



励志改变人生
编程改变命运



零基础学 数据结构

第2版

24小时多媒体教学视频

陈锐 成建设 等编著

本书特色

- ◎ 由浅入深，循序渐进，从零开始学数据结构，一点都不难
- ◎ 编程基础、编程进阶、编程应用、项目实战、上机练习
- ◎ 实例讲解+综合项目开发+配套学习练习题

超值、大容量DVD

- ◎ 本书教学视频
- ◎ 本书例题源代码
- ◎ 本书教学PPT

本书技术支持

- ◎ 邮箱: nwuchenrui@126.com
- ◎ QQ群: 216732263



机械工业出版社
China Machine Press



零基础学数据结构（第2版）

陈锐 等著

ISBN: 978-7-111-46861-5

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

前言

第一篇 基础知识

第1章 数据结构概述

- 1.1 为什么要学习数据结构
- 1.2 基本概念和术语
- 1.3 数据的逻辑结构与存储结构
- 1.4 抽象数据类型及其描述
- 1.5 算法
- 1.6 算法分析
- 1.7 学好数据结构的秘诀
- 1.8 习题

第2章 C语言基础

- 2.1 C语言开发环境
- 2.2 递归与非递归
- 2.3 指针
- 2.4 参数传递
- 2.5 结构体与联合体
- 2.6 链表
- 2.7 小结

2.8 习题

第二篇 线性数据结构

第3章 线性表

3.1 线性表的定义及抽象数据类型

3.2 线性表的顺序表示与实现

3.3 线性表的链式表示与实现

3.4 循环单链表

3.5 双向链表

3.6 静态链表

3.7 综合案例：一元多项式的表示与相乘

3.8 小结

3.9 习题

第4章 栈

4.1 栈的定义与抽象数据类型

4.2 栈的顺序表示与实现

4.3 栈的链式表示与实现

4.4 栈的典型应用

4.5 栈与递归

4.6 小结

4.7 习题

第5章 队列

5.1 队列的定义与抽象数据类型

5.2 队列的顺序存储及实现

5.3 队列的链式存储及实现

5.4 双端队列

5.5 综合案例：动画模拟停车场管理系统

5.6 小结

5.7 习题

第6章 串

6.1 串的定义及抽象数据类型

6.2 串的顺序表示与实现

6.3 串的堆分配表示与实现

6.4 串的块链式存储表示与实现

6.5 串的模式匹配

6.6 小结

6.7 习题

第7章 数组

7.1 数组的定义及抽象数据类型

7.2 数组的顺序表示与实现

7.3 特殊矩阵的压缩存储

7.4 稀疏矩阵的压缩存储

7.5 稀疏矩阵应用举例

7.6 稀疏矩阵的十字链表表示与实现

7.7 小结

7.8 习题

第8章 广义表

8.1 广义表的定义及抽象数据类型

8.2 广义表的头尾链表表示与实现

8.3 广义表的扩展线性链表表示与实现

8.4 小结

8.5 习题

第三篇 非线性数据结构

第9章 树

9.1 树的相关概念及抽象数据类型

9.2 二叉树的相关概念及抽象数据类型

9.3 二叉树的存储表示与实现

9.4 遍历二叉树

9.5 遍历二叉树的应用

9.6 线索二叉树

9.7 树、森林与二叉树

9.8 综合案例：哈夫曼树

9.9 小结

9.10 习题

第10章 图

10.1 图的定义与相关概念

10.2 图的存储结构

10.3 图的遍历

10.4 图的连通性问题

10.5 有向无环图

10.6 最短路径

10.7 图的应用举例

10.8 小结

10.9 习题

第四篇 查找与排序

第11章 查找

11.1 基本概念

11.2 静态查找

11.3 动态查找

11.4 B-树与B+树

11.5 哈希表

11.6 小结

11.7 习题

第12章 内排序

12.1 基本概念

12.2 插入排序

12.3 交换排序

12.4 选择排序

12.5 归并排序

12.6 基数排序

12.7 小结

12.8 习题

第13章 外排序

13.1 外存的存取特性

13.2 磁盘排序

13.3 磁带排序

13.4 小结

参考文献

光盘内容

前言

《零基础学数据结构》自问世以来，已被许多高校选为数据结构教材，得到了众多读者的关心和问候，受到读者的喜欢和好评。广大读者非常期待第2版，同时对本书的修订提出了不少宝贵意见。有这么多热心读者关心本书，我感到非常欣慰，在此也对所有关注本书的朋友们说声谢谢！希望更多的朋友关注本书，以及提出更多的改进建议。

经过修订后，本书案例更加丰富，语言表达更加简练、准确，替换了部分重复性的案例，保留了精华内容，修订了书中的错误和不足之处，保证所有程序能正确运行，视频讲解更加针对重点、难点进行分析。

“数据结构”作为计算机专业的一门专业基础课程，对于初学者来说，许多专业术语较为抽象，不容易理解和掌握，本书采用通俗的语言进行讲解，针对每个知识点都给出例子和图表，便于读者理解和掌握。本书内容全面，涵盖数据结构的所有知识点，全书算法采用C语言实现，所有代码均在Visual C++6.0环境下调试通过，所有案例都是完整程序，能直接运行。

本书不仅适合正在学习数据结构的学生作为自学教材，也适合作为计算机专业学生考研辅导用书和参加软考人员的辅导书。

修订的内容

1. 更正了书中的错误

本书第1版有些算法描述中存在一些不易察觉的错误，第2版重新对书中代码进行了全部调试，把错误的地方一一更正。根据读者提出的宝贵建议，对第1版中一些错误的表述也进行了修改。

2. 修订了书中的内容

本书第1版有些概念描述不够准确，第2版对所有已发现的不恰当地方进行了修改，在不易理解的地方增加了图表，重新表述了许多概念和定义，使本书更易于理解，而且增加了近年考研题目，内容更加完善。

3. 补充替换书中的案例

这次改版，删除了第1版的一些案例，并补充了一些较大型的案例，如迷宫问题、模拟停车场管理系统等，增加了近两年的考研算法

试题，减少了一些重复性的案例，保留了一些具有代表性的案例，使本书更加实用。

4. 视频讲解突出重点、难点

在本书配套的视频中，作者针对数据结构中的一些重点和难点部分进行详细分析，特别是对一些典型案例做了详细分析，通过学习本书并结合视频讲解，可使每一位读者都能真正理解并掌握数据结构中的每一个知识点。

本书的第1~4章和第9章由陈锐编写，第7~8章由成建设编写，第5章由张立编写，第6章由李得强编写，其他章节由李铁塔、蔡洪涛、付海涛、段小涛、申文彬、郑苗苗编写。

为什么要学数据结构

如果你打算今后从事软件开发，或从事计算机科研、教学等工作，必须要学好数据结构这门课程。首先，因为数据结构作为计算机专业的专业基础课程，是计算机考研的必考科目之一，如果打算报考计算机专业的研究生，你必须学好它；其次，数据结构是计算机软考、计算机等级考试等相关考试的必考内容之一，要是想顺利通过这些考试，你也必须学好它；最后，数据结构还是你今后毕业，进入各

软件公司、事业单位的必考内容之一，想要找到好工作，也必须学好它。

即使你没有以上考虑，作为一名计算机从业人员，数据结构是其他后续计算机专业课程的基础，许多课程都会用到数据结构知识。有如此多的理由，你必须掌握好数据结构。

如何学好数据结构

对于初学者来说，数据结构这门课有许多抽象的东西，不是太容易掌握。万事开头难，只要你掌握了方法和技巧，学任何东西就会变得很容易，学习数据结构也是如此。要想学好数据结构，首先应该有信心，要有战胜困难的决心，特别一开始不要有畏惧心理，这一点很重要；其次就是要掌握好C语言，C语言是基础，因为本书中的算法都是用C语言描述的（其他大多数数据结构图书也采用C语言描述），即使之前没有掌握好C语言也没有关系，只要有C语言基础就行，可以边学数据结构边巩固C语言知识。

有了以上两点，你就离成功不远了，数据结构也没有那么可怕，其实就是概念抽象了点，本书已经进行了通俗的讲解，再多联系实际生活，学习数据结构就会变得很轻松。

最后一点就是多上机，多思考，本书中所有算法都用C语言表述，并给出完整程序，你只需要把程序看懂，然后上机多调试，锻炼C语言的应用技巧，对数据结构中的一些算法思想就可以融会贯通，真正领会其中的内涵。

如何使用本书

本书全面讲解了数据结构的相关知识，案例非常丰富，第2版加入了作者对数据结构的理解，还修订了很多错误和不足之处。本书用通俗易懂的语言描述抽象的概念，配套视频针对重点和难点进行了讲解，方便大家理解与学习。

本书可以作为学习数据结构的自学教材，也可以作为案头必备的参考书，值得珍藏。本书很适合初学数据结构的读者阅读，也可作为参加计算机考研学生的辅导书。

在使用本书过程中，可以边看书，边听视频讲解，视频讲解主要针对本书中的难点和重点，每学完一部分内容，可以在电脑上调试本书配套的代码，认真领会算法的思想，并思考为什么要这样实现。

相信在学完本书后，大家会在数据结构和算法方面有很大的收获。预祝大家在学习本书时有一个愉快的旅程。

致谢

我要感谢帮助本书问世的所有人，尤其是机械工业出版社的李华君编辑，他十分看重本书的应用价值，在他的努力下本书才得以顺利出版，对此，我深怀感激。

还要感谢我的导师张蕾教授。她是对我职业生涯最有影响的人之一，她丰富的知识储备及敏锐的洞察力极大地影响了我的学习态度，促使我的学习能力和认识能力有了很大提高，也为本书的编写奠定了良好的知识与技术基础。

耿国华老师在数据结构和算法领域有很高的造诣，耿教授在数据结构与算法领域给了我很大启发。

还要感谢我的家人，在他们的默默付出与鼓励下，我才能顺利写完本书。

最后还要感谢温县教育局电教馆全体同仁的帮助与鼓励，尤其要感谢张全仕馆长对我写作的关心与支持。

由于作者水平有限，书中难免存在一些不足之处，恳请读者批评指正。读者可通过邮箱nwuchenrui@126.com与作者联系，也可通过QQ群（216732263）与作者交流。

陈锐

第一篇 基础知识

第1章 数据结构概述

数据结构是一门研究如何用计算机描述事物及其之间关系的学问，它是计算机学科的专业基础课程，是今后从事计算机软件开发的重要基础。它主要研究数据在计算机中的存储表示和对数据的处理方法。数据结构把数据划分为集合、线、树和图4种类型，后3种是数据结构研究的重点。本章旨在让读者对数据结构有个总体的把握，首先介绍了数据结构的有关概念，接着介绍了什么是抽象数据类型及抽象数据类型的描述方法，然后介绍了数据的逻辑结构与存储结构，最后介绍了算法的定义、算法的描述方法、算法设计的要求以及如何分析算法的效率高低。

本章重点和难点：

- 数据结构的基本概念
- 什么是抽象数据类型
- 如何计算算法的时间复杂度

1.1 为什么要学习数据结构

1. 数据结构的发展变迁

数据结构在计算机科学与技术专业中是一门综合性的专业基础课。在国外，数据结构作为一门独立的课程是从1968年才开始设立的，但在此之前其有关内容已散见于编译原理及操作系统之中。20世纪60年代中期，美国的一些大学开始开设相关课程，但当时的课程名称并不叫数据结构。

1968年，美国的唐·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，结构化程序设计成为程序设计方法学的主要内容，人们越来越重视数据结构。

从20世纪70年代中期到80年代，各种版本的数据结构著作相继出现。目前，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，从抽象数据类型和面向对象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。

2. 数据结构的地位

在我国，数据结构已经不仅仅是计算机专业的核心课程，还是其他非计算机专业的主要选修课程之一。数据结构的研究不仅涉及计算机硬件的研究范围，而且与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。

开发所有计算机系统软件和应用软件都要用到各种类型的数据结构。因此，要想更好地运用计算机来解决实际问题，仅掌握几种计算机程序设计语言是难以应付众多复杂问题的。要想有效地使用计算机、充分发挥计算机的性能，还必须学习和掌握好数据结构方面的有关知识。打好“数据结构”这门课程的扎实基础，对于学习计算机专业的其他课程，如操作系统、编译原理、数据库管理系统、软件工程、人工智能等都是十分有益的。

在计算机发展的初期，人们使用计算机的目的主要是处理数值计算问题。当我们使用计算机来解决一个具体问题时，一般需要经过几个步骤，首先要从该具体问题中抽象出一个适当的数学模型；然后设计或选择一个解决此数学模型的算法；最后编出程序进行调试、测

试，直至得到最终的解答。例如，求解桥梁结构中应力的数学模型的线性方程组，该方程组可以使用迭代算法来求解。

由于当时所涉及的运算对象是简单的整型、实型或布尔类型数据，所以程序设计者的主要精力是集中于程序设计的技巧上，而无须重视数据结构。随着计算机应用领域的扩大和软、硬件的发展，非数值计算问题显得越来越重要。据统计，当今处理非数值计算性问题占用了90%以上的机器时间。这类问题涉及的数据结构更为复杂，数据元素之间的相互关系一般无法用数学方程式加以描述。因此，解决这类问题的关键不再是数学分析和计算方法，而是要设计出合适的数据结构，才能有效地解决问题。

学习数据结构的目的是为了了解计算机处理对象的特性，将实际问题中所涉及的处理对象在计算机中表示出来并对它们进行处理。与此同时，通过算法训练来提高学生的思维能力，通过程序设计的技能训练来促进学生的综合应用能力和专业素质的提高。

1.2 基本概念和术语

本节主要介绍数据结构有关的一些基本概念和术语，以便读者在今后的学习过程中有统一的认识与理解。这些概念与术语将在以后的章节中频繁出现。

1. 数据

数据（data）是描述客观事物的符号，能输入到计算机中并能被计算机程序处理的符号集合。它是计算机程序加工的“原料”。例如，一个文字处理程序（如Microsoft Word）的处理对象就是字符串，一个数值计算程序的处理对象就是整型和浮点型数据。因此，数据的含义非常广泛，如整型、浮点型等数值类型及字符、声音、图像、视频等非数值数据都属于数据范畴。

2. 数据元素

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考虑和处理。一个数据元素可由若干个[数据项](#)

（data item）组成，数据项是数据不可分割的最小单位。例如，一个学校的教职工基本情况表包括编号、姓名、性别、籍贯、所在院系、

出生年月及职称等数据项。这里的数据元素也称为记录。教职工基本情况如表1-1所示。

表1-1 教职工基本情况

编 号	姓 名	性 别	籍 贯	所 在 院 系	出 生 年 月	职 称
05002	胡志刚	男	河南	化工学院	1970.10	教授
01156	张 琳	女	北京	文学院	1978.08	副教授
03527	李 燕	女	陕西	信息学院	1981.11	讲师

3. 数据对象

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，正整数数据对象是集合 $N=\{1, 2, 3, \cdots\}$ ，字母字符数据对象是集合 $C=\{ 'A' , ' B' , ' C' , \cdots\}$ 。

4. 数据结构

数据结构（data structure）即数据的组织形式，它是数据元素之间存在的一种或多种特定关系的数据元素集合。在现实世界中，任何事物都是有内在联系的，而不是孤立存在的，同样在计算机中，数据元素不是孤立的、杂乱无序的，而是具有内在联系的数据集合。例如，表1-1的教职工基本情况表是一种表结构，学校的组织机构是一种层次结构，城市之间的交通路线属于图结构，如图1-1和图1-2所示。

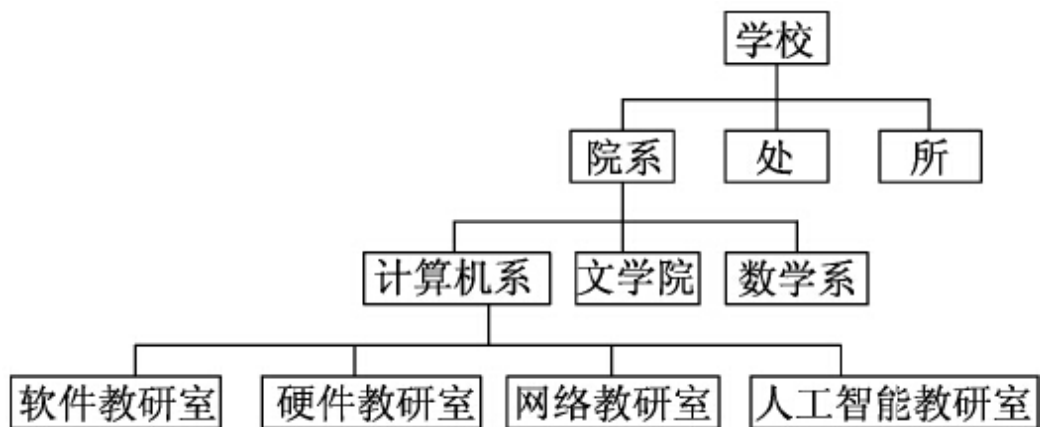


图1-1 学校组织机构图

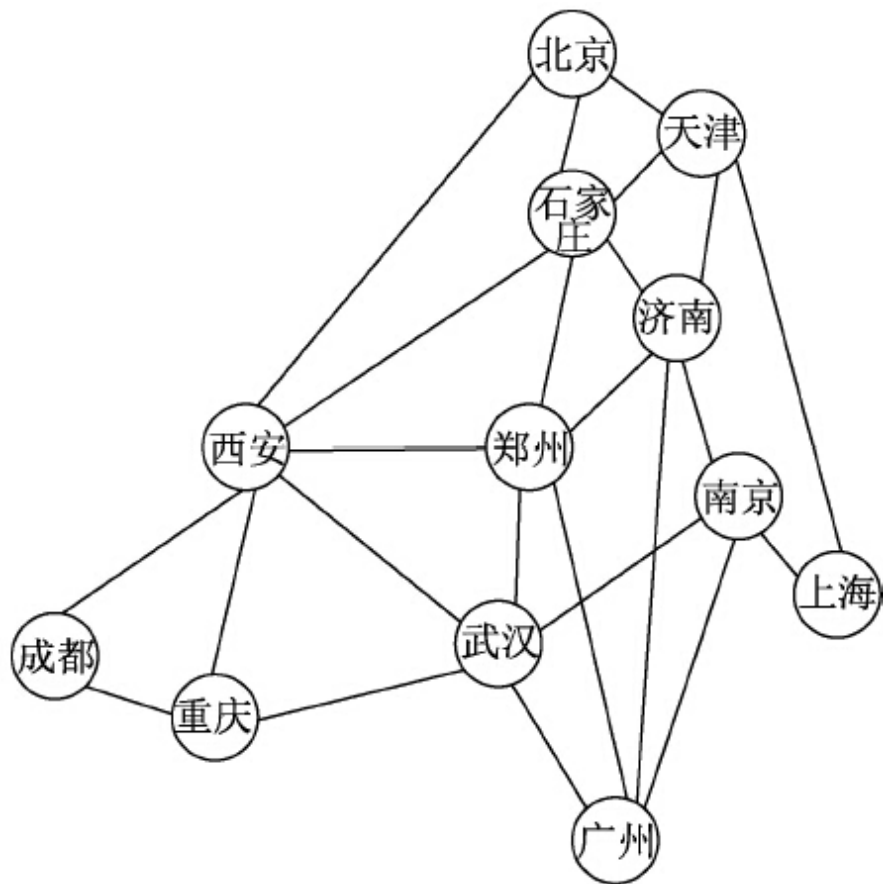


图1-2 城市之间交通路线图

5. 数据类型

数据类型（data type）用来刻画一组性质相同的数据及其上的操作。数据类型是按照值的不同进行划分的。在高级语言中，每个变量、常量和表达式都有各自的取值范围，该类型就说明了变量或表达式的取值范围和所能进行的操作。例如，C语言中的字符类型规定了所占空间是8位，也就是它的取值范围，同时也定义了在其范围内可以进行赋值运算、比较运算等。

在C语言中，按照取值的不同，数据类型还可以分为原子类型和结构类型两类。原子类型是不可以再分解的基本类型，包括整型、实型、字符型等。结构类型是由若干个类型组合而成，是可以再分解的。例如，整型数组是由若干整型数据组成的，结构体类型的值也是由若干个类型范围的数据构成，它们的类型都是相同的。

在计算机处理的发展历史上，计算机已经不仅仅能够处理数值信息了，计算机所能处理的对象包括数值、字符、文字、声音、图像及视频等信息。任何信息只要经过数字化处理，能够让计算机识别，都能够进行处理。当然，这需要对要处理的信息进行抽象描述，让计算机理解。

1.3 数据的逻辑结构与存储结构

数据结构的主要任务就是通过分析数据对象的结构特征，包括逻辑结构及数据对象之间的关系，然后把逻辑结构表示成计算机可实现的物理结构，从而便于计算机处理。本节主要介绍数据的逻辑结构表示和存储结构的表示。

1.3.1 逻辑结构

数据的**逻辑结构**（logical structure）是指在数据对象中数据元素之间的相互关系。数据元素之间存在不同的逻辑关系构成了以下4种结构类型。

（1）**集合**。结构中的数据元素除了同属于一个集合外，数据元素之间没有其他关系。这就像数学中的自然数集合，集合中的所有元素都属于该集合，除此之外，没有其他特性。例如，数学中的正整数集合{5, 67, 978, 20, 123, 18}，集合中的数除了属于正整数外，元素之间没有其他关系。数据结构中的集合关系就类似于数学中的集合。集合表示如图1-3所示。

（2）**线性结构**。结构中的数据元素之间是一一对应的关系。线性结构如图1-4所示。数据元素之间有一种先后的次序关系，a、b、c是

一个线性表，其中，a是b的前驱，b是a的后继。

(3) **树形结构**。结构中的数据元素之间存在一种一对多的层次关系，树形结构如图1-5所示。这就像学校的组织结构图，学校下面是教学的院系、行政机构及一些研究所。

(4) **图结构**。结构中的数据元素是多对多的关系，图1-6就是一个图结构。城市之间的交通路线图就是多对多的关系，a、b、c、d、e、f、g是7个城市，城市a和城市b、e、f都存在一条直达路线，而城市b也和a、c、f存在一条直达路线。

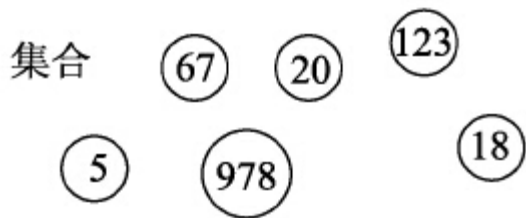


图1-3 集合结构

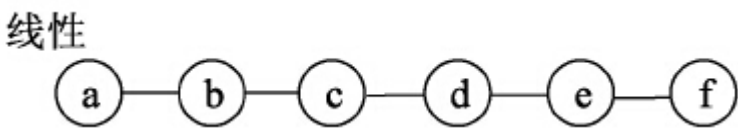


图1-4 线性结构

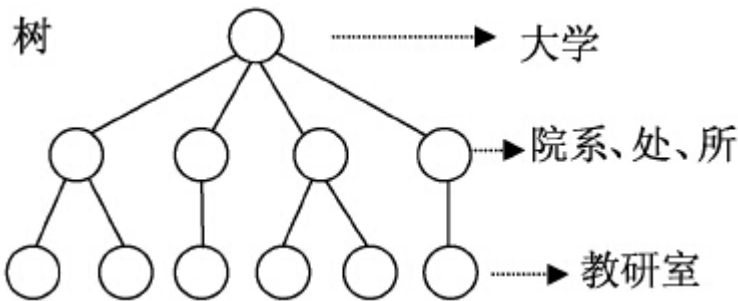


图1-5 树形结构

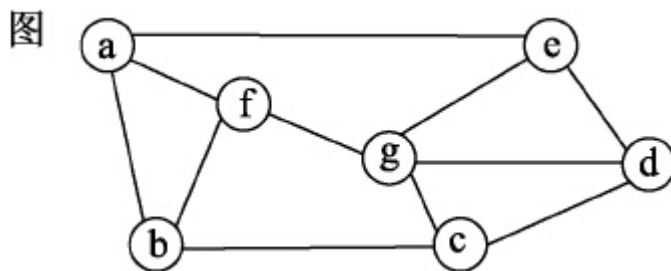


图1-6 图结构

1.3.2 存储结构

存储结构（storage structure）也称为**物理结构**（physical structure），指的是数据的逻辑结构在计算机中存储形式。数据的存储结构应能正确反映数据元素之间的逻辑关系。

数据元素的存储结构形式通常有顺序存储结构和链式存储结构两种。顺序存储是把数据元素存放在一组地址连续的存储单元里，其数据元素间的逻辑关系和物理关系是一致的。采用顺序存储的字符串“abcdef”的存储结构如图1-7所示。链式存储是把数据元素存放在任意的存储单元里，这组存储单元可以是连续的，也可以是不连续的，数据元素的存储关系并不能反映其逻辑关系，因此需要借助指针来表示数据元素之间的逻辑关系。字符串“abcdef”的链式存储结构如图1-8所示。

	⋮
0600	a
0601	b
0602	c
0603	d
0604	e
0605	f
	⋮

图1-7 顺序存储结构

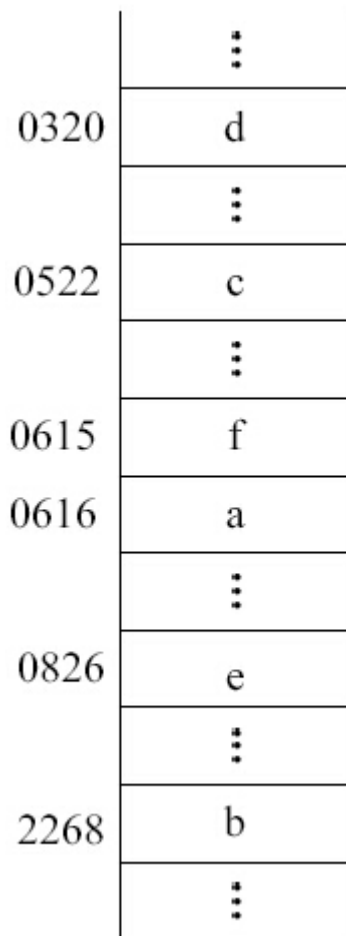


图1-8 链式存储结构

数据的逻辑结构和物理结构是密切相关的，今后在学习数据结构的过程中，读者将会发现，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所采用的存储结构。

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，但本书是建立在高级程序设计语言层次上的数据结构的操作，因此不能像上面那样直接以内存地址来描述存储结构，但可以借助高级程序设计语言中提供的数据类型来描述它，例如，可

以用C语言中的一维数组类型来描述顺序存储结构用C语言中的自引用类型描述链式存储结构，其中用指针来表示元素之间的逻辑关系。

1.4 抽象数据类型及其描述

为了在计算机上实现某种操作，需要把处理的对象描述成计算机能识别的形式，即一定形式的数据类型，并定义其上的一组操作。这在数据结构这门课中称为抽象数据类型。

1.4.1 什么是抽象数据类型

抽象数据类型（abstract data type, ADT）是描述具有某种逻辑关系的数学模型，并对在该数学模型上进行的一组操作。抽象数据类型描述的是一组逻辑上的特性，与在计算机内部表示无关。计算机中的整数数据类型是一个抽象数据类型，不同的处理器可能实现方法不同，但其逻辑特性相同，即加、减、乘、除等运算是一致的。在用户看来，各种计算机所提供的整数类型数学特性都是相同的，因此，“抽象”的意义在于数据类型的数学抽象特性，而不是它们的实现方法。

抽象数据类型不仅包括在计算机中已经定义了的数据类型，例如整型、浮点型等，还包括用户自己定义的数据类型，例如结构体类型、类等。

一个抽象数据类型定义了一个数据对象、数据对象中数据元素之间的关系及对数据元素的操作。抽象数据类型通常是指用来解决应用问题的数据模型，包括数据的定义和操作。

抽象数据类型体现了程序设计中的问题分解、抽象和信息隐藏特性。抽象数据类型把实际生活中的问题分解为多个规模小且容易处理的问题，然后建立起一个计算机能处理的数据模型，并把每个功能模块的实现细节作为一个独立的单元，从而使具体实现过程隐藏起来。这就类似人们日常生活中盖房子，把盖房子分成若干个小任务：地皮审批、图纸设计、施工、装修等，工程管理人员负责地皮的审批，地皮审批下来之后，工程技术人员根据用户需求设计图纸，建筑工人根据设计好的图纸进行施工（包括打地基、砌墙、安装门窗等），盖好房子后请装修工人装修。

盖房子的过程与抽象数据类型中的问题分解类似，工程管理人员不需要了解图纸如何设计，工程技术人员不需要了解打地基和砌墙的具体过程，装修工人不需要知道怎么画图纸和怎样盖房子，这就是抽象数据类型中的信息隐藏。

1.4.2 抽象数据类型的描述

对于初学者来说，抽象数据类型不太容易理解，用一大堆公式会让不少读者迷茫，因此，本书采用通俗的语言去讲解抽象数据类型。

本书把抽象数据类型分为两个部分来描述，即数据对象集合和基本操作集合。其中，数据对象集合包括数据对象的定义及数据对象中元素之间关系的描述，基本操作集合是对数据对象的运算的描述。数据对象和数据关系的定义可采用数学符号和自然语言描述，基本操作的定义格式如下。

基本操作名（参数表）：初始条件和操作结果描述。

例如线性表的抽象数据类型，描述如下。

1. 数据对象集合

线性表的数据对象集合为 $\{a_1, a_2, \dots, a_n\}$ ，每个元素的类型均为DataType。其中，除了第一个元素 a_1 外，每一个元素有且只有一个直接前驱元素，除了最后一个元素 a_n 外，每一个元素有且只有一个直接后继元素。数据元素之间的关系是一对一的关系。

2. 基本操作集合

线性表的基本操作如下所述。

(1) InitList (&L)：初始化操作，建立一个空的线性表L。这就像是在日常生活中，一所院校为了方便管理建立一个教职工基本情

况表，准备登记教职工信息。

(2) ListEmpty (L) : 若线性表L为空，返回1，否则返回0。这就像是刚刚建立了教职工基本情况表，还没有登记教职工信息。

(3) GetElem (L, i, &e) : 返回线性表L的第i个位置元素值给e。这就像在教职工基本情况表中，根据给定序号查找某个教师信息。

(4) LocateElem (L, e) : 在线性表L中查找与给定值e相等的元素，如果查找成功返回该元素在表中的序号表示成功，否则返回0表示失败。这就像在教职工基本情况表中，根据给定的姓名查找教师信息。

(5) InsertList (&L, i, e) : 在线性表L中的第i个位置插入新元素e。这就类似于经过招聘考试，引进了一名教师，这个教师信息被登记到教职工基本情况表中。

(6) DeleteList (&L, i, &e) : 删除线性表L中的第i个位置元素，并用e返回其值。这就像某个教职工到了退休年龄或者被调入其他学校，需要将该教职工从教职工基本情况表中删除。

(7) ListLength (L) : 返回线性表L的元素个数。这就像查看教职工基本情况表中有多少个教职工。

(8) ClearList (&L)：将线性表L清空。这就像学校被撤销，不需要再保留教职工基本信息，将这些教职工信息全部清空。

大多数教材用以下方式描述线性表的抽象数据类型。

```
ADT List
{
  数据对象: D={ai|ai
  ∈ElemSet
  , i=1
  , 2
  , ..., n
  , n
  ≥0}
  数据关系: R={<ai-1
  , ai>|ai-1
  , ai
  ∈D
  , i=2
  , 3
  , ..., n}
  基本操作如下。
  (1
  ) InitList
  (&L
  )
  初始条件: 表L
  不存在。
  操作结果: 建立一个空的线性表L
  。
  (2
  ) ListEmpty
  (L
  )
  初始条件: 表L
  存在。
  操作结果: 若表L
  为空, 返回1
  , 否则返回0
  。
  (3
  ) GetElem
  (L
  , i
  , &e
  )
  初始条件: 表L
  存在, 且i
  值合法, 即1
  ≤i
  ≤ListLength
  (L
  )
  )。
  操作结果: 返回表L
  的第i
  个位置元素值给e
  。
```

```

    (4
) LocateElem
    (L
, e
)
初始条件: 表L
存在, 且e
为合法元素值。
操作结果: 在表L
中查找与给定值e
相等的元素。如果查找成功, 则返回该元素在表中的序号; 如果这样的元素不存在, 则返回0
。
    (5
) InsertList
    (&L
, i
, e
)
初始条件: 表L
存在, e
为合法元素且1
≤i
≤ListLength
(L
)。
操作结果: 在表L
中的第i
个位置插入新元素e
。
    (6
) DeleteList
    (&L
, i
, &e
)
初始条件: 表L
存在且1
≤i
≤ListLength
(L
)。
操作结果: 删除表L
中的第i
个位置元素, 并用e
返回其值。
    (7
) ListLength
    (L
)
初始条件: 表L
存在。
操作结果: 返回表L
的元素个数。
    (8
) ClearList
    (&L
)
初始条件: 表L
存在。
操作结果: 将表L
清空。
}ADT List

```

以上是抽象数据类型的两种描述方式，显然前者会更容易被理解和接受。

需要注意的是，在基本操作的描述过程中，参数传递有两种方式，一种是数值传递，另一种是引用传递。前者仅仅是将数值传递给形参，并不返回结果；后者其实是把实参的地址传递给形参，实参和形参其实是同一个变量，被调用函数通过修改该变量的值返回给调用函数，从而把结果带回。在描述算法时，在参数前加上&表示引用传递；如果参数前没有&，表示是数值传递。

1.5 算法

在数据类型建立起来之后，就要对这些数据类型进行操作，建立起运算的集合即程序。运算的建立、方法好坏直接决定着计算机程序运行效率的高低。如何建立一个比较好的运算集合就是算法要研究的问题。本节主要介绍算法的定义、算法的特性、算法的描述及算法与数据结构的关系。

1.5.1 数据结构与算法的关系

算法与数据结构关系密切。两者既有联系又有区别。

数据结构与算法的联系可用一个公式描述，即程序=算法+数据结构。数据结构是算法实现的基础，算法总是要依赖于某种数据结构来实现的。算法的操作对象是数据结构。算法的设计和选择要同时结合数据结构，简单地说，数据结构的设计就是选择存储方式，如确定问题中的信息是用普通变量存储还是用数组存储或其他更加复杂的数据结构存储。算法设计的实质就是对实际问题要处理的数据选择一种恰当的存储结构，并在选定的存储结构上设计一个好的算法。

数据结构是算法设计的基础。用一个形象的比喻来解释就是，开采煤矿过程中，煤矿以各种形式深埋于地下，矿体的结构就相当于计

计算机领域的数据结构，而煤就相当于一个个数据元素；开采煤矿然后运输、加工这些“操作”技术就相当于算法；显然，如何开采、如何运输必须考虑到煤矿的存储（物理）结构，只拥有开采技术而没有煤矿是没有任何意义的。算法设计必须要考虑到数据结构的构造，算法设计是不可能独立于数据结构存在的。另外，数据结构的设计和选择需要为算法服务。如果某种数据结构不利于算法实现它将没有太大的实际意义。知道某种数据结构的典型操作才能设计出好的算法。总之，算法的设计同时伴有数据结构的设计，两者都是为最终解决问题服务的。

数据结构与算法的区别在于数据结构关注的是数据的逻辑结构、存储结构以及基本操作，而算法更多的是关注如何在数据结构的基础上解决实际问题。算法是编程思想，数据结构则是这些思想的基础。

1.5.2 什么是算法

算法（algorithm）是解决特定问题求解步骤的描述，在计算机中表现为有限的操作序列。操作序列包括了一组操作，每一个操作都完成特定的功能。例如求n个数中最大者的问题，其算法描述如下。

（1）定义一个变量max和一个数组a[]，分别用来存放最大数和数组的元素，并假定第一个数最大，赋给max。

```
max=a[0]  
;
```

(2) 依次把数组a中其余的n-1个数与max进行比较，遇到较大的数时，将其赋给max。

```
for(i=1;i<n;i++)  
    if(max<a[i])  
        max=a[i];
```

最后，max中的数就是n个数中的最大者。

1.5.3 算法的五大特性

算法具有以下5个特性。

(1) **有穷性**。有穷性指的是算法在执行有限的步骤之后，自动结束而不会出现无限循环，并且每一个步骤在可接受的时间内完成。

(2) **确定性**。算法的每一步骤都具有确定的含义，不会出现二义性。算法在一定条件下只有一条执行路径，也就是相同的输入只能有一个唯一的输出结果。

(3) **可行性**。算法的每一步都必须是可行的，也就是说，每一步都能够通过执行有限次数完成。

(4) **输入**。算法具有零个或多个输入。

(5) **输出**。算法至少有一个或多个输出。输出的形式可以是打印输出也可以是返回一个或多个值。

1.5.4 算法的描述

算法的描述方式有多种，如自然语言、伪代码（或称为类语言）、程序流程图及程序设计语言（如C语言）等。其中，自然语言描述可以是汉语或英语等文字描述；伪代码形式类似于程序设计语言形式，但是不能直接运行；程序流程图的优点是直观，但是不易直接转化为可运行的程序；程序设计语言形式是完全采用像C、C++、Java等语言描述，可以直接在计算机上运行。

例如判断正整数 m 是否为质数，算法可用以下几种方式描述。

1. 自然语言描述法

利用自然语言描述“判断 m 是否为质数”的算法如下。

①输入正整数 m ，令 $i=2$ 。

②如果 $i \leq \sqrt{m}$ ，则令 m 对 i 求余，将余数送入中间变量 r ；否则输出“ m 是质数”，算法结束。

③判断 r 是否为零。如果为零，输出“ m 不是质数”，算法结束；如果 r 不为零，则令 i 增加1，转到步骤②执行。

因为上述算法采用自然语言描述不具有直观性和良好的可读性，采用程序流程图描述比较直观，可读性好，但是不能直接转化为计算机程序，移植性不好。

2. 程序流程图法

判断 m 是否为质数的程序流程图如图1-9所示。我们采用类C语言描述和C语言描述如下：

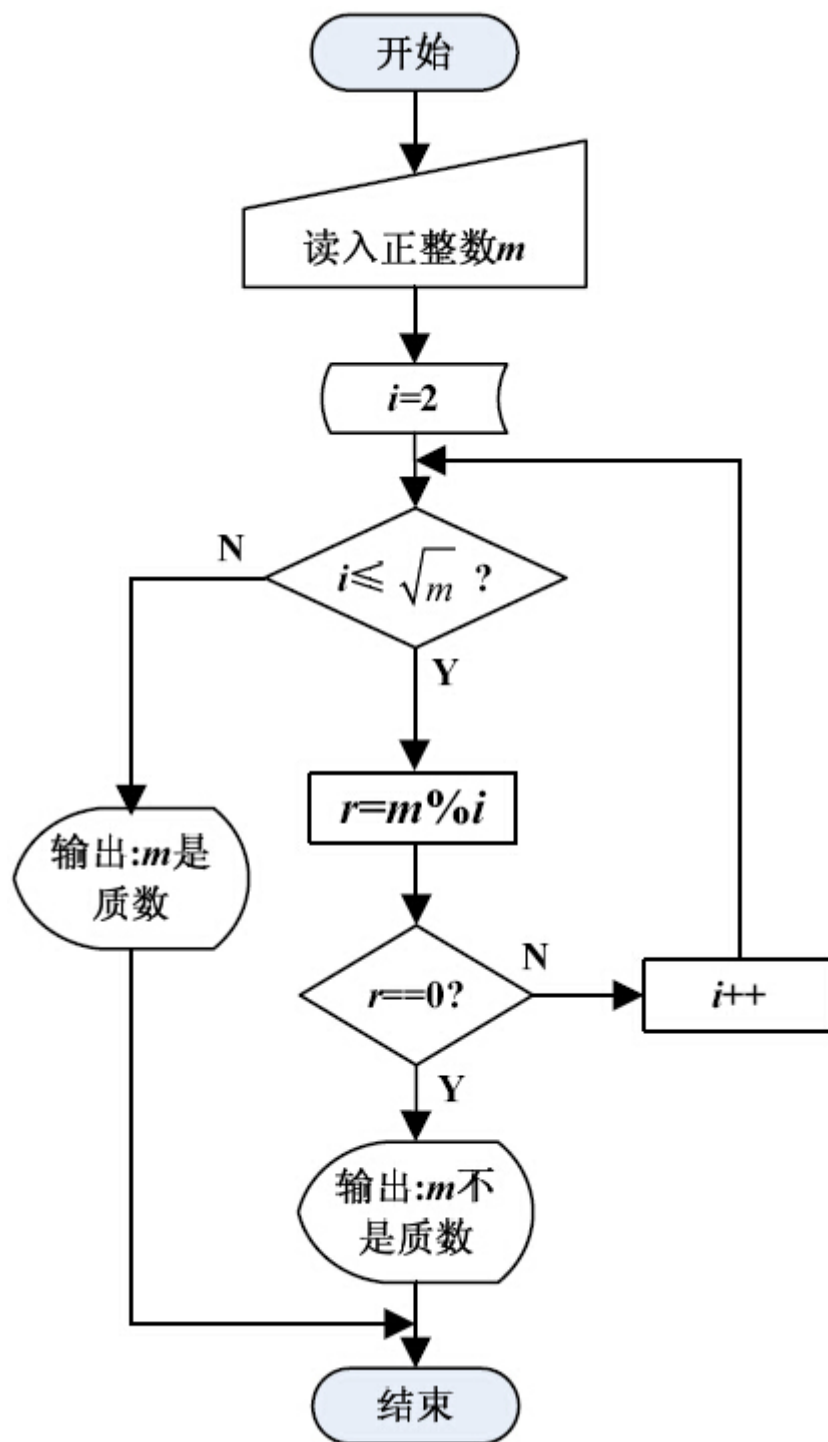


图1-9 判断 m 是否为质数的程序流程图

3. 类语言法

类C语言描述如下。

```
void IsPrime()
/*
判断m
是否为质数*/
{
    scanf(m);          /*
输入正整数m*/
    for(i=2;i<=sqrt(m);i++)
    {
        r=m%i;          /*
求余数*/
        if(r==0)          /*
如果m
能被整除*/
        {
            printf("m
不是质数! ");
            break;
        }
    }
    printf("m
是质数! ");
}
```

4. 程序设计语言法

C语言描述如下。

```
void IsPrime()
/*
判断m
是否为质数*/
{
    printf("
请输入一个正整数: ");
    scanf("%d",&m);          /*
输入正整数m*/
    for(i=2;i<=sqrt(m);i++)
    {
        r=m%i;          /*
求余数*/
        if(r==0)          /*
如果m
能被整除*/
        {
            printf("m
不是质数! \n");
            break;
        }
    }
}
```

```
printf("m  
是质数! \n");  
}
```

可以看出，类语言的描述除了没有变量的定义，输入和输出的写法之外，与程序设计语言的描述的差别不大，类语言的可以直接转化为可以直接运行的计算机程序。

为了方便读者学习和上机操作，本书所有算法均采用C程序设计语言描述，能直接上机运行。

1.6 算法分析

一个好的算法往往可以使程序运行的更快，衡量一个算法的好坏往往将算法效率和存储空间作为重要依据。算法的效率需要通过算法编制的程序在计算机上的运行时间来衡量，存储空间需求通过算法在执行过程中所占用的最大存储空间来衡量。本节主要介绍算法设计的要求、算法效率评价、算法的时间复杂度及算法的空间复杂度。

1.6.1 算法设计的4个目标

一个好的算法应该具备以下目标。

1. 算法的正确性

算法的**正确性**（correctness）是指算法至少应该包括对于输入、输出和加工处理无歧义性的描述，能正确反映问题的需求，且能够得到问题的正确答案。

通常算法的正确性应包括以下4个层次。

- （1）算法对应的程序没有语法错误。
- （2）对于几组输入数据能得到满足规格要求的结果。

(3) 对于精心选择的典型的、苛刻的带有刁难性的几组输入数据能得到满足规格要求的结果。

(4) 对于一切合法的输入都能得到产生满足要求的结果。

对于这4层算法正确性的含义，达到第4层意义上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。一般情况下，我们把层次3作为衡量一个程序是否正确的标准。

2. 可读性

算法主要是为了人们方便阅读和交流，其次才是计算机执行。可读性（readability）好有助于人们对算法的理解，晦涩难懂的程序往往隐含错误不易被发现，难以调试和修改。

3. 健壮性

当输入数据不合法时，算法也应该能做出反应或进行处理，而不会产生异常或莫名其妙的输出结果。例如，求一元二次方程根 $ax^2 + bx + c = 0$ 的算法，需要考虑多种情况，先判断 $b^2 - 4ac$ 的正负，如果为正数，则该方程有两个不同的实根；如果为负，表明该方程无实根；如果为零，表明该方程只有一个实根；如果 $a=0$ ，则该方程又变成了一

元一次方程，此时若 $b=0$ ，还要处理除数为零的情况。如果输入的 a 、 b 、 c 不是数值型，还要提示用户输入错误。

4. 高效率和低存储量

效率指的是算法的执行时间。对于同一个问题如果有多个算法能够解决，执行时间短的算法效率高，执行时间长的效率低。存储量需求指算法在执行过程中需要的最大存储空间。效率与低存储量需求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和运行空间显然有一定差别。设计算法时应尽量选择高效率和低存储量需求的算法。

1.6.2 算法效率评价

算法执行时间需通过依据该算法编制的程序在计算机上的运行时所耗费的时间来度量，而度量一个算法在计算机上的执行时间通常有如下两种方法。

1. 事后统计方法

目前计算机内部大都有计时功能，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测试程序和数据以分辨算法

的优劣。但是，这种方法有两个缺陷，一是必须依据算法事先编制好程序，这通常需要花费大量的时间与精力；二是时间的长短依赖计算机硬件和软件等环境因素，有时会掩盖算法本身的优劣。因此，人们常常采用事前分析估算的方法评价算法的好坏。

2. 事前分析估算方法

这主要在计算机程序编制前，对算法依据数学中的统计方法进行估算。这主要是因为算法的程序在计算机上的运行时间取决于以下因素。

- 算法采用的策略、方法。
- 编译产生的代码质量。
- 问题的规模。
- 书写的程序语言，对于同一个算法，语言级别越高，执行效率越低。
- 机器执行指令的速度。

在以上5个因素中，算法采用的不同的策略，或不同的编译系统，或不同的语言实现，或在不同的机器运行时，效率都不相同。抛开以上因素，算法效率则可以通过问题的规模来衡量。

一个算法由控制结构（顺序、分支和循环结构）和基本语句（赋值语句、声明语句和输入输出语句）构成，则算法的运行时间取决于两者执行时间的总和，所有语句的执行次数可以作为语句的执行时间的度量。语句的重复执行次数称为语句频度。

例如，斐波那契数列的算法和语句的频度如下。

每一条语句的频度	
f0=0;	1
f1=1;	1
printf("%d,%d",f0,f1);	1
for(i=2;i<=n;i++)	n
{	
fn=f0+f1;	n-1
printf(",%d",fn);	n-1
f0=f1;	n-1
f1=fn;	n-1
}	

每一条语句的右端是对应语句的频度（frequency count），即语句的执行次数。上面算法总的执行次数为 $T(n) = 1 + 1 + 1 + n + 4(n-1) = 5n - 1$ 。

1.6.3 算法的时间复杂度

在进行算法分析时，语句总的执行次数 $T(n)$ 是关于问题规模 n 的函数，进而分析 $T(n)$ 随 n 的变化情况并确定 $T(n)$ 的数量级。算法的时间复杂度，也就是算法的时间量度，记作 $T(n) = O(f(n))$ 。

它表示随问题规模 n 的增大，算法的执行时间的增长率和 $f(n)$ 的增长率相同，称作算法的**渐进时间复杂度**（asymptotic time complexity），简称为**时间复杂度**。其中， $f(n)$ 是问题规模 n 的某个函数。

一般情况下，随着 n 的增大， $T(n)$ 的增长较慢的算法为最优的算法。例如，在下列三段程序段中，给出原操作 $x=x+1$ 的时间复杂度分析。

(1)

```
1  
;
```

(2)

```
for  
  (i=1  
  ; i<=n  
  ; i++  
  )  
    x=x+1  
;
```

(3)

```
for  
  (i=1  
  ; i<=n  
  ; i++  
  )  
  for  
    (j=1  
    ; j<=n  
    ; j++
```

```
)  
    x=x+1  
;
```

程序段（1）的时间复杂度为 $O(1)$ ，称为常量阶；程序段（2）的时间复杂度为 $O(n)$ ，称为线性阶；程序段（3）的时间复杂度为 $O(n^2)$ ，称为平方阶。此外算法的时间复杂度还有对数阶 $O(\log_2 n)$ 、指数阶 $O(2^n)$ 等。

上面的斐波那契数列的时间复杂度 $T(n) = O(n)$ 。

常用的时间复杂度所耗费的时间从小到大依次是 $O(1) < O(\log_2 n) < O(n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$ 。

算法的时间复杂度是衡量一个算法好坏的重要指标。一般情况下，具有指数级的时间的复杂度算法只有当 n 足够小时才是可使用的算法。具有常量阶、线性阶、对数阶、平方阶和立方阶的时间复杂度算法是常用的算法。一些常见函数的增长率如图1-10所示。

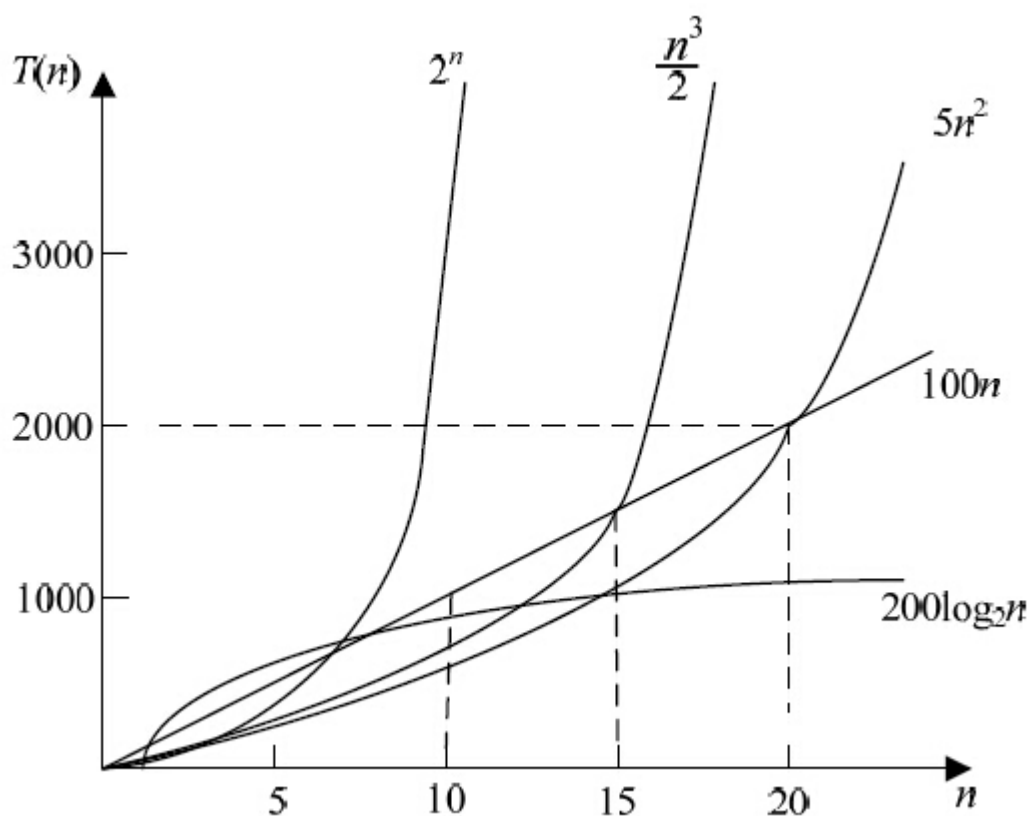


图1-10 常见函数的增长率

一般情况下，算法的时间复杂度只需要考虑关于问题规模 n 的增长率或阶数。例如以下程序段。

```

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

```

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

在有些情况下，算法的基本操作的重复执行次数还依赖于输入的数据集。例如如下冒泡排序算法。

```
void BubbleSort(int a[],int n)
{
    int i,j,t;
    change=TRUE;
    for(i=1;i<=n-1&&change;i++)
    {
        change=FALSE;
        for(j=1;j<=n-i;j++)
            if(a[j]>a[j+1])
            {
                t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
                change=TRUE;
            }
    }
}
```

交换相邻两个整数为该算法中的基本操作。当数组a中的初始序列为从小到大有序排列时，基本操作的执行次数为0；当数组中初始序列从大到小排列时，基本操作的执行次数为 $n(n-1)/2$ 。对这类算法的分析，一种方法是计算所有情况的平均值，这种时间复杂的计算方法称为平均时间复杂度；另外一种方法是计算最坏情况下的时间复杂度，这种方法称为最坏时间复杂度。若数组a中初始输入数据可能出现 $n!$ 种的排列情况的概率相等，则冒泡排序的平均时间复杂度为 $T(n) = O(n^2)$ 。

然而，在很多情况下，各种输入数据集出现的概率难以确定，算法的平均复杂度也就难以确定。因此，另一种更可行也更为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

法执行时间的上界。例如，上面冒泡排序的最坏时间复杂度为数组a中初始序列为从大到小有序，则冒泡排序算法在最坏情况下的时间复杂度为 $T(n) = O(n^2)$ 。一般情况下，本书以后讨论的时间复杂度，在没有特殊说明情况下，都指的是最坏情况下的时间复杂度。

1.6.4 算法的空间复杂度

空间复杂度 (space complexity) 作为算法所需存储空间的量度，记做 $S(n) = O(f(n))$ 。其中， n 为问题的规模， $f(n)$ 为语句关于 n 的所占存储空间的函数。一般情况下，一个程序在机器上执行时，除了需要存储程序本身的指令、常数、变量和输入数据外，还需要存储对数据操作的存储单元。若输入数据所占空间只取决于问题本身，和算法无关，这样只需要分析该算法在实现时所需的辅助单元即可。若算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法为原地工作，空间复杂度为 $O(1)$ 。

1.7 学好数据结构的秘诀

作为计算机专业的一名“老兵”，笔者个人学习、研究数据结构和算法已经近10年了，在学习的过程中，也遇到不少问题，为了让读者在学习数据结构的过程中少走弯路，下面分享一下笔者个人的一些经验，谈谈关于如何学好“数据结构”的一些粗浅认识。

1. 明确数据结构的重要性，树立学好数据结构的信心

数据结构是计算机科学与技术专业的核心课程，不仅仅涉及计算机硬件的研究范围，并且与计算机软件的研究有着更为密切的关系，“数据结构”课程还是操作系统、数据库原理、编译原理、人工智能、算法设计与分析等课程的基础。数据结构是计算机专业硕士研究生入学考试的必考科目之一，还是计算机软件水平考试、等级考试的必考内容之一，数据结构在计算机专业中的重要性不言而喻。

万事开头难，学习任何一样新东西，都是比较困难的，对于初学者而已，数据结构的确是一门不容易掌握的专业基础课，但你一定要树立学好数据结构的信心，主要困难无非有两个，一个是数据结构的概念比较抽象，不容易理解；另一个是没有熟练掌握一门程序设计语

言。面对以上困难，只要我们见招拆招，其实也没有什么可怕的，不过选择一本好教材是十分有必要的。

2. 熟练掌握程序设计语言，变腐朽为神奇

程序语言是学习数据结构和算法设计的基础，很显然，没有良好的程序设计语言能力，就不能很好地把算法用程序设计语言描述出来，程序设计语言和数据结构、算法的关系就像是画笔和画家的思想关系一样，程序设计语言就是这画笔，数据结构、算法就是画家的思想，即便画家的水平很高，如果不会使用画笔，再美的图画也无法给我们展现出来。

可见，要想学好数据结构，必须至少熟练掌握一门程序设计语言，如C语言、C++语言等。

3. 结合生活实际，变抽象为具体

数据结构是一项把实际问题抽象化和进行复杂程序设计的工程。它要求学生不仅具备C语言等高级程序设计语言的基础，而且还要学会掌握把复杂问题抽象成计算机能够解决的离散的数学模型的能力。在学习数据结构的过程中，要将各种结构与实际生活结合起来，把抽象的东西具体化，以便理解。例如学到队列时，很自然就会联想到火车

站售票窗口前面排起的长长的队伍，这支长长的队伍其实就是队列的具体化，这样就会很容易理解关于队列的概念，如队头、队尾、出队、入队等。

4. 多思考，多上机实践

数据结构既是一门理论性较强的学科，也是一门实践性很强的学科。特别是对于初学者而言，接触到的算法相对较少，编写算法还不够熟练，俗话说“熟能生巧，勤能补拙”，因此，只有多看有关算法和数据结构方面的图书，认真理解其中的算法思想，除了阅读算法之外，还要自己动手写算法，并在电脑上上机调试，这样才能知道编写的算法是否正确，存在哪些错误和缺陷，以避免今后再犯类似错误，长此以往，自己的算法和数据结构水平才能快速提高。

有的表面上看是正确的程序，在电脑上运行后才发现隐藏的错误，特别是很细微的错误，只有多试几组数据，才知道程序到底是不是正确。因此，对于一个程序或算法，除了仔细阅读程序或算法判断是否存在逻辑错误外，还需要上机调试，在可能出错的地方设置断点，单步跟踪调试程序，观察各变量的变化情况，才能找到具体哪个地方出了问题。有时，可能仅仅是误敲了一个符号或变量，就可能产生错误，这种错误往往不容易发现，只有上机调试才能知道。因此，在学习数据结构与算法的时候一定要多上机实践。

只要能做到以上几点，选择一本好的数据结构教材或参考书（最好算法完全用C语言实现，有完整代码），加上读者的勤奋，学好数据结构自然不在话下。

1.8 习题

1. 研究数据结构就是研究（ ）。

- A. 数据的逻辑结构
- B. 数据的存储结构
- C. 数据的逻辑结构和存储结构
- D. 数据的逻辑结构、存储结构及其基本操作

2. 算法分析的两个主要方面是（ ）。

- A. 空间复杂度和时间复杂度
- B. 正确性和简单性
- C. 可读性和文档性
- D. 数据复杂性和程序复杂性

3. 具有线性的数据结构是（ ）。

- A. 图
- B. 树

C. 广义表

D. 栈

4. 计算机中的算法指的是解决某一个问题的有限运算序列，它必须具备输入、输出、（ ）等5个特性。

A. 可执行性、可移植性和可扩充性

B. 可执行性、有穷性和确定性

C. 确定性、有穷性和稳定性

D. 易读性、稳定性和确定性

5. 下面程序段的时间复杂度是（ ）。

```
for (i=0; i<m; i++)  
    for (j=0; j<n; j++)  
        a[i][j]=i*j;
```

A. $O(m^2)$

B. $O(n^2)$

C. $O(m*n)$

D. $O(m+n)$

6. 算法是（ ）。

A. 计算机程序

B. 解决问题的计算方法

C. 排序算法

D. 解决问题的有限运算序列

7. 某算法的语句执行频度为 $(3n + n \log_2 n + n^2 + 8)$ ，其时间复杂度表示（ ）。

A. $O(n)$

B. $O(n \log_2 n)$

C. $O(n^2)$

D. $O(\log_2 n)$

8. 下面程序段的时间复杂度为（ ）。

```
i=1;
while(i<=n)
    i=i*3;
```

A. $O(n)$

B. $O(3n)$

C. $O(\log_3 n)$

D. $O(n^3)$

9. 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数据元素以及它们之间的（）和运算等的学科。

A. 结构

B. 关系

C. 运算

D. 算法

10. 下面程序段的时间复杂度是（）。

```
i=s=0;
while(s<n){
    i++;s+=i;
}
```

A. $O(n)$

B. $O(n^2)$

C. $O(\log_2 n)$

D. $O(n^3)$

11. 通常从正确性、易读性、健壮性、高效性等4个方面评价算法的质量，以下解释错误的是（ ）。

A. 正确性算法应能正确地实现预定的功能

B. 易读性算法应易于阅读和理解，以便调试、修改和扩充

C. 健壮性当环境发生变化时，算法能适当地做出反应或进行处理，不会产生不需要的运行结果

D. 高效性即达到所需要的时间性能

第2章 C语言基础

C语言作为数据结构的算法描述语言，广泛应用于系统软件和应用软件的开发。在真正开始学习数据结构知识之前，笔者先带领读者复习C语言中的一些重点和难点，为数据结构的学习扫清障碍。本章主要针对C语言的重点和难点部分进行了细致讲解，主要内容包括C语言开发环境、函数与递归、指针、参数传递、动态内存分配及结构体、联合体。

本章重点和难点：

- 递归函数的实现和递归如何转化为非递归
- 指针数组、数组指针、函数指针及理解与使用
- 理解传地址调用中变量的变化情况
- 链表的操作

2.1 C语言开发环境

C语言常见的开发环境有多种，如LCC、Turbo C 2.0、Visual C++、Borland C++，本节主要介绍使用最为广泛的Turbo C 2.0和Visual C++6.0。

2.1.1 Turbo C 2.0开发环境

1989年，美国Borland公司推出了Turbo C 2.0（Turbo C简称为TC），它是集编辑、编译、链接和运行于一体的C程序集成开发环境。Turbo C 2.0界面简单，上手容易，使用方便，通过一个简单的主界面可以很容易编辑、编译和链接程序，正所谓“麻雀虽小，五脏俱全”，它是初学者广泛使用的开发工具。

1. 进入Turbo C 2.0环境

在Windows系统中，运行Turbo C 2.0有两种方式，一是直接双击Turbo C 2.0文件夹下的文件TC.EXE运行；二是切换到DOS命令行下通过命令行进入。下面重点介绍命令行方式如何进入Turbo C 2.0开发环境，具体步骤如下。

①假如Turbo C 2.0编译程序安装在D:\TC目录下，那么需要单击【开始】|【运行】命令，打开【运行】对话框，在文本框中输入“command”或者“cmd”，如图2-1所示。

②单击【确定】按钮，进入DOS运行环境，如图2-2所示。

③输入“d:”后按【Enter】键，切换到盘符“D:\”目录。接着输入“cd tc”命令后按【Enter】键，进入“D:\tc”目录，如图2-3所示。

④输入“tc”后按【Enter】键，进入Turbo C 2.0集成开发环境主界面，如图2-4所示。



图2-1 “运行”对话框

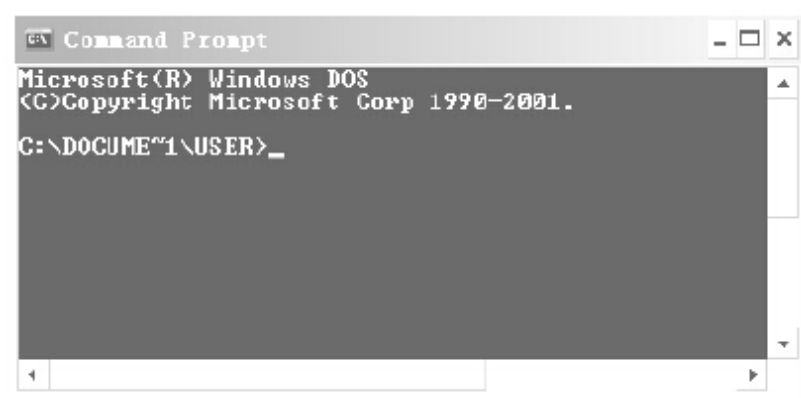


图2-2 进入DOS运行环境



图2-3 DOS命令行操作示意图



图2-4 Turbo C 2.0集成开发环境主界面

从图2-4可以看到，Turbo C 2.0集成开发环境的主菜单包括【File】、【Edit】、【Run】、【Compile】、【Project】、【Options】、【Debug】和【Break/watch】8个菜单项。这8个菜单项分别代表的是【文件】、【编辑】、【运行】、【编译】、【工程】、【选项】、【调试】和【中断/查看】。

2. 设置Turbo C 2.0的运行环境

在Turbo C 2.0的运行环境中，不能使用鼠标，只能使用键盘。通过按键盘上的【F10】键先激活菜单，然后通过按四个方向键【↑】、【↓】、【←】、【→】对菜单和菜单项进行选择，被选中的菜单项以黑底白字显示。

在使用Turbo C 2.0之前，需要对该运行环境进行一些相关的设置。在C语言程序中，经常会调用一些系统函数，而这些函数存在于Turbo C 2.0的系统文件中，因此，只有设置好了系统文件的路径，才能正确运行C语言程序，否则就会出现编译错误。

在使用Turbo C 2.0之前，首先需要修改Turbo C 2.0的包含文件目录和库文件目录为当前文件所在的目录，其操作步骤如下。

①按【F10】键激活【Options】菜单，并按【Enter】键打开下拉菜单，然后按【↓】方向键选中【Directories】命令。

②按【Enter】键，进入下一级菜单，然后选中【Include directories】命令，按【Enter】键打开【Include directories】对话框，将默认的路径“C:\TURBOC2\INCLUDE”修改为“D:\TC\INCLUDE”。如图2-5所示。

③按【Enter】键返回上一级菜单，按【↓】方向键，选中【Library directories】命令，按【Enter】键打开【Library directories】对话框，将默认的路径“C:\TURBOC2\LIB”修改为“D:\TC\LIB”，并按【Enter】键。

④为了在下次使用Turbo C 2.0时不再进行以上修改，需要保存上面的环境设置。这需要去掉Turbo C 2.0配置文件“TCCONFIG.TC”的只读属性，否则无法保存以上设置。执行【Options】|【Save options】命令，并按【Enter】键，打开【Config File】对话框，然后按【Enter】键弹出【Verify】提示对话框，直接输入“Y”即可完成修改。

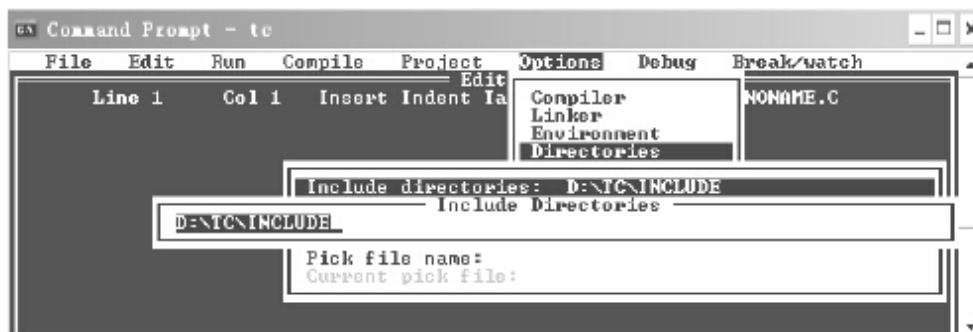


图2-5 设置“包含头文件”的路径

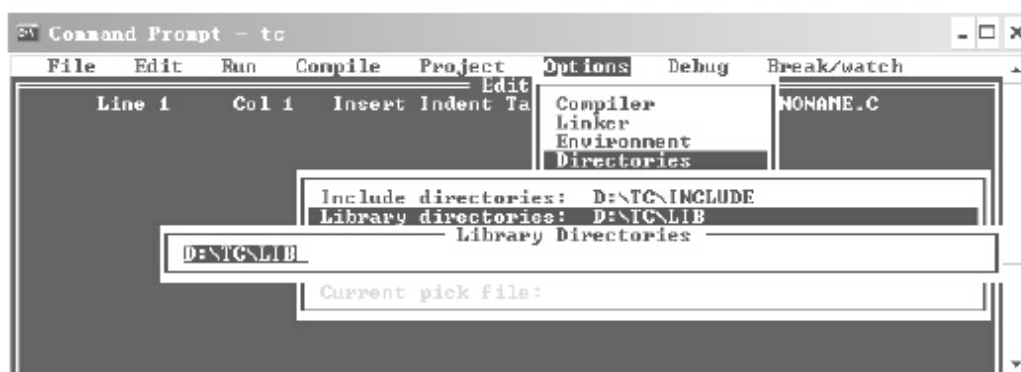


图2-6 设置库文件路径

3. 使用Turbo C 2.0编写C语言程序

按【F10】键激活菜单，并选中【Edit】菜单后按【Enter】键，输入以下代码。

```
void main()
{
    printf("hello world!\n");
}
```

按【F10】激活菜单，执行【Compile】|【Compile to OBJ】命令，对程序进行编译，然后执行【Compile】|【Link EXE file】命令，链接程序，如果程序没有错误，就会生成.EXE文件。

这时，我们就可以执行【Run】|【Run】命令或者按快捷键【Ctrl+F9】运行程序，最后通过执行【Run】|【Use Screen】命令或者按快捷键【Alt+F5】查看运行结果，如图2-7所示。

这时，如果按【Esc】键，就会返回Turbo C 2.0集成开发环境。

如果程序中有错误，在编译或链接时就会出现错误提示信息，如图2-8是一个链接错误的提示信息。按任意键，会以不同颜色将光标定位到错误行，用户可以根据系统提示的错误原因修改错误，并重新编译、链接运行程序。

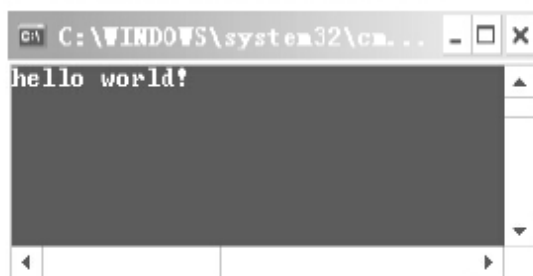


图2-7 程序运行结果

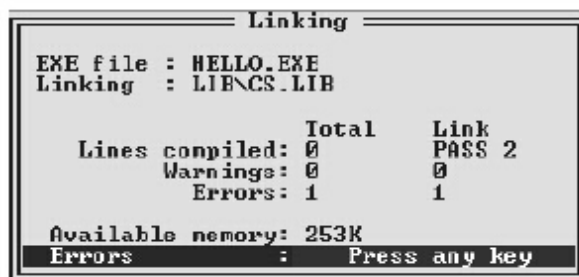


图2-8 链接错误的提示信息

其他常见操作，如执行【File】|【Load】命令和【File】|【Pick】命令，用于加载存在磁盘中的文件；执行【File】|【Save】、【File】|【Save as】和【File】|【Write to】命令，用于将程序保存在磁盘上；执行【File】|【Quit】命令或按快捷键【Alt+X】，可以退出Turbo C 2.0集成开发环境。

2.1.2 Visual C++6.0开发环境

Visual C++6.0是强大的C/C++软件开发工具，使用非常广泛，已经成为程序员首选的开发工具。利用它不仅可以开发控制台应用程序，还可以开发Windows SDK、MFC等应用程序。因为本书主要讲解利用C语言描述算法，所以仅介绍如何使用Visual C++6.0开发控制台程序。

用Visual C++6.0编写C语言程序需要如下两个步骤，一是创建一个新的Visual C++6.0控制台项目；二是在该项目中创建C程序文件，并编辑C语言程序。

1. 新建Visual C++6.0控制台项目

①执行【文件】|【新建】命令，弹出【新建】对话框，单击【工程】标签，在项目文件选择【Win32 Console Application】命令，在“工程名称”文本框中输入项目名称“Test”，单击【位置】按钮，选择保存路径为“D:\C程序\Test”，如图2-9所示。

②单击【确定】按钮，打开【Win32 Console Application】对话框，即新建控制台项目的第一步。这一步有4个选项，选择【一个空工程】选项，就是要建立一个空项目，如图2-10所示。



图2-9 新建一个控制台项目

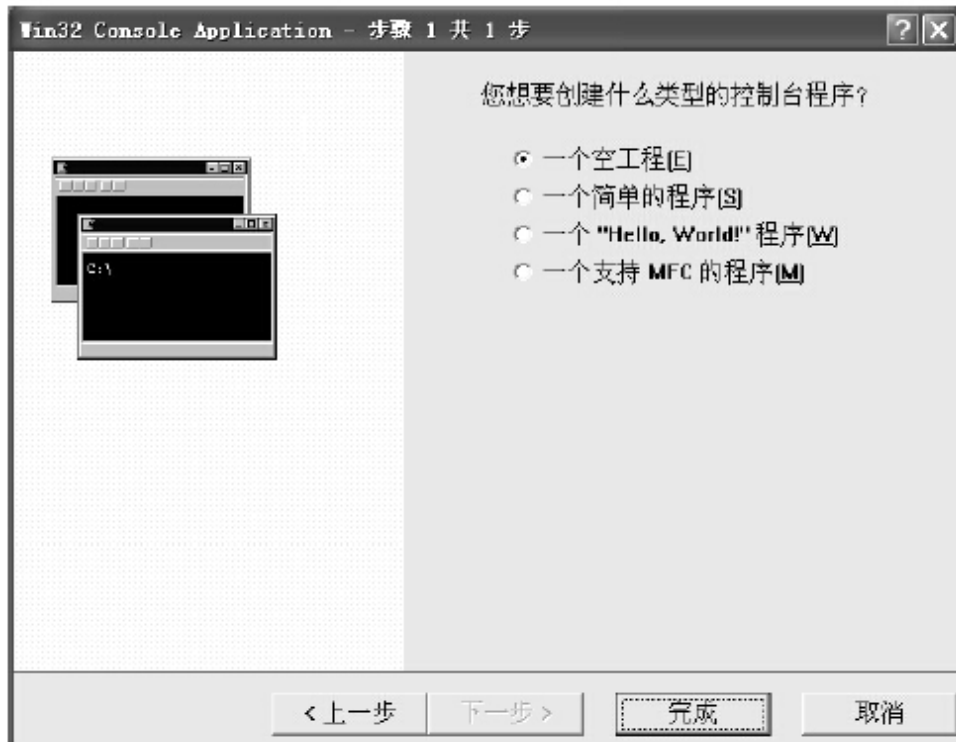


图2-10 控制台应用程序设置

③单击【完成】按钮，打开【新建工程信息】对话框，即新建立的控制台项目信息：已经建立了一个空项目，且项目中没有文件。该项目路径为“D:\C程序\Test”，如图2-11所示。

④单击【确定】按钮，进入控制台的项目环境中，如图2-12所示。



图2-11 控制台应用程序提示信息

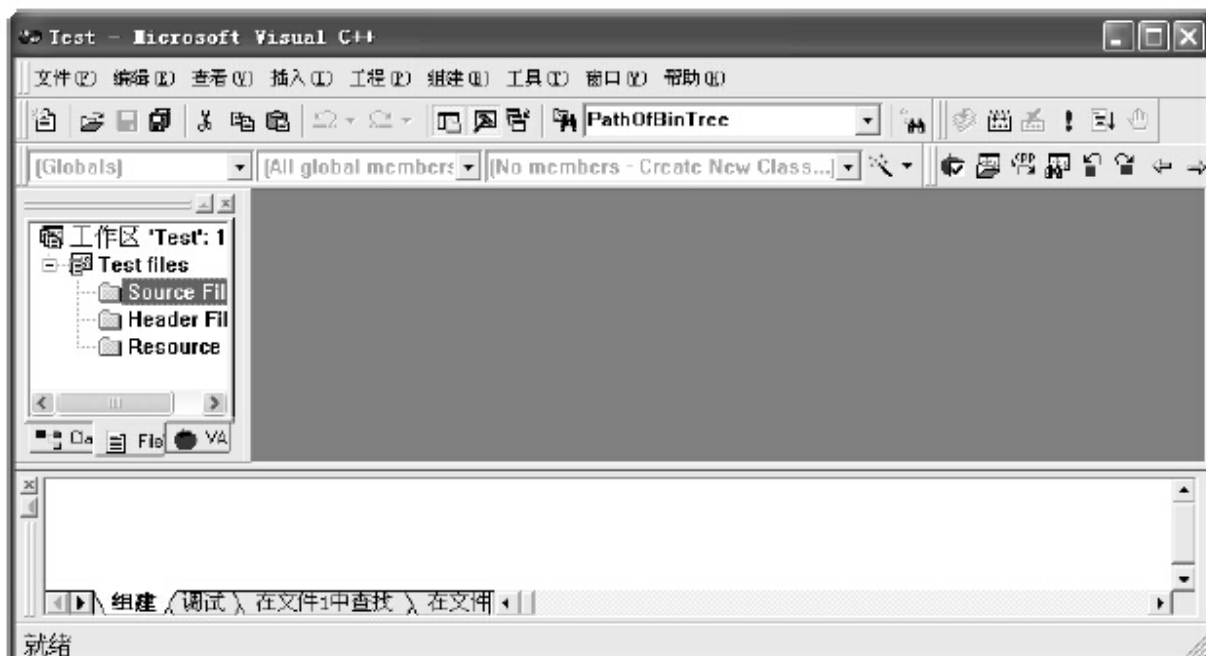


图2-12 控制台项目开发环境

这样，就建立了一个名字为Test的项目文件，下面就需要在该新建的控制台项目中建立一个C程序文件，即建立一个扩展名为“.c”的文件。

2. 在控制台项目中新建一个C程序文件

①单击【FileView】标签，再单击“Test”前的【+】按钮将其展开，然后右击“Source Files”选项，在弹出的快捷菜单中选择【添加文件到目录】命令，准备在项目中添加一个C程序文件，如图2-13所示。

②单击【添加文件到目录】命令，打开【插入文件到工程】对话框，并输入文件名“test.c”，如图2-14所示。单击【确定】按钮，弹出一个提示信息对话框，提示指定的test.c文件不存在，是否要建立一个文件，如图2-15所示。

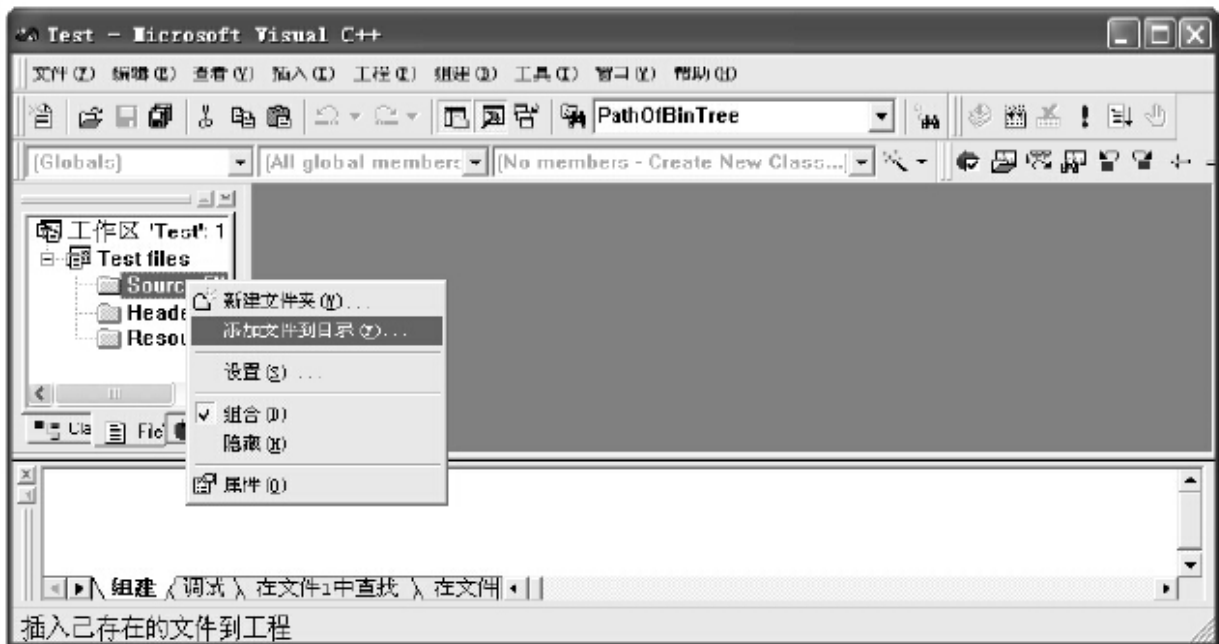


图2-13 在控制台项目中添加C程序文件








图2-14 在控制台项目中添加一个“test.c”文件对话框

③单击【是】按钮，就为项目建立了一个C程序文件，并命名为“test.c”。这时就可以编写C语言程序了。

④在编辑区编写一个简单的C程序，如图2-16所示。

其实，在Visual C++6.0环境中编辑C语言程序、编译、链接及运行程序都非常简单。编辑好程序后，只需要单击一个命令按钮就可以查看到运行结果了。

下面，单击工具栏    中的图标可编译程序，单击图标  可链接程序，单击图标  运行程序，其运行结果如图2-17所示。

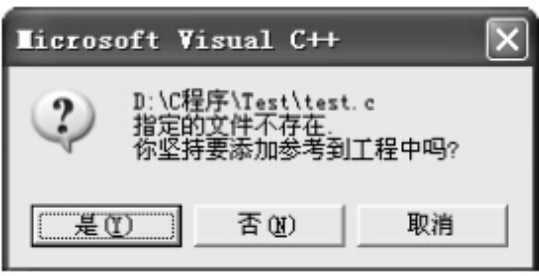


图2-15 添加一个“test.c”文件提示信息对话框

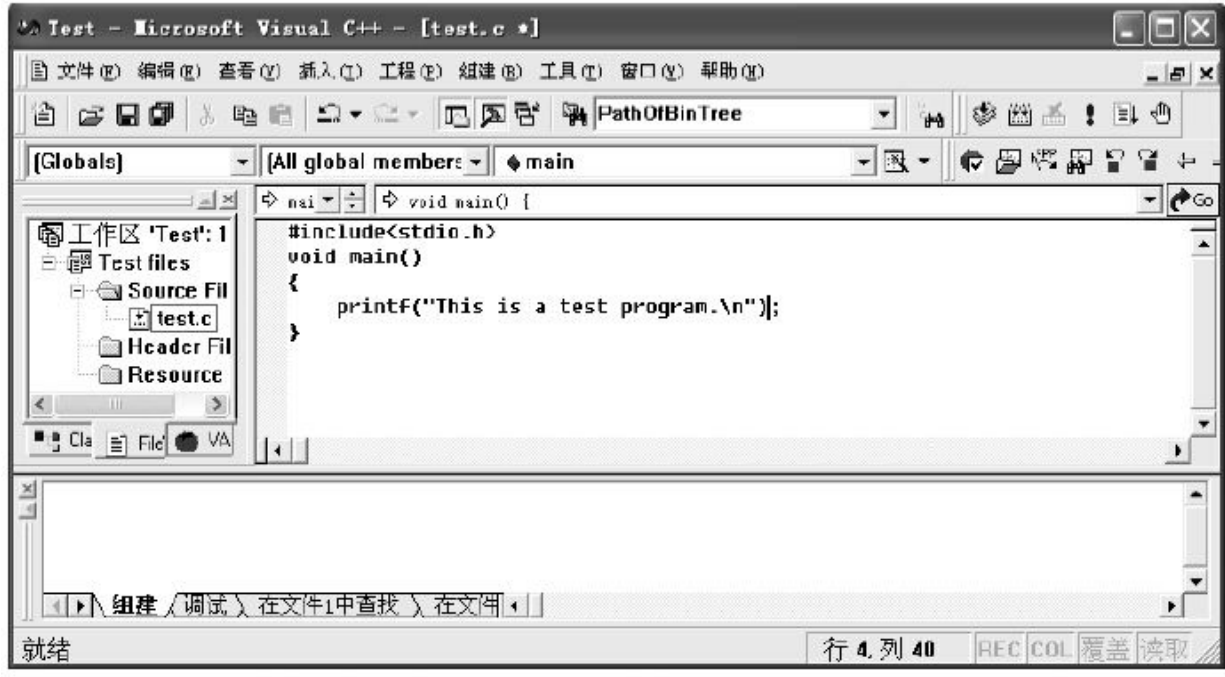


图2-16 “test.c”源程序

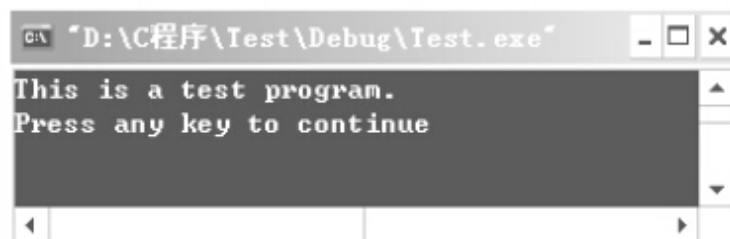


图2-17 “Test.c”程序运行结果

从图2-16中可以看到，Visual C++ 6.0开发环境中包括【文件】、【编辑】、【查看】、【插入】、【工程】、【组建】、【工具】、【窗口】和【帮助】9个主菜单。其中，【文件】菜单下面主要包括【新建】、【打开】、【保存】和【退出】等命令，通过【新建】命令，可以创建新的项目文件和C程序文件；【打开】命令可以打开一个已经存在的文件，【保存】命令可以将程序文件保存；【退出】命令可以用来退出Visual C++ 6.0程序窗口。

【组建】菜单下面主要包括【编译】、【组建】、【全部重建】、【开始调试】和【执行】等命令。其中，【编译】命令用来编译项目文件；【链接】命令用来将所有用到的程序文件链接到一起并生成扩展名为.exe的文件；【全部重建】命令可以将项目中的所有文件进行编译；【开始调试】命令包括断点调试和单步调试；【执行】命令用来运行可执行文件。

2.2 递归与非递归

在学习C语言的过程中递归是C语言的重点和难点。在数据结构与算法实践过程中，经常会遇到利用递归实现算法的情况。递归是一种分而治之、将复杂问题转换为简单问题的求解方法。使用递归可以使编写的程序简洁、结构清晰，程序的正确性很容易证明，不需要了解递归调用的具体细节。

本节主要介绍函数的递归调用、如何使用递归巧妙解决看上去比较复杂的问题。

2.2.1 函数的递归调用

简单来说，函数的递归调用就是自己调用自己，即一个函数在调用其他函数的过程中，又出现了对自身的调用，这种函数称为递归函数。函数的递归调用可分为直接递归调用和间接递归调用。其中，在函数中直接调用自己称为函数的直接递归调用，如图2-18所示；如果函数f1调用了函数f2，函数f2又调用了f1，这种调用方式称为间接递归调用，如图2-19所示。

函数的递归调用就是自己调用自己，可以直接调用自己也可以间接调用自己。

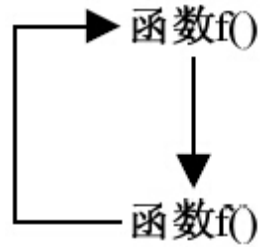


图2-18 直接递归调用过程

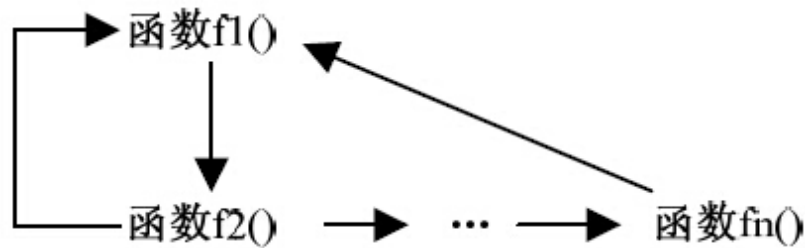


图2-19 间接递归调用过程

在用递归解决实际问题时，递归函数只知道最基本问题的解。在递归函数中，遇到基本问题时仅仅返回一个值，在解决较为复杂的问题时，通过将复杂的问题化解为比原有问题更简单、规模更小的问题，最后把复杂问题变成一个基本问题，而基本问题的答案是已知的，基本问题解决后，比基本问题大一点的问题也得到解决，直到原有问题得到解决。

例如，利用递归求 n 的阶乘 $n!$ 。 n 的阶乘递归定义为 $n! = n * (n-1)!$ ，当 $n=5$ 时，则有

```

5
! =5*4
!
4
! =4*3
!
3
! =3*2

```

```
!
2
! =2*1
!
1
! =1*0
!
0
! =1
```

递归计算 $5!$ 的过程如图2-20所示。因为 $5! = 5 * (5-1)!$ ，因此，如果能求出 $(5-1)!$ ，也就能求出 $5!$ ；又因为 $(5-1)! = (5-1) * (5-2)!$ ；因此，如果能求出 $(5-2)!$ ，则也就能求出 $(5-1)!$ ；……最后一直递归到 $1! = 1 * 0!$ ，因此，如果能求出 $0!$ ，也就能求出 $1!$ ； $0!$ 值是1，按上述分析过程逆向推回去，最后便可求得 $5!$ 。

这样就把求解问题 $5!$ 分解为求5这个基本问题与求 $4!$ 这个比较复杂的问题，接着继续把求解 $4!$ 分解为求4这个基本问题与求 $3!$ 比较复杂的问题，直到把原问题变成求解 $0! = 1$ 这个最基本的已知问题为止。

根据上述分析可知，求解可分成两个阶段，第一阶段是由未知逐步推得已知的过程，称为“回推”；第二阶段是与回推过程相反的过程，即由已知逐步推得最后结果的过程，称为“递推”。其中，左半部分是回推过程，回推过程在计算出 $0! = 1$ 时停止调用；右半部分是递推过程，直到计算出 $5! = 120$ 为止。

2.2.2 递归应用举例

下面以具体的例子讲解递归的调用。

【例2-1】 利用递归求 $n!$ 。

【分析】 前面已经分析了递归实现 $n!$ 的整个过程（见图2-20），只需要做到以下两点就可完成求 $n!$ ：

（1）当 $n=0$ （递归调用结束，即递归的出口）时，返回1。

（2）当 $n \neq 0$ 时，需要把复杂问题分解成较为简单的问题，直到分解成最简单的问题 $0! = 1$ 为止。

递归求 $n!$ 的算法实现如下。

```
#include<stdio.h>
long factorial(int n);
void main()
{
    int num;
    for(num=0;num<10;num++)
        printf("%d!=%ld\n",num,factorial(num));
}
long factorial(int n)
/*
递归求n
! 函数实现*/
{
    if(n==0)                                /*
当n=0
时，递归调用出口*/
        return 1;                          /*0
! =1
是最基本问题的解*/
    else                                     /*
否则 */
        return n*factorial(n-1);           /*
递归调用将问题分解成较为简单的问题*/
}
```

程序运行结果如图2-21所示。

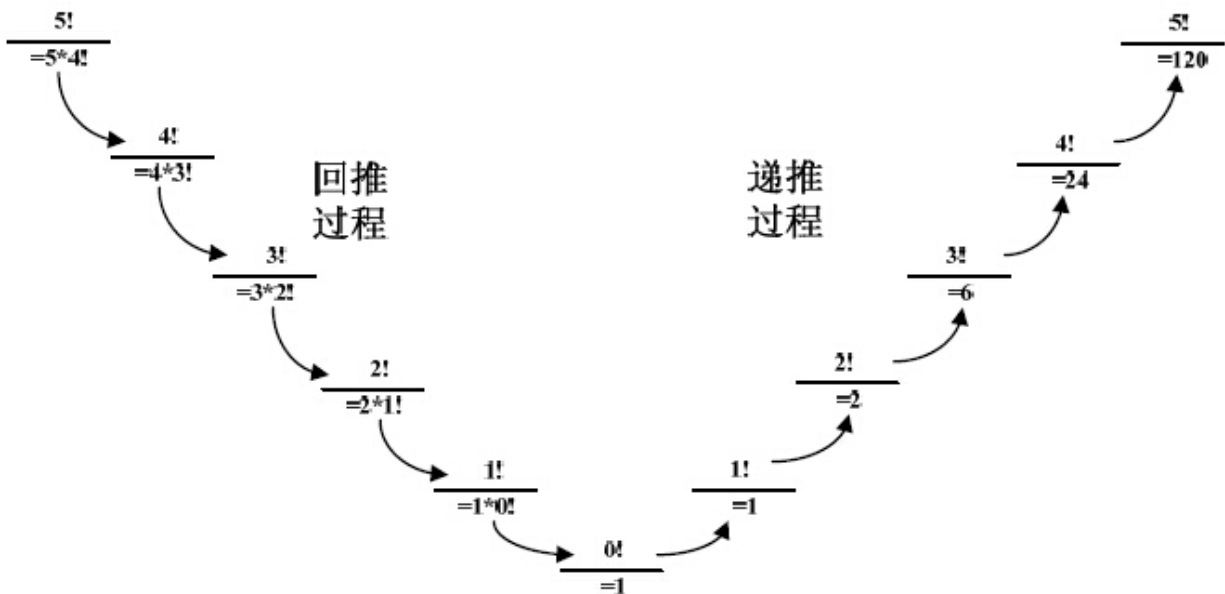


图2-20 求5! 的间接递归调用过程

```

C:\ "D:\数据结构\例2_1\Deb...
0!=1
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
8!=40320
9!=362880
Press any key to continue

```

图2-21 递归求n! 的运行结果

【例2-2】 要求利用递归实现求n个数中的最大者。

【分析】 假设元素序列存放在数组a中，数组a中n个元素的最大者可以通过将a[n-1]与前n-1个元素最大者比较之后得到。当n=1时，

有 $\text{findmax}(a, n) = a[0]$ ；当 $n > 1$ 时，有 $\text{findmax}(a, n) = (a[n-1] > \text{findmax}(a, n-1) ? a[n-1] : \text{findmax}(a, n-1))$ 。

也就是说，数组 a 中只有一个元素时，最大者是 $a[0]$ ；超过一个元素时，则要比较最后一个元素 $a[n-1]$ 和前 $n-1$ 个元素中的最大者，其中较大的一个即所求。而前 $n-1$ 个元素的最大者需要继续调用 findmax 函数。

求 n 个数中的最大者的递归算法实现如下。

```
#include<stdio.h>
#define N 200
int findmax(int a[],int n);
void main()
{
    int a[N],n,i;
    printf("
请输入n
的值:");

    scanf("%d",&n);
    printf("
请依次输入%d
个数: \n",n);

    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("
在这%d
个数中，最大的元素是:%d\n",n,findmax(a,n));
}
int findmax(int a[],int n)
{
    int m;
    if(n<=1)
        return a[0];
    else
    {
        m=findmax(a,n-1);
        return a[n-1]>=m?a[n-1]:m;
    }
}
```

程序的运行结果如图2-22所示。

【例2-3】 利用递归函数输出正整数和等于n的所有不增的正整数和式。例如，当n=5时，不增的和式如下。

```
5=5
5=4+1
5=3+2
5=3+1+1
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

【分析】 引入数组a，用来存放分解出来的和数，其中，a[k]存放第k步分解出来的和数。递归函数应设置3个参数，第1个参数是数组名a，用来将数组中的元素传递给被调用函数；第2个参数i表示本次递归调用要分解的数；第3个参数k是本次递归调用将要分解出的第k个和数。递归函数的原型如下。

```
void rd(int a[],int i,int k)
```

对将要分解的数i，可分解出来的数j共有i种可能选择，它们是i、i-1、…、2、1。但为了保证分解出来的和数依次构成不增的正整数数列，要求从i分解出来的和数j不能超过a[k-1]，即上次分解出来的和数。

特别地，为保证对第一步（k=1）分解也成立，程序可在a[0]预置n，即第一个和数最大为n。在分解过程中，当分解出来的数j==i时，说明已完成一个和式分解，应将和式输出；当分解出来的数j<i时，说明还有i-j需要进行第k+1次分解。

和为n的所有不增正整数和式分解算法实现如下。

```
#include<stdio.h>
#define N 100
void rd(int a[],int i,int k);
void main()
{
    int n,a[N];
    printf("
请输入一个正整数n(1
≤n
≤20):");
    scanf("%d",&n);
    a[0]=n;
    printf("
不增的和式分解结果:\n");
    rd(a,n,1);
}
void rd(int a[],int i,int k)
{
    int j,p;
    for(j=i;j>=1;j--)
    {
        if(j<=a[k-1])
        {
            a[k]=j;
            if(j==i)
            {
                printf("%d=%d",a[0],a[1]);
                for(p=2;p<=k;p++)
                    printf("+%d",a[p]);
                printf("\n");
            }
            else
                rd(a,i-j,k+1);
        }
    }
}
```

程序运行结果如图2-23所示。

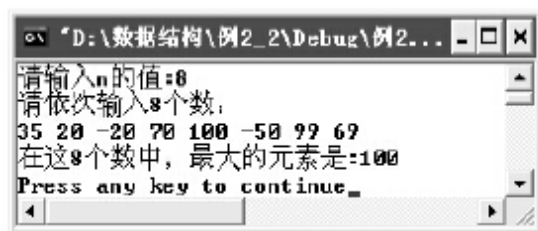


图2-22 递归实现求n个数的最大者程序运行结果

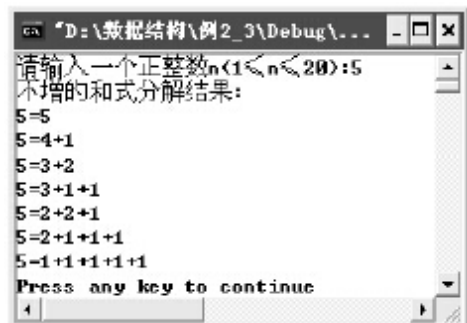


图2-23 和式分解

2.2.3 迭代与递归

迭代与递归是程序设计中最常用的两种结构。任何能使用递归解决的问题都能使用迭代的方法解决。迭代和递归的区别是，迭代使用的是循环结构，递归使用的是选择结构。使用递归能够使程序的结构更清晰，设计出的程序更简洁、程序更容易让人理解。

但是，递归也有许多不利之处，大量的递归调用会耗费大量的时间和内存。每次递归调用都会建立函数的一个备份，会占用大量的内存空间。迭代则不需要反复调用函数和占用额外的内存。通过分析递归求 n 的阶乘 $n!$ 的计算过程，我们可以把它转化为非递归实现，其非递归实现如下。

```
int NonRecFact(int n)
/*
非递归求前n
的阶乘*/
{
    int i,s=1;
    for(i=1;i<=n;i++) /*
利用迭代求n
的阶乘*/
        s*=i;
```

```
        return s;  
    }
```

对于大整数问题，考虑到 n 值非常大的情况，运算结果超出一般整数的位数，可以用一维数组存储长整数，数组中的每个元素只存储长整数的一位数字。如有 m 位长整数 N 用数组 $a[]$ 存储， $N=a[m]*10^{m-1}+a[m-1]*10^{m-2}+\dots+a[2]*10^1+a[1]*10^0$ ，并用 $a[0]$ 存储长整数 N 的位数 m ，即 $a[0]=m$ 。按上述约定，数组的每个元素存储 $k!$ 的每一位数字，并从低位到高位依次存储于数组的第2个元素、第3个元素……例如， $6!=720$ 在数组中的存储形式如图2-24所示。

其中，第一个元素3表示长整数是一个3位数，接着是低位到高位
的0、2、7，表示长整数720。

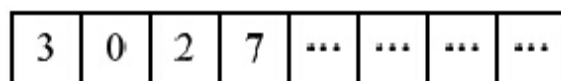


图2-24 $k!$ 在数组中的存储情况

在计算阶乘 $k!$ 时，可以采用对已求得的阶乘 $(k-1)!$ 连续累加 $k-1$ 次（即得到 $k*(k-1)!$ ）后得到。例如，已知 $5!=120$ ，计算 $6!$ ，可对原来的120再累加5次120（即得到 $6*5$ ）得到720。具体程序实现如下。

```
#include<stdio.h>  
#include<malloc.h>  
#define N 100  
void fact(int a[],int k)  
{  
    int *b,m,i,j,r,carry;  
    m=a[0];
```

```

        b=(int*)malloc(sizeof(int)*(m+1));
        for(i=1;i<=m;i++)
            b[i]=a[i];
        for(j=1;j<k;j++)
        {
            for(carry=0,i=1;i<=m;i++)
            {
                r=(i<=a[0]?a[i]+b[i]:a[i])+carry;
                a[i]=r%10;
                carry=r/10;
            }
            if(carry)
                a[++m]=carry;
        }
        free(b);
        a[0]=m;
    }
    void write(int *a,int k)
    {
        int i;
        printf("%4d!=",k);
        for(i=a[0];i>0;i--)
            printf("%d",a[i]);
        printf("\n");
    }
    void main()
    {
        int a[N],n,k;
        printf("
请输入正整数n
的值:");

        scanf("%d",&n);
        a[0]=1;a[1]=1;
        write(a,1);
        for(k=2;k<=n;k++)
        {
            fact(a,k);
            write(a,k);
        }
    }

```

对于较为简单的递归问题，可以利用简单的迭代将其转化为非递归。而对于较为复杂的递归问题，需要通过利用数据结构中的栈来消除递归。

2.3 指针

指针是C语言中的一个重要概念，也是最不容易掌握的内容。指针常常用在函数的参数传递和动态内存分配中。指针与数组相结合，使引用数组成分的形式更加多样化，访问数组元素的手段更加灵活；指针与结构体相结合，利用系统提供的动态存储手段，能构造出各种复杂的动态数据结构；利用指针形参，使函数能实现传递地址形参和函数形参的要求。在“数据结构”课程中，指针的使用非常频繁，因此，要想真正掌握数据结构，就需要灵活、正确地使用指针。本节主要介绍指针变量的概念、指针变量的引用、指针与数组、函数指针与指针函数。

2.3.1 什么是指针

指针是一种变量，也称指针变量，它的值不是整数、浮点数和字符，而是内存地址。指针的值就是变量的地址，而变量又拥有一个具体值。因此，可以理解为变量名直接引用了一个值，指针间接地引用了一个值。

在理解指针之前，先来了解下地址的概念。图2-25展示了变量在内存中的存储情况。假设a、b、c、d、bPtr分别是5个变量，其中，a、b、c、d是整型变量，bPtr是指针变量。整型变量在内存中占用4个字节，变量a的存放地址是2000、2001、2002和2003四个内存单元，变量b存放在2004~2007内存单元中，变量bPtr存放在4600~4603四个内存单元中。整型变量a、b、c、d的内容分别是25、12、78、5，而指针变量bPtr的内容是一个地址，为2004开始的内存地址，即bPtr存放的是变量b的地址，换句话说，就是bPtr指向变量b的存储位置，可以用一个箭头表示从地址是4600的位置指向变量地址为2004的位置。

一个存放变量地址的类型称为该变量的“指针”。如果有一个变量用来存放另一个变量的地址，则称这个变量为指针变量。在图2-26中，qPtr用来存放变量q的地址，qPtr就是一个指针变量。

在C语言中，所有变量在使用前都需要声明。例如，声明一个指针变量的语句如下。

```
int *qPtr,q;
```

q是整型变量，表示要存放一个整数类型的值；qPtr是一个整型指针变量，表示要存放一个变量的地址，而这个变量是整数类型。qPtr叫做一个指向整型的指针。

说明 在声明指针变量时，“*”只是一个指针类型标识符，指针变量的声明也可以写成int*qPtr。

指针变量可以在声明时赋值，也可以在声明后赋值。例如，在声明时为指针变量赋值的语句如下。

```
int q=12;
:
int *qPtr=&q;
```

或在声明后为指针变量赋值，语句如下。

```
int q=12,*qPtr;
qPtr=&q;
```

这两种赋值方法都是把变量q的地址赋值给指针变量qPtr。qPtr=&q叫做指向变量q，其中，&是取地址运算符，表示返回变量q的地址。指针变量qPtr与变量q的关系如图2-26所示。

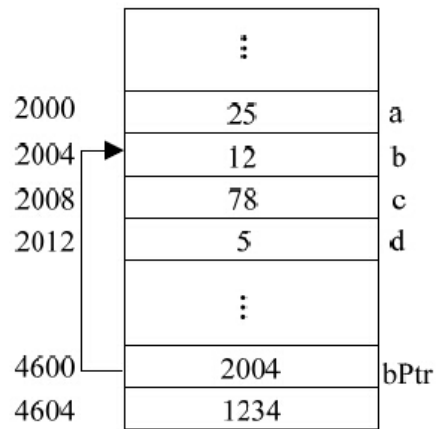


图2-25 指针变量在内存中的表示

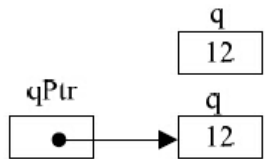


图2-26 q直接引用一个值和qPtr间接引用一个变量q

直接引用和间接引用可以用日常生活中的两个抽屉来形象说明。有两个抽屉A和B，抽屉A有一把钥匙，抽屉B也有一把钥匙。为了方便，可以把两把钥匙都带在身上，需要取抽屉A中的东西时直接用钥匙A打开抽屉；也可以为了安全，把钥匙A放到抽屉B中，把抽屉B的钥匙带在身上，需要取抽屉A中的东西时，先打开抽屉B，再取出抽屉A的钥匙，然后打开抽屉A，取出需要的东西。前一种方法就相当于通过变量直接引用，后一种方法相当于通过指针间接引用。其中，抽屉B的钥匙相当于指针变量，抽屉A的钥匙相当于一般的变量。

2.3.2 指针变量的间接引用

与普通变量一样，指针变量也可以对数据进行操作。指针变量主要通过取地址运算符&和指针运算符*来存取数据。例如，&a指的是变量a的地址，*ptr表示变量ptr所指向的内存单元存放的内容。下面通过具体例子说明&和*运算符及指针变量的使用。

【例2-4】 利用变量和指针变量存取数据。

【分析】 主要考查如何利用&和*运算符来存取变量中的数据，取地址运算符&和指针运算符*是互逆的操作，应灵活掌握两个运算符的使用技巧。程序代码如下。

```
#include<stdio.h>
void main()
{
    int q=12;
    int *qPtr;
    声明指针变量qPtr/*
    qPtr=&q;
    指针变量qPtr指向变量q*/
    打印变量q的地址和qPtr的内容*/
    printf("q的地址是: %p\nqPtr中的内容是: %p\n", &q, qPtr);
    打印q的值和qPtr指向变量的内容*/
    printf("q的值是: %d\n*qPtr的值是: %d\n", q, *qPtr);
    运算符'&'和'*'是互逆的*/
    printf("&*qPtr=%p, *qPtr=%p\n", &*qPtr, *qPtr);
    因此有&*qPtr=*qPtr\n", &*qPtr, *qPtr);
}
```

程序运行结果如图2-27所示。



图2-27 利用变量与指针变量进行存取操作的运行结果

&和*作为单目运算符，结合性是从右到左，优先级别相同，因此对于表达式*&qPtr来说，先进行*运算，后进行&运算。因为qPtr是指向变量q的，所以*qPtr的值为q，*&qPtr就是对q取地址，即&q，q的地址。*&qPtr是先进行取地址运算即&qPtr，即qPtr的地址，然后进行*运算，那么*&qPtr就是qPtr本身，即q的地址。因此，*&qPtr和*&qPtr是等价的。

注意 指针变量只能用来存放地址，不能将一个整型值赋给一个指针变量。而且指针变量的类型应和所指向的变量的类型一致，例如整型指针只能指向整型变量，不能指向浮点型变量。指针变量是一种数据类型，表明该变量用来存放变量的地址；变量指针是变量的地址。

2.3.3 指针与数组

指针可以与变量结合，也可以与数组结合使用。指针数组和数组指针是两个截然不同的概念，指针数组是一种数组，该数组存放的是一组变量的地址。数组指针是一个指针，表示该指针是指向数组的指针。

1. 指向数组元素的指针

指针可以指向变量，也可以指向数组及数组中的元素。

例如定义一个整型数组和一个指针变量，语句如下。

```
int a[5]={10,20,30,40,50};
int *aPtr;
```

这里的a是一个数组，它包含了5个整型数据。变量名a就是数组a的首地址，它与&a[0]等价。如果令aPtr=&a[0]或者aPtr=a，则aPtr也指向了数组a的首地址。如图2-28所示。

也可以在定义指针变量时直接赋值，如以下语句是等价的。

```
int *aPtr=&a[0];
int *aPtr;
aPtr =&a[0];
```

与整型、浮点型数据一样，指针也可以进行算术运算，但含义却不同。当一个指针加1（或减）1并不是指针值增加（或减少）1，而是使指针指向的位置向后（或向前）移动了一个位置，即加上（或减去）该整数与指针指向对象的大小的乘积。例如对于`aPtr+=3`，如果一个整数占用4个字节，则相加后`aPtr=2000+4*3=2012`（这里假设指针的初值是2000）。同样指针也可以进行自增（++）运算和自减（--）运算。

也可以用一个指针变量减去另一个指针变量。例如，指向数组元素的指针`aPtr`的地址是2008，另一个指向数组元素的指针`bPtr`的地址是2000，则`a=aPtr-bPtr`的运算结果就是把从`aPtr`到`bPtr`之间的元素个数赋给`a`，元素个数为 $(2008-2000)/4=2$ （假设整数占用4个字节）。

我们也可以通过指针来引用数组元素。例如以下语句。

```
*(aPtr+2);
```

如果`aPtr`是指向`a[0]`，即数组`a`的首地址，则`aPtr+2`就是数组`a[2]`的地址，`*(aPtr+2)`就是30。

注意 指向数组的指针可以进行自增或自减运算，但是数组名则不能进行自增或自减运算，这是因为数组名是一个常量指针，它是一个常量，常量值是不能改变的。

【例2-5】 用指针引用数组元素并打印输出。

【分析】 主要考查指针与数组结合进行的运算，有指针对数组的引用及指针的加、减运算。指针及数组对元素操作的实现如下。

```
#include<stdio.h>
void main()
{
    int a[5]={10,20,30,40,50};
    int *aPtr,i;
    指针变量声明*/
    aPtr=&a[0];
    指针变量指向变量q*/
```

```

        for(i=0;i<5;i++)                                /*
通过数组下标引用元素的方式输出数组元素*/
        printf("a[%d]=%d\n",i,a[i]);
        for(i=0;i<5;i++)                                /*
通过数组名引用元素的方式输出数组元素*/
        printf("*(a+%d)=%d\n",i,*(a+i));
        for(i=0;i<5;i++)                                /*
通过指针变量下标引用元素的方式输出数组元素*/
        printf("aPtr[%d]=%d\n",i,aPtr[i]);
        for(aPtr=a,i=0;aPtr<a+5;aPtr++,i++)              /*
通过指针变量偏移引用元素的方式输出数组元素*/
        printf("*(aPtr+%d)=%d\n",i,*aPtr);
}

```

程序中共有4个for循环，其中第一个for循环是利用数组的下标访问数组的元素，第二个for循环是利用使用数组名访问数组的元素，在C语言中，地址也可以像一般的变量一样进行加、减运算，但是指针的加1和减1表示的是一个元素单元。第三个for循环是利用指针访问数组中的元素，第四个for循环则是先将指针偏移，然后访问该指针所指向的内容。

上述4种访问数组元素的方法表明，在C语言中指针的运用非常灵活。

程序运行结果如图2-29所示。

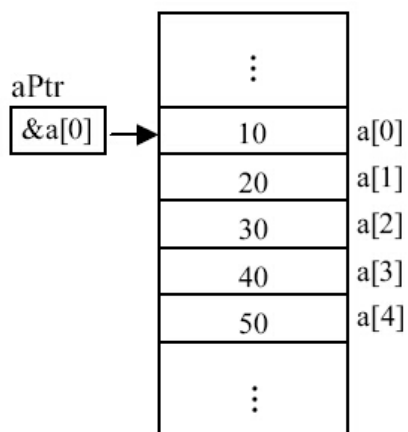


图2-28 数组的指针与数组在内存中的关系

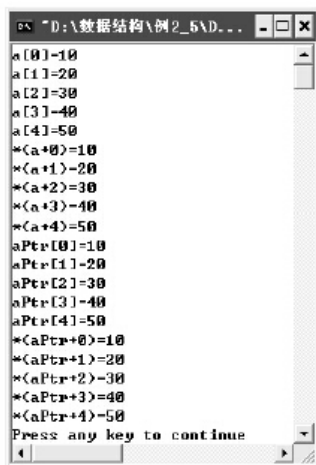


图2-29 指针引用数组元素的运行结果

2. 指针数组

指针数组 其实也是一个数组，只是数组中的元素是指针类型的数据。换句话说，指针数组中的每一个元素都是一个指针变量。

定义指针数组的方式如下。

```
int *p[4];
```

由于[]运算符优先级比*高，p优先与[]结合，形成p[]数组形式，然后与*结合，表示该数组是指针类型的，每个数组元素是一个指向整型的变量。从字面上理解，指针数组首先是一个数组，这个数组存放的是指针类型的变量。

指针数组常常用于存储一些长度不等的字符串数据，有的读者可能会问，为什么不存放在二维数组中？这是因为字符串长度不等，若将这些字符串存放在二维数组中，就需要定义一个能容纳最长字符串的二维数组，这样就会出现一部分存储空间不能得到有效利用。

例如字符串“C Programming Language”、“Assembly Language”、“Data Structure”和“Natural Language”在二维数组中的存储情况，如图2-30所示。

C		P	r	o	g	r	a	m	m	i	n	g		L	a	n	g	u	a	g	e	\0
A	s	s	e	m	b	l	y		L	a	n	g	u	a	g	e	\0					
D	a	t	a		S	t	r	u	c	t	u	r	e	\0								
N	a	t	u	r	a	l		L	a	n	g	u	a	g	e	\0						

图2-30 字符串在二维数组中的存储情况

不难看出，利用二维数组保存多个字符串时，为了保证能存储所有的字符串，必须按最长的字符串长度来定义二维数组的列数。为了节省存储单元，可以采用指针数组保存字符串，定义如下。

```
char *s[4]={"C Programming Language","Assembly Language","Java Language","Natural Language"};
```

在指针数组中存储字符串的情况如图2-31所示。

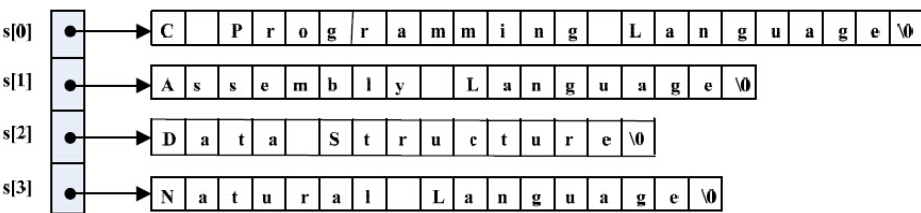


图2-31 字符串在指针数组中的存储情况

这样在字符串比较多且长度不一时，就可以大大地节省内存空间。

【例2-6】 用指针数组保存字符串并将字符串打印输出。

【分析】 主要考查指针的应用及对指针数组概念的理解，其实`s[4]`就是一个特殊的数组，`s[0]`、`s[1]`、`s[2]`、`s[3]`分别存放指向4个字符串指针，即数组保存的是各个字符串的首地址。代码如下。

```
#include<stdio.h>
void main()
{
    /*
    定义指针数组*/
    char *s[4]={"C Programming Language","Assembly Language"," Data Structure ","Natural Language"};
    int n=4;
    /*
    指针数组元素的个数*/
    int i;
    char *aPtr;
    /*
    第1
    种方法输出：通过数组名输出字符串*/
    printf("
    第1
    种方法输出：通过指针数组的数组名输出字符串:\n");
```

```

        for(i=0;i<n;i++)
            printf("
第%d
个字符串: %s\n",i+1,s[i]);
/*
第2
种方法输出: 通过指向数组的指针输出字符串*/
printf("
第2
种方法输出: 通过指向数组的指针输出字符串:\n");
for(aPtr=s[0],i=0;i<n;aPtr=s[i])
{
    printf("
第%d
个字符串: %s\n",i+1,aPtr);
    i++;
}
}

```

程序运行结果如图2-32所示。

【例2-7】 利用指针数组实现对一组变量的值按照从小到大排序，排序时交换指向变量的指针值。

【分析】 主要考查指针数组的概念及用法。让指针数组的各成分分别指向各变量，通过指针数组引用变量的值并交换之，使存放在指针数组中各分量指向的变量值从小到大顺序排列。实现代码如下。

```

#include<stdio.h>
void bubble(int *a[],int n);
void main()
{
    int a,b,c,d,e,f,i;
    int *vp[]={&a,&b,&c,&d,&e,&f};
    printf("
请输入a,b,c,d,e,f
的值: ");
    scanf("%d%d%d%d%d%d",&a,&b,&c,&d,&e,&f);
    bubble(vp,sizeof(vp)/sizeof(vp[0]));
    for(i=0;i<sizeof(vp)/sizeof(vp[0]);i++)
        printf("%4d",*vp[i]);
    printf("\n");
}
void bubble(int *a[],int n)
{
    int i,j,flag,t;
    for(i=0;i<n-1;i++)
    {
        flag=0;
        for(j=0;j<n-1-i;j++)
            if(*a[j]>*a[j+1])
            {
                t=*a[j];
                *a[j]=*a[j+1];
                *a[j+1]=t;
                flag=1;
            }
        if(flag==0)
            return;
    }
}

```

程序运行结果如图2-33所示。

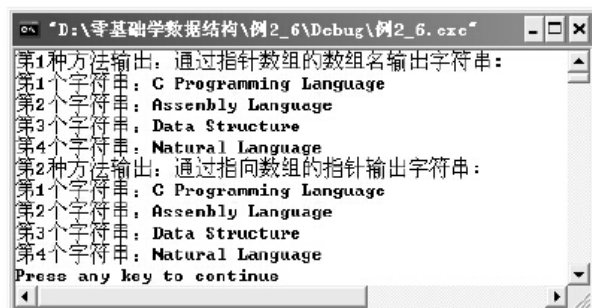


图2-32 字符数组输出结果

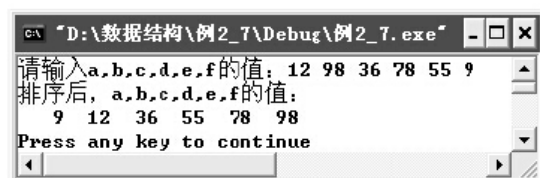


图2-33 程序运行结果

3. 数组指针

数组指针 是指向数组的一个指针。如下定义。

```
int (*p)[4];
```

其中，p是指向一个拥有4个元素的数组指针，数组中每个元素都为整型。与前面刚刚介绍过的指针数组做比较，这里定义的数组指针多了一对括号，*p两边的括号不可以省略。这里定义的p仅仅是一个指针，不过这个指针有点特殊，这个p指向的是包含4个元素的一维数组。数组指针p与它指向的数组之间的关系可以用图2-34来表示。

如果有如下语句。

```
int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};  
p=a;
```

数组指针p与数组a中元素之间的关系如图2-35所示。其中，(*p)[0]、(*p)[1]、(*p)[2]、(*p)[3]分别保存的是元素值为1、2、3、4的地址。p、p+1和p+2分别指向二维数组的第一行、第二行和第三行，p+1表示将指针p移动到下一行。

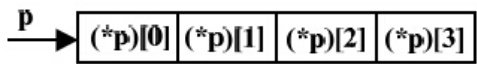


图2-34数组指针p的表示

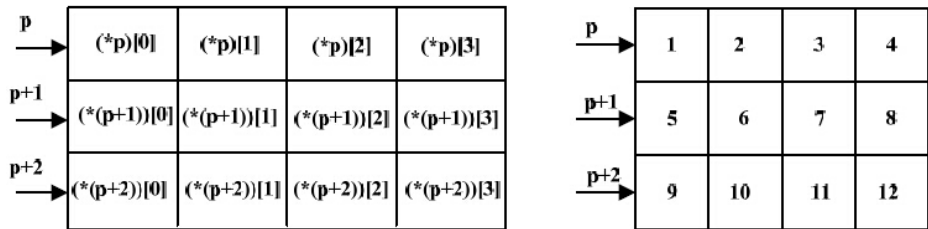


图2-35 数组指针p与二维数组的对应关系

$* (p+1) + 2$ 表示数组a第1行第2列的元素的地址，即 $\&a[1][2]$ ， $* (* (p+1) + 2)$ 表示 $a[1][2]$ 的值即7，其中1表示行，2表示列。下面编程输出以上数组指针的值和数组的内容。

【例2-8】 在屏幕上打印图2-35中数组指针p及数组a中的元素。

【分析】 主要考查利用数组指针引用数组中的元素的方法。数组指针p与数组a中元素的对应关系如图2-35所示。通过利用数组指针p引用数组a中的元素并输出p的值，以验证对指针引用的正确性，加深对数组指针的理解。实现代码如下。

```
#include<stdio.h>
void main()
{
    int a[3][4]={1,2,3,4},{5,6,7,8},{9,10,11,12}};
    int (*p)[4];
    声明数组指针变量p*/
    int row,col;
    p=a;
    指针p
    指向数组元素为4
    的数组*/
    打印输出数组指针p
    指向的数组的值*/
    for(row=0;row<3;row++)
    {
        for(col=0;col<4;col++)
            printf("a[%d,%d]=%-4d",row,col,*(* (p+row)+col));
        printf("\n");
    }
    通过改变指针p
    修改数组a
    的行地址，改变col
    的值修改数组a
    的列地址*/
    for(p=a,row=0;p<a+3;p++,row++)
    {
        for(col=0;col<4;col++)
            printf("( *p[%d] ) [%d]=%p",row,col, (*(p)+col));
        printf("\n");
    }
}
```

程序运行结果如图2-36所示。

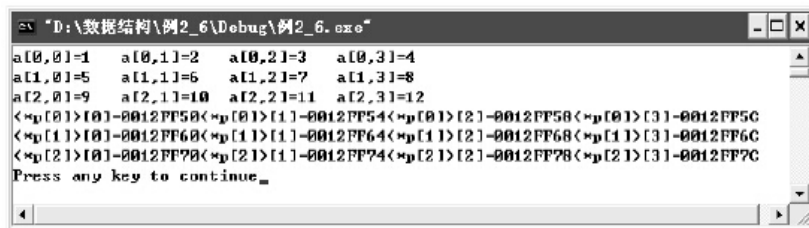


图2-36 打印输出数组指针和数组元素

注意 区别数组指针和指针数组。数组指针首先是一个指针，并且它是一个指向数组的指针。指针数组首先是一个数组，并且它是保存指针变量的数组。

2.3.4 指针函数与函数指针

与指针数组、数组指针一样，指针函数与函数指针也是一对孪生兄弟，也是常常容易混淆的概念。顾名思义，指针函数是一种函数，它表示函数的返回值是指针类型；函数指针是指针的一种，它表示该指针指向一个函数。

1. 指针函数

指针函数 是指函数的返回值是指针类型的函数。在C语言中，一个函数的返回值可以是整型、实型和字符型，也可以是指针类型。例如，以下是一个指针函数的定义。

```
float *func(int a,int b);
```

其中，func是函数名，前面的“*”表明返回值的类型是指针类型，因为前面的类型标识符是float，所以返回的指针是指向浮点型的。该函数有两个参数，参数类型都是整型。下面我们还是通过一个具体实例来介绍指针函数的用法。

【例2-9】 假设若干个学生的成绩在二维数组中存放，要求输入学生编号，利用指针函数实现其成绩的输出。

【分析】 主要考查指针函数的使用。学生成绩存放在二维数组中，每一行存放一个学生的成绩，通过输入学生编号，返回该学生存放成绩的地址，然后利用指针输出每一门的学生成绩。

绩。程序实现如下。

```
#include<stdio.h>
int *FindAddress(int (*ptr)[4],int n);          /*
声明查找成绩地址函数*/
void Display(int a[][4],int n,int *p);          /*
声明输出成绩函数*/
void main()
{
    int row,n=4;
    int *p;
    int score[3][4]={{83,78,79,88},{71,88,92,63},{99,92,87,80}};
    printf("

请输入学生编号(1
或2
或3).
输入0
退出程序.\n");
    scanf("%d",&row);                                /*
输入要输出学生成绩的编号*/
    while(row)
    {
        if(row==1||row==2||row==3)
        {
            printf("

第%d
个学生的成绩4
门课的成绩是: \n",row);

            p=FindAddress(score,row-1);                /*
调用指针函数*/
            Display(score,n,p);                        /*
调用输出成绩函数*/
            printf("

请输入学生编号(1
或2
或3).
输入0
退出程序.\n");

            scanf("%d",&row);
        }
        else
        {
            printf("

输入不合法,请重新输入(1
或2
或3)
, 输入0
退出程序.\n");

            scanf("%d",&row);
        }
    }
}

int *FindAddress(int (*ptrScore)[4],int n)
/*
查找某条学生成绩记录地址函数。通过传递的行地址找到要查找学生成绩的地址,并返回行地址*/
{
    int *ptr;
    ptr=(ptrScore+n);                                /*
修改行地址,即找到学生的第一门课成绩的地址*/
    return ptr;
}

void Display(int a[][4],int n,int *p)
/*
输出学生成绩的实现函数。利用传递过来的指针输出每门课的成绩*/
{
    int col;
    for(col=0;col<n;col++)
        printf("%4d",*(p+col));                    /*
输出查找学生的每门课成绩*/
    printf("\n");
}
```

程序运行结果如图2-37所示。

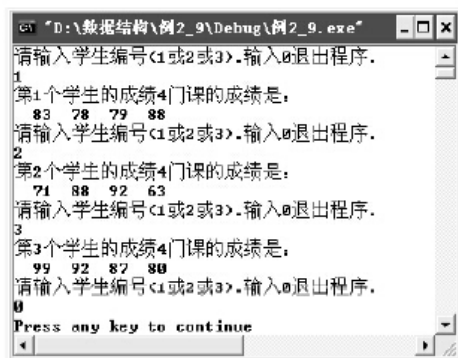


图2-37 通过指针函数返回指针并输出成绩的运行结果

主函数通过语句 `p=FindAddress (score, row-1)`；调用指针函数 `*FindAddress (int (*ptrScore) [4], int n)`，并把二维数组的行地址传递给形式参数 `ptrScore`，在 `*FindAddress (int (*ptrScore) [4], int n)` 中，执行语句 `ptr=*(ptrScore+n)`，返回行指针 `ptr`，然后调用 `Display (score, n, p)` 输出成绩。在 `Display (int a[][4], int n, int*p)` 中，通过 `p+col` 改变列地址，即找到该学生成绩的每门课的位置，依次输出每门课的成绩。

2. 函数指针

指针可以指向变量、数组，也可以指向函数，指向函数的指针就是函数指针。与数组名类似，函数名就是程序在内存中的起始地址。指向函数的指针可以把地址传递给函数，也可以从函数返回给指向函数的指针。

【例2-10】 通过一个函数求两个数的乘积，并通过函数指针调用该函数。

【分析】 主要考查函数指针的调用方法。程序实现代码如下。

```
#include<stdio.h>                                     /*
包含输入输出*/
int Mult(int a,int b);                                /*
声明两个数乘积的函数*/
void main()
{
    int a,b;
    int (*func)(int,int);                             /*
声明一个函数指针*/
    printf("
请输入两个数:");
    scanf("%d,%d",&a,&b);
    /*
第1
种调用函数的方法: 函数名调用求两个数乘积函数*/
    printf("
...

```

```

第1
种调用函数的方法：函数名调用求两个数乘积的函数:\n");
printf("%d*d=%d\n",a,b,Mult(a,b));          /*
通过函数名调用*/
/*
第2
种调用函数的方法：函数指针调用求两个数乘积的函数*/
func=Mult;                                  /*
函数指针指向求乘积函数*/
printf("
第2
种调用函数的方法：函数指针调用求两个数乘积函数:\n");
printf("%d*d=%d\n",a,b,(*func)(a,b));        /*
通过函数指针调用函数*/
}
int Mult(int x,int y)                        /*
实现求两个数乘积函数*/
{
    return x*y;
}

```

程序运行结果如图2-38所示。

语句 `int (*func) (int, int);` 声明一个指向函数的指针变量，并且所指向的函数返回值为整型，它有两个整型参数。

在声明函数中，由于 `*` 运算符的优先级比 `()` 运算符高，所以 `()` 不可以省略。

语句 `func=Mult;` 表示函数指针 `func` 指向函数 `Mult`，`func` 和 `Mult` 均指向函数 `Mult` 的起始地址，程序在编译阶段会被翻译成一行行指令并被装入到内存区域，如图2-39所示。

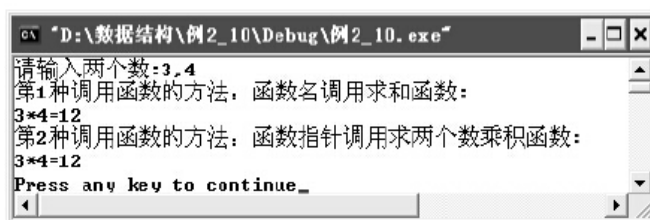


图2-38 函数指针调用求两个数乘积函数的运行结果

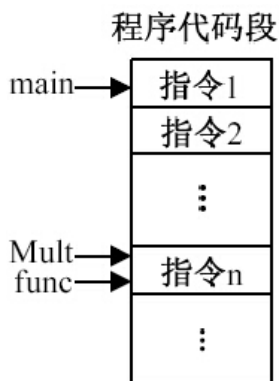


图2-39 函数指针在内存中的表示

其中，主函数中的语句（*func）（a，b）；是执行调用求两个数乘积函数的，因为函数本身就是一个地址，也可以写成func（a，b）的形式。

函数指针还可以作为参数传递给其他函数。

【例2-11】 利用函数指针作为函数参数，实现选择排序算法的升序排列和降序排列。

【分析】 主要考查函数指针作为函数参数的使用。程序实现代码如下。

```
#include<stdio.h> /*
包含输入输出函数*/
#define N 10 /*
数组元素个数*/
int Ascending(int a,int b); /*
是否进行升序排列*/
int Descending(int a,int b); /*
是否进行降序排列*/
void swap(int *,int *); /*
交换数据*/
void SelectSort(int a[],int n,int (*compare)(int,int));/*
选择排序，函数指针作为参数调用*/
void Display(int a[],int n); /*
输出数组元素*/
void main()
{
    int a[N]={22,55,12,7,19,65,81,3,30,52};
    int flag;
    while(1)
    {
        printf("1:
从小到大排序.\n2:
从大到小排序.\n0:
结束!\n");
        printf("
请输入: ");
        scanf("%d",&flag);
        switch(flag)
        {
            case 1:
                printf("
排序前的数据为:");
                Display(a,N);
                SelectSort(a,N,Ascending); /*
从小到大排序，将函数作为参数传递*/
                printf("
从小到大排列后的元素序列为:");
                Display(a,N);
                break;
            case 2:
                printf("
排序前的数据为:");
                Display(a,N);
                SelectSort(a,N,Descending); /*
从大到小排序，将函数作为参数传递*/
                printf("
从大到小排列后的元素序列为:");
                Display(a,N);
                break;
            case 0:
                printf("
程序结束!\n");
                return;
                break;
            default:
                printf("
输入数据不合法，请重新输入.\n");
                break;
        }
    }
}
/*
选择排序，将函数作为参数传递，判断是从从小到大还是从大到小排序*/
void SelectSort(int a[],int n,int (*compare)(int,int))
{
    int i,j,k;
```

```

        for(i=0;i<n;i++)
        {
            j=i;
            for(k=i+1;k<n;k++)
                if ((*compare) (a[k],a[j]))
                    j=k;
            swap(&a[i],&a[j]);
        }
    }
    /*
    交换数组的元素*/
    void swap(int *a,int *b)
    {
        int t;
        t=*a;
        *a=*b;
        *b=t;
    }
    /*
    判断相邻数据大小，如果前者大，升序排列需要交换*/
    int Ascending(int a,int b)
    {
        if(a<b)
            return 1;
        else
            return 0;
    }
    /*
    判断相邻数据大小，如果前者大，降序排列需要交换*/
    int Descending(int a,int b)
    {
        if(a>b)
            return 1;
        else
            return 0;
    }
    /*
    输出数组元素*/
    void Display(int a[],int n)
    {
        int i;
        for(i=0;i<n;i++)
            printf("%4d",a[i]);
        printf("\n");
    }

```

程序运行结果如图2-40所示。

```

D:\数据结构\例2_11\Debug\例2_11.exe
1:从小到大排序。
2:从大到小排序。
0:结束!
请输入: 1
非序前的数据为: 22 55 12 7 19 65 81 3 38 52
从小到大排列后的元素序列为: 3 7 12 19 22 38 52 55 65 81
1:从小到大排序。
2:从大到小排序。
0:结束!
请输入: 2
非序前的数据为: 3 7 12 19 22 38 52 55 65 81
从大到小排列后的元素序列为: 81 65 55 52 38 22 19 12 7 3
1:从小到大排序。
2:从大到小排序。
0:结束!
请输入: 0
程序结束!
Press any key to continue

```

图2-40 函数指针作为函数参数传递的排序运行结果

其中，函数SelectSort (a, N, Ascending) 中的参数Ascending是一个函数名，传递给函数定义void SelectSort (int a[], int n, int (*compare) (int, int)) 中的函数指针compare，这样指针就指向了Ascending。从而可以在执行语句 (*compare) (a[j],

a[j+1]) 时调用函数Ascending (int a, int b) 判断是否需要交换数组中两个相邻的元素，然后调用swap (&a[j], &a[j+1]) 进行交换。

3. 函数指针数组

假设有3个函数f1、f2和f3，可以把这3个函数作为数组元素存放在一个数组中，需要定义一个指向函数的指针数组指向这三个函数，代码如下。

```
void (*f[3])(int)={f1,f2,f3};
```

f是包含3个指向函数指针元素的数组，f[0]、f[1]和f[2]分别指向函数f1、f2和f3。通过函数指针f调用函数的形式如下。

```
f[n](m); /*
和m
都是整数*/
```

【例2-12】 声明一个指向函数的指针数组，并通过指针调用函数。

【分析】 主要考查指向函数的指针数组的使用。程序实现如下。

```
#include<stdio.h>
void f1(int n); /*
函数f1
声明*/
void f2(int n); /*
函数f2
声明*/
void f3(int n); /*
函数f3
声明*/
void main()
{
    void (*f[3])()={f1,f2,f3}; /*
    声明指向函数的指针数组*/
    int flag;
    printf("
调用函数请输入1
、2
或者3
，结束程序请输入0
。 \n");
    scanf("%d",&flag);
    while(flag)
    {
        if(flag==1||flag==2||flag==3)
        {
            f[flag-1](flag); /*
            通过函数指针调用数组中的函数*/
            printf("
请输入1
、2
或者3
，输入0
结束程序.\n");
            scanf("%d",&flag);
        }
        else
```



```

        {
            printf("
请输入一个合法的数 (1~3),
输入0
结束程序.\n");
            scanf("%d",&flag);
        }
    }
    printf("
程序结束.\n");
}
void f1(int n) /*
函数f1
的定义*/
{
    printf("
函数f%d:
调用第%d
个函数! \n",n,n);
}
void f2(int n) /*
函数f2
的定义*/
{
    printf("
函数f%d:
调用第%d
个函数! \n",n,n);
}
void f3(int n) /*
函数f3
的定义*/
{
    printf("
函数f%d:
调用第%d
个函数! \n",n,n);
}

```

程序的运行结果如图2-41所示。

```

D:\零基础学数据结构\例2_12\Debug...
调用函数请输入1、2或者3，结束程序请输入0。
1
函数f1:调用第1个函数!
请输入1、2或者3，输入0结束程序。
2
函数f2:调用第2个函数!
请输入1、2或者3，输入0结束程序。
3
函数f3:调用第3个函数!
请输入1、2或者3，输入0结束程序。
0
程序结束。
Press any key to continue

```

图2-41 指针调用保存在数组中的函数的运行结果

注意 函数指针不能执行像f+1、f++、f--等运算。

2.4 参数传递

在程序设计过程中，参数传递是经常会遇到的情况。在C语言中，函数的参数传递的方式通常有两种，一种是传值的方式；另一种是传地址的方式。本节主要介绍传值调用和传地址调用。

2.4.1 传值调用

在函数调用时，一般情况下，调用函数和被调用函数之间会有参数传递。调用函数后面括号里面的参数是**实际参数**，被调用函数中的参数是**形式参数**。传值调用是建立参数的一个副本并把值传递给形式参数，在被调用函数中修改形式参数的值，并不会影响到调用函数实际参数的值。

【例2-13】 编写一个函数，求两个整数的最大公约数。

【分析】 通过传值调用的方式，把实际参数的值传递给形式参数，其实形式参数是实际参数的一个副本（拷贝）。其程序实现如下。

```
#include <stdio.h>                                /*
包含输入输出函数*/
int GCD(int m,int n);                               /*
求两个整数的最大公约数的函数声明*/
void main()
{
    int a,b,v;
    printf("
    .... . . . . .
```

```

请输入两个整数:");
scanf("%d,%d",&a,&b);
v=GCD(a,b);
调用求两个数中的较大者的函数*/
printf("%d
和%d
的最大公约数为:%d\n",a,b,v);
}
int GCD(int m,int n)
/*
求两个整数的最大公约数，并返回公约数*/
{
    int r;
    r=m;
    do
    {
        m=n;
        n=r;
        r=m%n;
    }while(r);
    return n;
}

```

程序的输出结果如图2-42所示。

假设输入两个数15和25，在主函数中，将15和25分别赋值给实际参数a和b，通过语句s=GCD（a，b）调用实现函数GCD（int x，int y）也就是所谓的被调用函数，将15和25分别传递给被调用函数的形式参数m和n。然后求m和n的最大公约数，通过语句return n；将最大公约数5返回给主函数，即被调用函数，因此输出结果为5。

上述函数参数传递属于参数的单向传递，即a和b可以把值分别传递给m和n，而不可以把m和n传递给a和b。在传值调用中，实际参数和形式参数分别占用不同的内存单元，形式参数是实际参数的一个副本，实际参数和形式参数的值的改变都不会相互受到影响，如图2-43所示。这就像有一张身份证原件，它的复印件就是个副本，复印件的

丢失不会影响到身份证原件的存在，身份证原件的丢失也不会影响到复印件的存在。

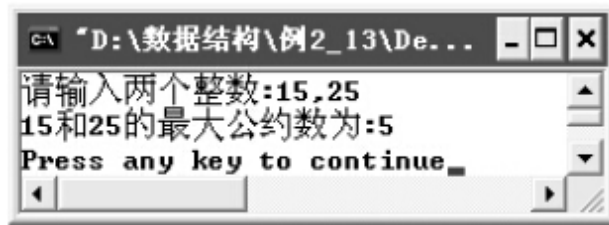


图2-42 求两个整数的最大公约数运行结果

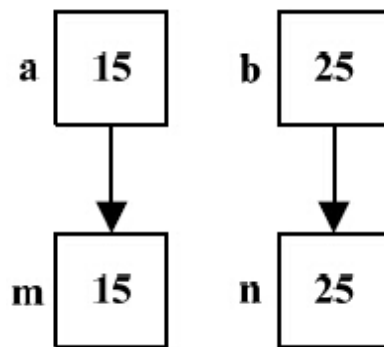


图2-43 参数传递过程

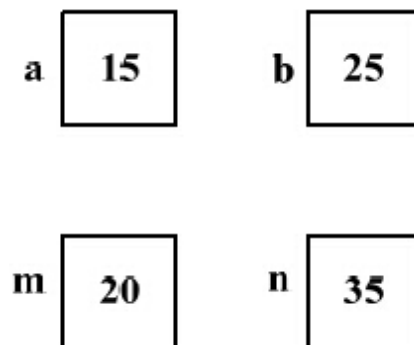


图2-44 形式参数改变后的情况

在调用函数时，形式参数被分配存储单元，并把15和25传递给形式参数，在函数调用结束，形式参数被分配的存储单元被释放，形式

参数不复存在，而主函数中的实际参数仍然存在，并且其值不会受到影响。在被调用函数中，如果改变形式参数的值，假设把m和n的值分别改变为20和35，a和b的值不会改变，如图2-44所示。

2.4.2 传地址调用

C语言通过指针（地址）实现传地址调用。在函数调用过程中，如果需要在被调用函数中修改参数值，则需要把实际参数的地址传递给形式参数，通过修改该地址的内容改变形式参数的值，以达到修改调用函数中实际参数的目的。

【例2-14】 编写一个求两个整数较大者和较小者的函数，要求用传地址方式实现。

【分析】 通过传地址调用的方式，把两个实际参数传递给形式参数。在被调用函数中，先比较两个形式参数值的大小，如果前者小于后者，则交换两个参数值，其中，前者为大，后者为小。传地址调用时，在调用函数和被调用函数中，对参数的操作其实都是在对同一块内存操作，实际参数和形式参数共用同一块内存。其程序实现如下。

```
#include <stdio.h>                /*
包含输入输出函数*/
void Swap(int *x,int *y);          /*
函数声明*/
void main()
{
    int a,b;
    printf("
请输入两个整数: \n");
    scanf("%d,%d",&a,&b);
```

```

        if(a<b)
            Swap(&a,&b);
/*
两个数中如果前者小，则交换两个值，使其较大的保存在a
中较小保存在b
中*/

    printf("
在两个整数%d
和%d
中，较大者为:%d,
较小者为:%d\n",a,b,a,b);
}
void Swap(int *x,int *y)
/*
交换两个数，较大的保存在*x
中，较小的保存在*y
中*/
{
    int z;
    z=*x;
    *x=*y;
    *y=z;
}

```

程序的运行结果如图2-45所示。

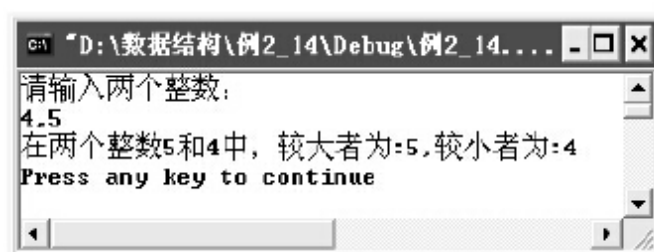


图2-45 传地址方式求两个整数的较大者和较小者的程序运行结果

在主函数中，如果 $a < b$ ，则调用 $\text{Swap}(\&a, \&b)$ 函数交换两个数。其中，实际参数是变量的地址，就是把地址传递给被调用函数 $\text{Swap}(\text{int}^*x, \text{int}^*y)$ 中的形式参数 x 和 y ， x 和 y 是指针变量，即指针 x 和 y 指向变量 a 和 b 。这样，交换 $*x$ 和 $*y$ 的值就是交换 a 和 b 的值。函数调用时，实际参数和形式参数的变化情况如图2-46所示。

如果要修改多个参数的值并返回给调用函数，该怎么办呢？这就需要
将数组名作为参数传递给被调用函数。数组名作为参数传递时，传
递的是整个数组。数组名是数组的首地址，如果把数组名作为实际参
数，在函数调用时，会把数组的首地址传递给形式参数。这样形式参
数就可以根据数组的首地址访问整个数组并对其操作，这是因为整个
数组元素的地址是连续的。

下面是一个数组名作为参数传递的例子。

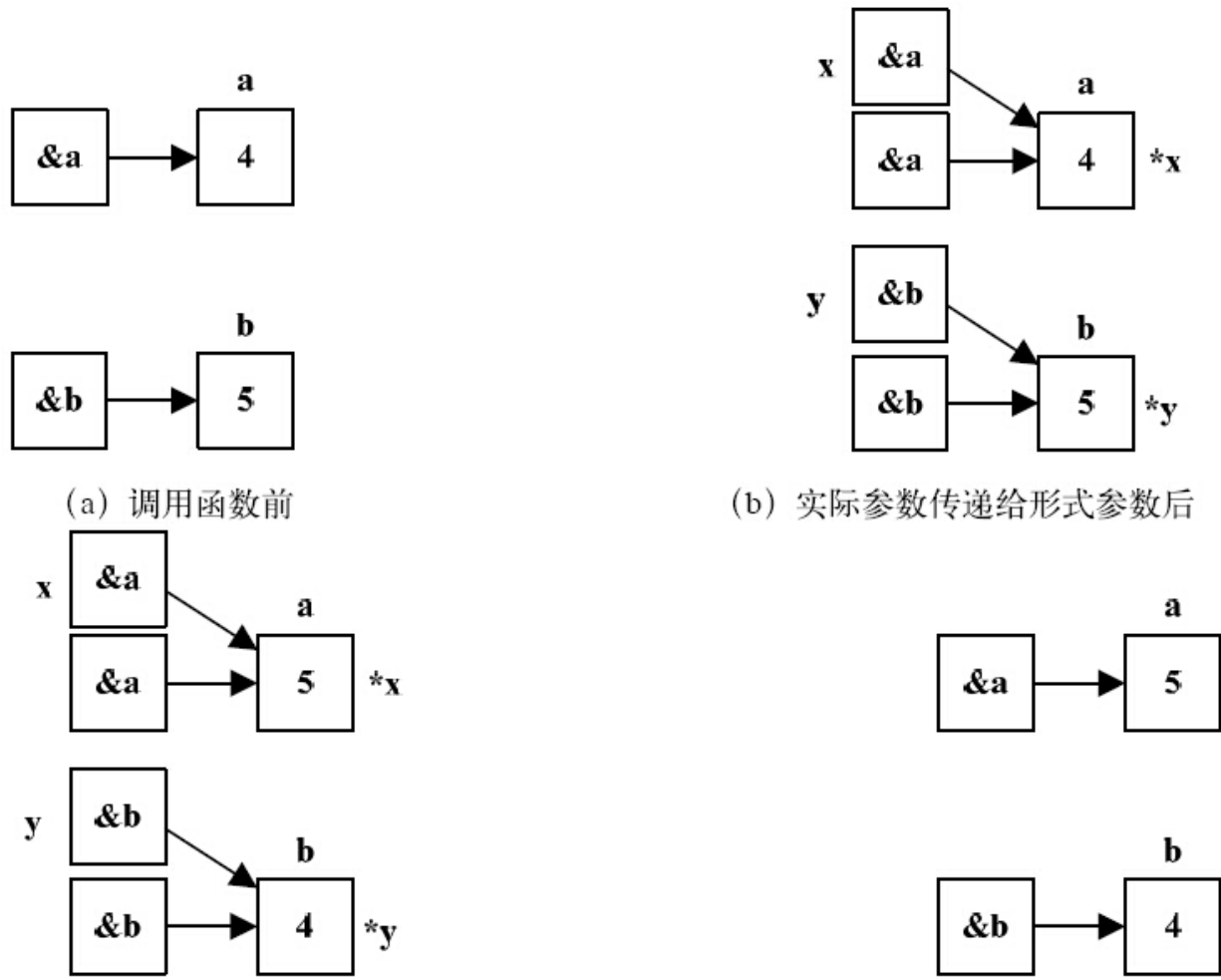


图2-46 实际参数和形式参数的变化情况

【例2-15】 编写函数，要求将数组中的n个元素的值分别减去20。

【分析】 数组名作为参数传递给被调用函数，实际上是把数组的起始地址传递给形式参数。因为数组在内存中存储的连续性，可以利用数组下标和指针访问数组中的每一个元素，这样在被调用函数中就可以对整个数组进行操作，无须将每一个数据元素作为参数传递给被调用函数。将数组名作为参数传递，调用函数和被调用函数都是对占同一块内存单元的数组进行操作。其程序实现如下。

```
#include <stdio.h>                                /*
包含输入输出函数*/
#define N 10
void SubArray1(int *x,int n);                      /*
数组名作为参数的函数原型*/
void SubArray2(int *aPtr,int n);                  /*
指针作为参数的函数原型*/
void main()
{
    int a[N]={51,52,53,54,55,56,57,58,59,60};
    int i;
    printf("
原来数组中的元素为:\n");
    for(i=0;i<N;i++)
        printf("%4d",a[i]);
    printf("\n");
    printf("
数组中的元素第一次减去20
之后为:\n");
    SubArray1(a,N);                                /*
调用SubArray1
: 数组名作为参数的函数*/
    for(i=0;i<N;i++)
        printf("%4d",a[i]);
    printf("\n");
    printf("
数组中的元素第二次减去20
之后为:\n");
    SubArray2(a,N);                                /*
调用SubArray2
: 指针作为参数的函数*/
    for(i=0;i<N;i++)
        printf("%4d",a[i]);
    printf("\n");
}
void SubArray1(int b[],int n)
/*
数组名作为参数，将数组中的元素减去20*/
```



```

{
    int i;
    for(i=0;i<n;i++)
        b[i]=b[i]-20;
}
void SubArray2(int *aPtr,int n)
/*
指针作为参数，将数组中的元素减去20*/
{
    int i;
    for(i=0;i<n;i++)
        *(aPtr+i)=*(aPtr+i)-20;
}

```

程序运行结果如图2-47所示。

该函数以两种方式实现了函数调用，即数组名作为形式参数和指针作为形式参数。在许多情况下，数组和指针效果是一样的。

在没有调用函数SubArray1 (int b[], int n) 之前，数组a在内存中的情况如图2-48所示。数组元素被保存在一个联系的存储单元中，数组名a指向数组的第一个元素。在调用函数SubArray1 (int b[], int n) 后，参数传递给数组名b，b也指向数组a的第一个元素，然后对所有元素减去20，如图2-49所示。调用函数SubArray2 (int*aPtr, int n) 后，参数传递给指针变量aPtr，aPtr也指向了数组a，并再一次把数组元素减去20，如图2-50所示。



图2-47 数组中的元素减去20后的程序运行结果

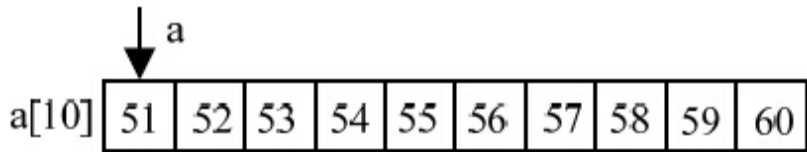


图2-48 未调用函数前

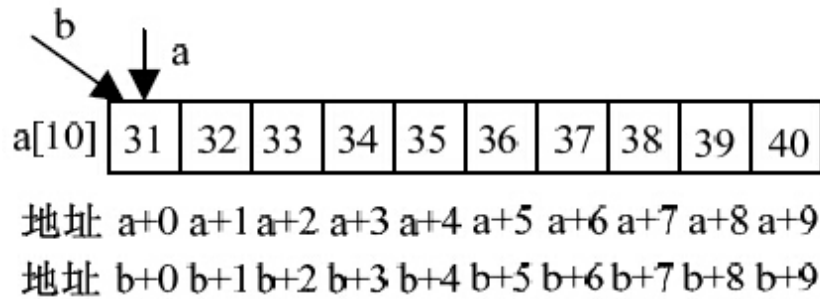


图2-49 第一次数组元素被减去20后

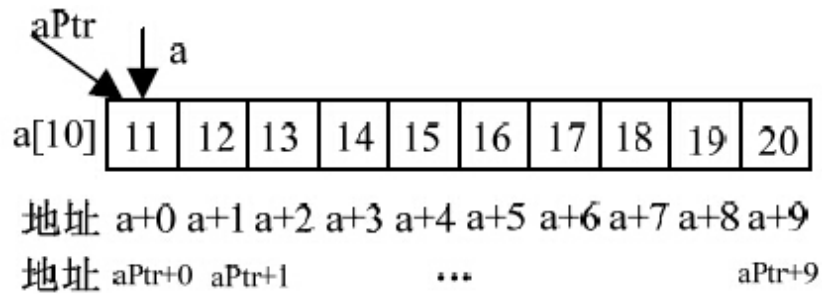


图2-50 第二次数组元素被减去20后

注意 在传值调用中，参数传递是单向传递，只能由实际参数传递给形式参数，而不能把形式参数反向传递给实际参数。而在传地址调用中，对形式参数的操作，即是对实际参数的操作，它们拥有同一块内存单元，属于双向传递。

2.5 结构体与联合体

结构体和联合体（也称共用体）是自定义的数据类型，用于构造非数值数据类型，在处理实际问题中应用非常广泛。数据结构中的链表、队列、树、图等结构都需要用到结构体。本节主要介绍结构体、联合体及其应用。

2.5.1 结构体的定义

一个教职工基本情况表包括编号、姓名、性别、职称、学历和联系电话等信息，每个数据信息的类型并不相同，使用前面学过的数据类型不能将这些信息有效组织起来。每一个教职工都包含编号、姓名、性别、职称、学历和联系电话等数据项，这些数据项放在一起构成的信息称为一个记录。例如，一个教师基本情况表如表2-1所示。

表2-1 教师基本情况表

编 号	姓 名	性 别	职 称	学 历	联 系 电 话
10101	王 欢	女	教授	博士	88308523
10102	刘和刚	男	教授	本科	88308233
10103	胡志刚	男	副教授	硕士	88308758

要用C语言描述表中的某一条记录，需要定义一种特殊的类型，这种类型就是结构体类型。它的定义如下。

```
struct teacher
/*
结构体类型*/
{
    int no;                /*
编号*/
    char name[20];         /*
姓名*/
    char sex[4];           /*
性别*/
    char headship[8];      /*
职称*/
    char degree[6];        /*
学历*/
    long int phone;        /*
... ..*/
}
```

```
联系电话*/  
};
```

其中，`struct teacher`就是新的数据类型——结构体类型，`no`、`name`、`sex`、`headship`、`degree`和`phone`为结构体类型的成员，表示记录中的数据项。这样，结构体类型`struct teacher`就可以完整地表示一个教师信息了。

定义结构体类型的一般格式如下。

```
struct  
结构体名  
{  
成员列表;  
};
```

`struct`与结构体名合在一起构成结构体类型，结构体名与变量名的命名规则一样。前面的`student`就是结构体名。使用一对花括号将成员列表括起来，在右花括号外使用一个分号作为定义结构体类型的结束。前面的`no`、`name`、`sex`等都是结构体类型的成员，每个成员需要说明其类型，就像定义变量一样。

注意 在定义结构体类型时，`struct`不可以省略。`struct`和结构体名一起构成结构体类型，例如，前面的`struct teacher`是结构体类型，而`teacher`只是结构体名。不要遗漏结构体外的分号，这是非常容易出错的地方。

像上面介绍的`struct teacher`类型是无法用C语言中的整型、浮点型、字符型等基本数据类型描述的，只能用其他数据类型构造出新的数据类型。例如，一个学生记录可能包括学号、姓名、性别、年龄及成绩等数据项，学生的记录可以用以下的结构体类型描述。

```
struct student  
{  
    char *no;  
    int *name;  
    char sex;  
    int age;  
    float score;  
};
```

其中，struct是关键字，表示是一个结构体类型，大括号里面是结构体的成员；struct student是一个类型名，如果要定义一个结构体变量，例如如下语句。

```
struct student stu1;
```

stu1就是类型为结构体struct student类型的变量。如果给结构体变量stu1的成员分别赋值，语句如下。

```
stu1.no="13001";
stu1.age=20;
stu1.name="Zhu Tong";
stu1.sex='m';
stu1.score=89.0;
```

则stu1的结构如图2-51所示。

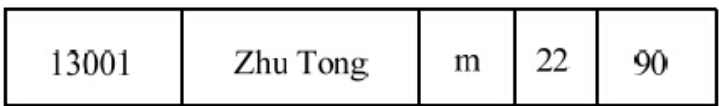


图2-51 stu1的结构

结构体的变量定义也可以和定义结构体类型同时定义。例如如下语句。

```
struct student
{
    char *no;
    int *name;
    char sex;
    int age;
    float score;
}stu1;
```

同样，也可以定义结构体数组类型。结构体变量的定义与初始化可以分开进行，也可以在结构体数组的定义的时候初始化，例如如下语句。

```
struct student
{
    char *no;
    char *name;
    char sex;
    int age;
    float score;
```

```
}stu[2]={{"13001","Zhu Tong",'m',22,90},
          {"13002","Guo Jing",'f',21,82}};
```

数组中各个元素在内存中的情况如图2-52所示。

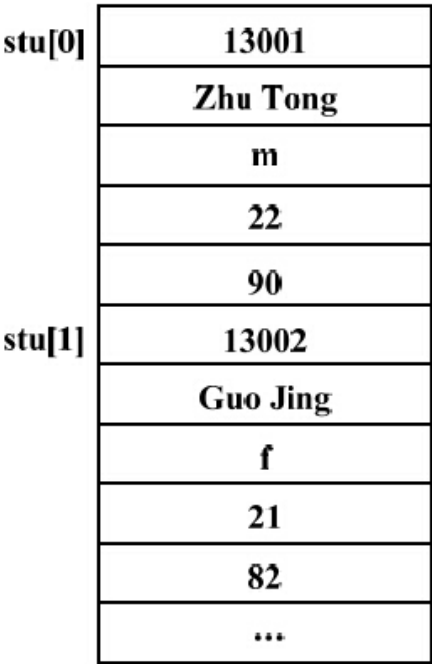


图2-52 结构体数组stu在内存中的结构

2.5.2 指向结构体的指针

指针可以指向整型、浮点型、字符等基本类型变量，同样也可以指向结构体变量。指向结构体变量的指针的值是结构体变量的起始地址。指针可以指向结构体，也可以指向结构体数组。指向结构体的指针和指向变量和指向数组的指针的用法类似。

【例2-16】 利用指向结构体数组的指针输出学生基本信息。

【说明】 指向结构体的指针与指向数组的指针一样，结构体中的成员变量地址是连续的，将指针指向结构体数组，就可直接访问结构体中的所有成员。程序实现

如下。

```
#include <stdio.h>                                /*
包含输入输出函数*/
#define N 10
/*
结构体类型及变量定义、初始化*/
struct student
{
    char *no;
    char *name;
    char sex;
    int age;
    float score;
}stu[3]={{"13001","Zhu Tong",'m',22,90.0},
        {"13002","Li Hua",'f',21,82.0},
        {"13003","Yang Yang",'m',22,95.0}};

void main()
{
    struct student *p;
    printf("
学生基本情况表:\n");
    printf("
学号
姓名
性别
年龄
成绩\n");
    for(p=stu;p<stu+3;p++)                        /*
通过指向结构体的指针输出学生信息*/
        printf("%-8s%-12s%-8c%-8d%-8.1f\n",p->no,p->name,p->sex,p->age,p->score);
}
```

程序运行结果如图2-53所示。

首先定义了一个指向结构体的指针变量p，在循环体中，指针指向结构体数组p=stu，即指针指向了结构体变量的起始地址。通过p->no、p->name等访问各个成员。如果p+1，表示数组中第2个元素stu[1]的起始地址。p+2表示数组中的第3个元素地址，如图2-54所示。

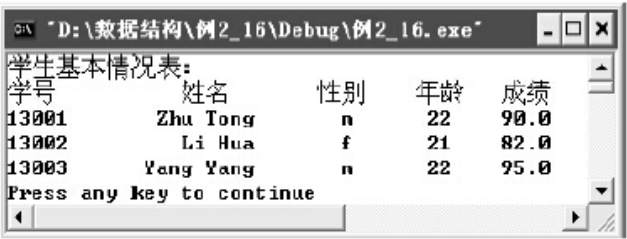


图2-53 通过结构体指针输出学生信息

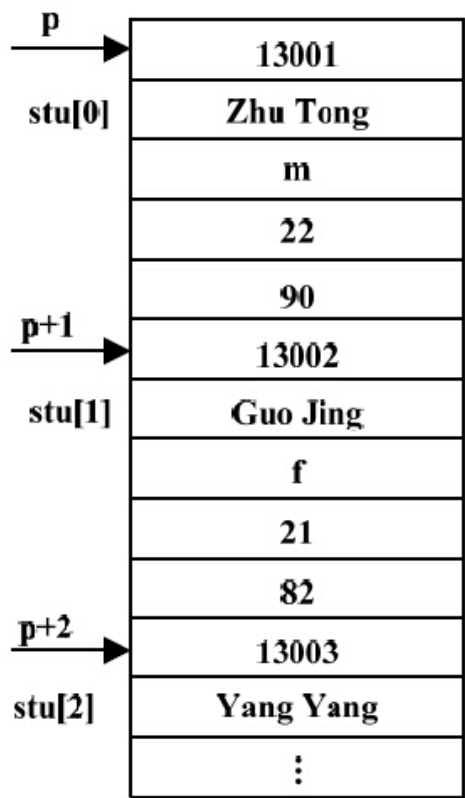


图2-54 指向结构体数组的指针在内存的情况

2.5.3 用typedef定义数据类型

通常情况下，在定义结构体类型时，使用关键字typedef为新的数据类型起一个好记的名字。typedef是C语言中的关键字，它的主要作用是为类型重新命名，一般形式如下。

```
typedef
类型名1
类型名2
```

其中，类型名1是已经存在的类型，如int、float、char、long等；也可以是结构体类型，如struct student。类型名2是你重新起的名字，命名规则与变量名的命名规则类似，必须是一个合法的标识符。

1. 使用typedef为基本数据类型重新命名

例如如下语句。

```
typedef int COUNT;           /*
将int
型重新命名为COUNT*/
typedef float SCORE;        /*
将float
型重新命名为SCORE*/
```

经过以上重新定义变量，COUNT就代表了int，SCORE就表示了float。这样，以上语句与如下语句等价。

```
int a,b,c;                   /*
定义int
型变量a
、 b
、 c*/
COUNT a,b,c;               /*
定义COUNT
型变量a
、 b
、 c*/
```

2. 使用typedef为数组类型重新命名

例如如下代码是将NUM定义数组类型。

```
typedef int NUM[20];         /*NUM
被定义为新的数组类型*/
```

代码表示NUM被定义为数组类型，该数组的长度为20，类型为int。可以使用NUM定义int型数组，代码如下。

```
NUM a;                       /*
使用NUM
定义int
型数组*/
```

a表示长度为20的int型数组，它与如下代码等价。

```
int a[20];                      /*
使用int
定义数组*/
```

3. 使用typedef为指针类型重新命名

使用typedef为指针类型变量重新命名与重新命名数组类型的方法是类似的。例如如下语句。

```
typedef float *POINTER;         /*POINT
被定义为指针类型*/
```

POINTER表示指向float类型的指针类型。如果要定义一个float类型的指针变量p，代码如下。

```
POINTER p;                      /*
使用POINTER
定义指针变量*/
```

p被定义为指向float类型的指针变量。同样，也可以使用typedef重新为指向函数的指针类型命名，例如定义一个函数指针类型，代码如下。

```
typedef int (*PTR)(int,int);    /*PTR
被定义为函数指针类型*/
```

PTR被定义为函数指针类型，PTR是指向返回值为int且有两个int型参数的函数指针。如下语句使用PTR定义变量。

```
PTR pm;                          /*
使用PTR
定义一个函数指针变量pm*/
```

pm被定义为一个函数指针变量。

4. 使用typedef为用户自定义数据类型重新命名

用户自己定义的数据类型主要包括结构体、联合体、枚举类型，最为常用的是为结构体类型重新命名，联合体和枚举类型的命名方法与结构体的重新命名方法类似。例如，将一个结构体命名为DATE，代码如下。

```
typedef struct                                /*  
为结构体类型重新命名*/  
{  
    int year;  
    int month;  
    int day;  
}DATE;
```

从类型名DATE可以很容易看出，DATE是表示日期的类型。上面的类型重新定义是在定义结构体类型的同时为结构体命名；也可以先定义结构体类型，然后重新为结构体命名，代码如下。

```
struct date                                  /*  
定义结构体类型*/  
{  
    int year;  
    int month;  
    int day;  
};  
typedef struct date DATE;                    /*  
为结构体类型重新命名*/
```

以上两段代码是等价的。注意，date和DATE是两个不同的名字，C语言是区分大小写的。接下来，就可以使用DATE定义变量了，代码如下。

```
DATE d;                                      /*  
定义变量d*/
```

上面的变量定义与如下变量定义等价。

```
struct date d;
```

2.5.4 联合体

与结构体一样，联合体也是一种派生的数据类型。但是与结构体不同的是，联合体的成员共享同一个存储空间。

1. 定义联合体类型及变量

定义联合体的方式与定义结构体的方式类似，定义联合体的一般形式如下。

```
union
共用体名
{
成员列表;
}
变量列表;
```

其中，union是C语言的关键字，用来定义联合体类型。例如，一个联合体的类型定义如下。

```
union data;
定义联合体类型和变量abc*/
{
    int a;
    float b;
    char c;
    double d;
}abc;
```

以上代码是将联合体类型与变量同时定义，当然也可以先定义联合体类型，然后定义变量，例如如下语句。

```
union data                                     /*
定义联合体类型union abc*/
{
    int a;
    float b;
    char c;
    double d;
};
union data abc;                               /*
定义联合体类型的变量abc*/
```

当然也可以省略联合体名data，代码如下。

```
union /*
省略了联合体名data*/
{
```

```
int a;  
float b;  
char c;  
double d;  
};  
union data abc;          /*  
定义联合体类型的变量abc*/
```

以上3段代码是等价的。

从联合体的类型定义可以看出，联合体与结构体的定义非常相似。但是联合体与结构体在存储方式上却是不同的。联合体中的成员在同一时刻只有一个有效，联合体中的成员占用同一块内存单元。上面定义的变量abc在内存中的情况如图2-55所示。

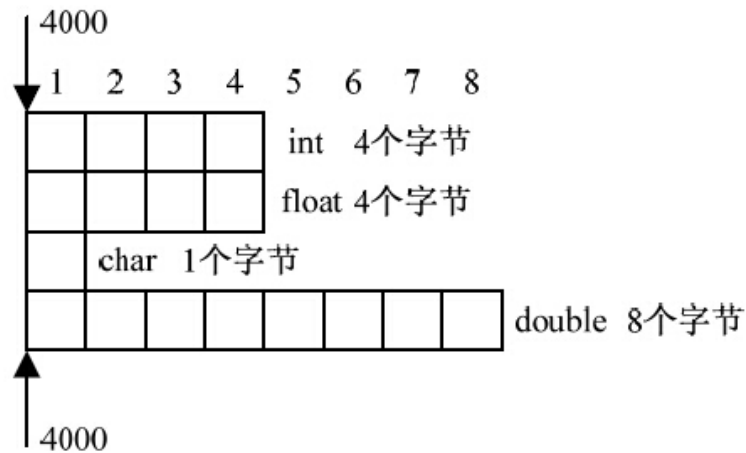


图2-55 变量abc在内存中的情况

其中，变量abc中包含4个成员，它以占用内存单元最长的成员作为变量的长度。因此，abc占用8个字节，而不是 $4+4+1+8=17$ 个字节。

2. 引用联合体成员变量

引用联合体成员变量的方式与引用结构体成员变量的方式相同。例如，前面定义了联合体变量abc，引用变量中的成员的代码如下。

```
abc.a          /*
引用联合体变量中的成员a*/
abc.b          /*
引用联合体变量中的成员b*/
abc.c          /*
引用联合体变量中的成员c*/
```

不可以整体对联合体变量进行输入和输出，以下代码的写法是错误的。

```
scanf("%d",&abc); /*
错误！试图整体为联合体变量输入数据*/
printf("%d",&abc); /*
错误！试图整体输出联合体变量的值*/
```

正确的写法应该是分别输入和输出成员变量的值，代码如下。

```
scanf("%d",&abc.a); /*
正确！为联合体变量的成员a
输入数据*/
printf("%f",&abc.b); /*
正确！输出联合体变量的成员b
的值*/
```

3. 使用联合体应该注意的问题

在使用联合体时，需要注意以下几个问题。

(1) 联合体变量中的成员共同占有同一块内存单元，同时只能有一个成员存放其中。同一时刻，只能有一个成员起作用，其他成员不起作用。

(2) 联合体变量有效的成员是最后被赋值的成员，每存入一个新的数据，前面的成员的值不起作用。例如如下代码。

```
abc.a=5;
abc.b=7.5;
abc.c='r';
```

经过3次赋值之后，只有最后一个赋值语句有效，即联合体变量abc中的成员c的值为字符'r'，成员a、b、d中的值没有意义。

(3) 联合体变量中的各个成员的地址都是相同的，每个成员的存放都是从这个地址开始存放。例如&abc.a、&abc.b、&abc的值都是相同的。

(4) 不能对联合体变量像结构体一样地赋值。这是因为同一时刻只能有一个成员有效。例如如下代码是错误的：

```
union data                                     /*
定义联合体类型*/
{
    int a;
    float b;
    char c;
    double d;
}abc={3,7.9,'x',5.6};                          /*
错误！同一时刻只能有一个成员有效，不能这样赋值*/
```

(5) 不能将联合体变量作为函数的参数。

4. 联合体应用举例

【例2-17】 建立一个教师和学生基本情况登记表，其中，教师基本情况由编号、姓名、性别、年龄、职业和职称构成，学生基本情况由编号、姓名、性别、年龄、职业和班级构成。

【分析】 教师和学生的基本情况基本相同，只有一项不同，教师有职称而没有班级，学生有班级而没有职称，因此可以将职称和班级放在一起，构成一个联合体。将这个联合体类型与编号、姓名、性别、年龄、职业放在一起构成一个结构体，结构体的定义如下。

```
struct STAFFER                                /*
定义结构体类型*/
{
    int num;
    char name[20];
    char sex[4];
    int age;
    int job;
    union                                     /*
定义联合体类型*/
    {
        int class;
        char prof[20];
    }category;
    . . . . .
```

```
定义联合体变量*/
};
```

结构体类型为struct STAFFER，为了简便，可以将结构体类型重新命名，结构体类型的定义可以写成如下形式。

```
typedef struct
{
    int num;
    char name[20];
    char sex[4];
    int age;
    int job;
    union
    {
        int class;
        char prof[20];
    }category;
}STAFFER;
/*
定义联合体类型*/
/*
定义联合体变量*/
/*
结构体类型为STAFFER*/
```

使用typedef重新对结构体定义后，结构体类型为STAFFER。完整的程序代码如下。

```
#include<stdio.h>
/*-----
定义结构体类型STAFFER-----*/
typedef struct
{
    int num;
    char name[20];
    char sex[4];
    int age;
    int job;
    union
    {
        int class;
        char prof[20];
    }category;
}STAFFER;
/*-----
结构体类型定义结束-----*/
void main()
{
    STAFFER staf[20];
    int i;
    /*-----
    输入3
    条记录-----*/
    for(i=0;i<3;i++)
    {
        printf("
编号: ");
        scanf("%d",&staf[i].num);
        /*
        输入编号*/
        printf("
姓名: ");
        scanf("%s",staf[i].name);
        /*
        输入姓名*/
```



```

        printf("
性别: ");
scanf("%s", staf[i].sex);          /*
输入性别*/
        printf("
年龄: ");
scanf("%d", &staf[i].age);          /*
输入年龄*/
        printf("
职业 (1
表示教师, 0
表示学生)
: ");
scanf("%d", &staf[i].job);          /*
输入职业*/
        if(staf[i].job==1)          /*
如果输入的是1
, 表示教师*/
        {
            printf("
职称: ");
scanf("%s", staf[i].category.prof);    /*
输入职称*/
        }
        else if(staf[i].job==0)    /*
如果输入的是0
, 则表示学生*/
        {
            printf("
班级: ");
scanf("%d", &staf[i].category.class);    /*
输入班级*/
        }
    }
/*-----
记录输入结束-----*/
/*-----
输出记录-----*/
    printf("
编号
姓名
性别
年龄
职业
班级/
职称\n");
    for(i=0; i<3; i++)
    {
        if(staf[i].job==1)          /*
如果是教师*/
            printf("%7d%7s%5s%6d%6s%8s\n", staf[i].num, staf[i].name, staf[i].sex,
            staf[i].age, "
教师", staf[i].category.prof);
        else if(staf[i].job==0)    /*
如果是学生*/
            printf("%7d%7s%5s%6d%6s%8d\n", staf[i].num, staf[i].name, staf[i].sex,
            staf[i].age, "
学生", staf[i].category.class);
    }
}
/*-----
记录输出结束-----*/

```

程序运行结果如图2-56所示。

```

D:\数据结构\例2_17\Debug\例2_17.exe
编号: 201301005
姓名: 吴林霜
性别: 女
年龄: 23
职业<1表示教师, 0表示学生>: 0
班级: 1302
编号: 201301005
姓名: 张燕
性别: 女
年龄: 35
职业<1表示教师, 0表示学生>: 1
职称: 副教授
编号: 201301028
姓名: 王林涛
性别: 男
年龄: 36
职业<1表示教师, 0表示学生>: 1
职称: 讲师
  编号  姓名  性别  年龄  职业  班级/职称
201301005 吴林霜 女  23  学生  1302
201301005 张燕  女  35  教师  副教授
201301028 王林涛 男  36  教师  讲师
Press any key to continue

```

图2-56 程序运行结果

2.6 链表

在C语言中，处理已知数量的数据可以使用数组。如果事先并不知道要处理的数据的个数，则需要使用链表结构。链表需要动态分配内存，链表的长度随时可以发生变化。在今后学习数据结构的过程中，数据结构的单链表、树、图等结构还需要用到链表结构。本节主要给大家介绍内存的动态分配、释放及链表。

2.6.1 内存的动态分配与释放

在C语言中，内存的动态分配与释放往往离不开malloc函数和free函数。

1. malloc——动态内存分配函数

函数malloc的主要作用是分配一块长度为size的内存空间。函数原型如下。

```
void *malloc(unsigned int size);
```

其中，参数size是要分配的内存空间的大小（字节），这一块内存空间是连续的。如果分配成功，则函数的返回值是一个指向新分配内存空间的起始位置。如果分配不成功，则返回NULL。

（1）函数malloc常常与运算符sizeof配合使用。例如，要分配一个大小为20的int型的内存空间，代码如下。

```
int *p;                                /*
定义一个int
型的指针变量p*/
p=(int*)malloc(sizeof(int)*20);        /*
分配一个大小为20
的int
型的内存空间，p
指向新开辟的空间首地址*/
```

因为这是为int型的数据开辟内存空间，所以需要将函数的返回值void*转换为int*。当然，以上代码也可以放在一起写成一行，代码如下。

```
int *p=(int*)malloc(sizeof(int)*20);           /*p
指向新分配一个20
个int
型的内存空间的首地址*/
```

以上两段代码是等价的。

(2) 也可以为一个结点类型分配内存空间，代码如下。

```
int *p;                                           /*
定义一个int
型的指针变量p*/
p=(struct student*)malloc(sizeof(struct student)); /*
分配一个大小为1
的struct student
型的内存空间*/
```

其中，struct student是上面定义的结构体类型，表示一个结点。通过以上动态分配就可以得到一个结点的内存空间，然后就可以将数据存入该结点了。

2. free——动态内存释放函数

函数free的主要作用是将动态分配的内存空间释放。它的函数原型如下。

```
void free(void *p);
```

其中，参数p指向要释放的内存空间。函数free没有返回值。

函数原型malloc和free都在头文件stdlib.h和alloc.h中定义。使用动态内存分配函数和动态内存释放函数需要注意以下几点。

- 在调用函数malloc分配内存空间时，最好检查下是否分配成功，即检查函数的返回值是否等于NULL，以保证程序的正确运行。
- 在使用完由malloc函数分配的内存空间时，要利用free函数将该内存空间及时释放。
- 不能使用已经被free函数释放的内存空间。

2.6.2 什么是链表

链表是一种常用的数据结构。链表是一种特殊的结构体类型，它有一个指针类型的成员指向自身，该指针指向与结构体一样的类型。例如如下语句。

```
struct node
{
    int data;
    struct data *next;
};
```

这就是一种自引用结构体类型。自引用结构体类型为struct node，该结构体类型有两个成员：一个是整型成员data，一个是指针成员next。成员next是指向结构体为struct node类型的指针。通过这种形式定义的结构体通过next指针把两个结构体变量连在一起，如图2-57所示。

这种自引用结构体单元称为**结点**。结点之间通过箭头连接起来，构成一个表，称为**链表**。链表中指向第一结点的指针称为头指针，通过头指针，可以访问链表的每一个结点。链表的最后一个结点的指针部分用空（^）表示。为了方便操作，在链表的第一个结点之前增加一个结点，称为**头结点**。一个带头结点的链表如图2-58所示。

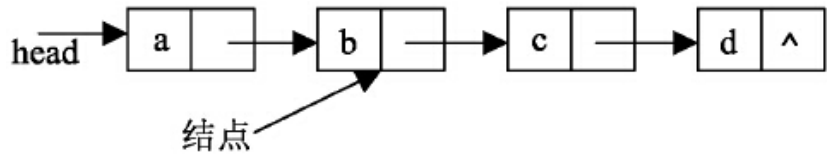


图2-57 不带头结点的链表

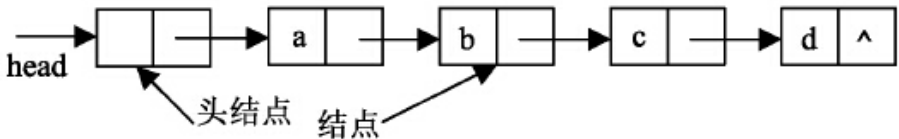


图2-58 带头结点的链表

2.6.3 创建链表

创建链表首先需要生成结点，然后将结点利用指针域连接起来，这样就构成了一个链表。只是生成链表时，使用的是动态内存分配。假设要创建的链表的结点类型定义如下。

```

struct student                                /*
定义结点*/
{
    long no;                                  /*
学号*/
    char name[20];                            /*
姓名*/
    char addr[30];                            /*
地址*/
    struct student *next;                     /*
指向结构体的指针*/
};

```

1. 初始时，链表为空

定义3个指向struct student类型的指针变量h、prev和cur，h代表头指针，指向第一个结点。初始时，令h=NULL，表示链表为空。链表为空就是链表这没有结点存在，头指针head的值为NULL。代码如下。

```

LIST *h=NULL,*prev,*cur;

```

初始状态如图2-59所示。

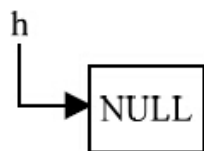


图2-59 空链表

2. 在链表中插入第1个结点

①动态生成一个结点，由指针cur指向该结点，代码如下。

```
cur=(struct student *)malloc(sizeof(struct student));          /*  
动态分配结点*/
```

②因为这是第一个结点，所以使用头指针h指向该结点，代码如下。

```
h=cur;
```

③此时链表中只有一个结点，因此将该结点的next域置为NULL，代码如下。

```
cur->next=NULL;
```

④最后需要为该结点输入数据，代码如下。

```
scanf("%d %s %s",&cur->no,cur->name,cur->addr);
```

输入的数据如下。

```
10901  
郭庆峰  
北京<  
回车>
```

⑤令pre指针指向该结点。pre指针指向的结点表示要插入结点的前一个结点。在插入一个新结点后，当前结点变为下一个结点的前一个结点，需要使用pre指向该结点。代码如下。

```
prev = cur;
```

此时，链表的状态如图2-60所示。

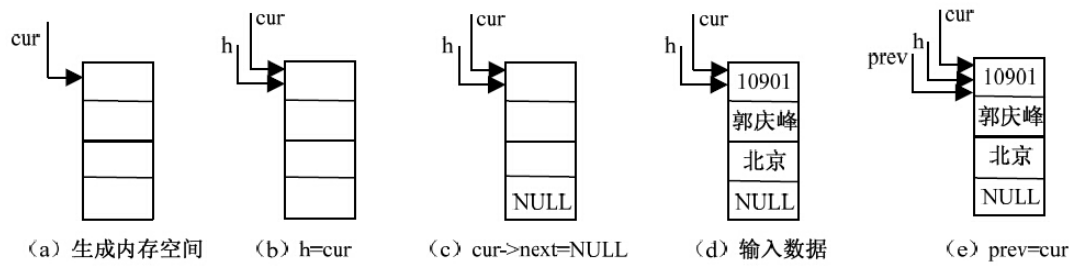


图2-60 插入第一个结点的过程

3. 插入链表的第2个结点

①动态生成一个结点，由指针cur指向该结点，代码如下。

```
cur=(struct student *)malloc(sizeof(struct student)); /*  
动态分配结点*/
```

②因为当前结点一定是最后一个结点，所以将next指针域置为NULL，代码如下。

```
cur->next=NULL;
```

③令指针prev的next域指向当前结点，使cur指向的结点成为链表中的最后一结点，代码如下。

```
prev->next=cur;
```


④为该结点输入数据，代码如下。

```
scanf("%d %s %s",&cur->no,cur->name,cur->addr);
```

输入的数据如下。

```
10902
王志恒
陕西<
回车>
```

⑤将prev指向最后一个结点，代码如下。

```
prev = cur;
```

此时，链表的状态如图2-61所示。

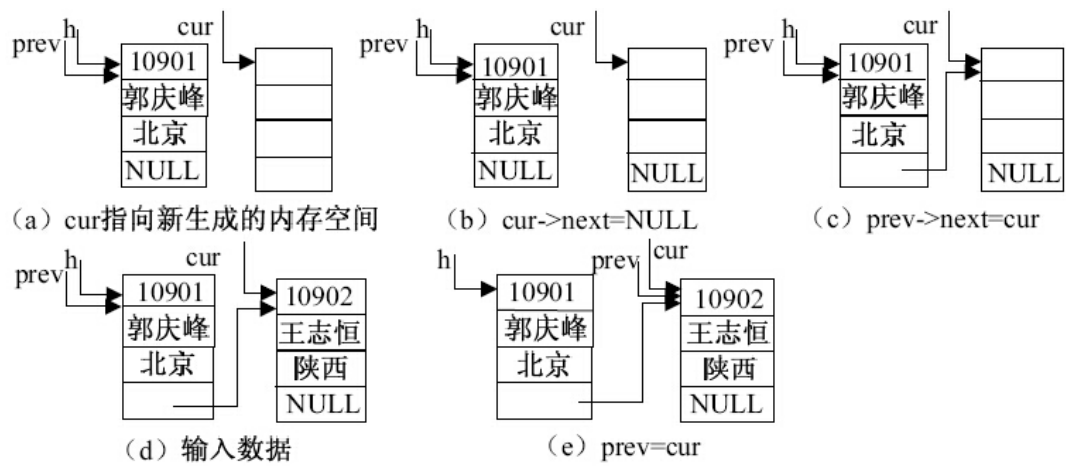


图2-61 在链表中插入第2个结点的过程

4. 插入第3个结点

在链表的末尾插入第3个结点的方法与第3步的方法类似，过程如图2-62所示。

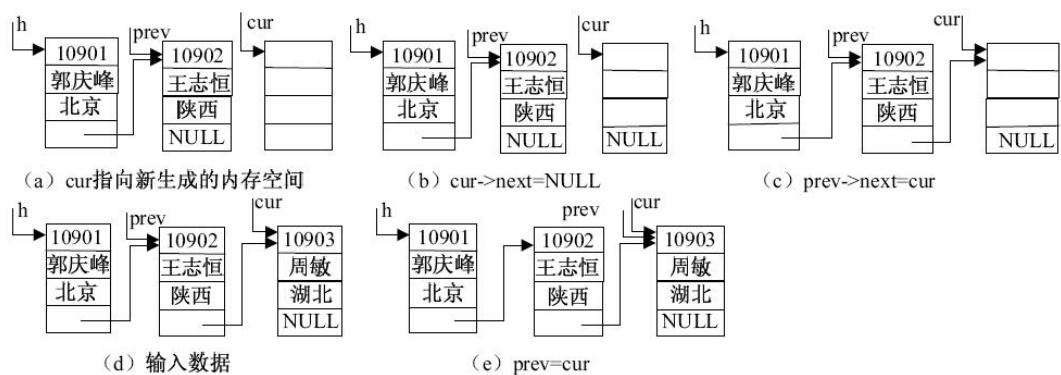


图2-62 在链表中插入第3个结点的过程

5. 按照以上方法插入第4个结点

插入的过程如图2-63所示。

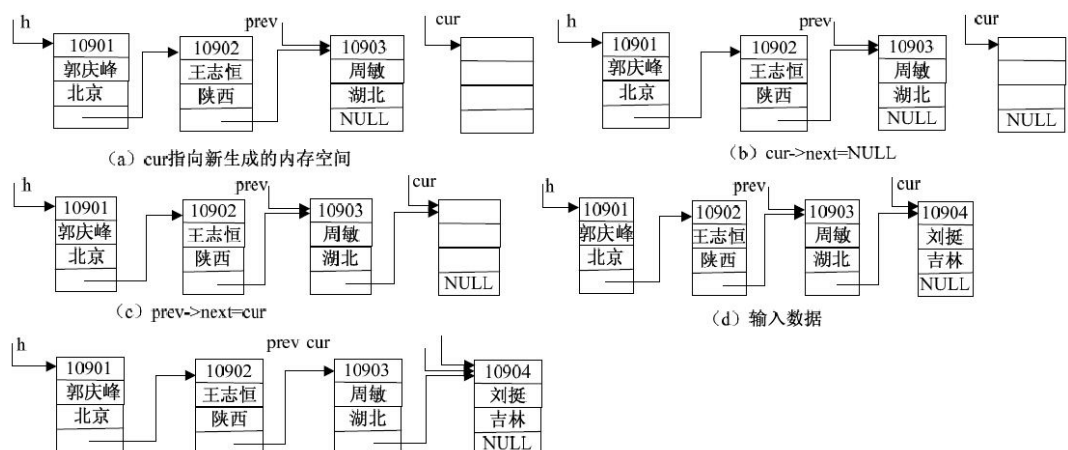


图2-63 在链表中插入第4个结点的过程

这样，经过以上操作就构造出了一个具有4个结点的链表。

经过以上分析，得到创建链表的程序如下。

```

struct student
/*
定义结点*/
{
    long no;
    /*
    学号*/
    char name[20];
    /*
    姓名*/
    char addr[30];
    /*
    ...

```

```

地址*/
    struct student *next;
    /*
指向结构体指针*/
};
typedef struct student LIST;
/*
将类型重新命名为LIST*/
LIST *CreateList()
/*
创建链表的函数定义*/
{
    LIST *h,*prev,*cur;
    /*
定义指向LIST
的指针变量*/
    int i,n;
    /*
定义变量*/
    h=NULL;
    /*
初始时头指针h
为空*/
    printf("
输入结点个数: \n");
    scanf("%d",&n);
    /*
输入链表中结点的个数*/
    for(i=0;i<n;i++)
    {
        cur=(LIST *)malloc(sizeof(LIST));
        /*
动态生成一个结点空间*/
        cur->next=NULL;
        /*
将cur
的next
置为NULL*/
        if(h==NULL)
            /*
如果h=NULL
, 表示当前正在处理第一个结点*/
            h=cur;
            /*
令h
指向第一个结点*/
        else
            /*
如果不是第一个结点*/
            prev->next=cur;
            /*
令链表中最后一个结点的next
指向cur*/
            scanf("%d %s %s",&cur->no,cur->name,cur->addr);
            /*
为cur
指向的结点输入数据*/
            prev = cur;
            /*
将prev
指向最后一个结点*/
        }
        return h;
        /*
返回头指针h*/
    }
}

```

初始时，链表为空，头指针h为NULL。在链表中，将h==NULL作为链表为空的条件。指针prev永远指向链表的最后一个结点，当要插入cur指向的结点，则需要让prev指向的结点的next域指向cur指向的结点，这样就将cur指向的结点插入了链表，成为最后一个结点，主要通过以下代码实现。

```
prev->next=cur;
```

然后让prev指向最后一结点，准备下一次插入新结点。在函数的最后，将头指针h返回给调用函数。这样调用函数就可以利用头指针访问链表中的每一个结点。

2.6.4 链表的输出操作

链表的输出操作就是输出链表中每个结点的数据。要输出链表中的结点的数据，需要从链表的头指针出发，令指针p指向第一个结点，输出p指向的结点的数据。然后通过结点的指针域next找到下一个结点，并输出数据。依次类推，直到最后一个结点。

输出链表中的每个结点的数据的过程如图2-64所示。

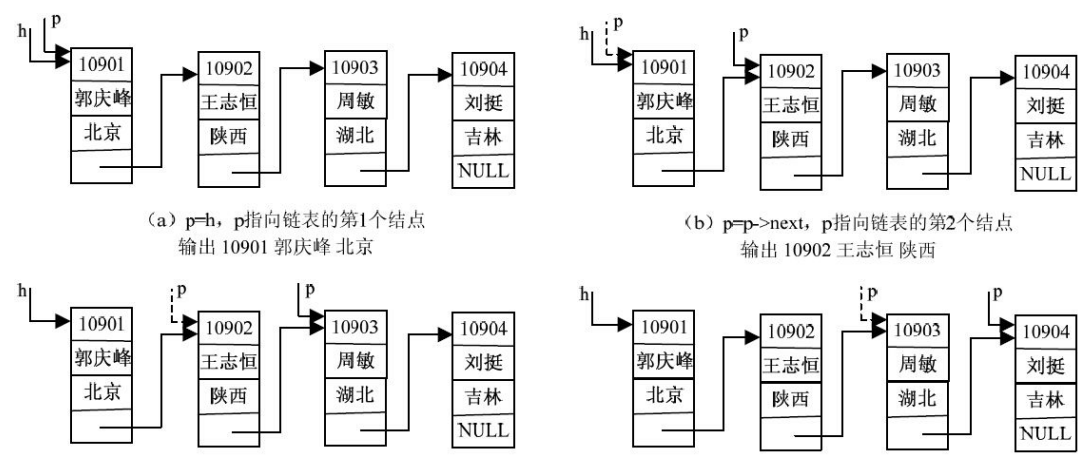


图2-64 输出链表中每个结点记录的过程

图中的虚线表示前一个指针所在的位置。在（a）图中，p指向第一个结点，并输出该结点的数据。然后令 $p=p \rightarrow next$ ，p指向了第二个结点，并输出第二个结点的数据。当p指向最后一个结点时， $p \rightarrow next$ 为NULL，表示该结点之后没有结点，停止输出。

在输出过程中，主要通过如下代码找到下一个结点。

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

将p->next赋值给p，而p->next是下一个结点的地址，也就是说将下一个结点的地址赋值给p，这样p的地址就是下一个结点的地址，即p指向下一个结点。

完整的链表输出操作的程序代码如下。

```
void DispList (LIST *h)
{
    LIST *p=h;                               /*
    定义指针p
    并指向链表的第一个结点*/
    while (p!=NULL)                             /*
    如果p
    不为NULL*/
    {
        printf("%d %s %s\n",p->no,p->name,p->addr);    /*
    输出p
    指向结点的数据*/
        p=p->next;                               /*
    指针p
    指向下一个结点*/
    }
}
```

下面编写一个main函数，在main函数中调用CreateList和DispList，测试一下程序的正确性，代码如下。

```
LIST *CreateList();
void DispList (LIST *h);
void main()
{
    LIST *head;
    head=CreateList();
    printf("
    学号
    姓名
    地址\n");
    DispList(head);
}
```

程序的运行结果如图2-65所示。

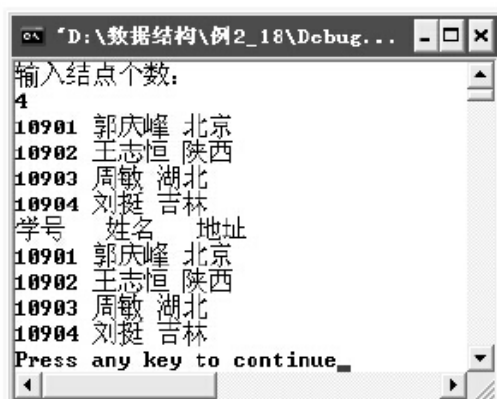


图2-65 程序运行结果

2.6.5 链表的插入操作

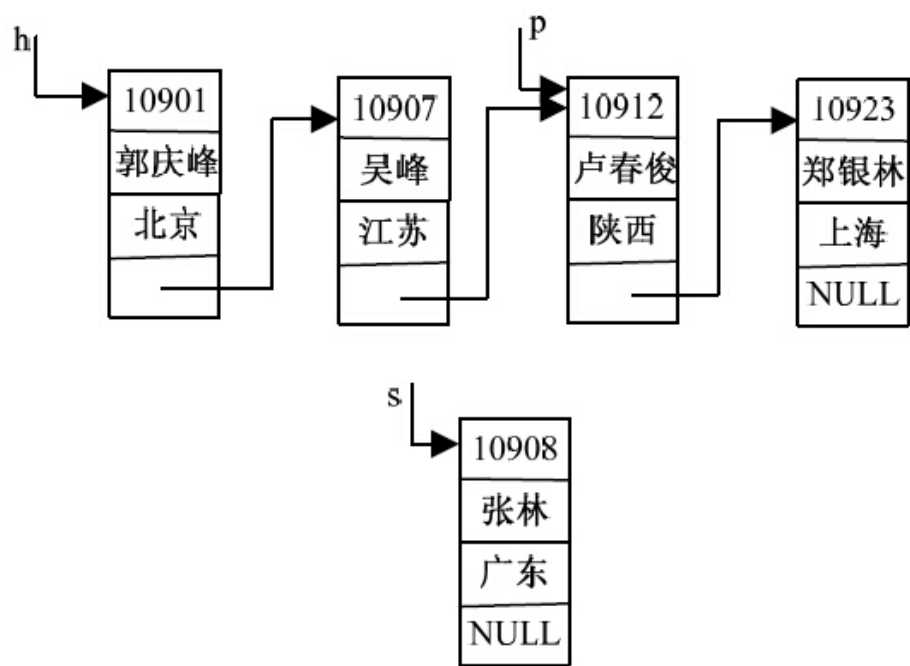
链表的插入操作是指将一个结点插入到链表中。在由学生构成的链表中，结点是按照学号的从小到大进行排列。如果将一个结点插入到链表中，需要处理以下两个问题：找到要插入的位置和如何插入新结点。

1. 寻找插入的位置

要在链表中寻找插入的位置，先看生活中的一个例子。有一群学生手拉手按照个子从低到高排成一队，要将一个新学生插入其中，要求这一队学生仍然是按照个子从低到高排列。这可以将这个新学生与第一个学生进行比较，如果它的个子比第一个学生高，则与第二个学生的个子比较，如果比第二个学生的个子高，则继续与第三个学生进行比较，如果它的个子低于第三个学生的个子，则将其插入到第三个学生之前第二个学生之后。

在链表中寻找插入的位置与此类似。从链表的第一个结点出发，将新结点的学号与链表中的结点的学号进行比较，如果新结点的学号比前一个结点的学号大，而小于后一个结点的学号，则将新结点插入到两者之间。例如，s指向要插入

的结点，它的学号是10908，需要将s->no依次与链表中第一个结点先进行比较，因为s->no>p->no，所以将指针p向后移动，即p=p->next，p指向第二个结点，因为s->no>p->no，所以继续让p向后移动，即执行p=p->next，此时p指向第三个结点，因为s->no<=p->no，则要插入的位置应该是第二个结点与第三个结点之间，如图2-66所示。



s->no<=p->no，s指向的结点应插入到第2个结点与第3个结点之间

图2-66 s指向的结点寻找插入的位置

2. 插入新结点

新结点要插入的位置有3种可能，即在第一个结点之前、在第一个结点和最后一个结点之间、在最后一个结点之后。

(1) 将新结点插入第一个结点之前

如果要插入的结点的学号小于第一个结点的学号，则要将该结点插入第一个结点之前。例如，链表的第一个结点的学号是10903，而要插入的结点的学号是10902，因为 $s \rightarrow no < h \rightarrow no$ ，则需要将s指向的结点插入到h指向的结点之前，如图2-67所示。

插入过程分为两步操作，第1步需要将s指向的结点的指针域指向第一个结点，即 $s \rightarrow next = h$ ；第2步令头指针h指向s指向的结点，即 $h = s$ ，这是因为s指向的结点变为了第一个结点。

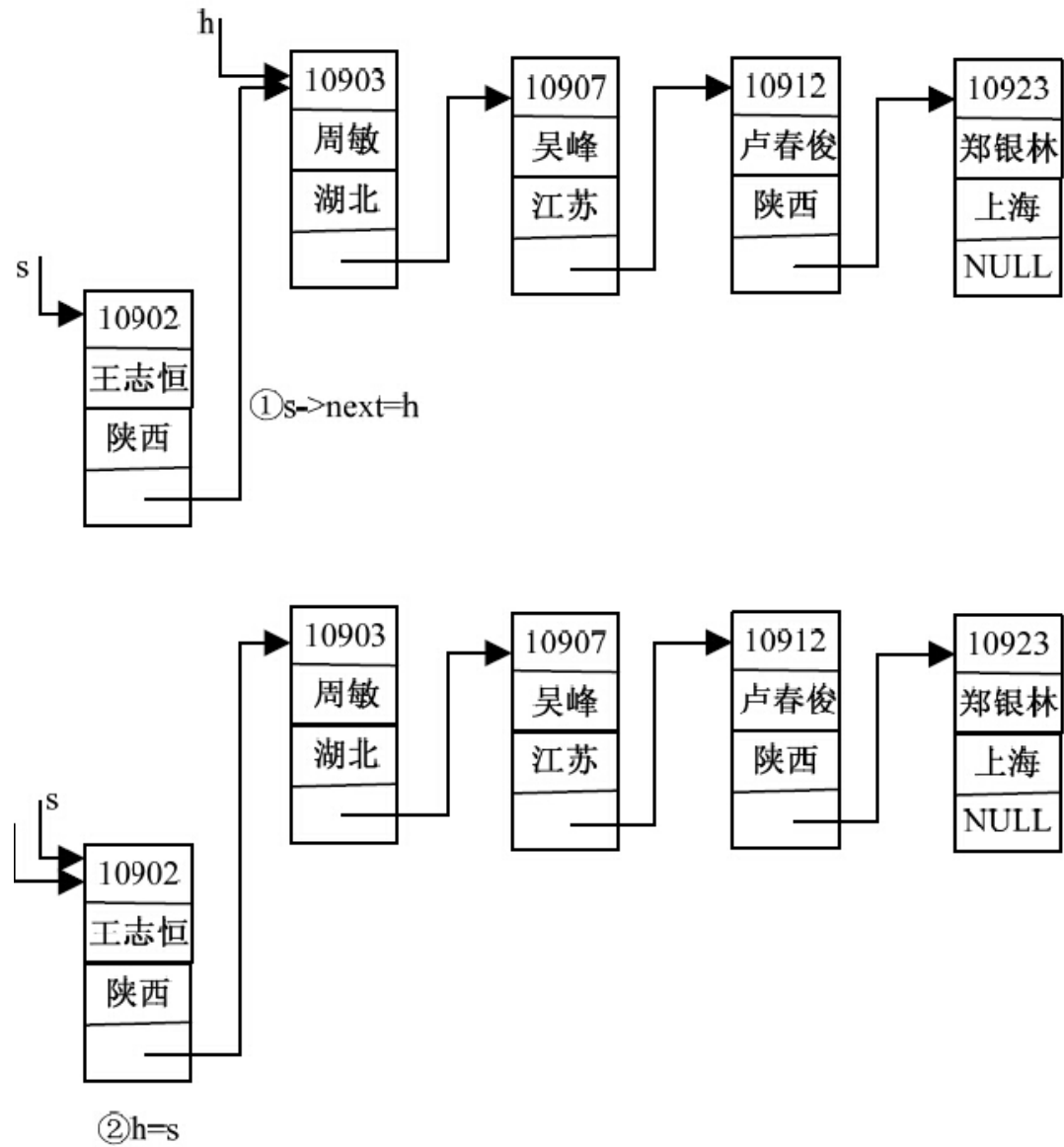


图2-67 在第一个结点之前插入新结点的过程

(2) 将新结点插入链表的中间

插入的新结点位于链表中某一个结点，不是第一个结点之前，也不是最后一个结点之后，这种情况如图2-68所示。s指向的结点应该插入到第2个结点和第3个结点之间，其中pre指向要插入的位置之前的结点，p指向要插入的位置之后的结点。

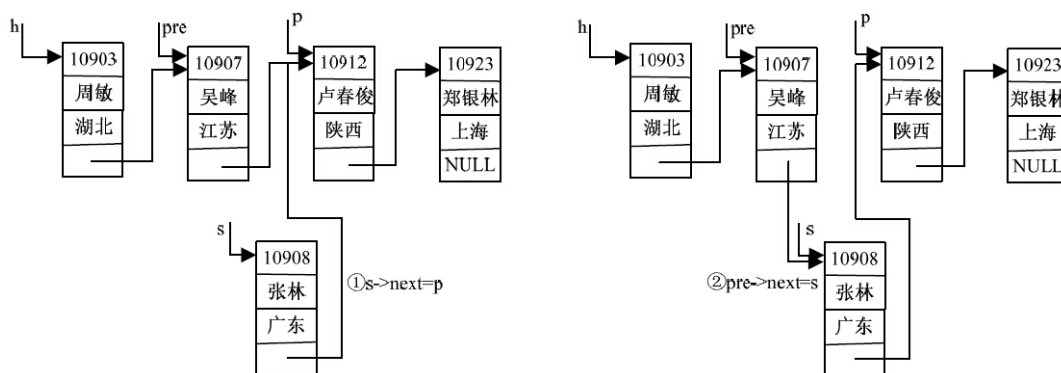


图2-68 在链表的中间插入一个新结点的过程

要将s指向的结点插入到链表中，需要进行以下两步操作，第1步将s指向的结点的指针域指向p，即 $s \rightarrow next = p$ ；第2步将pre指向的结点的指针域指向s指向的结点，即 $pre \rightarrow next = s$ 。需要说明的是，这两个步骤可以相互交换，不会影响到程序的结果。

(3) 将新结点插入链表的最后一个结点之后

如果新结点的学号大于最后一个结点，则需要将新结点插入到最后一个结点之后，如图2-69所示。这时，新结点称为链表的最后一个结点。

在这种情况下，只需要一步操作，即将s指向的结点插入p指向的结点之后即可，即 $p \rightarrow next = s$ 。

```

LIST *InsertNode (LIST *h, LIST *s)
{
    LIST *pre, *p;                                /*
定义指针变量pre                                */
和p*/
    p=h;                                           /*
令p                                           */
指向第一个结点*/
    if (h==NULL)                                    /*
如果链表为空*/
    {
        h=s;                                       /*s
结点就是表的第一个结点，让头指针h
指向该结点*/
        s->next=NULL;                                /*
将s
指向的结点的指针域置为NULL*/
    }
    else                                           /*
如果链表不为空*/
    {
        while ( (s->no>p->no) && (p->next!=NULL) )    /*
在链表中查找要插入的位置*/
        {
            pre=p;                                   /*pre
指向p
指向的结点的前面的一个结点*/
            p=p->next;                                /*p
指向下一个结点*/
        }
        if (s->no<=p->no)                            /*
如果s
指向的结点的学号小于等于p
指向的学号*/
        {
            if (h==p)                                /*

```

```

如果要插入的位置在第一个结点之前*/
{
    h=s;                      /*
则让头指针h
指向新结点*/
    s->next=p;                /*
让新结点的指针域指向第一个结点*/
}
else                          /*
如果插入的位置在中间*/
{
    pre->next=s;              /*
将前一个结点的指针域指向新结点*/
    s->next=p;                /*
让新结点的指针域指向p
指向的结点*/
}
else                          /*
如果要插入的位置位于最后一个结点之后*/
{
    p->next=s;                /*
将最后一个结点即p
指向的结点的指针域指向新结点*/
    s->next=NULL;            /*
令s
指向的结点的指针域置为空*/
}
return h;                    /*
返回头指针*/
}

```

在链表中插入新结点时，需要说明以下几点。

- 在插入新结点时，需要考虑链表是否为空的情况。如果链表为空，则新结点直接作为链表的第一个结点，就是让头指针h指向该结点。代码如下。

```

if (h==NULL)                  /*
如果链表为空*/
{
    h=s;                      /*s
结点就是表的第一个结点，让头指针h
指向该结点*/
    s->next=NULL;              /*
将s
指向的结点的指针域置为NULL*/
}

```

- 如果链表不为空，则需要在链表中查找要插入的位置。在查找时，用到两个指针pre和p。其中，pre指向p指向的前一个结点。要插入的新结点位置应该为pre和p之间。查找插入位置的代码如下。

```

while ((s->no>p->no) && (p->next!=NULL)) /*
在链表中查找要插入的位置*/
{
    pre=p;                      /*pre
...

```

```
指向p
指向的结点的前面的一个结点*/
p=p->next; /*p
指向下一个结点*/
}
```

其中，条件有两个，即 $s \rightarrow no > p \rightarrow no$ 和 $p \rightarrow next \neq NULL$ ，前者是比较学号的大小，后者控制链表是否结束。因此，退出该循环有两个可能，一个是 $s \rightarrow no \leq p \rightarrow no$ ，一个是 $p \rightarrow next == NULL$ 。

- 如果 $s \rightarrow no \leq p \rightarrow no$ ，说明找到了要插入的位置，该位置存在两种可能，一个是插入的位置位于第一个结点之前；另一个是插入的位置在链表中间。对于第一种情况，只需要将新结点的指针域指向第一个结点，然后让头指针指向新结点，新结点成为链表的第一个结点。代码如下。

```
if (h==p) /*
如果要插入的位置在第一个结点之前*/
{
    h=s; /*
    则让头指针h
    指向新结点*/
    s->next=p; /*
    让新结点的指针域指向第一个结点*/
}
```

对于第二种情况，需要将新结点插入 pre 和 p 指向的结点之间，代码如下。

```
pre->next=s; /*
将前一个结点的指针域指向新结点*/
s->next=p; /*
让新结点的指针域指向p
指向的结点*/
```

- 如果 $p \rightarrow next == NULL$ ，则 p 指向的是最后一个结点。此时新结点的学号大于最后一个结点的学号，需要将新结点插入到最后一个结点之后，代码如下。

```
p->next=s; /*
将最后一个结点即p
指向的结点的指针域指向新结点*/
s->next=NULL; /*
令s
指向的结点的指针域置为空*/
```

2.6.6 链表的删除操作

链表的删除操作就是将链表中的某个结点删除。例如有6个小孩手拉手玩耍，有个小孩离开之后，剩余的5个小孩继续手拉手保持队形不变。这就类似于链表的生删除操作，其中小孩手拉手就像是一个链表，每个小孩像链表中的结点。

在链表中删除某个结点后，链表中剩下的结点次序保持不变，将剩下的结点相连接仍然是一个链表。下面通过具体的实例说明链表的删除操作。

如果要删除学号为10903的结点，需要先找到该结点，然后从链表中将该结点删除。删除该结点就是将该结点从链表中脱离出来，使该结点不再是链表的一部分。最后释放结点占用的内存空间。删除过程如图2-70所示。

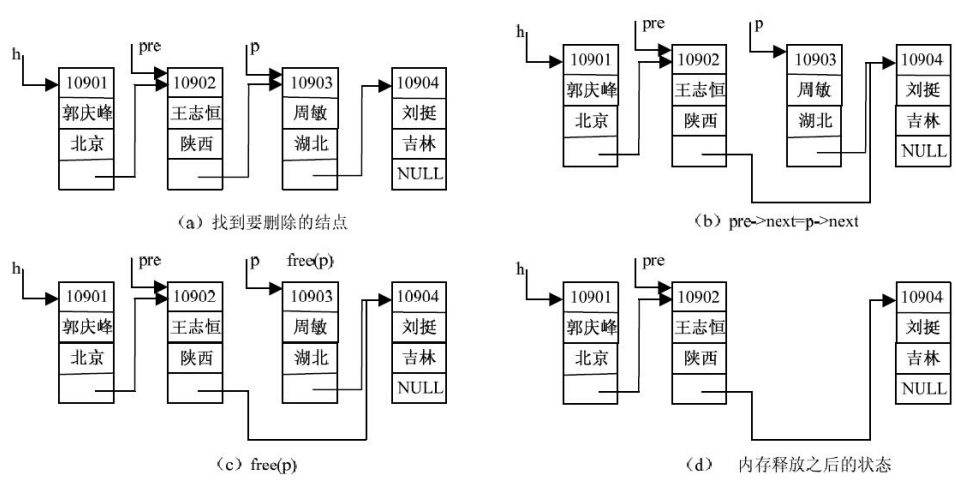


图2-70 删除学号为10903结点的过程

①删除一个结点，需要使用两个指针 pre 和 p ，其中 p 指向要删除的结点， pre 指向要删除结点的前一个结点，如图2-70 (a) 所示。

②将 p 指向的结点从链表中去除，也就是让 pre 指向的结点直接指向 p 指向的下一个结点，这样在顺着链表找到到第二个结点后，顺着第二个结点的指针域直接

找到了第四个结点，因为第二结点的指针域不再指向了第三个结点。要实现这个操作，需要将p指向的下一个结点的地址赋值给pre->next，即pre->next=p->next。如图2-63（b）所示。

③因为结点的内存空间是动态分配的，所以需要使用函数free将这些空间释放掉。如图2-63 (c) 所示。将结点释放掉之后，链表的状态就是图2-63 (d)。

删除操作的程序代码如下所示。

```

LIST *DeleteNode(LIST * h, long no)
{
    LIST *pre, *p; /*
    定义指针变量*/
    if (h==NULL) /*
    如果链表为空，则不能删除结点*/
    {
        printf("
        链表为空，不能删除结点!\n");
        return NULL; /*
    返回空指针*/
    }
    p=h; /*
    将p
    指向第一个结点*/
    while (p->no!=no&& p->next!=NULL) /*
    如果当前结点不是要删除的结点*/
    {
        pre=p; /*pre
        指向p
        指向的结点*/
        p=p->next; /*p
        指向下一个结点*/
    }
    if (p->no==no) /*
    如果p
    指向的结点是要删除的结点*/
    {
        if (p==h) /*
        如果要删除的结点是第一个结点*/
            h=p->next; /*
        则头指针指向第二个结点*/
        else /*
        如果要删除的结点不是第一个结点*/
            pre->next=p->next; /*
        则让pre
        的指针域指向p
        的下一个结点，即将p
        删除*/
        free(p); /*
        释放p
        指向的结点的内存空间*/
        printf("
        结点被成功删除.\n");
    }
    else /*
    如果没有找到要删除的结点*/
        printf("
        没有发现要删除的结点!\n"); /*
    输出提示信息*/
    return h; /*
    返回链表的头指针*/
}

```

在删除操作过程中，需要说明以下几点。

- 首先要判断链表是否为空，如果为空，则不能进行删除操作，代码如下。

```
if (h==NULL) /*
如果链表为空，则不能删除结点*/
{
    printf("
链表为空，不能删除结点!\n");
    return; /*
直接返回*/
}
```

判断链表是否为空的条件是`h==NULL`。如果为空，则将程序直接返回，不再执行下面的操作。

- 如果链表不为空，则需要从链表的第一个结点开始，即头指针`h`出发，依次比较结点的学号。先让指针`p`指向第一个结点，代码如下。

```
p=h; /*
将p
指向第一个结点*/
```

接着比较结点的学号即`p->no==no`。如果等于`no`，则需要继续比较，代码如下。

```
while (p->no!=no&& p->next!=NULL) /*
如果当前结点不是要删除的结点*/
{
    pre=p; /*pre
指向p
指向的结点*/
    p=p->next; /*p
指向下一个结点*/
}
```

- 如果找到链表中要删除的结点，则需要继续判断，看要删除的结点是第一个结点还是中间结点。如果要删除的结点是第一个结点，则需要将头指针`h`指向第二结点，然后释放`p`指向的结点内存空间，代码如下。
-

```
if (p==h) /*
如果要删除的结点是第一个结点*/
    h=p->next; /*
则头指针指向第二个结点*/
```

要删除的结点是第一个结点的情况如图2-71所示。

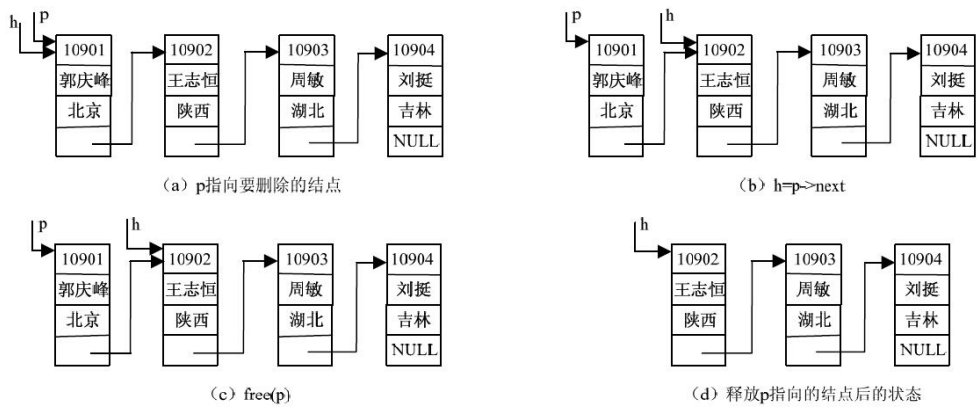


图2-71 删除第一个结点的过程

- 如果没有找到要删除的结点，也就是当 $p \rightarrow next \neq \text{NULL}$ 条件不成立时，输出提示信息。
- 在程序的最后要返回链表的头指针 h ，供调用函数或其他函数使用。

2.6.7 链表的综合操作

前面介绍了链表的各种操作，包括创建链表、输出链表、插入链表和删除链表，本节将通过main函数调用这些链表操作，测试程序的正确性。

【例2-18】 编写一个main函数，分别调用函数CreateList创建链表，调用InsertNode插入新结点，调用函数DeleteNode删除结点，调用函数DispList输出结点的值。

实现程序如下。

```

#include<stdio.h>
struct student                                /*
定义结点*/
{
    long no;                                /*
学号*/
    char name[20];                          /*
姓名*/
    char addr[30];                          /*
地址*/
    struct student *next;                  /*
指向结构体指针*/
};
typedef struct student LIST;                /*
重新定义结点类型*/
LIST *CreateList();                        /*
声明函数CreateList*/
LIST *InsertNode(LIST *h,LIST *s);         /*
声明函数InsertNode*/
LIST *DeleteNode(LIST * h,long no);        /*
声明函数DeleteNode*/
void DispList(LIST *h);                    /*
声明函数DispList*/
void main()
{
    LIST *head,*p;                          /*
定义指向结点的指针*/
    long no;                                /*
定义一个长整型,表示输入的学号*/
    head=CreateList();                      /*
调用函数CreateList
创建链表*/
    printf("
学号
姓名
地址\n");
    DispList(head);                          /*
调用函数DispList
输出链表中的结点*/
    printf("
输入要插入的结点元素: \n");
    p=(LIST*)malloc(sizeof(LIST));          /*
动态生成一个新结点*/
    scanf("%d %s %s",&p->no,p->name,p->addr); /*
输入结点的数据*/
    head=InsertNode(head,p);                /*
调用函数InsertNode
插入新结点*/
    printf("
学号
姓名
地址\n");
    DispList(head);                          /*
调用函数DispList
输出插入新结点后的链表*/
    printf("
请输入一个要删除结点的学号:\n ");
    scanf("%d",&no);                        /*
输入要删除结点的学号*/
    head=DeleteNode(head,no);               /*
调用函数DeleteNode
输出学号为no
的结点*/
    if(head!=NULL)
    {
        printf("
学号
姓名
地址\n");
        DispList(head);                      /*
调用函数DispList
输出删除结点后的链表*/
    }
}

```

程序运行结果如图2-72所示。

该程序省略了函数CreateList、InsertNode、DeleteNode、DispList的定义。这几个函数的具体实现请参见2.2.1~2.2.4小节。



```
G:\ "D:\数据结构\例2_19\Debug..."
输入结点个数:
4
10903 周敏 湖北
10907 吴峰 江苏
10912 卢春俊 陕西
10231 郑银林 上海
学号 姓名 地址
10903 周敏 湖北
10907 吴峰 江苏
10912 卢春俊 陕西
10231 郑银林 上海
输入要插入的结点元素:
10908 张林 广东
学号 姓名 地址
10903 周敏 湖北
10907 吴峰 江苏
10908 张林 广东
10912 卢春俊 陕西
10231 郑银林 上海
请输入一个要删除结点的学号:
10912
结点被成功删除.
学号 姓名 地址
10903 周敏 湖北
10907 吴峰 江苏
10908 张林 广东
10231 郑银林 上海
Press any key to continue
```

图2-72 程序运行结果

2.6.8 链表应用举例：一元多项式的相加

假设一元多项式为 $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ ，一元多项式的每一项由系数和指数构成，因此要表示一元多项式，需要定义一个结构体。结构体由两个部分构成，分别为coef和exp，分别表示系数和指数。定义结构体的代码如下。

```
struct node
{
    /*
    系数*/      float coef;          /*
    指数*/      int exp;             /*
};
```

如果用结构体数组表示多项式的每一项，则需要n+1个数组元素存放多项式（假设n为最高次数）。遇到指数不连续且指数之间跨越非常大时，例如，多项式 $2x^{500} + 1$ ，则需要数组的长度为501。这显然会浪费很多内存单元。

为了有效利用内存空间，可以使用链表表示多项式，多项式的每一项使用结点表示。结点由系数、指数和指针域3个部分构成，结构如图2-73所示。

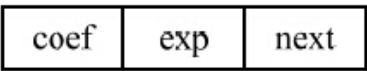


图2-73 多项式每一项的结点结构

结点用C语言描述如下。

```
struct node
{
    /*
    系数*/      float coef;          /*
    指数*/      int exp;             /*
    指针域*/    struct node *next;   /*
};
```

为了操作方便，将链表按照指数的从高到低进行排列，即降幂排列。一个最高次数为n的多项式构成的链表如图2-74所示。

例如，有两个一元多项式 $p(x) = 3x^2 + 2x + 1$ 和 $q(x) = 5x^3 + 3x + 2$ ，链表表示如图2-75所示。

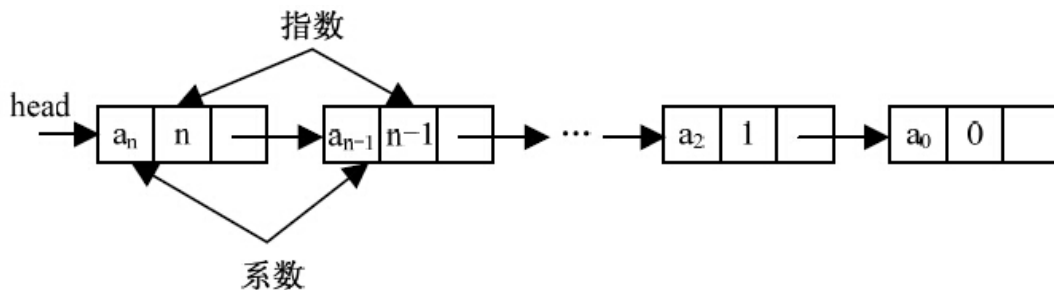


图2-74 一元多项式的链表结构

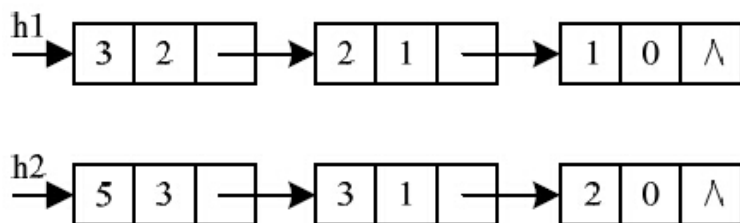


图2-75 一元多项式的链表表示

如果要将两个多项式相加，需要比较两个多项式的指数项后决定。当两个多项式的两项中指数相同时，才将系数相加。如果两个多项式的指数不相等，则多项式该项和的系数是其中一个多项式的系数。实现代码如下。

```

if (s1->exp==s2->exp)                                /*
如果两个指数相等，则将系数相加*/
{
    c=s1->coef+s2->coef;
    e=s1->exp;
    s1=s1->next;
    s2=s2->next;
}
else if (s1->exp>s2->exp)                             /*
如果s1
的指数大于s2
的指数，则将s1
的指数作为结果*/
{
    c=s1->coef;
    e=s1->exp;
    s1=s1->next;
}
else                                                    /*
如果s1
的指数小于等于s2
，则将s2
的指数作为结果*/
{
    c=s2->coef;
    e=s2->exp;
    s2=s2->next;
}

```

其中，s1和s2分别指向两个链表表示的表达式。因为表达式是按照指数从大到小排列的，所以在指数不等时，将指数大的作为结果。指数小的还要继续进行比较。例如，如果当前s1指向系数为3，指数为2的结点即（3，2），s2指向（3，1）的结点，因为s1->exp>s2->exp，所以将s1的结点作为结果。在s1指向（2，1）时，还要与s2的（3，1）相加，得到（5，1）。

如果相加后的系数不为0，则需要生成一个结点存放到链表中，代码如下。

```
if(c!=0)
{
    p=(ListNode*)malloc(sizeof(ListNode));
    p->coef=c;
    p->exp=e;
    p->next=NULL;
    if(s==NULL)
        s=p;
    else
        r->next=p;
    r=p;
}
```

如果在一个链表已经到达末尾，而另一个链表还有结点，需要将剩下的结点插入新链表中，代码如下。

```
while(s1!=NULL)
{
    c=s1->coef;
    e=s1->exp;
    s1=s1->next;
    if(c!=0)
    {
        p=(ListNode*)malloc(sizeof(ListNode));
        p->coef=c;
        p->exp=e;
        p->next=NULL;
        if(s==NULL)
            s=p;
        else
            r->next=p;
        r=p;
    }
}
while(s2!=NULL)
{
    c=s2->coef;
    e=s2->exp;
    s2=s2->next;
    if(c!=0)
    {
        p=(ListNode*)malloc(sizeof(ListNode));
        p->coef=c;
        p->exp=e;
        p->next=NULL;
        if(s==NULL)
            s=p;
```

```

        else
            r->next=p;
        r=p;
    }
}

```

最后，s指向的链表就是两个多项式的和。

【例2-19】 依次输入两个多项式，编写程序求两个多项式的和。

该程序实现可以分为创建多项式、输出多项式、求两个多项式的和几个部分。

①创建多项式。依次输入多项式的系数和指数，创建多项。当输入“0，0”即系数和指数都为0时，输入结束。创建多项式的代码如下。

```

#include<stdio.h>
#include<stdlib.h>
struct node
{
    int exp;
    float coef;
    struct node *next;
};
typedef struct node ListNode;
ListNode *createpoly()
/*
创建多项式链表*/
{
    ListNode *h=NULL, *p, *q=NULL;
    int e;
    float c;
    printf("
请输入系数和指数:");
    scanf("%f,%d",&c,&e);
    while(e!=0||c!=0)
    {
        p=(ListNode*)malloc(sizeof(ListNode));
        p->coef=c;
        p->exp=e;
        p->next=NULL;
        if(h==NULL)
            h=p;
        else
            q->next=p;
        q=p;
        printf("
请输入系数和指数:");
        scanf("%f,%d",&c,&e);
    }
    return h;
}

```

②输出多项式。为了避免在指数为0时输出系数，增加一个条件语句。如果指数为0，则只输出系数。输出链表的代码如下。

```
void disppoly(ListNode *h)
/*
输出多项式*/
{
    ListNode *p;
    p=h;
    while (p!=NULL)
    {
        if (p->exp==0)
            printf("%.2f",p->coef);
        else
            printf("%fx^%d",p->coef,p->exp);
        p=p->next;
        if (p!=NULL)
            printf("+");
    }
    printf("\n");
}
```

③求两个多项式的和，代码如下。

```
ListNode *addpoly(ListNode *h1,ListNode *h2)
/*
将两个多项式相加*/
{
    ListNode *p,*r=NULL,*s1,*s2,*s=NULL;
    float c;
    int e;
    s1=h1;
    s2=h2;
    while (s1!=NULL&& s2!=NULL)
    {
        if (s1->exp==s2->exp)
        {
            c=s1->coef+s2->coef;
            e=s1->exp;
            s1=s1->next;
            s2=s2->next;
        }
        else if (s1->exp>s2->exp)
        {
            c=s1->coef;
            e=s1->exp;
            s1=s1->next;
        }
        else
        {
            c=s2->coef;
            e=s2->exp;
            s2=s2->next;
        }
        if (c!=0)
        {
            p=(ListNode*)malloc(sizeof(ListNode));
            p->coef=c;
            p->exp=e;
            p->next=NULL;
            if (s==NULL)
                s=p;
            else
                r->next=p;
            r=p;
        }
    }
}
```

```

    }
    while (s1!=NULL)
    {
        c=s1->coef;
        e=s1->exp;
        s1=s1->next;
        if (c!=0)
        {
            p=(ListNode*)malloc(sizeof(ListNode));
            p->coef=c;
            p->exp=e;
            p->next=NULL;
            if (s==NULL)
                s=p;
            else
                r->next=p;
            r=p;
        }
    }
    while (s2!=NULL)
    {
        c=s2->coef;
        e=s2->exp;
        s2=s2->next;
        if (c!=0)
        {
            p=(ListNode*)malloc(sizeof(ListNode));
            p->coef=c;
            p->exp=e;
            p->next=NULL;
            if (s==NULL)
                s=p;
            else
                r->next=p;
            r=p;
        }
    }
    return s;
}

```

④释放链表所占用的内存空间。在使用完链表后，需要释放链表所占用的内存空间，代码如下。

```

void deletepoly(ListNode *h)
/*
释放多项式所占用内存单元*/
{
    ListNode *p,*r=h;
    while (r!=NULL)
    {
        p=r->next;
        free(r);
        r=p;
    }
}

```

⑤测试，代码如下。

```

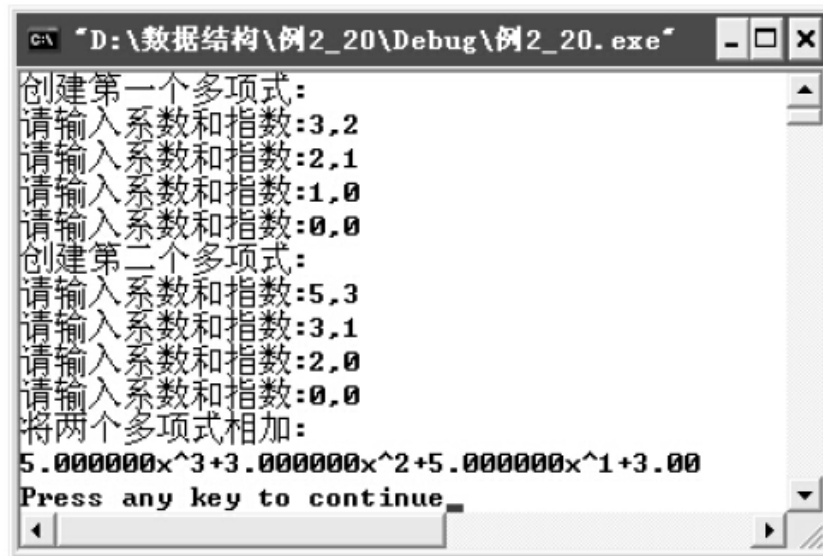
void main()
{
    ListNode *head1,*head2,*head;
    printf("
创建第一个多项式:\n");
    head1=createpoly();
}

```



```
        printf("
创建第二个多项式:\n");
        head2=createpoly();
        printf("
将两个多项式相加:\n");
        head=addpoly(head1,head2);
        disppoly(head);
        deletepoly(head);
    }
```

程序运行结果如图2-76所示。



```
C:\ "D:\数据结构\例2_20\Debug\例2_20.exe"
创建第一个多项式:
请输入系数和指数:3,2
请输入系数和指数:2,1
请输入系数和指数:1,0
请输入系数和指数:0,0
创建第二个多项式:
请输入系数和指数:5,3
请输入系数和指数:3,1
请输入系数和指数:2,0
请输入系数和指数:0,0
将两个多项式相加:
5.000000x^3+3.000000x^2+5.000000x^1+3.00
Press any key to continue
```

图2-76 程序运行结果

2.7 小结

本章主要介绍了C语言的重点和难点部分，目的是为了今后学习数据结构扫清障碍。首先对常用的C语言开发环境Turbo C 2.0和VC++6.0做了介绍，接着围绕着C语言中的重点和难点——递归、指针、参数传递、结构体、链表，结合典型案例进行了详细分析、讲解。

递归是C语言及算法设计中常常使用的技术，递归可以把复杂的问题变成与原问题类似且规模小的问题加以解决，使用递归使程序的结构很清晰，更具有层次性，写出的程序简洁易懂。使用递归只需要少量的程序就可以描述解决问题需要的重复计算过程，大大减少了程序的代码量。任何使用递归解决的问题都能使用迭代的方法解决。

指针是C语言的精髓所在。指针不仅可以与变量结合起来使用，还可以与数组、函数相结合，使用指针能很方便地操作字符串、动态分配内存。指针使用不当，也常常出现一些致命错误，这种错误十分隐蔽，难以发现，这就需要读者熟练使用指针，以避免或减少错误的发生。

在C语言中，函数的参数传递有两种：传值调用和传地址调用。其中，前者是一种单向值传递方式，实际参数和形式参数分别占用不同

的内存空间。后者是一种双向的值传递方式，实际参数和形式参数占用同一块内存单元。

结构体属于用户自己定义的类型，它常常用于非数值程序设计中，特别是在今后学习数据结构的过程中，链表、栈、队列、树及图等都会利用结构体类型。

2.8 习题

一、选择题

1. 设有如下定义。

```
int arr[]={6,7,8,9,10};  
int * ptr;
```

则下列程序段的输出结果为_____。

```
ptr=arr;  
* (ptr+2)+=2;  
printf ("%d,%d\n",*ptr,* (ptr+2));
```

A. 8, 10

B. 6, 8

C. 7, 9

D. 6, 10

2. 以下程序的输出结果是_____。

```
main()  
{ int i,k,a[10],p[3];  
  k=5;  
  for (i=0;i<10;i++) a[i]=i;  
  for (i=0;i<3;i++) p[i]=a[i*(i+1)];  
  for (i=0;i<3;i++) k+=p[i]*2;
```

```
printf("%d\n",k);  
}
```

A. 20

B. 21

C. 22

D. 23

3. 执行以下程序段后，m的值为_____。

```
int a[2][3]={ {1,2,3},{4,5,6} };  
int m,*p;  
p=&a[0][0];  
m=(*p)*(* (p+2))*(* (p+4));
```

A. 15

B. 14

C. 13

D. 12

4. 有以下程序。

```
main()  
{ char a[]="programming", b[]="language";  
  char *p1,*p2;  
  int i;  
  p1=a; p2=b;  
  for(i=0;i<7;i++)
```

```
        if (*(p1+i)==*(p2+i)) printf("%c",*(p1+i));  
    }  
}
```

输出结果是_____。

A. gm

B. rg

C. or

D. ga

5. 有以下程序。

```
void f(int *x,int *y)  
{ int t;          t=*x;*x=*y;*y=t; }  
main()  
{ int a[8]={1,2,3,4,5,6,7,8},i,*p,*q;  
  p=a;q=&a[7];  
  while(p) {f(p,q);p++;q--;}  
  for(i=0;i<8;i++)printf("%d",a[i]);  
}
```

程序运行后的输出结果是_____。

A. 8, 2, 3, 4, 5, 6, 7, 1

B. 5, 6, 7, 8, 1, 2, 3, 4

C. 1, 2, 3, 4, 5, 6, 7, 8

D. 8, 7, 6, 5, 4, 3, 2, 1

6. 有以下程序。

```
#include <STDIO.H>
struct stu
{   int num;
    char name[10];
    int age;
};
void fun(struct stu *p)
{   printf("%s\n", (*p).name);   }
main()
{struct stu students[3]={ {9801,"Zhang",20}, {9802,"Wang",19}, {9803,"Zhao",18} };
  fun(students+2);
}
```

输出结果是_____。

A. Zhang

B. Zhao

C. Wang

D. 18

7. 设有以下说明和定义。

```
typedef union
{   long i; int k[5]; char c; }DATE;
struct date
{   int cat; DATE cow; double dog; } too;
DATE max;
```

则下列语句的执行结果是_____。

```
printf ("%d",sizeof (struct date ) +sizeof(max));
```

A. 26

B. 30

C. 18

D. 8

8. 以下选项中，能定义s为合法的结构体变量的是_____。

A. typedef struct abc

B. struct

```
{ double a
;
;
char b[10]
;
} s
;
;

{ double a
;
;
char b[10]
;
} s
;
;
```

C. struct ABC

D. typedef ABC

```
{ double a
;
;
char b[10]
;

{ double a
;
;
char b[10]
;
```

```
;  
}  
ABC s  
;  
ABC s  
;
```

二、填空题

1. 有以下定义和语句，则sizeof (a) 的值是____，而sizeof (a.share) 的值是____。

```
struct date  
{    int day;  
  int month;  
  int year;  
  union{  int share1    float share2;    }share;  
}a;
```

2. 若要使指针p指向一个double类型的动态存储单元，请填空。

p=

malloc(sizeof(double));

3. 下面程序的输出结果是____。

```
char b[]="abcd";  
main()  
{  char *chp;  
  for(chp=b; *chp: chp+=2) printf("%s",chp);  
  printf("\n");  
}
```

4. 以下程序的功能是:将无符号八进制数字构成的字符串转换为十进制整数。例如输入的字符串为556, 则输出十进制整数366。请填空。

```
#include<stdio.h>
main()
{  char *p, s[6];      int n;      p=s;      gets(p);      n=*p-'0';
  while(
    !='\0')  n=n*8+*p-'0';
  printf("%d \n",n);
}
}
```

三、编程题

1. 使用指针实现将数组中的元素按照逆序存放。
2. 编写一个求字符串子串的函数substr (char*str1, char str2[], int s, int m) , 要求将字符串str1从位置s开始的m个字符复制到字符串数组str2中。
3. 递归实现函数, 将一个整数转化为十进制数字输出。
4. 约瑟夫环问题。13个人围成一圈, 从第1个人开始报数, 报到3退出圈子, 按顺序输出退出圈子的序号。
5. 一头刚出生的小牛, 4年后生一头小牛, 以后每年生一头。现有一头刚出生的小牛, 问20年后共有牛多少头?

6. 递归实现将一个字符串颠倒后重新存放在原字符串中。

第二篇 线性数据结构

第3章 线性表

线性表是一种最简单的线性结构。线性结构的特点是在非空的有限集合中存在唯一的一个被称为“第一个”的数据元素，存在唯一的一个被称为“最后一个”的数据元素。第一个元素没有直接前驱元素，最后一个元素没有直接后继元素，其他元素都有唯一的前驱元素和唯一的后继元素。线性表有两种存储结构，即顺序存储结构和链式存储结构。本章主要介绍线性表的定义及运算、线性表的顺序存储、线性表的链式存储、循环链表、双向链表及链表的运用。

本章重点和难点：

- 顺序表和单链表的基本操作实现
- 静态链表的存储表示与基本操作实现

3.1 线性表的定义及抽象数据类型

线性表 (Linear_List) 是最简单且最常用的一种线性结构。本节主要介绍线性表的逻辑结构及在线性表上的运算。

3.1.1 线性表的逻辑结构

线性表 是由 n 个类型相同的数据元素组成的有限序列，记为 $(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 。其中，这里的数据元素可以是原子类型，也可以是结构类型。线性表的数据元素存在着序偶关系，即数据元素之间具有一定的次序。在线性表中，数据元素 a_{i-1} 在 a_i 的前面， a_i 又在 a_{i+1} 的前面，可以把 a_{i-1} 称为 a_i 的直接前驱元素， a_i 称为 a_{i+1} 的直接前驱元素。 a_i 称为 a_{i-1} 的直接后继元素， a_{i+1} 称为 a_i 的直接后继元素。

线性表的逻辑结构如图3-1所示。

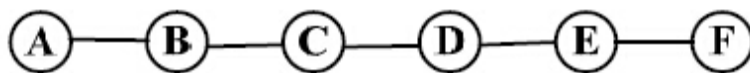


图3-1 线性表的逻辑结构

我们学过的英文单词就是简单的线性表，例如“China”、“Science”、“Structure”。其中每一个英文字母就是一个数据元

素，每个数据元素之间存在着唯一的顺序关系。如“China”中字母C后面是字母h，字母h后面是字母i。

在较为复杂的线性表中，一个数据元素可以由若干个数据项组成，在表3-1所示的一所学校的教职工情况表中，一个数据元素由姓名、性别、出生年月、籍贯、学历、职称及任职时间7个数据项组成。数据元素也称为记录。

表3-1 教职工情况表

姓 名	性 别	出 生 年 月	籍 贯	学 历	职 称	任 职 时 间
王 欢	女	1958 年 10 月	河南	本科	教授	2000 年 10 月
康全宝	男	1969 年 5 月	陕西	研究生	副教授	2002 年 10 月
冯全义	女	1978 年 12 月	四川	研究生	讲师	2006 年 11 月
⋮	⋮	⋮	⋮	⋮	⋮	⋮

知识点 在线性表中，除了第一个元素 a_1 ，每个元素有且仅有一个直接前驱元素；除了最后一个元素 a_n ，每个元素有且只有一个直接后继元素。

3.1.2 线性表的抽象数据类型

线性表的抽象数据类型包括数据对象集合和基本操作集合。

1. 数据对象集合

线性表的数据对象集合为 $\{a_1, a_2, \dots, a_n\}$ ，元素类型为DataType。

数据元素之间的关系是一对一的关系。除了第一个元素 a_1 外，每个元素有且只有一个直接前驱元素，除了最后一个元素 a_n 外，每个元素有且只有一个直接后继元素。

2. 基本操作集合

(1) InitList (&L)：初始化操作，建立一个空的线性表L。这就像是在日常生活中，一所院校为了方便管理，建立一个教职工基本情况表，准备登记教职工信息。

(2) ListEmpty (L)：若线性表L为空，返回1，否则返回0。这就像是刚刚建立了教职工基本情况表，还没有登记教职工信息。

(3) GetElem (L, i, &e)：返回线性表L的第i个位置元素值给e。这就像在教职工基本情况表中，根据给定序号查找某个教师信息。

(4) LocateElem (L, e)：在线性表L中查找与给定值e相等的元素，如果查找成功返回该元素在表中的序号表示成功，否则返回0表示失败。这就像在教职工基本情况表中，根据给定的姓名查找教师信息。

(5) InsertList (&L, i, e) : 在线性表L中的第i个位置插入新元素e。这就类似于经过招聘考试,引进了一名教师,这个教师信息登记到教职工基本情况表中。

(6) DeleteList (&L, i, &e) : 删除线性表L中的第i个位置元素,并用e返回其值。这就像某个教职工到了退休年龄或者调入其他学校,需要将该教职工从教职工基本情况表中删除。

(7) ListLength (L) : 返回线性表L的元素个数。这就像查看教职工基本情况表中有多少个教职工。

(8) ClearList (&L) : 将线性表L清空。这就像学校被撤销,不需要再保留教职工基本信息,将这些教职工信息全部清空。

3.2 线性表的顺序表示与实现

在了解了线性表的基本概念和逻辑结构之后，接下来就需要将线性表的逻辑结构转化为计算机能识别的存储结构，以便实现线性表的操作。线性表的存储结构主要有顺序存储结构和链式存储结构两种。本节的主要介绍线性表的顺序存储结构及顺序存储结构下的操作实现。

3.2.1 线性表的顺序存储结构

线性表的顺序存储指的是将线性表中的各个元素依次存放在一组地址连续的存储单元中。

假设线性表的每个元素需占用 m 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 $i+1$ 个元素的存储位置 $LOC(a_{i+1})$ 和第 i 个元素的存储位置 $LOC(a_i)$ 之间满足关系 $LOC(a_{i+1}) = LOC(a_i) + m$ 。

线性表中第 i 个元素的存储位置与第一个元素 a_1 的存储位置满足以下关系。

$$LOC(a_i) = LOC(a_1) + (i-1) * m$$

其中，第一个元素的位置 $LOC(a_1)$ 称为起始地址或基地址。

线性表的这种机内表示称为线性表的顺序存储结构或顺序映像 (sequential mapping)，通常将这种方法存储的线性表称为**顺序表**。顺序表

逻辑上相邻的元素在物理上也是相邻的。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3-2）。只要确定了第一个元素的起始位置，线性表中的任一元素都可以随机存取，因此，线性表的顺序存储结构是一种随机存取的存储结构。

存储地址	内存状态	元素在线性表中的顺序
addr	a₁	1
addr+m	a₂	2
	⋮	⋮
addr+(i-1)*m	a_i	i
	⋮	⋮
addr+(n-1)*m	a_n	n
	⋮	⋮

图3-2 线性表存储结构

由于C语言的数组具有随机存取特点，因此可采用数组来描述顺序表。顺序表的存储结构描述如下。

```
#define ListSize 100
typedef struct
{
    DataType list[ListSize];
    int length;
}SeqList;
```

其中，DataType表示数据元素类型，list用于存储线性表中的数据元素，length用来表示线性表中数据元素的个数，SeqList是结构体类型名。

如果要定义一个顺序表，代码如下。

```
SeqList L;
```

如果要定义一个指向顺序表的指针，代码如下。

```
SeqList *L;
```

3.2.2 顺序表的基本运算

在顺序存储结构中，线性表的基本运算如下（以下算法的实现保存在文件“SeqList.h”中）。

①初始化线性表。

```
void InitList(SeqList *L)
/*
初始化线性表*/
{
    L->length=0;    /*
把线性表的长度置为0*/
}
```

②判断线性表是否为空。

```
int ListEmpty(SeqList L)
/*
判断线性表是否为空，线性表为空返回1
， 否则返回0*/
{
    if(L.length==0)
        return 1;
    else
        return 0;
}
```

③按序号查找。先判断序号是否合法，如果合法，把对应位置的元素赋给e，并返回1表示查找成功；否则返回-1表示查找失败。按序号查找的算法实现如下。

```
int GetElem(SeqList L,int i,DataType *e)
/*
查找线性表中第i
个元素。查找成功将该值返回给e
，并返回1
表示成功；否则返回-1
表示失败。*/
{
    if(i<1||i>L.length)                /*
在查找第i
个元素之前，判断该序号是否合法*/
        return -1;
    *e=L.list[i-1];                      /*
将第i
个元素的值赋给e*/
    return 1;
}
```

④按内容查找。从线性表中的第一个元素开始，依次与e比较，如果相等，返回该序号表示成功；否则返回0表示查找失败。按内容查找的算法实现如下。

```
int LocateElem(SeqList L,DataType e)
/*
查找线性表中元素值为e
的元素*/
{
    int i;
    for(i=0;i<L.length;i++)              /*
从第一个元素开始与e
进行比较*/
        if(L.list[i]==e)                  /*
若存在与e
值相等的元素*/
            return i+1;                  /*
返回该元素在线性表中的序号*/
    return 0;                             /*
否则，返回0*/
}
```

⑤插入操作。插入操作就是在线性表L中的第i个位置插入新元素e，使线性表 $\{a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n\}$ 变为 $\{a_1, a_2, \dots, a_{i-1}, e, a_i, \dots, a_n\}$ ，线性表的长度也由n变成n+1。

要在顺序表中的第*i*个位置上插入元素*e*，首先将第*i*个位置以后的元素依次向后移动1个位置，然后把元素*e*插入第*i*个位置。移动元素时要从后往前移动元素，先移动最后1个元素，再移动倒数第2个元素，依次类推。

例如，在线性表{9，12，6，15，20，10，4，22}中，要在第5个元素之前插入1个元素28，需要将序号为8、7、6、5的元素依次向后移动1个位置，然后在第5号位置插入元素28，这样，线性表就变成了{9，12，6，15，28，20，10，4，22}，如图3-3所示。

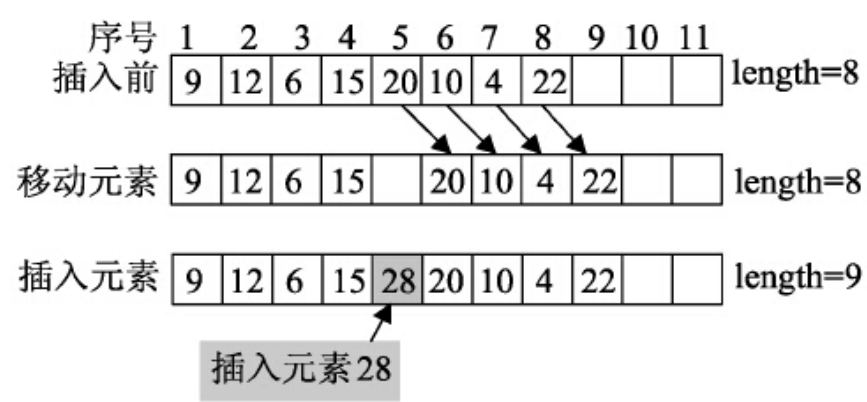


图3-3 在顺序表中插入元素28的过程

插入元素之前要判断插入的位置是否合法，顺序表是否已满，在插入元素后要将表长增加1。插入元素的算法实现如下。

```
int InsertList(SeqList *L,int i,DataType e)
/*
在顺序表的第i
个位置插入元素e
，插入成功返回1
，如果插入位置不合法返回-1
，顺序表满返回0*/
{
    int j;
    if(i<1||i>L->length+1)
    在插入元素前，判断插入位置是否合法*/
    {
        printf("
插入位置i
不合法! \n");
        return -1;
    }
    else if(L->length>=ListSize)
    ... ..
```

```

在插入元素前，判断顺序表是否已经满，不能插入元素*/
{
    printf("
顺序表已满，不能插入元素。\\n");
    return 0;
}
else
{
    for(j=L->length;j>=i;j--)          /*
将第i
个位置以后的元素依次后移*/
        L->list[j]=L->list[j-1];
    L->list[i-1]=e;                      /*
插入元素到第i
个位置*/
    L->length=L->length+1;              /*
将顺序表长增1*/
    return 1;
}
}

```

插入元素的位置 i 的合法范围应该是 $1 \leq i \leq L \rightarrow \text{length} + 1$ 。当 $i=1$ 时，插入位置是在第一个元素之前，对应C语言数组中的第0个元素；当 $i=L \rightarrow \text{length} + 1$ 时，插入位置是最后一个元素之后，对应C语言数组中的最后一个元素之后的位置。当插入位置是 $i=L \rightarrow \text{length} + 1$ 时，不需要移动元素；当插入位置是 $i=0$ 时，则需要移动所有元素。

⑥删除第 i 个元素。删除第 i 个元素之后，线性表 $\{a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n\}$ 变为 $\{a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n\}$ ，线性表的长度由 n 变成 $n-1$ 。

为了删除第 i 个元素，需要将第 $i+1$ 后面的元素依次向前移动一个位置，将前面的元素覆盖。移动元素时要先将第 $i+1$ 个元素移动到第 i 个位置，再将第 $i+2$ 个元素移动到第 $i+1$ 个位置，依次类推，直到最后一个元素移动到倒数第二个位置。最后将顺序表的长度减1。

例如要删除线性表 $\{9, 12, 6, 15, 28, 20, 10, 4, 22\}$ 的第4个元素，需要依次将序号为5、6、7、8、9的元素向前移动一个位置，并将表长减1，如图3-4所示。

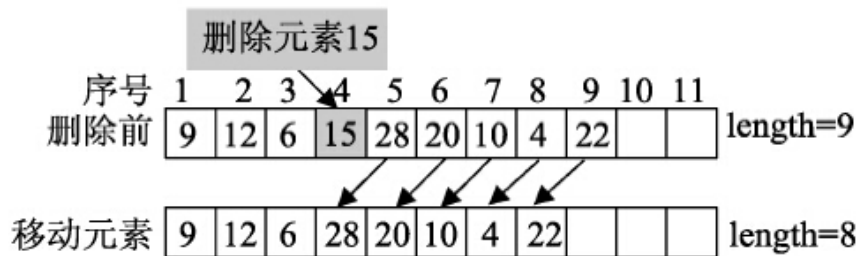


图3-4 删除元素15的过程

在进行删除操作时，先判断顺序表是否为空，若不空，接着判断序号是否合法，若不空且合法，则将要删除的元素赋给e，并把该元素删除，将表长减1。删除第i个元素的算法实现如下。

```

int DeleteList(SeqList *L,int i,DataType *e)
{
    int j;
    if (L->length<=0)
    {
        printf("
顺序表已空不能进行删除!\n");
        return 0;
    }
    else if (i<1||i>L->length)
    {
        printf("
删除位置不合适!\n");
        return -1;
    }
    else
    {
        *e=L->list[i-1];
        for (j=i;j<=L->length-1;j++)
            L->list[j-1]=L->list[j];
        L->length=L->length-1;
        return 1;
    }
}

```

删除元素的位置i的合法范围应该是 $1 \leq i \leq L \rightarrow \text{length}$ 。当i=1时，表示要删除第一个元素，对应C语言数组中的第0个元素；当i=L->length时，要删除的是最后一个元素。

⑦求线性表的长度，代码如下。

```

int ListLength(SeqList L)
{

```

```
    return L.length;
}
```

⑧清空顺序表，代码如下。

```
void ClearList(SeqList *L)
{
    L->length=0;
}
```

3.2.3 顺序表的实现算法分析

在顺序表的实现算法中，除了按内容查找运算、插入和删除操作外，算法的时间复杂度均为 $O(1)$ 。

在按内容查找的算法中，若要查找的是第一个元素，则仅需要进行一次比较；若要查找的是最后一个元素，则需要比较 n 次才能找到该元素（设线性表的长度为 n ）。

设 P_i 表示在第 i 个位置上找到与 e 相等的元素的概率，若在任何位置上找到元素的概率相等，即 $P_i = 1/n$ ，则查找元素需要的平均比较次数为

$E_{loc} = \sum_{i=1}^n p_i * i = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}$ ，因此，按内容查找的平均时间复杂度为 $O(n)$ 。

在顺序表中插入元素时，时间主要耗费在元素的移动上。若插入的位置在第一个位置，则需要移动元素的次数为 n 次；如果要将元素插入倒数第二个位置，则仅需把最后一个元素向后移动；若要将元素插入最后一个位置，即第 $n+1$ 个位置，则不需要移动元素。设 P_i 表示在第 i 个位置上插入元素的概率，假设在任何位置上找到元素的概率相等，即 $P_i = 1/(n+1)$ 。则在顺序表的第 i 个位

置插入元素时，需要移动元素的平均次数为 $E_{\text{ins}} = \sum_{i=1}^{n+1} p_i * (n-i+1) = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{n}{2}$ ，因此，插入操作的平均时间复杂度为 $O(n)$ 。

在顺序表的删除算法中，时间主要耗费仍在元素的移动上。如果要删除的是第一个元素，则需要移动元素次数为 $n-1$ 次；如果要删除的是最后一个元素，则需要移动 0 次。设 P_i 表示删除第 i 个位置上的元素的概率，假设在任何位置上找到元素的概率相等，即 $P_i = 1/n$ 。则在顺序表中删除第 i 个元素时，需要移动元素的平均次数为 $E_{\text{del}} = \sum_{i=1}^n p_i * (n-i) = \frac{1}{n} \sum_{i=1}^n (n-i) = \frac{n-1}{2}$ ，因此，删除操作的平均时间复杂度为 $O(n)$ 。

3.2.4 顺序表的优缺点

线性表的顺序存储结构的优缺点如下。

1. 优点

- (1) 无须为表示表中元素之间的关系而增加额外的存储空间。
- (2) 可以快速地存取表中任一位置的元素。

2. 缺点

- (1) 插入和删除操作需要移动大量的元素。
- (2) 使用前须事先分配好存储空间，当线性表长度变化较大时，难以确定存储空间的容量。分配空间过大会造成存储空间的巨大浪费，分配的空间过

小，难以适应问题的需要。

3.2.5 顺序表应用举例

在掌握了顺序表的基本操作之后，本小节通过几个具体实例来加强对顺序表的知识点掌握。

【例3-1】 假设线性表LA和LB分别表示两个集合A和B，利用线性表的基本运算实现新的集合 $A=A \cup B$ ，即扩大线性表LA，将存在于线性表B中且不存在于A中的元素插入A中。

【分析】 只有依次从线性表LB中取出每个数据元素，并依次在线性表LA中查找该元素，如果LA中不存在该元素，则将该元素插入LA中。程序的实现代码如下所示。

```
#include<stdio.h>                                /*
包含输入输出头文件*/
#define ListSize 100
typedef int DataType;                             /*
定义元素类型为整型*/
#include"SeqList.h"
void UnionAB(SeqList *A,SeqList B);               /*
将LB
中但不在LA
中的元素插入到LA
中*/
void main()
{
    int i,flag;
    DataType e;
    DataType a[]={2,3,17,20,9,31};
    DataType b[]={8,31,5,17,22,9,48,67};
    SeqList LA,LB;                                /*
声明顺序表LA
和LB*/
    InitList(&LA);                                  /*
初始化顺序表LA*/
    InitList(&LB);                                  /*
初始化顺序表LB*/
    for(i=0;i<sizeof(a)/sizeof(a[0]);i++)          /*
将数组a
中的元素插入到表LA
中*/
    {
        if(InsertList(&LA,i+1,a[i])==0)
        {
            printf("
位置不合法");
```

```

        return;
    }
    }
    for(i=0;i<sizeof(b)/sizeof(b[0]);i++) /*
将数组a
中的元素插入到表LB
中*/
    {
        if(InsertList(&LB,i+1,b[i])==0)
        {
            printf("
位置不合法");
            return;
        }
    }
    printf("
顺序表LA
中的元素: \n");
    for(i=1;i<=LA.length;i++) /*
输出顺序表LA
中的每个元素*/
    {
        flag=GetElem(LA,i,&e); /*
返回顺序表LA
中的每个元素到e
中*/
        if(flag==1)
            printf("%4d",e);
    }
    printf("\n");
    printf("
顺序表LB
中的元素: \n");
    for(i=1;i<=LB.length;i++) /*
输出顺序表LB
中的每个元素*/
    {
        flag=GetElem(LB,i,&e); /*
返回顺序表LB
中的每个元素到e
中*/
        if(flag==1)
            printf("%4d",e);
    }
    printf("\n");
    printf("
将LB
中但不在LA
中的元素插入到LA
中: \n");
    UnionAB(&LA,LB); /*
将LB
中但不在LA
中的元素插入到LA
中*/
    for(i=1;i<=LA.length;i++) /*
输出LA
中所有元素*/
    {
        flag=GetElem(LA,i,&e);
        if(flag==1)
            printf("%4d",e);
    }
    printf("\n");
}
void UnionAB(SeqList *LA,SeqList LB)
/*
删除A
中出现B
的元素的函数实现*/
{
    int i,flag,pos;
    DataType e;
    for(i=1;i<=LB.length;i++)
    {

```

```

        flag=GetElem(LB,i,&e);                                /*
依次把LB
中每个元素取出赋给e*/
        if(flag==1)
        {
            pos=LocateElem(*LA,e);                            /*
在LA
中查找和LB
中取出的元素e
相等的元素*/
            if(!pos)
                InsertList(LA,LA->length+1,e);                /*
如果找到该元素，将元素插入到LA
中*/
        }
    }
}

```

程序运行结果如图3-5所示。

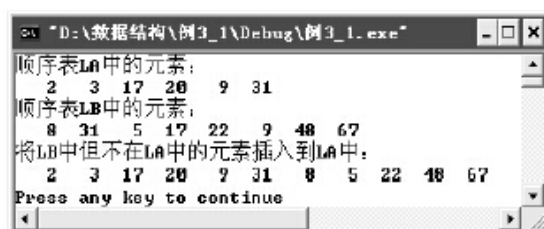


图3-5 顺序表 $A \cup B$ 程序运行结果

说明 在设计程序时需要用到头文件“SeqList.h”，而在顺序表的类型定义中包含DataType数据类型和顺序表长度，所以在包含#include“SeqList.h”之前首先进行宏定义。宏定义、类型定义和包含文件语句的次序如下。

```

#define ListSize 100
typedef int DataType;
#include"SeqList.h"

```

【例3-2】 编写一个算法，把一个顺序表分拆成两个部分，使顺序表中小于等于0的元素位于左端，大于0的元素位于右端。要求不占用额外的存储空间。例如顺序表（-12，3，-6，-10，20，-7，9，-20）经过分拆调整后变为（-12，-20，-6，-10，-7，20，9，3）。

【分析】 设置两个指示器i和j，分别扫描顺序表中的元素，i和j分别从顺序表的左端和右端开始扫描。如果i遇到小于等于0的元素，略过不处理，继续向前扫描；如果遇到大于0的元素，暂停扫描。如果j遇到大于0的元素，略过不处理，继续向前扫描；如果遇到小于等于0的元素，暂停扫描。如果i和j都停下来，则交换i和j指向的元素。重复执行直到 $i \geq j$ 为止。

算法描述如下。

```
void SplitSeqList(SeqList *L)
/*
将顺序表L
分成两个部分：左边是小于等于0
的元素，右边是大于0
的元素*/
{
    int i,j;                                /*
    定义两个指示器i
    和j*/
    DataType e;
    i=0,j=(*L).length-1;                    /*
    指示器i
    和j
    分别指示顺序表的左端和右端元素*/
    while(i<j)
    {
        while(L->list[i]<=0)                /*i
        遇到小于等于0
        的元素*/
            i++;                            /*
        略过*/
        while(L->list[j]>0)                  /*j
        遇到大于0
        的元素*/
            j--;                            /*
        略过*/
        if(i<j)                             /*
        交换i
        和j
        指向的元素*/
        {
            e=L->list[i];
            L->list[i]=L->list[j];
            L->list[j]=e;
        }
    }
}
```

测试程序如下。

```
#include<stdio.h>
#include"SeqList.h"
void SplitSeqList(SeqList *L);
void main()
{
    int i,flag,n;
```

```

        DataType e;
        SeqList L;
        int a[]={-12,3,-6,-10,20,-7,9,-20};
        InitList(&L);
/*
初始化顺序表L*/
        n=sizeof(a)/sizeof(a[0]);
        for(i=1;i<=n;i++)
/*
将数组a
的元素插入到顺序表L
中*/
        {
            if (InsertList(&L,i,a[i-1])==0)
            {
                printf("
位置不合法");
                return;
            }
        }
        printf("
顺序表L
中的元素: \n");
        for(i=1;i<=L.length;i++)
/*
输出顺序表L
中的每个元素*/
        {
            flag=GetElem(L,i,&e);
/*
返回顺序表L
中的每个元素到e
中*/
            if (flag==1)
                printf("%4d",e);
        }
        printf("\n");
        printf("
顺序表L
调整后(
左边元素<=0,
右边元素>0):\n");
        SplitSeqList(&L);
/*
调整顺序表*/
        for(i=1;i<=L.length;i++)
/*
输出调整后顺序表L
中所有元素*/
        {
            flag=GetElem(L,i,&e);
            if (flag==1)
                printf("%4d",e);
        }
        printf("\n");
    }
}

```

程序运行结果如图3-6所示。

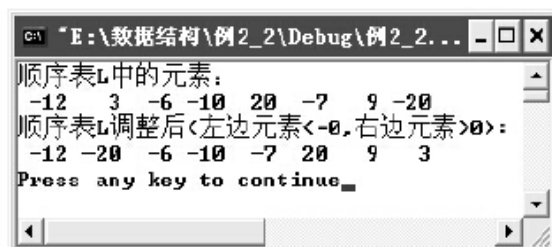


图3-6 程序运行结果

【例3-3】 试设计一种表示任意长的整数的数据结构，计算任意给定的两个整数之和的算法。

【分析】 C语言提供的整数范围为 $-2^{31} \sim 2^{31}-1$ ，超出这个范围的整数该如何表示呢？可以利用数组来存储，数组中的每一个元素存放一个数字，数组A和B分别存储两个整数，在将两个整数相加时，从低位到高位依次将对位相加，如果和大于9，则将高位上加上进位1，并将和减去10后存储到当前位。具体实现代码如下。

```
#include<stdio.h>
#define MaxLen 100
typedef int sqliist[MaxLen];
int input(sqliist A)
{
    int i;
    for(i=0;i<MaxLen;i++)
        A[i]=0;
    printf("
输入一个正整数的各位(
输入-1
结束)\n");
    i=0;
    while(1)
    {
        scanf("%d",&A[i]);
        if(A[i]<0)
            break;
        i++;
    }
    return i;
}
void output(sqliist A,int low,int high)
{
    int i;
    for(i=low;i<high;i++)
        printf("%d",A[i]);
    printf("\n");
}
void move(sqliist A,int na)
{
    int i;
    for(i=0;i<na;i++)
        A[MaxLen-i-1]=A[na-i-1];
}
int add(sqliist *A,int na,sqliist B,int nb)
{
    int nc,i,j,length=0;
    if(na>nb)
        nc=na;
    else
        nc=nb;
    move(*A,na);
    move(B,nb);
    for(i=MaxLen-1;i>=MaxLen-nc;i--)
    {
        j=(*A)[i]+B[i];
        if(j>9){*
和大于9*/
```

```

        {
            (*A)[i-1]=(*A)[i-1]+1; /*
高位加上1*/
            (*A)[i]=j-10; /*
和减去10
后存储到当前位*/
        }
        else
            (*A)[i]=j;
        if(i==MaxLen-nc)/*
处理最高位*/
        {
            if(j>9)
            {
                (*A)[i-1]=1;
                length=nc+1;
            }
            else
                length=nc;
        }
    }
    return length;
}
void main()
{
    sqliist A,B;
    int na,nb,nc;
    na=input(A);
    nb=input(B);
    printf("
整数A:");
    output(A,0,na);
    printf("
整数B:");
    output(B,0,nb);
    nc=add(&A,na,B,nb);
    printf("
相加后的结果:");
    output(A,MaxLen-nc,MaxLen);
}

```

程序的运行结果如图3-7所示。

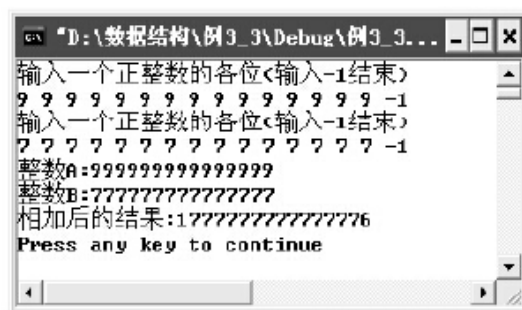


图3-7 两个整数相加的程序运行结果

【考研真题】 设将 n ($n>1$) 个整数存放到一维数组 R 中，试设计一个在时间和空间两方面都尽可能高效的算法，将 R 中保存的序列循环左移 p ($0<p<n$) 个

位置，即把R中的数据序列由 $(x_0, x_1, \dots, x_{n-1})$ 变换为 $(x_p, x_{p+1}, \dots, x_{n-1}, x_0, x_1, \dots, x_{p-1})$ 。要求如下。

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想，采用C语言描述算法。
- (3) 说明所设计算法的时间复杂度和空间复杂度。

【分析】 该题目主要考查对顺序表的掌握情况，具有一定的灵活性。

(1) 先将这n个元素序列 $(x_0, x_1, \dots, x_p, x_{p+1}, \dots, x_{n-1})$ 就地逆置，得到 $(x_{n-1}, x_{n-2}, \dots, x_p, x_{p-1}, \dots, x_0)$ ，然后再将前n-p个元素 $(x_{n-1}, x_{n-2}, \dots, x_p)$ 和后p个元素 $(x_{p-1}, x_{p-2}, \dots, x_0)$ 分别就地逆置，得到最终结果 $(x_p, x_{p+1}, \dots, x_{n-1}, x_0, x_1, \dots, x_{p-1})$ 。

(2) 算法实现，可用Reverse和LeftShift两个函数实现。

```
void Reverse(int R[],int left,int right)
{
    int k=left,j=right,t;
    while(k<j)
    {
        t=R[k];
        R[k]=R[j];
        R[j]=t;
        k++;
        j--;
    }
}
void LeftShift(int R[],int n,int p)
{
    If(p>0 && p<n)
    {
        Reverse(R,0,n-1);           //
        将全部元素逆置
        Reverse(R,0,n-p-1);         //
        逆置前n-p
        个元素
        Reverse(R,n-p,n-1);         //
        逆置后n
        个元素
    }
}
```

(3) 上述算法的时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

3.3 线性表的链式表示与实现

在解决实际问题时，有时并不适合采用线性表的顺序存储结构，例如两个一元多项式的相加、相乘。这就需要采用线性表另一种存储结构——链式存储，本节主要介绍单链表的存储结构及单链表的运算。

3.3.1 单链表的存储结构

线性表的链式存储是采用一组任意的存储单元存放线性表的元素。这组存储单元可以是连续的，也可以是不连续的。因此，为了表示每个元素 a_i 与其直接后继元素 a_{i+1} 的逻辑关系，除了存储元素本身的信息外，还需要存储一个指示其直接后继元素的信息（即直接后继元素的地址）。这两部分构成的存储结构称为**结点**（node）。结点包括**数据域**和**指针域**两个域，数据域存放数据元素的信息，指针域存放元素的直接后继元素的存储地址。指针域中存储的信息称为指针。结点结构如图3-8所示。

通过指针域将线性表中 n 个结点元素按照逻辑顺序链在一起就构成了**链表**，如图3-9所示。由于链表中每一个结点的指针域只有一个，所以将这样的链表称为**线性链表**或者**单链表**。

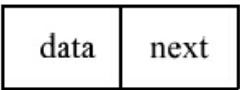


图3-8 结点结构

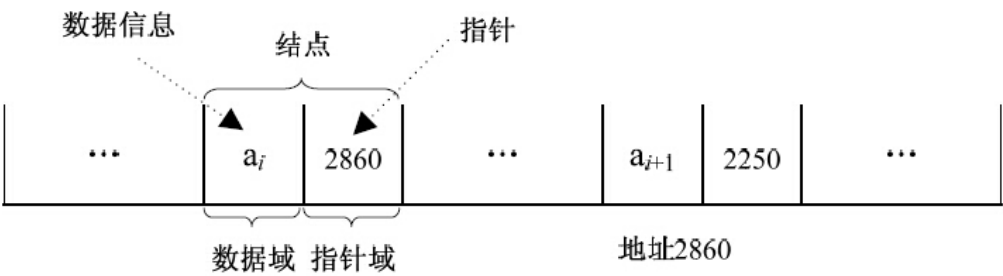


图3-9 单链表

例如，链表（Hou, Geng, Zhou, Hao, Chen, Liu, Yang）在计算机中的存储情况如图3-10所示。

单链表的每个结点的地址存放在其直接前驱结点的指针域中，第一个结点没有直接前驱结点，因此需要一个**头指针** 指向第一个结点。由于表中的最后一个元素没有直接后继元素，需要将单链表的最后一个结点的指针域置为“空”（NULL）。

存取链表必须从头指针head开始，头指针指向链表的第一个结点，通过头指针可以找到链表中的每一个元素。

一般情况下，我们只关心链表中结点的逻辑顺序，而不关心它的实际存储位置。通常用箭头表示指针，把链表表示成通过箭头链接起来的序列。图3-10所示的线性表可以表示成如图3-11的序列。

	存储地址	数据域	指针域
头指针 head <div>32</div>	6	Hao	36
	19	Zhou	6
	32	Hou	51
	36	Chen	47
	43	Yang	NULL
	47	Liu	43
	51	Geng	19

图3-10 线性表的链式存储结构

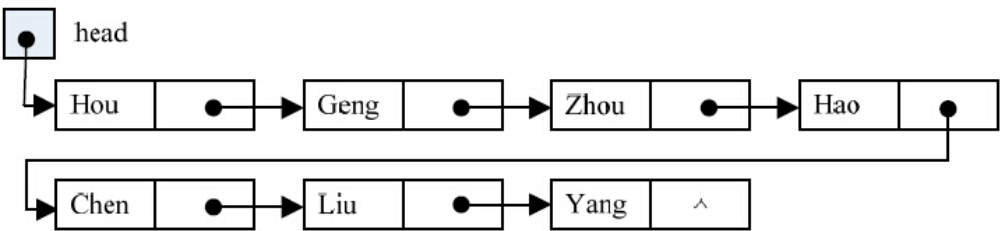


图3-11 单链表的逻辑状态

为了操作方便，在单链表的第一个结点之前增加一个结点，称为**头结点**。头结点的数据域可以存放如线性表的长度等信息，头结点的指针域存放第一个元素结点的地址信息，使其指向第一个元素结点。带头结点的单链表如图3-12所示。

若带头结点的链表为空链表，则头结点的指针域为“空”，如图3-13所示。

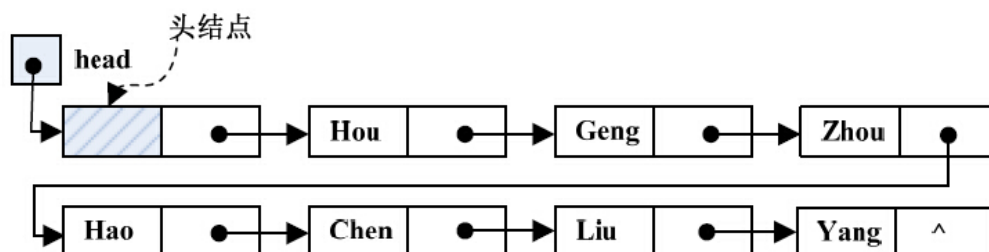


图3-12 带头结点的单链表

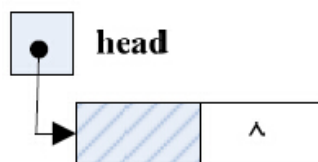


图3-13 带头结点的单链表

注意 初学者需要区分头指针和头结点的区别。头指针是指向链表第一个结点的指针，若链表有头结点，则是指向头结点的指针。头指针链表的必要元素具有标识作用，所以常用头指针冠以链表的名字。头结点是为了操作的统一和方便而设立的，放在第一个元素结点之前，不是链表的必要元素。有了头结点，对在第一个元素结点前插入结点和删除第一个结点，其操作与其他结点的操作就统一了。

单链表的存储结构用C语言描述如下。

```
typedef struct Node
{
    DataType data;
    struct Node *next;
}ListNode,*LinkList;
```

```
LinkList L;
```

```
ListNode *L;
```

单链表上的基本运算有链表的创建、单链表的插入、单链表的删除、单链表的长度等，以下是带头结点的单链表的基本运算具体实现（保存在文件“LinkList.h”中）。

```
void InitList(LinkList *head)
/*
初始化单链表*/
{
    if ((*head=(LinkList)malloc(sizeof(ListNode)))==NULL) /*
为头结点分配一个存储空间*/
        exit(-1);
    (*head)->next=NULL; /*
将单链表的头结点指针域置为空*/
}
```

```
int ListEmpty(LinkList head)
/*
判断单链表是否为空*/
{
    if (head->next==NULL)                /*
如果单链表头结点的指针域为空*/
        return 1;                        /*
返回1*/
    else                                  /*
否则*/
        return 0;                        /*
返回0*/
}
```

③按序号查找操作。从单链表的头指针head出发，利用结点的指针域依次扫描链表的结点，并进行计数，直到计数为i，就找到了第i个结点。如果查找成功，返回该结点的指针，否则返回NULL表示查找失败。按序号查找的算法实现如下。

```
ListNode *Get(LinkList head,int i)
/*
按序号查找单链表中第i
个结点。查找成功返回该结点的指针表示成功；否则返回NULL
表示失败。*/
{
    ListNode *p;
    int j;
    if(ListEmpty(head)) /*
如果链表为空*/
        return NULL; /*
返回NULL*/
    if(i<1) /*
如果序号不合法*/
        return NULL; /*
则返回NULL*/
    j=0;
    p=head;
    while(p->next!=NULL&& j<i)
    {
        p=p->next;
        j++;
    }
    if(j==i) /*
找到第i
个结点*/
        return p; /*
返回指针p*/
    else /*
否则*/
        return NULL; /*
返回NULL*/
}
```

查找元素时，要注意判断条件p->next!=NULL，保证p的下一个结点不为空，如果没有这个条件，就无法保证执行循环体中的p=p->next语句。

④按内容查找，查找元素值为e的结点。从单链表中的头指针开始，依次与e比较，如果找到返回该元素结点的指针；否则返回NULL。查找元素值为e的结点的算法实现如下。

```
ListNode *LocateElem(LinkList head,DataType e)
/*
按内容查找单链表中元素值为e
的元素，若查找成功则返回对应元素的结点指针，否则返回NULL
表示失败。*/
{
    ListNode *p;
    p=head->next; /*
指针p
指向第一个结点*/
    while(p)
    {
        if(p->data==e) /*
没有找到与e
```

```

相等的元素*/
        p=p->next;
继续找下一个元素*/
    else
找到与e
相等的元素*/
        break;
退出循环*/
    }
    return p;
}

```

⑤定位操作。定位操作与按内容查找类似，只是返回的是该结点的序号。从单链表的头指针出发，依次访问每个结点，并将结点的值与e比较，如果相等，返回该序号表示成功；如果没有与e值相等的元素，返回0表示失败。定位操作的算法实现如下。

```

int LocatePos(LinkList head,DataType e)
/*
查找线性表中元素值为e
的元素，查找成功将对应元素的序号返回，否则返回0
表示失败。*/
{
    ListNode *p;
    int i;
    if(ListEmpty(head))
在查找第i
个元素之前，判断链表是否为空*/
        return 0;
    p=head->next;
指针p
指向第一个结点*/
    i=1;
    while(p)
    {
        if(p->data==e)
找到与e
相等的元素*/
            return i;
返回该序号*/
        else
        {
            p=p->next;
            i++;
        }
    }
    if(!p)
如果没有找到与e
相等的元素*/
        return 0;
    return 0;
}

```

⑥在第i个位置插入元素e。插入成功返回1，否则返回0；如果没有与e值相等的元素，返回0表示失败。

假设存储元素e的结点为p，要将p指向的结点插入pre和pre->next之间，根本不需要移动其他结点，只需要让p指向结点的指针和pre指向结点的指针做一点改变即可。

即先把*pre的直接后继结点变成*p的直接后继结点，然后把*p变成*pre的直接后继结点，如图3-14所示，代码如下。

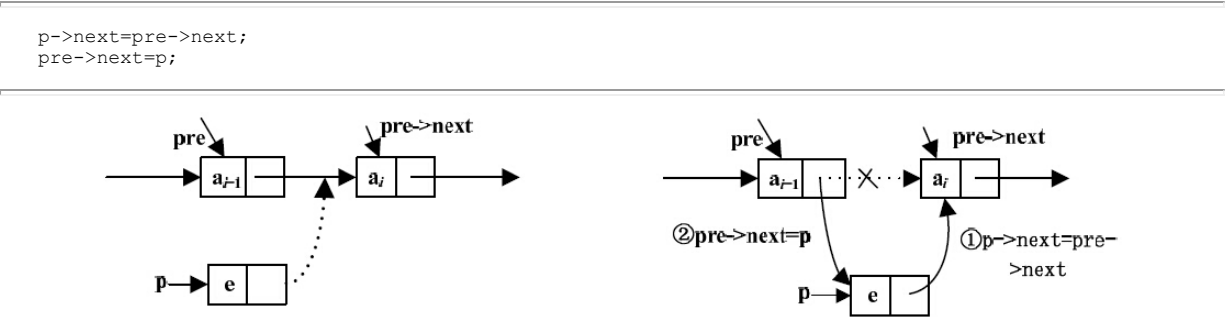


图3-14 在*pre结点之后插入新结点*p

注意 插入结点的两行代码不能颠倒顺序。如果先进行pre->next=p，后进行p->next=pre->next操作，则第一条代码就会覆盖pre->next的地址，pre->next的地址就变成了p的地址，执行p->next=pre->next就等于执行p->next=p，这样pre->next就与上级断开了链接，造成尴尬的局面，如图3-15所示。

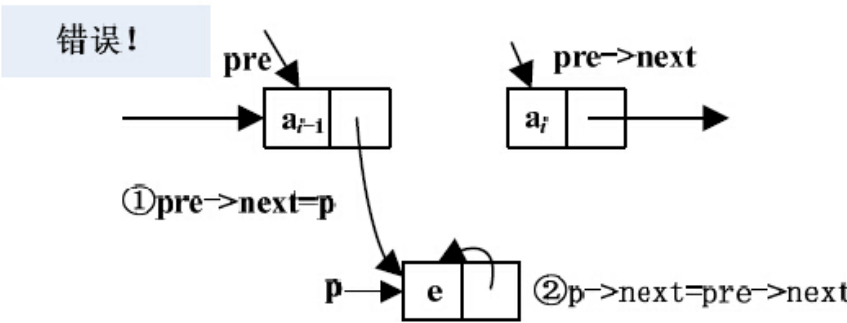


图3-15 插入结点代码顺序颠倒后，*(pre->next)结点与上级断开链接

如果要在单链表的第i个位置插入一个新元素e，首先需要在链表中找到其直接前驱结点，即第i-1个结点，并由指针pre指向该结点，如图3-16所示。然后申请一个新结点空间，由p指向该结点，将值e赋值给p指向结点的数据域，最后修改*p和*pre结点的指针域，如图3-17所示。这样就完成了结点的插入操作。

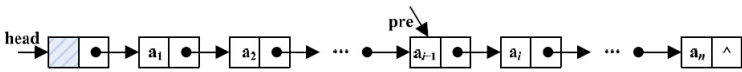


图3-16 找到第i个结点的直接前驱结点

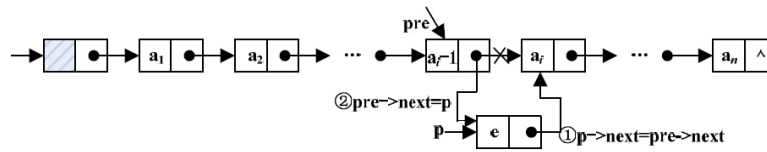


图3-17 将新结点插入第i个位置

在单链表的第i个位置插入新数据元素e的算法实现如下。

```

int InsertList(LinkList head,int i,DataType e)
/*
在单链表中第i
个位置插入一个结点，结点的元素值为e
。插入成功返回1
，失败返回0*/
{
    ListNode *pre,*p; /*
定义第i
个元素的前驱结点指针pre
，指针p
指向新生成的结点*/
    int j;
    pre=head; /*
指针p
指向头结点*/
    j=0;
    while (pre->next!=NULL&& j<i-1) /*
找到第i
-1
个结点，即第i
个结点的前驱结点*/
    {
        pre=pre->next;
        j++;
    }
    if (j!=i-1) /*
如果没找到，说明插入位置错误*/
    {
        printf("
插入位置错误! ");
        return 0;
    }
    /*
新生成一个结点，并将e
赋值给该结点的数据域*/
    if ((p=(ListNode*)malloc(sizeof(ListNode)))==NULL)
        exit(-1);
    p->data=e;
    /*
插入结点操作*/
    p->next=pre->next;
    pre->next=p;
    return 1;
}

```

⑦删除第i个结点。

假设p指向第i个结点，要将*p结点删除，只需要绕过它的直接前驱结点的指针，使其直接指向它的直接后继结点即可删除链表的第i个结点，如图3-18所示。

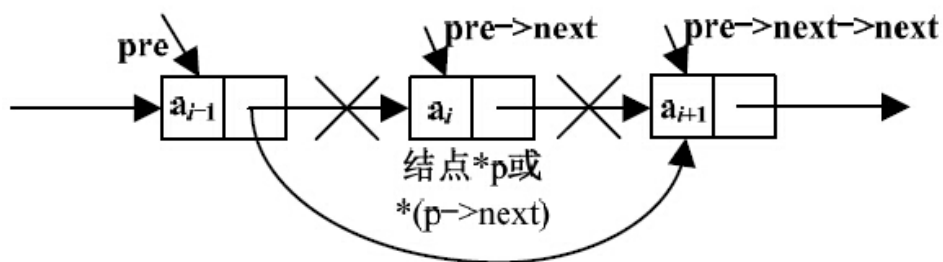


图3-18 删除*pre的直接后继结点

将单链表中第 i 个结点删除可分为3步，第一步找到第 i 个结点的直接前驱结点，即第 $i-1$ 个结点，并用 pre 指向该结点， p 指向其直接后继结点，即第 i 个结点，如图3-19所示；第二步将 $*p$ 结点的数据域赋值给 e ；第三步删除第 i 个结点，即 $pre->next=p->next$ ，并释放 $*p$ 结点的内存空间。删除过程如图3-20所示。

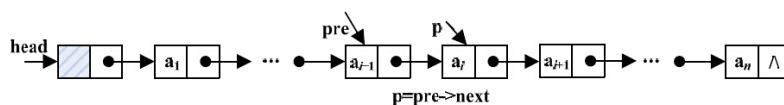


图3-19 找到第 $i-1$ 个结点和第 i 个结点

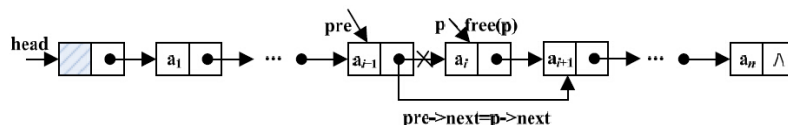


图3-20 删除第 i 个结点

删除第 i 个结点的算法实现如下。

```

int DeleteList(LinkList head,int i,DataType *e)
/*
删除单链表中的第i
个位置的结点。删除成功返回1
，失败返回0*/
{
    ListNode *pre,*p;
    int j;
    pre=head;
    j=0;
    while(pre->next!=NULL&&pre->next->next!=NULL&&j<i-1)/*
判断是否找到前驱结点*/
    {
        pre=pre->next;
        j++;
    }
    if(j!=i-1)/*
    如果没有找到要删除的结点位置，说明删除位置有误*/
    {
        printf("
删除位置有误");
    }
}

```

```

        return 0;
    }
    /*
    指针p
    指向单链表中的第i
    个结点，并将该结点的数据域值赋值给e*/
    p=pre->next;
    *e=p->data;
    /*
    将前驱结点的指针域指向要删除结点的下一个结点，也就是将p
    指向的结点与单链表断开*/
    pre->next=p->next;
    free(p);
    /*
    释放p
    指向的结点*/
    return 1;
}

```

注意 在查找第i-1个结点时，要注意不可遗漏判断条件pre->next->next!=NULL，确保第i个结点非空。如果没有此判断条件，而pre指针指向了单链表的最后一个结点，在执行循环后的p=pre->next，*e=p->data操作时，p指针就指向的是NULL指针域，会产生致命错误。

⑧求表长操作。求表长操作即返回单链表的元素个数，求单链表的表长算法实现代码如下。

```

int ListLength(LinkList head)
/*
求表长操作*/
{
    ListNode *p;
    int count=0;
    p=head;
    while (p->next!=NULL)
    {
        p=p->next;
        count++;
    }
    return count;
}

```

⑨销毁链表操作，实现代码如下。

```

void DestroyList(LinkList head)
/*
销毁链表*/
{
    ListNode *p,*q;
    p=head;
    while (p!=NULL)
    {
        q=p;
        p=p->next;
        free(q);
    }
}

```

3.3.3 单链表存储结构与顺序存储结构的优缺点

下面简单对单链表存储结构和顺序存储结构进行对比。

1. 存储分配方式

顺序存储结构用一组连续的存储单元依次存储线性表的数据元素。单链表采用链式存储结构，用一组任意的存储单元存放线性表的数据元素。

2. 时间性能

采用顺序存储结构时，查找操作时间复杂度为 $O(1)$ ，插入和删除操作需要移动平均一半的数据元素，时间复杂度为 $O(n)$ 。采用单链表存储结构时，查找操作时间复杂度为 $O(n)$ ，插入和删除操作不需要大量移动元素，时间复杂度仅为 $O(1)$ 。

3. 空间性能

采用顺序存储结构时，需要预先分配存储空间，分配的空间过大会造成浪费，分配的过小不能满足问题需要。采用单链表存储结构时，可根据需要临时分配，不需要估计问题的规模大小，只要内存够就可以分配，还可以用于一些特殊情况，如一元多项式的表示。

3.3.4 单链表应用举例

【例3-4】 已知两个单链表A和B，其中的元素都是非递减排列，编写算法将单链表A和B合并得到一个递减有序的单链表C（值相同的元素只保留一个），并要求利用原链表结点空间。

【分析】 此题为单链表合并问题。利用头插法建立单链表，使先插入元素值小的结点在链表末尾，后插入元素值大的结点在链表表头。初始时，单链表C为空（插入的是C的第一个结点），将单链表A和B中较小的元素值结点插入C中；单链表C不为空时，比较C和将插入结点的元素值大小，值不同时插入到C中，值相同时，释放该结点。当A和B中有一个链表为空时，将剩下的结点依次插入C中。程序的实现代码如下。

```
/*
头文件*/
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
typedef int DataType;
typedef struct Node
{
    DataType data;
    struct Node *next;
}ListNode,*LinkedList;
#include"LinkedList.h" /*
包含单链表基本操作实现文件*/
void MergeList(LinkedList A,LinkedList B,LinkedList *C); /*
函数声明：将单链表A
和B
的元素合并到C
中*/
void main()
{
    int i;
    DataType a[]={8,10,15,21,67,91};
    DataType b[]={5,9,10,13,21,78,91};
    LinkedList A,B,C; /*
声明单链表A
和B*/
    ListNode *p;
    InitList(&A); /*
初始化单链表A*/
    InitList(&B); /*
初始化单链表B*/
    for(i=1;i<=sizeof(a)/sizeof(a[0]);i++) /*
利用数组元素创建单链表A*/
    {
        if(InsertList(A,i,a[i-1])==0)
        {
            printf("
插入位置不合法!");
            return;
        }
    }
    for(i=1;i<=sizeof(b)/sizeof(b[0]);i++) /*
利用数组元素创建单链表B*/
    {
        if(InsertList(B,i,b[i-1])==0)
        {
            printf("
插入位置不合法!");
            return;
        }
    }
    printf("
单链表A
中的元素有%d
个: \n",ListLength(A));
    for(i=1;i<=ListLength(A);i++) /*
输出单链表A*/
    {
        p=Get(A,i); /*
返回单链表A
中的每个结点的指针*/
        if(p)
            printf("%4d",p->data); /*
. . . . .
```

```

输出单链表A
中的每个元素*/
        }
        printf("\n");
        printf("
单链表B
中的元素有%d
个: \n",ListLength(B));
        for(i=1;i<=ListLength(B);i++)
            /*
输出单链表B*/
            {
                p=Get(B,i);
                /*
返回单链表B
中的每个结点的指针*/
                if(p)
                    printf("%4d",p->data);
            /*
输出单链表B
中的每个元素*/
            }
            printf("\n");
            MergeList(A,B,&C);
            /*
将单链表A
和B
中的元素合并到C
中*/
            printf("
将A
和B
的元素合并成一个递减有序的单链表C
,C
中有%d
个元素: \n",ListLength(C));
            for(i=1;i<=ListLength(C);i++)
            {
                p=Get(C,i);
                /*
返回单链表C
中每个结点的指针*/
                if(p)
                    printf("%4d",p->data);
            /*
显示输出C
中所有元素*/
            }
            printf("\n");
        }
    }
void MergeList(LinkList A,LinkList B,LinkList *C)
/*
将非递减排列的单链表A
和B
中的元素合并到C
中,使C
中的元素按递减排列,相同值的元素只保留一个*/
{
    ListNode *pa,*pb,*qa,*qb; /*
定义指向单链表A
,B
的指针*/
    pa=A->next;
    /*pa
指向单链表A*/
    pb=B->next;
    /*pb
指向单链表B*/
    free(B);
    /*
释放单链表B
的头结点*/
    *C=A;
    /*
初始化单链表C
,利用单链表A
的头结点作为C
的头结点*/
    (*C)->next=NULL;
    /*
单链表C
初始时空*/
    /*
利用头插法将单链表A
和B
中的结点插入到单链表C
中(先插入元素值较小的结点)*/
    while(pa&&pb)
    /*
单链表A
和B
均不空时*/
    {

```

```

指向结点元素值较小时，将pa
指向的结点插入到C
中*/
        if (pa->data<pb->data) /*pa
        {
            qa=pa;
            pa=pa->next;
            if ((*C)->next==NULL) /*
            {
                qa->next=(*C)->next;
                (*C)->next=qa;
            }
            else if ((*C)->next->data<qa->data) /*pa
            指向的结点元素值不同于已有结点元素值时，才插入结点*/
            {
                qa->next=(*C)->next;
                (*C)->next=qa;
            }
            else /*
            否则，释放元素值相同的结点*/
            {
                free(qa);
            }
        }
        else /*pb
        指向结点元素值较小，将pb
        指向的结点插入到C
        中*/
        {
            qb=pb;
            pb=pb->next;
            if ((*C)->next==NULL) /*
            {
                qb->next=(*C)->next;
                (*C)->next=qb;
            }
            else if ((*C)->next->data<qb->next) /*pb
            指向的结点元素值不同于已有结点元素时，才将结点插入*/
            {
                qb->next=(*C)->next;
                (*C)->next=qb;
            }
            else /*
            否则，释放元素值相同的结点*/
            {
                free(qb);
            }
        }
    }
    while (pa) /*
    如果pb
    为空、pa
    不为空，则将pa
    指向的后继结点插入到C
    中*/
    {
        qa=pa;
        pa=pa->next;
        if ((*C)->next&&(*C)->next->data<qa->data)
        {
            qa->next=(*C)->next;
            (*C)->next=qa;
        }
        else
            free(qa);
    }
    while (pb) /*
    如果pa
    为空、pb
    不为空，则将pb
    指向的后继结点插入到C
    中*/
    {
        qb=pb;
        pb=pb->next;
        if ((*C)->next&&(*C)->next->data<qb->data)
        {
            qb->next=(*C)->next;
            (*C)->next=qb;
        }
    }

```



```

        }
        else
            free(qb);
    }
}

```

程序的运行结果如图3-21所示。

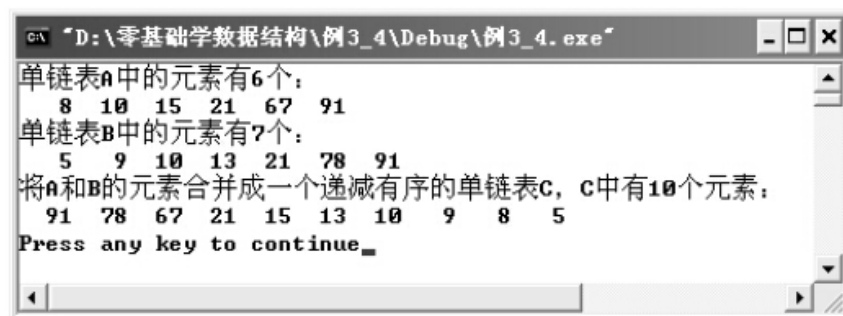


图3-21 合并单链表的程序运行结果

在将两个单链表A和B的合并算法MergeList中，需要特别注意的是，不要遗漏单链表为空时的处理。当单链表为空时，将结点插入C中，代码如下。

```

if ((*C)->next==NULL) /*
    单链表C
    为空时，直接将结点插入C
    中*/
{
    qa->next=(*C)->next;
    (*C)->next=qa;
}

```

针对这个题目，经常会遗漏单链表为空的情况，以下代码遗漏了单链表为空的情况。

```

if ((*C)->next&&(*C)->next->data<qb->next)
{
    qb->next=(*C)->next;
    (*C)->next=qb;
}

```

所以，对于初学者而言，写完算法后，一定要上机调试下算法的正确性。

【例3-5】 利用单链表的基本运算，求两个集合的交集。

【分析】 假设A和B是两个带头结点的单链表，分别表示两个给定的集合A和B，求 $C=A \cap B$ 。先将单链表A和B分别从小到大排序，然后依次比较两个单链表中的元素值大小，pa指向A中当前比较的结点，pb指向B中当前比较的结点，如果pa->data<pb->data，则pa指向A中下一个结点；如果pa->data>pb->data，则pb指向B中下一个结点；如果pa->data==pb->data，则将当前结点插入C中。

程序实现如下。

```
/*
头文件*/
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
/*
单链表类型定义*/
typedef int DataType;
typedef struct Node
{
    DataType data;
    struct Node *next;
}ListNode,*LinkList;
#include"LinkList.h"
单链表基本操作实现文件*/
void Sort(LinkList S);
void DispList( LinkList S);
void Interction(LinkList A,LinkList B,LinkList *C);    /*
求A
和B
的交集的函数声明*/
void main()
{
    int i;
    DataType a[]={5,9,6,20,70,58,44,81};
    DataType b[]={21,81,8,31,5,66,20,95,50};
    LinkList A,B,C;

    声明单链表A
    、B
    和C*/

    ListNode *p;
    InitList(&A);

    初始化单链表A*/
    InitList(&B);

    初始化单链表B*/
    for(i=1;i<=sizeof(a)/sizeof(a[0]);i++)

    将数组a
    中元素插入到单链表A
    中*/
    {
        if(InsertList(A,i,a[i-1])==0)
        {
            printf("
位置不合法");
            return;
        }
    }
    for(i=1;i<=sizeof(b)/sizeof(b[0]);i++)

    将数组b
    中元素插入单链表B
    中*/
    {
        if(InsertList(B,i,b[i-1])==0)
        {
            printf("
位置不合法");
            return;
        }
    }
}
```

```

    }
    printf("
单链表A
中的元素有%d
个: \n",ListLength(A));
    for (i=1;i<=ListLength(A);i++)
        /*
输出单链表A
中的每个元素*/
    {
        p=Get(A,i);
        /*
返回单链表A
中的每个结点的指针*/
        if (p)
            printf("%4d",p->data);
        /*
输出单链表A
中的每个元素*/
    }
    printf("\n");
    printf("
单链表B
中的元素有%d
个: \n",ListLength(B));
    for (i=1;i<=ListLength(B);i++)
    {
        p=Get(B,i);
        /*
返回单链表B
中的每个结点的指针*/
        if (p)
            printf("%4d",p->data);
        /*
输出单链表B
中的每个元素*/
    }
    printf("\n");
    Interction(A,B,&C);
    /*
求A
和B
的交集*/
    printf("A
和B
的交集有%d
个元素: \n",ListLength(C));
    for (i=1;i<=ListLength(C);i++)
    {
        p=Get(C,i);
        /*
返回单链表C
中每个结点的指针*/
        if (p)
            printf("%4d",p->data);
        /*
显示输出C
中所有元素*/
    }
    printf("\n");
}
void Interction(LinkList A,LinkList B,LinkList *C)
/*
求A
和B
的交集*/
{
    ListNode *pa,*pb,*pc;
    Sort(A);
    printf("
排序后A
中的元素:\n");
    DispList(A);
    Sort(B);
    printf("
排序后B
中的元素:\n");
    DispList(B);
    pa=A->next;
    pb=B->next;
    *C=(LinkList)malloc(sizeof(ListNode));
    (*C)->next=NULL;
    while (pa&&pb)
    {
        if (pa->data<pb->data)
            pa=pa->next;
        else if (pa->data>pb->data)
            pb=pb->next;
        else

```

```

        {
            pc=(ListNode*)malloc(sizeof(ListNode));
            pc->data=pa->data;
            pc->next=(*C)->next;
            (*C)->next=pc;
            pa=pa->next;
            pb=pb->next;
        }
    }
}

void Sort(LinkList S)
/*
利用选择排序法对链表s
进行从小到大排序*/
{
    ListNode *p,*q,*r;
    DataType t;
    p=S->next;
    while(p->next)
    {
        r=p;
        q=p->next;
        while(q)
        {
            if(r->data>q->data)
                r=q;
            q=q->next;
        }
        if(p!=r)
        {
            t=p->data;
            p->data=r->data;
            r->data=t;
        }
        p=p->next;
    }
}

void DispList(LinkList S)
/*
输出链表*/
{
    ListNode *p,*q;
    p=S->next;
    while(p)
    {
        q=p;
        printf("%4d",p->data);
        p=p->next;
    }
    printf("\n");
}

```

程序的运行结果如图3-22所示。

图3-22 求A和B交集的程序运行结果

【考研真题】 假设一个带有表头结点的单链表，结点结构如下。

data	link
------	------

假设该链表只给出了头指针list，在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点（k为正整数）。若查找成功，算法输出该结点数据域的值，并返回1；否则返回0。要求如下。

- (1) 描述算法的基本设计思想。
- (2) 描述算法的详细实现步骤。
- (3) 根据设计思想和实现步骤，采用程序设计语言描述算法。

【分析】 这是2009年的考研试题，主要考查对链表的掌握程度，这个题目比较灵活，利用一般的思维方式不容易实现。

(1) 算法的基本思想：定义两个指针p和q，初始时均指向头结点的下一个结点。p指针沿着链表移动，当p指针移动到第k个结点时，q指针与p指针同步移动，当p指针移动到链表表尾结点时，q指针所指向的结点即为倒数第k个结点。

(2) 算法的详细步骤如下。

- ①令count=0，p和q指向链表的第一个结点。
- ②若p为空，则转向e执行。
- ③若count等于k，则q指向下一个结点；否则令count++。
- ④令p指向下一个结点，转向b执行。

⑤若count等于k，则查找成功，输出结点的数据域的值，并返回1；否则，查找失败，返回0。

(3) 算法实现代码如下。

```
typedef struct LNode
{
    int data;
    struct LNode *link;
}*LinkList;
int SearchNode(LinkList list,int k)
{
    LinkList p,q;
    int count=0;//
    计数器初值为0
    p=q=list->link;                                //p
    和q
    指向链表的第一个结点
    while(p!=NULL)
    {
        if(count<k)                                //
        让p
        移到第k
        个结点后
            count++;
        else
            q=q->link;                                //
        当p
        移到第k
        个结点后，q
        开始与p
        同步移动下一个结点
        p=p->link;                                    //p
        移动到下一个结点
    }
    if(count<k)
        return 0;
    else
    {
        printf("
        倒数第%d
        个结点元素值为%d\n",k,q->data); //
        输出倒数第k
        个结点的值
        return 1;
    }
}
```

3.4 循环单链表

循环单链表是首尾相连的单链表，是另一种形式的单链表。本节主要从循环单链表的存储结构并结合实例讲解循环单链表。

3.4.1 循环链表的链式存储

循环单链表（circular linked list）是首尾相连的一种单链表。将单链表的最后一个结点的指针域由空指针改为指向头结点或第一个结点，整个链表就形成一个环，这样的单链表称为**循环单链表**。从表中任何一个结点出发均可找到表中其他结点。

与单链表类似，循环单链表也可分为带头结点结构和不带头结点结构两种。对于不带头结点的循环单链表，当表不为空时，最后一个结点的指针域指向头结点，如图3-23所示。对于带头结点的循环单链表，当表为空时，头结点的指针域指向头结点本身，如图3-24所示。



图3-23 循环单链表

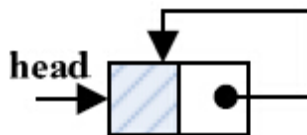


图3-24 结点为空的循环单链表

循环单链表与单链表在结构、类型定义及实现方法上都是一样的，唯一的区别仅在于判断链表是否为空的条件下。判断单链表为空的条件是 $\text{head} \rightarrow \text{next} == \text{NULL}$ ，判断循环单链表为空的条件是 $\text{head} \rightarrow \text{next} == \text{head}$ 。

在单链表中，访问第一个结点的时间复杂度为 $O(1)$ ，而访问最后一个结点则需要将整个单链表扫描一遍，故时间复杂度为 $O(n)$ 。对于循环单链表，只需设置一个**尾指针**（利用 rear 指向循环单链表的最后一个结点）而不设置头指针，就可以直接访问最后一个结点，时间复杂度为 $O(1)$ 。访问第一个结点即 $\text{rear} \rightarrow \text{next} \rightarrow \text{next}$ ，时间复杂度也为 $O(1)$ 。如图3-25所示。



图3-25 仅设置尾指针的循环单链表

在循环单链表中设置尾指针，还可以使有些操作变得简单，例如要将如图3-26所示的两个循环单链表（尾指针分别为 LA 和 LB ）合并成一个链表，只需要将一个表的表尾和另一个表的表头连接即可，如图3-27所示。

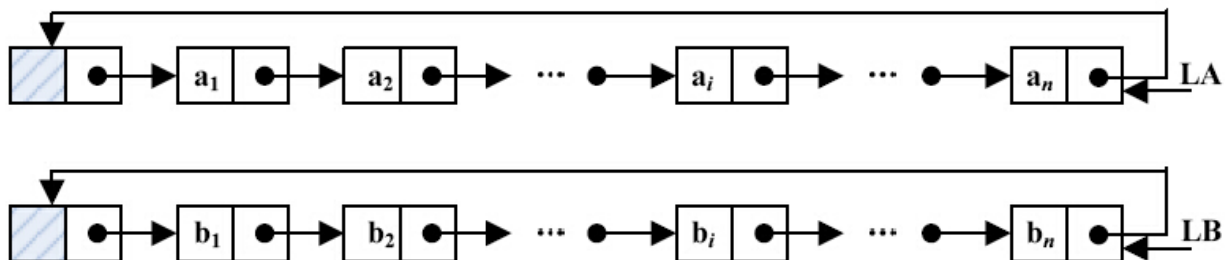


图3-26 两个设置尾指针的循环单链表

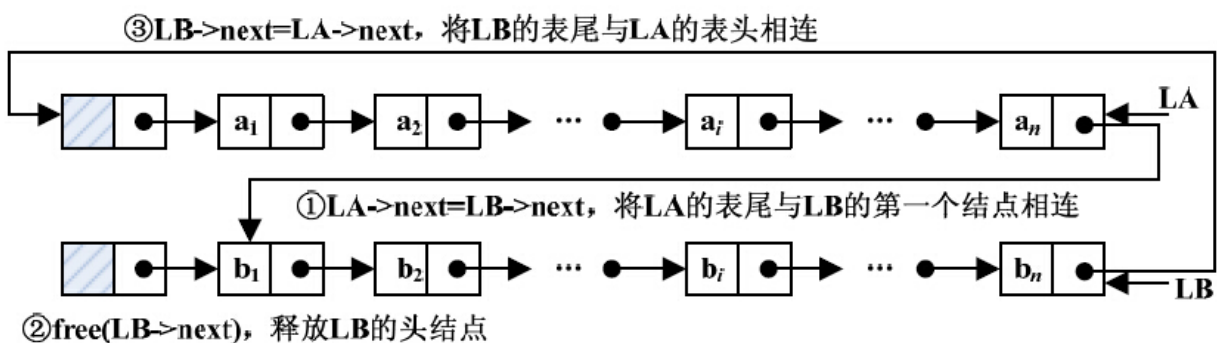


图3-27 合并两个设置尾指针的循环单链表

将循环单链表合并为一个循环单链表只需要3步操作，第一步将LA的表尾与LB的第一个结点相连，即 $LA \rightarrow next = LB \rightarrow next \rightarrow next$ ；第二步释放LB的头结点，即 $free(LB \rightarrow next)$ ；第三步将LB的表尾与LA的表头相连，即 $LB \rightarrow next = LA \rightarrow next$ 。

对于设置了头指针的两个循环单链表（头指针分别是head1和head2），要将其合并成一个循环单链表，需要先找到两个链表的最后一个结点，分别增加一个尾指针，分别使其指向最后一个结点。然后将第一个链表的尾指针与第二个链表的第一个结点连接起来，第二个链表的尾指针与第一个链表的第一个结点连接起来，就形成了一个循环链表。

合并两个循环单链表的算法实现如下。

```
LinkList Link(LinkList head1, LinkList head2)
/*
将两个链表head1
和head2
连接在一起形成一个循环链表*/
{
    ListNode *p, *q;
    p=head1;
    while(p->next!=head1)          /*
指针p
指向链表的最后一个结点*/
        p=p->next;
    q=head2;
    while(q->next!=head2)          /*
指针q
指向链表的最后一个结点*/
        q=q->next;
    p->next=head2->next;           /*
将第一个链表的尾端连接到第二个链表的第一个结点*/
    q->next=head1;                 /*
将第二个链表的尾端连接到第一个链表的第一个结点*/
    return head1;
}
```

说明 部分图书把循环单链表中的头结点称为哨兵结点。

3.4.2 循环单链表应用举例

【例3-6】 已知一个带哨兵结点h的循环单链表中的数据元素含有正数和负数，试编写一个算法，构造两个循环单链表，使一个循环单链表中只含正数，另一个循环单链表只含负数。

【分析】 初始时，先创建两个空的单链表ha和hb，然后依次查看指针p指向的结点元素值，如果值为正数，则将其插入ha中，否则将其插入hb中。最后使最后一个结点的指针域指向头结点，构成循环单链表。

程序实现代码如下所示。

```
/*
头文件*/
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
/*
单链表类型定义*/
typedef int DataType;
typedef struct Node
{
    DataType data;
    struct Node *next;
}ListNode, *LinkedList;
/*
函数声明*/
LinkedList CreateCycList();/*
创建一个循环单链表*/
void Split(LinkedList ha,LinkedList hb);/*
按ha
中元素值的正数和负数分成两个循环单链表ha
和hb*/
void DispCycList(LinkedList head);/*
输出循环单链表*/
void main()
{
    LinkedList ha,hb=NULL;
    ListNode *s,*p;
    ha=CreateCycList();
    p=ha;
    while(p->next!=ha)/*
找ha
的最后一个结点, p
指向该结点*/
        p=p->next;
    /*
为ha
添加哨兵结点*/
    s=(ListNode*)malloc(sizeof(ListNode));
    s->next=ha;
    ha=s;
    p->next=ha;
    /*
创建一个空的循环单链表hb*/
    s=(ListNode*)malloc(sizeof(ListNode));
    s->next=hb;
    hb=s;
    Split(ha,hb);
    printf("
输出循环单链表A(
正数):\n");
    DispCycList(ha);
    printf("
输出循环单链表B(
负数):\n");
    DispCycList(hb);
}
void Split(LinkedList ha,LinkedList hb)
/*
将一个循环单链表ha
构造成两个循环单链表, 其中ha
中的元素只含正数, hb
... ..*/
```

```

中的元素只含负数*/
{
    ListNode *ra,*rb,*p=ha->next;
    int v;
    ra=ha;
    ra->next=NULL;
    rb=hb;
    rb->next=NULL;
    while(p!=ha)
    {
        v=p->data;
        if(v>0)/*
若元素值大于0
, 插入ha
中*/
        {
            ra->next=p;
            ra=p;
        }
        else/*
若元素值小于0
, 插入hb
中*/
        {
            rb->next=p;
            rb=p;
        }
        p=p->next;
    }
    ra->next=ha;/*
变为循环单链表*/
    rb->next=hb;/*
变为循环单链表*/
}
LinkedList CreateCycList()
/*
创建循环单链表*/
{
    ListNode *h=NULL,*s,*t=NULL;
    DataType e;
    int i=1;
    printf("
创建一个循环单链表(
输入0
表示创建链表结束):\n");
    while(1)
    {
        printf("
请输入第%d
个结点的data
域值:", i);

        scanf("%d",&e);
        if(e==0)
            break;
        if(i==1)
        {
            h=(ListNode*)malloc(sizeof(ListNode));
            h->data=e;
            h->next=NULL;
            t=h;
        }
        else
        {
            s=(ListNode*)malloc(sizeof(ListNode));
            s->data=e;
            s->next=NULL;
            t->next=s;
        }
        i++;
    }
    t->next=h;
}

```

```

        t=s;
    }
    i++;
}
if(t!=NULL)
    t->next=h;
return h;
}
void DispCycList(LinkList h)
/*
输出循环单链表*/
{
    ListNode *p=h->next;
    if(p==NULL)
    {
        printf("
链表为空!\n");
        return;
    }
    while(p->next!=h)
    {
        printf("%4d",p->data);
        p=p->next;
    }
    printf("%4d",p->data);
    printf("\n");
}

```

程序运行结果如图3-28所示。

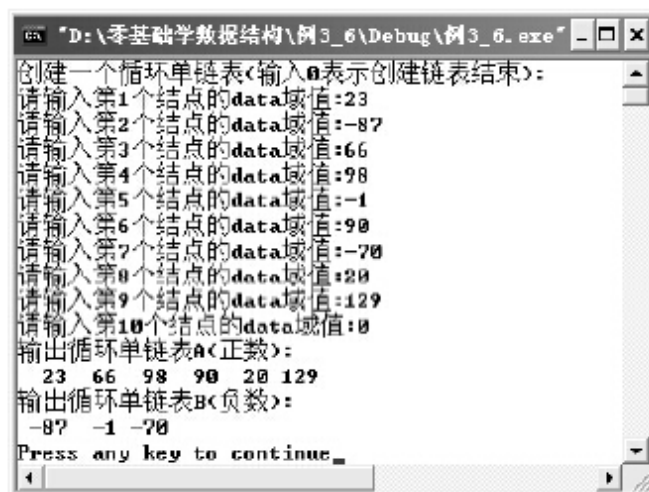


图3-28 程序运行结果

从以上程序容易看出，循环单链表的创建与单链表的创建基本一样，只是最后增加了如下一条语句使最后一个结点指向第一个结点，

构成一个循环单链表。

```
if (t!=NULL)
    t->next=h;
```

3.5 双向链表

在单链表和循环单链表中，每一个结点的指针域只有一个，只能根据指针域查找后继结点，要查找指针p指向结点的直接前驱结点，必须从p指针出发，顺着指针域把整个链表访问一遍，才能找到该结点，其时间复杂度是 $O(n)$ 。因此，要访问某个结点的前驱结点，效率太低，为了便于操作，可以将单链表设计成双向链表。本节主要介绍双向链表的存储结构及双向链表存储结构下的操作实现。

3.5.1 双向链表的存储结构

顾名思义，双向链表（double linked list）就是链表中的每个结点有两个指针域：一个指向直接前驱结点，另一个指向直接后继结点。双向链表的每个结点有data域、prior域和next域3个域。双向链表的结点结构如图3-29所示。

其中，data域为数据域，存放数据元素；prior域为前驱结点指针域，指向直接前驱结点；next域为后继结点指针域，指向直接后继结点。

与单链表类似，也可以为双向链表增加一个头结点，这样使某些操作更加方便。双向链表也有循环结构，称为双向循环链表（double

circular linked list)。带头结点的双向循环链表如图3-30所示。
双向循环链表为空的情况如图3-31所示，判断带头结点的双向循环链表为空的条件是 $\text{head} \rightarrow \text{prior} == \text{head}$ 或 $\text{head} \rightarrow \text{next} == \text{head}$ 。



图3-29 双向链表的结点结构

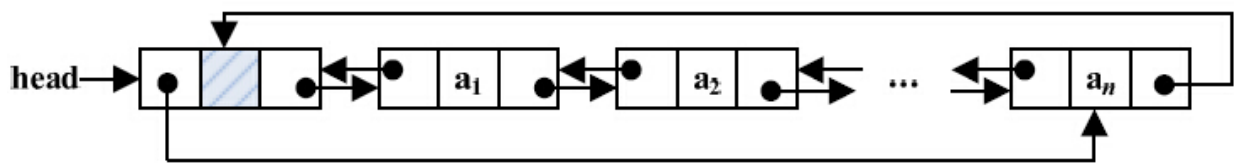


图3-30 带头结点的双向循环链表

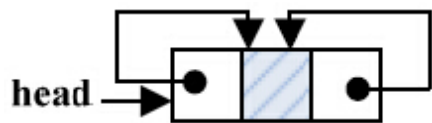


图3-31 带头结点的空双向循环列表

在双向链表中，因为每个结点既有前驱结点的指针域又有后继结点的指针域，所以查找结点非常方便。对于带头结点的双向链表中，如果链表为空，则有 $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{prior}$ 。

双向链表的结点存储结构描述如下。

```
typedef struct Node
{
    DataType data;
    struct Node *prior;
```



```
struct Node *next;  
}DListNode,*DLinkedList;
```

3.5.2 双向链表的插入和删除操作

在双向链表中，有些操作如求链表的长度、查找链表的第 i 个结点等，仅涉及一个方向的指针，与单链表中的算法实现基本没什么区别。但是对于双向循环链表的插入和删除操作，因为涉及前驱结点和后继结点的指针，所以需要修改两个方向上的指针。

1. 在第 i 个位置插入元素值为 e 的结点

首先找到第 i 个结点，用 p 指向该结点；再申请一个新结点，由 s 指向该结点，将 e 放入数据域；然后修改 p 和 s 指向的结点的指针域，修改 s 的prior域，使其指向 p 的直接前驱结点，即 $s \rightarrow \text{prior} = p \rightarrow \text{prior}$ ；修改 p 的直接前驱结点的next域，使其指向 s 指向的结点，即 $p \rightarrow \text{prior} \rightarrow \text{next} = s$ ；修改 s 的next域，使其指向 p 指向的结点，即 $s \rightarrow \text{next} = p$ ；修改 p 的prior域，使其指向 s 指向的结点，即 $p \rightarrow \text{prior} = s$ 。插入操作指针修改情况如图3-32所示。

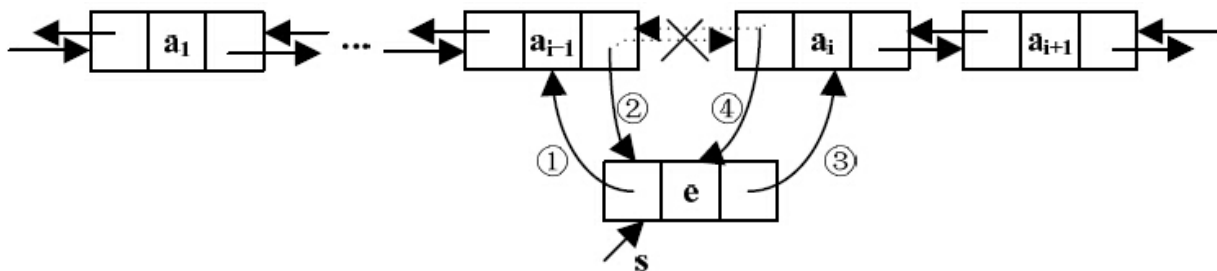


图3-32 双向循环链表的插入结点操作过程

插入操作算法实现如下。

```
int InsertDList(DListLink head,int i,DataType e)
{
    DListNode *p,*s;
    int j;
    p=head->next;
    j=0;
    while(p!=head&& j<i)
    {
        p=p->next;
        j++;
    }
    if(j!=i)
    {
        printf("
插入位置不正确");
        return 0;
    }
    s=(DListNode*)malloc(sizeof(DListNode));
    if(!s)
        return -1;
    s->data=e;
    s->prior=p->prior;
    p->prior->next=s;
    s->next =p;
    p->prior=s;
    return 1;
}
```

2. 删除第i个结点

首先找到第i个结点，用p指向该结点；然后修改p指向的结点的直接前驱结点和直接后继结点的指针域，从而将p与链表断开。将p指向的结点与链表断开需要两步，第一步，修改p的前驱结点的next域，使其指向p的直接后继结点，即p->prior->next=p->next；第二步，修改p的直接后继结点的prior域，使其指向p的直接前驱结点，即p->next->prior=p->prior。删除操作指针修改情况如图3-33所示。

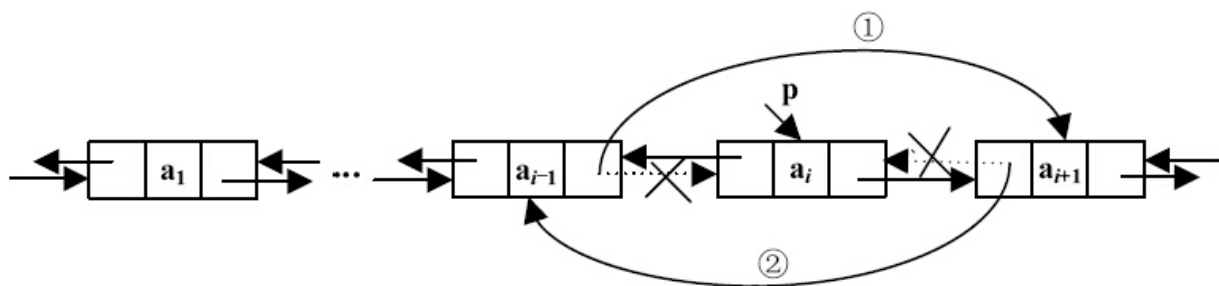


图3-33 双向循环链表的删除结点操作过程

删除操作算法实现如下。

```

int DeleteDList(DListLink head,int i,DataType *e)
{
    DListNode *p;
    int j;
    p=head->next;
    j=0;
    while(p!=head&& j<i)
    {
        p=p->next;
        j++;
    }
    if(j!=i)
    {
        printf("
删除位置不正确");
        return 0;
    }
    p->prior->next=p->next;
    p->next->prior =p->prior;
    free(p);
    return 1;
}

```

插入和删除操作的时间耗费主要在查找结点上，两者的时间复杂度都为 $O(n)$ 。

说明 双向链表的插入和删除操作需要修改结点的prior域和next域，比单链表操作要复杂些，因此要注意修改结点的指针域的顺序。

3.5.3 双向链表应用举例

【例3-7】约瑟夫问题。有n个小朋友，编号分别为1, 2, …, n, 按编号围成一个圆圈，他们按顺时针方向从编号为k的人由1开始报数，报数为m的人出列，他的下一个人重新从1开始报数，数到m的人出列，照这样重复下去，直到所有人都出列。编写一个算法，输入n、k和m，按照出列顺序输出编号。

【分析】解决约瑟夫问题可以分为3个步骤，第一步创建一个具有n个结点的不带头结点的双向循环链表（模拟编号从1~n的圆圈可以利用循环单链表实现，这里采用双向循环链表实现），编号从1到n，代表n个小朋友；第二步找到第k个结点，即第一个开始报数的人；第三步，编号为k的人从1开始报数，并开始计数，报到m的人出列即将该结点删除。继续从下一个结点开始报数，直到最后一个结点被删除。程序代码实现如下。

```
/*
头文件*/
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
/*
双向链表类型定义*/
typedef int DataType;
typedef struct Node
{
    DataType data;
    struct Node *prior;
    struct Node *next;
}DListNode,*DLinkedList;
/*
函数声明*/
DLinkedList CreatedCList(int n);/*
创建一个长度为n
的双向循环链表的函数声明*/
void Josephus(DLinkedList head,int n,int m,int k);          /*
```

```

在长度为n
的双向循环链表中，报数为编号为m
的出列*/
int InitDList(DLinkedList *head);
void main()
{
    DLinkedList h;
    int n,k,m;
    printf("
输入环中人的个数n=");
    scanf("%d",&n);
    printf("
输入开始报数的序号k=");
    scanf("%d",&k);
    printf("
报数为m
的人出列m=");
    scanf("%d",&m);
    h=CreateDCList(n);
    Josephus(h,n,m,k);
}
void Josephus(DLinkedList head,int n,int m,int k)
/*
在长度为n
的双向循环链表中，从第k
个人开始报数，数到m
的人出列*/
{
    DListNode *p,*q;
    int i;
    p=head;
    for(i=1;i<k;i++) /*
从第k
个人开始报数*/
    {
        q=p;
        p=p->next;
    }
    while(p->next!=p)
    {
        for(i=1;i<m;i++) /*
数到m
的人出列*/
        {
            q=p;
            p=p->next;
        }
        q->next=p->next; /*
将p
指向的结点删除，即报数为m
的人出列*/
        p->next->prior=q;
        printf("%4d",p->data);/*
输出被删除的结点*/
        free(p);
        p=q->next; /*p
指向下一个结点，重新开始报数*/
    }
    printf("%4d\n",p->data);
}
DLinkedList CreateDCList(int n)
/*
创建双向循环链表*/
{
    DLinkedList head=NULL;
    DListNode *s,*q;
    int i;

```

```

        for(i=1;i<=n;i++)
        {
            s=(DListNode*)malloc(sizeof(DListNode));
            s->data=i;
            s->next=NULL;
            /*
将新生成的结点插入到双向循环链表*/
            if(head==NULL)
            {
                head=s;
                s->prior=head;
                s->next=head;
            }
            else
            {
                s->next=q->next;
                q->next=s;
                s->prior=q;
                head->prior=s;
            }
            q=s;
        }
        /*q
始终指向链表得最后一个结点*/
        return head;
    }
int InitDList(DLinkedList *head)
/*
初始化双向循环链表*/
{
    *head=(DLinkedList)malloc(sizeof(DListNode));
    if(!head)
        return -1;
    (*head)->next=*head;
    /*
使头结点的prior
指针和next
指针指向自己*/
    (*head)->prior=*head;
    return 1;
}

```

程序运行结果如图3-34所示。

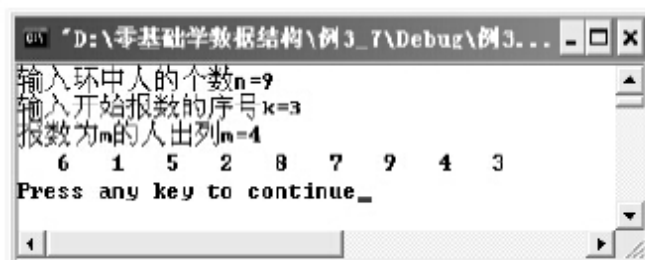


图3-34 约瑟夫问题程序运行结果

在创建双向循环链表CreateDCList函数中，根据创建的是否为第一个结点分为两种情况处理，如果是第一个结点，则让该结点的前驱结点指针域和后继结点指针域都指向该结点，并让头指针指向该结点，代码如下。

```
head=s;
s->prior=head;
s->next=head;
```

切记不要漏掉s->next=head或s->prior=head，否则在程序运行时会出现错误。

如果不是第一个结点，则将新结点插入双向链表的尾部，代码如下。

```
s->next=q->next;
q->next=s;
s->prior=q;
head->prior=s;
```

注意 语句s->next=q->next和q->next=s的顺序不能颠倒，另外不要忘记让头结点的prior域指向s。

3.6 静态链表

前面介绍的各种链表结点的分配与释放都是由函数malloc和free动态实现，因此称为动态链表。但是有的高级程序设计语言（如Basic、Fortran等）没有指针类型，就不能利用上述办法动态地创建链表，可以巧妙利用静态链表实现动态链表的功能。本节主要介绍静态链表的存储结构及实现。

3.6.1 静态链表的存储结构

可利用一维数组实现静态链表，其类型说明如下。

```
#define ListSize 100
typedef struct
{
    DataType data;
    int cur;
}SListNode;
typedef struct
{
    SListNode list[ListSize];
    int av;
}SLinkList;
```

在以上静态链表的类型定义中，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0分量可看成是头结点，其指针域指示链表的第一个结点。表中的最后一个结点的指针域为0，指向头结点，这样就构成一个静态循环链表。

SListNode是一个结点类型，SLinkList是一个静态链表类型，av是备用链表的指针，即av指向静态链表中一个未使用的位置。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。例如，线性表（Yang，Zheng，Feng，Xu，Wu，Wang，Geng）的静态链表存储情况如图3-35所示。

在数组中的编号	数据域	cur域
0		1
1	Yang	2
2	Zheng	3
3	Feng	4
4	Xu	5
5	Wu	6
6	Wang	7
7	Geng	0
8		
9		

图3-35 静态链表

设s为SlinkList类型的变量，则s[0].cur指示头结点，如果令i=s[0].cur，则s[i].data表示表中的第1个元素“Yang”，s[i].cur指示第2个元素在数组的位置。这与动态链表的操作类似，i=s[i].cur将指针向前移动，相当于p=p->next操作，游标cur代表指针next。

3.6.2 静态链表的基本运算

①初始化静态链表。只需要让静态链表的游标cur指向下一个结点，并将链表的最后一个结点的cur域置为0。初始化静态链表的算法实现如下。

```
void InitSList(SLinkList *L)
/*
静态链表的初始化*/
{
    int i;
    for(i=0;i<ListSize;i++)
        (*L).list[i].cur=i+1;
    (*L).list[ListSize-1].cur=0;
    (*L).av=1;
}
```

②分配结点。从备用链表中取出一个结点空间，分配给要插入链表中的元素，并返回要插入结点的位置。分配结点的算法实现如下。

```
int AssignNode(SLinkList L)
/*
从备用链表中取下一个结点空间，分配给要插入链表中的元素*/
{
    int i;
    i=L.av;
    L.av=L.list[i].cur;
    return i;
}
```

③回收结点。结点用过之后，要将空闲结点回收，以便其他结点使用，使其称为备用链表的空间。回收结点的算法实现如下。

```
void FreeNode(SLinkList L,int pos)
/*
将空闲结点插入备用链表*/
{
    L.list[pos].cur=L.av;
```

```

    L.av=pos;
}

```

④插入操作。在静态链表中第*i*个位置插入一个数据元素*e*。首先从备用链表中取出一个可用的结点，然后将其插入静态链表的第*i*个位置。

例如，在图3-36所示的静态链表中的第5个元素后插入元素“Hao”，首先从备用链表中取出一个空闲结点，即 $k=L.av$ ，图3-36中为数组编号为8的结点；再使备用链表指向下一个有用结点，修改备用链表指针，使其指向下一个结点，即 $L.av=L.list[k].cur$ ；然后把元素“Hao”放在新空闲结点中，即 $(*L).list[k].data=e$ ；再将新结点插入到对应的位置，这需要先找到前一个结点，让前一个结点的 cur 域指向新结点，然后让新结点的 cur 域指向下一个结点，图3-36中为 $L.list[5].cur=L.list[8].cur$ ， $L.list[8].cur=6$ 。插入过程如图3-36所示。

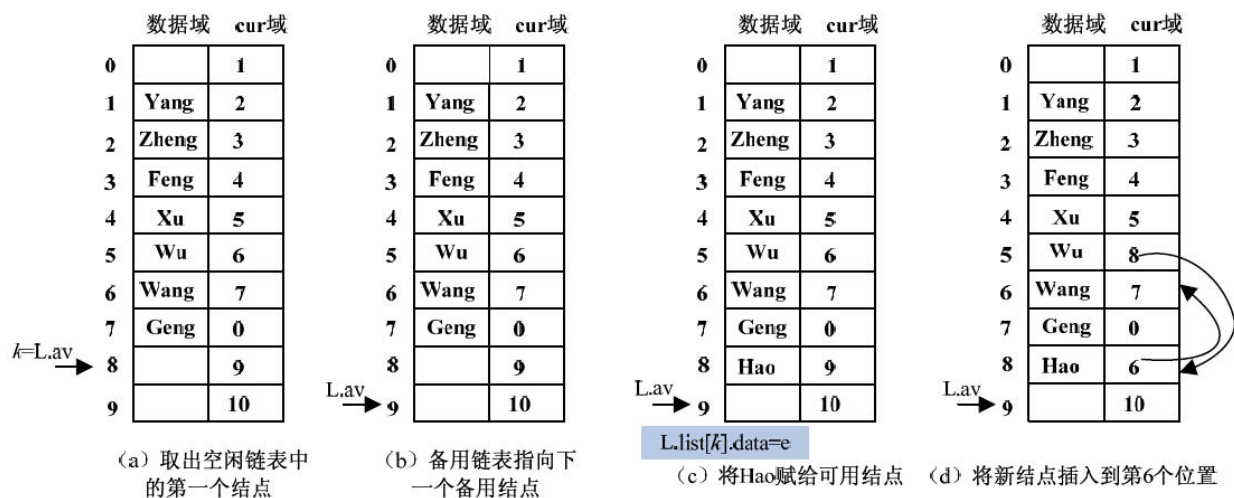


图3-36 在静态链表中插入元素后

插入操作的算法实现如下。

```
void InsertSList(SLinkList *L,int i,DataType e)
/*
插入操作*/
{
    int j,k,x;
    k=(*L).av;
    (*L).av=(*L).list[k].cur;
    (*L).list[k].data=e;
    j=(*L).list[0].cur;
    for(x=1;x<i-1;x++)
        j=(*L).list[j].cur;
    (*L).list[k].cur=(*L).list[j].cur;
    (*L).list[j].cur=k;
}
```

⑤删除操作。将静态链表中第*i*个位置的元素删除分为两种情况，如果删除的是第1个结点，则将头结点的cur域指向第2个结点，代码如下。

```
k=(*L).list[0].cur;
(*L).list[0].cur=(*L).list[k].cur;
```

如果删除的不是第1个结点，则先找到第*i*-1个元素的位置，修改cur域使其指向第*i*+1个元素，例如要删除图3-37所示的静态链表中的第3个元素，需要根据游标找到第2个元素，将其cur域修改为第4个元素的位置，即L.list[2].cur=L.list[3].cur。算法实现代码如下。

```
j=(*L).list[0].cur;
for(x=1;x<i-1;x++)
    j=(*L).list[j].cur;
k=(*L).list[j].cur;
(*L).list[j].cur=(*L).list[k].cur;
```

然后将被删除的结点空间放到备用链表中。

```
(*L).list[k].cur=(*L).av;  
(*L).av=k;
```

删除第3个结点的操作如图3-37所示。

在数组中的编号

数据域	cur域
	1
Yang	2
Zheng	4
Feng	4
Xu	5
Wu	8
Wang	7
Geng	0
Liu	6

图3-37 删除静态链表的第3个结点

删除操作的实现代码如下。

```
void DeleteSList(SLinkList *L,int i,DataType*e)  
/*  
删除操作*/  
{  
    int j,k,x;  
    if(i==1)  
    {  
        k=(*L).list[0].cur;  
        (*L).list[0].cur=(*L).list[k].cur;  
    }  
    else  
    {  
        j=(*L).list[0].cur;  
        for(x=1;x<i-1;x++)
```

```
        j=(*L).list[j].cur;
        k=(*L).list[j].cur;
        (*L).list[j].cur=(*L).list[k].cur;
    }
    (*L).list[k].cur=(*L).av;
    *e=(*L).list[k].data;
    (*L).av=k;
}
```

3.6.3 静态链表应用举例

【例3-8】 利用静态链表的基本操作创建静态链表，通过键盘输入要插入的元素及位置向静态链表中插入元素，通过输入删除元素的位置删除静态链表中的元素。例如，创建一个静态链表{20，15，-82，37，52，8，-90}，在静态链表的第6个位置插入元素85后，静态链表变为{20，15，-82，37，52，85，8，-90}，输入1，删除第1个元素，静态链表变为{15，-82，37，52，85，8，-90}。

【分析】 静态链表的插入和删除操作与单链表的操作类似，静态链表通过k=L.list[k].cur找到链表元素的下一个元素，插入和删除只需要修改静态链表的cur域。以上操作都可通过调用静态链表的基本操作实现，在插入元素的过程中，需要从备用链表中取出可用结点；在删除元素的过程中，需要注意将删除结点空间放入备用链表。程序的实现代码如下。

```
/*
包含头文件*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
/*
静态链表类型定义*/
typedef int DataType;
```

```

#define ListSize 100
typedef struct
{
    DataType data;
    int cur;
}SListNode;
typedef struct
{
    SListNode list[ListSize];
    int av;
}SLinkList;
#include "SLinkList.h"
/*
函数声明*/
void PrintDList(SLinkList L,int n);
void main()
{
    SLinkList L;
    int i,num;

    int pos;
    int e;
    DataType a[]={20,15,-82,37,52,8,-90};
    num=sizeof(a)/sizeof(a[0]);
    InitSList(&L);
    for(i=1;i<=num;i++)
        InsertSList(&L,i,a[i-1]);

    printf("
静态链表中的元素:");
    PrintDList(L,num);
    printf("
请输入要插入的元素及位置:");
    scanf("%d",&e);
    getchar();
    scanf("%d",&pos);
    getchar();
    InsertSList(&L,pos,e);
    printf("
插入元素后,
静态链表中的元素:");
    PrintDList(L,num+1);
    printf("
请输入要删除元素的位置:");
    scanf("%d",&pos);
    getchar();
    DeleteSList(&L,pos,&e);
    printf("
元素%d
已被删除。\\n",e);
    printf("
删除元素后,
静态链表中的元素:");
    PrintDList(L,num);
}
void PrintDList(SLinkList L,int n)
/*
输出静态链表中的所有元素*/
{
    int j,k;
    k=L.list[0].cur;
    for(j=1;j<=n;j++)
    {
        printf("%4d",L.list[k].data);
        k=L.list[k].cur;
    }
    printf("\\n");
}

```

程序运行结果如图3-38所示。

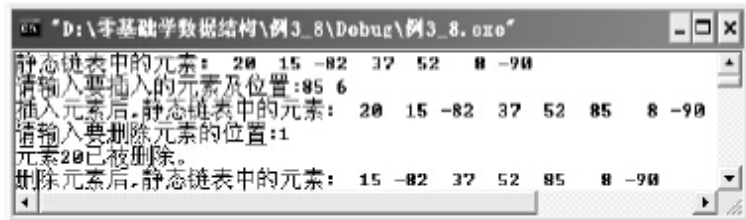


图3-38 静态链表插入与删除操作的程序运行结果

3.7 综合案例：一元多项式的表示与相乘

一元多项式的相乘是线性表在生活中一个实际应用，它涵盖了本节所学到的链表的各种操作。通过使用链表实现一元多项式的相乘，巩固大家对链表的理解与掌握。

3.7.1 一元多项式的表示

在数学中，一个一元多项式 $A_n(x)$ 可以写成降幂的形式，即 $A_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ ，如果 $a_n \neq 0$ ，则 $A_n(x)$ 称为 n 阶多项式。一个 n 阶多项式由 $n+1$ 个系数构成。一个 n 阶多项式的系数可以用线性表 $(a_n, a_{n-1}, \dots, a_1, a_0)$ 表示。

线性表的存储可以采用顺序存储结构，这样使多项式的一些操作变得更加简单。可以定义一个维数为 $n+1$ 的数组 $a[n+1]$ ， $a[n]$ 存放系数 a_n ， $a[n-1]$ 存放系数 a_{n-1} ， \dots ， $a[0]$ 存放系数 a_0 。但是，实际情况是可能多项式的阶数（最高的指数项）会很高，多项式的每个项的指数会差别很大，这可能会浪费很多的存储空间。例如一个多项式 $P(x) = 10x^{2001} + x + 1$ ，若采用顺序存储，则存放系数需要2002个存储空间，但是存储有用的数据只有3个。若只存储非零系数项，还必须存储相应的指数信息。

一元多项式 $A_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ 的系数和指数同时存放，可以表示成一个线性表，线性表的每一个数据元素由一个二

元组构成。因此，多项式 $A_n(x)$ 可以表示成线性表 $((a_n, n), (a_{n-1}, n-1), \dots, (a_1, 1), (a_0, 0))$ 。

多项式 $P(x)$ 可以表示成 $((10, 2001), (1, 1), (1, 0))$ 的形式。

因此，多项式可以采用链式存储方式表示，每一项可以表示成一个结点，结点的结构由存放系数的coef域、存放指数的expn域和指向下一个结点的next指针域3个域组成，如图3-39所示。

结点结构可以用C语言描述如下。

```
typedef struct polyn
{
    float coef;
    int expn;
    struct polyn *next;
}PloyNode, *PLinkList;
```

例如，多项式 $S(x) = 9x^8 + 5x^4 + 6x^2 + 7$ 可以表示成链表，如图3-40所示。

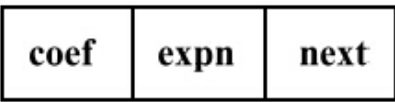


图3-39 多项式的结点结构

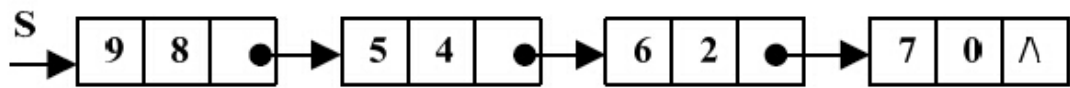


图3-40 一元多项式的链表表示

3.7.2 一元多项式相乘

两个一元多项式的相乘运算，需要将一个多项式的每一项的指数与另一个多项式的每一项的指数相加，并将其系数相乘。假设两个多项式 $A_n(x)$

$(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ 和 $B_m(x) = b_m x^m + b_{m-1} x^{m-1} + \dots + b_1 x + b_0$ ，要将这两个多项式相乘，就是将多项式 $A_n(x)$ 中的每一项与 $B_m(x)$ 相乘，相乘的结果用线性表表示为 $((a_n * b_m, n+m), (a_{n-1} * b_m, n+m-1), \dots, (a_1, 1), (a_0, 0))$ 。

例如，两个多项式 $A(x)$ 和 $B(x)$ 相乘后得到 $C(x)$ ， $A(x) = 5x^4 + 3x^2 + 3x$ ， $B(x) = 7x^3 + 5x^2 + 6x$ ， $C(x) = 35x^7 + 25x^6 + 51x^5 + 36x^4 + 33x^3 + 18x^2$ ，表示成链式存储结构如图3-41所示。

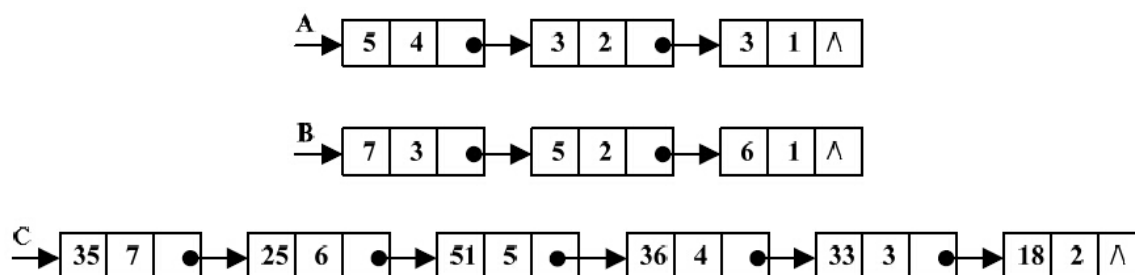


图3-41 多项式的链表表示

算法思想：设A、B和C分别是多项式 $A(x)$ 、 $B(x)$ 和 $C(x)$ 对应链表的头指针，要计算 $A(x)$ 和 $B(x)$ 的乘积，先计算出 $A(x)$ 和 $B(x)$ 的最高指数和，即 $4+3=7$ ，则 $A(x)$ 和 $B(x)$ 的乘积 $C(x)$ 的指数范围在 $0\sim 7$ 之间。然后将 $A(x)$ 的各项按照指数降幂排列，将 $B(x)$ 按照指数升幂排列，分别设两个指针pa和pb，pa用来指向链表A，pb用来指向链表B，从第一个结点开始计算两个链表的expn域的和，并将其与k比较（k为指数和的范围，从7到0递减），使链表的和呈递减排列。若和小于k，则 $pb=pb \rightarrow next$ ；若和等

于k，则求出两个多项式系数的乘积，并将其存入新结点中。若和大于k，则pa=pa->next。这样就可以得到多项式A(x)和B(x)的乘积C(x)。算法结束后重新将链表B逆置，将链表B恢复原样。

程序代码实现如下。

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
/*
一元多项式结点类型定义*/
typedef struct polyn
{
    float coef;          /*
存放一元多项式的系数*/
    int expn;             /*
存放一元多项式的指数*/
    struct polyn *next;
}PolyNode, *PLinkList;
void OutPut(PLinkList head)
/*
输出一元多项式*/
{
    PolyNode *p=head->next;
    while(p)
    {
        printf("%1.1f",p->coef);
        if (p->expn)
            printf("*x^d",p->expn);
        if (p->next&& p->next->coef>0)
            printf("+");
        p=p->next;
    }
}
void main()
{
    PLinkList A,B,C;
    A=CreatePolyn();
    printf("A(x)=");
    OutPut(A);
    printf("\n");
    B=CreatePolyn();
    printf("B(x)=");
    OutPut(B);
    printf("\n");
    C=MultiplyPolyn(A,B);
    printf("C(x)=A(x)*B(x)=");
    OutPut(C);
    /*
输出结果*/
    printf("\n");
}
PLinkList CreatePolyn()
/*
创建一元多项式，使一元多项式呈指数递减*/
{
    PolyNode *p,*q,*s;
    PolyNode *head=NULL;
    int expn2;
    float coef2;
    head=(PLinkList)malloc(sizeof(PolyNode));
    /*
动态生成一个头结点*/
```

```

        if(!head)
            return NULL;
        head->coef=0;
        head->expn=0;
        head->next=NULL;
        do
        {
            printf("
输入系数coef(
系数和指数都为0
结束)");

            scanf("%f",&coef2);
            printf("
输入指数exp(
系数和指数都为0
结束)");

            scanf("%d",&expn2);
            if((long)coef2==0&&expn2==0)
                break;
            s=(PolyNode*)malloc(sizeof(PolyNode));
            if(!s)
                return NULL;
            s->expn=expn2;
            s->coef=coef2;
            q=head->next;
            /*q
指向链表的第一个结点,即表尾*/
            p=head;
            /*p
指向q
的前驱结点*/
            while(q&&expn2<q->expn)
                /*
将新输入的指数与q
指向的结点指数比较*/
            {
                p=q;
                q=q->next;
            }
            if(q==NULL||expn2>q->expn)
                /*q
指向要插入结点的位置,p
指向要插入结点的前驱*/
            {
                p->next=s;
                /*
将s
结点插入到链表中*/
                s->next=q;
            }
            else
            {
                q->coef+=coef2;
                /*
如果指数与链表中结点指数相同,则将系数相加即可*/
            } while(1);
            return head;
        }
    }
    PolyNode *Reverse(PLinkList head)
    /*
    将生成的链表逆置,使一元多项式呈指数递增形式*/
    {
        PolyNode *q,*r,*p=NULL;
        q=head->next;
        while(q)
        {
            r=q->next;
            /*r
指向链表的待处理结点*/
            q->next=p;
            /*
将链表结点逆置*/
            p=q;
            /*p
指向刚逆置后链表结点*/
            q=r;
            /*q
指向下一准备逆置的结点*/
        }
        head->next=p;
        /*
将头结点的指针指向已经逆置后的链表*/
        return head;
    }
}

```

```

PolyNode *MultiplyPolyn(PLinkList A, PLinkList B)
/*
多项式的乘积*/
{
    PolyNode *pa, *pb, *pc, *u, *head;
    int k, maxExp;
    float coef;
    head = (PLinkList) malloc(sizeof(PolyNode)); /*
动态生成头结点*/
    if (!head)
        return NULL;
    head->coef = 0.0;
    head->expn = 0;
    head->next = NULL;
    if (A->next != NULL && B->next != NULL)
        maxExp = A->next->expn + B->next->expn; /*maxExp
为两个链表指数的和的最大值*/
    else
        return head;
    pc = head;
    B = Reverse(B); /*
使多项式B
(x
) 呈指数递增形式*/
    for (k = maxExp; k >= 0; k--) /*
多项式的乘积指数范围为0-maxExp*/
    {
        pa = A->next;
        while (pa != NULL && pa->expn > k) /*
找到pa
的位置*/
            pa = pa->next;
        pb = B->next;
        while (pb != NULL && pa != NULL && pa->expn + pb->expn < k) /*
如果和小于k
, 使pb
移到下一个结点*/
            pb = pb->next;
        coef = 0.0;
        while (pa != NULL && pb != NULL)
        {
            if (pa->expn + pb->expn == k) /*
如果在链表中找到对应的结点, 即和等于k
, 求相应的系数*/
            {
                coef += pa->coef * pb->coef;
                pa = pa->next;
                pb = pb->next;
            }
            else if (pa->expn + pb->expn > k) /*
如果和大于k
, 则使pa
移到下一个结点*/
                pa = pa->next;
            else
                pb = pb->next; /*
如果和小于k
, 则使pb
移到下一个结点*/
        }
        if (coef != 0.0) /*
如果系数不为0
, 则生成新结点, 并将系数和指数分别赋值给新结点. 并将结点插入链表中*/
        {
            u = (PolyNode*) malloc(sizeof(PolyNode));
            u->coef = coef;
            u->expn = k;
            u->next = pc->next;
            pc->next = u;
            pc = u;
        }
    }
}

```

```

    }
    B=Reverse(B);
    /*
完成多项式乘积后，将B
(x
) 呈指数递减形式*/
    return head;
}

```

程序运行结果如图3-42所示。

```

D:\零基础学数据结构\polylist\Debug\polylist.exe
输入系数coef<系数和指数都为0结束>5
输入指数exp<系数和指数都为0结束>4
输入系数coef<系数和指数都为0结束>3
输入指数exp<系数和指数都为0结束>2
输入系数coef<系数和指数都为0结束>3
输入指数exp<系数和指数都为0结束>1
输入系数coef<系数和指数都为0结束>0
输入指数exp<系数和指数都为0结束>0
A(x)=5.0*x^4+3.0*x^2+3.0*x^1
输入系数coef<系数和指数都为0结束>7
输入指数exp<系数和指数都为0结束>3
输入系数coef<系数和指数都为0结束>5
输入指数exp<系数和指数都为0结束>2
输入系数coef<系数和指数都为0结束>6
输入指数exp<系数和指数都为0结束>1
输入系数coef<系数和指数都为0结束>0
输入指数exp<系数和指数都为0结束>0
B(x)=7.0*x^3+5.0*x^2+6.0*x^1
G(x)=A(x)*B(x)=35.0*x^7+25.0*x^6+51.0*x^5+36.0*x^4+33.0*x^3+18.0*x^2
Press any key to continue.

```

图3-42 一元多项式的相乘程序运行结果

3.8 小结

线性表中的元素之间是一一对应的关系，除了第一个元素外，其他元素只有唯一的直接前驱，除了最后一个元素外，其他元素只有唯一的直接后继。

线性表有顺序存储和链式存储两种存储方式。采用顺序存储结构的线性表称为顺序表，采用链式存储结构的线性表称为链表。

顺序表中数据元素的逻辑顺序与物理顺序一致，因此可以随机存取。链表是靠指针域表示元素之间的逻辑关系。

链表又分为单链表和双向链表，这两种链表又可构成单循环链表、双向循环链表。单链表只有一个指针域，指针域指向直接后继结点。双向链表的一个指针域指向直接前驱结点，另一个指针域指向直接后继结点。

顺序表的优点是可以随机存取任意一个元素，算法实现较为简单，存储空间利用率高，缺点是需要预先分配存储空间，存储规模不好确定，插入和删除操作需要移动大量元素。链表的优点是不需要事先确定存储空间的大小，插入和删除元素不需要移动大量元素；缺点是只能从第一个结点开始顺序存取元素，存储单元利用率不高，算法

实现较为复杂，因涉及指针操作，操作不当，会产生无法预料的内存错误。

3.9 习题

一、单选题

1. 对线性表，在下列哪种情况下应当采用链表表示？（）
 - A. 经常需要随机地存取元素
 - B. 经常需要进行插入和删除操作
 - C. 表中元素需要占据一片连续的存储空间
 - D. 表中元素的个数不变

2. 若长度为 n 的线性表采用顺序存储结构，在其第 i 个位置插入一个新元素算法的时间复杂度为（）。
 - A. $O(\log_2 n)$
 - B. $O(1)$
 - C. $O(n)$
 - D. $O(n^2)$

3. 若一个线性表中最常用的操作是取第*i*个元素和找第*i*个元素的前趋元素，则采用（ ）存储方式最节省时间。

A. 顺序表

B. 单链表

C. 双链表

D. 单循环链表

4. 在一个长度为*n*的顺序表中，在第*i*个元素之前插入一个新元素时，需向后移动（ ）个元素。

A. $n-i$

B. $n-i+1$

C. $n-i-1$

D. i

5. 非空的循环单链表head的尾结点p满足（ ）。

A. $p \rightarrow next == head$

B. $p \rightarrow next == NULL$

C. $p == \text{NULL}$

D. $p == \text{head}$

6. 在双向循环链表中，在p指针所指的结点后插入一个指针q所指向的新结点，修改指针的操作是（ ）。

A. $p \rightarrow \text{next} = q$; $q \rightarrow \text{prior} = p$; $p \rightarrow \text{next} \rightarrow \text{prior} = q$; $q \rightarrow \text{next} = q$;

B. $p \rightarrow \text{next} = q$; $p \rightarrow \text{next} \rightarrow \text{prior} = q$; $q \rightarrow \text{prior} = p$; $q \rightarrow \text{next} = p \rightarrow \text{next}$;

C. $q \rightarrow \text{prior} = p$; $q \rightarrow \text{next} = p \rightarrow \text{next}$; $p \rightarrow \text{next} \rightarrow \text{prior} = q$; $p \rightarrow \text{next} = q$;

D. $q \rightarrow \text{next} = p \rightarrow \text{next}$; $q \rightarrow \text{prior} = p$; $p \rightarrow \text{next} = q$; $p \rightarrow \text{next} = q$;

7. 线性表采用链式存储时，结点的存储地址（ ）。

A. 必须是连续的

B. 必须是不连续的

C. 连续与否均可

D. 和头结点的存储地址相连续

8. 在一个长度为 n 的顺序表中删除第 i 个元素，需要向前移动（）个元素。

A. $n-i$

B. $n-i+1$

C. $n-i-1$

D. $i+1$

9. 从表中任一结点出发，都能扫描整个表的是（）。

A. 单链表

B. 顺序表

C. 循环链表

D. 静态链表

10. 在具有 n 个结点的单链表上查找值为 x 的元素时，其时间复杂度为（）。

A. $O(n)$

B. $O(1)$

C. $O(n^2)$

D. $O(n-1)$

11. 一个顺序表的第一个元素的存储地址是90，每个元素的长度为2，则第6个元素的存储地址是（ ）。

A. 98

B. 100

C. 102

D. 106

12. 在一个单链表中，若删除p所指向结点的后续结点，则执行（ ）。

A. $p \rightarrow next = p \rightarrow next \rightarrow next;$

B. $p = p \rightarrow next;$ $p \rightarrow next = p \rightarrow next \rightarrow next;$

C. $p = p \rightarrow next;$

D. $p = p \rightarrow next \rightarrow next;$

13. 已知指针p和q分别指向某单链表中第一个结点和最后一个结点。假设指针s指向另一个单链表中某个结点，则在s所指结点之后插入上述链表应执行的语句为（ ）。

A. $q \rightarrow next = s \rightarrow next; s \rightarrow next = p;$

B. $s \rightarrow next = p; q \rightarrow next = s \rightarrow next$

C. $p \rightarrow next = s \rightarrow next; s \rightarrow next = q;$

D. $s \rightarrow next = q; p \rightarrow next = s \rightarrow next;$

14. 在单链表中，指针p指向元素为x的结点，实现删除x的后继的语句是（ ）。

A. $p = p \rightarrow next;$

B. $p \rightarrow next = p \rightarrow next \rightarrow next;$

C. $p \rightarrow next = p;$

D. $p = p \rightarrow next \rightarrow next;$

二、算法分析题

1. 函数实现单链表的插入算法，请在空格处将算法补充完整。

```
int ListInsert(LinkList L,int i,ElemType e)
{
    LNode *p,*s;int j;
    p=L;j=0;
    while((p!=NULL)&&(j<i-1))
    {
        p=p->next;j++;
    }
    if(p==NULL||j>i-1) return ERROR;
    s=(LNode *)malloc(sizeof(LNode));
    s->data=e;
    (1)          ;
    (2)          ;
    return OK;
}
```

2. 函数ListDelete_sq实现顺序表删除算法，请在空格处将算法补充完整。

```
int ListDelete_sq(SqList *L,int i)
{
    int k;
    if(i<1||i>L->length) return ERROR;
    for(k=i-1;k<L->length-1;k++)
        L->slist[k]=
    (1
    )          ;
    (2
    )          ;
    return OK;
}
```

3. 写出算法的功能。

```
int L(head){
    node * head;
    int n=0;
    node *p;
    p=head;
    while(p!=NULL)
    { p=p->next;
      n++;
    }
    return(n);
}
```

三、算法设计题

1. 编写算法，实现带头结点单链表的逆置算法。
2. 已知有两个带头结点的单链表A和B，A和B中的元素由小到大排列，设计一个算法，求A和B的交集C，将A和B中相同的元素插入C中。
3. 顺序表A和顺序表B的元素都是非递减排列，利用线性表的基本运算，将它们合并成一个顺序表C，要求C也是非递减排列。例如， $A = (6, 11, 11, 23)$ ， $B = (2, 10, 12, 12, 21)$ ，则 $C = (2, 6, 10, 11, 11, 12, 12, 21, 23)$ 。
4. 已知有两个顺序表A和B，A中的元素按照递增排列，B中的元素按照递减排列。试编写一个算法，将A和B合并成一个顺序表，使其按照递增有序排列，要求不占用额外的存储单元。
5. 利用单链表的基本运算，实现如果在单链表A中出现的元素，在单链表B中也出现，则将A中该元素删除。

分析 如果把单链表看成是集合，这其实是求两个集合的差集A-B，即所有属于集合A而不属于集合B的元素。具体实现是，对于单链表A中的每个元素e，在单链表B中进行查找，如果在B中存在与A相同的元素，则将元素从A中删除。

6. 将单链表A和B合并得到C，C中的元素仍按照非递减排列。

7. 已知有两个带头结点的双向循环链表A和B，它们的元素均是按照递增排列，编写算法，将A和B合并成一个双向循环链表，并使合并后链表中的元素也按照递增排列。

分析 可以使用原有结点空间而不建立新结点合并双向循环链表。首先以A的头结点建立新的空双向链表；分别用指针p和q指向链表A和B的第一个结点，依次比较p和q指示的结点元素大小，取下较小的结点作为新链表的结点，插入到新链表的表尾，重复这一过程一直到A和B中有一个链表为空；如果A和B中有一个链表为空而另一个链表不为空时，将不空的链表剩下的部分插入新链表的表尾。

8. 设A和B是两个顺序表，其元素按从小到大的顺序排列。编写一个算法，将A和B中相同元素组成一个新的从大到小的有序顺序表C，并分析算法的时间复杂度。

第4章 栈

栈是一种操作受限的线性表。栈具有线性表的结构特点，即每一个元素只有一个前驱元素和后继元素（除了第一个元素和最后一个元素外），但它只允许在表的一端进行插入和删除操作。与线性表一样，栈也有两种存储结构，即顺序存储结构和链式存储结构。在实际生活中，栈的应用十分广泛，在表达式求值、括号匹配常常用到栈的设计思想。

本章重点和难点：

- 栈的顺序表示与算法实现
- 栈的链式表示与算法实现
- 求算术表达式的值
- 迷宫求解问题
- 递归的消除

4.1 栈的定义与抽象数据类型

4.1.1 什么是栈

栈（stack）也称为堆栈，它是限定仅在表尾进行插入和删除操作的线性表。对栈来说，表尾（允许操作的一端）称为**栈顶**

（top），另一端称为**栈底**（bottom）。栈顶是动态变化的，它由一个称为栈顶指针（top）的变量指示。当表中没有元素时，称为**空栈**。

栈的插入操作称为入栈或进栈，删除操作称为出栈或退栈。

在栈 $S = (a_1, a_2, \dots, a_n)$ 中， a_1 称为栈底元素， a_n 称为栈顶元素，由栈顶指针top指示。栈中的元素按照 a_1, a_2, \dots, a_n 的顺序进栈，当前的栈顶元素为 a_n 。如图4-1所示。最先进栈的元素一定是栈底元素，最后进栈的元素一定是栈顶元素。每次删除的元素是栈顶元素，也就是最后进栈的元素。因此，栈是一种后进先出（Last In First Out, LIFO）的线性表。

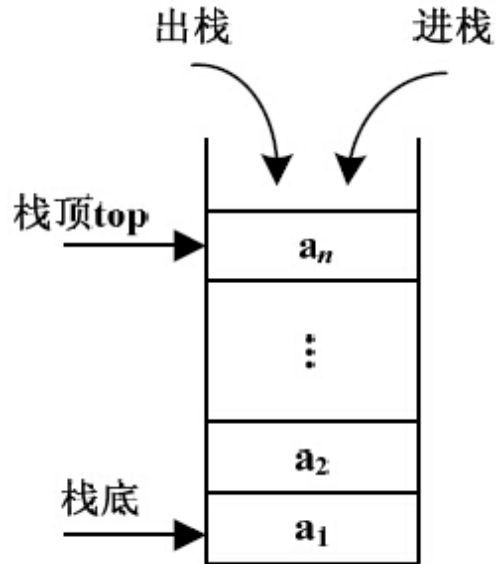


图4-1 栈

在软件应用中，栈的后进先出特性应用非常普遍，例如，使用浏览器上网时，浏览器的左上角有一个“后退”按钮，单击后可以按访问顺序的逆序加载浏览过的网页。

把栈想象成一个桶，先放进去的东西在最下面，后放进去的东西在最上面，最先取出来的是最后放进去的，最后取出来的是最先放进去的。这也像在日常生活中有一摞盘子，放盘子时，一个一个往上堆放，取盘子时，只能从上往下取，最后放上的盘子最先取下来，最先放的盘子最后取下来。

若将元素a、b、c和d依次入栈，最后将栈顶元素出栈，栈顶指针top的变化情况如图4-2所示。

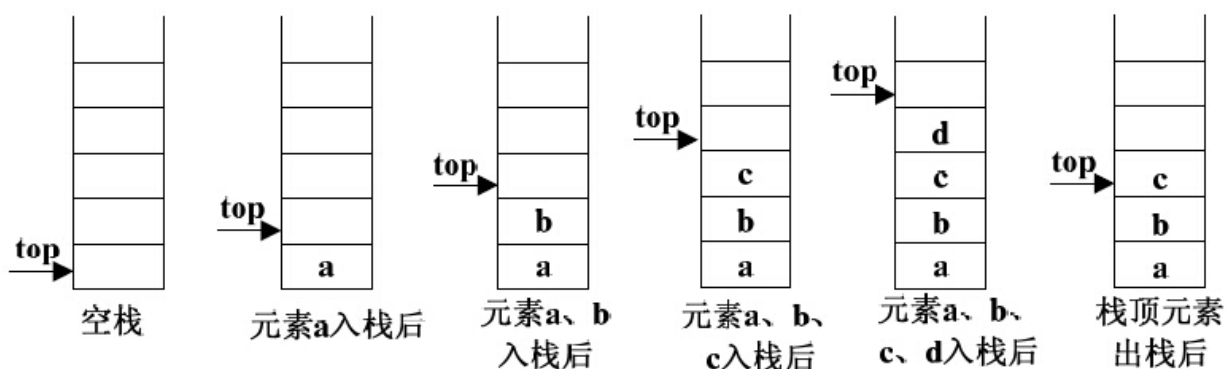


图4-2 栈的插入和删除过程

若abc为一个进栈序列，则出栈序列有5种可能，即abc、acb、bac、bca和cba、但cab是不可能的输出序列。这是因为c先出栈，意味着c曾进栈，既然c最先出栈，那么说明a和b已经进栈了，此时b一定在a的上面，那么出栈序列一定是cba，不可能是cab。这个题目主要考查对栈的后进先出特性的掌握情况，遇到类似问题，可以把这3个元素依次放入栈中，一个一个去尝试，就会知道哪个是不能的序列，哪个是可能的序列。

4.1.2 栈的抽象数据类型

1. 数据对象集合

栈的数据对象集合为 $\{a_1, a_2, \dots, a_n\}$ ，每个元素都有相同的类型DataType。

栈中数据元素之间是一一对应的关系。栈具有线性表的特点，除了第一个元素 a_1 外，每一个元素有且只有一个直接前驱元素；除了最后一个元素 a_n 外，每一个元素有且只有一个直接后继元素。

2. 基本操作集合

- `InitStack (&S)`：初始化操作，建立一个空栈S。这就像日常生活中，准备好了一个箱子，准备往里面摆盘子。

- `StackEmpty (S)`：若栈S为空，返回1，否则返回0。栈空就像日常生活中，准备好了箱子，箱子还是空的，里面没有盘子；栈不空，说明箱子里已经有了盘子。

- `GetTop (S, &e)`：返回栈S的栈顶元素给e。栈顶元素就像箱子里面最上面的那个盘子。

- `PushStack (&S, e)`：在栈S中插入元素e，使其成为新的栈顶元素。这就像日常生活中，在箱子里新放入了一个盘子，这个盘子成为一摞盘子中最上面的一个。

- `PopStack (&S, &e)`：删除栈S的栈顶元素，并用e返回其值。这就像是把箱子里的最上面的那个盘子取出来。

- StackLength (S) : 返回栈S的元素个数。这就像放在箱子里的盘子总共有多少个。

- ClearStack (S) : 清空栈S。这就像把箱子里的盘子全部取出来。

4.2 栈的顺序表示与实现

栈有两种存储结构，即顺序存储和链式存储。本节主要介绍栈的顺序存储结构及其操作实现。

4.2.1 栈的顺序存储结构

采用顺序存储结构的栈称为**顺序栈**。顺序栈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，可利用C语言中的数组作为顺序栈的存储结构，同时附设一个栈顶指针top，用于指向顺序栈的栈顶元素。当top=0时表示空栈。

栈的顺序存储结构类型描述如下。

```
#define StackSize 100
typedef struct
{
    DataType stack[StackSize];
    int top;
}SeqStack;
```

其中，DataType为元素的数据类型，stack用于存储栈中的数据元素的数组，top为栈顶指针。

当栈中元素已经有StackSize个时，称为栈满。如果继续进栈操作则会产生溢出，称为**上溢**。对空栈进行删除操作，称为**下溢**。

顺序栈的结构如图4-3所示。元素a、b、c、d、e、f、g、h依次进栈后，a为栈底元素，h为栈顶元素。在实际操作中，栈顶指针指向栈顶元素的下一个位置。



图4-3 顺序栈结构

顺序栈涉及的一些基本操作如下。

①初始化栈，将栈顶指针置为0，即令S.top=0。

②判断栈空条件为S.top==0，栈满条件为S.top==StackSize-1。

③栈的长度（即栈中元素个数）为S.top。

④进栈操作，先判断栈是否已满，若未满，将元素压入栈中，即S.stack[S.top]=e，然后使栈顶指针加1，即S.top++。出栈操作，先判断栈是否为空，若非空，使栈顶指针减1，即S.top--，然后元素出栈，即e=S.stack[S.top]。

4.2.2 顺序栈的基本运算

顺序栈的基本运算如下（以下算法的实现保存在文件“SeqStack.h”中）。

①初始化栈，代码如下。

```
void InitStack(SeqStack *S)
/*
初始化栈*/
{
    S->top=0;          /*
把栈顶指针置为0*/
}
```

②判断栈是否为空，代码如下。

```
int StackEmpty(SeqStack S)
/*
判断栈是否为空，栈为空返回1
.....*/
```

```

, 否则返回0*/
{
    if(S.top==0)                /*
如果栈顶指针top
为0*/
        return 1;                /*
返回1*/
    else                        /*
否则*/
        return 0;                /*
返回0*/
}

```

③取栈顶元素。在取栈顶元素前，先判断栈是否为空，如果栈为空，则返回0表示取栈顶元素失败；否则，将栈顶元素赋值给e，并返回1表示取栈顶元素成功。取栈顶元素的算法实现如下。

```

int GetTop(SeqStack S, DataType *e)
/*
取栈顶元素。将栈顶元素值返回给e
, 返回1
表示成功, 返回0
表示失败。*/
{
    if(S.top<=0)                /*
如果栈为空*/
    {
        printf("
栈已经空!\n");
        return 0;
    }
    else                        /*
否则*/
    {
        *e=S.stack[S.top-1];    /*
在取栈顶元素*/
        return 1;
    }
}

```

④将元素e入栈。在将元素e进栈前，需要先判断栈是否已满，如果栈满，返回0表示进栈操作失败；否则将元素e压入栈中，然后将栈顶指针top增1，并返回1表示进栈操作成功。进栈操作的算法实现如下。

```

int PushStack(SeqStack *S,DataType e)
/*
将元素e
进栈, 元素进栈成功返回1
, 否则返回0.*/
{
    if(S->top>=StackSize)        /*
如果栈已满*/
    {
        printf("
栈已满, 不能将元素进栈! \n");
        return 0;
    }
    else                        /*
.....

```

```

否则*/
{
    S->stack[S->top]=e;          /*
元素e
进栈*/
    S->top++;                    /*
修改栈顶指针*/
    return 1;
}

```

⑤将栈顶元素出栈。在将元素出栈前，需要先判断栈是否为空。如果栈为空，则返回0；如果栈不为空，则先使栈顶指针减1，然后将栈顶元素值赋值给e，返回1，表示出栈成功。出栈操作的算法实现如下。

```

int PopStack(SeqStack *S,DataType *e)
/*
出栈操作。将栈顶元素出栈，并将其赋值给e
。出栈成功返回1
，否则返回0*/
{
    if(S->top==0)                /*
如果栈为空*/
    {
        printf("
栈中已经没有元素，不能进行出栈操作!\n");
        return 0;
    }
    else                          /*
否则*/
    {
        S->top--;                /*
先修改栈顶指针，即出栈*/
        *e=S->stack[S->top];      /*
将出栈元素赋给e*/
        return 1;
    }
}

```

⑥求栈的长度，代码如下。

```

int StackLength(SeqStack S)
/*
求栈的长度*/
{
    return S.top;
}

```

⑦清空栈，代码如下。

```

void ClearStack(SeqStack *S)
/*
清空栈*/
{
    S->top=0;                    /*
... ..

```

```
将栈顶指针置为0*/  
}
```

4.2.3 顺序栈应用举例

【例4-1】 利用顺序栈的基本操作，将元素A、B、C、D、E、F依次进栈，然后将F和E出栈，再将G和H进栈，最后将元素全部出栈，并依次输出出栈元素。

【分析】 主要考查栈的基本操作和栈的后进先出特性，实现代码如下。

```
/*  
包含头文件*/  
#include<stdio.h>  
#include<stdlib.h>  
/*  
类型定义*/  
typedef char DataType;  
#include "SeqStack.h" /*  
包含栈的基本操作实现*/  
void main()  
{  
    SeqStack S; /*  
    定义一个栈*/  
        int i;  
        DataType a[]={ 'A', 'B', 'C', 'D', 'E', 'F' };  
        DataType e;  
    InitStack(&S); /*  
    初始化栈*/  
    for(i=0;i<sizeof(a)/sizeof(a[0]);i++) /*  
    将数组a  
    中元素依次进栈*/  
    {  
        if(PushStack(&S,a[i])==0)  
        {  
            printf("  
栈已满，不能进栈！");  
            return;  
        }  
    }  
    printf("  
依次出栈的元素是：");  
    if(PopStack(&S,&e)==1) /*  
    元素F  
    出栈*/  
        printf("%4c",e);  
    if(PopStack(&S,&e)==1) /*  
    元素E  
    出栈*/  
        printf("%4c",e);  
    printf("\n");  
    printf("  
当前的栈顶元素是：");  
    if(GetTop(S,&e)==0) /*  
    取栈顶元素*/  
    {  
        printf("  
栈已空！");  
        return;  
    }  
    else  
        printf("%4c\n",e);  
    printf("  
将元素G
```

```

、H
依次入栈。\\n");
    if(PushStack(&S, 'G')==0)                                /*
元素G
进栈*/
    {
        printf("
栈已满，不能进栈!");
        return;
    }
    if(PushStack(&S, 'H')==0)                                /*
元素H
进栈*/
    {
        printf("
栈已满，不能进栈!");
        return;
    }
    printf("
当前栈中的元素个数是: %d\\n", StackLength(S));            /*
输出栈中元素个数*/
    printf("
将栈中元素出栈，出栈的序列是: \\n");
    while(!StackEmpty(S))                                    /*
如果栈不空，将所有元素出栈*/
    {
        PopStack(&S, &e);
        printf("%4c", e);
    }
    printf("\\n");
}

```

程序的运行结果如图4-4所示。

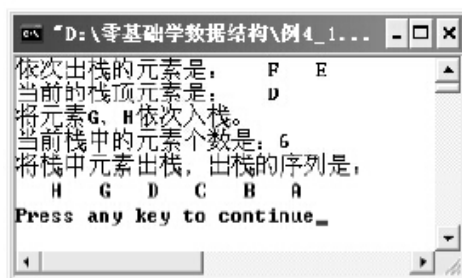


图4-4 栈的基本操作运行结果

【例4-2】 两个栈S1和S2都采用顺序结构存储，并且共享一个存储区。为了尽可能利用存储空间，减少溢出的可能，采用栈顶相向、迎面增长的方式，试设计S1和S2有关入栈和出栈的算法。

【分析】 该题是哈尔滨工业大学的考研试题，主要考查共享栈的算法设计。在设计共享栈时，应注意两个栈的栈顶指针变化和栈满、栈空条件。

1. 什么是栈的共享

在使用顺序栈时，因为栈空间的大小难以准确估计，可能会出现有的栈还有空闲空间。为了能充分利用栈的空间，可以让多个栈共享一个足够大的连续存储空间，通过利用栈顶指针能灵活移动的特性，使多个栈存储空间互相补充，存储空间得到有效利用，这就是**栈的共享**。

最常见的是两个栈的共享。栈的共享原理是利用栈底固定，栈顶迎面增长的方式。可通过两个栈共享一个一维数组实现，两个栈的栈底设置在数组的两端，当有元素进栈时，栈顶位置从栈的两端迎面增长，当两个栈的栈顶相遇时，栈满。

共享栈的数据结构类型描述如下。

```
typedef struct
{
    DataType stack[StackSize];
    int top[2];
}SSeqStack;
```

其中，top[0]和top[1]分别是两个栈的栈顶指针。

用一维数组表示的共享栈如图4-5所示。

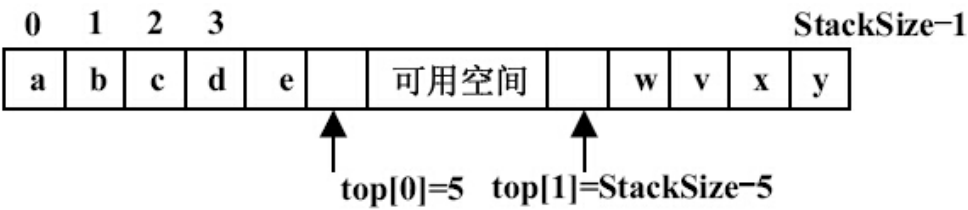


图4-5 共享栈

2. 共享栈的基本运算

下面是共享栈的基本运算（以下算法保存在文件“SSeqStack.h”中）。

①初始化栈，代码如下。

```
void InitStack(SSeqStack *S)
/*
共享栈的初始化*/
{
    S->top[0]=0;
    S->top[1]=StackSize-1;
}
```

②取栈顶元素。首先判断要取哪个栈的栈顶元素，接着还要判断栈是否为空，如果栈为空，则返回0表示取栈顶元素失败；如果栈不为空，则将栈顶元素返回给e，并返回1表示取栈顶元素成功。取栈顶元素的算法实现如下。

```
int GetTop(SSeqStack S, DataType *e,int flag)
/*
取栈顶元素。将栈顶元素值返回给e
，并返回1
表示成功；否则返回0
表示失败。*/
{
    switch(flag)
    {
        case 1:          /*
为1
，表示要取左端栈的栈顶元素*/
            if (S.top[0]==0)
                return 0;
            *e=S.stack[S.top[0]-1];
            break;
        case 2:          /*
为2
，表示要取右端栈的栈顶元素*/
            if (S.top[1]==StackSize-1)
                return 0;
            *e=S.stack[S.top[1]+1];
            break;
        default:
            return 0;
    }
    return 1;
}
```

③将元素e入栈。在将元素入栈之前，需要先判断栈是否已满，如果栈已满，则返回0表示进栈操作失败；否则先通过标志变量flag判断哪个栈需要进栈操作，然后将元素e进栈，并修改栈顶指针，最后返回1表示进栈操作成功。将元素e入栈的算法实现如下。

```
int PushStack(SSeqStack *S,DataType e,int flag)
/*
将元素e
入共享栈。进栈成功返回1
.....
```

```

, 否则返回0*/
{
    if (S->top[0]==S->top[1])          /*
如果共享栈已满*/
        return 0;                      /*
返回0
, 进栈失败*/
    switch(flag)
    {
        case 1:                        /*
当flag
为1
, 表示将元素进左端的栈*/
            S->stack[S->top[0]]=e;      /*
元素进栈*/
            S->top[0]++;                /*
修改栈顶指针*/
            break;
        case 2:                        /*
当flag
为2
, 表示将元素要进右端的栈*/
            S->stack[S->top[1]]=e;      /*
元素进栈*/
            S->top[1]--;                /*
修改栈顶指针*/
            break;
        default:
            return 0;
    }
    return 1;                          /*
返回1
, 进栈成功*/
}

```

④将栈顶元素出栈。在将栈顶元素出栈之前，首先根据标志flag判断要将哪个栈的栈顶元素出栈，如果flag为1，则需将左端栈的栈顶元素出栈，令栈顶指针减1，并把栈顶元素赋给e；如果flag为2，则需将右端栈的栈顶元素出栈，令栈顶指针加1，并把栈顶元素赋给e。最后返回1表示出栈操作成功。如果栈为空，返回0表示出栈操作失败。将栈顶元素出栈的算法实现如下所示。

```

int PopStack(SSeqStack *S,DataType *e,int flag)
{
    switch(flag)                        /*
在出栈操作之前，判断哪个栈要进行出栈操作*/
    {
        case 1:                        /*
为1
, 表示左端的栈需要出栈操作*/
            if (S->top[0]==0)           /*
左端的栈为空*/
                return 0;              /*
返回0
, 出栈操作失败*/
            S->top[0]--;                 /*
修改栈顶指针，元素出栈操作*/
            *e=S->stack[S->top[0]];      /*
将出栈的元素赋给e*/
            break;
        case 2:                        /*
为2
, 表示右端的栈需要出栈操作*/
            if (S->top[1]==StackSize-1) /*
右端的栈为空*/

```

```

        return 0;                                /*
返回0
, 出栈操作失败*/
        S->top[1]++;                                /*
修改栈顶指针, 元素出栈操作*/
        *e=S->stack[S->top[1]];                    /*
将出栈的元素赋给e*/
        break;
    default:
        return 0;
    }
    return 1;                                    /*
返回1
, 出栈操作成功*/
}

```

⑤判断栈是否为空。先根据标志flag判断哪个栈为空, 如果栈为空, 则返回1表示栈为空; 否则返回0表示栈不为空。判断共享栈是否为空的算法实现如下。

```

int StackEmpty(SSeqStack S,int flag)
/*
判断栈是否为空。如果栈为空, 返回1
; 否则, 返回0
。*/
{
    switch(flag)
    {
        case 1:                                    /*
为1
, 表示判断左端的栈是否为空*/
            if(S.top[0]==0)
                return 1;
            break;
        case 2:                                    /*
为2
, 表示判断右端的栈是否为空*/
            if(S.top[1]==StackSize-1)
                return 1;
            break;
        default:
            printf("
输入的flag
参数错误!");
            return -1;
    }
    return 0;
}

```

3. 测试代码

测试代码如下。

```

/*
包含头文件*/
#include<stdio.h>
#include<stdlib.h>
#define StackSize 100
typedef int DataType;
#include "SSeqStack.h" /*
包含共享栈的基本类型定义和基本操作实现*/

```

```

void main()
{
    SSeqStack S;                                /*
    定义一个共享栈*/
        int i;
        DataType a[]={10,20,30,40,50,60};
        DataType b[]={100,200,300,500};
        DataType e1,e2;
    InitStack(&S);                                /*
    初始化共享栈*/
    for(i=0;i<sizeof(a)/sizeof(a[0]);i++)        /*
    将数组a
    中元素依次进左端栈*/
    {
        if(PushStack(&S,a[i],1)==0)
        {
            printf("
            栈已满，不能进栈！");
            return;
        }
        for(i=0;i<sizeof(b)/sizeof(b[0]);i++)    /*
    将数组b
    中元素依次进右端栈*/
        {
            if(PushStack(&S,b[i],2)==0)
            {
                printf("
                栈已满，不能进栈！");
                return;
            }
            if(GetTop(S,&e1,1)==0)
            {
                printf("
                栈已空");
                return;
            }
            if(GetTop(S,&e2,2)==0)
            {
                printf("
                栈已空");
                return;
            }
            printf("
            左端栈的栈顶元素是：%d
            ，右端栈的栈顶元素是：%d\n",e1,e2);
            printf("
            左端栈的出栈的元素次序是：");
            while(!StackEmpty(S,1)) /*
            将左端栈元素出栈*/
            {
                PopStack(&S,&e1,1);
                printf("%5d",e1);
            }
            printf("\n");
            printf("
            右端栈的出栈的元素次序是：");
            while(!StackEmpty(S,2)) /*
            将右端栈元素出栈*/
            {
                PopStack(&S,&e2,2);
                printf("%5d",e2);
            }
            printf("\n");
        }
    }
}

```

程序运行结果如图4-6所示。



图4-6 共享栈基本操作运行结果

因为采用数组作为共享栈的存储结构, 所以栈顶指针的变化刚好相反。当左端栈进行入栈操作时, 栈顶指针需要 $\text{top}++$, 当右端栈进行入栈操作时, 栈顶指针需要 $\text{top}--$ 。当左端栈进行出栈操作时, 栈顶指针需要 $\text{top}--$, 当右端栈进行出栈操作时, 栈顶指针需要 $\text{top}++$ 。左端栈的判空条件是 $\text{S.top}[0]==0$, 右端栈的判空条件是 $\text{S.top}[1]==\text{StackSize}-1$ 。共享栈满的判断条件是 $\text{S}\rightarrow\text{top}[0]==\text{S}\rightarrow\text{top}[1]$ 。

4.3 栈的链式表示与实现

在顺序栈中，由于顺序存储结构需要事先静态分配，而存储规模往往又难以确定，如果栈空间分配过小，可能会造成溢出；如果栈空间分配过大，又造成存储空间浪费。因此，为了克服顺序存储的缺点，采用链式存储结构表示栈。本节主要介绍栈的存储结构及链栈的基本运算。

4.3.1 栈的链式存储结构

栈的链式存储结构是用一组不一定连续的存储单元来存放栈中数据元素的。一般来说，当栈中数据元素的数目变化较大或不确定时，使用链式存储结构作为栈的存储结构是比较合适的。人们将用链式存储结构表示的栈称为链栈 或链式栈 。

链栈通常用单链表表示。插入和删除操作都在栈顶指针的位置进行，这一端称为栈顶 ，通常由栈顶指针top指示。为了操作方便，通常在链栈中设置一个头结点，用栈顶指针top指向头结点，头结点的指针指向链栈的第一个结点。例如，元素a、b、c、d依次入栈的链栈如图4-7所示。

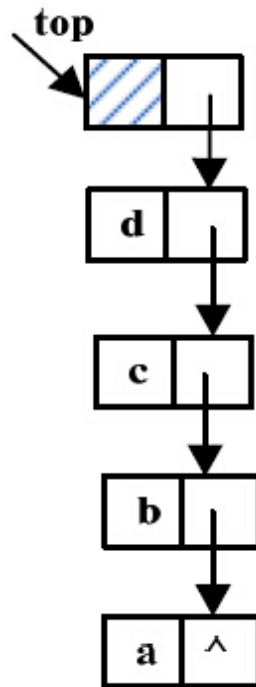


图4-7 带头结点的链栈

栈顶指针top始终指向头结点，最先入栈的元素在链栈的栈底，最后入栈的元素成为栈顶元素。由于链栈的操作都是在链表的表头位置进行，因而链栈的基本操作的时间复杂度均为 $O(1)$ 。

链栈的结点类型描述如下。

```
typedef struct node
{
    DataType data;
    struct node *next;
} LStackNode, *LinkStack;
```

对于带头结点的链栈，初始化链栈时，有`top->next=NULL`，判断栈空的条件为`top->next==NULL`。对于不带头结点的链栈，初始化链栈时，有`top=NULL`，判断栈空的条件为`top==NULL`。

采用链式存储的栈不必事先估算栈的最大容量，只要系统有可用的空间，就能随时为结点申请空间，不存在栈满的问题。但在用完链栈后，应释放其空间。

4.3.2 链栈的基本运算

链栈的基本运算实现如下（以下算法的实现保存在文件“LinkStack.h”中）。

①初始化链栈。初始化链栈需要先为头结点分配存储单元，然后将头结点的指针域置为空。初始化链栈的算法实现如下。

```
void InitStack(LinkStack *top)
/*
链栈的初始化*/
{
    if ((*top=(LinkStack)malloc(sizeof(LStackNode)))==NULL)    /*
为头结点分配一个存储空间*/
        exit(-1);
    (*top)->next=NULL;    /*
将链栈的头结点指针域置为空*/
}
```

②判断链栈是否为空。如果头结点指针域为空，说明链栈为空，返回1；否则返回0。判断链栈是否为空的算法实现如下。

```
int StackEmpty(LinkStack top)
/*
判断链栈是否为空*/
{
    if (top->next==NULL)    /*
如果头结点的指针域为空*/
        return 1;    /*
返回1*/
    else    /*
否则*/
        return 0;    /*
--
```

```
    返回0*/  
}
```

③将元素e入栈。先动态生成一个结点，用p指向该结点，将元素e值赋给*p结点的数据域，然后将新结点插入链表的第一个结点之前。把新结点插入链表中分为两个步骤，第一步p->next=top->next；第二步top->next=p。进栈操作如图4-8所示。

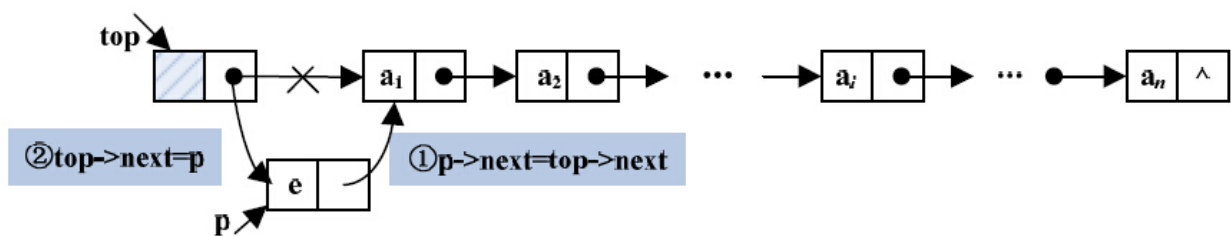


图4-8 进栈操作

注意 在插入新结点时，需要注意插入结点的顺序不能颠倒。

将元素e入栈的算法实现如下。

```
int PushStack(LinkStack top, DataType e)
/*
将元素e
入栈，进栈成功返回1*/
{
    LStackNode *p;                /*
    定义指针p
    ，指向新生成的结点*/
    if ((p = (LStackNode*) malloc(sizeof(LStackNode))) == NULL) /*
    动态生成新结点*/
    {
        printf("
内存分配失败!");
        exit(-1);
    }
    p->data = e;                    /*
    将e
    赋给p
    指向的结点数据域*/
    p->next = top->next;            /*
    指针p
    指向头结点*/
    top->next = p;                  /*
    ... ..
```



```

栈顶结点的指针域指向新插入的结点*/
    return 1;
}

```

④将栈顶元素出栈。先判断栈是否为空，如果栈为空，返回0表示出栈操作失败；否则，将栈顶元素出栈，并将栈顶元素值赋给e，最后释放结点空间，返回1表示出栈操作成功。出栈操作如图4-9所示。

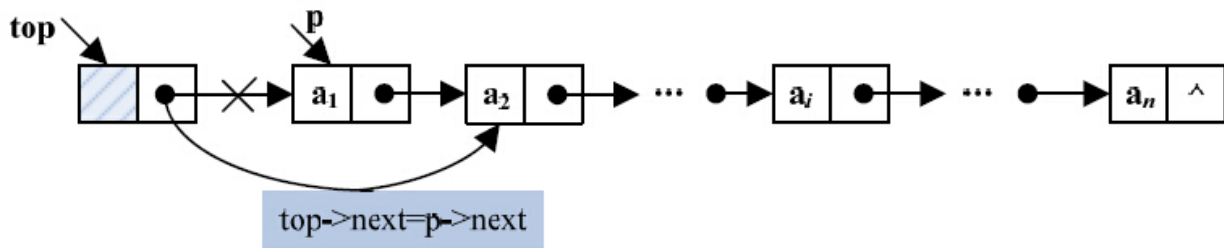


图4-9 出栈操作

将栈顶元素出栈的算法实现代码如下。

```

int PopStack(LinkStack top,DataType *e)
/*
将栈顶元素出栈。删除成功返回1
，失败返回0*/
{
    LStackNode *p;
    p=top->next;
    if(!p)
    判断链栈是否为空*/
    {
        printf("
栈已空");
        return 0;
    }
    top->next=p->next;
    将栈顶结点与链表断开，即出栈*/
    *e=p->data;
    将出栈元素赋值给e*/
    free(p);
    释放p
    指向的结点*/
    return 1;
}

```

⑤取栈顶元素。在取栈顶元素前要判断链栈是否为空，如果为空，则返回0表示取栈顶元素失败；否则将栈顶元素赋给e，并返回1表示取栈顶元素成功。取栈顶元素的算法实现如下。

```
int GetTop(LinkStack top,DataType *e)
/*
取栈顶元素。取栈顶元素成功返回1
，否则返回0*/
{
    LStackNode *p;
    p=top->next;
    指针p
    指向栈顶结点*/
    if(!p)
        如果栈为空*/
    {
        printf("
栈已空");
        return 0;
    }
    *e=p->data;
    将p
    指向的结点元素赋值给e*/
    return 1;
}
```

⑥求栈的长度。栈的长度就是链栈的元素个数。从栈顶指针即从链表的头指针开始，通过指针域找到下一个结点，并使用变量count计数，直到栈底为止，count的值就是栈的长度，将count返回即可，求栈的长度的时间复杂度为 $O(n)$ 。求链栈长度的算法实现如下。

```
int StackLength(LinkStack top)
/*
求栈的长度操作*/
{
    LStackNode *p;
    int count=0;
    定义一个计数器，并初始化为0*/
    p=top;
    指向栈顶指针*/
    while (p->next!=NULL)
        如果栈中还有结点*/
    {
        p=p->next;
        依次访问栈中的结点*/
        count++;
        每次找到一个结点，计数器累加1*/
    }
}
```

```
    }  
    return count; /*  
    返回栈的长度*/  
}
```

⑦销毁链栈。在程序结束时要释放动态申请的结点空间。从栈顶开始，通过栈顶指针依次通过free函数释放结点空间，直到栈底为止。销毁链栈的算法实现如下。

```
void DestroyStack(LinkStack top)  
/*  
销毁链栈。通过一个指针指向栈顶指针，从栈顶开始，依次释放结点空间，直到最后一个结点*/  
{  
    LStackNode *p,*q;  
    p=top;  
    while(!p) /*  
如果栈还有结点*/  
    {  
        q=p; /*q  
就是要释放的结点*/  
        p=p->next; /*p  
指向下一个结点，即下次要释放的结点*/  
        free(q); /*  
释放q  
指向的结点空间*/  
    }  
}
```

4.3.3 链栈应用举例

【例4-3】 利用链表模拟栈实现将十进制数5678转换为对应的八进制数。

【分析】 进制转换是计算机实现计算的基本问题。根据掌握的C语言知识，可以采用辗转相除法实现将十进制数转换为八进制数。将5678转换为八进制数的过程如图4-10所示。

转换后的八进制数为 $(13056)_8$ 。通过观察图4-10的转换过程不难看出，每次不断利用被除数除以8得到商数后，记下余数，又将商数作为新的被除数继续除以8，直到商数为0为止，把得到的余数排列起来就是转换后的八进制数。依据此原理，十进制数N转换为八进制的算法如下。

①将N除以8，记下其余数；

②判断商是否为零，如果为零，结束程序；否则，将商送N，转到①继续执行。

得到的余数序列倒过来排序就是八进制数。需要注意的是，这些得到的八进制数是从低位到高位产生的，最先得到的余数是八进制数的最低位，最后得到的余数是八进制数的最高位。得到的位序正好与八进制数的位序相反，这恰好可利用栈的“后进先出”特性，先把得到的余数序列放入栈保存，最后依次出栈得到八进制数。

在利用链表实现将十进制数转换为八进制数时，可以将每次得到的余数按照头插法插入链表，即将元素入栈，然后从链表的头指针开始依次输出结点的元素值，就得到了八进制数，即将元素出栈。这正好是元素的入栈与出栈操作。也可以利用栈的基本操作实现栈的进制转换。

十进制转换为八进制的算法描述如下。

```

/*
包含头文件*/
#include<stdio.h>
#include<stdlib.h>
typedef int DataType;
typedef struct node
{
    DataType data;
    struct node *next;
}LStackNode,*LinkStack;
void Coersion(int N)
/*
利用链表模拟栈将十进制数转换为八进制数*/
{
    LStackNode *p,*top=NULL; /*
    定义指向结点的指针和栈顶指针top
    ,并初始化栈为空*/
    do
    {
        p=(LStackNode*)malloc(sizeof(LStackNode)); /*
        动态生成新结点*/
        p->data=N%8; /*
        将余数送入新结点的数据域*/
        p->next=top; /*
        将新结点插入到原栈顶结点之前,使其成为新的栈顶*/
        top=p; /*
        栈顶指针指向刚插入链表的结点,成为栈顶*/
        N=N/8;
    }while(N!=0);
    while(top!=NULL) /*
    如果栈不空,从栈顶开始输出栈顶元素*/
    {
        p=top; /*p
        指向栈顶*/
        printf("%d",p->data); /*
        输出栈顶元素*/
        top=top->next; /*
        栈顶结点元素出栈*/
        free(p); /*
        释放栈顶结点*/
    }
}
void main()
{
    int n;
    printf("
请输入一个十进制数:\n");
    scanf("%d",&n);
    printf("
转换后的八进制数为:\n");
    Coersion(n);
    printf("\n");
}

```

程序运行结果如图4-11所示。

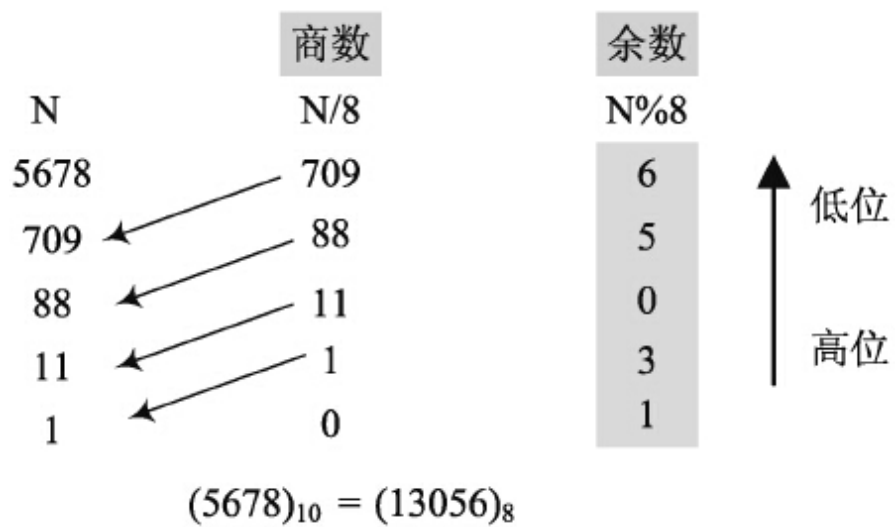


图4-10 十进制数5678转换为八进制数的过程

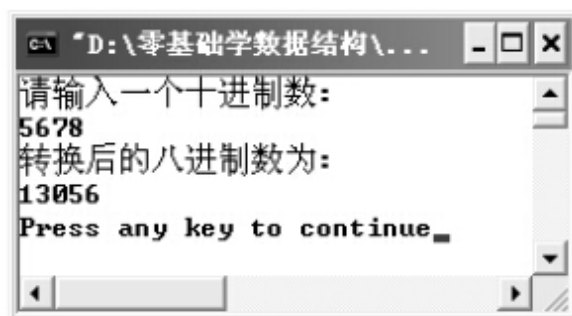


图4-11 链栈操作的程序运行结果

4.4 栈的典型应用

“后进先出”特性，使栈结构的应用非常广泛。例如，在计算机程序的编译和运行过程中，需要利用栈的“后进先出”特性对程序的语法进行检查，如进制转换、括号的匹配、表达式求值、迷宫求解。本节通过几个例子介绍栈的具体应用。

4.4.1 括号匹配

在计算机中，常见的括号有花括号、方括号和圆括号3种。{和}、[和]、（和）分别是匹配的括号，括号的嵌套顺序是任意的，即{[（）]}和（[{}（）]）等为正确的格式，{[]}、[（）}和{（）}]等为不正确的格式。例如，有括号序列如图4-12所示。

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

图4-12 括号序列

当计算机接受了第1个括号{后，它期待着与第8个括号}匹配，然而等来的却是第2个出现的括号[，此时第1个括号的期待只能暂时靠边等待，最迫切的期待出现的括号变成了第7个即将出现的]，结果等来的第3个括号是（，它的期待程度比第2个括号更为迫切，这样最期待

的括号又变成了)，在第4个括号)出现之后，第3个括号的期望得到满足，第2个括号的期望又成了当前最迫切的任务了。然而，迎来的第5个括号却是(，那么现在第5个括号的期望又成为首要解决的问题，……依次类推，直到第1个括号的期望得到满足，即直到第8个括号}出现，问题最终得到解决，说明这个括号序列是匹配的。

不难看出，括号匹配的处理过程符合栈的后进先出的特点。因此，解决括号匹配问题可以利用栈来实现，设置一个栈，依次读入括号序列。如果读入的是左括号，则进栈；若是右括号，则与栈顶的括号进行比较，是否匹配，如果匹配，则栈顶的括号出栈。如果栈不空，并且此时的括号不匹配，则说明整个括号序列不匹配。当括号序列读入完毕，且栈为空时，说明整个括号序列是匹配的。

【例4-4】 任意给定一个数学表达式如 $\{5 * (9 - 2) - [15 - (8 - 3) / 2]\} + 3 * (6 - 4)$ ，试设计一个算法判断表达式的括号是否匹配。

【分析】 检验括号是否匹配可以设置一个栈，每读入一个括号，如果是左括号，则直接进栈。

(1) 如果读入的是右括号。

- 且与当前栈顶的左括号是同类型的，说明这一对括号是匹配的，则将栈顶的左括号出栈，否则不匹配。

- 且栈已经为空，说明缺少左括号，该括号序列不匹配。

(2) 如果输入序列已经读完，而栈中仍然有等待匹配的左括号，说明缺少右括号，该括号序列不匹配。

(3) 如果读入的是数字字符，则不进行处理，直接读入下一个字符。

(4) 当输入序列和栈同时变为空时，说明括号完全匹配。

算法程序实现如下所示。

```

/*
包含头文件*/
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
#include "string.h"
/*
宏定义和链栈类型定义*/
typedef char DataType;
typedef struct node
{
    DataType data;
    struct node *next;
}LStackNode,*LinkStack;
#include"LinkStack.h"
/*
包含链栈实现文件*/
int Match(DataType e,DataType ch);
/*
检验括号是否匹配函数*/
void main()
{
    LinkStack S;
    char *p;
    DataType e;
    DataType ch[60];
    InitStack(&S);
/*
初始化链栈*/
    printf("
请输入算术表达式(
可以包含括号'{'','[]','()'):\n");
    gets(ch);
    p=ch;
/*p
指向输入的括号表达式*/
    while(*p)
/*
判断p
指向的字符是否是字符串结束标记*/
    {

```

```

        switch(*p)
        {
            case '(':
            case '[':
            case '{':
                PushStack(S, *p++);
                /*
如果是左括号，将括号进栈*/
                break;
            case ')':
            case ']':
            case '}':
                if (StackEmpty(S))
                    /*
如果是右括号且栈已空，说明缺少左括号*/
                {
                    printf("
缺少左括号.\n");
                    return;
                }
                else
                {
                    GetTop(S, &e);
                    /*
如果栈不空，读入的是右括号，则取出栈顶的括号*/
                    if (Match(e, *p))
                        /*
将栈顶的括号与读入的右括号进行比较*/
                        PopStack(S, &e);
                    /*
如果栈顶括号与读入的右括号匹配，则将栈顶的括号出栈*/
                    else
                        /*
如果栈顶括号与读入的括号不匹配，则说明此括号序列不匹配*/
                    {
                        printf("
左右括号不匹配.\n");
                        return;
                    }
                }
            default:
                /*
如果是其他字符，则不处理，直接将p
指向下一个字符*/
                p++;
        }
    }
    if (StackEmpty(S))
        /*
如果字符序列读入完毕，且栈已空，说明括号序列匹配*/
        printf("
括号匹配.\n");
    else
        /*
如果字符序列读入完毕，且栈不空，说明缺少右括号*/
        printf("
缺少右括号.\n");
}

int Match(DataType e, DataType ch)
/*
判断左右两个括号是否为同类型的括号，同类型则返回1
，否则返回0*/
{
    if (e == '(' && ch == ')')
        return 1;
    else if (e == '[' && ch == ']')
        return 1;
    else if (e == '{' && ch == '}')
        return 1;
    else
        return 0;
}

```

程序的运行结果如图4-13所示。

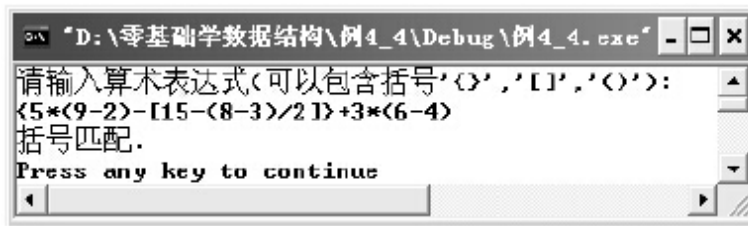


图4-13 括号匹配程序运行结果

4.4.2 求算术表达式的值

表达式求值是程序设计编译中的基本问题，它的实现是栈应用的一个典型例子。本节介绍一种简单并广为使用的算法，称为算符优先法。计算机编译系统利用栈的“后进先出”特性把人们便于理解的表达式翻译成计算机能够正确理解的表示序列。

一个算术表达式是由操作数、运算符和分界符组成。为了简化问题，假设算术运算符仅由加、减、乘、除4种运算符和左、右圆括号组成。

例如一个算术表达式为 $6 + (7 - 1) * 3 + 10 / 2$ ，这种算术表达式中的运算符总是出现在两个操作数之间，这种算术表达式称为**中缀表达式**。计算机编译系统在计算一个算术表达式之前，要将中缀表达式转换为**后缀表达式**，然后对后缀表达式进行计算。后缀表达式就是算术运算符出现在操作数之后，并且不含括号。

计算机在求算术表达式的值时分为如下两个步骤。

①将中缀表达式转换为后缀表达式。

②求后缀表达式的值。

1. 将中缀表达式转换为后缀表达式

要将一个算术表达式的中缀形式转化为相应的后缀形式，首先要了解算术四则运算的规则。算术四则运算的规则如下。

(1) 先乘除，后加减。

(2) 同级别的运算从左到右进行计算。

(3) 先括号内，后括号外。

举例算术表达式 $6 + (7 - 1) * 3 + 10 / 2$ 转换为后缀表达式为 $6\ 7\ 1\ -\ 3 * + 10\ 2 / +$ 。

不难看出，转换后的后缀表达式具有以下两个特点。

- 后缀表达式与中缀表达式的操作数出现顺序相同，只是运算符先后顺序改变了。

- 后缀表达式不出现括号。

正因为后缀表达式的以上特点，所以编译系统不必考虑运算符的优先关系。仅需要从左到右依次扫描后缀表达式的各个字符，遇到运算符时，直接对运算符前面的两个操作数进行运算即可。

如何将中缀表达式转换为后缀表达式呢？根据中缀表达式与后缀表达式中的操作数次序相同，只是运算符次序不同的特点，设置一个栈，用于存放运算符。依次读入表达式中的每个字符，如果是操作数，则直接输出。如果是运算符，则比较栈顶元素符与当前运算符的优先级，然后进行处理，直到整个表达式处理完毕。我们约定#作为后缀表达式的结束标志，假设 θ_1 为栈顶运算符， θ_2 为当前扫描的运算符。则中缀表达式转换为后缀表达式的算法描述如下。

①初始化栈，并将#入栈。

②若当前读入的字符是操作数，则将该操作数输出，并读入下一字符。

③若当前字符是运算符，记作 θ_2 ，则将 θ_2 与栈顶的运算符 θ_1 比较。若 θ_1 优先级低于 θ_2 ，则将 θ_2 进栈；若 θ_1 优先级高于 θ_2 ，则将 θ_1 出栈并将其作为后缀表达式输出。然后继续比较新的栈顶运算符 θ_1 与当前运算符 θ_2 的优先级，若 θ_1 的优先级与 θ_2 相等，且 θ_1 为 (， θ_2 为)，则将 θ_1 出栈，继续读入下一个字符。

④如果 θ_2 的优先级与 θ_1 相等，且 θ_1 和 θ_2 都为#，将 θ_1 出栈，栈为空。则完成中缀表达式转换为后缀表达式，算法结束。

运算符的优先关系如表4-1所示。

表4-1 运算符的优先关系

$\theta_1 \backslash \theta_2$	+	-	*	/	()	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

初始化一个空栈，用来对运算符进行出栈和入栈操作。中缀表达式 $6+(7-1)*3+10/2\#$ 转换为后缀表达式的具体过程如图4-14所示（为了便于描述，可在要转换表达式的末尾加一个#作为结束标记）。

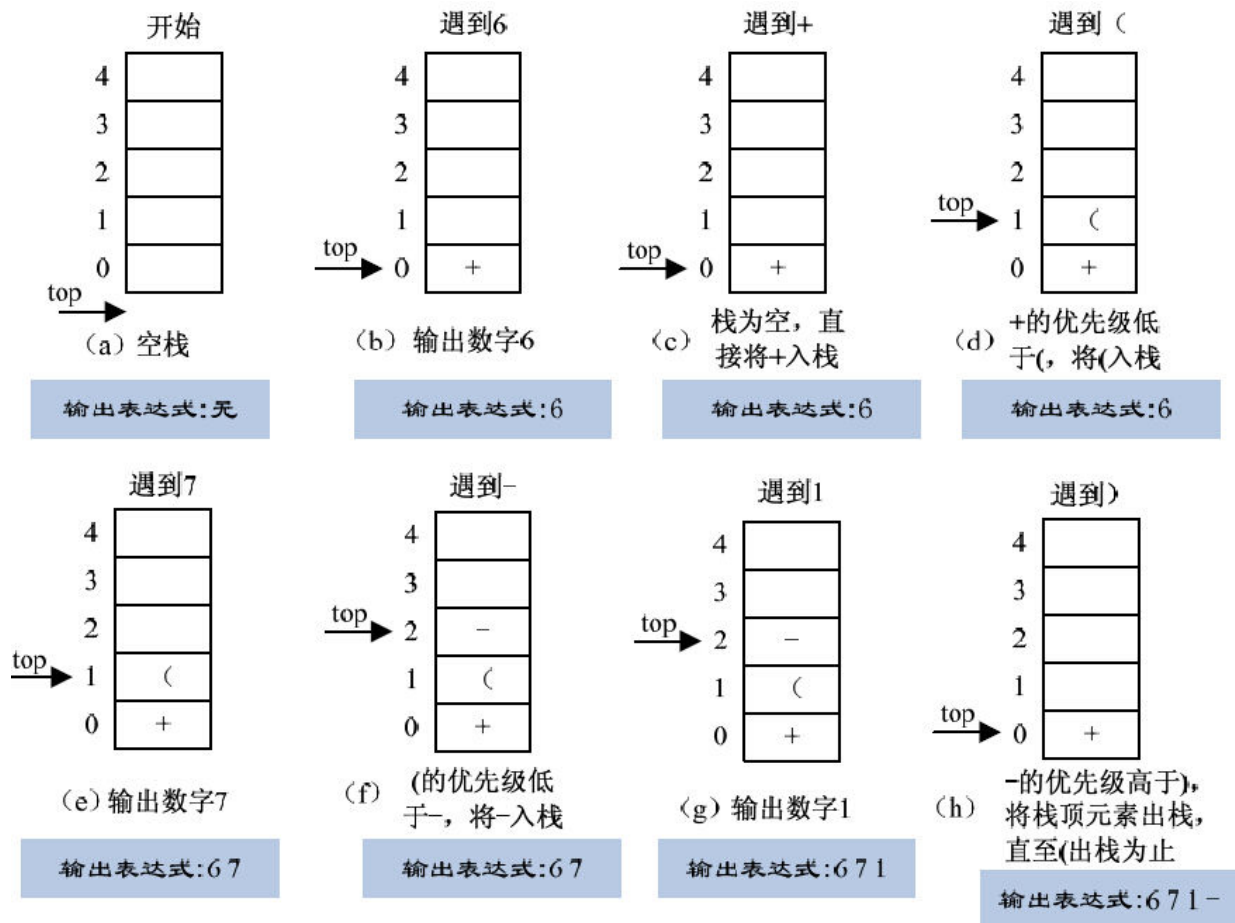


图4-14 中缀表达式 $6+(7-1)*3+10/2$ 转换为后缀表达式的过程

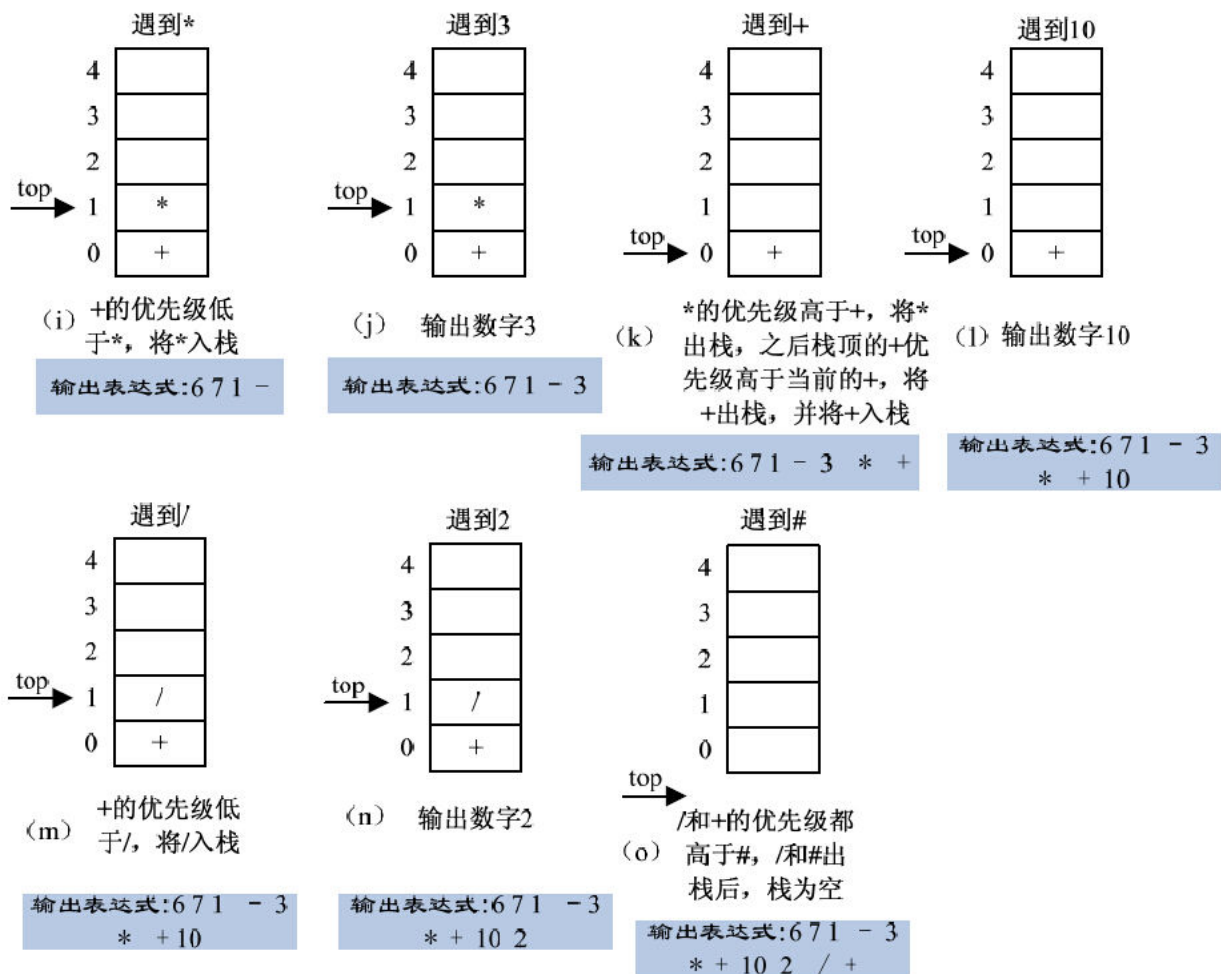


图4-14 (续)

2. 求后缀表达式的值

将中缀表达式转换为后缀表达式后, 就可以计算后缀表达式的值了。计算后缀表达式值的规则是, 依次读入后缀表达式中的每个字符, 如果是操作数, 则将操作数进入栈; 如果是运算符, 则将处于栈顶的两个操作数出栈, 然后利用当前运算符进行运算, 将运行结果入栈, 直到整个表达式处理完毕。

利用上述规则，求后缀表达式的 $6\ 7\ 1-3*+10/+$ 的值的运算过程如图4-15所示。

3. 算法实现

【例4-5】 通过键盘输入一个表达式，如 $6+(7-1)*3+10/2$ ，要求将其转换为后缀表达式，并计算该表达式的值。

【分析】 设置两个字符数组str和exp，str用于存放中缀表达式的字符串，exp用于存放后缀表达式的字符串。利用栈将中缀表达式转换为后缀表达式的方法是依次扫描中缀表达式，如果遇到数字则将其直接存入数组exp中；如果遇到的是运算符，则将栈顶运算符与当前运算符比较，如果当前的运算符的优先级高于栈顶运算符的优先级，则将当前运算符进栈；如果栈顶运算符的优先级高于当前运算符的优先级，则将栈顶运算符出栈，并保存到数组exp中。

中缀表达式 $6+(7-1)*3+10/2$ 转换为后缀表达式的处理流程如图4-16所示。

其中， θ_1 为栈顶运算符， θ_2 为当前扫描的运算符。

求后缀表达式的值时，依次扫描后缀表达式中的每个字符，如果是数字字符，将其转换为数字（数值型数据），并将其入栈；如果是运算符，则将栈顶的两个数字出栈，进行加、减、乘、除运算，并将

结果入栈。当后缀表达式对应的字符串处理完毕后，将栈中元素出栈，即为所求表达式的值。

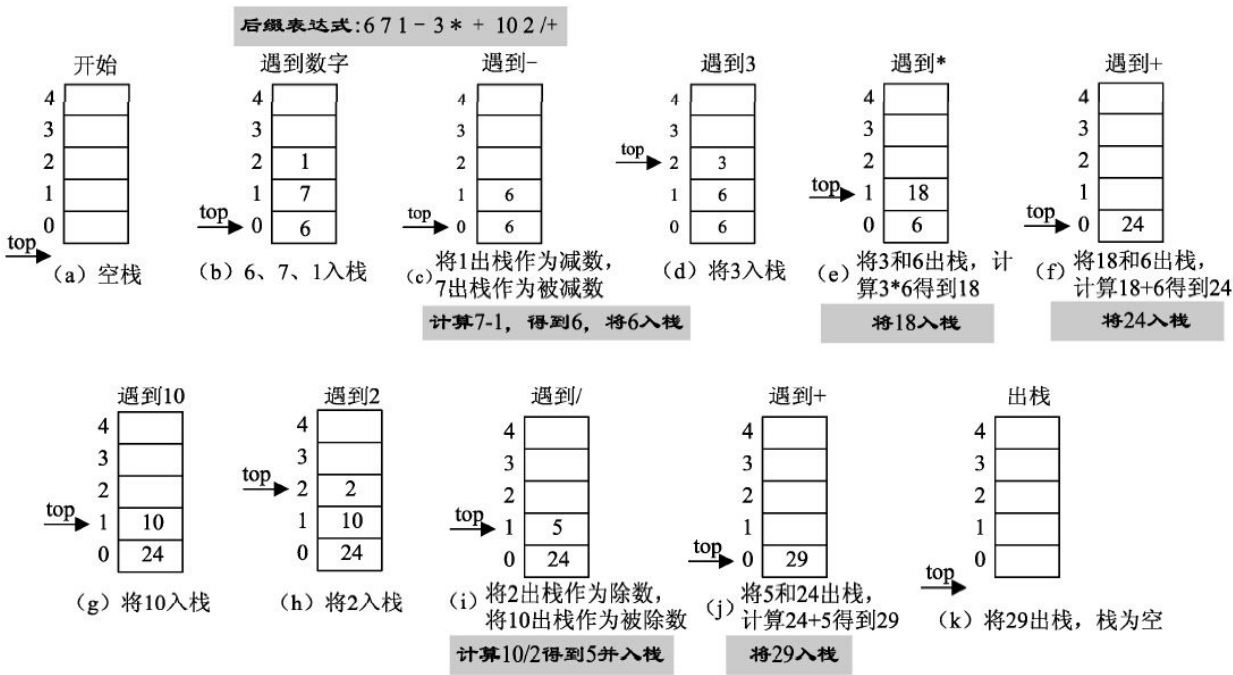


图4-15 后缀表达式 6 7 1 - 3 * + 10 2 / + 的运算过程

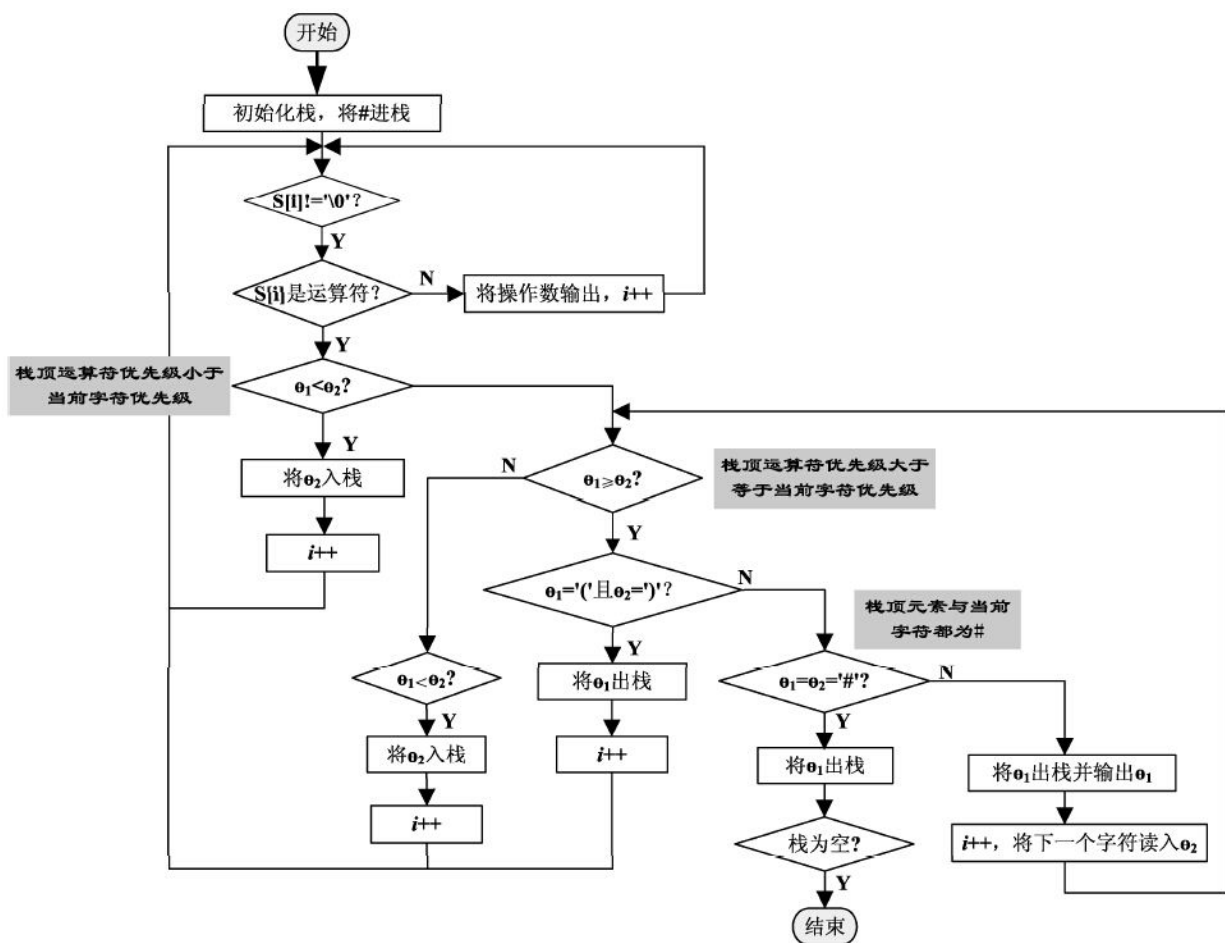


图4-16 中缀表达式转换为后缀表达式的流程图

求算术表达式的值的程序如下。

```

/*
包含头文件*/
#include<stdio.h>
#include<string.h>
/*
包含顺序栈基本操作实现函数*/
typedef char DataType;
#include"SeqStack.h"
#define MaxSize 50
/*
操作数栈定义*/
typedef struct
{
    float data[MaxSize];
    int top;
}OpStack;
/*
函数声明*/
void TranslateExpress(char s1[],char s2[]);

```

```

float ComputeExpress(char s[]);
void main()
{
    char a[MaxSize],b[MaxSize];
    float f;
    printf("
请输入一个算术表达式: \n");
    gets(a);
    printf("
中缀表达式为: %s\n",a);
    TranslateExpress(a,b);
    printf("
后缀表达式为: %s\n",b);
    f=ComputeExpress(b);
    printf("
计算结果: %f\n",f);
}
float ComputeExpress(char a[])
/*
计算后缀表达式的值*/
{
    OpStack S; /*
定义一个操作数栈*/
    int i=0,value;
    float x1,x2;
    float result;
    S.top=-1; /*
初始化栈*/
    while(a[i]!='\0') /*
依次扫描后缀表达式中的每个字符*/
    {
        if(a[i]!=' '&&a[i]>='0'&&a[i]<='9') /*
如果当前字符是数字字符*/
        {
            value=0;
            while(a[i]!=' ') /*
如果不是空格,说明数字字符是两位数以上的数字字符*/
            {
                value=10*value+a[i]-'0';
                i++;
            }
            S.top++;
            S.data[S.top]=value; /*
处理之后将数字进栈*/
        }
        else /*
如果当前字符是运算符*/
        {
            switch(a[i]) /*
将栈中的数字出栈两次,然后用当前的运算符进行运算,再将结果入栈*/
            {
                case '+':
                    x1=S.data[S.top];
                    S.top--;
                    x2=S.data[S.top];
                    S.top--;
                    result=x1+x2;
                    S.top++;
                    S.data[S.top]=result;
                    break;
                case '-':
                    x1=S.data[S.top];
                    S.top--;
                    x2=S.data[S.top];
                    S.top--;
                    result=x2-x1;
                    S.top++;

```

```

        S.data[S.top]=result;
        break;
    case '*':
        x1=S.data[S.top];
        S.top--;
        x2=S.data[S.top];
        S.top--;
        result=x1*x2;
        S.top++;
        S.data[S.top]=result;
        break;
    case '/':
        x1=S.data[S.top];
        S.top--;
        x2=S.data[S.top];
        S.top--;
        result=x2/x1;
        S.top++;
        S.data[S.top]=result;
        break;
    }
    i++;
}
}
if(!S.top!=-1) /*
如果栈不空，将结果出栈，并返回*/
{
    result=S.data[S.top];
    S.top--;
    if(S.top==-1)
        return result;
    else
    {
        printf("
表达式错误");
        exit(-1);
    }
}
}
void TranslateExpress(char str[],char exp[])
/*
把中缀表达式转换为后缀表达式*/
{
    SeqStack S; /*
    定义一个栈，用于存放运算符*/
    char ch;
    DataType e;
    int i=0,j=0;
    InitStack(&S);
    ch=str[i];
    i++;
    while(ch!='\0') /*
    依次扫描中缀表达式中的每个字符*/
    {
        switch(ch)
        {
            case '(': /*
            如果当前字符是左括号，则将其进栈*/
                PushStack(&S,ch);
                break;
            case ')': /*
            如果是右括号，将栈中的运算符出栈，并将其存入数组exp
            中*/
                while(GetTop(S,&e)&&e!='(')
                {
                    PopStack(&S,&e);
                    exp[j]=e;

```

```

        j++;
        exp[j]=' ';
        /*
加上空格*/
        j++;
    }
    PopStack(&S,&e);
    /*
将左括号出栈*/
    break;
    case '+':
    case '-':
        /*
如果遇到的是 '+'
和 '-'
, 因为其优先级低于栈顶运算符的优先级, 所以先将栈顶字符出栈, 并将其存入数组exp
中, 然后将当前运算符进栈*/
        while(!StackEmpty(S) && GetTop(S, &e) && e!='(')
        {
            PopStack(&S,&e);
            exp[j]=e;
            j++;
            exp[j]=' ';
            /*
加上空格*/
            j++;
        }
        PushStack(&S,ch);
        /*
当前运算符进栈*/
        break;
        case '*':
        /*
如果遇到的是 '*'
和 '/',
先将同级运算符出栈, 并存入数组exp
中, 然后将当前的运算符进栈*/
        case '/':
            while(!StackEmpty(S) && GetTop(S, &e) && e=='/' || e=='*')
            {
                PopStack(&S,&e);
                exp[j]=e;
                j++;
                exp[j]=' ';
                /*
加上空格*/
                j++;
            }
            PushStack(&S,ch);
            /*
当前运算符进栈*/
            break;
            case ' ':
                /*
如果遇到空格, 忽略*/
                break;
            default:
                /*
如果遇到的是操作数, 则将操作数直接送入数组exp
中, 并在其后添加一个空格, 用来分隔数字字符*/
                while(ch>='0' && ch<='9')
                {
                    exp[j]=ch;
                    j++;
                    ch=str[i];
                    i++;
                }
                i--;
                exp[j]=' ';
                j++;
            }
            ch=str[i];
            /*
读入下一个字符, 准备处理*/
            i++;
        }
    }

```

```

        while(!StackEmpty(S))                /*
将栈中所有剩余的运算符出栈，送入数组exp
中*/
        {
            PopStack(&S,&e);
            exp[j]=e;
            j++;
            exp[j]=' ';                        /*
加上空格*/
            j++;
        }
        exp[j]='\0';
    }

```

程序运行结果如图4-17所示。

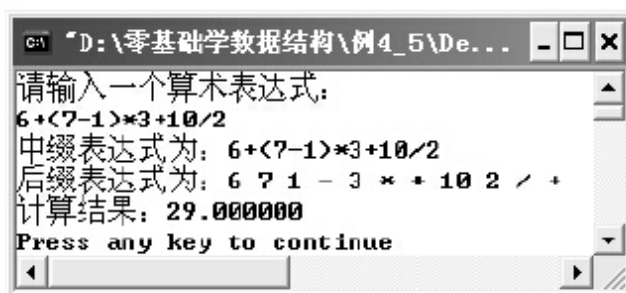


图4-17 算术表达式程序运行结果

注意 在将中缀表达式转换为后缀表达式的过程中，每输出一个数字字符，需要在其后补一个空格，与其他相邻数字字符隔开，否则一连串数字字符放在一起无法区分是一个数字还是两个数字。

在ComputeExpress函数中，遇到-运算符时，先出栈的为减数，后出栈的为被减数；对于/运算也一样。

4.4.3 迷宫求解

求迷宫中从入口到出口的路径是经典的程序设计问题。通常采用穷举法即从入口出发，顺某一个方向向前探索，若能走通，则继续往前走；否则沿原路返回，换另一个方向继续探索，直到探索到出口为止。为了保证在任何位置都能原路返回，显然需要用一个后进先出的栈来保存从入口到当前位置的路径。

可以用如图4-18所示的方块迷宫，空白方块表示通道，带阴影的方块表示墙。

所求路径必须是简单路径，即求得的路径上不能重复出现同一通道块。求迷宫中一条路径的算法的基本思路是，如果当前位置“可通”，则纳入“当前路径”，并继续朝下一个位置探索，即切换下一个位置为当前位置，如此重复直至到达出口；如果当前位置不可通，则应沿“来向”退回到前一通道块，然后朝“来向”之外的其他方向继续探索；如果该通道块的四周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（东、南、西、北）4个方向上相邻的方块。假设入口位置为（1，1），出口位置为（8，8），根据以上算法搜索出来的一条路径如图4-19所示。

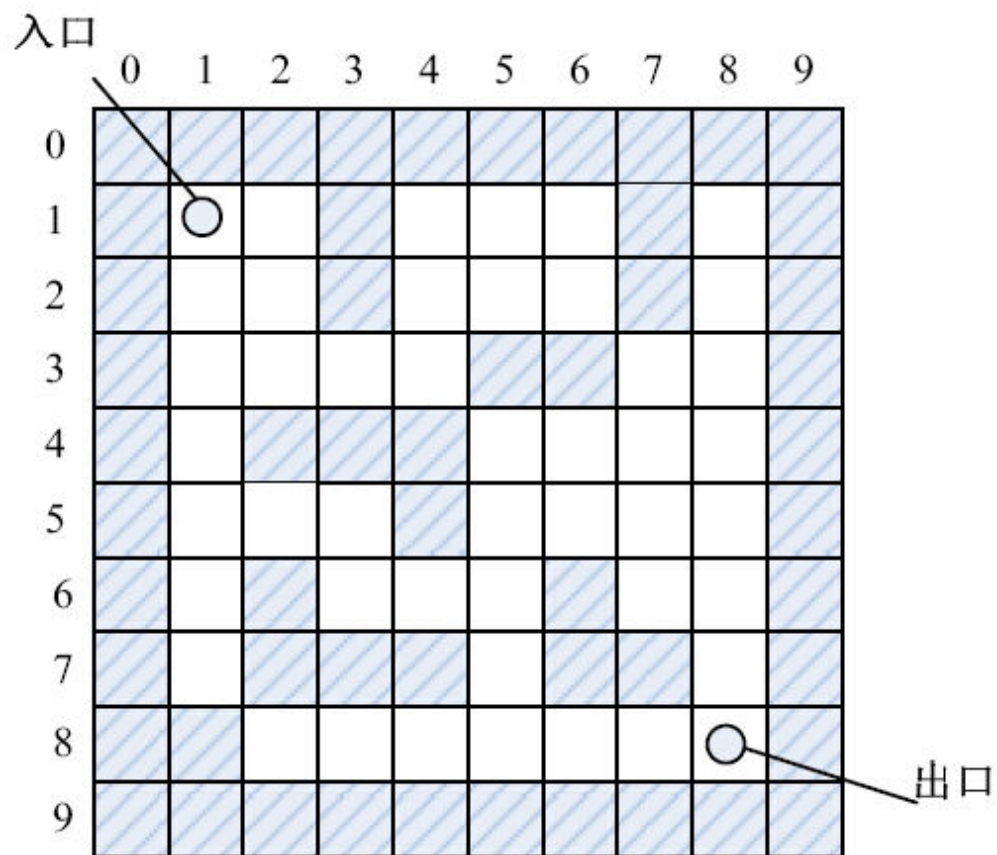


图 4-18 迷宫

图4-18 迷宫

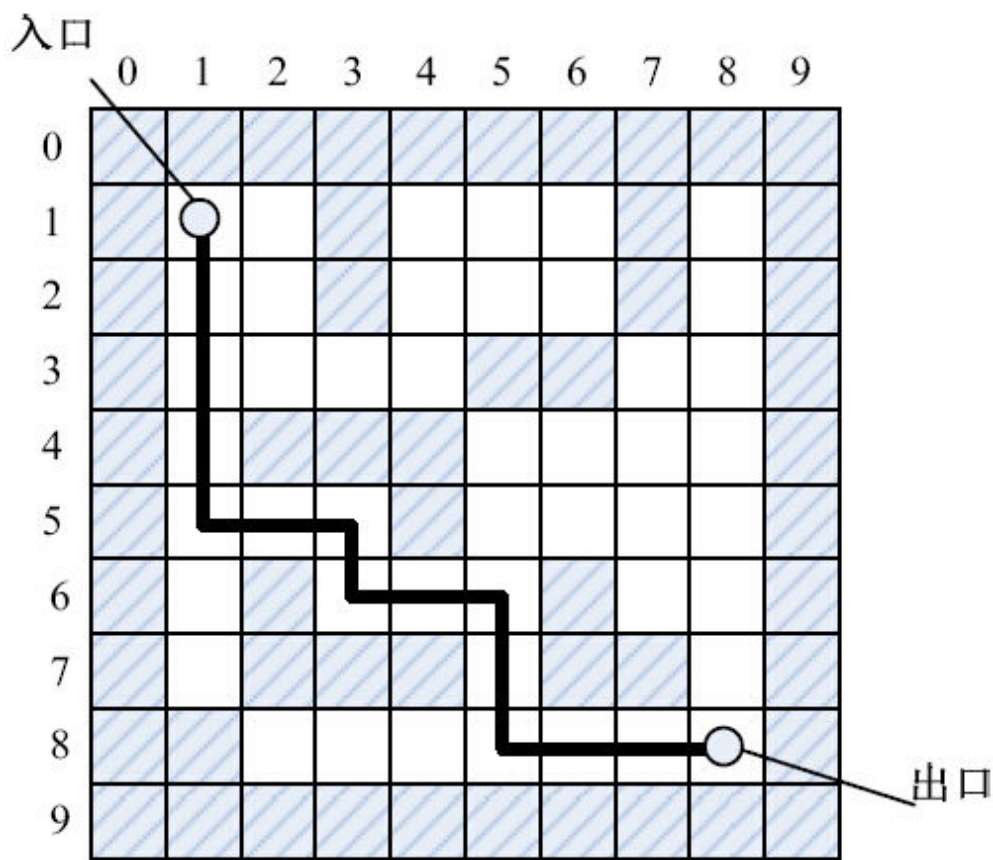


图4-19 迷宫中的一条可通路径

【例4-6】 参考图4-18所示迷宫，编写算法求一条从入口到出口的路径。

在程序的实现中，定义墙元素值为0，可通过路径为1，不能通过路径为-1。完整的求解迷宫程序如下。

```

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
typedef struct
{
    int x; /*
行值 */
    int y; /*
列值 */
} PosType; /*
... ..

```

```

迷宫坐标位置类型 */
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(0
~3
表示东~北) */
}DataType; /*
栈的元素类型 */
#include "SeqStack.h"
#define MAXLENGTH 40 /*
设迷宫的最大行列为40 */
typedef int MazeType[MAXLENGTH][MAXLENGTH]; /*
迷宫数组类型[
行][
列] */
MazeType m; /*
迷宫数组 */
int x,y; /*
迷宫的行数,列数 */
PosType begin,end; /*
迷宫的入口坐标,
出口坐标 */
int curstep=1; /*
当前足迹,初值(
在入口处)
为1 */
void Init(int k)
/*
设定迷宫布局(
墙为值0,
通道值为k) */
{
    int i,j,x1,y1;
    printf("
请输入迷宫的行数,
列数(
包括外墙)
: ");
    scanf("%d,%d",&x,&y);
    for(i=0;i<x;i++) /*
定义周边值为0(
外墙) */
    {
        m[0][i]=0; /*
行周边 */
        m[x-1][i]=0;
    }
    for(i=0;i<y-1;i++)
    {
        m[i][0]=0; /*
列周边 */
        m[i][y-1]=0;
    }
    for(i=1;i<x-1;i++)
        for(j=1;j<y-1;j++)
            m[i][j]=k; /*
定义除外墙,其余都是通道,初值为k */
    printf("
请输入迷宫内墙单元数: ");
    scanf("%d",&j);
    printf("
请依次输入迷宫内墙每个单元的行数,
....

```

```

列数: \n");

                                for(i=1;i<=j;i++)
                                {
                                    scanf("%d,%d",&x1,&y1);
                                    m[x1][y1]=0; /*
修改墙的值为0 */

                                }
                                printf("
迷宫结构如下:\n");

                                Print();
                                printf("
请输入入口的行数,
列数: ");

                                scanf("%d,%d",&begin.x,&begin.y);
                                printf("
请输入出口的行数,
列数: ");

                                scanf("%d,%d",&end.x,&end.y);
}
void Print()
/*
输出迷宫的解(m
数组) */
{
    int i,j;
    for(i=0;i<x;i++)
    {
        for(j=0;j<y;j++)
            printf("%3d",m[i][j]);
        printf("\n");
    }
}
int Pass(PosType b)
/*
当迷宫m
的b
点的序号为1(
可通过路径)
, 返回1
; 否则返回0*/
{
    if(m[b.x][b.y]==1)
        return 1;
    else
        return 0;
}
void FootPrint(PosType a)
/*
使迷宫m
的a
点的值变为足迹(curstep) */
{
    m[a.x][a.y]=curstep;
}
void NextPos(PosType *c,int di)
/*
根据当前位置及移动方向,求得下一位置 */
{
    PosType direc[4]={0,1},{1,0},{0,-1},{-1,0}}; /*
行增量,
列增量},
移动方向,
依次为东南西北 */

    (*c).x+=direc[di].x;
    (*c).y+=direc[di].y;
}
void MarkPrint(PosType b)

```

```

/*
使迷宫m
的b
点的序号变为-1(
不能通过的路径) */
{
    m[b.x][b.y]=-1;
}
int MazePath(PosType start,PosType end)
/*
若迷宫m
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1
；否则返回0*/
{
    SeqStack S; /*
顺序栈 */
    PosType curpos; /*
当前位置 */
    DataType e; /*
栈元素 */
    InitStack(&S); /*
初始化栈 */
    curpos=start; /*
当前位置在入口 */
    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.ord=curstep;
            e.seat=curpos;
            e.di=0;
            PushStack(&S,e); /*

入栈当前位置及状态 */

            curstep++; /*
足迹加1 */

            if (curpos.x==end.x&&curpos.y==end.y) /*
到达终点(
出口) */

                return 1;
            NextPos(&curpos,e.di); /*
由当前位置及移动方向，确定下一个当前位置 */
        }
        else/*
当前位置不能通过 */
        {
            if(!StackEmpty(S)) /*
栈不空 */
            {
                PopStack(&S,&e); /*
退栈到前一位置 */

                curstep--; /*
足迹减1 */

                while(e.di==3&&!StackEmpty(S)) /*
前一位置处于最后一个方向(
北) */
                {
                    MarkPrint(e.seat); /*
在前一位置留下不能通过的标记(-1) */

```

```

再退回一步 */
足迹再减1 */

没到最后一个方向 (
北) */

换下一个方向探索 */
入栈该位置的下一个方向 */
足迹加1 */
确定当前位置 */
确定下一个当前位置是该新方向上的相邻块 */

        }
    }while(!StackEmpty(S));
    return 0;
}
void main()
{
    Init(1); /*
初始化迷宫, 通道值为1 */
    if(MazePath(begin,end)) /*
有通路 */
    {
        printf("
此迷宫从入口到出口的一条路径如下:\n");
        Print(); /*
输出此通路 */
    }
    else
        printf("
此迷宫没有从入口到出口的路径\n");
}

        PopStack(&S,&e); /*
        curstep--; /*
    }
    if(e.di<3) /*
    {
        e.di++; /*
        PushStack(&S,e); /*
        curstep++; /*
        curpos=e.seat; /*
        NextPos(&curpos,e.di); /*
    }
}

```

程序运行结果如图4-20所示。

```
D:\零基础学数据结构\例4_6\Debug\例4_6.exe
请输入迷宫的行数,列数<包括外墙>: 10,10
请输入迷宫内墙单元数: 18
请依次输入迷宫内墙每个单元的行数,列数:
1,3
1,7
2,3
2,7
3,5
3,6
4,2
4,3
4,4
5,4
6,2
6,6
7,2
7,3
7,4
7,6
7,7
8,1
迷宫结构如下:
0 0 0 0 0 0 0 0 0 0
0 1 1 0 1 1 1 0 1 0
0 1 1 0 1 1 1 0 1 0
0 1 1 1 1 0 0 1 1 0
0 1 0 0 0 1 1 1 1 0
0 1 1 1 0 1 1 1 1 0
0 1 0 1 1 1 0 1 1 0
0 1 0 0 0 1 0 0 1 0
0 0 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0
请输入入口的行数,列数: 1,1
请输入出口的行数,列数: 8,8
此迷宫从入口到出口的一条路径如下:
0 0 0 0 0 0 0 0 0 0
0 1 2 0 -1 -1 -1 0 1 0
0 1 3 0 -1 -1 -1 0 1 0
0 5 4 -1 -1 0 0 1 1 0
0 6 0 0 0 1 1 1 1 0
0 7 8 9 0 1 1 1 1 0
0 1 0 10 11 12 0 1 1 0
0 1 0 0 0 13 0 0 1 0
0 0 1 1 1 14 15 16 17 0
0 0 0 0 0 0 0 0 0 0
Press any key to continue
```

图4-20 迷宫求解程序运行结果

4.5 栈与递归

栈的后进先出的思想还体现在递归函数中。本节主要讲解栈与递归调用的关系、递归利用栈的实现过程、递归与非递归的转换。

4.5.1 递归

先来看一个经典的递归例子：斐波那契数列，对于学过C语言的同学来说，这个数列应该不陌生。

1. 斐波那契数列

如果兔子在出生两个月后就有繁殖能力，以后一对兔子每个月能生出一对兔子，假设所有兔子都不死，那么一年以后可以繁殖多少对兔子呢？

不妨拿新出生的一对小兔子来分析下。第一、二个月小兔子没有繁殖能力，所以还是一对；两个月后，生下一对小兔子，共有2对兔子；三个月后，老兔子又生下一对，因为小兔子还没有繁殖能力，所以一共是3对兔子；依次类推，可以得出表4-1。

表4-1 每月兔子的对数

经过的月数	1	2	3	4	5	6	7	8	9	10	11	12
兔子对数	1	1	2	3	5	8	13	21	34	55	89	144

从表4-1中不难看出，数字1、1、2、3、5、8……构成了一个数列，这个数列有个十分明显的特征，即前面相邻两项之和构成后一项，可用数学函数表示如下。

$$\text{Fib}(n) = \begin{cases} 0, & \text{当 } n=0 \text{ 时} \\ 1, & \text{当 } n=1 \text{ 时} \\ \text{Fib}(n-1) + \text{Fib}(n-2), & \text{当 } n>1 \text{ 时} \end{cases}$$

如果要打印出斐波那契数列的前40项，常规的迭代方法实现代码如下。

```

void main()
{
    int i,a[40];
    a[0]=0;
    a[1]=1;
    printf("%4d",a[0]);
    printf("%4d",a[1]);
    for(i=2;i<40;i++)
    {
        a[i]=a[i-1]+a[i-2];
        printf("%4d",a[i]);
    }
}

```

以上代码比较简单，不用过多解释，如果用递归实现，代码会更加简洁。

```

int Fib (int n)
{
    if(n==0)
        return 0;
    else if(n==1)

```

```

        return 1;
    else
        return Fib(n-1)+Fib(n-2);
}
void main()
{
    int i;
    for(i=0;i<40;i++)
        printf("%4d",Fib(i));
}

```

从结构上看，递归要比非递归代码逻辑上更清晰，但是想要真正搞明白需要好好思考下。遇到这种情况，可以先动手在纸上画一画，模拟代码的执行过程，例如，当 $n=4$ 时，代码执行过程如图4-21所示。

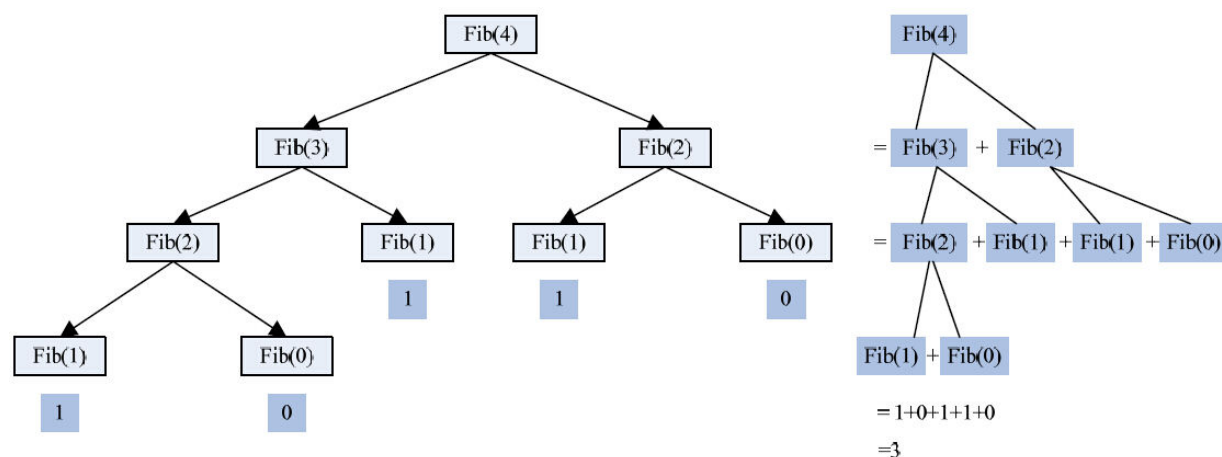


图4-21 斐波那契数列的执行过程

2. 什么是递归函数

递归是指在函数的定义中，在定义自己的同时又出现了对自身的调用。如果一个函数在函数体中直接调用自己，称为[直接递归函数](#)。如果经过一系列的中间调用，间接调用自己的函数称为[间接递归函数](#)。

例如， n 的阶乘的递归函数定义如下。

$$\text{fact}(n) \begin{cases} 1, & \text{当 } n=0 \text{ 时} \\ n * \text{fact}(n-1), & \text{当 } n > 0 \text{ 时} \end{cases}$$

n的阶乘递归函数C语言程序实现如下。

```
int fact(int n)
{
    if(n==1)
        return 1;
    else
        return n*fact(n-1);
}
```

Ackermann函数定义如下。

$$\text{Ack}(m,n) \begin{cases} n+1, & \text{当 } m=0 \text{ 时} \\ \text{Ack}(m-1,1), & \text{当 } m \neq 0, n=0 \text{ 时} \\ \text{Ack}(m-1, \text{Ack}(m,n-1)), & \text{当 } m \neq 0, n \neq 0 \text{ 时} \end{cases}$$

Ackerman递归函数C语言程序实现如下。

```
int Ack(int m,int n)
{
    if(m==0)
        return n+1;
    else if(n==0)
        return Ack(m-1,1);
    else
        return Ack(m-1,Ack(m,n-1));
}
```

3. 分析递归调用过程

递归问题可以被分解成规模小、性质相同的问题加以解决。在之后将要介绍的广义表、二叉树等都具有递归的性质，它们的操作可以用递归实现。下面以著名的汉诺塔问题为例来分析递归调用的过程。

n 阶汉诺塔问题。假设有3个塔座A、B、C，在塔座A上放置有 n 个直径大小各不相同、从小到大编号为1、2、 \dots 、 n 的圆盘，如图4-22所示。要求将A轴上的 n 个圆盘移动到塔座C上并要求按照同样的叠放顺序排列，圆盘移动时必须遵循以下规则。

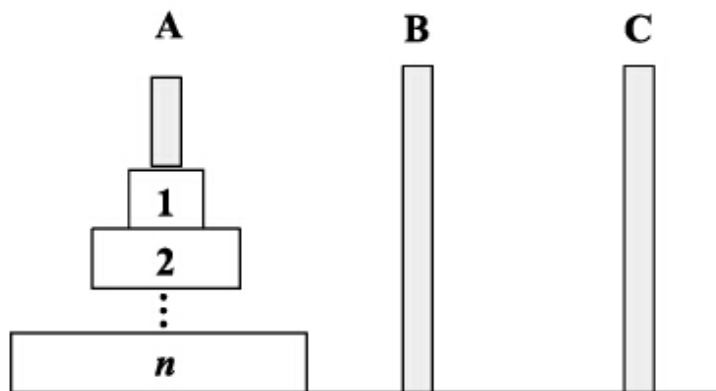


图4-22 n 阶汉诺塔初始状态

- (1) 每次只能移动一个圆盘。
- (2) 圆盘可以放置在A、B和C中的任何一个塔座。
- (3) 任何时候都不能将一个较大的圆盘放在较小的圆盘上。

如何实现将放在A上的圆盘按照规则移动到C上呢？当 $n=1$ 时，直接将编号为1的圆盘从塔座A移动到C上即可。当 $n>1$ 时，需利用塔座B作为

辅助塔座，先将放置在编号为n之上的n-1个圆盘从塔座A上移动到B上，然后将编号为n的圆盘从塔座A移动到C上，最后将塔座B上的n-1个圆盘移动到塔座C上。那现在将n-1个圆盘从一个塔座移动到另一个塔座又成为与原问题类似的问题，只是规模减小了1，故可用同样的方法解决。显然这是一个递归的问题，汉诺塔的递归算法描述如下。

```
void Hanoi(int n,char a,char b,char c)
/*
将塔座A
上按照从小到大自上而下编号为1
到n
的那个圆盘按照规则搬到塔座C
上，B
可以作为辅助塔座*/
{
    if(n==1)
        Move(a,1,c);          /*
将编号为1
的圆盘从A
移动到C*/
    else
    {
        Hanoi(n-1,a,c,b);      /*
将编号为1
到n-1
的圆盘从A
移动到B
，C
作为辅助塔座*/
        Move(a,n,c);          /*
将编号为n
的圆盘从A
移动到C*/
        Hanoi(n-1,b,a,c);      /*
将编号为1
到n-1
的圆盘从B
移动到C
，A
作为辅助塔座*/
    }
}
```

下面以n=3为例，观察一下汉诺塔递归调用的具体过程。在函数体中，当n>1，经历3个过程移动圆盘。第1个过程，将编号为1和2的圆盘从塔座A移动到B；第2个过程，将编号为3的圆盘从塔座A移动到C；第3

个过程，将编号为1和2的圆盘从塔座B移动到C。递归调用过程如图4-23所示。

第1个过程，通过调用Hanoi (2, a, c, b) 实现。Hanoi (2, a, c, b) 又调用自己，完成将编号为1的圆盘从塔座A移动到C，编号为2的圆盘从塔座A移动到B，编号为1的圆盘从塔座C移动到B，如图4-24和图4-25所示。

第2个过程完成编号3从塔座A移动到C，如图4-26所示。

第3个过程通过调用Hanoi (2, b, a, c) 实现圆盘移动。通过再次递归完成将编号为1的圆盘从塔座B移动到A，将编号2的圆盘从塔座B移动到C，将编号为1的圆盘从塔座A移动到C，如图4-26和图4-27所示。

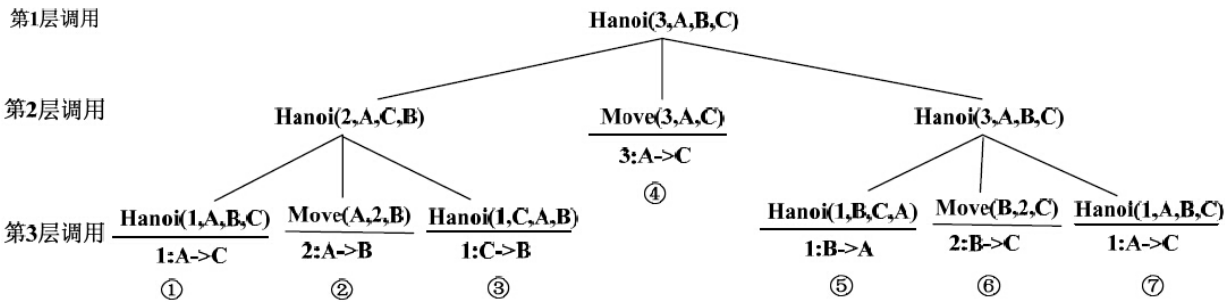


图4-23 汉诺塔递归调用过程

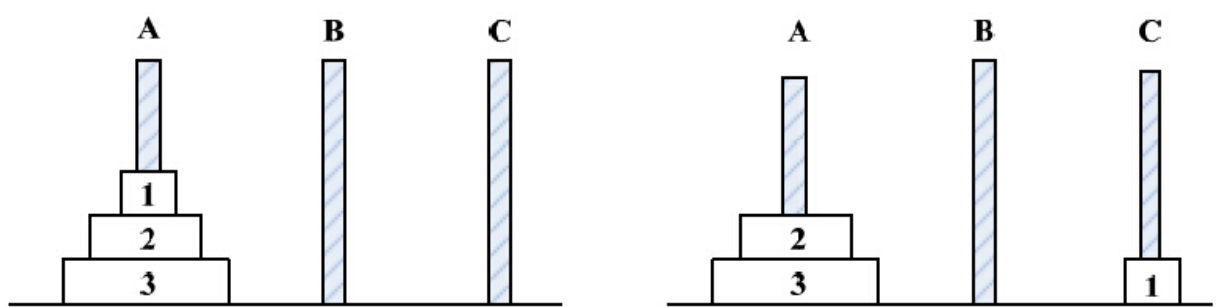


图4-24 将编号为1的圆盘从塔座A移动到C上

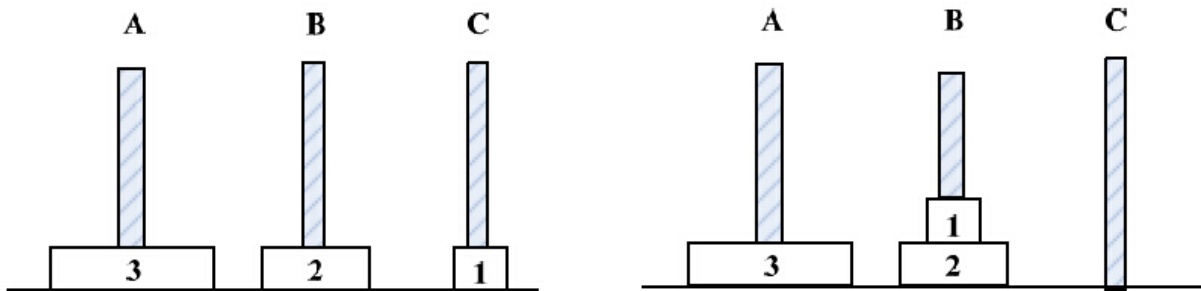


图4-25 将编号为2的圆盘从塔座A移动到B上，编号为1的圆盘从塔座C移动到B上

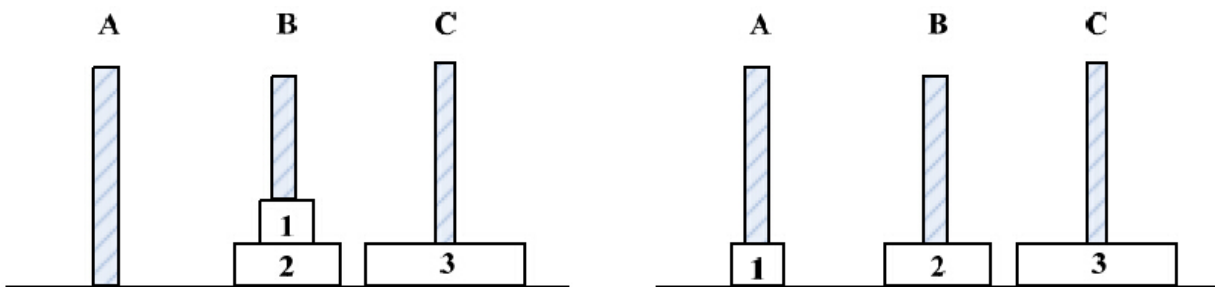


图4-26 将编号为3的圆盘从塔座A移动到C上，编号为1的圆盘从塔座B移动到A上

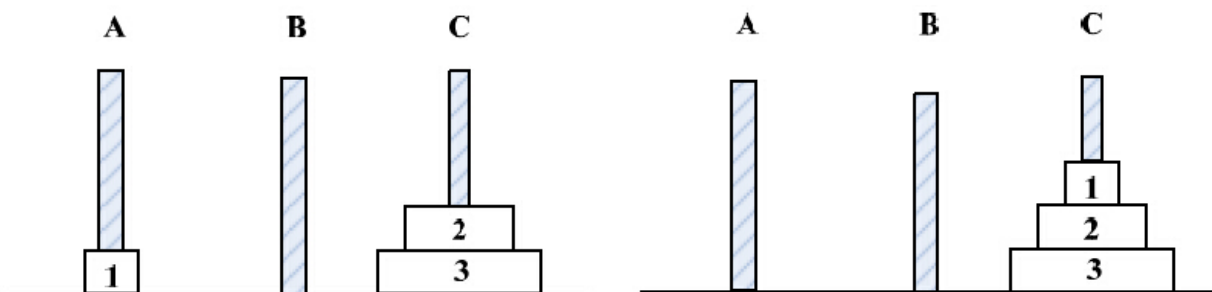


图4-27 将编号为2的圆盘从塔座B移动到C上，编号为1的圆盘从塔座A移动到C上

递归的实现本质上就是把嵌套调用变成栈实现。在递归调用过程中，被调用函数在执行前系统要完成如下3件事情。

- (1) 将所有参数和返回地址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空间。
- (3) 将控制转到被调用函数的入口。

当被调用函数执行完毕，返回给调用函数前，系统同样需要完成如下3个任务。

- (1) 保存被调用函数的执行结果。
- (2) 释放被调用函数的数据存储区。
- (3) 将控制转到调用函数的返回地址处。

在有多层嵌套调用时，后调用的先返回，刚好满足后进先出的特性，因此递归调用是通过栈实现的。函数递归调用过程中，在递归结束前，每调用一次，就进入下一层。当一层递归调用结束时，返回到上一层。

为了保证递归调用能正确执行，系统设置了一个工作栈作为递归函数运行期间使用的数据存储区。每一层递归包括实在参数、局部变量及上一层的返回地址等构成一个工作记录。每进入下一层，新的工作栈记

录被压入栈顶。每返回到上一层，就从栈顶弹出一个工作记录。因此，当前层的工作记录是栈顶工作记录，被称为活动记录。递归过程产生的栈由系统自动管理，类似用户自己定义的栈。

4.5.2 消除递归

用递归编写的程序结构清晰，算法也容易实现，读算法的人也容易理解，但递归算法的执行效率比较低，这是因为递归需要反复入栈，时间和空间都比较开销大。

为了避免这种开销，我们需要消除递归，消除递归方法通常有两种，一种是对简单的递归可以直接用迭代，通过循环结构就可以消除；另一种方法是利用栈的方式实现。例如， n 的阶乘就是一个简单的递归，可以直接利用迭代就可以消除递归。 n 的阶乘的非递归算法如下。

```
int fact(int n)
{
    int f,i;
    f=1;
    for(i=1;i<=n;i++)
        f=f*i;
    return f;
}
```

当然， n 的阶乘的递归算法也可以转换为利用栈实现的非递归算法。

【例4-7】 编写求n的阶乘的递归算法与利用栈实现的非递归算法。

【分析】 利用栈模拟实现求n的阶乘。定义一个二维数组，数组的第一维用于存放本层参数n，第二维用于存放本层要返回的结果。

当n=3时，递归调用过程如图4-28所示。

在递归函数调用的过程中，各参数入栈情况如图4-29所示。为便于描述，我们用f代替fact表示函数。

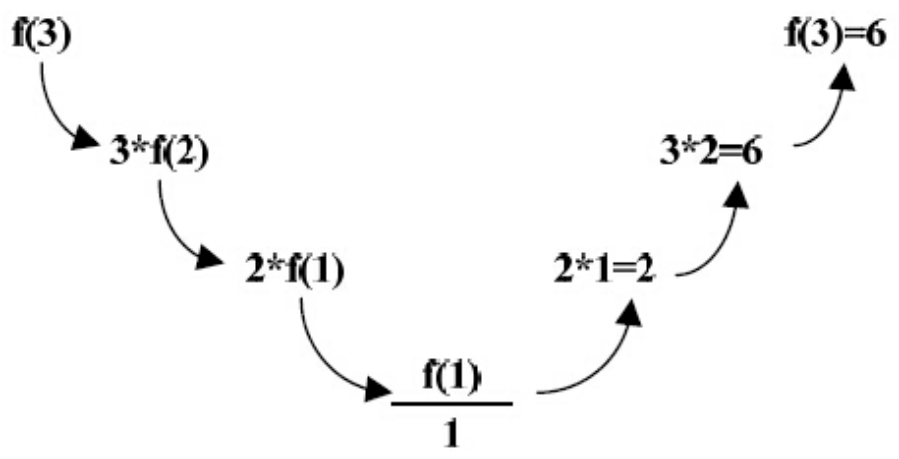


图4-28 递归调用过程



图4-29 递归调用入栈过程

当 $n=1$ 时，递归调用开始逐层返回，参数开始出栈，如图4-30所示。

n 的阶乘递归与非递归算法实现如下。

```
/*
头文件*/
#include<stdio.h>
#define MaxSize 100
/*
递归与非递归函数声明*/
int fact1(int n);
int fact2(int n);
void main()
{
    int f,n;
    printf("
请输入一个正整数 (n<15)
: ");
    scanf("%d",&n);
    printf("
递归实现n
的阶乘:");
    f=fact1(n);
    printf("n!=%4d\n",f);
    f=fact2(n);
    printf("
利用栈非递归实现n
的阶乘:");
    printf("n!=%4d\n",f);
}
int fact1(int n)
/*n
的阶乘递归实现*/
{
    if(n==1) /*
递归函数出口。当n=1
时，开始返回到上一层*/
        return 1;
    else
        return n*fact1(n-1); /*n
的阶乘递归实现。把一个规模为n
的问题转化为n-1
的问题*/
}
int fact2(int n)
/*n
的阶乘非递归实现*/
{
    int s[MaxSize][2],top=-1; /*
定义一个二维数组，并将栈顶指针置为-1*/
    /*
栈顶指针加1
，将工作记录入栈*/
```

```

    top++;
    s[top][0]=n;
记录每一层的参数*/
    s[top][1]=0;
记录每一层的结果返回值*/
    do
    {
        if(s[top][0]==1)
递归出口*/
        {
            s[top][1]=1;
            printf("n=%4d, fact=%4d\n",s[top][0],s[top][1]);
        }
        if(s[top][0]>1&& s[top][1]==0)
通过栈模拟递归的递推过程，将问题依次入栈*/
        {
            top++;
            s[top][0]=s[top-1][0]-1;
            s[top][1]=0;
将结果置为0
，还没有返回结果*/
            printf("n=%4d, fact=%4d\n",s[top][0],s[top][1]);
        }
        if(s[top][1]!=0)
模拟递归的返回过程，将每一层调用的结果返回*/
        {
            s[top-1][1]=s[top][1]*s[top-1][0];
            printf("n=%4d, fact=%4d\n",s[top-1][0],s[top-1][1]);
            top--;
        }
    }while(top>0);
    return s[0][1];
}
```

程序运行结果如图4-31所示。

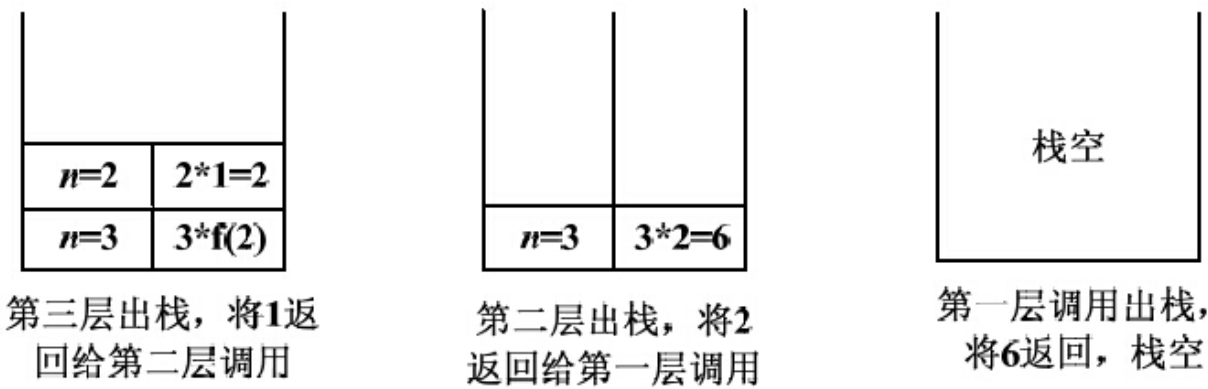
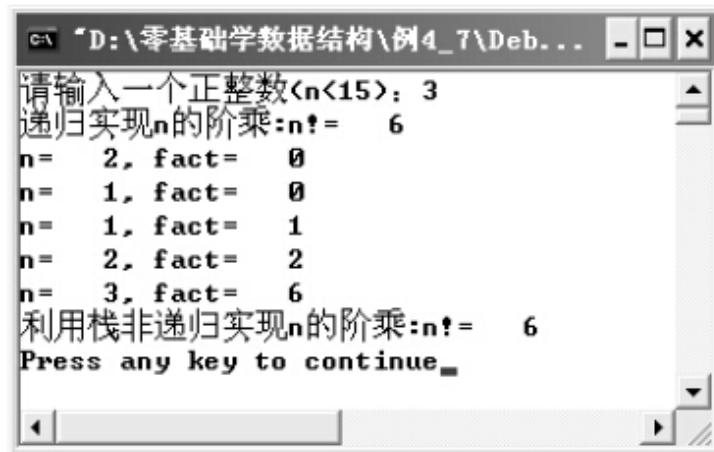


图4-30 递归调用出栈过程



```
C:\ "D:\零基础学数据结构\例4_7\Deb..."
请输入一个正整数(n<15): 3
递归实现n的阶乘:n!= 6
n= 2, fact= 0
n= 1, fact= 0
n= 1, fact= 1
n= 2, fact= 2
n= 3, fact= 6
利用栈非递归实现n的阶乘:n!= 6
Press any key to continue
```

图4-31 n的阶乘程序运行结果

利用栈实现的非递归过程可分为以下几个步骤。

- ①设置一个工作栈，用于保存递归工作记录，包括实参、返回地址等。
- ②将调用函数传递过来的参数和返回地址入栈。
- ③利用循环模拟递归分解过程，逐层将递归过程的参数和返回地址入栈。当满足递归结束条件时，依次逐层退栈，并将结果返回给上一层，直到栈空为止。

4.6 小结

栈是一种只允许在线性表的一端进行插入和删除操作的线性表。

与线性表类似，栈也有顺序存储和链式存储两种存储方式。采用顺序存储结构的栈称为顺序栈，采用链式存储结构的栈称为链栈。

栈的后进先出特性使栈在编译处理等方面发挥了极大的作用。例如，数制转换、括号匹配、表达式求值等正是利用栈的后进先出特性解决的。

递归的调用过程也是系统借助栈的特性实现的。因此，可利用栈模拟递归调用过程，可以设置一个栈，用于存储每一层递归调用的信息，包括实际参数、局部变量及上一层的返回地址等。每进入一层，将工作记录压入栈顶。每退出一层，将栈顶的工作记录弹出。这样就可以将递归转化为非递归，从而消除了递归。

4.7 习题

一、选择题

1. 一个栈的输入序列为a, b, c, d, e, 则栈的不可能输出的序列是 ()。

A. a, b, c, d, e

B. d, e, c, b, a

C. d, c, e, a, b

D. e, d, c, b, a

2. 设计一个判别表达式中括号是否配对的算法, 采用 () 数据结构最佳。

A. 顺序表

B. 链表

C. 队列

D. 栈

3. 表达式 $a * (b + c) - d$ 的后缀表达式是（ ）。

A. $abcd+-$

B. $abc+*d-$

C. $abc*+d-$

D. $-+*abcd$

4. 将递归算法转换成对应的非递归算法时，通常需要使用（ ）来保存中间结果。

A. 队列

B. 栈

C. 链表

D. 树

5. 栈的插入和删除操作在（ ）。

A. 栈底

B. 栈顶

C. 任意位置

D. 指定位置

6. 判定一个顺序栈S（栈空间大小为n）为空的条件是（）。

A. $S \rightarrow \text{top} == 0$

B. $S \rightarrow \text{top} \neq 0$

C. $S \rightarrow \text{top} == n$

D. $S \rightarrow \text{top} \neq n$

二、算法分析题

已知栈的基本操作函数如下。

构造空栈	<code>int InitStack(SqStack *S);</code>	<code>//</code>
判断栈空	<code>int StackEmpty(SqStack *S);</code>	<code>//</code>
入栈	<code>int Push(SqStack *S, ElemType e);</code>	<code>//</code>
出栈	<code>int Pop(SqStack *S, ElemType *e);</code>	<code>//</code>

函数conversion实现十进制数转换为八进制数，请将如下函数补充完整。

```
void conversion(){
    InitStack(S);
    scanf("%d", &N);
    while(N){
        (1
        )
    }
}
```

```

        N=N/8;
    }
    while(
(2
)    ){
        Pop(S,&e);
        printf("%d",e);
    }
}

```

三、算法设计题

1. 建立一个顺序栈。从键盘上输入若干个字符，以回车键结束，实现元素的入栈操作。然后依次输出栈中的元素，实现出栈操作。要求顺序栈结构由栈顶指针、栈底指针和存放元素的数组构成。

2. 建立一个链栈。从键盘上输入若干个字符，以回车键结束，实现元素的入栈操作。然后依次输出栈中的元素，实现出栈操作。

3. 试利用栈的基本操作编写一个行编辑程序，当前一个字符有误时，输入#消除；当前面一行有误时，输入@消除当前行的字符序列。

分析 主要利用栈的后进先出特性，更正行编辑程序的字符序列错误输入。算法思想是：逐个检查输入的字符序列，如果当前的字符不是#和@，则将该字符进栈。如果是字符#，将栈顶的字符出栈。如果当前字符是@，则清空栈。

为了处理错误的输入，可以设置一个栈，当读入一个字符时，如果这个字符不是#或@，将该字符进栈。如果读入的字符是#，将栈顶的

字符出栈。如果读入的字符是@，则将栈清空。

4. 实现n阶汉诺塔的非递归要求保存每一层递归调用的工作记录，需要定义一个栈结构，栈结构定义如下。

```
typedef struct
{
    char x;
    char y;
    char z;
    int flag;
    int num;
}Stack;
```

其中，x、y、z表示3个塔座；flag是一个标志，flag为1时表示需要将大问题分解，为0时表示问题已经变成最小的问题，可以直接移动圆盘；num表示当前的圆盘数。

第5章 队列

与栈一样，队列也是一种操作受限的线性表。队列在操作系统和事务管理等软件设计中应用广泛，如键盘输入缓冲区问题就是利用队列的思想实现的。本章主要介绍队列的定义、队列的顺序存储和链式存储及其应用。

本章重点和难点：

- 队列的顺序表示与实现
- 队列的链式表示与实现

5.1 队列的定义与抽象数据类型

队列只允许在表的一端进行插入操作，在另一端进行删除操作。

5.1.1 什么是队列

队列（queue）是一种先进先出（First In First Out, FIFO）的线性表，它只允许在表的一端进行插入，另一端删除元素。这与日常生活中的排队是一致的，最早进入队列的元素最早离开。在队列中，允许插入的一端称为**队尾**（front），允许删除的一端称为**队头**（rear）。

假设队列为 $q = (a_1, a_2, \dots, a_i, \dots, a_n)$ ，那么 a_1 为队头元素， a_n 为队尾元素。进入队列时，是按照 a_1, a_2, \dots, a_n 的顺序进入的，退出队列时也是按照这个顺序退出的。也就是说，当先进入队列的元素都退出之后，后进入队列的元素才能退出。即只有当 a_1, a_2, \dots, a_{n-1} 都退出队列以后， a_n 才能退出队列。图5-1所示是队列的示意图。

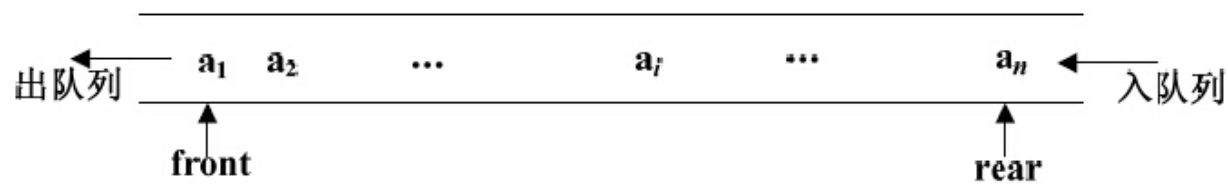


图5-1 队列

例如在日常生活中，人们在医院排队挂号就是一个队列。新来挂号的人到队尾排队，形成新的队尾，即入队，在队首的人挂完号离开，即出队。在程序设计中也经常会遇到排队等待服务的问题。一个典型的例子就是操作系统中的多任务处理。在计算机系统中，同时有几个任务等待输出，那么就要按照请求输出的先后顺序进行输出。

5.1.2 队列的抽象数据类型

1. 数据对象集合

队列的数据对象集合为 $\{a_1, a_2, \dots, a_n\}$ ，每个元素都具有相同的数据类型DataType。

队列中的数据元素之间也是一对一的关系。除第一个元素 a_1 外，每一个元素有且只有一个直接前驱元素；除最后一个元素 a_n 外，每一个元素有且只有一个直接后继元素。

2. 基本操作集合

- InitQueue (&Q)：初始化操作，建立一个空队列Q。这就像日常生活中医院新增一个挂号窗口，前来看病的人就可以排队在这里挂号看病。

- QueueEmpty (Q) : 若Q为空队列, 返回1, 否则返回0。这就类似于挂号窗口前是否还有人排队挂号。

- EnQueue (&Q, e) : 插入元素e到队列Q的队尾。这就像前来挂号的人都要到队列的最后排队挂号。

- DeQueue (&Q, &e) : 删除Q的队首元素, 并用e返回其值。这就像排在最前面的人挂完号离开队列。

- Gethead (Q, &e) : 用e返回Q的队首元素。这就像询问排队挂号的人是谁一样。

- ClearQueue (&Q) : 将队列Q清空。这就像所有排队的人都挂完了号离开队列。

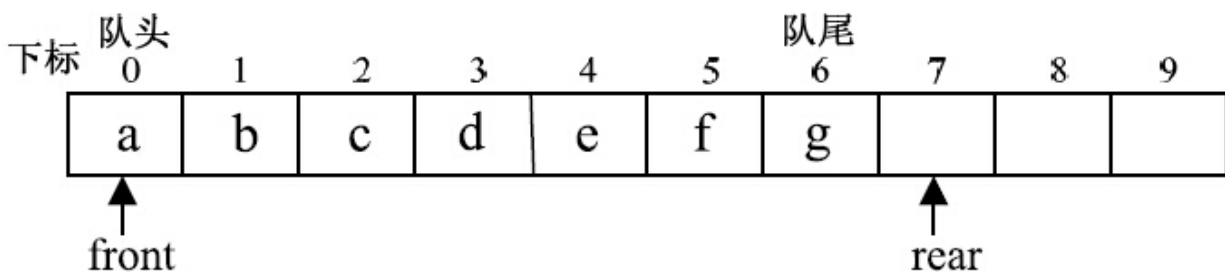


图5-2 顺序队列

在使用队列前，先初始化队列，此时，队列为空，队头指针front和队尾指针rear都指向队列的第一个位置，即 $front=rear=0$ ，如图5-3所示。

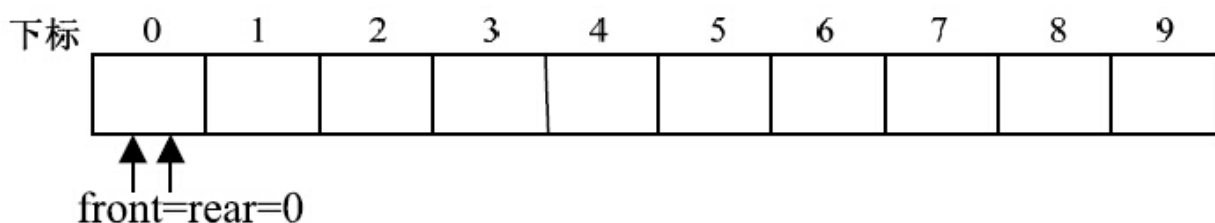


图5-3 顺序队列为空

每一个元素进入队列，队尾指针rear就会增1，若元素a、b、c依次进入空队列，front指向第一个元素，rear指向下标为3的存储单元，如图5-4所示。

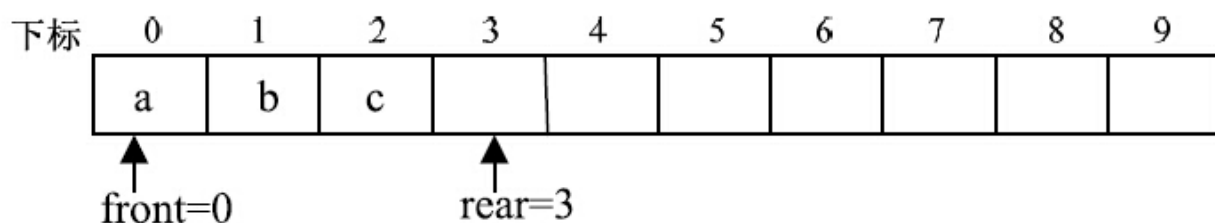


图5-4 插入3个元素后的顺序队列

当一个元素出队列时，队头指针front增1，队头元素即a出队后，front向后移动一个位置，指向下一个位置，rear不变，如图5-5所示。

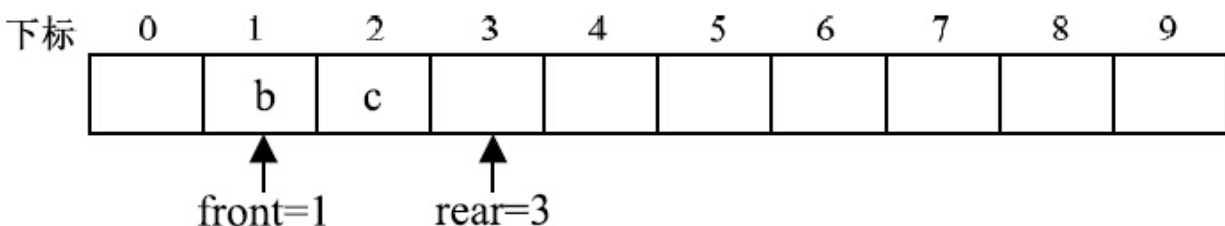


图5-5 删除队头元素a后的顺序队列

注意 在非空队列中，队头指针front指向队头元素的位置，队尾指针rear指向队尾元素的下一个位置；队满指的是元素占据了队列中的所有存储空间，没有空闲的存储空间可以插入元素；队空指的是队列中没有一个元素，也叫空队列。

5.2.2 顺序队列的“假溢出”

在对顺序队列进行插入和删除操作的过程中，可能会出现“假溢出”现象。经过多次插入和删除操作后，实际上队列还有存储空间，但是又无法向队列中插入元素，我们将这种溢出称为“假溢出”。

例如在图5-2所示的队列中插入3个元素h、i、j，然后删除2个元素a、b，就会出现如图5-6所示的情况。当插入元素j后，队尾指针rear将越出数组的下界而造成“假溢出”。

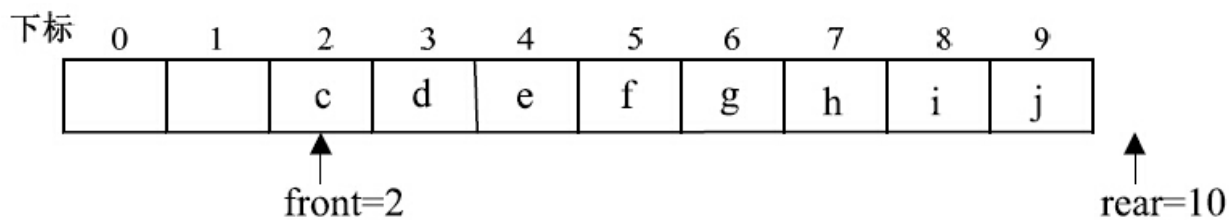


图5-6 插入元素h、i、j和删除元素a、b后的“假溢出”

5.2.3 顺序循环队列的表示

为了避免出现顺序队列的“假溢出”，通常采用顺序循环队列实现队列的顺序存储。

1. 顺序循环队列

为了充分利用存储空间，消除这种“假溢出”现象，当队尾指针rear和队头指针front到达存储空间的最大值（假定队列的存储空间为QueueSize）的时候，让队尾指针和队头指针转化为0，这样就可以将元素插入到队列还没有利用的存储单元中。例如在图5-6中插入元素j之后，rear将变为0，可以继续将元素插入下标为0的存储单元中。这样就把顺序队列使用的存储空间构造成一个逻辑上首尾相连的循环队列。

当队尾指针rear达到最大值QueueSize-1时，前提是队列中还有存储空间，若要插入元素，就要把队尾指针rear变为0；当队头指针front达到最大值QueueSize-1时，若要将队头元素出队，要让队头指针front变为0。这可通过取余操作实现队列的首位相连。例如，假设QueueSize=10，当队尾指针rear=9时，若要将新元素入队，则先令

$\text{rear} = (\text{rear} + 1) \% 10 = 0$ ，然后将元素存入队列的第0号单元，通过取余操作实现了队列的逻辑上的首尾相连。

2. 顺序循环队列的队空和队满判断

但是，在顺序循环队列在队空和队满的情况下，队头指针 front 和队尾指针 rear 同时都会指向同一个位置，即 $\text{front} == \text{rear}$ ，如图5-7所示。即队列为空时，有 $\text{front} = 0$ 、 $\text{rear} = 0$ ，因此 $\text{front} == \text{rear}$ ；队满时也有 $\text{front} = 0$ 、 $\text{rear} = 0$ ，因此 $\text{front} == \text{rear}$ 。

为了区分这队空还是队满，通常采用如下两个方法。

(1) 增加一个标志位。设这个标志位为 flag ，初始时，有 $\text{flag} = 0$ ；当入队列成功，则 $\text{flag} = 1$ ；出队列成功，有 $\text{flag} = 0$ 。则队列为空的判断条件为 $\text{front} == \text{rear} \ \&\& \ \text{flag} == 0$ ，队列满的判断条件为 $\text{front} == \text{rear} \ \&\& \ \text{flag} == 1$ 。

(2) 少用一个存储单元。队空的判断条件为 $\text{front} == \text{rear}$ ，队满的判断条件为 $\text{front} == (\text{rear} + 1) \% \text{QueueSize}$ 。那么，入队的操作语句为 $\text{rear} = (\text{rear} + 1) \% \text{QueueSize}$ ， $\text{Q}[\text{rear}] = \text{x}$ 。出队的操作语句为 $\text{front} = (\text{front} + 1) \% \text{QueueSize}$ 。少用一个存储单元的顺序循环队列队满情况如图5-8所示。

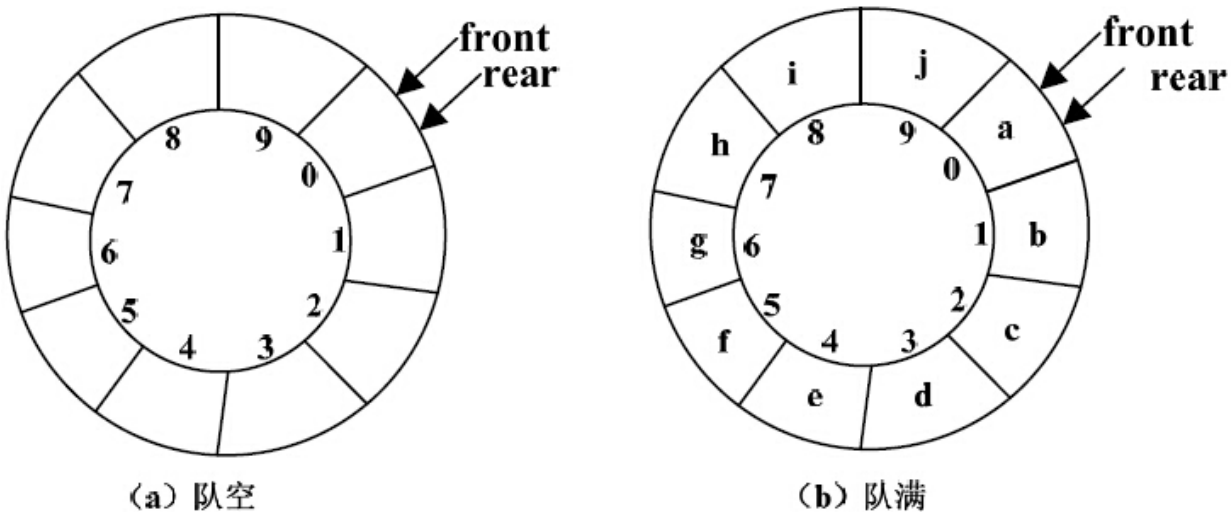


图5-7 顺序循环队列队空和队满状态

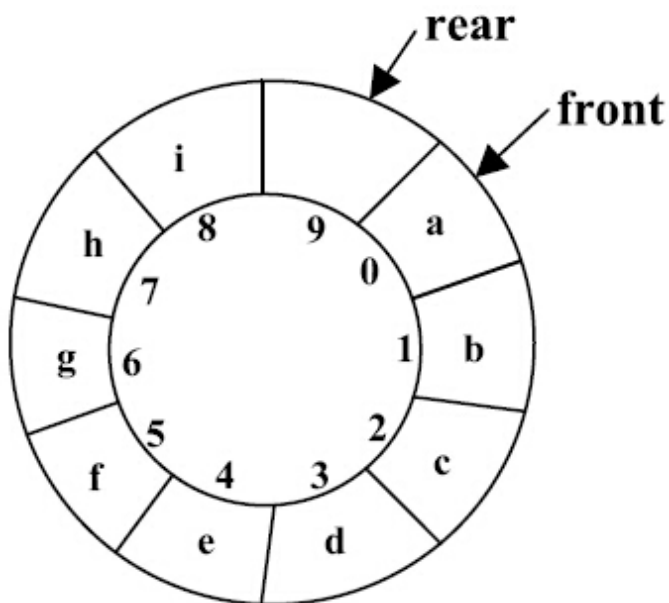


图5-8 少用一个存储单元的顺序循环队列队满状态

顺序循环队列类型描述如下。

```

#define QueueSize 60          /*
队列的容量*/
typedef struct Squeue{
    DataType queue[QueueSize];
    int front, rear;          /*
队头指针和队尾指针*/
}SeqQueue;

```

其中，queue用来存储队列中的元素，front和rear分别表示队头指针和队尾指针，取值范围为 $0 \sim \text{QueueSize}$ 。

顺序循环队列的主要操作说明如下。

(1) 初始化时，设置 $\text{SQ.front} = \text{SQ.rear} = 0$ 。

(2) 循环队列队空的条件为 $\text{SQ.front} == \text{SQ.rear}$ ，队满的条件为 $\text{SQ.front} == (\text{SQ.rear} + 1) \% \text{QueueSize}$ 。

(3) 入队操作时，先判断队列是否已满，若队列未满，则将元素值e存入队尾指针指向的存储单元，然后将队尾指针加1后取模。

(4) 出队操作时，先判断队列是否为空，若队列不空，则先把队头指针指向的元素值赋给e，即取出队头元素，然后将队头指针加1后取模。

(5) 循环队列的长度为 $(\text{SQ.rear} + \text{QueueSize} - \text{SQ.front}) \% \text{QueueSize}$ 。

注意 对于顺序循环队列中的入队操作和出队操作，front和rear移动时都要进行取模运算，以避免“假溢出”。

5.2.4 顺序循环队列的基本运算

顺序循环队列的基本运算算法实现如下（以下算法的实现保存在文件SeqQueue.h中）。

①初始化队列，代码如下。

```
void InitQueue(SeqQueue *SCQ)
/*
顺序循环队列的初始化*/
{
    SCQ->front=SCQ->rear=0;    /*
把队头指针和队尾指针同时置为0*/
}
```

②判断队列是否为空。若队头指针与队尾指针相等，则队列为空；否则队列不为空。算法实现如下。

```
int QueueEmpty(SeqQueue SCQ)
/*
判断顺序循环队列是否为空，队列为空返回1
，否则返回0*/
{
    if(SCQ.front== SCQ.rear)    /*
当顺序循环队列为空时*/
        return 1;    /*
返回1*/
    else    /*
否则*/
        return 0;    /*
返回0*/
}
```

③将元素e入队。在将元素入队（即把元素插入到队尾）之前，先判断队列是否已满。如果队列未满，则执行插入运算，然后队尾指针加1把队尾指针向后移动。入队操作的算法实现如下。

```
int EnQueue(SeqQueue *SCQ,DataType e)
/*
将元素e
插入到顺序循环队列SQ
中，插入成功返回1
，否则返回0*/
```

```

{
    if (SCQ->front== (SCQ->rear+1)%QueueSize)
        /*
在插入新的元素之前，判断队尾指针是否到达数组的最大值，即是否上溢*/
        return 0;
    SCQ->queue[SCQ->rear]=e;
    /*
在队尾插入元素e */
    SCQ->rear=(SCQ->rear+1)%QueueSize;
    /*
队尾指针向后移动一个位置*/
    return 1;
}

```

④将队头元素出队。在队头元素出队（即删除队头元素）之前，先判断队列是否为空。若队列不空，则删除队头元素，然后将队头指针向后移动，使其指向下一个元素。出队操作的算法实现如下。

```

int DeQueue(SeqQueue *SCQ,DataType *e)
/*
将队头元素出队，并将该元素赋值给e
，删除成功返回1
，否则返回0*/
{
    if (SCQ->front==SCQ->rear)
        /*
在删除元素之前，判断顺序循环队列是否为空*/
        return 0;
    else
    {
        *e=SCQ->queue[SCQ->front];
        /*
将要删除的元素赋值给e*/
        SCQ->front=(SCQ->front+1)%QueueSize;
        /*
将队头指针向后移动一个位置，指向新的队头*/
        return 1;
    }
}

```

⑤取队头元素。先判断顺序循环队列是否为空，如果队列为空，则返回0表示取队头元素失败；否则，把队头元素赋给e，并返回1表示取队头元素成功。取队头元素的算法实现如下。

```

int GetHead (SeqQueue SCQ,DataType *e)
/*
取队头元素，并将该元素赋值给e
，取元素成功返回1
，否则返回0*/
{
    if (SCQ.front==SCQ.rear)
        /*
在取队头元素之前，判断顺序循环队列是否为空*/

```



```
        return 0;
    else
    {
        *e=SCQ.queue[SCQ.front];           /*
将队头元素赋值给e
，取出队头元素*/
        return 1;
    }
}
```

⑥清空队列，算法实现如下。

```
void ClearQueue(SeqQueue *SCQ)
/*
清空队列*/
{
    SCQ->front=SCQ->rear=0;           /*
将队头指针和队尾指针都置为0*/
}
```

5.2.5 顺序循环队列举例

【例5-1】 假设在周末舞会上，男士们和女士们进入舞厅时，各自排成一队。跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。若两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲。现要求写一算法模拟上述舞伴配对问题。

【分析】 先入队的男士或女士先出队配成舞伴。因此该问题具有典型的先进先出特性，可用队列作为算法的数据结构。

在算法实现时，假设男士和女士的记录存放在一个数组中作为输入，然后依次扫描该数组的各元素，并根据性别来决定是进入男队还是女队。当这两个队列构造完成之后，依次将两队当前的队头元素出

队来配成舞伴，直至某队列变空为止。此时，若某队仍有等待配对者，算法输出此队列中等待者的人数及排在队头的等待者的名字，他（或她）将是下一轮舞曲开始时第一个可获得舞伴的人。

舞伴问题实现代码如下。

```
#include<stdio.h>
typedef struct{
    char name[20];
    char sex; /*
性别, 'F'
表示女性, 'M'
表示男性*/
}Person;
typedef Person DataType; /*
将队列中元素的数据类型改为Person */
#include"SeqQueue.h"
void DancePartner(DataType dancer[],int num)
/*
结构数组dancer
中存放跳舞的男女, num
是跳舞的人数*/
{
    int i;
    DataType p;
    SeqQueue Mdancers,Fdancers;
    InitQueue(&Mdancers);/*
男士队列初始化*/
    InitQueue(&Fdancers);/*
女士队列初始化*/
    for(i=0;i<num;i++){/*
依次将跳舞者依其性别入队*/
        p=dancer[i];
        if(p.sex=='F')
            EnQueue(&Fdancers,p); /*
排入女队*/
        else
            EnQueue(&Mdancers,p); /*
排入男队*/
    }
    printf("
配对成功的舞伴分别是: \n");
    while(!QueueEmpty(Fdancers)&&!QueueEmpty(Mdancers)){
        /*
依次输入男女舞伴名*/
        DeQueue(&Fdancers,&p); /*
女士出队*/
        printf("%s  ",p.name);/*
打印出队女士名*/
        DeQueue(&Mdancers,&p); /*
男士出队*/
        printf("%s\n",p.name); /*
打印出队男士名*/
    }
    if(!QueueEmpty(Fdancers)){ /*
输出女士剩余人数及队头女士的名字*/
```

```

        printf("
还有%d
名女士等待下一轮舞曲.\n",DancerCount(Fdancers));
        GetHead(Fdancers,&p);    /*
取队头*/
        printf("%s
将在下一轮中最先得到舞伴.\n",p.name);
    }
    else if(!QueueEmpty(Mdancers)){/*
输出男队剩余人数及队头者名字*/
        printf("
还有%d
名男士等待下一轮舞曲.\n",DancerCount(Mdancers));
        GetHead(Mdancers,&p);
        printf("%s
将在下一轮中最先得到舞伴.\n",p.name);
    }
}
int DancerCount(SeqQueue Q)
/*
队列中等待配对的人数*/
{
    return (Q.rear-Q.front+QueueSize)%QueueSize;
}
void main()
{
    int i,n;
    DataType dancer[30];
    printf("
请输入舞池中排队的人数:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("
姓名:");
        scanf("%s",dancer[i].name);
        getchar();
        printf("
性别:");
        scanf("%c",&dancer[i].sex);
    }
    DancePartner(dancer,n);
}

```

程序的运行结果如图5-9所示。

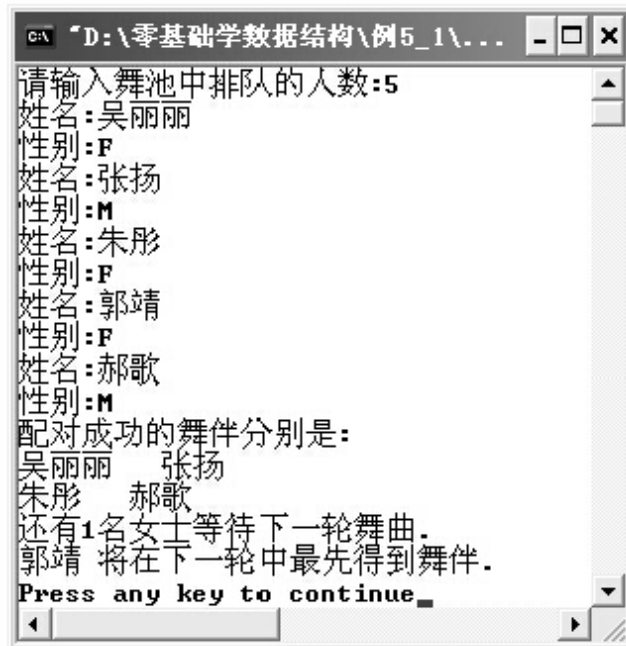


图5-9 顺序循环队列程序运行结果

5.3 队列的链式存储及实现

采用链式存储的队列称为**链式队列** 或**链队列**。链式队列在插入和删除过程中，不需要移动大量的元素，只需要改变指针的位置即可。本节主要介绍链式队列的表示、链式队列的实现和链式队列的应用。

5.3.1 链式队列的表示

顺序队列在插入和删除操作过程中需要移动大量元素，这样算法的效率会比较低，为了避免以上问题，可采用链式存储结构表示队列。

1. 链式队列

链式队列 通常用链表实现。一个链队列显然需要两个分别指示队头和队尾的指针（分别称为队头指针和队尾指针）才能唯一确定。这里，与单链表一样，为了操作方便，给链队列添加一个头结点，并令队头指针front指向头结点，用队尾指针rear指向最后一个结点。一个不带头结点的链式队列和带头结点的链队列分别如图5-10、图5-11所示。

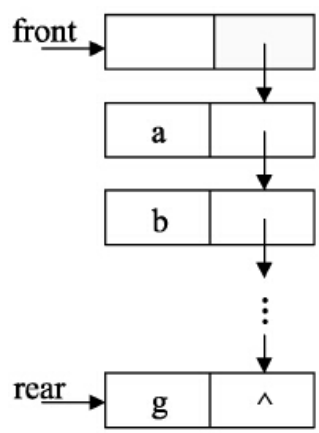


图5-10 不带头结点的链式队列

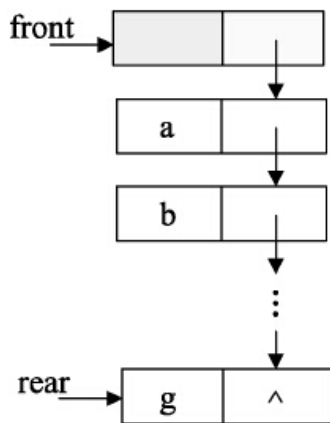


图5-11 带头结点的链式队列

对于带头结点的链式队列，当队列为空时，队头指针front和队尾指针rear都指向头结点，如图5-12所示。

链式队列中，插入和删除操作只需要移动队头指针和队尾指针，这两种操作的指针变化如图5-13、图5-14和图5-15所示。图5-13表示在队列中插入元素a的情况，图5-14表示队列中插入了元素a、b、c之后的情况，图5-15表示元素a出队列的情况。

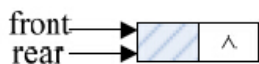


图5-12 带头结点的空链式队列

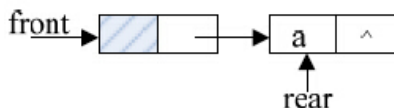


图5-13 在链式队列中插入一个元素a

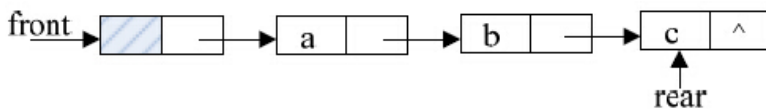


图5-14 在链式队列中插入一个元素c

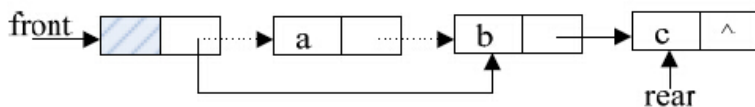


图5-15 在链式队列中删除一个元素a

链式队列的类型描述如下。

```
/*
结点类型定义*/
typedef struct QNode
{
    DataType data;
    struct QNode* next;
}LQNode,*QueuePtr;
/*
队列类型定义*/
typedef struct
{
    QueuePtr front;
    QueuePtr rear;
}LinkQueue;
```

2. 链式循环队列

将链式队列的首尾相连就构成了链式循环队列。在链式循环队列中，可以只设置队尾指针，如图5-16所示。当队列为空时，如图5-17所示，队列LQ为空的判断条件为LQ.rear->next==LQ.rear。

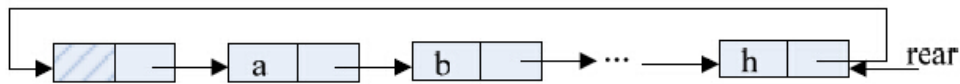


图5-16 链式循环队列

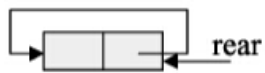


图5-17 空链式循环队列

5.3.2 链式队列的基本运算

链式队列的基本运算算法实现如下（以下队列基本操作实现代码保存在文件“LinkQueue.h”中）。

①初始化队列，算法实现如下。

```
void InitQueue(LinkQueue *LQ)
/*
初始化链式队列*/
{
    LQ->front=LQ->rear=(LQNode*)malloc(sizeof(LQNode));
    if(LQ->front==NULL) exit(-1);
```

```

    LQ->front->next=NULL;          /*
把头结点的指针域置为为0*/
}

```

②判断队列是否为空，算法实现如下。

```

int QueueEmpty(LinkQueue LQ)
/*
判断链式队列是否为空，队列为空返回1
，否则返回0*/
{
    if (LQ.rear->next==NULL)        /*
当链式队列为空时*/
        return 1;                    /*
返回1*/
    else                             /*
否则*/
        return 0;                    /*
返回0*/
}

```

③将元素e入队。先为新结点申请一个空间，然后将e赋给数据域，并使原队尾元素结点的指针域指向新结点，队尾指针指向新结点，从而将结点加入队列中。操作过程如图5-18所示。

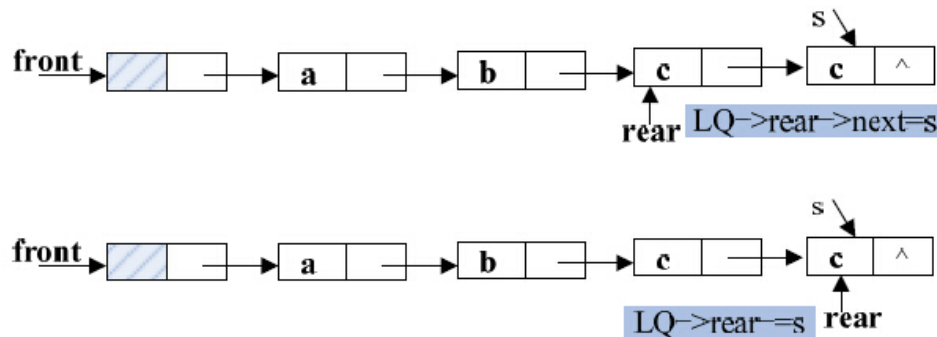


图5-18 将元素e入队的操作过程

将元素e入队的算法实现如下。

```

int EnQueue(LinkQueue *LQ,DataType e)
/*
将元素e
插入到链式队列LQ
中，插入成功返回1*/
{
    LQNode *s;
    s=(LQNode*)malloc(sizeof(LQNode)); /*
为将要入队的元素申请一个结点的空间*/
    if (!s) exit(-1);                  /*
如果申请空间失败，则退出并返回参数-1*/
    s->data=e;                          /*
将元素值赋值给结点的数据域*/
    s->next=NULL;                       /*
将结点的指针域置为空*/
    LQ->rear->next=s;                   /*
将原来队列的队尾指针指向s*/
}

```



```
    LQ->rear=s;
    将队尾指针指向s*/
    return 1;
}
```

④将队头元素出队。删除队头元素时，应首先通过队头指针和队尾指针是否相等判断队列是否已空。若队列非空，则删除队头元素，然后将指向队头元素的指针向后移动，使其指向下一个元素。将队头元素出队的算法实现如下。

```
int DeQueue(LinkQueue *LQ,DataType *e)
/*
删除链式队列中的队头元素，并将该元素赋给e
，删除成功返回1
，否则返回0*/
{
    LQNode *s;
    if (LQ->front==LQ->rear) /*
在删除元素之前，判断链式队列是否为空*/
        return 0;
    else
    {
        s=LQ->front->next; /*
使指针s
指向队头元素的指针*/
        *e=s->data; /*
将要删除的队头元素赋给e*/
        LQ->front->next=s->next; /*
使头结点的指针指向指针s
的下一个结点*/
        if (LQ->rear==s) LQ->rear=LQ->front; /*
如果要删除的结点是队尾，则使队尾指针指向队头指针*/
        free(s); /*
释放指针s
指向的结点空间*/
        return 1;
    }
}
```

⑤取队头元素，算法实现如下。

```
int GetHead (LinkQueue LQ,DataType *e)
/*
取链式队列中的队头元素，并将该元素赋给e
，取元素成功返回1
，否则返回0*/
{
    LQNode *s;
    if (LQ.front==LQ.rear) /*
在取队头元素之前，判断链式队列是否为空*/
        return 0;
    else
    {
        s=LQ.front->next; /*
将指针p
指向队列的第一个元素即队头元素*/
        *e=s->data; /*
将队头元素赋给e
，取出队头元素*/
        return 1;
    }
}
```

⑥清空队列。在使用完队列之后，需要将链式队列中的结点空间全部释放。先将队尾指针指向队头结点的下一个结点，再释放队头指针指向的结点，然后将队头指针指向队尾指针，重复执行以上过程，就可用释放掉所有结点空间。清空队列的算法实现如下。

```
void ClearQueue(LinkQueue *LQ)
/*
清空队列*/
{
    while (LQ->front!=NULL)
    {
        LQ->rear=LQ->front->next;    /*
队尾指针指向队头指针指向的下一个结点*/
        free (LQ->front);            /*
释放队头指针指向的结点*/
        LQ->front=LQ->rear;          /*
队头指针指向队尾指针*/
    }
}
```

5.3.3 链式队列举例

【例5-2】 编写一个算法，判断任意给定的字符序列是否为回文。所谓回文是指一个字符序列以中间字符为基准两边字符完全相同，即顺着看和倒着看是相同的字符序列。例如，字符序列“XYZMTATMZYX”为回文，而字符序列“XYZBZXY”不是回文。

【分析】 这个题目是典型的考查栈和队列的应用，可通过构造栈和队列实现。可先把一个字符序列分别存入队列和栈，然后将字符出队列和出栈，比较出队列的字符和出栈的字符是否相等，若相等，则继续取出队列和栈中的下一个字符进行比较，直到栈和队列为空，表明该字符序列为回文；若有字符不相等，则该字符序列不是回文。

具体实现时，采用链栈和只有队尾指针的链式循环队列作为存储结构，算法实现如下。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<malloc.h>
typedef char DataType; /*
数据类型为字符类型*/
/*
链式堆栈结点类型定义*/
typedef struct snode
{
    DataType data;
    struct snode *next;
}LSNode;
/*
只有队尾指针的链式循环队列类型定义*/
```

```

typedef struct QNode
{
    DataType data;
    struct QNode *next;
}LQNode,*LinkQueue;
void InitStack(LSNode **head)
/*
带头结点的链式堆栈初始化*/
{
    if(((*head=(LSNode*)malloc(sizeof(LSNode)))==NULL)/*
为头结点分配空间*/
    {
        printf("
分配结点不成功");
        exit(-1);
    }
    else
        (*head)->next=NULL;
/*
头结点的指针域设置为空*/
}
int StackEmpty(LSNode *head)
/*
判断带头结点链式堆栈是否为空。如果堆栈为空，返回1
，否则返回0*/
{
    if(head->next==NULL)
/*
如果堆栈为空，返回1
，否则返回0*/
        return 1;
    else
        return 0;
}
int PushStack(LSNode *head,DataType e)
/*
链式堆栈进栈。进栈成功返回1
，否则退出*/
{
    LSNode *s;
    if((s=(LSNode*)malloc(sizeof(LSNode)))==NULL)/*
为结点分配空间，失败退出程序并返回-1*/
        exit(-1);
    else
    {
        s->data=e;
/*
把元素值赋值给结点的数据域*/
        s->next=head->next;
/*
将结点插入到栈顶*/
        head->next=s;
        return 1;
    }
}
int PopStack(LSNode *head,DataType *e)
/*
链式堆栈出栈，需要判断堆栈是否为空。出栈成功返回1
，否则返回0*/
{
    LSNode *s=head->next;
/*
指针s
指向栈顶结点*/
    if(StackEmpty(head))
/*
判断堆栈是否为空*/
        return 0;
    else
    {
        head->next=s->next;
/*
头结点的指针指向第二个结点位置*/
        *e=s->data;
/*
要出栈的结点元素赋值给e*/
        free(s);
/*
释放要出栈的结点空间*/
        return 1;
    }
}
void InitQueue(LinkQueue *rear)
/*
将带头结点的链式循环队列初始化为空队列，需要把头结点的指针指向头结点*/
{
    if(((*rear=(LQNode*)malloc(sizeof(LQNode)))==NULL)
        exit(-1);
/*
如果申请结点空间失败退出*/
    else
        (*rear)->next=*rear;
/*
队尾指针指向头结点*/
}

```

```

int QueueEmpty(LinkQueue rear)
/*
判断链式队列是否为空，队列为空返回1
，否则返回0*/
{
    if (rear->next==rear) /*
判断队列是否为空。当队列为空时，返回1
，否则返回0*/
        return 1;
    else
        return 0;
}
int EnQueue(LinkQueue *rear,DataType e)
/*
将元素e
插入到链式队列中，插入成功返回1*/
{
    LQNode *s;
    s=(LQNode*)malloc(sizeof(LQNode)); /*
为将要入队的元素申请一个结点的空间*/
    if(!s) exit(-1); /*
如果申请空间失败，则退出并返回参数-1*/
    s->data=e; /*
将元素值赋值给结点的数据域*/
    s->next=(*rear)->next; /*
将新结点插入链式队列*/
    (*rear)->next=s;
    *rear=s; /*
修改队尾指针*/
    return 1;
}
int DeQueue(LinkQueue *rear,DataType *e)
/*
删除链式队列中的队头元素，并将该元素赋值给e
，删除成功返回1
，否则返回0*/
{
    LQNode *f,*p;
    if(*rear==(*rear)->next) /*
在删除队头元素即出队列之前，判断链式队列是否为空*/
        return 0;
    else
    {
        f=(*rear)->next; /*
使指针f
指向头结点*/
        p=f->next; /*
使指针p
指向要删除的结点*/
        if(p==*rear) /*
处理队列中只有一个结点的情况*/
        {
            *rear=(*rear)->next; /*
使指针rear
指向头结点*/
            (*rear)->next=*rear;
        }
        else
            f->next=p->next; /*
使头结点指向要出队列的下一个结点*/
        *e=p->data; /*
把队头元素值赋值给e*/
        free(p); /*
释放指针p
指向的结点*/
        return 1;
    }
}
void main()
{
    LinkQueue LQueue1,LQueue2; /*
定义链式循环队列*/
    LNode *LStack1,*LStack2; /*
定义链式堆栈*/
    char str1[]="XYZMTATMZYX"; /*
回文字符序列1*/
    char str2[]="XYZBZXY"; /*
回文字符序列2*/
    char q1,s1,q2,s2;
    int i;
    InitQueue(&LQueue1); /*
初始化链式循环队列1*/
    InitQueue(&LQueue2); /*
初始化链式循环队列2*/
}

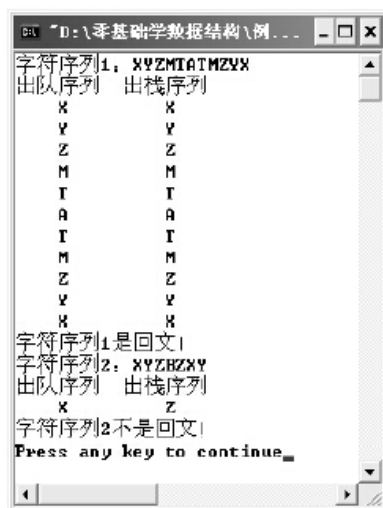
```

```

        InitStack(&LStack1);                                /*
初始化链式堆栈1*/
        InitStack(&LStack2);                                /*
初始化链式堆栈2*/
        for(i=0;i<strlen(str1);i++)
        {
            EnQueue(&LQueue1,str1[i]);                      /*
依次把字符序列1
入队*/
            EnQueue(&LQueue2,str2[i]);                      /*
依次把字符序列2
入队*/
            PushStack(LStack1,str1[i]);                      /*
依次把字符序列1
进栈*/
            PushStack(LStack2,str2[i]);                      /*
依次把字符序列2
进栈*/
        }
        printf("
字符序列1
: %s\n",str1);
        printf("
出队序列
出栈序列\n");
        while(!StackEmpty(LStack1))                        /*
判断堆栈1
是否为空*/
        {
            DeQueue(&LQueue1,&q1);                          /*
字符序列依次出队,并把出队元素赋值给q*/
            PopStack(LStack1,&s1);                          /*
字符序列出栈,并把出栈元素赋值给s*/
            printf("%5c",q1);                                /*
输出字符序列1*/
            printf("%10c\n",s1);
            if(q1!=s1)                                       /*
判断字符序列1
是否是回文*/
            {
                printf("
字符序列1
不是回文! ");
                return;
            }
        }
        printf("
字符序列1
是回文! \n");
        printf("
字符序列2
: %s\n",str2);
        printf("
出队序列
出栈序列\n");
        while(!StackEmpty(LStack2))                        /*
判断堆栈2
是否为空*/
        {
            DeQueue(&LQueue2,&q2);                          /*
字符序列依次出队,并把出队元素赋值给q*/
            PopStack(LStack2,&s2);                          /*
字符序列出栈,并把出栈元素赋值给s*/
            printf("%5c",q2);                                /*
输出字符序列2*/
            printf("%10c\n",s2);
            if(q2!=s2)                                       /*
判断字符序列2
是否是回文*/
            {
                printf("
字符序列2
不是回文! \n");
                return;
            }
        }
        printf("
字符序列2
是回文! \n");
    }

```

程序运行结果如图5-19所示。



```
D:\基础学数据结构\例...
字符序列1: XYZMIAIMZVX
出队序列 出栈序列
X      X
Y      Y
Z      Z
M      M
I      I
A      A
I      I
M      M
Z      Z
Y      Y
X      X
字符序列1是回文!
字符序列2: XYZBZXY
出队序列 出栈序列
X      Z
字符序列2不是回文!
Press any key to continue
```

图5-19 回文判断程序运行结果

5.4 双端队列

双端队列和栈、队列一样，也是一种操作受限的线性表。本节主要介绍双端队列的定义和双端队列的应用。

5.4.1 什么是双端队列

双端队列是限定插入和删除操作在表两端进行的线性表。这两端分别称为端点1和端点2。双端队列可以在队列的任何一端进行插入和删除操作，而一般的队列要求在一端插入元素，在另一端删除元素。一个双端队列如图5-20所示。

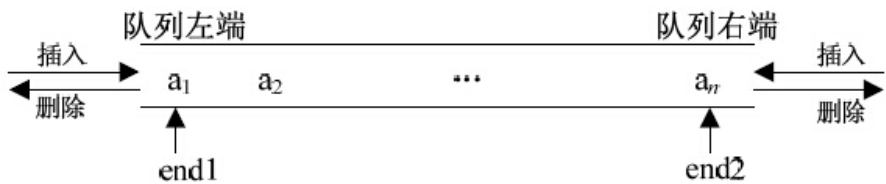


图5-20 双端队列

在图5-20中，可以在队列的左端或右端插入元素，也可以在队列的左端或右端删除元素。其中，end1和end2分别是双端队列的指针。

在实际应用中，还有输入受限和输出受限的双端队列。所谓输入受限的双端队列指的是只允许在队列的一端插入元素，而两端都能删除元素的队列。所谓输出受限的双端队列指的是只允许在队列的一端删除元素，两端都能输入元素的队列。

5.4.2 双端队列的应用

采用一个一维数组作为双端队列的数据存储结构，试编写入队算法和出队算法。双端队列为空的状态如图5-21所示。

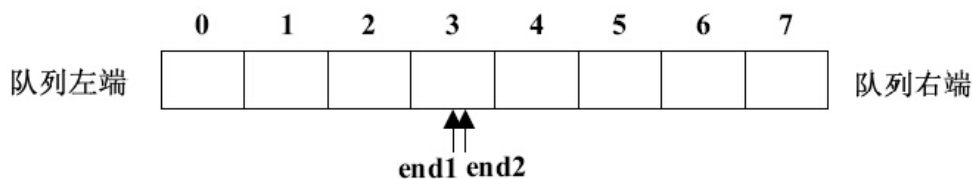


图5-21 双端队列的初始状态（队列为空）

在实际操作过程中，用循环队列实现双端队列的操作是比较恰当的。元素a、b、c依次进入左端的队列，元素e、f依次进入右端的队列，如图5-22所示。

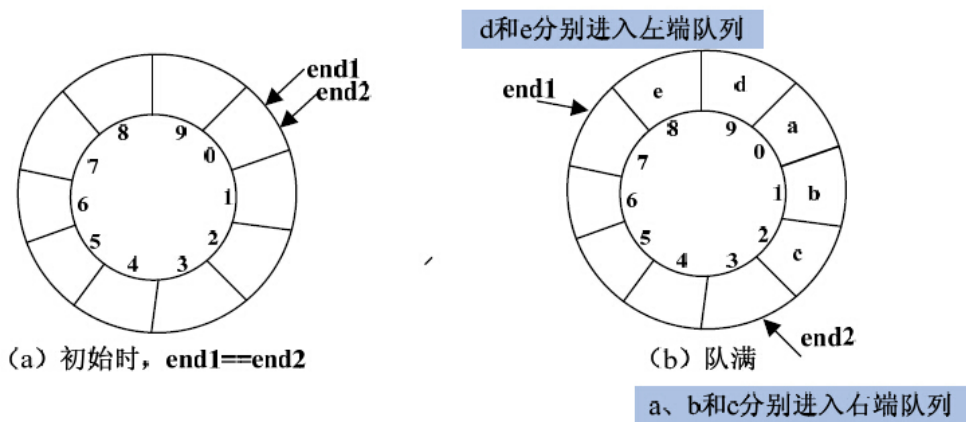


图5-22 双端队列插入元素之后

注意 双端队列虽然是两个队列共享一个存储空间，但是每个队列只有一个指针，在上述实现的过程中，一般情况下，要求仅在一端进行插入和删除操作，这一点是与一般的队列是有差别的。

【例5-3】 编写算法，实现双端队列的入队和出队操作，要求如下。

- (1) 当队列满时，最多只能有一个存储空间为空。
- (2) 在进行插入和删除元素时，队列中的其他元素不动。

【分析】 设双端队列为Q，初始时，队列为空，有 $Q.end1 == Q.end2$ ，队满的条件为 $(Q.end1 - 1 + QueueSize) \% QueueSize == Q.end2$ 或 $(Q.end2 + 1 + QueueSize) \% QueueSize == Q.end1$ 。对于左端的队列，当元素入队时，需要执行 $Q.end-1$ 操作；当元素

出队列时，需要执行Q.end1+1操作。对于右端的队列，当元素入队时，需要执行Q.end2+1操作；当元素出队列时，需要执行Q.end2-1操作。

双端队列的入队和出队算法实现如下。

```

#define QueueSize 10                                /*
定义双端队列的大小*/
typedef char DataType;                               /*
定义数据类型为字符类型*/
typedef struct DQueue                                /*
双端队列的类型定义*/
{
    DataType queue[QueueSize];
    int end1,end2;                                   /*
双端队列的队尾指针*/
}DQueue;
int EnQueue(DQueue *DQ,DataType e,int tag)
/*
将元素e
插入到双端队列中。如果成功返回1
， 否则返回0*/
{
    switch(tag)
    {
        case 1:                                     /*1
表示左端的队列元素入队*/
            if ((DQ->end1+QueueSize)%QueueSize!=DQ->end2) /*
判断队列是否已满*/
            {
                DQ->queue[DQ->end1]=e;                /*
元素e
入队*/
                DQ->end1=(DQ->end1-1+ QueueSize)%QueueSize; /*
移动队列指针*/
                return 1;
            }
            else
                return 0;

        case 2:                                     /*2
表示右端队列的元素入队*/
            if ((DQ->end2+1+QueueSize)%QueueSize!=DQ->end1) /*
判断队列是否已满*/
            {
                DQ->queue[DQ->end2]=e;                /*
元素e
入队*/
                DQ->end2=(DQ->end2+1+QueueSize)%QueueSize; /*
移动队列指针*/
                return 1;
            }
            else
                return 0;
    }
    return 0;
}
int DeQueue(DQueue *DQ,DataType *e,int tag)
/*
将元素出队列，并将出队列的元素赋值给e
。如果出队列成功返回1
， 否则返回0*/
{
    switch(tag)
    {
        case 1:                                     /*1
表示左端队列元素出队*/
            if (DQ->end1!=DQ->end2)                    /*
判断队列是否为空*/
            {
                DQ->end1=(DQ->end1+1+QueueSize)%QueueSize; /*
元素出队列*/
                *e=DQ->queue[DQ->end1];                /*
将出队列的元素赋值给e*/
                return 1;
            }
            else
                return 0;
    }
}

```

```
case 2: /*2
表示右端队列元素出队*/
判断队列是否为空*/
元素出队列*/
将出队列的元素赋值给e*/
        if (DQ->end2!=DQ->end1) /*
        {
            DQ->end2=(DQ->end2-1+QueueSize)%QueueSize; /*
            *e=DQ->queue[DQ->end2]; /*
            return 1;
        }
        else
            return 0;
    }
    return 0;
}
```

5.5 综合案例：动画模拟停车场管理系统

设停车场是一个可停放 n 辆汽车的狭长通道，且只有一个大门可供汽车进出。汽车在停车场内按车辆到达时间的先后顺序，依次由北向南排列（大门在最南端，最先到达的第一辆车停放在车场的最北端）。若停车场内已经停满 n 辆车，那么后来的车只能在门外的便道上等候。一旦有车开走，则排在便道上的第一辆车即可开入。当停车场内某辆车要离开时，在它之后进入的车辆必须先退出车场为它让路，待该辆车开出大门外，其他车辆再按原次序进入车场。每辆停放在车场的车在它离开停车场时必须按它停留的时间长短缴纳费用。试为停车场编制按上述要求进行管理的模拟程序。

以栈模拟停车场，以队列模拟车场外的便道，按照从终端读入数据的序列进行模拟管理。每一组输入数据包括汽车的“到达”（‘A’表示）或“离去”（‘D’表示）信息、汽车标识（牌照号）以及到达或离去的时刻3个数据项。对每一组输入数据进行操作后的输出信息为：若是车辆到达，则输出汽车在停车场内或者便道上的停车位置；若是车辆离去，则输出汽车在停车场停留的时间和应缴纳的费用（便道上停留的时间不收费）。栈以顺序结构实现，队列以链表结构实现。

设 $n=2$ ，输入数据为（‘A’，1，5），（‘A’，2，10），（‘D’，1，15），（‘A’，3，20），（‘A’，4，25），（‘A’，5，30），（‘D’，2，35），（‘D’，4，40），（‘E’，0，0）。每一组输入数据包括汽车“到达”或“离去”信息、汽车牌照号码及到达或离去的时刻3个数据项，其中，‘A’表示到达；‘D’表示离去，‘E’表示输入结束，（‘A’，1，5）表示1号牌照车在5这个时刻到达，（‘D’，1，15）表示1号牌照车在15这个时刻离去。

根据栈的后进先出和队列的先进先出的性质，需要用栈模拟停车场，用队列模拟便道，当停车场停满车后，再进入的汽车需要停在便道上。算法思想很简单，当有汽车准备停车时，判断栈是否已满，如果栈未满，则将汽车信息入栈；如果栈满，则将汽车信息入队列；当有汽车离开时，先依次将栈中的元素出栈，依次暂存到另一个栈中，等该车辆离开后，再依次将暂存栈中的元素依次进入停车场栈，并将停在便道上的汽车入栈。

如下是一个带动画的模拟停车场程序。

```
#include <stdio.h>
#include <stdlib.h>           //malloc
#include <time.h>             //
获取系统时间所用函数
#include <conio.h>            //getch()
#include <windows.h>          //
设置光标信息 malloc
#define MaxSize 5             /*
定义停车场栈长度*/
#define PRICE 0.05           /*
每车每分钟收费值*/
#define BASEPRICE 0.5        //
```

```

基础停车费
#define Esc 27 //
退出系统
#define Exit 3 //
结束对话
#define Stop 1 //
停车
#define Drive 2 //
取车
int jx=0,jy=32; //
全局变量日志打印位置
typedef struct
{int hour;
int minute;
}Time,*PTime; //
时间结点*/
typedef struct //
定义栈元素的类型即车辆信息结点*/
{int num; //
车牌号*/
Time arrtime; //
到达时刻或离区时刻*/
}CarNode; //
定义栈,
模拟停车场*/
{CarNode stack[MaxSize];
int top;
}SqStackCar;
typedef struct node //
定义队列结点的类型*/
{int num; //
车牌号*/
struct node *next;
}QueueNode;
typedef struct //
定义队列,
模拟便道*/
{QueueNode *front,*rear;
}LinkQueueCar;
/*
函数声明*/
PTime get_time();
CarNode getcarInfo();
void qingping(int a);
void gotoxy(int x,int y);
void printlog(Time t,int n,int io,char ab,int po,double f);
void printstop(int a,int num,int x0,int y0);
void printleave(int a,int po,int num);
/*
初始化栈*/
void InitSeqStack(SqStackCar *s)
{
s->top=-1;
}
/* push
入站函数 */
int push(SqStackCar *s,CarNode x) //
数据元素x
入指针s
所指的栈
{
if(s->top==MaxSize-1)
return(0); //
如果栈满,
返回0
else
{
s->stack[++s->top]=x; //
栈不满,
到达车辆入栈
return(1);
}
}
/*
栈顶元素出栈*/
CarNode pop(SqStackCar *s)
{
CarNode x;
if(s->top<0)
{
x.num=0;
x.arrtime.hour=0;
x.arrtime.minute=0;
return(x); //
如果栈空,
返回空值
}
else
{
s->top--;
return(s->stack[s->top+1]); //
栈不空,
返回栈顶元素
}
}

```

```

/*
初始化队列*/
void InitLinkQueue(LinkQueueCar *q)
{
    q->front=(QueueNode*) malloc (sizeof(QueueNode));    //
    产生一个新结点,
    作头结点
    if (q->front!=NULL)
    {
        q->rear=q->front;
        q->front->next=NULL;
        q->front->num=0;    //
    }
    头结点的num
    保存队列中数据元素的个数
}
/*
数据入队列*/
void EnLinkQueue(LinkQueueCar *q,int x)
{
    QueueNode *p;
    p=(QueueNode*) malloc (sizeof(QueueNode));    //
    产生一个新结点
    p->num=x;
    p->next=NULL;
    q->rear->next=p;    //
    新结点入队列
    q->rear=p;
    q->front->num++;    //
    队列元素个数加1
}
/*
数据出队列*/
int DeLinkQueue(LinkQueueCar *q)
{
    QueueNode *p;
    int n;
    if (q->front==q->rear)    //
    队空返回0
        return (0);
    else
    {
        p=q->front->next;
        q->front->next=p->next;
        if (p->next==NULL)
            q->rear=q->front;
        n=p->num;
        free (p);
        q->front->num--;
        return (n);    //
    }
    返回出队的数据信息
}
/*****
车辆到达 *****/
//
参数: 停车栈
停车队列
车辆信息
//
返回值: 空
//
功能: 对传入的车辆进行入栈
栈满则入队列
void Arrive(SqStackCar *stop,LinkQueueCar *lq,CarNode x)
{
    int f;
    f=push(stop,x);    //
    入栈
    if (f==0)    //
    栈满
    {
        EnLinkQueue(lq,x.num);    //
        入队
        printstop(1,lq->front->num,0,23);
        printlog(x.arrtime,x.num,1,'B',lq->front->num,0);
        qingping(0);    printf("
您的车停在便道%d
号车位上\n",lq->front->num); //
更新对话
    }
    else
    {
        printstop(0,stop->top+1,0,23);
        printlog(x.arrtime,x.num,1,'P',stop->top+1,0);
        qingping(0);    printf("
您的车停在停车场%d
号车位上\n",stop->top+1); //
更新对话
    }
    qingping(1);    printf("
按任意键继续");
    getch();
}
/*****
车辆离开 *****/

```

```

//
参数: 停车栈指针s1
, 暂存栈指针s2
, 停车队列指针p
, 车辆信息x
//
返回值: 空
//
功能: 查找栈中s1
的x
并出栈, 栈中没有则查找队p
中并出队, 打印离开收费信息
void Leave(SqStackCar *s1, SqStackCar *s2, LinkQueueCar *p, CarNode x)
{
    double fee=0;
    int position=s1->top+1; //
    车辆所在车位
    int n, f=0;
    CarNode y;
    QueueNode *q;
    while((s1->top > -1) && (f!=1)) //
    当栈不空且未找到x
    {
        y=pop(s1);
        if(y.num!=x.num)
        {
            n=push(s2, y);
            position--;
        }
        else
            f=1;
    }
    if(y.num==x.num) //
    找到x
    {
        gotoxy(33,17); printf("%d:%-2d", (x.arrrtime.hour-y.arrrtime.hour),
        (x.arrrtime.minute-y.arrrtime.minute) );
        fee=((x.arrrtime.hour-y.arrrtime.hour)*60+(x.arrrtime.minute-y.arrrtime.minute))*PRICE+BASEPRICE;
        gotoxy(48,17); printf("%2.1f
    元\n", fee);
        qingping(0); printf("
    确认您的车辆信息");
        qingping(1); printf("
    按任意键继续");
        getch();
        while(s2->top>-1)
        {
            y=pop(s2);
            f=push(s1, y);
        }
        n=DeLinkQueue(p);
        if(n!=0)
        {
            y.num=n;
            y.arrrtime=x.arrrtime;
            f=push(s1, y);
            printleave(p->front->num+1, position, s1->top+1); //
            出栈动画
            printlog(x.arrrtime, x.num, 0, 'P', position, fee);
            printlog(y.arrrtime, y.num, 1, 'P', s1->top+1, 0);
        }
        else
        {
            printleave(0, position, s1->top+2);
            printlog(x.arrrtime, x.num, 0, 'P', position, fee);
        }
    }
    若栈中无x
    else //
    {
        while(s2->top > -1) //
        还原栈
        {
            y=pop(s2);
            f=push(s1, y);
        }
        q=p->front;
        f=0;
        position=1;
        while(f==0 && q->next!=NULL) //
        当队不空且未找到x
        {
            if(q->next->num!=x.num)
            {
                q=q->next;
                position++;
            }
            else //
            找到x
            {
                q->next=q->next->next;
                p->front->num--;
                if(q->next==NULL)
                p->rear=p->front;
                gotoxy(33,17); printf("0:0");
            }
        }
    }
}

```

```

        gotoxy(48,17); printf("0
元");
        qingping(0); printf("
您的车将离便道");
        qingping(1); printf("
按任意键继续");
        getch();
        printleave(-1,position,p->front->num+1); //
        printlog(x.arrrtime,x.num,0,'B',position,0);
        f=1;
    }
    if(f==0) //
        {
            qingping(0); printf("
停车场和便道上均无您的车");
            qingping(1); printf("
按任意键继续");
            getch();
        }
}
/*
获取系统时间*/
//
返回PTime
类型
PTime get_time()
{
    Time *t;
    time_t timer;
    struct tm *tblock;
    t=(Time*)malloc(sizeof(Time));
    timer=time(NULL);
    tblock=localtime(&timer);
    t->minute=tblock->tm_min;
    t->hour=tblock->tm_hour;
    return t;
}
/*
移动光标*/
//
将光标移动到 (x
,y
) 点
void gotoxy(int x,int y)
{
    COORD coord;
    coord.X=x;
    coord.Y=y+3;
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE),coord);
}
/*
画图*/
//
画出系统界面
void panitPL()
{
    int i,j,x,y,a[2][4]={2,0,0,1,-2,0,0,-1}; //
    方向];
    gotoxy(20,4);
    printf("*****
对话框*****");
    x=18,y=6; //
    起始点
    for(i=0;i<2 ;i++)
    {
        for(j=0; j<20; j++)
        {
            x+=a[i][0]; y+=a[i][1];
            gotoxy(x,y);
            printf("
=");
        }
        x+=a[i][0]; y+=a[i][1];
        gotoxy(x,y);
        if(i==0)
            printf("
");
        else
            printf("
");
        for(j=0; j<12; j++)
        {
            x+=a[i][2]; y+=a[i][3];
            gotoxy(x,y);
            printf("
");
        }
    }
}

```

```

        }
        x+=a[i][2];    y+=a[i][3];
        gotoxy(x,y);
        if(i==0)
            printf("
    』");
            else
                printf("
    Ⅱ");
        }
        gotoxy(22,8);
        printf("
    小张: ");
        gotoxy(22,11);
        printf("
    顾客: ");
        gotoxy(22,14);    printf("*****
    停车信息 *****");
        gotoxy(23,15);    printf("
    车牌号: ");
        gotoxy(42,15);    printf("
    时间: ");
        gotoxy(23,17);    printf("
    停车时长: ");
        gotoxy(42,17);    printf("
    收费: ");
    }
    /*
    清屏函数*/
    //
    更新对话框前将原对话框空
    void qingping(int a)
    {
        if(a==0)                //
        清空小张的对话
        {
            gotoxy(28,8);    printf("                ");
            gotoxy(28,9);    printf("                ");
            gotoxy(28,8);
        }
        else if(a==1)            //
        清空顾客的对话
        {
            gotoxy(28,11);    printf("                ");
            gotoxy(28,12);    printf("                ");
            gotoxy(28,13);    printf("                ");
            gotoxy(28,11);
        }
        else                    //
        清空车辆信息
        {
            gotoxy(31,15);    printf("                ");
            gotoxy(48,15);    printf("                ");
            gotoxy(33,17);    printf("                ");
            gotoxy(48,17);    printf("                ");
            gotoxy(31,15);
        }
    }
    //
    用上下键移动选择
    int getkey()
    {
        char c;
        int x=28,y=11;
        while(1)
        {
            gotoxy(x,11);    printf(" ");
            gotoxy(x,12);    printf(" ");
            gotoxy(x,13);    printf(" ");
            gotoxy(x,y);    printf(">>");

            c=getch();
            if(c==13) return y-10;                //
            按【Enter
            】键返回当前选项
            if(c!=-32) continue;                //
            不是方向键进行下次循环
            c=getch();
            if(c==72) if(y>11) y--;                //
            上
            if(c==80) if(y<13) y++;                //
            下
        }
    }
    //
    输入车辆信息
    CarNode getcarInfo()
    {
        PTime T;
        CarNode x;
    }

```



```

qingping(0);    printf("
请输入您的车牌号\n");
qingping(1);    printf("
在下面输入车辆信息");
qingping(2);
scanf("%d",&(x.num));
T=get_time();
x.arrtime=*T;
gotoxy(48,15);  printf("%d:%d",x.arrtime.hour,x.arrtime.minute);
getch();
return x;
}
//
打印停车场
void printcar()
{
    gotoxy(0,20);

    printf("
    出场暂放区
    1      2      3      4      5
    ");
    printf("-----");
    printf("
    主车道
    ");
    printf("-----");
    printf("12      11      10      9      8      7      6      5      4      3      2      1
    ");
    printf("
    便道
    停车区
    1      2      3      4      5
    ");
    printf("
    停车场管理日志\n\n");
    printf("
    时间
    车牌号
    进(1)/
    出(0)
    车位(B
    便道P
    停车场)
    收费(
    元)  ");
}
//
打印日志记录
void printlog(Time t,int n,int io,char ab,int po,double f)
{
    jy++;
    gotoxy(jx,jy);
    printf("
    时间
    车牌号
    进(1)/
    出(0)
    车位(B
    便道P
    停车场)
    收费(
    元)  ");
    if(io==0)
        printf("
        gotoxy(jx,jy);
        printf("
        gotoxy(jx,jy);
        printf(" %d: %d / %d",t.hour,t.minute,n);
}
void printstop(int a,int num,int x0,int y0)
{

```

```

static char *car="
[ ] ";
// int x0=0,y0=23;
int x=0,y=28;
if (a==0)
{
    x=(num+6)*6;
    for (;x0<72;x0++)
    {
        gotoxy(x0,y0); printf("%s",car); Sleep(30);
        gotoxy(x0,y0); printf(" ");
    }
    for (;y0<y;y0++)
    {
        gotoxy(x0,y0); printf("%s",car); Sleep(100);
        gotoxy(x0,y0); printf(" ");
    }
    for (;x0>x;x0--)
    {
        gotoxy(x0,y0); printf("%s",car); Sleep(50);
        gotoxy(x0,y0); printf(" ");
    }
    gotoxy(x,y);
    printf("%s",car);
}
else
{
    x=(12-num)*6;
    y=y-3;
    for (;x0<x;x0++)
    {
        gotoxy(x0,y0); printf("%s",car); Sleep(30);
        gotoxy(x0,y0); printf(" ");
    }
    gotoxy(x,y);
    printf("%s",car);
}
}
void printleave(int a,int po,int num)
{
    static char *car="
[ ] ";
int x0=0,y0=23;
int x=0,y=28;
int i;
if (a== -1)
{
    x=(12-po)*6;
    y=y-3;
    gotoxy(x,y); printf(" ");
    gotoxy(x,y-2); printf("%s",car);
    Sleep(100);
    if (12>num)
    {
        gotoxy((12-num)*6,y);
        printf(" ");
    }
    gotoxy(x,y); printf("%s",car);
    for (;x>x0;x--)
    {
        gotoxy(x,y-2); printf("%s",car); Sleep(30);
        gotoxy(x,y-2); printf(" ");
    }
}
else
{
    i=num+1;
    for (;num>po;num--)
    {
        x=(num+6)*6; y=28;
        for (;x<72;x++)
        {
            gotoxy(x,y); printf("%s",car); Sleep(30);
            gotoxy(x,y); printf(" ");
        }
        for (;y>21;y--)
        {
            gotoxy(x,y); printf("%s",car); Sleep(50);
            gotoxy(x,y); printf(" ");
        }
        for (;x>(i-num+6)*6;x--)
        {
            gotoxy(x,y); printf("%s",car); Sleep(30);
            gotoxy(x,y); printf(" ");
        }
        gotoxy(x,y); printf("%s",car);
    }
    x=(po+6)*6; y=28;
    for (;x<72;x++)

```

```

        {
            gotoxy(x,y);    printf("%s",car);    Sleep(30);
            gotoxy(x,y);    printf("    ");
        }
        for(;y>23;y--)
        {
            gotoxy(x,y);    printf("%s",car);    Sleep(50);
            gotoxy(x,y);    printf("    ");
        }
        for(;x>0;x--)
        {
            gotoxy(x,y);    printf("%s",car);    Sleep(30);
            gotoxy(x,y);    printf("    ");
        }
        num++;
        for(;i-num>0;num++)
        {
            x=(i-num+6)*6;    y=21;
            for(;x<72;x++)
            {
                gotoxy(x,y);    printf("%s",car);    Sleep(30);
                gotoxy(x,y);    printf("    ");
            }
            for(;y<28;y++)
            {
                gotoxy(x,y);    printf("%s",car);    Sleep(50);
                gotoxy(x,y);    printf("    ");
            }
            for(;x>(num-1+6)*6;x--)
            {
                gotoxy(x,y);    printf("%s",car);    Sleep(30);
                gotoxy(x,y);    printf("    ");
            }
            gotoxy(x,y);    printf("%s",car);
        }
        if(a>0)
        {
            x=66;
            y=25;
            gotoxy(x,y);    printf("    ");
            gotoxy(x,y-2);    printf("%s",car);    Sleep(100);
            if(12>a)
            {
                gotoxy((12-a)*6,y);
                printf("    ");
            }
            if(a>1)
            {
                gotoxy(x,y);    printf("%s",car);
            }
            printstop(0,i-1,x,y-2);
        }
    }
}

void main(void)
{
    int i,a;
    char c;
    SqStackCar s1,s2;
    LinkQueueCar p;

    InitSeqStack(&s1);
    InitSeqStack(&s2);
    InitLinkQueue(&p);
    printf("
停车场管理系统\n\n");
    printf("*****
欢迎光临    *****\n");
    printf("
收费标准: 基础费0.5
元, 每分钟收取0.05
元, 收费精确到0.1
元\n");
    printf("    PS:
车牌号由阿拉伯数字组成");
    panitPL();
    printcar();
    c=0;
    gotoxy(0,-3);

    接受按键
    while(1)

    按【Esc
    】键退出系统
    {
        for(i=2;i>-1;i--)

        初始化对话框
        qingping(i);
    }
}

```

```
printf("
按ESC
退出系统,
其他键开始对话");

c=getch();
if(c==Esc)
{
    qingping(0);
    break;
}
while(1)
{
    qingping(2);
    gotoxy(28,8);          printf("
欢迎来到停车场!我是管理员小张。");
    gotoxy(28,9);          printf("
请您按↓选择需要的服务");
    gotoxy(28,11); printf("  1.
我要停车");
    gotoxy(28,12); printf("  2.
我要取车");
    gotoxy(28,13); printf("  3.
结束对话");
    //
    a=getkey();
    if(a==Exit)
    {
        printf("
结束服务。");
        break;
    }
    switch(a)
    {
    case Stop:
        //
        Arrive(&s1,&p,getcarInfo());
        break;
    case Drive:
        //
        Leave(&s1,&s2,&p,getcarInfo());
        break;
    }
}
}
```

程序运行结果如图5-23所示。

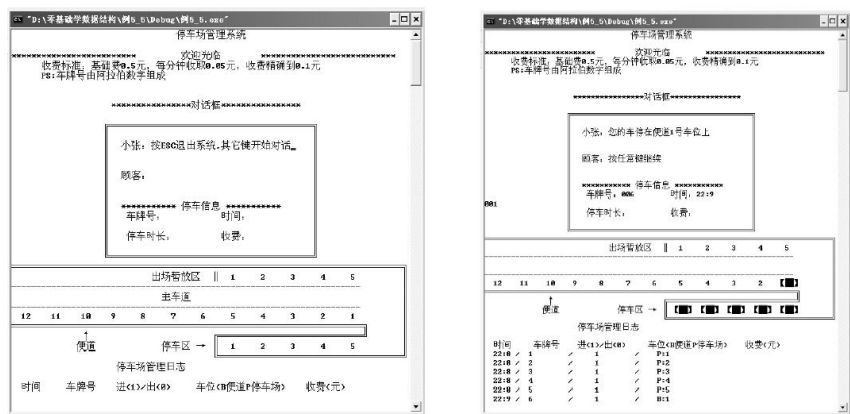


图5-23 停车场管理系统程序运行结果

本程序是一位本科生的毕业设计，不仅涵盖了栈和队列的内容，还要求熟练使用C语言中的一些关于字符和时间方面的函数，如gotoxy、localtime等。

5.6 小结

队列是只允许在表的一端进行插入操作，在另一端进行删除操作的线性表。

队列有顺序存储和链式存储两种存储方式。采用顺序存储结构的队列称为顺序队列，采用链式存储结构的队列称为链式队列。

顺序队列存在“假溢出”的问题，顺序队列的“假溢出”不是因为存储空间不足而产生的，为了避免“假溢出”，可用循环队列表示顺序队列。

为了区分循环队列的队空还是队满，有设置一个标志位和少用一个存储单元两种方案。

5.7 习题

一、选择题

1. 判断一个循环队列Q（最多n个元素）为满的条件是（）。

A. $Q \rightarrow rear == Q \rightarrow front$

B. $Q \rightarrow rear == Q \rightarrow front + 1$

C. $Q \rightarrow front == (Q \rightarrow rear + 1) \% n$

D. $Q \rightarrow front == (Q \rightarrow rear - 1) \% n$

2. 队列的插入操作是在（）。

A. 队尾

B. 队头

C. 队列任意位置

D. 队头元素后

3. 循环队列的队头和队尾指针分别为front和rear，则判断循环队列为空的条件是（）。

A. `front==rear`

B. `front==0`

C. `rear==0`

D. `front=rear+1`

4. 在一个链队列中，`front`和`rear`分别为头指针和尾指针，则插入一个结点`s`的操作为（ ）。

A. `front=front->next`

B. `s->next=rear; rear=s`

C. `rear->next=s; rear=s;`

D. `s->next=front; front=s;`

5. 一个队列的入队序列是1, 2, 3, 4, 则队列的出队序列是（ ）。

A. 1, 2, 3, 4

B. 4, 3, 2, 1

C. 1, 4, 3, 2

D. 3, 4, 1, 2

6. 依次在初始为空的队列中插入元素a、b、c、d以后，紧接着做了两次删除操作，此时的队头元素是（）。

A. a

B. b

C. c

D. d

7. 循环队列用数组A[0, m-1]存放其元素值，已知其头尾指针分别是front和rear，则当前队列中的元素个数是（）。

A. $(\text{rear} - \text{front} + m) \% m$

B. $\text{rear} - \text{front} + 1$

C. $\text{rear} - \text{front} - 1$

D. $\text{rear} - \text{front}$

8. 在一个链队列中，假定front和rear分别为队头指针和队尾指针，删除一个结点的操作是（）。

A. front=front->next

B. rear=rear->next

C. rear->next=front

D. front->next=rear

二、算法分析题

1. 写出如下算法的功能。

```
int  function(SqQueue *Q,ElemType *e){
    if(Q->front==Q->rear)
        return ERROR;
    *e=Q->base[Q->front];
    Q->front=(Q->front+1)%MAXSIZE;
    return OK;
}
```

2. 阅读算法f2，并回答下列问题。

(1) 设队列Q=（1，3，5，2，4，6），写出执行算法f2后的队列Q。

(2) 简述算法f2的功能。

```
void  f2(Queue *Q){
    DataType  e;
    if (!QueueEmpty(Q)){
        e=DeQueue(Q);
        f2(Q);
        EnQueue(Q,e);
    }
}
```

```
}  
}
```

三、算法设计题

1. 要求顺序循环队列不损失一个空间全部能够得到有效利用，请采用设置标志位tag的方法解决“假溢出”问题，实现顺序循环队列算法。

2. 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点，试编写相应的队列初始化、入队列和出队列的算法。

3. 要求顺序循环队列不损失一个空间全部能够得到有效利用，请采用设置标志位tag的方法解决“假溢出”问题，实现顺序循环队列算法。

提示 考查循环队列的入队和出队算法思想。设标志位为tag，初始化为tag=0，当入队列成功tag=1、出队列成功tag=0时，队列为空的判断条件为front==rear&&tag==0，队列满的判断条件为front==rear&&tag==1。

4. 利用链式循环队列实现如图5-24所示杨辉三角的打印输出。

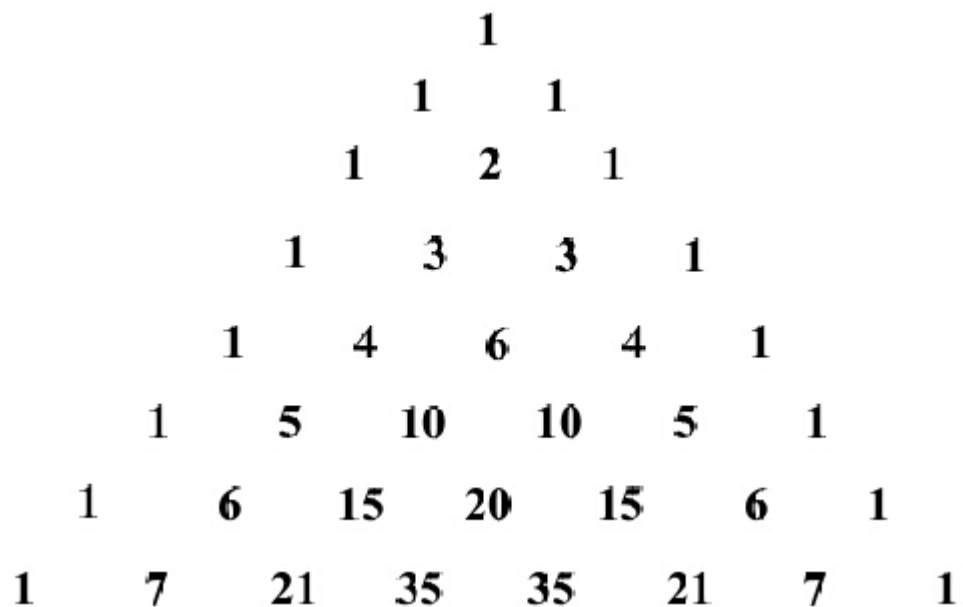


图5-24 8阶的杨辉三角

提示 本题主要考查队列的先入先出特性。设置一个队列，利用第*i*-1行元素产生第*i*行元素，第*i*行中间元素是它上一行*i*-1行对应位置元素与对应位置前一个元素之和。

第6章 串

计算机上的非数值处理对象基本上是字符串数据。在开发C语言程序的过程中，源程序和目标程序都是字符串数据。字符串一般简称为串，它也是一种重要的线性结构。在进销存等事物处理中，顾客的姓名和地址、货物的名称、产地和规格都是字符串数据，信息管理系统、信息检索系统、问答系统、自然语言翻译程序等都是以字符串数据作为处理对象的。本章主要介绍串的定义、串的顺序存储、堆分配存储、块链存储及串的模式匹配。

本章重点和难点：

- 串的顺序存储表示与实现
- 串的堆分配存储表示与实现
- 串的链式表示与实现
- 串的模式匹配算法

6.1 串的定义及抽象数据类型

串是仅由字符组成的一种特殊的线性表。本节主要介绍串的定义和串的抽象数据类型。

6.1.1 什么是串

串（string），或字符串，是由零个或多个字符组成的有限序列，一般记作 $S = "a_1 a_2 \cdots a_n"$ 。其中， S 是串名，用双引号括起来的字符序列是串的值， a_i （ $1 \leq i \leq n$ ）可以是字母、数字或其他字符， n 是串的长度。当 $n=0$ 时，串称为**空串**（null string）。

串中任意个连续的字符组成的子序列称为该串的**子串**。相应地，包含子串的串称为**主串**。通常将字符在串中的序号称为该字符在串中的位置。子串在主串中的位置以子串的第一个字符在主串中的位置来表示。

例如， a 、 b 、 c 、 d 是4个串， $a = \text{"northwest"}$ ， $b = \text{"university"}$ ， $c = \text{"northwestuniversity"}$ ， $d = \text{"northwest university"}$ ，可知它们的长度分别为9、10、19、20， a 和 b 都是 c 和 d 的子串， a 在 c 和 d 的位置都为1， b 在 c 的位置是10， b 在 d 的位置是11。

当且仅当两个串的值相等时，两个串是相等的。也就是说，只有当两个串的长度相等，且串中各个对应位置的字符均相等，两个串才是相等的。例如，上面的4个串a、b、c、d两两都不相等。

注意 串中的元素必须用一对双引号括起来，但是双引号并不属于串，双引号的作用仅仅是为了使其与变量名或常量相区别。严蔚敏版《数据结构》中串用单引号将字符括起来，这一点与本书略有不同，本书定义的串是用双引号括起来的，这主要是与C语言表示方法尽量统一。

在串`d="northwest university"`中，d是一个串的变量名，字符序列northwest university是串的值。

由一个或多个空格组成的串，称为**空格串**。空格串的长度是串中空格字符的个数。请注意，空格串不是空串。

6.1.2 串的抽象数据类型

1. 数据对象集合

串的数据对象集合为 $\{a_1, a_2, \dots, a_n\}$ ，每个元素的类型均为字符。

串是一种特殊的线性表，区别仅仅在于串的数据对象为字符集合。串具有线性表的特征：除了第一个元素 a_1 外，每一个元素有且只有一个直接前驱元素；除最后一个元素 a_n 外，每一个元素有且只有一个直接后继元素。数据元素之间的关系是一对一的关系。

2. 基本操作集合

串的操作通常不是以单个元素作为操作对象，往往是一连串的字符作为操作对象。例如，在串中查找某个子串，将一个串连接在另一个串的末尾、将串中的某个子串替换为另一个子串等。

这里为了方便说明，特定义以下几个串。

$S = \text{"Jiaozuo is a beautiful city"}$

$T = \text{"Jiaozuo is a tourist city of Henan Province"}$

$R = \text{"beautiful"}$

$V = \text{"tourist"}$

串的基本操作主要如下。

(1) $\text{StrAssign}(\&S, \text{cstr})$ ：串的赋值操作。把字符串常量 cstr 赋值给 S 。

(2) StrEmpty (S) : 判断串是否为空。如果是空串, 则返回1, 否则返回0。

(3) StrLength (S) : 求串的长度。返回串中的字符个数。

例如, StrLength (S) =27, StrLength (T) =43,
StrLength (R) =8, StrLength (V) =7。

(4) StrCopy (&T, S) : 串的复制。由字符串S复制产生一个与S完全相同的另一个字符串T。

(5) StrCompare (S, T) : 串的比较。比较串S和T的每个字符的ASCII值的大小, 如果S的值大于T, 则返回1; 如果S的值等于T, 则返回0; 如果S的值小于T, 则返回-1。

例如, StrCompare (S, T) =-1, 因串S和串T的第14个字符不相等, 字符'b'的ASCII值小于字符't'的ASCII值, 所以返回-1。

(6) StrInsert (&S, pos, T) : 串的插入操作。在串S的第pos个位置插入串T, 若插入成功, 返回1; 否则返回0。

例如, 若在串S中的第24个位置插入字符串"and charming"后, 即 StrInsert (S, 24, "and charming"), 串S="Jiaozuo is a beautiful and charming city"。

(7) StrDelete (&S, pos, len) : 串的删除操作。如果在串S中删除第pos个字符开始, 长度为len的字符串。如果找到并删除成功, 返回1; 否则返回0。

例如, 若删除串T中第26个位置起的18个字符, 即StrDelete (T, 26, 18) , 则T="Jiaozuo is a tourist city"。

(8) StrConcat (&T, S) : 串的连接。将串S连接在串T的末尾。连接成功, 返回1; 否则, 返回0。

例如, StrCat (T, S) 就是将串S连接在串T的末尾, 连接后有T="Jiaozuo is a tourist city of Henan Province Jiaozuo is a beautiful city"。

(9) SubString (&Sub, S, pos, len) : 截取子串。截取串S中从第pos个字符起的长度为len的连续字符, 并赋给Sub。截取成功返回1, 否则返回0。

例如, SubString (Sub, S, 14, 9) 就是把串S中的第14个字符起长度为9的字符串赋给Sub, 则Sub="beautiful"。

(10) StrReplace (&S, T, V) : 串的替换。若串S中存在子串T, 则用V替换串S中的所有子串T。替换操作成功, 返回1; 否则返回0。

例如，StrReplace (S, R, V) 就是把串S中的子串R替换为串V，替换后有S="Jiaozuo is a tourist city"。

(11) StrIndex (S, pos, T) : 返回子串的定位。若主串S中存在与串T的值相等的子串，则返回子串T在主串S中第pos个字符起的第一次出现的位置，否则返回0。

例如，StrIndex (S, 6, R) =14就是在串S中的第6个字符开始查找子串R第一次出现的位置，beautiful在第14个位置出现。

(12) StrClear (&S) : 清空串。将串的值清空。

(13) StrDestroy (&S) : 销毁串。将串的存储空间销毁。

6.2 串的顺序表示与实现

串也有顺序存储和链式存储两种存储方式。最为常用的是串的顺序存储表示，操作起来更为方便。本节主要介绍串的顺序存储结构及顺序存储结构下的操作实现。

6.2.1 串的顺序存储结构

采用顺序存储结构的串称为顺序串，又称定长顺序串。一般采用字符型数组存放顺序串。当定义了一个字符数组时，数组的起始地址已经确定。

在串的顺序存储结构中，确定串的长度有两种方法，一种就是在串的末尾加上一个结束标记，在C语言中，在定义串时，系统会自动在串值的最后添加‘\0’作为结束标记。例如，一个字符数组的定义如下。

```
char str[]="Northwest University";
```

则串“Northwest University”在内存中的存放形式如图6-1所示。

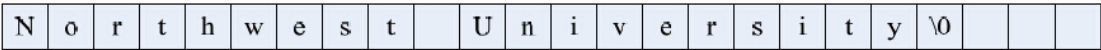


图6-1 “Northwest University”在内存中的存放形式

串“Northwest University”的长度为20，不包括结束标记“\0”。此时的串长为隐含值，显然不便于某些操作。

另一种方法是用一个变量length存放串的长度，通常这种方法更为常用。例如，串“Northwest University”在内存中用设置串的长度的方法如图6-2所示。

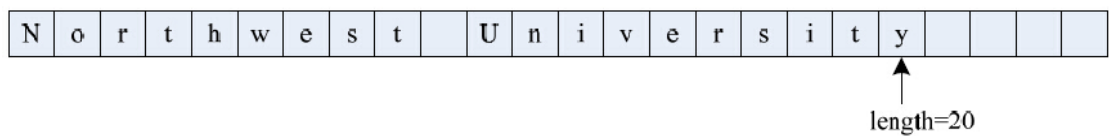


图6-2 利用串长度表示“Northwest University”的长度

串的顺序存储结构类型描述如下。

```
#define MaxLen 80
typedef struct
{
    char str[MaxLen];
    int length;
}SeqString;
```

其中，MaxLen表示串的最大长度，str是存储串的字符数组，length为串的长度。

6.2.2 顺序串的基本运算

在顺序存储结构中，串的基本运算如下（以下算法的实现保存在文件“SeqString.h”中）。

①串的赋值。串的赋值即把字符串常量cstr中的每一个字符赋值给串S。串的赋值算法实现如下。

```
void StrAssign(SeqString *S,char cstr[])
/*
串的赋值操作*/
{
    int i=0;
    for(i=0;cstr[i]!=
```

```
'\0
';i++)          /*
将常量cstr
中的字符赋值给串S*/
    S->str[i]=cstr[i];
    S->length=i;
}
```

②判断串是否为空，算法实现如下。

```
int StrEmpty(SeqString S)
/*
判断串是否为空，串为空返回1
， 否则返回0*/
{
    if(S.length==0)                /*
如果串的长度等于0*/              /*
        return 1;                    /*
    返回1*/
    else                            /*
    否则*/                          /*
        return 0;                    /*
    返回0*/
}
```

③求串的长度，算法实现如下。

```
int StrLength(SeqString S)
/*
求串的长度*/
{
    return S.length;
}
```

④串的复制，算法实现如下。

```
void StrCopy(SeqString *T,SeqString S)
/*
串的复制操作.*/
{
    int i;
    for(i=0;i<S.length;i++)        /*
将串S
的字符赋值给串T*/
        T->str[i]=S.str[i];
    T->length=S.length;            /*
将串S
的长度赋值给串T*/
}
```

⑤比较两个串的大小。串的比较操作就是比较串S和T的每个字符的ASCII值的大小，如果S的值大于T，则返回正值；如果S的值等于T，则返回0；如果S的值小于T，则返回负值。

比较两个串的大小算法实现如下。

```
int StrCompare(SeqString S,SeqString T)
/*
串的比较操作*/
{
    int i;
    for(i=0;i<S.length&& i<T.length;i++)          /*
从第一个字符开始比较两个串中的字符*/
        if(S.str[i]!=T.str[i])                    /*
如果有不相等的字符*/
            return (S.str[i]-T.str[i]);           /*
返回两个字符的差值*/
    return (S.length-T.length);                    /*
如果比较完毕，返回两个串的长度的差值*/
}
```

⑥在串S的第pos位置插入串T。若插入成功，返回1；否则返回0。

串的插入操作具体实现分为如下3种情况。

第1种情况是在S中插入T后串长不超过能容纳的最长字符，即 $S \rightarrow \text{length} + T.\text{length} \leq \text{MaxLen}$ ，则先将串S中pos后的字符向后移动len个位置，然后将串T插入S中即可。

第2种情况是若将T插入S后，串长超过能容纳的最长字符，但T能完全插入S中，即 $S \rightarrow \text{length} + T.\text{length} > \text{MaxLen}$ ，则将串S中pos后的字符往后移len个位置后，S中的部分字符被舍弃。

第3种情况是将T插入S中，有 $S \rightarrow \text{length} + T.\text{length} > \text{MaxLen}$ ，且T不能完全被插入到S中，则T中部分字符和S中第len个位置之后的字符均被舍弃。

串的插入操作的算法实现如下。

```
int StrInsert(SeqString *S,int pos,SeqString T)
/*
串的插入。在S
中第pos
个位置插入T
分为三种情况*/
{
    int i;
    if (pos<0||pos-1>S->length)          /*
插入位置不正确，返回0*/
    {
        printf("
插入位置不正确");
        return 0;
    }
    if (S->length+T.length<=MaxLen) /*
第1
种情况，插入子串后串长≤MaxLen
，子串T
完整地插入到串S
中*/
    {
        /*
在插入子串T
前，将S
中pos
后的字符向后移动len
个位置*/
        for(i=S->length+T.length-1;i>=pos+T.length-1;i--)
            S->str[i]=S->str[i-T.length];
        /*
将串T
插入到S
中*/
        for(i=0;i<T.length;i++)
            S->str[pos+i-1]=T.str[i];
        S->length=S->length+T.length;
        return 1;
    }
    /*
第2
种情况，子串可以完全插入到S
中，但是S
中的字符将会被截掉*/
    else if (pos+T.length<=MaxLen)
    {
        for(i=MaxLen-1;i>T.length+pos-1;i--)          /*
将S
中pos
以后的字符整体移动到数组的最后*/
            S->str[i]=S->str[i-T.length];
        for(i=0;i<T.length;i++)                          /*
将T
插入到S
中*/
            S->str[i+pos-1]=T.str[i];
        S->length=MaxLen;
        return 0;
    }
    /*
第3
种情况，子串T
不能被完全插入到S
中，T
中将会有字符被舍弃*/
    else
    {
        for(i=0;i<MaxLen-pos;i++)                      /*
...
*/
    }
```

```

将T
直接插入到S
中，插入之前不需要移动S
中的字符*/
        S->str[i+pos-1]=T.str[i];
        S->length=MaxLen;
        return 0;
    }
}

```

⑦删除串S中pos开始的len个字符。删除成功返回1，否则返回0。串的删除算法实现如下。

```

int StrDelete(SeqString *S,int pos,int len)
/*
在串S
中删除pos
开始的len
个字符*/
{
    int i;
    if(pos<0||len<0||pos+len-1>S->length) /*
如果参数不合法，则返回0*/
    {
        printf("
删除位置不正确，参数len
不合法");
        return 0;
    }
    else
    {
        for(i=pos+len;i<=S->length-1;i++) /*
将串S
的第pos
个位置以后的len
个字符覆盖掉*/
            S->str[i-len]=S->str[i];
        S->length=S->length-len; /*
修改串S
的长度*/
        return 1;
    }
}

```

⑧将串S连接在串T的末尾。串的连接操作可分为两种情况，一种是连接后串长 $T \rightarrow \text{length} + S.\text{length} \leq \text{MaxLen}$ ，则直接将串S连接在串T的尾部；另一种是连接后串长 $T \rightarrow \text{length} + S.\text{length} \geq \text{MaxLen}$ 且串T的长度 $< \text{MaxLen}$ ，则串S会有字符丢失。串的连接算法实现如下。

```

int StrConcat(SeqString *T,SeqString S)
/*
将串S
连接在串T
的末尾*/
{

```

```

    int i, flag;
    /*
第1
种情况，连接后的串长小于等于MaxLen
，将S
直接连接在串T
末尾*/
    if (T->length+S.length<=MaxLen)
    {
        for(i=T->length; i<T->length+S.length; i++) /*
串S
直接连接在T
的末尾*/
            T->str[i]=S.str[i-T->length];
        T->length=T->length+S.length; /*
修改串T
的长度*/
        flag=1; /*
修改标志，表示S
完整连接到T
中*/
    }
    /*
第2
种情况，连接后串长大于MaxLen
，S
部分被连接在串T
末尾*/
    else if (T->length<MaxLen)
    {
        for(i=T->length; i<MaxLen; i++) /*
将串S
部分连接在T
的末尾*/
            T->str[i]=S.str[i-T->length];
        T->length=MaxLen; /*
修改串T
的长度*/
        flag=0; /*
修改标志，表示S
部分被连接在T
中*/
    }
    return flag;
}

```

⑨清空串操作，算法实现如下。

```

void StrClear(SeqString *S)
/*
清空串，只需要将串的长度置为0
即可*/
{
    S->length=0;
}

```

6.2.3 顺序串应用举例

【例6-1】 要求编写一个删除字符串“abcdeabdbcdadaabdecdf”中所有子串“abd”的程序。

【分析】 主要考查串的创建、定位、删除等基本操作的用法。删除所有子串的程序实现如下。

```
#include<stdio.h>
#include<string.h>
#define MaxLen 60
typedef struct
{
    char str[MaxLen];
    int length;
}SeqString;
int DelSubString(SeqString *S,int pos,int n);
void StrPrint(SeqString S);          /*
串的输出函数声明*/
int Index(SeqString *S1,SeqString *S2)
{
    int i=0,j,k;
    while(i<S1->length)
    {
        j=0;
        if(S1->str[i]==S2->str[j])
        {
            k=i+1;
            j++;
            while(k<S1->length && j<S2->length && S1->str[k]==S2->str[j])
            {
                k++;
                j++;
            }
            if(j==S2->length)
                break;
            else
                i++;
        }
        else
            i++;
    }
    if(i>=S1->length)
        return -1;
    else
        return i+1;
}
int DelSubString(SeqString *S,int pos,int n)
{
    int i;
    if(pos+n>S->length)
        return 0;
    for(i=pos+n-1;i<S->length;i++)
        S->str[i-n]=S->str[i];
    S->length=S->length-n;
    S->str[S->length]='\0';
    return 1;
}
int StrLength(SeqString *S)
{
    return S->length;
}
void DelAllString(SeqString *S1,SeqString *S2)
{
    int n;
```

```

        n=Index(S1,S2);
        while(n>=0)
        {
            DelSubString(S1,n,StrLength(S2));
            n=Index(S1,S2);
        }
    }
void CreateString(SeqString *s,char str[])
{
    strcpy(s->str,str);
    s->length=strlen(str);
}
void main()
{
    SeqString S1,S2;
    char str[MaxLen];
    printf("
字符串:");
    gets(str);
    CreateString(&S1,str);
    printf("
子串:");
    gets(str);
    CreateString(&S2,str);
    DelAllString(&S1,&S2);
    printf("
删除所有子串后的字符串:");
    StrPrint(S1);
}
void StrPrint(SeqString S)
{
    int i;
    for(i=0;i<S.length;i++)
    {
        printf("%c",S.str[i]);
    }
    printf("\n");
}

```

程序的运行结果如图6-3所示。

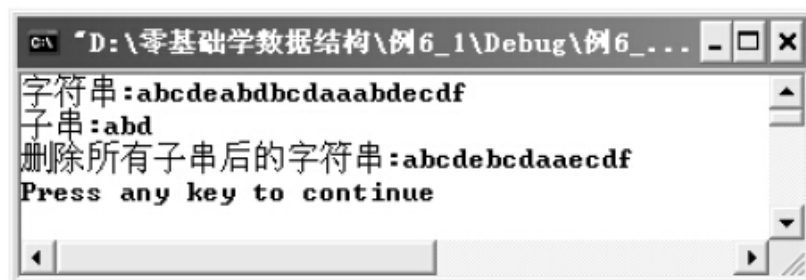


图6-3 串的基本操作程序运行结果

6.3 串的堆分配表示与实现

在采用静态顺序存储表示的顺序串中，在串的插入操作、串的连接及串的替换操作中，如果串的长度超过了MaxLen，串会被截掉一部分。为了克服顺序串静态分配的缺点，可以使用动态存储分配表示串并实现串的基本操作——串的堆分配表示与实现。

6.3.1 堆分配的存储结构

采用堆分配存储表示的串称为**堆串**。堆串仍然采用一组地址连续的存储单元，存放串中的字符。但是，堆串的存储空间是在程序的执行过程中动态分配的。

在C语言中，函数malloc和free管理堆的存储空间。利用函数malloc为串动态分配一块存储空间，若分配成功，返回存储空间起始地址的指针，作为串的基地址（起始地址）。如果内存单元使用完，调用函数free释放内存空间。

堆串的类型定义如下：

```
typedef struct
{
    char *str;
    int length;
}HeapString;
```

其中，str是指向堆串的起始地址的指针，length表示堆串的长度。

6.3.2 堆串的基本运算

堆串的基本运算与静态分配的顺序串类似，其算法实现如下（堆串的算法的实现保存在文件“HeapString.h”中）。

①初始化堆串，算法实现如下。

```
InitString(HeapString *S)
/*
串的初始化操作*/
{
    S->length=0; /*
将串的长度置为0*/
    S->str='\0'; /*
将串置的值为空*/
}
```

②将字符串cstr中的字符赋给串S，算法实现如下。

```
void StrAssign(HeapString *S,char cstr[])
/*
串的赋值操作*/
{
    int i=0,len;
    if(S->str)
        free(S->str);
    for(i=0;cstr[i]!='\0';i++); /*
求cstr
字符串的长度*/
    len=i;
    if(!i) /*
如果字符串cstr
的长度为0
，则将串s
的长度置为0
，内容置为空*/
    {
        S->str='\0';
        S->length=0;
    }
    else
    {
        S->str=(char*)malloc(len*sizeof(char)); /*
为串动态分配存储空间*/
        if(!S->str)
            exit(-1);
        for(i=0;i<len;i++) /*
将字符串cstr
的内容赋值给串S*/
            S->str[i]=cstr[i];
        S->length=len; /*
... ..
```

```
将串的长度置为0*/  
    }  
}
```

③复制串，算法实现如下。

```
void StrCopy(HeapString *T,HeapString S)  
/*  
串的复制操作.*/  
{  
    int i;  
    T->str=(char*)malloc(S.length*sizeof(char));          /*  
为串动态分配存储空间*/  
    if(!T->str)  
        exit(-1);  
    for(i=0;i<S.length;i++)                                /*  
将串S  
的字符赋值给串T*/  
        T->str[i]=S.str[i];  
    T->length=S.length;                                     /*  
将串S  
的长度赋值给串T*/  
}
```

④在S中第pos个位置插入T，算法实现如下。

```
int StrInsert(HeapString *S,int pos,HeapString T)  
/*  
在S  
中第pos  
个位置插入T */  
{  
    int i;  
    if(pos<0||pos-1>S->length)                            /*  
插入位置不正确，返回0*/  
    {  
        printf("  
插入位置不正确");  
        return 0;  
    }  
    S->str=(char*)realloc(S->str,(S->length+T.length)*sizeof(char));  
    if(!S->str)  
    {  
        printf("  
内存分配失败");  
        exit(-1);  
    }  
    for(i=S->length-1;i>=pos-1;i--)                          /*  
将串S  
中第pos  
个位置的字符往后移动T.length  
个位置*/  
        S->str[i+T.length]=S->str[i];  
    for(i=0;i<T.length;i++)                                /*  
将串T  
的字符赋值到S  
中*/  
        S->str[pos+i-1]=T.str[i];
```

```

        S->length=S->length+T.length;          /*
修改串的长度*/
        return 1;
}

```

⑤删除串S中pos开始的len个字符，算法实现如下。

```

int StrDelete(HeapString *S,int pos,int len)
/*
在串S
中删除pos
开始的len
个字符*/
{
    int i;
    char *p;
    if(pos<0||len<0||pos+len-1>S->length)          /*
如果参数不合法，则返回0*/
    {
        printf("
删除位置不正确，参数len
不合法");
        return 0;
    }
    p=(char*)malloc(S->length-len);                  /*p
指向动态分配的内存单元*/
    if(!p)
        exit(-1);
    for(i=0;i<pos-1;i++)                             /*
将串第pos
位置之前的字符复制到p
中*/
        p[i]=S->str[i];
    for(i=pos-1;i<S->length-len;i++)                  /*
将串第pos+len
位置以后的字符复制到p
中*/
        p[i]=S->str[i+len];
    S->length=S->length-len;                             /*
修改串的长度*/
    free(S->str);                                         /*
释放原来的串S
的内存空间*/
    S->str=p;                                             /*
将串的str
指向p
字符串*/
    return 1;
}

```

⑥将串S连接在串T的末尾，算法实现如下。

```

int StrConcat(HeapString *T,HeapString S)
/*
将串S
连接在串T
的后面*/
{

```

```

        int i;
        T->str=(char*)realloc(T->str, (T->length+S.length)*sizeof(char));    /*
重新分配内存空间, 使串的长度为S
和T
的长度和, T
中原来的内容不变*/
        if(!T->str)
        {
            printf("
分配空间失败");
            exit(-1);
        }
        else
        {
            for(i=T->length;i<T->length+S.length;i++)    /*
串S
直接连接在T
的末尾*/
                T->str[i]=S.str[i-T->length];
            T->length=T->length+S.length;    /*
修改串T
的长度*/
        }
        return 1;
    }
}

```

⑦销毁串，算法实现如下。

```

void StrDestroy (HeapString *S)
/*
销毁串，只需要将串的长度置为0
即可*/
{
    if(S->str)
        free(S->str);    /*
释放串S
的内存空间*/
    S->str='\0';    /*
将串的内容置为空*/
    S->length=0;    /*
将串的长度置为0*/
}

```

6.4 串的块链式存储表示与实现

由于串也是一种线性表，因此串也可以采用链式存储表示。本节主要介绍串的块链式存储及其基本运算。

6.4.1 串的块链式存储结构

和线性表的链式存储结构类似，串也可以采用链表存储串值。由于串结构的特性——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。例如一个结点包含4个字符，即结点大小为4的链串如图6-4所示；一个结点包含1个字符，即结点大小为1的链串如图6-5所示。

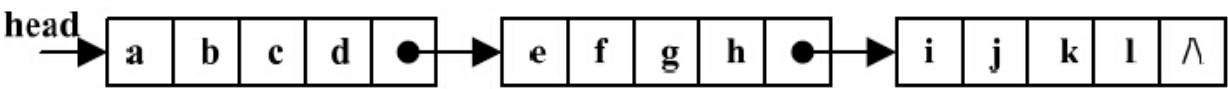


图6-4 一个结点包含4个字符的链串结构

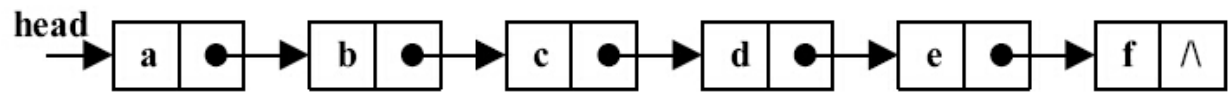


图6-5 结点大小为1的链串结构

每个结点存放多个字符，可以有效地利用存储空间。当结点大小大于1时，由于串长不一定刚好为结点大小的整数倍，则链表中的最后

一个结点不一定全部被串值占满，此时通常补上特殊的字符“#”或其他非串值字符。例如一个包含10个字符的链串（在最后补上两个“#”），如图6-6所示。

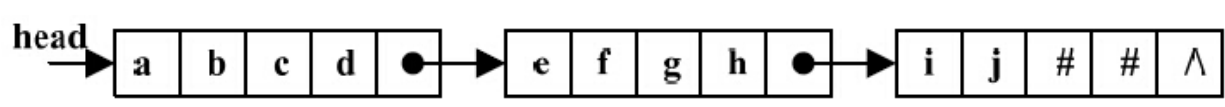


图6-6 填充两个“#”的链串结构

为了方便串的操作，当以链表存储串值时，除表头指针外，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。这样的串存储结构称为[块链结构](#)。

串的块链存储结构类型描述如下。

```
#define ChunkSize 10
#define stuff '#'
/*
串的结点类型定义*/
typedef struct Chunk
{
    char ch[ChunkSize];
    struct Chunk *next;
}Chunk;
/*
块链串的类型定义*/
typedef struct
{
    Chunk *head;
    Chunk *tail;
    int length;
}LinkString;
```

当ChunkSize等于1时，链串就变成一个普通链表。当ChunkSize大于1时，链串中的每个结点可以存放多个字符。head表示头指针，指向

链串的第一个结点。tail表示尾指针，指向链串的最后一个结点。
length表示链串中字符的个数。

6.4.2 块链串的基本运算

块链串基本操作的算法实现如下（算法实现保存在文件“LinkString.h”中）。

①初始化块链串，实现代码如下。

```
void InitString(LinkString *S)
/*
初始化块链串*/
{
    S->length=0;                /*
将串的长度置为0*/
    S->head=S->tail=NULL;      /*
将串的头指针和尾指针置为空*/
}
```

②串的赋值。先求出块链串的结点个数len，然后动态生成结点，将字符串cstr中的字符赋值给链串的数据域。如果是最后一个结点且结点的数据域没有填满，则用‘#’填充。实现代码如下。

```
int StrAssign(LinkString *S,char *cstr)
/*
生成一个其值等于cstr
的串s
。成功返回1
，否则返回0*/
{
    int i,j,k,len;
    Chunk *p,*q;
    len=strlen(cstr);          /*len
为链串的长度 */
    if(!len)
        return 0;
    S->length=len;
    j=len/ChunkSize;          /*j
... ..
```

```

        while(p) /*
块链没结束 */
        {
            for(i=0;i<ChunkSize;i++)
                if(p->ch[i]!=stuff) /*
如果当前字符不是填充的特殊字符#
, 则将s
中字符赋值给q*/
                    *q++=(p->ch[i]);

            p=p->next;
        }
        (*cstr)[S.length]=0; /*
在字符串的末尾添加结束标志*/
        return 1;
    }

```

④比较两个串的大小，即比较串S和T的每个字符的ASCII值的大小。如果S的值大于T，则返回正值；如果S的值等于T，则返回0；如果S的值小于T，则返回负值。具体实现方法是先分别将串S和T转换为字符串p和q，然后依次比较p和q中的每个字符。如果两个字符相等，继续比较下一个；否则将两个字符的差值赋值给flag。如果其中一个串已经比较完毕或者两个串都已经比较完毕，则返回两个串的长度的差值；否则返回flag。

算法实现如下。

```

int StrCompare(LinkString S,LinkString T)
/*
串的比较操作。若s
的值大于t
, 则返回正值; 若s
的值等于t
, 则返回0
: 若s
的值小于t
, 则返回负值*/
{
    char *p,*q;
    int flag;
    if(!ToChars(S,&p)) /*
将串s
转换为字符串p*/
        return 0;
    if(!ToChars(T,&q)) /*
将串t
转换为字符串q*/

```

```

        return 0;
for(;*p!='\0'&&*q!='\0';)
    if(*p==*q)
    {
        p++;
        q++;
    }
    else
        flag=*p-*q;
free(p); /*
释放p
的空间 */
free(q); /*
释放q
的空间*/
if(*p=='\0' || *q=='\0')
    return S.length-T.length;
else
    return flag;
}

```

⑤将串T插入串S的第pos个位置。先将串S和T分别转换为字符串s1和t1，然后动态申请内存空间，将字符串t1插入到s1中，最后将字符串s1转换为串S。算法实现如下。

```

int StrInsert(LinkString *S, int pos, LinkString T)
/*
串的插入操作。在串S
的第pos
个位置插入串T*/
{
    char *t1,*s1;
    int i,j;
    int flag;
    if(pos<1||pos>S->length+1) /*
如果插入位置不合法*/
        return 0;
    if(!ToChars(*S,&s1)) /*
将串S
转换为字符串s1*/
        return 0;
    if(!ToChars(T,&t1)) /*
将串T
转换为字符串t1*/
        return 0;
    j=strlen(s1); /*j
为字符串s1
的长度*/
    s1=(char*)realloc(s1,(j+strlen(t1)+1)*sizeof(char)); /*
为s1
重新分配空间*/
    for(i=j;i>=pos-1;i--)
        s1[i+strlen(t1)]=s1[i]; /*
将字符串s1
中的第pos
以后的字符向后移动strlen(t1)
个位置*/
}

```

```

        for(i=0;i<(int)strlen(t1);i++)    /*
在字符串s1
中插入t1*/
            s1[pos+i-1]=t1[i];
        InitString(S);                    /*
释放S
的原有存储空间*/
        flag=StrAssign(S,s1);            /*
由s1
生成串S*/
        free(t1);
        free(s1);
        return flag;
    }

```

⑥将串S中的第pos个字符起的长度为len的子串删除。删除成功返回1，否则返回0。将串S转换为字符串str，然后将字符串中第pos个字符起的len个字符删除，最后将字符串转换为串S。链串的删除算法实现如下。

```

int StrDelete(LinkString *S,int pos,int len)
/*
串的删除操作。将串S
中的第pos
个字符起长度为len
的子串删除*/
{
    char *str;
    int i;
    int flag;
    if(pos<1||pos>S->length-len+1||len<0)    /*
参数不合法*/
        return 0;
    if(!ToChars(*S,&str))                    /*
将串S
转换为字符串str*/
        return 0;
    for(i=pos+len-1;i<=(int)strlen(str);i++) /*
将字符串中第pos
个字符起的长度为len
的子串删除*/
        str[i-len]=str[i];
    InitString(S);                            /*
释放S
的原有存储空间*/
    flag=StrAssign(S,str);                    /*
将字符串str
转换为串S*/
    free(str);
    return flag;
}

```

⑦清空串，代码如下。

```
void ClearString(LinkString *S)
/*
清空串操作。将串的空间释放*/
{
    Chunk *p,*q;
    p=S->head;
    while(p)
    {
        q=p->next;
        free(p);
        p=q;
    }
    S->head=S->tail=NULL;
    S->length=0;
}
```

6.5 串的模式匹配

串的模式匹配在串的各种操作中是经常用到的一个算法，也是本书的一个难点之一。串的模式匹配也称为子串的定位操作，即查找子串在主串中出现的位置。本节主要介绍串的朴素模式匹配算法——Brute-Force及改进算法——KMP算法。

6.5.1 朴素模式匹配算法——Brute-Force

子串的定位操作串通常称为**模式匹配**，是各种串处理系统中最重要操作之一。设有主串S和子串T，如果在主串S中找到一个与子串T相等的串，则返回串T的第一个字符在串S中的位置。其中，主串S又称为**目标串**，子串T又称为**模式串**。

Brute-Force算法的思想是从主串 $S = "s_0 s_1 \cdots s_{n-1}"$ 的第pos个字符开始与模式串 $T = "t_0 t_1 \cdots t_{m-1}"$ 的第一个字符比较，如果相等则继续逐个比较后续字符；否则从主串的下一个字符开始重新与模式串T的第一个字符比较，依次类推。如果在主串S中存在与模式串T相等的连续字符序列，则匹配成功，函数返回模式串T中第一个字符在主串S中的位置；否则函数返回-1表示匹配失败。

假设主串为 $S = "ababcbacacbab"$ ，模式串为 $T = "abcac"$ ，S的长度为 $n=13$ ，T的长度为 $m=5$ 。用变量i表示主串S中当前正在比较字符的下标，变量j表示子串T中当前正在比较字符的下标。模式匹配的过程如图6-7所示。

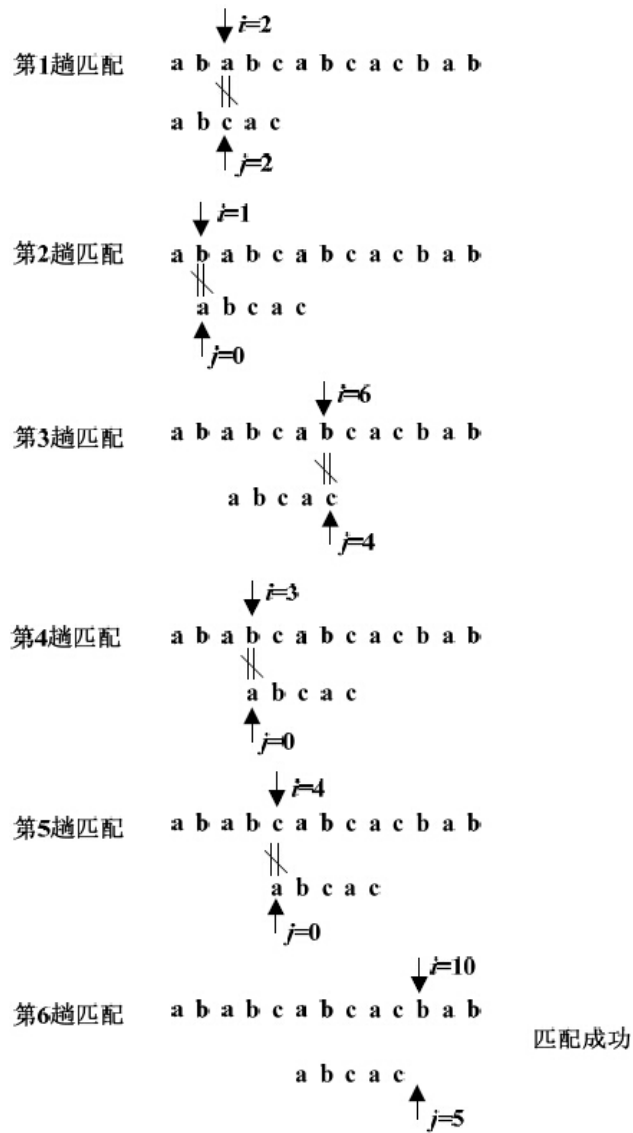


图6-7 经典的模式匹配过程

假设串采用顺序存储方式存储，则Brute-Force匹配算法如下。

```

int B_FIndex(SeqString S,int pos,SeqString T)
/*
在主串S
中的第pos
个位置开始查找模式串T
，如果找到返回子串在主串的位置；否则，返回-1*/
{
    int i,j;
    i=pos-1;
    j=0;
    while(i<S.length&& j<T.length)
    {
        if (S.str[i]==T.str[j]) /*
如果串S
和串T
中对应位置字符相等，则继续比较下一个字符*/
        {
            i++;
            j++;
        }
    }
}

```

```

        else /*
        如果当前对应位置的字符不相等，则从串s
        的下一个字符开始，T
        的第0
        个字符开始比较*/
        {
            i=i-j+1;
            j=0;
        }
    }
    if(j>=T.length) /*
    如果在S
    中找到串T
    ，则返回子串T
    在主串S
    的位置*/
        return i-j+1;
    else
        return -1;
}

```

Brute-Force匹配算法简单且容易理解，并且进行某些文本处理时，效率也比较高，如检查“Information”是否存在于下列主串“Northwest University International Admissions Welcome to study at Northwest University, Xi’ an, China.”中时，上述算法中while循环次数（即进行单个字符比较的次数）为98，恰好为（S.length-2），这就是说，除了主串中呈黑体的2个字符（每个字符比较了两次）外，其他字符均只和模式串比较了一次。在这种情况下，此算法的时间复杂度为 $O(n+m)$ 。其中，n和m分别为主串和模式串的长度。

然而在有些情况下，该算法的效率却很低。例如设主串S=“aaaaaaaaaaaab”，模式串T=“aab”。其中，n=14，m=4。因为模式串的前3个字符是“aaa”，主串的前13个字符也是“aaa”，每趟比较模式串的最后一个字符与主串中的字符不相等，所以均需要将主串的指针回退，从主串的下一个字符开始与模式串的第一个字符重新比较。在整个匹配过程中，主串的指针需要回退9次，匹配不成功的比较次数是 $10*4$ ，成功匹配的比较次数是4次，因此总的比较次数是 $10*4+4=11*4$ 即 $(n-m+1)*m$ 。

可见，Brute-Force匹配算法在最好的情况下，即主串的前m个字符刚好与模式串相等，时间复杂度为 $O(m)$ 。在最坏的情况下，Brute-Force匹配算法的时间复杂度是 $O(n*m)$ 。

在Brute-Force算法中，即使主串与模式串已有多个字符经过比较相等，只要有一个字符不相等，就需要将主串的比较位置回退。

6.5.2 KMP算法

KMP算法是由D. E. Knuth、J. H. Morris、V. R. Pratt共同提出的，因此称为KMP算法（Knuth-Morris-Pratt算法）。KMP算法在Brute-Force算法的基础上有较大改进，可在 $O(n+m)$ 时间数量

级上完成串的模式匹配，主要是消除了主串指针的回退，使算法效率有了很大程度的提高。

1. KMP算法思想

KMP算法的基本思想是在每一趟匹配过程中出现字符不等时，不需要回退主串的指针，而是利用已经得到前面“部分匹配”的结果，将模式串向右滑动若干个字符后，继续与主串中的当前字符进行比较。

那到底向右滑动多少个字符呢？假设主串 $S = \text{"ababcbabcacbab"}$ ，模式串 $T = \text{"abcac"}$ ，KMP算法匹配过程如图6-8所示。

从图6-8中可以看出，KMP算法的匹配次数由Brute-Force算法的6趟减少为3趟。回顾图6-7的匹配过程，在第3趟匹配过程中，当 $i=6$ 、 $j=4$ 时，主串中的字符与模式串中的字符不相等，又从 $i=3$ 、 $j=0$ 开始比较。经过仔细观察，其实， $i=3$ 、 $j=0$ ， $i=4$ 、 $j=0$ ， $i=5$ 、 $j=0$ 这3次比较都是没有必要的。因为从第3趟的部分匹配可得出： $S_2 = T_0 = 'a'$ ， $S_3 = T_1 = 'b'$ ， $S_4 = T_2 = 'c'$ ， $S_5 = T_3 = 'a'$ ， $S_6 \neq T_4$ ，因为 $S_3 = T_1$ 且 $T_0 \neq T_1$ ，所以 $S_3 \neq T_0$ ，所以没有必要从 $i=3$ 、 $j=0$ 开始比较。又因为 $S_4 = T_2$ 且 $T_0 \neq T_2$ ，故 $S_4 \neq T_0$ ，所以 S_4 与 T_0 也没有必要从 $i=4$ 、 $j=0$ 开始比较。又因为 $S_5 = T_3$ 且 $T_0 \neq T_3$ ，故 $S_5 \neq T_0$ ，所以也没有必要将 S_5 与 T_0 进行比较。

也就是说，根据第3趟的部分匹配可以得出结论，Brute-Force算法的第4趟、第5趟是没有必要的（模式串的第1个字符与主串的第4、5个字符不相等），第6趟也没有必要将主串的第6个字符与模式串的第1个字符比较（两个字符相等）。因此，只需要将模式串向右滑动3个字符，从 $i=6$ 、 $j=1$ 开始比较。

同理，在第1趟匹配过程中，当出现字符不相等时，只需将模式串向右滑动2个字符从 $i=2$ 、 $j=0$ 开始比较即可。在整个KMP算法中，主串中的 i 指针没有回退。

下面来讨论一般情况。假设主串 $S = \text{"s}_0 \text{ s}_1 \cdots \text{s}_{n-1} \text{"}$ ，模式串 $T = \text{"t}_0 \text{ t}_1 \cdots \text{t}_{m-1} \text{"}$ 。在模式匹配过程中，如果出现字符不匹配的情况，即当 $S_i \neq T_j$ ($0 \leq i < n$, $0 \leq j < m$) 时，有

$$\text{"s}_{i-j} \text{ s}_{i-j+1} \cdots \text{s}_{i-1} \text{"} = \text{"t}_0 \text{ t}_1 \cdots \text{t}_{j-1} \text{"}$$

假设子串即模式串存在可重叠的真子串，即

$$t_0 t_1 \cdots t_{k-1} = t_{j-k} t_{j-k+1} \cdots t_{j-1}$$

也就是说，子串中存在从 t_0 开始到 t_{k-1} 与从 t_{j-k} 到 t_{j-1} 的重叠子串。根据上面两个等式，可以得出 $s_{i-k} s_{i-k+1} \cdots s_{i-1} = t_0 t_1 \cdots t_{k-1}$ ，如图6-9所示。

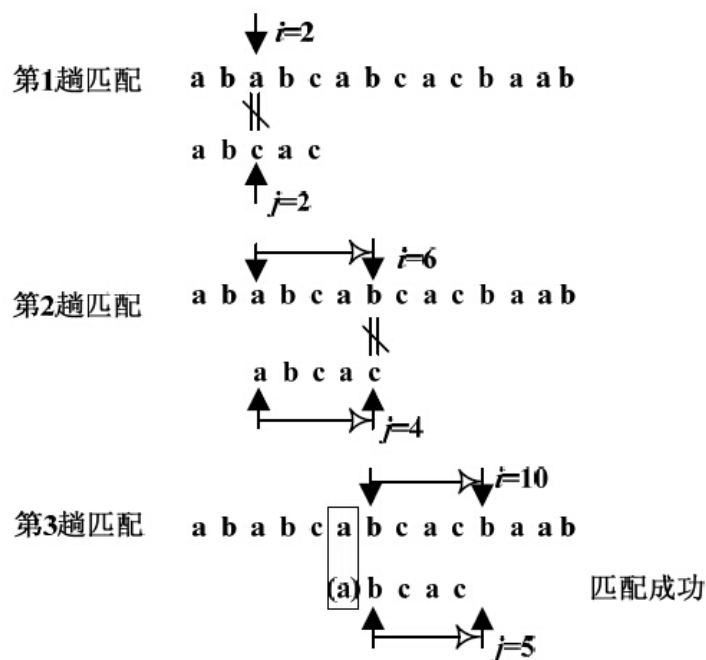


图6-8 KMP算法的匹配过程

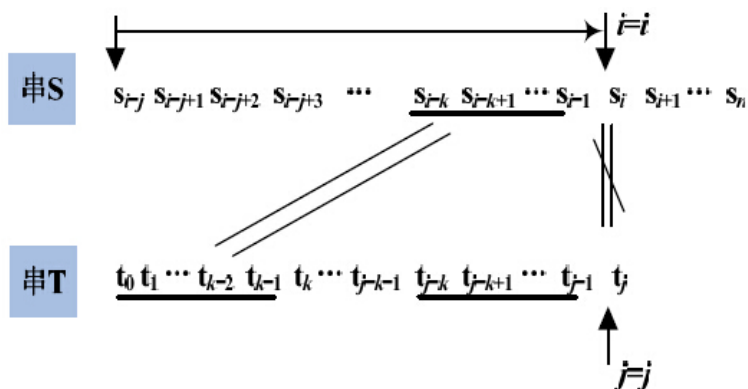


图6-9 在子串有重叠时主串与子串模式匹配过程

匹配的过程中，当主串中的第 $i+1$ 个字符与模式串中的第 $j+1$ 个字符不等时，仅需将模式串向右滑动至第 $k+1$ 个字符与主串中的第 $i+1$ 个字符对齐，即 s_i 与 t_k 对齐，此时，模式串中子串 $t_0 t_1 \cdots t_{k-1}$ 必定与主串中的子串 $s_{i-k} s_{i-k+1} \cdots s_{i-1}$ 相等，因此，匹配只需从主串中的第 $i+1$ 个字符与模式串中的第 $k+1$ 个字符开始比较。

如果令next[j]=k，则next[j]表示当模式串中的第j个字符与主串中的对应的字符不相等时，在模式串中需要重新与主串中与该字符进行比较的字符的位置。模式串中的next函数定义如下。

$$\text{next}[j] \begin{cases} -1, & \text{当} j=0 \text{ 时} \\ \text{Max}\{k|0 < k < j \text{ 且 "t}_0\text{t}_1\cdots\text{t}_{k-1}" = \text{"t}_{j-k}\text{t}_{j-k+1}\cdots\text{t}_{j-1}"\}, & \text{当该集合不空时} \\ 0, & \text{其他情况} \end{cases}$$

其中，第1种情况，next[j]的函数是为了方便算法设计而定义的；第2种情况，如果子串（模式串）中存在重叠的真子串，则next[j]的取值就是k，即模式串的最长子串的长度；第3种情况，如果模式串中不存在重叠的子串，则从子串的第一个字符开始比较。

由此可以推出模式串“abacabaaad”的next函数值如表6-1所示。

表6-1 模式串“abacabaaad”的next函数值

j	0	1	2	3	4	5	6	7	8	9
模式串	a	b	a	c	a	b	a	a	a	d
next[j]	-1	0	0	1	0	1	2	3	1	1

KMP算法的模式匹配过程：如果模式串T中存在真子串“t₀t₁…t_{k-1}”=“t_{j-k}t_{j-k+1}…t_{j-1}”，当模式串T的t_j与主串S的s_i不相等，则按照next[j]=k将模式串向右滑动，从主串中的s_i与模式串的t_k开始比较；如果s_i=t_k，则主串与模式串的指针各自增1，继续比较下一个字符；如果s_i≠t_k，则按照next[next[j]]将模式串继续向右滑动，将主串中的s_i与模式串中的next[next[j]]字符进行比较；如果仍然不相等，则按照以上方法，将模式串继续向右滑动，直到next[j]=-1为止。这时，模式串不再向右滑动，从s_{i+1}开始与t₀进行比较。

利用next函数值的一个模式匹配举例如图6-10所示。

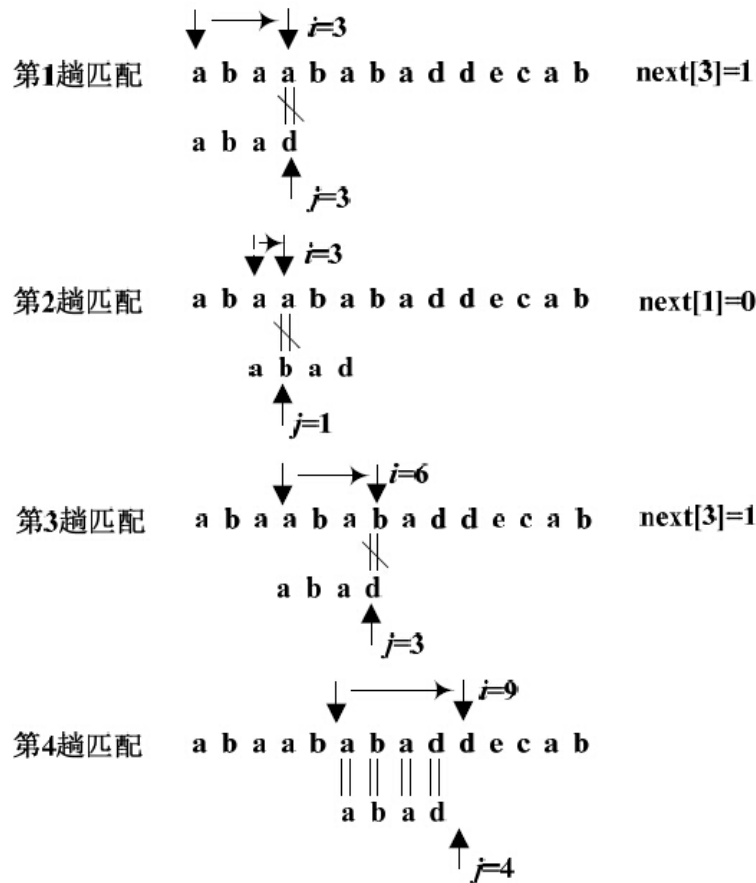


图6-10 利用next函数的模式匹配过程

利用模式串T的next函数值求T在主串S中的第pos个字符之后的位置的KMP算法描述如下。

```
int KMP_Index(SeqString S,int pos,SeqString T,int next[])
/*KMP
模式匹配算法。利用模式串T
的next
函数在主串S
中的第pos
个位置开始查找模式串T
，如果找到返回模式串在
主串的位置；否则，返回-1*/
{
    int i,j;
    i=pos-1;
    j=0;
    while(i<S.length&&j<T.length)
    {
        if(j==-1||S.str[i]==T.str[j]) /*
如果j=-1
或当前字符相等，则继续比较后面的字符*/
        {
            i++;
            j++;
        }
        else /*
如果当前字符不相等，则将模式串向右移动*/
            j=next[j]; /*
数组next
保存next
函数值*/
        }
        if(j>=T.length) /*
匹配成功，返回子串在主串中的位置*/
            return i-T.length+1;
        else /*
            ...
            */
    }
```

```
    否则返回-1*/  
    return -1;  
}
```

2. 求next函数值

KMP模式匹配算法是建立在模式串的next函数值已知的基础上的。下面来讨论如何求模式串的next函数值。

从上面的分析可以看出，模式串的next函数值的取值与主串无关，仅与模式串相关。根据模式串next函数定义，next函数值可用递推的方法得到。

设 $\text{next}[j]=k$ ，表示在模式串T中存在以下关系：

$$t_0 t_1 \cdots t_{k-1} = t_{j-k} t_{j-k+1} \cdots t_{j-1}$$

其中， $0 < k < j$ ， k 为满足等式的最大值，即不可能存在 $k' > k$ 满足以上等式。那么计算 $\text{next}[j+1]$ 的值可能有如下两种情况出现。

(1) 如果 $t_j = t_k$ ，则表示在模式串T中满足关系 $t_0 t_1 \cdots t_k = t_{j-k} t_{j-k+1} \cdots t_j$ ，并且不可能存在 $k' > k$ 满足以上等式。因此有 $\text{next}[j+1]=k+1$ ，即 $\text{next}[j+1]=\text{next}[j]+1$ 。

(2) 如果 $t_j \neq t_k$ ，则表示在模式串T中满足关系 $t_0 t_1 \cdots t_k \neq t_{j-k} t_{j-k+1} \cdots t_j$ 。在这种情况下，可以把求next函数值的问题看成一个模式匹配的问题。目前已经有 $t_0 t_1 \cdots t_{k-1} = t_{j-k} t_{j-k+1} \cdots t_{j-1}$ ，但是 $t_j \neq t_k$ ，把模式串T向右滑动到 $k' = \text{next}[k]$ ，如果有 $t_j = t_{k'}$ ，则表示模式串中有 $t_0 t_1 \cdots t_{k'} = t_{j-k'} t_{j-k'+1} \cdots t_j$ ，因此有 $\text{next}[j+1]=k'+1$ ，即 $\text{next}[j+1]=\text{next}[k]+1$

如果 $t_j \neq t_{k'}$ ，则将模式串继续向右滑动到第 $\text{next}[k']$ 个字符与 t_j 比较。如果仍不相等，则将模式串继续向右滑动到下标为 $\text{next}[\text{next}[k']]$ 字符与 t_j 比较。依次类推，直到 t_j 和模式串中某个字符匹配成功或不存在任何 k' ($1 < k' < j$) 满足 $t_0 t_1 \cdots t_{k'} = t_{j-k'} t_{j-k'+1} \cdots t_j$ ，则有 $\text{next}[j+1]=0$ 。

以上讨论的是如何根据next函数的定义递推得到next函数值。例如，模式串 $T = \text{"abcdabcdabe"}$ 的next函数值如表6-2所示。

表6-2 模式串“abcdabcdabe”的next函数值

j	0	1	2	3	4	5	6	7	8	9
模式串	c	b	c	a	a	c	b	c	b	c
next[j]	-1	0	0	1	0	0	1	2	3	2

在表6-2中，如果已经求得前3个字符的next函数值，现在求next[3]，因为next[2]=0且t₂=t₀，则next[3]=next[2]+1=1。接着求next[4]，因为t₂=t₀，但“t₂t₃”≠“t₀t₁”，则需要将t₃与下标为next[1]=0的字符即t₀比较，因为t₀≠t₃，则next[4]=0。

同理，在求得next[8]=3后，如何求next[9]？因为“t₅t₆t₇”≠“t₀t₁t₂”，但t₈≠t₃，则比较t₁与t₈的值是否相等（next[3]=1），有t₁=t₈，则next[9]=k'+1=1+1=2。

求next函数值的算法描述如下。

```
void GetNext(SeqString T,int next[])
/*
求模式串T
的next
函数值并存入数组next*/
{
    int j,k;
    j=0;
    k=-1;
    next[0]=-1;
    while(j<T.length)
    {
        if(k==-1||T.str[j]==T.str[k])/*
若k=-1
或当前字符相等，则继续比较后面字符将函数值存入next
数组*/
        {
            j++;
            k++;
            next[j]=k;
        }
        else
            /*
            如果当前字符不相等，则将模式串向右移动继续比较*/
            k=next[k];
    }
}
```

求next函数值的算法时间复杂度是O（m）。一般情况下，模式串的长度比主串的长度要小得多，因此，对整个字符串的匹配来说，增加这点时间是值得的。

3. 改进的求next函数算法

上述求next函数值有时也存在缺陷。例如，主串S=“aaaacabacaaaba”与模式串T=“aaaab”进行匹配时，当i=4、j=4时，s₄≠t₄，而因为next[0]=-1，next[1]=0，next[2]=1，next[3]=2，next[4]=3所以需要将主串的s₄与子串中的t₃、t₂、t₁、t₀依次进行比较。因模式串中

的 t_3 与 t_0 、 t_1 、 t_2 都相等，没有必要将这些字符与主串的 s_3 进行比较，仅需要直接将 s_4 与 t_0 进行比较。

一般地，在求得 $next[j]=k$ 后，如果模式串中的 $t_j=t_k$ ，则当主串中的 $s_i \neq t_j$ 时，不必再将 s_i 与 t_k 比较，而直接与 $t_{next[k]}$ 比较。因此，可以将求 $next$ 函数值的算法进行修正，即在求得 $next[j]=k$ 之后，判断 t_j 是否与 t_k 相等，如果相等，还需继续将模式串向右滑动，使 $k'=next[k]$ ，判断 t_j 是否与 $t_{k'}$ 相等，直到两者不等为止。

例如，模式串 $T="abcdabcdabc"$ 的函数值与改进后的函数值如表6-3所示。

表6-3 模式串“abcdabcdabc”的next函数值

j	0	1	2	3	4	5	6	7	8	9	10
模式串	a	b	c	d	a	b	c	d	a	b	d
$next[j]$	-1	0	0	0	0	1	2	3	4	5	6
$nextval[j]$	-1	0	0	0	-1	0	0	0	-1	0	6

其中， $nextval[j]$ 中存放改进后的 $next$ 函数值。在表6-3中，如果主串中对应的字符 s_i 与模式串 T 对应的 t_8 失配，则应取 $t_{next[8]}$ 与主串的 s_i 比较，即 t_4 与 s_i 比较，因为 $t_4=t_8='a'$ ，所以也一定与 s_i 失配，则取 $t_{next[4]}$ 与 s_i 比较，即 t_0 与 s_i 比较，又 $t_0='a'$ ，也必然与 s_i 失配，则取 $next[0]=-1$ ，这时，模式串停止向右滑动。其中， t_4 、 t_0 与 s_i 比较是没有意义的，所以需要修正 $next[8]$ 和 $next[4]$ 的值为-1。同理，用类似的方法修正其他 $next$ 的函数值。

求 $next$ 函数值的改进算法描述如下。

```
void GetNextVal(SeqString T,int nextval[])
/*
求模式串T
的next
函数值的修正值并存入数组nextval*/
{
    int j,k;
    j=0;
    k=-1;
    nextval[0]=-1;
    while(j<T.length)
    {
        if(k== -1 || T.str[j]==T.str[k]) /*
如果k=-1
或当前字符相等，则继续比较后面的字符并将函
数值存入到nextval
数组*/
        {
            j++;
            k++;
            if(T.str[j]!=T.str[k]) /*
如果所求的nextval[j]
与已有的nextval[k]
不相等，则
将k
存放在nextval
中*/
            {
                k=-1;
            }
        }
        nextval[j]=k;
        j++;
    }
}
```



```
printf("
模式串T
的next
和改进后的next
值:\n");

PrintArray(T,next,nextval,StrLength(T)); /*

输出模式串T
的next
值和nextval
值*/

find=B_FIndex(S,l,T,&count1); /*

朴素模式串匹配*/

if(find>0)
    printf("Brute-Force
算法的比较次数为:%2d\n",count1);
find=KMP_Index(S,l,T,next,&count2);
if(find>0)
    printf("
利用next
的KMP
算法的比较次数为:%2d\n",count2);
find=KMP_Index(S,l,T,nextval,&count3);
if(find>0)
    printf("
利用nextval
的KMP
匹配算法的比较次数为:%2d\n",count3);
/*
第2
个例子*/

StrAssign(&S,"abccbaaaababcabcbccabcbcabccbcbbb"); /*

给主串S
赋值*/

StrAssign(&T,"abcabcbc"); /*

给模式串T
赋值*/

GetNext(T,next); /*

求next
函数值*/

GetNextVal(T,nextval); /*

求改进后的next
函数值*/

printf("
模式串T
的next
和改进后的next
值:\n");

PrintArray(T,next,nextval,StrLength(T)); /*

输出模式串T
的next
值和nextval
值*/

find=B_FIndex(S,l,T,&count1); /*

朴素模式串匹配*/

if(find>0)
    printf("Brute-Force
算法的比较次数为:%2d\n",count1);
find=KMP_Index(S,l,T,next,&count2);
if(find>0)
    printf("
利用next
的KMP
算法的比较次数为:%2d\n",count2);
find=KMP_Index(S,l,T,nextval,&count3);
if(find>0)
    printf("
利用nextval
的KMP
匹配算法的比较次数为:%2d\n",count3);
}

void PrintArray(SeqString T,int next[],int nextval[],int length)
/*
模式串T
的next
值与nextval
值输出函数*/
{
    int j;
    printf("j:\t\t");
    for(j=0;j<length;j++)
        printf("%3d",j);
    printf("\n");
    printf("
模式串:\t\t");
    for(j=0;j<length;j++)
        printf("%3c",T.str[j]);
    printf("\n");
    printf("next[j]:\t");
    for(j=0;j<length;j++)
        printf("%3d",next[j]);
    printf("\n");
    printf("nextval[j]:\t");
```

```

        for(j=0;j<length;j++)
            printf("%3d",nextval[j]);
        printf("\n");
    }

```

2. 模式串匹配实现

这部分主要包括朴素的Brute-Force算法与KMP算法实现代码，如下。

```

int B_FIndex(SeqString S,int pos,SeqString T,int *count)
/*
在主串S
中的第pos
个位置开始查找子串T
, 如果找到返回子串在主串的位置; 否则, 返回-1*/
{
    int i,j;
    i=pos-1;
    j=0;
    *count=0;
    保存主串与模式串的比较次数*/
    while(i<S.length&&j<T.length)
    {
        if(S.str[i]==T.str[j])
            /*
            若串S
            和串T
            中对应位置字符相等, 则继续比较下一个字符*/
            {
                i++;
                j++;
            }
            else
                /*
                若当前对应位置的字符不相等, 则从串S
                的下一个字符开始, T
                的第0
                个字符开始比较*/
                {
                    i=i-j+1;
                    j=0;
                }
                (*count)++;
            }
            if(j>=T.length)
                /*
                如果在S
                中找到串T
                , 则返回子串T
                在主串S
                的位置*/
                return i-j+1;
            else
                return -1;
        }
    }
    int KMP_Index(SeqString S,int pos,SeqString T,int next[],int *count)
    /*KMP
    模式匹配算法。利用模式串T
    的next
    函数在主串S
    中的第pos
    个位置开始查找子串T
    , 如果找到返回子串在主串
    的位置; 否则, 返回-1*/
    {
        int i,j;
        i=pos-1;
        j=0;
        *count=0;
        保存主串与模式串的比较次数*/
        while(i<S.length&&j<T.length)
        {
            if(j==-1||S.str[i]==T.str[j])
                /*
                如果j=-1
                或当前字符相等, 则继续比较后面的字符*/
                {
                    i++;
                    j++;
                }
                else
                    /*
                    如果当前字符不相等, 则将模式串向右移动*/
                    j=next[j];
                    (*count)++;
            }
            if(j>=T.length)
                /*
                .....

```

```
匹配成功，返回子串在主串中的位置。否则返回-1*/
    return i-T.length+1;
else
    return -1;
}
```

3. 求next函数值部分

这部分包括KMP算法中的求next函数值及改进的求next函数值代码实现，如下。

```
void GetNext(SeqString T,int next[])
/*
求模式串T
的next
函数值并存入数组next*/
{
    int j,k;
    j=0;
    k=-1;
    next[0]=-1;
    while(j<T.length)
    {
        if(k==-1||T.str[j]==T.str[k])    /*
如果k=-1
或当前字符相等，则继续比较后面的字符并
将函数值存入到next
数组*/
        {
            j++;
            k++;
            next[j]=k;
        }
        else    /*
如果当前字符不相等，则将模式串向右移动继续比较*/
            k=next[k];
    }
}

void GetNextVal(SeqString T,int nextval[])
/*
求模式串T
的next
函数值的修正值并存入数组next*/
{
    int j,k;
    j=0;
    k=-1;
    nextval[0]=-1;
    while(j<T.length)
    {
        if(k==-1||T.str[j]==T.str[k])    /*
若k=-1
或当前字符相等，则继续比较后面的字符并将函数
值存入nextval
数组*/
        {
            j++;
            k++;
            if(T.str[j]!=T.str[k])    /*
如果所求的nextval[j]
与已有的nextval[k]
不相等，则
将k
存放在nextval
中*/
                nextval[j]=k;
            else
                nextval[j]=nextval[k];
        }
        else    /*
如果当前字符不相等，则将模式串向右移动继续比较*/
            k=nextval[k];
    }
}
```

程序运行结果如图6-11所示。

```
GA "D:\零基础学数据结构\例6_4\Debug\例6_4.exe"
模式串T的next和改进后的next值:
j:      0 1 2 3 4 5 6 7 8 9 10 11
模式串: a b a a b a c a b a b a
next[j]: -1 0 0 1 1 2 3 0 1 2 3 2
nextval[j]: -1 0 -1 1 0 -1 3 -1 0 -1 3 -1
Brute-Force算法的比较次数为:40
利用next的KMP算法的比较次数为:31
利用nextval的KMP匹配算法的比较次数为:30
模式串I的next和改进后的next值:
j:      0 1 2 3 4 5 6 7
模式串: a b c a b c b c
next[j]: -1 0 0 0 1 2 3 0
nextval[j]: -1 0 0 -1 0 0 3 0
Brute-Force算法的比较次数为:26
利用next的KMP算法的比较次数为:25
利用nextval的KMP匹配算法的比较次数为:24
Press any key to continue
```

图6-11 串的模式匹配程序运行结果

6.6 小结

串是由零个或多个字符组成的有限序列。其中，含零个字符的串称为空串。串中的字符可以是字母、数字或其他字符。串中任意个连续的字符组成的子序列称为串的子串，相应地，包含子串的串称为主串。

两个串相等当且仅当两个串中对应位置的字符相等并且长度相等。注意空串与空格串的区别。

串也有顺序存储结构和链式存储结构两种存储结构。

串的链式存储结构也称为块链的存储结构，它是采用一个“块”作为结点的数据域，存储串中的若干个字符。但是这种结构在串的各种操作中会带来不便，因为在串的操作过程中，需要判断一个结点是否结束，需要一个块一个块取数据和存储数据。串的长度可能不是块大小的整数倍，因此在最后的一个结点的数据域空出的部分用‘#’填充。

由于串的顺序存储结构在串的各种操作中实现方便，并且存储空间的利用率很高，所以串的顺序存储结构更常用。

串的模式匹配有两种方法：朴素模式匹配（即Brute-Force算法）与串的改进算法（即KMP算法）。对于Brute-Force算法，在每次出现主串与模式串的字符不相等时，主串的指针均需回退。而KMP算法根据模式串中的next函数值，消除了主串中的字符与模式串中的字符不匹配时主串指针的回退，提高了算法的效率。

6.7 习题

一、选择题

1. 设有两个串S1和S2，求串S2在S1中首次出现位置的运算称作（）。

A. 连接

B. 求子串

C. 模式匹配

D. 判断子串

2. 已知串S="aaab"，则next数组值为（）。

A. 0123

B. 1123

C. 1231

D. 1211

3. 串与普通的线性表相比较，它的特殊性体现在（）。

A. 顺序的存储结构

B. 链式存储结构

C. 数据元素是一个字符

D. 数据元素任意

4. 设串长为 n ，模式串长为 m ，则KMP算法所需的附加空间为（ ）。

A. $O(m)$

B. $O(n)$

C. $O(m*n)$

D. $O(n \log_2 m)$

5. 空串和空格串（ ）。

A. 相同

B. 不相同

C. 可能相同

D. 无法确定

6. 设SUBSTR (S, i, k) 是求S中从第i个字符开始的连续k个字符组成的子串的操作, 则对于S="Beijing&Nanjing", SUBSTR (S, 4, 5) = () 。

A. "ijing"

B. "jing&"

C. "ingNa"

D. "ing&N"

二、算法分析题

1. 函数实现串的模式匹配算法, 请在空格处将算法补充完整。

```
int index_bf(sqstring*s,sqstring *t,int start){
    int i=start-1,j=0;
    while(i<s->len&& j<t->len)
        if(s->data[i]==t->data[j]){
            i++;j++;
        }else{
            i=          ;j=0;
        }
    if(j>=t->len)
        return          ;
    else
        return -1;
}}
```

2. 写出下面算法的功能。

```
int function(SqString *s1,SqString *s2){
    int i;
    for(i=0;i<s1->length&&i<s1->length;i++)
```

```
        if(s->data[i]!=s2->data[i])
            return s1->data[i]-s2->data[i];
        return s1->length-s2->length;
    }
```

3. 写出算法的功能。

```
int fun(sqstring *s,sqstring *t,int start){
    int i=start-1,j=0;
    while(i<s->len&& j<t->len)
        if(s->data[i]==t->data[j]){
            i++;j++;
        }else{
            i=i-j+1;j=0;
        }
    if(j>=t->len)
        return i-t->len+1;
    else
        return -1;
}
```

三、算法设计题

1. 利用串的基本运算，编写算法实现将主串S中子串T删除。假设主串S="abaaccaacbbcbcacbbc"，子串T="caacb"，则将子串删除后主串S="abaacbbcbcacbbc"。

2. 实现字符串的比较函数与字符串的副本函数。字符串的比较函数原型为int strcmp (char*s1, char*s2; 字符串的副本函数原型为char*strcpy (char*dest, char*src) 。

3. 编写一个算法，计算子串T在主串S中出现的次数。

第7章 数组

数组可看成是一种扩展的线性结构，其特殊性不像栈和队列限制操作数据元素，而在于数据元素的构成上。数组中的数据元素可以是单个元素也可以是一个线性表。本章主要介绍数组的定义、数组的顺序存储与实现、特殊矩阵的压缩存储、稀疏矩阵的压缩存储。

本章重点和难点：

- 数组在内存中的存储地址与数组下标之间的关系
- 特殊矩阵的压缩存储
- 稀疏矩阵的三元组表示与实现

7.1 数组的定义及抽象数据类型

数组是一种特殊的线性表，表中的元素可以是原子类型，也可以是一个线性表。

7.1.1 重新认识数组

学过C语言的对数组不会陌生，数组是一般高级语言都支持的数据类型，下面从数据结构的角度来认识下数组。**数组**（Array）是存储在n个连续内存单元相同类型数据元素的一种线性结构。从逻辑结构上看，数组可以看做一般线性表的扩展。一维数组即为线性表，二维数组可以定义为“数据元素为一维数组（线性表）”的线性表。依次类推，即可得到多维数组的定义。

一个形式化的数组描述为一个含有n个元素的一维数组可以表示成线性表 $A = (a_0, a_1, \dots, a_{n-1})$ 。其中， a_i ($0 \leq i \leq n-1$) 是表A中的元素，元素个数是n。

一个m行n列的二维数组可以看成是每个元素由列向量构成的线性表，例如，图7-1所示的二维数组可以表示成 $A = (p_0, p_1, \dots, p_r)$ ($r=n-1$)，每个元素 p_j ($0 \leq j \leq r$) 又是一个列向量，即 $p_j = (a_{0,j}, a_{1,j}, \dots, a_{m-1,j})$ ($0 \leq j \leq n-1$)。

同理，图7-1所示的二维数组也可以表示成每个元素由行向量构成的线性表，例如，线性表A可以表示成 $B = (q_0, q_1, \dots, q_{s-1})$ ($s=m-1$)， q_j ($0 \leq j \leq m-1$) 是一个行向量，即 $q_j = (a_{j,0}, a_{j,1}, \dots, a_{j,n-1})$ 。如图7-2所示。

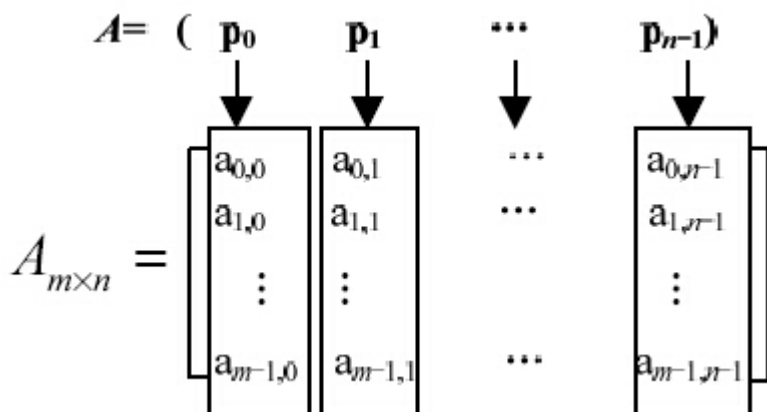


图7-1 每个元素看成列向量的二维数组

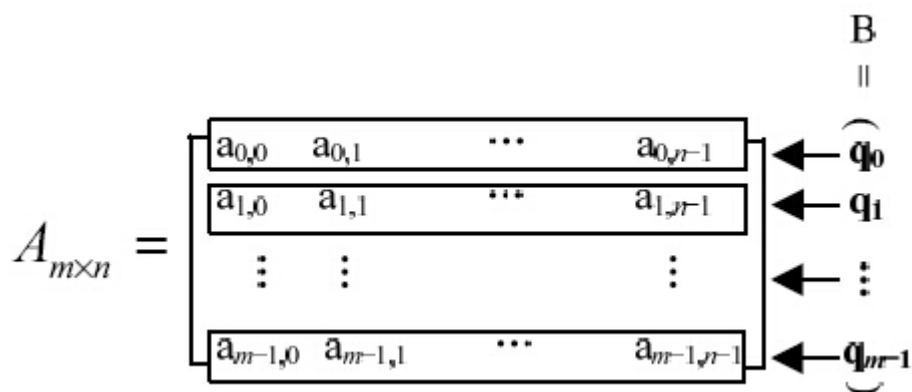


图7-2 每个元素看成是行向量的二维数组

因此，二维数组可看做线性表的线性表。同理，一个n维数组也可以看成是一个线性表，其中线性表中的每个数据元素是n-1维的数组。

n维数组中的每个元素处于n个向量中，每个元素有n个前驱元素，也有n个后继元素。

注意 数组的定义与线性表的定义相似，但线性表仅定义为n个相同类型数据元素的线性结构，并不要求其数据元素一定存储在连续的内存单元中；而数组不仅定义为n个相同类型数据元素的线性结构，而且要求其数据元素一定存储在连续的内存单元中。

7.1.2 数组的抽象数据类型

1. 数据对象集合

数组的数据对象集合为 $\{a_{j_1 j_2 \dots j_n} \mid n (> 0)\}$ 称为数组的维数， $j_i = 0, 1, \dots, b_{i-1}$ ，其中， $0 \leq i \leq n$ 。 b_i 是数组的第i维长度， j_i 是数组的第i维下标}。在一个二维数组中，若把数组看成是由列向量组成的线性表，那么元素 a_{ij} 的前驱元素是 $a_{i-1, j}$ ，后继元素是 $a_{i+1, j}$ ；若把数组看成是由行向量组成的线性表，那么元素 a_{ij} 的前驱元素是 $a_{i, j-1}$ ，后继元素是 $a_{i, j+1}$ 。

数组是一个特殊的线性表。

2. 基本操作集合

(1) InitArray (&A, n, bound1, ..., boundn) : 初始化操作。
如果维数和各维的长度合法, 则构造数组A, 并返回1, 表示成功。

(2) DestroyArray (&A) : 销毁数组操作。

(3) GetValue (A, &e, index1, ..., indexn) : 返回数组的元素操作。如果下标合法, 将数组A中对应的元素赋值给e, 并返回1, 表示成功。

(4) AssignValue (&A, e, index1, ..., indexn) : 设置数组的元素值操作。如果下标合法, 将数组A中由下标index1, ..., indexn指定的元素值置为e。

(5) LocateArray (A, ap, &offset) : 数组的定位操作。根据数组的元素下标, 求出该元素在数组中的相对地址。

7.2 数组的顺序表示与实现

由于一般不对数组进行插入和删除操作，也就是说，数组一旦建立，则结构中的数据元素个数和元素之间的关系就不再变动，因此，数组理所当然采用的是顺序存储结构。本节主要介绍数组的顺序存储结构及顺序存储结构下的基本操作。

7.2.1 数组的顺序存储结构

计算机中的存储器结构是一维（线性）结构，而数组是一个多维结构，如果要将一个多维结构存放在一个一维的存储单元里，这就需要先将其多维的数组转换成一个一维线性序列，才能将其存放在存储器中。

数组的存储方式有两种，一种是以行序为主序（row major order）的存储方式，另一种是以列序为主序（column major order）的存储方式，对于如图7-3所示的数组A来说，二维数组A以行序为主序的存储顺序为 $a_{0,0}, a_{0,1}, \dots, a_{0,n-1}, a_{1,0}, a_{1,1}, \dots, a_{1,n-1}, \dots, a_{m-1,0}, a_{m-1,1}, \dots, a_{m-1,n-1}$ ，以列序为主序的存储顺序为 $a_{0,0}, a_{1,0}, \dots, a_{m-1,0}, a_{0,1}, a_{1,1}, \dots, a_{m-1,1}, \dots, a_{0,n-1}, a_{1,n-1}, \dots, a_{m-1,n-1}$ 。数组A在计算机内存中的存储形式如图7-3所示。

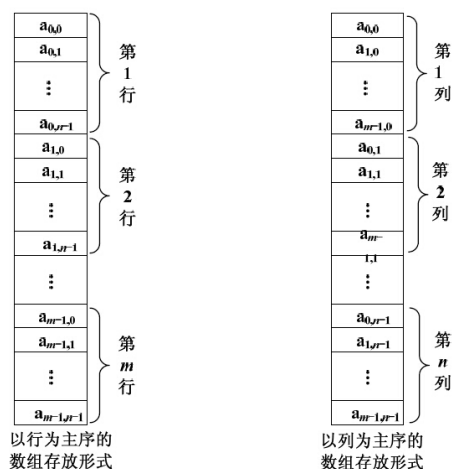


图7-3 数组在内存中的存放形式

根据数组的维数和各维的长度就能为数组分配存储空间。因为数组中的元素连续存放，所以任意给定一个数组的下标，就可以求出相应数组元素的存储位置。

下面说明以行序为主序的数组元素的存储地址与数组的下标之间的关系。设每个元素占 m 个存储单元，则二维数组 A 中的任何一个元素 a_{ij} 的存储位置可以由以下公式确定。

$$\text{Loc}(i, j) = \text{Loc}(0, 0) + (i \times n + j) \times m$$

其中， $\text{Loc}(i, j)$ 表示元素 a_{ij} 的存储地址， $\text{Loc}(0, 0)$ 表示元素 a_{00} 的存储地址，即二维数组的起始地址（也称为基地址）。

推广到更一般的情况，可以得到 n 维数组中数据元素的存储地址与数组的下标之间的关系为 $\text{Loc}(j_1, j_2, \dots, j_n) = \text{Loc}(0, 0, \dots, 0) + (b_1 * b_2 * \dots * b_{n-1} * j_0 + b_2 * b_3 * \dots * b_{n-1} * j_1 + \dots + b_{n-1} * j_{n-2} + j_{n-1}) * L$

其中， b_i ($1 \leq i \leq n-1$) 是第 i 维的长度， j_i 是数组的第 i 维下标。

数组的顺序存储结构类型定义如下。

```
#define MaxArraySize 3
#include<stdarg.h> /*
标准头文件，包含va_start
、va_arg
、va_end
宏定义*/
typedef struct
{
    DataType *base; /*
数组元素的基地址*/
    int dim; /*
数组的维数*/
    int *bounds; /*
数组的每一维之间的界限的地址*/
    int *constants; /*
数组存储映像常量基地址*/
}Array;
```

其中，base是数组元素的基地址，dim是数组的维数，bounds是数组的每一维之间的界限的地址，constants是数组存储映像常量基地址。

数组的顺序存储表示如图7-4所示。

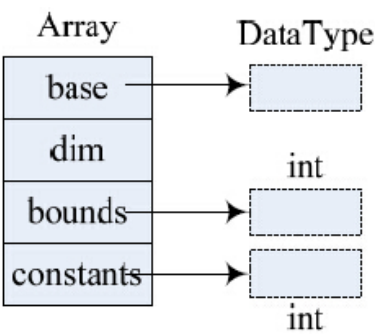


图7-4 数组的顺序存储结构

7.2.2 数组的基本运算

数组的基本运算实现如下所示（以下算法的实现保存在文件“SeqArray.h”中）。

①初始化数组。初始化数组就是根据数组的维数和各维的长度构造一个数组，为数组所有元素分配内存单元，确定每一维的界限地址，便于以后为数组赋值和获取数组中的元素。构造成功则返回1表示成功。数组的初始化算法实现如下。

```
int InitArray(Array *A,int dim,...)
/*
、... ..
```

```

初始化数组*/
{
    int elemtotal=1,i;                                /*elemtotal
是数组元素总数，初值为1*/
    va_list ap;
    if(dim<1||dim>MaxArraySize)                        /*
如果维数不合法，返回0*/
        return 0;
    A->dim=dim;
    A->bounds=(int *)malloc(dim*sizeof(int)); /*
分配一个dim
大小的内存单元*/
    if(!A->bounds)
        exit(-1);
    va_start(ap,dim);                                /*dim
是一个固定参数，即可变参数的前一个参数*/
    for(i=0;i<dim;++i)
    {
        A->bounds[i]=va_arg(ap,int);                /*
依次取得可变参数，即各维的长度*/
        if(A->bounds[i]<0)
            return -1; //
在math.h
中定义为4
        elemtotal*=A->bounds[i];                    /*
得到数组中元素总的个数*/
    }
    va_end(ap);
    A->base=(DataType *)malloc(elemtotal*sizeof(DataType)); /*
为数组分配所有元素分配内存空间*/
    if(!A->base)
        exit(-1);
    A->constants=(int *)malloc(dim*sizeof(int));        /*
为数组的常量基址分配内存单元*/
    if(!A->constants)
        exit(-1);
    A->constants[dim-1]=1;
    for(i=dim-2;i>=0;--i)
        A->constants[i]=A->bounds[i+1]*A->constants[i+1];
    return 1;
}

```

若维数dim和各维长度合法，则构造相应的数组A如图7-5所示。

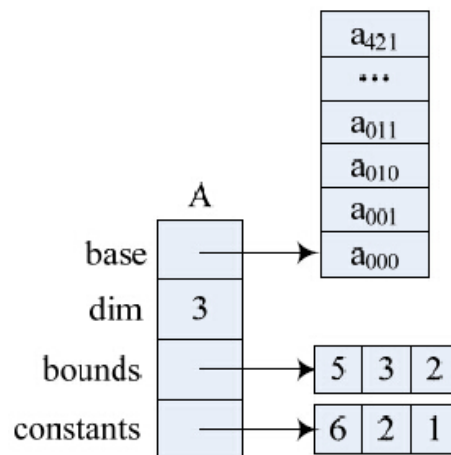


图7-5 数组A[5][3][2]的存储结构

在初始化数组时，使用变长参数表传递参数，即用“...”表示形式参数。这种变长的形式参数主要用于参数不确定的情况，数组可能是二维的，也可能是三维的，参数的个数也就会不同，在这种情况下，就需要在被调用函数的形式参数列表中使用变长形式参数。在函数的形式参数表中，至少要有一个固定的参数在变长的形式参数表的前面。

在具体的实现当中，可使用C语言头文件stdarg.h包含的宏va_list、va_arg、va_start和va_end。首先定义一个指向可变参数的指针ap，然后调用va_start (ap, dim)使ap指向dim的下一参数，即第一个变长参数，然后调用va_arg (ap, int)返回可变参数的值，最后使用完毕va_end (ap)结束对可变参数的获取。

②数组的赋值，代码如下。

```
int AssignValue(Array A,DataType e,...)
/*
数组的赋值操作。将e
的值赋给的指定的数组元素*/
{
    va_list ap;
    int offset;
    va_start(ap,e);
    if(LocateArray(A,ap,&offset)==0)          /*
找到元素在数组中的相对位置*/
        return 0;
    va_end(ap);
    *(A.base+offset)=e;                      /*
将e
赋给该元素*/
    return 1;
}
```

利用宏va_start得到指向变长参数的指针，然后调用定位函数LocateArray (A, ap, &offset)得到元素在数组中的偏移值，最后将元素e赋给该元素。

③返回数组中指定的元素。利用宏va_start获得指向变长参数的指针，然后调用定位函数LocateArray (A, ap, &offset)得到元素在数组中的偏移值，最后将该元素的值赋给e。返回数组中指定的元素的算法实现如下。

```
int GetValue(DataType *e,Array A, ...)
/*
返回数组中指定的元素，将指定的数组的下标的元素赋给e*/
{
    va_list ap;
```

```

        int offset;
        va_start(ap,A);
        if(LocateArray(A,ap,&offset)==0)           /*
找到元素在数组中的相对位置*/
            return 0;
        va_end(ap);
        *e=(A.base+offset);                       /*
将元素值赋给e*/
        return 1;
    }

```

④数组定位。根据给定数组中元素的下标，求出该元素在数组中的相对位置。利用宏va_arg(ap, int)从类型为va_list的参数ap中依次得到传递来的每一维的长度，然后与数组映像常量基地址相乘，得到数组元素的偏移地址。数组的定位算法实现如下。

```

int LocateArray(Array A,va_list ap,int *offset)
/*
根据数组中元素的下标，求出该元素在A
中的相对地址offset*/
{
    int i,instand;
    *offset=0;
    for(i=0;i<A.dim;i++)
    {
        instand=va_arg(ap,int); /*
依次返回可变参数列表中的参数值，即每一维的长度*/
        if(instand<0||instand>=A.bounds[i])
            return 0;
        *offset+=A.constants[i]*instand;
    }
    return 1;
}

```

⑤销毁数组，代码如下。

```

void DestroyArray(Array *A)
/*
销毁数组。释放内存单元*/
{
    if(A->base)
        free(A->base);
    if(A->bounds)
        free(A->bounds);
    if(A->constants)
        free(A->constants);
    A->base=A->bounds=A->constants=NULL;           /*
将各个指针指向空*/
    A->dim=0;
}

```

7.2.3 数组应用举例

【例7-1】 利用数组的基本运算对数组进行初始化，然后输出数组的值。

【分析】 主要考查数组中元素的地址与下标之间的转换关系，帮助读者理解并熟练掌握数组的基本操作实现，学会利用元素的下标和根据基地址求元素在数组中的相对地址，并掌握涉及可变参数的宏va_list、va_arg、va_start和va_end的使用。

数组操作的实现代码如下。

```
/*
包含头文件*/
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
#include<stdarg.h>          /*
包含va_start
、va_arg
、va_end
宏定义*/
typedef int DataType;
#include"SeqArray.h"
void main()
{
    Array A;
    int i,j,k;
    int dim=3,bound1=5,bound2=3,bound3=2;          /*
初始化数组的维数和各维的长度*/
    DataType e;
    InitArray(&A,dim,bound1,bound2,bound3);          /*
构造一个5×3×2
的三维数组A*/
    printf("
数组A
的各维的长度是:");
    for(i=0;i<dim;i++)          /*
输出数组A
的各维的长度*/
        printf("%3d",A.bounds[i]);
    printf("\n
数组A
的常量基址是:");
    for(i=0;i<dim;i++)          /*
输出数组A
的常量基址*/
        printf("%3d",A.constants[i]);
    printf("\n%d*%d*%d
的数组元素如下:\n",bound1,bound2,bound3);
    for(i=0;i<bound1;i++)
    {
        for(j=0;j<bound2;j++)
        {
            for(k=0;k<bound3;k++)
            {
                AssignValue(A,100*i+10*j+k,i,j,k);          /*
将元素赋给A*/
                GetValue(&e,A,i,j,k);          /*
将数组A
中的元素赋给e*/
                printf("A[%d][%d][%d]=%3d\t",i,j,k,e); /*
输出数组A
中的元素*/
            }
            printf("\n");
        }
        printf("\n");
    }
    printf("
按照数组的线性序列输出元素,
即利用基地址输出元素:\n");
    for(i=0;i<bound1*bound2*bound3;i++)          /*
按照线性序列输出数组A
中的元素*/
    {
```

```

        printf("
第%d
个元素=%3d\t", i+1, A.base[i]);
        if ((i+1) % bound2 == 0)
            printf("\n");
    }
    DestroyArray(&A);
}

```

程序的运行结果如图7-6所示。

```

D:\零基础学数据结构\例7_1\Debug\例7_1.exe
数组A的各维的长度是: 5 3 2
数组A的常量基址是: 6 2 1
5*3*2的数组元素如下:
A[0][0][0]= 0  A[0][0][1]= 1
A[0][1][0]= 10 A[0][1][1]= 11
A[0][2][0]= 20 A[0][2][1]= 21

A[1][0][0]=100 A[1][0][1]=101
A[1][1][0]=110 A[1][1][1]=111
A[1][2][0]=120 A[1][2][1]=121

A[2][0][0]=200 A[2][0][1]=201
A[2][1][0]=210 A[2][1][1]=211
A[2][2][0]=220 A[2][2][1]=221

A[3][0][0]=300 A[3][0][1]=301
A[3][1][0]=310 A[3][1][1]=311
A[3][2][0]=320 A[3][2][1]=321

A[4][0][0]=400 A[4][0][1]=401
A[4][1][0]=410 A[4][1][1]=411
A[4][2][0]=420 A[4][2][1]=421

按照数组的线性序列输出元素,即利用基地址输出元素:
第1个元素- 0 第2个元素- 1 第3个元素- 10
第4个元素- 11 第5个元素- 20 第6个元素- 21
第7个元素- 100 第8个元素- 101 第9个元素- 110
第10个元素- 111 第11个元素- 120 第12个元素- 121
第13个元素- 200 第14个元素- 201 第15个元素- 210
第16个元素- 211 第17个元素- 220 第18个元素- 221
第19个元素- 300 第20个元素- 301 第21个元素- 310
第22个元素- 311 第23个元素- 320 第24个元素- 321
第25个元素- 400 第26个元素- 401 第27个元素- 410
第28个元素- 411 第29个元素- 420 第30个元素- 421
Press any key to continue

```

图7-6 数组操作程序运行结果

7.3 特殊矩阵的压缩存储

矩阵是科学计算、工程数学，尤其是数值分析经常研究的对象。在高级语言中，通常使用二维数组来存储矩阵。在有些高阶矩阵中，非零元素非常少，此时若使用二维数组将造成存储空间的浪费，这时可只存储部分元素，从而提高存储空间的利用率。这种存储方式称为矩阵的**压缩存储**。所谓压缩存储指的是为多个相同值的元素只分配一个存储单元，对值为零的元素不分配存储单元。

非零元素非常少（远小于 $m \times n$ ）或元素分布呈一定规律的矩阵称为特殊矩阵。本节主要介绍特殊矩阵（对称矩阵、三角矩阵、对角矩阵）的压缩存储。

7.3.1 对称矩阵的压缩存储

如果一个 n 阶的矩阵 A 中的元素满足 $a_{ij} = a_{ji}$ （ $0 \leq i, j \leq n-1$ ），则称这种矩阵为 **n 阶对称矩阵**。

对于对称矩阵，每一对对称元素值相同，只需要为每一对对称元素分配一个存储空间，这样就可以将 n^2 个元素存储在 $n(n+1)/2$ 个存储单元里。 n 阶对称矩阵 A 和下三角矩阵如图7-7所示。

$$A_{n \times n} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \quad \text{对称矩阵}$$

$$A_{n \times n} = \begin{bmatrix} a_{0,0} & & & \\ a_{1,0} & a_{1,1} & & \\ \vdots & \vdots & \ddots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \quad \text{下三角矩阵}$$

图7-7 n阶对称矩阵与下三角矩阵

假设用一维数组s存储对称矩阵A的上三角或下三角元素，则一维数组s的下标k与n阶对称矩阵A的元素 a_{ij} 之间的对应关系为

$$k = \begin{cases} \frac{i*(i+1)}{2} + j, & \text{当 } i \geq j \\ \frac{j*(j+1)}{2} + i, & \text{当 } i < j \end{cases}$$

当 $i \geq j$ 时，矩阵A以下三角形式存储， $\frac{i*(i+1)}{2} + j$ 为矩阵A中元素的线性序列编号；当 $i < j$ 时，矩阵A以上三角形式存储， $\frac{j*(j+1)}{2} + i$ 为矩阵A中元素的线性序列编号。任意给定一组下标（i，j），就可以确定矩阵A在一维数组s中的存储位置。s称为n阶对称矩阵A的压缩存储。

矩阵的下三角元素的压缩存储表示如图7-8所示。

$k =$	0	1	2	3		$\frac{n*(n-1)}{2}$		$\frac{n*(n+1)}{2} - 1$
	a_{00}	a_{10}	a_{11}	a_{20}	...	$a_{n-1,0}$...	$a_{n-1,n-1}$

图7-8 对称矩阵的压缩存储

注意 一般情况下，以行序为主序存储矩阵中的元素。

7.3.2 三角矩阵的压缩存储

三角矩阵 可分为两种，为上三角矩阵和下三角矩阵。其中，下三角元素均为常数C或零的n阶矩阵称为**上三角矩阵**，上三角元素均为常数C或零的n阶矩阵称为**下三角矩阵**。n×n的上三角矩阵和下三角矩阵如图7-9所示。

$$A_{n \times n} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ & a_{1,1} & \cdots & a_{1,n-1} \\ & C & \ddots & \vdots \\ & & & a_{n-1,n-1} \end{bmatrix} \quad \text{上三角矩阵}$$

$$A_{n \times n} = \begin{bmatrix} a_{0,0} & & & \\ a_{1,0} & a_{1,1} & & C \\ \vdots & \vdots & \ddots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \quad \text{下三角矩阵}$$

图7-9 上三角矩阵与下三角矩阵

上三角矩阵的压缩原则是只存储上三角的元素，不存储下三角的零元素（或只用一个存储单元存储下三角的非零元素）。下三角矩阵的存储元素与上三角压缩存储类似。如果用一维数组来存储三角矩阵，则需要存储 $n * (n+1) / 2 + 1$ 个元素。一维数组的下标k与矩阵的下标（i, j）的对应关系如下。

$$k = \begin{cases} \frac{i * (2n - i + 1)}{2} + j - i & \text{当 } i \leq j \\ \frac{n * (n + 1)}{2} & \text{当 } i > j \end{cases} \quad \text{上三角矩阵}$$

$$k = \begin{cases} \frac{i * (i + 1)}{2} + j, & \text{当 } i \geq j \\ \frac{n * (n + 1)}{2}, & \text{当 } i < j \end{cases} \quad \text{下三角矩阵}$$

其中，第 $k = \frac{n*(n+1)}{2}$ 个位置存放的是常数C或者零元素。上述公式可根据等差数列推导得出。

关于一个以行为主序与以列为主序压缩存储相互转换的情况，例如，设有一个 $n \times n$ 的上三角矩阵A的上三角元素已按行为主序连续存放在数组b中，请设计一个算法trans将b中元素按列为主序连续存放在数组c中。当 $n=5$ 时，矩阵A如图7-10所示。

$$A_{5 \times 5} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 6 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 12 \\ 0 & 0 & 0 & 13 & 14 \\ 0 & 0 & 0 & 0 & 15 \end{bmatrix}$$

图7-10 5×5 上三角矩阵

其中， $b = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)$ ， $c = (1, 2, 6, 3, 7, 10, 4, 8, 11, 13, 5, 9, 12, 14, 15)$ 。那如何根据数组b得到c呢？

【分析】 本题主要考查特殊矩阵的压缩存储中对数组下标的灵活使用程度。用i和j分别表示矩阵中元素的行列下标，用k表示压缩矩阵b元素的下标。解答本题的关键是找出以行为主序和以列为主序数组下标的对应关系（初始时， $i=0$ ， $j=0$ ， $k=0$ ），即

$c[j * (j+1) / 2 + i] = b[k]$, 其中, $j * (j+1) / 2 + i$ 就是根据等差数列得出的。根据这种对应关系, 直接把 b 中的元素赋给 c 中对应的位置即可。但是读出 c 中一行即 b 中的一行 (元素 1、2、3、4、5) 之后, 还要改变行下标 i 和列下标 j , 开始读 6、7、8 元素时, 列下标 j 需要从 1 开始, 行下标 i 也需要增加 1, 依次类推, 可以得出修改行下标和列下标的办法为, 当一行还没有结束时, $j++$; 否则 $i++$ 并修改下一行的元素个数及 i 、 j 的值, 直到 $k = n(n+1) / 2$ 为止。

根据以上分析, 相应的压缩矩阵转换算法如下。

```
void trans(int b[],int c[],int n)
/*
将b
中元素按列为主序连续存放到数组c
中*/
{
    int step=n,count=0,i=0,j=0,k;
    for(k=0;k<n*(n+1)/2;k++)
    {
        count++;
        /*
记录一行是否读完*/
        c[j*(j+1)/2+i]=b[k];/*
把以行为主序的数存放到对应以列为主序的数组中*/
        if(count==step)/*
一行读完后*/
        {
            step--;
            count=0;/*
下一行重新开始计数*/
            i++;/*
下一行的开始行*/
            j=n-step;/*
一行读完后,下一轮的开始列*/
        }
        else
            j++;/*
一行还没有读完,继续下一列的数*/
    }
}
```

7.3.3 对角矩阵的压缩存储

对角矩阵（也叫带状矩阵）是另一类特殊的矩阵。所谓对角矩阵，就是所有的非零元素都集中在以主对角线为中心的带状区域内（对角线的个数为奇数）。也就是说除了主对角线和主对角线上、下若干条对角线上的元素外，其他元素的值均为零。一个3对角矩阵如图7-11所示。

通过观察，可以发现以上对角矩阵具有以下特点。

当 $i=0$ 、 $j=1, 2$ 时，即第一行有2个非零元素；当 $0 < i < n-1$ ， $j=i-1, i, i+1$ 时，即第2行到第 $n-1$ 行之间有3个非零元素；当 $i=n-1$ 、 $j=n-2, n-1$ 时，即最后一行有2个非零元素。除此以外，其他元素均为零。

除了第1行和最后1行的非零元素为2个，其余各行非零元素为3个，因此，若用一维数组存储这些非零元素，需要 $2+3*(n-2)+2=3n-2$ 个存储单元。对角矩阵的压缩存储在数组中的情况如图7-12所示。

$$A_{6 \times 6} = \begin{bmatrix} 8 & 5 & 0 & 0 & 0 & 0 \\ 2 & 12 & 9 & 0 & 0 & 0 \\ 0 & 6 & 5 & 11 & 0 & 0 \\ 0 & 0 & 10 & 7 & 6 & 0 \\ 0 & 0 & 0 & 9 & 3 & 7 \\ 0 & 0 & 0 & 0 & 2 & 15 \end{bmatrix}$$

图7-11 3对角矩阵

$k=$	0	1	2	3	4	5	6	7	$3*n-3$
矩阵	a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	$a_{n-1,n-1}$

图7-12 对角矩阵的压缩存储

下面确定一维数组的下标k与矩阵中元素的下标(i, j)之间的关系。先确定下标为(i, j)的元素与第一个元素之间在一维数组中的关系, Loc(i, j)表示 a_{ij} 在一维数组中的位置, Loc(0, 0)表示第一个元素的在一维数组中的地址。

$\text{Loc}(i, j) = \text{Loc}(0, 0) + \text{前}i-1\text{行的非零元素个数} + \text{第}i\text{行的非零元素个数}$, 其中, 前 $i-1$ 行的非零元素个数为 $3 * (i-1) - 1$, 第 i 行的非零元素个数为 $j-i+1$ 。其中,

$$t_i = \begin{cases} -1, & \text{当} i > j \\ 0, & \text{当} i = j \\ 1, & \text{当} i < j \end{cases}$$

因此, $\text{Loc}(i, j) = \text{Loc}(0, 0) + 3 * i + j - i = \text{Loc}(0, 0) + 2 * i + j$,
则 $\text{Loc}(i, j) = \text{Loc}(0, 0) + 2 * i + j$ 。

7.4 稀疏矩阵的压缩存储

稀疏矩阵中的大多数元素是零，为了节省存储单元，需要对稀疏矩阵进行压缩存储。本节主要介绍稀疏矩阵的定义、稀疏矩阵的抽象数据类型、稀疏矩阵的三元组表示及算法实现。

7.4.1 什么是稀疏矩阵

所谓**稀疏矩阵**，假设在 $m \times n$ 矩阵中有 t 个元素不为零，令 $\delta = \frac{t}{m \times n}$ ， δ 为矩阵的稀疏因子，如果 $\delta \leq 0.05$ ，则称矩阵为稀疏矩阵。通俗来讲，若矩阵中大多数元素值为零，只有很少的非零元素，这样的矩阵就是稀疏矩阵。

例如，图7-13所示是一个 6×7 的稀疏矩阵。

$$M_{6 \times 7} = \begin{bmatrix} 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 2 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 4 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

图7-13 6×7 稀疏矩阵

7.4.2 稀疏矩阵抽象数据类型

1. 数据对象集合

在C语言中，稀疏矩阵其实是一个特殊的二维数组。数组中的元素大多数是零，只有少数的非零元素。从数据结构的角度上看，稀疏矩阵可看成是线性表的线

性表。

2. 基本操作集合

(1) CreateMatrix (&M) : 创建稀疏M。根据输入的行号、列号和元素值创建稀疏矩阵。

(2) DestroyMatrix (&M) : 销毁稀疏矩阵M。将稀疏矩阵的行数、列数、非零元素的个数置为零。

(3) PrintMatrix (M) : 打印稀疏矩阵中的元素。按照以行为主序或以列为主序输出稀疏矩阵的元素。

(4) CopyMatrix (M, &N) : 稀疏矩阵的复制操作。由稀疏矩阵M复制得到稀疏矩阵N。

(5) AddMatrix (M, N, &Q) : 稀疏矩阵的相加操作。将两个稀疏矩阵M和N的对应行和列的元素相加，将结果存入稀疏矩阵Q。

(6) SubMatrix (M, N, &Q) : 稀疏矩阵的相减操作。将两个稀疏矩阵M和N的对应行和列的元素相减，将结果存入稀疏矩阵Q。

(7) MultMatrix (M, N, &Q) : 稀疏矩阵的相乘操作。将两个稀疏矩阵M和N相乘，将结果存入稀疏矩阵Q。

(8) TransposeMatrix (M, &N) : 稀疏矩阵的转置操作。将稀疏矩阵M中的元素对应的行和列互换，得到转置矩阵N。

7.4.3 稀疏矩阵的三元组表示

为了节省内存单元，需要对稀疏矩阵进行压缩存储。在进行压缩存储的过程中，我们可以只存储稀疏矩阵的非零元素，为了表示非零元素在矩阵中的位置，还需存储非零元素对应的行和列的位置（i，j）。即可以通过存储非零元素的行号、列号和元素值实现稀疏矩阵的压缩存储，这种存储表示称为稀疏矩阵的三元组表示。三元组的结点结构如图7-14所示。

图7-14中的非零元素可以用三元组（（0，3，6），（1，1，3），（2，2，7），（2，3，2），（3，0，9），（3，4，-2），（4，2，4），（4，3，3），（5，4，8））表示。将这些三元组按照行序为主序存放在结构体数组中，如图7-15所示，其中k表示数组的下标。

i	j	e
非零元素 的行号	非零元素 的列号	非零元 素的值

图7-14 稀疏矩阵的三元组结点结构

k	i	j	e
0	0	3	6
1	1	1	3
2	2	2	7
3	2	3	2
4	3	0	9
5	3	4	-2
6	4	2	4
7	4	3	3
8	5	4	8

图7-15 稀疏矩阵的三元组存储结构

一般情况下，数组采用顺序存储结构，采用顺序存储结构的三元组称为三元组顺序表。三元组顺序表的类型描述如下。

```
#define MaxSize 200
typedef struct          /*
三元组类型定义*/
{
    int i;              /*
非零元素的行号*/
    int j;              /*
非零元素的列号*/
    DataType e;
}Triple;
typedef struct          /*
矩阵类型定义*/
{
    Triple data[MaxSize];
    int m;              /*
矩阵的行数*/
    int n;              /*
矩阵的列数*/
    int len;            /*
矩阵中非零元素的个数*/
}TriSeqMatrix;
```

7.4.4 稀疏矩阵的三元组实现

稀疏矩阵的基本运算的算法实现如下（算法实现保存在文件“TriSeqMatrix.h”中）。

①创建稀疏矩阵。根据输入的行号、列号和元素值，创建一个稀疏矩阵。注意按照行优先顺序输入。创建成功返回1，否则返回0。算法实现如下。

```
int CreateMatrix(TriSeqMatrix *M)
/*
创建稀疏矩阵（按照行优先顺序输入非零元素值）*/
{
    int i,m,n;
    DataType e;
    int flag;
    printf("
请输入稀疏矩阵的行数、列数及非零元素个数: ");
    scanf("%d,%d,%d",&M->m,&M->n,&M->len);
    if(M->len>MaxSize)
        return 0;
    for(i=0;i<M->len;i++)
    {
        do
        {
            printf("
请按行序顺序输入第%d
个非零元素所在的行(0
~%d),
列(0
~%d),
.....
");
```

```

元素值:",
                                i+1,M->m-1,M->n-1);
                                scanf("%d,%d,%d",&m,&n,&e);
                                flag=0;
初始化标志位*/
                                if (m<0||m>M->m||n<0||n>M->n)
                                flag=1;
如果行号或列号正确,标志位为1*/
                                /*
若输入的顺序正确,则标志位为1*/
                                if (i>0&&m<M->data[i-1].i||m==M->data[i-1].i&&n<=M->data[i-1].j)
                                flag=1;
                                }while(flag);
                                M->data[i].i=m;
                                M->data[i].j=n;
                                M->data[i].e=e;
                                }
                                return 1;
}

```

②复制稀疏矩阵。为了得到稀疏矩阵M的一个副本N,只需将稀疏矩阵M的非零元素的行号、列号及元素值依次赋给矩阵N的行号、列号及元素值。复制稀疏矩阵的算法实现如下。

```

void CopyMatrix(TriSeqMatrix M,TriSeqMatrix *N)
/*
由稀疏矩阵M
复制得到另一个副本N*/
{
    int i;
    N->len=M.len;
修改稀疏矩阵N
的非零元素的个数*/
    N->m=M.m;
修改稀疏矩阵N
的行数*/
    N->n=M.n;
修改稀疏矩阵N
的列数*/
    for(i=0;i<M.len;i++) /*
把M
中非零元素的行号、列号及元素值依次赋值给N
的行号、列号及元素值*/
    {
        N->data[i].i=M.data[i].i;
        N->data[i].j=M.data[i].j;
        N->data[i].e=M.data[i].e;
    }
}

```

③转置稀疏矩阵。转置稀疏矩阵就是将矩阵中元素由原来的存放位置 (i, j) 变为 (j, i), 也就是说将元素的行列互换。例如, 图7-13所示的6×7矩阵, 经过转置后变为7×6矩阵, 并且矩阵中的元素也要以主对角线为准进行交换。

将稀疏矩阵转置的方法是将矩阵M的三元组中的行和列互换，就可以得到转置后的矩阵N，如图7-16所示。稀疏矩阵的三元组顺序表转置过程如图7-17所示。

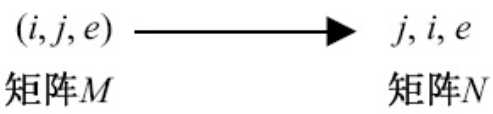


图7-16 稀疏矩阵转置

行列下标互换后，还需要将行、列下标重新进行排序，才能保证转置后的矩阵也是以行序优先存放的。为了避免这种排序，以矩阵中列顺序优先的元素进行转置，然后按照顺序依次存放到转置后的矩阵中，这样经过转置后得到的三元组顺序表正好是以行序为主序存放的。具体算法实现大致有两种：

（1）逐次扫描三元组顺序表M，第1次扫描M，找到j=0的元素，将行号和列号互换后存入到三元组顺序表N中，即找到（3，0，9），将行号和列号互换，把（3，0，9）直接存入N中，作为N的第一个元素。然后第2次扫描M，找到j=1的元素，将行号和列号互换后存入到三元组顺序表N中；依次类推，直到所有元素都存放至N中，最后得到的三元组顺序表N如图7-18所示。

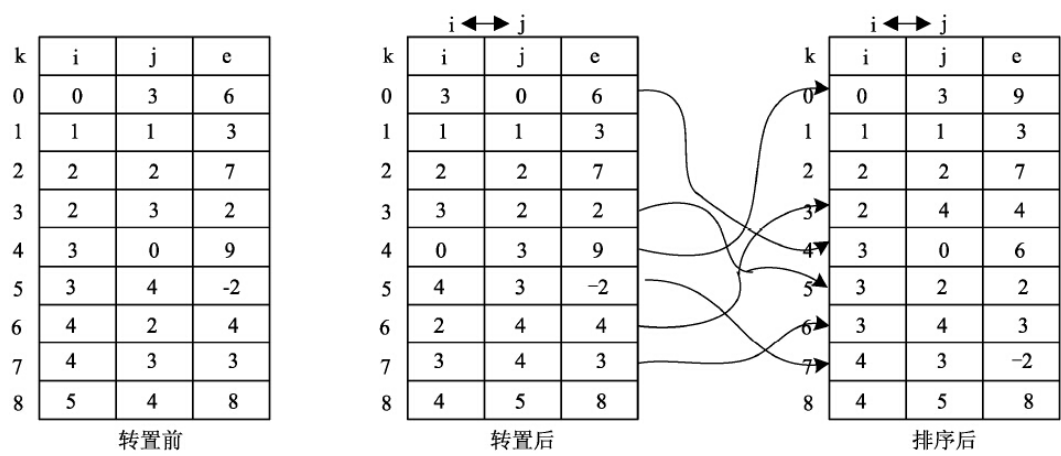


图7-17 矩阵转置的三元组表示

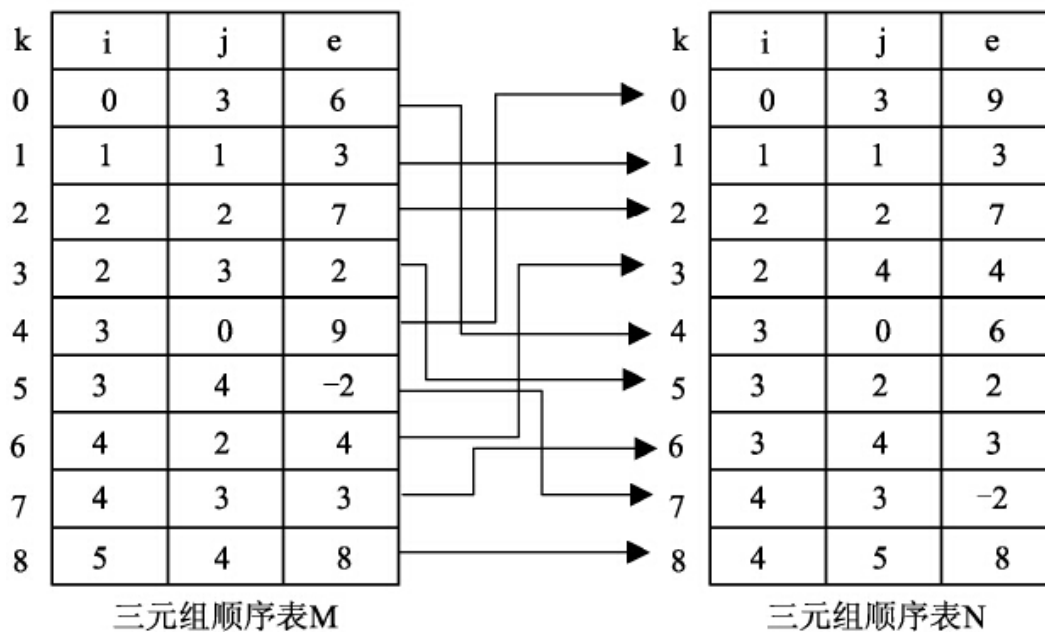


图7-18 稀疏矩阵转置的三元组顺序表表示

稀疏矩阵转置的算法实现如下。

```

void TransposeMatrix(TriSeqMatrix M, TriSeqMatrix *N)
/*
稀疏矩阵的转置*/
{
    int i, k, col;
    N->m=M.n;
    N->n=M.m;
    N->len=M.len;
    if (N->len)
    {
        k=0;
        for(col=0; col<M.n; col++) /*
按照列号扫描三元组顺序表*/
            for(i=0; i<M.len; i++)
                if (M.data[i].j==col) /*
如果元素的列号是当前列，则进行转置*/
                {
                    N->data[k].i=M.data[i].j;
                    N->data[k].j=M.data[i].i;
                    N->data[k].e=M.data[i].e;
                    k++;
                }
    }
}

```

通过分析该转置算法，其时间主要耗费在for语句的两层循环上，故算法的时间复杂度是 $O(n \times \text{len})$ ，即与M的列数及非零元素的个数成正比。我们知道，一般矩阵的转置算法为：

```

for(col=0;col<M.n;++col)
    for(row=0;row<M.len;row++)
        N[col][row]=M[row][col];

```

其时间复杂度为 $O(n*m)$ 。当非零元素的个数 len 与 $m*n$ 同数量级时，稀疏矩阵的转置算法时间复杂度就变为 $O(m*n^2)$ 了。假设在 200×500 的矩阵中，有 $len=20\ 000$ 个非零元素，虽然三元组存储节省了存储空间，但时间复杂度提高了，因此稀疏矩阵的转置仅适用于 $len \ll m*n$ 的情况。

(2) 稀疏矩阵的快速转置。按照 M 中三元组的次序进行转置，并将转置后的三元组置入 N 中恰当位置。若能预先确定矩阵 M 中的每一列第一个非零元在 N 中的应有位置，那么对 M 中的三元组进行转置时，便可直接放到 N 中的恰当位置。

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 N 中的应有位置。

设置两个数组 num 和 $position$ ， $num[col]$ 表示三元组顺序表 M 中第 col 列的非零元素个数， $position[col]$ 表示 M 中的第 col 列的第一个非零元素在 N 中的恰当位置。

依次扫描三元组顺序表 M ，可以得到每一列非零元素的个数，即 $num[col]$ 。 $position[col]$ 的值可以由 $num[col]$ 得到，显然， $position[col]$ 与 $num[col]$ 存在如下关系。

$position[0]=0;$

$position[col]=position[col-1]+num[col-1]$ ，其中 $1 \leq col \leq M.n-1$ 。

例如，图7-13所示的稀疏矩阵的 $num[col]$ 和 $position[col]$ 的值如表7-1所示。

表7-1 矩阵 M 的 $num[col]$ 与 $position[col]$ 的值

列号 col	0	1	2	3	4	5	6
num[col]	1	1	2	3	2	0	0
position[col]	0	1	2	4	7	9	9

算法实现如下所示。

```

void FastTransposeMatrix(TriSeqMatrix M, TriSeqMatrix *N)
/*
稀疏矩阵的快速转置运算*/
{
    int i, k, t, col, *num, *position;
    num = (int *) malloc((M.n+1) * sizeof(int)); /*
数组num
用于存放M
中的每一列非零元素个数*/
    position = (int *) malloc((M.n+1) * sizeof(int)); /*
数组position
用于存放N
中每一行中非零元
素的第一个位置*/

    N->n = M.m;
    N->m = M.n;
    N->len = M.len;
    if (N->len)
    {
        for (col = 0; col < M.n; ++col)
            num[col] = 0; /*
初始化num
数组*/

        for (t = 0; t < M.len; t++) /*
计算M
中每一列非零元素的个数*/
            num[M.data[t].j]++;

        position[0] = 0; /*N
中第一行的第一个非零元素的序号为0*/
        for (col = 1; col < M.n; col++) /*
将N
中第col
行的第一个非零元素的位置*/
            position[col] = position[col-1] + num[col-1];

        for (i = 0; i < M.len; i++) /*
依据position
对M
进行转置，存入N*/
        {
            col = M.data[i].j;
            k = position[col]; /*
取出N
中非零元素应该存放的位置，赋值给k*/
            N->data[k].i = M.data[i].j;
            N->data[k].j = M.data[i].i;
            N->data[k].e = M.data[i].e;
            position[col]++; /*
修改下一个非零元素应该存放的位置*/
        }
    }
    free(num);
    free(position);
}

```

先扫描M，得到M中每一列非零元素的个数，存放到num中。然后根据num[col]和position[col]的关系，求出N中每一行第一个非零元素的位置。初始时，

position[col]是M的第col列第一个非零元素的位置，每个M中的第col列的非零元素存入N中，则将position[col]加1，使position[col]的值始终为下一个要转置的非零元素应存放的位置。

该算法中有4个并列的单循环，循环次数分别为n和M.len，因此总的时间复杂度为 $O(n+len)$ 。当M的非零元素个数len与 $m*n$ 处于同一个数量级时，算法的时间复杂度变为 $O(m*n)$ ，与经典的矩阵转置算法时间复杂度相同。

(3) 销毁稀疏矩阵，代码如下。

```
void DestroyMatrix(TriSeqMatrix *M)
/*
销毁稀疏矩阵*/
{
    M->m=M->n=M->len=0;
}
```

7.5 稀疏矩阵应用举例

上一节介绍了稀疏矩阵的定义及稀疏矩阵的三元组顺序存储的算法实现，接下来通过分析并实现两个矩阵相加和相乘的算法，帮读者巩固对稀疏矩阵相关算法的掌握。

7.5.1 三元组表示的稀疏矩阵相加

【例7-2】 有两个稀疏矩阵A和B，相加得到C，如图7-19所示。请利用十字链表实现两个稀疏矩阵的相加，并输出结果。

$$A_{4 \times 4} = \begin{bmatrix} 0 & 5 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -2 \end{bmatrix} \quad B_{4 \times 4} = \begin{bmatrix} 0 & 0 & 4 & 0 \\ 0 & -3 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 \end{bmatrix} \quad C_{4 \times 4} = \begin{bmatrix} 0 & 5 & 4 & 0 \\ 3 & -3 & 0 & 2 \\ 0 & 0 & 3 & 0 \\ 8 & 0 & 0 & -2 \end{bmatrix}$$

图7-19 十字链表表示的稀疏矩阵的相加

【提示】 矩阵中两个元素相加可能会出现如下3种情况。

(1) A中的元素 $a_{ij} \neq 0$ 且B中的元素 $b_{ij} \neq 0$ ，但是结果可能为零，如果结果为零，则不保存元素值；如果结果不为零，则将结果保存在C中。

(2) A中的第 (i, j) 个位置存在非零元素 a_{ij} ，而B中不存在非零元素，则只需要将该值赋值给 C_{ij} 。

(3) B中的第(i, j)个位置存在非零元素 b_{ij} ，而A中不存在非零元素，则只需要将 b_{ij} 赋值给 C_{ij} 。

两个稀疏矩阵相加的算法实现如下。

```
int AddMatrix(TriSeqMatrix A, TriSeqMatrix B, TriSeqMatrix *C)
/*
将两个矩阵A
和B
对应的元素值相加，得到另一个稀疏矩阵C*/
{
    int m=0, n=0, k=-1;
    if (A.m!=B.m || A.n!=B.n) /*
如果两个矩阵的行数与列数不相等，则不能够进行相加运算*/
        return 0;
    C->m=A.m;
    C->n=A.n;
    while (m<A.len && n<B.len)
    {
        switch (CompareElement(A.data[m].i, B.data[n].i)) /*
比较两个矩阵对应元素的行号*/
        {
            case -1:
                C->data[++k]=A.data[m++]; /*
将矩阵A
，即行号小的元素赋值给C*/
                break;
            case 0:
                /*
如果矩阵A
和B
的行号相等，则比较列号*/
                switch (CompareElement(A.data[m].j, B.data[n].j))
                {
                    case -1: /*
如果A
的列号小于B
的列号，则将矩阵A
的元素赋值给C*/
                        C->data[++k]=A.data[m++];
                        break;
                    case 0: /*
如果A
和B
的行号、列号均相等，则将两元素相加，存入C*/
                        C->data[++k]=A.data[m++];
                        C->data[k].e+=B.data[n++].e;
                        if (C->data[k].e==0) /*
如果两个元素的和为0
，则不保存*/
                            k--;
                        break;
                    case 1: /*
如果A
的列号大于B
的列号，则将矩阵B
的元素赋值给C*/
                        C->data[++k]=B.data[n++];
```

```

        }
        break;
        case 1:
            /*
            如果A
            的行号大于B
            的行号，则将矩阵B
            的元素赋值给C*/
            C->data[++k]=B.data[n++];
        }
    }
    while (m<A.len)
        /*
        如果矩阵A
        的元素还没处理完毕，则将A
        中的元素赋值给C*/
        C->data[++k]=A.data[m++];
    while (n<B.len)
        /*
        如果矩阵B
        的元素还没处理完毕，则将B
        中的元素赋值给C*/
        C->data[++k]=B.data[n++];
    C->len=k+1;
    /*
    修改非零元素的个数*/
    if (k>MaxSize)
        return 0;
    return 1;
}

```

m和n分别为矩阵A和B的当前处理的非零元素下标，初始时为0。需要特别注意的是，最后求得的非零元素个数为k+1，k为非零元素最后一个元素的下标。

比较两个矩阵的元素值的函数算法实现如下。

```

int CompareElement(int a,int b)
/*
比较两个矩阵的元素值大小。前者小于后者，返回-1
；相等，返回0
；大于，返回1*/
{
    if (a<b)
        return -1;
    if (a==b)
        return 0;
    return 1;
}

```

测试函数代码如下。

```

/*
包含头文件*/
#include<stdlib.h>
#include<stdio.h>
#include<malloc.h>
/*
稀疏矩阵类型定义*/
#define MaxSize 200
typedef int DataType;
typedef struct                                /*
三元组类型定义*/
{
    int i,j;
    DataType e;
}Triple;
typedef struct                                /*
矩阵类型定义*/
{
    Triple data[MaxSize];
    int rpos[MaxSize];
    int m,n,len;                                /*
矩阵的行数，列数和非零元素的个数*/
}TriSeqMatrix;
/*
函数声明*/
int AddMatrix(TriSeqMatrix A,TriSeqMatrix B,TriSeqMatrix *C);
void PrintMatrix(TriSeqMatrix M);
int CreateMatrix(TriSeqMatrix *M);            /*
创建稀疏矩阵函数在文件TriSeqMatrix.h
中*/
int CompareElement(int a,int b);
void main()
{
    TriSeqMatrix M,N,Q;
    CreateMatrix(&M);
    PrintMatrix(M);
    CreateMatrix(&N);
    PrintMatrix(N);
    AddMatrix(M,N,&Q);
    PrintMatrix(Q);
}
void PrintMatrix(TriSeqMatrix M)
/*
输出稀疏矩阵*/
{
    int i;
    printf("
稀疏矩阵是%d
行×%d
列，共%d
个非零元素。\\n",M.m,M.n,M.len);
    printf("
行
列
元素值\\n");
    for(i=0;i<M.len;i++)
        printf("%2d%6d%8d\\n",M.data[i].i,M.data[i].j,M.data[i].e);
}

```

程序运行结果如图7-20所示。

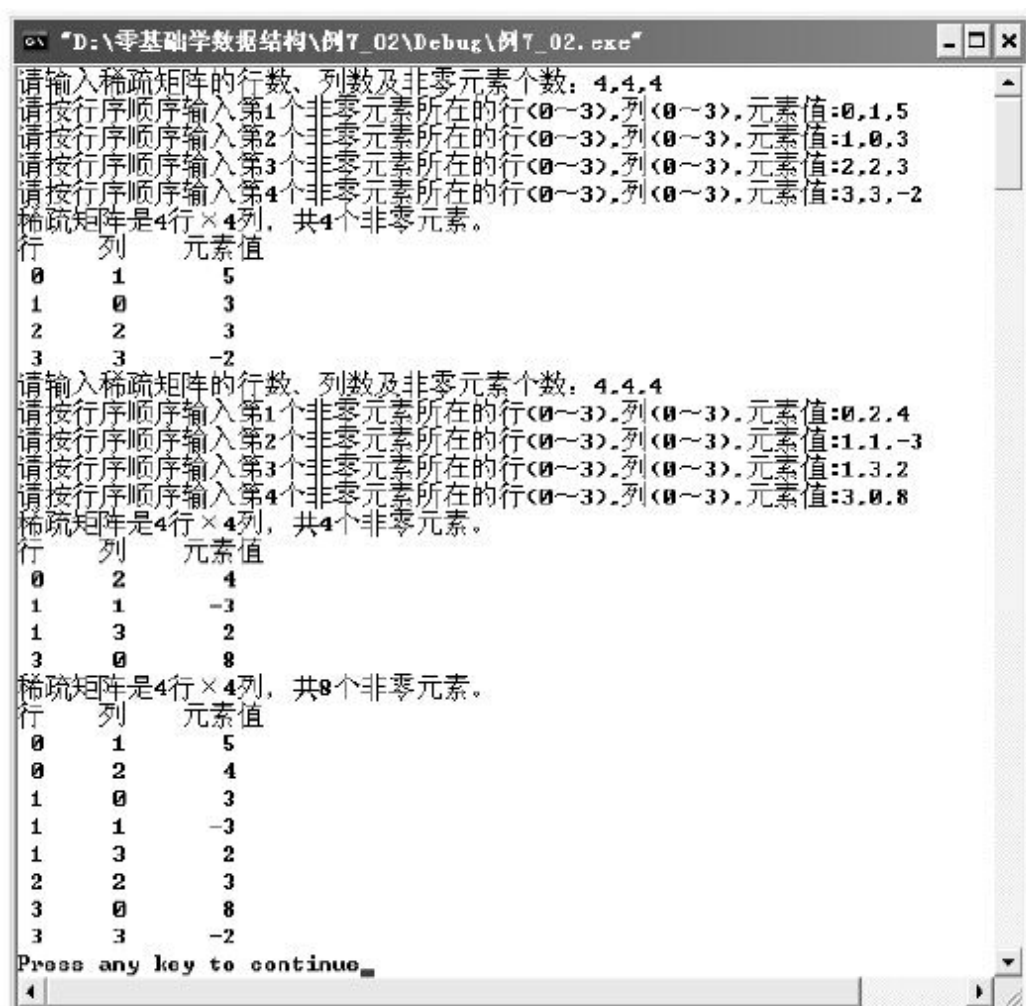


图7-20 两个稀疏矩阵相加程序运行结果

两个稀疏矩阵A和B相减的算法实现与相加算法实现类似，只需要将相加算法中的+改成-即可，也可以将第二个矩阵的元素值都乘上-1，然后调用矩阵相加的函数即可。稀疏矩阵相减的算法实现如下。

```

int SubMatrix(TriSeqMatrix A,TriSeqMatrix B,TriSeqMatrix *C)
/*
稀疏矩阵的相减*/
{
    int i;
    for(i=0;i<B.len;i++)

```



```

        B.data[i].e*=-1;                                /*
将矩阵B
的元素都乘-1
, 然后将两个矩阵相加*/
        return AddMatrix(A,B,C);
    }

```

7.5.2 三元组表示的稀疏矩阵相乘

假设矩阵M是 $m_1 \times n_1$ 的矩阵, N是 $m_2 \times n_2$ 的矩阵, 如果矩阵M的列数与矩阵N的行数相等, 即 $n_1 = m_2$, 则两个矩阵M和N是可以相乘的。

两个矩阵相乘的计算公式为 $Q[i][j] = \sum_{k=0}^{n_1-1} M[i][k] \times N[k][j]$, $0 \leq i < m_1, 0 \leq j < n_2$, 相应地,

两个矩阵相乘的经典算法用C语言程序描述如下。

```

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

```

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

1. 算法思想

当M和N为稀疏矩阵并采用三元组存储时, 就不能套用上述算法。下面通过具体例子来探讨稀疏矩阵相乘的三元组算法实现。

【例7-3】 设有采用三元组顺序表存储的两个稀疏矩阵M和N试编写一个算法，实现M和N相乘。M和N相乘后得到结果Q，如图7-21所示。

$$M_{3 \times 4} = \begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad N_{4 \times 2} = \begin{bmatrix} 0 & 3 \\ 1 & 0 \\ 0 & 0 \\ 0 & 2 \end{bmatrix} \quad Q_{3 \times 2} = \begin{bmatrix} 0 & 19 \\ 2 & 0 \\ 0 & 3 \end{bmatrix}$$

图7-21 矩阵相乘

它们的三元组M.data、N.data和Q.data如图7-22所示。

k	i	j	e
0	0	0	3
1	0	3	5
2	1	1	2
3	2	0	8

(a) 三元组顺序表M

k	i	j	e
0	0	1	3
1	1	0	1
2	3	1	2

(b) 三元组顺序表N

k	i	j	e
0	0	1	19
1	1	0	2
2	2	1	3

(c) 三元组顺序表Q

图7-22 矩阵M、N、Q的三元组顺序表

在矩阵相乘的经典算法中，不管M[i][k]和N[k][j]是否为零，都要进行一次乘法运算，而实际上，只要两者有一个值为零时，其乘积一定为零。在两个稀疏矩阵相乘时，应避免这种无效操作，换句话说，为求两个稀疏矩阵的乘积，只需在M.data和N.data中找出相应的各对元素（即M.data中的j值和N.data中i值相等的各对元素）相乘即可。

例如，M.data[0]表示的非零元素(0, 0, 3)只需要和N.data[0]表示的非零元素(0, 1, 3)相乘，M.data[1]表示的非零元素(0, 3, 5)只需要和N.data[2]表示的非零元素(3, 1, 2)相乘，这是因为(0, 0, 3)的列号与(0, 1, 3)的行号相等，(0, 3, 5)的列号与(3, 1, 2)的行号相等。

注意 两个稀疏矩阵的乘积不一定是稀疏矩阵，反之，矩阵的每一个 $M(i, k) * N(k, j)$ 不为零，其累加和 $Q[i][j]$ 可能为零。因此，乘积矩阵Q中的元素是否为非零元素，只有在求得累加和后才能知道。由于乘积矩阵Q和M中的元素的行号一致，又M中的元素排列是以M的行序为主序的，由此可对Q进行逐行处理，先求得累加和的中间结果，在一行结束时，判断该中间变量的累加和是否为零，如果为零，再将其存入Q中的相应位置。

在求矩阵的每一行的累加和时，需要设置两个数组num和rpos，其中num[row]保存三元组顺序表中的每一行的非零元素个数，rpos[row]保存三元组顺序表中第row行的第一个非零元素的位置。num[row]与rpos[row]的关系如下。

$rpos[0]=0;$

$rpos[row]=rpos[row-1]+num[row-1], 1 \leq row \leq M.m \text{ 或 } N.m$

在算法实现过程中，还需要设置一个变量tp，用来控制三元组顺序表中每一行元素的边界（每一行多少个非零元素），作为循环控制条件的结束，代码如下。

```
if(arow<A.m-1)
    tp=A.rpos[arow+1];
else
    tp=A.len;
```

图7-19中的稀疏矩阵M和N对应的num[row]与rpos[row]的值如表7-2所示。

表7-2 矩阵M和N的num[row]和rpos[row]的值

(a)矩阵 M 的 num[row]和 rpos[row]的值					(b) 矩阵 N 的 num[row]和 rpos[row]的值					
行号 row	0	1	2	(3)	行号 row	0	1	2	3	(4)
num[row]	2	1	1		num[row]	1	1	0	1	
rpos[row]	0	2	3	4	rpos[row]	0	1	2	2	3

修改后的三元组顺序表的类型定义如下。

```
#define MaxSize 100
typedef int DataType;
typedef struct                                /*
三元组类型定义*/
{
    int i,j;
    DataType e;
}Triple;
typedef struct                                /*
矩阵类型定义*/
{
    Triple data[MaxSize];
    int rpos[MaxSize]; /*
用于存储三元组中的每一行的第一非零元素的位置*/
    int m,n,len;        /*
矩阵的行数，列数和非零元素的个数*/
}TriSeqMatrix;
```

与前面的三元组顺序表相比，增加了一个成员数组rpos，用于存储三元组中每一行的第一个非零元素位置。

2. 算法实现

(1) 根据num[row]和rpos[row]之间的关系，先求出num[row]和rpos[row]的值。

(2) 依次扫描矩阵A中的每一行，找到A中列号与B中行号相等的非零元素，将其元素值相乘，存入数组temp中。具体做法为扫描A中的每一行元素，然后取出第row行的元素，并将其列号赋给brow，即brow=A.data[p].j，在矩阵B中找到行号为brow的元素（即q=B.rpos[brow]），取出其列号ccol=B.data[q].j，最后计算A和B对应元素的乘积，并存入数组中，即temp[ccol]+=A.data[p].e*B.data[q].e。

按照以上方法将所得元素值累加，一行结束后，将结果为非零元素存入C中，即可得到矩阵的乘积。

两个矩阵相乘的算法实现如下。

```
void MultMatrix(TriSeqMatrix A,TriSeqMatrix B,TriSeqMatrix *C)
/*
稀疏矩阵A
和B
相乘得到C*/
{
    int i,k,tp,tq,p,q,arow,brow,ccol;
    int temp[MaxSize];
    ... ..
}
```

```

累加器*/
        int num[MaxSize];
        if (A.n!=B.m)                                /*
如果矩阵A
的列与B
的行不相等，则返回*/
        return;
        C->m=A.m;                                    /*
初始化C
的行数、列数和非零元素的个数*/
        C->n=B.n;
        C->len=0;
        if (A.len*B.len==0)                            /*
只要有一个矩阵的长度为0
，则返回*/
        return;
        /*
求矩阵B
中每一行第一个非零元素的位置*/
        for (i=0;i<B.m;i++)                            /*
初始化num*/
        num[i]=0;
        for (k=0;k<B.len;k++)                            /*num
存放矩阵B
中每一行非零元素的个数*/
        {
            i=B.data[k].i;
            num[i]++;
        }
        B.rpos[0]=0;
        for (i=1;i<B.m;i++)                            /*rpos
存放矩阵B
中每一行第一个非零元素的位置*/
        B.rpos[i]=B.rpos[i-1]+num[i-1];
        /*
求矩阵A
中每一行第一个非零元素的位置*/
        for (i=0;i<A.m;i++)                            /*
初始化化num*/
        num[i]=0;
        for (k=0;k<A.len;k++)
        {
            i=A.data[k].i;
            num[i]++;
        }
        A.rpos[0]=0;
        for (i=1;i<A.m;i++)                            /*rpos
存放矩阵A
中每一行第一个非零元素的位置*/
        A.rpos[i]=A.rpos[i-1]+num[i-1];
        /*
计算两个矩阵的乘积-*/
        for (arow=0;arow<A.m;arow++)                    /*
依次扫描矩阵A
的每一行*/
        {
            for (i=0;i<B.n;i++)                            /*
初始化累加器temp*/
            temp[i]=0;
            C->rpos[arow]=C->len;
            /*
对每个非0
元处理*/
            if (arow<A.m-1)
                tp=A.rpos[arow+1];
            else
                tp=A.len;

```

```

        for (p=A.rpos[arow];p<tp;p++)
        {
            brow=A.data[p].j;                                /*
取出A
中的列号*/

            if (brow<B.m-1)
                tq=B.rpos[brow+1];
            else
                tq=B.len;
            for (q=B.rpos[brow];q<tq;q++) /*

依次取出B
中的第brow
行，与A
中的元素相乘*/

            {
                ccol=B.data[q].j;
                temp[ccol]+=A.data[p].e*B.data[q].e; /*

把乘积存入temp
中*/

            }
        }
        for (ccol=0;ccol<C->n;ccol++) /*

将temp
中元素依次赋值给C*/

        {
            if (temp[ccol])
            {
                if (++C->len>MaxSize)
                    return;
                C->data[C->len-1].i=arow;
                C->data[C->len-1].j=ccol;
                C->data[C->len-1].e=temp[ccol];
            }
        }
    }
}

```

测试代码如下。

```

/*
包含头文件*/
#include<stdlib.h>
#include<stdio.h>
#include<malloc.h>
/*
稀疏矩阵类型定义*/
#define MaxSize 200
typedef int DataType;
typedef struct                                     /*
三元组类型定义*/
{
    int i,j;
    DataType e;
}Triple;
typedef struct                                     /*
矩阵类型定义*/
{
    Triple data[MaxSize];
    int rpos[MaxSize];

    int m,n,len;                                     /*
矩阵的行数、列数和非零元素的个数*/
}TriSeqMatrix;
/*
.....

```

```

函数声明*/
void MultMatrix(TriSeqMatrix A, TriSeqMatrix B, TriSeqMatrix *C);
void PrintMatrix(TriSeqMatrix M);
int CreateMatrix(TriSeqMatrix *M); /*
创建稀疏矩阵函数在文件TriSeqMatrix.h
中*/
void main()
{
    TriSeqMatrix M, N, Q;
    CreateMatrix(&M);
    PrintMatrix(M);
    CreateMatrix(&N);
    PrintMatrix(N);
    MultMatrix(M, N, &Q);
    PrintMatrix(Q);
}
void PrintMatrix(TriSeqMatrix M)
/*
稀疏矩阵的输出*/
{
    int i;
    printf("
稀疏矩阵是%d
行×%d
列，共%d
个非零元素。\\n", M.m, M.n, M.len);
    printf("
行
列
元素值\\n");
    for(i=0; i<M.len; i++)
        printf("%2d%6d%8d\\n", M.data[i].i, M.data[i].j, M.data[i].e);
}

```

程序运行结果如图7-23所示。


```

D:\零基础学数据结构\例7_2\Debug\例7_2.exe
请输入稀疏矩阵的行数、列数及非零元素个数: 3,4,4
请按行序顺序输入第0个非零元素所在的行<1~3>,列<1~4>,元素值:0,0,3
请按行序顺序输入第1个非零元素所在的行<1~3>,列<1~4>,元素值:0,3,5
请按行序顺序输入第2个非零元素所在的行<1~3>,列<1~4>,元素值:1,1,2
请按行序顺序输入第3个非零元素所在的行<1~3>,列<1~4>,元素值:2,0,1
稀疏矩阵是3行×4列,共4个非零元素。
行   列   元素值
0    0    3
0    3    5
1    1    2
2    0    1
请输入稀疏矩阵的行数、列数及非零元素个数: 4,2,3
请按行序顺序输入第0个非零元素所在的行<1~4>,列<1~2>,元素值:0,1,3
请按行序顺序输入第1个非零元素所在的行<1~4>,列<1~2>,元素值:1,0,1
请按行序顺序输入第2个非零元素所在的行<1~4>,列<1~2>,元素值:3,1,2
稀疏矩阵是4行×2列,共3个非零元素。
行   列   元素值
0    1    3
1    0    1
3    1    2
稀疏矩阵是3行×2列,共3个非零元素。
行   列   元素值
0    1    19
1    0    2
2    1    3
Press any key to continue_

```

图7-23 稀疏矩阵相乘的程序运行结果

该算法的时间复杂度是 $O(A.len * B.n)$ ，当 $A.len$ 与 $A.m * A.n$ 同数量级时，该算法的时间复杂度接近一般矩阵的相乘运算的时间复杂度 $O(A.m * A.n * B.n)$ 。

7.6 稀疏矩阵的十字链表表示与实现

从前面的例子可以看出，采用三元组顺序表对两个稀疏矩阵的相加和相乘运算时，算法实现比较复杂，且需要移动大量的元素，时间复杂度也会增加。为了避免以上问题，可以采用稀疏矩阵的另一种存储结构——链式存储。

7.6.1 稀疏矩阵的十字链表表示

当矩阵中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。在这种情况下，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，每个非零元可用含有5个域的结点表示，其中， i 、 j 和 e 分别表示非零元素的行号、列号和元素值，向右指针域 $right$ 用以链接同一行中下一个非零元，向下指针域 $down$ 用以链接同一列中下一个非零元。

同一行的非零元由 $right$ 链接构成一个线性链表，同一列的非零元由 $down$ 链接构成一个线性链表。每个非零元素既是某一行链表的一个元素，又是某一系列链表的一个元素，整个链表构成一个十字交叉的形状，故称这样的存储结构为十字链表。

在十字链表中，再增加一个指向行链表的头指针和指向列链表的头指针，每一行的头指针和列指针存放在一维数组中。十字链表中的结点如图7-24所示。



图7-24 十字链表中的结点结构

例如，3×4矩阵对应的十字链表如图7-25所示。

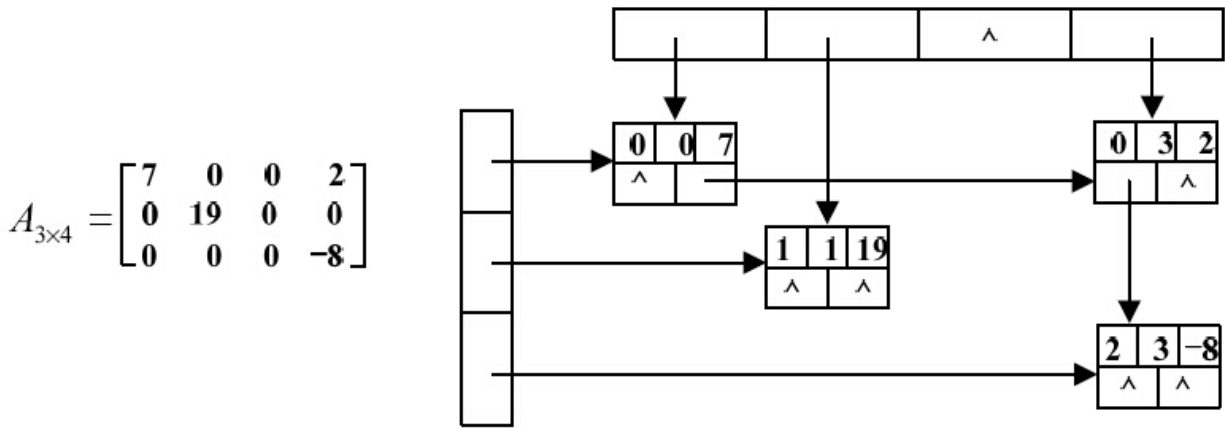


图7-25 稀疏矩阵的十字链表表示

若该元素的同一行或同一列没有非零元素，则将right或down置为 \wedge 。

十字链表的类型描述如下。

```
typedef struct OLNode
{
    int i,j;
```

```
        DataType e;
        struct OListNode *right,*down;
    }OListNode,*OLink;
    typedef struct
    {
        OLink *rowhead,*colhead;
        int m,n,len;
    }CrossList;
```

其中，i和j分别表示稀疏矩阵中非零元素的行号和列号，e为非零元素值，right指向同一行的下一个非零元素，down指向同一列的下一个非零元素。rowhead和colhead分别存放指向行链表和列链表的指针，m和n分别表示稀疏矩阵的行数和列数，len表示稀疏矩阵中非零元素的个数。

7.6.2 十字链表的基本运算

稀疏矩阵的十字链表的基本操作的算法实现如下（算法实现保存在文件“CrossList.h”中）。

①初始化稀疏矩阵。需要将十字链表的行链表和列链表的指针置为NULL，并将系数矩阵的行数、列数和非零元素个数置为零。初始化稀疏矩阵的算法实现如下。

```
void InitMatrix(CrossList *M)
/*
初始化稀疏矩阵*/
{
    M->rowhead=M->colhead=NULL;
    M->m=M->n=M->len=0;
}
```

②稀疏矩阵的插入操作。稀疏矩阵的插入操作与创建操作类似。稀疏矩阵的插入操作就是将一个新结点插入稀疏矩阵中，分为行插入和列插入两个步骤插入新结点。行插入和列插入的执行步骤是类似的，如果是行或列的第一个元素，则直接插入；否则找到要插入的位置后插入。稀疏矩阵的插入操作算法实现如下。

```

void InsertMatrix(CrossList *M,OLink p)
/*
按照行序将p
插入到稀疏矩阵中*/
{
    OLink q=M->rowhead[p->i];          /*q
指向待插行链表*/
    if(!q||p->j<q->j)                    /*
待插的行表空或p
所指结点的列值小于首结点的列值，则直接插入*/
    {
        p->right=M->rowhead[p->i];
        M->rowhead[p->i]=p;
    }
    else
    {
        while(q->right&&q->right->j<p->j)/*q
所指不是尾结点且q
的下一结点的列值小于p
所指结点的列值*/
        {
            q=q->right;
            p->right=q->right;
            q->right=p;
        }
        q=M->colhead[p->j];              /*q
指向待插列链表*/
        if(!q||p->i<q->i)                  /*
待插的列表空或p
所指结点的行值小于第一个结点的行值*/
        {
            p->down=M->colhead[p->j];
            M->colhead[p->j]=p;
        }
        else
        {
            while(q->down&&q->down->i<p->i)/*q
所指不是尾结点且q
的下一结点的行值小于p
所指结点的行值*/
            {
                q=q->down;
                p->down=q->down;
                q->down=p;
            }
            M->len++;
        }
    }
}

```

在十字链表中插入新结点*p的过程如图7-26所示。

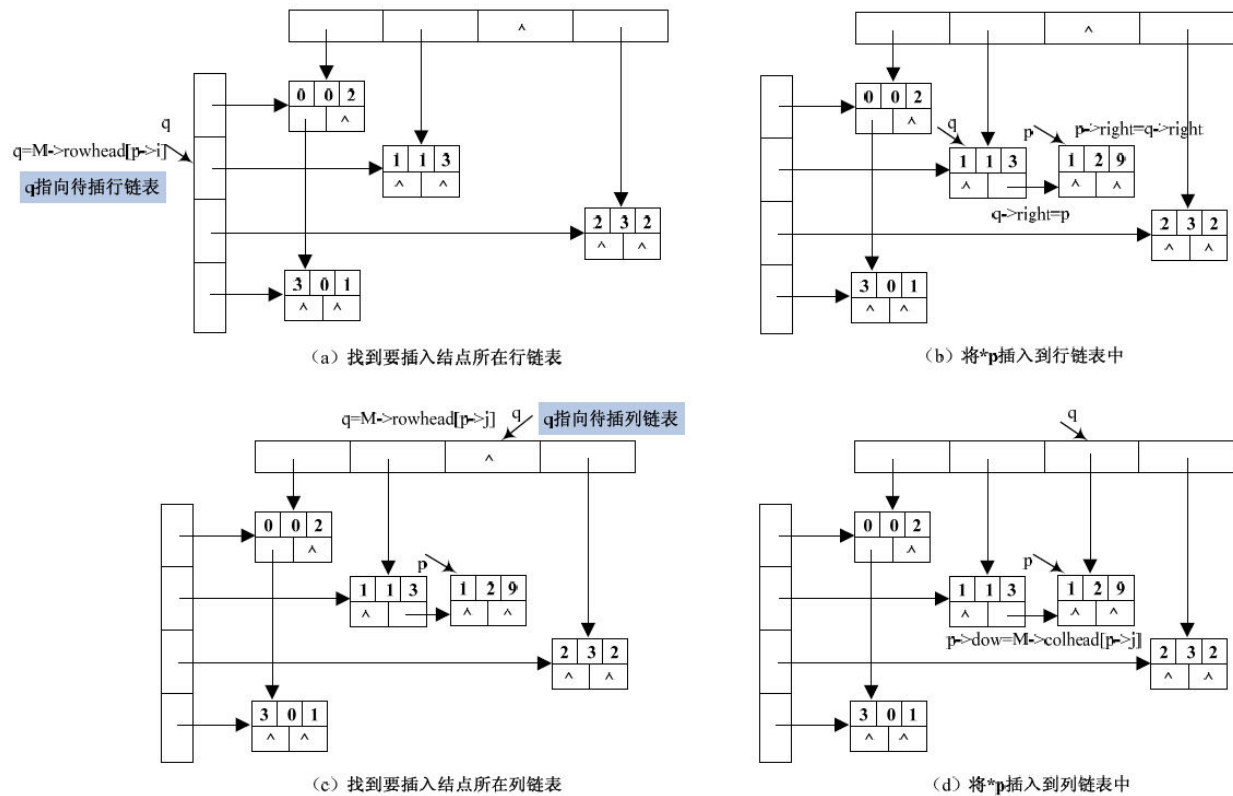


图7-26 在十字链表中插入新结点*p的过程

③稀疏矩阵的销毁。销毁稀疏矩阵需要按行依次释放结点。销毁稀疏矩阵的算法实现如下。

```
void DestroyMatrix(CrossList *M)
/*
销毁稀疏矩阵*/
{
    int i;
    OLink p, q;
    for (i=0; i<M->m; i++)
    {
        p = *(M->rowhead+i);
        while (p)
        {
            q = p;
            p = p->right;
            free(q);
        }
    }
}
```

```
        free(M->rowhead);  
        free(M->colhead);  
        InitMatrix(M);  
    }
```

7.7 小结

数组是一种扩展类型的线性表，元素 a_i 可以是原子也可以是一个线性表。

一般情况下，数组的存放是以顺序存储结构的形式存放。采用顺序存储结构的数组具有随机存取的特点，方便数组中元素的查找等操作。在C语言中，矩阵通常以二维数组存储。

常见的特殊矩阵有对称矩阵、三角矩阵和对角矩阵3种。特殊矩阵可以通过转换，存储在一个一维数组中，这种存储方式可以节省存储空间，称为特殊矩阵的压缩存储。

稀疏矩阵也需要压缩存储，稀疏矩阵的压缩存储通常分为稀疏矩阵的三元组顺序表表示和稀疏矩阵的十字链表表示两种方式。

三元组顺序表通过存储矩阵中非零元素的行号、列号和非零元素值，来唯一确定该元素及在矩阵中的位置。三元组顺序表通常利用一个一维数组实现，采用的是顺序存储结构。

三元组顺序表在实现创建、复制、转置、输出等操作比较方便，但是在进行矩阵的相加和相乘的运算中，时间的复杂度比较高。

7.8 习题

一、选择题

1. 稀疏矩阵的常见压缩存储方法有（）两种。
 - A. 二维数组和三维数组
 - B. 三元组和散列表
 - C. 三元组和十字链表
 - D. 散列表和十字链表
2. 采用稀疏矩阵的三元组表形式进行压缩存储，若要完成对三元组表进行转置，只要将行和列对换，这种说法（）。
 - A. 正确
 - B. 错误
 - C. 无法确定
 - D. 以上均不对
3. 对一些特殊矩阵采用压缩存储的目的主要是为了（）。

- A. 表达变得简单
- B. 对矩阵元素的存取变得简单
- C. 去掉矩阵中的多余元素
- D. 减少不必要的存储空间的开销

4. 设矩阵A是一个对称矩阵，为了节省存储，将其下三角部分按行序存放在一维数组B[1, $n(n-1)/2$]中，对下三角部分中任一元素 a_{ij} ($i \geq j$)，在一维数组B的下标位置k的值是 ()。

- A. $i(i-1)/2 + j - 1$
- B. $i(i-1)/2 + j$
- C. $i(i+1)/2 + j - 1$
- D. $i(i+1)/2 + j$

5. 假设以三元组表表示稀疏矩阵，则与如图7-27所示三元组表对应的 4×5 的稀疏矩阵是（注：矩阵的行列下标均从1开始） ()。

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

图7-27 三元组表

$$A. \begin{pmatrix} 0 & -8 & 0 & 6 & 0 \\ 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -5 & 0 & 4 & 0 & 0 \end{pmatrix}$$

$$B. \begin{pmatrix} 0 & -8 & 0 & 6 & 0 \\ 7 & 0 & 0 & 0 & 3 \\ -5 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$C. \begin{pmatrix} 0 & -8 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 7 & 0 & 0 & 0 & 0 \\ -5 & 0 & 4 & 0 & 0 \end{pmatrix}$$

$$D. \begin{pmatrix} 0 & -8 & 0 & 6 & 0 \\ 7 & 0 & 0 & 0 & 0 \\ -5 & 0 & 4 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

二、算法设计题

1. 已知一个稀疏矩阵是以三元组顺序表存储，请写一个将三元组按矩阵形式输出的算法。
2. 编写一个用十字链表创建稀疏矩阵的算法。
3. 已知稀疏矩阵是以十字链表形式存储，请写一个两个矩阵相乘的算法。

4. 将矩阵顺时针旋转 90° 。例如，将一个 5×5 的矩阵顺时针旋转 90° ，如图7-28所示。

1	2	3	4	5	21	16	11	6	1
6	7	8	9	10	22	17	12	7	2
11	12	13	14	15	23	18	13	8	3
16	17	18	19	20	24	19	14	9	4
21	22	23	24	25	25	20	15	10	5

图7-28 旋转前后的矩阵

说明 对于任意 N 阶方阵，如果 N 是偶数，则矩阵构成 $N/2$ 圈；如果 N 是奇数，则矩阵构成 $(N-1)/2$ 圈。将矩阵顺时针旋转 90° ，就是将每圈的元素在4个方位依次轮换位置。例如，对于 5×5 矩阵来说，将6放在原来4的位置，4放在原来20的位置，20放在原来22的位置，22放在原来6的位置。

第8章 广义表

广义表是线性表的推广，也是一种扩展的线性数据结构。广义表被广泛应用于人工智能等领域的表处理语言Lisp中，在Lisp中，广义表作为一种基本的数据结构，就连程序语法也用一系列的广义表来表示。本章主要给大家介绍广义表的定义、广义表的顺序存储与实现、广义表的递归算法。

本章重点和难点：

- 广义表的头尾链表表示与算法实现
- 广义表的扩展线性链表表示与算法实现

8.1 广义表的定义及抽象数据类型

广义表是一种特殊的线性表，是线性表的扩展。广义表中的元素可以是单个元素，也可以是一个广义表。本节主要介绍广义表的定义和广义表的抽象数据类型。

8.1.1 什么是广义表

广义表，也称为**列表**（lists），是由 n 个类型相同的数据元素 $(a_1, a_2, a_3, \dots, a_n)$ 组成的有限序列。其中，广义表中的元素 a_i 可以是单个元素，也可以是一个广义表。

通常，广义表记做 $GL = (a_1, a_2, a_3, \dots, a_n)$ 。其中， GL 是广义表的名字， n 是广义表的长度。如果广义表中的 a_i 是单个元素，则称 a_i 是**原子**。如果广义表中的 a_i 是一个广义表，则称 a_i 是广义表的子表。

习惯上用大写字母表示广义表的名字，用小写字母表示原子。

在广义表 GL 中， a_1 称为广义表 GL 的**表头**（head），其余元素组成的表 (a_2, a_3, \dots, a_n) 称为广义表 GL 的**表尾**（tail）。广义

表是一个递归的定义，因为在描述广义表时又用到了广义表的概念。
如下是一些广义表的例子。

(1) $A = ()$ ，广义表A是长度为0的空表。

(2) $B = (a)$ ，B是一个长度为1且元素为原子的广义表（其实就是前面讨论过的一般的线性表）。

(3) $C = (a, (b, c))$ ，C是长度为2的广义表。其中，第1个元素是原子a，第2个元素是一个子表(b, c)。

(4) $D = (A, B, C)$ ，D是一个长度为3的广义表，这3个元素都是子表，第1个元素是一个空表A。

(5) $E = (a, E)$ ，E是一个长度为2的递归广义表，相当于
 $E = (a, (a, (a, (a, (a, \dots))))$ 。

由上述定义和例子可推出如下广义表的重要结论

(1) 广义表的元素既可以是原子，也可以是子表，子表的元素可以是元素，也可以是子表。广义表的结构是一个多层次的结构。

(2) 一个广义表还可以是另一个广义表的元素。例如A、B和C是D的子表，在表D中不需要列出A、B和C的元素。

(3) 广义表可以是递归的表，即广义表可以是本身的一个子表。
例如E就是一个递归的广义表。

任何一个非空广义表的表头可以是一个原子，也可以是一个广义表，而表尾一定是一个广义表。例如 $\text{head}(A) = ()$ ， $\text{head}(A) = ()$ ， $\text{head}(C) = a$ ， $\text{tail}(C) = ((b, c))$ ， $\text{head}(D) = A$ ， $\text{tail}(D) = (B, C)$ 其中， $\text{head}(A)$ 表示取广义表A的表头元素， $\text{tail}(A)$ 表示取广义表A的表尾元素。

注意 广义表 $()$ 和 $(())$ 不同，前者是空表，长度为0；后者长度为1，表示元素值为空表的广义表，可分解得到表头、表尾均为空表 $()$ 。

8.1.2 广义表的抽象数据类型

1. 数据对象集合

广义表的数据对象集合为 $\{a_i \mid 1 \leq i \leq n, a_i \text{ 可以是原子, 也可以是广义表}\}$ 。例如， $A = (a, (b, c))$ 是一个广义表，A中包含两个元素a和 (b, c) ，第2个元素为子表，包含了2个元素b和c。若把 (b, c) 看成一个整体，则a和 (b, c) 构成了一个线性表，在子表 (b, c) 的内部，b和c又构成了线性表。故广义表可看作是线性表的扩展。

2. 基本操作集合

(1) GetHead (L) : 求广义表的表头。如果广义表是空表, 则返回NULL; 否则返回指向表头结点的指针。

(2) GetTail (L) : 求广义表的表尾。如果广义表是空表, 则返回NULL; 否则返回指向表尾结点的指针。

(3) GListLength (L) : 返回广义表的长度。如果广义表是空表, 则返回0; 否则返回广义表的长度。

(4) CopyGList (&T, L) : 复制广义表。由广义表L复制得到广义表T。复制成功返回1, 否则返回0。

(5) GListDepth (L) : 求广义表的深度。广义表的深度就是广义表中括号嵌套的层数。如果广义表是空表, 则返回1, 否则返回广义表的深度。

8.2 广义表的头尾链表表示与实现

由于广义表中的数据元素具有不同的结构（或是原子，或是广义表），因此很难用顺序存储结构表示，通常采用链式存储结构表示广义表，本节就来介绍其中一种链式存储——广义表的头尾链表本节就来介绍其中一种链式存储——广义表的头尾链表。

8.2.1 广义表的头尾链表存储结构

因广义表中有原子和子表两种元素，所以广义表的链表结点也分为原子结点和子表结点两种，其中，子表结点包含标志域、指向表头的指针域和指向表尾的指针域3个域。原子结点包含标志域和值域两个域。表结点和原子结点的存储结构如图8-1所示。

其中，tag=1表示是子表，hp和tp分别指向表头结点和表尾结点，tag=0表示原子，atom用于存储原子的值。

广义表的这种存储结构称为头尾链表存储表示。例如用头尾链法表示的广义表A=（），B=（a），C=（a，（b，c）），D=（A，B，C），E=（a，E）如图8-2所示。

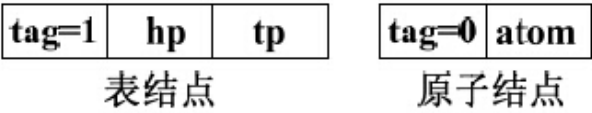


图8-1 表结点和原子结点的存储结构

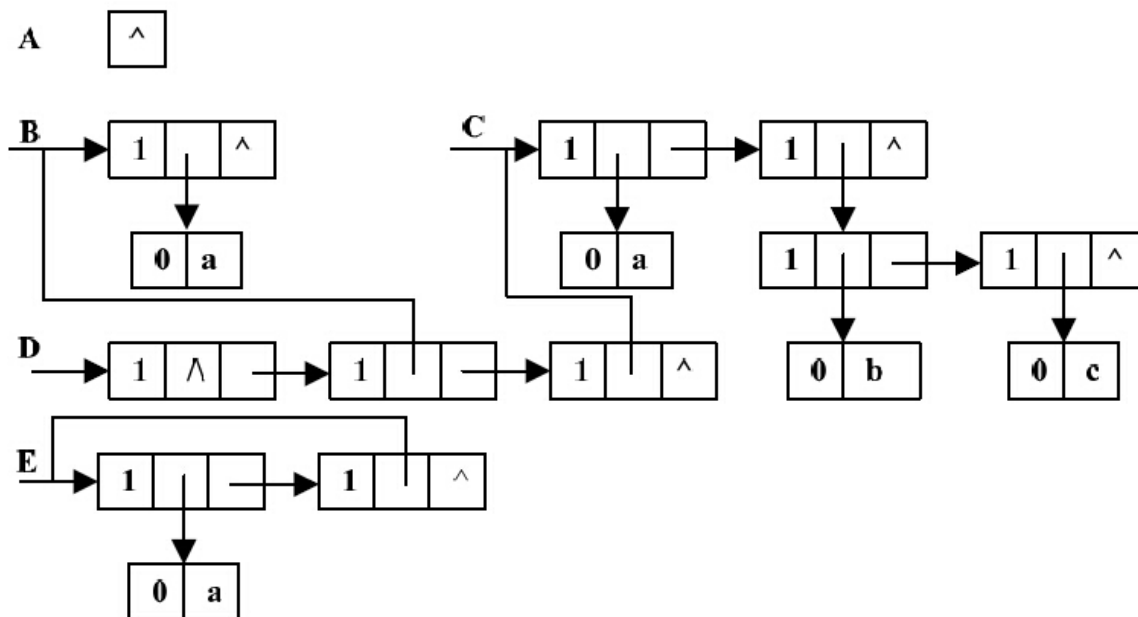


图8-2 广义表的存储结构

广义表的头尾链表存储结构类型描述如下。

```

typedef enum{ATOM,LIST}ElemTag;           /*ATOM=0
, 表示原子, LIST=1
, 表示子表*/
typedef struct
{
    ElemTag tag;                           /*
标志位tag
用于区分元素是原子还是子表*/
    union
    {
        AtomType atom;                    /*AtomType
是原子结点的值域, 用户自己定义类型*/
        struct
        {
            struct GLNode *hp,*tp; /*hp
指向表头, tp
指向表尾*/
        }ptr;
    };
}*GList,GLNode;

```

8.2.2 广义表的基本运算

采用头尾链表存储结构表示的广义表的基本运算实现如下所述（以下算法的实现保存在文件“GList.h”中）。

①求广义表的表头，代码如下。

```
GLNode* GetHead(GList L)
/*
求广义表的表头结点操作*/
{
    GLNode *p;
    if(!L) /*
如果广义表为空表，则返回NULL*/
    {
        printf("
该广义表是空表!");
        return NULL;
    }
    p=L->ptr.hp; /*
将广义表的表头指针赋值给p*/
    if(!p)
        printf("
该广义表的表头是空表");
    else if(p->tag==LIST)
        printf("
该广义表的表头是非空的子表。");
    else
        printf("
该广义表的表头是原子。");
    return p;
}
```

②求广义表的表尾，代码如下。

```
GLNode* GetTail(GList L)
/*
求广义表的表尾*/
{
    if(!L) /*
如果广义表为空表，则返回NULL*/
    {
        printf("
该广义表是空表!");
        return NULL;
    }
    return L->ptr.hp; /*
如果广义表不是空表，则返回指向表尾结点的指针*/
}
```

③求广义表的长度。求广义表的长度只需要沿着表尾指针tp查找下去，统计子表个数，直到tp为NULL为止。如果广义表是空表，则广义表的长度为

0。否则将指针L指向结点的表尾指针，统计广义表的长度。求广义表的长度的算法实现如下。

```
int GListLength(GList L)
/*
求广义表的长度*/
{
    int length=0;
    while(L) /*
如果广义表非空，则将p
指向表尾指针，统计表的长度*/
    {
        L=L->ptr.tp;
        length++;
    }
    return length;
}
```

④由广义表L复制得到广义表T。任何一个非空的广义表都可以分解为表头和表尾，反之，一对表头和表尾可以唯一确定一个广义表。由此，复制一个广义表只需要复制表头和表尾，然后合在一起就构成一个广义表。复制广义表的算法实现如下。

```
void CopyList(GList *T,GList L)
/*
复制广义表。由广义表L
复制得到广义表T*/
{
    if(!L) /*
如果广义表为空，则T
为空表*/
        *T=NULL;
    else
    {
        *T=(GList)malloc(sizeof(GLNode)); /*
表L
不空，为T
建立一个表结点*/
        if(*T==NULL)
            exit(-1);
        (*T)->tag=L->tag;
        if(L->tag==ATOM) /*
复制原子*/
            (*T)->atom=L->atom;
        else /*
递归复制子表*/
        {
            CopyList(&((*T)->ptr.hp),L->ptr.hp);
            CopyList(&((*T)->ptr.tp),L->ptr.tp);
        }
    }
}
```

⑤求广义表的深度。广义表的深度就是广义表中括弧的重数，设广义表为 $GL = (a_1, a_2, a_3, \dots, a_n)$ ， a_i 或为原子，或为子表，则求广义表GL的深度可分解为n个子问题，每个子问题就是求 a_i 的深度。若 a_i 是原子，则深度为0；若 a_i 是广义表，则按上述一样处理。广义表GL的深度为所有元素 a_i 的深度的最大值加1。空表也是广义表，并由定义可知空表的深度为1。

求广义表的深度的递归算法有两个终结状态：空表和原子。只要求出 a_i 的深度，广义表的深度就容易求出了，显然，它比子表的深度的最大值多1。

广义表 $GL = (a_1, a_2, a_3, \dots, a_n)$ 的深度GlistDepth (GL) 的递归定义如下。

GlistDepth (GL) =1, 当GL为空表时；

GlistDepth (GL) =0, 当GL为原子时；

GlistDepth (GL) =Max {GlistDepth (a_i) }+1, 当 $n \geq 1$ 时

根据以上定义可写出求广义表深度的算法。假设L是Glist类型的变量，则L=NULL表明广义表为空表，L->tag=0表明为原子，反之，L指向表结点，该结点的hp指向表头（即第1个子表），tp指向表尾（即hp指向的第2个子表）。

求广义表的深度的算法实现如下。

```

int GListDepth(GList L)
/*
求广义表的深度操作*/
{
    intmax,depth;
    GLNode *p;
    if(!L)
    如果广义表为空, 则返回1*/
        return 1;
    if(L->tag==ATOM)
    如果广义表是原子, 则返回0*/
        return 0;
    for(max=0,p=L;p=p->ptr.tp)
    逐层处理广义表*/
    {
        depth=GListDepth(p->ptr.hp);
        if(max<depth)
            max=depth;
    }
    return max+1;
}

```

广义表深度的递归算法的执行过程，其实就是访问广义表的每个结点，首先求得每个子表的深度，然后得到整个广义表的深度。例如，递归实现求广义表 $A = ((a), (), (a, (b, c)))$ 的深度过程如图8-3所示。

图8-3中的虚线表示递归的路线，旁边的数字表示返回当前子表的深度。可以看出，每当一个子表结束，就要对本层进行加1。例如，对于子表 (b, c) ，当返回到上一层时，因这是一个子表，所以返回1。

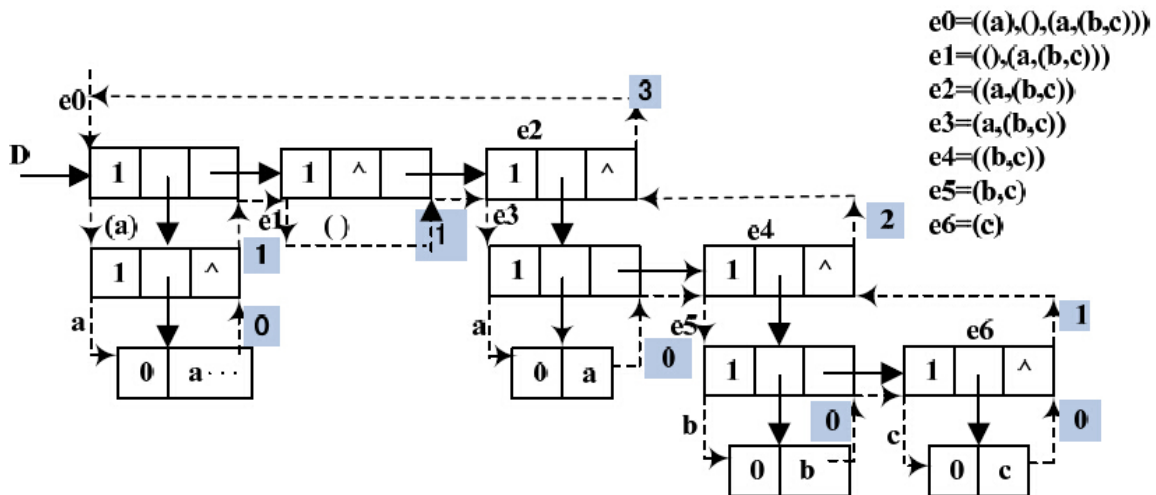


图8-3 递归求解广义表深度的过程

8.2.3 广义表应用举例（采用头尾链表存储结构）

【例8-1】 设广义表采用头尾链表存储结构，编写算法，建立一个广义表，并求出广义表的长度、深度和原子个数。

【分析】 主要考查对广义表的头尾链表存储结构的理解与基本操作。因为广义表是递归定义的，所以可以使用递归的方法创建广义表、求广义表的长度、深度和原子个数。

1. 创建广义表

创建广义表分为以下3步。

①分离出表头和表尾。根据输入的字符串，通过找到串的第一个逗号，逗号之前的元素为表头，逗号之后的元素为表尾。代码如下。

```
void DistributeString(SeqString *Str, SeqString *HeadStr)
/*
将串Str
分离成两个部分，HeadStr
为第一个逗号之前的子串，Str
为逗号后的子串*/
{
    int len, i, k;
    SeqString Ch, Ch1, Ch2, Ch3;
    len = StrLength(*Str);                               /*len
为Str
的长度*/
    StrAssign(&Ch1, ",");                                /*
将字符', '
, '('
和')'
分别赋给Ch1, Ch2
和Ch3*/
    StrAssign(&Ch2, "(");
    StrAssign(&Ch3, ")");
    SubString(&Ch, *Str, 1, 1);                           /*Ch
保存Str
的第一个字符*/
    for (i = 1, k = 0; i <= len && StrCompare(Ch, Ch1) || k != 0; i++) /*
去掉字符串中的", "
、" ("
和") "*/
    {
```



```

        SubString(&Ch,*Str,i,1);          /*
取出Str
的第一个字符*/
        if(!StrCompare(Ch,Ch2))          /*
如果第一个字符是'('
, 则令k
加1*/
            k++;
        else if(!StrCompare(Ch,Ch3))     /*
如果当前字符是')'
, 则令k
减去1*/
            k--;
    }
    if(i<=len)                            /*
串Str
中存在','
, 它是第i-1
个字符*/
    {
        SubString(HeadStr,*Str,1,i-2);  /*HeadStr
保存串Str','
前的字符*/
        SubString(Str,*Str,i,len-i+1);  /*Str
保存串Str','
后的字符*/
    }
    else                                  /*
串Str
中不存在','*/
    {
        StrCopy(HeadStr,*Str);          /*
将串Str
的内容复制到串HeadStr*/
        StrClear(Str);                  /*
清空串Str*/
    }
}

```

②将表头作为参数，通过递归创建表结点。

③若表尾不空，则将已经创建的表结点的表尾指针指向表尾结点。然后重新分离出表头和表尾，为新的表头创建结点，重复执行以上步骤，直到串为空为止。

创建广义表的算法实现如下。

```

void CreateList(GList *L,SeqString S)
/*
采用头尾链表创建广义表*/
{
    SeqString Sub,HeadSub,Empty;
    GList p,q;
    StrAssign(&Empty,"()");
    if(!StrCompare(S,Empty))          /*
如果输入的串是空串则创建一个空的广义表*/
        *L=NULL;
}

```

```

else
{
    if (!(*L=(GList)malloc(sizeof(GLNode)))) /*
为广义表生成一个结点*/
        exit(-1);
    if (StrLength(S)==1) /*
广义表是原子，则将原子的值赋值给广义表结点*/
    {
        (*L)->tag=ATOM;
        (*L)->atom=S.str[0];
    }
    else /*
如果是子表*/
    {
        (*L)->tag=LIST;
        p=*L;
        SubString(&Sub,S,2,StrLength(S)-2); /*
将s
去除最外层的括号，然后赋值给Sub*/
        do
        {
            DistributeString(&Sub,&HeadSub); /*
将Sub
分离出表头和表尾分别赋值给HeadSub
和Sub*/
            CreateList(&(p->ptr.hp),HeadSub); /*
递归调用生成广义表*/
            q=p;
            if (!StrEmpty(Sub)) /*
如果表尾不空，则生成结点p
，并将尾指针域指向p*/
            {
                if (! (p=(GLNode *)malloc(sizeof(GLNode))))
                    exit(-1);
                p->tag=LIST;
                q->ptr.tp=p;
            }
        }while(!StrEmpty(Sub));
        q->ptr.tp=NULL;
    }
}
}

```

2. 输出广义表

如果该元素是原子，则直接输出；否则先输出广义表的表头，然后输出广义表的表尾。这与求广义表的深度操作类似。输出广义表的程序代码如下。

```

void PrintGList(GList L)
/*
输出广义表的元素*/
{
    if (L) /*
如果广义表不空*/
    {
        if (L->tag==ATOM) /*
如果是原子，则输出*/

```

```

                                printf("%c ",L->atom);
                                else
                                {
                                    PrintGList (L->ptr.hp);          /*
递归访问L
的表头*/
                                    PrintGList (L->ptr.tp);          /*
递归访问L
的表尾*/
                                }
                                }
}

```

3. 测试函数

测试代码如下。

```

/*
头文件和广义表、串的基本操作的实现文件*/
#include <stdio.h>
#include<malloc.h>
#include<stdlib.h>
#include<string.h>
typedef char AtomType;
#include"GList.h"
#include"SeqString.h"
/*
函数声明*/
void CreateList(GList *L,SeqString S);
void DistributeString(SeqString *Str,SeqString *HeadStr);
void PrintGList(GList L);
int Sum3(GList L);
int GListAtomNum(GList L);
void StrPrint(SeqString S);
void main()
{
    GList L,T;
    SeqString S;
    int depth,length;
    /*
将字符串赋值给串S*/
    StrAssign(&S,"((a,b),(),(a,(b,c,d)),(a,e))");
    CreateList(&L,S);
    /*
由串创建广义表L*/
    printf("
广义表");
    StrPrint(S);
    printf("
中的元素:\n");
    PrintGList(L);
    /*
输出广义表中的元素*/
    length=GListLength(L);
    /*
求广义表的长度*/
    printf("\n
广义表L
的长度length=%2d\n",length);
    depth=GListDepth(L);
    /*
求广义表的深度*/
    printf("
广义表L
的深度depth=%2d\n",depth);
    printf("
广义表中的原子个数: %d\n",GListAtomNum(L));
    CopyList(&T,L);

```

```

        printf("
由广义表L
复制得到广义表T.\n
广义表T
的元素为:\n");
        PrintGList(T);
        length=GListLength(T);          /*
求广义表的长度*/
        printf("\n
广义表T
的长度length=%2d\n",length);
        depth=GListDepth(T);            /*
求广义表的深度*/
        printf("
广义表T
的深度depth=%2d\n",depth);
        StrAssign(&S,"((3,4),5,((6,3)))");
        CreateList(&L,S);
        printf("
广义表");
        StrPrint(S);
        GListAtomNum(L);
        printf("
中的原子个数: %d\n",GListAtomNum(L));
    }
    int GListAtomNum(GList L)
    {
        if(L==NULL)
            return 0;
        if(L->tag==0)
            return 1;
        if(L->tag==1)
            return GListAtomNum(L->ptr.hp)+GListAtomNum(L->ptr.tp);
    }

```

程序运行结果如图8-4所示。

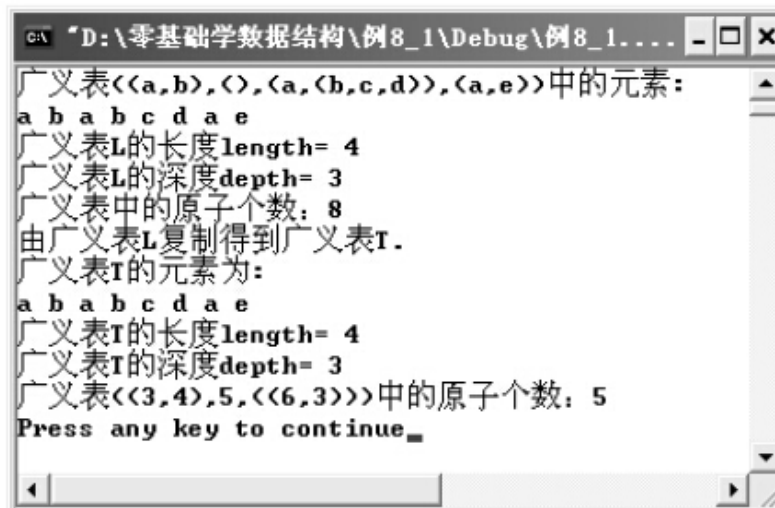


图8-4 广义表的基本操作程序运行结果

例如某一年计算机考研关于广义表的问题。广义表 $L = (a_1, a_2, \dots, a_n)$ ，其中 a_i ($i=1, 2, \dots, n$) 或是原子或是子表，编写一个函数计算一个广义表的所有原子结点数据域之和。

计算广义表的所有原子结点数据域之和的递归模型如下。

$$f(L) = \begin{cases} 0, & \text{若 } L = \text{NULL} \\ L \rightarrow \text{ptr}.\text{atom} + f(L \rightarrow \text{ptr}.\text{hp}), & \text{若 } L \rightarrow \text{tag} = 0 \\ f(L \rightarrow \text{ptr}.\text{tp}) + f(L \rightarrow \text{ptr}.\text{hp}), & \text{若 } L \rightarrow \text{tag} = 1 \end{cases}$$

算法实现如下。

```
int Sum(GList L)
{
    int m, n;
    if (L == NULL)
        return 0;
    else
    {
        if (L->tag == 0)
            n = L->atom - '0';
        else
            n = Sum(L->ptr.hp);
        if (L->ptr.tp != NULL)
            m = Sum(L->ptr.tp);
        else
            m = 0;
    }
    return m + n;
}
```

8.3 广义表的扩展线性链表表示与实现

广义表除了头尾链表存储结构外，还有一种叫做扩展线性链表的存储结构。

8.3.1 广义表的扩展线性链表存储结构

采用扩展线性链表表示的广义表也包含两种结点，分别为表结点和原子结点，这两种结点都包含3个域。其中，表结点由标志域tag、表头指针域hp和表尾指针域tp构成，原子结点由标志域、原子的值域和表尾指针域构成。

标志域tag用来区分当前结点是表结点还是原子结点，tag=0时为原子结点，tag=1时为表结点。hp和tp分别指向广义表的表头和表尾，atom用来存储原子结点的值。扩展性链表的结点结构如图8-5所示。



图8-5 扩展性链表结点存储结构

例如，A=（），B=（a），C=（a，（b，c）），D=（A，B，C），E=（a，E，），则广义表A、B、C、D、E的扩展性链表存储结构如图8-6所示。

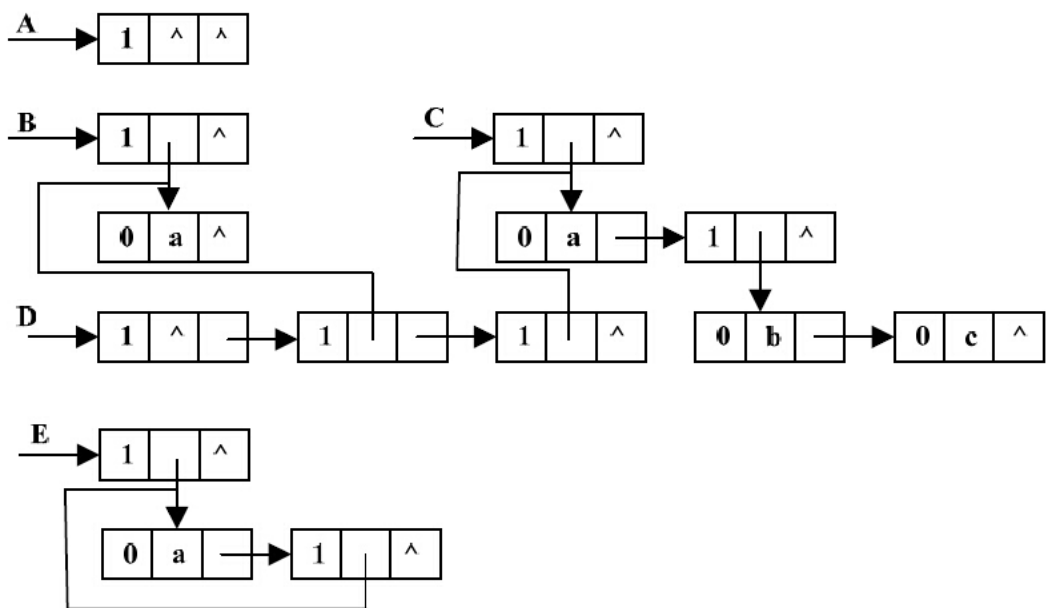


图8-6 广义表的扩展性链表表示

广义表的扩展线性链表存储结构的类型描述如下。

```

typedef enum{ATOM,LIST}ElemTag; /*ATOM=0
, 表示原子, LIST=1
, 表示子表*/
typedef struct
{
    ElemTag tag;                                /*
标志位tag
用于区分元素是原子还是子表*/
    union
    {
        AtomType atom;                        /*AtomType
是原子结点的值域, 用户自己定义类型*/
        struct GLNode *hp;                    /*hp
指向表头*/
    }ptr;
    struct GLNode *tp;                        /*tp
指向表尾*/
}*GList, GLNode;

```

8.3.2 广义表的基本运算

采用扩展性链表存储结构表示的广义表的基本运算实现如下（以下算法的实现保存在文件“GList2.h”中）。

①求广义表的表头，代码如下。

```

GLNode* GetHead(GList L)
/*
求广义表的表头结点*/
{
    GLNode *p;
    p=L->ptr.hp;
    将广义表的表头指针赋值给p*/
    if(!p)
    如果广义表为空表，则返回NULL*/
    {
        printf("
该广义表是空表！");
        return NULL;
    }
    else if(p->tag==LIST)
        printf("
该广义表的表头是非空的子表。");
    else
        printf("
该广义表的表头是原子。");
    return p;
}

```

②求广义表的表尾，代码如下。

```

GLNode* GetTail(GList L)
/*
求广义表的表尾*/
{
    GLNode *p,*tail;
    p=L->ptr.hp;
    if(!p)
    如果广义表为空表，则返回NULL*/
    {
        printf("
该广义表是空表！");
        return NULL;
    }
    tail=(GLNode*)malloc(sizeof(GLNode));
    生成tail
    结点*/
    tail->tag=LIST;
    将标志域置为LIST*/
    tail->ptr.hp=p->tp;
    将tail
    的表头指针域指向广义表的表尾*/
    tail->tp=NULL;
    将tail
    的表尾指针域置为空*/
    return tail;
    返回指向广义表表尾结点的指针*/
}

```

③求广义表的长度，代码如下。

```

int GListLength(GList L)
/*
求广义表的长度*/
{
    int length=0;
    初始化广义表的长度*/
    GLNode *p=L->ptr.hp;
    while(p)
    如果广义表非空，则将p
    指向表尾指针，统计表的长度*/

```

```

    {
        length++;
        p=p->tp;
    }
    return length;
}

```

④复制广义表，代码如下。

```

void CopyList (GList *T,GList L)
/*
复制广义表。由广义表L
复制得到广义表T*/
{
    if (!L)                                     /*
如果广义表为空，则T
为空表*/
        *T=NULL;
    else
    {
        *T=(GList)malloc(sizeof(GLNode));      /*
表L
不空，为T
建立一个表结点*/

        if (*T==NULL)
            exit (-1);
        (*T)->tag=L->tag;
        if (L->tag==ATOM)                       /*
复制原子*/
            (*T)->ptr.atom=L->ptr.atom;
        else
            CopyList (&(*T)->ptr.hp,L->ptr.hp); /*
递归复制表头*/

        if (L->tp==NULL)
            (*T)->tp=L->tp;
        else
            CopyList (&(*T)->tp,L->tp);          /*
递归复制表尾*/
    }
}

```

⑤求广义表的深度。如果广义表是空表，即L->tag==LIST&&L->ptr.hp==NULL，则返回1。如果是原子，即L->tag==ATOM，则返回0。

如果是一个非空的广义表，则递归调用GlistDepth函数求广义表的深度。先用头指针域找到下一层的子表，如果该层还有子表，则继续利用头指针域找到下一层，直到该层次结点为原子或者是空表，返回到上一层，并返回所求深度值。然后在该层中利用表尾指针找到该表的表尾，继续利用表头指针进行扫描，重复执行以上操作，直到所有层都返回。

求广义表的深度的算法实现如下。

```

int GListDepth(GList L)
/*
求广义表的深度操作*/
{
    int max,depth;
    GLNode *p;
    if (L->tag==LIST&&L->ptr.hp==NULL)          /*
如果广义表为空，则返回1*/
        return 1;
    if (L->tag==ATOM)                             /*
如果广义表是原子，则返回0*/
        return 0;
    p=L->ptr.hp;
    for (max=0;p;p=p->tp)                          /*
逐层处理广义表*/
    {
        depth=GListDepth(p);
        if (max<depth)
            max=depth;
    }
    return max+1;
}

```

8.3.3 广义表应用举例（扩展线性链表存储结构）

【例8-2】 编写一个算法，判断两个广义表是否相等（采用扩展线性链表存储结构）。两个广义表相等的含义是指两个广义表具有相同的结构，对应原子结点的数据域值也相等。

【分析】 判定两个广义表是否相等的递归模型如下。

根据以上递归模型，很容易可写出算法，具体如下。

$$\text{Equal}(P,Q) = \begin{cases} \text{true,} & \text{若 } P \rightarrow \text{tag}=0 \text{ 且 } Q \rightarrow \text{tag}=0 \text{ 且 } P \rightarrow \text{ptr.atom}=Q \rightarrow \text{ptr.atom} \\ \text{Equal}(P \rightarrow \text{ptr.hp}, Q \rightarrow \text{ptr.hp}), & \text{若 } P \rightarrow \text{tag}=1 \text{ 且 } Q \rightarrow \text{tag}=1 \\ \text{Equal}(P \rightarrow \text{tp}, Q \rightarrow \text{tp}), & \\ \text{false,} & \text{若 } P \rightarrow \text{tag}=0 \text{ 且 } Q \rightarrow \text{tag}=0 \text{ 且 } P \rightarrow \text{ptr.atom} \neq Q \rightarrow \text{ptr.atom} \\ \text{false,} & \text{若 } P \rightarrow \text{tag}=0 \text{ 且 } Q \rightarrow \text{tag}=1 \text{ 或 } P \rightarrow \text{tag}=1 \text{ 且 } Q \rightarrow \text{tag}=0 \\ \text{false,} & \text{若 } P=\text{NULL} \text{ 且 } Q \neq \text{NULL} \text{ 若 } P \neq \text{NULL} \text{ 且 } Q=\text{NULL} \end{cases}$$

1. 判定两个广义表是否相等

判定两个广义表是否相等的代码如下。

```

int EqualGList(GList P,GList Q)
{
    int flag=1;

```

```

        if (P!=NULL && Q!=NULL)
        {
            if (P->tag==0 && Q->tag!=0)
                flag=0;
            if (P->tag!=0 && Q->tag==0)
                flag=0;
            if (P->tag==0 && Q->tag==0)
            {
                if (P->ptr.atom!=Q->ptr.atom)
                    flag=0;
            }
            if (P->tag==1 && Q->tag==1)
                flag=EqualGList (P->ptr.hp,Q->ptr.hp);
            if (flag)
                flag=EqualGList (P->tp,Q->tp);
        }
        else
        {
            if (P==NULL && Q!=NULL)
                flag=0;
            if (P!=NULL && Q==NULL)
                flag=0;
        }
        return flag;
    }
}

```

2. 创建广义表

创建广义表分为以下3步。

①分离出表头和表尾。根据输入的字符串，通过找到串的第一个逗号，逗号之前的元素为表头，逗号之后的元素为表尾。

②将表头作为参数，通过递归创建表头结点。

③若表尾不空，则递归创建表尾结点。重复执行以上步骤，直到串为空为止。

创建广义表的算法实现如下。

```

void CreateList(GList *L, SeqString S)
/*
采用扩展线性链表创建广义表*/
{
    SeqString Sub, HeadSub, Empty;
    GList p;
    StrAssign (&Empty, " () ");
    if (!(*L=(GList)malloc(sizeof (GLNode)))) /*
为广义表生成一个结点*/
        exit (-1);
    if (!StrCompare (S, Empty)) /*
如果输入的串是空串则创建一个空的广义表*/
    {
        (*L)->tag=LIST;
        (*L)->ptr.hp=(*L)->tp=NULL;
    }
}

```

```

        else
        {
            if (StrLength(S) == 1) /*
广义表是原子，则将原子的值赋值给广义表结点*/
            {
                (*L) -> tag = ATOM;
                (*L) -> ptr.atom = S.str[0];
                (*L) -> tp = NULL;
            }
            else /*
如果是子表*/
            {
                (*L) -> tag = LIST;
                (*L) -> tp = NULL;
                SubString(&Sub, S, 2, StrLength(S) - 2); /*
将S
去除最外层的括号，然后赋值给Sub*/
                DistributeString(&Sub, &HeadSub); /*
将Sub
分离出表头和表尾分别赋值给HeadSub
和Sub*/
                CreateList(&((*L) -> ptr.hp), HeadSub); /*
递归调用生成广义表*/
                p = (*L) -> ptr.hp;
                while (!StrEmpty(Sub)) /*
如果表尾不空，则生成结点p
，并将尾指针域指向p*/
                {
                    DistributeString(&Sub, &HeadSub);
                    CreateList(&(p -> tp), HeadSub);
                    p = p -> tp;
                }
            }
        }
    }
}

```

3. 输出广义表

输出广义表的实现代码如下。

```

void PrintGLList(GList L)
/*
以广义表的形式输出*/
{
    if (L -> tag == LIST)
    {
        printf("("); /*
如果子表存在，先输出左括号 */
        if (L -> ptr.hp == NULL) /*
如果子表为空，则输出' '
字符 */
            printf(" ");
        else /*
递归输出表头*/
            PrintGLList(L -> ptr.hp);
        printf(")"); /*
在子表的最后输出右括号 */
    }
    else /*
如果是原子，则输出结点的值*/
        printf("%c", L -> ptr.atom);
    if (L -> tp != NULL)
    {
        printf(", "); /*
输出逗号*/
        PrintGLList(L -> tp); /*
递归输出表尾*/
    }
}

```

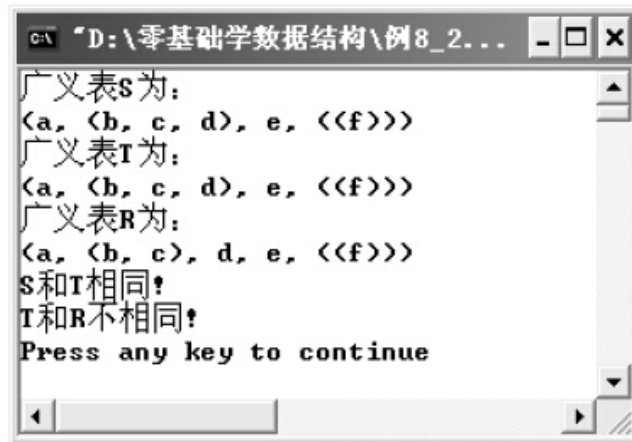
```
}  
}
```

4. 测试代码

测试代码如下。

```
#include<stdio.h>
#include<malloc.h>
typedef char AtomType;
#include"GList2.h"
#include"SeqString.h"
void CreateList(GList *L,SeqString S);
void DistributeString(SeqString *Str,SeqString *HeadStr);
void PrintGList(GList L);
int EqualGList(GList P,GList Q);
void main()
{
    GList S,T,R;
    int flag;
    SeqString S1,T1,R1;
    StrAssign(&S1,"(a,(b,c,d),e,((f)))");
    StrAssign(&T1,"(a,(b,c,d),e,((f)))");
    StrAssign(&R1,"(a,(b,c),d,e,((f)))");
    CreateList(&S,S1);
    CreateList(&T,T1);
    CreateList(&R,R1);
    printf("
广义表S
为: \n");
    PrintGList(S);
    printf("\n
广义表T
为: \n");
    PrintGList(T);
    printf("\n
广义表R
为: \n");
    PrintGList(R);
    flag=EqualGList(S,T);
    if(flag==1)
        printf("\nS
和T
相同!\n");
    else
        printf("\nS
和T
不相同!\n");
    flag=EqualGList(T,R);
    if(flag==1)
        printf("\nT
和R
相同!\n");
    else
        printf("T
和R
不相同!\n");
}
```

程序运行结果如图8-7所示。



```
C:\ "D:\零基础学数据结构\例8_2..."
广义表S为:
(a, (b, c, d), e, ((f)))
广义表T为:
(a, (b, c, d), e, ((f)))
广义表R为:
(a, (b, c), d, e, ((f)))
S和T相同!
T和R不相同!
Press any key to continue
```

图8-7 求广义表的长度和深度程序运行结果

8.4 小结

与数组一样，广义表也是一种扩展的线性表。广义表是由 n 个相同数据类型的数据元素 $(a_0, a_1, a_2, \dots, a_{n-1})$ 组成的有限序列。

由于广义表中的数据元素既可以是原子，也可以是广义表。广义表的链式存储结构有两种：广义表的头尾链表存储表示和广义表的扩展线性链表存储表示。

广义表的基本运算包括广义表的创建、求广义表的表头和表尾操作、广义表的深度和长度及复制广义表。广义表的深度指的是括号的嵌套层数。广义表的长度指的是最外层元素的个数。例如，广义表 $D = ((a, b), (), (a, (b, c, d)), (a, e))$ 的长度为4，深度为3。

8.5 习题

一、选择题

1. 设广义表 $L = ((a, b, c))$ ，则 L 的长度和深度分别为（）。

A. 1和1

B. 1和3

C. 1和2

D. 2和3

2. 广义表 $((a), a)$ 的表尾是（）。

A. a

B. (a)

C. $()$

D. $((a))$

3. 一个非空广义表的表头（）。

A. 不可能是子表

B. 只能是子表

C. 只能是原子

D. 可以是子表或原子

4. 广义表 $G = (a, b(c, d, (e, f)), g)$ 的长度是 ()。

A. 3

B. 4

C. 7

D. 8

5. 广义表 (a, b, c) 的表尾是 ()。

A. b, c

B. (b, c)

C. c

D. (c)

6. 广义表A= ((a) , a) 的表头是 () 。

A. a

B. (a)

C. b

D. ((a))

二、算法设计题

1. 编写算法，要求计算一个广义表的原子结点个数。
2. 假设广义表以头尾链表方式存储，请写出以广义表形式输出的算法。
3. 请写出求广义表长度的非递归算法。

第三篇 非线性数据结构

第9章 树

第3~8章介绍的线性表、栈、队列、串、数组和广义表都属于线性结构，本章的树和第10章的图都属于非线性数据结构。线性数据结构中的元素之间是一一对应的关系，而本章要介绍的树中的元素之间是一对多的关系。树这种结构在实际应用中也非常广泛，它主要应用在文件系统、目录组织等数据处理中。本章主要介绍树的定义、二叉树的定义与性质、二叉树的表示与实现、二叉树的遍历、二叉树的线索化、树与森林的转换及哈夫曼树。

本章重点和难点：

- 树与二叉树的性质
- 二叉树的各种递归与非递归遍历算法
- 二叉树的线索化
- 树、二叉树、森林的相互转化

- 哈夫曼树与哈夫曼编码

9.1 树的相关概念及抽象数据类型

树是一种非线性的数据结构，树中元素之间的关系是一对多的层次关系。本节主要介绍树的定义、抽象数据类型及存储结构。

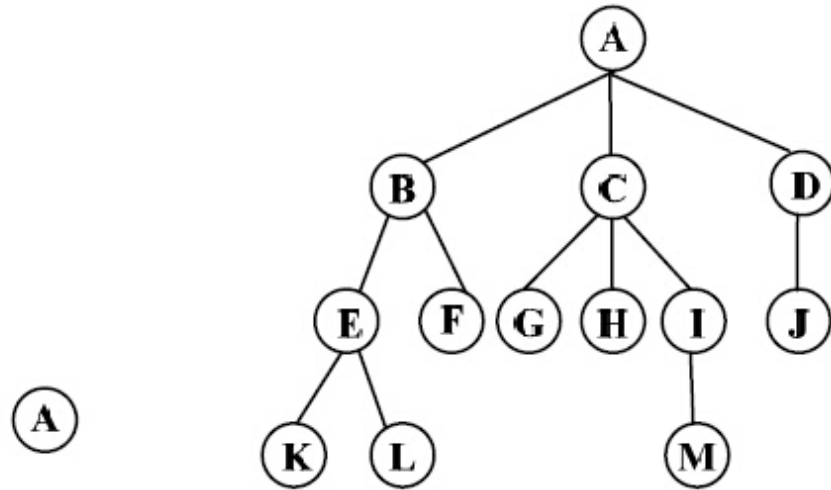
9.1.1 什么是树

树（tree）是 n （ $n \geq 0$ ）个结点的有限集合，一棵非空树中具有以下特点。

（1）有且只有一个称为根（root）的结点。

（2）当 $n > 1$ 时，其余 $n-1$ 个结点可以划分为 m 个有限集合 T_1, T_2, \dots, T_m ，且这 m 个有限集合不相交，其中 T_i （ $1 \leq i \leq m$ ）又是一棵树，称为根的子树（subtree）。

当 $n=0$ 时，称为空树；当 $n>0$ 时，称为非空树。树的逻辑结构如图9-1所示。



(a) 只有根结点的树

(b) 一般的树

图9-1 树的逻辑结构

图9-1 (a) 是一棵只有根结点的树，图9-1 (b) 是一棵有14个结点的树，其中，A是根结点，其余结点分成3个互不相交的子集 $T_1 = \{B, E, F, K, L\}$ 、 $T_2 = \{C, G, H, I, M\}$ 和 $T_3 = \{D, J\}$ 。其中， T_1 、 T_2 和 T_3 分别是一棵树，它们都是根结点A的子树。 T_1 的根结点是B，其余的4个结点又分为2个不相交的子集 $T_{11} = \{E, K, L\}$ 和 $T_{12} = \{F\}$ 。其中， T_{11} 和 T_{12} 都是 T_1 的子树，E是 T_{11} 的根结点， $\{K, L\}$ 是E的子树。

图9-1 (b) 所示的树看上去像一棵颠倒过来的树，根结点就像是树根，子树像一棵树的枝杈。一棵树中只有一个根结点。树的最末端的结点称为叶子结点，即K、L、F、G、H、M和J都是叶子结点，类似树的叶子，这些结点没有子树。

一棵树的根与子树是一对多的关系，例如，结点C有3棵子树 $T_{21} = \{G, M\}$ 、 $T_{22} = \{H\}$ 和 $T_{23} = \{I, N\}$ ，而 T_{21} 、 T_{22} 和 T_{23} 只有一个根结点C。

9.1.2 树的相关概念

在介绍树的算法之前，先介绍有关树的一些概念。

树的结点：包含一个数据元素及指向其他结点的分支信息。

结点的度（degree）：结点的子树的个数称为结点的度。例如，结点B有2棵子树，因此度为2。

树的度：树中各结点的度的最大值。例如，图9-1（b）中的树的度为3，因为结点A和C的度都为3，它们是树中拥有最大度的结点。

叶子结点：也称为终端结点，度为零的结点即没有子树的结点称为叶子结点。例如，结点K、L、F、G、H、M和J都是叶子结点。

非终端结点：度不为零的结点也称为分支结点。例如，A、B、C、D、E、I等都是非终端结点。

孩子（child）与**双亲**（parent）：结点的子树的根称为孩子，相应地，该结点称为双亲。例如， $\{G, M\}$ 是根结点C的子树，而C又是这棵子树的根结点，因此，G是C的孩子。而C是G的双亲。

兄弟（sibling）：同一个双亲的孩子之间互称为兄弟。例如，E和F是B的孩子，故E和F互为兄弟，同理，G、H和I互为兄弟。

祖先与子孙：从根结点到达一个结点所经分支上的所有结点称为该结点的祖先。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。例如，I、C和A都为M的祖先，{E，F，K，L}是B的子树，故E、F、K和L都是B的子孙。

树的层次：从根结点起，根结点为第1层，根结点的孩子结点为第2层，依此类推，如果某一个结点是第L层，则其孩子结点位于第L+1层。图9-1（b）所示的树的层次为4。

树的深度（depth）：树中所有结点的层次最大值称为树的深度，也称为树的高度。例如，图9-1（b）中树的深度为4。

有序树：如果树中各棵子树之间是有先后次序的，则称该树为有序树。

无序树：如果树中各棵子树之间没有先后次序，则称该树为无序树。

森林（forest）：m棵互不相交的树构成一个森林。若把一棵非空的树的根结点删除，则该树就变成了一个森林，森林中的树由原来

的根结点的各棵子树构成。反之，把一个森林加上一个根结点，则该森林就变成一棵树。

9.1.3 树的逻辑表示

树的逻辑表示方法可以分为树形表示法、文氏图表示法、广义表表示法和凹入表示法4种。

树形表示法如图9-1所示。树形表示法是最常见的表示法，它能直观、形象地表示出树的逻辑结构和结点之间的关系。

文氏图是一种集合表示法，对于其中任意两个集合，或者不相交，或者一个包含另一个。文氏图表示法如图9-2所示。

根作为由子树森林组成的表的名字写在表的左边，图9-1（b）所示的树可用广义表表示如下。

(A (B (E (K, L) , F) , C (G (M) , H, I (N)) ,
D (J))))

凹入表示法与一本书的章节目录类似，章、节、小节逐个凹入。图9-1（b）所示的树采用凹入表示法如图9-3所示。

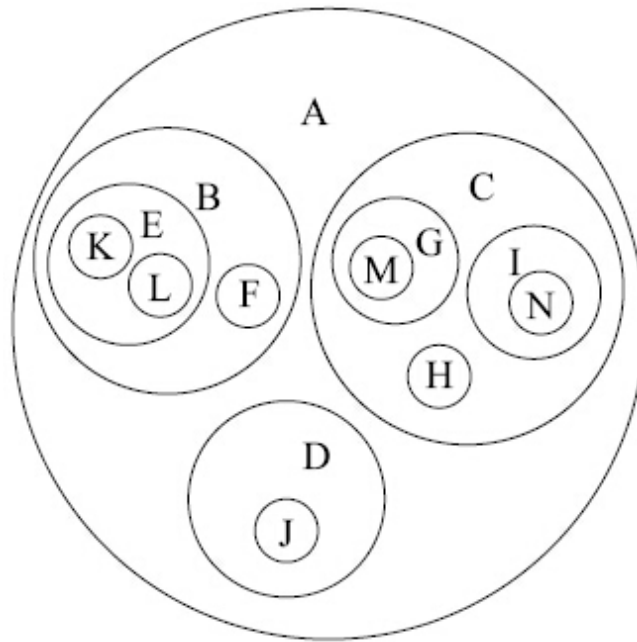


图9-2 树的文氏图表示法



图9-3 树的凹入法表示法

这4种表示树的形式中，一般比较常见的是树形表示法和广义表表示法。

9.1.4 树的抽象数据类型

1. 数据对象集合

树的数据对象集合为树的各个结点的集合。一个结点只有一个双亲结点，但可能有多个孩子结点。因此，树中元素结点之间是一对多的关系。例如，在树（A（B（E（K，L），F），C（G，H，I（M）），D（J）））中，G、H和I是结点C的孩子结点，而G、H和I只有一个双亲结点C，结点C只有一个双亲结点A。

2. 基本操作集合

(1) InitTree (&T)

操作结果：将T初始化为一棵空树。

(2) CreateTree (&T)

- 初始条件：树T不存在。
- 操作结果：创建树T。

(3) DestroyTree (&T)

- 初始条件：树T已存在。
- 操作结果：销毁树T。

(4) TreeEmpty (T)

- 初始条件：树T已存在。
- 操作结果：如果T是空树，则返回1；否则返回0。

(5) Root (T)

- 初始条件：树T已存在。
- 操作结果：如果树T非空，则返回树的根结点，否则返回NULL。

(6) Parent (T, e)

- 初始条件：树T已存在，e是T中的某个结点。
- 操作结果：如果e不是根结点，则返回该结点的双亲，否则返回NULL。

(7) FirstChild (T, e)

- 初始条件：树T已存在，e是T中的某个结点。

- 操作结果：如果e不是叶子结点，则返回该结点的第一个孩子结点，否则，返回NULL。

(8) NextSibling (T, e)

- 初始条件：树T已存在，e是树T中的某个结点。
- 操作结果：如果e不是其双亲结点的最后一个孩子结点，则返回它的下一个兄弟结点，否则返回NULL。

(9) InsertChild (&T, p, Child)

- 初始条件：树T已存在，p指向T中的某个结点。
- 操作结果：在树T中，指针p指向T中的某个结点，将非空树child插入T中，使child成为p指向的结点的子树。

(10) DeleteChild (&T, p, i)

- 初始条件：树T已存在，p指向T中的某个结点。
- 操作结果：在树T中，指针p指向T的某个结点，将p所指向的结点的第i棵子树删除。如果删除成功，返回1，否则返回0。

(11) TraverseTree (T)

- 初始条件：树T已存在。

- 操作结果：按照某种次序对树中的每个结点进行访问（如输出），并且每个结点访问且仅访问一次，即树的遍历。

(12) TreeDepth (T)

- 初始条件：树T已存在。
- 操作结果：如果树非空，返回树的深度，如果是空树，返回0。
树的深度即树的结点层次的最大值。

9.1.5 树的存储结构

通常情况下，树的存储结构有双亲表示法、孩子表示法和孩子兄弟表示法3种。

1. 双亲表示法

双亲表示法是利用一组连续的存储单元存储树的每个结点，并利用一个指示器表示结点的双亲结点在树中的相对位置。结点结构如图9-4所示。

其中，data存放数据元素信息，parent存放该结点的双亲在数组中的下标。

在C语言中，通常采用数组存储树中的结点，这类似于静态链表的实现。一棵树结构及树的双亲表示法如图9-5所示。

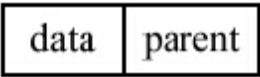


图9-4 双亲表示法的结点结构

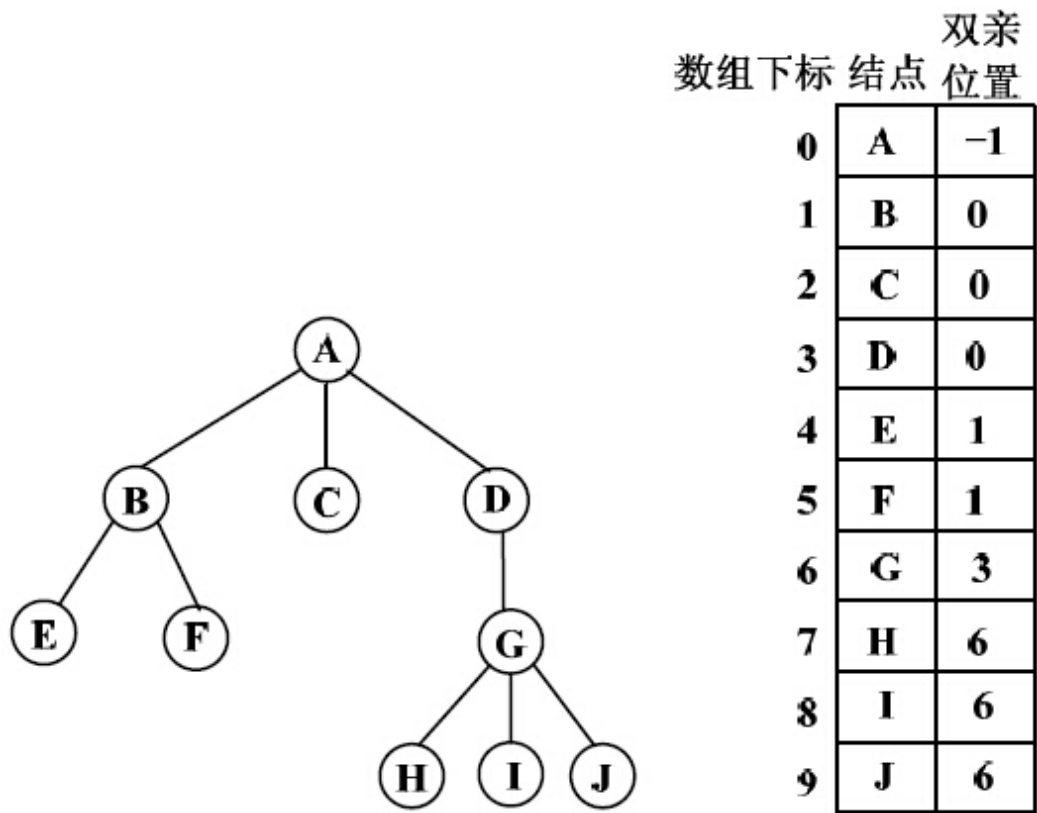


图9-5 树的双亲表示法

其中，树的根结点的双亲位置用-1表示。

在采用双亲表示法存储树结构时，根据给定结点查找其双亲结点非常容易，我们可通过反复调用求双亲结点，很快能找到树的根结点。采用双亲表示法的树类型定义如下。

```
#define MaxSize 200
typedef struct Pnode          /*
双亲表示法的结点定义*/
{
    DataType data;
    int parent;                /*
指示结点的双亲*/
}PNode;
typedef struct                /*
双亲表示法的类型定义*/
{
    PNode node[MaxSize];
    int num;                   /*
结点的个数*/
}PTree;
```

2. 孩子表示法

孩子表示法是将双亲结点的孩子结点构成一个链表，然后让双亲结点指向这个链表，这样的链表称为孩子链表。若树中有n个结点，就有n个孩子链表。n个结点的数据及头指针构成一个顺序表。图9-5所示的树的孩子表示法如图9-6所示，其中，^表示空。

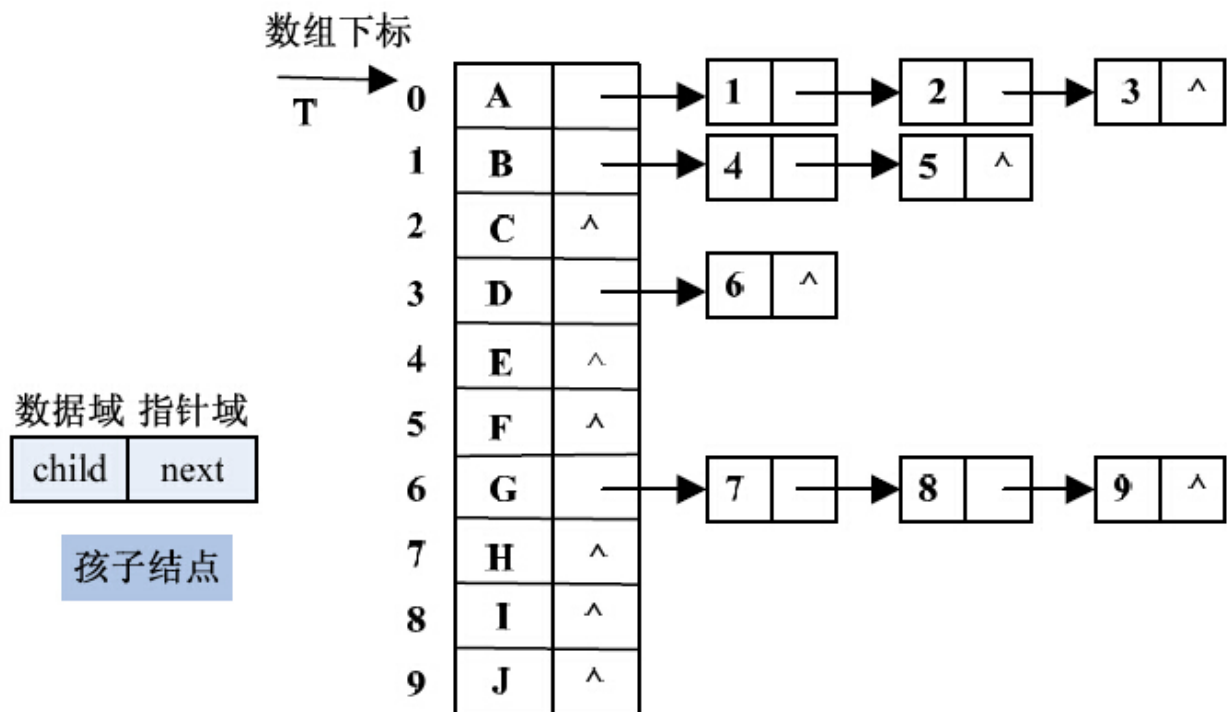


图9-6 树的孩子表示法



图9-7 表头结点

为此，需要设计两个结点结构，一个是孩子链表的孩子结点，如图9-6所示，child是数据域，存放结点在表头数组中的下标，next是指针域，存放指向下一个孩子结点的指针；另一个是表头数组的表头结点，如图9-7所示，data存储结点的数据信息，firstchild存储孩子链表的头指针。

树的孩子表示法的类型定义如下。

```

#define MaxSize 200
typedef struct CNode
孩子结点*/
{
    int child;
    struct CNode*next;
指向下一个结点*/
}ChildNode;
typedef struct
表头结构*/
{
    DataType data;
    ChildNode *firstchild;
孩子链表的指针*/
}DataNode;
typedef struct
孩子表示法类型定义*/
{
    DataNode node[MaxSize];
    int num,root;
结点的个数，根结点在顺序表中的位置*/
}CTree;

```

树的孩子表示法使得已知一个结点查找其孩子结点变得非常容易。通过表头结点指向的链表，找到该结点的每个孩子结点。但是查找双亲结点并不方便，这可将双亲表示法与孩子表示法结合起来，即在结点的顺序表中增加一个表示双亲结点位置的域，这样无论查找双亲结点还是孩子结点都非常方便，图9-8就是将两者结合起来的带双亲的孩子链表。

3. 孩子兄弟表示法

孩子兄弟表示法也称为树的二叉链表表示法。孩子兄弟表示法采用链表作为存储结构，结点包含一个数据域和两个指针域。数据域存放结点的数据信息，一个指针域用来指示结点的第一个孩子结点，另一个指针域用来指示结点的下一个兄弟结点。

图9-5所示的树对应的孩子兄弟表示及结点结构如图9-9所示。

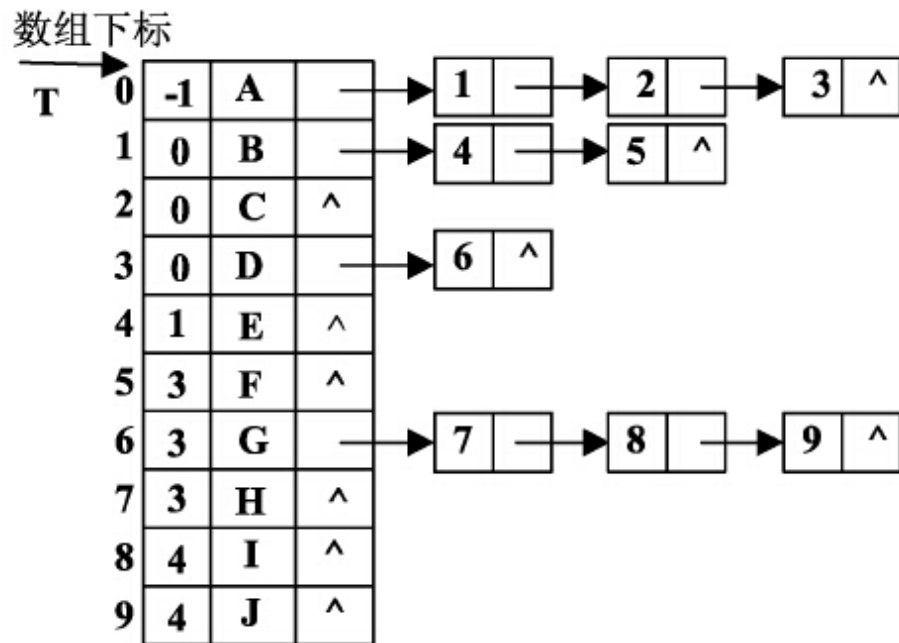


图9-8 带双亲的孩子链表

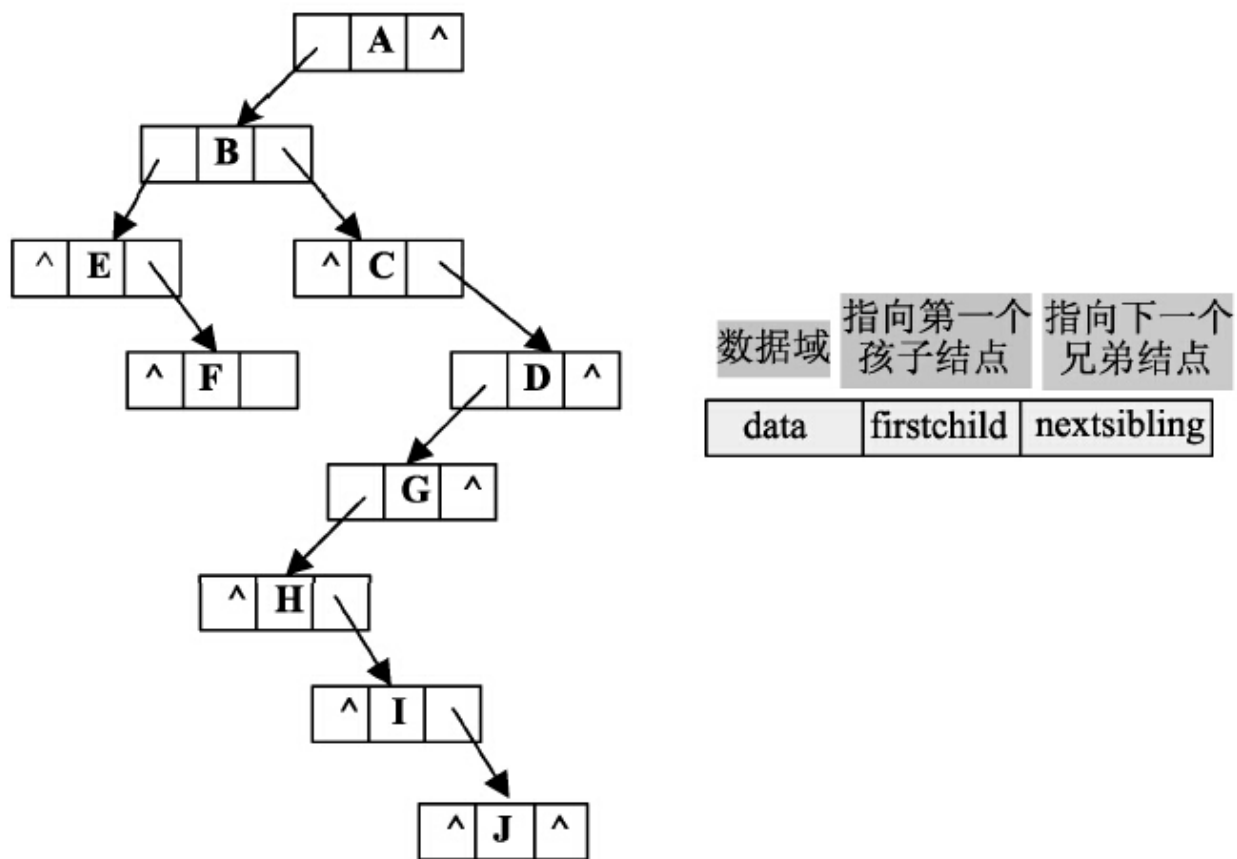


图9-9 树的孩子兄弟表示法

树的孩子兄弟表示法的类型定义如下。

```

typedef struct CSNode
孩子兄弟表示法的类型*/
{
    DataType data;
    struct CSNode*firstchild,*nextsibling;
指向第一个孩子和下一个兄弟*/
}CSNode,*CSTree;

```

其中，指针firstchild指向结点的第一个孩子结点，nextsibling指向结点的下一个兄弟结点。

孩子兄弟表示法是树的最常用的存储结构，利用树的孩子兄弟表示法可以实现树的各种操作。例如，要查找树中D的第3个孩子结点，则只需要从D的firstchild找到第一个孩子结点，然后顺着结点的nextsibling域走两步，就可以找到D的第3个孩子结点H。

9.2 二叉树的相关概念及抽象数据类型

在探讨一般的树之前，先研究一种比较特殊的树——二叉树。二叉树具有树的一般特性，与一般的树相比，它的结构简单，更有利于读者对树这个抽象概念的掌握。本节主要介绍二叉树的定义、基本性质及二叉树的抽象数据类型。

9.2.1 什么是二叉树

二叉树 (binary tree) 是由 n ($n \geq 0$) 个结点构成的另一种树结构。它的特点是每个结点最多只有两棵子树（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（称为左孩子和右孩子），次序不能颠倒。若 $n=0$ ，则称该二叉树为空二叉树。

在二叉树中，任何一个结点的度只可能是0、1和2。

二叉树有5种基本形态，如图9-10所示。

在如图9-11所示的二叉树中，D是B的左孩子结点，E是B的右孩子结点，H是E的左孩子结点，D既没有左孩子结点也没有右孩子结点。

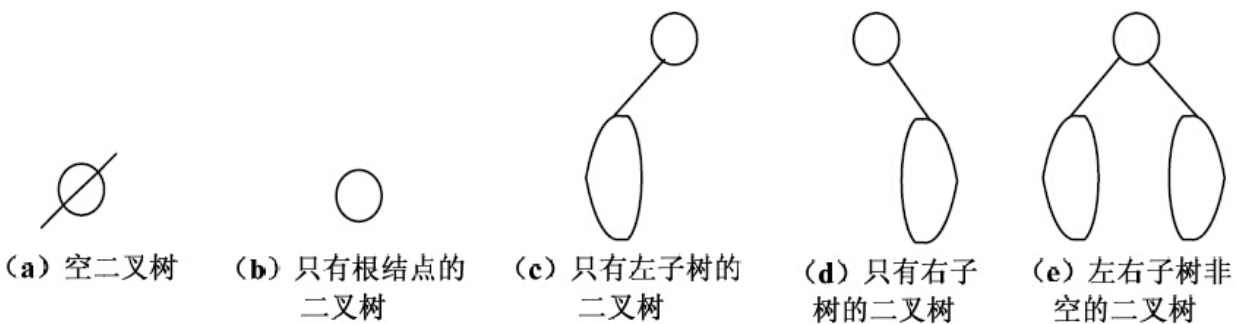


图9-10 二叉树的5种基本形态

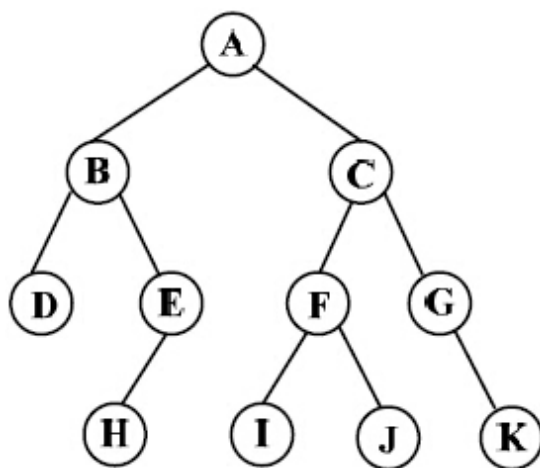


图9-11 二叉树

9.2.2 二叉树的性质

二叉树具有以下重要的性质。

性质1 二叉树的第 k ($k \geq 1$) 层上至多有 2^{k-1} 个结点
($k \geq 1$) 。

证明： 利用数学归纳法证明此性质。

(1) 当 $k=1$ 时，只有一个根结点，显然有 $2^{k-1} = 2^{1-1} = 2^0 = 1$ ，命题成立。

(2) 假设对于所有的 j ($1 \leq j < k$) 命题成立，即第 j 层上至多有 2^{j-1} 个结点，那么，可以证明 $j=k$ 时命题也成立。由归纳假设，第 $k-1$ 层上至多有 2^{k-2} 个结点。由于二叉树中的每个结点的度至多为2，则在第 k 层上的最大结点数为第 $k-1$ 层上的最大结点数的2倍，即 $2 \times 2^{k-2} = 2^{k-2+1} = 2^{k-1}$ 。

性质2 深度为 k ($k \geq 1$) 的二叉树至多有 $2^k - 1$ 个结点。

证明： 由性质1可知，第 i 层结点的最多个数 2^{i-1} ，将深度为 k 的二叉树中的每一层的结点的最大值相加，就得到二叉树中结点的最大值，因此深度为 k 的二叉树的结点总数至多为

$$\sum_{i=1}^k (\text{第 } i \text{ 层的结点最大个数}) = \sum_{i=1}^k 2^{i-1} = 2^0 + 2^1 + \dots + 2^{k-1} = \frac{2^0(2^k - 1)}{2 - 1} = 2^k - 1。$$

性质3 对于任何一棵二叉树 T ，如果终端结点数为 n_0 ，度为2的结点数为 n_2 ，则有 $n_0 = n_2 + 1$ 。

证明： 假设二叉树的结点数为 n ，度为1的结点数为 n_1 ，则 n 等于度为0、度为1和度为2的结点总数的和，即 $n = n_0 + n_1 + n_2$ 。

再看二叉树的分支数，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 $B=n_1+2n_2$ ，于是得到 $n=B+1=n_1+2n_2+1$ 。

联合上述两式 $n=n_0+n_1+n_2$ 和 $n=n_1+2n_2+1$ ，可得到 $n_0+n_1+n_2=n_1+2n_2+1$ ，即 $n_0=n_2+1$ 。命题得证。

1. 满二叉树

满二叉树和完全二叉树是两种特殊的二叉树。每层结点都是满的二叉树称为**满二叉树**，即在满二叉树中，每一层的结点都具有最大的结点个数。图9-12所示就是一棵满二叉树。在满二叉树中，每个结点的度或者为2，或者为0（即叶子结点），不存在度为1的结点。

2. 完全二叉树

对满二叉树的结点进行连续编号，约定从根结点开始，自上到下，自左到右，可以得到如图9-13所示的带编号的满二叉树。

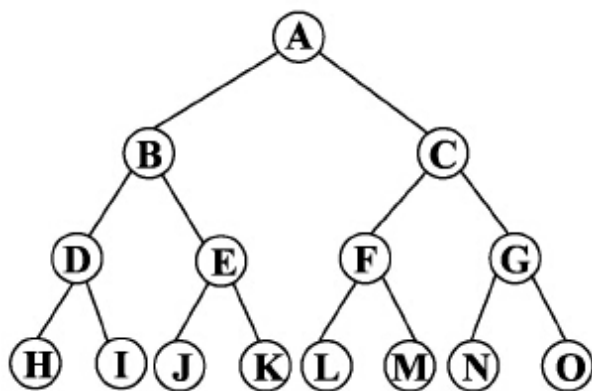


图9-12 满二叉树

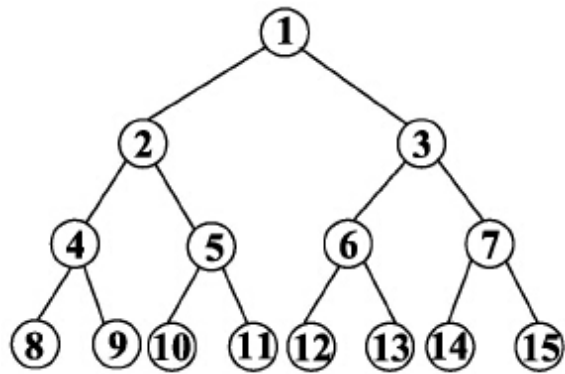


图9-13 带编号的满二叉树

在一棵具有 m ($0 \leq m \leq n$) 个结点的二叉树中，若每个结点都与满二叉树的编号从1到 m 一一对应时，称为**完全二叉树**。一棵完全二叉树及编号如图9-14所示，而图9-15所示的树不是一棵完全二叉树。

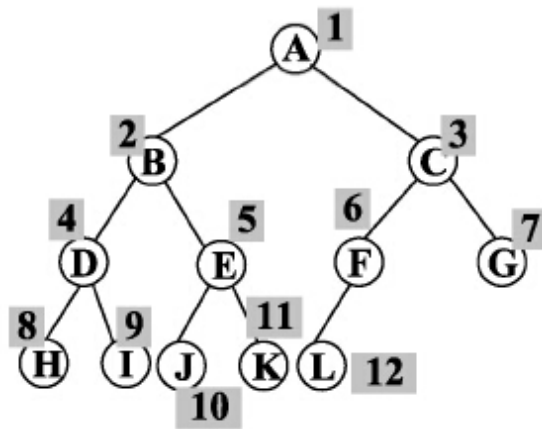


图9-14 完全二叉树及编号

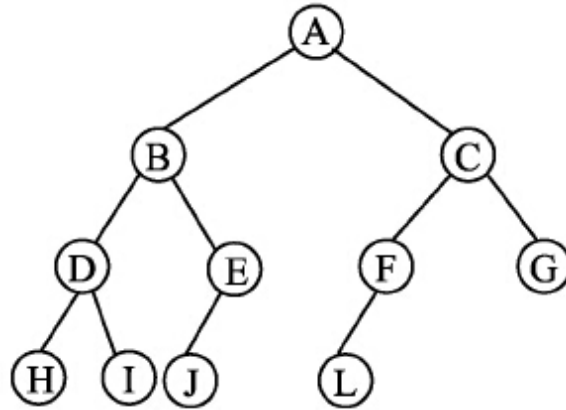


图9-15 非完全二叉树

由此可以得出结论，如果二叉树的层数为 k ，则满二叉树的叶子结点一定是在第 k 层，而完全二叉树的叶子结点一定在第 k 层或者第 $k-1$ 层出现。满二叉树一定是完全二叉树，而完全二叉树却不一定是满二叉树。

性质4 具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ ($\lfloor x \rfloor$ 表示不大于 x 的最大整数)。

证明： 假设完全二叉树的深度为 k 。根据性质2， $k-1$ 层满二叉树的结点总数为 $n_1 = 2^{k-1} - 1$ ， k 层满二叉树的结点总数为 $n_2 = 2^k - 1$ 。因此有 $n_1 < n \leq n_2$ ，即 $n_1 + 1 \leq n < n_2 + 1$ ，又 $n_1 = 2^{k-1} - 1$ ， $n_2 = 2^k - 1$ ，故得到 $2^{k-1} - 1 \leq n < 2^k - 1$ ，同时对不等式两边取对数，有 $k-1 \leq \log_2 n < k$ 。因为 k 是整数， $k-1$ 也是整数，所以 $k-1 = \lfloor \log_2 n \rfloor$ ，即 $k = \lfloor \log_2 n \rfloor + 1$ 。命题得证。

性质5 如果对于 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，每层从左至右），则对任一结点 i （ $1 \leq i \leq n$ ）有以下性质。

（1）如果 $i=1$ ，则结点 i 是二叉树的根，无双亲；若 $i>1$ ，则其双亲结点为 $\lfloor i/2 \rfloor$ 。

（2）如果 $2i>n$ ，则结点 i 无左孩子。否则，其左孩子是 $2i$ 。

（3）如果 $2i+1>n$ ，则结点 i 无右孩子。否则，其右孩子是 $2i+1$ 。

证明：（1）只要先证明了性质（2）和性质（3），便可由（2）和（3）得到性质（1）。

当 $i=1$ 时，该结点一定是根结点，根结点没有双亲结点。当 $i>1$ 时，分两种情况讨论。

- 设编号为 m 的结点是编号为 i 的双亲结点。如果编号为 i 的结点是序号为 m 的结点的左孩子结点，则根据性质（2）有 $2m=i$ ，即 $m=i/2$ 。

- 如果编号为 i 的结点是编号为 m 结点的右孩子结点，则根据性质（3）有 $2m+1=i$ ，即 $m=(i-1)/2=i/2-1/2$ 。综上以上两种情况，当 $i>1$ 时，编号为 i 的结点的双亲结点编号为 $\lfloor i/2 \rfloor$ 。结论得证。

(2) 利用数学归纳法。当 $i=1$ 时, 有 $2i=2$, 如果 $2>n$, 则二叉树中不存在编号为2的结点, 也就不存在编号为 i 的左孩子结点。如果 $2\leq n$, 则该二叉树中存在两个结点, 编号2是编号为 i 的结点的左孩子结点的编号。

假设编号 $i=k$ 时, 当 $2k\leq n$ 时, 编号为 k 的结点的左孩子结点存在且编号为 $2k$, 当 $2k>n$ 时, 编号为 k 的结点的左孩子结点不存在。

当 $i=k+1$ 时, 在完全二叉树中, 如果编号为 $k+1$ 的结点的左孩子结点存在 ($2i\leq n$), 则其左孩子结点的编号为编号为 k 的结点的右孩子结点编号加1, 即编号为 $k+1$ 的结点的左孩子结点编号为

$(2k+1)+1=2(k+1)=2i$ 。因此, 当 $2i>n$ 时, 编号为 i 的结点的左孩子不存在。结论得证。

(3) 利用数学归纳法。当 $i=1$ 时, 如果 $2i+1=3>n$, 则该二叉树中不存在序号为3的结点, 即编号为 i 的结点的右孩子不存在。如果 $2i+1=3\leq n$, 则该二叉树存在编号为3的结点, 且序号为3的结点是编号 i 的结点的右孩子结点。

假设编号 $i=k$ 时, 当 $2k+1\leq n$ 时, 编号为 k 的结点的右孩子结点存在且编号为 $2k+1$, 当 $2k+1>n$ 时, 序号为 k 的结点的右孩子结点不存在。

当 $i=k+1$ 时, 在完全二叉树中, 如果编号为 $k+1$ 的结点的右孩子结点存在 ($2i+1\leq n$), 则其右孩子结点的序号为编号为 k 的结点的右孩

子结点序号加2，即编号为 $k+1$ 的结点的右孩子结点编号为

$(2k+1) + 2 = 2(k+1) + 1 = 2i+1$ 。因此，当 $2i+1 > n$ 时，编号为 i 的结点的右孩子不存在。结论得证。

9.2.3 二叉树的抽象数据类型

1. 数据对象集合

二叉树的数据对象集合为二叉树中的各个结点构成的集合。根结点没有双亲结点，其他结点只有一个双亲结点。每个结点的孩子可能是0个、1个和2个。

2. 基本操作集合

(1) InitBitTree (&T)

- 操作结果：构造空二叉树T。

(2) CreateBitTree (&T)

- 初始条件：二叉树T不存在。
- 操作结果：创建二叉树T。

(3) DestroyBitTree (&T)

- 初始条件：二叉树T已存在。
- 操作结果：如果二叉树存在，则将该二叉树销毁。

(4) InsertChild (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，c非空，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p所指结点的左子树或右子树，p所指结点的原有左子树或右子树成为c的右子树。插入成功返回1，否则返回0。

(5) LeftChild (&T, e)

- 初始条件：二叉树T存在，e是T中的某个结点。
- 操作结果：如果结点e存在左孩子结点，则返回e的左孩子结点元素值，否则返回空。

(6) RighthChild (&T, e)

- 初始条件：二叉树T存在，e是T中的某个结点。
- 操作结果：如果结点e存在右孩子结点，则返回e的右孩子结点元素值，否则返回空。

(7) DeleteChild (p, int LR)

- 初始条件：二叉树T存在，p指向T中的某个结点，LR为0或1。
- 操作结果：根据LR为0或1，删除T中p所指向的左子树或右子树。如果删除成功，返回1，否则返回0。

(8) PreOrderTraverse (T)

- 初始条件：二叉树T存在。
- 操作结果：先序遍历二叉树T。二叉树的先序遍历，就是先访问根结点，再访问左子树，最后访问右子树的顺序，访问且对每个结点访问一次的操作。

(9) InOrderTraverse (T)

- 初始条件：二叉树T存在。
- 操作结果：中序遍历二叉树。二叉树的中序遍历，就是按照先访问左子树，再访问根结点，最后访问右子树的次序对二叉树中的每个结点访问，且仅访问一次的操作。

(10) PostOrderTraverse (T)

- 初始条件：二叉树T存在。

- 操作结果：后序遍历二叉树T。二叉树的后序遍历，就是按照先访问左子树，再访问右子树，最后访问根结点的次序对二叉树中的每个结点访问，且仅访问一次的操作。

(11) LevelTraverse (T)

- 初始条件：二叉树T存在。
- 操作结果：层次遍历二叉树T。二叉树的层次遍历，就是按照从上到下，从左到右，依次对二叉树中的每个结点进行访问。

(12) BitTreeDepth (T)

- 初始条件：二叉树T存在。
- 操作结果：求二叉树T的深度。二叉树的深度即二叉树的结点层次的最大值。如果二叉树非空，返回二叉树的深度；如果二叉树为空，返回0。

9.3 二叉树的存储表示与实现

二叉树存储结构也有顺序存储和链式存储两种。其中，链式存储是二叉树最常用的存储结构。

9.3.1 二叉树的顺序存储

若对一棵二叉树按照层次从上到下、自左至右进行编号，则对于完全二叉树来说，每个结点的编号可通过二叉树的性质计算得到，因此，完全二叉树中的结点可依次存放在一维数组中，它的顺序存储如图9-16所示。

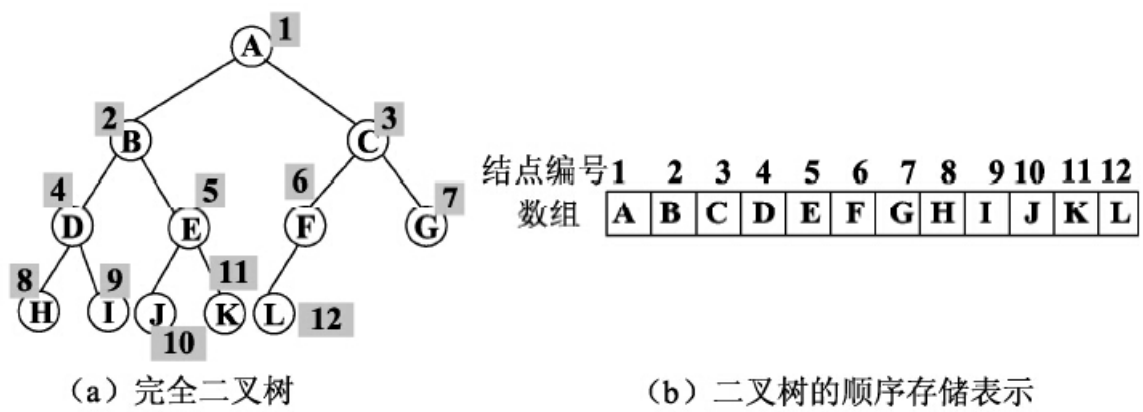


图9-16 完全二叉树的顺序存储表示

按照同样的方法，若将非完全二叉树的结点也按照从上到下、从左到右的顺序依次存放在一维数组中。为了能够正确反映二叉树中结点之间的逻辑关系，还需要在一维数组中空出二叉树中不存在结点的位置（用 \wedge 表示），如图9-17所示。

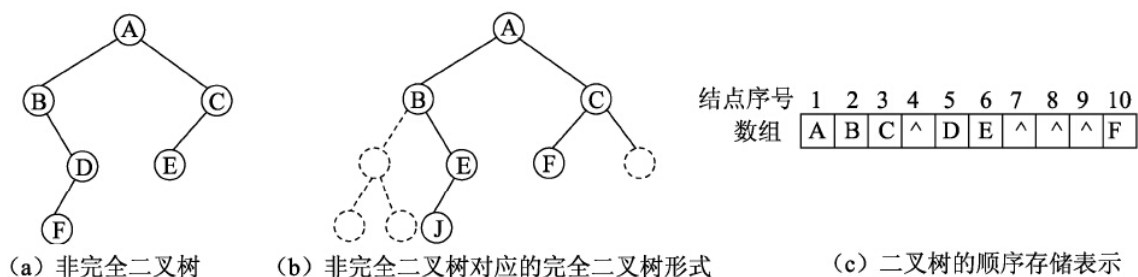


图9-17 非完全二叉树的顺序存储表示

不难看出，完全二叉树来采用顺序存储是比较适合的，因为采用顺序存储不仅能表示出结点之间的关系，还节省了内存空间。但对于非完全二叉树来说，这种存储方式极大地浪费了存储空间。特别是在每个结点只有右孩子，而没有左孩子的情况下，则需要占用 $2^k - 1$ 个存储单元，而实际只存储了 k 个有效结点。

9.3.2 二叉树的链式存储

根据二叉树的定义，二叉树的结点由一个数据元素及左、右两个分支构成，为了表示二叉树，每个结点应至少包含数据域、指向左孩子的指针域和指向右孩子的指针域3个域，如图9-18所示。其中，data域存放结点信息，lchild域指向该结点的左孩子结点，rchild域指向该结点的右孩子结点。利用这种结点结构得到的二叉树存储结构称为二叉链表。从二叉链表的根结点开始，通过结点的左右孩子指针域就可以访问二叉树中的每一个结点。

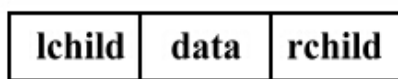


图9-18 二叉链表存储结构结点

一棵二叉树的二叉链表如图9-19所示。

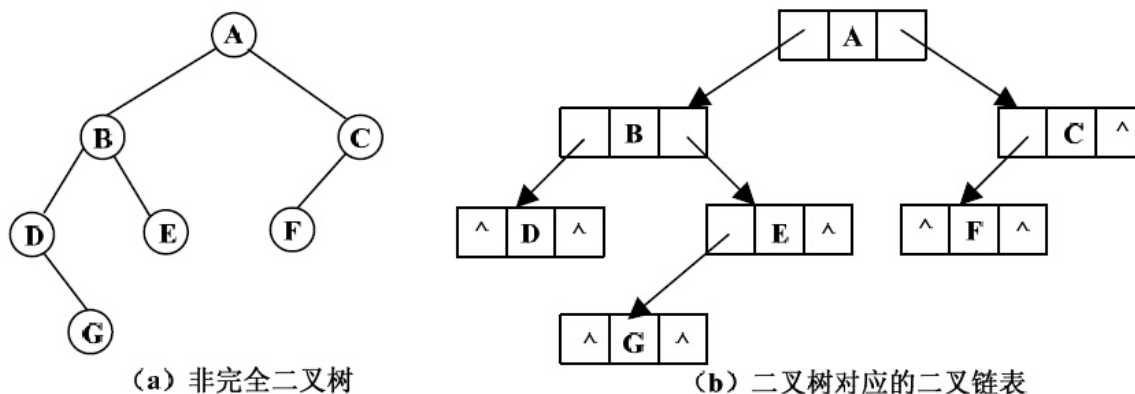


图9-19 二叉树的二叉链表存储表示

有时为了能找到任意一个结点的双亲结点，在二叉链表的存储结构中增加一个指向双亲结点的指针域parent，结点的存储结构如图9-20所示。这种结点结构得到的二叉树存储结构称为三叉链表。



图9-20 三叉链表结点结构

最为普遍的情况是将二叉树采用二叉树链表表示，二叉链表的结点结构描述如下。

```
typedef struct Node                                /*  
二叉链表的结点结构*/  
{  
    DataType data;                               /*  
数据域*/  
    struct Node *lchild;                         /*  
指向左孩子结点*/  
    struct Node *rchild;                         /*  
指向右孩子结点*/  
}*BiTree,BitNode;
```

9.3.3 二叉树的基本运算

采用二叉链表存储结构表示的二叉树的基本运算实现如下（以下算法的实现保存在文件“LinkBiTree.h”中）。

①初始化二叉树，代码如下。

```
void InitBiTree(BiTree *T)
/*
  二叉树的初始化*/
{
    *T=NULL;
}
```

②销毁二叉树，代码如下。

```
void DestroyBiTree(BiTree *T)
/*
  销毁二叉树*/
{
    if(*T) /*
如果是非空二叉树*/
    {
        if((*T)->lchild)
            DestroyBiTree(&((*T)->lchild));
        if((*T)->rchild)
            DestroyBiTree(&((*T)->rchild));
        free(*T);
        *T=NULL;
    }
}
```

③创建二叉树。根据二叉树的递归定义，先生成二叉树的根结点，将元素值赋给结点的数据域，然后递归创建左子树和右子树。其中#表示空。创建二叉树的算法实现如下。

```
void CreateBiTree(BiTree *T)
/*
  创建二叉树的递归实现*/
{
    DataType ch;
    scanf("%c", &ch);
    if(ch=='#')
        *T=NULL;
```

```

        else
        {
            *T=(BiTree)malloc(sizeof(BitNode));
生成根结点*/
            if(!(*T))
                exit(-1);
            (*T)->data=ch;
            CreateBitTree(&((*T)->lchild));
构造左子树*/
            CreateBitTree(&((*T)->rchild));
构造右子树*/
        }
    }
}

```

④二叉树的插入操作。如果p指向的是非空二叉树，则判断LR的值，如果LR为0，则将子树c插入T中，使c成为*p的左子树，结点*p原来的左子树成为c的右子树；如果LR为1，则将子树插入T中，使c成为*p的右子树，结点*p原来的右子树成为c的右子树。这里的c与T不相交且右子树为空。二叉树插入操作的算法实现如下。

```

int InsertChild(BiTree p,int LR,BiTree c)
/*
二叉树的插入操作*/
{
    if(p)
如果p
指向的二叉树非空*/
    {
        if(LR==0)
        {
            c->rchild=p->lchild;
的原来的左子树成为c
的右子树*/
            p->lchild=c;
子树c
作为p
的左子树*/
        }
        else
        {
            c->rchild=p->rchild;
的原来的右子树作为c
的右子树*/
            p->rchild=c;
子树c
作为p
的右子树*/
        }
        return 1;
    }
    return 0;
}

```

⑤返回二叉树T的左孩子结点元素值。如果存在元素值为e的结点，并且该结点的左孩子结点存在，则返回该结点的左孩子结点的元素值。返回e的左孩子结点元素值的算法实现如下。

```
DataType LeftChild(BiTree T,DataType e)
/*
返回二叉树的左孩子结点元素值操作*/
{
    BiTree p;
    if(T)
    如果二叉树不空*/
    {
        p=Point(T,e);
        是元素值e
        的结点的指针*/
        if(p&& p->lchild)
        如果p
        不为空且p
        的左孩子结点存在*/
        return p->lchild->data;
        返回p
        的左孩子结点的元素值*/
    }
    return;
}
```

⑥返回e的右孩子结点元素值。如果元素值为e的结点存在，并且该结点的右孩子结点存在，则返回e的右孩子结点的元素值。返回e的右孩子结点元素值的算法实现如下。

```
DataType RightChild(BiTree T,DataType e)
/*
返回二叉树的右孩子结点元素值*/
{
    BiTree p;
    if(T)
    如果二叉树不空*/
    {
        p=Point(T,e);
        是元素值e
        的结点的指针*/
        if(p&& p->rchild)
        如果p
        不为空且p
        的右孩子结点存在*/
        return p->rchild->data;
        返回p
        的右孩子结点的元素值*/
    }
    return;
}
```

⑦返回二叉树给定结点的指针。在二叉树中查找指向元素值为e的结点，如果找到该结点，则返回该结点的指针，否则返回NULL。

定义一个队列Q，用来存放二叉树中结点的指针，从根结点出发，判断结点的值是否等于e，如果相等，则返回该结点的指针；否则将指向该结点左孩子结点的指针和指向右孩子结点的指针入队列。然后将队头的指针出队列，判断该指针指向的结点的元素值是否等于e，若相等则返回该结点的指针，否则继续将指向左孩子结点的指针和指向右孩子结点的指针入队列。重复执行此操作，直到队列为空。

返回二叉树指定结点的指针的算法实现如下。

```
BiTree Point(BiTree T,DataType e)
/*
查找元素值为e
的结点的指针*/
{
    BiTree Q[MaxSize];          /*
定义一个队列，用于存放二叉树中结点的指针*/
    int front=0,rear=0;          /*
初始化队列*/
    BitNode *p;
    if(T)                        /*
如果二叉树非空*/
    {
        Q[rear]=T;
        rear++;
        while(front!=rear)      /*
如果队列非空*/
        {
            p=Q[front];        /*
取出队头指针*/
            front++;            /*
将队头指针出队*/
            if(p->data==e)
                return p;
            if(p->lchild)        /*
如果左孩子结点存在，将左孩子指针入队*/
            {
                Q[rear]=p->lchild; /*
左孩子结点的指针入队*/
                rear++;
            }
            if(p->rchild)        /*
如果右孩子结点存在，将右孩子指针入队*/
            {
                Q[rear]=p->rchild; /*
右孩子结点的指针入队*/
                rear++;
            }
        }
    }
}
```



```
    }  
    }  
    return NULL;  
}
```

⑧删除子树操作。先判断p指向的子树是否为空，如果不为空，则判断LR的值。如果LR为0，则删除p指向结点的左子树；如果LR为1，则删除p指向结点的右子树。如果删除成功，返回1，否则返回0。删除子树操作的算法实现如下。

```
int DeleteLeftChild(BiTree p,int LR)  
/*  
  二叉树的左删除操作*/  
{  
    if(p) /*  
    如果p  
    不空*/  
    {  
        if(LR==0)  
            DestroyBitTree(&(p->lchild)); /*  
    删除左子树*/  
        else  
            DestroyBitTree(&(p->rchild)); /*  
    删除右子树*/  
        return 1;  
    }  
    return 0;  
}
```

9.4 遍历二叉树

在二叉树的一些应用中，经常需要在树中查找具有某种特征的结点，这就是二叉树的遍历问题。本节主要介绍什么是遍历二叉树、二叉树的先序遍历、二叉树的中序遍历及二叉树的后序遍历。

9.4.1 什么是遍历二叉树

遍历二叉树（traversing binary tree）即按照某种规律对二叉树的每个结点进行访问，使得每个结点仅被访问一次的操作。这里的访问可以是统计结点的数据信息、输出结点信息等。

二叉树的遍历不同于线性表的遍历，对于二叉树来说，每个结点有两棵子树，因而需要寻找一种规律，使得二叉树的结点能排列在一个线性队列上，从而便于遍历。从这个意义上讲，二叉树的遍历过程其实也是将二叉树的非线性序列转换成一个线性序列的过程。

回顾二叉树的定义，二叉树是由根结点、左子树和右子树构成。二叉树的基本结构如图9-21所示。如果能依次遍历这3个部分，就是遍历了整棵二叉树。如果用D、L、R分别代表遍历根结点、遍历左子树和遍历右子树，根据组合原理，有6种遍历方案，分别是DLR、DRL、LDR、LRD、RDL和RLD。

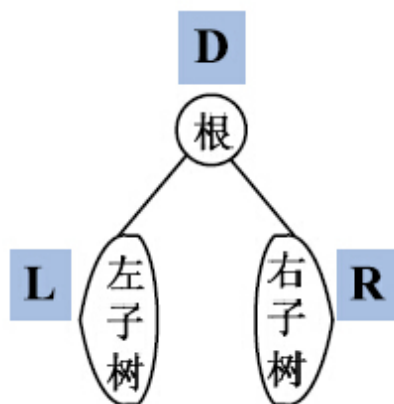


图9-21 二叉树的基本结构

如果限定先左后右的次序，则以上6种遍历方案只剩下3种方案，分别是DLR、LDR和LRD。其中，DLR称为先序（根）遍历，LDR称为中序（根）遍历，LRD称为后序（根）遍历。

9.4.2 遍历二叉树

1. 先序遍历二叉树

先序遍历二叉树的递归定义如下。

如果二叉树为空，则执行空操作。如果二叉树非空，则执行以下操作。

- (1) 访问根结点。
- (2) 先序遍历左子树。

(3) 先序遍历右子树。

根据二叉树的递归定义，对每一棵二叉树重复执行以上的遍历操作，就可以得到二叉树的先序序列。例如，图9-22所示二叉树的先序序列为A、B、D、G、E、H、I、C、F、J。

下面分析A的左子树{B, D, E, G, H, I}先序遍历过程。根据先序遍历的递归定义，先访问根结点B，然后先序遍历B的左子树{D, G}，最后先序遍历B的右子树{E, H, I}。访问过B之后，开始遍历B的左子树{D, G}，在子树{D, G}中，先访问根结点D，因D没有左子树，故遍历其右子树G。接着按照同样的方法遍历B的右子树。最后得到结点A的左子树先序序列B、D、G、E、H、I。

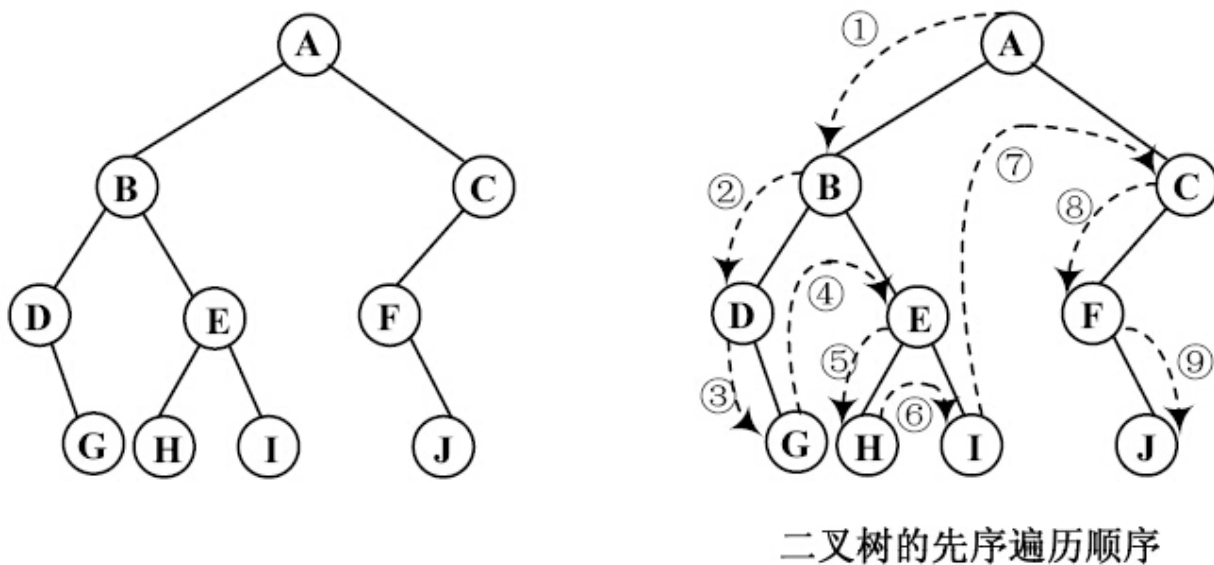


图9-22 二叉树

由二叉树的先序遍历递归定义，可得到先序遍历二叉树的递归算法如下。

```
void PreOrderTraverse(BiTree T)
/*
先序遍历二叉树的递归实现*/
{
    if(T) /*
    如果二叉树不为空*/
    {
        printf(
        "%2c", T->data); /*
        访问根结点*/
        PreOrderTraverse(T->lchild); /*
        先序遍历左子树*/
        PreOrderTraverse(T->rchild); /*
        先序遍历右子树*/
    }
}
```

2. 中序遍历二叉树

中序遍历二叉树的递归定义如下。

如果二叉树为空，则执行空操作。如果二叉树非空，则执行以下操作。

- (1) 中序遍历左子树。
- (2) 访问根结点。
- (3) 中序遍历右子树。

根据二叉树的中序递归定义，对每一棵二叉树重复执行以上的递归遍历操作，就可以得到二叉树的中序序列。例如，图9-22所示的二叉树中序遍历顺序如图9-23所示，得到的中序序列为D、G、B、H、E、I、A、F、J、C。图中阴影表示中序遍历的第一个结点。

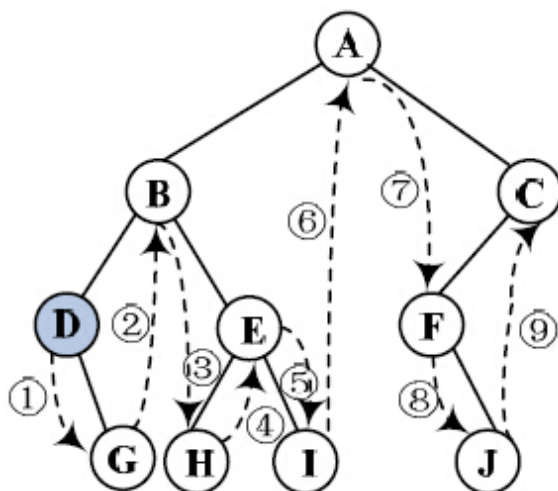


图9-23 二叉树的中序遍历顺序

下面分析A的左子树{B, D, E, G, H, I}中序遍历过程。根据中序遍历的递归定义，需要先中序遍历B的左子树{D, G}，然后访问根结点B，最后中序遍历B的右子树{E, H, I}。在子树{D, G}中，D是根结点，没有左子树，因此就轮到访问根结点D，接着遍历D的右子树，因为右子树只有一个结点G，所以访问G。

在左子树遍历完毕之后，访问根结点B。最后轮到遍历B的右子树{E, H, I}，E是子树{E, H, I}的根结点，需要先遍历左子树{H}，因为左子树只有一个H，所以访问H，然后访问根结点E，最后轮到遍历右

子树{I}，右子树也只有一个结点，所以访问I，B的右子树访问完毕。
因此，A的右子树的中序序列为D、G、B、H、E和I。

经过分析发现，中序序列中的根结点把中序序列分为左右两棵子树序列，左边为左子树序列，右边是右子树序列。

依据二叉树的中序遍历递归定义，得到中序遍历二叉树的递归算法如下。

```
void InOrderTraverse(BiTree T)
/*
中序遍历二叉树的递归实现*/
{
    if(T)
    如果二叉树不为空*/
    {
        InOrderTraverse(T->lchild);
        中序遍历左子树*/
        printf(
        "%2c", T->data);
        访问根结点*/
        InOrderTraverse(T->rchild);
        中序遍历右子树*/
    }
}
```

3. 后序遍历二叉树

后序遍历二叉树的递归定义如下。

如果二叉树为空，则执行空操作。如果二叉树非空，则执行以下操作。

(1) 后序遍历左子树。

(2) 后序遍历右子树。

(3) 访问根结点。

根据二叉树的后序递归定义，对每一棵二叉树重复执行以上的递归遍历操作，就可以得到二叉树的后序序列。图9-22所示的二叉树的后序遍历顺序如图9-24所示，得到的二叉树的后序序列为G、D、H、I、E、B、J、F、C、A。

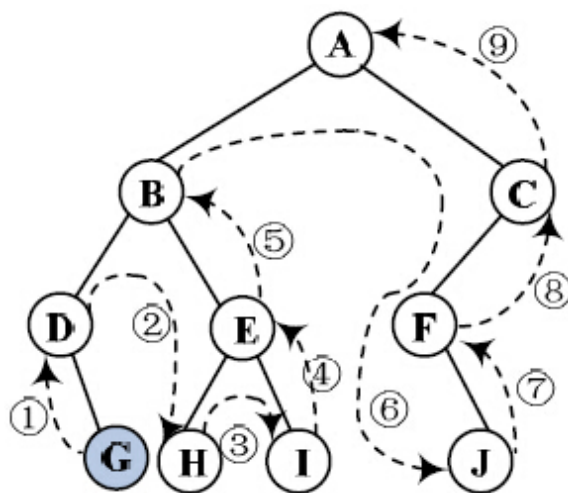


图9-24 二叉树的后序遍历顺序

若要后序遍历A的左子树{B, D, E, G, H, I}，根据后序遍历的递归定义，需要先后序遍历B的左子树{D, G}，接着后序遍历B的右子树{E, H, I}，最后访问根结点B。在子树{D, G}中，D是根结点，没有左子树，因此遍历D的右子树，因为右子树只有一个结点G，所以访问G，接着访问根结点D。

在B的左子树遍历完毕之后，接着遍历B的右子树{E, H, I}，E是子树{E, H, I}的根结点，需要先遍历左子树{H}，因为左子树只有一个H，所以访问H，然后遍历右子树{I}，右子树也只有一个结点，所以访问I，最后访问子树{E, H, I}的根结点E。此时，B的左、右子树均访问完毕。最后访问结点B。因此，A的左子树的后序序列为G、D、H、I、E和B。

依据后序遍历二叉树的递归定义，得到后序遍历二叉树的递归算法如下。

```
void PostOrderTraverse(BiTree T)
/*
后序遍历二叉树的递归实现*/
{
    if(T) /*
如果二叉树不为空*/
    {
        PostOrderTraverse(T->lchild); /*
后序遍历左子树*/
        PostOrderTraverse(T->rchild); /*
后序遍历右子树*/
        printf("%2c",T->data); /*
访问根结点*/
    }
}
```

9.4.3 非递归遍历二叉树——基于栈的递归消除

第4章已经对递归的消除作了具体讲解，现在利用栈来实现中序遍历二叉树的非递归算法。

1. 中序遍历二叉树

下面探讨中序遍历二叉树的非递归算法实现。

算法思想： 从二叉树的根结点开始，将根结点的指针入栈，执行以下操作。

(1) 若左孩子结点存在，将左孩子结点入栈。重复执行此操作，直到左孩子为空。

(2) 将栈顶元素出栈，访问该结点。

(3) 若右孩子不为空，则访问右孩子结点。

(4) 重复执行 (1) (2) (3)，直到栈空为止。

中序遍历二叉树的非递归算法实现如下。

```
void InOrderTraverse(BiTree T)
/*
中序遍历二叉树的非递归实现*/
{
    BiTree stack[MaxSize];          /*
    定义一个栈，用于存放结点的指针*/
    int top;                          /*
    定义栈顶指针*/
    BitNode *p;                      /*
    定义一个结点的指针*/
    top=0;                           /*
    初始化栈*/
    p=T;
    while (p!=NULL || top>0)
    {
        while (p!=NULL)              /*
        如果p
        不空，访问根结点，遍历左子树*/
        {
            stack[top++]=p;          /*
            将p
            入栈*/
            p=p->lchild;              /*
            遍历左子树*/
        }
        if (top>0)                    /*
        如果栈不空*/
```

```

        {
            p=stack[--top];          /*
栈顶元素出栈*/
            printf("%2c",p->data);    /*
访问根结点*/
            p=p->rchild;              /*
遍历右子树*/
        }
    }
}

```

2. 后序遍历二叉树

下面探讨后序遍历二叉树的非递归算法实现。

算法思想： 从二叉树的根结点开始，将根结点的指针入栈，执行以下操作。

(1) 若左孩子结点存在，将左孩子结点入栈。重复执行此操作，直到结点的左孩子不存在。

(2) 若栈顶结点的右子树为空，或者栈顶结点的右孩子为刚访问过的结点，则退栈并访问该结点，然后将当前结点的指针置为空。

(3) 否则走右子树。

(4) 重复执行以上 (1) 和 (2)，直到栈空为止。

后序遍历二叉树的非递归算法实现如下。

```

void PostOrderTraverse(BiTree T)
/*
后序遍历二叉树的非递归实现*/
{
    BiTree stack[MaxSize];
    ... ..
}
/*

```

```

定义一个栈，用于存放结点的指针*/
    int top; /*
定义栈顶指针*/
    BitNode *p,*q; /*
定义结点的指针*/
    top=0; /*
初始化栈*/
    p=T,q=NULL; /*
初始化结点的指针*/
    while(p!=NULL||top>0)
    {
        while(p!=NULL) /*
如果p
不空，访问根结点，遍历左子树*/
        {
            stack[top++]=p; /*
将p
入栈*/
            p=p->lchild; /*
遍历左子树*/
        }
        if(top>0) /*
如果栈不空*/
        {
            p=stack[top-1]; /*
取栈顶元素*/
            if(p->rchild==NULL||p->rchild==q) /*
如果p
没有右孩子结点，或右孩子结点已经访问过*/
            {
                printf("%2c",p->data); /*
访问根结点*/
                q=p; /*
用q
保存刚刚访问过的结点*/
                p=NULL; /*
准备遍历右子树或准备访问根结点*/
                top--; /*
出栈*/
            }
            else
                p=p->rchild;
        }
    }
}

```

9.5 遍历二叉树的应用

二叉树的遍历应用非常广泛，本节主要通过几个例子来说明遍历二叉树的典型应用。本节主要给大家介绍如何利用遍历二叉树的算法思想输出二叉树及计算二叉树的结点。

9.5.1 按层次输出二叉树

打印输出二叉树的方式有多种，可以是按照先序、中序、后序的方式输出二叉树，还可以是按层次输出二叉树。

按层次输出二叉树的结点可利用队列实现，先定义一个队列`queue`，用来存放结点信息。从根结点出发，依次把每一层的结点入队，当一层结点入队完毕之后，将队头元素出队，输出该结点，然后判断结点是否存在左、右孩子，如果存在，则将左、右孩子入队。重复执行以上操作，直到队空为止。最后得到的输出序列就是按二叉树层次的输出序列。

以上算法的执行流程如图9-25所示。

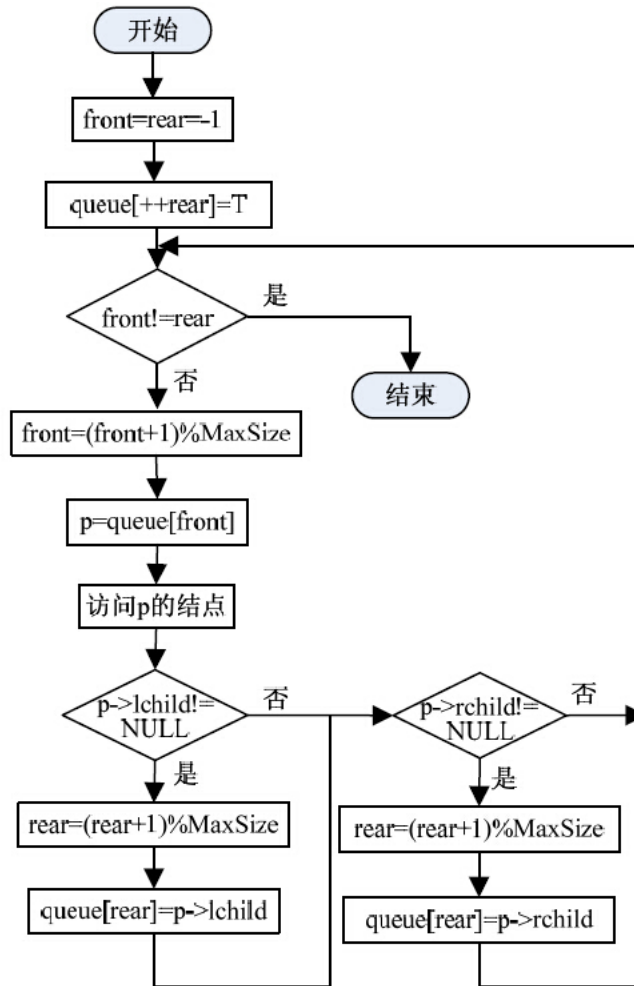


图9-25 按层次输出二叉树结点的程序流程图

按层次输出二叉树的算法实现如下。

```

void LevelPrint(BiTree T)
/*
按层次打印二叉树*/
{
    BiTree queue[MaxSize];
    BitNode *p;
    int front, rear;
    定义队列的队头指针和队尾指针*/
    front=rear=-1;
    队列初始化为空*/
    rear++;
    队尾指针加1*/
    queue[rear]=T;
    将根结点指针入队*/
    while(front!=rear)
    如果队列不为空*/
    {
        front=(front+1)%MaxSize;
        p=queue[front];
        取出队头元素*/
        printf("%c ", p->data);
        输出根结点*/
        if(p->lchild!=NULL)
        如果左孩子不为空, 将左孩子结点指针入队*/
        {
            rear=(rear+1)%MaxSize;
            queue[rear]=p->lchild;
        }
        if(p->rchild!=NULL)
        如果右孩子不为空, 将右孩子结点指针入队*/
    }
}
  
```

```

    {
        rear=(rear+1)%MaxSize;
        queue[rear]=p->rchild;
    }
}

```

9.5.2 二叉树的计数

二叉树的计数也可以通过遍历二叉树来实现，关于二叉树计数的算法有求二叉树叶子结点的个数、非叶子结点的个数。

1. 计算二叉树叶子结点的个数

求二叉树的叶子结点的个数递归定义如下。

$$\text{leaf}(T) = \begin{cases} 0, & \text{当} T = \text{NULL} \text{时} \\ 1, & \text{当} T \text{的左右孩子均为空时} \\ \text{leaf}(T \rightarrow \text{lchild}) + \text{leaf}(T \rightarrow \text{rchild}), & \text{其他情况} \end{cases}$$

翻译成汉语，上面公式的含义为，当二叉树为空时，叶子结点个数为0；当二叉树只有一个根结点时，根结点就是叶子结点，叶子结点个数为1；其他情况下，计算左子树与右子树中叶子结点的和。由此可得到统计叶子结点个数的算法。

求二叉树叶子结点个数的算法实现如下。

```

int LeafNum(BiTree T)
/*
求二叉树中叶子结点的个数*/
{
    if(!T) /*
如果是空二叉树，返回0*/
        return 0;
    else
        if(!T->lchild && !T->rchild) /*
如果左子树和右子树都为空，返回1*/
            return 1;
        else
            return LeafNum(T->lchild) + LeafNum(T->rchild); /*
把左子树叶子结点个数与右子树叶子结
点个数相加*/
}

```

2. 求二叉树的非叶子结点个数

二叉树的非叶子结点的个数的递归定义如下。

$$\text{NotLeaf}(T) = \begin{cases} 0, & \text{当} T = \text{NULL} \text{时} \\ 0, & \text{当} T \text{的左右孩子均为空时} \\ \text{NotLeaf}(T \rightarrow \text{lchild}) + \text{NotLeaf}(T \rightarrow \text{rchild}) + 1, & \text{其他情况} \end{cases}$$

含义为，当二叉树为空时，非叶子结点个数为0；当二叉树只有根结点时，根结点为叶子结点，非叶子结点个数为0；在其他情况下，非叶子结点个数为左子树与右子树中非叶子结点的个数再加1（根结点）。

求二叉树中非叶子结点个数的算法实现如下。

```
int NotLeafNum(BiTree T)
/*
求二叉树中非叶子结点的个数*/
{
    if (!T) /*
如果是空二叉树*/
        return 0; /*
返回0*/
    else if (!T->lchild&&!T->rchild) /*
如果是叶子结点*/
        return 0; /*
返回0*/
    else /*
如果是非叶子结点，也不是根结点*/
        return NotLeafNum(T->lchild)+NotLeafNum(T->rchild)+1; /*
左右子树的非叶子结点个数
加根结点个数*/
}
```

3. 计算二叉树的所有结点数

二叉树的所有结点的个数的递归定义如下。

$$\text{AllNodes}(T) = \begin{cases} 0, & \text{当} T = \text{NULL} \text{时} \\ 1, & \text{当} T \text{-lchild} = \text{NULL} \text{且} T \text{-rchild} = \text{NULL} \\ \text{AllNodes}(T \text{-lchild}) + \text{AllNodes}(T \text{-rchild}) + 1, & \text{其他情况} \end{cases}$$

若二叉树为空，则结点个数为0；在二叉树不空的情况下，若左、右子树为空，则结点数为1；否则，二叉树的结点数为左、右子树的结点数之和加1。

求二叉树中所有结点个数的算法实现如下。

```
int AllNodes(BiTree T)
/*
求二叉树中所有结点的个数*/
{
    if (!T) /*
如果是空二叉树*/
        return 0; /*
返回0*/
    else if (!T->lchild&&!T->rchild) /*
如果是叶子结点*/
        return 1; /*
返回1*/
    else /*
如果是非叶子结点，也不是根结点*/
        return AllNodes(T->lchild)+AllNodes(T->rchild)+1; /*
左右子树结点个数加根结点个数*/
}
```

4. 计算二叉树的深度

二叉树的深度递归定义如下。

$$\text{depth}(T) = \begin{cases} 0, & \text{当 } T = \text{NULL} \text{ 时} \\ 1, & \text{当 } T \text{ 的左右孩子均为空时} \\ \max(\text{depth}(T \rightarrow \text{lchild}), \text{depth}(T \rightarrow \text{rchild})) + 1, & \text{其他情况} \end{cases}$$

含义为，当二叉树为空时，其深度为0；当二叉树只有根结点时，即结点的左、右子树均为空，二叉树的深度为1；在其他情况下，求二叉树的左、右子树的最大值再加1（根结点）。由此，得到二叉树的深度的算法如下。

```
int BitTreeDepth(BiTree T)
/*
计算二叉树的深度*/
{
    if (T == NULL)
        return 0;
    return
        BitTreeDepth(T->lchild) > BitTreeDepth(T->rchild) ? 1 + BitTreeDepth(T->lchild) :
        1 + BitTreeDepth(T->rchild);
}
```

9.5.3 求叶子结点的最大最小枝长

求二叉树的所有叶子结点的最大枝长的递归模型如下。

$$\text{MaxLeaf}(b) = \begin{cases} 0, & \text{若 } b = \text{NULL} \\ \max(\text{MaxLeaf}(b \rightarrow \text{left}), \text{MaxLeaf}(b \rightarrow \text{right})) + 1, & \text{若 } b \neq \text{NULL} \end{cases}$$

求二叉树的所有叶子结点的最小枝长的递归模型如下。

$$\text{MinLeaf}(b) = \begin{cases} 0, & \text{若 } b = \text{NULL} \\ \min(\text{MinLeaf}(b \rightarrow \text{left}), \text{MinLeaf}(b \rightarrow \text{right})) + 1, & \text{若 } b \neq \text{NULL} \end{cases}$$

相应的算法如下。

```
void MaxMinLeaf(BiTree T, int *max, int *min)
{
    int m1, m2, n1, n2;
    if (T == NULL)
    {
        *max = 0;
        *min = 0;
    }
    else
    {
        MaxMinLeaf(T->lchild, m1, n1);
        MaxMinLeaf(T->rchild, m2, n2);
        *max = (m1 > m2 ? m1 : m2) + 1;
        *min = (m1 < m2 ? m1 : m2) + 1;
    }
}
```

9.5.4 判断两棵二叉树是否相似

所谓两棵二叉树 T_1 和 T_2 相似的定义是，或者 T_1 和 T_2 都是空树，或者 T_1 和 T_2 的根结点相似，且 T_1 和 T_2 的左子树和右子树都相似。

判断两棵二叉树是否相似可在用同样的次序遍历这两棵二叉树的过程中进行，因此可用递归算法实现。设 $t1$ 和 $t2$ 分别是指向二叉树 T_1 和 T_2 中当前结点的指针，初始时指向根结点。若 $t1$ 和 $t2$ 皆为空，则这两棵二叉树相似，返回1；若 $t1$ 和 $t2$ 中有一个为空，而另一个不为空，则这两棵二叉树不相似，返回0；否则遍历这两棵二叉树的左子树，检查是否相似，然后遍历右子树，检查是否相似。

算法实现如下。

```
int BiTree_Like(BiTree t1, BiTree t2)
{
    if (t1 == NULL && t2 == NULL)
        return 1;
    if ( (t1 != NULL && t2 == NULL) || (t1 == NULL && t2 != NULL) )
        return 0;
    if (BiTree_Like(t1->lchild, t2->lchild))
        /*
    若左子树相似 */
        return (BiTree_Like(t1->rchild, t2->rchild)); /*
    则两棵是否相似取决于右子树 */
    else
        return 0;
}
```

9.5.5 交换二叉树的左右子树

同样，在遍历二叉树的过程中也可以交换各个结点的左右子树，算法实现如下。

```
Void BiTree_Swap(BiTree T)
{
    BiTree p;
    if (T != NULL)
        if (T->lchild != NULL || T->rchild != NULL) /*
    若T
    的两棵子树不同时为空，则交换两棵子树 */
        {
            p = T->lchild;
            T->lchild = T->rchild;
            T->rchild = p;
        }
    if (T->lchild != NULL) /*
    若T
    的左子树不为空，则将左子树的左右子树交换之 */
        BiTree_Swap(T->lchild);
    if (T->rchild != NULL) /*
    若T
    的右子树不为空，则将右子树的左右子树交换之 */
        BiTree_Swap(T->rchild);
}
```

注意 也可用后序遍历的方式实现交换左右两棵子树，但不宜用中序遍历的方式实现，因为若用中序遍历的算法，则仅交换了根结点的左右孩子。

9.5.6 求根结点到 r 结点之间的路径

假设二叉树采用二叉链表方式存储，root指向根结点，r所指结点为任一结点，试编写算法，求出从根结点到r结点之间的路径。

分析 由于后序遍历的过程中，访问到r所指结点时，此时栈中所有结点均为r所指的祖先，这些祖先便构成了一条从根结点到r所指结点之间的路径，故可采用后序遍历，算法实现如下。

```
void path(BiTree root, BitNode *r)
{
    BitNode *p,*q;
    int i,top=0;
    BitNode *s[StackSize];
    q=NULL;
    p=root;
    while(p!=NULL || top!=0)
    {
        while(p!=NULL)
            /*
遍历左子树*/
        {
            top++;
            if(top>=StackSize)
                exit(-1);
            s[top]=p;
            p=p->lchild;
        }
        if(top>0) /*
若栈不为空*/
        {
            p=s[top];
            if(p->rchild == NULL || p->rchild==q) /*
根结点*/
            {
                if(p==r) /*
找到r
所指结点，则输出从根结点到r
所指结点之间的结点*/
                {
                    for(i=1;i<=top;i++)
                        printf("%4d",s[i]->data);
                    top=0;
                }
                else
                {
                    q=p; /*
用q
保存刚刚遍历过的结点*/
                    top--;
                    p=NULL;
                }
            }
            else
                p=p->rchild; /*
遍历右子树*/
        }
    }
}
```

本算法与后序非递归遍历二叉树算法唯一不同的地方是增加了如下语句。

```
if(p==r) /*
找到r
所指结点，则输出从根结点到r
所指结点之间的结点*/
{
    for(i=1;i<=top;i++)
        printf(
"%4d
",s[i]->data);
    top=0;
}
```

意即，如果找到r所指结点，则输出从根结点到r所指结点的路径。

9.6 线索二叉树

采用二叉链表作为二叉树的存储结构，只能找到结点的左、右孩子结点，而不能直接找到该结点的直接前驱和后继结点信息，这种信息只能在对二叉树的遍历过程中才能找到，显然这并不是最直接、最简便的方法。为了能快速找到任何一个结点的直接前驱和直接后继信息，需要对二叉树进行线索化。本节主要介绍的线索二叉树的定义、中序线索二叉树的、线索二叉树的遍历及线索二叉树的简单应用。

9.6.1 什么是线索化二叉树

为了在遍历二叉树的过程中能直接找到任何一个结点的直接前驱结点或者直接后继结点，可在二叉链表结点中增加两个指针域，一个用来指示结点的前驱；另一个用来指向结点的后继。如果这样做，需要为每个结点增加两个域的存储单元，也会使结点结构的利用率大大下降。

在二叉链表的存储结构中， n 个结点的二叉链表具有 $n+1$ 个空指针域（根据二叉树的分支特点，分支数目为 $B=n-1$ ，即非空链域为 $n-1$ 个，故空链域有 $2n - (n-1) = n+1$ 个）。

因此，可以利用这些空指针域存放结点的直接前驱和直接后继的信息。假定，若结点存在左子树，则指针域`lchild`指示其左孩子结点，否则指针域`lchild`指示其直接前驱结点；若结点存在右子树，则指针域`rchild`指示其右孩子结点，否则指针域`rchild`指示其直接后继结点。

另外增加两个标志域ltag和rtag，分别用来区分指针域指向的是左孩子结点还是直接前驱结点，右孩子结点还是直接后继结点，这样的结点存储结构如图9-26所示。



图9-26 结点的存储结构

当ltag=0时，lchild指示结点的左孩子；当ltag=1时，lchild指示结点的直接前驱结点。当rtag=0时，rchild指示结点的右孩子；当rtag=1时，lchild指示结点的直接后继结点。

由这种存储结构构成的二叉链表称为二叉树的线索二叉树。采用这种存储结构的二叉链表称为线索链表。其中，指向结点直接前驱和直接后继的指针称为线索。在二叉树的先序遍历过程中，加上线索之后，得到先序线索二叉树。同理，在二叉树的中序（后序）遍历过程中，加上线索之后，得到中序（后序）线索二叉树。二叉树按照某种遍历方式使二叉树变为线索二叉树的过程称为二叉树的线索化。图9-27所示就是将二叉树进行先序、中序和后序遍历得到的线索二叉树。

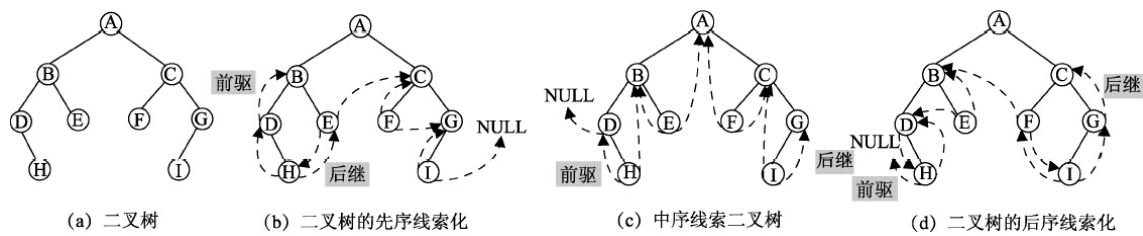


图9-27 二叉树的线索化

线索二叉树的存储结构类型描述如下。

```
typedef enum {Link, Thread} PointerTag; /*Link=0
表示指向孩子结点, Thread=1
表示指向前驱结点或后继结点*/
typedef struct Node
线索二叉树存储结构类型定义*/
{
    DataType data; /*
数据域*/
    struct Node *lchild, rchild; /*
指向左孩子结点的指针和右孩子结点的指针*/
    PointerTag ltag, rtag; /*
标志域*/
} *BiThrTree, BiThrNode;
```

9.6.2 线索二叉树

在二叉树遍历的过程中, 可得到结点的前驱信息和后继信息, 同时将结点的空指针域修改为其直接前驱或直接后继信息。因此, 二叉树的线索化就是对二叉树的遍历过程。这里以二叉树的中序线索化为例介绍二叉树的线索化。

为了便于算法操作, 在二叉链表中增加一个头结点。头结点的数据域可以存放二叉树的结点信息, 也可以为空。令头结点的指针域lchild指向二叉树的根结点, 指针域rchild指向二叉树中序遍历时的最后一个结点, 二叉树中的第一个结点的线索指针指向头结点。在初始化时, 使二叉树的头结点指针域lchild和rchild均指向头结点, 并将头结点的标志域ltag置为Link, 标志域rtag置为Thread。

线索化后的二叉树像一个循环链表, 既可以从线索二叉树中的第一个结点出发沿着结点的后继线索指针遍历整个二叉树, 也可以从线索二叉树的最

后一个结点出发沿着结点的前驱线索指针遍历整个二叉树。经过线索化的二叉树及存储结构如图9-28所示。

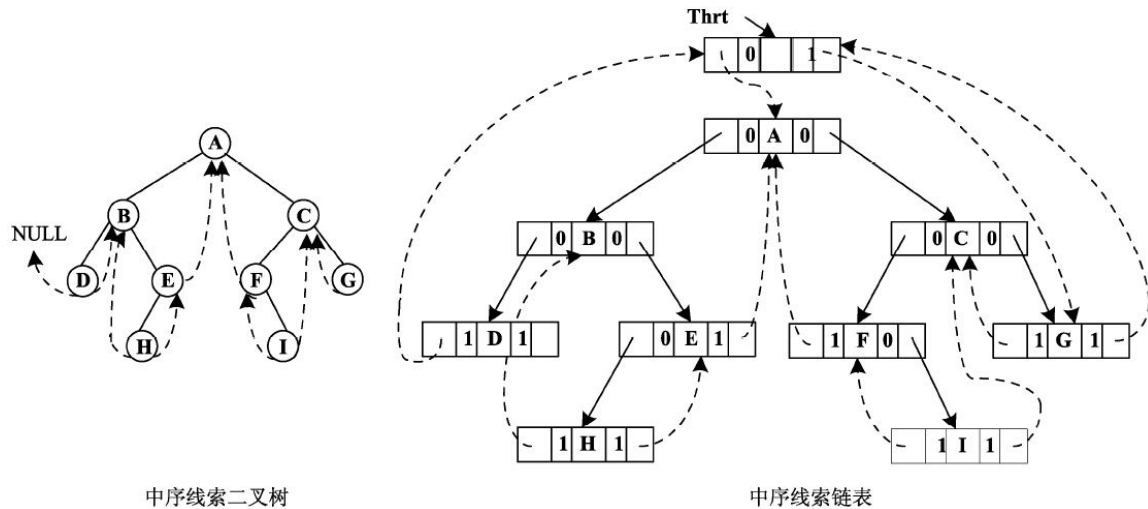


图9-28 中序线索二叉树

中序线索二叉树的算法实现如下。

```
BiThrTree pre; /*pre
始终指向已经线索化的结点*/
int InOrderThreading(BiThrTree *Thrt,BiThrTree T)
/*
通过中序遍历二叉树T
，使T
中序线索化。Thrt
是指向头结点的指针*/
{
    if(!(*Thrt=(BiThrTree)malloc(sizeof(BiThrNode)))) /*
为头结点分配内存单元*/
        exit(-1);
    /*
将头结点线索化*/
    (*Thrt)->ltag=Link; /*
修改前驱线索标志*/
    (*Thrt)->rtag=Thread; /*
修改后继线索标志*/
    (*Thrt)->rchild=*Thrt; /*
将头结点的rchild
指针指向自己*/
    if(!T) /*
如果二叉树为空，则将lchild
指针指向自己*/
        (*Thrt)->lchild=*Thrt;
    else
    {
        (*Thrt)->lchild=T; /*
将头结点的左指针指向根结点*/
        pre=*Thrt; /*
将pre
指向已经线索化的结点*/
```

```

        InThreading(T);                                /*
中序遍历进行中序线索化*/
        /*
将最后一个结点线索化*/
        pre->rchild=*Thrt;                            /*
将最后一个结点的右指针指向头结点*/
        pre->rtag=Thread;                              /*
修改最后一个结点的rtag
标志域*/
        (*Thrt)->rchild=pre;                          /*
将头结点的rchild
指针指向最后一个结点*/
    }
    return 1;
}
void InThreading(BiThrTree p)
/*
二叉树的中序线索化*/
{
    if(p)
    {
        InThreading(p->lchild);                        /*
左子树线索化*/
        if(!p->lchild)                                  /*
前驱线索化*/
        {
            p->ltag=Thread;
            p->lchild=pre;
        }
        if(!pre->rchild)                                /*
后继线索化*/
        {
            pre->rtag=Thread;
            pre->rchild=p;
        }
        pre=p;                                          /*pre
指向的结点线索化完毕,使p
指向的结点成为前驱*/
        InThreading(p->rchild);                        /*
右子树线索化*/
    }
}

```

9.6.3 遍历线索二叉树

遍历线索二叉树，就是根据线索查找结点的前驱和后继。

1. 在中序线索二叉树中查找结点的直接前驱

在中序线索二叉树中，结点*p（即指针p指向的结点）的直接前驱就是其左子树的最右下端结点。若p->ltag=1，那么p->lchild指向的结点就是p的直接前驱结点。例如，图9.28所示的二叉树中，结点I的前驱标志域为1，

即Thread，则直接前驱为F，即lchild指向的结点。如果p->ltag=0，对于结点C，它的直接前驱为I，即结点C的左子树的最右下端结点。查找结点的直接前驱的算法实现如下。

```
BiThrNode *InOrderPre(BiThrNode *p)
/*
在中序线索树中找结点*p
的直接前驱*/
{
    BiThrNode *pre;
    if (p->ltag==Thread) /*
如果p
的标志域ltag
为线索，则p
的左子树结点即为前驱*/
        return p->lchild;
    else
    {
        pre=p->lchild; /*
查找p
的左孩子的最右下端结点*/
        while (pre->rtag==Link) /*
右子树非空时，沿右链往下查找*/
            pre=pre->rchild;
        return pre; /*pre
就是最右下端结点*/
    }
}
```

2. 在中序线索二叉树中查找结点的直接后继

在中序线索二叉树中，查找结点*p的中序直接后继与查找结点的直接前驱类似。若p->rtag=1，那么p->rchild指向的结点就是p的直接后继结点。例如，在图9-30中，结点E的后继标志域为1，即Thread，则其直接后继为A，即rchild指向的结点。若p->rtag=0，对于结点A，它的直接后继为F，即A的右子树的最左下端结点。查找结点的直接后继的算法实现如下。

```
BiThrNode *InOrderPost(BiThrNode *p)
/*
在中序线索树中查找结点*p
的直接后继*/
{
    BiThrNode *pre;
    if (p->rtag==Thread) /*
如果p
的标志域ltag
```

```

为线索，则p
的右子树结点即为后继*/
        return p->rchild;
    else
    {
        pre=p->rchild;
        /*
        查找p
        的右孩子的最左下端结点*/
        while (pre->ltag==Link)
            /*
            左子树非空时，沿左链往下查找*/
            pre=pre->lchild;
        return pre;
        /*pre
        就是最左下端结点*/
    }
}

```

3. 中序遍历线索二叉树

中序遍历线索二叉树可分为3步：第1步，从根结点出发，找到二叉树的最左下端结点并访问之；第2步，判断该结点的右标志域是否为线索指针，若为线索指针即 $p \rightarrow rtag == Thread$ ，表明 $p \rightarrow rchild$ 指向的是后继结点，则将指针移动到右孩子结点，并访问右孩子结点；第3步，将当前指针指向该右孩子结点。重复执行以上3步，就可访问完二叉树中的所有结点。中序遍历线索二叉树的过程，就是线索查找后继和查找右子树的最左下端结点的过程。中序遍历线索二叉树的算法实现如下。

```

int InOrderTraverse(BiThrTree T, int (* visit)(BiThrTree e))
/*
中序遍历线索二叉树。其中visit
是函数指针，指向访问结点的函数实现*/
{
    BiThrTree p;
    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))
                /*
                打印*/
                return 0;
            while(p->rtag==Thread && p->rchild!=T)
                /*
                访问后继结点*/
                {
                    p=p->rchild;
                    visit(p);
                }
            p=p->rchild;
        }
}

```

```
    }  
    return 1;  
}
```

由此可得出结论：对于中序线索二叉树，若ltag=0，则直接前驱为左子树的最右下端结点；若rtag=0，则其直接后继为右子树的最左下端结点。

4. 在后序线索二叉树中查找后继结点

在后序线索二叉树中查找后继较复杂些，可分3种情况，若结点x是二叉树的根，则其后继为空；若结点x是其双亲结点的右孩子或是其双亲结点的做孩子且其双亲没有右孩子，则其后继即为双亲结点；若结点x是其双亲结点的左孩子，且其双亲有右孩子，则其后继为双亲的右子树上按后序遍历得到的第一个结点。例如图9-29（d）为后序线索二叉树，结点H的后继为D，结点D的后继为E，结点F的后继为I，结点I的后继为G。

9.6.4 线索二叉树应用举例

【例9-1】 编写算法，创建如图9-28所示的二叉树，并将其中序线索化。任给一个结点，要求查找该结点的直接前驱和直接后继，例如，结点F的直接前驱是A，其直接后继是I。

【分析】 主要考查二叉树的中序线索化操作和在中序线索二叉树中查找结点的前驱和后继算法实现。

算法实现代码如下。

```

#include <stdio.h>
#include <malloc.h>
#include<stdlib.h>
#define MaxSize 100
/*
线索二叉树类型定义*/
typedef char DataType;
typedefenum {Link,Thread}PointerTag;
typedefstruct Node/*
结点类型*/
{
    DataType data;
    struct Node *lchild, *rchild; /*
左右孩子子树*/
    PointerTagltag,rtag; /*
线索标志域*/
}BiThrNode;
typedefBiThrNode *BiThrTree; /*
二叉树类型*/
/*
函数声明*/
void CreateBitTree2(BiThrTree *T,charstr[]); /*
创建线索二叉树*/
void InThreading(BiThrTree p); /*
中序线索化二叉树*/
intInOrderThreading(BiThrTree *Thrt,BiThrTree T); /*
通过中序遍历二叉树T
, 使T
中序线索化。 Thrt
是指向头结点的指针*/
intInOrderTraverse(BiThrTreeT,int (* visit)(BiThrTree e)); /*
中序遍历线索二叉树*/
int Print(BiThrTree T); /*
打印二叉树中的结点及线索标志*/
BiThrNode *FindPoint(BiThrTreeT,DataType e); /*
在线索二叉树中查找结点为e
的指针*/
BiThrNode *InOrderPre(BiThrNode *p); /*
查找中序线索二叉树的中序前驱*/
BiThrNode *InOrderPost(BiThrNode *p); /*
查找中序线索二叉树的中序后继*/
BiThrTree pre; /*pre
始终指向已经线索化的结点*/
void DestroyBitTree(BiThrTree *T); /*
销毁线索二叉树*/
void main()
/*
测试程序*/
{
    DataTypech;
    BiThrTreeT,Thrt;
    BiThrNode *p,*pre,*post;
    CreateBitTree2(&T,"(A(B(D,E(H)),C(F(I),G)))");
    printf("
输出线索二叉树的结点、前驱及后继信息: \n");
    InOrderThreading(&Thrt,T);
    printf("
序列前驱标志结点后继标志\n");
    InOrderTraverse(Thrt,Print);
    printf("
请输入要查找哪个结点的前驱和后继:");
    ch=getchar();
    p=FindPoint(Thrt,ch);
    pre=InOrderPre(p);
    printf("
元素%c
的中序直接前驱元素是:%c\n",ch,pre->data);
    post=InOrderPost(p);
    printf("
元素%c
的中序直接后继元素是:%c\n",ch,post->data);
    DestroyBitTree(&Thrt);
}

```

```

}
int Print(BiThrTree T)
/*
打印线索二叉树中的结点及线索*/
{
    static int k=1;
    printf("%2d\t%s\t\t%2c\t\t%s\t\n",k++,T->ltag=="Link"? "Thread",
                                                T->data,
                                                T->rtag=="Thread"? "Link");

    return 1;
}
void DestroyBitTree(BiThrTree *T)
/*
销毁二叉树*/
{
    if(*T) /*
如果是非空二叉树*/
    {
        if((*T)->lchild)
            DestroyBitTree(&((*T)->lchild));
        if((*T)->rchild)
            DestroyBitTree(&((*T)->rchild));
        free(*T);
        *T=NULL;
    }
}
void CreateBitTree2(BiThrTree *T, char str[])
/*
利用括号嵌套的字符串建立二叉链表*/
{
    char ch;
    BiThrTree stack[MaxSize]; /*
定义栈，用于存放指向二叉树中结点的指针*/
    int top=-1; /*
初始化栈顶指针*/
    int flag,k;
    BiThrNode *p;
    *T=NULL,k=0;
    ch=str[k];
    while(ch!='\0') /*
如果字符串没有结束*/
    {
        switch(ch)
        {
            case '(':
                stack[++top]=p;
                flag=1;
                break;
            case ')':
                top--;
                break;
            case ',':
                flag=2;
                break;
            default:
                p=(BiThrTree)malloc(sizeof(BiThrNode));
                p->data=ch;
                p->lchild=NULL;
                p->rchild=NULL;
                if(*T==NULL) /*
如果是第一个结点，表示是根结点*/
                    *T=p;
                else
                {
                    switch(flag)
                    {
                        case 1:
                            stack[top]->lchild=p;
                            break;
                        case 2:
                            stack[top]->rchild=p;
                            break;
                    }
                }
            }
        }
    }
}

```

```

        }
        if (stack[top]->lchild)
            stack[top]->ltag=Link;
        if (stack[top]->rchild)
            stack[top]->rtag=Link;
    }
    }
    ch=str[++k];
}
BiThrNode *FindPoint(BiThrTreeT, DataType e)
/*
中序遍历线索二叉树，返回元素值为e
的结点的指针。*/
{
    BiThrTree p;
    p=T->lchild;                                /*p
指向根结点*/
    while(p!=T)                                /*
如果不是空二叉树*/
    {
        while(p->ltag==Link)
            p=p->lchild;
        if (p->data==e)                        /*
找到结点，返回指针*/
            return p;
        while (p->rtag==Thread&&p->rchild!=T)    /*
访问后继结点*/
        {
            p=p->rchild;
            if (p->data==e)                    /*
找到结点，返回指针*/
                return p;
        }
        p=p->rchild;
    }
    return NULL;
}

```

程序运行结果如图9-29所示。

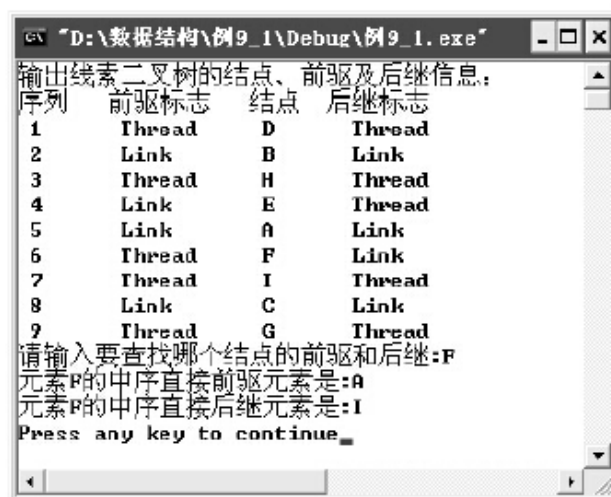


图9-29 线索二叉树的操作程序运行结果

【例9-2】 已知p中序线索二叉树中的一个非根结点，请编写一个不设栈删除以p为根的子树非递归算法。

【分析】 在中序线索二叉树中删除以p为根的子树，首先需要找到3个相关的结点，一个是p的双亲结点，以便将它的左孩子或右孩子指针置为空，来删除子树；一个是子树p的中序序列中第一个结点的前驱结点，以便修改它的后继线索；一个是子树p的中序序列中最后一个结点的后继结点，以便修改它的前驱线索。若p为其双亲结点的左孩子，则双亲结点为子树p的中序序列中最后一个结点的后继；若p为其双亲结点的右孩子，则双亲结点为子树p的中序序列中第一个结点的前驱。因此，可按下列步骤删除子树p。

①从结点p开始，沿结点的左指针向下查找，直到找到左指针为线索的结点，此结点即为子树p的中序序列的第一个结点。再由该结点的前驱线索找到前驱结点pre，若p为pre的右孩子，则将pre的右孩子指针置为空线索。

②从结点p开始，沿结点的右指针向下查找，直到找到右指针为线索的结点，此结点即为子树p的中序序列的最后一个结点。再由该结点的后继线索找到后继结点succ，若p为succ的左孩子，则将succ的左孩子指针置为空线索。

③若pre的右指针为线索，则将pre的右指针指向succ；若succ的左指针为线索，则将succ的左指针指向pre。

④释放子树p的结点。

算法描述如下。

```

void DelSubTree(BiThrTree p)
{
    BiThrTree pre, succ, q, r;
    q = p;
    while (q->ltag == Link)          /*
找到子树p
的中序序列的第一个结点q*/
        q = q->lchild;
    pre = q->lchild;                  /*
子树p
的中序序列的前驱保存在pre
中*/
    if (pre != NULL && pre->rtag == Link && pre->rchild == p)
        /*
若p
为pre
的右孩子，则将pre
的右指针置为空线索*/
        {
            pre->rtag = Thread;
            pre->rchild = NULL;
        }
    r = p;
    while (r->rtag == Link)          /*
找子树p
的中序序列的最后一个结点r*/
        r = r->rchild;
    succ = r->rchild; /*
子树p
的中序序列的后继保存在succ
中*/
    if (succ != NULL && succ->ltag == Link && succ->lchild == p)
        /*
若p
为pre
的左孩子，则将pre
的左指针置为空线索*/
        {
            succ->ltag = Thread;
            succ->lchild = NULL;
        }
    if (pre != NULL && pre->rtag == Thread)
        pre->rchild = succ;          /*
修改pre
的后继线索*/
    if (succ != NULL && succ->ltag == Thread)
        succ->lchild = pre;          /*
修改succ
的前驱线索*/
    r->rchild = NULL;
    while (q != NULL)                /*
从中序序列的结点q
开始，释放结点空间*/
    {
        r = q->rchild;
        if (q->rtag == Link)
            while (r->ltag == Link) /*
找出q
的后继保存在r
中*/
                r = r->rchild;
        free(q);                    /*
释放结点q
的空间*/
        q = r;
    }
}

```

9.7 树、森林与二叉树

树、森林和二叉树作为树的类型，它们之间是可以相互转换的。本节主要介绍森林与二叉树的相互转换、树与森林的遍历。

9.7.1 树转换为二叉树

树的孩子兄弟表示和二叉树的二叉链表在存储方式是相同的，也就是说，从它们的相同的物理结构可以得到一棵树，也可以得到一棵二叉树。树与二叉树的存储结构如图9-30所示。

树如何转换为二叉树呢？我们知道，一棵树的结点没有左右之分，而二叉树的结点有左右孩子之分。为了表述方便，约定树中的每一个孩子结点按照从左至右的顺序编号。例如，图9-32中结点a有3个孩子结点b、c和d，约定b为a的第1个孩子结点，c是a的第2个孩子结点，d是a的第3个孩子结点。

将一棵树转换为二叉树的步骤如下。

①加线。在所有兄弟结点之间加一条连线。

②去线。对树中的每个结点，只保留每个结点与它的第一个孩子结点间的连线，删除它与其他孩子结点的连线。

③调整。以树的根结点为轴心，将整棵树顺时针旋转一定的角度，使之结构层次分明。第一个孩子为二叉树中结点的左孩子，兄弟转换过来的孩子为右孩子。

图9-31 给出了树转换为二叉树的过程。

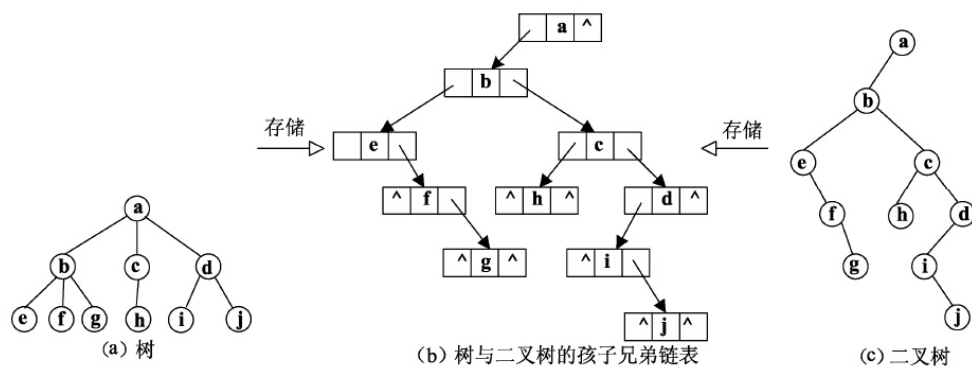


图9-30 树与二叉树的孩子兄弟链表

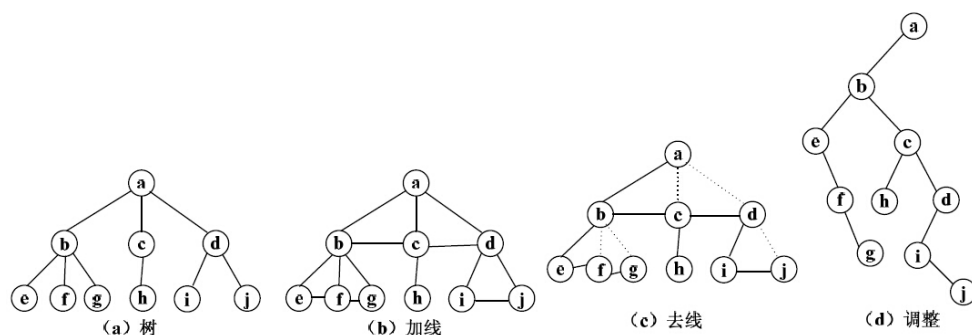


图9-31 将树转换为二叉树的过程

树转换为对应的二叉树后，树中每个结点的第1个孩子变为二叉树的左孩子结点，第2个孩子结点变为第1个孩子结点的右孩子结点，第3个孩子结点变为第2个孩子结点右孩子结点。

9.7.2 森林转换为二叉树

森林是若干棵树组成的集合。森林也可以转换为对应的二叉树，方法如下。

①把森林中的每棵树都转换为二叉树。

②第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子，用线连接起来。当所有的二叉树连接起来后就得到了由森林转换来的二叉树，最后进行相应的调整，使其层次分明。

森林转换为二叉树的过程如图9-32所示。

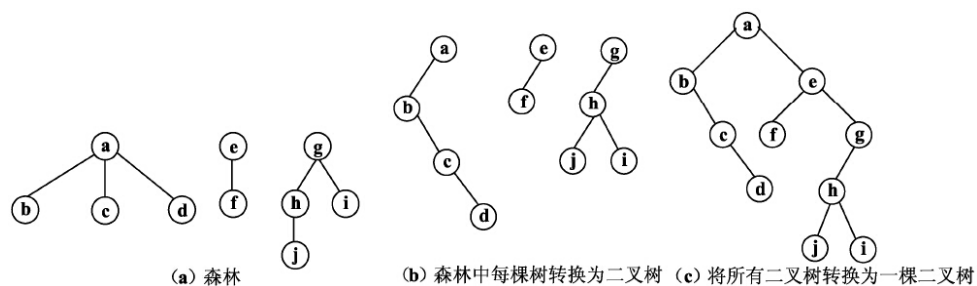


图9-32 森林转换为二叉树的过程

9.7.3 二叉树转换为树和森林

二叉树转换为树或者森林，就是将树和森林转换为二叉树的逆过程。把一棵二叉树转换为树的方法如下。

①加线。若某结点的左孩子结点存在，则将该结点的左孩子的右孩子结点、右孩子的右孩子结点……都与该结点用线条连接。

②去线。删除原二叉树中所有结点与右孩子结点的连线。

③调整。使结构层次分明。

一棵二叉树转换为树的过程如图9-33所示。

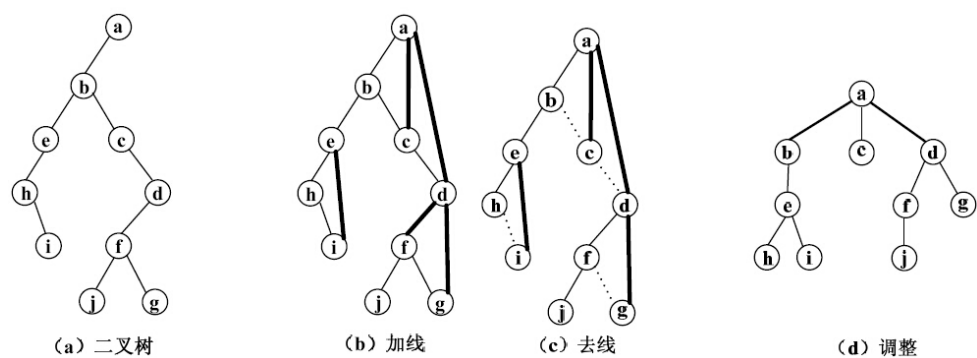


图9-33 二叉树转换为树的过程

与二叉树转换树的方法类似，二叉树转换为森林的过程如图9-34所示。

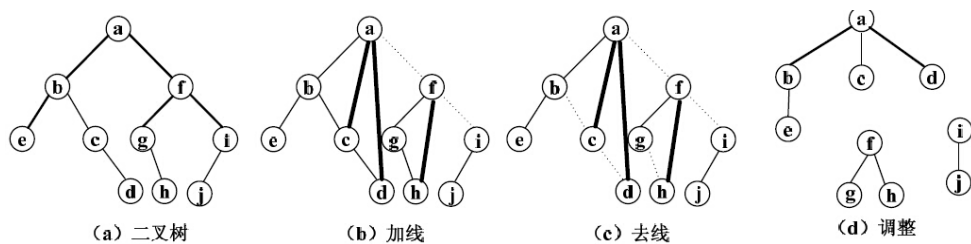


图9-34 二叉树转换为森林的过程

9.7.4 树和森林的遍历

与二叉树的遍历类似，树和森林的遍历也是按照某种规律对树或者森林中的每个结点进行访问，且仅访问一次的操作。

1. 树的遍历

通常情况下，按照访问树中根结点的先后次序，树的遍历方式分为先根遍历和后根遍历两种。先根遍历的步骤如下。

- ①访问根结点。
- ②按照从左到右的顺序依次先根遍历每一棵子树。

例如图9-33所示树先根遍历后得到的结点序列是a、b、e、f、g、c、h、d、i、j。

后根遍历的步骤如下。

- ①按照从左到右的顺序依次后根遍历每一棵子树。
- ②访问根结点。

例如，图9-33所示树后根遍历后得到的结点序列是e、f、g、b、h、c、i、j、d、a。

2. 森林的遍历

森林的遍历的方法有先序遍历和中序遍历两种。

先序遍历森林的方法如下。

- ①访问森林中第一棵树的根结点。
- ②先序遍历第一棵树的根结点的子树。
- ③先序遍历森林中剩余的树。

例如，图9-34所示森林先序遍历得到的结点序列是a、b、e、c、d、f、g、h、i、j。

中序遍历森林的方法如下。

- ①中序遍历第一棵树的根结点的子树。
- ②访问森林中第一棵树的根结点。
- ③中序遍历森林中剩余的树。

例如，图9-34所示森林中序遍历得到的结点序列是e、b、c、d、a、g、h、f、j、i。

从森林与二叉树之间的转换规则可知，当森林转换为二叉树时，其第一棵树的子树森林转换为左子树，剩余的树的森林转换为右子树。上述森林的先序和中序遍历即为其对应二叉树的先序和中序遍历。

9.7.5 树与二叉树应用举例

任何一棵二叉树只有唯一的先序、中序和后序序列，反过来，已知先序和中序序列、中序和后序序列、先序和后序序列，能唯一确定一棵二叉树吗？下面就来探讨这个问题。

1. 由先序序列和中序序列唯一确定一棵二叉树

先序遍历二叉树时，需要先访问根结点，然后先序遍历左子树，最后先序遍历右子树。因此，在二叉树的先序遍历过程中，根结点一定是第一个访问的结点。在中序遍历二叉树时，先中序遍历左子树，然后是根结点，最后遍历右子树。因此，在二叉树的中序序列中，根结点位于左右子树序列的中间，把序列分为两部分，左边序列为左子树结点，右边是右子树结点。

根据先序序列的左子树部分和中序序列左子树部分，左子树的根结点可继续将中序序列分为左子树和右子树两个部分，依次类推，就可以构造出二叉树。

设结点的先序序列为（A，B，C，D，E，F，G），中序序列为（B，D，C，A，F，E，G），图9-35给出了确定二叉树的过程。

在先序序列的第一个结点一定是根结点，故A为根结点，则A的左子树为{B，C}，右子树为{D，E，F，G}，在观察先序序列和中序序列，对于A的左子树来说，先序序列为B、C，中序序列为C、B，B一定是C的根结点，而C一定是B的左孩子，故现在就画出了A的左子树。

现在再看A的右子树，右子树先序序列为D、E、F、G，中序序列为E、F、D、G，则D一定是这棵子树的根结点，那么E、F就是D的左子树，G为D的右子树。再观察D的左子树先序序列和中序序列，E为F的根，F为右子树，这样就构造出了整棵二叉树。

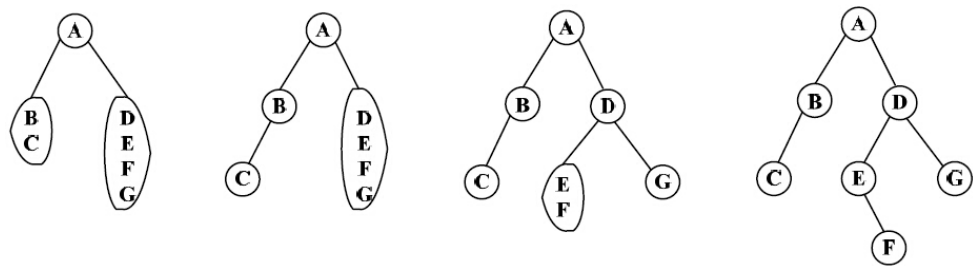


图9-35 由先序序列和中序序列确定的二叉树过程

由先序序列和中序序列构造二叉树的算法实现如下。

```
void CreateBiTree1(BiTree *T,char *pre,char *in,int len)
/*
由先序序列和中序序列构造二叉树*/
{
```

```

int k;
char *temp;
if (len <= 0)
{
    *T = NULL;
    return;
}
*T = (BitNode*) malloc (sizeof (BitNode)); /*
生成根结点*/
(*T) -> data = *pre;
for (temp = in; temp < in + len; temp++) /*
在中序序列in
中找到根结点所在的位置*/
    if (*pre == *temp)
        break;
k = temp - in; /*
左子树的长度*/
CreateBiTree1 (&((*T) -> lchild), pre + 1, in, k); /*
建立左子树*/
CreateBiTree1 (&((*T) -> rchild), pre + 1 + k, temp + 1, len - 1 - k); /*
建立右子树*/
}

```

2. 由中序序列和后序序列唯一确定一棵二叉树

由中序序列和后序序列也可以唯一确定一棵二叉树。后序遍历二叉树的顺序是先后序遍历左子树，接着后序遍历右子树，最后是访问根结点。因此，在二叉树的后序序列中，最后一个结点元素一定是根结点。在中序遍历二叉树的过程中，先中序遍历左子树，然后是根结点，最后遍历右子树。因此，在二叉树的中序序列中，根结点将中序序列分为左子树序列和右子树序列两部分。由中序序列的左子树结点个数，通过扫描后序序列，可以将后序序列分为左子树序列和右子树序列。依次类推，就可以构造出二叉树。

设结点的中序序列为 (C, E, B, D, A, H, G, I, F)，后序序列为 (E, C, D, B, H, I, G, F, A)，图9-36给出了唯一确定一棵二叉树的过程。

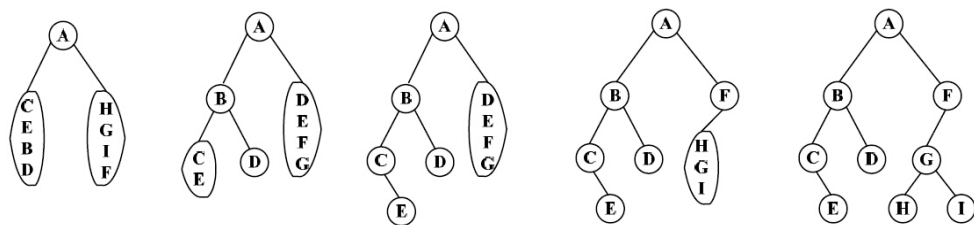


图9-36 由中序序列和后序序列确定二叉树的过程

由后序序列可知，A为二叉树的根结点，左子树为{C, E, B, D}，右子树为{H, G, I, F}，然后再观察A的左子树的中序序列和后序序列，从后序序列可知B为该子树的根结点，再由中序序列，{C, B}为B的左子树，D为B的右孩子，还按照上述方法，继续由B的左

子树的中序序列和后序序列可知，再根据中序序列得知，C为E的根结点，再由中序序列可知E为C的右子树，这样就构造出了A的左子树。

下面来构造A的右子树，因A的右子树中序序列为H、G、I、F，后序序列为H、I、G、F，则A的右子树的根结点为F，即后序序列的最后一个结点，再根据其左子树序列，左子树为{H，G，I}，F没有右子树。然后再根据F的左子树的后序序列H，I，G可知，F左子树根结点为G，再由中序序列可知H为G的左孩子，I为G的右孩子。这样就构造出了A的右子树。

由中序序列和后序序列构造二叉树的算法如下。

```
void CreateBiTree2(BiTree *T,char *in,char *post,int len)
/*
由中序序列和后序序列构造二叉树*/
{
    int k;
    char *temp;
    if(len<=0)
    {
        *T=NULL;
        return;
    }
    for(temp=in;temp<in+len;temp++) /*
在中序序列in
中找到根结点所在的位置*/
        if(*(post+len-1)==*temp)
        {
            k=temp-in;
            /*
左子树的长度*/
            (*T)=(BitNode*)malloc(sizeof(BitNode));
            (*T)->data =*temp;
            break;
        }
    CreateBiTree2(&((*T)->lchild),in,post,k);
    /*
建立左子树*/
    CreateBiTree2(&((*T)->rchild),in+k+1,post+k,len-k-1); /*
建立右子树*/
}
```

3. 由先序序列和后序序列不能唯一确定二叉树

那么，给定先序序列和后序序列可以唯一确定一棵二叉树吗？答案是不能。假设有一个先序序列为（A，B，C），一个后序序列为（C，B，A），可以构造出两棵树，如图9-37所示。



图9-37 由先序序列和后序序列确定的两棵二叉树

由此可知，给定先序序列和后序序列不能唯一确定二叉树。

4. 程序举例

【例9-3】 编写算法，已知先序序列（E，B，A，C，F，H，G，I，K，J）和中序序列（A，B，C，D，E，F，G，H，I，J，K）、中序序列（A，B，C，D，E，F，G，H，I，J，K）和后序序列（A，C，D，B，G，J，K，I，H，F，E），构造一棵二叉树。

```

#include"stdio.h"
#include"stdlib.h"
#include"string.h"
#include<conio.h>
#define MaxSize 100
/*
二叉树类型定义*/
typedef struct Node
{
    char data;
    struct Node * lchild,*rchild;
}BitNode,*BiTree;
/*
函数声明*/
void CreateBiTree1(BiTree *T,char *pre,char *in,int len);
void CreateBiTree2(BiTree *T,char *in,char *post,int len);
void Visit(BiTree T,BiTree pre,char e,int i);
void PrintLevel(BiTree T);
void PrintTLR(BiTree T);
void PrintLRT(BiTree T);
void PrintLevel(BiTree T);
void main()
{
    BiTree T,ptr=NULL;
    char ch;
    int len;
    char pre[MaxSize],in[MaxSize],post[MaxSize];
    T=NULL;
    /*
    由中序序列和后序序列构造二叉树*/
    printf("
    由先序序列和中序序列构造二叉树: \n");
    printf("
    请你输入先序的字符串序列: ");
    gets(pre);
    printf("
    请你输入中序的字符串序列: ");
    gets(in);
    len=strlen(pre);
    CreateBiTree1(&T,pre,in,len);
    /*
    后序和层次输出二叉树的结点*/
    printf("
    你建立的二叉树后序遍历结果是: \n");

```

```

        PrintLRT(T);
        printf("\n\n");
你建立的二叉树层次遍历结果是: \n";
        PrintLevel(T);
        printf("\n\n");
        printf("
请你输入你要访问的结点: ");
        ch=getchar();getchar();
        Visit(T,ptr,ch,1);
        /*
由中序序列和后序序列构造二叉树*/
        printf("
由先序序列和中序序列构造二叉树: \n");
        printf("
请你输入中序的字符串序列: ");
        gets(in);
        printf("
请你输入后序的字符串序列: ");
        gets(post);
        len=strlen(post);
        CreateBiTree2(&T,in,post,len);
        /*
先序和层次输出二叉树的结点*/
        printf("\n\n");
你建立的二叉树先序遍历结果是: \n";
        PrintTLR(T);
        printf("\n\n");
你建立的二叉树层次遍历结果是: \n";
        PrintLevel(T);
        printf("\n\n");
        printf("
请你输入你要访问的结点: ");
        ch=getchar();getchar();
        Visit(T,ptr,ch,1);
    }
void PrintLevel(BiTree T)
/*
按层次输出二叉树的结点*/
{
    BiTree Queue[MaxSize];
    int front,rear;
    if(T==NULL)
        return;
    front=-1;
    /*
初始化队列*/
    rear=0;
    Queue[rear]=T;
    while(front!=rear)
    /*
如果队列不为空*/
    {
        front++;
        /*
将队头元素出队*/
        printf("%4c",Queue[front]->data);
        /*
输出队头元素*/
        if(Queue[front]->lchild!=NULL)
            /*
如果队头元素的左孩子结点不为空,则将左孩子入队*/
            {
                rear++;
                Queue[rear]=Queue[front]->lchild;
            }
        if(Queue[front]->rchild!=NULL)
            /*
如果队头元素的右孩子结点不为空,则将右孩子入队*/
            {
                rear++;
                Queue[rear]=Queue[front]->rchild;
            }
    }
}
void PrintTLR(BiTree T)
/*
先序输出二叉树的结点*/
{
    if(T!=NULL)
    {
        printf("%4c ",T->data);
        /*
输出根结点*/
        PrintTLR(T->lchild);
        /*
先序遍历左子树*/
        PrintTLR(T->rchild);
        /*
先序遍历右子树*/
    }
}
void PrintLRT(BiTree T)
/*
后序输出二叉树的结点*/

```

```

{
    if (T!=NULL)
    {
        PrintLRT(T->lchild);          /*
先序遍历左子树*/
        PrintLRT(T->rchild);          /*
先序遍历右子树*/
        printf("%4c",T->data);        /*
输出根结点*/
    }
}
void Visit(BiTree T,BiTree pre,char e,int i)
/*
访问结点e*/
{
    if(T==NULL&&pre==NULL)
    {
        printf("\n
对不起！你还没有建立二叉树，先建立再进行访问！\n");
        return;
    }
    if(T==NULL)
        return;
    else if (T->data==e)                /*
如果找到结点e
，则输出结点的双亲结点*/
    {
        if(pre!=NULL)
        {
            printf("%2c
的双亲结点是:%2c\n",e,pre->data);
            printf("%2c
结点在%2d
层上\n",e,i);
        }
        else
            printf("%2c
位于第1
层，无双亲结点！\n",e);
    }
    else
    {
        Visit(T->lchild,T,e,i+1);      /*
遍历左子树*/
        Visit(T->rchild,T,e,i+1);      /*
遍历右子树*/
    }
}

```

程序的运行结果如图9-38所示。

从程序结果上看，两次输入输出的序列其实是一棵二叉树，这也印证了程序的正确性。由先序序列和中序序列确定的二叉树如图9-39所示。

```

D:\零基础学数据结构\例9_4\Debug\例9_4.exe
由先序序列和中序序列构造二叉树:
请你输入先序的字符串序列: EBADCFHGIKJ
请你输入中序的字符串序列: ABCDEFGHIJK
你建立的二叉树后序遍历结果是:
A C D B G J K I H F E
你建立的二叉树层次遍历结果是:
E B F A D H C G I K J
请你输入你要访问的结点: H
H的双亲结点是: F
H结点在 3层上
由先序序列和中序序列构造二叉树:
请你输入中序的字符串序列: ABCDEFGHIJK
请你输入后序的字符串序列: ACDBGJKIHFE
你建立的二叉树先序遍历结果是:
E B A D C F H G I K J
你建立的二叉树层次遍历结果是:
E B F A D H C G I K J
请你输入你要访问的结点: K
K的双亲结点是: I
K结点在 5层上
Press any key to continue

```

图9-38 由给定序列构造二叉树运行结果

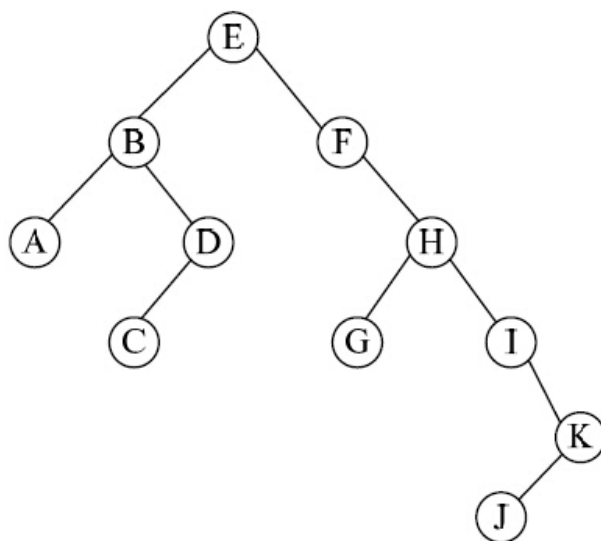


图9-39 由先序序列和中序序列确定的二叉树

【例9-4】 一棵树可用广义表形式表示，如图9-40所示的一棵树可表示为A（B（D，E，F），C（G，H）），请编写算法对以孩子链表表示的树输出其广义表的表示形式。

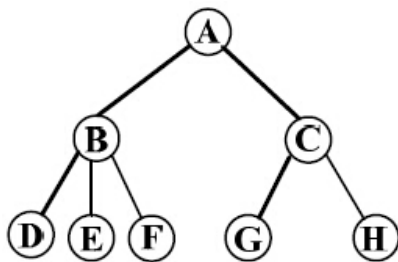


图9-40 树

【分析】 在树的广义表形式中，如果不考虑括号和逗号，那么结点是按先根遍历序列排列的，因此，可以对以孩子兄弟链表表示的树进行先根遍历，并在输出各子树的结点时加上括号。

当树不为空时，输出根结点；若根结点有孩子，则先输出左括号“（”，再依次输出各子树的广义表形式，并用逗号隔开，最后输出右括号“）”。显然，这是一个递归的过程，算法描述如下。

```
void ListTree(CSTree T)
{
    CSTree p;
    if (T==NULL)
        return;
    printf("%c",T->data);          /*
输出根结点*/
    p=T->firstchild;
    if (p!=NULL)                  /*
若根结点有孩子*/
    {
        printf("(");            /*
输出左括号*/
        ListTree(p);
        p=p->nextsibling;
        while (p!=NULL)
        {
            printf(",");        /*
输出逗号，以分隔各子树*/
            ListTree(p);        /*
输出下一个子表的广义表形式*/
            p=p->nextsibling;
        }
        printf(")");           /*
输出右括号*/
    }
}
```

【例9-5】 编写算法，创建一棵树，按照层次输出这棵树的结点，通过输入要修改的结点元素值修改结点元素值，最后插在这棵树中指定位置插入一棵子树，并输出这棵树的信息。

【分析】 本题采用双亲表示法顺序结构存储树，结点结构类型定义如下。

```
#define MAXSIZE 100
typedef struct
{
    TElemType data;
    int parent; /*
双亲位置域 */
} PTNode;
typedef struct
{
    PTNode nodes[MAXSIZE];
    int n; /*
结点数 */
} PTree;
```

存储结构如图9-41所示。

对于如图9-42（a）所示的树，其存储表示如图9-42（b）所示。

在创建树的过程中，按层次顺序创建树，每读入一个字符，将其存放在树的结点数组中，同时将该结点信息入队列，以备以后添加孩子结点信息。每为一个结点增加完孩子结点，就将队头结点出队，为此结点添加孩子结点信息，相应代码如下。

```
while (i<MAXSIZE&&!QueueEmpty(q)) /*
数组未满足且队不空 */
{
    DeQueue (&q,&qq); /*
出队一个结点 */
    printf("
请按长幼顺序输入结点%c
的所有孩子: ",qq.name);
    gets(c);
    l=strlen(c);
    for (j=0;j<l;j++)
    {
        (*T).nodes[i].data=c[j];
        (*T).nodes[i].parent=qq.num;
        p.name=c[j];
        p.num=i;
        EnQueue (&q,p);
        /*
该结点入队 */
        i++;
    }
}
```

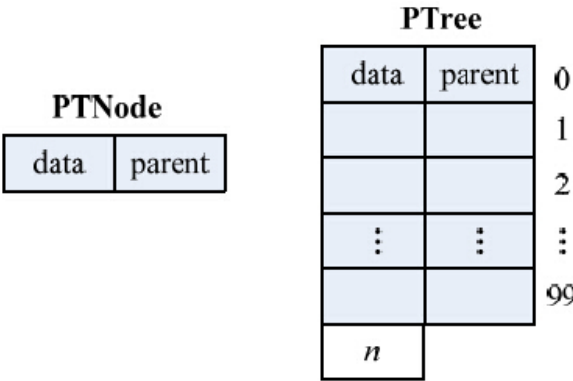
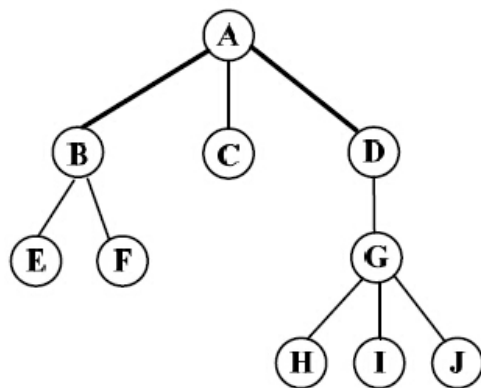


图9-41 双亲表示法树的结点结构



(a) 树

A	-1	0
B	0	1
C	0	2
D	0	3
E	1	4
F	1	5
G	3	6
H	6	7
I	6	8
J	6	9
⋮	⋮	⋮
		99
10		

(b) 树的存储结构

图9-42 树及存储结构

完整程序代码如下。

```

#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<string.h>
#define MAXSIZE 100
/*
定义队列元素类型 */
typedef char TElemType;
typedef struct
{
    int num;
    TElemType name;
}QElemType;
#include"LinkQueue.h"
TElemType Nil=' '; /*
以空格符表示空 */
/*
树的双亲表存储表示*/
typedef struct
{
    TElemType data;
    int parent; /*
双亲位置域 */
} PTNode;
typedef struct
{
    PTNode nodes[MAXSIZE];
    int n; /*
结点数 */
} PTree;
#define ClearTree InitTree /*
二者操作相同 */
#define DestroyTree InitTree /*
二者操作相同 */

```

```

void InitTree(PTree *T)
/*
操作结果: 构造空树T */
{
    (*T).n=0;
}
void CreateTree(PTree *T)
/*
操作结果: 构造树T */
{
    LinkQueue q;
    QElemType p,qq;
    int i=1,j,l;
    char c[MAXSIZE]; /*
临时存放孩子结点数组 */
    InitQueue(&q); /*
初始化队列 */
    printf("
请输入根结点(
字符型,
空格为空格): ");
    scanf("%c%c",&(*T).nodes[0].data); /*
根结点序号为0
, %*c
吃掉回车符 */
    if((*T).nodes[0].data!=Nil) /*
非空树 */
    {
        (*T).nodes[0].parent=-1; /*
根结点无双亲 */
        qq.name=(*T).nodes[0].data;
        qq.num=0;
        EnQueue(&q,qq); /*
该结点入队 */
        while(i<MAXSIZE&&!QueueEmpty(q)) /*
数组未满且队不空 */
        {
            DeQueue(&q,&qq); /*
出队一个结点 */
            printf("
请按长幼顺序输入结点%c
的所有孩子: ",qq.name);
            gets(c);
            l=strlen(c);
            for(j=0;j<l;j++)
            {
                (*T).nodes[i].data=c[j];
                (*T).nodes[i].parent=qq.num;
                p.name=c[j];
                p.num=i;
                EnQueue(&q,p); /*
该结点入队 */
                i++;
            }
            if(i>MAXSIZE)
            {
                printf("
结点数超过数组最大容量.\n");
                exit(-1);
            }
            (*T).n=i;
        }
        else
            (*T).n=0;
    }
}
int TreeEmpty(PTree T)
/*
初始条件: 树T
存在。操作结果: 若T
为空树, 则返回1
, 否则返回0*/
{
    if(T.n)
        return 0;
    else
        return 1;
}
int TreeDepth(PTree T)
/*
初始条件: 树T
存在。操作结果: 返回T
的深度 */
{
    int k,m,def,max=0;

```



```

        for (k=0;k<T.n;++k)
        {
            def=1; /*
初始化本结点的深度 */
            m=T.nodes[k].parent;
            while (m!=-1)
            {
                m=T.nodes[m].parent;
                def++;
            }
            if (max<def)
                max=def;
        }
        return max; /*
最大深度 */
    }
TElemType Root(PTree T)
/*
初始条件: 树T
存在。操作结果: 返回T
的根 */
{
    int i;
    for (i=0;i<T.n;i++)
        if (T.nodes[i].parent<0)
            return T.nodes[i].data;
    return Nil;
}
TElemType Value(PTree T,int i)
/*
初始条件: 树T
存在, i
是树T
中结点的序号。操作结果: 返回第i
个结点的值 */
{
    if (i<T.n)
        return T.nodes[i].data;
    else
        return Nil;
}
int Assign(PTree *T,TElemType cur_e,TElemType value)
/*
初始条件: 树T
存在, cur_e
是树T
中结点的值。操作结果: 改cur_e
为value */
{
    int j;
    for (j=0;j<(*T).n;j++)
    {
        if ((*T).nodes[j].data==cur_e)
        {
            (*T).nodes[j].data=value;
            return 1;
        }
    }
    return 0;
}
TElemType Parent(PTree T,TElemType 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 " */
    int j;
    for (j=1;j<T.n;j++) /*
根结点序号为0 */
        if (T.nodes[j].data==cur_e)
            return T.nodes[T.nodes[j].parent].data;
    return Nil;
}
TElemType LeftChild(PTree T,TElemType 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 " */
        int i,j;
        for (i=0;i<T.n;i++)
            if (T.nodes[i].data==cur_e) /*
找到cur_e
, 其序号为i */
                break;
        for (j=i+1;j<T.n;j++) /*
根据树的构造函数, 孩子的序号>
其双亲的序号 */
            if (T.nodes[j].parent==i) /*
根据树的构造函数, 最左孩子(
长子)
的序号<
其他孩子的序号 */
                return T.nodes[j].data;
        return Nil;
}
TElemType RightSibling(PTree T,TElemType 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(
下一个)
兄弟, 则返回它的右兄弟, 否则返回 " 空 " */
{
    int i;
    for (i=0;i<T.n;i++)
        if (T.nodes[i].data==cur_e) /*
找到cur_e
, 其序号为i */
            break;
    if (T.nodes[i+1].parent==T.nodes[i].parent)
        /*
根据树的构造函数, 若cur_e
有右兄弟的话则右兄弟紧接其后 */
        return T.nodes[i+1].data;
    return Nil;
}
void Print(PTree T)
/*
输出树T */
{
    int i;
    printf("
结点个数=%d\n",T.n);
    printf("
结点
双亲\n");
    for (i=0;i<T.n;i++)
    {
        printf("      %c",Value(T,i)); /*
结点 */
        if (T.nodes[i].parent>=0) /*
有双亲 */
            printf("      %c",Value(T,T.nodes[i].parent)); /*
双亲 */
        printf("\n");
    }
}
int InsertChild(PTree *T,TElemType p,int i,PTree c)
/*
初始条件: 树T
存在, p
是T
中某个结点, 1
≤i
≤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
结点的第i
棵子树 */
{
    int j,k,l,f=1,n=0; /*

```

```

    设交换标志f
    的初值为1
    , p
    的孩子数n
    的初值为0 */
    PTNode t;
    if(!TreeEmpty(*T)) /* T
    不空 */
    {
        for(j=0;j<(*T).n;j++) /*
        在T
        中找p
        的序号 */
            if((*T).nodes[j].data==p) /* p
            的序号为j */
                break;
            l=j+1; /*
            如果c
            是p
            的第1
            棵子树, 则插在j+1
            处 */
            if(i>l) /* c
            不是p
            的第1
            棵子树 */
            {
                for(k=j+1;k<(*T).n;k++) /*
                从j+1
                开始找p
                的前i-1
                个孩子 */
                    if((*T).nodes[k].parent==j) /*
                    当前结点是p
                    的孩子 */
                    {
                        n++; /*
                        孩子数加1 */
                        if(n==i-1) /*
                        找到p
                        的第i-1
                        个孩子, 其序号为k1 */
                            break;
                    }
                    l=k+1; /* c
                    插在k+1
                    处 */
                } /* p
                的序号为j
                , c
                插在l
                处 */
                if(l<(*T).n) /*
                插入点l
                不在最后 */
                for(k=(*T).n-1;k>=l;k--) /*
                依次将序号l
                以后的结点向后移c.n
                个位置 */
                {
                    (*T).nodes[k+c.n]=(*T).nodes[k];
                    if((*T).nodes[k].parent>=l)
                        (*T).nodes[k+c.n].parent+=c.n;
                }
                for(k=0;k<c.n;k++)
                {
                    (*T).nodes[l+k].data=c.nodes[k].data; /*
                    依次将树c
                    的所有结点插于此处 */
                    (*T).nodes[l+k].parent=c.nodes[k].parent+l;
                }
                (*T).nodes[l].parent=j; /*
                树c
                的根结点的双亲为p */
                (*T).n+=c.n; /*
                树T
                的结点数加c.n
                个 */
            }
        while(f)
        { /*

```

从插入点之后，将结点仍按层序排列 */

交换标志置0 */

如果结点j
的双亲排在结点j+1
的双亲之后（
树没有按层序排列）
，交换两结点*/

交换标志置1 */

改变双亲序号 */

双亲序号改为j+1 */

双亲序号改为j */

树T
不存在 */

void TraverseTree(PTree T,void(*Visit)(TElemType))
/*

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：层序遍历树T，
对每个结点调用函数Visit
一次且仅一次 */
{

```
    int i;  
    for(i=0;i<T.n;i++)  
        Visit(T.nodes[i].data);  
    printf("\n");  
}
```

```
void vi(TElemType c)  
{  
    printf("%c ",c);  
}
```

```
void main()  
{
```

```
    int i;  
    PTree T,p;  
    TElemType e,e1;  
    InitTree(&T);  
    CreateTree(&T);  
    printf("构造树T  
后，  
树空否? %d(1:  
是 0:  
否)  
树根为%c  
树的深度为%d\n",  
TreeEmpty(T),Root(T),TreeDepth(T));  
printf("层序遍历树T:\n");  
TraverseTree(T,vi);  
printf("请输入待修改的结点的值  
新值: ");  
scanf("%c%c%c%c", &e, &e1);  
Assign(&T,e,e1);  
printf("层序遍历修改后的树T:\n");  
TraverseTree(T,vi);
```

```
f=0; /*
```

```
for(j=1;j<(*T).n-1;j++)  
    if((*T).nodes[j].parent>(*T).nodes[j+1].parent)  
        {/*
```

```
        t=(*T).nodes[j];  
        (*T).nodes[j]=(*T).nodes[j+1];  
        (*T).nodes[j+1]=t;  
        f=1; /*
```

```
for(k=j;k<(*T).n;k++) /*
```

```
    if((*T).nodes[k].parent==j)  
        (*T).nodes[k].parent++; /*
```

```
    else if((*T).nodes[k].parent==j+1)  
        (*T).nodes[k].parent--; /*
```

```
}
```

```
    }  
    return 1;
```

```
    }  
    else /*
```

```
        return 0;
```

```

        printf("%c
的双亲是%c,
长子是%c,
下一个兄弟是%c\n",e1,Parent(T,e1),LeftChild(T,e1),RightSibling(T,e1));
        printf("
建树p:\n");
        InitTree(&p);
        CreateTree(&p);
        printf("
层序遍历树p:\n");
        TraverseTree(p,vi);
        printf("
将树p
插到树T
中, 请输入T
中p
的双亲结点
子树序号: ");
        scanf("%c%d%c",&e,&i);
        InsertChild(&T,e,i,p);
        Print(T);
}

```

程序运行结果如图9-43所示。

```

D:\零基础学数据结构\例9_5\Debug\例9_5.exe
请输入根结点<字符型,空格为空>: A
请按长幼顺序输入结点A的所有孩子: BCD
请按长幼顺序输入结点B的所有孩子: EF
请按长幼顺序输入结点C的所有孩子:
请按长幼顺序输入结点D的所有孩子: G
请按长幼顺序输入结点E的所有孩子:
请按长幼顺序输入结点F的所有孩子:
请按长幼顺序输入结点G的所有孩子: HIJ
请按长幼顺序输入结点H的所有孩子:
请按长幼顺序输入结点I的所有孩子:
请按长幼顺序输入结点J的所有孩子:
构造树T后,树空否? 0(1:是 0:否) 树根为A 树的深度为4
层序遍历树T:
A B C D E F G H I J
请输入待修改的结点的值 新值: G M
层序遍历修改后的树T:
A B C D E F M H I J
M的双亲是D,长子是H,下一个兄弟是
建立树p:
请输入根结点<字符型,空格为空>: R
请按长幼顺序输入结点R的所有孩子: xyz
请按长幼顺序输入结点x的所有孩子:
请按长幼顺序输入结点y的所有孩子:
请按长幼顺序输入结点z的所有孩子:
层序遍历树p:
R x y z
将树p插到树T中,请输入I中p的双亲结点 子树序号: D 2
结点个数=14
结点 双亲
A
B A
C A
D A
E B
F B
M D
R D
H M
I M
J M
x R
y R
z R
Press any key to continue

```

图9-43 树的相关操作程序运行结果

9.8 综合案例：哈夫曼树

哈夫曼（Huffman）树又称**最优二叉树**。它是一种带权路径长度最短的树，应用非常广泛。本节主要介绍什么是哈夫曼树、哈夫曼编码及哈夫曼编码算法的实现。

9.8.1 什么是哈夫曼树

在介绍什么是哈夫曼树之前，先来了解以下几个概念。

1. 路径和路径长度

路径是指在树中从一个结点到另一个结点所走过的路程。路径长度是一个结点到另一个结点之间的分支数目。树的路径长度是指从树的树根到每一个结点的路径长度的和。

2. 树的带权路径长度

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 $WPL = \sum_{i=1}^n w_i \times l_i$ 其中， n 是树中叶子结点的个数， w_i 是第 i 个叶子结点的权值， l_i 是第 i 个叶子结点的路径长度。

例如，图9-44所示的二叉树的带权路径长度分别是
 $WPL=7\times 2+5\times 2+2\times 2+4\times 2=36$ 、 $WPL=4\times 2+7\times 3+5\times 3+2\times 1=46$ 、
 $WPL=7\times 1+5\times 2+2\times 3+4\times 3=35$ ，因此，第3棵树的带权路径长度最小，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最小，它其实就是一棵哈夫曼树。

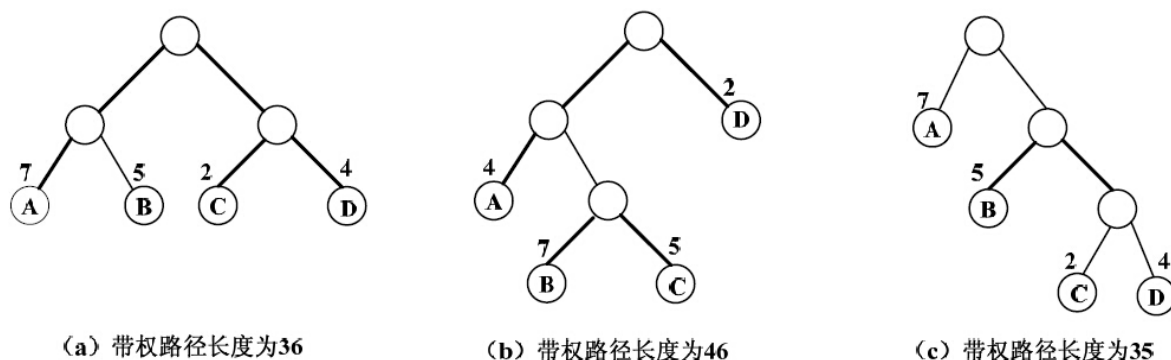


图9-44 二叉树的带权路径长度

3. 哈夫曼树

哈夫曼树就是带权路径长度最小的树，权值最小的结点远离根结点，权值越大的结点越靠近根结点。

哈夫曼树的构造算法如下。

①由给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 构成 n 棵只有根结点的二叉树集合 $F=\{T_1, T_2, \dots, T_n\}$ ，其中每棵二叉树 T_i 中只有一个带权为 w_i 的根结点，其左右子树均为空。

②在二叉树集合F中选取两棵根结点的权值最小和次小的树作为左、右子树构造一棵新的二叉树，新二叉树的根结点的权重为这两棵子树根结点的权重之和。

③在二叉树集合F中删除这两棵二叉树，并将新得到的二叉树加入到集合F中。

④重复步骤②和③，直到集合F中只剩下一棵二叉树为止。这颗二叉树就是哈夫曼树。

例如，假设给定一组权值{2, 3, 6, 8}，按照哈夫曼构造的算法对集合的权重构造哈夫曼树的过程如图9-45所示。

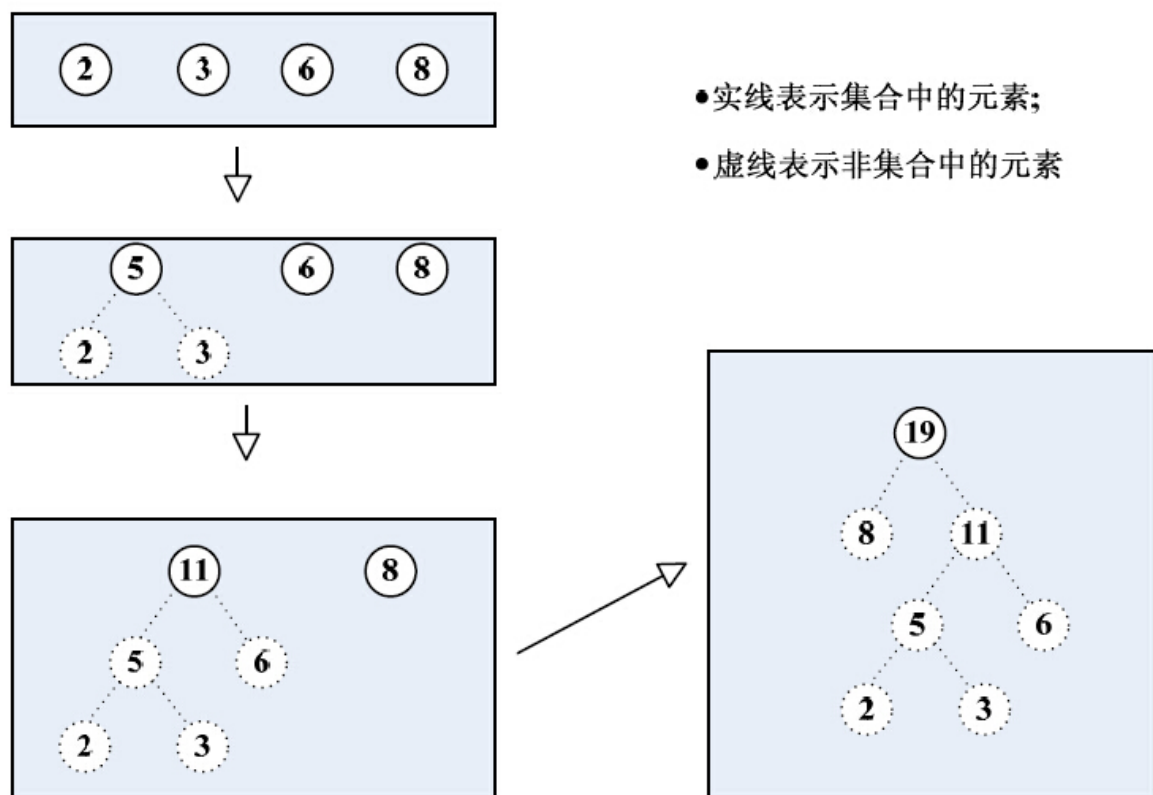


图9-45 哈夫曼构造过程

9.8.2 哈夫曼编码

在电报的传输过程中，需将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“ABACCDA”，它只有4种字符，只需两个字符的串就可分辨。假设A、B、C、D的编码分别为00、01、10、11，则上述7个字符的电文便为“00010010101100”，总长14位，对方接收后可按两位分隔进行译码。

当然，在传送电文时，希望电文的长度尽可能短。如果按照每个字符进行长度不等进行编码，将出现频率高的字符采用尽可能短的编码，则电文的代码长度就会减少。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为总长为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个字符子串“0000”就可能有多种译法，可能是“AAAA”，也可能是“ABA”，也可能是“BB”，因此，所设计的长短不等的编码必须满足任一个字符的编码都不是另一个字符的编码的前缀的要求，这样的编码称为前缀编码。

二叉树可以用来设计二进制的前缀编码。假设如图9-47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，且约定的左孩子分支为0，右孩子分支为1，从根结点到每个叶子结点经过的分支组成的0和1序列就是结点的前缀编码。字符a的编码为0，字符b的编码为110，字符c的编码为111，字符d的编码为10。

那又如何得到使电文长度最短的二进制前缀编码呢？具体构造方法如下。

假设需要编码的字符集合为 $\{c_1, c_2, \dots, c_n\}$ ，相应地，字符在电文中的出现次数为 $\{w_1, w_2, \dots, w_n\}$ ，以字符 c_1 、 c_2 、 \dots 、 c_n 作为叶子结点，以 w_1 、 w_2 、 \dots 、 w_n 为对应叶子结点的权值构造一棵二叉树，按照以上构造方法，字符集合为 $\{a, b, c, d\}$ ，各个字符相应的出现次数为 $\{4, 1, 1, 2\}$ ，这些字符作为叶子结点构成的哈夫曼树如图9-46所示。

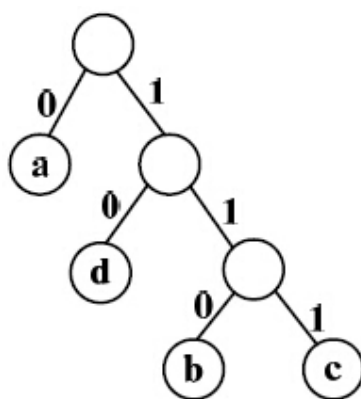


图9-46 哈夫曼树

因此，可以得到电文abdaacda的哈夫曼编码为01101000111100，共13个二进制字符。这样就保证了电文的编码达到最短。

9.8.3 哈夫曼编码算法的实现

【例9-6】 假设一个字符序列为 $\{a, b, c, d\}$ ，对应的权重为 $\{2, 3, 6, 8\}$ 。试构造一棵哈夫曼树，然后输出相应的哈夫曼编码。

【分析】 哈夫曼树的类型定义如下。

```
typedef struct                                /*
哈夫曼树类型定义*/
{
    unsigned int weight;
    unsigned int parent,lchild,rchild;
}HTNode,*HuffmanTree;
typedef char **HuffmanCode;    /*
存放哈夫曼编码*/
```

HuffmanCode为一个二级指针，相当于二维数组，用来存放每一个叶子结点的哈夫曼编码。初始时，将每一个叶子结点的双亲结点域、左孩子域和右孩子域初始化为0。若有n个叶子结点，则非叶子结点有n-1个，所以总共结点数目是2n-1个。同时也要将剩下的n-1个双亲结点域初始化为0，这主要是为了查找权值最小的结点方便。

①创建哈夫曼树，构造哈夫曼编码。

依次选择两个权值最小的结点s1和s2分别作为左子树结点和右子树结点，并为其双亲结点赋予一个地址，双亲结点的权值为s1和s2的权值之和。修改它们的parent域，使它们指向同一个双亲结点，双亲结点的左子树为权值最小的结点，右子树为权值次小的结点。重复执行这种操作n-1次，即求出n-1个非叶子结点的权值。这样就构造出了一棵哈夫曼树。代码如下。

```
/*
构造哈夫曼树HT*/
for(i=n+1;i<=m;i++)
{
    Select(HT,i-1,&s1,&s2);
    (*HT)[s1].parent=(*HT)[s2].parent=i;
    (*HT)[i].lchild=s1;
    (*HT)[i].rchild=s2;
    (*HT)[i].weight=(*HT)[s1].weight+(*HT)[s2].weight;
}
```

求哈夫曼编码的方式有两种，即从根结点开始到叶子结点正向求哈夫曼编码和从叶子结点到根结点逆向求哈夫曼编码，这里只给出从根结点出发到叶子结点求哈夫曼编码的算法，其算法思想为，从编号为 $2n-1$ 的结点开始，即根结点开始，依次通过判断左孩子和右孩子是否存在进行编码，若左孩子存在则编码为0，若右孩子存在则编码为1；同时，利用weight域作为结点是否已经访问的标志位，若左孩子结点已经访问则将相应的weight域置为1，若右孩子结点也已经访问过则将相应的weight域置为2，若左孩子和右孩子都已经访问过则回退至双亲结点。按照这个思路，直到所有结点都已经访问过，并回退至根结点，算法结束。

从根结点到叶子结点求哈夫曼编码的算法实现如下。

```
void HuffmanCoding(HuffmanTree *HT,HuffmanCode *HC,int *w,int n)
/*
构造哈夫曼树HT
，并从根结点到叶子结点求哈夫曼编码并保存在HC
中*/
{
    int s1,s2,i,m;
    unsigned int r,cdlen;
    char *cd;
    HuffmanTree p;
    if(n<=1)
        return;
    m=2*n-1;
    *HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode));
    for(p=*HT+1,i=1;i<=n;i++,p++,w++)
    {
        (*p).weight=*w;
        (*p).parent=0;
        (*p).lchild=0;
        (*p).rchild=0;
    }
    for(;i<=m;++i,++p)
        (*p).parent=0;
    /*
构造哈夫曼树HT*/
    for(i=n+1;i<=m;i++)
    {
        Select(HT,i-1,&s1,&s2);
        (*HT)[s1].parent=(*HT)[s2].parent=i;
        (*HT)[i].lchild=s1;
        (*HT)[i].rchild=s2;
        (*HT)[i].weight=(*HT)[s1].weight+(*HT)[s2].weight;
    }
}
/*
从根结点到叶子结点求哈夫曼编码并保存在HC
中*/
```

```

中*/
        *HC=(HuffmanCode)malloc((n+1)*sizeof(char*));
        cd=(char*)malloc(n*sizeof(char));
        r=m;
从根结点开始*/
        cdlen=0;
编码长度初始化为0*/
        for(i=1;i<=m;i++)
            (*HT)[i].weight=0;
将weight
域作为状态标志*/
        while(r)
        {
            if((*HT)[r].weight==0)
            {
                if((*HT)[r].lchild!=0)
                {
                    r=(*HT)[r].lchild;
                    cd[cdlen++]='0';
                }
                else if((*HT)[r].rchild==0)
                {
                    if((*HT)[r].weight==1)
                    {
                        (*HC)[r]=(char *)malloc((cdlen+1)*sizeof(char));
                        cd[cdlen]='\0';
                        strcpy((*HC)[r],cd);
                    }
                    else if((*HT)[r].weight==1)
                    {
                        (*HT)[r].weight=2;
                        if((*HT)[r].rchild!=0)
                        {
                            r=(*HT)[r].rchild;
                            cd[cdlen++]='1';
                        }
                        else
                        {
                            if左孩子结点和右孩子结点都已经访问过,则退回到双亲结点*/
                            {
                                r=(*HT)[r].parent;
                                --cdlen;
                                编码长度减1*/
                            }
                        }
                    }
                    free(cd);
                }
            }
        }
    }
}

```

②查找权值最小和次小的两个结点，代码如下。

```

int Min(HuffmanTree t,int n)
/*
返回树中n
个结点中权值最小的结点序号*/
{
    int i,flag;
    int f=infinity;
    ...
}

```

```

为一个无限大的值*/
        for(i=1;i<=n;i++)
            if(t[i].weight<f&&f==0)
                f=t[i].weight,flag=i;
        t[flag].parent=1;
给选中的结点的双亲结点赋值1
, 避免再次查找该结点*/
        return flag;
    }
void Select(HuffmanTree *t,int n,int *s1,int *s2)
/*
在n
个结点中选择两个权值最小的结点序号, 其中s1
最小, s2
次小*/
{
    int x;
    *s1=Min(*t,n);
    *s2=Min(*t,n);
    if((*t)[*s1].weight>(*t)[*s2].weight)/*
若序号s1
的权值大于s2
的权值, 将两者交换, 使s1
最小, s2
次小*/
    {
        x=*s1;
        *s1=*s2;
        *s2=x;
    }
}

```

③测试代码部分, 代码如下。

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<malloc.h>
#define infinity 65535 /*
定义一个无限大的值*/
/*
哈夫曼树类型定义*/
typedef struct
{
    unsigned int weight;
    unsigned int parent,lchild,rchild;
}HTNode,*HuffmanTree;
typedef char **HuffmanCode; /*
存放哈夫曼编码*/
int Min(HuffmanTree t,int n);
void Select(HuffmanTree *t,int n,int *s1,int *s2);
void HuffmanCoding(HuffmanTree *HT,HuffmanCode *HC,int *w,int n);
void main()
{
    HuffmanTree HT;
    HuffmanCode HC;
    int *w,n,i;
    printf("
请输入叶子结点的个数: ");
    scanf("%d",&n);
    w=(int*)malloc(n*sizeof(int)); /*
为n
个结点的权值分配内存空间*/
    for(i=0;i<n;i++)
    {

```

```

                                printf("
请输入第%d
个结点的权值:",i+1);
                                scanf("%d",&w[i]);
                                }
                                HuffmanCoding (&HT, &HC, w, n);
                                for (i=1; i<=n; i++)
                                {
                                    printf("
哈夫曼编码:");
                                    puts (HC[i]);
                                }
                                /*
                                释放内存空间*/
                                for (i=1; i<=n; i++)
                                    free (HC[i]);
                                free (HC);
                                free (HT);
                                }

```

在程序的最后，要记得释放动态申请的内存空间。

在算法的实现过程中，数组HT在初始时和哈夫曼树生成后的状态如图9-47所示。

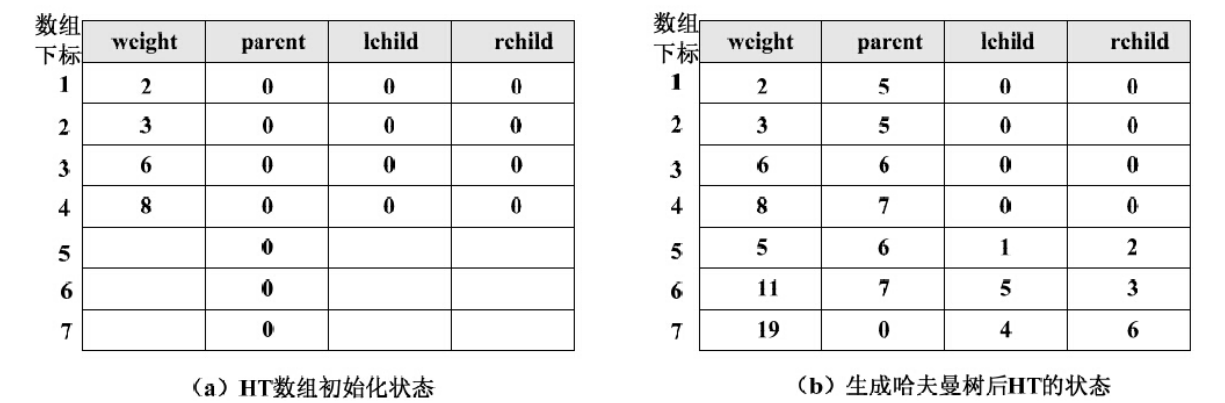


图9-47 数组HT在初始化和生成哈夫曼树后的状态变化情况

生成的哈夫曼树如图9-48所示，不难看出，权值为2、3、6和8的哈夫曼编码分别是100、101、11和0。

程序运行结果如图9-49所示。

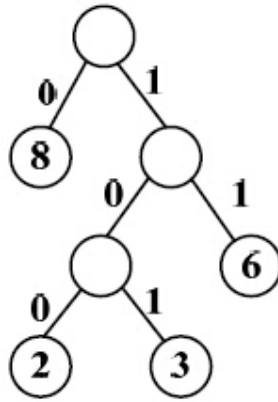


图9-48 哈夫曼树

```
D:\零基础学数据结构\例9_05\...
请输入叶子结点的个数: 4
请输入第1个结点的权值:2
请输入第2个结点的权值:3
请输入第3个结点的权值:6
请输入第4个结点的权值:8
哈夫曼编码:100
哈夫曼编码:101
哈夫曼编码:11
哈夫曼编码:0
Press any key to continue_
```

图9-49 哈夫曼编码程序运行结果

9.9 小结

树是我们学习数据结构课程中的一个重点和难点部分，也是各种考试常考内容之一。树反映的是一种层次结构的关系。树中结点之间是一种一对多的关系。

树与二叉树的定义都是递归的。树中的子树没有次序之分，二叉树的左右子树是有次序的，分别叫做左子树和右子树。

在二叉树中，有两种特殊的树，即满二叉树和完全二叉树。满二叉树中每个非叶子结点都存在左子树和右子树，所有的叶子结点都处在同一层次上。完全二叉树是指与满二叉树的前 n 个结点结构相同，满二叉树是一种特殊的完全二叉树。

二叉树的存储结构有顺序存储和链式存储两种。完全二叉树可以采用顺序存储，采用顺序存储结构可以实现随机存取，实现比较方便。一般来说，一棵二叉树并不一定是完全二叉树，采用顺次存储结构会浪费大量的存储空间。通常情况下，采用二叉链表表示二叉树。二叉链表中的结点包括一个数据域和两个指针域。数据存放结点的值信息，两个指针域分别指向左孩子结点和右孩子结点。

二叉树的遍历是一种常用的操作。二叉树的遍历分为先序遍历、中序遍历和后序遍历。

采用二叉链表表示的二叉树，不能直接找到该结点的直接前驱和后继结点信息，为了能快速找到任何一个结点的直接前驱和直接后继信息，需要对二叉树进行线索化。

哈夫曼树是一种特殊的二叉树，树中只有叶子结点和度为2的结点。哈夫曼树是带权路径最小的二叉树，也称为最优二叉树。

9.10 习题

一、选择题

1. 二叉树的深度为 k ，则二叉树最多有（ ）个结点。

A. $2k$

B. 2^{k-1}

C. $2^k - 1$

D. $2k-1$

2. 用顺序存储的方法，将完全二叉树中所有结点按层逐个从左到右的顺序存放在一维数组 $R[1..N]$ 中，若结点 $R[i]$ 有右孩子，则其右孩子是（ ）。

A. $R[2i-1]$

B. $R[2i+1]$

C. $R[2i]$

D. $R[2/i]$

3. 在一棵具有5层的满二叉树中结点总数为（ ）。

A. 31

B. 32

C. 33

D. 16

4. 由二叉树的前序和后序遍历序列（ ）唯一确定这棵二叉树。

A. 能

B. 不能

5. 某二叉树的中序序列为ABCDEFGH，后序序列为BDCAFHGE，则其左子树中结点数目为（ ）。

A. 3

B. 2

C. 4

D. 5

6. 若以 {4, 5, 6, 7, 8} 作为权值构造哈夫曼树, 则该树的带权路径长度为 ()。

A. 67

B. 68

C. 69

D. 70

7. 将一棵有100个结点的完全二叉树从根这一层开始, 每一层上从左到右依次对结点进行编号, 根结点的编号为1, 则编号为49的结点的左孩子编号为 ()。

A. 98

B. 99

C. 50

D. 48

8. 表达式 $a * (b + c) - d$ 的后缀表达式是 ()。

A. $abcd+-$

B. $abc+*d-$

C. $abc*+d-$

D. $-+*abcd$

9. 对某二叉树进行先序遍历的结果为ABDEFC，中序遍历的结果为DBFEAC，则后序遍历的结果是（ ）。

A. DBFEAC

B. DFEBCA

C. BDFECA

D. BDEFAC

10. 按照二叉树的定义，具有3个结点的二叉树有（ ）种。

A. 3

B. 4

C. 5

D. 6

二、算法分析题

1. 函数depth实现返回二叉树的高度，请在空格处将算法补充完整。

```
int depth(Bitree *t){
    if(t==NULL)
        return 0;
    else{
        hl=depth(t->lchild);
        hr=
        ;
        if(
    )
        return hl+1;
    else
        return hr+1;
    }
}
```

2. 写出下面算法的功能。

```
Bitree *function(Bitree *bt){
    Bitree *t,*t1,*t2;
    if(bt==NULL)
        t=NULL;
    else{
        t=(Bitree *)malloc(sizeof(Bitree));
        t->data=bt->data;
        t1=function(bt->left);
        t2=function(bt->right);
        t->left=t2;
        t->right=t1;
    }
    return(t);
}
```

3. 写出下面算法的功能。

```
void function(Bitree *t){
    if(t!=NULL){
        function(t->lchild);
        function(t->rchild);
        printf("%d",t->data);
    }
}
```


} }

三、综合题

1. 假设一棵二叉树的先序序列为EBADCFHGIKJ，中序序列为ABCDEFGHIJK，请画出该二叉树。
2. 已知二叉树的先序遍历序列为ABCDEFGH，中序遍历序列为CBEDFAGH，画出二叉树。
3. 已知权值集合为{5, 7, 2, 3, 6, 9}，要求给出哈夫曼树，并计算带权路径长度WPL。
4. 对于如图9-50所示的二叉树，请写出先序、中序、后序遍历的序列。

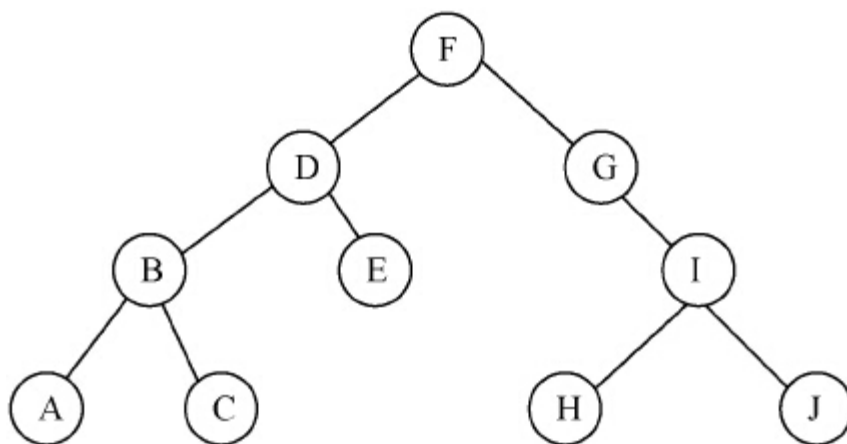


图9-50 二叉树

四、算法设计题

1. 给出求二叉树的所有结点的算法实现。
2. 编写一个算法，判断二叉树是否是完全二叉树。
3. 在二叉链表存储结构的二叉树中，p是指向二叉树中的某个结点的指针，编写算法，求p的所有祖先结点。
4. 编写算法，创建一个如图9-51所示的二叉树，并按照先序遍历、中序遍历和后序遍历的方式输出二叉树的每个结点的值。

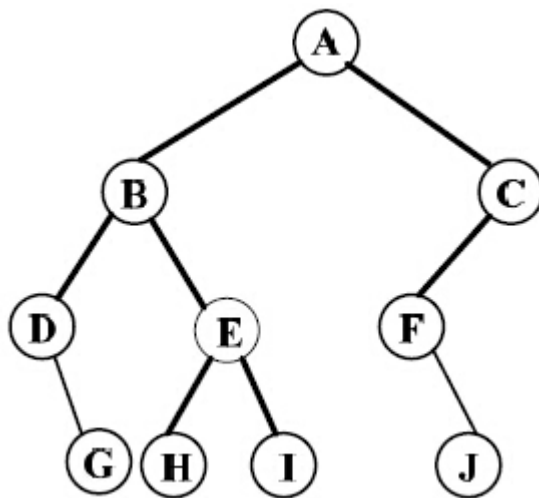


图9-51 二叉树

5. 编写一个判断两颗二叉树是否相似的算法。

相似二叉树指的是二叉树的结构相似。假设存在两棵二叉树T1和T2，T1和T2都是空二叉树或者T1和T2都不为空树，且T1和T2的左、右

子树的结构分别相似。则称 T_1 和 T_2 是相似二叉树。

与相似二叉树相对应的是等价二叉树，两棵二叉树等价是指不仅两棵二叉树相似，且所有二叉树上对应结点的数据元素也相等。

6. 利用孩子兄弟表示法创建一棵树，对树进行层次遍历、先序遍历，并求出树的深度。

第10章 图

图（graph）是一种比线性表、树更为复杂的数据结构。在线性表中，数据元素之间呈线性关系，即每个元素只有一个直接前驱元素和一个直接后继元素。在树结构中，数据元素之间有明显的层次关系，即每个结点只有一个直接前驱结点，但可有多个直接后继结点，而在图结构中，每个结点即可有多个直接前驱结点，也可有多个直接后继结点。图的最早应用可以追溯到18世纪数学家欧拉（Euler）利用图解决了著名的哥尼斯堡桥的问题，为图在现代科学技术领域的应用奠定了基础。

图的应用领域十分广泛，如化学分析、工程设计、遗传学、人工智能等。本章主要介绍图的定义、图的存储结构、图的遍历、最小生成树、关键路径和最短路径。

本章重点和难点：

- 图的定义及性质
- 图的邻接矩阵和邻接表表示
- 图的各种遍历算法实现

- 最小生成树
- 关键路径
- 最短路径

10.1 图的定义与相关概念

图是一种非线性的数据结构，图中的数据元素之间的关系是多对多的关系。本节主要介绍有关图的定义和相关概念。

10.1.1 什么是图

图是由数据元素集合 V 与边的集合 E 构成的。在图中，数据元素通常称为**顶点**（Vertex）。其中，顶点集合 V 不能为空，边表示顶点之间的关系。

（1）若 $\langle x, y \rangle \in E$ ，则 $\langle x, y \rangle$ 表示从顶点 x 到顶点 y 存在一条**弧**（Arc）， x 称为**弧尾**（tail）或**起始点**（initial node）， y 称为**弧头**（head）或**终端点**（terminal node）。这样的图称为**有向图**（digraph），如图10-1（a）所示。

（2）如果 $\langle x, y \rangle \in E$ 且有 $\langle y, x \rangle \in E$ ，即 E 是对称的，则用无序对 (x, y) 代替有序对 $\langle x, y \rangle$ 和 $\langle y, x \rangle$ ，表示 x 与 y 之间存在一条**边**（edge），这样的图称为**无向图**（undigraph），如图10-1（b）所示。

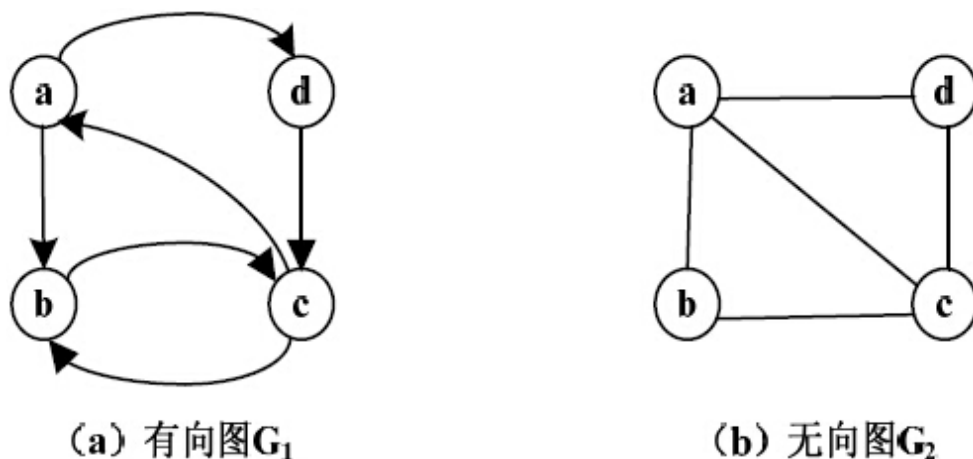


图10-1 有向图 G_1 与无向图 G_2

图的形式化定义为 $G = (V, E)$ ，其中， $V = \{x \mid x \in \text{数据元素集合}\}$ ， $E = \{\langle x, y \rangle \mid \text{Path}(x, y) \wedge (x \in V, y \in V)\}$ 。 $\text{Path}(x, y)$ 表示 $\langle x, y \rangle$ 的意义或信息。

在图10-1中，有向图 G_1 可以表示为 $G_1 = (V_1, E_1)$ ，其中，顶点的集合为 $V_1 = \{a, b, c, d\}$ ，边的集合为 $E_1 = \{\langle a, b \rangle, \langle a, d \rangle, \langle b, c \rangle, \langle c, a \rangle, \langle c, b \rangle, \langle d, c \rangle\}$ 。无向图 G_2 可以表示为 $G_2 = (V_2, E_2)$ ，其中，顶点的集合为 $V_2 = \{a, b, c, d\}$ ，边的集合为 $E_2 = \{(a, b), (a, c), (a, d), (b, c), (c, d)\}$ 。

注意 有向图的边称为弧，无向图的边称为边。图中顶点的顺序可以是任意的，任何一个顶点都可以作为图的第一个顶点。

10.1.2 图的相关概念

下面介绍与图有关的一些概念。

1. 邻接点

对于无向图 $G=(V, E)$ ，若边 $(v_i, v_j) \in E$ ，则称 v_i 和 v_j 互为邻接点 (adjacent)，即 v_i 和 v_j 相邻接。边 (v_i, v_j) 依附于顶点 v_i 和 v_j ，或者说边 (v_i, v_j) 与顶点 v_i 、 v_j 相关联。对于有向图 $G=(V, A)$ ，若弧 $\langle v_i, v_j \rangle \in A$ ，则称顶点 v_i 邻接到顶点 v_j ，顶点 v_j 邻接自顶点 v_i ，弧 $\langle v_i, v_j \rangle$ 和顶点 v_i 、 v_j 相关联。

无向图 G_2 的边的集合为 $E=\{(a, b), (a, c), (a, d), (b, c), (c, d)\}$ ，顶点 a 和 b 互为邻接点，边 (a, b) 依附于顶点 a 和 b 。顶点 c 和 d 互为邻接点，边 (c, d) 依附于顶点 c 和 d 。有向图 G_1 的弧的集合为 $A=\{\langle a, b \rangle, \langle a, d \rangle, \langle b, c \rangle, \langle c, a \rangle, \langle c, b \rangle, \langle d, c \rangle\}$ ，顶点 a 邻接到顶点 b ，弧 $\langle a, b \rangle$ 与顶点 a 和 b 相关联。顶点 c 邻接自顶点 d ，弧 $\langle d, c \rangle$ 与顶点 d 和 c 相关联。

2. 顶点的度

对于无向图，顶点 v 的度是指与 v 相关联的边的数目，记作 $TD(v)$ 。对于有向图，以顶点 v 为弧头的数目称为顶点 v 的入度 (indegree)，记作 $ID(v)$ 。以顶点 v 为弧尾的数目称为 v 的出度

(outdegree)，记作 $OD(v)$ 。顶点 v 的度 (degree) 为 $TD(v) = ID(v) + OD(v)$ 。

无向图 G_2 中顶点 a 的度为3，顶点 b 的度为2，顶点 c 的度为3，顶点 d 的度为2。有向图 G_1 的弧的集合为 $A = \{ \langle a, b \rangle, \langle a, d \rangle, \langle b, c \rangle, \langle c, a \rangle, \langle c, b \rangle, \langle d, c \rangle \}$ ，顶点 a 、 b 、 c 和 d 的入度分别为1、2、2和1，顶点 a 、 b 、 c 和 d 的出度分别为2、1、2和1，顶点 a 、 b 、 c 和 d 的度分别为3、3、4和2。

若图的顶点的个数为 n ，边数或弧数为 e ，顶点 v_i 的度记作 $TD(v_i)$ ，则顶点的度与弧或者边数满足关系
$$e = \frac{1}{2} \sum_{i=1}^n TD(v_i)$$
。

3. 路径

无向图 G 中，从顶点 v 到顶点 v' 的**路径** (path) 是从 v 出发，经过一系列的顶点序列到达顶点 v' 。如果 G 是有向图，则路径也是有向的，路径的长度是路径上弧或边的数目。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外，其他顶点不重复出现的回路，称为**简单回路** 或**简单环**。

在图10-1所示的有向图 G_1 中，顶点序列 $a \rightarrow d \rightarrow c \rightarrow a$ 构成了一个简单回路。在无向图 G_2 中，从顶点 a 到顶点 c 所经过的路径为 a 、 d 、

c（或a、b、c）。

4. 子图

假设存在两个图 $G=\{V, E\}$ 和 $G'=\{V', E'\}$ ，若 G' 的顶点和关系都是 V 的子集，即有 $V'\subseteq V, E'\subseteq E$ ，则 G' 为 G 的子图，如图10-2所示。

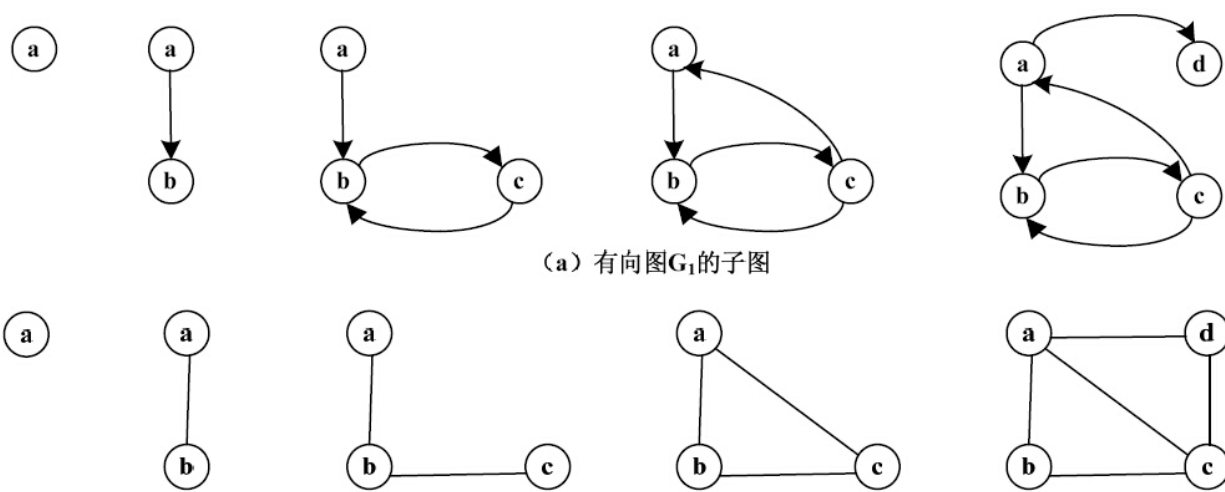


图10-2 有向图 G_1 与无向图 G_2 的子图

5. 连通图和强连通图

对于无向图 G ，如果从顶点 v_i 到顶点 v_j 存在路径，则称 v_i 到 v_j 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）。无向图中的极大连通子图称为**连通分量**。无向图 G_3 与连通分量如图10-3所示。

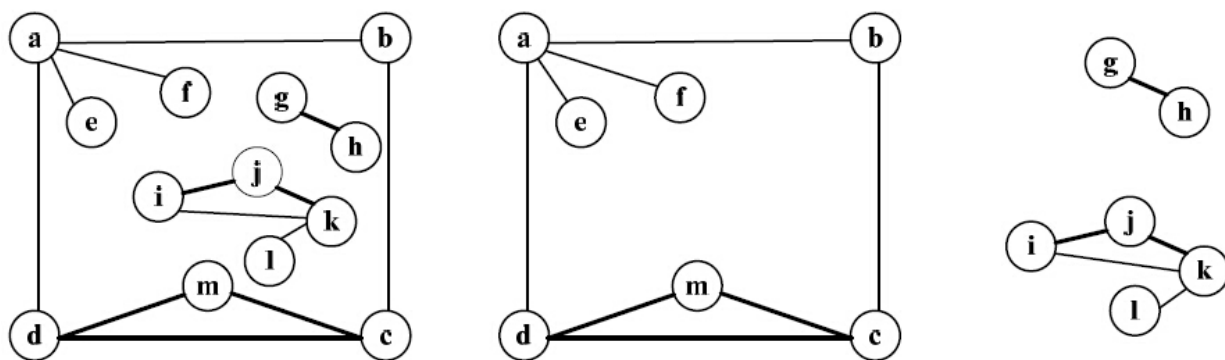
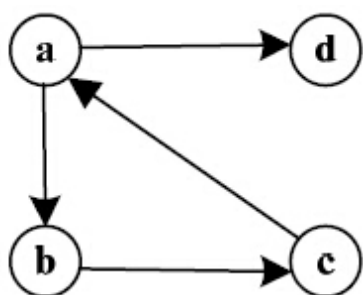
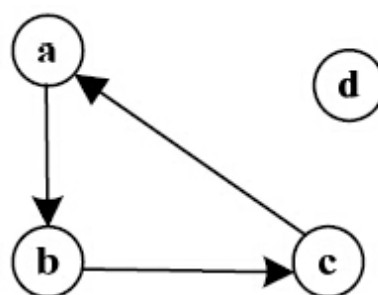


图10-3 无向图 G_3 的连通分量

对于有向图 G ，如果对每一对顶点 v_i 和 v_j ，且 $v_i \neq v_j$ ，从 v_i 到 v_j 和从 v_j 到 v_i 都存在路径，则 G 为**强连通图**。有向图中的极大强连通子图称为有向图的**强连通分量**。有向图 G_4 与强连通分量如图10-4所示。



(a) 有向图 G_4



(b) 有向图 G_4 的两个强连通分量

图10-4 有向图 G_4 及强连通分量

在如图10-4所示的强连通分量中，顶点集合分别为 $\{a, b, c\}$ 和 $\{d\}$ ， a 到任何一个顶点都有路径， b 、 c 到任何一个顶点也存在路径。

6. 完全图

若图的顶点数目是 n ，图的边（弧）的数目是 e 。若不存在顶点到自身的边或弧，即若存在 $\langle v_i, v_j \rangle$ ，则有 $v_i \neq v_j$ 。对于无向图，边数 e 的取值范围为 $0 \sim n(n-1)/2$ 。将具有 $n(n-1)/2$ 条边的无向图称为**完全图**（completed graph）或无向完全图。对于有向图，弧数 e 的取值范围是 $0 \sim n(n-1)$ 。具有 $n(n-1)$ 条弧的有向图称为**有向完全图**。

7. 稀疏图和稠密图

具有 $e < n \log n$ 条弧或边的图称为**稀疏图**，反之称为**稠密图**。

8. 生成树

一个连通图的**生成树**是一个极小连通子图，它含有图的全部顶点，但只有足以构成一棵树的 $n-1$ 条边。如果在该生成树中添加一条边，则一定会在图中出现一个环。一棵具有 n 个顶点的生成树仅有 $n-1$ 条边，如果少于 $n-1$ 条边，则该图是非连通的；多于 $n-1$ 条边，则一定有环的出现。反过来，具有 $n-1$ 条边的图不一定能构成生成树。一个图的生成树不一定是唯一的。图10-5所示是无向图 G_5 中最大连通分量的一棵生成树。

9. 网

在图的边或弧上，有时标有与它们相关的数，这种与图的边或弧相关的数称作**权**（weight）。这些权可以表示从一个顶点到另一个顶点的距离或代价。这种带权的图称为**网**（network），如图10-6所示。

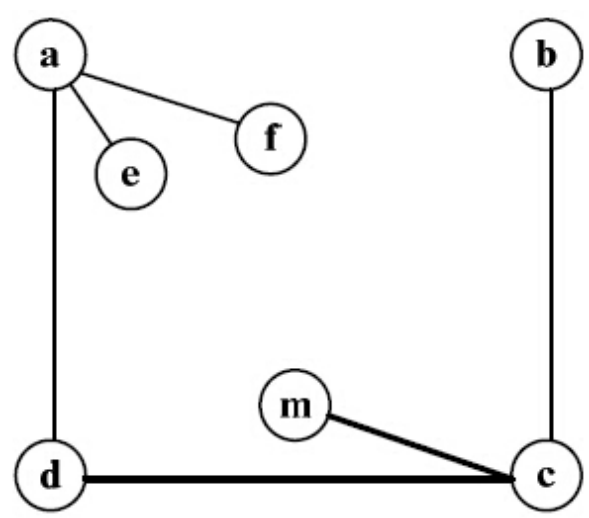


图10-5 有向图G₅ 的生成树

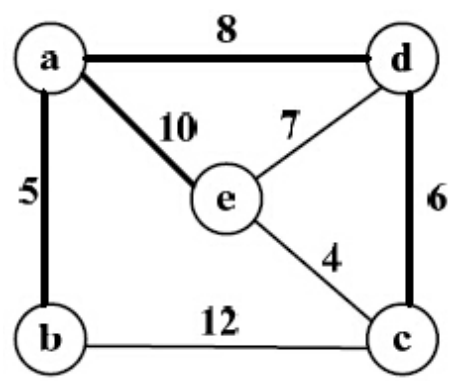


图10-6 网

10.1.3 图的抽象数据类型

1. 数据对象集合

图的数据对象为图的各个顶点和边的集合。图中的顶点是没有先后次序的。图分为有向图和无向图，图中结点之间的关系用弧或边表示，通过弧或边相连的顶点相邻接或相关联。

图中顶点之间是多对多的关系，即任何一个顶点可以有与之邻接或关联的顶点。

2. 基本操作集合

(1) CreateGraph (&G)：创建图。根据顶点和边或弧构造一个图G。

(2) DestroyGraph (&T)：销毁图的操作。如果图G存在，则将图G销毁。

(3) LocateVertex (G, v)：返回顶点v在图中的位置。在图G中查找顶点v，如果找到该顶点，返回顶点在图G中的位置。

(4) GetVertex (G, i)：返回图G中序号i对应的值。i是图G某个顶点的序号，返回图G中序号i对应的值。

(5) FirstAdjVertex (G, v) : 返回v的第一个邻接顶点。在图G中查找v的第一个邻接顶点, 并将其返回。如果在G中没有邻接顶点, 则返回-1。

(6) NextAdjVertex (G, v, w) : 返回v的下一个邻接顶点。在图G中查找v的下一个邻接顶点, 即w的第一个邻接顶点, 找到返回其值, 否则, 返回-1。

(7) InsertVertex (&G, v) : 图的顶点插入操作。在图G中增加新的顶点v, 并将图的顶点数增1。

(8) DeleteVertex (&G, v) : 图的顶点删除操作。将图G中的顶点v及相关联的弧删除。

(9) InsertArc (&G, v, w) : 图的弧插入操作。在图G中增加弧 $\langle v, w \rangle$ 。对于无向图, 还要插入弧 $\langle w, v \rangle$ 。

(10) DeleteArc (&G, v, w) : 图的弧删除操作。在图G中删除弧 $\langle v, w \rangle$ 。对于无向图, 还要删除弧 $\langle w, v \rangle$ 。

(11) DFSTraverseGraph (G) : 图的深度优先遍历操作。从图中的某个顶点出发, 对图进行深度优先遍历。

(12) BFSTraverseGraph (G) : 图的广度优先遍历操作。从图中的某个顶点出发, 对图进行广度优先遍历。

10.2 图的存储结构

在前面几章讨论的数据结构中，除了广义表和树外，都可以有两类不同的存储结构，图的存储方式主要有邻接矩阵表示法、邻接表表示法、十字链表表示法和邻接多重链表表示法4种。

10.2.1 邻接矩阵（数组表示法）

1. 什么是图的邻接矩阵

图的邻接矩阵可利用两个数组实现，一个是一维数组，用来存储图中的顶点信息；另一个是二维数组，用来存储图中顶点之间的关系，该二维数组称为邻接矩阵。如果图是一个无权图，则邻接矩阵表示为：

$$A[i][j] = \begin{cases} 1, & \text{当 } \langle v_i, v_j \rangle \in E \text{ 或 } (v_i, v_j) \in E \\ 0, & \text{反之} \end{cases}$$

$$\text{对于带权图，有 } A[i][j] = \begin{cases} w_{ij}, & \text{当 } \langle v_i, v_j \rangle \in E \text{ 或 } (v_i, v_j) \in E \\ \infty, & \text{反之} \end{cases}$$

其中， w_{ij} 表示顶点*i*与顶点*j*构成的弧或边的权值，如果顶点之间不存在弧或边，则用 ∞ 表示。

例如图10-1中，两个图弧和边的集合分别为 $A = \{\langle a, b \rangle, \langle a, d \rangle, \langle b, c \rangle, \langle c, a \rangle, \langle c, b \rangle, \langle d, c \rangle\}$ 和 $E = \{(a, b), (a, c), (a, d), (b, c), (c, d)\}$ ，它们的邻接矩阵表示如图10-7所示。

$$G_1 = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

(a) 有向图 G_1 的邻接矩阵

$$G_2 = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

(b) 无向图 G_2 的邻接矩阵

图10-7 图的邻接矩阵表示

在无向图的邻接矩阵中，如果有边 (a, b) 存在，则 $\langle a, b \rangle$ 和 $\langle b, a \rangle$ 的对应位置都置为1。

带权图的邻接矩阵表示如图10-8所示。

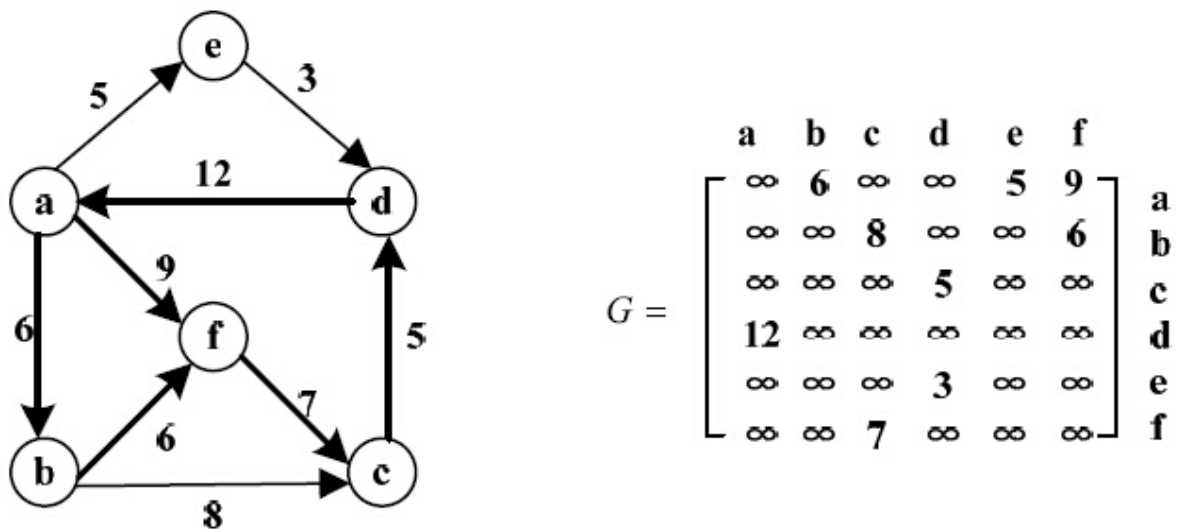


图10-8 带权图的邻接矩阵表示

图的邻接矩阵存储结构描述如下。

```
#define INFINITY 65535 /*65535
被认为是一个无穷大的值*/
#define MaxSize 50 /*
顶点个数的最大值*/
typedef enum{DG,DN,UG,UN}GraphKind; /*
图的类型：有向图、有向网、无向图和无向网*/
```

```

typedef struct
{
    VRType adj;                                /*
    对于无权图，用1
    表示相邻，0
    表示不相邻；对于带权图，存储权值*/
    InfoPtr *info;                             /*
    与弧或边的相关信息*/
}ArcNode, AdjMatrix[MaxSize][MaxSize];
typedef struct                                /*
    图的类型定义*/
{
    VertexType vex[MaxSize];                 /*
    用于存储顶点*/
    AdjMatrix arc;                             /*
    邻接矩阵，存储边或弧的信息*/
    int vexnum, arcnum;                       /*
    顶点数和边（弧）的数目*/
    GraphKind kind;                           /*
    图的类型*/
}MGraph;

```

其中，数组vex用于存储图中的顶点信息，如a、b、c、d；arcs用于存储图中顶点信息。

2. 邻接矩阵应用举例

【例10-1】 试编写一个算法，采用邻接矩阵创建一个如图10-8所示的有向网G。

【分析】 主要考查图的邻接矩阵表示与算法实现。图的创建包括两个部分，一个是创建顶点，顶点信息可存储到一个向量（一维数组）中；另一个是创建弧的信息，包括弧的相关顶点和权值，可存储到二维数组中。其中，二维数组的两个下标分别表示两个相关顶点在矩阵中的行号和列号，权值存入数组中。

创建图可分为如下所述3步。

①根据输入有向图的顶点数、弧（边）数和各个顶点，创建一个数组，存储顶点信息，代码如下。

```
printf("
请输入有向图N
的顶点数,
弧数,
弧的信息(
是:1,
否:0): ");
scanf("%d,%d,%d",&(*N).vexnum,&(*N).arcnum,&InfoFlag);
printf("
请输入%d
个顶点的值(<%d
个字符):\n",N->vexnum,MaxSize);
for(i=0;i<N->vexnum;i++)          /*
创建一个数组，用于保存网的各个顶点*/
    scanf("%s",N->vex[i]);
```

②将邻接矩阵所有值初始化为无穷大，表示顶点之间没有弧相连，代码如下。

```
for(i=0;i<N->vexnum;i++)          /*
初始化邻接矩阵*/
    for(j=0;j<N->vexnum;j++)
    {
        N->arc[i][j].adj=INFINITY;
        N->arc[i][j].info=NULL;    /*
弧的信息初始化为空*/
    }
```

③输入每条弧的弧尾、弧头和权值，调用定位函数LocateVertex，确定每个顶点在数组中的编号，将权值存放在相应的位置，代码如下。

```
printf("
请输入%d
条弧的弧尾
弧头
权值(
以空格作为间隔): \n",N->arcnum);
for(k=0;k<N->arcnum;k++)
{
    scanf("%s%s%d",v1,v2,&w);    /*
输入两个顶点和弧的权值*/
    i=LocateVertex(*N,v1);
    j=LocateVertex(*N,v2);
    N->arc[i][j].adj=w;
}
```

这样就完成了图的邻接矩阵的创建。

完整算法实现代码如下。

```
/*
头文件及图的类型*/
#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<stdlib.h>
typedef char VertexType[4];
typedef char InfoPtr;
typedef int VRType;
#define INFINITY 65535 /*
定义一个无限大的值*/
#define MaxSize 50 /*
顶点个数的最大值*/
typedef enum{DG,DN,UG,UN}GraphKind; /*
图的类型：有向图、有向网、无向图和无向网*/
typedef struct
{
    VRType adj; /*
对于无权图，用1
表示相邻，0
表示不相邻；对于带权图，存储权值*/
    InfoPtr *info; /*
与弧或边的相关信息*/
}ArcNode,AdjMatrix[MaxSize][MaxSize];
typedef struct /*
图的类型定义*/
{
    VertexType vex[MaxSize]; /*
用于存储顶点*/
    AdjMatrix arc; /*
邻接矩阵，存储边或弧的信息*/
    int vexnum,arcnum; /*
顶点数和边（弧）的数目*/
    GraphKind kind; /*
图的类型*/
}MGraph;
void CreateGraph(MGraph *N);
int LocateVertex(MGraph N,VertexType v);
void DestroyGraph(MGraph *N);
void DisplayGraph(MGraph N);
void main()
{
    MGraph N;
    printf("
创建一个网：\n");
    CreateGraph(&N);
    printf("
输出网的顶点和弧：\n");
    DisplayGraph(N);
    printf("
销毁网：\n");
    DestroyGraph(&N);
}
void CreateGraph(MGraph *N)
/*
采用邻接矩阵表示法创建有向网N*/
{
```

```

        int i,j,k,w;
        VertexType v1,v2;
        printf("
请输入有向网N
的顶点数,
弧数: ");

        scanf("%d,%d",&(*N).vexnum,&(*N).arcnum);
        printf("
请输入%d
个顶点的值(<%d
个字符):\n",N->vexnum,MaxSize);
        for(i=0;i<N->vexnum;i++) /*
创建一个数组,用于保存网的各个顶点*/
            scanf("%s",N->vex[i]);
        for(i=0;i<N->vexnum;i++) /*
初始化邻接矩阵*/
            for(j=0;j<N->vexnum;j++)
            {
                N->arc[i][j].adj=INFINITY;
                N->arc[i][j].info=NULL; /*
弧的信息初始化为空*/
            }
        printf("
请输入%d
条弧的弧尾
弧头
权值(
以空格作为间隔): \n",N->arcnum);
        for(k=0;k<N->arcnum;k++)
        {
            scanf("%s%s%d",v1,v2,&w); /*
输入两个顶点和弧的权值*/
            i=LocateVertex(*N,v1);
            j=LocateVertex(*N,v2);
            N->arc[i][j].adj=w;
        }
        N->kind=DN; /*
图的类型为有向网*/
    }
    int LocateVertex(MGraph N,VertexType v)
    /*
    在顶点向量中查找顶点v
    ,找到返回在向量的序号,否则返回-1*/
    {
        int i;
        for(i=0;i<N.vexnum;++i)
            if(strcmp(N.vex[i],v)==0)
                return i;
        return -1;
    }
    void DestroyGraph(MGraph *N)
    /*
    销毁网*/
    {
        int i,j;
        for(i=0;i<N->vexnum;i++) /*
释放弧的相关信息*/
            for(j=0;j<N->vexnum;j++)
                if(N->arc[i][j].adj!=INFINITY) /*
如果存在弧*/
                    if(N->arc[i][j].info!=NULL) /*
如果弧有相关信息,释放该信息所占空间*/
                    {
                        free(N->arc[i][j].info);
                        N->arc[i][j].info=NULL;
                    }
        N->vexnum=0; /*
将网的顶点数置为0*/
        N->arcnum=0; /*
.....

```

```

将网的弧的数目置为0*/
}
void DisplayGraph(MGraph N)
/*
输出邻接矩阵存储表示的图N*/
{
    int i,j;
    printf("

有向网具有%d
个顶点%d
条弧，顶点依次是：",N.vexnum,N.arcnum);
    for(i=0;i<N.vexnum;++i) /*
输出网的顶点*/
        printf("%s ",N.vex[i]);
    printf("\n

有向网N
的:\n"); /*
输出网N
的弧*/

    printf("
序号i=");
    for(i=0;i<N.vexnum;i++)
        printf("%8d",i);
    printf("\n");
    for(i=0;i<N.vexnum;i++)
    {
        printf("%8d",i);
        for(j=0;j<N.vexnum;j++)
            printf("%8d",N.arc[i][j].adj);
        printf("\n");
    }
}

```

程序运行结果如图10-9所示。

```

D:\零基础学数据结构\例10_1\Debug\例10_1.exe
创建一个网:
请输入有向网N的顶点数、弧数: 6,9
请输入6个顶点的值(<50个字符>):
a b c d e f
请输入9条弧的弧尾 弧头 权值(以空格作为间隔):
a b 6
a e 5
a f 9
b c 8
b f 6
c d 5
d a 12
e d 3
f c 7
输出网的顶点和弧:
有向网具有6个顶点9条弧，顶点依次是: a b c d e f
有向网N的:
序号i=
0 65535 1 6 65535 3 65535 4 5 9
1 65535 65535 8 65535 65535 6
2 65535 65535 65535 5 65535 65535
3 12 65535 65535 65535 65535
4 65535 65535 65535 3 65535 65535
5 65535 65535 7 65535 65535
销毁网:
Press any key to continue_

```

图10-9 采用邻接矩阵创建图的运行结果

10.2.2 邻接表

邻接表（adjacency list）是图的一种链式存储方式。采用邻接表表示图一般需要两个表结构：边表和表头结点表。

在邻接表中，对图中的每个顶点都建立一个单链表，第 i 个单链表中的结点表示依附于顶点 v_i 的边（对有向图来说是以顶点 v_i 为尾的弧），这种链表称为**边表**，其中结点称为**弧结点**。弧结点由3个域组成，分别为**邻接点域**（adjvex）、**数据域**（info）和**指针域**

（nextarc），邻接点域表示与相应的表头顶点相邻接顶点的位置，数据域存储与边或弧的信息，指针域用来指示与表头相邻接的下一个顶点。

在每个链表前面设置一个头结点，除了设有存储各个顶点信息的数据域（data）外，还设有指向对应边表中第一个结点的链域（firstarc），这种表称为**表头结点表**。相应地，结点称为**表头结点**。通常情况下，表头结点采用顺序存储结构实现，这样可以随机地访问任意顶点。

边表结点和表头结点的结构如图10-10所示。

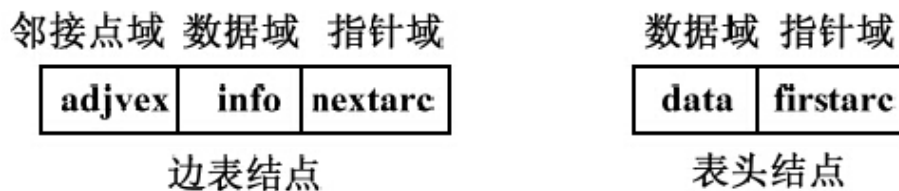


图10-10 边表结点和表头结点存储结构

图10-1所示的图G₁和G₂用邻接表表示如图10-11所示。

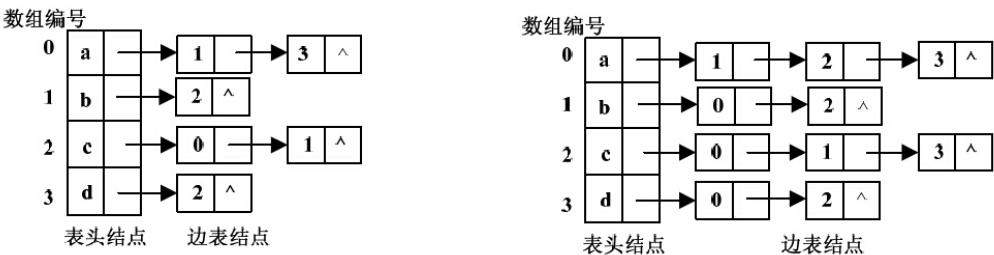


图10-11 图的邻接表表示

图10-8所示的带权图的邻接表如图10-12所示。

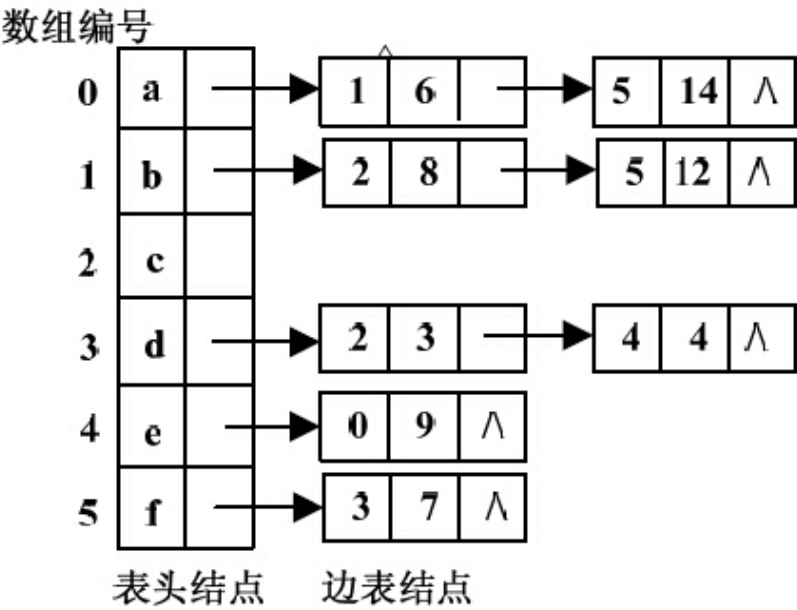


图10-12 带权图的邻接表表示

图的邻接表存储结构描述如下。

```
#define MaxSize 50 /*
顶点个数的最大值*/
typedef enum{DG,DN,UG,UN}GraphKind; /*
图的类型：有向图、有向网、无向图和无向网*/
typedef struct ArcNode /*
边结点的类型定义*/
{
```



```

    int adjvex;                                /*
弧指向的顶点的位置*/
    InfoPtr *info;                             /*
与弧相关的信息*/
    struct ArcNode *nextarc;                    /*
指示下一个与该顶点相邻接的顶点*/
}ArcNode;
typedef struct VNode                           /*
头结点的类型定义*/
{
    VertexType data;                           /*
用于存储顶点*/
    ArcNode *firstarc;                         /*
指示第一个与该顶点邻接的顶点*/
}VNode,AdjList[MaxSize];
typedef struct                                  /*
图的类型定义*/
{
    AdjList vertex;
    int vexnum,arcnum;                         /*
图的顶点数目与弧的数目*/
    GraphKind kind;                           /*
图的类型*/
}AdjGraph;

```

如果无向图G中有n个顶点和e条边，则图采用邻接表表示，需要n个头结点和2e个表结点。在e远小于 $n(n-1)/2$ 时，采用邻接表存储表示显然要比采用邻接矩阵表示更能节省空间。

在图的邻接表存储结构中，某个顶点的度正好等于该顶点对应链表中的结点个数。对于有向图的邻接表来说，某顶点对应链表的结点个数等于某个顶点的出度。

有时为了便于求某个顶点的入度，需要建立一个有向图的逆邻接链表，也就是为每个顶点 v_i 建立一个以 v_i 为弧头的链表。在邻接表中，边表结点的邻接点域的值为i的个数，就是顶点 v_i 的入度。因此如果要求某个顶点的入度，则需要对整个邻接表进行遍历。图10-1所示的有向图 G_1 的逆邻接链表如图10-13所示。

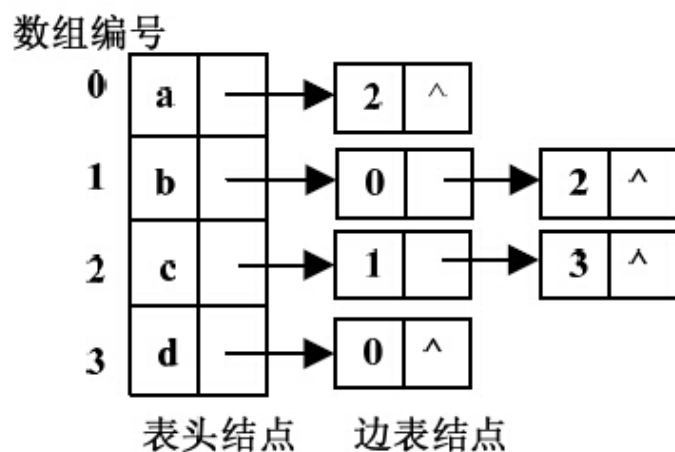


图10-13 有向图 G_1 的逆邻接链表

【例10-2】 编写创建如图10-1所示的无向图（假设采用邻接表表示图）的算法。

【分析】 主要考查图的邻接表存储结构。图的创建包括两个部分，即创建表头结点和边表结点。其中，表头结点利用一个数组实现，数组包括两个域，一个是保存顶点的值；一个是指针，用于指向与顶点相关联的顶点对应结点。代码如下。

```

for (i=0; i<G->vexnum; i++)          /*
将顶点存储在表头结点中*/
{
    scanf("%s", G->vertex[i].data);
    G->vertex[i].firstarc=NULL; /*
将相关联的顶点置为空*/
}

```

边表结点利用链表实现，主要存储与对应表头中的顶点的关联顶点。创建邻接表类似于创建单链表。每生成一个边表结点，需要查找该边表结点元素值在表头结点中的编号，然后将该结点插入边表的表头。

算法实现如下。

```

/*
头文件*/
#include<stdlib.h>
#include<stdio.h>
#include<malloc.h>
#include<string.h>
/*
图的邻接表类型定义*/
typedef char VertexType[4];
typedef char InfoPtr;
typedef int VRType;
#define MaxSize 50 /*
顶点个数的最大值*/
typedef enum{DG,DN,UG,UN}GraphKind; /*
图的类型：有向图、有向网、无向图和无向网*/
typedef struct ArcNode /*
边表结点的类型定义*/
{
    int adjvex; /*
弧指向的顶点的位置*/
    InfoPtr *info; /*
与弧相关的信息*/
    struct ArcNode *nextarc; /*
指示下一个与该顶点相邻接的顶点*/
}ArcNode;
typedef struct VNode /*
表头结点的类型定义*/
{
    VertexType data; /*
用于存储顶点*/
    ArcNode *firstarc; /*
指示第一个与该顶点邻接的顶点*/
}VNode,AdjList[MaxSize];
typedef struct /*
图的类型定义*/
{
    AdjList vertex;
    int vexnum,arcnum; /*
图的顶点数目与弧的数目*/
    GraphKind kind; /*
图的类型*/
}AdjGraph;
/*
函数声明*/
int LocateVertex(AdjGraph G,VertexType v);
void CreateGraph(AdjGraph *G);
void DisplayGraph(AdjGraph G);
void DestroyGraph(AdjGraph *G);
void main()
{
    AdjGraph G;
    printf("
采用邻接矩阵创建无向图G
: \n");
    CreateGraph(&G);
    printf("
输出无向图G
: ");
    DisplayGraph(G);
    DestroyGraph(&G);
}
void CreateGraph(AdjGraph *G)
/*
采用邻接表存储结构，创建无向图G*/
{
    int i,j,k;
    VertexType v1,v2; /*
定义两个顶点v1

```

```

和v2*/
        ArcNode *p;
        printf("
输入图的顶点数,
边数(
逗号分隔): ");
        scanf("%d,%d",&(*G).vexnum,&(*G).arcnum);
        printf("
输入%d
个顶点的值:\n",G->vexnum);
        for(i=0;i<G->vexnum;i++) /*
将顶点存储在表头结点中*/
        {
                scanf("%s",G->vertex[i].data);
                G->vertex[i].firstarc=NULL; /*
将相关联的顶点置为空*/
        }
        printf("
输入弧尾和弧头(
以空格作为间隔):\n");
        for(k=0;k<G->arcnum;k++) /*
建立边链表*/
        {
                scanf("%s%s",v1,v2);
                i=LocateVertex(*G,v1);
                j=LocateVertex(*G,v2);
                /*j
为入边i
为出边创建邻接表*/
                p=(ArcNode*)malloc(sizeof(ArcNode));
                p->adjvex=j;
                p->info=NULL;
                p->nextarc=G->vertex[i].firstarc;
                G->vertex[i].firstarc=p;
                /*i
为入边j
为出边创建邻接表*/
                p=(ArcNode*)malloc(sizeof(ArcNode));
                p->adjvex=i;
                p->info=NULL;
                p->nextarc=G->vertex[j].firstarc;
                G->vertex[j].firstarc=p;
        }
        (*G).kind=UG;
}
int LocateVertex(AdjGraph G,VertexType v)
/*
返回图中顶点对应的位置*/
{
        int i;
        for(i=0;i<G.vexnum;i++)
                if(strcmp(G.vertex[i].data,v)==0)
                        return i;
        return -1;
}
void DestroyGraph(AdjGraph *G)
/*
销毁无向图G*/
{
        int i;
        ArcNode *p,*q;
        for(i=0;i<(*G).vexnum;++i) /*
释放图中的边表结点*/
        {
                p=G->vertex[i].firstarc; /*p
指向边表的第一个结点*/
                if(p!=NULL) /*
如果边表不为空,则释放边表的结点*/
                {

```

```

                                q=p->nextarc;
                                free(p);
                                p=q;
                                }
                                }
                                (*G).vexnum=0;                                /*
将顶点数置为0*/
                                (*G).arcnum=0;                                /*
将边的数目置为0*/
                                }
void DisplayGraph(AdjGraph G)
/*
图的邻接表存储结构的输出*/
{
    int i;
    ArcNode *p;
    printf("%d
个顶点: \n",G.vexnum);
    for(i=0;i<G.vexnum;i++)
        printf("%s ",G.vertex[i].data);
    printf("\n%d
条边:\n",2*G.arcnum);
    for(i=0;i<G.vexnum;i++)
    {
        p=G.vertex[i].firstarc;                                /*
将p
指向边表的第一个结点*/
        while(p)                                /*
输出无向图的所有边*/
        {
            printf("%s
-%s ",G.vertex[i].data,G.vertex[p->adjvex].data);
            p=p->nextarc;
        }
        printf("\n");
    }
}

```

程序的运行结果如图10-14所示。

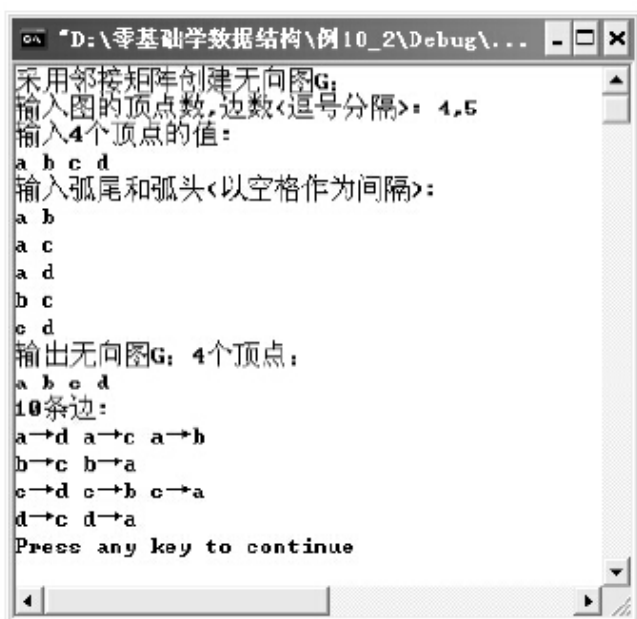


图10-14 采用邻接表创建图的程序运行结果

10.2.3 十字链表

十字链表（orthogonal list）是有向图的另一种链式存储结构，它可以看做将有向图的邻接表与逆邻接链表结合起来的一种链表。十字链表中结点的结构如图10-15所示。

tailvex	headvex	hlink	tlink	info
---------	---------	-------	-------	------

(a) 弧结点

data	firstin	firstout
------	---------	----------

(b) 顶点结点

图10-15 十字链表中的结点结构

弧结点包含5个域，分别为尾域tailvex、头域headvex、infor域和两个指针域hlink、tlink，尾域tailvex用于表示弧尾顶点在图中的位置，

头域headvex表示弧头顶点在图中的位置，info域表示弧的相关信息，指针域hlink指向弧头相同的下一条弧，tlink指向弧尾相同的下一条弧。

顶点结点包含3个域，分别为data域和firstin域、firstout域，data域存储与顶点相关的信息，如顶点的名称；firstin域和firstout域是两个指针域，分别指向以该顶点为弧头和弧尾的第一个弧结点。

有向图G₁ 的十字链表存储表示如图10-16所示。

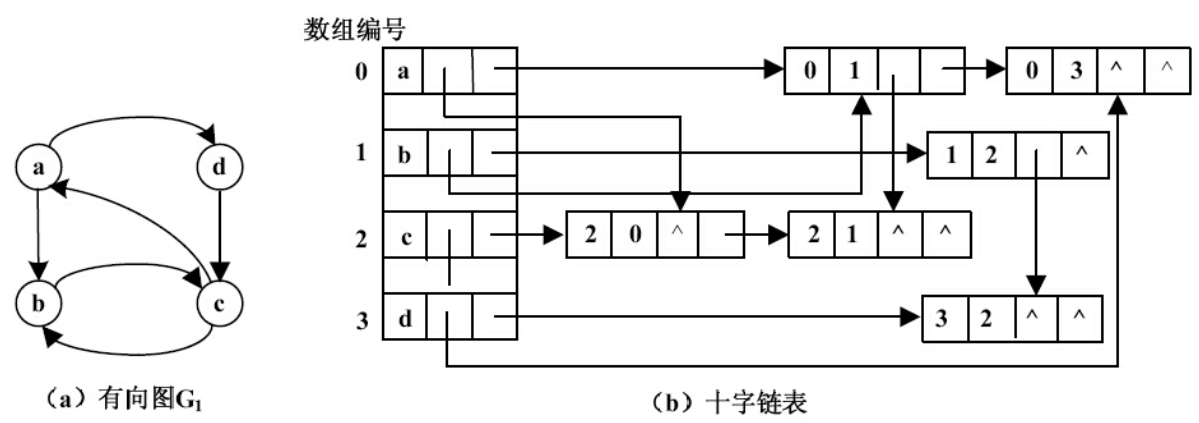


图10-16 有向图G₁ 的十字链表存储结构

有向图的十字链表存储结构描述如下。

```
#define MaxSize 50 /*
顶点个数的最大值*/
typedef struct ArcNode /*
弧结点的类型定义*/
{
    int headvex,tailvex; /*
    弧的头顶点和尾顶点位置*/
    InfoPtr *info; /*
    与弧相关的信息*/
    struct *hlink,*tlink; /*
    指示弧头和弧尾相同的结点*/
}ArcNode;
typedef struct VNode /*
顶点结点的类型定义*/
{
    VertexType data; /*
    用于存储顶点*/
    ArcNode *firstin,*firstout; /*
    分别指向顶点的第一条入弧和出弧*/
```

```
}VNode;
typedef struct                                /*
图的类型定义*/
{
    VNode vertex[MaxSize];
    int vexnum, arcnum;                      /*
图的顶点数目与弧的数目*/
}OLGraph;
```

十字链表中的表头结点即顶点结点之间不是链接存储，而是顺序存储。在图的十字链表中，可以很容易找到以某个顶点为弧尾和弧头的弧。

10.2.4 邻接多重链表

邻接多重链表（adjacency multilist）是无向图的另一种链式存储结构。在无向图的邻接表存储表示中，虽然很容易求得顶点和边的各种信息，但是对于每一条边（ v_i, v_j ）都有两个结点，分别存储在第*i*个和第*j*个链表中，这给图的某些操作带来不便，例如，要删除一条边，此时需要找到表示同一条边的两个顶点。因此，在进行这一类操作时，采用邻接多重链表比较合适，邻接多重链表是将图的一条边用一个结点表示，它的结点结构如图10-17所示。

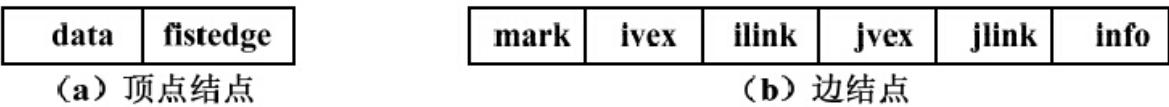


图10-17 邻接多重表的结点结构

顶点结点包含两个域，分别为data域和firstedge域，data为数据域，存储顶点的数据信息；firstedge为指针域，指示依附于顶点的第一条边。边结点有6个域，分别为mark域、ivex域、ilink域、jvex域、

jlink域和info域，mark域用来表示边是否被检索过，ivex域和jvex域表示依附于边的两个顶点在图中的位置，ilink域指向依附于顶点ivex的下一条边，jlink域指向依附于顶点jvex的下一条边，info域表示与边相关的信息。

无向图G₂ 的多重链表如图10-18所示。

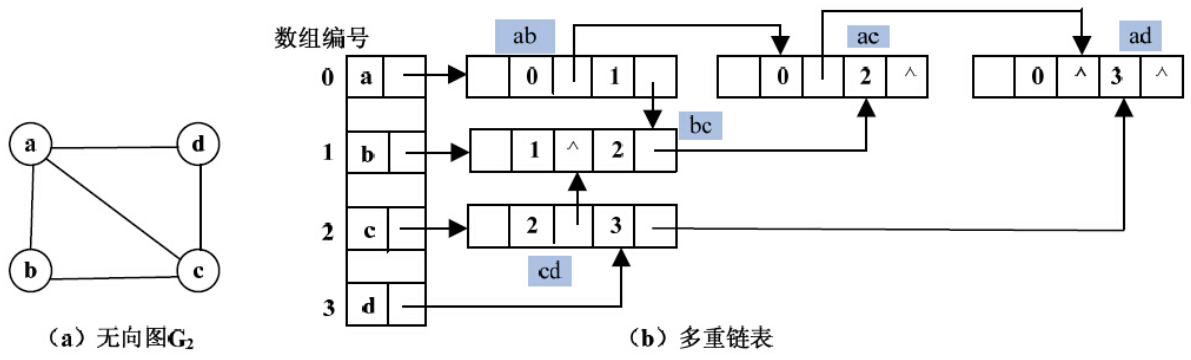


图10-18 无向图G₂ 的多重链表

无向图的多重链表存储结构描述如下。

```
#define MaxSize 50 /*
顶点个数的最大值*/
typedef struct EdgeNode /*
边结点的类型定义*/
{
    int mark, ivex, jvex; /*
    访问标志和边的两个顶点位置*/
    InfoPtr *info; /*
    与边相关的信息*/
    struct *ilink, *jlink; /*
    指示与边顶点相同的结点*/
}EdgeNode;
typedef struct VNode /*
顶点结点的类型定义*/
{
    VertexType data; /*
    用于存储顶点*/
    EdgeNode *firstedge; /*
    指向依附于顶点的第一条边*/
}VexNode;
typedef struct /*
图的类型定义*/
{
    VexNode vertex[MaxSize];
    int vexnum, edgenum; /*
```

```
图的顶点数目与边的数目*/  
}AdjMultiGraph;
```

10.3 图的遍历

与树的遍历类似，从图中某一顶点出发访问遍图中其余顶点，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。图的遍历方式主要有深度优先搜索和广度优先搜索两种。

10.3.1 图的深度优先搜索

1. 什么是图的深度优先搜索遍历

图的深度优先搜索（depth_first search）遍历类似于树的先根遍历，是树的先根遍历的推广。图的深度优先遍历的思想是：假设初始状态是图中所有顶点未曾被访问，从图中某个顶点 v_0 出发，访问顶点 v_0 ，然后依次从 v_0 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_0 有路径相通的顶点都被访问到；若此时图中还有顶点未被访问，则另选图中一个未被访问的顶点作为起始点，重复执行上述过程，直到图中所有的顶点都被访问过。

图的深度优先搜索遍历过程如图10-19所示，实箭头表示访问顶点的方向，虚箭头表示回溯，数字表示访问或回溯的次序。

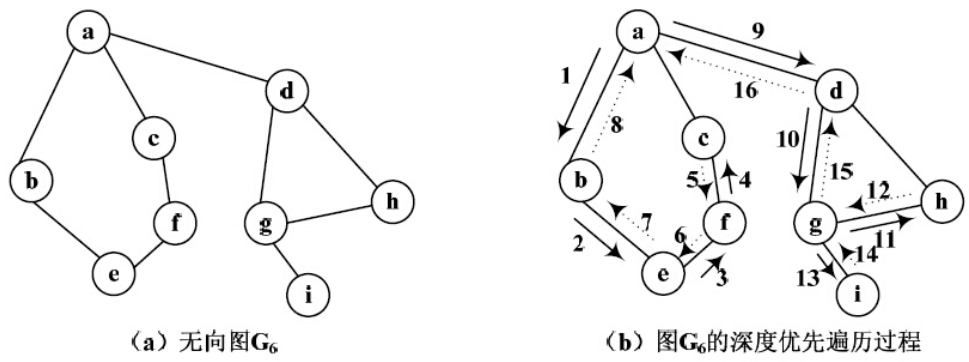


图10-19 图 G_6 及深度优先遍历过程

图的深度优先搜索遍历过程如下。

(1)从顶点a出发，因a还未被访问过，首先访问a。

(2)因a的邻接点有b、c、d，先访问a的第1个邻接点b。

(3)顶点b还有一个邻接点e未被访问过，故访问顶点e。

(4)顶点e的邻接点f还未被访问过，故访问顶点f。

(5)顶点f的邻接点c还未被访问过，故访问顶点c。

(6)顶点c的邻接点都被已经访问过，此时回溯到前一个顶点f。

(7)同理，顶点f、e、b都被已经访问过，且没有其他未访问的邻接点，因此，回溯到顶点a。

(8)顶点a的邻接点d还没有被访问过，故访问顶点d。

(9)顶点d的邻接点有g和h两个，先访问第一个顶点g。

(10)顶点g的邻接点有h和i两个，先访问第一个顶点h。

(11)顶点h的邻接点都被已经访问过，回溯到前一个顶点g。

(12)顶点g的未被访问过的邻接点只有i，故访问顶点i。

(13)顶点i所有的邻接点都已经能够被访问过，回溯到顶点g。

(14)同理，顶点g、d都没有未被访问的邻接点，回溯到顶点a。

(15)顶点a所有的邻接点都被已经访问过，得到图的深度优先搜索遍历的序列为a、b、e、f、c、d、g、h、i。

在图的深度优先搜索遍历过程中，图中可能存在回路，因此，在访问某个顶点之后，沿着某条路径遍历，有可能又回到该顶点。例如，在访问顶点a之后，接着访问顶点b、e、f、c，顶点c的邻接点是顶点a，因顶点c的邻接点是a，因此会继续沿着边（c，a）再次访问顶点a。为了避免再次访问已经访问过的顶点，需要设置一个数组visited[n]，作为一个标志记录结点是否访问过，其初值为0，一旦某个顶点被访问，则其相应的分量被置为1。

2. 图的深度优先搜索遍历的算法实现

图的深度优先遍历（邻接表实现）的算法描述如下。

```
int visited[MaxSize];    /*
访问标志数组*/
void DFSTraverse(AdjGraph G)
/*
从第1
个顶点起，深度优先搜索遍历图G*/
{
    int v;
    for (v=0;v<G.vexnum;v++)
        visited[v]=0;          /*
访问标志数组初始化为未被访问*/
    for (v=0;v<G.vexnum;v++)
        if (!visited[v])
            DFS(G,v);          /*
对未访问的顶点v
进行深度优先搜索遍历*/
    printf("\n");
}
void DFS(AdjGraph G,int v)
/*
从顶点v
出发递归深度优先搜索遍历图G*/
{
    int w;
    visited[v]=1;              /*
访问标志设置为已访问*/
    Visit(G.vertex[v].data);    /*
访问第v
个顶点 */
    for (w=FirstAdjVertex(G,G.vertex[v].data);w>=0;w=NextAdjVertex(G,G.vertex[v].data,G.v ertex[w].data))
        if (!visited[w])
            DFS(G,w);          /*
递归调用DFS
对v
的尚未访问的序号为w
的邻接顶点*/
}
```

如果该图是一个无向连通图或者一个强连通图，则只需要调用一次DFS（G，v）就可以遍历整个图，否则需要多次调用DFS（G，v）。在遍历图时，对图中的每个顶点至多调用一次DFS（G，v）函数，因为一旦某个顶点被标志为已被访问，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。其时间耗费取决于所采用的存储结构。当用二维数组表示邻接矩阵作为图的存储结构时，查找每个顶点的邻接点所需

时间为 $O(n^2)$ ，其中 n 为图中的顶点数。当以邻接表作为图的存储结构时，查找邻接点的时间为 $O(e)$ ，其中， e 为无向图边的数目或有向图弧的数目。由此，当以邻接表作为存储结构时，深度优先搜索遍历图的时间复杂度为 $O(n+e)$ 。

图的深度优先搜索遍历算法DFS的另外一种写法如下。

```
void DFS(AdjGraph G,int v)
/*
从顶点v
出发递归深度优先搜索遍历图G*/
{
    ArcNode *p;
    visited[v]=1;
    Visit(G.vertex[v].data);
    访问标志设置为已访问*/
    访问第v
    个顶点 */
    p=G.vertex[v].firstarc;
    取v
    的边表头指针，p
    指向v
    的邻接点*/
    while(p)
    /*
    依次搜索v
    的邻接点*/
    {
        if(!visited[p->adjvex])
        /*
        若v
        尚未被访问*/
        DFS(G,p->adjvex);
        以v
        的邻接点纵深搜索*/
        p=p->nextarc;
        找v
        的下一个邻接点*/
    }
}
```

以邻接表作为存储结构，查找 v 的第一个邻接点，算法实现如下。

```
int FirstAdjVertex(AdjGraph G,VertexType v)
/*
返回顶点v
的第一个邻接顶点的序号*/
{
    ArcNode *p;
    int vl;
    vl=LocateVertex(G,v);
    为顶点v
    在图G
    中的序号*/
    p=G.vertex[vl].firstarc;
    if(p)
    /*
    如果顶点v
    的第一个邻接点存在，返回邻接点的序号，否则返回-1 */
    return p->adjvex;
    else
    return -1;
}
```

以邻接表作为存储结构，查找 v 的相对于 w 的下一个邻接点，算法实现如下。

```

int NextAdjVertex (AdjGraph G,VertexType v,VertexType w)
/*
  返回v
  的相对于w
  的下一个邻接顶点的序号*/
{
    ArcNode *p,*next;
    int v1,w1;
    v1=LocateVertex (G,v);          /*v1
    为顶点v
    在图G
    中的序号*/
    w1=LocateVertex (G,w);          /*w1
    为顶点w
    在图G
    中的序号*/

    for (next=G.vertex[v1].firstarc;next;)
        if (next->adjvex!=w1)
            next=next->nextarc;

    p=next;                          /*p
    指向顶点v
    的邻接顶点w
    的结点*/

    if (!p||!p->nextarc)              /*
    如果w
    不存在或w
    是最后一个邻接点，则返回-1*/
        return -1;
    else
        return p->nextarc->adjvex;    /*
    返回v
    的相对于w
    的下一个邻接点的序号*/
}

```

10.3.2 图的广度优先搜索

本小节介绍图的广度优先搜索遍历的定义和算法实现。

1. 什么是图的广度优先搜索遍历

图的**广度优先搜索**（breadth_first search）遍历类似于树的层次遍历过程。图的广度优先搜索遍历的思想是：从图的某个顶点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他们的邻接点，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，直至图中所有已访问的顶点的邻接点都被访问到；若此时图中还有顶点未被访问，则另选图中一个未曾被访问的顶点作为起始点，重复上述过程，直至图中的所有顶点都被访问到为止。

例如，图G₆的广度优先搜索遍历的过程如图10-20所示。其中，箭头表示广度遍历的方向，旁边的数字表示遍历的次序。图G₆的广度优先搜索遍历的过程如下。

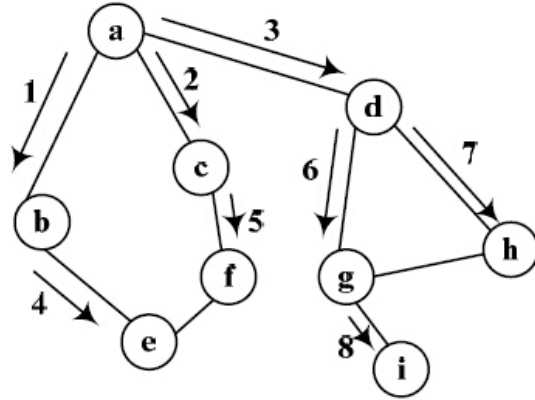


图10-20 图 G_6 的广度优先搜索遍历过程

- ①首先从顶点a出发，因a还未被访问过，首先访问顶点a。
- ②顶点a的邻接点有b、c、d，先访问a的第一个邻接点b。
- ③顶点a的邻接点c还没有被访问，故访问邻接点c。
- ④顶点a的邻接点d还没有被访问，故访问邻接点d。
- ⑤顶点b邻接点e还没有被访问，故访问顶点e。
- ⑥顶点c的邻接点f还没有被访问过，故访问顶点f。
- ⑦顶点d的邻接点有g和h，且都未被访问过，先访问第一个顶点g。
- ⑧顶点d的邻接点h还没有被访问，故访问h。
- ⑨顶点e、f、h不存在未被访问的邻接点，顶点g未被访问的邻接点只有i，故访问顶点i。至此，图 G_6 所有的顶点已经访问完毕。

因此，图 G_6 的广度优先搜索遍历序列为a、b、c、d、e、f、g、h、i。

2. 图的广度优先搜索遍历的算法实现

与深度优先搜索遍历类似，在图的广度优先搜索遍历过程中也需要一个访问标志数组 `visited[MaxSize]`，用来表示顶点是否被访问过。初始时，将图中的所有顶点的标志数组 `visited[vi]` 都初始化为0，表示顶点未被访问。从第一个顶点 `v0` 出发，访问该顶点并将标志数组置为1；然后将 `v0` 入队，当队列不为空时，将队头元素（顶点）出队，依次访问该顶点的所有邻接点，同时将标志数组对应位置1，并将其邻接点依次入队。依次类推，直到图中的所有顶点都已被访问过。

图的广度优先搜索遍历的算法实现如下。

```

void BFSTraverse(AdjGraph G)
/*
从第一个顶点出发，按广度优先非递归遍历图G*/
{
    int v, front, rear;
    ArcNode *p;
    int queue[MaxSize];
    定义一个队列*/
    front=rear=-1;
    初始化队列*/
    for (v=0;v<G.vexnum;v++)
    初始化标志位*/
        visited[v]=0;
    v=0;
    visited[v]=1;
    设置访问标志为1
    , 表示已经被访问过*/
    Visit(G.vertex[v].data);
    rear=(rear+1)%MaxSize;
    queue[rear]=v;
    入队列*/
    while (front<rear)
    如果队列不为空*/
    {
        front=(front+1)%MaxSize;
        v=queue[front];
        队头元素出队赋值给v*/
        p=G.vertex[v].firstarc;
        while (p!=NULL)
        遍历序号为v
        的所有邻接点*/
        {
            如果该顶点未被访问过*/
            if(visited[p->adjvex]==0)
            {
                visited[p->adjvex]=1;
                Visit(G.vertex[p->adjvex].data);
                rear=(rear+1)%MaxSize;
                queue[rear]=p->adjvex;
            }
            p=p->nextarc;
            指向下一个邻接点*/
        }
    }
}

```

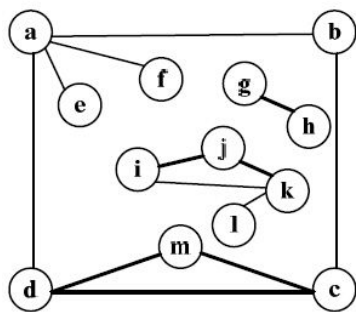
设图的顶点个数为 n ，边（弧）的数目为 e ，则采用邻接表实现图的广度优先遍历的时间复杂度为 $O(n+e)$ 。图的深度优先遍历和广度优先遍历的结果并不是唯一的，这主要与图的存储结点的位置有关。

10.4 图的连通性问题

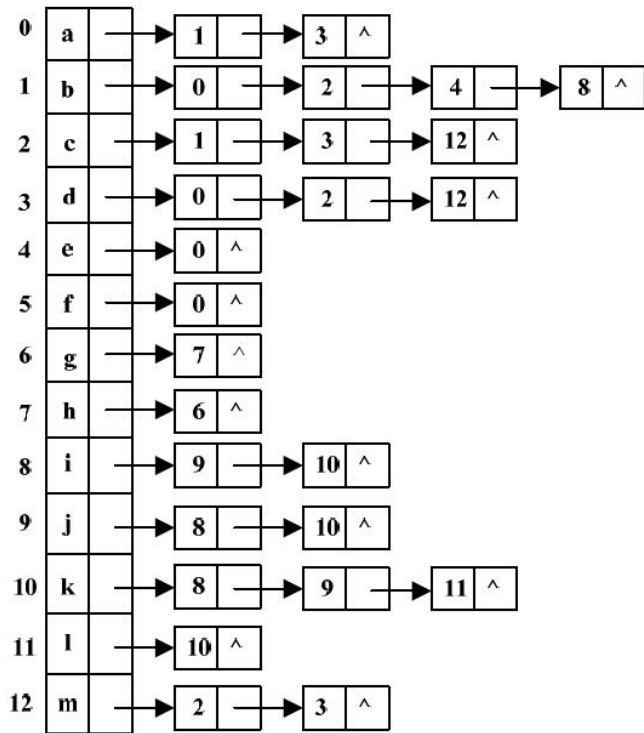
前面介绍了连通图和连通分量的概念，如何判断一个图是否为连通图呢？如何求解一个连通图的连通分量呢？本节讨论这个问题。

10.4.1 无向图的连通分量与最小生成树

在对无向图进行遍历时，对于连通图，仅需从图的任何一个顶点出发进行深度优先搜索遍历或广度优先搜索遍历，就可访问到图中的所有顶点；对于非连通图，则需从多个顶点出发进行搜索，而且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。图10-3中的非连通图 G_3 的邻接表如图10-21所示。图 G_3 是非连通图且有3个连通分量，因此在对图 G_3 进行深度优先遍历时，需要从图的至少3个顶点（顶点a、顶点g和顶点i）出发，才能完成对图中的每个顶点的访问。对图 G_3 进行深度遍历，经过3次递归调用得到的3个序列，分别为a、b、c、d、m、e、f；g、h；i、j、k、l。这3个顶点集分别加上依附于这些顶点的边，就构成了非连通图 G_3 的两个连通分量，如图10-3（b）所示。



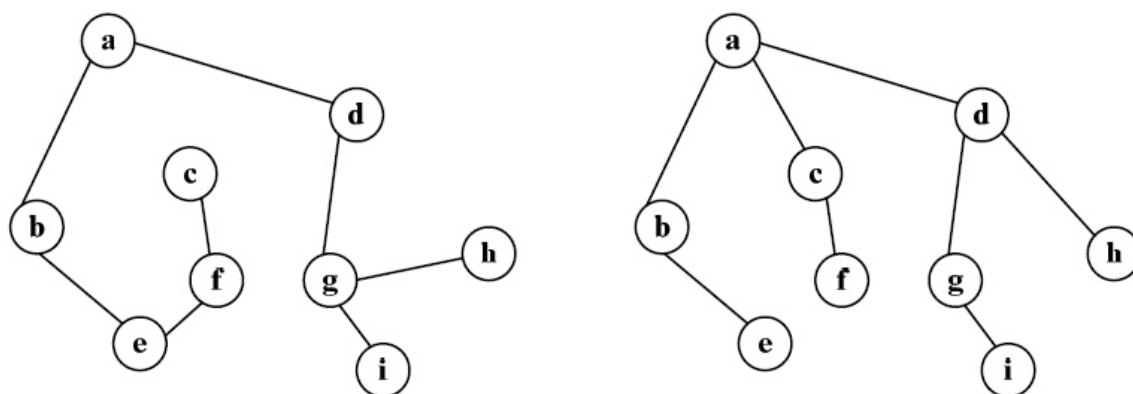
(a) 无向图 G_3



(b) 无向图 G_3 的邻接表

图10-21 图 G_3 的邻接表

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 $T(G)$ 是遍历图过程中经过的边的集合， $B(G)$ 是剩余边的集合。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，根据10.1节的定义，它是连通图的一棵生成树。由深度优先搜索得到的为深度优先生成树，对于连通图，由广度优先搜索得到的为广度优先生成树。图10-22所示就是对应图 G_6 的深度优先生成树和广度优先生成树。



(a) 图 G_6 的深度优先生成树

(b) 图 G_6 的广度优先生成树

图10-22 图 G_6 的深度优先生成树和广度优先生成树

对于非连通图，从某一个顶点出发，对图进行深度优先搜索遍历或者广度优先搜遍历，按照访问路径会得到若干棵生成树，这些生成树放在一起就构成了森林。对图 G_3 进行深度优先搜索得到的森林如图10-23所示。

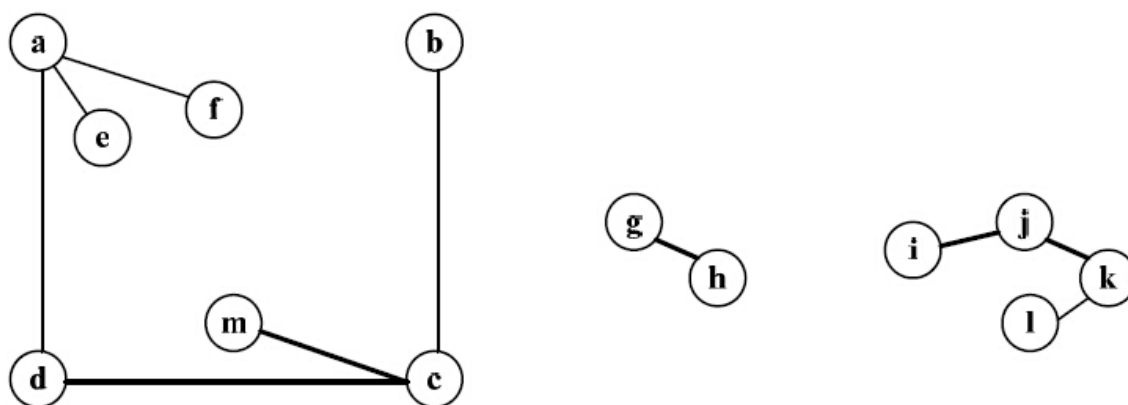


图10-23 图 G_3 的深度优先生成森林

为了判断一个图是否为连通图，可通过对图进行深度优先搜索或广度优先搜索，若调用遍历图的函数不止一次，则说明该图是非连通的，否则该图为连通图。

10.4.2 最小生成树

许多应用问题都是一个求无向连通图的最小生成树问题。假设要在 n 个城市之间铺设光缆，主要目标是要使这 n 个城市的任意两个之间都可以通信，且使铺设光缆的总费用最低。

在每两个城市之间都可以铺设光缆， n 个城市之间最多可能铺设 $n(n-1)/2$ 条光缆，但铺设光缆的费用很高，且各个城市之间铺设光缆的费用不同，那么，如何在这些可能的线路中选择 $n-1$ 条以使总的费用最少呢？

用连通网来表示 n 个城市及 n 个城市间可能铺设的光缆，其中网的顶点表示城市，边表示两个城市之间的光缆线路，赋予边的权值表示相应的造价。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，我们要选择一条这样一棵生成树，也就是使总的造价最少。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**）问题，其中一棵生成树的代价就是树上所有边的代价之和。代价在网中通过权值来表示，一棵生成树的代价就是生成树各边的代价之和。

最小生成树有多种算法，其中大多数算法都利用了最小生成树的MST性质，具体如下。

假设一个连通网 $N=(V, E)$ ， V 是顶点的集合， E 是边的集合， V 有一个非空子集 U 。如果 (u, v) 是一条具有最小权值的边，其中， $u \in U$ ， $v \in V-U$ ，那么一定存在一棵包含边 (u, v) 的最小生成树。

下面用反证法证明以上MST性质。

假设所有最小生成树都不存在这样的一条边 (u, v) 。设 T 是连通网 N 中的一棵最小生成树，如果将边 (u, v) 加入 T 中，根据生成树的定义， T 一定出现包含 (u, v) 的回路。另外， T 中一定存在一条边 (u', v') 的权值大于或等于 (u, v) 的权值，如果删除边 (u', v') ，则得到一棵代价小于或等于 T 的生成树 T' 。 T' 是包含边 (u, v) 的最小生成树，这与假设矛盾。由此，性质得证。

普里姆 (prim) 算法和克鲁斯卡尔 (kruskal) 算法就是利用 MST 性质构造的最小生成树算法。

1. 普里姆算法

假设 $N = \{V, E\}$ 是连通网， TE 是 N 的最小生成树边的集合，执行如下操作。

① 初始时，令 $U = \{u_0\} (u_0 \in V)$ ， $TE = \Phi$ 。

② 对于所有边 $u \in U$ ， $v \in V - U$ 的边 $(u, v) \in E$ ，将一条代价最小的边 (u_0, v_0) 放到集合 TE 中，同时将顶点 v_0 放进集合 U 中。

③ 重复执行步骤 (2)，直到 $U = V$ 为止。

这时，边集合 TE 一定有 $n-1$ 条边， $T = \{V, TE\}$ 就是连通网 N 的最小生成树。

例如，图 10-24 就是利用普里姆算法构造最小生成树的过程。

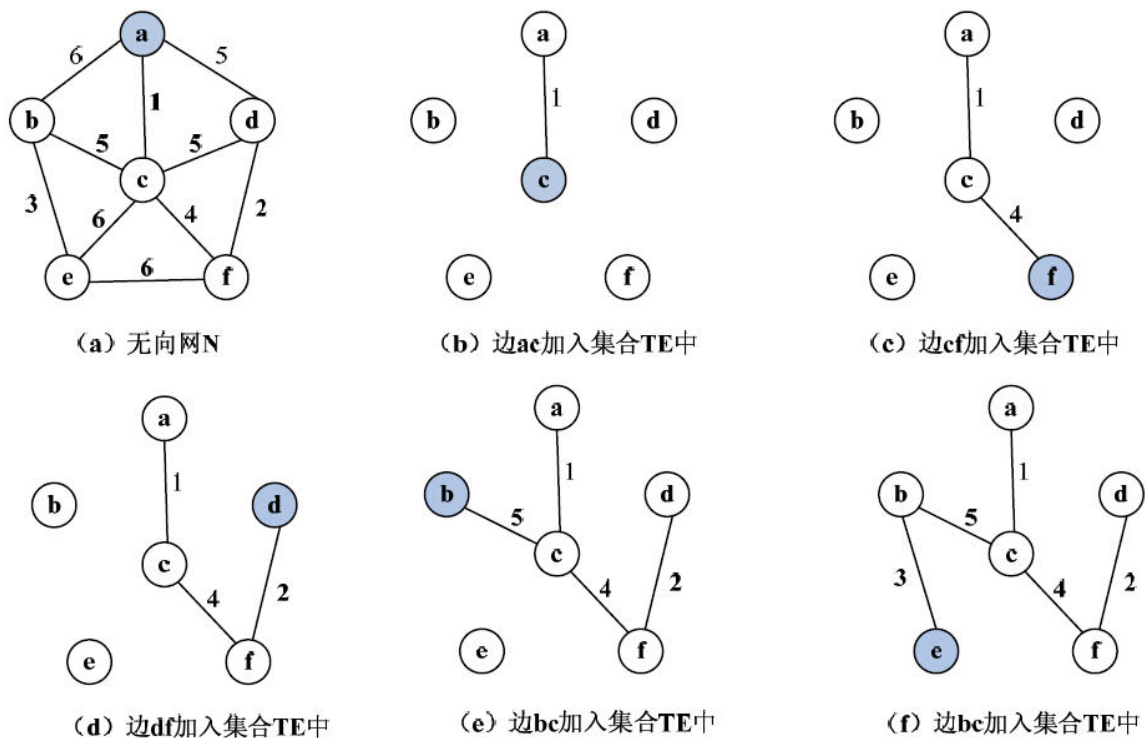


图10-24 利用普里姆算法构造最小生成树的过程

初始时，集合 $U=\{a\}$ ，集合 $V-U=\{b, c, d, e, f\}$ ，边集合为 Φ 。只有一个元素 $a \in U$ ，将 a 从 U 中取出，比较顶点 a 与集合 $V-U$ 中顶点构成的代价最小边，在 (a, b) 、 (a, c) 、 (a, d) 中， (a, c) 的权值最小，故将顶点 c 加入到集合 U 中，边 (a, c) 加入到 TE 中，此时有 $U=\{a, c\}$ ， $V-U=\{b, d, e, f\}$ ， $TE=\{(a, c)\}$ 。目前集合 U 的元素与集合 $V-U$ 的元素构成的所有边为 (a, b) 、 (a, d) 、 (b, c) 、 (c, d) 、 (c, e) 和 (c, f) ，其中代价最小的边为 (c, f) ，故把顶点 f 加入集合 U 中，边 (c, f) 加入 TE 中，此时有 $U=\{a, c, f\}$ ， $V-U=\{b, d, e\}$ ， $TE=\{(a, c), (c, f)\}$ 。依次类推，直到所有顶点都加入 U 中。

为实现这个算法，需附设一个辅助数组 $closeedge[MaxSize]$ ，以记录 U 到 $V-U$ 最小代价的边。对于每个顶点 $v \in V-U$ ，在辅助数组中存在一个相应分

量closeedge[v]，它包括adjvex和lowcost两个域，adjvex域用来表示该边中属于U中的顶点，lowcost域存储该边对应的权值。用公式描述为

$$\text{closeedge}[v].\text{lowcost} = \text{Min}(\{\text{cost}(u, v) \mid u \in U\})$$

根据普里姆算法构造最小生成树，其对应过程中各个参数的变化情况如表10-1所示。

表10-1 普里姆算法各个参数的变化

<div><div><div><div><div></div><div><i>i</i></div></div></div><div><div><div><i>j</i></div></div></div></div></div> <div>closeedge[<i>j</i>]</div>	0	1	2	3	4	5	<i>U</i>	<i>V-U</i>	<i>k</i>	(<i>U</i> ₀ , <i>V</i> ₀)
adjvex lowcost	0	a 6	a 1	a 5			{a}	{b,c,d,e,f}	2	(a,c)

(续)

<div><div><div><div><div></div><div><i>i</i></div></div></div><div><div><div><i>j</i></div></div></div></div></div> <div>closeedge[<i>j</i>]</div>	0	1	2	3	4	5	<i>U</i>	<i>V-U</i>	<i>K</i>	(<i>U</i> ₀ , <i>V</i> ₀)
adjvex lowcost	0	c 5	0	a 5	c 6	c 4	{a,c}	{b,d,e,f}	5	(c,f)
adjvex lowcost	0	c 5	0	f 2	c 6	0	{a,c,f}	{b,d,e}	3	(d,f)
adjvex lowcost	0	c 5	0	0	c 6	0	{a,c,d,f}	{b,e}	1	(b,c)
adjvex lowcost	0	0	0	0	b 3	0	{a,b,c,d,f}	{e}	4	(b,e)
adjvex lowcost	0	0	0	0	0	0	{a,b,c,d,e,f}	{}		

普里姆算法描述如下。

```
/*
记录从顶点集合U
到V-U
的代价最小的边的数组定义*/
typedef struct
{
    VertexType adjvex;
    VRType lowcost;
}closeedge[MaxSize];
void MiniSpanTree_PRIM (MGraph G,VertexType u)
/*
利用普里姆算法求从第u
个顶点出发构造网G
的最小生成树T*/
{
    int i,j,k;
    closeedge closedge;
    k=LocateVertex(G,u);
    ... ..
    /*k
```



```

为顶点u
对应的序号*/
数组初始化*/
    for (j=0;j<G.vexnum;j++)
    {
        strcpy(closedge[j].adjvex,u);
        closedge[j].lowcost=G.arc[k][j].adj;
    }
    closedge[k].lowcost=0;
初始时集合U
只包括顶点u*/
    printf("
最小代价生成树的各条边为:\n");
    for (i=1;i<G.vexnum;i++)
    {
        k=MiniNum(closedge,G);
        printf("(%s-%s)\n",closedge[k].adjvex,G.vex[k]);
        closedge[k].lowcost=0;
        第k
        顶点并入U
        集*/
        for (j=0;j<G.vexnum;j++)
            if (G.arc[k][j].adj<closedge[j].lowcost)
            {
                strcpy(closedge[j].adjvex,G.vex[k]);
                closedge[j].lowcost=G.arc[k][j].adj;
            }
    }
}

```

普里姆算法中有两个嵌套的for循环，假设顶点的个数是n，则第一层循环的频度为n-1，第二层循环的频度为n，因此该算法的时间复杂度为 $O(n^2)$ ，与网中的边数无关，因此普里姆算法适用于求边稠密的最小生成树。

对于图10-23中的网，利用普里姆算法，将输出生成树上的5条边（a，c）、（c，f）、（d，f）、（b，c）、（b，e）。

【例10-3】 创建一个如图10-24所示的无向网N，然后利用普里姆算法求无向网的最小生成树。

【分析】 主要考查普里姆算法求无向网的最小生成树算法。数组closedge有adjvex域和lowcost域两个域，adjvex域用来存放依附于集合U的

顶点，lowcost域用来存放数组下标对应的顶点到顶点（adjvex中的值）的最小权值。因此，查找无向网N中的最小权值的边就是在数组lowcost中找到最小值，输出生成树的边后，要将新的顶点对应的数组值赋值为0，即将新顶点加入集合U。依次类推，直到所有的顶点都加入集合U中。

```
#include "malloc.h"
#include "stdlib.h"
#include "stdio.h"
#include <string.h>
#define MAX_VERTEX_NUM 20 /*
顶点个数的最大值*/
#define MAX_NAME 3 /*
顶点字符串的最大长度+1 */
#define INFINITY 65535 /*65535
代表整型最大值*/
typedef int VRType;
typedef char InfoType;
typedef char VertexType[MAX_NAME];
//
邻接矩阵的数据结构
typedef struct
{
    VRType adj; //
    顶点关系类型。对无权图，用1(
    是)
    或0(
    否)
    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(
    可无)
    } ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
//
图的数据结构
typedef struct
{
    VertexType vexs[MAX_VERTEX_NUM]; //
    顶点向量
    AdjMatrix arcs; //
    邻接矩阵
    int vexnum, //
    图的当前顶点数
    arcnum; //
    图的当前弧数
} MGraph;
//
记录从顶点集U
到v-u
的代价最小的边的辅助数组定义
typedef struct
{
    VertexType adjvex;
    VRType lowcost;
} Closedge[MAX_VERTEX_NUM];
//
若G
中存在顶点u,
则返回该顶点在图中位置;
否则返回-1
。
```

```

int LocateVex(MGraph G,VertexType u)
{
    int i;
    for(i = 0; i < G.vexnum; ++i)
        if( strcmp(u, G.vexs[i]) == 0)
            return i;
    return -1;
}
//
采用数组 (
邻接矩阵)
表示法,
构造无向网G
。
int CreateAN(MGraph *G)
{
    int i,j,k,w;
    VertexType va,vb;
    printf("

请输入无向网G
的顶点数和边数 (
以空格作为间隔): ");
    scanf("%d%d%c",&(*G).vexnum,&(*G).arcnum);
    printf("

请输入%d
个顶点的值 (
每个顶点的字符数<=d
个字符):\n",(*G).vexnum,MAX_NAME);
    for(i=0;i<(*G).vexnum;++i)
        //
        scanf("%s",(*G).vexs[i]);
    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].adj=INFINITY;    //
            (*G).arcs[i][j].info=NULL;
        }
    printf("

请输入%d
条边的顶点1
顶点2
权值 (
以空格作为间隔): \n",(*G).arcnum);
    for(k=0;k<(*G).arcnum;++k)
    {
        scanf("%s%d%c",va,vb,&w);
        // %*c
        吃掉回车符
        i=LocateVex(*G,va);
        j=LocateVex(*G,vb);
        (*G).arcs[i][j].adj=(*G).arcs[j][i].adj=w; //
        无向图
    }
    return 1;
}
//
求closedge.lowcost
的最小正值
int MiniNum(Closededge edge,MGraph G)
{
    int i=0,j,k,min;
    while(!edge[i].lowcost)
        i++;
    min=edge[i].lowcost;
    //
    第一个不为0
    的值
    k=i;
    for(j=i+1;j<G.vexnum;j++)
        if(edge[j].lowcost>0)
            if(min>edge[j].lowcost)
            {

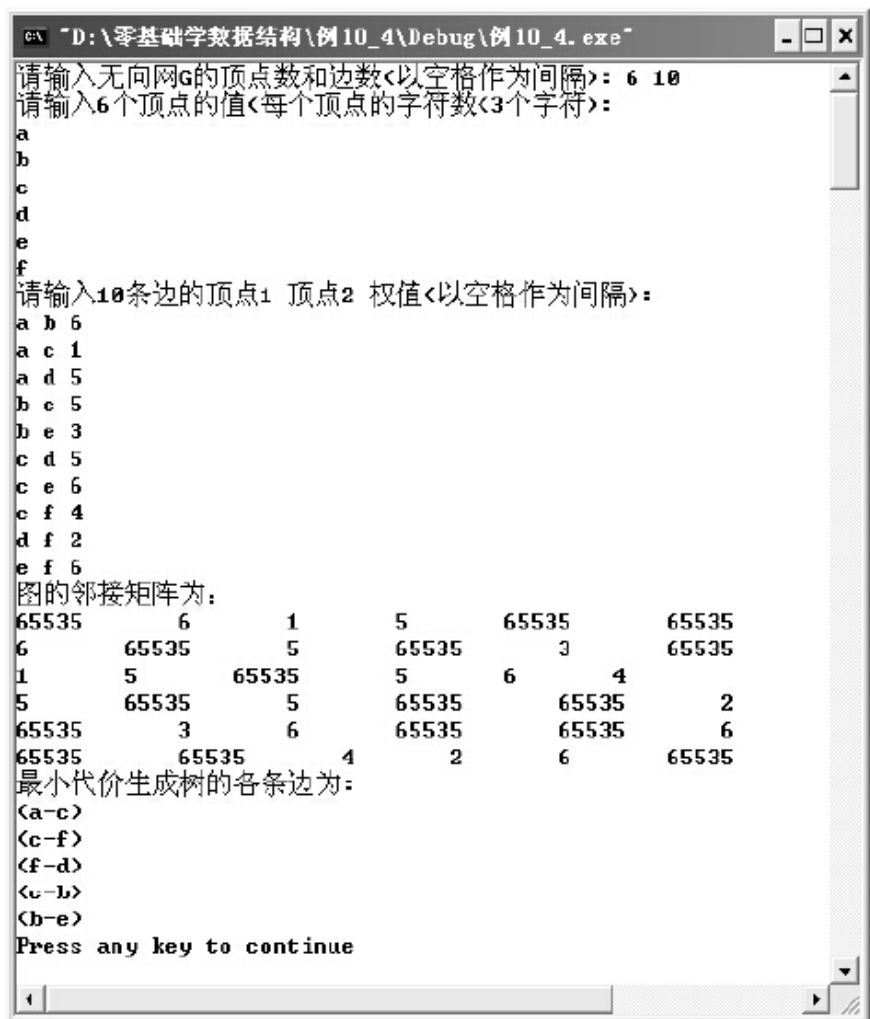
```

```

        min=edge[j].lowcost;
        k=j;
    }
    return k;
}
//
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
void MiniSpanTree_PRIM(MGraph G,VertexType u)
{
    int i,j,k;
    Closededge closededge;
    k=LocateVex(G,u);
    for (j=0;j<G.vexnum;++j)
        //
        // 辅助数组初始化
        {
            if(j!=k)
            {
                strcpy(closededge[j].adjvex,u);
                closededge[j].lowcost=G.arcs[k][j].adj;
            }
        }
        closededge[k].lowcost=0;
        //
        // 初始时,U={u}
        printf("
        最小代价生成树的各条边为:\n");
        for (i=1;i<G.vexnum;++i)
        { //
            // 选择其余G.vexnum-1
            // 个顶点
            k=MiniNum(closededge,G);
            //
            // 求出T
            // 的下一个结点: 第k
            // 顶点
            printf("(%s-%s)\n",closededge[k].adjvex,G.vexs[k]); //
            // 输出生成树的边
            closededge[k].lowcost=0;
            //
            // 第k
            // 顶点并入U
            // 集
            for (j=0;j<G.vexnum;++j)
                if(G.arcs[k][j].adj<closededge[j].lowcost)
                {
                    //
                    // 新顶点并入U
                    // 集后重新选择最小边
                    strcpy(closededge[j].adjvex,G.vexs[k]);
                    closededge[j].lowcost=G.arcs[k][j].adj;
                }
        }
    }
}
void main()
{
    int i,j;
    MGraph G;
    CreateAN(&G);
    printf("
    图的邻接矩阵为: \n");
    for(i=0; i<G.vexnum; i++)
    {
        for(j=0; j<G.vexnum; j++)
            printf("%d\t",G.arcs[i][j].adj);
        printf("\n");
    }
    MiniSpanTree_PRIM(G,G.vexs[0]);
}

```

程序运行结果如图10-25所示。



```
C:\ "D:\零基础学数据结构\例10_4\Debug\例10_4.exe"
请输入无向网G的顶点数和边数<以空格作为间隔>: 6 10
请输入6个顶点的值<每个顶点的字符数<3个字符>:
a
b
c
d
e
f
请输入10条边的顶点1 顶点2 权值<以空格作为间隔>:
a b 6
a c 1
a d 5
b c 5
b e 3
c d 5
c e 6
c f 4
d f 2
e f 6
图的邻接矩阵为:
65535      6      1      5      65535      65535
6      65535      5      65535      3      65535
1      5      65535      5      6      4
5      65535      5      65535      65535      2
65535      3      6      65535      65535      6
65535      65535      4      2      6      65535
最小代价生成树的各条边为:
<a-c>
<c-f>
<f-d>
<c-b>
<b-e>
Press any key to continue
```

图10-25 普里姆算法运行结果

这个程序中最难理解的是数组closedge的变化，数组closedge的adjvex域存放的是每条边的起始点名字，closedge的下标表示每条边的终结点下标，closedge表示边的最小代价。在MiniSpanTree_PRIM算法中，先确定集合U的元素，即出发顶点，代码如下。

```
k=LocateVex(G,u);
```

然后对closedge进行初始化，即把初始顶点赋给closedge的adjvex域，最小代价边赋给lowcost域，并将第一个顶点加入U集。例如，当u=a时，k=0，初始化后，closedge数组的值如图10-25（a）所示，代码如下。

```
        for (j=0;j<G.vexnum;++j) //
辅助数组初始化
        {
            if (j!=k)
            {
                strcpy(closedge[j].adjvex,u);
                closedge[j].lowcost=G.arcs[k][j].adj;
            }
        }
        closedge[k].lowcost=0; //
初始时,U={u}
```

调用函数MiniNum求代价最小的边，k为终结点的序号，例如，图10-24中的无向网N第一条代价最小的边为（a，c），则有k=2，所以closedge[2].adjvex为a，G.vexs[2]为c，输出（a，c），并将第k个顶点加入U集，代码如下。

```
        for (i=1;i<G.vexnum;++i)
        { //
选择其余G.vexnum-1
个顶点
            k=MiniNum(closedge,G); //
        求出T
        的下一个结点：第k
        顶点
            printf("(%s-%s)\n",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
```

更新closedge数组中的lowcost域，使其为当前状况下边的代价最小，代码如下。

```
        for (j=0;j<G.vexnum;++j)
            if (G.arcs[k][j].adj<closedge[j].lowcost)
            {
                //
新顶点并入U
        ... ..
```

集后重新选择最小边

```
        strcpy(closedge[j].adjvex,G.vexs[k]);  
        closedge[j].lowcost=G.arcs[k][j].adj;  
    }  
}
```

数组closedge的adjvex域和lowcost域的变化情况如图10-26所示。

2. 克鲁斯卡尔算法

克鲁斯卡尔算法从另一途径求网的最小生成树，连通网为 $N=\{V, E\}$ ，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 中不同的连通分量中，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 中所有顶点都在同一连通分量上为止。

例如，图10-24所示的无向图 N 利用卡鲁斯卡尔算法构造最小生成树的过程如图10-27所示。

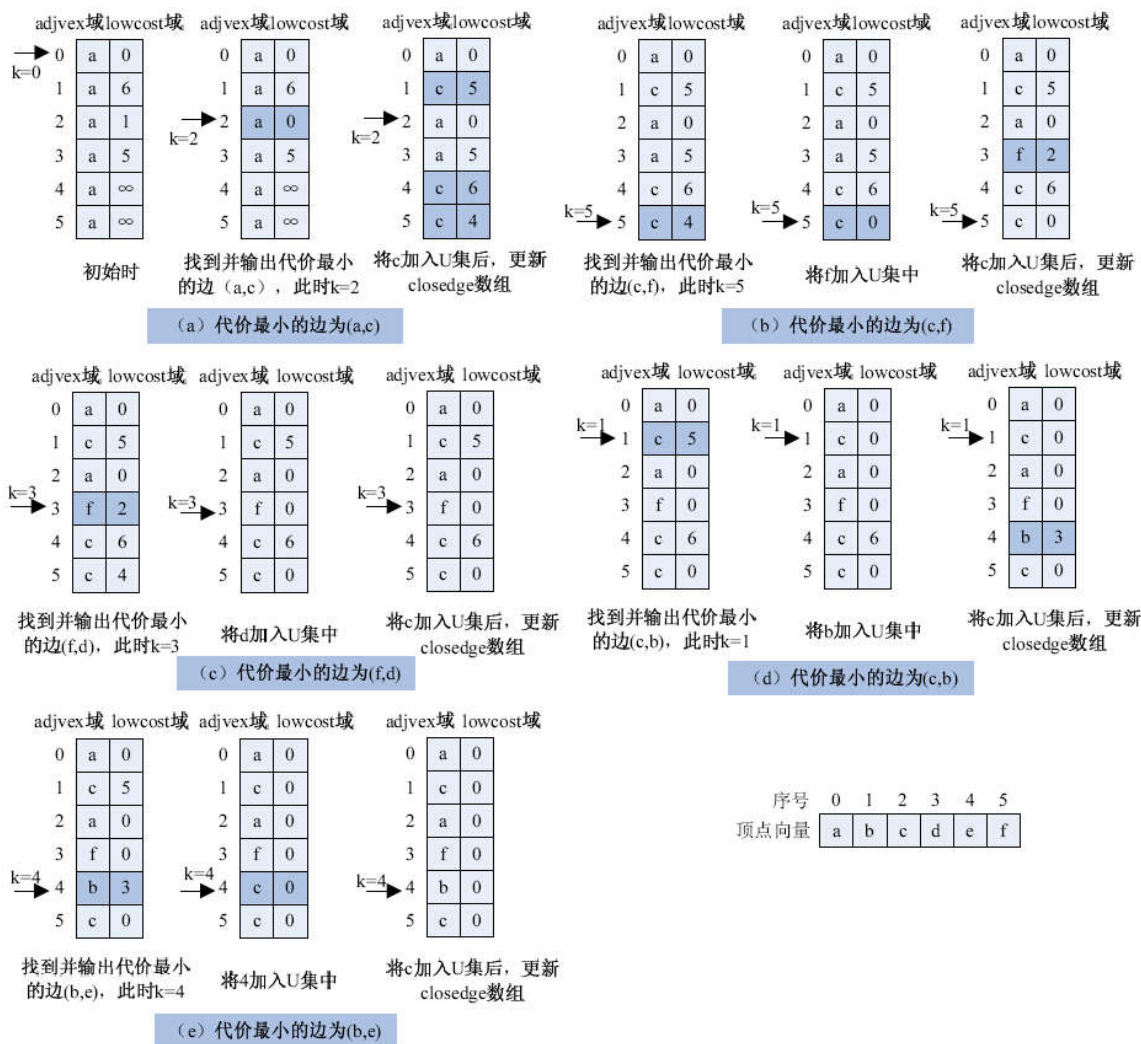


图10-26 数组closedge的变化情况

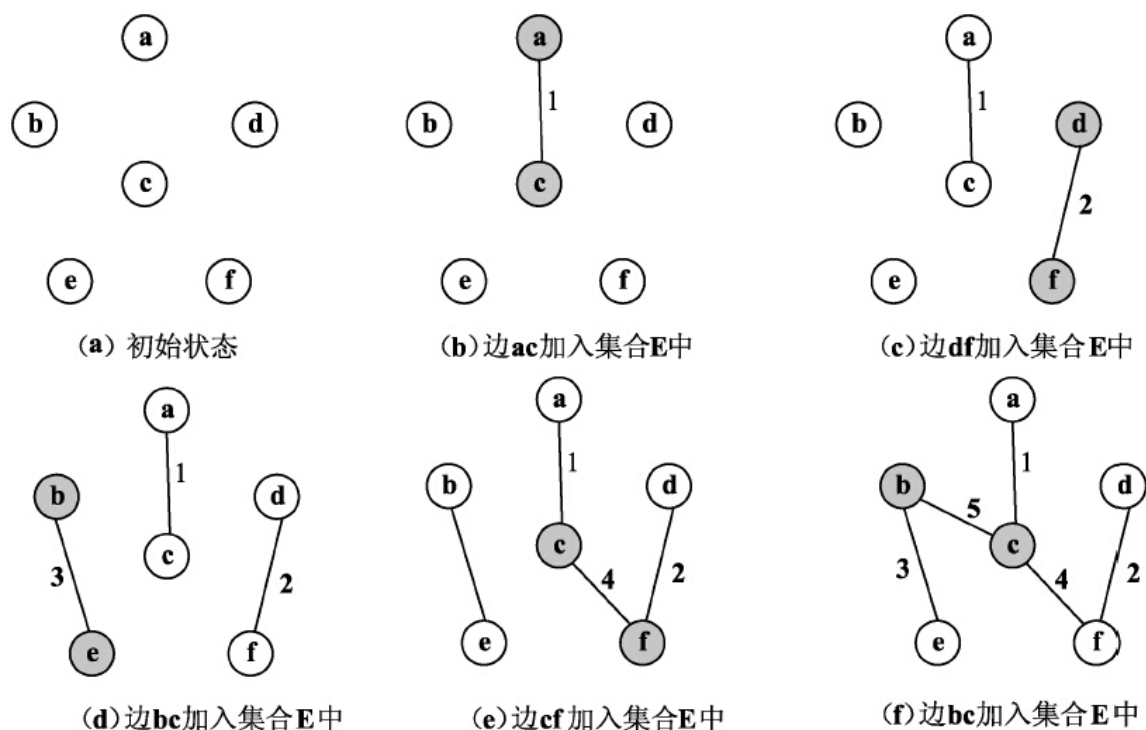


图10-27 克鲁斯卡尔算法构造最小生成树的过程

初始时，边的集合 E 为空集，顶点 a 、 b 、 c 、 d 、 e 、 f 分别属于不同的集合集合，假设 $U_1 = \{a\}$ ， $U_2 = \{b\}$ ， $U_3 = \{c\}$ 、 $U_4 = \{d\}$ 、 $U_5 = \{e\}$ 、 $U_6 = \{f\}$ 。图中含有10条边，将这10条边按照权值从小到大排列，依次取出最小的边且依附于边的两个顶点属于不同的集合，则将该边加入集合 E 中，并将这两个顶点合并为一个集合，重复执行类似操作直到所有顶点都属于一个集合为止。

这10条边中，权值最小的是边 (a, c) ，其权值 $\text{cost}(a, c) = 1$ ，并且 $a \in U_1$ ， $c \in U_3$ ， U_1 和 U_3 分别属于不同的集合，故将边 (a, c) 加入集合 E 中，并将两个顶点所在的集合归并为一个集合， $E = \{(a, c)\}$ ， $U_1 = U_3 = \{a, c\}$ 。在剩下的边的集合中，边 (d, f) 权值最小，且 $d \in U_4$ ， $f \in U_5$ ， $U_3 \neq U_4$ ，因此，将边 (d, f) 加入边的集合 E 中，合并顶点集

合，有 $E=\{(a, c), (d, f)\}$ ， $U_1=U_3=U_4=U_6=\{a, c, d, f\}$ 。然后继续从剩下的边的集合中选择权值最小的边，依次加入 E 中，并合并顶点集合，直到所有的顶点都属于同一顶点集合。

卡鲁斯卡尔算法描述如下。

```
void Kruskal(MGraph G)
/*
克鲁斯卡尔算法求最小生成树*/
{
    int set[MaxSize], i, j;
    int a=0, b=0, min=G.arc[a][b].adj, k=0;
    for (i=0; i<G.vexnum; i++) /*
初始时，各顶点分别属于不同的集合*/
        set[i]=i;
    printf("
最小生成树的各条边为:\n");
    while (k<G.vexnum-1) /*
查找所有最小权值的边*/
    {
        for (i=0; i<G.vexnum; i++) /*
在矩阵的上三角查找最小权值的边*/
            for (j=i+1; j<G.vexnum; j++)
                if (G.arc[i][j].adj<min)
                {
                    min=G.arc[i][j].adj;
                    a=i;
                    b=j;
                }
        min=G.arc[a][b].adj=INFINITY; /*
删除上三角中最小权值的边，下次不再查找*/
        if (set[a]!=set[b]) /*
如果边的两个顶点在不同的集合*/
        {
            printf("%s-%s\n", G.vex[a], G.vex[b]); /*
输出最小权值的边*/
            k++;
            for (i=0; i<G.vexnum; i++)
                if (set[i]==set[b]) /*
将顶点b
所在集合并入顶点a
集合中 */
                    set[i]=set[a];
        }
    }
}
```

克鲁斯卡尔算法的时间复杂度为 $O(e \log e)$ （其中， e 为网中边的数目），因此，它相对于普里姆算法来说，适合于求边稀疏的最小生成树。

10.5 有向无环图

一个无环的有向图被称为有向无环图（Directed Acycline Graph, DAG），有向无环图可用来描述工程或系统的进行过程，如一个工程的施工过程图、学生学习课程的制约关系图等。

10.5.1 AOV网与拓扑排序

由AOV网可以得到拓扑排序。在学习拓扑排序之前，先来介绍一下AOV网。

1. 什么是AOV网

几乎所有工程都可分为若干个称为活动的子工程，而这些子工程之间通常受一些条件的制约，如某些子工程的开始必须在另一些子工程完成之后才能进行。用图的顶点表示活动，用弧表示活动之间的优先关系的有向无环图称为AOV网（activity on vertex network），即顶点表示活动的网。

在AOV网中，若从顶点 v_i 到顶点 v_j 之间存在一条有向路径，则顶点 v_i 是顶点 v_j 的前驱，顶点 v_j 为顶点 v_i 的后继。若 $\langle v_i, v_j \rangle$ 是有向网的一条弧，则称顶点 v_i 是顶点 v_j 的直接前驱，顶点 v_j 是顶点 v_i 的直接后继。

例如，一个软件工程专业的学生必须修完一系列基本课程才能毕业，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它的基础先修课程才能开始，如《数据结构》是在学习完《程序设计基础》和《离散数学》之后才能开始学习。这些先决条件定义了课程之间的优先次序。例如软件工程专业的课程及先决条件如表10-2所示。

表10-2 软件工程专业课程关系表

课 程 编 号	课 程 名 称	先修课程编号
C ₁	高等数学	无
C ₂	程序设计基础	无
C ₃	离散数学	C ₁ ,C ₂
C ₄	数据结构	C ₂ ,C ₃
C ₅	算法设计与分析	C ₂ ,C ₄
C ₆	普通物理	C ₁
C ₇	计算机组成原理	C ₆
C ₈	操作系统	C ₄ ,C ₇
C ₉	编译原理	C ₄ ,C ₅
C ₁₀	线性代数	C ₁

这些课程之间的关系利用有向图可以更清楚地表示，如图10-28所示。

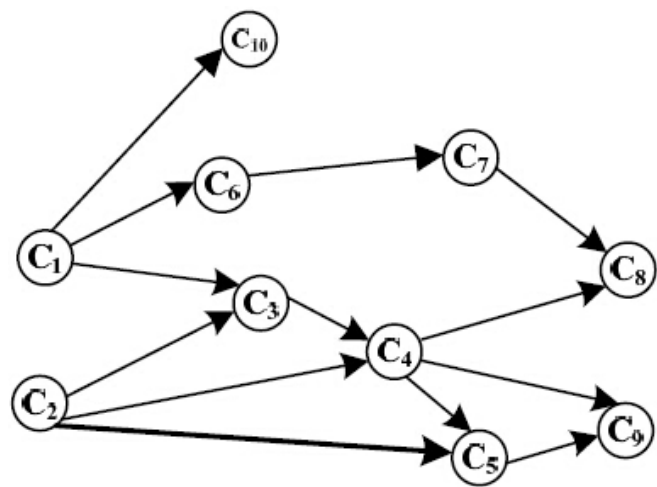


图10-28 表示课程之间优先关系的有向图

在AOV网中，不应该出现有向环，因为存在环意味着某项活动以自己为先决条件，显然这是不可能的。若设计出这样的流程图，工程就无法进行；对于程序的流程图来说，就是一个死循环。因此，对给定的有向图，应首先判断网中是否存在环，检测的办法就要利用有向图的拓扑排序知识了。

2. 拓扑排序

拓扑排序的方法如下。

①在有向图中任意选择一个没有前驱的顶点即入度为零的顶点，将该顶点输出。

②从图中删除该顶点和所有以它为尾的弧。

③重复执行步骤①和②，直至所有顶点均已被输出或者当前图中不存在无前驱的顶点为止（说明有向图中存在环）。

按照以上方法，可得到图10-28所示的有向图的两个拓扑序列（当然还可构造其他拓扑序列）为（ $C_1, C_6, C_{10}, C_7, C_2, C_3, C_4, C_5, C_8, C_9$ ）和（ $C_2, C_1, C_3, C_4, C_5, C_9, C_{10}, C_6, C_7, C_8$ ）图10-29展示了一个完整的拓扑序列的构造过程，其拓扑序列为 $V_1, V_2, V_4, V_3, V_5, V_6$ 。

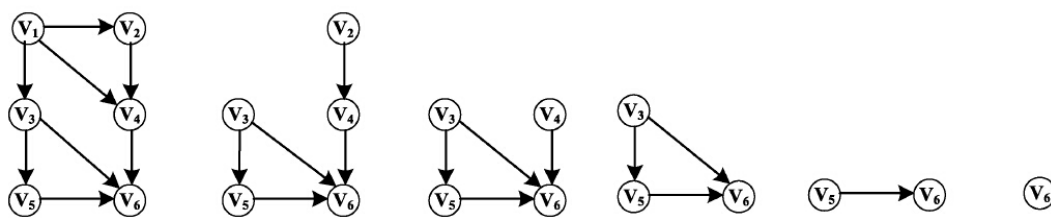


图10-29 AOV网构造拓扑序列的过程

对有向图拓扑排序后，如果图中的顶点全部输出，表示图中不存在回路，则表明该有向图为AOV网；如果图中还存在未输出的顶点，表示图中存在回路。

针对以上算法步骤，可采用邻接表作为有向图的存储结构，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，算法实现：遍历邻接表，将各个顶点的入度保存在数组indegree中；将入度为零的顶点入栈，依次将栈顶元素出栈并输出该顶点，对该顶点的邻接顶点的入度减1，如果邻

接顶点的入度为零，则入栈，否则将下一个邻接顶点的入度减1并进行相同的处理；然后继续将栈中元素出栈；重复执行以上操作，直到栈空为止。

有向图的拓扑排序算法描述如下。

```

int TopologicalSort(AdjGraph G)
/*
有向图G
的拓扑排序。如果图G
没有回路，则输出G
的一个拓扑序列并返回1
，否则返回0*/
{
    int i,k,count=0;
    int indegree[MaxSize];          /*
存放各顶点当前入度*/
    SeqStack S;
    ArcNode *p;
    /*
将图中各顶点的入度保存在数组indegree
中*/
    for(i=0;i<G.vexnum;i++)          /*
将数组indegree
赋初值*/
    {
        indegree[i]=0;
        for(i=0;i<G.vexnum;i++)
        {
            p=G.vertex[i].firstarc;
            while(p!=NULL)
            {
                k=p->adjvex;
                indegree[k]++;
                p=p->nextarc;
            }
        }
    }
    /*
对图G
进行拓扑排序*/
    InitStack(&S);                    /*
初始化栈S*/
    for(i=0;i<G.vexnum;i++)          /*
将所有入度为零的顶点入栈*/
    {
        if(!indegree[i])
            PushStack(&S,i);
        while(!StackEmpty(S))        /*
如果栈S
不为空，则将栈顶元素出栈，输出该顶点*/
        {
            PopStack(&S,&i);          /*
将栈顶元素出栈*/
            printf("%s ",G.vertex[i].data); /*
输出编号为i
的顶点*/
            count++;                  /*
将已输出顶点数加1*/
            for(p=G.vertex[i].firstarc;p=p->nextarc) /*
处理编号为i
的顶点的所有邻接顶点*/
            {
                k=p->data.adjvex;
                if(!(--indegree[k]))    /*
如果编号为i
的邻接顶点的入度减1
后变为0
，则将其入栈*/
                PushStack(&S,k);
            }
        }
    }
    if(count<G.vexnum)                /*

```

```

图G
中还有未输出的顶点，则存在回路，否则可以构成一个拓扑序列*/
{
    printf("
该有向图有回路\n");
    return 0;
}
else
{
    printf("
该图可以构成一个拓扑序列.\n");
    return 1;
}
}

```

对有 n 个顶点和 e 条弧的有向图来说，建立求各顶点的入度的时间复杂度为 $O(e)$ ，将零入度的顶点入栈的时间复杂度为 $O(n)$ ；在拓扑排序过程中，若有有向图无环，则每个顶点进一次栈，出一次栈，入度减1操作在while语句中总共执行 e 次，因此，拓扑排序总的时间复杂度为 $O(n+e)$ 。

10.5.2 AOE网与关键路径

AOV网描述了活动之间的优先关系，是一个定性的研究，而AOE网就是一个定量的研究。如整个工程的最短完成时间、各个子工程影响整个工程的程度、每个子工程的最短完成时间和最长完成时间，都需要利用AOE网的相关知识来解决，通过研究事件与活动之间的关系，从而可以确定整个工程的最短完成时间，明确活动之间的相互影响，确保整个工程的顺利进行。

1. 什么是AOE网

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，顶点表示事件（event），弧表示活动，权表示活动持续的时间，权值表示子工程的活动需要的时间。通常可用AOE网估算工程的完成时间。

图10-30是一个具有11个活动的AOE网，其中 v_1, v_2, \dots, v_9 表示9个事件， $\langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \dots, \langle v_8, v_9 \rangle$ 表示11个活动， a_1, a_2

、...、 a_{11} 表示活动的执行时间。进入顶点的有向弧表示的活动已经完成，从顶点出发的有向弧表示的活动可以开始。顶点 v_1 表示整个工程的开始， v_9 表示整个工程的结束。顶点 v_5 表示活动 a_4 、 a_5 已经完成，活动 a_7 和 a_8 可以开始。完成活动 a_1 和活动 a_2 分别需要6天和4天。

在某事件（顶点表示）发生之后，活动（该顶点出发的有向弧）才能开始。活动完成之后，之后的事件才会发生。

由于整个工程只有一个开始点和一个完成点，对于AOE网来说，网中只有一个入度为零的点（称为源点）和一个出度为零的点（称为汇点）。

2. 关键路径

AOE网需要研究的问题是完成整个工程至少需要多少时间？以及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个活动持续时间之和。最长的路径就是[关键路径](#)（critical path）。在AOE网中，关键路径其实就是完成工程的最短时间所经过的路径。关键路径表示了完成工程的最短工期。

下面是和关键路径有关的几个概念。

(1) 事件 v_i 的最早发生时间 $ve(i)$ ：从源点到顶点 v_i 的最长路径长度，称为事件 v_i 的最早发生时间，记作 $ve(i)$ 。求解 $ve(i)$ 可以从源点 $ve(0)=0$ 开始，按照拓扑排序规则根据递推得到，即 $ve(i) = \max\{ve(k) + dut(\langle k, i \rangle) \mid \langle k, i \rangle \in T, 1 \leq i \leq n-1\}$ ，其中， T 是所有以第 i 个顶点为弧头的弧的集合， $dut(\langle k, i \rangle)$ 表示弧 $\langle k, i \rangle$ 对应的活动的持续时间。例如，已知 v_2 的最早发生

时间为 $ve(2)=6$ ， v_3 的最早发生时间为 $ve(3)=4$ ，活动 a_4 和 a_5 的持续时间为1，故 v_5 的最早发生时间为 $ve(5)=\text{Max}(6+1, 4+1)=7$ ，如图10-31所示。

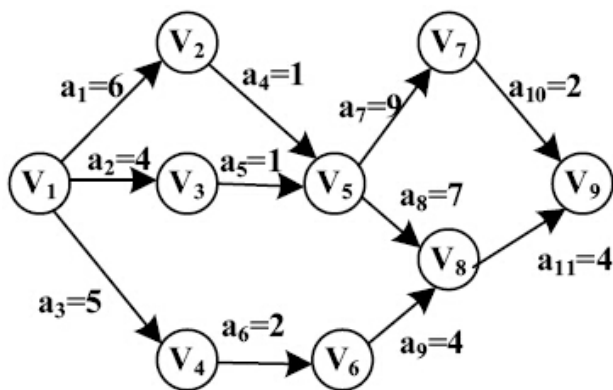


图10-30 一个AOE网

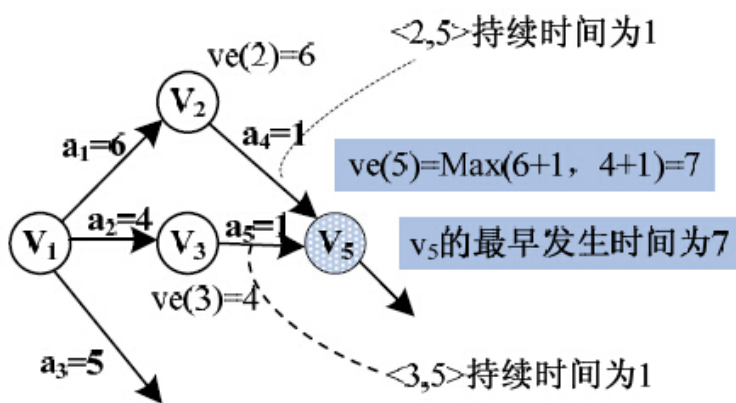


图10-31 v_5 的最早发生时间

(2) 事件 v_i 的最晚发生时间 $vl(i)$ ：在保证整个工程正常完成的前提下，活动的最迟开始时间，记作 $vl(i)$ 。在求解事件 v_i 的最早发生时间 $ve(i)$ 的前提下 $vl(n-1)=ve(n-1)$ 下，从汇点开始，向源点推进得到 $vl(i)=\text{Min}\{vl(k)-\text{dut}(\langle i, k \rangle) \mid \langle i, k \rangle \in S, 0 \leq i \leq n-2\}$ ，其中， S 是所有以第 i 个顶点为弧尾的弧的集合， $\text{dut}(\langle i, k \rangle)$ 表示弧 $\langle i, k \rangle$ 对应的活动的持续时间。

(3) 活动 a_i 的最早开始时间 $e(i)$ ：如果弧 $\langle v_k, v_j \rangle$ 表示活动 a_i ，当事件 v_k 发生之后，活动 a_i 才开始。因此，事件 v_k 的最早发生时间也就是活

动 a_i 的最早开始时间，即 $e(i) = ve(k)$ 。

(4) 活动 a_i 的最晚开始时间 $l(i)$ ：在不推迟整个工程完成时间的基础上，活动 a_i 最迟必须开始的时间。如果弧 $\langle v_k, v_j \rangle$ 表示活动 a_i 持续时间为 $dut(\langle k, j \rangle)$ ，则活动 a_i 的最晚开始时间 $l(i) = vl(j) - dut(\langle k, j \rangle)$ 。例如，因事件 v_8 的最晚开始时间为14，活动 a_8 的持续时间为7，所以 a_8 的最晚开始时间为 $14 - 7 = 7$ 。如图10-32所示。

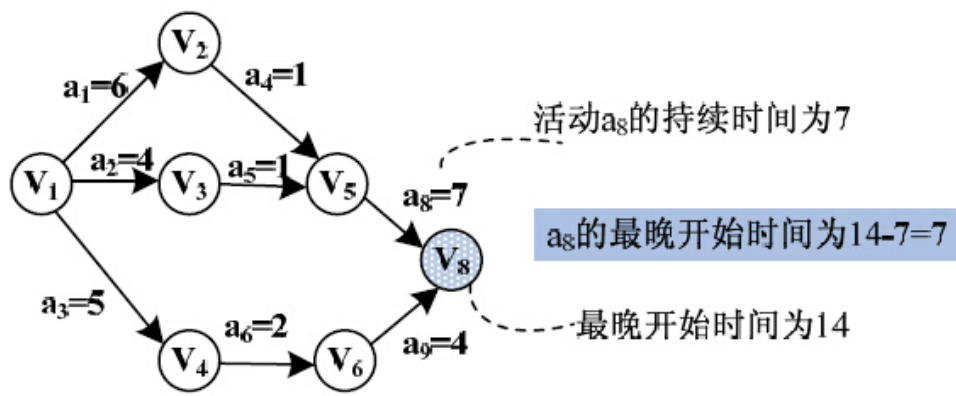


图10-32 活动 a_8 的最晚开始时间

(5) 活动 a_i 的松弛时间：活动 a_i 的最晚开始时间与最早开始时间之差，记做 $l(i) - e(i)$ 。

在图10-30所示的AOE网中，从源点 v_1 到汇点 v_9 的关键路径是 $(v_1, v_2, v_5, v_8, v_9)$ ，路径长度为18，也就是说 v_9 的最早发生时间为18。活动 a_6 的最早开始时间是5，最晚开始时间是8，这意味着，如果 a_6 推迟3天开始或延迟3天完成，都不会影响到整个工程的进度。

当 $e(i) = l(i)$ 时，对应的活动 a_i 称为关键活动。在关键路径上的所有活动都称为关键活动，非关键活动提前完成或推迟完成并不会影响到整个工程的进度。例如，活动 a_6 是非关键活动， a_8 是关键活动。

求关键路径的算法如下。

①对网中的顶点进行拓扑排序，如果得到的拓扑序列顶点个数小于网中顶点数，则说明网中有环存在，不能求关键路径，终止算法；否则从源点 v_0 开始，求出各个顶点的最早发生时间 $ve(i)$ 。

②从汇点 v_n 出发 $vl(n-1)=ve(n-1)$ ，按照逆拓扑序列求其他顶点的最晚发生时间 $vl(i)$ 。

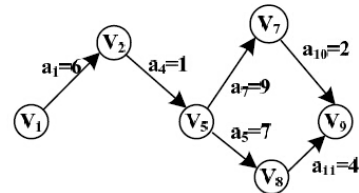
③由各顶点的最早发生时间 $ve(i)$ 和最晚发生时间 $vl(i)$ ，求出每个活动 a_i 的最早开始时间 $e(i)$ 和最晚开始时间 $l(i)$ 。

④找出所有满足条件 $e(i)=l(i)$ 的活动 a_i ， a_i 即是关键活动。

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：（1）在拓扑排序之前设置初值，令 $ve[i]=0$ ；（2）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 $ve[j]+dut(<i, k>)>ve[k]$ ，则 $ve[k]=ve[j]+dut(<i, k>)$ ；（3）为了能按逆拓扑排序序列计算各顶点的 vl 值，需记下在拓扑排序的过程中求得的拓扑有序序列，这需要再拓扑排序算法中，增加一个记录逆拓扑有序序列，则在计算求得各顶点的 ve 值后，从栈顶到栈底便是逆拓扑有序序列。

利用AOE网的关键路径算法，图10-30所示的网中顶点对应事件最早发生时间 ve 、最晚发生时间 vl 及弧对应活动最早发生时间 e 、最晚发生时间如图10-33所示。

顶点	ve	vl	活动	e	l	l-e
v ₁	0	0	a ₁	0	0	0
v ₂	6	6	a ₂	0	2	2
v ₃	4	6	a ₃	0	3	3
v ₄	5	8	a ₄	6	6	0
v ₅	7	7	a ₅	4	6	2
v ₆	7	10	a ₆	5	8	3
v ₇	16	16	a ₇	7	7	0
v ₈	14	14	a ₈	7	7	0
v ₉	18	18	a ₉	7	14	7
			a ₁₀	16	16	0
			a ₁₁	14	14	0



两条关键路径(v₁,v₂,v₅,v₇,v₉)和(v₁,v₂,v₅,v₈,v₉)

图10-33 图10-29所示AOE网顶点发生时间与活动的开始时间

显然，网的关键路径有（v₁，v₂，v₅，v₈，v₉）和（v₁，v₂，v₅，v₇，v₉）两条，对应的关键活动是a₁、a₄、a₅、a₁₁和a₁、a₄、a₇和a₁₀。

关键路径经过的顶点满足条件 $ve(i) = vl(i)$ ，即当事件的最早发生时间与最晚发生时间相等时，该顶点一定在关键路径之上。同样，关键活动的弧满足条件 $e(i) = l(i)$ ，即当活动的最早开始时间与最晚开始时间相等时，该活动一定是关键活动。

求每一个顶点的最早开始时间，首先要将网中的顶点进行拓扑排序。在对顶点进行拓扑排序过程中，同时计算顶点的最早发生时间 $ve(i)$ 。从源点开始，由与源点相关联的弧的权值，可以得到该弧相关联顶点对应事件的最早发生时间。同时定义一个栈T，保存顶点的逆拓扑序列。拓扑排序和求 $ve(i)$ 的算法实现如下。

```

int TopologicalOrder(AdjGraph N,SeqStack *T)
/*
采用邻接表存储结构的有向网N
的拓扑排序，并求各顶点对应事件的最早发生时间ve*/
/*
如果N
无回路，则用栈T
返回N
的一个拓扑序列，
并返回1，
否则为0*/
{

```

```

        int i,k,count=0;
        int indegree[MaxSize];
        /*
数组indegree
存储各顶点的入度*/
        SeqStack S;
        ArcNode *p;
        /*
将图中各顶点的入度保存在数组indegree
中*/
        FindInDegree(N,indegree);
        InitStack(&S);
        /*
初始栈S*/
        for(i=0;i<N.vexnum;i++)
        if(!indegree[i])
            /*
将入度为零的顶点入栈*/
            PushStack(&S,i);
        InitStack(T);
        /*
初始化拓扑序列顶点栈*/
        for(i=0;i<N.vexnum;i++)
            /*
初始化ve*/
            ve[i]=0;
        while(!StackEmpty(S))
            /*
如果栈S
不为空*/
            {
                PopStack(&S,&i);
                /*
从栈S
将已拓扑排序的顶点j
弹出*/
                printf("%s ",N.vertex[i].data);
                PushStack(T,i);
                /*j
号顶点入逆拓扑排序栈T*/
                count++;
                /*
对入栈T
的顶点计数*/
                for(p=N.vertex[i].firstarc;p=p->nextarc)
                    /*
处理序号为i
的顶点的每个邻接点*/
                    {
                        k=p->adjvex;
                        /*
顶点序号为k*/
                        if(--indegree[k]==0)
                            /*
如果k
的入度减1
后变为0
, 则将k
入栈S*/
                            PushStack(&S,k);
                        if(ve[i]+*(p->info)>ve[k])
                            /*
计算顶点k
对应的事件的最早发生时间*/
                            ve[k]=ve[i]+*(p->info);
                    }
            }
        if(count<N.vexnum)
        {
            printf("
该有向网有回路\n");
            return 0;
        }
        else
            return 1;
    }

```

在上面的算法中，语句if (ve[i]+*(p->info) >ve[k]) ve[k]=ve[i]+*(p->info) 就是求顶点k的对应事件的最早发生时间，域info保存的是对应弧的权值，在这里将图的邻接表类型定义做了简单的修改。

在求出事件的最早发生时间之后，按照逆拓扑序列就可以推出事件的最晚发生时间、活动的最早开始时间和最晚开始时间。在求出所有参数之后，如果 $ve(i) = vl(i)$ ，输出关键路径经过的顶点。如果 $e(i) = l(i)$ ，将与对应弧关联的两个顶点存入数组e，用来输出关键活动。关键路径算法实现如下。

```

int CriticalPath(AdjGraph N)
/*
输出N
的关键路径*/
{
    int vl[MaxSize]; /*
事件最晚发生时间*/
    SeqStack T;
    int i, j, k, e, l, dut, value, count, e1[MaxSize], e2[MaxSize];
    ArcNode *p;
    if (!TopologicalOrder(N, &T)) /*
如果有环存在，则返回0*/
        return 0;
    value = ve[0];
    for (i = 1; i < N.vexnum; i++)
        if (ve[i] > value) /*value
为事件的最早发生时间的最大值*/
            value = ve[i];
    for (i = 0; i < N.vexnum; i++) /*
将顶点事件的最晚发生时间初始化*/
        vl[i] = value;
    while (!StackEmpty(T)) /*
按逆拓扑排序求各顶点的vl
值*/
        for (PopStack(&T, &j), p = N.vertex[j].firstarc; p; p = p->nextarc) /*
弹出栈T
的元素,
赋给j, p
指向j
的后继事件k*/
        {
            k = p->adjvex;
            dut = * (p->info); /*dut
为弧<j, k>
的权值*/
            if (vl[k] - dut < vl[j]) /*
计算事件j
的最迟发生时间*/
                vl[j] = vl[k] - dut;
        }
    printf("\n
事件的最早发生时间和最晚发生时间\n");
    for (i = 0; i < N.vexnum; i++) /*
输出顶点对应的事件的最早发生时间最晚发生时间*/
        printf("%d    %d    %d\n", i, ve[i], vl[i]);
    printf("
关键路径为: (");
    for (i = 0; i < N.vexnum; i++) /*
输出关键路径经过的顶点*/
        if (ve[i] == vl[i])
            printf("%s ", N.vertex[i].data);
    printf("\n");
    count = 0;
    printf("
活动最早开始时间和最晚开始时间\n");
    for (e = 1; e < N.vexnum; e++) /*
求活动的最早开始时间e
和最晚开始时间l*/
        for (p = N.vertex[j].firstarc; p; p = p->nextarc)
        {
            k = p->adjvex;

```

```

为弧<j,k>
的权值*/
就是活动<j,k>
的最早开始时间*/
就是活动<j,k>
的最晚开始时间*/
→%s %3d %3d %3d\n",N.vertex[j].data,N.vertex[k].data,e,l,l-e);
将关键活动保存在数组中*/
{
    e1[count]=j;
    e2[count]=k;
    count++;
}
printf("
关键活动为: ");
for(k=0;k<count;k++)
输出关键路径*/
{
    i=e1[k];
    j=e2[k];
    printf("(%s
→%s) ",N.vertex[i].data,N.vertex[j].data);
}
printf("\n");
return l;
}
dut=*(p->info); /*dut
e=ve[j]; /*e
l=vl[k]-dut; /*l
printf("%s
if (e==l) /*
{
    e1[count]=j;
    e2[count]=k;
    count++;
}
printf("
for(k=0;k<count;k++) /*
{
    i=e1[k];
    j=e2[k];
    printf("(%s
→%s) ",N.vertex[i].data,N.vertex[j].data);
}
printf("\n");
return l;
}

```

在以上两个算法中，其求解事件的最早发生时间和最晚发生时间为0（n+e）。如果网中存在多个关键路径，则需要同时改进所有的关键路径才能提高整个工程的进度。

10.6 最短路径

在日常生活中，经常会遇到求两个地点之间的最短路径的问题，如一位旅客要从A市到B市，总是希望选择一条途中中转次数最少的路线。用图的弧（或者边）表示两个城市的线路，权值表示城市之间的距离，这样就可以把一个实际问题转化为求图的顶点之间的最短路径问题。本节的主要学习内容包括从某个顶点到其余各顶点的最短路径、任一对顶点之间的最短路径。

10.6.1 从某个顶点到其余各顶点的最短路径

先来讨论下从某个顶点出发到其余各顶点的最短路径问题，即单源点最短路径问题。将有向图中路径上的第一个顶点称为源点（source），最后一个顶点称为终点（destination）。

1. 迪杰斯特拉算法——从某个顶点到其余顶点的最短路径

带权有向图 G_7 从 v_0 出发到其他各个顶点的最短路径如图10-34所示。

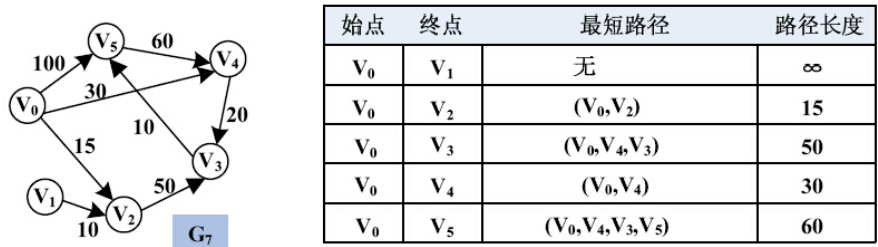


图10-34 图 G_7 从顶点 v_0 到其他各个顶点的最短路径

从图10-34中可以看出，从顶点 v_0 到顶点 v_3 有 (v_0, v_2, v_3) 和 (v_0, v_4, v_3) 两条路径，前者的路径长度为65，后者的路径长度为50。因此， (v_0, v_4, v_3) 是从顶点 v_0 到顶点 v_3 的最短路径。从顶点 v_0 到顶点 v_5 有 (v_0, v_5) 、 (v_0, v_2, v_3, v_5) 和 (v_0, v_4, v_3, v_5) 3条路径，第1条路径长度为100，第2条路径长度为75，第3条路径长度为60。因此， (v_0, v_4, v_3, v_5) 是从顶点 v_0 到顶点 v_5 的最短路径。

如何求得这些路径呢？伟大的计算机科学家迪杰斯特拉（Dijkstra）提出了一个按路径长度递增的次序求最短路径算法。它的基本思想是根据路径长度递增求解从顶点 v_0 到其他各顶点的最短路径。

设有一个带权有向图 $D=(V, E)$ ，定义一个数组 $dist$ ，数组中的每个元素 $dist[i]$ 表示顶点 v_0 到顶点 v_i 的最短路径长度。 $dist$ 的初始状态是：若从顶点 v_0 到顶点 v_i 存在弧，则 $dist[i]$ 是弧 $\langle v_0, v_i \rangle$ 的权值；否则， $dist[i]$ 为 ∞ 。显然，长度为 $dist[j]=\min\{dist[i] | v_i \in V\}$ 的路径表示从顶点 v_0 出发到顶点 v_j 的最短路径，此路径为 (v_0, v_j) 。也就是说，在所有的顶点 v_0 到顶点 v_j 的路径中， $dist[j]$ 是最短的一条路径。

那么，下一条长度次短的路径是哪一条呢？假设该次短路径的终点为 v_k ，则可想而知，这条路径或者是 (v_0, v_k) ，或者是 (v_0, v_j, v_k) ，它的长度或者是 v_0 到 v_k 的弧上的权值，或者是 $dist[j]$ 和从 v_j 到 v_k 的弧上的权值之和。

一般情况下，假设 S 表示求出的最短路径对应终点的集合，则可证明：下一条最短路径（设终点为 v_x ）或者是终点为 v_x 的最短路径或者是弧 $\langle v_0, v_x \rangle$ ，或者是中间经过集合 S 中某个顶点然后到达顶点 v_x 的所经过的路径。这用反证法证明此结论。假设该最短路径有一个顶点 v_y 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，这种情况是不可能出现的。因为最短路径是按照路径长度的递增顺序产生的，所以长度更短的路径已经出现，其终点一定在集合 S 中。因此假设不成立。

例如，从图10-34可以看出， (v_0, v_4) 是从 v_0 到 v_4 的最短路径， (v_0, v_4, v_3) 是从 v_0 到 v_3 的最短路径，经过了顶点 v_4 ； (v_0, v_4, v_3, v_5) 是从 v_0 到 v_5 的最短路径，经过了顶点 v_3 。

因此，在一般情况下，下一条长度次短的路径的长度一定是 $dist[j]=\min\{dist[i] | v_i \in V-S\}$ ， $dist[i]$ 或者是弧 $\langle v_0, v_i \rangle$ 上的权值，或者是 $dist[k]$ （ $v_k \in S$ ）与弧 $\langle v_k, v_i \rangle$ 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 arc 表示带权的有向图， $arc[i][j]$ 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 $arc[i][j]$ 为 ∞ 。 S 为已找到从 v 出发的最短路径的终点的集合，初始时， S 只包括源点 v_0 ，即 $S=\{v_0\}$ ， $V-S$ 包括除 v_0 以外的图中的其他顶点。 v_0 到其他顶点的路径初始化为 $dist[i]=G.arc[0][i].adj$ 。

②选择距离顶点 v_i 最短的顶点 v_j ，使得 $\text{dist}[j]=\text{Min}\{\text{dist}[i] \mid v_i \in V-S\}$ 。 $\text{dist}[j]$ 表示从 v_0 到 v_j 的最短路径长度， v_j 表示对应的终点。

③修改从 v_0 到顶点 v_i 的最短路径长度，其中 $v_i \in S$ 。如果有 $\text{dist}[k]+G.\text{arc}[k][i] < \text{dist}[i]$ ，则修改 $\text{dist}[i]$ ，使得 $\text{dist}[i]=\text{dist}[k]+G.\text{arc}[k][i].\text{adj}$ 。

④重复执行②和③共 $n-1$ 次，可求出所有从 v_0 到其他顶点的最短路径长度。

2. 迪杰斯特拉算法实现

求最短路径的迪杰斯特拉算法描述如下。

```
typedef int PathMatrix[MaxSize][MaxSize]; /*
定义一个保存最短路径的二维数组*/
typedef int ShortPathLength[MaxSize]; /*
定义一个保存从顶点v0
到顶点v
的最短距离的数组*/
void Dijkstra (MGraph N,int v0, PathMatrix path,ShortPathLength dist)
/*
用Dijkstra
算法求有向网N
的v0
顶点到其余各顶点v
的最短路径path[v]
和最短路径长度dist[v]*/
/*final[v]
为1
表示v
∈S
，即已经求出从v0
到v
的最短路径*/
{
    int v,w,i,k,min;
    int final[MaxSize]; /*
记录v0
到该点的最短路径是否已求出*/
    for (v=0;v<N.vexnum;v++) /*
数组dist
存储v0
到v
的最短距离，初始化为v0
到v
的弧的距离*/
    {
        final[v]=0;
        dist[v]=N.arc[v0][v].adj;
        for (w=0;w<N.vexnum;w++)
            path[v][w]=0;
        if (dist[v]<INFINITY) /*
如果从v0
到v
有直接路径，则初始化路径数组*/
        {
            path[v][v0]=1;
            path[v][v]=1;
        }
        dist[v0]=0; /*v0
到v0
的路径为0*/
        final[v0]=1; /*v0
顶点并入集合S*/
        /*
从v0
到其余G.vexnum-1
个顶点的最短路径，并将该顶点并入集合S*/
        for (i=1;i<N.vexnum;i++)
        {
            min=INFINITY;
            for (w=0;w<N.vexnum;w++)
                if (!final[w]&&dist[w]<min) /*
```

```

在不属于集合S
的顶点中找到离v0
最近的顶点*/
{
    v=w;
    /*
    将其离v0
    最近的顶点w
    赋给v
    , 其距离赋给min*/
    min=dist[w];
}
final[v]=1;
/*
将v
并入集合S*/
for (w=0;w<N.vexnum;w++)
/*
利用新并入集合S
的顶点, 更新v0
到不属于集合S
的顶点的最短路径长度和最短路径数组*/
if (!final[w]&&min<INFINITY&&N.arc[v][w].adj<INFINITY&&(min+N.arc[v][w].adj<dist[w]))
{
    dist[w]=min+N.arc[v][w].adj;
    for (k=0;k<N.vexnum;k++)
        path[w][k]=path[v][k];
    path[w][w]=1;
}
}
}

```

其中，二维数组path[v][w]如果为1，则表示从顶点 v_0 到顶点v的最短路径经过顶点w；一维数组dist[v]表示从顶点 v_0 到顶点v的当前求出的最短路径长度。先利用 v_0 到其他顶点的弧的对应的权值将数组path和dist初始化，然后找出从 v_0 到顶点v（不属于集合S）的最短路径，并将v并入集合S，最短路径长度赋给min。接着利用新并入的顶点v，更新 v_0 到其他顶点（不属于集合S）的最短路径长度和最短路径数组。重复执行以上步骤，直到从 v_0 到所有其他顶点的最短路径都求出为止。

该算法的时间耗费主要在3个for循环语句，第一个for循环共执行n次，第二个for循环共执行n-1次，第三个for循环执行n次，如果不考虑每次求解最短路径的耗费，则该算法的时间复杂度是 $O(n^2)$ 。若用带权的邻接表作为有向图的存储结构，虽然修改dist的时间可以减少，但由于在dist中选择最小元素的时间不变，所以点的时间复杂度仍为 $O(n^2)$ 。

利用迪杰斯特拉算法求最短路径的思想，图10-34所示图 G_7 的带权邻接矩阵和从顶点 v_0 到其他顶点的最短路径求解过程如图10-35所示。

终点	路径长度和路径数组	从顶点 v_0 到其他各顶点的最短路径的求解过程				
		$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
v_1	dist path	∞	∞	∞	∞	∞ 无
v_2	dist path	15 (v_0, v_2)				
v_3	dist path	∞	65 (v_0, v_2, v_3)	50 (v_0, v_4, v_3)		
v_4	dist path	30 (v_0, v_4)	30 (v_0, v_4)			
v_5	dist path	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)	
最短路径终点		v_2	v_4	v_3	v_5	
集合S		{ v_0, v_2 }	{ v_0, v_2, v_4 }	{ v_0, v_2, v_3, v_4 }	{ v_0, v_1, v_5, v_3, v_5 }	

图10-35 带权图 G_7 的从顶点 v_0 到其他各顶点的最短路径求解过程示意图

【例10-4】 创建一个如图10-34所示的有向网N，输出该有向网N中从 v_0 开始到其他各顶点的最短路径及最短路径长度。

算法实现代码如下。

```
#include<stdio.h>
#include<string.h>
#include<malloc.h>
#include<stdlib.h>
typedef char VertexType[4];
typedef char InfoPtr;
typedef int VRType;
#define INFINITY 65535 /*
定义一个无限大的值*/
#define MaxSize 50 /*
顶点个数的最大值*/
typedef int PathMatrix[MaxSize][MaxSize]; /*
定义一个保存最短路径的二维数组*/
typedef int ShortPathLength[MaxSize]; /*
定义一个保存从顶点v0
到顶点v
的最短距离的数组*/
typedef enum{DG,DN,UG,UN}GraphKind; /*
图的类型：有向图、有向网、无向图和无向网*/
typedef struct
{
    VRType adj; /*
对于无权图，用1
表示相邻，0
表示不相邻；对于带权图，存储权值*/
    InfoPtr *info; /*
与弧或边的相关信息*/
}ArcNode,AdjMatrix[MaxSize][MaxSize]; /*
图的类型定义*/
{
    VertexType vex[MaxSize]; /*
用于存储顶点*/
    AdjMatrix arc; /*
邻接矩阵，存储边或弧的信息*/
    int vexnum,arcnum; /*
顶点数和边（弧）的数目*/
    GraphKind kind; /*
图的类型*/
}MGraph;
typedef struct /*
添加一个存储网的行、列和权值的类型定义*/
{
    int row;
    int col;
    int weight;
}GNode;
void CreateGraph(MGraph *N,GNode *value,int vnum,int arcnum,VertexType *ch);
void DisplayGraph(MGraph N);
void Dijkstra(MGraph N,int v0,PathMatrix path,ShortPathLength dist);
void main()
{
    int i,vnum=6,arcnum=8;
    MGraph N;
```

```

        GNode value[]={0,2,15},{0,4,30},{0,5,100},{1,2,10},
                        {2,3,50},{3,5,10},{4,3,20},{5,4,60}};
        VertexType ch={"v0","v1","v2","v3","v4","v5"};
        PathMatrix path;
用二维数组存放最短路径所经过的顶点*/
        ShortPathLength dist;
用一维数组存放最短路径长度*/
        CreateGraph(&N,value,vnum,arcnum,ch);    /*
创建有向网N*/
        DisplayGraph(N);                          /*
输出有向网N*/
        Dijkstra(N,0,path,dist);
        printf("%s
到各顶点的最短路径长度为: \n",N.vex[0]);
        for(i=0;i<N.vexnum;i++)
            if(i!=0)
                printf("%s-%s:%d\n",N.vex[0],N.vex[i],dist[i]);
    }
void CreateGraph(MGraph *N,GNode *value,int vnum,int arcnum,VertexType *ch)
/*
采用邻接矩阵表示法创建有向网N*/
{
    int i,j,k;
    N->vexnum=vnum;
    N->arcnum=arcnum;
    for(i=0;i<vnum;i++)
将各个顶点赋值给vex域*/
        strcpy(N->vex[i],ch[i]);
    for(i=0;i<N->vexnum;i++)
初始化邻接矩阵*/
        for(j=0;j<N->vexnum;j++)
        {
            N->arc[i][j].adj=INFINITY;
            N->arc[i][j].info=NULL;
弧的信息初始化为空*/
        }
        for(k=0;k<arcnum;k++)
        {
            i=value[k].row;
            j=value[k].col;
            N->arc[i][j].adj=value[k].weight;
        }
        N->kind=DN;
图类型为有向网*/
}
void DisplayGraph(MGraph N)
/*
输出邻接矩阵存储表示的图N*/
{
    int i,j;
    printf("
有向网具有%d
个顶点%d
条弧, 顶点依次是: ",N.vexnum,N.arcnum);
    for(i=0;i<N.vexnum;++i)
输出网的顶点*/
        printf("%s ",N.vex[i]);
    printf("\n
有向网N
的:\n");
输出网N
的弧*/
    printf("
序号i=");
    for(i=0;i<N.vexnum;i++)
        printf("%8d",i);
    printf("\n");
    for(i=0;i<N.vexnum;i++)
    {
        printf("%8d",i);
        for(j=0;j<N.vexnum;j++)
            printf("%8d",N.arc[i][j].adj);
        printf("\n");
    }
}

```

程序运行结果如图10-36所示。

```

D:\零基础学数据结构问题代码\第10章\例10_4\Debug\例10_...
有向网具有6个顶点8条弧，顶点依次是：v0 v1 v2 v3 v4 v5
有向网N的：
序号i=      0      1      2      3      4      5
0      65535      65535      15      65535      30      100
1      65535      65535      10      65535      65535      65535
2      65535      65535      65535      50      65535      65535
3      65535      65535      65535      65535      65535      10
4      65535      65535      65535      65535      65535      65535
5      65535      65535      65535      65535      60      65535
v0到各顶点的最短路径长度为：
v0-v1:65535
v0-v2:15
v0-v3:50
v0-v4:30
v0-v5:60
Press any key to continue

```

图10-36 迪杰斯特拉算法求从 v_0 到其他各顶点最短路径的程序运行结果

10.6.2 每一对顶点之间的最短路径

如果要计算每一对顶点之间的最短路径，需每次以一个顶点为出发点，将迪杰斯特拉算法重复执行 n 次，就可以得到每一对顶点的最短路径。总的时间复杂度为 $O(n^3)$ 。下面介绍由另一位伟大的计算机科学家弗洛伊德（Floyd）提出的另一个算法，其时间复杂度也是 $O(n^3)$ ，但形式简单些。

1. 各个顶点之间的最短路径算法思想

弗洛伊德算法的思想是：假设求顶点 v_i 到顶点 v_j 的最短路径，如果从顶点 v_i 到顶点 v_j 有弧，则从 v_i 到 v_j 存在一条长度为 $\text{arc}[i][j]$ 的路径，但该路径不一定是 v_i 到 v_j 的最短路径，还需进行 n 次试探。首先需要从顶点 v_0 开始，如果有路径 (v_i, v_0, v_j) 存在，则比较路径 (v_i, v_j) 和 (v_i, v_0, v_j) ，选择两者中最短的一个且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, \dots, v_1) 和 (v_1, \dots, 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 $(v_i, \dots, v_1, \dots, v_j)$ 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。把它和已经得到的从 v_i 到 v_j 中间顶点序号不大于0的最短路径相比较，从中选择中间顶点的序号不大于1的最短路径之后，再增加1个顶点 v_2 ，继续进行试探。

依次类推，在一般情况下，若 (v_i, \dots, v_k) 和 (v_k, \dots, 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 $(v_i, \dots, v_k, \dots, v_j)$ 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，其长度较短者就是从 v_i 到 v_j 的中间顶点的序号不大于 k 的最短路径。经过 n 次比较，可以得到从 v_i 到 v_j 的中间顶点序号不大于 $n-1$ 的最短路径。依照这种方法，可以得到各个顶点之间的最短路径。

假设采用邻接矩阵存储带权有向图 G ，则各个顶点之间的最短路径可以保存在一个 n 阶方阵 D 中，每次求出的最短路径可以用矩阵表示为 D^{-1} 、 D^0 、 D^1 、 D^2 、 \dots 、 D^{n-1} 。其中 $D^{-1}[i][j]=G.arc[i][j].adj$ ， $D^k[i][j]=\text{Min}\{D^{k-1}[i][j], D^{k-1}[i][k]+D^{k-1}[k][j] \mid 0 \leq k \leq n-1\}$ 。其中， $D^k[i][j]$ 表示从顶点 v_i 到顶点 v_j 的中间顶点序号不大于 k 的最短路径长度，而 $D^{n-1}[i][j]$ 即为从顶点 v_i 到顶点 v_j 的最短路径长度。

2. 各个顶点之间的最短路径算法实现

求各个顶点之间的最短路径算法实现如下。

```

void Floyd(MGraph N, PathMatrix path, ShortPathLength dist)
/*
用Floyd
算法求有向图N
的各项点v
和w
之间的最短路径，其中path[v][w][u]
表示u
是从v
到w
当前求得最短路径上的顶点*/
{
    int u, v, w, i;
    for (v=0; v<N.vexnum; v++)
        /*
初始化数组path
和dist*/
        for (w=0; w<N.vexnum; w++)
        {
            dist[v][w]=N.arc[v][w].adj;
            /*
初始时，顶点v
到顶点w
的最短路径为v
到w
的弧的权值*/
            for (u=0; u<N.vexnum; u++)
                path[v][w][u]=0;
            /*
路径矩阵初始化为零*/
            if (dist[v][w]<INFINITY)
                /*
如果v
到w
有路径，
则由v
到w
的路径经过v
和w
两点*/
                {
                    path[v][w][v]=1;
                    path[v][w][w]=1;
                }
        }
    for (u=0; u<N.vexnum; u++)
        for (v=0; v<N.vexnum; v++)
            for (w=0; w<N.vexnum; w++)
                if (dist[v][u]<INFINITY&&dist[u][w]<INFINITY&&dist[v][u]+dist[u][w]<dist[v][w])
                    /*
从v
...

```

```
经u
到w
的一条路径为当前最短的路径*/
{
    dist[v][w]=dist[v][u]+dist[u][w];          /*
更新v
到w
的最短路径*/
for (i=0;i<N.vexnum;i++)                      /*
从v
到w
的路径经过从v
到u
和从u
到w
的所有路径*/
    path[v][w][i]=path[v][u][i]||path[u][w][i];
}
```

根据弗洛伊德算法，图10-37所示的带权有向图G₈ 的每一对顶点之间的最短路径P和最短路径长度D求解过程如图10-38所示。

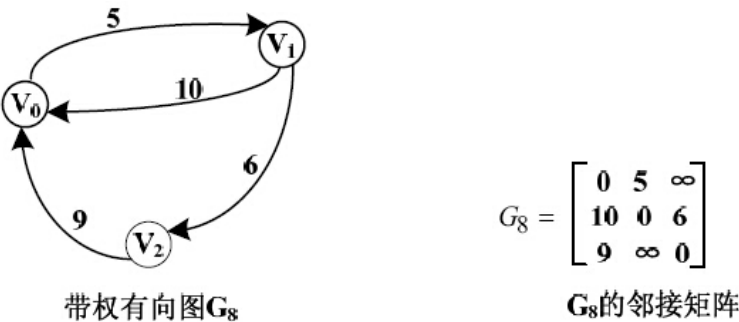


图10-37 带权有向图G₈ 及邻接矩阵

D	D ⁻¹			D ⁰			D ¹			D ²		
	0	1	2	0	1	2	0	1	2	0	1	2
0	0	5	∞	0	5	∞	0	5	11	0	5	11
1	10	0	6	10	0	6	10	0	6	10	0	6
2	9	∞	0	9	14	0	9	14	0	9	14	0
P	P ⁻¹			P ⁰			P ¹			P ²		
	0	1	2	0	1	2	0	1	2	0	1	2
0		v ₀ v ₁			v ₀ v ₁			v ₀ v ₁	v ₀ v ₁ v ₂		v ₀ v ₁	v ₀ v ₁ v ₂
1	v ₁ v ₀		v ₁ v ₂	v ₁ v ₀		v ₁ v ₂	v ₁ v ₀		v ₁ v ₂	v ₁ v ₀		v ₁ v ₂
2	v ₂ v ₀			v ₂ v ₀	v ₂ v ₀ v ₁		v ₂ v ₀	v ₂ v ₀ v ₁		v ₂ v ₀	v ₂ v ₀ v ₁	

图10-38 带权有向图G₈ 的各个顶点之间的最短路径及长度

10.6.3 最短路径应用举例

带权图（权值非负，表示边连接的两个顶点间的距离）的最短路径问题是找出从初始顶点到目标顶点之间的一条最短路径。假设从初始顶点到目标顶点之间存在路径，解决问题的方法如下。

- ①设最短路径初始时仅包含初始顶点，令当前顶点u为初始顶点。

②选择离 u 最近且尚未在最短路径中的一个顶点 v ，加入最短路径中，修改当前顶点 $u=v$ 。

③重复执行步骤①和②，直到 u 是目标顶点为止。

请问上述方法能否求得最短路径？若该方法可行，请证明之；否则，举例说明。

【分析】 该题目是2009年的考研题，主要考查最短路径的掌握情况。

上述方法不一定能求出最短路径。例如图10-39。

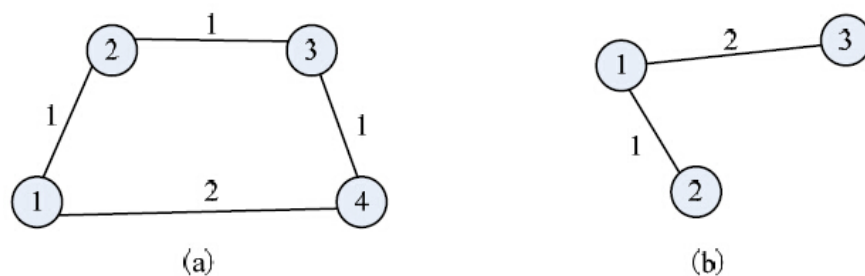


图10-39 求带权图

对于图10-39 (a)，设初始顶点为1，目标顶点为4，求从顶点1到顶点4之间的最短路径。显然这两个顶点之间的最短路径为2。而利用题中给出的方法求得的最短路径为 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ，长度为3，这条路径并不是顶点1到顶点4之间的最短路径。

对于图10-39 (b)，设初始顶点为1，目标顶点为3，求从顶点1到顶点3之间的最短路径，利用题目给出的方法，求出 $1 \rightarrow 2$ 之后，无法求出顶点1到顶点3的路径。

10.7 图的应用举例

本节将通过几个具体实例来说明图的具体应用。

【例10-5】 有一个邻接表存储的图G，分别设计实现如下要求的算法。

- (1) 求出图G中每个顶点的出度。
- (2) 求出图G中出度最大的一个顶点，输出该顶点的编号。
- (3) 计算图G中出度为零的顶点数。
- (4) 判断图G中是否存在边 $\langle i, j \rangle$ 。

【分析】 主要考查对图的邻接表存储特点和基本操作掌握情况。从图的表头结点出发，依次访问边表结点，并进行计数，就可得到相应每个顶点的出度。问题（1）、（2）、（3）可归结为一个问题，其中求某个顶点的出度可以写成一个函数`OutDegree (AdjGraph G, int v)`，在求（1）、（2）、（3）的问题时，可调用该函数实现。

对于问题（4），可令`p=G.vertex[i].firstarc`，然后依次遍历p指向链表中的每个结点，看该结点的序号是否为j，如果为j，则说明图中存在弧 $\langle i, j \rangle$ ；若`p==NULL`，则说明图中不存在弧 $\langle i, j \rangle$ 。代码如下。

```
while (p!=NULL && p->adjvex!=j)
    p=p->nextarc;
```

算法实现如下。

```

/*
包含头文件*/
#include<stdlib.h>
#include<stdio.h>
#include<malloc.h>
#include<string.h>
typedef int DataType; /*
栈中的元素类型定义*/
#include"SeqStack.h"
/*
图的邻接表类型定义*/
typedef char VertexType[4];
typedef int InfoPtr; /*
定义为整型, 为了存放权值*/
typedef int VRType;
#define MaxSize 50 /*
顶点个数的最大值*/
typedef enum{DG,DN,UG,UN}GraphKind; /*
图的类型: 有向图、有向网、无向图和无向网*/
typedef struct ArcNode /*
边结点的类型定义*/
{
    int adjvex; /*
弧指向的顶点的位置*/
    InfoPtr *info; /*
弧的权值*/
    struct ArcNode *nextarc; /*
指示下一个与该顶点相邻接的顶点*/
}ArcNode;
typedef struct VNode /*
头结点的类型定义*/
{
    VertexType data; /*
用于存储顶点*/
    ArcNode *firstarc; /*
指示第一个与该顶点邻接的顶点*/
}VNode,AdjList[MaxSize];
typedef struct /*
图的类型定义*/
{
    AdjList vertex;
    int vexnum,arcnum; /*
图的顶点数目与弧的数目*/
    GraphKind kind; /*
图的类型*/
}AdjGraph;
int LocateVertex(AdjGraph G,VertexType v);
void CreateGraph(AdjGraph *G);
void DisplayGraph(AdjGraph G);
void DestroyGraph(AdjGraph *G);
int OutDegree(AdjGraph G,int v);
void AllOutDegree(AdjGraph G);
void MaxOutDegree(AdjGraph G);
void ZeroOutDegree(AdjGraph G);
void ExistArc(AdjGraph G);
void main()
{
    AdjGraph G;
    CreateGraph(&G); /*
采用邻接表存储结构创建有向图G*/
    DisplayGraph(G); /*
输出有向图G*/
    AllOutDegree(G); /*
有向图G
各顶点的出度*/
    MaxOutDegree(G); /*
求有向图G
出度最大的顶点*/
    ExistArc(G); /*
判断有向图G
中是否存在弧<i,j>*/
    DestroyGraph(&G); /*
销毁图G*/
}

```

```

int OutDegree(AdjGraph G,int v)
{
    ArcNode *p;
    int n=0;
    p=G.vertex[v].firstarc;
    while(p!=NULL)
    {
        n++;
        p=p->nextarc;
    }
    return n;
}
void AllOutDegree(AdjGraph G)
{
    int i;
    printf("(1)");
    各顶点的出度:\n";
    for(i=0;i<G.vexnum;i++)
        printf("
顶点%s:%d\n",G.vertex[i].data,OutDegree(G,i));
    printf("\n");
}
void MaxOutDegree(AdjGraph G)
{
    int maxv=0,maxds=0,i,x;
    for(i=0;i<G.vexnum;i++)
    {
        x=OutDegree(G,i);
        if(x>maxds)
        {
            maxds=x;
            maxv=i;
        }
    }
    printf("(2)");
    最大出度的顶点是%s,
    出度为%d\n",G.vertex[maxv].data,maxds);
}
void ZeroOutDegree(AdjGraph G)
{
    int i,x;
    printf("(3)");
    出度为零的顶点:");
    for(i=0;i<G.vexnum;i++)
    {
        x=OutDegree(G,i);
        if(x==0)
            printf("%s\n",G.vertex[i].data);
    }
}
void ExistArc(AdjGraph G)
{
    int i,j;
    VertexType v1,v2;
    ArcNode *p;
    printf("(4)");
    是否存在弧<i,j>\n";
    printf("
请输入弧的弧头和弧尾:");
    scanf("%s%s*c",v1,v2);
    i=LocateVertex(G,v1);
    j=LocateVertex(G,v2);
    p=G.vertex[i].firstarc;
    while(p!=NULL && p->adjvex!=j)
        p=p->nextarc;
    if(p==NULL)
        printf("
不存在弧<%s,%s>\n",v1,v2);
    else
        printf("
存在弧<%s,%s>\n",v1,v2);
}
int LocateVertex(AdjGraph G,VertexType v)
/*
.....

```

```

返回图中顶点对应的位置*/
{
    int i;
    for(i=0;i<G.vexnum;i++)
        if(strcmp(G.vertex[i].data,v)==0)
            return i;
    return -1;
}
void CreateGraph(AdjGraph *G)
/*
采用邻接表存储结构，创建有向图*/
{
    int i,j,k;
    VertexType v1,v2; /*
    定义两个弧v1
    和v2*/
    ArcNode *p;
    printf("
请输入图的顶点数，
边数(
以逗号分隔): ");
    scanf("%d,%d",&(*G).vexnum,&(*G).arcnum);
    printf("
请输入%d
个顶点的值:",G->vexnum);
    for(i=0;i<G->vexnum;i++) /*
    将顶点存储在头结点中*/
    {
        scanf("%s",G->vertex[i].data);
        G->vertex[i].firstarc=NULL; /*
    将相关联的顶点置为空*/
    }
    printf("
请输入弧尾和弧头(
以空格作为分隔):\n");
    for(k=0;k<G->arcnum;k++) /*
    建立边链表*/
    {
        scanf("%s%s%c",v1,v2);
        i=LocateVertex(*G,v1);
        j=LocateVertex(*G,v2);
        /*j
    为弧头i
    为弧尾创建邻接表*/
        p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=j;
        /*
    将p
    指向的结点插入边表中*/
        p->nextarc=G->vertex[i].firstarc;
        G->vertex[i].firstarc=p;
    }
    (*G).kind=DG;
}
void DestroyGraph(AdjGraph *G)
/*
销毁图*/
{
    int i;
    ArcNode *p,*q;
    for(i=0;i<G->vexnum;++i) /*
    释放网中的边表结点*/
    {
        p=G->vertex[i].firstarc; /*p
    指向边表的第一个结点*/
        if(p!=NULL) /*
    如果边表不为空，则释放边表的结点*/
        {
            q=p->nextarc;
            free(p);
            p=q;
        }
    }
    (*G).vexnum=0; /*
    . . . . .

```

```

将顶点数置为0*/
(*G).arcnum=0;
/*
将边的数目置为0*/
}
void DisplayGraph(AdjGraph G)
/*
输出图G
的各条弧*/
{
    int i;
    ArcNode *p;
    printf("
该图中有%d
个顶点: ",G.vexnum);
    for(i=0;i<G.vexnum;i++)
        printf("%s ",G.vertex[i].data);
    printf("\n
图中共有%d
条弧:\n",G.arcnum);
    for(i=0;i<G.vexnum;i++)
    {
        p=G.vertex[i].firstarc;
        while(p)
        {
            printf("<%s,%s> ",G.vertex[i].data,G.vertex[p->adjvex].data);
            p=p->nextarc;
        }
        printf("\n");
    }
}

```

程序运行结果如图10-40所示。

```
G:\ "D:\零基础学数据结构\例10_5\Debug\例10... - □ ×
请输入图的顶点数,边数<以逗号分隔>: 5,6
请输入5个顶点的值:v1 v2 v3 v4 v5
请输入弧尾和弧头<以空格作为分隔>:
v1 v2
v1 v3
v1 v4
v2 v5
v3 v4
v4 v5
该图中有5个顶点: v1 v2 v3 v4 v5
图中共有6条弧:
<v1,v4> <v1,v3> <v1,v2>
<v2,v5>
<v3,v4>
<v4,v5>

<1>各顶点的出度:
顶点v1:3
顶点v2:1
顶点v3:1
顶点v4:1
顶点v5:0

<2>最大出度的顶点是v1,出度为3
<4>是否存在弧<i,j>
请输入弧的弧头和弧尾:v2 v5
存在弧<v2,v5>
Press any key to continue
```

图10-40 程序运行结果

【例10-6】 设计一个算法，判断无向图G是否为一棵树。

【分析】 一个无向图G是一棵树的条件是G必须是无回路的连通图或是有n-1条边的连通图，这里我们采用后者作为判断条件。

对连通的判定，可通过判断能否遍历全部顶点来实现，算法如下。

```
int IsTree(AdjGraph *G)
{
    int vNum=0,eNum=0,i;
    for(i=0;i<G->vexnum;i++)
        visited[i]=0;
    DFS(G,0,&vNum,&eNum);
    if(vNum==G->vexnum && eNum==2*(G->vexnum-1))
        return 1;
    else
        return 0;
}
```

```

void DFS (AdjGraph *G,int v,int *vNum,int *eNum)
{
    ArcNode *p;
    visited[v]=1;
    (*vNum)++;
    p=G->vertex[v].firstarc;
    while (p!=NULL)
    {
        (*eNum)++;
        if (visited[p->adjvex]==0)
            DFS (G,p->adjvex,vNum,eNum);
        p=p->nextarc;
    }
}

```

在深度搜索遍历的过程中，同时对遍历过的顶点和边数计数，当全部顶点都遍历过且边数为 $2 * (n - 1)$ 时，这个图就是一棵树，否则不是一棵树。

判断无向图G是否为一棵树的算法实现如下。

```

/*
头文件*/
#include<stdlib.h>
#include<stdio.h>
#include<malloc.h>
#include<string.h>
/*
图的邻接表类型定义*/
typedef char VertexType[4];
typedef char InfoPtr;
typedef int VRType;
#define MaxSize 50 /*
顶点个数的最大值*/
typedef enum {DG,DN,UG,UN} GraphKind; /*
图的类型：有向图、有向网、无向图和无向网*/
typedef struct ArcNode /*
边表结点的类型定义*/
{
    int adjvex; /*
    弧指向的顶点的位置*/
    InfoPtr *info; /*
    与弧相关的信息*/
    struct ArcNode *nextarc; /*
    指示下一个与该顶点相邻接的顶点*/
}ArcNode; /*
typedef struct VNode /*
表头结点的类型定义*/
{
    VertexType data; /*
    用于存储顶点*/
    ArcNode *firstarc; /*
    指示第一个与该顶点相邻接的顶点*/
}VNode,AdjList[MaxSize];
typedef struct /*
图的类型定义*/
{
    AdjList vertex;
    int vexnum,arcnum; /*
    图的顶点数目与弧的数目*/
    GraphKind kind; /*
    图的类型*/
}AdjGraph;
/*
函数声明*/

```



```

int LocateVertex(AdjGraph G,VertexType v);
void CreateGraph(AdjGraph *G);
void DisplayGraph(AdjGraph G);
void DestroyGraph(AdjGraph *G);
int IsTree(AdjGraph *G);
void DFS(AdjGraph *G,int v,int *vNum,int *eNum);
int visited[MaxSize];
void main()
{
    AdjGraph G;
    printf("
采用邻接矩阵创建无向图G
: \n");

    CreateGraph(&G);
    printf("
输出无向图G
: ");

    DisplayGraph(G);
    if(IsTree(&G))
        printf("
无向图G
是一棵树!\n");
    else
        printf("
无向图G
不是一棵树!\n");
    DestroyGraph(&G);
}
int IsTree(AdjGraph *G)
{
    int vNum=0,eNum=0,i;
    for(i=0;i<G->vexnum;i++)
        visited[i]=0;
    DFS(G,0,&vNum,&eNum);
    if(vNum==G->vexnum && eNum==2*(G->vexnum-1))
        return 1;
    else
        return 0;
}
void DFS(AdjGraph *G,int v,int *vNum,int *eNum)
{
    ArcNode *p;
    visited[v]=1;
    (*vNum)++;
    p=G->vertex[v].firstarc;
    while(p!=NULL)
    {
        (*eNum)++;
        if(visited[p->adjvex]==0)
            DFS(G,p->adjvex,vNum,eNum);
        p=p->nextarc;
    }
}
void CreateGraph(AdjGraph *G)
/*
采用邻接表存储结构, 创建无向图G*/
{
    int i,j,k;
    VertexType v1,v2;
    /*
    定义两个顶点v1
    和v2*/

    ArcNode *p;
    printf("
输入图的顶点数,
边数(
逗号分隔): ");

    scanf("%d,%d",&(*G).vexnum,&(*G).arcnum);
    printf("
输入%d
个顶点的值:\n",G->vexnum);
    for(i=0;i<G->vexnum;i++)
        /*
    将顶点存储在表头结点中*/
    {
        scanf("%s",G->vertex[i].data);
    }
}

```

```

        G->vertex[i].firstarc=NULL;  /*
将相关联的顶点置为空*/
    }
    printf("
输入弧尾和弧头(
以空格作为间隔):\n");
    for (k=0;k<G->arcnum;k++)          /*
建立边链表*/
    {
        scanf("%s%s",v1,v2);
        i=LocateVertex(*G,v1);
        j=LocateVertex(*G,v2);
        /*j
为入边i
为出边创建邻接表*/

        p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=j;
        p->info=NULL;
        p->nextarc=G->vertex[i].firstarc;
        G->vertex[i].firstarc=p;
        /*i
为入边j
为出边创建邻接表*/

        p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=i;
        p->info=NULL;
        p->nextarc=G->vertex[j].firstarc;
        G->vertex[j].firstarc=p;
    }
    (*G).kind=UG;
}
int LocateVertex(AdjGraph G,VertexType v)
/*
返回图中顶点对应的位置*/
{
    int i;
    for(i=0;i<G.vexnum;i++)
        if(strcmp(G.vertex[i].data,v)==0)
            return i;
    return -1;
}
void DestroyGraph(AdjGraph *G)
/*
销毁无向图G*/
{
    int i;
    ArcNode *p,*q;
    for(i=0;i<(*G).vexnum;++i)          /*
释放图中的边表结点*/
    {
        p=G->vertex[i].firstarc;        /*p
指向边表的第一个结点*/
        if(p!=NULL)                      /*
如果边表不为空,则释放边表的结点*/
        {
            q=p->nextarc;
            free(p);
            p=q;
        }
    }
    (*G).vexnum=0;                      /*
将顶点数置为0*/
    (*G).arcnum=0;                      /*
将边的数目置为0*/
}
void DisplayGraph(AdjGraph G)
/*
图的邻接表存储结构的输出*/
{
    int i;
    ArcNode *p;
    printf("%d
个顶点: \n",G.vexnum);
    for (i=0;i<G.vexnum;i++)

```

```

        printf("%s ",G.vertex[i].data);
    printf("\n%d
条边:\n",2*G.arcnum);
    for(i=0;i<G.vexnum;i++)
    {
        p=G.vertex[i].firstarc;          /*
将p
指向边表的第一个结点*/
        while(p)                          /*
输出无向图的所有边*/
        {
            printf("%s
-%s ",G.vertex[i].data,G.vertex[p->adjvex].data);
            p=p->nextarc;
        }
        printf("\n");
    }
}

```

程序运行结果如图10-41所示。

```

C:\ "D:\零基础学数据结构\例10_6\De...
采用邻接矩阵创建无向图G:
输入图的顶点数,边数(逗号分隔): 5,4
输入5个顶点的值:
v1 v2 v3 v4 v5
输入弧尾和弧头(以空格作为间隔):
v1 v2
v1 v3
v2 v4
v2 v5
输出无向图G: 5个顶点:
v1 v2 v3 v4 v5
8条边:
v1→v3 v1→v2
v2→v5 v2→v4 v2→v1
v3→v1
v4→v2
v5→v2
无向图G是一棵树!
Press any key to continue

```

图10-41 判断无向图G是否为一棵树的程序运行结果

10.8 小结

图中元素之间是一种多对多的关系。

图由顶点和边（弧）构成，根据边的有向和无向可以将图分为两种：有向图和无向图。将带权的有向图称为有向网，带权的无向图称为无向网。

图的存储结构有邻接矩阵存储结构、邻接表存储结构、十字链表存储结构和邻接多重表存储结构4种。其中，最常用的是邻接矩阵存储和邻接表存储。

图的遍历分为广度优先搜索和深度优先搜索两种。图的广度优先搜索遍历类似于树的层次遍历，图的深度优先搜索遍历类似于树的先根遍历。

一个连通图的生成树是指一个极小连通子图，假设图中有 n 个顶点，则它包含图中 n 个顶点和构成一棵树的 $n-1$ 条边。

构造最小生成树的算法主要有两个，即普里姆算法和克鲁斯卡尔算法。

关键路径是指路径最长的路径，关键路径表示了完成工程的最短工期。关键路径上的活动称为关键活动，关键活动可以决定整个工程

完成任务的日期。非关键活动不能决定工程的进度。

最短路径是指从一个顶点到另一个顶点路径长度最小的一条路径。求最短路径的算法主要有两个，即迪杰斯特拉算法和弗洛伊德算法。

10.9 习题

一、选择题

1. 如果从无向图的任一顶点出发进行一次深度优先搜索即可访问所有顶点，则该图一定是（ ）。

A. 完全图

B. 连通图

C. 有回路

D. 一棵树

2. 关键路径是事件结点网络中（ ）。

A. 从源点到汇点的最长路径

B. 从源点到汇点的最短路径

C. 最长的回路

D. 最短的回路

3. 下面（ ）可以判断出一个有向图中是否有环（回路）。

A. 广度优先遍历

B. 拓扑排序

C. 求最短路径

D. 求关键路径

4. 采用邻接表存储的图，其深度优先遍历类似于二叉树的（ ）。

A. 中序遍历

B. 先序遍历

C. 后序遍历

D. 按层次遍历

5. 无向图的邻接矩阵是一个（ ）。

A. 对称矩阵

B. 零矩阵

C. 上三角矩阵

D. 对角矩阵

6. 邻接表是图的一种（ ）。

A. 顺序存储结构

B. 链式存储结构

C. 索引存储结构

D. 散列存储结构

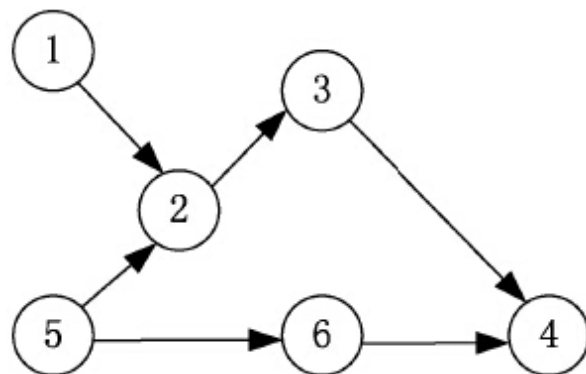
7. 下面有向图所示的拓扑排序的结果序列是（ ）。

A. 125634

B. 516234

C. 123456

D. 521643



8. 任一个有向图的拓扑序列（ ）。

- A. 不存在
- B. 有一个
- C. 一定有多个
- D. 有一个或多个

9. 采用邻接表存储的图的广度优先遍历算法类似于二叉树的
()。

- A. 先序遍历
- B. 中序遍历
- C. 后序遍历
- D. 按层次遍历

二、综合题

1. AOE网G如图10-42所示，求关键路径（要求标明每个顶点的最早发生时间和最迟发生时间，并画出关键路径）。

2. 已知有向图G如图10-43所示，根据迪杰斯特拉算法求顶点 v_0 到其他顶点的最短距离（给出求解过程）。

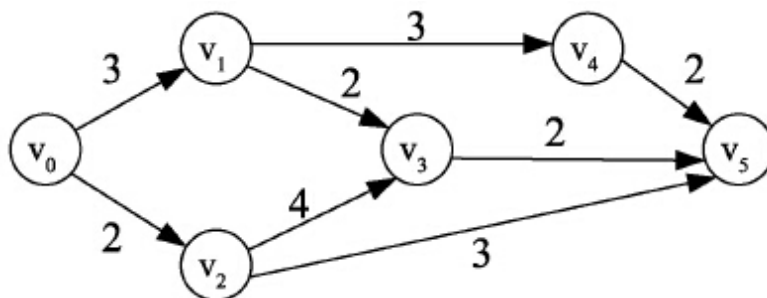


图10-42 AOE网G

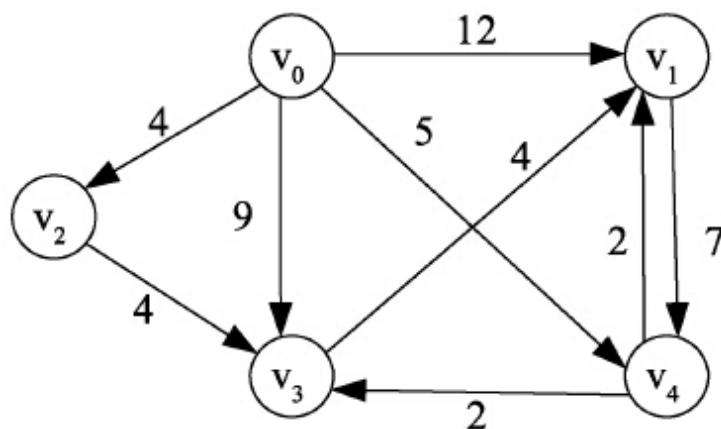


图10-43 有向图G

3. 已知图G如图10-44所示，根据Prim算法，构造最小生成树（要求给出生成过程）。

4. 如图10-45所示的AOE网，求解如下问题。

(1) 事件的最早开始时间 ve 和最迟开始时间 vl 。

(2) 出关键路径。

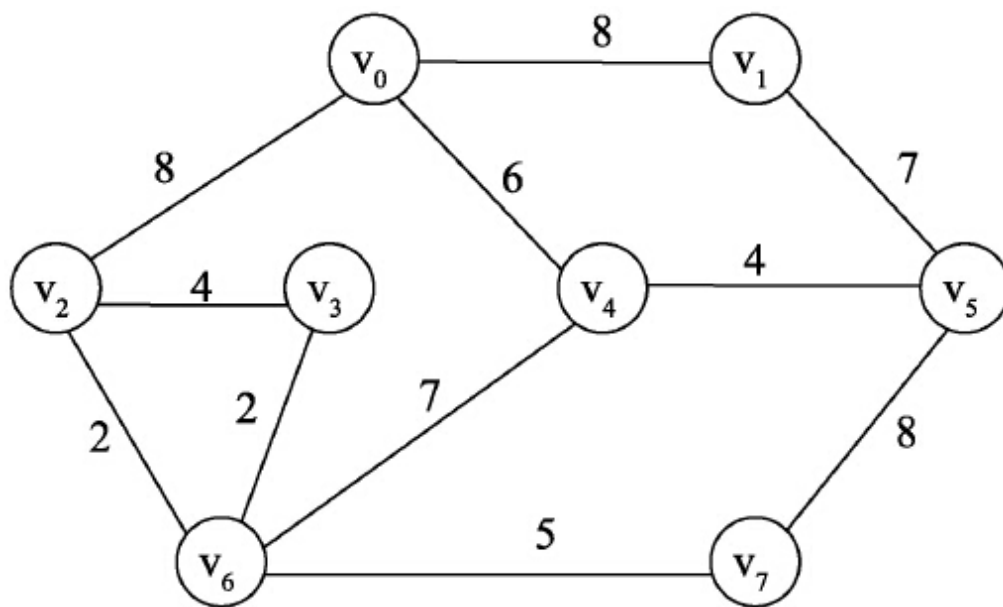


图10-44 图G

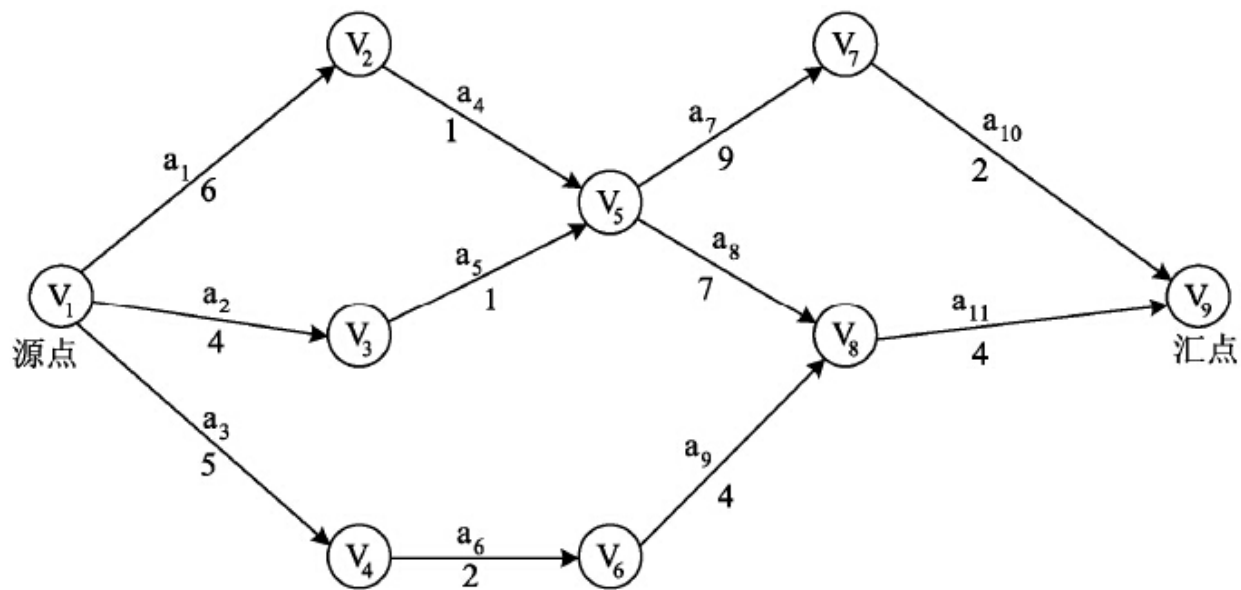


图10-45 AOE网

三、算法设计题

1. 编写一个算法，判断有向图是否存在回路。

2. 采用邻接表创建一个无向图 G_6 ，并实现对图的深度优先遍历和图的广度优先遍历。

3. 采用邻接表创建如图10-30所示的有向网，并求网中顶点的拓扑序列，然后计算该有向网的关键路径。

4. 编写一个算法，判断无向图是否是连通图，如果是连通图，则返回1，否则返回0。

5. 一个连通图采用邻接表作为存储结构，设计一个算法实现从顶点 v 出发的深度优先搜索遍历的非递归过程。

6. 创建一个无向图，求图中从顶点 u 到顶点 v 的一条简单路径，并输出所在路径。

分析 主要考查图深度优先遍历。通过从顶点 u 开始对图进行广度优先遍历，如果访问到顶点 v ，则说明从顶点 u 到顶点 v 存在一条路径。因为在图的遍历过程中，要求每个顶点只能访问一次，所以该路径一定是简单路径。在遍历过程中，将当前访问到的顶点都记录下来，就得到了从顶点 u 到顶点 v 的简单路径。可以利用一个一维数组 $parent$ 记录访问过的顶点，如 $parent[u]=w$ ，表示顶点 w 是 u 的前驱顶点。如果 u 到 v 是一条简单路径，则输出该路径。

第四篇 查找与排序

第11章 查找

本书在第3章至第10章中已经介绍了各种线性和非线性的数据结构，本章将讨论另一种在实际应用中大量使用的重要技术——查找。在计算机处理非数值问题时，查找是一种经常使用和非常重要的操作。本章主要介绍查找的基本概念、静态查找、动态查找、哈希表及查找。

本章重点和难点：

- 折半查找算法
- 索引顺序表的查找
- 二叉排序树和平衡二叉树
- B-树
- 哈希表的构造与查找

11.1 基本概念

在介绍有关查找的算法之前，先介绍与查找相关的基本概念。

关键字（key）与**主关键字**（primary key）：数据元素中某个数据项的值。如果该关键字可以将所有的数据元素区别开来，也就是说可以唯一标识一个数据元素，则该关键字称为主关键字，否则称为**次关键字**（secondary key）。特别地，如果数据元素只有一个数据项，则数据元素的值即是关键字。

查找表（search table）：是由同一种类型的数据元素构成的集合。查找表中的数据元素是完全松散的，数据元素之间没有直接的联系。

查找（searching）：根据关键字在特定的查找表中找到一个与给定关键字相同的数据元素的操作。如果在表中找到相应的数据元素，则称查找是成功的，否则称查找是失败的。例如表11-1中为教师基本情况，如果要查找职称为“教授”并且性别是“男”的教师，则可以先利用职称将记录定位，然后在性别中查找为“男”的记录。

表11-1 教师基本情况信息表

编 号	姓 名	性 别	出 生 年 月	所 在 院 系	职 称	学 历
200109001	周永平	男	1970.09	计算机科学	教授	博士
(续)						
编 号	姓 名	性 别	出 生 年 月	所 在 院 系	职 称	学 历
200609002	王 超	男	1972.12	软件工程	教授	硕士
201109107	陈 冲	女	1982.01	文学院	讲师	本科
201309021	周俊阁	女	1983.11	化学系	讲师	博士
200509008	冯高峰	男	1977.07	数学系	副教授	博士

对查找表经常进行的操作有查询某个“特定的”的数据元素是否在查找表中、检索某个“特定的”数据元素的各种属性、在查找表中插入一个数据元素、从查找表中删除某个数据元素。若对查找表只进行前两种查找操作，则称此类查找表为静态查找表（static search table），相应的查找方法称为静态查找（static search）。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，则称此类查找为动态查找表（dynamic search table），相应的查找方法为动态查找（dynamic search）。

在日常生活中，人们几乎每天都在进行查找工作，例如在电话号码簿中查找某人的电话号码，在字典中查找某个字的读音和含义，电话号码簿和字典就可看做是一张查找表。

通常为了讨论查找的方便，要查找的数据元素中仅仅包含关键字。

平均查找长度 (average search length)：是指在查找过程中，需要比较关键字的平均次数，它是衡量查找算法的效率标准。平均查找长度的数学定义式为 $ASL = \sum_{i=1}^n P_i C_i$ 。其中， P_i 表示查找表中第*i*个数据元素的概率， C_i 表示在找到第*i*个数据元素时与关键字比较的次数。

11.2 静态查找

静态查找可分为顺序表、有序顺序表和索引顺序表的查找

11.2.1 顺序表的查找

顺序表的查找过程为从表的一端开始，逐个与关键字进行比较，若某个数据元素的关键字与给定的关键字相等，则查找成功，函数返回该数据元素所在的顺序表的位置；否则查找失败，返回0。

顺序表的存储结构如下。

```
#define MaxSize 100
typedef struct
{
    KeyType key;
}DataType;
typedef struct
{
    DataType list[MaxSize];
    int length;
}SSTable;
```

顺序表的查找算法描述如下。

```
int SeqSearch(SSTable S,DataType x)
/*
在顺序表中查找关键字为x
的元素，如果找到返回该元素在表中的位置，否则返回0*/
{
    int i=0;
    while(i<S.length&&S.list[i].key!=x.key) /*
从顺序表的第一个元素开始比较*/
        i++;
    if (S.list[i].key==x.key)
        return i+1;
    else
        return 0;
}
```

以上算法也可以通过设置监视哨的方法实现，算法描述如下。

```
int SeqSearch2(SSTable S,DataType x)
/*
设置监视哨S.list[0]
，在顺序表中查找关键字为x
的元素，如果找到返回该元素在表中的位置，否则返回0*/
{
    int i=S.length;
    S.list[0].key=x.key;          /*
将关键字存放在第0
号位置，防止越界*/
    while (S.list[i].key!=x.key)    /*
从顺序表的最后一个元素开始向前比较*/
        i--;
    return i;
}
```

以上算法是从表的最后一个元素开始与关键字进行比较，其中，S.list[0]被称为监视哨，可以防止出现数组越界。

下面分析带监视哨查找算法的效率。假设表中有n个数据元素，且数据元素在表中的出现的概率都相等，即 $1/n$ ，则顺序表在查找成功时的平均查找长度为 $ASL_{成功} = \sum_{i=1}^n P_i C_i = \sum_{i=1}^n \frac{1}{n} * (n-i+1) = \frac{n+1}{2}$ ，即查找成功时平均比较次数约为表长的一半。在查找失败时，即要查找的元素没有在表中，则每次比较都需要进行n+1次。

11.2.2 有序顺序表的查找

所谓有序顺序表，就是顺序表中的元素是以关键字进行有序排列的。对于有序顺序表的查找有两种方法，即顺序查找和折半查找。

1. 顺序查找

有序顺序表的顺序查找算法与顺序表的查找算法类似。在通常情况下，无须比较表中的所有元素。如果要查找的元素在表中，则返回该元素的序号，否则返回0。例如，一个有序顺序表的数据元素序列为{15, 21, 32, 39, 46, 55, 65, 76}，如果要查找数据元素关键字为52，从最后一个元素出发依次将

表中元素与52进行比较，当比较到46时就不需要再往前比较了。因前面的元素值都小于关键字52，因此，故表中不存在要查找的关键字。查找算法描述如下。

```
int SeqSearch2(SSTable S,DataType x)
/*
设置监视哨S.list[0]
，在有序顺序表中查找关键字为x
的元素，如果找到返回该元素在表中的位置，否则返回0*/
{
    int i=S.length;
    S.list[0].key=x.key;
    /*
将关键字存放在第0
号位置，防止越界*/
    while(S.list[i].key>x.key)
        /*
从有序顺序表的最后一个元素开始向前比较*/
        i--;
    return i;
}
```

设表中的元素个数为n，且要查找的数据元素在表中出现的概率相等即为1/n，则有序顺序表在查找成功时的平均查找长度为

$$ASL_{成功} = \sum_{i=1}^n P_i C_i = \sum_{i=1}^n \frac{1}{n} * (n - i + 1) = \frac{n+1}{2}$$
，即查找成功时平均比较次数约为表长的一半。在

查找失败时，即要查找的元素没有在表中，则有序顺序表在查找失败时的平均

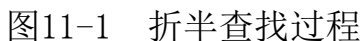
查找长度为
$$ASL_{失败} = \sum_{i=1}^n P_i C_i = \sum_{i=1}^n \frac{1}{n} * (n - i + 1) = \frac{n+1}{2}$$
，即查找失败时平均比较次数也同样约为表长的一半。

2. 折半查找

折半查找（binary search）又称为二分查找，这种查找算法要求待查找的元素序列必须是从小到大批列的有序序列。

折半查找即将待查找元素与表中间的元素进行比较，如果两者相等，则说明查找成功，否则利用中间位置将表分成两部分；如果待查找元素小于中间位

例如一个有序顺序表为(7, 15, 22, 29, 41, 55, 67, 78, 81, 99), 如果要查找元素67, 利用折半查找算法思想, 折半查找的过程如图11-1所示。



其中，图中low和high表示两个指针，分别指向待查找元素的下界和上界，指针mid指向low和high的中间位置，即 $mid = (low + high) / 2$ 。

初始时, $low=0$, $high=9$, $mid=(0+9)/2=4$, 因为 $list[mid]<x$, 所以需要在右半区间继续查找 x 。此时有 $low=5$, $high=9$, $mid=(5+9)/2=7$, 因为 $list[mid]>x$, 所以需要在左半区间继续查找 x 。此时有 $low=5$, $high=6$, $mid=5$, 因为 $list[mid]<x$, 所以需要在右半区间继续查找 x 。此时有 $low=6$, $high=6$, $mid=6$, 因为 $list[mid]==x$, 所以查找成功。

折半查找的算法描述如下。

```

int BinarySearch(SSTable S,DataType x)
/*
在有序顺序表中折半查找关键字为x
的元素，如果找到返回该元素在表中的位置，否则返回0*/
{
    int low,high,mid;
    low=0,high=S.length-1;          /*
设置待查找元素范围的下界和上界*/
    while(low<=high)
    {
        mid=(low+high)/2;
        if (S.list[mid].key==x.key)    /*
如果找到元素，则返回该元素所在的位置*/
            return mid+1;
        else if (S.list[mid].key<x.key) /*
如果mid
所指示的元素小于关键字，则修改low
指针*/
            low=mid+1;
        else if (S.list[mid].key>x.key) /*
如果mid
所指示的元素大于关键字，则修改high
指针*/
            high=mid-1;
    }
    return 0;
}

```

折半查找过程可以用一个判定树来描述。从图11-1中可以看出，查找元素41需要比较1次，查找元素78需要比较2次，查找元素55需要比较3次，查找元素67需要比较4次。整个查找过程可以用二叉判定树来表示，如图11-2所示。

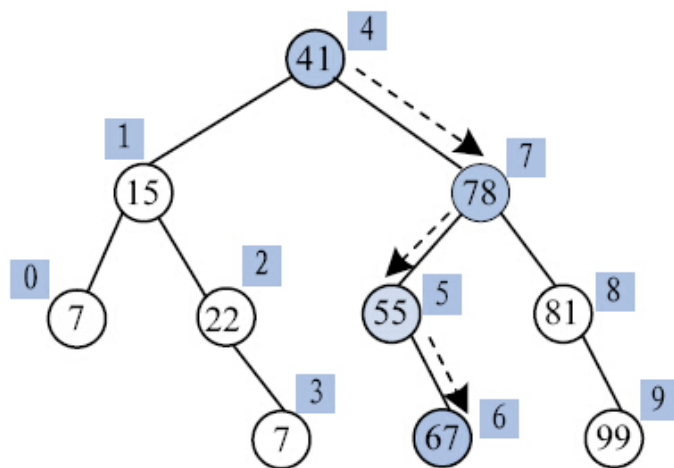


图11-2 折半查找元素67的判定树

其中，结点旁边的序号为该元素在序列中的下标。从图11-2中的判定树不难看出，查找元素67的过程正好是从根结点到元素值为67的结点的路径。查找元素67的比较次数正好是该元素在判定树中的所在层次。因此，如果表中有n个元素，折半查找成功时，至多需要比较的次数为 $\lfloor \log_2 n \rfloor + 1$ 。

对于具有n个结点的有序表（恰好构成一个深度为h的满二叉树）来说，有 $h = \lfloor \log_2(n+1) \rfloor$ ，二叉树中第i层的结点个数是 2^{i-1} 。假设表中每个元素的查找概率相等，即 $P_i = 1/n$ ，则有序表在折半查找成功时的平均查找长度为

ASL_{成功} = $\sum_{i=1}^n P_i C_i = \sum_{i=1}^h \frac{1}{n} * i * 2^{i-1} = \frac{n+1}{n} \log_2(n+1) + 1$ 。查找失败时，有序表的折半查找失败时的平均查找长度为

$$ASL_{失败} = \sum_{i=1}^n P_i C_i = \sum_{i=1}^h \frac{1}{n} * \log_2(n+1) = \log_2(n+1)。$$

11.2.3 索引顺序表的查找

当顺序表中的数据量非常大时，无论使用前述哪种查找算法都需要很长的时间，此时提高查找效率的一个常用方法就是在顺序表中建立索引表。建立索引表的方法是将顺序表分为几个单元，然后分别为这几个单元建立一个索引，原来的顺序表称为主表，提供索引的表称为索引表。索引表中只存放主表中要查找的数据元素的主关键字和索引信息。

图11-3是一个主表和一个按关键字建立的索引表结构图，其中，索引表包括两部分，即顺序表中每个单元的最大关键字和顺序表中每个单元的第一个元素的下标（即每个单元的起始地址）。

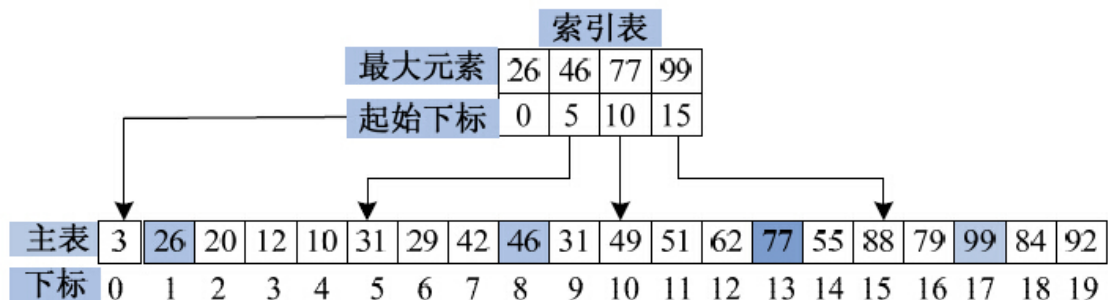


图11-3 索引顺序表

这样的表称为索引顺序表，要使查找效率高，索引表必须有序，但主表中的元素不一定要按关键字有序排列。[索引顺序表的查找](#) 也称[分块查找](#)。

从图11-3可以看出，索引表将主表分为4个单元，每个单元包含5个元素。要查找主表中的某个元素，需要分为两步查找，第一步需要确定要查找元素所在的单元；第二步在该单元查找指定的元素。例如，要查找元素62，首先需要将62与索引表中的元素进行比较，因为 $46 < 62 < 77$ ，所以需要在第3个单元查找，该单元的起始下标是10，因此从主表中的下标为10的位置开始查找62，直到找到该元素为止。如果在该单元中没有找到62，则说明主表中不存在该元素，查找失败。

因索引表中的元素的关键字是有序的，故在确定元素所在主表的单元时，既可采用顺序查找法也可采用折半查找法，但对于主表，只能采用顺序法查找。索引顺序表的平均查找长度可以表示为 $ASL = L_{index} + L_{unit}$ ， L_{index} 是索引表的平均查找长度， L_{unit} 是单元中元素的平均查找长度。

假设主表中的元素个数为 n ，并将该主表平均分为 b 个单元，且每个单元有 s 个元素，即 $b = n/s$ 。如果表中的元素查找概率相等，则每个单元中元素的查找概率就是 $1/s$ ，主表中每个单元的查找概率是 $1/b$ 。如果用顺序查找法查索引

表中的元素，则索引顺序表查找成功时的平均查找长度为

$ASL_{成功} = L_{index} + L_{unit} = \frac{1}{b} \sum_{i=1}^b i + \frac{1}{s} \sum_{j=1}^s j = \frac{b+1}{2} + \frac{s+1}{2} = \frac{1}{2} * (\frac{n}{s} + s) + 1$ 。如果用折半查找法查找索引表中的元素，则有 $L_{index} = \frac{b+1}{b} \log_2(b+1) + 1 \approx \log_2(b+1) - 1$ ，将其带入 $ASL_{成功} = L_{index} + L_{unit}$ 中，则索引顺序表查找成功时的平均查找长度为

$$ASL_{成功} = L_{index} + L_{unit} = \log_2(b+1) - 1 + \frac{1}{s} \sum_{j=1}^s j = \log_2(b+1) - 1 + \frac{s+1}{2} \approx \log_2(n/s+1) + \frac{s}{2}。$$

当然，如果主表中每个单元中的元素个数是不相等，就需要在索引表中增加一项，即用来存储主表中每个单元元素的个数，将这种利用索引表示的顺序表称为不等长索引顺序表。例如，一个不等长的索引表如图11-4所示。

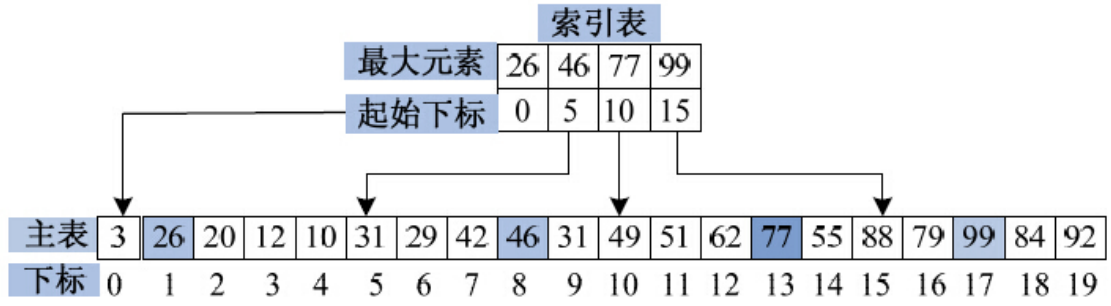


图11-4 不等长索引顺序表

11.2.4 静态查找应用举例

【例11-1】 给定一组元素序列，利用顺序表查找、有序顺序表查找和索引顺序表查找值x的元素。

【分析】 主要考查静态查找的3种算法思想。

1. 顺序表的查找方式的实现

这部分主要包括顺序表的查找、有序顺序表的查找和索引顺序表的查找。

程序实现代码如下。

```
int SeqSearch(SSTable S,DataType x)
/*
在顺序表中查找关键字为x
的元素，如果找到返回该元素在表中的位置，否则返回0*/
{
    int i=0;
    while(i<S.length&&S.list[i].key!=x.key) /*
从顺序表的第一个元素开始比较*/
        i++;
    if(S.list[i].key==x.key)
        return i+1;
    else
        return 0;
}
int BinarySearch(SSTable S,DataType x)
/*
在有序顺序表中折半查找关键字为x
的元素，如果找到返回该元素在表中的位置，否则返回0*/
{
    int low,high,mid;
    low=0,high=S.length-1; /*
设置待查找元素范围的下界和上界*/
    while(low<=high)
    {
        mid=(low+high)/2;
        if(S.list[mid].key==x.key) /*
如果找到元素，则返回该元素所在的位置*/
            return mid+1;
        else if(S.list[mid].key<x.key) /*
如果mid
所指示的元素小于关键字，则修改low
指针*/
            low=mid+1;
        else if(S.list[mid].key>x.key) /*
如果mid
所指示的元素大于关键字，则修改high
指针*/
            high=mid-1;
    }
    return 0;
}
int SeqIndexSearch(SSTable S,IndexTable T,int m,DataType x)
/*
在主表S
中查找关键字为x
的元素，T
为索引表。如果找到返回该元素在表中的位置，否则返回0*/
{
    int i,j,bl;
    for(i=0;i<m;i++) /*
通过索引表确定要查找元素所在的单元*/
        if(T[i].maxkey>=x.key)
            break;
    if(i>=m) /*
如果要查找的元素不在索引顺序表中，则返回0*/
        return 0;
    j=T[i].index; /*
要查找的元素在的主表的第j
单元*/
    if(i<m-1) /**bl
为第j
单元的长度*/
        bl=T[i+1].index-T[i].index;
    else
        bl=S.length-T[i].index;
```

```

        while(j<T[i].index+bl)
            if(S.list[j].key==x.key)          /*
如果找到关键字，则返回该关键字在主表中所在的位置*/
                return j+1;
            else
                j++;
        return 0;
}

```

2. 测试代码

这部分主要包括头文件、顺序表的类型定义、函数声明和主函数。程序实现如下。

```

/*
头文件和顺序表的类型定义*/
#include<stdio.h>
#include<stdlib.h>
#define MaxSize 100
#define IndexSize 20
typedef int KeyType;
typedef struct                                /*
元素的定义*/
{
    KeyType key;
}DataType;
typedef struct                                /*
顺序表的类型定义*/
{
    DataType list[MaxSize];
    int length;
}SSTable;
typedef struct                                /*
索引表的类型定义*/
{
    KeyType maxkey;
    int index;
}IndexTable[IndexSize];
int SeqSearch(SSTable S,DataType x);
int BinarySearch(SSTable S,DataType x);
int SeqIndexSearch(SSTable S,IndexTable T,int m,DataType x);
void main()
{
    SSTable S1={{265,55,15,67,9,64,88,50},8};
    SSTable S2={{20,28,37,40,46,55,64,76},8};
    SSTable S3={{12,8,26,20,19,40,29,43,35,33,56,50,63,55,58,75,64,66,85,79},20};
    IndexTable T={{26,0},{43,5},{63,10},{85,15}};
    DataType x={64};
    int pos;
    if((pos=SeqSearch(S1,x))!=0)
        printf("
顺序表的查找：关键字32
在主表中的位置是：%2d\n",pos);
    else
        printf("
查找失败！\n");
    if((pos=BinarySearch(S2,x))!=0)
        printf("
折半查找：关键字32
在主表中的位置是：%2d\n",pos);
    else
        printf("
... ..

```

```
查找失败! \n");  
        if ((pos=SeqIndexSearch(S3,T,4,x))!=0)  
            printf("  
索引顺序表的查找: 关键字32  
在主表中的位置是: %2d\n",pos);  
        else  
            printf("  
查找失败! \n");  
    }  
}
```

程序运行结果如图11-5所示。

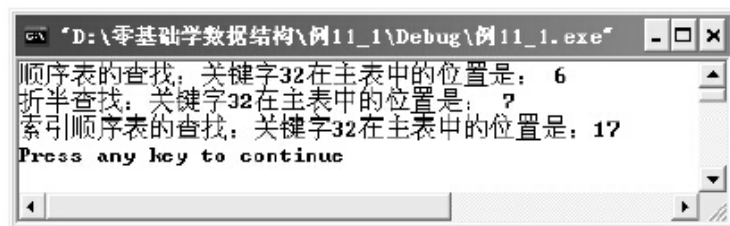


图11-5 表的静态查找程序运行结果

11.3 动态查找

动态查找的特点是表结构本身是在查找过程中动态生成的，即对于给定关键字key，若表中存在其关键字等于key的元素，则查找成功，否则插入关键字等于key的元素。动态查找包括二叉树和树结构两种类型的查找。本节主要介绍二叉排序树的查找、平衡二叉树的查找。

11.3.1 二叉排序树

二叉排序树也称为二叉查找树。二叉排序树的查找是一种常用的动态查找方法。下面介绍如何在二叉排序树中进行查找、插入结点和删除结点。

1. 什么是二叉排序树

二叉排序树（binary sort tree）或者是一棵空二叉树，或者二叉树具有以下性质。

（1）如果二叉树的左子树不为空，则左子树上的每一个结点的值均小于其对应根结点的值。

(2) 如果二叉树的右子树不为空，则右子树上的每一个结点的值均大于其对应根结点的值。

(3) 该二叉树的左子树和右子树也满足性质 (1) 和 (2)，即左子树和右子树也是一棵二叉排序树。

显然，这是一个递归的二叉排序树定义。例如，一棵二叉排序树如图11-6所示。

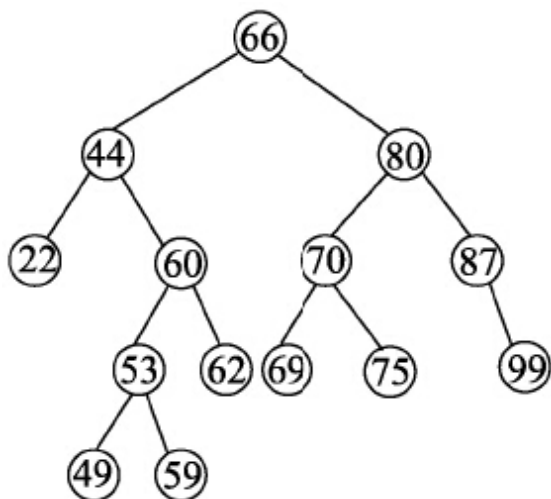


图11-6 二叉排序树

从图11-6中不难看出，每个结点元素值都大于其所有左子树结点元素的值，且小于其所有右子树中结点元素值。例如，80大于左孩子的结点元素值70，小于右孩子结点元素值87。

2. 查找算法

二叉排序树中每个结点的值都大于其所有左子树结点的值，而小于其所有右子树中结点的值，如果要查找与二叉树中某个关键字相等的结点，可以从根结点开始，与给定的关键字比较，如果相等，则查找成功；如果关键字小于根结点的值，则在其左子树中查找；如果给定的关键字大于根结点的值，则在其右子树中查找。

采用二叉树的链式存储结构的二叉排序树的类型定义如下。

```
typedef struct Node
{
    DataType data
;
    struct Node*lchild
, *rchild
;
}BiTreeNode
, *BiTree
;
```

二叉排序树的查找算法描述如下。

```
BiTree BSTSearch
(BiTree T
, DataType x
)
/*
二叉排序树的查找，如果找到元素x
，则返回指向结点的指针，否则返回NULL*/
{
    BiTreeNode*p
;
    if
    (T
    !=NULL
    )
    /*
    如果二叉排序树不为空*/
    {
        p=T
;
        while
        (p
        !=NULL
        )
        {
            if
            (p->data.key==x.key
            .
```

```

)          /*
如果找到，则返回指向该结点的指针*/
return p
;
else if
(x.key<p->data.key
)          /*
如果关键字小于p
指向的结点的值，则在左子树中查找*/
p=p->lchild
;
else
p=p->rchild
;          /*
如果关键字大于p
指向的结点的值，则在右子树中查找*/
}
}
return NULL
;
}

```

假设存在一棵二叉排序树，在该二叉排序树中查找元素75的过程如图11-7所示。从根结点开始，将75与66比较，因为 $75 > 66$ ，所以需要在右子树中查找75是否存在。因为 $75 < 80$ ，所以需要在元素80的结点的左子树中查找元素75。因为 $75 > 70$ ，所以需要在元素80的结点的右子树中查找75。因为元素70结点的右孩子结点与75相等，所以查找成功。

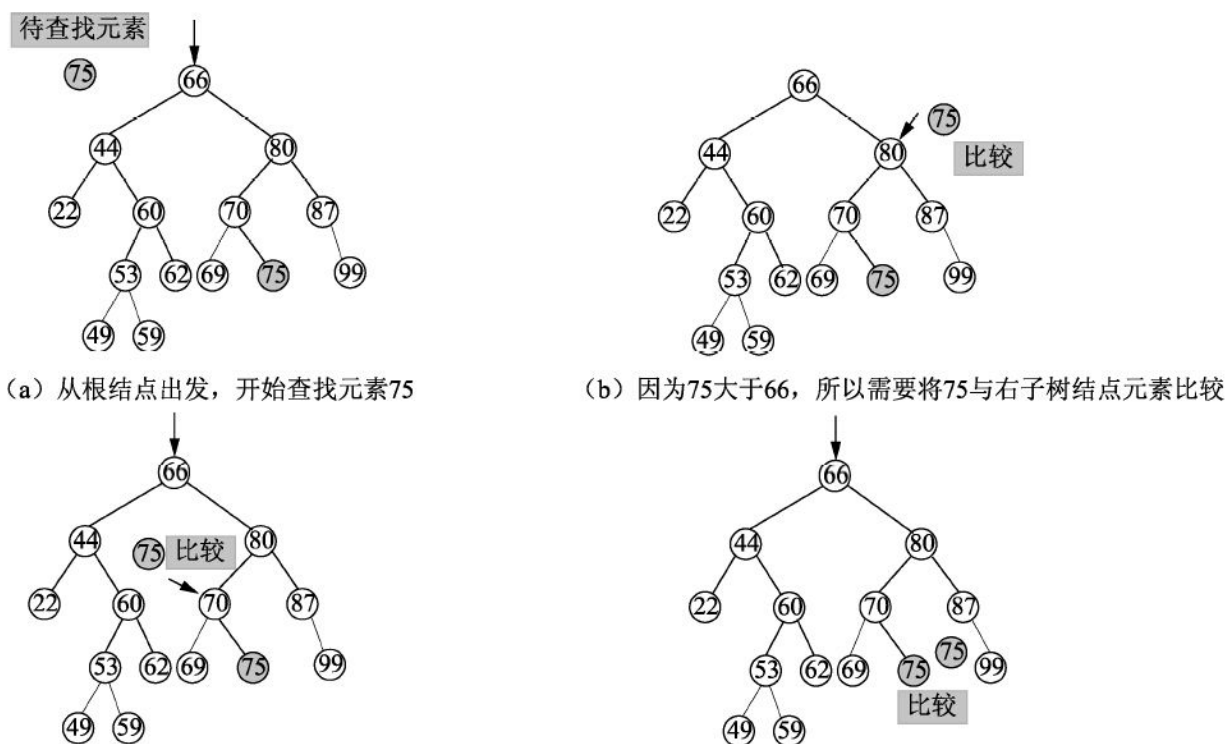


图11-7 二叉排序树的查找过程

如果要查找关键字为26的元素，当比较到结点为22的元素时，因为关键字22对应的结点不存在右子树，所以查找失败，返回NULL。

在二叉排序树的查找过程中，查找某个结点的过程正好是走了从根结点到要查找结点的路径，其比较的次数正好是路径长度+1，这类似于折半查找，与折半查找不同的是由n个结点构成的判定树是唯一的，而由n个结点构成的二叉排序树则不唯一。例如，对于图11-8所示的两棵二叉排序树，其元素的关键字序列分别是{66, 35, 81, 23, 47, 69, 92}和{23, 35, 47, 66, 69, 81, 92}。

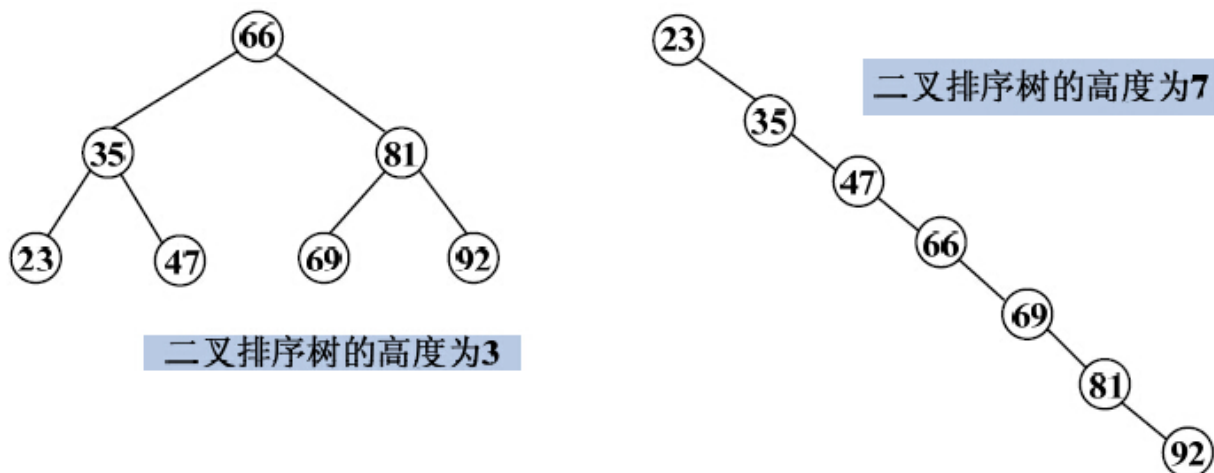


图11-8 两种不同形态的二叉排序树

假设每个元素的查找概率都相等，则图11-8所左边的树的元素序列平均查找长度为 $ASL_{成功} = (1/7) \times (1+2 \times 2+3 \times 4) = 17/7$ ，右边的树的平均查找长度为 $ASL_{成功} = (1/7) \times (1+2+3+4+5+6+7) = 28/7$ 。因此，树的平均查找长度与树的形态有关。如果二叉排序树有 n 个结点，则在最坏的情况下，平均查找长度为 n 。在最好的情况下，平均查找长度为 $\log_2 n$ 。

3. 二叉排序树的插入

二叉排序树的结构不是一次生成的，而是在查找的过程中，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结点或右孩子结点。因此，可以说二叉排序树的插入操作过程其实就是二叉排序树的建立过程。在算法的实现过程

中，需要设置一个指向下一个要访问结点的双亲结点指针parent，记下前驱结点的位置，以便在查找失败时进行插入操作。

若在二叉树中查找x结束后返回NULL，说明查找失败，需要将x插入二叉树排序；若parent->data.key<x.key，则需要将parent的左指针指向x，使x成为parent的左孩子结点；若parent->data.key>x.key，则需要将parent的右指针指向x，使x成为parent的右孩子结点；若二叉排序树为空树，则使x成为根结点。需要注意的是，二叉树的插入操作都是在叶子结点处进行。

二叉排序树的插入操作算法描述如下。

```
int BSTInsert
(BiTree*T
, DataType x
)
/*
二叉排序树的插入操作，如果树中不存在元素x
，则将x
插入正确的位置并返回1
，否则返回0*/
{
BiTreeNode*p
, *cur
, *parent=NULL
;
cur=*T
;
while
(
cur
!=NULL
)
{
if
(
cur->data.key==x.key
)
/*
如果二叉树中存在元素为x
的结点，则返回0*/
return 0
;
parent=cur
;
/*parent
指向cur
的前驱结点*/
if
```

```

    (x.key<cur->data.key
    ) /*
    如果关键字小于p
    指向的结点的值，则在左子树中查找*/
    cur=cur->lchild
    ;
    else
    cur=cur->rchild
    ; /*
    如果关键字大于p
    指向的结点的值，则在右子树中查找*/
    }
    p=
    (BiTreeNode*
    ) malloc
    (sizeof
    (BiTreeNode
    ) ); /*
    生成结点*/
    if
    (! p
    )
    exit
    (-1
    ) ;
    p->data=x
    ;
    p->lchild=NULL
    ;
    p->rchild=NULL
    ;
    if
    (! parent
    ) /*
    如果二叉树为空，则第一结点成为根结点*/
    *T=p
    ;
    else if
    (x.key<parent->data.key
    ) /*
    如果关键字小于parent
    指向的结点，则将x
    成为parent
    的左孩子*/
    parent->lchild=p
    ;
    else /*
    如果关键字大于parent
    指向的结点，则将x
    成为parent
    的右孩子*/
    parent->rchild=p
    ;
    return 1
    ;
    }

```

假设一个元素序列{55, 43, 66, 88, 18, 80, 33, 21, 72}，根据二叉排序树的插入算法思想，对应的二叉排序树插入过程如图11-9

所示。

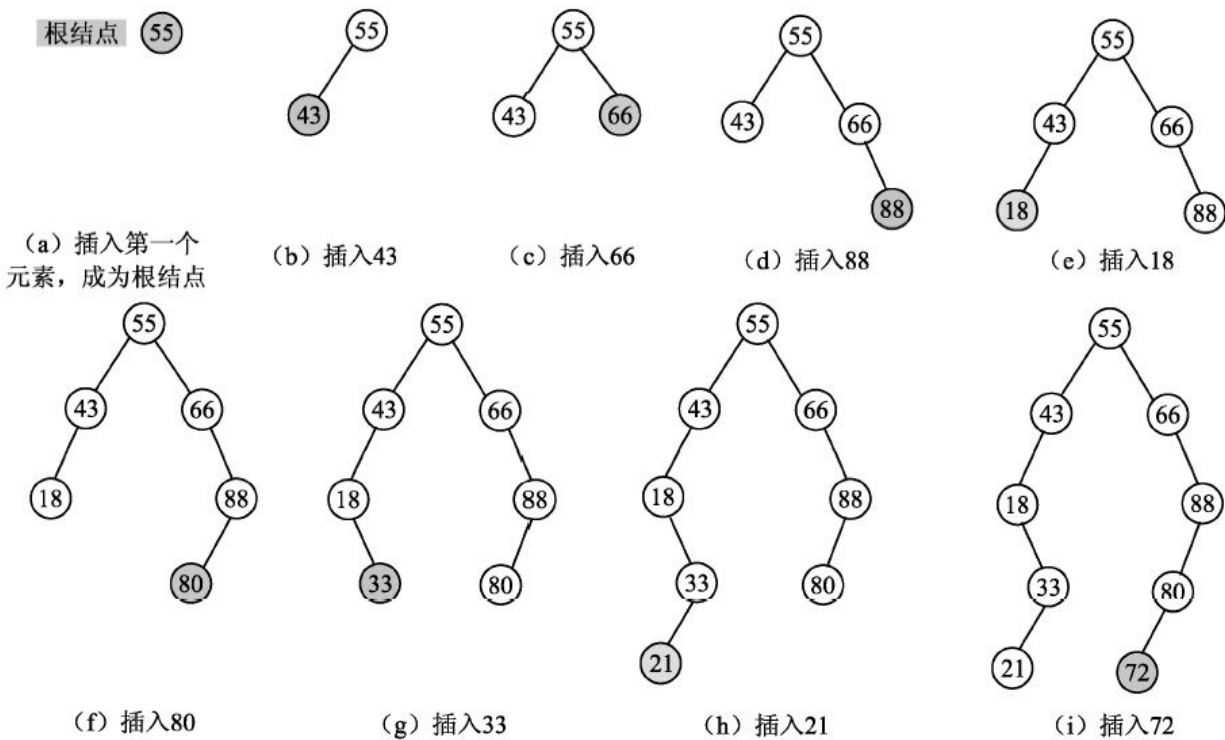


图11-9 二叉排序树的插入操作过程

从图11-9可以看出，通过中序遍历二叉排序树，可以得到一个关键字有序的序列{18, 21, 33, 43, 55, 66, 72, 80, 88}。

因此，构造二叉排序树的过程就是对一个无序的序列排序的过程，且每次插入结点都是叶子结点，在二叉排序树的插入操作过程中，不需要移动结点，仅需要移动结点指针，实现较为容易。

4. 二叉排序树的删除

对于一般的二叉树来说，删去树中的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变成森林，破坏了整棵树的结构。然而对于二叉树来说，删除树中的一个结点其实是删除有序序列的一个记录，只要在删除某个结点之后依旧保持二叉树的特性即可。

那么如何在二叉树排序树中删除一个结点呢？假设要删除的结点由指针s指示，指针p指向s的双亲结点，设s为f的左孩子结点。删除一个结点分为如下3种情况讨论，各种删除情形如图11-10所示。

(1) 如果s指向的结点为叶子结点，其左子树和右子树为空，删除叶子结点不会影响到树的结构特性，因此只需要修改p的指针即可。

(2) 如果s指向的结点只有左子树或只有右子树，在删除了结点*s后，只需要将s的左子树 s_L 或右子树 s_R 作为f的左孩子即 $p \rightarrow lchild = s \rightarrow lchild$ 或 $p \rightarrow lchild = s \rightarrow rchild$ 。

(3) 若s存在左子树和右子树，在删除结点T之前，二叉排序树的中序序列为 $\{\dots Q_L Q \dots X_L XY_L YTT_R P \dots\}$ ，因此，在删除了结点S之后，使该二叉树仍然保持原来的性质不变的调整方法有两种，一种是使结点T的左子树作为结点P的左子树，结点T的右子树作为结点Y的右子树；另一种是使结点T的直接前驱取代结点T，并删除T的直接前驱结点Y，然后令结点Y原来的左子树作为结点X的右子树，如图11-10所示。

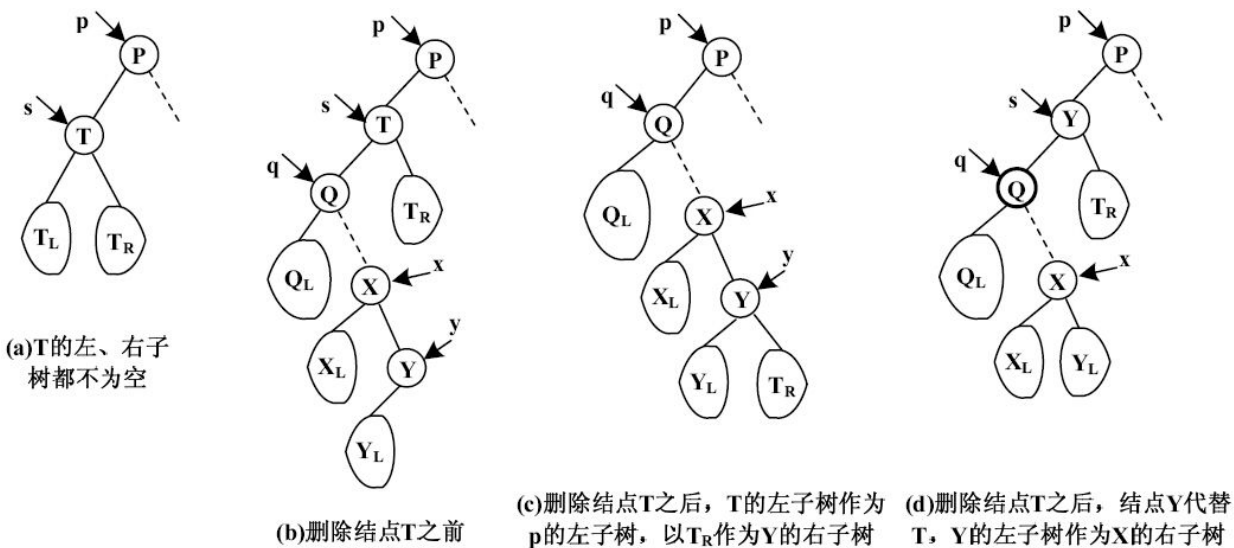


图11-10 删除二叉排序树结点后的调整

二叉排序树的删除操作算法描述如下。

```

int BSTDelete
(BiTree*T
, DataType x
)
/*
在二叉排序树T
中存在值为x
的数据元素时，删除该数据元素结点，并返回1
， 否则返回0*/
{
if
(! *T
)
/*
如果不存在值为x
的数据元素，则返回0*/
return 0
;
else
{
if
(x.key==
(*T
)->data.key
) /*
如果找到值为x
的数据元素，则删除该结点*/
DeleteNode
(T
);
else if
(*T
)->data.key>x.key
) /*
如果当前元素值大于x
...

```

```

的值，则在该结点的左子树中查找并删除之*/
BSTDelete
(
    &
    (*T
) ->lchild
, x
);
else
    /*
    如果当前元素值小于x
    的值，则在该结点的右子树中查找并删除之*/
    BSTDelete
    (
        &
        (*T
    ) ->rchild
    , x
    );
return 1
;
}
}
void DeleteNode
(BiTree*s
)
/*
从二叉排序树中删除结点s
，并使该二叉排序树性质不变*/
{
    BiTree q
    , x
    , y
    ;
    if
    (
        (! (*s
    ) ->rchild
    )
    )
        /*
        如果s
        的右子树为空，则直接s
        的左子树*/
        {
            q=*s
            ;
            *s=
            (*s
            ) ->lchild
            ;
            free
            (q
            );
        }
        else if
        (
            (! (*s
            ) ->lchild
            )
        )
            /*
            如果s
            的左子树为空，则直接s
            的右子树*/
            {
                q=*s
                ;
                *s=
                (*s
                ) ->rchild
                ;
                free
                (q
                );
            }
        }
    }
}

```

```

else
/*
如果s
的左、右子树都存在，则使s
的直接前驱结点代替s
，并使其直接前驱结点的左子树成为其双亲结点的右子树结点*/
{
x=*s
;
y=
(*s
)->lchild
;
while
(y->rchild
) /*
查找s
的直接前驱结点，y
为s
的直接前驱结点，x
为y
的双亲结点*/
{
x=y
;
y=y->rchild
;
}
(*s
)->data=y->data /*
结点s
被y
取代*/
if
(x
!=*s
) /*
如果结点s
的左孩子结点存在右子树*/
x->rchild=y->lchild
; /*
使y
的左子树成为x
的右子树*/
else /*
如果结点s
的左孩子结点不存在右子树*/
x->lchild=y->lchild
; /*
使y
的左子树成为x
的左子树*/
free
(y
);
}
}

```

在算法的实现过程中，通过调用Delete (T) 来完成删除当前结点的操作，而函数BSTDelete (& (*T) ->lchild, x) 和

BSTDelete (& (*T) ->rchild, x) 则是实现在删除结点后，利用参数 T->lchild和T->rchild完成连接左子树和右子树，使二叉排序树性质保持不变。

5. 二叉排序树应用举例

【例11-2】 编写算法，利用二叉树的插入算法创建一棵二叉排序树，然后输入一个元素，查找该元素是否存在，最后输出这棵树的中序序列。

算法实现代码如下。

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
typedef int KeyType
;
typedef struct          /*
元素的定义*/
{
    KeyType key
;
}DataType
;
typedef struct Node      /*
二叉排序树的类型定义*/
{
    DataType data
;
    struct Node*lchild
, *rchild
;
}BiTreeNode
, *BiTree
;
void DeleteNode
(BiTree*s
);
int BSTDelete
(BiTree*T
, DataType x
);
void InOrderTraverse
(BiTree T
);
```

```

BiTree BSTSearch
    (BiTree T
    , DataType x
    );
int BSTInsert
    (BiTree*T
    , DataType x
    );
void main
    ()
    {
    BiTree T=NULL
    , p
    ;
    DataType table[]={55
    , 43
    , 66
    , 88
    , 18
    , 80
    , 33
    , 21
    , 72}
    ;
    int n=sizeof
    (table
    )/sizeof
    (table[0]
    );
    DataType x={80}
    , s={18}
    ;
    int i
    ;
    for
    (i=0
    ; i<n
    ; i++
    )
    BSTInsert
    (&T
    , table[i]
    );
    printf
    ("
    关键字序列为: "
    );
    for
    (i=0
    ; i<n
    ; i++
    )
    printf
    ("%4d"
    , table[i]
    );
    printf
    ("\n"
    );
    printf
    ("
    中序遍历二叉排序树得到的序列为: \n"
    );
    InOrderTraverse
    (T
    );
    p=BSTSearch

```

```

    (T
    , x
    );
    if
    (p
    !=NULL
    )
    printf
    ("\n
    二叉排序树查找:
    元素关键字%d
    查找成功.\n"
    , x
    );
    else
    printf
    ("\n
    二叉排序树查找:
    没有找到元素%d.\n"
    , x
    );
    BSTDelete
    (&T
    , s
    );
    printf
    ("
    删除元素%d
    后, 中序遍历二叉排序树得到的序列为: \n"
    , s.key
    );
    InOrderTraverse
    (T
    );
    printf
    ("\n"
    );
}
void InOrderTraverse
(BiTree T
)
/*
中序遍历二叉排序树的递归实现*/
{
    if
    (T
    )
    /*
    如果二叉排序树不为空*/
    {
        InOrderTraverse
        (T->lchild
        ); /*
        中序遍历左子树*/
        printf
        ("%4d"
        , T->data
        ); /*
        访问根结点*/
        InOrderTraverse
        (T->rchild
        ); /*
        中序遍历右子树*/
    }
}

```

程序运行结果如图11-11所示。



图11-11 二叉排序树的查找、删除操作程序运行结果

11.3.2 平衡二叉树

若二叉排序树的深度为 n ，在最坏的情况下平均查找长度为 n ，为了减少二叉排序树的查找次数，需要对二叉树排序树进行平衡化处理，平衡化处理得到的二叉树称为平衡二叉树。

1. 什么是平衡二叉树

平衡二叉树（balanced binary tree）又称为AVL树，它或者是一棵空树，或者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。

若将二叉树中结点的**平衡因子BF**（balance factor）定义为结点的左子树的深度减去右子树的深度，则平衡二叉树中每个结点的平衡因子只可能是-1、0和1。如图11-12所示为两棵平衡二叉树，结点的右

边数值表示平衡因子，因为该二叉树既是二叉排序树又是平衡树，因此，该二叉树称为平衡二叉排序树。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11-13所示为两棵不平衡的二叉树。

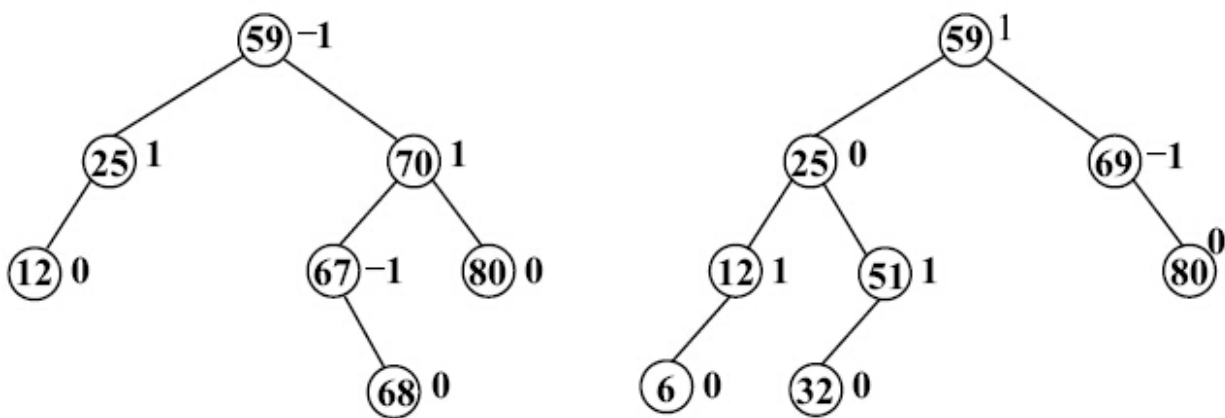


图11-12 平衡二叉树

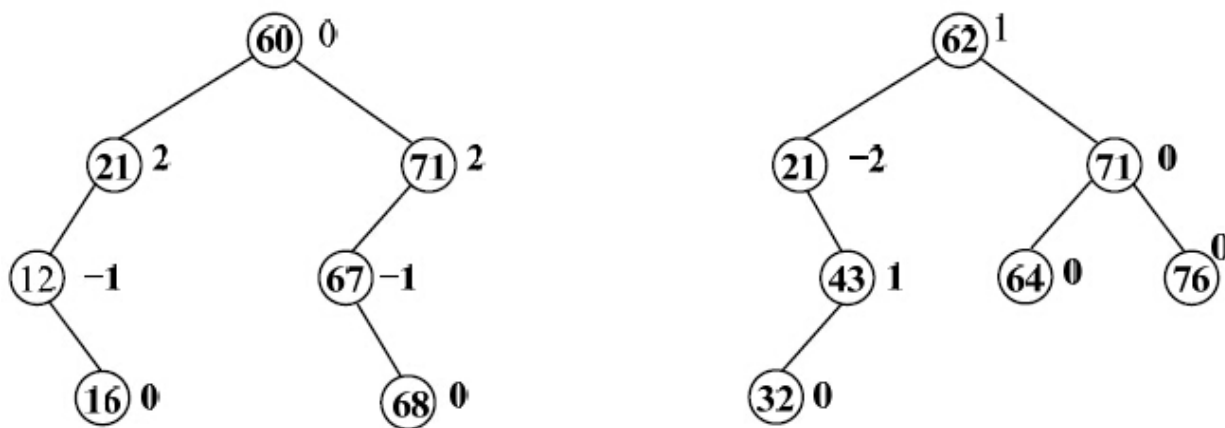


图11-13 不平衡二叉树

如果二叉排序树是平衡二叉树，则其平均查找长度与 $\log_2 n$ 是同数量级的。

2. 二叉排序树的平衡处理

在二叉排序树中插入一个新结点后，如何保证该二叉树是平衡二叉排序树呢？设有一个关键字序列{7, 36, 45, 78, 65}，依照此关键字序列建立二叉排序树，且使该二叉排序树是平衡二叉排序树。

初始时，二叉树为空树，因此是平衡二叉树。插入结点7和36后，该二叉树依然是平衡的，结点7和结点36的平衡因子分别为-1和0。当插入结点45后，结点7的平衡因子变为-2，二叉树变为不平衡，这需要进行调整。以结点36为轴进行逆时针旋转，将二叉树变为以36为根，各个结点的平衡因子都为0，二叉树恢复了平衡。继续插入结点78，二叉树仍然为平衡的。当插入结点65后，该二叉树又失去了平衡，为了保持二叉排序树的性质，又要保证该二叉树是平衡的，需要进行两次调整，先以结点78为轴进行顺时针旋转，然后以结点65为轴进行逆时针旋转。构造平衡二叉排序树的过程如图11-14所示。

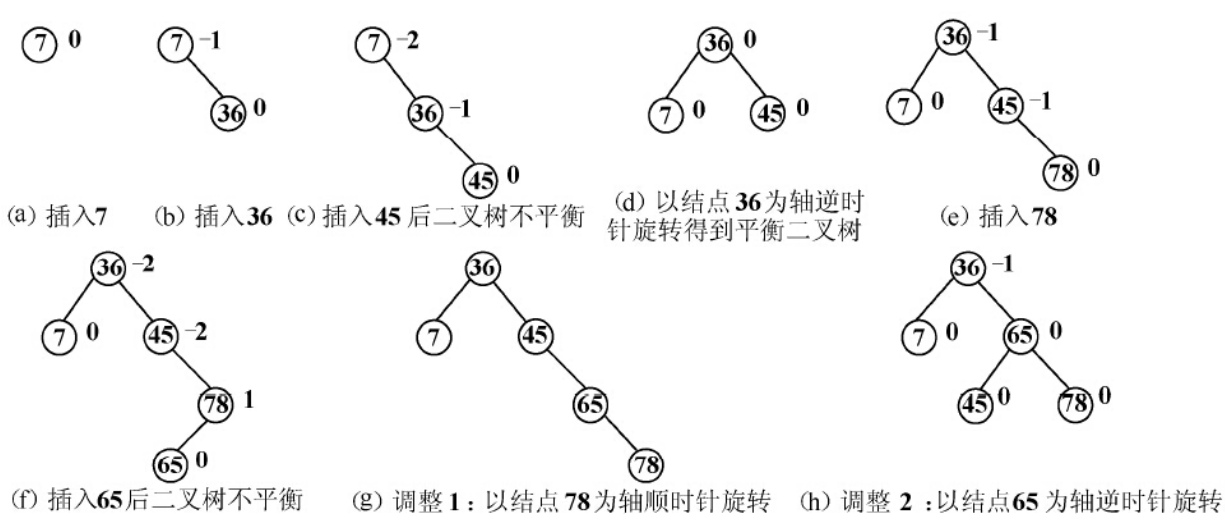


图11-14 平衡二叉树的调整过程

通过上述平衡化处理可以得出一个结论，即通过让插入点最近的祖先结点恢复平衡使上一层祖先结点恢复平衡。因此，为了使二叉排序树恢复平衡，需要从离插入点最近的结点开始调整。平衡化二叉排序树可分为以下4种情形。

(1) LL型

LL型是指在离插入点最近的失衡结点的左子树的左子树中插入结点，导致二叉排序树失去平衡，如图11-15所示。距离插入点最近的失衡结点为A，插入新结点X后，结点A的平衡因子由1变为2，该二叉排序树失去平衡。为了使二叉树恢复平衡且保持二叉排序树的性质不变，可以使结点A作为结点B的右子树，结点B的右子树作为结点A的左子树。这样就恢复了该二叉排序树的平衡，这相当于以结点B为轴，对结点A进行顺时针旋转。

为平衡二叉排序树的每个结点增加一个域bf，用来表示对应结点的平衡因子。平衡二叉排序树的类型描述如下。

```
typedef struct BSTNode /*
平衡二叉排序树的类型定义*/
{
    DataType data
;
    int bf
; /*
结点的平衡因子*/
    struct BSTNode*lchild
, *rchild
; /*
左、右孩子指针*/
```

```

}BSTNode
, *BSTree
;

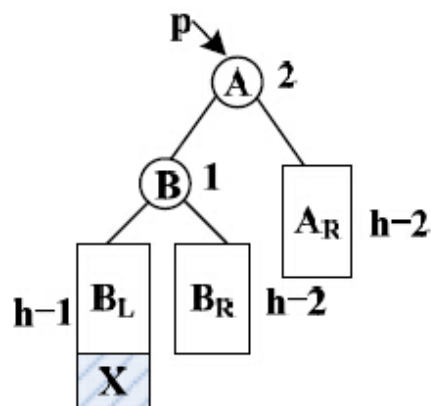
```

对LL型二叉排序树的平衡化处理代码如下。

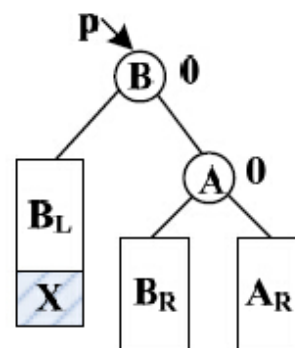
```

BSTree b
;
b=p->lchild
;
指向p
的左子树的根结点*/
p->lchild=b->rchild
;
/*
将b
的右子树作为p
的左子树*/
b->rchild=p
;
/*p
作为b
的右子树*/
p->bf=b->bf=0
;
/*
修改平衡因子*/

```



(a) 插入结点X后二叉树失去平衡



(b) 以结点B为轴进行顺时针旋转调整，使二叉树恢复平衡

图11-15 LL型二叉排序树的调整

(2) LR型

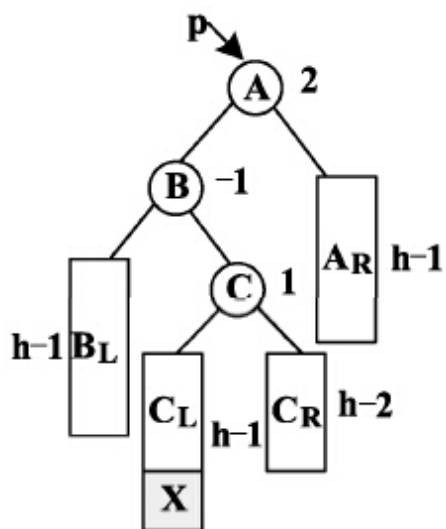
LR型是指在离插入点最近的失衡结点的左子树的右子树中插入结点，导致二叉排序树失去平衡，如图11-16所示。

距离插入点最近的失衡结点为A，在C的左子树 C_L 下插入新结点X后，结点A的平衡因子由1变为2，该二叉排序树失去平衡。为了使二叉树恢复平衡且保持二叉排序树的性质不变，以结点B为轴，对结点C先做了一次逆时针旋转；然后以结点C为轴对结点A做了一次顺时针旋转，即结点B作为结点C的左子树，结点C的左子树作为结点B的右子树。将结点C作为新的根结点，结点A作为C的右子树的根结点，结点C的右子树作为A的左子树。这样二叉排序树就恢复平衡了。

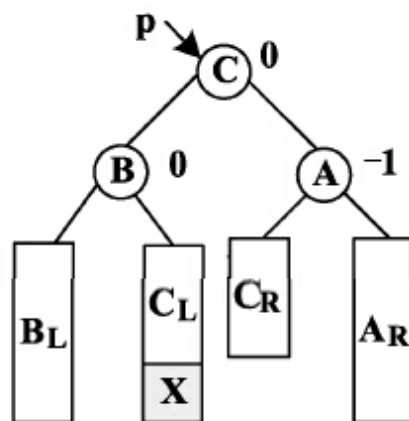
(3) RL型

RL型是指在离插入点最近的失衡结点的右子树的左子树中插入结点，导致二叉排序树失去平衡，如图11-17所示。

距离插入点最近的失衡结点为A，在C的右子树 C_R 下插入新结点X后，结点A的平衡因子由-1变为-2，失去平衡。为了使二叉排序树恢复平衡，以结点B为轴，对结点C先做了一次顺时针旋转；然后以结点C为轴对结点A做了一次逆时针旋转，即结点B作为结点C的右子树，结点C的右子树作为结点B的左子树。将结点C作为新的根结点，结点A作为C的左子树的根结点，结点C的左子树作为A的右子树。

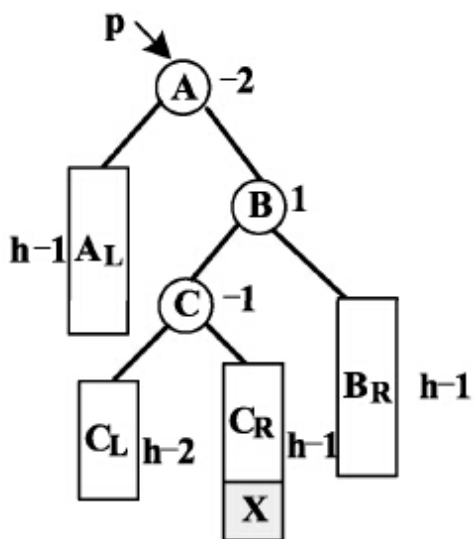


(a) 插入结点X后二叉树失去平衡

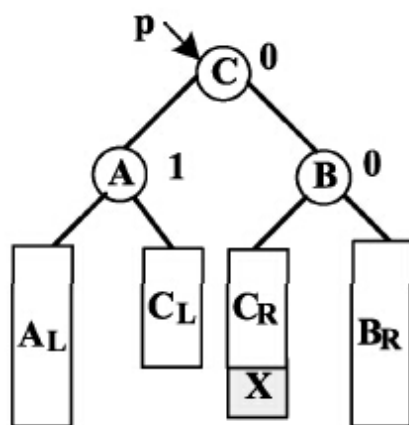


(b) 以结点B为轴进行逆时针旋转，
然后以C为轴对A进行顺时针旋转

图11-16 LR型二叉排序树的调整



(a) 插入结点X后二叉树失去平衡



(b) 以结点B为轴对C进行顺时针旋转，

然后以C为轴对A进行逆时针旋转

图11-17 RL型二叉排序树的调整

(4) RR型

RR型是指在离插入点最近的失衡结点的右子树的右子树中插入结点，导致二叉排序树失去平衡，如图11-18所示。

距离插入点最近的失衡结点为A，在结点B的右子树 B_R 下插入新结点X后，结点A的平衡因子由-1变为-2，失去平衡。为了使二叉排序树恢复平衡，以结点B为轴，对结点A做一次逆时针旋转，即结点A作为B的左子树的根结点，结点B的左子树作为A的右子树。

综合分析上述4种情形，在平衡二叉排序树中插入一个结点e后，保持二叉排序树平衡的算法描述如下。

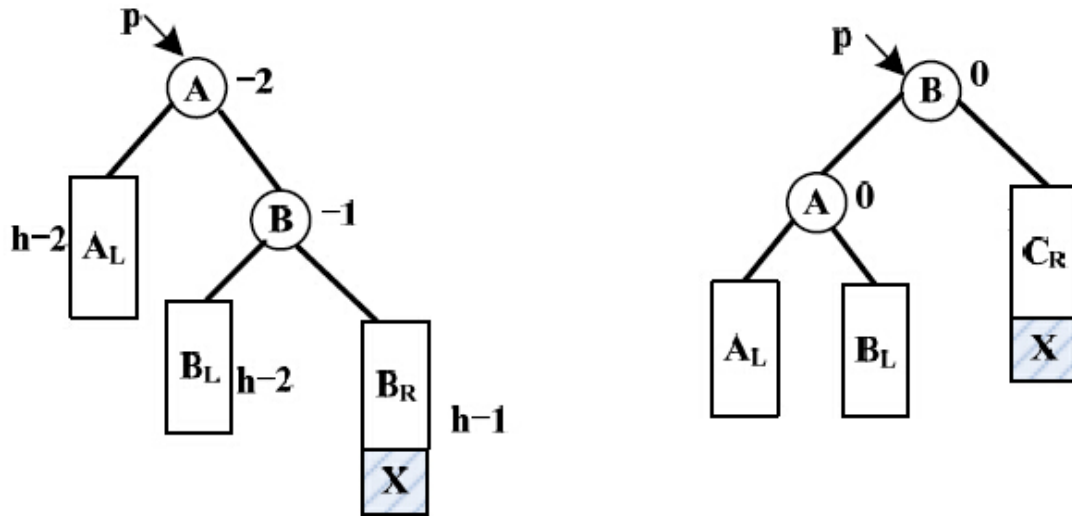
①若平衡二叉排序树是空树，则插入的结点e作为根结点，并使该树的深度增1。

②若二叉树中已存在与结点e的关键字相等的结点，则无须插入。

③若结点e的关键字小于要插入位置的结点的关键字，则将e插入该结点的左子树位置，并使该结点的左子树高度增1，同时修改该结点的平衡因子；如果该结点的平衡因子绝对值大于1，则需要进行平衡化处理。

④若结点e的关键字大于要插入位置的结点的关键字，则将e插入该结点的右子树位置，并将该结点的右子树高度增1，同时修改该结点

的平衡因子；如果该结点的平衡因子绝对值大于1，则进行平衡化处理。



(a) 插入结点X后二叉树失去平衡 (b) 以结点B为轴对A进行逆时针旋转

图11-18 RR型二叉排序树的调整过程

二叉排序树的平衡化处理算法可以实现包括平衡二叉排序树的插入操作和平衡处理。平衡二叉排序树的插入算法实现如下。

```

int InsertAVL
(BSTree*T
, DataType e
, int*taller
)
/*
如果在平衡的二叉排序树T
中不存在与e
有相同关键字的结点，则将e
插入并返回1
，否则返回0*/
/*
如果插入新结点后使二叉排序树失去平衡，则进行平衡旋转处理*/
{
if
(! *T
)
/*
如果二叉排序树为空，则插入新结点，将taller
置为1*/
{
*T=
(BSTree

```

```

) malloc
(sizeof
BSTNode
) );
(*T
)->data=e
;
(*T
)->lchild=
(*T
)->rchild=NULL
;
(*T
)->bf=0
;
*taller=1
;
}
else
{
if
(e.key==
(*T
)->data.key
) /*
如果树中存在和e
的关键字相等，则不进行插入操作*/
{
*taller=0
;
return 0
;
}
if
(e.key<
(*T
)->data.key
) /*
如果e
的关键字小于当前结点的关键字，则继续在*T
的左子树中进行查找*/
{
if
(! InsertAVL
(&
(*T
)->lchild
, e
, taller
) )
return 0
;
if
(*taller
) /*
已插入*T
的左子树中且左子树“长高”*/
{
switch
( (*T
)->bf
) /*
检查*T
的平衡度*/
{
case 1: /*
在插入之前，左子树比右子树高，需要作左平衡处理*/

```

```

LeftBalance
(
T
);
*taller=0
;
break
;
case 0: /*
在插入之前，左、右子树等高，树增高将taller
置为1*/
(*T
)->bf=1
;
*taller=1
;
break
;
case 1: /*
在插入之前，右子树比左子树高，现左、右子树等高*/
(*T
)->bf=0
;
*taller=0
;
}
}
}
else
{ /*
应继续在*T
的右子树中进行搜索*/
if
(! InsertAVL
(&
(*T
)->rchild
, e
, taller
))
return 0
;
if
(*taller
) /*
已插入T
的右子树且右子树“长高”*/
{
switch
( (*T
)->bf
) /*
检查T
的平衡度*/
{
case 1: /*
在插入之前，左子树比右子树高，现左、右子树等高*/
(*T
)->bf=0
;
*taller=0
;
break
;
case 0: /*
在插入之前，左、右子树等高，现因右子树增高而使树增高*/
(*T
)->bf=-1

```

```

;
*taller=1
;
break
;
case-1: /*
在插入之前，右子树比左子树高，需要作右平衡处理*/
RightBalance
(T
);
*taller=0
;
}
}
}
}
return 1
;
}

```

3. 二叉排序树的平衡处理算法

二叉排序树的平衡处理算法实现包括LL型、LR型、RL型和RR型4种情形。

(1) LL型的平衡处理算法

对于LL型的失去平衡的情形，只需要对离插入点最近的失衡结点进行一次顺时针旋转处理即可。其算法实现如下。

```

void RightRotate(BSTree *p)
/*
对以*p
为根的二叉排序树进行右旋，处理之后p
指向新的根结点，即旋转处理之前的左子树的根结点*/
{
    BSTree lc;
    lc=(*p)->lchild; /*lc
指向p
的左子树的根结点*/
    (*p)->lchild=lc->rchild; /*
将lc
的右子树作为p
的左子树*/
    lc->rchild=*p;
    (*p)->bf=lc->bf=0;
    *p=lc; /*p
... ..

```



```

作右旋平衡处理*/
    }
}

```

(3) RL型的平衡处理算法

对于RL型的失去平衡的情形，需要进行两次旋转处理，先进行一次顺时针旋转，然后再进行一次逆时针旋转处理。其算法实现如下。

```

void RightBalance(BSTree *T)
/*
对以指针T
所指结点为根的二叉树作右旋转平衡处理，并使T
指向新的根结点*/
{
    BSTree rc,rd;
    rc=(*T)->rchild;          /*rc
指向*T
的右子树根结点*/
    switch(rc->bf)             /*
调用RR
型平衡处理。检查*T
的右子树的平衡度，并作相应平衡处理*/
    {
        case -1:
            (*T)->bf=rc->bf=0;
            LeftRotate(T);
            break;

        case 1:
            /*RL
型平衡处理。新结点插入*T
的右孩子的左子树上，需要进行双旋处理*/
            rd=rc->lchild;      /*rd
指向*T
的右孩子的左子树的根结点*/
            switch(rd->bf)      /*
修改*T
及其右孩子的平衡因子*/
            {
                case -1:
                    (*T)->bf=1;
                    rc->bf=0;
                    break;

                case 0:
                    (*T)->bf=rc->bf=0;
                    break;

                case 1:
                    (*T)->bf=0;
                    rc->bf=-1;
            }
            rd->bf=0;
            RightRotate(&(*T)->rchild); /*
对*T
的右子树作右旋平衡处理*/
            LeftRotate(T);      /*

```

```
对*T  
作左旋平衡处理*/  
}  
}
```

(4) RR型的平衡处理算法

对于RR型的失去平衡的情形，只需要对离插入点最近的失衡结点进行一次逆时针旋转处理即可。算法实现如下。

```
void LeftRotate(BSTree *p)  
/*  
对以*p  
为根的二叉排序树进行左旋，处理之后p  
指向新的根结点，即旋转处理之前的右子树的根结点*/  
{  
    BSTree rc;  
    rc=(*p)->rchild;                /*rc  
指向p  
的右子树的根结点*/  
    (*p)->rchild=rc->lchild;        /*  
将rc  
的左子树作为p  
的右子树*/  
    rc->lchild=*p;  
    *p=rc;                          /*p  
指向新的根结点*/  
}
```

平衡二叉排序树的查找过程与二叉排序树类似，其比较次数最多为树的深度，如果树的结点个数为 n ，则时间复杂度为 $O(\log_2 n)$ 。

11.4 B-树与B+树

本节介绍一般动态查找树，包括B-树与B+树。

11.4.1 B-树

与二叉排序树类似，B-树是一种特殊的m叉动态查找树。下面介绍B-树的结构与查找算法。

1. 什么是B-树

B-是一种平衡的多路查找树，也称为m路（叉）查找树，它在文件系统中很有用。一棵m路（阶）B-树或者是一棵空树，或者是满足以下性质的m叉树。

- （1）任何一个结点最多有m棵子树。
- （2）或者是根结点，或者是叶子结点，或者至少有两棵子树。
- （3）除根结点外，所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树。
- （4）所有的非终端结点的结构如下。

<i>n</i>	P₀	K₁	P₁	K₁	...	K_n	P_n
----------	----------------------	----------------------	----------------------	----------------------	-----	----------------------	----------------------

其中， n 为结点中关键字的个数， $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ ， P_i 为指向子树根结点的指针，并且 P_i 指向子树的每个结点关键字都小于 K_{i+1} （ $i=0, 1, \dots, n-1$ ）。

(5) 所有叶子结点处于同一层次上，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）。

例如，一棵深度为4的4阶的B-树如图11-19所示。

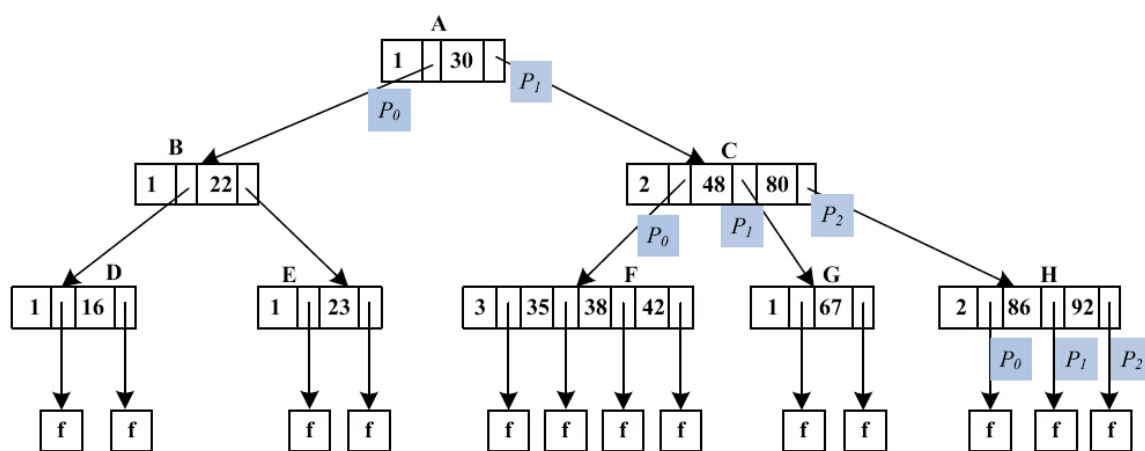


图11-19 一棵深度为4的4阶的B-树

B-树中查找某个关键字与二叉排序树的查找类似。例如，要查找关键字为92的元素，从根结点出发，先将92与A结点的关键字30比较，因 $92 > 30$ ，故需要在 P_1 指向的子树中查找。指针 P_1 指向结点C，将92与结点C中的关键字逐个比较，因有 $92 > 80$ ，故应在 P_2 指向的子树中查找。指针 P_2 指向结点H，故将92与结点H中的关键字逐个比较，结点H中存在关键字为92的元素，故查找成功。

2. B-树的查找

B-树的查找其实是对二叉排序树查找的扩展，与二叉排序树不同的地方是，B-树中每个结点有不只一棵子树。在B-树中查找某个结点时，需要先判断要查找的结点在哪棵子树上，然后在结点中逐个查找目标结点。

B-树的类型描述如下。

```
#define m 4                                /*B-
树的阶数*/
typedef struct BTNode                       /*B-
树类型定义*/
{
    int keynum;                             /*
每个结点中的关键字个数*/
    struct BTNode *parent;                  /*
指向双亲结点*/
    KeyType data[m+1];                     /*
结点中关键字信息*/
    struct BTNode *ptr[m+1];               /*
指针向量*/
}BTNode,*BTree;
```

B-树的查找算法描述如下。

```
typedef struct                             /*
返回结果类型定义*/
{
    BTNode *pt;                             /*
指向找到的结点*/
    int pos;                                /*
关键字在结点中的序号*/
    int flag;                               /*
查找成功与否标志*/
}result;
result BTreeSearch(BTree T, KeyType k)
/*
在m
阶B-
树T
上查找关键字k
，返回结果为r(pt,pos,flag)
。如果查找成功，则标志flag
为1
，pt
指向关键字为 k
的结点，否则特征值tag=0*/
{
    BTree p=T,q=NULL;
    int i=0,found=0;
    result r;
    while(p&&!found)
    {
        i=Search(p,k);                     /*p->data[i].key
≤k<p->data[i+1].key*/
        if(i>0&&p->data[i].key==k.key)      /*
如果找到要查找的关键字，标志found
...
```

```

        置为1*/
                                found=1;
                                else
                                {
                                    q=p;
                                    p=p->ptr[i];
                                }
                                }
                                if(found)
                                查找成功，返回结点的地址和位置序号*/
                                {
                                    r.pt=p;
                                    r.flag=1;
                                    r.pos=i;
                                }
                                else
                                查找失败，返回k
                                的插入位置信息*/
                                {
                                    r.pt=q;
                                    r.flag=0;
                                    r.pos=i;
                                }
                                return r;
                            }
                            int Search(BTree T,KeyType k)
                            /*
                            在T
                            指向的结点中查找关键字为k
                            的序号*/
                            {
                                int i=1,n=T->keynum;
                                while(i<=n&&T->data[i].key<=k.key)
                                    i++;
                                return i-1;
                            }

```

3. B-树的插入操作

在B-树上插入结点与在二叉树上插入结点类似，都是插入前后保证B-树仍然是一棵排序树，即结点左子树中每个结点的关键字小于根结点的关键字，右子树结点的关键字大于根结点的关键字。但由于B-树结点中的关键字个数必须大于等于 $\lceil m/2 \rceil - 1$ ，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，否则对该结点进行分裂。

例如，图11-20所示为一棵3阶的B-树（省略了叶子结点），当给定一棵B-树的阶之后，就确定了这棵树的最少子树个数为和最多子树个数，这棵B-

树确定了每个结点要求最少1个关键字，最多2个关键字。

假设要在该B-树中依次插入关键字53、22、85和59，过程如下。

①插入关键字53。首先从根结点出发，确定关键字53应插入的位置，因 $51 < 53 < 70$ ，故应插入结点E中。插入后结点E中的关键字个数大于1（即 $\lceil m/2 \rceil - 1$ ）小于2（即 $m - 1$ ），故插入成功。插入后B-树如图11-21所示。

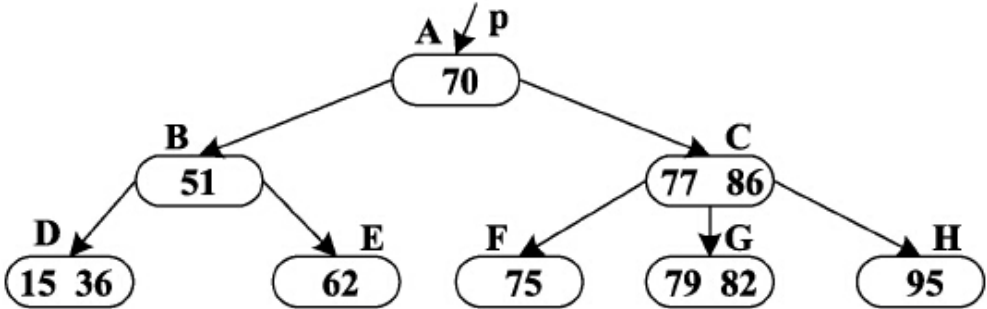


图11-20 一棵3阶的B-树

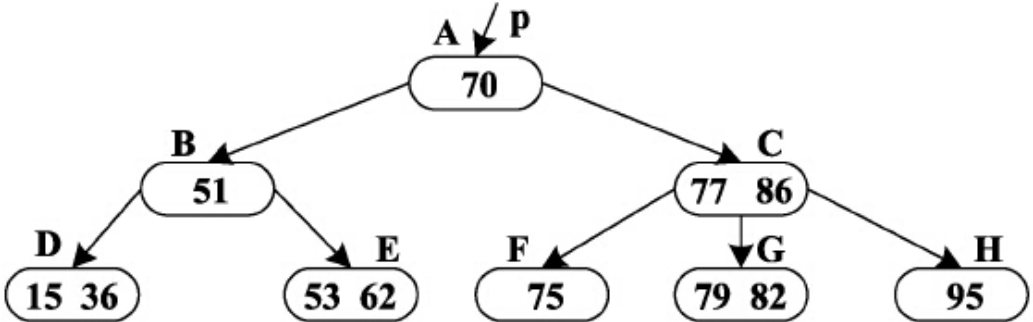


图11-21 插入关键字53的过程

②插入关键字22。从根结点出发确定关键字22应插入的位置，因 $22 < 51$ ，故22应该插入结点D中。插入后结点D的关键字个数大于2，故还应将结点D分裂，关键字22上升到双亲结点B中，关键字36进入新生成的结点

D' 中，变成结点B中 P_1 指向的结点，关键字15保留在结点D中，仍然由 P_0 指向该结点。插入关键字22的过程如图11-22所示。

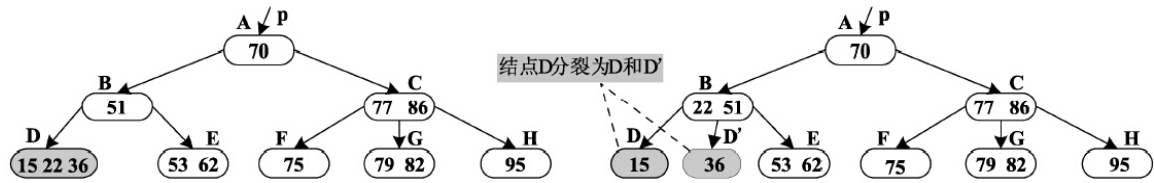


图11-22 插入关键字22的过程

③插入关键字85。首先确定关键字85应插入的位置，因 $85 > 70$ 且 $77 < 85 < 86$ ，故85应插入结点G中。插入后结点G中的关键字个数大于2，还需要将结点G分裂，关键字82上升到结点C中，关键字85被插入新结点G'中，关键字79保留在结点F中，结点C的 P_1 指向结点C， P_2 指向结点G'。插入关键字85的过程及结点G分裂过程如图11-23所示。

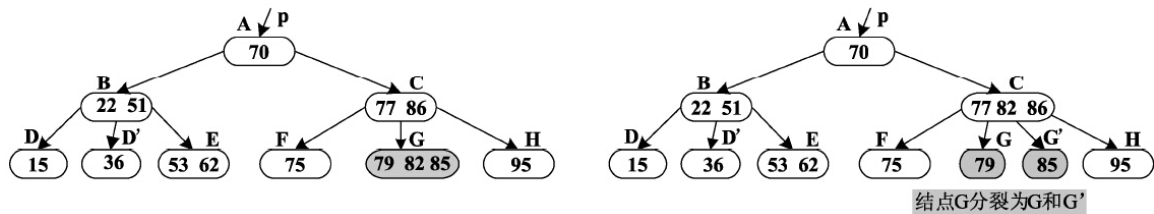


图11-23 插入关键字85及结点G的分裂过程

此时，结点C的关键字个数大于2，且C的子树大于3，因此，需要将结点C进行分裂，并对C的子树进行重新组合。仍将中间的关键字即82上升到双亲结点A中，关键字77保留在C中，关键字86被插入新结点C'中，结点A的 P_1 指针指向结点C'，结点A的 P_2 指针指向结点C。结点C的分裂过程如图11-24所示。

④插入关键字59。从根结点出发确定关键字59的插入位置，因 $51 < 59 < 70$ ，故应将关键字59插入结点E中，如图11-25所示。

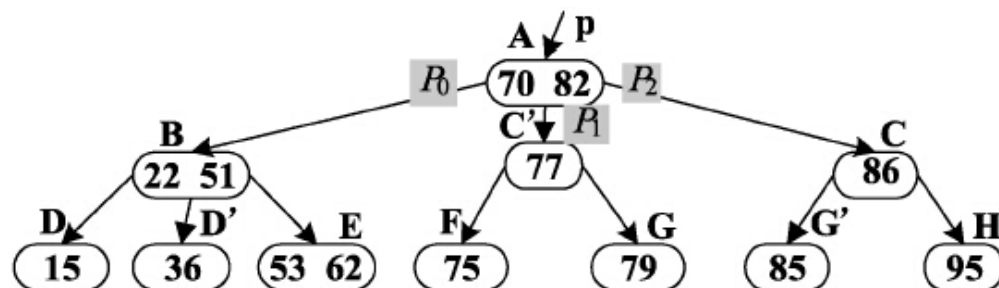


图11-24 结点C分裂为结点C'和C'的过程

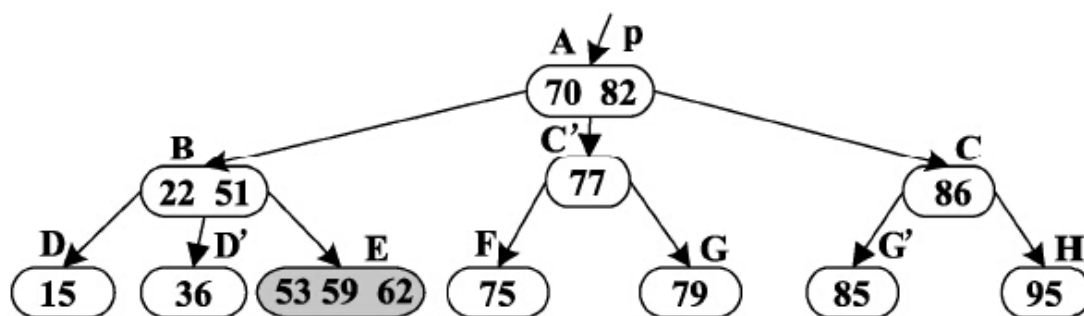


图11-25 插入关键字59后的情形

此时，结点E中的关键字个数大于2，需要将结点E分裂，把中间的关键字59提升到双亲结点B中，关键字62保留在结点E中，关键字53被插入新的结点E'中，并使结点B的 P_2 指针指向结点E'，结点B的 P_3 指针指向结点E。结点E的分裂过程如图11-26所示。

因结点B中的关键字个数大于2，还需要进一步对结点B进行分裂，其中关键字51被提升到双亲结点A中，关键字22保留在结点B中，关键字59被插入新结点B'中，结点A的 P_0 和 P_1 指针分别指向结点B和B'。结点B被分裂的过程如图11-27所示。

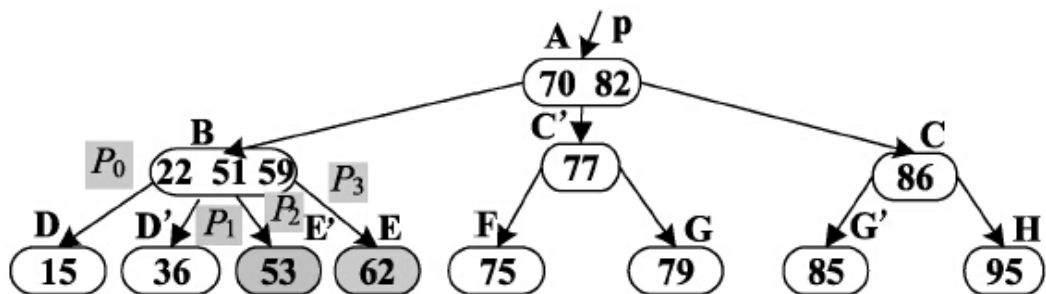


图11-26 结点E的分裂过程

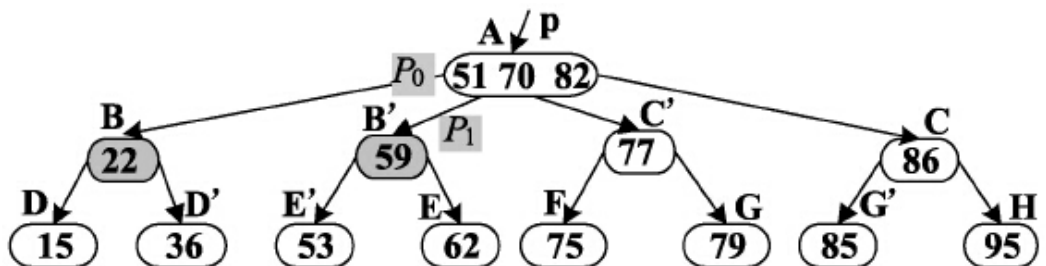


图11-27 结点B的分裂过程

关键字70提升到结点A后，结点A的关键字个数大于2，因此，还需要继续对结点A进行分裂。因结点A是根结点，故需要生成一个新结点R作为根结点。与上面的操作类似，也需要将结点A中位于中间的关键字70插入R中，关键字51保留在结点A中，关键字82被插入新结点A'中，结点R的 P_0 和 P_1 指针分别指向结点A和A'。结点A的分裂过程如图11-28所示。

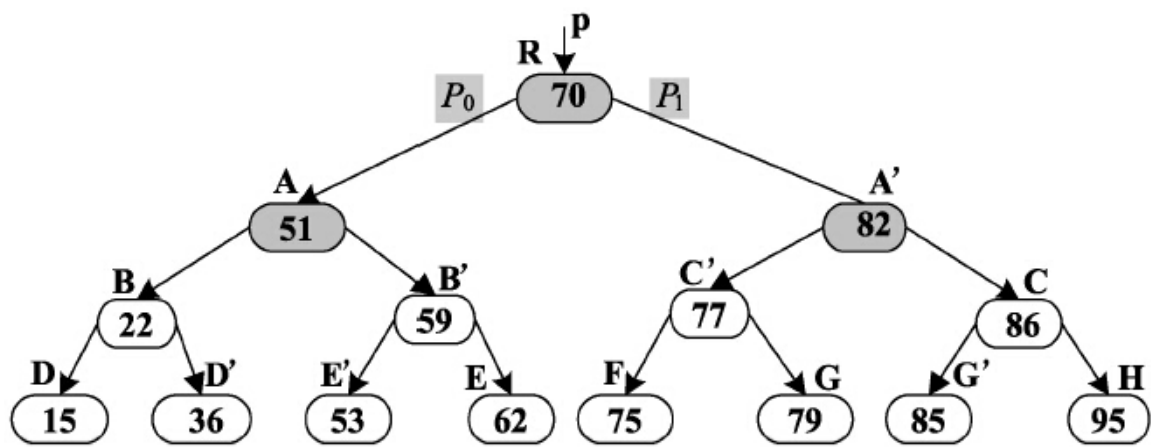


图11-28 结点A的分裂过程

在B-树中插入关键字的算法实现如下。

```

void BTreeInsert(BTree *T,DataType k,BTree p,int i)
/*
在m
阶B-
树T
上结点*p
插入关键字k
。如果结点关键字个数>m-1
, 则进行结点分裂调整*/
{
    BTree ap=NULL,newroot;
    int finished=0;
    int s,i;
    DataType rx;
    if(*T==NULL) /*
如果树*T
为空, 则生成的结点作为根结点*/
    {
        *T=(BTree)malloc(sizeof(BTNode));
        (*T)->keynum=1;
        (*T)->parent=NULL;
        (*T)->data[1]=k;
        (*T)->ptr[0]=NULL;
        (*T)->ptr[1]=NULL;
    }
    else
    {
        rx=k;
        while(p&!=finished)
        {
            Insert(&p,i,rx,ap); /*
将rx->key
和ap
分别插入p->key[i+1]
和p->ptr[i+1]
中 */
            if(p->keynum<m) /*
如果关键字个数小于m
, 则表示插入完成*/
            {
                finished=1;
            }
            else /*
分裂结点*p*/
            {
                s=(m+1)/2;
                split(&p,&ap); /*
将p->key[s+1..m],p->ptr[s..m]
和p->recptr[s+1..m]
移入新结点*ap*/
                rx=p->data[s];
                p=p->parent;
                if(p)
                    i=Search(p,rx); /*
在双亲结点*p
中查找rx->key
的插入位置*/
            }
        }
        if(!finished) /*
生成含信息(T,rx,ap)
的新的根结点*T
, 原T

```

```

和ap
为子树指针*/
        {
            newroot=(BTree)malloc(sizeof(BTNode));
            newroot->keynum=1;
            newroot->parent=NULL;
            newroot->data[1]=rx;
            newroot->ptr[0]=*T;
            newroot->ptr[1]=ap;
            *T=newroot;
        }
    }
void Insert(BTree *p,int i,DataType k,BTree ap)
/*
将r->key
和ap
分别插入p->key[i+1]
和p->ptr[i+1]
中 */
{
    int j;
    for(j=(*p)->keynum;j>i;j--) /*
空出p->data[i+1] */
    {
        (*p)->data[j+1]=(*p)->data[j];
        (*p)->ptr[j+1]=(*p)->ptr[j];
    }
    (*p)->data[i+1].key=k.key;
    (*p)->ptr[i+1]=ap;
    (*p)->keynum++;
}
void split(BTree *p,BTree *ap)
/*
将结点p
分裂成两个结点，前一半保留，后一半移入新生成的结点ap*/
{
    int i,s=(m+1)/2;
    *ap=(BTree)malloc(sizeof(BTNode)); /*
生成新结点ap*/
    (*ap)->ptr[0]=(*p)->ptr[s]; /*
后一半移入ap*/
    for(i=s+1;i<=m;i++)
    {
        (*ap)->data[i-s]=(*p)->data[i];
        if((*ap)->ptr[i-s])
            (*ap)->ptr[i-s]->parent=*ap;
    }
    (*ap)->keynum=m-s;
    (*ap)->parent=(*p)->parent;
    (*p)->keynum=s-1; /*p
的前一半保留，修改keynum*/
}

```

4. B-树的删除操作

若要在B-树中删除一个关键字，首先要利用B-树的查找算法找到关键字所在的结点，然后将该关键字从该结点删除。如果删除该关键字后该结点中的关键字个数仍然大于等于 $\lceil m/2 \rceil - 1$ ，则删除完成；否则需要对结点进行合并。

B-树的删除操作可分为如下所述3种情况。

(1) 若删除的关键字所在结点的关键字个数大于等于 $\lceil m/2 \rceil$ ，则仅需将关键字 K_i 和对应的指针 P_i 从该结点中删除即可。删除该关键字后，该结点的关键字个数仍满足大于等于 $\lceil m/2 \rceil - 1$ 。例如，图11-29显示了从结点E中删除关键字62的情形。

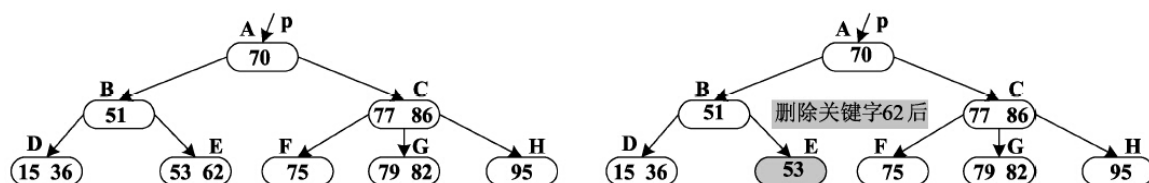


图11-29 删除关键字62的过程

(2) 被删除的关键字所在结点的关键字个数等于 $\lceil m/2 \rceil - 1$ ，而与该结点相邻的右兄弟（或左兄弟）结点中的关键字个数大于 $\lceil m/2 \rceil - 1$ ，则删除关键字后，需要将其兄弟结点中最小（或最大）的关键字上移至双亲结点中，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的结点中。例如，把图11-29所示的关键字95删除后，需要将关键字82上移至双亲结点C中，并把关键字86下移至结点H中，得到如图11-30所示的B-树。

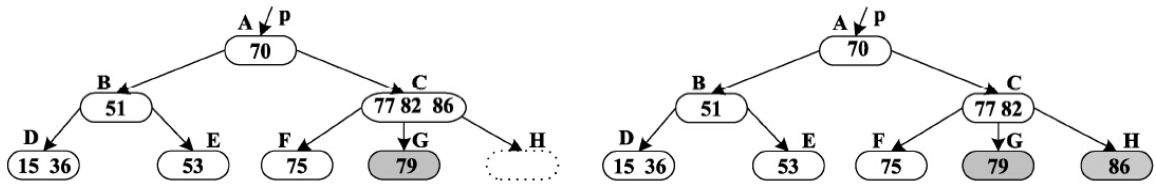


图11-30 删除关键字95的过程

(3) 被删除的关键字所在结点和其相邻的兄弟结点中的关键字个数均等于 $\lceil m/2 \rceil - 1$ ，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 P_i 所指向，则在删除关键字之后，它所在的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 P_i 所指的兄弟结点中。若没有右兄弟结点，则合并至左兄弟结点中。例如，删除关键字86后，需要把关键字86的左兄弟结点的关键字79与其双亲结点中的关键字82合并到一起，得到如图11-31所示的B-树。

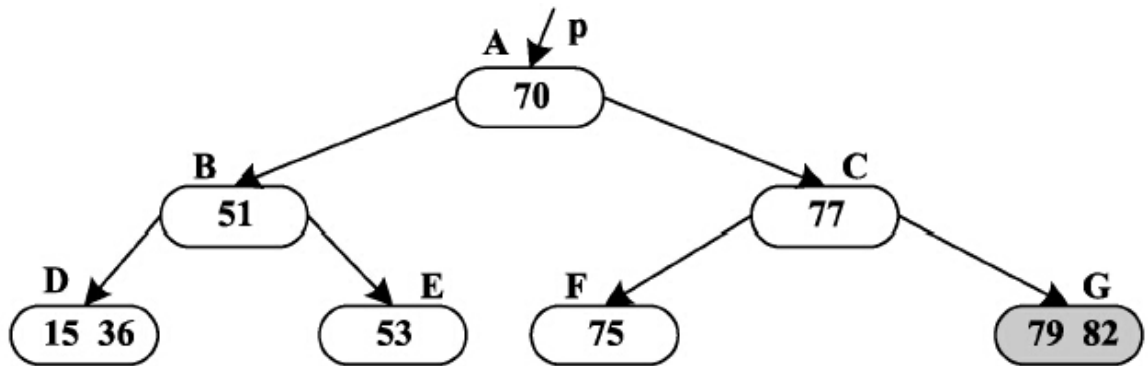


图11-31 删除关键字86的过程

11.4.2 B+树

B+树是B-树的一种变形树。一棵 m 阶的B+树和 m 阶的B-树的差异在于如下3点。

(1) 有n棵子树的结点必有n个关键字，即关键字个数与结点的子树个数相等。

(2) 所有叶子结点中包含了全部关键字的信息，及指向这些关键字记录的指针，且叶子结点本身依关键字的大小从小到大顺序链接。

(3) 所有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。

图11-32所示为一棵3阶B+树，通常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，即从最小关键字起顺序查找和从根结点开始进行随机查找。

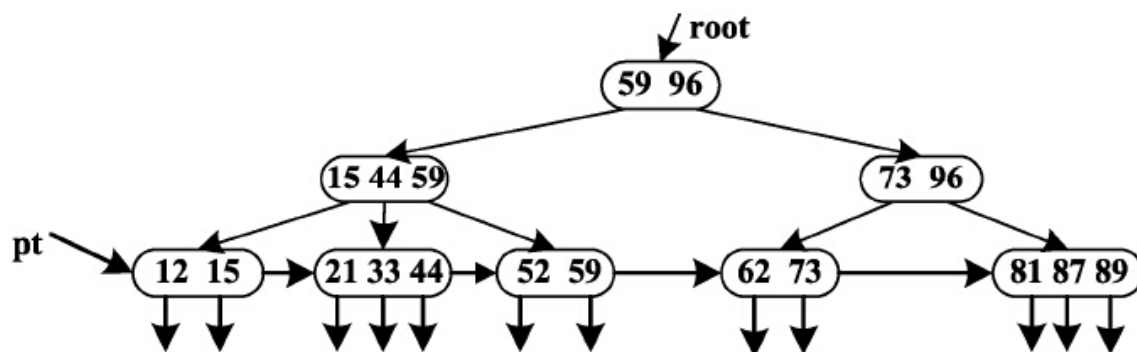


图11-32 一棵3阶的B+树

从根结点对B+树进行查找给定的关键字，需要从根结点开始经过非叶子结点到叶子结点进行查找。查找每一个结点，无论查找是否成功，都是走了一条从根结点到叶子结点的路径。在B+树上插入一个关键字和删除一个关键字都是在叶子结点中进行，在插入关键字时，要保证每个结点中的关键字个数不能大于 m ，否则需要对该结点进行分裂。在删除关键字时，要保证每个结点中的关键字个数不能小于 $\lceil m/2 \rceil$ ，否则需要与兄弟结点合并。

11.5 哈希表

在前面介绍过的有关查找的算法都经过了一系列比较过程，查找算法效率的高低取决于比较的次数。如果不经比较就能确定要查找元素的位置，那么查找效率就会大大提高，这就需建立一种数据元素的关键字与数据元素存放地址之间的对应关系，通过数据元素的关键字直接确定其存放的位置。这就是本节要介绍的哈希表。

11.5.1 什么是哈希表

如何在查找元素的过程中不与给定的关键字进行比较，就能确定所查找元素的存放位置。这就需要在元素的关键字与元素的存储位置之间建立起一种对应关系，使得元素的关键字与唯一的存储位置对应。有了这种对应关系，在查找某个元素时，只需要利用这种确定的对应关系，由给定的关键字就可以直接找到该元素。 key 表示元素的关键字， f 表示对应关系，则 $f(key)$ 表示元素的存储地址，这种对应关系 f 称为哈希函数，利用哈希函数可以建立哈希表。哈希函数也称为散列函数。

例如，一个班级有30名学生，将这些学生按各自姓氏的拼音排序，姓氏首字母相同的学生放在一起。根据学生姓名的拼音首字母建立的哈希表如表11-2所示。

表11-2 哈希表示例

序 号	姓 氏 拼 音	学 生 姓 名
1	A	安紫衣
2	B	白小翼
3	C	陈立本、陈冲
4	D	邓华
5	E	
6	F	冯峰
7	G	耿敏、弓宁
8	H	何山、郝建华
⋮	⋮	⋮

例，如在查找姓名为“冯峰”的学生时，就可以从序号为6的一行直接找到该学生。这种方法要比在一堆杂乱无章的姓名中查找要方便得多，但是，如果要查找姓名为“郝建华”的学生，拼音首字母为“H”的学生有多个，这就需要在该行中顺序查找。像这种不同的关键字key出现在同一地址上，即有 $key1 \neq key2$ ， $f(key1) = f(key2)$ 的情况称为哈希冲突。

在一般情况下，元素的关键字越多，越容易发生冲突，在设计哈希表时，应尽可能避免冲突的发生。只有少发生冲突，才能尽可能快地利用关键字找到对应的元素。因此，为了更加高效的查找集合中的某个元素，不仅需要建立一个哈希函数，还需要一个解决哈希函数冲突的方法。所谓哈希表，就是根据哈希函数和解决冲突的方法将元素的关键字映射在一个有限的且连续的地址，并将元素存储在该地址上的表中。

11.5.2 哈希函数的构造方法

构造哈希函数的目的主要是使哈希地址尽可能地均匀分布以减少或避免产生冲突、使计算方法尽可能简便以提高运算效率。哈希函数的构造方法主

要有以下几种。

1. 直接定址法

直接定址法就是直接取元素的值的线性函数值作为哈希函数的地址。直接定址法表示为 $h(key) = x * key + y$ ，其中 x 和 y 是常数。直接定址法的计算比较简单且不会发生冲突。例如，任给一组元素{115, 231, 372, 55, 567, 880, 412, 137}，如果令 $x=1$ ， $y=0$ ，则存储8个元素就需要占用825（最大的元素减去最小的元素即 $880-55$ ）个内存单元。

由于这种方法产生的哈希函数地址十分分散，造成内存的大量浪费，因此一般不采用这种构造方法。

2. 平方取中法

平方取中法就是把元素值的平方的其中几位作为哈希函数的地址。由于一个数经过平方后，每一位数字都与该数的每一位相关，因此，采用平方取中法得到的哈希地址与元素的每一位都相关，使哈希地址有了较好的分散性，从而避免冲突的发生。

例如给定关键字 $key=1234$ ，则关键字取平方后即 $key^2=1522756$ ，取中间的3位得到哈希函数的地址，即 $h(key)=227$ 。在该方法中，具体取哪几位作为哈希函数的地址根据具体情况决定。

3. 折叠法

折叠法是将元素平均分割为若干等份，最后一个部分如果不够可以空缺，然后将这几个等份叠加求和作为哈希地址。这种方法主要用在元素的位数特别多且每一个元素的位数分布大体相当的情况。例如，给定一个元素为23467523072，可以按照3位将该元素分割为几个部分，其折叠计算过程如下，然后去掉进位，将111作为元素key的哈希地址。

$$\begin{array}{r} 234 \\ 675 \\ 230 \\ + 72 \\ \hline h(key)=1111 \end{array}$$

4. 除留余数法

该方法通过对元素取余，将得到的余数作为哈希地址。设哈希表长为m，p为小于等于m的最大质数，则哈希函数为 $h(key) = key \% p$ 。除留余数法是一种最简单，也是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，也可在折叠、平方取中等运算之后取模。

例如，在一组元素{66, 235, 332, 29, 43, 58, 77, 38}中，设哈希表长m为14，取p=13，则这组元素的哈希地址存储情况如图11-33所示。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
hash地址		66	235	29	43		58	332				77	38	

图11-33 元素在哈希表的存储情况

在求解关键字的哈希地址时， p 的取值十分关键，一般情况下， p 为质数或者除去小于20的质因数的合数。

11.5.3 处理冲突的方法

在构造哈希函数的过程中，会不可避免地出现冲突情况。所谓处理冲突，就是在有冲突发生时，为产生冲突的关键字找到另一个地址存放该关键字。在解决冲突的过程中，可能会得到一系列哈希地址 h_i ($i=1, 2, \dots, n$)，也就是发生第一冲突时，经过处理后得到第一新地址记作 h_1 ，如果 h_1 仍然会冲突，则处理后得到第二个地址 h_2 ， \dots ，依次类推，直到 h_n 不产生冲突，将 h_n 作为关键字的存储地址。

处理冲突的方法比较常用的主要有开放定址法、再哈希法和链地址法。

1. 开放定址法

开放定址法是解决冲突比较常用的方法。开放定址法就是利用哈希表中的空地址存储产生冲突的关键字。当冲突发生时，按照下列公式处理冲突。

$$h_i = (h(\text{key}) + d_i) \% m, \text{ 其中 } i=1, 2, \dots, m-1$$

其中， $h(\text{key})$ 为哈希函数， m 为哈希表长， d_i 为地址增量。地址增量 d_i 可以以下3种方法获得。

(1) 线性探测再散列：在冲突发生时，地址增量 d_i 依次取1, 2, \dots , $m-1$ 自然数列，即 $d_i = 1, 2, \dots, m-1$ 。

(2) 二次探测再散列：在冲突发生时，地址增量 d_i 依次取自然数的平方，即 $d_i = 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2$ 。

(3) 伪随机数再散列：在冲突发生时，地址增量 d_i 依次取随机数序列。

例如，在长度为14的哈希表中，元素序列37，561，49，86在哈希表中的存放情况如图11-34所示。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
hash地址			561						86		49	37		

图11-34 冲突发生前的哈希表

当要插入元素232时，哈希函数 $h(232) = 149\%13 = 11$ ，而单元11已经有元素存在，产生冲突，利用线性探测再散列法解决冲突，即 $h_1 = (11+1)\%14 = 12$ ，所以把232存放在单元12中，如图11-35所示。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
hash地址			561						86		49	37	232	

图11-35 插入元素232后的哈希表

当要插入元素50时，哈希函数 $h(50) = 50\%13 = 11$ ，而单元11已经有元素存在，产生冲突，利用线性探测再散列法解决冲突，即 $h_1 = (11+1)\%14 = 12$ ，仍然冲突；继续利用线性探测法，即 $h_2 = (11+2)\%14 = 13$ ，单元13空闲，因此将50存放在单元13中，如图11-36所示。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
hash地址			561						86		49	37	232	50

图11-36 插入元素50后的哈希表

当冲突发生时，也可以使用二次探测再散列和伪随机数再散列处理冲突。

2. 再哈希法

再哈希法就是在冲突发生时，利用另外一个哈希函数再次求哈希函数的地址，直到冲突不再发生为止，即 $h_i = \text{rehash}(key)$ ， $i=1, 2, \dots, n$ ，其中，rehash表示不同的哈希函数。这种再哈希法一般不容易再次发生冲突，但是需要事先构造多个哈希函数，这是一件不太容易也不现实的事情。

3. 链地址法

链地址法就是将具有相同散列地址的关键字用一个线性链表存储起来。每个线性链表设置一个头指针指向该链表。链地址法的存储表示类似于图的邻接表表示。在每一个链表中，所有的元素都是按照关键字有序排列。链地址法的主要优点是在哈希表中增加元素和删除元素方便。

例如一组元素序列{66, 53, 123, 77, 89, 48, 274, 92, 26, 230}，可以采用哈希函数 $h(key) = key \% 13$ 构造哈希表，用链地址法处理冲突，哈希表如图11-37所示。

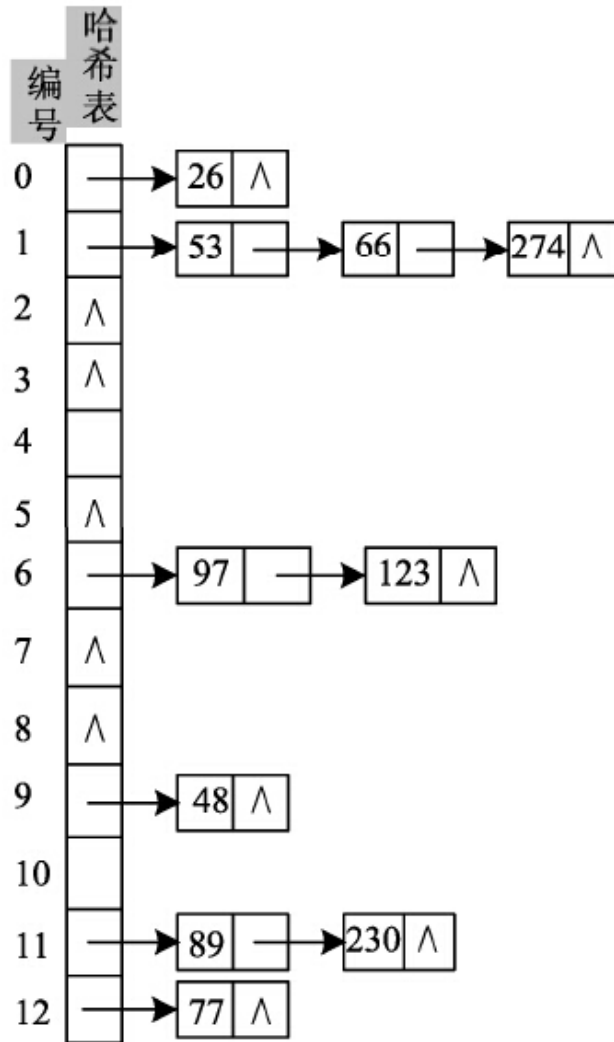


图11-37 用链地址法处理冲突的哈希表

对于图11-37采用链地址法构造的哈希表，在查找概率相等的情况下，查找成功时的平均查找长度为 $ASL_{succ} = \frac{1}{8} \times (1 \times 6 + 2 \times 3 + 3) = 1.875$ 。

11.5.4 哈希表应用举例

【例11-3】 给定一组元素的关键字 $hash[] = \{78, 90, 66, 70, 155, 82, 123, 231\}$ ，利用除留余数法和线性探测再散列法将元素存储在哈希表

中，并查找给定的关键字，求平均查找长度。

【分析】 主要考查哈希函数的构造方法、冲突解决的办法。算法实现主要包括构建哈希表、在哈希表中查找给定的关键字、输出哈希表及求平均查找长度。关键字的个数是8个，假设哈希表的长度m为11，p为11，利用除留余数法求哈希函数即 $h(key) = key \% p$ ，利用线性探测再散列法解决冲突即 $h_i = (h(key) + d_i)$ ，哈希表如图11-38所示。

	0	1	2	3	4	5	6	7	8	9	10
hash地址	66	78	90	155	70	82	123	231			
冲突次数	1	1	1	3	1	1	5	8			

图11-38 哈希表

哈希表的查找过程就是利用哈希函数和处理冲突创建哈希表的过程。例如要查找 $key=155$ ，由哈希函数 $h(155) = 155 \% 11 = 1$ ，此时与第1号单元中的关键字78比较，因为 $155 \neq 78$ ，又 $h_1 = (1+1) \% 11 = 2$ ，所以将第2号单元的关键字90与155比较，因为 $155 \neq 90$ ，又 $h_2 = (1+2) \% 11 = 3$ ，所以将第3号单元中关键字155与key比较，因为 $key=155$ ，所以查找成功，返回序号2。

尽管利用哈希函数可以直接找到对应的元素，但是仍然会不可避免地有冲突产生，在查找的过程中，比较仍会是不可避免，因此，仍然以平均查找长度衡量哈希表查找的效率高低。假设每个关键字的查找概率都是相等的，则在图11-38中的哈希表中查找某个元素成功时的平均查找长度为

$$ASL_{成功} = \frac{1}{8} \times (1 \times 5 + 3 + 5 + 8) = 2.625。$$

1. 哈希表的操作

这部分主要包括哈希表的创建、查找与求哈希表平均查找长度，实现代码如下。

```
void CreateHashTable(HashTable *H,int m,int p,int hash[],int n)
/*
构造一个空的哈希表，并处理冲突*/
{
    int i,sum,addr,di,k=1;
    (*H).data=(DataType*)malloc(m*sizeof(DataType)); /*
为哈希表分配存储空间*/
    if(!(*H).data)
        exit(-1);
    for(i=0;i<m;i++) /*
初始化哈希表*/
    {
        (*H).data[i].key=-1;
        (*H).data[i].hi=0;
    }
    for(i=0;i<n;i++) /*
求哈希函数地址并处理冲突*/
    {
        sum=0; /*
冲突的次数*/
        addr=hash[i]%p; /*
利用除留余数法求哈希函数地址*/
        di=addr;
        if(((*H).data[addr].key!=-1) /*
如果不冲突则将元素存储在表中*/
        {
            (*H).data[addr].key=hash[i];
            (*H).data[addr].hi=1;
        }
        else /*
用线性探测再散列法处理冲突*/
        {
            do
            {
                di=(di+k)%m;
                sum+=1;
            } while(((*H).data[di].key!=-1);
            (*H).data[di].key=hash[i];
            (*H).data[di].hi=sum+1;
        }
    }
    (*H).curSize=n; /*
哈希表中关键字个数为n*/
    (*H).tableSize=m; /*
哈希表的长度*/
}
int SearchHash(HashTable H,KeyType k)
/*
在哈希表H
中查找关键字k
的元素*/
{
    int d,d1,m;
    m=H.tableSize;
    d=d1=k%m; /*
求k
的哈希地址*/
```

```

        while(H.data[d].key!=-1)
        {
            if(H.data[d].key==k) /*
如果是查找的关键词k
, 则返回k
的位置*/
                return d;

            else /*
继续往后查找*/
                d=(d+1)%m;
                if(d==d1) /*
如果查找了哈希表中的所有位置, 没有找到返回0*/
                    return 0;
        }
        return 0; /*
该位置不存在关键词k*/
    }
    void HashASL(HashTable H,int m)
    /*
求哈希表的平均查找长度*/
    {
        float average=0;
        int i;
        for(i=0;i<m;i++)
            average=average+H.data[i].hi;
        average=average/H.curSize;
        printf("
平均查找长度ASL=%.2f",average);
        printf("\n");
    }

```

2. 测试部分

这部分主要包括头文件、函数声明、类型定义、主函数与哈希表的输出, 实现代码如下。

```

/*
包含头文件*/
#include<stdlib.h>
#include<stdio.h>
#include<malloc.h>
typedef int KeyType
;
typedef struct /*
元素类型定义*/
{
    KeyType key
; /*
关键词*/
    int hi
; /*
冲突次数*/
}DataType
;
typedef struct /*
哈希表类型定义*/
{
    DataType*data
;
    int tableSize

```

```

: /*
哈希表的长度*/
int curSize
: /*
表中关键字个数*/
}HashTable
:
void CreateHashTable
    (HashTable*H
    , int m
    , int p
    , int hash[]
    , int n
    );
int SearchHash
    (HashTable H
    , KeyType k
    );
void DisplayHash
    (HashTable H
    , int m
    );
void HashASL
    (HashTable H
    , int m
    );
void DisplayHash
    (HashTable H
    , int m
    )
/*
输出哈希表*/
{
    int i
    ;
    printf
    (
    "
哈希表地址: "
    );
    for
    (i=0
    ; i<m
    ; i++
    )
    printf
    ("%5d"
    , i
    );
    printf
    ("\n"
    );
    printf
    (
    "
关键字key:"
    );
    for
    (i=0
    ; i<m
    ; i++
    )
    printf
    ("%5d"
    , H.data[i].key
    );
    printf
    ("\n"
    );
    printf
    (
    "
冲突次数: "
    );
    for

```

```

    (i=0
    ; i<m
    ; i++
    )
    printf
    ("%5d"
    , H.data[i].hi
    ) ;
    printf
    ("\n"
    ) ;
}
void main
(
)
{
    int hash[]={78
    , 90
    , 66
    , 70
    , 155
    , 82
    , 123
    , 231}
    ;
    int m=11
    , p=11
    , n=8
    , pos
    ;
    KeyType k
    ;
    HashTable H
    ;
    CreateHashTable
    (&H
    , m
    , p
    , hash
    , n
    ) ;
    DisplayHash
    (H
    , m
    ) ;
    k=155
    ;
    pos=SearchHash
    (H
    , k
    ) ;
    printf
    (
    "
    关键字%d
    在哈希表中的位置为: %d\n"
    , k
    , pos
    ) ;
    HashASL
    (H
    , m
    ) ;
}

```

程序运行结果如图11-39所示。

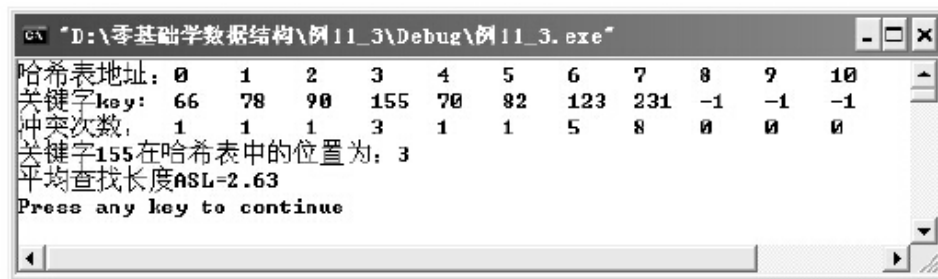


图11-39 哈希表的创建与查找程序运行结果

【考研真题】 将关键字序列（7，8，30，11，18，9，14）散列存储在散列表中，散列表的存储空间是一个下标从0开始的一维数组，散列函数为 $H(key) = (key * 3) \text{ MOD } 7$ ，处理冲突采用线性探测再散列法，要求装填因子为0.7。

- （1）请画出构造的散列表。
- （2）分别计算等概率情况下查找成功和查找不成功的平均查找长度。

【分析】 该题目是2010年的考研题目，主要考查哈希表的构造和平均查找长度的概念。

（1）根据给出的散列函数 $H(key) = (key * 3) \text{ MOD } 7$ 和处理冲突方法构造哈希表，如表11-3所示。

表11-3 哈希表

下标	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	
冲突次数	1	2		1		1	1	3	3	

(2) 查找成功的平均查找长度为 $ASL_{成功} = (4*1+2*3+1*2) / 7 = 12/7$,
查找不成功的平均查找长度为 $ASL_{不成功}$
 $= (3+2+1+2+1+5+4+3+2+1) / 10 = 24/10 = 12/5$ 。

11.6 小结

查找分为静态查找与动态查找两种。

静态查找主要有顺序表、有序顺序表和索引顺序表的查找。其中，索引顺序表的查找是为主表建立一个索引，根据索引确定元素所在的范围，这样可以有效地提高查找的效率。

动态查找有二叉排序树、平衡二叉树、B-树和B+树。静态查找中顺序表的平均查找长度为 $O(n)$ ，折半查找的平均查找长度为 $O(\log_2 n)$ 。动态查找中的二叉排序树的查找类似于折半查找，其平均查找长度为 $O(\log_2 n)$ 。

哈希表大大减少了与元素的关键字的比较次数。建立哈希表的方法主要有：直接定址法、平方取中法、折叠法和除留余数法等。

解决冲突最为常用的方法主要有两个，即开放定址法和链地址法。

11.7 习题

一、选择题

1. 已知一个有序表为 (11, 22, 33, 44, 55, 66, 77, 88, 99), 则折半查找55需要比较 () 次。

A. 1

B. 2

C. 3

D. 4

2. 设哈希表长 $m=14$, 哈希函数 $H(\text{key}) = \text{key} \text{ MOD } 11$ 。表中已有4个结点 $\text{addr}(15)=4$, $\text{addr}(38)=5$, $\text{addr}(61)=6$, $\text{addr}(84)=7$, 其余地址为空, 如用二次探测再散列处理冲突, 则关键字为49的地址为 ()。

A. 8

B. 3

C. 5

D. 9

3. 在散列查找中，平均查找长度主要与（）有关。

A. 散列表长度

B. 散列元素个数

C. 装填因子

D. 处理冲突方法

4. 采用折半查找法查找长度为 n 的线性表时，每个元素的平均查找长度为（）。

A. $O(n^2)$

B. $O(n \log_2 n)$

C. $O(n)$

D. $O(\log_2 n)$

5. 有一个有序表为 {1, 3, 9, 12, 32, 41, 45, 62, 75, 77, 82, 95, 100}，当折半查找值为82的结点时，（）次比较后查找成功。

A. 11

B. 5

C. 4

D. 8

6. 以下说法错误的是（ ）。

A. 散列法存储的思想是由关键字值决定数据的存储地址。

B. 散列表的结点中只包含数据元素自身的信息，不包含指针。

C. 负载因子是散列表的一个重要参数，它反映了散列表的饱满程度。

D. 散列表的查找效率主要取决于散列表构造时选取的散列函数和处理冲突的方法。

7. 有一个有序表为{1, 3, 9, 12, 32, 41, 45, 62, 75, 77, 82, 95, 100}，当折半查找值为82的结点时，（ ）次比较后查找成功。

A. 1

B. 4

C. 2

D. 8

8. 在各种查找方法中，平均查找承担与结点个数 n 无关的查找方法是（ ）。

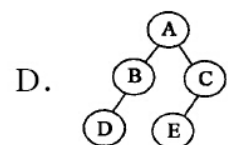
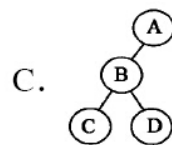
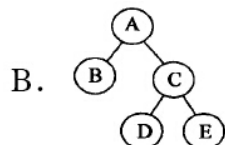
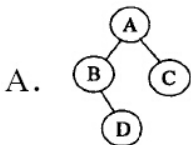
A. 顺序查找

B. 折半查找

C. 哈希查找

D. 分块查找

9. 下列二叉树中，不平衡的二叉树是（ ）。



10. 对一棵二叉排序树按（ ）遍历，可得到结点值从小到大的排列序列。

A. 先序

B. 中序

C. 后序

D. 层次

11. 对线性表进行折半查找时，要求线性表必须（ ）。

A. 以顺序方式存储

B. 以链接方式存储

C. 以顺序方式存储，且结点按关键字有序排序

D. 以链接方式存储，且结点按关键字有序排序

二、综合题

1. 设哈希表HT表长 m 为13，哈希函数为 $H(k) = k \text{ MOD } m$ ，给定的关键字值序列为{19, 14, 23, 10, 68, 20, 84, 27, 55, 11}。试求出用线性探测法解决冲突时所构造的哈希表，并求出在等概率的情况下查找成功的平均查找长度ASL。

2. 设有一组关键字{19, 1, 23, 14, 55, 20, 84, 27, 68, 11, 10, 77}，采用哈希函数 $H(\text{key}) = \text{key} \text{ MOD } 13$ ，采用开放地址法的二次探测再散列方法解决冲突，试在0—18的散列空间中对关键字序列构造哈希表，画出哈希表，并求其查找成功时的平均查找长度。

3. 已知关键字序列{11, 2, 13, 26, 5, 18, 4, 9}, 设哈希表表长为16, 哈希函数 $H(\text{key}) = \text{key} \text{ MOD } 13$, 处理冲突的方法为线性探测法, 请给出哈希表, 并计算在等概率的条件下的平均查找长度。

三、算法设计题

1. 给定一个递增有序的元素序列, 利用折半查找算法查找值为x的元素的递归算法。

2. 以图11-40所示的索引顺序表为例, 编写一个查找关键字68的算法。

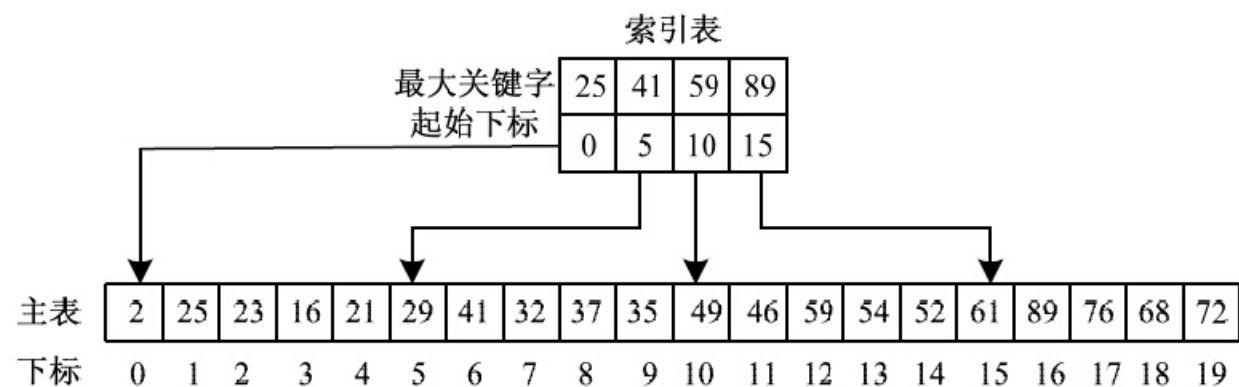


图11-40 索引顺序表

3. 利用哈希函数 $h(\text{key}) = 3 * k \% 11$, 采用链地址法处理冲突, 对关键字集合{22, 43, 53, 45, 30, 12, 2, 56}构造一个哈希表, 并求出在查找每一个元素相等概率的情况下的平均查找长度。

第12章 内排序

排序（sorting）是计算机程序设计的一个特别重要的技术，计算机的各个应用领域都有它的身影。如在处理学生考试成绩和元素的查找等都涉及对数据的排序。本章主要介绍几种常用的排序技术，包括插入排序、选择排序、交换排序、归并排序和基数排序。

本章重点和难点：

- 希尔排序
- 快速排序
- 堆排序
- 归并排序
- 基数排序

12.1 基本概念

在介绍有关排序的算法之前，先来介绍与排序相关的基本概念。
本节主要介绍排序的基本概念及相关概念。

排序： 把一个无序的元素序列按照元素的关键字递增或递减排列为有序的序列。

假设包含 n 个元素（记录）的序列 (E_1, E_2, \dots, E_n) 对应的关键字为 (k_1, k_2, \dots, k_n) ，需确定 $1, 2, \dots, n$ 的一种排列 p_1, p_2, \dots, p_n ，使关键字满足非递减（或非递增）关系 $k_{p_1} \leq k_{p_2} \leq \dots \leq k_{p_n}$ ，从而使元素构成一个非递减（或非递增）的序列 $(E_{p_1}, E_{p_2}, \dots, E_{p_n})$ ，这样的一种操作被称为**排序**。

稳定排序 和不稳定排序： 在待排序的记录序列中，若存在两个或两个以上关键字相等的记录。假设 $k_i = k_j$ （ $1 \leq i \leq n, 1 \leq j \leq n, i \neq j$ ），且排序前对应的记录 E_i 领先于 E_j （即 $i < j$ ）。在排序之后，如果元素 E_i 仍领先于 E_j ，则称这种排序采用的方法是**稳定**的。如果经过排序之后元素 E_j 领先于 E_i （即 $i < j$ ），则称这种排序方法是**不稳定**的。

一个排序算法的好坏主要可以通过时间复杂度、空间复杂度和稳定性来衡量。无论算法稳定还是不稳定，都不会影响到排序的最终结

果。

内排序 和 **外排序**：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，可将排序方法分为内部排序和外部排序两类。内部排序也称为内排序，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；外部排序也称为外排序，指的是待排序记录的数据量很大，以致内存一次不能容纳全部记录，在排序的过程中需要不断对外存进行访问的排序过程。

内排序的方法有许多，按照排序过程中采用的策略将排序分为插入排序、选择排序、交换排序和归并排序。

在排序过程中，需要以下两种基本操作。

- (1) 比较两个元素相应关键字的大小。
- (2) 将元素从一个位置移动到另一个位置。

第(1)种操作对大多数排序方法来说都是必要的，第(2)种操作可通过改变记录的存储方式可以避免。

待排序的记录序列可有下列3种存储方式。

(1) 顺序存储。待排序的元素存储在一组连续的存储单元中，这类似于线性表的顺序存储，在序列中相邻的两个记录 E_i 和 E_j ，它

们的物理位置也相邻。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，则实现排序必须移动记录。

(2) 链式存储。待排序元素存储在一组不连续的存储单元中，这类似于线性表的链式存储，序列中相邻的两个记录 E_i 和 E_j ，其物理位置不一定相邻。在这种存储方式中，记录之间的关系由附设的指针指示，在进行排序时，不需要移动元素，只需要修改指针即可。

(3) 静态链表。带排序记录存放在静态链表中，记录之间的关系由被称为游标的指针指示，实现排序不需要移动元素，只需要修改游标即可。

为了算法实现方便，本章的排序算法主要采用顺序存储，相应的元素类型描述如下。

```
#define MaxSize 100
typedef int KeyType;
typedef struct /*
数据元素类型定义*/
{
    KeyType key; /*
关键字*/
}DataType;
typedef struct /*
顺序表类型定义*/
{
    DataType data[MaxSize];
    int length;
}SqList;
```

12.2 插入排序

插入排序的算法思想：将待排序元素分为已排序子集和未排序子集，依次从未排序子集中的一个元素插入已排序子集中，使已排序子集仍然有序；重复执行以上过程，直到所有元素都有序为止。

插入排序大致可以分为直接插入排序、折半插入排序和希尔排序。

12.2.1 直接插入排序

直接插入排序（straight insertion sort）是一种最简单的插入排序算法。基本算法思想：假设待排序元素有 n 个，初始时，已排序子集只有一个元素，即第1个元素；未排序子集是剩下的 $n-1$ 个元素。

例如有4个待排序元素22、6、17和8，排序前的状态如图12-1所示。

初始时： {22} {6 17 8}
有序集 无序集

图12-1 初始状态

第1趟排序：将无序集中的第一个元素，也就是6与有序集中的元素22进行比较，因为 $22 > 6$ ，所以需要先将22向后移动一个位置，然后将6插入第一个位置，如图12-2所示。其中，阴影部分表示无序集，白色部分表示有序集。

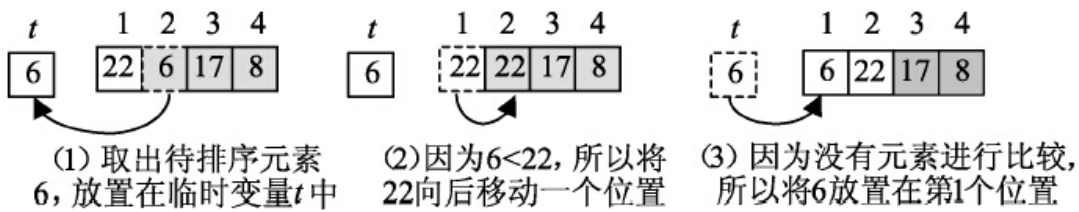


图12-2 第1趟排序过程

第2趟排序：将无序集的元素17从右到左依次与有序集中的元素比较，即先与22比较，因为 $17 < 22$ ，所以先将22向后移动一个位置，然后比较17与第1个元素6的大小，因为 $17 > 6$ ，所以将17放在第2个元素的位置，如图12-3所示。

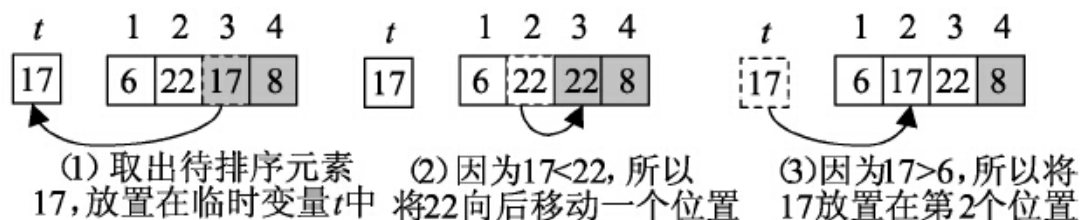


图12-3 第2趟排序过程

第3趟排序：将待排序集合中的元素8与已经有序的元素集合从右到左依次比较，先与22比较。因为 $8 < 22$ ，所以需要将22向后移动一个位置并与前一个元素17比较。由于 $8 < 17$ ，将17向后移动并继续与6进行比较。因为 $8 > 6$ ，所以将8放在第2个位置，如图12-4所示。

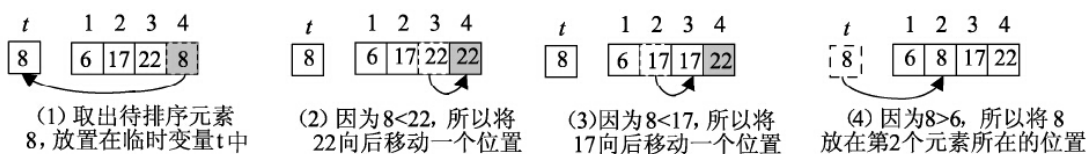


图12-4 第3趟排序过程

经过以上排序之后，已经有序的集合有4个元素，无序集合为空集。此时直接插入排序完毕，整个序列为一个有序序列。

假设待排序元素有8个，分别是17、46、32、87、58、9、50、38。使用直接插入排序对该元素序列的排序过程如图12-5所示。

排序前: {17} {46 32 87 58 9 50 38}
 第1趟排序后: {17 46} { 32 87 58 9 50 38}
 第2趟排序后: {17 32 46} { 87 58 9 50 38}
 第3趟排序后: {17 32 46 87} { 58 9 50 38}
 第4趟排序后: {17 32 46 58 87} { 9 50 38}
 第5趟排序后: {9 17 32 46 58 87} { 50 38}
 第6趟排序后: {9 17 32 46 50 58 87} { 38}
 第7趟排序后: {9 17 32 38 46 50 58 87} {}
 最终排序结果: 9 17 32 38 46 50 58 87

图12-5 直接插入排序过程

在图12-5中，所有元素被花括号分为两个集合，前一部分为有序集合，后一部分为无序集合。直接插入排序就是将无序集中的元素依次插入有序集中的对应位置，直到无序集为空为止。

相应地，直接插入排序算法描述如下。

```

void InsertSort(SqList *L)
/*
直接插入排序*/
{
    int i,j;
    DataType t;
    for(i=1;i<L->length;i++)          /*
前i
个元素已经有序，从第i+1
个元素开始与前i
个有序的关键字比较*/
    {
        t=L->data[i+1];                /*
取出第i+1
个元素，即待排序的元素*/
        j=i;
        while(j>0&& t.key<L->data[j].key) /*
寻找当前元素的合适位置*/
        {
            L->data[j+1]=L->data[j];
            j--;
        }
        L->data[j+1]=t;                /*
将当前元素插入合适的位置*/
    }
}
  
```

从上面的算法可以看出，直接插入排序算法简单且容易实现。直接插入排序算法的时间复杂度在最好的情况下是所有的元素的关键字都已经有序，此时外层的for循环的循环次数是n-1，而内层的while循环的语句执行次数为0，因此直接插入排序算法在最好的情况下的时间复杂度为O(n)。在最坏的情况下，即所有元素的关键字都是按照逆序排列，则内层while循环的比较次数均为i+1，则整个比较次数为 $\sum_{i=1}^{n-1} (i+1) = \frac{(n+2)(n-1)}{2}$ ，移动次数为 $\sum_{i=1}^{n-1} (i+2) = \frac{(n+4)(n-1)}{2}$ ，即在最坏情况下时间复杂度为O(n²)。如果元素的关键字是随机排列的，其比较次数和移动次数约为n²/4，此时直接插入排序的时间复杂度为O(n²)。

直接插入排序算法只利用了一个临时变量，因此其空间复杂度为O(1)。

12.2.2 折半插入排序

折半插入排序 算法是直接插入排序的改进。它的主要改进在于在已经有序的集合中使用折半查找法确定待排序元素的插入位置，找到要插入的位置后，将待排序元素插入相应的位置。

假设待排序元素有7个，分别为67、53、73、21、34、98、12。使用折半插入排序对该元素序列第一趟排序的过程如图12-6所示。

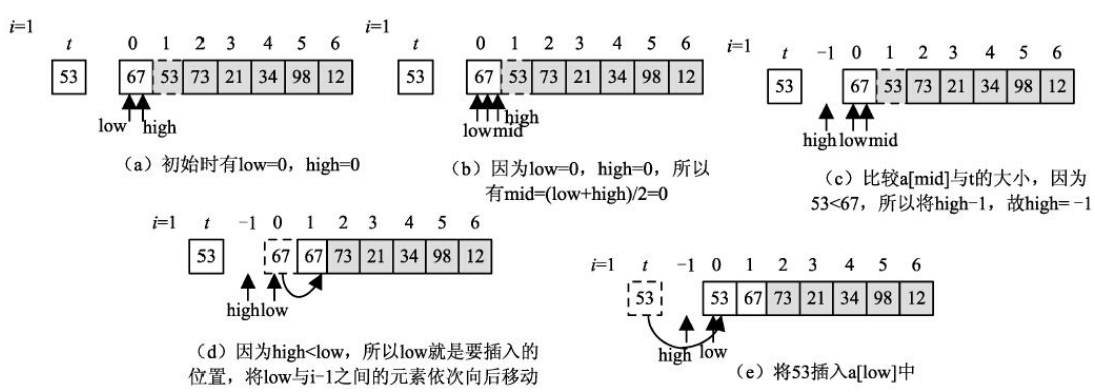


图12-6 折半插入排序第1趟排序过程

其中， $i=1$ 表示第一趟排序，待排序元素为 $a[1]$ ， t 中存放待排序元素。当 $low>high$ 时， low 指向的位置为要插入元素的位置。依次将 $low\sim i-1$ 之间的元素依次向后移动一个位置，然后将 t 的值插入即可。

第2趟折半插入排序过程如图12-7所示。

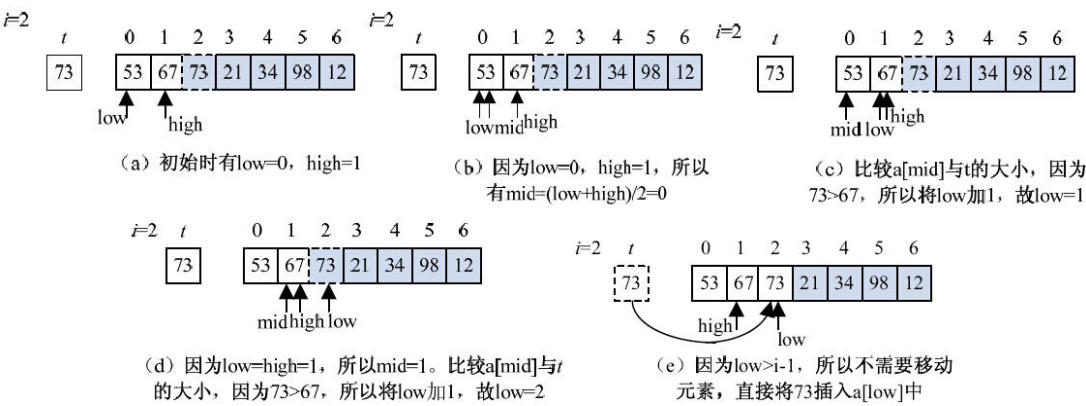


图12-7 第2趟折半插入排序过程

从以上两趟排序过程可以看出，折半插入排序与直接插入排序的区别仅仅在于查找插入的位置的方法不同。一般情况下，折半查找的效率要高于顺序查找的效率，因此折半插入排序算法可以减少比较的次数。

通过对直接插入排序算法简单修改，得到折半插入排序算法，实现代码如下。

```
void BinInsertSort(SqList *L)
/*
折半插入排序*/
{
    int i,j,mid,low,high;
    DataType t;
    for(i=1;i<L->length;i++) /*
前i
个元素已经有序，从第i+1
个元素开始与前i
个的有序的关键字比较*/
    {
        t=L->data[i+1];
        ...
    }
```

```

取出第i+1
个元素，即待排序的元素*/
        low=1, high=i;
        while (low<=high)                /*
利用折半查找思想寻找当前元素的合适位置*/
        {
            mid=(low+high)/2;
            if (L->data[mid].key>t.key)
                high=mid-1;
            else
                low=mid+1;
        }
        for (j=i; j>=low; j--) /*
移动元素，空出要插入的位置*/
            L->data[j+1]=L->data[j];
        L->data[low]=t;                /*
将当前元素插入合适的位置*/
    }
}

```

从时间上比较，折半插入排序仅减少了关键字的比较次数，而记录的移动次数不变，因此，折半插入排序的时间复杂度为 $O(n^2)$ 。

12.2.3 希尔排序

希尔排序（shell's sort）也称为缩小增量排序（diminishing increment sort），它也属于插入排序类的算法，但时间效率比前几种排序有较大改进。

从对直接插入排序的分析可知，其算法时间复杂度为 $O(n^2)$ ，但是若待排序记录序列为“正序”，其时间复杂度为 $O(n)$ 。由此可设想，若待排序记录序列按关键字基本有序，直接插入排序的效率就可大大提高。从另一个方面来看，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。希尔排序正是综合考虑这两点对直接插入排序进行改进得到的一种插入排序方法。

希尔排序算法的基本思想是先将整个待排序记录分割成若干子序列，利用直接插入排序对子序列进行排序，待整个序列中的记录基本有序时，再对全部记录进行一次直接插入排序。

假设待排序的元素有 n 个，对应的关键字分别是 a_1, a_2, \dots, a_n ，设距离（增量）为 $c_1 = 4$ 的元素为同一个子序列，则元素的关键字 $a_1, a_5, \dots, a_i, a_{i+5}, \dots, a_{n-5}$ 为一个子序列，同理，关键字 $a_2, a_6, \dots, a_{i+1}, a_{i+6}, \dots, a_{n-4}$ 为一个子序列。然后分别对同一个子序列的关键字利用直接插入排序进行排序。之后，缩小增量令 $c_2 = 2$ ，分别对同一个子序列的关键字进行插入排序。依次类推，最后令增量为1，这时只有一个子序列，对整个元素进行排序，完成希尔排序的具体过程。

设待排序元素为48、26、66、57、32、85、55、19，使用希尔排序算法对该元素序列的排序过程如图12-8所示。

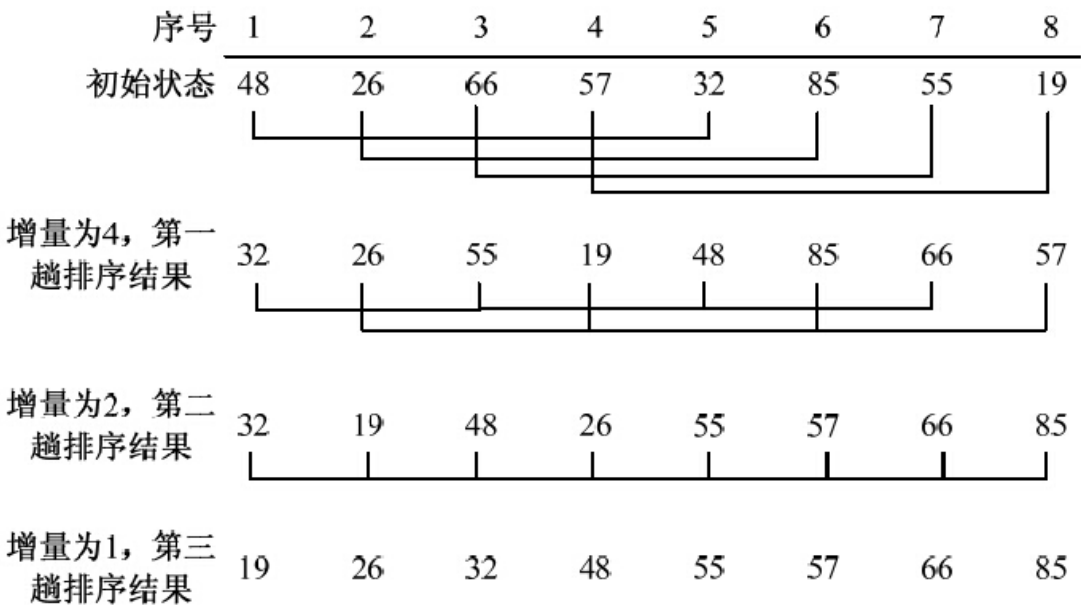


图12-8 希尔排序过程

增量依次为4、2、1，当增量为4时，第1个元素与第5个元素为一组，第2个元素与第6个元素为一组，第3个元素与第7个元素为一组，第4个元素与第8个元素为一组，本组内的元素进行直接插入排序，即完成第1趟希尔排序。当增量为2时，第1、3、5、7个元素构成一组，第2、4、6、8个元素构成一组，各组中的元

素进行直接插入排序，即完成第2趟直接插入排序。当增量为1时，将所有的元素进行直接插入排序，此时所有的元素都按照从小到大排列，希尔排序算法结束。

相应地，希尔排序的算法可描述如下。

```
void ShellInsert(SqList *L,int c)
/*
对顺序表L
进行一趟希尔排序，c
是增量*/
{
    int i,j;
    DataType t;
    for(i=c+1;i<=L->length;i++) /*
将距离为c
的元素作为一个子序列进行排序*/
    {
        if(L->data[i].key<L->data[i-c].key) /*
如果后者小于前者，则需要移动元素*/
        {
            t=L->data[i];
            for(j=i-c;j>0&&t.key<L->data[j].key;j=j-c)
                L->data[j+c]=L->data[j];
            L->data[j+c]=t; /*
依次将元素插入正确的位置*/
        }
    }
}

void ShellInsertSort(SqList *L,int delta[],int m)
/*
希尔排序，每次调用算法ShellInsert,delta
是存放增量的数组*/
{
    int i;
    for(i=0;i<m;i++) /*
进行m
趟希尔插入排序*/
    {
        ShellInsert(L,delta[i]);
    }
}
```

希尔排序的分析是一个非常复杂的事情，问题主要在于希尔排序选择的增量，但是经过大量的研究，当增量的序列为 $2^{m-k+1}-1$ 时，其中 m 为排序的次数， $1 \leq k \leq t$ ，其时间复杂度为 $O(n^{3/2})$ 。希尔排序的空间复杂度为 $O(1)$ 。因为希尔排序按照增量对每个子序列进行排序，有可能两个相等的关键字分别处于不同的序列中，造成排序过程中两者顺序颠倒，所以希尔排序算法是一种不稳定的排序算法。

12.2.4 插入排序应用举例

【例12-1】 任意给定一组元素序列，利用直接插入排序、折半插入排序和希尔排序对该序列进行排序。

【分析】 主要考查直接插入排序、折半插入排序和希尔排序的算法思想。

程序实现代码如下。

```
/*
头文件*/
#include<stdio.h>
#include<stdlib.h>
#define MaxSize 100
typedef int KeyType;
typedef struct /*
数据元素类型定义*/
{
    KeyType key; /*
关键字*/
}DataType;
typedef struct /*
顺序表类型定义*/
{
    DataType data[MaxSize];
    int length;
}SqList;
void InitSeqList(SqList *L,DataType a[],int n);
void InsertSort(SqList *L);
void ShellInsert(SqList *L,int c);
void ShellInsertSort(SqList *L,int delta[],int m);
void BinInsertSort(SqList *L);
void DispList(SqList L,int n);
void main()
{
    DataType a[]={78,29,45,10,80,21,55,3,60,32};
    int delta[]={5,3,1};
    int n=10,m=3;
    SqList L;
    /*
直接插入排序*/
    InitSeqList(&L,a,n);
    printf("[
排序前]
");
    DispList(L,n);
    InsertSort(&L);
    printf("[
直接插入排序结果]");
    DispList(L,n);
    /*
折半插入排序*/
    InitSeqList(&L,a,n);
    printf("[
排序前]
");
    DispList(L,n);
    BinInsertSort(&L);
    printf("[
折半插入排序结果]");
    DispList(L,n);
    /*
希尔排序*/
```

```

        InitSeqList(&L,a,n);
        printf("[
排序前]      ");
        DispList(L,n);
        ShellInsertSort(&L,delta,m);
        printf("[
希尔排序结果] ");
        DispList(L,n);
    }
void InitSeqList(SqList *L,DataType a[],int n)
/*
顺序表的初始化*/
{
    int i;
    for(i=1;i<=n;i++)
    {
        L->data[i]=a[i-1];
    }
    L->length=n;
}
void DispList(SqList L,int n)
/*
顺序表的输出*/
{
    int i;
    for(i=1;i<=n;i++)
        printf("%4d",L.data[i].key);
    printf("\n");
}

```

程序运行结果如图12-9所示。

```

D:\零基础学数据结构\例12_1\Debug\例12_1.exe
[排序前]      78  29  45  10  80  21  55  3  60  32
[直接插入排序结果]  3  10  21  29  32  45  55  60  78  80
[排序前]      78  29  45  10  80  21  55  3  60  32
[折半插入排序结果]  3  10  21  29  32  45  55  60  78  80
[排序前]      78  29  45  10  80  21  55  3  60  32
[希尔排序结果]  3  10  21  29  32  45  55  60  78  80
Press any key to continue

```

图12-9 各种插入排序程序运行结果

【例12-2】 编写一个插入排序算法，要求用链表实现。

【分析】 主要考查插入排序的算法思想和链表的操作。

算法思想：首先创建一个链表，将待排序元素依次插入链表中。将待排序链表分为两个部分，即有序序列和待排序序列。初始时，有序序列中没有元素，令 $L \rightarrow next = NULL$ 。指针 p 指向待排序的链表，若有序序列为空，将 p 指向的第一个结

点插入空链表L中。然后将有序链表即L指向的链表的每一个结点与p指向的结点比较，并将结点*p插入L指向的链表的恰当位置。重复执行上述操作，直到待排序链表为空。此时，L就是一个有序链表。

插入排序程序的实现如下。

```
/*
头文件*/
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
typedef int DataType;          /*
元素类型定义为整型*/
typedef struct Node             /*
单链表类型定义*/
{
    DataType data;
    struct Node *next;
}ListNode,*LinkedList;
#include"LinkedList.h"
void InsertSort(LinkedList L);
void CreateList(LinkedList L,DataType a[],int n);
void CreateList(LinkedList L,DataType a[],int n)
/*
创建单链表*/
{
    int i;
    for(i=1;i<=n;i++)
        InsertList(L,i,a[i-1]);
}
void main()
{
    LinkedList L,p;
    int n=8;
    DataType a[]={76,55,10,21,65,90,5,38};
    InitList(&L);
    CreateList(L,a,n);
    printf("
排序前的元素序列: \n");
    for(p=L->next;p!=NULL;p=p->next)
        printf("%4d ",p->data);
    printf("\n");
    InsertSort(L);
    printf("
排序后的元素序列: \n");
    for(p=L->next;p!=NULL;p=p->next)
        printf("%4d ",p->data);
    printf("\n");
}
void InsertSort(LinkedList L)
/*
链式存储结构下的插入排序*/
{
    ListNode *p=L->next,*pre,*q;
    L->next=NULL;
    初始时，已排序链表为空*/
    while(p!=NULL)
        是指向待排序的结点*/
        {
            if (L->next==NULL)
                如果*p
                是第一个结点，则插入L
                ，并令已排序的最后一个结点的指针域为空*/
                {
                    L->next=p;
```

```

        p=p->next;
        L->next->next=NULL;
    }
    else
        /*p
指向待排序的结点，在L
指向的已经排好序的链表中查找插入位置*/
    {
        pre=L;
        q=L->next;
        while (q!=NULL&&q->data<p->data) /*
在q
指向的有序表中寻找插入位置*/
        {
            pre=q;
            q=q->next;
        }
        q=p->next;
        /*q
指向p
的下一个结点，保存待排序的指针位置*/
        p->next=pre->next;
        /*
将结点*p
插入结点*pre
的后面*/
        pre->next=p;
        p=q;
        /*p
指向下一个待排序的结点*/
    }
}
}

```

程序运行结果如图12-10所示。

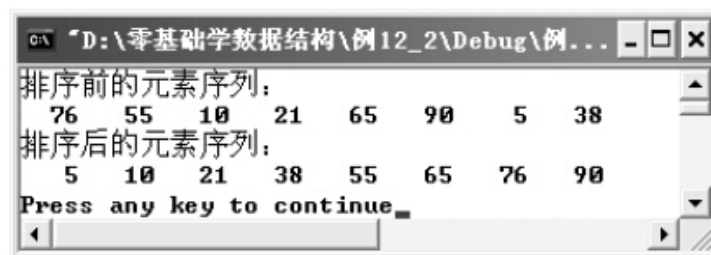


图12-10 采用链式存储结构的插入排序程序运行结果

12.3 交换排序

交换排序的基本思想是通过依次交换逆序的元素实现排序。本节主要介绍两种交换排序，即冒泡排序和快速排序。

12.3.1 冒泡排序

冒泡排序（bubble sort）是一种简单的交换类排序算法，它是通过交换相邻的两个数据元素，逐步将待排序序列变成有序序列。它的基本算法思想如下。

假设待排序元素有 n 个，从第1个元素开始，依次交换相邻的两个逆序元素，直到最后一个元素为止。当第1趟排序结束，就会将最大的元素移动到序列的末尾。然后按照以上方法进行第2趟排序，次大的元素将会被移动到序列的倒数第2个位置。依次类推，经过 $n-1$ 趟排序后，整个元素序列就成了一个有序的序列。每趟排序过程中，值小的元素向前移动，值大的元素向后移动，就像气泡一样向上升，因此将这种排序方法称为冒泡排序。

例如有5个待排序元素55、26、48、63和37。

第1趟排序：从第1个元素开始，将第1个元素与第2个元素进行比较，因为 $55 > 26$ ，所以需要交换55与26；然后比较第2个元素与第3个元

素，因为 $55 > 48$ ，所以交换55与48；接着比较第3个与第4个元素，即55与63，因为 $55 < 63$ ，所以不需要交换；最后比较第4个元素与第5个元素，因为 $63 > 37$ ，所以交换63与37。此时，完成第1趟排序，最大的元素63被移动到序列的最末端。第1趟排序过程如图12-11所示。

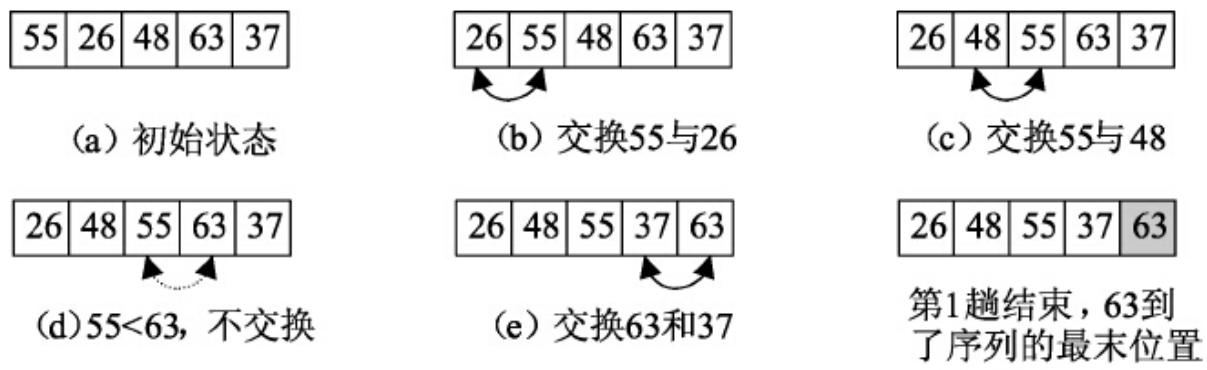


图12-11 第1趟排序过程

第2趟排序：从第1个元素开始，依次比较第1个元素与第2个元素、第2个元素与第3个元素、第3个元素与第4个元素，如果前者大于后者，则交换之；否则不进行交换。第2趟排序过程如图12-12所示。

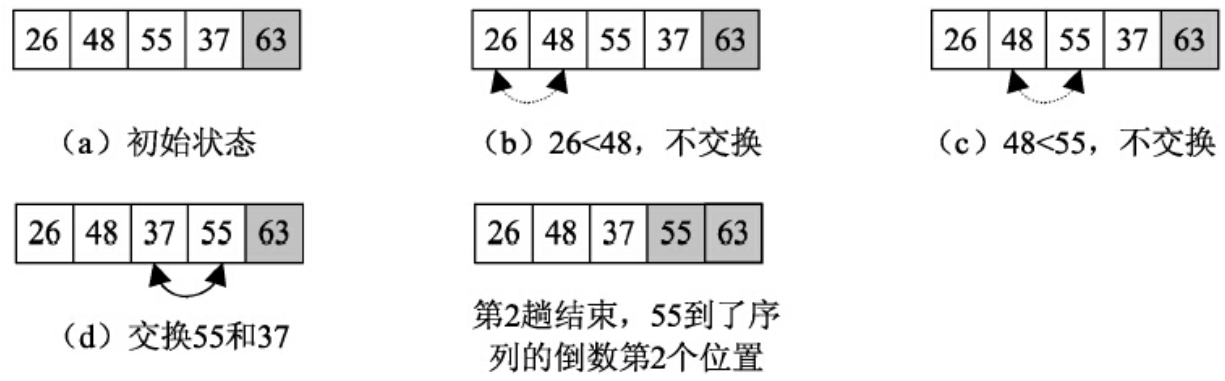
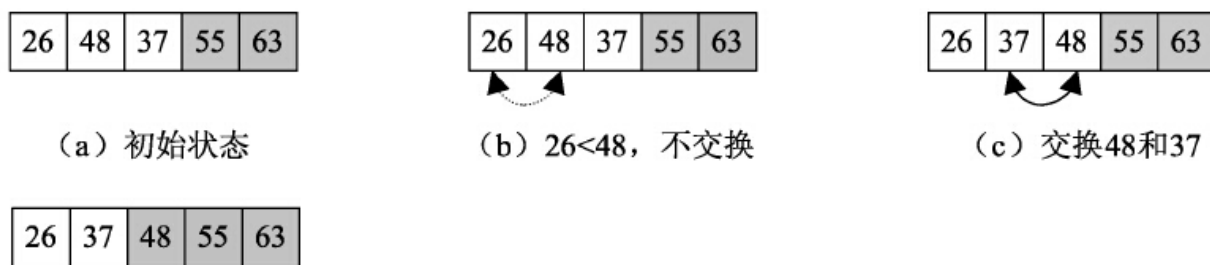


图12-12 第2趟排序过程

第3趟排序：按照上述方法，依次比较两个相邻元素，交换逆序的元素。第3趟排序过程如图12-13所示。



第3趟结束，48到了序列的倒数第3个位置

图12-13 第3趟排序过程

第4趟排序：此时，待排序元素只剩下26和37，只需要进行一次比较即可。因为 $26 < 37$ ，所以不需要交换。第4趟排序过程如图12-14所示。

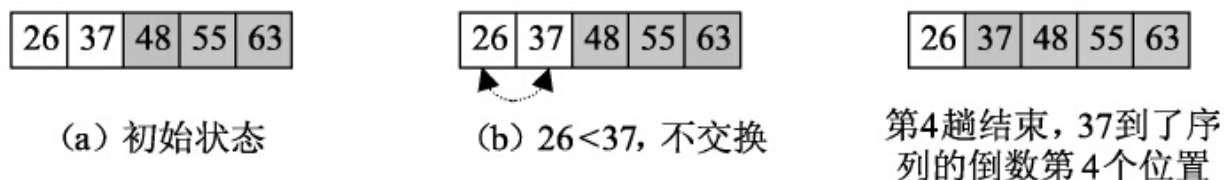


图12-14 第4趟排序过程

经过以上4趟冒泡排序后，待排序元素只剩下最后1个，因为其余元素都已经处于正确的位置，所以剩下的1个元素也位于正确的位置上。要将5个元素按照冒泡排序从小到大排列，需要进行4趟排序过程。

设待排序元素为56、72、44、31、99、21、69、80，使用冒泡排序对该元素序列排序的过程如图12-15所示。

序号	1	2	3	4	5	6	7	8
初始状态	56	72	44	31	99	21	69	80
第1趟排序结果:	56	44	31	72	21	69	80	99
第2趟排序结果:	44	31	56	21	69	72	80	99
第3趟排序结果:	31	44	21	56	69	72	80	99
第4趟排序结果:	31	21	44	56	69	72	80	99
第5趟排序结果:	21	31	44	56	69	72	80	99
第6趟排序结果:	21	31	44	56	69	72	80	99
第7趟排序结果:	21	31	44	56	69	72	80	99

图12-15 冒泡排序全过程

在冒泡排序中，如果待排序元素的个数为n，则需要n-1趟排序；对于第i趟排序，需要比较的次数为i-1。

从图12-15中看出，在第5趟排序结束后，该元素其实已经有序，第6趟和第7趟排序就不需要进行比较了。因此，在设计算法时，可以设置

一个标志flag，如果在某一趟循环中，所有元素应经有序，则令flag=0，表明该序列已经有序，停止后面的比较操作。

冒泡排序的算法描述如下。

```
void BubbleSort(SqList *L,int n)
/*
冒泡排序*/
{
    int i,j,flag;
    DataType t;
    for(i=1;i<=n-1&&flag;i++)                /*
需要进行n-1
趟排序*/
    {
        flag=0;
        for(j=1;j<=n-i;j++)                /*
每一趟排序需要比较n-i
次*/
        {
            if(L->data[j].key>L->data[j+1].key)
            {
                t=L->data[j];
                L->data[j]=L->data[j+1];
                L->data[j+1]=t;
                flag=1;
            }
        }
    }
}
```

容易看出，若初始序列为正序，则只需要进行一趟排序，在排序过程中进行n-1次关键字的比较，且不需要移动记录；反之，若初始序列为逆序，则需要进行n-1趟排序，需进行 $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ 次比较，并进行等数量级的移动操作。因此，总的时间复杂度为 $O(n^2)$ 。冒泡排序是一种稳定的排序算法。

12.3.2 快速排序

快速排序 (quick sort) 算法是冒泡排序的一种改进，与冒泡排序类似，快速排序也是通过逐渐消除待排序元素序列中逆序元素来实现排序的；不同的是，快速排序一趟排序仅需要交换一次元素就消除了多个逆序元素，这些逆序元素可能是不相邻的。

快速排序的算法思想是从待排序记录序列中选取一个记录（通常是第一个记录）作为枢轴，其关键字设为key，然后将其余关键字小于key的记录移至前面，而将关键字大于key的记录移至后面，结果将待排序记录序列分为两个子表，最后将关键字key的记录插入其分界线的位置。这个过程称为一趟快速排序。通过这一趟划分后，就可以关键字为key的记录为界将待排序序列分为两个子表，前面的子表所有记录的关键字均不大于key，后面子表的所有记录的关键字均不小于key。继续对分割后的子表进行上述划分，直至所有子表的表长不超过1为止，此时待排序的记录就成了一个有序序列。

设待排序序列存放在数组a[n]中，n为元素个数，设置两个指针i和j，初值分别为1和n，令a[1]作为枢轴元素赋给pivot，a[1]相当于空单元，然后执行以下操作。

①j从右往左扫描，若 $a[j].key < pivot.key$ ，将a[j]移至a[i]中，此时a[j]相当于空单元，并执行一次i++操作。

②i从左至右扫描，直至 $a[i].key > pivot.key$ ，将a[i]移至a[j]中，并执行一次j--操作。

③重复执行①和②，直到出现 $i \geq j$ ，则将元素pivot移动到a[i]中。此时整个元素序列在位置i被划分成两个部分，前一部分的元素关键字都小于a[i].key，后一部分元素的关键字都大于等于a[i].key。至此，即完成了一趟快速排序。

按照以上方法对a[i]左边的子表和a[i]右边的子表也可继续进行以上划分操作。例如一组待排序元素序列为55，22，44，67，35，77，18，69，依据快速排序的算法思想，得到第1趟排序的过程如图12-16所示。

经过第1趟快速排序后，元素序列以55为中心将序列划分为两个部分，左边的元素都小于55，右边的部分都大于55。快速排序正是将每个部分都以枢轴元素为中心不断地划分元素序列，直到每个序列中的元素只有一个，不能继续划分为止。

设待排序元素序列为55、22、44、67、35、77、18、69，用快速排序算法对该元素序列的排序过程如图12-17所示。

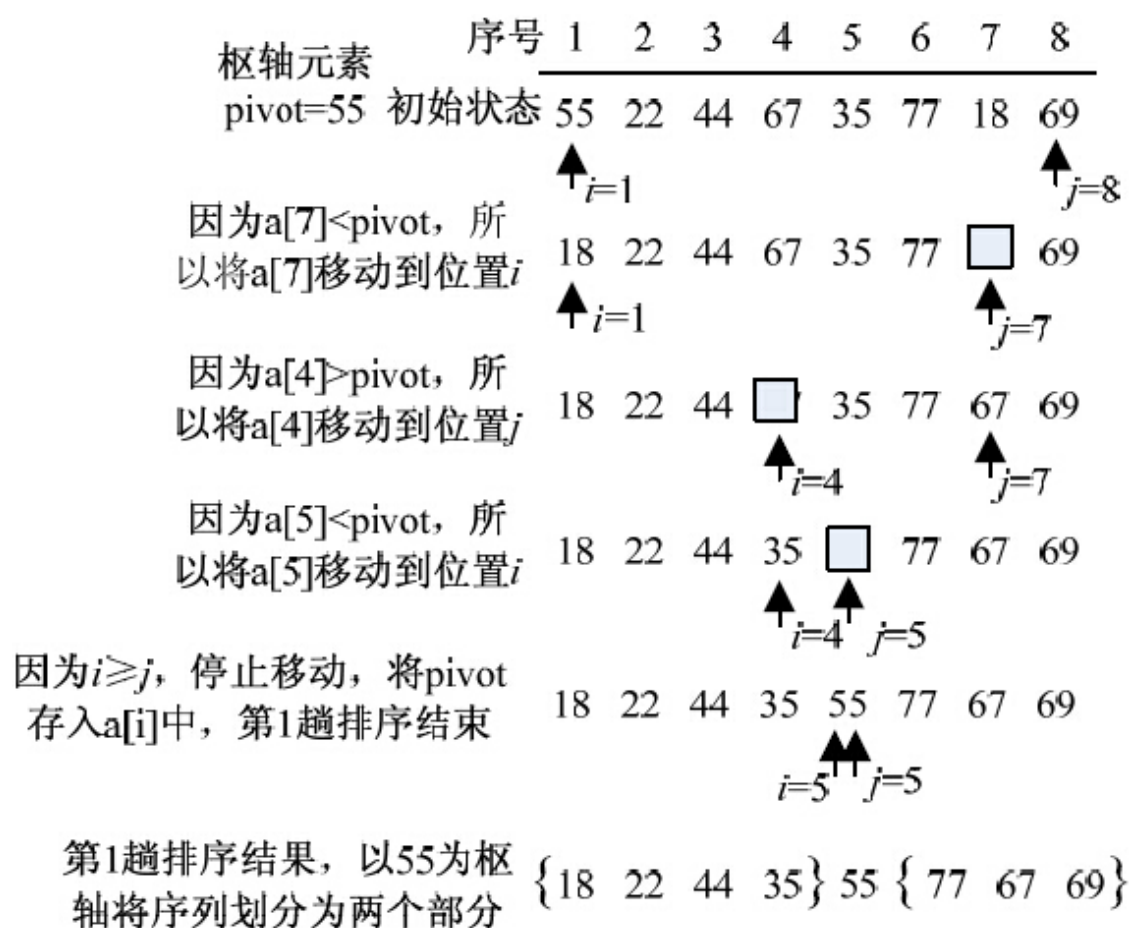


图12-16 第1趟快速排序过程

序号	1	2	3	4	5	6	7	8
初始状态	55	22	44	67	35	77	18	69
第1趟排序结果	{18	22	44	35}	55	{77	67	69}
第2趟排序结果	18	{22	44	35}	55	{69	67}	77
第3趟排序结果	18	22	{44	35}	55	{67}	69	77
第4趟排序结果	18	22	{35}	44	55	{67}	69	77
快速排序最终结果	18	22	35	44	55	67	69	77

图12-17 快速排序过程

快速排序可通过递归调用实现，通过上面的排序过程不难看出，排序的过程其实就是不断地对元素序列进行划分，直到每个部分不能划分时即完成快速排序。

进行一趟快速排序，即将元素序列进行一次划分，算法描述如下。

```

int Partition(SqList *L,int low,int high)
/*
对顺序表L.r[low..high]
的元素进行一趟排序，使枢轴前面的元素关键字小于枢轴元素的关键字，枢轴后面的
元素关键字大于等于枢轴元素的关键字，并返回枢轴位置*/
{
    DataType t;
    KeyType pivotkey;
    pivotkey=(*L).data[low].key;          /*
将表的第一个元素作为枢轴元素*/
    t=(*L).data[low];
    while(low<high)                        /*
从表的两端交替着向中间扫描*/
    {
        while(low<high&&(*L).data[high].key>=pivotkey) /*
从表的末端向前扫描*/
            high--;
        if(low<high)                        /*
... ..

```

```

将当前high
指向的元素保存在low
位置*/
{
    (*L).data[low]=(*L).data[high];
    low++;
}
while(low<high&&(*L).data[low].key<=pivotkey) /*
从表的始端向后扫描*/
    low++;
if(low<high) /*
将当前low
指向的元素保存在high
位置*/
{
    (*L).data[high]=(*L).data[low];
    high--;
}
(*L).data[low]=t; /*
将枢轴元素保存在low=high
的位置*/
return low; /*
返回枢轴所在位置*/
}

```

通过多次递归调用一次划分算法即一趟排序算法，可实现快速排序，其算法描述如下。

```

void QuickSort(SqList *L,int low,int high)
/*
对顺序表L
进行快速排序*/
{
    int pivot;
    if(low<high) /*
如果元素序列的长度大于1*/
    {
        pivot=Partition(L,low,high); /*
将待排序序列L.r[low..high]
划分为两部分*/
        QuickSort(L,low,pivot-1); /*
对左边的子表进行递归排序，pivot
是枢轴位置*/
        QuickSort(L,pivot+1,high); /*
对右边的子表进行递归排序*/
    }
}

```

容易看出，快速排序是一种不稳定的排序算法，其空间复杂度为 $O(\log_2 n)$ 。

快速排序在最好的情况下是每趟排序将序列一分两半，从表中间开始，将表分成两个大小相同的子表，类似折半查找，这样快速排序的划分的过程就将元素序列构成一个完全二叉树的结构，分解的次数等于树的深度即 $\log_2 n$ ，因此快速排序总的比较次数为 $T(n) \leq n + 2T(n/2) \leq n + 2 * (n/2 + 2 * T(n/4)) = 2n + 4T(n/4) \leq 3n + 8T(n/8) \leq \dots \leq n \log_2 n + nT(1)$ 。因此，在最好的情况下，时间复杂度为 $O(n \log_2 n)$ 。

快速排序在最坏的情况下是已经有序，第1趟经过 $n-1$ 次比较，第1条记录仍在原位置，左边为空表，右边为 $n-1$ 记录的表。第2趟 $n-1$ 个记录经过 $n-2$ 次比较，第2个记录在原位置，左边为空表，右边为 $n-2$ 个记录的表，依次类推，共需进行 $n-1$ 趟排序，其比较次数为

$$\sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2} \quad \text{因此时间复杂度为 } O(n^2)。$$

在平均情况下，快速排序的时间复杂度为 $O(n \log_2 n)$ 。

12.3.3 交换排序应用举例

【例12-3】 编写算法，使用冒泡排序和快速排序算法对给定的一组关键字序列（37，22，43，32，19，12，89，26，48，92）进行排序，并输出每趟排序结果。

【分析】 主要考查两种交换排序即冒泡排序和快速排序的算法思想。这两种算法都是对存在逆序的元素进行交换，从而实现排序。主要

区别在于冒泡排序通过比较相邻的两个元素，并对两个相邻的逆序元素进行交换；而快速排序则是选定一个枢轴元素作为参考元素，设置两个指针，分别从表头和表尾开始，将当前扫描的元素与枢轴元素进行比较，存在逆序的元素不一定是相邻的元素，如果存在逆序，则交换之。

将前面介绍的冒泡和快速排序做简单修改，使其能输出每趟排序的结果，代码如下。

```
#include<stdio.h>
#include<stdlib.h>
#define MaxSize 50
typedef int KeyType;
typedef struct /*
数据元素类型定义*/
{
    KeyType key; /*
关键字*/
}DataType;
typedef struct /*
顺序表类型定义*/
{
    DataType data[MaxSize];
    int length;
}SqList;
void InitSeqList(SqList *L,DataType a[],int n);
void DispList(SqList L);
void DispList2(SqList L,int count);
void DispList3(SqList L,int pivot,int count);
void HeapSort(SqList *H);
void BubbleSort(SqList *L,int n);
void QuickSort(SqList *L);
int Partition(SqList *L,int low,int high);
void DispList(SqList L)
/*
输出表中的元素*/
{
    int i;
    for(i=1;i<=L.length;i++)
        printf("%4d",L.data[i].key);
    printf("\n");
}
void DispList2(SqList L,int count)
/*
输出表中的元素（用于冒泡排序算法调用）*/
{
    int i;
    printf("
第%d
趟排序结果:",count);
    for(i=1;i<=L.length;i++)
        printf("%4d",L.data[i].key);
    printf("\n");
}
```

```

}
void DispList3(SqList L,int pivot,int count)
/*
输出每一趟排序后的元素序列（用于快速排序算法调用）*/
{
    int i;
    printf("
第%d
趟排序结果:[",count);
    for(i=1;i<pivot;i++)
        printf("%-4d",L.data[i].key);
    printf("]");
    printf("%3d ",L.data[pivot].key);
    printf("[");
    for(i=pivot+1;i<=L.length;i++)
        printf("%-4d",L.data[i].key);
    printf("]");
    printf("\n");
}
void InitSeqList(SqList *L,DataType a[],int n)
/*
顺序表的初始化*/
{
    int i;
    for(i=1;i<=n;i++)
    {
        L->data[i]=a[i-1];
    }
    L->length=n;
}
void main()
{
    DataType a[]={55,22,44,67,35,77,18,69};
    SqList L;
    int n=sizeof(a)/sizeof(a[0]);
    /*
冒泡排序*/
    InitSeqList(&L,a,n);
    printf("
冒泡排序前:");
    DispList(L);
    BubbleSort(&L,n);
    printf("
冒泡排序结果:");
    DispList(L);
    /*
快速排序*/
    InitSeqList(&L,a,n);
    printf("
快速排序前:");
    DispList(L);
    QuickSort(&L);
    printf("
快速排序结果:");
    DispList(L);
}
/*
冒泡排序算法部分*/
void BubbleSort(SqList *L,int n)
/*
冒泡排序*/
{
    int i,j,flag;
    DataType t;
    static int count=1;
    for(i=1;i<=n-1&&flag;i++)
        需要进行n-1
        .....
}
/*

```

```

一趟排序*/
    {
        flag=0;
        for (j=1; j<=n-i; j++) /*
每一趟排序需要比较n-i
次*/
            if (L->data[j].key>L->data[j+1].key)
            {
                t=L->data[j];
                L->data[j]=L->data[j+1];
                L->data[j+1]=t;
                flag=1;
            }
        DispList2(*L, count);
        count++;
    }
}
/*
快速排序算法部分*/
void QSort(SqList *L, int low, int high)
/*
对顺序表L
进行快速排序*/
{
    int pivot;
    static count=1;
    if (low<high) /*
如果元素序列的长度大于1*/
    {
        pivot=Partition(L, low, high); /*
将待排序序列L.r[low..high]
划分为两部分*/
        DispList3(*L, pivot, count); /*
输出每次划分的结果*/
        count++;
        QSort(L, low, pivot-1); /*
对左边的子表进行递归排序, pivot
是枢轴位置*/
        QSort(L, pivot+1, high); /*
对右边的子表进行递归排序 */
    }
}
void QuickSort(SqList *L)
/*
对顺序表L
作快速排序*/
{
    QSort(L, 1, (*L).length);
}
int Partition(SqList *L, int low, int high)
/*
对顺序表L.r[low..high]
的元素进行一趟排序, 使枢轴前面的元素关键字小于枢轴元素的关键字, 枢轴后面的
元素关键字大于等于枢轴元素的关键字, 并返回枢轴位置*/
{
    DataType t;
    KeyType pivotkey;
    pivotkey=(*L).data[low].key; /*
将表的第一个元素作为枢轴元素*/
    t=(*L).data[low];
    while (low<high) /*
从表的两端交替地向中间扫描*/
    {
        while (low<high&&(*L).data[high].key>=pivotkey) /*
从表的末端向前扫描*/
            high--;
        if (low<high) /*
... ..

```

```

将当前high
指向的元素保存在low
位置*/
{
    (*L).data[low]=(*L).data[high];
    low++;
}
while(low<high&&(*L).data[low].key<=pivotkey)/*
从表的始端向后扫描*/
    low++;
if(low<high)/*
将当前low
指向的元素保存在high
位置*/
{
    (*L).data[high]=(*L).data[low];
    high--;
}
(*L).data[low]=t;/*
将枢轴元素保存在low=high
的位置*/
}
return low;/*
返回枢轴所在位置*/
}

```

程序运行结果如图12-18所示。

```

D:\学基础学数据结构\例12_2\Debug\例12_2.exe
冒泡排序前: 55 22 44 67 35 77 18 69
第1趟排序结果: 22 44 55 35 67 18 69 77
第2趟排序结果: 22 44 35 55 18 67 69 77
第3趟排序结果: 22 35 44 18 55 67 69 77
第4趟排序结果: 22 35 18 44 55 67 69 77
第5趟排序结果: 22 18 35 44 55 67 69 77
第6趟排序结果: 18 22 35 44 55 67 69 77
第7趟排序结果: 18 22 35 44 55 67 69 77
冒泡排序结果: 18 22 35 44 55 67 69 77
快速排序前: 55 22 44 67 35 77 18 69
第1趟排序结果: [18 22 44 35 ] 55 [77 67 69 ]
第2趟排序结果: [ ] 18 [22 44 35 55 77 67 69 ]
第3趟排序结果: [18 ] 22 [44 35 55 77 67 69 ]
第4趟排序结果: [18 22 35 ] 44 [55 77 67 69 ]
第5趟排序结果: [18 22 35 44 55 67 69 ] 77 [ ]
第6趟排序结果: [18 22 35 44 55 67 ] 69 [77 ]
快速排序结果: 18 22 35 44 55 67 69 77
Press any key to continue

```

图12-18 交换排序的程序运行结果

12.4 选择排序

选择排序（selection sort）的基本思想是每一趟排序从 $n-i+1$ 个记录中选取关键字最小的记录作为有序序列的第 i 个记录。其中最常用的是简单选择排序。

12.4.1 简单选择排序

简单选择排序是一种简单的选择类排序算法，它是通过依次找到待排序元素序列中最小的数据元素，并将其放在序列的最前面，从而使待排序元素序列变为有序序列。它的基本算法思想描述如下。

假设待排序的元素序列有 n 个，在第一趟排序过程中，从 n 个元素序列中选择最小的元素，并将其放在元素序列的最前面即第一个位置。在第二趟排序过程中，从剩余的 $n-1$ 个元素中选择最小的元素，将其放在第二个位置。依次类推，直到没有待比较的元素，简单选择排序算法结束。

简单选择排序的算法描述如下。

```
void SelectSort(SqList *L,int n)
/*
简单选择排序*/
{
    int i,j,k;
    DataType t;
    /*
将第i
个元素的关键字与后面[i+1..n]
个元素的关键字比较，将关键字最小的元素放在第i
个位置*/
    for(i=1;i<=n-1;i++)
    {
        j=i;
        for(k=i+1;k<=n;k++) /*
关键字最小的元素的序号为j*/
            if(L->data[k].key<L->data[j].key)
                j=k;
        if(j!=i) /*
如果序号i
不等于j
，则需要将序号i
和序号j
的元素交换*/
        {
            t=L->data[i];
            L->data[i]=L->data[j];
            L->data[j]=t;
        }
    }
}
```

例如一组元素的关键字序列为（76，31，19，20，6，83，60，52），则简单选择排序的过程如图12-19所示。

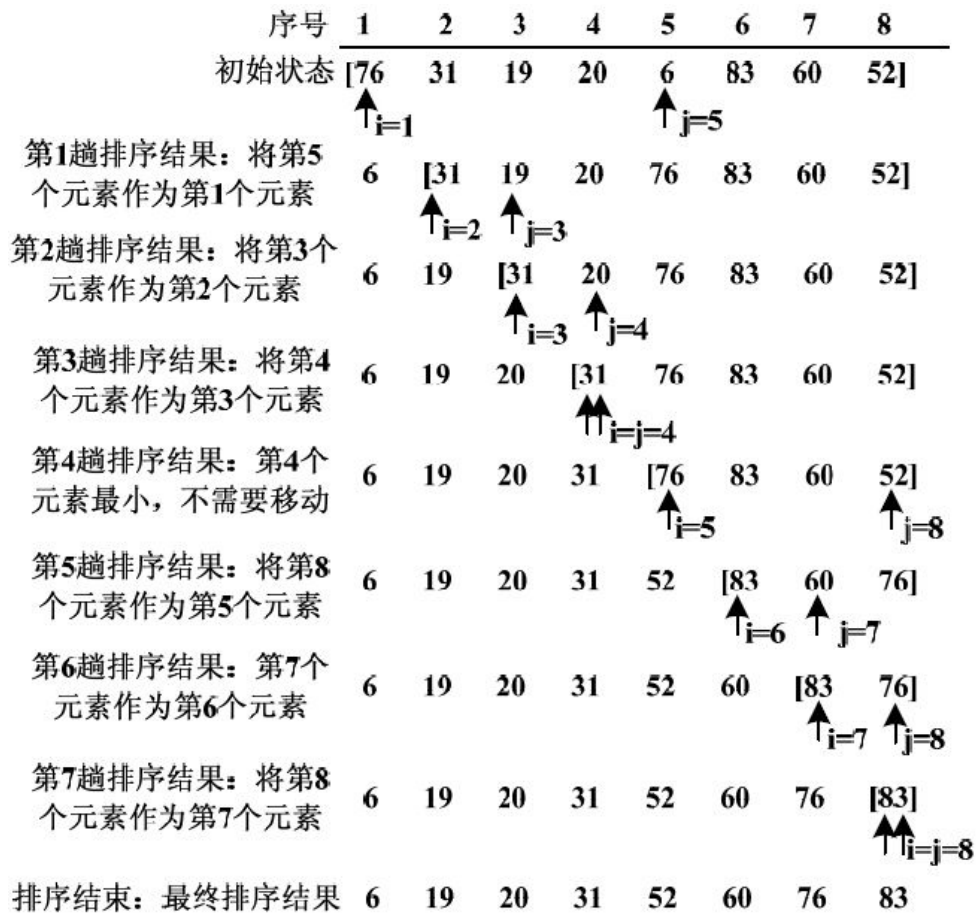


图12-19 简单选择排序全过程

简单选择是一种不稳定的排序算法，在最好的情况下，待排序元素序列按照非递减排列，则不需要移动元素；在最坏的情况下，待排序元素按照非递增排列，则在每一趟排序都需要移动元素，移动元素的次数为3（n-1）。在任何情况下，简单选择排序算法都需要进行n（n-1）/2次的比较。综上所述，简单选择排序算法的时间复杂度是O（n²）。

简单选择排序的空间复杂度是O（1）。

12.4.2 堆排序

堆排序的算法思想主要是利用了二叉树的性质进行排序。下面主要介绍堆的定义、创建堆和堆排序。

1. 什么是堆和堆排序

堆排序（heap sort）利用二叉树的树形结构进行排序。堆中的每一个结点都大于（或小于）其孩子结点。堆的数学形式定义为：假设存在n个元素，其关键字序列为（ $k_1, k_2, \dots, k_i, \dots, k_n$ ），如果有：

$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \text{ 或 } \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases}$$

其中， $i=1,2,\dots,\lfloor \frac{n}{2} \rfloor$ 。则称此元素序列构成一个堆。如果将这些元素的关键字存放在一维数组中，将此一维数组中的元素与完全二叉树一一对应，则完全二叉树中的每个非叶子结点的值都不小于（或不大于）孩子结点的值。

在堆中，堆的根结点元素值一定是所有结点元素值的最大值或最小值。例如序列（89, 77, 65, 62, 32, 55, 60, 48）和（18, 37, 29, 48, 50, 43, 33, 69, 77, 60）都是堆，相应的完全二叉树表示如图12-20所示。

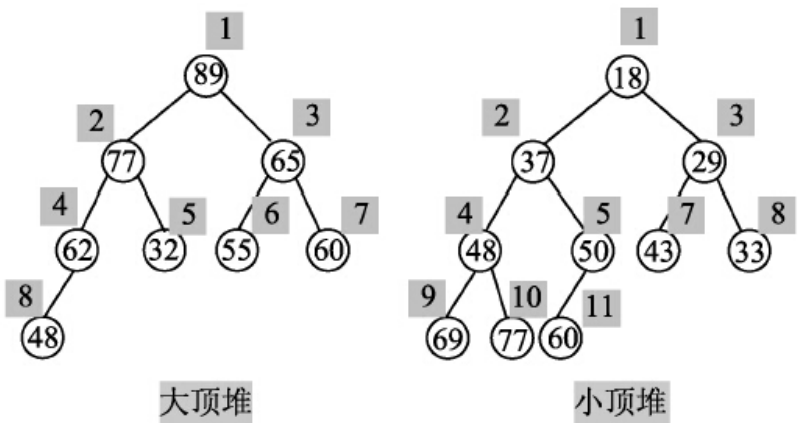


图12-20 堆的示意图

在图12-20所示的堆中，一个是非叶子结点的元素值不小于其孩子结点的值，这样的堆称为**大顶堆**。另一个是非叶子结点的元素值不大于其孩子结点的元素值，这样的堆称为**小顶堆**。

按照完全二叉树的编号次序，将元素序列的关键字依次存放在相应的结点。然后从叶子结点开始，从互为兄弟的两个结点中（没有兄弟结点除外），选择一个较大（或较小）者与其双亲结点比较，如果该结点大于（或小于）双亲结点，则将两者进行交换，使较大（或较小）者成为双亲结点。将所有的结点都做类似操作，直到根结点为止。这时，根结点的元素值的关键字最大（或最小）。

如果将堆中的根结点（堆顶）输出之后将剩余的 $n-1$ 个结点的元素值重新建立一个堆，则新堆的堆顶元素值是次大（或次小）值，将该堆顶元素输出，然后将剩余的 $n-2$ 个结点的元素值重新建立一个堆。反复执行以上操作，直到堆中没有结点，就构成了一个有序序列，这样的重复建堆并输出堆顶元素的过程称为**堆排序**。

2. 建堆

堆排序的过程就是建立堆和不断调整使剩余结点构成新堆的过程。假设将待排序的元素的关键字存放在数组 a 中，第1个元素的关键字 $a[1]$ 表示二叉树的根结点，剩下的元素的关键字 $a[n]$ 分别与二叉树中的结点按照层次从左到右一一对应。例如， $a[1]$ 的左孩子结点存放在 $a[2]$ 中，右孩子结点存放在 $a[3]$ 中， $a[i]$ 的左孩子结点存放在 $a[2i]$ 中，右孩子结点存放在 $a[2i+1]$ 中。

如果是大顶堆，则有 $a[i].key \geq a[2i].key$ 且 $a[i].key \geq a[2i+1].key (i=1, 2, \dots, \lfloor \frac{n}{2} \rfloor)$ 。如果是小顶堆，则有 $a[i].key \leq a[2i].key$ 且 $a[i].key \leq a[2i+1].key (i=1, 2, \dots, \lfloor \frac{n}{2} \rfloor)$ 。

建立一个大顶堆就是将一个无序的关键字序列构建为一个满足条件 $a[i] \geq a[2i]$ 且 $a[i] \geq a[2i+1] (i=1, 2, \dots, \lfloor \frac{n}{2} \rfloor)$ 的序列。

建立大顶堆的算法思想：从位于元素序列中的最后一个非叶子结点（即第 $\lfloor \frac{n}{2} \rfloor$ 个元素）开始，逐层比较，直到根结点为止。假设当前结点的序号为*i*，则当前元素为*a*[*i*]，其左、右孩子结点元素分别为*a*[2*i*]和*a*[2*i*+1]。将*a*[2*i*].key和*a*[2*i*+1].key之中的较大者与*a*[*i*]比较，如果孩子结点元素值大于当前结点值，则交换两者；否则不进行交换。逐层向上执行此操作，直到根结点，这样就建立了一个大顶堆。建立小顶堆的算法与此类似。

例如给定一组元素序列（27，58，42，53，42，69，50，62），建立大顶堆的过程如图12-21所示。

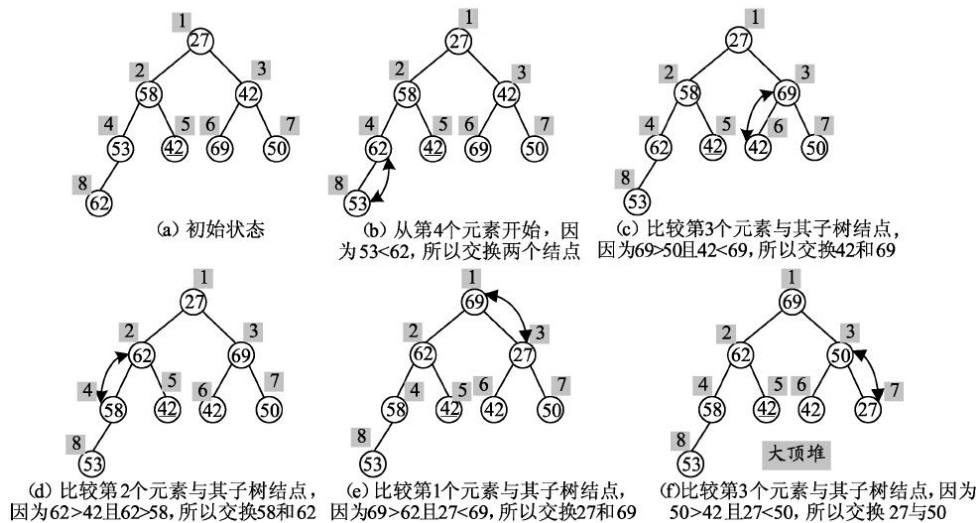


图12-21 建立大顶堆的过程

从图12-21容易看出，建立后的大顶堆中的孩子结点元素值都小于等于双亲结点元素值，其中，根结点的元素值69是最大的元素。创建后的堆的元素序列为69，62，50，58，42，42，27，53。

相应地，建立大顶堆的算法描述如下所示。

```
void CreateHeap(SqList *H,int n)
/*
建立大顶堆*/
{
    int i;
    for(i=n/2;i>=1;i--) /*
从序号n/2
开始建立大顶堆*/
        AdjustHeap(H,i,n);
}
void AdjustHeap(SqList *H,int s,int m)
/*
.....
*/
```

```

调整H.data[s...m]
的关键字，使其成为一个大顶堆*/
{
    DataType t;
    int j;
    t=(*H).data[s];
    将根结点暂时保存在t
    中*/
    for(j=2*s;j<=m;j*=2)
    {
        if(j<m&&(*H).data[j].key<(*H).data[j+1].key) /*
        沿关键字较大的孩子结点向下筛选*/
            j++;
        为关键字较大的结点的下标*/
        if(t.key>(*H).data[j].key) /*
        如果孩子结点的值小于根结点的值，则不进行交换*/
            break;
        (*H).data[s]=(*H).data[j];
        s=j;
    }
    (*H).data[s]=t;
    将根结点插入正确位置*/
}

```

3. 调整堆

建立好一个大顶堆后，当输出堆顶元素后，如何调整剩下的元素，使其构成一个新的大顶堆呢？其实，这也是一个建堆的过程，由于除了堆顶元素外，剩下的元素本身就具有 $a[i].key \geq a[2i].key$ 且 $a[i].key \geq a[2i+1].key (i=1,2,\dots,\lfloor \frac{n}{2} \rfloor)$ 的性质，关键字按照由大到小逐层排列，因此，调整剩下的元素构成新的大顶堆只需要从上往下进行比较找出最大的关键字，并将其放在根结点的位置就又构成了新的堆。

具体实现：当堆顶元素输出后，可以将堆顶元素放在堆的最后，即将第1个元素与最后一个元素交换 $a[1] \leftrightarrow a[n]$ ，则需要调整的元素序列就是 $a[1 \cdots n-1]$ 。从根结点开始，如果其左、右子树结点元素值大于根结点元素值，选择较大的一个进行交换。即如果 $a[2] > a[3]$ ，则将 $a[1]$ 与 $a[2]$ 比较；如果 $a[1] < a[2]$ ，则将 $a[1]$ 与 $a[2]$ 交换，否则不交换。如果 $a[2] < a[3]$ ，则将 $a[1]$ 与 $a[3]$ 比较；如果 $a[1] < a[3]$ ，则将 $a[1]$ 与 $a[3]$ 交换，否则不交换。重复执行此操作，直到叶子结点不存在，就完成了堆的调整，构成了一个新堆。

例如一个大顶堆的关键字序列为 (69, 62, 50, 58, 42, 42, 27, 53)，当输出69后，调整剩余的元素序列为大顶堆的过程如图12-22所示。

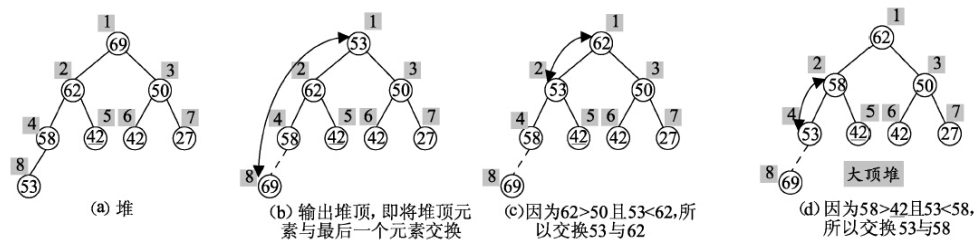


图12-22 输出堆顶元素后，调整堆的过程

如果重复地输出堆顶元素，即将堆顶元素与堆的最后一个元素交换，然后重新调整剩余的元素序列使其构成一个新的大顶堆，直到没有需要输出的元素为止，就会把元素序列构成一个有序的序列，即完成了一个排序的过程。

调整堆的算法实现如下。

```

void HeapSort(SqList *H)
/*
对顺序表H
进行堆排序*/
{
    DataType t;
    int i;
    CreateHeap(H, H->length);    /*
创建堆*/
    for (i = (*H).length; i > 1; i--)    /*
将堆顶元素与最后一个元素交换，重新调整堆*/
    {
        t = (*H).data[1];
        (*H).data[1] = (*H).data[i];
        (*H).data[i] = t;
        AdjustHeap(H, 1, i-1);    /*
将 (*H).data[1..i-1]
调整为大顶堆*/
    }
}

```

例如一个大顶堆的元素的关键字序列为（69，62，50，58，42，42，27，53），其相应的完整的堆排序过程如图12-23所示。

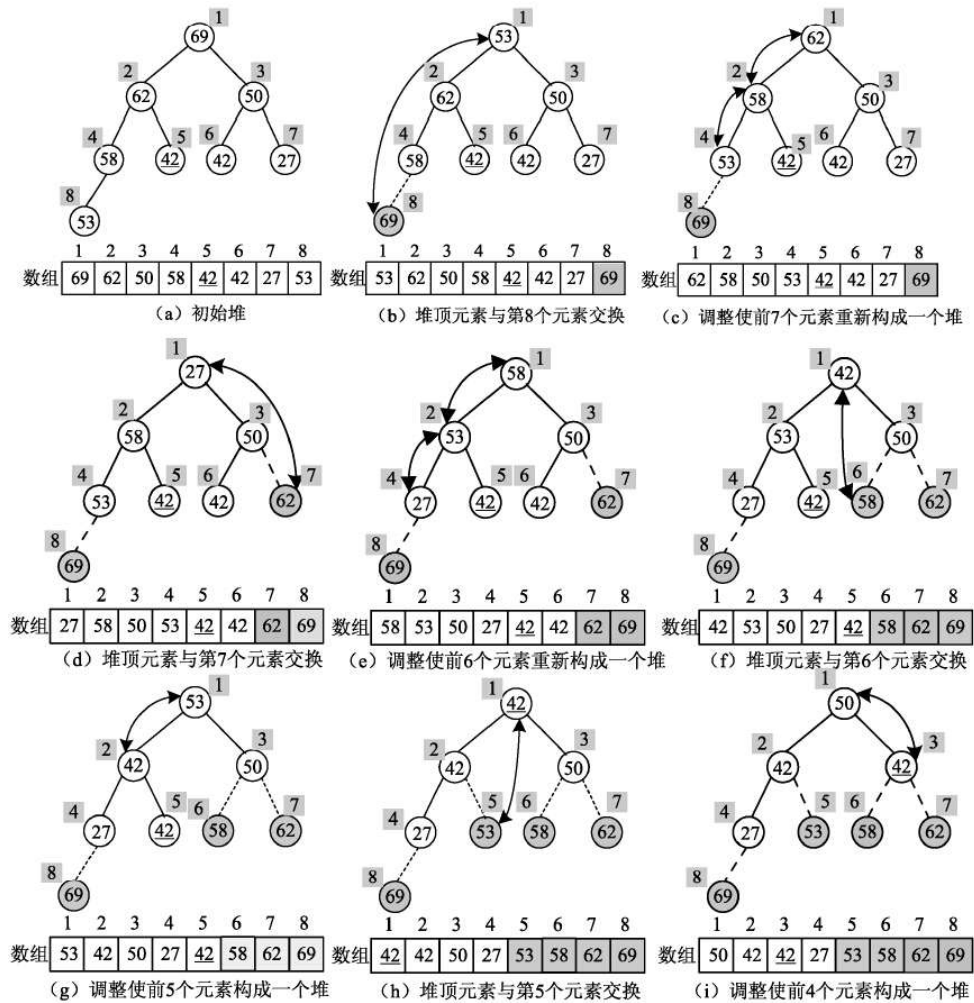


图12-23 一个完整的堆排序过程

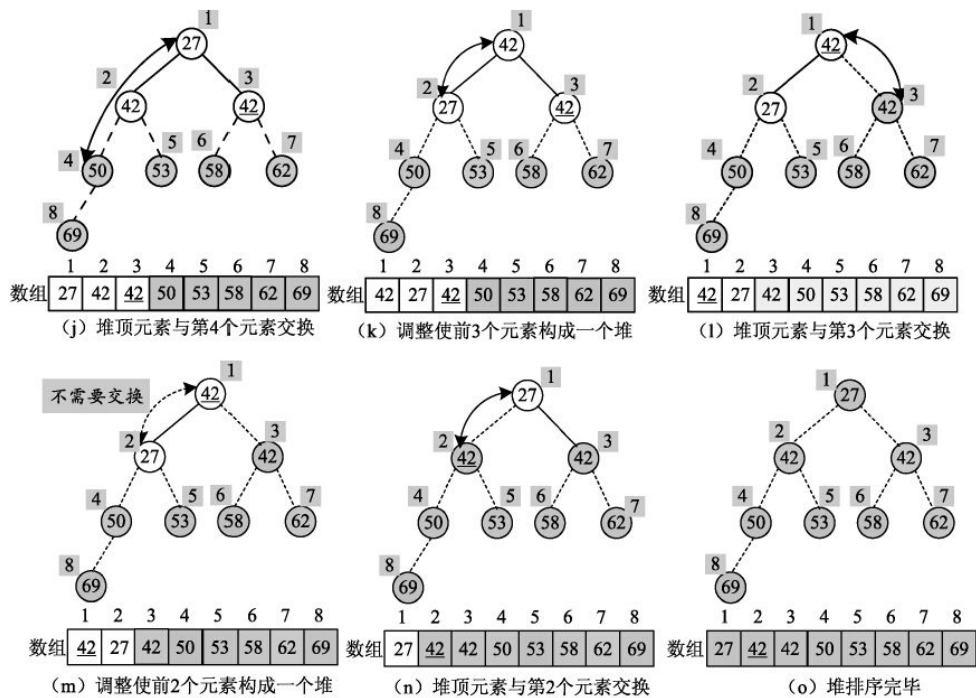


图12-23 （续）

从上面的例子不难看出，堆排序属于不稳定的排序算法。

堆排序的时间耗费主要是在建立堆和调整堆时。一个深度为 h ，元素个数为 n 的堆，其调整算法的比较次数最多为 $2(h-1)$ 次，而建立一个堆，其比较次数最多为 $4n$ 。一个完整的堆排序过程总共的比较次数为 $2(\lfloor \log_2(n-1) \rfloor + \lfloor \log_2(n-2) \rfloor + \dots + \lfloor \log_2 2 \rfloor) < 2n \log_2 n$ ，因此，堆排序平均时间复杂度和最坏情况下的时间复杂度都是 $O(n \log_2 n)$ 。

堆排序的空间复杂度为 $O(1)$ 。

12.4.3 选择排序应用举例

【例12-4】 编写算法，利用简单选择排序和堆排序算法对一组关键字序列（69，62，50，58，42，42，27，53）进行排序，要求输出每趟排序的结果。

【分析】 简单选择排序和堆排序都是一种不稳定的排序方法。它们的主要思想是每次从待排序元素中选择关键字最小（或最大）的元素，经过不断交换，重复执行以上操

作，最后形成一个有序的序列。

算法实现代码如下。

```
/*
头文件*/
#include<stdio.h>
#include<stdlib.h>
#define MaxSize 50
typedef int KeyType;
typedef struct /*
数据元素类型定义*/
{
    KeyType key; /*
关键字*/
}DataType;
typedef struct /*
顺序表类型定义*/
{
    DataType data[MaxSize];
    int length;
}SqList;
void InitSeqList(SqList *L,DataType a[],int n);
void DispList(SqList L,int n);
void AdjustHeap(SqList *H,int s,int m);
void CreateHeap(SqList *H,int n);
void HeapSort(SqList *H);
void SelectSort(SqList *L,int n);
void main()
{
    DataType a[]={69,62,50,58,42,42,27,53};
    SqList L;
    int n=sizeof(a)/sizeof(a[0]);
    /*
简单选择排序*/
    InitSeqList(&L,a,n);
    printf("[
排序前]
");
    DispList(L,n);
    SelectSort(&L,n);
    printf("[
简单选择排序结果]")
    DispList(L,n);
    /*
堆排序*/
    InitSeqList(&L,a,n);
    printf("[
排序前]
");
    DispList(L,n);
    HeapSort(&L);
    printf("[
堆排序结果]
");
    DispList(L,n);
}
void InitSeqList(SqList *L,DataType a[],int n)
/*
顺序表的初始化*/
{
    int i;
    for(i=1;i<=n;i++)
    {
        L->data[i]=a[i-1];
    }
    L->length=n;
}
void DispList(SqList L,int n)
/*
输出表中的元素*/
{
    int i;
    for(i=1;i<=n;i++)
        printf("%4d",L.data[i].key);
    printf("\n");
}
void HeapSort(SqList *H)
/*
调整后的堆排序算法，使其能输出每趟的排序结果*/
{
```

```

        DataType t;
        int i;
        CreateHeap(H,H->length);          /*
创建堆*/
        for(i=(*H).length;i>1;i--)        /*
将堆顶元素与最后一个元素交换,重新调整堆*/
        {
            t=(*H).data[1];
            (*H).data[1]=(*H).data[i];
            (*H).data[i]=t;
            AdjustHeap(H,1,i-1);          /*
将(*H).data[1..i-1]
调整为大顶堆*/
            printf("[
第%d
趟排序后结果] ",H->length-i+1);
            DispList(*H,H->length);
        }
    }
}

```

程序运行结果如图12-24所示。

```

D:\季基础学数据结构\例12_3\Debug\例12_3.exe
[排序前]      69  62  50  58  42  42  27  53
[简单选择排序结果] 27  42  42  50  53  58  62  69
[排序前]      69  62  50  58  42  42  27  53
[第1趟排序后结果] 62  58  50  53  42  42  27  69
[第2趟排序后结果] 58  53  50  27  42  42  62  69
[第3趟排序后结果] 53  42  50  27  42  58  62  69
[第4趟排序后结果] 50  42  42  27  53  58  62  69
[第5趟排序后结果] 42  27  42  50  53  58  62  69
[第6趟排序后结果] 42  27  42  50  53  58  62  69
[第7趟排序后结果] 27  42  42  50  53  58  62  69
[堆排序结果]   27  42  42  50  53  58  62  69
Press any key to continue

```

图12-24 选择排序程序运行结果

【例12-5】 编写算法,对关键字序列(76, 20, 99, 32, 60, 53, 11, 8, 42)进行选择排序,要求使用链表实现。

【分析】 主要考查选择排序的算法思想和链表的操作。具体实现时,设置两个指针p和q,分别指向已排序链表和未排序链表。初始时,先创建一个链表,q指向该链表,p指向的链表为空。然后从q指向的链表中找到一个元素值最小的结点,将其取出并插入p指向的链表中。重复执行以上操作直到q指向的链表为空,此时p指向的链表就是一个有序链表。

算法实现代码如下。

```

/*
头文件*/
#include<stdio.h>
#include<malloc.h>

```



```

#include<stdlib.h>
typedef int DataType;          /*
元素类型定义为整型*/
typedef struct Node            /*
单链表类型定义*/
{
    DataType data;
    struct Node *next;
}ListNode,*LinkList;
#include"LinkList.h"
void SelectSort(LinkList L);
void CreateList(LinkList L,DataType a[],int n);
void CreateList(LinkList L,DataType a[],int n)
/*
创建单链表*/
{
    int i;
    for(i=1;i<=n;i++)
        InsertList(L,i,a[i-1]);
}
void main()
{
    LinkList L,p;
    DataType a[]={76,20,99,32,60,53,11,8,42};
    int n=sizeof(a)/sizeof(a[0]);
    InitList(&L);
    CreateList(L,a,n);
    printf("
排序前的元素序列: \n");
    for(p=L->next;p!=NULL;p=p->next)
        printf("%4d ",p->data);
    printf("\n");
    SelectSort(L);
    printf("
排序后的元素序列: \n");
    for(p=L->next;p!=NULL;p=p->next)
        printf("%4d ",p->data);
    printf("\n");
}
void SelectSort(LinkList L)
/*
用链表实现选择排序。将链表分为两段，p
指向应经排序的链表部分，q
指向未排序的链表部分*/
{
    ListNode *p,*q,*t,*s;
    p=L;
    while(p->next->next!=NULL)
    {
        for(s=p,q=p->next;q->next!=NULL;q=q->next)    /*
用q
指针进行遍历链表*/
            if(q->next->data<s->next->data)    /*
如果q
指针指向的元素值小于s
指向的元素值，则s=q*/
                s=q;
        if(s!=q)    /*
如果*s
不是最后一个结点，则将s
指向的结点链接到p
指向的链表后面*/
        {
            t=s->next;    /*
将结点*t
从q
指向的链表中取出*/
            s->next=t->next;
            t->next=p->next;    /*
将结点*t
插入p
指向的链表中*/
            p->next=t;
        }
        p=p->next;
    }
}

```

程序运行结果如图12-25所示。



图12-25 采用链式存储结构的选择排序程序运行结果

12.5 归并排序

归并排序（merging sort）的算法思想是将两个或两个以上的元素有序序列合并为一个有序序列，也就是说，待排序元素序列被划分为若干个子序列，每个子序列都是有序的，通过将有序子序列合并为整体有序的序列就是归并排序。其中，最常见的是2路归并排序。

12.5.1 2路归并排序算法

2路归并排序 的主要思想是假设元素的个数是 n ，将每个元素作为一个有序的子序列，然后将相邻的两个子序列两两归并，得到 $\lceil \frac{n}{2} \rceil$ 个长度为2的有序子序列；再将相邻的两个有序子序列两两归并，得到 $\lceil \frac{n}{4} \rceil$ 个长度为4的有序子序列；如此重复，直至得到一个长度为 n 的有序序列为止。

一组元素的关键字序列为（50，22，61，35，87，12，19，75），2路归并排序的过程如图12-26所示。

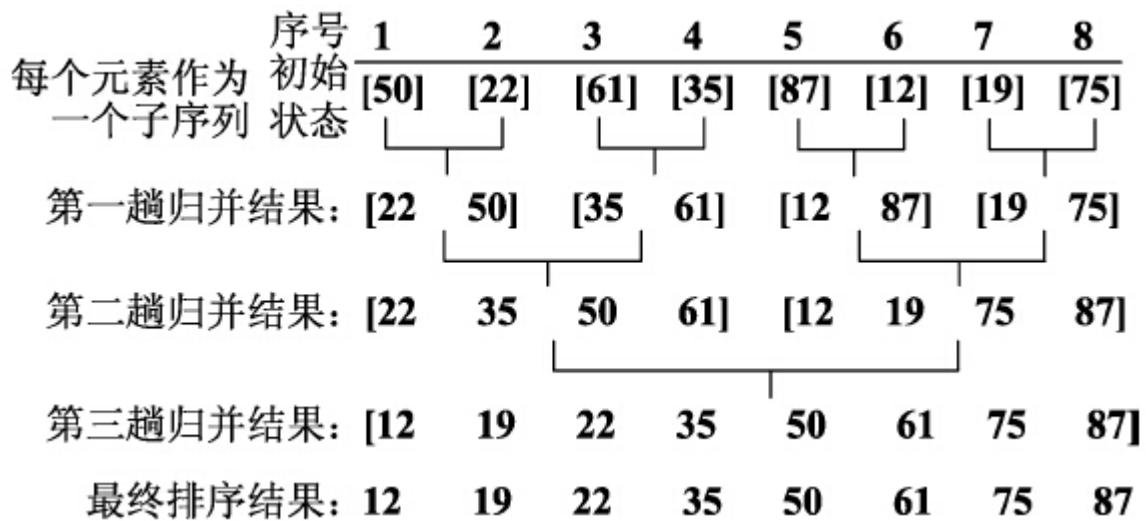


图12-26 2路归并排序过程

2路归并排序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，其算法描述如下。

```

void Merge(DataType s[],DataType t[],int low,int mid,int high)
/*
将有序的s[low...mid]
和s[mid+1..high]
归并为有序的t[low..high]*/
{
    int i,j,k;
    i=low,j=mid+1,k=low;
    while(i<=mid&&j<=high)    /*
将s
中元素由小到大合并到t*/
    {
        if(s[i].key<=s[j].key)
        {
            t[k]=s[i++];
        }
        else
        {
            t[k]=s[j++];
        }
        k++;
    }
    while(i<=mid)                /*
将剩余的s[i..mid]
复制到t*/
        t[k++]=s[i++];
    while(j<=high)                /*
将剩余的s[j..high]
复制到t*/
        t[k++]=s[j++];
}

```

以上是归并两个子表的算法，可通过递归调用以上算法归并所有子表从而实现2路归并排序。其2路归并算法描述如下。

```
void MergeSort(DataType s[],DataType t[],int low, int high)
/*2
路归并排序，将s[low...high]
归并排序并存储到t[low...high]
中*/
{
    int mid;
    DataType t2[MaxSize];
    if(low==high)
        t[low]=s[low];
    else
    {
        mid=(low+high)/2; /*
将s[low...high]
分为s[low...mid]
和s[mid+1...high]*/
        MergeSort(s,t2,low,mid); /*
将s[low...mid]
归并为有序的t2[low...mid]*/
        MergeSort(s,t2,mid+1,high); /*
将s[mid+1...high]
归并为有序的t2[mid+1...high]*/
        Merge(t2,t,low,mid,high); /*
将t2[low...mid]
和t2[mid+1...high]
归并到t[low...high]*/
    }
}
```

容易看出，归并排序需要与元素个数相等的空间作为辅助空间，因此归并排序的空间复杂度为 $O(n)$ 。由于2路归并排序过程中所使用的空间过大，因此，它主要被用在外部排序中。2路归并排序算法需要多次递归调用自己，其递归调用的过程可以构成一个二叉树的结构，它的时间复杂度为 $T(n) \leq n + 2T(n/2) \leq n + 2(n/2 + 2T(n/4)) = 2n + 4T(n/4) \leq 3n + 8T(n/8) \leq \dots \leq n \log_2 n + nT(1)$ ，即 $O(n \log_2 n)$ 。

2路归并排序是一种稳定的排序算法。

12.5.2 归并排序应用举例

【例12-6】 编写算法，请使用2路归并排序对一组关键字（50，22，61，35，87，12，19，75）进行排序。

算法实现代码如下。

```
/*
头文件*/
#include<stdio.h>
#include<stdlib.h>
#define MaxSize 100
typedef int KeyType;
typedef struct /*
数据元素类型定义*/
{
    KeyType key; /*
关键字*/
}DataType;
typedef struct /*
顺序表类型定义*/
{
    DataType data[MaxSize];
    int length;
}SqlList;
void InitSeqList(SqlList *L,DataType a[],int start,int n);
void DispList(SqlList L);
void DispArray(DataType a[],int low,int high);
void MergeSort(DataType s[],DataType t[],int low, int high);
void Merge(DataType s[],DataType t[],int low,int mid,int high);
int N=0;
void main()
{
    DataType a[]={50,22,61,35,87,12,19,75};
    DataType b[MaxSize];
    int n=sizeof(a)/sizeof(a[0]);
    SqlList L,L2;
    /*
归并排序*/
    InitSeqList(&L,a,0,n); /*
将数组a[0...n-1]
初始化为顺序表L*/
    printf("
归并排序前:  ");
    DispList(L);
    MergeSort(L.data,b,1,n);
    InitSeqList(&L2,b,1,n); /*
将数组b[1...n]
... ..
... ..
```


```

初始化为顺序表L2*/
printf("
归并排序结果: ");
DispList(L2);
}
void InitSeqList(Sqlist *L,DataType a[],int start,int n)
/*
顺序表的初始化*/
{
    int i,k;
    for(k=1,i=start;i<start+n;i++,k++)
    {
        L->data[k]=a[i];
    }
    L->length=n;
}
void DispList(Sqlist L)
/*
输出表中的元素*/
{
    int i;
    for(i=1;i<=L.length;i++)
        printf("%4d",L.data[i].key);
    printf("\n");
}
void DispArray(DataType a[],int low,int high)
{
    int i;
    for(i=low;i<=high;i++)
        printf("%4d",a[i]);
    printf("\n");
}
void Merge(DataType s[],DataType t[],int low,int mid,int high)
/*
将有序的s[low...mid]
和s[mid+1...high]
归并为有序的t[low...high]*/
{
    int i,j,k;
    i=low,j=mid+1,k=low;
    while(i<=mid&&j<=high)    /*
将s
中元素由小到大合并到t*/
    {
        if(s[i].key<=s[j].key)
        {
            t[k]=s[i++];
        }
        else
        {
            t[k]=s[j++];
        }
        k++;
    }
    while(i<=mid)    /*
将剩余的s[i...mid]
复制到t*/
        t[k++]=s[i++];
    while(j<=high)    /*
将剩余的s[j...high]
复制到t*/
        t[k++]=s[j++];
    printf("
第%d
次归并后: ",++N);

```

```
        DispArray(t,low,high);  
    }  
}
```

程序运行结果如图12-27所示。



```
例12_6\Debug\例12_6...  
归并排序前: 50 22 61 35 87 12 19 75  
第1次归并后: 22 50  
第2次归并后: 35 61  
第3次归并后: 22 35 50 61  
第4次归并后: 12 87  
第5次归并后: 19 75  
第6次归并后: 12 19 75 87  
第7次归并后: 12 19 22 35 50 61 75 87  
归并排序结果: 12 19 22 35 50 61 75 87  
Press any key to continue
```

图12-27 归并排序程序运行结果

12.6 基数排序

基数排序是一种与前面所述各种排序方法完全不同的方法，前面的排序主要通过元素的关键字进行比较和移动记录这两种操作，而实现基数排序则不需要进行对关键字比较。本节主要介绍基数排序的算法及实现。

12.6.1 基数排序算法

基数排序主要是利用多个关键字进行排序，在日常生活中，扑克牌就是一种多关键字的排序问题。扑克牌有4种花色即红桃、方块、梅花和黑桃，每种花色从A到K共13张牌。

将一副扑克牌的排序过程看成由花色和面值两个关键字进行排序的问题，若规定花色和面值的顺序如下。

- 花色：黑桃<梅花<方块<红桃。
- 面值：A<2<3<4<5<6<7<8<9<10<J<Q<K

并进一步规定花色的优先级高于面值，则一副扑克牌从小到大的顺序为黑桃A、黑桃2、…、黑桃K；梅花A、梅花2、…、梅花K；方块A、方块2、…、方块K；红桃A、红桃2、…、红桃K。具体进行排序时有两种做法。

(1) 先按花色分成4类，然后再按面值对每一类从小到大排序，该方法称为“高位优先”排序法。

(2) 分配和收集交替进行，首先按面值从小到大把牌摆成13打（每打4张牌），然后将每打牌按面值的次序收集到一起，再对这些牌按花色摆成4打，每打13张牌，最后把这4打牌按花色的次序收集到一块，于是就得到了上述序列。

该方法称为“低位优先”排序法。

基数排序正是借助这种思想，对不同类的元素进行分类，然后对同一类中的元素进行排序，通过这样的一种过程，完成对元素序列的排序。在基数排序中，通常将对不同元素的分类称为分配，排序的过程称为收集。

具体算法思想是假设第 i 个元素 a_i 的关键字 key_i ， key_i 是由 d 位十进制组成，即 $key_i = k_i^d k_i^{d-1} \dots k_i^1$ ，其中 k_i^1 为最低位， k_i^d 为最高位，关键字的每一位数字都可作为一个子关键字。首先将元素序列按照最低的关键字进行排序，然后从低位到高位直到最高位依次进行排序，这样就完成了排序过程。

例如一组元素的关键字序列为（236，128，34，567，321，793，317，106），这组关键字位数最多的是3位，在排序之前，首先将所有元素都转换为3位数字组成的数，即（236，128，034，567，321，793，317，106）。对这组元素进行基数排序需要进行3趟分配和收集，首先需要对该元素序列的关键字的最低位即个位上的数字进行分配和收集，然后对十位数字进行分配和收集，最后是对最高位的数字进行分配和收集。一般情况下，采用链表实现基数排序。

对最低位进行分配和收集的过程如图12-28所示。其中，数组 $f[i]$ 保存第 i 个链表的头指针，数组 $r[i]$ 保存第 i 个链表的尾指针。

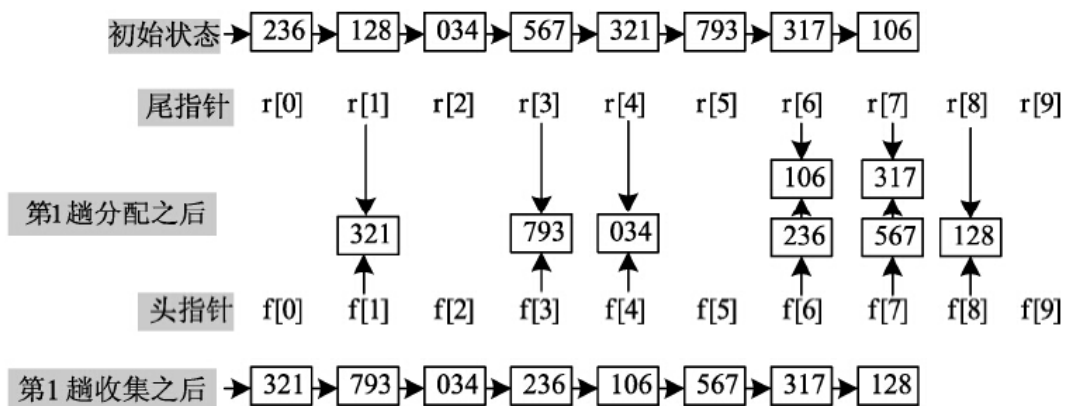


图12-28 第1趟分配和收集过程

对十位数字分配和收集的过程如图12-29所示。

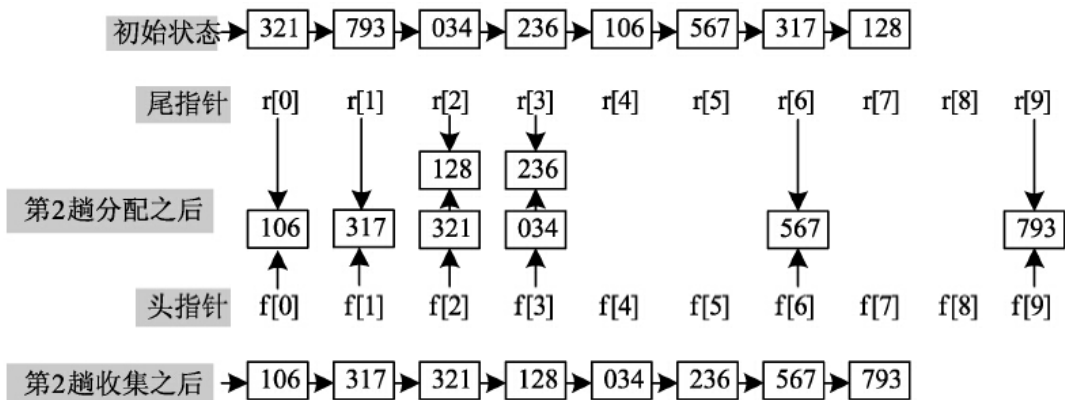


图12-29 第2趟分配和收集过程

对百位数字分配和收集的过程如图12-30所示。

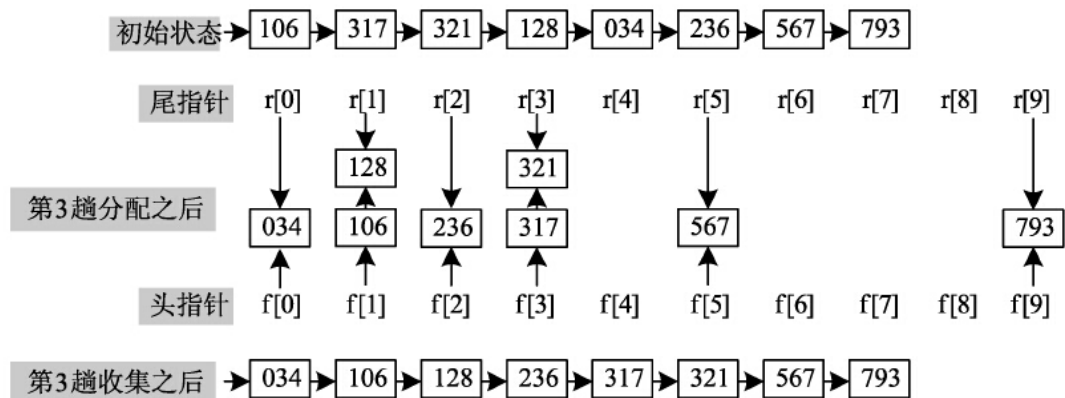


图12-30 第3趟分配和收集过程

由上很容易看出，经过第1趟排序即对个位数字作为关键字进行分配后，关键字被分为10类，个位数字相同的数字被划分为一类，对分配后的元素进行收集之后，得到以个位数字非递减排列的元素。同理，经过第2趟分配和收集后，得到以十位数字非递减排列的元素序列。经过第3趟分配和收集后，得到最终的排序结果。

基数排序的算法主要包括分配和收集，通过静态链作为存储方式。静态链表类型定义描述如下。

```
#define MaxNumKey 6                                /*
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 MaxSize 1000
typedef int KeyType;
typedef struct
{
    KeyType key[MaxNumKey];                          /*
关键字*/
    int next;                                         /*
}SListCell;
静态链表的结点类型*/
typedef struct
{
    SListCell data[MaxSize];                        /*
存储元素，data[0]
为头结点*/
    int keynum;                                       /*
每个元素的当前关键字个数*/
    int length;                                       /*
静态链表的当前长度*/
}SList;
静态链表类型*/
typedef int addr[Radix];                            /*
指针数组类型*/
```

基数排序的分配算法实现如下所示。

```
void Distribute(SListCell data[],int i,addr f,addr r)
/*
为data
中的第i
个关键字key[i]
建立Radix
个子表，使同一子表中元素的key[i]
相同*/
/*f[0..Radix-1]
和r[0..Radix-1]
分别指向各个子表中第一个和最后一个元素*/
```

```

{
    int j,p;
    for (j=0;j<Radix;j++) /*
将各个子表初始化为空表*/
        f[j]=0;
        for (p=data[0].next;p;p=data[p].next)
        {
            j=trans(data[p].key[i]); /*
将对应的关键字字符转化为整数类型*/
            if (!f[j]) /*f[j]
是空表,则f[j]
指示第一个元素*/
                f[j]=p;
            else
                data[r[j]].next=p;
            r[j]=p; /*
将p
所指的结点插入第j
个子表中*/
        }
}

```

其中，数组f[j]和数组r[j]分别存放第j个子表的第一个元素的位置和最后一个元素的位置。基数排序的收集算法实现如下。

```

void Collect(SListCell data[],addr f,addr r)
/*
按key[i]
将f[0..Radix-1]
所指各子表依次链接成一个静态链表*/
{
    int j,t;
    for (j=0;!f[j];j++); /*
找第一个非空子表/
        data[0].next=f[j]; /*data[0].next
指向第一个非空子表中第一个结点*/
        t=r[j];
        while (j<Radix-1)
        {
            for (j=j+1;j<Radix-1&&!f[j];j++); /*
找下一个非空子表*/
            if (f[j]) /*
将非空链表连接在一起*/
            {
                data[t].next=f[j];
                t=r[j];
            }
        }
        data[t].next=0; /*t
指向最后一个非空子表中的最后一个结点*/
}

```

基数排序通过多次调用分配算法和收集算法，从而实现排序，其算法实现如下。

```

void RadixSort(SList *L)
/*
对L
进行基数排序,使得L
成为按关键字非递减的静态链表, L.r[0]
为头结点*/

```

```

{
    int i;
    addr f,r;
    for(i=0;i<(*L).keynum;i++)
        /*
        由低位到高位依次对各关键字进行分配和收集*/
    {
        Distribute((*L).data,i,f,r);
        /*
        第i趟分配*/
        Collect((*L).data,f,r);
        /*
        第i趟收集*/
    }
}

```

容易看出，基数排序需要 $2 \times \text{Radix}$ 个队列指针，分别指向每个队列的队头和队尾。假设待排序的元素为 n 个，每个元素的关键字为 d 个，则基数排序的时间复杂度为 $O(d \times (n + \text{Radix}))$ 。

12.6.2 基数排序应用举例

【例12-7】 编写一个基数排序算法，对给定的一组关键字（236，128，34，567，321，793，317，106）进行排序，要求输出每排序的结果。

【分析】 主要考查基数排序的算法思想。基数排序就是利用多个关键字先进行分配，然后再对每趟排序结果进行收集，多趟分配和收集后得到最终的排序结果。十进制数有0~9共10个数字，利用10个链表分别存放每个关键字各个位为0~9的元素，然后通过收集将每个链表连接在一起，构成一个链表，通过3次分配和收集完成排序。

1. 分配和收集算法

这部分主要包括基数排序的分配、收集，算法实现代码如下。

```

void Distribute(SListCell data[],int i,addr f,addr r)
/*
为data
中的第i
个关键字key[i]
建立Radix
...

```

```

    个子表，使同一子表中元素的key[i]
    相同*/
    /*f[0..Radix-1]
    和r[0..Radix-1]
    分别指向各个子表中第一个和最后一个元素*/
    {
        int j,p;
        for(j=0;j<Radix;j++)
            /*
    将各个子表初始化为空表*/
        {
            f[j]=0;
            for(p=data[0].next;p=p->next)
            {
                j=trans(data[p].key[i]);
                /*
    将对应的关键字字符转化为整数类型*/
                if(!f[j])
                    /*f[j]
    是空表，则f[j]
    指示第一个元素*/
                {
                    f[j]=p;
                    data[r[j]].next=p;
                    r[j]=p;
                    /*
    将p
    所指的结点插入第j
    个子表中*/
                }
            }
        }
    void Collect(SListCell data[],addr f,addr r)
    /*
    按key[i]
    将f[0..Radix-1]
    所指各子表依次链接成一个静态链表*/
    {
        int j,t;
        for(j=0;!f[j];j++);
        /*
    找第一个非空子表*/
        data[0].next=f[j];
        /*data[0].next
    指向第一个非空子表中第一个结点*/
        t=r[j];
        while(j<Radix-1)
        {
            for(j=j+1;j<Radix-1&&!f[j];j++);
            /*
    找下一个非空子表*/
            if(f[j])
                /*
    将非空链表连接在一起*/
            {
                data[t].next=f[j];
                t=r[j];
            }
        }
        data[t].next=0;
        /*t
    指向最后一个非空子表中的最后一个结点*/
    }
    void RadixSort(SList *L)
    /*
    对L
    进行基数排序，使得L
    成为按关键字非递减的静态链表，L.r[0]
    为头结点*/
    {
        int i;
        addr f,r;
        for(i=0;i<(*L).keynum;i++)
            /*
    由低位到高位依次对各关键字进行分配和收集*/
        {
            Distribute((*L).data,i,f,r);
            /*
    第i
    趟分配*/
            Collect((*L).data,f,r);
            /*
    第i
    趟收集*/
            printf("
    第%d
    趟收集后:",i+1);
            PrintList2(*L);
        }
    }

```

2. 静态链表的初始化

这部分主要包括求出关键字最大的元素并通过该元素值得到子关键字的个数
将每个元素的关键字转换为字符类型（不足的位数用字符0补齐）、将每个结点
通过链域链接起来构成一个链表。

静态链表的算法实现如下所示。

```
void InitList(SList *L,DataType a[],int n)
/*
初始化静态链表L*/
{
    char ch[MaxNumKey],ch2[MaxNumKey];
    int i,j,max=a[0].key;
    for(i=1;i<n;i++)
        /*
将最大的关键字存入max*/
        if(max<a[i].key)
            max=a[i].key;
    (*L).keynum=(int)(log10(max))+1;
    /*
求子关键字的个数*/
    (*L).length=n;
    /*
待排序个数*/
    for(i=1;i<=n;i++)
    {
        itoa(a[i-1].key,ch,10);
        /*
将整型转化为字符,
并存入ch*/
        for(j=strlen(ch);j<(*L).keynum;j++)
            /*
如果ch
的长度<max
的位数,
则在ch
前补'0'*/
            {
                strcpy(ch2,"0");
                strcat(ch2,ch);
                strcpy(ch,ch2);
            }
        for(j=0;j<(*L).keynum;j++)
            /*
将每个关键字的各位数字存入key*/
            (*L).data[i].key[j]=ch[(*L).keynum-1-j];
    }
    for(i=0;i<(*L).length;++i)
        /*
初始化静态链表*/
        (*L).data[i].next=i+1;
    (*L).data[(*L).length].next=0;
}
```

3. 测试代码

测试代码如下。

```

#include<stdio.h>
#include<malloc.h>
#include<math.h>
#define MaxNumKey 6 /*
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 MaxSize 1 000
#define N 6
typedef int KeyType; /*
定义关键字类型为字符型*/
typedef struct
{
    KeyType key[MaxNumKey]; /*
关键字*/
    int next; /*
}SListCell;
静态链表的结点类型*/
typedef struct
{
    SListCell data[MaxSize]; /*
存储元素，data[0]
为头结点*/
    int keynum; /*
每个元素的当前关键字个数*/
    int length; /*
静态链表的当前长度*/
}SList; /*
静态链表类型*/
typedef int addr[Radix]; /*
指针数组类型*/
typedef struct
{
    KeyType key; /*
关键字*/
}DataType;
void PrintList(SList L);
void PrintList2(SList L);
void InitList(SList *L,DataType d[],int n);
int trans(char c);
void Distribute(SListCell data[],int i,addr f,addr r);
void Collect(SListCell data[],addr f,addr r);
void RadixSort(SList *L);
int trans(char c)
/*
将字符c
转化为对应的整数*/
{
    return c-'0';
}
void main()
{
    DataType d[N]={268,126,63,730,587,184};
    SList L;
    int *adr;
    InitList(&L,d,N);
    printf("
待排序元素个数是%d
个，关键字个数为%d
个\n",L.length,L.keynum);
    printf("
排序前的元素:\n");
    PrintList2(L);
    printf("
排序前的元素的存放位置:\n");
    PrintList(L);
    RadixSort(&L);
    printf("
排序后元素的存放位置:\n");
    PrintList(L);
}
void PrintList(SList L)
/*
按数组序号形式输出静态链表*/
{

```

```

        int i,j;
        printf("
序号
关键字
地址\n");

        for(i=1;i<=L.length;i++)
        {
            printf("%2d",i);
            for(j=L.keynum-1;j>=0;j--)
                printf("%c",L.data[i].key[j]);
            printf("    %d\n",L.data[i].next);
        }
    }
void PrintList2(SList L)
/*
按链表形式输出静态链表*/
{
    int i=L.data[0].next,j;
    while(i)
    {
        for(j=L.keynum-1;j>=0;j--)
            printf("%c",L.data[i].key[j]);
        printf(" ");
        i=L.data[i].next;
    }
    printf("\n");
}

```

程序运行结果如图12-31所示。

```

D:\零基础学数据结构\例12_7\Debug\例12_7.exe
待排序元素个数是8个，关键字个数为3个
排序前的元素：
236 128 034 567 321 793 317 106
排序前的元素的存放位置：
序号 关键字 地址
1    236    2
2    128    3
3    034    4
4    567    5
5    321    6
6    793    7
7    317    8
8    106    0
第1趟收集后：321 793 034 236 106 567 317 128
第2趟收集后：106 317 321 128 034 236 567 793
第3趟收集后：034 106 128 236 317 321 567 793
排序后元素的存放位置：
序号 关键字 地址
1    236    7
2    128    1
3    034    8
4    567    6
5    321    4
6    793    0
7    317    5
8    106    2
Press any key to continue

```

图12-31 基数排序运行结果示意图

12.7 小结

排序可分为插入排序、选择排序、交换排序、归并排序和基数排序。

直接插入排序算法实现最为简单，时间复杂度在最好、最坏和平均情况下都为 $O(n^2)$ 。

简单选择排序算法的时间复杂度在最好、最坏和平均情况下都是 $O(n^2)$ ，而堆排序的时间复杂度在最好、最坏和平均情况下都是 $O(n\log_2 n)$ 。

冒泡排序的平均时间复杂度为 $O(n^2)$ ，快速排序在最好和平均情况下时间复杂度为 $O(n\log_2 n)$ ，最坏情况下时间复杂度为 $O(n^2)$ 。

归并排序时间复杂度在最好、最坏和平均情况下都为 $O(n\log_2 n)$ 。

基数排序是一种不需要对关键字进行比较的一种排序算法。在任何情况下，基数排序的时间复杂度均为 $O(d(n+rd))$ 。

从稳定性来看，直接插入排序、冒泡排序、归并排序和基数排序属于稳定的排序算法，希尔排序、快速排序、简单选择排序、堆排序

属于不稳定的排序算法。

12.8 习题

一、选择题

1. 若需要在 $O(n \log_2 n)$ 的时间内完成对数组的排序，且要求排序是稳定的，则可选的排序方法是（ ）。

- A. 快速排序
- B. 堆排序
- C. 归并排序
- D. 直接插入排序

2. 下列排序方法中（ ）方法是不稳定的。

- A. 冒泡排序
- B. 选择排序
- C. 堆排序
- D. 直接插入排序

3. 一个序列中有10000个元素，若只想得到其中前10个最小元素，则最好采用（）方法。

A. 快速排序

B. 堆排序

C. 插入排序

D. 归并排序

4. 一组待排序序列为（46，79，56，38，40，84），则利用堆排序的方法建立的初始堆为（）。

A. 79，46，56，38，40，80

B. 84，79，56，38，40，46

C. 84，79，56，46，40，38

D. 84，56，79，40，46，38

5. 快速排序方法在（）情况下最不利于发挥其长处。

A. 要排序的数据量太大

B. 要排序的数据中有多个相同值

C. 要排序的数据已基本有序

D. 要排序的数据个数为奇数

6. 排序时扫描待排序记录序列，顺次比较相邻的两个元素的大小，逆序时就交换位置，这是（）的基本思想。

A. 堆排序

B. 直接插入排序

C. 快速排序

D. 冒泡排序

7. 在任何情况下，时间复杂度均为 $O(n \log n)$ 的不稳定的排序方法是（）。

A. 直接插入

B. 快速排序

C. 堆排序

D. 归并排序

8. 如果将所有中国人按照生日来排序，则使用（）算法最快。

A. 归并排序

B. 希尔排序

C. 快速排序

D. 基数排序

9. 在对 n 个元素的序列进行排序时，堆排序所需要的附加存储空间是（ ）。

A. $O(\log_2 n)$

B. $O(1)$

C. $O(n)$

D. $O(n \log_2 n)$

所采用的排序方法是（ ）。

A. 选择排序

B. 希尔排序

C. 归并排序

D. 快速排序

10. 设有1024个无序的元素，希望用最快的速度挑选出其中前5个最大的元素，最好选用（）。

A. 冒泡排序

B. 选择排序

C. 快速排序

D. 堆排序

二、综合题

1. 写出用直接插入排序将关键字序列{54, 23, 89, 48, 64, 50, 25, 90, 34}排序过程的每一趟结果。

2. 有一关键字序列（265, 301, 751, 129, 937, 863, 742, 694, 076, 438），写出希尔排序的每趟排序结果。（取增量为5, 3, 1）

3. 对关键字序列（72, 87, 61, 23, 94, 16, 05, 58）进行堆排序，使之按关键字递减次序排列（最小堆），请写出排序过程中得到的初始堆和前三趟的序列状态。

三、算法设计题

1. 编写一个算法，将有序表A= (4, 8, 34, 56, 89, 103) 和 B= (23, 45, 78, 90) 合并为一个有序表C。

2. 采用链表作为存储结构，请编写冒泡排序算法，对元素的关键字序列 (25, 67, 21, 53, 60, 103, 12, 76} 进行排序。

3. 利用链表作为存储结构，对关键字序列 (45, 67, 21, 98, 12, 39, 81, 53) 进行选择排序。

4. 利用链表作为存储结构，对关键字序列 (87, 34, 22, 93, 102, 56, 39, 21) 进行插入排序。

5. 采用非递归算法实现快速排序算法，对元素的关键字序列 (34, 92, 23, 12, 60, 103, 2, 56} 进行排序。

第13章 外排序

外排序通常应用于数据量非常大的情况，在排序的过程中，待排序的数据不断地在内存与外存之间交换，将已经排序的放在外存，待排序的数据调入内存。比较常用的外排序方法是多路归并法，根据存储待排序文件的存储介质，可分为磁盘排序和磁带排序。本章主要介绍外存的存取特性、磁盘排序和磁带排序。

本章重点和难点：

- 磁盘和磁带存取的过程
- 磁盘的多路归并排序
- 磁带的多路归并排序

13.1 外存的存取特性

计算机的存储器分为内存储器和外存储器，分别简称为内存和外存。本节主要介绍有关外存信息的存取及相关定义。

内存具有随机存取、存取速度快的优点，但容量小、价格昂贵。外存也称为辅存，其存储容量大且价格便宜，但存取速度慢。外存主要包括磁盘和磁带，磁盘通过随机存取数据，磁带通过顺序存取数据。

1. 磁盘信息的存取

磁盘是一种随机存取的设备，可以直接存取设备上的数据。磁盘由盘片、磁头、盘片主轴、磁头控制器等组成，其中，盘片是用于存储数据的。盘片被划分为多个同心圆，称为磁道，磁道由外向内从0开始顺序编号，所有的数据信息被记录在磁道上。磁盘一般由多个盘片构成，每一个盘片包含两个面，其中，最上面和最下面的外侧的一面不存储信息。例如，一个磁盘由6个盘片组成，则有10个面可保存信息。

所有的盘面的同一磁道构成一个圆柱，称为柱面，位于同一柱面上的磁道由上往下从0开始编号。每个磁道还可以划分为若干个部分，

称为扇区，每个扇区的大小是512字节。扇区所在的磁道地址称为扇区号。

例如，某磁盘有10个有效记录面，记录面上有效记录区域的内径为20cm，外径是30cm，道密度为10道/mm，每个磁道有16个扇区，每个扇区记录512字节，则该磁盘的容量是：记录面数 \times 磁道数 \times 磁道扇区数 \times 扇区的字节数 $=10 \times (30 - 20) / 2 \times 10 \times 10 \times 16 \times 512 = 40960000\text{B} \approx 39\text{MB}$ 。

磁盘的控制器用来控制磁头读写盘片的数据信息，磁盘的磁头可以分为可变和固定两种。其中，固定磁头的每一个磁道上都有磁头，读写信息时，磁头不动，通过移动磁盘来读写信息。可变磁头是可以移动的，通过磁头的移动来读写数据，一个面上有一个磁头。

要存取某一数据信息，首先需要找到数据所在的柱面，移动磁头到所在的柱面即磁道，然后移动磁头到具体的数据存放位置，因此，要存取磁盘上的数据信息所需要的时间由3个部分构成：寻道时间、等待时间和传输时间。其中，寻道时间 t_{seek} 就是读写磁头寻找数据所在磁道并定位的时间，等待时间 t_{w} 就是将磁头移动到数据信息所在的起始位置，传输时间 t_{rw} 就是读或写数据所需要的时间。读写数据的所用时间可表示为 $T = t_{\text{seek}} + t_{\text{w}} + t_{\text{rw}}$ 。

磁盘的旋转速度非常快，约为7200转/分，因此其读写速度也非常快，磁盘旋转一圈所用的时间不会超过10ms，磁盘读写信息所用的时间大部分是在寻道时间上。因此，一般情况下，将数据信息存放在同一磁道或同一柱面上以方便查找，节省寻道时间。

2. 磁带信息的存取

磁带存储器是以磁带作为存储介质，由磁带控制器、磁带驱动器构成的设备。磁带控制器控制磁带驱动器进行读写操作，磁带驱动器是以磁带为存储介质的数字磁性记录装置，主要由读写磁头、磁带传送装置等构成。

磁带作为主要的存储介质，一般磁带长3600英尺、宽0.5英寸。磁带可以分为7道带和9道带，7道带的磁带，每一排可以存储8位即一个字节，其中7位是数据位，1位奇偶校验位。通常情况下，磁带的存储密度是每英寸800位或1600位，磁带的移动磁头速度是每秒200英寸。

磁带是一种顺序读写的设备。所谓顺序读写，就是数据记录按照顺序依次存储在磁带上，假设磁头所在记录是第1条，如果需要对第k条记录进行读写，则必须先将磁头从第1条记录移动到第2条、第3条，一直到第k条记录，然后才能进行读写，这类似于在单链表中对结点的访问。

磁带是一种启停设备，当磁头正在读写磁带上的记录，如果要停止读写，必须经过一个减速的过程，然后才能停止。同样，从磁头静止，到开始读写数据时，磁头是经过一个加速旋转的过程，然后才能匀速旋转达到稳定状态。因此，在磁带的每个相邻的记录之间，需要一个空白区域，这个空白区域就称为间隙。间隙的大小通常为0.25~0.75英寸。磁带的存储结构如图13-1所示。



图13-1 磁带的存储结构

磁带是以块作为存储单元的，这样可以增加磁带空间的利用率。磁头读取一块数据记录所需要的时间由两个部分构成：延迟时间和数据传输时间。延迟时间就是磁头从当前记录移动到需要读写记录所在的起始地址所用的时间 t_w ，数据传输时间就是读写该记录所需的时间 t_{rw} 。可表示为 $T=t_w+t_{rw}$ 。

由于磁带是顺序存取的设备，在数据记录读写的过程中，非常不方便，例如，如果磁头位于磁带的末尾，即最后一条记录上，而需要读写的记录是第1条，则需要将磁头顺序移动到该记录上，这就需要比较长的等待时间。一般情况下，磁带这种顺序存取的设备用在读取信息量比较大，同时不需要频繁移动磁头的情况。

13.2 磁盘排序

在外排序的方法中，归并排序是最为常用的方法。这种方法的基本思想是：先把要排序的文件分段送入内存，并对这些子段调入内存进行排序，再将这些排好序的子段送入外存，称为归并段；然后将这些归并段不断地调入内存进行归并排序，使得归并段不断扩大，直到整个待排序文件有序。本节主要介绍归并排序的基本方法和多路归并排序。

13.2.1 归并排序的基本方法

在外排序中，归并排序的基本思想方法可分为两步：

第一步：将待排序的 n 个记录文件按照内存缓冲区的大小，分割为长度为 m 的若干个子文件，将这些子文件分批地不断调入内存，进行归并排序，生成有序的若干个子文件，并放在外存。将这些有序的子文件称为归并段。

第二步：对已经有序的归并段继续不断在内存和外存互换，从而进行归并排序，直到待排序文件构成一个有序的序列。

例如，某个待排序的文件有1200条记录（ $D_1, D_2, \dots, D_{1200}$ ），内存缓冲区可容纳300条记录，要对该文件进行排序，需要将该文

件等分为4块，每块包含300条记录。并将每块读入内存，利用归并排序法进行排序，可得到4个归并段 R_1 、 R_2 、 R_3 和 R_4 。其中， R_1 包含记录 $D_1 \sim D_{300}$ ， R_2 包含记录 $D_{301} \sim D_{600}$ ， R_3 包含记录 $D_{601} \sim D_{900}$ ， R_4 包含记录 $D_{901} \sim D_{1200}$ 。将这4个归并段送回外存。

将每个归并段3等分，每一部分包含100条记录。同时，将内存缓冲区也分为3个部分，每一个部分可容纳100条记录，前两个部分作为输入缓冲区，后一部分作为输出缓冲区。分别将归并段 R_1 和 R_2 的前100条记录送入输入缓冲区，对这200条记录进行归并排序，当排序完毕后将记录送入输出缓冲区，当输出缓冲区装满100条记录后，就存入到磁盘。当输入缓冲区为空时，将归并段中其他记录送入输入缓冲区，进行归并排序。重复执行以上过程，直到归并段 R_1 和 R_2 有序。这时，该有序记录共有600条。

按照以上方法归并 R_3 和 R_4 ，得到有序的600条记录。然后分别对两个新生成的归并段继续归并，直到待排序文件中所有记录都已经有序，则该文件的1200条记录均为有序序列。整个归并排序的过程如图13-2所示。

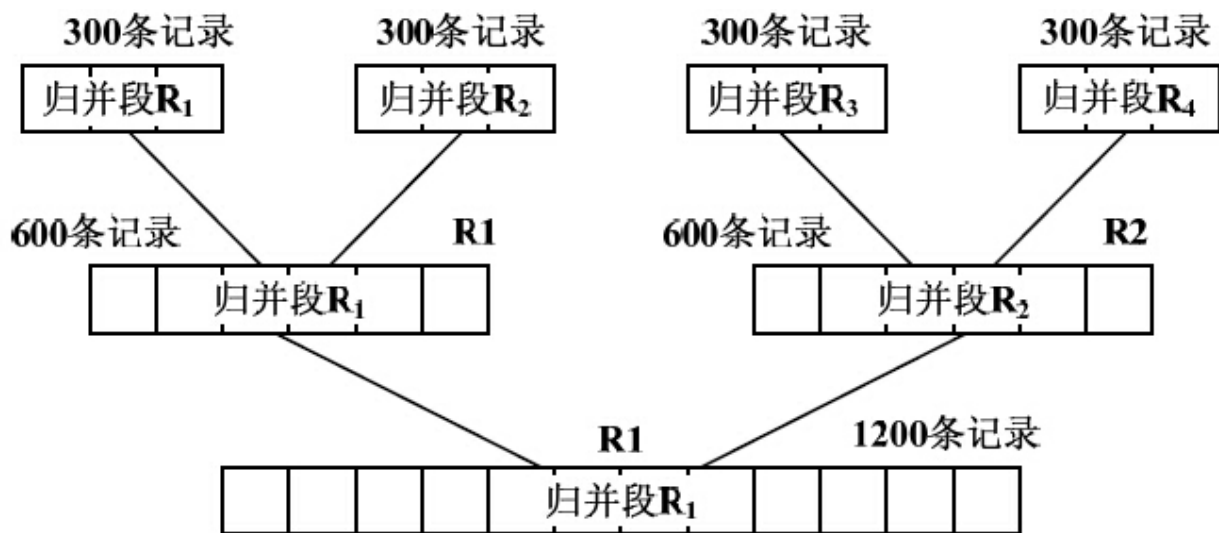


图13-2 归并段的归并排序过程

下面分析以上归并需要的时间。为了表示上的方便，用 t_s 表示最长寻道时间， t_w 表示最长等待时间， t_{rw} 表示读或写100条记录所用的时间， t_{io} 表示读或写100条记录所用的时间即 $t_{io} = t_s + t_w + t_{rw}$ ， t_{is} 表示300条记录内部排序所用的时间， nt_m 表示将 n 条记录从输入缓冲区归并到输出缓冲区花费的时间。

因此，总的归并时间可计算如下：

(1) 读入1200条记录并进行归并，则需要的时间为 $12t_{io} + 4t_{is}$ ；

(2) 将两个归并段即 R_1 和 R_2 归并为 R_1 ， R_3 和 R_4 归并为 R_2 ，则需要花费的时间为 $12t_{io} + 1200t_m$ ；

(3) 然后将 R_1 和 R_2 归并为 R_1 的需要的时间为 $12t_{iO} + 1200t_m$ 。总共需要的时间为 $36t_{iO} + 4t_{iS} + 2400t_m$ 。

由此可以看出，归并排序可以通过减少对记录的扫描次数，从而减少读写的次数及IO次数。这主要通过多路归并排序的方法来减少时间的耗费。

13.2.2 多路归并排序

多路归并可以有效地减少时间的开销。对于上面提到的2路归并，在初始时，如果归并段是 m 个，则将构成的归并树的层数是 $\lceil \log_2 m \rceil + 1$ ，需要进行 $\lceil \log_2 m \rceil$ 趟归并才能完成排序。而对于 k 路归并，就是将 k 个归并段进行归并，则将构成 $\lceil \log_k m \rceil + 1$ 层的归并树，也就是需要进行 $\lceil \log_k m \rceil$ 趟的归并。例如，如果 $k=4$ ，则16个归并段的归并过程如图13-3所示。

进行多路归并带来的问题是，例如对于 k 路归并，在进行内部排序过程中，为了确定要输出的记录，需要在 k 个记录中寻找关键字最小的记录，逐个比较的话需要比较 $k-1$ 次，这样代价就会比较大，为了减少比较的代价，可以利用选择树进行 k 路归并。

选择树是一种完全二叉树，在该完全二叉树中，叶子结点存放各个归并段的当前待排序的元素，非叶子结点则表示对应孩子结点的最

小的一个，从叶子结点开始，依次将当前互为兄弟结点中关键字最小的一个存入双亲结点，则根结点的关键字就是当前归并段中最小的一个关键字，即为将要输出的记录。一个8路归并排序的选择树如图13-4所示。

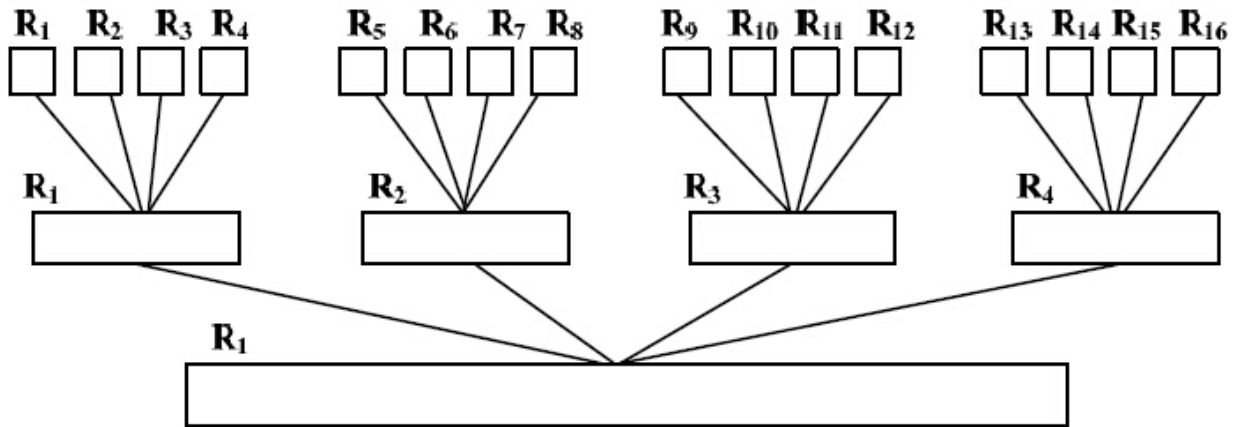


图13-3 具有16个归并段的4路归并过程

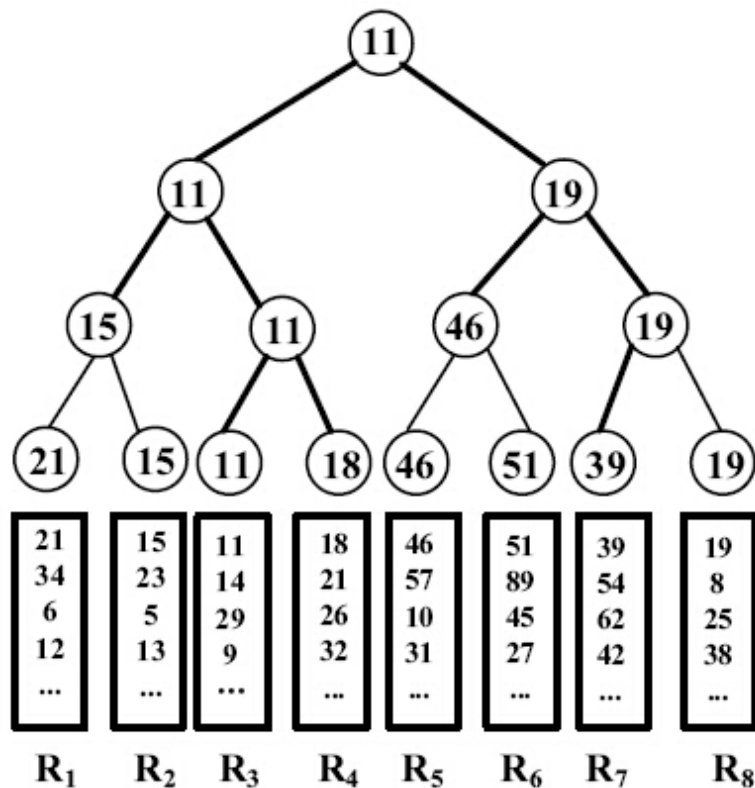


图13-4 8路归并的选择树

这种选择树类似于体育淘汰比赛，每个小组的获胜者参加下一轮的比赛，而根结点就表示整个比赛的冠军，因为根结点永远是代表全胜者，所以将这种选择树称为胜方树。在图13-4中，表示的是一个8路归并的胜方树，根结点表示当前8个归并段中记录最小的关键字是11，是归并段 R_3 中的记录。当该记录输出后，归并段 R_3 中的下一条记录将进入胜方树，继续选择下一个关键字最小的记录。

从图13-4中可以看出，输出关键字最小的记录，只有刚建立胜方树的时候需要进行 $k-1$ 次比较，即非叶子结点的个数次的比较。当一个关键字最小的记录输出后，只需要进行 $\lceil \log_2 m \rceil - 1$ 次的比较，即只需要将上次最小关键字所在的那棵二叉树比较即可。

与胜方树相对应的还有一个称为败方树的二叉树。与胜方树相反，在败方树中，将兄弟结点中关键字较大的一个存入双亲结点中，而将较小的关键字继续进行比较，在根结点上附加一个结点，将最终的优胜者存放在其中。一个8路归并的败方树如图13-5所示。

在图13-5中，每个结点的旁边表示当前层次中较小的关键字，该败方树的最小的关键字是11。

在败方树中，当输出当前归并段中最小的关键字的记录后，后面进行的比较次数减少了许多。只需要将归并段的下一个关键字key存入

叶子结点，并将key与其双亲结点的关键字进行比较，将两者中较大的一个存入双亲结点，较小的一个继续与上一层的双亲结点进行比较。重复执行以上操作，直到根结点。因此，当关键字11的记录输出后，败方树的修改如图13-6所示。

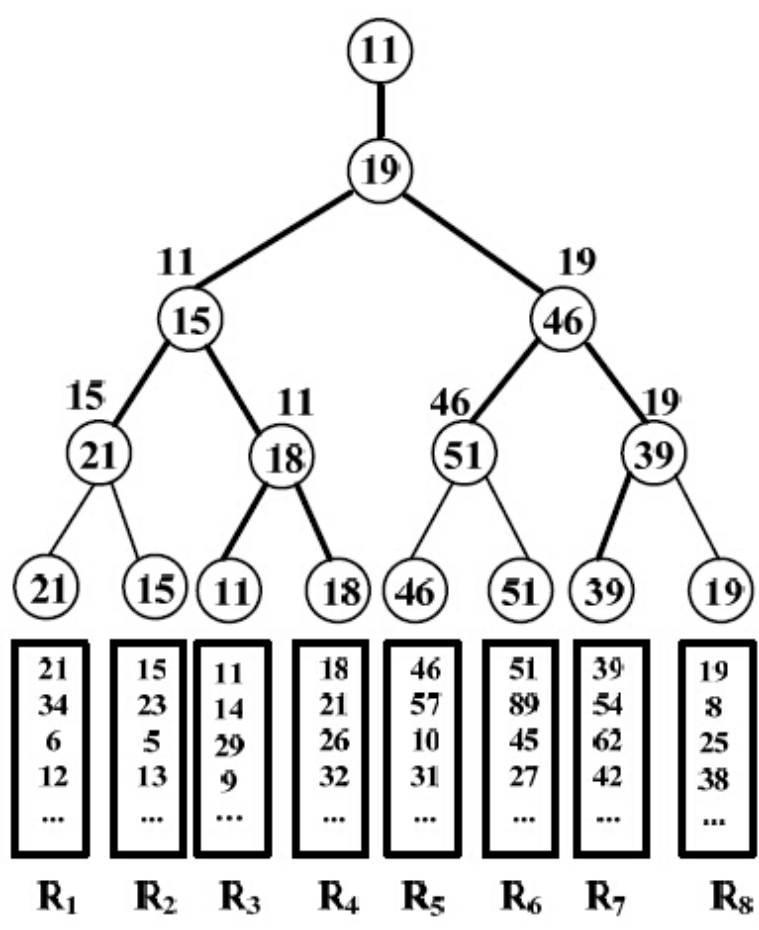


图13-5 8路归并的败方树

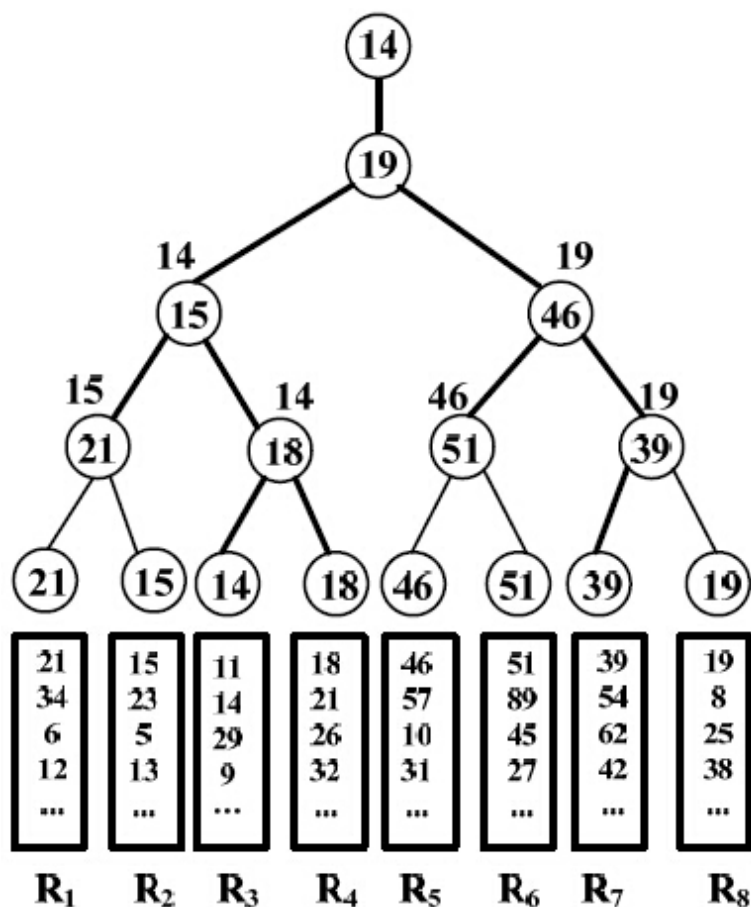


图13-6 8路归并的败方树的修改过程

容易看出，败方树的比较次数要比胜方树的次数要少得多，它减少了与兄弟结点的比较。但是，在进行内排序过程中，如果采用败方树进行内排序，当 k 增大时，内存的缓冲区也要相应的增加，但是内存的缓冲区是固定的，在内存排序时，则需要将内存交换的记录块的长度减小。这样就增加了内存与外存之间交换的次数，这样会造成时间的耗费过大，因此， k 值不能任意增大。

13.3 磁带排序

磁带排序与磁盘排序类似，它也需要先将待排序文件进行内排序，产生出归并段，将其存放在磁带上，然后将这些归并段调入内存，反复归并，直到磁带中所有的记录都排序完毕。在磁带排序过程中，最为常用的方法是非平衡归并排序。

13.3.1 2路归并排序

磁带的外排序与磁盘的外排序类似，都是对待排序的 n 条记录分批不断调入内存，进行归并排序后存入磁带，这样不断地在内存与外存之间交换，使归并段不断增大，最终完成排序。

例如，某个待排序的文件有1200条记录（ $D_1, D_2, \dots, D_{1200}$ ），能够使用的磁带驱动器（磁带机）有4台，分别是 T_1 、 T_2 、 T_3 和 T_4 。内存缓冲存储器可存储300条记录，因此将待排序的1200条记录分为4块，每块包含300条记录。其中，记录 $D_1 \sim D_{300}$ 属于第一块，记录 $D_{301} \sim D_{600}$ 属于第二块，记录 $D_{601} \sim D_{900}$ 属于第三块，记录 $D_{901} \sim D_{1200}$ 属于第四块。

磁带的外排序步骤如下：

(1) 分别将第1、2、3和4块调入内存，进行内排序，生成归并段 R_1 、 R_2 、 R_3 和 R_4 。然后将这4个归并段存储在磁带机 T_1 和 T_2 上。得到归并段如图13-7所示。

(2) 将归并段 R_1 和 R_2 调入内存进行归并排序，将 R_3 和 R_4 调入内存进行归并排序。把每次归并产生的归并段分别存储在磁带 T_3 和 T_4 上，最终产生新的归并段 R_1 和 R_2 ， R_1 和 R_2 分别包含600条记录，并存储在磁带 T_3 和 T_4 上。如图13-8所示。



图13-7 初次归并得到的4个归并段

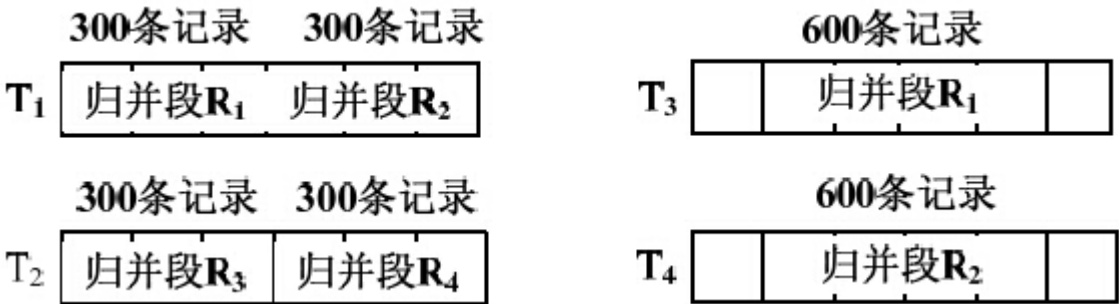


图13-8 将归并段 R_1 和 R_2 、 R_3 和 R_4 归并后得到的新的归并段

(3) 将归并段 R_1 和 R_2 按照步骤 (2) 进行归并后得到如13-9所示的归并段 R_1 并存储在磁带机 T_1 上。

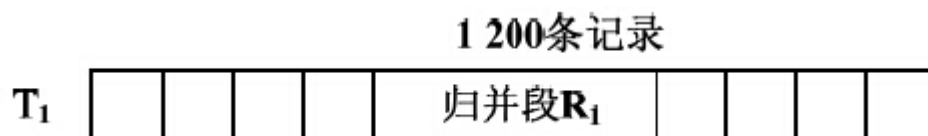


图13-9 最终归并得到的1200条有序记录

不难看出，磁带的归并排序与磁盘的归并排序的思想一样，其排序所用的时间也主要是在于对记录的扫描次数，归并过程中主要时间的耗费在寻找记录上，而为了避免这种耗费，可以采用多路归并，但是多路归并是以增加磁带机的数量为代价的。对于 k 路归并，需要 $k+1$ 台磁带机，其中 k 台磁带机用于存放输入数据，剩下的1台用于存储输出数据。而如果这样做，在对输出的数据排序时，还需要进行扫描，因此，可以使用 $2k$ 台磁带机来避免这种情况，其中， k 台磁带机用于存储输入的数据，另外 k 台磁带机用于存储输出数据。在排序过程中，这些磁带机来回交换，就避免了对磁带记录的扫描。

13.3.2 多路非平衡归并排序

非平衡归并排序是为了减少磁带机的使用，并能减少排序过程中的扫描次数。其主要做法是，不同的磁带机上输入的归并段个数不同，通过对归并段的非均匀分配，从而减少磁盘机的使用。

下面介绍一种利用 $k+1$ 台磁带机实现 k 路归并排序的做法。初始时，归并段不均匀第分布在前 k 个磁带机上，第 $k+1$ 台磁带机用于输出

存储，初始时空。每一趟归并，只是部分的记录参加归并，在每一趟归并完成之后，存放记录最少的磁带机变成空带，将作为下一趟的输出使用。

下面通过利用4个磁带机实现3路归并的情况。例如，要对14个归并段进行3路归并排序，初始时，归并段不均衡地分布在 T_1 、 T_2 和 T_3 上， T_4 作为输出磁带机。

第一步：将 T_1 、 T_2 和 T_3 中的 R_1 、 R_2 、 R_3 归并为新的归并段 R_1 ，将 R_4 、 R_5 、 R_6 归并为新的归并段 R_2 ，将 R_7 、 R_8 、 R_9 归并为新的归并段 R_3 ，然后将 R_1 、 R_2 、 R_3 存储在磁带机 T_4 中，此时， T_3 为空。第一步归并后的情况如图13-11所示。

第二步：将 T_1 、 T_2 和 T_4 中的 R_{10} 、 R_{11} 、 R_1 归并为新的归并段 R_1 ，将 R_{12} 、 R_{13} 、 R_2 归并为新的归并段 R_2 ，然后将 R_1 、 R_2 存储在磁带机 T_3 中，此时 T_2 为空，第二步归并后的情况如图13-12所示。

第三步：将 T_1 、 T_3 和 T_4 中的 R_{14} 、 R_1 、 R_3 归并为新的归并段 R_1 ，并将 R_1 存储在 T_2 中，此时 T_4 为空，则第三步归并后的情况如图13-13所示。

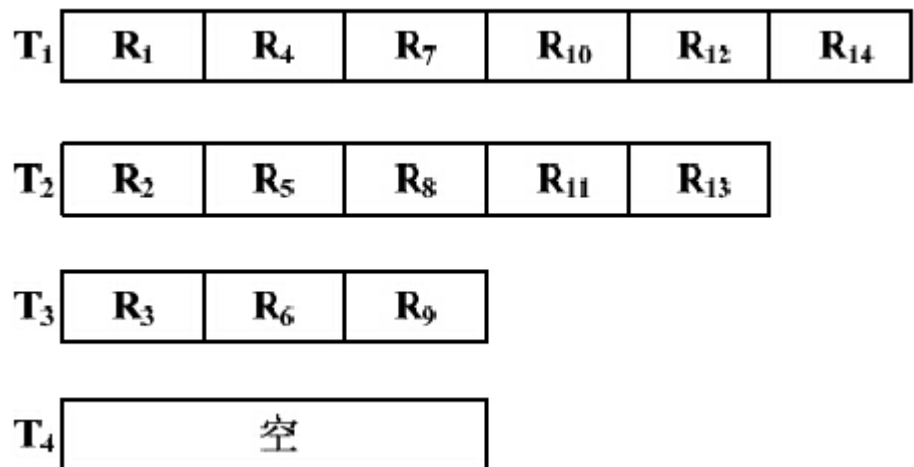


图13-10 待归并排序段初始状态

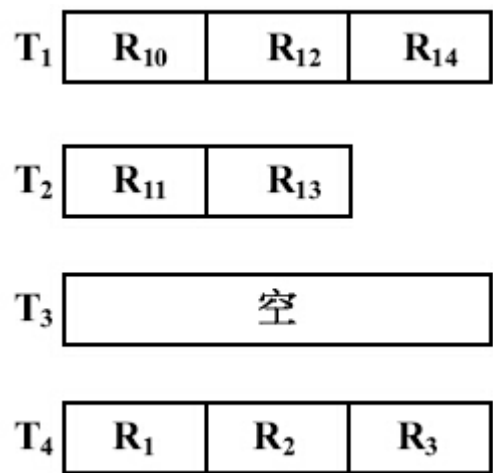


图13-11 第一步归并后

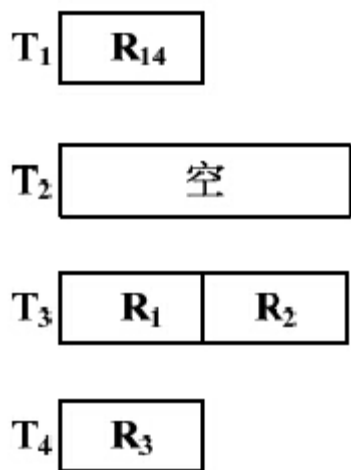


图13-12 第二步归并后

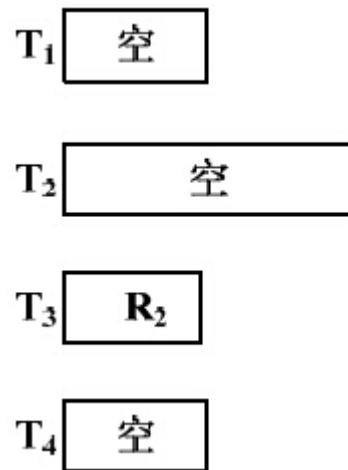


图13-13 第三步归并后

第四步，只剩下一个归并段 R_2 ，存储在 T_3 中， R_2 就是排序的结果。磁带的多路非平衡归并排序可以既节省磁带机的使用，也避免过多地在磁带中寻找记录。

13.4 小结

在对大量的数据记录进行排序时，需要进行外排序。外排序通过将待排序的记录选择一部分调入内存，在内存中对这一部分数据记录排序，然后将排序结果存储在外存中。通过以上反复地在内存和外存进行调换和排序的过程称为外排序。

外排序的主要方法是归并排序。在外排序中，归并排序可分为两个阶段进行。第一阶段：将待排序的 n 个记录文件按照内存缓冲区的大小，分割为长度为 m 的若干个子文件，将这些子文件分批地不断调入内存，进行归并排序，生成有序的若干个子文件，并放在外存。将这些有序的子文件称为归并段。第二阶段：对已经有序的归并段继续不断在内存和外存互换，从而进行归并排序，直到待排序文件构成一个有序的序列。

磁带排序与磁盘排序类似，最为常用的方法是非平衡多路归并排序。非平衡多路归并排序是为了减少磁带机的使用，并能减少排序过程中的扫描次数。其主要做法是，不同的磁带机上输入的归并段个数不同，通过对归并段的非均匀分配，从而减少磁盘机的使用次数。每一趟归并，只是部分的记录参加归并，在每一趟归并完成之后，存放记录最少的磁带机变成空带，将作为下一趟的输出使用。

参考文献

- [1]严蔚敏. 数据结构[M]. 北京: 清华大学出版社, 2001.
- [2]耿国华. 数据结构[M]. 北京: 高等教育出版社, 2005.
- [3]陈锐. 零基础学数据结构[M]. 北京: 机械工业出版社, 2010.
- [4]徐塞红. 数据结构考研辅导[M]. 北京: 北京邮电大学出版社, 2002.
- [5]陈明. 实用数据结构[M]. 2版. 北京: 清华大学出版社, 2010.
- [6]Robert Sedgewick. 算法: C语言实现(第1~4部分)[M]. 霍红卫, 译. 北京: 机械工业出版社, 2009.
- [7]吴仁群. 数据结构简明教程[M]. 北京: 机械工业出版社, 2011.
- [8]朱站立. 数据结构[M]. 西安: 西安电子科技大学出版社, 2003.
- [9]陈锐. C语言从入门到精通[M]. 北京: 电子工业出版社, 2010.
- [10]冼镜光. C语言名题百则[M]. 北京: 机械工业出版社, 2005.

[11] 夏宽理. C程序设计实例详解[M]. 上海: 复旦大学出版社, 1996.

[12] 李春葆, 曾慧, 张植民. 数据结构程序设计题典[M]. 北京: 清华大学出版社, 2002.

[13] 杨明, 杨萍. 研究生入学考试要点、真题解析与模拟考卷[M]. 北京: 电子工业出版社, 2003.

[14] 唐发根. 数据结构[M]. 2版. 北京: 科学出版社, 2004.

[15] 杨峰. 妙趣横生的算法[M]. 北京: 清华大学出版社, 2010.

[16] Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed. 数据结构(C语言版)[M]. 李建中, 张岩, 李治军, 译. 北京: 机械工业出版社, 2006.

[17] 陈锐. C语言入门与提高[M]. 北京: 北京希望电子出版社, 2012.

[18] 陈锐. C/C++常用函数与算法速查手册[M]. 北京: 中国铁道出版社, 2012.

[19] 陈守礼, 胡潇琨, 李玲. 算法与数据结构考研试题精析[M]. 北京: 机械工业出版社, 2007.

[20] 李春葆, 尹为民, 蒋晶珏. 数据结构联考辅导教程[M]. 北京: 清华大学出版社, 2011.

[21] Cormen T H. 算法导论 (原书第2版) [M]. 潘金贵, 译. 北京: 机械工业出版社, 2006.

[22] Donald E Knuth. 计算机程序设计艺术卷1: 基本算法 (英文版第3版) [M]. 北京: 人民邮电出版社, 2010.

[23] 朱站立, 张选平. 数据结构学习指导与典型题解[M]. 西安: 西安交通大学出版社, 2002.

[24] 周伟, 刘泱, 王征勇. 2013年计算机专业基础综合历年统考真题及思路分析[M]. 北京: 机械工业出版社, 2012.

光盘内容

光盘下载地址：<http://pan.baidu.com/s/1ntyqEz3>