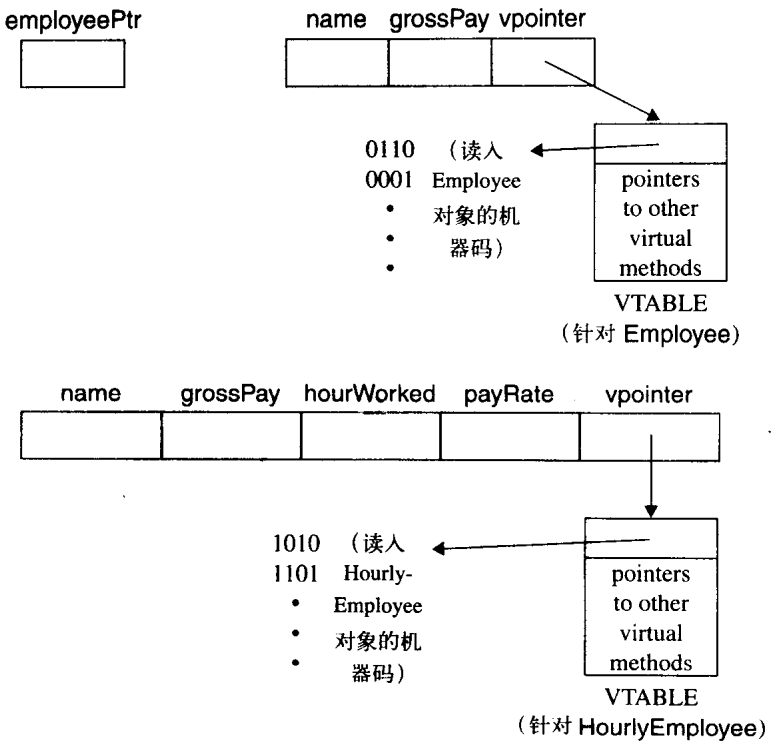
648

这个代码的优美之处在于不需要重新编写**while**循环，findBestPaid()方法就可以处理雇员输入或计时雇员输入。

A3.3 动态连接

A3.2节的代码提出了一个问题：C++编译器是怎样为类似employeePtr->readInto()的消息产生相应的机器码的？阐述这个问题的另一种方式是：当一些信息在运行之前不可用时，在编译时，方法标识符readInto是如何被**界定**在当前版本中的——在Employee或是HourlyEmployee里？答案很简单：连接不能在编译时完成，必须延迟直到运行时！这种延迟连接也称作是**动态连接**或是**迟连接**。

虚方法在项目上花费了运行时间代价。至少，编译器不能**嵌入**一个虚方法。使用嵌入，函数调用可以用函数体的代码替换；这节约了运行时间，因为绕过了通常函数调用的保存—调用—恢复—返回机制。而且，对使用虚方法的每个类，编译器创建了一个特殊的表VTABLE来保存虚方法代码的存储地址。并且为这样的类的每个对象给定了一个保密字段vpointer，它指向相应的VTABLE地址。下面的图说明了A3.2节末尾的示例中发生的事件：



649

在运行时，employeePtr将指向一个Employee对象或一个HourlyEmployee对象，并且将调用相应的readInto版本。

对虚方法而言，vpointer在运行时带来的额外工作是必需的，但对其他方法而言是浪费时间。为了提高效率，每个虚方法必须用关键字**virtual**在基类头文件中进行声明。因此Employee.h将有

```
virtual bool readInto();
```

这使得编译器能够对虚方法支持迟连接，对非虚方法提供稍快些的代码。最后是对基类开发者职责的影响，即在对应用或子类一无所知时判定方法是否应当是虚方法！它的结果是只有当基类版本中有关键字**virtual**时，方法才是虚的，并且重载的运算符不能是虚的。因此如果不是readInto，而是重载插入运算符**operator>>**，将不能应用多态性。

650

参考文献

A

Accredited Standards Committee X3, INFORMATION PROCESSING SYSTEMS, *Draft Proposed International Standard for Information Systems—Programming Language C++*, Washington, DC, 1997.

ACM/IEEE-CS Joint Curriculum Task Force, *Computing Curricula 1991*, Association for Computing Machinery, New York, 1991.

Adel'son-Vel'skii, G. M., and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Mathematics*, vol. 3, 1962, pp. 1259–1263.

Albir, S. S., *UML in a Nutshell*, O'Reilly & Associates, Inc., Sebastopol, CA, 1998.

Andersson, A., T. Hagerup, S. Nilsson, and R. Raman, "Sorting in Linear Time?" *Proceedings of the 27th* Annual ACM Symposium on the Theory of Computing, 1995.

B

Bayer, R., "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms," *Acta Informatica*, vol. 1, no. 4, 1972, pp. 290–306.

Bergin, J., *Data Structure Programming with the Standard Template Library in C++*, Springer-Verlag, New York, 1998.

Budd, T., *Data Structures in C++ Using the Standard Template Library*, Addison-Wesley Longman, Reading, MA, 1998.

C

Collins, W. J., *Data Structures and the Java Collections Framework*, McGraw-Hill, New York, 2002.

Cormen, T., C. Leierson, and R. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1992.

D

Dale, N., "If You Were Lost on a Desert Island, What One ADT Would You Like to Have with You?" *Proceedings of the Twenty-First SIGCSE Technical Symposium*, vol. 22, no. 1, 1990, pp. 139–142.

Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematick*, vol. 1, 1959, pp. 269–271.

Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

E

Eckel, B., *Thinking in C++*, Prentice-Hall, Englewood Cliffs, NJ, 1995.

G

Gries, D., *Science of Programming*, Springer-Verlag, New York, 1981.

Guibas, L., and R. Sedgewick, "A Diochromatic Framework for Balanced Trees", *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, 1978, pp. 8–21.

H

Heileman, G. L., *Data Structures, Algorithms, and Object-Oriented Programming*, McGraw-Hill, New York, 1996.

Hoare, C. A. R., "Quicksort," *Computer Journal*, vol. 5, no. 4, April 1962, pp. 10–15.

Horowitz, E., S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*, W. H. Freeman, New York, 1995.

Huffman, D. A., "A Model for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, vol. 40, 1952, pp. 1098–1101.

J

Jamsa, K., *Success with C++*, Boyd & Fraser, Danvers, MA, 1995.

K

Knuth, D. E., *The Art of Computer Programming*, vol. 1: Fundamental Algorithms, 2d ed., Addison-Wesley, Reading, MA, 1973. (1)

Knuth, D. E., *The Art of Computer Programming,* vol. 2: *Seminumerical Algorithms,* 2d ed., Addison-Wesley, Reading, MA, 1973. (2)

Knuth, D. E., *The Art of Computer Programming,* Vol. 3: *Sorting and Searching,* Addison-Wesley, Reading, MA, 1973. (3)

Kruse, R. L., *Data Structures and Program Design,* Prentice-Hall, Englewood Cliffs, NJ, 1987.

M

Meyer, B., *Object-oriented Software Construction,* Prentice-Hall International, London, 1988.

Meyers, S., *Effective STL,* Addison-Wesley, Boston, MA, 2001.

Musser, D., and A. Saini, *STL Tutorial and Reference Guide,* Addison-Wesley, Reading, MA, 1996.

N

Nelson, M., *C++ Programmer's Guide to the Standard Template Library,* IDG Books Worldwide, Foster City, CA, 1995.

Noonan, R. E., "An Object-Oriented View of Backtracking," *SIGCSE Bulletin,* vol. 32, no. 1, March 2000, pp. 362–366.

P

Pfleeger, S. L., *Software Engineering: Theory and Practice,* Prentice-Hall, Upper Saddle River, NJ, 1998.

Prim, R. C., "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal,* vol. 36, 1957, pp. 1389–1401.

R

Rawlins, G. J., *Compared to What? An Introduction to the Analysis of Algorithms,* Computer Science Press, New York, NY, 1992.

Riel, A. J., *Object-Oriented Design Heuristics,* Addison-Wesley, Reading, MA, 1996.

Roberts, S., *Thinking Recursively,* John Wiley & Sons, New York, 1986.

S

Sahni, S., *Data Structures, Algorithms, and Applications in Java,* McGraw-Hill, Burr Ridge, IL, 2000.

Savitch, W., *Problem Solving with C++: The Object of Programming,* 2d edition, Addison-Wesley Longman, Reading, MA, 1999.

Schaffer, R., and R. Sedgewick, "The Analysis of Heapsort," *Journal of Algorithms,* vol. 14, 1993, pp. 76–100.

Shaffer, C., *A Practical Introduction to Data Structures and Algorithm Analysis,* Prentice-Hall, Upper Saddle River, NJ, 1998.

Simmons, G. J., ed., *Contemporary Cryptology: The Science of Information Integrity,* IEEE Press, New York, 1992.

Sommerville, I., *Software Engineering,* 6th ed., Addison-Wesley, Reading, MA, 2001.

Stepanov, A. A., and M. Lee, *The Standard Template Library,* Technical Report HPL-94-34, April 1994, revised July 7, 1995.

W

Wallace, S. P., *Programming Web Graphics,* O'Reilly & Associates, Sebastopol, CA, 1999.

Weiss, M. A., *Data Structures and Problem Solving Using C++,* 2nd ed., Addison-Wesley Longman, Reading, MA, 2000.

Whitehead, A. N., and B. Russell, *Principia Mathematica,* Cambridge University Press, Cambridge, England, 1910 (volume 1), 1912 (volume 2), 1913 (volume 3).

Williams J. W., "Algorithm 232: Heapsort," *Communication of the ACM,* vol. 7, no. 6, 1964, pp. 347–348.

Wirth, N., *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood Cliffs, NJ, 1976.

索　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

A

Abstraction(抽象)
  data (数据),6-8
  principle of (……的原理),64
Access (访问)
  constant-time (常数时间),462
  protected,18-20
  random(随机),163,462
Activation records (活动记录),145,278
Active functions (活化函数),144-145
Acyclic undirected graphs (无环无向图),566
Adjacency-list design(邻接表设计),592,605
Adjacency matrix (邻接矩阵),605
  completing implementation of(完全实现),614
adjustPath method(adjustPath方法),368-370
Aggregation (聚合),69,71
Algorithms (算法),318-324
  backtracking (回溯),322,575
  binary searching (折半查找),125-135,515-517
  breadthFirst(广度优先),322-324
  decimal-to-binary (十进制到二进制的转换),99-102
  divide and conquer (分治法),500
  factorials (阶乘),94-99
  generic,in container classes (容器类的通用型算法),58
  graph (图),571-587
  greedy (贪心算法),468,611-612
  inOrder (中序),318-320
  maze searching (迷宫搜索),152,304-305
  permutations (置换),142-144
  postOrder (后序),320-321
  preOrder (前序),321-322
  run-times for sorting (排序的运行时间),500
  sorting (排序),501
  Towers of Hanoi (汉诺塔),103-111,154-156,631
  tree_sort,483-484
Allocator parameters (Allocator参数),165
American National Standards Institute (美国国家标准化组织),164
amortizedTime(n),179-180,197-198
Ancestor classes (祖先类),18,368
Application class (应用类),114-115
Applications (应用)
  of AVL trees,a simple spell- checker (AVL树的应用,一个简单的拼写检查器),380-383,389-390
  of deques,very long integers (双端队列的应用,非常长的整数) ,198
  of hash_map class (hash_map类的应用),538-539
  of lists,a line editor,(链表的应用,一个行编辑器),228-240,244-245
  of priority queues,Huffman codes (优先队列的应用, 霍夫曼编码),456-468
  of recursion,a maze (递归的应用,一个迷宫),117-125
  of sets,spell-checker,revisted (集合的应用,又一次构造拼写检查器),425-431
  of stacks,converting infix to postfix (堆栈的应用,将中缀变换成后缀),285-295
  of stacks,how recursion is implemented (堆栈的应用, 递归是如何实现的),277-285
  of vectors,high-precision arithmetic (向量的应用,高精度算法),184-190
Arguments,template (变元,模板),44
Array buckets (桶数组),521-522,534
Arrays (数组),41-42
  associative (关联),428
  map (映射),193
  and pointers (和指针),39-40
Associative arrays (关联数组),428
Associative containers (关联容器类),325
  in the Standard Template Library (在标准模板库中),422-425
Automatic type conversion (自动类型转换),217
Average height,of a BinSearchTree (折半查找树的平均高度),345
averageSpace(n),74
averageTime(n),74,197,334,342,514
AVL tree application,a simple spell-checker (AVL树的应用, 一个简单的拼写检查器),380-383

AVL trees (AVL树),353-390
AVLTree class (AVLTree类),364-367
balanced binary search trees (平衡折半查找树),354
correctness of the insert method (插入方法的正确性),377-380
enhancing the SpellChecker project (改进SpellChecker项目),389-390
the erase method in the AVLTree class (AVLTree类中的erase方法),388-389
the fixAfterInsertion method (fixAfterInsertion方法),367-377
function objects (函数对象),361-364
height of (……的高度),360-361
rotations (旋转),354-358

B

BackTrack class (BackTrack类),113,115
Backtracking application,a maze (回溯应用，一个迷宫),117-125
Backtracking strategy (回溯策略),322,575
through a network (通过网络),615-617
Balanced binary search trees,in AVL trees (平衡折半查找树，AVL树),354
balanceFactor field (平衡因子字段),366-375
Base,of logarithms(对数的底数),622
Base classes (基类),18
Bidirectional iterators (双向迭代器),206
Big-O notation (大O表示法),166
getting estimates quickly (快速估算),77-81
in program implementation (在程序的实现中),74-77
smallest upper bound in (最小上界),76
upper bound in (上界),74,82
binary_search algorithm (binary_search算法),126-134,152
Binary search trees (折半查找树),307-351
the average height of (平均高度),345
balanced,in AVL trees (平衡的,在AVL树中),354
BinSearchTree class (BinSearchTree类),307,325-327
BinSearchTree iterators (BinSearchTree迭代器),342-345
fields and implementation of (字段和实现),329-334,350-351
Iterator class for (迭代类),327-329
recursive methods for (递归方法),334-342
Binary searching (折半查找),126,515-517
iterative,in recursion (迭代的,在递归中),134
in recursion (在递归中),125-135
Binary Tree Theorem (二叉树定理),315-316,345,347-348,386
Binary trees (二叉树)
binary tree theorem (二叉树定理),314-317
definition and properties (定义和属性),308-324
external path length (外部路径长度),317-318
traversals of (遍历),318-324
Binding,dynamic (连接,动态的),649-650
Black-height(黑色高度),397
Bottom-up testing (自底向上测试),72
Boundary values (边界值),71
Branches (树枝),308
Breadth-first iterators (广度优先迭代器),571-574
breadthFirst traversal,level-by-level (广度优先遍历,逐层地),322-324
Bubble Sort (冒泡排序),505-506
Buckets (桶),524
array (数组),521-522,534

C

C++ language (C++语言),164,214,362,422,464
classes in (C++中的类),1-33
Calling object (调用对象),3
Car wash simulation (洗车处仿真),264-274
analysis of the CarWash methods (CarWash方法的分析),272
extending (扩展),298-300
implementation of the CarWash class (CarWash类的实现),268-272
program design (程序设计),266-268
randomizing the arrival times (随机化到达时间),272-274
Casting,in program implementation (类型转换，在程序实现中),86-87
Chained hashing (链式散列),524,535-537
Chaining (链接)
in building a symbol table (在构造一个符号表中),561
in hash classes (在散列类中),522-530
Chains (链),316-317
char (字符),186,188,633,644
casting to int (类型转换成整型),87
Children,deleted node with (子节点,删除有子女的节点),410-421
Class declarations (类的声明),12,14
Classes (类),1-33。参阅Ancestor classes; Base classes; Derived classes; Desendant classes; Subclasses; Superclasses
Application (应用),114-115
AVLTree,364-367

BackTrack,113,115
BinSearchTree,325-327
CarWash,implementation of (CarWash类的实现),268-272
Company,17,68
constructors for (……的构造器),8-10
container (容器),42-59
data abstraction (数据抽象),6-8
deque (双端队列),192-198
Editor,232-240
Employee,10-17,67-70,647-648
friends in (友元),26-27
hash_map,517-539
hash_set,540
HourlyEmployee,20-23,647-648
Huffman (霍夫曼),461-468
information hiding in (信息隐藏),28
inheritance in (继承),17-18,23
instances of (实例),2
Iterator(迭代器),50-52,113,530-531
Linked (链式),44-48,91
LinkedDriver,91
list(链表),205-251
map (映射),427-431
method interfaces (方法接口),2-3
multimap (多映射),431
multiset (多集合),427
network (网络),588-607
and objects (和对象),3-6
operator overloading in (运算符的重载),25-26
overloading operator= and operator >> in (重载运算符=和运算符>>),27
priority_queue,440-444,454-455
protected access to (**protected**访问),18-20
queue (队列),255-258,305
rb_tree,399-408
Sequence(序列),33
set (集合),427
SimpleClass,30-32
stack (堆栈),274-277
stream (流),24
string (字符串),633-645
struct (结构),14,46,123
value_type,537-538
vector (向量),166-174,177-184
very_long_int,185-190
Clusters (聚类),549
Codomains (值域),619
Collision handler,quotient-offset (冲突处理器,商偏移),550
Collisions (冲突),521
command_check method (command_check方法),232,235
Company class (Company类),17,68
Company method (Company方法),17
Company project (Company项目),17
Compare method (compare方法),383
Compiler directives (编译器指令),13
Complete tree (完全树),312
Complete undirected graphs (完全无向图),564
Composition (复合),69-70
Computer simulation,with queues (使用队列进行计算机仿真),261-264
Concordance, building (创建词汇索引), 437-438
Conditional operators (条件运算符),183
Connected graphs (连通图)
directed (有向的),567
undirected (无向的),566,578
Connectedness,in graph algorithms (在图算法中的连通性),578-579
Constant reference parameters (常量引用参数),14
Constant-time access (常数时间访问),462
Constructor headings (构造器头),216
constructor-initializer section (构造器初始化部分),216
field initializations (字段初始化),216
Constructors (构造器),8-10
copy (拷贝),167,209
default (缺省),9-10,14
public,51
in string class (在字符串类中),634-635
zero-parameter (0-参数),9
Container adaptors (容器配接器)
circular (环型),260
in queues(队列中的……),259-260
stack class as (堆栈类作为……),276-277
Container classes (容器类),42-59
in data structures (数据结构中),58-59
defining the other iterator operators (定义其他的迭代器运算符),52
design and implementation of the iterator class (迭代器类的设计与实现),50-52
destructors (析构器),53-54
generic algorithms (通用型算法),54-58
iterators (迭代器),48-50
linked structures (链式结构),44-48
overloading operator= (重载运算符=),54
pop_front method (pop_front方法),52-53

in the Standard Template Library (在标准模板库中),58-59
storage structures for (存储结构),44
Containers (容器),42
associative, in the Standard Template Library (在标准模板库里的关联性),422-425
vectors compared to other (向量与其他容器的对比),176-177
Contiguous design,for queues (队列的连续设计),260-261
Contract,between developer and user (开发者和用户间的合约),8,68
Conveting infix to postfix application (将中缀转换成后缀的应用),285-295
converting from infix to postfix (将中缀转换成后缀),291
postfix notation (后缀表示法),286-289
prefix notation (前缀表示法),292-295
tokens (记号),290-291
transition matrix (转换矩阵),289-290
Copy constructors (Copy构造器),167,209
Copy function (Copy函数),93
Corollaries, Vector-Iterator (向量-迭代器推论),174
Correctness (正确性)
of the insert method in AVL trees (AVL树中插入方法的……),377-380
in program implementation,feasibility of (在程序实现中,……的可行性),73
Cost, of recursion (递归的代价),145-146
Cycles (回路),566

D

Data Abstraction, principle of (数据抽象原理),6-8,27-28,48-49,82
Data structures (数据结构)
in container classes (在容器类中),58-59
implementation of (……的实现),59
Date class (Date类),2-7
Deallocation, of dynamic variables (动态变量的存储单元释放),40-41
Decimal-to-binary recursion (十进制到二进制的递归转换),99-102
Fibonacci numbers (斐波纳契数),102
Decision trees (决策树),482
Declarations (声明)
class (类),12,14
struct (结构),14
Decoding a message with priority queues (使用优先队列解码消息),473-475
Default constructors (缺省构造器),9-10,14
delete_command method (delete_command方法),236-237
Deleted node (删除节点)
a leaf (树叶),409-410
a leaf or with two children (一个树叶或是有两个子女的节点),411-421
with two children (有两个子女的节点),410-411
deleteLink method (deleteLink方法),337-341,349
delete operator (delete运算符),40-41,52
Dense networks (密集网),607
Dependency diagrams (依赖关系图),67-71
Depth (深度),311
Depth-first iterators (深度优先迭代器),574
Depth-first search (深度优先搜索),322
Deque application, very long integers (双端队列的应用，非常长的整数),198
Deque class (Deque类)
fields and implementation of (……的字段和实现),192-198
Hewlett-Packard's (惠普的),198
Deque methods, differences between list methods and (Deque方法，与list方法之间的区别),213-214
Deques (双端队列),163-204
alternative implementation of (……的另一种实现),204
in the Standard Template Library (在标准模板库中),164
Dereference-and-select operator (脱引用和选择运算符),37
Dereference operator (脱引用运算符),37,58
Derived classes (派生类),18
Descendant classes (子孙类),18
Destructors, in container classes (容器类中的析构器),53-54
Developer, contract with user (开发者和用户之间的合约),8,68
Developmental life cycle, in software engineering (软件工程中的开发生命周期),64
Directed graphs (digraphs) (有向图),567-568
connected (连通的),567
undirected (无向的),568
Directed trees (有向树),568
Directives(指令)
compiler (编译器),13
using (使用),13-14
Divide and conquer algorithms (分治法),500
Domains (定义域),619
Double hashing (双散列),550-554
in building a symbol table (在构造一个符号表时),561
Double rotation (双旋转),357,386
Drivers (驱动器),72

in program implementation (在程序实现中),72
Dynamic binding (动态连接),649-650
Dynamic-variable assignments, versus pointer-variable assignments (动态变量赋值与指针变量赋值对比),40
Dynamic variables (动态变量),36
deallocation of (……的存储单元释放),40-41

E

Edge-related methods, implementation of (Edge-related方法的实现),594-596
Edges (边),563-568
Editor class (Editor类)
design of (……的设计),232-234
extending (扩展),244-251
implementation of (……的实现),234-240
Efficiency of methods, estimating, in program (程序中方法的效率的估算)
implementation (实现),74
Eight-queens problem (八皇后问题),156-158
Element, 619
Employee class (Employee类),10-17,67-70,647-648
definition of (……的定义),15-17
Encoding (编码)
Huffman (霍夫曼),457
prefix-free (无前缀),456
equalTo method (equalTo方法),29
erase method (erase方法),349
in the AVLTree class (在AVLTree类中),388-389
in hash classes (在散列类中),543-548
erase method in red-black trees (红黑树中的erase方法),408-421
if the deleted node had two children (如果被删除的节点有两个子女),410-411
if the deleted node was a leaf (如果被删除的节点是一个树叶),409-410
if the deleted node was a leaf or had two children (如果被删除的节点是一个树叶或有两个子女),411-421
Evaluating a condition, with queues (使用队列求条件的值),300-304
Event-driven simulations (事件驱动仿真),266
Execution frames (执行结构框架)
for factorials (阶乘的……),96-99
in recursion (递归的……),96-99
explicit,255
Exponential-time method (指数级时间方法),80
External path length, in binary trees (二叉树的外部路径长度),317-318
External Path Length Theorem (外部路径长度定理),345,483

F

Factorials (阶乘),94-99
execution frames for (……的执行结构框架),96-99
in recursion (递归的……),94-99
Stirling's approximation of (……的Stirling的近似值),453
Fairness, incorporating in priority queues (优先队列的公平合并),454
Fast sorts (快速排序),483-500
Feedback (反馈),263
Fibonacci numbers (斐波纳契数),627-628
in recursion (在递归中),102
Fibonacci tree (斐波纳契树),360
Field initializations, in constructor headings (构造器头中字段的初始化),216
Fields (字段),2
balanceFactor (平衡因子),366-375
in the BinSearchTree class (在BinSearchTree类中),329-334
in the deque class (在双端队列类中),192-198
different forms of (……的不同形式),6-7
in the hash_map class (在hash_map类中),519
in the iterator class, 530-531
in the list class (在链表类中),214-221
in the network class (在网络类中),591-593
object (对象),9
in the priority_queue class (在priority_queue类中),443-444
private,15,18-19,423
in program design (在程序设计中),67
protected,26
public,26
of the vector class, possible (向量类的可能的字段),177
FIFO (先入先出)。参阅First in, first out
Files (文件)
header (头文件),12
sorting (排序),509-511
source (源文件),15
find method (find方法),351
Finite sequences (有限序列),620
First component (第一组件),619
First in, first out(FIFO), 254
fixAfterInsertion method, in AVL trees (AVL树中的fixAfterInsertion方法),367-377
float (浮点),25
Forward-iterators (前向迭代器),176

Friends, in classes (类的友元),26-28,539
Full tree (满树),311
Function objects(functors),in AVL trees (AVL树中的函数对象(函子)),361-364
Functions (函数),619
  active (活化),144-145
  mathematical (数学的),619-620
  recursive (递归的),145
Functors(函子)。参阅Function objects

G

Garbage (无用单元),40
generateInitialState method (generateInitialState方法),116
Generating minimum spanning trees, with graph algorithms (利用图算法求最小生成树),579-584
Generating permutations (生成置换),135-144
  estimating time and space requirements (估算时间代价和存储代价),142-144
  in recursion (递归地……),135-144
Generic algorithms, in container classes (容器类的通用型算法),54-58
getCopyOf method (getCopyOf方法),11,14,23
getlinefunction, 238
get_minimum_spanning_tree method (get_minimum_spanning_tree方法),599-601
get_shortest_path method (get_shortest_path方法),601-603
Global methods, implementation of (全局方法的实现),596-599
Graph algorithms (图算法),571-587
  connectedness in (连通性),578-579
  finding the shortest path through a network (在网络中寻找最短路),584-587
  generating a minimum spanning tree (生成一个最小生成树),579-584
  iterators in (迭代器),571-578
Graphs (图),563-617
  and backtracking through a network (和网络中的回溯),607-610,615-617
  completing the adjacency-matrix implementation (完成邻接矩阵),614
  directed (有向的),567-568
  and trees (和树),568-569
  undirected (无向的),564-566
  weighted (带权的),569
Greedy algorithms (贪心算法),468,611-612

H

Hash classes(散列类)。参阅open-address hashing
  applications of (……的应用),538-539
  chaining (链式),522-530
  double hashing (双散列),550-554
  erase method (erase方法),543-548
  fields and implementation of the iterator class (字段及其迭代器类的实现),530-531
  primary clustering (主聚类),549-550
run-time comparison of chaining and double hashing in building a symbol table (通过构造一个符号表,比较链式和双散列的运行时间),561
the value_type class (value_type类),537-538
Hashing (散列),519-522
  chained, analysis of (分析链式散列),535-537
hash_map class (hash_map类),517-539
  analysis of chained hashing (分析链式散列),535-537
  fields in (……中的字段),519
  implementation of (……的实现),532-535
  timing (计时),539
hash_set class (hash_set类),540
Header files (头文件),12
Header node (头节点),215
Heap Sort (堆排序),485-487
  analysis of (……的分析),486-487
  example of (……的示例),485-486
Heaps (堆),36,444-454
  versus stacks (和堆栈比较),37-38
Height (高度)
  of an AVL tree (AVL树的……),360-361
  of a BinSearchTree (折半查找树的……),310,345
  of a red-black tree (红黑树的……),394-399
Helper methods (辅助方法),189
Hewlett-Packard(惠普的)
  deque class (双端队列类),198
  rb_tree class (rb_tree类), 399
Hiding information (信息隐藏),28
High-precision arithmetic (高精度算法),184-190
  design of the very_long_int class (very_long_int类的设计),185-187
  expanding the very_long_int class (扩展very_long_int类),190
  implementation of the very_long_int class (very_long_int类的实现),187-190
hint iterators (提示迭代器),431
Holes (陷阱),446
HourlyEmployee class (HourlyEmployee类),20-23,647-648
Huffman codes application (霍夫曼编码的应用),456-468
  design of the huffman class (huffman类的设计),461-463

implementation of the huffman class (huffman类的实现),463-468
Huffman encoding (霍夫曼编码),457
Huffman trees (霍夫曼树),458

I

Implementation (实现)
of the BinSearchTree class (BinSearchTree类的……),329-334,350-351
of data structures (数据结构的……),59
of the deque class (双端队列类的……),192-198
of deques (双端队列的……),204
of the hash_map class (hash_map类的……),532-535
of the iterator class (迭代器类的……),530-531
of the iterator class in container classes (容器类中的迭代器类的……),50-52
of the list class (链表类的……),214-221
of the network class(网络类的……),593-594
of the priority_queue class (优先队列类的……),443-444,454-455
of recursion (递归的……),277-285
of the string class (字符串类的……),644-645
of the vector class (向量类的……),177-184
Incorrect versions (错误版本),147-148
Increment operator (增量运算符),58
Indirect recursion (间接递归),144-145
Induction, mathematical (数学归纳法),623-631
Infinite recursion (无穷递归),96
Infix, converting to postfix (中缀转换成后缀),291
Information hiding (信息隐藏),28
Inheritance (继承),17-18,23
multiple (多重),24
Inlining (嵌入),649
inOrder traversal, left-root-right (左-根-右，中序遍历),318-320,325
Insertion Sort (插入排序),478-481
analysis of (……的分析),481
example of (……的示例),480-481
Insert method (插入方法)
in AVL trees, correctness of (在AVL树中的正确性),377-380
in rb_tree class (在rb_tree类中),399-408,483
Instances, of a class (一个类的实例),2
Integers, very long (非常长的整数),184,198
Interface methods (方法接口),2-3,10-12
for the list class (列表类的),207-211
for the queue class (队列类的),255-257
for the stack class (堆栈类的),274-275
for the vector class (向量类的),166-174
int variables (整型变量)
casting to char (转换成字符型),86,641
unsigned (无符号的),85,136,170,183,242,633-635,640,644
Intractable problems (棘手的问题),80-81
isValid method (isValid方法),2,7,29
Iterative binary search, in recursion (迭代折半查找，递归的),134
Iterative functions (迭代函数),98
Iterative maze search, with queues (使用队列的迭代迷宫搜索),304-305
Iterative version, of the Towers of Hanoi game (汉诺塔游戏的迭代版本),154-156
Iterator class (迭代器类),113
in the BinSearchTree class (在BinSearchTree类中),327-329
in container classes (在容器类中),50-52
Iterator-equality tests (迭代器等同性测试),58
Iterator interfaces, in lists (链表中的迭代器接口),211-213
Iterator operators, defining, in container classes (容器类中迭代器运算符的定义),52
Iterators (迭代器),49,60,571-578
bidirectional (双向的),206
in BinSearchTree (在折半查找树中),342-345
breadth-first (广度优先),571-574
in container classes (在容器类中),48-50
depth-first (深度优先),574
in graph algorithms (在图算法中),571-578
hint (提示),431
in lists (在列表中),225
random-access (随机存取),478
start(start迭代器),197
vector (向量),174-176

K

Keywords(关键字)
class(类),14,45
const(常量),14
friend(友元),26
inline(嵌入),55
operator(运算符),25
public,14
template(模板),45,55
virtual(虚),650

Knight's tour problem(马的遍历问题),158-161

L

Labs(实验)
average height of a BinSearchTree(折半查找树的平均高度),345
call to erase(调用erase),421
Company project(Company项目),17
converting from infix to postfix(将中缀转换成后缀),291
drivers(驱动器),72
dynamic-variable assignments versus pointer-variable assignments(动态变量赋值与指针变量赋值对比),40
Fibonacci numbers(斐波纳契数),102
function objects(函数对象),364
generic algorithms(通用型算法),58
Hewlett-Packard's deque class, details of(惠普的双端队列的详细资料),198
inheritance, details of(继承的详细资料),23
iterative binary search(迭代折半查找),134
iterator operators, defining(定义迭代器运算符),52
iterators(迭代器),225
list class, implementation details for(链表类的实现细节),224
map classes(映射类),431
multimap classes(多映射类),431
multiset classes(多集合类),427
overloading operator=(重载运算符=),54
overloading operator= and operator >>(重载运算符=和运算符>>),27
pointer-variable assignments versus dynamic-variable assignments(指针变量赋值与动态变量赋值的对比),40
priority queues, incorporating fairness in(优先队列的公平合并),454
randomizing(随机化),274
a red-black tree insertion(红黑树的插入),408
run-times for sorting algorithms(排序算法的运行时间),500
set classes(集合类),427
timing a hash_map(计时一个hash_map),539
timing and randomness(计时和随机),86
timing the sequence containers(计时顺序容器),224
traveling salesperson problem(货郎担问题),604
vector class, implementation details for(向量类的实现细节),184
very_long_int class,expanding(扩展very_long_int类),190
Late binding(迟连接),649-650
Laws(定理)。参阅Corollaries;Principles;Rules
Moore's(摩尔定理),63
Universal Array-Pointer(通用数组指针定理),40
least method(least方法),189
Leaves(树叶),308
deleted node as(删除叶节点),409-421
minimum number of(最小数量),318
Left child(左子女),310
Left rotation(左旋转),354-356,384
Length of sequence(序列长度),620
Lexicographic order(词典顺序),144
Life cycle, of software development (软件开发生命周期),64
Line editor application(行编辑器应用),228-240
design of the Editor class(Editor类的设计),232-234
implementation of the Editor class(Editor类的实现),234-240
Linked class(链式类),44-48
expansion of(……的扩展),91
Linked container(链式容器类),48-49
Linked lists(链表),206,522
Linked structures, in container classes(容器类的链式结构),44-48
LinkedDriver class(LinkedDriver类),91
List application, a line editor(链表的应用,一个行编辑器),228-240,244-251
List methods(链表的方法),208
Lists(链表),205-251
alternative implementations of the list class(链表类的另一种实现),226-228
an alternate design and implementation of the list class(链表类的一个候选设计和实现),251
differences between list methods and vector or deque methods(链表方法和向量或双端队列方法之间的区别),213-214
fields and implementation of the list class(链表类的字段及实现),214-221
iterator interfaces(迭代器接口),211-213
iterators in(……中的迭代器),225
method interfaces for the list class(链表类的方法接口),207-211
storage of list nodes(链表节点的存储),221-224
timing the sequence containers(计时顺序容器),224
Load factors(加载因子),529
Logarithms(对数),621-622
base of(……的底数),622
natural(自然对数),622

M

make_heap,486-487
makesMoreThan method(makeMoreThan方法),14,25
Map arrays(映射数组),193
Map class(映射类),427-431
Maps(映射),620
Mathematical background(数学背景),619-632
  functions and sequences(函数和序列),619-620
  induction and recursion(归纳和递归),630-631
  logarithms(对数),621-622
  mathematical induction(数学归纳法),623-631
  sums and products(累加和累乘),620-621
Mathematical Induction, Principle of(数学归纳法的原理),315,623-630
Mathematical models(数学模型),263
Matrix, transition(转换矩阵),289-290
Maze searching(迷宫搜索),152
  iterative, with queues(使用队列迭代),304-305
Mean arrival time(平均到达时间),272
Member selection operator(成员选择运算符),37
Memory leaks(内存泄漏),40,297
Merge sort(归并排序),487-493
  analysis of(……的分析),492-493
  example of(……的示例),490-492
Messages(消息),2
  decoding, with priority queues(使用优先队列解码),473-475
Method declarations(方法声明),2
Method identifiers,readInto(方法标识符,readInto),25
Method interfaces(方法接口),2-3,10
  for the list class(链表类的),207-211
  in program design(在程序设计中),67
  for the queue class(队列类的),255-257
  for the stack class(堆栈类的),274-275
  for the vector class(向量类的),166-174
Methods(方法),2。参阅Insert method
  adjustPath,368-370
  CarWash, analysis of(CarWash方法的分析),272
  command_check,232,235
  Company,17
  compare,383
  delete_command,236-237
  deleteLink,337-341,349
  edge-related, implementation of(与边相关的方法的实现),594-596
  equalTo,29
  erase,349,388-389
  exponential-time,80
  find,351
  fixAfterInsertion, in AVL trees(AVL树中的fixAfterInsertion),367-377
  generateInitialState,116
  getCopyOf,11,14,23
  get_minimum_spanning_tree,599-601
  get_shortest_path,601-603
  global,implementation of(global的实现),596-599
  helper(辅助方法),189
  insert. in AVL trees(AVL树中的insert方法),377-380
  isValid,2,7,29
  least,189
  list,208
  makesMoreThan,14,25
  overriding(方法的覆盖),18
  parse,234
  polynomial-time(多项式时间方法),80
  pop_back,91,169,220-221,226
  pop_front,52-53,91,191,194-195,220-221,260,305
  protected(受保护方法),269
  push_back,174,179-180,194,200,219-221,223,260-261,305
  push_front,48-49,60,191,198,219-221,223
  recursive, in binary search trees(折半查找树中的递归),334-342
  tryToSolve,115-116,124,283,304
  virtual(虚方法),649
Method validation, in program implementation(程序实现中的方法验证),71-72
Minimum spanning tree, generating with graph algorithms(由图算法产生的最小生成树),579-584
Moore's Law(摩尔定理),63
Multimap class(多映射类),431
Multiple inheritance(多重继承),24
Multiset class(多集合类),427
myDate object(myDate对象),3

N

Natural logarithms(自然对数),622
Neighbors(邻居),565
Network class(网络类),588-607
  alternative design and implementation of(……的另一种设计和实现),605-607
  developing(开发),588
  fields in(……中的字段),591-593

get_minimum_spanning_tree method(最小生成树方法),599-601
get_shortest_path method(最短路方法),601-603
implementation of(……的实现),593-594
implementation of edge-related methods in(edge-related方法的实现),594-596
implementation of global methods in(global方法的实现),596-599
time estimates for the network methods(网络方法的时间花费估算),604
traveling salesperson problem(货郎担问题),604
Networks(网络),563-617
backtracking through(回溯通过网络),607-610,615-617
completing the adjacency-matrix implementation(完整邻接矩阵实现),614
dense(密集),607
and directed graphs(和有向图),567-568
finding shortest path through(寻找最短通路),584-587
trees in(……中的树),568-569
and undirected graphs(和无向图),564-566
new operator(new运算符),36-37,40-41,219
Nodes(节点),46-48
deleted(删除),409-421
header(头),215
list, storage of(链表存储),221-224
Nonmember functions, in string class(字符串类的非成员函数),635-638

O

Object fields(对象字段),9
Objects(对象),3-6
calling(调用),3
function, in AVL trees(AVL树中的函数),361-364
OperatorStack,292-294
Out-of-scope(超出作用域),53
Open-address hashing(开放地址散列),540-557
analysis of(……的分析),554-557
double hashing(双散列),550-554
erase method(erase方法),543-548
primary clustering(主聚类),549-550
Open-Closed Principle(开放－封闭原理),18
operator!=(运算符!=),49,58,251,327,519
operator()(运算符()),362-363,399-400,423,430,441,522
operator*(运算符*),49,58,251,327,343,519,599
operator+(运算符+),187-190,198
operator++(运算符++),49,51,58,157,251,327,342,519,536-537,573,577,598
operator<(运算符<),126,325,362,364,387,400,478,493
operator<<(运算符<<),24,26,187,190,464
overloading(重载),27
operator<?(运算符<?),363
operator=(运算符=),54,62
overloading(重载),27,54
operator= =(运算符= =),49,58,62,326-327,343-344,518-519,537,599
operator>(运算符>),25,363,462,592,600
overloading(重载),27
operator>>(运算符>>),24-25,187,238,650
overloading(重载),27
operator[ ](运算符[ ]),40,44,174,176,205,213,427,430,518,535,644
operator--(运算符--),61,327,433
operator::(运算符::),588
Operators(运算符)
conditional(条件),183
delete(删除),40-41,52
dereference(脱引用),37
dereference-and-select(脱引用和选择),37
member selection(成员选择),37
new,36-37,40-41,219
overloading(重载),25-27,54
postincrement(后加),211
preincrement(前加),211
scope-resolution(作用域解析),16
in string class(在字符串类中),635
operatorStack objects(operatorStack对象),292-294
out-of-scope objects(超出作用域的对象),53
Overloading operator=(重载运算符=),27
in container classes(在容器类中),54
Overloading operator>>(重载运算符>>),27
Overriding methods(覆盖方法),18

P

Palindromes(回文),149
Parameters(参数)
Allocator(分配器),165
constant reference(常量引用),14
reference(引用),38-39
template(模板),45
Parents(父节点),310
Parse method(parse方法),234

Partitioning, in Quick Sort(快速排序的分割),494-497
Path length, external, for binary trees(二叉树的外部路径长度),317-318
Path Rule(路径规则),392-393,410,413,418-419,433
Paths(路径) 310, 392, 565
  cycles in(回路),566
  length of(长度),566
Performance specifications(性能规格说明),81
Permutations, estimating time and space requirements for generating(估算生成置换的时间代价和存储代价),142-144
Pointer fields(指针字段),39
Pointer-variable assignments, versus dynamic-variable assignments(指针变量赋值与动态变量赋值的对比),40
Pointer variables(指针变量),36
Pointers(指针),36-41
  arrays and(数组和),39-40
  deallocation of dynamic variables(动态变量的存储单元释放),40-41
  the heap versus the stack(堆和堆栈的对比),37-38
  reference parameters(引用参数),38-39
  void(空),122-123
Poisson distribution(泊松分布),272
Polymorphism(多态性),647-650
  dynamic binding(动态连接),649-650
  importance of(……的重要性),648-649
Polynomial-time method(多项式时间方法),80
pop_back method(pop_back方法),91,169,220-221,226
pop_front method(pop_front方法),91,191,194-195,220-221,260,305
  in container classes(在容器类中),52-53
pop_heap,445,448-452,486
Postconditions(后置条件),2
Postfix(后缀)
  converting from infix(中缀转换成后缀),291
  notation for(表示法),286-289,296-297
Postincrement operators(后加运算符),211
postOrder traversal, left-right-root(左-右-根,后序遍历),320-321
Preconditions(前置条件),2-3
Prefix, notation for(前缀表示法),292-295
Prefix-free encoding(无前缀编码),456
Preincrement operators(前加运算符),211
preOrder traversal, root-left-right(根-左-右,前序遍历),321-322
Prepending(添加),466
Principles(原理)。参阅Corollaries;Laws;Rules
  Abstraction(抽象),64
  Data Abstraction(数据抽象),7-8,27-28,48-49,82
  Information Hiding(信息隐藏),28
  Mathematical Induction(数学归纳),315,623-630
  Open-Closed(开放－封闭),18
Priority queue application, Huffman codes(优先队列应用,霍夫曼编码),456-468
priority_queue class(priority_queue类),440-443
  alternative designs and implementations of(……的另一种设计与实现),454-455
  fields and implementation of(……的字段与实现),443-444
Priority queues(优先队列),439-475
  decoding a message(解码一个消息),473-475
  and heaps(和堆),444-454
  incorporating fairness in(公平合并),454
private fields(**private**字段),15,18-19,423
private level of protection(**private**级保护),15,18-20,28
Problem analysis(问题分析),64-65
  system tests(系统测试),66
Problems, intractable(棘手的问题),80-81
Production programs(成品程序),87
Products(累乘),620-621
Program design(程序设计),67-71
  dependency diagrams(依赖关系图),67-71
  method interfaces and fields(方法接口和字段),67
  for the simulated car wash(洗车处仿真),266-268
Program implementation(程序实现),71-87
  Big-O notation in(大O表示法),74-77
  casting in(强制转换),86-87
  drivers in(驱动器),72
  estimating the efficiency of methods(估算方法的效率),74
  feasibility of correctness(正确性实现的可行性),73
  getting Big-O estimates quickly(快速获取大O估算),77-81
  method validation in(方法验证),71-72
  randomness in(随机性),84-86
  run-time analysis of(运行时间分析),83-84
  timing and randomness in(计时与随机性),86
  trade-offs in(平衡折中),81-83
Program maintenance(程序维护),87-88
Programming projects(编程项目)
  backtracking through a network(回溯通过一个网络),615-617
  BinSearchTree class, alternative implementation of(BinSearchTree类的另一种实现),350-351
  building a concordance(创建一个词汇索引),437-438
  car wash simulation, extending(扩展洗车仿真),298-300
  completing the adjacency-matrix implementation(完成邻

接矩阵的实现),614
decoding a message(解码一个消息),473-475
deque class, alternative implementation of(deque类的另一种实现),204
Editor class, extending(扩展Editor类),244-251
eight-queens problem(八皇后问题),156-158
erase method in the AVLTree class(AVLTree类的erase方法),388-389
evaluating a condition(求一个条件的值),300-304
iterative maze search(迭代的迷宫搜索),304-305
iterative version of Towers of Hanoi game(汉诺塔游戏的迭代版本),154-156
knight's tour(马的遍历),158-161
Linked class(Linked类)
expansion of(……的扩充),91
extending(扩展),62
list class, alternate design and implementation of(List类的另一种设计和实现),251
queue class, alternate design of(queue类的另一个设计),305
run-time comparison of chaining and double hashing in building a symbol table(链式和双散列在构造一个符号表中的运行时间比较),561
Sequence class(Sequence类),33
simple thesaurus(简单辞典),436-437
sorting a file(排序一个文件),509-511
SpellChecker project, enhancing(改进的SpellChecker项目),389-390
very_long_int class, extending(very_long_int类的扩展),203-204
Programs(程序)
production(成品),87
robust(健壮),65
protected access(**protected**访问),18-20
protected fields(**protected**字段),26
protected methods(**protected**方法),269
protection(保护)
private level of(**private**级的),15,18-20,26,28
protected level of(**protected**级的),19-20,26-28,216,332,487
public level of(**public**级的),16,20,26,28,184,216
public,20
public constructors(**public**构造器),51
public fields(**public**字段),26,28
public level of protection(**public**级保护),16,20,184,216
in Nodes(在Nodes中的),46-48
push_back method(push_back方法),174,179-180,187,194,200,219-221,223,260,305
push_front method(push_front方法),48-49,60,191,198,219-221,223
push_heap,445-448,450-451,453

Q

Queue application, a simulated car wash(队列应用,一个仿真洗车程序),264-274,298-300
Queue class(队列类)
alternate design of(另一个设计),305
method interfaces for(方法接口),255-257
using(使用),257-258
Queues(队列),254-261
computer simulation with(计算机仿真),261-264
container adaptors in(容器配接器),259-260
a contiguous design for(一个接近的设计),260-261
evaluating a condition with(求一个条件的值),300-304
iterative maze search with(迭代的迷宫搜索),304-305
priority in(优先级),439-475
and stacks(和堆栈),274-277
Quick Sort(快速排序),493-500
analysis of(……的分析),497-499
example of partitioning(分割示例),494-497
the threshold(阈值),499-500
Quotient-offset collison handler(商偏移冲突处理器),550

R

Radix Sort(基数排序),506-508
Random access(随机存取),163,462
Random-access iterators(随机存取迭代器),478
Randomizing(随机化)
in program implementation(在程序实现中),84-86
in the simulated car wash(在仿真洗车实现中),272-274
Random number generator(随机数生成器),84-85
Random numbers(随机数),84
rb_tree class(rb_tree类)
Hewlett-Packard(惠普的),399
insert method in(插入方法),399-408,483
Reachable vertices(可达的顶点),571
readInto call(readInto调用),11,650
readInto method identifier(readInto方法标识符),25
Recursion(递归),93-161
backtracking(回溯),112-125
binary search(折半查找),125-135
cost of(代价),145-146
decimal-to-binary(十进制到二进制的转换),99-102
eight-queens problem(八皇后问题),156-158

execution frames(运行结构框架),96-99
factorials(阶乘),94-99
Fibonacci numbers(斐波纳契数),102
generating permutations(生成置换),135-144
implementation of(……的实现),277-285
indirect(间接),144-145
infinite(无穷),96
iterative binary search(迭代折半查找),134
knight's tour problem(马的遍历问题),158-161
mathematical(数学的),630-631
Towers of Hanoi game(汉诺塔游戏),103-111,154-156
Recursion application, a maze(递归应用,一个迷宫),117-125
Recursive functions(递归函数),145
Recursive methods, in binary search trees(折半查找树的递归方法),334-342
Red-black tree application spell-checker revisited(红黑树的应用,再次讨论拼写检查器),425-431
Red-black trees(红黑树),391-438
call to erase in which all four cases apply(四种情况下erase的调用),421
erase method(erase方法),408-421
height of a red-black tree(红黑树的高度),394-399
Hewlett-Packard's rb_tree class(惠普的rb_tree类),399
insert method in the rb_tree class(rb_tree类中的插入方法),399-408
red-black tree insertion with all three cases(三种情况下的红黑树插入),408
in searching(搜索),517
Standard Template Library's associative containers(标准模板库的关联容器),422-425
Red Rule(红色规则),392-393,403,418-419
Reference parameters, for pointers(引用参数,指针),38-39
Reserved words(保留字)
explicit,225
public,20
Responsibilities(职责),2
Return statement(返回语句),430
Reusable components, writing(编写可复用软件组件),17-18
Right child(右子女),310
Right rotation, 384
Robust programs(健壮的程序),65
Root items(根项),568
Rotations, in AVL trees(AVL树的旋转),354-358,384-386
Rules(规则)。参阅Corollaries; Laws; Principles
Path(路径),392-393,410,413,418-419,433
Red(红色),392-393,403-404,418-419
Splitting(分裂),78
Subclass Substitution(子类替换),23
Run-time analysis(运行时间分析),83-84
of chaining and double hashing in building a symbol table(在构造一个符号表中的链式与双散列),561
of program implementation(程序实现),83-84
Run-times, for sorting algorithms(排序算法的运行时间),500

S

Salesperson problem(货郎担问题),604
Sample function(示例函数),281-282
Scope-resolution operator(作用域解析运算符),16
Searching(查找),513-561。参阅Maze searching
binary(折半),126,515-517
depth-first(深度优先),322
framework to analyze(分析查找的框架),514
red-black trees in(红黑树),517
sequential(顺序),125,514-515
Second component(第二成分),619
Sequence class(Sequence类),33
Sequence containers, timing(计时顺序容器),224
Sequences(序列)
length of(长度),620
mathematical(数学),619-620
Sequential searching(顺序查找),125,514-515
Set application(集合应用),425-431
building a concordance(构造一个词汇索引),437-438
map class(映射类),427-431
multimap class(多映射类),431
multiset class(多集合类),427
a simple thesaurus(一个简单的辞典),436-437
Set class(集合类),427
the Standard Template Library's associative container(标准模板库的关联容器),422-425
Sets(集合),619
Shortest path through a network, finding with graph algorithms(使用图算法,寻找通过网络的最短路),584-587
SimpleClass class(SimpleClass类),30-32
Simulated car wash(洗车仿真),264-274,298-300
Smallest upper bound, in Big-O notation(大O表示法的最小上界),76
Software engineering(软件工程),63-91
Big-O notation(大O表示法),74-77
casting(类型转换),86-87
dependency diagrams(依赖关系图),67-71
drivers(驱动器),72
estimating the efficiency of methods(估算方法的效率),74
further expansion of Linked class(Linked类的进一步扩

充),91
getting Big-O estimates quickly(快速大O估算),77-81
is correctness feasible?(正确性实现的可行性),73
method interfaces and fields(方法接口和字段),67
method validation(方法验证),71-72
problem analysis(问题分析),64-65
program design(程序设计),67-71
program implementation(程序实现),71-87
program maintenance, 87-88
randomness(随机性),84-86
run-time analysis(运行时间分析),83-84
software developmental life cycle(软件开发生命周期),64
system tests(系统测试),66
timing and randomness(计时和随机性),86
trade-offs(平衡折中),81-83
writing reusable components(编写可复用软件组件),17-18
Sort algorithms(排序算法),500-501
sort_heap,486-487
Sorting(排序),477-511
divide and conquer algorithms(分治法),500
fast sorts(快速排序),483-500
of files(文件),509-511
Heap Sort(堆排序),485-487
Insertion Sort(插入排序),478-481
Merge Sort(归并排序),487-493
Quick Sort(快速排序),493-500
run-times for sorting algorithms(排序算法的运行时间),500
speed of(速度),481-483
stable(稳定性),478
Tree Sort(树排序),483-485
Source Code link(源代码连接),7
Source files(源文件),15
Spanning tree(生成树),580
generating a minimum(求出最小生成树),579-584
Spell-checker project(拼写检查项目),380-383,425-431
building a concordance(构造一个辞典),437-438
enhancing(改进的),389-390
map class(映射类),427-431
multimap class(多映射类),431
multiset class(多集合类),427
set class(集合类),427
a simple thesaurus(一个简单的辞典),436-437
Splitting, Rule of(分裂规则),78
Stable sorting(稳定排序),478
Stack application(堆栈应用)
converting infix to postfix(将中缀转换成后缀),285-295
how recursion is implemented(递归是如何实现的),96,277-285
Stack frames(堆栈帧),278
Stacks(堆栈),274-277
versus heaps(和堆对比),37-38
method interfaces for the stack class(堆栈类的方法接口),274-275
the stack class as a container adaptor(堆栈类作为容器配接器),276-277
using the stack class(使用堆栈),275-276
Standard Template Library(标准模板库),1,8,164
for container classes(容器类),58-59
deques in(双端队列),164
set class in(集合类),422-425
vectors in(向量),164
start iterator(start迭代器),197
Striling's approximation, of factorials(Stirling's的阶乘近似),453
Storage of list nodes(链表节点的存储),221-224
Storage structures for container classes(容器类的存储结构),35-62
arrays(数组),41-42
arrays and pointers(数组和指针),39-40
container classes(容器类),42-59
data structures and the Standard Template Library(数据结构和标准模板库),58-59
deallocation of dynamic variables(动态变量的存储单元释放),40-41
defining iterator operators(定义迭代器运算符),52
design and implementation of the iterator class(迭代器类的设计和实现),50-52
destructors(析构器),53-54
extending Linked class(扩展的Linked类),62
generic algorithms(通用型算法),54-58
the heap versus the stack(堆和堆栈对比),37-38
iterators(迭代器),48-50
linked structures(链式结构),44-48
overloading operator=(重载运算符=),54
pointer fields(指针字段),39
pointer-variable assignments versus dynamic-variable assignments (指针变量赋值与动态变量赋值对比),40
pointers(指针),36-41
pop_front method(pop_front方法),52-53
reference parameters(引用参数),38-39
Stream classes(流类),24
string class(字符串类),633-645
constructors(构造器),634-635

declaration of(声明),633-643
design and implementation of(设计和实现),644-645
methods that are neither constructors nor operators(既不是构造器也不是运算符的方法),638-642
nonmember functions(非成员函数),635-638
operators(运算符),635
String processing program(字符串处理程序),642-643
struct classes(结构类),14,46,123
struct objects(结构对象),50,83,329
Structures, linked in container classes(容器类中的链式结构),44-48
Subclasses(子类),18
Subclass Substitution Rule(子类替换规则),23
Sums(累加和),620-621
Superclasses(超类),18
Switch statements(Switch语句),7,87
Symbol tables(符号表),291
run-time comparison of chaining and double hashing in building(构造过程中链式与双散列的运行时间比较),561
Synonyms(同义词),521
System analysis(系统分析),66
System testing(系统测试),72

T

Templates(模板),44
arguments(变元),44
parameters(参数),45
Testing(测试)
bottom-up(自底向上),72
system(系统),72
Thesaurus(辞典),436-437
Threshold, in Quick Sort(阈值，在快速排序中),499-500
Time estimates, for network methods(网络方法的时间花费估算),604
Time function, in run-time analysis(运行时间分析中的time函数),83-84
Timing(计时)
a hash_map,539
in program implementation(在程序实现中),86
sequence containers(顺序容器),224
Tokens(记号),290-291
Towers of Hanoi game(汉诺塔游戏),103-111,154-156,631
iterative version of(迭代版本),154-156
recurrence relation in(递推关系),110-111
Trade-offs, in program implementation(程序实现中的平衡折中),81-83
Transition matrix(转换矩阵),289-290
Traveling salesperson problem(货郎担问题),604
Traversals of a binary tree(二叉树的遍历),318-324
breadthFirst(广度优先),322-324
inOrder(中序),318-320
postOrder(后序),320-321
preOrder(前序),321-322
Tree insertion, red-black(红黑树的插入),408
Trees(树),568-569。参阅AVL trees; Binary search trees; Binary trees; Directed trees; Fibonacci tree; Huffman trees Red-black trees; Spanning trees; Two-tree; Undirected trees
Tree Sort(树排序),483-485
analysis of(分析),485
example of(示例),484
tree_sort algorithm(tree_sort算法),483-484
tryToSolve method(tryToSolve方法),115-116,124,283,304
Two-tree(二-树),311-312,318
Type conversion, automatic(自动类型转换),217
typedef,37,256

U

UML(统一建模语言)。参阅Unified Modeling Language
Undirected graphs(无向图),564-566
acyclic(无环),566
complete(完全),564
connected(连通),566
directed(有向),568
Undirected trees(无向树),568
Unified Modeling Language(UML)(统一建模语言),69-71
Uniform Hashing Assumption(均匀散列设想),529,535-537
Universal Array-Pointer Law(通用数组指针定理),40
unsigned int,85,136,170,183,242,633-635,640,644
unsigned longs,526,529
Upper bound, in Big-O notation(大O表示法的上界),74,82
User, contract with developer(用户和开发者的合约),8,68

V

Validation, of methods(方法验证),71,72
Value_type class(Value_type类),537-538
Vector application, high-precision arithmetic(向量应用,高精度算法),184-190
Verctor-Iterator Corollary(向量－迭代器推论),174
Vector Iterators(向量迭代器),174-176
Vector methods(向量方法),166-167
differences between list methods and(与链表方法的区

别),213-214
Vectors(向量),165-184
comparison to other containers(与其他容器的对比),176-177
implementation of the vector class(向量类的实现),177-184
method interfaces for the vector class(向量类的方法接口),166-174
possible fields of the vector class(向量类可能的字段),177
in the Standard (Template Library在标准模板库中),164
Vertices(顶点),563-568
adjacent(邻接),565
reachable(可达),571
very_long_int class(very_long_int类),185,198,200-202
design of(设计),185-187
expanding(扩充),190
extending(扩展),203-204
implementation of(实现),187-190
Very long integers(很长的整数),184,198
Virtual methods(虚方法),649
void pointer(空指针),122-123

W

Weighted graphs(加权图),569
Weights(权),569
worstSpace(n),74,102
worstTime(n),74,90,98,102,197-198,209,318,321,342,364,514
writeBinary function(writeBinary函数),99-100,102,279-281,283

Z

Zero-parameter constructors(0-参数构造器),9