累计发行超100万册

清华大学计算机系列教材

数据结构（C语言版）

严蔚敏　吴伟民　编著

清华大学出版社

大家网
TopSage.com

累计发行超100万册

清华大学计算机系列教材

数据结构（C语言版）

严蔚敏　吴伟民　编著

清华大学出版社
北京

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

>> 更多精品资料请访问大家论坛计算机区…

内 容 简 介

《数据结构》(C 语言版)是为“数据结构”课程编写的教材,也可作为学习数据结构及其算法的 C 程序设计的参考教材。

本书的前半部分从抽象数据类型的角度讨论各种基本类型的数据结构及其应用;后半部分主要讨论查找和排序的各种实现方法及其综合分析比较。其内容和章节编排与 1992 年 4 月出版的《数据结构》(第二版)基本一致,但在本书中更突出了抽象数据类型的概念。全书采用类 C 语言作为数据结构和算法的描述语言。

本书概念表述严谨,逻辑推理严密,语言精炼,用词达意,并有配套出版的《数据结构题集》(C 语言版),既便于教学,又便于自学。

本书后附有光盘。光盘中含有可在 DOS 环境下运行的以类 C 语言描述的“数据结构算法动态模拟辅助教学软件,以及在 Windows 环境下运行的以类 PASCAL 或类 C 两种语言描述的“数据结构算法动态模拟辅助教学软件”。

本书可作为计算机类专业或信息类相关专业的本科或专科教材,也可供从事计算机工程与应用工作的科技工作者参考。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。
版权所有,侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

数据结构:C 语言版/严蔚敏,吴伟民编著. —北京:清华大学出版社,2007
(清华大学计算机系列教材)
ISBN 978-7-302-14751-0

Ⅰ. 数… Ⅱ. ①严… ②吴… Ⅲ. ①数据结构—高等学校—教材 ②C 语言—程序设计—高等学校—教材 Ⅳ. TP311.12. TP312

中国版本图书馆 CIP 数据核字(2007)第 024916 号

责任编辑:范素珍
责任印制:王秀菊
出版发行:清华大学出版社 **地 址**:北京清华大学学研大厦 A 座
http://www.tup.com.cn **邮 编**:100084
社 总 机:010-62770175 **邮 购**:010-62786544
投稿与读者服务:010-62776969,c-service@tup.tsinghua.edu.cn
质 量 反 馈:010-62772015,zhiliang@tup.tsinghua.edu.cn
印 刷 者:北京密云胶印厂
装 订 者:北京市密云县京文制本装订厂
经 销:全国新华书店
开 本:185×260 **印张**:21.75 **字数**:493 千字
附光盘 1 张
印 次:2009 年 9 月第30 次印刷
印 数:408001~458000
定 价:30.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话:(010)62770177 转 3103 产品编号:025648-01/TP

www.TopSage.com

大家网
TopSage.com

前　言

“数据结构”是计算机程序设计的重要理论技术基础，它不仅是计算机学科的核心课程，而且已成为其他理工专业的热门选修课。本书是为“数据结构”课程编写的教材，其内容选取符合教学大纲要求，并兼顾学科的广度和深度，适用面广。

本书的第1章综述数据、数据结构和抽象数据类型等基本概念；第2章至第7章从抽象数据类型的角度，分别讨论线性表、栈、队列、串、数组、广义表、树和二叉树以及图等基本类型的数据结构及其应用；第8章综合介绍操作系统和编译程序中涉及的动态存储管理的基本技术；第9章至第11章讨论查找和排序，除了介绍各种实现方法之外，并着重从时间上进行定性或定量的分析和比较；第12章介绍常用的文件结构。用过《数据结构》(第二版)的读者容易看出，本书内容和章节编排与1992年4月出版的《数据结构》(第二版)基本一致，但在本书中更突出了抽象数据类型的概念。对每一种数据结构，都分别给出相应的抽象数据类型规范说明和实现方法。

全书中采用类C语言作为数据结构和算法的描述语言，在对数据的存储结构和算法进行描述时，尽量考虑C语言的特色，如利用数组的动态分配实现顺序存储结构等。虽然C语言不是抽象数据类型的理想描述工具，但鉴于目前和近一两年内，“面向对象程序设计”并非数据结构的先修课程，故本书未直接采用类和对象等设施，而是从C语言中精选了一个核心子集，并增添C++语言的引用调用参数传递方式等，构成了一个类C描述语言。它使本书对各种抽象数据类型的定义和实现简明清晰，既不拘泥于C语言的细节，又容易转换成能上机执行的C或C++程序。

从课程性质上讲，“数据结构”是一门专业技术基础课。它的教学要求是：学会分析研究计算机加工的数据结构的特性，以便为应用涉及的数据选择适当的逻辑结构、存储结构及其相应的算法，并初步掌握算法的时间分析和空间分析的技术。另一方面，本课程的学习过程也是复杂程序设计的训练过程，要求学生编写的程序结构清楚和正确易读，符合软件工程的规范。如果说高级语言程序设计课程对学生进行了结构化程序设计(程序抽象)的初步训练的话，那么数据结构课程就要培养他们的数据抽象能力。本书将用规范的数学语言描述数据结构的定义，以突出其数学特性，同时，通过若干数据结构应用实例，引导学生学习数据类型的使用，为今后学习面向对象的程序设计作一些

www.TopSage.com

铺垫。

本书可作为计算机类专业的本科或专科教材，也可以作为信息类相关专业的选修教材，讲授学时可为50至80。教师可根据学时、专业和学生的实际情况，选讲或不讲目录页中带**的章节，甚至删去第5、8、11和12章。本书文字通俗，简明易懂，便于自学，也可供从事计算机应用等工作的科技人员参考。只需掌握程序设计基本技术便可学习本书。若具有离散数学和概率论的知识，则对书中某些内容更易理解。如果将本书《数据结构》(C语言版)和《数据结构》(第二版)作为关于数据结构及其算法的C和Pascal程序设计的对照教材，则有助于快速且深刻地掌握这两种语言。

与本书配套的还有《数据结构题集》(C语言版)，由清华大学出版社出版。书中提供配套的习题和实习题，并可作为学习指导手册。

严蔚敏　清华大学计算机科学与技术系
吴伟民　广东工业大学计算机学院

www.TopSage.com

大家网
TopSage.com

目　录

第 1 章　绪论 …… 1
1.1　什么是数据结构 …… 1
1.2　基本概念和术语 …… 4
1.3　抽象数据类型的表示与实现 …… 9
1.4　算法和算法分析 …… 13
1.4.1　算法 …… 13
1.4.2　算法设计的要求 …… 13
1.4.3　算法效率的度量 …… 14
1.4.4　算法的存储空间需求 …… 17
第 2 章　线性表 …… 18
2.1　线性表的类型定义 …… 18
2.2　线性表的顺序表示和实现 …… 21
2.3　线性表的链式表示和实现 …… 27
2.3.1　线性链表 …… 27
2.3.2　循环链表 …… 35
2.3.3　双向链表 …… 35
2.4　一元多项式的表示及相加 …… 39
第 3 章　栈和队列 …… 44
3.1　栈 …… 44
3.1.1　抽象数据类型栈的定义 …… 44
3.1.2　栈的表示和实现 …… 45
3.2　栈的应用举例 …… 48
3.2.1　数制转换 …… 48
3.2.2　括号匹配的检验 …… 49
3.2.3　行编辑程序 …… 49
3.2.4　迷宫求解 …… 50
3.2.5　表达式求值 …… 52
**3.3　栈与递归的实现 …… 54
3.4　队列 …… 58
3.4.1　抽象数据类型队列的定义 …… 58
3.4.2　链队列——队列的链式表示和实现 …… 60
3.4.3　循环队列——队列的顺序表示和实现 …… 63
**3.5　离散事件模拟 …… 65

www.TopSage.com

第 4 章　串 …… 70
4.1　串类型的定义 …… 70
4.2　串的表示和实现 …… 72
4.2.1　定长顺序存储表示 …… 73
4.2.2　堆分配存储表示 …… 75
4.2.3　串的块链存储表示 …… 78
**4.3　串的模式匹配算法 …… 79
4.3.1　求子串位置的定位函数 Index(S,T,pos) …… 79
4.3.2　模式匹配的一种改进算法 …… 80
4.4　串操作应用举例 …… 84
4.4.1　文本编辑 …… 84
**4.4.2　建立词索引表 …… 86
第 5 章　数组和广义表 …… 90
5.1　数组的定义 …… 90
5.2　数组的顺序表示和实现 …… 91
5.3　矩阵的压缩存储 …… 95
5.3.1　特殊矩阵 …… 95
5.3.2　稀疏矩阵 …… 96
5.4　广义表的定义 …… 106
5.5　广义表的存储结构 …… 109
**5.6　m 元多项式的表示 …… 110
**5.7　广义表的递归算法 …… 112
5.7.1　求广义表的深度 …… 113
5.7.2　复制广义表 …… 115
5.7.3　建立广义表的存储结构 …… 115
第 6 章　树和二叉树 …… 118
6.1　树的定义和基本术语 …… 118
6.2　二叉树 …… 121
6.2.1　二叉树的定义 …… 121
6.2.2　二叉树的性质 …… 123
6.2.3　二叉树的存储结构 …… 126
6.3　遍历二叉树和线索二叉树 …… 128
6.3.1　遍历二叉树 …… 128
6.3.2　线索二叉树 …… 132
6.4　树和森林 …… 135
6.4.1　树的存储结构 …… 135
6.4.2　森林与二叉树的转换 …… 137
6.4.3　树和森林的遍历 …… 138

www.TopSage.com

大家网
TopSage.com

**6.5 树与等价问题 …… 139
6.6 赫夫曼树及其应用 …… 144
6.6.1 最优二叉树(赫夫曼树)…… 144
6.6.2 赫夫曼编码…… 146
**6.7 回溯法与树的遍历 …… 149
**6.8 树的计数 …… 152
第7章 图…… 156
7.1 图的定义和术语 …… 156
7.2 图的存储结构 …… 160
7.2.1 数组表示法…… 161
7.2.2 邻接表…… 163
7.2.3 十字链表…… 164
7.2.4 邻接多重表…… 166
7.3 图的遍历 …… 167
7.3.1 深度优先搜索…… 167
7.3.2 广度优先搜索…… 169
7.4 图的连通性问题 …… 170
7.4.1 无向图的连通分量和生成树…… 170
** 7.4.2 有向图的强连通分量…… 172
7.4.3 最小生成树…… 173
** 7.4.4 关节点和重连通分量…… 176
7.5 有向无环图及其应用 …… 179
7.5.1 拓扑排序…… 180
7.5.2 关键路径…… 183
7.6 最短路径 …… 186
7.6.1 从某个源点到其余各顶点的最短路径…… 187
7.6.2 每一对顶点之间的最短路径…… 190
第8章 动态存储管理…… 193
8.1 概述 …… 193
8.2 可利用空间表及分配方法 …… 195
8.3 边界标识法 …… 198
8.3.1 可利用空间表的结构…… 198
8.3.2 分配算法…… 199
8.3.3 回收算法…… 201
8.4 伙伴系统 …… 203
8.4.1 可利用空间表的结构…… 203
8.4.2 分配算法…… 204
8.4.3 回收算法…… 205

www.TopSage.com

**8.5 无用单元收集 …… 206
**8.6 存储紧缩 …… 212
第9章 查找 …… 214
9.1 静态查找表 …… 216
9.1.1 顺序表的查找 …… 216
9.1.2 有序表的查找 …… 218
** 9.1.3 静态树表的查找 …… 222
9.1.4 索引顺序表的查找 …… 225
9.2 动态查找表 …… 226
9.2.1 二叉排序树和平衡二叉树 …… 227
9.2.2 B_树和 B^+ 树 …… 238
** 9.2.3 键树 …… 247
9.3 哈希表 …… 251
9.3.1 什么是哈希表 …… 251
9.3.2 哈希函数的构造方法 …… 253
9.3.3 处理冲突的方法 …… 256
9.3.4 哈希表的查找及其分析 …… 259
第10章 内部排序 …… 263
10.1 概述 …… 263
10.2 插入排序 …… 265
10.2.1 直接插入排序 …… 265
10.2.2 其他插入排序 …… 266
10.2.3 希尔排序 …… 271
10.3 快速排序 …… 272
10.4 选择排序 …… 277
10.4.1 简单选择排序 …… 277
10.4.2 树形选择排序 …… 278
10.4.3 堆排序 …… 279
10.5 归并排序 …… 283
10.6 基数排序 …… 284
10.6.1 多关键字的排序 …… 284
10.6.2 链式基数排序 …… 286
10.7 各种内部排序方法的比较讨论 …… 288
第11章 外部排序 …… 293
11.1 外存信息的存取 …… 293
11.2 外部排序的方法 …… 295
**11.3 多路平衡归并的实现 …… 297
**11.4 置换-选择排序 …… 299

www.TopSage.com

大家网
TopSage.com

** 11.5 最佳归并树 …… 304
第 12 章 文件 …… 306
12.1 有关文件的基本概念 …… 306
12.2 顺序文件 …… 308
12.3 索引文件 …… 311
12.4 ISAM 文件和 VSAM 文件 …… 313
12.4.1 ISAM 文件 …… 313
12.4.2 VSAM 文件 …… 316
12.5 直接存取文件(散列文件) …… 317
12.6 多关键字文件 …… 319
12.6.1 多重表文件 …… 319
12.6.2 倒排文件 …… 319
附录 A 名词索引 …… 322
附录 B 函数索引 …… 329
参考书目 …… 334

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

>> 更多精品资料请访问大家论坛计算机区…

大家网
TopSage.com

第1章 绪 论

自1946年第一台计算机问世以来，计算机产业的飞速发展已远远超出人们对它的预料，在某些生产线上，甚至几秒钟就能生产出一台微型计算机，产量猛增，价格低廉，这就使得它的应用范围迅速扩展。如今，计算机已深入到人类社会的各个领域。计算机的应用已不再局限于科学计算，而更多地用于控制、管理及数据处理等非数值计算的处理工作。与此相应，计算机加工处理的对象由纯粹的数值发展到字符、表格和图像等各种具有一定结构的数据，这就给程序设计带来一些新的问题。为了编写出一个“好”的程序，必须分析待处理的对象的特性以及各处理对象之间存在的关系。这就是“数据结构”这门学科形成和发展的背景。

1.1 什么是数据结构

一般来说，用计算机解决一个具体问题时，大致需要经过下列几个步骤：首先要从具体问题抽象出一个适当的数学模型，然后设计一个解此数学模型的算法，最后编出程序，进行测试、调整直至得到最终解答。寻求数学模型的实质是分析问题，从中提取操作的对象，并找出这些操作对象之间含有的关系，然后用数学的语言加以描述。例如，求解梁架结构中应力的数学模型为线性方程组；预报人口增长情况的数学模型为微分方程。然而，更多的非数值计算问题无法用数学方程加以描述。下面请看3个例子。

例1-1 图书馆的书目检索系统自动化问题。当你想借阅一本参考书但不知道书库中是否有的时候；或者，当你想找某一方面的参考书而不知图书馆内有哪些这方面的书时，你都需要到图书馆去查阅图书目录卡片。在图书馆内有各种名目的卡片：有按书名编排的，有按作者编排的，还有按分类编排的，等等。若利用计算机实现自动检索，则计算机处理的对象便是这些目录卡片上的书目信息。列在一张卡片上的一本书的书目信息可由登录号、书名、作者名、分类号、出版单位和出版时间等若干项组成，每一本书都有惟一的一个登录号，但不同的书目之间可能有相同的书名，或者有相同的作者名，或者有相同的分类号。由此，在书目自动检索系统中可以建立一张按登录号顺序排列的书目文件和3张分别按书名、作者名和分类号顺序排列的索引表，如图1.1所示。由这4张表构成的文件便是书目自动检索的数学模型，计算机的主要操作便是按照某个特定要求（如给定书名）对书目文件进行查询。诸如此类的还有查号系统自动化、仓库账目管理等。在这类文档管理的数学模型中，计算机处理的对象之间通常存在着的是一种最简单的线性关系，这类数学模型可称为线性的数据结构。

例1-2 计算机和人对弈问题。计算机之所以能和人对弈是因为有人将对弈的策略事先已存入计算机。由于对弈的过程是在一定规则下随机进行的，所以，为使计算机能灵活对弈就必须对对弈过程中所有可能发生的情况以及相应的对策都考虑周全，并且，一个

www.TopSage.com

001	高等数学	樊映川	S01	…
002	理论力学	罗远祥	L01	…
003	高等数学	华罗庚	S01	…
004	线性代数	栾汝书	S02	…
⋮	⋮	⋮	⋮	⋮

高等数学	001,003,…
理论力学	002,…
线性代数	004,…
⋮	

樊映川	001,…
华罗庚	003,…
栾汝书	004,…
⋮	

L	002,…
S	001,003,…
⋮	

图 1.1　图书目录文件示例

“好”的棋手在对弈时不仅要看棋盘当时的状态，还应能预测棋局发展的趋势，甚至最后结局。因此，在对弈问题中，计算机操作的对象是对弈过程中可能出现的棋盘状态——称为格局。例如图 1.2(a)所示为井字棋[①]的一个格局，而格局之间的关系是由比赛规则决定的。通常，这个关系不是线性的，因为从一个棋盘格局可以派生出几个格局，例如从图 1.2(a)所示的格局可以派生出 5 个格局，如图 1.2(b)所示，而从每一个新的格局又可派生出 4 个可能出现的格局。因此，若将从对弈开始到结束的过程中所有可能出现的格局都画在一张图上，则可得到一棵倒长的“树”。“树根”是对弈开始之前的棋盘格局，而所有的“叶子”就是可能出现的结局，对弈的过程就是从树根沿树杈到某个叶子的过程。“树”可以是某些非数值计算问题的数学模型，它也是一种数据结构。

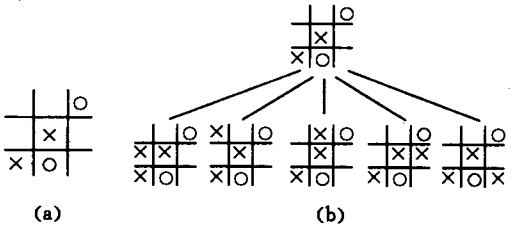


图 1.2　井字棋对弈“树”

(a) 棋盘格局示例；(b) 对弈树的局部。

例 1-3　多叉路口交通灯的管理问题。通常，在十字交叉路口只需设红、绿两色的交通灯便可保持正常的交通秩序，而在多叉路口需设几种颜色的交通灯才能既使车辆相互之间不碰撞，又能达到车辆的最大流通。假设有一个如图 1.3(a)所示的五叉路口，其中 C 和 E 为单行道。在路口有 13 条可行的通路，其中有的可以同时通行，如 A→B 和 E→C，

① 井字棋由两人对弈。棋盘为 3×3 的方格，当一方的 3 个棋子占同一行、或同一列、或同一对角线时便为胜方。

www.TopSage.com

大家网 TopSage.com

而有的不能同时通行，如 E→B 和 A→D。那么，在路口应如何设置交通灯进行车辆的管理呢？

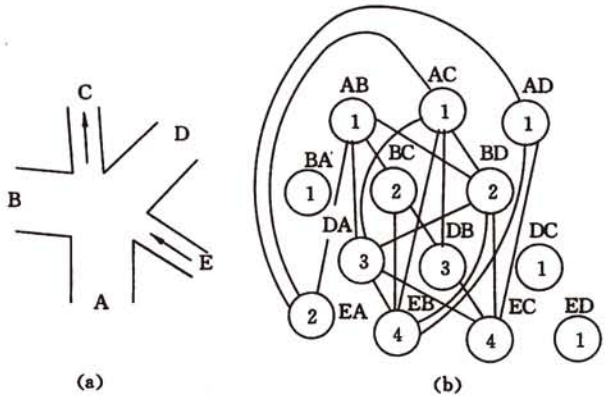


图 1.3　五叉路口交通管理示意图

(a) 五叉路口；(b) 表示通路的图

通常，这类交通、道路问题的数学模型是一种称为“图”的数据结构。例如在此例的问题中，可以图中一个顶点表示一条通路，而通路之间互相矛盾的关系以两个顶点之间的连线表示。如在图 1.3(b)中，每个圆圈表示图 1.3(a)所示五叉路口上的一条通路，两个圆圈之间的连线表示这两个圆圈表示的两条通路不能同时通行。设置交通灯的问题等价为对图的顶点的染色问题，要求对图上的每个顶点染一种颜色，并且要求有线相连的两个顶点不能具有相同颜色，而总的颜色种类应尽可能地少。图 1.3(b)所示为一种染色结果，圆圈中的数字表示交通灯的不同颜色，例如 3 号色灯亮时只有 D→A 和 D→B 两条路可通行。

综上 3 个例子可见，描述这类非数值计算问题的数学模型不再是数学方程，而是诸如表、树和图之类的数据结构。因此，简单说来，数据结构是一门研究非数值计算的程序设计问题中计算机的操作对象以及它们之间的关系和操作等的学科。

“数据结构”作为一门独立的课程在国外是从 1968 年才开始设立的。在这之前，它的某些内容曾在其他课程，如表处理语言中有所阐述。1968 年在美国一些大学的计算机系的教学计划中，虽然把“数据结构”规定为一门课程，但对课程的范围仍没有作明确规定。当时，数据结构几乎和图论，特别是和表、树的理论为同义语。随后，数据结构这个概念被扩充到包括网络、集合代数论、格、关系等方面，从而变成了现在称之为《离散结构》的内容。然而，由于数据必须在计算机中进行处理，因此，不仅考虑数据本身的数学性质，而且还必须考虑数据的存储结构，这就进一步扩大了数据结构的内容。近年来，随着数据库系统的不断发展，在“数据结构”课程中又增加了文件管理(特别是大型文件的组织等)的内容。

1968 年美国唐·欧·克努特教授开创了“数据结构”的最初体系，他所著的《计算机程序设计技巧》第一卷《基本算法》是第一本较系统地阐述数据的逻辑结构和存储结构及其操作的著作。从 20 世纪 60 年代末到 70 年代初，出现了大型程序，软件也相对独立，结构程序设计成为程序设计方法学的主要内容，人们就越来越重视“数据结构”，认为程序设

www.TopSage.com

计的实质是对确定的问题选择一种好的结构，加上设计一种好的算法。从 20 世纪 70 年代中期到 80 年代初，各种版本的数据结构著作就相继出现。

目前在我国，“数据结构”也已经不仅仅是计算机专业的教学计划中的核心课程之一，而且是其他非计算机专业的主要选修课程之一。

“数据结构”在计算机科学中是一门综合性的专业基础课。“数据结构”的研究不仅涉及到计算机硬件（特别是编码理论、存储装置和存取方法等）的研究范围，而且和计算机软件的研究有着更密切的关系，无论是编译程序还是操作系统，都涉及到数据元素在存储器中的分配问题。在研究信息检索时也必须考虑如何组织数据，以便查找和存取数据元素更为方便。因此，可以认为“数据结构”是介于数学、计算机硬件和计算机软件三者之间的一门核心课程（如图 1.4 所示）。在计算机科学中，“数据结构”不仅是一般程序设计（特别是非数值计算的程序设计）的基础，而且是设计和实现编译程序、操作系统、数据库系统及其他系统程序和大型应用程序的重要基础。

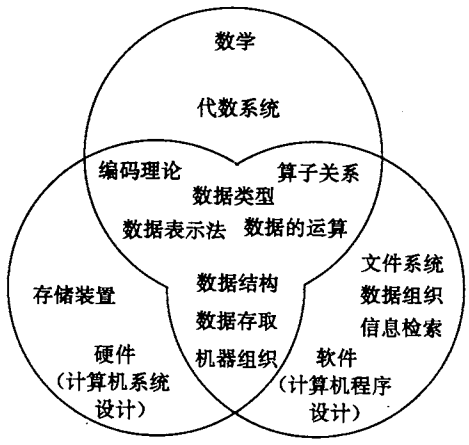


图 1.4 “数据结构”所处的地位

值得注意的是，“数据结构”的发展并未终结，一方面，面向各专门领域中特殊问题的数据结构得到研究和发展，如多维图形数据结构等；另一方面，从抽象数据类型的观点来讨论数据结构，已成为一种新的趋势，越来越被人们所重视。

1.2 基本概念和术语

在本节中，我们将对一些概念和术语赋以确定的含义，以便与读者取得“共同的语言”。这些概念和术语将在以后的章节中多次出现。

数据(data)是对客观事物的符号表示，在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。它是计算机程序加工的“原料”。例如，一个利用数值分析方法解代数方程的程序，其处理对象是整数和实数；一个编译程序或文字处理程序的处理对象是字符串。因此，对计算机科学而言，数据的含义极为广泛，如图像、声音等都可以通过编码而归之于数据的范畴。

数据元素(data element)是数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。例如，例 1-2 中的“树”中的一个棋盘格局，例 1-3 中的“图”中的一个圆圈都被称为一个数据元素。有时，一个数据元素可由若干个**数据项**(data item)组成，例如，例 1-1 中一本书的书目信息为一个数据元素，而书目信息中的每一项（如书名、作者名等）为一个数据项。数据项是数据的不可分割的最小单位。

数据对象(data object)是性质相同的数据元素的集合，是数据的一个子集。例如，整数数据对象是集合 $N=\{0,\pm1,\pm2,\cdots\}$，字母字符数据对象是集合 $C=\{'A',$

www.TopSage.com

大家网 TopSage.com

'B',…,'Z'}。

数据结构(data structure)是相互之间存在一种或多种特定关系的数据元素的集合。这是本书对数据结构的一种简单解释①。从 1.1 节中 3 个例子可以看到,在任何问题中,数据元素都不是孤立存在的,而是在它们之间存在着某种关系,这种数据元素相互之间的关系称为**结构**(structure)。根据数据元素之间关系的不同特性,通常有下列 4 类基本结构:(1) **集合** 结构中的数据元素之间除了"同属于一个集合"的关系外,别无其他关系②;(2) **线性结构** 结构中的数据元素之间存在一个对一个的关系;(3) **树形结构** 结构中的数据元素之间存在一个对多个的关系;(4) **图状结构或网状结构** 结构中的数据元素之间存在多个对多个的关系。图 1.5 为上述 4 类基本结构的关系图。由于"集合"是数据元素之间关系极为松散的一种结构,因此也可用其他结构来表示它。

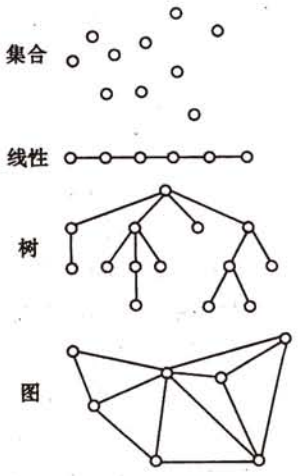


图 1.5 4 类基本结构关系图

数据结构的形式定义为:数据结构是一个二元组

$$\text{Data_Structure} = (D,S) \tag{1-1}$$

其中:D 是数据元素的有限集,S 是 D 上关系的有限集。下面举两个简单例子说明之。

例 1-4 在计算机科学中,复数可取如下定义:复数是一种数据结构

$$\text{Complex} = (C,R) \tag{1-2}$$

其中:C 是含两个实数的集合{c1,c2};$R=\{P\}$,而 P 是定义在集合 C 上的一种关系{⟨c1,c2⟩},其中有序偶⟨c1,c2⟩表示 c1 是复数的实部,c2 是复数的虚部。

例 1-5 假设我们需要编制一个事务管理的程序,管理学校科学研究课题小组的各项事务,则首先要为程序的操作对象——课题小组设计一个数据结构。假设每个小组由 1 位教师、1~3 名研究生及 1~6 名本科生组成,小组成员之间的关系是:教师指导研究生,而由每位研究生指导一至两名本科生。则可以如下定义数据结构:

$$\text{Group} = (P,R) \tag{1-3}$$

其中:$P=\{T,G_1,\cdots,G_n,S_{11}\cdots S_{nm}$③$,1\leqslant n\leqslant 3,1\leqslant m\leqslant 2\}$

$R=\{R_1,R_2\}$

$R_1=\{\langle T,G_i\rangle \mid 1\leqslant i\leqslant n,1\leqslant n\leqslant 3\}$

$R_2=\{\langle G_i,S_{ij}\rangle \mid 1\leqslant i\leqslant n,1\leqslant j\leqslant m,1\leqslant n\leqslant 3,1\leqslant m\leqslant 2\}$

上述数据结构的定义仅是对操作对象的一种数学描述,换句话说,是从操作对象抽象出来的数学模型。结构定义中的"关系"描述的是数据元素之间的逻辑关系,因此又称为

---

① 对于数据结构这个概念,至今尚未有一个被一致公认的定义,不同的人在使用这个词时所表达的意思有所不同。

② 这和数学中的集合概念是一致的。

③ T 表示导师,G 表示研究生,S 表示大学生。

www.TopSage.com

数据的**逻辑结构**。然而，讨论数据结构的目的是为了在计算机中实现对它的操作，因此还需研究如何在计算机中表示它。

数据结构在计算机中的表示（又称映像）称为数据的**物理结构**，又称**存储结构**。它包括数据元素的表示和关系的表示。在计算机中表示信息的最小单位是二进制数的一位，叫做**位**(bit)。在计算机中，我们可以用一个由若干位组合起来形成的一个位串表示一个数据元素（如用一个字长的位串表示一个整数，用 8 位二进制数表示一个字符等），通常称这个位串为**元素**[①](element)或**结点**(node)。当数据元素由若干数据项组成时，位串中对应于各个数据项的子位串称为**数据域**(data field)。因此，元素或结点可看成是数据元素在计算机中的映像。

数据元素之间的关系在计算机中有两种不同的表示方法：**顺序映像**和**非顺序映像**，并由此得到两种不同的存储结构：**顺序存储结构**和**链式存储结构**。顺序映像的特点是借助元素在存储器中的相对位置来表示数据元素之间的逻辑关系。例如，假设用两个字长的位串表示一个实数，则可以用地址相邻的 4 个字长的位串表示一个复数，如图 1.6(a)为表示复数 $z1=3.0-2.3i$ 和 $z2=-0.7+4.8i$ 的顺序存储结构；非顺序映像的特点是借助指示元素存储地址的**指针**(pointer)表示数据元素之间的逻辑关系，如图 1.6(b)为表示复数 $z1$ 的链式存储结构，其中实部和虚部之间的关系用值为“0415”的指针来表示（0415 是虚部的存储地址）[②]。数据的逻辑结构和物理结构是密切相关的两个方面，以后读者会看到，任何一个算法的设计取决于选定的数据（逻辑）结构，而算法的实现依赖于采用的存储结构。

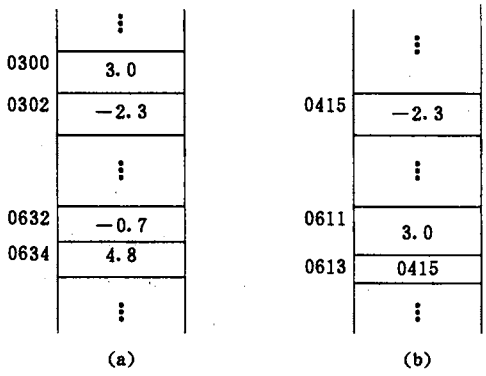


图 1.6　复数存储结构示意图

(a) 顺序存储结构；(b) 链式存储结构

如何描述存储结构呢。虽然存储结构涉及数据元素及其关系在存储器中的物理位置，但由于本书是在高级程序语言的层次上讨论数据结构的操作，因此不能如上那样直接以内存地址来描述存储结构，但我们可以借用高级程序语言中提供的“数据类型”来描述

---

① 本书中有时也把数据元素简称为元素，读者应从上下文去理解分辨之。

② 在实际应用中，像复数这类极简单的结构不需要采用链式存储结构，在此仅为了简化讨论而作为假例引用之。

www.TopSage.com

大家网 TopSage.com

它，例如可以用所有高级程序语言中都有的“一维数组”类型来描述顺序存储结构，以C语言提供的“指针”来描述链式存储结构。假如我们把C语言看成是一个执行C指令和C数据类型的虚拟处理器，那么本书中讨论的存储结构是数据结构在C虚拟处理器中的表示，不妨称它为**虚拟存储结构**。

数据类型(data type)是和数据结构密切相关的一个概念，它最早出现在高级程序语言中，用以刻画(程序)操作对象的特性。在用高级程序语言编写的程序中，每个变量、常量或表达式都有一个它所属的确定的数据类型。类型明显或隐含地规定了在程序执行期间变量或表达式所有可能取值的范围，以及在这些值上允许进行的操作。因此数据类型是一个值的集合和定义在这个值集上的一组操作的总称。例如，C语言中的整型变量，其值集为某个区间上的整数(区间大小依赖于不同的机器)，定义在其上的操作为加、减、乘、除和取模等算术运算。

按“值”的不同特性，高级程序语言中的数据类型可分为两类：一类是非结构的**原子类型**。原子类型的值是不可分解的，例如C语言中的基本类型(整型、实型、字符型和枚举类型)、指针类型和空类型。另一类是**结构类型**。结构类型的值是由若干成分按某种结构组成的，因此是可以分解的，并且它的成分可以是非结构的，也可以是结构的。例如数组的值由若干分量组成，每个分量可以是整数，也可以是数组等。在某种意义上，数据结构可以看成是“一组具有相同结构的值”，则结构类型可以看成由一种数据结构和定义在其上的一组操作组成。

实际上，在计算机中，数据类型的概念并非局限于高级语言中，每个处理器①(包括计算机硬件系统、操作系统、高级语言、数据库等)都提供了一组原子类型或结构类型。例如，一个计算机硬件系统通常含有“位”、“字节”、“字”等原子类型，它们的操作通过计算机设计的一套指令系统直接由电路系统完成，而高级程序语言提供的数据类型，其操作需通过编译器或解释器转化成低层，即汇编语言或机器语言的数据类型来实现。引入“数据类型”的目的，从硬件的角度看，是作为解释计算机内存中信息含义的一种手段，而对使用数据类型的用户来说，实现了信息的隐蔽，即将一切用户不必了解的细节都封装在类型中。例如，用户在使用“整数”类型时，既不需要了解“整数”在计算机内部是如何表示的，也不需要知道其操作是如何实现的。如“两整数求和”，程序设计者注重的仅仅是其“数学上求和”的抽象特性，而不是其硬件的“位”操作如何进行。

抽象数据类型(Abstract Data Type，简称ADT)是指一个数学模型以及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关，即不论其内部结构如何变化，只要它的数学特性不变，都不影响其外部的使用。

抽象数据类型和数据类型实质上是一个概念。例如，各个计算机都拥有的“整数”类型是一个抽象数据类型，尽管它们在不同处理器上实现的方法可以不同，但由于其定义的数学特性相同，在用户看来都是相同的。因此，“抽象”的意义在于数据类型的数学抽象特性。

---

① 在此指广义的处理器，包括计算机的硬件系统和软件系统。

www.TopSage.com

另一方面，抽象数据类型的范畴更广，它不再局限于前述各处理器中已定义并实现的数据类型（也可称这类数据类型为固有数据类型），还包括用户在设计软件系统时自己定义的数据类型。为了提高软件的复用率，在近代程序设计方法学中指出，一个软件系统的框架应建立在数据之上，而不是建立在操作之上（后者是传统的软件设计方法所为）。即在构成软件系统的每个相对独立的模块上，定义一组数据和施于这些数据上的一组操作，并在模块内部给出这些数据的表示及其操作的细节，而在模块外部使用的只是抽象的数据和抽象的操作。显然，所定义的数据类型的抽象层次越高，含有该抽象数据类型的软件模块的复用程度也就越高。

一个含抽象数据类型的软件模块通常应包含定义、表示和实现 3 个部分。

如前所述，抽象数据类型的定义由一个值域和定义在该值域上的一组操作组成。若按其值的不同特性，可细分为下列 3 种类型：

原子类型（atomic data type）属原子类型的变量的值是不可分解的。这类抽象数据类型较少，因为一般情况下，已有的固有数据类型足以满足需求。但有时也有必要定义新的原子数据类型，例如数位为 100 的整数。

固定聚合类型（fixed-aggregate data type）属该类型的变量，其值由确定数目的成分按某种结构组成。例如，复数是由两个实数依确定的次序关系构成。

可变聚合类型（variable-aggregate data type）和固定聚合类型相比较，构成可变聚合类型“值”的成分的数目不确定。例如，可定义一个“有序整数序列”的抽象数据类型，其中序列的长度是可变的。

显然，后两种类型可统称为结构类型。

和数据结构的形式定义相对应，抽象数据类型可用以下三元组表示

$$(D,S,P) \tag{1-4}$$

其中，D 是数据对象，S 是 D 上的关系集，P 是对 D 的基本操作集。本书采用以下格式定义抽象数据类型：

```
ADT 抽象数据类型名 {
    数据对象：〈数据对象的定义〉
    数据关系：〈数据关系的定义〉
    基本操作：〈基本操作的定义〉
}ADT 抽象数据类型名
```

其中，数据对象和数据关系的定义用伪码描述，基本操作的定义格式为

```
基本操作名(参数表)
    初始条件：〈初始条件描述〉
    操作结果：〈操作结果描述〉
```

基本操作有两种参数：赋值参数只为操作提供输入值；引用参数以 & 打头，除可提供输入值外，还将返回操作结果。“初始条件”描述了操作执行之前数据结构和参数应满足的条件，若不满足，则操作失败，并返回相应出错信息。“操作结果”说明了操作正常完成之后，数据结构的变化状况和应返回的结果。若初始条件为空，则省略之。

www.TopSage.com

大家网
TopSage.com

例 1-6 抽象数据类型三元组的定义：

```
ADT Triplet {
    数据对象:D={e1,e2,e3|e1,e2,e3∈ElemSet(定义了关系运算的某个集合)}
    数据关系:R1={<e1,e2>,<e2,e3>}
    基本操作:
      InitTriplet( &T, v1, v2, v3 )
        操作结果:构造了三元组 T,元素 e1,e2 和 e3 分别被赋以参数 v1,v2 和 v3 的值。
      DestroyTriplet( &T )
        操作结果:三元组 T 被销毁。
      Get( T, i, &e )
        初始条件:三元组 T 已存在,1≤i≤3。
        操作结果:用 e 返回 T 的第 i 元的值。
      Put( &T, i ,e )
        初始条件:三元组 T 已存在,1≤i≤3。
        操作结果:改变 T 的第 i 元的值为 e。
      IsAscending( T )
        初始条件:三元组 T 已存在。
        操作结果:如果 T 的 3 个元素按升序排列,则返回 1,否则返回 0。
      IsDescending( T )
        初始条件:三元组 T 已存在。
        操作结果:如果 T 的 3 个元素按降序排列,则返回 1,否则返回 0。
      Max( T, &e )
        初始条件:三元组 T 已存在。
        操作结果:用 e 返回 T 的 3 个元素中的最大值。
      Min( T, &e )
        初始条件:三元组 T 已存在。
        操作结果:用 e 返回 T 的 3 个元素中的最小值。
}ADT Triplet
```

多形数据类型(polymorphic data type)是指其值的成分不确定的数据类型。例如，例 1-6 中定义的抽象数据类型 Triplet，其元素 e1、e2 和 e3 可以是整数或字符或字符串，甚至更复杂地由多种成分构成(只要能进行关系运算即可)。然而，不论其元素具有何种特性，元素之间的关系相同，基本操作也相同。从抽象数据类型的角度看，具有相同的数学抽象特性，故称之为**多形数据类型**。显然，需借助面向对象的程序设计语言如 C ++ 等实现之。本书中讨论的各种数据类型大多是多形数据类型，限于本书采用类 C 语言作为描述工具，故只讨论含有确定成分的数据元素的情况。如例 1-6 中的 ElemSet 是某个确定的、将由用户自行定义的、含某个关系运算的数据对象。

1.3 抽象数据类型的表示与实现

抽象数据类型可通过固有数据类型来表示和实现，即利用处理器中已存在的数据类型来说明新的结构，用已经实现的操作来组合新的操作。由于本书在高级程序设计语言

www.TopSage.com

的虚拟层次上讨论抽象数据类型的表示和实现，并且讨论的数据结构及其算法主要是面向读者，故采用介于伪码和C语言之间的类C语言作为描述工具，有时也用伪码描述一些只含抽象操作的抽象算法。这使得数据结构与算法的描述和讨论简明清晰，不拘泥于C语言的细节，又能容易转换成C或者C++程序。

本书采用的类C语言精选了C语言的一个核心子集，同时做了若干扩充修改，增强了语言的描述功能。以下对其作简要说明。

(1) 预定义常量和类型：

```
// 函数结果状态代码
#define  TRUE         1
#define  FALSE        0
#define  OK           1
#define  ERROR        0
#define  INFEASIBLE   -1
#define  OVERFLOW     -2
// Status 是函数的类型，其值是函数结果状态代码
typedef  int  Status;
```

(2) 数据结构的表示(存储结构)用类型定义(typedef)描述。数据元素类型约定为ElemType，由用户在使用该数据类型时自行定义。

(3) 基本操作的算法都用以下形式的函数描述：

```
函数类型 函数名(函数参数表) {
  // 算法说明
  语句序列
} // 函数名
```

除了函数的参数需要说明类型外，算法中使用的辅助变量可以不作变量说明，必要时对其作用给予注释。一般而言，a、b、c、d、e等用作数据元素名，i、j、k、l、m、n等用作整型变量名，p、q、r等用作指针变量名。当函数返回值为函数结果状态代码时，函数定义为**Status**类型。为了便于算法描述，除了值调用方式外，增添了C++语言的引用调用的参数传递方式。在形参表中，以&打头的参数即为引用参数。

(4) 赋值语句有

```
简单赋值   变量名 = 表达式;
串联赋值   变量名 1 = 变量名 2 = … = 变量名 k = 表达式;
成组赋值   (变量名 1,…,变量名 k) = (表达式 1,…,表达式 k);
           结构名 = 结构名;
           结构名 = (值 1,…,值 k);
           变量名[ ] = 表达式;
           变量名[起始下标..终止下标] = 变量名[起始下标..终止下标];
交换赋值   变量名←→变量名;
条件赋值   变量名 = 条件表达式 ? 表达式 T : 表达式 F;
```

(5) 选择语句有

```
条件语句 1   if (表达式) 语句;
条件语句 2   if (表达式) 语句;
```


www.TopSage.com

大家网
TopSage.com

```
            else 语句;
开关语句 1  switch (表达式) {
                case 值 1: 语句序列 1; break;
                …
                case 值 n: 语句序列 n; break;
                default: 语句序列 n+1;
            }
开关语句 2  switch {
                case 条件 1: 语句序列 1; break;
                …
                case 条件 n: 语句序列 n; break;
                default: 语句序列 n+1;
            }
```

(6) 循环语句有

```
for 语句        for (赋初值表达式序列; 条件; 修改表达式序列) 语句;
while 语句      while (条件) 语句;
do-while 语句   do {
                    语句序列;
                }while (条件);
```

(7) 结束语句有

```
函数结束语句        return 表达式;
                    return;
case 结束语句       break;
异常结束语句        exit (异常代码);
```

(8) 输入和输出语句有

```
输入语句   scanf([格式串],变量 1,…,变量 n);
输出语句   printf([格式串],表达式 1,…,表达式 n);
```

通常省略格式串。

(9) 注释

```
单行注释   // 文字序列
```

(10) 基本函数有

```
求最大值          max (表达式 1,…,表达式 n)
求最小值          min (表达式 1,…,表达式 n)
求绝对值          abs (表达式)
求不足整数值      floor (表达式)
求进位整数值      ceil (表达式)
判定文件结束      eof (文件变量) 或 eof
判定行结束        eoln (文件变量) 或 eoln
```

(11) 逻辑运算约定

与运算&&：对于 A &&B，当 A 的值为 0 时，不再对 B 求值。
或运算||：对于 A ||B，当 A 的值为非 0 时，不再对 B 求值。

www.TopSage.com

例 1-7 抽象数据类型 Triplet 的表示和实现。

```
// - - - - - 采用动态分配的顺序存储结构 - - - - -
typedef ElemType * Triplet; // 由 InitTriplet 分配 3 个元素存储空间

// - - - - - 基本操作的函数原型说明 - - - - - -
Status InitTriplet (Triplet &T, ElemType v1, ElemType v2, ElemType v3);
  // 操作结果:构造了三元组 T,元素 e1,e2 和 e3 分别被赋以参数 v1,v2 和 v3 的值。
Status DestroyTriplet (Triplet &T);
  // 操作结果:三元组 T 被销毁。
Status Get (Triplet T, int i, ElemType &e);
  // 初始条件:三元组 T 已存在,1≤i≤3。
  // 操作结果:用 e 返回 T 的第 i 元的值。
Status Put (Triplet &T, int i, ElemType e);
  // 初始条件:三元组 T 已存在,1≤i≤3。
  // 操作结果:改变 T 的第 i 元的值为 e。
Status IsAscending (Triplet T);
  // 初始条件:三元组 T 已存在。
  // 操作结果:如果 T 的 3 个元素按升序排列,则返回 1,否则返回 0。
Status IsDescending (Triplet T);
  // 初始条件:三元组 T 已存在。
  // 操作结果:如果 T 的 3 个元素按降序排列,则返回 1,否则返回 0。
Status Max (Triplet T, ElemType &e);
  // 初始条件:三元组 T 已存在。
  // 操作结果:用 e 返回 T 的 3 个元素中的最大值。
Status Min (Triplet T, ElemType &e);
  // 初始条件:三元组 T 已存在。
  // 操作结果:用 e 返回 T 的 3 个元素中的最小值。

// - - - - - 基本操作的实现 - - - - -
Status InitTriplet (Triplet &T, ElemType v1, ElemType v2, ElemType v3) {
  // 构造三元组 T,依次置 T 的 3 个元素的初值为 v1,v2 和 v3。
  T = (ElemType *) malloc (3 * sizeof(ElemType));   // 分配 3 个元素的存储空间
  if (!T) exit(OVERFLOW);   // 分配存储空间失败
  T[0] = v1;    T[1] = v2;    T[2] = v3;
  return OK;
}// InitTriplet
Status DestroyTriplet (Triplet &T) {
  // 销毁三元组 T。
  free(T);    T = NULL;
  return OK;
}// DestroyTriplet
Status Get (Triplet T, int i, ElemType &e) {
  // 1≤i≤3,用 e 返回 T 的第 i 元的值。
  if (i<1 || i>3) return ERROR;
  e = T[i-1];
  return OK;
}// Get
Status Put (Triplet &T, int i, ElemType e) {
  // 1≤i≤3,置 T 的第 i 元的值为 e。
  if (i<1 || i>3) return ERROR;
  T[i-1] = e;
  return OK;
```


www.TopSage.com

大家网
TopSage.com

```
}// Put
Status IsAscending (Triplet T) {
  // 如果 T 的 3 个元素按升序排列,则返回 1,否则返回 0。
  return (T[0] <= T[1]) && (T[1] <= T[2]);
}  // IsAscending
Status IsDescending (Triplet T) {
  // 如果 T 的 3 个元素按降序排列,则返回 1,否则返回 0。
  return (T[0] >= T[1]) && (T[1]>= T[2]);
}// IsDescending
Status Max (Triplet T, ElemType &e) {
  // 用 e 返回指向 T 的最大元素的值。
  e = (T[0] >= T[1]) ? ((T[0] >= T[2]) ? T[0] : T[2])
                     : ((T[1] >= T[2]) ? T[1] : T[2]);
  return OK;
}// Max
Status Min (Triplet T, ElemType &e) {
  // 用 e 返回指向 T 的最小元素的值。
  e = (T[0] <= T[1]) ? ((T[0] <= T[2]) ? T[0] : T[2])
                     : ((T[1] <= T[2]) ? T[1] : T[2]);
  return OK;
}// Min
```

1.4 算法和算法分析

1.4.1 算法

算法(algorithm)是对特定问题求解步骤的一种描述,它是指令的有限序列,其中每一条指令表示一个或多个操作;此外,一个算法还具有下列 5 个重要特性:

(1) **有穷性** 一个算法必须总是(对任何合法的输入值)在执行有穷步之后结束,且每一步都可在有穷时间[①]内完成。

(2) **确定性** 算法中每一条指令必须有确切的含义,读者理解时不会产生二义性。并且,在任何条件下,算法只有惟一的一条执行路径,即对于相同的输入只能得出相同的输出。

(3) **可行性** 一个算法是能行的,即算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现的。

(4) **输入** 一个算法有零个或多个的输入,这些输入取自于某个特定的对象的集合。

(5) **输出** 一个算法有一个或多个的输出,这些输出是同输入有着某些特定关系的量。

1.4.2 算法设计的要求

通常设计一个"好"的算法应考虑达到以下目标。

(1) **正确性**[②](correctness) 算法应当满足具体问题的需求。通常一个大型问题的

---

① 在此,有穷的概念不是纯数学的,而是在实际上是合理的,可接受的。

② 有关算法正确性的严格证明,请参阅参考书目[8]。

www.TopSage.com

需求，要以特定的规格说明方式给出，而一个实习问题或练习题，往往就不那么严格，目前多数是用自然语言描述需求，它至少应当包括对于输入、输出和加工处理等的明确的无歧义性的描述。设计或选择的算法应当能正确地反映这种需求；否则，算法的正确与否的衡量准则就不存在了。

“正确”一词的含义在通常的用法中有很大差别，大体可分为以下 4 个层次：a. 程序不含语法错误；b. 程序对于几组输入数据能够得出满足规格说明要求的结果；c. 程序对于精心选择的典型、苛刻而带有刁难性的几组输入数据能够得出满足规格说明要求的结果；d. 程序对于一切合法的输入数据都能产生满足规格说明要求的结果。显然，达到第 d 层意义下的正确是极为困难的，所有不同输入数据的数量大得惊人，逐一验证的方法是不现实的。对于大型软件需要进行专业测试，而一般情况下，通常以第 c 层意义的正确性作为衡量一个程序是否合格的标准。

(2) **可读性**(readability)　算法主要是为了人的阅读与交流，其次才是机器执行。可读性好有助于人对算法的理解；晦涩难懂的程序易于隐藏较多错误，难以调试和修改。

(3) **健壮性**(robustness)　当输入数据非法时，算法也能适当地做出反应或进行处理，而不会产生莫明其妙的输出结果。例如，一个求凸多边形面积的算法，是采用求各三角形面积之和的策略来解决问题的。当输入的坐标集合表示的是一个凹多边形时，不应继续计算，而应报告输入出错。并且，处理出错的方法应是返回一个表示错误或错误性质的值，而不是打印错误信息或异常，并中止程序的执行，以便在更高的抽象层次上进行处理。

(4) **效率与低存储量需求**　通俗地说，效率指的是算法执行的时间。对于同一个问题如果有多个算法可以解决，执行时间短的算法效率高。存储量需求指算法执行过程中所需要的最大存储空间。效率与低存储量需求这两者都与问题的规模有关。求 100 个人的平均分与求 1000 个人的平均分所花的执行时间或运行空间显然有一定的差别。

1.4.3　算法效率的度量

算法执行时间需通过依据该算法编制的程序在计算机上运行时所消耗的时间来度量。而度量一个程序的执行时间通常有两种方法。

(1) 事后统计的方法　因为很多计算机内部都有计时功能，有的甚至可精确到毫秒级，不同算法的程序可通过一组或若干组相同的统计数据以分辨优劣。但这种方法有两个缺陷：一是必须先运行依据算法编制的程序；二是所得时间的统计量依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优劣。因此人们常常采用另一种事前分析估算的方法。

(2) 事前分析估算的方法　一个用高级程序语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：

① 依据的算法选用何种策略；

② 问题的规模，例如求 100 以内还是 1000 以内的素数；

③ 书写程序的语言，对于同一个算法，实现语言的级别越高，执行效率就越低；

④ 编译程序所产生的机器代码的质量；

www.TopSage.com

大家网 TopSage.com

⑤ 机器执行指令的速度。

显然，同一个算法用不同的语言实现，或者用不同的编译程序进行编译，或者在不同的计算机上运行时，效率均不相同。这表明使用绝对的时间单位衡量算法的效率是不合适的。撇开这些与计算机硬件、软件有关的因素，可以认为一个特定算法“运行工作量”的大小，只依赖于问题的规模（通常用整数量 n 表示），或者说，它是问题规模的函数。

一个算法是由控制结构（顺序、分支和循环 3 种）和原操作（指固有数据类型的操作）构成的，则算法时间取决于两者的综合效果。为了便于比较同一问题的不同算法，通常的做法是，从算法中选取一种对于所研究的问题（或算法类型）来说是基本操作的原操作，以该基本操作重复执行的次数作为算法的时间量度。

例如，在如下所示的两个 $N\times N$ 矩阵相乘的算法中，“乘法”运算是“矩阵相乘问题”的基本操作。整个算法的执行时间与该基本操作（乘法）重复执行的次数 n^3 成正比，记作 $T(n)=O(n^3)$[①]。

```
for (i = 1; i<= n; ++ i)
    for (j = 1; j<= n; ++ j) {
        c[i][j] = 0;
        for (k = 1; k<= n; ++ k)
            c[i][j] += a[i][k] * b[k][j];
    }
```

一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数 $f(n)$，算法的时间量度记作

$$T(n) = O(f(n)) \tag{1-5}$$

它表示随问题规模 n 的增大，算法执行时间的增长率和 $f(n)$ 的增长率相同，称做算法的**渐近时间复杂度**(asymptotic time complexity)，简称**时间复杂度**。

显然，被称做问题的基本操作的原操作应是其重复执行次数和算法的执行时间成正比的原操作，多数情况下它是最深层循环内的语句中的原操作，它的执行次数和包含它的语句的频度相同。语句的**频度**(frequency count)指的是该语句重复执行的次数，例如，在下列 3 个程序段中：

```
(a) { ++x; s = 0; }
(b) for (i = 1; i<= n; ++ i) { ++x; s += x; }
(c) for (j = 1; j<= n; ++ j)
        for (k = 1; k<= n; ++ k) { ++x; s += x; }
```

含基本操作“x 增 1”的语句的频度分别为 1、n 和 n^2，则这 3 个程序段的时间复杂度分别为 $O(1)$、$O(n)$和 $O(n^2)$，分别称为常量阶、线性阶和平方阶。算法还可能呈现的时间复杂度有对数阶 $O(\log n)$、指数阶 $O(2^n)$等。不同数量级时间复杂度的性状如图 1.7 所示。从图中可见，我们应该尽可能选用多项式阶 $O(n^k)$的算法，而不希望用指数阶的算法。

一般情况下，对一个问题（或一类算法）只需选择一种基本操作来讨论算法的时间复

---

① “O”的形式定义为[2]：若 $f(n)$是正整数 n 的一个函数，则 $x_n=O(f(n))$表示存在一个正的常数 M，使得当 $n\geqslant n_0$ 时都满足 $|x_n|\leqslant M|f(n)|$。

www.TopSage.com

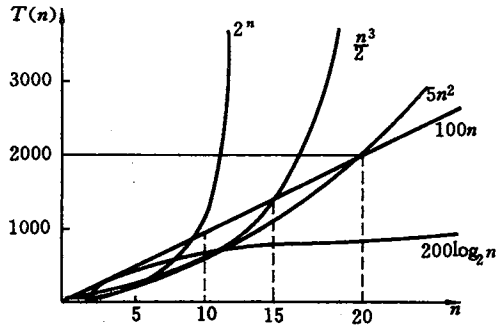


图 1.7　常见函数的增长率

杂度即可，有时也需要同时考虑几种基本操作，甚至可以对不同的操作赋予不同权值，以反映执行不同操作所需的相对时间，这种做法便于综合比较解决同一问题的两种完全不同的算法。

由于算法的时间复杂度考虑的只是对于问题规模 n 的增长率，则在难以精确计算基本操作执行次数(或语句频度)的情况下，只需求出它关于 n 的增长率或阶即可。例如，在下列程序段中：

```
for (i = 2; i<= n; ++ i)
    for (j = 2; j<= i - 1; ++ j) { ++ x; a[i][j] = x; }
```

语句 ++ x 的执行次数关于 n 的增长率为 n^2，它是语句频度表达式 $(n-1)(n-2)/2$ 中增长最快的项。

有的情况下，算法中基本操作重复执行的次数还随问题的输入数据集不同而不同。例如在下列起泡排序的算法中：

```
void bubble_sort(int a[ ], int n) {
    // 将 a 中整数序列重新排列成自小至大有序的整数序列。
    for (i = n - 1, change = TRUE; i≥1 && change; -- i) {
        change = FALSE;
        for (j = 0; j<i; ++ j)
            if (a[j] > a[j + 1]) {a[j]←→a[j + 1]; change = TRUE; }
    }
}// bubble_sort
```

“交换序列中相邻两个整数”为基本操作。当 a 中初始序列为自小至大有序，基本操作的执行次数为 0；当初始序列为自大至小有序时，基本操作的执行次数为 $n(n-1)/2$。对这类算法的分析，一种解决的办法是计算它的平均值，即考虑它对所有可能的输入数据集的期望值，此时相应的时间复杂度为算法的平均时间复杂度。如假设 a 中初始输入数据可能出现 $n!$ 种的排列情况的概率相等，则起泡排序的平均时间复杂度 $T_{avg}(n)=O(n^2)$，然而，在很多情况下，各种输入数据集出现的概率难以确定，算法的平均时间复杂度也就难以确定。因此，另一种更可行也更常用的办法是讨论算法在最坏情况下的时间复杂度，即分析最坏情况以估算算法执行时间的一个上界。例如，上述起泡排序的最坏情况为 a 中

www.TopSage.com

大家网
TopSage.com

初始序列为自大至小有序，则起泡排序算法在最坏情况下的时间复杂度为$T(n)=O(n^2)$。在本书以后各章中讨论的时间复杂度，除特别指明外，均指最坏情况下的时间复杂度。

实践中我们可以把事前估算和事后统计两种办法结合起来使用。以两个矩阵相乘为例，若上机运行两个 10×10 的矩阵相乘，执行时间为 12ms，则由算法的时间复杂度 $T(n)=O(n^3)$ 可估算两个 31×31 的矩阵相乘所需时间大致为 $(31/10)^3\cdot12\text{ms}\approx358\text{ms}$。

1.4.4 算法的存储空间需求

类似于算法的时间复杂度，本书中以**空间复杂度**(space complexity)作为算法所需存储空间的量度，记作

$$S(n)=O(f(n)) \tag{1-6}$$

其中 n 为问题的规模(或大小)。一个上机执行的程序除了需要存储空间来寄存本身所用指令、常数、变量和输入数据外，也需要一些对数据进行操作的工作单元和存储一些为实现计算所需信息的辅助空间。若输入数据所占空间只取决于问题本身，和算法无关，则只需要分析除输入和程序之外的额外空间，否则应同时考虑输入本身所需空间(和输入数据的表示形式有关)。若额外空间相对于输入数据量来说是常数，则称此算法为**原地工作**，第 10 章讨论的有些排序算法就属于这类。又如果所占空间量依赖于特定的输入，则除特别指明外，均按最坏情况来分析。

www.TopSage.com

第2章 线 性 表

从第 2 章至第 4 章将讨论线性结构。线性结构的特点是：在数据元素的非空有限集中，(1)存在惟一的一个被称做“第一个”的数据元素；(2)存在惟一的一个被称做“最后一个”的数据元素；(3)除第一个之外，集合中的每个数据元素均只有一个前驱；(4)除最后一个之外，集合中每个数据元素均只有一个后继。

2.1 线性表的类型定义

线性表(linear_list)是最常用且最简单的一种数据结构。简言之，一个线性表是 n 个数据元素的有限序列。至于每个数据元素的具体含义，在不同的情况下各不相同，它可以是一个数或一个符号，也可以是一页书，甚至其他更复杂的信息。例如，26 个英文字母的字母表：

$$(A,B,C,\cdots,Z)$$

是一个线性表，表中的数据元素是单个字母字符。又如，某校从 1978 年到 1983 年各种型号的计算机拥有量的变化情况，可以用线性表的形式给出：

$$(6,17,28,50,92,188)$$

表中的数据元素是整数。

在稍复杂的线性表中，一个数据元素可以由若干个**数据项**(item)组成。在这种情况下，常把数据元素称为**记录**(record)，含有大量记录的线性表又称**文件**(file)。

例如，一个学校的学生健康情况登记表如图 2.1 所示，表中每个学生的情况为一个记录，它由姓名、学号、性别、年龄、班级和健康状况等 6 个数据项组成。

姓 名	学 号	性 别	年 龄	班 级	健康状况
王小林	790631	男	18	计 91	健康
陈红	790632	女	20	计 91	一般
刘建平	790633	男	21	计 91	健康
张立立	790634	男	17	计 91	神经衰弱
⋮	⋮	⋮	⋮	⋮	⋮

图 2.1 学生健康情况登记表

综合上述 3 个例子可见，线性表中的数据元素可以是各种各样的，但同一线性表中的元素必定具有相同特性，即属同一数据对象，相邻数据元素之间存在着序偶关系。若将线性表记为

$$(a_1,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n) \tag{2-1}$$

www.TopSage.com

大家网 TopSage.com

则表中 a_{i-1} 领先于 a_i，a_i 领先于 a_{i+1}，称 a_{i-1} 是 a_i 的直接前驱元素，a_{i+1} 是 a_i 的直接后继元素。当 $i=1,2,\cdots,n-1$ 时，a_i 有且仅有一个直接后继，当 $i=2,3,\cdots,n$ 时，a_i 有且仅有一个直接前驱。

线性表中元素的个数 $n(n\geqslant 0)$ 定义为线性表的长度，$n=0$ 时称为空表。在非空表中的每个数据元素都有一个确定的位置，如 a_1 是第一个数据元素，a_n 是最后一个数据元素，a_i 是第 i 个数据元素，称 i 为数据元素 a_i 在线性表中的位序。

线性表是一个相当灵活的数据结构，它的长度可根据需要增长或缩短，即对线性表的数据元素不仅可以进行访问，还可进行插入和删除等。

抽象数据类型线性表的定义如下：

```
ADT List {
  数据对象:D = { a_i | a_i ∈ ElemSet, i = 1,2,…,n, n≥0 }
  数据关系:R1 = { <a_{i-1},a_i> | a_{i-1},a_i ∈ D, i = 2,…,n }
  基本操作:
    InitList( &L )
      操作结果:构造一个空的线性表 L。
    DestroyList( &L )
      初始条件:线性表 L 已存在。
      操作结果:销毁线性表 L。
    ClearList( &L )
      初始条件:线性表 L 已存在。
      操作结果:将 L 重置为空表。
    ListEmpty( L )
      初始条件:线性表 L 已存在。
      操作结果:若 L 为空表,则返回 TRUE,否则返回 FALSE。
    ListLength( L )
      初始条件:线性表 L 已存在。
      操作结果:返回 L 中数据元素个数。
    GetElem( L, i, &e )
      初始条件:线性表 L 已存在,1≤i≤ListLength(L)。
      操作结果:用 e 返回 L 中第 i 个数据元素的值。
    LocateElem( L, e, compare() )
      初始条件:线性表 L 已存在,compare()是数据元素判定函数。
      操作结果:返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。若这样的数据元素
                不存在,则返回值为 0。
    PriorElem( L, cur_e, &pre_e )
      初始条件:线性表 L 已存在。
      操作结果:若 cur_e 是 L 的数据元素,且不是第一个,则用 pre_e 返回它的前驱,否则操作
                失败,pre_e 无定义。
    NextElem( L, cur_e, &next_e )
      初始条件:线性表 L 已存在。
      操作结果:若 cur_e 是 L 的数据元素,且不是最后一个,则用 next_e 返回它的后继,否则操
                作失败,next_e 无定义。
    ListInsert( &L, i, e )
      初始条件:线性表 L 已存在,1≤i≤ListLength(L) + 1。
      操作结果:在 L 中第 i 个位置之前插入新的数据元素 e,L 的长度加 1。
    ListDelete(&L, i, &e)
```


www.TopSage.com

初始条件：线性表 L 已存在且非空，1≤i≤ListLength(L)。
操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1。
ListTraverse(L, visit())
初始条件：线性表 L 已存在。
操作结果：依次对 L 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失败。
} ADT List

对上述定义的抽象数据类型线性表，还可进行一些更复杂的操作，例如，将两个或两个以上的线性表合并成一个线性表；把一个线性表拆开成两个或两个以上的线性表；重新复制一个线性表等。

例 2-1 假设利用两个线性表 LA 和 LB 分别表示两个集合 A 和 B（即线性表中的数据元素即为集合中的成员），现要求一个新的集合 $A=A\cup B$。这就要求对线性表作如下操作：扩大线性表 LA，将存在于线性表 LB 中而不存在于线性表 LA 中的数据元素插入到线性表 LA 中去。只要从线性表 LB 中依次取得每个数据元素，并依值在线性表 LA 中进行查访，若不存在，则插入之。上述操作过程可用下列算法描述之。

```
void union(List &La, List Lb) {
  // 将所有在线性表 Lb 中但不在 La 中的数据元素插入到 La 中
  La_len = ListLength(La);  Lb_len = ListLength(Lb); // 求线性表的长度
  for (i = 1; i <= Lb_len; i++) {
    GetElem(Lb, i, e);                    // 取 Lb 中第 i 个数据元素赋给 e
    if (!LocateElem(La, e, equal)) ListInsert(La, ++La_len①, e);
                                  // La 中不存在和 e 相同的数据元素，则插入之
  }
}// union
```

算法 2.1

例 2-2 已知线性表 LA 和 LB 中的数据元素按值非递减有序排列，现要求将 LA 和 LB 归并为一个新的线性表 LC，且 LC 中的数据元素仍按值非递减有序排列。例如，设

$$LA = (3,5,8,11)$$
$$LB = (2,6,8,9,11,15,20)$$

则

$$LC = (2,3,5,6,8,8,9,11,11,15,20)$$

从上述问题要求可知，LC 中的数据元素或是 LA 中的数据元素，或是 LB 中的数据元素，则只要先设 LC 为空表，然后将 LA 或 LB 中的元素逐个插入到 LC 中即可。为使 LC 中元素按值非递减有序排列，可设两个指针 i 和 j 分别指向 LA 和 LB 中某个元素，若设 i 当前所指的元素为 a，j 当前所指的元素为 b，则当前应插入到 LC 中的元素 c 为

$$c=\begin{cases} a & \text{当 } a\leqslant b \text{ 时} \\ b & \text{当 } a>b \text{ 时} \end{cases}$$

① ++La_len 表示参数 La_len 的值先增 1，然后再传递给函数。若数学符号 ++ 在参量名之后，则表示先将参数传递给函数，然后参数的值再增 1，以后均类同。

www.TopSage.com

大家网 TopSage.com

显然，指针 i 和 j 的初值均为 1，在所指元素插入 LC 之后，在 LA 或 LB 中顺序后移。上述归并算法如算法 2.2 所示。

```
void MergeList(List La, List Lb, List &Lc) {
    // 已知线性表 La 和 Lb 中的数据元素按值非递减排列。
    // 归并 La 和 Lb 得到新的线性表 Lc,Lc 的数据元素也按值非递减排列。
    InitList(Lc);
    i = j = 1;k = 0;
    La_len = ListLength(La); Lb_len = ListLength(Lb);
    while ((i <= La_len) && (j <= Lb_len)) {// La 和 Lb 均非空
      GetElem(La, i, ai);  GetElem(Lb, j, bj);
      if (ai <= bj)  {ListInsert(Lc, ++k, ai); ++i; }
      else {ListInsert(Lc, ++k, bj); ++j; }
    }
    while (i <= La_len) {
        GetElem(La, i++, ai); ListInsert(Lc, ++k, ai);
    }
    while (j <= Lb_len) {
        GetElem(Lb, j++, bj); ListInsert(Lc, ++k, bj);
    }
}// MergeList
```

算法 2.2

上述两个算法的时间复杂度取决于抽象数据类型 List 定义中基本操作的执行时间。假如 GetElem 和 ListInsert 这两个操作的执行时间和表长无关，LocateElem 的执行时间和表长成正比，则算法 2.1 的时间复杂度为 $O(\text{ListLength}(LA)\times\text{ListLength}(LB))$，算法 2.2 的时间复杂度则为 $O(\text{ListLength}(LA)+\text{ListLength}(LB))$。虽然算法 2.2 中含 3 个(**while**)循环语句，但只有当 i 和 j 均指向表中实际存在的数据元素时，才能取得数据元素的值并进行相互比较；并且当其中一个线性表的数据元素均已插入到线性表 LC 中后，只要将另外一个线性表中的剩余元素依次插入即可。因此，对于每一组具体的输入(LA 和 LB)，后两个(**while**)循环语句只执行一个循环体。

2.2 线性表的顺序表示和实现

线性表的顺序表示指的是用一组地址连续的存储单元依次存储线性表的数据元素。

假设线性表的每个元素需占用 l 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第 $i+1$ 个数据元素的存储位置 $LOC(a_{i+1})$ 和第 i 个数据元素的存储位置 $LOC(a_i)$ 之间满足下列关系：

$$LOC(a_{i+1}) = LOC(a_i) + l$$

一般来说，线性表的第 i 个数据元素 a_i 的存储位置为

$$LOC(a_i) = LOC(a_1) + (i-1)\times l \qquad (2\text{-}2)$$

www.TopSage.com

式中 $LOC(a_1)$是线性表的第一个数据元素 a_1 的存储位置，通常称做线性表的起始位置或基地址。

线性表的这种机内表示称做线性表的顺序存储结构或顺序映像(sequential mapping)，通常，称这种存储结构的线性表为顺序表。它的特点是，为表中相邻的元素 a_i 和 a_{i+1} 赋以相邻的存储位置 $LOC(a_i)$ 和 $LOC(a_{i+1})$。换句话说，以元素在计算机内"物理位置相邻"来表示线性表中数据元素之间的逻辑关系。每一个数据元素的存储位置都和线性表的起始位置相差一个和数据元素在线性表中的位序成正比的常数(见图 2.2)。由此，只要确定了存储线性表的起始位置，线性表中任一数据元素都可随机存取，所以线性表的顺序存储结构是一种随机存取的存储结构。

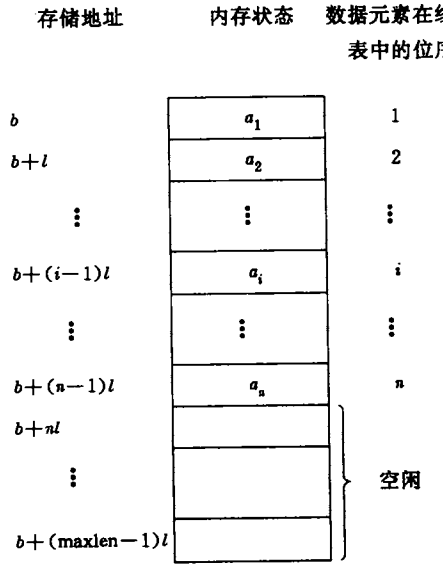


图 2.2 线性表的顺序存储结构示意图

由于高级程序设计语言中的数组类型也有随机存取的特性，因此，通常都用数组来描述数据结构中的顺序存储结构。在此，由于线性表的长度可变，且所需最大存储空间随问题不同而不同，则在 C 语言中可用动态分配的一维数组，如下描述。

```
//------ 线性表的动态分配顺序存储结构 ------
#define  LIST_INIT_SIZE    100  // 线性表存储空间的初始分配量
#define  LISTINCREMENT      10  // 线性表存储空间的分配增量
typedef  struct {
    ElemType  *elem;          // 存储空间基址
    int       length;         // 当前长度
    int       listsize;       // 当前分配的存储容量(以 sizeof(ElemType)为单位)
}SqList;
```

在上述定义中，数组指针 elem 指示线性表的基地址，length 指示线性表的当前长度。顺序表的初始化操作就是为顺序表分配一个预定义大小的数组空间，并将线性表的当前长度设为"0"(参见算法 2.3)。listsize 指示顺序表当前分配的存储空间大小，一旦因插入

www.TopSage.com

大家网 TopSage.com

元素而空间不足时，可进行再分配，即为顺序表增加一个大小为存储 LISTINCREMENT 个数据元素的空间。

```
Status InitList_Sq(SqList &L) {
    // 构造一个空的线性表 L。
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(ElemType));
    if (! L.elem) exit(OVERFLOW);      // 存储分配失败
    L.length = 0;                      // 空表长度为 0
    L.listsize = LIST_INIT_SIZE;       // 初始存储容量
    return OK;
}// InitList_Sq
```

算法 2.3

在这种存储结构中，容易实现线性表的某些操作，如随机存取第 i 个数据元素等。只是要特别注意的是，C 语言中数组的下标从“0”开始，因此，若 L 是 SqList 类型的顺序表，则表中第 i 个数据元素是 L.elem[i−1]。下面重点讨论线性表的插入和删除两种操作在顺序存储表示时的实现方法。

如 2.1 节中所述，线性表的插入操作是指在线性表的第 $i-1$ 个数据元素和第 i 个数据元素之间插入一个新的数据元素，就是要使长度为 n 的线性表

$$(a_1,\cdots,a_{i-1},a_i,\cdots,a_n)$$

变成长度为 $n+1$ 的线性表

$$(a_1,\cdots,a_{i-1},b,a_i,\cdots,a_n)$$

数据元素 a_{i-1} 和 a_i 之间的逻辑关系发生了变化。在线性表的顺序存储结构中，由于逻辑上相邻的数据元素在物理位置上也是相邻的，因此，除非 $i=n+1$，否则必须移动元素才能反映这个逻辑关系的变化。

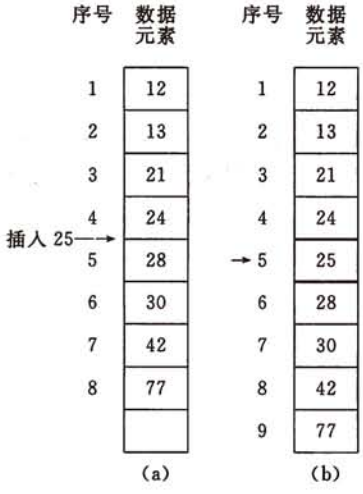


图 2.3 线性表插入前后的状况
(a) 插入前 $n=8$；
(b) 插入后 $n=9$

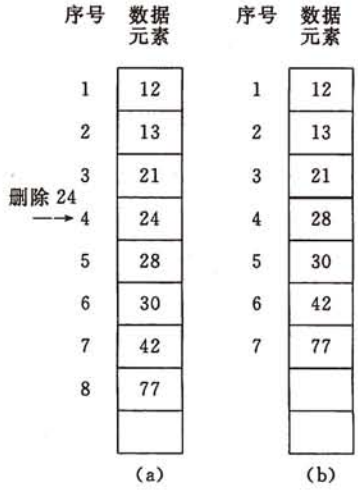


图 2.4 线性表删除前后的状况
(a) 删除前 $n=8$；
(b) 删除后 $n=7$

www.TopSage.com

例如，图 2.3 表示一个线性表在进行插入操作的前、后，其数据元素在存储空间中的位置变化。为了在线性表的第 4 和第 5 个元素之间插入一个值为 25 的数据元素，则需将第 5 个至第 8 个数据元素依次往后移动一个位置。

一般情况下，在第 $i(1\leqslant i\leqslant n)$ 个元素之前插入一个元素时，需将第 n 至第 i（共 $n-i+1$）个元素向后移动一个位置，如算法 2.4 所示。

```
Status ListInsert_Sq(SqList &L, int i, ElemType e) {
    // 在顺序线性表 L 中第 i 个位置之前插入新的元素 e,
    // i 的合法值为 1≤i≤ListLength_Sq(L) + 1
    if (i < 1 || i > L.length + 1) return ERROR;  // i 值不合法
    if (L.length >= L.listsize) {          // 当前存储空间已满,增加分配
      newbase = (ElemType *)realloc(L.elem,
                    (L.listsize + LISTINCREMENT) * sizeof (ElemType));
      if (!newbase)exit(OVERFLOW);         // 存储分配失败
      L.elem = newbase;                    // 新基址
      L.listsize += LISTINCREMENT;         // 增加存储容量
    }
    q = &(L.elem[i - 1]);                  // q 为插入位置
    for (p = &(L.elem[L.length - 1]); p >= q; --p) *(p + 1) = *p;
                                           // 插入位置及之后的元素右移
    *q = e;          // 插入 e
    ++L.length;      // 表长增 1
    return OK;
}// ListInsert_Sq
```

算法 2.4

反之，线性表的删除操作是使长度为 n 的线性表

$$(a_1,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$$

变成长度为 $n-1$ 的线性表

$$(a_1,\cdots,a_{i-1},a_{i+1},\cdots,a_n)$$

数据元素 a_{i-1}、a_i 和 a_{i+1} 之间的逻辑关系发生变化，为了在存储结构上反映这个变化，同样需要移动元素。如图 2.4 所示，为了删除第 4 个数据元素，必须将从第 5 个至第 8 个元素都依次往前移动一个位置。

一般情况下，删除第 $i(1\leqslant i\leqslant n)$ 个元素时需将从第 $i+1$ 至第 n（共 $n-i$）个元素依次向前移动一个位置，如算法 2.5 所示。

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e) {
    // 在顺序线性表 L 中删除第 i 个元素,并用 e 返回其值
    // i 的合法值为 1≤i≤ListLength_Sq(L)
    if ((i < 1) || (i > L.length)) return ERROR;  // i 值不合法
    p = &(L.elem[i - 1]);                         // p 为被删除元素的位置
    e = *p;                                       // 被删除元素的值赋给 e
    q = L.elem + L.length - 1;                    // 表尾元素的位置
    for (++p; p <= q; ++p) *(p - 1) = *p;         // 被删除元素之后的元素左移
```


www.TopSage.com

大家网 TopSage.com

```
    --L.length;                                    // 表长减 1
    return OK;
}// ListDelete_Sq
```

算法 2.5

从算法 2.4 和 2.5 可见，当在顺序存储结构的线性表中某个位置上插入或删除一个数据元素时，其时间主要耗费在移动元素上（换句话说，移动元素的操作为预估算法时间复杂度的基本操作），而移动元素的个数取决于插入或删除元素的位置。

假设 p_i 是在第 i 个元素之前插入一个元素的概率，则在长度为 n 的线性表中插入一个元素时所需移动元素次数的期望值（平均次数）为

$$E_{is} = \sum_{i=1}^{n+1} p_i(n-i+1) \tag{2-3}$$

假设 q_i 是删除第 i 个元素的概率，则在长度为 n 的线性表中删除一个元素时所需移动元素次数的期望值（平均次数）为

$$E_{dl} = \sum_{i=1}^{n} q_i(n-i) \tag{2-4}$$

不失一般性，我们可以假定在线性表的任何位置上插入或删除元素都是等概率的，即

$$p_i = \frac{1}{n+1}, \quad q_i = \frac{1}{n}$$

则式(2-3)和(2-4)可分别简化为式(2-5)和(2-6)：

$$E_{is} = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2} \tag{2-5}$$

$$E_{dl} = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2} \tag{2-6}$$

由式(2-5)和(2-6)可见，在顺序存储结构的线性表中插入或删除一个数据元素，平均约移动表中一半元素。若表长为 n，则算法 ListInsert_Sq 和 ListDelete_Sq 的时间复杂度为 $O(n)$。

现在我们来讨论 2.1 节中例 2-1 和例 2-2 的操作在顺序存储结构的线性表中的实现方法和时间复杂度的分析。容易看出，顺序表的“求表长”和“取第 i 个数据元素的时间复杂度均为 $O(1)$，又这两个例子中进行的“插入”操作均在表尾进行，则不需要移动元素。因此，算法 2.1 的执行时间主要取决于查找函数 LocateElem 的执行时间。在顺序表 L 中查访是否存在和 e 相同的数据元素的最简便的方法是，令 e 和 L 中的数据元素逐个比较之，如算法 2.6 所示。从算法 2.6 中可见。基本操作是“进行两个元素之间的比较”，若 L 中存在和 e 相同的元素 a_i，则比较次数为 $i(1 \leqslant i \leqslant \text{L.length})$，否则为 L.length，即算法 LocateElem_Sq 的时间复杂度为 $O(\text{L.length})$。由此，对于顺序表 La 和 Lb 而言，union 的时间复杂度为 $O(\text{La.length} \times \text{Lb.length})$。

```
int LocateElem_Sq(SqList L, ElemType e,
                  Status ( * compare)(ElemType, ElemType)) {
  // 在顺序线性表 L 中查找第 1 个值与 e 满足 compare()的元素的位序
```


www.TopSage.com

```
    // 若找到,则返回其在 L 中的位序,否则返回 0
    i = 1;              // i 的初值为第 1 个元素的位序
    p = L.elem;         // p 的初值为第 1 个元素的存储位置
    while (i <= L.length && !(*compare)(*p++, e)) ++i;
    if (i <= L.length)  return i;
    else return 0;
}// LocateElem_Sq
```

算法 2.6

对于"顺序表的合并",则从算法 2.2 可直接写出形式上极其相似的算法 2.7。显然,算法 2.7 中的基本操作为"元素赋值",算法的时间复杂度为 $O(La.length+Lb.length)$。

```
void MergeList_Sq(SqList La, SqList Lb, SqList &Lc) {
    // 已知顺序线性表 La 和 Lb 的元素按值非递减排列
    // 归并 La 和 Lb 得到新的顺序线性表 Lc,Lc 的元素也按值非递减排列
    pa = La.elem; pb = Lb.elem;
    Lc.listsize = Lc.length = La.length + Lb.length;
    pc = Lc.elem = (ElemType *)malloc(Lc.listsize * sizeof(ElemType));
    if (!Lc.elem)exit(OVERFLOW);  // 存储分配失败
    pa_last = La.elem + La.length - 1;
    pb_last = Lb.elem + Lb.length - 1;
    while (pa <= pa_last && pb <= pb_last) {     // 归并
      if (*pa <= *pb) *pc++= *pa++;
      else *pc++= *pb++;
    }
    while (pa <= pa_last) *pc++= *pa++;          // 插入 La 的剩余元素
    while (pb <= pb_last) *pc++ = *pb++;         // 插入 Lb 的剩余元素
}  // MergeList_Sq
```

算法 2.7

若对算法 2.7 中第一个循环语句的循环体作如下修改:以"开关语句"代替"条件语句",即分出元素比较的第三种情况,当 *pa= *pb 时,只将两者中之一插入 Lc,则该算法完成的操作和算法 union 完全相同,而时间复杂度却不同。算法 2.7 之所以有线性的时间复杂度,其原因有二:①由于 La 和 Lb 中元素依值递增(同一集合中元素不等),则对 Lb 中每个元素,不需要在 La 中从表头至表尾进行全程搜索;②由于用新表 Lc 表示"并集",则插入操作实际上是借助"复制"操作来完成的[①]。为得到元素依值递增(或递减)的有序表,可利用 10.3 节讨论的快速排序,其时间复杂度为 $O(n\log n)$(其中 n 为待排序的元素个数)。由此可见,若以线性表表示集合并进行集合的各种运算,应先对表中元素进行排序。

---

① 若将 Lb 中元素插入 La,为保持 La 中元素递增有序,必须移动元素(除非插入的元素值大于 La 中所有的元素)。

www.TopSage.com

大家网 TopSage.com

2.3 线性表的链式表示和实现

从上一节的讨论中可见，线性表的顺序存储结构的特点是逻辑关系上相邻的两个元素在物理位置上也相邻，因此可以随机存取表中任一元素，它的存储位置可用一个简单、直观的公式来表示。然而，从另一方面来看，这个特点也铸成了这种存储结构的弱点：在作插入或删除操作时，需移动大量元素。本节我们将讨论线性表的另一种表示方法——链式存储结构，由于它不要求逻辑上相邻的元素在物理位置上也相邻，因此它没有顺序存储结构所具有的弱点，但同时也失去了顺序表可随机存取的优点。

2.3.1 线性链表

线性表的链式存储结构的特点是用一组**任意的**存储单元存储线性表的数据元素（这组存储单元可以是连续的，也可以是不连续的）。因此，为了表示每个数据元素 a_i 与其直接后继数据元素 a_{i+1} 之间的逻辑关系，对数据元素 a_i 来说，除了存储其本身的信息之外，还需存储一个指示其直接后继的信息（即直接后继的存储位置）。这两部分信息组成数据元素 a_i 的存储映像，称为**结点**（node）。它包括两个域：其中存储数据元素信息的域称为**数据域**；存储直接后继存储位置的域称为**指针域**。指针域中存储的信息称做**指针**或**链**。n 个结点（$a_i(1\leqslant i\leqslant n)$ 的存储映像）链结成一个**链表**，即为线性表

$$(a_1,a_2,\cdots,a_n)$$

的**链式存储结构**。又由于此链表的每个结点中只包含一个指针域，故又称**线性链表**或**单链表**。

例如，图 2.5 所示为线性表

(ZHAO,QIAN,SUN,LI,ZHOU,WU,ZHENG,WANG)

的线性链表存储结构，整个链表的存取必须从**头指针**开始进行，头指针指示链表中第一个结点（即第一个数据元素的存储映像）的存储位置。同时，由于最后一个数据元素没有直接后继，则线性链表中最后一个结点的指针为“空”（NULL）。

头指针 H

31

存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25

图 2.5 线性链表示例

用线性链表表示线性表时，数据元素之间的逻辑关系是由结点中的指针指示的。换句话说，指针为数据元素之间的逻辑关系的映像，则逻辑上相邻的两个数据元素其存储的

www.TopSage.com

物理位置不要求紧邻，由此，这种存储结构为非顺序映像或链式映像。

通常我们把链表画成用箭头相链接的结点的序列，结点之间的箭头表示链域中的指针。如图 2.5 的线性链表可画成如图 2.6 所示的形式，这是因为在使用链表时，关心的只是它所表示的线性表中数据元素之间的逻辑顺序，而不是每个数据元素在存储器中的实际位置。

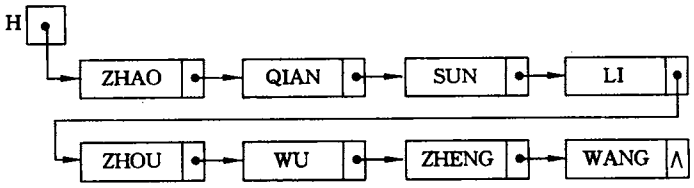


图 2.6　线性链表的逻辑状态

由上述可见，单链表可由头指针惟一确定，在 C 语言中可用“结构指针”来描述。

```
    //- - - - - 线性表的单链表存储结构 - - - - -
typedef struct   LNode {
    ElemType       data;
    struct LNode    * next;
}LNode, * LinkList;
```

假设 L 是 LinkList 型的变量，则 L 为单链表的头指针，它指向表中第一个结点。若 L 为“空”(L＝NULL)，则所表示的线性表为“空”表，其长度 n 为“零”。有时，我们在单链表的第一个结点之前附设一个结点，称之为**头结点**。头结点的数据域可以不存储任何信息，也可存储如线性表的长度等类的附加信息，头结点的指针域存储指向第一个结点的指针(即第一个元素结点的存储位置)。如图 2.7(a)所示，此时，单链表的头指针指向头结点。若线性表为空表，则头结点的指针域为“空”，如图 2.7(b)所示。

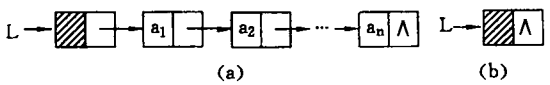


图 2.7　带头结点的单链表
(a) 非空表；(b) 空表

在线性表的顺序存储结构中，由于逻辑上相邻的两个元素在物理位置上紧邻，则每个元素的存储位置都可从线性表的起始位置计算得到。而在单链表中，任何两个元素的存储位置之间没有固定的联系。然而，每个元素的存储位置都包含在其直接前驱结点的信息之中。假设 p 是指向线性表中第 i 个数据元素(结点 a_i)①的指针，则 p－>next 是指向第 i＋1 个数据元素(结点 a_{i+1})的指针。换句话说，若 p－>data＝a_i，则 p－>next－>data＝a_{i+1}。由此，在单链表中，取得第 i 个数据元素必须从头指针出发寻找，因此，单链表是

① 结点 a_i 指其数据域为 a_i 的结点，而 p 结点则指指针 p 所指向的结点(即其存储位置存放在 p 中的结点)。以后均类同。

www.TopSage.com

大家网 TopSage.com

表是非随机存取的存储结构。下面我们看函数 GetElem 在单链表中的实现。

```
Status GetElem_L(LinkList L, int i, ElemType &e) {
    // L为带头结点的单链表的头指针。
    // 当第 i 个元素存在时,其值赋给 e 并返回 OK,否则返回 ERROR
    p = L->next; j = 1;          // 初始化,p 指向第一个结点,j 为计数器
    while (p && j<i) {           // 顺指针向后查找,直到 p 指向第 i 个元素或 p 为空
      p = p->next; ++j;
    }
    if (!p || j>i) return ERROR;  // 第 i 个元素不存在
    e = p->data;                  // 取第 i 个元素
    return OK;
}// GetElem_L
```

算法 2.8

算法 2.8 的基本操作是比较 j 和 i 并后移指针 p，**while** 循环体中的语句频度与被查元素在表中位置有关，若 $1 \leqslant i \leqslant n$，则频度为 $i-1$，否则频度为 n，因此算法 2.8 的时间复杂度为 $O(n)$。

在单链表中，又如何实现“插入”和“删除”操作呢？

假设我们要在线性表的两个数据元素 a 和 b 之间插入一个数据元素 x，已知 p 为其单链表存储结构中指向结点 a 的指针。如图 2.8(a)所示。

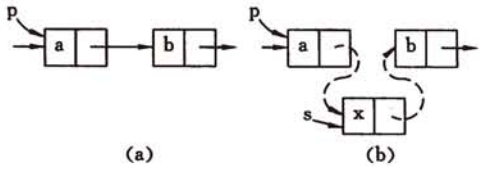


图 2.8 在单链表中插入结点时指针变化状况
(a) 插入前； (b) 插入后

为插入数据元素 x，首先要生成一个数据域为 x 的结点，然后插入在单链表中。根据插入操作的逻辑定义，还需要修改结点 a 中的指针域，令其指向结点 x，而结点 x 中的指针域应指向结点 b，从而实现 3 个元素 a、b 和 x 之间逻辑关系的变化。插入后的单链表如图 2.8(b)所示。假设 s 为指向结点 x 的指针，则上述指针修改用语句描述即为

```
s -> next = p -> next;   p -> next = s;
```

反之，如图 2.9 所示在线性表中删除元素 b 时，为在单链表中实现元素 a、b 和 c 之间逻辑关系的变化，仅需修改结点 a 中的指针域即可。假设 p 为指向结点 a 的指针，则修改指针的语句为

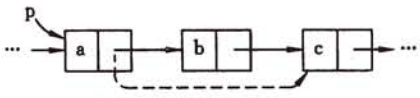


图 2.9 在单链表中删除结点时指针变化状况

```
p -> next = p -> next -> next;
```

可见，在已知链表中元素插入或删除的确切位置的情况下，在单链表中插入或删除一个结点时，仅需修改指针而不需要移动元素。算法 2.9 和算法 2.10 分别为 ListInsert 和 ListDelete 在单链表中的实现。

```
Status ListInsert_L(LinkList &L, int i, ElemType e) {
    // 在带头结点的单链线性表 L 中第 i 个位置之前插入元素 e
    p = L;  j = 0;
    while (p && j < i-1) {p = p->next; ++j;}  // 寻找第 i-1 个结点
```


www.TopSage.com

```
    if (!p||j>i-1) return ERROR;                    // i小于1或者大于表长加1
    s=(LinkList) malloc(sizeof(LNode));             // 生成新结点
    s->data=e; s->next=p->next;                     // 插入L中
    p->next=s;
    return OK;
}// ListInsert L
```

算法 2.9

```
Status ListDelete_L(LinkList &L, int i, ElemType &e) {
    // 在带头结点的单链线性表L中,删除第i个元素,并由e返回其值
    p=L; j=0;
    while(p->next&&j<i-1) {   // 寻找第i个结点,并令p指向其前趋
        p=p->next; ++j;
    }
    if(!(p->next)||j>i-1) return ERROR;   // 删除位置不合理
    q=p->next; p->next=q->next;           // 删除并释放结点
    e=q->data; free(q);
    return OK;
}// ListDelete_L
```

算法 2.10

容易看出,算法 2.9 和算法 2.10 的时间复杂度均为 $O(n)$。这是因为,为在第 i 个结点之前插入一个新结点或删除第 i 个结点,都必须首先找到第 $i-1$ 个结点,即需修改指针的结点,从算法 2.8 的讨论中,我们已经得知,它的时间复杂度为 $O(n)$。

在算法 2.9 和 2.10 中,我们还分别引用了 C 语言中的两个标准函数 malloc 和 free。通常,在设有"指针"数据类型的高级语言中均存在与其相应的过程或函数。假设 p 和 q 是 LinkList 型的变量,则执行 p=(LinkList)**malloc**(**sizeof**(LNode))的作用是由系统生成一个 LNode 型的结点,同时将该结点的起始位置赋给指针变量 p;反之,执行 **free**(q)的作用是由系统回收一个 LNode 型的结点,回收后的空间可以备作再次生成结点时用。因此,单链表和顺序存储结构不同,它是一种动态结构。整个可用存储空间可为多个链表共同享用,每个链表占用的空间不需预先分配划定,而是可以由系统应需求即时生成。因此,建立线性表的链式存储结构的过程就是一个动态生成链表的过程。即从"空表"的初始状态起,依次建立各元素结点,并逐个插入链表。算法 2.11 是一个从表尾到表头逆向建立单链表的算法,其时间复杂度为 $O(n)$。

```
void CreateList_L(LinkList &L, int n) {
    // 逆位序输入n个元素的值,建立带表头结点的单链线性表L。
    L=(LinkList) malloc(sizeof(LNode));
    L->next=NULL;      // 先建立一个带头结点的单链表
    for(i=n; i>0; --i) {
        p=(LinkList) malloc(sizeof(LNode));      // 生成新结点
```


www.TopSage.com

大家网 TopSage.com

```
        scanf(&p->data);                    // 输入元素值
        p->next = L->next; L->next = p;     // 插入到表头
    }
}// CreateList_L
```

算法 2.11

下面讨论如何将两个有序链表并为一个有序链表。

假设头指针为 La 和 Lb 的单链表分别为线性表 LA 和 LB 的存储结构，现要归并 La 和 Lb 得到单链表 Lc，按照 2.1 节中算法 MergeList 的思想，需设立 3 个指针 pa、pb 和 pc，其中 pa 和 pb 分别指向 La 表和 Lb 表中当前待比较插入的结点，而 pc 指向 Lc 表中当前最后一个结点，若 pa->data≤pb->data，则将 pa 所指结点链接到 pc 所指结点之后，否则将 pb 所指结点链接到 pc 所指结点之后。显然，指针的初始状态为：当 LA 和 LB 为非空表时，pa 和 pb 分别指向 La 和 Lb 表中第一个结点，否则为空；pc 指向空表 Lc 中的头结点。由于链表的长度为隐含的，则第一个循环执行的条件是 pa 和 pb 皆非空，当其中一个为空时，说明有一个表的元素已归并完，则只要将另一个表的剩余段链接在 pc 所指结点之后即可。由此得到归并两个单链表的算法，如算法 2.12 所示。

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc) {
    // 已知单链线性表 La 和 Lb 的元素按值非递减排列。
    // 归并 La 和 Lb 得到新的单链线性表 Lc，Lc 的元素也按值非递减排列。
    pa = La->next;  pb = Lb->next;
    Lc = pc = La;                   // 用 La 的头结点作为 Lc 的头结点
    while (pa && pb) {
        if (pa->data <= pb->data) {
            pc->next = pa; pc = pa; pa = pa->next;
        }
        else {pc->next = pb; pc = pb; pb = pb->next; }
    }
    pc->next = pa ? pa : pb;        // 插入剩余段
    free(Lb);                       // 释放 Lb 的头结点
}// MergeList_L
```

算法 2.12

读者容易看出，算法 2.12 的时间复杂度和算法 2.7 相同，但空间复杂度不同。在归并两个链表为一个链表时，不需要另建新表的结点空间，而只需将原来两个链表中结点之间的关系解除，重新按元素值非递减的关系将所有结点链接成一个链表即可。

有时，也可借用一维数组来描述线性链表，其类型说明如下所示：

```
//------ 线性表的静态单链表存储结构 ------
#define  MAXSIZE 1 000  // 链表的最大长度
typedef  struct {
    ElemType  data;
    int       cur;
```


www.TopSage.com

```
}component, SLinkList[MAXSIZE];
```

这种描述方法便于在不设“指针”类型的高级程序设计语言中使用链表结构。在如上描述的链表中，数组的一个分量表示一个结点，同时用游标(指示器 cur)代替指针指示结点在数组中的相对位置。数组的第零分量可看成头结点，其指针域指示链表的第一个结点。例如图 2.10(a)中所示为和图 2.6 相同的线性表。这种存储结构仍需要预先分配一个较大的空间，但在作线性表的插入和删除操作时不需移动元素，仅需修改指针，故仍具有链式存储结构的主要优点。例如，图 2.10(b)展示了图 2.10(a)所示线性表在插入数据元素“SHI”和删除数据元素“ZHENG”之后的状况。为了和指针型描述的线性链表相区别，我们给这种用数组描述的链表起名叫**静态链表**。

0		1
1	ZHAO	2
2	QIAN	3
3	SUN	4
4	LI	5
5	ZHOU	6
6	WU	7
7	ZHENG	8
8	WANG	0
9		
10		

(a)

0		1
1	ZHAO	2
2	QIAN	3
3	SUN	4
4	LI	**9**
5	ZHOU	6
6	WU	**8**
7	ZHENG	8
8	WANG	0
9	**SHI**	**5**
10		

(b)

图 2.10 静态链表示例

(a) 修改前的状态；(b) 修改后的状态

假设 S 为 SLinkList 型变量，则 S[0].cur 指示第一个结点在数组中的位置，若设 i=S[0].cur，则 S[i].data 存储线性表的第一个数据元素，且 S[i].cur 指示第二个结点在数组中的位置。一般情况，若第 i 个分量表示链表的第 k 个结点，则 S[i].cur 指示第k+1个结点的位置。因此在静态链表中实现线性表的操作和动态链表相似，以整型游标 i 代替动态指针 p，i=S[i].cur 的操作实为指针后移(类似于 p=p->next)，例如，在静态链表中实现的定位函数 LocateElem 如算法 2.13 所示。

```
int  LocateElem_SL(SLinkList S, ElemType e) {
    // 在静态单链线性表 L 中查找第 1 个值为 e 的元素。
    // 若找到，则返回它在 L 中的位序，否则返回 0。
    i = S[0].cur;                                  // i 指示表中第一个结点
    while (i && S[i].data != e) i = S[i].cur;      // 在表中顺链查找
    return  i;
}// LocateElem_SL
```

算法 2.13

www.TopSage.com

大家网 TopSage.com

类似地可写出在静态链表中实现插入和删除操作的算法。从图 2.10 的例子可见，指针修改的操作和前面描述的单链表中的插入与删除的算法 2.9、2.10 类似，所不同的是，需由用户自己实现 **malloc** 和 **free** 这两个函数。为了辨明数组中哪些分量未被使用，解决的办法是将所有未被使用过以及被删除的分量用游标链成一个备用的链表，每当进行插入时便可从备用链表上取得第一个结点作为待插入的新结点；反之，在删除时将从链表中删除下来的结点链接到备用链表上。

现以集合运算 $(A-B)\cup(B-A)$ 为例来讨论静态链表的算法。

例 2-3 假设由终端输入集合元素，先建立表示集合 A 的静态链表 S，而后在输入集合 B 的元素的同时查找 S 表，若存在和 B 相同的元素，则从 S 表中删除之，否则将此元素插入 S 表。

为使算法清晰起见，我们先给出 3 个过程：①将整个数组空间初始化成一个链表；②从备用空间取得一个结点；③将空闲结点链结到备用链表上，分别如算法 2.14、2.15 和 2.16 所示。

```
void InitSpace_SL(SLinkList &space) {
    // 将一维数组 space 中各分量链成一个备用链表，space[0].cur 为头指针，
    // "0"表示空指针
    for (i = 0; i < MAXSIZE-1; ++i) space[i].cur = i+1;
    space[MAXSIZE-1].cur = 0;
}// InitSpace_SL
```

算法 2.14

```
    int Malloc_SL(SLinkList &space) {
    // 若备用空间链表非空，则返回分配的结点下标，否则返回 0
    i = space[0].cur;
    if (space[0].cur) space[0].cur = space[i].cur;
    return i;
}// Malloc_SL
```

算法 2.15

```
void Free_SL(SLinkList &space, int k) {
      // 将下标为 k 的空闲结点回收到备用链表
      space[k].cur = space[0].cur; space[0].cur = k;
}// Free_SL
```

算法 2.16

```
void difference(SLinkList &space, int &S) {
    // 依次输入集合 A 和 B 的元素，在一维数组 space 中建立表示集合(A-B)∪(B-A)
    // 的静态链表，S 为其头指针。假设备用空间足够大，space[0].cur 为其头指针。
    InitSpace_SL(space);                  // 初始化备用空间
    S = Malloc_SL(space);                 // 生成 S 的头结点
    r = S;                                // r 指向 S 的当前最后结点
    scanf(m, n);                          // 输入 A 和 B 的元素个数
    for (j = 1; j <= m; ++j) {            // 建立集合 A 的链表
        i = Malloc_SL(space);             // 分配结点
        scanf(space[i].data);             // 输入 A 的元素值
```


www.TopSage.com

```
        space[r].cur = i; r = i;          // 插入到表尾
    }//for
    space[r].cur = 0;                     // 尾结点的指针为空
    for (j = 1; j <= n; ++j) {            // 依次输入 B 的元素,若不在当前表中,则插入,否则
                                          // 删除
        scanf(b); p = S; k = space[S].cur;  // k 指向集合 A 中第一个结点
        while (k != space[r].cur && space[k].data != b) {// 在当前表中查找
            p = k; k = space[k].cur;
        }//while
        if (k == space[r].cur) {          // 当前表中不存在该元素,插入在 r 所指结点之后,且 r
                                          // 的位置不变
          i = Malloc_SL(space);
          space[i].data = b;
          space[i].cur = space[r].cur;
          space[r].cur = i;
        }//if
        else {                            // 该元素已在表中,删除之
          space[p].cur = space[k].cur;
          Free_SL(space, k);
          if (r == k) r = p;              // 若删除的是 r 所指结点,则需修改尾指针
        }//else
    }//for
}// difference
```

算法 2.17

在算法 2.17 中,只有一个处于双重循环中的循环体(在集合 A 中查找依次输入的 b),其最大循环次数为:外循环 n 次,内循环 m 次,故算法 2.17 的时间复杂度为 $O(m\times n)$。

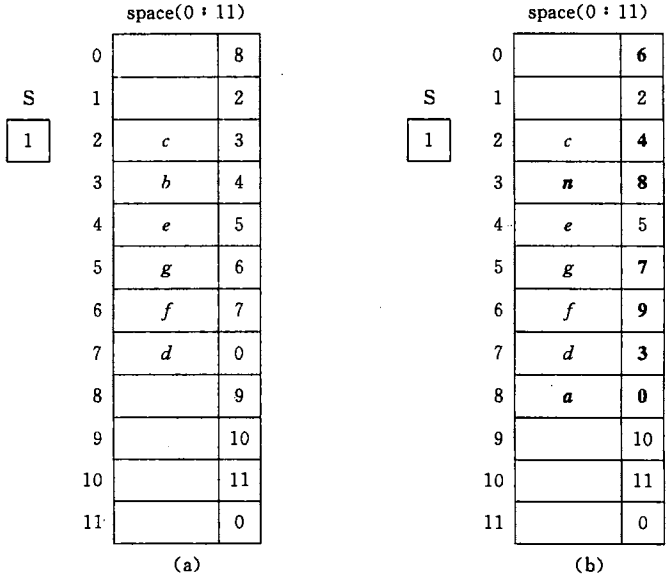


图 2.11 运算前后的静态链表

(a) 表示 A 的链表 S; (b) 表示$(A-B)\cup(B-A)$的链表 S

www.TopSage.com

大家网 TopSage.com

图 2.11 是算法 2.17 执行的示意图。假设集合 $A=(c,b,e,g,f,d)$，$B=(a,b,n,f)$，则图 2.11(a)所示为输入集合 A 的元素之后建成的链表 S 和备用空间链表的状况，图 2.11(b)所示为逐个输入集合 B 的元素并在链表 S 中依次插入 a、删除 b、插入 n、删除 f 后的状况。space[0].cur 为备用链表的头指针，r 的值为 7。

2.3.2 循环链表

循环链表(circular linked list)是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。由此，从表中任一结点出发均可找到表中其他结点，如图 2.12 所示为单链的循环链表。类似地，还可以有多重链的循环链表。

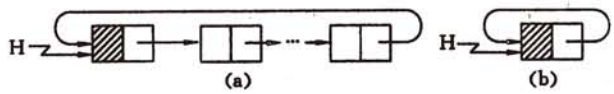


图 2.12 单循环链表

(a) 非空表； (b) 空表

循环链表的操作和线性链表基本一致，差别仅在于算法中的循环条件不是 p 或 p->next是否为空，而是它们是否等于头指针。但有的时候，若在循环链表中设立尾指针而不设头指针(如图2.13(a)所示)，可使某些操作简化。例如将两个线性表合并成一个表时，仅需将一个表的表尾和另一个表的表头相接。当线性表以图 2.13(a)的循环链表作存储结构时，这个操作仅需改变两个指针值即可，运算时间为$O(1)$。合并后的表如图 2.13(b)所示。

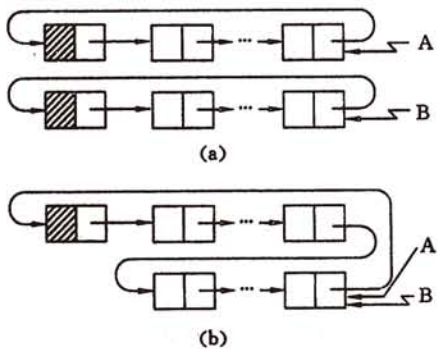


图 2.13 仅设尾指针的循环链表

(a) 两个链表； (b) 合并后的表

2.3.3 双向链表

以上讨论的链式存储结构的结点中只有一个指示直接后继的指针域，由此，从某个结点出发只能顺指针往后寻查其他结点。若要寻查结点的直接前趋，则需从表头指针出发。换句话说，在单链表中，NextElem 的执行时间为 $O(1)$，而 PriorElem 的执行时间为$O(n)$。为克服单链表这种单向性的缺点，可利用**双向链表**(double linked list)。

顾名思义，在双向链表的结点中有两个指针域，其一指向直接后继，另一指向直接前趋，在 C 语言中可描述如下：

```
//------ 线性表的双向链表存储结构 ------
typedef struct  DuLNode {
    ElemType            data;
    struct DuLNode      *prior;
```


www.TopSage.com

```
    struct DuLNode        * next;
}DuLNode, * DuLinkList;
```

和单链的循环表类似，双向链表也可以有循环表，如图 2.14(c)所示，链表中存有两个环，图 2.14(b)所示为只有一个表头结点的空表。在双向链表中，若 d 为指向表中某一

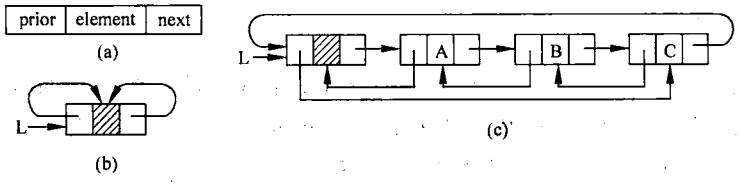


图 2.14　双向链表示例

(a) 结点结构；(b) 空的双向循环链表；(c) 非空的双向循环链表

结点的指针(即 d 为 DuLinkList 型变量)，则显然有

$$d \rightarrow next \rightarrow prior = d \rightarrow prior \rightarrow next = d$$

这个表示式恰当地反映了这种结构的特性。

在双向链表中，有些操作如：ListLength、GetElem 和 LocateElem 等仅需涉及一个方向的指针，则它们的算法描述和线性链表的操作相同，但在插入、删除时有很大的不同，在双向链表中需同时修改两个方向上的指针，图 2.15 和图 2.16 分别显示了删除和插入结点时指针修改的情况。它们的算法分别如算法 2.19 和 2.18 所示，两者的时间复杂度均为 $O(n)$。

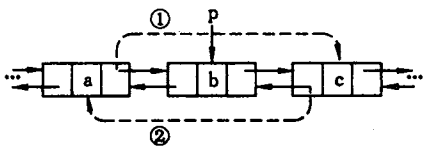


图 2.15　在双向链表中删除结点时指针变化状况

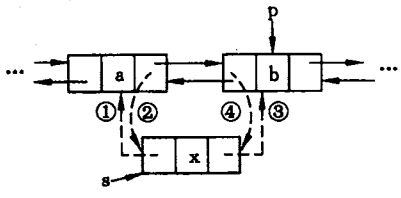


图 2.16　在双向链表中插入一个结点时指针的变化状况

```
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e) {
    // 在带头结点的双链循环线性表 L 中第 i 个位置之前插入元素 e,
    // i 的合法值为 1≤i≤表长+1。
    if (!(p = GetElemP_DuL(L, i)))  // 在 L 中确定插入位置
      return ERROR;                 // p=NULL, 即插入位置不合法
    if (!(s = (DuLinkList)malloc(sizeof(DuLNode)))) return ERROR;
    s->data = e;
    s->prior = p->prior;  p->prior->next = s;
    s->next = p;          p->prior = s;
    return OK;
}// ListInsert_DuL
```

算法　2.18

www.TopSage.com

大家网 TopSage.com

```
Status ListDelete_DuL(DuLinkList &L, int i, ElemType &e) {
    // 删除带头结点的双链循环线性表 L 的第 i 个元素,i 的合法值为 1≤i≤表长
    if (!(p = GetElemP_DuL(L, i)))   // 在 L 中确定第 i 个元素的位置指针 p
        return ERROR;                // p = NULL, 即第 i 个元素不存在
    e = p->data;
    p->prior->next = p->next;
    p->next->prior = p->prior;
    free(p);  return OK;
}// ListDelete_DuL
```

算法 2.19

从本节的讨论中可见,由于链表在空间的合理利用上和插入、删除时不需要移动等的优点,因此在很多场合下,它是线性表的首选存储结构。然而,它也存在着实现某些基本操作,如求线性表的长度时不如顺序存储结构的缺点;另一方面,由于在链表中,结点之间的关系用指针来表示,则数据元素在线性表中的"位序"的概念已淡化,而被数据元素在线性链表中的"位置"所代替。为此,从实际应用角度出发重新定义线性链表及其基本操作。

一个带头结点的线性链表类型定义如下:

```
typedef struct LNode {// 结点类型
    ElemType              data;
    struct LNode          *next;
} *Link, *Position;

typedef struct {          // 链表类型
    Link  head, tail;     // 分别指向线性链表中的头结点和最后一个结点
    int   len;            // 指示线性链表中数据元素的个数
}LinkList;

Status MakeNode( Link &p, ElemType e );
    // 分配由 p 指向的值为 e 的结点,并返回 OK;若分配失败,则返回 ERROR
void  FreeNode( Link &p );
    // 释放 p 所指结点

Status InitList( LinkList &L );
    // 构造一个空的线性链表 L
Status DestroyList( LinkList &L );
    // 销毁线性链表 L,L 不再存在
Status ClearList( LinkList &L );
    // 将线性链表 L 重置为空表,并释放原链表的结点空间
Status InsFirst( Link h, Link s );
    // 已知 h 指向线性链表的头结点,将 s 所指结点插入在第一个结点之前
Status DelFirst( Link h, Link &q );
    // 已知 h 指向线性链表的头结点,删除链表中的第一个结点并以 q 返回
Status Append ( LinkList &L, Link s );
    // 将指针 s 所指(彼此以指针相链)的一串结点链接在线性链表 L 的最后一个结点
    // 之后,并改变链表 L 的尾指针指向新的尾结点
```


www.TopSage.com

```
Status Remove ( LinkList &L, Link &q );
    // 删除线性链表 L 中的尾结点并以 q 返回,改变链表 L 的尾指针指向新的尾结点
Status InsBefore ( LinkList & L, Link &p, Link s );
    // 已知 p 指向线性链表 L 中的一个结点,将 s 所指结点插入在 p 所指结点之前,
    // 并修改指针 p 指向新插入的结点
Status InsAfter ( LinkList &L, Link &p, Link s );
    // 已知 p 指向线性链表 L 中的一个结点,将 s 所指结点插入在 p 所指结点之后,
    // 并修改指针 p 指向新插入的结点
Status SetCurElem ( Link &p, ElemType e);
    // 已知 p 指向线性链表中的一个结点,用 e 更新 p 所指结点中数据元素的值
ElemType GetCurElem ( Link p );
    // 已知 p 指向线性链表中的一个结点,返回 p 所指结点中数据元素的值
Status ListEmpty ( LinkList L );
    // 若线性链表 L 为空表,则返回 TRUE,否则返回 FALSE
int  ListLength( LinkList L );
    // 返回线性链表 L 中元素个数
Position GetHead ( LinkList L );
    // 返回线性链表 L 中头结点的位置
Position GetLast ( LinkList L );
    // 返回线性链表 L 中最后一个结点的位置
Position PriorPos( LinkList L, Link p );
    // 已知 p 指向线性链表 L 中的一个结点,返回 p 所指结点的直接前驱的位置,
    // 若无前驱,则返回 NULL
Position NextPos ( LinkList L, Link p );
    // 已知 p 指向线性链表 L 中的一个结点,返回 p 所指结点的直接后继的位置,
    // 若无后继,则返回 NULL
Status LocatePos ( LinkList L, int i, Link & p ) ;
    // 返回 p 指示线性链表 L 中第 i 个结点的位置并返回 OK,i 值不合法时返回 ERROR
Position LocateElem (LinkList L, ElemType e, Status ( * compare)(ElemType, ElemType));
    // 返回线性链表 L 中第 1 个与 e 满足函数 compare()判定关系的元素的位置,
    // 若不存在这样的元素,则返回 NULL
Status ListTraverse(LinkList L, Status ( * visit)() );
    // 依次对 L 的每个元素调用函数 visit()。一旦 visit()失败,则操作失败。
```

在上述定义的线性链表的基本操作中,除了 DestroyList、ClearList、Remove、InsBefore、PriorPos、LocatePos、LocateElem 和 ListTraverse 的时间复杂度和表长成正比之外,其他操作的时间复杂度都和表长无关,Append 操作的时间复杂度则和插入的结点数成正比。利用这些基本操作,容易实现诸如在第 i 个元素之前插入元素或删除第 i 个元素或合并两个线性表等操作,如算法 2.20 和 2.21 所示。

```
Status ListInsert_L(LinkList &L, int i, ElemType e) {
    // 在带头结点的单链线性表 L 的第 i 个元素之前插入元素 e
    if ( !LocatePos (L, i-1, h))  return ERROR;  // i 值不合法
    if ( !MakeNode (s, e))  return ERROR;        // 结点存储分配失败
    InsFirst (h, s);  // 对于从第 i 个结点开始的链表,第 i-1 个结点是它的头结点
    return OK;
}  // ListInsert_L
```


www.TopSage.com

大家网
TopSage.com

算法 2.20

```
Status MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc
                                    int ( * compare)(ElemType,ElemType)){
  // 已知单链线性表 La 和 Lb 的元素按值非递减排列。
  // 归并 La 和 Lb 得到新的单链线性表 Lc,Lc 的元素也按值非递减排列。
  if (! InitList(Lc))  return ERROR;          // 存储空间分配失败
  ha = GetHead (La);  hb = GetHead (Lb);  // ha 和 hb 分别指向 La 和 Lb 的头结点
  pa = NextPos (La,ha);  pb = NextPos (Lb,hb);  // pa 和 pb 分别指向 La 和 Lb 中当前结点
  while (pa&&pb) {                                  // La 和 Lb 均非空
    a = GetCurElem (pa);  b = GetCurElem (pb);  // a 和 b 为两表中当前比较元素
    if (( * compare)(a,b) <=0) {  //  a≤b
      DelFirst (ha,q);  Append (Lc,q);  pa = NextPos (La,ha);  }
    else {     //  a>b
      DelFirst (hb,q);  Append (Lc,q);  pb = NextPos (Lb,hb);  }
  }//while
  if (pa)  Append (Lc, pa);                        // 链接 La 中剩余结点
  else  Append (Lc,pb);                            // 链接 Lb 中剩余结点
  FreeNode (ha);    FreeNode (hb);                 // 释放 La 和 Lb 的头结点
  return OK;
}// MergeList_L
```

算法 2.21

算法 2.20 和算法 2.21 分别为算法 2.9 和算法 2.12 的改写形式,它们的时间复杂度和前面讨论相同。

2.4 一元多项式的表示及相加

符号多项式的操作,已经成为表处理的典型用例。在数学上,一个一元多项式$P_n(x)$可按升幂写成:

$$P_n(x) = p_0 + p_1x + p_2x^2 + \cdots + p_nx^n$$

它由 $n+1$ 个系数惟一确定。因此,在计算机里,它可用一个线性表 P 来表示:

$$P = (p_0, p_1, p_2, \cdots, p_n)$$

每一项的指数 i 隐含在其系数 p_i 的序号里。

假设 $Q_m(x)$是一元 m 次多项式,同样可用线性表 Q 来表示:

$$Q = (q_0, q_1, q_2, \cdots, q_m)$$

不失一般性,设 $m<n$,则两个多项式相加的结果 $R_n(x)=P_n(x)+Q_m(x)$ 可用线性表 R 表示:

$$R = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \cdots, p_m + q_m, p_{m+1}, \cdots, p_n)$$

显然,我们可以对 P、Q 和 R 采用顺序存储结构,使得多项式相加的算法定义十分简洁。至此,一元多项式的表示及相加问题似乎已经解决了。然而,在通常的应用中,多项式的次数可能很高且变化很大,使得顺序存储结构的最大长度很难确定。特别是在处理

www.TopSage.com

式的次数可能很高且变化很大，使得顺序存储结构的最大长度很难确定。特别是在处理形如

$$S(x) = 1 + 3x^{10\,000} + 2x^{20\,000}$$

的多项式时，就要用一长度为 20 001 的线性表来表示，表中仅有 3 个非零元素，这种对内存空间的浪费是应当避免的，但是如果只存储非零系数项则显然必须同时存储相应的指数。

一般情况下的一元 n 次多项式可写成

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \cdots + p_mx^{e_m} \tag{2-7}$$

其中，p_i 是指数为 e_i 的项的非零系数，且满足

$$0 \leqslant e_1 < e_2 < \cdots < e_m = n$$

若用一个长度为 m 且每个元素有两个数据项(系数项和指数项)的线性表

$$((p_1,e_1),(p_2,e_2),\cdots,(p_m,e_m)) \tag{2-8}$$

便可惟一确定多项式 $P_n(x)$。在最坏情况下，$n+1(=m)$个系数都不为零，则比只存储每项系数的方案要多存储一倍的数据。但是，对于 $S(x)$类的多项式，这种表示将大大节省空间。

对应于线性表的两种存储结构，由式(2-8)定义的一元多项式也可以有两种存储表示方法。在实际的应用程序中取用哪一种，则要视多项式作何种运算而定。若只对多项式进行“求值”等不改变多项式的系数和指数的运算，则采用类似于顺序表的顺序存储结构即可，否则应采用链式存储表示。本节中将主要讨论如何利用线性链表的基本操作来实现一元多项式的运算。

抽象数据类型一元多项式的定义如下：

```
ADT Polynomial {
    数据对象:D={a_i | a_i∈TermSet, i=1,2,…,m,  m≥0
            TermSet 中的每个元素包含一个表示系数的实数和表示指数的整数 }

    数据关系:R1={<a_i-1,a_i> | a_i-1,a_i∈D, 且 a_i-1 中的指数值<a_i 中的指数值,i=2,…,n }
    基本操作:
        CreatPolyn ( &P, m )
      操作结果:输入 m 项的系数和指数,建立一元多项式 P。
        DestroyPolyn ( &P )
      初始条件:一元多项式 P 已存在。
      操作结果:销毁一元多项式 P。
        PrintPolyn ( P )
      初始条件:一元多项式 P 已存在。
      操作结果:打印输出一元多项式 P。
        PolynLength( P )
      初始条件:一元多项式 P 已存在。
      操作结果:返回一元多项式 P 中的项数。
        AddPolyn ( &Pa, &Pb )
      初始条件:一元多项式 Pa 和 Pb 已存在。
      操作结果:完成多项式相加运算,即:Pa = Pa + Pb,并销毁一元多项式 Pb。
        SubtractPolyn ( &Pa, &Pb )
```


www.TopSage.com

大家网
TopSage.com

操作结果：完成多项式相减运算，即：Pa = Pa - Pb，并销毁一元多项式 Pb。

MultiplyPolyn (&Pa, &Pb)

初始条件：一元多项式 Pa 和 Pb 已存在。

操作结果：完成多项式相乘运算，即：Pa = Pa × Pb，并销毁一元多项式 Pb。

}ADT Polynomial

实现上述定义的一元多项式，显然应采用链式存储结构。例如，图 2.17 中的两个线性链表分别表示一元多项式 $A_{17}(x)=7+3x+9x^8+5x^{17}$ 和一元多项式 $B_8(x)=8x+22x^7-9x^8$。从图中可见，每个结点表示多项式中的一项。

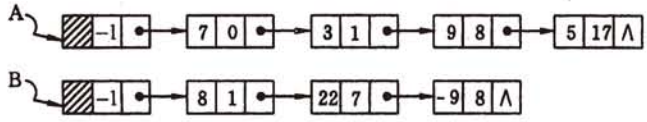


图 2.17　多项式表的单链存储结构

如何实现用这种线性链表表示的多相式的加法运算？

根据一元多项式相加的运算规则：对于两个一元多项式中所有指数相同的项，对应系数相加，若其和不为零，则构成“和多项式”中的一项；对于两个一元多项式中所有指数不相同的项，则分别复抄到“和多项式”中去。

在此，按照上述抽象数据类型 **Polynomial** 中基本操作的定义，“和多项式”链表中的结点无需另生成，而应该从两个多项式的链表中摘取。其运算规则如下：假设指针 qa 和 qb 分别指向多项式 A 和多项式 B 中当前进行比较的某个结点，则比较两个结点中的指数项，有下列 3 种情况：①指针 qa 所指结点的指数值＜指针 qb 所指结点的指数值，则应摘取 qa 指针所指结点插入到“和多项式”链表中去；②指针 qa 所指结点的指数值＞指针 qb 所指结点的指数值，则应摘取指针 qb 所指结点插入到“和多项式”链表中去；③指针 qa 所指结点的指数值＝指针 qb 所指结点的指数值，则将两个结点中的系数相加，若和数不为零，则修改 qa 所指结点的系数值，同时释放 qb 所指结点；反之，从多项式 A 的链表中删除相应结点，并释放指针 qa 和 qb 所指结点。例如，由图 2.17 中的两个链表表示的多项式相加得到的“和多项式”链表如图 2.18 所示，图中的长方框表示已被释放的结点。

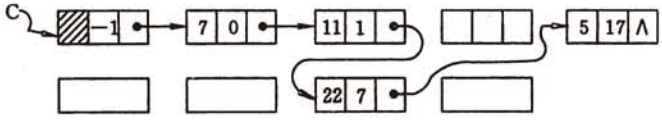


图 2.18　相加得到的和多项式

上述多项式的相加过程和上一节讨论的归并两个有序表的过程极其类似，不同之处仅在于，后者在比较数据元素时只出现两种情况。因此，多项式相加的过程也完全可以利用线性链表的基本操作来完成。

需要附加说明的是，在 2.3 节末定义的线性链表类型适用于一般的线性表，而表示一元多项式的应该是有序链表。有序链表的基本操作定义与线性链表有两处不同，一是

www.TopSage.com

元多项式的应该是有序链表。有序链表的基本操作定义与线性链表有两处不同，一是LocateElem的职能不同，二是需增加按有序关系进行插入的操作OrderInsert，现说明如下：

```
Status LocateElem (LinkList L, ElemType e, Position &q,
                                        int ( * compare)(ElemType, ElemType));
    // 若有序链表 L 中存在与 e 满足判定函数 compare()取值为 0 的元素，则 q 指示 L 中第一个
    // 值为 e 的结点的位置，并返回 TRUE；否则 q 指示第一个与 e 满足判定函数 compare()取
    // 值>0 的元素的前驱的位置，并返回 FALSE
Status OrderInsert ( LinkList &L, ElemType e, int ( * compare)(ElemType, ElemType));
    // 按有序判定函数 compare()的约定，将值为 e 的结点插入到有序链表 L 的适当位置上
```

例 2-4　抽象数据类型 Polynomial 的实现。

```
typedef struct {  // 项的表示，多项式的项作为 LinkList 的数据元素
    float  coef;       // 系数
    int    expn;       // 指数
}term, ElemType;  // 两个类型名：term 用于本 ADT，ElemType 为 LinkList 的数据对象名

typedef LinkList polynomial;        // 用带表头结点的有序链表表示多项式

      //------ 基本操作的函数原型说明 ------
void CreatPolyn ( polynomail &P, int m );
    // 输入 m 项的系数和指数，建立表示一元多项式的有序链表 P
void DestroyPolyn ( polynomail &P );
    // 销毁一元多项式 P
void PrintPolyn ( polynomail P );
    // 打印输出一元多项式 P
int PolynLength( polynomail P );
    // 返回一元多项式 P 中的项数
void AddPolyn ( polynomail &Pa, polynomail &Pb );
    // 完成多项式相加运算，即：Pa = Pa + Pb，并销毁一元多项式 Pb
void SubtractPolyn ( polynomail &Pa, polynomail &Pb );
    // 完成多项式相减运算，即：Pa = Pa - Pb，并销毁一元多项式 Pb
void MultiplyPolyn ( polynomail &Pa, polynomail &Pb );
    // 完成多项式相乘运算，即：Pa = Pa × Pb，并销毁一元多项式 Pb

    //------ 基本操作的算法描述(部分) ------
int cmp ( term a, term b );
    // 依 a 的指数值<(或=)(或>)b 的指数值，分别返回 -1、0 和+1

void CreatPolyn ( polynomail &P, int m ) {
    // 输入 m 项的系数和指数，建立表示一元多项式的有序链表 P
    InitList (P);     h = GetHead (P);
    e.coef = 0.0;  e.expn = -1;  SetCurElem (h, e);  // 设置头结点的数据元素
    for ( i=1; i<=m; ++i ) {  // 依次输入 m 个非零项
      scanf (e.coef, e.expn);
      if (!LocateElem ( P, e, q, ( * cmp)()) ) {  // 当前链表中不存在该指数项
        if (MakeNode (s,e))  InsFirst ( q, s );  // 生成结点并插入链表
      }
    }
```


www.TopSage.com

大家网
TopSage.com

```
}// CreatPolyn
```

算法 2.22

```
void AddPolyn ( polynomial &Pa, polynomial &Pb)  {
    // 多项式加法:Pa = Pa + Pb,利用两个多项式的结点构成"和多项式"。
    ha = GetHead (Pa);  hb = GetHead (Pb);  // ha 和 hb 分别指向 Pa 和 Pb 的头结点
    qa = NextPos (Pa,ha);  qb = NextPos (Pb,hb);  // qa 和 qb 分别指向 Pa 和 Pb 中当前结点
    while (qa && qb) {  // qa 和 qb 均非空
      a = GetCurElem (qa);  b = GetCurElem (qb);  // a 和 b 为两表中当前比较元素
      switch ( *cmp(a,b)) {
        case -1:  // 多项式 PA 中当前结点的指数值小
          ha = qa;  qa = NextPos (Pa, qa);  break;
        case 0:     // 两者的指数值相等
            sum = a.coef + b.coef ;
            if ( sum != 0.0 ) {  // 修改多项式 PA 中当前结点的系数值
              SetCurElem (qa,sum) ;  ha = qa;  }
            else {  // 删除多项式 PA 中当前结点
                DelFirst (ha, qa);  FreeNode (qa);  }
            DelFirst (hb, qb);  FreeNode (qb);  qb = NextPos (Pb, hb);
            qa = NextPos (Pa, ha);  break;
        case 1:     // 多项式 PB 中当前结点的指数值小
            DelFirst (hb, qb);  InsFirst( ha, qb);
            qb = NextPos (Pb, hb);  ha = NextPos(Pa,ha);break;
      }//switch
    }//while
    if (!ListEmpty (Pb))  Append (Pa, qb);     // 链接 Pb 中剩余结点
    FreeNode (hb);  // 释放 Pb 的头结点
}// AddPolyn
```

算法 2.23

两个一元多项式相乘的算法,可以利用两个一元多项式相加的算法来实现,因为乘法运算可以分解为一系列的加法运算。假设 $A(x)$和 $B(x)$为式(2-7)的多项式,则

$$\begin{aligned} M(x) &= A(x) \times B(x) \\ &= A(x) \times [b_1 x^{e_1} + b_2 x^{e_2} + \cdots + b_n x^{e_n}] \\ &= \sum_{i=1}^{n} b_i A(x) x^{e_i} \end{aligned}$$

其中,每一项都是一个一元多项式。

www.TopSage.com

第3章 栈和队列

栈和队列是两种重要的线性结构。从数据结构角度看，栈和队列也是线性表，其特殊性在于栈和队列的基本操作是线性表操作的子集，它们是操作受限的线性表，因此，可称为限定性的数据结构。但从数据类型角度看，它们是和线性表大不相同的两类重要的抽象数据类型。由于它们广泛应用在各种软件系统中，因此在面向对象的程序设计中，它们是多型数据类型。本章除了讨论栈和队列的定义、表示方法和实现外，还将给出一些应用的例子。

3.1 栈

3.1.1 抽象数据类型栈的定义

栈(stack) 是限定仅在表尾进行插入或删除操作的线性表。因此，对栈来说，表尾端有其特殊含义，称为**栈顶**(top)，相应地，表头端称为**栈底**(bottom)。不含元素的空表称为空栈。

假设栈 $S=(a_1,a_2,\cdots,a_n)$，则称 a_1 为栈底元素，a_n 为栈顶元素。栈中元素按 a_1，$a_2,\cdots,a_n$ 的次序进栈，退栈的第一个元素应为栈顶元素。换句话说，栈的修改是按后进先出的原则进行的(如图 3.1(a)所示)。因此，栈又称为**后进先出**(last in first out)的线性表(简称 **LIFO** 结构)，它的这个特点可用图 3.1(b)所示的铁路调度站形象地表示。

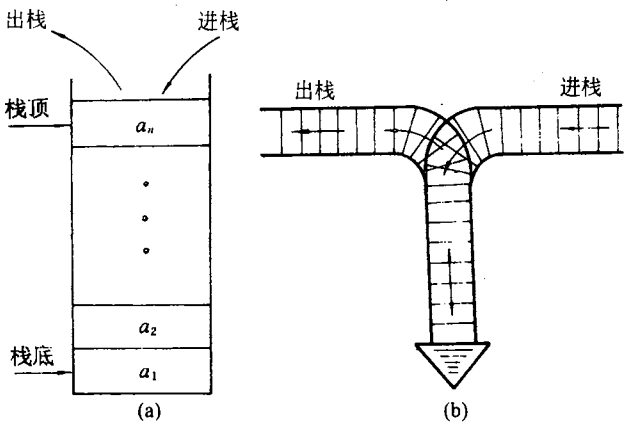


图 3.1 栈

(a) 栈的示意图； (b) 用铁路调度站表示栈

栈的基本操作除了在栈顶进行插入或删除外，还有栈的初始化、判空及取栈顶元素等。下面给出栈的抽象数据类型的定义：

www.TopSage.com

大家网
TopSage.com

```
ADT Stack {
    数据对象: D = { a_i | a_i ∈ ElemSet, i = 1,2,…,n,  n≥0 }
    数据关系: R1 = { <a_{i-1}, a_i> | a_{i-1}, a_i ∈ D, i = 2,…,n }
              约定 a_n 端为栈顶,a_1 端为栈底。
    基本操作:
        InitStack(&S)
      操作结果:构造一个空栈 S。
        DestroyStack(&S)
      初始条件:栈 S 已存在。
      操作结果:栈 S 被销毁。
        ClearStack(&S)
      初始条件:栈 S 已存在。
      操作结果:将 S 清为空栈。
        StackEmpty(S)
      初始条件:栈 S 已存在。
      操作结果:若栈 S 为空栈,则返回 TRUE,否则 FALSE。
        StackLength(S)
      初始条件:栈 S 已存在。
      操作结果:返回 S 的元素个数,即栈的长度。
        GetTop(S, &e)
      初始条件:栈 S 已存在且非空。
      操作结果:用 e 返回 S 的栈顶元素。
        Push(&S, e)
      初始条件:栈 S 已存在。
      操作结果:插入元素 e 为新的栈顶元素。
        Pop(&S, &e)
      初始条件:栈 S 已存在且非空。
      操作结果:删除 S 的栈顶元素,并用 e 返回其值。
        StackTraverse(S,visit())
      初始条件:栈 S 已存在且非空。
      操作结果:从栈底到栈顶依次对 S 的每个数据元素调用函数 visit()。一旦 visit()失败,
则操作失效。
}ADT Stack
```

本书在以后各章中引用的栈大多为如上定义的数据类型,栈的数据元素类型在应用程序内定义,并称插入元素的操作为入栈,删除栈顶元素的操作为出栈。

3.1.2 栈的表示和实现

和线性表类似,栈也有两种存储表示方法。

顺序栈,即栈的顺序存储结构是利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素,同时附设指针 top 指示栈顶元素在顺序栈中的位置。通常的习惯做法是以 top=0 表示空栈,鉴于 C 语言中数组的下标约定从 0 开始,则当以 C 作描述语言时,如此设定会带来很大不便;另一方面,由于栈在使用过程中所需最大空间的大小很难估计,因此,一般来说,在初始化设空栈时不应限定栈的最大容量。一个较合理的做法是:先为栈分配一个基本容量,然后在应用过程中,当栈的空间不够使用时再逐段扩大。为此,可

www.TopSage.com

设定两个常量:STACK_INIT_SIZE(存储空间初始分配量)和 STACKINCREMENT(存储空间分配增量),并以下述类型说明作为顺序栈的定义。

```
typedef struct {
  SElemType  * base;
  SElemType  * top;
  int    stacksize;
}SqStack;
```

其中,stacksize 指示栈的当前可使用的最大容量。栈的初始化操作为:按设定的初始分配量进行第一次存储分配,base 可称为栈底指针,在顺序栈中,它始终指向栈底的位置,若 base 的值为 **NULL**,则表明栈结构不存在。称 top 为栈顶指针,其初值指向栈底,即 top=base 可作为栈空的标记,每当插入新的栈顶元素时,指针 top 增 1;删除栈顶元素时,指针 top 减 1,因此,非空栈中的栈顶指针始终在栈顶元素的下一个位置上。图 3.2 展示了顺序栈中数据元素和栈顶指针之间的对应关系。

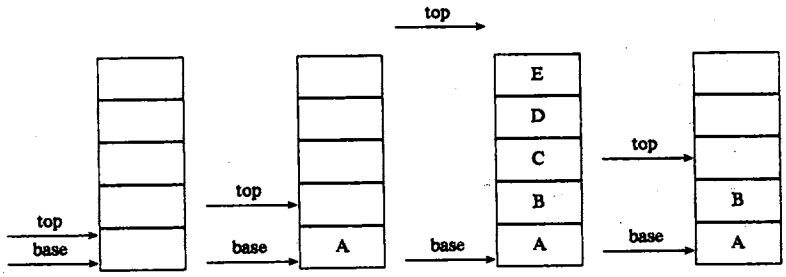


图 3.2 栈顶指针和栈中元素之间的关系

以下是顺序栈的模块说明。

```
// ===== ADT Stack 的表示与实现 =====

    // ----- 栈的顺序存储表示 -----

#define  STACK_INIT_SIZE  100;      // 存储空间初始分配量
#define  STACKINCREMENT    10;      // 存储空间分配增量
typedef struct {
    SElemType  * base;  //在栈构造之前和销毁之后,base 的值为 NULL
    SElemType  * top;   //栈顶指针
    int  stacksize;     //当前已分配的存储空间,以元素为单位
}SqStack;
// ----- 基本操作的函数原型说明 -----
Status InitStack (SqStack &S);
    // 构造一个空栈 S
Status DestroyStack (SqStack &S);
    // 销毁栈 S,S 不再存在
Status ClearStack (SqStack &S);
    // 把 S 置为空栈
Status StackEmpty (SqStack S);
    // 若栈 S 为空栈,则返回 TRUE,否则返回 FALSE
```


www.TopSage.com

大家网
TopSage.com

```
int StackLength (SqStack S);
    // 返回 S 的元素个数,即栈的长度
Status GetTop (SqStack S, SElemType &e);
    // 若栈不空,则用 e 返回 S 的栈顶元素,并返回 OK;否则返回 ERROR
Status Push (SqStack &S, SElemType e);
    // 插入元素 e 为新的栈顶元素
Status Pop (SqStack &S, SElemType &e);
    // 若栈不空,则删除 S 的栈顶元素,用 e 返回其值,并返回 OK;否则返回 ERROR
Status StackTraverse(SqStack S, Status ( * visit)());
    // 从栈底到栈顶依次对栈中每个元素调用函数 visit()。一旦 visit()失败,则操作失败

    // ------ 基本操作的算法描述(部分) ------
Status InitStack (SqStack &S) {
    // 构造一个空栈 S
    S.base = (SElemType * )malloc(STACK_INIT_SIZE * sizeof(SElemType));
    if (!S.base) exit (OVERFLOW);     //存储分配失败
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    return OK;
}// InitStack

Status GetTop(SqStack S, SElemType &e) {
  // 若栈不空,则用 e 返回 S 的栈顶元素,并返回 OK;否则返回 ERROR
  if (S.top == S.base)  return ERROR;
    e = * (S.top - 1);
    return OK;
}// GetTop

Status Push (SqStack &S, SElemType e) {
    // 插入元素 e 为新的栈顶元素
    if (S.top - S.base >= S.stacksize) {//栈满,追加存储空间
      S.base = (SElemType * ) realloc ( S.base,
                (S.stacksize + STACKINCREMENT) * sizeof (SElemType));
      if (!S.base) exit (OVERFLOW); //存储分配失败
      S.top = S.base + S.stacksize;
      S.stacksize += STACKINCREMENT;
    }
    * S.top++ = e;
    return OK;
}//Push

Status Pop (SqStack &S, SElemType &e) {
    // 若栈不空,则删除 S 的栈顶元素,用 e 返回其值,并返回
       OK;否则返回 ERROR
    if (S.top == S.base) return ERROR;
    e = * --S.top;
    return OK;
}// Pop
```

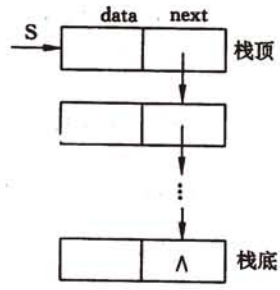


图 3.3 链栈示意图

www.TopSage.com

栈的链式表示——链栈如图 3.3 所示。由于栈的操作是线性表操作的特例，则链栈的操作易于实现，在此不作详细讨论。

3.2 栈的应用举例

由于栈结构具有后进先出的固有特性，致使栈成为程序设计中的有用工具。本节将讨论几个栈应用的典型例子。

3.2.1 数制转换

十进制数 N 和其他 d 进制数的转换是计算机实现计算的基本问题，其解决方法很多，其中一个简单算法基于下列原理：

$N=(N \text{ div } d)\times d+N \bmod d$（其中：div 为整除运算，mod 为求余运算）

例如：$(1348)_{10}=(2504)_8$，其运算过程如下：

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

假设现要编制一个满足下列要求的程序：对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数。由于上述计算过程是从低位到高位顺序产生八进制数的各个数位，而打印输出，一般来说应从高位到低位进行，恰好和计算过程相反。因此，若将计算过程中得到的八进制数的各位顺序进栈，则按出栈序列打印输出的即为与输入对应的八进制数。

```
void conversion () {
  // 对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数
  InitStack(S);        // 构造空栈
  scanf ("%d",N);
  while (N) {
    Push(S, N % 8);
    N = N/8;
  }
  while (!StackEmpty(s)) {
    Pop(S,e);
    printf ( "%d", e );
  }
}// conversion
```

算法 3.1

这是利用栈的后进先出特性的最简单的例子。在这个例子中，栈操作的序列是直线

www.TopSage.com

大家网 TopSage.com

式的，即先一味地入栈，然后一味地出栈。也许，有的读者会提出疑问：用数组直接实现不也很简单吗？仔细分析上述算法不难看出，栈的引入简化了程序设计的问题，划分了不同的关注层次，使思考范围缩小了。而用数组不仅掩盖了问题的本质，还要分散精力去考虑数组下标增减等细节问题。

3.2.2 括号匹配的检验

假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，即([]())或[([][])]等为正确的格式，[(])或([())或(())均为不正确的格式。检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。例如考虑下列括号序列：

```
[([][])]
1 2 3 4 5 6 7 8
```

当计算机接受了第一个括号后，它期待着与其匹配的第八个括号的出现，然而等来的却是第二个括号，此时第一个括号“[”只能暂时靠边，而迫切等待与第二个括号相匹配的、第七个括号“)”的出现，类似地，因等来的是第三个括号“[”，其期待匹配的程度较第二个括号更急迫，则第二个括号也只能靠边，让位于第三个括号，显然第二个括号的期待急迫性高于第一个括号；在接受了第四个括号之后，第三个括号的期待得到满足，消解之后，第二个括号的期待匹配就成为当前最急迫的任务了，……，依次类推。可见，这个处理过程恰与栈的特点相吻合。由此，在算法中设置一个栈，每读入一个括号，若是右括号，则或者使置于栈顶的最急迫的期待得以消解，或者是不合法的情况；若是左括号，则作为一个新的更急迫的期待压入栈中，自然使原有的在栈中的所有未消解的期待的急迫性都降了一级。另外，在算法的开始和结束时，栈都应该是空的。此算法将留给读者作为习题完成。

3.2.3 行编辑程序

一个简单的行编辑程序的功能是：接受用户从终端输入的程序或数据，并存入用户的数据区。由于用户在终端上进行输入时，不能保证不出差错，因此，若在编辑程序中，“每接受一个字符即存入用户数据区”的做法显然不是最恰当的。较好的做法是，设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入出差错，并在发现有误时可以及时更正。例如，当用户发现刚刚键入的一个字符是错的时，可补进一个退格符“#”，以表示前一个字符无效；如果发现当前键入的行内差错较多或难以补救，则可以键入一个退行符“@”，以表示当前行中的字符均无效。例如，假设从终端接受了这样两行字符：

```
whli# #ilr#e(s# *s)
  outcha@putchar(*s= # ++);
```

则实际有效的是下列两行：

```
while (*s)
  putchar(*s++);
```

为此，可设这个输入缓冲区为一个栈结构，每当从终端接受了一个字符之后先作如下

www.TopSage.com

判别：如果它既不是退格符也不是退行符，则将该字符压入栈顶；如果是一个退格符，则从栈顶删去一个字符；如果它是一个退行符，则将字符栈清为空栈。上述处理过程可用算法3.2描述。

```
void LineEdit() {
  //利用字符栈 S,从终端接收一行并传送至调用过程的数据区。
  InitStack(S);                //构造空栈 S
  ch = getchar();              //从终端接收第一个字符
  while (ch != EOF) {//EOF 为全文结束符
    while (ch != EOF && ch != '\n') {
      switch (ch) {
        case '#': Pop(S, c);       break; //仅当栈非空时退栈
        case '@': ClearStack(S);   break; //重置 S 为空栈
        default : Push(S, ch);     break; //有效字符进栈,未考虑栈满情形
      }
      ch = getchar();  // 从终端接收下一个字符
    }
    将从栈底到栈顶的栈内字符传送至调用过程的数据区;
    ClearStack(S);        // 重置 S 为空栈
    if (ch != EOF)  ch = getchar();
  }
  DestroyStack(S);
}// LineEdit
```

算法 3.2

3.2.4 迷宫求解

求迷宫中从入口到出口的所有路径是一个经典的程序设计问题。由于计算机解迷宫时，通常用的是“穷举求解”的方法，即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止。为了保证在任何位置上都能沿原路退回，显然需要用一个后进先出的结构来保存从入口到当前位置的路径。因此，在求迷宫通路的算法中应用“栈”也就是自然而然的事了。

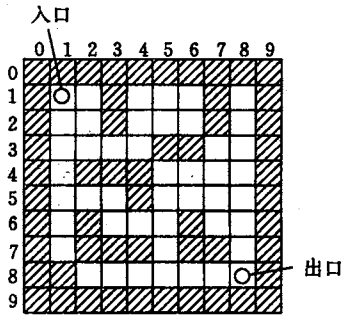


图 3.4 迷宫

首先，在计算机中可以用如图3.4所示的方块图表示迷宫。图中的每个方块或为通道（以空白方块表示），或为墙（以带阴影线的方块表示）。所求路径必须是简单路径，即在求得的路径上不能重复出现同一通道块。

假设“当前位置”指的是“在搜索过程中某一时刻所在图中某个方块位置”，则求迷宫中一条路径的算法的基本思想是：若当前位置“可通”，则纳入“当前路径”，并继续朝“下一位置”探索，即切换“下一位置”为“当前位置”，如此重复直至到达出口；若当前位置“不可

www.TopSage.com

大家网 TopSage.com

通”，则应顺着“来向”退回到“前一通道块”，然后朝着除“来向”之外的其他方向继续探索；若该通道块的四周 4 个方块均“不可通”，则应从“当前路径”上删除该通道块。所谓“下一位置”指的是“当前位置”四周 4 个方向（东、南、西、北）上相邻的方块。假设以栈 S 记录“当前路径”，则栈顶中存放的是“当前路径上最后一个通道块”。由此，“纳入路径”的操作即为“当前位置入栈”；“从当前路径上删除前一通道块”的操作即为“出栈”。

求迷宫中一条从入口到出口的路径的算法可简单描述如下：

```
设定当前位置的初值为入口位置；
do {
  若当前位置可通，
  则{  将当前位置插入栈顶；                    // 纳入路径
       若该位置是出口位置，则结束；             // 求得路径存放在栈中
       否则切换当前位置的东邻方块为新的当前位置；
    }
  否则，
      若栈不空且栈顶位置尚有其他方向未经探索，
        则设定新的当前位置为沿顺时针方向旋转找到的栈顶位置的下一相邻块；
      若栈不空但栈顶位置的四周均不可通，
        则{  删去栈顶位置；                     // 从路径中删去该通道块
             若栈不空，则重新测试新的栈顶位置，
               直至找到一个可通的相邻块或出栈至栈空；
          }
}while (栈不空)；
```

在此，尚需说明一点的是，所谓当前位置**可通**，指的是**未曾走到过的通道块**，即要求该方块位置不仅是通道块，而且既不在当前路径上（否则所求路径就不是简单路径），也不是曾经纳入过路径的通道块（否则只能在死胡同内转圈）。

```
typedef struct {
  int         ord;          // 通道块在路径上的"序号"
  PosType     seat;         // 通道块在迷宫中的"坐标位置"
  int         di;           // 从此通道块走向下一通道块的"方向"
}SElemType;                 // 栈的元素类型

Status MazePath ( MazeType maze, PosType start, PosType end ) {
  // 若迷宫 maze 中存在从入口 start 到出口 end 的通道，则求得一条存放在栈中（从栈底到栈
  // 顶），并返回 TRUE；否则返回 FALSE
    InitStack(S);  curpos = start;        // 设定"当前位置"为"入口位置"
    curstep = 1;                          // 探索第一步
    do {
      if (Pass (curpos)) {  // 当前位置可以通过，即是未曾走到过的通道块
          FootPrint (curpos);             // 留下足迹
          e = ( curstep, curpos, 1 );
          Push (S,e);                     // 加入路径
          if (curpos == end)  return (TRUE);  // 到达终点（出口）
          curpos = NextPos ( curpos, 1 );     // 下一位置是当前位置的东邻
          curstep++ ;                         // 探索下一步
```


www.TopSage.com

```
    }// if
    else {  // 当前位置不能通过
    if (!StackEmpty(S)) {
      Pop (S,e);
      while (e.di == 4 && !StackEmpty(S)) {
        MarkPrint (e.seat);  Pop (S,e);        // 留下不能通过的标记,并退回一步
      }// while
      if (e.di<4) {
        e.di++;   Push ( S, e);                // 换下一个方向探索
        curpos = NextPos (e.seat e.di );       // 设定当前位置是该新方向上的相邻块
      }// if
    }// if
  }// else
 }while ( !StackEmpty(S) );
 return (FALSE);
}// MazePath
```

算法 3.3

3.2.5 表达式求值

表达式求值是程序设计语言编译中的一个最基本问题。它的实现是栈应用的又一个典型例子。这里介绍一种简单直观、广为使用的算法，通常称为“算符优先法”。

要把一个表达式翻译成正确求值的一个机器指令序列，或者直接对表达式求值，首先要能够正确解释表达式。例如，要对下面的算术表达式求值：

$$4+2\times3-10/5$$

首先要了解算术四则运算的规则。即：

(1) 先乘除，后加减；

(2) 从左算到右；

(3) 先括号内，后括号外。

由此，这个算术表达式的计算顺序应为

$$4+2\times3-10/5=4+6-10/5=10-10/5=10-2=8$$

算符优先法就是根据这个运算优先关系的规定来实现对表达式的编译或解释执行的。

任何一个表达式都是由操作数(operand)、运算符(operator)和界限符(delimiter)组成的，我们称它们为单词。一般地，操作数既可以是常数也可以是被说明为变量或常量的标识符；运算符可以分为算术运算符、关系运算符和逻辑运算符 3 类；基本界限符有左右括号和表达式结束符等。为了叙述的简洁，我们仅讨论简单算术表达式的求值问题。这种表达式只含加、减、乘、除 4 种运算符。读者不难将它推广到更一般的表达式上。

我们把运算符和界限符统称为算符，它们构成的集合命名为 OP。根据上述 3 条运算规则，在运算的每一步中，任意两个相继出现的算符 θ_1 和 θ_2 之间的优先关系至多是下面 3 种关系之一；

$\theta_1<\theta_2$　θ_1 的优先权低于 θ_2

www.TopSage.com

大家网
TopSage.com

$\theta_1=\theta_2$　θ_1 的优先权等于 θ_2

$\theta_1>\theta_2$　θ_1 的优先权高于 θ_2

表 3.1 定义了算符之间的这种优先关系。

表 3.1　算符间的优先关系

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	≐	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

由规则(3)，+、−、* 和/为 θ_1 时的优先性均低于“(”但高于“)”，由规则(2)，当 $\theta_1=\theta_2$ 时，令 $\theta_1>\theta_2$，“#”是表达式的结束符。为了算法简洁，在表达式的最左边也虚设一个“#”构成整个表达式的一对括号。表中的“(”=“)”表示当左右括号相遇时，括号内的运算已经完成。同理，“#”=“#”表示整个表达式求值完毕。“)”与“(”、“#”与“)”以及“(”与“#”之间无优先关系，这是因为表达式中不允许它们相继出现，一旦遇到这种情况，则可以认为出现了语法错误。在下面的讨论中，我们暂假定所输入的表达式不会出现语法错误。

为实现算符优先算法，可以使用两个工作栈。一个称做 OPTR，用以寄存运算符；另一个称做 OPND，用以寄存操作数或运算结果。算法的基本思想是：

(1) 首先置操作数栈为空栈，表达式起始符“#”为运算符栈的栈底元素；

(2) 依次读入表达式中每个字符，若是操作数则进 OPND 栈，若是运算符则和 OPTR 栈的栈顶运算符比较优先权后作相应操作，直至整个表达式求值完毕(即 OPTR 栈的栈顶元素和当前读入的字符均为“#”)。

算法 3.4 描述了这个求值过程。

```
OperandType  EvaluateExpression() {
  // 算术表达式求值的算符优先算法。设 OPTR 和 OPND 分别为运算符栈和运算数栈，
  // OP 为运算符集合。
   InitStack (OPTR);   Push (OPTR,'#');
   initStack (OPND);   c = getchar();
   while (c!='#' || GetTop(OPTR)!='#') {
     if (!In(c, OP)){Push((OPND, c); c = getchar(); }     // 不是运算符则进栈
     else
       switch (Precede(GetTop(OPTR), c)) {
         case '<':   // 栈顶元素优先权低
                Push(OPTR, c);  c = getchar();
                break;
         case '=':   // 脱括号并接收下一字符
                Pop(OPTR, x);   c = getchar();
                break;
```


www.TopSage.com

```
            case '>':    // 退栈并将运算结果入栈
                Pop(OPTR, theta);
                Pop(OPND, b);  Pop(OPND, a);
                Push(OPND, Operate(a, theta, b));
                break;
            }// switch
        }// while
        return GetTop(OPND);
    }// EvaluateExpression
```

算法 3.4

算法中还调用了两个函数。其中 Precede 是判定运算符栈的栈顶运算符 θ_1 与读入的运算符 θ_2 之间优先关系的函数；Operate 为进行二元运算 $a\ \theta\ b$ 的函数，如果是编译表达式，则产生这个运算的一组相应指令并返回存放结果的中间变量名；如果是解释执行表达式，则直接进行该运算，并返回运算的结果。

例 3-1 利用算法 EvaluateExpression_reduced 对算术表达式 3 * (7－2)求值，操作过程如下所示。

步骤	OPTR 栈	OPND 栈	输入字符	主 要 操 作
1	#		3 * (7－2) #	PUSH(OPND,'3')
2	#	3	* (7－2) #	PUSH(OPTR,' * ')
3	# *	3	(7－2) #	PUSH(OPTR,'(')
4	# * (	3	7－2) #	PUSH(OPND,'7')
5	# * (	3 7	－2) #	PUSH(OPTR,'－')
6	# * (－	3 7	2) #	PUSH(OPND,'2')
7	# * (－	3 7 2	) #	operate('7','－','2')
8	# * (	3 5	) #	POP(OPTR){消去一对括号}
9	# *	3 5	#	operate('3',' * ','5')
10	#	15	#	RETURN(GETTOP (OPND))

3.3 栈与递归的实现

栈还有一个重要应用是在程序设计语言中实现递归。一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称做递归函数。

递归是程序设计中一个强有力的工具。其一，有很多数学函数是递归定义的，如大家熟悉的阶乘函数

$$\mathrm{Fact}(n)=\begin{cases}1 & \text{若 } n=0\\ n\cdot\mathrm{Fact}(n-1) & \text{若 } n>0\end{cases} \tag{3-1}$$

2 阶 Fibonacci 数列

$$\mathrm{Fib}(n)=\begin{cases}0 & \text{若 } n=0\\ 1 & \text{若 } n=1\\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2) & \text{其他情形}\end{cases} \tag{3-2}$$

和 Ackerman 函数

www.TopSage.com

大家网 TopSage.com

$$\mathrm{Ack}(m,n)=\begin{cases}n+1 & \text{若 } m=0\\ \mathrm{Ack}(m-1,1) & \text{若 } n=0\\ \mathrm{Ack}(m-1,\mathrm{Ack}(m,n-1)) & \text{其他情形}\end{cases} \tag{3-3}$$

等；其二，有的数据结构，如二叉树、广义表等，由于结构本身固有的递归特性，则它们的操作可递归地描述；其三，还有一类问题，虽然问题本身没有明显的递归结构，但用递归求解比迭代求解更简单，如八皇后问题、Hanoi 塔问题等。

例 3-2 （n 阶 Hanoi 塔问题）假设有 3 个分别命名为 X、Y 和 Z 的塔座，在塔座 X 上插有 n 个直径大小各不相同、依小到大编号为 1,2,…，n 的圆盘（如图 3.5 所示）。现要求将 X 轴上的 n 个圆盘移至塔座 Z 上并仍按同样顺序叠排，圆盘移动时必须遵循下列规则：

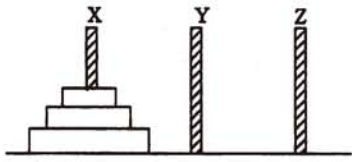


图 3.5　3 阶 Hanoi 塔问题的初始状态

（1）每次只能移动一个圆盘；

（2）圆盘可以插在 X、Y 和 Z 中的任一塔座上；

（3）任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

如何实现移动圆盘的操作呢？当 $n=1$ 时，问题比较简单，只要将编号为 1 的圆盘从塔座 X 直接移至塔座 Z 上即可；当 $n>1$ 时，需利用塔座 Y 作辅助塔座，若能设法将压在编号为 n 的圆盘之上的 $n-1$ 个圆盘从塔座 X（依照上述法则）移至塔座 Y 上，则可先将编号为 n 的圆盘从塔座 X 移至塔座 Z 上，然后再将塔座 Y 上的 $n-1$ 个圆盘（依照上述法则）移至塔座 Z 上。而如何将 $n-1$ 个圆盘从一个塔座移至另一个塔座的问题是一个和原问题具有相同特征属性的问题，只是问题的规模小 1，因此可以用同样的方法求解。由此可得如算法 3.5 所示的求解 n 阶 Hanoi 塔问题的 C 函数。

```
void hanoi (int n, char x, char y, char z)
    // 将塔座 x 上按直径由小到大且自上而下编号为 1 至 n 的 n 个圆盘按规则搬到
    // 塔座 z 上，y 可用作辅助塔座。
    // 搬动操作 move (x, n, z) 可定义为(c 是初值为 0 的全局变量，对搬动计数)：
    // printf("%i. Move disk %i from  %c  to  %c\n", ++c, n, x, z);
1   {
2       if (n==1)
3         move(x, 1, z);              //将编号为 1 的圆盘从 x 移到 z
4       else {
5         hanoi(n-1, x, z, y);        //将 x 上编号为 1 至 n-1 的圆盘移到 y, z 作辅助塔
6         move(x, n, z);              //将编号为 n 的圆盘从 x 移到 z
7         hanoi(n-1, y, x, z);        //将 y 上编号为 1 至 n-1 的圆盘移到 z, x 作辅助塔
8       }
9   }
```

算法　3.5

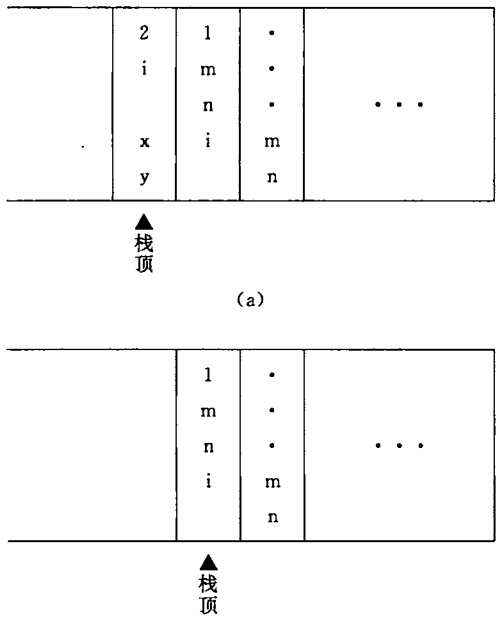
显然，这是一个递归函数，在函数的执行函数中，需多次进行自我调用。那末，这个递归函数是如何执行的？先看任意两个函数之间进行调用的情形。

与汇编程序设计中主程序和子程序之间的链接及信息交换相类似，在高级语言编制

www.TopSage.com

的程序中，调用函数和被调用函数[①]之间的链接及信息交换需通过栈来进行。

通常，当在一个函数的运行期间调用另一个函数时，在运行被调用函数之前，系统需先完成 3 件事：(1)将所有的实在参数、返回地址等信息传递给被调用函数保存；(2)为被调用函数的局部变量分配存储区；(3)将控制转移到被调函数的入口。而从被调用函数返回调用函数之前，系统也应完成 3 件工作：(1)保存被调函数的计算结果；(2)释放被调函数的数据区；(3)依照被调函数保存的返回地址将控制转移到调用函数。当有多个函数构成嵌套调用时，按照“后调用先返回”的原则，上述函数之间的信息传递和控制转移必须通过“栈”来实现，即系统将整个程序运行时所需的数据空间安排在一个栈中，每当调用一个函数时，就为它在栈顶分配一个存储区，每当从一个函数退出时，就释放它的存储区，则当前正运行的函数的数据区必在栈顶。例如，在图 3.6(c)所示主函数 main 中调用了函数 first，而在函数 first 中又调用了函数 second，则图 3.6(a)展示了当前正在执行函数 second 中某个语句时栈的状态，而图 3.6(b)展示从函数 second 退出之后正执行函数 first 中某个语句时栈的状态(图中以语句标号表示返回地址)。




```
void  first(int s,  int t);
void  second(int d);
void  main(){
    int  m, n;
    ...
    first (m, n);
    1: ...
}

int  first (int s,  int t){
    int  i;
    ...
    second (i);
    2: ...
}

int  second (int d){
    int  x, y;
    ...
}
```

(c)

图 3.6　主函数 main 执行期间运行栈的状态

一个递归函数的运行过程类似于多个函数的嵌套调用，只是调用函数和被调用函数是同一个函数，因此，和每次调用相关的一个重要的概念是递归函数运行的“层次”。假设调用该递归函数的主函数为第 0 层，则从主函数调用递归函数为进入第 1 层；从第 i 层递归调用本函数为进入“下一层”，即第 $i+1$ 层。反之，退出第 i 层递归应返回至“上一层”，

① 若在函数 A 中调用了函数 B，则称函数 A 为调用函数，称函数 B 为被调用函数。

www.TopSage.com

大家网 TopSage.com

即第 $i-1$ 层。为了保证递归函数正确执行，系统需设立一个“递归工作栈”[①]作为整个递归函数运行期间使用的数据存储区。每一层递归所需信息构成一个“工作记录”，其中包括所有的实在参数、所有的局部变量以及上一层的返回地址。每进入一层递归，就产生一个新的工作记录压入栈顶。每退出一层递归，就从栈顶弹出一个工作记录，则当前执行层的工作记录必是递归工作栈栈顶的工作记录，称这个记录为“活动记录”，并称指示活动记录的栈顶指针为“当前环境指针”。

例如，图 3.7 展示了语句

$$\text{hanoi}(3,a,b,c) \tag{3-4}$$

执行过程（从主函数进入递归函数到退出递归函数而返回至主函数）中递归工作栈状态的变化情况。由于算法 3.5 所示的递归函数中只含 4 个值参数，则每个工作记录包含 5 个数据项：返回地址和 4 个实在参数，并以递归函数中的语句行号表示返回地址，同时假设主函数的返回地址为 0。图 3.7 中▶表示栈顶指针。

实际上，在调用函数和被调用函数之间不一定传递参数的值，也可以传递参数的地址。通常，每个程序设计语言都有它自己约定的传递方法（包括被调用函数的执行结果如何返回调用函数等），读者将会在后续课程中学到其细节。

由于递归函数结构清晰，程序易读，而且它的正确性容易得到证明，因此，利用允许递归调用的语言（例如 C 语言）进行程序设计时，给用户编制程序和调试程序带来很大方便。因为对这样一类递归问题编程时，不需用户自己而由系统来管理递归工作栈。

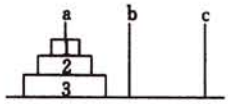
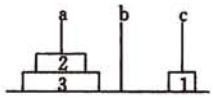
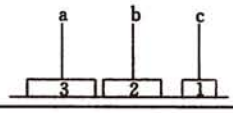
递归运行的层次	运行语句行号	递归工作栈状态（返址，n 值，x 值，y 值，z 值）	塔与圆盘的状态	说　明
1	1,2,4,5	▶ 0,3,a,b,c		由主函数进入第一层递归后，运行至语句（行）5，因递归调用而进入下一层。
2	1,2,4,5	▶ 6,2,a,c,b 0,3,a,b,c		由第一层的语句（行）5 进入第二层递归，执行至语句（行）5。
3	1,2,3,9	▶ 6,1,a,b,c 6,2,a,c,b 0,3,a,b,c		由第二层的语句（行）5 进入第三层递归，执行语句（行）3，将 1 号圆盘由 a 移至 c 后从语句（行）9 退出第三层递归，返回至第二层的语句（行）6。
2	6,7	▶ 6,2,a,c,b 0,3,a,b,c		将 2 号圆盘由 a 移至 b 后，从语句（行）7 进入下一层递归。

图 3.7　Hanoi 塔的递归函数运行示意图

① 在实际的系统中，一般都综合考虑递归调用和非递归调用统一处理，在此，我们只讨论直接递归调用的处理机制。

www.TopSage.com

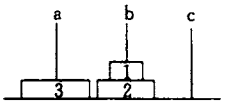
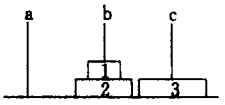
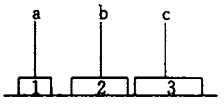
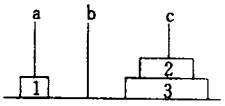
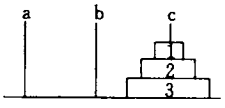
递归运行的层次	运行语句行号	递归工作栈状态（返址，n值，x值，y值，z值）	塔与圆盘的状态	说　明
3	1,2,3,9	**8,1,c,a,b** 6,2,a,c,b 0,3,a,b,c		将1号圆盘由c移至b后，从语句(行)9退出第三层，返回至第二层的语句(行)8。
2	8,9	6,2,a,c,b 0,3,a,b,c		从语句(行)9退出第二层，返回至第一层的语句(行)6。
1	6,7	0,3,a,b,c		将3号圆盘由a移至c后，从语句(行)7进入下一层递归。
2	1,2,4,5	**8,2,b,a,c** 0,3,a,b,c		从第二层的语句(行)5进入第三层递归
3	1,2,3,9	**6,1,b,c,a** 8,2,b,a,c 0,3,a,b,c		将1号圆盘由b移至a后，从语句(行)9退出第三层递归，返回至第二层语句(行)6。
2	6,7	8,2,b,a,c 0,3,a,b,c		将2号圆盘由b移至c后，从语句(行)7进入下一层递归。
3	1,2,3,9	**8,1,a,b,c** 8,2,b,a,c 0,3,a,b,c		将1号圆盘由a移至c后，从语句(行)9退出第三层，返回至第二层语句(行)8。
2	8,9	8,2,b,a,c 0,3,a,b,c		从语句(行)9退出第二层，返回至第一层语句(行)8。
1	8,9	0,3,a,b,c		从语句(行)9退出递归函数，返回至主函数
0		栈空		继续运行主函数

图　3.7(续)

3.4　队　　列

3.4.1　抽象数据类型队列的定义

和栈相反，**队列**(queue)是一种**先进先出**(first in first out，缩写为 **FIFO**)的线性表。它只允许在表的一端进行插入，而在另一端删除元素。这和我们日常生活中的排队是一

www.TopSage.com

大家网 TopSage.com

致的，最早进入队列的元素最早离开。在队列中，允许插入的一端叫做队尾(rear)，允许删除的一端则称为队头(front)。假设队列为 $q=(a_1,a_2,\cdots,a_n)$，那么，a_1 就是队头元素，a_n 则是队尾元素。队列中的元素是按照 $a_1,a_2,\cdots,a_n$ 的顺序进入的，退出队列也只能按照这个次序依次退出，也就是说，只有在 $a_1,a_2,\cdots,a_{n-1}$ 都离开队列之后，a_n 才能退出队列。图 3.8 是队列的示意图。

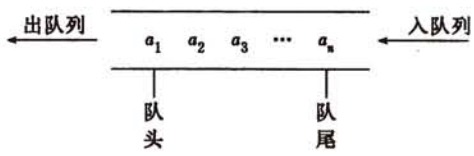


图 3.8 队列的示意图

队列在程序设计中也经常出现。一个最典型的例子就是操作系统中的作业排队。在允许多道程序运行的计算机系统中，同时有几个作业运行。如果运行的结果都需要通过通道输出，那就要按请求输出的先后次序排队。每当通道传输完毕可以接受新的输出任务时，队头的作业先从队列中退出作输出操作。凡是申请输出的作业都从队尾进入队列。

队列的操作与栈的操作类似，也有 8 个，不同的是删除是在表的头部(即队头)进行。

下面给出队列的抽象数据类型定义：

```
ADT Queue {
    数据对象:D = {a_i | a_i ∈ ElemSet, i = 1,2,⋯,n, n≥0 }
    数据关系:Rl = {<a_i-1,a_i> | a_i-1, a_i ∈ D, i = 2,⋯,n }
            约定其中 a_1 端为队列头,a_n 端为队列尾。
    基本操作:
      InitQueue(&Q)
          操作结果:构造一个空队列 Q。
      DestroyQueue(&Q)
          初始条件:队列 Q 已存在。
          操作结果:队列 Q 被销毁,不再存在。
      ClearQueue(&Q)
          初始条件:队列 Q 已存在。
          操作结果:将 Q 清为空队列。
      QueueEmpty(Q)
          初始条件:队列 Q 已存在。
          操作结果:若 Q 为空队列,则返回 TRUE,否则 FALSE。
      QueueLength(Q)
          初始条件:队列 Q 已存在。
          操作结果:返回 Q 的元素个数,即队列的长度。
      GetHead(Q, &e)
          初始条件:Q 为非空队列。
          操作结果:用 e 返回 Q 的队头元素。
      EnQueue(&Q, e)
          初始条件:队列 Q 已存在。
          操作结果:插入元素 e 为 Q 的新的队尾元素。
      DeQueue(&Q, &e)
          初始条件:Q 为非空队列。
          操作结果:删除 Q 的队头元素,并用 e 返回其值。
```


www.TopSage.com

QueueTraverse(Q,visit())

初始条件：Q已存在且非空。

操作结果：从队头到队尾，依次对Q的每个数据元素调用函数visit()。一旦visit()失败，则操作失败。

}ADT Queue

和栈类似，在本书以后各章中引用的队列都应是如上定义的队列类型。队列的数据元素类型在应用程序内定义。

除了栈和队列之外，还有一种限定性数据结构是**双端队列**(deque)。

双端队列是限定插入和删除操作在表的两端进行的线性表。这两端分别称做端点1和端点2(如图3.9(a)所示)。也可像栈一样，可以用一个铁道转轨网络来比喻双端队列，如图3.9(b)所示。在实际使用中，还可以有输出受限的双端队列(即一个端点允许插入和删除，另一个端点只允许插入的双端队列)和输入受限的双端队列(即一个端点允许插入和删除，另一个端点只允许删除的双端队列)。而如果限定双端队列从某个端点插入的元素只能从该端点删除，则该双端队列就蜕变为两个栈底相邻接的栈了。

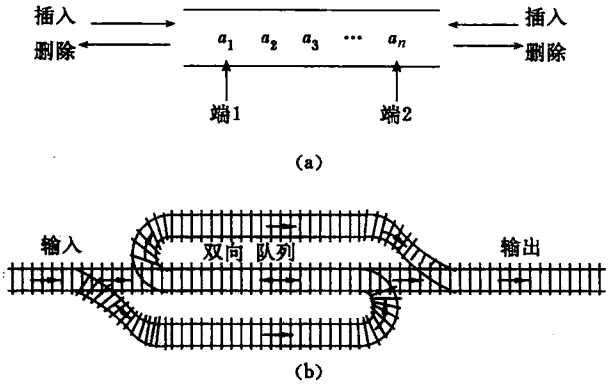


图3.9 双端队列示意图

(a) 双端队列； (b) 铁道转轨网

尽管双端队列看起来似乎比栈和队列更灵活，但实际上在应用程序中远不及栈和队列有用，故在此不作详细讨论。

3.4.2 链队列——队列的链式表示和实现

和线性表类似，队列也可以有两种存储表示。

用链表表示的队列简称为**链队列**，如图3.10所示。一个链队列显然需要两个分别指示队头和队尾的指针(分别称为头指针和尾指针)才能惟一确定。这里，和线性表的单链表一样，为了操作方便起见，我们也给链队列添加一个头结点，并令头指针指向头结点。由此，空的链队列的判决条件为头指针和尾指针均指向头结点，如图3.11(a)所示。

链队列的操作即为单链表的插入和删除操作的特殊情况，只是尚需修改尾指针或头指针，图3.11(b)～(d)展示了这两种操作进行时指针变化的情况。下面给出链队列类型的模块说明。

www.TopSage.com

大家网 TopSage.com

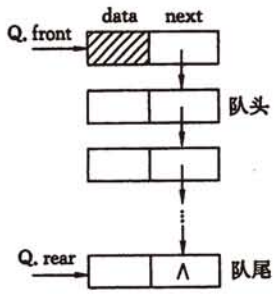


图 3.10 链队列示意图

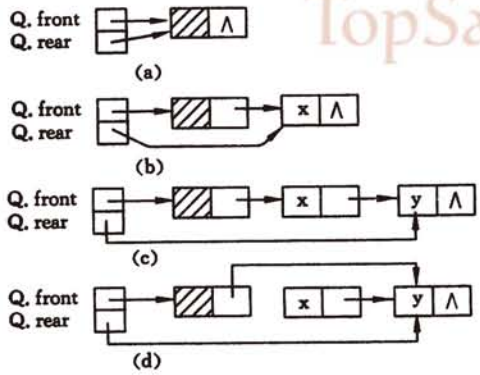


图 3.11 队列运算指针变化状况

(a) 空队列； (b) 元素 x 入队列；

(c) 元素 y 入队列； (d) 元素 x 出队列

```
// ===== ADT Queue 的表示与实现 =====

// ----- 单链队列——队列的链式存储结构 -----
typedef struct QNode {
    QElemType       data;
    struct QNode    * next;
}QNode, * QueuePtr;
typedef struct {
    QueuePtr  front;   // 队头指针
    QueuePtr  rear;    // 队尾指针
}LinkQueue;

// ----- 基本操作的函数原型说明 -----
Status InitQueue (LinkQueue &Q)
    // 构造一个空队列 Q
Status DestroyQueue (LinkQueue &Q)
    // 销毁队列 Q,Q 不再存在
Status ClearQueue (LinkQueue &Q)
    // 将 Q 清为空队列
Status QueueEmpty (LinkQueue Q)
    // 若队列 Q 为空队列,则返回 TRUE,否则返回 FALSE
int QueueLength (LinkQueue Q)
    // 返回 Q 的元素个数,即为队列的长度
Status GetHead (LinkQueue Q, QElemType &e)
    // 若队列不空,则用 e 返回 Q 的队头元素,并返回 OK;否则返回 ERROR
Status EnQueue (LinkQueue &Q, QElemType e)
    // 插入元素 e 为 Q 的新的队尾元素
Status DeQueue (LinkQueue &Q, QElemType &e)
    // 若队列不空,则删除 Q 的队头元素,用 e 返回其值,并返回 OK;
    // 否则返回 ERROR
```


www.TopSage.com

```
Status QueueTraverse(LinkQueue Q, visit())
    // 从队头到队尾依次对队列 Q 中每个元素调用函数 visit()。一旦 visit 失败,则操作失败。

// ------ 基本操作的算法描述(部分) ------
Status InitQueue (LinkQueue &Q) {
    // 构造一个空队列 Q
    Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode));
    if (!Q.front) exit (OVERFLOW);                    //存储分配失败
    Q.front->next = NULL;
    return OK;
}

Status DestroyQueue (LinkQueue &Q) {
    // 销毁队列 Q
    while (Q.front) {
      Q.rear = Q.front->next;
      free (Q.front);
      Q.front = Q.rear;
    }
    return OK;
}

Status EnQueue (LinkQueue &Q, QElemType e) {
    // 插入元素 e 为 Q 的新的队尾元素
    p = (QueuePtr) malloc (sizeof (QNode));
    if (!p)  exit (OVERFLOW);                    //存储分配失败
    p->data = e;     p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return OK;
}

Status DeQueue (LinkQueue &Q, QElemType &e) {
    // 若队列不空,则删除 Q 的队头元素,用 e 返回其值,并返回 OK;
    // 否则返回 ERROR
    if (Q.front == Q.rear) return ERROR;
    p = Q.front->next;
    e = p->data;
    Q.front->next = p->next;
    if (Q.rear == p) Q.rear = Q.front;
    free (p);
    return OK;
}
```

在上述模块的算法描述中,请读者注意删除队列头元素算法中的特殊情况。一般情况下,删除队列头元素时仅需修改头结点中的指针,但当队列中最后一个元素被删后,队

www.TopSage.com

大家网
TopSage.com

列尾指针也丢失了，因此需对队尾指针重新赋值(指向头结点)。

3.4.3 循环队列——队列的顺序表示和实现

和顺序栈相类似，在队列的顺序存储结构中，除了用一组地址连续的存储单元依次存放从队列头到队列尾的元素之外，尚需附设两个指针 front 和 rear 分别指示队列头元素及队列尾元素的位置。为了在 C 语言中描述方便起见，在此我们约定：初始化建空队列时，令 front＝rear＝0，每当插入新的队列尾元素时，“尾指针增 1”；每当删除队列头元素时，“头指针增 1”。因此，在非空队列中，头指针始终指向队列头元素，而尾指针始终指向队列尾元素的下一个位置，如图 3.12 所示。

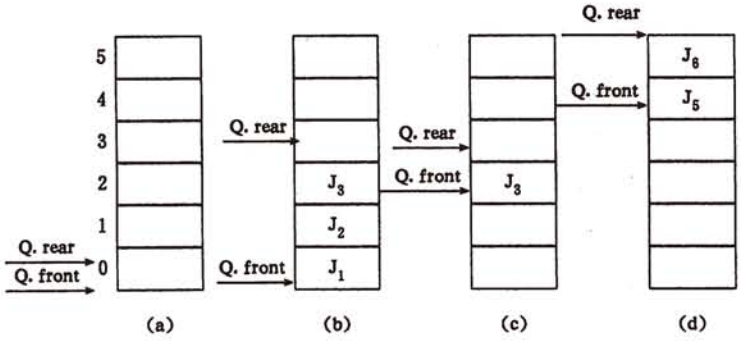


图 3.12 头、尾指针和队列中元素之间的关系

(a) 空队列；(b) J_1、J_2 和 J_3 相继入队列；(c) J_1 和 J_2 相继被删除；

(d) J_4、J_5 和 J_6 相继插入队列之后 J_3 及 J_4 被删除

假设当前为队列分配的最大空间为 6，则当队列处于图 3.12(d)的状态时不可再继续插入新的队尾元素，否则会因数组越界而遭致程序代码被破坏。然而此时又不宜如顺序栈那样，进行存储再分配扩大数组空间，因为队列的实际可用空间并未占满。一个较巧妙的办法是将顺序队列臆造为一个环状的空间，如图 3.13 所示，称之为循环队列。指针和队列元素之间关系不变，如图 3.14(a)所示循环队列中，队列头元素是 J_3，队列尾元素是 J_5，之后 J_6、J_7 和 J_8 相继插入，则队列空间均被占满，如图 3.14(b)所示，此时 Q. front＝Q. rear；反之，若 J_3、J_4 和 J_5 相继从图 3.14(a)的队列中删除，使队列呈“空”的状态，如图 3.14(c)所示。此时也存在关系式 Q. front＝Q. rear，由此可见，只凭等式 Q. front＝Q. rear 无法判别队列空间是“空”还是“满”。可有两种处理方法：其一是另设一个标志位以区别队列是“空”还是“满”；其二是少用一个元素空间，约定以“队列头指针在队列尾指针的下一位置(指环状的下一位置)上”作为队列呈“满”状态的标志。

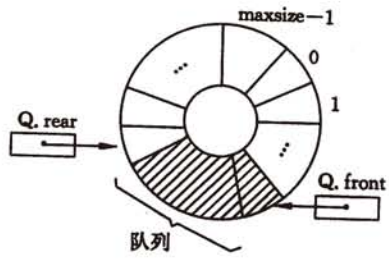


图 3.13 循环队列示意图

从上述分析可见，在 C 语言中不能用动态分配的一维数组来实现循环队列。如果用户的应用程序中设有循环队列，则必须为它设定一个最大队列长度；若用户无法预估所用

www.TopSage.com

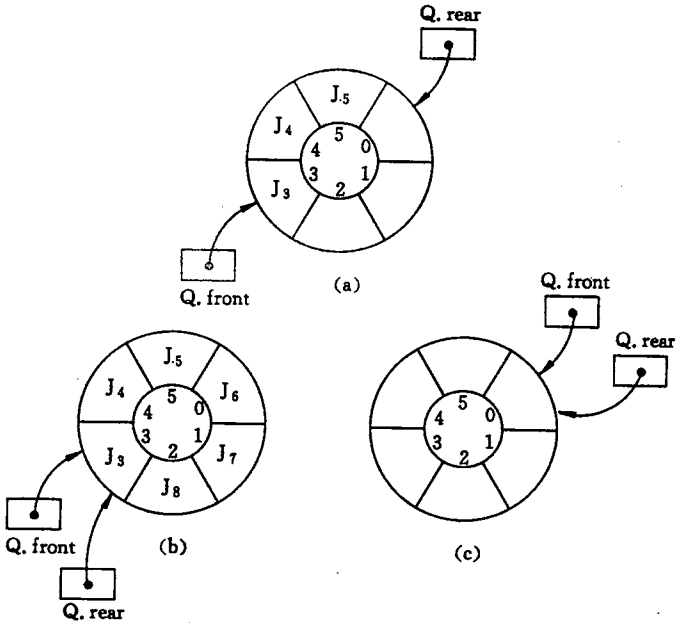


图 3.14　循环队列的头尾指针

(a) 一般情况；(b) 队列满时；(c) 空队列

队列的最大长度，则宜采用链队列。

循环队列类型的模块说明如下：

```
// ----- 循环队列——队列的顺序存储结构 -----
#define MAXQSIZE  100     //最大队列长度
typedef struct {
    QElemType  * base;     //初始化的动态分配存储空间
    int  front;            //头指针，若队列不空，指向队列头元素
    int  rear;             //尾指针，若队列不空，指向队列尾元素的下一个位置
}SqQueue;

// ----- 循环队列的基本操作的算法描述 -----
Status InitQueue (SqQueue &Q) {
    // 构造一个空队列 Q
    Q.base = (QElemType * ) malloc (MAXQSIZE * sizeof (QElemType));
    if (!Q.base) exit (OVERFLOW);  //存储分配失败
    Q.front = Q.rear = 0;
    return OK;
}

int QueueLength (SqQueue Q) {
    // 返回 Q 的元素个数，即队列的长度
    return (Q.rear - Q.front + MAXQSIZE) % MAXQSIZE;
}
```


www.TopSage.com

大家网
TopSage.com

```
Status EnQueue (SqQueue &Q, QElemType e) {
    // 插入元素 e 为 Q 的新的队尾元素
    if ((Q.rear + 1) % MAXQSIZE == Q.front) return ERROR; //队列满
    Q.base[Q.rear] = e;
    Q.rear = (Q.rear + 1) % MAXQSIZE;
    return OK;
}

Status DeQueue (SqQueue &Q, QElemType &e) {
    // 若队列不空,则删除 Q 的队头元素,用 e 返回其值,并返回 OK;
    // 否则返回 ERROR
    if (Q.front == Q.rear)  return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front + 1) % MAXQSIZE;
    return OK;
}
```

3.5 离散事件模拟

在日常生活中,我们经常会遇到许多为了维护社会正常秩序而需要排队的情景。这样一类活动的模拟程序通常需要用到队列和线性表之类的数据结构,因此是队列的典型应用例子之一。这里将向读者介绍一个银行业务的模拟程序。

假设某银行有 4 个窗口对外接待客户,从早晨银行开门起不断有客户进入银行。由于每个窗口在某个时刻只能接待一个客户,因此在客户人数众多时需在每个窗口前顺次排队,对于刚进入银行的客户,如果某个窗口的业务员正空闲,则可上前办理业务;反之,若 4 个窗口均有客户所占,他便会排在人数最少的队伍后面。现在需要编制一个程序以模拟银行的这种业务活动并计算一天中客户在银行逗留的平均时间。

为了计算这个平均时间,我们自然需要掌握每个客户到达银行和离开银行这两个时刻,后者减去前者即为每个客户在银行的逗留时间。所有客户逗留时间的总和被一天内进入银行的客户数除便是所求的平均时间。称客户到达银行和离开银行这两个时刻发生的事情为"事件",则整个模拟程序将按事件发生的先后顺序进行处理,这样一种模拟程序称做事件驱动模拟。算法 3.6 描述的正是上述银行客户的离散事件驱动模拟程序。

```
void Bank_Simulation(int CloseTime) {
    // 银行业务模拟,统计一天内客户在银行逗留的平均时间。

    OpenForDay ( );                                    // 初始化
    while (MoreEvent) {
      EventDrived(OccurTime, EventType);               // 事件驱动
      switch (EventType) {
        case 'A' : CustomerArrived( );    break;       // 处理客户到达事件
        case 'D' : CustomerDeparture( );  break;       // 处理客户离开事件
        default : Invalid( );
      }// switch
```


www.TopSage.com

```
    }// while
    CloseForDay;                          // 计算平均逗留时间
}// Bank_Simulation
```

算法 3.6

下面讨论模拟程序的实现，首先要讨论模拟程序中需要的数据结构及其操作。

算法 3.6 处理的主要对象是“事件”，事件的主要信息是事件类型和事件发生的时刻。算法中处理的事件有两类：一类是客户到达事件；另一类是客户离开事件。前一类事件发生的时刻随客户到来自然形成；后一类事件发生时刻则由客户事务所需时间和等待所耗时间而定。由于程序驱动是按事件发生时刻的先后顺序进行，则事件表应是有序表，其主要操作是插入和删除事件。

模拟程序中需要的另一种数据结构是表示客户排队的队列，由于前面假设银行有 4 个窗口，因此程序中需要 4 个队列，队列中有关客户的主要信息是客户到达的时刻和客户办理事务所需的时间。每个队列中的队头客户即为正在窗口办理事务的客户，他办完事务离开队列的时刻就是即将发生的客户离开事件的时刻，这就是说，对每个队头客户都存在一个将要驱动的客户离开事件。因此，在任何时刻即将发生的事件只有下列 5 种可能：(1)新的客户到达；(2)1 号窗口客户离开；(3)2 号窗口客户离开；(4)3 号窗口客户离开；(5)4 号窗口客户离开。

从以上分析可见，在这个模拟程序中只需要两种数据类型：有序链表和队列。它们的数据元素类型分别定义如下：

```
typedef  struct {
    int OccurTime;      // 事件发生时刻
    int NType;          // 事件类型,0 表示到达事件,1 至 4 表示四个窗口的离开事件
}Event, ElemType;       // 事件类型,有序链表 LinkList 的数据元素类型

typedef  LinkList  EventList      // 事件链表类型,定义为有序链表

typedef  struct {
    int  ArrivalTime;       // 到达时刻
    int  Duration;          // 办理事务所需时间
}QElemType;                 // 队列的数据元素类型
```

现在我们详细分析算法 3.6 中的两个主要操作步骤是如何实现的。

先看对新客户到达事件的处理。

由于在实际的银行中，客户到达的时刻及其办理事务所需时间都是随机的，在模拟程序中可用随机数来代替。不失一般性，假设第一个顾客进门的时刻为 0，即是模拟程序处理的第一个事件，之后每个客户到达的时刻在前一个客户到达时设定。因此在客户到达事件发生时需先产生两个随机数：其一为此时刻到达的客户办理事务所需时间 durtime；其二为下一客户将到达的时间间隔 intertime，假设当前事件发生的时刻为 occurtime，则下一个客户到达事件发生的时刻为 occurtime＋intertime。由此应产生一个新的客户到达事件插入事件表；刚到达的客户则应插入到当前所含元素最少的队列中；若该队列在插

www.TopSage.com

大家网 TopSage.com

入前为空，则还应产生一个客户离开事件插入事件表。

客户离开事件的处理比较简单。首先计算该客户在银行逗留的时间，然后从队列中删除该客户后查看队列是否空，若不空则设定一个新的队头客户离开事件。

最后我们给出在上述数据结构下实现的银行事件驱动模拟程序，如算法 3.7 所示。

```
    // 程序中用到的主要变量
EventList    ev;                // 事件表
Event        en;                // 事件
LinkQueue    q[5];              // 4 个客户队列
QElemType    customer;          // 客户记录
int  TotalTime, CustomerNum;    // 累计客户逗留时间，客户数

int cmp (Event a,  Event b);
  // 依事件 a 的发生时刻< 或 = 或> 事件 b 的发生时刻分别返回 -1 或 0 或 1

void OpenForDay() {
    // 初始化操作
    TotalTime = 0;  CustomerNum = 0;          // 初始化累计时间和客户数为 0
    InitList (ev);                            // 初始化事件链表为空表
    en.OccurTime = 0;  en.NType = 0;          // 设定第一个客户到达事件
    OrderInsert (ev, en, cmp);                // 插入事件表
    for (i = 1; i<= 4; ++ i)  InitQueue (q[i]); // 置空队列
}// OpenForDay

void CustomerArrived() {
    // 处理客户到达事件，en.NType = 0。
    ++ CustomerNum;
    Random(durtime, intertime);               // 生成随机数
    t = en.OccurTime + intertime;             // 下一客户到达时刻
    if (t<CloseTime)                          // 银行尚未关门，插入事件表
      OrderInsert (ev, (t, 0), cmp);
    i = Minimum(q);                           // 求长度最短队列
    EnQueue (q[i], (en.OccurTime, durtime));
    if (QueueLength (q[i]) == 1)
      OrderInsert (ev, (en.OccurTime + durtime, i), cmp);
                                              // 设定第 i 队列的一个离开事件并插入事件表
}// CustomerArrived

void CustomerDeparture() {
    // 处理客户离开事件，en.NType>0。
    i = en.NType;  DelQueue (q[i], customer);  // 删除第 i 队列的排头客户
    TotalTime += en.OccurTime - customer.ArrivalTime;
                                              // 累计客户逗留时间
    if (!QueueEmpty(q[i])) {          // 设定第 i 队列的一个离开事件并插入事件表
       GetHead (q[i], customer);
       OrderInsert (ev, (en.OccurTime + curtomer.Duration, i), (*cmp)());
    }
```


www.TopSage.com

```
}// CustomerDeparture

void Bank_Simulation(int CloseTime) {
    OpenForDay( );                                    // 初始化
    while (!ListEmpty(ev)) {
      DelFirst (GetHead (ev), p);  en = GetCurElem(p);
      if (en.NType == 0)
          CustomerArrived( );                         // 处理客户到达事件
      else CustomerDeparture( );                      // 处理客户离开事件
    }
    // 计算并输出平均逗留时间
    printf("The Average Time is %f\n", (float)TotalTime / CustomerNum);
}// Bank_Simulation
```

算法 3.7

例 3-3 假设每个客户办理业务的时间不超过 30 分钟；两个相邻到达银行的客户的时间间隔不超过 5 分钟。模拟程序从第一个客户到达时间为“0”开始起运行。

删除事件表上第一个结点，得到 en. OccurTime＝0，因为 en. NType＝0，则随即得到两个随机数(23,4)，生成一个下一客户到达银行的事件(OccurTime＝4，NType＝0)插入事件表；刚到的第一位客户排在第一个窗口的队列中(ArrivalTime＝0，Duration＝23)，由于他是排头，故生成一个客户将离开的事件(OccurTime＝23，NType＝1)插入事件表。

删除事件表上第一个结点，仍是新客户到达事件(因为 en. NType＝0)，en. OccurTime＝4，得到随机数为(3,1)，则下一客户到达银行的时间为 OccurTime＝4＋1＝5，由于此时第二个窗口是空的，则刚到的第二位客户为第二个队列的队头(ArrivalTime＝4，Duration＝3)，因而生成一个客户将离开的事件(OccurTime＝7，NType＝2)插入事件表。

删除事件表上第一个结点，仍是新客户到达事件，en. OccurTime＝5，得到随机数(11,3)，则插入事件表的新事件为(OccurTime＝8，NType＝0)，同时，刚到的第三位客户成为第三个队列的队头(ArrivalTime＝5，Duration＝11)，因而插入事件表的新事件为(OccurTime＝16，NType＝3)。

删除事件表的第一个结点，因为 NType＝2，说明是第二个窗口的客户离开银行 en. OccurTime＝7，删去第二个队列的队头，curstomer. ArrivalTime＝4，则他在银行的逗留时间为 3 分钟。

依次类推，在模拟开始后的一段时间内，事件表和队列的状态如图 3. 15 所示，ev. first 为链表头指针。

www.TopSage.com

大家网
TopSage.com

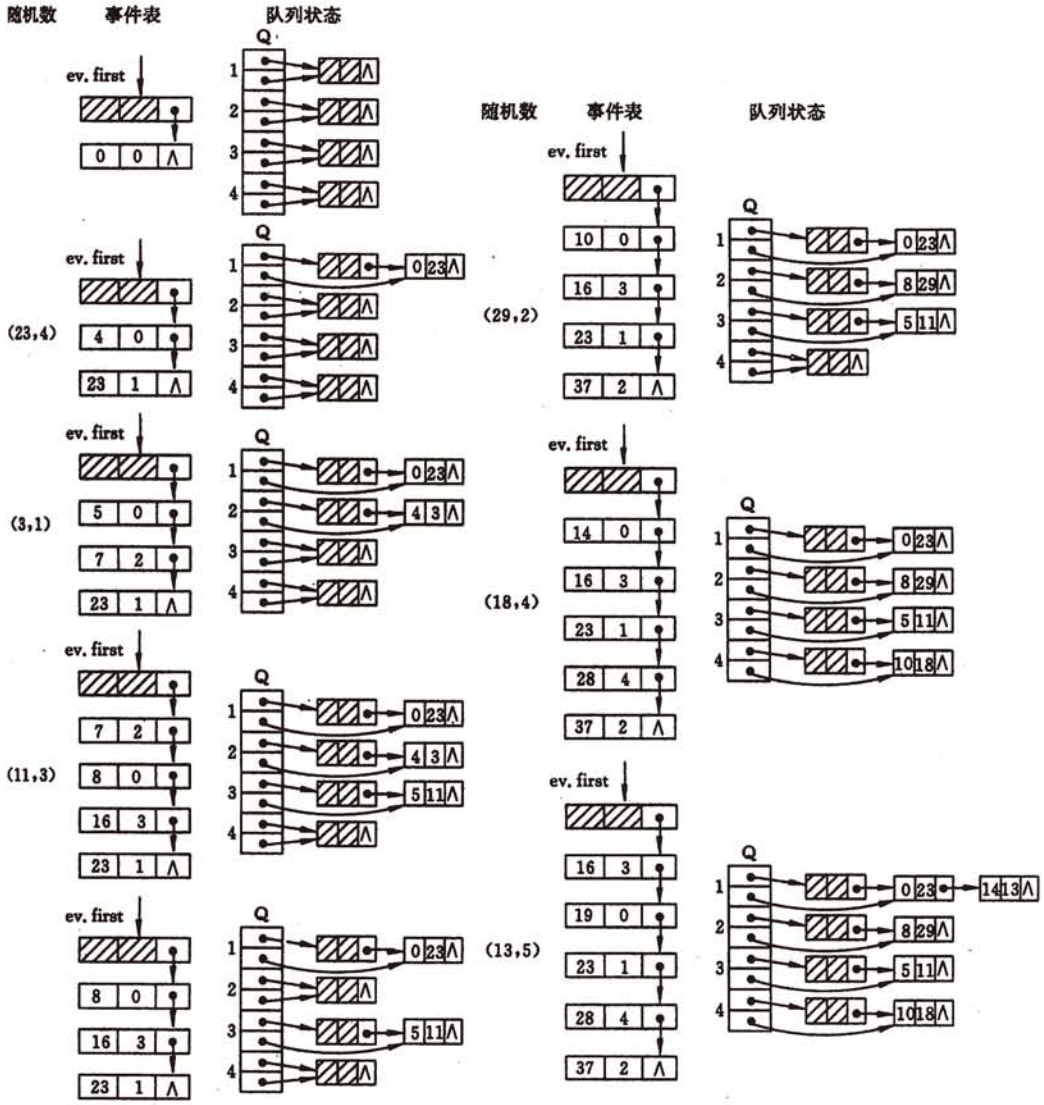


图 3.15　事件驱动模拟(算法 3.7)过程中事件表和队列状态变化状况

www.TopSage.com

第4章 串

计算机上的非数值处理的对象基本上是字符串数据。在较早的程序设计语言中，字符串是作为输入和输出的常量出现的。随着语言加工程序的发展，产生了字符串处理。这样，字符串也就作为一种变量类型出现在越来越多的程序设计语言中，同时也产生了一系列字符串的操作。字符串一般简称为串。在汇编和语言的编译程序中，源程序和目标程序都是字符串数据。在事务处理程序中，顾客的姓名和地址以及货物的名称、产地和规格等一般也是作为字符串处理的。又如信息检索系统、文字编辑程序、问答系统、自然语言翻译系统以及音乐分析程序等，都是以字符串数据作为处理对象的。

然而，现今我们使用的计算机的硬件结构主要是反映数值计算的需要的，因此，在处理字符串数据时比处理整数和浮点数要复杂得多。而且，在不同类型的应用中，所处理的字符串具有不同的特点，要有效地实现字符串的处理，就必须根据具体情况使用合适的存储结构。这一章，我们将讨论一些基本的串处理操作和几种不同的存储结构。

4.1 串类型的定义

串(string)(或字符串)是由零个或多个字符组成的有限序列，一般记为

$$s = 'a_1 a_2 \cdots a_n' \qquad (n \geqslant 0) \tag{4-1}$$

其中，s 是串的名，用单引号括起来的字符序列是串的值；$a_i(1 \leqslant i \leqslant n)$ 可以是字母、数字或其他字符；串中字符的数目 n 称为串的**长度**。零个字符的串称为**空串**(null string)，它的长度为零。

串中任意个连续的字符组成的子序列称为该串的**子串**。包含子串的串相应地称为**主串**。通常称字符在序列中的序号为该字符在串中的**位置**。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

例如，假设 a、b、c、d 为如下的 4 个串：

$$a='\text{BEI}' \quad , \quad b='\text{JING}'$$
$$c='\text{BEIJING}' \quad , \quad d='\text{BEI JING}'$$

则它们的长度分别为 3、4、7 和 8；并且 a 和 b 都是 c 和 d 的子串，a 在 c 和 d 中的位置都是 1，而 b 在 c 中的位置是 4，在 d 中的位置则是 5。

称两个串是**相等**的，当且仅当这两个串的值相等。也就是说，只有当两个串的长度相等，并且各个对应位置的字符都相等时才相等。例如上例中的串 a、b、c 和 d 彼此都不相等。

值得一提的是，串值必须用一对单引号括起来，但单引号本身不属于串，它的作用只是为了避免与变量名或数的常量混淆而已。

例如在程序设计语言中

www.TopSage.com

大家网
TopSage.com

x='123';

则表明 x 是一个串变量名,赋给它的值是字符序列 123。又如

tsing='TSING'

中,tsing 是一个串变量名,而字符序列 TSING 是其值。

在各种应用中,空格常常是串的字符集合中的一个元素,因而可以出现在其他字符中间。由一个或多个空格组成的串' '称为**空格串**(blank string,请注意:此处不是空串)。它的长度为串中空格字符的个数。为了清楚起见,以后我们用符号"∅"来表示"空串"。

串的逻辑结构和线性表极为相似,区别仅在于串的数据对象约束为字符集。然而,串的基本操作和线性表有很大差别。在线性表的基本操作中,大多以"单个元素"作为操作对象,例如在线性表中查找某个元素、求取某个元素、在某个位置上插入一个元素和删除一个元素等;而在串的基本操作中,通常以"串的整体"作为操作对象,例如在串中查找某个子串、求取一个子串、在串的某个位置上插入一个子串以及删除一个子串等。

串的抽象数据类型的定义如下:

ADT String {

数据对象: $D=\{ a_i \mid a_i \in CharacterSet, i=1,2,\cdots,n, n\geqslant 0 \}$

数据关系: $R_1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,\cdots,n \}$

基本操作:

StrAssign (&T, chars)

初始条件:chars 是字符串常量。

操作结果:生成一个其值等于 chars 的串 T。

StrCopy (&T, S)

初始条件:串 S 存在。

操作结果:由串 S 复制得串 T。

StrEmpty (S)

初始条件:串 S 存在。

操作结果:若 S 为空串,则返回 TRUE,否则返回 FALSE。

StrCompare (S, T)

初始条件:串 S 和 T 存在。

操作结果:若 S > T,则返回值 > 0;若 S = T,则返回值 = 0;若 S < T,则返回值< 0。

StrLength (S)

初始条件:串 S 存在。

操作结果:返回 S 的元素个数,称为串的长度。

ClearString (&S)

初始条件:串 S 存在。

操作结果:将 S 清为空串。

Concat (&T, S1, S2)

初始条件:串 S1 和 S2 存在。

操作结果:用 T 返回由 S1 和 S2 联接而成的新串。

SubString (&Sub, S, pos, len)

初始条件:串 S 存在,$1\leqslant pos\leqslant StrLength(S)$ 且 $0\leqslant len\leqslant StrLength(S)-pos+1$。

操作结果:用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。

Index (S, T, pos)

初始条件:串 S 和 T 存在,T 是非空串,$1\leqslant pos\leqslant StrLength(S)$。

操作结果:若主串 S 中存在和串 T 值相同的子串,则返回它在主串 S 中第 pos 个字符之后第一次出现的位置;否则函数值为 0。

www.TopSage.com

Replace (&S, T, V)

初始条件：串 S,T 和 V 存在，T 是非空串。

操作结果：用 V 替换主串 S 中出现的所有与 T 相等的不重叠的子串。

StrInsert (&S, pos, T)

初始条件：串 S 和 T 存在，1≤pos≤StrLength(S) + 1。

操作结果：在串 S 的第 pos 个字符之前插入串 T。

StrDelete (&S, pos, len)

初始条件：串 S 存在，1≤pos≤StrLength(S) - len + 1。

操作结果：从串 S 中删除第 pos 个字符起长度为 len 的子串。

DestroyString (&S)

初始条件：串 S 存在。

操作结果：串 S 被销毁。

}ADT String

对于串的基本操作集可以有不同的定义方法，读者在使用高级程序设计语言中的串类型时，应以该语言的参考手册为准。在上述抽象数据类型定义的 13 种操作中，串赋值 StrAssign、串比较 StrCompare、求串长 StrLength、串联接 Concat 以及求子串 SubString5 种操作构成串类型的最小操作子集。即这些操作不可能利用其他串操作来实现，反之，其他串操作(除串清除 ClearString 和串销毁 DestroyString 外)均可在这个最小操作子集上实现。

例如，可利用判等、求串长和求子串等操作实现定位函数 Index(S,T,pos)。算法的基本思想为：在主串 S 中取从第 i(i 的初值为 pos)个字符起、长度和串 T 相等的子串和串 T 比较，若相等，则求得函数值为 i，否则 i 值增 1 直至串 S 中不存在和串 T 相等的子串为止，如算法 4.1 所示。

```
int Index (String S, String T, int pos) {
  // T为非空串。若主串 S 中第 pos 个字符之后存在与 T 相等的子串，
  // 则返回第一个这样的子串在 S 中的位置，否则返回 0
  if (pos > 0) {
    n = StrLength(S);  m = StrLength(T);  i = pos;
    while ( i <= n-m+1) {
        SubString (sub, S, i, m);
        if (StrCompare(sub,T) != 0)      ++i ;
        else return i ;          // 返回子串在主串中的位置
    }// while
  }// if
  return 0;                      // S 中不存在与 T 相等的子串
}// Index
```

算法 4.1

4.2 串的表示和实现

如果在程序设计语言中，串只是作为输入或输出的常量出现，则只需存储此串的串值，即字符序列即可。但在多数非数值处理的程序中，串也以变量的形式出现。

www.TopSage.com

大家网
TopSage.com

串有 3 种机内表示方法，分别介绍如下。

4.2.1 定长顺序存储表示

类似于线性表的顺序存储结构，用一组地址连续的存储单元存储串值的字符序列。在串的定长顺序存储结构中，按照预定义的大小，为每个定义的串变量分配一个固定长度的存储区，则可用定长数组如下描述之。

```
// - - - - - 串的定长顺序存储表示 - - - - -
#define  MAXSTRLEN 255     // 用户可在 255 以内定义最大串长
typedef  unsigned char  SString[MAXSTRLEN + 1];     // 0 号单元存放串的长度
```

串的实际长度可在这预定义长度的范围内随意，超过预定义长度的串值则被舍去，称之为“截断”。对串长有两种表示方法：一是如上述定义描述的那样，以下标为 0 的数组分量存放串的实际长度，如 PASCAL 语言中的串类型采用这种表示方法；二是在串值后面加一个不计入串长的结束标记字符，如在有的 C 语言中以“\0”表示串值的终结。此时的串长为隐含值，显然不便于进行某些串操作。

在这种存储结构表示时如何实现串的操作，下面以串联接和求子串为例讨论之。

1. 串联接 Concat(&T,S1,S2)

假设 S1、S2 和 T 都是 SString 型的串变量，且串 T 是由串 S1 联结串 S2 得到的，即串 T 的值的前一段和串 S1 的值相等，串 T 的值的后一段和串 S2 的值相等，则只要进行相应的“串值复制”操作即可，只是需按前述约定，对超长部分实施“截断”操作。基于串 S1 和 S2 长度的不同情况，串 T 值的产生可能有如下 3 种情况：(1) S1[0]+S2[0]≤MAXSTRLEN，如图 4.10(a)所示，得到的串 T 是正确的结果；(2) S1[0]<MAXSTRLEN 而 S1[0]+S2[0]>MAXSTRLEN，则将串 S2 的一部分截断，得到的串 T 只包含串 S2 的一个子串，如图 4.1(b)所示；(3) S1[0]=MAXSTRLEN，则得到的串 T 并非联接结果，而和串 S1 相等。上述算法描述如算法 4.2 所示。

```
Status Concat(SString &T,SString S1, SString S2) {
    // 用 T 返回由 S1 和 S2 联接而成的新串。若未截断，则返回 TRUE,否则 FALSE。
    if (S1[0] + S2[0] <= MAXSTRLEN) {        // 未截断
        T[1..S1[0]] = S1[1..S1[0]];
        T[S1[0] + 1..S1[0] + S2[0]] = S2[1..S2[0]];
        T[0] = S1[0] + S2[0];   uncut = TRUE;
    }
    else if (S1[0] < MAXSTRLEN) {            // 截断
        T[1..S1[0]] = S1[1..S1[0]];
        T[S1[0] + 1..MAXSTRLEN] = S2[1..MAXSTRLEN - S1[0]];
        T[0] = MAXSTRLEN;  uncut = FALSE;
    }
    else {                                   // 截断(仅取 S1)
        T[0..MAXSTRLEN] = S1[0..MAXSTRLEN];
                                 // T[0] == S1[0] == MAXSTRLEN
```


www.TopSage.com

```
        uncut = FALSE;
    }
    return uncut;
}// Concat
```

算法 4.2

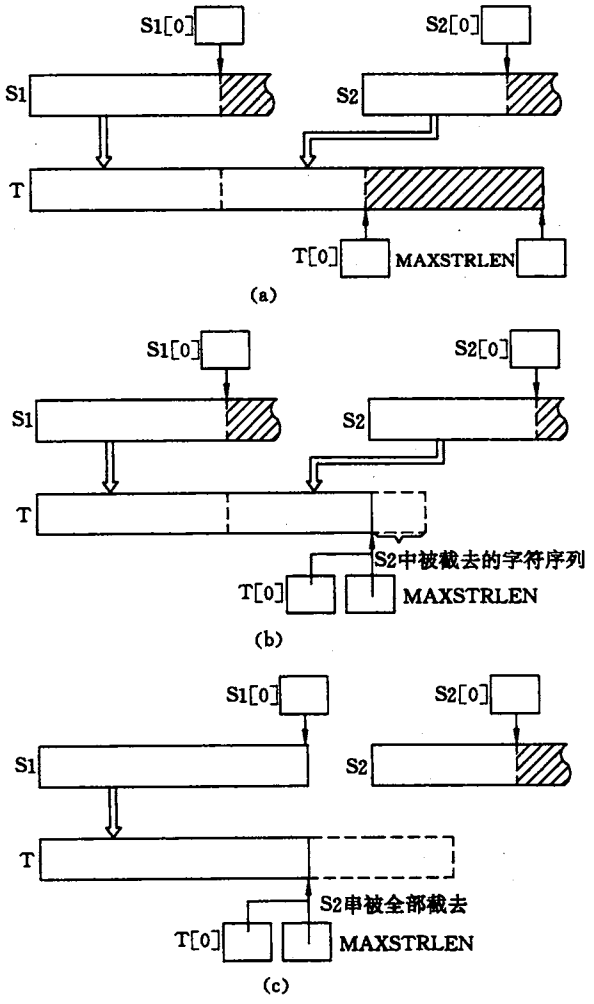


图 4.1 串的联结操作 Concat(T,S1,S2)示意图

(a) S1[0]+S2[0]≤MAXSTRLEN;

(b) S1[0]<MAXSTRLEN 而 S1[0]+S2[0]>MAXSTRLEN;

(c) S1[0]=MAXSTRLEN

2. 求子串 SubString(& Sub,S,pos,len)

求子串的过程即为复制字符序列的过程,将串 S 中从第 pos 个字符开始长度为 len 的字符序列复制到串 Sub 中。显然,本操作不会有需截断的情况,但有可能产生用户给出的参数不符合操作的初始条件,当参数非法时,返回 ERROR。其算法描述如算法 4.3 所示。

www.TopSage.com

大家网
TopSage.com

```
Status SubString(SString &Sub, SString S, int pos, int len) {
    // 用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。
    // 其中,1≤pos≤StrLength(S) 且 0≤len≤StrLength(S) - pos + 1。
    if (pos < 1 || pos > S[0] || len < 0 || len > S[0] - pos + 1)
        return ERROR;
    Sub[1..len] = S[pos..pos + len - 1];
    Sub[0] = len;     return OK;
} // SubString
```

算法 4.3

综上两个操作可见,在顺序存储结构中,实现串操作的原操作为"字符序列的复制",操作的时间复杂度基于复制的字符序列的长度。另一操作特点是,如果在操作中出现串值序列的长度超过上界 MAXSTRLEN 时,约定用截尾法处理,这种情况不仅在求联接串时可能发生,在串的其他操作中,如插入、置换等也可能发生。克服这个弊病惟有不限定串长的最大长度,即动态分配串值的存储空间。

4.2.2 堆分配存储表示

这种存储表示的特点是,仍以一组地址连续的存储单元存放串值字符序列,但它们的存储空间是在程序执行过程中动态分配而得。在 C 语言中,存在一个称之为"堆"的自由存储区,并由 C 语言的动态分配函数 malloc()和 free()来管理。利用函数 malloc()为每个新产生的串分配一块实际串长所需的存储空间,若分配成功,则返回一个指向起始地址的指针,作为串的基址,同时,为了以后处理方便,约定串长也作为存储结构的一部分。

```
    // - - - - - 串的堆分配存储表示 - - - - -
typedef struct {
    char   * ch;       // 若是非空串,则按串长分配存储区,否则 ch 为 NULL
    int    length;     // 串长度
}HString;
```

这种存储结构表示时的串操作仍是基于"字符序列的复制"进行的。例如,串复制操作 StrCopy(&T,S)的实现算法是,若串 T 已存在,则先释放串 T 所占空间,当串 S 不空时,首先为串 T 分配大小和串 S 长度相等的存储空间,然后将串 S 的值复制到串 T 中;又如串插入操作 StrInsert(&S,pos,T)的实现算法是,为串 S 重新分配大小等于串 S 和串 T 长度之和的存储空间,然后进行串值复制,如算法 4.4 所示。

```
Status StrInsert(HString &S, int pos, HString T) {
    // 1≤pos≤StrLength(S) + 1。在串 S 的第 pos 个字符之前插入串 T。
    if (pos<1 || pos > S.length + 1) return ERROR;   // pos 不合法
    if (T.length) {                    // T 非空,则重新分配空间,插入 T
      if (!(S.ch = (char * ) realloc(S.ch, (S.length + T.length) * sizeof(char))))
          exit(OVERFLOW);
      for (i = S.length - 1; i>= pos - 1; --i)   // 为插入 T 而腾出位置
          S.ch[i + T.length] = S.ch[i];
      S.ch[pos - 1..pos + T.length - 2] = T.ch[0..T.length - 1]; // 插入 T
```


www.TopSage.com

```
        S.length + = T.length;
    }
    return OK;
}// StrInsert
```

算法 4.4

以上两种存储表示通常为高级程序设计语言所采用。由于堆分配存储结构的串既有顺序存储结构的特点，处理方便，操作中对串长又没有任何限制，更显灵活，因此在串处理的应用程序中也常被选用。以下所示为只含最小操作子集的 HString 串类型的模块说明。

```
        // ===== ADT String 的表示与实现 =====
// ----- 串的堆分配存储表示 -----
typedef struct {
    char   *ch;       // 若是非空串，则按串长分配存储区，否则 ch 为 NULL
    int    length;    // 串长度
}HString;

// - - - - - 基本操作的函数原型说明 - - - - -
Status StrAssign (HString &T, char *chars);
    // 生成一个其值等于串常量 chars 的串 T
int StrLength (HString S);
    // 返回 S 的元素个数，称为串的长度。
int StrCompare(HString S, HString T)
    // 若 S >T，则返回值 >0；若 S = T，则返回值 = 0；若 S <T，则返回值 <0
Status ClearString (HString &S);
    // 将 S 清为空串，并释放 S 所占空间。
Status Concat (HString &T, HString S1, HString S2);
    // 用 T 返回由 S1 和 S2 联接而成的新串。
HString SubString (HString S, int pos, int len);
    // 1≤pos≤StrLength(S)且 0≤len≤StrLength(S) - pos + 1。
    // 返回串 S 的第 pos 个字符起长度为 len 的子串。

        // - - - - - 基本操作的算法描述 - - - - -
Status StrAssign(HString &T, char *chars) {
    // 生成一个其值等于串常量 chars 的串 T
    if (T.ch)  free(T.ch);                    // 释放 T 原有空间
    for (i = 0, c = chars;  c;  ++i, ++c);  // 求 chars 的长度 i
    if (!i) {T.ch = NULL;    T.length = 0; }
    else {
      if (!(T.ch = (char *)malloc(i* sizeof(char))))
        exit (OVERFLOW);
      T.ch[0..i-1] = chars[0..i-1];
      T.length = i;
    }
    return OK;
```


www.TopSage.com

大家网
TopSage.com

```
}// StrAssign

int StrLength(HString S) {
    // 返回 S 的元素个数,称为串的长度。
    return S.length;
}// StrLength

int StrCompare(HString S, HString T) {
    // 若 S >T,则返回值 >0;若 S = T,则返回值 = 0;若 S <T,则返回值 <0
    for (i = 0; i<S.length && i<T.length; ++i)
      if (S.ch[i] != T.ch[i])  return S.ch[i] - T.ch[i];
    return S.length-T.length;
}// StrCompare

Status ClearString(HString &S) {
    // 将 S 清为空串。
    if (S.ch)  {free(S.ch);    S.ch = NULL; }
    S.length = 0;
    return OK;
}// ClearString

Status Concat(HString &T, HString S1, HString S2) {
    // 用 T 返回由 S1 和 S2 联接而成的新串。
    if (T.ch)  free(T.ch);          // 释放旧空间
    if (!(T.ch = (char *) malloc((S1.length + S2.length) * sizeof(char))))
        exit (OVERFLOW);
    T.ch[0..S1.length - 1] = S1.ch[0..S1.length - 1];
    T.length = S1.length + S2.length;
    T.ch[S1.length..T.length - 1] = S2.ch[0..S2.length - 1];
    return OK;
    }// Concat

Status SubString(HString &Sub, HString S, int pos, int len) {
    // 用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。
    // 其中,1≤pos≤StrLength(S)且 0≤len≤StrLength(S) - pos + 1。
    if (pos < 1 || pos > S.length || len < 0 || len > S.length - pos + 1)
        return ERROR;
    if (Sub.ch)  free (Sub.ch);                          // 释放旧空间
    if (!len) {Sub.ch = NULL;  Sub.length = 0; }         // 空子串
    else {                                               // 完整子串
      Sub.ch = (char *)malloc(len * sizeof(char));
      Sub.ch[0..len - 1] = S.ch[pos - 1..pos + len - 2];
      Sub.length = len;
    }
    return OK;
}// SubString
```


www.TopSage.com

4.2.3 串的块链存储表示

和线性表的链式存储结构相类似，也可采用链表方式存储串值。由于串结构的特殊性——结构中的每个数据元素是一个字符，则用链表存储串值时，存在一个“结点大小”的问题，即每个结点可以存放一个字符，也可以存放多个字符。例如，图 4.2(a)是结点大小为 4(即每个结点存放 4 个字符)的链表，图 4.2(b)是结点大小为 1 的链表。当结点大小大于 1 时，由于串长不一定是结点大小的整倍数，则链表中的最后一个结点不一定全被串值占满，此时通常补上“#”或其他的非串值字符(通常“#”不属于串的字符集，是一个特殊的符号)。

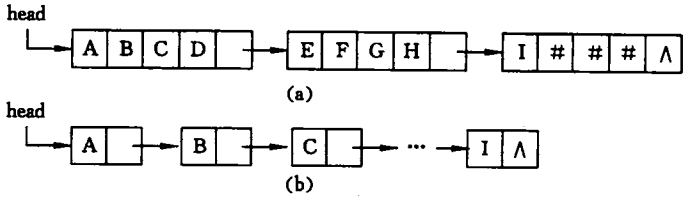


图 4.2 串值的链表存储方式

(a) 结点大小为 4 的链表；(b) 结点大小为 1 的链表

为了便于进行串的操作，当以链表存储串值时，除头指针外还可附设一个尾指针指示链表中的最后一个结点，并给出当前串的长度。称如此定义的串存储结构为**块链结构**，说明如下：

```
                // == == == 串的块链存储表示 == == ==
#define  CHUNKSIZE  80        // 可由用户定义的块大小
typedef  struct Chunk {
    char  ch[CHUNKSIZE];
    struct Chunk  * next;
}Chunk;
typedef struct {
    Chunk * head, * tail;     // 串的头和尾指针
    int     curlen;           // 串的当前长度
}LString;
```

由于在一般情况下，对串进行操作时，只需要从头向尾顺序扫描即可，则对串值不必建立双向链表。设尾指针的目的是为了便于进行联结操作，但应注意联结时需处理第一个串尾的无效字符。

在链式存储方式中，结点大小的选择和顺序存储方式的格式选择一样都很重要，它直接影响着串处理的效率。在各种串的处理系统中，所处理的串往往很长或很多，例如，一本书的几百万个字符，情报资料的成千上万个条目。这要求我们考虑串值的存储密度。存储密度可定义为

$$\text{存储密度}=\frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

www.TopSage.com

大家网 TopSage.com

显然，存储密度小（如结点大小为 1 时），运算处理方便，然而，存储占用量大。如果在串处理过程中需进行内、外存交换的话，则会因为内外存交换操作过多而影响处理的总效率。应该看到，串的字符集的大小也是一个重要因素。一般地，字符集小，则字符的机内编码就短，这也影响串值的存储方式的选取。

串值的链式存储结构对某些串操作，如联接操作等有一定方便之处，但总的说来不如另外两种存储结构灵活，它占用存储量大且操作复杂。此外，串值在链式存储结构时串操作的实现和线性表在链表存储结构中的操作类似，故在此不作详细讨论。

4.3 串的模式匹配算法

4.3.1 求子串位置的定位函数 Index(S, T, pos)

子串的定位操作通常称做串的**模式匹配**（其中 T 称为**模式串**），是各种串处理系统中最重要的操作之一。在 4.1 节中曾借用串的其他基本操作给出了定位函数的一种算法。根据算法 4.1 的基本思想，采用定长顺序存储结构，可以写出不依赖于其他串操作的匹配算法，如算法 4.5 所示。

```
int Index(SString S, SString T, int pos) {
    // 返回子串 T 在主串 S 中第 pos 个字符之后的位置。若不存在，则函数值为 0。
    // 其中，T 非空，1≤pos≤StrLength(S)。
    i = pos;    j = 1;
    while (i <= S[0] && j <= T[0]) {
      if (S[i] == T[j]) {++i;    ++j;}  // 继续比较后继字符
      else {i = i-j+2;    j = 1;}        // 指针后退重新开始匹配
    }
    if (j > T[0])  return  i-T[0];
    else return 0;
}// Index
```

算法 4.5

在算法 4.5 的函数过程中，分别利用计数指针 i 和 j 指示主串 S 和模式串 T 中当前正待比较的字符位置。算法的基本思想是：从主串 S 的第 pos 个字符起和模式的第一个字符比较之，若相等，则继续逐个比较后续字符；否则从主串的下一个字符起再重新和模式的字符比较之。依次类推，直至模式 T 中的每个字符依次和主串 S 中的一个连续的字符序列相等，则称**匹配成功**，函数值为和模式 T 中第一个字符相等的字符在主串 S 中的序号，否则称**匹配不成功**，函数值为零。图 4.3 展示了模式 T='abcac'和主串 S 的匹配过程（pos=1）。

算法 4.5 的匹配过程易于理解，且在某些应用场合，如文本编辑等，效率也较高，例如，在检查模式'STING'是否存在于下列主串中时，

'A STRING SEARCHING EXAMPLE CONSISTING OF SIMPLE TEXT'

上述算法中的 WHILE 循环次数（即进行单个字符比较的次数）为 41，恰好为（Index+T

www.TopSage.com

[0]－1)＋4，这就是说，除了主串中呈黑体的 4 个字符，每个字符比较了两次以外，其他字符均只和模式进行一次比较。在这种情况下，此算法的时间复杂度为 $O(n+m)$。其中 n 和 m 分别为主串和模式的长度。然而，在有些情况下，该算法的效率却很低。例如，当模式串为'00000001'，而主串为'00000000000000000000000000000000000000000000000000000000000000001'时，由于模式中前 7 个字符均为"0"，又，主串中前 52 个字符均为"0"，每趟比较都在模式的最后一个字符出现不等，此时需将指针 i 回溯到 i－6 的位置上，并从模式的第一个字符开始重新比较，整个匹配过程中指针 i 需回溯 45 次，则 WHILE 循环次数为 46 * 8(index * m)。可见，算法 4.5 在最坏情况下的时间复杂度为 $O(n*m)$。这种情况在只有 0、1 两种字符的文本串处理中经常出现，因为在主串中可能存在多个和模式串"部分匹配"的子串，因而引起指针 i 的多次回溯。01 串可以用在许多应用之中。比如，一些计算机的图形显示就是把画面表示为一个 01 串，一页书就是一个几百万个 0 和 1 组成的串。在二进位计算机上实际处理的都是 01 串。一个字符的 ASCII 码也可以看成是 8 个二进位的 01 串。包括汉字存储在计算机中处理时也是作为一个 01 串和其他的字符串一样看待。因此在下一节，我们将介绍另一种较好的模式匹配算法。

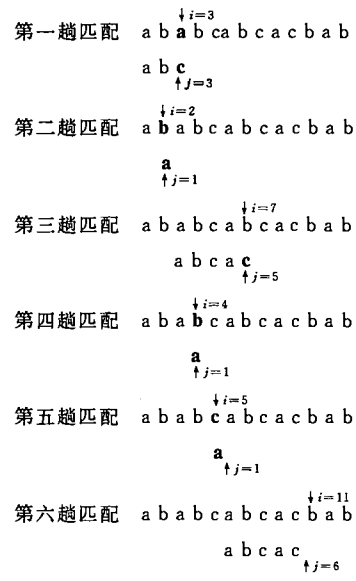


图 4.3 算法 4.5 的匹配过程

4.3.2 模式匹配的一种改进算法

这种改进算法是 D. E. Knuth 与 V. R. Pratt 和 J. H. Morris 同时发现的，因此人们称它为克努特—莫里斯—普拉特操作(简称为 KMP 算法)。此算法可以在 $O(n+m)$的时间数量级上完成串的模式匹配操作。其改进在于：每当一趟匹配过程中出现字符比较不等时，不需回溯 i 指针，而是利用已经得到的"部分匹配"的结果将模式向右"滑动"尽可能远的一段距离后，继续进行比较。下面先从具体例子看起。

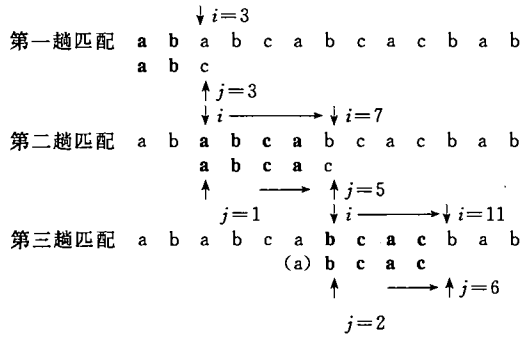


图 4.4 改进算法的匹配过程示例

回顾图 4.3 中的匹配过程示例，在第三趟的匹配中，当 $i=7$、$j=5$ 字符比较不等时，又从 $i=4$、$j=1$ 重新开始比较。然后，经仔细观察可发现，在 $i=4$ 和 $j=1$，$i=5$ 和 $j=1$ 以及 $i=6$ 和 $j=1$ 这 3 次比较都是不必进行的。因为从第三趟部分匹配的结果就可得出，主串中第 4、5 和 6 个字符必然是'b'、'c'和'a'(即模式串中第 2、3 和 4 个字符)。因为模式中的第一个字符是 a，因此它无需再和这 3 个字符进行比较，而仅需

www.TopSage.com

大家网 TopSage.com

将模式向右滑动3个字符的位置继续进行 $i=7$、$j=2$ 时的字符比较即可。同理，在第一趟匹配中出现字符不等时，仅需将模式向右移动两个字符的位置继续进行 $i=3$、$j=1$ 时的字符比较。由此，在整个匹配的过程中，i 指针没有回溯，如图4.4所示。

现在讨论一般情况。假设主串为 $'s_1 s_2 \cdots s_n'$，模式串为 $'p_1 p_2 \cdots p_m'$，从上例的分析可知，为了实现改进算法，需要解决下述问题：当匹配过程中产生"失配"（即 $s_i \neq p_j$）时，模式串"向右滑动"可行的距离多远，换句话说，当主串中第 i 个字符与模式中第 j 个字符"失配"（即比较不等）时，主串中第 i 个字符（i 指针不回溯）应与模式中哪个字符再比较？

假设此时应与模式中第 k（$k<j$）个字符继续比较，则模式中前 $k-1$ 个字符的子串必须满足下列关系式(4-2)，且不可能存在 $k'>k$ 满足下列关系式(4-2)

$$'p_1 p_2 \cdots p_{k-1}' = 's_{i-k+1} s_{i-k+2} \cdots s_{i-1}' \tag{4-2}$$

而已经得到的"部分匹配"的结果是

$$'p_{j-k+1} p_{j-k+2} \cdots p_{j-1}' = 's_{i-k+1} s_{i-k+2} \cdots s_{i-1}' \tag{4-3}$$

由式(4-2)和式(4-3)推得下列等式

$$'p_1 p_2 \cdots p_{k-1}' = 'p_{j-k+1} p_{j-k+2} \cdots p_{j-1}' \tag{4-4}$$

反之，若模式串中存在满足式(4-4)的两个子串，则当匹配过程中，主串中第 i 个字符与模式中第 j 个字符比较不等时，仅需将模式向右滑动至模式中第 k 个字符和主串中第 i 个字符对齐，此时，模式中头 $k-1$ 个字符的子串 $'p_1 p_2 \cdots p_{k-1}'$ 必定与主串中第 i 个字符之前长度为 $k-1$ 的子串 $'s_{i-k+1} s_{i-k+2} \cdots s_{i-1}'$ 相等，由此，匹配仅需从模式中第 k 个字符与主串中第 i 个字符比较起继续进行。

若令 $next[j]=k$，则 $next[j]$ 表明当模式中第 j 个字符与主串中相应字符"失配"时，在模式中需重新和主串中该字符进行比较的字符的位置。由此可引出模式串的 *next* 函数的定义：

$$next[j] = \begin{cases} 0 & \text{当 } j = 1 \text{ 时} \\ \text{Max}\{k \mid 1 < k < j \text{ 且 } 'p_1 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}'\} & \text{当此集合不空时} \\ 1 & \text{其他情况} \end{cases} \tag{4-5}$$

由此定义可推出下列模式串的 *next* 函数值：

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
$next[j]$	0	1	1	2	2	3	1	2

在求得模式的 *next* 函数之后，匹配可如下进行：假设以指针 i 和 j 分别指示主串和模式中正待比较的字符，令 i 的初值为 *pos*，j 的初值为1。若在匹配过程中 $s_i=p_j$，则 i 和 j 分别增1，否则，i 不变，而 j 退到 $next[j]$ 的位置再比较，若相等，则指针各自增1，否则 j 再退到下一个 *next* 值的位置，依次类推，直至下列两种可能：一种是 j 退到某个 *next* 值（$next[next[\cdots next[j]\cdots]]$）时字符比较相等，则指针各自增1，继续进行匹配；另一种是 j 退到值为零（即模式的第一个字符"失配"），则此时需将模式继续向右滑动一个位置，即从主串的下一个字符 s_{i+1} 起和模式重新开始匹配。图4.5所示正是上述匹配过程的一个例子。

KMP算法如算法4.6所示，它在形式上和算法4.5极为相似。不同之处仅在于：当

www.TopSage.com

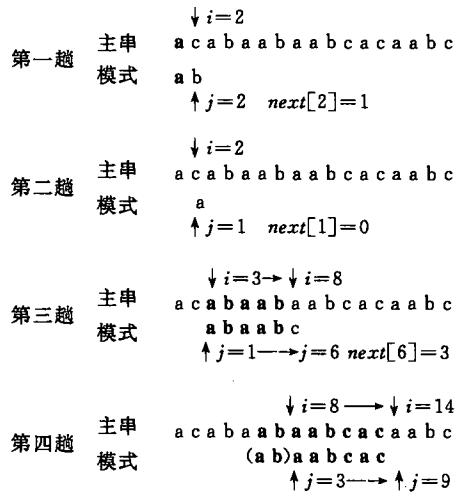


图 4.5　利用模式的 *next* 函数进行匹配的过程示例

匹配过程中产生“失配”时，指针 i 不变，指针 j 退回到 $next[j]$所指示的位置上重新进行比较，并且当指针 j 退至零时，指针 i 和指针 j 需同时增 1。即若主串的第 i 个字符和模式的第 1 个字符不等，应从主串的第 $i+1$ 个字符起重新进行匹配。

```
int Index_KMP(SString S, SString T, int pos) {
    // 利用模式串 T 的 next 函数求 T 在主串 S 中第 pos 个字符之后的位置的
    //  KMP 算法。其中，T 非空，1≤pos≤StrLength(S)。
    i = pos;    j = 1;
    while (i <= S[0] && j <= T[0]) {
        if ( j = = 0 || S[i] = = T[j]) { ++i;   ++j; }        // 继续比较后继字符
        else  j = next[j];                                    // 模式串向右移动
    }
    if (j > T[0])  return  i - T[0];                          // 匹配成功
    else return 0;
}// Index_KMP
```

算法　4.6

KMP 算法是在已知模式串的 *next* 函数值的基础上执行的，那么，如何求得模式串的 *next* 函数值呢？

从上述讨论可见，此函数值仅取决于模式串本身而和相匹配的主串无关。我们可从分析其定义出发用递推的方法求得 *next* 函数值。

由定义得知

$$next[1] = 0 \tag{4-6}$$

设　$next[j]=k$，这表明在模式串中存在下列关系：

$$'p_1 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}' \tag{4-7}$$

其中 k 为满足 $1<k<j$ 的某个值，并且不可能存在 $k'>k$ 满足等式(4-7)。此时 $next[j+1]=?$可能有两种情况：

www.TopSage.com

大家网 TopSage.com

(1) 若 $p_k=p_j$，则表明在模式串中

$$'p_1\cdots p_k' = 'p_{j-k+1}\cdots p_j' \tag{4-8}$$

并且不可能存在 $k'>k$ 满足等式(4-8)，这就是说 $next[j+1]=k+1$，即

$$next[j+1] = next[j]+1 \tag{4-9}$$

(2) 若 $p_k\neq p_j$，则表明在模式串中

$$'p_1\cdots p_k' \neq 'p_{j-k+1}\cdots p_j'$$

此时可把求 *next* 函数值的问题看成是一个模式匹配的问题，整个模式串既是主串又是模式串，而当前在匹配的过程中，已有 $p_{j-k+1}=p_1$，$p_{j-k+2}=p_2$，…，$p_{j-1}=p_{k-1}$，则当 $p_j\neq p_k$ 时应将模式向右滑动至以模式中的第 $next[k]$ 个字符和主串中的第 j 个字符相比较。若 $next[k]=k'$，且 $p_j=p_{k'}$，则说明在主串中第 $j+1$ 个字符之前存在一个长度为 k'（即 $next[k]$）的最长子串，和模式串中从首字符起长度为 k' 的子串相等，即

$$'p_1\cdots p_{k'}' = 'p_{j-k'+1}\cdots p_j' \qquad (1<k'<k<j) \tag{4-10}$$

这就是说 $next[j+1]=k'+1$ 即

$$next[j+1] = next[k]+1 \tag{4-11}$$

同理，若 $p_j\neq p_{k'}$，则将模式继续向右滑动直至将模式中第 $next[k']$ 个字符和 p_j 对齐，……，依次类推，直至 p_j 和模式中某个字符匹配成功或者不存在任何 $k'(1<k'<j)$ 满足等式(4-10)，则

$$next[j+1] = 1 \tag{4-12}$$

例如，图 4.6 中的模式串，已求得前 6 个字符的 *next* 函数值，现求 $next[7]$，因为 $next[6]=3$，又 $p_6\neq p_3$，则需比较 p_6 和 p_1（因为 $next[3]=1$），这相当于将子串模式向右滑动。由于 $p_6\neq p_1$，而且 $next[1]=0$，所以 $next[7]=1$，而因为 $p_7=p_1$，则 $next[8]=2$。

j	1	2	3	4	5	6	7	8
模式	a	b	a	a	b	c	a	c
$next[j]$	0	1	1	2	2	3	1	2

(a b a)

(a)

图 4.6 模式串的 *next* 函数值

根据上述分析所得结果(式(4-6)、(4-9)、(4-11)和(4-12))，仿照 KMP 算法，可得到求 *next* 函数值的算法，如算法 4.7 所示。

```
void get_next(SString T, int next[ ]) {
    // 求模式串 T 的 next 函数值并存入数组 next。
    i = 1;    next[1] = 0;    j = 0;
    while (i < T[0]) {
      if (j = = 0 || T[i] = = T[j]) { ++ i;   ++ j;  next[i] = j; }
      else  j = next[j];
    }
}// get_next
```

算法 4.7

算法 4.7 的时间复杂度为 $O(m)$。通常，模式串的长度 m 比主串的长度 n 要小得多，因此，对整个匹配算法来说，所增加的这点时间是值得的。

最后，要说明两点：

(1) 虽然算法 4.5 的时间复杂度是 $O(n*m)$，但在一般情况下，其实际的执行时间近

www.TopSage.com

似于 $O(n+m)$，因此至今仍被采用。KMP 算法仅当模式与主串之间存在许多“部分匹配”的情况下才显得比算法 4.5 快得多。但是 KMP 算法的最大特点是指示主串的指针不需回溯，整个匹配过程中，对主串仅需从头至尾扫描一遍，这对处理从外设输入的庞大文件很有效，可以边读入边匹配，而无需回头重读。

(2) 前面定义的 *next* 函数在某些情况下尚有缺陷。例如模式'a a a a b'在和主串'a a a b a a a a b'匹配时，当 $i=4$、$j=4$ 时 s. ch[4]≠t. ch[4]，由 *next*[j] 的指示还需进行 $i=4$、$j=3$，$i=4$、$j=2$，$i=4$、$j=1$ 这 3 次比较。实际上，因为模式中第 1、2、3 个字符和第 4 个字符都相等，因此不需要再和主串中第 4 个字符相比较，而可以将模式一气向右滑动 4 个字符的位置直接进行 $i=5$、$j=1$ 时的字符比较。这就是说，若按上述定义得到 *next*[j]=k，而模式中 $p_j=p_k$，则当主串中字符 s_i 和 p_j 比较不等时，不需要再和 p_k 进行比较，而直接和 $P_{next[k]}$ 进行比较，换句话说，此时的 *next*[j] 应和 *next*[k] 相同。由此可得计算 *next* 函数修正值的算法如算法 4.8 所示。此时匹配算法不变。

j	1 2 3 4 5
模式	*a a a a b*
next[j]	0 1 2 3 4
nextval[j]	0 0 0 0 4

```
void get_nextval(SString T, int nextval[ ]) {
    // 求模式串 T 的 next 函数修正值并存入数组 nextval。
    i = 1;     nextval[1] = 0;     j = 0;
    while (i < T[0]) {
      if (j == 0 || T[i] == T[j]) {
          ++i;  ++j;
          if (T[i] != T[j])  nextval[i] = j;
          else  nextval[i] = nextval[j];
      }
      else  j = nextval[j];
    }
}// get_nextval
```

算法 4.8

4.4 串操作应用举例

4.4.1 文本编辑

文本编辑程序是一个面向用户的系统服务程序，广泛用于源程序的输入和修改，甚至用于报刊和书籍的编辑排版以及办公室的公文书信的起草和润色。文本编辑的实质是修改字符数据的形式或格式。虽然各种文本编辑程序的功能强弱不同，但是其基本操作是一致的，一般都包括串的查找、插入和删除等基本操作。

为了编辑的方便，用户可以利用换页符和换行符把文本划分为若干页，每页有若干行（当然，也可不分页而把文件直接划成若干行）。我们可以把文本看成是一个字符串，称为文本串。页则是文本串的子串，行又是页的子串。

比如有下列一段源程序：

www.TopSage.com

大家网
TopSage.com

```
main (){
  float  a,b,max;
  scanf ("%f,%f", &a, &b);
  if a>b  max=a;
  else max=b;
};
```

我们可以把此程序看成是一个文本串。输入到内存后如图 4.7 所示。图中"↵"为换行符。

201

m	a	i	n	(	)	{	↵			f	l	o	a	t		a	,	b	,
m	a	x	;	↵			s	c	a	n	f	(	"	%	f	,	%	f	"
,	&	a	,	&	b	)	;	↵			i	f		a	>	b			m
a	x	=	a	;	↵			e	l	s	e			m	a	x	=	b	;
↵	}	↵																	

图 4.7　文本格式示例

为了管理文本串的页和行，在进入文本编辑的时候，编辑程序先为文本串建立相应的页表和行表，即建立各子串的存储映像。页表的每一项给出了页号和该页的起始行号。而行表的每一项则指示每一行的行号、起始地址和该行子串的长度。假设图 4.7 所示文本串只占一页，且起始行号为 100，则该文本串的行表如图 4.8 所示。

行　号	起始地址	长　度
100	201	8
101	209	17
102	226	24
103	250	17
104	267	15
105	282	2

图 4.8　图 4.7 所示文本串的行表

文本编辑程序中设立页指针、行指针和字符指针，分别指示当前操作的页、行和字符。如果在某行内插入或删除若干字符，则要修改行表中该行的长度。若该行的长度超出了分配给它的存储空间，则要为该行重新分配存储空间，同时还要修改该行的起始位置。如果要插入或删除一行，就要涉及行表的插入或删除。若被删除的行是所在页的起始行，则还要修改页表中相应页的起始行号(修改为下一行的行号)。为了查找方便，行表是按行号递增顺序存储的，因此，对行表进行的插入或删除运算需移动操作位置以后的全部表项。页表的维护与行表类似，在此不再赘述。由于访问是以页表和行表作为索引的，所以在作行和页的删除操作时，可以只对行表和页表作相应的修改，不必删除所涉及的字符。这可以节省不少时间。

以上概述了文本编辑程序中的基本操作。其具体的算法，读者可在学习本章之后自行编写。

www.TopSage.com

4.4.2 建立词索引表

信息检索是计算机应用的重要领域之一。由于信息检索的主要操作是在大量的存放在磁盘上的信息中查询一个特定的信息，为了提高查询效率，一个重要的问题是建立一个好的索引系统。例如我们在 1.1 节中提到过的图书馆书目检索系统中有 3 张索引表，分别可按书名、作者名和分类号编排。在实际系统中，按书名检索并不方便，因为很多内容相似的书籍其书名不一定相同。因此较好的办法是建立“书名关键词索引”。

例如，与图 4.9(a)中书目相应的关键词索引表如图 4.9(b)所示，读者很容易从关键词索引表中查询到他所感兴趣的书目。为了便于查询，可设定此索引表为按词典有序的线性表。下面要讨论的是如何从书目文件生成这个有序词表。

书号	书　名
005	Computer Data Structures
010	Introduction to Data Structures
023	Fundamentals of Data Structures
034	The Design and Analysis of Computer Algorithms
050	Introduction to Numerical Analysis
067	Numerical Analysis

(a)

关键词	书号索引
algorithms	034
analysis	034,050,067
computer	005,034
data	005,010,023
design	034
fundamentals	023
introduction	010,050
numerical	050,067
structures	005,010,023

(b)

图 4.9　书目文件及其关键词索引表

(a) 书目文件；(b) 关键词索引表

重复下列操作直至文件结束：

(1) 从书目文件中读入一个书目串；

(2) 从书目串中提取所有关键词插入词表；

(3) 对词表中的每一个关键词，在索引表中进行查找并作相应的插入操作。

为识别从书名串中分离出来的单词是否是关键词，需要一张常用词表(在英文书名中的“常用词”指的是诸如“an”、“a”、“of”、“the”等词)。顺序扫描书名串，首先分离单词，然后查找常用词表，若不和表中任一词相等，则为关键词，插入临时存放关键词的词表中。

在索引表中查询关键词时可能出现两种情况：其一是索引表上已有此关键词的索引项，只要在该项中插入书号索引即可；其二是需在索引表中插入此关键词的索引项，插入应按字典有序原则进行。下面就重点讨论这第三个操作的具体实现。

首先设定数据结构。

词表为线性表，只存放一本书的书名中若干关键词，其数量有限，则采用顺序存储结构即可，其中每个词是一个字符串。

索引表为有序表，虽是动态生成，在生成过程中需频繁进行插入操作，但考虑索引表

www.TopSage.com

大家网 TopSage.com

主要为查找用，为了提高查找效率(采用第 9 章中将讨论的折半查找)，宜采用顺序存储结构；表中每个索引项包含两个内容：其一是关键词，因索引表为常驻结构，则应考虑节省存储，采用堆分配存储表示的串类型；其二是书号索引，由于书号索引是在索引表的生成过程中逐个插入，且不同关键词的书号索引个数不等，甚至可能相差很多，则宜采用链表结构的线性表。

```
#define  MaxBookNum  1000      // 假设只对 1000 本书建索引表
#define  MaxKeyNum   2500      // 索引表的最大容量
#define  MaxLineLen  500       // 书目串的最大长度
#define  MaxWordNum  10        // 词表的最大容量

typedef  struct {
    char  *item[ ];  // 字符串的数组
    int   last;      // 词表的长度
}WordListType;       // 词表类型(顺序表)
typedef int ElemType; // 定义链表的数据元素类型为整型(书号类型)
typedef  struct {
    HString  key;    // 关键词
    LinkList bnolist; // 存放书号索引的链表
}IdxTermType;        // 索引项类型
typedef  struct {
    IdxTermType  item[MaxKeyNum+1];
    int          last;
}IdxListType;        // 索引表类型(有序表)

    // 主要变量
char   *buf;         // 书目串缓冲区
WordListType  wdlist; // 词表

    // 基本操作
void InitIdxList (IdxListType &idxlist);
    // 初始化操作，置索引表 idxlist 为空表，且在 idxlist.item[0]设一空串
void GetLine (FILE f);
    // 从文件 f 读入一个书目信息到书目串缓冲区 buf
void ExtractKeyWord (ElemType &bno);
    // 从 buf 中提取书名关键词到词表 wdlist，书号存入 bno
Status InsIdxList (IdxListType &idxlist,  ElemType bno);
    // 将书号为 bno 的书名关键词按词典顺序插入索引表 idxlist
void PutText (FILE g,  IdxListType idxlist);
    // 将生成的索引表 idxlist 输出到文件 g

void main() {  // 主函数
    if (f = openf ("BookInfo.txt", "r"))
      if (g = openf ("BookIdx.txt", "w")) {
          InitIdxList (idxlist);              // 初始化索引表 idxlist 为空表
          while (!feof (f)) {
            GetLine (f);                      // 从文件 f 读入一个书目信息到 buf
            ExtractKeyWord (BookNo);          // 从 buf 提取关键词到词表，书号存入 BookNo
```


www.TopSage.com

```
            InsIdxList (idxlist, BookNo);     // 将书号为 BookNo 的关键词插入索引表
          }
          PutText (g, idxlist);               // 将生成的索引表 idxlist 输出到文件 g
      }
}// main
```

算法 4.9

为实现在索引表上进行插入，要先实现下列操作：

```
void GetWord ( int i,  HString &wd);
    // 用 wd 返回词表 wdlist 中第 i 个关键词。
int Locate (IdxListType idxlist,  HString wd,  Boolean &b);
    // 在索引表 idxlist 中查询是否存在与 wd 相等的关键词。若存在，则返回其在索引表
    // 中的位置，且 b 取值 TRUE；否则返回插入位置，且 b 取值 FALSE
void InsertNewKey (IdxListType &idxlist,  int i,  HString wd);
    // 在索引表 idxlist 的第 i 项上插入新关键词 wd，并初始化书号索引的链表为空表
Status InsertBook (IdxListType &idxlist,  int i,  int bno);
    // 在索引表 idxlist 的第 i 项中插入书号为 bno 的索引
```

由此可得索引表的插入算法如算法 4.10 所示。

```
Status InsIdxList (IdxListType &idxlist,  int bno) {
    for (i = 0;  i<wdlist.last;  ++i) {
        GetWord (i, wd);     j = Locate (idxlist, wd, b);
        if (!b)  InsertNewKey (idxlist, j, wd);          //  插入新的索引项
        if (!InsertBook (idxlist,j,bno)) return OVERFLOW; //  插入书号索引
    }
    return OK;
}// InsertIdxList
```

算法 4.10

其中 4 个操作的具体实现分别如算法 4.11、4.12、4.13 和 4.14 所示。

```
void GetWord ( int i,  HString &wd) {
  p = *(wdlist.item + i);        // 取词表中第 i 个字符串
  StrAssign (wd, p);             // 生成关键字字符串
}// GetWord
```

算法 4.11

```
int Locate (IdxListType &idxlist, HString wd, Boolean &b) {
    for ( i = idxlist.last - 1;
          (m = StrCompare (idxlist.item[i].key, wd)) > 0;  --i );
    if (m == 0)  {b = TRUE;    return i;  }          // 找到
    else {b = FALSE;    return i+1;  }
}// Locate
```

算法 4.12

www.TopSage.com

大家网
TopSage.com

```
void InsertNewKey (int i,  StrType wd) {
    for (j = idxlist.last - 1;  j>= i;  --j)          // 后移索引项
         idxlist.item[j + 1] = idxlist.item[j];
    // 插入新的索引项
    StrCopy (idxlist.item[i].key, wd);                // 串赋值
    InitList (idxlist.item[i].bnolist);               // 初始化书号索引表为空表
    ++idxlist.last;
}// InsertNewKey
```

算法 4.13

```
Status InsertBook (IdxListType &idxlist,  int i,  int bno) {
    if (!MakeNode (p, bno)) return ERROR;             // 分配失败
    Append (idxlist.item[i].bnolist, p);              // 插入新的书号索引
    return OK;
}// InsertBook
```

算法 4.14

www.TopSage.com

第5章　数组和广义表

前几章讨论的线性结构中的数据元素都是非结构的原子类型，元素的值是不再分解的。本章讨论的两种数据结构——数组和广义表可以看成是线性表在下述含义上的扩展：表中的数据元素本身也是一个数据结构。

数组是读者已经很熟悉的一种数据类型，几乎所有的程序设计语言都把数组类型设定为固有类型。本章以抽象数据类型的形式讨论数组的定义和实现，使读者加深对数组类型的理解。

5.1　数组的定义

类似于线性表，抽象数据类型数组可形式地定义为：

ADT Array {

数据对象：$j_i = 0,\cdots,b_i - 1,\quad i = 1,2,\cdots,n$，

$D = \{a_{j_1 j_2 \cdots j_n} \mid n(>0)$称为数组的维数，$b_i$ 是数组第 i 维的长度，

j_i 是数组元素的第 i 维下标，$a_{j_1 j_2 \cdots j_n} \in ElemSet\}$

数据关系：$R = \{R1, R2, \cdots, Rn\}$

$Ri = \{< a_{j_1 \cdots j_i \cdots j_n}, a_{j_1 \cdots j_i + 1 \cdots j_n} > \mid$

$0 \leqslant j_k \leqslant b_k - 1,\quad 1 \leqslant k \leqslant n \quad$且 $k \neq i$，

$0 \leqslant j_i \leqslant b_i - 2$，

$a_{j_1 \cdots j_i \cdots j_n}, a_{j_1 \cdots j_i + 1 \cdots j_n} \in D,\ i = 2,\cdots,n\}$

基本操作：

InitArray(&A, n, bound1, …, boundn)

操作结果：若维数 n 和各维长度合法，则构造相应的数组 A，并返回 OK。

DestroyArray(&A)

操作结果：销毁数组 A。

Value(A, &e, index1, …, indexn)

初始条件：A 是 n 维数组，e 为元素变量，随后是 n 个下标值。

操作结果：若各下标不超界，则 e 赋值为所指定的 A 的元素值，并返回 OK。

Assign(&A, e, index1, …, indexn)

初始条件：A 是 n 维数组，e 为元素变量，随后是 n 个下标值。

操作结果：若下标不超界，则将 e 的值赋给所指定的 A 的元素，并返回 OK。

} **ADT** Array

这是一个 C 语言风格的定义。从上述定义可见，n 维数组中含有 $\prod_{i=1}^{n} b_i$ 个数据元素，每个元素都受着 n 个关系的约束。在每个关系中，元素 $a_{j_1 j_2 \cdots j_n}(0 \leqslant j_i \leqslant b_i - 2)$都有一个直接后继元素。因此，就其单个关系而言，这 n 个关系仍是线性关系。和线性表一样，所有的数据元素都必须属于同一数据类型。数组中的每个数据元素都对应于一组下标(j_1，

www.TopSage.com

大家网 TopSage.com

$j_2,\cdots,j_n$)，每个下标的取值范围是 $0\leqslant j_i\leqslant b_i-1$，$b_i$ 称为第 i 维的长度($i=1,2,\cdots,n$)。显然，当 $n=1$ 时，n 维数组就退化为定长的线性表。反之，n 维数组也可以看成是线性表的推广。由此，我们也可以从另一个角度来定义 n 维数组。

我们可以把二维数组看成是这样一个定长线性表：它的每个数据元素也是一个定长线性表。例如。图 5.1(a)所示是一个二维数组，以 m 行 n 列的矩阵形式表示。它可以看成是一个线性表

$$A=(\alpha_0,\alpha_1,\cdots,\alpha_p)\qquad(p=m-1\text{ 或 }n-1)$$

其中每个数据元素 α_j 是一个列向量形式的线性表

$$\alpha_j=(a_{0j},a_{1j},\cdots,a_{m-1,j})\qquad 0\leqslant j\leqslant n-1$$

(如图 5.1(b)所示)或者 α_i 是一个行向量形式的线性表

$$\alpha_i=(a_{i0},a_{i1},\cdots,a_{i,n-1})\qquad 0\leqslant i\leqslant m-1$$

(如图 5.1(c)所示)。在 C 语言中，一个二维数组类型可以定义为其分量类型为一维数组类型的一维数组类型，也就是说，

```
typedef  ElemType  Array2[m][n];
```

等价于

```
typedef  ElemType  Array1[n];
typedef  Array1    Array2[m];
```

同理，一个 n 维数组类型可以定义为其数据元素为 $n-1$ 维数组类型的一维数组类型。

$$A_{m\times n}=\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1}\\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1}\\ \vdots & \vdots & \vdots & & \vdots\\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1}\end{bmatrix}$$

(a)

$$A_{m\times n}=\begin{bmatrix}\begin{bmatrix}a_{00}\\ a_{10}\\ \vdots\\ a_{m-1,0}\end{bmatrix}\begin{bmatrix}a_{01}\\ a_{11}\\ \vdots\\ a_{m-1,1}\end{bmatrix}\cdots\begin{bmatrix}a_{0,n-1}\\ a_{1,n-1}\\ \vdots\\ a_{m-1,n-1}\end{bmatrix}\end{bmatrix}$$

(b)

$$A_{m\times n}=((a_{00}a_{01}\cdots a_{0,n-1}),(a_{10}a_{11}\cdots a_{1,n-1}),\cdots,(a_{m-1,0}a_{m-1,1}\cdots a_{m-1,n-1}))$$

(c)

图 5.1　二维数组图例

(a)矩阵形式表示；　(b)列向量的一维数组；　(c)行向量的一维数组

数组一旦被定义，它的维数和维界就不再改变。因此，除了结构的初始化和销毁之外，数组只有存取元素和修改元素值的操作。

5.2　数组的顺序表示和实现

由于数组一般不作插入或删除操作，也就是说，一旦建立了数组，则结构中的数据元素个数和元素之间的关系就不再发生变动。因此，采用顺序存储结构表示数组是自然的事了。

由于存储单元是一维的结构，而数组是个多维的结构，则用一组连续存储单元存放数组的数据元素就有个次序约定问题。例如图 5.1 (a) 的二维数组可以看成如图 5.1(c)的

www.TopSage.com

一维数组，也可看成如图 5.1(b)的一维数组。对应地，对二维数组可有两种存储方式：一种以列序为主序(column major order)的存储方式，如图 5.2(a)所示；一种是以行序为主序(row major order)的存储方式，如图 5.2(b)所示。在扩展 BASIC、PL/1、COBOL、PASCAL 和 C 语言中，用的都是以行序为主序的存储结构，而在 FORTRAN 语言中，用的是以列序为主序的存储结构。

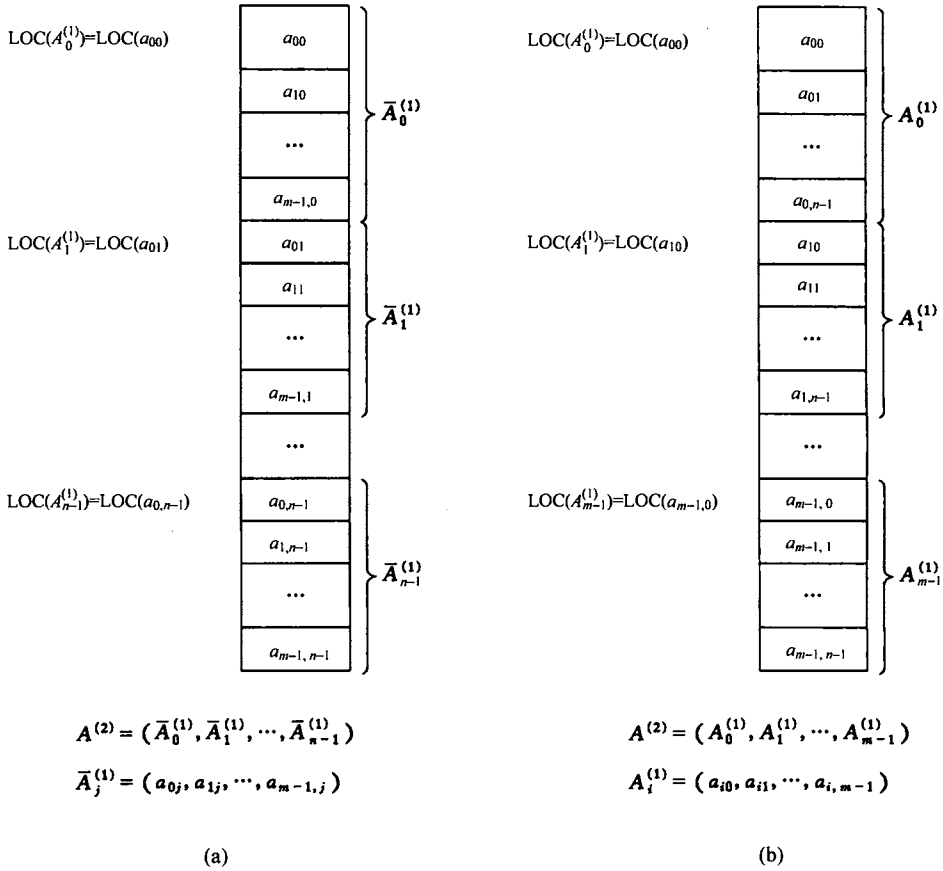


图 5.2　二维数组的两种存储方式

(a) 以列序为主序；　(b) 以行序为主序

由此，对于数组，一旦规定了它的维数和各维的长度，便可为它分配存储空间。反之，只要给出一组下标便可求得相应数组元素的存储位置。下面仅用以行序为主序的存储结构为例予以说明。

假设每个数据元素占 L 个存储单元，则二维数组 A 中任一元素 a_{ij} 的存储位置可由下式确定

$$\mathrm{LOC}(i,j)=\mathrm{LOC}(0,0)+(b_2\times i+j)L \tag{5-1}$$

式中，LOC(i,j)是 a_{ij} 的存储位置；LOC$(0,0)$是 a_{00} 的存储位置，即二维数组 A 的起始存储位置，也称为基地址或基址。

将式(5-1)推广到一般情况，可得到 n 维数组的数据元素存储位置的计算公式：

www.TopSage.com

大家网 TopSage.com

$$\begin{aligned}\mathrm{LOC}(j_1,j_2,\cdots,j_n)&=\mathrm{LOC}(0,0,\cdots,0)+(b_2\times\cdots\times b_n\times j_1+b_3\times\cdots\times b_n\times j_2\\&\qquad+\cdots+b_n\times j_{n-1}+j_n)L\\&=\mathrm{LOC}(0,0,\cdots,0)+\Big(\sum_{i=1}^{n-1}j_i\prod_{k=i+1}^{n}b_k+j_n\Big)L\end{aligned}$$

可缩写成

$$\mathrm{LOC}(j_1,j_2,\cdots,j_n)=\mathrm{LOC}(0,0,\cdots,0)+\sum_{i=1}^{n}c_ij_i \tag{5-2}$$

其中 $c_n=L, c_{i-1}=b_i\times c_i,\quad 1<i\leqslant n$。

式(5-2)称为 n 维数组的映像函数。容易看出，数组元素的存储位置是其下标的线性函数，一旦确定了数组的各维的长度，c_i 就是常数。由于计算各个元素存储位置的时间相等，所以存取数组中任一元素的时间也相等。我们称具有这一特点的存储结构为随机存储结构。

下面是数组的顺序存储表示和实现。

```
//----- 数组的顺序存储表示 -----
#include <stdarg.h>          // 标准头文件，提供宏 va_start、va_arg 和 va_end，
                             // 用于存取变长参数表
#define  MAX_ARRAY_DIM  8    // 假设数组维数的最大值为 8
typedef struct {
    ElemType   *base;        // 数组元素基址，由 InitArray 分配
    int        dim;          // 数组维数
    int        *bounds;      // 数组维界基址，由 InitArray 分配
    int        *constants;   // 数组映像函数常量基址，由 InitArray 分配
}Array;

// ----- 基本操作的函数原型说明 -----
Status InitArray(Array &A, int dim, ...);
    // 若维数 dim 和随后的各维长度合法，则构造相应的数组 A，并返回 OK。
Status DestroyArray(Array &A);
    // 销毁数组 A。
Status Value(Array A, ElemType &e, ...);
    // A 是 n 维数组，e 为元素变量，随后是 n 个下标值。
    // 若各下标不超界，则 e 赋值为所指定的 A 的元素值，并返回 OK。
Status Assign(Array &A, ElemType e, ...);
    // A 是 n 维数组，e 为元素变量，随后是 n 个下标值。
    // 若下标不超界，则将 e 的值赋给所指定的 A 的元素，并返回 OK。
// ----- 基本操作的算法描述 -----
Status InitArray(Array &A, int dim, ...) {
    // 若维数 dim 和各维长度合法，则构造相应的数组 A，并返回 OK
    if (dim < 1 || dim > MAX_ARRAY_DIM)  return ERROR;
    A.dim = dim;
    A.bounds = (int *)malloc(dim * sizeof(int));
    if (!A.bounds)  exit(OVERFLOW);
    // 若各维长度合法，则存入 A.bounds，并求出 A 的元素总数 elemtotal
    elemtotal = 1;
```


www.TopSage.com

```
    va_start(ap, dim);          // ap 为 va_list 类型,是存放变长参数表信息的数组
    for (i=0; i<dim; ++i) {
        A.bounds[i] = va_arg(ap, int);
        if (A.bounds[i]<0) return UNDERFLOW;
        elemtotal *= A.bounds[i];
    }
    va_end(ap);
    A.base = (ElemType *)malloc(elemtotal * sizeof(ElemType));
    if (!A.base) exit(OVERFLOW);
    // 求映像函数的常数 c_i,并存入 A.constants[i-1], i=1,…,dim
    A.constants = (int *)malloc(dim * sizeof(int));
    if (!A.constants)exit(OVERFLOW);
    A.constants[dim-1] = 1;  // L=1,指针的增减以元素的大小为单位
    for ( i=dim-2;  i>=0;  --i )
      A.constants[i] = A.bounds[i+1] * A.constants[i+1];
    return OK;
}

Status DestroyArray(Array &A) {
    // 销毁数组 A。
    if (!A.base) return ERROR;
    free(A.base);     A.base = NULL;
    if !(A.bounds) return ERROR;
    free(A.bounds);     A.bounds = NULL;
    if !(A.constants)  return ERROR;
    free(A.constants);     A.constants = NULL;
    return OK;
}

Status Locate(Array A, va_list ap, int &off) {
    // 若 ap 指示的各下标值合法,则求出该元素在 A 中相对地址 off
    off = 0;
    for ( i=0;  i<A.dim;  ++i ) {
        ind = va_arg(ap, int);
        if (ind<0 || ind>=A.bounds[i])  return OVERFLOW;
        off += A.constants[i] * ind;
    }
    return OK;
}

Status Value(Array A, ElemType &e, …) {
    // A 是 n 维数组,e 为元素变量,随后是 n 个下标值。
    // 若各下标不超界,则 e 赋值为所指定的 A 的元素值,并返回 OK。
    va_start(ap, e);
    if ((result = Locate(A, ap, off))<=0) return result;
    e = *(A.base + off);
    return OK;
```


www.TopSage.com

大家网
TopSage.com

```
}

Status Assign(Array &A, ElemType e, …) {
    // A 是 n 维数组,e 为元素变量,随后是 n 个下标值。
    // 若下标不超界,则将 e 的值赋给所指定的 A 的元素,并返回 OK。
    va_start(ap, e);
    if ((result = Locate(A, ap, off))<= 0) return result;
    *(A.base + off) = e;
    return OK;
}
```

5.3 矩阵的压缩存储

矩阵是很多科学与工程计算问题中研究的数学对象。在此,我们感兴趣的不是矩阵本身,而是如何存储矩阵的元,从而使矩阵的各种运算能有效地进行。

通常,用高级语言编制程序时,都是用二维数组来存储矩阵元。有的程序设计语言中还提供了各种矩阵运算,用户使用时都很方便。

然而,在数值分析中经常出现一些阶数很高的矩阵,同时在矩阵中有许多值相同的元素或者是零元素。有时为了节省存储空间,可以对这类矩阵进行**压缩存储**。所谓压缩存储是指:为多个值相同的元只分配一个存储空间;对零元不分配空间。

假若值相同的元素或者零元素在矩阵中的分布有一定规律,则我们称此类矩阵为**特殊矩阵**;反之,称为**稀疏矩阵**。下面分别讨论它们的压缩存储。

5.3.1 特殊矩阵

若 n 阶矩阵 A 中的元满足下述性质

$$a_{ij} = a_{ji} \qquad 1 \leqslant i,j \leqslant n$$

则称为 n 阶对称矩阵。

对于对称矩阵,我们可以为每一对对称元分配一个存储空间,则可将 n^2 个元压缩存储到 $n(n+1)/2$ 个元的空间中。不失一般性,我们可以行序为主序存储其下三角(包括对角线)中的元。

假设以一维数组 $sa[n(n+1)/2]$作为 n 阶对称矩阵 A 的存储结构,则 $sa[k]$和矩阵元 a_{ij} 之间存在着一一对应的关系:

$$k = \begin{cases} \dfrac{i(i-1)}{2} + j - 1 & \text{当 } i \geqslant j \\ \dfrac{j(j-1)}{2} + i - 1 & \text{当 } i < j \end{cases} \tag{5-3}$$

对于任意给定一组下标(i,j),均可在 sa 中找到矩阵元 a_{ij},反之,对所有的 $k=0,1,2,\cdots,\dfrac{n(n+1)}{2}-1$,都能确定 $sa[k]$中的元在矩阵中的位置(i,j)。由此,称 $sa[n(n+1)/2]$为 n 阶对称矩阵 A 的压缩存储(见图 5.3)。

www.TopSage.com

	a_{11}	a_{21}	a_{22}	a_{31}	…	$a_{n,1}$	…	$a_{n,n}$
$k=$	0	1	2	3		$\frac{n(n-1)}{2}$		$\frac{n(n+1)}{2}-1$

图 5.3　对称矩阵的压缩存储

这种压缩存储的方法同样也适用于三角矩阵。所谓下(上)三角矩阵是指矩阵的上(下)三角(不包括对角线)中的元均为常数 c 或零的 n 阶矩阵。则除了和对称矩阵一样，只存储其下(上)三角中的元之外，再加一个存储常数 c 的存储空间即可。

在数值分析中经常出现的还有另一类特殊矩阵是对角矩阵。在这种矩阵中，所有的非零元都集中在以主对角线为中心的带状区域中。即除了主对角线上和直接在对角线上、下方若干条对角线上的元之外，所有其他的元皆为零。如图 5.4 所示。对这种矩阵，我们也可按某个原则(或以行为主，或以对角线的顺序)将其压缩存储到一维数组上。

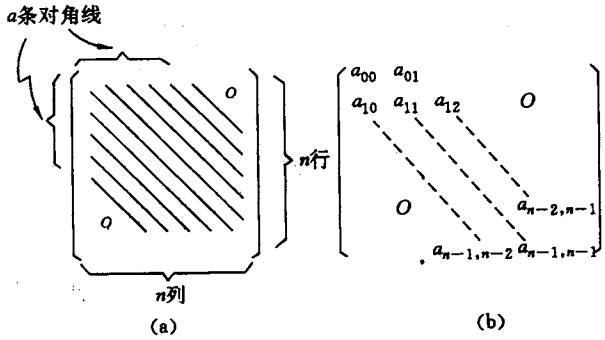


图 5.4　对角矩阵

(a)一般情形；(b)三对角矩阵

在所有这些我们统称为特殊矩阵的矩阵中，非零元的分布都有一个明显的规律，从而我们都可将其压缩存储到一维数组中，并找到每个非零元在一维数组中的对应关系。

然而，在实际应用中我们还经常会遇到另一类矩阵，其非零元较零元少，且分布没有一定规律，我们称之为稀疏矩阵。这类矩阵的压缩存储就要比特殊矩阵复杂。这就是下一节我们要讨论的问题。

5.3.2　稀疏矩阵

什么是稀疏矩阵？人们无法给出确切的定义，它只是一个凭人们的直觉来了解的概念。假设在 $m\times n$ 的矩阵中，有 t 个元素不为零。令 $\delta=\frac{t}{m\times n}$，称 δ 为矩阵的**稀疏因子**。通常认为 $\delta\leqslant 0.05$ 时称为稀疏矩阵。矩阵运算的种类很多，在下列抽象数据类型稀疏矩阵的定义中，只列举了几种常见的运算。

抽象数据类型稀疏矩阵的定义如下：

```
ADT SparseMatrix {
    数据对象:D = {a_ij | i = 1,2,…,m; j = 1,2,…,n;
                  a_i,j ∈ ElemSet, m 和 n 分别称为矩阵的行数和列数}
```


www.TopSage.com

大家网
TopSage.com

数据关系:R = {Row, Col}

Row = {$<a_{i,j},a_{i,j+1}>$ | $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n-1$}

Col = {$<a_{i,j},a_{i+1,j}>$ | $1 \leqslant i \leqslant m-1$, $1 \leqslant j \leqslant n$}

基本操作:

CreateSMatrix(&M);

操作结果:创建稀疏矩阵 M。

DestroySMatrix(&M);

初始条件:稀疏矩阵 M 存在。

操作结果:销毁稀疏矩阵 M。

PrintSMatrix(M);

初始条件:稀疏矩阵 M 存在。

操作结果:输出稀疏矩阵 M。

CopySMatrix(M, &T);

初始条件:稀疏矩阵 M 存在。

操作结果:由稀疏矩阵 M 复制得到 T。

AddSMatrix(M, N, &Q);

初始条件:稀疏矩阵 M 与 N 的行数和列数对应相等。

操作结果:求稀疏矩阵的和 Q = M + N。

SubtMatrix(M, N, &Q);

初始条件:稀疏矩阵 M 与 N 的行数和列数对应相等。

操作结果:求稀疏矩阵的差 Q = M - N。

MultSMatrix(M, N, &Q);

初始条件:稀疏矩阵 M 的列数等于 N 的行数。

操作结果:求稀疏矩阵乘积 Q = M × N。

TransposeSMatrix(M, &T);

初始条件:稀疏矩阵 M 存在。

操作结果:求稀疏矩阵 M 的转置矩阵 T。

}**ADT** SparseMatrix

如何进行稀疏矩阵的压缩存储呢?

按照压缩存储的概念,只存储稀疏矩阵的非零元。因此,除了存储非零元的值之外,还必须同时记下它所在行和列的位置(i,j)。反之,一个三元组(i,j,a_{ij})惟一确定了矩阵 A 的一个非零元。由此,稀疏矩阵可由表示非零元的三元组及其行列数惟一确定。例如,下列三元组表

((1,2,12),(1,3,9),(3,1,−3),(3,6,14),(4,3,24),(5,2,18),(6,1,15),(6,4,−7))

加上(6,7)这一对行、列值便可作为图 5.5 中矩阵 M 的另一种描述。而由上述三元组表的不同表示方法可引出稀疏矩阵不同的压缩存储方法。

$$M=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix} \quad T=\begin{bmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 5.5 稀疏矩阵 M 和 T

www.TopSage.com

1. 三元组顺序表

假设以顺序存储结构来表示三元组表，则可得稀疏矩阵的一种压缩存储方式——我们称之为三元组顺序表。

```
//-- -- - 稀疏矩阵的三元组顺序表存储表示 -- -- -
#define  MAXSIZE  12500      // 假设非零元个数的最大值为 12500
typedef struct {
    int          i, j;          // 该非零元的行下标和列下标
    ElemType     e;
}Triple;
typedef struct {
    Triple  data[MAXSIZE + 1]; // 非零元三元组表,data[0]未用
    int       mu, nu, tu;       // 矩阵的行数、列数和非零元个数
}TSMatrix;
```

在此，data 域中表示非零元的三元组是以行序为主序顺序排列的，从下面的讨论中读者容易看出这样做将有利于进行某些矩阵运算。下面将讨论在这种压缩存储结构下如何实现矩阵的转置运算。

转置运算是一种最简单的矩阵运算。对于一个 $m\times n$ 的矩阵 M，它的转置矩阵 T 是一个 $n\times m$ 的矩阵，且 $T(i,j)=M(j,i)$，$1\leqslant i\leqslant n$，$1\leqslant j\leqslant m$。例如，图 5.5 中的矩阵 M 和 T 互为转置矩阵。

显然，一个稀疏矩阵的转置矩阵仍然是稀疏矩阵。假设 a 和 b 是 TSMatrix 型的变量，分别表示矩阵 M 和 T。那么，如何由 a 得到 b 呢？

从分析 a 和 b 之间的差异可见只要做到：(1)将矩阵的行列值相互交换；(2)将每个三元组中的 i 和 j 相互调换；(3)重排三元组之间的次序便可实现矩阵的转置。前二条是容易做到的，关键是如何实现第三条。即如何使 b. data 中的三元组是以 T 的行(M 的列)为主序依次排列的。

i	j	v
1	2	12
1	3	9
3	1	−3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	−7

a. data

i	j	v
1	3	−3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	−7
6	3	14

b. data

可以有两种处理方法：

(1) 按照 b. data 中三元组的次序依次在 a. data 中找到相应的三元组进行转置。换句话说，按照矩阵 M 的列序来进行转置。为了找到 M 的每一列中所有的非零元素，需要对其三元组表 a. data 从第一行起整个扫描一遍，由于 a. data 是以 M 的行序为主序来存放每个非零元的，由此得到的恰是 b. data 应有的顺序。其具体算法描述如算法 5.1 所示。

www.TopSage.com

大家网
TopSage.com

```
Status TransposeSMatrix(TSMatrix M, TSMatrix &T) {
    // 采用三元组表存储表示,求稀疏矩阵 M 的转置矩阵 T。
    T.mu = M.nu;  T.nu = M.mu;  T.tu = M.tu;
    if (T.tu) {
      q = 1;
      for (col = 1; col<= M.nu; ++col)
          for (p = 1; p<= M.tu; ++p)
            if (M.data[p].j == col) {
                T.data[q].i = M.data[p].j;  T.data[q].j = M.data[p].i;
                T.data[q].e = M.data[p].e;  ++q; }
    }
    return OK;
}// TransposeSMatrix
```

算法 5.1

分析这个算法,主要的工作是在 p 和 col 的两重循环中完成的,故算法的时间复杂度为 $O(nu \cdot tu)$[①],即和 M 的列数及非零元的个数的乘积成正比。我们知道,一般矩阵的转置算法为

```
for (col=1; col<=nu; ++col)
    for (row=1; row<=mu; ++row)
      T[col][row]=M[row][col];
```

其时间复杂度为 $O(mu \times nu)$。当非零元的个数 tu 和 $mu \times nu$ 同数量级时,算法 5.1 的时间复杂度就为 $O(mu \times nu^2)$了(例如,假设在 100×500 的矩阵中有 tu=10 000 个非零元),虽然节省了存储空间,但时间复杂度提高了,因此算法 5.1 仅适于 $tu \ll mu \times nu$ 的情况。

(2) 按照 a.data 中三元组的次序进行转置,并将转置后的三元组置入 b 中恰当的位置。如果能预先确定矩阵 M 中每一列(即 T 中每一行)的第一个非零元在 b.data 中应有的位置,那么在对 a.data 中的三元组依次作转置时,便可直接放到 b.data 中恰当的位置上去。为了确定这些位置,在转置前,应先求得 M 的每一列中非零元的个数,进而求得每一列的第一个非零元在 b.data 中应有的位置。

在此,需要附设 num 和 cpot 两个向量。num[col]表示矩阵 M 中第 col 列中非零元的个数,cpot[col]指示 M 中第 col 列的第一个非零元在 b.data 中的恰当位置。显然有

$$\begin{cases} \text{cpot}[1] = 1; \\ \text{cpot}[\text{col}] = \text{cpot}[\text{col}-1] + \text{num}[\text{col}-1] \quad 2 \leqslant \text{col} \leqslant \text{a.nu} \end{cases} \tag{5-4}$$

例如,对图 5.5 的矩阵 M,num 和 cpot 的值如表 5.1 所示。

表 5.1 矩阵 M 的向量 cpot 的值

col	1	2	3	4	5	6	7
num[col]	2	2	2	1	0	1	0
cpot[col]	1	3	5	7	8	8	9

① 在此,我们将 M.nu 和 M.tu 简写成 nu 和 tu,以下同。

www.TopSage.com

这种转置方法称为快速转置，其算法如算法 5.2 所示。

```
Status FastTransposeSMatrix(TSMatrix M, TSMatrix &T) {
  // 采用三元组顺序表存储表示，求稀疏矩阵 M 的转置矩阵 T。
  T.mu = M.nu;  T.nu = M.mu;  T.tu = M.tu;
  if (T.tu) {
    for (col = 1; col<= M.nu; ++col)  num[col] = 0;
    for (t = 1; t<= M.tu; ++t)  ++num[M.data[t].j];  // 求 M 中每一列含非零元个数
    cpot[1] = 1;
    // 求第 col 列中第一个非零元在 b.data 中的序号
    for (col = 2; col<= M.nu; ++col)  cpot[col] = cpot[col - 1] + num[col - 1];
    for (p = 1; p<= M.tu; ++p) {
      col = M.data[p].j;    q = cpot[col];
      T.data[q].i = M.data[p].j;  T.data[q].j = M.data[p].i;
      T.data[q].e = M.data[p].e;  ++cpot[col];
    }// for
  }//if
  return OK;
}// FastTransposeSMatrix
```

算法 5.2

这个算法仅比前一个算法多用了两个辅助向量[①]。从时间上看，算法中有 4 个并列的单循环，循环次数分别为 nu 和 tu，因而总的时间复杂度为 $O(nu+tu)$。在 M 的非零元个数 tu 和 mu×nu 等数量级时，其时间复杂度为 $O(mu\times nu)$，和经典算法的时间复杂度相同。

三元组顺序表又称有序的双下标法，它的特点是，非零元在表中按行序有序存储，因此便于进行依行顺序处理的矩阵运算。然而，若需按行号存取某一行的非零元，则需从头开始进行查找。

2. 行逻辑链接的顺序表

为了便于随机存取任意一行的非零元，则需知道每一行的第一个非零元在三元组表中的位置。为此，可将上节快速转置矩阵的算法中创建的，指示“行”信息的辅助数组 cpot 固定在稀疏矩阵的存储结构中。称这种“带行链接信息”的三元组表为行逻辑链接的顺序表，其类型描述如下：

```
typedef struct {
  Triple  data[MAXSIZE + 1];      // 非零元三元组表
  int     rpos[MAXRC + 1];        // 各行第一个非零元的位置表
  int     mu, nu, tu;             // 矩阵的行数、列数和非零元个数
}RLSMatrix;
```

在下面讨论的两个稀疏矩阵相乘的例子中，容易看出这种表示方法的优越性。

---

① 只要将计算 cpot 的算法稍稍改动一下，也可以只占一个向量空间。

www.TopSage.com

大家网 TopSage.com

两个矩阵相乘的经典算法也是大家所熟悉的。若设

$$Q = M \times N$$

其中，M 是 $m_1 \times n_1$ 矩阵，N 是 $m_2 \times n_2$ 矩阵。当 $n_1 = m_2$ 时有：

```
for (i = 1; i<= m1; ++ i)
    for (j = 1; j<= n2; ++ j) {
        Q[i][j] = 0;
        for (k = 1; k<= n1; ++ k) Q[i][j] += M[i][k] * N[k][j];
    }
```

此算法的时间复杂度是 $O(m_1 \times n_1 \times n_2)$。

当 M 和 N 是稀疏矩阵并用三元组表作存储结构时，就不能套用上述算法。假设 M 和 N 分别为

$$M = \begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix} \qquad N = \begin{pmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{pmatrix} \tag{5-5}$$

则 $Q = M \times N$ 为

$$Q = \begin{pmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{pmatrix}$$

它们的三元组 M. data、N. data 和 Q. data 分别为：

i	j	e
1	1	3
1	4	5
2	2	−1
3	1	2

M. data

i	j	e
1	2	2
2	1	1
3	1	−2
3	2	4

N. data

i	j	e
1	2	6
2	1	−1
3	2	4

Q. data

那么如何从 M 和 N 求得 Q 呢？

(1) 乘积矩阵 Q 中元素

$$Q(i,j) = \sum_{k=1}^{n_1} M(i,k) \times N(k,j) \qquad \begin{matrix} 1 \leqslant i \leqslant m_1 \\ 1 \leqslant j \leqslant n_2 \end{matrix} \tag{5-6}$$

在经典算法中，不论 $M(i,k)$ 和 $N(k,j)$ 的值是否为零，都要进行一次乘法运算，而实际上，这两者有一个值为零时，其乘积也为零。因此，在对稀疏矩阵进行运算时，应免去这种无效操作，换句话说，为求 Q 的值，只需在 M. data 和 N. data 中找到相应的各对元素(即 M. data 中的 j 值和 N. data 中的 i 值相等的各对元素)相乘即可。

例如，M. data[1]表示的矩阵元(1,1,3)只要和 N. data[1]表示的矩阵元(1,2,2)相乘；而 M. data[2]表示的矩阵元(1,4,5)则不需和 N 中任何元素相乘，因为 N. data 中没有 i 为 4 的元素。由此可见，为了得到非零的乘积，只要对 M. data[1..M. tu]中的每个元

www.TopSage.com

素$(i,k,M(i,k))$ $(1\leqslant i\leqslant m_1, 1\leqslant k\leqslant n_1)$，找到 N. data 中所有相应的元素$(k,j,N(k,j))$ $(1\leqslant k\leqslant m_2, 1\leqslant j\leqslant n_2)$相乘即可，为此需在 N. data 中寻找矩阵 N 中第 k 行的所有非零元。在稀疏矩阵的行逻辑链接的顺序表中，N. rpos 为我们提供了有关信息。例如，式(5-5)中的矩阵 N 的 rpos 值如表 5.2 所示。

表 5.2　矩阵 N 的 rpos 值

row	1	2	3	4
rpos[row]	1	2	3	5

并且，由于 rpos[row]指示矩阵 N 的第 row 行中第一个非零元在 N. data 中的序号，则 rpos[row+1]−1 指示矩阵 N 的第 *row* 行中最后一个非零元在 N. data 中的序号。而最后一行中最后一个非零元在 N. data 中的位置显然就是 N. tu 了。

(2) 稀疏矩阵相乘的基本操作是：对于 M 中每个元素 M. data[p](p=1,2,…,M. tu)，找到 N 中所有满足条件 M. data[p]. j=N. data[q]. i 的元素 N. data[q]，求得 M. data[p]. v 和 N. data[q]. v 的乘积，而从式(5-6)得知，乘积矩阵 Q 中每个元素的值是个累计和，这个乘积 M. data[p]. v×N. data[q]. v 只是 Q[i][j]中的一部分。为便于操作，应对每个元素设一累计和的变量，其初值为零，然后扫描数组 M，求得相应元素的乘积并累加到适当的求累计和的变量上。

(3) 两个稀疏矩阵相乘的乘积不一定是稀疏矩阵。反之，即使式(5-6)中每个分量值 $M(i,k)\times N(k,j)$不为零，其累加值 Q[i][j]也可能为零。因此乘积矩阵 Q 中的元素是否为非零元，只有在求得其累加和后才能得知。由于 Q 中元素的行号和 M 中元素的行号一致，又 M 中元素排列是以 M 的行序为主序的，由此可对 Q 进行逐行处理，先求得累计求和的中间结果(Q 的一行)，然后再压缩存储到 Q. data 中去。

由此，两个稀疏矩阵相乘($Q=M\times N$)的过程可大致描述如下：

```
Q初始化;
if (Q是非零矩阵) {  // 逐行求积
    for (arow = 1; arow<= M.mu; ++arow) {     //  处理M的每一行
        ctemp[ ] = 0;              // 累加器清零
        计算Q中第arow行的积并存入ctemp[ ]中;
        将ctemp[ ]中非零元压缩存储到Q.data;
    }// for arow
}// if
```

算法 5.3 是上述过程求精的结果。

```
Status MultSMatrix(RLSMatrix M, RLSMatrix N, RLSMatrix &Q) {
    //求矩阵乘积Q=M×N,采用行逻辑链接存储表示。
    if (M.nu != N.mu) return ERROR;
    Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0;              // Q初始化
    if (M.tu * N.tu != 0) {     // Q是非零矩阵
      for (arow = 1; arow<= M.mu; ++arow) {     // 处理M的每一行
        ctemp[ ] = 0;                        // 当前行各元素累加器清零
```


www.TopSage.com

大家网
TopSage.com

```
        Q.rpos[arow] = Q.tu + 1;
        if(arow<M.mu)tp = M.rpos[arow + 1];
        else{tp = M.tu + 1;}
        for (p = M.rpos[arow]; p<tp; ++p){  // 对当前行中每一个非零元
          brow = M.data[p].j;                // 找到对应元在 N 中的行号
          if (brow < N.mu )  t = N.rpos[brow + 1];
          else  {t = N.tu + 1; }
          for (q = N.rpos[brow];  q< t;  ++q) {
            ccol = N.data[q].j;               // 乘积元素在 Q 中列号
            ctemp[ccol] += M.data[p].e * N.data[q].e;
          }// for q
        }// 求得 Q 中第 crow( =arow)行的非零元
        for (ccol = 1; ccol<= Q.nu; ++ccol)     // 压缩存储该行非零元
          if (ctemp[ccol]) {
            if ( ++Q.tu > MAXSIZE) return ERROR;
            Q.data[Q.tu] = (arow, ccol, ctemp[ccol]);
          }// if
        }// for arow
      }// if
      return OK;
    }// MultSMatrix
```

算法　5.3

分析上述算法的时间复杂度有如下结果：累加器 ctemp 初始化的时间复杂度为 $O(M.mu \times N.nu)$，求 Q 的所有非零元的时间复杂度为 $O(M.tu \times N.tu/N.mu)$，进行压缩存储的时间复杂度为 $O(M.mu \times N.nu)$，因此，总的时间复杂度就是 $O(M.mu \times N.nu + M.tu \times N.tu/N.mu)$。

若 M 是 m 行 n 列的稀疏矩阵，N 是 n 行 p 列的稀疏矩阵，则 M 中非零元的个数 $M.tu = \delta_M \times m \times n$，$N$ 中非零元的个数 $N.tu = \delta_N \times n \times p$，此时算法 5.3 的时间复杂度就是 $O(m \times p \times (1 + n\delta_M\delta_N))$，当 $\delta_M < 0.05$ 和 $\delta_N < 0.05$ 及 $n < 1\,000$ 时，算法 5.3 的时间复杂度就相当于 $O(m \times p)$，显然，这是一个相当理想的结果。

如果事先能估算出所求乘积矩阵 Q 不再是稀疏矩阵，则以二维数组表示 Q，相乘的算法也就更简单了。

3. 十字链表

当矩阵的非零元个数和位置在操作过程中变化较大时，就不宜采用顺序存储结构来表示三元组的线性表。例如，在作“将矩阵 B 加到矩阵 A 上”的操作时，由于非零元的插入或删除将会引起 A.data 中元素的移动。为此，对这种类型的矩阵，采用链式存储结构表示三元组的线性表更为恰当。

在链表中，每个非零元可用一个含 5 个域的结点表示，其中 i、j 和 e 这 3 个域分别表示该非零元所在的行、列和非零元的值，向右域 right 用以链接同一行中下一个非零元，向下域 down 用以链接同一列中下一个非零元。同一行的非零元通过 right 域链接成一个线性链表，同一列的非零元通过 down 域链接成一个线性链表，每个非零元既是某个行链表中的一个结点，又是某个列链表中的一个结点，整个矩阵构成了一个十字交叉的链表，故称这样的存储结构为十字链表，可用两个分别存储行链表的头指针和列链表的头指针

www.TopSage.com

的一维数组表示之。例如，式（5-5）中的矩阵 M 的十字链表如图 5.6 所示。

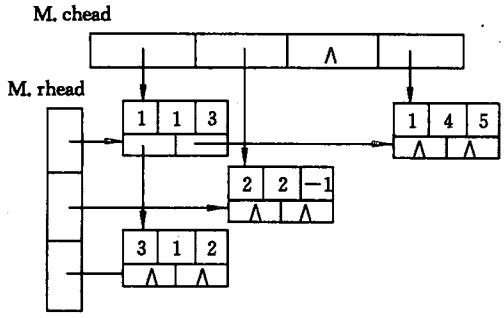


图 5.6　稀疏矩阵 M 的十字链表

算法 5.4 是稀疏矩阵的十字链表表示和建立十字链表的算法。

```
        // ------ 稀疏矩阵的十字链表存储表示 ------
typedef struct OLNode {
    int             i, j;           // 该非零元的行和列下标
    ElemType        e;
    struct OLNode * right, * down;  // 该非零元所在行表和列表的后继链域
}OLNode;   * OLink;

typedef struct {
    OLink * rhead, * chead;         // 行和列链表头指针向量基址由 CreateSMatrix 分配.
    int     mu, nu, tu;             // 稀疏矩阵的行数、列数和非零元个数
}CrossList;

Status CreateSMatrix_OL (CrossList &M) {
  // 创建稀疏矩阵 M。采用十字链表存储表示。
  if (M) free(M);
  scanf(&m, &n, &t );     // 输入 M 的行数、列数和非零元个数
  M.mu := m;  M.nu := n;  M.tu := t;
  if (!(M.rhead = (OLink * )malloc((m + 1) * sizeof(OLink))))exit(OVERFLOW);
  if (!(M.chead = (OLink * )malloc((n + 1) * sizeof(OLink)))) exit(OVERFLOW);
  M.rhead [ ] = M.chead [ ] = NULL ;     // 初始化行列头指针向量;各行列链表为空链表
  for (scanf(&i, &j, &e); i!= 0; scanf(&i, &j, &e)) {// 按任意次序输入非零元
    if (!(p = (OLNode * )malloc(sizeof(OLNode)))) exit(OVERFLOW);
    p->i = i;  p->j = j;  p->e = e;                    // 生成结点
    if (M.rhead[i] == NULL || M.rhead [i]->j>j)  {p->right = M.rhead [i];M.rhead [i] = p;}
    else {       // 寻查在行表中的插入位置
      for ( q = M.rhead [i]; (q->right) && q->right->j < j;  q = q->right );
      p->right = q->right;  q ->right = p;          } // 完成行插入
    if (M.chead [j] == NULL || M.chead[j]->i>i)  {p->down = M.chead [j]; M.chead [j] = p;}
    else {       // 寻查在列表中的插入位置
      for ( q = M.chead[j]; (q->down) && q->down->i < i;  q = q->down );
      p->down = q->down;  q->down = p;       }     // 完成列插入
  }
```


www.TopSage.com

```
    return OK;
}// CreateSMatrix_OL
```

大家网 TopSage.com

算法 5.4

对于 m 行 n 列且有 t 个非零元的稀疏矩阵，算法 5.4 的执行时间为 $O(t\times s)$，$s=\max\{m,n\}$，这是因为每建立一个非零元的结点时都要寻查它在行表和列表中的插入位置，此算法对非零元输入的先后次序没任何要求。反之，若按以行序为主序的次序依次输入三元组，则可将建立十字链表的算法改写成 $O(t)$ 数量级的（t 为非零元的个数）。

下面我们讨论在十字链表表示稀疏矩阵时，如何实现“将矩阵 B 加到矩阵 A 上”的运算。

两个矩阵相加和第 2 章中讨论的两个一元多项式相加极为相似，所不同的是一元多项式中只有一个变元（即指数项），而矩阵中每个非零元有两个变元（行值和列值），每个结点既在行表中又在列表中，致使插入和删除时指针的修改稍为复杂，故需更多的辅助指针。

假设两个矩阵相加后的结果为 A'，则和矩阵 A' 中的非零元 a_{ij}' 只可能有 3 种情况。它或者是 $a_{ij}+b_{ij}$；或者是 a_{ij}（$b_{ij}=0$ 时）；或者是 b_{ij}（$a_{ij}=0$ 时）。由此，当将 B 加到 A 上去时，对 A 矩阵的十字链表来说，或者是改变结点的 val 域值（$a_{ij}+b_{ij}\neq 0$），或者不变（$b_{ij}=0$），或者插入一个新结点（$a_{ij}=0$）。还有一种可能的情况是：和 A 矩阵中的某个非零元相对应，和矩阵 A' 中是零元，即对 A 的操作是删除一个结点（$a_{ij}+b_{ij}=0$）。由此，整个运算过程可从矩阵的第一行起逐行进行。对每一行都从行表头出发分别找到 A 和 B 在该行中的第一个非零元结点后开始比较，然后按上述 4 种不同情况分别处理之。

假设非空指针 pa 和 pb 分别指向矩阵 A 和 B 中行值相同的两个结点，$pa==$**NULL** 表明矩阵 A 在该行中没有非零元，则上述 4 种情况的处理过程为：

(1) 若 pa==**NULL** 或 pa->j>pb->j，则需要在 A 矩阵的链表中插入一个值为 b_{ij} 的结点。此时，需改变同一行中前一结点的 right 域值，以及同一列中前一结点的 down 域值。

(2) 若 pa->j<pb->j，则只要将 pa 指针往右推进一步。

(3) 若 pa->j==pb->j 且 pa->e+pb->e!=0，则只要将 $a_{ij}+b_{ij}$ 的值送到 pa 所指结点的 e 域即可，其他所有域的值都不变。

(4) 若 pa->j==pb->j 且 pa->e+pb->e==0，则需要在 A 矩阵的链表中删除 pa 所指的结点。此时，需改变同一行中前一结点的 right 域值，以及同一列中前一结点的 down 域值。

为了便于插入和删除结点，还需要设立一些辅助指针。其一是，在 A 的行链表上设 pre 指针，指示 pa 所指结点的前驱结点；其二是，在 A 的每一列的链表上设一个指针 hl[j]，它的初值和列链表的头指针相同，即 hl[j]=chead[j]。

下面对将矩阵 B 加到矩阵 A 上的操作过程作一个概要的描述。

(1) 初始，令 pa 和 pb 分别指向 A 和 B 的第一行的第一个非零元素的结点，即

```
pa=A.rhead[1];  pb=B.rhead[1];  pre=NULL;
```

且令 hl 初始化

```
for (j=1; j<=A.nu; ++j)  hl[j]=A.chead[j];
```


www.TopSage.com

(2) 重复本步骤，依次处理本行结点，直到 B 的本行中无非零元素的结点，即 pb==NULL 为止：

① 若 pa==NULL 或 pa->j>pb->j(即 A 的这一行中非零元素已处理完)，则需在 A 中插入一个 pb 所指结点的复制结点。假设新结点的地址为 p，则 A 的行表中的指针作如下变化：

```
if (pre == NULL)  A.rhead[p->i]=p;
else {pre->right=p; }
p->right=pa;  pre=p;
```

A 的列链表中的指针也要作相应的改变。首先需从 hl[p->j]开始找到新结点在同一列中的前驱结点，并让 hl[p->j]指向它，然后在列链表中插入新结点：

```
if (! A.chead[p->j]||A.chead[p->j]->i>p->i)
{p->down=A.chead[p->j]; A.chead[p->j]=p;}
 else {p->down=hl[p->j]->down;  hl[p->j]->down=p; }
 hl[p->j]=p;
```

② 若 pa!=NULL 且 pa->j<pb->j，则令 pa 指向本行下一个非零元结点，即

```
pre=pa;  pa=pa->right;
```

③ 若 pa->j == pb->j，则将 B 中当前结点的值加到 A 中当前结点上，即

```
pa->e+=pb->e;
```

此时若 pa->e!=0，则指针不变，否则删除 A 中该结点，即行表中指针变为

```
if (pre == NULL)  A.rhead[pa->i]=pa->right;
else {pre->right=pa->right; }
p=pa;  pa=pa->right;
```

同时，为了改变列表中的指针，需要先找到同一列中的前驱结点，且让 hl[pa->j]指向该结点，然后如下修改相应指针：

```
if (A.chead[p->j] == p)  A.chead[p->j]=hl[p->j]=p->down;
else {hl[p->j]->down=p->down; }
free (p);
```

(3) 若本行不是最后一行，则令 pa 和 pb 指向下一行的第一个非零元结点，转(2)；否则结束。

通过对这个算法的分析可以得出下述结论：从一个结点来看，进行比较、修改指针所需的时间是一个常数；整个运算过程在于对 A 和 B 的十字链表逐行扫描，其循环次数主要取决于 A 和 B 矩阵中非零元素的个数 ta 和 tb；由此算法的时间复杂度为 $O(ta+tb)$。

5.4 广义表的定义

顾名思义，广义表是线性表的推广。也有人称其为列表(lists，用复数形式以示与统称的表 list 的区别)。广泛地用于人工智能等领域的表处理语言 LISP 语言，把广义表作

www.TopSage.com

大家网
TopSage.com

为基本的数据结构，就连程序也表示为一系列的广义表。

抽象数据类型广义表的定义如下：

```
ADT GList {
  数据对象:D={e_i | i=1,2,…,n;  n≥0;  e_i∈AtomSet 或 e_i∈GList,
                AtomSet 为某个数据对象 }
  数据关系:R1={<e_{i-1}, e_i> | e_{i-1},e_i∈D,  2≤i≤n }
  基本操作:
    InitGList(&L);
      操作结果:创建空的广义表 L。
    CreateGList(&L, S);
      初始条件:S 是广义表的书写形式串。
      操作结果:由 S 创建广义表 L。
    DestroyGList(&L);
      初始条件:广义表 L 存在。
      操作结果:销毁广义表 L。
    CopyGList(&T, L);
      初始条件:广义表 L 存在。
      操作结果:由广义表 L 复制得到广义表 T。
    GListLength(L);
      初始条件:广义表 L 存在。
      操作结果:求广义表 L 的长度,即元素个数。
    GListDepth(L);
      初始条件:广义表 L 存在。
      操作结果:求广义表 L 的深度。
    GListEmpty(L);
      初始条件:广义表 L 存在。
      操作结果:判定广义表 L 是否为空。
    GetHead(L);
      初始条件:广义表 L 存在。
      操作结果:取广义表 L 的头。
    GetTail(L);
      初始条件:广义表 L 存在。
      操作结果:取广义表 L 的尾。
    InsertFirst_GL(&L, e);
      初始条件:广义表 L 存在。
      操作结果:插入元素 e 作为广义表 L 的第一元素。
    DeleteFirst_GL(&L, &e);
      初始条件:广义表 L 存在。
      操作结果:删除广义表 L 的第一元素,并用 e 返回其值。
    Traverse_GL(L, Visit());
      初始条件:广义表 L 存在。
```


www.TopSage.com

操作结果：遍历广义表 L，用函数 Visit 处理每个元素。

}**ADT** GList

广义表一般记作

$$LS = (\alpha_1, \alpha_2, \cdots, \alpha_n)$$

其中，LS 是广义表$(\alpha_1, \alpha_2, \cdots, \alpha_n)$的名称，$n$ 是它的长度。在线性表的定义中，$a_i (1 \leqslant i \leqslant n)$只限于是单个元素。而在广义表的定义中，$\alpha_i$ 可以是单个元素，也可以是广义表，分别称为广义表 LS 的**原子**和**子表**。习惯上，用大写字母表示广义表的名称，用小写字母表示原子。当广义表 LS 非空时，称第一个元素 α_1 为 LS 的**表头**(Head)，称其余元素组成的表$(\alpha_2, \alpha_3, \cdots, \alpha_n)$是 LS 的表尾(Tail)。

显然，广义表的定义是一个递归的定义，因为在描述广义表时又用到了广义表的概念。下面列举一些广义表的例子。

(1) $A=(\ )$——A 是一个空表，它的长度为零。

(2) $B=(e)$——列表 B 只有一个原子 e，B 的长度为 1。

(3) $C=(a,(b,c,d))$——列表 C 的长度为 2，两个元素分别为原子 a 和子表(b,c,d)。

(4) $D=(A,B,C)$——列表 D 的长度为 3，3 个元素都是列表。显然，将子表的值代入后，则有 $D=((\ ),(e),(a,(b,c,d)))$。

(5) $E=(a,E)$——这是一个递归的表，它的长度为 2。E 相当于一个无限的列表 $E=(a,(a,(a,\cdots)))$。

从上述定义和例子可推出列表的 3 个重要结论：

(1) 列表的元素可以是子表，而子表的元素还可以是子表……由此，列表是一个多层次的结构，可以用图形象地表示。例如图 5.7 表示的是列表 D。图中以圆圈表示列表，以方块表示原子。

(2) 列表可为其他列表所共享。例如在上述例子中，列表 A、B 和 C 为 D 的子表，则在 D 中可以不必列出子表的值，而是通过子表的名称来引用。

(3) 列表可以是一个递归的表，即列表也可以是其本身的一个子表。例如列表 E 就是一个递归的表。

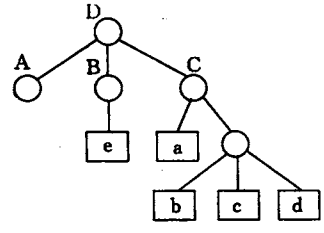


图 5.7　列表的图形表示

根据前述对表头、表尾的定义可知：任何一个非空列表其表头可能是原子，也可能是列表，而其表尾必定为列表。例如：

GetHead(B)=e,　　GetTail(B)=( ),

GetHead(D)=A,　　GetTail(D)=(B,C),

由于(B,C)为非空列表，则可继续分解得到：

GetHead((B,C))=B,　　GetTail((B,C))=(C),

值得提醒的是列表( )和(( ))不同。前者为空表，长度 $n=0$；后者长度 $n=1$，可分解得到其表头、表尾均为空表( )。

www.TopSage.com

大家网
TopSage.com

5.5 广义表的存储结构

由于广义表$(\alpha_1,\alpha_2,\cdots,\alpha_n)$中的数据元素可以具有不同的结构(或是原子,或是列表),因此难以用顺序存储结构表示,通常采用链式存储结构,每个数据元素可用一个结点表示。

如何设定结点的结构?由于列表中的数据元素可能为原子或列表,由此需要两种结构的结点:一种是表结点,用以表示列表;一种是原子结点,用以表示原子。从上节得知:若列表不空,则可分解成表头和表尾;反之,一对确定的表头和表尾可惟一确定列表。由此,一个表结点可由 3 个域组成:标志域、指示表头的指针域和指示表尾的指针域;而原子结点只需两个域:标志域和值域(如图 5.8 所示)。其形式定义说明如下:

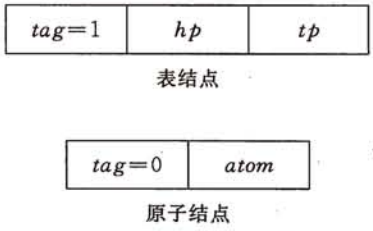


图 5.8 列表的链表结点结构

```
// ------ 广义表的头尾链表存储表示 ------
typedef enum {ATOM, LIST }ElemTag; // ATOM == 0:原子,LIST == 1:子表
typedef struct GLNode {
    ElemTag  tag;              // 公共部分,用于区分原子结点和表结点
    union {                    // 原子结点和表结点的联合部分
      AtomType  atom;          // atom 是原子结点的值域,AtomType 由用户定义
      struct {struct GLNode   * hp, * tp;}ptr;
                               // ptr 是表结点的指针域,ptr.hp 和 ptr.tp 分别指向表头和表尾
  };
} * GList;                     // 广义表类型
```

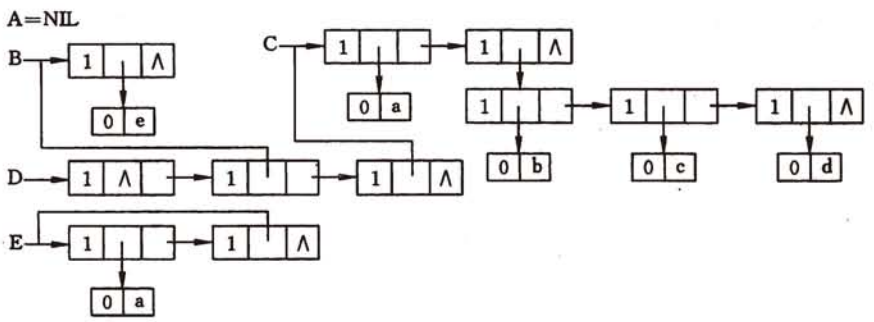


图 5.9 广义表的存储结构示例

上节中曾列举了广义表的例子,它们的存储结构如图 5.9 所示。在这种存储结构中有几种情况:(1)除空表的表头指针为空外,对任何非空列表,其表头指针均指向一个表结点,且该结点中的 hp 域指示列表表头(或为原子结点,或为表结点),tp 域指向列表表尾(除非表尾为空,则指针为空,否则必为表结点);(2)容易分清列表中原子和子表所在层次。如在列表 D 中,原子 a 和 e 在同一层次上,而 b、c 和 d 在同一层次且比 a 和 e 低一

www.TopSage.com

层，B 和 C 是同一层的子表；(3)最高层的表结点个数即为列表的长度。以上 3 个特点在某种程度上给列表的操作带来方便。也可采用另一种结点结构的链表表示列表，如图 5.10 和图 5.11 所示。其形式定义说明如下：

```
// ------ 广义表的扩展线性链表存储表示 ------
typedef enum {ATOM, LIST }ElemTag; // ATOM == 0:原子,LIST == 1:子表
typedef struct GLNode {
    ElemTag          tag;     // 公共部分，用于区分原子结点和表结点
    union {                   // 原子结点和表结点的联合部分
        AtomType     atom;    // 原子结点的值域
        struct GLNode  * hp;  // 表结点的表头指针
    };
    struct GLNode    * tp;    // 相当于线性链表的 next,指向下一个元素结点
} * GList;                    // 广义表类型 GList 是一种扩展的线性链表
```

对于列表的这两种存储结构，读者只要根据自己的习惯掌握其中一种结构即可。



图 5.10　列表的另一种结点结构

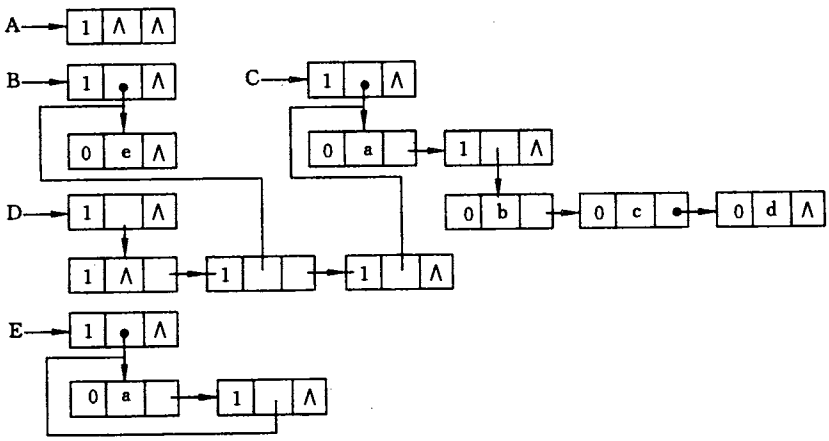


图 5.11　列表的另一种链表表示

5.6　m 元多项式的表示

在一般情况下使用的广义表多数既非是递归表，也不为其他表所共享。对广义表可以这样来理解，广义表中的一个数据元素可以是另一个广义表，一个 m 元多项式的表示就是广义表的这种应用的典型实例。

在第 2 章中，我们曾作为线性表的应用实例讨论了一元多项式，一个一元多项式可以用一个长度为 m 且每个数据元素有两个数据项(系数项和指数项)的线性表来表示。

www.TopSage.com

大家网 TopSage.com

这里，将讨论如何表示 m 元多项式。一个 m 元多项式的每一项，最多有 m 个变元。如果用线性表来表示，则每个数据元素需要 $m+1$ 个数据项，以存储一个系数值和 m 个指数值。这将产生两个问题：一是无论多项式中各项的变元数是多是少，若都按 m 个变元分配存储空间，则将造成浪费；反之，若按各项实际的变元数分配存储空间，就会造成结点的大小不匀，给操作带来不便。二是对 m 值不同的多项式，线性表中的结点大小也不同，这同样会引起存储管理的不便。因此，由于 m 元多项式中每一项的变化数目的不均匀性和变元信息的重要性，故不适于用线性表表示。例如三元多项式

$$P(x,y,z) = x^{10}y^3z^2 + 2x^6y^3z^2 + 3x^5y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz + 15$$

其中各项的变元数目不尽相同，而 y^3、z^2 等因子又多次出现，如若改写为

$$P(x,y,z) = ((x^{10} + 2x^6)y^3 + 3x^5y^2)z^2 + ((x^4 + 6x^3)y^4 + 2y)z + 15$$

情况就不同了。现在，我们再来看这个多项式 P，它是变元 z 的多项式，即 $Az^2 + Bz + 15z^0$，只是其中 A 和 B 本身又是一个 (x,y) 的二元多项式，15 是 z 的零次项的系数。进一步考察 $A(x,y)$，又可把它看成是 y 的多项式。$Cy^3 + Dy^2$，而其中 C 和 D 为 X 的一元多项式。循此以往，每个多项式都可看作是由一个变量加上若干个系数指数偶对组成。

任何一个 m 元多项式都可如此做：先分解出一个主变元，随后再分解出第二个变元，等等。由此，一个 m 元的多项式首先是它的主变元的多项式，而其系数又是第二变元的多项式，由此可用广义表来表示 m 元多项式。例如上述三元多项式可用式(5-7)的广义表表示，广义表的深度即为变元个数。

$$P=z((A,2),(B,1),(15,0))^{①} \tag{5-7}$$

其中 $A=y((C,3),(D,2))$

$C=x((1,10),(2,6))$

$D=x((3,5))$

$B=y((E,4),(F,1))$

$E=x((1,4),(6,3))$

$F=x((2,0))$

可类似于广义表的第二种存储结构来定义表示 m 元多项式的广义表的存储结构。链表的结点结构为：



表结点　　原子结点

其中 exp 为指数域，coef 为系数域，hp 指向其系数子表，tp 指向同一层的下一结点。其形式定义说明如下：

```
typedef struct MPNode {
  ElemTag          tag;    // 区分原子结点和表结点
  int              exp;    // 指数域
  union {
       float       coef;   // 系数域
```

① 我们在广义表的括弧之前加一个变元，以示各层的变元。

www.TopSage.com

```
        struct MPNode * hp;     // 表结点的表头指针
    };
    struct MPNode       * tp;   // 相当于线性链表的 next,指向下一个元素结点
} * MPList;                     // m 元多项式广义表类型
```

式(5-7)的广义表的存储结构如图 5.12 所示，在每一层上增设一个表头结点并利用 exp 指示该层的变元，可用一维数组存储多项式中所有变元，故 exp 域存储的是该变元在一维数组中的下标。头指针 p 所指表结点中 exp 的值 3 为多项式中变元的个数。可见，这种存储结构可表示任何元的多项式。

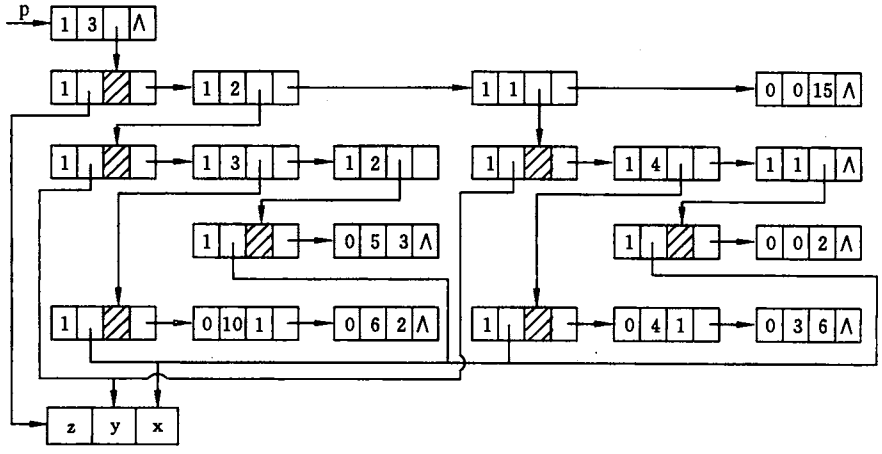


图 5.12　三元多项式 $p(x,y,z)$的存储结构示意图

5.7　广义表的递归算法

在第 3 章中曾提及，递归函数结构清晰、程序易读，且容易证明正确性，因此是程序设计的有力工具，但有时递归函数的执行效率很低，因此使用递归应扬长避短。在程序设计的过程中，我们并不一味追求递归。如果一个问题的求解过程有明显的递推规律，我们也很容易写出它的递推过程(如求阶乘函数 $f(n)=n!$ 的值)，则不必要使用“递归”。反之，在对问题进行分解、求解的过程中得到的是和原问题性质相同的子问题(如 Hanoi 塔问题)，由此自然得到一个递归算法，且它比利用栈实现的非递归算法更符合人们的思维逻辑，因而更易于理解。但是要熟练掌握递归算法的设计方法也不是件轻而易举的事情。在本节中，我们不打算全面讨论如何设计递归算法，只是以广义表为例，讨论如何利用“分治法”(Divide and Conquer)进行递归算法设计的方法。

对这类问题设计递归算法时，通常可以先写出问题求解的递归定义。和第二数学归纳法类似，递归定义由基本项和归纳项两部分组成。

递归定义的基本项描述了一个或几个递归过程的终结状态。虽然一个有限的递归(且无明显的迭代)可以描述一个无限的计算过程，但任何实际应用的递归过程，除错误情况外，必定能经过有限层次的递归而终止。所谓终结状态指的是不需要继续递归而可直

www.TopSage.com

大家网 TopSage.com

接求解的状态。如例 3-3 的 n 阶 Hanoi 塔问题，在 $n=1$ 时可以直接求得解，即将圆盘从 X 塔座移动到 Z 塔座上。一般情况下，若递归参数为 n，则递归的终结状态为 $n=0$ 或 $n=1$ 等。

递归定义的归纳项描述了如何实现从当前状态到终结状态的转化。递归设计的实质是：当一个复杂的问题可以分解成若干子问题来处理时，其中某些子问题与原问题有相同的特征属性，则可利用和原问题相同的分析处理方法；反之，这些子问题解决了，原问题也就迎刃而解了。递归定义的归纳项就是描述这种原问题和子问题之间的转化关系。仍以 Hanoi 塔问题为例。原问题是将 n 个圆盘从 X 塔座移至 Z 塔座上，可以把它分解成 3 个子问题：(1)将编号为 1 至 $n-1$ 的 $n-1$ 个圆盘从 X 塔座移至 Y 塔座；(2)将编号为 n 的圆盘从 X 塔座移至 Z 塔座；(3)将编号为 1 至 $n-1$ 的圆盘从 Y 塔座移至 Z 塔座。其中(1)和(3)的子问题和原问题特征属性相同，只是参数($n-1$ 和 n)不同，由此实现了递归。

由于递归函数的设计用的是归纳思维的方法，则在设计递归函数时，应注意：(1)首先应书写函数的首部和规格说明，严格定义函数的功能和接口(递归调用的界面)，对求精函数中所得的和原问题性质相同的子问题，只要接口一致，便可进行递归调用；(2)对函数中的每一个递归调用都看成只是一个简单的操作，只要接口一致，必能实现规格说明中定义的功能，切忌想得太深太远。正如用第二数学归纳法证明命题时，由归纳假设进行归纳证明时绝不能怀疑归纳假设是否正确。

下面讨论广义表的 3 种操作。首先约定所讨论的广义表都是非递归表且无共享子表。

5.7.1 求广义表的深度

广义表的深度定义为广义表中括弧的重数，是广义表的一种量度。例如，多元多项式广义表的深度为多项式中变元的个数。

设非空广义表为

$$LS=(\alpha_1,\alpha_2,\cdots,\alpha_n)$$

其中 $\alpha_i(i=1,2,\cdots,n)$或为原子或为 LS 的子表，则求 LS 的深度可分解为 n 个子问题，每个子问题为求 α_i 的深度，若 α_i 是原子，则由定义其深度为零，若 α_i 是广义表，则和上述一样处理，而 LS 的深度为各 $\alpha_i(i=1,2,\cdots,n)$的深度中最大值加 1。空表也是广义表，并由定义可知空表的深度为 1。

由此可见，求广义表的深度的递归算法有两个终结状态：空表和原子，且只要求得 $\alpha_i(i=1,2,\cdots,n)$的深度，广义表的深度就容易求得了。显然，它应比子表深度的最大值多 1。

广义表

$$LS=(\alpha_1,\alpha_2,\cdots,\alpha_n)$$

的深度 DEPTH(LS)的递归定义为

基本项：　DEPTH(LS)=1　　当 LS 为空表时

　　DEPTH(LS)=0　　当 LS 为原子时

归纳项：　$\mathrm{DEPTH}(LS)=1+\underset{1\leqslant i\leqslant n}{\mathrm{Max}}\{\mathrm{DEPTH}(\alpha_i)\}$　$n\geqslant 1$

www.TopSage.com

由此定义容易写出求深度的递归函数。假设 L 是 GList 型的变量，则 L=NULL 表明广义表为空表，L->tag=0 表明是原子。反之，L 指向表结点，该结点中的 hp 指针指向表头，即为 L 的第一个子表，而结点中的 tp 指针所指表尾结点中的 hp 指针指向 L 的第二个子表。在第一层中由 tp 相连的所有尾结点中的 hp 指针均指向 L 的子表。由此，求广义表深度的递归函数如算法 5.5 所示。

```
int GListDepth(GList L) {
    // 采用头尾链表存储结构，求广义表 L 的深度。
    if (!L) return 1;                    // 空表深度为 1
    if (L->tag == ATOM) return 0;        // 原子深度为 0
    for (max = 0, pp = L; pp; pp = pp->ptr.tp) {
        dep = GListDepth(pp->ptr.hp); // 求以 pp->ptr.hp 为头指针的子表深度
        if (dep > max) max = dep;
    }
    return max + 1;                      // 非空表的深度是各元素的深度的最大值加 1
}// GListDepth
```

算法 5.5

上述算法的执行过程实质上是遍历广义表的过程，在遍历中首先求得各子表的深度，然后综合得到广义表的深度。例如，图 5.13 展示了求广义表 D 的深度的过程。图中用虚线示意遍历过程中指针 L 的变化状况，在指向结点的虚线旁标记的是将要遍历的子

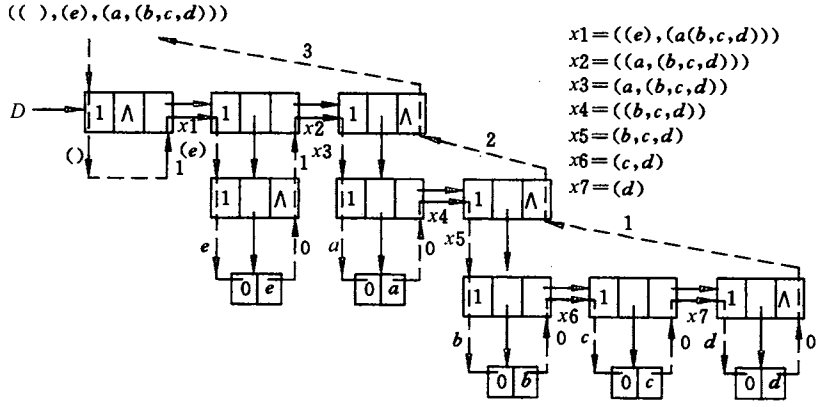


图 5.13 求广义表 D 的深度的过程

表，而在从结点射出的虚线旁标记的数字是刚求得的子表的深度，从图中可见广义表 $D=(A,B,C)=(( ),(e),(a,(b,c,d)))$ 的深度为 3。若按递归定义分析广义表 D 的深度，则有：

```
DEPTH(D) = 1+Max {DEPTH(A),DEPTH(B),DEPTH(C)}
    DEPTH(A) = 1;
    DEPTH(B) = 1+Max {DEPTH(e)} = 1+0=1;
    DEPTH(C) = 1+Max {DEPTH(a),DEPTH((b,c,d))} = 2
      DEPTH(a) = 0
```


www.TopSage.com

大家网
TopSage.com

$$\begin{aligned}\text{DEPTH}((b,c,d)) &= 1+\text{Max}\{\text{DEPTH}(a),\text{DEPTH}(b),\text{DEPTH}(c)\}\\ &= 1+0=1\end{aligned}$$

由此，DEPTH(D)=1+Max{1,1,2}=3。

5.7.2 复制广义表

在5.5节中曾提及，任何一个非空广义表均可分解成表头和表尾，反之，一对确定的表头和表尾可惟一确定一个广义表。由此，复制一个广义表只要分别复制其表头和表尾，然后合成即可。假设LS是原表，NEWLS是复制表，则复制操作的递归定义如下。

基本项：InitGList(NEWLS){置空表}，当LS为空表时。

归纳项：COPY(GetHead(LS)->GetHead(NEWLS)) {复制表头}

COPY(GetTail(LS)->GetTail(NEWLS)) {复制表尾}

若原表以图5.9的链表表示，则复制表的操作便是建立相应的链表。只要建立和原表中的结点一一对应的新结点，便可得到复制表的新链表。由此可写出复制广义表的递归算法如算法5.6所示。

```
Status CopyGList(GList &T, GList L) {
    // 采用头尾链表存储结构，由广义表 L 复制得到广义表 T。
    if (!L) T = NULL;  // 复制空表
    else {
        if (!(T = (GList)malloc(sizeof(GLNode)))) exit(OVERFLOW);  // 建表结点
        T->tag = L->tag;
        if (L->tag == ATOM) T->atom = L->atom;  // 复制单原子
        else {CopyGList(T->ptr.hp, L->ptr.hp);
                        // 复制广义表 L->ptr.hp 的一个副本 T->ptr.hp
              CopyGList(T->ptr.tp, L->ptr.tp);
                        // 复制广义表 L->ptr.tp 的一个副本 T->ptr.tp
        }//else
    } //else
    return OK;
}// CopyGList
```

算法 5.6

注意，这里使用了变参，使得这个递归函数简单明了，直截了当地反映出广义表的复制过程，读者可试以广义表C为例循序察看过程，以便得到更深刻的了解。

5.7.3 建立广义表的存储结构

从上述两种广义表操作的递归算法的讨论中可以发现：在对广义表进行的操作下递归定义时，可有两种分析方法。一种是把广义表分解成表头和表尾两部分；另一种是把广义表看成是含有n个并列子表(假设原子也视作子表)的表。在讨论建立广义表的存储结构时，这两种分析方法均可。

假设把广义表的书写形式看成是一个字符串S，则当S为非空白串时广义表非空。此时可以利用5.4节中定义的取列表表头GetHead和取列表表尾GetTail两个函数建立

www.TopSage.com

广义表的链表存储结构。这个递归算法和复制的递归算法极为相似，读者可自行试之。下面就第二种分析方法进行讨论。

广义表字符串 S 可能有两种情况：(1) S='( )'(带括弧的空白串)；(2) $S=(\alpha_1,\alpha_2,\cdots,\alpha_n)$，其中 $\alpha_i(i=1,2,\cdots,n)$是 S 的子串。对应于第一种情况 S 的广义表为空表，对应于第二种情况 S 的广义表中含有 n 个子表，每个子表的书写形式即为子串 $\alpha_i(i=1,2,\cdots,n)$。此时可类似于求广义表的深度，分析由 S 建立的广义表和由 $\alpha_i(i=1,2,\cdots,n)$建立的子表之间的关系。假设按图 5.8 所示结点结构来建立广义表的存储结构，则含有 n 个子表的广义表中有 n 个表结点序列。第 $i(i=1,2,\cdots,n-1)$个表结点中的表尾指针指向第$i+1$个表结点。第 n 个表结点的表尾指针为 NULL，并且，如果把原子也看成是子表的话，则第 i 个表结点的表头指针 hp 指向由 α_i 建立的子表$(i=1,2,\cdots,n)$。由此，由 S 建广义表的问题可转化为由 $\alpha_i(i=1,2,\cdots,n)$建子表的问题。又，α_i 可能有 3 种情况：(1)带括弧的空白串；(2)长度为 1 的单字符串；(3)长度>1 的字符串。显然，前两种情况为递归的终结状态，子表为空表或只含一个原子结点，后一种情况为递归调用。由此，在不考虑输入字符串可能出错的前提下，可得下列建立广义表链表存储结构的递归定义。

基本项：置空广义表　　当 S 为空表串时

建原子结点的子表　　当 S 为单字符串时

归纳项：假设 sub 为脱去 S 中最外层括弧的子串，记为$'s_1,s_2,\cdots,s_n'$，其中 $s_i(i=1,2,\cdots,n)$为非空字符串。对每一个 s_i 建立一个表结点，并令其 hp 域的指针为由 s_i 建立的子表的头指针，除最后建立的表结点的尾指针为 NULL 外，其余表结点的尾指针均指向在它之后建立的表结点。

假定函数 sever(str,hstr)的功能为，从字符串 str 中取出第一个“,”之前的子串赋给 hstr，并使 str 成为删去子串 hstr 和‘,’之后的剩余串，若串 str 中没有字符‘,’，则操作后的 hstr 即为操作前的 str，而操作后的 str 为空串 NULL。根据上述递归定义可得到建广义表存储结构的递归函数如算法 5.7 所示。函数 sever 如算法 5.8 所示。

```
Status CreateGList(GList &L, SString S) {
  // 采用头尾链表存储结构，由广义表的书写形式串 S 创建广义表 L。设 emp = "()"
    if (StrCompare(S, emp))  L = NULL;  // 创建空表
    else {
      if (!(L = (GList) malloc (sizeof (GLNode)))) exit (OVERFLOW); // 建表结点
      if (StrLength(S) == 1) {L->tag = ATOM;  L->atom = S; }  // 创建单原子广义表
      else {
        L->tag = LIST;  p = L;
        SubString(sub, S, 2, StrLength(S) - 2);                    // 脱外层括号
        do {  // 重复建 n 个子表
          sever(sub, hsub);  // 从 sub 中分离出表头串 hsub
          CreateGList(p->ptr.hp, hsub);  q = p;
          if (!StrEmpty(sub)) {     // 表尾不空
            if (!(p = (GLNode *) malloc (sizeof(GLNode))))
                exit (OVERFLOW);
            p->tag = LIST;  q->ptr.tp = p;
          }// if
```


www.TopSage.com

大家网
TopSage.com

```
        }while (!StrEmpty(sub));
        q->ptr.tp = NULL;
      }else
    }//else
    return OK;
}// CreateGList
```

算法 5.7

```
Status sever(SString &str, SString &hstr) {
    // 将非空串 str 分割成两部分:hsub 为第一个',' 之前的子串,str 为之后的子串
    n = StrLength(str);  i=0;  k=0;  // k 记尚未配对的左括号个数
    do{                                  // 搜索最外层的第一个逗号
     ++i;
        SubString(ch, str, i, 1);
        if (ch == '(')   ++k;
        else if (ch ==')')   --k;
    }while (i<n && (ch! =',' || k! =0));
    if (i<n)
      {SubString(hstr, str, 1, i-1); SubString(str, str, i+1, n-i) }
    else {StrCopy(hstr, str);  ClearString(str) }
}// sever
```

算法 5.8

www.TopSage.com

第6章　树和二叉树

树型结构是一类重要的非线性数据结构。其中以树和二叉树最为常用，直观看来，树是以分支关系定义的层次结构。树结构在客观世界中广泛存在，如人类社会的族谱和各种社会组织机构都可用树来形象表示。树在计算机领域中也得到广泛应用，如在编译程序中，可用树来表示源程序的语法结构。又如在数据库系统中，树形结构也是信息的重要组织形式之一。本章重点讨论二叉树的存储结构及其各种操作，并研究树和森林与二叉树的转换关系，最后介绍几个应用例子。

6.1　树的定义和基本术语

树(Tree)是 $n(n\geqslant 0)$ 个结点的有限集。在任意一棵非空树中：(1) 有且仅有一个特定的称为根(Root)的结点；(2) 当 $n>1$ 时，其余结点可分为 $m(m>0)$ 个互不相交的有限集 $T_1,T_2,\cdots,T_m$，其中每一个集合本身又是一棵树，并且称为根的**子树**(SubTree)。例如，在图 6.1 中，(a) 是只有一个根结点的树；(b)是有 13 个结点的树，其中 A 是根，其余结点分成 3 个互不相交的子集：$T_1=\{B,E,F,K,L\}$，$T_2=\{C,G\}$，$T_3=\{D,H,I,J,M\}$；T_1、T_2 和 T_3 都是根 A 的子树，且本身也是一棵树。例如 T_1，其根为 B，其余结点分为两个互不相交的子集：$T_{11}=\{E,K,L\}$，$T_{12}=\{F\}$。T_{11} 和 T_{12} 都是 B 的子树。而 T_{11} 中 E 是根，$\{K\}$ 和 $\{L\}$ 是 E 的两棵互不相交的子树，其本身又是只有一个根结点的树。

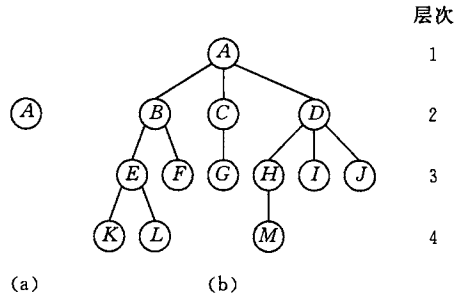


图 6.1　树的示例

(a) 只有根结点的树；(b) 一般的树

上述树的结构定义加上树的一组基本操作就构成了抽象数据类型树的定义。

```
ADT Tree {
  数据对象 D:D 是具有相同特性的数据元素的集合。
  数据关系 R:若 D 为空集,则称为空树;
      若 D 仅含一个数据元素,则 R 为空集,否则 R={H},H 是如下二元关系:
      (1) 在 D 中存在惟一的称为根的数据元素 root,它在关系 H 下无前驱;
      (2) 若 D-{root}≠Φ,则存在 D-{root}的一个划分 D1,D2,…,Dm(m>0),对任意 j≠k(1≤
          j,k≤m)有 Dj ∩ Dk = Φ,且对任意的 i(1≤i≤m),惟一存在数据元素 xi ∈ Di,
          有<root,xi>∈H;
      (3) 对应于 D-{root}的划分,H-{<root,x1>,…,<root,xm>}有惟一的一个划分 H1,
          H2,…,Hm(m>0),对任意 j≠k(1≤j,k≤m)有 Hj ∩ Hk = Φ,且对任意 i(1≤i≤m),Hi 是
          Di 上的二元关系,(Di,{Hi})是一棵符合本定义的树,称为根 root 的子树。
```


www.TopSage.com

大家网
TopSage.com

基本操作 P：

InitTree(&T)；

操作结果：构造空树 T。

DestroyTree(&T)；

初始条件：树 T 存在。

操作结果：销毁树 T。

CreateTree(&T, definition)；

初始条件：definition 给出树 T 的定义。

操作结果：按 definition 构造树 T。

ClearTree(&T)；

初始条件：树 T 存在。

操作结果：将树 T 清为空树。

TreeEmpty(T)；

初始条件：树 T 存在。

操作结果：若 T 为空树，则返回 TRUE，否则 FALSE。

TreeDepth(T)；

初始条件：树 T 存在。

操作结果：返回 T 的深度。

Root(T)；

初始条件：树 T 存在。

操作结果：返回 T 的根。

Value(T, cur_e)；

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：返回 cur_e 的值。

Assign(T, cur_e, value)；

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：结点 cur_e 赋值为 value。

Parent(T, cur_e)；

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 是 T 的非根结点，则返回它的双亲，否则函数值为“空”。

LeftChild(T, cur_e)；

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 是 T 的非叶子结点，则返回它的最左孩子，否则返回“空”。

RightSibling(T, cur_e)；

初始条件：树 T 存在，cur_e 是 T 中某个结点。

操作结果：若 cur_e 有右兄弟，则返回它的右兄弟，否则函数值为“空”。

InsertChild(&T, &p, i, c)；

初始条件：树 T 存在，p 指向 T 中某个结点，$1 \leqslant i \leqslant$ p 所指结点的度 + 1，非空树 c 与 T 不相交。

操作结果：插入 c 为 T 中 p 指结点的第 i 棵子树。

DeleteChild(&T, &p, i)；

初始条件：树 T 存在，p 指向 T 中某个结点，$1 \leqslant i \leqslant$ p 指结点的度。

操作结果：删除 T 中 p 所指结点的第 i 棵子树。

TraverseTree(T, Visit())；

初始条件：树 T 存在，Visit 是对结点操作的应用函数。

操作结果：按某种次序对 T 的每个结点调用函数 visit()一次且至多一次。一旦 visit()失败，则操作失败。

www.TopSage.com

}**ADT** Tree

树的结构定义是一个递归的定义，即在树的定义中又用到树的概念，它道出了树的固有特性。树还可有其他的表示形式，如图 6.2 所示为图 6.1(b)中树的各种表示。其中(a)是以嵌套集合(即是一些集合的集体，对于其中任何两个集合，或者不相交，或者一个包含另一个)的形式表示的；(b) 是以广义表的形式表示的，根作为由子树森林组成的表的名字写在表的左边；(c) 用的是凹入表示法(类似书的编目)。表示方法的多样化，正说明了树结构在日常生活中及计算机程序设计中的重要性。一般说来，分等级的分类方案都可用层次结构来表示，也就是说，都可导致一个树结构。

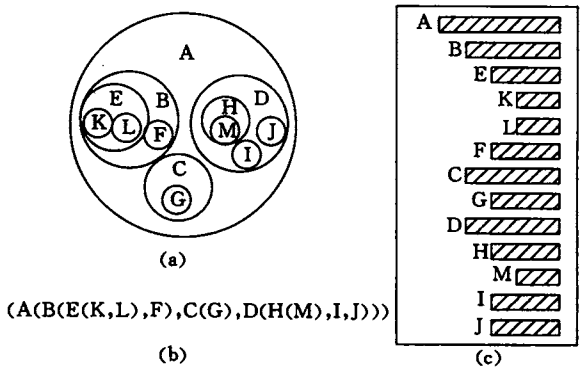


图 6.2　树的其他 3 种表示法

下面列出树结构中的一些基本术语。

树的**结点**包含一个数据元素及若干指向其子树的分支。结点拥有的子树数称为**结点的度**(Degree)。例如，在图 6.1(b)中，A 的度为 3，C 的度为 1，F 的度为 0。度为 0 的结点称为**叶子**(Leaf)或**终端结点**。图 6.1(b)中的结点 K、L、F、G、M、I、J 都是树的叶子。度不为 0 的结点称为**非终端结点**或**分支结点**。除根结点之外，分支结点也称为内部结点。**树的度**是树内各结点的度的最大值。如图 6.1(b)的树的度为 3。结点的子树的根称为该结点的**孩子**(Child)，相应地，该结点称为孩子的**双亲**(Parent)。例如，在图 6.1(b)所示的树中，D 为 A 的子树 T_3 的根，则 D 是 A 的孩子，而 A 则是 D 的双亲，同一个双亲的孩子之间互称**兄弟**(Sibling)。例如，H、I 和 J 互为兄弟。将这些关系进一步推广，可认为 D 是 M 的祖父。结点的**祖先**是从根到该结点所经分支上的所有结点。例如，M 的祖先为 A、D 和 H。反之，以某结点为根的子树中的任一结点都称为该结点的**子孙**。如 B 的子孙为 E、K、L 和 F。

结点的**层次**(Level)从根开始定义起，根为第一层，根的孩子为第二层。若某结点在第 l 层，则其子树的根就在第 $l+1$ 层。其双亲在同一层的结点互为**堂兄弟**。例如，结点 G 与 E、F、H、I、J 互为堂兄弟。树中结点的最大层次称为树的**深度**(Depth)或高度。图 6.1(b)所示的树的深度为 4。

如果将树中结点的各子树看成从左至右是有次序的(即不能互换)，则称该树为**有序树**，否则称为**无序树**。在有序树中最左边的子树的根称为第一个孩子，最右边的称为最后

www.TopSage.com

大家网 TopSage.com

一个孩子。

森林(Forest)是 $m(m\geqslant 0)$棵互不相交的树的集合。对树中每个结点而言,其子树的集合即为森林。由此,也可以森林和树相互递归的定义来描述树。

就逻辑结构而言,任何一棵树是一个二元组 $Tree=(root,F)$,其中:$root$ 是数据元素,称做树的根结点;F 是 $m(m\geqslant 0)$棵树的森林,$F=(T_1,T_2,\cdots,T_m)$,其中 $T_i=(r_i,F_i)$称做根 $root$ 的第 i 棵子树;当 $m\neq 0$ 时,在树根和其子树森林之间存在下列关系:

$$RF=\{<root,r_i>\mid i=1,2,\cdots,m,m>0\}$$

这个定义将有助于得到森林和树与二叉树之间转换的递归定义。

树的应用广泛,在不同的软件系统中树的基本操作集不尽相同。

6.2 二 叉 树

在讨论一般树的存储结构及其操作之前,我们首先研究一种称为二叉树的抽象数据类型。

6.2.1 二叉树的定义

二叉树(Binary Tree)是另一种树型结构,它的特点是每个结点至多只有两棵子树(即二叉树中不存在度大于 2 的结点),并且,二叉树的子树有左右之分,其次序不能任意颠倒。

抽象数据类型二叉树的定义如下:

```
ADT BinaryTree {
 数据对象 D:D 是具有相同特性的数据元素的集合。
 数据关系 R:
  若 D = Φ,则 R = Φ,称 BinaryTree 为空二叉树;
  若 D≠Φ,则 R = {H},H 是如下二元关系:
    (1) 在 D 中存在惟一的称为根的数据元素 root,它在关系 H 下无前驱;
    (2) 若 D - {root}≠Φ,则存在 D - {root} = {D_l, D_r},且 D_l ∩ D_r = Φ;
    (3) 若 D_l ≠Φ,则 D_l 中存在惟一的元素 x_l,<root,x_l>∈H,且存在 D_l 上的
        关系 H_l ⊂H; 若 D_r ≠Φ,则 D_r 中存在惟一的元素 x_r, <root,x_r>∈H,
        且存在 D_r 上的关系 H_r ⊂H; H = {<root,x_l>,<root,x_r>, H_l, H_r};
    (4) (D_l,{H_l})是一棵符合本定义的二叉树,称为根的左子树,(D_r,{H_r})是一棵符合本定义
        的二叉树,称为根的右子树。
 基本操作 P:
  InitBiTree(&T);
    操作结果:构造空二叉树 T。
  DestroyBiTree(&T);
    初始条件:二叉树 T 存在。
    操作结果:销毁二叉树 T。
  CreateBiTree(&T, definition);
```


www.TopSage.com

初始条件：definition 给出二叉树 T 的定义。

操作结果：按 definition 构造二叉树 T。

ClearBiTree(&T)；

初始条件：二叉树 T 存在。

操作结果：将二叉树 T 清为空树。

BiTreeEmpty(T)；

初始条件：二叉树 T 存在。

操作结果：若 T 为空二叉树，则返回 TRUE，否则 FALSE。

BiTreeDepth(T)；

初始条件：二叉树 T 存在。

操作结果：返回 T 的深度。

Root(T)；

初始条件：二叉树 T 存在。

操作结果：返回 T 的根。

Value(T, e)；

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的值。

Assign(T, &e, value)；

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：结点 e 赋值为 value。

Parent(T, e)；

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：若 e 是 T 的非根结点，则返回它的双亲，否则返回“空”。

LeftChild(T, e)；

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的左孩子。若 e 无左孩子，则返回“空”。

RightChild(T, e)；

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的右孩子。若 e 无右孩子，则返回“空”。

LeftSibling(T, e)；

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的左兄弟。若 e 是 T 的左孩子或无左兄弟，则返回“空”。

RightSibling(T, e)；

初始条件：二叉树 T 存在，e 是 T 中某个结点。

操作结果：返回 e 的右兄弟。若 e 是 T 的右孩子或无右兄弟，则返回“空”。

InsertChild(T, p, LR, c)；

初始条件：二叉树 T 存在，p 指向 T 中某个结点，LR 为 0 或 1，非空二叉树 c 与 T 不相交且右子树为空。

操作结果：根据 LR 为 0 或 1，插入 c 为 T 中 p 所指结点的左或右子树。p 所指结点的原有左或右子树则成为 c 的右子树。

DeleteChild(T, p, LR)；

初始条件：二叉树 T 存在，p 指向 T 中某个结点，LR 为 0 或 1。

www.TopSage.com

大家网 TopSage.com

操作结果：根据 LR 为 0 或 1，删除 T 中 p 所指结点的左或右子树。

PreOrderTraverse(T, Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：先序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit()失败，则操作失败。

InOrderTraverse(T, Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：中序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit()失败，则操作失败。

PostOrderTraverse(T, Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：后序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit()失败，则操作失败。

LevelOrderTraverse(T, Visit());

初始条件：二叉树 T 存在，Visit 是对结点操作的应用函数。

操作结果：层序遍历 T，对每个结点调用函数 Visit 一次且仅一次。一旦 visit()失败，则操作失败。

}**ADT** BinaryTree

上述数据结构的递归定义表明二叉树或为空，或是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。由于这两棵子树也是二叉树，则由二叉树的定义，它们也可以是空树。由此，二叉树可以有 5 种基本形态，如图 6.3 所示。

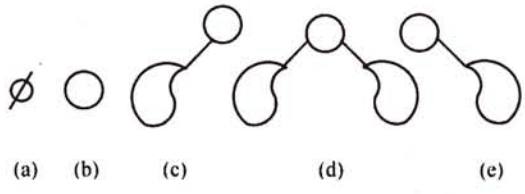


图 6.3　二叉树的 5 种基本形态

(a) 空二叉树；(b) 仅有根结点的二叉树；(c) 右子树为空的二叉树；(d) 左、右子树均非空的二叉树；(e) 左子树为空的二叉树

6.1 节中引入的有关树的术语也都适用于二叉树。

6.2.2　二叉树的性质

二叉树具有下列重要特性。

性质 1　在二叉树的第 i 层上至多有 2^{i-1} 个结点($i \geqslant 1$)。

利用归纳法容易证得此性质。

$i=1$ 时，只有一个根结点。显然，$2^{i-1}=2^0=1$ 是对的。

现在假定对所有的 j，$1 \leqslant j < i$，命题成立，即第 j 层上至多有 2^{j-1} 个结点。那么，可以证明 $j=i$ 时命题也成立。

由归纳假设：第 $i-1$ 层上至多有 2^{i-2} 个结点。由于二叉树的每个结点的度至多为 2，

www.TopSage.com

故在第 i 层上的最大结点数为第 $i-1$ 层上的最大结点数的 2 倍，即 $2\times 2^{i-2}=2^{i-1}$。

性质 2 深度为 k 的二叉树至多有 2^k-1 个结点，($k\geqslant 1$)。

由性质 1 可见，深度为 k 的二叉树的最大结点数为

$$\sum_{i=1}^{k}(\text{第 } i \text{ 层上的最大结点数}) = \sum_{i=1}^{k}2^{i-1} = 2^k - 1$$

性质 3 对任何一棵二叉树 T，如果其终端结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

设 n_1 为二叉树 T 中度为 1 的结点数。因为二叉树中所有结点的度均小于或等于 2，所以其结点总数为

$$n = n_0 + n_1 + n_2 \tag{6-1}$$

再看二叉树中的分支数。除了根结点外，其余结点都有一个分支进入，设 B 为分支总数，则 $n=B+1$。由于这些分支是由度为 1 或 2 的结点射出的，所以又有 $B=n_1+2n_2$。于是得

$$n = n_1 + 2n_2 + 1 \tag{6-2}$$

由式(6-1)和(6-2)得

$$n_0 = n_2 + 1$$

完全二叉树和满二叉树是两种特殊形态的二叉树。

一棵深度为 k 且有 2^k-1 个结点的二叉树称为**满二叉树**。如图 6.4(a)所示是一棵深度为 4 的满二叉树，这种树的特点是每一层上的结点数都是最大结点数。

可以对满二叉树的结点进行连续编号，约定编号从根结点起，自上而下，自左至右。由此可引出完全二叉树的定义。深度为 k 的，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时，称之为**完全二叉树**。[1] 如图 6.4(b)所示为一棵深度为 4 的完全二叉树。显然，这种树的特点是：(1) 叶子结点只可能在层次最大的两层上出现；(2) 对任一结点，若其右分支下的子孙的最大层次为 l，则其左分支下的子孙的最大层次必为 l 或 $l+1$。如图 6.4 中(c)和(d)不是完全二叉树。

完全二叉树将在很多场合下出现，下面介绍完全二叉树的两个重要特性。

性质 4 具有 n 个结点的完全二叉树的深度为 $\lfloor\log_2 n\rfloor+1$[2]。

证明：假设深度为 k，则根据性质 2 和完全二叉树的定义有

$$2^{k-1}-1 < n \leqslant 2^k - 1 \quad \text{或} \quad 2^{k-1} \leqslant n < 2^k$$

于是 $k-1\leqslant\log_2 n<k$，因为 k 是整数，所以 $k=\lfloor\log_2 n\rfloor+1$。

性质 5 如果对一棵有 n 个结点的完全二叉树(其深度为 $\lfloor\log_2 n\rfloor+1$)的结点按层序编号(从第 1 层到第 $\lfloor\log_2 n\rfloor+1$ 层，每层从左到右)，则对任一结点 i($1\leqslant i\leqslant n$)，有

(1) 如果 $i=1$，则结点 i 是二叉树的根，无双亲；如果 $i>1$，则其双亲 PARENT(i)是结点 $\lfloor i/2\rfloor$。

(2) 如果 $2i>n$，则结点 i 无左孩子(结点 i 为叶子结点)；否则其左孩子 LCHILD(i)

---

① 在各种版本的数据结构书中，对完全二叉树的定义均不相同。本书中将一律以此定义为准。

② 符号 $\lfloor x\rfloor$ 表示不大于 x 的最大整数，反之，$\lceil x\rceil$ 表示不小于 x 的最小整数。

www.TopSage.com

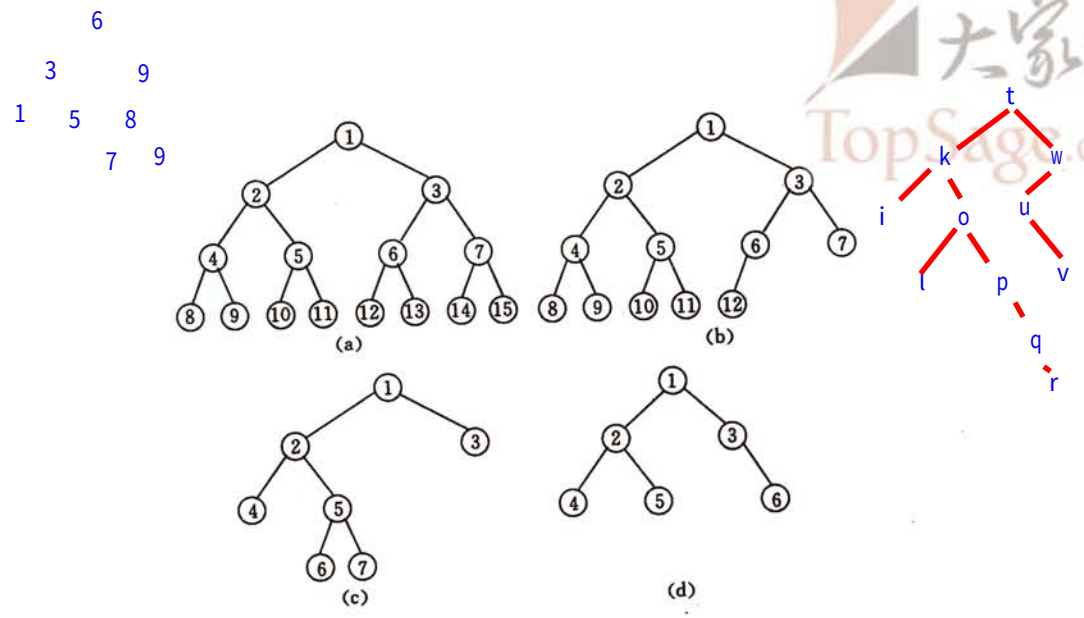


图 6.4　特殊形态的二叉树

(a) 满二叉树；(b) 完全二叉树；(c)和(d)非完全二叉树

是结点 $2i$。

(3) 如果 $2i+1>n$，则结点 i 无右孩子；否则其右孩子 RCHILD(i)是结点 $2i+1$。

我们只要先证明(2)和(3)，便可以从(2)和(3)导出(1)。

对于 $i=1$，由完全二叉树的定义，其左孩子是结点 2。若 $2>n$，即不存在结点 2，此时结点 i 无左孩子。结点 i 的右孩子也只能是结点 3，若结点 3 不存在，即 $3>n$，此时结点 i 无右孩子。

对于 $i>1$ 可分两种情况讨论：(1)设第 j ($1\leqslant j\leqslant\lfloor\log_2 n\rfloor$)层的第一个结点的编号为 i (由二叉树的定义和性质 2 可知 $i=2^{j-1}$)，则其左孩子必为第 $j+1$ 层的第一个结点，其编号为 $2^j=2(2^{j-1})=2i$，若 $2i>n$，则无左孩子；其右孩子必为第 $j+1$ 层的第二个结点，其编号为 $2i+1$，若 $2i+1>n$，则无右孩子；(2)假设第 j($1\leqslant j\leqslant\lfloor\log_2 n\rfloor$)层上某个结点的编号为 i($2^{j-1}\leqslant i<2^j-1$)，且 $2i+1<n$，则其左孩子为 $2i$，右孩子为 $2i+1$，又编号为 $i+1$ 的结点是编号为 i 的结点的右兄弟或者堂兄弟，若它有左孩子，则编号必为 $2i+2=2(i+1)$，若它有右孩子，则其编号必为 $2i+3=2(i+1)+1$。

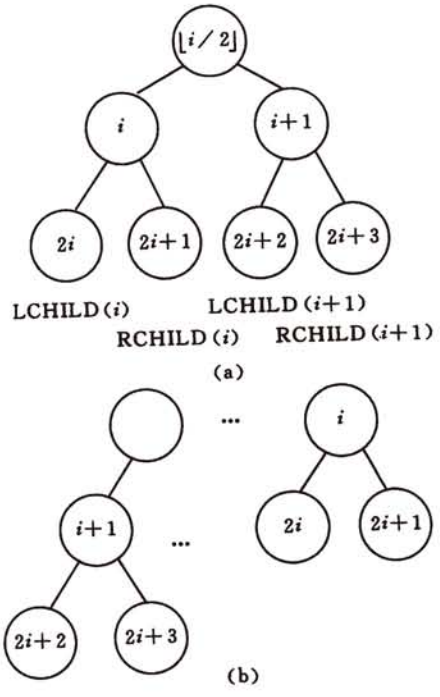


图 6.5　完全二叉树中结点 i 和 $i+1$ 的左、右孩子

(a) 结点 i 和 $i+1$ 在同一层上；

(b) 结点 i 和 $i+1$ 不在同一层上

www.TopSage.com

所示为完全二叉树上结点及其左、右孩子结点之间的关系。

6.2.3 二叉树的存储结构

1. 顺序存储结构

```
//------ 二叉树的顺序存储表示 ------
#define  MAX_TREE_SIZE  100                    // 二叉树的最大结点数
typedef TElemType  SqBiTree[MAX_TREE_SIZE];    // 0 号单元存储根结点
SqBiTree  bt;
```

按照顺序存储结构的定义，在此约定，用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素，即将完全二叉树上编号为 i 的结点元素存储在如上定义的一维数组中下标为 $i-1$ 的分量中。例如，图 6.6(a)所示为图 6.4(b)所示完全二叉树的顺序存储结构。对于一般二叉树，则应将其每个结点与完全二叉树上的结点相对照，存储在一维数组的相应分量中，如图 6.4(c) 所示二叉树的顺序存储结构如图 6.6(b)所示，图中以“0”表示不存在此结点。由此可见，这种顺序存储结构仅适用于完全

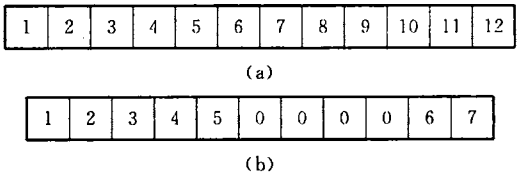


图 6.6 二叉树的顺序存储结构

(a) 完全二叉树；(b)一般二叉树

二叉树。因为，在最坏的情况下，一个深度为 k 且只有 k 个结点的单支树(树中不存在度为 2 的结点)却需要长度为 2^k-1 的一维数组。

2. 链式存储结构

设计不同的结点结构可构成不同形式的链式存储结构。由二叉树的定义得知，二叉树的结点(如图 6.7(a)所示)由一个数据元素和分别指向其左、右子树的两个分支构成，则表示二叉树的链表中的结点至少包含 3 个域：数据域和左、右指针域，如图 6.7(b)所示。有时，为了便于找到结点的双亲，则还可在结点结构中增加一个指向其双亲结点的指针域，如图 6.7(c)所示。利用这两种结点结构所得二叉树的存储结构分别称之为二叉链表和三叉链表，如图 6.8 所示。链表的头指针指向二叉树的根结点。容易证得，在含有 n 个结点的二叉链表中有 $n+1$ 个空链域。在 6.3 节中我们将会看到可以利用这些空链域存储其他有用信息，从而得到另一种链式存储结构——线索链表。以下是二叉链表的定义和部分基本操作的函数原型说明。

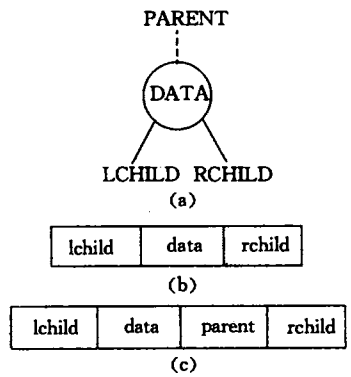


图 6.7 二叉树的结点及其存储结构

(a) 二叉树的结点；

(b) 含有两个指针域的结点结构；

(c) 含有三个指针域的结点结构

www.TopSage.com

大家网
TopSage.com

```
// - - - - - 二叉树的二叉链表存储表示 - - - - -
typedef struct BiTNode {
   TElemType        data;
   struct BiTNode   * lchild, * rchild; // 左右孩子指针
}BiTNode, * BiTree;

// - - - - - 基本操作的函数原型说明(部分) - - - - -
Status CreateBiTree(BiTree &T);
   // 按先序次序输入二叉树中结点的值(一个字符),空格字符表示空树,
   // 构造二叉链表表示的二叉树 T。
Status PreOrderTraverse(BiTree T, Status ( * Visit)(TElemType e));
   // 采用二叉链表存储结构,Visit 是对结点操作的应用函数。
   // 先序遍历二叉树 T,对每个结点调用函数 Visit 一次且仅一次。
   // 一旦 visit()失败,则操作失败。
Status InOrderTraverse(BiTree T, Status ( * Visit)(TElemType e));
   // 采用二叉链表存储结构,Visit 是对结点操作的应用函数。
   // 中序遍历二叉树 T,对每个结点调用函数 Visit 一次且仅一次。
   // 一旦 visit()失败,则操作失败。
Status PostOrderTraverse(BiTree T, Status ( * Visit)(TElemType e));
   // 采用二叉链表存储结构,Visit 是对结点操作的应用函数。
   // 后序遍历二叉树 T,对每个结点调用函数 Visit 一次且仅一次。
   // 一旦 visit()失败,则操作失败。
Status LevelOrderTraverse(BiTree T, Status ( * Visit)(TElemType e));
   // 采用二叉链表存储结构,Visit 是对结点操作的应用函数。
   // 层序遍历二叉树 T,对每个结点调用函数 Visit 一次且仅一次。
   // 一旦 visit()失败,则操作失败。
```

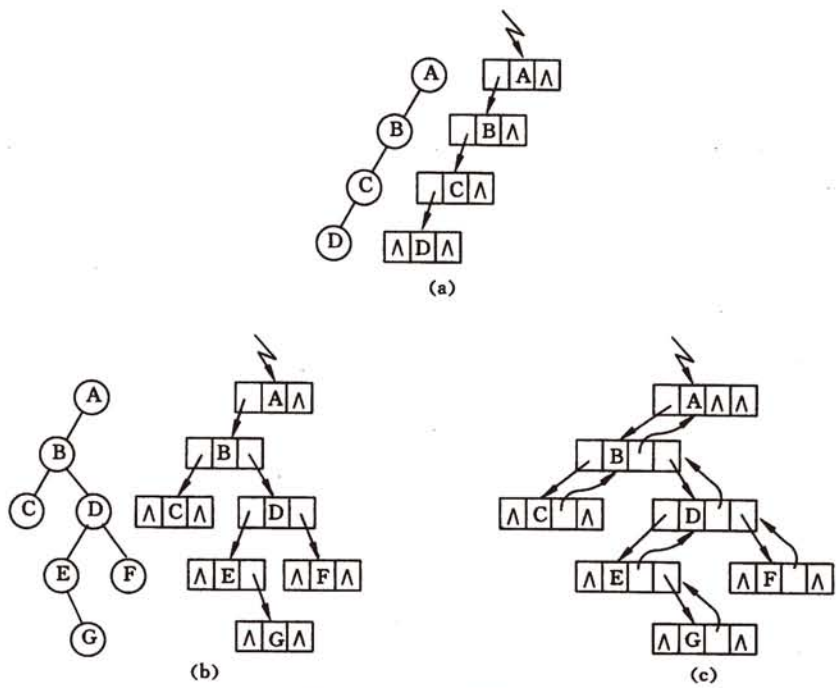


图 6.8 链表存储结构

(a) 单支树的二叉链表; (b) 二叉链表; (c) 三叉链表

www.TopSage.com

在不同的存储结构中，实现二叉树的操作方法也不同，如找结点 x 的双亲 PARENT(T,e)，在三叉链表中很容易实现，而在二叉链表中则需从根指针出发巡查。由此，在具体应用中采用什么存储结构，除根据二叉树的形态之外还应考虑需进行何种操作。读者可试以 6.2 节中定义的各种操作对以上 3 种存储结构进行比较。

6.3 遍历二叉树和线索二叉树

6.3.1 遍历二叉树

在二叉树的一些应用中，常常要求在树中查找具有某种特征的结点，或者对树中全部结点逐一进行某种处理。这就提出了一个**遍历二叉树**(traversing binary tree)的问题，即如何按某条搜索路径巡访树中每个结点，使得每个结点均被访问一次，而且仅被访问一次。“访问”的含义很广，可以是对结点作各种处理，如输出结点的信息等。遍历对线性结构来说，是一个容易解决的问题。而对二叉树则不然，由于二叉树是一种非线性结构，每个结点都可能有两棵子树，因而需要寻找一种规律，以便使二叉树上的结点能排列在一个线性队列上，从而便于遍历。

回顾二叉树的递归定义可知，二叉树是由 3 个基本单元组成：根结点、左子树和右子树。因此，若能依次遍历这三部分，便是遍历了整个二叉树。假如以 L、D、R 分别表示遍历左子树、访问根结点和遍历右子树，则可有 DLR、LDR、LRD、DRL、RDL、RLD 这 6 种遍历二叉树的方案。若限定先左后右，则只有前 3 种情况，分别称之为先(根)序遍历、中(根)序遍历和后(根)序遍历。基于二叉树的递归定义，可得下述遍历二叉树的递归算法定义。

先序遍历二叉树的操作定义为：

若二叉树为空，则空操作；否则

(1) 访问根结点；

(2) 先序遍历左子树；

(3) 先序遍历右子树。

中序遍历二叉树的操作定义为：

若二叉树为空，则空操作；否则

(1) 中序遍历左子树；

(2) 访问根结点；

(3) 中序遍历右子树。

后序遍历二叉树的操作定义为：

若二叉树为空，则空操作；否则

(1) 后序遍历左子树；

(2) 后序遍历右子树；

(3) 访问根结点。

算法 6.1 给出了先序遍历二叉树基本操作的递归算法在二叉链表上的实现。读者可类似地实现中序遍历和后序遍历的递归算法，此处不再一一列举。

www.TopSage.com

大家网 TopSage.com

```
Status PreOrderTraverse( BiTree T, Status ( * Visit)(TElemType e) ) {
  // 采用二叉链表存储结构,Visit 是对数据元素操作的应用函数,
  // 先序遍历二叉树 T 的递归算法,对每个数据元素调用函数 Visit。
  // 最简单的 Visit 函数是:
  //     Status PrintElement( TElemType e ) {   // 输出元素 e 的值
  //       printf( e );                         // 实用时,加上格式串
  //       return OK;
  //     }
  // 调用实例:PreOrderTraverse(T, PrintElement);
  if (T) {
    if (Visit(T->data))
      if (PreOrderTraverse(T->lchild,Visit))
        if (PreOrderTraverse(T->rchild,Visit))  return OK;
    return ERROR;
  }else  return OK;
} // PreOrderTraverse
```

算法 6.1

例如图 6.9 所示的二叉树[1]表示下述表达式

$$a+b*(c-d)-e/f$$

若先序遍历此二叉树,按访问结点的先后次序将结点排列起来,可得到二叉树的先序序列为

$$-+a*b-cd/ef \tag{6-3}$$

类似地,中序遍历此二叉树,可得此二叉树的中序序列为

$$a+b*c-d-e/f \tag{6-4}$$

后序遍历此二叉树,可得此二叉树的后序序列为

$$abcd-*+ef/- \tag{6-5}$$

从表达式来看,以上 3 个序列(6-3)、(6-4)和(6-5)恰好为表达式的前缀表示(波兰式)、中缀表示和后缀表示(逆波兰式)。

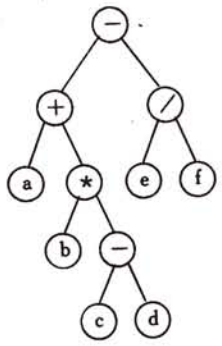


图 6.9 表达式(a+b*(c-d)-e/f)的二叉树

从上述二叉树遍历的定义可知,3 种遍历算法之不同处仅在于访问根结点和遍历左、右子树的先后关系。如果在算法中暂且抹去和递归无关的 Visit 语句,则 3 个遍历算法完全相同。由此,从递归执行过程的角度来看先序、中序和后序遍历也是完全相同的。图 6.10(b)中用带箭头的虚线表示了这 3 种遍历算法的递归执行过程。其中,向下的箭头表示更深一层的递归调用,向上的箭头表示从递归调用退出返回;虚线旁的三角形、圆形和方形内的字符分别表示在先序、中序和后序遍

---

1 以二叉树表示表达式的递归定义如下:若表达式为数或简单变量,则相应二叉树中仅有一个根结点,其数据域存放该表达式信息;若表达式=(第一操作数)(运算符)(第二操作数),则相应的二叉树中以左子树表示第一操作数;右子树表示第二操作数;根结点的数据域存放运算符(若为一元算符,则左子树为空)。操作数本身又为表达式。

www.TopSage.com

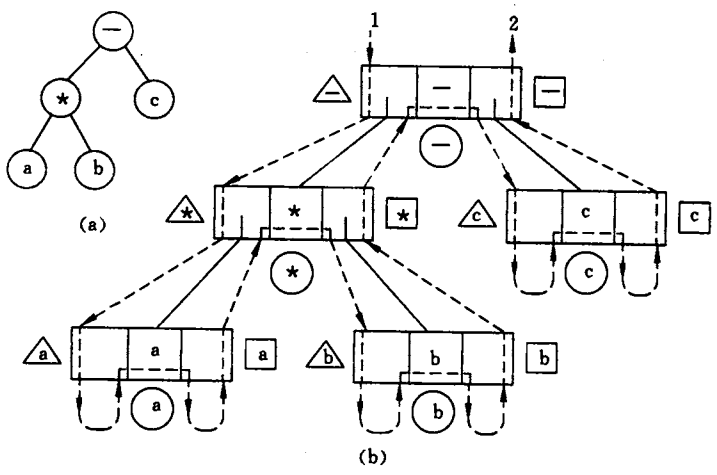


图 6.10　3 种遍历过程示意图

(a) 表达式(a * b－c)的二叉树；(b) 遍历的递归执行过程

历二叉树过程中访问结点时输出的信息。例如，由于中序遍历中访问结点是在遍历左子树之后、遍历右子树之前进行，则带圆形的字符标在向左递归返回和向右递归调用之间。由此，只要沿虚线从 1 出发到 2 结束，将沿途所见的三角形(或圆形、或方形)内的字符记下，便得遍历二叉树的先序(或中序、或后序)序列。例如，从图 6.10(b)分别可得图 6.10(a)所示表达式的前缀表示(－ * abc)、中缀表示(a * b－c)和后缀表示(ab * c－)。

仿照递归算法执行过程中递归工作栈的状态变化状况可直接写出相应的非递归算法。例如，从中序遍历递归算法执行过程中递归工作栈的状态可见：(1)工作记录中包含两项，其一是递归调用的语句编号，其二是指向根结点的指针，则当栈顶记录中的指针非空时，应遍历左子树，即指向左子树根的指针进栈；(2)若栈顶记录中的指针值为空，则应退至上一层，若是从左子树返回，则应访问当前层即栈顶记录中指针所指的根结点；(3)若是从右子树返回，则表明当前层的遍历结束，应继续退栈。从另一角度看，这意味着遍历右子树时不再需要保存当前层的根指针，可直接修改栈顶记录中的指针即可。由此可得两个中序遍历二叉树的非递归算法如算法 6.2 和 6.3 所示，供读者分析比较，以加深理解。

```
Status InOrderTraverse(BiTree T, Status ( * Visit)(TElemType e)) {
  // 采用二叉链表存储结构，Visit 是对数据元素操作的应用函数。
  // 中序遍历二叉树 T 的非递归算法，对每个数据元素调用函数 Visit。
  InitStack(S);  Push(S, T);      // 根指针进栈
  while (!StackEmpty(S)) {
    while (GetTop(S, p) && p) Push(S, p->lchild); // 向左走到尽头
    Pop(S, p);                    // 空指针退栈
    if (!StackEmpty(S)) {         // 访问结点，向右一步
      Pop(S, p);  if (!Visit(p->data)) return ERROR;
      Push(S, p->rchild);
    }// if
  }// While
```


www.TopSage.com

大家网
TopSage.com

```
    return OK;
}// InOrderTraverse
```

算法 6.2

```
Status InOrderTraverse(BiTree T, Status ( * Visit)(TElemType e)) {
   // 采用二叉链表存储结构,Visit 是对数据元素操作的应用函数。
   // 中序遍历二叉树 T 的非递归算法,对每个数据元素调用函数 Visit。
   InitStack(S);  p = T;
   while (p || !StackEmpty(S)) {
      if (p) {Push(S, p);  p = p->lchild; } // 根指针进栈,遍历左子树
      else {        // 根指针退栈,访问根结点,遍历右子树
         Pop(S, p);  if (!Visit(p->data)) return ERROR;
         p = p->rchild;
      }// else
   }// While
   return OK;
}// InOrderTraverse
```

算法 6.3

"遍历"是二叉树各种操作的基础,可以在遍历过程中对结点进行各种操作,如:对于一棵已知树可求结点的双亲,求结点的孩子结点,判定结点所在层次等,反之,也可在遍历过程中生成结点,建立二叉树的存储结构。例如,算法 6.4 是一个按先序序列建立二叉树的二叉链表的过程。对图 6.8(b)所示二叉树,按下列次序顺序读入字符

A B C Φ Φ D E Φ G Φ Φ F Φ Φ Φ

(其中 Φ 表示空格字符)可建立相应的二叉链表。

```
Status CreateBiTree(BiTree &T) {
   // 按先序次序输入二叉树中结点的值(一个字符),空格字符表示空树,
   // 构造二叉链表表示的二叉树 T。
   scanf(&ch);
   if (ch == ' ') T = NULL;
   else {
      if (!(T = (BiTNode * )malloc(sizeof(BiTNode)))) exit(OVERFLOW);
      T->data = ch;              // 生成根结点
      CreateBiTree(T->lchild);   // 构造左子树
      CreateBiTree(T->rchild);   // 构造右子树
   }
   return OK;
}// CreateBiTree
```

算法 6.4

对二叉树进行遍历的搜索路径除了上述按先序、中序或后序外,还可从上到下、从左到右按层次进行。

显然,遍历二叉树的算法中的基本操作是访问结点,则不论按哪一种次序进行遍历,对含 n 个结点的二叉树,其时间复杂度均为 $O(n)$。所需辅助空间为遍历过程中栈的最大容量,即树的深度,最坏情况下为 n,则空间复杂度也为 $O(n)$。遍历时也可采用二叉树的

www.TopSage.com

其他存储结构，如带标志域的三叉链表（参见算法 6.13），此时因存储结构中已存有遍历所需足够信息，则遍历过程中不需另设栈，也可和 8.5 节将讨论的遍历广义表的算法相类似，采用带标志域的二叉链表作存储结构，并在遍历过程中利用指针域暂存遍历路径，也可省略栈的空间，但这样做将使时间上有很大损失。

6.3.2 线索二叉树

从上节的讨论得知：遍历二叉树是以一定规则将二叉树中结点排列成一个线性序列，得到二叉树中结点的先序序列或中序序列或后序序列。这实质上是对一个非线性结构进行线性化操作，使每个结点（除第一个和最后一个外）在这些线性序列中有且仅有一个直接前驱和直接后继（在不至于混淆的情况，我们省去直接二字）①。例如在图 6.9 所示的二叉树的结点的中序序列 a+b*c-d-e/f 中'c'的前驱是'*'，后继是'-'。

但是，当以二叉链表作为存储结构时，只能找到结点的左、右孩子信息，而不能直接得到结点在任一序列中的前驱和后继信息，这种信息只有在遍历的动态过程中才能得到。

如何保存这种在遍历过程中得到的信息呢？一个最简单的办法是在每个结点上增加两个指针域 fwd 和 bkwd，分别指示结点在依任一次序遍历时得到的前驱和后继信息。显然，这样做使得结构的存储密度大大降低。另一方面，在有 n 个结点的二叉链表中必定存在 $n+1$ 个空链域。由此设想能否利用这些空链域来存放结点的前驱和后继的信息。

试作如下规定：若结点有左子树，则其 lchild 域指示其左孩子，否则令 lchild 域指示其前驱；若结点有右子树，则其 rchild 域指示其右孩子，否则令 rchild 域指示其后继。为了避免混淆，尚需改变结点结构，增加两个标志域

lchild	LTag	data	RTag	rchild

其中：

$$\text{LTag}=\begin{cases}0 & \text{lchild 域指示结点的左孩子}\\ 1 & \text{lchild 域指示结点的前驱}\end{cases}$$

$$\text{RTag}=\begin{cases}0 & \text{rchild 域指示结点的右孩子}\\ 1 & \text{rchild 域指示结点的后继}\end{cases}$$

以这种结点结构构成的二叉链表作为二叉树的存储结构，叫做**线索链表**，其中指向结点前驱和后继的指针，叫做**线索**。加上线索的二叉树称之为**线索二叉树**（Threaded Binary Tree）。例如图 6.11(a)所示为中序线索二叉树，与其对应的中序线索链表如图 6.11(b)所示。其中实线为指针（指向左、右子树），虚线为线索（指向前驱和后继）。对二叉树以某种次序遍历使其变为线索二叉树的过程叫做**线索化**。

在线索树上进行遍历，只要先找到序列中的第一个结点，然后依次找结点后继直至其后继为空时而止。

如何在线索树中找结点的后继？以图 6.11 的中序线索树为例来看，树中所有叶子结点的右链是线索，则右链域直接指示了结点的后继，如结点 b 的后继为结点 *。树中所有

① 注意在本节下文中提到的"前驱"和"后继"均指以某种次序遍历所得序列中的前驱和后继。

www.TopSage.com

大家网
TopSage.com

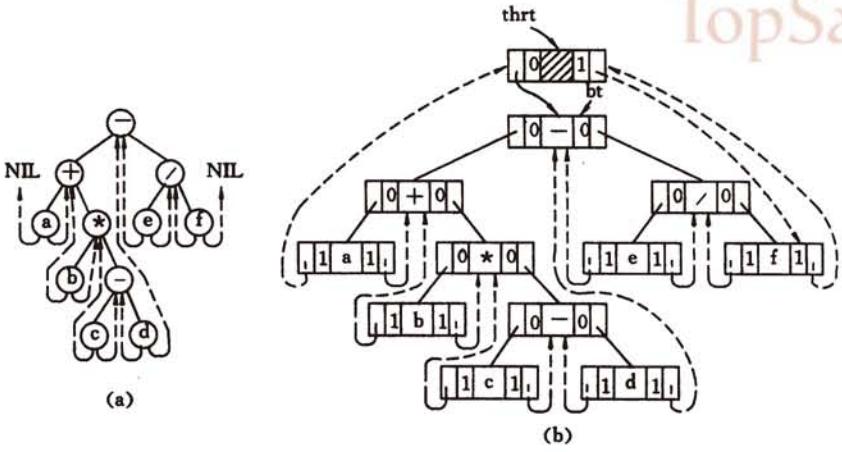


图 6.11　线索二叉树及其存储结构

(a) 中序线索二叉树；(b) 中序线索链表

非终端结点的右链均为指针，则无法由此得到后继的信息。然而，根据中序遍历的规律可知，结点的后继应是遍历其右子树时访问的第一个结点，即右子树中最左下的结点。例如在找结点 * 的后继时，首先沿右指针找到其右子树的根结点“−”，然后顺其左指针往下直至其左标志为 1 的结点，即为结点 * 的后继，在图中是结点 c。反之，在中序线索树中找结点前驱的规律是：若其左标志为“1”，则左链为线索，指示其前驱，否则遍历左子树时最后访问的一个结点(左子树中最右下的结点)为其前驱。

在后序线索树中找结点后继较复杂些，可分 3 种情况：(1)若结点 x 是二叉树的根，则其后继为空；(2)若结点 x 是其双亲的右孩子或是其双亲的左孩子且其双亲没有右子树，则其后继即为双亲结点；(3) 若结点 x 是其双亲的左孩子，且其双亲有右子树，则其后继为双亲的右子树上按后序遍历列出的第一个结点。例如图 6.12 所示为后序后继线索二叉树，结点 B 的后继为结点 C，结点 C 的后继为结点 D，结点 F 的后继为结点 G，而结点 D 的后继为结点 E。可见，在后序线索化树上找后继时需知道结点双亲，即需带标志域的三叉链表作存储结构。

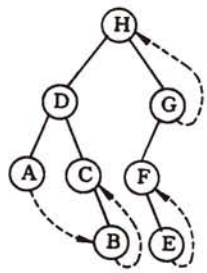


图 6.12　后序后继线索二叉树

可见，在中序线索二叉树上遍历二叉树，虽则时间复杂度亦为 $O(n)$，但常数因子要比上节讨论的算法小，且不需要设栈。因此，若在某程序中所用二叉树需经常遍历或查找结点在遍历所得线性序列中的前驱和后继，则应采用线索链表作存储结构。

```
//------二叉树的二叉线索存储表示------
typedef enum PointerTag { Link, Thread };   // Link==0:指针,Thread==1:线索
typedef struct BiThrNode {
  TElemType          data;
  struct BiThrNode   *lchild, *rchild;  // 左右孩子指针
```


www.TopSage.com

```
    PointerTag        LTag, RTag;          // 左右标志
}BiThrNode, *BiThrTree;
```

为方便起见，仿照线性表的存储结构，在二叉树的线索链表上也添加一个头结点，并令其 lchild 域的指针指向二叉树的根结点，其 rchild 域的指针指向中序遍历时访问的最后一个结点；反之，令二叉树中序序列中的第一个结点的 lchild 域指针和最后一个结点 rchild 域的指针均指向头结点。这好比为二叉树建立了一个双向线索链表，既可从第一个结点起顺后继进行遍历，也可从最后一个结点起顺前驱进行遍历（如图 6.11(b)所示）。下述算法 6.5 正是以双向线索链表为存储结构时对二叉树进行遍历的算法。

```
Status InOrderTraverse_Thr(BiThrTree T, Status (*Visit)(TElemType e)) {
  // T 指向头结点，头结点的左链 lchild 指向根结点，可参见线索化算法。
  // 中序遍历二叉线索树 T 的非递归算法，对每个数据元素调用函数 Visit。
  p = T->lchild;                          // p 指向根结点
  while (p != T) {                        // 空树或遍历结束时，p==T
    while (p->LTag==Link) p = p->lchild;
    if (!Visit(p->data)) return ERROR;    // 访问其左子树为空的结点
    while (p->RTag==Thread && p->rchild!=T) {
      p = p->rchild;  Visit(p->data);     // 访问后继结点
    }
    p = p->rchild;
  }
  return OK;
}// InOrderTraverse_Thr
```

算法 6.5

那么，又如何进行二叉树的线索化呢？由于线索化的实质是将二叉链表中的空指针改为指向前驱或后继的线索，而前驱或后继的信息只有在遍历时才能得到，因此线索化的过程即为在遍历的过程中修改空指针的过程。为了记下遍历过程中访问结点的先后关系，附设一个指针 pre 始终指向刚刚访问过的结点，若指针 p 指向当前访问的结点，则 pre 指向它的前驱。由此可得中序遍历建立中序线索化链表的算法如算法 6.6 和 6.7 所示。

```
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T) {
  // 中序遍历二叉树 T，并将其中序线索化，Thrt 指向头结点。
  if (!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode)))) exit (OVERFLOW);
  Thrt->LTag = Link;  Thrt->RTag = Thread;        // 建头结点
  Thrt->rchild = Thrt;                            // 右指针回指
  if (!T) Thrt->lchild = Thrt;                    // 若二叉树空，则左指针回指
  else {
    Thrt->lchild = T;  pre = Thrt;
    InThreading(T);                               // 中序遍历进行中序线索化
    pre->rchild = Thrt;  pre->RTag = Thread;      // 最后一个结点线索化
    Thrt->rchild = pre;
  }
  return OK;
}// InOrderThreading
```

算法 6.6

www.TopSage.com

大家网
TopSage.com

```
void InThreading(BiThrTree p) {
  if (p) {
    InThreading(p->lchild);  // 左子树线索化
    if (!p->lchild) {p->LTag = Thread;  p->lchild = pre;}  // 前驱线索
    if (!pre->rchild) {pre->RTag = Thread;  pre->rchild = p;}// 后继线索
    pre = p;                 // 保持 pre 指向 p 的前驱
    InThreading(p->rchild);  // 右子树线索化
  }
}// InThreading
```

算法 6.7

6.4 树和森林

这一节我们将讨论树的表示及其遍历操作，并建立森林与二叉树的对应关系。

6.4.1 树的存储结构

在大量的应用中，人们曾使用多种形式的存储结构来表示树。这里，我们介绍 3 种常用的链表结构。

1. 双亲表示法

假设以一组连续空间存储树的结点，同时在每个结点中附设一个指示器指示其双亲结点在链表中的位置，其形式说明如下：

```
//------ 树的双亲表存储表示 ------
#define MAX_TREE_SIZE  100
typedef struct PTNode {  // 结点结构
  TElemType  data;
  int        parent;  // 双亲位置域
}PTNode;
typedef struct {       // 树结构
  PTNode  nodes[MAX_TREE_SIZE];
  int     r,n;         // 根的位置和结点数
}PTree;
```

例如，图 6.13 展示一棵树及其双亲表示的存储结构。

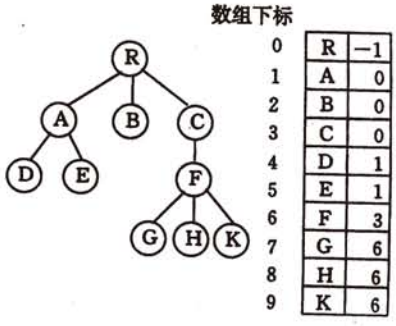


图 6.13 树的双亲表示法示例

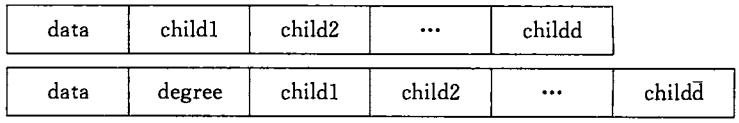
这种存储结构利用了每个结点(除根以外)只有惟一的双亲的性质。PARENT(T,x)操作可以在常量时间内实现。反复调用 PARENT 操作，直到遇见无双亲的结点时，便找到了树的根，这就是 ROOT(x)操作的执行过程。但是，在这种表示法中，求结点的孩子时需要遍历整个结构。

2. 孩子表示法

由于树中每个结点可能有多棵子树，则可用多重链表，即每个结点有多个指针域，其

www.TopSage.com

中每个指针指向一棵子树的根结点，此时链表中的结点可以有如下两种结点格式：



若采用第一种结点格式，则多重链表中的结点是同构的，其中 d 为树的度。由于树中很多结点的度小于 d，所以链表中有很多空链域，空间较浪费，不难推出，在一棵有 n 个结点度为 k 的树中必有 n(k−1)+1 个空链域。若采用第二种结点格式，则多重链表中的结点是不同构的，其中 $\bar{d}$ 为结点的度，degree 域的值同 $\bar{d}$。此时，虽能节约存储空间，但操作不方便。

另一种办法是把每个结点的孩子结点排列起来，看成是一个线性表，且以单链表作存储结构，则 n 个结点有 n 个孩子链表(叶子的孩子链表为空表)。而 n 个头指针又组成一个线性表，为了便于查找，可采用顺序存储结构。这种存储结构可形式地说明如下：

```
//------ 树的孩子链表存储表示 ------
typedef struct CTNode {      // 孩子结点
   int              child;
   struct CTNode * next;
} * ChildPtr;
typedef struct {
   TElemType  data;
   ChildPtr  firstchild;  // 孩子链表头指针
}CTBox;
typedef struct {
   CTBox  nodes[MAX_TREE_SIZE];
   int    n, r;          // 结点数和根的位置;
}CTree;
```

图 6.14(a)是图 6.13 中的树的孩子表示法。与双亲表示法相反，孩子表示法便于那些涉及孩子的操作的实现，却不适用于 PARENT(T,x)操作。我们可以把双亲表示法和孩子表示法结合起来，即将双亲表示和孩子链表合在一起。图 6.14(b)就是这种存储结构的一例，它和图 6.14(a)表示的是同一棵树。

3. 孩子兄弟表示法

又称二叉树表示法，或二叉链表表示法。即以二叉链表作树的存储结构。链表中结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点，分别命名为 firstchild 域和 nextsibling 域。

```
//------ 树的二叉链表(孩子-兄弟)存储表示 ------
typedef struct CSNode {
   ElemType          data;
   struct CSNode   * firstchild, * nextsibling;
}CSNode, * CSTree;
```

图 6.15 是图 6.13 中的树的孩子兄弟链表。利用这种存储结构便于实现各种树的操作。首先易于实现找结点孩子等的操作。例如：若要访问结点 x 的第 i 个孩子，则只要先

www.TopSage.com

大家网
TopSage.com

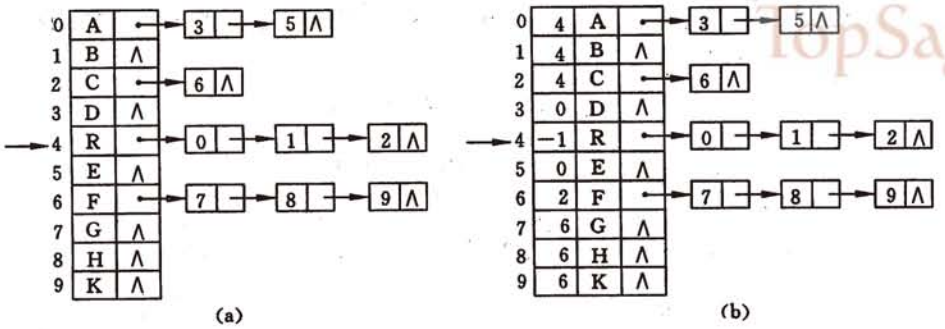


图 6.14　图 6.13 的树的另外两种表示法

(a) 孩子链表；(b) 带双亲的孩子链表

从 firstchild 域找到第 1 个孩子结点，然后沿着孩子结点的 nextsibling 域连续走 i−1 步，便可找到 x 的第 i 个孩子。当然，如果为每个结点增设一个 PARENT 域，则同样能方便地实现 PARENT(T,x)操作。

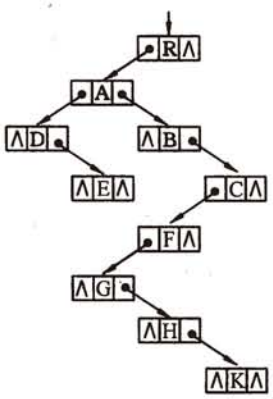


图 6.15　图 6.13 中树的二叉链表表示法

6.4.2　森林与二叉树的转换

由于二叉树和树都可用二叉链表作为存储结构，则以二叉链表作为媒介可导出树与二叉树之间的一个对应关系。也就是说，给定一棵树，可以找到惟一的一棵二叉树与之对应，从物理结构来看，它们的二叉链表是相同的，只是解释不同而已。图 6.16 直观地展示了树与二叉树之间的对应关系。

从树的二叉链表表示的定义可知，任何一棵和树对

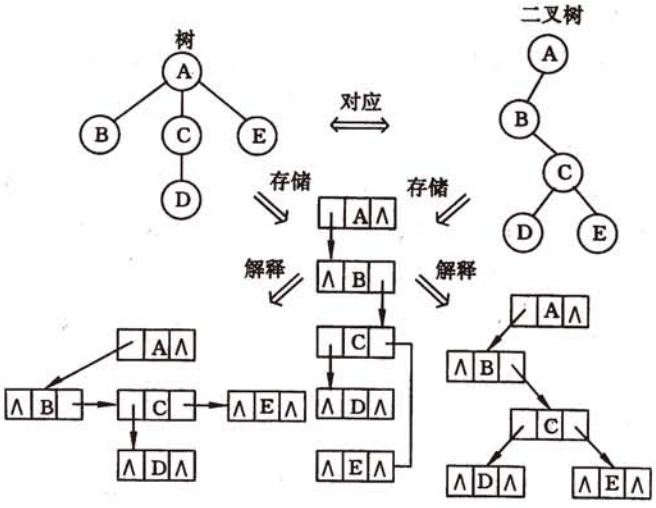


图 6.16　树与二叉树的对应关系示例

www.TopSage.com

应的二叉树，其右子树必空。若把森林中第二棵树的根结点看成是第一棵树的根结点的兄弟，则同样可导出森林和二叉树的对应关系。

例如，图 6.17 展示了森林与二叉树之间的对应关系。

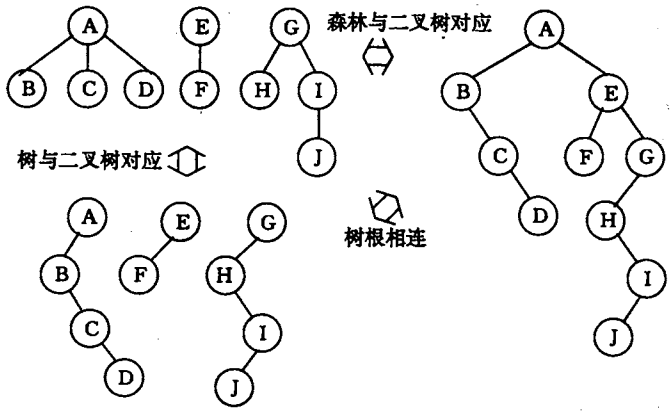


图 6.17　森林与二叉树的对应关系示例

这个一一对应的关系导致森林或树与二叉树可以相互转换，其形式定义如下：

1. 森林转换成二叉树

如果 $F=\{T_1,T_2,\cdots,T_m\}$ 是森林，则可按如下规则转换成一棵二叉树 $B=(root,LB,RB)$。

(1) 若 F 为空，即 $m=0$，则 B 为空树；

(2) 若 F 非空，即 $m\neq0$，则 B 的根 $root$ 即为森林中第一棵树的根 $ROOT(T_1)$；B 的左子树 LB 是从 T_1 中根结点的子树森林 $F_1=\{T_{11},T_{12},\cdots,T_{1m1}\}$ 转换而成的二叉树；其右子树 RB 是从森林 $F'=\{T_2,T_3,\cdots,T_m\}$ 转换而成的二叉树。

2. 二叉树转换成森林

如果 $B=(root,LB,RB)$ 是一棵二叉树，则可按如下规则转换成森林 $F=\{T_1,T_2,\cdots,T_m\}$：

(1) 若 B 为空，则 F 为空；

(2) 若 B 非空，则 F 中第一棵树 T_1 的根 $ROOT(T_1)$ 即为二叉树 B 的根 $root$；T_1 中根结点的子树森林 F_1 是由 B 的左子树 LB 转换而成的森林；F 中除 T_1 之外其余树组成的森林 $F'=\{T_2,T_3,\cdots,T_m\}$ 是由 B 的右子树 RB 转换而成的森林。

从上述递归定义容易写出相互转换的递归算法。同时，森林和树的操作亦可转换成二叉树的操作来实现。

6.4.3　树和森林的遍历

由树结构的定义可引出两种次序遍历树的方法：一种是先根(次序)遍历树，即：先访问树的根结点，然后依次先根遍历根的每棵子树；另一种是后根(次序)遍历，即：先依次后根遍历每棵子树，然后访问根结点。

www.TopSage.com

大家网
TopSage.com

例如，对图 6.16 的树进行先根遍历，可得树的先根序列为

A B C D E

若对此树进行后根遍历，则得树的后根序列为

B D C E A

按照森林和树相互递归的定义，我们可以推出森林的两种遍历方法：

1. 先序遍历森林

若森林非空，则可按下述规则遍历之：

(1) 访问森林中第一棵树的根结点；

(2) 先序遍历第一棵树中根结点的子树森林；

(3) 先序遍历除去第一棵树之后剩余的树构成的森林。

2. 中序遍历森林

若森林非空，则可按下述规则遍历之：

(1) 中序遍历森林中第一棵树的根结点的子树森林；

(2) 访问第一棵树的根结点；

(3) 中序遍历除去第一棵树之后剩余的树构成的森林。

若对图 6.17 中森林进行先序遍历和中序遍历，则分别得到森林的先序序列为

A B C D E F G H I J

中序序列为

B C D A F E H J I G

由上节森林与二叉树之间转换的规则可知，当森林转换成二叉树时，其第一棵树的子树森林转换成左子树，剩余树的森林转换成右子树，则上述森林的先序和中序遍历即为其对应的二叉树的先序和中序遍历。若对图 6.17 中和森林对应的二叉树分别进行先序和中序遍历，可得和上述相同的序列。

由此可见，当以二叉链表作树的存储结构时，树的先根遍历和后根遍历可借用二叉树的先序遍历和中序遍历的算法实现之。

6.5 树与等价问题

在离散数学中，对等价关系和等价类的定义是：

如果集合 S 中的关系 R 是自反的、对称的和传递的，则称它为一个等价关系。

设 R 是集合 S 的等价关系。对任何 $x \in S$，由 $[x]_R = \{y \mid y \in S \wedge xRy\}$ 给出的集合 $[x]_R \subseteq S$ 称为由 $x \in S$ 生成的一个 R 等价类。

若 R 是集合 S 上的一个等价关系，则由这个等价关系可产生这个集合的惟一划分。即可以按 R 将 S 划分为若干不相交的子集 $S_1, S_2, \cdots$，它们的并即为 S，则这些子集 S_i 便称为 S 的 R 等价类。

等价关系是现实世界中广泛存在的一种关系，许多应用问题可以归结为按给定的等价关系划分某集合为等价类，通常称这类问题为等价问题。

例如在 FORTRAN 语言中，可以利用 EQUIVALENCE 语句使数个程序变量共享同

www.TopSage.com

一存储单位，这问题实质就是按 EQUIVALANCE 语句确定的关系对程序中的变量集合进行划分，所得等价类的数目即为需要分配的存储单位，而同一等价类中的程序变量可被分配到同一存储单位中去。此外，划分等价类的算法思想也可用于求网络的最小生成树等图的算法中。

应如何划分等价类呢？假设集合 S 有 n 个元素，m 个形如$(x,y)(x,y\in S)$的等价偶对确定了等价关系 R，需求 S 的划分。

确定等价类的算法可如下进行：

(1) 令 S 中每个元素各自形成一个只含单个成员的子集，记作 $S_1,S_2,\cdots,S_n$。

(2) 重复读入 m 个偶对，对每个读入的偶对(x,y)，判定 x 和 y 所属子集。不失一般性，假设 $x\in S_i,y\in S_j$，若 $S_i\neq S_j$，则将 S_i 并入 S_j 并置 S_i 为空(或将 S_j 并入 S_i 并置 S_j 为空)。则当 m 个偶对都被处理过后，$S_1,S_2,\cdots,S_n$ 中所有非空子集即为 S 的 R 等价类。

从上述可见，划分等价类需对集合进行的操作有 3 个：其一是构造只含单个成员的集合；其二是判定某个单元素所在子集；其三是归并两个互不相交的集合为一个集合。由此，需要一个包含上述 3 种操作的抽象数据类型 MFSet。

```
ADT MFSet {
    数据对象： 若设 S 是 MFSet 型的集合，则它由 n(n>0)个子集 Si(i = 1,2,…,n)构成，每个子
               集的成员都是子界[-maxnumber..maxnumber]内的整数；
    数据关系：S1 ∪ S2 ∪ … ∪ Sn = S   Si⊂S(i = 1,2,…,n)
    基本操作：
  Initial(&S, n, x1, x2,…,xn);
    操作结果：初始化操作。构造一个由 n 个子集(每个子集只含单个成员 xi)构成的集合 S。
  Find(S, x);
    初始条件：S 是已存在的集合，x 是 S 中某个子集的成员。
    操作结果：查找函数。确定 S 中 x 所属子集 Si。
  Merge(&S , i, j );
    初始条件：Si 和 Sj 是 S 中的两个互不相交的非空集合。
    操作结果：归并操作。将 Si 和 Sj 中的一个并入另一个中。
}ADT MFSet;
```

以集合为基础(结构)的抽象数据类型可用多种实现方法，如用位向量表示集合或用有序表表示集合等。如何高效地实现以集合为基础的抽象数据类型，则取决于该集合的大小以及对此集合所进行的操作。根据 MFSet 类型中定义的查找函数和归并操作的特点，我们可利用树型结构表示集合。约定：以森林 $F=(T_1,T_2,\cdots,T_m)$表示 MFSet 型的集合 S，森林中的每一棵树 $T_i(i=1,2,\cdots,n)$表示 S 中的一个元素——子集 $S_i(S_i\subset S,i=1,2,\cdots,n)$，树中每个结点表示子集中的一个成员 x，为操作方便起见，令每个结点中含有一个指向其双亲的指针，并约定根结点的成员兼作子集的名称。例如，图 6.18(a)和(b)中的两棵树分别表示子集 $S_1=\{1,3,6,9\}$和 $S_2=\{2,8,10\}$。显然，这样的树形结构易于实现上述两种集合的操作。由于各子集中的成员均不相同，则实现集合的“并”操作，只要将一棵子集树的根指向另一棵子集树的根即可。例如：图6.18(c)中 $S_3=S_1\cup S_2$。同时，完成找某个成员所在集合的操作，只要从该成员结点出发，顺链而进，直至找到树的根结点为止。

www.TopSage.com

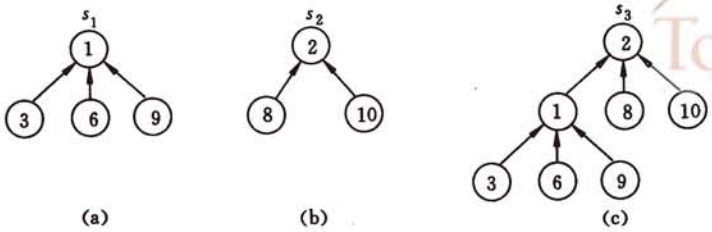


图 6.18 集合的一种表示法

(a) $S_1=\{1,3,6,9\}$; (b) $S_2=\{2,8,10\}$; (c) $S_3=S_1 \cup S_2$

为便于实现这样两种操作，应采用双亲表示法作存储结构，如下所示：

```
// - - - - - ADT MFSet 的树的双亲表存储表示 - - - - -
typedef PTree  MFSet;
```

此时，查找函数和归并操作的实现如算法 6.8 和算法 6.9 所示。

```
int find_mfset(MFSet S, int i) {
   // 找集合 S 中 i 所在子集的根。
   if (i<1 || i>S.n)  return  -1;       // i 不属 S 中任一子集
   for (j = i; S.nodes[j].parent>0; j = S.nodes[j].parent);
   return j;
}// find_mfset
```

算法 6.8

```
Status merge_mfset(MFSet &S, int i, int j) {
   // S.nodes[i]和 S.nodes[j]分别为 S 的互不相交的两个子集 Si 和 Sj 的根结点。
   // 求并集 Si∪Sj。
   if (i<1 || i>S.n || j<1 || j>S.n) return ERROR;
   S.nodes[i].parent = j;
   return OK;
}// merge_mfset
```

算法 6.9

算法 6.8 和算法 6.9 的时间复杂度分别为 $O(d)$和 $O(1)$，其中 d 是树的深度。从前面的讨论可知，这种表示集合的树的深度和树形成的过程有关。试看一个极端的例子。假设有 n 个子集 $S_1,S_2,\cdots,S_n$，每个子集只有一个成员 $S_i=\{i\}(i=1,2,\cdots,n)$，可用 n 棵只有一个根结点的树表示，如图 6.19(a)表示。现作 $n-1$ 次“并”操作，并假设每次都是含成员多的根结点指向含成员少的根结点，则最后得到的集合树的深度为 n，如图 6.19(b)所示。如果再加上在每次“并”操作之后都要进行查找成员“1”所在子集的操作，则全部操作的时间便是 $O(n^2)$了。

改进的办法是在作“并”操作之前先判别子集中所含成员的数目，然后令含成员少的子集树根结点指向含成员多的子集的根。为此，需相应地修改存储结构：令根结点的 parent 域存储子集中所含成员数目的负值。修改后的“并”操作算法如算法 6.10 所示。

www.TopSage.com

```
void mix_mfset (MFSet & S, int i, int j) {
  // S.nodes[i]和 S.nodes[j]分别为 S 的互不相交
  // 的两个子集 Si 和 Sj 的根结点。求并集 Si∪Sj
  if (i<1 || i>S.n || j<1 || j>S.n)
    return ERROR;
  if (S.nodes[i].parent>S.nodes[j].parent) {
  // Si 所含成员数比 Sj 少
    S.nodes[j].parent += S.nodes[i].parent;
    S.nodes[i].parent = j;
  }else {
    S.nodes[i].parent += S.nodes[j].parent;
    S.nodes[j].parent = i;
  }
  return OK;
}// mix_mfset
```

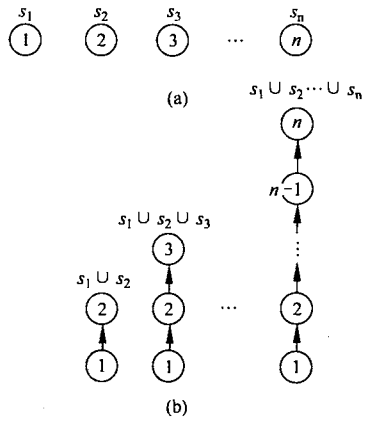


图 6.19 “并”操作的一种极端情形
(a) n 个集合； (b) “并”操作

算法 6.10

可以证明，按算法 6.10 进行“并”操作得到的集合树，其深度不超过$\lfloor \log_2 n \rfloor + 1$[①]，其中$n$为集合 S 中所有子集所含成员数的总和。

由此，利用算法 find_mfset 和 mix_mfset 解等价问题的时间复杂度为 $O(n\log_2 n)$（当集合中有 n 个元素时，至多进行 $n-1$ 次 mix 操作）。

例 6-1 假设集合 $S=\{x \mid 1 \leqslant x \leqslant n$ 是正整数$\}$，R 是 S 上的一个等价关系。

$$R = \{(1,2),(3,4),(5,6),(7,8),(1,3),(5,7),(1,5),\cdots\}$$

现求 S 的等价类。

以 MFSet 类型的变量 S 表示集合 S，S 中成员个数为 S.n。开始时，由于每个成员自成一个等价类，则 S.nodes[i].parent 的值均为-1。之后，每处理一个等价偶对(i,j)，首先必须确定 i 和 j 各自所属集合，若这两个集合相同，则说明此等价关系是多余的，无需作处理；否则就合并这两个集合。图 6.20 展示了处理 R 中前 7 个等价关系时 S 的变化状况（图中省去了结点的数据域），图 6.21(a)所示为和最后一个 S 状态相应的树的形态。显然，随着子集逐对合并，树的深度也越来越大，为了进一步减少确定元素所在集合的时间，我们还可进一步将算法 6.8 改进为算法 6.11。当所查元素 i 不在树的第二层时，在算法中增加一个“压缩路径”的功能，即将所有从根到元素 i 路径上的元素都变成树根的孩子。

---

① 用归纳法证明之：

当 $i=1$ 时，树中只有一个根结点，即深度为 1，又$\lfloor \log_2 1 \rfloor + 1 = 1$，∴正确。

假设 $i \leqslant n-1$ 时成立，试证 $i=n$ 时亦成立。不失一般性，可以假设此树是由含有 $m(1 \leqslant m \leqslant n/2)$个元素、根为 j 的树 S_j 和含有 $n-m$ 个元素、根为 k 的树 S_k 合并而得，按算法 6.10 根 j 指向根 k，即 k 为合并后的根结点。

若合并前子树 S_j 的深度＜子树 S_k 的深度，则合并后的树深和 S_k 相同，不超过$\lfloor \log_2 (n-m) \rfloor + 1$，显然不超过$\lfloor \log_2 n \rfloor + 1$。

若合并前子树 S_j 的深度≥子树 S_k 的深度，则合并后的树深为 S_j 的树深＋1，即$(\lfloor \log_2 m \rfloor + 1) + 1 = \lfloor \log_2 (2m) \rfloor + 1 \leqslant \lfloor \log_2 n \rfloor + 1$。

www.TopSage.com

大家网
TopSage.com

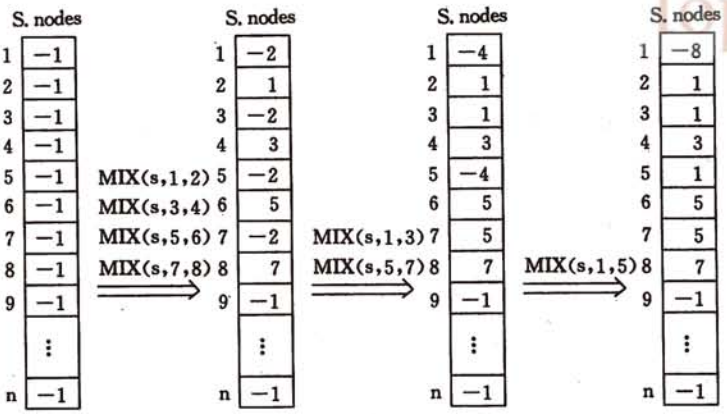


图 6.20　求等价类过程示例

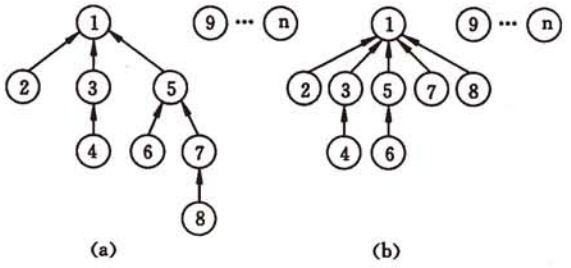


图 6.21　表示集合的树

(a) 压缩路径之前；(b) 压缩路径之后

```
int fix_mfset(MFSet &S, int i) {
   // 确定 i 所在子集,并将从 i 至根路径上所有结点都变成根的孩子结点。
   if (i<1 || i>S.n) return -1;          // i 不是 S 中任一子集的成员
   for (j = i; S.nodes[j].parent>0; j = S.nodes[j].parent);
   for (k = i; k!= j; k = t) {
      t = S.nodes[k].parent;  S.nodes[k].parent = j;
   }
   return j;
}// fix_mfset
```

算法　6.11

假设例 6-1 中 R 的第 8 个等价偶对为(8,9)，则在执行 fix(s,8)的操作之后图 6.21(a)的树就变成图 6.21(b)的树。

已经证明，利用算法 fix_mfset 和 mix_mfset 划分大小为 n 的集合为等价类的时间复杂度为 $O(n\alpha(n))$[6]。其中 $\alpha(n)$是一个增长极其缓慢的函数，若定义单变量的阿克曼函数为 $A(x)=A(x,x)$，则函数 $\alpha(n)$定义为 $A(x)$的拟逆，即 $\alpha(n)$的值是使 $A(x)\geqslant n$ 成立的最小 x。所以，对于通常所见到的正整数 n 而言，$\alpha(n)\leqslant 4$。

www.TopSage.com

6.6 赫夫曼树及其应用

赫夫曼(Huffman)树，又称最优树，是一类带权路径长度最短的树，有着广泛的应用。本节先讨论最优二叉树。

6.6.1 最优二叉树(赫夫曼树)

首先给出路径和路径长度的概念。从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支数目称做**路径长度**。**树的路径长度**是从树根到每一结点的路径长度之和。6.2.1 节中定义的完全二叉树就是这种路径长度最短的二叉树。

若将上述概念推广到一般情况，考虑带权的结点。结点的带权路径长度为从该结点到树根之间的路径长度与结点上权的乘积。**树的带权路径长度**为树中所有叶子结点的带权路径长度之和，通常记作 $WPL=\sum_{k=1}^{n} w_k l_k$ 。

假设有 n 个权值$\{w_1, w_2, \cdots, w_n\}$，试构造一棵有 n 个叶子结点的二叉树，每个叶子结点带权为 w_i，则其中带权路径长度 WPL 最小的二叉树称做**最优二叉树**或**赫夫曼树**。

例如，图 6.22 中的 3 棵二叉树，都有 4 个叶子结点 a、b、c、d，分别带权 7、5、2、4，它们的带权路径长度分别为

(a) $WPL=7\times2+5\times2+2\times2+4\times2=36$

(b) $WPL=7\times3+5\times3+2\times1+4\times2=46$

(c) $WPL=7\times1+5\times2+2\times3+4\times3=35$

其中以(c)树的为最小。可以验证，它恰为赫夫曼树，即其带权路径长度在所有带权为 7、5、2、4 的 4 个叶子结点的二叉树中居最小。

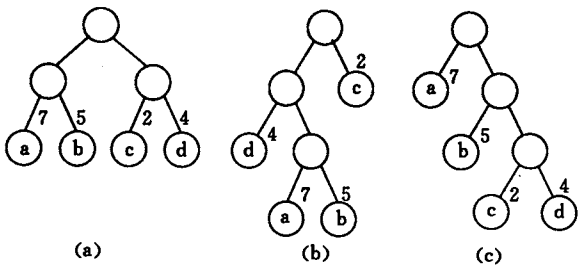


图 6.22 具有不同带权路径长度的二叉树

在解某些判定问题时，利用赫夫曼树可以得到最佳判定算法。例如，要编制一个将百分制转换成五级分制的程序。显然，此程序很简单，只要利用条件语句便可完成。如：

```
if (a<60) b = "bad";
else if (a<70) b = "pass";
     else if (a<80) b = "general";
          else if (a<90) b = "good";
               else b = "excellent";
```


www.TopSage.com

大家网 TopSage.com

这个判定过程可以图 6.23(a)的判定树来表示。如果上述程序需反复使用，而且每次的输入量很大，则应考虑上述程序的质量问题，即其操作所需时间。因为在实际生活中，学生的成绩在 5 个等级上的分布是不均匀的。假设其分布规律如下表所示：

分数	0—59	60—69	70—79	80—89	90—100
比例数	0.05	0.15	0.40	0.30	0.10

则 80%以上的数据需进行 3 次或 3 次以上的比较才能得出结果。假定以 5,15,40,30 和 10 为权构造一棵有 5 个叶子结点的赫夫曼树，则可得到如图 6.23(b)所示的判定过程，它可使大部分的数据经过较少的比较次数得出结果。但由于每个判定框都有两次比较，将这两次比较分开，我们得到如图 6.23(c)所示的判定树，按此判定树可写出相应的程序。假设现有 10 000 个输入数据，若按图 6.23(a)的判定过程进行操作，则总共需进行 31 500 次比较；而若按图 6.23(c)的判定过程进行操作，则总共仅需进行 22 000 次比较。

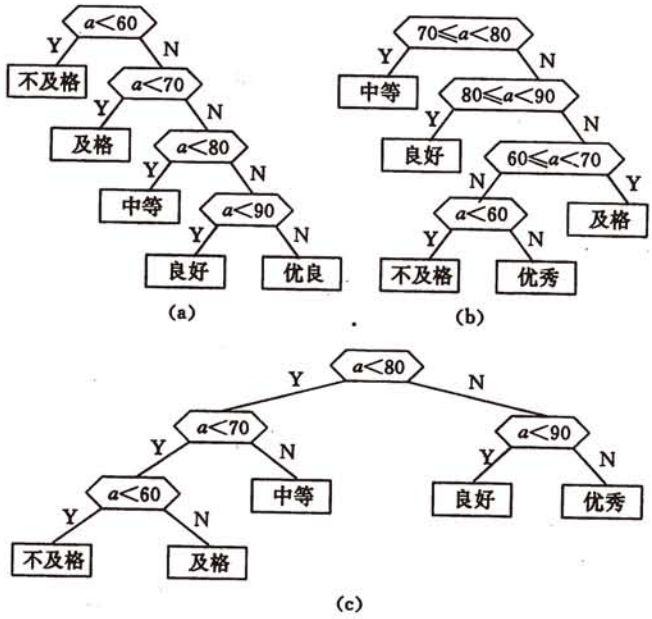


图 6.23　转换五级分制的判定过程

那么，如何构造赫夫曼树呢？赫夫曼最早给出了一个带有一般规律的算法，俗称赫夫曼算法。现叙述如下：

被访问的概率

(1) 根据给定的 n 个权值$\{w_1, w_2, \cdots, w_n\}$构成 n 棵二叉树的集合 $F=\{T_1, T_2, \cdots, T_n\}$，其中每棵二叉树 T_i 中只有一个带权为 w_i 的根结点，其左右子树均空。

(2) 在 F 中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

(3) 在 F 中删除这两棵树，同时将新得到的二叉树加入 F 中。

(4) 重复(2)和(3)，直到 F 只含一棵树为止。这棵树便是赫夫曼树。

www.TopSage.com

例如，图 6.24 展示了图 6.22(c)的赫夫曼树的构造过程。其中，根结点上标注的数字是所赋的权。

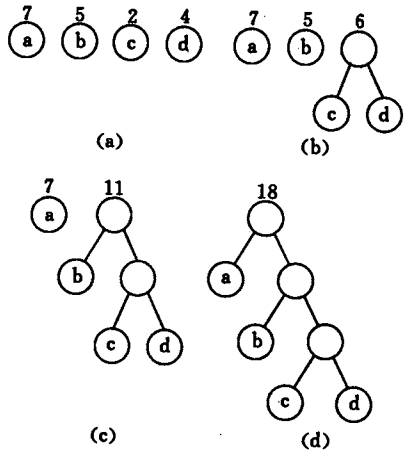


图 6.24　赫夫曼树的构造过程

算法的具体描述和实际问题所采用的存储结构有关，将留在下节进行讨论。

6.6.2　赫夫曼编码

目前，进行快速远距离通信的主要手段是电报，即将需传送的文字转换成由二进制的字符组成的字符串。例如，假设需传送的电文为'A B A C C D A'，它只有 4 种字符，只需两个字符的串便可分辨。假设 A、B、C、D 的编码分别为 00、01、10 和 11，则上述 7 个字符的电文便为'00010010101100'，总长 14 位，对方接收时，可按二位一分进行译码。

当然，在传送电文时，希望总长尽可能地短。如果对每个字符设计长度不等的编码，且让电文中出现次数较多的字符采用尽可能短的编码，则传送电文的总长便可减少。如果设计 A、B、C、D 的编码分别为 0、00、1 和 01，则上述 7 个字符的电文可转换成总长为 9 的字符串'000011010'。但是，这样的电文无法翻译，例如传送过去的字符串中前 4 个字符的子串'0000'就可有多种译法，或是'AAAA'，或是'ABA'，也可以是'BB'等。因此，若要设计长短不等的编码，则必须是任一个字符的编码都不是另一个字符的编码的前缀，这种编码称做**前缀编码**。

可以利用二叉树来设计二进制的前缀编码。假设有一棵如图 6.25 所示的二叉树，其 4 个叶子结点分别表示 A、B、C、D 这 4 个字符，且约定左分支表示字符'0'，右分支表示字符'1'，则可以从根结点到叶子结点的路径上分支字符组成的字符串作为该叶子结点字符的编码。读者可以证明，如此得到的必为二进制前缀编码。如由图 6.25 所得 A、B、C、D 的二进制前缀编码分别为 0、10、110 和 111。

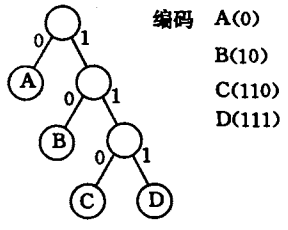


图 6.25　前缀编码示例

又如何得到使电文总长最短的二进制前缀编码呢？假设每种字符在电文中出现的次数为 w_i，其编码长度为 l_i，电文中只有 n 种字符，则电文总长为 $\sum_{i=1}^{n} w_i l_i$。对应到二叉树上，若置 w_i 为叶子结点的权，l_i 恰为从根到叶子的路径长度。则 $\sum_{i=1}^{n} w_i l_i$ 恰为二叉树上带权路径长度。由此可见，设计电文总长最短的二进制前缀编码即为以 n 种字符出现的频率作权，设计一棵赫夫曼树的问题，由此得到的二进制前缀编码便称为赫夫曼编码。

下面讨论具体做法。

由于赫夫曼树中没有度为 1 的结点(这类树又称严格的(strict)(或正则的)二叉树)，

www.TopSage.com

大家网 TopSage.com

则一棵有 n 个叶子结点的赫夫曼树共有 $2n-1$ 个结点，可以存储在一个大小为 $2n-1$ 的一维数组中。如何选定结点结构？由于在构成赫夫曼树之后，为求编码需从叶子结点出发走一条从叶子到根的路径；而为译码需从根出发走一条从根到叶子的路径。则对每个结点而言，既需知双亲的信息，又需知孩子结点的信息。由此，设定下述存储结构：

```
// ----- 赫夫曼树和赫夫曼编码的存储表示 -----
typedef struct {
   unsigned int  weight;
   unsigned int  parent, lchild, rchild;
}HTNode, *HuffmanTree;        // 动态分配数组存储赫夫曼树
typedef char **HuffmanCode; // 动态分配数组存储赫夫曼编码表
```

求赫夫曼编码的算法如算法 6.12 所示。

```
void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n) {
   // w存放n个字符的权值(均>0),构造赫夫曼树HT,并求出n个字符的赫夫曼编码HC。
   if (n<=1) return;
   m = 2 * n - 1;
   HT = (HuffmanTree)malloc((m+1) * sizeof(HTNode));     // 0号单元未用
   for (p=HT, i=1; i<=n; ++i, ++p, ++w)   *p = { *w, 0, 0, 0 };
   for ( ; i<=m; ++i, ++p)   *p = { 0, 0, 0, 0 };
   for (i=n+1; i<=m; ++i) {      // 建赫夫曼树
      // 在HT[1..i-1]选择parent为0且weight最小的两个结点,其序号分别为s1和s2。
      Select(HT, i-1, s1, s2);
      HT[s1].parent = i;  HT[s2].parent = i;
      HT[i].lchild = s1;  HT[i].rchild = s2;
      HT[i].weight = HT[s1].weight + HT[s2].weight;
   }
   // --- 从叶子到根逆向求每个字符的赫夫曼编码 ---
   HC = (HuffmanCode)malloc((n+1)*sizeof(char *));  // 分配n个字符编码的头指针向量
   cd = (char *)malloc(n*sizeof(char));          // 分配求编码的工作空间
   cd[n-1] = "\0";                               // 编码结束符。
   for (i=1; i<=n; ++i) {                        // 逐个字符求赫夫曼编码
      start = n-1;                               // 编码结束符位置
      for (c=i, f=HT[i].parent; f!=0; c=f, f=HT[f].parent) // 从叶子到根逆向求编码
         if (HT[f].lchild == c)  cd[--start] = "0";
         else cd[--start] = "1";
      HC[i] = (char *)malloc((n-start)*sizeof(char)); // 为第i个字符编码分配空间
      strcpy(HC[i], &cd[start]);                 // 从cd复制编码(串)到HC
   }
   free(cd);                  // 释放工作空间
}// HuffanCoding
```

算法 6.12

向量 HT 的前 n 个分量表示叶子结点，最后一个分量表示根结点。各字符的编码长度不

www.TopSage.com

等，所以按实际长度动态分配空间。在算法 6.12 中，求每个字符的赫夫曼编码是从叶子到根逆向处理的。也可以从根出发，遍历整棵赫夫曼树，求得各个叶子结点所表示的字符的赫夫曼编码，如算法 6.13 所示。

```
//------无栈非递归遍历赫夫曼树，求赫夫曼编码
HC = (HuffmanCode)malloc((n+1) * sizeof(char *));
p = m;  cdlen = 0;
for (i=1; i<=m; ++i)  HT[i].weight = 0; // 遍历赫夫曼树时用作结点状态标志
while (p) {
   if (HT[p].weight==0) {                    // 向左
      HT[p].weight = 1;
      if (HT[p].lchild != 0) {p = HT[p].lchild;  cd[cdlen++] = "0"; }
      else if (HT[p].rchild == 0) {          // 登记叶子结点的字符的编码
         HC[p] = (char *)malloc((cdlen+1) * sizeof(char));
         cd[cdlen] = "\0";  strcpy(HC[p], cd);     // 复制编码(串)
      }
   }
   else if (HT[p].weight==1) {               // 向右
      HT[p].weight = 2;
      if (HT[p].rchild != 0) {p = HT[p].rchild;  cd[cdlen++] = "1"; }
   }else {                                   // HT[p].weight==2,退回
    HT[p].weight = 0; p = HT[p].parent; --cdlen;  // 退到父结点，编码长度减 1
   }// else
}// While
```

算法　6.13

译码的过程是分解电文中字符串，从根出发，按字符'0'或'1'确定找左孩子或右孩子，直至叶子结点，便求得该子串相应的字符。具体算法留给读者去完成。

例 6-2　已知某系统在通信联络中只可能出现 8 种字符，其概率分别为 0.05，0.29，0.07，0.08，0.14，0.23，0.03，0.11，试设计赫夫曼编码。

设权 $w=(5,29,7,8,14,23,3,11)$，$n=8$，则 $m=15$，按上述算法可构造一棵赫夫曼树如图 6.26 所示。其存储结构 HT 的初始状态如图 6.27(a)所示，其终结状态如图 6.27(b)所示，所得赫夫曼编码如图 6.27(c)所示。

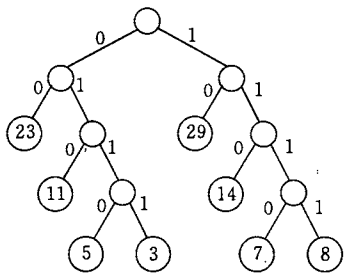


图 6.26　例 6-2 的赫夫曼树

www.TopSage.com

大家网 TopSage.com

HT

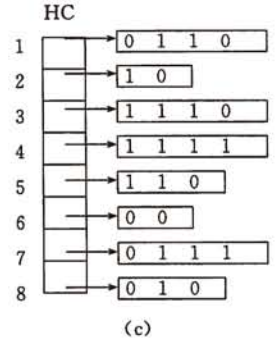
	weight	parent	lchild	rchild
1	5	0	0	0
2	29	0	0	0
3	7	0	0	0
4	8	0	0	0
5	14	0	0	0
6	23	0	0	0
7	3	0	0	0
8	11	0	0	0
9	-	0	0	0
10	-	0	0	0
11	-	0	0	0
12	-	0	0	0
13	-	0	0	0
14	-	0	0	0
15	-	0	0	0

(a)

HT

	weight	parent	lchild	rchild
1	5	9	0	0
2	29	14	0	0
3	7	10	0	0
4	8	10	0	0
5	14	12	0	0
6	23	13	0	0
7	3	9	0	0
8	11	11	0	0
9	8	11	1	7
10	15	12	3	4
11	19	13	8	9
12	29	14	5	10
13	42	15	6	11
14	58	15	2	12
15	100	0	13	14

(b)



(c)

图 6.27 例 6-2 的存储结构
(a) HT 的初态；
(b) HT 的终态；
(c) 赫夫曼编码 HC

6.7 回溯法与树的遍历

在程序设计中，有相当一类求一组解、或求全部解或求最优解的问题，例如读者熟悉的八皇后问题等，不是根据某种确定的计算法则，而是利用试探和**回溯**(Backtracking)的搜索技术求解。回溯法也是设计递归过程的一种重要方法，它的求解过程实质上是一个先序遍历一棵"状态树"的过程，只是这棵树不是遍历前预先建立的，而是隐含在遍历过程中，但如果认识到这点，很多问题的递归过程设计也就迎刃而解了。为了说明问题，先看一个简单例子。

例 6-3 求含 n 个元素的集合的幂集。

集合 A 的幂集是由集合 A 的所有子集所组成的集合。如：$A=\{1,2,3\}$，则 A 的幂集

$$\rho(A) = \{\{1,2,3\},\{1,2\},\{1,3\},\{1\},\{2,3\},\{2\},\{3\},\Phi\} \qquad (6\text{-}6)$$

当然，可以用 5.7 节介绍的分治法来设计这个求幂集的递归过程。在此，从另一角度分析问题。幂集的每个元素是一个集合，它或是空集，或含集合 A 中一个元素，或含集合 A 中两个元素，或等于集合 A。反之，从集合 A 的每个元素来看，它只有两种状态：它或属幂集的元素集，或不属幂集元素集。则求幂集 $\rho(A)$ 的元素的过程可看成是依次对集合

www.TopSage.com

A 中元素进行“取”或“舍(弃)”的过程，并且可以用一棵如图 6.28 所示的二叉树，来表示

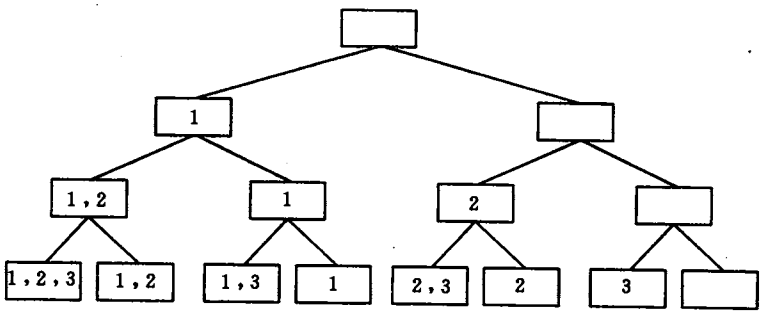


图 6.28　幂集元素在生成过程中的状态图

过程中幂集元素的状态变化状况，树中的根结点表示幂集元素的初始状态(为空集)；叶子结点表示它的终结状态(如图 6.28 中 8 个叶子结点表示式(6-6)中幂集 $\rho(A)$ 的 8 个元素)；而第 $i(i=2,3,\cdots,n-1)$ 层的分支结点，则表示已对集合 A 中前 $i-1$ 个元素进行了取/舍处理的当前状态(左分支表示“取”，右分支表示“舍”)。因此求幂集元素的过程即为先序遍历这棵状态树的过程，如算法 6.14 所描述。

```
void PowerSet(int i, int n) {
  // 求含 n 个元素的集合 A 的幂集ρ (A)。进入函数时已对 A 中前 i-1 个元素作了取舍处理，
  // 现从第 i 个元素起进行取舍处理。若 i>n,则求得幂集的一个元素,并输出之。
  // 初始调用:PowerSet(1, n);
  if (i>n) 输出幂集的一个元素;
  else {  取第 i 个元素;  PowerSet(i+1, n);
          舍第 i 个元素;  PowerSet(i+1, n);
       }
}// PowerSet
```

算法　6.14

对算法6.14求精需确定数据结构。假设以线性表表示集合，则求精后的算法如算法 6.15 所示。

```
void GetPowerSet(int i, List A, List &B) {
  // 线性表 A 表示集合 A,线性表 B 表示幂集ρ (A)的一个元素。
  // 局部量 k 为进入函数时表 B 的当前长度。第一次调用本函数时,B 为空表,i=1。
  if (i>ListLength(A)) Output(B);  // 输出当前 B 值,即ρ (A)的一个元素
  else {GetElem(A, i, x);              k = ListLength(B);
        ListInsert(B, k+1, x);         GetPowerSet(i+1, A, B);
        ListDelete(B, k+1, x);         GetPowerSet(i+1, A, B);
       }
}// GetPowerSet
```

算法　6.15

图 6.28 中的状态变化树是一棵满二叉树，树中每个叶子结点的状态都是求解过程中

www.TopSage.com

大家网 TopSage.com

可能出现的状态(即问题的解)。然而很多问题用回溯和试探求解时,描述求解过程的状态树不是一棵满的多叉树。当试探过程中出现的状态和问题所求解产生矛盾时,不再继续试探下去,这时出现的叶子结点不是问题的解的终结状态。这类问题的求解过程可看成是在约束条件下进行先序(根)遍历,并在遍历过程中剪去那些不满足条件的分支。

例 6-4 求 4 皇后问题的所有合法布局(作为例子,我们将 8 皇后问题简化为 4 皇后问题)。

图 6.29 展示求解过程中棋盘状态的变化情况。这是一棵四叉树,树上每个结点表示

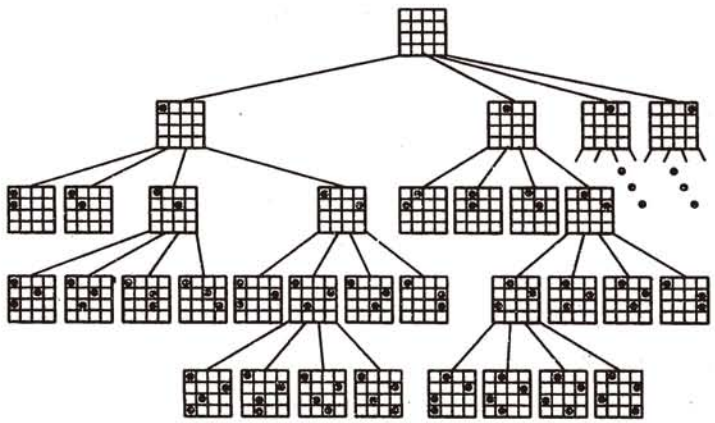


图 6.29 四皇后问题的棋盘状态树

一个局部布局或一个完整的布局。根结点表示棋盘的初始状态:棋盘上无任何棋子。每个(皇后)棋子都有 4 个可选择的位置,但在任何时刻,棋盘的合法布局都必须满足 3 个约束条件,即任何两个棋子都不占据棋盘上的同一行、或者同一列、或者同一对角线。图 6.29中除结点 a 之外的叶子结点都是不合法的布局。

求所有合法布局的过程即为在上述约束条件下先根遍历图 6.29 的状态树的过程。遍历中访问结点的操作为,判别棋盘上是否已得到一个完整的布局(即棋盘上是否已摆上 4 个棋子),若是,则输出该布局;否则依次先根遍历满足约束条件的各棵子树,即首先判断该子树根的布局是否合法,若合法,则先根遍历该子树,否则剪去该子树分支。算法 6.16为求所有合法布局的伪码算法:

```
void Trial(int i, int n) {
  // 进入本函数时,在 n×n 棋盘前 i-1 行已放置了互不攻击①的 i-1 个棋子。
  // 现从第 i 行起继续为后续棋子选择合适①位置。
  // 当 i>n 时,求得一个合法布局,输出之。
  if (i>n) 输出棋盘的当前布局;      // n 为 4 时,即为 4 皇后问题
  else for (j=1; j<=n; ++j) {
          在第 i 行第 j 列放置一个棋子;
          if (当前布局合法) Trial(i+1, n);
```

① 即满足前述的 3 个约束条件。

www.TopSage.com

```
            移走第 i 行第 j 列的棋子;
        }
    }// Trial
```

算法 6.16

算法 6.16 可进一步求精，在此从略。算法 6.16 可作为回溯法求解的一般模式，类似问题有骑士游历、迷宫问题、选最优解问题等等。

6.8 树的计数

本节将讨论的树的计数问题的提法是：具有 n 个结点的不同形态的树有多少棵？下面我们先讨论二叉树的情况，然后可将结果推广到树。

在讨论二叉树的计数之前应先明确两个不同的概念。

称二叉树 T 和 T' 相似是指：二者都为空树或者二者均不为空树，且它们的左右子树分别相似。

称二叉树 T 和 T' 等价是指：二者不仅相似，而且所有对应结点上的数据元素均相同。

二叉树的计数问题就是讨论具有 n 个结点、互不相似的二叉树的数目 b_n。

在 n 值很小的情况下，可直观地得到：$b_0=1$ 为空树；$b_1=1$ 是只有一个根结点的树；$b_2=2$ 和 $b_3=5$，它们的形态分别如图 6.30(a)和图 6.30(b)所示。那么，在 $n>3$ 时又如何呢？

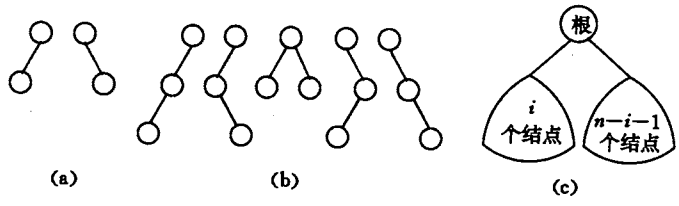


图 6.30 二叉树的形态

(a) $n=2$； (b) $n=3$； (c) 一般情形 $n>1$

一般情况下，一棵具有 $n(n>1)$ 个结点的二叉树可以看成是由一个根结点、一棵具有 i 个结点的左子树和一棵具有 $n-i-1$ 个结点的右子树组成(如图 6.30(c)所示)，其中 $0\leqslant i\leqslant n-1$。由此可得下列递推公式：

$$\begin{cases} b_0 = 1 \\ b_n = \sum_{i=0}^{n-1} b_i \, b_{n-i-1}, \quad n \geqslant 1 \end{cases} \tag{6-7}$$

可以利用生成函数来讨论这个递推公式。

对序列

$$b_0, b_1, \cdots, b_n \cdots$$

定义生成函数

www.TopSage.com

大家网 TopSage.com

$$B(z)=b_0+b_1z+b_2z^2+\cdots+b_nz^n+\cdots$$
$$=\sum_{k=0}^{\infty}b_kz^k \tag{6-8}$$

因为

$$B^2(z)=b_0b_0+(b_0b_1+b_1b_0)z+(b_0b_2+b_1b_1+b_2b_0)z^2+\cdots$$
$$=\sum_{p=0}^{\infty}\Big(\sum_{i=0}^{p}b_ib_{p-i}\Big)z^p$$

根据(6-7)

$$B^2(z)=\sum_{p=0}^{\infty}b_{p+1}z^p \tag{6-9}$$

由此得

$$zB^2(z)=B(z)-1$$

即

$$zB^2(z)-B(z)+1=0$$

解此二次方程得

$$B(z)=\frac{1\pm\sqrt{1-4z}}{2z}$$

由初值 $b_0=1$，应有 $\lim\limits_{z\to0}B(z)=b_0=1$

所以

$$B(z)=\frac{1-\sqrt{1-4z}}{2z}$$

利用二项式展开

$$(1-4z)^{\frac{1}{2}}=\sum_{k=0}^{\infty}\binom{\frac{1}{2}}{k}(-4z)^k \tag{6-10}$$

当 $k=0$ 时，式(6-10)的第一项为 1，故有

$$B(z)=\frac{1}{2}\sum_{k=1}^{\infty}\binom{\frac{1}{2}}{k}(-1)^{k-1}2^{2k}z^{k-1}$$
$$=\sum_{m=0}^{\infty}\binom{\frac{1}{2}}{m+1}(-1)^m2^{2m+1}z^m$$
$$=1+z+2z^2+5z^3+14z^4+42z^5+\cdots \tag{6-11}$$

对照(6-8)和(6-11)而得

$$b_n=\binom{\frac{1}{2}}{n+1}(-1)^n2^{2n+1}$$
$$=\frac{\frac{1}{2}\left(\frac{1}{2}-1\right)\left(\frac{1}{2}-2\right)\cdots\left(\frac{1}{2}-n\right)}{(n+1)!}(-1)^n2^{2n+1}$$

www.TopSage.com

$$b_n = \frac{1}{n+1} \cdot \frac{(2n)!}{n!n!} = \frac{1}{n+1} C_{2n}^n \tag{6-12}$$

因此，含有 n 个结点的不相似的二叉树有 $\frac{1}{n+1}C_{2n}^n$ 棵。

我们还可以从另一个角度来讨论这个问题。从二叉树的遍历已经知道，任意一棵二叉树结点的前序序列和中序序列是惟一的。反过来，给定结点的前序序列和中序序列，能否确定一棵二叉树呢？又是否惟一呢？

由定义，二叉树的前序遍历是先访问根结点 D，其次遍历左子树 L，最后遍历右子树 R。即在结点的前序序列中，第一个结点必是根 D；而另一方面，由于中序遍历是先遍历左子树 L，然后访问根 D，最后遍历右子树 R，则根结点 D 将中序序列分割成两部分：在 D 之前是左子树结点的中序序列，在 D 之后是右子树结点的中序序列。反过来，根据左子树的中序序列中结点个数，又可将前序序列除根以外分成左子树的前序序列和右子树的前序序列两部分。依次类推，便可递归得到整棵二叉树。

例 6-5 已知结点的前序序列和中序序列分别为：

前序序列：A B C D E F G

中序序列：C B E D A F G

则可按上述分解求得整棵二叉树。其构造过程如图 6.31 所示。首先由前序序列得知二

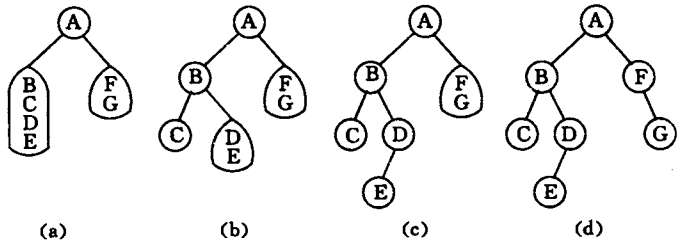


图 6.31 由前序和中序序列构造一棵二叉树的过程

叉树的根为 A，则其左子树的中序序列为(CBED)，右子树的中序序列为(FG)。反过来得知其左子树的前序序列必为(BCDE)，右子树的前序序列为(FG)。类似地，可由左子树的前序序列和中序序列构造得 A 的左子树，由右子树的前序序列和中序序列构造得 A 的右子树。

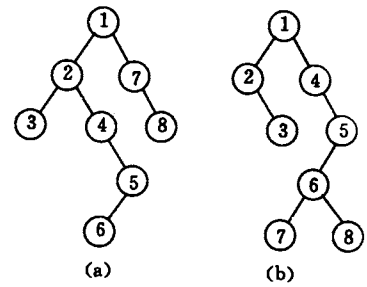


图 6.32 具有不同中序序列的二叉树

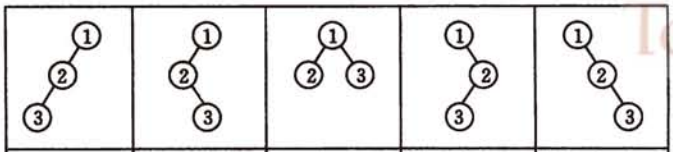
上述构造过程说明了给定结点的前序序列和中序序列，可确定一棵二叉树。至于它的惟一性，读者可试用归纳法证明之。

我们可由此结论来推论具有 n 个结点的不同形态的二叉树的数目。

假设对二叉树的 n 个结点从 1 到 n 加以编号，且令其前序序列为 $1,2,\cdots,n$，则由前面的讨论可知，不同的二叉树所得中序序列不同。如图 6.32 所示两棵有 8 个结点的二叉树，它们的前序序列都是 12345678，而(a)树的中序序列为 32465178，(b)树的中序序列为

www.TopSage.com

大家网
TopSage.com



栈状态	访问	栈状态	访问	栈状态	访问	栈状态	访问	栈状态	访问
空		空		空		空		空	
1		1		1		1		1	
1 2		1 2		1 2		空	1	空	1
1 2 3		1	2	1	2	2		2	
1 2	3	1 3		空	1	2 3		空	2
1	2	1	3	3		2	3	3	
空	1	空	1	空	3	空	2	空	3

图 6.33　中序遍历时进栈和出栈的过程

23147685。因此，不同形态的二叉树的数目恰好是前序序列均为 12…n 的二叉树所能得到的中序序列的数目。而中序遍历的过程实质上是一个结点进栈和出栈的过程。二叉树的形态确定了其结点进栈和出栈的顺序，也确定了其结点的中序序列。例如图 6.33 中所示为 $n=3$ 时不同形态的二叉树在中序遍历时栈的状态和访问结点次序的关系。由此，由前序序列 12…n 所能得到的中序序列的数目恰为数列 12…n 按不同顺序进栈和出栈所能得到的排列的数目。这个数目为[①]

$$C_{2n}^{n}-C_{2n}^{n-1}=\frac{1}{n+1}C_{2n}^{n} \qquad (6\text{-}13)$$

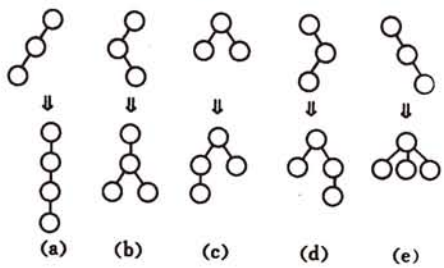


图 6.34　具有不同形态的树和二叉树

由二叉树的计数可推得树的计数。由“6.4.2 森林与二叉树的转换”中可知一棵树可转换成惟一的一棵没有右子树的二叉树，反之亦然。则具有 n 个结点有不同形态的树的数目 t_n 和具有 $n-1$ 个结点互不相似的二叉树的数目相同。即 $t_n=b_{n-1}$。图 6.34 展示了具有 4 个结点的树和具有 3 个结点的二叉树的关系。从图中可见，在此讨论树的计数是指有序树，因此(c)和(d)是两棵有不同形态的树(在无序树中，它们被认为是相同的)。

① 参考书目[3]中译本第 457 页。

www.TopSage.com

第7章 图

图(Graph)是一种较线性表和树更为复杂的数据结构。在线性表中,数据元素之间仅有线性关系,每个数据元素只有一个直接前驱和一个直接后继;在树形结构中,数据元素之间有着明显的层次关系,并且每一层上的数据元素可能和下一层中多个元素(即其孩子结点)相关,但只能和上一层中一个元素(即其双亲结点)相关;而在图形结构中,结点之间的关系可以是任意的,图中任意两个数据元素之间都可能相关。由此,图的应用极为广泛,特别是近年来的迅速发展,已渗入到诸如语言学、逻辑学、物理、化学、电讯工程、计算机科学以及数学的其他分支中。

读者在"离散数学"课程中已学习了图的理论,在此仅应用图论的知识讨论如何在计算机上实现图的操作,因此主要学习图的存储结构以及若干图的操作的实现。

7.1 图的定义和术语

图是一种数据结构,加上一组基本操作,就构成了抽象数据类型。抽象数据类型图的定义如下:

ADT Graph {

数据对象 V:V 是具有相同特性的数据元素的集合,称为顶点集。

数据关系 R:

R = {VR}

VR = {<v,w> | v,w∈V 且 P(v,w),<v,w>表示从 v 到 w 的弧,
谓词 P(v,w)定义了弧<v,w>的意义或信息 }

基本操作 P:

CreateGraph(&G,V,VR);

初始条件:V 是图的顶点集,VR 是图中弧的集合。

操作结果:按 V 和 VR 的定义构造图 G。

DestroyGraph(&G);

初始条件:图 G 存在。

操作结果:销毁图 G。

LocateVex(G, u);

初始条件:图 G 存在,u 和 G 中顶点有相同特征。

操作结果:若 G 中存在顶点 u,则返回该顶点在图中位置;否则返回其他信息。

GetVex(G, v);

初始条件:图 G 存在,v 是 G 中某个顶点。

操作结果:返回 v 的值。

PutVex(&G, v, value);

初始条件:图 G 存在,v 是 G 中某个顶点。

操作结果:对 v 赋值 value。

www.TopSage.com

大家网
TopSage.com

FirstAdjVex(G, v);
初始条件:图 G 存在,v 是 G 中某个顶点。
操作结果:返回 v 的第一个邻接顶点。若顶点在 G 中没有邻接顶点,则返回"空"。
NextAdjVex(G, v, w);
初始条件:图 G 存在,v 是 G 中某个顶点,w 是 v 的邻接顶点。
操作结果:返回 v 的(相对于 w 的)下一个邻接顶点。若 w 是 v 的最后一个邻接点,则返回"空"。
InsertVex(&G, v);
初始条件:图 G 存在,v 和图中顶点有相同特征。
操作结果:在图 G 中增添新顶点 v。
DeleteVex(&G, v);
初始条件:图 G 存在,v 是 G 中某个顶点。
操作结果:删除 G 中顶点 v 及其相关的弧。
InsertArc(&G, v, w);
初始条件:图 G 存在,v 和 w 是 G 中两个顶点。
操作结果:在 G 中增添弧<v,w>,若 G 是无向的,则还增添对称弧<w,v>。
DeleteArc(&G, v, w);
初始条件:图 G 存在,v 和 w 是 G 中两个顶点。
操作结果:在 G 中删除弧<v,w>,若 G 是无向的,则还删除对称弧<w,v>。
DFSTraverse(G, Visit());
初始条件:图 G 存在,Visit 是顶点的应用函数。
操作结果:对图进行深度优先遍历。在遍历过程中对每个顶点调用函数 Visit 一次且仅一次。一旦 visit()失败,则操作失败。
BFSTraverse(G, Visit());
初始条件:图 G 存在,Visit 是顶点的应用函数。
操作结果:对图进行广度优先遍历。在遍历过程中对每个顶点调用函数 Visit 一次且仅一次。一旦 visit()失败,则操作失败。

}**ADT** Graph

在图中的数据元素通常称做**顶点**(Vertex),V 是顶点的有穷非空集合;VR 是两个顶点之间的关系的集合。若$\langle v,w\rangle \in VR$,则$\langle v,w\rangle$表示从 v 到 w 的一条**弧**(Arc),且称 v 为**弧尾**(Tail)或初始点(Initial node),称 w 为**弧头**(Head)或终端点(Terminal node),此时的图称为**有向图**(Digraph)。若$\langle v,w\rangle \in VR$ 必有$\langle w,v\rangle \in VR$,即 VR 是对称的,则以无序对(v,w)代替这两个有序对,表示 v 和 w 之间的一条**边**(Edge),此时的图称为**无向图**(Undigraph)。例如图7.1(a)中 G_1 是有向图,定义此图的谓词 $P(v,w)$则表示从 v 到 w 的一条单向通路。

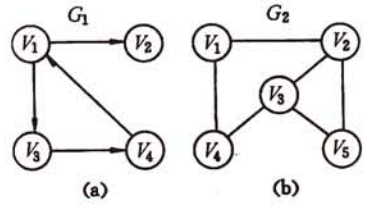


图 7.1 图的示例
(a) 有向图 G_1;(b) 无向图 G_2

$$G_1 = (V_1, \{A_1\})$$

其中:

$$V_1 = \{v_1, v_2, v_3, v_4\}$$
$$A_1 = \{\langle v_1, v_2\rangle, \langle v_1, v_3\rangle, \langle v_3, v_4\rangle, \langle v_4, v_1\rangle\}$$

www.TopSage.com

图 7.1(b)中 G_2 为无向图。

$$G_2 = (V_2\{E_2\})$$

其中 $V_2=\{v_1,v_2,v_3,v_4,v_5\}$

$E_2=\{(v_1,v_2),(v_1,v_4)(v_2,v_3),(v_2,v_5),(v_3,v_4),(v_3,v_5)\}$

我们用 n 表示图中顶点数目，用 e 表示边或弧的数目。在下面的讨论中，我们不考虑顶点到其自身的弧或边，即若$\langle v_i,v_j\rangle\in VR$，则 $v_i\neq v_j$，那么，对于无向图，e 的取值范围是 0 到$\frac{1}{2}n(n-1)$。有$\frac{1}{2}n(n-1)$条边的无向图称为**完全图**(Completed graph)。对于有向图，e 的取值范围是 0 到 $n(n-1)$。具有 $n(n-1)$条弧的有向图称为**有向完全图**。有很少条边或弧(如 $e<n\log n$)的图称为**稀疏图**(Sparse graph)，反之称为**稠密图**(Dense graph)。

有时图的边或弧具有与它相关的数，这种与图的边或弧相关的数叫做**权**(Weight)。这些权可以表示从一个顶点到另一个顶点的距离或耗费。这种带权的图通常称为**网**(Network)。

假设有两个图 $G=(V,\{E\})$和 $G'=(V',\{E'\})$，如果 $V'\subseteq V$ 且 $E'\subseteq E$，则称 G'为 G 的**子图**(Subgraph)。例如，图 7.2 是子图的一些例子。

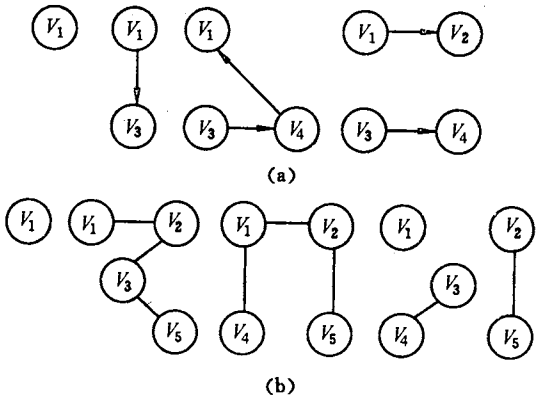


图 7.2　子图示例

(a) G_1 的子图；(b) G_2 的子图

对于无向图 $G=(V,\{E\})$，如果边$(v,v')\in E$，则称顶点 v 和 v'互为**邻接点**(Adjacent)，即 v 和 v'相邻接。边(v,v')**依附**(Incident)于顶点 v 和 v'，或者说(v,v')和顶点 v 和 v'**相关联**。顶点 v 的**度**(Degree)是和 v 相关联的边的数目，记为 $TD(V)$。例如，G_2 中顶点 v_3 的度是 3。对于有向图 $G=(V,\{A\})$，如果弧$\langle v,v'\rangle\in A$，则称顶点 v 邻接**到**顶点 v'，顶点 v'邻接**自**顶点 v。弧$\langle v,v'\rangle$和顶点 v,v'相关联。以顶点 v 为头的弧的数目称为 v 的**入度**(InDegree)，记为 $ID(v)$；以 v 为尾的弧的数目称为 v 的**出度**(Outdegree)，记为 $OD(v)$；顶点 v 的度为 $TD(v)=ID(v)+OD(v)$。例如，图 G_1 中顶点 v_1 的入度$ID(v_1)=1$，出度 $OD(v_1)=2$，度 $TD(v_1)=ID(v_1)+OD(v_1)=3$。一般地，如果顶点 v_i 的度记为 $TD(v_i)$，那么一个有 n 个顶点，e 条边或弧的图，满足如下关系

$$e=\frac{1}{2}\sum_{i=1}^{n}TD(v_i)$$

www.TopSage.com

大家网 TopSage.com

无向图 $G=(V,\{E\})$中从顶点 v 到顶点 v'的**路径**(Path)是一个顶点序列($v=v_{i,0}$, $v_{i,1},\cdots,v_{i,m}=v'$),其中$(v_{i,j-1},v_{i,j})\in E,1\leqslant j\leqslant m$。如果 G 是有向图,则路径也是有向的,顶点序列应满足$\langle v_{i,j-1},v_{i,j}\rangle\in E,1\leqslant j\leqslant m$。路径的长度是路径上的边或弧的数目。第一个顶点和最后一个顶点相同的路径称为**回路**或**环**(Cycle)。序列中顶点不重复出现的路径称为**简单路径**。除了第一个顶点和最后一个顶点之外,其余顶点不重复出现的回路,称为简单回路或简单环。

在无向图 G 中,如果从顶点 v 到顶点 v'有路径,则称 v 和 v'是**连通**的。如果对于图中任意两个顶点 v_i、$v_j\in V$,v_i 和 v_j 都是连通的,则称 G 是**连通图**(Connected Graph)。图 7.1(b)中的 G_2 就是一个连通图,而图 7.3(a)中的 G_3 则是非连通图,但 G_3 有 3 个连通分量,如图 7.3(b)所示。所谓**连通分量**(Connected Component),指的是无向图中的极大连通子图。

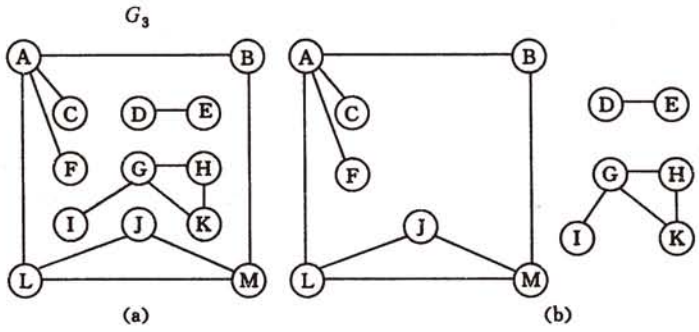


图 7.3　无向图及其连通分量

(a) 无向图 G_3;　(b) G_3 的 3 个连通分量

在有向图 G 中,如果对于每一对 $v_i,v_j\in V,v_i\neq v_j$,从 v_i 到 v_j 和从 v_j 到 v_i 都存在路径,则称 G 是**强连通图**。有向图中的极大强连通子图称做有向图的**强连通分量**。例如图 7.1(a)中的 G_1 不是强连通图,但它有两个强连通分量,如图 7.4 所示。

一个连通图的**生成树**是一个极小连通子图,它含有图中全部顶点,但只有足以构成一棵树的 $n-1$ 条边。图 7.5 是 G_3 中最大连通分量的一棵生成树。如果在一棵生成树上添加一条边,必定构成一个环,因为这条边使得它依附的那两个顶点之间有了第二条路径。

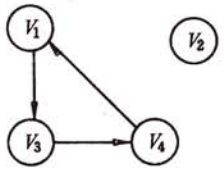


图 7.4　G_1 的两个强连通分量

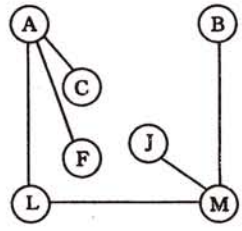


图 7.5　G_3 的最大连通分量的一棵生成树

一棵有 n 个顶点的生成树有且仅有 $n-1$ 条边。如果一个图有 n 个顶点和小于 $n-1$ 条边,则是非连通图。如果它多于 $n-1$ 条边,则一定有环。但是,有 $n-1$ 条边的图不一定是生成树。

www.TopSage.com

如果一个有向图恰有一个顶点的入度为 0，其余顶点的入度均为 1，则是一棵有向树。一个有向图的**生成森林**由若干棵有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的弧。图 7.6 所示为其一例。

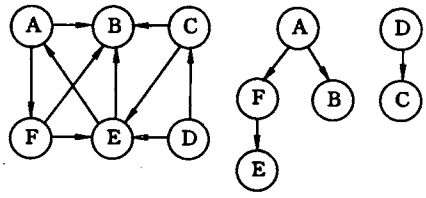


图 7.6　一个有向图及其生成森林

在前述图的基本操作的定义中，关于“顶点的位置”和“邻接点的位置”只是一个相对的概念。因为，从图的逻辑结构的定义来看，图中的顶点之间不存在全序的关系（即无法将图中顶点排列成一个线性序列），任何一个顶点都可被看成是第一个顶点；另一方面，任一顶点的邻接点之间也不存在次序关系。但为了操作方便，我们需要将图中顶点按任意的顺序排列起来（这个排列和关系 *VR* 无关）。由此，所谓“顶点在图中的位置”指的是该顶点在这个人为的随意排列中的位置（或序号）。同理，可对某个顶点的所有邻接点进行排队，在这个排队中自然形成了第一个或第 k 个邻接点。若某个顶点的邻接点的个数大于 k，则称第 $k+1$ 个邻接点为第 k 个邻接点的下一个邻接点，而最后一个邻接点的下一个邻接点为“空”。

7.2　图的存储结构

在前面几章讨论的数据结构中，除了广义表和树以外，都可以有两类不同的存储结构，它们是由不同的映像方法（顺序映像和链式映像）得到的。由于图的结构比较复杂，任意两个顶点之间都可能存在联系，因此无法以数据元素在存储区中的物理位置来表示元素之间的关系，即图没有顺序映像的存储结构，但可以借助数组的数据类型表示元素之间的关系。另一方面，用多重链表表示图是自然的事，它是一种最简单的链式映像结构，即以一个由一个数据域和多个指针域组成的结点表示图中一个顶点，其中数据域存储该顶点的信息，指针域存储指向其邻接点的指针，如图 7.7 所示为图 7.1 中有向图 G_1 和无向图 G_2 的多重链表。但是，由于图中各个结点的度数各不相同，最大度数和最小度数可能相差很多，因此，若按度数最大的顶点设计结点结构，则会浪费很多存储单元；反之，若按每个顶点自己的度数设计不同的结点结构，又会给操作带

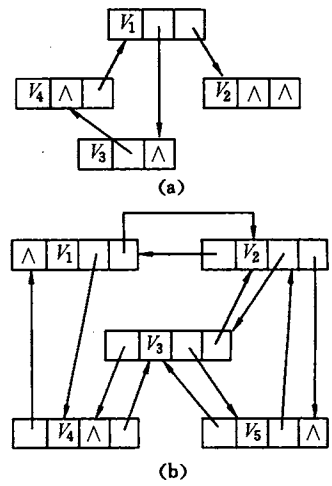


图 7.7　图的多重链表

(a) G_1 的多重链表；(b) G_2 的多重链表

www.TopSage.com

大家网 TopSage.com

来不便。因此，和树类似，在实际应用中不宜采用这种结构，而应根据具体的图和需要进行的操作，设计恰当的结点结构和表结构。常用的有邻接表、邻接多重表和十字链表。下面分别讨论。

7.2.1 数组表示法

用两个数组分别存储数据元素（顶点）的信息和数据元素之间的关系（边或弧）的信息。其形式描述如下：

```
//------ 图的数组(邻接矩阵)存储表示 ------
#define INFINITY          INT_MAX              // 最大值∞
#define MAX_VERTEX_NUM    20                   // 最大顶点个数
typedef enum {DG, DN, UDG, UDN} GraphKind;     //{有向图,有向网,无向图,无向网}
typedef struct ArcCell {
   VRType       adj;  // VRType 是顶点关系类型。对无权图,用 1 或 0
                      // 表示相邻否;对带权图,则为权值类型。
   InfoType   * info; // 该弧相关信息的指针
}ArcCell,  AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
typedef struct {
   VertexType  vexs[MAX_VERTEX_NUM];          // 顶点向量
   AdjMatrix   arcs;                          // 邻接矩阵
   int         vexnum, arcnum;                // 图的当前顶点数和弧数
   GraphKind   kind;                          // 图的种类标志
}MGraph;
```

例如，图 7.1 中 G_1 和 G_2 的邻接矩阵如图 7.8 所示。以二维数组表示有 n 个顶点的图时，需存放 n 个顶点信息和 n^2 个弧信息的存储量。若考虑无向图的邻接矩阵的对称性，则可采用压缩存储的方式只存入矩阵的下三角（或上三角）元素。

$$G1.arcs=\begin{bmatrix}0&1&1&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{bmatrix},\quad G2.arcs=\begin{bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{bmatrix}$$

图 7.8 图的邻接矩阵

借助于邻接矩阵容易判定任意两个顶点之间是否有边（或弧）相连，并容易求得各个顶点的度。对于无向图，顶点 v_i 的度是邻接矩阵中第 i 行（或第 i 列）的元素之和，即

$$TD(v_i)=\sum_{j=0}^{n-1}A[i][j](n=MAX_VERTEX_NUM)$$

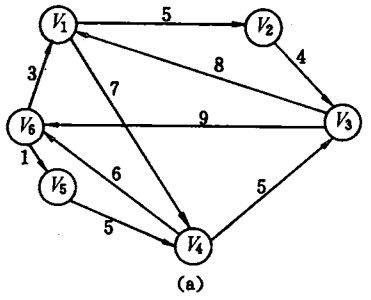
对于有向图，第 i 行的元素之和为顶点 v_i 的出度 $OD(v_i)$，第 j 列的元素之和为顶点 v_j 的入度 $ID(v_j)$。

网的邻接矩阵可定义为

$$A[i][j]=\begin{cases}w_{i,j} & \text{若}\langle v_i,v_j\rangle\text{或}(v_i,v_j)\in VR\\ \infty & \text{反之}\end{cases}$$

例如，图 7.9 列出了一个有向网和它的邻接矩阵。

www.TopSage.com



$$\begin{bmatrix} \infty & 5 & \infty & 7 & \infty & \infty \\ \infty & \infty & 4 & \infty & \infty & \infty \\ 8 & \infty & \infty & \infty & \infty & 9 \\ \infty & \infty & 5 & \infty & \infty & 6 \\ \infty & \infty & \infty & 5 & \infty & \infty \\ 3 & \infty & \infty & \infty & 1 & \infty \end{bmatrix}$$

(b)

图 7.9　网及其邻接矩阵

(a) 网 N；(b) 邻接矩阵

算法 7.1 是在邻接矩阵存储结构 MGraph 上对图的构造操作的实现框架，它根据图 G 的种类调用具体构造算法。如果 G 是无向网，则调用算法 7.2。构造一个具有 n 个顶点和 e 条边的无向网 G 的时间复杂度是 $O(n^2+e\cdot n)$，其中对邻接矩阵 G.arcs 的初始化耗费了 $O(n^2)$ 的时间。

```
Status CreateGraph( MGraph &G ) {
    // 采用数组(邻接矩阵)表示法,构造图 G。
    scanf(&G.kind);
    switch (G.kind) {
      case  DG: return CreateDG(G);    // 构造有向图 G
      case  DN: return CreateDN(G);    // 构造有向网 G
      case UDG: return CreateUDG(G);   // 构造无向图 G
      case UDN: return CreateUDN(G);   // 构造无向网 G
      default : return ERROR;
    }
}// CreateGraph
```

算法　7.1

```
Status CreateUDN(MGraph &G) {
    // 采用数组(邻接矩阵)表示法,构造无向网 G。
    scanf(&G.vexnum, &G.arcnum, &IncInfo);     // IncInfo 为 0 则各弧不含其他信息
    for (i=0; i<G.vexnum; ++i)  scanf (&G.vexs[i]);  // 构造顶点向量
    for (i=0; i<G.vexnum; ++i)                 // 初始化邻接矩阵
      for (j=0; j<G.vexnum; ++j)  G.arcs[i][j] = {INFINITY, NULL};   //{adj,info}
    for (k=0; k<G.arcnum; ++k) {               // 构造邻接矩阵
      scanf(&v1, &v2, &w);                     // 输入一条边依附的顶点及权值
      i = LocateVex(G, v1);  j = LocateVex(G, v2);  // 确定 v1 和 v2 在 G 中位置
      G.arcs[i][j].adj = w;                    // 弧<v1,v2>的权值
      if (IncInfo) Input( *G.arcs[i][j].info);   // 若弧含有相关信息,则输入
      G.arcs[j][i] = G.arcs[i][j];             // 置<v1,v2>的对称弧<v2,v1>
    }
    return OK;
}// CreateUDN
```

算法　7.2

www.TopSage.com

大家网 TopSage.com

在这个存储结构上也易于实现 7.2 节所列的基本操作。如，FIRST_ADJ(G,v)找 v 的第一个邻接点。首先，由 $LOC_VERTEX(G,v)$找到 v 在图 G 中的位置，即 v 在一维数组 vexs 中的序号 i，则二维数组 arcs 中第 i 行上第一个 adj 域的值为“1”的分量所在列号 j，便为 v 的第一个邻接点在图 G 中的位置。同理，下一个邻接点在图 G 中的位置便为 j 列之后第一个 adj 域的值为“1”的分量所在列号。

7.2.2 邻接表

邻接表(Adjacency List) 是图的一种链式存储结构。在邻接表中，对图中每个顶点建立一个单链表，第 i 个单链表中的结点表示依附于顶点 v_i 的边(对有向图是以顶点 v_i 为尾的弧)。每个结点由 3 个域组成，其中邻接点域(adjvex)指示与顶点 v_i 邻接的点在图中的位置，链域(nextarc)指示下一条边或弧的结点；数据域(info)存储和边或弧相关的信息，如权值等。每个链表上附设一个表头结点。在表头结点中，除了设有链域(firstarc)指向链表中第一个结点之外，还设有存储顶点 v_i 的名或其他有关信息的数据域(data)。如下图所示



这些表头结点(可以链相接)通常以顺序结构的形式存储，以便随机访问任一顶点的链表。例如图 7.10(a)和(b)所示分别为图 7.1 中 G_1 和 G_2 的邻接表。一个图的邻接表存储结构可形式地说明如下：

```
//------ 图的邻接表存储表示 ------
#define MAX_VERTEX_NUM  20
typedef struct ArcNode {
   int                adjvex;        // 该弧所指向的顶点的位置
   struct ArcNode   *nextarc;        // 指向下一条弧的指针
   InfoType          *info;          // 该弧相关信息的指针
}ArcNode;
typedef struct VNode {
   VertexType  data;                 // 顶点信息
   ArcNode     *firstarc;            // 指向第一条依附该顶点的弧的指针
}VNode, AdjList[MAX_VERTEX_NUM];
typedef struct {
   AdjList  vertices;
   int      vexnum, arcnum;          // 图的当前顶点数和弧数
   int      kind;                    // 图的种类标志
}ALGraph;
```

若无向图中有 n 个顶点、e 条边，则它的邻接表需 n 个头结点和 $2e$ 个表结点。显然，在边稀疏$\left(e \ll \frac{n(n-1)}{2}\right)$的情况下，用邻接表表示图比邻接矩阵节省存储空间，当和边相关的信息较多时更是如此。

www.TopSage.com

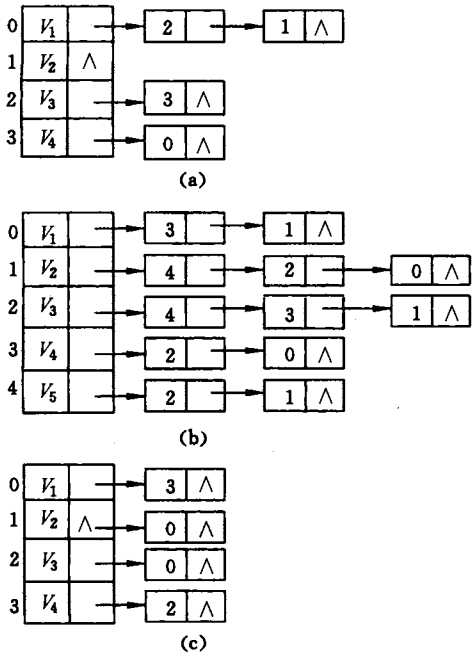


图 7.10 邻接表和逆邻接表

(a) G_1 的邻接表； (b) G_2 的邻接表； (c) G_1 的逆邻接表

在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中的结点数；而在有向图中，第 i 个链表中的结点个数只是顶点 v_i 的出度，为求入度，必须遍历整个邻接表。在所有链表中其邻接点域的值为 i 的结点的个数是顶点 v_i 的入度。有时，为了便于确定顶点的入度或以顶点 v_i 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 v_i 建立一个链接以 v_i 为头的弧的表，例如图 7.10(c)所示为有向图 G_1 的逆邻接表。

在建立邻接表或逆邻接表时，若输入的顶点信息即为顶点的编号，则建立邻接表的时间复杂度为 $O(n+e)$，否则，需要通过查找才能得到顶点在图中位置，则时间复杂度为 $O(n \cdot e)$。

在邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，但要判定任意两个顶点(v_i 和 v_j)之间是否有边或弧相连，则需搜索第 i 个或第 j 个链表，因此，不及邻接矩阵方便。

7.2.3 十字链表

十字链表(Orthogonal List)是有向图的另一种链式存储结构。可以看成是将有向图的邻接表和逆邻接表结合起来得到的一种链表。在十字链表中，对应于有向图中每一条弧有一个结点，对应于每个顶点也有一个结点。这些结点的结构如下所示：

弧结点

tailvex	headvex	hlink	tlink	info

顶点结点

data	firstin	firstout

在弧结点中有 5 个域：其中尾域(tailvex)和头域(headvex)分别指示弧尾和弧头这两个顶点在图中的位置，链域 hlink 指向弧头相同的下一条弧，而链域 tlink 指向弧尾相同的下一条弧，info 域指向该弧的相关信息。弧头相同的弧在同一链表上，弧尾相同的弧也在同一链表上。它们的头结点即为顶点结点，它由 3 个域组成：其中 data 域存储和顶点相关的信息，如顶点的名称等；firstin 和 firstout 为两个链域，分别指向以该顶点为弧头或弧尾的第一个弧结点。例如，图 7.11(a)中所示图的十字链表如图 7.11(b)所示。若将有向图的邻接矩阵看成是稀疏矩阵的话，则十字链表也可以看成是邻接矩阵的链表存储结构，在图的十字链表中，弧结点所在的链表非循环链表，结点之间相对位置自然形成，不一定按顶点序号有序，表头结点即顶点结点，它们之间不是链接，而是顺序存储。

www.TopSage.com

大家网 TopSage.com

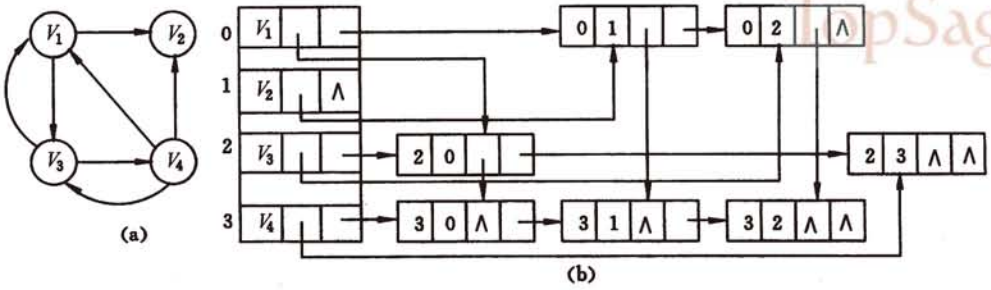


图 7.11　有向图的十字链表

有向图的十字链表存储表示的形式说明如下所示：

```
// - - - - - 有向图的十字链表存储表示 - - - - -
#define MAX_VERTEX_NUM  20
typedef struct ArcBox {
   int              tailvex, headvex; // 该弧的尾和头顶点的位置
   struct ArcBox   *hlink, *tlink;    // 分别为弧头相同和弧尾相同的弧的链域
   InfoType        *info;             // 该弧相关信息的指针
}ArcBox;
typedef struct VexNode {
   VertexType  data;
   ArcBox      *firstin, *firstout; // 分别指向该顶点第一条入弧和出弧
}VexNode;
typedef struct {
   VexNode  xlist[MAX_VERTEX_NUM]; // 表头向量
   int      vexnum, arcnum;        // 有向图的当前顶点数和弧数
}OLGraph;
```

只要输入 n 个顶点的信息和 e 条弧的信息，便可建立该有向图的十字链表，其算法如算法 7.3 所示。

```
Status CreateDG(OLGraph &G) {
  // 采用十字链表存储表示，构造有向图 G(G.kind = DG)。
  scanf(&G.vexnum, &G.arcnum, &IncInfo);          // IncInfo 为 0 则各弧不含其他信息
  for (i = 0; i<G.vexnum; ++i) {                  // 构造表头向量
    scanf(&G.xlist[i].data);                      // 输入顶点值
    G.xlist[i].firstin = NULL; G.xlist[i].firstout = NULL;  // 初始化指针
  }
  for (k = 0; k<G.arcnum; ++k) {                  // 输入各弧并构造十字链表
    scanf(&v1, &v2);                              // 输入一条弧的始点和终点
    i = LocateVex(G, v1);  j = LocateVex(G, v2);  // 确定 v1 和 v2 在 G 中位置
    p = (ArcBox *) malloc (sizeof (ArcBox));      // 假定有足够空间
     *p = {i, j, G.xlist[j].firstin, G.xlist[i].firstout, NULL}   // 对弧结点赋值
      // {tailvex, headvex, hlink, tlink, info}
    G.xlist[j].firstin = G.xlist[i].firstout = p;  // 完成在入弧和出弧链头的插入
```


www.TopSage.com

```
    if (IncInfo) Input( * p->info);                    // 若弧含有相关信息,则输入
  }
}// CreateDG
```

算法 7.3

在十字链表中既容易找到以 v_i 为尾的弧,也容易找到以 v_i 为头的弧,因而容易求得顶点的出度和入度(或需要,可在建立十字链表的同时求出)。同时,由算法 7.3 可知,建立十字链表的时间复杂度和建立邻接表是相同的。在某些有向图的应用中,十字链表是很有用的工具。

7.2.4 邻接多重表

邻接多重表(Adjacency Multilist)是无向图的另一种链式存储结构。虽然邻接表是无向图的一种很有效的存储结构,在邻接表中容易求得顶点和边的各种信息。但是,在邻接表中每一条边(v_i, v_j)有两个结点,分别在第 i 个和第 j 个链表中,这给某些图的操作带来不便。例如在某些图的应用问题中需要对边进行某种操作,如对已被搜索过的边作记号或删除一条边等,此时需要找到表示同一条边的两个结点。因此,在进行这一类操作的无向图的问题中采用邻接多重表作存储结构更为适宜。

邻接多重表的结构和十字链表类似。在邻接多重表中,每一条边用一个结点表示,它由如下所示的 6 个域组成:

mark	ivex	ilink	jvex	jlink	info

其中,mark 为标志域,可用以标记该条边是否被搜索过;ivex 和 jvex 为该边依附的两个顶点在图中的位置;ilink 指向下一条依附于顶点 ivex 的边;jlink 指向下一条依附于顶点 jvex 的边,info 为指向和边相关的各种信息的指针域。每一个顶点也用一个结点表示,它由如下所示的两个域组成:

data	firstedge

其中,data 域存储和该顶点相关的信息,firstedge 域指示第一条依附于该顶点的边。例如,图 7.12 所示为无向图 G_2 的邻接多重表。在邻接多重表中,所有依附于同一顶点的边串联在同一链表中,由于每条边依附于两个顶点,则每个边结点同时链接在两个链表中。可见,对无向图而言,其邻接多重表和邻接表的差别,仅仅在于同一条边在邻接表中用两个结点表示,而在邻接多重表中只有一个结点。因此,除了在边结点中增加一个标志域外,邻接多重表所需的存储量和邻接表相同。在邻接多重表上,各种基本操作的实现亦和邻接表相似。邻接多重表的类型说明如下:

```
// - - - - - 无向图的邻接多重表存储表示 - - - - -
#define MAX_VERTEX_NUM  20
typedef emnu {unvisited, visited} VisitIf;
typedef struct EBox {
  VisitIf      mark;          // 访问标记
```


www.TopSage.com

大家网
TopSage.com

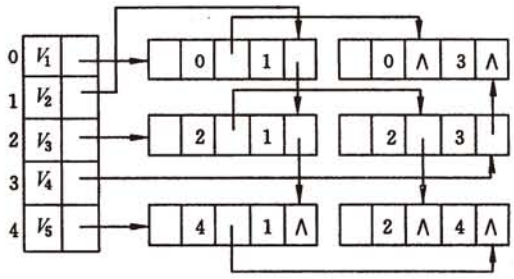


图 7.12　无向图 G_2 的邻接多重表

```
    int            ivex, jvex;      // 该边依附的两个顶点的位置
    struct EBox   *ilink, *jlink;// 分别指向依附这两个顶点的下一条边
    InfoType      *info;           // 该边信息指针
}EBox;
typedef struct VexBox {
    VertexType  data;
    EBox      *firstedge;          // 指向第一条依附该顶点的边
}VexBox;
typedef struct {
    VexBox  adjmulist[MAX_VERTEX_NUM];
    int       vexnum, edgenum;     // 无向图的当前顶点数和边数
}AMLGraph;
```

7.3　图的遍历

和树的遍历类似，在此，我们希望从图中某一顶点出发访遍图中其余顶点，且使每一个顶点仅被访问一次。这一过程就叫做**图的遍历**(Traversing Graph)。图的遍历算法是求解图的连通性问题、拓扑排序和求关键路径等算法的基础。

然而，图的遍历要比树的遍历复杂得多。因为图的任一顶点都可能和其余的顶点相邻接。所以在访问了某个顶点之后，可能沿着某条路径搜索之后，又回到该顶点上。例如图 7.1(b)中的 G_2，由于图中存在回路，因此在访问了 v_1，v_2，v_3，v_4 之后，沿着边〈v_4，v_1〉又可访问到 v_1。为了避免同一顶点被访问多次，在遍历图的过程中，必须记下每个已访问过的顶点。为此，我们可以设一个辅助数组 visited[0..n−1]，它的初始值置为“假”或者零，一旦访问了顶点 v_i，便置 visited[i]为“真”或者为被访问时的次序号。

通常有两条遍历图的路径：深度优先搜索和广度优先搜索。它们对无向图和有向图都适用。

7.3.1　深度优先搜索

深度优先搜索(Depth_First Search)遍历类似于树的先根遍历，是树的先根遍历的

www.TopSage.com

推广。

假设初始状态是图中所有顶点未曾被访问，则深度优先搜索可从图中某个顶点 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

以图 7.13(a)中无向图 G_4 为例，深度优先搜索遍历图的过程如图 7.13(b)所示[①]。假设从顶点 v_1 出发进行搜索，在访问了顶点 v_1 之后，选择邻接点 v_2。因为 v_2 未曾访问，

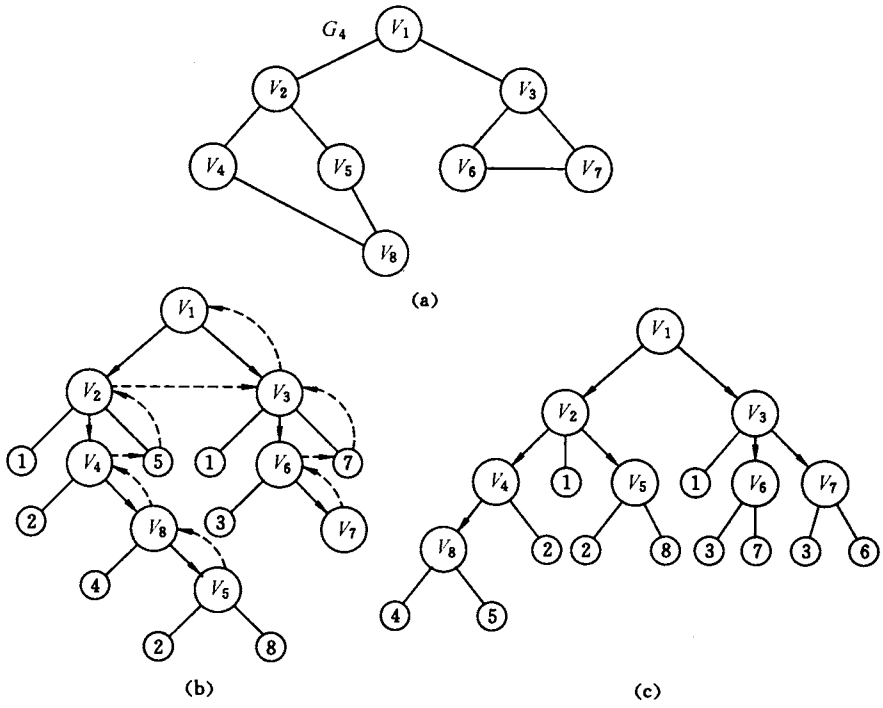


图 7.13　遍历图的过程

(a) 无向图 G_4；(b) 深度优先搜索的过程；(c) 广度优先搜索的过程

则从 v_2 出发进行搜索。依次类推，接着从 v_4，v_8、v_5 出发进行搜索。在访问了 v_5 之后，由于 v_5 的邻接点都已被访问，则搜索回到 v_8。由于同样的理由，搜索继续回到 v_4，v_2 直至 v_1，此时由于 v_1 的另一个邻接点未被访问，则搜索又从 v_1 到 v_3，再继续进行下去。由此，得到的顶点访问序列为：

$$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_8 \rightarrow v_5 \rightarrow v_3 \rightarrow v_6 \rightarrow v_7$$

显然，这是一个递归的过程。为了在遍历过程中便于区分顶点是否已被访问，需附设访问标志数组 visited[0..n−1]，其初值为“false”，一旦某个顶点被访问，则其相应的分量置为“true”。整个图的遍历如算法 7.4 和 7.5 所示，其中 $w \geqslant 0$ 表示存在邻接点。

---

① 图中以带箭头的粗实线表示遍历时的访问路径，以带箭头的虚线表示回溯的路径。图中的小圆圈表示已被访问过的邻接点，大圆圈表示访问的邻接点。

www.TopSage.com

大家网
TopSage.com

```
// - - - 算法 7.4 和 7.5 使用的全局变量 - - -
    Boolean visited[MAX];              // 访问标志数组
    Status ( * VisitFunc)(int v);      // 函数变量

    void DFSTraverse(Graph G, Status ( * Visit)(int v)) {
                                       // 对图 G 作深度优先遍历。
      VisitFunc = Visit;               // 使用全局变量 VisitFunc,使 DFS 不必设函数指针参数
      for (v = 0; v<G.vexnum; ++v) visited[v] = FALSE;  // 访问标志数组初始化
      for (v = 0; v<G.vexnum; ++v)
        if (!visited[v])  DFS(G, v);   // 对尚未访问的顶点调用 DFS
    }
```

算法 7.4

```
void DFS(Graph G, int v) {
  // 从第 v 个顶点出发递归地深度优先遍历图 G。
  visited[v] = TRUE;   VisitFunc(v);   // 访问第 v 个顶点
  for ( w = FirstAdjVex(G, v);  w>=0;  w = NextAdjVex(G, v, w) )
    if (!visited[w])  DFS(G, w);       // 对 v 的尚未访问的邻接顶点 w 递归调用 DFS
}
```

算法 7.5

分析上述算法,在遍历图时,对图中每个顶点至多调用一次 DFS 函数,因为一旦某个顶点被标志成已被访问,就不再从它出发进行搜索。因此,遍历图的过程实质上是对每个顶点查找其邻接点的过程。其耗费的时间则取决于所采用的存储结构。当用二维数组表示邻接矩阵作图的存储结构时,查找每个顶点的邻接点所需时间为 $O(n^2)$,其中 n 为图中顶点数。而当以邻接表作图的存储结构时,找邻接点所需时间为 $O(e)$,其中 e 为无向图中边的数或有向图中弧的数。由此,当以邻接表作存储结构时,深度优先搜索遍历图的时间复杂度为 $O(n+e)$。

7.3.2 广度优先搜索

广度优先搜索(Breadth_First Search)遍历类似于树的按层次遍历的过程。

假设从图中某顶点 v 出发,在访问了 v 之后依次访问 v 的各个未曾访问过的邻接点,然后分别从这些邻接点出发依次访问它们的邻接点,并使"先被访问的顶点的邻接点"先于"后被访问的顶点的邻接点"被访问,直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中尚有顶点未被访问,则另选图中一个未曾被访问的顶点作起始点,重复上述过程,直至图中所有顶点都被访问到为止。换句话说,广度优先搜索遍历图的过程是以 v 为起始点,由近至远,依次访问和 v 有路径相通且路径长度为 1,2,…的顶点。例如,对图 G_4 进行广度优先搜索遍历的过程如图 7.13(c)所示,首先访问 v_1 和 v_1 的邻接点 v_2 和 v_3,然后依次访问 v_2 的邻接点 v_4 和 v_5 及 v_3 的邻接点 v_6 和 v_7,最后访问 v_4 的邻接点 v_8。由于这些顶点的邻接点均已被访问,并且图中所有顶点都被访问,由此完成了图的遍历。得到的顶点访问序列为

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \rightarrow v_8$$

www.TopSage.com

和深度优先搜索类似，在遍历的过程中也需要一个访问标志数组。并且，为了顺次访问路径长度为2、3、…的顶点，需附设队列以存储已被访问的路径长度为1，2，…的顶点。广度优先遍历的算法如算法7.6所示。

```
void BFSTraverse(Graph G, Status ( * Visit)(int v)) {
    // 按广度优先非递归遍历图 G。使用辅助队列 Q 和访问标志数组 visited。
    for (v = 0; v<G.vexnum; ++v)  visited[v] = FALSE;
    InitQueue(Q);                              // 置空的辅助队列 Q
    for ( v = 0;  v<G.vexnum;  ++v )
        if ( !visited[v]) {                    // v 尚未访问
          visited[v] = TRUE; Visit(v);
          EnQueue(Q, v);                       // v 入队列
          while (!QueueEmpty(Q))  {
            DeQueue(Q, u);                     // 队头元素出队并置为 u
            for ( w = FirstAdjVex(G, u);  w> = 0;  w = NextAdjVex(G, u, w))
              if (! Visited[w])  {             //w 为 u 的尚未访问的邻接顶点
                Visited[w] = TRUE;  Visit(w);
                EnQueue(Q,W);
              } // if
           }// while
    }// if
}// BFSTraverse
```

算法 7.6

分析上述算法，每个顶点至多进一次队列。遍历图的过程实质上是通过边或弧找邻接点的过程，因此广度优先搜索遍历图的时间复杂度和深度优先搜索遍历相同，两者不同之处仅仅在于对顶点访问的顺序不同。

7.4 图的连通性问题

在这一节中，我们将利用遍历图的算法求解图的连通性问题，并讨论最小代价生成树以及重连通性与通信网络的经济性和可靠性的关系。

7.4.1 无向图的连通分量和生成树

在对无向图进行遍历时，对于连通图，仅需从图中任一顶点出发，进行深度优先搜索或广度优先搜索，便可访问到图中所有顶点。对非连通图，则需从多个顶点出发进行搜索，而每一次从一个新的起始点出发进行搜索过程中得到的顶点访问序列恰为其各个连通分量中的顶点集。例如，图7.3中的G_3是非连通图，按照图7.14所示G_3的邻接表进行深度优先搜索遍历，3次调用DFS过程（分别从顶点A、D和G出发）得到的顶点访问序列为：

A L M J B F C　　D E　　G K H I

这3个顶点集分别加上所有依附于这些顶点的边，便构成了非连通图G_3的3个连通

www.TopSage.com

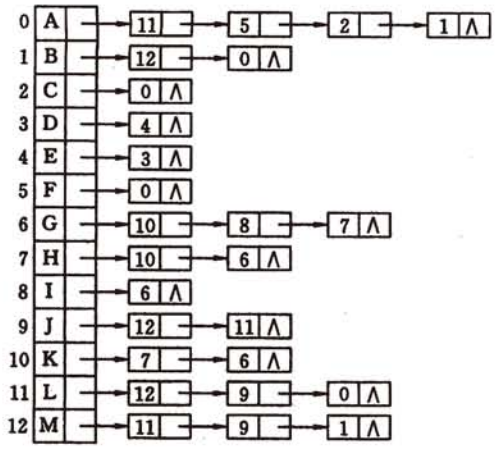


图 7.14　G_3 的邻接表

$B(G)$中的边。

对于非连通图，每个连通分量中的顶点集，和遍历时走过的边一起构成若干棵生成树，这些连通分量的生成树组成非连通图的生成森林。例如，图 7.15(c)所示为 G_3 的深度优先生成森林，它由 3 棵深度优先生成树组成。

假设以孩子兄弟链表作生成森林的存储结构，则算法 7.7 生成非连通图的深度优先生成森林，其中 DFSTree 函数如算法 7.8 所示。显然，算法 7.7 的时间复杂度和遍历相同。

```
void DFSForest(Graph G, CSTree &T) {
  // 建立无向图 G 的深度优先生成森林的
  // (最左)孩子(右)兄弟链表 T。
  T = NULL;
  for (v = 0; v<G.vexnum; ++v)
    visited[v] = FALSE;
  for (v = 0; v<G.vexnum; ++v)
    if (!visited[v]) {
      p = (CSTree) malloc (sizeof (CSNode));   // 分配根结点
      *p = { GetVex(G,v), NULL, NULL };        // 给该结点赋值
      if (!T)  T = p;                          // 是第一棵生成树的根(T 的根)
      else  q->nextsibling = p;                // 是其他生成树的根(前一棵的根的"兄弟")
      q = p;                                   // q 指示当前生成树的根
      DFSTree(G, v, p);                        // 建立以 p 为根的生成树
    }
```

分量，如图 7.3(b)所示。

设 $E(G)$为连通图 G 中所有边的集合，则从图中任一顶点出发遍历图时，必定将 $E(G)$分成两个集合 $T(G)$和 $B(G)$，其中 $T(G)$是遍历图过程中历经的边的集合；$B(G)$是剩余的边的集合。显然，$T(G)$和图 G 中所有顶点一起构成连通图 G 的极小连通子图，按照 7.1 节的定义，它是连通图的一棵生成树，并且称由深度优先搜索得到的为深度优先生成树；由广度优先搜索得到的为广度优先生成树。例如，图 7.15(a)和(b)所示分别为连通图 G_4 的深度优先生成树和广度优先生成树，图中虚线为集合

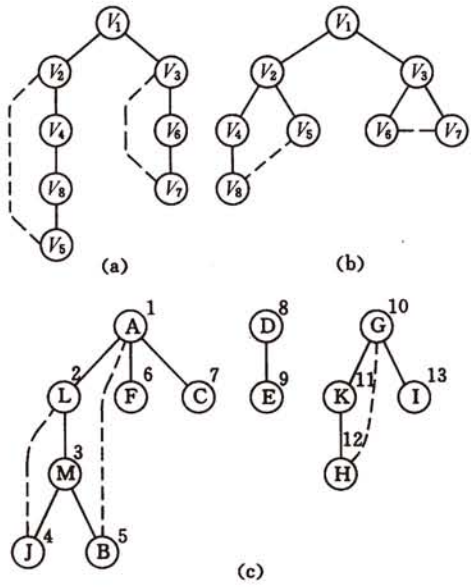


图 7.15　生成树和生成森林

(a) G_4 的深度优先生成树；

(b) G_4 的广度优先生成树；

(c) G_3 的深度优先生成森林

www.TopSage.com

```
}// DFSForest
```

算法 7.7

```
void DFSTree(Graph G, int v, CSTree &T) {
  // 从第 v 个顶点出发深度优先遍历图 G,建立以 T 为根的生成树。
  visited[v] = TRUE;  first = TRUE;
  for (w = FisrtAdjVex(G, v);  w >= 0;  w = NextAdjVex(G, v, w))
    if (!visited[w]) {
      p = (CSTree) malloc (sizeof (CSNode));    // 分配孩子结点
       *p = { GetVex(G,w), NULL, NULL };
      if (first) {                               // w 是 v 的第一个未被访问的邻接顶点
        T->lchild = p;  first = FALSE;           // 是根的左孩子结点
      }// if
      else {                                     // w 是 v 的其他未被访问的邻接顶点
        q->nextsibling = p;                      // 是上一邻接顶点的右兄弟结点
      }// else
      q = p;
      DFSTree(G, w, q);      // 从第 w 个顶点出发深度优先遍历图 G,建立子生成树 q
    }// if
}// DFSTree
```

算法 7.8

7.4.2 有向图的强连通分量

深度优先搜索是求有向图的强连通分量的一个新的有效方法。假设以十字链表作有向图的存储结构,则求强连通分量的步骤如下:

(1) 在有向图 G 上,从某个顶点出发沿以该顶点为尾的弧进行深度优先搜索遍历,并按其所有邻接点的搜索都完成(即退出 DFS 函数)的顺序将顶点排列起来。此时需对 7.3.1中的算法作如下两点修改:(a) 在进入 DFSTraverse 函数时首先进行计数变量的初始化,即在入口处加上 count=0 的语句;(b) 在退出 DFS 函数之前将完成搜索的顶点号记录在另一个辅助数组 finished[vexnum]中,即在 DFS 函数结束之前加上 finished[++count]=v 的语句。

(2) 在有向图 G 上,从最后完成搜索的顶点(即 finished[vexnum−1]中的顶点)出发,沿着以该顶点为头的弧作逆向的深度优先搜索遍历,若此次遍历不能访问到有向图中所有顶点,则从余下的顶点中最后完成搜索的那个顶点出发,继续作逆向的深度优先搜索遍历,依次类推,直至有向图中所有顶点都被访问到为止。此时调用 DFSTraverse 时需作如下修改:函数中第二个循环语句的边界条件应改为 v 从 finished[vexnum−1]至 finished[0]。

由此,每一次调用 DFS 作逆向深度优先遍历所访问到的顶点集便是有向图 G 中一个强连通分量的顶点集。

例如图 7.11 所示的有向图,假设从顶点 v_1 出发作深度优先搜索遍历,得到 finished

www.TopSage.com

大家网 TopSage.com

数组中的顶点号为(1,3,2,0);则再从顶点 v_1 出发作逆向的深度优先搜索遍历,得到两个顶点集$\{v_1,v_3,v_4\}$和$\{v_2\}$,这就是该有向图的两个强连通分量的顶点集。

上述求强连通分量的第二步,其实质为:(1)构造一个有向图 G_r,设 $G=(V,\{A\})$,则 $G_r=(V,\{A_r\})$,对于所有$\langle v_i,v_j\rangle\in A$,必有$\langle v_j,v_i\rangle\in A_r$。即 G_r 中拥有和 G 方向相反的弧;(2) 在有向图 G_r 上,从顶点 finished[vexnum−1]出发作深度优先搜索遍历。可以证明,在 G_r 上所得深度优先生成森林中每一棵树的顶点集即为 G 的强连通分量的顶点集[8]。

显然,利用遍历求强连通分量的时间复杂度亦和遍历相同。

7.4.3 最小生成树

假设要在 n 个城市之间建立通信联络网,则连通 n 个城市只需要 $n-1$ 条线路。这时,自然会考虑这样一个问题,如何在最节省经费的前提下建立这个通信网。

在每两个城市之间都可以设置一条线路,相应地都要付出一定的经济代价。n 个城市之间,最多可能设置 $n(n-1)/2$ 条线路,那么,如何在这些可能的线路中选择 $n-1$ 条,以使总的耗费最少呢?

可以用连通网来表示 n 个城市以及 n 个城市间可能设置的通信线路,其中网的顶点表示城市,边表示两城市之间的线路,赋于边的权值表示相应的代价。对于 n 个顶点的连通网可以建立许多不同的生成树,每一棵生成树都可以是一个通信网。现在,我们要选择这样一棵生成树,也就是使总的耗费最少。这个问题就是构造连通网的最小代价生成树(Minimum Cost Spanning Tree)(简称为**最小生成树**)的问题。一棵生成树的代价就是树上各边的代价之和。

构造最小生成树可以有多种算法。其中多数算法利用了最小生成树的下列一种简称为 MST 的性质:假设 $N=(V,\{E\})$是一个连通网,U 是顶点集 V 的一个非空子集。若(u,v)是一条具有最小权值(代价)的边,其中 $u\in U,v\in V-U$,则必存在一棵包含边(u,v)的最小生成树。

可以用反证法证明之。假设网 N 的任何一棵最小生成树都不包含(u,v)。设 T 是连通网上的一棵最小生成树,当将边(u,v)加入到 T 中时,由生成树的定义,T 中必存在一条包含(u,v)的回路。另一方面,由于 T 是生成树,则在 T 上必存在另一条边(u',v'),其中 $u'\in U,v'\in V-U$,且 u 和 u'之间,v 和 v'之间均有路径相通。删去边(u',v'),便可消除上述回路,同时得到另一棵生成树 T'。因为(u,v)的代价不高于(u',v'),则 T'的代价亦不高于 T,T'是包含(u,v)的一棵最小生成树。由此和假设矛盾。

普里姆(Prim)算法和克鲁斯卡尔(Kruskal)算法是两个利用 MST 性质构造最小生成树的算法。

下面先介绍普里姆算法。

假设 $N=(V,\{E\})$是连通网,TE 是 N 上最小生成树中边的集合。算法从 $U=\{u_0\}$ $(u_0\in V)$,$TE=\{\}$开始,重复执行下述操作:在所有 $u\in U,v\in V-U$ 的边$(u,v)\in E$ 中找一条代价最小的边(u_0,v_0)并入集合 TE,同时 v_0 并入 U,直至 $U=V$ 为止。此时 TE 中必有 $n-1$ 条边,则 $T=(V,\{TE\})$为 N 的最小生成树。

www.TopSage.com

为实现这个算法需附设一个辅助数组 closedge，以记录从 U 到 $V-U$ 具有最小代价的边。对每个顶点 $v_i \in V-U$，在辅助数组中存在一个相应分量 closedge[i－1]，它包括两个域，其中 lowcost 存储该边上的权。显然，

closedge[i－1]. lowcost＝Min{cost(u,v_i)[①] | u∈U}

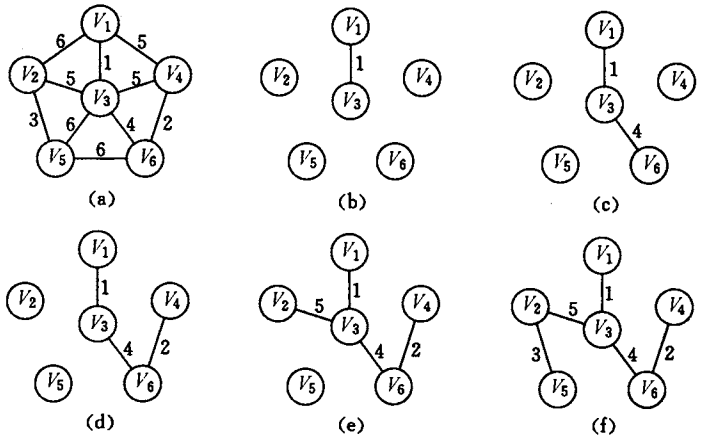


图 7.16　普里姆算法构造最小生成树的过程

vex 域存储该边依附的在 U 中的顶点。例如，图 7.16 所示为按普里姆算法构造网的一棵最小生成树的过程，在构造过程中辅助数组中各分量值的变化如图 7.17 所示。初始状态

i / closedge	1	2	3	4	5	U	V－U	k
adjvex	v_1	v_1	v_1			{v_1}	{v_2, v_3, v_4, v_5, v_6}	2
lowcost	6	1	5					
adjvex	v_3		v_1	v_3	v_3	{v_1, v_3}	{v_2, v_4, v_5, v_6}	5
lowcost	5	0	5	6	4			
adjvex	v_3		v_6	v_3		{v_1, v_3, v_6}	{v_2, v_4, v_5}	3
lowcost	5	0	2	6	0			
adjvex	v_3			v_3		{v_1, v_3, v_6, v_4}	{v_2, v_5}	1
lowcost	5	0	0	6	0			
adjvex				v_2		{v_1, v_3, v_6, v_1, v_2}	{v_5}	4
lowcost	0	0	0	3	0			
adjvex						{v_1, v_3, v_6, v_4, v_2, v_5}	{}	
lowcost	0	0	0	0	0			

图 7.17　图 7.16 构造最小生成树过程中辅助数组中各分量的值

时，由于 $U=\{v_1\}$，则到 $V-U$ 中各顶点的最小边，即为从依附于顶点 1 的各条边中，找到一条代价最小的边 $(u_0, v_0)=(1,3)$ 为生成树上的第一条边，同时将 v_0（$=v_3$）并入集合 U。然后修改辅助数组中的值。首先将 closedge[2]. lowcost 改为'0'，以示顶点 v_3 已并入 U。

① cost(u,v)表示赋于边(u,v)的权。

www.TopSage.com

大家网 TopSage.com

然后，由于边(v_3,v_2)上的权值小于 closedge[1].lowcost，则需修改 closedge[1]为边(v_3,v_2)及其权值。同理修改 closedge[4]和 closedge[5]。依次类推，直到 $U=V$。假设以二维数组表示网的邻接矩阵，且令两个顶点之间不存在的边的权值为机内允许的最大值(INT_MAX)，则普里姆算法如算法 7.9 所示。

```
void MiniSpanTree_PRIM( MGraph G, VertexType u ) {
  // 用普里姆算法从第 u 个顶点出发构造网 G 的最小生成树 T,输出 T 的各条边。
  // 记录从顶点集 U 到 V-U 的代价最小的边的辅助数组定义:
  //  struct {
  //       VertexType  adjvex;
  //       VRType      lowcost;
  //  }closedge[MAX_VERTEX_NUM];
  k = LocateVex ( G, u );
  for ( j = 0; j<G.vexnum; ++ j )      // 辅助数组初始化
     if (j! = k)  closedge[j] = { u, G.arcs[k][j].adj };  // {adjvex, lowcost }
  closedge[k].lowcost = 0;             // 初始,U = {u}
  for (i = 1; i<G.vexnum; ++ i) {      // 选择其余 G.vexnum - 1 个顶点
     k = minimum(closedge);            // 求出 T 的下一个结点:第 k 顶点
     // 此时 closedge[k].lowcost =
     //             MIN{ closedge[v_i].lowcost | closedge[v_i].lowcost>0, v_i∈V-U }
     printf(closedge[k].adjvex, G.vexs[k]);       // 输出生成树的边
     closedge[k].lowcost = 0;           // 第 k 顶点并入 U 集
     for (j = 0; j<G.vexnum; ++ j)
        if (G.arcs[k][j].adj < closedge[j].lowcost)       // 新顶点并入 U 后重新选择最
                                                          小边
           closedge[j] = { G.vexs[k], G.arcs[k][j].adj };
  }
}// MiniSpanTree
```

算法 7.9

例如，对图 7.16(a)中的网，利用算法 7.9，将输出生成树上的 5 条边为：{(v_1,v_3),(v_3,v_6),(v_6,v_4),(v_3,v_2),(v_2,v_5)}。

分析算法 7.9，假设网中有 n 个顶点，则第一个进行初始化的循环语句的频度为 n，第二个循环语句的频度为 $n-1$。其中有两个内循环：其一是在 closedge[v].lowcost 中求最小值，其频度为 $n-1$；其二是重新选择具有最小代价的边，其频度为 n。由此，普里姆算法的时间复杂度为 $O(n^2)$，与网中的边数无关，因此适用于求边稠密的网的最小生成树。

而克鲁斯卡尔算法恰恰相反，它的时间复杂度为 $O(e\log e)$(e 为网中边的数目)，因此它相对于普里姆算法而言，适合于求边稀疏的网的最小生成树。

克鲁斯卡尔算法从另一途径求网的最小生成树。假设连通网 $N=(V,\{E\})$，则令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 $T=(V,\{\})$，图中每个顶点自成一个连通分量。在 E 中选择代价最小的边，若该边依附的顶点落在 T 中不同的连通分量上，则将此边加入到 T 中，否则舍去此边而选择下一条代价最小的边。依次类推，直至 T

www.TopSage.com

中所有顶点都在同一连通分量上为止。

例如，图 7.18 所示为依照克鲁斯卡尔算法构造一棵最小生成树的过程。代价分别为

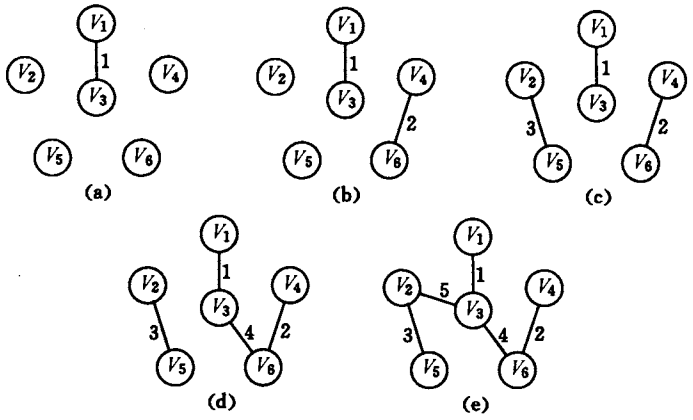


图 7.18　克鲁斯卡尔算法构造最小生成树的过程

1,2,3,4 的 4 条边由于满足上述条件，则先后被加入到 T 中，代价为 5 的两条边(v_1,v_4)和(v_3,v_4)被舍去。因为它们依附的两顶点在同一连通分量上，它们若加入 T 中，则会使 T 中产生回路，而下一条代价(=5)最小的边(v_2,v_3)联结两个连通分量，则可加入 T。由此，构造成一棵最小生成树。

上述算法至多对 e 条边各扫描一次，假若以第 9 章将介绍的"堆"来存放网中的边，则每次选择最小代价的边仅需 $O(\log e)$的时间(第一次需 $O(e)$)。又生成树 T 的每个连通分量可看成是一个等价类，则构造 T 加入新的边的过程类似于求等价类的过程，由此可以以 6.5 节中介绍的 MFSet 类型来描述 T，使构造 T 的过程仅需 $O(e\log e)$的时间，由此，克鲁斯卡尔算法的时间复杂度为 $O(e\log e)$。

7.4.4　关节点和重连通分量

假若在删去顶点 v 以及和 v 相关联的各边之后，将图的一个连通分量分割成两个或两个以上的连通分量，则称顶点 v 为该图的一个**关节点**(articulation point)。一个没有关节点的连通图称为是**重连通图**(biconnected graph)。在重连通图上，任意一对顶点之间至少存在两条路径，则在删去某个顶点以及依附于该顶点的各边时也不破坏图的连通性。若在连通图上至少删去 k 个顶点才能破坏图的连通性，则称此图的连通度为 k。关节点和重连通在实际中有较多应用。显然，一个表示通信网络的图的连通度越高，其系统越可靠，无论是哪一站点出现故障或遭到外界破坏，都不影响系统的正常工作；又如，一个航空网若是重连通的，则当某条航线因天气等某种原因关闭时，旅客仍可从别的航线绕道而行；再如，若将大规模集成电路的关键线路设计成重连通的话，则在某些元件失效的情况下，整个片子的功能不受影响，反之，在战争中，若要摧毁敌方的运输线，仅需破坏其运输网中的关节点即可。

例如，图 7.19 中图 G_5 是连通图，但不是重连通图。图中有 4 个关节点 A、B、D 和 G。若删去顶点 B 以及所有依附顶点 B 的边，G_5 就被分割成 3 个连通分量{A、C、F、L、M、

www.TopSage.com

大家网 TopSage.com

$J\}$、$\{G、H、I、K\}$和$\{D、E\}$。类似地，若删去顶点 A 或 D 或 G 以及所有依附于它们的边，则 G_5 被分割成两个连通分量，由此，关节点亦称割点。

利用深度优先搜索便可求得图的关节点，并由此可判别图是否是重连通的。

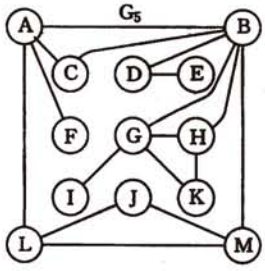


图 7.19　连通图 G_5

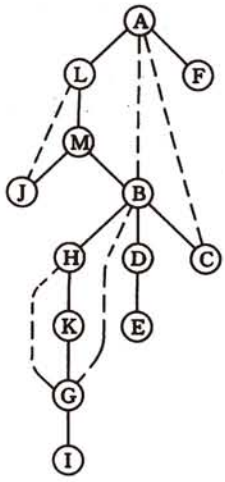


图 7.20　G_5 的深度优先生成树

图 7.20 所示为从顶点 A 出发深度优先搜索遍历图 G_5 所得深度优先生成树，图中实线表示树边，虚线表示回边(即不在生成树上的边)。对树中任一顶点 v 而言，其孩子结点为在它之后搜索到的邻接点，而其双亲结点和由回边联结的祖先结点是在它之前搜索到的邻接点。由深度优先生成树可得出两类关节点的特性：

(1) 若生成树的根有两棵或两棵以上的子树，则此根顶点必为关节点。因为图中不存在联结不同子树中顶点的边，因此，若删去根顶点，生成树便变成生成森林。如图 7.20 中的顶点 A。

(2) 若生成树中某个非叶子顶点 v，其某棵子树的根和子树中的其他结点均没有指向 v 的祖先的回边，则 v 为关节点。因为，若删去 v，则其子树和图的其他部分被分割开来。如图 7.20 中的顶点 B、D 和 G。

若对图 Graph=(V、{Edge})重新定义遍历时的访问函数 visited，并引入一个新的函数 low，则由一次深度优先搜索遍历便可求得连通图中存在的所有关节点。

定义 visited[v]为深度优先搜索遍历连通图时访问顶点 v 的次序号；定义

$$\text{low(v)} = \text{Min}\left\{ \text{visited[v]}, \text{low[w]}, \text{visited[k]} \;\middle|\; \begin{array}{l} \text{w 是顶点 v 在深度优先生成树上的孩子结点；} \\ \text{k 是顶点 v 在深度优先生成树上由回边联结的} \\ \text{祖先结点；} \\ (\text{v,w}) \in \text{Edge}, \\ (\text{v,k}) \in \text{Edge}, \end{array} \right\}$$

若对于某个顶点 v，存在孩子结点 w 且 low[w]≥visited[v]，则该顶点 v 必为关节点。因为当 w 是 v 的孩子结点时，low[w]≥visited[v]，表明 w 及其子孙均无指向 v 的祖先的回边。

由定义可知，visited[v]值即为 v 在深度优先生成树的前序序列中的序号，只需将

www.TopSage.com

DFS 函数中头两个语句改为 visited[v_0] = ++count(在 DFSTraverse 中设初值 count=1)即可;low[v]可由后序遍历深度优先生成树求得,而 v 在后序序列中的次序和遍历时退出 DFS 函数的次序相同,由此修改深度优先搜索遍历的算法便可得到求关节点的算法(见算法 7.10 和算法 7.11)。

```
void FindArticul(ALGraph G) {
  // 连通图 G 以邻接表作存储结构,查找并输出 G 上全部关节点。全局量 count
     对访问计数。
  count = 1;  visited[0] = 1;              // 设定邻接表上 0 号顶点为生成树的根
  for ( i = 1;  i<G.vexnum; ++ i )  visited[i] = 0; // 其余顶点尚未访问
  p = G.vertices[0].firstarc;   v = p->adjvex;
  DFSArticul(G, v);                        // 从第 v 顶点出发深度优先查找关节点。
  if (count < G.vexnum) {                  // 生成树的根有至少两棵子树
       printf (0, G.vertices[0].data);     // 根是关节点,输出
       while (p->nextarc) {
          p = p->nextarc;  v = p->adjvex;
          if (visited[v] == 0)  DFSArticul(g, v);
       }// while
  }// if
}// FindArticul
```

算法 7.10

```
void DFSArticul(ALGraph G, int v0) {
  // 从第 v0 个顶点出发深度优先遍历图 G,查找并输出关节点。
  visited[v0] = min = ++count;  // v0 是第 count 个访问的顶点
  for (p = G.vertices[v0].firstarc;  p;  p = p->nextarc) {   // 对 v0 的每个邻接顶点检查
     w = p->adjvex;                // w 为 v0 的邻接顶点
     if (visited[w] == 0) {        // w 未曾访问,是 v0 的孩子
        DFSArticul(G, w);          // 返回前求得 low[w]
        if (low[w] < min) min = low[w];
        if (low[w] >= visited[v0])  printf(v0, G.vertices[v0].data); // 关节点
     }else if (visited[w] < min)   min = visited[w];
                                   // w 已访问, w 是 v0 在生成树上的祖先
  }// for
  low[v0] = min;
}// DFSArticul
```

算法 7.11

例如,图 G_5 中各顶点计算所得 visited 和 low 的函数值如下所列:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
G.vertices[i].data	A	B	C	D	E	F	G	H	I	J	K	L	M
visited[i]	1	5	12	10	11	13	8	6	9	4	7	2	3
low[i]	1	1	1	5	10	1	5	5	8	2	5	1	1
求得 low 值的顺序	13	9	8	7	6	12	3	5	2	1	4	11	10

www.TopSage.com

大家网 TopSage.com

其中J是第一个求得low值的顶点，由于存在回边(J,L)，则low[J]=Min{visited[J]、visited[L]}=2。顺便提一句，上述算法中将指向双亲的树边也看成是回边，由于不影响关节点的判别，因此，为使算法简明起见，在算法中没有区别之。

由于上述算法的过程就是一个遍历的过程，因此，求关节点的时间复杂度仍为$O(n+e)$。若尚需输出双连通分量，仅需在算法中增加一些语句即可，在此不再详述，留给读者自己完成。

7.5 有向无环图及其应用

一个无环的有向图称做**有向无环图**(directed acycline graph)，简称DAG图。DAG图是一类较有向树更一般的特殊有向图，如图7.21列示了有向树、DAG图和有向图的例子。

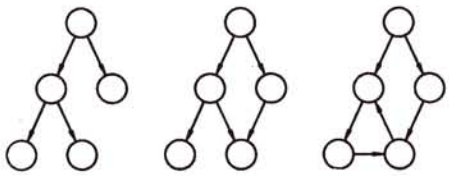


图7.21 有向树、DAG图和有向图一例

有向无环图是描述含有公共子式的表达式的有效工具。例如下述表达式

((a+b)*(b*(c+d))+(c+d)*e)*((c+d)*e)

可以用第6章讨论的二叉树来表示，如图7.22所示。仔细观察该表达式，可发现有一些相同的子表达式，如(c+d)和(c+d)*e等，在二叉树中，它们也重复出现。若利用有向无环图，则可实现对相同子式的共享，从而节省存储空间。例如图7.23所示为表示同一表达式的有向无环图。

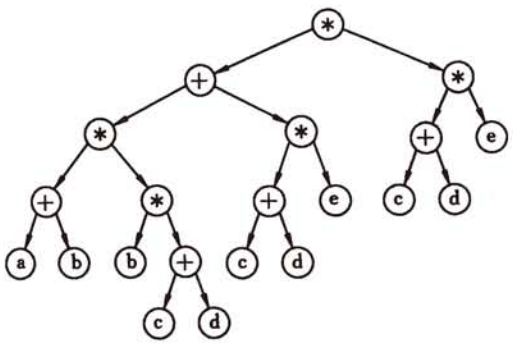


图7.22 用二叉树描述表达式

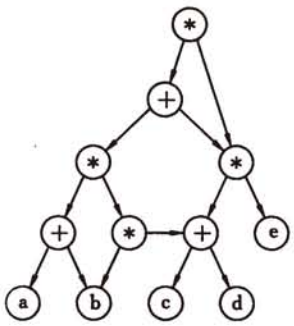


图7.23 描述表达式的有向无环图

检查一个有向图是否存在环要比无向图复杂。对于无向图来说，若深度优先遍历过程中遇到回边(即指向已访问过的顶点的边)，则必定存在环；而对于有向图来说，这条回边有可能是指向深度优先生成森林中另一棵生成树上顶点的弧。但是，如果从有向图上某个顶点v出发的遍历，在dfs(v)结束之前出现一条从顶点u到顶点v的回边(如图7.24所示)，由于u在生成树上是v的子孙，则有向图中必定存在包含顶点v和u的环。

有向无环图也是描述一项工程或系统的进行过程的有效工具。除最简单的情况之

www.TopSage.com

外，几乎所有的工程(project)都可分为若干个称做**活动**(activity)的子工程，而这些子工程之间，通常受着一定条件的约束，如其中某些子工程的开始必须在另一些子工程完成之后。对整个工程和系统，人们关心的是两个方面的问题：一是工程能否顺利进行；二是估算整个工程完成所必须的最短时间，对应于有向图，即为进行拓扑排序和求关键路径的操作。下面分别就这两个问题讨论之。

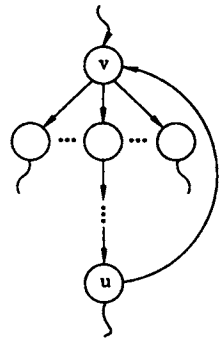


图 7.24　含有环的有向图的深度优先生成树

7.5.1　拓扑排序

什么是**拓扑排序**(Topological Sort)？简单地说，由某个集合上的一个偏序得到该集合上的一个全序，这个操作称之为拓扑排序。回顾离散数学中关于偏序和全序的定义：

若集合 X 上的关系 R 是自反的、反对称的和传递的，则称 R 是集合 X 上的偏序关系。

设 R 是集合 X 上的**偏序**(Partial Order)，如果对每个 $x,y\in X$ 必有 xRy 或 yRx，则称 R 是集合 X 上的全序关系。

直观地看，偏序指集合中仅有部分成员之间可比较，而全序指集合中全体成员之间均可比较。例如，图 7.25 所示的两个有向图，图中弧〈x,y〉表示 $x\leqslant y$，则(a)表示偏序，

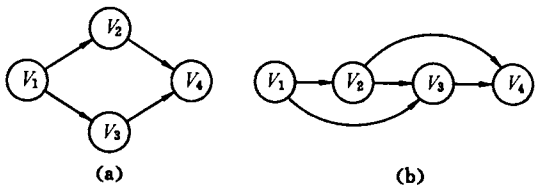


图 7.25　表示偏序和全序的有向图
(a) 表示偏序；(b)表示全序

(b)表示全序。若在(a)的有向图上人为地加一个表示 $v_2\leqslant v_3$ 的弧(符号"≤"表示 v_2 领先于 v_3)，则(a)表示的亦为全序，且这个全序称为**拓扑有序**(Topological Order)，而由偏序定义得到拓扑有序的操作便是拓扑排序。

一个表示偏序的有向图可用来表示一个流程图。它或者是一个施工流程图，或者是一个产品生产的流程图，再或是一个数据流图(每个顶点表示一个过程)。图中每一条有向边表示两个子工程之间的次序关系(领先关系)。

例如，一个软件专业的学生必须学习一系列基本课程(如图 7.26 所示)，其中有些课程是基础课，它独立于其他课程，如《高等数学》；而另一些课程必须在学完作为它的基础的先修课程才能开始。如，在《程序设计基础》和《离散数学》学完之前就不能开始学习《数据结构》。这些先决条件定义了课程之间的领先(优先)关系。这个关系可以用有向图更清楚地表示，如图 7.27 所示。图中顶点表示课程，有向边(弧)表示先决条件。若课程 i

www.TopSage.com

大家网 TopSage.com

是课程 j 的先决条件，则图中有弧〈i,j〉。

这种用顶点表示活动，用弧表示活动间的优先关系的有向图称为顶点表示活动的网(Activity On Vertex Network)，简称 AOV-网。在网中，若从顶点 i 到顶点 j 有一条有向路径，则 i 是 j 的前驱；j 是 i 的后继。若〈i,j〉是网中一条弧，则 i 是 j 的直接前驱；j 是 i 的直接后继。

课程编号	课程名称	先决条件
C_1	程序设计基础	无
C_2	离散数学	C_1
C_3	数据结构	C_1,C_2
C_4	汇编语言	C_1
C_5	语言的设计和分析	C_3,C_4
C_6	计算机原理	C_{11}
C_7	编译原理	C_5,C_3
C_8	操作系统	C_3,C_6
C_9	高等数学	无
C_{10}	线性代数	C_9
C_{11}	普通物理	C_9
C_{12}	数值分析	C_9,C_{10},C_1

图 7.26　软件专业的学生必须学习的课程

在 AOV-网中，不应该出现有向环，因为存在环意味着某项活动应以自己为先决条件。显然，这是荒谬的。若设计出这样的流程图，工程便无法进行。而对程序的数据流图来说，则表明存在一个死循环。因此，对给定的 AOV-网应首先判定网中是否存在环。检测的办法是对有向图构造其顶点的拓扑有序序列，若网中所有顶点都在它的拓扑有序序列中，则该 AOV-网中必定不存在环。例如，图 7.27 的有向图有如下两个拓扑有序序列：

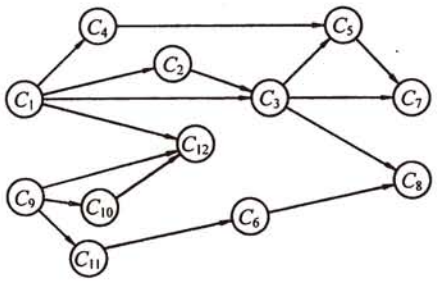


图 7.27　表示课程之间优先关系的有向图

$$(C_1,C_2,C_3,C_4,C_5,C_7,C_9,C_{10},C_{11},C_6,C_{12},C_8)$$

和

$$(C_9,C_{10},C_{11},C_6,C_1,C_{12},C_4,C_2,C_3,C_5,C_7,C_8)$$

(当然，对此图也可构造得其他的拓扑有序序列)。若某个学生每学期只学一门课程的话，则他必须按拓扑有序的顺序来安排学习计划。

如何进行拓扑排序？解决的方法很简单：

www.TopSage.com

(1) 在有向图中选一个没有前驱的顶点且输出之。

(2) 从图中删除该顶点和所有以它为尾的弧。

重复上述两步，直至全部顶点均已输出，或者当前图中不存在无前驱的顶点为止。后一种情况则说明有向图中存在环。

以图 7.28(a)中的有向图为例，图中，v_1 和 v_6 没有前驱，则可任选一个。假设先输出 v_6，在删除 v_6 及弧$\langle v_6, v_4\rangle$，$\langle v_6, v_5\rangle$之后，只有顶点 v_1 没有前驱，则输出 v_1 且删去 v_1 及弧$\langle v_1, v_2\rangle$、$\langle v_1, v_3\rangle$和$\langle v_1, v_4\rangle$，之后 v_3 和 v_4 都没有前驱。依次类推，可从中任选一个继续进行。整个拓扑排序的过程如图 7.28 所示。最后得到该有向图的拓扑有序序列为：

$v_6 - v_1 - v_4 - v_3 - v_2 - v_5$。

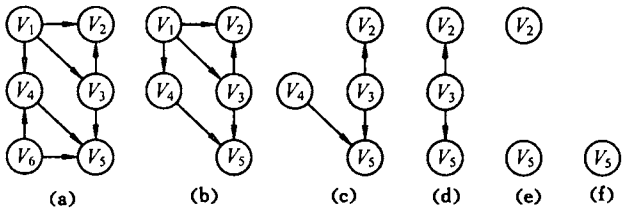


图 7.28 AOV-网及其拓扑有序序列产生的过程

(a) AOV-网；(b) 输出 v_6 之后；(c) 输出 v_1 之后；

(d) 输出 v_4 之后；(e) 输出 v_3 之后；(f) 输出 v_2 之后

如何在计算机中实现？针对上述两步操作，我们可采用邻接表作有向图的存储结构，且在头结点中增加一个存放顶点入度的数组(indegree)。入度为零的顶点即为没有前驱的顶点，删除顶点及以它为尾的弧的操作，则可换以弧头顶点的入度减 1 来实现。

为了避免重复检测入度为零的顶点，可另设一栈暂存所有入度为零的顶点，由此可得拓扑排序的算法如算法 7.12 所示。

```
Status TopologicalSort(ALGraph G) {
  // 有向图 G 采用邻接表存储结构。
  // 若 G 无回路，则输出 G 的顶点的一个拓扑序列并返回 OK，否则 ERROR。
  FindInDegree(G, indegree);              // 对各顶点求入度 indegree[0..vernum-1]
  InitStack(S);
  for (i=0; i<G.vexnum; ++i)              // 建零入度顶点栈 S
    if (!indegree[i]) Push(S, i);         // 入度为 0 者进栈
  count = 0;                              // 对输出顶点计数
  while (!StackEmpty(S)) {
    Pop(S, i); printf(i, G.vertices[i].data); ++count; // 输出 i 号顶点并计数
    for (p=G.vertices[i].firstarc; p; p=p->nextarc) {
      k = p->adjvex;                      // 对 i 号顶点的每个邻接点的入度减 1
      if (!(--indegree[k])) Push(S, k); // 若入度减为 0，则入栈
    }// for
  }// while
  if (count<G.vexnum) return ERROR;       // 该有向图有回路
  else return OK;
}// TopologicalSort
```

算法 7.12

www.TopSage.com

大家网 TopSage.com

分析算法 7.12，对有 n 个顶点和 e 条弧的有向图而言，建立求各顶点的入度的时间复杂度为 $O(e)$；建零入度顶点栈的时间复杂度为 $O(n)$；在拓扑排序过程中，若有向图无环，则每个顶点进一次栈，出一次栈，入度减 1 的操作在 WHILE 语句中总共执行 e 次，所以，总的时间复杂度为 $O(n+e)$。上述拓扑排序的算法亦是下节讨论的求关键路径的基础。

当有向图中无环时，也可利用深度优先遍历进行拓扑排序，因为图中无环，则由图中某点出发进行深度优先搜索遍历时，最先退出 DFS 函数的顶点即出度为零的顶点，是拓扑有序序列中最后一个顶点。由此，按退出 DFS 函数的先后记录下来的顶点序列（如同求强连通分量时 finished 数组中的顶点序列）即为逆向的拓扑有序序列。

7.5.2 关键路径

与 AOV-网相对应的是 AOE-网（Activity On Edge）即边表示活动的网。AOE-网是一个带权的有向无环图，其中，顶点表示事件（Event），弧表示活动，权表示活动持续的时间。通常，AOE-网可用来估算工程的完成时间。

例如，图 7.29 是一个假想的有 11 项活动的 AOE-网。其中有 9 个事件 $v_1, v_2, v_3, \cdots, v_9$，每个事件表示在它之前的活动已经完成，在它之后的活动可以开始。如 v_1 表示整个工程开始，v_9 表示整个工程结束，v_5 表示 a_4 和 a_5 已经完成，a_7 和 a_8 可以开始。与每个活动相联系的数是执行该活动所需的时间。比如，活动 a_1 需要 6 天，a_2 需要 4 天等。

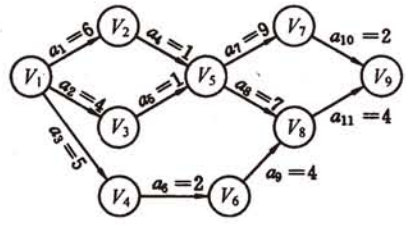


图 7.29 一个 AOE-网

由于整个工程只有一个开始点和一个完成点，故在正常的情况（无环）下，网中只有一个入度为零的点（称做源点）和一个出度为零的点（叫做汇点）。

和 AOV-网不同，对 AOE-网有待研究的问题是：(1)完成整项工程至少需要多少时间？(2)哪些活动是影响工程进度的关键？

由于在 AOE-网中有些活动可以并行地进行，所以完成工程的最短时间是从开始点到完成点的最长路径的长度（这里所说的路径长度是指路径上各活动持续时间之和，不是路径上弧的数目）。路径长度最长的路径叫做**关键路径**（Critical Path）。假设开始点是 v_1，从 v_1 到 v_i 的最长路径长度叫做事件 v_i 的最早发生时间。这个时间决定了所有以 v_i 为尾的弧所表示的活动的最早开始时间。我们用 $e(i)$ 表示活动 a_i 的最早开始时间。还可以定义一个活动的最迟开始时间 $l(i)$，这是在不推迟整个工程完成的前提下，活动 a_i 最迟必须开始进行的时间。两者之差 $l(i)-e(i)$ 意味着完成活动 a_i 的时间余量。我们把 $l(i)=e(i)$ 的活动叫做关键活动。显然，关键路径上的所有活动都是关键活动，因此提前完成非关键活动并不能加快工程的进度。例如图 7.29 中的网，从 v_1 到 v_9 的最长路径是 $(v_1, v_2, v_5, v_8, v_9)$，路径长度是 18，即 v_9 的最早发生时间是 18。而活动 a_6 的最早开始时间是 5，最迟开始时间是 8，这意味着：如果 a_6 推迟 3 天开始或者延迟 3 天完成，都不会影响整个工程的完成。因此，分析关键路径的目的是辨别哪些是关键活动，以便争取提高关

www.TopSage.com

键活动的工效，缩短整个工期。

由上分析可知，辨别关键活动就是要找 $e(i)=l(i)$ 的活动。为了求得 AOE-网中活动的 $e(i)$ 和 $l(i)$，首先应求得事件的最早发生时间 $ve(j)$ 和最迟发生时间 $vl(j)$。如果活动 a_i 由弧 $\langle j,k\rangle$ 表示，其持续时间记为 $dut(\langle j,k\rangle)$，则有如下关系

$$\begin{aligned} e(i) &= ve(j) \\ l(i) &= vl(k) - dut(\langle j,k\rangle) \end{aligned} \tag{7-1}$$

求 $ve(j)$ 和 $vl(j)$ 需分两步进行：

(1) 从 $ve(0)=0$ 开始向前递推

$$ve(j) = \operatorname{Max}_i\{ve(i) + dut(\langle i,j\rangle)\} \tag{7-2}$$
$$\langle i,j\rangle \in T, \quad j = 1,2,\cdots,n-1$$

其中，T 是所有以第 j 个顶点为头的弧的集合。

(2) 从 $vl(n-1)=ve(n-1)$ 起向后递推

$$vl(i) = \operatorname{Min}_j\{vl(j) - dut(\langle i,j\rangle)\} \tag{7-3}$$
$$\langle i,j\rangle \in S, \quad i = n-2,\cdots,0$$

其中，S 是所有以第 i 个顶点为尾的弧的集合。

这两个递推公式的计算必须分别在拓扑有序和逆拓扑有序的前提下进行。也就是说，$ve(j-1)$ 必须在 v_j 的所有前驱的最早发生时间求得之后才能确定，而 $vl(j-1)$ 则必须在 v_j 的所有后继的最迟发生时间求得之后才能确定。因此，可以在拓扑排序的基础上计算 $ve(j-1)$ 和 $vl(j-1)$。

由此得到如下所述求关键路径的算法：

(1) 输入 e 条弧 $\langle j,k\rangle$，建立 AOE-网的存储结构；

(2) 从源点 v_0 出发，令 $ve[0]=0$，按拓扑有序求其余各顶点的最早发生时间 $ve[i]$ $(1\leqslant i\leqslant n-1)$。如果得到的拓扑有序序列中顶点个数小于网中顶点数 n，则说明网中存在环，不能求关键路径，算法终止；否则执行步骤(3)。

(3) 从汇点 v_n 出发，令 $vl[n-1]=ve[n-1]$，按逆拓扑有序求其余各顶点的最迟发生时间 $vl[i](n-2\geqslant i\geqslant 2)$；

(4) 根据各顶点的 ve 和 vl 值，求每条弧 s 的最早开始时间 $e(s)$ 和最迟开始时间 l(s)。若某条弧满足条件 $e(s)=l(s)$，则为关键活动。

如上所述，计算各顶点的 ve 值是在拓扑排序的过程中进行的，需对拓扑排序的算法作如下修改：(a)在拓扑排序之前设初值，令 $ve[i]=0(0\leqslant i\leqslant n-1)$；(b)在算法中增加一个计算 v_j 的直接后继 v_k 的最早发生时间的操作：若 $ve[j]+dut(\langle j,k\rangle)>ve[k]$，则 $ve[k]=ve[j]+dut(\langle j,k\rangle)$；(c)为了能按逆拓扑有序序列的顺序计算各顶点的 vl 值，需记下在拓扑排序的过程中求得的拓扑有序序列，这需要在拓扑排序算法中，增设一个栈以记录拓扑有序序列，则在计算求得各顶点的 ve 值之后，从栈顶至栈底便为逆拓扑有序序列。

先将算法 7.12 改写成算法 7.13，则算法 7.14 便为求关键路径的算法。

www.TopSage.com

大家网 TopSage.com

```
Status TopologicalOrder(ALGraph G, Stack &T) {
  // 有向网 G 采用邻接表存储结构,求各顶点事件的最早发生时间 ve(全局变量)。
  // T 为拓扑序列顶点栈,S 为零入度顶点栈。
  // 若 G 无回路,则用栈 T 返回 G 的一个拓扑序列,且函数值为 OK,否则为 ERROR。
  FindInDegree(G, indegree); // 对各顶点求入度 indegree[0..vernum-1]
  建零入度顶点栈 S;
  InitStack(T); count = 0; ve[0..G.vexnum-1] = 0; // 初始化
  while (!StackEmpty(S)) {
    Pop(S, j); Push(T, j); ++count; // j 号顶点入 T 栈并计数
    for (p=G.vertices[j].firstarc; p; p=p->nextarc) {
      k = p->adjvex; // 对 j 号顶点的每个邻接点的入度减 1
      if (--indegree[k] == 0) Push(S, k); // 若入度减为 0,则入栈
      if (ve[j]+*(p->info) > ve[k]) ve[k] = ve[j]+*(p->info);
    }// for *(p->info)=dut(<j,k>)

  }// while
  if (count<G.vexnum) return ERROR; // 该有向网有回路
  else return OK;
}// TopologicalOrder
```

算法 7.13

```
Status CriticalPath((ALGraph G) {
  // G 为有向网,输出 G 的各项关键活动。
  if (!TopologicalOrder(G, T)) return ERROR;
  vl[0..G.vexnum-1] = ve[G.vexnum-1]; // 初始化顶点事件的最迟发生时间
  while (!StackEmpty(T)) // 按拓扑逆序求各顶点的 vl 值
    for (Pop(T, j), p=G.vertices[j].firstarc; p; p=p->nextarc) {
      k=p->adjvex; dut=*(p->info); // dut<j,k>
      if (vl[k]-dut<vl[j]) vl[j]=vl[k]-dut;
    }//for
  for (j=0; j<G.vexnum; ++j) // 求 ee,el 和关键活动
    for (p=G.vertices[j]; .firstarc;p; p=p->nextarc) {
      k=p->adjvex; dut=*(p->info);
      ee=ve[j]; el=vl[k]-dut;
      tag = (ee==el)?'*':' ';
      printf(j, k, dut, ee, el, tag); // 输出关键活动
    }
}// CriticalPath
```

算法 7.14

由于逆拓扑排序必定在网中无环的前提下进行,则亦可利用 DFS 函数,在退出 DFS 函数之前按式(7-3)计算顶点 v 的 vl 值(因为此时 v 的所有直接后继的 vl 值都已求出)。

这两种算法的时间复杂度均为 $O(n+e)$,显然,前一种算法的常数因子要小些。由于计算弧的活动最早开始时间和最迟开始时间的复杂度为 $O(e)$,所以总的求关键路径的时间复杂度为 $O(n+e)$。

例如,对图 7.30(a)所示网的计算结果如图 7.31 所示,可见 a_2、a_5 和 a_7 为关键活动,组成一条从源点到汇点的关键路径,如图 7.30(b)所示。

www.TopSage.com

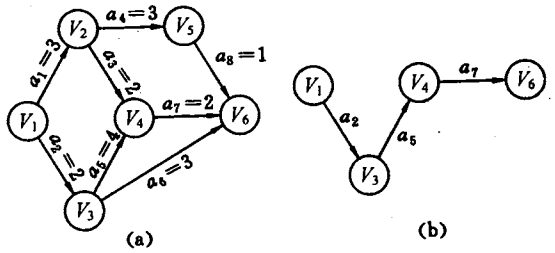


图 7.30　AOE-网及其关键路径

(a) AOE-网；(b) 关键路径

顶点	ve	vl	活动	e	l	l-e
v_1	0	0	a_1	0	1	1
v_2	3	4	a_2	0	0	0
v_3	2	2	a_3	3	4	1
v_4	6	6	a_4	3	4	1
v_5	6	7	a_5	2	2	0
v_6	8	8	a_6	2	5	3
			a_7	6	6	0
			a_8	6	7	1

图 7.31　图 7.30(a)所示 AOE-网中顶点的发生时间和活动的开始时间

对于图 7.29 所示的网，可计算求得关键活动为 a_1，a_4，a_7，a_8，a_{10}和 a_{11}。如图 7.32 所示，它们构成两条关键路径：(v_1，v_2，v_5，v_7，v_9)和(v_1，v_2，v_5，v_8，v_9)。

实践已经证明：用 AOE-网来估算某些工程完成的时间是非常有用的。实际上，求关键路径的方法本身最初就是与维修和建造工程一起发展的。但是，由于网中各项活动是互相牵涉的，因此，影响关键活动的因素亦是多方面的，任何一项活动持续时间的改变都会影响关键路径的改变。例如，对于图 7.30(a)所示的网来说，若 a_5 的持续时间改为 3，则可发现，关键活动数量增加，关键路径也增加。若同时将 a_4 的时间改成 4，则(v_1，v_3，v_4，v_6)不再是关键路径。由此可见，关键活动的速度提高是有限度的。只有在不改变网的关键路径的情况下，提高关键活动的速度才有效。

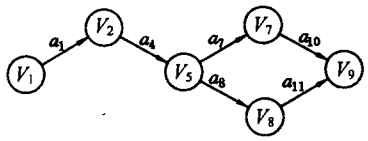


图 7.32　图 7.29 所示网的关键路径

另一方面，若网中有几条关键路径，那么，单是提高一条关键路径上的关键活动的速度，还不能导致整个工程缩短工期，而必须提高同时在几条关键路径上的活动的速度。

7.6　最短路径

假若要在计算机上建立一个交通咨询系统则可以采用图的结构来表示实际的交通网络。如图 7.33 所示，图中顶点表示城市，边表示城市间的交通联系。这个咨询系统可以回答。旅客提出的各种问题。例如，一位旅客要从 A 城到 B 城，他希望选择一条途中中转次数最少的路线。假设图中每一站都需要换车，则这个问题反映到图上就是要找一条

www.TopSage.com

大家网
TopSage.com

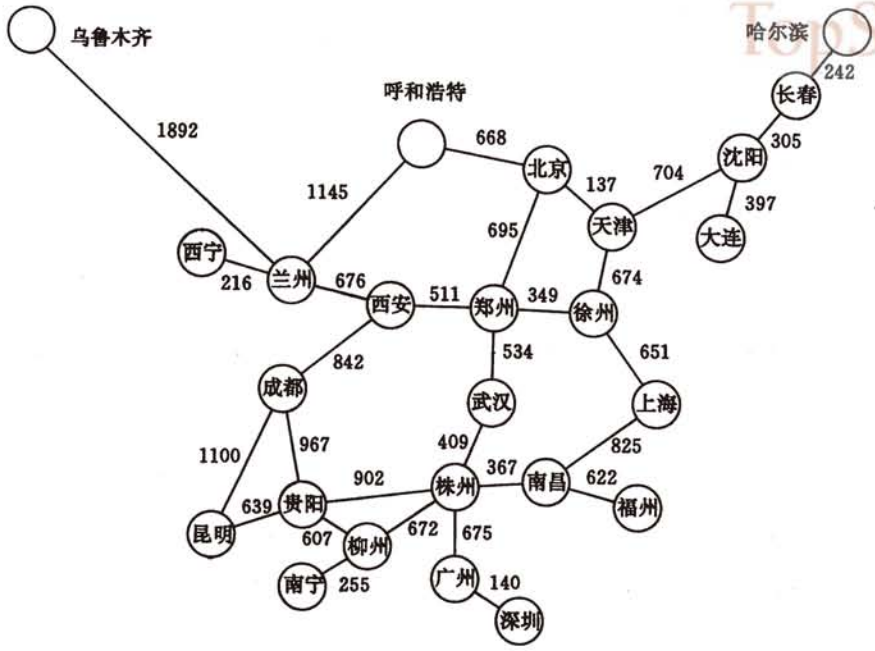


图 7.33　一个表示交通网的例图

从顶点 A 到 B 所含边的数目最少的路径。我们只需从顶点 A 出发对图作广度优先搜索，一旦遇到顶点 B 就终止。由此所得广度优先生成树上，从根顶点 A 到顶点 B 的路径就是中转次数最少的路径，路径上 A 与 B 之间的顶点就是途径的中转站数，但是，这只是一类最简单的图的最短路径问题。有时，对于旅客来说，可能更关心的是节省交通费用；而对于司机来说，里程和速度则是他们感兴趣的信息。为了在图上表示有关信息，可对边赋以权，权的值表示两城市间的距离，或途中所需时间，或交通费用等等。此时路径长度的度量就不再是路径上边的数目，而是路径上边的权值之和。考虑到交通图的有向性（如航运，逆水和顺水时的船速就不一样），本节将讨论带权有向图，并称路径上的第一个顶点为**源点**（Sourse），最后一个顶点为**终点**（Destination）。下面讨论两种最常见的最短路径问题。

7.6.1　从某个源点到其余各顶点的最短路径

我们先来讨论单源点的最短路径问题：给定带权有向图 G 和源点 v，求从 v 到 G 中其余各顶点的最短路径。

例如，图 7.34 所示带权有向图 G_6 中从 v_0 到其余各顶点之间的最短路径，如图 7.35 所示。从图中可见，从 v_0 到 v_3 有两条不同的路径：(v_0, v_2, v_3) 和 (v_0, v_4, v_3)，前者长度为 60，而后者的长度为 50。因此，后者是从 v_0 到 v_3 的最短路径；而从 v_0 到 v_1 没有路径。

如何求得这些路径？迪杰斯特拉（Dijkstra）提出了一个按路径长度递增的次序产生最短路径的算法。

首先，引进一个辅助向量 D，它的每个分量 $D[i]$ 表示当前所找到的从始点 v 到每个

www.TopSage.com

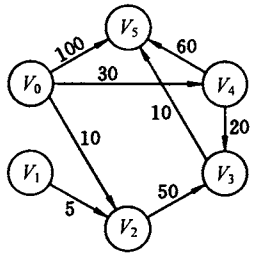


图 7.34　带权有向图 G_6

始点	终点	最短路径	路径长度
v_0	v_1	无	
	v_2	(v_0, v_2)	10
	v_3	(v_0, v_4, v_3)	50
	v_4	(v_0, v_4)	30
	v_5	(v_0, v_4, v_3, v_5)	60

图 7.35　有向图 G_6 中从 v_0 到其余各点的最短路径

终点 v_i 的最短路径的长度。它的初态为：若从 v 到 v_i 有弧，则 $D[i]$为弧上的权值；否则置 $D[i]$为∞。显然，长度为

$$D[j] = \underset{i}{\text{Min}}\{D[i] \mid v_i \in V\}$$

的路径就是从 v 出发的长度最短的一条最短路径。此路径为(v, v_j)。

那么，下一条长度次短的最短路径是哪一条呢？假设该次短路径的终点是 v_k，则可想而知，这条路径或者是(v, v_k)，或者是(v, v_j, v_k)。它的长度或者是从 v 到 v_k 的弧上的权值，或者是 $D[j]$和从 v_j 到 v_k 的弧上的权值之和。

一般情况下，假设 S 为已求得最短路径的终点的集合，则可证明：下一条最短路径(设其终点为 x)或者是弧(v, x)，或者是中间只经过 S 中的顶点而最后到达顶点 x 的路径。这可用反证法来证明。假设此路径上有一个顶点不在 S 中，则说明存在一条终点不在 S 而长度比此路径短的路径。但是，这是不可能的。因为我们是按路径长度递增的次序来产生各最短路径的，故长度比此路径短的所有路径均已产生，它们的终点必定在 S 中，即假设不成立。

因此，在一般情况下，下一条长度次短的最短路径的长度必是

$$D[j] = \underset{i}{\text{Min}}\{D[i] \mid v_i \in V - S\}$$

其中，$D[i]$或者是弧(v, v_i)上的权值，或者是 $D[k](v_k \in S)$和弧(v_k, v_i)上的权值之和。

根据以上分析，可以得到如下描述的算法：

(1) 假设用带权的邻接矩阵 $arcs$ 来表示带权有向图，$arcs[i][j]$表示弧$\langle v_i, v_j \rangle$上的权值。若$\langle v_i, v_j \rangle$不存在，则置 $arcs[i][j]$为∞(在计算机上可用允许的最大值代替)。S 为已找到从 v 出发的最短路径的终点的集合，它的初始状态为空集。那么，从 v 出发到图上其余各顶点(终点)v_i 可能达到的最短路径长度的初值为：

$$D[i] = arcs[\text{Locate Vex}(G, v)][i] \quad v_i \in V$$

(2) 选择 v_j，使得

$$D[j] = \text{Min}\{D[i] \mid v_i \in V - S\}$$

v_j 就是当前求得的一条从 v 出发的最短路径的终点。令

$$S = S \cup \{j\}$$

(3) 修改从 v 出发到集合 $V-S$ 上任一顶点 v_k 可达的最短路径长度。如果

$$D[j] + arcs[j][k] < D[k]$$

则修改 $D[k]$为

www.TopSage.com

大家网 TopSage.com

$$D[k] = D[j] + arcs[j][k]$$

(4) 重复操作(2)、(3)共 $n-1$ 次。由此求得从 v 到图上其余各顶点的最短路径是依路径长度递增的序列。

算法 7.15 为用 C 语言描述的迪杰斯特拉算法。

```
void ShortestPath_DIJ( MGraph G, int v0, PathMatrix &P, ShortPathTable &D ) {
  // 用 Dijkstra 算法求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v]及其带权长度 D[v]。
  // 若 P[v][w]为 TRUE,则 w 是从 v0 到 v 当前求得最短路径上的顶点。
  // final[v]为 TRUE 当且仅当 v∈S,即已经求得从 v0 到 v 的最短路径。
  for (v = 0; v<G.vexnum; ++v) {
    final[v] = FALSE;  D[v] = G.arcs[v0][v];
    for (w = 0; w<G.vexnum; ++w)  P[v][w] = FALSE;     // 设空路径
    if (D[v] < INFINITY) {P[v][v0] = TRUE;  P[v][v] = TRUE; }
  }// for
  D[v0] = 0;  final[v0] = TRUE;                    // 初始化,v0 顶点属于 S 集
  //开始主循环,每次求得 v0 到某个 v 顶点的最短路径,并加 v 到 S 集
  for (i = 1; i<G.vexnum; ++i) {                   // 其余 G.vexnum - 1 个顶点
    min = INFINITY;                                // 当前所知离 v0 顶点的最近距离
    for (w = 0; w<G.vexnum; ++w)
      if (!final[w])                               // w 顶点在 V - S 中
        if (D[w]<min) {v = w;  min = D[w]; }       // w 顶点离 v0 顶点更近
    final[v] = TRUE;                               // 离 v0 顶点最近的 v 加入 S 集
    for (w = 0; w<G.vexnum; ++w)  // 更新当前最短路径及距离
      if (!final[w] && (min + G.arcs[v][w]<D[w])) {  // 修改 D[w]和 P[w], w∈V - S
        D[w] = min + G.arcs[v][w];
        P[w] = P[v];  P[w][w] = TRUE;  // P[w] = P[v] + [w]
      }// if
  }// for
}// ShortestPath_DIJ
```

算法 7.15

例如,图 7.34 所示有向网 G_6 的带权邻接矩阵为

$$\begin{bmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

若对 G_6 施行迪杰斯特拉算法,则所得从 v_0 到其余各顶点的最短路径,以及运算过程中 D 向量的变化状况,如下所示:

www.TopSage.com

终点	从 v_0 到各终点的 D 值和最短路径的求解过程				
	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
v_1	∞	∞	∞	∞	∞ 无
v_2	10 (v_0,v_2)				
v_3	∞	60 (v_0,v_2,v_3)	50 (v_0,v_4,v_3)		
v_4	30 (v_0,v_4)	30 (v_0,v_4)			
v_5	100 (v_0,v_5)	100 (v_0,v_5)	90 (v_0,v_4,v_5)	60 (v_0,v_4,v_3,v_5)	
v_j	v_2	v_4	v_3	v_5	
S	$\{v_0,v_2\}$	$\{v_0,v_2,v_4\}$	$\{v_0,v_2,v_3,v_4\}$	$\{v_0,v_2,v_3,v_4,v_5\}$	

我们分析这个算法的运行时间。第一个 FOR 循环的时间复杂度是 $O(n)$，第二个 FOR 循环共进行 $n-1$ 次，每次执行的时间是 $O(n)$。所以总的时间复杂度是 $O(n^2)$。如果用带权的邻接表作为有向图的存储结构，则虽然修改 D 的时间可以减少，但由于在 D 向量中选择最小分量的时间不变，所以总的时间仍为 $O(n^2)$。

人们可能只希望找到从源点到某一个特定的终点的最短路径，但是，这个问题和求源点到其他所有顶点的最短路径一样复杂，其时间复杂度也是 $O(n^2)$的。

7.6.2 每一对顶点之间的最短路径

解决这个问题的一个办法是：每次以一个顶点为源点，重复执行迪杰斯特拉算法 n 次。这样，便可求得每一对顶点之间的最短路径。总的执行时间为 $O(n^3)$。

这里要介绍由弗洛伊德(Floyd)提出的另一个算法。这个算法的时间复杂度也是 $O(n^3)$，但形式上简单些。

弗洛伊德算法仍从图的带权邻接矩阵 *cost* 出发，其基本思想是：

假设求从顶点 v_i 到 v_j 的最短路径。如果从 v_i 到 v_j 有弧，则从 v_i 到 v_j 存在一条长度为 $arcs[i][j]$的路径，该路径不一定是最短路径，尚需进行 n 次试探。首先考虑路径 (v_i,v_0,v_j)是否存在(即判别弧(v_i,v_0)和(v_0,v_j)是否存在)。如果存在，则比较(v_i,v_j)和(v_i,v_0,v_j)的路径长度取长度较短者为从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径。假如在路径上再增加一个顶点 v_1，也就是说，如果$(v_i,\cdots,v_1)$和$(v_1,\cdots,v_j)$分别是当前找到的中间顶点的序号不大于 0 的最短路径，那么$(v_i,\cdots,v_1,\cdots,v_j)$就有可能是从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径。将它和已经得到的从 v_i 到 v_j 中间顶点序号不大于 0 的最短路径相比较，从中选出中间顶点的序号不大于 1 的最短路径之后，再增加一个顶点 v_2，继续进行试探。依次类推。在一般情况下，若$(v_i,\cdots,v_k)$和$(v_k,\cdots,v_j)$分别是从 v_i 到 v_k 和从 v_k 到 v_j 的中间顶点的序号不大于 $k-1$ 的最短路径，则将$(v_i,\cdots,v_k,\cdots,v_j)$和已经得到的从 v_i 到 v_j 且中间顶点序号不大于 $k-1$ 的最短路径相比较，其长

www.TopSage.com

大家网 TopSage.com

度较短者便是从 v_i 到 v_j 的中间顶点的序号不大于 k 的最短路径。这样，在经过 n 次比较后，最后求得的必是从 v_i 到 v_j 的最短路径。按此方法，可以同时求得各对顶点间的最短路径。

现定义一个 n 阶方阵序列

$$D^{(-1)},D^{(0)},D^{(1)},\cdots,D^{(k)},\cdots,D^{(n-1)}$$

其中

$$D^{(-1)}[i][j]=G.arcs[i][j]$$

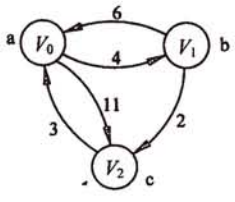
$$D^{(k)}[i][j]=Min\{D^{(k-1)}[i][j],D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\} \quad 0\leqslant k\leqslant n-1$$

从上述计算公式可见，$D^{(1)}[i][j]$是从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径的长度；$D^{(k)}[i][j]$是从 v_i 到 v_j 的中间顶点的序号不大于 k 的最短路径的长度；$D^{(n-1)}[i][j]$就是从 v_i 到 v_j 的最短路径的长度。

由此可得算法 7.16。

```
void ShortestPath_FLOYD( MGraph G, PathMatrix &P[], DistancMatrix &D ) {
  // 用 Floyd 算法求有向网 G 中各对顶点 v 和 w 之间的最短路径 P[v][w]及其
  // 带权长度 D[v][w]。若 P[v][w][u]为 TRUE,则 u 是从 v 到 w 当前求得最
  // 短路径上的顶点。
  for (v = 0; v<G.vexnum; ++v)          // 各对结点之间初始已知路径及距离
    for (w = 0; w<G.vexnum; ++w) {
      D[v][w] = G.arcs[v][w];
      for (u = 0; u<G.vexnum; ++u)  P[v][w][u] = FALSE;
      if (D[v][w] < INFINITY) {     // 从 v 到 w 有直接路径
        P[v][w][v] = TRUE; P[v][w][w] = TRUE;
      }// if
    }// for
  for (u = 0; u<G.vexnum; ++u)
    for (v = 0; v<G.vexnum; ++v)
      for (w = 0; w<G.vexnum; ++w)
        if (D[v][u] + D[u][w] < D[v][w]) {// 从 v 经 u 到 w 的一条路径更短
          D[v][w] = D[v][u] + D[u][w];
          for (i = 0; i<G.vexnum; ++i)
            P[v][w][i] = P[v][u][i] || P[u][w][i];
        }// if
}// ShortestPath_FLOYD
```

算法 7.16



(a)

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

(b)

图 7.36 带权有向图

(a) 有向网 G_7； (b) 邻接矩阵

www.TopSage.com

例如，利用上述算法，可求得图 7.36 所示带权有向图 G_7 的每一对顶点之间的最短路径及其路径长度如图 7.37 所示。

D	$D^{(-1)}$			$D^{(0)}$			$D^{(1)}$			$D^{(2)}$		
	0	1	2	0	1	2	0	1	2	0	1	2
0	0	4	11	0	4	11	0	4	6	0	4	6
1	6	0	2	6	0	2	6	0	2	5	0	2
2	3	∞	0	3	7	0	3	7	0	3	7	0
P	$P^{(-1)}$			$P^{(0)}$			$P^{(1)}$			$P^{(2)}$		
	0	1	2	0	1	2	0	1	2	0	1	2
0		AB	AC		AB	AC		AB	ABC		AB	ABC
1	BA		BC	BA		BC	BA		BC	BCA		BC
2	CA			CA	CAB		CA	CAB		CA	CAB	

图 7.37　图 7.36 中有向图的各对顶点间的最短路径及其路径长度

www.TopSage.com

大家网
TopSage.com

第8章　动态存储管理

8.1　概　　述

在前面各章的讨论中，对每一种数据结构虽都介绍了它们在内存储器中的映像，但只是借助高级语言中的变量说明加以描述，并没涉及具体的存储分配。而实际上，结构中的每个数据元素都占有一定的内存位置，在程序执行的过程中，数据元素的存取是通过对应的存储单元来进行的。在早期的计算机上，这个存储管理的工作是由程序员自己来完成的。在程序执行之前，首先需将用机器语言或汇编语言编写的程序输送到内存的某个固定区域上，并预先给变量和数据分配好对应的内存地址(绝对地址或相对地址)。在有了高级语言之后，程序员不需要直接和内存地址打交道，程序中使用的存储单元都由逻辑变量(标识符)来表示，它们对应的内存地址都是由编译程序在编译或执行时进行分配。

另一方面，当计算机是被单个用户使用时，那么整个内存除操作系统占用一部分之外，都归这个用户的程序使用(如 PDP-11/03 的内存为 32KB，系统占用 4KB，用户程序可用 28KB)。但在多用户分时并发系统中，多个用户程序共享一个内存区域，此时每个用户程序使用的内存就由操作系统来进行分配了。并且，在总的内存不够使用时，还可采用自动覆盖技术。

对操作系统和编译程序来说，存储管理都是一个复杂而又重要的问题。不同语言的编译程序和不同的操作系统可以采用不同的存储管理方法。它们采用的具体做法，读者将在后续课程——编译原理和操作系统中学习。本课程仅就动态存储管理中涉及的一些基本技术进行讨论。

动态存储管理的基本问题是系统如何应用户提出的“请求”分配内存？又如何回收那些用户不再使用而“释放”的内存，以备新的“请求”产生时重新进行分配？提出请求的用户可能是进入系统的一个作业，也可能是程序执行过程中的一个动态变量。因此，在不同的动态存储管理系统中，请求分配的内存量大小不同。通常在编译程序中是一个或几个字，而在系统中则是几千、几万，甚至是几十万。然而，系统每次分配给用户(不论大小)都是一个地址连续的内存区。为了叙述方便起见，在下面的讨论中，将统称已分配给用户使用的地址连续的内存区为“占用块”，称未曾分配的地址连续的内存区为“可利用空间块”或“空闲块”。

显然，不管什么样的动态存储管理系统，在刚开工时，整个内存区是一个“空闲块”(在编译程序中称之为“堆”)。随着用户进入系统，先后提出存储请求，系统则依次进行分配。因此，在系统运行的初期，整个内存区基本上分隔成两大部分：低地址区包含若干占用块；高地址区(即分配后的剩余部分)是一个“空闲块”。例如图8.1(a)所示为依次给 8 个用户进行分配后的系统的内存状态。经过一段时间以后，有的用户运行结束，它所占用的内存区变成空闲块，这就使整个内存区呈现出占用块和空闲块犬牙交错的状态。如图 8.1(b)

www.TopSage.com

所示。

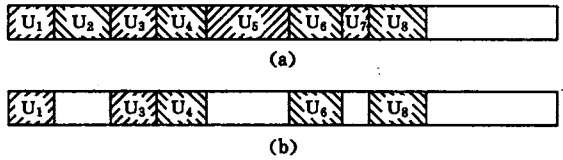
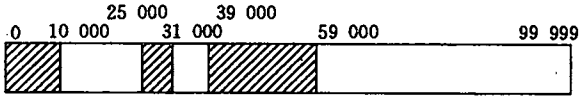


图 8.1　动态存储分配过程中的内存状态

(a) 系统运行初期；(b) 系统运行若干时间之后

假如此时又有新的用户进入系统请求分配内存，那么，系统将如何做呢？

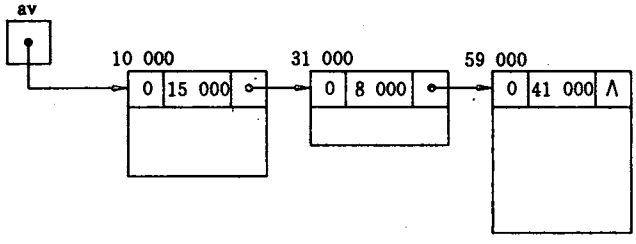
通常有两种做法：一种策略是系统继续从高地址的空闲块中进行分配，而不理会已分配给用户的内存区是否已空闲，直到分配无法进行（即剩余的空闲块不能满足分配的请求）时，系统才去回收所有用户不再使用的空闲块，并且重新组织内存，将所有空闲的内存区连接在一起成为一个大的空闲块。另一种策略是用户一旦运行结束，便将它所占内存区释放成为空闲块，同时，每当新的用户请求分配内存时，系统需要巡视整个内存区中所有空闲块，并从中找出一个“合适”的空闲块分配之。由此，系统需建立一张记录所有空闲块的“可利用空间表”，此表的结构可以是“目录表”，也可以是“链表”。如图 8.2 所示为某



(a)

起始地址	内存块大小	使用情况
10 000	15 000	空闲
31 000	8 000	空闲
59 000	41 000	空闲

(b)



(c)

图 8.2　动态存储管理过程中的内存状态和可利用空间表

(a) 内存状态；(b) 目录表；(c) 链表

系统运行过程中的内存状态及其两种结构的可利用空间表。其中图 8.2(b)是目录表，表中每个表目包括 3 项信息：初始地址、空闲块大小和使用情况。图 8.2(c)是链表，表中一个结点表示一个空闲块，系统每次进行分配或回收即为在可利用空间表中删除或插入一个结点。

www.TopSage.com

大家网
TopSage.com

下面将分别讨论利用不同策略进行动态存储管理的方法。

8.2 可利用空间表及分配方法

这一节主要讨论利用可利用空间表进行动态存储分配的方法。目录表的情况比较简单，这类系统将在操作系统课程中作详细介绍，在此仅就链表的情况进行讨论。

如上所述，可利用空间表中包含所有可分配的空闲块，每一块是链表中的一个结点。当用户请求分配时，系统从可利用空间表中删除一个结点分配之；当用户释放其所占内存时，系统即回收并将它插入到可利用空间表中。因此，可利用空间表亦称做“存储池”。根据系统运行的不同情况，可利用空间表可以有下列 3 种不同的结构形式：

第一种情况是系统运行期间所有用户请求分配的存储量大小相同。对此类系统，通常的做法是，在系统开始运行时将归它使用的内存区按所需大小分割成若干大小相同的块，然后用指针链接成一个可利用空间表。由于表中结点大小相同，则分配时无需查找，只要将第一个结点分配给用户即可；同样，当用户释放内存时，系统只要将用户释放的空闲块插入在表头即可。可见，这种情况下的可利用空间表实质上是一个链栈。这是一种最简单的动态存储管理的方式，如第 2 章的“2.3.1 线性链表”中的静态链表就是一例。

第二种情况，系统运行期间用户请求分配的存储量有若干种大小的规格。对此类系统，一般情况下是建立若干个可利用空间表，同一链表中的结点大小相同。例如，某动态存储管理系统中的用户将请求分配 2 个字、4 个字或 8 个字的内存块，则系统建立 3 个结点大小分别为 3 个字、5 个字和 9 个字的链表，它们的表头指针分别为 av2、av4 和 av8。如图 8.3 所示，每个结点中的第一个字设有链域(link)、标志域(tag)和结点类型域(type)。其中：类型域为区别 3 种大小不同的结点而设，type 的值为“0”、“1”或“2”，分别表示结点大小为 2 个字、4 个字或 8 个字；标志域 tag 为“0”或“1”分别表示结点为空闲块或占用块；链域中存储指向同一链表中下一结点的指针，而结点中的值域是其大小分别为 2、4 和 8 个字的连续空间。此时的分配和回收的方法在很大程度上和第一种情况类似，只是当结点大小和请求分配的量相同的链表为空时，需查询结点较大的链表，并从中取出一个结点，将其中一部分内存分配给用户，而将剩余部分插入到相应大小的链表中。回收时，也只要将释放的空闲块插入到相应大小的链表的表头中去即可。然而，这种情况的系统还有一个特殊的问题要处理：即当结点与请求相符的链表和结点更大的链表均为空时，分配不能进行，而实际上内存空间并不一定不存在所需大小的连续空间，只是由于在系统运行过程中，频繁出现小块的分配和回收，使得大结点链表中的空闲块被分隔成小块后插入在小结点的链表中，此时若要使系统能继续运行，就必须重新组织内存，即执行“存储紧缩”的操作。除此之外，上述这个系统本身的分配和回收的算法都比较简单，读者可自行写出。

第 3 种情况，系统在运行期间分配给用户的内存块的大小不固定，可以随请求而变。因此，可利用空间表中的结点即空闲块的大小也是随意的。通常，操作系统中的可利用空间表属这种类型。

系统刚开始工作时，整个内存空间是一个空闲块，即可利用空间表中只有一个大小为

www.TopSage.com

整个内存区的结点，随着分配和回收的进行，可利用空间表中的结点大小和个数也随之而变，上述图 8.2(c)中的链表即为这种情况的可利用空间表。

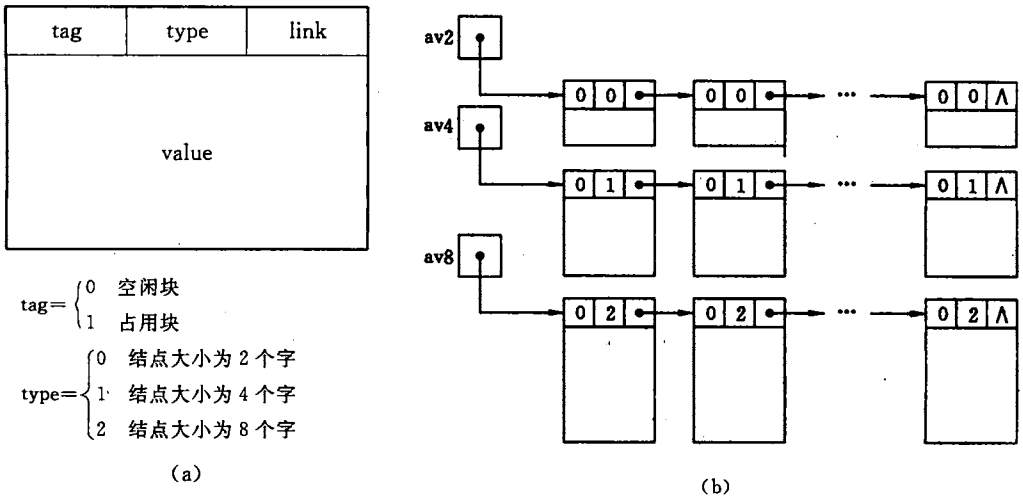


图 8.3　有 3 种大小结点的可利用空间表

(a) 结点结构；(b) 可利用空间表

由于链表中结点大小不同，则结点的结构与前两种情况也有所不同，结点中除标志域和链域之外，尚需有一个结点大小域(size)，以指示空闲块的存储量，如图 8.4 所示。结点中的 space 域是一个地址连续的内存空间。

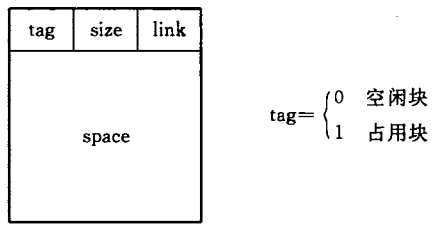


图 8.4　空闲块的大小随意的结点结构

由于可利用空间表中的结点大小不同，则在分配时就有一个如何分配的问题。假设某用户需大小为 n 的内存，而可利用空间表中仅有一块大小为 $m \geqslant n$ 的空闲块，则只需将其中大小为 n 的一部分分配给申请分配的用户，同时将剩余大小为 $m-n$ 的部分作为一个结点留在链表中即可。然而，若可利用空间表中有若干个不小于 n 的空闲块时，该分配哪一块呢？通常，可有 3 种不同的分配策略：

(1) **首次拟合法**。从表头指针开始查找可利用空间表，将找到的第一个大小不小于 n 的空闲块的一部分分配给用户。可利用空间表本身既不按结点的初始地址有序，也不按结点的大小有序。则在回收时，只要将释放的空闲块插入在链表的表头即可。例如，在图 8.2(c)的状态时有用户 U_9 进入系统并申请 7KB 的内存，系统在可利用空间表中进行查询，发现第一个空闲块即满足要求，则将此块中大小为 7KB 的一部分分配之，剩余 8KB 的空闲块仍在链表中，如图 8.5(a)所示。图 8.5(d)为分配给用户的占用块。

www.TopSage.com

大家网 TopSage.com

(2) **最佳拟合法**。将可利用空间表中一个不小于 n 且最接近 n 的空闲块的一部分分配给用户。则系统在分配前首先要对可利用空间表从头到尾扫视一遍，然后从中找出一块不小于 n 且最接近 n 的空闲块进行分配。显然，在图 8.2(c)的状态时，系统就应该将第二个空闲块的一部分分配给用户 U_9，分配后的可利用空间表如图 8.5(b)所示。在用最佳拟合法进行分配时，为了避免每次分配都要扫视整个链表。通常，预先设定可利用空间表的结构按空间块的大小自小至大有序，由此，只需找到第一块大于 n 的空闲块即 进行分配，但在回收时，必须将释放的空闲块插入到合适的位置上去。

(3) **最差拟合法**。将可利用空间表中不小于 n 且是链表中最大的空闲块的一部分分配给用户。例如在图 8.2(c)的状态时，就应将大小为 41KB 的空闲块中的一部分分配给

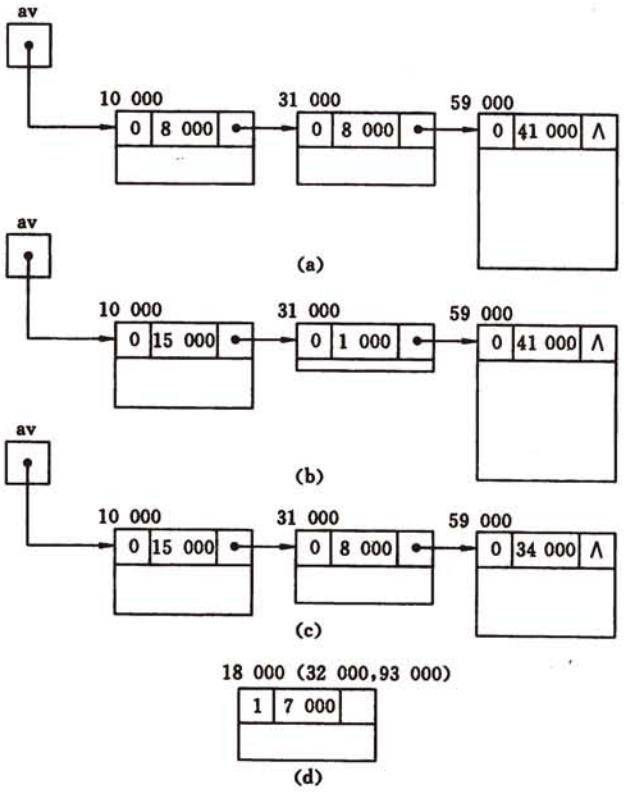


图 8.5　结点大小随意的可利用空间表

(a) 按首次拟合原则进行分配；(b) 按最佳拟合原则进行分配；

(c) 按最差拟合原则进行分配；(d) 分配给用户的占用块

用户，分配后的可利用空间表如图 8.5(c)所示。显然，为了节省时间，此时的可利用空间表的结构应按空闲块的大小自大至小有序。这样，每次分配无需查找，只需从链表中删除第一个结点，并将其中一部分分配给用户，而剩余部分作为一个新的结点插入到可利用空间表的适当位置上去。当然，在回收时亦需将释放的空闲块插入到链表的适当位置上去。

上述 3 种分配策略各有所长。一般来说，最佳拟合法适用于请求分配的内存大小范围较广的系统。因为按最佳拟合的原则进行分配时，总是找大小最接近请求的空闲块，由

www.TopSage.com

此系统中可能产生一些存储量甚小而无法利用的小片内存，同时也保留那些很大的内存块以备响应后面将发生的内存量特大的请求，从而使整个链表趋向于结点大小差别甚远的状态。反之，由于最差拟合每次都从内存量最大的结点中进行分配，从而使链表中的结点大小趋于均匀，因此它适用于请求分配的内存大小范围较窄的系统。而首次拟合法的分配是随机的，因此它介于两者之间，通常适用于系统事先不掌握运行期间可能出现的请求分配和释放的信息的情况。从时间上来比较，首次拟合在分配时需查询可利用空间表，而回收时仅需插入在表头即可；最差拟合恰相反，分配时无需查询链表，而回收时为将新的“空闲块”插入在链表中适当的位置上，需先进行查找；最佳拟合无论分配和回收，均需查找链表，因此最费时间。

因此，不同的情景需采用不同的方法，通常在选择时需考虑下列因素：用户的逻辑要求；请求分配量的大小分布；分配和释放的频率以及效率对系统的重要性等等。

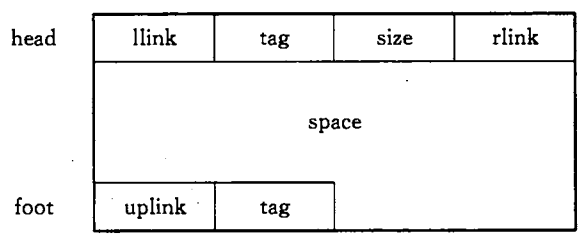
在实际使用的系统中回收空闲块时还需考虑一个“结点合并”的问题。这是因为系统在不断进行分配和回收的过程中，大的空闲块逐渐被分割成小的占用块，在它们重又成为空闲块回收之后，即使是地址相邻的两个空闲块也只是作为两个结点插入到可利用空间表中，以致使得后来出现的大容量的请求分配无法进行，为了更有效地利用内存，就要求系统在回收时应考虑将地址相邻的空闲块合并成尽可能大的结点。换句话说，在回收空闲块时，首先应检查地址与它相邻的内存是否是空闲块。具体实现的方法将在下面两节中讨论的动态存储管理系统中加以详细说明。

8.3 边界标识法

边界标识法(boundary tag method)是操作系统中用以进行动态分区分配的一种存储管理方法，它属于上一节讨论中的第三种情况。系统将所有的空闲块链接在一个双重循环链表结构的可利用空间表中；分配可按首次拟合进行，也可按最佳拟合进行。系统的特点在于：在每个内存区的头部和底部两个边界上分别设有标识，以标识该区域为占用块或空闲块，使得在回收用户释放的空闲块时易于判别在物理位置上与其相邻的内存区域是否为空闲块，以便将所有地址连续的空闲存储区组合成一个尽可能大的空闲块。下面分别就系统的可利用空间表的结构及其分配和回收的算法进行讨论。

8.3.1 可利用空间表的结构

可利用空间表中的结点结构如下所示：



www.TopSage.com

大家网 TopSage.com

它表示一个空闲块。整个结点由 3 部分组成。其中 space 为一组地址连续的存储单元，是可以分配给用户使用的内存区域，它的大小由 head 中的 size 域指示，并以头部 head 和底部 foot 作为它的两个边界；在 head 和 foot 中分别设有标志域 tag，且设定空闲块中 tag 的值为“0”，占用块中 tag 的值为“1”；foot 位于结点底部，因此它的地址是随结点中 space 空间的大小而变的。为讨论简便起见，我们假定内存块的大小以“字”为单位来计，地址也以“字”为单位来计，结点头部中的 size 域的值为整个结点的大小，包括头部 head 和底部 foot 所占空间，并假设 head 和 foot 各占一个字的空间，但在分配时忽略不计。

借助 C 语言，在此将可利用空间表的结点结构定义为如下说明的数据类型：

```
typedef struct WORD {                    // WORD:内存字类型
  union {   // head 和 foot 分别是结点的第一个字和最后的字
    WORD    *llink;                      // 头部域,指向前驱结点
    WORD    *uplink;                     // 底部域,指向本结点头部
  };
  int        tag;                        // 块标志,0:空闲,1:占用,头部和尾部均有。
  int        size;                       // 头部域,块大小
  WORD       *rlink;                     // 头部域,指向后继结点
  OtherType  other;                      // 字的其他部分
}WORD, head, foot, *Space;               // *Space:可利用空间指针类型

#define  FootLoc(p)  p+p->size-1         // 指向 p 所指结点的底部
```

可利用空间表设为双重循环链表。head 中的 llink 和 rlink 分别指向前驱结点和后继结点。表中不设表头结点，表头指针 pav 可以指向表中任一结点，即任何一个结点都可看成是链表中的第一个结点；表头指针为空，则表明可利用空间表为空。foot 中的 uplink 域也为指针，它指向本结点，它的值即为该空闲块的首地址。例如图 8.6(a)是一个占有 100KB 内存空间的系统在运行开始时的可利用空间表。

8.3.2 分配算法

分配的算法比较简单，假设我们采用首次拟合法进行分配，则只要从表头指针 pav 所指结点起，在可利用空间表中进行查找，找到第一个容量不小于请求分配的存储量(n)的空闲块时，即可进行分配。为了使整个系统更有效地运行，在边界标识法中还作了如下两条约定：

(1) 假设找到的此块待分配的空闲块的容量为 m 个字(包括头部和底部)，若每次分配只是从中分配 n 个字给用户，剩余 $m-n$ 个字大小的结点仍留在链表中，则在若干次分配之后，链表中会出现一些容量极小总也分配不出去的空闲块，这就大大减慢了分配(查找)的速度。弥补的办法是：选定一个适当的常量 e，当 $m-n \leqslant e$ 时，就将容量为 m 的空闲块整块分配给用户；反之，只分配其中 n 个字的内存块。同时，为了避免修改指针，约定将该结点中的高地址部分分配给用户。

(2) 如果每次分配都从同一个结点开始查找的话，势必造成存储量小的结点密集在头指针 pav 所指结点附近，这同样会增加查询较大空闲块的时间。反之，如果每次分配从

www.TopSage.com

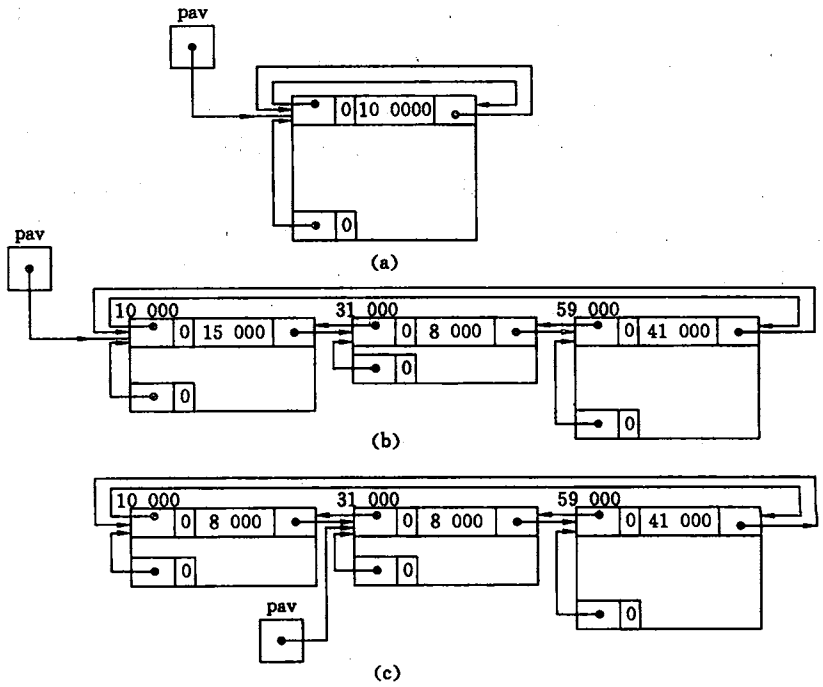


图 8.6 某系统的可利用空间表

(a) 初始状态；(b) 运行若干时间后的状态；(c) 进行再分配后的状态

不同的结点开始进行查找，使分配后剩余的小块均匀地分布在链表中，则可避免上述弊病。实现的方法是，在每次分配之后，令指针 pav 指向刚进行过分配的结点的后继结点。

例如，图 8.6(b)所示可利用空间表在进行分配之后的状态如图 8.6(c)所示。

算法 8.1 是上述分配策略的算法描述。

```
Space AllocBoundTag (Space &pav, int n) {
  // 若有不小于 n 的空闲块，则分配相应的存储块，并返回其首地址；否则返回
  // NULL。若分配后可利用空间表不空，则 pav 指向表中刚分配过的结点的后继
  // 结点
  for (p = pav; p && p->size<n && p->rlink! = pav;
       p = p->rlink);                          // 查找不小于 n 的空闲块
  if ( !p || p->size < n )  return NULL;       // 找不到，返回空指针
  else {                                       // p 指向找到的空闲块
     f = FootLoc(p);                           // 指向底部
     pav = p->rlink;                           // pav 指向 * p 结点的后继结点。
     if ( p->size - n <= e ) {                 // 整块分配，不保留 <= e 的剩余量
        if (pav == p)  pav = NULL;             // 可利用空间表变为空表
        else {                                 // 在表中删除分配的结点
           pav->llink = p->llink;  p->llink->rlink = pav;
        }//if
        p->tag = f->tag = 1;                   // 修改分配结点的头部和底部标志
     }//if
     else {                                    // 分配该块的后 n 个字
```


www.TopSage.com

大家网
TopSage.com

```
            f->tag = 1;                          // 修改分配块的底部标志
            p->size - = n;                       // 置剩余块大小
            f = FootLoc(p);                      // 指向剩余块底部
            f->tag = 0;   f->uplink = p;         // 设置剩余块底部
            p = f+1;                             // 指向分配块头部
            p->tag = 1;   p->size = n;           // 设置分配块头部
        }
      return p;                                  // 返回分配块首地址
     }// else
}// AllocBoundTag
```

算法 8.1

8.3.3 回收算法

一旦用户释放占用块，系统需立即回收以备新的请求产生时进行再分配。为了使物理地址毗邻的空闲块结合成一个尽可能大的结点，则首先需要检查刚释放的占用块的左、右紧邻是否为空闲块。由于本系统在每个内存区(无论是占用块或空闲块)的边界上都设有标志值，则很容易辨明这一点。

假设用户释放的内存区的头部地址为p，则与其低地址紧邻的内存区的底部地址为p－1；与其高地址紧邻的内存区的头部地址为 p＋p->size，它们中的标志域就表明了这两个邻区的使用状况：若(p－1)->tag＝0；则表明其左邻为空闲块，若(p＋p->size)->tag＝0；则表明其右邻为空闲块。

若释放块的左、右邻区均为占用块，则处理最为简单，只要将此新的空闲块作为一个结点插入到可利用空闲表中即可；若只有左邻区是空闲块，则应与左邻区合并成一个结点；若只有右邻区是空闲块，则应与右邻区合并成一个结点；若左、右邻区都是空闲块，则应将 3 块合起来成为一个结点留在可利用空间表中，下面我们就这 4 种情况分别描述它们的算法：

(1) 释放块的左、右邻区均为占用块。此时只要作简单插入即可。由于边界标识法在按首次拟合进行分配时对可利用空间表的结构没有任何要求，则新的空闲块插入在表中任何位置均可。简单的做法就是插入在 pav 指针所指结点之前(或之后)，可描述如下：

```
p->tag = 0;   FootLoc(p)->uplink = p;   FootLoc(p)->tag = 0;
if (!pav)  pav = p->llink = p->rlink = p;
else {  q = pav->llink;
        p->rlink = pav;   p->llink = q;
        q->rlink = pav->llink = p;
        pav = p;     // 令刚释放的结点为下次分配时的最先查询的结点
}
```

(2) 释放块的左邻区为空闲块，而右邻区为占用块。由于释放块的头部和左邻空闲块的底部毗邻，因此只要改变左邻空闲块的结点：增加结点的 size 域的值且重新设置结点的底部即可。描述如下：

```
n = p->size;                                 // 释放块的大小
s = (p-1)->uplink;                           // 左邻空闲块的头部地址
```


www.TopSage.com

```
s->size + = n;                                    // 设置新的空闲块大小
f = p+n-1;   f->uplink = s;   f->tag = 0;         // 设置新的空闲块底部
```

(3) 释放块的右邻区为空闲块，而左邻区为占用块。由于释放块的底部和右邻空闲块的头部毗邻，因此，当表中结点由原来的右邻空闲块变成合并后的大空闲块时，结点的底部位置不变，但头部要变，由此，链表中的指针也要变。描述如下：

```
t = p+p->size;       // 右邻空闲块的头部地址
p->tag = 0;          // p为合并后的结点头部地址
q = t->llink;        // q为*t结点在可利用空间表中的前驱结点的头部地址
p->llink = q;   q->rlink = p;        // q指向*p的前驱
ql = t->rlink;              // ql为*t结点在可利用空间表中的后继结点的头部地址
p->rlink = ql;   ql->llink = p;      // ql指向*p的后继
p->size + = t->size;                 // 新的空闲块的大小
FootLoc(t)->uplink = p;              // 底部指针指向新结点的头部
```

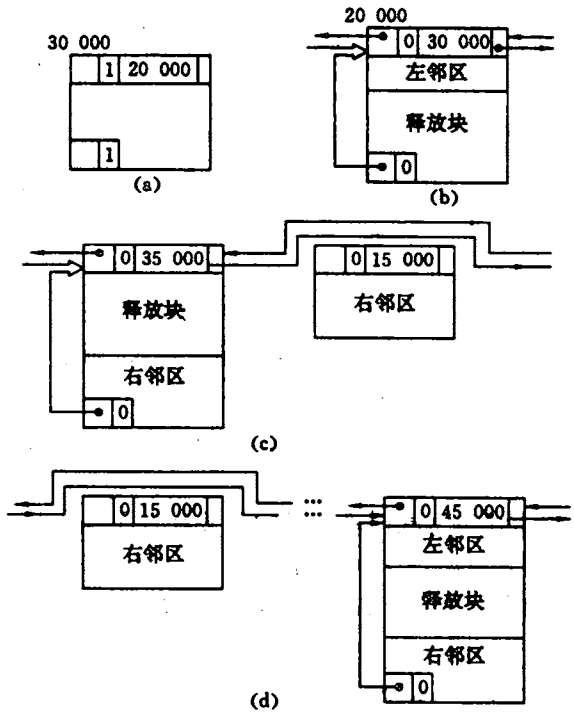


图 8.7　回收存储块后的可利用空间表

(a) 释放的存储块；　(b) 左邻区是空闲块的情况；

(c) 右邻区是空闲块的情况；　(d) 左、右邻区均是空闲块的情况

(4) 释放块的左、右邻区均为空闲块。为使 3 个空闲块连接在一起成为一个大结点留在可利用空间表中，只要增加左邻空闲块的 space 容量，同时在链表中删去右邻空闲块结点即可。所作改变可描述如下：

```
n = p->size;                         // 释放块的大小
s = (p-1)->uplink;                   // 指向左邻块
```


www.TopSage.com

大家网
TopSage.com

```
t = p+p->size;                          // 指向右邻块
s->size = n+t->size;                    // 设置新结点的大小
q = t->llink;   ql = t->rlink;          // q! = ql
q->rlink = ql;  ql->llink = q;          // 删去右邻空闲块结点
FootLoc(t)->uplink = s;                 // 新结点底部指针指向其头部
```

总之，边界标识法由于在每个结点的头部和底部设立了标识域，使得在回收用户释放的内存块时，很容易判别与它毗邻的内存区是否是空闲块，且不需要查询整个可利用空间表便能找到毗邻的空闲块与其合并之；再者，由于可利用空间表上结点既不需依结点大小有序，也不需依结点地址有序，则释放块插入时也不需查找链表。由此，不管是哪一种情况，回收空闲块的时间都是个常量，和可利用空间表的大小无关。惟一的缺点是增加了结点底部所占的存储量。

在上述后 3 种情况下，可利用空间表的变化如图 8.7 所示。

8.4 伙伴系统

伙伴系统(buddy system)是操作系统中用到的另一种动态存储管理方法。它和边界标识法类似，在用户提出申请时，分配一块大小“恰当”的内存区给用户；反之，在用户释放内存区时即回收。所不同的是：在伙伴系统中，无论是占用块或空闲块，其大小均为 2 的 k 次幂(k 为某个正整数)。例如：当用户申请 n 个字的内存区时，分配的占用块大小为 2^k 个字($2^{k-1}<n\leqslant 2^k$)。由此，在可利用空间表中的空闲块大小也只能是 2 的 k 次幂。若总的可利用内存容量为 2^m 个字，则空闲块的大小只可能为 2^0、2^1、…、2^m。下面我们仍和上节一样，分 3 个问题来介绍这个系统。

8.4.1 可利用空间表的结构

假设系统的可利用内存空间容量为 2^m 个字(地址从 0 到 2^m-1)，则在开始运行时，整个内存区是一个大小为 2^m 的空闲块，在运行了一段时间之后，被分隔成若干占用块和空闲块。为了再分配时查找方便起见，我们将所有大小相同的空闲块建于一张子表中。每个子表是一个双重链表，这样的链表可能有 $m+1$ 个，将这 $m+1$ 个表头指针用向量结构组织成一个表，这就是伙伴系统中的可利用空间表。

双重链表中的结点结构如图 8.8(a)所示，其中 head 为结点头部，是一个由 4 个域组成的记录，其中的 llink 域和 rlink 域分别指向同一链表中的前驱和后继结点；tag 域为值取“0”“1”的标志域，kval 域的值为 2 的幂次 k；space 是一个大小为 2^k-1 个字的连续内存空间(和前面类似，仍假设 head 占一个字的空间)。

可利用空间表的初始状态如图 8.8(b)所示，其中 m 个子表都为空表，只有大小为 2^m 的链表中有一个结点，即整个存储空间。表头向量的每个分量由两个域组成，除指针域外另设 nodesize 域表示该链表中空闲块的大小，以便分配时查找方便。此可利用空间表的数据类型，示意描述如下：

```
#define  m  16                  // 可利用空间总容量 64K 字的 2 的幂次，子表的个数为 m+1
typedef struct WORD_b {
```


www.TopSage.com

```
   WORD_b     *llink;          // 指向前驱结点
   int        tag;             // 块标志,0:空闲,1:占用。
   int        kval;            // 块大小,值为 2 的幂次 k
   WORD_b     *rlink;          // 头部域,指向后继结点
   OtherType  other;           // 字的其他部分
}WORD_b, head;                 // WORD:内存字类型,结点的第一个字也称为 head
typedef struct HeadNode {
   int     nodesize;          // 该链表的空闲块的大小
   WORD_b  *first;            // 该链表的表头指针
}FreeList[m+1];                // 表头向量类型
```

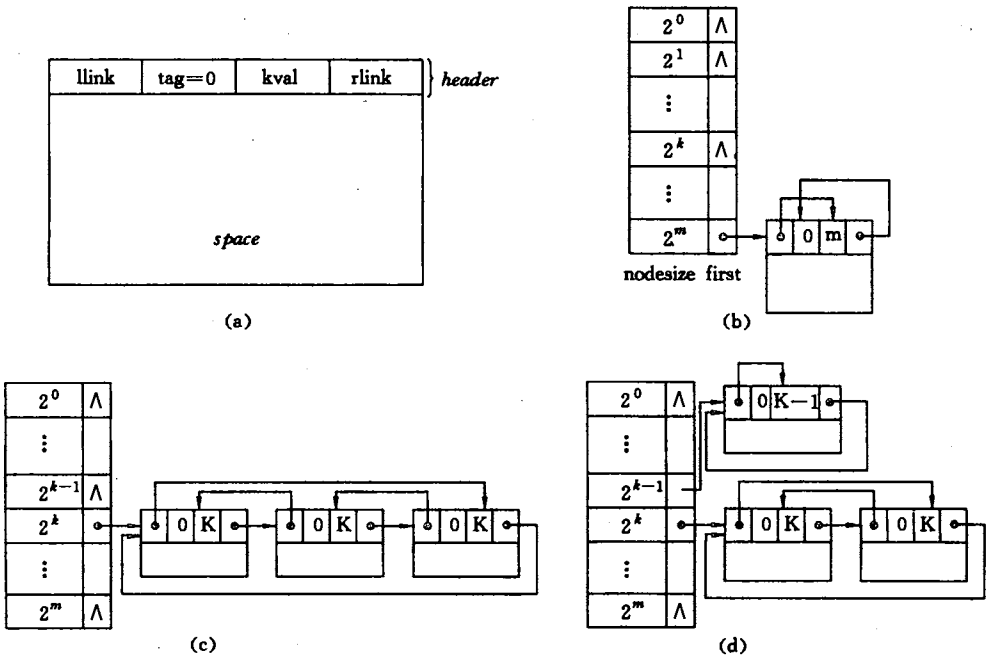


图 8.8 伙伴系统中的可利用空间表

(a) 空闲块的结点结构；(b) 表的初始状态；(c) 分配前的表；(d) 分配后的表

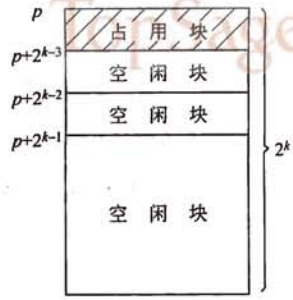
8.4.2 分配算法

当用户提出大小为 n 的内存请求时，首先在可利用表上寻找结点大小与 n 相匹配的子表，若此子表非空，则将子表中任意一个结点分配之即可；若此子表为空，则需从结点更大的非空子表中去查找，直至找到一个空闲块，则将其中一部分分配给用户，而将剩余部分插入相应的子表中。

假设分配前的可利用空间表的状态如图 8.8(c)所示。若 $2^{k-1}<n\leqslant 2^k-1$，又第 $k+1$ 个子表不空，则只要删除此链表中第一个结点并分配给用户即可；若 $2^{k-2}<n\leqslant 2^{k-1}-1$，此时由于结点大小为 2^{k-1} 的子表为空，则需从结点大小为 2^k 的子表中取出一块，将其中一半分配给用户，剩余的一半作为一个新结点插入在结点大小为 2^{k-1} 的子表中，如图8.8(d)所示。若 $2^{k-i-1}<n\leqslant 2^{k-i}-1$($i$ 为小于 k 的整数)，并且所有结点小于 2^k 的子表均为空，

www.TopSage.com

则同样需从结点大小为 2^k 的子表中取出一块，将其中 2^{k-i} 的一小部分分配给用户，剩余部分分割成若干个结点分别插入在结点大小为 2^{k-i}、2^{k-i+1}、…、2^{k-1} 的子表中。假设从第 $k+1$ 个子表中删除的结点的起始地址为 p，且假设分配给用户的占用块的初始地址为 p（占用块为该空闲块的低地址区），则插入上述子表的新结点的起始地址分别为 $p+2^{k-i}$、$p+2^{k-i+1}$、…、$p+2^{k-1}$，如右图所示（图中 $i=3$）。



下面用算法语言描述之：

```
WORD_b* AllocBuddy (FreeList &avail, int n) {
  // avail[0..m]为可利用空间表,n 为申请分配量，若有不小于 n 的空闲块，
  // 则分配相应的存储块,并返回其首地址;否则返回 NULL
  for (k=0; k<=m && ( avail[k].nodesize<n+1 || !avail[k].first);
       ++k);                                  // 查找满足分配要求的子表
  if (k>m) return NULL;                       // 分配失败,返回 NULL
  else {                                      // 进行分配
    pa = avail[k].first;                      // 指向可分配子表的第一个结点
    pre = pa->llink;   suc = pa->rlink;       // 分别指向前驱和后继
    if (pa==suc) avail[k].first = NULL;       // 分配后该子表变成空表
    else {                                    // 从子表删去 *pa 结点
      pre->rlink = suc;   suc->llink = pre;   avail[k].first = suc;
    }
    for (i=1;  avail[k-i].nodesize>=n+1;  ++i) {
      pi = pa+2^(k-i);     pi->rlink = pi;    pi->llink = pi;
      pi->tag = 0;     pi->kval = k-i;     avail[k-i].first = pi;
    }  // 将剩余块插入相应子表
    pa->tag = 1;   pa->kval = k-(--i);
  }
  return pa;
}// AllocBuddy
```

算法 8.2

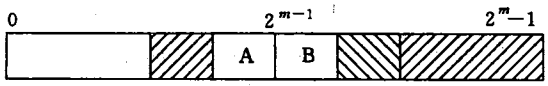
8.4.3 回收算法

在用户释放不再使用的占用块时，系统需将这新的空闲块插入到可利用空间表中去。这里，同样有一个地址相邻的空闲块归并成大块的问题。但是在伙伴系统中仅考虑互为“伙伴”的两个空闲块的归并。

何谓“伙伴”？如前所述，在分配时经常需要将一个大的空闲块分裂成两个大小相等的存储区，这两个由同一大块分裂出来的小块就称之“互为伙伴”。例如：假设 p 为大小为 2^k 的空闲块的初始地址，且 p MOD $2^{k+1}=0$，则初始地址为 p 和 $p+2^k$ 的两个空闲块互为伙伴。在伙伴系统中回收空闲块时，只当其伙伴为空闲块时才归并成大块。也就是说，若有两个空闲块，即使大小相同且地址相邻，但不是由同一大块分裂出来的，也不归并在一起。例如图中的 A、B 两个空闲块不是伙伴。

由此，在回收空闲块时，应首先判别其伙伴是否为空闲块，若否，则只要将释放的空闲

www.TopSage.com



块简单插入在相应子表中即可；若是，则需在相应子表中找到其伙伴并删除之，然后再判别合并后的空闲块的伙伴是否是空闲块。依此重复，直到归并所得空闲块的伙伴不是空闲块时，再插入到相应的子表中去。

起始地址为 p，大小为 2^k 的内存块，其伙伴块的起始地址为：

$$\text{buddy}(p,k)=\begin{cases}p+2^k & (\text{若 } p \text{ MOD } 2^{k+1}=0)\\ p-2^k & (\text{若 } p \text{ MOD } 2^{k+1}=2^k)\end{cases}$$

例如，假设整个可利用内存区大小为 $2^{10}=1\,024$（地址从 0 到 1 023），则大小为 2^8，起始地址为 512 的伙伴块的起始地址为 768；大小为 2^7，起始地址为 384 的伙伴块的起始地址为 256。

整个释放算法在此不再详细列出，请读者自行补充。

总之，伙伴系统的优点是算法简单、速度快；缺点是由于只归并伙伴而容易产生碎片。

8.5 无用单元收集

以上 3 节讨论的问题都是如何利用可利用空间表来进行动态存储管理。它的特点是：在用户请求存储时进行分配；在用户释放存储时进行回收，即系统是应用户的需求来进行存储分配和回收的。因此，在这类存储管理系统中，用户必须明确给出“请求”和“释放”的信息。如在多用户分时并发的操作系统中，当用户程序进入系统时即请求分配存储区；反之，当用户程序执行完毕退出系统时即释放所占存储。又如，在使用 C 语言编写程序时，用户是通过 malloc 和 free 两个函数来表示请求分配和释放存储的。但有时会因为用户的疏漏或结构本身的原因致使系统在不恰当的时候或没有进行回收而产生“无用单元”或“悬挂访问”的问题。

“无用单元”是指那些用户不再使用而系统没有回收的结构和变量。例如下列 C 程序段

```
p = malloc(size);
⋮
p = NULL;
```

执行的结果，是使执行 p=malloc(size)为用户分配的结点成为无用单元，无法得到利用；而下列程序段

```
p = malloc(size);
⋮
q = p;
free(p);
```

执行的结果使指针变量 q 悬空，如果所释放的结点被再分配而继续访问指针 q 所指结点，则称这种访问为“悬挂访问”，并且由此引起的恶劣后果是可想而知的。

www.TopSage.com

大家网
TopSage.com

另一方面，由于结构本身的某些特性，也会产生同上类似问题。

例如在某用户程序中有 3 个广义表结构，如图 8.9 所示，L_1、L_2 和 L_3 分别为它们的表头指针，L_4 是 L_1 和 L_2 共享的子表，L_3 本身又为 L_2 共享，则 L_5 为 3 个广义表所共享。

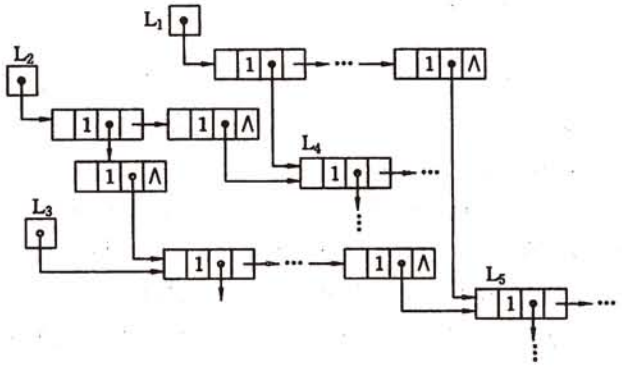


图 8.9　含有共享子表的广义表

在这种情况下，表结点的释放就成为一个问题。假设表 L_1 不再使用，而表 L_2 和 L_3 尚在使用，若释放表 L_1，即自 L_1 指针起，顺链将所有结点回收到可利用空间表中(包括子表 L_4 和 L_5 上所有结点)，这就破坏了表 L_2 和 L_3，从而产生“悬挂访问”；反之，若不将表 L_1 中结点释放，则当 L_2 和 L_3 两个表也不被使用时，这些结点由于未曾“释放”无法被再分配而成为“无用单元”。

如何解决这个问题？有两条途径：

(1) 使用访问计数器：在所有子表或广义表上增加一个表头结点，并设立一个“计数域”，它的值为指向该子表或广义表的指针数目。只有当该计数域的值为零时，此子表或广义表中结点才被释放。

(2) 收集无用单元：在程序运行的过程中，对所有的链表结点，不管它是否还有用，都不回收，直到整个可利用空间表为空。此时才暂时中断执行程序，将所有当前不被使用的结点链接在一起，成为一个新的可利用空间表，而后程序再继续执行。显然，在一般情况下，是无法辨别哪些结点是当前未被使用的。然而，对于一个正在运行的程序，哪些结点正在使用是容易查明的，这只要从所有当前正在工作的指针变量出发，顺链遍历，那么，所有链结在这些链上的结点都是占用的。反之，可利用存储空间中的其余结点就都是无用的了。

由此，收集无用单元应分两步进行：第一步是对所有占用结点加上标志。回顾第 5 章的广义表的存储结构可在每个结点上再加设一个标志(mark)域，假设在无用单元收集之前所有结点的标志域均置为“0”，则加上标志就是将结点的标志域置为“1”；第二步是对整个可利用存储空间顺序扫描一遍，将所有标志域为“0”的结点链接成一个新的可利用空间表。值得注意的是：上述第二步是容易进行的，而第一步是在极其困难的条件(即可利用存储几乎耗用殆尽)下进行的，因此，人们的精力主要集中在研究标志算法上。下面我们介绍 3 种标志算法。

www.TopSage.com

(1) 递归算法　从上面所述可知，加标志的操作实质上是遍历广义表，将广义表中所有结点的标志域赋值“1”。我们可写出遍历(加标志)算法的递归定义如下：

若列表为空，则无需遍历；若是一个数据元素，则标志元素结点；反之，则列表非空，首先标志表结点；然后分别遍历表头和表尾。

这个算法很简单，易于用允许递归的高级语言描述之。但是，它需要一个较大的实现递归用的栈的辅助内存，这部分内存不能用于动态分配。并且，由于列表的层次不定，使得栈的容量不易确定，除非是在内存区中开辟一个相当大的区域留作栈，否则就有可能由于在标志过程中因栈的溢出而使系统瘫痪。

(2) 非递归算法　程序中附设栈(或队列)实现广义表的遍历。从广义表的存储结构来看，表中有两种结点：一种是元素结点，结点中没有指针域；另一种是表结点，结点中包含两个指针域：表头指针和表尾指针，则它很类似于二叉树的二叉链表。列表中的元素结点相当于二叉树中的叶子结点，可以类似于遍历二叉树写出遍历表的非递归算法，只是在算法中应尽量减少栈的容量。

例如，类似于二叉树的前序遍历，对广义表则为：当表非空时，在对表结点加标志后，先顺表头指针逐层向下对表头加标志，同时将同层非空且未加标志的表尾指针依次入栈，直到表头为空表或为元素结点时停止，然后退栈取出上一层的表尾指针。反复上述进行过程，直到栈空为止。这个过程也可以称做深度优先搜索遍历。因为它和图的深度优先搜索遍历很相似。

显然，还可以类似于图的广度优先搜索遍历，对列表进行广度优先搜索遍历，或者说是对列表按层次遍历。同样，为实现这个遍历需附设一个队列(这两个算法和二叉树或图的遍历极为相似，故在此不作详细描述，读者完全可以自己写出)。在这两种非递归算法中，虽然附设的栈或队列的容量比递归算法中的栈的容量小，但和递归算法有同样的问题仍需要一个不确定量的附加存储，因此也不是理想的方法。

(3) 利用表结点本身的指针域标记遍历路径的算法　无论是在递归算法中还是在深度优先搜索的非递归算法中，不难看出，设栈的目的都是为了记下遍历时指针所走的路径，以便在遍历表头之后可以沿原路退回，继而对表尾进行遍历。如果我们能用别的方法记下指针所走路径，则可以免除附设栈。在下面介绍的算法中就是利用已经标志过的表结点中的 tag、hp 和 tp 域来代替栈记录遍历过程中的路径。例如：对图 8.10 中的广义表

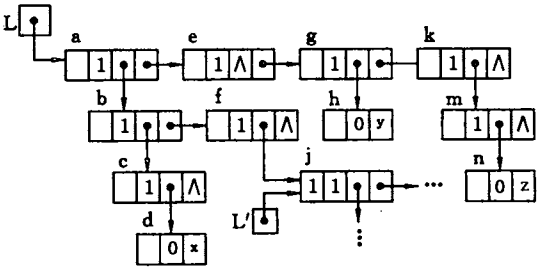


图 8.10　待遍历的广义表

www.TopSage.com

大家网 TopSage.com

加标志。假设在递归算法中指针 p 指向刚加上标志的 b 结点，则：①当指针 p 由 b 移向表头 c 之前需将 b 入栈(此时 a 已在栈中)；②在表头标志之后需退栈，然后指针 p 在由 b 移向表尾 f 时需再次将 b 入栈；③在 b 的表尾标志完之后应连续两次退栈，使 p 重又指向 a。与此对应，在本算法中不设栈。而是当指针 p 由 b 移向 c 之前，先将 b 结点中的 hp 域的值改为指向 a，并将 b 结点中的 tag 域的值改为“0”；而当指针 p 由 b 移向 f 之前，则先将 b 结点中的 tp 域的值改为指向 a，tag 域的值改为“1”。

下面详细叙述算法的基本思想(注：假设图 8.10 中的广义表 L′已加上标志)。

算法中设定了 3 个互相关联的指针：当 p 指向某个表结点时；t 指向 p 的母表结点；q 指向 p 的表头或表尾。如图 8.11 中(a)和(b)所示。

当 q 指向 p 的表头结点时，可能有 3 种情况出现：①设 p 的表头只是一个元素结点，则遍历表头仅需对该表头结点打上标志后即令 q 指向 p 的表尾；②设 p 的表头为空表或是已加上标志的子表，则无需遍历表头只要令 q 指向 p 的表尾即可；③设 p 的表头为未加标志的子表，则需先遍历表头子表，即 p 应赋 q 的值，t 相应往下移动改赋 p 的值。为了记下 t 指针移动的路径，以便在 p 退回原结点时同时能找到 p 的母表结点(即使 t 退回到原来的值)，则在修改这个指针的值之前，应先记下 t 移动的路径，即令 p 所指结点的 hp 域的值为 t，且 tag 域的值为“0”。

另一方面，当 q 指向 p 的表尾时，也可能有两种情况出现：①p 的表尾为未加标志的子表，则需遍历表尾的子表，同样 p、t 指针要作相应的移动。为了记下当前表结点的母表结点，同样要在改动 p、t 指针的值之前先记下路径；即令 p 所指结点的 tp 域的值改为 t，然后令 t 赋值 p，p 赋值 q；②p 的表尾为“空”或是已加上标志的子表，此时表明 p 所指的表已加上标志，则 p 应退回到其母表结点即 t 所指结点，相应地 t 也应后退一步，即退到 t 结点的母表结点。综上所述可知，t 的移动路径已记录在 t 结点的 hp 域或 tp 域中，究竟是哪一个？则要由辨别 tag 域的值来定。它不仅指示 t 应按哪个指针所指路径退回，而且指示了下一步应做什么。若 t 结点是其母表表头，则应继续遍历其母表的表尾。若 t 结点是其母表的表尾，则应继续找更高一层的母表结点。整个算法大致描述如下：(GL 为广义表的头指针)

```
t = NULL;   p = GL;   finished = FALSE;
while (!finished) {
   while (p->mark==0) {
      p->mark = 1;
      MarkHead(p);    // 若表头是未经遍历的非空子表，则修改指针记录路径，
   }                  // 且 p 指向表头；否则 p 不变
   q = p->p.tp;
   if (q && q->mark==0) MarkTail(p);  // 修改指针记录路径，且 p 指向表尾
   else BackTrack(finished);  // 若从表尾回溯到第一个结点，则 finished 为 TRUE
}
```

求精后的广义表遍历算法如算法 8.3 所示。

```
void MarkList(GList GL) {
   // 遍历非空广义表 GL(GL! =NULL 且 GL->mark==0)，对表中所有未加标志的结点加标志。
   t = NULL;   p = GL;   finished = FALSE;   // t 指示 p 的母表
```


www.TopSage.com

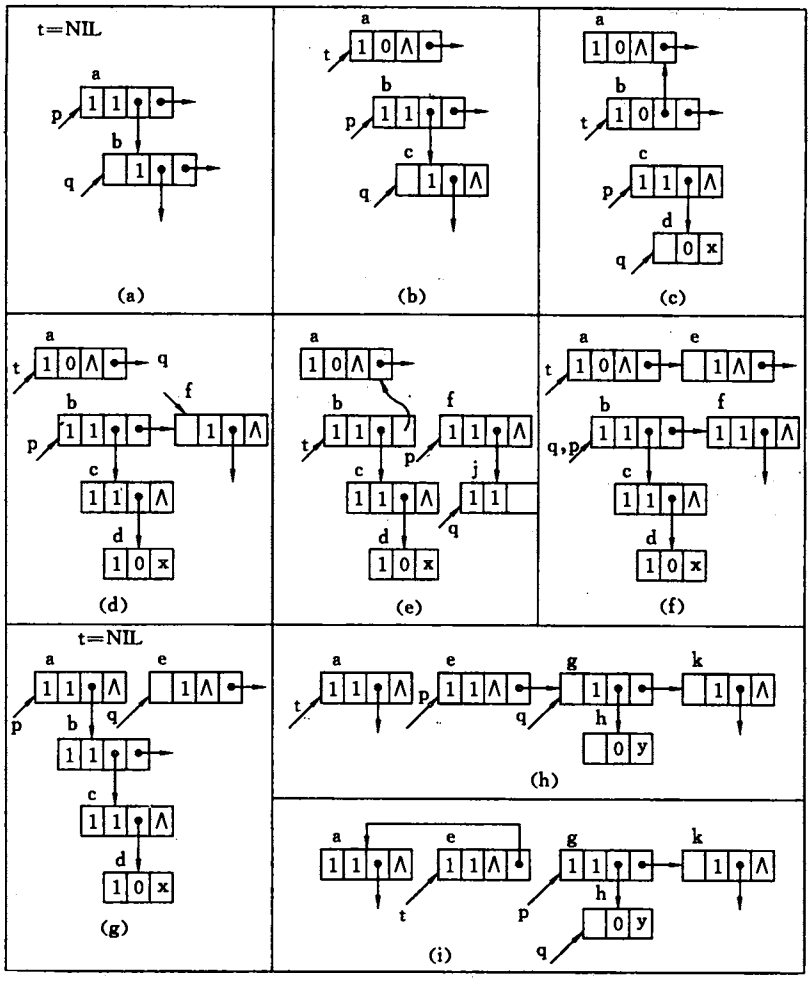


图 8.11　遍历广义表

(a) 指针初始化,q 指向表头;　　(b)和(c) 指针向表头方向推进一步;

(d) 对元素结点加标志后指针后退,q 指向表尾;　　(e) 指针向表尾方向推进一步,q 指向表头;

(f) 指针后退一步;　　(g) 指针继续后退,q 指向表尾;

(h) 指针向表尾方向推进一步,q 指向表尾(表头为空表);

(i) 指针继续向表尾方向推进一步,q 指向表头

```
while (! finished) {
    while (p->mark == 0) {
        p->mark = 1;
        // MarkHead(p)的细化:
        q = p->p.hp;    // q 指向 * p 的表头
        if (q && q->mark == 0) {
            if (q->tag == 0) q->mrak = 1;              // ATOM,表头为原子结点
            else {p->p.hp = t;  p->tag = 0;  t = p;  p = q; }     // 继续遍历子表
        }
    }  // 完成对表头的标志
    q = p->p.tp;      // q 指向 * p 的表尾
```


www.TopSage.com

大家网 TopSage.com

```
        if (q && q->mark == 0) {p->p.tp = t;  t = p;  p = q; }      // 继续遍历表尾
        else {  // BackTrack(finished)的细化:
           while (t && t->tag == 1) {  // LIST,表结点,从表尾回溯
              q = t;  t = q->p.tp;  q->p.tp = p;  p = q;
           }
           if (! t) finished = TRUE;  // 结束
           else {  // 从表头回溯
              q = t;  t = q->p.hp;  q->p.hp = p;  p = q;  p->tag = 1;
           }  // 继续遍历表尾
        }
     }
  }// MarkList
```

算法 8.3

图 8.11 展示对图 8.10 中的广义表进行遍历加标志时各指针的变化状况。(a)为算法8.3开始执行时的状态。(b)和(c)为指针向表头方向移动并改变结点的 hp 域指针的情形。(d)表示当表头遍历完成将对表尾进行标志时的指针变化情况。从(e)和(f)读者可看到指针回溯的情形。在此省略了继续遍历时的指针变化状况,有兴趣的读者可试之补充。

比较上述 3 种算法各有利弊。第 3 种算法在标志时不需要附加存储,使动态分配的可利用空间得到充分利用,但是由于在算法中,几乎是每个表结点的指针域的值都要作两次改变,因此时间上的开销相当大,而且,一旦发生中断,整个系统瘫痪,无法重新启动运行。而非递归算法操作简单,时间上要比第 3 种算法省得多,然而它需要占有一定空间,使动态分配所用的存储量减少。总之,无用单元收集是很费时间的,不能在实时处理的情况下应用。

通常,无用单元的收集工作是由编译程序中的专用系统来完成的,它也可以作为一个标准函数由用户自行调用(类似于 free 函数的使用)。不论哪一种情况,系统都要求用户建立一个初始变量表登录用户程序中所有链表的表头指针,以便从这些指针出发进行标志。

下面我们可以对无用单元收集算法作某种定量估计。如上所述,整个算法分两步进行:第一步对占用结点加标志,不管用哪一种算法,其所用时间都和结点数成正比。假设总的占用结点数为 N,则标志过程所需时间为 c_1N(其中 c_1 为某个常数);第二步是从可用空间的第一个结点起,顺序扫描,将所有未加标志的结点链结在一起。假设可用空间总共含有 M 个结点,则所需时间为 c_2M(其中 c_2 为某个常数)。由此,收集算法总的时间为 c_1N+c_2M,同时收集到的无用结点个数为 $M-N$。

显然,无用单元收集这项工作的效率和最后能收集到的可以重新分配的无用结点数有关。我们用收集一个无用结点所需的平均时间 $(c_1N+c_2M)/(M-N)$ 来度量这个效率。假设以 $\rho=N/M$ 表示内存使用的密度,则上述平均时间为 $(c_1\rho+c_2)/(1-\rho)$。当内存中 3/4 的结点为无用结点,即 $\rho=1/4$ 时,收集一个结点所需平均时间为 $1/3c_1+4/3c_2$。反之,当内存中 1/4 的结点为无用结点,即 $\rho=3/4$ 时,收集一个结点所需平均时间为 $3c_1+4c_2$。由此可见,可利用内存区中只有少量的结点为无用结点时,收集无用单元的操

www.TopSage.com

作的效率很低。不仅如此，而且当系统重又恢复运行时，这些结点又很快被消耗掉，导致另一次无用单元的收集。如此下去有可能造成恶性循环，以至最后整个系统瘫痪。解决的办法可以由系统事先确定一个常数 k，当收集到的无用单元数为 k 或更少时系统就不再运行下去。

8.6 存储紧缩

前面几节中讨论的动态存储管理方法都有一个共同的特点，即建立一个“空闲块”或“无用结点”组成的可利用空间表，这个可利用空间表采用链表结构，其结点大小可以相同，也可以不同。

这一节将要介绍另一种结构的动态存储管理方法。在整个动态存储管理过程中，不

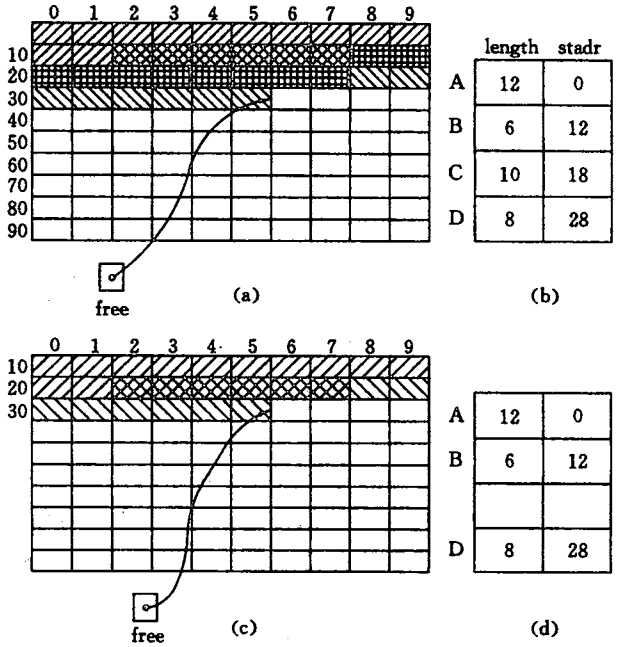


图 8.12 堆存储管理示意图

(a) 堆空间；(b) 串的存储映像；(c) 紧缩后的堆；(d) 修改后的存储映像

管哪个时刻，可利用空间都是一个地址连续的存储区，在编译程序中称之为“堆”，每次分配都是从这个可利用空间中划出一块。其实现办法是：设立一个指针，称之为堆指针，始终指向堆的最低(或最高)地址。当用户申请 N 个单位的存储块时，堆指针向高地址(或低地址)移动 N 个存储单位，而移动之前的堆指针的值就是分配给用户的占用块的初始地址。回顾第 4 章中提及的串值存储空间的动态分配就是用的这种堆的存储管理。例如，某个串处理系统中有 A、B、C、D 这 4 个串，其串值长度分别为 12、6、10 和 8。假设堆指针 free 的初值为零，则分配给这 4 个串值的存储空间的初始地址分别为 0、12、18 和 28，如图 8.12(a)和(b)所示，分配后的堆指针的值为 36。因此，这种堆结构的存储管理的

www.TopSage.com

大家网 TopSage.com

分配算法非常简单。反之，回收用户释放的空闲块就比较麻烦。由于系统的可利用空间始终是一个地址连续的存储块，因此回收时必须将所释放的空闲块合并到整个堆上去才能重新使用，这就是**“存储紧缩”**的任务。通常，有两种做法：一种是一旦有用户释放存储块即进行回收紧缩，例如，图 8.12(a)的堆，在 c 串释放存储块时即回收紧缩成为图 8.12(c)的堆，同时修改串的存储映像成图 8.12(d)的状态；另一种是在程序执行过程中不回收用户随时释放的存储块，直到可利用空间不够分配或堆指针指向最高地址时才进行存储紧缩。此时紧缩的目的是将堆中所有的空闲块连成一片，即将所有的占用块都集中到可利用空间的低地址区，而剩余的高地址区成为一整个地址连续的空闲块，如图 8.13 所示，其中(a)为紧缩前的状态，(b)为紧缩后的状态。

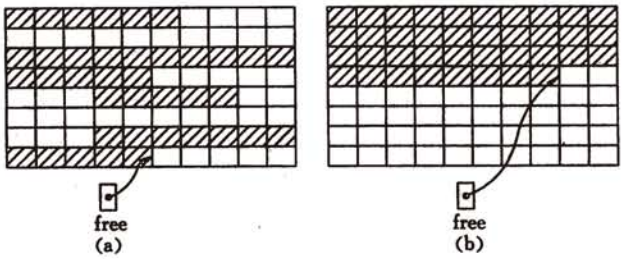


图 8.13 紧缩前后的堆(存储空间)

(a) 紧缩前； (b) 紧缩后

和上节讨论的无用单元收集类似，为实现存储紧缩，首先要对占用块进行“标志”，标志算法和上节类同(存储块的结构可能不同)；其次需进行下列 4 步操作：

(1) 计算占用块的新地址。从最低地址始巡查整个存储空间，对每一个占用块找到它在紧缩后的新地址。为此，需设立两个指针随巡查向前移动，这两个指针分别指示占用块在紧缩之前和之后的原地址和新地址。因此，在每个占用块的第一个存储单位中，除了设立长度域(存储该占用块的大小)和标志域(存储区别该存储块是占用块或空闲块的标志)之外，还需设立一个新地址域，以存储占用块在紧缩后应有的新地址，即建立一张新、旧地址的对照表。

(2) 修改用户的初始变量表，以便在存储紧缩后用户程序能继续正常运行。

(3) 检查每个占用块中存储的数据。若有指向其他存储块的指针，则需作相应修改。

(4) 将所有占用块迁移到新地址去。这实质上是作传送数据的工作。

至此，完成了存储紧缩的操作。最后，将堆指针赋以新值(即紧缩后的空闲存储区的最低地址)。

可见，存储紧缩法比无用单元收集法更为复杂，前者不仅要传送数据(进行占用块迁移)，而且要修改所有占用块中的指针值。因此，存储紧缩也是一个系统操作，且非不得已就不用。

www.TopSage.com

第9章 查 找

本书在第2章至第7章中已经介绍了各种线性或非线性的数据结构，在这一章将讨论另一种在实际应用中大量使用的数据结构——查找表。

查找表(Search Table)是由同一类型的数据元素(或记录)构成的集合。由于“集合”中的数据元素之间存在着完全松散的关系，因此查找表是一种非常灵便的数据结构。

对查找表经常进行的操作有：(1)查询某个“特定的”数据元素是否在查找表中；(2)检索某个“特定的”数据元素的各种属性；(3)在查找表中插入一个数据元素；(4)从查找表中删去某个数据元素。若对查找表只作前两种统称为“查找”的操作，则称此类查找表为**静态查找表**(Static Search Table)。若在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已存在的某个数据元素，则称此类表为**动态查找表**(Dynamic Search Table)。

在日常生活中，人们几乎每天都要进行“查找”工作。例如，在电话号码簿中查阅“某单位”或“某人”的电话号码；在字典中查阅“某个词”的读音和含义等等。其中“电话号码簿”和“字典”都可视作是一张查找表。

在各种系统软件或应用软件中，查找表也是最常见的结构之一。如编译程序中符号表、信息处理系统中信息表等等。

由上述可见，所谓“查找”即为在一个含有众多的数据元素(或记录)的查找表中找出某个“特定的”数据元素(或记录)。

为了便于讨论，必须给出这个“特定的”词的确切含义。首先需引入一个“关键字”的概念。

关键字(Key)是数据元素(或记录)中某个数据项的值，用它可以标识(识别)一个数据元素(或记录)。若此关键字可以惟一地标识一个记录，则称此关键字为**主关键字**(Primary Key)(对不同的记录，其主关键字均不同)。反之，称用以识别若干记录的关键字为**次关键字**(Secondary Key)。当数据元素只有一个数据项时，其关键字即为该数据元素的值。

查找(Searching) 根据给定的某个值，在查找表中确定一个其关键字等于给定值的记录或数据元素。若表中存在这样的一个记录，则称**查找是成功**的，此时查找的结果为给出整个记录的信息，或指示该记录在查找表中的位置；若表中不存在关键字等于给定值的记录，则称**查找不成功**，此时查找的结果可给出一个“空”记录或“空”指针。

例如，当用计算机处理大学入学考试成绩时，全部考生的成绩可以用图9.1所示表的结构储存在计算机中，表中每一行为一个记录，考生的准考证号为记录的关键字。假设给定值为179326，则通过查找可得考生陆华的各科成绩和总分，此时查找为成功的。若给定值为179238，则由于表中没有关键字为179238的记录，则查找不成功。

如何进行查找？显然，在一个结构中查找某个数据元素的过程依赖于这个数据元素在结构中所处的地位。因此，对表进行查找的方法取决于表中数据元素依何种关系(这个

www.TopSage.com

大家网 www.TopSage.com

关系是人为地加上的)组织在一起的。例如查电话号码时,由于电话号码簿是按用户(集体或个人)的名称(或姓名)分类且依笔划顺序编排,则查找的方法就是先顺序查找待查用户的所属类别,然后在此类中顺序查找,直到找到该用户的电话号码为止。又如,查阅英文单词时,由于字典是按单词的字母在字母表中的次序编排的,因此查找时不需要从字典中第一个单词比较起,而只要根据待查单词中每个字母在字母表中的位置查到该单词。

准考证号	姓名	各科成绩							总分
		政治	语文	外语	数学	物理	化学	生物	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
179325	陈红	85	86	88	100	92	90	45	586
179326	陆华	78	75	90	80	95	88	37	543
179327	张平	82	80	78	98	84	96	40	558
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

图 9.1 高考成绩表示例

同样,在计算机中进行查找的方法也随数据结构不同而不同。正如前所述,本章讨论的查找表是一种非常灵便的数据结构。但也正是由于表中数据元素之间仅存在着"同属一个集合"的松散关系,给查找带来不便。为此,需在数据元素之间人为地加上一些关系,以便按某种规则进行查找,即以另一种数据结构来表示查找表。本章将分别就静态查找表和动态查找表两种抽象数据类型讨论其表示和操作实现的方法。

在本章以后各节的讨论中,涉及的关键字类型和数据元素类型统一说明如下:

典型的关键字类型说明可以是

```
typedef float   KeyType;     // 实型
typedef int     KeyType;     // 整型
typedef char    *KeyType;    // 字符串型
```

数据元素类型定义为:

```
typedef struct {
      KeyType key;           // 关键字域
      ...                    // 其他域
}SElemType;
```

对两个关键字的比较约定为如下的宏定义:

```
// - - 对数值型关键字
    #define  EQ(a, b)  ((a) = = (b))
    #define  LT(a, b)  ((a) < (b))
    #define  LQ(a, b)  ((a) <= (b))
    ...
    // - - 对字符串型关键字
    #define  EQ(a, b)  (!strcmp((a), (b)))
    #define  LT(a, b)  (strcmp((a), (b)) < 0)
    #define  LQ(a, b)  (strcmp((a), (b)) <= 0)
    ...
```


www.TopSage.com

9.1 静态查找表

抽象数据类型静态查找表的定义为：

```
ADT StaticSearchTable {
  数据对象 D:D 是具有相同特性的数据元素的集合。各个数据元素均含有类型相同,可惟一标识
           数据元素的关键字。
  数据关系 R:数据元素同属一个集合。
  基本操作 P:
    Create(&ST, n);
      操作结果:构造一个含 n 个数据元素的静态查找表 ST。
    Destroy(&ST);
      初始条件:静态查找表 ST 存在。
      操作结果:销毁表 ST。
    Search(ST, key);
      初始条件:静态查找表 ST 存在,key 为和关键字类型相同的给定值。
      操作结果:若 ST 中存在其关键字等于 key 的数据元素,则函数值为该元素的值或在表中的
               位置,否则为"空"。
    Traverse(ST, Visit());
      初始条件:静态查找表 ST 存在,Visit 是对元素操作的应用函数。
      操作结果:按某种次序对 ST 的每个元素调用函数 visit()一次且仅一次。一旦 visit()失
               败,则操作失败。
}ADT StaticSearchTable
```

静态查找表可以有不同的表示方法，在不同的表示方法中，实现查找操作的方法也不同。

9.1.1 顺序表的查找

以顺序表或线性链表表示静态查找表，则 Search 函数可用顺序查找来实现。本节中只讨论它在顺序存储结构模块中的实现，在线性链表模块中实现的情况留给读者去完成。

```
//- - - - - 静态查找表的顺序存储结构 - - - - -
typedef struct {
  ElemType *elem;     // 数据元素存储空间基址,建表时按实际长度分配,0 号单元留空
  int      length;    // 表长度
}SSTable;
```

下面讨论顺序查找的实现。

顺序查找(Sequential Search)的查找过程为：从表中最后一个记录开始，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值比较相等，则查找成功，找到所查记录；反之，若直至第一个记录，其关键字和给定值比较都不等，则表明表中没有所查记录，查找不成功。此查找过程可用算法 9.1 描述之。

```
int Search_Seq(SSTable ST, KeyType key) {
  // 在顺序表 ST 中顺序查找其关键字等于 key 的数据元素。若找到,则函数值为
  // 该元素在表中的位置,否则为 0。
```


www.TopSage.com

大家网
TopSage.com

```
    ST.elem[0].key = key;                                    // "哨兵"
    for (i = ST.length;  !EQ(ST.elem[i].key,key);  - - i);  // 从后往前找
    return i;                                                // 找不到时,i 为 0
}// Search_Seq
```

算法 9.1

这个算法的思想和第 2 章中的函数 LocateElbm_Sq 一致。只是在 Search_Seq 中，查找之前先对 ST.elem[0]的关键字赋值 key，目的在于免去查找过程中每一步都要检测整个表是否查找完毕。在此，ST.elem[0]起到了监视哨的作用。这仅是一个程序设计技巧上的改进，然而实践证明，这个改进能使顺序查找在 ST.length≥1 000 时，进行一次查找所需的平均时间几乎减少一半(参阅参考书目[1]中 342 页表 7.1)。当然，监视哨也可设在高下标处。

查找操作的性能分析

在第 1 章中曾提及，衡量一个算法好坏的量度有 3 条：时间复杂度(衡量算法执行的时间量级)、空间复杂度(衡量算法的数据结构所占存储以及大量的附加存储)和算法的其他性能。对于查找算法来说，通常只需要一个或几个辅助空间。又，查找算法中的基本操作是"将记录的关键字和给定值进行比较"，因此，通常以"其关键字和给定值进行过比较的记录个数的平均值"作为衡量查找算法好坏的依据。

定义：为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的**平均查找长度**(Average Search Length)。

对于含有 n 个记录的表，查找成功时的平均查找长度为

$$ASL = \sum_{i=1}^{n} P_i C_i \tag{9-1}$$

其中：P_i 为查找表中第 i 个记录的概率，且 $\sum_{i=1}^{n} P_i = 1$；

C_i 为找到表中其关键字与给定值相等的第 i 个记录时，和给定值已进行过比较的关键字个数。显然，C_i 随查找过程不同而不同。

从顺序查找的过程可见，C_i 取决于所查记录在表中的位置。如：查找表中最后一个记录时，仅需比较一次；而查找表中第一个记录时，则需比较 n 次。一般情况下 C_i 等于 $n-i+1$。

假设 n=ST.length，则顺序查找的平均查找长度为

$$ASL = nP_1 + (n-1)P_2 + \cdots + 2P_{n-1} + P_n \tag{9-2}$$

假设每个记录的查找概率相等，即

$$P_i = 1/n$$

则在等概率情况下顺序查找的平均查找长度为

$$ASL_{SS} = \sum_{i=1}^{n} P_i C_i$$

$$= \frac{1}{n} \sum_{i=1}^{n} (n-i+1)$$

www.TopSage.com

$$ASL_{ss}=\frac{n+1}{2} \tag{9-3}$$

有时，表中各个记录的查找概率并不相等。例如：将全校学生的病历档案建立一张表存放在计算机中，则体弱多病同学的病历记录的查找概率必定高于健康同学的病历记录。由于式(9-2)中的 ASL 在 $P_n\geqslant P_{n-1}\geqslant\cdots\geqslant P_2\geqslant P_1$ 时达到极小值。因此，对记录的查找概率不等的查找表若能预先得知每个记录的查找概率，则应先对记录的查找概率进行排序，使表中记录按查找概率由小至大重新排列，以便提高查找效率。

然而，在一般情况下，记录的查找概率预先无法测定。为了提高查找效率，我们可以在每个记录中附设一个访问频度域，并使顺序表中的记录始终保持按访问频度非递减有序的次序排列，使得查找概率大的记录在查找过程中不断往后移，以便在以后的逐次查找中减少比较次数。或者在每次查找之后都将刚查找到的记录直接移至表尾。

顺序查找和我们后面将要讨论到的其他查找算法相比，其缺点是平均查找长度较大，特别是当 n 很大时，查找效率较低。然而，它有很大的优点是：算法简单且适应面广。它对表的结构无任何要求，无论记录是否按关键字有序[①]均可应用，而且，上述所有讨论对线性链表也同样适用。

容易看出，上述对平均查找长度的讨论是在 $\sum_{i=1}^{n}P_i=1$ 的前提下进行的，换句话说，我们认为每次查找都是“成功”的。在本章开始时曾提到，查找可能产生“成功”与“不成功”两种结果，但在实际应用的大多数情况下，查找成功的可能性比不成功的可能性大得多，特别是在表中记录数 n 很大时，查找不成功的概率可以忽略不计。当查找不成功的情形不能忽视时，查找算法的平均查找长度应是查找成功时的平均查找长度与查找不成功时的平均查找长度之和。

对于顺序查找，不论给定值 key 为何值，查找不成功时和给定值进行比较的关键字个数均为 $n+1$。假设查找成功与不成功的可能性相同，对每个记录的查找概率也相等，则 $P_i=1/(2n)$，此时顺序查找的平均查找长度为

$$\begin{aligned}ASL'_{ss}&=\frac{1}{2n}\sum_{i=1}^{n}(n-i+1)+\frac{1}{2}(n+1)\\&=\frac{3}{4}(n+1)\end{aligned} \tag{9-4}$$

在本章的以后各节中，仅讨论查找成功时的平均查找长度和查找不成功时的比较次数，但哈希表例外。

9.1.2 有序表的查找

以有序表表示静态查找表时，Search 函数可用折半查找来实现。

折半查找(Binary Search)的查找过程是：先确定待查记录所在的范围(区间)，然后逐

---

① 若表中所有记录的关键字满足下列关系

ST.elem[i].key≤ST.elem[i+1].key　i=1,2,…,n−1

则称表中记录按关键字有序。

www.TopSage.com

大家网 TopSage.com

步缩小范围直到找到或找不到该记录为止。

例如：已知如下 11 个数据元素的有序表(关键字即为数据元素的值)：

(05,13,19,21,37,56,64,75,80,88,92)

现要查找关键字为 21 和 85 的数据元素。

假设指针 *low* 和 *high* 分别指示待查元素所在范围的下界和上界，指针 *mid* 指示区间的中间位置，即 $mid=\lfloor(low+high)/2\rfloor$。在此例中，*low* 和 *high* 的初值分别为 1 和 11，即[1,11]为待查范围。

下面先看给定值 key=21 的查找过程：

```
05  13  19  21  37  56  64  75  80  88  92
↑low            ↑mid            ↑high
```

首先令查找范围中间位置的数据元素的关键字 ST. elem[mid]. key 与给定值 key 相比较，因为 ST. elem[mid]. key>key，说明待查元素若存在，必在区间[*low*,*mid*−1]的范围内，则令指针 high 指向第 *mid*−1 个元素，重新求得 $mid=\lfloor(1+5)/2\rfloor=3$

```
05  13  19  21  37  56  64  75  80  88  92
↑low    ↑mid    ↑high
```

仍以 ST. elem[mid]. key 和 key 相比，因为 ST. elem[mid]. key<key，说明待查元素若存在，必在[*mid*+1,*high*]范围内，则令指针 low 指向第 *mid*+1 个元素，求得 *mid* 的新值为 4，比较 ST. elem[mid]. key 和 key，因为相等，则查找成功，所查元素在表中序号等于指针 mid 的值。

```
05  13  19  21  37  56  64  75  80  88  92
        ↑low↑high
        ↑mid
```

再看 key=85 的查找过程：

```
05  13  19  21  37  56  64  75  80  88  92
↑low            ↑mid            ↑high
```

ST. elem[mid]. key<key 令 *low*=*mid*+1

```
                        ↑low    ↑mid    ↑high
```

ST. elem[mid]. key<key 令 *low*=*mid*+1

```
                                    ↑low↑high
                                    ↑mid
```

ST. elem[mid]. key>key 令 *high*=*mid*−1

```
                                ↑high ↑low
```

此时因为下界 *low*>上界 *high*，则说明表中没有关键字等于 key 的元素，查找不成功。

从上述例子可见，折半查找过程是以处于区间中间位置记录的关键字和给定值比较，若相等，则查找成功，若不等，则缩小范围，直至新的区间中间位置记录的关键字等于给定值或者查找区间的大小小于零时(表明查找不成功)为止。

上述折半查找过程如算法 9.2 描述所示。

www.TopSage.com

```
int Search_Bin ( SSTable ST, KeyType key ) {
  // 在有序表 ST 中折半查找其关键字等于 key 的数据元素。若找到,则函数值为
  // 该元素在表中的位置,否则为 0。
  low = 1;  high = ST.length;                                  // 置区间初值
  while (low <= high) {
    mid = (low + high) / 2;
    if  (EQ (key , ST.elem[mid].key))  return  mid;            // 找到待查元素
    else if  (LT (key , ST.elem[mid].key))  high = mid - 1;    // 继续在前半区间进行查找
    else  low = mid + 1;                                       // 继续在后半区间进行查找
  }
  return 0;                                                    // 顺序表中不存在待查元素
}// Search_Bin
```

算法 9.2

折半查找的**性能分析**

先看上述 11 个元素的表的具体例子。从上述查找过程可知:

找到第⑥个元素仅需比较 1 次;找到第③和第⑨个元素需比较 2 次;找到第①、④、⑦和⑩个元素需比较 3 次;找到第②、⑤、⑧和⑪个元素需比较 4 次。

这个查找过程可用图 9.2 所示的二叉树来描述。树中每个结点表示表中一个记录,结点中的值为该记录在表中的位置,通常称这个描述查找过程的二叉树为判定树,从判定树上可见,查找 21 的过程恰好是走了一条从根到结点④的路径,和给定值进行比较的关键字个数为该路径上的结点数或结点④在判定树上的层次数。类似地,找到有序表中任一记录的过程就是走了一条从根结点到与该记录相应的结点的路径,和给定值进行比较的关键字个数恰为该结点在判定树上的层次数。因此,折半查找法在查找成功时进行比较的关键字个数最多不超过树的深度,而具有 n 个结点的判定树的深度为$\lfloor\log_2 n\rfloor+1$[①],所以,折半查找法在查找成功时和给定值进行比较的关键字个数至多为$\lfloor\log_2 n\rfloor+1$。

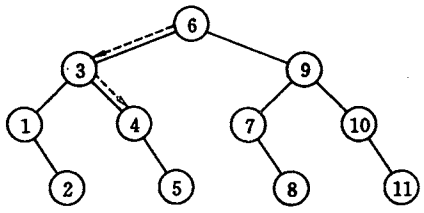


图 9.2 描述折半查找过程的判定树及查找 21 的过程

如果在图 9.2 所示判定树中所有结点的空指针域上加一个指向一个方形结点的指针,如图 9.3 所示。并且,称这些方形结点为判定树的外部结点(与之相对,称那些圆形结点为内部结点),那么折半查找时查找不成功的过程就是走了一条从根结点到外部结点的路径,和给定值进行比较的关键字个数等于该路径上内部结点个数,例如:查找 85 的过程即为走了一条从根到结点 9-10 的路径。因此,折半查找在查找不成功时和给定值进行比较的关键字个数最多也不超过$\lfloor\log_2 n\rfloor+1$。

那么,折半查找的平均查找长度是多少呢?

---

① 判定树非完全二叉树,但它的叶子结点所在层次之差最多为 1,则 n 个结点的判定树的深度和 n 个结点的完全二叉树的深度相同。

www.TopSage.com

大家网
TopSage.com

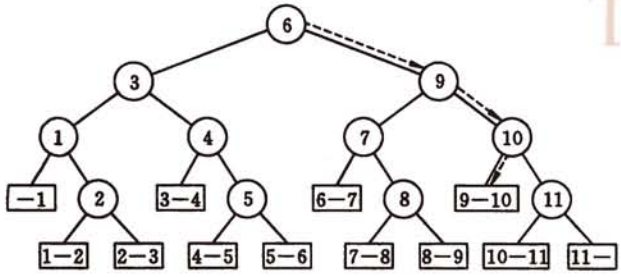


图 9.3　加上外部结点的判定树和查找 85 的过程

为讨论方便起见，假定有序表的长度 $n=2^h-1$（反之，$h=\log_2(n+1)$），则描述折半查找的判定树是深度为 h 的满二叉树。树中层次为 1 的结点有 1 个，层次为 2 的结点有 2 个，……，层次为 h 的结点有 2^{h-1} 个。假设表中每个记录的查找概率相等 $\left(P_i=\frac{1}{n}\right)$，则查找成功时折半查找的平均查找长度

$$\begin{aligned} ASL_{bs} &= \sum_{i=1}^{n} P_i C_i \\ &= \frac{1}{n}\sum_{j=1}^{h} j \cdot 2^{j-1} \\ &= \frac{n+1}{n}\log_2(n+1) - 1^{①} \end{aligned} \tag{9-5}$$

对任意的 n，当 n 较大（$n>50$）时，可有下列近似结果

$$ASL_{bs} = \log_2(n+1) - 1 \tag{9-6}$$

可见，折半查找的效率比顺序查找高，但折半查找只适用于有序表，且限于顺序存储结构（对线性链表无法有效地进行折半查找）。

以有序表表示静态查找表时，进行查找的方法除折半查找之外，还有斐波那契查找和插值查找。

斐波那契查找是根据斐波那契序列[②]的特点对表进行分割的。假设开始时表中记录个数比某个斐波那契数小 1，即 $n=F_u-1$，然后将给定值 key 和 ST. elem[F_{u-1}]. key 进行比较，若相等，则查找成功；若 key＜ST. elem[F_{u-1}]. key，则继续在自 ST. elem[1]至 ST. elem[$F_{u-1}-1$]的子表中进行查找，否则继续在自 ST. elem[$F_{u-1}+1$]至 ST. elem[F_u-1]的子表中进行查找，后一子表的长度为 $F_{u-2}-1$。斐波那契查找的平均性能比折半查找好，但最坏情况下的性能（虽然仍是 $O(\log n)$）却比折半查找差。它还有一个优点就是分割时只需进行加、减运算。

---

① $ASL_{bs} = \sum_{i=1}^{n} P_i C_i = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{n}\sum_{j=1}^{h} j \cdot 2^{j-1} = \frac{1}{n}\left(\sum_{i=0}^{h-1} 2^i + 2\sum_{i=0}^{h-2} 2^i + \cdots + 2^{h-1}\sum_{i=0}^{0} 2^i\right)$

$= \frac{1}{n}[h \cdot 2^h - (2^0 + 2^1 + \cdots + 2^{h-1})] = \frac{1}{n}[(h-1)2^h + 1]$

$= \frac{1}{n}[(n+1)(\log_2(n+1) - 1) + 1] = \frac{n+1}{n}\log_2(n+1) - 1$

② 这种序列可定义为：$F_0=0$，$F_1=1$，$F_i=F_{i-1}+F_{i-2}$，$i\geqslant 2$

www.TopSage.com

插值查找是根据给定值 key 来确定进行比较的关键字 ST. elem[i]. key 的查找方法。令 $i=\frac{\text{key}-\text{ST. elem[l]. key}}{\text{ST. elem[h]. key}-\text{ST. elem[l]. key}}(\text{h}-\text{l}+1)$，其中 ST. elem[l]和 ST. elem[h]分别为有序表中具有最小关键字和最大关键字的记录。显然，这种插值查找只适于关键字均匀分布的表，在这种情况下，对表长较大的顺序表，其平均性能比折半查找好。

9.1.3 静态树表的查找

上一小节对有序表的查找性能的讨论是在“等概率”的前提下进行的，即当有序表中各记录的查找概率相等时，按图 9.2 所示判定树描述的查找过程来进行折半查找，其性能最优。如果有序表中各记录的查找概率不等，情况又如何呢？

先看一个具体例子。假设有序表中含 5 个记录，并且已知各记录的查找概率不等，分别为 $p_1=0.1, p_2=0.2, p_3=0.1, p_4=0.4$ 和 $p_5=0.2$。则按式(9-1)的定义，对此有序表进行折半查找，查找成功时的平均查找长度为

$$\sum_{i=1}^{5} P_i C_i = 0.1\times 2+0.2\times 3+0.1\times 1+0.4\times 2+0.2\times 3 = 2.3$$

但是，如果在查找时令给定值先和第 4 个记录的关键字进行比较，比较不相等时再继续在左子序列或右子序列中进行折半查找，则查找成功时的平均查找长度为

$$\sum_{i=1}^{5} P_i C_i = 0.1\times 3+0.2\times 2+0.1\times 3+0.4\times 1+0.2\times 2 = 1.8$$

这就说明，当有序表中各记录的查找概率不等时，按图 9.2 所示判定树进行折半查找，其性能未必是最优的。那么此时应如何进行查找呢？换句话说，描述查找过程的判定树为何类二叉树时，其查找性能最佳？

如果只考虑查找成功的情况，则使查找性能达最佳的判定树是其带权内路径长度之和 PH 值[①]

$$PH = \sum_{i=1}^{n} w_i h_i \tag{9-7}$$

取最小值的二叉树。其中：n 为二叉树上结点的个数(即有序表的长度)；h_i 为第 i 个结点在二叉树上的层次数；结点的权 $w_i=cp_i(i=1,2,\cdots,n)$，其中 p_i 为结点的查找概率，c 为某个常量。称 PH 值取最小的二叉树为**静态最优查找树**(Static Optimal Search Tree)。由于构造静态最优查找树花费的时间代价较高，因此在本书中不作详细讨论，有兴趣的读者可查阅参考书目[1]。在此向读者介绍一种构造近似最优查找树的有效算法。

已知一个按关键字有序的记录序列

$$(r_l, r_{l+1}, \cdots, r_h) \tag{9-8}$$

其中

$$\text{r}_\text{l}\text{. key} < \text{r}_{\text{l+1}}\text{. key} < \cdots < \text{r}_\text{h}\text{. key}$$

与每个记录相应的权值为

$$w_l, w_{l+1}, \cdots, w_h \tag{9-9}$$

现构造一棵二叉树，使这棵二叉树的带权内路径长度 PH 值在所有具有同样权值的二叉

① PH 值和平均查找长度成正比。

www.TopSage.com

大家网 TopSage.com

树中近似为最小，称这类二叉树为**次优查找树**(Nearly Optimal Search Tree)。

构造次优查找树的方法是：首先在式(9-8)所示的记录序列中取第 $i(l\leqslant i\leqslant h)$ 个记录构造根结点$\textcircled{r_i}$，使得

$$\Delta P_i = \left|\sum_{j=i+1}^{h} w_j - \sum_{j=l}^{i-1} w_j\right| \tag{9-10}$$

取最小值($\Delta P_i = \min\limits_{l\leqslant j\leqslant h}\{\Delta P_j\}$)，然后分别对子序列$\{r_l, r_{l+1}, \cdots, r_{i-1}\}$和$\{r_{i+1}, \cdots, r_h\}$构造两棵次优查找树，并分别设为根结点$\textcircled{r_i}$的左子树和右子树。

为便于计算 ΔP，引入累计权值和

$$sw_i = \sum_{j=l}^{i} w_j \tag{9-11}$$

并设 $w_{l-1}=0$ 和 $sw_{l-1}=0$，则

$$\begin{cases} sw_{i-1} - sw_{l-1} = \sum\limits_{j=l}^{i-1} w_j \\ sw_h - sw_i = \sum\limits_{j=i+1}^{h} w_j \end{cases} \tag{9-12}$$

$$\begin{aligned} \Delta P_i &= \left|(sw_h - sw_i) - (sw_{i-1} - sw_{l-1})\right| \\ &= \left|(sw_h + sw_{l-1}) - sw_i - sw_{i-1}\right| \end{aligned} \tag{9-13}$$

由此可得构造次优查找树的递归算法如算法 9.3 所示。

```
void SecondOptimal(BiTree &T,ElemType R[],float sw[],int low,int high){
  // 由有序表 R[low..high]及其累计权值表 sw(其中 sw[0]==0)递归构造次优查找树 T。
  i = low; min = abs(sw[high] - sw[low]); dw = sw[high] + sw[low - 1];
  for (j = low + 1; j<= high; ++ j)              // 选择最小的 ΔPi 值
    if (abs(dw - sw[j] - sw[j - 1]) < min) {
      i = j; min = abs(dw - sw[j] - sw[j - 1]);
    }
  T = (BiTree)malloc(sizeof(BiTNode));
  T->data = R[i];                                // 生成结点
  if (i == low)  T->lchild = NULL;               // 左子树空
  else  SecondOptimal(T->lchild, R, sw, low, i - 1);   // 构造左子树
  if (i == high)  T->rchild = NULL;              // 右子树空
  else  SecondOptimal(T->rchild, R, sw, i + 1, high);  // 构造右子树
}// SecondOptimal
```

算法 9.3

例 9-1 已知含 9 个关键字的有序表及其相应权值为：

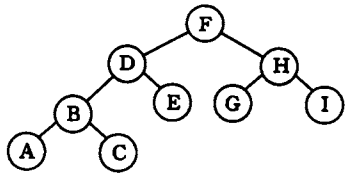
关键字	A	B	C	D	E	F	G	H	I
权值	1	1	2	5	3	4	4	3	5

则按算法 9.3 构造次优查找树的过程中累计权值 SW 和 ΔP 的值如图 9.4(a)所示，构造所得次优二叉查找树如图 9.4(b)所示。

www.TopSage.com

j	0	1	2	3	4	5	6	7	8	9
key_j		A	B	C	D	E	F	G	H	I
W_j	0	1	1	2	5	3	4	4	3	5
SW_j	0	1	2	4	9	12	16	20	23	28
ΔP_j		27	25	22	15	7	0	8	15	23
(根)							↑i			
ΔP_j		11	9	6	1	9		8	1	7
(根)					↑i				↑i	
ΔP_j		3	1	2		0		0		0
(根)			↑i			↑i		↑i		↑i
ΔP_j		0		0						
(根)		↑i		↑i						

(a)



(b)

图 9.4 构造次优二叉查找树示例

(a) 累计权值和 ΔP 值；(b) 次优查找树

由于在构造次优查找树的过程中，没有考察单个关键字的相应权值，则有可能出现被选为根的关键字的权值比与它相邻的关键字的权值小。此时应作适当调整：选取邻近的权值较大的关键字作次优查找树的根结点。

例 9-2 已知含 5 个关键字的有序表及其相应权值为

关键字	A	B	C	D	E
权值	1	30	2	29	3

则按算法 9.3 构造所得次优查找树如图 9.5(a)所示，调整处理后的次优查找树如图 9.5(b)所示。容易算得，前者的 PH 值为 132，后者的 PH 值为 105。

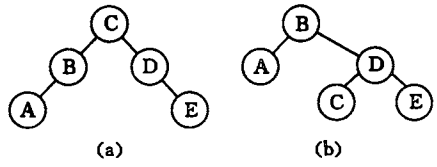


图 9.5 根的权小于子树根权的情况

(a) 调整之前的次优查找树；

(b) 调整之后的次优查找树

大量的实验研究表明，次优查找树和最优查找树的查找性能之差仅为 1%～2%，很少超过 3%，而且构造次优查找树的算法的时间复杂度为 $O(n\log n)$，因此算法 9.3 是构造近似最优二叉查找树的有效算法。

从次优查找树的结构特点可见，其查找过程类似于折半查找。若次优查找树为空，则查找不成功，否则，首先将给定值 *key* 和其根结点的关键字相比，若相等，则查找成功，该根结点的记录即为所求；否则将根据给定值 *key* 小于或大于根结点的关键字而分别在左子树或右子树中继续查找直至查找成功或不成功为止(算法描述和下节讨论的二叉排序

www.TopSage.com

大家网 www.TopSage.com

树的查找算法类似，在此省略）。由于查找过程恰是走了一条从根到待查记录所在结点（或叶子结点）的一条路径，进行过比较的关键字个数不超过树的深度，因此，次优查找树的平均查找长度和 log$n$ 成正比。可见，在记录的查找概率不等时，可用次优查找树表示静态查找树，故又称静态树表，按有序表构造次优查找树的算法如算法 9.4 所示。

```
typedef  BiTree SOSTree;      // 次优查找树采用二叉链表的存储结构
Status CreateSOSTree(SOSTree &T, SSTable ST) {
      // 由有序表 ST 构造一棵次优查找树 T。ST 的数据元素含有权域 weight。
      if (ST.length == 0) T = NULL;
      else {
        FindSW(sw, ST);       // 按照由有序表 ST 中各数据元素的 weight 域求累计权值表 sw。
        SecondOpiamal(T, ST.elem, sw, 1, ST.length);
      }
      return OK;
    }// CreateSOSTree
```

算法 9.4

9.1.4 索引顺序表的查找

若以索引顺序表表示静态查找表，则 Search 函数可用分块查找来实现。

分块查找又称索引顺序查找，这是顺序查找的一种改进方法。在此查找法中，除表本身以外，尚需建立一个“索引表”。例如，图 9.6 所示为一个表及其索引表，表中含有 18 个记录，可分成 3 个子表（R_1，R_2，…，R_6）、（R_7，R_8，…，R_{12}）、（R_{13}，R_{14}，…，R_{18}），对每个子表（或称块）建立一个索引项，其中包括两项内容：关键字项（其值为该子表内的最大关键字）和指针项（指示该子表的第一个记录在表中位置）。索引表按关键字有序，则表或者有序或者分块有序。所谓“分块有序”指的是第二个子表中所有记录的关键字均大于第一个子表中的最大关键字，第三个子表中的所有关键字均大于第二个子表中的最大关键字，……，依次类推。

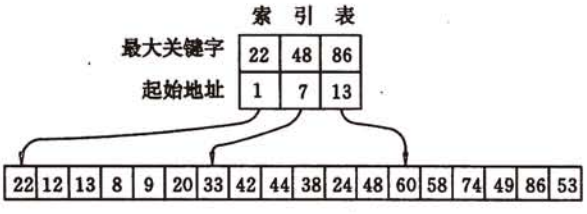


图 9.6 表及其索引表

因此，分块查找过程需分两步进行。先确定待查记录所在的块（子表），然后在块中顺序查找。假设给定值 $key=38$，则先将 key 依次和索引表中各最大关键字进行比较，因为 $22<key<48$，则关键字为 38 的记录若存在，必定在第二个子表中，由于同一索引项中的指针指示第二个子表中的第一个记录是表中第 7 个记录，则自第 7 个记录起进行顺序查找，直到 ST.elem[10].key=key 为止。假如此子表中没有关键字等于 key 的记录（例如：$key=29$ 时自第 7 个记录起至第 12 个记录的关键字和 key 比较都不等），则查找不成功。

由于由索引项组成的索引表按关键字有序，则确定块的查找可以用顺序查找，亦可用

www.TopSage.com

折半查找，而块中记录是任意排列的，则在块中只能是顺序查找。

由此，分块查找的算法即为这两种查找算法的简单合成。

分块查找的平均查找长度为

$$ASL_{bs} = L_b + L_w \tag{9-14}$$

其中：L_b 为查找索引表确定所在块的平均查找长度，L_w 为在块中查找元素的平均查找长度。

一般情况下，为进行分块查找，可以将长度为 n 的表均匀地分成 b 块，每块含有 s 个记录，即 $b=\lceil n/s \rceil$；又假定表中每个记录的查找概率相等，则每块查找的概率为 $1/b$，块中每个记录的查找概率为 $1/s$。

若用顺序查找确定所在块，则分块查找的平均查找长度为

$$ASL_{bs} = L_b + L_w = \frac{1}{b}\sum_{j=1}^{b} j + \frac{1}{s}\sum_{i=1}^{s} i = \frac{b+1}{2} + \frac{s+1}{2}$$

$$= \frac{1}{2}\left(\frac{n}{s} + s\right) + 1 \tag{9-15}$$

可见，此时的平均查找长度不仅和表长 n 有关，而且和每一块中的记录个数 s 有关。在给定 n 的前提下，s 是可以选择的。容易证明，当 s 取 $\sqrt{n}$ 时，ASL_{bs} 取最小值 $\sqrt{n}+1$。这个值比顺序查找有了很大改进，但远不及折半查找。

若用折半查找确定所在块，则分块查找的平均查找长度为

$$ASL'_{bs} \approx \log_2\left(\frac{n}{s} + 1\right) + \frac{s}{2} \tag{9-16}$$

9.2 动态查找表

在这一节和下一节中，我们将讨论动态查找表的表示和实现。动态查找表的特点是，表结构本身是在查找过程中动态生成的，即对于给定值 *key*，若表中存在其关键字等于 *key* 的记录，则查找成功返回，否则插入关键字等于 *key* 的记录。以下是动态查找表的定义：

抽象数据类型动态查找表的定义如下：

```
ADT DynamicSearchTable {
  数据对象 D:D 是具有相同特性的数据元素的集合。各个数据元素均含有类型相同，可惟一标识
            数据元素的关键字。
  数据关系 R:数据元素同属一个集合。
  基本操作 P:
    InitDSTable(&DT);
      操作结果:构造一个空的动态查找表 DT。
    DestroyDSTable(&DT);
      初始条件:动态查找表 DT 存在。
      操作结果:销毁动态查找表 DT。
    SearchDSTable(DT, key);
      初始条件:动态查找表 DT 存在,key 为和关键字类型相同的给定值。
```


www.TopSage.com

大家网 TopSage.com

操作结果：若 DT 中存在其关键字等于 key 的数据元素，则函数值为该元素的值或在表中的位置，否则为“空”。

InsertDSTable(&DT, e);

初始条件：动态查找表 DT 存在，e 为待插入的数据元素。

操作结果：若 DT 中不存在其关键字等于 e.key 的数据元素，则插入 e 到 DT。

DeleteDSTable(&DT, key);

初始条件：动态查找表 DT 存在，key 为和关键字类型相同的给定值。

操作结果：若 DT 中存在其关键字等于 key 的数据元素，则删除之。

TraverseDSTable(DT, Visit());

初始条件：动态查找表 DT 存在，Visit 是对结点操作的应用函数。

操作结果：按某种次序对 DT 的每个结点调用函数 Visit()一次且至多一次。一旦 Visit()失败，则操作失败。

}**ADT** DynamicSearchTable

动态查找表亦可有不同的表示方法。在本节中将讨论以各种树结构表示时的实现方法。

9.2.1 二叉排序树和平衡二叉树

1. 二叉排序树及其查找过程

什么是二叉排序树？

二叉排序树(Binary Sort Tree) 或者是一棵空树；或者是具有下列性质的二叉树：(1)若它的左子树不空，则左子树上**所有**结点的值**均小于**它的根结点的值；(2)若它的右子树不空，则右子树上**所有**结点的值**均大于**它的根结点的值；(3)它的左、右子树也分别为二叉排序树。

例如图 9.7 所示为两棵二叉排序树。

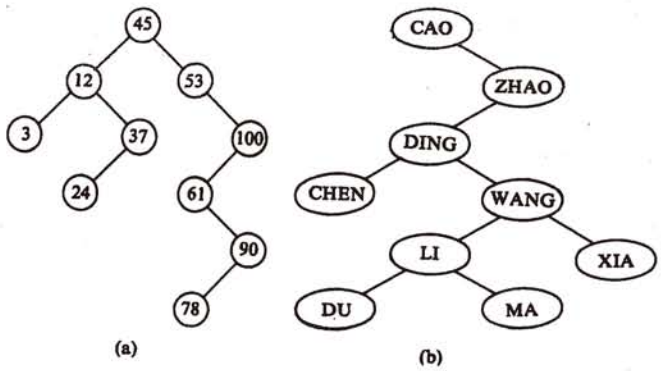


图 9.7 二叉排序树示例

二叉排序树又称二叉查找树，根据上述定义的结构特点可见，它的查找过程和次优二叉树类似。即当二叉排序树不空时，首先将给定值和根结点的关键字比较，若相等，则查找成功，否则将依据给定值和根结点的关键字之间的大小关系，分别在左子树或右子树上继续进行查找。通常，可取二叉链表作为二叉排序树的存储结构，则上述查找过程如算法 9.5(a)所描述。

www.TopSage.com

```
BiTree SearchBST(BiTree T,KeyType key){
  // 在根指针 T 所指二叉排序树中递归地查找某关键字等于 key 的数据元素,
  // 若查找成功,则返回指向该数据元素结点的指针,否则返回空指针
  if((!T) || EQ(key,T->data.key)) return(T);        // 查找结束
  else if LT(key,T->data.key) return(SearchBST(T->lchild,key));
                                                    // 在左子树中继续查找
  else return(SearchBST(T->rchild,key));            // 在右子树中继续查找
}// SearchBST
```

算法 9.5(a)

例如:在图 9.7(a)所示的二叉排序树中查找关键字等于 100 的记录(树中结点内的数均为记录的关键字)。首先以 $key=100$ 和根结点的关键字作比较,因为 $key>45$,则查找以㊺为根的右子树,此时右子树不空,且 $key>53$,则继续查找以结点㊸为根的右子树,由于 key 和㊸的右子树根的关键字 100 相等,则查找成功,返回指向结点⑩⓪的指针值。又如在图 9.7(a)中查找关键字等于 40 的记录,和上述过程类似,在给定值 key 与关键字 45、12 及 37 相继比较之后,继续查找以结点㊲为根的右子树,此时右子树为空,则说明该树中没有待查记录,故查找不成功,返回指针值为"NULL"。

2. 二叉排序树的插入和删除

和次优二叉树相对,二叉排序树是一种动态树表。其特点是,树的结构通常不是一次生成的,而是在查找过程中,当树中不存在关键字等于给定值的结点时再进行插入。新插入的结点一定是一个新添加的叶子结点,并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点。为此,需将上一小节的二叉排序树的查找算法改写成算法 9.5(b),以便能在查找不成功时返回插入位置。插入算法如算法 9.6 所示。

```
Status SearchBST(BiTree T,KeyType key,BiTree f,BiTree &p){
  // 在根指针 T 所指二叉排序树中递归地查找其关键字等于 key 的数据元素,若查找成功,
  // 则指针 p 指向该数据元素结点,并返回 TRUE,否则指针 p 指向查找路径上访问的
  // 最后一个结点并返回 FALSE,指针 f 指向 T 的双亲,其初始调用值为 NULL
  if(!T)  {p=f; return FALSE;}                                  // 查找不成功
  else  if EQ(key,T->data.key) {p=T; return TRUE;}              // 查找成功
  else  if LT(key,T->data.key) return SearchBST(T->lchild,key,T,p); // 在左子树中继续查找
  else  return SearchBST(T->rchild,key,T,p);                    // 在右子树中继续查找
}// SearchBST
```

算法 9.5(b)

```
Status InsertBST(BiTree &T, ElemType e ) {
  // 当二叉排序树 T 中不存在关键字等于 e.key 的数据元素时,插入 e 并返回 TRUE,
  // 否则返回 FALSE
  if (!SearchBST ( T, e.key, NULL, p ) {                        // 查找不成功
    s = (BiTree) malloc (sizeof (BiTNode));
    s->data = e;  s->lchild = s->rchild = NULL;
    if  ( !p ) T = s;                       // 被插结点 *s 为新的根结点
    else if LT(e.key, p->data.key) p->lchild = s;  // 被插结点 *s 为左孩子
    else  p->rchild = s;                    // 被插结点 *s 为右孩子
    return TRUE;
```


www.TopSage.com

大家网 TopSage.com

```
    }
    else return FALSE;                    // 树中已有关键字相同的结点,不再插入
}   // Insert BST
```

算法 9.6

若从空树出发,经过一系列的查找插入操作之后,可生成一棵二叉树。设查找的关键字序列为{45,24,53,45,12,24,90},则生成的二叉排序树如图 9.8 所示。

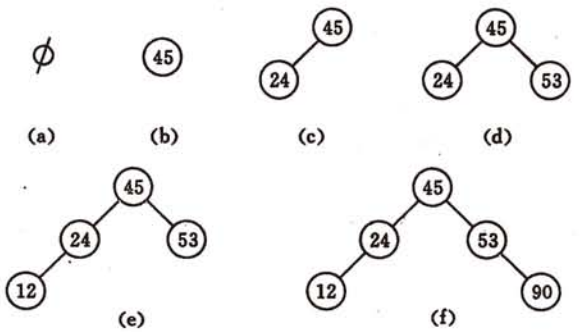


图 9.8 二叉排序树的构造过程

(a) 空树;(b) 插入 45;(c) 插入 24;(d) 插入 53;(e) 插入 12;(f) 插入 90

容易看出,中序遍历二叉排序树可得到一个关键字的有序序列(这个性质是由二叉排序树的定义决定的,读者可以自己证明之)。这就是说,一个无序序列可以通过构造一棵二叉排序树而变成一个有序序列,构造树的过程即为对无序序列进行排序的过程。不仅如此,从上面的插入过程还可以看到,每次插入的新结点都是二叉排序树上新的叶子结点,则在进行插入操作时,不必移动其他结点,仅需改动某个结点的指针,由空变为非空即可。这就相当于在一个有序序列上插入一个记录而不需要移动其他记录。它表明,二叉排序树既拥有类似于折半查找的特性,又采用了链表作存储结构,因此是动态查找表的一种适宜表示。

同样,在二叉排序树上删去一个结点也很方便。对于一般的二叉树来说,删去树中一个结点是没有意义的。因为它将使以被删结点为根的子树成为森林,破坏了整棵树的结构。然而,对于二叉排序树,删去树上一个结点相当于删去有序序列中的一个记录,只要在删除某个结点之后依旧保持二叉排序树的特性即可。

那么,如何在二叉排序树上删去一个结点呢?假设在二叉排序树上被删结点为 * p[①](指向结点的指针为 p),其双亲结点为 * f(结点指针为 f),且不失一般性,可设 * p 是 * f 的左孩子(图 9.9(a)所示)。

下面分 3 种情况进行讨论:

(1) 若 * p 结点为叶子结点,即 P_L 和 P_R 均为空树。由于删去叶子结点不破坏整棵树的结构,则只需修改其双亲结点的指针即可。

(2) 若 * p 结点只有左子树 P_L 或者只有右子树 P_R,此时只要令 P_L 或 P_R 直接成为其

① 以下均简称指针 p(或 f 等)所指结点为 * p(或 * f 等)结点,P_L 和 P_R 分别表示其左子树和右子树。

www.TopSage.com

双亲结点 * f 的左子树即可。显然，作此修改也不破坏二叉排序树的特性。

(3) 若 * p 结点的左子树和右子树均不空。显然，此时不能如上简单处理。从图 9.9(b)可知，在删去 * p 结点之前，中序遍历该二叉树得到的序列为{…C_LC…Q_LQS_LSPP_RF…}，在删去 * p 之后，为保持其他元素之间的相对位置不变，可以有两种做法：其一是令 * p 的左子树为 * f 的左子树，而 * p 的右子树为 * s 的右子树，如图 9.9(c)所示；其二是令 * p的直接前驱(或直接后继)替代 * p，然后再从二叉排序树中删去它的直接前驱(或直接后继)。如图 9.9(d)所示，当以直接前驱 * s 替代 * p 时，由于 * s 只有左子树 S_L，则在删去 * s 之后，只要令 S_L 为 * s 的双亲 * q 的右子树即可。

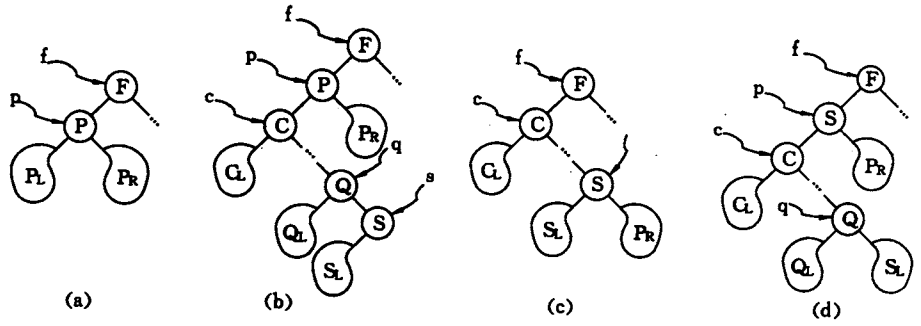


图 9.9 在二叉排序树中删除 * p

(a) 以 * f 为根的子树；(b) 删除 * p 之前；

(c) 删除 * p 之后，以 P_R 作为 * s 的右子树的情形；

(d) 删除 * p 之后，以 * s 替代 * p 的情形

在二叉排序树上删除一个结点的算法如算法 9.7 所示，其中由前述 3 种情况综合所得的删除操作如算法 9.8 所示。

```
Status DeleteBST (BiTree &T,  KeyType key ) {
  // 若二叉排序树 T 中存在关键字等于 key 的数据元素时，则删除该数据元素结点，
  // 并返回 TRUE；否则返回 FALSE
  if (!T) return FALSE;                                  // 不存在关键字等于 key 的数据元素
  else {
    if (EQ (key, T->data.key))  {return Delete (T)};  // 找到关键字等于 key 的数据元素
    else if (LT (key, T->data.key))  return DeleteBST ( T->lchild, key );
    else  return DeleteBST ( T->rchild, key );
  }
}// DeleteBST
```

算法 9.7

其中删除操作过程如算法 9.8 所描述：

```
Status Delete ( BiTree &p ){
  // 从二叉排序树中删除结点 p，并重接它的左或右子树
  if (!p->rchild) {   // 右子树空则只需重接它的左子树
    q = p;  p = p->lchild;  free(q);
```


www.TopSage.com

大家网
TopSage.com

```
  }
  else if (!p->lchild) {   // 只需重接它的右子树
    q = p;  p = p->rchild;  free(q);
  }
  else {   // 左右子树均不空
    q = p;  s = p->lchild;
    while (s->rchild) {q = s;  s = s->rchild }   // 转左,然后向右到尽头
    p->data = s->data;                           // s指向被删结点的"前驱"
    if (q != p ) q->rchild = s->lchild;          // 重接*q的右子树
    else  q->lchild = s->lchild;                 // 重接*q的左子树
    delete s;
  }
    return  TRUE;
}// Delete
```

算法 9.8

3. 二叉排序树的查找分析

从前述的两个查找例子($key=100$ 和 $key=40$)可见,在二叉排序树上查找其关键字等于给定值的结点的过程,恰是走了一条从根结点到该结点的路径的过程,和给定值比较的关键字个数等于路径长度加 1(或结点所在层次数),因此,和折半查找类似,与给定值比较的关键字个数不超过树的深度。然而,折半查找长度为 n 的表的判定树是惟一的,而含有 n 个结点的二叉排序树却不惟一。图 9.10 中(a)和(b)的两棵二叉排序树中结点的值都相同,但前者由关键字序列(45,24,53,12,37,93)构成,而后者由关键字序列(12,24,37,45,53,93)构成。(a)树的深度为 3,而(b)树的深度为 6。再从平均查找长度来看,假设 6 个记录的查找概率相等,为 1/6,则(a)树的平均查找长度为

$$ASL_{(a)} = \frac{1}{6}[1+2+2+3+3+3] = 14/6$$

而(b)树的平均查找长度为

$$ASL_{(b)} = \frac{1}{6}[1+2+3+4+5+6] = 21/6$$

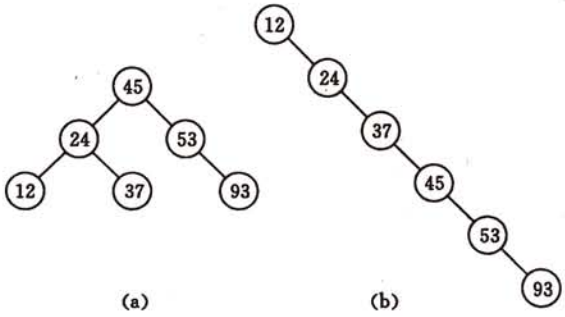


图 9.10 不同形态的二叉查找树

(a) 关键字序列为(45,24,53,12,37,93)的二叉排序树;

(b) 关键字序列为(12,24,37,45,53,93)的单支树

www.TopSage.com

因此，含有 n 个结点的二叉排序树的平均查找长度和树的形态有关。当先后插入的关键字有序时，构成的二叉排序树蜕变为单支树。树的深度为 n，其平均查找长度为 $\frac{n+1}{2}$（和顺序查找相同），这是最差的情况。显然，最好的情况是二叉排序树的形态和折半查找的判定树相同，其平均查找长度和 $\log_2 n$ 成正比。那么，它的平均性能如何呢？

假设在含有 $n(n\geqslant 1)$ 个关键字的序列中，i 个关键字小于第一个关键字，$n-i-1$ 个关键字大于第一个关键字，则由此构造而得的二叉排序树在 n 个记录的查找概率相等的情况下，其平均查找长度为

$$P(n,i)=\frac{1}{n}[1+i*(P(i)+1)+(n-i-1)(P(n-i-1)+1)] \qquad (9\text{-}17)$$

其中 $P(i)$ 为含有 i 个结点的二叉排序树的平均查找长度，则 $P(i)+1$ 为查找左子树中每个关键字时所用比较次数的平均值，$P(n-i-1)+1$ 为查找右子树中每个关键字时所用比较次数的平均值。又假设表中 n 个关键字的排列是“随机”的，即任一个关键字在序列中将是第 1 个，或第 2 个，……，或第 n 个的概率相同，则可对(9-17)式从 i 等于 0 至 $n-1$ 取平均值

$$\begin{aligned}P(n)&=\frac{1}{n}\sum_{i=0}^{n-1}P(n,i)\\&=1+\frac{1}{n^2}\sum_{i=0}^{n-1}[iP(i)+(n-i-1)P(n-i-1)]\end{aligned}$$

容易看出上式括弧中的第一项和第二项对称。又，$i=0$ 时 $iP(i)=0$，则上式可改写为

$$P(n)=1+\frac{2}{n^2}\sum_{i=1}^{n-1}iP(i) \qquad n\geqslant 2 \qquad (9\text{-}18)$$

显然，$P(0)=0$，$P(1)=1$。

由式(9-18)可推得

$$\sum_{j=0}^{n-1}jP(j)=\frac{n^2}{2}[P(n)-1]$$

又

$$\sum_{j=0}^{n-1}jP(j)=(n-1)P(n-1)+\sum_{j=0}^{n-2}jP(j)$$

由此可得

$$\frac{n^2}{2}[P(n)-1]=(n-1)P(n-1)+\frac{(n-1)^2}{2}[P(n-1)-1]$$

即

$$P(n)=\left(1-\frac{1}{n^2}\right)P(n-1)+\frac{2}{n}-\frac{1}{n^2} \qquad (9\text{-}19)$$

由递推公式(9-19)和初始条件 $P(1)=1$ 可推得：

$$\begin{aligned}P(n)&=2\frac{n+1}{n}\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n+1}\right)-1\\&=2\left(1+\frac{1}{n}\right)\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\right)+\frac{2}{n}-1\end{aligned}$$

则当 $n\geqslant 2$ 时

$$P(n)\leqslant 2\left(1+\frac{1}{n}\right)\ln n \qquad (9\text{-}20)$$

www.TopSage.com

大家网 TopSage.com

由此可见，在随机的情况下，二叉排序树的平均查找长度和 log$n$ 是等数量级的。然而，在某些情况下（有人研究证明，这种情况出现的概率约为 46.5%）[1]，尚需在构成二叉排序树的过程中进行“平衡化”处理，成为二叉平衡树。

4. 平衡二叉树

平衡二叉树（Balanced Binary Tree 或 Height-Balanced Tree）又称 AVL 树。它或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过 1。若将二叉树上结点的**平衡因子** BF（Balance Factor）定义为该结点的左子树的深度减去它的右子树的深度，则平衡二叉树上所有结点的平衡因子只可能是－1、0 和 1。只要二叉树上有一个结点的平衡因子的绝对值大于 1，则该二叉树就是不平衡的。如图 9.11(a)所示为两棵平衡二叉树，而图 9.11(b)所示为两棵不平衡的二叉树，结点中的值为该结点的平衡因子。

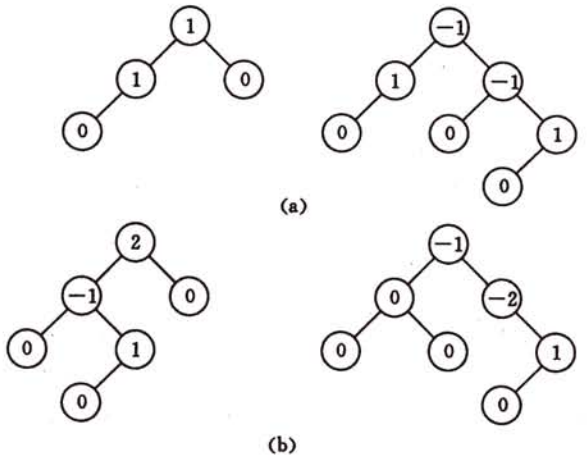


图 9.11　平衡与不平衡的二叉树及结点的平衡因子

（a）平衡二叉树；（b）不平衡的二叉树

我们希望由任何初始序列构成的二叉排序树都是 AVL 树。因为 AVL 树上任何结点的左右子树的深度之差都不超过 1，则可以证明它的深度和 log$n$ 是同数量级的（其中 n 为结点个数）。由此，它的平均查找长度也和 log$n$ 同数量级。

如何使构成的二叉排序树成为平衡树呢？先看一个具体例子（参见图 9.12）。假设表中关键字序列为(13,24,37,90,53)。空树和 1 个结点⑬的树显然都是平衡的二叉树。在插入 24 之后仍是平衡的，只是根结点的平衡因子 BF 由 0 变为－1；在继续插入 37 之后，由于结点⑬的 BF 值由－1 变成－2，由此出现了不平衡的现象。此时好比一根扁担出现一头重一头轻的现象，若能将扁担的支撑点由⑬改至㉔，扁担的两头就平衡了。由此，可以对树作一个向左逆时针“旋转”的操作，令结点㉔为根，而结点⑬为它的左子树，此时，结点⑬和㉔的平衡因子都为 0，而且仍保持二叉排序树的特性。在继续插入 90 和 53 之后，由于结点㊲的 BF 值由－1 变成－2，排序树中出现了新的不平衡的现象，需进行调整。但此时由于结点㊸插在结点㊿的左子树上，因此不能如上作简单调整。对于以结点㊲为根的子树来说，既要保持二叉排序树的特性，又要平衡，则必须以㊸作为根结点，而使㊲成

www.TopSage.com

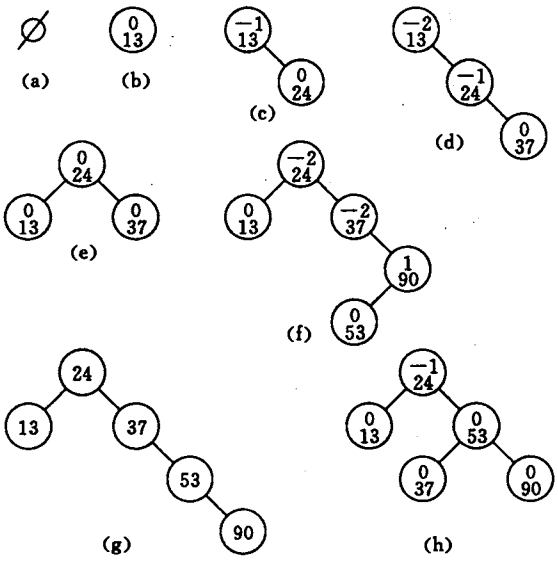


图 9.12　平衡树的生成过程

(a) 空树；(b) 插入 13；(c) 插入 24；(d)插入 37；(e) 向左逆时针右旋转平衡；
(f) 相继插入 90 和 53；(g)第一次向右顺时针旋转；(h) 第二次向左逆时针旋转平衡之

为它的左子树的根，⑩成为它的右子树的根。这好比对树作了两次“旋转”操作——先向右顺时针，后向左逆时针(见图 9.12(f)－(h))，使二叉排序树由不平衡转化为平衡。

一般情况下，假设由于在二叉排序树上插入结点而失去平衡的最小子树根结点的指针为 a(即 a 是离插入结点最近，且平衡因子绝对值超过 1 的祖先结点)，则失去平衡后进行调整的规律可归纳为下列 4 种情况：

(1) 单向右旋平衡处理：由于在 * a 的左子树根结点的左子树上插入结点，* a 的平衡因子由 1 增至 2，致使以 * a 为根的子树失去平衡，则需进行一次向右的顺时针旋转操作，如图 9.13(a)所示。

(2) 单向左旋平衡处理：由于在 * a 的右子树根结点的右子树上插入结点，* a 的平衡因子由－1 变为－2，致使以 * a 为根结点的子树失去平衡，则需进行一次向左的逆时针旋转操作。如图 9.13(c)所示。

(3) 双向旋转(先左后右)平衡处理：由于在 * a 的左子树根结点的右子树上插入结点，* a 的平衡因子由 1 增至 2，致使以 * a 为根结点的子树失去平衡，则需进行两次旋转(先左旋后右旋)操作。如图 9.13(b)所示。

(4) 双向旋转(先右后左)平衡处理：由于在 * a 的右子树根结点的左子树上插入结点，* a 的平衡因子由－1 变为－2，致使以 * a 为根结点的子树失去平衡，则需进行两次旋转(先右旋后左旋)操作。如图 9.13(d)所示。

上述 4 种情况中，(1)和(2)对称，(3)和(4)对称。旋转操作的正确性容易由“保持二叉排序树的特性：中序遍历所得关键字序列自小至大有序”证明之。同时，从图 9.13 可见，无论哪一种情况，在经过平衡旋转处理之后，以 * b 或 * c 为根的新子树为平衡二叉

www.TopSage.com

大家网
TopSage.com

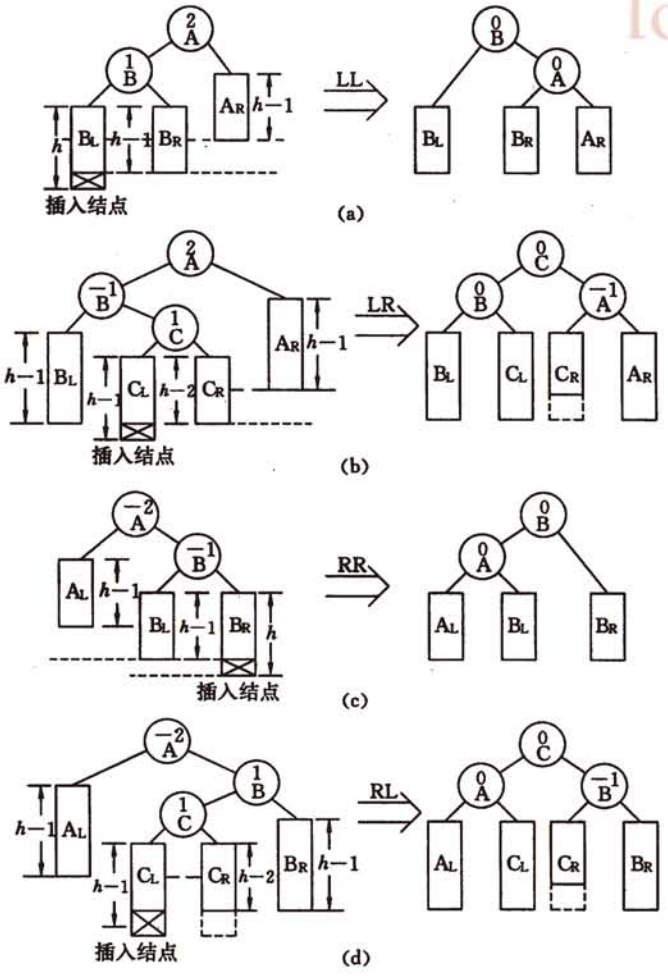


图 9.13　二叉排序树的平衡旋转图例

(a) LL 型；(b) LR 型；(c) RR 型；(d) RL 型

树，而且它的深度和插入之前以 * a 为根的子树相同。因此，当平衡的二叉排序树因插入结点而失去平衡时，仅需对最小不平衡子树进行平衡旋转处理即可。因为经过旋转处理之后的子树深度和插入之前相同，因而不影响插入路径上所有祖先结点的平衡度。

在平衡的二叉排序树 BBST 上插入一个新的数据元素 e 的递归算法可描述如下：

(1) 若 BBST 为空树，则插入一个数据元素为 e 的新结点作为 BBST 的根结点，树的深度增 1；

(2) 若 e 的关键字和 BBST 的根结点的关键字相等，则不进行插入；

(3) 若 e 的关键字小于 BBST 的根结点的关键字，而且在 BBST 的左子树中不存在和 e 有相同关键字的结点，则将 e 插入在 BBST 的左子树上，并且当插入之后的左子树深度增加(+1)时，分别就下列不同情况处理之：

① BBST 的根结点的平衡因子为 −1(右子树的深度大于左子树的深度)：则将根结

www.TopSage.com

点的平衡因子更改为0，BBST的深度不变；

② BBST的根结点的平衡因子为0(左、右子树的深度相等)：则将根结点的平衡因子更改为1，BBST的深度增1；

③ BBST的根结点的平衡因子为1(左子树的深度大于右子树的深度)：若BBST的左子树根结点的平衡因子为1，则需进行单向右旋平衡处理，并且在右旋处理之后，将根结点和其右子树根结点的平衡因子更改为0，树的深度不变；

若BBST的左子树根结点的平衡因子为－1，则需进行先向左、后向右的双向旋转平衡处理，并且在旋转处理之后，修改根结点和其左、右子树根结点的平衡因子，树的深度不变；

(4) 若e的关键字大于BBST的根结点的关键字，而且在BBST的右子树中不存在和e有相同关键字的结点，则将e插入在BBST的右子树上，并且当插入之后的右子树深度增加(＋1)时，分别就不同情况处理之。其处理操作和(三)中所述相对称，读者可自行补充。

假设在“6.2.3 二叉树的存储结构”中定义的二叉链表的结点中增加一个存储结点平衡因子的域bf，则上述在平衡的二叉排序树BBST上插入一个新的数据元素e的递归算法如算法9.11所示，其中，左平衡处理的算法如算法9.12所示。算法9.9和算法9.10分别描述了在平衡处理中进行右旋操作和左旋操作时修改指针的情况。右平衡处理的算法和左平衡处理的算法类似，读者可自己补充。

二叉排序树的类型定义为：

```
typedef struct BSTNode {
  ElemType          data;
  int               bf;                 // 结点的平衡因子
  struct BSTNode * lchild, * rchild; // 左、右孩子指针
}BSTNode, * BSTree;

void R_Rotate ( BSTree &p ) {
  // 对以 * p 为根的二叉排序树作右旋处理，处理之后 p 指向新的树根结点，即旋转
  // 处理之前的左子树的根结点
  lc = p->lchild;                  // lc 指向的 * p 的左子树根结点
  p->lchild = lc->rchild;          // lc 的右子树挂接为 * p 的左子树
  lc->rchild = p; p = lc;          // p 指向新的根结点
}// R_Rotate
```

算法 9.9

```
void L_Rotate ( BSTree &p ) {
  // 对以 * p 为根的二叉排序树作左旋处理，处理之后 p 指向新的树根结点，即旋转
  // 处理之前的右子树的根结点
  rc = p->rchild;                  // rc 指向的 * p 的右子树根结点
  p->rchild = rc->lchild;          // rc 的左子树挂接为 * p 的右子树
  rc->lchild = p; p = rc;          // p 指向新的根结点
}// L_Rotate
```

算法 9.10

www.TopSage.com

大家网
TopSage.com

```
#define LH +1        // 左高
#define EH  0        // 等高
#define RH -1        // 右高
Status InsertAVL (BSTree &T, ElemType e, Boolean &taller) {
  // 若在平衡的二叉排序树 T 中不存在和 e 有相同关键字的结点,则插入一个数据元素
  // 为 e 的新结点,并返回 1,否则返回 0。若因插入而使二叉排序树失去平衡,则作平衡
  // 旋转处理,布尔变量 taller 反映 T 长高与否
  if (!T) {// 插入新结点,树"长高",置 taller 为 TRUE
    T = (BSTree) malloc (sizeof(BSTNode));  T->data = e;
    T->lchild = T->rchild = NULL;  T->bf = EH;  taller = TRUE;
  }
  else {
    if (EQ(e.key, T->data.key))              // 树中已存在和 e 有相同关键字的结点
      { taller = FALSE;  return 0;  }        // 则不再插入
    if (LT(e.key, T->data.key)) {            // 应继续在 *T 的左子树中进行搜索
      if (!InsertAVL (T->lchild, e, taller) )  return 0;     //未插入
      if (taller)                            // 已插入到 *T 的左子树中且左子树"长高"
        switch (T->bf){                      // 检查 *T 的平衡度
          case LH:                           // 原本左子树比右子树高,需要作左平衡处理
            LeftBalance ( T );  taller = FALSE;break;
          case EH:                           // 原本左、右子树等高,现因左子树增高而使树增高
            T->bf = LH;  taller = TRUE;  break;
          case RH:                           // 原本右子树比左子树高,现左、右子树等高
            T->bf = EH;  taller = FALSE;  break;
        }// switch (T->bf)
    }// if
    else {                                   // 应继续在 *T 的右子树中进行搜索
      if (!InsertAVL (T->rchild, e, taller) ) return 0;     //未插入
      if (taller)                            // 已插入到 *T 的右子树且右子树长高
        switch (T->bf){                      // 检查 *T 的平衡度
          case LH:                           // 原本左子树比右子树高,现左、右子树等高
            T->bf = EH;  taller = FALSE;  break;
          case EH:                           // 原本左、右子树等高,现因右子树增高而使树增高
            T->bf = RH;  taller = TRUE;  break;
          case RH:                           // 原本右子树比左子树高,需要作右平衡处理
            RightBalance ( T );  taller = FALSE;  break;
        }// switch (T->bf)
      }// else
    }// else
    return 1;
}// InsertAVL
```

算法 9.11

```
void LeftBalance ( BSTree &T ){
  // 对以指针 T 所指结点为根的二叉树作左平衡旋转处理,本算法结束时,指针 T 指向
  // 新的根结点
  lc = T->lchild;              // lc 指向 *T 的左子树根结点
  switch (lc->bf) {            // 检查 *T 的左子树的平衡度,并作相应平衡处理
    case LH:                   // 新结点插入在 *T 的左孩子的左子树上,要作单右旋处理
        T->bf = lc->bf = EH;
        R_Rotate( T );  break;
    case RH:                   // 新结点插入在 *T 的左孩子的右子树上,要作双旋处理
        rd = lc->rchild;       // rd 指向 *T 的左孩子的右子树根
```


www.TopSage.com

```
        switch (rd->bf) {    // 修改 *T 及其左孩子的平衡因子
            case LH: T->bf = RH;  lc->bf = EH;  break;
            case EH: T->bf = lc->bf = EH;       break;
            case RH: T->bf = EH;  lc->bf = LH;  break;
        }// switch (rd->bf)
        rd->bf = EH;
        L_Rotate ( T->lchild );     // 对 *T 的左子树作左旋平衡处理
        R_Rotate ( T );             // 对 *T 作右旋平衡处理
    }// switch (lc->bf)
} // LeftBalance
```

算法 9.12

5. 平衡树查找的分析

在平衡树上进行查找的过程和排序树相同，因此，在查找过程中和给定值进行比较的关键字个数不超过树的深度。那么，含有 n 个关键字的平衡树的最大深度是多少呢？为解答这个问题，我们先分析深度为 h 的平衡树所具有最少结点数。

假设以 N_h 表示深度为 h 的平衡树中含有的最少结点数。显然，$N_0=0$，$N_1=1$，$N_2=2$，并且 $N_h=N_{h-1}+N_{h-2}+1$。这个关系和斐波那契序列极为相似。利用归纳法容易证明：当 $h\geqslant 0$ 时 $N_h=F_{h+2}-1$，而 F_h 约等于 $\varphi^h/\sqrt{5}$（其中 $\varphi=\frac{1+\sqrt{5}}{2}$）[1]，则 N_h 约等于 $\varphi^{h+2}/\sqrt{5}-1$。反之，含有 n 个结点的平衡树的最大深度为 $\log_\varphi(\sqrt{5}(n+1))-2$。因此，在平衡树上进行查找的时间复杂度为 $O(\log n)$。

上述对二叉排序树和二叉平衡树的查找性能的讨论都是在等概率的前提下进行的，若查找概率不等，则类似于“9.1.3 静态树表的查找”中的讨论。为了提高查找效率，需要对待查记录序列先进行排序，使其按关键字递增（或递减）有序，然后再按算法 9.4 构造一棵次优查找树。显然，次优查找树也是一棵二叉排序树，但次优查找树不能在查找过程中插入结点生成。二叉排序树（或称二叉查找树）是动态树表，最优或次优查找树是静态树表。

9.2.2 B-树和 B^+ 树

1. B-树及其查找

B-树是一种平衡的多路查找树，它在文件系统中很有用。在此先介绍这种树的结构及其查找算法。

一棵 m 阶的 B-树，或为空树，或为满足下列特性的 m 叉树：

(1) 树中每个结点至多有 m 棵子树；

(2) 若根结点不是叶子结点，则至少有两棵子树；

(3) 除根之外的所有非终端结点至少有$\lceil m/2 \rceil$棵子树；

(4) 所有的非终端结点中包含下列信息数据

$$(n, A_0, K_1, A_1, K_2, A_2, \cdots, K_n, A_n)$$ ①

① 实际上在 B-树的每个结点中还应包含 n 个指向每个关键字的记录的指针。

www.TopSage.com

大家网 TopSage.com

其中：$K_i(i=1,\cdots,n)$为关键字，且 $K_i<K_{i+1}(i=1,\cdots,n-1)$；$A_i(i=0,\cdots,n)$为指向子树根结点的指针，且指针 A_{i-1}所指子树中所有结点的关键字均小于 $K_i(i=1,\cdots,n)$，A_n所指子树中所有结点的关键字均大于 K_n，$n(\lceil m/2\rceil-1\leqslant n\leqslant m-1)$为关键字的个数(或 $n+1$为子树个数)。

(5) 所有的叶子结点都出现在同一层次上，并且不带信息(可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空)。

例如图 9.14 所示为一棵 4 阶的 B-树，其深度为 4。

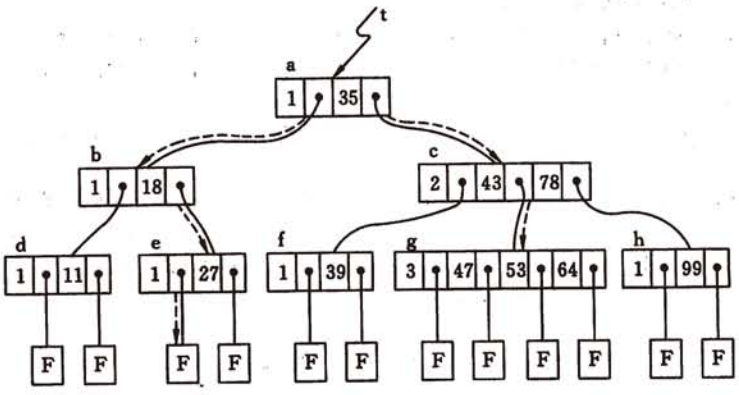


图 9.14　一棵 4 阶的 B-树

由 B-树的定义可知，在 B-树上进行查找的过程和二叉排序树的查找类似。例如，在图 9.14 的 B-树上查找关键字 47 的过程如下：首先从根开始，根据根结点指针 t 找到 * a 结点，因 * a 结点中只有一个关键字，且给定值 47＞关键字 35，则若存在必在指针 A_1 所指的子树内，顺指针找到 * c 结点，该结点有两个关键字(43 和 78)，而 43＜47＜78，则若存在必在指针 A_1 所指的子树中。同样，顺指针找到 * g 结点，在该结点中顺序查找找到关键字 47，由此，查找成功。查找不成功的过程也类似，例如在同一棵树中查找 23。从根开始，因为 23＜35，则顺该结点中指针 A_0 找到 * b 结点，又因为 * b 结点中只有一个关键字 18，且 23＞18，所以顺结点中第二个指针 A_1 找到 * e 结点。同理因为 23＜27，则顺指针往下找，此时因指针所指为叶子结点，说明此棵 B-树中不存在关键字 23，查找因失败而告终。

由此可见，在 B-树上进行查找的过程是一个顺指针查找结点和在结点的关键字中进行查找交叉进行的过程。

由于 B-树主要用作文件的索引，因此它的查找涉及外存的存取，在此略去外存的读写，只作示意性的描述。假设结点类型如下说明：

```
#define  m  3                          // B-树的阶，暂设为 3
typedef struct BTNode {
   int              keynum;            // 结点中关键字个数，即结点的大小
   struct BTNode   *parent;            // 指向双亲结点
   KeyType          key[m+1];          // 关键字向量，0 号单元未用
   struct BTNode   *ptr[m+1];          // 子树指针向量
```


www.TopSage.com

```
  Record          * recptr[m+1];        // 记录指针向量,0 号单元未用
}BTNode, * BTree;                       // B-树结点和 B-树的类型
typedef struct {
  BTNode * pt;                          // 指向找到的结点
  int      i;                           // 1..m,在结点中的关键字序号
  int      tag;                         // 1:查找成功,0:查找失败
}Result;                                // B-树的查找结果类型
```

则算法 9.13 简要地描述了 B-树的查找操作的实现。

```
Result SearchBTree(BTree T, KeyType K) {
  // 在 m 阶 B-树 T 上查找关键字 K,返回结果(pt,i,tag)。若查找成功,则特征值 tag=1,指针 pt
  // 所指结点中第 i 个关键字等于 K;否则特征值 tag=0,等于 K 的关键字应插入在指针 pt 所指
  // 结点中第 i 和第 i+1 个关键字之间
  p=T;  q=NULL;  found=FALSE;  i=0;   // 初始化,p 指向待查结点,q 指向 p 的双亲
  while (p && !found) {
    i=Search(p, K);                   // 在 p->key[1..keynum]中查找,
                                      // i 使得:p->key[i]<=K<p->key[i+1]
    if (i>0 && p->key[i]==K) found=TRUE;  // 找到待查关键字
    else {q=p;  p=p->ptr[i]; }
  }
  if (found) return (p,i,1);          // 查找成功
  else return (q,i,0);                // 查找不成功,返回 K 的插入位置信息
}// SearchBTree
```

算法 9.13

2. B-树查找分析

从算法 9.11 可见,在 B-树上进行查找包含两种基本操作:(1)在 B-树中找结点;(2)在结点中找关键字。由于 B-树通常存储在磁盘上,则前一查找操作是在磁盘上进行的(在算法 9.11 中没有体现),而后一查找操作是在内存中进行的,即在磁盘上找到指针 p 所指结点后,先将结点中的信息读入内存,然后再利用顺序查找或折半查找查询等于 K 的关键字。显然,在磁盘上进行一次查找比在内存中进行一次查找耗费时间多得多,因此,在磁盘上进行查找的次数、即待查关键字所在结点在 B-树上的层次数,是决定 B-树查找效率的首要因素。

现考虑最坏的情况,即待查结点在 B-树上的最大层次数。也就是,含 N 个关键字的 m 阶 B-树的最大深度是多少?

先看一棵 3 阶的 B-树。按 B-树的定义,3 阶的 B-树上所有非终端结点至多可有两个关键字,至少有一个关键字(即子树个数为 2 或 3,故又称 2-3 树)。因此,若关键字个数≤2 时,树的深度为 2 (即叶子结点层次为 2);若关键字个数≤6 时,树的深度不超过 3。反之,若 B-树的深度为 4,则关键字的个数必须≥7(参见图 9.15(g)),此时,每个结点都含有可能的关键字的最小数目。

一般情况的分析可类似二叉平衡树进行,先讨论深度为 $l+1$ 的 m 阶 B-树所具有的最少结点数。

根据 B-树的定义,第一层至少有 1 个结点;第二层至少有 2 个结点;由于除根之外的

www.TopSage.com

大家网
TopSage.com

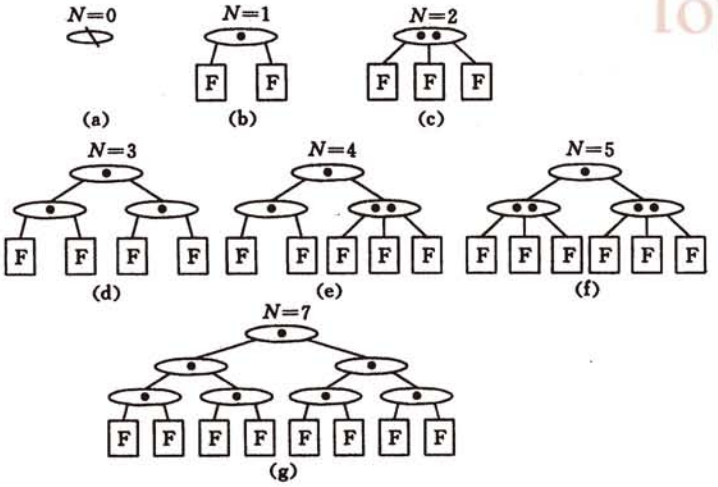


图 9.15　不同关键字数目的 B-树

(a) 空树；(b) $N=1$；(c) $N=2$；(d) $N=3$；(e) $N=4$；(f) $N=5$；(g) $N=7$

每个非终端结点至少有$\lceil m/2\rceil$ 棵子树，则第三层至少有 $2(\lceil m/2\rceil)$个结点；……；依次类推，第 $l+1$ 层至少有 $2(\lceil m/2\rceil)^{l-1}$个结点。而 $l+1$ 层的结点为叶子结点。若 m 阶 B-树中具有 N 个关键字，则叶子结点即查找不成功的结点为 $N+1$，由此有：

$$N+1 \geqslant 2*(\lceil m/2\rceil)^{l-1}$$

反之

$$l \leqslant \log_{\lceil m/2\rceil}\left(\frac{N+1}{2}\right)+1 \tag{9-21}$$

这就是说，在含有 N 个关键字的 B-树上进行查找时，从根结点到关键字所在结点的路径上涉及的结点数不超过 $\log_{\lceil m/2\rceil}\left(\frac{N+1}{2}\right)+1$。

3. B-树的插入和删除

B-树的生成也是从空树起，逐个插入关键字而得。但由于 B-树结点中的关键字个数必须$\geqslant\lceil m/2\rceil-1$，因此，每次插入一个关键字不是在树中添加一个叶子结点，而是首先在最低层的某个非终端结点中添加一个关键字，若该结点的关键字个数不超过 $m-1$，则插入完成，否则要产生结点的“分裂”，如图 9.16 所示。

例如，图 9.16(a)所示为 3 阶的 B-树(图中略去 F 结点(即叶子结点))，假设需依次插入关键字 30，26，85 和 7。首先通过查找确定应插入的位置。由根 * a 起进行查找，确定 30 应插入在 * d 结点中，由于 * d 中关键字数目不超过 2(即 $m-1$)，故第一个关键字插入完成。插入 30 后的 B-树如图 9.16(b)所示。同样，通过查找确定关键字 26 亦应插入在 * d 结点中。由于 * d 中关键字的数目超过 2，此时需将 * d 分裂成两个结点，关键字 26 及其前、后两个指针仍保留在 * d 结点中，而关键字 37 及其前、后两个指针存储到新产生的结点 * d′中。同时，将关键字 30 和指示结点 * d′的指针插入到其双亲结点中。由于 * b 结点中的关键字数目没有超过 2，则插入完成。插入后的 B-树如图 9.16(d)所示。类似

www.TopSage.com

地，在 * g中插入 85 之后需分裂成两个结点，而当 70 继而插入到双亲结点时，由于 * e 中关键字数目超过 2，则再次分裂为结点 * e 和 * e′，如图 9.16(g)所示。最后在插入关键字 7 时，* c、* b 和 * a 相继分裂，并生成一个新的根结点 * m，如图 9.16(h)～(j)所示。

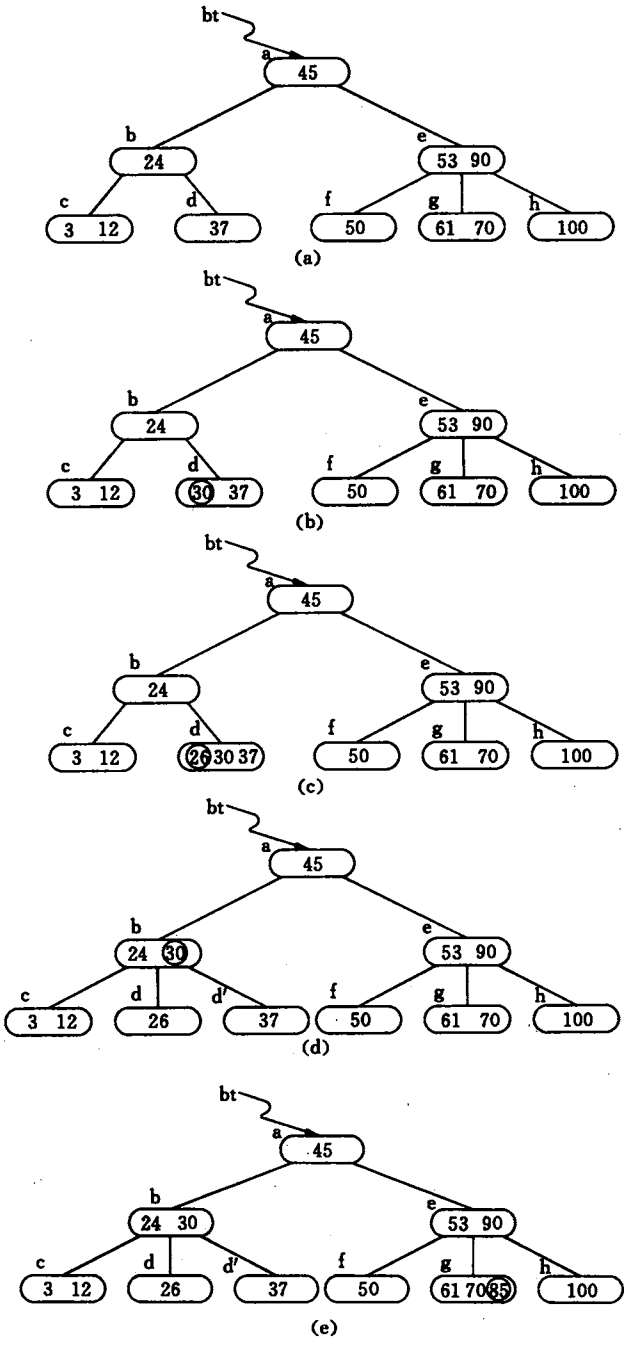


图 9.16　在 B-树中进行插入(省略叶子结点)

(a) 一棵 2-3 树；(b) 插入 30 之后；(c)、(d) 插入 26 之后；(e)～(g) 插入 85 之后；(h)～(j) 插入 7 之后

www.TopSage.com

大家网
TopSage.com

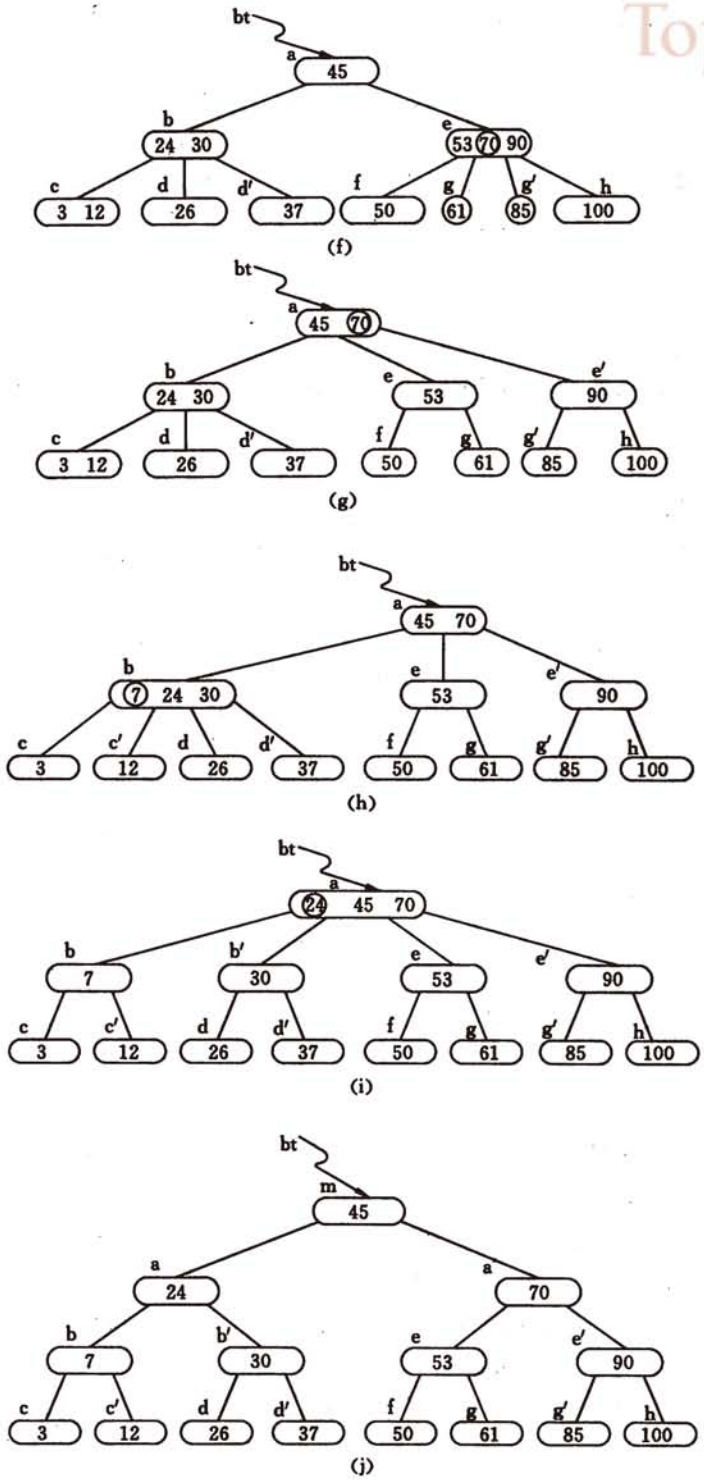


图 9.16 （续）

www.TopSage.com

一般情况下，结点可如下实现“分裂”。

假设 * p 结点中已有 $m-1$ 个关键字，当插入一个关键字之后，结点中含有信息为：

$$m, A_0, (K_1, A_1), \cdots, (K_m, A_m)$$

且其中 $K_i < K_{i+1}$ $1 \leqslant i < m$

此时可将 * p 结点分裂为 * p 和 * p′两个结点，其中 * p 结点中含有信息为

$$\lceil m/2 \rceil - 1, A_0, (K_1, A_1), \cdots, (K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1}) \tag{9-22}$$

* p′结点中含有信息

$$m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (K_{\lceil m/2+1 \rceil}, A_{\lceil m/2+1 \rceil}), \cdots, (K_m, A_m) \tag{9-23}$$

而关键字 $K_{\lceil m/2 \rceil}$和指针 * p′一起插入到 * p 的双亲结点中。

在 B-树上插入关键字的过程如算法 9.14 所示，其中 q 和 i 是由查找函数 SearchBTree 返回的信息而得。

```
Status InsertBTree (BTree &T, KeyType K, BTree q, int i ) {
  // 在 m 阶 B-树 T 上结点 * q 的 key[i]与 key[i + 1]之间插入关键字 K。
  // 若引起结点过大，则沿双亲链进行必要的结点分裂调整，使 T 仍是 m 阶 B-树。
  x = K;  ap = NULL;  finished = FALSE;
  while (q && !finished) {
    Insert(q, i, x, ap);      // 将 x 和 ap 分别插入到 q->key[i + 1]和 q->ptr[i + 1]
    if (q->keynum < m) finished = TRUE;    // 插入完成
    else {                                 // 分裂结点 * q
      s = ⌈m/2⌉;  split(q, s, ap);  x = q->key[s];
      // 将 q->key[s + 1..m], q->ptr[s..m]和 q->recptr[s + 1..m]移入新结点 * ap
      q = q->parent;
      if (q)  i = Search(q, x);            // 在双亲结点 * q 中查找 x 的插入位置
    }// else
  }// while
  if (!finished)           // T 是空树(参数 q 初值为 NULL)或者根结点已分裂为结点 * q 和 * ap
    NewRoot(T, q, x, ap);  // 生成含信息(T,x,ap)的新的根结点 * T,原 T 和 ap 为子树指针
  return  OK;
}// InsertBTree
```

算法 9.14

反之，若在 B-树上删除一个关键字，则首先应找到该关键字所在结点，并从中删除之，若该结点为最下层的非终端结点，且其中的关键字数目不少于$\lceil m/2 \rceil$，则删除完成，否则要进行“合并”结点的操作。假若所删关键字为非终端结点中的 K_i，则可以指针 A_i 所指子树中的最小关键字 Y 替代 K_i，然后在相应的结点中删去 Y。例如，在图 9.16(a)的 B-树上删去 45，可以 * f 结点中的 50 替代 45，然后在 * f 结点中删去 50。因此，下面我们可以只需讨论删除最下层非终端结点中的关键字的情形。有下列 3 种可能：

(1) 被删关键字所在结点中的关键字数目不小于$\lceil m/2 \rceil$则只需从该结点中删去该关键字 K_i 和相应指针 A_i，树的其他部分不变，例如，从图 9.16(a)所示 B-树中删去关键字 12，删除后的 B-树如图 9.17(a)所示。

(2) 被删关键字所在结点中的关键字数目等于$\lceil m/2 \rceil - 1$，而与该结点相邻的右兄弟(或左兄弟)结点中的关键字数目大于$\lceil m/2 \rceil - 1$，则需将其兄弟结点中的最小(或最

www.TopSage.com

大家网
TopSage.com

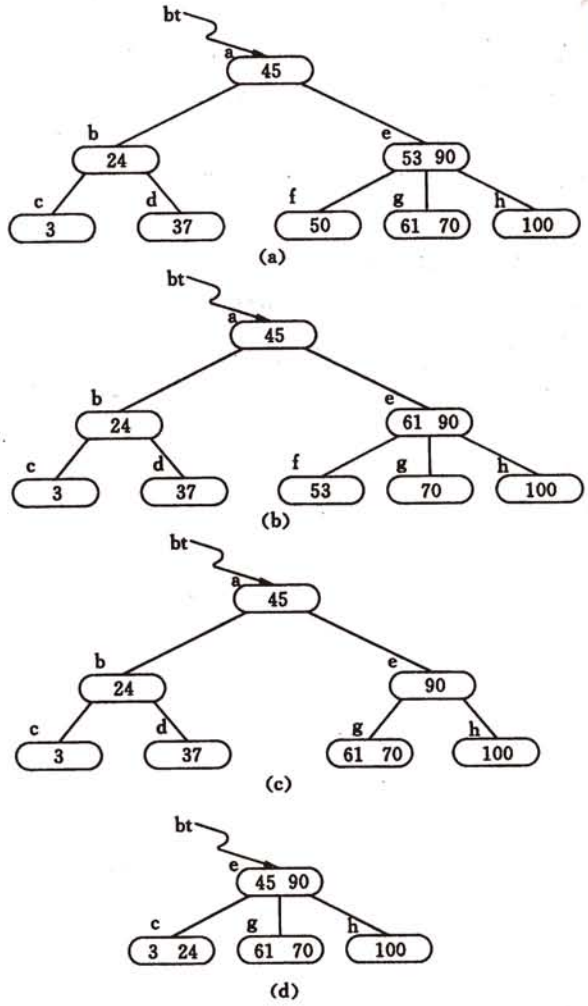


图 9.17　在 B-树中删除关键字的情形

大)的关键字上移至双亲结点中,而将双亲结点中小于(或大于)且紧靠该上移关键字的关键字下移至被删关键字所在结点中。例如,从图 9.17(a)中删去 50,需将其右兄弟结点中的 61 上移至 * e 结点中,而将 * e 结点中的 53 移至 * f,从而使 * f 和 * g 中关键字数目均不小于$\lceil m/2 \rceil - 1$,而双亲结点中的关键字数目不变,如图 9.17(b)所示。

(3) 被删关键字所在结点和其相邻的兄弟结点中的关键字数目均等于$\lceil m/2 \rceil - 1$。假设该结点有右兄弟,且其右兄弟结点地址由双亲结点中的指针 A_i 所指,则在删去关键字之后,它所在结点中剩余的关键字和指针,加上双亲结点中的关键字 K_i 一起,合并到 A_i 所指兄弟结点中(若没有右兄弟,则合并至左兄弟结点中)。例如,从图 9.17(b)所示 B-树中删去 53,则应删去 * f 结点,并将 * f 中的剩余信息(指针"空")和双亲 * e 结点中的 61 一起合并到右兄弟结点 * g 中。删除后的树如图 9.17(c)所示。如果因此使双亲结点中的关键字数目小于$\lceil m/2 \rceil - 1$,则依次类推作相应处理。例如,在图 9.17(c)的 B-树中

www.TopSage.com

删去关键字 37 之后，双亲 b 结点中剩余信息（“指针 c”）应和其双亲 * a 结点中关键字 45 一起合并至右兄弟结点 * e 中，删除后的 B-树如图 9.17(d)所示。

在 B-树中删除结点的算法在此不再详述，请读者参阅参考书目[1]后自己写出。

4. B^+ 树

B^+ 树是应文件系统所需而出的一种 B-树的变型树①。一棵 m 阶的 B^+ 树和 m 阶的 B-树的差异在于：

(1) 有 n 棵子树的结点中含有 n 个关键字。

(2) 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

(3) 所有的非终端结点可以看成是索引部分，结点中仅含有其子树（根结点）中的最大（或最小）关键字。

例如图 9.18 所示为一棵 3 阶的 B^+ 树，通常在 B^+ 树上有两个头指针，一个指向根结点，另一个指向关键字最小的叶子结点。因此，可以对 B^+ 树进行两种查找运算：一种是从最小关键字起顺序查找，另一种是从根结点开始，进行随机查找。

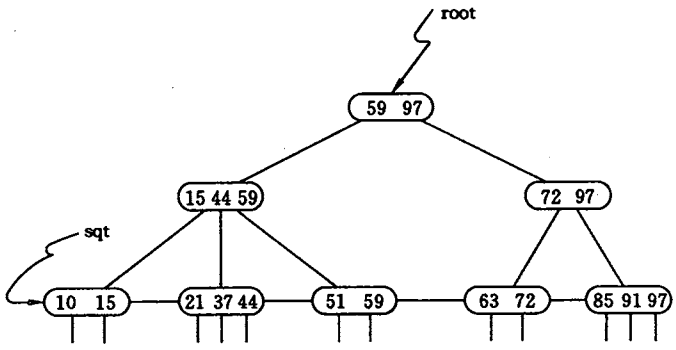


图 9.18 一棵 3 阶的 B^+ 树

在 B^+ 树上进行随机查找、插入和删除的过程基本上与 B-树类似。只是在查找时，若非终端结点上的关键字等于给定值，并不终止，而是继续向下直到叶子结点。因此，在 B^+ 树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。B^+ 树查找的分析类似于 B-树。B^+ 树的插入仅在叶子结点上进行，当结点中的关键字个数大于 m 时要分裂成两个结点，它们所含关键字的个数分别为$\lceil \frac{m+1}{2} \rceil$和$\lceil \frac{m+1}{2} \rceil$。并且，它们的双亲结点中应同时包含这两个结点中的最大关键字。B^+ 树的删除也仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界关键字”存在。若因删除而使结点中关键字的个数少于$\lceil \frac{m}{2} \rceil$时，其和兄弟结点的合并过程亦和 B-树类似。

① 严格说来，它已不是第六章中定义的树了。

www.TopSage.com

大家网
TopSage.com

9.2.3 键树

键树又称**数字查找树**(Digital Search Trees)。它是一棵度≥2 的树,树中的每个结点中不是包含一个或几个关键字,而是只含有组成关键字的符号。例如,若关键字是数值,则结点中只包含一个数位;若关键字是单词,则结点中只包含一个字母字符。这种树会给某种类型关键字的表的查找带来方便。

假设有如下 16 个关键字的集合

{CAI、CAO、LI、LAN、CHA、CHANG、WEN、CHAO、YUN、YANG、LONG、WANG、ZHAO、LIU、WU、CHEN} (9-24)

可对此集合作如下的逐层分割。

首先按其首字符不同将它们分成 5 个子集:

{CAI、CAO、CHA、CHANG、CHAO、CHEN},{WEN、WANG、WU},{ZHAO},{LI、LAN、LONG、LIU},{YUN、YANG},

然后对其中 4 个关键字个数大于 1 的子集再按其第二个字符不同进行分割。若所得子集的关键字多于 1 个,则还需按其第三个字符不同进行再分割。依此类推,直至每个小子集中只包含一个关键字为止。例如对首字符为 C 的集合可进行如下的分割:

{{(CAI)、(CAO)}、{{(CHA)、(CHANG)、(CHAO)}、(CHEN)}}

显然,如此集合、子集和元素之间的层次关系可以用一棵树来表示,这棵树便为键树。例如,上述集合及其分割可用图 9.19 所示的键树来表示。树中根结点的五棵子树分别表示

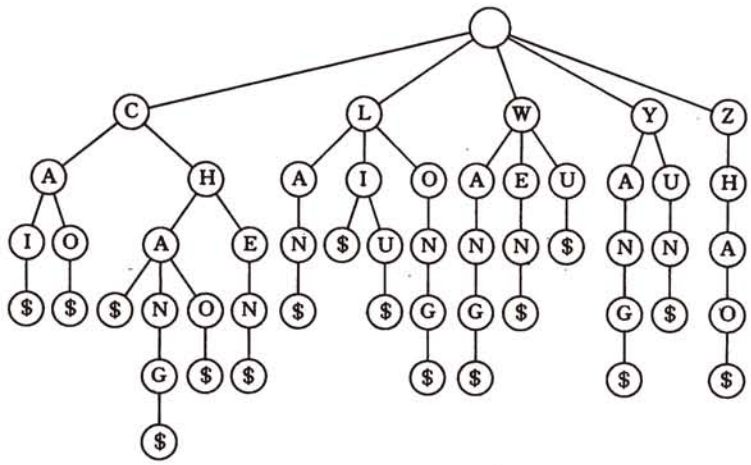


图 9.19 表示式(9-24)关键字集的一棵键树

首字符为 C、L、W、Y 和 Z 的 5 个关键字子集。从根到叶子结点路径中结点的字符组成的字符串表示一个关键字,叶子结点中的特殊符号 $ 表示字符串的结束。在叶子结点还含有指向该关键字记录的指针。

为了查找和插入方便,我们约定键树是有序树,即同一层中兄弟结点之间依所含符号自左至右有序,并约定结束符 $ 小于任何字符。

www.TopSage.com

通常，键树可有两种存储结构。

(1) 以树的孩子兄弟链表来表示键树，则每个分支结点包括 3 个域：symbol 域：存储关键字的一个字符；first 域：存储指向第一棵子树根的指针；next 域：存储指向右兄弟的指针。同时，叶子结点的 infoptr 域存储指向该关键字记录的指针。此时的键树又称**双链树**。例如，图 9.19 所示键树的双链树如图 9.20所示(图中只画出第一棵子树，其余部分省略)。

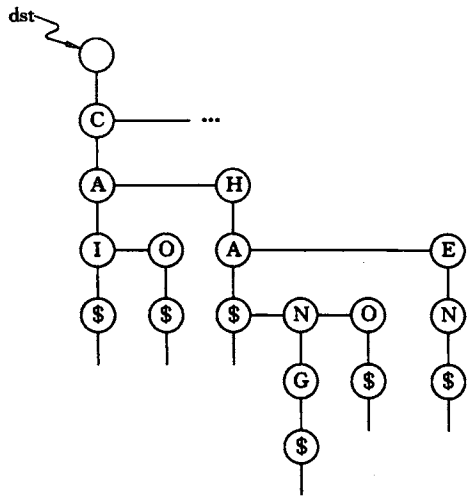


图 9.20 双链树示例

双链树的查找可如下进行：假设给定值为 K.ch(0..num－1)，其中 K.ch[0]至 K.ch[num－2]表示待查关键字中 num－1 个字符，K.ch[num－1]为结束符 $，从双链树的根指针出发，顺 first 指针找到第一棵子树的根结点，以 K.ch[0]和此结点的 symbol 域比较，若相等，则顺 first 域再比较下一字符，否则沿 next 域顺序查找。若直至“空”仍比较不等，则查找不成功。

如果对双链树采用以下存储表示

```
#define MAXKEYLEN  16          // 关键字的最大长度
typedef struct {
  char ch[MAXKEYLEN];          // 关键字
  int num;                     // 关键字长度
}KeysType;                     // 关键字类型
typedef enum { LEAF, BRANCH }  NodeKind;  // 结点种类:{叶子,分支}
typedef struct DLTNode {
  char symbol;
  struct DLTNode  * next;      // 指向兄弟结点的指针
  NodeKind  kind;
  union {
    Record   * infoptr;        // 叶子结点的记录指针
    struct DLTNode   * first;  // 分支结点的孩子链指针
  }
}DLTNode, * DLTree;            // 双链树的类型
```

则在双链树中查找记录的操作由算法 9.15 实现。

```
Record * SearchDLTree (DLTree T, KeysType K) {
  // 在非空双链树 T 中查找关键字等于 K 的记录,若存在,则返回指向该记录的指针,否则返回空
  // 指针
  p = T->first;     i = 0;      // 初始化
  while (p && i < K.num) {
    while (p && p->symbol != K.ch[i]) p = p->next; // 查找关键字的第 i 位
    if (p && i<K.num - 1)  p = p->first;           // 准备查找下一位
    ++ i;
```


www.TopSage.com

大家网
TopSage.com

```
    }   // 查找结束
    if (!p)then return NULL;          // 查找不成功
    else return   p->infoptr;         // 查找成功
}//Search DLTree
```

算法 9.15

键树中每个结点的最大度 d 和关键字的“基”有关，若关键字是单词，则 $d=27$，若关键字是数值，则 $d=11$。键树的深度 h 则取决于关键字中字符或数位的个数。假设关键字为随机的（即关键字中每一位取基内任何值的概率相同），则在双链树中查找每一位的平均查找长度为 $\frac{1}{2}(1+d)$。又假设关键字中字符（或数位）的个数都相等，则在双链树中进行查找的平均查找长度为 $\frac{h}{2}(1+d)$。

在双链树中插入或删除一个关键字，相当于在树中某个结点上插入或删除一棵子树，在此不再详述。

（2）若以树的多重链表表示键树，则树的每个结点中应含有 d 个指针域，此时的键树又称 Trie 树[①]。若从键树中某个结点到叶子结点的路径上每个结点都只有一个孩子，则可将该路径上所有结点压缩成一个“叶子结点”，且在该叶子结点中存储关键字及指向记录的指针等信息。例如，图 9.19 所示键树中，从结点 Z 到结点 $ 为单支树，则在图 9.21 相应的 Trie 树中只有一个含有关键字 ZHAO 及相关信息的叶子结点。由此，在 Trie 树中有两种结点：分支结点（含有 d 个指针域和一个指示该结点中非空指针域的个数的整数域）和叶子结点（含有关键字域和指向记录的指针域）。在分支结点中不设数据域，每个分支结点所表示的字符均由其双亲结点中（指向该结点）的指针所在位置决定。

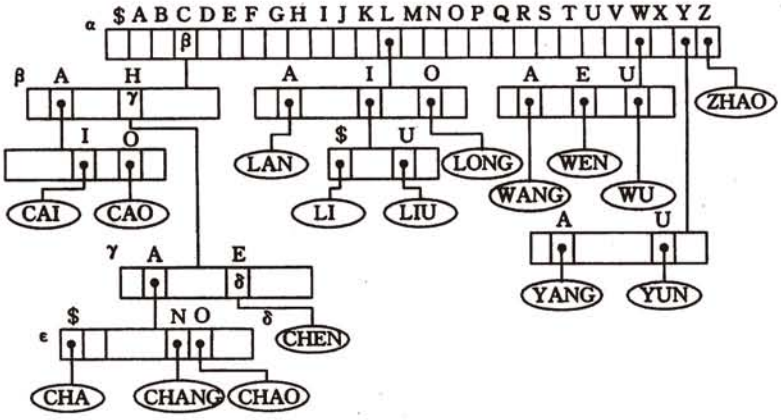


图 9.21　表示(9-24)关键字集的 Trie 树（深度＝5）

在 Trie 树上进行查找的过程为：从根结点出发，沿和给定值相应的指针逐层向下，直至叶子结点，若叶子结点中的关键字和给定值相等，则查找成功，若分支结点中和给定值

① trie 这个词是从 retrieve（检索）中取中间四个字符而构成，读音同（try）。

www.TopSage.com

相应的指针为空，或叶结点中的关键字和给定值不相等，则查找不成功。若设

```
typedef struct TrieNode {
  NodeKind  kind;
  union {
    struct {KeysType  K;  Record  * infoptr; } lf;   // 叶子结点
    struct {TrieNode * ptr[27];  int num; }    bh;   // 分支结点
  };
}TrieNode, * TrieTree;     // 键树类型
```

则键树查找操作可如算法 9.16 实现之。

```
Record * SearchTrie (TrieTree T, KeysType K) {
  // 在键树 T 中查找关键字等于 K 的记录。
  for ( p=T, i=0;                                    // 对 K 的每个字符逐个查找
        p && p->kind == BRANCH && i<K.num;           // * p 为分支结点
        p=p->bh.ptr[ord(K.ch[i])], ++i );            // ord① 求字符在字母表中序号
  if (p && p->kind == LEAF && p->lf.K == K ) return p->lf.infoptr;   // 查找成功
  else return NULL;                                  // 查找不成功
}// SearchTrie
```

算法 9.16

从上述查找过程可见，在查找成功时走了一条从根到叶子结点的路径。例如，在图 9.21 上，查找关键字 CHEN 的过程为：从根结点 α 出发，经 β、γ 结点，最后到达叶子结点 δ。而查找 CHAI 的过程为从根结点 α 出发，经 β、γ 结点后到 ε 结点。由于该结点中和字符'I'相应的指针为空，则查找不成功。由此，其查找的时间依赖于树的深度。我们可以对关键字集选择一种合适的分割，以缩减 Trie 树的深度。例如，根据(9-24)中关键字集的特点，可作如下分割。先按首字符不同分成多个子集之后，然后按最后一个字符不同分割每个子集，再按第二个字符……，前后交叉分割。由此得到如图 9.22 所示的 Trie 树，在该树上，除两个叶子结点在第四层上外，其余叶子结点均在第三层上。还可限制 Trie

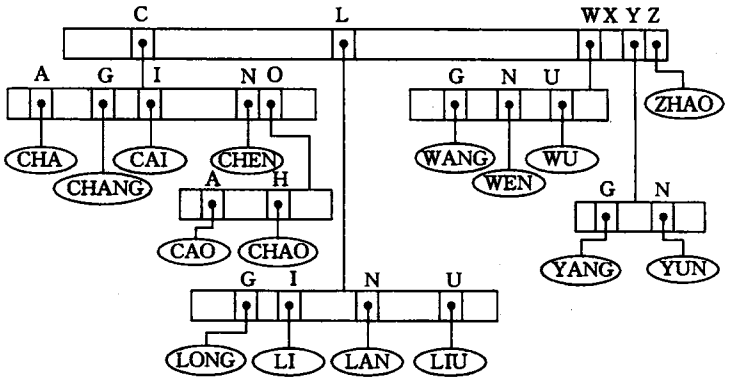


图 9.22　对(9-24)关键字集采用另一种分割法得到的 Trie 树(深度=4)

① 假设 ord 过程将 K.ch[i]字符转换成该字符在字母表中序号，并假设字符'$'的序号为零。

www.TopSage.com

大家网 TopSage.com

树的深度，假设允许 Trie 树的最大深度为 l，则所有直至 $l-1$ 层皆为同义词的关键字都进入同一叶子结点。若分割得合适，则可使每个叶子结点中只含有少数几个同义词。当然也可增加分支的个数以减少树的深度。

在 Trie 树上易于进行插入和删除，只是需要相应地增加和删除一些分支结点。当分支结点中 num 域的值减为 1 时，便可被删除。

双链树和 Trie 树是键树的两种不同的表示方法，它们有各自的特点。从其不同的存储结构特性可见，若键树中结点的度较大，则采用 Trie 树结构较双链树更为合适。

综上对树表的讨论可见，它们的查找过程都是从根结点出发，走了一条从根到叶子（或非终端结点）的路径，其查找时间依赖于树的深度。由于树表主要用作文件索引，因此结点的存取还涉及外部存储设备的特性，故在此没有对它们作平均查找长度的分析。

9.3 哈 希 表

9.3.1 什么是哈希表

在前面讨论的各种结构（线性表、树等）中，记录在结构中的相对位置是随机的，和记录的关键字之间不存在确定的关系，因此，在结构中查找记录时需进行一系列和关键字的比较。这一类查找方法建立在“比较”的基础上。在顺序查找时，比较的结果为“=”与“≠”两种可能；在折半查找、二叉排序树查找和 B-树查找时，比较的结果为“<”、“=”和“>”3 种可能。查找的效率依赖于查找过程中所进行的比较次数。

理想的情况是希望不经过任何比较，一次存取便能得到所查记录，那就必须在记录的存储位置和它的关键字之间建立一个确定的对应关系 f，使每个关键字和结构中一个惟一的存储位置相对应。因而在查找时，只要根据这个对应关系 f 找到给定值 K 的像 $f(K)$。若结构中存在关键字和 K 相等的记录，则必定在 $f(K)$ 的存储位置上，由此，不需要进行比较便可直接取得所查记录。在此，我们称这个对应关系 f 为**哈希**（Hash）函数，按这个思想建立的表为哈希表。

我们可以举一个哈希表的最简单的例子。假设要建立一张全国 34 个地区的各民族人口统计表，每个地区为一个记录，记录的各数据项为：

编号	地区名	总人口	汉族	回族	…

显然，可以用一个一维数组 C(1..30)来存放这张表，其中 C[i]是编号为 i 的地区的人口情况。编号 i 便为记录的关键字，由它惟一确定记录的存储位置 C[i]。例如：假设北京市的编号为 1，则若要查看北京市的各民族人口，只要取出 C[1]的记录即可。假如把这个数组看成是哈希表，则哈希函数 $f(key)=key$。然而，很多情况下的哈希函数并不如此简单。可仍以此为例，为了查看方便应以地区名作为关键字。假设地区名以汉语拼音的字符表示，则不能简单地取哈希函数 $f(key)=key$，而是首先要将它们转化为数字，有时还要作些简单的处理。例如我们可以有这样的哈希函数：(1)取关键字中第一个字母在字母表中的序号作为哈希函数。例如：BEIJING 的哈希函数值为字母“B”在字母表中的序号，

www.TopSage.com

等于 02;或(2)先求关键字的第一个和最后一个字母在字母表中的序号之和,然后判别这个和值,若比 30(表长)大,则减去 30。例如:TIANJIN 的首尾两个字母"T"和"N"的序号之和为 34,故取 04 为它的哈希函数值;或(3)先求每个汉字的第一个拼音字母的 ASCII 码(和英文字母相同)之和的八进制形式,然后将这个八进制数看成是十进制数再除以 30 取余数,若余数为零则加上 30 而为哈希函数值。例如:HENAN 的头两个拼音字母为"H"和"N",它们的 ASCII 码之和为 $(226)_8$,以 $(226)_{10}$ 除以 $(30)_{10}$ 得余数为 16,则 16 为 HENAN 的哈希函数值,即记录在数组中的下标值。上述人口统计表中部分关键字在这 3 种不同的哈希函数情况下的哈希函数值如表 9.1 所列:

表 9.1 简单的哈希函数示例

key	BEIJING (北京)	TIANJIN (天津)	HEBEI (河北)	SHANXI (山西)	SHANGHAI (上海)	SHANDONG (山东)	HENAN (河南)	SICHUAN (四川)
$f_1(key)$	02	20	08	19	19	19	08	19
$f_2(key)$	09	04	17	28	28	26	22	03
$f_3(key)$	04	26	02	13	23	17	16	16

从这个例子可见:

(1) 哈希函数是一个映像,因此哈希函数的设定很灵活[①],只要使得任何关键字由此所得的哈希函数值都落在表长允许范围之内即可;

(2) 对不同的关键字可能得到同一哈希地址,即 $key1 \neq key2$,而 $f(key1)=f(key2)$,这种现象称**冲突**(collision)。具有相同函数值的关键字对该哈希函数来说称做**同义词**(synonym)。例如:关键字 HEBEI 和 HENAN 不等,但 f_1(HEBEI)$=f_1$(HENAN),又如:f_2(SHANXI)$=f_2$(SHANGHAI);f_3(HENAN)$=f_3$(SICHUAN)。这种现象给建表造成困难,如在第一种哈希函数的情况下,因为山西、上海、山东和四川这 4 个记录的哈希地址均为 19,而 C[19]只能存放一个记录,那么其他 3 个记录存放在表中什么位置呢?并且,从上表 3 个不同的哈希函数的情况可以看出,哈希函数选得合适可以减少这种冲突现象。特别是在这个例子中。只可能有 30 个记录,可以仔细分析这 30 个关键字的特性,选择一个恰当的哈希函数来避免冲突的发生。

然而,在一般情况下,冲突只能尽可能地少,而不能完全避免。因为,哈希函数是从关键字集合到地址集合的映像。通常,关键字集合比较大,它的元素包括所有可能的关键字,而地址集合的元素仅为哈希表中的地址值。假设表长为 n,则地址为 0 到 $n-1$。例如,在 C 语言的编译程序中可对源程序中的标识符建立一张哈希表。在设定哈希函数时考虑的关键字集合应包含所有可能产生的关键字;假设标识符定义为以字母为首的 8 位字母或数字,则关键字(标识符)的集合大小为 $C_{52}^1 * C_{62}^7 * 7! = 1.288899 \times 10^{14}$,而在一个源程序中出现的标识符是有限的,设表长为 1000 足矣。地址集合中的元素为 0 到 999。因此,在一般情况下,哈希函数是一个压缩映像,这就不可避免产生冲突。因此,在建造哈

① Hash(哈希)的原意本是杂凑。

www.TopSage.com

大家网 TopSage.com

希表时不仅要设定一个“好”的哈希函数，而且要设定一种处理冲突的方法。

综上所述，可如下描述哈希表：根据设定的哈希函数 $H(key)$ 和处理冲突的方法将一组关键字映像到一个有限的连续的地址集（区间）上，并以关键字在地址集中的“像”作为记录在表中的存储位置，这种表便称为**哈希表**，这一映像过程称为哈希造表或**散列**，所得存储位置称**哈希地址或散列地址**。

下面分别就哈希函数和处理冲突的方法进行讨论。

9.3.2 哈希函数的构造方法

构造哈希函数的方法很多。在介绍各种方法之前，首先需要明确什么是“好”的哈希函数。

若对于关键字集合中的任一个关键字，经哈希函数映像到地址集合中任何一个地址的概率是相等的，则称此类哈希函数为**均匀的**（Uniform）哈希函数。换句话说，就是使关键字经过哈希函数得到一个“随机的地址”，以便使一组关键字的哈希地址均匀分布在整个地址区间中，从而减少冲突。

常用的构造哈希函数的方法有：

1. 直接定址法

取关键字或关键字的某个线性函数值为哈希地址。即：

$$H(key)=key \text{ 或 } H(key)=a \cdot key+b$$

其中 a 和 b 为常数（这种哈希函数叫做自身函数）。

例如：有一个从 1 岁到 100 岁的人口数字统计表，其中，年龄作为关键字，哈希函数取关键字自身。如表 9.2 所示：

表 9.2 直接定址哈希函数例之一

地址	01	02	03	…	25	26	27	…	100
年龄	1	2	3	…	25	26	27	…	…
人数	3000	2000	5000	…	1050	…	…	…	…
⋮									

这样，若要询问 25 岁的人有多少，则只要查表的第 25 项即可。

又如：有一个解放后出生的人口调查表，关键字是年份，哈希函数取关键字加一常数：$H(key)=key+(-1948)$，如表 9.3 所示。

表 9.3 直接定址哈希函数例之二

地址	01	02	03	…	22	…
年份	1949	1950	1951	…	1970	…
人数	…	…	…	…	15000	…
⋮						

www.TopSage.com

这样，若要查 1970 年出生的人数，则只要查第(1970－1948)＝22 项即可。

由于直接定址所得地址集合和关键字集合的大小相同。因此，对于不同的关键字不会发生冲突。但实际中能使用这种哈希函数的情况很少。

2. 数字分析法

假设关键字是以 r 为基的数(如：以 10 为基的十进制数)，并且哈希表中可能出现的关键字都是事先知道的，则可取关键字的若干数位组成哈希地址。

例如有 80 个记录，其关键字为 8 位十进制数。假设哈希表的表长为 100_{10}，则可取两位十进制数组成哈希地址。取哪两位？原则是使得到的哈希地址尽量避免产生冲突，则需从分析这 80 个关键字着手。假设这 80 个关键字中的一部分如下所列：

⋮

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

⋮

①	②	③	④	⑤	⑥	⑦	⑧

对关键字全体的分析中我们发现：第①②位都是“8 1”，第③位只可能取 1、2、3 或 4，第⑧位只可能取 2、5 或 7，因此这 4 位都不可取。由于中间的 4 位可看成是近乎随机的，因此可取其中任意两位，或取其中两位与另外两位的叠加求和后舍去进位作为哈希地址。

3. 平方取中法

取关键字平方后的中间几位为哈希地址。这是一种较常用的构造哈希函数的方法。通常在选定哈希函数时不一定能知道关键字的全部情况，取其中哪几位也不一定合适，而一个数平方后的中间几位数和数的每一位都相关，由此使随机分布的关键字得到的哈希地址也是随机的。取的位数由表长决定。

例如：为 BASIC 源程序中的标识符建立一个哈希表。假设 BASIC 语言中允许的标识符为一个字母，或一个字母和一个数字。在计算机内可用两位八进制数表示字母和数字，如图 9.23(a)所示。取标识符在计算机中的八进制数为它的关键字。假设表长为 $512=2^9$，则可取关键字平方后的中间 9 位二进制数为哈希地址。例如，图 9.23(b)列出了一些标识符及它们的哈希地址。

4. 折叠法

将关键字分割成位数相同的几部分(最后一部分的位数可以不同)，然后取这几部分的叠加和(舍去进位)作为哈希地址，这方法称为折叠法(folding)。关键字位数很多，而且关键字中每一位上数字分布大致均匀时，可以采用折叠法得到哈希地址。

www.TopSage.com

大家网 TopSage.com

A	B	C	…	Z	0	1	2	…	9
01	02	03		32	60	61	62		71

(a)

记 录	关 键 字	(关键字)^2	哈希地址($2^{17}\sim 2^9$)
A	0100	0 010000	010
I	1100	1 210000	210
J	1200	1 440000	440
I0	1160	1 370400	370
P1	2061	4 310541	310
P2	2062	4 314704	314
Q1	2161	4 734741	734
Q2	2162	4 741304	741
Q3	2163	4 745651	745

(b)

图 9.23

(a) 字符的八进制表示对照表； (b) 标识符及其哈希地址

例如：每一种西文图书都有一个国际标准图书编号(ISBN)，它是一个 10 位的十进制数字，若要以它作关键字建立一个哈希表，当馆藏书种类不到 10 000 时，可采用折叠法构造一个四位数的哈希函数。在折叠法中数位叠加可以有移位叠加和间界叠加两种方法。移位叠加是将分割后的每一部分的最低位对齐，然后相加；间界叠加是从一端向另一端沿分割界来回折叠，然后对齐相加。如国际标准图书编号 0-442-20586-4 的哈希地址分别如图 9.24(a)和(b)所示。

```
      5864                  5864
      4220                  0224
+)      04            +)      04
-------------         -------------
     10088                 6092
H(key)= 0088          H(key)= 6092
```

(a) (b)

图 9.24 由折叠法求得哈希地址

(a) 移位叠加； (b) 间界叠加

5. 除留余数法

取关键字被某个不大于哈希表表长 m 的数 p 除后所得余数为哈希地址。即

$$H(key) = key \quad \text{MOD} \quad p,\ p \leqslant m$$

这是一种最简单，也最常用的构造哈希函数的方法。它不仅可以对关键字直接取模(MOD)，也可在折叠、平方取中等运算之后取模。

值得注意的是，在使用除留余数法时，对 p 的选择很重要。若 p 选的不好，容易产生同义词。请看下面 3 个例子。

假设取标识符在计算机中的二进制表示为它的关键字(标识符中每个字母均用两位

www.TopSage.com

八进制数表示),然后对 $p=2^6$ 取模。这个运算在计算机中只要移位便可实现,将关键字左移直至只留下最低的 6 位二进制数。这等于将关键字的所有高位值都忽略不计。因而使得所有最后一个字符相同的标识符,如 a1,i1,temp1,cp1 等均成为同义词。

若 p 含有质因子 pf,则所有含有 pf 因子的关键字的哈希地址均为 pf 的倍数。例如,当 $p=21(=3\times7)$时,下列含因子 7 的关键字对 21 取模的哈希地址均为 7 的倍数。

关键字	28	35	63	77	105
哈希地址	7	14	0	14	0

假设有两个标识符 xy 和 yx,其中 x、y 均为字符,又假设它们的机器代码(6 位二进制数)分别为 $c(x)$和 $c(y)$,则上述两个标识符的关键字分别为

$$key1 = 2^6c(x)+c(y) \text{ 和 } key2 = 2^6c(y)+c(x)$$

假设用除留余数法求哈希地址,且 $p=tq$,t 是某个常数,q 是某个质数。则当 $q=3$ 时,这两个关键字将被散列在差为 3 的地址上。因为

$$[H(key1)-H(key2)] \text{ MOD } q$$
$$=\{[2^6c(x)+c(y)] \text{ MOD } p-[2^6c(y)+c(x)] \text{ MOD } p\} \text{ MOD } q$$
$$=\{2^6c(x) \text{ MOD } p+c(y) \text{ MOD } p-2^6c(y) \text{ MOD } p-c(x) \text{ MOD } p\} \text{ MOD } q$$
$$=\{2^6c(x) \text{ MOD } q+c(y) \text{ MOD } q-2^6c(y) \text{ MOD } q-c(x) \text{ MOD } q\} \text{ MOD } q$$

(因对任一 x 有$(x \text{ MOD } (t*q)) \text{ MOD } q=(x \text{ MOD } q) \text{ MOD } q$)

当 $q=3$ 时,上式为

$$=\{(2^6 \text{ MOD } 3)c(x) \text{ MOD } 3+c(y) \text{ MOD } 3-(2^6 \text{ MOD } 3)c(y) \text{ MOD } 3-c(x) \text{ MOD } 3\} \text{ MOD } 3$$
$$=0 \text{ MOD } 3$$

由众人的经验得知:一般情况下,可以选 p 为质数或不包含小于 20 的质因数的合数。

6. 随机数法

选择一个随机函数,取关键字的随机函数值为它的哈希地址,即 H(key)=random(key),其中 random 为随机函数。通常,当关键字长度不等时采用此法构造哈希函数较恰当。

实际工作中需视不同的情况采用不同的哈希函数。通常,考虑的因素有:

(1) 计算哈希函数所需时间(包括硬件指令的因素);

(2) 关键字的长度;

(3) 哈希表的大小;

(4) 关键字的分布情况;

(5) 记录的查找频率。

9.3.3 处理冲突的方法

在"9.3.1 什么是哈希表"中曾提及均匀的哈希函数可以减少冲突,但不能避免,因

www.TopSage.com

大家网 TopSage.com

此，如何处理冲突是哈希造表不可缺少的另一方面。

假设哈希表的地址集为 0～($n-1$)，冲突是指由关键字得到的哈希地址为 j($0 \leqslant j \leqslant n-1$)的位置上已存有记录，则“处理冲突”就是为该关键字的记录找到另一个“空”的哈希地址。在处理冲突的过程中可能得到一个地址序列 H_i　$i=1,2,\cdots,k$，($H_i \in [0,n-1]$)。即在处理哈希地址的冲突时，若得到的另一个哈希地址 H_1 仍然发生冲突，则再求下一个地址 H_2，若 H_2 仍然冲突，再求得 H_3。依次类推，直至 H_k 不发生冲突为止，则 H_k 为记录在表中的地址。

通常用的处理冲突的方法有下列几种：

1. 开放定址法

$$H_i = (H(key) + d_i) \text{ MOD } m \quad i = 1,2,\cdots,k \ (k \leqslant m-1) \tag{9-25}$$

其中：$H(key)$ 为哈希函数；m 为哈希表表长；d_i 为增量序列，可有下列 3 种取法：(1) $d_i=1,2,3,\cdots,m-1$，称线性探测再散列；(2) $d_i=1^2,-1^2,2^2,-2^2,3^2,\cdots,\pm k^2$，($k \leqslant m/2$)称二次探测再散列；(3) $d_i=$伪随机数序列，称伪随机探测再散列。

例如，在长度为 11 的哈希表中已填有关键字分别为 17，60，29 的记录（哈希函数 $H(key)=key$ MOD 11），现有第四个记录，其关键字为 38，由哈希函数得到哈希地址为 5，产生冲突。若用线性探测再散列的方法处理时，得到下一个地址 6，仍冲突；再求下一个地址 7，仍冲突；直到哈希地址为 8 的位置为“空”时止，处理冲突的过程结束，记录填入哈希表中序号为 8 的位置。若用二次探测再散列，则应该填入序号为 4 的位置。类似地可得到伪随机再散列的地址(参见图 9.25)。

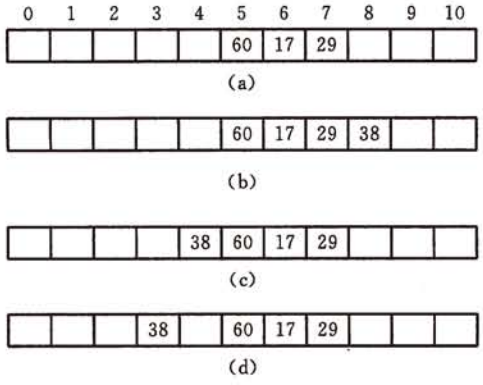


图 9.25　用开放定址处理冲突时，关键字为 38 的记录插入前后的哈希表

(a) 插入前；(b) 线性探测再散列；(c) 二次探测再散列；(d) 伪随机探测再散列，伪随机数列 9，…

从上述线性探测再散列的过程中可以看到一个现象：当表中 $i,i+1,i+2$ 位置上已填有记录时，下一个哈希地址为 i、$i+1$、$i+2$ 和 $i+3$ 的记录都将填入 $i+3$ 的位置，这种在处理冲突过程中发生的两个第一个哈希地址不同的记录争夺同一个后继哈希地址的现象称做“二次聚集”，即在处理同义词的冲突过程中又添加了非同义词的冲突，显然，这种现象对查找不利。但另一方面，用线性探测再散列处理冲突可以保证做到：只要哈希表未填满，总能找到一个不发生冲突的地址 H_k，而二次探测再散列只有在哈希表长 m 为形如

www.TopSage.com

$4j+3$(j 为整数)的素数时才可能[1]，随机探测再散列，则取决于伪随机数列。

2. 再哈希法

$$H_i = RH_i(key) \qquad i = 1,2,\cdots,k \tag{9-26}$$

RH_i 均是不同的哈希函数，即在同义词产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生。这种方法不易产生“聚集”，但增加了计算的时间。

3. 链地址法

将所有关键字为同义词的记录存储在同一线性链表中。假设某哈希函数产生的哈希地址在区间[0,$m-1$]上，则设立一个指针型向量

```
Chain ChainHash[m];
```

其每个分量的初始状态都是空指针。凡哈希地址为 i 的记录都插入到头指针为 ChainHash[i]的链表中。在链表中的插入位置可以在表头或表尾；也可以在中间，以保持同义词在同一线性链表中按关键字有序。

例 9-3 已知一组关键字为 (19,14,23,01,68,20,84,27,55,11,10,79)

则按哈希函数 $H(key)=key$ MOD 13 和链地址法处理冲突构造所得的哈希表如图 9.26 所示。

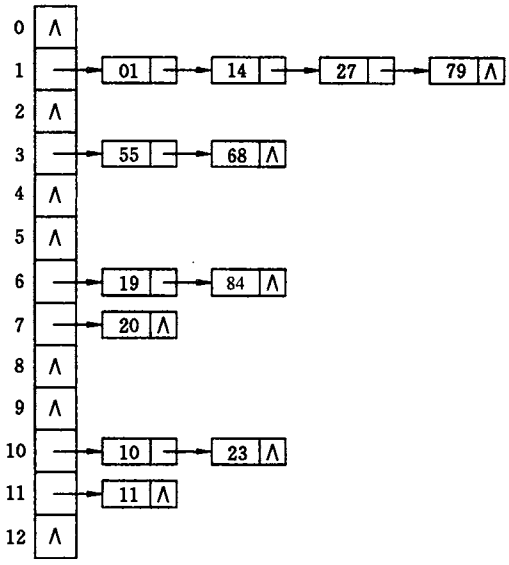


图 9.26 链地址法处理冲突时的哈希表

(同一链表中关键字自小至大有序)

4. 建立一个公共溢出区

这也是处理冲突的一种方法。假设哈希函数的值域为[0,$m-1$]，则设向量 HashTable[0..$m-1$]为基本表，每个分量存放一个记录，另设立向量 OverTable[0..v]为溢出表。所有关键字和基本表中关键字为同义词的记录，不管它们由哈希函数得到的哈希地址是什么，一旦发生冲突，都填入溢出表。

www.TopSage.com

大家网
TopSage.com

9.3.4 哈希表的查找及其分析

在哈希表上进行查找的过程和哈希造表的过程基本一致。给定 K 值，根据造表时设定的哈希函数求得哈希地址，若表中此位置上没有记录，则查找不成功；否则比较关键字，若和给定值相等，则查找成功；否则根据造表时设定的处理冲突的方法找“下一地址”，直至哈希表中某个位置为“空”或者表中所填记录的关键字等于给定值时为止。

算法 9.17 为以开放定址等方法(除链地址法外)处理冲突的哈希表的查找过程。

```
// - - - 开放定址哈希表的存储结构 - - -
int  hashsize[] = { 997, ... };        // 哈希表容量递增表，一个合适的素数序列
typedef struct {
   ElemType   * elem;                  // 数据元素存储基址，动态分配数组
   int        count;                   // 当前数据元素个数
   int        sizeindex;               // hashsize[sizeindex]为当前容量
}HashTable;

#define  SUCCESS  1
#define  UNSUCCESS  0
#define  DUPLICATE  - 1

Status SearchHash (HashTable H, KeyType K, int &p, int &c) {
   // 在开放定址哈希表 H 中查找关键码为 K 的元素，若查找成功，以 p 指示待查数据
   // 元素在表中位置，并返回 SUCCESS；否则，以 p 指示插入位置，并返回UNSUCCESS，
   // c 用以计冲突次数，其初值置零，供建表插入时参考
   p = Hash(K);                              // 求得哈希地址
   while( H.elem[p].key != NULLKEY &&        // 该位置中填有记录
        !EQ(K, H.elem[p].key) )              // 并且关键字不相等
      collision(p, ++c);                     // 求得下一探查地址 p
   if EQ(K, H.elem[p].key)
      return SUCCESS;          // 查找成功，p 返回待查数据元素位置
   else return UNSUCCESS;      // 查找不成功(H.elem[p].key == NULLKEY)，
                               // p 返回的是插入位置
}// SearchHash
```

算法 9.17

算法 9.18 通过调用查找算法(算法 9.17)实现了开放定址哈希表的插入操作。

```
Status InsertHash (HashTable &H, Elemtype e) {
  // 查找不成功时插入数据元素 e 到开放定址哈希表 H 中，并返回 OK；若冲突次数
  // 过大，则重建哈希表
  c = 0;
  if( SearchHash ( H, e.key, p, c ) )
    return DUPLICATE;                                  // 表中已有与 e 有相同关键字的元素
  else if ( c < hashsize[H.sizeindex]/2 ) {            // 冲突次数 c 未达到上限，(c 的阀值可调)
      H.elem[p] = e;   ++H.count;   return OK;         // 插入 e
     }
  else {RecreateHashTable(H);return UNSUCCESS;}        // 重建哈希表
}// InsertHash
```

算法 9.18

www.TopSage.com

例 9-4 已知例 9-3 中所示的一组关键字按哈希函数 $H(key)=key$ MOD 13 和线性探测处理冲突构造所得哈希表 a. elem[0..15]如图 9.27 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	14	01	68	27	55	19	20	84	79	23	11	10			

图 9.27 哈希表 a. elem[0：15]

(其哈希函数为 $H(key)=key$ MOD 13，处理冲突为线性探测再散列)

给定值 $K=84$ 的查找过程为：首先求得哈希地址 $H(84)=6$，因 a. elem[6]不空且 a. elem[6]. key≠84，则找第一次冲突处理后的地址 $H_1=(6+1)$ MOD $16=7$，而 a. elem[7]不空且 a. elem[7]. key≠84，则找第二次冲突处理后的地址 $H_2=(6+2)$ MOD $16=8$，a. elem[8]不空且 a. elem[8]. key=84，则查找成功，返回记录在表中序号 8。

给定值 $K=38$ 的查找过程为：先求哈希地址 $H(38)=12$，a. elem[12]不空且 a. elem[12]. key≠38，则找下一地址 $H_1=(12+1)$ MOD $16=13$，由于 a. elem[13]是空记录，则表明表中不存在关键字等于 38 的记录。

从哈希表的查找过程可见：

(1) 虽然哈希表在关键字与记录的存储位置之间建立了直接映像，但由于“冲突”的产生，使得哈希表的查找过程仍然是一个给定值和关键字进行比较的过程。因此，仍需以平均查找长度作为衡量哈希表的查找效率的量度。

(2) 查找过程中需和给定值进行比较的关键字的个数取决于下列三个因素：哈希函数，处理冲突的方法和哈希表的装填因子。

哈希函数的“好坏”首先影响出现冲突的频繁程度。但是，对于“均匀的”哈希函数可以假定：不同的哈希函数对同一组随机的关键字，产生冲突的可能性相同，因为一般情况下设定的哈希函数是均匀的，则可不考虑它对平均查找长度的影响。

对同样一组关键字，设定相同的哈希函数，则不同的处理冲突的方法得到的哈希表不同，它们的平均查找长度也不同。如例 9-3 和例 9-4 中的两个哈希表，在记录的查找概率相等的前提下，前者(链地址法)的平均查找长度为

$$ASL(12)=\frac{1}{12}(1\cdot 6+2\cdot 4+3+4)=1.75$$

后者(线性探测再散列)的平均查找长度为

$$ASL(12)=\frac{1}{12}(1\cdot 6+2+3\cdot 3+4+9)=2.5$$

容易看出，线性探测再散列在处理冲突的过程中易产生记录的二次聚集，如既使得哈希地址不相同的记录又产生新的冲突；而链地址法处理冲突不会发生类似情况，因为哈希地址不同的记录在不同的链表中。

在一般情况下，处理冲突方法相同的哈希表，其平均查找长度依赖于哈希表的装填因子。

哈希表的装填因子定义为

$$\alpha=\frac{\text{表中填入的记录数}}{\text{哈希表的长度}}$$

www.TopSage.com

大家网 TopSage.com

α 标志哈希表的装满程度。直观地看，α 越小，发生冲突的可能性就越小；反之，α 越大，表中已填入的记录越多，再填记录时，发生冲突的可能性就越大，则查找时，给定值需与之进行比较的关键字的个数也就越多。

可以证明：[1][2]

线性探测再散列的哈希表查找成功时的平均查找长度为

$$S_{nl} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \tag{9-27}$$

随机探测再散列、二次探测再散列和再哈希的哈希表查找成功时的平均查找长度为

$$S_{nr} \approx -\frac{1}{\alpha}\ln(1-\alpha) \tag{9-28}$$

链地址法处理冲突的哈希表查找成功时的平均查找长度为

$$S_{nc} \approx 1+\frac{\alpha}{2} \tag{9-29}$$

由于哈希表在查找不成功时所用比较次数也和给定值有关，则可类似地定义哈希表在查找不成功时的平均查找长度为：查找不成功时需和给定值进行比较的关键字个数的期望值。同样可证明，不同的处理冲突方法构成的哈希表查找不成功时的平均查找长度分别为

$$U_{nl} \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) \tag{9-30}$$

—— 线性探测再散列

$$U_{nr} \approx \frac{1}{1-\alpha} \tag{9-31}$$

—— 伪随机探测再散列等

$$U_{nc} \approx \alpha + e^{-\alpha} \tag{9-32}$$

—— 链地址

下面仅以随机探测的一组公式为例进行分析推导。

先分析长度为 m 的哈希表中装填有 n 个记录时查找不成功的平均查找长度。这个问题相当于要求在这张表中填入第 $n+1$ 个记录时所需作的比较次数的期望值。

假定：(1)哈希函数是均匀的。即产生表中各个地址的概率相等；(2)处理冲突后产生的地址也是随机的。

若设 p_i 表示前 i 个哈希地址均发生冲突的概率；q_i 表示需进行 i 次比较才找到一个“空位”的哈希地址(即前 $i-1$ 次发生冲突，第 i 次不冲突)的概率。则有：

$$p_1 = \frac{n}{m} \qquad q_1 = 1-\frac{n}{m}$$

$$p_2 = \frac{n}{m}\cdot\frac{n-1}{m-1} \qquad q_2 = \frac{n}{m}\cdot\left(1-\frac{n-1}{m-1}\right)$$

$$\vdots \qquad\qquad \vdots$$

$$p_i = \frac{n}{m}\cdot\frac{n-1}{m-1}\cdots\frac{n-i+1}{m-i+1} \qquad q_i = \frac{n}{m}\cdot\frac{n-1}{m-1}\cdots\frac{n-i+2}{m-i+2}\left(1-\frac{n-i+1}{m-i+1}\right)$$

www.TopSage.com

$$\vdots \qquad\qquad \vdots$$

$$p_n=\frac{n}{m}\cdot\frac{n-1}{m-1}\cdots\frac{1}{m-n+1} \qquad q_n=\frac{n}{m}\cdots\frac{2}{m-n+2}\left(1-\frac{1}{m-n+1}\right)$$

$$p_{n+1}=0 \qquad q_{n+1}=\frac{n}{m}\cdots\frac{1}{m-n+1}$$

可见，在 p_i 和 q_i 之间，存在关系式

$$q_i = p_{i-1} - p_i \tag{9-33}$$

由此，当长度为 m 的哈希表中已填有 n 个记录时，查找不成功的平均查找长度为

$$\begin{aligned}U_n &= \sum_{i=1}^{n+1} q_i C_i = \sum_{i=1}^{n+1}(p_{i-1}-p_i)i\\ &=1+p_1+p_2+\cdots+p_n-(n+1)p_{n+1}\\ &=\frac{1}{1-\frac{n}{m+1}} \quad \text{（用归纳法证明）}\\ &\approx\frac{1}{1-\alpha}\end{aligned}$$

由于哈希表中 n 个记录是先后填入的，查找每一个记录所需比较次数的期望值，恰为填入此记录时找到此哈希地址时所进行的比较次数的期望值。因此，对表长为 m、记录数为 n 的哈希表，查找成功时的平均查找长度为

$$S_n = \sum_{i=1}^{n-1} p_i C_i = \sum_{i=0}^{n-1} p_i U_i$$

设对 n 个记录的查找概率相等。即 $p_i=\frac{1}{n}$ 则

$$\begin{aligned}S_n &=\frac{1}{n}\sum_{i=0}^{n-1}U_i=\frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-\frac{i}{m}}\\ &\approx\frac{m}{n}\int_0^{\alpha}\frac{\mathrm{d}x}{1-x}\approx-\frac{1}{\alpha}\ln(1-\alpha)\end{aligned}$$

从以上分析可见，哈希表的平均查找长度是 α 的函数，而不是 n 的函数。由此，不管 n 多大，我们总可以选择一个合适的装填因子以便将平均查找长度限定在一个范围内。

值得提醒的是，若要在非链地址处理冲突的哈希表中删除一个记录，则需在该记录的位置上填入一个特殊的符号，以免找不到在它之后填入的“同义词”的记录。

最后要说明的是，对于预先知道且规模不大的关键字集，有时也可以找到不发生冲突的哈希函数，因此，对频繁进行查找的关键字集，还是应尽力设计一个完美的(perfect)的哈希函数。例如，对 PASCAL 语言中的 26 个保留字可设定下述无冲突的哈希函数

$$\mathrm{H(key)} = \mathrm{L} + \mathrm{g(key[1])} + \mathrm{g(key[L])} \tag{9-34}$$

其中 L 为保留字长度，key[1]为第一个字符，key[L]为最后一个字符，g(x)为从字符到数字的转换函数，例如 g(F)=15，g(N)=13，H(FUNCTION)=8+15+13=36。所得哈希表长度为 37(请参见参考书目[14]327 页～328 页)。

www.TopSage.com

大家网
TopSage.com

第10章 内部排序

10.1 概 述

排序(Sorting)是计算机程序设计中的一种重要操作,它的功能是将一个数据元素(或记录)的任意序列,重新排列成一个按关键字有序的序列。

从第9章的讨论中容易看出,为了查找方便,通常希望计算机中的表是按关键字有序的。因为有序的顺序表可以采用查找效率较高的折半查找法,其平均查找长度为$\log_2(n+1)-1$,而无序的顺序表只能进行顺序查找,其平均查找长度为$(n+1)/2$。又如建造树表(无论是二叉排序树或B-树)的过程本身就是一个排序的过程。因此,学习和研究各种排序方法是计算机工作者的重要课题之一。

为了便于讨论,在此首先要对排序下一个确切的定义:

假设含n个记录的序列为

$$\{R_1,R_2,\cdots,R_n\} \tag{10-1}$$

其相应的关键字序列为

$$\{K_1,K_2,\cdots,K_n\}$$

需确定$1,2,\cdots,n$的一种排列$p_1,p_2,\cdots,p_n$,使其相应的关键字满足如下的非递减(或非递增[①])关系

$$K_{p_1}\leqslant K_{p_2}\leqslant\cdots\leqslant K_{p_n} \tag{10-2}$$

即使式(10-1)的序列成为一个按关键字有序的序列

$$\{R_{p_1},R_{p_2},\cdots,R_{p_n}\} \tag{10-3}$$

这样一种操作称为排序。

上述排序定义中的关键字K_i可以是记录$R_i(i=1,2,\cdots,n)$的主关键字,也可以是记录R_i的次关键字,甚至是若干数据项的组合。若K_i是主关键字,则任何一个记录的无序序列经排序后得到的结果是惟一的;若K_i是次关键字,则排序的结果不惟一,因为待排序的记录序列中可能存在两个或两个以上关键字相等的记录。假设$K_i=K_j(1\leqslant i\leqslant n,1\leqslant j\leqslant n,i\neq j)$,且在排序前的序列中$R_i$领先于$R_j$(即$i<j$)。若在排序后的序列中$R_i$仍领先于$R_j$,则称所用的**排序方法是稳定的**;反之,若可能使排序后的序列中R_j领先于R_i,则称所用的**排序方法是不稳定的**。[②]

由于待排序的记录数量不同,使得排序过程中涉及的存储器不同,可将排序方法分为两大类:一类是**内部排序**,指的是待排序记录存放在计算机随机存储器中进行的排序过程;另一类是**外部排序**,指的是待排序记录的数量很大,以致内存一次不能容纳全部记录,

① 若将式(10-2)中的"≤"号改为"≥"号,则满足非递增关系。

② 对不稳定的排序方法,只要举出一组关键字的实例说明它的不稳定性即可。

www.TopSage.com

在排序过程中尚需对外存进行访问的排序过程。本章先集中讨论内部排序，将在下一章中讨论外部排序。

内部排序的方法很多，但就其全面性能而言，很难提出一种被认为是最好的方法，每一种方法都有各自的优缺点，适合在不同的环境（如记录的初始排列状态等）下使用。如果按排序过程中依据的不同原则对内部排序方法进行分类，则大致可分为插入排序、交换排序、选择排序、归并排序和计数排序等五类；如果按内部排序过程中所需的工作量来区分，则可分为3类：(1)简单的排序方法，其时间复杂度为 $O(n^2)$；(2)先进的排序方法，其时间复杂度为 $O(n\log n)$；(3)基数排序，其时间复杂度为 $O(d \cdot n)$。本章仅就每一类介绍一两个典型算法，有兴趣了解更多算法的读者可阅读 D. E. 克努特著《计算机程序设计技巧》[2]（第三卷，排序和查找）。读者在学习本章内容时应注意，除了掌握算法本身以外，更重要的是了解该算法在进行排序时所依据的原则，以利于学习和创造更加新的算法。

通常，在排序的过程中需进行下列两种基本操作：(1)比较两个关键字的大小；(2)将记录从一个位置移动至另一个位置。前一个操作对大多数排序方法来说都是必要的，而后一个操作可以通过改变记录的存储方式来予以避免。待排序的记录序列可有下列3种存储方式：(1)待排序的一组记录存放在地址连续的一组存储单元上。它类似于线性表的顺序存储结构，在序列中相邻的两个记录 R_j 和 R_{j+1}（$j=1,2,\cdots,n-1$），它们的存储位置也相邻。在这种存储方式中，记录之间的次序关系由其存储位置决定，则实现排序必须借助移动记录；(2)一组待排序记录存放在静态链表①中，记录之间的次序关系由指针指示，则实现排序不需要移动记录，仅需修改指针即可；(3)待排序记录本身存储在一组地址连续的存储单元内，同时另设一个指示各个记录存储位置的地址向量，在排序过程中不移动记录本身，而移动地址向量中这些记录的“地址”，在排序结束之后再按照地址向量中的值调整记录的存储位置。在第二种存储方式下实现的排序又称（链）表排序，在第三种存储方式下实现的排序又称地址排序。在本章的讨论中，设待排序的一组记录以上述第一种方式存储，且为了讨论方便起见，设记录的关键字均为整数。即在以后讨论的大部分算法中，待排记录的数据类型设为：

```
#define  MAXSIZE  20          // 一个用作示例的小顺序表的最大长度
typedef  int  KeyType;        // 定义关键字类型为整数类型
typedef struct {
   KeyType  key;              // 关键字项
   InfoType  otherinfo;       // 其他数据项
}RedType;                     // 记录类型
typedef struct {
   RedType  r[MAXSIZE+1];     // r[0]闲置或用作哨兵单元
   int      length;           // 顺序表长度
}SqList;                      // 顺序表类型
```

① 因为在排序过程中，只是改变记录之间的次序关系，而不进行插入、删除操作，且在排序结束时尚需调整记录，故采用静态链表。

www.TopSage.com

大家网
TopSage.com

10.2 插入排序

10.2.1 直接插入排序

直接插入排序(Straight Insertion Sort)是一种最简单的排序方法,它的基本操作是将一个记录插入到已排好序的有序表中,从而得到一个新的、记录数增 1 的有序表。

例如,已知待排序的一组记录的初始排列如下所示:①

$$R(49),R(38),R(65),R(97),R(76),R(13),R(27),R(\overline{49}),\cdots \qquad (10\text{-}4)$$

假设在排序过程中,前 4 个记录已按关键字递增的次序重新排列,构成一个含 4 个记录的有序序列

$$\{R(38),R(49),R(65),R(97)\} \qquad (10\text{-}5)$$

现要将式(10-4)中第 5 个(即关键字为 76 的)记录插入上述序列,以得到一个新的含 5 个记录的有序序列,则首先要在式(10-5)的序列中进行查找以确定 $R(76)$所应插入的位置,然后进行插入。假设从 $R(97)$起向左进行顺序查找,由于 $65<76<97$,则 $R(76)$应插入在 $R(65)$和 $R(97)$之间,从而得到下列新的有序序列

$$\{R(38),R(49),R(65),R(76),R(97)\} \qquad (10\text{-}6)$$

称从式(10-5)到式(10-6)的过程为一趟直接插入排序。一般情况下,第 i 趟直接插入排序的操作为:在含有 $i-1$ 个记录的有序子序列 r[1..i−1]②中插入一个记录 r[i]后,变成含有 i 个记录的有序子序列 r[1..i];并且,和顺序查找类似,为了在查找插入位置的过程中避免数组下标出界,在 r[0]处设置监视哨。在自 $i-1$ 起往前搜索的过程中,可以同时后移记录。整个排序过程为进行 $n-1$ 趟插入,即:先将序列中的第 1 个记录看成是一个有序的子序列,然后从第 2 个记录起逐个进行插入,直至整个序列变成按关键字非递减有序序列为止。其算法如算法 10.1 所示:

```
void InsertSort ( SqList &L) {
  // 对顺序表 L 作直接插入排序。
  for ( i = 2; i<= L.length; ++ i )
     if (LT(L.r[i].key, L.r[i - 1].key)) {        // "<",需将 L.r[i]插入有序子表
       L.r[0] = L.r[i];                           // 复制为哨兵
       L.r[i] = L.r[i - 1];
       for ( j = i - 2;  LT(L.r[0].key, L.r[j].key);  --j )
           L.r[j + 1] = L.r[j];                   // 记录后移
       L.r[j + 1] = L.r[0];                       // 插入到正确位置
     }
}// InsertSort
```

算法 10.1

以式(10-4)中关键字为例,按照算法 10.1 进行直接插入排序的过程如图 10.1 所示。③

---

① R(x)表示关键字为 x 的记录,以下同。

② r[1..i−1]表示参与排序的顺序表中下标从 1 到 $i-1$ 的记录序列,以后同。

③ 为简便起见,图中省略记录 R 的符号,而只列其关键字。

www.TopSage.com

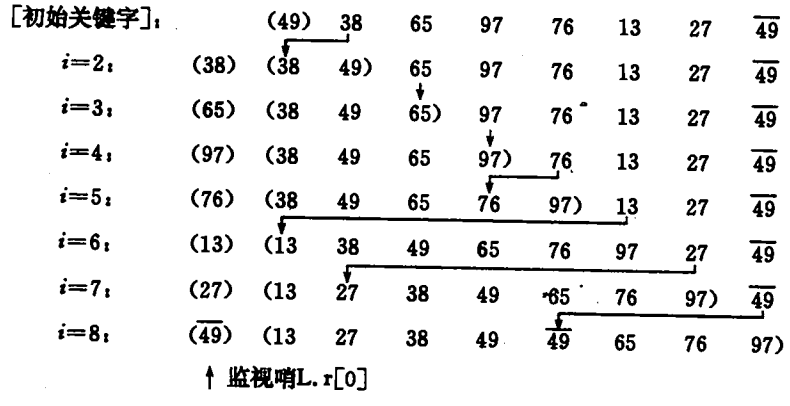


图 10.1 直接插入排序示例

从上面的叙述可见，直接插入排序的算法简洁，容易实现，那么它的效率如何呢？

从空间来看，它只需要一个记录的辅助空间，从时间来看，排序的基本操作为：比较两个关键字的大小和移动记录。先分析一趟插入排序的情况。算法 10.1 中里层的 for 循环的次数取决于待插记录的关键字与前 $i-1$ 个记录的关键字之间的关系。若 L.r[i].key<L.r[1].key，则内循环中，待插记录的关键字需与有序子序列 L.r[1..i−1]中 $i-1$ 个记录的关键字和监视哨中的关键字进行比较，并将 L.r[1..i−1]中 $i-1$ 个记录后移。则在整个排序过程(进行 $n-1$ 趟插入排序)中，当待排序列中记录按关键字非递减有序排列(以下称之为“正序”)时，所需进行关键字间比较的次数达最小值 $n-1\left(\text{即}\sum_{i=2}^{n}1\right)$，记录不需移动；反之，当待排序列中记录按关键字非递增有序排列(以下称之为“逆序”)时，总的比较次数达最大值 $(n+2)(n-1)/2\left(\text{即}\sum_{i=2}^{n}i\right)$，记录移动的次数也达最大值 $(n+4)(n-1)/2\left(\text{即}\sum_{i=2}^{n}(i+1)\right)$。若待排序记录是随机的，即待排序列中的记录可能出现的各种排列的概率相同，则我们可取上述最小值和最大值的平均值，作为直接插入排序时所需进行关键字间的比较次数和移动记录的次数，约为 $n^2/4$。由此，直接插入排序的时间复杂度为 $O(n^2)$。

10.2.2 其他插入排序

从上一节的讨论中可见，直接插入排序算法简便，且容易实现。当待排序记录的数量 n 很小时，这是一种很好的排序方法。但是，通常待排序序列中的记录数量 n 很大，则不宜采用直接插入排序。由此需要讨论改进的办法。在直接插入排序的基础上，从减少“比较”和“移动”这两种操作的次数着眼，可得下列各种插入排序的方法。

1. 折半插入排序

由于插入排序的基本操作是在一个有序表中进行查找和插入，则从 9.1 节的讨论中可知，这个“查找”操作可利用“折半查找”来实现，由此进行的插入排序称之为折半插入排

www.TopSage.com

大家网
TopSage.com

序(Binary Insertion Sort),其算法如算法 10.2 所示。

```
void BInsertSort (SqList &L) {
  // 对顺序表 L 作折半插入排序。
  for ( i = 2; i<= L.length; ++ i ) {
     L.r[0] = L.r[i];                          // 将 L.r[i]暂存到 L.r[0]
     low = 1;   high = i - 1;
     while (low<= high) {                      // 在 r[low..high]中折半查找有序插入的位置
        m = (low + high)/2;                                    // 折半
        if (LT(L.r[0].key, L.r[m].key))  high = m - 1;         // 插入点在低半区
        else  low = m + 1;                                     // 插入点在高半区
     }// while
     for ( j = i - 1; j>= high + 1; -- j ) L.r[j + 1] = L.r[j];    // 记录后移
     L.r[high + 1] = L.r[0];                                   // 插入
  }// for
}// BInsertSort
```

算法 10.2

从算法 10.2 容易看出,折半插入排序所需附加存储空间和直接插入排序相同,从时间上比较,折半插入排序仅减少了关键字间的比较次数,而记录的移动次数不变。因此,折半插入排序的时间复杂度仍为 $O(n^2)$。

2. 2-路插入排序

2-路插入排序是在折半插入排序的基础上再改进之,其目的是减少排序过程中移动记录的次数,但为此需要 n 个记录的辅助空间。具体做法是:另设一个和 L.r 同类型的数组 d,首先将 L.r[1]赋值给 d[1],并将 d[1]看成是在排好序的序列中处于中间位置的记录,然后从 L.r 中第 2 个记录起依次插入到 d[1]之前或之后的有序序列中。先将待插记录的关键字和 d[1]的关键字进行比较,若 L.r[i].key<d[1].key,则将 L.r[i]插入到 d[1]之前的有序表中。反之,则将 L.r[i]插入到 d[1]之后的有序表中。在实现算法时,可将 d 看成是一个循环向量,并设两个指针 first 和 final 分别指示排序过程中得到的有序序列中的第一个记录和最后一个记录在 d 中的位置。具体算法留作习题由读者自己写出。

仍以式(10-4)中的关键字为例,进行 2-路插入排序的过程如图 10.2 所示。

在 2-路插入排序中,移动记录的次数约为 $n^2/8$。因此,2-路插入排序只能减少移动记录的次数,而不能绝对避免移动记录。并且,当 L.r[1]是待排序记录中关键字最小或最大的记录时,2-路插入排序就完全失去它的优越性。因此,若希望在排序过程中不移动记录,只有改变存储结构,进行表插入排序。

3. 表插入排序

```
#define SIZE  100                  // 静态链表容量
typedef struct {
  RcdType   rc;                    // 记录项
  int       next;                  // 指针项
}SLNode;                           // 表结点类型
typedef struct {
```


www.TopSage.com

```
  SLNode  r[SIZE];          // 0 号单元为表头结点
  int   length;             // 链表当前长度
}SLinkListType;             // 静态链表类型
```

[初始关键字]:　49　38　65　97　76　13　27　$\overline{49}$

排序过程中 d 的状态如下:

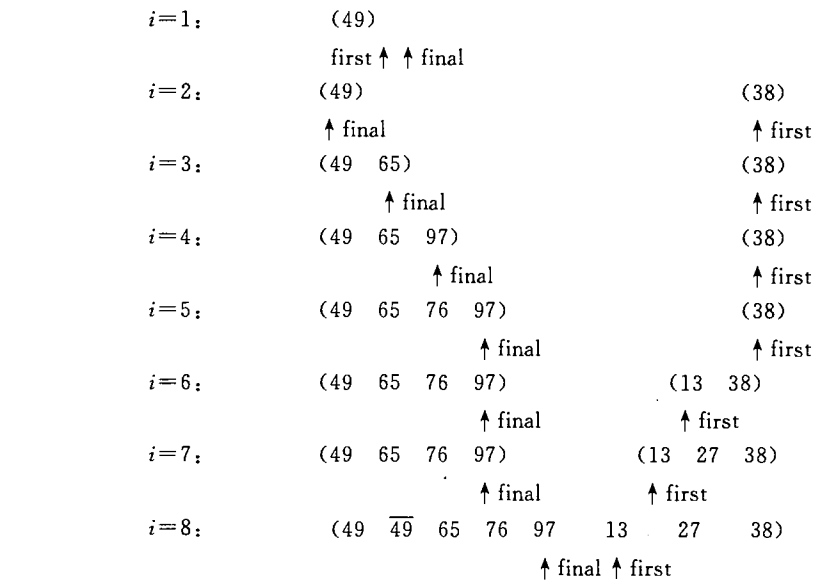


图 10.2　2-路插入排序示例

假设以上述说明的静态链表类型作为待排记录序列的存储结构,并且,为了插入方便起见,设数组中下标为"0"的分量为表头结点,并令表头结点记录的关键字取最大整数 MAXINT。则表插入排序的过程描述如下:首先将静态链表中数组下标为"1"的分量(结点)和表头结点构成一个循环链表,然后依次将下标为"2"至"n"的分量(结点)按记录关键字非递减有序插入到循环链表中。仍以式(10-4)中的关键字为例,表插入排序的过程如图 10.3 所示(图中省略记录的其他数据项)。

从表插入排序的过程可见,表插入排序的基本操作仍是将一个记录插入到已排好序的有序表中。和直接插入排序相比,不同之处仅是以修改 $2n$ 次指针值代替移动记录,排序过程中所需进行的关键字间的比较次数相同。因此,表插入排序的时间复杂度仍是 $O(n^2)$。

另一方面,表插入排序的结果只是求得一个有序链表,则只能对它进行顺序查找,不能进行随机查找,为了能实现有序表的折半查找,尚需对记录进行重新排列。

重排记录的做法是:顺序扫描有序链表,将链表中第 i 个结点移动至数组的第 i 个分量中。例如,图 10.4(a)是经表插入排序后得到的有序链表 SL。根据头结点中指针域的指示,链表的第一个结点,即关键字最小的结点是数组中下标为 6 的分量,其中记录应移至数组的第一个分量中,则将 SL.r[1]和 SL.r[6]互换,并且为了不中断静态链表中的"链",即在继续顺链扫描时仍能找到互换之前在 SL.r[1]中的结点,令互换之后的 SL.r[1]中指针域的值改为"6"(见图 10.4(b))。推广至一般情况,若第 i 个最小关键

www.TopSage.com

大家网 TopSage.com

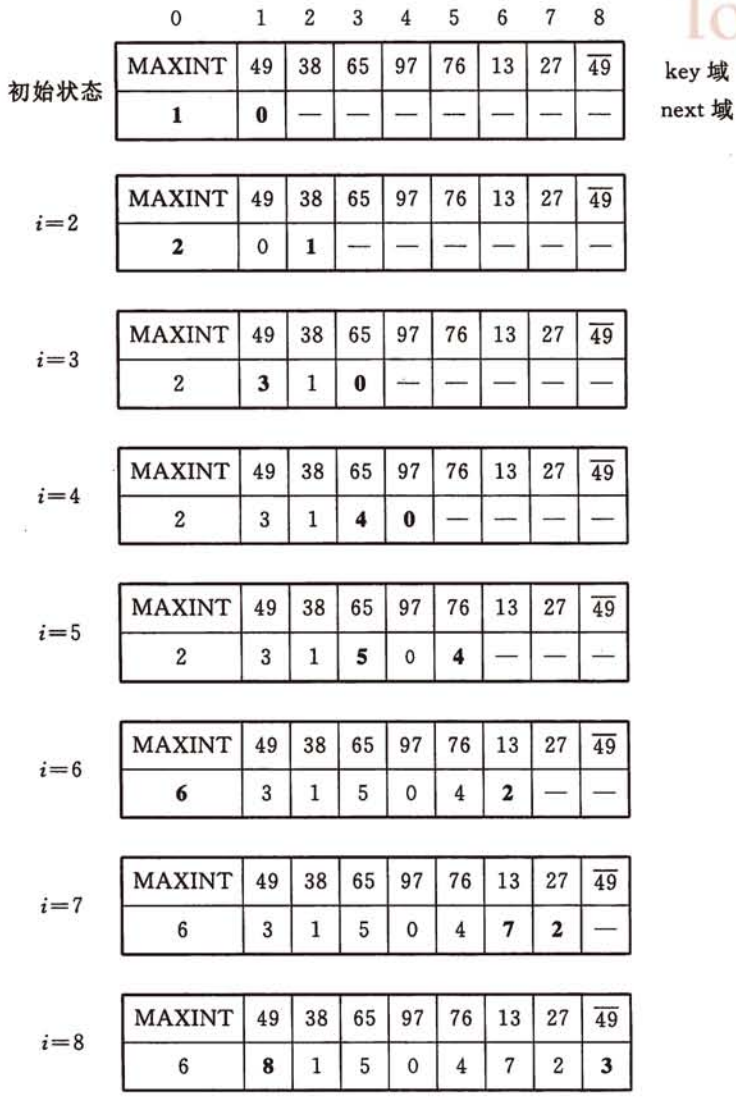


图 10.3　表插入排序示例

字的结点是数组中下标为 p 且 $p>i$ 的分量，则互换 SL.r[i]和 SL.r[p]，且令 SL.r[i]中指针域的值改为 p；由于此时数组中所有小于 i 的分量中已是“到位”的记录，则当 $p<i$ 时，应顺链继续查找直到 $p\geqslant i$ 为止。图 10.4 所示为重排记录的全部过程。

算法 10.3 描述了上述重排记录的过程。容易看出，在重排记录的过程中，最坏情况是每个记录到位都必须进行一次记录的交换，即 3 次移动记录，所以重排记录至多需进行 $3(n-1)$次记录的移动，它并不增加表插入排序的时间复杂度。

```
void Arrange ( SLinkListType &SL ) {
  // 根据静态链表 SL 中各结点的指针值调整记录位置，使得 SL 中记录按关键字非递减
  // 有序顺序排列
  p = SL.r[0].next;                    // p 指示第一个记录的当前位置
```


www.TopSage.com

```
  for ( i = 1; i<SL.length; ++ i ) {          // SL.r[1..i-1]中记录已按关键字有序排列,
                                              // 第 i 个记录在 SL 中的当前位置应不小于 i
    while (p<i)  p = SL.r[p].next;            // 找到第 i 个记录,并用 p 指示其在 SL 中当前位置
    q = SL.r[p].next;                         // q 指示尚未调整的表尾
    if ( p!= i ) {
      SL.r[p]←→SL.r[i];                       // 交换记录,使第 i 个记录到位
      SL.r[i].next = p;                       // 指向被移走的记录,使得以后可由 while 循环找回
    }
    p = q;                                    // p 指示尚未调整的表尾,为找第 i+1 个记录作准备
  }
}// Arrange
```

算法 10.3

初始状态 (a)

	0	1	2	3	4	5	6	7	8
maxint		49	38	65	97	76	13	27	52
6		8	1	5	0	4	7	2	3

$i=1$, $p=6$ (b)

	0	1	2	3	4	5	6	7	8
maxint		**13**	38	65	97	76	**49**	27	52
6		**(6)**	1	5	0	4	**8**	2	3

$i=2$, $p=7$ (c)

	0	1	2	3	4	5	6	7	8
maxint		13	**27**	65	97	76	49	**38**	52
6		(6)	**(7)**	5	0	4	8	**1**	3

$i=3$, $p=(2)$, 7 (d)

	0	1	2	3	4	5	6	7	8
maxint		13	27	**38**	97	76	49	**65**	52
6		(6)	(7)	**(7)**	0	4	8	**5**	3

$i=4$, $p=(1)$, 6 (e)

	0	1	2	3	4	5	6	7	8
maxint		13	27	38	**49**	76	**97**	65	52
6		(6)	(7)	(7)	**(6)**	4	**0**	5	3

$i=5$, $p=8$ (f)

	0	1	2	3	4	5	6	7	8
maxint		13	27	38	49	**52**	97	65	**76**
6		(6)	(7)	(7)	(6)	**(8)**	0	5	**4**

$i=6$, $p=(3)$, 7 (g)

	0	1	2	3	4	5	6	7	8
maxint		13	27	38	49	52	**65**	**97**	76
6		(6)	(7)	(7)	(6)	(8)	**(7)**	**0**	4

$i=7$, $p=(5)$, 8 (h)

	0	1	2	3	4	5	6	7	8
maxint		13	27	38	49	52	65	**76**	**97**
6		(6)	(7)	(7)	(6)	(8)	(7)	**(8)**	**0**

图 10.4 重排静态链表数组中记录的过程

www.TopSage.com

大家网
TopSage.com

10.2.3 希尔排序

希尔排序(Shell's Sort)又称"缩小增量排序"(Diminishing Increment Sort),它也是一种属插入排序类的方法,但在时间效率上较前述几种排序方法有较大的改进。

从对直接插入排序的分析得知,其算法时间复杂度为 $O(n^2)$,但是,若待排记录序列为"正序"时,其时间复杂度可提高至 $O(n)$。由此可设想,若待排记录序列按关键字"基本有序",即序列中具有下列特性

$$\text{L. r[i]. key} < \max_{1 \leqslant j < i} \{\text{L. r[j]. key}\} \tag{10-7}$$

的记录较少时,直接插入排序的效率就可大大提高,从另一方面来看,由于直接插入排序算法简单,则在 n 值很小时效率也比较高。希尔排序正是从这两点分析出发对直接插入排序进行改进得到的一种插入排序方法。

它的基本思想是:先将整个待排记录序列分割成为若干子序列分别进行直接插入排序,待整个序列中的记录"基本有序"时,再对全体记录进行一次直接插入排序。

仍以式(10-4)中的关键字为例,先看一下希尔排序的过程。初始关键字序列如图10.5的第1行所示。首先将该序列分成5个子序列$\{R_1, R_6\}$,$\{R_2, R_7\}$,…,$\{R_5, R_{10}\}$,如图10.5的第2行至第6行所示,分别对每个子序列进行直接插入排序,排序结果如图10.5的第7行所示,从第1行的初始序列得到第7行的序列的过程称为一趟希尔排序。然后进行第二趟希尔排序,即分别对下列3个子序列:$\{R_1, R_4, R_7, R_{10}\}$,$\{R_2, R_5, R_8\}$和$\{R_3, R_6, R_9\}$进行直接插入排序,其结果如图10.5的第11行所示,最后对整个序列进行一趟直接插入排序。至此,希尔排序结束,整个序列的记录已按关键字非递减有序排列。

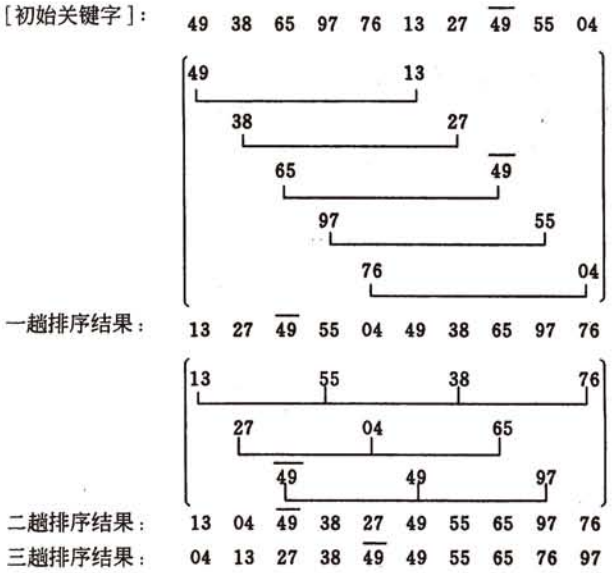


图 10.5 希尔排序示例

从上述排序过程可见,希尔排序的一个特点是:子序列的构成不是简单地"逐段分割",而是将相隔某个"增量"的记录组成一个子序列。如上例中,第一趟排序时的增量为

www.TopSage.com

5，第二趟排序时的增量为 3，由于在前两趟的插入排序中记录的关键字是和同一子序列中的前一个记录的关键字进行比较，因此关键字较小的记录就不是一步一步地往前挪动，而是跳跃式地往前移，从而使得在进行最后一趟增量为 1 的插入排序时，序列已基本有序，只要作记录的少量比较和移动即可完成排序，因此希尔排序的时间复杂度较直接插入排序低。下面用算法语言描述上述希尔排序的过程，为此先将算法 10.1 改写成如算法 10.4 所示的一般形式。希尔排序算法如算法 10.5 所示。

```
void ShellInsert ( SqList &L, int dk ) {
  // 对顺序表 L 作一趟希尔插入排序。本算法是和一趟直接插入排序相比，作了以下修改：
  //      1. 前后记录位置的增量是 dk，而不是 1；
  //      2. r[0]只是暂存单元，不是哨兵。当 j<= 0 时，插入位置已找到。
  for ( i = dk + 1; i<= L.length; ++ i )
     if (LT(L.r[i].key, L.r[i - dk].key)) {   // 需将 L.r[i]插入有序增量子表
       L.r[0] = L.r[i];                       // 暂存在 L.r[0]
       for (j = i - dk;  j>0 && LT(L.r[0].key, L.r[j].key);  j -= dk)
          L.r[j + dk] = L.r[j];               // 记录后移，查找插入位置
       L.r[j + dk] = L.r[0];                  // 插入
     }
}// ShellInsert
```

算法 10.4

```
void ShellSort (SqList &L, int dlta[], int t) {
  // 按增量序列 dlta[0..t - 1]对顺序表 L 作希尔排序。
  for (k = 0; k<t; ++ k)
     ShellInsert(L, dlta[k]);          // 一趟增量为 dlta[k]的插入排序
}// ShellSort
```

算法 10.5

希尔排序的分析是一个复杂的问题，因为它的时间是所取“增量”序列的函数，这涉及一些数学上尚未解决的难题。因此，到目前为止尚未有人求得一种最好的增量序列，但大量的研究已得出一些局部的结论。如有人指出，当增量序列为 $dlta[k]=2^{t-k+1}-1$ 时，希尔排序的时间复杂度为 $O(n^{3/2})$，其中 t 为排序趟数，$1\leqslant k\leqslant t\leqslant\lfloor\log_2(n+1)\rfloor$。还有人在大量的实验基础上推出：当 n 在某个特定范围内，希尔排序所需的比较和移动次数约为 $n^{1.3}$，当 $n\to\infty$ 时，可减少到 $n(\log_2 n)^{2[2]}$。增量序列可以有各种取法①，但需注意：应使增量序列中的值没有除 1 之外的公因子，并且最后一个增量值必须等于 1。

10.3 快速排序

这一节讨论一类藉助“交换”进行排序的方法，其中最简单的一种就是人们所熟知的**起泡排序(Bubble Sort)**。

---

① 其他增量序列如：

…9,5,3,2,1　　$dlta[k]=2^{t-k}+1$　$0\leqslant k\leqslant t\leqslant\lfloor\log_2(n-1)\rfloor$

…40,13,4,1　　$dlta[k]=\frac{1}{2}(3^{t-k}-1)$　$0\leqslant k\leqslant t\leqslant\lfloor\log_3(2n+1)\rfloor$

www.TopSage.com

大家网 TopSage.com

起泡排序的过程很简单。首先将第一个记录的关键字和第二个记录的关键字进行比较，若为逆序(即 L. r[1]. key>L. r[2]. key)，则将两个记录交换之，然后比较第二个记录和第三个记录的关键字。依次类推，直至第 $n-1$ 个记录和第 n 个记录的关键字进行过比较为止。上述过程称做第一趟起泡排序，其结果使得关键字最大的记录被安置到最后一个记录的位置上。然后进行第二趟起泡排序，对前 $n-1$ 个记录进行同样操作，其结果是使关键字次大的记录被安置到第 $n-1$ 个记录的位置上。一般地，第 i 趟起泡排序是从 L. r[1]到 L. r[n−i+1]依次比较相邻两个记录的关键字，并在“逆序”时交换相邻记录，其结果是这 $n-i+1$ 个记录中关键字最大的记录被交换到第 $n-i+1$ 的位置上。整个排序过程需进行 $k(1\leqslant k<n)$趟起泡排序，显然，判别起泡排序结束的条件应该是“在一趟排序过程中没有进行过交换记录的操作”。图 10.6 展示了起泡排序的一个实例。从图中可见，在起泡排序的过程中，关键字较小的记录好比水中气泡逐趟向上飘浮，而关键字较大的记录好比石块往下沉，每一趟有一块“最大”的石头沉到水底(请参见“1.4.3 节算法效率的度量”中起泡排序的算法)。

初始关键字	第一趟排序后	第二趟排序后	第三趟排序后	第四趟排序后	第五趟排序后	第六趟排序后
49	38	38	38	38	13	**13**
38	49	49	49	13	27	**27**
65	65	65	13	27	38	**38**
97	76	13	27	49	**49**	
76	13	27	49	**49**		
13	27	49	**65**			
27	49	**76**				
49	**97**					

图 10.6　起泡排序示例

分析起泡排序的效率，容易看出，若初始序列为“正序”序列，则只需进行一趟排序，在排序过程中进行 $n-1$ 次关键字间的比较，且不移动记录；反之，若初始序列为“逆序”序列，则需进行 $n-1$ 趟排序，需进行 $\sum_{i=n}^{2}(i-1)=n(n-1)/2$ 次比较，并作等数量级的记录移动。因此，总的时间复杂度为 $O(n^2)$。

快速排序(Quick Sort)是对起泡排序的一种改进。它的基本思想是，通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

假设待排序的序列为{L. r[s]，L. r[s+1]，…，L. r[t]}，首先任意选取一个记录(通常可选第一个记录 L. r[s])作为**枢轴**(或支点)(pivot)，然后按下述原则重新排列其余记录：将所有关键字较它小的记录都安置在它的位置之前，将所有关键字较它大的记录都安置在它的位置之后。由此可以该“枢轴”记录最后所落的位置 i 作分界线，将序列

www.TopSage.com

{L.r[s],…, L.r[t]}分割成两个子序列{L.r[s],L.r[s+1],…,L.r[i-1]}和{L.r[i+1],L.r[i+2],…,L.r[t]}。这个过程称做一趟快速排序(或一次划分)。

一趟快速排序的具体做法是:附设两个指针 low 和 high,它们的初值分别为 low 和 high,设枢轴记录的关键字为 pivotkey,则首先从 high 所指位置起向前搜索找到第一个关键字小于 pivotkey 的记录和枢轴记录互相交换,然后从 low 所指位置起向后搜索,找到第一个关键字大于 pivotkey 的记录和枢轴记录互相交换,重复这两步直至 low=high 为止。其算法如算法 10.6(a)所示。

```
int Partition (SqList &L, int low, int high) {
  // 交换顺序表 L 中子表 L.r[low..high]的记录,使枢轴记录到位,并返回其所在位置,此时
  // 在它之前(后)的记录均不大(小)于它。
  pivotkey = L.r[low].key;                          // 用子表的第一个记录作枢轴记录
  while (low<high) {                                // 从表的两端交替地向中间扫描
    while (low<high && L.r[high].key>=pivotkey)
      --high;
    L.r[low]←→L.r[high];                            // 将比枢轴记录小的记录交换到低端
    while (low<high && L.r[low].key<=pivotkey)
      ++low;
    L.r[low]←→L.r[high];                            // 将比枢轴记录大的记录交换到高端
  }
  return low;                                       // 返回枢轴所在位置
}// Partition
```

算法 10.6(a)

具体实现上述算法时,每交换一对记录需进行 3 次记录移动(赋值)的操作。而实际上,在排序过程中对枢轴记录的赋值是多余的,因为只有在一趟排序结束时,即 low=high 的位置才是枢轴记录的最后位置。由此可改写上述算法,先将枢轴记录暂存在 r[0]的位置上,排序过程中只作 r[low]或 r[high]的单向移动,直至一趟排序结束后再将枢轴记录移至正确位置上。如算法 10.6 (b)所示。

```
int Partition (SqList &L, int low, int high) {
  // 交换顺序表 L 中子表 r[low..high]的记录,枢轴记录到位,并返回其所在位置,此时
  // 在它之前(后)的记录均不大(小)于它。
  L.r[0] = L.r[low];                                // 用子表的第一个记录作枢轴记录
  pivotkey = L.r[low].key;                          // 枢轴记录关键字
  while (low<high) {                                // 从表的两端交替地向中间扫描
    while (low<high && L.r[high].key>=pivotkey) --high;
    L.r[low] = L.r[high];                           // 将比枢轴记录小的记录移到低端
    while (low<high && L.r[low].key<=pivotkey) ++low;
    L.r[high] = L.r[low];                           // 将比枢轴记录大的记录移到高端
  }
  L.r[low] = L.r[0];                                // 枢轴记录到位
  return low;                                       // 返回枢轴位置
}// Partition
```

算法 10.6(b)

以式(10-4)中的关键字为例,一趟快排的过程如图 10.7(a)所示。整个快速排序的过程可递归进行。若待排序列中只有一个记录,显然已有序,否则进行一趟快速排序后再分

www.TopSage.com

大家网 TopSage.com

别对分割所得的两个子序列进行快速排序，如图 10.7(b)所示。

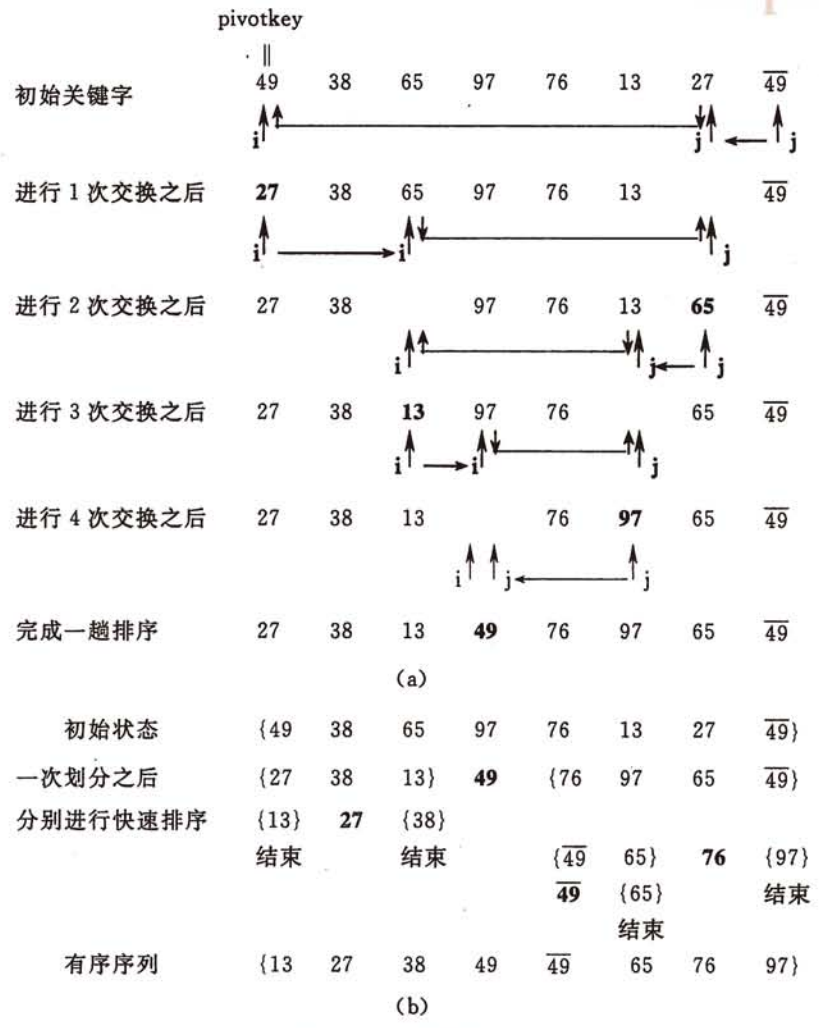


图 10.7　快速排序示例

(a) 一趟快排过程；(b) 排序的全过程

递归形式的快速排序算法如算法 10.7 和算法 10.8 所示。

```
void QSort (SqList &L, int low, int high) {
    // 对顺序表 L 中的子序列 L.r[low..high]作快速排序
    if (low < high) {                          // 长度大于 1
        pivotloc = Partition(L, low, high);    // 将 L.r[low..high]一分为二
        QSort(L, low, pivotloc - 1);           // 对低子表递归排序,pivotloc 是枢轴位置
        QSort(L, pivotloc + 1, high);          // 对高子表递归排序
    }
}// QSort
```

算法　10.7

www.TopSage.com

```
void QuickSort(SqList &L) {
  // 对顺序表 L 作快速排序。
  QSort(L, 1, L.length);
}// QuickSort
```

算法 10.8

快速排序的平均时间为 $T_{avg}(n)=knlnn$，其中 n 为待排序序列中记录的个数，k 为某个常数，经验证明，在所有同数量级的此类(先进的)排序方法中，快速排序的常数因子 k 最小。因此，就平均时间而言，快速排序是目前被认为是最好的一种内部排序方法。

下面我们来分析快速排序的平均时间性能。

假设 $T(n)$为对 n 个记录 L.r[1..n]进行快速排序所需时间，则由算法 QuickSort 可见，

$$T(n)=T_{pass}(n)+T(k-1)+T(n-k)$$

其中 $T_{pass}(n)$为对 n 个记录进行一趟快速排序 Partition(L,1,n)所需时间，从算法 10.7 可见，它和记录数 n 成正比，可以 cn 表示之(c 为某个常数)；$T(k-1)$和 $T(n-k)$分别为对L.r[1..k-1]和 L.r[k+1..n]中记录进行快速排序 QSort(L,1,k-1)和 QSort(L,k+1,n)所需时间。假设待排序列中的记录是随机排列的，则在一趟排序之后，k 取 1 至 n 之间任何一值的概率相同，快速排序所需时间的平均值则为

$$\begin{aligned} T_{avg}(n) &= cn+\frac{1}{n}\sum_{k=1}^{n}[T_{avg}(k-1)+T_{avg}(n-k)] \\ &= cn+\frac{2}{n}\sum_{i=0}^{n-1}T_{avg}(i) \end{aligned} \tag{10-8}$$

假定 $T_{avg}(1)\leqslant b$(b 为某个常量)，类同式(9-19)的推导，由式(10-8)可推出

$$\begin{aligned} T_{avg}(n) &= \frac{n+1}{n}T_{avg}(n-1)+\frac{2n-1}{n}c \\ &< \frac{n+1}{2}T_{avg}(1)+2(n+1)\left(\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n+1}\right)c \\ &< \left(\frac{b}{2}+2c\right)(n+1)\ln(n+1) \quad n\geqslant 2 \end{aligned} \tag{10-9}$$

通常，快速排序被认为是，在所有同数量级($O(n\log n)$)的排序方法中，其平均性能最好。但是，若初始记录序列按关键字有序或基本有序时，快速排序将蜕化为起泡排序，其时间复杂度为 $O(n^2)$。为改进之，通常依“三者取中”的法则来选取枢轴记录，即比较 L.r[s].key、L.r[t].key 和 L.r$\left[\left\lfloor\frac{s+t}{2}\right\rfloor\right]$.key，取三者中其关键字取中值的记录为枢轴，只要将该记录和 L.r[s]互换，算法 10.6(b)′不变。经验证明，采用三者取中的规则可大大改善快速排序在最坏情况下的性能。然而，即使如此，也不能使快速排序在待排记录序列已按关键字有序的情况下达到 $O(n)$的时间复杂度。为此，可如下所述修改“一次划分”算法：在指针 high 减 1 和 low 增 1 的同时进行“起泡”操作，即在相邻两个记录处于“逆序”时进行互换，同时在算法中附设两个布尔型变量分别指示指针 low 和 high 在从两端向中间的移动过程中是否进行过交换记录的操作，若指针 low 在从低端向中间的移动过

www.TopSage.com

大家网 www.TopSage.com

程中没有进行交换记录的操作，则不再需要对低端子表进行排序；类似地，若指针 high 在从高端向中间的移动过程中没有进行交换记录的操作，则不再需要对高端子表进行排序。显然，如此“划分”将进一步改善快速排序的平均性能。

由以上讨论可知，从时间上看，快速排序的平均性能优于前面讨论过的各种排序方法，从空间上看，前面讨论的各种方法，除 2-路插入排序之外，都只需要一个记录的附加空间即可，但快速排序需一个栈空间来实现递归。若每一趟排序都将记录序列均匀地分割成长度相接近的两个子序列，则栈的最大深度为$\lfloor \log_2 n \rfloor + 1$（包括最外层参量进栈），但是，若每趟排序之后，枢轴位置均偏向子序列的一端，则为最坏情况，栈的最大深度为 n。如果改写算法10.7，在一趟排序之后比较分割所得两部分的长度，且先对长度短的子序列中的记录进行快速排序，则栈的最大深度可降为 $O(\log n)$。

10.4 选择排序

选择排序(Selection Sort)的基本思想是：每一趟在 $n-i+1(i=1,2,\cdots,n-1)$个记录中选取关键字最小的记录作为有序序列中第 i 个记录。其中最简单且为读者最熟悉的是**简单选择排序**(Simple Selection Sort)。

10.4.1 简单选择排序

一趟简单选择排序的操作为：通过 $n-i$ 次关键字间的比较，从 $n-i+1$ 个记录中选出关键字最小的记录，并和第 $i(1\leqslant i\leqslant n)$个记录交换之。

显然，对 L.r[1..n]中记录进行简单选择排序的算法为：令 i 从 1 至 $n-1$，进行 $n-1$ 趟选择操作，如算法 10.9 所示。容易看出，简单选择排序过程中，所需进行记录移动的操作次数较少，其最小值为“0”，最大值为 $3(n-1)$。然而，无论记录的初始排列如何，所需进行的关键字间的比较次数相同，均为 $n(n-1)/2$。因此，总的时间复杂度也是$O(n^2)$。

```
void SelectSort (SqList &L) {
  // 对顺序表 L 作简单选择排序。
  for (i = 1; i<L.length; ++ i) {          // 选择第 i 小的记录，并交换到位
    j = SelectMinKey(L, i);                // 在 L.r[i..L.length]中选择 key 最小的记录
    if (i!= j) L.r[i]←→L.r[j];             // 与第 i 个记录交换
  }
}// SelectSort
```

算法 10.9

那么，能否加以改进呢？

从上述可见，选择排序的主要操作是进行关键字间的比较，因此改进简单选择排序应从如何减少“比较”出发考虑。显然，在 n 个关键字中选出最小值，至少进行 $n-1$ 次比较，然而，继续在剩余的 $n-1$ 个关键字中选择次小值就并非一定要进行 $n-2$ 次比较，若能利用前 $n-1$ 次比较所得信息，则可减少以后各趟选择排序中所用的比较次数。实际上，体育比赛中的锦标赛便是一种选择排序。例如，在 8 个运动员中决出前 3 名至多需要 11 场

www.TopSage.com

比赛，而不是 7+6+5=18 场比赛(它的前提是，若乙胜丙，甲胜乙，则认为甲必能胜丙)。例如，图 10.8(a)中最低层的叶子结点中 8 个选手之间经过第一轮的 4 场比赛之后选拔出 4 个优胜者“CHA”、“BAO”、“DIAO”和“WANG”，然后经过两场半决赛和一场决赛之后，选拔出冠军“BAO”。显然，按照锦标赛的传递关系，亚军只能产生于分别在决赛，半决赛和第一轮比赛中输给冠军的选手中。由此，在经过“CHA”和“LIU”、“CHA”和“DIAO”的两场比赛之后，选拔出亚军“CHA”，同理，选拔殿军的比赛只要在“ZHAO”、“LIU”和“DIAO”3 个选手之间进行即可。按照这种锦标赛的思想可导出树形选择排序。

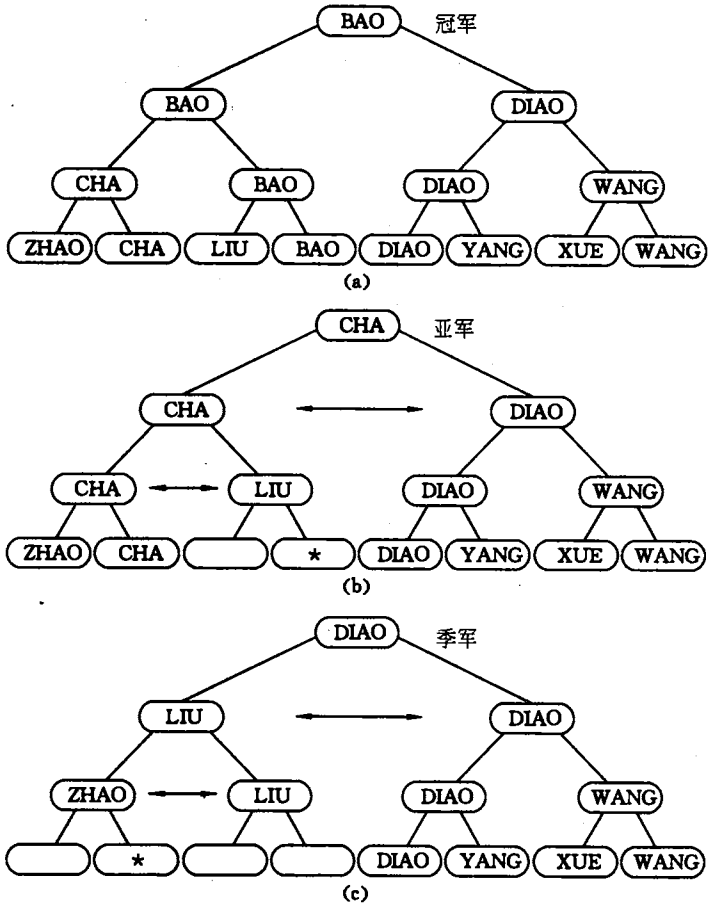


图 10.8　锦标赛过程示意图

(a) 选拔冠军的比赛程序；(b)选拔亚军的两场比赛；(c) 选拔季军的两场比赛

10.4.2　树形选择排序

树形选择排序(Tree Selection Sort)，又称锦标赛排序(Tournament Sort)，是一种按照锦标赛的思想进行选择排序的方法。首先对 n 个记录的关键字进行两两比较，然后在其中$\lceil \frac{n}{2} \rceil$个较小者之间再进行两两比较，如此重复，直至选出最小关键字的记录为止。

www.TopSage.com

这个过程可用一棵有 n 个叶子结点的完全二叉树表示。例如，图 10.9(a)中的二叉树表示从 8 个关键字中选出最小关键字的过程。8 个叶子结点中依次存放排序之前的 8 个关键字，每个非终端结点中的关键字均等于其左、右孩子结点中较小的关键字，则根结点中的关键字即为叶子结点中的最小关键字。在输出最小关键字之后，根据关系的可传递性，欲选出次小关键字，仅需将叶子结点中的最小关键字(13)改为“最大值”，然后从该叶子结点开始，和其左(或右)兄弟的关键字进行比较，修改从叶子结点到根的路径上各结点的关键字，则根结点的关键字即为次小关键字。同理，可依次选出从小到大的所有关键字(参见图 10.9(b)和(c))。由于含有 n 个叶子结点的完全二叉树的深度为$\lceil \log_2 n \rceil + 1$，则在树形选择排序中，除了最小关键字之外，每选择一个次小关键字仅需进行$\lceil \log_2 n \rceil$次比较，因此，它的时间复杂度为 $O(n\log n)$。但是，这种排序方法尚有辅助存储空间较多、和“最大值”进行多余的比较等缺点。为了弥补，威洛姆斯(J. willioms)在 1964 年提出了另一种形式的选择排序——堆排序。

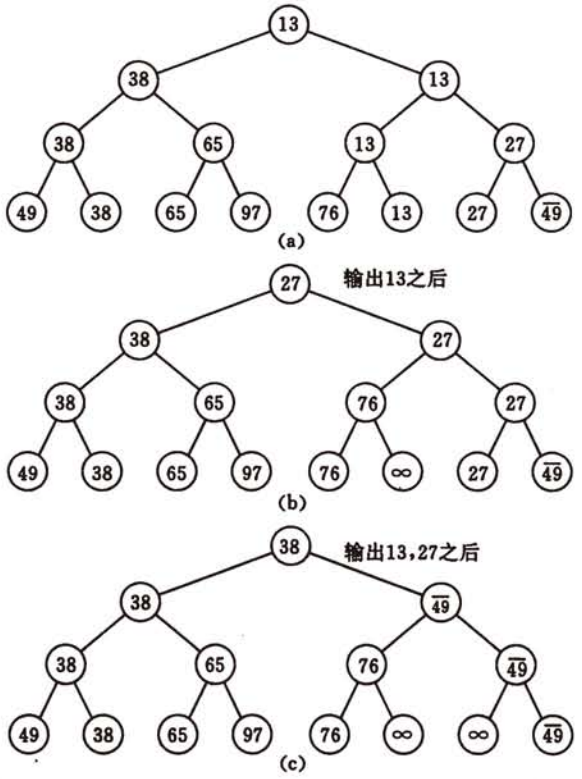


图 10.9　树形选择排序示例

(a) 选出最小关键字为 13；(b) 选出次小关键字为 27；(c) 选出居第三的关键字为 38

10.4.3　堆排序

堆排序(Heap Sort)只需要一个记录大小的辅助空间，每个待排序的记录仅占有一个存储空间。

www.TopSage.com

堆的定义如下：n 个元素的序列$\{k_1,k_2,\cdots,k_n\}$当且仅当满足下关系时，称之为堆。

$$\begin{cases} k_i \leqslant k_{2i} \\ k_i \leqslant k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geqslant k_{2i} \\ k_i \geqslant k_{2i+1} \end{cases}$$

$$\left(i = 1,2,\cdots,\left\lfloor \frac{n}{2} \right\rfloor\right)$$

若将和此序列对应的一维数组（即以一维数组作此序列的存储结构）看成是一个完全二叉树，则堆的含义表明，完全二叉树中所有非终端结点的值均不大于（或不小于）其左、右孩子结点的值。由此，若序列$\{k_1,k_2,\cdots,k_n\}$是堆，则堆顶元素（或完全二叉树的根）必为序列中 n 个元素的最小值（或最大值）。例如，下列两个序列为堆，对应的完全二叉树如图 10.10 所示。

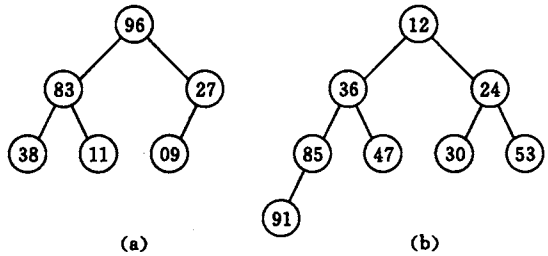


图 10.10　堆的示例

（a）堆顶元素取最大值；（b）堆顶元素取最小值

$\{96,83,27,38,11,09\}$

$\{12,36,24,85,47,30,53,91\}$

若在输出堆顶的最小值之后，使得剩余 $n-1$ 个元素的序列重又建成一个堆，则得到 n 个元素中的次小值。如此反复执行，便能得到一个有序序列，这个过程称之为**堆排序**。

由此，实现堆排序需要解决两个问题：(1)如何由一个无序序列建成一个堆？(2)如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？

下面先讨论第二个问题。例如，图 10.11(a)是个堆，假设输出堆顶元素之后，以堆中最后一个元素替代之，如图 10.11(b)所示。此时根结点的左、右子树均为堆，则仅需自上至下进行调整即可。首先以堆顶元素和其左、右子树根结点的值比较之，由于右子树根结点的值小于左子树根结点的值且小于根结点的值，则将 27 和 97 交换之；由于 97 替代了 27 之后破坏了右子树的“堆”，则需进行和上述相同的调整，直至叶子结点，调整后的状态如图 10.11(c)所示，此时堆顶为 $n-1$ 个元素中的最小值。重复上述过程，将堆顶元素 27 和堆中最后一个元素 97 交换且调整，得到如图 10.11(d)所示新的堆。

我们称这个自堆顶至叶子的调整过程为“筛选”。

从一个无序序列建堆的过程就是一个反复“筛选”的过程。若将此序列看成是一个完全二叉树，则最后一个非终端结点是第$\lfloor n/2 \rfloor$个元素，由此“筛选”只需从第$\lfloor n/2 \rfloor$个元素开始。例如，图 10.12(a)中的二叉树表示一个有 8 个元素的无序序列

$$\{49,38,65,97,76,13,27,\overline{49}\}$$

www.TopSage.com

大家网 TopSage.com

则筛选从第 4 个元素开始，由于 97>$\overline{49}$，则交换之，交换后的序列如图 10.12(b)所示，同理，在第 3 个元素 65 被筛选之后序列的状态如图 10.12(c)所示。由于第 2 个元素 38 不大于其左、右子树根的值，则筛选后的序列不变。图 10.12(e)所示为筛选根元素 49 之后建成的堆。

堆排序的算法如算法 10.11 所示，其中筛选的算法如算法 10.10 所示。为使排序结果和 10.1 节中的定义一致，即：使记录序列按关键字非递减有序排列，则在堆排序的算法

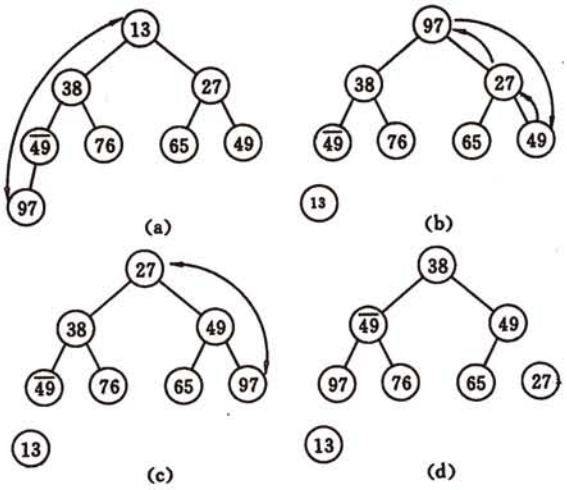


图 10.11　输出堆顶元素并调整建新堆的过程

(a) 堆；(b) 13 和 97 交换后的情形；(c) 调整后的新堆；

(d) 27 和 97 交换后再进行调整建成的新堆

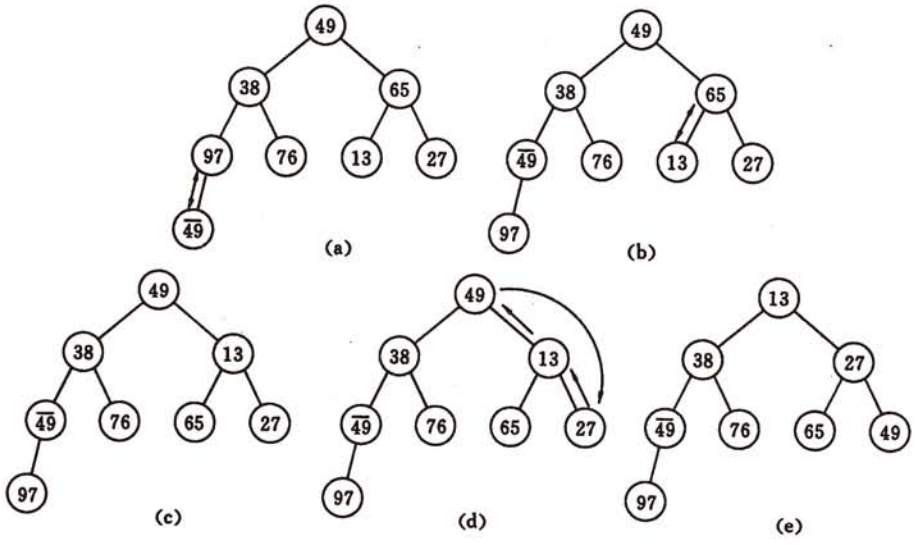


图 10.12　建初始堆过程示例

(a) 无序序列；(b) 97 被筛选之后的状态；(c) 65 被筛选之后的状态；

(d) 38 被筛选之后的状态；(e) 49 被筛选之后建成的堆

www.TopSage.com

中先建一个“大顶堆”，即先选得一个关键字为最大的记录并与序列中最后一个记录交换，然后对序列中前 $n-1$ 记录进行筛选，重新将它调整为一个“大顶堆”，如此反复直至排序结束。由此，“筛选”应沿关键字较大的孩子结点向下进行。

```
typedef SqList HeapType;          // 堆采用顺序表存储表示

void HeapAdjust (HeapType &H, int s, int m) {
  // 已知 H.r[s..m]中记录的关键字除 H.r[s].key 之外均满足堆的定义，本函数调整 H.r[s]
  // 的关键字，使 H.r[s..m]成为一个大顶堆(对其中记录的关键字而言)
  rc = H.r[s];
  for ( j = 2 * s; j<= m; j * = 2 ) {   // 沿 key 较大的孩子结点向下筛选
    if ( j<m && LT(H.r[j].key,H.r[j + 1].key) )   ++ j;     // j 为 key 较大的记录的下标
    if ( !LT(rc.key,H.r[j].key) )  break;                   // rc 应插入在位置 s 上
    H.r[s] = H.r[j];  s = j;
  }
  H.r[s] = rc;                                              // 插入
}// HeapAdjust
```

算法 10.10

```
void HeapSort ( HeapType &H ) {
  // 对顺序表 H 进行堆排序。
  for ( i = H.length/2; i>0; --i )        // 把 H.r[1..H.length]建成大顶堆
    HeapAdjust ( H, i, H.length );
  for ( i = H.length; i>1; --i ) {
    H.r[1]←→H.r[i];                       // 将堆顶记录和当前未经排序子序列 Hr[1..i]中
                                          // 最后一个记录相互交换
    HeapAdjust(H, 1, i - 1);              // 将 H.r[1..i - 1] 重新调整为大顶堆
  }
}// HeapSort
```

算法 10.11

堆排序方法对记录数较少的文件并不值得提倡，但对 n 较大的文件还是很有效的。因为其运行时间主要耗费在建初始堆和调整建新堆时进行的反复“筛选”上。对深度为 k 的堆，筛选算法中进行的关键字比较次数至多为 $2(k-1)$次，则在建含 n 个元素、深度为 h 的堆时，总共进行的关键字比较次数不超过 $4n$。[①] 又，n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$，则调整建新堆时调用HeapAdjust过程 $n-1$ 次，总共进行的比较次数不超过

① 由于第 i 层上的结点数至多为 2^{i-1}，以它们为根的二叉树的深度为 $h-i+1$，则调用 $\lfloor \frac{n}{2} \rfloor$ 次 HeapAdjust 过程时总共进行的关键字比较次数不超过下式之值：

$$\sum_{i=h-1}^{1} 2^{i-1} \cdot 2(h-i) = \sum_{i=h-1}^{1} 2^{i} \cdot (h-i) = \sum_{j=1}^{h-1} 2^{h-j} \cdot j \leqslant (2n) \sum_{j=1}^{h-1} j/2^{j} \leqslant 4n$$

www.TopSage.com

大家网 TopSage.com

下式之值，

$$2(\lfloor\log_2(n-1)\rfloor+\lfloor\log_2(n-2)\rfloor+\cdots+\log_2 2)<2n(\lfloor\log_2 n\rfloor)$$

由此，堆排序在最坏的情况下，其时间复杂度也为 $O(n\log n)$。相对于快速排序来说，这是堆排序的最大优点。此外，堆排序仅需一个记录大小供交换用的辅助存储空间。

10.5 归并排序

归并排序(Merging Sort)是又一类不同的排序方法。“归并”的含义是将两个或两个以上的有序表组合成一个新的有序表。它的实现方法早已为读者所熟悉，无论是顺序存储结构还是链表存储结构，都可在 $O(m+n)$[①]的时间量级上实现。利用归并的思想容易实现排序。假设初始序列含有 n 个记录，则可看成是 n 个有序的子序列，每个子序列的长度为 1，然后两两归并，得到$\lceil\frac{n}{2}\rceil$个长度为 2 或 1 的有序子序列；再两两归并，……，如此重复，直至得到一个长度为 n 的有序序列为止，这种排序方法称为 2-路归并排序。例如图 10.13 为 2-路归并排序的一个例子。

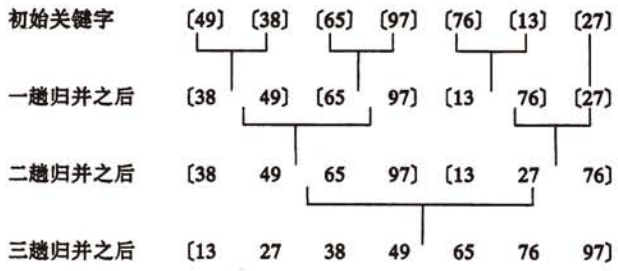


图 10.13 2-路归并排序示例

2-路归并排序中的核心操作是将一维数组中前后相邻的两个有序序列归并为一个有序序列，其算法(类似于算法 2.7)如算法 10.12 所示。

```
void Merge (RcdType SR[], RcdType &TR[], int i, int m, int n) {
  // 将有序的 SR[i..m]和 SR[m+1..n]归并为有序的 TR[i..n]
  for (j=m+1, k=i; i<=m && j<=n; ++k) {          // 将 SR 中记录由小到大地并入 TR
    if (LQ(SR[i].key,SR[j].key)) TR[k] = SR[i++];
    else TR[k] = SR[j++];
  }
  if (i<=m) TR[k..n] = SR[i..m];                 // 将剩余的 SR[i..m]复制到 TR
  if (j<=n) TR[k..n] = SR[j..n];                 // 将剩余的 SR[j..n]复制到 TR
}// Merge
```

算法 10.12

① 假设两个有序表的长度分别为 m 和 n。

www.TopSage.com

一趟归并排序的操作是，调用$\lceil \frac{n}{2h} \rceil$次算法 merge 将 SR[1..n]中前后相邻且长度为 h 的有序段进行两两归并，得到前后相邻、长度为 $2h$ 的有序段，并存放在 TR[1..n]中，整个归并排序需进行$\lceil \log_2 n \rceil$趟。可见，实现归并排序需和待排记录等数量的辅助空间，其时间复杂度为 $O(n\log n)$。

递归形式的 2-路归并排序的算法如算法 10.13 和算法 10.14 所示。值得提醒的是，递归形式的算法在形式上较简洁，但实用性很差。其非递归形式的算法可查阅参考书目[1]。

与快速排序和堆排序相比，归并排序的最大特点是，它是一种稳定的排序方法。但在一般情况下，很少利用 2-路归并排序法进行内部排序，其他形式的归并排序如本书习题集中习题 10.17 所述。

```
void MSort ( RcdType SR[], RcdType &TR1[], int s, int t ) {
  // 将 SR[s..t]归并排序为 TR1[s..t]。
  if (s == t) TR1[s] = SR[s];
  else {
    m = (s + t)/2;              // 将 SR[s..t]平分为 SR[s..m]和 SR[m + 1..t]
    MSort (SR, TR2, s, m);      // 递归地将 SR[s..m]归并为有序的 TR2[s..m]
    MSort (SR, TR2, m + 1, t);  // 递归地将 SR[m + 1..t]归并为有序的 TR2[m + 1..t]
    Merge (TR2, TR1, s, m, t);  // 将 TR2[s..m]和 TR2[m + 1..t]归并到 TR1[s..t]
  }
}// MSort
```

算法 10.13

```
void MergeSort (SqList &L) {
  // 对顺序表 L 作归并排序。
  MSort(L.r, L.r, 1, L.length);
}// MergeSort
```

算法 10.14

10.6 基数排序

基数排序(Radix Sorting)是和前面所述各类排序方法完全不相同的一种排序方法。从前几节的讨论可见，实现排序主要是通过关键字间的比较和移动记录这两种操作，而实现基数排序不需要进行记录关键字间的比较。基数排序是一种借助多关键字排序的思想对单逻辑关键字进行排序的方法。

10.6.1 多关键字的排序

什么是多关键字排序问题？先看一个具体例子。

已知扑克牌中 52 张牌面的次序关系为：

$$\clubsuit 2 < \clubsuit 3 < \cdots < \clubsuit A < \blacklozenge 2 < \blacklozenge 3 < \cdots < \blacklozenge A$$

www.TopSage.com

大家网 TopSage.com

$$<♥2<♥3<\cdots<♥A<♠2<♠3<\cdots<♠A$$

每一张牌有两个“关键字”：花色(♣<◆<♥<♠)和面值(2<3<…<A)，且“花色”的地位高于“面值”，在比较任意两张牌面的大小时，必须先比较“花色”，若“花色”相同，则再比较面值。由此，将扑克牌整理成如上所述次序关系时，通常采用的办法是：先按不同“花色”分成有次序的4堆，每一堆的牌均具有相同的“花色”，然后分别对每一堆按“面值”大小整理有序。

也可采用另一种办法：先按不同“面值”分成13堆，然后将这13堆牌自小至大叠在一起(“3”在“2”之上，“4”在“3”之上，……，最上面的是4张“A”)，然后将这付牌整个颠倒过来再重新按不同“花色”分成4堆，最后将这4堆牌按自小至大的次序合在一起(♣在最下面，♠在最上面)，此时同样得到一付满足如上次序关系的牌。这两种整理扑克牌的方法便是两种多关键字的排序方法。

一般情况下，假设有 n 个记录的序列

$$\{R_1,R_2,\cdots,R_n\} \tag{10-10}$$

且每个记录 R_i 中含有 d 个关键字($K_i^0,K_i^1,\cdots,K_i^{d-1}$)，则称序列(10-10)对关键字($K^0,K^1,\cdots,K^{d-1}$)有序是指：对于序列中任意两个记录 R_i 和 R_j($1\leqslant i<j\leqslant n$)都满足下列有序关系[①]：

$$(K_i^0,K_i^1,\cdots,K_i^{d-1})<(K_j^0,K_j^1,\cdots,K_j^{d-1})$$

其中 K^0 称为最主位关键字，K^{d-1} 称为最次位关键字。为实现多关键字排序，通常有两种方法：第一种方法是：先对最主位关键字 K^0 进行排序，将序列分成若干子序列，每个子序列中的记录都具有相同的 K^0 值，然后分别就每个子序列对关键字 K^1 进行排序，按 K^1 值不同再分成若干更小的子序列，依次重复，直至对 K^{d-2} 进行排序之后得到的每一子序列中的记录都具有相同的关键字($K^0,K^1,\cdots,K^{d-2}$)，而后分别每个子序列对 K^{d-1} 进行排序，最后将所有子序列依次联接在一起成为一个有序序列，这种方法称之为**最高位优先**(Most Significant Digit first)法，简称MSD法；第二种方法是从最次位关键字 K^{d-1} 起进行排序。然后再对高一位的关键字 K^{d-2} 进行排序，依次重复，直至对 K^0 进行排序后便成为一个有序序列。这种方法称之为**最低位优先**(Least Significant Digit first)法，简称LSD法。

MSD和LSD只约定按什么样的“关键字次序”来进行排序，而未规定对每个关键字进行排序时所用的方法。但从上面所述可以看出这两种排序方法的不同特点：若按MSD进行排序，必须将序列逐层分割成若干子序列，然后对各子序列分别进行排序；而按LSD进行排序时，不必分成子序列，对每个关键字都是整个序列参加排序，但对 $K^i(0\leqslant i\leqslant d-2)$ 进行排序时，只能用稳定的排序方法。另一方面，按LSD进行排序时，在一定的条件下(即对前一个关键字 $K^i(0\leqslant i\leqslant d-2)$ 的不同值，后一个关键字 K^{i+1} 均取相同值)，也可以不利用前几节所述各种通过关键字间的比较来实现排序的方法，而是通过若干次“分配”和“收集”来实现排序，如上述第二种整理扑克牌的方法那样。

---

① $(a^0,a^1,\cdots,a^{d-1})<(b^0,b^1,\cdots,b^{d-1})$ 是指必定存在 l，使得：当 $s=0,\cdots,l-1$ 时，$a^s=b^s$，而 $a^l<b^l$。

www.TopSage.com

10.6.2 链式基数排序

基数排序是借助“分配”和“收集”两种操作对单逻辑关键字进行排序的一种内部排序方法。

有的逻辑关键字可以看成由若干个关键字复合而成的。例如，若关键字是数值，且其值都在 $0 \leqslant K \leqslant 999$ 范围内，则可把每一个十进制数字看成一个关键字，即可认为 K 由 3 个关键字(K^0，K^1，K^2)组成，其中 K^0 是百位数，K^1 是十位数，K^2 是个位数；又若关键字 K 是由 5 个字母组成的单词，则可看成是由 5 个关键字(K^0，K^1，K^2，K^3，K^4)组成，其中 K^{j-1}是(自左至右的)第 $j+1$ 个字母。由于如此分解而得的每个关键字 K^j 都在相同的范围内(对数字，$0 \leqslant K^j \leqslant 9$，对字母 $'A' \leqslant K^j \leqslant 'Z'$)，则按 LSD 进行排序更为方便，只要从最低数位关键字起，按关键字的不同值将序列中记录“分配”到 RADIX 个队列中后再“收集”之，如此重复 d 次。按这种方法实现排序称之为基数排序，其中“基”指的是 RADIX 的取值范围，在上述两种关键字的情况下，它们分别为“10”和“26”。

实际上，早在计算机出现之前，利用卡片分类机对穿孔卡上的记录进行排序就是用的这种方法。然而，在计算机出现之后却长期得不到应用，原因是所需的辅助存储量(RADIX×N 个记录空间)太大。直到 1954 年有人提出用“计数”代替“分配”才使基数排序得以在计算机上实现，但此时仍需要 n 个记录和 2×RADIX 个计数单元的辅助空间。此后，有人提出用链表作存储结构，则又省去了 n 个记录的辅助空间。下面我们就来介绍这种“链式基数排序”的方法。

先看一个具体例子。首先以静态链表存储 n 个待排记录，并令表头指针指向第一个记录，如图 10.14(a)所示；第一趟分配对最低数位关键字(个位数)进行，改变记录的指针值将链表中的记录分配至 10 个链队列中去，每个队列中的记录关键字的个位数相等，如图 10.14(b)所示，其中 f[i]和 e[i]分别为第 i 个队列的头指针和尾指针；第一趟收集是改变所有非空队列的队尾记录的指针域，令其指向下一个非空队列的队头记录，重新将 10 个队列中的记录链成一个链表，如图 10.14(c)所示；第二趟分配，第二趟收集及第三趟分配和第三趟收集分别是对十位数和百位数进行的，其过程和个位数相同，如图 10.14(d)～(g)所示。至此排序完毕。

在描述算法之前，尚需定义新的数据类型

```
#define  MAX_NUM_OF_KEY  8          // 关键字项数的最大值
#define  RADIX           10         // 关键字基数，此时是十进制整数的基数
#define  MAX_SPACE   10000
typedef struct {
  KeysType keys[MAX_NUM_OF_KEY];    // 关键字
  InfoType otheritems;              // 其他数据项
  int next;
}SLCell;                            // 静态链表的结点类型
typedef struct {
  SLCell  r[MAX_SPACE];             // 静态链表的可利用空间，r[0]为头结点
  int     keynum;                   // 记录的当前关键字个数
  int     recnum;                   // 静态链表的当前长度
}SLList;                            // 静态链表类型
typedef int ArrType[RADIX];         // 指针数组类型
```


www.TopSage.com

大家网
TopSage.com

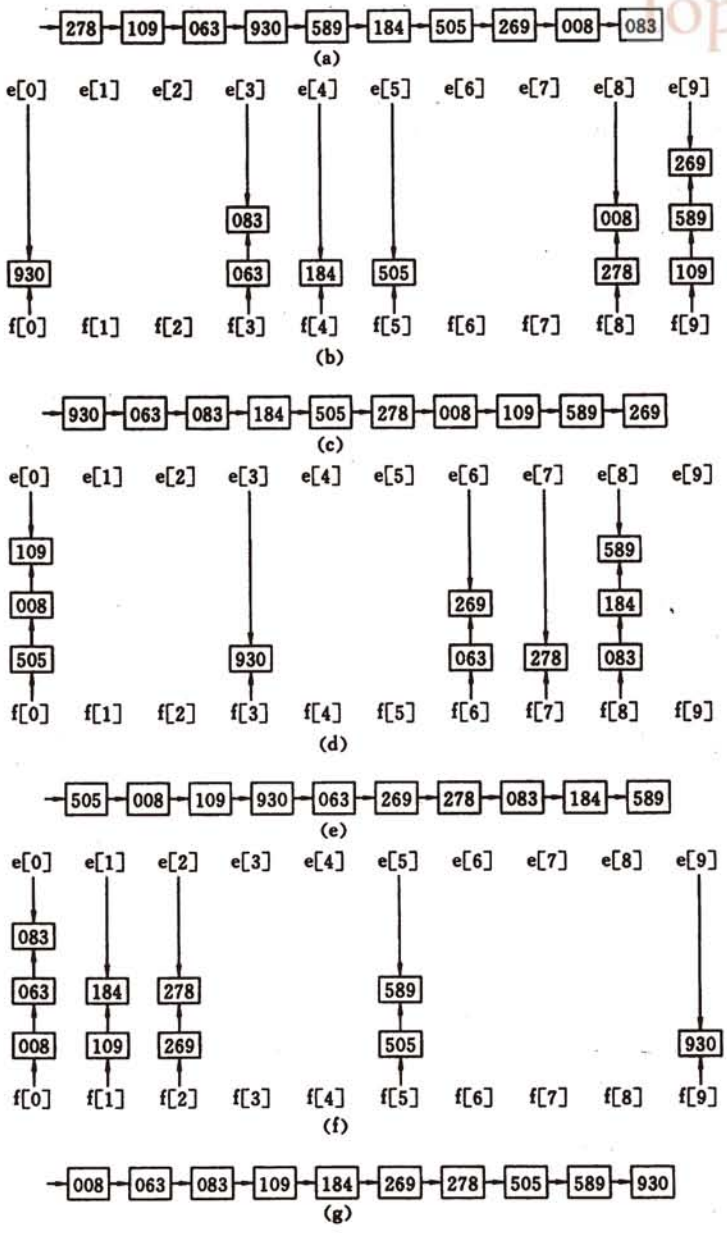


图 10.14　链式基数排序示例

(a) 初始状态；(b) 第一趟分配之后；(c) 第一趟收集之后；(d) 第二趟分配之后；(e)第二趟收集之后；(f)第三趟分配之后；(g)第三趟收集之后的有序文件

算法 10.15 为链式基数排序中一趟分配的算法，算法 10.16 为一趟收集的算法，算法 10.17 为链式基数排序的算法。从算法中容易看出，对于 n 个记录(假设每个记录含 d 个关键字，每个关键字的取值范围为 rd 个值)进行链式基数排序的时间复杂度为 $O(d(n+rd))$，其中每一趟分配的时间复杂度为 $O(n)$，每一趟收集的时间复杂度为 $O(rd)$，整个排序需进行 d 趟分配和收集。所需辅助空间为 $2rd$ 个队列指针。当然，由于

www.TopSage.com

需用链表作存储结构，则相对于其他以顺序结构存储记录的排序方法而言，还增加了 n 个指针域的空间。

```
void Distribute (SLCell &r, int i, ArrType &f, ArrType &e) {
  // 静态链表 L 的 r 域中记录已按(keys[0],...,keys[i-1])有序。
  // 本算法按第 i 个关键字 keys[i]建立 RADIX 个子表，使同一子表中记录的 keys[i]相同.
  // f[0..RADIX-1]和 e[0..RADIX-1]分别指向各子表中第一个和最后一个记录。
  for (j=0; j<Radix; ++j) f[j] = 0;          // 各子表初始化为空表
  for (p=r[0].next; p; p=r[p].next) {
    j = ord(r[p].keys[i]);                   // ord 将记录中第 i 个关键字映射到[0..RADIX-1],
    if ( !f[j] ) f[j] = p;
    else r[e[j]].next = p;
    e[j] = p;                                // 将 p 所指的结点插入第 j 个子表中
  }
}// Distribute
```

算法 10.15

```
  void Collect (SLCell &r, int i, ArrType f, ArrType e) {
  // 本算法按 keys[i]自小至大地将 f[0..RADIX-1]所指各子表依次链接成一个链表，
  // e[0..RADIX-1]为各子表的尾指针。
  for ( j=0; !f[j]; j=succ(j)); // 找第一个非空子表，succ 为求后继函数
  r[0].next = f[j]; t = e[j];   // r[0].next 指向第一个非空子表中第一个结点
  while ( j<RADIX ) {
    for (j=succ(j); j<RADIX-1 && !f[j]; j=succ(j) );      // 找下一个非空子表
    if ( f[j] ) {r[t].next = f[j]; t = e[j]; }            // 链接两个非空子表
  }
  r[t].next = 0;                 // t 指向最后一个非空子表中的最后一个结点
}// Collect
```

算法 10.16

```
void RadixSort(SLList &L) {
  // L 是采用静态链表表示的顺序表。
  // 对 L 作基数排序，使得 L 成为按关键字自小到大的有序静态链表，L.r[0]为头结点。
  for (i=0; i<L.recnum; ++i) L.r[i].next = i+1;
  L.r[L.recnum].next = 0;          // 将 L 改造为静态链表
  for (i=0; i<L.keynum; ++i) {     // 按最低位优先依次对各关键字进行分配和收集
    Distribute(L.r, i, f, e);      // 第 i 趟分配
    Collect(L.r, i, f, e);         // 第 i 趟收集
  }
}// RadixSort
```

算法 10.17

10.7 各种内部排序方法的比较讨论

综合比较本章内讨论的各种内部排序方法，大致有如下结果(见下页表)。

从下表可以得出如下几个结论：

www.TopSage.com

大家网 TopSage.com

排序方法	平均时间	最坏情况	辅助存储
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n\log n)$	$O(n^2)$	$O(\log n)$
堆 排 序	$O(n\log n)$	$O(n\log n)$	$O(1)$
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n)$
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(rd)$

(1) 从平均时间性能而言，快速排序最佳，其所需时间最省，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。而后两者相比较的结果是，在 n 较大时，归并排序所需时间较堆排序省，但它所需的辅助存储量最多。

(2) 上表中的“简单排序”包括除希尔排序之外的所有插入排序，起泡排序和简单选择排序，其中以直接插入排序为最简单，当序列中的记录“基本有序”或 n 值较小时，它是最佳的排序方法，因此常将它和其他的排序方法，诸如快速排序、归并排序等结合在一起使用。

(3) 基数排序的时间复杂度也可写成 $O(d \cdot n)$。因此，它最适用于 n 值很大而关键字较小的序列。若关键字也很大，而序列中大多数记录的“最高位关键字”均不同，则亦可先按“最高位关键字”不同将序列分成若干“小”的子序列，而后进行直接插入排序。

(4) 从方法的稳定性来比较，基数排序是稳定的内排方法，所有时间复杂度为 $O(n^2)$ 的简单排序法也是稳定的，然而，快速排序、堆排序和希尔排序等时间性能较好的排序方法都是不稳定的。一般来说，排序过程中的“比较”是在“相邻的两个记录关键字”间进行的排序方法是稳定的。值得提出的是，稳定性是由方法本身决定的，对不稳定的排序方法而言，不管其描述形式如何，总能举出一个说明不稳定的实例来。反之，对稳定的排序方法，总能找到一种不引起不稳定的描述形式。由于大多数情况下排序是按记录的主关键字进行的，则所用的排序方法是否稳定无关紧要。若排序按记录的次关键字进行，则应根据问题所需慎重选择排序方法及其描述算法。

综上所述，在本章讨论的所有排序方法中，没有哪一种是绝对最优的。有的适用于 n 较大的情况，有的适用于 n 较小的情况，有的……因此，在实用时需根据不同情况适当选用，甚至可将多种方法结合起来使用。

本章讨论的多数排序算法是在顺序存储结构上实现的，因此在排序过程中需进行大量记录的移动。当记录很大(即每个记录所占空间较多)时，时间耗费很大，此时可采用静态链表作存储结构。如表插入排序、链式基数排序，以修改指针代替移动记录。但是，有的排序方法，如快速排序和堆排序，无法实现表排序。在这种情况下可以进行“地址排序”，即另设一个地址向量指示相应记录；同时在排序过程中不移动记录而移动地址向量中相应分量的内容。例如对图 10.15(a)所示记录序列进行地址排序时，可附设向量 adr(1：8)。在开始排序之前令 adr[i]：=i，凡在排序过程中需进行 r[i]：=r[j]的操作时，均以 adr[i]：=adr[j]代替，则在排序结束之后，地址向量中的值指示排序后的记录的次序，r[adr[1]]为关键字最小的记录，r[adr[8]]为关键字最大的记录，如图 10.15(b)所

www.TopSage.com

示。最后在需要时可根据 adr 的值重排记录的物理位置。重排算法如下：

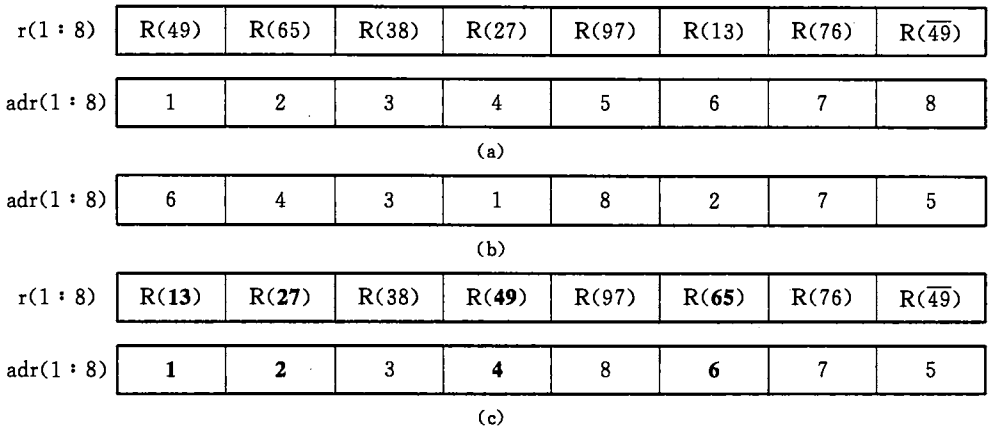


图 10.15　地址排序示例

(a) 待排记录和地址向量的初始状态；(b) 排序结束后的地址向量；(c) 重排记录过程中的状态

从 $i=1$ 起依次检查每个分量位置上的记录是否正确到位。若 adr[i]=i，则 r[i]中恰为第 i 个最小关键字的记录，该位置上的记录不需要调整；若 adr[i]=k≠i，则说明 r[k]中记录是第 i 个最小关键字的记录，应在暂存记录 r[i]之后将 r[k]中记录移至 r[i]的位置[①]上。类似地，若 adr[k]≠k，则应将 r[adr[k]]中记录移至 r[k]的位置上。依次类推，直至找到某个值 j=adr[adr[…adr[k]…]]，等式 adr[j]=i 成立时，将暂存记录移至 r[j]的位置上。至此完成一个调整记录位置的小循环。例如图 10.15 的例子，由于图 10.15(b)中 adr[1]=6，则在暂存 R(49)以后，需将 R(13)从 r[6]的位置移至 r[1]的位置。又，因为 adr[6]=2，则应将 R(65)从 r[2]的位置移至 r[6]的位置。同理，将 R(27)移至 r[2]的位置，此时，因 adr[4]=1，则 R(49)应置入 r[4]的位置上。完成上述调整后的记录及地址向量的状态如图 10.15(c)所示。算法 10.18 即为上述重排记录的算法。

```
void Rearrange ( SqList &L, int adr[] ) {
  // adr 给出顺序表 L 的有序次序，即 L.r[adr[i]]是第 i 小的记录。
  // 本算法按 adr 重排 L.r，使其有序。
  for ( i = 1; i<L.length; ++i )
    if (adr[i] != i) {
      j = i;   L.r[0] = L.r[i];        // 暂存记录 L.r[i]
      while (adr[j] != i ) {           // 调整 L.r[adr[j]]的记录到位直到 adr[j] = i 为止
        k = adr[j];   L.r[j] = L.r[k];
        adr[j] = j;   j = k;
      }
      L.r[j] = L.r[0];   adr[j] = j;   // 记录按序到位
    }
}// Rearrange
```

算法　10.18

从上述算法容易看出，除了在每个小循环中要暂存一次记录外，所有记录均一次移动

① r[i]的位置指的是 r 数组中第 i 个分量，下同。

www.TopSage.com

大家网 TopSage.com

到位。而每个小循环至少移动两个记录，则这样的小循环至多有$\lfloor n/2 \rfloor$个，所以重排记录的算法中至多移动记录$\lfloor 3n/2 \rfloor$次。

本节最后要讨论的一个问题是，“内部排序可能达到的最快速度是什么”。我们已经看到，本章讨论的各种排序方法，其最坏情况下的时间复杂度或为$O(n^2)$，或为$O(n\log n)$，其中$O(n^2)$是它的上界，那么$O(n\log n)$是否是它的下界，也就是说，能否找到一种排序方法，它在最坏情况下的时间复杂度低于$O(n\log n)$呢？

由于本章讨论的各种排序方法，除基数排序之外，都是基于“关键字间的比较”这个操作进行的，则均可用一棵类似于图 10.16 所示的判定树来描述这类排序方法的过程。

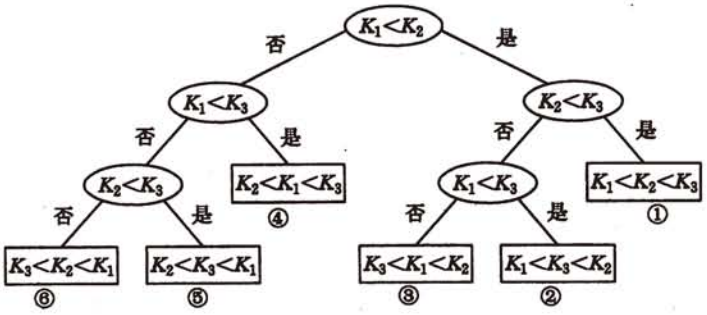


图 10.16　描述排序过程的判定树

图 10.16 的判定树表示 3 个关键字分别为K_1、K_2和K_3的记录进行直接插入排序的过程，树中每个非终端结点表示两个关键字间的一次比较，其左、右子树分别表示这次比较所得的两种结果。假设$K_1 \neq K_2 \neq K_3 \neq K_1$，则排序之前依次排列的这 3 个记录$\{R_1, R_2, R_3\}$之间只可能有下列 6 种关系：(1)$K_1<K_2<K_3$；(2)$K_1<K_3<K_2$；(3) $K_3<K_1<K_2$；(4) $K_2<K_1<K_3$；(5) $K_2<K_3<K_1$；(6) $K_3<K_2<K_1$，换句话说，这 3 个记录经过排序只可能得到下列 6 种结果：(1)$\{R_1, R_2, R_3\}$；(2)$\{R_1, R_3, R_2\}$；(3)$\{R_3, R_1, R_2\}$；(4)$\{R_2, R_1, R_3\}$；(5)$\{R_2, R_3, R_1\}$；(6)$\{R_3, R_2, R_1\}$，而图 10.16 中的判定树上 6 个终端结点恰好表示这 6 种排序结果。判定树上进行的每一次比较都是必要的，因此，这个判定树足以描述通过“比较”进行的排序过程。并且，对每一个初始序列经排序达到有序所需进行的“比较”次数，恰为从树根到和该序列相应的叶子结点的路径长度。由于图 10.16 的判定树的深度为 4，则对 3 个记录进行排序至少要进行 3 次比较。

推广至一般情况，对n个记录进行排序至少需进行多少次关键字间的比较，这个问题等价于，给定n个不同的砝码和一台天平，按重量的大小顺序排列这些砝码所需要的最少称重量次数问题。由于含n个记录的序列可能出现的初始状态有$n!$个，则描述n个记录排序过程的判定树必须有$n!$个叶子结点。因为，若少一个叶子，则说明尚有两种状态没有分辨出来。我们已经知道，若二叉树的高度为h，则叶子结点的个数不超过2^{h-1}；反之，若有u个叶子结点，则二叉树的高度至少为$\lceil \log_2 u \rceil + 1$。这就是说，描述$n$个记录排序的判定树上必定存在一条长度为$\lceil \log_2 (n!) \rceil$的路径。由此得到下述结论：任何一个借助“比较”进行排序的算法，在最坏情况下所需进行的比较次数至少为$\lceil \log_2 (n!) \rceil$。然而，这只是一个理论上的下界，一般的排序算法在$n>4$时所需进行的比较次数均大于此值，直到

www.TopSage.com

1956年，H. B. Demuth首先找到了对5个数进行排序只需要7次比较的方法[2]之后，Lester Ford和Selmer Johnson将其推广，提出了归并插入①(Merge Insertion)排序，在$n<11$时所用的比较次数和$\lceil \log_2(n!) \rceil$相同。②。根据斯特林公式，有$\lceil \log_2(n!) \rceil = O(n\log n)$，上述结论从数量级上告诉我们，借助于“比较”进行排序的算法在最坏情况下能达到的最好的时间复杂度为$O(n\log n)$。

① 归并插入排序的过程请参见《题集》第10章中最后一题。

② 下表中$B(n)$、$M(n)$和$F(n)$分别表示对n个数进行折半插入排序、归并排序和归并插入排序时在最坏情况下所需进行的比较次数[2]：

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\lceil \log_2 n! \rceil$	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45	49	53
$F(n)$	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	46	50	54
$B(n)$	0	1	3	5	8	11	14	17	21	25	29	33	37	41	45	49	54	59
$M(n)$	0	1	3	5	9	11	14	17	25	27	30	33	38	41	45	49	65	67

www.TopSage.com

大家网
TopSage.com

第11章 外部排序

上一章中已提到，外部排序指的是大文件的排序，即待排序的记录存储在外存储器上，在排序过程中需进行多次的内、外存之间的交换。因此，在本章讨论外部排序之前，首先需要了解对外存信息进行存取的特点。

11.1 外存信息的存取

计算机一般有两种存储器：内存储器（主存）和外存储器（辅存）。内存的信息可随机存取，且存取速度快，但价格贵、容量小。外存储器包括磁带和磁盘（或磁鼓），前者为顺序存取的设备，后者为随机存取的设备。

1. 磁带信息的存取

磁带是薄薄涂上一层磁性材料的一条窄带。现在使用的磁带大多数有1/2英寸宽，最长可达3 600英尺，绕在一个卷盘上。使用时，将磁带盘放在磁带机上，驱动器控制磁带盘转动，带动磁带向前移动。通过读/写头就可以读出磁带上的信息或者把信息写入磁带中（图11.1）。

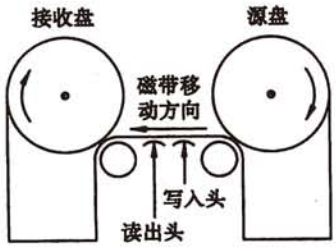


图11.1 磁带运动示意图

在1/2英寸宽的带面上可以记录9位或7位二进制信息（通常称为9道带或7道带）。以9道带为例，每一横排就可表示一个字符（8位表示一个字符，另一位作奇偶校验位）。因此，磁带上可记下各种文字信息或二进制信息。在磁带上信息按字符组[①]存放，而不是按字符存放。

磁带上信息的密度通常为每英寸800位或1 600位或6 250位（即每英寸的二进制字符数），移动速度是每秒200英寸。

磁带不是连续运转的设备，而是一种启停设备（启停时间约为5毫秒），它可以根据读/写的需要随时启动和停止。由于读/写信息应在旋转稳定时进行，而磁带从静止状态启动后，要经过一个加速的过程才能达到稳定状态；反之，在读/写结束后，从运动状态到完全停止，要经过一个减速的过程。因此，在磁带上相邻两组字符组（记录）之间要留一空白区，叫做**间隙** IRG(Inter Record Gap)。根据启停时间的需要，这个间隙通常为1/4～3/4英寸。如果每个字符组的长度是80个字符，IRG为3/4英寸，则对密度为每英寸1 600个字符的磁带，其利用率仅为1/16，有15/16的带用于IRG（参见图11.2(a)）。

为了有效地利用磁带，常常用组成块的办法来减少IRG的个数。在每次写信息时，不是按用户给出的字符组记入磁带，而是将若干个字符组合并成一块后一次写入磁带。于是，每个字符组间就没有IRG，而变成块间的间隙 IBG(Inter Block Gap)。图11.2(b)

① 字符组在操作系统的一些描述中称为“记录”。注意：操作系统中的“记录”不同于本书前面定义的记录。

www.TopSage.com

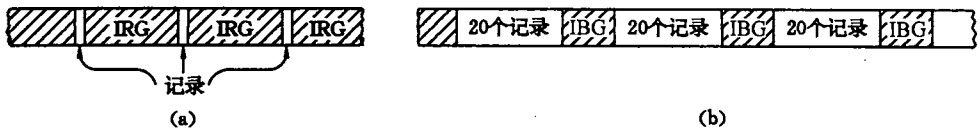


图 11.2 磁带上信息存放示意图

(a) 字符组长 80 字符的磁带； (b) 成块存放的磁带

表示将 20 个长度为 80 字符的字符组存放在磁带上的一个物理块中的情况。

成块的办法可以减少 IRG 的数目，从而可以提高磁带的利用率，块的长度大于 IBG 的长度。

成块还可减少 I/O 操作。因为一次 I/O 操作可把整个物理块都读到内存缓冲区中，然后再从缓冲区中取出所需要的信息(一个字符组)。每当要读一个字符组时，首先要查缓冲区中是否已有，若有，则不必执行 I/O 操作，直接从缓冲区读取即可。

软件要有处理成块、解块和保存字符组的功能。在使用者看来，每次读/写的却只是一个字符组[11]。

是否物理块越大，数据越紧凑，效率就越高呢？实际上不是这样的。物理块不能太大，通常只有 1K 字节～8K 字节。这是因为如果一次读写太长，则出错的概率就增大，可靠性就降低；此外，若块太大，则在内存开辟的缓冲区就大，从而耗费内存空间也多。

在磁带上读写一块信息所需的时间由两部分组成：

$$T_{I/O} = t_a + n \cdot t_w$$

其中：t_a 为延迟时间，读/写头到达传输信息所在物理块起始位置所需时间；t_w 为传输一个字符的时间。

显然，延迟时间和信息在磁带上的位置、当前读/写头所在位置有关。例如，若读/写头在第 i 和第 $i+1$ 个物理块之间的间隙上，则读第 $i+1$ 个物理块上的信息仅需几毫秒；若读/写头位于磁带的始端，而要读的信息在磁带的尾端，则必须使磁带向前运动，跳过中间的许多块，直到所需信息通过读/写头时才能得到，这可能需要几分钟的时间。因此，由于磁带是顺序存取的设备，则读/写信息之前先要进行顺序查找，并且当读/写头位于磁带尾端，而要读的信息在磁带始端时，尚需使磁带倒转运动。这是顺序存取设备的主要缺点，它使检索和修改信息很不方便。因此，顺序存取设备主要用于处理变化少、只进行顺序存取的大量数据。

2. 磁盘信息的存取

磁盘是一种直接存取的存储设备(DASD)。它是以存取时间变化不大为特征的。它不像磁带那样只能进行顺序存取，而可以直接存取任何字符组。它的容量大、速度快，存取速度比磁带快得多。磁盘是一个扁平的圆盘(与电唱机的唱片类似)，盘面上有许多称为磁道的圆圈，信息就记载在磁道上。由于磁道的圆圈为许多同心圆，所以可以直接存取。磁盘可以是单片的，也可以由若干盘片组成盘组。每一片上有两个面。以 6 片盘组为例，由于最顶上和最低下盘片的外侧面不存信息，所以总共只有 10 个面可用来保存信息，如图 11.3 所示。

磁盘驱动器执行读/写信息的功能。盘片装在一个主轴上，并绕主轴高速旋转，当磁

www.TopSage.com

大家网 TopSage.com

道在读/写头下通过时，便可以进行信息的读/写。

可以把磁盘分为固定头盘和活动头盘。固定头盘的每一道上都有独立的磁头，它是固定不动的，专负责读/写某一道上的信息。

活动头盘的磁头是可移动的。盘组也是可变的。一个面上只有一个磁头，它可以从该面上的一道移动到另一道。磁头装在一个动臂上，不同面上的磁头是同时移动的，并处于同一圆柱面上。各个面上半径相同的磁道组成一个圆柱面，圆柱面的个数就是盘片面上的磁道数。通常，每个面上有 200～400 道。在磁盘上标明一个具体信息必须用一个三维地址：柱面号、盘面号、块号。

其中，柱面号确定读/写头的径向运动，而块号确定信息在盘片圆圈上的位置。

为了访问一块信息，首先必须找柱面，移动臂使磁头移动到所需柱面上（称为定位或寻查）；然后等待要访问的信息转到磁头之下；最后，读/写所需信息。

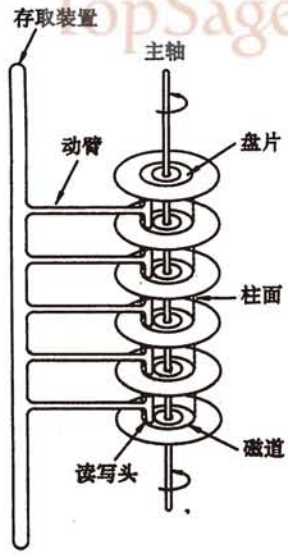


图 11.3　活动头盘示意图

所以，在磁盘上读写一块信息所需的时间由 3 部分组成：

$$T_{I/O}=t_{seek}+t_{la}+n\cdot t_{wm}$$

其中：t_{seek} 为**寻查时间**（seek time），即读/写头定位的时间；

t_{la} 为**等待时间**（latency time），即等待信息块的初始位置旋转到读写头下的时间；

t_{wm} 为**传输时间**（transmission time）。

由于磁盘的旋转速度很快，约 2 400～3 600 转/分，则等待时间最长不超过 25 毫秒（旋转一圈的时间），磁盘的传输速率一般在 10^5 字符/秒和 5×10^5 字符/秒之间，则在磁盘上读/写信息的时间主要花在寻查时间上（其最大寻查时间约为 0.1 秒）。因此，在磁盘上存放信息时应将相关的信息放在同一柱面或邻近柱面上，以求在读/写信息时尽量减少磁头来回移动的次数，以避免不必要的寻查时间。

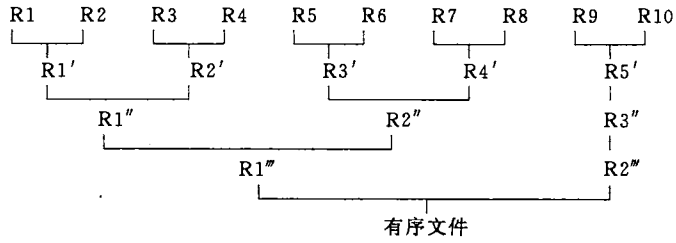
11.2　外部排序的方法

外部排序基本上由两个相对独立的阶段组成。首先，按可用内存大小，将外存上含 n 个记录的文件分成若干长度为 l 的子文件或**段**（segment），依次读入内存并利用有效的内部排序方法对它们进行排序，并将排序后得到的有序子文件重新写入外存，通常称这些有序子文件为**归并段**或**顺串**（run）；然后，对这些归并段进行逐趟归并，使归并段（有序的子文件）逐渐由小至大，直至得到整个有序文件为止。显然，第一阶段的工作是上一章已经讨论过的内容。本章主要讨论第二阶段即归并的过程。先从一个具体例子来看外排中的归并是如何进行的？

假设有一个含 10 000 个记录的文件，首先通过 10 次内部排序得到 10 个初始归并段 R1～R10，其中每一段都含 1 000 个记录。然后对它们作如下图所示的两两归并，直至得

www.TopSage.com

到一个有序文件为止。



从上图可见，由 10 个初始归并段到一个有序文件，共进行了 4 趟归并，每一趟从 m 个归并段得到$\lceil m/2 \rceil$个归并段。这种归并方法称为 2-路平衡归并。

将两个有序段归并成一个有序段的过程，若在内存进行，则很简单，上一章中的 merge 过程便可实现此归并。但是，在外部排序中实现两两归并时，不仅要调用 merge 过程，而且要进行外存的读/写，这是由于我们不可能将两个有序段及归并结果段同时存放在内存中的缘故。在 11.1 节中已经提到，对外存上信息的读/写是以“物理块”为单位的。假设在上例中每个物理块可以容纳 200 个记录，则每一趟归并需进行 50 次“读”和 50 次“写”，4 趟归并加上内部排序时所需进行的读/写使得在外排中总共需进行 500 次的读/写。

一般情况下，

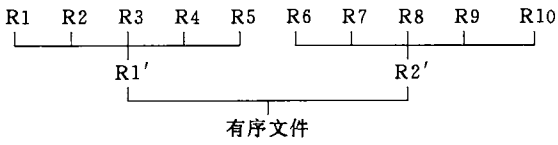
外部排序所需总的时间＝内部排序(产生初始归并段)所需的时间($m\times t_{IS}$)＋
外存信息读写的时间($d\times t_{IO}$)＋
内部归并所需的时间($s\times ut_{mg}$)　　(11-1)

其中：t_{IS}是为得到一个初始归并段进行内部排序所需时间的均值；t_{IO}是进行一次外存读/写时间的均值；ut_{mg}是对 u 个记录进行内部归并所需时间；m 为经过内部排序之后得到的初始归并段的个数；s 为归并的趟数；d 为总的读/写次数。由此，上例 10 000 个记录利用 2-路归并进行外排所需总的时间为：

$$10\times t_{IS}+500\times t_{IO}+4\times 10\ 000t_{mg}$$

其中 t_{IO}取决于所用的外存设备，显然，t_{IO}较 t_{mg}要大得多。因此，提高外排的效率应主要着眼于减少外存信息读写的次数 d。

下面来分析 d 和“归并过程”的关系。若对上例中所得的 10 个初始归并段进行 5-路平衡归并(即每一趟将 5 个或 5 个以下的有序子文件归并成一个有序子文件)，则从下图可见，仅需进行二趟归并，外排时总的读/写次数便减至 2×100＋100＝300，比 2-路归并减少了 200 次的读/写。



可见，对同一文件而言，进行外排时所需读/写外存的次数和归并的趟数 s 成正比。

www.TopSage.com

大家网 TopSage.com

而在一般情况下，对 m 个初始归并段进行 k-路平衡归并时，归并的趟数

$$s = \lfloor \log_k m \rfloor \tag{11-2}$$

可见，若增加 k 或减少 m 便能减少 s。下面分别就这两个方面讨论之。

11.3 多路平衡归并的实现

从式(11-2)得知，增加 k 可以减少 s，从而减少外存读/写的次数。但是，从下面的讨论中又可发现，单纯增加 k 将导致增加内部归并的时间 ut_{mg}。那么，如何解决这个矛盾呢?

先看 2-路归并。令 u 个记录分布在两个归并段上，按 merge 过程进行归并。每得到归并后的一个记录，仅需一次比较即可，则得到含 u 个记录的归并段需进行 $u-1$ 次比较。

再看 k-路归并。令 u 个记录分布在 k 个归并段上，显然，归并后的第一个记录应是 k 个归并段中关键字最小的记录，即应从每个归并段的第一个记录的相互比较中选出最小者，这需要进行 $k-1$ 次比较。同理，每得到归并后的有序段中的一个记录，都要进行 $k-1$ 次比较。显然，为得到含 u 个记录的归并段需进行 $(u-1)(k-1)$ 次比较。由此，对 n 个记录的文件进行外排时，在内部归并过程中进行的总的比较次数为 $s(k-1)(n-1)$。假设所得初始归并段为 m 个，则由式(11-2)可得内部归并过程中进行比较的总的次数为

$$\lfloor \log_k m \rfloor (k-1)(n-1)t_{mg} = \left\lfloor \frac{\log_2 m}{\log_2 k} \right\rfloor (k-1)(n-1)t_{mg} \tag{11-3}$$

由于 $\frac{k-1}{\log_2 k}$ 随 k 的增长而增长，则内部归并时间亦随 k 的增长而增长。这将抵消由于增大 k 而减少外存信息读写时间所得效益，这是我们所不希望的。然而，若在进行 k-路归并时利用**“败者树”**(Tree of Loser)，则可使在 k 个记录中选出关键字最小的记录时仅需进行 $\lfloor \log_2 k \rfloor$ 次比较，从而使总的归并时间由式(11-3)变为 $\lfloor \log_2 m \rfloor (n-1)t_{mg}$，显然，这个式子和 k 无关，它不再随 k 的增长而增长。

那末，什么是“败者树”? 它是树形选择排序的一种变型。相对地，我们可称图 10.8 和图 10.9 中的二叉树为“胜者树”，因为每个非终端结点均表示其左、右孩子结点中的“胜者”。反之，若在双亲结点中记下刚进行完的这场比赛中的败者，而让胜者去参加更高一层的比赛，便可得到一棵**“败者树”**。例如，图 11.4(a)所示为一棵实现 5-路归并的败者树 ls[0..4]，图中方形结点表示叶子结点(也可看成是外结点)，分别为 5 个归并段中当前参加归并选择的记录的关键字；败者树中根结点 ls[1]的双亲结点 ls[0]为“冠军”，在此指示各归并段中的最小关键字记录为第三段中的当前记录；结点 ls[3]指示 b1 和 b2 两个叶子结点中的败者即 b2，而胜者 b1 和 b3(b3 是叶子结点 b3、b4 和 b0 经过两场比赛后选出的获胜者)进行比较，结点 ls[1]则指示它们中的败者为 b1。在选得最小关键字的记录之后，只要修改叶子结点 b3 中的值，使其为同一归并段中的下一个记录的关键字，然后从该结点向上和双亲结点所指的关键字进行比较，败者留在该双亲结点，胜者继续向上直至树根的双亲。如图 11.4(b)所示，当第 3 个归并段中第 2 个记录参加归并时，选得的最小关键字记录为第一个归并段中的记录。为了防止在归并过程中某个归并段变空，可以在每个归并段中附加一个关键字为最大值的记录。当选出的“冠军”记录的关键字为最大值

www.TopSage.com

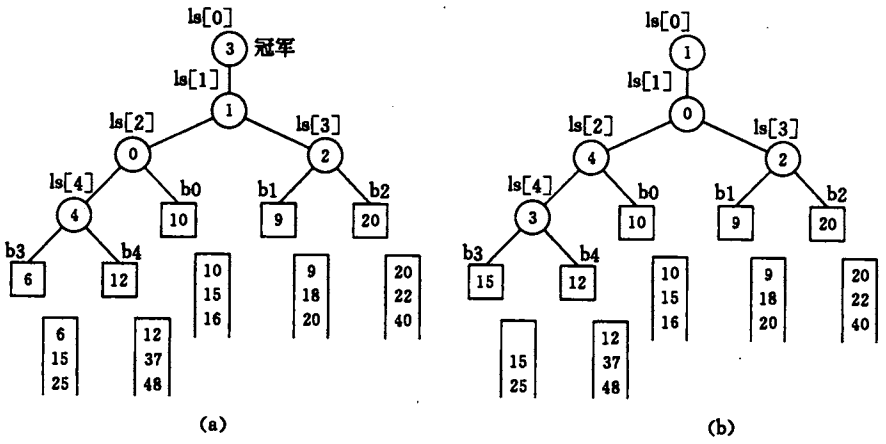


图 11.4　实现 5-路归并的败者树

时，表明此次归并已完成。由于实现 k-路归并的败者树的深度为$\lceil \log_2 k \rceil + 1$，则在 k 个记录中选择最小关键字仅需进行$\lceil \log_2 k \rceil$次比较。败者树的初始化也容易实现，只要先令所有的非终端结点指向一个含最小关键字的叶子结点，然后从各个叶子结点出发调整非终端结点为新的败者即可。

下面的算法 11.1 简单描述利用败者树进行 k-路归并的过程。为了突出如何利用败者树进行归并，在算法中避开了外存信息存取的细节，可以认为归并段已在内存。算法 11.2 描述在从败者树选得最小关键字的记录之后，如何从叶到根调整败者树选得下一个最小关键字。算法 11.3 为初建败者树的过程的算法描述。

```
typedef int LoserTree[k];          // 败者树是完全二叉树且不含叶子，可采用顺序存储结构
typedef struct {
   KeyType key;
}ExNode, External[k+1];            // 外结点，只存放待归并记录的关键字

void K_Merge (LoserTree &ls, External &b) {
   // 利用败者树 ls 将编号从 0 到 k-1 的 k 个输入归并段中的记录归并到输出归并段。
   // b[0]至 b[k-1]为败者树上的 k 个叶子结点，分别存放 k 个输入归并段中当前记录的关键字。
   for (i=0; i<k; ++i)  input(b[i].key);          // 分别从 k 个输入归并段读入该段当前
                                                  // 第一个记录的关键字到外结点
   CreateLoserTree(ls);            // 建败者树 ls，选得最小关键字为 b[ls[0]].key
   while (b[ls[0]].key != MAXKEY) {
      q = ls[0];                   // q 指示当前最小关键字所在归并段
      output (q);                  // 将编号为 q 的归并段中当前(关键字为 b[q].key)的记录
                                   // 写至输出归并段
      input (b[q].key,q);          // 从编号为 q 的输入归并段中读入下一个记录的关键字
      Adjust(ls, q);               // 调整败者树，选择新的最小关键字
   }// while
   output (ls[0]);                 // 将含最大关键字 MAXKEY 的记录写至输出归并段
}// K_Merge
```

算法　11.1

www.TopSage.com

大家网
TopSage.com

```
void Adjust (LoserTree &ls, int s) {
   // 沿从叶子结点 b[s]到根结点 ls[0]的路径调整败者树
   t = (s+k)/2;       // ls[t]是 b[s]的双亲结点
   while (t>0) {
      if (b[s].key > b[ls[t]].key) s←→ls[t];      // s 指示新的胜者
      t = t/2;
   }
   ls[0] = s;
}// Adjust
```

算法 11.2

```
void CreateLoserTree(LoserTree &ls) {
   // 已知 b[0]到 b[k-1]为完全二叉树 ls 的叶子结点存有 k 个关键字,沿从叶子
   // 到根的 k 条路径将 ls 调整成为败者树。
   b[k].key = MINKEY;                          // 设 MINKEY 为关键字可能的最小值
   for (i=0; i<k; ++i)  ls[i] = k;             // 设置 ls 中"败者"的初值
   for (i=k-1; i>=0; --i) Adjust(ls, i);       // 依次从 b[k-1],b[k-2],...,b[0]出发调
                                               // 整败者
}// CreateLoserTree
```

算法 11.3

最后要提及一点,k 值的选择并非越大越好,如何选择合适的 k 是一个需要综合考虑的问题。

11.4 置换-选择排序

由 11-2 式得知,归并的趟数不仅和 k 成反比,也和 m 成正比,因此,减少 m 是减少 s 的另一条途径。然而,我们从 11.2 节的讨论中也得知,m 是外部文件经过内部排序之后得到的初始归并段的个数,显然,$m=\lceil n/l \rceil$,其中 n 为外部文件中的记录数,l 为初始归并段中的记录数。回顾上一章讨论的各种内排方法,在内排过程中移动记录和对关键字进行比较都是在内存中进行的。因此,用这些方法进行内部排序得到的各个初始归并段的长度 l(除最后一段外)都相同,且完全依赖于进行内部排序时可用内存工作区的大小,则 m 也随其而限定。由此,若要减少 m,即增加 l,就必须探索新的排序方法。

置换-选择排序(Replacement-Selection Sorting)是在树形选择排序的基础上得来的,它的特点是:在整个排序(得到所有初始归并段)的过程中,选择最小(或最大)关键字和输入、输出交叉或平行进行。

先从具体例子谈起。已知初始文件含有 24 个记录,它们的关键字分别为 51,49,39,46,38,29,14,61,15,30,1,48,52,3,63,27,4,13,89,24,46,58,33,76。假设内存工作区可容纳 6 个记录,则按上章讨论的选择排序可求得如下 4 个初始归并段:

RUN1: 29,38,39,46,49,51

RUN2:1,14,15,30,48,61

www.TopSage.com

RUN3:3, 4,13,27,52,63

RUN4: 24,33,46,58,76,89

若按置换-选择进行排序，则可求得如下 3 个初始归并段：

RUN1: 29,38,39,46,49,51,61

RUN2:1, 3,14,15,27,30,48,52,63,89

RUN3:4,13,24,33,46,58,76

假设初始待排文件为输入文件 FI，初始归并段文件为输出文件 FO，内存工作区为 WA，FO 和 WA 的初始状态为空，并设内存工作区 WA 的容量可容纳 w 个记录，则置换-选择排序的操作过程为：

(1) 从 FI 输入 w 个记录到工作区 WA。

(2) 从 WA 中选出其中关键字取最小值的记录，记为 MINIMAX 记录。

(3) 将 MINIMAX 记录输出到 FO 中去。

(4) 若 FI 不空，则从 FI 输入下一个记录到 WA 中。

(5) 从 WA 中所有关键字比 MINIMAX 记录的关键字大的记录中选出最小关键字记录，作为新的 MINIMAX 记录。

(6) 重复(3)～(5)，直至在 WA 中选不出新的 MINIMAX 记录为止，由此得到一个初始归并段，输出一个归并段的结束标志到 FO 中去。

(7) 重复(2)～(6)，直至 WA 为空。由此得到全部初始归并段。

例如，以上所举之例的置换-选择过程如图 11.5 所示。

FO	WA	FI
空	空	51,49,39,46,38,29,14,61,15,30,1,48,52,3,63,27,4,…
空	51,49,39,46,38,29	14,61,15,30,1,48,52,3,63,27,4,…
29	51,49,39,46,38	14,61,15,30,1,48,52,3,63,27,4,…
29	51,49,39,46,38,14	61,15,30,1,48,52,3,63,27,4,…
29,38	51,49,39,46, ,14	61,15,30,1,48,52,3,63,27,4,…
29,38	51,49,39,46,61,14	15,30,1,48,52,3,63,27,4,…
29,38,39	51,49, ,46,61,14	15,30,1,48,52,3,63,27,4,…
29,38,39	51,49,15,46,61,14	30,1,48,52,3,63,27,4,…
29,38,39,46	51,49,15, ,61,14	30,1,48,52,3,63,27,4,…
29,38,39,46	51,49,15,30,61,14	1,48,52,3,63,27,4,…
29,38,39,46,49	51, ,15,30,61,14	1,48,52,3,63,27,4,…
29,38,39,46,49	51,1,15,30,61,14	48,52,3,63,27,4,…
29,38,39,46,49,51	,1,15,30,61,14	48,52,3,63,27,4,…
29,38,39,46,49,51	48,1,15,30,61,14	52,3,63,27,4,…
29,38,39,46,49,51,61	48,1,15,30, ,14	52,3,63,27,4,…

www.TopSage.com

大家网 TopSage.com

续表

FO	WA	FI
29,38,39,46,49,51,61	48,1,15,30,52,14	3,63,27,4,…
29,38,39,46,49,51,61,*	48,1,15,30,52,14	3,63,27,4,…
29,38,39,46,49,51,61,*,1	48, ,15,30,52,14	3,63,27,4,…
29,38,39,46,49,51,61,*,1	48,3,15,30,52,14	63,27,4,…
⋮	⋮	⋮

图 11.5 置换-选择排序过程示例

在 WA 中选择 MINIMAX 记录的过程需利用“败者树”来实现。关于“败者树”本身，上节已有详细讨论，在此仅就置换-选择排序中的实现细节加以说明。(1)内存工作区中的记录作为败者树的外部结点，而败者树中根结点的双亲结点指示工作区中关键字最小的记录；(2)为了便于选出 MINIMAX 记录，为每个记录附设一个所在归并段的序号，在进行关键字的比较时，先比较段号，段号小的为胜者；段号相同的则关键字小的为胜者；(3)败者树的建立可从设工作区中所有记录的段号均为“零”开始，然后从 FI 逐个输入 w 个记录到工作区时，自下而上调整败者树，由于这些记录的段号为“1”，则它们对于“零”段的记录而言均为败者，从而逐个填充到败者树的各结点中去。算法 11.4 是置换-选择排序的简单描述，其中，求得一个初始归并段的过程如算法 11.5 所述。算法 11.6 和算法 11.7 分别描述了置换-选择排序中的败者树的调整和初建的过程。

```
typedef struct {
  RcdType  rec;                    // 记录
  KeyType  key;                    // 从记录中抽取的关键字
  int      rnum;                   // 所属归并段的段号
}RcdNode, WorkArea[w];             // 内存工作区,容量为 w

void Replace_Selection (LoserTree &ls, WorkArea &wa, FILE * fi, FILE * fo) {
  // 在败者树 ls 和内存工作区 wa 上用置换-选择排序求初始归并段,fi 为输入文件
  // (只读文件)指针,fo 为输出文件(只写文件)指针,两个文件均已打开
  Construct_Loser (ls, wa);        // 初建败者树
  rc = rmax = 1;                   // rc 指示当前生成的初始归并段的段号,
                                   // rmax 指示 wa 中关键字所属初始归并段的最大段号
  while (rc <= rmax) {             // "rc = rmax + 1"标志输入文件的置换-选择排序已完成
    get_run (ls, wa);              // 求得一个初始归并段
    fwrite( &RUNEND_SYMBOL, sizeof(struct RcdType), 1, fo); // 将段结束标志写入输出文件
    rc = wa[ls[0]].rnum;           // 设置下一段的段号
  }
}// Replace_Selection
```

算法 11.4

```
void get_run (LoserTree &ls, WorkArea &wa) {
  // 求得一个初始归并段,fi 为输入文件指针,fo 为输出文件指针
  while (wa[ls[0]].rnum == rc) {           // 选得的 MINIMAX 记录属当前段时
    q = ls[0];                             // q 指示 MINIMAX 记录在 wa 中的位置
    minimax = wa[q].key;
```


www.TopSage.com

```
    fwrite (&wa[q].rec, sizeof( RcdType),1,fo);
                                        // 将刚选好的 MINIMAX 记录写入输出文件
    if (feof(fi)) {wa[q].rnum = rmax+1;  wa[q].key = MAXKEY }
                                        // 输入文件结束,虚设记录(属"rmax+1"段)
    else {                               // 输入文件非空时
        fread ( &wa[q].rec, sizeof(RcdTyp   1, fi );          // 从输入文件读入下一记录
        wa[q].key = wa[q].rec.key;      // 提取关键字
        if (wa[q].key < minimax) {      // 新读入的记录属下一段
          rmax = rc + 1;     wa[q].rnum = rmax;
        }
        else  wa[q].rnum = rc;          // 新读入的记录属当前段
    }
    Select_MiniMax (ls, wa, q);         // 选择新的 MINIMAX 记录
  }// while
}// get_run
```

算法 11.5

```
void Select_MiniMax (LoserTree &ls, WorkArea wa, int q) {
  // 从 wa[q]起到败者树的根比较选择 MINIMAX 记录,并由 q 指示它所在的归并段
  for ( t=(w+q)/2, p=ls[t];  t>0; t=t/2, p=ls[t])
    if ( wa[p].rnum<wa[q].rnum || wa[p].rnum==wa[q].rnum && wa[p].key<wa[q].key )
      q←→ls[t];       // q 指示新的胜利者
  ls[0] = q;
}// Select_MiniMax
```

算法 11.6

```
void Construct_Loser ( LoserTree &ls, WorkArea &wa ) {
  // 输入 w 个记录到内存工作区 wa,建得败者树 ls,选出关键字最小的记录并由 s 指示
  // 其在 wa 中的位置
  for (i=0; i<w; ++i)
    wa[i].rnum = wa[i].key = ls[i] = 0;                  // 工作区初始化
  for (i=w-1; i>=0; --i) {
    fread ( &wa[i].rec, sizeof( RcdType), 1, fi );       // 输入一个记录
    wa[i].key = wa[i].rec.key;                           // 提取关键字
    wa[i].rnum = 1;                                      // 其段号为"1"
    Select_MiniMax (ls, wa, i);                          // 调整败者
  }
}// Construct_Loser
```

算法 11.7

利用败者树对前面例子进行置换-选择排序时的局部状况如图 11.6 所示,其中图 11.6(a)～(g)显示了败者树建立过程中的状态变化状况。最后得到最小关键字的记录为 wa[0],之后,输出 wa[0].rec,并从 FI 中输入下一个记录至 wa[0],由于它的关键字小于刚刚输出的记录的关键字,则设此新输入的记录的段号为 2(如图 11.6(h)所示),而由于在输出 wa[1]之后新输入的关键字较 wa[1].key 大,则该新输入的记录的段号仍为 1(如图 11.6(i)所示)。图 11.6(j)所示为在输出 6 个记录之后选得的 MINIMAX 记录为

www.TopSage.com

大家网
TopSage.com

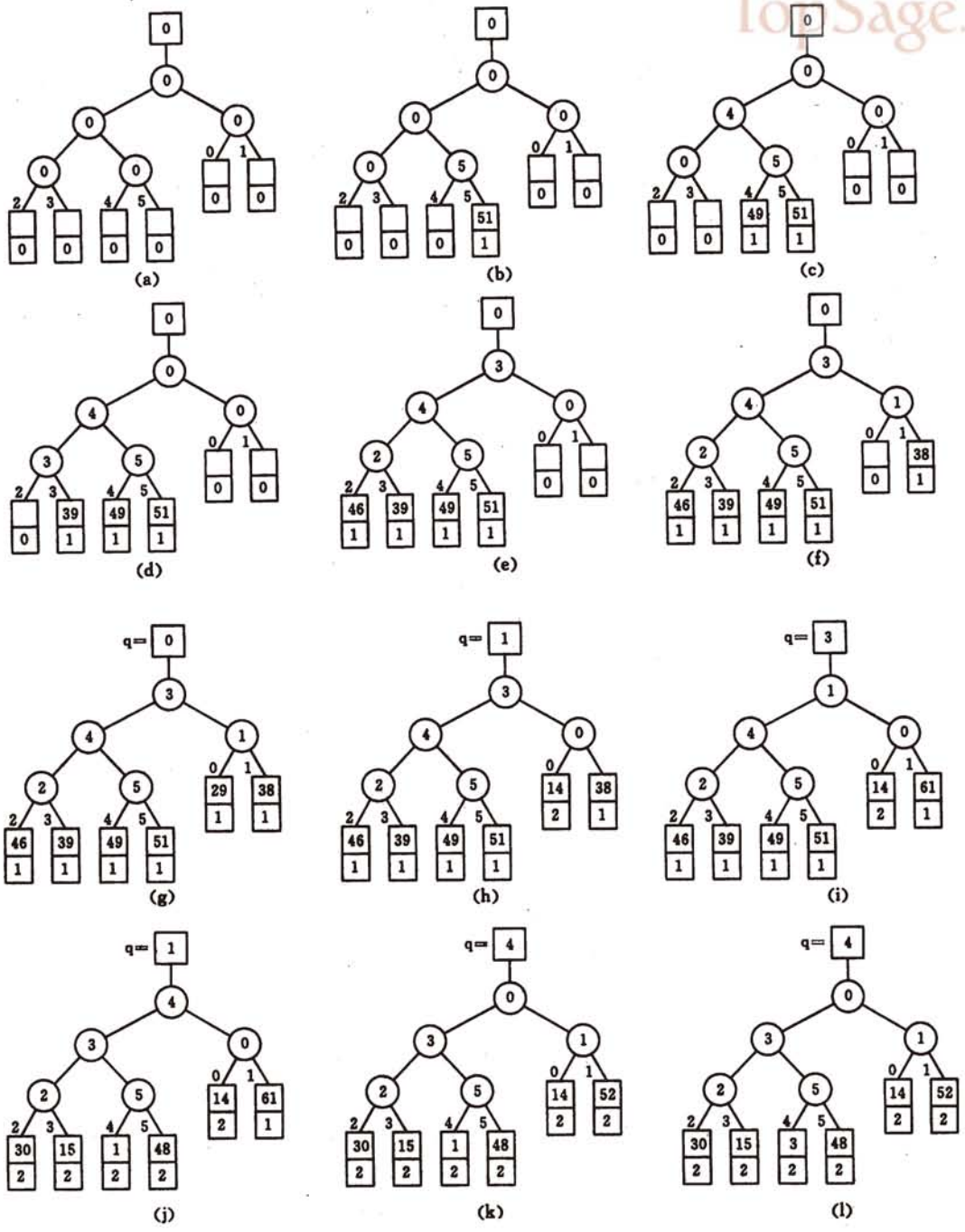


图 11.6　置换～选择过程中的败者树

(a)～(g)建立败者树，选出最小关键字记录 wa[0]；　(h)～(l)选好新的 MINIMAX 记录

wa[1]时的败者树。图 11.6(k)表明在输出该记录 wa[1]之后，由于输入的下一个记录的关键字较小，其段号亦为 2，致使工作区中所有记录的段号均为 2。由此败者树选出的新

www.TopSage.com

的 MINIMAX 记录的段号大于当前生成的归并段的序号，这说明该段已结束，而此新的 MINIMAX 记录应是下一归并段中的第一个记录。

从上述可见，由置换-选择排序所得初始归并段的长度不等。且可证明，当输入文件中记录的关键字为随机数时，所得初始归并段的平均长度为内存工作区大小 w 的两倍。这个证明是 E. F. Moore 在 1961 年从置换-选择排序和扫雪机的类比中得出的。

假设一台扫雪机在环形路上等速行进扫雪，又下雪的速度也是均匀的(即每小时落到地面上的雪量相等)，雪均匀地落在扫雪机的前、后路面上，边下雪边扫雪。显然，在某个时刻之后，整个系统达到平衡状态，路面上的积雪总量不变。且在任何时刻，整个路面上的积雪都形成一个均匀的斜面，紧靠扫雪机前端的积雪最厚，其深度为 h，而在扫雪机刚扫过的路面上的积雪深度为零。若将环形路伸展开来，路面积雪状态如图 11.7 所示。假设此刻路面积雪的总体积为 w，环形路一圈的长度为 l，由于扫雪机在任何时刻扫走的雪的深度均为 h，则扫雪机在环形路上走一圈扫掉的积雪体积为 lh 即 $2w$。

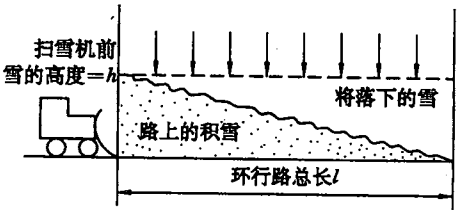


图 11.7　环形路上扫雪机系统平衡时的状态

将置换-选择排序与此类比，工作区中的记录好比路面的积雪，输出的 MINIMAX 记录好比扫走的雪，新输入的记录好比新下的雪，当关键字为随机数时，新记录的关键字比 MINIMAX 大或小的概率相等。若大，则属当前的归并段(好比落在扫雪机前面的积雪，在这一圈中将被扫走)；若小，则属下一归并段(好比落在扫雪机后面的积雪，在下一圈中才能扫走)。由此，得到一个初始归并段好比扫雪机走一圈。假设工作区的容量为 w，则置换-选择所得初始归并段长度的期望值便为 $2w$。

容易看出，若不计输入、输出的时间，则对 n 个记录的文件而言，生成所有初始归并段所需时间为 $O(n\log w)$。

11.5　最佳归并树

这一节要讨论的问题是，由置换-选择生成所得的初始归并段，其各段长度不等对平衡归并有何影响？

假设由置换-选择得到 9 个初始归并段，其长度(即记录数)依次为：9，30，12，18，3，17，2，6，24。现作 3-路平衡归并，其归并树(表示归并过程的图)如图 11.8 所示，图中每个圆圈表示一个初始归并段，圆圈中数字表示归并段的长度。假设每个记录占一个物理块，则两趟归并所需对外存进行的读/写次数为

$$(9+30+12+18+3+17+2+6+24)\times 2\times 2=484$$

若将初始归并段的长度看成是归并树中叶子结点的权，则此三叉树的带权路径长度的两

www.TopSage.com

大家网
TopSage.com

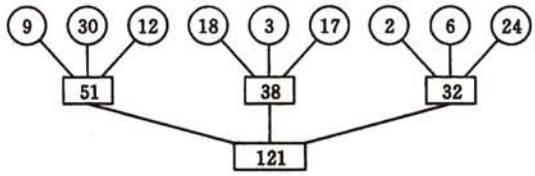


图 11.8　3-路平衡归并的归并树

倍恰为 484。显然，归并方案不同，所得归并树亦不同，树的带权路径长度（或外存读/写次数）亦不同。回顾在第 6 章中曾讨论了有 n 个叶子结点的带权路径长度最短的二叉树称赫夫曼树，同理，存在有 n 个叶子结点的带权路径长度最短的 3 叉、4 叉、…、k 叉树，亦称为赫夫曼树。因此，若对长度不等的 m 个初始归并段，构造一棵赫夫曼树作为归并树，便可使在进行外部归并时所需对外存进行的读/写次数达最少。例如，对上述 9 个初始归并段可构造一棵如图 11.9 所示的归并树，按此树进行归并，仅需对外存进行 446 次读/写，这棵归并树便称做最佳归并树。

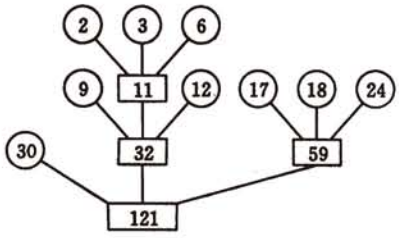


图 11.9　3-路平衡归并的最佳归并树

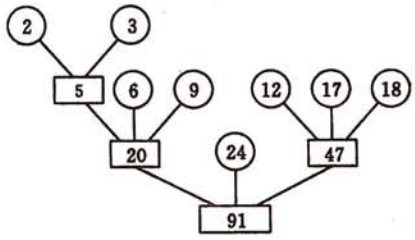


图 11.10　8 个归并段的最佳归并树

图 11.9 的赫夫曼树是一棵真正的 3 叉树，即树中只有度为 3 或 0 的结点。假若只有 8 个初始归并段，例如，在前面例子中少了一个长度为 30 的归并段。如果在设计归并方案时，缺额的归并段留在最后，即除了最后一次作 2-路归并外，其他各次归并仍都是 3-路归并，容易看出此归并方案的外存读/写次数为 386。显然，这不是最佳方案。正确的做法是，当初始归并段的数目不足时，需附加长度为零的“虚段”，按照赫夫曼树构成的原则，权为零的叶子应离树根最远，因此，这个只有 8 个初始归并段的归并树应如图 11.10 所示。

那么，如何判定附加虚段的数目？当 3 叉树中只有度为 3 和 0 的结点时，必有 $n_3=(n_0-1)/2$，其中，n_3 是度为 3 的结点数，n_0 是度为零的结点数。由于 n_3 必为整数，则 (n_0-1) MOD $2=0$。这就是说，对 3-路归并而言，只有当初始归并段的个数为偶数时，才需加 1 个虚段。

在一般情况下，对 k-路归并而言，容易推算得到，若 $(m-1)$ MOD $(k-1)=0$，则不需要加虚段，否则需附加 $k-(m-1)$ MOD $(k-1)-1$ 个虚段。换句话说，第一次归并为 $(m-1)$ MOD $(k-1)+1$ 路归并。

若按最佳归并树的归并方案进行磁盘归并排序，需在内存建立一张载有归并段的长度和它在磁盘上的物理位置的索引表。

www.TopSage.com

第12章 文　件

和表类似，文件是大量记录的集合。习惯上称存储在主存储器（内存储器）中的记录集合为表，称存储在二级存储器（外存储器）中的记录集合为文件。本章讨论文件在外存储器中的表示方法及其各种运算的实现方法。

12.1 有关文件的基本概念

• 文件及其类别

文件(file)是由大量性质相同的记录组成的集合。可按其记录的类型不同而分成两类：操作系统的文件和数据库文件。

操作系统中的文件仅是一维的连续的字符序列，无结构、无解释。它也是记录的集合，这个记录仅是一个字符组，用户为了存取、加工方便，把文件中的信息划分成若干组，每一组信息称为一个逻辑记录，且可按顺序编号。

数据库中的文件是带有结构的记录的集合；这类记录是由一个或多个数据项组成的集合，它也是文件中可存取的数据的基本单位。数据项是最基本的不可分的数据单位，也是文件中可使用的数据的最小单位。例如，图 12.1 所示为一个数据库文件，每个学生的情况是一个记录，它由 10 个数据项组成。

姓名	准考证号	政治	语文	数学	外语	物理	化学	生物	总分
王鸣	1501	78	90	104	95	87	83	40	577
刘青	1502	64	88	90	74	90	98	41	545
张朋	1503	90	101	85	89	76	87	42	570
崔永	1504	85	73	90	91	85	77	35	536
郑琳	1505	75	75	81	78	67	80	37	493
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

图 12.1　高考成绩文件

文件还可按记录的另一特性分成定长记录文件和不定长记录文件。若文件中每个记录含有的信息长度相同，则称这类记录为定长记录，由这类记录组成的文件称做**定长记录文件**；若文件中含有信息长度不等的不定长记录，则称**不定长记录文件**。

数据库文件还可按记录中关键字的多少分成单关键字文件和多关键字文件。若文件中的记录只有一个惟一标识记录的主关键字，则称单关键字文件；若文件中的记录除了含

www.TopSage.com

大家网 TopSage.com

有一个主关键字外，还含有若干个次关键字，则称为多关键字文件，记录中所有非关键字的数据项称为记录的属性。

·记录的逻辑结构和物理结构

记录的逻辑结构是指记录在用户或应用程序员面前呈现的方式，是用户对数据的表示和存取方式。

记录的物理结构是数据在物理存储器上存储的方式，是数据的物理表示和组织。

通常，记录的逻辑结构着眼在用户使用方便，而记录的物理结构则应考虑提高存储空间的利用率和减少存取记录的时间，它根据不同的需要及设备本身的特性可以有多种方式。从11.1节的讨论中已得知：一个物理记录指的是计算机用一条I/O命令进行读写的基本数据单位，对于固定的设备和操作系统，它的大小基本上是固定不变的，而逻辑记录的大小是由使用要求定的。在物理记录和逻辑记录之间可能存在下列3种关系：

(1) 一个物理记录存放一个逻辑记录。

(2) 一个物理记录包含多个逻辑记录。

(3) 多个物理记录表示一个逻辑记录。

总之，用户读/写一个记录是指逻辑记录，查找对应的物理记录则是操作系统的职责，图12.2简单表示了这种关系。图中的逻辑记录和物理记录满足上述第一种关系，物理记录之间用指针相链接。

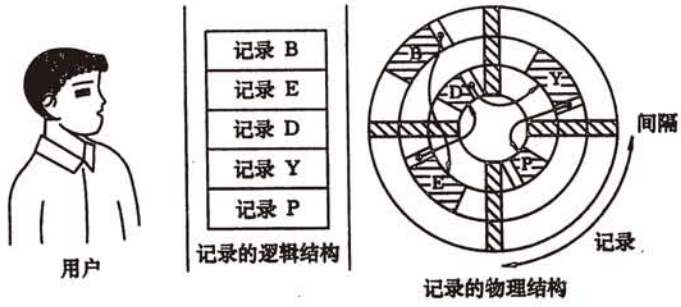


图12.2　记录的逻辑结构与物理结构差别示例

·文件的操作(运算)

文件的操作有两类：检索和修改。

文件的检索有下列3种方式：

(1) 顺序存取：存取下一个逻辑记录。

(2) 直接存取：存取第 i 个逻辑记录。

以上两种存取方式都是根据记录序号(即记录存入文件时的顺序编号)或记录的相对位置进行存取的。

(3) 按关键字存取：给定一个值，查询一个或一批关键字与给定值相关的记录。对数据库文件可以有如下4种查询方式：

① 简单询问：查询关键字等于给定值的记录。例如，在图12.1的文件中，给定一个准考证号码或学生姓名，查询相关记录。

www.TopSage.com

② 区域询问：查询关键字属某个区域内的记录。例如，在图 12.1 的文件中查询某某中学的学生成绩，则给定准考证号的某个数值范围。

③ 函数询问：给定关键字的某个函数。例如查询总分在全体学生的平均分以上的记录或处于中值的记录。

④ 布尔询问：以上 3 种询问用布尔运算组合起来的询问。例如，查询总分在 600 分以上且数学在 100 分以上，或者总分在平均分以下的外语在 98 分以上的全部记录。

文件的修改包括插入一个记录、删除一个记录和更新一个记录 3 种操作。

文件的操作可以有实时和批量两种不同方式。通常实时处理对应答时间要求严格，应在接收询问之后几秒钟内完成检索和修改，而批量处理则不然。不同的文件系统其使用有不同的要求。例如，一个民航自动服务系统，其检索和修改都应实时处理；而银行的账户系统需实时检索，但可进行批量修改，即可以将一天的存款和提款记录在一个事务文件上，在一天的营业之后再进行批量处理。

· 文件的物理结构

文件在存储介质(磁盘或磁带)上的组织方式称为文件的物理结构。文件可以有各种各样的组织方式，其基本方式有 3 种：顺序组织、随机组织和链组织。一个特定的文件应采用何种物理结构应综合考虑各种因素，如：存储介质的类型、记录的类型、大小和关键字的数目以及对文件作何种操作等。本章将介绍几种常用的文件的物理结构。

12.2 顺序文件

顺序文件(Sequential File)是记录按其在文件中的逻辑顺序依次进入存储介质而建立的，即顺序文件中物理记录的顺序和逻辑记录的顺序是一致的。若次序相继的两个物理记录在存储介质上的存储位置是相邻的，则又称**连续文件**；若物理记录之间的次序由指针相链表示，则称**串联文件**。

顺序文件是根据记录的序号或记录的相对位置来进行存取的文件组织方式。它的特点是：

(1) 存取第 i 个记录，必须先搜索在它之前的 $i-1$ 个记录。

(2) 插入新的记录时只能加在文件的末尾。

(3) 若要更新文件中的某个记录，则必须将整个文件进行复制。

由于顺序文件的优点是连续存取的速度快，因此主要用于只进行顺序存取、批量修改的情况。若对应答时间要求不严时亦可进行直接存取。

磁带是一种典型的顺序存取设备，因此存储在磁带上的文件只能是顺序文件。磁带文件适合于文件的数据量甚大、平时记录变化少、只作批量修改的情况。在对磁带文件作修改时，一般需用另一条复制带将原带上不变的记录复制一遍，同时在复制的过程中插入新的记录和用更改后的新记录代替原记录写入。为了修改方便起见，要求待复制的顺序文件按关键字有序(若非数据库文件，则可将逻辑记录号作为关键字)。

磁带文件的批处理过程可如下进行：

待修改的原始文件称做主文件，存放在一条磁带上，所有的修改请求集中构成一个文

www.TopSage.com

大家网 TopSage.com

件，称做事务文件，存放在另一台磁带上，尚需第三台磁带作为新的主文件的存储介质。主文件按关键字自小至大(或自大至小)顺序有序，事务文件必须和主文件有相同的有序关系。因此，首先对事务文件进行排序，然后将主文件和事务文件归并成一个新的主文件。图 12.3 为这个过程的示意图。在归并的过程中，顺序读出主文件与事务文件中的记录，比较它们的关键字并分别进行处理。对于关键字不匹配的主文件中的记录，则直接将其写入新主文件中。“更改”和“删去”记录时，要求其关键字相匹配。“删去”不用写入，而“更改”则要将更改后的新记录写入新主文件。“插入”时不要求关键字相匹配，可直接将事务文件上要插入的记录写到新主文件的适当位置。

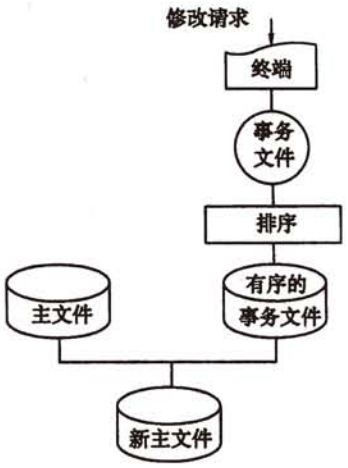


图 12.3　磁带文件批处理示意图

例如有一个银行的账目文件：其主文件保存着各储户的存款余额；每个储户作为一个记录，储户账号为关键字；记录按关键字从小到大顺序排列。一天的存入和支出集中在一个事务文件中，事务文件也按账号排序，成批地更改主文件并得到一个新的主文件，其过程如图 12.4 所示。

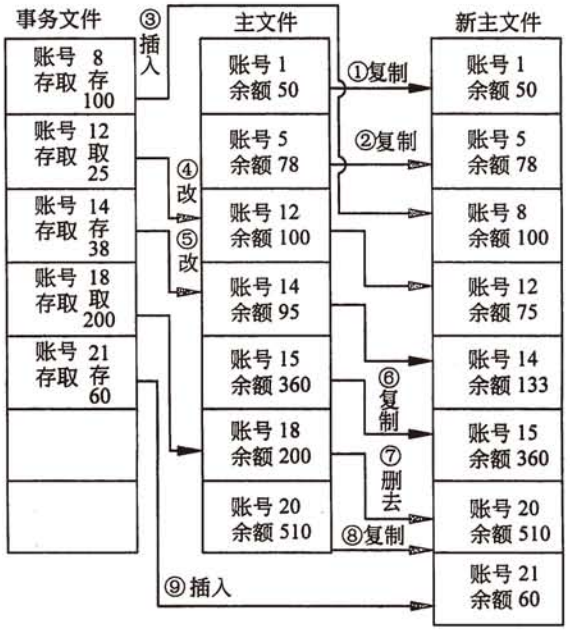


图 12.4　银行账目文件成批修改示意图

批处理的示意算法如算法 12.1 所示。算法中用到的各符号的含义说明如下：

f——主文件；g——事务文件；h——新主文件。上述三者都按关键字递增排列。事务文件的每个记录中，还增设一个代码以示修改要求，其中：“I”表示插入；“D”表示删去；

www.TopSage.com

“U”表示更改。

```
void MergeFile (FILE * f, FILE * g, FILE * h) {
  // 由按关键字递增有序的非空顺序文件 f 和 g 归并得到新文件 h,三个文件均已打
  // 开,其中,f 和 g 为只读文件,文件中各附加一个最大关键字记录,且 g 文件中对
  // 该记录的操作为插入。h 为只写文件。

  fread ( &fr, sizeof( RcdType), 1, f );
  fread ( &gr, sizeof( RcdType), 1, g );
  while ( !feof(f) || !feof(g))  {
    switch {
      case fr.key<gr.key;          // 复制“旧”主文件中记录
        fwrite ( &fr, sizeof( RcdType), 1, h );
        if (!feof(f)) fread ( &fr, sizeof(RcdType), 1, f ); break;
      case gr.code == 'D' && fr.key == gr.key;     // 删除“旧”主文件中记录,即不复制
        if (!feof(f)) fread ( &fr, sizeof(RcdType), 1, f );
        if (!feof(g)) fread ( &gr, sizeof( RcdType), 1, g );  break;
      case gr.code == 'I' && fr.key>gr.key;        // 插入,函数 P 把 gr 加工为 h 的结构
        fwrite ( P(gr), sizeof(RcdType), 1, h );
        if (!feof(g)) fread ( &gr, sizeof(RcdType), 1, g );  break;
      case gr.code == 'U' && fr.key == gr.key;     // 更改“旧”主文件中记录
        fwrite ( Q(fr,gr), sizeof(RcdType), 1, h );
                                        // 函数 Q 将 fr 和 gr 归并成一个 h 结构的记录
        if (!feof(f)) fread ( &fr, sizeof(RcdType), 1, f );
        if (!feof(g)) fread ( &gr, sizeof( RcdType), 1, g );  break;
      default ERROR( );                            // 其他均为出错情况
    }// switch
  }while
}// MergeFile
```

算法 12.1

分析批处理算法的时间。假设主文件包含 n 个记录,事务文件包含 m 个记录。一般情况下,事务文件较小,可以进行内部排序,则时间复杂度为 $O(m\log m)$。内部归并的时间复杂度为 $O(n+m)$,则总的内部处理的时间为 $O(m\log m+n)$。假设所有的输入/输出都是通过缓冲区进行的,并假设缓冲区大小为 s(个记录),则整个批处理过程中读/写外存的次数为 $2\cdot\lceil\frac{m}{s}\rceil+2\cdot\lceil\frac{m+n}{s}\rceil$①。

磁盘上的顺序文件的批处理和磁带文件类似,只是当修改项中没有插入,且更新时不增加记录的长度时,可以不建立新的主文件,而直接修改原来的主文件即可。显然,磁盘

① 此数为考虑全部修改项为插入时的上界。

www.TopSage.com

大家网 TopSage.com

文件的批处理可以在一台磁盘组上进行。

对顺序文件进行顺序查找类似于第 9 章讨论的顺序查找，其平均查找长度为 $(n'+1)/2$，其中 n' 为文件所含物理记录的数目（相对外存读/写而言，内存查找的时间可以忽略不计）。对磁盘文件可以进行分块查找或折半查找（对不定长文件不能进行折半查找）。但是，若文件很大，在磁盘上占多个柱面时，折半查找将引起磁头来回移动，增加寻查时间。

假若某个顺序文件，其记录修改的频率较低，则用批处理并不适宜，此时可另建立一个附加文件，以存储新插入和更新后的记录，待附加文件增大到一定程度时再进行批处理。在检索时可以先查主文件，若不成功再查附加文件，或反之。显然这将增加检索的时间，但可以采取其他措施弥补之，详细情况可参阅参考书目[13]。

12.3 索引文件

除了文件本身（称做数据区）之外，另建立一张指示逻辑记录和物理记录之间一一对应关系的表——索引表。这类包括文件数据区和索引表两大部分的文件称做**索引文件**。

图 12.5 所示为两个索引表的例子。索引表中的每一项称做**索引项**。不论主文件是否按关键字有序，索引表中的索引项总是按关键字（或逻辑记录号）顺序排列。若数据区中的记录也按关键字顺序排列，则称**索引顺序**文件。反之，若数据区中记录不按关键字顺序排列，则称**索引非顺序**文件。

逻辑记录号	标识	物理记录号
0	1	4
1	1	7
2	0	
3	1	10

关键字 k_i	物理记录号
101	15
119	04
123	31
125	11

图 12.5 索引表示例

索引表是由系统程序自动生成的。在记录输入建立数据区的同时建立一个索引表，表中的索引项按记录输入的先后次序排列，待全部记录输入完毕后再对索引表进行排序。例如，对应于图 12.6(a)的数据文件，其索引表如图 12.6(b)所示，而图 12.6(c)为文件记录输入过程中建立的索引表。

索引文件的检索方式为直接存取或按关键字（进行简单询问）存取，检索过程和第 9 章讨论的分块查找相类似，应分两步进行：首先，查找索引表，若索引表上存在该记录，则根据索引项的指示读取外存上该记录；否则说明外存上不存在该记录，也就不需要访问外存。由于索引项的长度比记录小得多，则通常可将索引表一次读入内存，由此在索引文件中进行检索只访问外存两次，即一次读索引，一次读记录。并且由于索引表是有序的，则查找索引表时可用折半查找法。

① 标识域指示该逻辑记录是否存在，若存在，则标识符为“1”，否则为“0”。

www.TopSage.com

物理记录号	职工号	姓名	职　务	其他
101	29	张珊	程序员	∘
103	05	李四	维修员	∘
104	02	王红	程序员	∘
105	38	刘琪	穿孔员	∘
108	31	∘	∘	∘
109	43	∘	∘	
110	17	∘	∘	
112	48	∘	∘	

(a)

	关键字	物理记录号
1	02	104
	05	103
	17	110
2	29	101
	31	108
	38	105
3	43	109
	48	112

(b)

关键字	物理记录号
29	101
05	103
02	104
38	105
31	108
43	109
17	110
48	112

(c)

图 12.6　索引非顺序文件示例

(a)文件数据区；(b)索引表；(c)输入过程中建立的索引表

索引文件的修改也容易进行。删除一个记录时，仅需删去相应的索引项；插入一个记录时，应将记录置于数据区的末尾，同时在索引表中插入索引项；更新记录时，应将更新后的记录置于数据区的末尾，同时修改索引表中相应的索引项。

当记录数目很大时，索引表也很大，以致一个物理块容纳不下。在这种情况下查阅索引仍要多次访问外存。为此，可以对索引表建立一个索引，称为查找表。假设图 12.6(b)的索引表需占用 3 个物理块的外存，每一个物理块容纳 3 个索引，则建立的查找表如图12.7所示。检索记录时，先查找查找表，再查索引表，然后读取记录。3 次访问外存即可。若查找表中项目还多，则可建立更高一级的索引。通常最高可有四级索引：数据文件→索引表→查找表→第二查找表→第三查找表。而检索过程从最高一级索引即第三查找表开始，仅需 5 次访问外存。

上述的多级索引是一种静态索引，各级索引均为顺序表结构。其结构简单，但修改很不方便，每次修改都要重组索引。因此，当数据文件在使用过程中记录变动较多时，应采用动态索引。如二叉排序树(或二叉平衡树)、B-树以及键树，这些都是树表结构，插入、删除都很方便。又由于它本身是层次结构，则无需建立多级索引，而且建立索引表的过程即排序的过程。通常，当数据文件的记录数不很多，内存容量足以容纳整个索引表时可采用二叉排序树(或平衡树)作索引，其查找性能已在第 9 章中进行了详细讨论。反之，当文件很大时，索引表(树表)本身也在外存，则查找索引时尚需多次访问外存，并且，访问外存的次数恰为查找路径上的结点数。显然，为减少访问外存的次数，就应尽量缩减索引表的深度。因此，此时宜采用 m 叉的 B-树作索引表。m 的选择取决于索引项的多少和缓冲区的大小。又，从“9.2.3/键树”的讨论可见，键树适用于作某些特殊类型的关键字的索引表。和上述对排序树的讨论类似，当索引表不大时，可采用双链表作存储结构(此时索引表在内存)；反之，则采用 Trie 树。总之，由于访问外存的时间比内存查找的时间大得多，所以

最大关键字	物理块号
17	1
38	2
46	3

图 12.7　图 12.6(b)中索引表的索引

www.TopSage.com

大家网
TopSage.com

对外存中索引表的查找效能主要取决于访问外存的次数，即索引表的深度。

显然，索引文件只能是磁盘文件。

综上所述，由于数据文件中记录不按关键字顺序排列，则必须对每个记录建立一个索引项，如此建立的索引表称之为**稠密索引**，它的特点是可以在索引表中进行“预查找”，即从索引表便可确定待查记录是否存在或作某些逻辑运算。如果数据文件中的记录按关键字顺序有序，则可对一组记录建立一个索引项，这种索引表称之为**非稠密索引**，它不能进行“预查找”，但索引表占用的存储空间少，管理要求低。下一节将介绍两种有用的索引顺序文件。

12.4 ISAM 文件和 VSAM 文件

12.4.1 ISAM 文件

索引顺序存取方法 ISAM 为 Indexed Sequential Access Methed 的缩写，它是一种专为磁盘存取设计的文件组织方式。由于磁盘是以盘组、柱面和磁道三级地址存取的设备，则可对磁盘上的数据文件建立盘组、柱面和磁道[①]三级索引。文件的记录在同一盘组上存放时，应先集中放在一个柱面上，然后再顺序存放在相邻的柱面上，对同一柱面，则应按盘面的次序顺序存放。例如图 12.8 为存放在一个磁盘组上的 ISAM 文件，每个柱面建立一个磁道索引，每个磁道索引项由两部分组成：基本索引项和溢出索引项，如图 12.9 所示，每一部分都包括关键字和指针两项，前者表示该磁道中最末一个记录的关键字（在此为最大关键字），后者指示该磁道中第一个记录的位置，柱面索引的每一个索引项也由关键字和指针两部分组成，前者表示该柱面中最末一个记录的关键字（最大关键字），后者指示该柱面上的磁道索引位置。柱面索引存放在某个柱面上，若柱面索引较大，占多个磁道时，则可建立柱面索引的索引—主索引。

在 ISAM 文件上检索记录时，先从主索引出发找到相应的柱面索引，再从柱面索引找到记录所在柱面的磁道索引，最后从磁道索引找到记录所在磁道的第一个记录的位置，由此出发在该磁道上进行顺序查找直至找到为止；反之，若找遍该磁道而不存在此记录，则表明该文件中无此记录。例如，查找关键字为 21 的记录时的查找路径如图 12.8 中的粗实线所示。

从图 12.8 中读者可看到，每个柱面上还开辟有一个溢出区；并且，磁道索引项中有溢出索引项，这是为插入记录所设置的。由于 ISAM 文件中记录是按关键字顺序存放的，则在插入记录时需移动记录，并将同一磁道上最末一个记录移至溢出区，同时修改磁道索引项。通常溢出区可有 3 种设置方法：(1)集中存放——整个文件设一个大的单一的溢出区；(2)分散存放——每个柱面设一个溢出区；(3)集中与分散相结合——溢出时记录先移至每个柱面各自的溢出区，待满之后再使用公共溢出区。图 12.8 是第二种设置法。

每个柱面的基本区是顺序存储结构，而溢出区是链表结构。同一磁道溢出的记录由

---

① 这里的磁道索引，实际为盘面索引，为遵循习惯仍称磁道索引。

www.TopSage.com

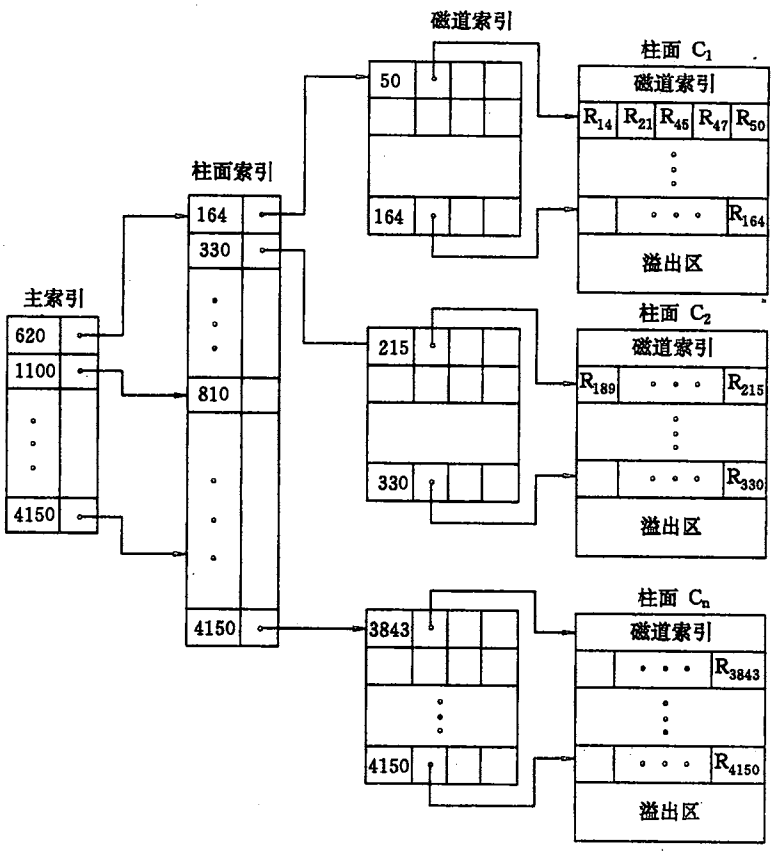


图 12.8 ISAM 文件结构示例

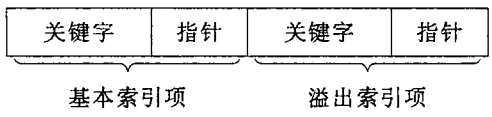
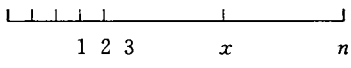


图 12.9 磁道索引项结构

指针相链，该磁道索引的溢出索引项中的关键字指示该磁道溢出的记录的最大关键字；而指针则指示在溢出区中的第一个记录。图 12.10 所示为插入记录和溢出处理的具体例子。其中(a)为插入前的某一柱面上的状态；(b)为插入 R_{65} 时，将第二道中关键字大于 65 的记录顺次后移，且使 R_{90} 溢出至溢出区的情况；(c)为插入 R_{65} 之后的状态，此时 2 道的基本索引项的关键字改为 80，且溢出索引项的关键字改为 90，其指针指向第 4 道第一个记录即 R_{90}；(d)是相继插入 R_{95} 和 R_{83} 后的状态，R_{95} 插入在第 3 道的第一个记录的位置而使 R_{145} 溢出。而由于 $80<83<90$，则 R_{83} 被直接插入到溢出区，作为第 2 道在溢出区的第一个记录，并将它的指针指向 R_{90} 的位置，同时修改第 2 道索引的溢出索引项的指针指向 R_{83}。



www.TopSage.com

大家网
TopSage.com

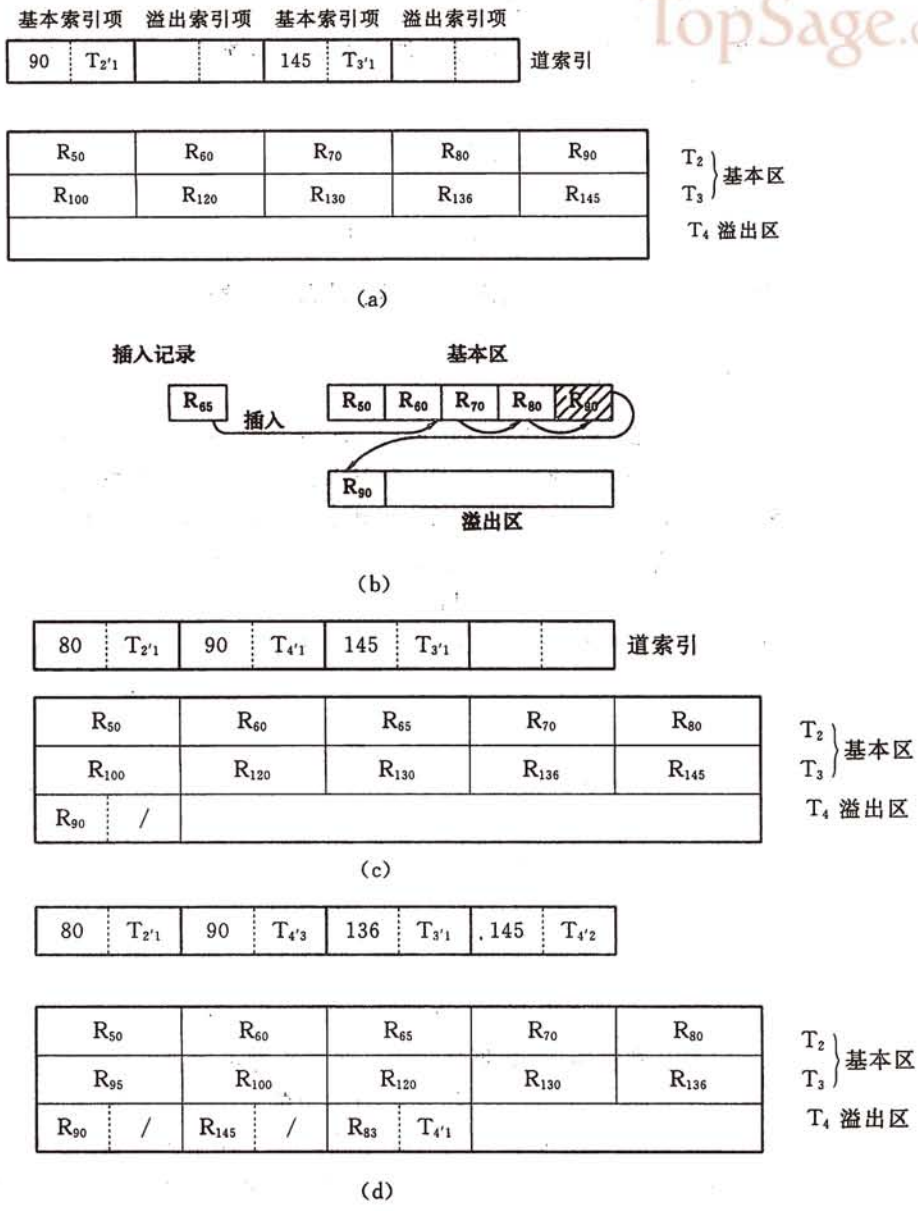


图 12.10 ISAM 文件的插入和溢出处理

(a) 插入前；(b)插入 R_{65} 时记录移动的情形；(c) 插入 R_{65} 后；(d) 先插入 R_{95} 再插入 R_{83} 后

ISAM 文件中删除记录的操作要比插入简单得多，只需找到待删除的记录，在其存储位置上作删除标记即可，而不需要移动记录或改变指针，但在经过多次的增删后，文件的结构可能变得很不合理。此时，大量的记录进入溢出区，而基本区中又浪费很多空间。因此，通常需要周期地整理 ISAM 文件。把记录读入内存，重新排列，复制成一个新的 ISAM 文件，填满基本区而空出溢出区。

最后，我们简单讨论一下 ISAM 文件中柱面索引的位置。

通常，磁道索引放在每个柱面的第一道上，那么，柱面索引是否也放在文件的第一个

www.TopSage.com

柱面上呢？由于每一次检索都需先查找柱面索引，则磁头需在各柱面间来回移动，我们希望磁头移动距离的平均值最小。假设文件占有 n 个柱面，柱面索引在第 x 柱面上。则磁头移动距离的平均值为：

$$\bar{s}=\frac{1}{n}\left[\sum_{i=1}^{x}(x-i)+\sum_{i=x+1}^{n}(i-x)\right]$$

$$=\frac{1}{n}\left[x^2-(n+1)x+\frac{n(n+1)}{2}\right]$$

令 $\frac{d\bar{s}}{dx}=0$，得到 $x=\frac{n+1}{2}$。这就是说，柱面索引应放在数据文件的中间位置的柱面上。

12.4.2 VSAM 文件

虚拟存储存取方法 VSAM 是 Virtual Storage Access Method 的缩写。这种存取方法利用了操作系统的虚拟存储器的功能，给用户提供方便。对用户来说，文件只有控制区间和控制区域等逻辑存储单位，与外存储器中柱面、磁道等具体存储单位没有必然的联系。用户在存取文件中的记录时，不需要考虑这个记录的当前位置是否在内存，也不需要考虑何时执行对外存进行“读/写”的指令。

VSAM 文件的结构如图 12.11 所示。它由 3 部分组成：索引集、顺序集和数据集。

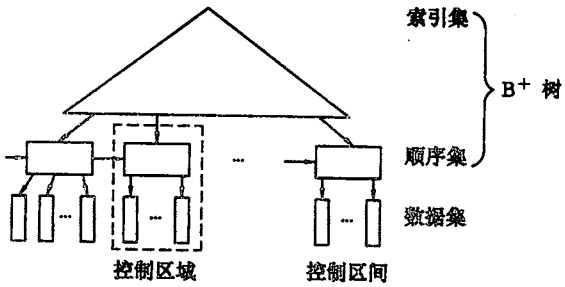


图 12.11 VSAM 文件的结构示意图

文件的记录均存放在数据集中，数据集中的一个结点称为**控制区间**(Control Interval)，它是一个 I/O 操作的基本单位，它由一组连续的存储单元组成。控制区间的大小可随文件不同而不同，但同一文件上控制区间的大小相同。每个控制区间含有一个或多个按关键字递增有序排列的记录。顺序集和索引集一起构成一棵 B^+ 树，为文件的索引部分。顺序集中存放每个控制区间的索引项。每个控制区间的索引项由两部分信息组成，即该控制区间中的最大关键字和指向控制区间的指针。若干相邻控制区间的索引项形成顺序集中的一个结点，结点之间用指针相链结，而每个结点又在其上一层的结点中建有索引，且逐层向上建立索引。所有的索引项都由最大关键字和指针两部分信息组成，这些高层的索引项形成 B^+ 树的非终端结点。因此，VSAM 文件既可在顺序集中进行顺序存取，又可从最高层的索引(B^+ 树的根结点)出发进行按关键字存取。顺序集中一个结点连同其对应的所有控制区间形成一个整体，称做**控制区域**(Control Range)。每个控制区间可视为一个逻辑磁道，而每个控制区域可视为一个逻辑柱面。

在 VSAM 文件中，记录可以是不定长的，则在控制区间中除了存放记录本身以外，还

www.TopSage.com

大家网 TopSage.com

有每个记录的控制信息(如记录的长度等)和整个区间的控制信息(如区间中存有的记录数等),控制区间的结构如图 12.12 所示。在控制区间上存取一个记录时需从控制区间的两端出发同时向中间扫描。

记录 1	…	记录 n	未利用的空闲空间	记录 n 的控制信息	…	记录 1 的控制信息	控制区间的控制信息

图 12.12　控制区间的结构示意图

VSAM 文件中没有溢出区,解决插入的办法是在初建文件时留有空间。一是每个控制区间内不填满记录,在最末一个记录和控制信息之间留有空隙;二是在每个控制区域中有一些完全空的控制区间,并在顺序集的索引中指明这些空区间。当插入新记录时,大多数的新记录能插入到相应的控制区间内,但要注意为了保持区间内记录的关键字自小至大有序,则需将区间内关键字大于插入记录关键字的记录向控制信息的方向移动。若在若干记录插入之后控制区间已满,则在下一个记录插入时要进行控制区间的分裂,即将近乎一半的记录移到同一控制区域中全空的控制区间中,并修改顺序集中相应索引。倘若控制区域中已经没有全空的控制区间,则要进行控制区域的分裂,此时顺序集中的结点亦要分裂,由此尚需修改索引集中的结点信息。但由于控制区域较大,很少发生分裂的情况。

在 VSAM 文件中删除记录时,需将同一控制区间中较删除记录关键字大的记录向前移动,把空间留给以后插入的新记录。若整个控制区间变空,则需修改顺序集中相应的索引项。

由此可见,VSAM 文件占有较多的存储空间,一般只能保持约 75%的存储空间利用率。但它的优点是:动态地分配和释放存储空间,不需要对文件进行重组,并能较快地对插入的记录进行查找,查找一个后插入记录的时间与查找一个原有记录的时间是相同的。

为了作性能上的优化,VSAM 用了一些其他的技术,如指针和关键字的压缩、索引的存放处理等。其详情读者可参阅参考书目[13]。

12.5　直接存取文件(散列文件)

直接存取文件指的是利用杂凑(Hash)法进行组织的文件。它类似于哈希表,即根据文件中关键字的特点设计一种哈希函数和处理冲突的方法将记录散列到存储设备上,故又称散列文件。

与哈希表不同的是,对于文件来说,磁盘上的文件记录通常是成组存放的。若干个记录组成一个存储单位,在散列文件中,这个存储单位叫做桶(Bucket)。假若一个桶能存放 m 个记录,这就是说,m 个同义词的记录可以存放在同一地址的桶中,而当第 $m+1$ 个同义词出现时才发生"溢出"。处理溢出也可采用哈希表中处理冲突的各种方法,但对散列文件,主要采用链地址法。

www.TopSage.com

当发生“溢出”时，需要将第 $m+1$ 个同义词存放到另一个桶中，通常称此桶为“溢出桶”；相对地，称前 m 个同义词存放的桶为“基桶”。溢出桶和基桶大小相同，相互之间用指针相链接。当在基桶中没有找到待查记录时，就顺指针所指到溢出桶中进行查找。因此，希望同一散列地址的溢出桶和基桶在磁盘上的物理位置不要相距太远，最好在同一柱面上。例如，某一文件有 18 个记录，其关键字分别为 278，109，063，930，589，184，505，269，008，083，164，215，330，810，620，110，384，355。桶的容量 $m=3$，桶数 $b=7$。用除留余数法作哈希函数 H(key)＝key MOD 7。由此得到的直接存取文件如图 12.13 所示。

在直接文件中进行查找时，首先根据给定值求得哈希地址(即基桶号)，将基桶的记录读入内存进行顺序查找，若找到关键字等于给定值的记录，则检索成功；否则，若基桶内没有填满记录或其指针域为空，则文件内不含有待查记录；否则根据指针域的值的指示将溢出桶的记录读入内存继续进行顺序查找，直至检索成功或不成功。因此，总的查找时间为：

$$T=a(te+ti)$$

其中：a 为存取桶数的期望值(相当于哈希表中的平均查找长度)，对链地址处理溢出来说，$a=1+\frac{\alpha}{2}$；te 为存取一个桶所需的时间；ti 为在内存中顺序查找一个记录所需时间。

α 为装载因子，在散列文件中

$$\alpha=\frac{n}{bm}$$

其中：n 为文件的记录数，b 为桶数，m 为桶的容量。显然，增加 m 可减少 α，也就使 a 减小，此时虽则使 ti 增大，但由于 $te>>ti$，则总的时间 T 仍可减少。图 12.14 展示了 α 和 a 的关系。

在直接存取文件中删除记录时，和哈希表一样，仅需对被删记录作一标记即可。

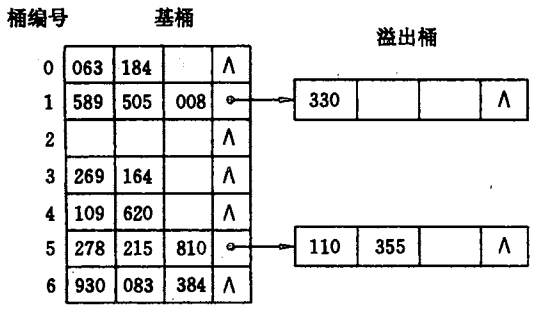


图 12.13　直接存取文件示例

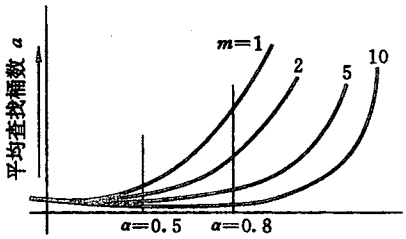


图 12.14　桶的容量和查找次数的关系

总之，直接存取文件的优点是：文件随机存放，记录不需进行排序；插入、删除方便，存取速度快，不需要索引区，节省存储空间。其缺点是：不能进行顺序存取，只能按关键字随机存取，且询问方式限于简单询问，并且在经过多次的插入、删除之后，也可能造成文件结构不合理，即溢出桶满而基桶内多数为被删除的记录。此时亦需重组文件。

www.TopSage.com

大家网
TopSage.com

12.6 多关键字文件

多关键字文件的特点是，在对文件进行检索操作时，不仅对主关键字进行简单询问，还经常需要对次关键字进行其他类型的询问检索。

例如，图 12.1 的高考成绩文件中，准考证号码为主关键字，“总分”和各单科成绩为次关键字。允许对此文件作如下询问：总分在 600 分以上的记录；数学的平均分数，等等。如果文件组织中只有主关键字索引，则为回答这些对次关键字的询问，只能顺序存取文件中的每一个记录进行比较，从而效率很低。为此，对多关键字文件，除了按以上几节讨论的方法组织文件之外，尚需建立一系列的次关键字索引。次关键字索引可以是稠密的，也可以是非稠密的；索引表可以是顺序表，也可以是树表。和主关键字索引表不同，每个索引项应包含次关键字、具有同一次关键字的多个记录的主关键字或物理记录号。下面讨论两种多关键字文件的组织方法。

12.6.1 多重表文件

多重表文件(Multilist File)的特点是：记录按主关键字的顺序构成一个串联文件，并建立主关键字的索引(称为主索引)；对每一个次关键字项建立次关键字索引(称为次索引)，所有具有同一次关键字的记录构成一个链表。主索引为非稠密索引，次索引为稠密索引。每个索引项包括次关键字、头指针和链表长度。

例如，图 12.15 所示为一个多重链表文件。其中，学号为主关键字，记录按学号顺序链接，为了查找方便，分成 3 个子链表，其索引如图 12.15(b)所示，索引项中的主关键字为各子表中的最大值。专业、已修学分和选修课目为 3 个次关键字项，它们的索引如图 12.15(c)～(e)所示，具有相同次关键字的记录链接在同一链表中。有了这些次关键字索引，便容易处理各种次关键字的询问。例如，若要查询已修学分在 400 分以上的学生，只要在索引表上查找 400～449 这一项，然后从它的链表头指针出发，列出该链表中所有记录即可。又如，若要查询是否有同时选修甲和乙课程的学生，则或从索引表上“甲”的头指针出发，或从“乙”的头指针出发，读出每个记录，察看是否同时选修这两门课程。此时可先比较两个链表的长度，显然应读出长度较短的链表中的记录。

多重链表文件易于构造，也易于修改。如果不要求保持链表的某种次序，则插入一个新记录是容易的，此时可将记录插在链表的头指针之后。但是，要删去一个记录却很繁琐，需在每个次关键字的链表中删去该记录。

12.6.2 倒排文件

倒排文件和多重表文件的区别在于次关键字索引的结构不同。通常，称倒排文件中的次关键字索引为倒排表，具有相同次关键字的记录之间不设指针相链，而在倒排表中该次关键字的一项中存放这些记录的物理记录号。例如，上例文件的倒排表如图 12.16 所示。

倒排表作索引的好处在于检索记录较快。特别是对某些询问，不用读取记录，就可得

www.TopSage.com

物理

记录号	姓　名	学　号		专　业		已修学分		选　修　课　目					
01	王　雯	1350	02	软件	02	412	03	丙	02	丁	03		
02	马小燕	1351	03	软件	07	398	07	甲	04	丙	03		
03	阮　森	1352	04	计算机	05	436	Λ	乙	05	丙	04	丁	05
04	苏明明	1353	Λ	应用	06	402	08	甲	06	丙	08		
05	田　永	1354	06	计算机	Λ	384	02	乙	07	丁	09		
06	杨　青	1355	07	应用	09	356	10	甲	07				
07	薛平平	1356	08	软件	08	398	Λ	甲	08	乙	Λ		
08	崔子健	1357	Λ	软件	Λ	408	01	甲	09	丙	Λ		
09	王　洪	1358	10	应用	10	370	05	甲	10	丁	Λ		
10	刘　倩	1359	Λ	应用	Λ	364	09	甲	Λ				

(a)

主关键字	头指针
1353	01
1357	05
1359	09

(b)

次关键字	头指针	长度
软　件	01	4
计算机	03	2
应　用	04	4

(c)

次关键字	头指针	长度
350～399	06	6
400～449	04	4

(d)

次关键字	头指针	长度
甲	02	7
乙	03	3
丙	01	5
丁	01	4

(e)

图 12.15　多重表文件示例

(a)数据文件；(b)主关键字索引；(c)“专业”索引；(d)“已修学分”索引；(e)“选修课目”索引

到解答，如询问“软件”专业的学生中有否选课程“乙”的，则只要将“软件”索引中的记录号和“乙”索引中的记录号作求“交”的集合运算即可。

在插入和删除记录时，倒排表也要作相应的修改，值得注意的是倒排表中具有同一次关键字的记录号是有序排列的，则修改时要作相应移动。

若数据文件非串链文件，而是索引顺序文件(如 ISAM 文件)，则倒排表中应存放记录的主关键字而不是物理记录号。

倒排文件的缺点是维护困难。在同一索引表中，不同的关键字其记录数不同，各倒排表的长度不等，同一倒排表中各项长度也不等。

www.TopSage.com

大家网
TopSage.com

软　件	01，02，07，08
计算机	03，05
应　用	04，06，09，10

(a)专业倒排表

350～399	02，05，06，07，09，10
400～449	01，03，04，08

(b)已修学分倒排表

甲	02，04，06，07，08，09，10
乙	03，05，07，
丙	01，02，03，04，08
丁	01，03，05，09

(c)选修课目倒排表

图 12.16　倒排文件索引示例

www.TopSage.com

附录A 名词索引

二 画	页
二叉树(binary tree)	121
二叉排序树 (binary sort tree)	227
二叉查找树 (binary search tree)	227
二次探测 (quadratic probing)	257
十字链表 (orthogonal list)	103,164
B-树	238
B^+树	246
Trie 树	249

三 画	页
三元组表 (list of 3-tuples)	97
广义表 (Lists) (generalized lists)	107
广度优先搜索 (breadth-first search)	169
子孙 (descendant)	120
子树 (subtree)	118
子图 (subgraph)	158
子串 (substring)	70
AOV-网 (Activity On Vertex network)	181
AOE-网 (Activity On Edge network)	183

四 画	页
元素 (element)	6
队列 (queue)	58
队头 (front)	59
队尾 (rear)	59
双向链表 (doubly linked list)	35
双端队列 (deque) (double—ended queue)	60
双亲 (parents)	120
双链树 (doubly linked tree)	248
中序遍历 (inorder traversal)	128,139
(表达式的)中缀表示 (infix notation)	129
无序树 (unordered tree)	120
无向图 (undirected graph) (undigraph)	157
无用单元收集 (garbage collection)	206
分配策略 (allocation strategy)	196

www.TopSage.com

大家网
TopSage.com

分块查找 (blocking search) 225
内部排序 (internal sorting) 263
文本编辑 (text editing) 84
文件 (files) 306
定长 (文件) (have fixed size records) 306
不定长 (文件) (have variable size records) 306
单关键字 (文件) (have only one key) 306
多关键字 (文件) (with more then one key) 306
开放定址 (open addressing) 257

五 画 页

头指针 (head pointer) 27
头结点 (head node) 28
边 (edge) 157
边界标识法 (boundary tag method) 198
生成树 (spanning tree) 159,170
最小 (生成树) (minimum) 173
生成森林 (spanning forest) 160
可利用空间表 (available space list) 194
平均查找长度 ASL (Average Search Length) 217
平衡二叉树 (balanced binary tree) 233
平衡因子 (balance factor) 233
平衡旋转 (balance rotation) 234
平方取中 (mid-square method) 254
平衡归并 (balanced merge) 296
归并排序 (merge sort) 283
归并插入排序 (merge insertion sort) 292
归并段 (merging segments) 295
外部排序 (external sorting) 263,293

六 画 页

存储密度 (storage density) 78
存储紧缩 (storage compaction) 212
存储结构 (storge structure) 6
顺序 (存储结构) (sequential) 6
链式 (存储结构) (linked) 6
先进先出 FIFO (First In First Out) 58
先序遍历 (preorder traversal) 128,139
(树的) 先根 (遍历) (preorder) 138
后进先出 LIFO (Last In First Out) 44
后序遍历 (postorder traversal) 128
(树的)后根(遍历) (postorder) 138

www.TopSage.com

(表达式的)后缀表示 (postfix notation) 129
回溯(backtracking) 149
有向图 (digraph) (directed graph) 157
有向无环图 (directed acycline graph) 179
有序树 (ordered tree) 120
有序段 (sorted segment) 296
伙伴系统 (buddy system) 203
网 (network) 158
关节点 (articulation point) 176
关键路径 (critical paths) 183
关键字 (Key) 214
主关键字 (primary key) 214
次关键字 (second key) 214
动态查找表 (Dynamic Searh Table) 214,226
同义词 (synonym) 252
冲突 (collision) 252,256
再哈希 (rehash) 258
伪随机探测 (random probing) 257
地址排序 (sorting by address) 264,289
延迟时间 (delay time) 294
寻查时间(seek time) 295
传输时间 (transmission time) 295
多路归并 (multi—way merge) 297
多重表文件 (multilist file) 319

七 画 页

时间复杂度 (time complexity) 15
渐近时间复杂度(asymptotic time complexity) 15
位 (bit) 6
串 (string) 70
空串 (null string) 70
空格串 (blank string) 71
层、层次 (level) 120
完全二叉树 (complete binary tree) 124
完全图 (complete graph) 158
邻接(点) (adjacent) 158
邻接矩阵 (adjacency matrix) 161
邻接表 (adjacency lists) 163
逆 (邻接表) (inverse) 164
邻接多重表 (adjacency multilists) 166
连通图 (connected graph) 159
强 (连通图) (strongly) 159

www.TopSage.com

大家网
TopSage.com

重连通（图）（biconnected） 176
连通分量（connected component） 159
折半查找（binary search） 218
折叠法（folding method） 254
移位叠加（shift folding） 255
间界叠加（folding at the boundaries） 255
判定树（decision tree） 145,220
希尔排序（Shell's method） 271
快速排序（qksort）（quicksort） 273
间隙（gap） 293
字符组间的间隙 IRG（Inter Record Gap） 293
块间的间隙 IBG（Inter Block Gap） 293

八　画　　页

空间复杂度（space complexity） 17
抽象数据类型（Abstract Data Type） 7
原子类型(atomic data type) 8
固定聚合类型(fixed-aggregate data type) 8
可变聚合类型(variable-aggregate data type) 8
多形数据类型(polymorphic data type) 9
线性表（linear lists） 18
线性链表（linear linked lists） 27
线性探测（linear probing） 257
线索二叉树（threaded binary trees） 132
线索链表（threaded linked lists） 132
单链表（singly linked lists） 27
图（graph） 156
度（degree） 120,158
入度（in—degree） 158
出度（out—degree） 158
顶点（vertex） 157
弧（arc） 157
直接前驱（immediate predecessor） 19
直接后继（immediate successor） 19
直接定址（immediately allocate） 253
拓扑排序（topological sort） 180
拓扑有序（topological order） 180
表排序（list sort） 264,267
败者树（tree of loser） 297
询问（query） 307,308
查找表（table） 214

www.TopSage.com

九　画	页
结点 (node)	6,120
孩子 (children)	120
祖先 (ancestor)	120
指针 (pointer)	27
指示器 (cursor)	32
栈 (stack)	44
栈顶 (top)	44
栈底 (bottom)	44
树 (tree)	118
树的计数 (enumeration of tree)	152
树的遍历 (traversal of tree)	138
哈希表 (Hash table)	251
哈希函数 (Hash function)	251
带权路径长度 (weighted path length)	144
(表达式的)前缀表示 (prefix notation)	129
前缀编码 (prefix code)	146
首次拟合 (first-fit)	196
查找 (searching)	214
顺序查找 (sequential search)	216
顺串 (runs)	295
顺序文件 (sequential file)	308
除留余数法 (division method)	255
选择排序 (selection sort)	277
简单(选择排序)(simple)	277
树形(选择排序)(tree)	278
段 (segments)	295

十　画	页
离散事件模拟 (discrete event simulation)	65
特殊矩阵 (special matrices)	95
递归函数 (recursive function)	54
根 (root)	118
起泡排序 (bubble sort)	272
索引顺序查找 (indexed sequential search)	225
索引顺序文件 (indexed sequential file)	311
ISAM (文件) (Indexed Sequential Access Method)	313
VSAM (文件) (Virtual Storage Access Method)	316
索引文件 (indexed file)	311
倒排文件 (inverted file)	319

www.TopSage.com

大家网
TopSage.com

十 一 画 页

深度 (depth) 113,120
深度优先搜索 DFS (Depth-First Search) 167
随机数法 (random number method) 256
排序 (sorting) 263
堆 (heap) 75,279
堆排序 (heapsort) 279
基数排序 (radix sort) 284,286
控制区间 (control interval) 316
控制区域 (control range) 316
桶 (bucket) 317
虚段 (dummy run) 305

十 二 画 页

链表 (linked list) 27
链地址法 (chaining) 258
链式基数排序 (linked radix sort) 286
循环链表 (circular linked list) 35
循环队列 (circular queue) 63
等价关系 (equivalence relations) 139
等价类 (equivalence classes) 139
等待时间 (latency time) 295
森林 (forest) 121
最优树 (optimal tree) 144
最小生成树 (minimal spanning tree) 173
最短路径 (shortest path) 186
最佳拟合 (best-fit) 197
最差拟合 (worst-fit) 197
最高位优先 MSD (Most Significant Digit first) 285
最低位优先 LSD (Least Significant Digit first) 285
最佳归并树 (optimal merge tree) 304
斐波那契序列 (Fibonacci numbers) 221
斐波那契查找 (Fibonacci search) 221
稀疏矩阵 (sparse matrix) 96
稀疏图 (sparse graph) 158
装填因子 (load factor) 260
插入排序 (insertion sort) 265
　直接 (插入) (straight) 265
　折半 (插入) (binary) 266
　2-路 (插入) (2-way) 267
　表 (插入) (table) 267
散列文件 (hashed file) 317

www.TopSage.com

十 三 画 页

数据 (data) 4
数据元素 (data element) 4
数据项 (data item) 4,18
数据对象 (data object) 4
数据关系 (data relation) 8
数据结构 (data structure) 5
逻辑结构 (logical structure) 6
物理结构 (physical structure) 6
数据类型 (data type) 7
数组 (arrays) 90
数字分析法 (digital analysis method) 254
数字查找树 (digital search tree) 247
频度 (frequency count) 15
路径 (path) 144,159
稠密图 (dense graph) 158
锦标赛排序 (tournament sort) 278
置换-选择排序 (replacement selection sort) 299
满二叉树 (full binary tree) 124

十 四 画 页

算法 (algorithm) 13
静态链表 (implementing linked lists using array) 32
模式匹配 (pattern matching) 79
静态查找表 (Static Search Table) 214,216
稳定的排序法 (stable sorting method) 263
缩小增量排序 (diminishing increment sort) 271
磁盘 (disk) 294
赫夫曼树 (Huffman tree) 144
赫夫曼编码 (Huffman codes) 146

www.TopSage.com

大家网
TopSage.com

附录B 函数索引

第 1 章		页
Bubble_sort (a[], n)	// 将 a 中整数序列重新排列成自小至大有序的整数序列	16

第 2 章		页
AddPolyn (&Pa, &Pb)	// 多项式加法:Pa = Pa+Pb	43
CreateList_L (&L, n)	// 逆位序输入 n 个元素的值,建立带表头结点的单链线性表 L	30
CreatPolyn (&P, m)	// 输入 m 项的系数和指数,建立表示一元多项式的有序链表 P	42
difference (&space, &S)	// 在一维数组 space 中建立表示集合(A−B)∪(B−A) // 的静态链表	33
Free_SL (&space, k)	// 将下标为 k 的空闲结点回收到备用链表	33
GetElem_L (L, i, &e)	// 由 e 返回带头结点的单链表 L 中第 i 个数据元素	29
InitList_Sq (&L)	// 构造一个空的顺序存储结构的线性表 L	23
InitSpace_SL (&space)	// 将一维数组 space 的各个分量链成一个备用链表	33
ListDelete_Sq (&L, i, &e)	// 在顺序线性表 L 中删除第 i 个元素,并由 e 返回其值	24
ListDelete_L (&L, i, &e)	// 在带头结点的单链线性表 L 中,删除第 i 个元素, // 并由 e 返回其值	30
ListDelete_DuL (&L, i, &e)	// 删除带头结点双链循环线性表 L 的第 i 个元素	37
ListInsert_Sq (&L, i, e)	// 在顺序线性表 L 的第 i 个元素之前插入新的元素 e	24
ListInsert_L (L, i, e)	// 在带头结点的单链线性表 L 的第 i 个元素之前插入元素 e	29,38
ListInsert_DuL (&L, i, e)	// 在带头结点的双链循环线性表 L 的第 i 个元素之前插入元素 e	36
LocateElem_Sq (L, e, (* compare))	// 在顺序线性表 L 中查找第 1 个值与 e 满足关系函数 // compare()的元素的位序	25
LocateElem_SL (S, e)	// 在静态单链线性表 L 中查找第 1 个值为 e 的元素	32
Malloc_SL (&space)	// 若备用空间链表非空,则返回分配的结点下标,否则返回 0	33
MergeList (La, Lb, &Lc)	// 归并有序线性表 La 和 Lb 得到新的有序线性表 Lc	21
MergeList_Sq (La, Lb, &Lc)	// 归并有序顺序表 La 和 Lb 得到新的有序顺序表 Lc	26
MergeList_L (&La, &Lb, &Lc)	// 归并有序单链表 La 和 Lb 得到新的有序单链表 Lc	31,39
union (&La, Lb)	// 将所有在线性表 Lb 中但不在 La 中的数据元素插入到 La	20

第 3 章		页
Bank_Simulation (CloseTime)	// 银行业务模拟,统计一天内客户在银行逗留的平均时间	68
Conversion ()	// 对于输入的任意一个非负十进制整数,打印输出与其等值的八进制数	48
DeQueue (&Q, &e)	// 删除非空链队列 Q 的队头元素,并用 e 返回其值	62
DeQueue (&Q, &e)	// 删除非空循环队列 Q 的队头元素,并用 e 返回其值	65
DestroyQueue (&Q)	// 销毁链队列 Q	62

www.TopSage.com

EnQueue (&Q, e)	// 插入元素 e 为链队列 Q 的新的队尾元素	62
EnQueue (&Q, e)	// 插入元素 e 为循环队列 Q 的新的队尾元素	65
EvaluateExpression()	// 按算符优先算法对从终端读入的算术表达式求值	53
GetTop (S, &e)	// 用 e 返回非空顺序栈 S 的栈顶元素	47
Hanoi (n, x, y, z)	// 将塔座 x 上编号为 1 至 n 的 n 个圆盘按规则搬到塔座 z 上	55
InitQueue (&Q)	// 构造一个空的链队列 Q	62
InitQueue (&Q)	// 构造一个空的循环队列 Q	64
InitStack (&S)	// 构造一个空的顺序栈 S	47
LineEdit()	// 一个简单的行编辑程序	50
MazePath (maze, start, end)	// 求迷宫 maze 中从入口 start 到出口 end 的一条通道	51
Push (&S, e)	// 插入元素 e 为顺序栈 S 新的栈顶元素	47
Pop (&S, &e)	// 删除非空顺序栈 S 的栈顶元素,用 e 返回其值	47
QueueLength (Q)	// 返回循环队列 Q 中的元素个数,即队列的长度	64

第 4 章 **页**

ClearString (&S)	// 将堆存储结构的串 S 清为空串	77
Concat (&T, S1, S2)	// 用 T 返回由顺序存储结构串 S1 和 S2 联接而成的新串	73
Concat (&T, S1, S2)	// 用 T 返回由堆存储结构的串 S1 和 S2 联接而成的新串	77
get_ next (T, &next[])	// 求模式串 T 的 next 函数值	83
get_ nextval (T, &nextval[])	// 求模式串 T 的 next 函数修正值	84
Index (S,T,pos)	// 返回非空子串 T 在主串 S 中的位置	72,79
Index_ KMP (S,T,pos)	// 利用 next 函数值求非空子串 T 在主串 S 中的位置	82
StrAssign (&T, * chars)	// 把串常量 chars 赋为堆存储结构的串 T 的值	76
StrCompare (S, T)	// 返回堆存储结构的串 S 和 T 的比较结果	77
InsIdxList(&idxlist,bno)	// 将书号为 bno 的关键词插入索引表 idxlist	88
StrInsert (&S, pos, T)	// 在堆存储结构的串 S 中第 pos 个字符之前插入串 T	75
StrLength (S)	// 返回堆存储结构的串 S 的长度	77
SubString (&Sub, S, pos, len)	// 用 Sub 返回顺序存储结构的串 S 中第 pos 个字符起长度为 // len 的子串	75
SubString (&Sub, S, pos, len)	// 用 Sub 返回堆存储结构的串 S 中第 pos 个字符起长度 // 为 len 的子串	77

第 5 章 **页**

CopyGList (&T, L)	// 采用头尾链表存储结构,由广义表 L 复制得到广义表 T	115
CreateGList (&L, S)	// 采用头尾链表存储结构,由广义表的书写形式串 S 创建广义表 L	116
CreateSMatrix_ OL (&M)	// 创建用十字链表存储表示的稀疏矩阵 M	104
FastTransposeSMatrix (M, &T)	// 求用三元组顺序表表示的稀疏矩阵 M 的转置矩阵 T	100
GListDepth (L)	// 采用头尾链表存储结构,求广义表 L 的深度	114
MultSMatrix (M, N, &Q)	// 求采用行逻辑链接存储表示的矩阵乘积 Q=M×N	102
Sever (&str, &hstr)	// 以第一个',' 为分界符,将非空串 str 分割成两部分:hsub 和 str	117
TransposeSMatrix (M, &T)	// 求采用三元组顺序表表示的稀疏矩阵 M 的转置矩阵 T	99

www.TopSage.com

大家网
TopSage.com

第 6 章

		页
CreateBiTree (&T)	// 按先序次序输入结点值,构造二叉链表表示的二叉树 T	131
find_ mfset (S, i)	// 找集合 S 中 i 所在子集的根	141
fix_ mfset (&S, i)	// 确定 i 所在子集,并将从 i 至根路径上所有结点都变成根的孩子结点	143
GetPowerSet (i,A,&B)	// 用线性表表示集合时求 A 的幂集 ρ(A)	150
HuffmanCoding (&HT, &HC, * w, n)	// 构造赫夫曼树 HT,并求得字符的赫夫曼编码 HC	147
InOrderTraverse (T, (* Visit))	// 中序遍历二叉树 T 的非递归算法	130,131
InOrderTraverse_ Thr (T, (* Visit))	// 中序遍历二叉线索链表表示的二叉树 T	134
InOrderThreading (&Thrt, T)	// 中序遍历二叉树 T,生成中序线索链表 Thrt	134
InThreading (p)	// 对以 P 为根的二叉树进行中序线索化	135
merge_ mfset (&S, i, j)	// 求 S 中两个互不相交的子集 Si 和 Sj 的并集 $S_i \cup S_j$	141
mix_ mfset (&S, i, j)	// 求两个互不相交的子集 S_i 和 S_j 的并集 $S_i \cup S_j$	142
PreOrderTraverse (T, (* Visit))	// 先序遍历二叉树 T 的递归算法	129
PowerSet (i, n)	// 求含 n 个元素的集合 A 的幂集 ρ(A)	150
Trial (i, n)	// 求 n 皇后问题棋盘的合法布局,并输出之	151

第 7 章

		页
BFSTraverse (G, (* Visit))	// 按广度优先遍历图 G,访问(Visit)图中所有顶点	170
CreateGraph (&G)	// 构造图 G 的数组(邻接矩阵)表示存储结构	162
CreateUDN (&G)	// 构造无向网 G 的数组(邻接矩阵)表示存储结构	162
CreateDG (&G)	// 构造有向图 G 的十字链表存储结构	165
CriticalPath (G)	// 输出有向网 G 的各项关键活动	185
DFS (G, v)	// 从第 v 个顶点出发递归地深度优先遍历图 G	169
DFSArticul (G, v0)	// 从第 v0 个顶点出发深度优先遍历图 G,查找并输出关节点	178
DFSForest (G, &T)	// 建立无向图 G 的深度优先生成森林的(最左)孩子(右)兄弟链表 T	171
DFSTraverse (G, (* Visit))	// 对图 G 作深度优先遍历	169
DFSTree (G, v, &T)	// 从第 v 个顶点出发深度优先遍历图 G,建立以 T 为根的生成树	172
FindArticul (G)	// 查找并输出连通图 G 上全部关节点	178
MiniSpanTree_ PRIM (G, u)	// 用普里姆算法从第 u 个顶点出发构造网 G 的最小生成树 T	175
ShortestPath_ DIJ (G, v0, &P, &D)	// 用 Dijkstra 算法求有向网 G 中从顶点 v0 到其余顶点 v 的最短路径 P[v]及其带权长度 D[v]	189
ShortestPath_ FLOYD (G, &P[], &D)	// 用 Floyd 算法求有向网 G 中各对顶点 v 和 w 之间的最短路径 P[v][w]及其带权长度 D[v][w]	191
TopologicalOrder (G, &T)	// 求有向网 G 中各顶点事件的最早发生时间 ve	185
TopologicalSort (G)	// 若 G 无回路,则输出 G 的顶点的一个拓扑序列	182

第 8 章

		页
AllocBoundTag (&pav, n)	// 边界标识法的存储分配算法	200
AllocBuddy (&avail, n)	// 伙伴系统的存储分配算法	205
MarkList (GL)	// 遍历非空广义表 GL,对表中所有未加标志的结点加标志	209

www.TopSage.com

第 9 章		页
CreateSOSTree (&T, ST)	// 由有序表 ST 构造一棵次优查找树 T	225
DeleteBST (&T, key)	// 若二叉排序树 T 中存在关键字等于 key 的数据元素时,则删除	
	// 该数据元素结点 p 并返回 TRUE;否则返回 FALSE	230
InsertAVL (&T, e, &taller)	// 在平衡的二叉排序树 T 中插入原树中不存在有相同关键字的	
	// 结点,若因插入而使二叉排序树失去平衡,则作平衡旋转处理	237
InsertBST (&T, e)	// 当二叉排序树 T 中不存在关键字等于 e.key 的数据元素时,	
	// 插入 e 并返回 TRUE	228
InsertBTree (&T, K, q, i)	// 在 m 阶 B 树 T 上结点 * q 的 key[i]与 key[i+1]之间插入	
	// 关键字 K	244
InsertHash (&H, e)	// 查找不成功时插入数据元素 e 到开放定址哈希表 H 中,并返回	
	// OK;若冲突次数过大,则重建哈希表	259
LeftBalance (&T)	// 对以指针 T 所指结点为根的二叉树作左平衡旋转处理	237
L_Rotate (&p)	// 对以 p↑为根的二叉排序树作左旋处理,处理之后 p 指向新的	
	// 树根结点	236
R_Rotate (&p)	// 对以 p↑为根的二叉排序树作右旋处理,处理之后 p 指向新	
	// 的树根结点	236
Search_Bin (ST, key)	// 在有序表 ST 中折半查找其关键字等于 key 的	
	// 数据元素	220
SearchBST (T, key, f, &p)	// 在 T 所指二叉排序树中递归查找其关键字等于 key 的	
	// 数据元素	228
SearchBTree (T, K)	// 在 m 阶 B 树 T 上查找关键字 K,返回结果(pt,i,tag)	240
SearchDLTree (T, K)	// 在非空双链树 T 中查找关键字等于 K 的记录	248
SearchHash (H, K, &p, &c)	// 在开放定址哈希表 H 中查找关键码为 K 的元素	259
Search_Seq (ST, key)	// 在顺序表 ST 中顺序查找其关键字等于 key 的数据元素	216
SearchTrie (T, K)	// 在键树 T 中查找关键字等于 K 的记录	250
SecondOptimal (&T, R[], sw[], low, high)	// 由有序表 R[low..high]及其累计权值表 sw	
	// 递归构造次优查找树 T	223

第 10 章		页
Arrange (&SL)	// 根据静态链表 SL 中各结点的指针值调整记录位置,使得 SL	
	// 中记录按关键字非递减有序顺序排列	269
BInsertSort (&L)	// 对顺序表 L 作折半插入排序	267
Collect (&r, i, f, e)	// 本算法按 keys[i]自小至大地将 f[0..RADIX-1]所指各子表	
	// 依次链接成一个链表, e[0..RADIX-1]为各子表的尾指针	288
Distribute (&r, i, &f, &e)	// 静态链表 L 的 r 域中记录已按(keys[0],...,keys[i-1])有序,	
	// 本算法按第 i 个关键字 keys[i]建立 RADIX 个子表,使同一子	
	// 表中记录的 keys[i]相同	288
HeapAdjust (&H, s, m)	// 已知 H.r[s..m]中记录的关键字除 H.r[s].key 之外均满足	
	// 堆的定义,本函数调整 H.r[s]的关键字,使 H.r[s..m]成为	

www.TopSage.com

大家网 TopSage.com

	// 一个大顶堆(对其中记录的关键字而言)	282
HeapSort (&H)	// 对顺序表 H 进行堆排序	282
InsertSort (&L)	// 对顺序表 L 作直接插入排序	265
Merge (SR[], &TR[], i, m, n)	// 将有序的 SR[i..m]和 SR[m+1..n]归并为有序的	
	// TR[i..n]	283
MergeSort (&L)	// 对顺序表 L 作归并排序	284
Msort (SR[], &TR1[], s, t)	// 将 SR[s..t]归并排序为 TR1[s..t]	284
Partition (&L, low, high)	// 交换顺序表 L 中子序列 L.r[low..high]的记录,使枢轴记录	
	// 到位,并返回其所在位置	274
Qsort (&L, low, high)	// 对顺序表 L 中的子序列 L.r[low..high]进行快速排序	275
QuickSort (&L)	// 对顺序表 L 进行快速排序	276
RadixSort (&L)	// L 是采用静态链表表示的顺序表,对 L 作基数排序,使得 L	
	// 成为按关键字自小到大的有序静态链表,L.r[0]为头结点	288
Rearrange (&L, adr[])	// adr 给出顺序表 L 的有序次序,即 L.r[adr[i]]是第 i 小的记录,	
	// 本算法按 adr 重排 L.r,使其有序	290
SelectSort (&L)	// 对顺序表 L 作简单选择排序	277
ShellInsert (&L, dk)	// 对顺序表 L 作一趟希尔插入排序,dk 为增量	272
ShellSort (&L, dlta[], t)	// 按增量序列 dlta[0..t-1]对顺序表 L 作希尔排序	272

第 11 章 **页**

Adjust (&ls, s)	// 沿从叶子结点 b[s]到根结点 ls[0]的路径调整败者树	299
Construct_Loser (&ls, &wa)	// 输入 w 个记录到内存工作区 wa,建得败者树 ls,选出关键字	
	// 最小的记录	302
CreateLoserTree (&ls)	// 已知 b[0]到 b[k-1]为完全二叉树 ls 的叶子结点存有 k	
	// 个关键字,沿从叶子到根的 k 条路径将 ls 调整成为败者树	299
get_run (&ls, &wa)	// 求得一个初始归并段	301
K_Merge (&ls, &b)	// 利用败者树 ls 将编号从 0 到 k-1 的 k 个输入归并段中	298
Replace_Selection (&ls, &wa, *fi, *fo)	// 在败者树 ls 和内存工作区 wa 上用置换-选择	
	// 排序求初始归并段,fi 为输入文件,fo 为输出文件	301
Select_MiniMax (&ls, wa, q)	// 从 wa[q]起到败者树的根比较选择 MINIMAX 记录,并由	
	// q 指示它所在的归并段	302

第 12 章 **页**

MergeFile (*f, *g, *h)	// 由按关键字递增有序的非空顺序文件 f 和 g 归并得到新文件 h	310

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：Java 视频教程 | Java SE | Java EE

.Net 技术精品资料下载汇总：ASP.NET 篇

.Net 技术精品资料下载汇总：C#语言篇

.Net 技术精品资料下载汇总：VB.NET 篇

撼世出击：C/C++编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 Ruby、Ruby on Rails 精品电子书等学习资料下载

数据库管理系统(DBMS)精品学习资源汇总：MySQL 篇 | SQL Server 篇 | Oracle 篇

平面设计优秀资源学习下载 | Flash 优秀资源学习下载 | 3D 动画优秀资源学习下载

最强 HTML/xHTML、CSS 精品学习资料下载汇总

最新 JavaScript、Ajax 典藏级学习资料下载分类汇总

网络最强 PHP 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 LinuxCBT 视频教程系列 Linux 快速学习视频教程一帖通

天罗地网：精品 Linux 学习资料大收集(电子书+视频教程) Linux 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

>> 更多精品资料请访问大家论坛计算机区…

参考书目

[1] Horowitz E, Sahni S. Fundamentals of Data Structures. Pitmen Publishing Limited, 1976

[2] Knuth D E. The Art of Computer Programming, volume 1/Fundamental Algorithms; volume3/ Sorting and Searching. Addison—Wesley Publishing Company, Inc., 1973

[3] Gotlieb C C, Gotlieb L R. Data Types and Structures. Prentice—Hall Inc., 1978

[4] Tenenbaum A M, Augensetein M J. Data Structures Using PASCAL. Prentice—Hall, Inc., 1981

[5] Baron R J, Shapiro L G. Data Structures and their Implementation. Van Nostrand Reinhold Company, 1980

[6] Aho A V, Hopcroft J E, Ullman J D. Data Structures and Algorithms. Addison—Wesley Publishing Company, Inc., 1983

[7] Esakov J, Weiss T. Data Structures: An Advanced Approach Using C. Prentice-Hall, Inc., 1989

[8] [美] S巴斯. 计算机算法:设计和分析引论. 朱洪等译. 上海:复旦大学出版社, 1985

[9] Wirth N. Algorithms+Dada Structures= Programs. Prentice—Hall, Inc., 1976

[10] Lewis T G, Smith M Z. Applying Data Structures. Houghton Mifflin Company, 1976

[11] Donovan J J. Operating System. McGraw—Hill, Inc., 1974

[12] Tremblay J P, Sorenson P G. An Introduction to Data Structure with Applications, Second Edition. McGraw—Hill, Inc., 1984

[13] 姚诗斌. 数据库系统基础. 计算机工程与应用, 1981 年第 8 期

[14] Stubbs D F, Wibre N W. Data Structures with Abstract Data Types and Pascal. Brooks/Cole Publishing Company, 1985

www.TopSage.com