应用型人才培养“十三五”精品规划教材

数 据 结 构

（C 语言版）

主　编　黄　翔　李梁奇
副主编　管侯斌　欧静敏
田卫红

西北工业大学出版社
西　安

【内容简介】 数据结构是计算机程序设计的重要理论技术基础课程，本书主要介绍了数据结构的基本概念、线性表的常用算法、二叉树和图的概念及常用算法等，并按照高等教育的培养要求，要求学生在具备数据结构的基础知识和常用算法的基础上，重点掌握算法的项目化应用技能。

本书可作为高等院校计算机类专业的教材，也可供编程爱好者阅读、参考。

图书在版编目（CIP）数据

数据结构 : C语言版 / 黄翘，李梁奇主编. — 西安：西北工业大学出版社，2019.8（2021.8重印）

ISBN 978-7-5612-6552-9

Ⅰ. ①数… Ⅱ. ①黄… ②李… Ⅲ. ①数据结构－高等学校－教材②C语言－程序设计－高等学校－教材 Ⅳ. ①TP311.12②TP312.8

中国版本图书馆CIP数据核字(2019)第182764号

SHUJU JIEGOU(C YUYAN BAN)

数据结构（C语言版）

责任编辑：吴玉梅　　**策划编辑**：华一瑾

责任校对：华一瑾　　**装帧设计**：杨晓敏

出版发行：西北工业大学出版社

通信地址：西安市友谊西路127号　　邮编：710072

电　　话：(029)88491757，88493844

网　　址：www.nwpup.com

印 刷 者：长沙长大成彩印有限公司

开　　本：710 mm×1 000 mm　　1/16

印　　张：17.25

字　　数：358千字

版　　次：2019年8月第1版　2021年8月第2次印刷

定　　价：48.00元

前言

数据结构是计算机程序设计的重要理论技术基础，它不仅是计算机学科的核心课程，而且已成为其他理工专业的热门选修课。通过数据结构课程的学习，学生可以利用数据结构的基本分析方法来提高编写程序的能力和应用计算机解决实际问题的能力。

数据结构是一门“教师难讲，学生难学”的课程。对于学生的教学，我们以“应用为主、够用就好”为原则，从实用出发，将数据结构的知识进行取舍，精心编写了本书，并且尽可能地细化了每个知识点。书中大部分操作都是以短小代码的形式编程，以便读者能够掌握其中的知识点，每章最后都有相关的项目训练，以满足高层次学生的需求。

本书分为 8 章，第 1 章绪论介绍了数据结构的基本概念，并对算法、算法分析作了简要说明；第 2 章到第 4 章介绍了线性表、特殊线性表、广义线性表等线性结构的基本定义及其常用算法的实现和基本应用；第 5 章和第 6 章介绍了非线性结构的二叉树和图，包括其逻辑特征、常用算法的实现和基本应用；第 7 章介绍了查找技术，并进行了简单的时间和空间的效率分析；第 8 章介绍了三大排序算法的概述、基本原理和算法实现，并进行了时间复杂度的分析。

本书由黄翘和李梁奇担任主编，由管侯斌、欧静敏和田卫红担任副主编。

编写本书曾参阅了相关文献资料，在此，谨向其作者深表谢意。

由于笔者水平有限，书中难免存在不妥、疏漏之处，敬请广大读者批评指正。

编 者

2019 年 3 月

目录

第1章 绪 论

计算机在发展初期，其应用范围是数值计算，所处理的数据都是整型、实型、布尔型等简单数据，从而，以此为加工对象的程序设计称为数值型程序设计。后来，随着电子技术的发展，计算机逐渐进入到商业、制造业等其他领域，并广泛地应用于数据处理和过程控制。与此相应，计算机能处理的数据也不再是简单的数值，而是字符串、图形、图像、语音等复杂的数据。这些复杂的数据不仅量大，而且具有一定的结构。例如一幅图像是一个由简单数值组成的矩阵，一个图形中的几何坐标可以组成表。此外，语言编译过程中所使用的栈、符号表和语法树，操作系统用到的队列、磁盘目录树等，都是有结构的数据。数据结构所研究的就是这些有结构的数据，因此，数据结构的知识不论对研制系统软件还是开发应用软件都是非常重要的。它是学习软件知识和提高软件设计水平的重要基础。

数据结构是一门研究数据表示和数据处理的科学。数据是计算机化的信息，它是计算机可以直接处理的最基本和最重要的对象。无论是进行科学计算或数据处理、过程控制及对文件的存储和检索及数据库技术等计算机应用，都是对数据进行加工处理的过程。因此，要设计出一个结构好而且效率高的程序，必须研究数据的特性和数据间的相互关系及其对应的存储表示，并利用这些特性和关系设计出相应的算法和程序。

知识目标

- 掌握数据结构的基本概念和术语的含义。
- 理解数据的类型及其分类原理。

- 掌握算法的定义、特性、数据结构与算法的关系。
- 理解算法的性能与度量，时间、空间复杂度的含义及作用。

能力目标

- 能应用数据的特点，解决实际问题。
- 能应用算法分析，解决实际问题。

1.1 数据结构的概念

数据结构是计算机科学与技术专业的专业基础课，是一门重要的核心课程。所有的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对众多复杂的课题的。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的有关知识。扎实打好数据结构这门课程的基础，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、软件工程以及人工智能等都是十分有益的。

1.1.1 为什么要学习数据结构

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，一般需要经过下列几个步骤：首先要从该具体问题抽象出一个适当的数学模型，然后设计或选择一个解此数学模型的算法，最后编出程序进行调试、测试，直至得到最终的解答。例如，求解梁架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，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。下面所列举的就是属于这一类的具体问题。

【例 1-1】学生信息检索系统。当我们需要查找某个学生的相关信息，或者想查询某个专业或年级的学生的有关情况的时候，只要我们建立了相关的数据结构，按照某种算法编写了相关程序，就可以实现计算机自动检索。由此，可以在学

生信息检索系统中建立一张按学号顺序排列的学生信息表和分别按姓名、专业、年级顺序排列的索引表，如图 1-1 所示。由这 4 张表构成的文件便是学生信息检索的数学模型，计算机的主要操作便是按照某个特定要求（如给定姓名）对学生信息文件进行查询。

学号	姓名	性别	专业	年级
980001	崔 × 志	男	计算机科学与技术	2000 级
980002	刘 × 芳	女	信息与计算科学	2000 级
990301	李 ×	女	数学与应用数学	2001 级
990302	张 × 会	男	信息与计算科学	2001 级
990303	贾 × 国	男	计算机科学与技术	2002 级
000801	陆 × 颖	女	计算机科学与技术	2002 级
000802	石 × 利	男	数学与应用数学	2002 级
000803	崔 × 靖	男	信息与计算科学	2002 级
010601	刘 × 芳	女	计算机科学与技术	2003 级
010602	史 × 斌	男	数学与应用数学	2003 级

(a) 学生信息表

崔 × 靖	8
崔 × 志	1
李 ×	3
刘 × 芳	2，9
陆 × 颖	6
贾 × 国	5
石 × 利	7
史 × 斌	10
张 × 会	4

(b) 姓名索引表

计算机科学与技术	1，5，6，9
信息与计算科学	2，4，8
数学与应用数学	3，7，10

(c) 专业索引表

2000 级	1，2
2001 级	3，4
2002 级	5，6，7，8
2003 级	9，10

(d) 年级索引表

图 1-1　学生信息查询系统中的数据结构

诸如此类的还有电话自动查号系统、考试查分系统、仓库库存管理系统等。在这类文档管理的数学模型中，计算机处理的对象之间通常存在着的是一种简单的线性关系，这类数学模型可称为线性的数据结构。

【例 1-2】八皇后问题。在八皇后问题中，处理过程不是根据某种确定的计算法则，而是利用试探和回溯的探索技术求解。为了求得合理布局，在计算机中要

存储布局的当前状态。从最初的布局状态开始，一步步地进行试探，每试探一步形成一个新的状态，整个试探过程形成了一棵隐含的状态树，如图 1-2 所示（为了描述方便，将八皇后问题简化为四皇后问题）。回溯法求解过程实质上就是一个遍历状态树的过程。在这个问题中所出现的树也是一种数据结构，它可以应用在许多非数值计算的问题中。

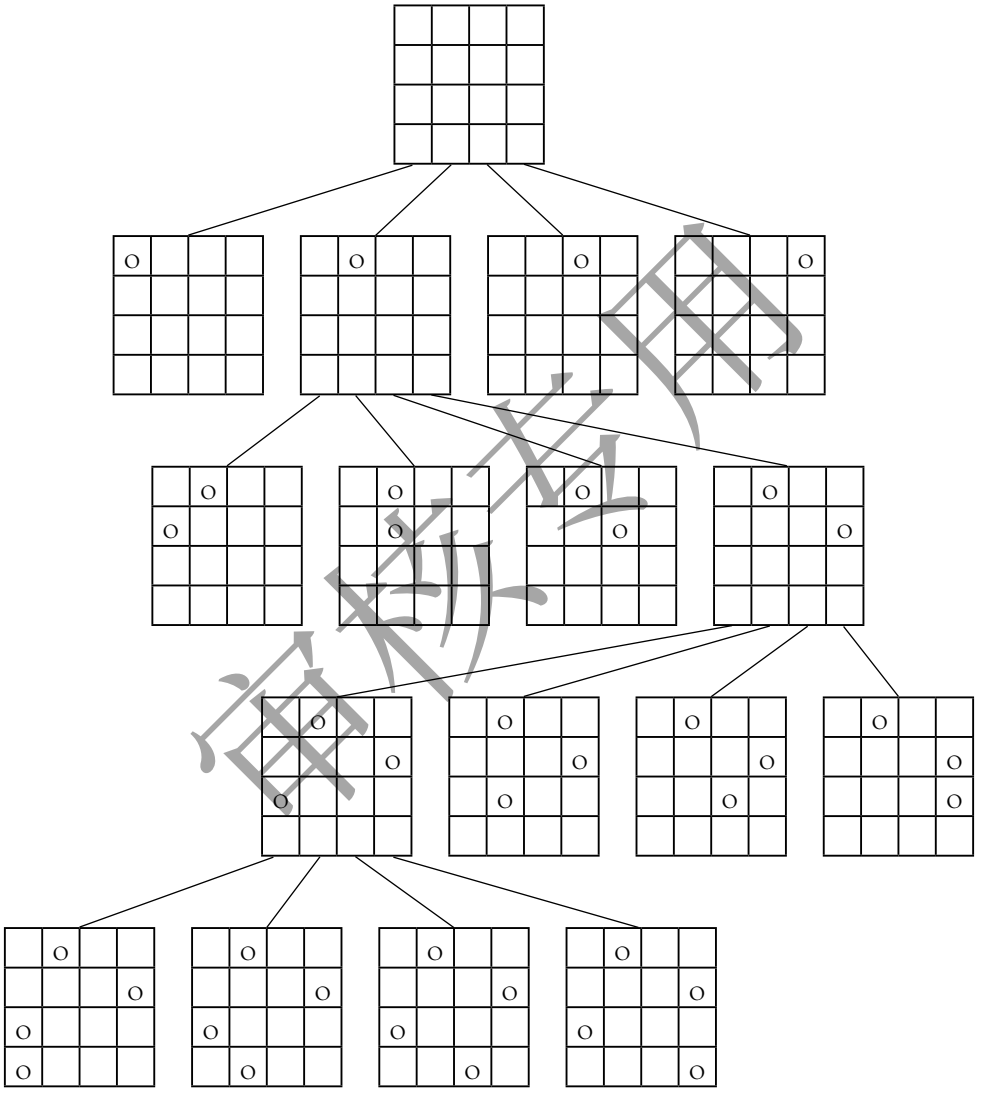


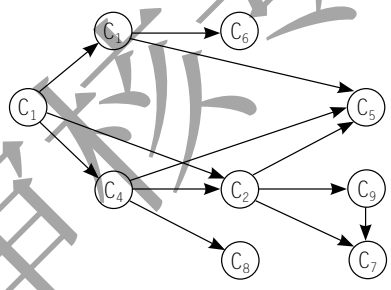
图 1-2　四皇后问题中隐含的状态树

【例 1-3】教学计划编排问题。一个教学计划包含许多课程，在教学计划包含的许多课程之间，有些必须按规定的先后次序进行，有些则没有次序要求。即

有些课程之间有先修和后续的关系，有些课程可以任意安排次序。这种各个课程之间的次序关系可用一个称作图的数据结构来表示，如图 1-3 所示。有向图中的每个顶点表示一门课程，如果从顶点 v_i 到 v_j 之间存在有向边 $<v_i, v_j>$，则表示课程 C_i 必须先于课程 C_j 进行。

课程编号	课程名称	先修课程
C_1	计算机导论	无
C_2	数据结构	C_1，C_4
C_3	汇编语言	C_1
C_4	C 程序设计语言	C_1
C_5	计算机图形学	C_2，C_3，C_4
C_6	接口技术	C_3
C_7	数据库原理	C_2，C_9
C_8	编译原理	C_4
C_9	操作系统	C_2

(a) 计算机专业的课程设置



(b) 表示课程之间优先关系的有向图

图 1-3 教学计划编排问题的数据结构

由以上三个例子可见，描述这类非数值计算问题的数学模型不再是数学方程，而是诸如表、树、图之类的数据结构。因此，可以说数据结构课程主要是研究非数值计算的程序设计问题中所出现的计算机操作对象以及它们之间的关系和操作的学科。

学习数据结构的目的是为了了解计算机处理对象的特性，是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。

1.1.2 有关概念和术语

在系统地学习数据结构知识之前，我们有必要先对一些基本概念和术语赋予确切的含义。

1. 数据（Data）

数据是信息的载体，它能够被计算机识别、存储和加工处理。它是计算机程序加工的原料。应用程序处理各种各样的数据。计算机科学中，所谓数据就是计算机加工处理的对象，它可以是数值数据，也可以是非数值数据。数值数据是一些整数、实数或复数，主要用于工程计算、科学计算和商务处理等；非数值数据包括字符、文字、图形、图像和语音等。

2. 数据项（Data Item）

数据项也称项或字段，是具有独立含义的标识单位，是数据不可分割的最小单位，如图 1-1(a) 中学生信息表的“学号”“姓名”“年级”等。数据项有名和值之分，数据项名是一个数据项的标识，用变量定义，而数据项值是它的一个可能取值，如图 1-1(a) 中学生信息表的“980001”是数据项“学号”的一个取值。数据项具有一定的类型，依数据项的取值类型而定。

3. 数据元素（Data Element）

数据元素是数据的基本单位。在不同的条件下，数据元素又可称为元素、节点、顶点、记录等。例如，学生信息检索系统中学生信息表中的一个记录、八皇后问题中状态树的一个状态、教学计划编排问题中的一个顶点等，都被称为一个数据元素。

有时，一个数据元素可由若干个数据项组成。例如，学籍管理系统中学生信息表的每一个数据元素就是一个学生记录。它包括学生的学号、姓名、性别、籍贯、出生年月和成绩等数据项。这些数据项可以分为两种：一种叫作初等项，如学生的性别、籍贯等，这些数据项是在数据处理时不能再分割的最小单位；另一种叫作组合项，如学生的成绩，它可以再划分为数学、物理、化学等更小的项。通常，在解决实际应用问题时是把每个学生记录当作一个基本单位进行访问和处理的。

4. 数据对象(Data Object)或数据元素类(Data Element Class)

数据对象或数据元素类是具有相同性质的数据元素的集合。在某个具体问题中，数据元素都具有相同的性质（元素值不一定相等），属于同一数据对象（数

据元素类），数据元素是数据元素类的一个实例。例如，在交通咨询系统的交通网中，所有的顶点是一个数据元素类，顶点 A 和顶点 B 各自代表一个城市，是该数据元素类中的两个实例，其数据元素的值分别为 A 和 B。

5. 数据结构（Data Structure）

数据结构是指互相之间存在着一种或多种关系的数据元素的集合。在任何问题中，数据元素之间都不会是孤立的，在它们之间都存在着这样或那样的关系，这种数据元素之间的关系称为结构。根据数据元素间关系的不同特性，通常有下列四类基本的结构，如图 1-4 所示为表示这四类基本结构的示意图。

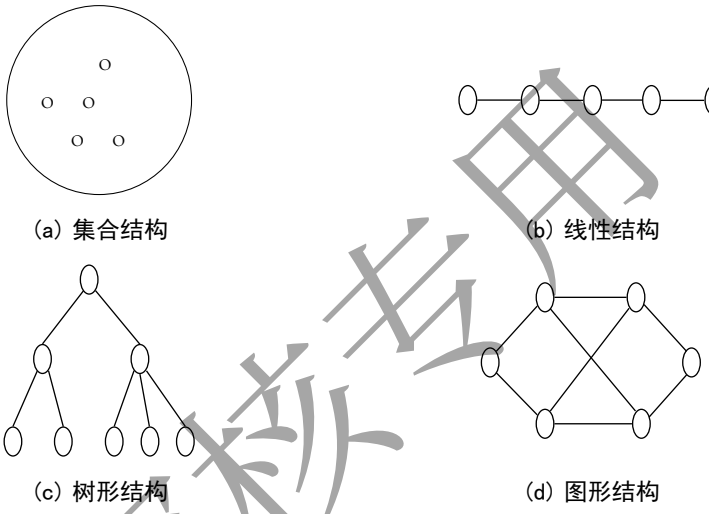


图 1-4 四类基本结构的示意图

（1）集合结构。在集合结构中，数据元素间的关系是“属于同一个集合”。集合是元素关系极为松散的一种结构。

（2）线性结构。该结构的数据元素之间存在着一对一的关系。

（3）树形结构。该结构的数据元素之间存在着一对多的关系。

（4）图形结构。该结构的数据元素之间存在着多对多的关系，图形结构也称作网状结构。

由于集合是数据元素之间关系极为松散的一种结构，因此也可用其他结构来表示它。

从上面所介绍的数据结构的概念中可以知道，一个数据结构有两个要素：一个是数据元素的集合，另一个是关系的集合。在形式上，数据结构通常可以采用一个二元组来表示。因此，数据结构的形式可定义为

$$\text{Data\_Structure} = (D, R) \qquad (1\text{-}1)$$

式中，D 是数据元素的有限集；R 是 D 关系上的有限集。

数据结构包括数据的逻辑结构和数据的物理结构。数据的逻辑结构可以看作是从具体问题抽象出来的数学模型，它与数据的存储无关。我们研究数据结构的目的是为了在计算机中实现对它的操作，为此还需要研究如何在计算机中表示一个数据结构。数据结构在计算机中的标识（又称映像）称为数据的物理结构，或称存储结构。它所研究的是数据结构在计算机中的实现方法，包括数据结构中元素的表示及元素间关系的表示。

数据的存储结构可采用顺序存储或链式存储的方法。

顺序存储方法是把逻辑上相邻的元素存储在物理位置相邻的存储单元中，由此得到的存储表示称为顺序存储结构。顺序存储结构是一种最基本的存储表示方法，通常借助于程序设计语言中的数组来实现。

链式存储方法对逻辑上相邻的元素不要求其物理位置相邻，元素间的逻辑关系通过附设的指针字段来表示，由此得到的存储表示称为链式存储结构，链式存储结构通常借助于程序设计语言中的指针来实现。

除了通常采用的顺序存储方法和链式存储方法外，有时为了查找的方便还可采用索引存储方法和散列存储方法。

1.1.3 数据结构课程的内容

数据结构是介于数学、计算机硬件和软件之间的一门计算机科学与技术专业的核心课程，是高级程序设计语言、编译原理、操作系统、数据库以及人工智能等课程的基础。同时，数据结构技术也广泛应用于信息科学、系统工程、应用数学以及各种工程技术领域。

数据结构课程集中讨论软件开发过程中的设计阶段、同时涉及编码和分析阶段的若干基本问题。此外，为了构造出好的数据结构及其实现，还需考虑数据结构及其实现的评价与选择。因此，数据结构的内容包括三个层次的五个“要素”，见表 1-1。

表 1-1 数据结构课程内容体系

	方　面	
层　次	数据表示	数据处理
抽象	逻辑结构	基本运算
实现	存储结构	算法
评价	不同数据结构的比较及算法分析	

数据结构的核心技术是分解与抽象。通过对问题的抽象，舍弃数据元素的具体内容，就得到逻辑结构。类似地，通过分解将处理要求划分成各种功能，再通过抽象舍弃实现细节，就得到运算的定义。上述两方面的结合使我们将问题变换为数据结构。这是一个从具体（即具体问题）到抽象（即数据结构）的过程。然后，通过增加对实现细节的考虑进一步得到存储结构和实现运算，从而完成设计任务。这是一个从抽象（即数据结构）到具体（即具体实现）的过程。熟练地掌握这两个过程是数据结构课程在专业技能培养方面的基本目标。

数据结构作为一门独立的课程在国外是从1968年才开始的，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设立相关课程，但当时的课程名称并不叫数据结构。1968年美国唐·欧·克努特教授开创了数据结构的最初体系，他所著的《计算机程序设计技巧》第一卷《基本算法》是第一本较系统地阐述数据的逻辑结构和存储结构及其操作的著作。从20世纪60年代末到20世纪70年代初，出现了大型程序，且软件也相对独立。从此，结构程序设计成为程序设计方法学的主要内容，人们也越来越重视数据结构。从20世纪70年代中期到20世纪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。

1.2 抽象数据类型

对于一个复杂问题，往往会涉及许许多多的因素。为了使问题得到简化，以便建立相应的数学模型进行深入研究，并进而加以解决，我们通常采用“抽象”这一思想方法，即抽取反映问题本质的东西，舍去其非本质的细节。

在计算机软件的发展过程中，抽象这一思想方法将到了充分的应用。我们回顾一下程序设计的几个阶段：二进制的机器指令；符号化的汇编语句；高级语言的执行语句；高级语言的过程（或函数）模块；面向对象的软件开发系统。在这些阶段中，后一阶段都是在前一阶段的基础上经过进一步的抽象后得以建立起来的。

通过一步步的抽象，不断地突出“做什么”，而将“怎么做”隐蔽起来，即把一切用户不必了解的细节封装起来，从而简化了问题。因此，抽象是程序设计最基本的思想方法。

现在介绍程序设计语言中出现的各种数据类型。

1.2.1 数据类型（Data Type）

数据类型是和数据结构密切相关的一个概念。它最早出现在高级程序设计语言中，用以刻画程序中操作对象的特性。在用高级语言编写的程序中，每个变量、常量或表达式都有一个它所属的确定的数据类型。类型显式地或隐含地规定了在程序执行期间变量或表达式所有可能的取值范围，以及在这些值上允许进行的操作。因此，数据类型是一个值的集合和定义在这个值集上的一组操作的总称。

在高级程序设计语言中，数据可分为原子类型和结构类型两类，原子类型的值是不可分解的。如 C 语言中整型、字符型、浮点型、双精度型等基本类型，分别用保留字 int，char，float，double 标识。而结构类型的值是由若干成分按某种结构组成的，因此是可分解的，并且它的成分可以是非结构的，也可以是结构的。例如，数组的值由若干分量组成，每个分量可以是整数，也可以是数组等。在某种意义上，数据结构可以看成是“一组具有相同结构的值”，而数据类型则可被看成是由一种数据结构和定义在其上的一组操作所组成的。

1.2.2 抽象数据类型（Abstract Data Type）

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

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

但在另一方面，抽象数据类型的范畴更广，它不再局限于前述各处理器中已定义并实现的数据类型，还包括用户在设计软件系统时自己定义的数据类型。为了提高软件的重用性，在近代程序设计方法学中，要求在构成软件系统的每个相对独立的模块上，定义一组数据和施于这些数据上的一组操作，并在模块的内部给出这些数据的表示及其操作的细节，而在模块的外部使用的只是抽象的数据及抽象的操作。这也就是面向对象的程序设计方法。

抽象数据类型的定义可以由一种数据结构和定义在其上的一组操作组成，而数据结构又包括数据元素及元素间的关系，因此抽象数据类型一般可以由元素、

关系及操作三种要素来定义。

抽象数据类型的特征是使用与实现相分离，实行封装和信息隐蔽。也就是说，在抽象数据类型设计时，把类型的定义与其实现分离开来。

一个软件系统可看作是由数据、操作过程和接口控制所组成的，当设计软件时，为便于从宏观上把握全局，要对它们进行抽象，即数据抽象、过程抽象和控制抽象。抽象数据类型不仅包含数学模型，还包含了模型上的运算。所以它将数据抽象和过程抽象结合为一体。抽象数据类型确定了一个数学模型，但将构成模型的具体细节加以隐蔽；它定义了一组运算，但又将运算的实现过程隐蔽了起来。

一个抽象数据类型的具体实现是可以多种多样的，而在软件中使用该抽象数据类型的地方则可以把它看作一般的初等类型，而不必管它的具体实现方法是什么，对抽象数据类型的定义及有关操作的设计、修改、完善等工作仅局限于相应的模块中。所以，抽象数据类型概念的引入，降低了大型软件设计的复杂性，使软件设计中普遍遵循的模块化、信息隐蔽、代码共享等思想得到更充分的体现。

1.3 算法和算法分析

算法与数据结构的关系紧密，在算法设计时先要确定相应的数据结构，而在讨论某一种数据结构时也必然会涉及相应的算法。下面就从算法特性、算法描述、算法性能分析与度量三个方面对算法进行介绍。

1.3.1 算法特性

算法（Algorithm）是对特定问题求解步骤的一种描述，是指令的有限序列。其中每一条指令表示一个或多个操作。一个算法应该具有以下 5 方面特性：

（1）有穷性。一个算法必须在有穷步之后结束，即必须在有限时间内完成。

（2）确定性。算法的每一步必须有确切的定义，无二义性。算法的执行对应着的相同的输入仅有唯一的一条路经。

（3）可行性。算法中的每一步都可以通过已经实现的基本运算的有限次执行得以实现。

（4）输入。一个算法具有零个或多个输入，这些输入取自特定的数据对象集合。

（5）输出。一个算法具有一个或多个输出，这些输出同输入之间存在某种特定的关系。

算法的含义与程序十分相似，但又有区别。一个程序不一定满足有穷性。例如，操作系统，只要整个系统不遭破坏，它将永远不会停止，即使没有作业需要处理，它仍处于动态等待中。因此，操作系统不是一个算法。另一方面，程序中的指令必须是机器可执行的，而算法中的指令则无此限制。算法代表了对问题的解，而程序则是算法在计算机上的特定的实现。一个算法若用程序设计语言来描述，则它就是一个程序。

算法与数据结构是相辅相成的。解决某一特定类型问题的算法可以选定不同的数据结构，而且选择恰当与否直接影响算法的效率的高低。反之，一种数据结构的优劣由各种算法的执行来体现。

要设计一个好的算法通常要考虑以下 4 点要求。

（1）正确。算法的执行结果应当满足预先规定的功能和性能要求。

（2）可读。一个算法应当思路清晰、层次分明、简单明了、易读易懂。

（3）健壮。当输入不合法数据时，应能作适当处理，不致引起严重后果。

（4）高效。有效使用存储空间和有较高的时间效率。

1.3.2 算法描述

算法可以使用各种不同的方法来描述。

最简单的方法是使用自然语言。用自然语言来描述算法的优点是简单且便于人们对算法的阅读；缺点是不够严谨。

可以使用程序流程图、N-S 图等算法描述工具，其特点是描述过程简洁、明了。用以上两种方法描述的算法不能够直接在计算机上执行，若要将它转换成可执行的程序还有一个编程的问题。

可以直接使用某种程序设计语言来描述算法，不过直接使用程序设计语言并不容易，而且不太直观，常常需要借助于注释才能使人看明白。

为了解决理解与执行这两者之间的矛盾，人们常常使用一种称为伪码语言的描述方法来进行算法描述。伪码语言介于高级程序设计语言和自然语言之间，它忽略高级程序设计语言中一些严格的语法规则与描述细节，因此它比程序设计语言更容易描述和被人理解，而比自然语言更接近程序设计语言。它虽然不能直接执行但很容易被转换成高级语言。

1.3.3 算法性能分析与度量

求解同一个问题，可以有许多不同的算法，那么如何来评价这些算法的好坏

呢？显然，首先选用的算法应该是“正确的”。此外，主要考虑以下 3 点：

（1）执行算法所耗费的时间。

（2）执行算法所耗费的存储空间，其中主要考虑辅助存储空间。

（3）算法应易于理解，易于编码，易于调试等。

当然我们希望选用一个所占存储空间小、运行时间短、其他性能也好的算法。然而，实际上很难做到十全十美。原因是上述要求有时相互抵触。要节约算法的执行时间往往要以牺牲更多的空间为代价；而为了节省空间又可能要以更多的时间作代价。因此我们只能根据具体情况有所侧重。若该程序使用次数较少，则力求算法简明易懂，易于转换为上机的程序。

当我们将一个算法转换成程序并在计算机上执行时，其运行所需要的时间取决于下列 4 方面因素：

（1）硬件的速度。

（2）书写程序的语言。实现语言的级别越高，其执行效率就越低。

（3）编译程序所生成目标代码的质量。对于代码优化较好的编译程序其所生成的程序质量较高。

（4）问题的规模。例如，求 100 以内的素数与求 1 000 以内的素数其执行时间必然是不同的。

显然，在各种因素都不能确定的情况下，很难比较出算法的执行时间。也就是说，使用执行算法的绝对时间来衡量算法的效率是不合适的。为此，可以将上述各种与计算机相关的软、硬件因素都确定下来。这样一个特定算法的运行工作量的大小就只依赖于问题的规模（通常用正整数 n 表示），或者说它是问题规模的函数。

1. 时间复杂度（Time complexity）

一个程序的时间复杂度是指程序运行从开始到结束所需要的时间。

一个算法是由控制结构和原操作构成的，其执行时间取决于两者的综合效果。为了便于比较同一问题的不同的算法，通常的做法是：从算法中选取一种对于所研究的问题来说是基本运算的原操作，以该原操作重复执行的次数作为算法的时间度量。一般情况下，算法中原操作重复执行的次数是规模 n 的某个函数 $T(n)$。

许多时候要精确地计算 $T(n)$ 是困难的，我们引入渐进时间复杂度在数量上估计一个算法的执行时间，也能够达到分析算法的目的。

定义（大“O”记号）：如果存在两个正常数 c 和 n_0，使得对所有的 n，

$n \geqslant n_0$，有：$f(n) \leqslant cg(n)$，则记为：$f(n) = O(g(n))$

例如，一个程序的实际执行时间为 $T(n) = 2.7n^3+3.8n^2+5.3$。则 $T(n) = O(n^3)$。

使用大“O”记号表示的算法的时间复杂度，称为算法的渐进时间复杂度（Asymptotic Complexity）。

当我们评价一个算法的时间性能时，主要标准是算法时间复杂度的数量级，即算法的渐近时间复杂度。例如，设有两个算法 A1 和 A2，求解同一问题，它们的时间复杂度分别是 $T1(n)=100n$，$T2(n) = n^2$，当输入量 $n<100$ 时，$T1(n)>T2(n)$，后者花费的时间较少。但是，随着问题规模 n 的增大，两个算法的时间开销之比 $n^2/100n$ 亦随着增大。也就是说，当问题规模较大时，算法 A1 比算法 A2 要有效得多，它们的渐近时间复杂度 $O(n)$ 和 $O(n^2)$，正是从宏观上评价了这两个算法在时间方面的质量。因此，在算法分析时，往往对算法的时间复杂度和渐近时间复杂度不予区分，而经常是将渐近时间复杂度 $T(n)=O(f(n))$ 简称为时间复杂度，其中的 $f(n)$ 一般是算法中频度最大的语句频度。

按数量级递增排列，常见的时间复杂度有常数阶 $O(1)$，对数阶 $O(\log_2 n)$，线性阶 $O(n)$，线性对数阶 $O(n\log_2 n)$，平方阶 $O(n^2)$，立方阶 $O(n^3)$，…，k 次方阶 $O(n^k)$，指数阶 $O(2^n)$。即

$$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < \cdots < O(2^n) \quad (1\text{-}2)$$

2. 空间复杂度（Space complexity）

一个程序的空间复杂度是指程序运行从开始到结束所需的存储量。

程序的一次运行是针对所求解的问题的某一特定实例而言的。例如，求解排序问题的排序算法的每次执行是对一组特定个数的元素进行排序。对该组元素的排序是排序问题的一个实例。元素个数可视为该实例的特征。

程序运行所需的存储空间包括以下两部分：

（1）固定部分。这部分空间与所处理数据的大小和个数无关，或者称与问题的实例的特征无关。主要包括程序代码、常量、简单变量、定长成分的结构变量所占的空间。

（2）可变部分。这部分空间大小与算法在某次执行中处理的特定数据的大小和规模有关。例如 100 个数据元素的排序算法与 1 000 个数据元素的排序算法所需的存储空间显然是不同的。

类似于算法的时间复杂度，空间复杂度记为

$$S(n)=O(f(n)) \quad (1\text{-}3)$$

式中，n 为问题的规模（或大小）。

1.4 项目训练

项目：学生信息管理

【题目要求】

利用数据结构的基本知识，构建学生信息管理系统。要求能输入并存储学生的学号、姓名、年级、专业等信息，并且可以输出信息。

【算法分析】

创建存储空间，定义存储表，依次在表中输入数据，查询数据表内容。

本章小结

（1）数据结构主要用来处理非数值计算性问题，这类问题设计的数据结构更复杂，数据元素之间的相互关系一般无法用数学方程式描述，需要设计合适的数据结构加以解决。

（2）数据是信息的载体，能被计算机识别、存储和处理，包括数值数据和非数值数据。数据项是数据的最小单位。数据对象是具有相同性质的数据元素的集合。数据结构是数据之间相互关系。

（3）数据类型是一个值的集合和定义在这个值集上的一组操作的总称，可分为原子类型和结构类型。

（4）抽象数据类型是指一个数学模型以及定义在该模型上的一组操作。

（5）算法是对特定问题求解步骤的一种描述，是指令的有限序列，具有确定性、有穷性、可行性等特性。它可以用自然语言、流程图、程序设计语言等来描述。

（6）算法的性能指标主要是所占存储空间、运行时间、易于编码及调试等。

习 题

1. 简述下列术语：数据、数据元素、数据对象、数据结构、存储结构、线性结构、算法、数据类型。

2. 说明数据结构的概念与程序设计语言中数据类型概念的区别和联系。

3. 讨论顺序存储结构和链式存储结构各自的特点、适用范围，并说明在实际应用中应如何选取数据存储结构。

4. 试举一个数据结构的例子，叙述其逻辑结构、存储结构、运算这三方面的内容。

5. 设 n 为正整数，利用大“O”记号将下列程序段的执行时间表示为 n 的函数。

①
```
i=1;k=100;
while (i<n){
    k=k+1;
    i+=10;
}
```

②
```
i=1;j=0;
while (i+j<=n)
    if (i>j)j++;
else i++;
```

③
```
x=n;//n 是不小于 1 的常数
y = 0;
while (x>=(y+1)*(y+1))
    y++;
```

④
```
x=91; y=100;
  while (y>0)
    if(x>100){x-=10;y--; }
    else x++;
```

⑤
```
for(i=1;i<=n-1;i++)
    { k=i;
        for(j=i+1;j<=n;j++)
            if(R[j]>R[j+1])k=j;
        t=R[k];R[k]=R[i];R[i]=t;
    }
```

⑥
```
for (i=1;i<=n;i++)
    for (j=1;j<=i;j++)
        for (k=1;k<=j;k++)
```

```
                x=x+y;
```

6. 试写一算法，自大至小依次输出顺序读入的 3 个整数 x、y 和 z 的值。

7. 已知 k 阶裴波那契序列的定义为

$$f_0 = 0,\ f_1=0,\ \cdots,\ f_{k-2}=0,\ f_{k-1}=1;$$

$$f_n=f_{n-1}+f_{n-2}+\cdots+f_{n-k},\ n=k,\ k+1,\ \cdots$$

试编写求 k 阶裴波那契序列的第 m 项值的函数算法，k 和 m 均以参数的形式在参量表中出现。

8. 编写一个程序计算一元多项式 $P_n(x)=p_0+p_1x+p_2x^2+\cdots+p_nx^n$ 的值 $P_n(x_0)$，设 n、x_0 和 $P_i(0 \leqslant i \leqslant n)$ 均为已知量，从键盘读入。所编写程序中 $P_i(0 \leqslant i \leqslant n)$ 的值是采用什么结构存储的?

9. 试编写算法计算 i！ *2^i 的值并存入数组 a[aaisize] 的第 i 个分量中 i = 1，2，…，n。假设计算机中允许的整数最大值为 Maxint，则当 n>arrsize 或对某个 $k(1 \leqslant k \leqslant n)$ 使 k！ *2^k>Maxint 时，应按出错处理。可有下列 3 种不同的处理方式：

（1）用 ERROR 语句终止执行并报告错误；

（2）用返回值 0 或 1 实现算法以区别正确返回或错误返回；

（3）在参数表中设置一整型变量以区别正确返回或某种错误返回。

试讨论这 3 种方法各自的优缺点，并以你认为较好的方式实现。

第2章 线 性 表

线性表（Linear List）是线性结构中最常用而又最简单的一种数据结构，几乎所有线性的关系都可以用线性表表示。线性表是线性结构的抽象，线性结构的特点是数据元素之间的一对一的线性关系，数据元素一个接一个地排列。因此，线性表可以想象为一种数据元素的序列。本章主要介绍线性表的逻辑结构和各种存储表示方法，以及定义在存储结构上的各种基本操作的实现。

知识目标

- 理解线性表的逻辑结构特征。
- 掌握顺序表的含义、特点、基本运算和相关算法分析。
- 掌握链表的含义、特点、基本运算和相关算法分析。
- 理解循环链表和双链表的逻辑结构特征及基本运算。
- 理解顺序表和链表的比较。

能力目标

- 能应用顺序表的理论设计算法，解决实际问题。
- 能应用链表的理论设计算法，解决实际问题。

2.1 线性表的逻辑结构

线性表是最基本、最简单、也是最常用的一种数据结构。线性表（Linear List）是数据结构的一种，一个线性表是 n 个具有相同特性的数据元素的有限序列。

线性表中数据元素之间的关系是一对一的关系，即除了第一个和最后一个数据元素之外，其他数据元素都是首尾相接的[注意，这句话只适用大部分线性表，而不是全部。比如，循环链表逻辑层次上也是一种线性表（存储层次上属于链式存储），但是把最后一个数据元素的尾指针指向了首位节点]。

2.1.1 线性表的定义

1. 线性表的定义

线性表是具有相同数据类型的 n（$n \geqslant 0$）个数据元素的有限序列（即集合），通常记为

$$\{a_1,a_2,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n\}$$

式中，n 为表长，当 n=0 时称为空表（即空集）。

在线性表中相邻元素之间存在着顺序关系。如对于元素而言，则称为直接前驱或直接后继。

2. 线性表举例

（1）简单线性表。在日常生活中有许多线性表的例子，如26个英文字母表：

$$\{a,b,c,d,e,f,g,\cdots,x,y,z\}$$

又例如一周七天：

$$\{1,2,3,4,5,6,7\}$$

（2）复杂的线性表。学生信息表就是一个典型的线性结构，见表2-1。

表2-1 学生信息表

学 号	姓 名	性 别	年 龄	成 绩
1001	李一	女	18	93
1002	王二	男	19	80
1003	张三	男	18	90
……				

在这个线性表中，每一个学生所对应的一行信息就是一个数据元素，也称为记录，包括学号、姓名、性别、年龄和成绩共五个数据项。记录之间是一对一的关系，除了第一条记录和最后一条记录，每条记录都只有一个直接前驱和直接后继。

3. 线性表的特点

（1）集合中必存在唯一的一个“第一元素”。

（2）集合中必存在唯一的一个“最后元素”。

（3）除最后一个元素之外，均有唯一的后继（后件）。

（4）除第一个元素之外，均有唯一的前驱（前件）。

2.1.2 线性表的基本操作

线性表是一种非常灵活的数据结构，它的长度可以根据问题的需要增加或减少，可以对数据元素进行访问、插入和删除等一系列基本操作。在解决实际问题过程中，将会遇到不同的运算对象和不同的数据类型。

线性表常用的基本操作：

（1）初始化线性表 InitList(L)，建立一个空表 L。

（2）求线性表的长度 GetLength(L)，返回线性表 L 的长度，即所含数据元素的个数。

（3）按位置查找操作 SearchList(L，i，x)，在线性表 L 查找第 i 位上的数据元素，其作用是若第 i 位上有数据元素，则由 x 传回该元素值，返回 1；若不存在返回 0。

（4）按值查找操作 Locate(L，x)，在线性表 L 查找一个与给定值 x 相等的数据元素，其作用是若存在一个或多个与 x 相等的数据元素，则返回元素所在位置的最小值或地址值；否则返回 0 或 NULL 值。

（5）插入操作 InsertList(L，i，x)，其作用是在线性表 L 的第 i 个位置上插入一个值为 x 的新元素。

（6）删除操作 DeleteList(L，i)，其作用是删除线性表 L 的第 i 个位置的数据元素并用 x 将其存储。

（7）显示操作 DispList(L)，其作用是依次扫描线性表 L，并输出各元素的值。

2.2 线性表的存储结构

数据结构在内存中的表示通常有两种形式，即顺序存储表示和链式存储表示。

2.2.1 线性表的顺序存储结构

线性表的顺序存储结构又称顺序表（Sequential List）。

1. 顺序表结构

线性表的顺序存储是指用一组地址连续的存储单元依次存储线性表的数据元素，我们把用这种存储形式存储的线性表称为顺序表。线性表的顺序存储表示又称为顺序表。顺序表的逻辑顺序和物理顺序是一致的。

假设顺序表 $\{a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n\}$，每个数据元素占用 d 个存储单元，则元素的存储位置为

$$\mathrm{Loc}(a_i)=\mathrm{Loc}(a_1)+(i-1)\times d \ , \ 1 \leqslant i \leqslant n \qquad (2\text{-}1)$$

其中，$\mathrm{Loc}(a_1)$ 是顺序表第一个元素的存储位置，通称为顺序表的起始地址。顺序存储结构示意图如图 2-1 所示。

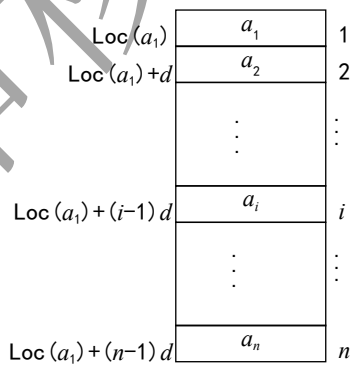


图 2-1 线性表顺序存储结构示意图

线性表的顺序存储结构 C 语言描述如下：

```
    #define  MAXLEN  100              /* 定义常量 MAXLEN 为 100 表示存储空间总量 */
    typedef  int  DataType;           /* 定义 DataType 为 int 类型 */
    typedef  struct                   /* 顺序表存储类型 */
```

```
{  DataType  data[MAXLEN];     /* 存放线性表的数组 */
   int  Length;                /*Length 是顺序表的长度 */
}SeqList;
```

2. 顺序表的运算

在定义了线性表顺序存储结构之后，就可以讨论在这种结构上如何实现有关数据的运算问题。在这种存储结构下，某些线性表的运算很容易实现，如求线性表的长度、取第 i 个数据元素以及求直接前驱和直接后继等。下面讨论线性表中数据元素的插入和删除运算。

（1）顺序表的建立。从键盘输入 n 个整数，并将这些整数存入顺序表中，修改表长后建立表 L。算法描述如下：

```
#include <stdio.h>
#define  MAXLEN  100           /* 定义常量 MAXLEN 为 100 表示存储空间总量 */
typedef  int  DataType;        /* 定义 ElemType 为 int 类型 */
typedef  struct                /* 顺序表存储类型 */
{  DataType  data[MAXLEN];     /* 存放线性表的数组 */
   int  Length;                /*Length 是顺序表的长度 */
}SeqList;

void  InitList( SeqList  *L )
{  /* 初始化顺序表 L 函数 */
    L->Length=0;               /* 初始化顺序表为空 */
}

void CreateList(SeqList *L,int n)
{ /* 建立顺序表并输入多个元素函数 */
  int i;
  printf(" 请输入 %d 个整数：",n);
  if(n> MAXLEN)
  {
   printf("overflow");         /* 如果 n 大于 MaxSize，出现上溢 */
   return 0;
  }
  for(i=0;i<n;i++)
```

```
        scanf("%d",&L->data[i]);
    L->Length=i;                          /* 设线性表的长度为 i*/
}

void DispList(SeqList *L)
{  /* 显示输出顺序表 L 的每个元素函数 */
     int i;
    for(i=0;i< L->Length;i++)
       printf("%5d ", L->data[i]);
}

main()
{
   SeqList L;
   int n;
   InitList( &L );
   printf(" 请输入建立线性表的个数：");
   scanf("%d",&n);
   CreateList(&L,n);
   printf(" 建立的线性表为 :");
   DispList(&L);
}
```

程序运行后结果如图 2-2 所示。

```
请输入建立线性表的个数：5
请输入 5 个整数：1 2 3 4 5
建立的线性表为：    1    2    3    4    5
```

图 2-2　建立顺序表运行结果图

（2）插入运算。插入运算是指在有 n 个元素的线性表的第 i（$1 \leqslant i \leqslant n$）个元素之前插入一个新元素 x。由于顺序表中的元素在机器内是连续存放的，要在第 i 元素之前插入一个新元素，就必须把第 n 个到第 i 个之间所有元素依次向后移动一个位置，空出第 i 个位置后，再将新元素 x 插入到第 i 个位置。新元素插入后，线性表长度变为 n+1。

顺序表插入元素的过程如图 2-3 所示。

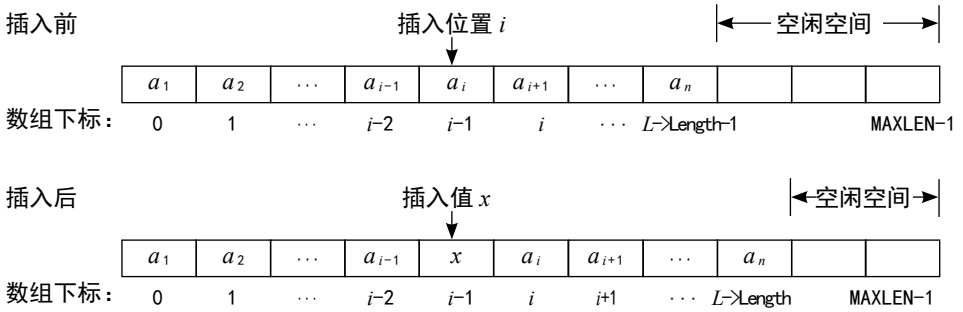


图 2-3 顺序表插入元素示意图

具体实现，程序代码如下：

```
#include <stdio.h>
#define MAXLEN 100              /* 定义常量 MAXLEN 为 100 表示存储空间总量 */
typedef int DataType;           /* 定义 ElemType 为 int 类型 */
typedef struct                  /* 顺序表存储类型 */
{ DataType data[MAXLEN];        /* 存放线性表的数组 */
  int Length;                   /*Length 是顺序表的长度 */
}SeqList;

void InitList( SeqList *L )
{ /* 初始化顺序表 L 函数 */
   L->Length=0;                 /* 初始化顺序表为空 */
}

void CreateList(SeqList *L,int n)
{ /* 建立顺序表并输入多个元素函数 */
  int i;
  printf(" 请输入 %d 个整数：",n);
  for(i=0;i<n;i++)
     scanf("%d",&L->data[i]);
  L->Length=i;                  /* 设线性表的长度为 i*/
}
int InsertList(SeqList *L, int i, DataType x)
{ /* 在顺序表 L 的第 i 位中插入新元素 x 函数 */
   int j;
  if (L->Length>=MAXLEN)
```

```
    {  printf (" 顺序表已满！ ");
       return 0;                          /* 表满，不能插入 */
    }
    if (i<1 || i>L->Length+1)             /* 检查给定的插入位置的正确性 */
    {  printf(" 插入位置出错！ ");
       return 0;
    }
    for (j=L->Length-1; j>=i-1; j--)          /* 节点移动 */
       L->data[j+1]=L->data[j];
    L->data[i-1]=x;                       /* 新元素插入 */
    L->Length++;                          /* 顺序表长度增 1 */
    return 1;                             /* 插入成功，返回 */
}

void DispList(SeqList *L)
{  /* 显示输出顺序表 L 的每个元素函数 */
    int i;
    for(i=0;i< L->Length;i++)
      printf("%5d ", L->data[i]);
}

main()
{
  SeqList L;
  int i,x,n;
  InitList( &L );
  printf(" 请输入建立线性表的个数：");
  scanf("%d",&n);
  CreateList(&L,n);
  printf(" 请输入要插入的位置：");
  scanf("%d",&i);
  printf(" 请输入要插入的元素值：");
  scanf("%d",&x);
  InsertList(&L,i,x);
  printf(" 建立的线性表为 :");
  DispList(&L);
}
```

程序运行后结果如图 2-4 所示。

```
请输入建立线性表的个数：5
请输入 5 个整数：1 2 3 4 5
请输入要插入的位置：3
请输入要插入的元素值：7
建立的线性表为：      1     2     7     3     4     5
```

图 2-4　顺序表插入元素运行结果图

（3）删除运算。删除运算是指在有 n 个元素的线性表中，删除其中的第 i（$1 \leqslant i \leqslant n$）个元素，使线性表的长度减 1。若要删除表中的第 i 个元素，就必须把表中的第 i+1 个到第 n 个之间的所有元素依次向前移动一个位置，以覆盖前一个位置上的内容，线性表的长度变为 n-1，如图 2-5 所示。

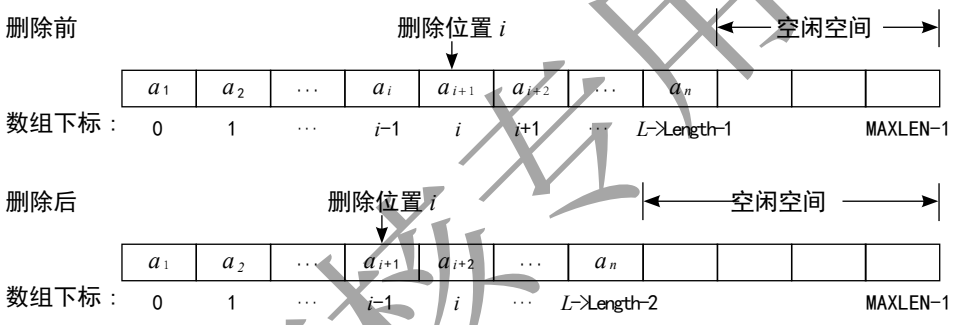


图 2-5　顺序表插入数据示意图

```
#include <stdio.h>
#define  MAXLEN  100              /* 定义常量 MAXLEN 为 100 表示存储空间总量 */
typedef  int  DataType;           /* 定义 ElemType 为 int 类型 */
typedef  struct                   /* 顺序表存储类型 */
{  DataType  data[MAXLEN];        /* 存放线性表的数组 */
   int  Length;                   /*Length 是顺序表的长度 */
}SeqList;

void  InitList( SeqList  *L )
{  /* 初始化顺序表 L 函数 */
    L->Length=0;                  /* 初始化顺序表为空 */
}
```

```
void CreateList(SeqList *L,int n)
{  /* 建立顺序表并输入多个元素函数 */
   int i;
   printf(" 请输入 %d 个整数：",n);
   for(i=0;i<n;i++)
      scanf("%d",&L->data[i]);
   L->Length=i;                          /* 设线性表的长度为 i*/
}
int DeleteList(SeqList *L, int i)
{   /* 在顺序表 L 中删除第 i 位元素函数 */
     int  j;
    if (L->Length==0)
    {  printf (" 顺序表为空！ ");
       return 0;                         /* 表空，不能删除 */
    }
    if (i<1 || i>L->Length)              /* 检查是否空表及删除位置的合法性 */
    {  printf (" 删除位置非法 ");
       return 0;
    }
       for(j=i;j<L->Length;j++)          /* 节点移动 */
       L->data[j-1]=L->data[j];
    L->Length--;                         /* 顺序表长度减 1*/
    return 1;                            /* 删除成功，返回 */
}

void DispList(SeqList *L)
{   /* 显示输出顺序表 L 的每个元素函数 */
     int i;
    for(i=0;i< L->Length;i++)
       printf("%5d ", L->data[i]);
}

main()
{
   SeqList L;
   int i,x,n;
   InitList( &L );
   printf(" 请输入建立线性表的个数：");
```

```
    scanf("%d",&n);
    CreateList(&L,n);
    printf(" 请输入要删除元素的位置：");
    scanf("%d",&i);
    DeleteList(&L, i);
    printf(" 建立的线性表为 :");
    DispList(&L);
}
```

程序运行结果如图所 2-6 示。

```
请输入建立线性表的个数：5
请输入 5 个整数：1 2 3 4 5
请输入要插入的位置：3
建立的线性表为：    1    2    4    5
```

图 2-6　顺序表删除元素运行结果图

2.2.2　线性表的链式存储结构

顺序表的特点是利用数据元素在物理位置上的邻接关系来表示节点间的逻辑关系，这样顺序表就不需要为表示节点间的逻辑关系而增加额外的空间，同时也可以直接存取表中的任一元素。但它也带有以下 3 方面的缺点：

（1）插入和删除操作需要移动大量的节点。

（2）表的容量难以预先确定。在为长度变化较大的线性表预先分配空间时，只能按照最大空间需求分配，造成空间利用率低。

（3）造成存储空间的“碎片”。因为顺序表存储要求占用连续的存储空间，即使空闲单元总数超过了表的容量，如果不连续，也无法使用。

鉴于顺序表的这些不足，我们考虑线性表的链式存储结构。

1. 链表的结构

线性表的链式存储结构简称链表（Linked List），是用一组任意的存储单元存储该线性表中的各个数据元素，存储单元可以连续，也可以不连续。因此，链表中数据元素的逻辑次序和物理次序不一定相同。为了能体现元素间的逻辑顺序，每个节点除了存储数据元素的信息外，还要存储其后继元素所在的地址信息。一个链表节点由两个域构成：存储数据元素信息的域称为数据域 (data)；存储直接后继存储位置的域称为指针域 (next)，如图 2-7 所示。

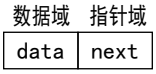


图 2-7 单链表的节点结构

单链表分为带头节点（其 next 域指向链表第一个节点的存储地址）和不带头节点两种类型。在许多情况下，带头节点的链表中每个节点的存储地址均放在其前驱节点中，这样算法对所有的节点处理可一致化，因此，本节讨论的单链表均指带头节点的单链表。带头节点的空单链表如图 2-8(a) 所示，带头节点的非空单链表如图 2-8(b) 所示。

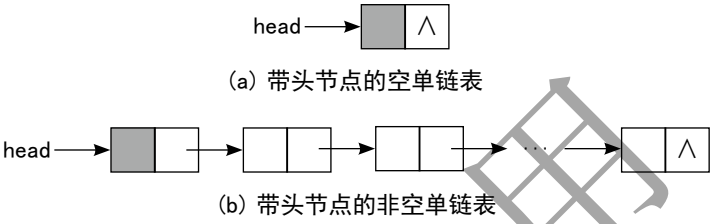


图 2-8 单链表示意图

其中，头节点的数据域可以不存储任何信息，也可以存放特殊的信息；头节点的指针域存储链表中第一个节点的地址。当头节点的指针域为空（即 NULL 或用“∧”表示），则此表为空表。在非空表中，当某个节点的指针域为空，表示它为链表的最后一个节点。

通常用“头指针”来标示一个线性链表，如头指针 head 是指某链表第一个节点的地址放到了指针变量 head 中，带头节点的链表的头节点的指针域为“NULL”则表示一个空表。由于单向链表不能随机存取存储的数据元素，在单向链表中存取第 i 个元素，必须从头指针出发寻找，其寻找的时间复杂度为 $O(n)$。

2. 单链表

（1）单链表的类型定义。单链表由一个个节点构成，我们用 C 语言的结构体指针描述如下：

```
typedef  int  DataType;     /* 定义 DataType 为 int 类型 */
typedef  struct  node       /* 单链表存储类型 */
{
  DataType  data;           /* 定义节点的数据域 */
  struct  node  *next;      /* 定义节点的指针域 */
} LinkList;
```

（2）建表。每读入有效的数据则申请一个节点，并将读取的数据存放到新

节点的数据域中，将节点的尾指针设为空指针（NULL），然后将新节点插入到当前链表尾部（Last 指针所指的节点后面），直到循环结束为止。

```
#include <stdio.h>
#include <malloc.h>
typedef  int  DataType;              /* 定义 DataType 为 int 类型 */
typedef  struct  node                /* 单链表存储类型 */
{
  DataType      data;                /* 定义节点的数据域 */
  struct  node   *next;              /* 定义节点的指针域 */
} LinkList;

LinkList  *InitList()
{  /* 初始化链表函数 */
   LinkList  *head;
   head=(LinkList*)malloc(sizeof(LinkList)); /* 动态分配一个节点空间 */
   head->next=NULL;
   return head;                      /* 头节点 L 指针域为空，表示空链表 */
}
void CreateList (LinkList *head,int n)
{ /* 尾插法建立链表函数 */
   LinkList *q,*p;                   /* 定义指针变量 */
   int i,a;
   q=head;
    printf(" 请输入 %d 个整数：",n);
   for(i=0;i<n;i++)
   {
     p=(LinkList *)malloc(sizeof(LinkList));
      scanf("%d",&a);
     p->data=a;                      /* 把 a 的值赋给节点 p*/
     q->next=p;                      /* 将节点 p 加入到节点 q 之后 */
     q=p;
      q->next=NULL;
    }
printf(" 建立链表操作成功！ ");
}

void DispList(LinkList *head)
```

```
{ /* 显示输出链表函数 */
  LinkList *p;
  p=head->next;
  while(p!=NULL)
  {
    printf("%5d",p->data);
    p=p->next;
  }
}

main()
{
  LinkList *head;
  int n;
  head=InitList();
  printf(" 请输入要建立线性表的长度：");
  scanf("%d",&n);
  CreateList (head,n);
  printf(" 建立后的线性表为：\n");
  DispList(head);
}
```

程序运行结果如图 2-9 所示。

```
请输入要建立线性表的长度：5
请输入 5 个整数：1 2 3 4 5
建立链表操作成功！建立后的线性表为：
    1    2    3    4    5Press any key to continue
```

图 2-9 建立单链表运行结果图

（3）插入。顺序表的插入操作需要移动大量的数据元素，而链表的插入只需修改指针而无须移动原来表中元素，那链表的插入操作是如何实现呢？

1）在指针所指的节点后插入新节点。若要在链表中指针 p 所指位置后面插入一个节点，则插入操作步骤如下：

先将节点 q 的指针域指向节点 p 的下一个节点（执行语句：q->next=p->next）；再将节点 p 的指针域改为指向新节点 q（执行语句：p->next=q）。插入节点的过程如图 2-10 所示。

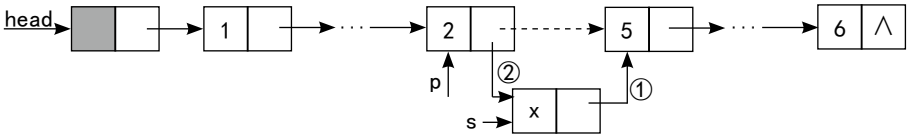


图 2-10　插入节点示意图

2）插入算法思路。在第 i 个位置插入新节点 q 的算法思路如下：由于单链表的节点结构是单向后指的，因此要完成此操作需要找到第 i 节点的前驱节点即第 $i-1$ 节点的指针 p，然后在已知节点 p 后方插入新节点即可。程序代码如下：

```
#include <stdio.h>
#include <malloc.h>
typedef  int  DataType;              /* 定义 DataType 为 int 类型 */
typedef  struct  node                /* 单链表存储类型 */
{
  DataType     data;                 /* 定义节点的数据域 */
  struct  node  *next;               /* 定义节点的指针域 */
} LinkList;

LinkList  *InitList()
{  /* 初始化链表函数 */
   LinkList  *head;
   head=(LinkList*)malloc(sizeof(LinkList)); /* 动态分配一个节点空间 */
   head->next=NULL;
   return head;                      /* 头节点 L 指针域为空，表示空链表 */
}
void CreateList(LinkList  *head,int n)
{ /* 尾插法建立链表函数 */
   LinkList *q,*p;                   /* 定义指针变量 */
   int i,a;
   q=head;
    printf(" 请输入 %d 个整数：",n);
   for(i=0;i<n;i++)
   {
     p=(LinkList *)malloc(sizeof(LinkList));
      scanf("%d",&a);
     p->data=a;                      /* 把 a 的值赋给节点 p*/
     q->next=p;                      /* 将节点 p 加入到节点 q 之后 */
     q=p;
```

```
        q->next=NULL;
      }
  }

  void InsertList(LinkList  *head, int i, DataType x)
  { /* 按位置插入元素函数 */
    int j=0;
    LinkList  *p,*q;
    p=head;
    while(p->next!=NULL && j<i-1)  /* 定位插入点 */
    {
      p=p->next;
      j++;
    }
    if(p!=NULL)                    /*p 不为空则将新节点插到 p 后 */
    {
       q=(LinkList *)malloc(sizeof(LinkList)); /* 生成新节点 s*/
      q->data=x;                   /* 将数据 x 放入新节点的数据域 */
      q->next=p->next;             /* 将新节点 q 的指针域与 p 节点后面元素
相连 */
       p->next=q;                  /* 将 p 与新节点 q 链接 */
    }
    else
      printf(" 插入元素失败 ");
  }


  void DispList(LinkList *head)
  { /* 显示输出链表函数 */
    LinkList *p;
    p=head->next;
    while(p!=NULL)
    {
      printf("%5d",p->data);
      p=p->next;
    }
  }
```

```
main()
{
  LinkList *head;
  int n,x,i;
  head=InitList();
  printf(" 请输入要建立线性表的长度：");
  scanf("%d",&n);
  CreateList(head,n);
  printf(" 请输入要插入的元素值：");
  scanf("%d",&x);
 printf(" 请输入要插入的元素位置：");
  scanf("%d",&i);
  InsertList(head,i,x);
  printf(" 插入后的线性表为：\n");
  DispList(head);
}
```

程序运行结果如图 2-11 所示。

```
请输入要建立线性表的长度：5
请输入 5 个整数：1 2 3 4 5
请输入要插入的元素值：6
请输入要插入的元素位置：3
插入后的线性表为：
   1    2    6    3    4    5Press any key to continue
```

图 2-11 单链表插入节点运行结果图

（4）删除。首先通过循环定位求出第 i 节点的前驱节点（第 i-1 节点）p 的地址，然后将指针 q 指向被删除节点，修改 p->next 指针，使其指向 q 后的节点，最后释放指针 q 所指节点。算法中注意 if(p->next!=NULL && j==i-1) 语句，只有当第 i-1 节点存在（j==i-1）而 p 又不是终端节点即（p->next!=NULL）时，才能确定被删除节点存在。

删除节点的过程如图 2-12 所示。

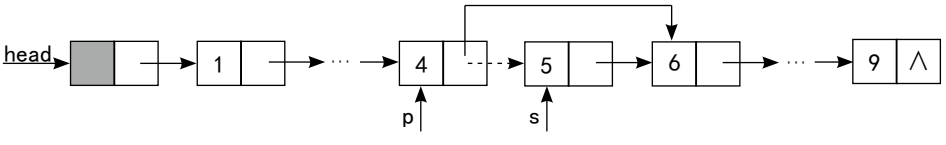


图 2-12 删除节点示意图

```
#include <stdio.h>
#include <malloc.h>
typedef  int  DataType;              /* 定义 DataType 为 int 类型 */
typedef  struct  node                /* 单链表存储类型 */
{
  DataType      data;                /* 定义节点的数据域 */
  struct  node   *next;              /* 定义节点的指针域 */
} LinkList;

LinkList  *InitList()
{  /* 初始化链表函数 */
   LinkList  *head;
   head=(LinkList*)malloc(sizeof(LinkList)); /* 动态分配一个节点空间 */
   head->next=NULL;
   return head;                      /* 头节点 L 指针域为空，表示空链表 */
}
void CreateList(LinkList  *head,int n)
{ /* 尾插法建立链表函数 */
   LinkList *q,*p;                   /* 定义指针变量 */
   int i,a;
   q=head;
    printf(" 请输入 %d 个整数： ",n);
   for(i=0;i<n;i++)
   {
     p=(LinkList *)malloc(sizeof(LinkList));
      scanf("%d",&a);
     p->data=a;                      /* 把 a 的值赋给节点 p*/
     q->next=p;                      /* 将节点 p 加入到节点 q 之后 */
     q=p;
      q->next=NULL;
    }
}

void DeleteList(LinkList *head,int i)
{ /* 按位置删除链表中元素函数 */
  int j=0;
  DataType x;
```

```
    LinkList  *p=head,*q;
    while(p->next!=NULL && j<i-1)  /* 定位插入点 */
    {
      p=p->next;
      j++;
    }
    if (p->next!=NULL && j==i-1)
    {
      q=p->next;                    /*q 为要删除节点 */
      x=q->data;                    /* 将要删除的数据放入指针变量 x 中 */
      p->next=q->next;              /* 将 p 节点的指针域与 q 节点后面元素
相连 */
      free(q);
      printf(" 删除第 %d 位上的元素 %d 成功！ ",i,x);
    }
    else
      printf(" 删除节点位置错误，删除失败！ ");
  }

  void DispList(LinkList *head)
  { /* 显示输出链表函数 */
    LinkList *p;
    p=head->next;
    while(p!=NULL)
    {
      printf("%5d",p->data);
      p=p->next;
    }
  }

  main()
  {
    LinkList *head;
    int n,i;
    head=InitList();
    printf(" 请输入要建立线性表的长度：");
    scanf("%d",&n);
    CreateList(head,n);
```

```
    printf(" 请输入要删除的元素位置：");
    scanf("%d",&i);
    DeleteList(head,i);
    printf(" 删除第 %d 位的元素后的线性表为：\n",i);
    DispList(head);
}
```

程序运行结果如图 2-13 所示。

```
请输入要建立线性表的长度：5
请输入 5 个整数：1 2 3 4 5
请输入要删除的元素位置：3
删除第 3 位上的元素 3 成功！删除第 3 位的元素后的线性表为：
    1    2    4    5Press any key to continue
```

图 2-13 单链表删除节点运行结果图

（5）查找。

1）按位置查找。首先判断 i 值是否大于表长，如果大于则输出位置错误信息；否则从链表的头节点开始，判断当前节点序号是否是 i，成立则提前结束循环。最后输出第 i 位上的节点的数据域值和位置 i。

```
#include <stdio.h>
#include <malloc.h>
typedef  int  DataType;              /* 定义 DataType 为 int 类型 */
typedef  struct  node                /* 单链表存储类型 */
{
  DataType    data;                  /* 定义节点的数据域 */
  struct  node  *next;               /* 定义节点的指针域 */
} LinkList;

LinkList  *InitList()
{  /* 初始化链表函数 */
   LinkList  *head;
   head=(LinkList*)malloc(sizeof(LinkList)); /* 动态分配一个节点空间 */
   head->next=NULL;
   return head;                      /* 头节点 L 指针域为空，表示空链表 */
}
void CreateList(LinkList  *head,int n)
{ /* 尾插法建立链表函数 */
```

```
    LinkList *q,*p;                    /* 定义指针变量 */
    int i,a;
    q=head;
     printf(" 请输入 %d 个整数：",n);
    for(i=0;i<n;i++)
    {
      p=(LinkList *)malloc(sizeof(LinkList));
       scanf("%d",&a);
      p->data=a;                       /* 把 a 的值赋给节点 p*/
      q->next=p;                       /* 将节点 p 加入到节点 q 之后 */
      q=p;
       q->next=NULL;
     }
}

void SearchList(LinkList *head,int i)
{  /* 在链表中按位置查找元素 */
   LinkList  *p;
   int j=0;
   p=head;                             /*p 指向链表的头节点 */
   while(p->next!=NULL && j<i)
   {
     p=p->next;
     j++;
   }
   if(j==i)                            /* 判断与给定的序号是否相等 */
     printf(" 在第 %d 位上的元素值为 %d！ ",i,p->data);
}

main()
{
   LinkList *head;
   int n,i;
   head=InitList();
   printf(" 请输入要建立线性表的长度：");
   scanf("%d",&n);
   CreateList(head,n);
   printf(" 请输入查找的元素位置（大于等于 1 的整数）：");
```

```
    scanf("%d",&i);
    SearchList(head,i);
}
```

程序运行结果如图 2-14 所示。

```
请输入要建立线性表的长度：5
请输入 5 个整数：1 2 3 4 5
请输入查找的元素位置（大于等于 1 的整数）：2
在第 2 位上的元素值为 2！ Press any key to continue
```

图 2-14 按位置查找元素运行结果图

2）按值查找。从链表的第一个元素节点开始，由前向后依次比较单链表中各节点数据域中的值，若某节点数据域中的值与给定的值 *x* 相等，则循环结束；否则继续向后比较直到表结束，然后判断指针 p，若 p 不为空表示单链表中有 *x* 个节点，输出查找成功的信息并输出 *x* 所在表中的位置；否则输出查找失败的信息。

```
#include <stdio.h>
#include <malloc.h>
typedef  int  DataType;              /* 定义 DataType 为 int 类型 */
typedef  struct  node                /* 单链表存储类型 */
{
  DataType      data;                /* 定义节点的数据域 */
  struct  node   *next;              /* 定义节点的指针域 */
} LinkList;

LinkList  *InitList()
{  /* 初始化链表函数 */
   LinkList  *head;
   head=(LinkList*)malloc(sizeof(LinkList)); /* 动态分配一个节点空间 */
   head->next=NULL;
   return head;                      /* 头节点 L 指针域为空，表示空链表 */
}
void CreateList(LinkList  *head,int n)
{ /* 尾插法建立链表函数 */
   LinkList *q,*p;                   /* 定义指针变量 */
   int i,a;
   q=head;
    printf(" 请输入 %d 个整数：",n);
```

```
    for(i=0;i<n;i++)
    {
      p=(LinkList *)malloc(sizeof(LinkList));
       scanf("%d",&a);
      p->data=a;                    /* 把 a 的值赋给节点 p*/
      q->next=p;                    /* 将节点 p 加入到节点 q 之后 */
      q=p;
       q->next=NULL;
     }
}

void Locate(LinkList  *head,DataType x)
{  /* 在链表中查找值为 x 的元素位置 */
   int  j=1;                        /* 计数器 */
   LinkList  *p;
   p=head->next;
   while(p!=NULL  &&  p->data!=x)       /* 查找及定位 */
   {  p=p->next;
      j++;
   }
   if(p!=NULL)
     printf(" 在表的第 %d 位找到值为 %d 的节点！ ",j,x);
   else
     printf(" 未找到值为 %d 的节点！ ",x);
}

main()
{
   LinkList *head;
   int n,x;
   head=InitList();
   printf(" 请输入要建立线性表的长度：");
   scanf("%d",&n);
   CreateList(head,n);
   printf(" 请输入查找的整数：");
   scanf("%d",&x);
   Locate(head,x);
}
```

程序运行结果如图 2-15 所示。

```
请输入要建立线性表的长度：5
请输入 5 个整数：1 2 3 4 5
请输入查找的整数：4
在表的第 4 位找到值为 4 的节点！ Press any key to continue
```

图 2-15 按值查找元素运行结果图

3. 循环链表

将单链表中的最后一个节点的指针指向链表中第一个节点，使整个链表构成一个环形，这种链表称为单循环链表，简称循环链表（Circular Linked List）。

从循环链表中的任意一个节点出发都可以找到表中其他节点。为了使空表与非空表的处理统一，通常循环链表也附设一个头节点，如图 2-16 所示。有时，在单循环链表中只设指向尾节点的尾指针 rear 而不设头指针，这样对链表头节点和尾节点的操作都变得方便了。

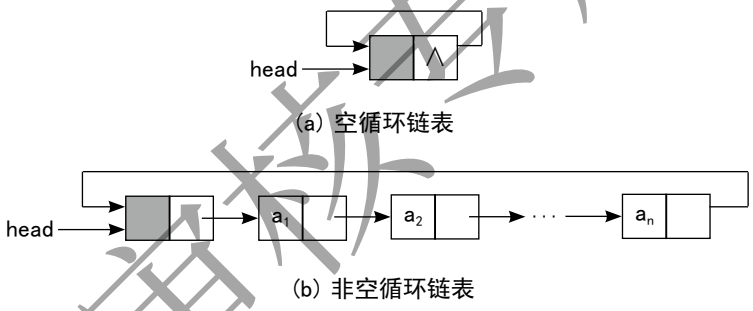


图 2-16 循环链表示意图

循环链表的运算和单链表基本一样，差别在于：当需要从头到尾扫描整个链表时，是否达到表尾的条件不同。在单链表中找表尾节点要判断某节点链域值是否为“空”；在循环链表中找表尾节点则要判断某节点的链域值是否等于头指针。

在循环链表中，从表中任一节点 p 出发，都可以找到它的直接前驱节点。算法如下：

```
LinkList prior(ListNode *p)
/* 求循环链表中任意一个节点的前驱节点 */
{
  ListNode *q;
  q=p->next;                /* 初始化 q 为 p 的直接后继 */
  while(q->next!=p)         /* 当 q 的直接后继为 p 时，q 是 p 的直接前驱 */
```

```
    q=q->next;
  return q;
}
```

4. 双链表

（1）双向链表的定义。在单链表和循环链表中，数据元素的节点除数据域外，只有一个指向其直接后继的指针域，若要查找其前驱节点就需要遍历链表。为了解决这种单向性的问题，可以在单链表的节点中增加一个指向其直接前驱节点的指针域，这样有两种不同方向链的链表就称为双（向）链表（Double Linked List），其节点结构如图 2-17(a) 所示。其中，data：数据域，存放数据元素；prior：前驱指针域，存放该节点的前驱节点地址；next：后继指针域，存放该节点的后继节点地址。给双链表加一表头节点成为带表头节点的双链表，如图 2-17(b) 所示。如果每条链都构成循环链表，就形成了双循环链表，如图 2-17(c) 所示。

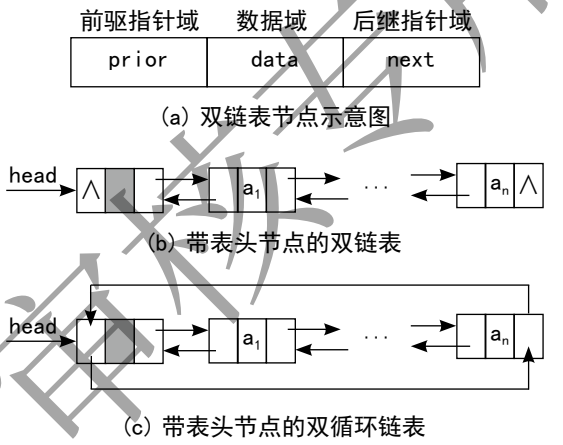


图 2-17　双链表示意图

双链表和单链表相比，每一个节点增加了一个指针域，虽然多占用了空间，但它给数据运算带来了方便。在双链表中，如果只涉及单向的指针，其运算与单链表的算法一致。如果运算涉及两个方向的指针，由于双链表的对称结构，其插入和删除操作都很容易。双链表有一个重要的特点是：若 p 是指向表中任一节点的指针，则有

```
    typedef int DataType;      /* DataType 为数据元素的类型，可以是任何类型的数据 */
    typedef struct Dnode       /* 节点类型定义 */
    {
```

```
    DataType data;          /* 节点的数据域 */
    struct Dnode *prior;    /* 节点的前驱指针域 */
    struct Dnode *next;     /* 节点的后继指针域 */
}DulLinkList;
```

1）插入。在双向链表中 p 指针指向的节点前插入新节点 s 操作。先创建一个以 *x* 为值的新节点 s，在 p 节点之前插入节点 s，则插入操作步骤如下：

第一步：将节点 s 的 prior 域指向节点 p 的前一个节点（执行语句：s->prior=p->prior）。

第二步：将节点 p 的前一个节点的 next 域指向节点 s（执行语句：p->prior->next=s）。

第三步：将节点 s 的 next 域指向 p 节点（执行语句：s->next=p）。

第四步：将节点 p 的 prior 域指向节点 s（执行语句：p->prior=s）。

具体如图 2-18 所示。

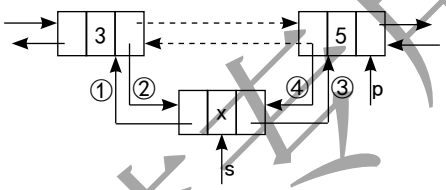


图 2-18 双链表插入数据示意图

双向循环链表的节点插入算法如下：

```
void DuIns_Elem(DuLinkList *p,DataType x)
{   /* 双向循环链表的插入节点函数 */
    DuLinkList *s;
    s=( DuLinkList *)malloc(sizeof(DuLinkList)); /* 生成新节点 s*/
    s->data=x;
    s->prior=p->prior;          /* 对应图 2-12 中的① */
    p->prior->next=s;           /* 对应图 2-12 中的② */
    s->next=p;                  /* 对应图 2-12 中的③ */
    p->prior=s;                 /* 对应图 2-12 中的④ */
}
```

2）在双向链表中删除 p 指针指向的节点操作。先要保证删除位置的正确性。在双向链表上找到删除位置节点地址，由 p 指向，先将节点 p 中的数据域中的值赋给指针变量 **x*，则删除，其操作步骤如下：

第一步：将节点 p 前一个节点的 next 域指向节点 p 的 next 域（执行语句：p->prior->next=p->next）。

第二步：将节点 p 后一个节点的 prior 域指向节点 p 的 prior 域（执行语句：p->next->prior=p->prior）。

第三步：释放节点 p 空间（执行语句：free(p)）。

删除节点的过程如图 2-19 所示。

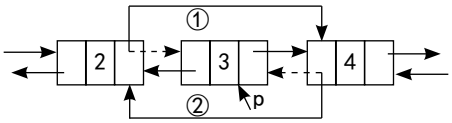


图 2-19　删除节点 p 示意图

双向循环链表的节点删除算法如下：

```
void DuDel_Elem(DuLinkList *p, DataType *x)
{  /* 双向循环链表的删除节点函数 */
   *x=p->data;
   p->prior->next=p->next;       /* 对应图 2-16 中的 a*/
   p->next->prior=p->prior;      /* 对应图 2-16 中的 b*/
   free(p);
}
```

2.3　顺序表和链表的比较

本章介绍了线性表的两种存储结构，顺序表和链表。顺序存储有以下优点：

（1）方法简单，高级语言都提供数组，实现容易。

（2）无须为表示线性表中元素的逻辑关系而额外增加存储开销。

（3）具有按元素序号随机访问的特点。

顺序存储的缺点如下：

（1）在顺序表中进行插入、删除操作时，需要顺序移动元素（平均移动次数是线性表长度的一半），因此对长度较大的线性表操作效率较低。

（2）要预先分配足够大的存储空间，空间分配过大会造成浪费，过小又有可能导致一些操作（如插入）失败。

顺序表的优点就是链表的缺点，而其缺点就是链表的优点。

在实际使用中空间采用什么存储结构呢？通常考虑以下三方面。

1. 基于空间的考虑

当线性表长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采

用顺序表作为存储结构。当线性表的长度变化较大，事先难以确定期限大小时，应采用链表存储结构。

2. 基于时间的考虑

基线性表的操作主要是进行查找，很少进行插入和删除时，采用顺序表为存储结构较好；对于频繁进行插入和删除的线性表，采用链表较好。

3. 基于语言的考虑

各种高级语言都提供有数组，但不一定提供指针，链表是基于指针的，所以链表的使用受高级语言的限制。

2.4 项 目 训 练

项目一：通信录管理

【题目要求】

利用链表结构解决通信录的使用问题，通过菜单实现其常用的几种功能：

1. 菜单内容

程序运行后，给出 6 个菜单项的内容和输入提示：

（0）退出管理系统。

（1）通信录链表的建立。

（2）通信录节点的插入。

（3）通信录节点的查询。

（4）通信录节点的删除。

（5）通信录节点的输出。

请选择 0~5：

2. 设计要求

使用数字 0~5 来选择菜单项，执行相应的操作，其他输入则不起作用。

【算法分析】

按照单链表的结构，设计单链表的建立、节点的插入、节点的查找、节点的删除和链表的输出等相应的子函数。为实现菜单的调用功能，需要设计一个函数用于输出提示信息和处理输入，并将返回值提供给主函数实现相应的子函数调用。

项目二：纸牌游戏

【题目要求】

（纸牌游戏）编号为 1~52 张牌，正面向上，从第二张开始，以 2 为基数，是 2 的倍数的牌翻一次，直到最后一张牌；然后从第三张牌开始，以 3 为基数，是 3 的倍数的牌翻一次，直到最后一张牌；直到以 52 为基数的翻过，输出：这时输出正面向上的牌有哪些？

【算法分析】

由于翻牌时有正面向上的牌，也有反面向上的牌，而且最后要计算正面向上的牌数，所以设置了标志位 key 来表示正面向上的牌的编号，key=1 表示正面向上，未翻牌的一直是 key=1; 经过翻牌就 key=0，此时表示经过翻牌，属于是反面向上。设计的单链表的存储结构为：

每个节点应包括三个域：存储该节点所对应的牌的编号信息 data 域、表示牌的正反面的标志的 key 域、存储其直接后继的存储位置的 next 域（指针域）。

可以使用尾插法建立单链表。因为编号为 1~52 的 52 张牌不一定存储在相邻的存储单元中，且在翻牌操作时要对编号依次进行判断，用指针来进行操作很简单。选用单链表这种数据结构来对 52 张牌进行链接存储。该链表中的每一个节点只有一个指针域。将编号为 1 到 52 的节点依次链接起来，就形成了利用尾插法建立单链表的操作。

本章小结

（1）线性表是一种具有一对一的线性关系的特殊数据结构。线性表有两种存储方法：用顺序存储方法来表示这种线性关系，得到顺序存储结构（即顺序表）；用链式存储方式来表示这种线性关系，得到线性表的链式存储结构（即链表）。

（2）线性表的链式存储结构，是通过节点之间的链接而得到的，链式存储结构有单链表、双向链表和循环链表等。

（3）单链表节点至少有两个域：一个数据域和一个指针域。双向链表节点至少含有三个域：一个数据域和两个指针域。

（4）循环链表不存在空指针，最后一个节点的指针指向表头，形成一个首尾相接的环。

（5）为了处理问题方便，在链表中增加一个头节点。

（6）顺序存储可以提高存储单元的利用率，不便于插入和删除运算。链式

存储会占用较多的存储空间，可以使用不连续的存储单元，插入、删除运算较方便。

习 题

一、选择题

1. 下面关于线性表的叙述中，错误的是（ ）。

A. 线性表采用顺序存储，必须占用一片连续的存储单元。

B. 线性表采用顺序存储，便于进行插入和删除操作。

C. 线性表采用链接存储，不必占用一片连续的存储单元。

D. 线性表采用链接存储，便于插入和删除操作。

2. 两个指针 P 和 Q，分别指向单链表的两个元素，P 所指元素是 Q 所指元素前驱的条件是（ ）。

A. P->next == Q->next　　B. P->next == Q

C. Q->next == P　　D. P == Q

3. 在单链表中，增加头节点的目的是（ ）。

A. 使单链表至少有一个节点　　B. 标志表中首节点的位置

C. 方便运算的实现　　D. 说明该单链表是线性表的链式存储结构

4. 在（ ）的运算中，使用顺序表比链表好。

A. 插入　　B. 根据序号查找

C. 删除　　D. 根据元素查找

5. 在单链表指针为 p 的节点之后插入指针为 s 的节点，正确的操作是（ ）。

A. p->next=s;s->next=p->next;

B. s->next=p->next;p->next=s;

C. p->next=s;p->next=s->next;

D. p->next=s->next;p->next=s;

6. 在一个长度为 n 的顺序表中，若要删除第 i（$1 \leqslant i \leqslant n$）个元素，则需向前移动（ ）个元素。

A. $n-i+1$　　B. $n-i-1$　　C. $n-i$　　D. i

7. 在一个长度为 n 的顺序表中，若要在第 i（$1 \leqslant i \leqslant n$）个元素前插入一个元素时，则需向后移动（ ）个元素。

A. $n-i+1$　　B. $n-i-1$　　C. $n-i$　　D. i

二、填空题

1. 线性表 $L=(a_1, a_2, \cdots, a_n)$ 采用顺序存储，假定删除表中任意元素的概率相同，则删除一个元素平均需要移动元素的个数是_____。

2. 顺序表相对于链表的优点是：_____和随机存取；链表相对于顺序表的优点是：_____方便。

3. 在单链表中要在已知节点 *P 之前插入一个新节点，需找到 *P 的直接前趋节点的地址，其查找的时间复杂度为_____。

4. 在长度为 n 的顺序表中，如果要在第 i 个元素前插入一个元素，要后移_____个元素。

三、综合题

1. 设计一个算法，将带头的单链表逆置。

2. 设计一个算法，删除顺序表中值为 x 的所有节点。

3. 用链表结构存储多项式，求两个多项式 A 加 B 的和。要求在建立多项式链表时，总是按照指数从大到小排序。

第 3 章　特殊线性表

特殊线性表主要包括栈、队列和串，其中栈又称为堆栈，是一种特殊的线性结构，它的插入、删除等操作只能在表的一端进行，其特点是“先进后出”的原则进行操作。队列也是一种运算受限制的线性表，与栈不同的是：队列是限制在表的两端进行操作的线性表，也是软件设计中常用的一种数据结构，队列的特点是按“先进先出”的原则进行操作。串是字符串的简称，它的每个数据元素都由一个字符组成。本章重点学习栈、队列和串这个特殊线性表的各种操作。

知识目标

- 理解特殊线性表的逻辑结构特征。
- 掌握栈的含义、特点、基本运算和相关算法分析。
- 掌握队列的含义、特点、基本运算和相关算法分析。
- 掌握串的含义、特点、基本运算和相关算法分析。

能力目标

- 能应用栈的理论设计算法，解决实际问题。
- 能应用队列的理论设计算法，解决实际问题。
- 能应用串的理论设计算法，解决实际问题。

3.1 栈

3.1.1 栈的逻辑结构

1. 栈的定义

栈(Stack)是运算受限的线性表，只允许在表的一端进行插入和删除操作。允许插入和删除的一端称为栈顶（top），栈顶将随着栈中数据元素的插入、删除而动态变化，通过栈顶指针表示当前数据元素的位置。另一端称为栈底（bottom），栈底是固定的。当表中没有数据元素时，称为空栈。

设有一个栈 $S=\{a_1, a_2, \cdots, a_n\}$，栈中元素按 $a_1, a_2, \cdots, a_n$ 的次序进栈，按 $a_n, \cdots, a_2, a_1$ 的顺序出栈。进栈的第一个元素 a_1 为栈底元素，出栈的第一个元素 a_n 为栈顶元素。这种后进先出的线性结构称为栈（Stack）。栈的操作是按照“后进先出”（Last In First Out）的原则进行的，如图 3-1 所示。

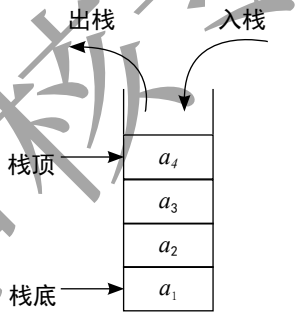


图 3-1 栈结构示意图

2. 栈的几个术语

（1）InitStack(&S)：构造一个空栈，即栈的初始化。

（2）StackEmpty(&S)：判栈空。若栈为空，返回 1；否则，返回 0。

（3）StackFull(&S)：判栈满。若栈满，返回 1；否则，返回 0。

（4）Push(&S，x)：入栈。将元素 x 插入为新的栈顶元素。

（5）Pop(&S，&x)：出栈。删除栈 S 的栈顶元素，用 x 返回栈顶元素值。

（6）GetTop(&S，&x)：取栈顶元素。用 x 返回栈顶元素值。

3. 应用实例

（1）建筑工地码砖。例如，在建筑工地上，使用的砖块从底往上一层一层地码放，在使用时，将从最上面一层一层地拿取。

（2）分币筒。公交车上的售票员用的分币筒就是一个最典型的栈。多个分币依次进入分币筒后，只能按照后进先出的次序退出分币筒。

3.1.2　顺序栈

1. 顺序栈定义

栈的顺序存储结构称为顺序栈（Sequential Stack），它是利用一组地址连续的存储空间依次存放从栈底到栈顶的数据元素。与顺序表类似，顺序栈采用一维数组实现。可以采用数组的任意一端作为栈底，通常设定数组中下标为 0 的一端作为栈底，同时设定指针 top 用于指示当前栈顶的位置。

顺序栈的类型定义如下：

```
#define MAXLED 100      /* 设定栈的长度，可根据实际问题具体定义 */
typedef int DataType;
typedef struct
{
 DataType  data[MAXLED];
 int top;                /* 设定栈顶指针 */
}SeqStack;
```

顺序栈操作示意图如图 3-2 所示，栈顶指针动态反映了栈中元素的变化情况，通常 0 下标端设为栈底。

（1）当 top=-1 时，表示栈空，如图 3-2(a) 所示。

（2）当 top=0 时，表示栈中有一个元素，如图 3-2(b) 表示栈中已输入一个元素 a_1。

（3）入栈时，栈顶指针上移，指针 top 加 1，如图 3-2(c) 是 4 个元素入栈后的状况。

（4）出栈时，栈顶指针下移，指针 top 减 1，如图 3-2(d) 是在 a_4，a_3 元素相继出栈后的情况。此时栈中还有 a_1，a_2 两个元素，top=1，指针已经指向了新的栈顶。但是出栈后的元素仍然还在原来的存储单元中，只是不在栈中了，因为栈是只能在栈顶进行操作的线性表。

（5）当 top=5 时，也即 top=MAXLEN-1，表示栈满，如图 3-2(e) 所示。

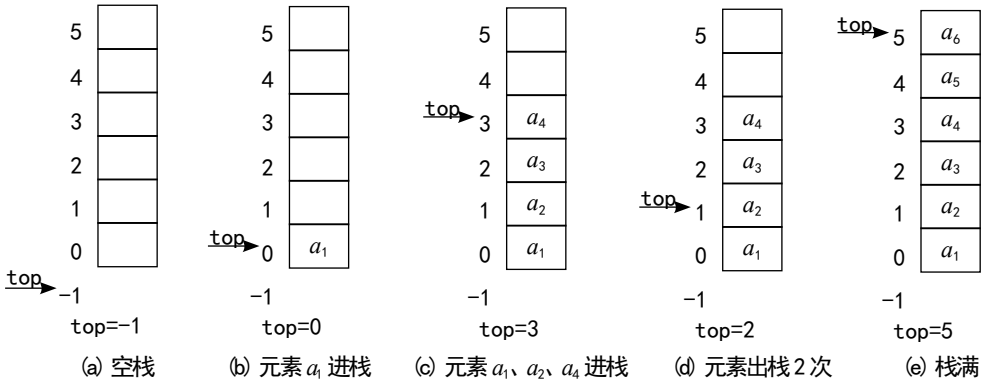


图 3-2 顺序栈中栈顶指针与栈中元素之间的关系

在栈的操作过程中，栈满时再做进栈操作会出现空间溢出，称为上溢，这是一种出错状态，应该避免。在栈空时再做出栈操作产生的溢出，称为下溢，这是正常现象，是程序控制转移的条件。

2. 顺序栈的基本操作实现

（1）初始化栈操作。首先创建一个空栈，然后将栈顶下标 top 初始化为 -1。其算法描述如下：

```
void  InitStack( SeqStack  *S)
{  /* 初始化栈函数 */
   S->top=-1;                    /* 初始化的顺序栈为空 */
}
```

（2）判断栈空操作。判断是否是空栈（即 *S*->top==-1），若栈 *S* 为空，则返回 1；否则返回 0。其算法描述如下：

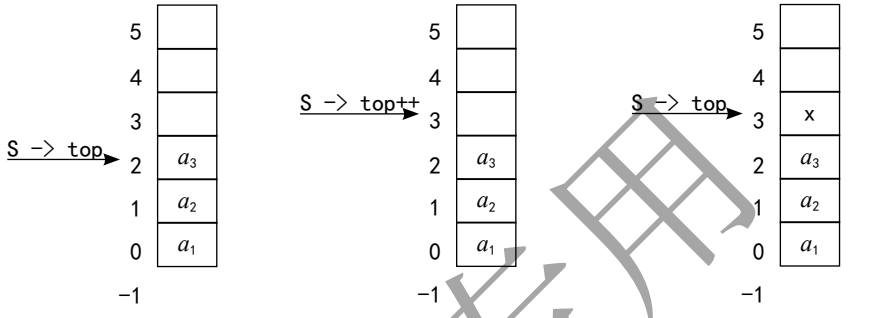
```
int EmptyStack(SeqStack *S)
{  /* 判断栈空函数 */
   if(S->top==-1)                /* 栈为空 */
      return 1;
   else
      return 0;
}
```

（3）判断栈满。判断是否是满栈（即 S->top==MAXLEN-1），若栈 *S* 为满，则返回 1；否则返回 0。其算法描述如下：

```
int FullStack(SeqStack *S)
{  /* 判断栈满函数 */
```

```
    if(S->top==MAXLEN-1)                    /* 栈为满 */
        return 1;
    else
        return 0;
}
```

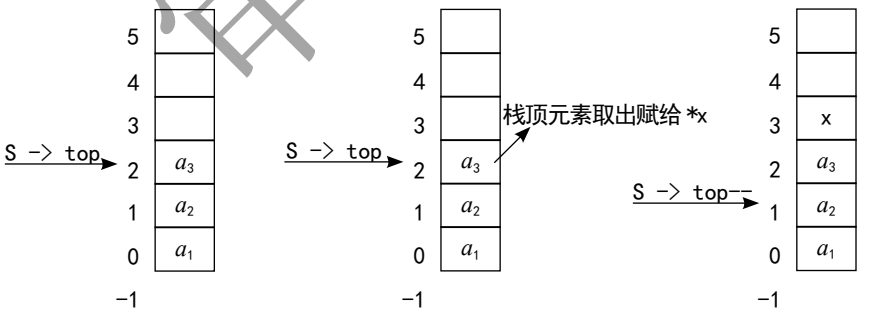
（4）进栈操作。进栈操作的过程如图 3-3 所示。先判断栈 S（见图 3-3(a)）是否为满，若不满再将记录栈顶的下标变量 top 加 1 如图 3-3(b) 所示，最后将进栈元素放进栈顶位置上如图 3-3(c) 所示。



(a) 进栈前栈 S 图　　(b) 栈顶指针增 1 后图　　(c) 进栈后栈 S 图

图 3-3　进栈操作过程图

（5）出栈操作。出栈操作的过程如图 3-4 所示。先判断栈 S（见图 3-4(a)）是否为空，若不空将栈顶元素取出赋给指针 x 所指的对象，如图 3-4(b) 所示，然后将记录栈顶的下标变量 top 减 1（但该元素还在数组内，只是栈顶指针已经改变位置），如图 3-4(c) 所示。



(a) 出栈前栈 S 图　　(b) 将栈顶元素赋给 *x 图　　(c) 出栈后栈 S 图

图 3-4　出栈操作过程图

```
/* 顺序栈操作 */
#include "stdio.h"
```

```
#define  MAXLEN  100              /* 顺序栈存储空间的总分配量 */
typedef  int  DataType;           /* 定义 DataType 为 int 类型 */
typedef  struct                   /* 顺序栈存储类型 */
{  DataType  data[MAXLEN];        /* 存放顺序栈的数组 */
   int  top;                      /* 记录栈顶元素位置的变量 */
}SeqStack;

void  InitStack( SeqStack  *S )
{  /* 初始化栈函数 */
   S->top=-1;                     /* 初始化的顺序栈为空 */
}

int EmptyStack(SeqStack *S)
{  /* 判断栈空函数 */
   if(S->top==-1)                 /* 栈为空 */
      return 1;
   else
      return 0;
}

int FullStack(SeqStack *S)
{  /* 判断栈满函数 */
   if(S->top==MAXLEN-1)                  /* 栈为满 */
      return 1;
   else
      return 0;
}

int Push(SeqStack *S,DataType x)
{  /* 进栈操作函数 */
   if(FullStack(S))          /* 调用判满函数 FullStack(S)，判断栈是否为满 */
   {  printf(" 栈满，不能进栈 !");
      return 0;                  /* 栈满不能进栈 */
   }
   else                          /* 栈不为满 */
   {  S->top++;
      S->data[S->top]=x;
      return 1;
```

```
    }
}

int Pop(SeqStack *S,DataType *x)
{  /* 出栈操作函数 */
   if(EmptyStack(S))                /* 调用判空函数 EmptyStack(S)，判断栈
是否为空 */
   {  printf(" 栈空，不能出栈 !");
      return 0;                     /* 栈空不能出栈 */
   }
   else                             /* 栈不为空 */
   {  *x=S->data[S->top];
      S->top--;
      return 1;
   }
}

int GetTop(SeqStack *S,DataType *x)
{  /* 取栈顶元素函数 */
   if(EmptyStack(S))                /* 调用判空函数 EmptyStack(S)，判断栈
是否为空 */
   {  printf(" 栈空，取栈顶元素失败 !");
      return 0;
   }
   else                             /* 栈不为空 */
   {  *x=S->data[S->top];
      return 1;
   }
}

main()
{
  int i,n,flag;
  SeqStack S;
  DataType x;
  InitStack(&S);
  printf(" 请输入要入栈的元素个数：");
  scanf("%d",&n);
```

```
    printf(" 请输入 %d 个入栈的整数：",n);
    for(i=0;i<n;i++)
    {
       scanf("%d",&x);
       flag=Push(&S,x);
    }
    if(flag==1)
    printf(" 请输入要出栈的元素个数：");
    scanf("%d",&n);
    printf(" 出栈的元素为：");
    for(i=0;i<n;i++)
    {
       flag=Pop(&S,&x);
       printf("%5d",x);
    }
    printf("\n");
    if(flag=GetTop(&S,&x))
       printf(" 当前的栈顶元素值为：%d",x);
}
```

程序运行结果如图 3-5 所示。

```
请输入要入栈的元素个数：5
请输入 5 个入栈的整数：1 2 3 4 5
请输入要出栈的元素个数：2
出栈的元素为：    5    4
当前的栈顶元素值为：3Press any key to continue
```

图 3-5　入栈出栈运行结果图

3.1.3　链栈

1. 链栈的定义

栈的链式存储结构称为链栈（Linked Stack）。链栈是运算受限的单链表，它的插入和删除操作都限定在表头位置上进行。栈顶指针就是链表的头指针，它唯一确定一个链栈。

链栈可以有头节点，也可以没有头节点。因为以单链表的头部做栈顶是最方便的，所以通常选用无头节点的单链表表示链栈，如图 3-6 所示。

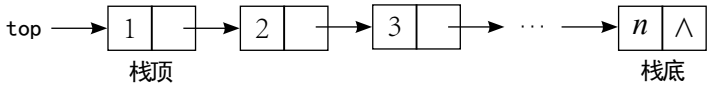


图 3-6　链栈结构示意图

2. 链栈的类型定义

与单链表的类型定义相同，在此用 LinkStack 命名。链栈的类型定义如下：

```
#define  MAXLEN  100              /* 数组最大长度为 100*/
typedef  int  DataType;           /* 定义 DataType 为 int 类型 */
typedef  struct  stacknode                /* 链栈存储类型 */
{
  DataType   data;                /* 定义节点的数据域 */
  struct  stacknode   *next;      /* 定义节点的指针域 */
} LinkStack;
```

3. 链栈的基本操作实现

（1）初始化栈操作。首先创建一个空栈，语句 S=NULL 将链栈类型的变量 *S* 标识为空，返回栈项指针 *S*。其算法描述如下：

```
LinkStack  *InitStack()
{  /* 初始化链栈函数 */
  LinkStack *S;
  S=NULL;
  return S;                                /* 初始化栈为空 */
}
```

（2）判断栈空操作。判断是否是空栈（即 S==NULL），若栈 *S* 为空，则返回 1；否则返回 0。其算法描述如下：

```
int EmptyStack(LinkStack *S)
{  /* 判断栈空函数 */
  if(S==NULL)                      /* 栈为空 */
     return 1;
  else
     return 0;
}
```

（3）进栈操作。先创建一个以 *x* 为值的新节点 p，其 data 域值为 *x*，则进栈操作步骤如下：

1）将新节点 p 的指针域指向原栈顶 S（执行语句：p->next=S）；

2）将栈顶 S 指向新节点 p（执行语句：S=p）。

进栈操作的过程如图 3-7 所示。

注意：进栈操作的①与②语句执行顺序不能颠倒，否则原 S 指针其后的链表将丢失。

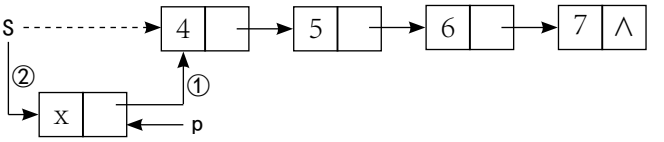


图 3-7　进栈操作过程图

进栈算法描述如下：

```
LinkStack *Push(LinkStack *S,DataType x)
{ /* 进栈函数 */
  LinkStack *p;
  p=(LinkStack *)malloc(sizeof(LinkStack));     /* 生成新节点 */
  p->data=x;    /* 将 x 放入新节点的数据域 */
  p->next=S;（①）          /* 将新节点插入链表表头之前 */
  S=p;    （②）            /* 新节点作为栈顶 */
  return S;                  /* 返回栈顶 S*/
}
```

（4）出栈操作。先将节点栈顶 q 数据域中的值赋给指针变量 *x，则删除操作步骤如下：

1）节点 p 指针域指向原栈顶 S（执行语句：p=S）。

2）栈顶 S 指向其下一个节点（执行语句：S=S->next）。

3）释放 p 节点空间（执行语句：free(p)）。出栈的过程如图 3-8 所示。

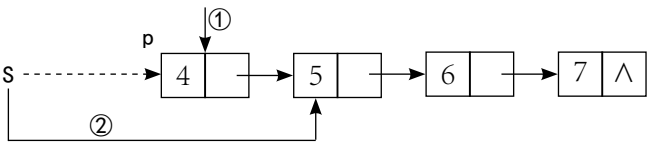


图 3-8　出栈操作过程图

```
LinkStack *Pop(LinkStack *S,DataType *x)
{ /* 出栈函数 */
  LinkStack *p;
  if(EmptyStack(S))        /* 调用判空函数 EmptyStack(S)，判断栈是否为空 */
```

```
  {
     printf("\t 栈空，不能出栈 !");
     return NULL;          /* 栈空不能出栈 */
  }
  else                     /* 栈不为空 */
  {
    *x=S->data;            /* 栈顶元素取出赋给 x*/
    p=S;  （①）            /*p 节点指向原栈顶 S*/
    S=S->next;  （②）      /* 原栈顶 S 指向其下一个节点 */
    free(p);               /* 释放原栈顶空间 */
    return S;              /* 返回栈顶 S*/
  }
}
```

链栈的综合操作，程序代码如下：

```
#include <stdio.h>
#include <malloc.h>
#define MAXLEN 100    /* 数组最大长度为 100*/
typedef int DataType;   /* 定义 DataType 为 int 类型 */
typedef struct stacknode                /* 链栈存储类型 */
{
  DataType      data;       /* 定义节点的数据域 */
  struct stacknode  *next;        /* 定义节点的指针域 */
} LinkStack;

LinkStack *InitStack()
{ /* 初始化链栈函数 */
  LinkStack *S;
  S=NULL;
  return S;                          /* 初始化栈为空 */
}

int EmptyStack(LinkStack *S)
{ /* 判断栈空函数 */
  if(S==NULL)              /* 栈为空 */
    return 1;
  else
    return 0;
```

```
    }

    LinkStack *Push(LinkStack *S,DataType x)
    { /* 进栈函数 */
      LinkStack *p;
      p=(LinkStack *)malloc(sizeof(LinkStack));/* 生成新节点 */
      p->data=x;                /* 将 x 放入新节点的数据域 */
      p->next=S;                /* 将新节点插入链表表头之前 */
      S=p;                      /* 新节点作为栈顶 */
      return S;                         /* 返回栈顶 S*/
    }

    LinkStack *Pop(LinkStack *S,DataType *x)
    { /* 出栈函数 */
      LinkStack *p;
      if(EmptyStack(S))         /* 调用判空函数 EmptyStack(S)，判断栈是否
为空 */
      {
         printf("\t 栈空，不能出栈 !");
         return NULL;           /* 栈空不能出栈 */
      }
      else                      /* 栈不为空 */
      {
        *x=S->data;             /* 栈顶元素取出赋给 x*/
        p=S;                    /*p 节点指向原栈顶 S*/
        S=S->next;              /* 原栈顶 S 指向其下一个节点 */
        free(p);                /* 释放原栈顶空间 */
        return S;               /* 返回栈顶 S*/
      }
    }

    int GetTop(LinkStack *S,DataType *x)
    { /* 获取栈顶元素函数 */
      if(EmptyStack(S))         /* 调用判空函数 EmptyStack(S)，判断栈是否
为空 */
      {
         printf(" 栈空 !");
         return 0;
```

```
    }
    else                        /* 栈不为空 */
    {
      *x=S->data;               /* 栈顶元素赋给变量 x*/
      return 1;
    }
}

main()
{
  int i,n,flag;
  LinkStack *S;
  DataType x;
  S=InitStack();
  printf(" 请输入要入栈的元素个数：");
  scanf("%d",&n);
  printf(" 请输入 %d 个整数进行入栈：",n);
  for(i=0;i<n;i++)
  {
     scanf("%d",&x);
     S=Push(S,x);
  }
  printf(" 请输入要出栈的元素个数：");
  scanf("%d",&n);
  printf(" 出栈的元素为：");
  for(i=0;i<n;i++)
  {
     S=Pop(S,&x);
     if(S!=NULL)
        printf("%5d",x);
  }
  printf("\n");
  if(flag=GetTop(S,&x))
  printf(" 当前的栈顶元素值为：%d",x);
}
```

程序运行结果如图 3-9 所示。

```
请输入要入栈的元素个数：5
请输入 5 个整数进行入栈：1 2 3 4 5
请输入要出栈的元素个数：3
出栈的元素为：      5     4     3
当前的栈顶元素值为：2Press any key to continue
```

图 3-9　链栈运行结果图

3.2　队　　列

3.2.1　队列的逻辑结构

1. 队列的定义

队列（Queue）是一种先进先出（First In First Out）的线性表，简称 FIFO 表。它只允许在表的一端进行插入，而在另一端进行删除。与栈类似，队列也是一种受限的线性表。这与我们日常生活中常见的排队是一致的，先到的人排在队伍前端，最先离开，新来的人总是排在队伍的最后。

队列在计算机中也有着广泛的应用，比如当需要打印多篇文档时，我们在文本编辑器中按一定的先后顺序，对文本选择了“打印”处理后，就可以将文档关闭了。系统会将这些要打印的文档按设定的先后顺序存放在打印缓冲区，逐一送入打印机打印。

在队列中，允许插入的一端叫队尾（rear），允许删除的一端叫队首或队头(front)。在队列的队尾插入一个数据元素的操作叫入队或进队，在队首删除一个数据元素的操作叫出队或退队。假设队列为 $q=\{a_1, a_2, a_3, \cdots, a_n\}$，其中 a_1 就是队首元素，a_n 就是队尾元素。入队时按照 a_1，a_2，a_3，…，a_n 的顺序进入，出队时也按照相同的顺序退出。如图 3-10 所示。

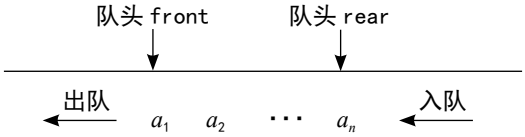


图 3-10　队列结构示意图

2. 队列的抽象数据类型定义

队列的抽象数据类型表示了队列中的数据对象、数据关系以及基本操作，定义如下：

ADT Queue{

数据对象：D={a_i|a_i 为 DataType 类型，$1 \leqslant i \leqslant n$，$n \geqslant 0$}　/*DataType 为自定义类型 */

数据关系：R={<a_{i-1}，a_i>|a_{i-1}，$a_i \in$ D，$2 \leqslant i \leqslant n$ }。在非空表中，除了首节点，每个节点都有且只有一个前驱节点；除了尾节点外，每个节点都有且只有一个后继节点。队列中的数据元素只能在队尾一端进行插入，在队首一端进行删除操作。它的主要特征如下：

（1）队列的主要特性就是“先进先出”，常用它的英文缩写表示，称为 FIFO 表。

（2）队列也是一种特殊的线性表。插入操作限定在线性表的一端进行，删除操作限定在线性表的另一端进行。

队列在日常生活中经常使用，如平时人们排队上公共汽车，排队的规则是不允许“插队”。后来的人需站在队尾，每次总是队头先上车。

3. 队列的基本操作

队列除了在队头进行出队和队尾进行入队外，还有初始化、判空等操作，常用的基本操作：

（1）初始化队列 InitQueue(Q)，构造一个空队列 Q。

（2）判断队列空 EmptyQueue(Q)，判断是否是空队列，若队列 Q 为空，则返回 1；否则返回 0。

（3）入队 InQueue(Q，x)，将数据元素 x 插入队列 Q 的队尾，使其为队列 Q 的队尾元素。

（4）出队 DeQueue(Q，x)，将队头元素赋给 x，并从队列 Q 中删除当前队头元素，而其后继元素成为队头元素。

（5）取队头元素 GetFront(Q，x)，将队头元素赋给 x 并返回，操作结果只是读取队头元素，队列 Q 不发生变化。

（6）显示队列元素 ShowQueue(Q)，将队列中所有元素按从队头开始依次输出。

3.2.2 顺序队列

队列的顺序存储结构称作顺序队列（Sequential Queue）。顺序队列与顺序表类似，采用一个一维数组存放数据元素。由于队列的队首和队尾位置要随着出队、入队操作不断变化，因此设置两个指针 front 和 rear 分别用来指向当前队首和队尾。

顺序队列的类型定义如下：

```
#define  MAXLEN  100              /* 顺序队列存储空间的总分配量 */
typedef  int  DataType;           /* 定义 DataType 为 int 类型 */
typedef  struct                   /* 顺序队列存储类型 */
{  DataType  data[MAXLEN];        /* 存放顺序队列的数组 */
   int  front;                    /* 记录队头元素位置的变量 */
   int  rear;                     /* 记录队尾元素位置的变量 */
}SeqQueue;
```

通常情况下，队列中的几种状态如图 3-11 所示。

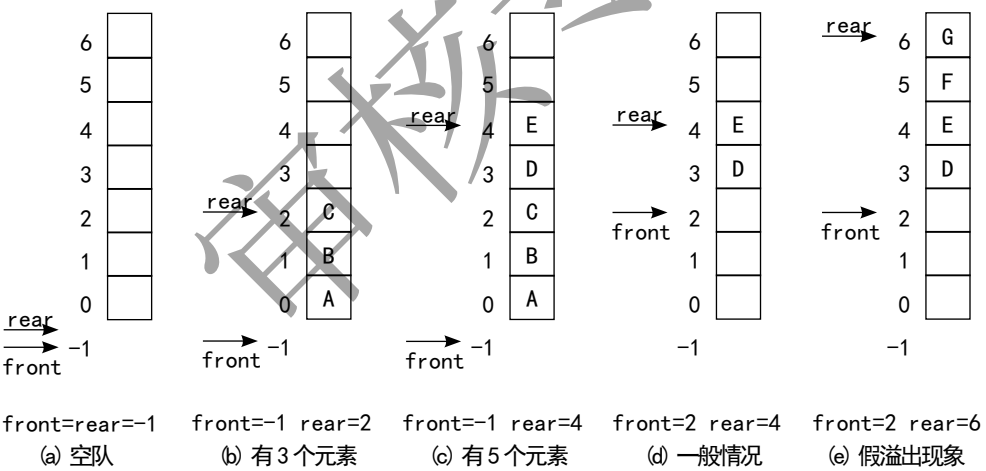


图 3-11 顺序队列中的几种状态

假设图 3-11 中的最大存储空间为 MAXLEN=7，则可以看出：当队列初始化时 front=rear= -1，队列为空如图 3-11(a) 所示；当 rear=MAXLEN-1=6 时，再加入新元素，数组越界而导致程序发生异常的错误，如图 3-11(e) 所示。但此队列为空或有剩余存储单元，这种现象称为“假溢出”。那又如何解决“假溢出”现象呢？请参阅 3.2.3 节的循环队列。

3.2.3　循环队列

1. 循环队列的定义

为了避免顺序队列的“假溢出”现象，我们可以把顺序队列想象成首尾相连的环，当队首指针和队尾指针达到数组下界时，能延续到数组上界，构成循环队列（Circle Queue）。例如，在图 3-12(b) 所示的循环队列示意图最大空间为 MAXLEN =8，数组下标为 0~7 之间。图 3-12(a) 中队头与队尾指针指向同一位置时为空队，非空队时如图 3-12(b) 中队头指针 front 指向队列中队头元素的前一个位置，队尾指针 rear 指向队列的队尾元素位置。

假设队列开辟的数组单元数为 MAXLEN，它的数组下标在 0~MAXLEN -1 之间，若使队头或队尾增 1，可以利用取模运算实现。

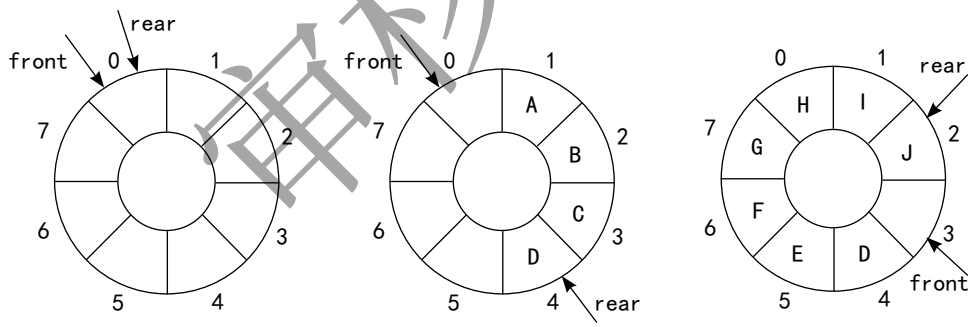
入队时的队尾指针加 1 操作修改为

$$rear=(rear+1)\%\ MAXLEN$$

出队时的队头指针加 1 操作修改为

$$front=(front+1)\%\ MAXLEN$$

循环队列解决了顺序队列的“假溢出”现象，但新的问题出现了：队满的条件不再是 rear== MAXLEN -1，而是 front==rear 和队空的条件相同了。



(a) 空队　　(b) 非空队　　(c) 满队

图 3-12　循环队列示意图

其解决方法为：损失一个单元不用，即当循环队列中元素的个数是 MAXLEN -1 时就认为队列已满。

如图 3-12(c) 所示为元素 A、B、C 出队，元素 E、F、G、H、I、J 依次入队后队满的情况。此时 front=3，rear=2，两者比较的结果为 (rear+1)% MAXLEN ==front（即 (2+1)%8==3），所以循环队列为满。

这样循环队列的队满条件就变成：

$$(rear+1)\%\ MAXLEN == front$$

队空条件不变为

$$front == rear$$

2. 循环队列的基本操作实现

（1）初始化队列操作。首先创建一个空队列，并将队头与队尾指针初始化为 0。其算法描述如下：

```
void InitQueue(SeqQueue *Q)
{  /* 初始化队列函数 */
   Q->front=Q->rear=0;            /* 指针初始化 */
}
```

（2）判断队列空操作。判断是否是空队列（即 Q->front == Q->rear），若队列 *Q* 为空，则返回 1；否则返回 0。其算法描述如下：

```
int EmptyQueue (SeqQueue *Q)
{  /* 判断队空函数 */
   if(Q->front==Q->rear)          /* 队列为空 */
      return 1;
   else
      return 0;
}
```

（3）入队操作。首先判断队列 *Q* 是否已满，若不满再将队中的记录队尾的下标变量 rear 加 1，最后将入队元素 x 放进队尾位置上。程序代码如下：

```
#include "stdio.h"
#define MAXLEN 100                /* 顺序队列存储空间的总分配量 */
typedef int DataType;             /* 定义 DataType 为 int 类型 */
typedef struct                    /* 顺序队列存储类型 */
{  DataType data[MAXLEN];         /* 存放顺序队列的数组 */
   int front;                     /* 记录队头元素位置的变量 */
   int rear;                      /* 记录队尾元素位置的变量 */
}SeqQueue;

void InitQueue(SeqQueue *Q)
{  /* 初始化队列函数 */
    Q->front=Q->rear=0;           /* 指针初始化 */
```

```
}

int InQueue(SeqQueue  *Q,DataType x)
{  /* 入队函数 */
    if((Q->rear+1)%MAXLEN==Q->front)                    /* 队列为满 */
   {  printf(" 队满，不能入队元素 !");
      return 0;                  /* 队满不能入队 */
   }
   else                          /* 队不为满 */
   {  Q->rear=(Q->rear+1)%MAXLEN;    /* 队尾指针增 1*/
      Q->data[Q->rear]=x;
      return 1;
    }
}

void ShowQueue(SeqQueue  *Q)
{  /* 显示队中元素函数 */
    int p=Q->front;
    if(p==Q->rear)
       printf(" 队列为空，无元素！ \n");
    else
    {
            printf(" 从队头起队列中的各元素为：");
            while(p!=Q->rear)
            {
                  printf("%5d",Q->data[p+1]);
               p++;
            }
    }
}

main()
{
  int i,n,flag;
  SeqQueue Q;
  DataType x;
  InitQueue(&Q);
```

```
    printf(" 请输入要入队的元素个数：");
    scanf("%d",&n);
    printf(" 请输入 %d 个入队的整数：",n);
    for(i=0;i<n;i++)
    {
       scanf("%d",&x);
       flag=InQueue(&Q,x);
    }
    if(flag==1)
       printf(" 入队操作成功！ ");
    else
       printf(" 入队操作失败！ ");
    printf("\n");
    ShowQueue(&Q);
}
```

程序运行结果如图 3-13 所示。

```
请输入要入队的元素个数：5
请输入 5 个入队的整数：1 2 3 4 5
入队操作成功！
从队头起队列中的各元素为：    1    2    3    4    5
```

图 3-13　入队列运行结果图

（4）出队操作。首先判断队列 Q 是否为空，若不空再将队中的记录队头的下标变量 front 加 1 后将队头元素取出赋给指针 x 所指的对象。程序代码如下：

```
/* 线性循环队列的各种操作 */
#include "stdio.h"
#define  MAXLEN  100              /* 顺序队列存储空间的总分配量 */
typedef  int  DataType;           /* 定义 DataType 为 int 类型 */
typedef  struct                   /* 顺序队列存储类型 */
{  DataType  data[MAXLEN];        /* 存放顺序队列的数组 */
   int  front;                    /* 记录队头元素位置的变量 */
   int  rear;                     /* 记录队尾元素位置的变量 */
}SeqQueue;

void  InitQueue(SeqQueue  *Q)
{  /* 初始化队列函数 */
```

```
    Q->front=Q->rear=0;            /* 指针初始化 */
}

int EmptyQueue (SeqQueue  *Q)
{  /* 判断队空函数 */
    if(Q->front==Q->rear)                    /* 队列为空 */
     return 1;
   else
     return 0;
}

int InQueue(SeqQueue  *Q,DataType x)
{  /* 入队函数 */
    if((Q->rear+1)%MAXLEN==Q->front)                /* 队列为满 */
   {  printf(" 队满，不能入队元素 !");
      return 0;                          /* 队满不能入队 */
   }
   else                                  /* 队不为满 */
   {  Q->rear=(Q->rear+1)%MAXLEN;              /* 队尾指针增 1*/
      Q->data[Q->rear]=x;
      return 1;
    }
}

int DeQueue(SeqQueue  *Q,DataType *x)
{  /* 出队函数 */
    if(EmptyQueue(Q))              /* 调用判空函数 EmptyQueue(Q)，判断队
列是否为空 */
   {  printf(" 队空，不能出队元素 !");
      return 0;                          /* 队空不能出队 */
   }
   else                                  /* 队不为空 */
   { Q->front=(Q->front+1)%MAXLEN;             /* 队头指针增 1*/
     *x=Q->data[Q->front];
     return 1;
   }
}
```

```
int GetFront(SeqQueue  *Q,DataType *x)
{  /* 取队头函数 */
    if(EmptyQueue (Q))              /* 调用判空函数 EmptyQueue(Q)，判断队
列是否为空 */
    {  printf(" 队空，无队头元素 !");
       return 0;
    }
    else                                /* 队不为空 */
    {  *x=Q->data[(Q->front+1)%MAXLEN];
       return 1;
    }
}

void ShowQueue(SeqQueue  *Q)
{  /* 显示队中元素函数 */
    int p=Q->front;
    if(p==Q->rear)
       printf(" 队列为空，无元素！ \n");
    else
    {
            printf(" 从队头起队列中的各元素为：");
            while(p!=Q->rear)
            {
                printf("%5d",Q->data[p+1]);
              p++;
            }
    }
}

main()
{
  int i,n,flag;
  SeqQueue Q;
  DataType x;
  InitQueue(&Q);
  printf(" 请输入要入队的元素个数：");
```

```
    scanf("%d",&n);
    printf(" 请输入 %d 个入队的整数：",n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&x);
        flag=InQueue(&Q,x);
    }
    printf(" 请输入要出队的元素个数：");
    scanf("%d",&n);
    printf(" 出队的元素顺序依次为：");
    for(i=0;i<n;i++)
    {
        flag=DeQueue(&Q,&x);
        printf("%5d",x);
    }
    printf("\n");
    if(flag=GetFront(&Q,&x))
        printf(" 当前的队头元素值为：%d",x);
    printf("\n");
    ShowQueue(&Q);
}
```

程序运行结果如图 3-14 所示。

```
请输入要入队的元素个数：5
请输入 5 个入队的整数：1 2 3 4 5
请输入要出队的元素个数：2
出队的元素顺序依次为：      1     2
当前的队头元素值为：3
从队头起队列中的各元素为：      3     4     5
```

图 3-14　出队列运行结果图

（5）取队头元素。将队头元素赋给指针 x 所指的对象，操作结果只是读取队头元素，队列 *Q* 不发生变化。其算法描述如下：

```
int GetFront(SeqQueue  *Q,DataType *x)
{  /* 取队头函数 */
    if(EmptyQueue (Q))
     /* 调用判空函数 EmptyQueue(Q)，判断队列是否为空 */
```

```
    {  printf(" 队空，无队头元素 !");
       return 0;
    }
    else                          /* 队不为空 */
    {  *x=Q->data[(Q->front+1)%MAXSIZE];
       return 1;
    }
}
```

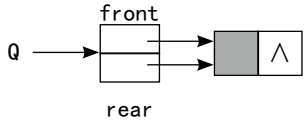
（6）显示队列元素。当队列不为空时，将队列中所有元素按从队头开始依次输出。操作结果只是读取队列中各元素，队列 *Q* 不发生变化。其算法描述如下：

```
void ShowQueue(SeqQueue  *Q)
{  /* 显示队中元素函数 */
   int p=Q->front;
   if(p==Q->rear)
      printf(" 队列为空，无元素！ \n");
   else
   {
         printf(" 从队头起队列中的各元素为：");
         while(p!=Q->rear)
         {
            printf("%5d",Q->data[p+1]);
            p++;
         }
   }
}
```

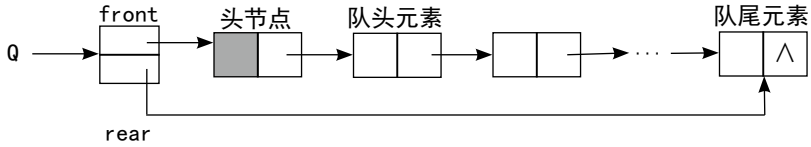

3.2.4 链队列

1. 链队列的定义

队列的链式存储结构称作链队列（Linked Queue）。它是限于在表头插入和表尾删除的单链表，显然需要设置指向队头的指针 front 和指向队尾的指针 rear。为了使空链队列与非空链队列的处理一致，链队列也加上头节点，如图 3-15 所示。



(a) 空链队列



(b) 非空链队列

图 3-15　链队列示意图

2. 链队列的类型定义

与单链表的类型定义相同，链队列的类型定义如下：

```
typedef  int  DataType;              /* 定义 DataType 为 int 类型 */
typedef  struct  qnode               /* 链队列存储类型 */
{  DataType    data;                 /* 定义节点的数据域 */
   struct  qnode  *next;             /* 定义节点的指针域 */
} LinkListQ;
typedef  struct
{
   LinkListQ *front,*rear;           /* 定义队列的队头指针和队尾指针 */
}LinkQueue;                          /* 链队列的头指针类型 */
```

3. 链队列的基本操作实现

（1）初始化队列操作。链队列的初始化即构造一个仅包含头节点的空链队列 *Q*。首先建立一个队列头节点（该节点中包含队头指针 front 和队尾指针 rear），然后再建立链队列中的头节点并将其指针域赋为空（NULL），然后将 Q->front 和 Q->rear 指针都指向该头节点，最后返回指针 *Q*。其算法描述如下：

```
void InitQueue(LinkQueue *Q)   /* 链队列初始化 */
{
  Q->front=(QNode*)malloc(sizeof(QNode));      /* 生成头节点 */
  if(Q->front==NULL)
    return 0;
  Q->rear=Q->front;
```

```
    Q->front->next=NULL;          /* 头节点的指针域置为空 */
}
```

（2）判断链队列空操作。判断是否是空队列（即 Q->front= =Q->rear），若链队列 Q 为空，则返回 1；否则返回 0。其算法描述如下：

```
int EmptyQueue(LinkQueue *Q)
{  /* 判断队空函数 */
   if(Q->front==Q->rear)          /* 链队为空 */
      return 1;
   else
      return 0;
}
```

（3）入队操作。先创建一个新节点 p，其 data 域值为 x，指针域为空，则入队操作步骤如下：

1）将新节点插入队尾，队尾指针域 Q->rear->next 指向新节点 p（执行语句：Q->rear->next=p）；

2）将队尾 Q->rear 指向新节点 p（执行语句：Q->rear=p）。

入队操作的过程如图 3-16 所示。

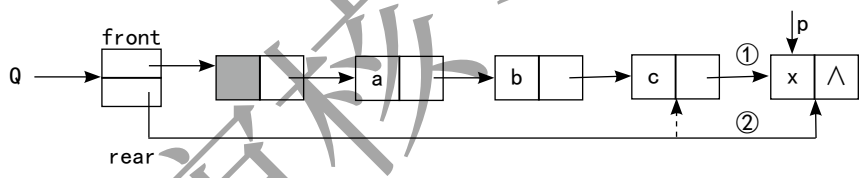


图 3-16　入队操作过程图

注意：由于链队列本身没有容量限制，因此，在进行入队操作时不用考虑队满情况。具体程序代码如下：

```
#include "stdio.h"
#include "malloc.h"
typedef  int  DataType;          /* 定义 DataType 为 int 类型 */
typedef  struct  qnode           /* 链队列存储类型 */
{ DataType    data;              /* 定义节点的数据域 */
  struct  qnode   *next;         /* 定义节点的指针域 */
} LinkListQ;
typedef  struct
{
  LinkListQ *front,*rear;                /* 定义队列的队头指针和队尾
```

```
指针 */
    }LinkQueue;                          /* 链队列的头指针类型 */

    LinkQueue  *InitQueue()
    {  /* 初始化链队列函数 */
       LinkQueue  *Q;
       LinkListQ  *p;
       Q=(LinkQueue *)malloc(sizeof(LinkQueue)); /* 建立链队列头指针所指
节点 */
       p=(LinkListQ *)malloc(sizeof(LinkListQ));    /* 建立链队列的头节点 */
       Q->front=p;                       /*Q 指针所指的 front 指针指向 p*/
       Q->rear=p;                        /*Q 指针所指的 rear 指针指向 p*/
       return Q;
    }

    int EmptyQueue(LinkQueue *Q)
    {  /* 判断队空函数 */
       if(Q->front==Q->rear)             /* 链队为空 */
          return 1;
       else
          return 0;
    }

    void InQueue(LinkQueue *Q,DataType x)
    {  /* 入队函数 */
       LinkListQ *p;
       p=(LinkListQ *)malloc(sizeof(LinkListQ));    /* 生成新节点 */
       p->data=x;                        /* 将 x 存入新节点的数据域 */
       p->next=NULL;
       Q->rear->next=p;                  /* 将新节点插入链队之后 */
       Q->rear=p;                        /* 队尾指针指向队尾元素 */
    }

    void ShowQueue(LinkQueue *Q)
    {  /* 显示队中元素函数 */
        LinkListQ *p=Q->front->next;
        if(p==NULL)
          printf(" 队列为空，无元素！ ");
```

```
    else
    {
            printf(" 从队列元素起栈中各元素为：");
      while(p!=NULL)
      {
        printf("%5d",p->data);
        p=p->next;
      }
    }
}

main()
{
  int i,n;
  LinkQueue *Q;
  DataType x;
  Q=InitQueue();
  printf(" 请输入要入队的元素个数：");
  scanf("%d",&n);
  printf(" 请输入 %d 个整数进行入队：",n);
  for(i=0;i<n;i++)
  {
     scanf("%d",&x);
     InQueue(Q,x);
  }
  ShowQueue(Q);
}
```


程序运行结果如图 3-17 所示。

```
请输入要入队的元素个数：5
请输入 5 个整数进行入队：1 2 3 4 5
从队列元素起栈中个元素为：    1    2    3    4    5
```

图 3-17　入队列运行结果图

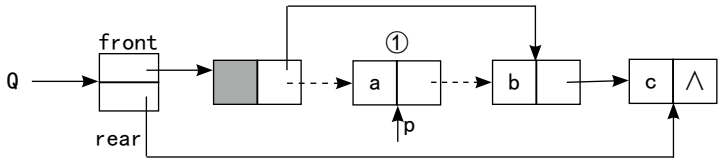
（4）出队操作。队头指针是指向队头元素的前一个节点，先将节点 p 指向队头元素，并将队头元素值取出赋给指针 x 所指的变量，则删除操作步骤如下：

1）将队头指针的指针域指向原队头元素的下一元素（新队头）的地址如图 3-18(a) 所示，（执行语句：Q->front->next=p->next）；

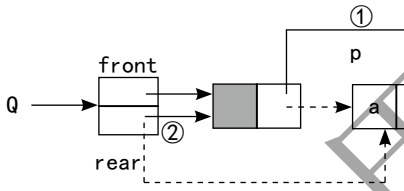
2）当原队列中仅有一个元素时如图 3-18(b) 所示，出队后队尾指针指向对头指针（执行语句：Q->rear=Q->front）；

3）释放 p 节点空间（执行语句：free(p)）。

出队的过程如图 3-18 所示。



(a) 一般链队列出列



(b) 仅含有一个元素的链队列出列

图 3-18　链队列出队操作过程图

```
#include "stdio.h"
#include "malloc.h"
typedef  int  DataType;              /* 定义 DataType 为 int 类型 */
typedef  struct  qnode               /* 链队列存储类型 */
{  DataType    data;                 /* 定义节点的数据域 */
   struct  qnode   *next;            /* 定义节点的指针域 */
} LinkListQ;
typedef  struct
{
   LinkListQ *front,*rear;           /* 定义队列的队头指针和队尾指针 */
}LinkQueue;                          /* 链队列的头指针类型 */

LinkQueue  *InitQueue()
{  /* 初始化链队列函数 */
   LinkQueue  *Q;
   LinkListQ  *p;
   Q=(LinkQueue *)malloc(sizeof(LinkQueue)); /* 建立链队列头指针所指节点 */
   p=(LinkListQ *)malloc(sizeof(LinkListQ));    /* 建立链队列的头节点 */
```

```
    Q->front=p;                    /*Q 指针所指的 front 指针指向 p*/
    Q->rear=p;                     /*Q 指针所指的 rear 指针指向 p*/
    return Q;
}

int EmptyQueue(LinkQueue *Q)
{  /* 判断队空函数 */
   if(Q->front==Q->rear)           /* 链队为空 */
      return 1;
   else
      return 0;
}

void InQueue(LinkQueue *Q,DataType x)
{  /* 入队函数 */
   LinkListQ *p;
   p=(LinkListQ *)malloc(sizeof(LinkListQ));    /* 生成新节点 */
   p->data=x;                      /* 将 x 存入新节点的数据域 */
   p->next=NULL;
   Q->rear->next=p;                /* 将新节点插入链队之后 */
   Q->rear=p;                      /* 队尾指针指向队尾元素 */
}

int DeQueue(LinkQueue *Q,DataType *x)
{  /* 出队函数 */
   LinkListQ *p;
   if(EmptyQueue(Q))               /* 调用判空函数 EmptyQueue(Q)，判断队
列是否为空 */
   {  printf(" 队空，不能出队元素 !");
    return 0;
   }
   else                            /* 队不为空 */
   {
      p=Q->front->next;            /*p 指向队头元素 */
      *x=p->data;                  /* 队头元素取出赋给 x*/
      Q->front->next=p->next;      /* 队头指针的指针域存放新队头元素的
地址 */
      if(p->next==NULL)            /* 队列中只含有一个元素出队 */
```

```
            Q->rear=Q->front;          /* 出队后队尾指针指向队头指针，此时
队空 */
            free(p);                   /* 释放原队头节点空间 */
            return 1;
        }
    }

    int GetFront(LinkQueue *Q,DataType *x)
    {  /* 获取队头元素函数 */
        if(EmptyQueue(Q))              /* 调用判空函数 EmptyQueue(Q)，判断队
列是否为空 */
        {
            printf(" 队空，无队头元素 !");
            return 0;
        }
        else                           /* 队不为空 */
        {
            *x=Q->front->next->data;   /* 队头元素赋给变量 x*/
            return 1;
        }
    }

    void ShowQueue(LinkQueue *Q)
    {  /* 显示队中元素函数 */
        LinkListQ *p=Q->front->next;
        if(p==NULL)
          printf(" 队列为空，无元素！ ");
        else
        {
                printf(" 从队列元素起栈中各元素为：");
            while(p!=NULL)
            {
              printf("%5d",p->data);
              p=p->next;
            }
        }
    }
```

```
main()
{
  int i,n,flag;
  LinkQueue *Q;
  DataType x;
  Q=InitQueue();
  printf(" 请输入要入队的元素个数：");
  scanf("%d",&n);
  printf(" 请输入 %d 个整数进行入队：",n);
  for(i=0;i<n;i++)
  {
    scanf("%d",&x);
    InQueue(Q,x);
  }
  printf(" 请输入要出队的元素个数：");
  scanf("%d",&n);
  printf(" 出队的元素顺序依次为：");
  for(i=0;i<n;i++)
  {
    flag=DeQueue(Q,&x);
    printf("%5d",x);
  }
  printf("\n");
  if(flag=GetFront(Q,&x))
  printf(" 当前的队头元素值为：%d",x);
  printf("\n");
  ShowQueue(Q);
}
```

程序运行结果如图 3-19 所示。

```
请输入要入队的元素个数：5
请输入 5 个整数进行入队：1 2 3 4 5
请输入要出队的元素个数：2
出队的元素顺序依次为：    1    2
当前的队头元素值为：3
从队列元素起栈中各元素为：    3    4    5
```

图 3-19　出队列运行结果图

3.3　串

3.3.1　串的定义

串（String）是字符串的简称。它是一种在数据元素的组成上具有一定约束条件的线性表，即要求组成线性表的所有数据元素都是字符。所以，也可以这样定义串：串是由零个或多个字符组成的有限序列。串一般记作：s="$a_1a_2...a_n$"（$n \geqslant 0$），其中，s 是串的名称，用双引号（""）括起来的字符序列是串的值；双引号本身不是串的值，它们是串的标记，以便将串与标识符（如变量名）加以区别，称为定界符；a_i（$1 \leqslant i \leqslant n$）可以是字母、数字或其他字符；串中字符的数目 n 被称作串的长度（或串长）。当 n=0 时，串中没有任何字符，其串的长度为 0，称为空串，用 Φ 表示。而空格串是有一个或者多个空格组成的串，串的长度为所含空格的个数。例如：

$s1$ =""

$s2$ ="　　"

$s1$ 中没有字符，是一个空串；而 $s2$ 中有两个空格字符，它的长度等于 2，它是一个空格串。

串中任意连续的字符组成的子序列被称为该串的子串。包含子串的串相应地又被称为该子串的主串。通常称字符在串中的序号为该字符在串中的位置。当一个字符在串中多次出现时，以该字符第一次在串中出现的位置为该字符在串中的位置。子串在主串中的位置则以子串在主串中第一次出现的第一个字符的位置来表示。两个串相等是指当且仅当两个串的长度相等且各个对应位置上的字符都相同时，两个串相等。

例如，有以下 4 个串：

S1="Welcome to Beijing！ "

S2="Welcome to"

S3="Welcometo "

S4="Beijing"

则各串长度及其之间的关系是如何呢？

（1）S1 的长度为 19，S2 的长度为 10，S3 的长度为 10，S4 的长度为 7。

（2）S2 和 S4 为 S1 的子串，S1 相对于 S2 和 S_4 为其主串，S2 在 S1 的位置为 1，S4 在 S1 的位置为 12。

（3）S3 不是 S1 的子串，因为它不是 S1 串中的连续字符组成的子序列（在 e 与 t 字母之间缺少一个空格）。

（4）S2 与 S3 串不相等，因为虽然两串的长度相等，但各个对应的字符不相同。

3.3.2 串的应用

在汇编语言和高级语言的编译程序中，源程序和目标程序都是以字符串表示的。在事务处理程序中，如客户的姓名、地址、邮政编码以及货物名称等，一般也是作为字符串数据处理的。另外，信息检索系统、文字编辑系统、语言翻译系统等，也都是以字符串数据作为处理对象的。

对照串的定义和线性表的定义可知，串是一种其数据元素固定为字符的线性表。但是，串的基本操作对象和线性表的操作对象却有很大的不同。线性表上的操作是针对其某个元素进行的，而串上的操作主要是针对串的整体或串的一部分子串进行的。这也是数据结构中把串单独作为一章节的原因。

3.3.3 串的基本操作

串的基本操作和线性表有很大差别。在线性表的基本操作中，大多以“单个元素”作为操作对象，如：在线性表中查找某个元素、求取某个元素、在某个位置上插入一个元素和删除一个元素等；而在串的基本操作中，通常以“串的整体”作为操作对象，如：在串中查找某个子串、求取一个子串、在串的某个位置上插入一个子串以及删除一个子串等。

串的基本操作有赋值、连接、求串长、求子串在主串中出现的位置、判断两个串是否相等以及删除子串等。

串的基本操作有很多，现在介绍部分基本运算。

（1）串的复制 StrCopy(S，T)，由串 T 复制得串 S。

（2）求串长度 StrLength(S)，返回串 S 的长度，即串 S 中的元素个数。

（3）串连接 Connect($s1$，$s2$)，将串 $s2$ 连接在 $s1$ 的尾部，形成一个新串。

（4）两串相等判断 Equal($s1$，$s2$) 判断两个串是否相等，若相等返回 1，否则返回 0。

（5）求子串 SubString(Sub，S，pos，len)，用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。

（6）串的定位（也称模式匹配）StrIndex(S，T)，若串 S 中存在与串 T 相同的子串，则返回它在串 S 中第一次出现的位置；否则返回 -1 或代表错误的值。

（7）串的插入 StrInsert(S，pos，T)，在串 S 的第 pos 个字符插入串 T。

（8）串的删除 StrDelete(S，pos，len)，从串 S 中删除第 pos 个字符起长度为 len 的子串。

（9）串的替换 StrReplace(S，T，V)，用 V 替换串 S 中出现的所有与 T 相等的不重叠子串。

（10）子串定位 Match(s，$s1$)，返回子串 s_1 在串 s 中第一次出现的位置。

（11）判断串空 StrEmpty(S)，若串 S 为空串，则返回 1；否则返回 0。

（12）串的比较 StrCompare(S，T)，若 $S>T$，则返回值 >0；若 $S=T$，则返回值等于 0；若 $S<T$，则返回值 <0。

3.3.4　串的存储结构

由于串是一种特殊的线性表，故串的存储结构和线性表的存储结构类似。但由于串是由若干单个字符组成，所以存储时有一些特殊技巧。

串的存储方式取决于即将对串所进行的操作。串在计算机中有三种表示方法：

（1）定长顺序存储结构。这种方法是将串定义成字符数组，是最简单的处理方法。此时，数组名即为串名，从而实现了从串名直接访问串值。用这种方法处理，串的存储空间是在编译阶段完成的，其大小不能更改。

（2）堆分配存储结构。这种表示方法的特点是仍用一组地址连续的存储单元依次存储串中字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。

（3）串的链式结构存储。在链式结构存储中，每个节点设定一个字符域 char 存放字符，设定一个指针域 next 存放所指向的下一个节点的地址。

1. 串的定长顺序存储结构

顺序串的类型定义与顺序表的定义相似，可以用一个字符型数组和一个整型变量表示，其中字符数组存储串，整型变量表示串的长度。

如串 S="Beijing"，字符串从 S.ch[0] 单元开始存放，用 '\0' 来表示串的结束，则串 S 的存储示意图如图 3-20 所示。

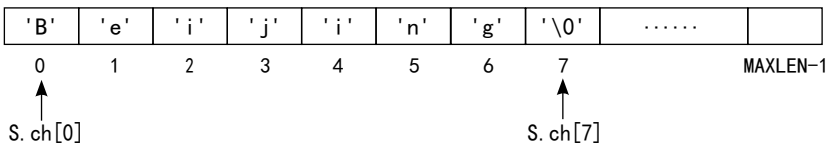


图 3-20　串 S 的顺序存放示意图

当计算机按字节（Byte）为单位编址时，一个存储单元刚好存放一个字符，串中相邻的字符顺序地存储在地址相邻的存储单元中。

当计算机按字（例如 1 个字为 32 位）为单位编址时，一个存储单元可以由 4 个字节组成。此时顺序存储结构又有非紧凑格式和紧凑格式两种存储方式。

（1）非紧凑存储。设串 S="Hello World"，计算机字长为 32 位（4 个 Byte），用非紧凑格式一个地址只能存一个字符，如图 3-21 所示。其优点是运算处理简单，但缺点是存储空间十分浪费。

（2）紧凑存储。同样存储 S="Hello World"，用紧凑格式一个地址能存四个字符，如图 3-22 所示。紧凑存储的优点是空间利用率高，缺点是对串中字符处理的效率低。

H			
e			
l			
l			
o			
W			
o			
r			
l			
d			

图 3-21　非紧凑格式

H	e	l	l
o		W	o
r	l	d	

图 3-22　紧凑格式

2. 串的堆分配存储结构

（1）堆分配存储的方法。堆分配存储结构的实现方法是，提供一个足够大的连续存储空间，作为串的可利用空间，用它来存储各串的串值。每当建立一个新串时，系统就从这个可利用空间中划分出一个大小和新串长度相等的空间给新串，若分配成功则返回一个指向起始地址的指针。为操作方便，将每个串的长度信息也作为存储结构的一部分。可使用 C 语言中动态分配函数库中的 malloc() 和 free() 函数来管理可利用空间。虽然这种存储表示仍以一组地址连续的存储单元存放串值，但它属于一种动态分配方式，所以也可看作一种动态存储分配的顺序表。

串的堆分配存储结构描述如下。

```
typedef  struct
{
   int  len;       /*len 存放串长 */
   char *ch;       /*ch 存放串的首地址，若是空串，则 ch 的值为 NULL*/
}HString;
```

（2）索引存储的例子：

设字符串：S1="boy"

S2="girl"

S3="man"

S4="woman"

用指针 free 指向堆中未使用空间的首地址。

索引表如图 3-23 所示。

考虑到对字符串的插入和删除操作，可能引起串的长度变化，在“堆”中为串值分配空间时，可预留适当的空间。这时，索引表的索引项应增加一个域，用于存储该串在“堆”中拥有的实际存储单元的数量。当字符串长度等于该串的实际存储单元时，就不能对串进行插入操作。

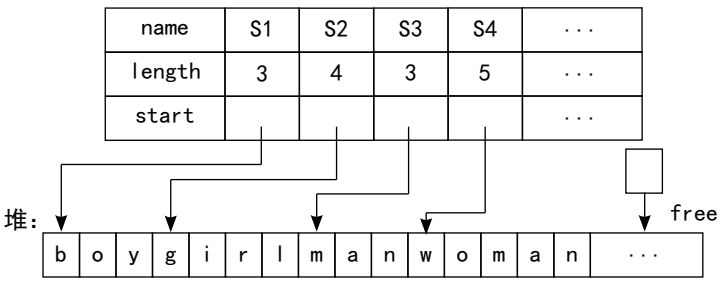


图 3-23　带长度的索引表

（3）“堆”的管理。C 语言中用动态分配函数 malloc 和 free 来管理“堆”。利用函数 malloc 为每个新串分配一块实际串长所需要的存储空间，分配成功则返回一个指向起始地址的指针，作为串的基址，同时，约定的串长也作为存储结构的一部分。函数 free 则用来释放用 malloc 分配的存储空间。

3. 串的链式存储结构

（1）链接存储的描述。用链表存储字符串，每个节点有两个域：一个数据域（data）和一个指针域（next）。在串的链式存储结构中，有如下结构：

数据域（data）—— 存放串中的字符；

指针域（next）—— 存放后继节点的地址。

以存储 S="Hello boy" 为例，链接存储结构如图 3-24 所示。

1）链接存储的优点 —— 插入、删除运算方便；

2）链接存储的缺点 —— 存储、检索效率较低。

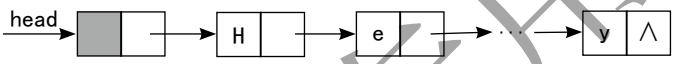


图 3-24　带头节点的非空单链表

（2）串的存储密度。在各种串的处理系统中，所处理的串往往很长或很多。例如一本书的数百万个字符，情报资料的数千万个条目，这就必须考虑字符串的存储密度，有

$$\text{存储密度} = \text{串值所占的存储位} / \text{实际分配的存储位} \qquad (3\text{-}1)$$

为了提高存储密度，节点的大小一般大于 1，所以称为块链结构。例如，采用块链结构的文本编辑系统中，一个节点可存放 80 个字符，此时节点的大小是 80。

当节点的大小大于 1 时，存放一个串需要的节点数目不一定是整数，而分配节点时总是以完整的节点为单位进行的。因此，为使一个串能存放在整数个节点里，应在串的末尾填上不属于串值的特殊字符，以表示串的终结。由于字符串的特殊性，用链表作为字符串的存储方式也不太实用，因此使用较少。

4. 顺序串的基本运算实现

在串的定长顺序存储结构中，串的类型定义描述如下：

```
    #define MAXLEN <最大串长>;          /*定义能处理的最大的串长度*/
    typedef struct
    {
```

```
    char  str[MAXLEN];      / * 定义可容纳 MAXLEN 个字符的字符数组 * /
    int  Len;               / * 定义当前实际串长度 * /
} String;
```

（1）求串的长度操作。求得一个字符串 S 的长度值并返回该值。程序代码如下：

```
#include "stdio.h"
#define  MAXLEN  100    /* 顺序串存储空间的总分配量 */
typedef  struct         /* 串结构定义 */
{
   char ch[MAXLEN];
   int Len;
}String;

int StrLength(String *S)
{  /* 求串长度函数 */
   int i=0;
   while(S->ch[i]!='\0')
     i++;
   S->Len=i;
   return (S->Len);
}

void CreatStr(String *S)
{   /* 建立一个新串 */
    gets(S->ch);
    S->Len=StrLength(S);
}

main()
{
   String  x,y,z;
   String *S=&x,*S1=&y,*S2=&z;
   printf(" 请输入一个字符串：");
   CreatStr(S);
   printf(" 该串值为：");
   if(S->ch[0]=='\0')
      printf(" 空串 ");
```

```
    else
      {  puts(S->ch);
              printf(" 该串的长度为：%d\n",S->Len);
      }
}
```

程序运行结果如图 3-25 所示。

```
请输入一个字符串：Welcome to BeiJing
该串值为：Welcome to BeiJing
该串的长度为：18
```

图 3-25　创建串运行结果图

（2）求子串操作。求串 S 从第 pos 位置开始，长度为 len 的子串，并将其存入到串 Sub 中。操作成功返回 1；不成功返回 0。程序代码如下：

```
#include "stdio.h"
#define  MAXLEN  100              /* 顺序串存储空间的总分配量 */
typedef  struct                   /* 串结构定义 */
{
   char ch[MAXLEN];
   int Len;
}String;

int StrLength(String *S)
{  /* 求串长度函数 */
  int i=0;
  while(S->ch[i]!='\0')
    i++;
  S->Len=i;
  return (S->Len);
}

void CreatStr(String *S)
{  /* 建立一个新串 */
    gets(S->ch);
    S->Len=StrLength(S);
}
```

```
int SubString(String *S,String *Sub,int pos,int len)
{  /* 求子串函数 */
   int j;
   if(pos<1 || pos>S->Len || len<1 || len>S->Len-pos+1)
   {
     Sub->Len=0;
     printf(" 参数错误 !\n");
     return 0;
   }
   else
   {
   for(j=0;j<len;j++)
      Sub->ch[j]=S->ch[pos+j-1];
   Sub->ch[j]='\0';
   Sub->Len=len;
   return 1;
   }
}

main()
{
  int i,len,flag;
  String  x,y,z;
  String *S=&x,*S1=&y,*S2=&z;
  printf(" 请输入一个字符串：");
  CreatStr(S);
  printf(" 该串值为：");
  if(S->ch[0]=='\0')
     printf(" 空串 ");
  else
  {  puts(S->ch);
     printf(" 该串的长度为：%d",S->Len);
  }
  printf("\n");
  printf(" 请输入从第几个字符开始求子串：");
  scanf("%d",&i);
  printf(" 请输入取出的子串长度：");
  scanf("%d",&len);
```

```
    flag=SubString(S,S1,i,len);
    if(flag)
    {  printf(" 求子串成功，原来主串为：");
       puts(S->ch);
       printf(" 所得子串为：");
       puts(S1->ch);
    }
    else
       printf(" 求子串失败！ ");
}
```

程序运行结果如图 3-26 所示。

```
请输入一个字符串：Welcome to BeiJing
该串值为：Welcome to BeiJing
该串的长度为：18
请输入从第几个字符开始求子串：4
请输入取出的字符长度：4
求子串成功，原生主串为：Welcome to BeiJing
所得子串为：come
```

图 3-26 求子串运行结果图

（3）删除子串的操作。将删除串 S 从指定位置 *i* 开始的连续 *l* 个字符。算法思路如下：首先判断删除位置和删除串长度是否出错，出错给出错误信息后返回 0。不出错则继续下面操作：将第 *i*+*l*-1 位的字符向前移到第 *i* 位上，以此类推，直到最后一个字符移动完毕。最后修改串 S 的长度，将新串 S 的尾部加上字符串结束标志‘\0’（不加该语句多次运行程序会出错），操作成功，返回 1。程序代码如下：

```
#include "stdio.h"
#define  MAXLEN  100              /* 顺序串存储空间的总分配量 */
typedef  struct                   /* 串结构定义 */
{
   char ch[MAXLEN];
   int Len;
}String;

int StrLength(String *S)
{ /* 求串长度函数 */
```

```
  int i=0;
  while(S->ch[i]!='\0')
    i++;
  S->Len=i;
  return (S->Len);
}

void CreatStr(String *S)
{  /* 建立一个新串 */
    gets(S->ch);
    S->Len=StrLength(S);
}


int  StrDelete(String *S,int i,int j)
{  /* 在串 s 中删除从指定位置 i 开始连续的 j 个字符 */
    int k;
    if(i+j-1>S->Len)
    {  printf(" 所要删除的子串超界！ ");
            return 0;
    }
    else
    {
            for(k=i+j-1;k<S->Len;k++,i++)
              S->ch[i-1]=S->ch[k];
            S->Len=S->Len-j;
            S->ch[S->Len]='\0';
            return 1;
    }
}

main()
{
  int i,len,flag;
  String  x,y,z;
  String *S=&x,*S1=&y,*S2=&z;
  printf(" 请输入一个字符串： ");
  CreatStr(S);
```

```
    printf(" 该串值为：");
    if(S->ch[0]=='\0')
      printf(" 空串 ");
    else
    {  puts(S->ch);
      printf(" 该串的长度为：%d",S->Len);
    }
    printf("\n");
    printf(" 请输入要删除的子串的起始位置：");
    scanf("%d",&i);
    printf(" 请输入要删除的子串的长度：");
    scanf("%d",&len);
    printf(" 原来串为：");
    puts(S->ch);
    if(flag=StrDelete(S,i,len))
    {  printf(" 删除子串成功，删除后的新的串为：");
      puts(S->ch);
    }
    else
      printf(" 删除子串失败！");
  }
```

程序运行结果如图 3-27 所示。

```
请输入一个字符串：Welcome to BeiJing
该串值为：Welcome to BeiJing
该串的长度为：18
请输入要删除的子串的起始位置：9
请输入要删除的子串的长度：2
原来串为：Welcome to BeiJing
删除子串成功，删除后的新的串为：Welcome  BeiJing
```

图 3-27　删除串运行结果图

（4）插入子串操作。在串 S 中的第 *i* 位置开始插入子串 S1。算法思路如下。首先判断插入子串的位置是否出错，然后再判断两串长度是否超过存储空间长度，都不出错则进行插入。插入方法如下：首先从最后一个字符开始向后移动子串 S1 长度，直到第 *i* 个字符结束。移动完毕将子串 S1 的每个字符复制到串 S 的第 *i* 位开始的各位置上，完成插入子串过程。最后修改串 S 的长度，将新串 S 的

尾部加上字符串结束标志‘\0’（不加该语句多次运行程序会出错），操作成功，返回 1。程序代码如下：

```
#include "stdio.h"
#define  MAXLEN  100              /* 顺序串存储空间的总分配量 */
typedef  struct                   /* 串结构定义 */
{
   char ch[MAXLEN];
   int Len;
}String;

int StrLength(String *S)
{  /* 求串长度函数 */
  int i=0;
  while(S->ch[i]!='\0')
    i++;
  S->Len=i;
  return (S->Len);
}

void CreatStr(String *S)
{  /* 建立一个新串 */
    gets(S->ch);
    S->Len=StrLength(S);
}

int StrInsert(String *S,String *S1,int i)
{  /* 在串 s 中插入子串 s1 函数 */
    int k;
    if(i>S->Len+1)
    {  printf(" 插入位置错误！ ");
            return 0;
    }
    else if(S->Len+S1->Len>MAXSIZE)
    {  printf(" 两串长度超过存储空间长度！ ");
            return 0;
    }
    else
```

```
    { for(k=S->Len-1;k>=i-1;k--)
            S->ch[S1->Len+k]=S->ch[k];
      for(k=0;k<S1->Len;k++)
        S->ch[i+k-1]=S1->ch[k];
      S->Len=S->Len+S1->Len;
      S->ch[S->Len]='\0';
      return 1;
    }
}

main()
{
  int i,flag;
  String x,y,z;
  String *S=&x,*S1=&y,*S2=&z;
  printf(" 请输入一个字符串：");
  CreatStr(S);
  printf(" 该串值为：");
  if(S->ch[0]=='\0')
    printf(" 空串 ");
  else
  { puts(S->ch);
    printf(" 该串的长度为：%d",S->Len);
  }
  printf("\n");
  printf(" 请输入要插入子串在主串中的位置：");
  scanf("%d",&i);
  printf(" 请输入一个子串：");getchar();
  CreatStr(S1);
  printf(" 原来主串为：");
  puts(S->ch);
  if(flag=StrInsert(S,S1,i))
  { printf(" 插入子串成功！插入后的新的主串为：");
    puts(S->ch);
  }
  else
    printf(" 插入子串失败！ ");
}
```

程序运行结果如图 3-28 所示。

```
请输入一个字符串：Welcome BeiJing
该串值为：Welcome BeiJing
该串的长度为：15
请输入要插入子串在主串中的位置：9
请输入一个子串：to
原来主串为：Welcome BeiJing
插入子串成功！插入后的新的串为：Welcome to BeiJing
```

图 3-28　插入串运行结果图

（5）判断两串相等的操作。判断串 S1 和串 S2 是否相等，如果相等，返回值为 0；若不相等，返回两个串对应第一个不相同位置字符 ASCII 码的差值。程序代码如下：

```
#include<stdio.h>
#define MAXLEN 100
typedef struct
{
   char ch[MAXLEN];
   int len;
}String;

int Equal(String s1,String s2)
{
   int i;
   if(s1.len!=s2.len)
      return 0;
   else
   {
      for(i=0;i<s1.len;i++)          /* 判断 s1 和 s2 中各对应位置上字符是否相等 */
         if(s1.ch[i]!=s2.ch[i])              /* 如果某一对应位置上字符不相等 */
            return 0;                 /* 返回函数值 0*/
      }                               /* 两个串完全相等时 */
      return 0;                       /* 返回函数值 1*/
   }
```

```
main()
{
   String a1={"Welcome to ",11},a2={"BeiJing",7};
   int r;
   r=Equal(a1,a2);                    /* 调用 equal 函数 */
   printf("\n%d",r);
}
```

（6）两串的连接操作。将第二个串 *T* 连接到第一个串 *S* 的后面，形成一个新的串存在 *S* 中；操作成功返回为 1，失败返回 0。

算法思路：如果连接后串长小于串 *S* 的存储长度 MAXSIZE，则将串 *T* 直接连接到串 *S* 的尾部，形成新串，修改串 *S* 的长度并设新串的字符串结束标志，操作成功返回 1；如果连接后串长大于 MAXSIZE，但串 *S* 长度小于 MAXSIZE，连接后串 *T* 部分字符序列被舍弃，将串 *T* 连接到串 *S* 的尾部，直到串 *S* 的存储空间满为止（留一个字符串结束标志位置），操作失败返回 0；否则说明串 *S* 的长度等于 MAXSIZE，串 *T* 不被连接，操作失败返回 0。程序代码如下：

```
#include<stdio.h>
#define MAXLEN 100
typedef struct
{
   char ch[MAXLEN];     /
   int len;
}String;

String Connect(String s1, String s2)
{
   String s; int i;
   if(s1.len+s2.len<=MAXLEN)   /* 当 s1 和 s2 的长度之和小于或等于
MAXLEN 时 */
   {
      for(i=0;i<s1.len;i++)          /* 将 s1 存放到 s 中 */
         s.ch[i]=s1.ch[i];
      for(i=0;i<s2.len;i++)          /* 将 s2 存放到 s 中 */
         s.ch[s1.len+i]=s2.ch[i];
      s.ch[s1.len+i]='\0';           /* 设置串结尾标志 */
      s.len=s1.len+s2.len;           /*s 的长度为 s1 和 s2 的长度之和 */
```

```
    }
    else                          /* 当 s1 和 s2 的长度之和大于 MAXLEN 时 */
      s.len=0;                    /* 不能连接，置 s 串长度为 0*/
    return(s);                    /* 连接成功，返回 s，不成功时返回空串 */
}

main()                            /* 主程序 */
{
  String a1={"Welcome to ",11},a2={"BeiJing",7},s;    /
  s=Connect(a1,a2);
  printf("\n%s\n%d",s.ch,s.len);
}
```

（7）串的定位操作。若串 S 中存在与串 T 相同的子串，则返回串 T 在串 S 中第一次出现的位置，否则返回 -1。算法思路如下：首先将 i、j 分别指向串 S 和串 T，当 i、j 没有指向两串尾时进行循环时，对应位置字符相同则继续比较下一个字符，如果不相同则令 i 等于 $i-j+1$ 且 j 等于 0 进行下一次的字符串首字符比较。循环完毕后若 j 大于等于串 T 的长度，说明串 S 中有子串 T，返回其首次出现的起始位置，否则说明串 S 中没有子串 T，返回 -1。程序代码如下：

```
#define MAXLEN 100
typedef struct
{
  char ch[MAXLEN];
  int len;
}String;

int Match(String s, String s1)
{
  int i,j,k;
  i=0;
  while(i<=s.len-s1.len)  /*i 为 s 串中字符的位置，每次前进一个位置，*/
  {                               /* 该循环执行到 s 串中剩余长度不够比较时为止 */
    j=i;                  /*j 用作临时计数变量 */
    k=0;                  /* 用 k 控制比较的长度小于 s1.len*/
    while((k<s1.len)&&(s.ch[j]==s1.ch[k]))        /* 比较过程 */
    {
```

```
            j=j+1;
            k=k+1;
        }
        if(k==s1.len)          /* 比较成功，返回 i 的位置 */
            return i;
        else                   /* 比较不成功，从 s 串中下一个字符继续比较 */
            i=i+1;
    }
    return -1;                 /* 比较结束时，未找到匹配字符串，返回标识 -1*/
}

main()
{
    String a={"Beijing Shanghai China",22};
    String a1={"Shanghai",8};
    int r;
    r=Match(a,a1);
    printf("\n%d",r);
}
```

（8）子串替换算法。串为顺序存储结构，要求实现用子串 V 将主串 S 中的所有串 T 均替换。子串替换算法主要思路采用先定位，然后调用删除子串函数删除该串 T，接着再调用插入子串函数将子串 V 插入到该位置中，然后再查找下一个位置反复执行删除串 T 和插入串 V 的操作，直到串 S 尾部。其算法描述如下：

```
void StrReplace(String *S,String *T,String *V)
{  /* 子串替换函数 */
    int i,m,n,p,q;
    n=S->Len;
    m=T->Len;
    q=V->Len;
    p=1;
do
    {
      i=StrIndex(S,T);              /* 调用定位函数得到子串 T 在主串 S 的
位置 */
      if(i!=-1)                     /* 当主串 S 中有该子串 T 时 */
      {
```

```
                DelStr(S,i,m);          /* 调用删除子串函数删除该子串 T*/
                InStr(S,V,i);           /* 调用插入子串函数插入子串 V*/
                p=i+q;
                S->Len=S->Len+q-m;      /* 修改主串 S 的长度 */
                n=S->Len;
            }
        }while((p<=n-m+1)&&(i!=-1));
    }
```

3.4　项 目 训 练

项目一：八皇后问题

【题目要求】

八皇后问题是在 8×8 的国际象棋棋盘上，安放 8 个皇后，要求没有一个皇后能够吃掉任何其他一个，即没有两个或两个以上的皇后占据棋盘上的同一行、同一列或同一对角线。

【算法分析】

算法基本思想如下：

从第 1 行起逐行放置皇后，每放置一个皇后均需要依次对第 1，2，…，8 列进行试探，并尽可能取小的列数。若当前试探的列位置是安全的，即不与已放置的其他皇后冲突，则将该行的列位置保存在栈中，然后继续在下一行上寻找安全位置；若当前试探的列位置不安全，则用下一列进行试探，当 8 列位置试探完毕都未找到安全位置时，就退栈回溯到上一行，修改栈顶保存的皇后位置，然后继续试探。

算法抽象描述如下：

（1）置当前行当前列均为 1。

（2）while（当前行号〈=8〉）。

（3）{ 检查当前行，从当前列起逐列试探，寻找安全列号。

（4）if（找到安全列号）。

（5）放置皇后，将列号记入栈中，并将下一行置成当前行，第一列置为当前列。

（6）else。

（7）退栈回溯到上一行，移去该行已放置的皇后，以该皇后所在列的下一列作为当前列。

（8）} 结束程序。

项目二：串模式匹配算法

【题目要求】

在串的操作中，在主串中查找模式串的模式匹配算法 —— 即求字串的位置的函数 Index(s，t)，是文本处理中最常用、最重要的操作之一。

所谓子串的定位就是求子串在主串中首次出现的位置，又称为模式匹配或串匹配。模式匹配的算法很多，在这里要求用最简单的朴素模式匹配算法。该算法的基本思想是将给定子串从第一个字符开始比较，找到首次与子串完全匹配的子串为止，并记住该位置。但为了实现统计子串出现的个数，不仅需要从主串的第一个字符位置开始比较，而且需要从主串的任一给定位置加检索匹配字符串，所以首先要给出两个算法：一个标准的朴素模式匹配算法；一个给定那个位置的匹配算法。

【算法分析】

设有 3 个指针：i，j，k。i 用于指示主串 S 每次开始比较的位置；指针 j 和 k 分别指示主串 S 和模式串 T 中当前正在等待比较的字符位置；一开始从主串 S 的第一个字符 (i=0，j=0) 和模式 T 的第一个字符 (k=0) 比较，若相等，则继续逐个比较后续字符 (j++，k++)，否则从主串的下一个字符 (i++) 起再重新和模式串 (j=0) 的字符开始比较。依次类推，直到模式 T 中的所有字符都比较完，而且一直相等，则匹配成功，返回位置 i；否则返回 -1，表示匹配失败。

本章小结

（1）理解栈的定义和特点，栈在表达式计算机中的使用，栈的顺序存储表示和链式存储栈表示，以及栈在程序设计中的应用。特别要注意，链式栈的栈顶在链头，插入和删除都在链头进行。

（2）队列是一种运算受限制的线性表，一般队列只允许在队尾进行插入操作，在队头进行删除操作。

（3）队列的逻辑结构和线性表也相同，数据元素之间存在一对一的关系，其主要特点是“先进先出”。

（4）串是有限个字符组成的序列，一个串的字符个数叫作串的长度，长度

为零的字符串称为空串。

（5）串是一种特殊的线性表，规定每个数据元素仅由一个字符组成。

习　　题

一、选择题

1. 对于栈操作数据的原则是（　　）。

A. 先进先出　　B. 后进先出

C. 后进后出　　D. 不分顺序

2. 有 6 个元素按 6，5，4，3，2，1 的顺序进栈，问下列（　　）不是合法的出栈序列?

A. 5 4 3 6 1 2　　B. 4 5 3 1 2 6

C. 3 4 6 5 2 1　　D. 2 3 4 1 5 6

3. 设有编号为 1，2，3，4 的四辆列车，顺序进入一个栈结构的站台，下列不可能的出站顺序为（　　）。

A. 1234　　B. 1243

C. 1324　　D. 1423

4. 对于队列操作数据的原则是（　　）。

A. 先进先出　　B. 后进先出

C. 先进后出　　D. 不分顺序

5. 最大容量为 n 的循环队列，队尾指针是 rear，队头是 front，则队空的条件是（　　）。

A. (rear+1)% n==front　　B. rear==front

C. rear+1==front　　D. (rear-1)% n==front

6. 设链栈中节点的结构：data 为数据域，next 为指针域，且 top 是栈顶指针。若想在链栈的栈顶插入一个由指针 *s* 所指的节点，则应执行下列（　　）操作。

A. s->next=top->next；top->next=s；

B. top->next=s；

C. s->next=top；top=top->next；

D. s->next=top；top=s；

7. 以下论述正确的是（　　）。

A.　空串与空格串是相同的

B. "tel" 是 "Teleptone" 的子串

C. 空串是零个字符的串

D. 空串的长度等于 1

8. 若 SubString(Sub,S,pos,len) 表示用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串的操作，则对于 S="Data Structure"，SubString(Sub,S,6,3) 的结果为（　　）。

A. "ta Str"　　B. "Str"

C. " tru"　　D. 以上均不正确

二、填空题

1. 循环队列的引入，目的是为了克服_____现象。

2. 在队列中存取数据应遵循的原则是_____。

3._____是被限定为只能在表的一端进行插入运算，在表的另一端进行删除运算的线性表。

4. 在队列中，允许插入的一端称为_____。

5. 在队列中，允许删除的一端称为_____。

6. 队列在进行出队操作时，首先要判断队列是否为_____。

7. 设 S=“My mather”，则 LenStr(*s*)=_____。

8. 两个字符串分别为：S1="Today is"，S2="24 July, 2011", ConcatStr(S1, S2) 的结果是：_____。

三、综合题

1. 将一个非负十进制整数转换成八进制数的算法实现。

2. 一个用单链表组成的循环队列，只设一个尾指针 rear，不设头指针，请编写如下算法：

（1）向循环队列中插入一个元素为 *x* 的节点；

（2）从循环队列中删除一个节点。

3. 有两个串 S1 和 S2，设计一个算法，求一个串 *T*，使其中的字符是 S1 和 S2 中的公共字符。

第 4 章　广义线性表

前面几章讨论的线性结构中的数据元素都是非结构的原子类型，元素的值是不再分解的。本章讨论的两种数据结构——广义表和数组，可以看成是线性表的扩展，表中的数据元素本身也是一个数据结构。

知识目标

- 理解广义表的概念、逻辑结构特征及存储结构。
- 理解数组的逻辑结构特征。
- 掌握数组顺序存储结构地址的计算。
- 理解特殊举证的存储结构。
- 掌握稀疏矩阵的存储结构、基本运算及相关算法分析。

能力目标

- 能应用稀疏举证三元组表的理论设计算法，解决实际问题。

4.1 广 义 表

4.1.1 广义表的定义

定义：广义表一般记作：LS=(a_1，a_2，…，a_n)，其中，LS 是广义表 (a_1，a_2，…，a_n) 的名称，n 是它的长度。广义表中 a_i 可以是单个元素，也可以是广义表，分别称为广义表 LS 的原子和子表。习惯上，用大写字母表示广义表的名称，小写字母表示原子。当广义表非空时，称第一个元素 a_1 为 LS 的表头，称其余元素组成的表 (a_2，a_3，…，a_n) 是 LS 的表尾。a_{i+1} 称为 a_i 的直接后继。

广义表通常用圆括号括起来，并用逗号分隔表中的元素。为了清楚起见，通常用大写字母表示广义表，用小写字母表示单个数据元素。下面是广义表的例子。

（1）A=( ) —— A 是一个空表，它的长度为零。

（2）B=(e) —— 列表 B 只有一个原子 e，B 的长度为 1。

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

（4）D=(A，B，C) —— 列表 D 的长度为 3，三个元素都是列表。显然，将子表的值代入后，则有 D=(( )，(e)，(a，(b，c，d)))。

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

广义表的长度指该广义表中所包含的元素（包括原子和子表）的个数。广义表的深度指该广义表中所包含括号的层数。

4.1.2 广义表的性质

从上述广义表的定义和例子可以得到广义表的下述重要性质：

（1）广义表是一种多层次的数据结构。广义表的元素可以是原子，也可以是子表，而子表的元素还可以是子表等。

（2）广义表可以是递归的表。广义表的定义并没有限制元素的递归，即广义表也可以是其自身的子表。例如列表 E 就是一个递归的表。

（3）广义表可以为其他表所共享。例如，列表 A、列表 B、列表 C 是列表 D 的共享子表。在列表 D 中可以不必列出子表的值，而用子表的名称来引用。

广义表可以看成是线性表的推广，线性表是广义表的特例。广义表的结构相

当灵活，在某种前提下，可以兼容线性表、数组、树和有向图等各种常用的数据结构。

当二维数组的每行（或每列）作为子表处理时，二维数组即为一个广义表。

4.1.3　广义表的存储

广义表中的数据元素可以具有不同的结构，因此难以用顺序的存储结构来表示，而链式存储结构分配灵活，易于解决广义表的共享与递归问题，所以通常都采用链式的存储结构来存储广义表。

广义表中的元素可以是数据元素，也可以是表，因此节点的结构也需要两种：一种是表节点，用以表示表；另一种是原子节点，用以表示原子。按节点形式的不同，广义表的链式存储结构又可以分为不同的两种存储方式。

1. 头尾表示法

由于广义表中的数据元素既可能是表也可能是单元素，相应地在头尾表示法中节点的结构形式有两种：一种是表节点，用以表示表；另一种是元素节点，用以表示原子。表节点由三个域：标志域（tag=1）、表头指针域（hp）、表尾指针域（tp）组成；而原子节点由两个域：标志域（tag=0）和值域（atom）组成。广义表的存储节点可定义如下：头尾表示法的节点形式如图 4-1 所示。

(a) 表节点

(b) 原子节点

图 4-1　广义表的头尾节点表示法节点存储结构

```
typedef enum {ATOM，LIST}ElemTag；/*ATOM=0: 原子，LIST=1: 子表*/
typedef Struct GLNode
{
ElemTag tag；      /* 公共部分，用于区分原子节点和表节点 */
union
{                  /* 原子节点和表节点的联合部分 */
DataType atom；  /*atom 是原子节点的值域，DataType 由用户定义
Struct
{
struct GLNod *hp，*tp；
```

```
    }ptr;                /*ptr 是表节点的指针域，ptr.hp 和 ptr.tp 分别指向表头和
表尾
    }
    }*GList              /* 广义表类型 */
```

对于 4.1.1 节所列举的广义表 *A*，*B*，*C*，*D*，*E*，采用头尾表示法的存储方式，其存储结构如图 4-2 所示。

在这种存储结构中有几种情况：

（1）除空表的表头指针为空外，对任何非空列表，其表头指针均指向一个表节点，且该节点中的 hp 域指示列表表头（或为原子节点，或为表节点），tp 域指向列表表尾（除非表尾为空，则指针为空，否则必为表节点）。

（2）容易分清列表中原子和子表所在层次，如在列表 *D* 中，原子 *a* 和 *e* 在同一层次上，而 *b*，*c* 和 *d* 在同一层次且比 *a* 和 *e* 低一层，*B* 和 *C* 是同一层的子表。

（3）最高层的表节点个数即为列表的长度。

以上三个特点在某种程度上为列表的操作带来方便。

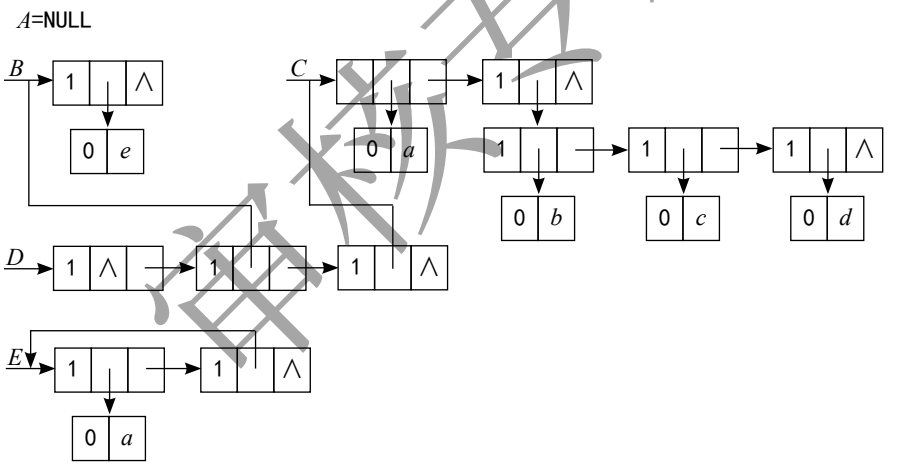


图 4-2　广义表的头尾表示法存储结构示意图

2. 孩子兄弟表示法

广义表的另一种表示法称为孩子兄弟表示法。在孩子兄弟表示法中，也有两种节点形式：一种是有孩子节点，用以表示子表；另一种是无孩子节点，用以表示原子。在有孩子节点中包括一个指向第一个孩子的指针和一个指向兄弟的指针；而在无孩子节点中包括一个指向兄弟的指针和该原子的元素值。为了能区分这两类节点，在节点中设置一个标志域。如果标志为 1，表示该节点为有孩子节点；

如果标志为 0，则表示节点为无孩子节点，如图 4-3 所示。

(a) 有孩子节点　　(b) 无孩子节点

图 4-3　广义表的孩子兄弟表示法节点存储结构

```
typedef enum{ATOM，LIST}ElemTag；
typedef struct GLNode
{
ElemTag tag；              /* 公共部分，用于区分原子节点和表节点 */
union
{                          /* 原子节点和表节点的联合部分 */
DataTyp atom;              /* 原子节点的值域 */
struct GLNode *hp;         /* 表节点的表头指针 */
};
Struct GLNode *tp;         /* 相当于线性链表的 next，指向下一个元素节点 */
}*GList;                   /* 广义表类型 GList 是一种扩展的线性链表 */
```

对于 4.1.1 节所列举的广义表 *A*、*B*、*C*、*D*、*E*，采用孩子兄弟表示法的存储方式，其存储结构如图 4-4 所示。

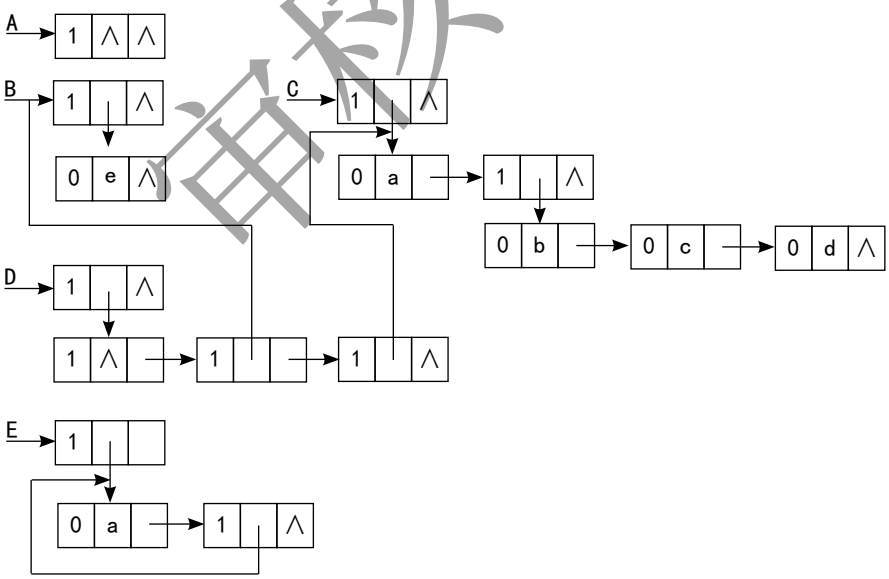


图 4-4　广义表的孩子兄弟表示法存储结构示意图

由如图 4-4 所示的存储结构示意图中可以看出，采用孩子兄弟表示法时，表

达式中的“（”（左括号）对应存储表示中的 tag=1 的节点，且最高层节点的 tp 域必为 NULL。

4.2 多维数组

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

4.2.1 多维数组的定义

数组是线性表的推广，可以看成是一种特殊的线性表，即线性表中数据元素本身也是一个线性表。它作为一种数据结构，它的特点是结构中的元素本身可以是具有某种结构的数据，但均属于同一数据类型。因此，数据结构可以简单地定义为：若线性表中的数据元素为非结构的简单元素，则称为一维数组，即向量；若一维数组中的数据元素又是一维数组结构，则称为二维数组；依次类推，若二维数组中的元素又是一个一维数组结构，则称为三维数组……

如图 4-5 所示是一个 m 行 n 列的二维数组。

$$A=\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

图 4-5 二维数组图例

其中，A 是数组结构的名称，整个数组元素可以看成是由 m 个行向量和 n 个列向量组成，其元素总数为 $m \times n$。在 C 语言中，二维数组中的数据元素可以表示成 a[表达式 1] [表达式 2]，表达式 1 和表达式 2 被称为下标表达式，如 a[i][j]。数组是数量和元素类型固定的有序序列，静态数组必须在定义的时候指定其大小和类型，而动态数组在程序运行中才分配内存空间。数组的每一个数据元素均有唯一的一组下标来标识，因此，在数组中不能进行插入和删除数据元素的操作。通常在各种高级语言中，数组一旦被定义，每一维的大小及上下界都不能改变。

在数组中通常做以下两种操作：

（1）取值操作：给定一组下标，读其对应的数据元素。

（2）赋值操作：给定一组下标，存储或修改与其对应的数据元素。

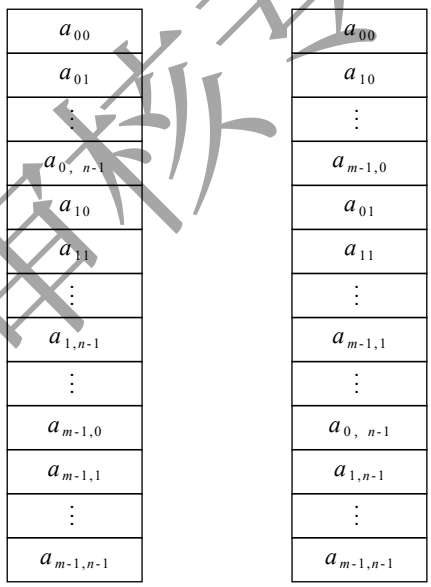
4.2.2　数组的存储结构与寻址

从理论上讲，数组可以使用两种存储结构，即顺序存储结构和链式存储结构。然而，由于数据结构没有插入和删除元素的操作，所以使用顺序存储结构更为适宜。数组一般不使用链式存储结构。

组成数组结构的元素可以是多维的，但存储数据元素的内存单元地址是唯一的。因此，在存储数组结构之前，需要解决将多维关系映射到一维关系的问题。

对于一维数组按下标顺序分配即可。对于多维数组分配时，要把其元素映像存储在一维存储器中。一般有两种存储方式：一是以行为主序（先行后列）的顺序存放，即一行分配完了接着分配下一行；另一种是以列为主序（先列后行）的顺序存放，即一列一列地分配。以行为主序的分配规律为：先排最右的下标，从右向左排，最后排最左的下标；以列为主序分配的规律恰好相反；先排最左的下标，从左向右排，最后排最右的下标。

以二维数组为例：设有 $m \times n$ 二维数组 A_{mn}，两种顺序下分配的结果如图 4-6 所示。



(a) 行序优先　　(b) 列序优先

图 4-6　二维数组的两种存储方式

对于任意元素，可以按元素的下标求其地址：

现以“以行为主序”的分配为例：设数组的基址为 $\mathrm{LOC}(a_{00})$，每个数组元素

占据 c 个地址单元，由于数组元素 a_{ij} 的前面有 i 行，每一行的元素个数为 n，在第 i 行中它的前面还有 j 个数组元素。

在 C 语言中，数组中每一维的下界定义为 0，那么 a_{ij} 的物理地址可用以下公式计算，有

$$\mathrm{LOC}(a_{ij}) = \mathrm{LOC}(a_{00}) + (i*n + j) * c \qquad (4\text{-}1)$$

同理对于三维数组 A_{mnp}，即 $m \times n \times p$ 数组，对于数组元素 a_{ijk} 其计算公式为

$$\mathrm{LOC}(a_{ijk})=\mathrm{LOC}(a_{000})+(i*n*p+j*p+k)*c \qquad (4\text{-}2)$$

4.3 矩阵的压缩存储

4.3.1 特殊矩阵的压缩存储

1. 压缩矩阵

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

2. 特殊矩阵

假若值相同的元素或者零元素在矩阵中的分布有一定规律，则被称为特殊矩阵；反之称为稀疏矩阵。

（1）对称矩阵。对称矩阵是指在一个 n 阶方阵中，有 $a_{ij}=a_{ji}$，其中 $0 \leqslant i < n$，$0 \leqslant j < n$。对称矩阵关于主对角线对称，为了节省空间，只需存储上三角或下三角部分即可。例如，只存储下三角的值，对角线之上的值通过对称关系映射过去即可。这样，原来需要 n^2 个存储单元，现在只需要 $n(n+1)/2$ 个存储单元，节省了大约一半的存储空间。

矩阵下三角部分中第 0 行有一个非零元素，第 1 行有 2 个非零元素，以此类推，SA 共需 $\sum_{i=0}^{n-1}(i+1)=n(n+1)/2$ 个存储单元即可。某元素 a_{ij}（$i \geqslant j$ 且 $0 \leqslant i$，$j \leqslant n-1$）前面有 i 行，共有 $1+2+\cdots+i=i*(i+1)/2$ 个元素，而又是其所在行中的第 $j+1$ 个，所以 a_{ij} 是 SA 中的第 $i*(i+1)/2+j+1$ 个元素，因此它在 SA 中的下标 k 与 i 和 j 的关系为：$k=i*(i+1)/2+j(i \geqslant j)$。

对于上三角部分的元素 $a_{ij}(i<j)$，有：$a_{ij}=a_{ji}$，访问与它对应的下三角中的元素 a_{ji} 即可，即 $k=j*(j+1)/2+i$。

综上所述，在 SA[0…$n(n+1)/2-1$] 中存储对称矩阵时，数组 SA 的下标 k 与矩阵元素 a_{ij} 的下标 i 和 j 的关系为

$$k=\begin{cases} i(i+1)/2+j, & i\geqslant j \\ j(j+1)/2+i, & i<j \end{cases} \tag{4-3}$$

（2）三角矩阵。三角矩阵是指 n 阶矩阵中的下三角（或上三角）的元素都是 0 或者一个常数的矩阵，如图 4-7 所示，可以分为上三角矩阵和下三角矩阵。图 4-7 中，c 为某个常数，图 4-7(a) 为下三角矩阵：主对角线以上均为同一个常数；图 4-7(b) 为上三角矩阵，主对角线以下均为同一个常数。

$$A=\begin{bmatrix} a_{00} & c & c & c \\ a_{10} & a_{11} & c & c \\ a_{20} & a_{21} & a_{22} & c \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}\quad (a)$$

$$A=\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ c & a_{11} & a_{12} & a_{13} \\ c & c & a_{22} & a_{23} \\ c & c & c & a_{33} \end{bmatrix}\quad (b)$$

图 4-7　三角矩阵图例

注意到三角矩阵的特点，与对称矩阵相似。下三角矩阵除了存储下三角中的元素之外，还要存储对角线上方的常数。因为该常数值相等，所以只存一个即可。这样，下三角矩阵一共需要存储 $n(n+1)/2+1$ 个元素，将其存入 SA[0…$n(n+1)/2$] 数组中，数组 SA 的下标 k 与矩阵元素的下标 i 和 j 的关系为

$$k=\begin{cases} i(i+1)/2+i, & i\geqslant j \\ n(n+1)/2, & i<j \end{cases} \tag{4-4}$$

（3）对角矩阵。对角矩阵也称为带状矩阵，这类矩阵所有的非零元素都集中在以主对角线为中心的带状区域中，即除了主对角线和其上下方若干条对角线的元素外，所有其他元素都为零（或同一个常数 c）。现在以三角矩阵为例，讲述对角矩阵的压缩存储，如图 4-8 所示。

$$A=\begin{bmatrix} 2 & 5 & 0 & 0 & 0 \\ 8 & 7 & 9 & 0 & 0 \\ 0 & 5 & 8 & 6 & 0 \\ 0 & 0 & 1 & 3 & 4 \\ 0 & 0 & 0 & 5 & 9 \end{bmatrix}$$

图 4-8　对角矩阵图例

三对角矩阵是指三条对角线以为的数据元素均为 0，且第一行和最后一行只有两个非零元素，其他行均为三个非零元素。因此，如果用一维数组存储矩阵中的非零元素，需要存储 $2+3(n-2)+2=3n-2$ 个非零元素。其压缩存储形式如图 4-9 所示。

$k=$	0	1	2	3	4	5	6	7	…	$3n-3$
矩阵	a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	…	$a_{n-1,n-1}$

图 4-9　三对角矩阵的压缩存储形式

三对角矩阵 A 中任一元素压缩存储后的向量 SA 中的序号为 $2+3(i-1)+(j-i+2)=2i+j+1$。由于 C 语言中数组下标从 0 开始，元素在向量 SA 中的下标为

$$k=2i+j,\ 0 \leqslant k \leqslant 3n-3 \qquad (4\text{-}5)$$

4.3.2　稀疏矩阵的压缩存储

在科学管理及工程计算中，常会遇到阶数很高的大型稀疏矩阵。稀疏矩阵是指矩阵中非零元素非常少，且分布不规律的矩阵。对于非零元素的比例并没有一个确定的定义。为此提出了只存储非零元素的方法。由于这类矩阵非零元素分布没有规律，为了能找到相应的元素，只存储非零元素的值是不够的，还要记下该值所在的行列下标。考虑处理方便，设定矩阵每一维的下界从 1 开始，于是采取如下方法：将非零元素所在的行、列以及它的值构成一个三元组（i，j，v），然后按某种规律存储这些三元组。显然，若要唯一地表示一个稀疏矩阵，还需要加上该矩阵的行、列值。如图 4-10 所示的稀疏矩阵 A，可以用三元组表 [（1，2，15），（2，3，7），（3，1，8），（3，4，-5），（4，6，12），（5，2，18），（6，3，25）] 加上行、列值（6，7）进行描述。

$$A=\begin{bmatrix} 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & -5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 25 & 0 & 0 & 0 & 0 \end{bmatrix},\quad B=\begin{bmatrix} 0 & 0 & 8 & 0 & 0 & 0 \\ 15 & 0 & 0 & 0 & 18 & 0 \\ 0 & 7 & 0 & 0 & 0 & 25 \\ 0 & 0 & -5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 12 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图 4-10　稀疏矩阵 A 和矩阵 B 图例

1. 稀疏矩阵的三元组表示

采用顺序存储结构表示稀疏矩阵时，将三元组按行优先的顺序且同一行中列

号从小到大的规律排列成一个线性表，成为三元组顺序表，如图 4-11(a) 所示。为了运算方便，矩阵的非零元素的个数也同时存储。

i	j	v
1	2	15
2	3	7
3	1	8
3	4	−5
4	6	12
5	2	18
6	3	25

(a) a. date

i	j	v
1	3	8
2	1	5
2	5	8
3	2	7
3	6	25
4	3	−5
6	4	2

(b) b. date

图 4-11　矩阵的三元组顺序表图例

在 C 语言中，三元组表定义如下：

```
#define SMAX  1024                  /* 一个足够大的数 */
typedef  struct
{
  int i,j;                          /* 非零元素的行、列 */
  DataType  v;                      /* 非零元素值 */
}SPNode;                            /* 三元组类型 */
typedef  struct
{
   int mu,nu,tu;                    /* 矩阵的行、列及非零元素的个数 */
   SPNode  data[SMAX];              /* 三元组表 */
 } SPMatrix;                        /* 三元组表的存储类型 */
```

这样的存储方法确实节省了存储空间，但矩阵的运算会变得复杂些。下面我们讨论这种存储方式下稀疏矩阵的转置运算。

转置运算是一种最简单的矩阵运算。对于一个 $m \times n$ 的矩阵 SPMatrixA，其转置矩阵 SPMatrixB 则是一个 $n \times m$ 的矩阵，且 $A(i, j) = B(j, i)$，$1 \leqslant i \leqslant n$，$1 \leqslant j \leqslant m$。如图 4-10 所示的稀疏矩阵 A 和 B 互为转置矩阵。

稀疏矩阵转置后仍然是稀疏矩阵，我们前面规定三元组是按行序且每行中的元素是按列号从小到大的规律顺序存放的，因此 B 也必须按此规律实现。B 的三元组顺序表如图 4-11(b) 所示。为了运算方便，矩阵的行和列都从 1 算起，三元组表 data 也从 1 单元用起。

假设 a 和 b 是 SPMatrix 类型的变量，分别表示矩阵 A 和 B，那么如何由 a 得到 b 呢?

方法一：

观察如图 4-11 所示的 (a) 和 (b)，可以看到从 *a* 到 *b* 需要做到：

（1）*a* 的行、列值转化成 *b* 的列、行值。

（2）将每个三元组中的 *i* 和 *j* 相互调换。

（3）重排三元组之间的次序。

重点在（3）如何实现，算法思路如下：

1）*A* 的行、列转化成 *B* 的行、列。

2）由于 *B* 的顺序是按照 *A* 的列序排列的，为了得到 b.data 的相应数据，需按照 *A* 的列序进行转置，也就是说，为了依次找到 *A* 每一列中所有的非零元素，需要在其三元组表 a.data 中从第一行起整个扫描一遍，并将找到的每个三元组的行、列交换后顺序存储到 b.data 中，恰好就是 b.data 中的应用顺序。算法如下：

```
SPMatrix * TransM1 (SPMatrix *A)                    /* 稀疏矩阵转置算法 */
{
  SPMatrix *B;
  int p,q,col;
  B=(SPMatrix *)malloc(sizeof(SPMatrix));           /* 申请存储空间 */
  B->mu=A->nu; B->nu=A->mu; B->tu=A->tu;
    /* 稀疏矩阵的行、列、元素个数赋值 */
  if (B->tu>0)                                      /* 有非零元素则转换 */
  {
     q=0;
  for (col=1; col<=(A->nu); col++)                  /* 按 A 的列序转换 */
     for (p=1; p<= (A->tu); p++)                    /* 扫描整个三元组表 */
      if (A->data[p].j==col )
      {
        B->data[q].i= A->data[p].j ;
        B->data[q].j= A->data[p].i ;
        B->data[q].v= A->data[p].v;
        q++;
      }
  }
  return B;                                         /* 返回转置矩阵的指针 */
}  /*TransM1*/
```

方法二：改进思路

b.data 第 1 个位置的数据一定是 A 中第 1 列的第 1 个非零元素，如果知道 A 第 1 列的非零元素的个数，那么第 1 列的第 1 个非零元素在 b.data 中的位置加上第 1 列的非零元素的个数就是第 2 列的第 1 个非零元素在 b.data 中的位置，如此类推，第 n 列的第一个非零元素在 b.data 中的位置就等于第 $n-1$ 列在 b.data 中的位置加上第 $n-1$ 列中非零元素的个数。而且，A 中的同一行，列数是从小到大排列的，那么对 A.data 只需从上到下扫描一遍即可。

在此之前，需准备好两组数据，即 A 每列的非零元个数和 A 中每列第 1 个非零元在 b.data 中所处的位置。在此引入两个向量来描述：num[col] 和 cpot[col]，num[col] 表示矩阵 A 中第 col 列的非零元素的个数（为了方便均从 1 单元用起），cpot［col］初始值表示矩阵 A 中的第 col 列的第一个非零元素在 b.data 中的位置。已知 cpot 的初始值为

$$\begin{aligned} &\text{cpot[col]=1;} \quad \text{col=1} \\ &\text{cpot[col]=cpot[col-1]+num[col-1];} \quad 2 \leqslant \text{col} \leqslant \text{n} \end{aligned} \tag{4-6}$$

例如，图 4-12 矩阵 A 的 num 和 cpot 的值如下：

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

图 4-12　矩阵 A 的向量 cpot 的值

依次扫描 a.data，当扫描到一个 col 列元素是，直接查找 cpot[col]，将其存放在 b.data 的该位置上，并且 cpot[col] 加 1，cpot[col] 中始终是下一个 col 列元素在 b.data 中的位置。算法如下：

```
#define ArraySize 100                /* 根据情况设定数组空间大小 */
SPMatrix * TransM2 (SPMatrix *A)     /* 稀疏矩阵转置改进算法 */
{
  SPMatrix *B;
  int  i,j,k;
  int num[ArraySize],cpot[ArraySize];
  B=(SPMatrix *)malloc(sizeof(SPMatrix));     /* 申请存储空间 */
  B->mu=A->nu; B->nu=A->mu; B->tu=A->tu;
  /* 稀疏矩阵的行、列及元素个数赋值 */
  if (B->tu>0)                       /* 有非零元素则转换 */
```

```
    {
      for (i=1;i<=A->nu;i++)
        num[i]=0;
    for (i=1;i<=A->tu;i++)                /* 求矩阵 A 中每一列非零元素的个数 */
      {  j= A->data[i].j;
         num[j]++;
       }
      cpot[1]=1;                          /* 求矩阵 A 中每一列第一个非零元素在
B.data 中的位置 */
      for (i=2;i<=A->nu;i++)
        cpot[i]= cpot[i-1]+num[i-1];
      for (i=1; i<= (A->tu); i++)         /* 扫描三元组表 */
      {
        j=A->data[i].j;                   /* 当前三元组的列号 */
        k=cpot[j];                        /* 当前三元组在 B.data 中的位置 */
        B->data[k].i= A->data[i].j ;
        B->data[k].j= A->data[i].i ;
        B->data[k].v= A->data[i].v;
        cpot[j]++;
       }
     }
      return B;                           /* 返回的是转置矩阵的指针 */
    }  /*TransM2*/
```

2. 稀疏矩阵的十字链表表示

三元组表可以看作是稀疏矩阵的顺序存储结构表示，但矩阵的非零元素数目和位置在操作过程中变化较大时，顺序存储结构表示的三元组表处理十分不便，对这种类型的矩阵采用链式存储结构更为合适。在本节中，我们介绍稀疏矩阵的一种链式存储结构 —— 十字链表。在某些情况下，采用十字链表表示稀疏矩阵是很方便的。

在十字链表中，每个非零元素存储为一个节点，元素节点由 5 个域组成。其中，*i* 域表示非零元素的行号，*j* 域表示非零元素的列号，*v* 域表示该非零元素的值，还有两个指针域：right 指针，用来链接同一行中下一个非零元素；down 指针，用来链接同一列中的下一个非零元素。这样，每一行每一列都链接成为一个线性链表，每个节点既是行又是列中的一个节点，整个矩阵构成十字交叉的链表，所以称为十字链表。其节点结构如图 4-13 所示。

图 4-13　十字链表的节点结构

为方便操作，每一行每一列增设一类表头节点，也由 5 个域组成。其中行、列的值均为 0，无实际意义。每一行表头节点的 right 域指向该行的第一个元素节点，每一列表头节点的 down 域指向该列的第一个元素节点。为了方便地找到每一行或每一列，将每行（列）的头节点链接起来，由于表头节点的值域空闲，用表头节点的值域作为链接各头节点的链域，即第 i 行（列）的头节点的值域指向第 i+1 行（列）的头节点，形成一个循环表。该循环表再增加一个头节点，也就是最后的总头节点，用指针 H 指向它。总头节点的 i 和 j 域存储矩阵的行数和列数。

由于非零元素节点的值域是 DataType 类型，在表头节点中需要设置成指针类型，为了使整个结构的节点一致，我们规定表头节点和其他节点结构相同，因此该域用一个联合来表示。综上，节点的结构定义如图 4-14 所示。

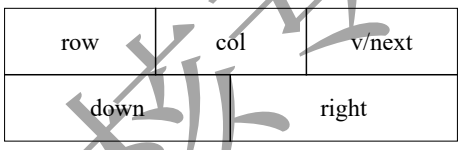


图 4-14　表头节点的结构

十字链表的类型定义如下：

```
typedef struct node
{
    int row,col;
    struct node *down,*right;
    union v_next
    {
            DataType v;
            struct node *next;
    }
}MNode,*MLink;
```

4.4 项 目 训 练

项目：统计成绩

【题目要求】

（1）使用数组和指针统计成绩。

（2）要求输入班级以下各科考试平均成绩：数学、物理、外语、政治、体育，要求统计出全班本学期总平均成绩以及得分最低的科目和该科目的成绩。

（3）本题考虑如何为字符分配内存，并将字符数组的内容也存入所分配的内存中，使用指针实现设计要求。

【算法分析】

设计步骤如下。

（1）设计循环将各门课成绩和总人数读入数组。

（2）为各门课程名称申请内存并赋值。

（3）为所求各项申请内存并赋值。

（4）求最小值和平均值。

（5）作相应的赋值及输出。

本 章 小 结

（1）数组是线性表的推广，数组作为一种数据结构，其特点是结构中的元素本身可以具有某种结构的数据，但属于同一类数据类型。一般的数组结构使用顺序存储结构，对多维数组分配时，要把它的元素映象存储在一维存储器中，一般有两种存储方式：“先行后列”和“先列后行”的顺序存放。

（2）矩阵通常用二维数组来表示，某些特殊矩阵，如三角矩阵、对称矩阵、对角矩阵、稀疏矩阵等，此类阶数很高而相同值或零元素很多，为了节约存储空间，可以对这类矩阵进行压缩存储。

（3）广义表是线性表的推广，广泛地用于人工智能等领域，把广义表作为基本的数据结构，程序也可表示为一系列的广义表。

习　　题

应用题

1. 设有一个二维数组 A[m][n]，假设 A[0][0] 存放位置在 644，A[2][2] 存放位置在 676，每个元素占一个空间，问 A[3][3] 存放在什么位置？

2. 设有一个 $n*n$ 的对称矩阵 A，试问：

（1）存放对称矩阵 A 上三角部分或下三角部分的一维数组 B 有多少元素？

（2）若在一维数组 B 中从 0 号位置开始存放，则对称矩阵中的任一元素在指导存下三角部分的情形下应存于一维数组的什么下标位置？给出计算公式。

第5章 树

在前面几章里讨论的数据结构都属于线性结构，线性结构的特点是逻辑结构简单，易于进行查找、插入和删除等操作，其主要用于对客观世界中具有单一的前驱和后继的数据关系进行描述，而现实中的许多事物的关系并非这样简单，如人类社会的族谱、各种社会组织机构、城市交通和通信等。这些事物中的联系都是非线性的，采用非线性结构进行描绘会更明确和便利。

非线性结构是指，在该结构中至少存在一个数据元素，有两个或两个以上的直接前驱（或直接后继）元素。树形结构和图形结构就是其中十分重要的非线性结构，可以用来描述客观世界中广泛存在的层次结构和网状结构的关系，如前面提到的族谱、城市交通等。在本书的第5，6章将重点讨论这两类非线性结构的有关概念、存储结构、在各种存储结构上所实施的一些运算以及有关的应用实例。

本章首先介绍树形结构中最简单、应用十分广泛的二叉树结构，然后对具有更一般意义的树结构进行讨论。

知识目标

- 了解树的基本概念和树的基本操作。
- 掌握二叉树的定义和基本性质。
- 掌握二叉树的存储结构和遍历方式。
- 了解树、森林与二叉树的相互转换。
- 掌握哈夫曼树的构造方法和哈夫曼编码方法。

能力目标

- 针对具体问题，能应用二叉树的知识进行分析，找到解决问题的方法并实现。

5.1　树的概述

5.1.1　树的定义和基本术语

1. 树的定义

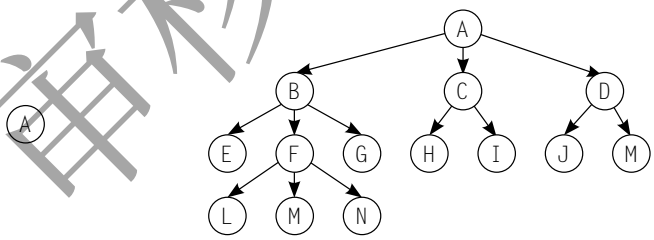
树（Tree）是 n（$n \geqslant 0$）个节点的有限集合。当 $n=0$ 时，称为空树；当 $n>0$ 时，称为非空树。任意一棵非空树满足以下两个条件：

（1）有且仅有一个特定的称为根（Root）的节点，它没有前驱；

（2）当 $n>0$ 时，除了根节点之外的其余节点被分为 m（$m \geqslant 0$）个互不相交的子集合 T_1，T_2，…，T_m，其中每一个集合又是一棵树，并称为这个根节点的子树（Subtree）。

显然，从树的定义上看，树是递归的。

树的示意图如图 5-1 所示。



(a) 只有根节点的树　　(b) 一般的树

图 5-1　树的示意图

2. 树的基本术语

（1）节点的度。节点所拥有的子树的个数称为该节点的度。

（2）叶节点。度为 0 的节点称为叶节点，或者称为终端节点。

（3）分枝节点。度不为 0 的节点称为分支节点，或者称为非终端节点。一棵树的节点除叶节点外，其余的都是分支节点。

（4）左孩子、右孩子、双亲、兄弟。树中一个节点的子树的根节点称为这个节点的孩子。在二叉树中，左子树的根称为左孩子，右子树的根称为右孩子。

这个节点称为它孩子节点的双亲。具有同一个双亲的孩子节点互称为兄弟。

（5）路径、路径长度。如果一棵树的一串节点 n_1，n_2，…，n_k 有如下关系：节点 n_i 是 n_i+1 的父节点（$1 \leqslant i<k$），就把 n_1，n_2，…，n_k 称为一条由 n_1 至 n_k 的路径。这条路径的长度是 $k-1$。

（6）祖先、子孙。在树中，如果有一条路径从节点 M 到节点 N，那么 M 就称为 N 的祖先，而 N 称为 M 的子孙。

（7）节点的层数。规定树的根节点的层数为 1，其余节点的层数等于它的双亲节点的层数加 1。

（8）树的深度。树中所有节点的最大层数称为树的深度。

（9）树的度。树中各节点度的最大值称为该树的度。

（10）森林。m（$m \geqslant 0$）棵互不相交的树的集合构成森林。任意一棵树，删去根节点就变成了森林。

5.1.2 树的抽象数据类型定义

树的抽象数据类型定义如下：

ADT Tree{

Data：Date 是具有相同数据类型及层次关系的数据元素的集合。

Operation

InitTree(&T)：构造空树 T。

CreateTree(&T,definition)：创建一棵树，返回指向根节点的指针。

DestroyTree(&T)：销毁树 T。

TreeEmpty(T)：若 T 为空树，返回 true，否则返回 false。

TreeDepth(T)：返回 T 的深度。

GetRoot(T)：返回 T 的根节点。

Parent(T,X)：若 X 是树 T 的非根节点，则返回它的双亲，否则返回空。

LeftChild（T,x）：若 x 是树 T 的非根节点，则返回它的最左孩子，否则返回空。

RightSibling(T,x)：若 x 有右兄弟，返回它的右兄弟，否则返回空。

InsertChi1d(&T,&p,i,c)：p 指向树 T 的某个节点，i 为所指节点 p 的度上加 1，非空树 c 与 T 不相交，操作结果为插入 c 为树 T 中 p 指节点的第 i 棵子树。

DeleteChild(&T,&p,i)：其中 p 指向树 T 的某个节点，i 为所指节点 p 的度，操作结果为删除 T 中 p 所指节点的第 i 棵子树。

}ADT Tree

5.1.3　树的遍历

树最基本的操作是遍历。树的遍历是指从根节点出发，按照某种次序访问树中所有节点，使得每个节点被访问一次且仅被访问一次。

由树的定义可知，一棵树由根节点和 m 棵子树构成，因此，只要依次遍历根节点和 m 棵子树，就可以遍历整颗树。树通常有前序（根）遍历、后序（根）遍历和层序（次）遍历三种方式。

1. 前序遍历

树的前序遍历操作定义为：

若树为空，则空操作返回；否则

（1）访问根节点；

（2）按照从左到右的顺序前序遍历根节点的每一棵子树。

对图 5-2 所示的树进行前序遍历，结果为：A B D E H I F C G。

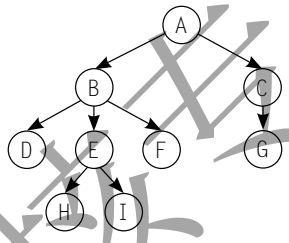


图 5-2

2. 后续遍历

树的后序遍历操作定义为：

若树为空，则空操作返回；否则

（1）按照从左到右的顺序后序遍历根节点的每一棵子树；

（2）访问根节点。

对图 5.2 所示的树进行后序遍历，结果为：D H I E F B G C A。

3. 层序遍历

树的层序遍历操作定义为：

从树的第一层（即根节点）开始，自上而下逐层遍历，在同一层中，按从左到右的顺序对节点逐个访问。

对图 5.2 所示的树进行层序遍历，结果为：A B C D E F G H I 。

5.1.4 树的存储结构

树中的某个节点的孩子可以有多个，所以仅仅使用简单的顺序结构或者链式结构是不能完全表示一整棵树的。但是充分利用顺序存储结构和链式存储结构的特点，完全可以实现对树的存储结构的表示。我们表示一棵树的方法有：双亲表示法，孩子表示法，孩子兄弟表示法。

1. 双亲表示法

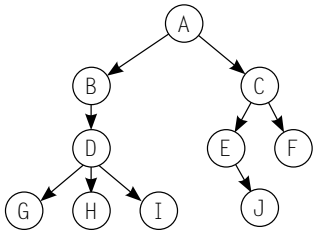
双亲作为索引关键词的一种存储方式，每个节点只有一个双亲，所以主要使用顺序存储结构。以一组连续空间存储树的节点，同时在每个节点中附设一个指示其双亲节点位置的指针域。节点结构如图 5-3 所示：

图 5-3 节点结构图

节点结构定义：

```
/* 树的双亲表示法节点结构定义 */
#defne MAX_TREE_SZE 100
typedef int TElemType;
typedef struct PTNode{                          // 节点结构

   TElemType data;                              // 节点数据
   int parent;                                  // 双亲位置
}PTNode;

typedef struct {                                // 树结构
   PTNode nodes[MAX_TREE_SZE];                  // 节点数组
   int r, n;            //r 是根位置，n 是节点数
}PTree;
```

如图 5-4 所示是一棵树及其双亲表示的存储结构。树中的节点一般按层序存储在一维数组中，根节点没有双亲，其双亲域为 -1。在这种存储结构中，根据节点的双亲域值就可以找到该节点的双亲节点。例如节点 D 的双亲节点下标为 1，数组中下标为 1 的节点是 B，所示 B 就是 D 的双亲节点。



下标	date	parent
0	A	−1
1	B	0
2	C	0
3	D	1
4	E	2
5	F	2
6	G	3
7	H	3
8	I	3
9	J	4

图 5-4　树的双亲表示法

双亲表示法的优点是 parent 指针域指向数组下标，所以找双亲节点的时间复杂度为 O(1), 向上一直找到根节点也快。缺点是由上向下找就十分慢，若要找节点的孩子或者兄弟，要遍历整个树。

2. 孩子表示法

孩子表示法也叫多重链表表示法。每个节点可能含有 m 个孩子，即每个节点发出 m 个链，分别指向它的一个孩子。

date	degree	Child 1	Child 2	……	Child d

图 5-5　节点结构图

其中，data：数据域，存放节点的数据信息；

deree：度域，存放该节点的度；

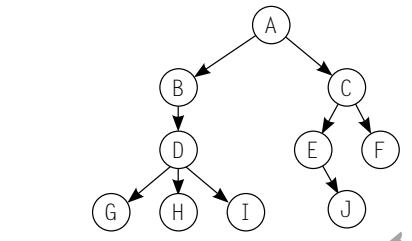
Child1~ Childd：指针域，指向该节点的孩子节点。

/* 树的孩子表示法节点结构定义 */

```
#define MAX_TREE_SIZE 100
typedef int TElemType;
typedef struct PTNode                    // 节点结构
{
   TElemType data;                       // 节点数据
   int child1;                           // 孩子 1 节点
   int child2;                           // 孩子 2 节点
   int child3;                           // 孩子 3 节点
```

```
}PTNode;

typedef struct                          // 树结构
{
   PTNode nodes[MAX_TREE_SIZE];         // 节点数组
   int r, n;                            //r 是根位置，n 是节点数
}PTree;
```

在用孩子表示法来表示树时，每个节点的指针域个数等于该节点的孩子个数。图 5-6 是孩子表示法存储示意图。



下标	date	Child 1	Child 2	Child 3
0	A	1	2	−1
1	B	3	−1	−1
2	C	4	5	−1
3	D	6	7	8
4	E	9	−1	−1
5	F	−1	−1	−1
6	G	−1	−1	−1
7	H	−1	−1	−1
8	I	−1	−1	−1
9	J	−1	−1	−1

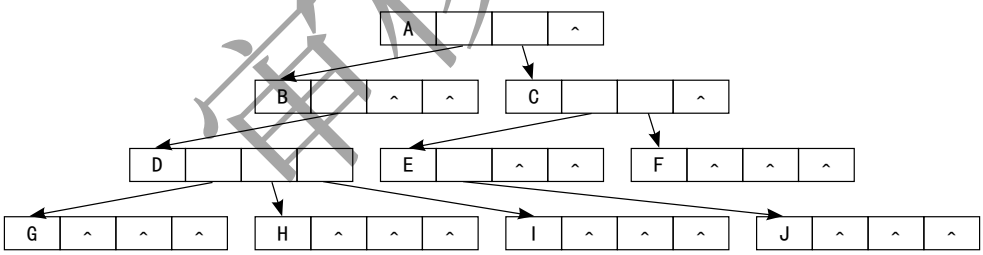


图 5-6　树的孩子表示法存储示意图

3. 孩子兄弟表示法

任意一棵树，它的节点的第一个孩子如果存在就是唯一节点，它的右兄弟如果存在，也是唯一的，因此，我们设置两个指针，分别指向该节点的第一个孩子和该节点的右兄弟。

在这种存储方式下，每个节点包含三部分内容：节点值、指向该节点第一个孩子的指针和指向该节点下一个兄弟节点的指针。图 5-7 所示节点的结构表。图

5-8 所示树的孩子兄弟链表表示法。

date	first child	rightsib

图 5-7　节点的结构表

孩子兄弟表示法节点结构定义：

```
typedef int TElemType;

typedef struct CSNode
{
   TElemType data;
   struct CSNode * firstchild, *rightsib;
}CSNode,*CSTree;
```

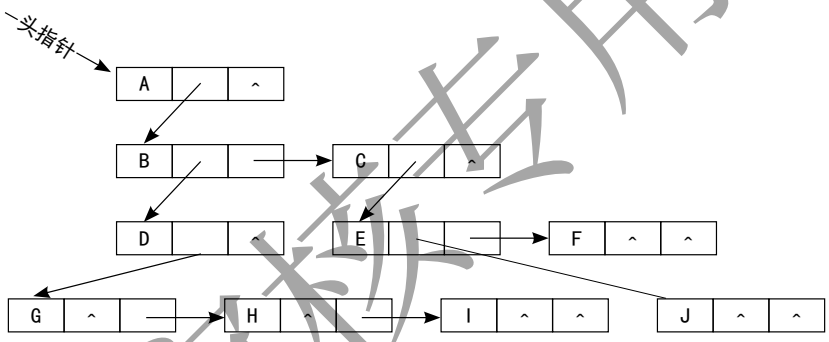


图 5-8　树的孩子兄弟表示法存储示意图

5.2　二叉树

5.2.1　二叉树的基本概念

1. 二叉树（Binary Tree）

二叉树是有限个元素的集合，该集合或者为空，或者由一个称为根 (Root) 的元素及两个不相交的、分别称为左子树和右子树的二叉树组成。当集合为空时，称该二叉树为空二叉树。在二叉树中，一个元素也称作一个节点。

二叉树是有序的，若将其左、右子树颠倒，就成为另一棵不同的二叉树。即

使树中节点只有一棵子树，也要区分它是左子树还是右子树。因此二叉树具有五种基本形态，如图 5-9 所示。

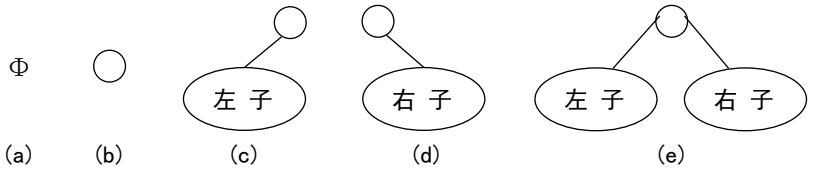


图 5-9　二叉树的 5 种基本形态

2. 二叉树的相关概念

（1）满二叉树。在一棵二叉树中，如果所有分支节点都存在左子树和右子树，并且所有叶子节点都在同一层上，这样的一棵二叉树称作满二叉树。如图 5-10(a) 所示。若所有节点要么是含有左右子树的分支节点，要么是叶子节点，但由于其叶子未在同一层上，则不是满二叉树，如图 5-10(b) 所示。

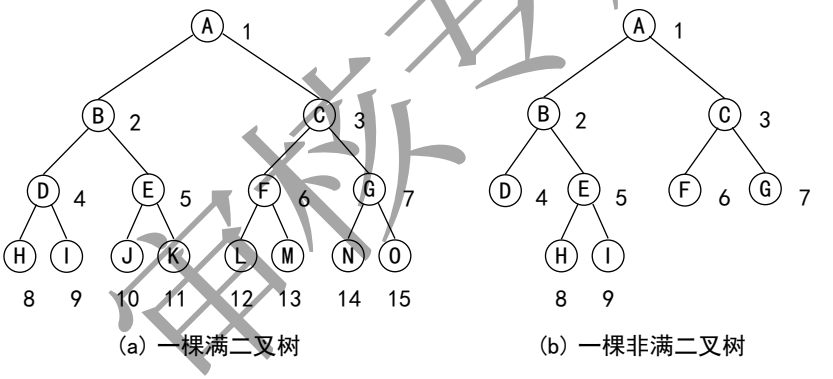
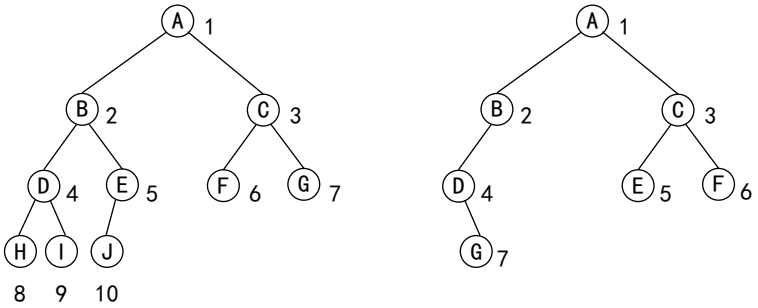


图 5-10　满二叉树和非满二叉树示意图

（2）完全二叉树。一棵深度为 k 的有 n 个节点的二叉树，对树中的节点按从上至下、从左到右的顺序进行编号，如果编号为 i（$1 \leqslant i \leqslant n$）的节点与满二叉树中编号为 i 的节点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。完全二叉树的特点是：叶子节点只能出现在最下层和次最下层，且最下层的叶子节点集中在树的左部。显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。图 5-11(a) 所示为一棵完全二叉树，图 5-10(b) 和图 5-11(b) 都不是完全二叉树。



(a) 一棵完全二叉树　　(b) 一棵非完全二叉树

图 5-11　完全二叉树和非完全二叉树示意图

5.2.2　二叉树的主要性质

性质 1　一棵非空二叉树的第 i 层上最多有 2^{i-1} 个节点（$i \geqslant 1$）。

该性质可由数学归纳法证明。证明略。

性质 2　一棵深度为 k 的二叉树中，最多具有 $2^k - 1$ 个节点。

证明：设第 i 层的节点数为 x_i（$1 \leqslant i \leqslant k$），深度为 k 的二叉树的节点数为 M，由性质 1 可知，x_i 最多为 2^{i-1}，则有

$$M = \sum_{i=1}^{k} x_i \leqslant \sum_{i=1}^{k} 2^{i-1} = 2^k - 1 \qquad (5\text{-}1)$$

性质 3　对于一棵非空的二叉树，如果叶子节点数为 n_0，度数为 2 的节点数为 n_2，则有

$$n_0 = n_2 + 1 \qquad (5\text{-}2)$$

证明：设 n 为二叉树的节点总数，n_1 为二叉树中度为 1 的节点数，则有

$$n = n_0 + n_1 + n_2 \qquad (5\text{-}3)$$

在二叉树中，除根节点外，其余节点都有唯一的进入分支。设 B 为二叉树中的分支数，那么有

$$B = n - 1 \qquad (5\text{-}4)$$

这些分支是由度为 1 或度为 2 的节点发出的，一个度为 1 的节点发出一个分支，一个度为 2 的节点发出两个分支，所以有

$$B = n_1 + 2n_2 \qquad (5\text{-}5)$$

综合式（5-3）、式（5-4）、式（5-5）可得

$$n_0 = n_2 + 1 \tag{5-6}$$

性质 4 具有 n 个节点的完全二叉树的深度 k 为 $[\log_2 n]+1$。

证明：根据完全二叉树的定义和性质 2 可知，当一棵完全二叉树的深度为 k、节点个数为 n 时，有

$$2^{k-1} - 1 < n \leqslant 2^k - 1 \tag{5-7}$$

即

$$2^{k-1} \leqslant n < 2^k \tag{5-8}$$

对不等式取对数，有

$$k - 1 \leqslant \log_2 n < k \tag{5-9}$$

由于 k 是整数，所以有 $k=[\log_2 n]+1$。

性质 5 对于具有 n 个节点的完全二叉树，如果按照从上至下和从左到右的顺序对二叉树中的所有节点从 1 开始顺序编号，则对于任意的序号为 i 的节点，有

（1）如果 $i>1$，则序号为 i 的节点的双亲节点的序号为 $i/2$（“/”表示整除）；如果 $i=1$，则序号为 i 的节点是根节点，无双亲节点。

（2）如果 $2i \leqslant n$，则序号为 i 的节点的左孩子节点的序号为 $2i$；如果 $2i>n$，则序号为 i 的节点无左孩子。

（3）如果 $2i + 1 \leqslant n$，则序号为 i 的节点的右孩子节点的序号为 $2i + 1$；如果 $2i + 1>n$，则序号为 i 的节点无右孩子。

如果对二叉树的根节点从 0 开始编号，则相应的 i 号节点的双亲节点的编号为 $(i-1)/2$，左孩子的编号为 $2i + 1$，右孩子的编号为 $2i + 2$。

此性质可采用数学归纳法证明。证明略。

5.3 二叉树的基本操作与存储实现

5.3.1 二叉树的存储

1. 顺序存储结构

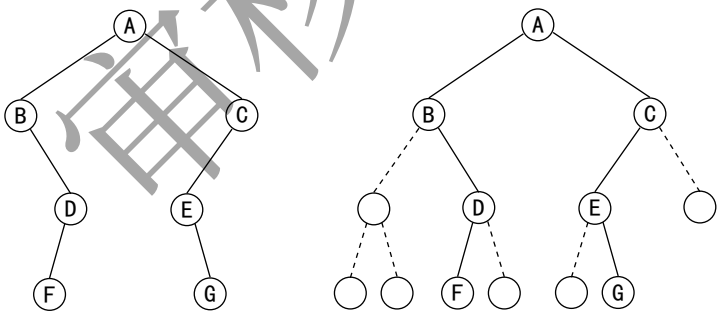
二叉树的顺序存储，就是用一组连续的存储单元存放二叉树中的节点。一般是按照二叉树节点从上至下、从左到右的顺序存储。这样节点在存储位置上的前

驱后继关系并不一定就是它们在逻辑上的邻接关系，然而只有通过一些方法确定某节点在逻辑上的前驱节点和后继节点，这种存储才有意义。因此，依据二叉树的性质，完全二叉树和满二叉树采用顺序存储比较合适，树中节点的序号可以唯一地反映出节点之间的逻辑关系，这样既能够最大可能地节省存储空间，又可以利用数组元素的下标值确定节点在二叉树中的位置，以及节点之间的关系。如图 5-11(a) 所示的完全二叉树的顺序存储示意如图 5-12 所示。

	A	B	C	D	E	F	G	H	I	J
数组下标	0	1	2	3	4	5	6	7	8	9

图 5-12　完全二叉树的顺序存储示意图

对于一般的二叉树，如果仍按从上至下和从左到右的顺序将树中的节点顺序存储在一维数组中，则数组元素下标之间的关系不能够反映二叉树中节点之间的逻辑关系，只有增添一些并不存在的空节点，使之成为一棵完全二叉树的形式，然后再用一维数组顺序存储。如图 5-13 所示给出了一棵一般二叉树改造后的完全二叉树形态和其顺序存储状态示意图。显然，这种存储对于需增加许多空节点才能将一棵二叉树改造成为一棵完全二叉树的存储时，会造成空间的大量浪费，不宜用顺序存储结构。最坏的情况是右单支树，如图 5-14 所示，一棵深度为 k 的右单支树，只有 k 个节点，却需分配 $2^k - 1$ 个存储单元。

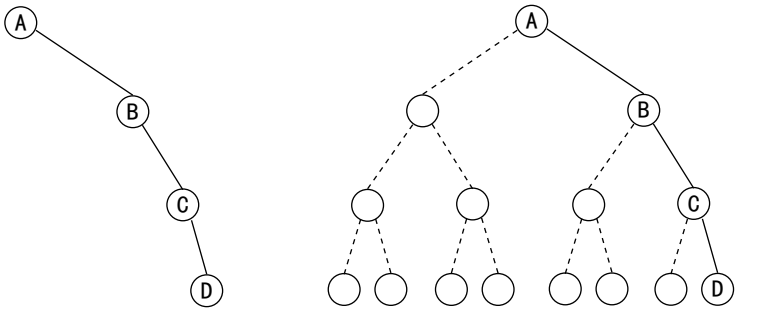


(a) 一般二叉树　　(b) 改造后的完全二叉树

A	B	C	∧	D	E	∧	∧	∧	F	∧	∧	G

(c) 改造后完全二叉树顺序存储状态

图 5-13　一般二叉树及其顺序存储示意图



（a）一棵右单支二叉树　　（b）改造后的右单支数对应的完全二叉树

A	∧	B	∧	∧	∧	C	∧	∧	∧	∧	∧	∧	∧	D

（c）单支数改造后完全二叉树的顺序存储状态

图 5-14　右单支二叉树及其顺序存储示意图

二叉树的顺序存储表示可描述为：

```
  #define MAXNODE …                    // 二叉树的最大节点数
typedef DataType SqBiTree[MAXNODE] //0 号单元存放根节点
SqBiTree bt;
```

即将 bt 定义为含有 MAXNODE 个 DataType 类型元素的一维数组。

2. 链式存储结构

二叉树的链式存储结构是指，用链表来表示一棵二叉树，即用链来指示元素的逻辑关系。通常有下面两种形式。

（1）二叉链表存储。链表中每个节点由三个域组成，除了数据域外，还有两个指针域，分别用来给出该节点左孩子和右孩子所在的链节点的存储地址。节点的存储的结构如图 5-15 所示。

lchild	data	rchild

图 5-15　二叉链表节点存储结构图

其中，data 域存放某节点的数据信息；lchild 与 rchild 分别存放指向左孩子和右孩子的指针，当左孩子或右孩子不存在时，相应指针域值为空（用符号 ∧ 或 NULL 表示）。

图 5-16(a) 给出了如图 5-11(b) 所示的一棵二叉树的二叉链表示。

二叉链表也可以带头节点的方式存放，如图 5-16(b) 所示。

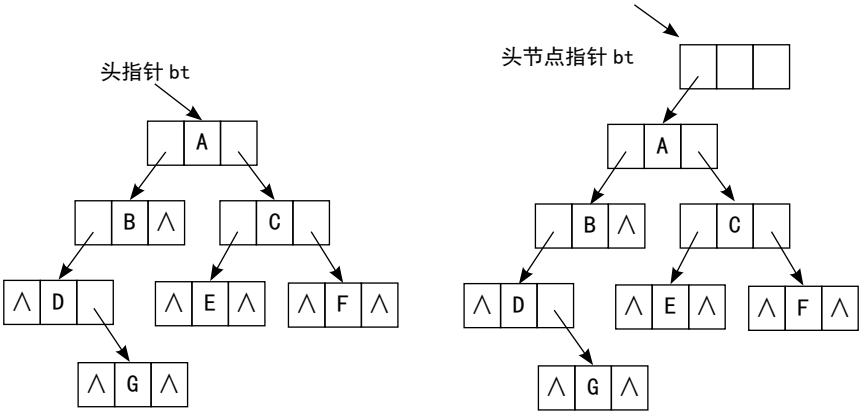


图 5-16　图 5-11(b) 所示二叉树的二叉链表示意图

（2）三叉链表存储。每个节点由四个域组成，具体结构如图 5-17 所示。

lchild	data	rchild	parent

图 5-17　三叉链表节点存储结构图

其中，data、lchild 以及 rchild 三个域的意义同二叉链表结构；parent 域为指向该节点双亲节点的指针。这种存储结构既便于查找孩子节点，又便于查找双亲节点；但是，相对于二叉链表存储结构而言，它增加了存储空间。

图 5-18 给出了如图 5-11(b) 所示的二叉树的三叉链表示。

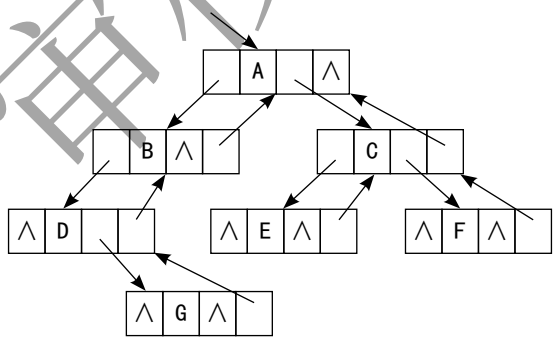


图 5-18　图 5-11(b) 所示二叉树的三叉链表表示示意图

尽管在二叉链表中无法由节点直接找到其双亲，但由于二叉链表结构灵活，操作方便，对于一般情况的二叉树，甚至比顺序存储结构还节省空间。因此，二叉链表是最常用的二叉树存储方式。如不加特殊说明，本书后面所涉及到的二叉树的链式存储结构都是指二叉链表结构。

二叉树的二叉链表存储表示可描述为：

```
typedef struct BiTNode{
        DataType data;
        struct BiTNode *lchild;*rchild;    // 左右孩子指针
    }BiTNode,*BiTree;
```

即将 BiTree 定义为指向二叉链表节点结构的指针类型。

5.3.2 二叉树的基本操作及实现

1. 二叉树的基本操作

（1）Create(data)：根据 data 值建立一棵空二叉树或一个节点。

（2）Create2(data,lbt,rbt)：生成一棵以 data 为根节点的数据域信息，以二叉树 lbt 和 rbt 为左子树和右子树的二叉树。

（3）InsertL(data,parent)：将数据域信息为 data 的节点插入到二叉树中，作为节点 parent 的左孩子节点。如果节点 parent 原来有左孩子节点，则将节点 parent 原来的左孩子节点作为节点 data 的左孩子节点。

（4）InsertR(data,parent)：将数据域信息为 data 的节点插入到二叉树中，作为节点 parent 的右孩子节点。如果节点 parent 原来有右孩子节点，则将节点 parent 原来的右孩子节点作为节点 data 的右孩子节点。

（5）DeleteL(parent)：在二叉树中删除节点 parent 的左子树。

（6）DeleteR(parent)：在二叉树中删除节点 parent 的右子树。

（7）Search(bt,data)：在二叉树中查找数据元素 data。

（8）Traverse(bt)：按某种方式遍历二叉树 bt 的全部节点。

2. 算法的实现

算法的实现依赖于具体的存储结构，当二叉树采用不同的存储结构时，上述各种操作的实现算法是不同的。下面讨论基于二叉链表存储结构的上述操作的实现算法。

（1）Create(bt) 初始建立二叉树 bt，并使 bt 指向头节点。在二叉树根节点前建立头节点，就如同在单链表前建立的头节点，可以方便后边的一些操作实现。建立二叉树的算法如下：

```
// 创建二叉树或节点
BTNode* Create(DataType data)
```

```
{
    BTNode* root = (BTNode*)malloc(sizeof(*root));
    root->data = data;
    root->lchild = NULL;
    root->rchild = NULL;
    return root;
}
```

（2）Create2(data,lbt,rbt) 建立一棵以 data 为根节点的数据域信息，以二叉树 lbt 和 rbt 为左右子树的二叉树。建立成功时返回所建二叉树节点的指针；建立失败时返回空指针。建立一棵已生成左右子树的二叉树的算法如下：

```
BiTree Create2(DataType data, BiTree lbt, BiTree rbt)
{
    BTNode* root = (BTNode*)malloc(sizeof(*root));
    root->data = data;
    root->lchild = lbt;
    root->rchild = rbt;
    return root;
}
```

（3）InsertL(data,parent) 在二叉树中的 parent 所指节点和其左子树之间插入数据元素为 data 的节点。算法如下：

```
int InsertL(DataType data, BiTree parent)
{
    if (parent == NULL)
        return 0;

    BiTree p = Create(data);
    if (parent->lchild == NULL)
        parent->lchild = p;
    else
    {
        p->lchild = parent->lchild;
        parent->lchild = p;
    }
    return 1;
}
```

（4）InsertR(data,parent) 功能类同于（3），算法略。

（5）DeleteL(parent) 在二叉树中删除节点 parent 的左子树。当 parent 或 parent 的左孩子节点为空时删除失败。删除成功时返回根节点指针；删除失败时返回空指针。

```
int DeleteL(BiTree parent)
{
  if (parent == NULL || parent->lchild == NULL)
    return 0;
  BiTree p = parent->lchild;
  parent->lchild = NULL;
  free(p); // 当 p 为非叶子节点时，这样删除仅释放了所删子树根节点的空间，
           // 若要删除子树分支中的节点，需用后面介绍的遍历操作来实现。
  return 1;
}
```

（6）DeleteR(parent) 功能类同于（5），只是删除节点 parent 的右子树。算法略。

操作 Search(bt,data) 实际是遍历操作 Traverse(bt) 的特例，关于二叉树的遍历操作的实现，将在下一节中重点介绍。

5.4 二叉树的遍历

5.4.1 二叉树的遍历方法及递归实现

二叉树的遍历是指按照某种顺序访问二叉树中的每个节点，使每个节点被访问一次且仅被访问一次。

遍历是二叉树中经常要用到的一种操作。因为在实际应用问题中，常常需要按一定顺序对二叉树中的每个节点逐个进行访问，或查找具有某一特点的节点，然后对这些满足条件的节点进行处理。

通过一次完整的遍历，可使二叉树中节点信息由非线性排列变为某种意义上的线性序列。也就是说，遍历操作使非线性结构线性化。

由二叉树的定义可知，一棵二叉树由根节点、根节点的左子树和根节点的右子树三部分组成。因此，只要依次遍历这三部分，就可以遍历整个二叉树。若以 D，L，R 分别表示访问根节点、遍历根节点的左子树、遍历根节点的右子树，

则二叉树的遍历方式有六种：DLR，LDR，LRD，DRL，RDL 和 RLD。如果限定先左后右，则只有前三种方式，即 DLR（称为先序遍历）、LDR（称为中序遍历）和 LRD（称为后序遍历）。

1. 先序遍历（DLR）

先序遍历的递归过程为：若二叉树为空，遍历结束。否则如下：

（1）访问根节点；

（2）先序遍历根节点的左子树；

（3）先序遍历根节点的右子树。

先序遍历二叉树的递归算法如下：

```
void PreOrderTraverse(BiTree bt)
{
   if (bt == NULL)
      return;
   Visit(bt->data);
   PreOrderTraverse(bt->lchild);
   PreOrderTraverse(bt->rchild);
}
```

对于图 5-11(b) 所示的二叉树，按先序遍历所得到的节点序列为 A B D G C E F。

2. 中序遍历（LDR）

中序遍历的递归过程为：若二叉树为空，遍历结束。否则如下：

（1）中序遍历根节点的左子树；

（2）访问根节点；

（3）中序遍历根节点的右子树。

中序遍历二叉树的递归算法如下：

```
void InOrderTraverse(BiTree bt)
{
   if (bt == NULL)
      return;
   InOrderTraverse(bt->lchild);
   Visit(bt->data);
   InOrderTraverse(bt->rchild);
}
```

对于图 5-11(b) 所示的二叉树，按中序遍历所得到的节点序列为 D G B A E C F。

3. 后序遍历（LRD）

后序遍历的递归过程为：若二叉树为空，遍历结束。否则如下：

（1）后序遍历根节点的左子树；

（2）后序遍历根节点的右子树；

（3）访问根节点。

后序遍历二叉树的递归算法如下：

```
void PostOrderTraverse(BiTree bt)
{
   if (bt == NULL)
      return;
   PostOrderTraverse(bt->lchild);
   PostOrderTraverse(bt->rchild);
   Visit(bt->data);
}
```

对于图 5-11(b) 所示的二叉树，按后序遍历所得到的节点序列为 G D B E F C A。

4. 层次遍历

二叉树的层次遍历，是指从二叉树的第一层（根节点）开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对节点逐个访问。对于图 5-11(b) 所示的二叉树，按层次遍历所得到的节点序列为 A B C D E F G。

由层次遍历的定义可以推知，在进行层次遍历时，对一层节点访问完后，再按照它们的访问次序对各个节点的左孩子和右孩子顺序访问，这样一层一层进行，先遇到的节点先访问，这与队列的操作原则比较吻合。因此，在进行层次遍历时，可设置一个队列结构，遍历从二叉树的根节点开始，首先将根节点指针入队列，然后从队头取出一个元素，每取一个元素，执行以下两个操作：

（1）访问该元素所指节点；

（2）若该元素所指节点的左、右孩子节点非空，则将该元素所指节点的左孩子指针和右孩子指针顺序入队。

此过程不断进行，当队列为空时，二叉树的层次遍历结束。

在下面的层次遍历算法中，二叉树以二叉链表存放，一维数组 Queue[MAXNODE] 用以实现队列，变量 front 和 rear 分别表示队头和队尾指

针，记录当前对头元素和队尾元素在数组中的位置。二叉树层次遍历算法如下：

```
void LevelOrderTaverse(BiTree bt)
{
   BiTree queue[MAXNODE];
   int front, rear;
   if (bt == NULL)
      return;
   front = -1;
   rear = 0;
   queue[rear] = bt;
   while (front != rear)
   {
      front++;
      Visit(queue[front]->data);   // 访问队首节点的数据域
      if (queue[front]->lchild != NULL)   // 将队首节点的左孩子节点入队列
      {
         rear++;
         queue[rear] = queue[front]->lchild;
      }
      if (queue[front]->rchild != NULL)   // 将队首节点的右孩子节点入队列
      {
         rear++;
         queue[rear] = queue[front]->rchild;
      }
   }
}
```


5.4.2　二叉树遍历的非递归实现

上述给出的二叉树先序、中序和后序三种遍历算法都是递归算法。当给出二叉树的链式存储结构以后，用具有递归功能的程序设计语言很方便就能实现上述算法。然而，并非所有程序设计语言都允许递归。另外，递归程序虽然简洁，但可读性一般不好，执行效率也不高。因此，就存在如何把一个递归算法转化为非递归算法的问题。解决这个问题的方法可以通过对三种遍历方法的实质过程的分析得到。

对于图 5-11(b) 所示的二叉树，其先序、中序和后序遍历都是从根节点 A 开

始的，且在遍历过程中经过节点的路线也是一样的，只是访问的时机不同而已。图 5-19 中所示的从根节点左外侧开始，由根节点右外侧结束的曲线，为遍历图 5-11(b) 的路线。沿着该路线按△标记的节点读得的序列为先序序列，按 * 标记读得的序列为中序序列，按⊕标记读得的序列为后序序列。

然而，这一路线正是从根节点开始沿左子树深入下去，当深入到最左端，无法再深入下去时，则返回，再逐一进入刚才深入时遇到节点的右子树，再进行如此的深入和返回，直到最后从根节点的右子树返回到根节点为止。先序遍历是在深入时遇到节点就访问，中序遍历是在从左子树返回时遇到节点访问，后序遍历是在从右子树返回时遇到节点访问。

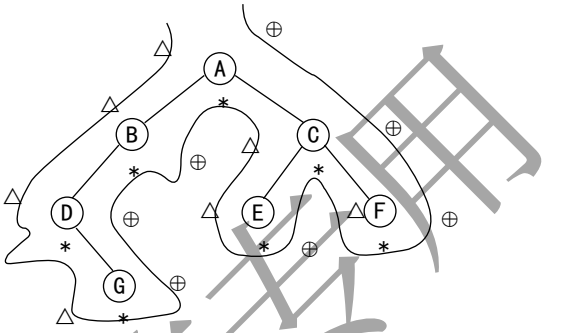


图 5-19　遍历图 5-11(b) 的路线示意图

在这一过程中，返回节点的顺序与深入节点的顺序相反，即后深入先返回，正好符合栈结构后进先出的特点。因此，可以用栈来帮助实现这一遍历路线。

在沿左子树深入时，深入一个节点入栈一个节点，若为先序遍历，则在入栈之前访问之；当沿左分支深入不下去时，则返回，即从堆栈中弹出前面压入的节点，若为中序遍历，则此时访问该节点，然后从该节点的右子树继续深入；若为后序遍历，则将此节点再次入栈，然后从该节点的右子树继续深入，与前面类同，仍为深入一个节点入栈一个节点，深入不下去再返回，直到第二次从栈里弹出该节点，即从右子树返回时，才访问之。

（1）先序遍历的非递归实现。在下面算法中，二叉树以二叉链表存放，一维数组 stack[MAXNODE] 用以实现栈，变量 top 用来表示当前栈顶的位置。先序遍历的非递归算法如下：

```
void NRPreOrderTraverse(BiTree bt)
{
   if (bt == NULL)
      return;
   BiTree stack[MAXNODE];
```

```
    BiTree p = bt;
    int top = -1;
    while (p || top != -1)
    {
        while (p != NULL)
        {
            Visit(p->data);   // 访问节点的数据域
            if (top < MAXNODE - 1)  // 将当前指针 p 压栈
            {
                top++;
                stack[top] = p;
            }
            else
                return; // 栈溢出

            p = p->lchild;        // 指针指向 p 的左孩子节点
        }
        if (top == -1)
            return;    // 栈空时结束
        else
        {
            p = stack[top];          // 从栈中弹出栈顶元素
            top--;
            p = p->rchild;           // 指针指向 p 的右孩子节点
        }
    }
}
```

对于图 5-11(b) 所示的二叉树，用该算法进行遍历过程中，栈 stack 和当前指针 p 的变化情况以及树中各节点的访问次序见表 5-1。

表 5-1　二叉树先序非递归遍历过程

步　骤	指针 p	栈 stack 内容	访问节点值
初态	A	空	
1	B	A	A
2	D	A，B	B
3	∧	A，B，D	D

续表

步　骤	指针 p	栈 stack 内容	访问节点值
4	G	A，B	
5	∧	A，B，G	G
6	∧	A，B	
7	∧	A	
8	C	空	
9	E	C	C
10	∧	C，E	E
11	∧	C	
12	F	空	
13	∧	F	F
14	∧	空	

（2）中序遍历的非递归实现。中序遍历的非递归算法的实现，只需将先序遍历的非递归算法中的 Visit(p->data) 移到 p=stack[top] 和 p=p->rchild 之间即可。

（3）后序遍历的非递归实现。由前面的讨论可知，后序遍历与先序遍历和中序遍历不同，在后序遍历过程中，节点在第一次出栈后，还需再次入栈，也就是说，节点要入两次栈，出两次栈，而访问节点是在第二次出栈时访问。因此，为了区别同一个节点指针的两次出栈，设置一标志 flag，令

$$\text{flag}=\begin{cases}1, & \text{第一次出栈，节点不可以访问}\\ 2, & \text{第二次出栈，节点可以访问}\end{cases} \qquad (5\text{-}10)$$

当节点指针进、出栈时，其标志 flag 也同时进、出栈。因此，可将栈中元素的数据类型定义为指针和标志 flag 合并的结构体类型。定义如下：

```
typedef struct{
    BiTree link;
    int flag;
  }StackType;
```

在算法中，一维数组 stack[MAXNODE] 用于实现栈的结构，指针变量 p 指向当前要处理的节点，整型变量 top 为栈顶指针，用来表示当前栈顶的位置，整型变量 sign 为节点 p 的标志量。后序遍历的非递归算法如下：

```
void NRPostOrderTraverse(BiTree bt)
{
   if (bt == NULL)
      return;
   StackType stack[MAXNODE];
   BiTree p = bt;
   int sign, top = -1;          // 栈顶位置初始化
   while (p || top != -1)
   {
      if (p != NULL)          // 节点第一次进栈
      {
         top++;
         stack[top].link = p;
         stack[top].flag = 1;
         p = p->lchild;       // 找该节点的左孩子
      }
      else
      {
         p = stack[top].link;
         sign = stack[top].flag;
         top--;
         if (sign == 1)          // 节点第二次进栈
         {
            top++;
            stack[top].link = p;
            stack[top].flag = 2;      // 标记第二次出栈
            p = p->rchild;
         }
         else
         {
            Visit(p->data);     // 访问该节点数据域值
            p = NULL;
         }
      }
   }
}
```

5.4.3 不用栈的二叉树遍历的非递归方法

上述介绍的二叉树的遍历算法可分为两类：一类是依据二叉树结构的递归性，采用递归调用的方式来实现；另一类非递归处理则是通过堆栈或队列（层次）来辅助实现。采用这两类方法对二叉树进行遍历时，栈或队列的使用都会带来额外空间的增加，递归调用的深度和栈的大小是动态变化的，都与二叉树的高度有关。因此，在最坏的情况下，即二叉树退化为单支树的情况下，递归的深度或栈需要的存储空间等于二叉树中的节点数。

还有一类二叉树的遍历算法，就是不用栈也不用递归来实现。常用的不用栈的二叉树遍历的非递归方法有以下三种：

（1）对二叉树采用三叉链表存放，即在二叉树的每个节点中增加一个双亲域 parent，这样，在遍历深入到不能再深入时，可沿着走过的路径回退到任何一棵子树的根节点，并再向另一方向走。由于这一方法的实现是在每个节点的存储上又增加一个双亲域，故其存储开销就会增加。

（2）用逆转链的方法，即在遍历深入时，每深入一层，就将其再深入的孩子节点的地址取出，并将其双亲节点的地址存入，当深入不下去需返回时，可逐级取出双亲节点的地址，沿原路返回。虽然此种方法是在二叉链表上实现的，没有增加过多的存储空间，但在执行遍历的过程中改变子女指针的值，这既是以时间换取空间，同时当有几个用户同时使用这个算法时将会发生问题。

（3）在线索二叉树上的遍历，即利用具有 n 个节点的二叉树中的叶子节点和一度节点的 $n+1$ 个空指针域，来存放线索，然后在这种具有线索的二叉树上遍历时，就可不需要栈，也不需要递归了。有关线索二叉树的详细内容，将在下一节中讨论。

5.5 线索二叉树

5.5.1 线索二叉树的定义及结构

1. 线索二叉树的定义

按照某种遍历方式对二叉树进行遍历，可以把二叉树中所有节点排列为一个线性序列。在该序列中，除第一个节点外，每个节点有且仅有一个直接前驱节点；

除最后一个节点外，每个节点有且仅有一个直接后继节点。但是，二叉树中每个节点在这个序列中的直接前驱节点和直接后继节点是什么，在二叉树的存储结构中并没有反映出来，只能在对二叉树遍历的动态过程中得到这些信息。为了保留节点在某种遍历序列中直接前驱和直接后继的位置信息，可以利用二叉树的二叉链表存储结构中的那些空指针域来指示。这些指向直接前驱节点和指向直接后继节点的指针被称为线索（thread），加了线索的二叉树称为线索二叉树。

线索二叉树将为二叉树的遍历提供许多方便。

2. 线索二叉树的结构

一个具有 n 个节点的二叉树若采用二叉链表存储结构，在 $2n$ 个指针域中只有 $n-1$ 个指针域是用来存储节点孩子的地址，而另外 $n+1$ 个指针域存放的都是 NULL。因此，可以利用某节点空的左指针域（lchild）指出该节点在某种遍历序列中的直接前驱节点的存储地址，利用节点空的右指针域（rchild）指出该节点在某种遍历序列中的直接后继节点的存储地址；对于那些非空的指针域，则仍然存放指向该节点左、右孩子的指针。这样，就得到了一棵线索二叉树。

由于序列可由不同的遍历方法得到，因此，线索树有先序线索二叉树、中序线索二叉树和后序线索二叉树三种。把二叉树改造成线索二叉树的过程称为线索化。

对图 5-11(b) 所示的二叉树进行线索化，得到先序线索二叉树、中序线索二叉树和后序线索二叉树分别如图 5-21(a)(b)(c) 所示。图中实线表示指针，虚线表示线索。

那么，在存储中，如何区别某节点的指针域内存放的是指针还是线索？通常可以采用下面两种方法来实现。

（1）为每个节点增设两个标志位域 ltag 和 rtag，令

$$\text{ltag}=\begin{cases}0, & \text{lchild 指向节点的左孩子}\\ 1, & \text{lchild 指向节点的前驱节点}\end{cases} \tag{5-11}$$

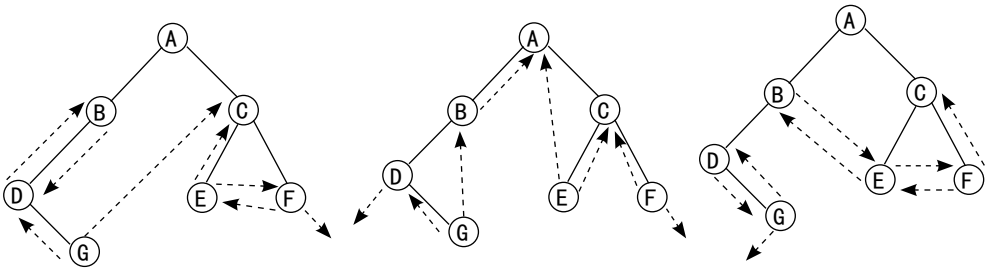
$$\text{rtag}=\begin{cases}0, & \text{rchild 指向节点的右孩子}\\ 1, & \text{rchild 指向节点的后继节点}\end{cases} \tag{5-12}$$

每个标志位令其只占一个 bit，这样就只需增加很少的存储空间。节点的结构如图 5-20 所示。

ltag	lchild	data	rchild	rtag

图 5-20　节点结构图

（2）不改变节点结构，仅在作为线索的地址前加一个负号，即负的地址表示线索，正的地址表示指针。



（a）先序线索二叉树　　（b）中序线索二叉树　　（c）后序线索二叉树

图 5-21　线索二叉树

这里我们按第一种方法来介绍线索二叉树的存储。为了将二叉树中所有空指针域都利用上，以及操作便利的需要，在存储线索二叉树时往往增设一头节点，其结构与其他线索二叉树的节点结构一样，只是其数据域不存放信息，其左指针域指向二叉树的根节点，右指针域指向自己。而原二叉树在某序遍历下的第一个节点的前驱线索和最后一个节点的后继线索都指向该头节点，如图 5-22 所示。

图 5.22 给出了图 5.21（b）所示的中序线索二叉树的存储结构。

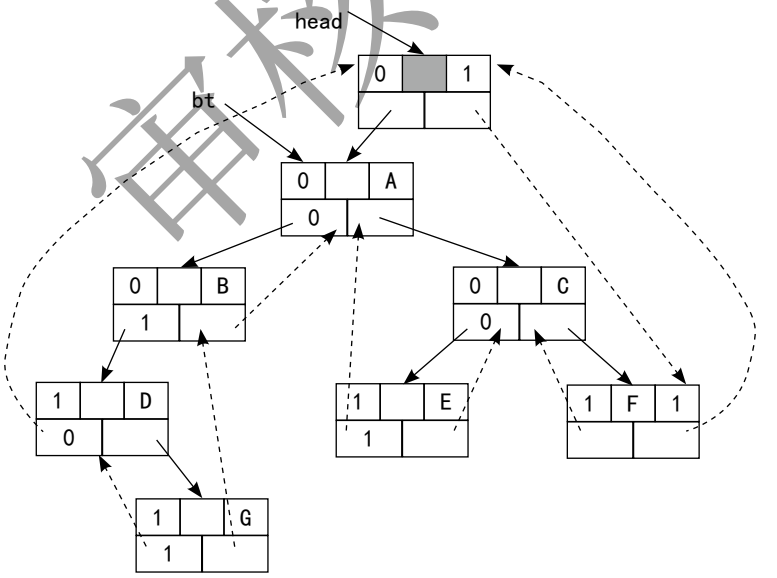


图 5-22　中序线索二叉树的存储示意

5.6　二叉树的应用

5.6.1　二叉树遍历的应用

在以上讨论的遍历算法中，访问节点的数据域信息，即操作 Visit(bt->data) 具有更一般的意义，需根据具体问题，对 bt 数据进行不同的操作。现在介绍几个遍历操作的典型应用。

1. 查找数据元素

Search(bt，x) 在 bt 为根节点指针的二叉树中查找数据元素 x。查找成功时返回该节点的指针；查找失败时返回空指针。

算法实现如下，注意遍历算法中的 Visit(bt->data) 等同于其中的一组操作步骤。

在 bt 二叉树中查找数据元素 data 的算法如下：

```
BiTree Search(BiTree bt, DataType data)
{
   BiTree p;
   if (bt)
   {
      if (bt->data == data)
         return bt;                    // 查找成功返回
      if (bt->lchild != NULL) // 在 bt->lchild 为根节点指针的二叉树中查找数据元素 data
      {
         p = Search(bt->lchild, data);
         if (p)
            return p;
      }
      if (bt->rchild != NULL) // 在 bt->rchild 为根节点指针的二叉树中查找数据元素 data
      {
         p = Search(bt->rchild, data);
         if (p)
            return p;
```

```
        }
    }
    return NULL;              // 查找失败返回
}
```

2. 统计给定二叉树中叶子节点的数目

（1）顺序存储结构的实现。统计给定二叉树中叶子节点的数目算法如下：

```
int CountLeaf1(SqDataType* bt, int index)
{// 一维数组 bt 为二叉树存储结构，index 为数组下标，函数值为叶子数。
    if (bt[index] == 0) //bt[i]=0 表示虚节点
        return 0;
    return CountLeaf1(bt, index * 2 + 1) + CountLeaf1(bt, index * 2 + 2);
}
```

（2）二叉链表存储结构的实现。统计给定二叉树中叶子节点的数目算法如下：

```
int CountLeaf2(BiTree bt) {
    // 开始时，bt 为根节点所在链节点的指针，返回值为 bt 的叶子数
    if (bt == NULL)
        return 0;
    if (bt->lchild == NULL && bt->rchild == NULL)
        return 1;
    return (CountLeaf2(bt->lchild) + CountLeaf2(bt->rchild));
}
```

3. 表达式运算

我们可以把任意一个算术表达式用一棵二叉树表示，图 5-23 所示为表达式 3*2+x-1/x+5 的二叉树表示。在表达式二叉树中，每个叶子节点都是操作数，每个非叶子节点都是运算符。对于一个非叶子节点，它的左、右子树分别是它的两个操作数。

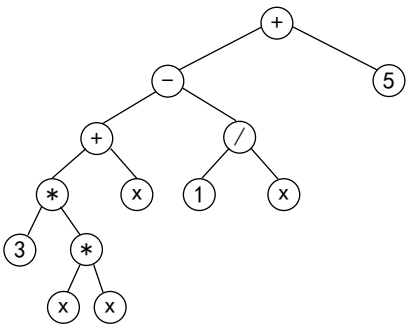


图 5-23　表达式 3*2+x-1/x+5 的二叉树表示

对该二叉树分别进行先序、中序和后序遍历，可以得到表达式的三种不同表示形式：

前缀表达式　+-+*3*xxx/1x5

中缀表达式　3*x*x+x-1/x+5

后缀表达式　3xx**x+1x/-5+

中缀表达式是经常使用的算术表达式，前缀表达式和后缀表达式分别称为波兰式和逆波兰式，它们在编译程序中有着非常重要的作用。

5.5.2　最优二叉树 —— 哈夫曼树

1. 问题引入

先来看一个例子。

【例 5-2】要编制一个将百分制转换为五级分制的程序。

显然，此程序很简单，利用条件语句便可完成。如：

```
if (a<50) b="bad";
 else if (a<70) b="pass"
   else if (a<80) b="general"
     else if (a<90) b="good"
       else b="excellent";
```

这个判定过程可用图 5-25(a) 所示的判定树来表示。如果上述程序需反复使用，而且每次的输入量很大，则应考虑上述程序的质量问题，即其操作所需要的时间。因为在实际中，学生的成绩在 5 个等级上的分布是不均匀的，假设其分布规律见表 5-2。

表 5-2　学生成绩等级分布规律表

分数	0~59	60~69	70~79	80~89	90~100
比例数	0.05	0.15	0.40	0.30	0.10

由表 5-2 可见：则 80% 以上的数据需进行三次或三次以上的比较才能得出结果。

若按图 5-24(b) 所示的判定过程进行判定，可使大部分的数据经过较少的比较次数就能得出结果。但由于每个判定框都有两次比较，将这两次比较分开，得到如图 5-24(c) 所示的判定树，按此判定树可写出相应的程序。

假设有 10 000 个输入数据，若按图 5-24(a) 的判定过程进行操作，则总共需进行 31 500 次比较；而若按图 5-24(c) 的判定过程进行操作，则总共仅需进行

22 000 次比较。

由此可见，同一个问题，采用不同的判定树来解决，效率是不一样的。我们希望出现概率高的结果能够更快地被搜索到，这样就提出了一个问题：以怎样的顺序搜索效率最高？这就是最优树要解决的问题。

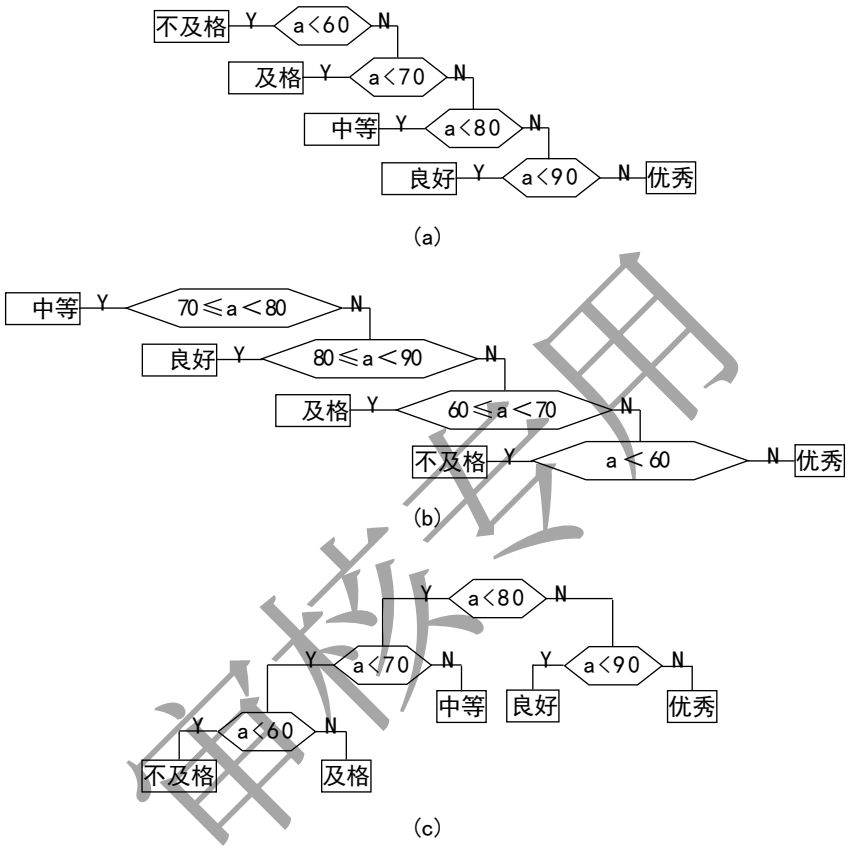


图 5-24　转换五级分制的判定过程

2. 哈夫曼树的基本概念及其构造方法

最优二叉树，也称哈夫曼（Haffman）树，是指对于一组带有确定权值的叶子节点，构造的具有最小带权路径长度的二叉树。

那么什么是二叉树的带权路径长度呢？

在前面我们介绍过路径和节点的路径长度的概念，而二叉树的路径长度则是指由根节点到所有叶子节点的路径长度之和。如果二叉树中的叶子节点都具有一定的权值，则可将这一概念加以推广。设二叉树具有 n 个带权值的叶节点，那么从根节点到各个叶子节点的路径长度与相应节点权值的乘积之和叫作二叉树的

带权路径长度，记为

$$\mathrm{WPL}=\sum_{k=1}^{n}W_kL_k \qquad (5\text{-}13)$$

式中，W_k 为第 k 个叶子节点的权值；L_k 为第 k 个叶子节点的路径长度。

如图 5-25 所示的二叉树，它的带权路径长度值 WPL=2×2 + 4×2 + 5×2 + 3×2=28。

在给定一组具有确定权值的叶子节点，可以构造出不同的带权二叉树。例如，给出 4 个叶节点，设其权值分别为 1，3，5，7，我们可以构造出形状不同的多个二叉树。这些形状不同的二叉树的带权路径长度将各不相同。图 5-26 给出了其中 5 个不同形状的二叉树。

这五棵树的带权路径长度分别为

图 (a)WPL=1×2 + 3×2 + 5×2 + 7×2=32

图 (b)WPL=1×3 + 3×3 + 5×2 + 7×1=29

图 (c)WPL=1×2 + 3×3 + 5×3 + 7×1=33

图 (d)WPL=7×3 + 5×3 + 3×2 + 1×1=43

图 (e)WPL=7×1 + 5×2 + 3×3 + 1×3=29

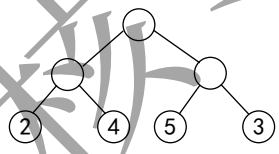


图 5-25　一个带权二叉树

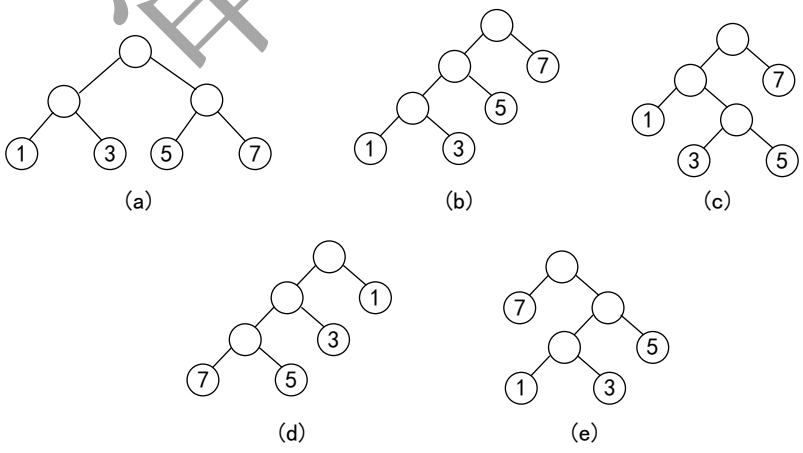


图 5-26　具有相同叶子节点和不同带权路径长度的二叉树

由此可见，由相同权值的一组叶子节点所构成的二叉树有不同的形态和不同的带权路径长度，那么如何找到带权路径长度最小的二叉树（即哈夫曼树）呢？根据哈夫曼树的定义，一棵二叉树要使其 WPL 值最小，必须使权值越大的叶子节点越靠近根节点，而权值越小的叶子节点越远离根节点。哈夫曼（Haffman）依据这一特点提出了一种方法，这种方法的基本思想如下：

（1）由给定的 n 个权值 $\{W_1, W_2, \cdots, W_n\}$ 构造 n 棵只有一个叶子节点的二叉树，从而得到一个二叉树的集合 $F=\{T_1, T_2, \cdots, T_n\}$。

（2）在 F 中选取根节点的权值最小和次小的两棵二叉树作为左、右子树构造一棵新的二叉树，这棵新的二叉树根节点的权值为其左、右子树根节点权值之和。

（3）在集合 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到集合 F 中。

（4）重复(2)(3)两步，当 F 中只剩下一棵二叉树时，这棵二叉树便是所要建立的哈夫曼树。

图 5-27 给出了前面提到的叶子节点权值集合为 $W=\{1, 3, 5, 7\}$ 的哈夫曼树的构造过程。可以计算出其带权路径长度为 29，由此可见，对于同一组给定叶子节点所构造的哈夫曼树，树的形状可能不同，但带权路径长度值是相同的，一定是最小的。

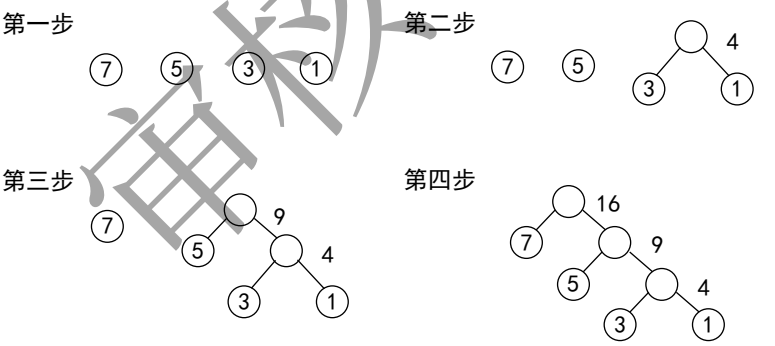


图 5-27　哈夫曼树的建立过程

3. 哈夫曼树的构造算法

为了方便操作，用静态链表作为哈夫曼树的存储。

在构造哈夫曼树时，设置一个结构数组 HuffNode 保存哈夫曼树中各节点的信息，根据二叉树的性质可知，具有 n 个叶子节点的哈夫曼树共有 $2n-1$ 个节点，所以数组 HuffNode 的大小设置为 $2n-1$，节点的结构形式如图 5-28 所示。

图 5-28　哈夫曼树节点结构

其中，weight 域保存节点的权值，lchild 和 rchild 域分别保存该节点的左、右孩子节点在数组 HuffNode 中的序号，从而建立起节点之间的关系。为了判定一个节点是否已加入到要建立的哈夫曼树中，可通过 parent 域的值来确定。初始时 parent 的值为 -1，当节点加入到树中时，该节点 parent 的值为其双亲节点在数组 HuffNode 中的序号，就不会是 -1 了。

构造哈夫曼树时，首先将由 n 个字符形成的 n 个叶子节点存放到数组 HuffNode 的前 n 个分量中，然后根据前面介绍的哈夫曼方法的基本思想，不断将两个小子树合并为一个较大的子树，每次构成的新子树的根节点顺序放到 HuffNode 数组中的前 n 个分量的后面。

下面给出哈夫曼树的构造算法。

```
HNodeType* HaffmanTree(int n)
{
   int min1, min2, index1, index2;
    HNodeType* HuffNode = (HNodeType*)malloc((2 * n - 1) *
sizeof(HNodeType));
   for (int i = 0;i < 2 * n - 1;i++)      // 初始化
   {
      HuffNode[i].weight = 0;
      HuffNode[i].parent = -1;
      HuffNode[i].lchild = -1;
      HuffNode[i].rchild = -1;
   }
   printf(" 请输入叶子节点的权值：\n");
   for (int i = 0;i < n;i++)
      scanf("%d", &HuffNode[i].weight); // 输入每个叶子节点的权值
   for (int i = 0;i < n - 1;i++)
   {
      min1 = min2 = MAXVALUE;
      index1 = index2 = 0;
      for (int j = 0;j < n + i;j++)
      {
         if (HuffNode[j].parent == -1 && HuffNode[j].weight < min1)
         {
```

```
                min2 = min1;
                index2 = index1;
                min1 = HuffNode[j].weight;
                index1 = j;
            }
            else
                if (HuffNode[j].parent == -1 && HuffNode[j].weight < min2)
                {
                    min2 = HuffNode[j].weight;
                    index2 = j;
                }
        }
        // 将找出的两棵子树合并为一棵子树
        HuffNode[index1].parent = n + i;
        HuffNode[index2].parent = n + i;
          HuffNode[n + i].weight = HuffNode[index1].weight +
HuffNode[index2].weight;
        HuffNode[n + i].lchild = index1;
        HuffNode[n + i].rchild = index2;
    }
    return HuffNode;
}
```

如图 5-27 所示的哈夫曼数最后的 HuffNode 如图 5-29 所示。

	weight	lchild	rchild	parent
0	7	-1	-1	6
1	5	-1	-1	5
2	3	-1	-1	4
3	1	-1	-1	4
4	4	3	2	5
5	9	4	2	6
6	16	0	5	-1

图 5-29　对图 5-27 所示的哈夫曼树最后的 HuffNode

4. 哈夫曼树在编码问题中的应用

在数据通讯中，经常需要将传送的文字转换成由二进制字符 0，1 组成的二

进制串，我们称之为编码。例如，假设要传送的电文为 ABACCDA，电文中只含有 A，B，C，D 四种字符，若这四种字符采用表 5-3(a) 所示的编码，则电文的代码为 000010000100100111 000，长度为 21。在传送电文时，我们总是希望传送时间尽可能短，这就要求电文代码尽可能短，显然，这种编码方案产生的电文代码不够短。表 5-3(b) 为另一种编码方案，用此编码对上述电文进行编码所建立的代码为 00010010101100，长度为 14。在这种编码方案中，四种字符的编码均为两位，是一种等长编码。如果在编码时考虑字符出现的频率，让出现频率高的字符采用尽可能短的编码，出现频率低的字符采用稍长的编码，构造一种不等长编码，则电文的代码就可能更短。如当字符 A，B，C，D 采用表 5-3(c) 的编码时，上述电文的代码为 0110010101110，长度仅为 13。

表 5-3　字符的 4 种不同的编码方案

字符	编码
A	000
B	010
C	100
D	111

(a)

字符	编码
A	00
B	01
C	10
D	11

(b)

字符	编码
A	0
B	110
C	10
D	111

(c)

字符	编码
A	01
B	010
C	001
D	10

(d)

哈夫曼树可用于构造使电文的编码总长最短的编码方案。具体做法如下：设需要编码的字符集合为 $\{d_1, d_2, \cdots, d_n\}$，它们在电文中出现的次数或频率集合为 $\{w_1, w_2, \cdots, w_n\}$，以 $d_1, d_2, \cdots, d_n$ 作为叶子节点，$w_1, w_2, \cdots, w_n$ 作为它们的权值，构造一棵哈夫曼树，规定哈夫曼树中的左分支代表 0，右分支代表 1，则从根节点到每个叶子节点所经过的路径分支组成的 0 和 1 的序列便为该节点对应字符的编码，我们称之为哈夫曼编码。

例如，对图 5-27 所得到的哈夫曼树进行编码的过程如图 5-30 所示。其中权值为 7 的字符编码为 0，权值为 5 的字符编码为 10，权值为 3 的字符编码为 110，权值为 1 的字符编码为 111。可看出权值越大编码长度越短，权值越小编码长度越长。

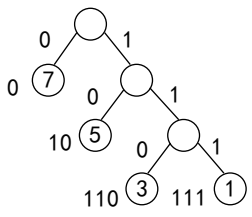


图 5-30　对图 5-27 所得哈夫曼树的叶节点进行编码

在哈夫曼编码树中，树的带权路径长度的含义是各个字符的码长与其出现次数或频度的乘积之和，也就是电文的代码总长或平均码长，所以采用哈夫曼树构造的编码是一种能使电文代码总长最短的不等长编码。

在建立不等长编码时，必须使任何一个字符的编码都不是另一个字符编码的前缀，这样才能保证译码的唯一性。例如表 5-3(d) 的编码方案，字符 A 的编码 01 是字符 B 的编码 010 的前缀部分，这样对于代码串 0101001，既是 AAC 的代码，也是 ABD 和 BDA 的代码，因此，这样的编码不能保证译码的唯一性，我们称之为具有二义性的译码。

然而，采用哈夫曼树进行编码，则不会产生上述二义性问题。因为，在哈夫曼树中，每个字符节点都是叶节点，它们不可能在根节点到其他字符节点的路径上，所以一个字符的哈夫曼编码不可能是另一个字符的哈夫曼编码的前缀，从而保证了译码的非二义性。

现在讨论实现哈夫曼编码的算法。实现哈夫曼编码的算法可分为两大部分：

（1）构造哈夫曼树。

（2）在哈夫曼树上求叶节点的编码。

求哈夫曼编码，实质上就是在已建立的哈夫曼树中，从叶子节点开始，沿节点的双亲链域回退到根节点，每回退一步，就走过了哈夫曼树的一个分支，从而得到一位哈夫曼码值，由于一个字符的哈夫曼编码是从根节点到相应叶子节点所经过的路径上各分支所组成的 0，1 序列，因此先得到的分支代码为所求编码的低位码，后得到的分支代码为所求编码的高位码。我们可以设置一结构数组 HuffCode 用来存放各字符的哈夫曼编码信息，数组元素的结构如图 5-31 所示。

图 5-31　数组元素结构图

其中，分量 bit 为一维数组，用来保存字符的哈夫曼编码，start 表示该编码在数组 bit 中的开始位置。所以，对于第 *i* 个字符，它的哈夫曼编码存放在 HuffCode[i].bit 中的从 HuffCode[i].start 到 *n* 的分量上。

哈夫曼编码算法描述如下。

```
#define MAXBIT 10              // 定义哈夫曼编码的最大长度
typedef struct {
   int bit[MAXBIT];
   int start;
}HCodeType;
void HaffmanCode()
```

```
{
    HCodeType HuffCode[MAXLEAF], cd;
    int i, j, c, p;
    int n = MAXNODE;
    HNodeType* HuffNode = HaffmanTree();       // 建立哈夫曼树
    for (i = 0;i < n;i++)                      // 求每个叶子节点的哈夫曼编码
    {
        cd.start = n - 1;
        c = i;
        p = HuffNode[c].parent;
        while (p != -1)                        // 由叶子节点向上直到树根
        {
            if (HuffNode[p].lchild == c)
                cd.bit[cd.start] = 0;
            else
                cd.bit[cd.start] = 1;
            cd.start--;
            c = p;
            p = HuffNode[c].parent;
        }
        for (j = cd.start + 1;j < n;j++)
            // 保存求出的每个叶节点的哈夫曼编码和编码的起始位
            HuffCode[i].bit[j] = cd.bit[j];
        HuffCode[i].start = cd.start;
    }
    for (i = 0;i < n;i++)                      // 输出每个叶子节点的哈夫曼编码
    {
        for (j = HuffCode[i].start + 1;j < n;j++)
            scanf("%d", &HuffCode[i].bit[j]);
    }
}#define MAXBIT 10                          // 定义哈夫曼编码的最大长度
typedef struct {
    int bit[MAXBIT];
    int start;
}HCodeType;
void HaffmanCoding(int n)
```

```
    {
        HNodeType* HuffNode = HaffmanTree(n);
        HCodeType* HuffCode = (HCodeType*)malloc(n *
sizeof(HCodeType));
        HCodeType code;
        int child, parent;
        for (int i = 0;i < n;i++)              // 求每个叶子节点的哈夫曼编码
        {
            code.start = MAXBIT - 1;
            child = i;
            parent = HuffNode[child].parent;
            while (parent != -1)               // 由叶子节点向上直到树根
            {
                if (HuffNode[parent].lchild == child)
                    code.bit[code.start] = 0;
                else
                    code.bit[code.start] = 1;
                code.start--;
                child = parent;
                parent = HuffNode[child].parent;
            }
            for (int j = code.start + 1;j < MAXBIT;j++)
                // 保存求出的每个叶节点的哈夫曼编码和编码的起始位
                HuffCode[i].bit[j] = code.bit[j];
            HuffCode[i].start = code.start;
        }

        printf(" 每个叶子节点的哈夫曼编码：\n");
        for (int i = 0;i < n;i++)
        {
            printf("%d\t", HuffNode[i].weight);
            for (int j = HuffCode[i].start + 1;j < MAXBIT;j++)
                printf("%d", HuffCode[i].bit[j]);
            printf("\n");
        }
        free(HuffNode);
```

```
    free(HuffCode);
}

void main()
{
    printf(" 请输入叶子节点个数：\n");
    int n;
    scanf("%d", &n);
    HaffmanCoding(n);
}
```

5.7 项目训练

项目：成绩转换

【题目要求】

使用哈夫曼树将百分制的成绩表转换为 ABCDE 五个等级。成绩 90 分以上为 A，80~89 分为 B，70~79 分为 C，60~69 分为 D，60 分以下为 E。

【算法分析】

利用判断语句依次判断分数等级，输出对应等级。难点在考察成绩分布规律，制定最优判断方案。

本章小结

（1）树和二叉树属于非线性结构。树的特点是：除了根节点和叶子节点，其它的节点都只有一个直接前驱和一个或多个直接后继。对于根节点则只有后继节点，而叶子节点则只有前驱节点。树的存储结构分为顺序存储和链式存储两大类。树的一个重要操作是对树进行遍历。

（2）二叉树有两种特殊形态：满二叉树和完全二叉树，有五个重要性质。二叉树的存储结构也分为顺序存储和链式存储两大类。对二叉树进行遍历是二叉树的一个重要操作，二叉树有许多操作是在遍历的基础上进行的，如求二叉树的节点数和二叉树的深度等。二叉树的遍历分为先序、中序、后序和层序遍历。将

二叉树链表中的空指针指向其某种遍历序列中的前驱节点或后继节点，就实现了二叉树的线索化。若需要经常在二叉树中查找节点在某种遍历序列中的前驱或后继节点，就可以采用线索链表作为二叉树的存储结构。

（3）哈夫曼树是最优二叉树，利用哈夫曼树求出的哈夫曼编码在通信领域有着广泛的应用。

习　题

1. 对于如图 5-32 所示二叉树，试给出：

（1）它的顺序存储结构示意图；

（2）它的二叉链表存储结构示意图；

（3）它的三叉链表存储结构示意。

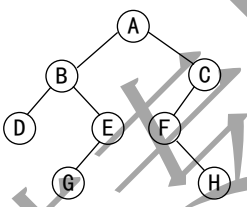


图 5-32　习题 1 图

2. 证明：在节点数多于 1 的哈夫曼树中不存在度为 1 的节点。

3. 证明：若哈夫曼树中有 n 个叶节点，则树中共有 $2n-1$ 个节点。

4. 证明：由二叉树的前序序列和中序序列可以唯一地确定一棵二叉树。

5. 已知一棵度为 m 的树中有 n_1 个度为 1 的节点，n_2 个度为 2 的节点，……，nm 个度为 m 的节点，问该树中共有多少个叶子节点？有多少个非终端节点？

6. 设高度为 h 的二叉树上只有度为 0 和度为 2 的节点，问该二叉树的节点数可能达到的最大值和最小值。

7. 求表达式（$a+b*(c-d))-e/f$ 的波兰式（前缀式）和逆波兰式（后缀式）。

8. 画出和下列已知序列对应的二叉树：

（1）二叉树的先序次序访问序列为：GFKDAIEBCHJ。

（2）二叉树的中序访问次序为：DIAEKFCJHBG。

9. 画出和下列已知序列对应的二叉树：

（1）二叉树的后序次序访问序列为：CFEGDBJLKIHA。

（2）二叉树的中序访问次序为：CBEFDGAJIKLH。

10. 画出和下列已知序列对应的二叉树：

（1）二叉树的层次序列为：ABCDEFGHIJ。

（2）二叉树的中序次序为：DBGEHJACIF。

11. 给定一棵用二叉表示的二叉树，其根指针为 root。试写出求二叉树节点的数目的算法（递归算法或非递归算法）。

12. 请设计一个算法，要求该算法把二叉树的叶节点按从左至右的顺序链成一个单链表。二叉树按 lchild-rchild 方式存储，链接时用叶节点的 rchild 域存放链指针。

13. 给定一棵用链表表示的二叉树，其根节点 root。试写出求二叉树的深度的算法。

14. 给定一棵用链表表示的二叉树，其根指针为 root。试写出求二叉树各节点的层数的算法。

15. 给定一棵用链表表示的二叉树，其根指针为 root。试写出将二叉树中所有节点的左、右子树相互交换的算法。

16. 一棵 n 个节点的完全二叉树以向量作为存储结构，试设计非递归算法对该完全二叉树进行前序遍历。

17. 在二叉树中查找值为 x 的节点，试设计打印值为 x 的节点的所有祖先节点算法。

18. 已知一棵二叉树的后序遍历序列和中序遍历序列，写出可以唯一确定一棵二叉树的算法。

19. 在中序线索二叉树上插入一个节点 p 作为树中某节点 q 的左孩子，试给出实现上述要求的算法。

20. 给出在中序线索二叉树上删除某节点 p 的左孩子节点的算法。

第6章 图

图是一种典型的非线性数据结构，它比线性结构和树形结构更为复杂。在线性结构中，数据元素之间满足一对一的关系；在树型结构中，数据元素之间满足一对多的关系；在图结构中，数据元素之间的关系是任意的，即多对多的关系。因此，图的结构有较强的表达能力，可以用于描述各种复杂的数据对象。图的应用十分广泛，典型的应用领域有电路分析、项目规划、鉴别化合物、统计力学、遗传学、人工智能以及语言等。

知识目标

- 掌握图的基本概念和基本术语。
- 熟练掌握图的存储结构。
- 熟练掌握图的遍历方法和算法。
- 理解图的几种应用的概念和算法。

能力目标

- 能熟练运用图的理论知识，解决实际的问题。

6.1　图的逻辑结构

6.1.1　图的定义和基本术语

1. 图（Graph）的定义

图是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为

$$G=(V, E) \tag{6-1}$$

其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合。

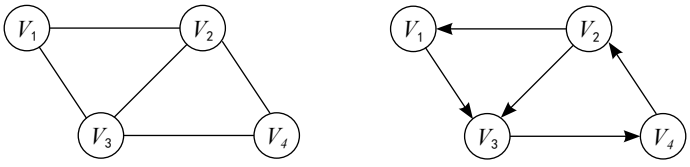
线性表中我们把数据元素叫元素，树中叫节点，而在图中的数据元素我们则称为顶点 (Vertex)。线性表可以没有数据元素，称为空表；树中可以没有节点，叫空树；图中强调顶点集合 V 要有穷且非空，边集合 E 可以为空，我们把顶点非空，边集合为空的图称为零图。

若顶点 V_i 和 V_j 之间的边没有方向，则称这条边为无向边，记（V_i，V_j）。如果图的任意两个顶点之间边都没有方向，则称这个图为无向图（Dndiected graph）。

若顶点 V_i 和 V_j 之间的边有方向，则称这条边为有向边（也称为弧），记 $<V_i$，$V_j>$，V_i 表示弧尾或者起点，V_j 表示弧头或者终点。如果图的任意两个顶点的边都有方向，则称这个图为有向图（Diected graph）。

图的示例如图 6-1 所示。

注意：无向图中（V_i，V_j）与（V_j，V_i）代表同一条边，但有向图中 $<V_i$，$V_j>$ 与 $<V_j$，$V_i>$ 则代表不同的两条弧。



(a) 无向图 G_1　　(b) 有向图 G_2

图 6-1　图的示例

2. 图的基本术语

（1）无向完全图（Undiected complete graph）和有向完全图（Dected complete graph）。在无向图中，任意两个顶点之间都存在边，则称该图为无向完全图，如图 6-2 所示。一个具有 n 个顶点的无向完全图共含有 $n(n-1)/2$ 条边。

在有向图中，任意两个顶点之间存在相反方向的两条弧，则称该图为有向完全图，如图 6-3 所示。一个具有 n 个顶点的有向完全图共含有 $n(n-1)$ 条弧。

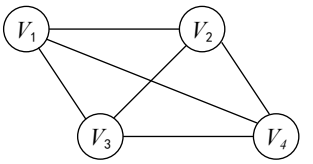
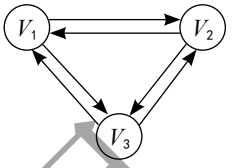


图 6-2　无向完全图



6.3 有向完全图

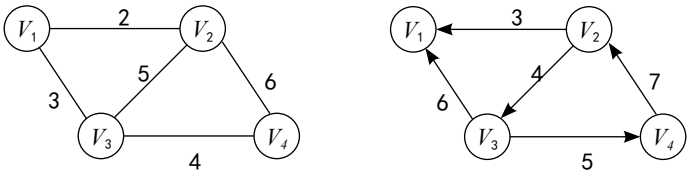
（2）邻接（Adjacent）和依附（Adhere）。无向图中，对于任意两个顶点 V_i 和顶点 V_j，若存在边 (V_i, V_j)，则称顶点 V_i 和顶点 V_j 互为邻接点，同时称边 (V_i, V_j) 依附于顶点 V_i 和顶点 V_j。

有向图中，对于任意两个顶点 V_i 和顶点 V_j，若存在弧 $<V_i, V_j>$，则称顶点 V_i 邻接到顶点 V_j，顶点 V_j 邻接自顶点 V_i，同时称弧 $<V_i, V_j>$ 依附于顶点 V_i 和顶点 V_j。

例如，图 6-1(a) 无向图 G_1 中，V_1 的邻接点有 V_2、V_3；V_2 的邻接点有 V_1、V_3、V_4。图 6-1(b) 无向图 G_2 中，V_1 的邻接点有 V_3；V_2 的邻接点有 V_1、V_3。

（3）权（Weight）和网（Network）。在图中，对图的边或弧附予有意义的数字量，称为权。在实际应用中，权可以表示具体的含义。比如，在电子线路图上，权表示两端点之间的电阻、电流或电压值。

这种边或弧上带有权值的图称为带权图或网。图 6-4 所示即为带权图。



(a) 无向网　　(b) 有向网

图 6-4　带权图或网

（4）度（Dregree）、入度（Indregree）和出度（Outdregree）。在无向图中，顶点 V 的度是指与该顶点相关联的边的数目，记为 D（V）。在有向图中，

顶点 V 的入度是指以该顶点为弧头的弧的数目，记为 ID（V）；顶点 V 的出度是指以该顶点为弧尾的弧的数目，记为 OD（V）；顶点 V 的度定义为 D（V）= ID（V）+ OD（V）。

例如，图 6-1(a) 无向图 G_1 中，顶点 V_2 的度为：D（V_2）=3。图 6-1(b) 有向图 G_2 中，顶点 V_2 的度为：D（V_2）= ID（V_2）+ OD（V_2）=1+2=3。

（5）子图（Subgraph）。对于图 G=（V，E），G’=（V’，E’），若 V’是 V 的子集，E’是 E 的子集，则称图 G’是 G 的子图。图 6-5 所示分别为图 6-1 的一个子图和有向图的一个子图。

(a) 图 G_1 的一个子图　　(b) 图 G_2 的一个子图

图 6-5　子图的例子

（6）路径（Path）、路径长度（Path length）和回路（Circuit）。无向图 G=(V，E) 中从顶点 V 到顶点 V' 的路径是一个顶点序列 V=V_{i0}，V_{i1}，V_{i2}，…，V_{in}=V'。其中 (V_{ij}-1，V_{ij}) ∈ E，$1 \leqslant j \leqslant n$。若 G 是有向图，则路径也是有方向的，顶点序列满足 <V_{ij}-1，V_{ij}> ∈ E。

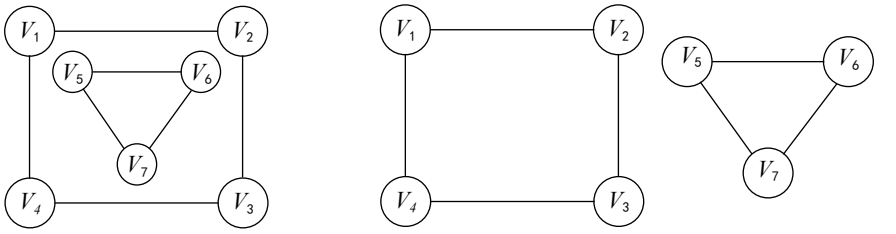
非带权图中，路径长度就是路径上边的个数；带权图中，路径长度即为路径上各边的权值之和。

第一个顶点和最后一个顶点相同的路径称为回路或环。在路径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，其余顶点不重复的回路称为简单回路或简单环。

例如，图 6-1(a) 无向图 G_1 中，其中一条简单路径：$V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_3$，简单回路：$V_1 \rightarrow V_3 \rightarrow V_4 \rightarrow V_2$。

图 6-1(b) 有向图 G_2 中，其中一条简单路径：$V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4$，简单回路：$V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_2$。

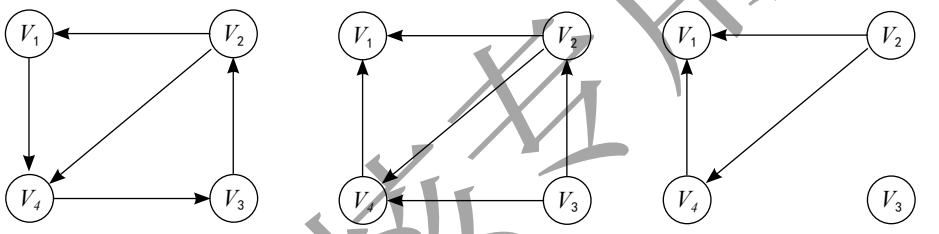
（7）连通图（Connected graph）和连通分量（Connected component）。在无向图中，如果从一个顶点 V_i 到另一个顶点 V_j($i \neq j$) 有路径，则称顶点 V_i 和 V_j 是连通的。如果图中任意两个顶点都是连通的，则称该图是连通图。图 6-1(a) 是连通图。非连通图的极大连通子图称为连通分量。极大的含义是指包含所有连通的顶点以及和这些顶点相关联的所有边。图 6-6(a) 是非连通图，图 6-6(b) 是它的两个连通分量。



（a）非连通图 G_3　　（b）G_3 有两个连通分量

图 6-6　非连通图及连通分量

（8）强连通图（Strongly connected graph）和强连通分量（Strongly connected component）。在有向图中，对图中任意一对顶点 V_i 和 $V_j(i \neq j)$，若从顶点 V_i 到顶点 V_j 和从顶点 V_j 到顶点 V_i 均有路径，则称该有向图是强连通图。非强连通图的极大强连通子图称为强连通分量。图 6-7(a) 是强连通图，图 6-7(b) 是非强连通图，图 6-7(c) 是它的两个强连通分量。



（a）强连通图　　（b）非强连通图 G_4　　（c）G_4 有两个强连通分量

图 6-7　强联通图、非强连通图及强连通分量

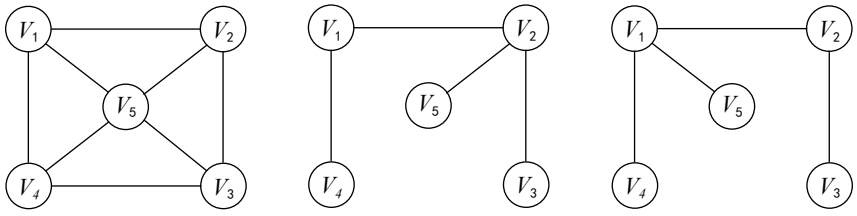
（9）生成树（Spanning tree）和生成森林（Spanning forest）。一个连通图的生成树，是含有该连通图的全部顶点的一个极小连通子图，如图 6-8 所示。

其中，一个连通图 G 的极小连通子图 T 具有以下特征：

1）T 包含 G 的所有 n 个顶点。

2）T 为连通子图。

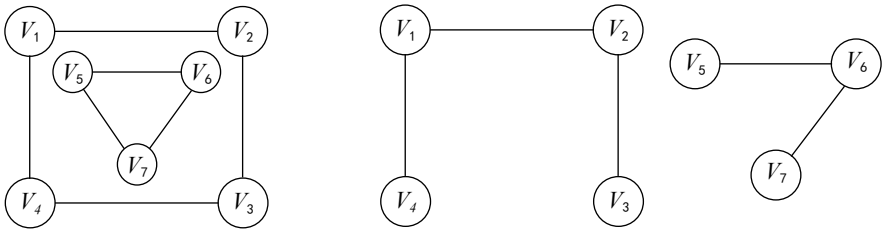
3）T 包含的边数最少。



（a）连通图 G_5　　（b）G_5 的其中两个生成树

图 6-8　连通图及其生成树

在非连通图中，由每个连通分量都可以得到一棵生成树，这些连通分量的生成树就组成了一个非连通图的生成森林，如图 6-9 所示。



(a) 非连通图 G_3　　(b) G_3 的生成树林

图 6-9　非连通图及其生成森林

（10）稀疏图（Sparse graph）和稠密图（Dense graph）。边数很少的图为稀疏图，边数很多的图为稠密图。稀疏图和稠密图常常是相对而言的。

6.1.2　图的抽象数据类型定义

图是一种与具体应用密切相关的数据结构，它的基本操作往往随应用不同而有很大差别，这里只讨论最基本的操作。

图 G 的抽象数据类型定义如下：

ADT　MGraph{

数据对象：$n=n$ 是具有相同特征的数据元素集合，称为顶点集。数据关系：DR={<v，w>|v，$w \in n$ 且 <v，w> 表示从 v 指向 w 的弧 }。

基本操作：

CreateMGraph

初始条件：n 是图的顶点集，e 是图的边集。

操作结果：按和 n 的 e 定义构造图 G。

DestroyMGraph

初始条件：图 G 存在。

操作结果：销毁图 G。

GetVex

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：返回 v 的值。

LocateVex

初始条件：图 G 存在，v 和 G 中顶点有相同特征。

操作结果：若 G 中存在顶点 v，则返回该顶点再图中的位置；否则返回空。

PutVex

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：对 v 赋值 u。

FirstAdjVex

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：返回的第一个邻接顶点。若顶点在 G 中没有邻接顶点，则返回空。

NextAdjVex

初始条件：图 G 存在，v 是 G 中某个顶点，w 是 v 的邻接顶点。

操作结果：返回 v（相对 w）的下一个邻接顶点。若 w 是 v 的最后一个邻接点，则返回空。

InsertVex

初始条件：图 G 存在，v 和图 G 中顶点有相同特征。

操作结果：在图 G 中增添新顶点 v（不增添与顶点相关的边，留待 I*nsert*E*dge*() 去做）。

DeleteVex

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：删除 G 中顶点 v 及其相关的弧。

InsertEdge

初始条件：图 G 存在，v 和 W 是 G 中两个顶点。

操作结果：在 G 中增添弧 <v，w>。

DeleteEdge

初始条件：图 G 存在，v 和 W 是 G 中两个顶点。

操作结果：在 G 中删除弧 <v，w>。

DFSTraverseM

初始条件：图 G 存在。

操作结果：对图进行深度优先遍历。

BFSTraverseM

初始条件：图 G 存在。

操作结果：对图进行广度优先遍历。

}ADT MGraph

6.2　图的存储结构

图的存储结构有多种。图的存储结构的选择取决于具体的应用和需要进行的预算。下面给出常用的三种存储结构：邻接矩阵、邻接表和边集数组。

6.2.1　邻接矩阵

图的邻接矩阵（Adjacency Matrix）分为两部分：顶点 V 和边 E 集合。因此，用一个一维数组存放图中所有顶点数据；用一个二维数组存放顶点间关系（边或弧）的数据，这个二维数组称为邻接矩阵。邻接矩阵又分为无向图邻接矩阵和有向图邻接矩阵。

设 $G=(V, E)$ 是一个图，其中 $V=\{v_1, v_2, \cdots, v_n\}$。$G$ 的邻接矩阵是一个具有下列性质的 n 阶方阵：

$$\text{Edge}[i][j]=\begin{cases}1 \text{ 或 } w_{ij}（G \text{ 为网}），<i, j>\in E \text{ 或 }（i, j）\in E \\ 0 \text{ 或 } \infty（G \text{ 为网}），\text{其他情况}\end{cases} \tag{6-2}$$

如图 6-10 所示为一个无向图及其邻接矩阵存储示意图。

$$\text{Vertex}[4]=\boxed{V_1}\boxed{V_2}\boxed{V_3}\boxed{V_4}$$

$$\text{Edge}=\begin{bmatrix}0 & 1 & 1 & 0\\ 1 & 0 & 1 & 1\\ 1 & 1 & 0 & 1\\ 0 & 1 & 1 & 0\end{bmatrix}$$

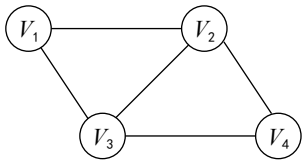


图 6-10　无向图及其邻接矩阵

如图 6-11 所示为一个有向图及其邻接矩阵存储示意图。

Vertex[4]= | V_1 | V_2 | V_3 | V_4 |

$$\text{Edge}=\begin{bmatrix}0&0&1&0\\1&0&1&0\\0&0&0&1\\0&1&0&0\end{bmatrix}$$

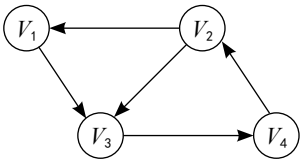


图 6-11 有向图及其邻接矩阵

如图 6-12 所示为一个无向网及其邻接矩阵存储示意图。

Vertex[4]= | V_1 | V_2 | V_3 | V_4 |

$$\text{Edge}=\begin{bmatrix}\infty&2&3&\infty\\2&\infty&5&6\\3&5&\infty&4\\\infty&6&4&\infty\end{bmatrix}$$

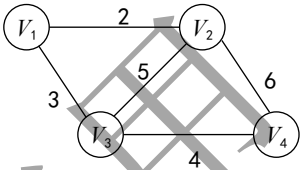


图 6-12 无向网及其邻接矩阵

如图 6-13 所示为一个有向网及其邻接矩阵存储示意图。

Vertex[4]= | V_1 | V_2 | V_3 | V_4 |

$$\text{Edge}=\begin{bmatrix}\infty&\infty&6&\infty\\3&\infty&4&\infty\\\infty&\infty&\infty&5\\\infty&7&\infty&\infty\end{bmatrix}$$

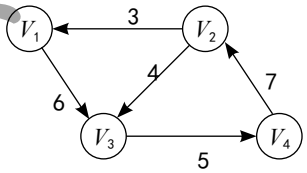


图 6-13 有向网及其邻接矩阵

由图的邻接矩阵，可以得出以下几个结论：

（1）无向图的邻接矩阵一定是对称的，而有向图的邻接矩阵不一定对称。因此，用邻接矩阵来表示一个具有 n 个顶点的有向图时需要 n^2 个单元来存储邻接矩阵；对有 n 个顶点的无向图则只存入上（下）三角阵中剔除了左上右下对角线上的 0 元素后剩余的元素，故只需 $1+2+\ldots+(n-1)=n(n-1)/2$ 个单元。

（2）无向图邻接矩阵的第 i 行（或第 i 列）非零元素的个数正好是第 i 个顶点的度。

（3）有向图邻接矩阵中第 i 行非零元素的个数为第 i 个顶点的出度，第 i 列非零元素的个数为第 i 个顶点的入度，第 i 个顶点的度为第 i 行与第 i 列非零元素个数之和。

（4）无向网（或有向网）的邻接矩阵与无向图（或有向图）的邻接矩阵类似，但要注意用权 wij 代替其中的 1；用∞代替 0。

（5）用邻接矩阵表示图，很容易确定图中任意两个顶点是否有边相连。

图的邻接矩阵存储表示如下：

```
#define MAX_VEX 50                          /* 定义最大顶点个数 */
typedef char VertexType;                    /* 用户定义顶点类型 */
typedef struct Graph
{
   VertexType  vexs[MAX_VERTEX];            /* 顶点向量 */
   int  AdjMatrix[MAX_VEX][MAX_VEX];              /* 邻接矩阵 */
   int  vexnum,arcnum;                      /* 图的当前顶点数和弧（边）数 */
 } MGraph;
```

图的邻接矩阵存储的具体算法如下：

```
#include<stdio.h>
#define MAX_VEX 50
int CreatCost(cost)
int cost[][MAX_VEX];                        /*cost 这个二维数组用于表示图的邻接矩阵 */
{
    int vexnum, arcnum, v1, v2;
    printf(" 请输入图的顶点数和弧数或边数 :\n");
    scanf(" %d,%d", &vexnum, &arcnum);
    for (int i = 0;i<vexnum;i++)
            for (int j = 0;j<vexnum;j++)
                    cost[i][j] = 0;
    for (int k = 0;k<arcnum;k++)
    {
            printf("v1,v2=");
            scanf(" %d,%d", &v1, &v2);      /* 输入所有边或所有弧的一对顶点 V1，V2*/
            cost[v1][v2] = 1;
            /*cost[v2][v1]=1;*/             /* 若为无向图则应加上此语句 */
    }
    return vexnum;
}
```

```
main()
{
    int cost[MAX_VEX][MAX_VEX];
    int vexnum = CreatCost(cost);          /* 建立图的邻接矩阵 */
    printf(" 图的邻接矩阵的输出结果为 :\n");
    for (int i = 0;i<vexnum;i++)
    {
            for (int j = 0;j<vexnum;j++)
                printf(" %3d", cost[i][j]);
            printf("\n");
    }
}
```

6.2.2 邻接表

对于图来说，邻接矩阵是不错的一种图存储结构，但是我们也发现，对于边数相对顶点较少的图，这种结构是存在对存储空间的极大浪费的。因此，我们考虑另外一种存储结构方式 —— 邻接表（Adiacency list）。

图的邻接表存储方法跟树的孩子链表示法相类似，是一种顺序存储和链式存储相结合的存储结构。邻接表的处理方法是这样的：图中顶点用一个一维数组存储，另外，对于顶点数组中，每个数据元素还需要存储指向第一个邻接点的指针，以便于查找该顶点的边信息；图中每个顶点 V_i 的所有邻接点构成一个线性表，由于邻接点的个数不定，所以用单链表存储，无向图称为顶点 V_i 的边表，有向图称为顶点 V_i 作为弧尾的出边表。

因此，在邻接表中存在两种节点结构：顶点表结构和边表结构，如图 6-14 所示。顶点表结构包含数据域（Vertex）：存放顶点信息，指针域（Firstedge），指向边表中第一个节点。边表节点由三个域组成：邻接点域（Adjvex），存放边的终点（即该顶点的邻接点）在顶点表中的下标；数据域（Info），存储与边或弧相关信息（如权值等），如边或弧上没信息即可省略；指针域（Next），指向边表中的下一个节点。

图 6-14 邻接表表示的节点结构

对于图 6-10 中无向图的邻接表表示如图 6-15 所示。

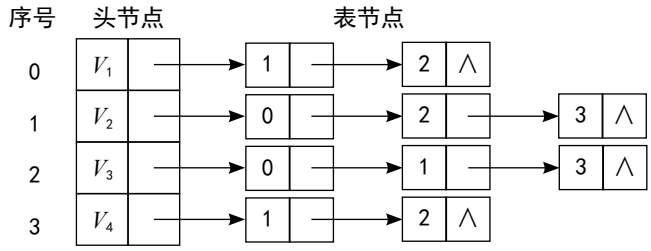


图 6-15　图 6-10 中无向图的邻接表

由图 6-15 可以看出各顶点的度为：D（V_1）=2；D（V_2）=3；D（V_3）=3；D（V_4）=2。

对于图 6-13 中有向网的邻接表表示如图 6-16 所示。

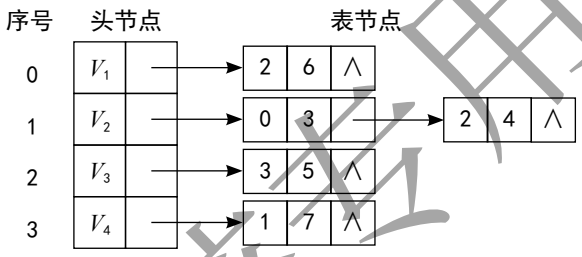


图 6-16　图 6-13 中有向网的邻接表

由图 6-16 可以看出各顶点的出度为：OD（V_1）=1；OD（V_2）=2；OD（V_3）=1；OD（V_4）=1。

对于图 6-13 中有向网的逆邻接表表示如图 6-16 所示。

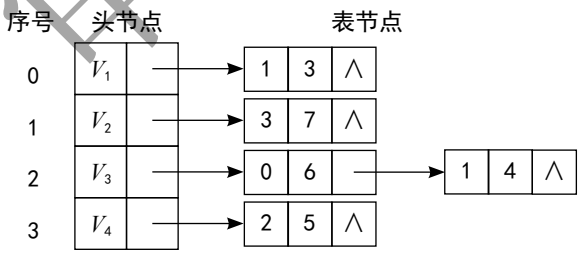


图 6-17　图 6-13 中有向网的逆邻接表

由图 6-17 可以看出各顶点的入度为：ID（V_1）=1；ID（V_2）=1；ID（V_3）=2；ID（V_4）=1。

由此可见，无向图的邻接表，顶点 V_i 的度等于第 i 个链表中的节点数；有向图的邻接表，顶点 V_i 的出度等于第 i 个链表中的节点数，求入度必须遍历整个邻

接表。为便于求 V_i 的入度，需建立有向图的逆邻接表。

在 C 语言中，图的邻接表存储表示如下：

```
#define MAX_VEX 50
typedef struct arcnode                    /* 定义表节点 */
{
   int vextex;
   struct arcnode*next;
}ARCNODE;
typedef struct vexnode                    /* 定义头节点 */
{
   int data;
   ARCNODE  *firstarc;
}VEXNODE;
VEXNODE adjlist[MAX_VEX];                 /* 定义表头向量 adjlist*/
```

由此可得建立无向图邻接表的完整算法如下：

```
#include<stdio.h>
#include<malloc.h>
#define MAX_VEX 50
typedef struct arcnode                    /* 定义表节点 */
{
    int vextex;
    struct arcnode*next;
}ARCNODE;
typedef struct vexnode                    /* 定义头节点 */
{
    int data;
    ARCNODE  *firstarc;
}VEXNODE;
VEXNODE adjlist[MAX_VEX];                 /* 定义表头向量 adjlist*/
int CreatAdjlist()                        /* 建立邻接表 */
{
    ARCNODE  *ptr;
    int arcnum, vexnum, v1, v2;
    printf(" 请输入图的顶点数和弧数或边数 :\n");
    scanf("%d,%d", &vexnum, &arcnum);
    for (int k = 0;k<vexnum;k++)
```

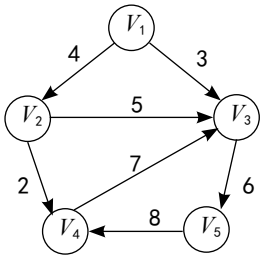
```
                adjlist[k].firstarc = NULL;      /* 为邻接表的 adjlist 数组各元素的
链域赋初值 */
        for (int k = 0;k<arcnum;k++)  /* 为 adjlist 数组的各元素分别建立各自的
链表 */
        {
                printf("v1,v2=");
                scanf("%d,%d", &v1, &v2);
                ptr = (ARCNODE*)malloc(sizeof(ARCNODE));
                /* 给节点 V1 的相邻节点 V2 分配内存空间 */
                ptr->vextex = v2;
                ptr->next = adjlist[v1].firstarc;
                adjlist[v1].firstarc = ptr;        /* 将相邻节点 V2 插入表头节点 V1
之后 */
                ptr = (ARCNODE*)malloc(sizeof(ARCNODE));
                /* 对于有向图此后的三行语句要删除 */
                ptr->vextex = v1;                  /* 给节点 V2 的相邻节点 V1 分配
内存空间 */
                ptr->next = adjlist[v2].firstarc;
                adjlist[v2].firstarc = ptr;        /* 将相邻节点 V1 插入表头节点 V2
之后 */
        }
        return vexnum;
    }
    main()
    {
        ARCNODE *p;
        int n = CreatAdjlist();                    /* 建立邻接表并返回顶点的个数 */
        printf(" 图的邻接表的输出结果如下 :\n");
        for (int i = 0;i<n;i++)                    /* 输出邻接表中各个链表的信息 */
        {
                printf("%d==>", i);
                p = adjlist[i].firstarc;
                while (p != NULL)
                {
                        printf("--->%d", p->vextex);
                        p = p->next;
                }
                printf("\n");
```

```
    }
}
```

6.2.3 边集数组

图的另外一种常见的存储方法是边集数组它是由两个一维数组构成，一个是存储顶点的信息，另一个是存储边的信息，这个边数组每个数据元素由一条边的起点下标（begin），终点下标（end）和权（weight）组成。它适用于一些以边为主的操作，比如带权图（网）用边集数组表示时，列出每条边所依附的两个顶点及边上的权，即每个数组元素代表一条边的信息。

如图 6-18 所示为一个有向网图的边集数组存储示意图。



(a) 有向网图

Vertex[5]= | V_1 | V_2 | V_3 | V_4 | V_5 |

下标	0	1	2	3	4	5	6
始点	1	1	2	2	3	4	5
终点	2	3	3	4	5	3	4
权值	4	3	5	2	6	7	8

(b) 边集数组存储

图 6-18 有向网图及其边集数组存储示意图

在 C 语言中，图的边集数组存储表示如下：

```
#define MAX_ARC 50
typedef struct edges
{
   int bv,ev,w;
}EDGES;
EDGES edgeset[MAX_ARC];
```

由此可得建立带权网图的边集数组的完整算法如下：

```
#include<stdio.h>
#define MAX_ARC 50
typedef struct edges
{
    int bv, ev, w;
}EDGES;
EDGES edgeset[MAX_ARC];
```

```
int CreatEdgeSet()                    /* 建立带权图的边集数组 */
{
    int arcnum, i;
    printf(" 请输入边数：\n");
    scanf("%d", &arcnum);
    for (i = 0;i<arcnum;i++)
    {
            printf("bv,ev,w=");
            scanf("%d,%d,%d", &edgeset[i].bv, &edgeset[i].ev,
&edgeset[i].w);
            /* 输入每条边的起、止顶点及边的权值 */
    }
    return(arcnum);
}
main()
{
    int arcnum = CreatEdgeSet(); /* 建立带权图的边集数组并返回其边数 */
    printf("bv ev w\n");
    for (int i = 0;i<arcnum;i++)    /* 输出边集数组中每一条边的信息 */
            printf("%d %d %d\n", edgeset[i].bv, edgeset[i].ev, edgeset[i].
w);
}
```


6.3　图 的 遍 历

图的遍历是图的一种基本操作，图的许多其他操作都是建立在遍历操作的基础之上。图的遍历指的是从图中的任一顶点出发，对图中的所有顶点访问一次且只访问一次。图的遍历操作和树的遍历操作功能相似。由于图结构本身的复杂性，所以图的遍历操作也较复杂，主要表现在以下四方面：

（1）在图结构中，没有一个“自然”的首节点，图中任意一个顶点都可作为第一个被访问的节点。

（2）在非连通图中，从一个顶点出发，只能够访问它所在的连通分量上的所有顶点，因此，还需考虑如何选取下一个出发点以访问图中其余的连通分量。

（3）在图结构中，如果有回路存在，那么一个顶点被访问之后，有可能沿回路又回到该顶点。

（4）在图结构中，一个顶点可以和其他多个顶点相连，当这样的顶点访问过后，存在如何选取下一个要访问的顶点的问题。

图的遍历方法目前常用的有深度优先搜索法和广度（宽度）优先搜索法两种算法，这两种遍历次序对无向图和有向图都适应。

6.3.1 深度优先搜索（DFS）

图的深度优先搜索 (Depth First Search)，和树的先序遍历比较类似。从图中某顶点 V 出发进行深度优先搜索基本思想是：

（1）访问顶点 V；

（2）依次从 V 的各个未被访问的邻接点出发，对图进行深度优先搜索遍历；直至图中和 V 有路径相通的顶点都被访问；

（3）若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。

对图进行深度优先搜索时，按访问顶点的先后次序得到的顶点序列称为图的深度优先搜索序列，简称 DFS 序列。一个图的 DFS 序列可能不唯一，它与算法的存储结构密切相关，现在以图 6-19(a) 所示无向图的邻接表存储结构为例说明，其邻接矩阵见图 6-20。

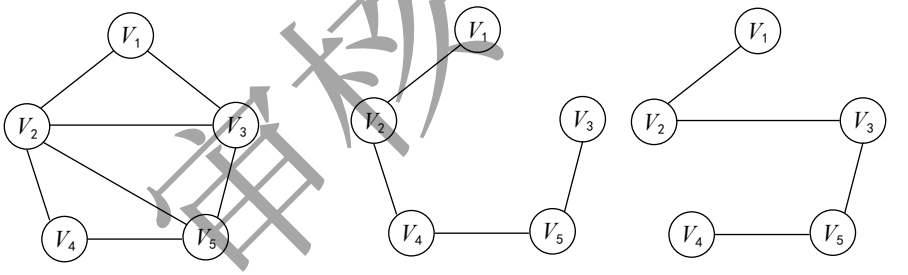


图 6-19 无向图及 DFS 生成树

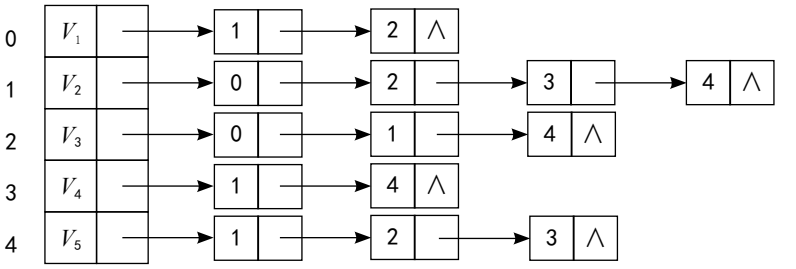


图 6-20 图 6-19(a) 的邻接表

搜索过程：假设从顶点 V_1 出发进行搜索，在访问了顶点 V_1 之后，因为 V_1 的邻接点分别有 V_2 和 V_3，选择邻接点 V_2。因为 V_2 未曾访问，则从 V_2 出发进行搜索，V_2 的未被访问过的邻接点分别有 V_3，V_4 和 V_5，选择 V_3，则从 V_3 出发进行搜索，V_3 的未被访问过的邻接点只有 V_5，选择 V_5，V_5 的未被访问过的邻接点只有 V_4。扫描整个邻接表，所有的顶点都已访问完毕，最终访问得到的顶点访问序列为：$V_1\ V_2\ V_3\ V_5\ V_4$。

对于不同的邻接表存储结构还可能得到 $V_1\ V_2\ V_4\ V_5\ V_3$，$V_1\ V_3\ V_2\ V_4\ V_5$，$V_1\ V_3\ V_5\ V_4\ V_2$ 等不同的 DFS 序列。

由图中的全部顶点和深度优先搜索过程所经过的边集，即构成了图的深度优先生成树。对于的图的存储结构不同，其 DFS 序列可以不唯一，DFS 生成树也可以不唯一。如图 6-19(b) 所示。

以邻接表为存储结构的深度优先搜索的算法如下：

```
#include<malloc.h>
#include<stdio.h>
#define MAX_VEX 50
typedef struct arcnode                    /*定义表节点 */
{
    int vextex;
    struct arcnode *next;
}ARCNODE;

typedef struct vexnode                    /* 定义头节点 */
{
    int data;
    ARCNODE* firstarc;
}VEXNODE;

VEXNODE adjlist[MAX_VEX];                 /* 定义表头向量 adjlist*/

int CreatAdjList()                        /* 建立邻接表 */
{
    ARCNODE*ptr;
    int arcnum, vexnum, v1, v2;
    printf(" 请输入图的顶点数和弧数或边数 :\n");
    scanf("%d,%d", &vexnum, &arcnum);
    for (int k = 0;k<vexnum;k++)
```

```
                adjlist[k].firstarc = NULL;      /* 为邻接链表的adjlist数组各元素
的链域赋初值 */
        for (int k = 0;k<arcnum;k++)             /* 为adjlist数组的各元素分别建立
各自的链表 */
        {
                printf("v1,v2=");
                scanf("%d,%d", &v1, &v2);
                ptr = (ARCNODE*)malloc(sizeof(ARCNODE));
                /* 给节点 V1 的相邻节点 V2 分配内存空间 */
                ptr->vextex = v2;
                ptr->next = adjlist[v1].firstarc;
                adjlist[v1].firstarc = ptr;      /* 将相邻节点 V2 插入表头节点 V1
之后 */
                ptr = (ARCNODE*)malloc(sizeof(ARCNODE));
                /* 对于有向图此后的四行语句要删除 */
                ptr->vextex = v1;                /* 给节点 V2 的相邻节点 V1 分配
内存空间 */
                ptr->next = adjlist[v2].firstarc;
                adjlist[v2].firstarc = ptr;      /* 将相邻节点 V1 插入表头节点 V2
之后 */
        }
        return vexnum;
    }

    void DFS(v)                  /* 从某顶点 V 出发按深度优先搜索进行图的遍历 */
    int v;
    {
        int w;
        ARCNODE *p;
        p = adjlist[v].firstarc;
        printf("%d\n", v);                       /* 输出访问的顶点 */
        adjlist[v].data = 1;                     /* 顶点标志域置1，表明已访问过 */
        while (p != NULL)
        {
                w = p->vextex;                   /* 取出顶点V的某相邻顶点的
序号 */
                if (adjlist[w].data == 0)
                        DFS(w);                  /* 如果该顶点未被访问过则递归调
```

```
用，从该顶点出发，沿着它的各相邻顶点向下搜索 */
                p = p->next;
        }
    }

    main()
    {
        int n = CreatAdjList(adjlist);          /* 建立邻接表并返回顶点的个数 */
        printf(" 深度优先搜索序列为 :\n");
        DFS(0);                    /* 从顶点 0 出发，按深度优先搜索进行图的遍历 */
    }
```

分析上述算法，在遍历时，对图中每个顶点至多调用一次 DFS 函数，因为一旦某个顶点被标志成已被访问，就不再从它出发进行搜索。因此，遍历图的过程实质上是对每个顶点查找其邻接点的过程。其耗费的时间则取决于所采用的存储结构。当用二维数组表示邻接矩阵图的存储结构时，查找每个顶点的邻接点所需时间为 $O(n^2)$，其中 n 为图中顶点数。而当以邻接表作图的存储结构时，找邻接点所需时间为 $O(e)$，其中 e 为无向图中边的数或有向图中弧的数。由此，当以邻接表作存储结构时，深度优先搜索遍历图的时间复杂度为 $O(n+e)$。

6.3.2　广度优先搜索（BFS）

广度优先搜索（Breadth_First Search）遍历类似于树的按层次遍历的过程。从图中某顶点 V 出发进行广度优先搜索基本思想是：

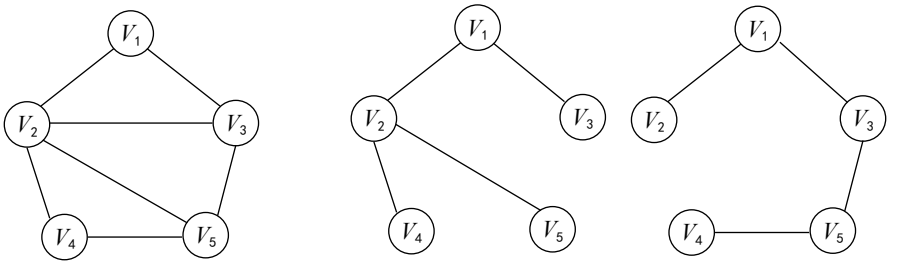
（1）访问顶点 V。

（2）访问 V 的所有未被访问的邻接点 V_1，V_2，…，V_k。

（3）依次从这些邻接点 V_1，V_2，…，V_k 出发，访问它们的所有未被访问的邻接点；并使用“先被访问的邻接点”先与“后被访问的邻接点”被访问，依此类推，直到图中所有访问过的顶点的邻接点都被访问到。

对于广度优先搜索算法，要记录与一个顶点相邻接的全部顶点，由于访问过这些顶点之后，还将按照先被访问的顶点就先访问它的邻接点的方式进行广度优先搜索。为此，需要用一个先进先出的队列来记录这些顶点。

对图进行广度优先搜索时，按访问顶点的先后次序得到的顶点序列称为图的广度优先搜索序列，简称 BFS 序列。一个图的 BFS 序列可能不唯一，它与算法的存储结构密切相关，现在以图 6-21 无向图及图 6-22 它的邻接表存储结构为例说明。



(a) 无向图　　(b) DFS 生成树

图 6-21　无向图及 BFS 生成树

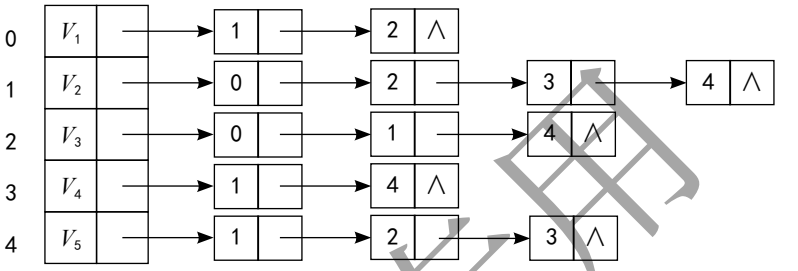


图 6-22　图 6-21(a) 的邻接表

搜索过程：假设从顶点 V_1 出发进行搜索，先访问了顶点 V_1，并将 V_1 入队，然后 V_1 出队。由于顶点 V_1 的相邻顶点分别 V_2 和 V_3，并且 V_2、V_3 未被访问，则访问 V_2、V_3，并将 V_2、V_3 入队，V_1 的邻接点访问完毕。V_2 出队，由于与顶点 V_2 相邻的顶点有 V_1、V_3、V_4、V_5，则访问其中未被访问过的顶点 V_4、V_5，并将 V_4、V_5 入队，此时 V_2 的邻接点访问完毕。V_3 出队，由于与 V_3 相邻的顶点均被访问过，V_4 出队，V_4 相邻的顶点也被访问过，V_5 出队，V_5 相邻的顶点也均被访问过，至此所有的顶点都已访问完毕，这时队列为空，遍历过程结束。最终访问得到的顶点访问序列为：$V_1\ V_2\ V_3\ V_4\ V_5$。

对于不同的邻接表存储结构还可能得到 $V_1\ V_2\ V_3\ V_5\ V_4$，$V_1\ V_3\ V_2\ V_5\ V_4$ 等不同的 BFS 序列。

由图中的全部顶点和广度优先搜索过程所经过的边集，即构成了图的广度优先生成树。对于的图的存储结构不同，其 BFS 序列可以不唯一，BFS 生成树也可以不唯一。如图 6-21(b) 所示。

以邻接表为存储结构的广度优先搜索的算法如下：

```
#include<stdio.h>
#include<malloc.h>
#define MAX_VEX 50
typedef struct arcnode                    /* 定义表节点 */
```

```
    {
        int vextex;
        struct arcnode  *next;
    }ARCNODE;

    typedef struct vexnode                  /* 定义头节点 */
    {
        int data;
        ARCNODE  *firstarc;
    }VEXNODE;

    VEXNODE adjlist[MAX_VEX];               /* 定义表头向量 adjlist*/
    int CreatAdjList()                      /* 建立邻接表 */
    {
        ARCNODE *ptr;
        int arcnum, vexnum, v1, v2;
        printf(" 请输入图的顶点数和弧数或边数 :\n");
        scanf("%d,%d", &vexnum, &arcnum);
        for (int k = 0;k<vexnum;k++)
            adjlist[k].firstarc = NULL;     /* 为邻接链表的 adjlist 数组各元素
的链域赋初值 */
        for (int k = 0;k<arcnum;k++)        /* 为 adjlist 数组的各元素分别建立
各自的链表 */
        {
            printf("v1,v2=");
            scanf("%d,%d", &v1, &v2);
            ptr = (ARCNODE*)malloc(sizeof(ARCNODE));
            /* 给节点 V1 的相邻节点 V2 分配内存空间 */
            ptr->vextex = v2;
            ptr->next = adjlist[v1].firstarc;
            adjlist[v1].firstarc = ptr;     /* 将相邻节点 V2 插入表头节点 V1
之后 */
            ptr = (ARCNODE*)malloc(sizeof(ARCNODE));
            /* 对于有向图此后的三行语句要删除 */
            ptr->vextex = v1;               /* 给节点 V2 的相邻节点 V1 分配
内存空间 */
            ptr->next = adjlist[v2].firstarc;
            adjlist[v2].firstarc = ptr;     /* 将相邻节点 V1 插入表头节点 V2
```

```
之后 */
        }
        return(vexnum);
    }

    void BFS(v)                              /* 从某顶点 V 出发按广度优先搜索
进行图的遍历 */
    int v;
    {
        int queue[MAX_VEX];
        int front = 0, rear = 1;
        /*int w;*/
        ARCNODE*p;
        printf(" 广度优先搜索序列为 :\n");
        p = adjlist[v].firstarc;
        printf("%d\n", v);                   /* 访问初始顶点 */
        adjlist[v].data = 1;
        queue[rear] = v;                     /* 初始顶点入队列 */
        while (front != rear)                /* 队列不为空时循环 */
        {
            front = (front + 1) % MAX_VEX;
            v = queue[front];                /* 按访问次序依次出队列 */
            p = adjlist[v].firstarc;         /* 找 V 的邻接点 */
            while (p != NULL)
            {
                if (adjlist[p->vextex].data == 0)
                {
                    adjlist[p->vextex].data = 1;
                    printf("%d\n", p->vextex); /* 访问该点并使之入队列 */
                    rear = (rear + 1) % MAX_VEX;
                    queue[rear] = p->vextex;
                }
                p = p->next;                 /* 找 V 的下一个邻接点 */
            }
        }
    }

    main()
```

```
    {
        int n = CreatAdjList(); /* 建立邻接表并返回顶点的个数 */
        BFS(0);                                    /* 从顶点 0 出发，按广度优先搜索
进行图的遍历 */
    }
```

分析上述算法，每个顶点至多进一次队列。遍历图的过程实质是通过边或弧找邻接点的过程，因此广度优先搜索遍历图的时间复杂度和深度优先搜索遍历相同，两者不同之处仅仅在于对顶点访问的顺序不同。

6.4　图 的 应 用

6.4.1　最小生成树

在一给定的无向图 $G=(V, E)$ 中，(u, v) 代表连接顶点 u 与顶点 v 的边（即），而 $w(u, v)$ 代表此边的权重，若存在 T 为 E 的子集（即）且为无循环图，使得的 $w(T)$ 最小，则此 T 为 G 的最小生成树，有

$$w(t)=\sum_{(u,v)\in t} w(u,v) \tag{6-2}$$

最小生成树其实是最小权重生成树的简称。

最小生成树有许多重要的应用。例如：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，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，因此另一个目标是要使铺设光缆的总费用最低。这就需要找到带权的最小生成树。

最小生成树可以用普里姆（Prim）算法或克鲁斯卡尔（Kruskal）算法求出。

1. 普里姆算法

普里姆（Prim）算法的基本思想：

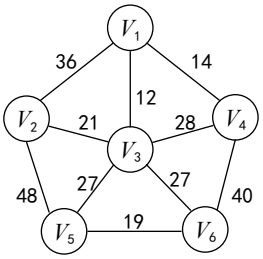
设 $G=(V, E)$ 是一个连通图网，其中顶点集合为 V，边集合为 E；令 $T=(U, \mathrm{TE})$ 是 G 的最小生成树。

（1）初始化：$U=\{v_0\}$，其中 v_0 为集合 V 中的任一节点（起始点），TE = {}，为空；

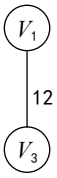
（2）重复下列操作：在所有 $u \in U$，$v \in V-U$ 的边中找一条代价最小的边 (u, v) 并入集合 TE，同时 v 并入 U，直到 $U=V$：

（3）此时 TE 中必有 $n-1$ 条边，T 就是最小生成树。

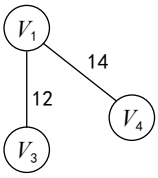
例如，对于 6.23(a) 所示连通网，图 6-23(b)~ 图 6-23(f) 给出了从顶点 V_1 出发，利用 Prim 算法构造最小生成树的过程。



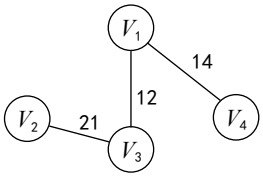
(a) 连通网，$U=\{V_1\}$
cost={(v_1, v_2) 36,
(v_1, v_3) 12, (v_1, v_4) 14,
(v_1, v_5) ∞, (v_1, v_6) ∞}



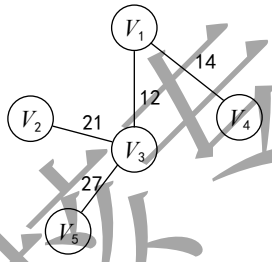
(b) $U=\{V_1, V_3\}$
cost={(v_1, v_4) 14,
(v_3, v_2) 21, (v_3, v_5) 27,
(v_3, v_6) 27}



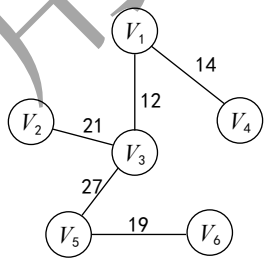
(c) $U=\{V_1, V_3, V_4\}$
cost={(v_3, v_2) 21,
(v_3, v_5) 27, (v_3, v_6) 27}



(d) $U=\{V_1, V_3, V_4, V_2,\}$
cost={(v_3, v_5) 27,
(v_3, v_6) 27}



(e) $U=\{V_1, V_3, V_4, V_2, V_5,\}$
cost={(v_5, v_6) 19}



(f) $U=\{V_1, V_3, V_4, V_2, V_5, V_6\}$
cost={}

图 6-23 Prim 算法构造最小生成树的过程

利用普里姆（Prim）算法构造最小生成树的完整程序如下：

```
#include <stdio.h>
#define MAX_VEX 50
int CreatCost(cost)
int cost[][MAX_VEX];                    /*cost 数组表示带权图的邻接矩阵 */
{
    int vexnum, arcnum, v1, v2, w;
    printf(" 请输入图的顶点数和弧数或边数 :\n");
    scanf("%d,%d", &vexnum, &arcnum);
    for (int i = 0;i<vexnum;i++)        /* 初始化带权图的邻接矩阵 */
        for (int j = 0;j<vexnum;j++)
```

```
                cost[i][j] = 32767;        /*32767 表示无穷大 */
        for (int k = 0;k<arcnum;k++)
        {
            printf("v1,v2,w = ");
            /* 输入所有边 ( 或弧 ) 的一对顶点 V1,V2 和权值 */
            scanf("%d,%d,%d", &v1, &v2, &w);
            cost[v1][v2] = w;
            cost[v2][v1] = w;
        }
        return(vexnum);
    }

    void Prim(cost, vexnum)          /*Prim 算法产生从顶点 V0 开始的最小生
成树 */
    int cost[][MAX_VEX], vexnum;
    {
        int lowcost[MAX_VEX], closest[MAX_VEX], min;
        for (int i = 0;i<vexnum;i++)
        {
            lowcost[i] = cost[0][i];    /* 初始化 */
            closest[i] = 0;                    /* 初始化 */
        }
        closest[0] = -1;                       /*V0 选入 U*/
        for (int i = 1;i<vexnum;i++)           /* 从 U 之外求离 U 中某一顶点最近
的顶点 */
        {
            min = 32767;
            int k = 0;
            for (int j = 0;j<vexnum;j++)
                if (closest[j] != -1 && lowcost[j]<min)
                {
                    min = lowcost[j];
                    k = j;
                }
            if (k)
            {
                printf("(%d,%d)%2d\n", closest[k], k, lowcost[k]);    /* 输出边
及其权值 */
```

```
                closest[k] = -1;          /*k 选入 U*/
                for (int j = 1;j<vexnum;j++)
                    if (closest[j] != -1 && cost[k][j]<lowcost[j])
                    {
                        lowcost[j] = cost[k][j];                    /* 由 k 的加入，修改 lowcost 数组 */
                        closest[j] = k;           /*k 加入到 U 中 */
                    }
            }
        }
    }

    main()
    {
        int cost[MAX_VEX][MAX_VEX];
        int vexnum = CreatCost(cost);              /* 建立图的邻接矩阵 */
        printf(" 输出结果如下 :\n");
        Prim(cost, vexnum);
    }
```

分析 Prim 算法，设连通网中有 n 个顶点，则第一个进行初始化的循环语句需要执行 $n-1$ 次，第二个循环共执行 $n-1$ 次，内嵌套两个循环，其一是在长度为 n 的数组中求最小值，需要执行 $n-1$ 次，其二是调整辅助数组，需要执行 $n-1$ 次，所以，Prim 算法的时间复杂度为 $O(n^2)$，与网中的边数无关，因此适用于求稠密网的最小生成树。

2. 克鲁斯卡尔算法

克鲁斯卡尔（Kruskal）算法的基本思想：

设 $G=(V, E)$ 是一个连通网，其中顶点集合为 V，边集合为 E；令 $T=(U,$ TE）是 G 的最小生成树。

（1）初始化。$U=V$，TE = {}，为空，这样 T 中各顶点各自构成一个连通分量。

（2）将图 G 中所有 e 个边按权值从小到大排序，依次考察边集合 E 中的各条边。

（3）循环。从权值最小的边开始遍历每条边，如果这条边连接的两个顶点于图 G 中不在同一个连通分量中添加这条边到图 TE 中，同时把两个连通分量连接成一个连通分量；如果这条边的两个顶点属于同一个连通分量，则舍去此边，

以免造成回路。

（4）直至图 G 中所有的顶点都在同一个连通分量中，也就是说当 T 中的连通分量个数为 1 时，此连通分量便为 G 的一棵最小生成树。

用克鲁斯卡尔（Kruskal）算法对图 6-24(a) 所示连通网构造最小生成树的过程如图 6-24(b)~ 图 6-24(f) 所示。

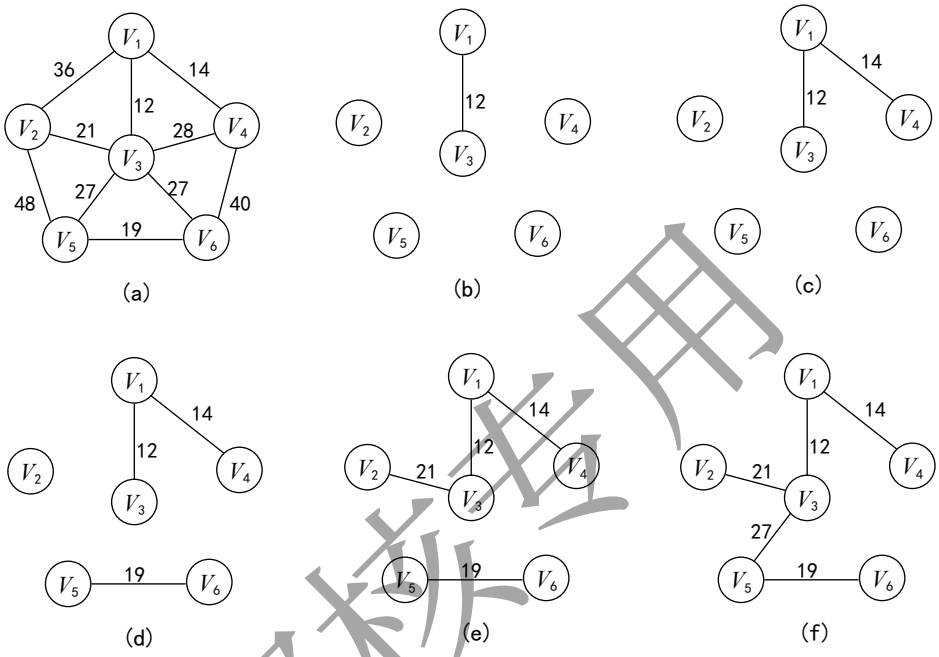


图 6-24　Kruskal 算法构造最小生成树的过程

利用克鲁斯卡尔（Kruskal）算法构造最小生成树的完整程序如下：

```
#include<stdio.h>
#include<malloc.h>
#define MAX_VEX 50
typedef struct edges                /* 定义边集数组元素结构 */
{
    int bv, ev, w;
}EDGES;
EDGES edgeset[MAX_VEX];             /* 定义边集数组，用于存储图的各条边 */

int CreatEdgeSet()                  /* 建立边集数组函数 */
{
    int arcnum;
```

```
        printf("\nInput arcnum : ");
        scanf("%d", &arcnum);          /* 输入图中的边数 */
        for (int i = 0;i<arcnum;i++)
        {
            printf("bv,ev,w = ");      /* 输入每条边的起、终点及边上的权值 */
            scanf("%d,%d,%d", &edgeset[i].bv, &edgeset[i].ev, &edgeset[i].
w);
        }
        return(arcnum);                /* 返回图中的边数 */
    }

    Sort(int n)                        /* 对边集数组按权值升序排序，其中 n 为数组元素
的个数，即图的边数 */
    {
        EDGES t;
        for (int i = 0; i < n - 1; i++)
        {
            for (int j = i + 1; j<n; j++)
                if (edgeset[i].w>edgeset[j].w)
                {
                    t = edgeset[i];
                    edgeset[i] = edgeset[j];
                    edgeset[j] = t;
                }
        }
    }

    int Seeks(int set[], int v         /* 确定顶点 V 所在的连通分量的根节点 */
    {
        int i = v;
        while (set[i]>0)
            i = set[i];
        return(i);
    }

    Kruskal(int e)                     /*Kruskal 算法求最小生成树，参数 e 为边
集数组中的边数 */
```

```
    {
        int set[MAX_VEX], v1, v2, i;
        printf("Output of Kruskal : \n");
        for (i = 0;i<MAX_VEX;i++)
            set[i] = 0;                    /*set 数组的初值为 0, 表示每一个顶点自成
一个分量 */
        i = 0;                             /*i 表示待获取的生成树中的边在边集数组
中的下标 */
        while (i<e)
        {
            v1 = Seeks(set, edgeset[i].bv);     /* 确定边的起始顶点所在的连通分
量的根节点 */
            v2 = Seeks(set, edgeset[i].ev);     /* 确定边的终止顶点所在的连通分
量的根节点 */
            if (v1 != v2)                  // 当边所依附的两个顶点不在同一连通分量
时 , 将该边加入生成树
            {
                printf("(%d,%d)  %d\n", edgeset[i].bv, edgeset[i].ev,
edgeset[i].w);
                set[v1] = v2;              /* 将 v1,v2 设为在同一连通分量中 */
            }
            i++;
        }
    }

    main()
    {
        int arcnum = CreatEdgeSet(); /* 建立图的边集数组 , 并返回其中的边数 */
        Sort(arcnum);                  /* 对边集数组按权值升序排序 */
        printf(" 输出结果如下：\n");
        printf("\nbv   ev   w \n");
        for (int i = 0;i<arcnum;i++)   /* 输出排序后的边集数组 */
            printf("%d    %d    %d\n", edgeset[i].bv, edgeset[i].ev, edgeset[i].
w);
        Kruskal(arcnum);               /* 利用克鲁斯卡尔算法求图的最小生
成树 */
    }
```

克鲁斯卡尔（Kruskal）算法的时间复杂度为 $O(e)$，其中 e 为加权连通图中边的个数。相对于普里姆（Prim）算法而言，克鲁斯卡尔（Kruskal）算法适用于求稀疏网的最小生成树。

6.4.2 最短路径

最短路径问题是图论研究中的一个经典算法应用问题。所谓最短路径问题是指：如果从图中某一顶点（源点）到达另一顶点（终点）的路径可能不止一条，如何找到一条路径使得沿此路径上各边的权值总和（称为路径长度）达到最小。

现在介绍两种比较常用的求最短路径算法：一个是求从某个源点到其他各顶点的最短路径的迪杰斯特拉（Dijkstra）算法；另一个是求每一对顶点之间的最短路径的弗洛伊德（Floyed）算法。

1. 求单源最短路径的迪杰斯特拉算法

迪杰斯特拉算法是典型的最短路径路由算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra 算法能得出最短路径的最优解，但由于它遍历计算的节点很多，所以效率低。

迪杰斯特拉算法的基本思想是：令 $G=(V, E)$ 为一个带权有向图，把图中的顶点集合 V 分成两组，第一组为已求出最短路径的顶点集合 S（初始时 S 中只有源节点，以后每求得一条最短路径，就将它对应的顶点加入到集合 S 中，直到全部顶点都加入到 S 中）；第二组是未确定最短路径的顶点集合 U。在加入过程中，总保持从源节点 v 到 S 中各顶点的最短路径长度不大于从源节点 v 到 U 中任何顶点的最短路径长度。

算法步骤如下：

（1）初始化时，S 只含有源节点。

（2）从 U 中选取一个距离 v 最小的顶点 k 加入 S 中（该选定的距离就是 v 到 k 的最短路径长度）。

（3）以 k 为新考虑的中间点，修改 U 中各顶点的距离；若从源节点 v 到顶点 u 的距离（经过顶点 k）比原来距离（不经过顶点 k）短，则修改顶点 u 的距离值，修改后的距离值是顶点 k 的距离加上 k 到 u 的距离。

（4）重复步骤（2）和（3），直到所有顶点都包含在 S 中。

迪杰斯特拉（Dijkstra）算法基于的存储结构如下：

（1）图的存储结构：带权的邻接矩阵存储结构。

（2）数组 dist[n]：每个分量 dist[i] 表示当前所找到的从始点 v 到终点 v_i 的最

短路径的长度。初态为：若从 v 到 v_i 有弧，则 dist[i] 为弧上权值；否则置 dist[i] 为 ∞。

（3）数组 path[n]：path[i] 是一个字符串，表示当前所找到的从始点 v 到终点 v_i 的最短路径。初态为：若从 v 到 v_i 有弧，则 path[i] 为 v_i；否则置 path[i] 空串。

（4）数组 s[n]：存放源点和已经生成的终点，其初态为只有一个源点 v。

以图 6-25 为例，执行迪杰斯特拉（Dijkstra）算法，则得到从顶点 V_0 到其余各顶点的最短路径，以及算法执行过程中数组 dist 和 path 的变化状况，见表 6-1。

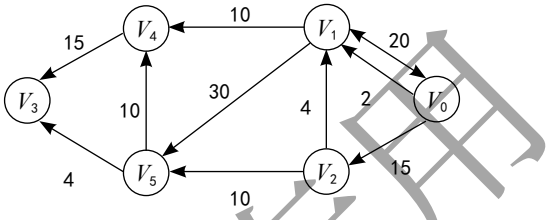


图 6-25　有向网

表 6-1 迪杰斯特拉（Dijkstra）算法的执行过程

循环	S	从V_0到各顶点的最短路径的求解过程				
		V_1(dist[1],path[1])	V_2(dist[2],path[2])	V_3(dist[3],path[3])	V_4(dist[4],path[4])	V_5(dist[5],path[5])
初始	{V_0}	(20、"V_0V_1")	(15、"V_0V_2")	(∞、"")	(∞、"")	(∞、"")
1	{V_0V_2}	(19、"$V_0V_2V_1$")	输出15 输出"V_0V_2"	(∞、"")	(∞、"")	(25、"$V_0V_2V_5$")
2	{$V_0V_2V_1$}	输出19 输出"$V_0V_2V_1$"		(∞、"")	(29、"$V_0V_2V_1V_4$")	(25、"$V_0V_2V_5$")
3	{$V_0V_2V_1$ V_5}			(29、"$V_0V_2V_5V_3$")	(29、"$V_0V_2V_1V_4$")	输出25 输出"$V_0V_2V_5$"
4	{$V_0V_2V_1$ V_5V_4}			(29、"$V_0V_2V_5V_3$")	输出29 输出"$V_0V_2V_1V_4$"	
5	{$V_0V_2V_1$ $V_5V_4V_3$}			输出29 输出"$V_0V_2V_5V_3$"		

用迪杰斯特拉算法，求单源最短路径的完整程序如下：

```
#include<stdio.h>
#include<malloc.h>
#define MAX_VEX 50
int CreatCost(cost)                    /* 建立图的邻接矩阵 */
```

```
    int cost[][MAX_VEX];                /*cost 数组表示图的邻接矩阵 */
    {
        int vexnum, arcnum, i, j, k, v1, v2, w;
        printf(" 请输入图的顶点数和弧数或边数 :\n");
        scanf("%d,%d", &vexnum, &arcnum);
        for (i = 0;i<vexnum;i++)
            for (j = 0;j<vexnum;j++)
                cost[i][j] = 9999;          /* 设 9999 代表无限大 */
        for (k = 0;k<arcnum;k++)
        {
            printf("v1,v2,w = ");
            scanf("%d,%d,%d", &v1, &v2, &w);        /* 输入所有边或弧的一对
顶点 V1,V2 */
            cost[v1][v2] = w;
        }
        return(vexnum);
    }

    void Dijkstra(cost, vexnum)                 /*Dijkstra 算法求从源点出发的最
短路径 */
    int cost[][MAX_VEX], vexnum;
    {
        int path[MAX_VEX], s[MAX_VEX], dist[MAX_VEX], n, w, v, sum, min,
v0;
        /*S 数组用于记录顶点 V 是否已经确定了最短路径 ,S[V]=1, 顶点 V 已经确定
了最短路径 ,S[V]=0, 顶点 V 尚未确定最短路径。dist 数组表示当前求出的从 V0 到
Vi 的最短路径。path 是路径数组 , 其中 path[i] 表示从源点到顶点 Vi 之间的最短路
径上 Vi 的前驱顶点 , 如有路径 (v0,v2,v4), 则 path[4]=2*/
        printf(" 输入源点 v0 : ");
        scanf("%d", &v0);                   /* 输入源点 V0 */
        for (int i = 0;i<vexnum;i++)
        {
            dist[i] = cost[v0][i];          /* 从源点 V0 到各顶点的最短路径为相应弧
上的权 */
            s[i] = 0;                       /* 初始化 */
            if (cost[v0][i]<9999)
                path[i] = v0;               /* 初始化，path 记录当前最短路径，即顶点
```

```
的直接前趋 */
        }
        s[v0] = 1;                          /* 将源点加入 S 集合中 */
        for (int i = 0;i<vexnum;i++)
        {
            min = 9999;                     /* 本例设各边上的权值均小于 9999*/
            for (int j = 0;j<vexnum;j++)        /* 从 S 集合外找出距离源点最近的
顶点 w*/
                if ((s[j] == 0) && (dist[j]<min))
                {
                    min = dist[j];
                    w = j;
                }
            s[w] = 1;                       /* 将 w 加入 S 集合 , 即 w 已是求出最短路
径的顶点 */
            for (v = 0;v<vexnum;v++) /* 根据 w 修改 dist[]*/
                if (s[v] == 0)              /* 修改未加入的顶点的路径长度 */
                    if (dist[w] + cost[w][v]<dist[v])
                    {
                        /* 修改 V-S 集合中各顶点的最短路径长度 */
                        dist[v] = dist[w] + cost[w][v];
                        path[v] = w;            /* 修改 V-S 集合中各顶点的最短
路径 */
                    }
        }
        printf(" 从源点 %d 到其他各顶点的最短路径：\n", v0);
        for (int i = 1;i<vexnum;i++)            /* 输出从某源点到其他各顶点的最
短路径 */
            if (s[i] == 1)
            {
                w = i;
                while (w != v0)
                {
                    printf("%d<--", w);
                    w = path[w];                /* 通过找到前驱顶点 , 反向输出最
短路径 */
                }
```

```
                printf("%d", w);
                printf("   %d\n", dist[i]);
            }
            else
            {
                printf("%d<--%d", i, v0);
                printf("   9999\n");              /* 不存在路径时，路径长度设为
9999*/
            }
        }
        main()
        {
            int cost[MAX_VEX][MAX_VEX];
            int vexnum = CreatCost(cost);          /* 建立图的邻接矩阵 */
            Dijkstra(cost, vexnum);
        }
```

分析迪杰斯特拉算法的时间性次能，设 n 是图的顶点个数，第一次循环执行行 $n-1$ 次；第二次循环也执行行 $n-1$ 次，内嵌两个并列的循环，第一次循环是在数组 dist 中求最小值，执行行 $n-1$ 次，第二次循环是修改数组 dist 和 path，需要执行 n 次，所以总的时间复杂度是 $O(n_2)$。

2. 求每一对顶点之间的短路径的弗洛伊德算法

我们通过迪杰斯特拉算法解决了从某个源点到其余各顶点的最短路径问题。从循环嵌套很容易得到此算法的时间复杂度为 $O(n^2)$。可是怎么只找到从源点到某一个特定终点的最短路径，其实这个问题和求源点到其他所有顶点的最短路径一样复杂，时间复杂度依然是 $O(n^2)$。

此时比较简单方法就是对每个顶点当作源点运行一次迪杰斯特拉算法，等于在原有算法的基础上，再来一次循环，此时整个算法的时间复杂度为 $O(n^3)$。对此，再来学习另一个求最短路径的算法 —— 弗洛伊德，它求所有顶点到所有顶点的最短路径，时间复杂度也为 $O(n^3)$，但其算法非常简洁优雅。

弗洛伊德算法的基本思想是：对于从 v_i 到 v_j 的弧，进行 n 次试探：首先考虑路径 v_i，v_0，v_j 是否存在，如果存在，则比较 v_i，v_j 和 v_i，v_0，v_j 的路径长度，取较短者为从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径。在路径上再增加一个顶点 v_1，依此类推，在经过 n 次比较后，最后求得的必是从顶点 v_i 到顶点 v_j 的最短路径。

弗洛伊德算法基于的存储结构：

（1）图的存储结构：带权的邻接矩阵存储结构。

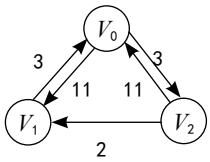
（2）数组 dist[n][n]：存放在迭代过程中求得的最短路径长度。迭代公式为

$$\text{dist}_{-1}[i][j]=\text{arc}[i][j] \qquad (6\text{-}3)$$

$$\text{dist}_k[i][j]=\min\{\text{dist}_{k-1}[i][j],\ \text{dist}_{k-1}[i][k]+\text{dist}_{k-1}[k][j]\} \quad 0 \leqslant k \leqslant n-1$$

（3）数组 path[n][n]：存放从 v_i 到 v_j 的最短路径，初始为 path[i][j]="v_iv_j"。

图 6-26 所示为一个有向网及其邻接矩阵。图 6-27 给出了用弗洛伊德算法求该有向图中每对顶点之间的最短路径过程中，数组 dist 和数组 path 的变化情况。



$$\begin{bmatrix} \infty & 11 & 4 \\ 3 & \infty & \infty \\ 6 & 2 & \infty \end{bmatrix}$$

图 6-26　一个有向网图及其邻接矩阵

$$\text{dist}_{-1}=\begin{bmatrix} \infty & 11 & 4 \\ 3 & \infty & \infty \\ 6 & 2 & \infty \end{bmatrix}\ \text{dist}_0=\begin{bmatrix} \infty & 11 & 4 \\ 3 & \infty & 7 \\ 6 & 2 & \infty \end{bmatrix}\ \text{dist}_1=\begin{bmatrix} \infty & 6 & 4 \\ 3 & \infty & 7 \\ 6 & 2 & \infty \end{bmatrix}\ \text{dist}_2=\begin{bmatrix} \infty & 6 & 4 \\ 3 & \infty & 7 \\ 5 & 2 & \infty \end{bmatrix}$$

$$\text{path}_{-1}=\begin{bmatrix} \square & V_0V_1 & V_0V_2 \\ V_1V_0 & \square & \square \\ V_2V_1 & V_2V_1 & \square \end{bmatrix}\ \text{path}_0=\begin{bmatrix} \square & V_0V_1 & V_0V_2 \\ V_1V_0 & \square & V_1V_0V_2 \\ V_2V_0 & V_2V_1 & \square \end{bmatrix}$$

$$\text{path}_1=\begin{bmatrix} \square & V_0V_2V_1 & V_0V_2 \\ V_1V_0 & \square & V_1V_0V_2 \\ V_2V_1 & V_2V_1 & \square \end{bmatrix}\ \text{path}_2=\begin{bmatrix} \square & V_0V_2V_1 & V_0V_2 \\ V_1V_0 & \square & V_1V_0V_2 \\ V_2V_1V_0 & V_2V_1 & \square \end{bmatrix}$$

图 6-27　Floyed 算法执行中数组 dist 和数组 path 的变化

用 Floyed 算法，求每一对顶点之间的最短路径的完整程序如下：

```
#include<stdio.h>
#include<malloc.h>
#define MAX_VEX 50
int CreatCost(cost)
int cost[][MAX_VEX];                /*cost 表示图的邻接矩阵 */
{
    int vexnum, arcnum, v1, v2, w;
    printf(" 请输入图的顶点数和弧数或边数 :\n");
```

```
        scanf("%d,%d", &vexnum, &arcnum);
        for (int i = 0;i<vexnum;i++)
            for (int j = 0;j<vexnum;j++)
                cost[i][j] = 9999;          /* 本例设 9999 代表无限大 */
        for (int k = 0;k<arcnum;k++)
        {
            printf("v1,v2,w = ");
            scanf("%d,%d,%d", &v1, &v2, &w);  /* 输入所有弧（或边）的一对顶点 V1,V2 */
            cost[v1][v2] = w;
        }
        return vexnum;                      /* 返回图的顶点数 */
    }

    int p[MAX_VEX][MAX_VEX];        /* 定义存放路径的数组 P*/
    void Floyed(cost, vexnum)
    int cost[][MAX_VEX], vexnum;    /*Floyed 算法求每一对顶点之间的最短路径 */
    {
        int a[MAX_VEX][MAX_VEX];
        for (int i = 0;i<vexnum;i++)
            for (int j = 0;j<vexnum;j++)
            {
                a[i][j] = cost[i][j];       /* 给 A 和 P 数组赋初值 */
                p[i][j] = -1;
            }
        for (int i = 0;i<vexnum;i++)    /* 同一顶点间的最短路径为零 */
            a[i][i] = 0;
        for (int k = 0;k<vexnum;k++) /* 通过递推求最短路径长度和路径 */
        {
            for (int i = 0;i<vexnum;i++)
                for (int j = 0;j<vexnum;j++)
                    if (a[i][k] + a[k][j]<a[i][j])
                    {
                        a[i][j] = a[i][k] + a[k][j];
                        p[i][j] = k;
                    }
```

```
        }
        printf(" 每对顶点间的最短路径：\n");
        for (int i = 0;i<vexnum;i++)
        {
            for (int j = 0;j<vexnum;j++)
                printf("%d  ", a[i][j]);
            printf("\n");
        }
        printf(" 每一对顶点间的最短路径上的各个点：\n");
        for (int i = 0;i<vexnum;i++)
            for (int j = 0;j<vexnum;j++)
            {
                printf("%d-->", i);
                PutPath(i, j);
                printf("%d \n", j);
            }
    }

    PutPath(int i, int j)              /* 输出一对顶点间的最短路径上的各个点 */
    {
        int k;
        k = p[i][j];
        if (k == -1)
            return;
        PutPath(i, k);
        printf("%d-->", k);
        PutPath(k, j);
    }

    main()
    {
        int cost[MAX_VEX][MAX_VEX];
        int vexnum = CreatCost(cost);        /* 建立图的邻接矩阵 */
        Floyed(cost, vexnum);                /* 调用算法求每一对顶点间的最短路径 */
    }
```

6.4.3 AOV 网与拓扑排序

1. AOE 网

在每一个工程中，可以将工程分为若干个子工程，这些子工程称为活动。如果用图中的顶点表示活动，以有向图的弧表示活动之间的优先关系，这样的有向图称为 AOV 网，即顶点表示活动的网。在 AOV 网中，如果从顶点 v_i 到顶点 v_j 之间存在一条路径 $<v_i, v_j>$，则顶点 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。活动中的制约关系可以通过 AOV 网来表示。例如图 6-28~ 图 6-29 代表的计算机专业课程，学习就是一个工程，每门课程的学习就是整个工程中的一个活动。

课程编号	课程名称	先修课程
C_1	计算机数学	
C_2	程序设计	
C_3	离散数学	C_1, C_2
C_4	数据结构	C_2, C_3
C_5	算法分析	C_2
C_6	编译技术	C_4, C_5
C_7	操作系统	C_4, C_9
C_8	物理	C_1
C_9	计算机原理	C_8

图 6-28 课程以及课程之间的优先关系

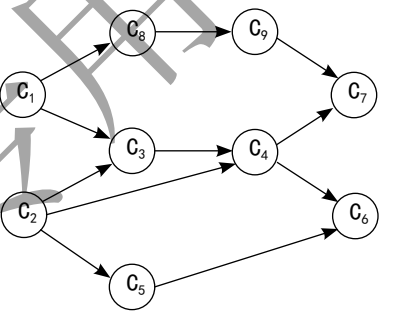


图 6-29 学生选课工程图 AOE 网

我们可以用上图的有向图来表示课程之间的先修关系。在这种有向图中，顶点表示课程学习活动，有向边表示课程之间的先修关系。例如顶点 C_1 到 C_8 有一条有向边，表示课程 C_1 必须在课程 C_8 之前先学习完。

在 AOV 网中，不允许出现环，如果出现环就表示某个活动是自己的先决条件。因此需要对 AOV 网判断是否存在环，可以利用有向图的拓扑排序进行判断。

2. 拓扑排序

拓扑排序就是将AOV网中的所有顶点排列成一个线性序列，并且序列满足以下条件：在 AOV 网中，如果从顶点 v_i 到 v_j 存在一条路径，则在该线性序列中，顶点 v_i 一定出现在顶点 v_j 之前。因此拓扑排序的过程就是将 AOV 网中的各个活动组成一个可行的实施方案。

对 AOV 网进行拓扑排序的基本思想：

（1）在 AOV 网中任意选择一个没有前驱的顶点，即顶点入度为零，将该顶

点输出。

（2）从 AOV 网中删除该顶点，并删除从该顶点出发的弧。

（3）重复执行（1）和（2），直达 AOV 网中所有都已经被输出，或者 AOV 网中不存在无前驱的顶点为止。

按照上述的基本思想，可以得到图 6-29 AOV 网的拓扑序列为 (C_1，C_8，C_9，C_2，C_3，C_5，C_4，C_6，C_7)、(C_1，C_8，C_9，C_2，C_5，C_3，C_4，C_7，C_6）或 (C_2，C_5，C_3，C_8，C_4，C_5，C_6，C_9，C_7) 等。由此可见一个 AOV 网的拓扑序列可能不唯一。

在对 AOV 网进行拓扑排序结束后，可能会出现两种情况：一种是 AOV 网中的顶点全部输出，表示网中不存在回路；另一种是 AOV 网中还存在没有输出的顶点，剩余的未输出顶点的入度都不为零，表示网中存在回路。

6.4.4　AOE 网与关键路径

AOE 网是以边表示活动的有向无环网，在 AOE 网中，具有最大路径长度的路径称为关键路径，关键路径表示完成工程的最短工期。

1. AOE 网

AOE 网是一个带权的有向无环图。其中用顶点表示事件，弧表示活动，权值表示两个活动持续的时间。AOE 网是以边表示活动的网。

AOV 网描述了活动之间的优先关系，可以认为是一个定性的研究，但是有时还需要定量地研究工程的进度，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。在 AOE 网中，通过研究事件和活动之间的关系，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。

在用 AOE 网表示一个工程计划时，用顶点表示各个事件，弧表示子工程的活动，权值表示子工程的活动需要的时间。在顶点表示事件发生之后，从该顶点出发的有向弧所表示的活动才能开始。在进入某个顶点的有向弧所表示的活动完成之后，该顶点表示的事件才能发生。

一个工程只有一个开始状态和一个结束状态。因此在 AOE 网中，只有一个入度为零的点表示工程的开始，称为源点；只有一个出度为零的点表示工程的结束，称为汇点。

AOE 具有以下性质：

（1）只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始。

（2）只有在进入某顶点的各活动都结束，该顶点所代表的事件才能发生。

图 6-30 所示为一个具有 10 个活动、8 个事件的 AOE 网：顶点 V_1，V_2，…，V_8 分别表示一个事件；弧 <V_1，V_2>，<V_1，V_3>，…，<V_7，V_8> 分别表示一个活动，用 a_1，a_2，…，a_{10} 代表这些活动。其中 V_1 为源点，是整个工程的开始，其入度为 0；V_8 为汇点，是整个工程的结束点，其出度为 0。

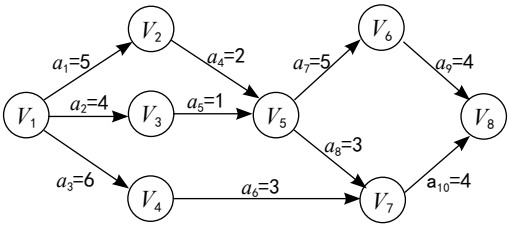


图 6-30 一个 AOE 网

2. 关键路径

关键路径是指在 AOE 网中从源点到汇点路径最长的路径。这里的路径长度是指路径上各个活动持续时间之和。关键路径上的活动称为关键活动。在 AOE 网中，有些活动是可以并行执行的，故完成整个工程所必须花费的时间应该为源点到汇点的最大路径长度。关键路径长度是整个工程所需的最短工期。

利用 AOE 网进行工程管理时需要解决的主要问题：

（1）计算完成整个工程的最短工期。

（2）确定关键路径，以找出哪些活动是影响工程进度的关键。

为了在 AOE 网 $G=(V, E)$ 中找出关键路径，需要定义以下几个参量：

（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[1]=0 开始，按照拓扑排序规则根据递推得到：

$$\text{ve}[i]=\text{Max}\{\text{ve}[i]+(<k, j>\in)|<k, j>T,\ 1\leqslant i\leqslant n-1\} \qquad (6\text{-}4)$$

其中 T 是所有以第 i 个顶点为弧头的弧的集合，dut(<k，j>) 表示弧 <k，j> 对应的活动的持续时间。

（2）事件 v_i 的最晚发生时间 vl[i]：在保证整个工程完成的前提下，活动最迟的开始时间，记作 vl[i]。z 求解 v_i 的最早发生时间 ve[i] 的前提 vl[n]=ve[n] 下，从汇点开始，向源点推进得到：

$$\text{vl}[i]=\text{Min}\{\text{vl}[k]-\text{dut}(<i, k>)|<i, k>\in S,\ 0\leqslant i\leqslant n-2\} \qquad (6\text{-}5)$$

其中 S 是所有以第 i 个顶点为弧尾的弧的集合，dut(<i，k>) 表示弧 <i，k> 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 <vk，vj> 表示活动 a_i 才开始。因

此事件 *vk* 的最早发生时间也就是活动 a_i 的最早开始时间，即

$$e[i]=ve[k] \tag{6-6}$$

（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 最迟必须开始的事件。如果弧 <*vk*，*vj*> 表示活动 a_i，持续时间为 dut(<*k*，*j*>)，则活动 a_i 的最晚开始时间为

$$l[i]=vl[j]-dut(<k，j>) \tag{6-7}$$

（5）活动 a_i 的松弛时间 l[*i*]-e[*i*]：活动 a_i 的最晚开始时间域最早开始时间之差就是活动 a_i 的松弛时间，记作：l[*i*]-e[*i*]。

当 e[*i*]=l[*i*] 时，对应的活动 a_i 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。

关键路径经过的顶点是满足条件 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，关键活动者的弧满足条件 e[*i*]=l[*i*]，即当活动的最早开始时间域最晚开始时间相等时，该活动一定是关键活动。因此，要求关键路径，需要首先求出网中每个顶点的对应事件的最早开始时间，然后推出事件的最晚开始时间和活动的最早、最晚开始时间，最后再判断顶点是否在关键路径之上，得到 AOE 网的关键路径。如图 6-31 虚线所示，给出了如图 6-30 所示 AOE 网的关键路径。

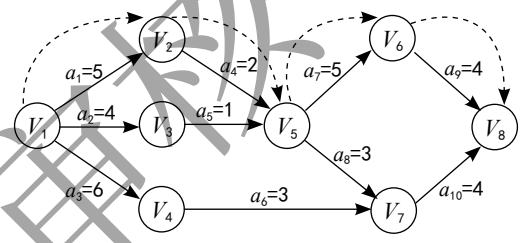


图 6-31 AOE 网的关键路径

图 6-32 中给出了图 6-31 中 AOE 网的关键路径的求解过程。

顶点	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8
ve[*i*]	0	5	4	6	7	12	10	16
vl[*i*]	0	5	6	8	7	12	11	16

活动	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
e[*i*]	0	0	0	5	4	6	7	7	12	10
l[*i*]	0	2	2	5	6	8	7	8	12	11
l[*i*]-e[*i*]	0	2	2	0	2	2	0	6	0	1

图 6-32　AOE 网的关键路径的求解

6.5 项 目 训 练

项目：电网建设造价模拟系统

【问题描述】

假设一个城市有 n 个小区，要实现 n 个小区之间的电网都能够相互接通，构造这个城市 n 个小区之间的电网，使总工程造价最低。请设计一个能满足要求的造价方案。

【设计思路】

在每个小区之间都可以设置一条电网线路，相应的都要付出一点经济代价。n 个小区之间最多可以有 $n(n-1)/2$ 条线路，选择其中的 $n-1$ 条使总的耗费最少。可以用连通网来表示 n 个城市之间以及 n 个城市之间可能设置的电网线路，其中网的顶点表示小区，边表示两个小区之间的线路，赋予边的权值表示相应的代价。对于 n 个顶点的连通网可以建立许多不同的生成树，每一颗生成树都可以是一个电路网。现在，我们要选择总耗费最少的生成树，就是构造连通网的最小代价生成树的问题，一颗生成树的代价就是树上各边的代价之和。

设 $G=(V, E)$ 是具有 n 个顶点的网络，$T=(U, TE)$ 为 G 的最小生成树，U 是 T 的顶点集合，TE 是 T 的边集合。Prim 算法的基本思想是：首先从集合 V 中任取一顶点（例如去顶点 v_0）放入集合 U 中，这时 $U=\{v_0\}$，TE=NULL。然后找出所有一个顶点在集合 U 里，另一个顶点在集合 $V-U$ 里的边，使权 $(u, v)(u \in U, v \in V-U)$ 最小，将该边放入 TE，并将顶点 v 加入集合 U。重复上述操作直到 $U=V$ 为止。这时 TE 中有 $n-1$ 条边，$T=(U, TE)$ 就是 G 的一颗最小生成树。

【数据结构】

假设图采用邻接矩阵表示法表示，用一对顶点的下标（在顶点表中的下标）表示一条边，定义如下：

```
typedef struct{
int start_vex， stop vex; // 边的起点和终点
AdjType weight; // 边的权
}Edge;
```

在构造最小生成树的过程中定义一个类型为 Edge 的数组 mst： Edge

mst[n-1]；其中，n 为网络中顶点的个数，算法结束时，mst 中存放求出的最小生成树的 n-1 条边。

可以用带权的无向图（即无向网）表示这 n 个小区之间的电网连接，其中顶点表示小区，权值表示城市之间电网建设的造价，构造一个无向网的最小生成树即是满足要求的最低电网连接造价方案。

本章小结

（1）图（Graph）是一种非线性数据结构，图中的每个元素既可有多个直接前驱，也可有多个直接后继。图分为有向图和无向图，有向图中的边（又称弧）是顶点的有序对，无向图中的边是顶点的无序对。

（2）图的存储结构常用的有邻接矩阵、邻接表和边集数组三种。图的邻接矩阵具有唯一性，而邻接表和边集数组不具有唯一性。

（3）图的遍历分为深度优先搜索和广度优先搜索。两种算法的具体实现依赖图的存储结构，在学习中应注意图的遍历算法与树的遍历算法之间的类似和差异。

（4）图的应用涉及面广泛，主要有：最小生成树（求最小生成树的两种方法：Prim 方法和 Kruskal 方法）、Dijkstra 和 Floyed 两类求最短路径问题的方法、拓扑排序以及求关键路径的基本方法。

习 题

一、选择题

1. 一个有 n 个顶点的连通无向图至少有（　　）条边。

A. n-1　　B. n　　C. n+1　　D. n+2

2. 具有 n 个顶点的完全有向图的弧数为（　　）。

A. $n(n-1)/2$　　B. $n(n-1)$　　C. n^2　　D. n^2-1

3. 下列算法中，（　　）算法用来求图中某顶点到其他所有顶点之间的最短路径。

A. Dijkstra　　B. Floyed　　C. Prim　　D. Kruskal

4. 设有向无环图 G 中的有向边集合 E={<1，2>，<2，3>，<3，4>，<1，4>}，则下列属于该有向图 G 的一种拓扑排序序列的是（　　）。

A. 1，2，3，4　　B. 2，3，4，1　　C. 1，4，2，3　　D. 1，2，4，3

5. 可以判断一个有向图中是否含有回路的方法为（　　）。

A. 广度优先遍历　　B. 深度优先遍历

C. 拓扑排序　　D. 求最短路径

二、填空题

1. 一个连通无向图有 5 个顶点 8 条边，则其生成树将要去掉_____条边。

2. 在树结构和图结构中，前驱和后继节点之间分别存在着_____和_____的联系。

3. 有 n 个顶点的连通图至少有_____条边；有 n 个顶点的强连通图则至少有_____条边。

4. 一个具有 n 个顶点的有向图至少有_____条弧。

5. 如果不知道一个图是有向图还是无向图，但是知道它的邻接矩阵是非对称的，那么这个图必定是_____。

6. 图的遍历方式一般有_____和_____两种。

7. Prim 算法的时间复杂度为_____，与边数无关，因此适用于求边稠密的网的最小生成树。

三、综合题

1. 画出无向图 6-33 的邻接矩阵和邻接链表示意图，并写出每个顶点的度。

2. 画出有向图 6-34 的邻接矩阵、邻接链表和逆邻接链表示意图，并写出每个顶点的入度、出度。

3. 对应图 6-35，写出从 v_1 出发的深度优先查找遍历结果和广度优先查找遍历结果各 3 个。

4. 写出图 6-36 的拓扑排序。

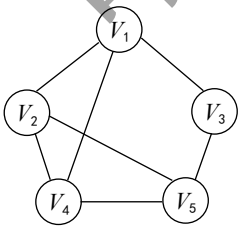


图 6-33 无向图

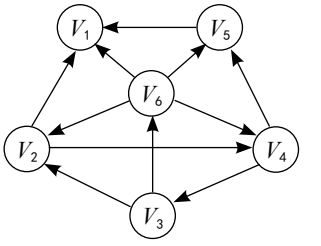


图 6-34 有向图

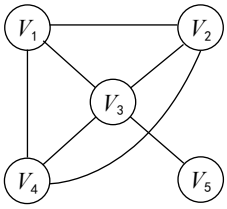


图 6-35 计算机找遍历结果

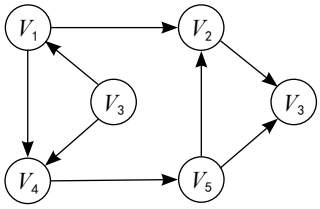


图 6-36 拓扑排序

第7章　查找技术

在日常生活中，查找是一个很普通的事情，人们几乎天天都要进行查找工作，如根据账单查找账单，在通信录中查找某人的信息，在字典上查找单词等。在计算机科学中，查找是数据处理领域中使用最频繁的一种基本操作，如数据库系统的信息维护、编译器对程序中变量名的管理等。

知识目标

- 理解查找和平均查找长度的概念。
- 掌握线性表的查找技术。
- 掌握树表的查找技术。
- 掌握散列表查找技术。

能力目标

- 能熟练运用的查找技术理论知识，解决实际的问题。

7.1 查找的基本概述

7.1.1 概念

在查找领域中，通常将数据元素称为记录（Record），其中涉及的基本概念有：

1. 关键码（Key）

可以标识一个记录的某个数据项称为关键码。关键码的值称为键值（Keyword）。若关键码可以唯一地标识一个记录的关键码，则称为主关键（Primary key）；不能唯一地标识一个记录的关键码，则称为次关键码（Second key）。

2. 查找表（Search Table）

具有相同类型的记录构成的集合即为查找表。由于“集合”中的数据元素之间存在着完全松散的关系，因此查找表是一种非常灵活的数据结构。

3. 查找（Search）

查找是在具有相同类型的记录构成的集合中找出满足给定条件的记录。给定的查找条件可能是多样化的，但为了便于讨论，我们把查找条件限制为“匹配”，即查找技术关键码等于给定值的记录。

4. 查找结果（Search Results）

若在查找表中找到了与给定值相匹配的记录，则称查找成功；否则，称查找失败。一般情况下，查找成功时，查找的结果为给出整个记录的信息，或指示该记录在查找表中的位置。查找不成功时，查找的结果应给出一个“空”记录或“空指针”，或将被查找的记录插入到查找表中。

5. 静态查找（Static Search）

仅能对查找表进行查操作，不涉及插入和删除操作的查找。静态查找适用于查找集合一经生成，便只对其进行查找，而不进行插入和删除操作，或经过一段时间的查找之后，集中地进行插入和删除等修改操作。

6. 动态查找（Dynamic Search）

涉及插入和删除操作的查找。动态查找适用于查找与插入和删除操作在同一阶段进行，例如当查找成功时，要删除找到的记录，当查找不成功时，要插入被查找的记录。

在计算机中进行查找的方法因数据结构的不同而不同。考虑采用的数据结构和查找方法，以提高查找速度，同时查找的虑到节省空间问题。本章讨论的查找结构和查找方法有以下 3 种。

（1）线性表。它适用静态查找，主要采用顺序查找技术、折半查找技术和分块查找技术。

（2）树表。适用于动态查找，主要采用二叉排序树的查找技术。

（3）散列表。静态查找和发展动态查找均适用，主要采用散列技术。

7.1.2 查找算法的性能

为了衡量查找算法的效率，需要在时间和空间两方面进行权衡。查找算法中的基本操作通常是将记录的关键码和给定值进行比较，其运行时间主要消耗在关键码的比较上。因此，应以关键码的比较次数来度量查找算法的时间性能。

通常把查找过程中为确定数据元素（或记录）在表中的位置所进行的关键字比较次数的期望值称为平均查找长度（Average Search Length，ASL），它是衡量查找算法优劣的时间标准。对一个含 n 个记录的表，查找成功情况下，其计算公式为

$$\mathrm{ASL}=\sum_{i=1}^{n}P_i * c_i \qquad (7\text{-}1)$$

式中，n 为问题规模，查找集合中的记录个数；P_i 为查找第 i 个记录的概率；c_i 为查找第 i 个记录所需要的关键码的比较次数。

显然，c_i 与算法密切相关，决定于算法；P_i 与算法无关，决定于具体应用。如果 P_i 是已知的，则平均查找长度 ASL 只是问题规模 n 的函数。

对于查找不成功的情况，平均查找长度即为查找失败对应的关键码的比较次数。查找算法总的平均查找长度应为查找成功与查找失败两种情况下的查找长度的平均。但在实际应用中，查找成功的可能性比查找不成功的可能性大得多，特别是在查找集合中的记录个数很多时，查找不成功的概率可以忽略不计。

7.2 线性表的查找技术

线性表中进行的查找通常属于静态查找，这种查找算法简单，主要适应于对小型查找集合的查找。线性表一般有顺序表链表两种存储结构，此时，可以采用顺序查找技术。

7.2.1 顺序查找

顺序查找（Sequential Search）又称为线性查找，是最基本的查找方法之一。其查找基本思想为：从线性表的一端向另一端逐个将关键码与给定值进行比较，若相等，则查找成功，返回该记录在表中的位置；若整个查找表检测完仍未找到与给定值相等的关键码，则查找失败，给出失败信息。如图 7-1 所示。

【例 7-1】查找 k=15。

图 7-1 顺序查找示意图

顺序查找的程序如下：

```
#include<stdio.h>
#define MAX 50              /* 定义一常量 MAX 为 50，为顺序表的大小 */

#define KeyType int
struct ElemType             /* 顺序表的定义 */
{
    KeyType key;            /* 节点关键字域 */
                            /* 若有其他数据，继续定义，这里假设只有 key 域 */
};
typedef struct ElemType SQList[MAX];

// 顺序查找算法
int SequentialSearch(list, k, n)            /* 在顺序表 list 中查找关键字为 k
```

```
的记录 */
    SQList list;
    int k;
    int n;                          /*n 为线性表 r 中的元素个数 */
    {
        int i = n;
        list[0].key = k;
        while (list[i].key != k   )                 /* 第 i 个元素是 k 吗 */
            i--;
        return i;
    }

    main()
    {
        SQList list = { 0,10,7,24,12,15,35,8,55 };      /* list 有 8 个元素，下标 0
中无元素 */
        int k = 15;
        int i = SequentialSearch(list, k, 8);       /* 找 k=15 的元素 */
        if (i != 0)
            printf("%d 的下标是 %d\n", k, i);
        else
            printf(" 无此元素 \n");
    }
```

由例题可以看出，存储下标为 0 的单元存放待查的记录，记录从下标为 1 的单元开始存放。然后从 n 开始倒着查，当某个 $r[i]$.key=k 时，表示查找成功，自然退出循环。若一直查不到，则直到 i=0，此时 r[0].key 的作用是保证 while 循环一定能够终止，不需要再循环终止条件中写入“i>0”就能免去每一步查找过程中都要检测整个表是否查找完毕，r[0].key 起到了哨兵的作用。所以在循环中不必控制下标 i 是否越界，这就使得运算的量大约减少一半。此查找函数结束时，根据返回的 i 值即可知查找结果。若 i 值大于 0，则查找成功，且 i 值即为找到的记录的位置。若 i 值等于 0，则表示查找不成功。

本算法中对于具有 n 个记录的顺序表，查找第 i 个记录时，需要进行 $n-i+1$ 次关键码的比较，即 $c_i = n-i+1$。在查找成功时，顺序查找的平均查找长度为

$$\mathrm{ASL} = \sum_{i=1}^{n} P_i * c_i = \sum_{i=1}^{n} P_i(n-i+1) \tag{7-2}$$

设每个记录的查找概率相等，即

$$p_i = \frac{1}{n} \tag{7-3}$$

则等概率情况下，有

$$\mathrm{ASL} = \sum_{i=1}^{n} \frac{1}{n}(n-i+1) = \frac{n+1}{2} = 0 \tag{7-4}$$

因此，查找成功时的平均查找长度为 O（n）。

查找不成功时，关键字的比较次数总是 n+1 次，则查找失败的平均查找长度为 O（n）。

顺序查找的优点：算法简单且适用面广，它对表的结构无任何要求。无论记录是否按关键码的大小有序，其算法均可应用，而且上述讨论对线性表的链式存储结构也同样适用，这里就不再详述单链表的顺序查找操作了。

顺序查找的缺点：平均查找长度较大，特别是当 n 很大时，查找效率较低。为了提高查找效率，查找表需依据“查找概率越高，比较次数越少，查找概率越低，比较次数越多”的原则来存储记录。

7.2.2 折半查找

折半查找（Binary Search）也叫二分查找。相对于顺序查找技术来说，折半查找技术的要求比较高，它要求线性表中的记录必须按关键码有序，并且必须采用顺序存储。折半查找技术一般只能应用于静态查找。

利用记录按关键码有序的特点，折半查找的基本思想为：在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键码相等，则查找成功；若给定值小于中间记录的关键码，则在中间记录的左半区继续查找；若给定值大于中间记录的关键码，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

【例 7-2】在有序表 {12，18，20，25，29，32，40，52，62，83，90，95，98} 中查找关键码为 25 和 67 的记录。

假设指针 low 和 high 分别指向待查记录范围的下界和上界，指针 mid 指向区间的中间位置，即 mid=（low+high）/2。Low 和 high 的初值分别为 1 和 13，即 [1，…，13] 为待查范围，mid=（1+13）/2=7。

（1）查找关键码为 25 的过程如图 7-2 所示。

（2）查找关键码为 67 的数据元素需经过以上相类似的过程，简单化过程如图 7-3 所示。

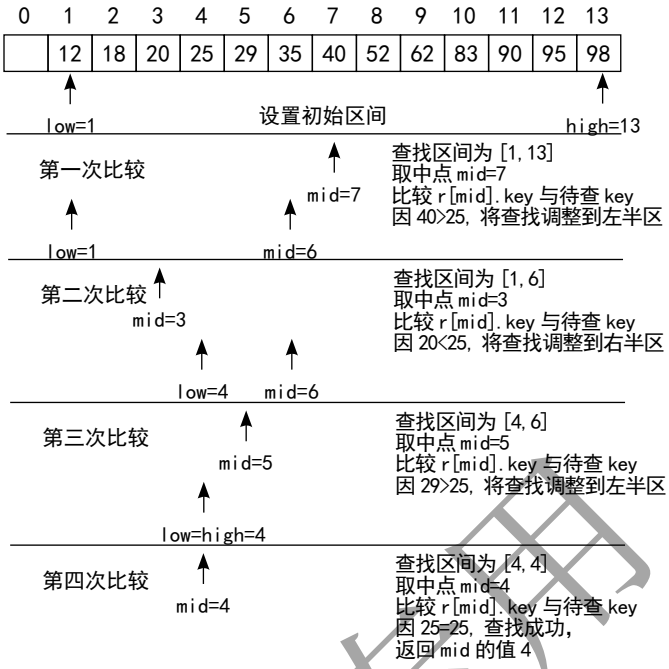


图 7-2　折半查找成功情况下的过程

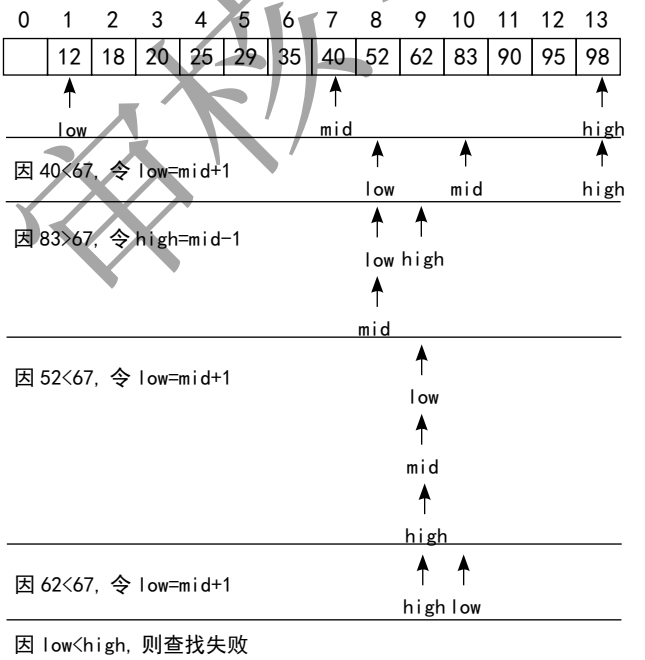


图 7-3　折半查找失败情况下的过程

折半查找的算法如下：

```
#include <stdio.h>
#define KeyType int
typedef struct {
    KeyType key;
}ElemType;

typedef struct {
    ElemType *elem;                 // 存放查找表中数据元素的数组
    int length;                     // 记录查找表中数据的总数量
}SSTable;

// 创建查找表
void CreateTable(SSTable *st, int length) {
    st->elem = (ElemType*)malloc(sizeof(ElemType) * length);
    st->length = length;
    printf(" 请输入表中 %d 个数据元素：\n", length);
    // 存储时，从数组下标为 1 的空间开始存储数据
    for (int i = 1; i <= length; i++) {
        scanf("%d", &(st->elem[i].key));
    }
}

// 折半查找算法
int Binsearch(SSTable *st, KeyType key) {
    int low = 1;                            // 初始状态 low 指针指向第一个关
键字
    int high = st->length;          //high 指向最后一个关键字
    int mid;
    while (low <= high)
    {
        mid = (low + high) / 2;             //int 本身为整形，所以，mid 每次
为取整的整数
        if (st->elem[mid].key == key)       // 如果 mid 指向的同要查找的相
等，返回 mid 所指向的位置
            return mid;
        else if (st->elem[mid].key>key)   // 如果 mid 指向的关键字较大，则
```

```
更新 high 指针的位置
                high = mid - 1;
            else                                    // 反之，则更新 low 指针的位置
                low = mid + 1;
        }
        return 0;
    }

    main(int argc, const char * argv[]) {
        SSTable st;
        CreateTable(&st, 13);
        printf(" 请输入查找数据的关键字：\n");
        int key;
        scanf("%d", &key);
        int location = Binsearch(&st, key);
        // 如果返回值为 0，则证明查找表中未查到 key 值，
        if (location == 0) {
            printf(" 查找表中无该元素 ");
        }
        else {
            printf(" 数据在查找表中的位置为：%d", location);
        }
    }
```

折半查找的运行过程可以用二叉树来描述，这棵树通常称为“判定树”。例如图 7-2 中的静态查找表中做折半查找的过程，对应的判定树如图 7-4 所示。

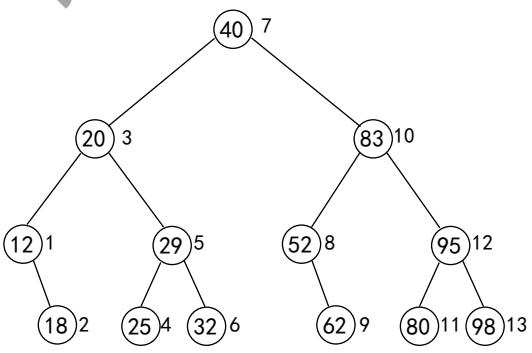


图 7-4　折半查找过程的判定树

在判定树中可以看到，如果想在查找表中查找 25 的位置，只需要进行 4 次

比较，依次和40、20、29、25进行比较，而比较的次数恰好是该关键字所在判定树中的层次（关键25在判定树中的第4层），由此可见，查找表中任意元素的过程恰好是从根节点到该元素节点的路径，与给定值比较的次数恰好是该路径上的节点数或该元素节点在判定树中的层次数。

对于具有 n 个节点（查找表中含有 n 个关键字）的判定树，它的层次数至多为 $\log_2 n+1$（如果结果不是整数，则做取整操作，例如：$\log_2 11+1=3+1=4$）。

同时，在查找表中各个关键字被查找概率相同的情况下，折半查找的平均查找长度为

$$ASL=\log_2(n+1)-1 \tag{7-5}$$

当 n 足够大时，可近似表示为 $\log_2(n)$。在查找速度上，通过比较折半查找的平均查找长度，同前面介绍的顺序查找相对比，明显折半查找的效率要高。但是折半查找算法只适用于有序表，同时仅限于查找表用顺序存储结构表示。

7.2.3 分块查找

分块查找又称索引顺序查找，是折半查找和顺序查找的一种改进方法，折半查找虽然具有很好的性能，但其前提条件时线性表顺序存储而且按照关键码排序，这一前提条件在节点树很大且表元素动态变化时是难以满足的。而顺序查找可以解决表元素动态变化的要求，但查找效率很低。如果既要保持对线性表的查找具有较快的速度，又要能够满足表元素动态变化的要求，则可采用分块查找的方法。

分块查找的速度虽然不如折半查找算法，但比顺序查找算法快得多，同时又不需要对全部节点进行排序。当节点很多且块数很大时，对索引表可以采用折半查找，这样能够进一步提高查找的速度。

分块查找的基本思想：把一个大的线性表分解成若干块，每块中的节点可以任意存放，但块与块之间必须排序。假设是按关键码值非递减的，那么这种块与块之间必须满足已排序要求，实际上就是对于任意的 i，第 i 块中的所有节点的关键码值都必须小于第 $i+1$ 块中的所有节点的关键码值。此外，还要建立一个索引表，把每块中的最大关键码值作为索引表的关键码值，按块的顺序存放到一个辅助数组中，显然这个辅助数组是按关键码值非递减排序的。查找时，首先在索引表中进行查找，确定要找的节点所在的块。由于索引表是排序的，因此，对索引表的查找可以采用顺序查找或折半查找；然后，在相应的块中采用顺序查找，即可找到对应的节点。

【例7-3】如图7-5所示为一个查找表及其索引表，该查找表中含有15个

记录，分成 3 个子表 B_1，B_2，B_3，索引表 A 的每一个元素包含两个字段，一个是该块的最大关键码值，另一个是指向子表的指针。

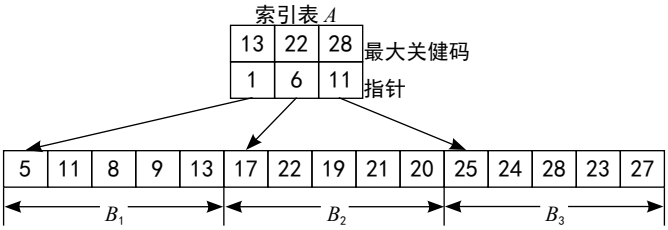


图 7-5　索引表查找示意图

若要查找关键码值为 19 的记录，首先用 19 与索引表中的最大关键码值进行比较，此时即可以用顺序查找技术，也可以用折半查找技术进行查找，如果索引表关键码多，采用折半查找技术，查找效率更快。由于 19<28 且 19>13，如果存在 19 这个记录的话，可确定 19 在子表 B_2 中，然后根据索引表关键码 22 的指针字段指出第二个子表 B_2 的首地址为 6，则从第 6 个记录开始顺序查找，直到在第 8 个记录中找到与给定值相等的关键码，也就是 B_2[8].key=19，查找成功。

分块查找的算法如下：

```
#include <stdio.h>
struct index                        /* 定义块的结构 */
{
    int key;
    int start;
    int end;
} tables[4];                        /* 定义结构体数组 */

int BlockSearch(int key, int a[])   /* 自定义实现分块查找 */
{
    int i, j;
    i = 1;
    while (i <= 3 && key > tables[i].key)          /* 确定在那个块中 */
        i++;
    if (i > 3)                      /* 大于分得的块数，则返回 0*/
        return 0;
    j = tables[i].start;            /*j 等于块范围的起始值 */
    while (j <= tables[i].end && a[j] != key)      /* 在确定的块内进行查找 */
```

```
            j++;
        if (j > tables[i].end) /* 如果大于块范围的结束值，则说明没有要查找的数 ,j
置 0*/
            j = 0;
        return j;
    }

    #define COUNT 16
    main()
    {
        int i, j = 0, key, a[COUNT];
        printf(" 请输入表中 %d 个数据元素：\n", COUNT);
        for (i = 1; i < COUNT; i++)
            scanf("%d", &a[i]);
        for (i = 1; i <= 3; i++)
        {
            tables[i].start = j + 1;        /* 确定每个块范围的起始值 */
            j = j + 1;
            tables[i].end = j + 4;          /* 确定每个块范围的结束值 */
            j = j + 4;
            tables[i].key = a[jj];          /* 确定每个块范围中元素的最大值 */
        }
        printf(" 请输入查找数据的关键字：\n");
        scanf("%d", &key);                  /* 输入要查询的数值 */
        int location = BlockSearch(key, a);             /* 调用函数进行查找 */
        if (location != 0)
            printf(" 数据在查找表中的位置为：%d", location);
        else
            printf(" 查找表中无该元素 ");
    }
```

分析整个过程，我们可以看到索引表按关键码有序排列，所以在索引表查找时既可以用顺序查找，也可以用折半查找；而在子表中查找时，由于块中记录是任意排列的，因此只能使用顺序查找。

假设 n 个记录的查找表均匀地分为 b 块，每块含有 s 个记录，即 $b=[n/s]$；又假设表中每个记录的查找概率是相等的，即每个子表查找的概率为 $1/b$，子表中每个记录的查找概率为 $1/s$。

若索引表查找时采用顺序查找算法，则分块查找的平均查找长度为

$$\mathrm{ASL}=\frac{1}{b}\sum_{j=1}^{b}j+\frac{1}{s}\sum_{i=1}^{s}i=\frac{b+1}{2}=\frac{s+1}{2}=\frac{1}{2}\left(\frac{n}{s}+s\right)+1 \qquad (7\text{-}6)$$

可见，此时的平均查找长度不仅和表长 n 有关，也和每一个子表中的记录个数有关，给定 n 的前提下，s 是可以选择的。容易证明，当 s 取时，ASL 取最小值 +1。

若索引表查找时用折半查找算法，则分块查找的平均查找长度为

$$\mathrm{ASL}\approx\log_2\left(\frac{n}{s}+1\right)+\frac{s}{2} \qquad (7\text{-}7)$$

7.3 树表的查找技术

树表查找是对树型存储结构所做的查找，属于动态查找。树型存储结构是一种多链表，该表中的每个节点包含有一个数据域和多个指针域，每个指针域指向一个后继节点。树型存储结构和树型逻辑结构是完全对应的，都是表示一个树形图，只是用存储结构中的链指针代替逻辑结构中的抽象指针罢了，因此，往往把树型存储结构（即树表）和树型逻辑结构（树）统称为树结构或树。在本节中，将分别讨论在树表上进行查找和修改操作的方法。

7.3.1 二叉排序树

1. 二叉排序树的定义

二叉排序树 (Binary Sort Tree，BST) 又称二叉查找树，是一种特殊的二叉树，它或者是一棵空树，或者是具有下列性质的二叉树：

（1）若它的右子树非空，则右子树上所有节点的值均大于根节点的值。

（2）若它的左子树非空，则左子树上所有节点的值均小于根节点的值。

（3）左、右子树本身又各是一棵二叉排序树。

从上述性质可推出二叉排序树的另一个重要性质：按中序遍历二叉排序树所得到的遍历序列是一个递增有序序列，这也是二叉排序树的名称由来。之前在折半查找中谈论的判定树就是一棵二叉排序树。如图 7-6 所示就是一棵二叉排序树，树中每个节点的关键字都大于它左子树中所有节点的关键字，而小于它的右子树中所有节点的关键字。若对其树进行中序遍历，得到的遍历序列为：24，44，65，70，72，86，92。可见此序列是一个有序序列。

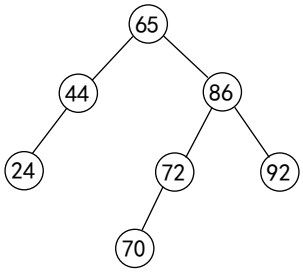


图 7-6 二叉排序树示例

通常以二叉链表作为二叉排序树的存储结构，定义如下：

```
#define KeyType int
typedef struct
{
    KeyType key;
    struct BSTNode *lchild, *rchild;
} BSTNode, *BSTree;
```

2. 二叉排序树的构造

构造二叉排序树也称为二叉排序树的生成，是从空的二叉排序树开始，依次插入一个个节点的过程。

对任意一组关键码集合序列 $\{R_1, R_2, \cdots\cdots, R_n\}$，构造二叉排序树的步骤为：

（1）初始化二叉排序树为一棵空树，读入第一个关键码 R_1 作为这棵二叉排序树的根节点。

（2）若 $R_2 < R_1$，令 R_2 为 R_1 左子树的根节点，否则 R_2 为 R_1 右子树的根节点。

（3）R_3，R_4，……，R_n，节点的插入重复步骤（2）。

【例 7-4】设关键码集合为 {65，85，75，50，68，35，89}，从空树开始建立二叉排序树过程如图 7-7 所示。

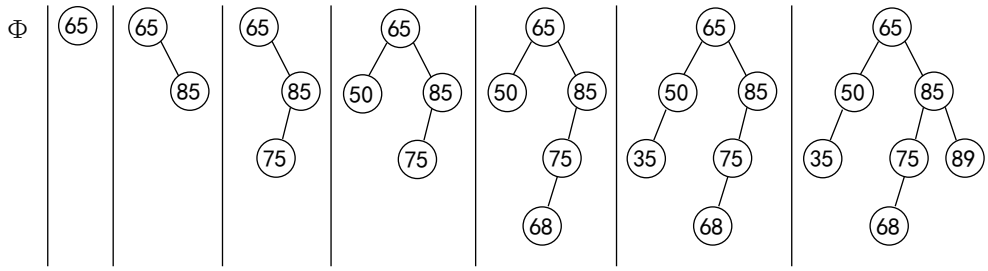


图 7-7 从空树开始构造二叉排序树的过程

3. 二叉排序树的插入

在二叉排序树中插入新节点，要保证插入后仍满足 BST 性质。其插入过程是：

（1）若二叉排序树 T 为空，则为待插入的关键字 kx 申请一个新节点，并令其为根；

（2）若二叉排序树 T 不为空，则将关键字 kx 与根 T 的关键字比较，若二者相等，则说明树中已有此关键字，无须插入，否则：

1）若关键字 kx $<$ $T\rightarrow$data.key，则将关键字 kx 插入根 T 的左子树中。

2）若关键字 kx $>$ $T\rightarrow$data.key，则将关键字 kx 插入根 T 的右子树中。

在子树中的插入过程与上述插入过程相同。如此进行下去，直到将关键字作为一个新的叶节点的关键字插入到二叉排序树中，或者直到发现树中已有此关键字为止。

例如，在如图 7-8(a) 所示的二叉排序树上插入值为的节点后的二排排序树如图 7-8(b) 所示。

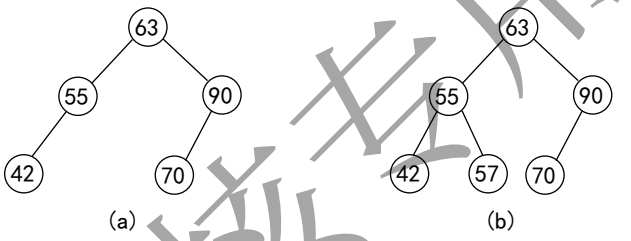


图 7-8 二叉排序树的插入示例

由上述插入过程可以看出，插入的节点是作为叶子插入到二叉排序树中。无论是插在左子树还是右子树中，都是按同样的方法处理的。

4. 二叉排序树的查找

二叉排序树可看成一个有序表，所以在二叉排序树上查找和二分查找类似，也是一个逐步缩小查找范围的过程。根据二叉排序树的定义，查找其关键字等于给定值 kx 的元素的基本思想为：若二叉排序树为空，则表明查找失败，应返回空指针，否则，若给定值 kx 等于根节点的关键字，则表明查找成功，则返回当前根节点指针；若给定值 kx 小于根节点的关键字，则继续在根节点的左子树中查找，若给定值 kx 大于根节点的关键字，则继续在根节点的右子树中查找。

例如，在图 7-6 所示二叉排序树示例中，设树中节点内的数字均为记录的关键码，在其中查找关键码为 72 的记录时，首先需要将 kx=72 与根节点的关键码相比较，由于 72>65，则继续以 65 为根节点的右子树上进行查找，接下来需要将 kx=72 与此右子树的根节点的关键码相比较，72<86，则继续在以 86 为根节

点的左子树上进行查找，且将 kx=72 与左子树的根节点的关键码 72 相比较，恰好与给定值 kx=72 相等，查找成功。又如，在图 7-6 中查找等于关键码 25 的记录和上述过程相似，在给定值 kx=25 分别与关键码 65、44、24 比较后，继续在以 24 为根节点的右子树查找时，此右子树为空，则查找失败。

5. 二叉排序树的删除

二叉排序树的节点删除要比二叉排序树节点的插入要复杂一些，不过也并不难，要分为几种情况进行讨论。二叉排序树节点的插入与删除都是在查找的基础上来做的。下方我们就假设找到了我们要删除的节点，根据节点含有的左右节点的个数来进行分类讨论。

（1）删除节点为叶子节点。删除的节点 P 没有左子树也没有右子树，也就是删除的节点为叶子节点。这种情况下我们只需要将其双亲节点的指针置空即可。如图 7-9 所示。

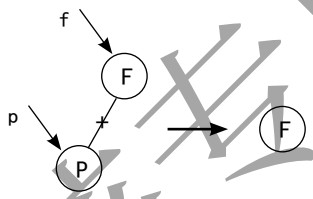


图 7-9 叶子节点的删除

（2）删除的节点 P 只有左子树 P_l 或只有右子树 P_r。只需要将该节点的左子树 P_l 或右子树 P_r 直接成为 P 的双亲节点 F 的左子树（当 P 是左子树）或右子树（当 P 是右子树）即可。如图 7-10 所示。

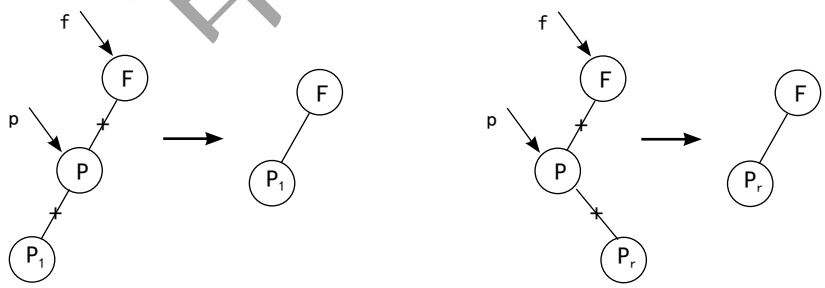


图 7-10 只有左子树或只有右子树的删除

（3）删除的节点既有左子树也有右子树。若待删除节点既有左子树又有右子树的话，相对于前面两种情况来说，稍微复杂些，可按中序遍历保持有序进行调整，如图 7-11 所示，删除 P 节点前，可得到中序遍历序列为：…，A_1，A，…，

B_1，B，C_1，C，P，P_r，F，…，为了保持二叉排序树性质不变，有两种调整方法：①直接令 P_1 为 F 相应的子树，令 P_r 为 P_1 中序遍历的最后一个节点 C（最右下节点）的右子树；②令 P 节点的直接前驱（对 P_1 子树中序遍历的最右下节点 C）替换 P 节点，再按②的方法删去原来的 C。

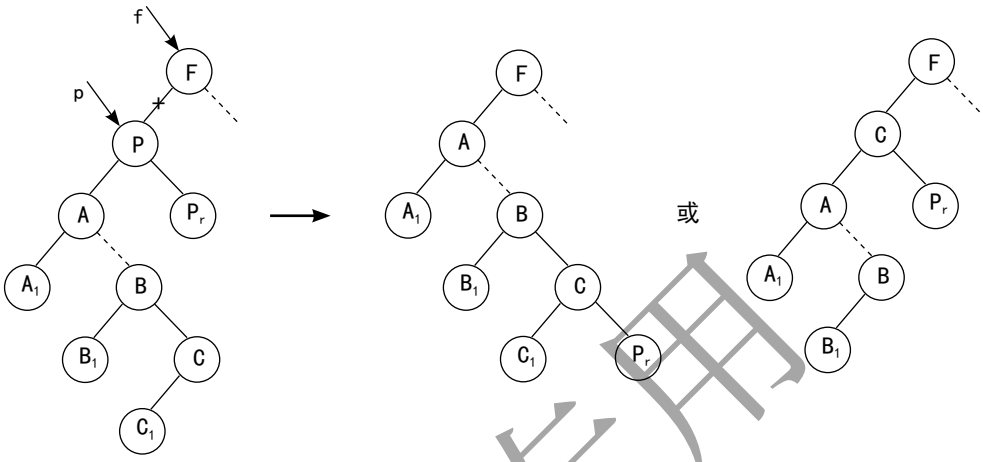


图 7-11 既有左子树也有右子树的删除

二叉排序树的插入、查找和删除操作算法描述如下：

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

#define KeyType int
typedef struct
{
    KeyType key;
    struct BSTNode *lchild, *rchild;
} BSTNode, *BSTree;

// 查找
int SearchBST(BSTree tree, int key, BSTree f, BSTree *p)
{
    if (!tree)
    {
        p = f;
        return 0;
    }
```

```
    else if (key == tree->key)
    {
        p = tree;
        return 1;
    }
    else if (key < tree->key)
        SearchBST(tree->lchild, key, tree, &p);
    else
        SearchBST(tree->rchild, key, tree, &p);
}

// 插入
int InsertBST(BSTree *tree, int key)
{
    if (!(*tree))
    {
        (*tree) = (BSTree)malloc(sizeof(BSTNode));
        (*tree)->key = key;
        (*tree)->lchild = (*tree)->rchild = NULL;
    }
    if (key == (*tree)->key)
        return 0;
    if (key>(*tree)->key)
        InsertBST(&(*tree)->rchild, key);
    else
        InsertBST(&(*tree)->lchild, key);
}

// 中序遍历
void InorderTraverse(BSTree tree)
{
    if (tree)
    {
        InorderTraverse(tree->lchild);
        printf("%d ", tree->key);
        InorderTraverse(tree->rchild);
    }
}
```

```
// 删除
void Delete(BSTree *p)
{
    BSTree q;
    if (!(*p)->lchild && !(*p)->rchild)
        p = NULL;
    else if (!(*p)->lchild) // 左子树为空，重接右子树
    {
        q = *p;
        *p = (*p)->rchild;
        free(q);
    }
    else if (!(*p)->rchild) // 右子树为空，重接左子树
    {
        q = *p;
        *p = (*p)->lchild;
        free(q);
    }
    else  // 左右子树均不为空
    {
        // 查找被删除节点 p 的左子树上的最右下节点 c，并替换 p
        q = *p;
        BSTree c = (*p)->lchild;
        while (c->rchild)
        {
            q = c;
            c = c->rchild;
        }
        (*p)->key = c->key;
        if (q != *p)
            q->rchild = c->lchild;
        else
            q->lchild = c->lchild;
        free(c);
    }
}
```

```
// 删除
int DeleteBST(BSTree *tree, int key)
{
    if (!(*tree))
        return 0;
    else
    {
        if (key == (*tree)->key)
            Delete(&(*tree));
        else if (key < (*tree)->key)
            DeleteBST(&(*tree)->lchild, key);
        else
            DeleteBST(&(*tree)->rchild, key);
    }
}

int main()
{
    int _key, n;
    BSTree tree = NULL, f = NULL, p = NULL;
    printf(" 输入节点数量：");
    scanf("%d", &n);
    printf(" 输入关键字：");
    while (n--)
    {
        scanf("%d", &_key);
        InsertBST(&tree, _key);
    }
    printf(" 中序遍历序列：");
    InorderTraverse(tree);
    printf("\n");
    while (1)
    {
        printf(" 输入要查找元素：");
        scanf("%d", &_key);
        if (SearchBST(tree, _key, f, p))
            printf(" 找到了 \n");
        else
```

```
            printf(" 没找到 \n");

        printf(" 输入要插入元素：");
        scanf("%d", &_key);
        InsertBST(&tree, _key);
        printf(" 中序遍历序列：");
        InorderTraverse(tree);
        printf("\n");

        printf(" 输入要删除元素：");
        scanf("%d", &_key);
        DeleteBST(&tree, _key);
        printf(" 中序遍历序列：");
        InorderTraverse(tree);
        printf("\n");
    }
}
```

7.3.2 平衡二叉树

平衡二叉树（Balanced Binary Tree）又被称为 AVL 树（有别于 AVL 算法），且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。这个方案很好地解决了二叉查找树退化成链表的问题，把插入、查找、删除的时间复杂度最好情况和最坏情况都维持在 $O(\log N)$。但是频繁旋转会使插入和删除牺牲掉 $O(\log N)$ 左右的时间，不过相对二叉查找树来说，时间上稳定了很多。

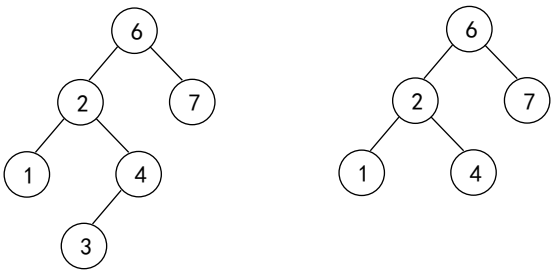


图 7-12 两棵二叉查找树，只有右边的是二叉平衡树

平衡二叉树大部分操作和二叉查找树类似，主要不同在于插入删除的时候平衡二叉树的平衡可能被改变，并且只有从那些插入点到根节点的路径上的节点

的平衡性可能被改变，因为只有这些节点的子树可能变化。本小节不再作详细的讲解。

7.4 散列表的查找技术

散列表查找不同于前面的几种查找方法，它是通过对记录的关键字值进行某种运算，直接求出记录文件的的地址，是关键字到地址的直接转换方法，而不需要反复比较。

7.4.1 散列表的定义

散列亦称哈希 (Hash) 同顺序、链式和索引存储结构一样，是存储线性表的又一种方法。散列存储的基本思想是：以线性表中的每个元素的关键码 key 为自变量，通过一种函数 H(key) 计算出函数值，把这个函数值解释为一块连续存储空间的单元地址 (即下标)，将该元素存储到这个单元中。散列存储中使用的函数 H(key) 称为散列函数或哈希函数，它实现关键码到存储地址的映射 (或称转换)，H(key) 的值称为散列地址或哈希地址；使用的数组空间是线性表进行散列存储的地址空间，所以被称为散列表或哈希表。当在散列表上进行查找时，首先根据给定的关键码 key，用与散列存储时使用的同一散列函数 H(key) 计算出散列地址，然后按此地址从散列表中取对应的元素。

【例 7-5】有个线性表 A=（22，42，54，62，39，81），为了散列存储该线性表，假设选取的散列函数为 H(key)=key%11。

通过这个函数对线性表中的关键码建立的查找表如图 7-13 所示。

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键码	22				81		39	62		42	54

图 7-13 查找表

7.4.2 散列函数的构造方法

构造散列函数的目标是使散列地址尽可能均匀分布在散列空间上，同时使计算尽可能简单。构造散列函数的方法很多，常用的构造散列函数的方法有如下几种。

1. 直接定址法

直接定址法是以关键字 key 本身或关键字加上某个常量 b 作为散列地址的方法。对应的散列函数 $H(\text{key})$ 为

$$H(\text{key}) = a \times \text{key} + b \quad （a，b 为常数）\qquad （7\text{-}8）$$

【例 7-6】关键码集合为 {10，20，30，60，70，80，90}，选取的散列函数为 $H(\text{key}) = \text{key}/10$，则散列表如图 7-14 所示。

散列地址	0	1	2	3	4	5	6	7	8	9
关键码		10	20	30			60	70	80	90

图 7-14　直接定址法数据存储

直接定址法计算简单，并且没有冲突。它适合于关键字的分布基本连续的情况，若关键字分布不连续，空号较多，将会造成较的空间浪费。

2. 除留余数法

除余数法是选择一个适当的 p（p ≤散列表长 m）去除关键字 k，所得余数作为散列地址的方法。对应的散列函数 H（key）为

$$H(\text{key}) = \text{key}\%\ p \qquad （7\text{-}9）$$

其中 p 最好选取小于或等于表长 m 的最大素数。如表长为 20，那么 p 选 19，若表长为 25，则 p 可选 23，…。表长 m 与模 p 的关系可按下表对应：

m=8，16，32，64，128，256，512，1 024，…

p=7，13，31，61，127，251，503，1 019，…

【例 7-7】关键码集合为 {31，62，74，36，49，77}，选取的散列函数为 $H(\text{key}) = \text{key}\%11$，则散列表如图 7-15 所示。

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键码	77			36		49		62	74	31	

图 7-15　除留余数法数据存储

除留余数法一种最简单，也是最常用的一种散列函数构造方法，并且这种方法不要求事先知道关键码的分布。

3. 数字分析法

数字分析法是假设有一组关键字，每个关键字由 n 位数字组成，如 k_1，k_2，…，k_n。数字选择法是从中提取数字分布比较均匀的若干位作为散列地址。

【例 7-8】有一组有 6 位数字组成的关键字，如图 7-16 表示，用数字分析法设计散列函数。

解：分析这一组关键字会发现，第 1，3，5 和 6 位数字分布不均匀，第 1 位数字全是 9 或 8，第 3 位基本上都是 2，第 5，6 两位上也都基本上是 5 和 6，故这 4 位不可取。而第 2，4 两位数字分布比较均匀，因此可取关键字中第 2、4 两位的组合作为散列地址，如图 7-16 的右边一列所示。

关键码						
①	②	③	④	⑤	⑥	散列地址
8	7	2	2	6	5	72
9	5	2	4	5	6	54
9	1	2	3	5	6	13
8	9	2	5	5	6	95
9	6	4	8	5	2	68
9	8	2	1	6	6	81

图 7-16 数字分析法示例

数字分析法适合于事先知道关键码的分布且关键码中有若干分布均匀的情况。

4. 平方取中法

平方取中法是取关键字平方的中间几位作为散列地址的方法，因为一个乘积的中间几位和乘数的每一位都相关，故由此产生的散列地址较为均匀，具体取多少位视实际情况而定。

【例 7-9】有一组关键字集合 (0100，0110，0111，1001，1010，1110)，平方之后得到新的数据集合 (0010000，0012100，0012321，1002001，1020100，123210)，那么，若表长为 1000，则可取其中第 3、4 和 5 位作为对应的散列地址为 (100，121，123，020，201，321)。

5. 折叠法

折叠法是首先把关键字分割成位数相同的几段(最后一段的位数可少一些)，段的位数取决于散列地址的位数，由实际情况而定，然后将它们的叠加和 (舍去最高进位) 作为散列地址的方法。

折叠法又分移位叠加和边界叠加。移位叠加是将各段的最低位对齐，然后相加；边界叠加则是将两个相邻的段沿边界来回折叠，然后对齐相加。

【例 7-10】关键字 key=98123658，散列地址为 3 位，则将关键字从左到右每三位一段进行划分，得到的三个段为 981，236 和 58，叠加后值为 1 275，

取低 3 位 275 作为关键字 98123658 的元素的散列地址；如若用边界叠加，即为 981，632 和 58 叠加后其值为 1671，取低 3 位得 671 作为散列地址。如图 7-17 所示

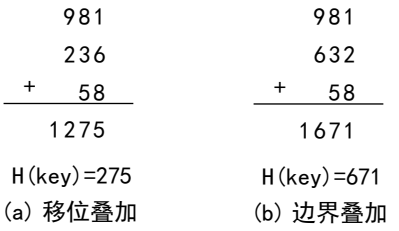


图 7-17 折叠法示例

折叠法适合用于关键码的位数很多，且关键码的每一位分布都不均匀的情况。折叠法事先不需要知道关键码的分布。

7.4.3 处理冲突方法

散列法构造表可通过散列函数的选取来减少冲突，但冲突一般不可避免，为此，需要有解决冲突的方法，常用的解决冲突的方法有三大类，即开放定址法、链地址法和公共溢出区。

1. 开放定址法

开放定址法又分为线性探查法、二次探查法和双重散列法。开放定址法解决冲突的基本思想是：使用某种方法在散列表中形成一个探查序列，沿着次序列逐个单元进行查找，直到找到一个空闲的单元时将新节点存入其中。假设散列表空间为 T[0..m-1]，散列函数 H(key)，开放定址法的一般形式为

$$H_i=(H(\text{key})+d_i)\%m \quad (0 \leqslant i \leqslant m-1)$$

式中，H(key) 为散列函数，m 为散列表长度，d_i 为增量序列。

由 d_i 的取值不同，可以分为以下三种方法：

（1）线性探查法。线性探查法的地址增量 d_i=1，2，…，m-1，其中，i 为探测次数。该方法一次探测下一个地址，知道有空的地址后插入，若整个空间都找不到空余的地址，则产生溢出。

【例 7-11】已知关键码序列为 {49，9，31，13，18，94，24，10，5}，散列函数为 H(key)=key %11，要求用线性探测法处理冲突。

解：H(49)=49%11=5，H(9)=9%11=9，H(13)=13%11=2，H(18)=18%11=7，H(94)=94%11=6，由散列函数得到的散列地址时，没有发生冲突，直接存入，成

功查找时只需要 1 步。

$H(31)=31\%11=9$，地址发生冲突，需寻找下一个空的散列地址。由 $H_1=(H(31)+1)\%11=10$，有空，将 31 存入。成功查找是需要 2 步。

$H(24)=24\%11=2$，地址发生冲突，需寻找下一个空的散列地址。由 $H_1=(H(24)+1)\%11=3$，有空，将 24 存入。成功查找是需要 2 步。

$H(10)=10\%11=10$，地址发生冲突，需寻找下一个空的散列地址。由 $H_1=(H(10)+1)\%11=0$，有空，将 10 存入。成功查找是需要 2 步。

$H(5)=5\%11=5$，地址发生冲突，需寻找下一个空的散列地址。由 $H_1=(H(5)+1)\%11=6$，地址冲突，继续 $H_2=(H(5)+2)\%11=7$，仍然冲突，继续 $H_3=(H(5)+3)\%11=8$，有空，将 5 存入。成功查找是需要 4 步。

根据上面分析可得到由线性探查法的散列表如图 7-18 所示。

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键码	10		13	24		49	94	18	5	9	31
探测次数	2		1	2		1	1	1	4	1	2

图 7-18 线性探查法处理冲突

线性探查法容易产生“聚集”现象。当表中的第 i、i+1、i+2 的位置上已经存储某些关键字，则下一次哈希地址为 i、i+1、i+2、i+3 的关键字都将企图填入到 i+3 的位置上，这种多个哈希地址不同的关键字争夺同一个后继哈希地址的现象称为“聚集”。聚集对查找效率有很大影响。

线性探查法这种处理冲突的方法思路清晰，算法简单。其算法如下：

```
#include<stdio.h>
#define COUNT 11                    /*COUNT 为表长 */
struct ElemType
{
    int key;
};

int LinearSearch(r, ht)             /* 线性探查 */
int r;                              /*r 为待查记录 */
struct ElemType ht[COUNT];          /*ht 为散列表 */
{  int i;
i = r % COUNT;    /* 计算 r 记录的散列地址 , 假设 H(key)=key%COUNT*/
while ((ht[i].key != 0) && (ht[i].key != r)) /*0 表示存储空间是开放的 */
```

```
    i = (i + 1) % COUNT;                /* 探测下一次 ,% 为取余运算 */
    if (ht[i].key == 0)
    ht[i].key = r;                      /* 查找不成功时插入该记录 */
    return(i);
    }

    main()
    {
        int i, j;
        struct ElemType ht[COUNT];
        for (i = 0;i<COUNT;i++)         /* 为关键字字段赋初值 0，表示存储空间是
开放的 */
            ht[i].key = 0;
        printf(" 请输入五个待查找记录 : ");
        for (i = 0;i<5;i++)
        {
            scanf("%d", &j);            /* 输入五个待查找记录 */
            LinearSearch(j, ht);        /* 调用查找函数 */
        }
        printf(" 输出散列表：");
        for (i = 0;i<COUNT;i++)         /* 输出结果 */
        {
            printf("%d,", ht[i].key);
        }
    }
```

（2）二次探查法。二次探测法的地址增量序列为 $d_i = 1^2, -1^2, 2^2, -2^2, \cdots, q^2, -q^2$ ($q <= m/2$)。

【例 7-12】已知关键码序列为 {49，9，31，13，18，94，24，10，5}，散列函数为 H(key)=key%11，要求用二次探查法处理冲突。

解：H(49)=49%11=5，H(9)=9%11=9，H(13)=13%11=2，H(18)=18%11=7，H(94)=94%11=6，由散列函数得到的散列地址时，没有发生冲突，直接存入，成功查找时只需要 1 步。

H(31)=31%11=9，地址发生冲突，需寻找下一个空的散列地址。由 H_1=(H(31)+1^2)%11=10，有空，将 31 存入。成功查找是需要 2 步。

H(24)=24%11=2，地址发生冲突，需寻找下一个空的散列地址。由

$H_1=(H(24)+1^2)\%11=3$，有空，将24存入。成功查找是需要2步。

$H(5)=5\%11=5$，地址发生冲突，需寻找下一个空的散列地址。由$H_1=(H(5)+1^2)\%11=6$，地址冲突，继续$H_2=(H(5)-1^2)\%11=4$，有空，将5存入。成功查找是需要3步。

根据上面分析可得到由二次探查法的散列表如图7-19所示。

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键码	10		13	24	5	49	94	18		9	31
探测次数	2		1	2	3	1	1	1		1	2

图7-19 二次探查法处理冲突

二次探测能有效避免“聚集”现象，即使不能够探测到哈希表上所有的存储单元，至少能够探测到一半。

（3）双重散列法。双重散列法是几种方法中最好的方法，它的探查散列函数为

$$d_0=\mathrm{Hash(key)}$$
$$d_i=d_0+i*\mathrm{ReHash(key))\%}m \quad (i=1,\ 2,\ \cdots,\ m-1) \tag{7-10}$$

其中，Hash(key)和ReHash(key)是两个不同的哈希函数，m为哈希表长度。

先用第一个函数Hash(key)对关键字计算哈希地址，若发生地址冲突，再用第二个函数ReHash(key)计算另一个哈希函数地址，直到冲突不再发生。

例如，$H(\mathrm{key})=a$时产生地址冲突，则计算$\mathrm{Re}H(\mathrm{key})=b$，探测的地址序列为$H_1=(a+b)\%m$，$H_2=(a+2b)\%m$，…，$H_{m-1}=(a+(m-1)b)\%m$。这种方法不易产生“聚集”，但增加了计算时间。

2. 链地址法

链地址法（又叫拉链法）是把具有相同散列地址的关键码（同义词）值放在同一个单链表中，称为同义词链表。有m个散列地址就有m个链表，同时用指针数组$T[0..m-1]$存放各个链表的头指针，凡是散列地址为i的记录都以节点方式插入到以$T[i]$为指针的单链表中。T中各分量的初值应为空指针。

【例7-13】已知关键字序列为{49，9，31，13，18，94，24，10，5}，哈希函数为$H(\mathrm{key})=\mathrm{key}\%11$，用拉链法处理冲突，建表如图7-20所示。

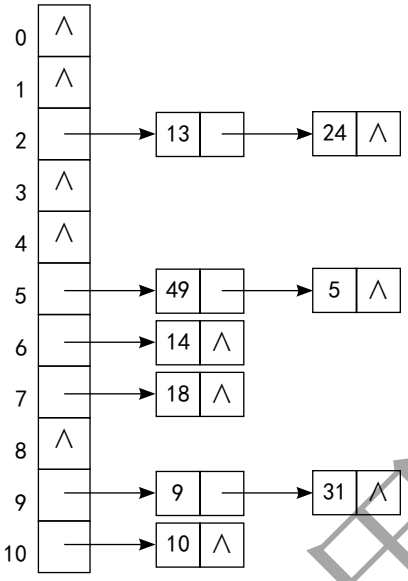


图 7-20　链地址法处理冲突

用链地址法处理冲突简单，无堆积现象。虽然比开放定址法多占用一些存储空间用做链接指针，但它可以减少在插入和查找过程中同关键字平均比较次数（平均查找长度）。

链地址法的查找及插入算法如下：

```
#include<stdio.h>
#define COUNT 11                    /*COUNT 为散列表基本区的长度 */
struct ElemType
{
    int key;
    struct ElemType *next;
};
struct ElemType ht[COUNT];      /* 定义表头向量兼散列表基本区 */

void LinkHash(ht, k)                /* 静态链接法散列算法 */
struct ElemType ht[COUNT];      /*ht 为散列表基本区 */
int k;                  /*k 为待查记录的关键字 */
{
    int i;
    struct ElemType *p, *q, *r;
```

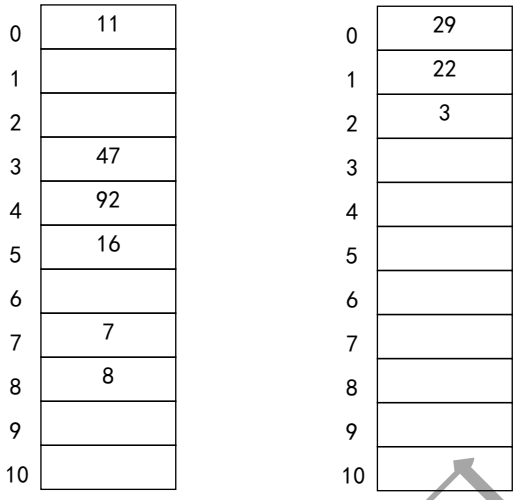
```
        i = k % COUNT;                  /* 确定 k 在基本区的散列地址 */
        if (ht[i].key == 0)             /* 表中没有关键字为 k 的记录 */
            ht[i].key = k;              /* 将关键字为 k 的查找记录 r 插入基本区 */
        else
            if (ht[i].key == k)
                printf(" 查找成功 ! %d,%d", i, k);              /* 查找到显示成功
信息等 */
            else
            {
                r = &ht[i];
                q = ht[i].next;         /* 进入链接区 ( 溢出区 ) 查找 ( 或插入 )*/
                while ((q->key != k) && (q != NULL))
                {
                    r = q;
                    q = q->next;
                }
                if (q != NULL)
                    printf(" 查找成功 ! %d,%d", i, k);          /* 查找到显示成功
信息等 */
                else                    /* 链接区中没有查到 , 插入该记录 */
                {
                    p = (struct ElemType*)malloc(sizeof(struct ElemType));
                    p->key = k;         /* 插入记录 */
                    p->next = NULL;
                    r->next = p;
                }
            }
    }

    main()
    {
        int i, j;
        struct ElemType *p;
        for (i = 0;i<COUNT;i++)             /* 初始化散列表 */
        {
```

```
        ht[i].key = 0;
        ht[i].next = NULL;
    }
    printf(" 请输入 %d 个待散列的数据 : ", COUNT);
    for (i = 1;i < COUNT;i++)                    /* 接收数据 */
    {
        scanf("%d", &j);
        LinkHash(ht, j);                         /* 查找或建立散列表 */
    }
    printf("\n");
    printf(" 输出静态链接法结果 :\n");
    for (i = 0;i<COUNT;i++)                      /* 输出散列表 */
    {
        printf("%d,", ht[i].key);                /* 输出链表头中的值 */
        p = ht[i].next;                          /* 为查找下面的元素做准备 */
        while (p != NULL)
        {
            printf("%d,", p->key);               /* 输出链表中的其他元素 */
            p = p->next;
        }
        printf("\n");
    }
    printf(" 请输入 1 个待散列的数据 : ");
    scanf("%d", &j);
    LinkHash(ht, j);                             /* 查找或建立散列表 */
}
```


3. 公共溢出区

公共溢出区的基本思想：散列表包含基本表和溢出表两部分（通常溢出表和基本表的大小相同），将发生冲突的记录存储在溢出表中。查找时，对给定值通过散列函数计算散列地址，先与基本表的相应单元进行比较，若相等，则查找成功；否则，再到溢出表中进行顺序查找。

【例 7-14】关键码集合 {49，9，31，13，18，94，24，10，5}，散列函数为 H(key)=key mod 11，用公共溢出区法处理冲突，构造的散列表如图 7-21 所示。



如图 7-21 公共溢出区处理冲突

7.4.4 散列查找的性能分析

在散列技术中，处理冲突的方法不同，得到的散列表不同，散列表的查找性能也不同。由于冲突的存在，产生冲突后的查找仍然是给定值与关键码进行比较的过程。所以，对散列表查找效率的量度依然采用平均查找长度。

在查找过程中，关键码的比较次数取决于产生冲突的概率。而影响冲突产生的因素有：

（1）散列函数是否均匀。散列函数是否均匀直接影响冲突产生的概率。一般情况下，任务所选的散列函数是均匀的，因此，可以不考虑散列函数对平均查找长度的影响。

（2）处理冲突的方法。就线性探查法（见例 7-11）、二次探查法（见例 7-12）和链地址法（见例 7-13）处理冲突来看，相同的关键码集合，相同的散列函数，但处理冲突的方法不同，则它们的平均查找长度不同：

线性探查法的平均查找长度 ASL=（5×1+3×2+1×4）/9=15/9

二次探查法的平均查找长度 ASL=（5×1+3×2+1×3）/9=14/9

链地址法的平均查找长度 ASL=（6×1+3×2）/9=12/9

（3）散列表的装填因子。构造散列函数时，为了使范围广泛的关键字域映射到一组指定的连续空间，我们放弃了一一对应的映射关系，引进了冲突，增加了查找时间，由于发生的冲突次数与表的填满程度直接有关，所以引进装填因子 α：

$$\alpha=\begin{cases}\text{表中填入的记录数}\\ \text{散列表的长度}\end{cases} \tag{7-11}$$

α 标志散列表的填满程度。α 越小，已填入的记录越少，发生冲突的机会就越小；α 越大，已填入的记录越多，发生冲突的机会就越大。因此，散列表查找成功的平均查找长度 S_n 和装填因子 α 有关。下面给出几种不同处理冲突方法的平均查找长度。

线性探查法成功查找时的平均查找长度为

$$S_{nl}\approx\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \tag{7-12}$$

查找不成功的平均查找长度为

$$U_{nl}\approx\frac{1}{2}\left(1+\frac{1}{(1+\alpha)^2}\right) \tag{7-13}$$

二次探查法成功查找时的平均查找长度为

$$S_{nr}\approx-\frac{1}{\alpha}\ln(1-\alpha) \tag{7-14}$$

查找不成功的平均查找长度为

$$S_{nr}\approx\frac{1}{1-\alpha} \tag{7-15}$$

链地址法成功查找时的平均查找长度为

$$S_{NC}\approx1+\frac{\alpha}{2} \tag{7-16}$$

查找不成功的平均查找长度为

$$S_{NC}\approx\alpha+\mathrm{e}^{-a} \tag{7-17}$$

由此可见，散列表的平均查找长度是装填因子的函数，而不是查找集合中记录个数 n 的函数，只是不同的处理冲突的方法有不同的函数，无论 n 多大，总可以选择一个合适的装填因子将平均查找长度限定在一个范围内。

散列查找技术存取速度快，也教节省空间，静态查找和动态查找均适用，但由于存取是随机的，因此，不便于顺序查找。

7.5 项目训练

项目：散列表的应用

【问题描述】

针对某个集体（比如你所在的班级）中的“名字”设计一个散列表，使得平均查找长度不超过 R，完成相应的建表和查表程序。

【设计思路】

假设人名为中国人姓名的汉语拼音形式。待填入散列表的人名共有 30 个，取平均查找长度的上限为 2。散列函数用除留余数法构造，用伪随机探测再散列发处理冲突。

【数据结构】

```
typedef struct
{
    char *py;                   // 名字的拼音
    int k;                      // 拼音所对应的整数
}NAME;
typedef struct    // 哈希表
{
    char *py;                   // 名字的拼音
    int k;                      // 拼音所对应的整数
    int si;                     // 查找长度
}HASH;
```


本章小结

（1）查找是数据处理中经常使用的一种重要运算。在许多软件系统中最耗时间的部分是查找。因此，研究高效的查找方法是本章的重点。

（2）线性表的查找主要包括顺序查找、折半查找和分块查找，顺序查找比较慢，但适用面广；速折半查找度快，但必须是有序表；分块查找是二者的折中方法。

（3）树表的查找主要讲述了二叉排序树的查找技术，充分地利用了二叉树或树的特点建立相应的数据结构，然后根据指定的值在二叉树上进行查找。

（4）散列法是利用散列函数的映射关系直接确定记录的位置，大大减少了与关键码比较的次数，效率快。

习　题

一、选择题

1. 请指出在顺序表｛2、5、7、10、14、15、18、23、35、41、52｝中，用折半查找法关键码 12 需要做（　　）次关键码比较。

A. 2　　B.3　　C. 4　　D. 5

2. 对一个排好序的线性表，用折半查找法检索表中的元素，被检索的表应当采用（　　）表示。

A. 顺序存储　　B. 链接存储

C. 散列法存储　　D. 存储表示不受限制

3. 设二叉排序树中有 n 个节点，则在二叉排序树的平均查找长度为（　　）。

A.$O(1)$　　B.$O(\log_2 n)$　　C.$O(n)$　　D.$O(n^2)$

4. 依次插入序列（50，72，43，85，75，20，35，45，65，30）后建立的二叉搜索树中，查找元素 35 要进行（　　）次元素间的比较。

A.4　　B.5　　C.7　　D.10

5. 对包含 N 个元素的散列表进行查找，平均查找长度（　　）。

A. 为 O(log2N)　　B. 为 O（N）

C. 不直接依赖于 N　　D. 上述三者都不是

6. 设散列表中有 m 个存储单元，散列函数 H（key）=key%p，则 p 最好选择（　　）。

A. 小于等于 m 的最大奇数　　B. 小于等于 m 的最大素数

C. 小于等于 m 的最大偶数　　D. 小于等于 m 的最大合数

7.（　　）是 HASH 查找的冲突处理方法。

A. 求舍法　　B. 平方取中法　　C. 折半查找法　D. 开放地址法

8. 如果要求一个线性表既能较快地查找，又能适应动态变化的要求，可以采用（　　）查找方法。

A. 分块　　B. 顺序　　C. 折半　　D. 散列

二、填空题

1. 对于长度为 n 的线性表，若进行顺序查找，则时间复杂度为_____；若采用折半法查找，则时间复杂度为_____。

2. 在分块查找中首先查找_____。然后再查找相应的_____。

3. 当所有节点的权值都相等时，用这些节点构造的二叉排序树_____遍历它们得到的序列的顺序是一样的。

4. 在散列存储中，装填因子 α 的值越大，则_____；α 的值越小，则_____。

三、综合题

1. 已经如下 11 个数据元素的有序表（6，14，19，21，36，57，63，76，81，89，93），请画出查找键值为 21（成功）和 85（失败）的查找过程。

2. 已知一组关键字为 {1，14，27，29，55，68，10，11，23}，则按散列函数 H(key)=key MOD 13 和链地址法处理冲突来构造散列表。

（1）画出所构造的散列表。

（2）在记录的查找概率相等的前提下，计算该表查找成功时的平均查找长度。

第8章 排 序

排序是数据处理中经常使用的一种操作，其主要目的是便于查找数据。在日常生活中，通过排序来提高数据查找性能的例子屡见不鲜，例如将学生的考试成绩按总分或者平均分进行排序；将职工按参加工作的年限进行排序；将网站中的帖子按点击率进行排序等。因此，认真研究和掌握排序算法是十分重要的。

本章主要介绍排序的基本概念、排序的种类、排序的过程及方法。

知识目标

- 了解排序的基本概念。
- 掌握直接插入排序和折半插入排序的算法实现。
- 理解希尔排序的算法思想。
- 掌握冒泡排序和快速排序的算法实现。
- 掌握直接选择排序的算法实现。
- 理解堆排序的算法思想。

能力目标

- 针对不同问题，能够选择适当的排序算法完成数据排序。

8.1　排序的基本概述

8.1.1　基本术语

（1）记录。通常将数据元素称为记录。

（2）排序。排序（Sorting）就是将一组数据元素按某个数据项递增或递减的次序重新排列的过程。

（3）正序和逆序。

正序：待排序序列中的记录已按关键码排好序。

逆序（反序）：待排序序列中记录的排列顺序与排好序的顺序正好相反。

（4）稳定性。假设按记录的次关键字进行排序，并且有次关键字相同的记录，如果经过排序后这些具有相同关键字的记录之间的相对次序保持不变，则称这种排序方法是稳定的；反之，则称这种排序方法是不稳定的。

（5）内排序和外排序。

内排序 (Internal Sorting) 是指在排序过程中，所有记录全部被存放在内存中，排序时不涉及数据的内、外存交换。根据排序过程中所用的策略不同，内排序可以分为插入排序、选择排序、交换排序、归并排序和基数排序五类。

外排序 (External Sorting) 是指由于待排序的记录数量太大，不能全部放置在内存中，只能一部分记录放置在内存中，另一部分记录放置在外存中，整个排序过程中需要在内外存之间多次交换数据才能完成。

（6）单键排序和多键排序。

单键排序：根据一个关键码进行排序。

多键排序：根据多个关键码进行排序。

设关键码分别为 k_1，k_2，…，k_m，多键排序有两种方法：

（1）依次对记录进行 m 次排序，第一次按 k_1 排序，第二次按 k_2 排序，依此类推。这种方法要求各趟排序所用的算法是稳定的；

（2）将关键码 k_1，k_2，…，k_m 分别视为字符串依次首尾连接在一起，形成一个新的字符串，然后，对记录序列按新形成的字符串排序。

不论哪种方法，多键排序都被转化成单键排序。所以，本章只讨论单键排序。

8.1.2 排序算法的性能

排序是数据处理中经常执行的一种操作，往往属于系统核心部分，因此排序算法的时间开销是衡量其好坏的重要指标。对于基于比较的内排序，在记录过程中常常需要进行下列两种操作：关键字之间的比较和记录位置的移动。所以，在待排序的记录个数一定的条件下，算法的执行时间主要消耗在关键字之间的比较和记录的移动上，因此，高效率的排序算法应该具有尽可能少的关键字比较次数和尽可能少的记录移动次数。

评价排序算法的另外一个主要标准是实现算法所需要的辅助存储空间。辅助存储空间是指在数据规模一定的条件下，除了存放待排序记录占用的存储空间之外，执行算法所需要的其他存储空间。

8.2 插入排序

插入排序（Insertion sort）是一种简单直观且稳定的排序算法。如果有一个已经有序的数据序列，要求在这个已经排好的数据序列中插入一个数，但要求插入后此数据序列仍然有序，这个时候就要用到插入排序法。插入排序的基本操作就是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据序列。插入排序算法的时间复杂度为 $O(n^2)$，是稳定的排序方法，适用于少量数据的排序。插入算法把要排序的数组分成两部分：第一部分包含了这个数组的所有元素，并且预留一个空间给插入元素，而第二部分就只包含待插入元素。在第一部分排序完成后，再将这个最后元素插入到已排好序的第一部分中。

根据插入位置的不同，将插入排序分为直接插入排序和折半插入排序。

8.2.1 直接插入排序

1. 基本思想

直接插入排序是一种简单的插入排序法，其基本原理是：把待排序的记录按其关键码值的大小逐个插入到一个已经排好序的有序序列中，直到所有的记录插入完为止，得到一个新的有序序列。

例如，已知待排序的一组记录是：

60，71，49，11，24，3，66

假设在排序过程中，前 3 个记录已按关键码值递增的次序重新排列，构成一个有序序列：

49，60，71

将待排序记录中的第 4 个记录（即 11）插入上述有序序列，以得到一个新的含 4 个记录的有序序列。首先，将 11 放入数组的第一个单元 r[0] 中，这个单元称为监视哨。然后从 71 起从右到左查找，11 小于 71，将 71 右移一个位置，11 小于 60，又将 60 右移一个位置，11 小于 49，再将 49 右移一个位置，这时将 11 与 r[0] 的值比较，11 ⩾ r[0]，它的插入位置就是 r[1]。假设 11 大于第一个值 r[1]。它的插入位置应该在 r[1] 和 r[2] 之间，由于 60 已经右移了，留出来的位置正好留给 11. 后面的记录依照同样的方法逐个插入到该有序序列中。若记录总数为 n 个，需进行 $n-1$ 趟排序，才能完成。

如图 8-1 所示是一个直接插入排序的例子，括号中的元素为排好序的部分。

初始序列	(15)	4	23	14	20	4
第一趟插入	(4	15)	23	14	20	4
第二趟插入	(4	15	23)	14	20	4
第三趟插入	(4	14	15	23)	20	4
第四趟插入	(4	14	15	20	23)	4
第五趟插入	(4	4	14	15	20	23)

图 8-1　直接插入排序举例

2. 直接插入排序算法

直接插入排序的算法思路：

（1）设置监视哨 r[0]，将待插入记录的值赋给 r[0]；

（2）设置开始查找的位置 j；

（3）在数组中进行搜索，搜索中将第 j 个记录后移，直至 r[0].key ⩾ r[j].key 为止；

（4）将 r[0] 插入 r[j+1] 的位置上。

直接插入排序算法如下：

```
#include<stdio.h>
#define LENGTH 7
main()
{
```

```
    /* 定义数组并赋初值，其中 r[0] 为监视哨，初值为 0*/
    int r[LENGTH + 1] = { 0,60,71,49,11,24,3,66 };
    for (int i = 2,j;i <= LENGTH;++i) /* 第一个数是有序的，为初始有序序列，i 从 2 开始 */
      if (r[i] < r[i - 1])        /* 如"<"，需将 r[i] 插入到前面有序序列中 */
      {                           /* 否则 r[i] 不需要插入，保持原来位置 */
        r[0] = r[i];              /*r[i] 的值放入监视哨中 */
        for (j = i - 1;r[0] < r[j];--j)
          r[j + 1] = r[j];        /* 记录后移 */
        r[j + 1] = r[0];          /* 插入到正确位置 */
      }
    for (int i = 1;i <= LENGTH;i++)
      printf( "%d  ", r[i]);              /* 输出排序后的数组元素 */
  }
```

程序运行结果：

3 11 24 49 60 66 71

3. 性能分析

对 n 个记录进行直接插入排序，需要进行 $n-1$ 趟。每趟中的主要操作是比较关键字和移动记录，而比较关键字和移动记录的次数取决于待排序记录序列的初始状态。

在最好情况下，待排序序列为正序，每趟只需与有序序列的最后一个记录的关键码比较一次，移动两次记录。总的比较次数为 $n-1$，记录移动的次数为 $2(n-1)$，因此，时间复杂度为 $O(n)$。

在最环情况下，待排序序列为逆序，在第 i 趟插入时，第 i 个记录必须与前面 $i-1$ 个记录的关键码以及监视哨做比较，并且每比较一次就要做一次记录的移动，则比较次数为 $(i+2)(i-1)/2$，记录的移动次数为 $(i+4)(i-1)/2$，因此，时间复杂度为 $O(n^2)$。

在平均情况下，待排序列中各种可能排列的概率相同，在插入第 i 个记录时平均需要比较有序部分全部记录的一半，所以总的比较次数为 $n^2/4$，移动次数为 $n^2/4$，因此，时间复杂度为 $O(n^2)$。

直接插入排序是一种稳定的排序方法。直接插入排序算法简单、容易实现，当序列中的记录基本有序或待排序记录较少时，它是最佳的排序方法。但是，当待排序的记录个数较多时，大量的比较和移动操作使直接插入排序算法的效率降低。

8.2.2　折半插入排序

将直接插入排序中寻找 r[i] 的插入位置的方法改为采用折半比较，即可得到折半插入排序算法。折半比较，就是在插入 r[i] 时，取 r[(i−1)/2] 的关键码值与 r[i] 的关键码值进行比较，如果 r[i] 的关键码值小于 r[(i−1)/2] 的关键码值，则说明 r[i] 只能插入 r[0] 到 r[(i−1)/2] 之间，故可以在 r[0] 到 r[(i−1)/2−1] 之间继续使用折半比较；否则只能插入 r[(i−1)/2] 到 r[i−1] 之间，故可以在 r[(i−1)/2+1] 到 r[i−1] 之间继续使用折半比较。如此循环，直到最后能够确定插入的位置为止。一般在 r[low] 和 r[high] 之间采用折半，其中间节点为 r[(low + high)/2]，经过一次比较即可排除一半记录，把可能插入的区间减小了一半，故称为折半。执行折半插入排序的前提是文件记录必须按顺序存储。

1. 基本思想

折半插入排序是将一个记录插入到已排好序的有序序列中时，通过折半查找的方法在有序序列中查找待排序记录的位置的排序方法。

2. 折半插入排序算法

折半插入排序算法的基本过程：

（1）计算 0~*i*−1 的中间点，用 *i* 索引处的元素与中间值进行比较，如果 *i* 索引处的元素大，说明要插入的这个元素应该在中间值和刚加入 *i* 索引之间，反之，就是在开始的位置和中间值之间，这样就完成了折半；

（2）在相应的半个范围里面找插入的位置时，不断地用（1）步骤缩小范围，范围依次缩小为 1/2，1/4，1/8，…快速地确定出第 *i* 个元素的插入位置；

（3）确定位置之后，将整个序列后移，并将元素插入到相应位置。

折半插入排序算法如下：

```
#include<stdio.h>
#define LENGTH 7
main()
{
   int low, high, mid;       /*low,high,mid 表示查找的上下界和中间位置 */
   int r[LENGTH + 1] = { 0,60,71,49,11,24,3,66 };
   for (int i = 2,j;i <= LENGTH;++i)          /*r[1] 是有序的，从 r[2] 开始排序 */
   {
      r[0] = r[i];                  /* 将 r[i] 暂时存到 r[0]*/
```

```
        low = 1;
        high = i - 1;              /* 置有序序列区间的初值 */
        while (low <= high)   /* 在 r[low] 到 r[high] 中折半查找插入位置 */
        {
          mid = (low + high) / 2;       /* 折半，取中间位置 mid*/
          if (r[0] < r[mid])
            high = mid - 1;             /* 插入位置在低半区 */
          else
            low = mid + 1;              /* 插入位置在高半区 */
        }
        for (j = i - 1;j >= high + 1;--j)
          r[j + 1] = r[j];              /* 插入位置以后的记录后移 */
        r[high + 1] = r[0];             /* 插入记录 */
      }
      for (int i = 1;i <= LENGTH;i++)
        printf( "%d  ", r[i]);          /* 输出排序后的有序序列 */
    }
```

程序运行结果：

3 11 24 49 60 66 71

3. 性能分析

折半插入排序与直接插入排序相比，只是减少了寻找插入位置时的比较次数，而记录的移动次数不变，因此折半插入排序的时间复杂度也是 $O(n^2)$。

折半插入排序与直接插入排序一样，在排序过程中只用了一个辅助单元 r[0]，因此其空间复杂度也是 $O(1)$，为就地排序。

折半插入排序是稳定的排序方法。因为在插入过程中，待插入记录与有序序列中的记录相等时，插入在后半部分，所以保证了算法的稳定性。

8.2.3 希尔排序

希尔排序 (Shell sort) 是对直接插入排序的一种改进，改进的着眼点是：

（1）若待排序记录按关键码基本有序，直接插入排序的效率很高。

（2）由于直接插入排序算法简单，则在待排序记录个数较少时效率也很高。

1. 基本思想

希尔排序的基本思想是先将待排序的记录序列按一定的增量分成若干个子

序列，具有相同增量的记录在同一子序列中，分别在各个子序列中进行直接插入排序。然后缩小增量重新进行分组和排序，直到增量为 1。此时待排序记录已基本有序，再对所有记录进行一次直接插入排序。

第一趟希尔排序时取增量 $d_1 = n/2$（n 为待排序记录数），以后每趟希尔排序的增量 $d_i = d_{i-1}/2$，直到最后一趟希尔排序的增量为 1。

希尔排序的过程如图 8-2 所示。

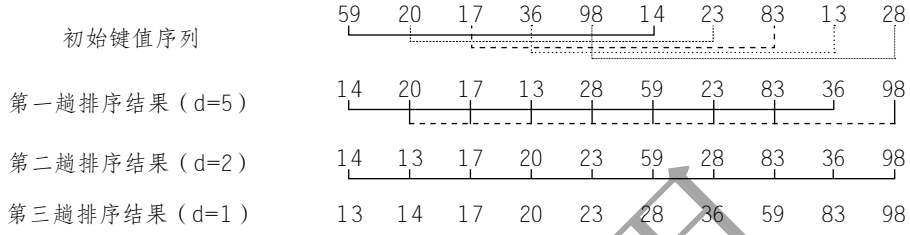


图 8-2　希尔排序的过程

2. 希尔排序算法

希尔排序程序如下：

```
#include<stdio.h>
#define LENGTH 10
main()
{
  int r[LENGTH] = { 59,20,17,36,98,14,23,83,13,28 };
  int d = LENGTH / 2;              /* 取第一个步长值 */
  while (d >= 1)                   /* 步长 d>=1, 对各组进行直接插入排序 */
  {
    for (int i = d,j; i < LENGTH; i++)      /* 对每组进行直接插入排序 */
    {
      int x = r[i];                /* 记录 r[i] 暂存入 x 中 */
      j = i - d;                   /* 确定每组中的记录 r[i] 前一个位置 */
      while (j >= 0 && x < r[j])   /* 在组中查找插入位置 */
      {
        r[j + d] = r[j];           /* 记录后移 */
        j = j - d;                 /* 记录位置前移一个步长 */
      }
      r[j + d] = x;                /* 插入记录 */
    }
```

```
        d = d / 2;                          /* 缩小步长值，取下一步长值 */
    }
    for (int i = 0;i < LENGTH; i++)
        printf( "%d  ", r[i]);              /* 输出排序后的有序序列 */
}
```

程序运行结果：

13 14 17 20 23 28 36 59 83 98

3. 性能分析

希尔排序开始时增量较大，每个子序列中的记录个数较少，从而排序速度较快；当增量较小时，虽然每个子序列中记录个数较多，但整个序列已基本有序，排序速度也较快。

希尔排序算法的时间性能是所取增量的函数，而到目前为止尚未有人求得一种最好的增量序列。

希尔排序的时间性能在 $O(n^2)$ 和 $O(n\log_2 n)$ 之间。当 n 在某个特定范围内，希尔排序所需的比较次数和记录的移动次数约为 $O(n1.3)$。

希尔排序是一种不稳定的排序方法。

8.3 交换排序

交换排序，就是根据序列中两个记录键值的比较结果来对换这两个记录在序列中的位置。交换排序的特点是：将键值较大的记录向序列的尾部移动，键值较小的记录向序列的前部移动。选择比较对象的方法不同，有不同的交换排序算法。常用的交换排序方法有冒泡排序和快速排序。

8.3.1 冒泡排序

冒泡排序（Bubble Sort），是计算机科学领域一种较简单的排序算法。

1. 基本思想

冒泡排序的基本思想是对所有相邻记录的关键字值进行比较，如果是逆序（r［i］>r［i+1］），则交换其位置，经过多趟排序，最终使整个序列有序。

冒泡排序处理过程如下。

第一趟排序，从第一条记录 r［1］开始，直到最后一条记录 r［n］，对两两相邻的记录依此比较，若发现为逆序，则立即交换其位置，最后使这 *n* 条记录中关键字最大的记录“下沉”到最底部，既被交换到第 *n* 个位置上，它不参与下一趟排序。

第二趟排序，从第一条记录 r［1］开始，直到第 *n*−1 条记录 r［n−1］，对两两相邻的记录依此比较，若发现为逆序，则立即交换其位置，最后使这 *n*−1 条记录中关键字最大的记录“下沉”到次底部，即被交换到第 *n*−1 个位置上，它不参与下一趟排序。如此反复，最多经过（*n*−1）趟冒泡排序，就可以使整个序列成为有序序列。

例如，有一组待排序的记录关键码（42,36,56,78,67,11,27,36），要求按照关键码由小到大进行排序，如图 8-3 所示。

初始状态	42	36	56	78	67	11	27	36
i=1	36	42	56	67	11	27	36	78
i=2	36	42	56	11	27	36	67	78
i=3	36	42	11	27	36	56	67	78
i=4	36	11	27	36	42	56	67	78
i=5	11	27	36	36	42	56	67	78
i=6	11	27	36	36	42	56	67	78
i=7	11	27	36	36	42	56	67	78

图 8-3　冒泡排序过程实例图

2. 冒泡排序算法

冒泡排序程序如下：

```
#include<stdio.h>
#define LENGTH 8
main()
{
  int r[LENGTH] = { 42,36,56,78,67,11,27,36 };   /* 定义数组并赋初值 */
  for (int i = 1;i <= LENGTH - 1;i++) {          /* 控制共进行 LENGTH-1
趟排序 */
    for (int j = 0;j < LENGTH - i;j++) {         /* 进行第 i 趟排序 */
      if (r[j] > r[j + 1])                       /* 判断相邻两记录是否逆序 */
      {
        int temp = r[j];
```

```
            r[j] = r[j + 1];
            r[j + 1] = temp;                    /* 如逆序，交换两记录 */
          }
        }
      }
      for (int i = 0;i < LENGTH;i++)
        printf( "%d  ", r[i]);                  /* 输出排序后的序列 */
      printf( "\n" );
    }
```

程序运行结果：

11 27 36 36 42 56 67 78

3. 性能分析

最好情况下，待排记录是正序，只需要一趟冒泡排序就可完成排序。比较次数是 $n-1$，移动次数是 0，因此最好情况下的时间复杂度是 $O(n)$。

最坏情况下，待排记录是逆序，每趟排序在无序序列中只有一个最大的记录被交换到最终位置，则需要进行 $n-1$ 趟排序。第 i（$1 \leqslant i < n$）趟排序执行了 $n-i$ 次关键码的比较和 $n-i$ 次记录的交换，总的比较次数是 $\frac{n(n-1)}{2}$。每一次比较都需要交换记录，所以总的移动次数是 $\frac{n(n-1)}{2}$。因此最坏情况下的时间复杂度是 $O(n^2)$。

平均情况下，待排序记录为随机序列，时间复杂度与最坏情况同数量级。

冒泡排序只需要一个辅助存储空间用于记录的交换，其空间复杂度是 $O(1)$，为就地排序。

冒泡排序是一种稳定的排序方法。

8.3.2 快速排序

快速排序（Quick Sort）是对冒泡排序的一种改进，改进的着眼点是，在冒泡排序中，记录的比较和移动是在相邻单元中进行的，记录每次交换只能上移或下移一个单元，因而总的比较次数和移动次数较多。

1. 基本思想

快速排序的基本思想是，通过一趟排序将待排记录分割成独立的两部分，其

中一部分的所有记录都比另外一部分的所有记录要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

一趟快速排序的过程为：

（1）取第一个记录为基准，并将其存入临时存储单元。将工作指针 *i* 和 *j* 分别指向第一个记录和最后一个记录。

（2）将 *j* 所指的记录与基准比较，若 *j* 所指的记录大于或等于基准，则将 *j* 指向前一个记录。重复上述过程，直到 *j* 所指的记录小于基准，则将该记录移到 *i* 处，并 *i*++。

（3）将 *i* 所指的记录与基准比较，若 *i* 所指的记录小于基准，则将 *i* 指向后一个记录。重复上述过程，直到 *i* 所指的记录大于或等于基准，则将该记录移到 *j* 处，并 *j*--。

（4）重复（2）和（3）步，直到 *i* 和 *j* 指向同一位置，即为基准的最终位置。

图 8-4 是快速排序的一趟排序过程，以第一个记录 48 为基准。快速排序的全部过程图 8-5 所示。

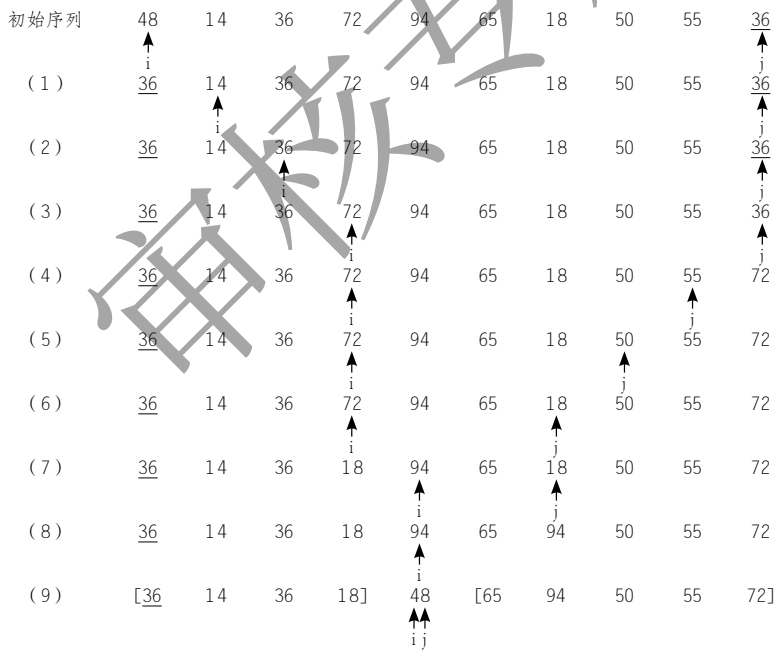


图 8-4　一趟快速排序的过程

初始状态	48	14	36	72	94	11	18	50	55	36
第一趟	[36	14	36	18]	48	65	94	50	55	72]
第二趟	[18	14]	36	[36]	48	[65	94	50	55	72]
第三趟	[14]	18	36	[36]	48	[65	94	50	55	72]
第四趟	[14]	18	36	[36]	48	[55	50]	65	[94	72]
第五趟	[14]	18	36	[36]	48	[50]	55	65	[94	72]
第六趟	[14]	18	36	[36]	48	[50]	55	65	[72]	94

图 8-5　快速排序的全过程

2. 快速排序算法

一趟快速排序的算法是：

（1）设置两个变量 i、j，排序开始的时候：$i=0$，$j=n-1$；

（2）以第一个数组元素作为关键数据，赋值给 key，即 key=r[0]；

（3）从 j 开始向前搜索，即由后开始向前搜索（j--），找到第一个小于 key 的值 r[j]，将 r[j] 和 r[i] 的值交换；

（4）从 i 开始向后搜索，即由前开始向后搜索（i++），找到第一个大于 key 的 r[i]，将 r[i] 和 r[j] 的值交换；

（5）重复第 3、4 步，直到 i=j；（3,4 步中，没找到符合条件的值，即 3 中 r[j] 不小于 key,4 中 r[i] 不大于 key 的时候改变 j、i 的值，使得 $j=j-1$，$i=i+1$，直至找到为止。找到符合条件的值，进行交换的时候 i，j 指针位置不变。另外，$i==j$ 这一过程一定正好是 i++ 或 j-- 完成的时候，此时令循环结束）。

快速排序算法程序如下：

```
#include<stdio.h>
#define LENGTH 10

int Partition(int r[], int s, int t)   /* 一趟快速排序算法，将基准记录移到正确位置 */
                                       /* 并返回其所在位置 */
{
  int i = s, j = t;
  int rp = r[s];                       /* 基准记录暂存入 rp*/
  while (i < j)                        /* 从序列的两端交替向中间扫描 */
  {
    while (i < j && r[j] >= rp)
      j--;                             /* 扫描比基准记录小的位置 */
```

```
        r[i] = r[j];                    /* 将比基准记录小的记录移到低端 */
        while (i < j && r[i] <= rp)
          i++;                          /* 扫描比基准记录大的位置 */
        r[j] = r[i];                    /* 将比基准记录大的记录移到高端 */
      }
      r[i] = rp;                        /* 基准记录到位 */
      return i;                         /* 返回基准记录位置 */
    }

    void Qsort(int r[], int s, int t)   /* 快速排序递归算法 */
    {
      if (s < t)                        /* 长度大于 1*/
      {
        int k = Partition(r, s, t);     /* 调用一趟快速排序算法将 r[s]..r[t] 一分为二 */
        Qsort(r, s, k - 1);             /* 对低端子序列递归排序 ,k 是支点位置 */
        Qsort(r, k + 1, t);             * 对高端子序列递归排序 */
      }
    }

    main()
    {
      int r[LENGTH] = { 48,14,36,72,94,65,18,50,55,36 };  /* 定义原始待排序序列 */
      Qsort(r, 0, LENGTH - 1);                  /* 调用快速排序算法 */
      for (int i = 0;i < LENGTH;i++)
        printf( "%d  ", r[i]);                  /* 输出排序后的有序序列 */
    }
```

程序运行结果：

14 18 36 36 48 50 55 65 72 94

3. 性能分析

快速排序中记录的移动次数小于比较次数，因此在讨论时间复杂度时仅考虑记录的比较次数即可。

在最好情况下，每次划分对一个记录定位后，该记录的左侧子序列与右侧子序列的长度相同。在具有 n 个记录的序列中，对一个记录定位要对整个待划分序

列扫描一遍，则所需时间为 $O(n)$。设 $T(m)$ 是对 n 个记录的序列进行排序的时间，每次划分后，正好把待划分区间划分为长度相等的两个子序列，则有：

$$
\begin{aligned}
&T(n) \leqslant 2T(n/2)+n\\
&\leqslant 2((n/4)+n/2)+n=4T(n/4)+2n\\
&\leqslant 4(2T(n/8)+n/4)+2n=8T(n/8)+3n\\
&\cdots\cdots\\
&\leqslant T(1)+n\log 2n=O(n\log 2\ n)
\end{aligned}
$$

因此，时间复杂度为 $O(n\log_2 n)$。

在最坏情况下，待排序记录序列正序或逆序，每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列为空）。此时，必须经过 $n-1$ 次递归调用才能把所有记录定位，而且第 i 趟划分需要经过 $n-i$ 次关键码的比较才能找到第 i 个记录的基准位置，因此，时间复杂度为 $O(n^2)$

在平均情况下，时间复杂度的数量级也为 $O(n\log_2 n)$。

由于快速排序是递归的，需要一个栈来存放每一层递归调用的必要信息，其最大容量应与递归调用的深度一致。快速排序是一种不稳定的排序方法。

8.4 选择排序

选择排序（Selection Sort）是一种简单直观的排序算法。它的工作原理是：第一次从待排序的记录中选出最小（或最大）的一个记录，存放在序列的起始位置，然后再从剩余的未排序记录中寻找到最小（或最大）记录，然后放到已排序的序列的末尾。以此类推，直到全部待排序记录的个数为零。选择排序是不稳定的排序方法。

8.4.1 简单选择排序

简单选择排序又称直接选择排序，是选择排序中最简单的一种。

1. 基本思想

第一趟排序从待排序记录 r[1]..r[n] 中选出最小的记录与 r[1] 交换，第二趟排序从待排序记录 r[2]..r[n] 中选出最小的记录与 r[2] 交换 .. 第 n-1 趟排序从待排序记录 r[n-1]..r[n] 中选出最小的记录与 r[n-1] 交换。进行 n-1 趟排序，得到一个从小到大的有序序列。

例如，有一组待排序的记录关键码，要求按照关键码有小到大进行排序，如图 8-6 所示。

图 8-6　简单选择排序的过程示例图

2. 简单选择排序算法

具体简单选择排序算法程序如下：

```
#include<stdio.h>
#define LENGTH 7
main()
{
   int r[LENGTH] = { 49,27,65,97,76,13,38 };
   int d = LENGTH / 2;              /* 取第一个步长值 */
   while (d >= 1)                   /* 步长 d>=1, 对各组进行直接插入排序 */
   {
      for (int i = d;i < LENGTH;i++)  /* 对每组进行直接插入排序 */
      {
         int x = r[i];              /* 记录 r[i] 暂存入 x 中 */
         int j = i - d;             /* 确定每组中的记录 r[i] 前一个位置 */
         while (j >= 0 && x < r[j])  /* 在组中查找插入位置 */
         {
            r[j + d] = r[j];        /* 记录后移 */
            j = j - d;              /* 记录位置前移一个步长 */
         }
         r[j + d] = x;              /* 插入记录 */
```

```
        }
        d = d / 2;                  /* 缩小步长值，取下一步长值 */
    }
    for (int i = 0;i < LENGTH; i++)
        printf( "%d  ", r[i]);          /* 输出排序后的有序序列 */
}
```

程序运行结果：

13 27 38 49 65 76 97

3. 性能分析

选择排序的交换操作介于 0 和 $(n-1)$ 次之间。选择排序的比较操作为 $\frac{n(n-1)}{2}$ 次之间。选择排序的赋值操作介于 0 和 3 $(n-1)$ 次之间。

无论记录的初始排列如何，比较次数与关键码的初始状态无关，第 i 趟排序需进行 n−i 次关键码的比较，而简单选择需进行 n−1 趟排序，则总的比较次数为 $\frac{n(n-1)}{2}$。

所以，总的时间复杂度为 $O(n^2)$，这是简单选择排序最好、最坏和平均的时间性能。

直接选择排序过程中只需要一个辅助存储空间，其空间复杂度为 $O(1)$，为就地排序。

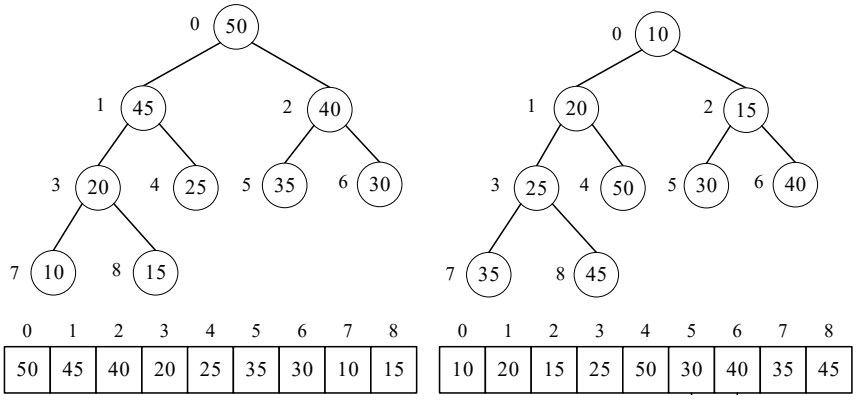
直接选择排序是不稳定的排序方法。

8.4.2 堆排序

堆排序（Heap Sort）是简单选择排序的一种改进，改进的着眼点是：如何减少关键码间的比较次数。若能利用每趟比较后的结果，也就是在找出键值最小记录的同时，也找出键值较小的记录，则可减少后面的选择中所用的比较次数，从而提高整个排序过程的效率。

1. 堆的定义

堆是具有下列性质的完全二叉树：每个节点的值都小于或等于其左右孩子节点的值（称为小根堆），或每个节点的值都大于或等于其左右孩子节点的值（称为大根堆），图 8-7 是两个堆的示意图。



（a）大根堆及其对应的序列　　（b）小根堆及其对应的序列

图 8-7　堆的示意图

2. 基本思想

堆排序的基本思想是，将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余 $n-1$ 个记录重新构造成一个堆，这样会得到 n 个记录的次小值。如此反复执行，便能得到一个有序序列了。

以大根堆为例，建立初始堆的步骤如下：

步骤一：构造初始堆。将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆）。

（1）假设给定无序序列结构如图 8-8 所示。

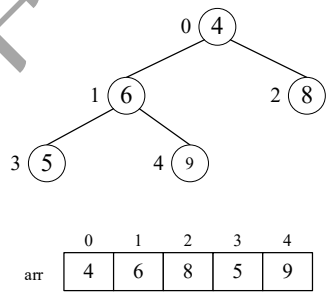


图 8-8　初始无序

（2）此时我们从最后一个非叶子节点开始（叶节点自然不用调整，第一个非叶子节点 arr.length/2-1=5/2-1=1，也就是下面的 6 节点），从左至右，从下至上进行调整，如图 8-9 所示。

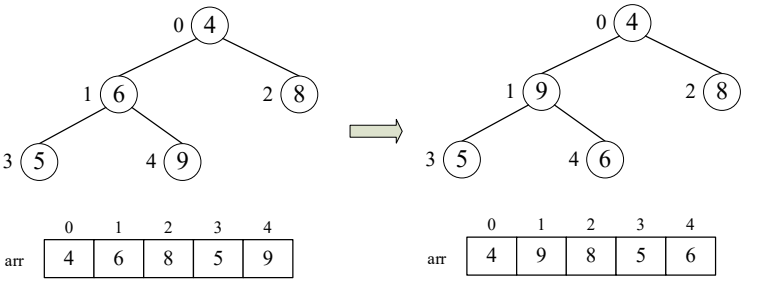


图 8-9　调整最后一个非叶子节点

（3）找到第二个非叶子节点 4，由于 [4,9,8] 中 9 元素最大，4 和 9 交换，如图 8-10 所示。

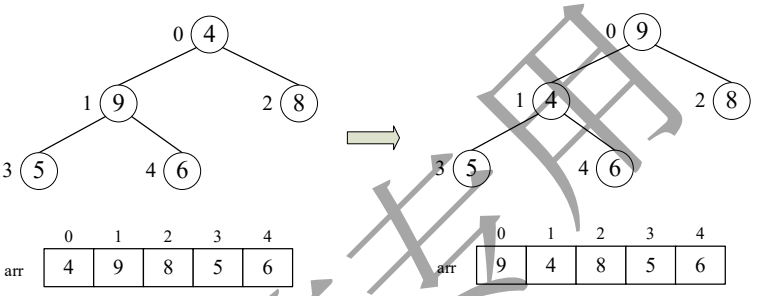


图 8-10　调整第二个非叶子节点

（4）这时，交换导致了子根 [4,5,6] 结构混乱，继续调整，[4,5,6] 中 6 最大，交换 4 和 6，如图 8-11 所示。

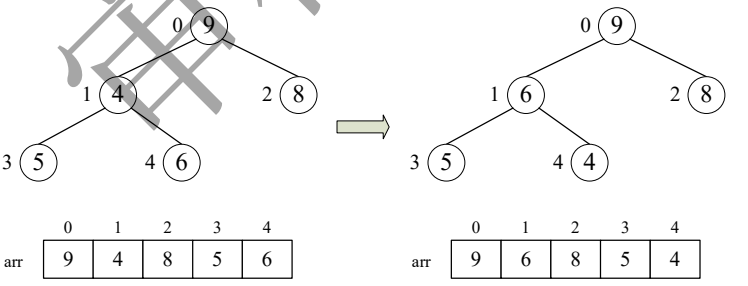


图 8-11　调整子根

此时，我们就将一个无需序列构造成了一个大顶堆。

步骤二：将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。

（1）将堆顶元素 9 和末尾元素 4 进行交换，如图 8-12 所示。

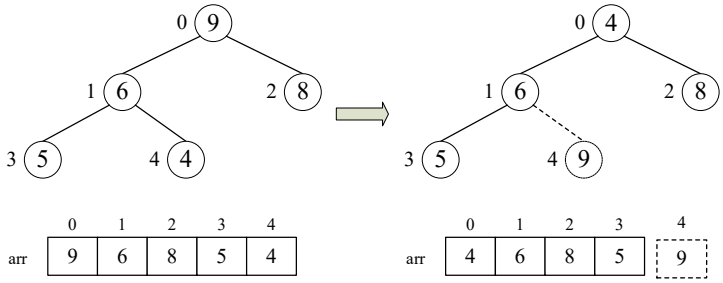


图 8-12　交换元素

（2）重新调整结构，使其继续满足堆定义，如图 8-13 所示。

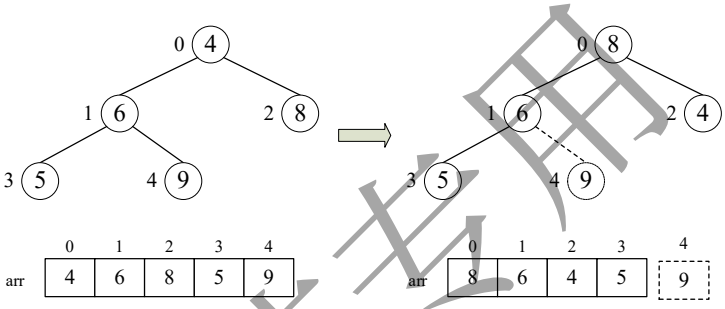


图 8-13　重新调整结构

（3）再将堆顶元素 8 与末尾元素 5 进行交换，得到第二大元素 8，如图 8-14 所示。

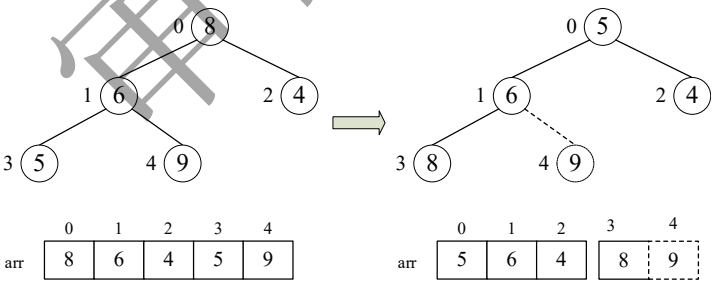


图 8-14　再次交换元素

（4）后续过程，继续进行调整，交换，如此反复进行，最终使得整个序列有序，如图 8-15 所示。

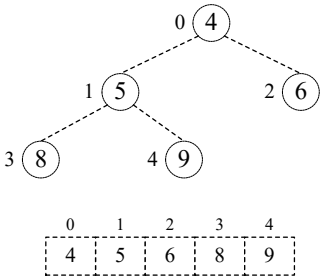


图 8-15　调整后续过程

3. 堆排序的算法

堆排序算法程序如下：

```
#include<stdio.h>
#define LENGTH 5

void HeapAdjust(int r[], int s, int m)  /* 把 r[s],..,r[m] 建立成大顶堆的算法 */
{
   int rc = r[s];                /* 记录关键字送 rc*/
   for (int i = 2 * s;i <= m;i *= 2)  /* 沿关键字较大的孩子节点向下搜索调整 */
   {
      if ((i < m) && (r[i] <= r[i + 1]))
         ++i;                  /* 若右孩子大于左孩子，则 i 为右孩子的下标 */
      if (rc > r[i])
         break;                /* 将 r[i] 调到父节点的位置 */
      r[s] = r[i];
      s = i;
   }
   r[s] = rc;                   /* 将 rc 插入到最终位置 */
}
main()
{
   int r[LENGTH + 1] = { 0,4,6,8,5,9 };  /* 给数组赋值 ,r[0] 无意义 */
   for (int i = LENGTH / 2;i > 0;--i)     /* 将待排序序列调整为大顶堆 */
      HeapAdjust(r, i, LENGTH + 1);  /* 调用建堆算法函数 */
   for (int i = LENGTH;i > 1;--i)       /* 排序 ( 重复进行记录交换和堆调整 )*/
   {
      int x = r[1];              /* 将堆顶记录与当前未经排序子序列 r[1..i] 中 */
```

```
            r[1] = r[i];              /* 最后一个记录相互交换 */
            r[i] = x;
              HeapAdjust(r, 1, i - 1);        /* 调用建堆算法将 r[1..i-1] 重新调整为大
顶堆 */
          }
          for (int i = 1;i <= LENGTH;i++)   /* 输出排序后的记录序列 */
            printf( "%d  ", r[i]);
          printf( "\n" );
      }
```

程序运行结果：

4 5 6 8 9

4. 性能分析

堆排序是一种选择排序，整体主要由构建初始堆以及交换堆顶元素和末尾元素并重建堆两部分组成。其中构建初始堆经推导复杂度为 $O(n)$，在交换并重建堆的过程中，需交换 $n-1$ 次，而重建堆的过程中，根据完全二叉树的性质，$[\log_2(n-1), \log_2(n-2)...1]$ 逐步递减，近似为 $n\log_2 n$。所以堆排序时间复杂度一般认为就是 $O(n\log_2 n)$ 级。

堆排序仅需一个记录大小供交换用的辅助存储空间。对于存在相同关键码的记录情况，堆排序是不稳定的。

8.5　排序方法的比较

从以下几个方面综合比较各种排序方法，结果如下：

1. 时间复杂度

直接插入排序、简单选择排序和冒泡排序这三种排序方法的时间复杂度均为 $O(n^2)$，所以适用 n 较小的情况。

堆排序和归并排序这种排序方法的时间复杂度为 $O(n\log_2 n)$，快速排序方法的平均时间复杂度也为 $O(n\log_2 n)$，但快速排序在最坏情况下的时间复杂度为 $O(n^2)$。

希尔排序方法的时间复杂度介于 $O(n\log_2 n)$ 与 $O(n^2)$ 之间。

2. 空间复杂度

快速排序的空间复杂度为 $O(\log_2 n)$（实现递归算法所需的栈空间）；其他排序方法的空间复杂度均为 $O(1)$。

3. 稳定性

直接插入排序、冒泡排序是稳定的，而希尔排序、简单选择排序、快速排序和堆排序是不稳定的。

综上所述，在所介绍的几种排序方法中，没有哪一种是绝对最优的。有的适用于 n 较大的情况，有的适用于 n 较小的情况。因此，在使用时需根据不同情况适当选用，甚至可将多种方法结合起来使用。

本章小结

内部排序分为插入排序、交换排序、选择排序这个三大类。插入排序分为直接插入排序、二分插入排序和希尔排序；交换排序分为冒泡排序和快速排序；选择排序有直接选择排序和堆排序两种。其中，直接插入排序、二分插入排序、冒泡排序和直接选择排序属于简单的排序方法，希尔排序、快速排序和堆排序属于改进的排序方法。

直接选择排序、希尔排序、快速排序和堆排序是不稳定的排序方法，直接插入排序、折半插入排序和冒泡排序是稳定的排序方法。

快速排序所需的辅助存储空间是 $O(\log_2 n)$，其他排序方法是 $O(1)$。

平均情况下排序速度最快的是快速排序，最坏情况下堆排序的效率最好，最好情况下直接插入排序、折半插入排序和冒泡排序的效率最好。

习题

一、选择题

1. 下述排序算法中，稳定的是_____。

A. 直接选择排序　　B. 直接插入排序

C. 快速排序　　D. 堆排序

2. 下列排序算法中，_____需要的辅助存储空间最大。

A. 快速排序　　B. 插入排序

C. 希尔排序　　D. 归并排序

3. 下列排序算法中，_____是稳定的。

A. 插入、希尔　　B. 冒泡、快速

C. 选择、堆排序　　D. 基数、归并

4. 下列各种排序算法中平均时间复杂度为 $O(n^2)$ 的是_____。

A. 快速排序　　B. 堆排序

C. 归并排序　　D. 冒泡排序

5. 在待排序的元素基本有序的前提下，效率最高的排序方法是_____。

A. 简单插入排序　　B. 简单选择排序

C. 快速排序　　D. 归并排序

6. 利用直接插入排序法的思想建立一个有序线性表的时间复杂度为_____。

A. $O(n)$　　B. $O(n\log_2 n)$

C. $O(n^2)$　　D. $O(\log_2 n)$

7. 对 n 个记录进行堆排序，所需要的辅助存储空间为_____。

A. $O(\log_2 n)$　　B. $O(n)$

C. $O(1)$　　D. $O(n^2)$

8. 快速排序在最坏情况下的时间复杂度为_____。

A. $O(\log_2 n)$　　B. $O(n\log_2 n)$

C. $O(n)$　　D. $O(n^2)$

9. 设有序列 12、42、37、19，当使用直接插入排序从小到大排序时，其比较次数为_____。

A.3　　B.4　　C.5　　D.6

10. 对数据元素序列 (49, 72, 68, 13, 38, 50, 97, 27) 排序，前三趟排序结束时的结果依次为：

第一趟：13, 72, 68, 49, 38, 50, 97, 27；

第二趟：13, 27, 68, 49, 38, 50, 97, 72；

第三趟：13, 27, 38, 49, 68, 50, 97, 72；

该排序采用的方法是_____。

A. 直接插入排序　　B. 直接选择排序

C. 冒泡排序　　D. 堆排序

二、填空题

1. 设一组初始记录关键字序列为 (49，38，65，97，76，13，27，50)，则以 d=4 为增量的一趟希尔排序结束后的结果为_____。

2. 对一组初始关键字序列（40，50，95，20，15，70，60，45，10）进行

冒泡排序，则第一趟需要进行相邻记录的比较的次数为_____，在整个排序过程中最多需要进行_____趟排序才可以完成。

3. 设有 n 个无序的记录关键字，则直接插入排序的时间复杂度为_____，快速排序的平均时间复杂度为_____。

4. 在堆排序的过程中，对任一分支节点进行筛运算的时间复杂度为_____，整个堆排序过程的时间复杂度为_____。

5. 对 n 个元素的序列进行冒泡排序时，最少的比较次数是_____。

6. 在插入和选择排序中，若初始数据基本正序，则选用_____；若初始数据基本反序，则选用_____。

7. 在对一组记录（54，38，96，23，15，72，60，45，83）进行直接插入排序时，当把第 7 个记录 60 插入到有序表时，为寻找插入位置需比较_____。

8. 设有一组初始关键字序列为 (24，35，12，27，18，26)，则第 3 趟直接插入排序结束后的结果是_____。

三、判断题

1. 直接选择排序是一种稳定的排序方法。（　　）

2. 冒泡排序在初始关键字序列为逆序的情况下执行的交换次数最多。（　　）

3. 希尔排序算法的时间复杂度为 $O(n^2)$。（　　）

4. 设初始记录关键字基本有序，则快速排序算法的时间复杂度为 $O(n\log_2 n)$。（　　）

5. 一组关键码已完全有序时，最快的排序方法是快速排序。（　　）

6. 快速排序是排序算法中平均性能最好的一种排序。（　　）

7. 堆是完全二叉树，完全二叉树不一定是堆。（　　）

8. 层次遍历初始堆可以得到一个有序的序列。（　　）

9. 从平均性能而言，快速排序最佳，其所需时间最省。（　　）

参考文献

[1] 严蔚敏，吴伟民 . 数据结构（C 语言版）[M]. 北京：清华大学出版社，2016.
[2] 刘畅，姚学峰 . 数据结构 [M]. 上海：上海交通大学出版社，2016.
[3] 邹岚 . 数据结构 [M].2 版 . 大连：大连理工大学出版社，2018.
[4] 李春葆 . 数据结构教程 [M]. 北京：清华大学出版社，2009.
[5] 安训国 . 数据结构 [M].5 版 . 大连：大连理工出版社，2016.
[6] 陈锐 . 数据结构 [M]. 北京：清华大学出版社，2012.
[7] 王红梅 . 数据结构：C++ 版 [M]. 北京：清华大学出版社，2008.
[8] 蒋文蓉 . 数据结构 [M]. 北京：高等教育出版社，2006.
[9] 林小茶 . 实用数据结构 [M]. 北京：清华大学出版社，2008.
[10] 陈越 . 数据结构 [M].2 版 . 北京：高等教育出版社，2016.