

2020 考研

计算机数据结构基础教程

中公教育学员内部专用

offcn

目 录

第一部分 导学.....	1
第二部分 知识精讲.....	3
第一章 绪论.....	3
第一节 数据结构的基本概念.....	3
第二节 算法及其效率度量.....	5
第二章 线性表.....	10
第一节 线性表的定义.....	10
第二节 线性表的顺序表示和实现.....	13
第三节 线性表的链式表示和实现.....	18
第三章 栈和队列.....	27
第一节 栈.....	27
第二节 栈的应用.....	31
第三节 队列.....	33
第四节 队列的应用.....	39
第四章 数组.....	41
第一节 数组.....	41
第五章 树和二叉树.....	49
第一节 树的定义和基本术语.....	49
第二节 二叉树.....	51
第三节 遍历二叉树和线索二叉树.....	55
第四节 树和森林.....	64
第五节 哈夫曼树及其应用.....	69
第六章 图.....	73

第一节 图的定义和术语.....	74
第二节 图的存储结构.....	76
第三节 图的遍历.....	79
第四节 最小生成树.....	84
第五节 有向无环图及其应用.....	87
第六节 最短路径.....	92
第七章 查找.....	96
第一节 静态查找表.....	96
第二节 动态查找表.....	101
第三节 哈希表.....	111
第八章 内部排序.....	116
第一节 插入排序.....	116
第二节 交换排序.....	121
第三节 选择排序.....	125
第四节 其他排序.....	129
第五节 各种排序方法比较.....	134

第一部分 导学

【学科介绍】

《数据结构》在计算机科学中是一门综合性的核心专业基础课，它是操作系统、数据库、编译原理等所有计算机类专业基础课和专业课的重要基础，它还是进行程序设计，尤其是进行高水平的应用程序和系统程序必不可少的基础。而且《数据结构》正逐渐发展成为众多理工专业的热门选修课。数据结构课程在整个课程体系中处于承上启下的核心地位，它一方面扩展和深化在离散数学、程序设计语言等课程学到的基本技术和方法，一方面为进一步学习其它专业课奠定坚实的理论与实践基础。

数据结构课程的学习，在掌握数据组织、存储和运算的基本原理和方法的基础上，还可以培养我们对各类数据结构和相关算法的分析和设计的能力，并能够编写出正确、清晰和较高质量的算法和程序。因此，计算机考研专业课中重点考查数据结构。

【考情分析】

1. 考查目标

数据结构的主要考查目标是学习数据的逻辑结构，存储结构以及相关的算法设计。目的是使同学们学会分析待加工处理数据的特性，以便选择适当的逻辑结构、存储结构以及进行相应的算法设计。在学会数据结构选择和算法设计的同时，还培养同学们的抽象思维能力、逻辑推理能力和形式化思维方法，增强分析问题和解决问题的能力，更重要的是培养专业兴趣、树立创新意识。

2. 真题分析

(1) 统考

①考试方式

笔试闭卷，考试时间为 180 分钟，满分为 150 分。

②题型

单项选择题（80 分，共 40 题）、综合应用题（70 分，共 7 题）。

③试卷结构

一共 4 门课程：数据结构、计算机组成原理、操作系统、计算机网络。其中数据结构占 47 分，包括单项选择题（1-11 题）、综合应用题（41 题算法设计题，13 分；42 题应用题，12 分）。

④考查重点

- a. 时间复杂度分析。
- b. 线性表的插入、删除、查找算法。
- c. 栈的出入栈过程，队列的出入队过程，循环队列的空满条件，栈的应用。

- d. 串的模式匹配，数组、矩阵元素的地址计算。
- e. 二叉树的性质、遍历、线索化，树、森林和二叉树的相互转化，Huffman 树、哈夫曼编码。
- f. 图的存储、遍历、应用（最小生成树、关键路径、拓扑排序、最短路径）。
- g. 查找中顺序查找、折半查找、二叉排序树、平衡二叉树、B 树、B+树、哈希表，平均查找长度的计算。
- h. 排序中直接插入排序、简单选择排序、快速排序、堆排序、冒泡排序、希尔排序、归并排序、基数排序的过程模拟以及性能分析。

（2）非统考

①考试方式

笔试闭卷，考试时间为 180 分钟，满分为 150 分。

②题型

数据结构的题型多样，常见的有判断题、选择题、填空题、阅读题（代码补全题、分析算法功能题）、问答题（基本概念题）、应用题、算法设计题。一般每个学校考 3-5 种题型。

③试卷结构

学校不同，考试科目不同，数据结构基本必考（北航不考），其他 3 门课程不同学校不同规定，考其中一门的也有，考两门的也有，只考数据结构的也有，也有考别的科目的（比如软件工程）。因此数据结构在专业课考查中分数占比很高。

④考查重点

同统考。有些学校还考广义表（概念，常考表头、表尾、长度、深度），串的基本概念，矩阵的转置。

【参考书目】

严蔚敏主编.《数据结构 C 语言版》. 清华大学出版社。

第二部分 知识精讲

第一章 绪论

【本章综述】

本章内容并不在考研大纲中，它是数据结构的一个概述。但读者千万不要忽视本章，更应该通过对本章的学习初步了解数据结构的基本内容和基本方法。**分析算法的时间复杂度和空间复杂度是本章的重点，属于必考内容**，一定要熟练掌握，每年都会结合算法设计题而出现。还多次直接以选择题的形式考查了时间复杂度的计算。掌握本章的基本概念，对后续章节的学习以及整个数据结构课程的理解是非常重要的。

【复习要点】

（一）数据结构基本概念

1. 数据结构相关的概念和术语
2. 数据结构的三要素：逻辑结构、物理结构和数据运算

（二）算法及其分析

1. 算法基本概念
2. 算法的时间复杂度和空间复杂度的分析与计算

第一节 数据结构的基本概念

数据（data）是对客观事物的符号表示，在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。

数据元素（data element）是数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。一个数据元素可由若干个**数据项**（data item）组成，数据项是数据的不可分割的最小单位。

数据对象（data object）是性质相同的数据元素的集合，是数据的一个子集。

数据结构（data structure）是相互之间存在一种或多种特定关系的数据元素的集合。这是本书对数据结构的一种简单解释。在任何问题中，数据元素都不是孤立存在的，而是在它们之间存在着某种关系，这种数据元素相互之间的关系称为**结构**（structure）。

根据数据元素之间关系的不同特性，通常有下列 4 类基本结构：

集合 结构中的数据元素之间除了“同属于一个集合”的关系外，别无其他关系；

线性结构 结构中的数据元素之间存在一个对一个的关系；

树形结构 结构中的数据元素之间存在一个对多个的关系；

图形结构或网状结构 结构中的数据元素之间存在多个对多个的关系。

下图是 4 类基本结构的关系图。

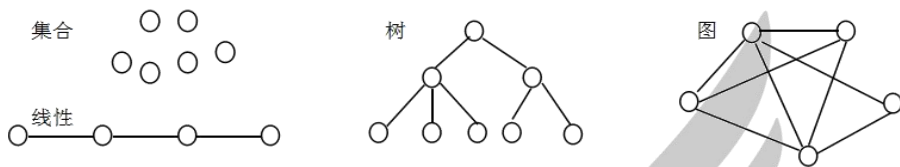


图 1-1 4 类基本结构关系图

【习题演练】

- 通常要求同一逻辑结构中的所有数据元素具有相同的特性，这意味着（ ）。
 - 数据具有同一特点
 - 不仅数据元素所包含的数据项的个数要相同，而且对应数据项的类型要一致
 - 每个数据元素都一样
 - 数据元素所包含的数据项的个数要相等
- 以下说法正确的是（ ）。
 - 数据元素是数据的最小单位
 - 数据项是数据的基本单位
 - 数据结构是带有结构的各数据项的集合
 - 一些表面上很不不同的数据可以有相同的逻辑结构

数据元素之间的逻辑关系，称为数据的**逻辑结构**。数据结构在计算机中的表示（又称映像）称为数据的**物理结构**，又称**存储结构**，它包括数据元素的表示和关系的表示。

元素或结点是数据元素在计算机中的映像。

数据元素之间的关系在计算机中有两种不同的表示方法：**顺序映像**和**非顺序映像**，并由此得到两种不同的存储结构：**顺序存储结构**和**链式存储结构**。顺序映像的特点是借助元素在存储器中的相对位置来表示数据元素之间的逻辑关系。非顺序映像的特点是借助指示元素存储地址的**指针**（pointer）表示数据元素之间的逻辑关系。

数据类型（data type）是一个值的集合和定义在这个值集上的一组操作的总称。

按“值”的不同特性，数据类型可分为两类：

一类是**基本类型**，其值是不可分解的，例如 C 语言中的整型、实型、字符型等。

另一类是**结构类型**，其值是可分解的，并且它的成分可以是原子的也可以是结构的。

抽象数据类型（Abstract Data Type，简称 ADT）是指一个数学模型以及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性，而与计算机内部如何表示和实现无关。抽象数据类型可以用三元组表示为（数据对象，数据关系，基本操作）。

【习题演练】

1. 在数据结构中，从逻辑上可以把数据结构分成（ ）。
A. 动态结构和静态结构 B. 紧凑结构和非紧凑结构
C. 线性结构和非线性结构 D. 内部结构和外部结构
2. 与数据元素本身的形式、内容、相对位置、个数无关的是数据的（ ）。
A. 存储结构 B. 存储实现
C. 逻辑结构 D. 运算实现

第二节 算法及其效率度量

一、算法的基本概念

1. 算法

算法（algorithm）是对特定问题求解步骤的一种描述，它是指令的有限序列。其中每一条指令表示一个或多个操作。

一个算法具有 5 个重要特性：

（1）**有穷性** 一个算法总是在执行有穷步之后结束，且每一步都可在有穷时间内完成。

（2）**确定性** 算法中每一条指令必须有确切的含义，不能产生二义性。并且相同的输入只能得出相同的输出。

（3）**可行性** 算法中描述的操作都可通过已经实现的基本运算执行有限次来实现的。

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

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

2. 算法设计的要求

通常设计一个“好”的算法应考虑达到以下目标。

(1) **正确性** 算法应当满足具体问题的需求。它至少应当包括对于输入、输出和加工处理等的明确的无歧义性的描述。

(2) **可读性** 算法主要是为了人的阅读与交流，其次才是机器执行。可读性好有助于人对算法的理解。

(3) **健壮性** 当输入数据非法时，算法也能适当地做出反应或进行处理，而不会产生莫名其妙的输出结果。

(4) **效率与低存储量需求** 效率指的是算法执行的时间。存储量需求指算法执行过程中所需要的最大存储空间。效率与低存储量需求这两者都与问题的规模有关。

二、算法的效率度量

算法的时间量度指的是算法中基本操作重复执行的次数。

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

$$T(n)=O(f(n))$$

通常称为**时间复杂度**，其中 O 的形式定义为：若 $f(n)$ 是正整数 n 的一个函数，则 $x_n=O(f(n))$ 表示存在一个正的常数 M ，使得当 $n \geq n_0$ 时都满足 $|x_n| \leq M|f(n)|$ 。

注意：基本操作是其重复执行的次数和算法的执行时间成正比的原操作，多数情况下它是最深层循环内的语句中的原操作，它的执行次数和包含它的语句的频度是相同的。语句的**频度**指的是该语句重复执行的次数。

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

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

计算时间复杂度最关键的基本操作。例如，在下列 3 个程序段中：

```
(1) {++x; s=0;}
(2) for (i=1; i <=n; ++i){ ++x; s+=x;}
(3) for (j=1; j<=n; ++j)
    for (k=1; k<=n; ++k) { ++x; s+=x;}
```

含基本操作“x 增 1”的语句的频度分别为 1、 n 和 n^2 ，则这 3 个程序段的时间复杂度分别为 $O(1)$ 、 $O(n)$ 和 $O(n^2)$ 。算法还可能呈现的时间复杂度有对数阶 $O(\log_2n)$ 、指数阶 $O(2^n)$ 等。

表 1-1 算法的时间量级分类表

名称	时间复杂度	说明
常量阶	$O(1)$	与问题规模无关的算法
线性阶	$O(n)$	与问题规模无相关的单重循环
平方阶	$O(n^2)$	与问题规模无相关的二重循环
立方阶	$O(n^3)$	与问题规模无相关的三重循环
指数阶	$O(e^n)$	较为复杂
对数阶	$O(\log_2n)$	折半查找算法
复合阶	如 $O(n\log_2n)$	堆排序算法
其他	不太确定	过于复杂

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

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

“交换序列中相邻两个整数”为基本操作。当 a 中初始序列为自小至大有序，基本操作的执行次数为 0；当初始序列为自大至小有序时，基本操作的执行次数为 $n(n-1)/2$ 。对这类算法的分析，一种解决的办法是计算它的平均值；即考虑它对所有可能的输入数据集的期望值，此时相应的时间复杂度为算法的**平均时间复杂度**。

很多情况下，各种输入数据集出现的概率难以确定，算法的平均时间复杂度难以确定，就会讨论算法的**最坏时间复杂度**，即分析最坏情况以估算算法执行时间的一个上界。

最好时间复杂度指的是最好情况下算法的时间复杂度。

时间复杂度的运算规则：

加法规则： $T(n)=T1(n)+T2(n)=O(f(n))+O(g(n))=O(\max(f(n), g(n)))$

乘法规则： $T(n)=T1(n) \times T2(n)=O(f(n)) \times O(g(n))=O(f(n) \times g(n))$

【习题演练】

1. 算法的时间复杂度取决于（ ）。

- A. 问题的规模
- B. 待处理数据的初态
- C. 计算机的配置
- D. A 和 B

三、算法的存储空间需求

类似于算法的时间复杂度，以**空间复杂度**（space complexity）作为算法所需存储空间的数量度，记作

$$S(n)=O(f(n))$$

其中 n 为问题的规模。一个上机执行的程序除了需要存储空间来寄存本身所用指令、常数、变量和输入数据外，也需要一些对数据进行操作的工作单元和存储一些为实现计算所需信息的辅助空间。若输入数据所占空间只取决于问题本身，和算法无关，则只需要分析除输入和程序之外的额外空间。

算法原地工作指的是额外空间相对于输入数据量来说是常数。

例如：将一维数组中的元素逆置存放，算法如下：

```
RevArray1(int a[], int n)
{
    int i, j, t;
    for(i=0, j=n-1; i<j; i++, j--)
    {
        t=a[i];
        a[i]=a[j];
        a[j]=t;
    }
}
```

分析该算法，由于基本语句就是循环内的交换语句，共执行了 $n/2$ 次，所以 $T(n) = n/2$

= $O(n)$ 。又因为算法的辅助空间只是 i 、 j 、 t ，3 个临时变量的空间，所以 $S(n) = 3 = O(1)$ 。而因为这里使用的辅助空间，不会随着问题的规模变化而变化，简而言之，就是常量级别的，我们就称其为原地工作。

【习题演练】

1. 试分析下面各程序段的时间复杂度。

- (1)

```
for (i=0; i<n; i++)  
    for (j=0; j<m; j++)  
        a[i][j]=0;
```
- (2)

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

```
x=0;  
for(i=1; i<n; i++)  
    for (j=1; j<=n-i; j++)  
        x++;
```
- (4)

```
x=n; //n>1  
y=0;  
while(x>=(y+1)*(y+1))  
    y++;
```

第二章 线性表

【本章综述】

线性表是考研的重中之重。常考算法设计题，基于顺序表或单链表，这类算法题实现起来比较容易而且代码量较少，但却要求具有最优的性能（时间复杂度和空间复杂度）。因此，应牢固掌握线性表的各种基本操作（基于两种存储结构），在平时的学习中应多注重积累和培养动手能力。

【复习重点】

（一）线性表

1. 线性表的概念和特征
2. 线性表的操作

（二）顺序表

1. 顺序表的定义
2. 在顺序表中查找、插入、删除，灵活运用

（三）链表

1. 链表的定义
2. 在单链表中查找、插入、删除、遍历，灵活运用
3. 循环链表及双向链表的定义、插入、删除

第一节 线性表的定义

一、线性表的基本概念

线性表（linear_list）是 n （ $n \geq 0$ ）个数据元素的有限序列。

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

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

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

则表中 a_{i-1} 领先于 a_i ， a_i 领先于 a_{i+1} ，称 a_{i-1} 是 a_i 的**直接前驱元素**， a_{i+1} 是 a_i 的**直接后继元**

素。当 $i=1, 2, \dots, n-1$ 时, a_i 有且仅有一个直接后继, 当 $i=2, 3, \dots, n$ 时, a_i 有且仅有一个直接前驱。

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

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

【习题演练】

1. 线性表 $L=(a_1, a_2, \dots, a_n)$, 下列说法正确的是 ()。
 - A. 每个元素都有一个直接前驱和一个直接后继
 - B. 线性表中至少有一个元素
 - C. 表中诸元素的排列必须是由小到大或由大到小
 - D. 除第一个和最后一个元素外, 其余每个元素都有一个且仅有一个直接前驱和直接后继。

二、线性表基本操作

初始化操作 $\text{InitList}(L)$: 其作用是建立一个空的线性表 L 。

求表长操作 $\text{ListLength}(L)$: 其作用是返回线性表的长度 L 。

元素定位操作 $\text{LocateElem}(L, x)$: 其作用是返回元素 x 在线性表 L 中第 1 次出现的位置, 若 x 存在, 则返回其位序, 否则返回 0。

取元素操作 $\text{GetElem}(L, i, e)$: 其作用是返回线性表 L 中的第 i 个元素, 并送给 e 。 i 的合理取值范围是 $1 \leq i \leq n$ 。

插入操作 $\text{ListInsert}(L, i, x)$: 其作用是在线性表 L 的第 i 个位置上插入一个值为 x 的元素。 i 的合理取值范围是 $1 \leq i \leq n+1$ 。

删除操作 $\text{ListDelete}(L, i, e)$: 其作用是删除线性表 L 中的第 i 个元素, 并送给 e 。 i 的合理取值范围是 $1 \leq i \leq n$ 。

输出操作 $\text{List}(L)$: 其作用是按先后顺序输出线性表 L 的所有元素。

除此之外, 还可进行一些更复杂的操作。例如, 将两个或两个以上的线性表合并成一个线性表; 把一个线性表拆开成两个或两个以上的线性表; 重新复制一个线性表等。

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

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

【例 1-2】已知线性表 LA 和 LB 中的数据元素按值非递减有序排列，现要求将 LA 和 LB 归并为一个新的线性表 LC，且 LC 中的数据元素仍按值非递减有序排列。例如，设

LA=(3,5,8,11)

LB=(2,6,8,9,11,15,20)

则

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

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

$$c = \begin{cases} a & a \leq b \\ b & a > b \end{cases}$$

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

```
void MergeList(List La, List Lb, List &Lc){
    // 已知线性表 La 和 Lb 中的数据元素按值非递减排列。
    // 归并 La 和 Lb 得到新的线性表 Lc。Lc 的数据元素也按值非递减排列。
    InitList(Lc);
    i=j=1; k=0;
```



```

    La_len=ListLength(La);
    Lb_len=ListLength(Lb);
    while((i<=La_len) &&(j<=Lb_len)){ //La 和 Lb 均非空
        GetElem(La, i, ai);
        GetElem(Lb, j, bj);
        if (ai<= bj) {
            ListInsert(Lc, ++k, ai);
            ++i; }
        else{
            ListInsert(Lc, ++k, bj);
            ++j; }
    }
    while(i<=La_len){
        GetElem(La, i++, ai);
        ListInsert(Lc, ++k, ai);
    }
    while(j<=Lb_len){
        GetElem(Lb, j++, bj);
        ListInsert(Lc, ++k, bj);
    }
} //MergeList

```

第二节 线性表的顺序表示和实现

一、顺序表的定义

线性表的顺序表示称作**顺序表**。它是用一组地址连续的存储单元依次存储线性表的数据元素。即以元素在计算机内“物理位置相邻”表示线性表中数据元素之间的逻辑关系。

假设线性表 (a_1, a_2, \dots, a_n) 的每个元素需占用 l 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置，且第 i 个元素 a_i 的存储地址用 $LOC(a_i)$ 表示。则线性表中第 $i+1$ 个数据元素的存储位置 $LOC(a_{i+1})$ 与第 i 个数据元素的存储位置 $LOC(a_i)$ 间满足下列关系。

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

一般来说，线性表的第 i 个数据元素 a_i 的存储位置为

$$LOC(a_i) = LOC(a_1) + (i-1) \times l \quad (1 \leq i \leq n)$$

式中 $LOC(a_1)$ 是线性表的第一个数据元素 a_1 的存储位置，称做线性表的**起始位置**或**基地址**。只要确定了存储线性表的起始位置，线性表中任一数据元素都可随机存取，所以线性表的顺序存储结构是一种**随机存取**的存储结构。

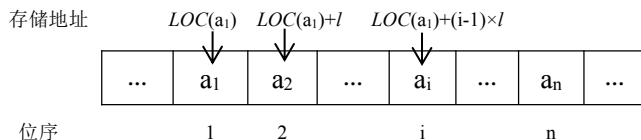


图 2-1 线性表的顺序存储结构示意图

通常用数组来描述数据结构中的顺序存储结构。在 C 语言中可用动态分配的一维数组，如下描述。

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

数组指针 `elem` 指示线性表的基地址，`length` 指示线性表的当前长度。`listsize` 指示顺序表当前分配的存储空间大小，一旦因插入元素而空间不足时，可进行再分配，即为顺序表增加一个大小为存储 `LISTINCREMENT` 个数据元素的空间。

【习题演练】

1. 顺序表中第一个元素的存储地址是 100，每个元素的长度为 2，则第 5 个元素的地址是（ ）。

- A. 110 B. 108 C. 100 D. 120

二、顺序表的基本操作

顺序表的“求表长”和“取第 i 个数据元素”的时间复杂度均为 $O(1)$ 。

1. 初始化

顺序表的初始化操作就是为顺序表分配一个预定义大小的数组空间，并将线性表的当前长度设为“0”。

```
Status InitList_ Sq(SqList &L){
    // 构造一个空的线性表 L。
```

```

L.elem=(ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
if(!L.elem) exit(-1);    // 存储分配失败
L.length=0;              // 空表长度为 0
L.listsize=LIST_INIT_SIZE;    // 初始存储容量
Return 1;                // 返回 1 表示操作成功
} // InitList_Sq

```

注意：C 语言中数组的下标从“0”开始，因此顺序表中第 i 个数据元素是 $L.elem[i-1]$ 。

2. 插入元素

线性表的插入操作是指在线性表的第 $i-1$ 个数据元素和第 i 个数据元素之间插入一个新的数据元素。一般情况下，在第 i ($1 \leq i \leq L.length$) 个元素之前插入一个元素时，需将第 $L.length$ 至第 i (共 $n-i+1$) 个元素向后移动一个位置，算法如下：

```

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

```

例如，图 2-2 表示一个线性表在进行插入操作的前、后，其数据元素在存储空间中的位置变化。为了在线性表的第 4 和第 5 个元素之间插入一个值为 25 的数据元素，则需将第 5 个至第 8 个数据元素依次往后移动一个位置。

在顺序表中某个位置插入一个数据元素时，其时间主要耗费在移动元素上，而移动元素的个数取决于插入元素的位置。

假设 p_i 是在第 i 个元素之前插入一个元素的概率，则在长度为 n 的线性表中插入一

个元素时所需移动元素次数的期望值（平均次数）为

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

不失一般性，假定在线性表的任何位置上插入元素都是等概率的，即

$$p_i = \frac{1}{n+1}$$

则：

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

在顺序表中插入一个数据元素，平均移动表中一半元素。若表长为 n ，则算法 ListInsert_Sq 的时间复杂度为 $O(n)$ 。

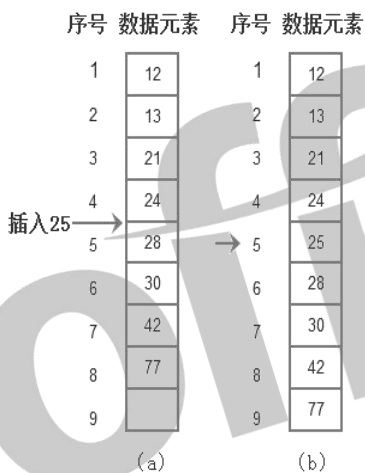


图 2-2 线性表插入前后的状况

(a) 插入前 $n=8$

(b) 插入后 $n=9$

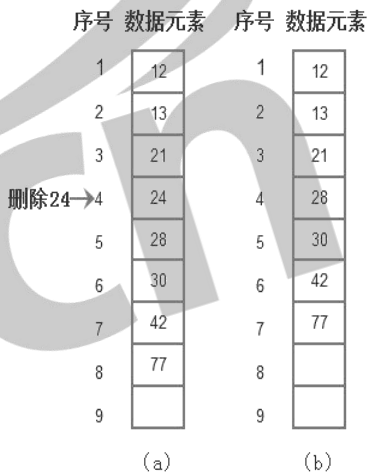


图 2-3 线性表删除前后的状况

(a) 删除前 $n=8$

(b) 删除后 $n=7$

3. 删除元素

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

```

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

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

如图 2-3 所示，为了删除第 4 个数据元素，必须将从第 5 个至第 8 个元素都依次往前移动一个位置。

在顺序表中某个位置删除一个数据元素时，其时间主要耗费在移动元素上，而移动元素的个数取决于删除元素的位置。

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

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

不失一般性，假定在线性表的任何位置上删除元素都是等概率的，即

$$q_i = \frac{1}{n}$$

则：

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

在顺序表中删除一个数据元素，平均约移动表中一半元素。若表长为 n ，则算法 ListDelete_Sq 的时间复杂度为 $O(n)$ 。

4. 按值查找

在顺序表 L 中查访是否存在和 e 相同的数据元素的方法是：令 e 和 L 中的数据元素逐个比较之，算法的时间复杂度为 $O(n)$ 。

```
int LocateElem_Sq(SqList L, ElemType e,
                  Status(*compare)(ElemType, ElemType)){
    // 在顺序线性表 L 中查找第 1 个值与 e 满足 compare() 的元素的位序
    // 若找到，则返回其在 L 中的位序，否则返回 0
    i=1;           // i 的初值为第 1 个元素的位序
    p=L.elem;      // p 的初值为第 1 个元素的存储位置
    while(i<=L.length && !(*compare)(*p++, e)) ++i;
    if(i<=L.length) return i;
    else return 0;
} // LocateElem_sq
```

5. 合并操作

```
void MergeList_Sq(SqList La, SqList Lb, SqList &Lc){
```

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

【习题演练】

- 在 n 个结点的顺序表中，算法的时间复杂度是 $O(1)$ 的操作是（ ）。
 - 访问第 i 个结点 ($1 \leq i \leq n$) 和求第 i 个结点的直接前驱 ($2 \leq i \leq n$)
 - 在第 i 个结点后插入一个新结点 ($1 \leq i \leq n$)
 - 删除第 i 个结点 ($1 \leq i \leq n$)
 - 将 n 个结点从小到大排序
- 向一个有 127 个元素的顺序表中插入一个新元素并保持原来顺序不变，平均要移动的元素个数为（ ）。

A. 8 B. 63.5 C. 63 D. 7

第三节 线性表的链式表示和实现

线性表的顺序存储结构的特点是逻辑关系上相邻的两个元素在物理位置上也相邻，因此可以随机存取表中任一元素，它的存储位置可用一个简单、直观的公式来表示。然而，这个特点也铸成了这种存储结构的弱点：在插入或删除元素时，需移动大量元素。

本节讨论线性表链式存储结构。

一、线性链表

1. 单链表的定义

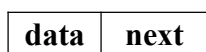
线性表的链式存储结构是用一组任意的存储单元存储线性表的数据元素（这组存储单元可以是连续的，也可以是不连续的）。每个数据元素 a_i 除了存储其本身的信息之外，还需存储一个指示其直接后继的信息（即直接后继的存储位置）。这两部分信息组成数据元素 a_i 的存储映像，称为**结点**（node）。它包括两个域：其中存储数据元素信息的域称为**数据域**；存储直接后继存储位置的域称为**指针域**。指针域中存储的信息称做**指针**或**链**。 n 个结点（ $a_i(1 \leq i \leq n)$ ）的存储映像）链结成一个**链表**，即为线性表的**链式存储结构**。由于此链表的每个结点中只包含一个指针域，故又称**线性链表**或**单链表**。

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

用线性链表表示线性表时，数据元素之间的逻辑关系是由结点中的指针指示的。换句话说，指针为数据元素之间的逻辑关系的映像，则逻辑上相邻的两个数据元素其存储的物理位置不要求紧邻，由此，这种存储结构为非顺序映像或链式映像。

单链表可由头指针唯一确定，在 C 语言中可用“结构指针”来描述线性表的单链表存储结构：

```
typedef struct LNode{
    ElemType    data;
    struct LNode *next;
}LNode, *LinkList;
```

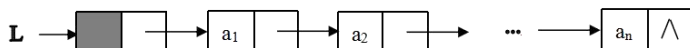


data: 数据域，存放结点的值

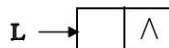
next: 指针域，存放结点的直接后继的地址

图 2-4 链表结点结构

假设 L 为单链表的头指针，它指向表中第一个结点。当线性表为“空”表时， L 为“空”($L=NULL$)。



(a) 非空表



(b) 空表

图 2-5 带头结点的单链表

在单链表的第一个结点之前附设一个结点，称之为**头结点**。头结点的数据域可以不存储任何信息，也可存储如线性表的长度等类的附加信息，头结点的指针域存储指向第

一个结点的指针（即第一个元素结点的存储位置）。如图 2-5（a）所示，单链表的头指针指向头结点。若线性表为空表，则头结点的指针域为“空”，如图 2-5（b）所示。

【习题演练】

- 链式存储的存储结构所占存储空间（ ）。
 - 分两部分，一部分存放结点值，另一部分存放表示结点间关系的指针
 - 只有一部分，存放结点值
 - 只有一部分，存储表示结点间关系的指针
 - 分两部分，一部分存放结点值，另一部分存放结点所占单元数
- 线性表若采用链式存储结构时，要求内存中可用存储单元的地址（ ）。
 - 必须是连续的
 - 部分地址必须是连续的
 - 一定是不连续的
 - 连续或不连续都可以
- 单链表的存储密度（ ）。
 - 大于 1
 - 等于 1
 - 小于 1
 - 不能确定

2. 单链表的操作

（1）取元素

在单链表中，取得第 i 个数据元素必须从头指针出发寻找。

```

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

算法的时间复杂度为 $O(n)$ 。

（2）插入元素

假设要在数据元素 a 和 b 之间插入一个数据元素 x ，已知 p 为单链表存储结构中指向结点 a 的指针。如图 2-6（a）所示。

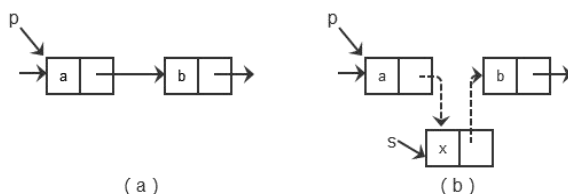


图 2-6 在单链表中插入结点时指针变化状况

为插入数据元素 x ，首先要生成一个数据域为 x 的结点，然后插入在单链表中。根据插入操作的逻辑定义，还需要修改结点 a 中的指针域，令其指向结点 x ，而结点 x 中的指针域应指向结点 b ，从而实现 3 个元素 a 、 b 和 x 之间逻辑关系的变化。插入后的单链表如图 2-6 (b) 所示。假设 s 为指向结点 x 的指针，则上述指针修改语句描述即为

$s \rightarrow \text{next} = p \rightarrow \text{next}; p \rightarrow \text{next} = s;$

以下是插入元素算法：

```

Status ListInsert_L(LinkList &L, int i, ElemType e){
    // 在带头结点的单链线性表 L 中第 i 个位置之前插入元素 e
    p=L; j=0;
    while (p&&j<i-1) {p=p->next; ++j; } // 寻找第 i-1 个结点
    if (! p|| j>i-1) return -1; // i 小于 1 或者大于表长加 1
    s=(LinkList) malloc (sizeof( LNode)) ; // 生成新结点
    s->data=e; s->next=p-> next; // 插入 L 中
    p->next= s;
    return 1;
} // ListInsert_L
    
```

(3) 删除元素

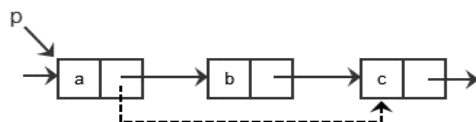


图 2-7 在单链表中删除结点时指针变化状况

如图 2-7 所示在线性表中删除元素 b 时，为在单链表中实现元素 a 、 b 和 c 之间逻辑关系的变化，仅需修改结点 a 中的指针域。假设 p 为指向结点 a 的指针，则修改指针的语句为

$p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next};$

以下是删除元素算法：

```

Status ListDelete_L(LinkList &L, int i, ElemType &e){
    // 在带头结点的单链线性表 L 中，删除第 i 个元素，并由 e 返回其值
    
```

```

p=L; j=0;
while (p->next&& j<i-1){ // 寻找第 i 个结点, 并令 p 指向其前趋
    p=p->next; ++j;
}
if(!(p->next) || j>i-1) return ERROR; // 删除位置不合理
q=p->next; p->next=q->next; // 删除并释放结点
e=q->data; free(q);
return OK;
} // ListDelete_L

```

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

在插入和删除算法中, 引用了 C 语言中的两个标准函数 `malloc` 和 `free`。`p=(LinkedList) malloc(sizeof(LNode))`的作用是由系统生成一个 `LNode` 型的结点, 同时将该结点的起始位置赋给指针变量 `p`; `free(q)`的作用是由系统回收一个 `LNode` 型的结点, 回收后的空间可以备作再次生成结点时用。因此, 单链表是一种动态结构。整个可用存储空间可为多个链表共享, 每个链表占用的空间不需预先分配划定, 而是可以由系统应需求即时生成。

(4) 逆向建立单链表操作 (即头插法)

建立线性表的链式存储结构的过程是一个动态生成链表的过程。即从“空表”的初始状态起, 依次建立各元素结点, 并逐个插入链表。下面算法是一个从表尾到表头逆向建立单链表的算法, 其时间复杂度为 $O(n)$ 。

```

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

```

(5) 合并操作

假设头指针为 `La` 和 `Lb` 的单链表分别为线性表 `LA` 和 `LB` 的存储结构, 现要归并 `La`

和 Lb 得到单链表 Lc，设立 3 个指针 pa、pb 和 pc，其中 pa 和 pb 分别指向 La 表和 Lb 表中当前待比较插入的结点，而 pc 指向 Lc 表中当前最后一个结点，若 $pa \rightarrow data \leq pb \rightarrow data$ ，则将 pa 所指结点链接到 pc 所指结点之后，否则将 pb 所指结点链接到 pc 所指结点之后。指针的初始状态为：当 LA 和 LB 为非空表时，pa 和 pb 分别指向 La 和 Lb 表中第一个结点，否则为空；pc 指向空表 Lc 中的头结点。由于链表的长度为隐含的，则第一个循环执行的条件是 pa 和 pb 皆非空，当其中一个为空时，说明有一个表的元素已归并完，则只要将另一个表的剩余段链接在 pc 所指结点之后即可。归并两个单链表的算法如下：

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

在归并两个链表为一个链表时，不需要另建新表的结点空间，而只需将原来两个链表中结点之间的关系解除，重新按元素值非递减的关系将所有结点链接成一个链表即可。

【习题演练】

- 线性表 L 在（ ）情况下适用于使用链式结构实现。
 - 需经常修改 L 中的结点值
 - 需不断对 L 进行删除插入
 - L 中含有大量的结点
 - L 中结点结构复杂
- 在单链表中，要将 s 所指结点插入到 p 所指结点之后，其语句应为（ ）。
 - $s \rightarrow next = p + 1; p \rightarrow next = s;$
 - $(*p).next = s; (*s).next = (*p).next;$
 - $s \rightarrow next = p \rightarrow next; p \rightarrow next = s \rightarrow next;$
 - $s \rightarrow next = p \rightarrow next; p \rightarrow next = s;$

二、循环链表

循环链表 (circular linked list) 是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。由此，从表中任一结点出发均可找到表中其他结点，如图 2-8 所示为单链的循环链表。

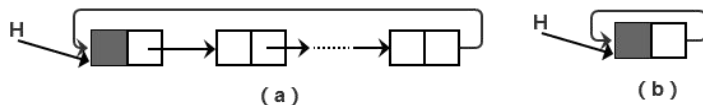


图 2-8 单循环链表

(a) 非空表； (b) 空表

循环链表的操作和线性链表基本一致，差别仅在于算法中的循环条件不是 p 或 $p \rightarrow \text{next}$ 是否为空，而是它们是否等于头指针。若在循环链表中设立尾指针而不设头指针，如图 2-9 (a) 所示，可使某些操作简化。例如将两个线性表合并成一个表时，仅需将一个表的表尾和另一个表的表头相接。当线性表以图 2-9 (a) 的循环链表作存储结构时，合并操作仅需改变两个指针值即可，运算时间为 $O(1)$ 。合并后的表如图 2-9 (b) 所示。

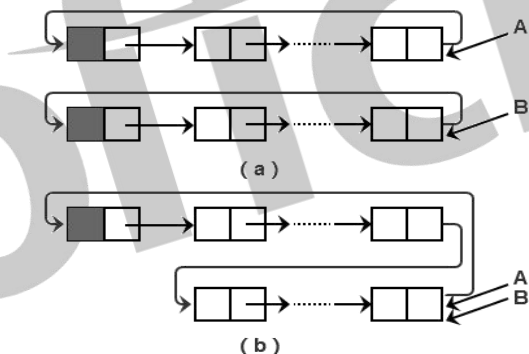


图 2-9 仅设尾指针的循环链表

(a) 两个链表； (b) 合并后的表

三、双向链表

在单链表中， NextElem 的执行时间为 $O(1)$ ，而 PriorElem 的执行时间为 $O(n)$ 。为克服单链表这种单向性的缺点，可利用**双向链表**(double linked list)。双向链表的结点中有两个指针域，其一指向直接后继，另一指向直接前趋，在 C 语言中如下描述线性表的双向链表存储结构：

```
typedef struct DuLNode{
    ElemType data;
    struct DuLNode *prior;
```

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

双向链表也可以有循环表，如图 2-10 (c) 所示，链表中存有两个环，图 2-10 (b) 所示为只有一个表头结点的空表。

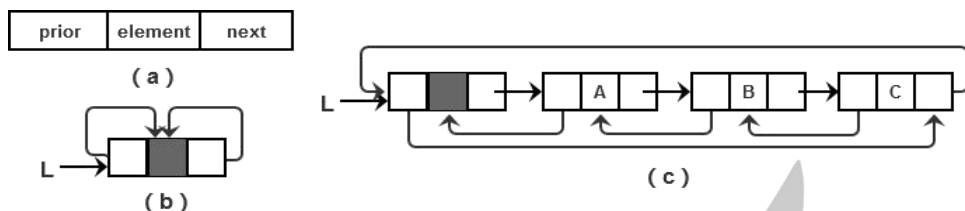


图 2-10 双向链表示例

(a) 结点结构 (b) 空的双向链表 (c) 非空的双向循环链表

在双向链表中，若 d 为指向表中某一结点的指针（即 d 为 $DuLinkList$ 型变量），则显然有

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

在双向链表中，有些操作如： $ListLength$ 、 $GetElem$ 和 $LocateElem$ 等仅需涉及一个方向的指针，则它们的算法描述和线性链表的操作相同，但在插入、删除时有很大的不同，在双向链表中需同时修改两个方向上的指针，图 2-11 和图 2-12 分别显示了删除和插入结点时指针修改的情况。

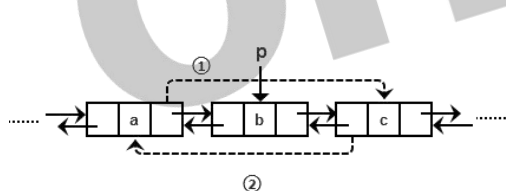


图 2-11 在双向链表中删除结点时指针变化状况

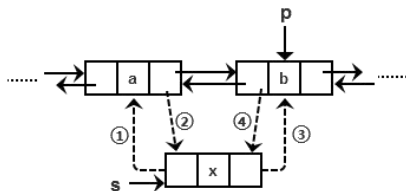


图 2-12 在双向链表中插入结点时指针变化状况

双链表的插入和删除算法分别如下，两者的时间复杂度均为 $O(n)$ 。

1. 插入

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

```
    return 1;
} // ListInsert_DuL
```

2. 删除

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

【习题演练】

- 在双向链表存储结构中, 删除 p 所指的结点时须修改指针 ()。
 - p->next->prior=p->prior; p->prior->next=p->next;
 - p->next=p->next->next; p->next->prior=p;
 - p->prior->next=p; p->prior=p->prior->prior;
 - p->prior=p->next->next; p->next=p->prior->prior;
- 在双向循环链表中, 在 p 指针所指的结点后插入 q 所指向的新结点, 其修改指针的操作是 ()。
 - p->next=q; q->prior=p; p->next->prior=q; q->next=q;
 - p->next=q; p->next->prior=q; q->prior=p; q->next=p->next;
 - q->prior=p; q->next=p->next; p->next->prior=q; p->next=q;
 - q->prior=p; q->next=p->next; p->next=q; p->next->prior=q;
- 带头结点的双循环链表 L 中只有一个元素结点的条件是 ()。
 - L->next->next=NULL B. L->next->next=L
 - L->next=NULL D. L->next=L

第三章 栈和队列

【本章综述】

本章内容通常以选择题的形式考查，题目不算难，但命题的形式比较灵活。其中栈（出入栈的过程、出栈序列的合法性）和队列的操作及其特征是重中之重，属每年必考内容。由于它们均是线性表的应用和推广，很容易出现在算法设计题中。此外，栈和队列的顺序存储结构、链式存储结构及其特点、双端队列的特点、栈和队列的常见应用，都是必须掌握的内容。

【复习重点】

（一）栈

1. 栈的概念、特点
2. 链栈和顺序栈
3. 入栈、出栈操作
4. 栈的应用

（二）队列

1. 队列的概念特点
2. 链队列和循环队列
3. 入队、出队操作
4. 队列的应用

第一节 栈

一、栈的定义

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

假设栈 $S=(a_1, a_2, \dots, a_n)$ ，则称 a_1 为栈底元素， a_n 为栈顶元素。栈中元素按 a_1, a_2, \dots, a_n 的次序进栈，退栈的第一个元素应为栈顶元素。即，栈的修改是按后进先出的原则进行的。因此，栈又称为**后进先出**（last in first out）的线性表（简称 **LIFO** 结构）。

栈的基本操作除了在栈顶进行插入或删除外，还有栈的初始化、判空及取栈顶元素等。其中插入元素的操作为**入栈**，删除栈顶元素的操作为**出栈**。

【习题演练】

1. 若让元素 1, 2, 3, 4, 5 依次进栈，则出栈次序不可能出现在（ ）种情况。
A. 5, 4, 3, 2, 1 B. 2, 1, 5, 4, 3
C. 4, 3, 1, 2, 5 D. 2, 3, 5, 4, 1
2. 若已知一个栈的入栈序列是 1, 2, 3, ..., n，其输出序列为 $p_1, p_2, p_3, \dots, p_n$ ，若 $p_1=n$ ，则 p_i 为（ ）。
A. i B. n-i C. n-i+1 D. 不确定

二、栈的表示和实现

和线性表类似，栈也有两种存储表示方法。

顺序栈：栈的顺序存储结构；

链栈：栈的链式存储结构。

1. 顺序栈

顺序栈，即栈的顺序存储结构，是利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针 top 指示栈顶元素在顺序栈中的位置。栈的 2 种基本形态：栈空 $top=0$ ；栈满 $top = MAXSIZE$ 。

顺序栈的定义如下：

```
#define MAX_SIZE 50 //定义栈中元素的最大个数
typedef struct {
    ElemType elem[MAX_SIZE]; //存放栈中元素
    int top; //栈顶指针
} SqStack;
```

顺序栈的基本操作如下：

(1) 初始化

```
InitStack(SqStack &s){
    s.top = 0;
}
```

(2) 入栈


```
bool Push ( SqStack &s,  ElemType x )
{
    if ( s.top == MAXSIZE )  return false;  // 栈满
    s.elem[top] = x;          // 存入栈顶位置
    s.top++;                  // 栈顶指针加 1
    return true;
}
```

(3) 出栈

```
bool Pop ( SqStack &s, ElemType &x )
{
    if ( s.top==0 )  return false;  // 栈空
    x = s.elem[s.top--];          // 栈顶元素出栈，指针减 1
    return true;
}
```

(4) 判空

```
bool IsEmpty(SqStack s)
{
    if(s.top ==0)
        return true;
    else
        return false;
}
```

(5) 栈读取栈顶元素

```
bool GetTop(SqStack s, ElemType &x )
{
    if(s.top ==0)
        return false;
    x = s.elem[s.top];
    return true;
}
```

同学们在学习其他教材时，顺序栈的实现方式会略有差异，但不要担心，万变不离其宗。关键是理解栈的实质。

2. 链栈

链栈，如图 3-1 所示。栈的操作是线性表操作的特例，则链栈的操作易于实现。

链栈存储结构：用不带头结点的单链表实现。类型定义同单链表。

链栈的基本形态：

栈空 条件: $S == \text{NULL}$

栈满 (一般不出现)

链栈的基本操作如下:

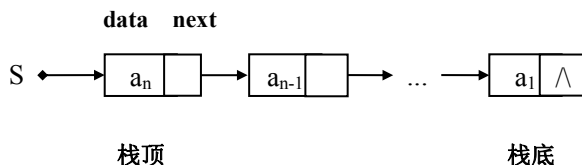


图 3-1 链栈示意图

(1) 入栈

bool Push (LinkList &s, DataType x)

```
{
    // 新建结点
    Linklist p = (LinkList) malloc (sizeof(LNode));
    if ( !p ) return false; // 失败
    p->data = x;
    // 插入栈顶
    p->next = s;
    s = p;
    return true;
}
```

(2) 出栈

bool Pop (LinkList &s, DataType &x)

```
{
    if ( s==NULL ) return false; // 栈空
    // 删除栈顶元素
    p = s;
    s = s->next;
    x = p->data;
    free ( p );
    return true;
}
```

(3) 取栈顶元素

bool Top (LinkList &s, DataType &x)

```
{
    if ( s==NULL ) return false; // 栈空
    x = s->data;
    return true;
}
```

}

【习题演练】

1. 链式栈结点为: (data, link), top 指向栈顶。若想摘除栈顶结点, 并将删除结点的值保存到 x 中, 则应执行操作 ()。

A. $x=top \rightarrow data; top=top \rightarrow link;$ B. $top=top \rightarrow link; x=top \rightarrow link;$

C. $x=top; top=top \rightarrow link;$ D. $x=top \rightarrow link;$

2. 若一个栈以向量 $V[1..n]$ 存储, 初始栈顶指针 top 设为 $n+1$, 则元素 x 进栈的正确操作是 ()。

A. $top++; V[top]=x;$ B. $V[top]=x; top++;$

C. $top--; V[top]=x;$ D. $V[top]=x; top--;$

第二节 栈的应用

一、数值转换

十进制数 N 和其他 d 进制数的转换解决方法很多, 其中一个简单算法基于下列原理:

$$N = (N \text{ div } d) \times d + N \text{ mod } d$$

其中: div 为整除运算, mod 为求余运算。

例如:

$(1348)_{10} = (2504)_8$, 其运算过程如下:

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

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

在这个例子中，栈操作的序列是直线式的，即先一味地入栈，然后一味地出栈。

二、括号匹配检验

例：检查表达式中的括号是否正确匹配。如{()[]}正确，([])]则错误。

分析：每个左括号都“期待”对应的右括号，匹配成功则可以消去。

思路：遇到左括号则入栈，遇到右括号则与栈顶括号相比较，如果匹配则消去，否则匹配失败。当然，如果栈中没有括号可以匹配，或者最后栈中还有未匹配的左括号，也都是匹配错误。

【习题演练】

1. 设计一个判别表达式中左、右括号是否配对出现的算法，采用（ ）数据结构最佳。

A. 线性表的顺序存储结构

B. 队列

B. 线性表的链式存储结构

D. 栈

三、栈与递归

栈在程序设计语言中实现递归。一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称作递归函数。

它通常把一个大型的复杂问题，层层转化为一个与原问题相似的规模较小的问题来求解，递归策略只需少量的代码就可以描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。但在通常情况下，它的效率并不是太高。

必须注意的是递归模型不能是循环定义的，其必须满足下面的两个条件：

(1) 递归表达式（递归体）。

(2) 边界条件（递归出口）。

递归的精髓在于能否将原始问题转化为属性相同单规模较小的问题。

在递归调用的过程中，系统为每一层的返回点、局部变量、传入实参等开辟了递归工作栈来进行数据存储，递归次数过多容易造成栈溢出等。而其效率不高的原因是递归调用过程中包含很多重复的计算。优点是代码简单，容易理解。

可以将递归算法转化为非递归算法，通常需要借助栈来实现这种转换。

【习题演练】

1. 栈在（ ）中有所应用。
A. 递归调用 B. 函数调用
C. 表达式求值 D. 前三个选项都有
2. 一个递归算法必须包括（ ）。
A. 递归部分 B. 终止条件和递归部分
C. 迭代部分 D. 终止条件和迭代部分

第三节 队列

一、队列的定义

队列是一种先进先出（first in first out，缩写 FIFO）的线性表，只允许在表的一端进行插入，在另一端删除元素，最早进入队列的元素最早离开，允许插入的一端叫做队尾（rear），允许删除的一端叫做队头（front）。队列的示意图如下：

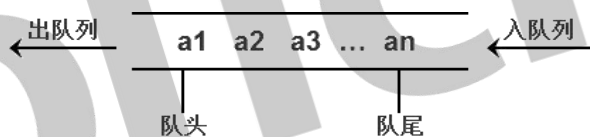


图 3-2 队列示意图

队列可以有两种存储表示：链队列和循环队列。

【习题演练】

1. 栈和队列的主要区别在于（ ）。
A. 它们的逻辑结构不一样 B. 它们的存储结构不一样
B. 所包含的元素不一样 D. 插入和删除操作的限定不一样
2. 队列的“先进先出”特性是指（ ）。
I. 最后插入队列中的元素总是最后被删除。
II. 当同时进行插入、删除操作时，总是插入操作优先。
III. 每当有删除操作时，总要先做一次插入操作。
IV. 每次从队列中删除的总是最早的元素。
A. 仅 I B. I 和 IV C. II 和 III D. IV

3. 设栈 S 和队列 Q 的初始状态为空, 元素 e1、e2、e3、e4、e5 和 e6 依次进入栈 S, 一个元素出栈后即进入 Q, 若 6 个元素出队的序列是 e2、e4、e3、e6、e5 和 e1, 则栈 S 的容量至少应该是 ()。

A. 2

B. 3

C. 4

D. 6

二、队列的链式表示和实现

用链表表示的队列简称为**链队列**。如图 3-3 所示。一个链队列需要两个分别指示队头和队尾的指针 (分别称为头指针和尾指针) 才能惟一确定。为了操作方便起见, 我们也给链队列添加一个头结点, 并令头指针指向头结点。由此, 空的链队列的判决条件为头指针和尾指针均指向头结点, 如图 3-4 (a) 所示。

链队列的操作即为单链表的插入和删除操作的特殊情况, 只是尚需修改尾指针或头指针, 图 3-4 (b) ~ (d) 展示了这两种操作进行时指针变化的情况。

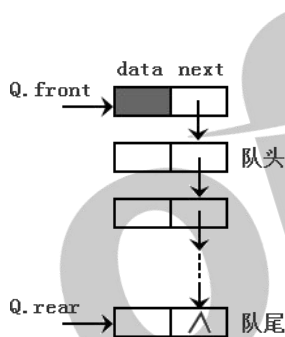


图 3-3 链队列示意图

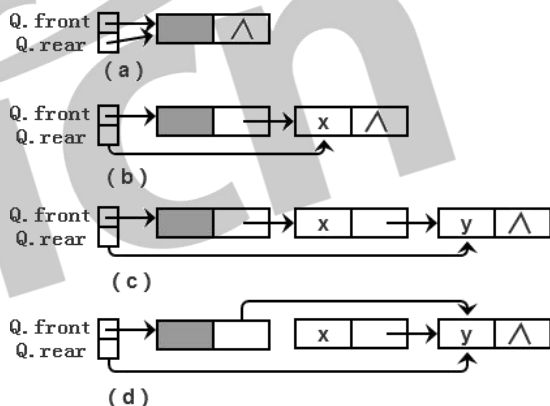


图 3-4 队列运算指针变化

(a) 空队列; (b) 元素 x 入队列; (c) 元素 y 入队列; (d) 元素 x 出队列

下面给出链队列的链式存储结构说明。

```
typedef struct QNode{
    QElemType data;
    struct QNode *next;
}QNode,*QueuePtr;
typedef struct Queue{
    QueuePtr front;
    QueuePtr rear;
}LinkQueue;
```

队列的基本操作算法:

1. 初始化

```
Status InitQueue(LinkQueue &Q){  
    //构造一个空队列  
    Q.front =Q.rear=(QueuePtr)malloc(sizeof(QNode));  
    If(!Q.front) exit(OVERFLOW); //存储分配失败  
    Q.front->next = NULL;  
    return OK;  
}
```

2. 入队

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

3. 出队

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

4. 销毁

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

}

在上述模块的算法描述中，应注意删除队列头元素算法中的特殊情况。一般情况下，删除队列头元素时仅需修改头结点中的指针，但当队列中最后一个元素被删后，队列尾指针也丢失了，因此需对队尾指针重新赋值（指向头结点）。

【习题演练】

1. 用链接方式存储的队列，在进行删除运算时（ ）。
 - A. 仅修改头指针
 - B. 仅修改尾指针
 - C. 头、尾指针都要修改
 - D. 头、尾指针可能都要修改

三、队列的顺序表示和实现

在队列的顺序存储结构中，除了用一组地址连续的存储单元依次存放从队列头到队列尾的元素之外，尚需附设两个指针 `front` 和 `rear` 分别指示队列头元素及队列尾元素的位置。

为了在 C 语言中描述方便起见，在此我们约定：初始化建空队列时，令 `front=rear=0`，每当插入新的队列尾元素时，“尾指针增 1”；每当删除队列头元素时，“头指针增 1”。因此，在非空队列中，头指针始终指向队列头元素，而尾指针始终指向队列尾元素的下一个位置，如图 3-5 所示。

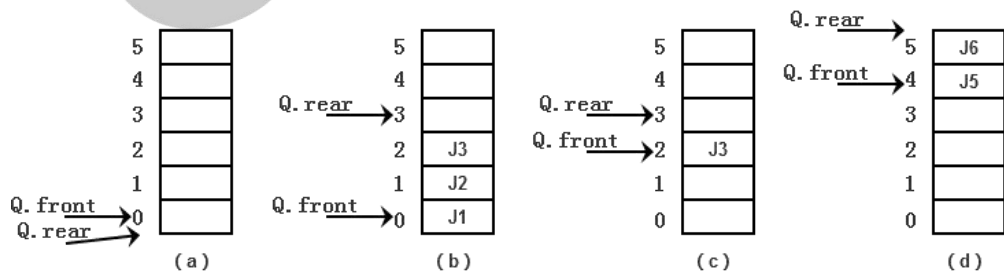


图 3-5 头、尾指针和队列中元素之间的关系

(a) 空队列；(b) J1、J2 和 J3 相继入队列；

(c) J1 和 J2 相继被删除；(d) J4、J5 和 J6 相继插入队列之后，J3 及 J4 相继被删除

假设当前为队列分配的最大空间为 6，则当队列处于图 3-5 (d) 的状态时不可再继续插入新的队尾元素，否则会因数组越界而遭致程序代码被破坏。然而此时又不宜如顺序栈那样，进行存储再分配扩大数组空间，因为队列的实际可用空间并未占满。一个较巧妙的办法是将顺序队列臆造为一个环状的空间，如图 3-6 所示，称之为**循环队列**。

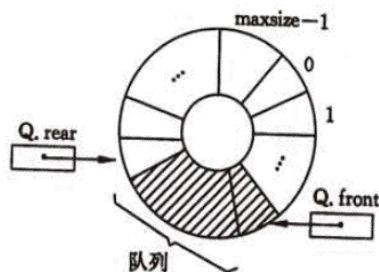


图 3-6 循环队列示意图

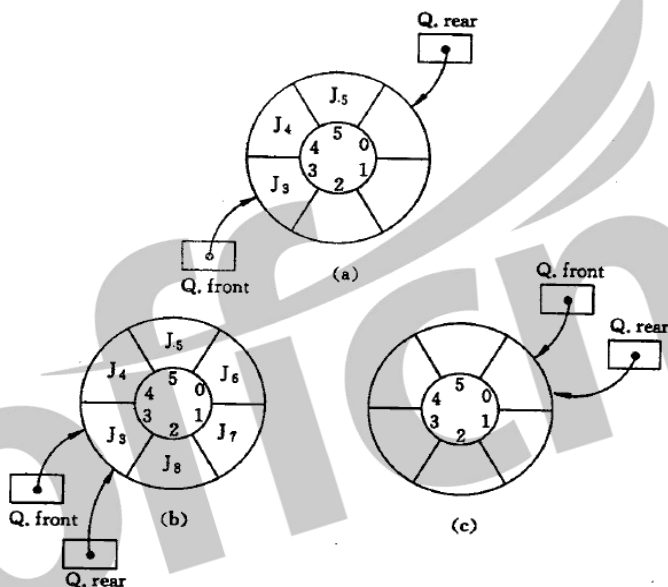


图 3-7 循环队列的头尾指针

(a) 一般情况；(b) 队列满时；(c) 空队列

指针和队列元素之间关系不变，如图 3-7 (a) 所示循环队列中，队列头元素是 J_3 ，队列尾元素是 J_5 ，之后 J_6 、 J_7 和 J_8 相继插入，则队列空间均被占满，如图 3-7 (b) 所示，此时 $Q.front=Q.rear$ ；反之，若 J_3 、 J_4 和 J_5 相继从图 3-7 (a) 的队列中删除，使队列呈“空”的状态，如图 3-7 (c) 所示。此时也存在关系式 $Q.front=Q.rear$ ，由此可见，只凭等式 $Q.front=Q.rear$ 无法判别队列空间是“空”还是“满”。可有两种处理方法：其一是另设一个标志位以区别队列是“空”还是“满”；其二是少用一个元素空间，约定以“队列头指针在队列尾指针的下一位置（指环状的下一位置）上”作为队列呈“满”状态的标志。

在 C 语言中不能用动态分配的一维数组来实现循环队列。如果用户的应用程序中有循环队列，则必须为它设定一个最大队列长度；若用户无法预估所用队列的最大长度，则宜采用链队列。

循环队列-队列的顺序存储结构说明如下:

```
#define MAXSIZE 100
```

```
typedef struct{
    QElemType    *base;
    int front;
    int rear;
}SqQueue;
```

循环队列的基本操作:

注意: 此处少用一个元素空间来区分队列空和满。

1. 初始化

```
bool InitQueue ( SqQueue &Q)
{ // 构造空队列
    Q.base =(QElemType *)malloc(MAXSIZE * sizeof(QElemType));
    if (! Q.base ) return false;
    Q.front= Q.rear=0;
    return true;
}
```

2. 求队列的长度

```
int QueueLength ( SqQueue Q)
{
    return (Q.rear - Q.front + MAXSIZE)%MAXSIZE;
}
```

3. 入队

```
bool EnQueue ( SqQueue &Q, QElemType x )
{
    if ( (Q.rear+1)%MAXSIZE==Q.front ) return false; // 队列满
    Q.elem [Q.rear] = x; // 入队列
    Q.rear = (Q.rear+1)%MAXSIZE;
    return true;
}
```

4. 出队

```
bool DeQueue ( SqQueue &Q, QElemType &x )
{
    if ( Q.front==Q.rear ) return false; // 队列空
    x = Q.elem [Q.front]; // 出队列
    Q.front = (Q.front+1)%MAXSIZE;
    return true;
}
```

}

【习题演练】

1. 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. $(\text{rear}+1)\%n == \text{front}$ B. $\text{rear} == \text{front}$
C. $\text{rear}+1 == \text{front}$ D. $(\text{rear}-1)\%n == \text{front}$

2. 数组 $Q[n]$ 用来表示一个循环队列， f 为当前队列头元素的前一位置， r 为队尾元素的位置，假定队列中元素的个数小于 n ，计算队列中元素个数的公式为（ ）。

- A. $r-f$ B. $(n+f-r)\%n$ C. $n+r-f$ D. $(n+r-f)\%n$

3. 循环队列存储在数组 $A[0..m]$ 中，则入队时的操作为（ ）。

- A. rear=rear+1
B. rear=(rear+1)%(m-1)
C. rear=(rear+1)%m
D. rear=(rear+1)%(m+1)

第四节 队列的应用

一、队列在层次遍历中的应用

在信息处理中一大类的问题需要逐层或逐行处理。这种问题的解决方法往往是在处理当前层或当前行时就对下一层或下一行做预处理，把处理顺序安排好，待当前层或当前行处理完毕，就可以处理下一层或下一行。使用队列是为了保存下一步的处理顺序。比如后序章节中将要学到的二叉树的层次遍历和图的广度优先搜索都使用了队列。

二、队列在计算机系统中的应用

队列在计算机系统中的应用非常广泛，以下仅从两个方面来简述队列在计算机系统中的作用：第一个方面是**解决主机与外部设备之间不匹配**的问题，第二方面是**解决由多用户引起的资源竞争问题**。

对于第一个方面，仅以主机和打印机之间速度不匹配的问题为例作简要说明。主机输出数据给打印机打印，输出数据的速度比打印数据的速度快的多，由于速度不匹配，若直接把输出的数据送给打印机打印显然是不行的。解决方法是设置一个打印数据缓冲区，主机把要打印输出的数据依次写入到这个缓冲区中，写满后就暂停输出，转去做别的事情。打印机就从缓冲区中按照先进先出的原则依次取出数据并打印，打印完后再向主机发出请求。主机接到请求后再向缓冲区写入打印数据。这样做既保证了打印数据的

正确，又使主机提高了效率。由此可见，打印数据缓冲区中所存储的数据就是一个队列。

对于第二方面，CPU（即中央处理器，它包括运算器和控制器）资源的竞争就是一个典型的例子。在一个带有多终端的计算机系统上，有多个用户需要 CPU 各自运行自己的程序，它们分别通过各自的终端向操作系统提出占用 CPU 的请求。操作系统通常按照每个请求在时间上的先后顺序，把它们排成一个队列，每次把 CPU 分配给队首请求的用户使用。当相应的程序运行结束或用完规定的时间间隔后，则令其出队，再把 CPU 分配给新的队首的用户使用。这样既满足了每个用户的请求，又使 CPU 能够正常运行。

【习题演练】

1. 为解决计算机主机与打印机之间速度不匹配的问题，通常设置一个打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构是（ ）。

- A. 栈 B. 队列 C. 树 D. 图

第四章 数组

【本章综述】

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

无论是统考还是非统考，数组（包括特殊矩阵）属于考研重点内容，主要的考查的知识点是数组元素（矩阵元素）地址的计算。广义表知识点不在统考大纲要求范围之内，但是非统考的院校常常涉及这部分知识，一般考查广义表的定义、特征、表头、表尾、长度、深度。

【复习重点】

（一）数组

1. 数组的定义及其存储
2. 数组元素的地址计算

（二）矩阵

1. 矩阵的压缩存储
2. 矩阵元素的地址计算

第一节 数组

一、数组的定义

数组是由 n ($n \geq 1$) 个相同类型的数据元素构成的有限序列，每个数据元素称为一个数组元素，每个元素受 n 个线性关系的约束，每个元素在 n 个线性关系中的序号称为该元素的下标，并称该数组为 n 维数组。

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

考研常考的两种数组：

一维数组

(a_1, a_2, \dots, a_n)

二维数组

$$\begin{aligned} &[(a_{0,0}, a_{0,1}, \dots, a_{0,n-1}), \\ &(a_{1,0}, a_{1,1}, \dots, a_{1,n-1}), \\ &\dots, \\ &(a_{n-1,0}, a_{n-1,1}, \dots, a_{n-1,n-1})] \end{aligned}$$

可见二维数组是元素是一维数组的一维数组。

二、数组的存储表示

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

由于存储单元是一维的结构，而数组是个多维的结构，则用一组连续存储单元存放数组的数据元素需要约定次序。二维数组可有两种存储方式：一种以列序为主序的存储方式，一种是以行序为主序的存储方式。

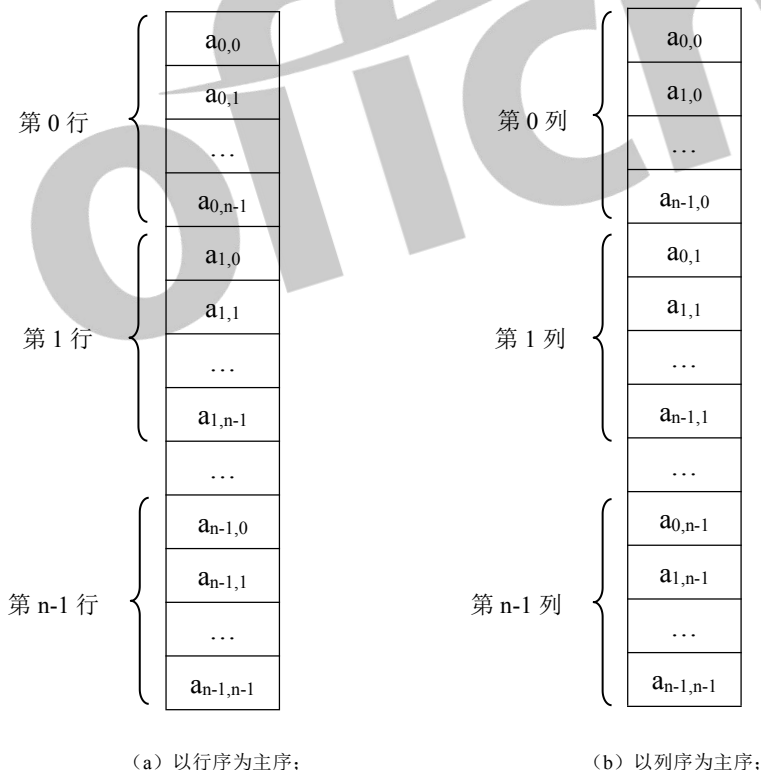


图 4-1 二维数组的两种存储方式

对于数组，一旦规定了它的维数和各维的长度，便可为它分配存储空间。反之，只要给出一组下标便可求得相应数组元素的存储位置。以一维数组 $A[0 \dots n-1]$ 为例，其存储结构关系式为：

$$\text{LOC}(a_i) = \text{LOC}(a_0) + (i) \times l \quad (0 \leq i \leq n)$$

其中 l 是各个数组元素所占的存储单元。

假设二维数组 A 的行下标与列下标的范围分别为 $[l_1, h_1]$ 与 $[l_2, h_2]$ ，每个数据元素占 L 个存储单元，则任一元素 a_{ij} 按行优先存储的结构关系式为：

$$\text{LOC}(a_{ij}) = \text{LOC}(a_{l_1, l_2}) + [(i - l_1) \times (h_2 - l_2 + 1) + (j - l_2)] \times L$$

式中， $\text{LOC}(a_{ij})$ 是 a_{ij} 的存储位置； $\text{LOC}(a_{l_1, l_2})$ 是 a_{l_1, l_2} 的存储位置，即二维数组 A 的起始存储位置，也称为**基地址**或**基址**。

当 l_1 和 l_2 的值均为 0，则上式变为

$$\text{LOC}(a_{ij}) = \text{LOC}(a_{0,0}) + [i \times (h_2 + 1) + j] \times L$$

按列优先存储的结构关系式为：

$$\text{LOC}(a_{ij}) = \text{LOC}(a_{l_1, l_2}) + [(j - l_2) \times (h_1 - l_1 + 1) + (i - l_1)] \times L$$

式中， $\text{LOC}(a_{ij})$ 是 a_{ij} 的存储位置； $\text{LOC}(a_{l_1, l_2})$ 是 a_{l_1, l_2} 的存储位置。

综上，数组元素的存储位置是其下标的线性函数，由于计算各个元素存储位置的时间相等，所以存取数组中任一元素的时间也相等。我们称具有这一特点的存储结构为**随机存储结构**。

【习题演练】

1. 假设以行序为主序存储二维数组 $A = \text{array}[1..100, 1..100]$ ，设每个数据元素占 2 个存储单元，基地址为 10，则 $\text{LOC}[5, 5] = ()$ 。

- A. 808 B. 818 C. 1010 D. 1020

2. 设有数组 $A[i, j]$ ，数组的每个元素长度为 3 字节， i 的值为 1 到 8， j 的值为 1 到 10，数组从内存首地址 BA 开始顺序存放，当用以列为主存放时，元素 $A[5, 8]$ 的存储首地址为 $()$ 。

- A. $BA + 141$ B. $BA + 180$ C. $BA + 222$ D. $BA + 225$

3. 二维数组 A 的每个元素是由 10 个字符组成的串，其行下标 $i = 0, 1, \dots, 8$ ，列下标 $j = 1, 2, \dots, 10$ 。若 A 按行先存储，元素 $A[8, 5]$ 的起始地址与当 A 按列先存储时的元素 $()$ 的起始地址相同。设每个字符占一个字节。

- A. $A[8, 5]$ B. $A[3, 10]$ C. $A[5, 8]$ D. $A[0, 9]$

4. 数组 $A[0..4, -1..-3, 5..7]$ 中含有元素的个数 $()$ 。

- A. 55 B. 45 C. 36 D. 16

三、矩阵的压缩存储

一般可以都用二维数组来存储矩阵元。然而有一些矩阵中有许多值相同的元素或者零元素。有时为了节省存储空间，可以对这类矩阵进行压缩存储。所谓压缩存储是指：为多个值相同的元只分配一个存储空间；对零元不分配空间。

特殊矩阵：是指非零元素或零元素的分布有一定规律的矩阵。反之，称为稀疏矩阵。下面分别讨论它们的压缩存储。

1. 对称矩阵

若一个 n 阶方阵 $A=(a_{ij})_{n \times n}$ 中的元素满足性质：

$$a_{ij}=a_{ji} \quad 1 \leq i, j \leq n \text{ 且 } i \neq j$$

则称 A 为 n 阶对称矩阵，如图 4-2 所示。

$$A = \begin{bmatrix} 1 & 5 & 1 & 3 & 7 \\ 5 & 0 & 8 & 0 & 0 \\ 1 & 8 & 9 & 2 & 6 \\ 3 & 0 & 2 & 5 & 1 \\ 7 & 0 & 6 & 1 & 3 \end{bmatrix} \quad A = \begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \dots & \dots & \dots & \dots & \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{bmatrix}$$

图 4-2 对称矩阵示例

对称矩阵中的元素关于主对角线对称，每一对对称元素 a_{ij} 和 $a_{ji}(i \neq j)$ 分配一个存储空间，则 n^2 个元素压缩存储到 $n(n+1)/2$ 个存储空间，能节约近一半的存储空间。

假设按“行优先顺序”存储下三角形(包括对角线)中的元素。

设用一维数组（向量） $sa[0 \dots n(n+1)/2-1]$ 存储 n 阶对称矩阵，如图 4-3 所示。为了便于访问，必须找出矩阵 A 中的元素的下标值(i, j)和向量 $sa[k]$ 的下标值 k 之间的对应关系。

K	0	1	2	3	4	5	n(n+1)/2-1		
sa	a ₁₁	a ₂₁	a ₂₂	a ₃₁	a ₃₂	a ₃₃	...	a _{n1}	a _{n2}	...	a _{nn}

图 4-3 对称矩阵的压缩存储示例

若 $i \geq j$ ，即 a_{ij} 在下三角区域中，要将 a_{ij} 直接保存在 sa 中。按行优先存储，则 a_{ij} 之前的 $i-1$ 行共有元素个数： $1+2+\dots+(i-1)=i \times (i-1)/2$ ，而在第 i 行有 $j-1$ 个元素，因此，元素 a_{ij} 保存在向量 sa 中时的下标值 k 之间的对应关系是：

$$k = i \times (i-1)/2 + j - 1 \quad i \geq j$$

若 $i < j$ ，即 a_{ij} 是在上三角区域中。因为 $a_{ij}=a_{ji}$ ，在向量 sa 中保存的是 a_{ji} 。依上述分析可得：

$$k = j \times (j-1) / 2 + i - 1 \quad i < j$$

对称矩阵元素 a_{ij} 保存在向量 sa 中时的下标值 k 与 (i,j) 之间的对应关系是:

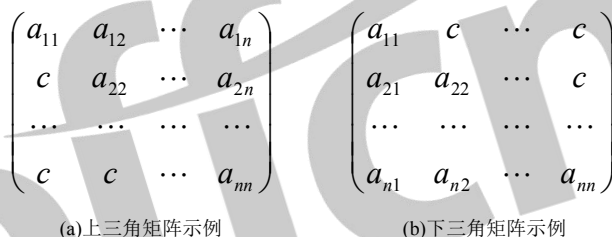
$$k = \begin{cases} i \times (i-1) / 2 + j - 1 & i \geq j \\ j \times (j-1) / 2 + i - 1 & i < j \end{cases} \quad 1 \leq i, j \leq n$$

根据上述的下标对应关系, 对于矩阵中的任意元素 a_{ij} , 均可在一维数组 sa 中唯一确定其位置 k ; 反之, 对所有 $k=1, 2, \dots, n(n+1)/2$, 都能确定 $sa[k]$ 中的元素在矩阵中的位置 (i,j) 。

2. 三角矩阵

以主对角线划分, 三角矩阵有上三角和下三角两种。

上三角矩阵的下三角 (不包括主对角线) 中的元素均为常数 c (一般为 0)。下三角矩阵正好相反, 它的主对角线上方均为常数, 如图 4-4 所示。



(a) 上三角矩阵示例
(b) 下三角矩阵示例

图 4-4 三角矩阵示例

三角矩阵中的重复元素 c 可共享一个存储空间, 其余的元素正好有 $n(n+1)/2$ 个, 因此, 三角矩阵可压缩存储到向量 $sa[0 \dots n(n+1)/2]$ 中, 其中 c 存放在向量的最后 1 个分量中。

上三角矩阵元素 a_{ij} 保存在向量 sa 中时的下标值 k 与 (i,j) 之间的对应关系是:

$$k = \begin{cases} (i-1) \times (2n-i+2) / 2 + j - 1 & i \leq j \\ n \times (n+1) / 2 & i > j \end{cases} \quad 1 \leq i, j \leq n$$

下三角矩阵元素 a_{ij} 保存在向量 sa 中时的下标值 k 与 (i,j) 之间的对应关系是:

$$k = \begin{cases} i \times (i-1) / 2 + j - 1 & i \geq j \\ n \times (n+1) / 2 & i < j \end{cases} \quad 1 \leq i, j \leq n$$

3. 对角矩阵

对角矩阵中, 除了主对角线和主对角线上或下方若干条对角线上的元素之外, 其余元素皆为零。即所有的非零元素集中在以主对角线为中心的带状区域中, 如图 4-5 所示为三对角矩阵。

图 4-5 三对角矩阵，非零元素仅出现在主对角 (a_{ii} , $1 \leq i \leq n$) 上、主对角线上的那条对角线 ($a_{i+1,i}$, $1 \leq i \leq n-1$)、主对角线下的那条对角线上 ($a_{i,i+1}$, $1 \leq i \leq n-1$)。当 $|i-j| > 1$ 时，元素 $a_{ij}=0$ 。

由此可知，一个 k (k 为奇数) 对角矩阵 A 是满足下述条件：当 $|i-j| > (k-1)/2$ 时， $a_{ij}=0$ 。

对角矩阵可按行优先顺序，将其压缩存储到一个向量中，并且也能找到每个非零元素和向量下标的对应关系。

仍然以三对角矩阵为例讨论。

当 $i=1$, $j=1, 2$, 或 $i=n$, $j=n-1, n$ 或 $1 < i < n-1, j=i-1, i, i+1$ 的元素 a_{ij} 外，其余元素都是 0。对这种矩阵，当以按“行优先顺序”存储时，第 1 行和第 n 行是 2 个非零元素，其余每行的非零元素都要是 3 个，则需存储的元素个数为 $3n-2$ 。

$$A = \begin{pmatrix} a_{11} & a_{12} & & & & & & & & & \\ a_{21} & a_{22} & a_{23} & & & & & & & & 0 \\ & a_{32} & a_{33} & a_{34} & & & & & & & \\ & & \dots & \dots & \dots & & & & & & \\ 0 & & & & a_{n-1n-2} & a_{n-1n-2} & a_{n-1n} & & & & \\ & & & & & a_{nn-1} & a_{nn} & & & & \end{pmatrix}$$

图 4-5 三对角矩阵示例

K	0	1	2	3	4	5	6	7	...	3n-4	3n-3
sa	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	...	a_{nn-1}	a_{nn}

图 4-6 三对角矩阵的压缩存储示例

如图 4-6 所示三对角矩阵的压缩存储形式。数组 sa 中的元素 $sa[k]$ 与三对角矩阵中的元素 a_{ij} 存在一一对应关系，在 a_{ij} 之前有 $i-1$ 行，共有 $3 \times (i-1) - 1$ 个非零元素，在第 i 行，有 $j-i+1$ 个非零元素，这样，非零元素 a_{ij} 的地址为：

$$LOC[a_{ij}] = LOC[a_{11}] + [3 \times (i-1) - 2 + (j-i+1) - 1] = LOC[a_{11}] + (2 \times i + j - 3)$$

上例中， a_{34} 对应着 $sa[10]$, $k=2 \times i + j - 3 = 2 \times 3 + 4 - 3 = 7$

故： $k=2 \times i + j - 3$

上述各种特殊矩阵，其非零元素的分布都是有规律的，因此总能找到一种方法将它们压缩存储到一个向量中，并找到矩阵中的每个非零元素与该向量的对应关系，通过这个关系，仍能对矩阵的元素进行随机存取。

4. 稀疏矩阵

假设 $m \times n$ 的矩阵中，有 t 个元素不为零。令 $\delta = \frac{t}{m \times n}$ ，称 δ 为矩阵的稀疏因子。

通常认为 $\delta \leq 0.05$ 时称为稀疏矩阵。

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

图 4-7 稀疏矩阵图

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

4-8 稀疏矩阵的三元组表示

按照压缩存储的概念，只存储稀疏矩阵的非零元。因此，除了存储非零元的值之外，还必须同时记下它所在的行和列的位置 (i, j) 。反之，一个三元组 (i, j, a_{ij}) 唯一确定了矩阵 A 的一个非零元。由此，稀疏矩阵可由表示非零元的三元组及其行列数唯一确定。如图 4-7 稀疏矩阵的三元组表示如图 4-8 所示。

【习题演练】

1. 设有一个 10 阶的对称矩阵 A ，采用压缩存储方式，以行序为主存储， a_{11} 为第一元素，其存储地址为 1，每个元素占一个地址空间，则 a_{85} 的地址为（ ）。

- A. 13 B. 32 C. 33 D. 40

2. 若对 n 阶对称矩阵 A 以行序为主序方式将其下三角形的元素(包括主对角线上所有元素)依次存放于一维数组 $B[1..(n(n+1))/2]$ 中，则在 B 中确定 a_{ij} ($i < j$) 的位置 k 的关系为（ ）。

- A. $i*(i-1)/2+j$ B. $j*(j-1)/2+i$ C. $i*(i+1)/2+j$ D. $j*(j+1)/2+i$

3. 将一个 $A[1...100][1...100]$ 的三对角矩阵，按行优先存入一维数组 $B[1...298]$ 中， A 中元素 $A[66][65]$ 在数组 B 中的位置 k 为（ ）。

- A. 198 B. 195 C. 197 D. 196

4. 适用于压缩存储稀疏矩阵的两种存储结构是（ ）。

- A. 三元组表和十字链表
- B. 三元组表和邻接矩阵
- C. 十字链表和二叉链表
- D. 邻接矩阵和十字链表

offcn

第五章 树和二叉树

【本章综述】

本章内容通常以选择题的形式考查，但不排除会有算法题涉及树的遍历。树和二叉树的性质遍历操作、转换、存储结构和操作特性等，满二叉树、完全二叉树、线索二叉树、哈夫曼树的定义和性质，树和森林与二叉树的转换关系等都是选择题必然会涉及的内容。

【复习重点】

（一）树的定义和基本术语

1. 树的基本术语及性质

（二）二叉树

1. 二叉树的定义

2. 二叉树的性质

3. 二叉树的存储结构

（三）遍历二叉树和线索二叉树

1. 遍历二叉树

2. 线索二叉树

（四）树和森林

1. 树的存储结构

2. 树、森林与二叉树的转换

3. 树和森林的遍历

（五）哈夫曼树及其应用

1. 哈夫曼树

2. 哈夫曼编码

第一节 树的定义和基本术语

一、树的基本术语及性质

1. 树

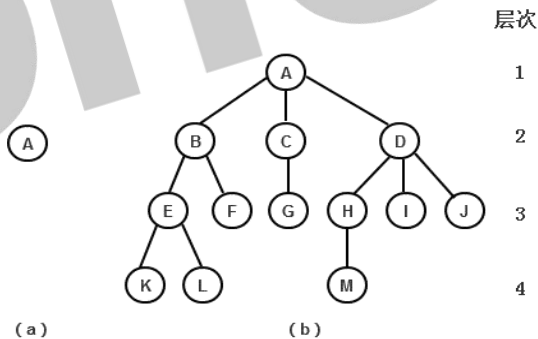
树 (Tree) 是 n ($n \geq 0$) 个结点的有限集。在任意一棵非空树中：(1) 有且仅有一个特定的称为**根** (Root) 的结点；(2) 当 $n > 1$ 时，其余结点可分为 m ($m > 0$) 个互不相交的有限集 T_1, T_2, \dots, T_m ，其中每一个集合本身又是一棵树，并且称为根的**子树** (SubTree)。

2. 结点、度

树的**结点**包含一个数据元素及若干指向其子树的分支。结点拥有的子树数称为结点的**度** (Degree)。度为 0 的结点称为**叶子** (Leaf) 或**终端结点**。度不为 0 的结点称为**非终端结点**或**分支结点**。除根结点之外，分支结点也称为**内部结点**。**树的度**是树内各结点的度的最大值。结点的子树的根称为该结点的**孩子** (Child)，相应地，该结点称为孩子的**双亲** (Parent)。同一个双亲的孩子之间互称**兄弟**。结点的**祖先**是从根到该结点所经分支上的所有结点。反之，以某结点为根的子树中的任一结点都称为该结点的**子孙**。

3. 结点的层次、树的高度

结点的**层次** (Level) 从根开始定义起，根为第一层，根的孩子为第二层。若某结点在第 Z 层，则其子树的根就在第 $Z+1$ 层。其双亲在同一层的结点互为**堂兄弟**。树中结点的最大层次称为树的**深度** (Depth) 或**高度**。



(a) 只有根结点的树； (b) 一般的树

图 5-1 树的示例

4. 有序树、无序树

如果将树中结点的各子树看成从左至右是有次序的 (即不能互换)，则称该树为**有序树**，否则称为**无序树**。在有序树中最左边的子树的根称为第一个孩子，最右边的称为最后一个孩子。

5. 路径和路径长度

树中两个结点之间的路径是由这两个结点之间所经过的结点序列构成的，而路径长

度是路径上所经过的边的个数。

6. 森林

森林 (Forest) 是 m ($m \geq 0$) 棵互不相交的树的集合。对树中每个结点而言, 其子树的集合即为森林。由此, 也可以森林和树相互递归的定义来描述树。如图 5-2 中的 3 棵树即组成一个森林。

7. 树的性质

- (1) 树的结点数等于所有结点的度数加 1。
- (2) 度为 m 的树中第 i 层上至多有 m^{i-1} 个结点 ($i \geq 1$)。
- (3) 高度为 h 的 m 叉树至多有 $(m^h - 1)/(m - 1)$ 个结点。
- (4) 具有 n 个结点的 m 叉树的最小高度为 $\lceil \log_m (n(m-1) + 1) \rceil$

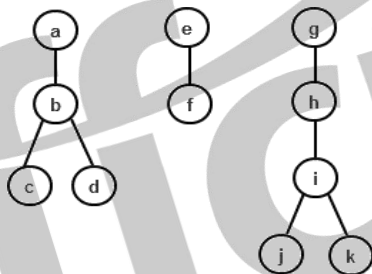


图 5-2 森林示例

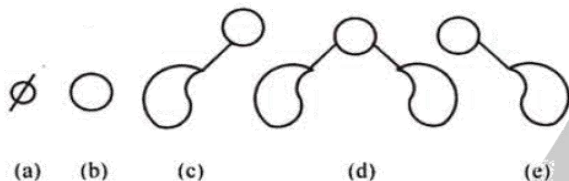
【习题演练】

1. 树最适合用来表示 () 的数据。
 - A. 有序
 - B. 无序
 - C. 任意元素之间具有多种联系
 - D. 元素之间具有分支层次关系
2. 一棵有 n 个结点的树的所有结点的度数之和为 ()。
 - A. $n-1$
 - B. n
 - C. $n+1$
 - D. $2n$
3. 度为 4、高度为 h 的树, 则 ()。
 - A. 至少有 $h+3$ 个结点
 - B. 至多有 $4h-1$ 个结点
 - C. 至多有 $4h$ 个结点
 - D. 至多有 $h+4$ 个结点

第二节 二叉树

一、二叉树的定义

二叉树 (Binary Tree) 是另一种树型结构, 它的特点是每个结点至多只有两棵子树 (二叉树中不存在度大于 2 的结点, 它的度可能是 0, 1, 2), 并且, 二叉树的子树有左右之分 (左孩子, 右孩子), 其次序不能任意颠倒。二叉树有五种基本形态。



(a) 空二叉树; (b) 仅有根结点的二叉树; (c) 右子树为空的二叉树;
(d) 左、右子树均非空的二叉树; (e) 左子树为空的二叉树。

图 5-3 二叉树的 5 种基本形态

二、二叉树的性质

性质 1 二叉树的第 i 层上至多有 2^{i-1} 个结点, ($i \geq 1$)。

性质 2 深度为 k 的二叉树至多有 $2^k - 1$ 个结点, ($k \geq 1$)。

性质 3 对任何一棵二叉树 T , 其叶子结点数为 n_0 , 度为 2 的结点数为 n_2 , 则 $n_0 = n_2 + 1$ 。

满二叉树: 一棵深度为 k , 且有 $2^k - 1$ 个结点的二叉树。可以给满二叉树的结点编号, 约定从根结点起, 从上至下, 从左至右, 如图 5-4(a)。

完全二叉树: 深度为 k 的, 有 n 个结点的二叉树, 当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时, 称之为完全二叉树, 如图 5-4(b)。这种树的特点是:

- (1) 叶子结点只可能在层次最大的两层上出现;
- (2) 对任一结点, 若其右分支下的子孙的最大层次为 l , 则其左分支下的子孙的最大层次必为 l 或 $l + 1$ 。

如图 5-4 中 (c) 和 (d) 不是完全二叉树。

性质 4 具有 n ($n > 0$) 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$ ¹。

性质 5 n 个结点的完全二叉树, 结点按层次编号, 则对任一结点有:

- (1) 如果 $i = 1$, 则结点 i 为二叉树的根, 无双亲; 如果 $i > 1$, 则其双亲是 $\lfloor i/2 \rfloor$ 结点。
- (2) 如果 $2i > n$, 则结点 i 无左孩子; 否则其左孩子是结点 $2i$ 。
- (3) 如果 $2i + 1 > n$, 则结点 i 无右孩子, 否则其右孩子是 $2i + 1$ 。

¹ 符号 $\lfloor \cdot \rfloor$ 表示不大于的最大整数, 反之, $\lceil \cdot \rceil$ 表示不小于的最小整数

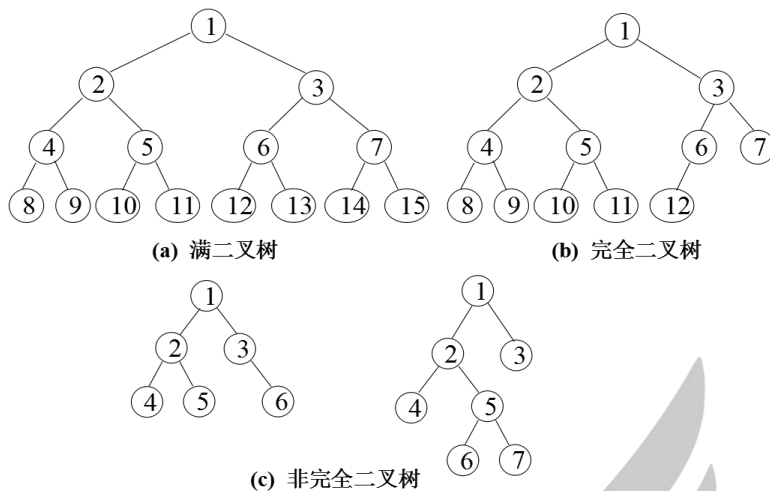


图 5-4 特殊形态的二叉树

【习题演练】

- 由 3 个结点可以构造出多少种不同的二叉树？（ ）
A. 2 B. 3 C. 4 D. 5
- 一棵完全二叉树上有 1001 个结点，其中叶子结点的个数是（ ）。
A. 250 B. 500 C. 254 D. 501
- 一个具有 1025 个结点的二叉树的高 h 为（ ）。
A. 11 B. 10 C. 11 至 1025 之间 D. 10 至 1024 之间
- 深度为 h 的满 m 叉树的第 k 层有（ ）个结点。（ $1 \leq k \leq h$ ）
A. m^{k-1} B. $m^k - 1$ C. m^{h-1} D. $m^h - 1$
- 二叉树有 n 个叶子，没有度为 1 的结点，共有____个结点。完全二叉树的第 3 层有 2 个叶子，则共有____个结点。
- 用二叉链表存储 n 个结点的二叉树（ $n > 0$ ），共有____个空指针域；采用三叉链表存储，共有____个空指针域。

三、二叉树的存储结构

1. 顺序存储结构

二叉树顺序存储结构是用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素，即将完全二叉树上编号为 i 的结点元素存储在某个数组下标为 $i-1$ 的分量中。

完全二叉树和满二叉树采用顺序存储比较合适，树中结点的序号可以唯一地反映出

结点之间的逻辑关系，这样既能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结点在二叉树中的位置，以及结点之间的关系。

但对于一般的二叉树，为了让数组下标能反映二叉树中结点之间的逻辑关系，只能添加一些并不存在的空结点让其每个结点与完全二叉树上的结点相对照，再存储到一维数组的相应分量中。然而，在最坏的情况下，一个高度为 H 且只有 H 个结点的单支树却需要占接近 2^H-1 个存储单元。二叉树的顺序存储结构如图 5-5 所示，其中 0 表示并不存在的空结点。

二叉树的顺序存储表示如下：

```
#define MAX_TREE_SIZE 100 // 二叉树的最大结点数
typedef TElemType SqBiTree[MAX_TREE_SIZE]; // 0 号单元存储根结点
SqBiTree bt;
```

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

(a)

1	2	3	4	5	0	0	0	0	6	7
---	---	---	---	---	---	---	---	---	---	---

(b)

(a) 完全二叉树；(b) 一般二叉树

图 5-5 二叉树的顺序存储结构

2. 链式存储结构

二叉树的结点 5-6 (a) 由一个数据元素和分别指向其左、右子树的两个分支构成，则表示二叉树的链表中的结点至少包含三个域：数据域和左、右指针域，如图 5-6 (b) 所示。为了便于找到结点的双亲，还可在结点结构中增加一个指向其双亲的指针域，如图 5-6 (c)。利用这两种结点结构所得二叉树的存储结构分别称之为**二叉链表**和**三叉链表**，如图 5-7 所示。链表的头指针指向二叉树的根结点。

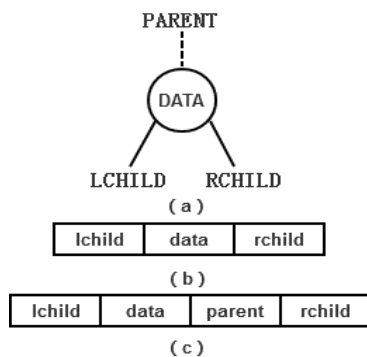
二叉链表：

```
typedef struct BTreeNode {
    DataType data;
    struct BTreeNode *lchild, *rchild;
} BTreeNode, *BinTree;
```

三叉链表：

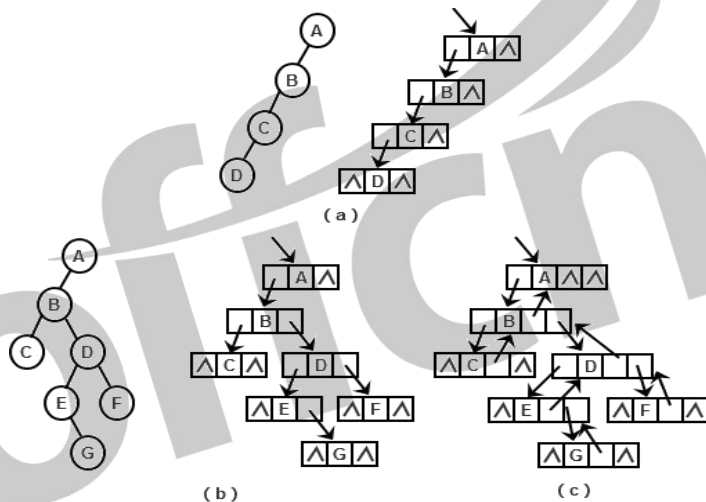
```
typedef struct BTreeNode {
    DataType data;
    struct BTreeNode *lchild, *rchild, *parent;
```

} BTreeNode, *BinTree;



(a) 二叉树的结点；(b) 含有两个指针域的结点结构；(c) 含有三个指针域的结点结构

图 5-6 二叉树的结点及其存储结构



(a) 单支树的二叉链表；(b) 二叉链表；(c) 三叉链表

图 5-7 链表存储结构

第三节 遍历二叉树和线索二叉树

一、遍历二叉树

遍历二叉树 (traversing binary tree) 是按某条搜索路径巡访树中每个结点，使得每个结点均被访问一次，而且仅被访问一次。

常见的四种遍历方式：先序遍历 (DLR)；中序遍历 (LDR)；后序遍历 (LRD)；层次遍历 (从上至下，从左至右)。

1. 先序遍历

先序遍历二叉树的操作定义为：

若二叉树为空，则空操作；否则

- (1) 访问根结点；
- (2) 先序遍历左子树；
- (3) 先序遍历右子树。

先序遍历的递归算法：

```
void Preorder ( BinTree bt )
{
    if ( bt ) {
        visit ( bt->data );      //访问当前结点
        Preorder ( bt->lchild ); //访问左孩子树
        Preorder ( bt->rchild ); //访问右孩子树
    }
}
```

先序遍历的非递归算法：

```
void Preorder ( BinTree bt, VisitFunc visit )
{
    InitStack ( S );
    p = bt;
    while ( p || ! StackEmpty(S) ) {
        if ( p ) {
            visit ( p ); // 先序访问结点的位置
            Push ( S, p );
            p = p->lchild;
        } else {
            Pop ( S, p );
            p = p->rchild;
        }
    }
}
```

或者，由于访问过的结点便可以弃之不用，只要能访问其左右子树即可，写出如下算法。

```
void Preorder ( BinTree bt, VisitFunc visit )
{
    InitStack ( S );
    Push ( S, bt );
```

```

while ( ! StackEmpty(S) ) {
    Pop ( S, p );
    if ( p ) {
        visit ( p );
        Push ( S, p->rchild );    // 先进栈，后访问，所以
        Push ( S, p->lchild );    // 这里先让右子树进栈
    }
}
}

```

2. 中序遍历

中序遍历二叉树的操作定义为：若二叉树为空，则空操作；否则

- (1) 中序遍历左子树；
- (2) 访问根结点；
- (3) 中序遍历右子树。

中序遍历的递归算法：

```

void Inorder ( BinTree bt )
{
    if ( bt ) {
        Inorder ( bt->lchild );    //访问左孩子树
        visit ( bt->data );        //访问当前结点
        Inorder ( bt->rchild );    //访问右孩子树
    }
}

```

中序遍历的非递归算法：

指针 p 从根开始，首先沿着左子树向下移动，同时入栈保存；当到达空子树后需要退栈访问结点，然后移动到右子树上去。

```

void InOrder ( BinTree bt, VisitFunc visit )
{
    InitStack ( S );
    p = bt;
    while ( p || ! StackEmpty(S) ) {
        if ( p ) {
            Push ( S, p );
            p = p->lchild;
        } else {
            Pop ( S, p );
            visit ( p );
            p = p->rchild;
        }
    }
}

```

```

        visit ( p ); // 中序访问结点的位置
        p = p->rchild;
    }
}
}

```

3. 后序遍历

后序遍历二叉树的操作定义为：

若二叉树为空，则空操作；否则

- (1) 后序遍历左子树；
- (2) 后序遍历右子树；
- (3) 访问根结点。

后序遍历的递归算法：

```

void Postorder ( BinTree bt )
{
    if ( bt ) {
        Postorder ( bt->lchild ); //访问左孩子树
        Postorder ( bt->rchild ); //访问右孩子树
        visit ( bt->data ); //访问当前结点
    }
}

```

后序遍历的非递归算法：

后序遍历时，分别从左子树和右子树共两次返回根结点，只有从右子树返回时才访问根结点，所以增加一个栈标记到达结点的次序。

```

void PostOrder ( BinTree bt, VisitFunc visit )
{
    InitStack ( S ), InitStack ( tag );
    p = bt;
    while ( p || ! StackEmpty(S) ) {
        if ( p ) {
            Push ( S, p ), Push ( tag, 1); // 第一次入栈
            p = p->lchild;
        } else {
            Pop ( S, p ), Pop ( tag, f);
            if ( f==1 ) {
                // 从左子树返回，二次入栈，然后 p 转右子树
                Push ( S, p ), Push ( tag, 2 );
            }
        }
    }
    visit ( p );
}

```

```

        p = p->rchild;
    } else {
        // 从右子树返回(二次出栈), 访问根结点, p 转上层
        visit ( p );
        p = NULL;    // 必须的, 使下一步继续退栈
    }
}
}
}
}

```

注：后序非递归遍历的过程中，栈中保留的是当前结点的所有祖先。这是和先序及中序遍历不同的。在某些和祖先有关的算法中，此算法很有价值。

4. 层次遍历

从根结点开始，从上到下、从左到右按层次遍历。

利用一个队列，首先将根（头指针）入队列，以后若队列不空则取队头元素 p ，如果 p 不空，则访问之，然后将其左右子树入队列，如此循环直到队列为空。

(1) 按队列实现的层次遍历

```

void LevelOrder ( BinTree bt )
{
    // 队列初始化为空
    InitQueue ( Q );
    // 根入队列
    EnQueue ( Q, bt );
    // 队列不空则继续遍历
    while ( ! QueueEmpty(Q) ) {
        DeQueue ( Q, p );
        if ( p!=NULL ) {
            visit ( p->data );
            // 左、右子树入队列
            If(p->lchild)
                EnQueue ( Q, p->lchild );
            If(p->rchild)
                EnQueue ( Q, p->rchild );
        }
    }
}

```

(2) 简易版队列的层次遍历

若队列表示为“数组 $q[]$ + 头尾 $front, rear$ ”有：

```
void LevelOrder ( BinTree bt )
{
    const int MAXSIZE = 1024;
    BinTree q[MAXSIZE];
    int front, rear;
    // 队列初始化为空
    front = rear = 0;
    // 根入队列
    q[rear] = bt; rear = ( rear+1 ) % MAXSIZE;
    // 队列不空则循环
    while ( front != rear ) {
        p = q[front]; front = ( front+1 ) % MAXSIZE;
        if ( p ) {
            visit ( p->data );
            // 左、右子树入队列
            q[rear] = p->lchild; rear = ( rear+1 ) % MAXSIZE;
            q[rear] = p->rchild; rear = ( rear+1 ) % MAXSIZE;
        }
    }
}
```

显然，遍历二叉树的算法中的基本操作是访问结点，则不论按哪一种次序进行遍历，对含 n 个结点的二叉树，其时间复杂度均为 $O(n)$ 。

5. 遍历二叉树的应用

已知遍历序列（前、中、后序），根据遍历序列画出二叉树。

（1）已知前序和中序序列，可以唯一确定二叉树。

例：前序：ABDEGCFH，中序：DBGAEFHC，画出二叉树。

分析：前序序列的第一个是根结点，这是解题的突破口。

步骤：①前序序列的第一个是根 ②在中序序列中标出根，分成左右子树 ③在前序序列中标出左右子树（根据结点个数即可）④分别对左右子树的前序和中序序列重复以上步骤直至完成。

（2）已知后序和中序序列，可以唯一确定二叉树。

例：后序：DGE BH FCA，中序：DBGAEFHC，画出二叉树。

分析：后序序列的最后一个为根，这是解题的突破口。

步骤：①后序序列的最后一个为根 ②在中序序列中标出根，分成左右子树 ③在后

序序列中标出左右子树（根据结点个数即可）④分别对左右子树的后序和中序序列重复以上步骤直至完成。

（3）已知前序和后序序列，不能唯一确定二叉树。

【习题演练】

1. 对二叉树的结点从 1 开始进行连续编号，要求每个结点的编号大于其左、右孩子的编号，同一结点的左右孩子中，其左孩子的编号小于其右孩子的编号，可采用（ ）遍历实现编号。

- A. 先序 B. 中序 C. 后序 D. 从根开始按层次遍历

2. 若二叉树采用二叉链表存储结构，要交换其所有分支结点左右子树的位置，利用（ ）遍历方法最合适。

- A. 前序 B. 中序 C. 后序 D. 按层次

3. 一棵非空的二叉树的先序遍历序列与后序遍历序列正好相反，则该二叉树一定满足（ ）。

- A. 所有的结点均无左孩子 B. 所有的结点均无右孩子
C. 只有一个叶子结点 D. 是任意一棵二叉树

二、线索二叉树

1. 线索二叉树

n 个结点的二叉链表中有 $n+1$ 个空指针，可以利用这些空指针指向某种遍历中结点的前驱或后继结点，同时需附加标志域，用以区分是子树还是线索。

规定：若结点有左子树，则其 lchild 域指示其左孩子，否则令 lchild 域指示其前驱；若结点有右子树，则其 rchild 域指示其右孩子，否则令 rchild 域指示其后继。为了避免混淆，增加两个标志域。

lchild	LTag	data	RTag	rchild
--------	------	------	------	--------

图 5-8 线索二叉树的结点

其中：

$$LTag = \begin{cases} 0 & \text{lchild 域指示结点的左孩子} \\ 1 & \text{lchild 域指示结点的前驱} \end{cases}$$

$$RTag = \begin{cases} 0 & \text{rchild 域指示结点的右孩子} \\ 1 & \text{rchild 域指示结点的后继} \end{cases}$$

以这种结点结构构成的二叉链表作为二叉树的存储结构，叫做**线索链表**，其中指向结点前驱和后继的指针，叫做**线索**。

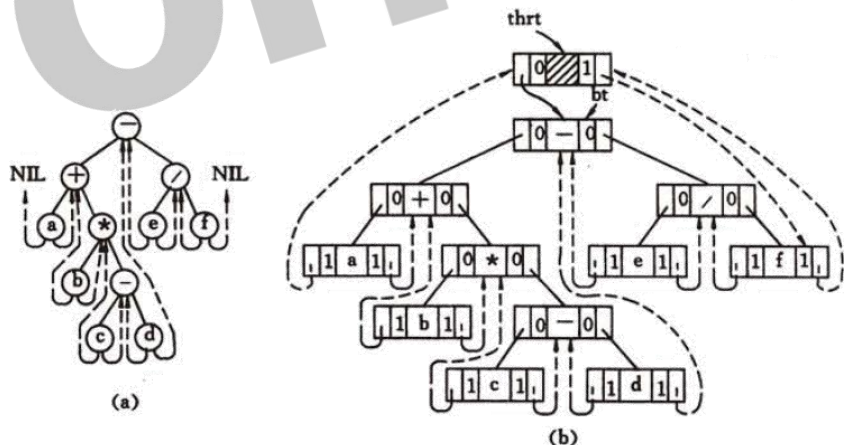
加上线索的二叉树称之为**线索二叉树**（Threaded Binary Tree）。如图 5-9（a）所示为中序线索二叉树，与其对应的中序线索链表如图 5-9（b）所示。其中实线为指针（指向左、右子树），虚线为线索（指向前驱和后继）。对二叉树以某种次序遍历使其变为线索二叉树的过程叫做**线索化**。

二叉树的二叉线索存储表示：

```
typedef enum PointerTag {Link, Thread}; //Link==0: 指针, Thread==1: 线索
typedef struct BiThrNode {
    TElemType data;
    struct BiThrNode * lchild, * rchild; //左右孩子指针
    PointerTag LTag, RTag;
} BiThrNode, *BiThrTree;
```

表 5-1 线索化二叉树的类型

	前驱、后继线索	前驱线索	后继线索
中序线索化	中序全线索	中序前驱线索	中序后继线索
前序线索化	前序全线索	前序前驱线索	前序后继线索
后序线索化	后序全线索	后序前驱线索	后序后继线索



(a)中序线索二叉树； (b)中序线索链表

图 5-9 线索二叉树及其结构

为方便起见仿照线性表的存储结构，在二叉树的线索链表上也添加一个头结点，并令其 lchild 域的指针指向二叉树的根结点，其 rchild 域的指针指向中序遍历时访问的最后一个结点；反之，令二叉树中序序列中的第一个结点的 lchild 域指针和最后一个结点

rchild 域的指针均指向头结点。这好比为二叉树建立了一个双向线索链表，既可从第一个结点起顺后继进行遍历，也可从最后一个结点起顺前驱进行遍历（如图 5-9（b）所示）。

2. 线索二叉树的遍历

在**线索树上进行遍历**，只要先找到序列中的第一个结点，然后依次找结点后继直至其后继为空时为止。

如何在线索树中找结点的后继？以图 5-9 的中序线索树为例来看，树中所有叶子结点的右链是线索，则右链域直接指示了结点的后继，如结点 b 的后继为结点*。树中所有非终端结点的右链均为指针，则无法由此得到后继的信息。然而，根据中序遍历的规律可知，结点的后继应是遍历其右子树时访问的第一个结点，即右子树中最左下的结点。例如在找结点*的后继时，首先沿右指针找到其右子树的根结点“-”，然后顺其左指针往下直至其左标志为 1 的结点，即为结点*的后继，在图中是结点 c。反之，在中序线索树中找结点前驱的规律是：若其左标志为“1”，则左链为线索，指示其前驱，否则遍历左子树时最后访问的一个结点（左子树中最右下的结点）为其前驱。

在后序线索树中找结点后继较复杂些，可分 3 种情况：（1）若结点 x 是二叉树的根，则其后继为空；（2）若结点 x 是其双亲的右孩子或是其双亲的左孩子且其双亲没有右子树，则其后继即为双亲结点；（3）若结点 x 是其双亲的左孩子，且其双亲有右子树，则其后继为双亲的右子树上按后序序列出的第一个结点。例如图 5-11 所示为后序后继线索二叉树，结点 B 的后继为结点 C，结点 C 的后继为结点 D，结点 F 的后继为结点 G，而结点 D 的后继为结点 E。可见，在后序线索化树上找后继时需知道结点双亲，即需带标志域的三叉链表作存储结构。

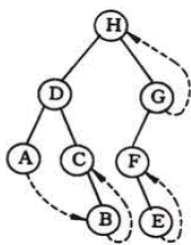


图 5-10 后序线索二叉树

3. 画线索二叉树

思路：先写出遍历序列，再画线索。

步骤：标出必要的空指针（前驱→左指针；后继→右指针，要点：“不要多标，也不要少标”）。

写出对应的遍历序列（前序，中序或后序）。对照遍历结果画线索。

例：画出图 5-12 中二叉树的后序后继线索。

步骤：先标出所有空的右指针（DGCH）；写出后序遍历结果：DGEHBHFC；标出后继线索：D→G, G→E, C→A, H→F。反复利用孩子和线索进行遍历，可以避免递归。

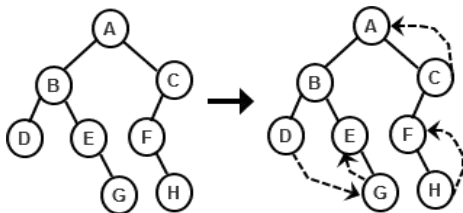


图 5-11 后序后继线索化

【习题演练】

- 若 X 是二叉中序线索树中一个有左孩子的结点，且 X 不为根，则 X 的前驱为（ ）。
 - X 的双亲
 - X 的右子树中最左的结点
 - X 的左子树中最右结点
 - X 的左子树中最右叶结点
- 引入二叉线索树的目的是（ ）。
 - 加快查找结点的前驱或后继的速度
 - 为了能在二叉树中方便的进行插入与删除
 - 为了能方便的找到双亲
 - 使二叉树的遍历结果唯一
- 设一棵二叉树的先序序列：A B D F C E G H，中序序列：B F D A G E H C
 - 画出这棵二叉树。
 - 画出这棵二叉树的后序线索树。
 - 将这棵二叉树转换成对应的树（或森林）。

第四节 树和森林

这一节我们将讨论树的表示及其遍历操作，并建立森林与二叉树的对应关系。

一、树的存储结构

树的常用存储结构：双亲表示法，孩子表示法，孩子兄弟表示法。

特点：双亲表示法容易求得双亲，但不容易求得孩子；孩子表示法容易求得孩子，但求双亲麻烦；两者可以结合起来使用。孩子兄弟表示法，容易求得孩子和兄弟，求双

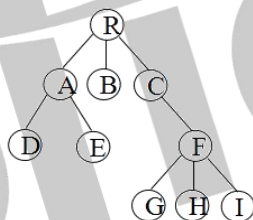
亲麻烦，也可以增加指向双亲的指针来解决。

1. 双亲表示法

假设以一组连续空间存储树的结点，同时在每个结点中附设一个指示器指示其双亲结点在链表中的位置，其形式说明如下：

树的双亲表存储表示：

```
# define MAX_TREE_SIZE      100
typedef struct PTNode{ // 结点结构
    TElemType data;
    int    parent // 双亲位置域
}PTNode;
typedef struct { // 树结构
    PTNode    nodes[MAX_TREE_SIZE];
    int  r, n; // 根的位置和结点数
}PTree;
```



0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	I	6

图 5-12 树的双亲表示法示例

2. 孩子表示法

把树中每个结点的孩子结点排列起来，看成是一个线性表，且以单链表作存储结构，则 n 个结点有 n 个孩子链表（叶子的孩子链表为空表）。而 n 个头指针又组成一个线性表，为了便于查找，可采用顺序存储结构。这种存储结构可形式地说明如下：

树的孩子链表存储表示：

```
typedef struct CTNode{ // 孩子结点
    int    child;
    struct CTNode  *next;
}* ChildPtr;
typedef struct{
    TElemType  data;
    ChildPtr  firstchild; // 孩子链表头指针
}CTBox;
```

```
typedef struct{
    CTBox  nodes[MAX_TREE_SIZE];
    int  n, r;    // 结点数和根的位置;
}CTree;
```

树的孩子表示法如图 5-13 所示，与双亲表示法相反，孩子表示法便于那些涉及孩子的操作的实现，却不适用于 PARENT(T, x)操作。

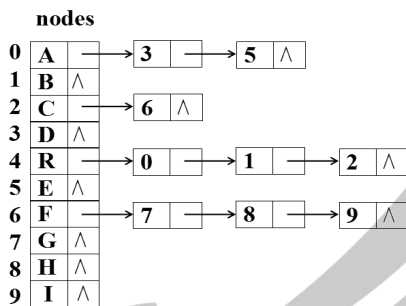


图 5-13 孩子表示法图例

3. 孩子兄弟表示法

又称二叉树表示法，或二叉链表表示法。即以二叉链表作树的存储结构。链表中结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点，分别命名为 firstchild 域和 nextsibling 域。

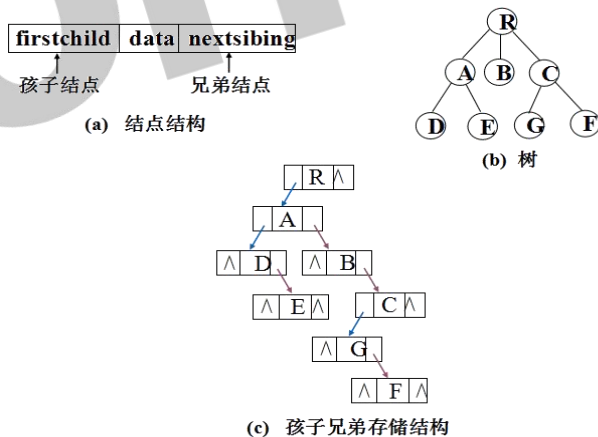


图 5-14 树的孩子兄弟表示法

树的二叉链表（孩子—兄弟）存储表示：

```
typedef struct CSNode{
    ElemType  data;
    struct CSNode  *firstchild,  *nextsibling;
}CSNode, *CSTree;
```

【习题演练】

1. 利用二叉链表存储树，则根结点的右指针是（ ）。
- A. 指向最左孩子 B. 指向最右孩子 C. 空 D. 非空
2. 在下列存储形式中，（ ）不是树的存储形式？
- A. 双亲表示法 B. 孩子链表表示法 C. 孩子兄弟表示法 D. 顺序存储表示法

二、树、森林与二叉树的转换

由于二叉树和树都可用二叉链表作为存储结构，则以二叉链表作为媒介可导出树与二叉树表之间的一个对应关系。也就是说，给定一棵树，可以找到惟一的一棵二叉树与之对应，从物理结构来看，它们的二叉链表是相同的，只是解释不同而已。图 5-15 展示了树与二叉树之间的对应关系。从树的二叉链表表示的定义可知，任何一棵和树对应的二叉树，其右子树必空。若把森林中第二棵树的根结点看成是第一棵树的根结点的兄弟，则同样可导出森林和二叉树的对应关系。例如，图 5-16 展示了森林和二叉树的对应关系。

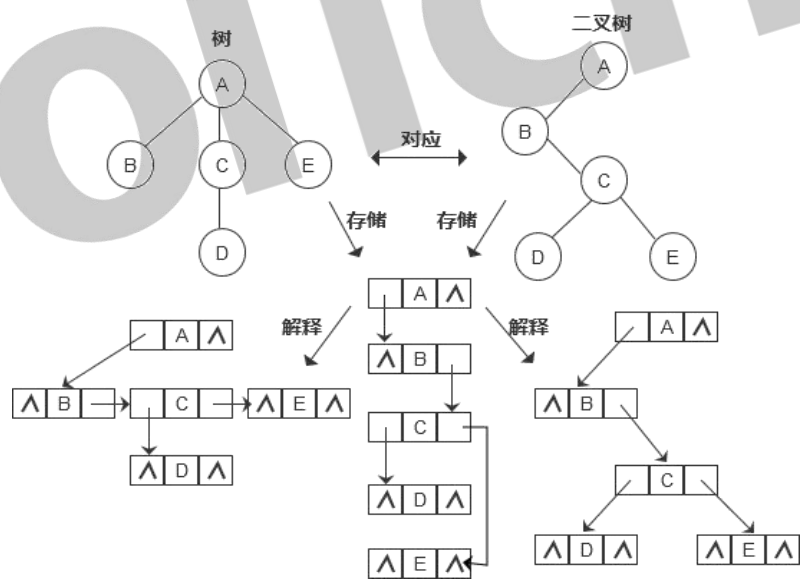


图 5-15 树与二叉树的对应关系图

表 5-2 树和二叉树的对应关系

树	对应的二叉树
根	根
第一个孩子	左孩子
下一个兄弟	右孩子

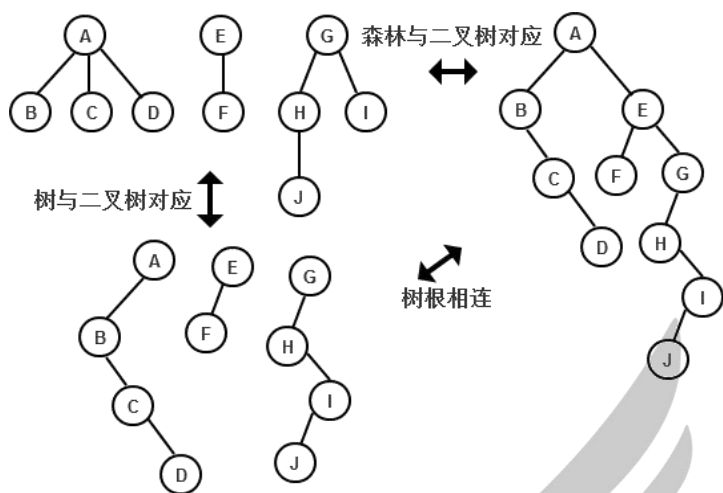


图 5-16 森林与二叉树的对应关系

【习题演练】

- 把一棵树转换为二叉树后，这棵二叉树的形态是（ ）。
 A. 唯一的
 B. 有多种
 C. 有多种，但根结点都没有左孩子
 D. 有多种，但根结点都没有右孩子
- 设 F 是一个森林， B 是由 F 变换得的二叉树。若 F 中有 n 个非终端结点，则 B 中右指针域为空的结点有（ ）个。
 A. $n-1$
 B. n
 C. $n+1$
 D. $n+2$

三、树和森林的遍历

1. 树的遍历

先根遍历，即：先访问根结点，然后依次先根遍历根的没棵子树。例如图 5-17 (a) 的先根遍历为 ABCDEFGHIJK。

后根遍历，即：先依次后根遍历每棵子树，然后访问根结点。例如图 5-17 (a) 的先根遍历为 CEDFBHGJKIA。

2. 森林的遍历

按照森林和树相互递归的定义，我们可以推出森林的两种遍历方法：

a. 先序遍历森林

- 若森林非空，则可按下述规则遍历之：
- (1) 访问森林中第一棵树的根结点；
 - (2) 先序遍历第一棵树中根结点的子树森林；
 - (3) 先序遍历除去第一棵树之后剩余的树构成的森林。

b. 中序遍历森林

- 若森林非空，则可按下述规则遍历之：
- (1) 中序遍历森林中第一棵树的根结点的子树森林；
 - (2) 访问第一棵树的根结点；
 - (3) 中序遍历除去第一棵树之后剩余的树构成的森林。

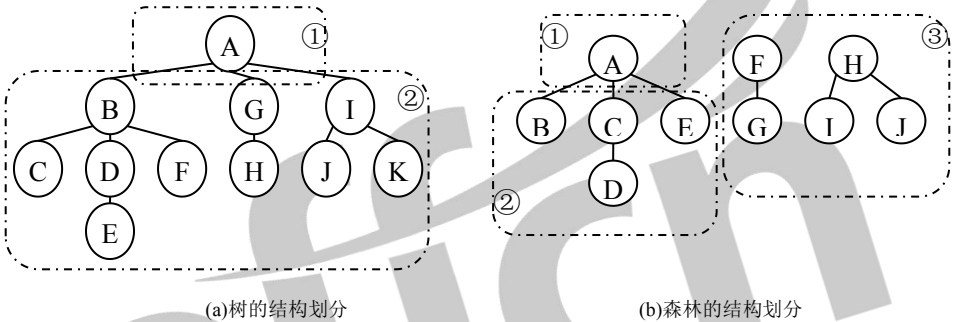


图 5-17

注：先序遍历森林，相当于依次先根遍历每一棵树；中序遍历森林相当于后根遍历每一棵树。

3. 遍历树、森林与遍历二叉树的关系

表 5-3 遍历树、森林和二叉树的关系

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

第五节 哈夫曼树及其应用

哈夫曼（Huffman）树，又称最优二叉树，是一类带权路径长度最短的树。

一、哈夫曼树

1. 哈夫曼树的基本概念

从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分

支数目称做**路径长度**。结点的最短路径长度为从该结点到树根之间的路径长度与结点上权的乘积。**树的路径长度**是从树根到每一结点的路径长度之和。**树的带权路径长度**：所有叶子结点的带权路径长度之和，通常记作

$$WPL = \sum_{k=1}^n w_k l_k$$

假设有 n 个权值 $\{w_1, w_2, \dots, w_n\}$ ，试构造一棵有 n 个叶子结点的二叉树，每个叶子结点带权为 w_i ，则其中带权路径长度 WPL 最小的二叉树称做**最优二叉树**或**哈夫曼树**。

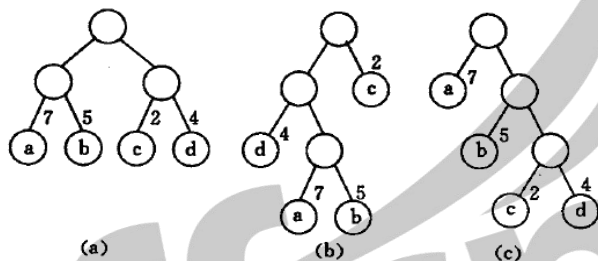


图 5-18 具有不同带权路径长度的二叉树

2. 哈夫曼树的构造

算法叙述如下：

(1) 根据给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 构成 n 棵二叉树的集合 $F = \{T_1, T_2, \dots, T_n\}$ ，其中每棵二叉树 T_i 中只有一个带权为 w_i 的根结点，其左右子树均空。

(2) 在 F 中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树根结点的权值为其左、右子树上根结点的权值之和。

(3) 在 F 中删除这两棵树，同时将新得到的二叉树加入 F 中。

(4) 重复 (2) 和 (3)，直到 F 只一棵树为止。这棵树便是哈夫曼树。

哈夫曼树的特征：

(1) 哈夫曼树中没有度为 1 的结点。

(2) 一棵有 n 个叶子结点的哈夫曼树共有 $2n-1$ 个结点（共新建了 $n-1$ 个结点）。

(3) 每个初始结点最终成为叶结点，并且权值越小的结点到根结点的路径长度越大。

图 5-19 展示了哈夫曼树的构造过程。

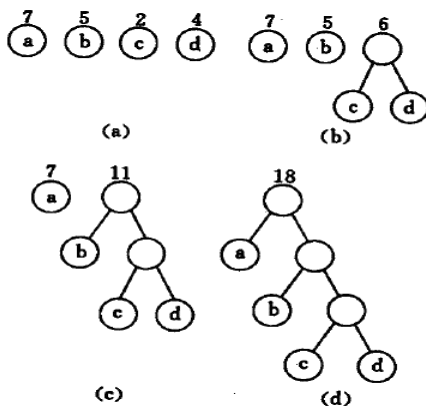


图 5-19 赫夫曼树的构造过程

【习题演练】

1. n ($n \geq 2$) 个权值均不相同的字符构成哈夫曼树，关于该树的叙述中，错误的是 ()。
 - A. 该树一定是一棵完全二叉树
 - B. 树中一定没有度为 1 的结点
 - C. 树中两个权值最小的结点一定是兄弟结点
 - D. 树中任一非叶结点的权值一定不小于下一层任一结点的权值
2. 设哈夫曼树中有 199 个结点，则该哈夫曼树中有 () 个叶子结点。
 - A. 99
 - B. 100
 - C. 101
 - D. 102

二、哈夫曼编码

对于待处理的一个字符串序列，如果对每个字符用同样长度的二进制位来表示，则称这种编码方式为**固定长度编码**。若允许对不同字符用不等长的二进制位表示，则这种方式称为**可变长度编码**，可变长度编码比固定长度编码好得多，其特点是对频率高的字符赋以短编码，而对频率较低的字符则赋以较长一些的编码，从而可以使字符平均编码长度减短，起到压缩数据的效果。**哈夫曼编码**是一种被广泛应用而且非常有效的**数据压缩编码**。

如果没有一个编码是另一个编码的前缀，则称这样的编码为**前缀编码**。如 0、101 和 100 是前缀编码。对前缀编码的解码也是很简单的，因为没有一个是其他码的前缀。

所以，可以识别出第一个编码，将它翻译为原码，再对余下的编码文件重复同样的解码操作。如 00101100 可被唯一地分析为 0、0、101 和 100。

由哈夫曼树得到哈夫曼编码是很自然的过程，首先，将每个出现的字符当做一个独立的结点，其权值为它出现的频度（或次数），构造出对应的哈夫曼树。显然，所有字符结点都出现在叶结点中。我们可以将字符的编码解释为从根至该字符的路径上边标记的序列，其中边标记为 0 表示“转向左孩子”，标记为 1 表示“转向右孩子”。

例：字符及其权值如下：A(6), B(7), C(1), D(5), E(2), F(8)，构造哈夫曼树和哈夫曼编码，计算带权路径长度。

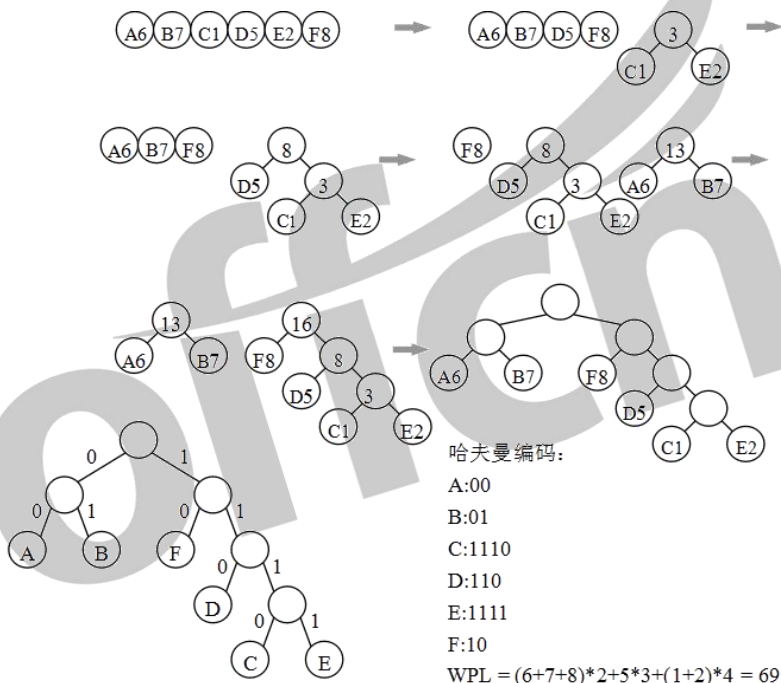


图 5-20 赫夫曼编码的求解过程

【习题演练】

1. 假设用于通信的电文仅由 8 个字母组成，字母在电文中出现的频率分别为 0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10。
 - (1) 试为这 8 个字母设计赫夫曼编码。
 - (2) 试设计另一种由二进制表示的等长编码方案。
 - (3) 对于上述实例，比较两种方案的优缺点。

第六章 图

【本章综述】

图算法的难度较大，因此主要掌握深度优先搜索与广度优先搜索的程序设计，其他内容以算法设计题形式出现的概率不大，通常会以选择题和综合题的形式考查。应掌握图的基本概念及基本性质（度、路径长度、回路、路径等）、图的存储结构（邻接矩阵、邻接表、邻接多重表和十字链表）及其特性、存储结构之间的转化、基于存储结构上的遍历操作和各种应用（拓扑排序、最小生成树、最短路径和关键路径）等。图的相关算法较多易混，但通常只要求掌握其基本思想和实现的步骤（能动手模拟），而算法的具体实现则不是重点。读者应将这些算法按不同的方式分类，对比记忆，并用一定量的实例练习。

【复习重点】

（一）图的定义和术语

1. 图的基本概念

（二）图的存储结构

1. 邻接矩阵
2. 邻接表
3. 逆邻接表

（三）图的遍历

1. 深度优先遍历、广度优先遍历
2. 图的连通性问题

（四）最小生成树

1. 最小生成树
2. 普里姆算法、克鲁斯卡尔算法

（五）有向无环图及其应用

1. 拓扑排序
2. 关键路径

（六）最短路径

1. 迪杰斯特拉算法

2. 弗洛伊德算法

第一节 图的定义和术语

一、图的基本概念

1. 图、有向图、无向图

图(Graph)中的数据元素通常称做**顶点(Vertex)**, V 是顶点的有穷非空集合; VR 是两个顶点之间的关系的集合。若 $\langle v, \omega \rangle \in VR$, 则 $\langle v, \omega \rangle$ 表示从 v 到 ω 的一条**弧(Arc)**, 且称 v 为**弧尾(Tail)**或初始点(Initial node), 称 ω 为**弧头(Head)**或终端点(Terminal node), 此时的图称为**有向图(Digraph)**。若 $\langle v, \omega \rangle \in VR$ 必有 $\langle \omega, v \rangle \in VR$, 即 VR 是对称的, 则以无序对 (v, ω) 代替这两个有序对, 表示 v 和 ω 之间的一条**边(Edge)**, 此时的图称为**无向图(Undigraph)**。

2. 完全图

用 n 表示图中顶点数目, 用 e 表示边或弧的数目。不考虑顶点到其自身的弧或边, 即若 $(v_i, v_j) \in VR$, 则 $v_i \neq v_j$, 那么, 对于无向图, e 的取值范围是 0 到 $n*(n-1)/2$ 。有 $n*(n-1)/2$ 条边的无向图称为**无向完全图(Completed graph)**。对于有向图, e 的取值范围是 0 到 $n(n-1)$ 。具有 $n(n-1)$ 条弧的有向图称为**有向完全图**。

3. 稀疏图、稠密图

有很少条边或弧(如 $e < n \log_2 n$) 的图称为**稀疏图(Sparse graph)**, 反之称为**稠密图(Dense graph)**。

4. 子图、权、网

若图 $G=(V, \{E\})$ 和 $G'=(V', \{E'\})$, 如果 $V' \subseteq V$ 且 $E' \subseteq E$, 则称 G' 为 G 的**子图(Subgraph)**。与图的边或弧相关的数叫做**权(Weight)**。带权的图通常称为**网(Network)**。

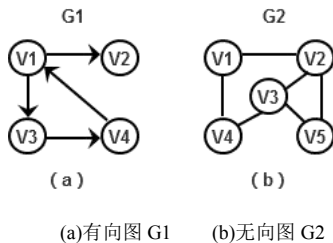


图 6-1 图的举例

5. 顶点的度、入度和出度

顶点 v 的**度** (Degree) 是和 v 相关联的边的数目, 记为 $TD(v)$ 。

对于无向图 $G=(V, \{E\})$, 如果边 $(v, v') \in E$, 则称顶点 v 和 v' 互为**邻接点** (Adjacent)。边 (v, v') **依附** (Incident) 于顶点 v 和 v' , 或者说 (v, v') 和顶点 v 和 v' **相关联**, 顶点 v 的

度是指依附于该顶点的边的条数, 记为 $TD(v)$ 。有 $\sum_{i=1}^n TD(v_i) = 2e$ 。

对于有向图 $G=(V, \{A\})$, 如果弧 $\langle v, v' \rangle \in A$, 则称顶点 v 邻接到顶点 v' , 顶点 v' 邻接自顶点 v 。以顶点 v 为头的弧的数目称为 v 的**入度** (InDegree), 记为 $ID(v)$; 以 v 为尾的弧的数目称为 v 的**出度** (Outdegree), 记为 $OD(v)$; 顶点 v 的**度**为 $TD(v)=ID(v)+OD(v)$ 。

有 $\sum_{i=1}^n ID(v_i) = \sum_{i=1}^n OD(v_i) = e$ 。

6. 路径、路径长度、回路、简单路径、简单回路

图 $G=(V, \{E\})$ 中从顶点 v 到顶点 v' 的**路径**(Path)是一个顶点序列。**路径长度**是路径上的边或弧的数目。第一个顶点和最后一个顶点相同的路径称为**回路**或**环** (Cycle)。序列中顶点不重复出现的路径称为**简单路径**。除了第一个顶点和最后一个顶点之外, 其余顶点不重复出现的回路, 称为**简单回路**或**简单环**。

7. 连通、连通图、连通分量

在无向图 G 中, 如果从顶点 v 到顶点 v' 有路径, 则称 v 和 v' 是**连通的**。图中任意两个顶点 $v_i, v_j \in V$, v_i 和 v_j 都是连通的, 则称 G 是**连通图** (Connected Graph)。所谓**连通分量** (Connected Component), 指的是无向图中的极大连通子图。

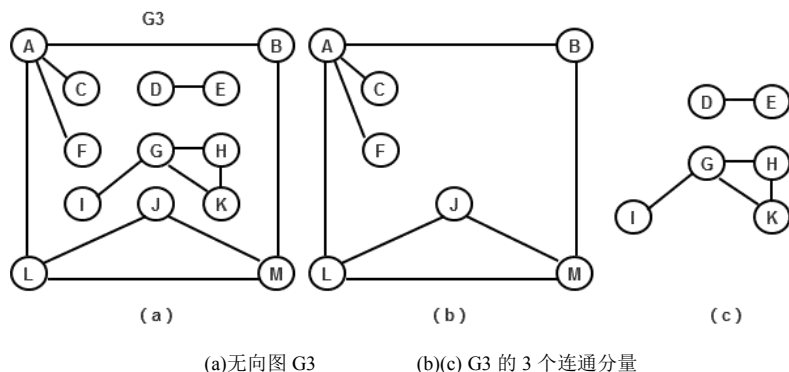


图 6-2 无向图及其连通分量

8. 强连通图、强连通分量

在有向图 G 中, 如果对于每一对 $v_i, v_j \in V$, $v_i \neq v_j$, 从 v_i 到 v_j 和从 v_j 到 v_i 都存在路径, 则称 G 是**强连通图**。有向图中的极大强连通子图称做有向图的**强连通分量**。一个连

通图的**生成树**是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的 $n-1$ 条边。

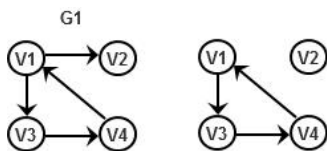


图 6-3 G_1 的两个强联通分量

9. 有向树

如果一个有向图恰有一个顶点的入度为 0，其余顶点的入度均为 1，则是一棵**有向树**。一个有向图的**生成森林**由若干棵有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的弧。

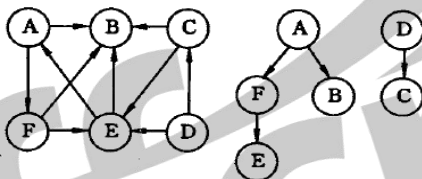


图 6-4 一个有向图及其生成森林

【习题演练】

1. 在一个图中，所有顶点的度数之和等于图的边数的（ ）倍。
A. $1/2$ B. 1 C. 2 D. 4
2. 在一个有向图中，所有顶点的入度之和等于所有顶点的出度之和的（ ）倍。
A. $1/2$ B. 1 C. 2 D. 4
3. 具有 n 个顶点的有向图最多有（ ）条边。
A. n B. $n(n-1)$ C. $n(n+1)$ D. n^2
4. G 是一个非连通无向图，共有 28 条边，则该图至少有（ ）个顶点。
A. 7 B. 8 C. 9 D. 10

第二节 图的存储结构

常见图的存储结构有：邻接矩阵，邻接表，逆邻接表，十字链表，邻接多重表。邻接多重表只适用于存储无向图，十字链表只适用于存储有向图。

一、邻接矩阵

用两个数组分别存储数据元素（顶点）的信息和数据元素之间的关系（边或弧）的信息。其形式描述如下：

```
const int MAX_VERTEX = 最大顶点个数;
typedef struct Graph { // 图
    VertexType    vexs[MAX_VERTEX]; // 顶点向量
    ArcType    arcs[MAX_VERTEX][MAX_VERTEX]; // 邻接矩阵
    int vexnum, arcnum; // 顶点和弧的个数
} Graph;
```

结点数为 n 的图 $G=(V,E)$ 的邻接矩阵 A 是 $n \times n$ 的，与边数无关，因此适用于稠密图。将 G 的顶点编号为 v_1, v_2, \dots, v_n 。对于一般的图，若 $(v_i, v_j) \in E$ ，则 $A[i][j]=1$ ，否则 $A[i][j]=0$ ，即顶点之间有边（弧）时矩阵元素为 1；否则为 0。对于网， $A[i][j]=\text{权值}$ ， $A[i][j]=\infty$ ，即有边（弧）时矩阵元素为权值；否则为 ∞ 。如图 7-5 为无向图和有向图的邻接矩阵。

无向图的邻接矩阵是对称的，可以采用压缩存储的方式只存入矩阵的下三角（或上三角）元素。

借助于邻接矩阵容易判定任意两个顶点之间是否有边（或弧）相连，并容易求得各个顶点的度。对于无向图，顶点 v_i 的度是邻接矩阵中第 i 行（或第 i 列）的元素之和，对于有向图，第 i 行的元素之和为顶点 v_i 的出度，第 i 列的元素之和为顶点 v_i 的入度。

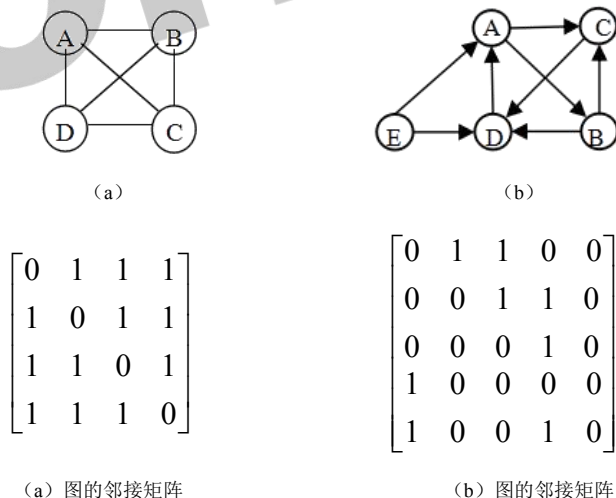


图 6-5 图的邻接矩阵

【习题演练】

1. n 个顶点的连通图用邻接矩阵表示时，该矩阵至少有（ ）个非零元素。

- A. n B. $2(n-1)$ C. $n/2$ D. n^2

二、邻接表

邻接表是图的一种链式存储结构。在邻接表中，对图中的每个顶点建立一个单链表，第 i 个单链表的结点表示依附于顶点 v_i 的边（对有向图是以顶点 v_i 为尾的弧）。每个结点由三个域组成，其中邻接点域（adjvex）指示与顶点 v_i 邻接的点在图中的位置，链域（nextarc）指示下一条边或弧结点；数据域（info）存储和边或弧相关的信息，如权值等。每个链表上附设一个表头结点。在表头结点中，除了设有链域（firstarc）指向链表中的第一个结点之外，还设有存储顶点 v_i 的名或其他有关信息的数据域（data）。如下所示：

表结点			头结点	
adjvex	nextarc	info	data	firstarc

这些表头结点（可以链相接）通常以顺序结构的形式存储，以便随机访问任一顶点的链表。如图 6-5 为无向图和有向图的邻接矩阵。

在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中的结点数；而在有向图中，第 i 个链表中的结点数只是顶点 v_i 的出度，为求入度必须遍历整个邻接表。在所有链表中其邻接点域的值为 i 的结点的个数是顶点 v_i 的入度。

一个图的邻接表存储结构可形式地说明如下：

```
typedef struct ArcNode { // 弧结点
    int adjvex; // 邻接点
    struct ArcNode *nextarc; // 下一个邻接点
    InfoType *info;
} ArcNode;

typedef struct VexNode { // 顶点结点
    VertexType data; // 顶点信息
    ArcNode *firstarc; // 第一个邻接点
} VexNode;

const int MAX_VERTEX = 100; // 最大顶点个数;

typedef struct Graph { // 图
    VexNode vexs[MAX_VERTEX]; // 顶点向量
    int vexnum, arcnum; // 顶点和弧的个数
} Graph;
```

在邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，但要判定任意两个顶点（ v_i 和 v_j ）直接是否有边或弧相连，则需搜索第 i 个或第 j 个链表，因此，不及邻接矩阵方便。邻接表适用于稀疏图。

注意：顶点的邻接点是无所谓次序的，所以图的邻接表的表示并不唯一。

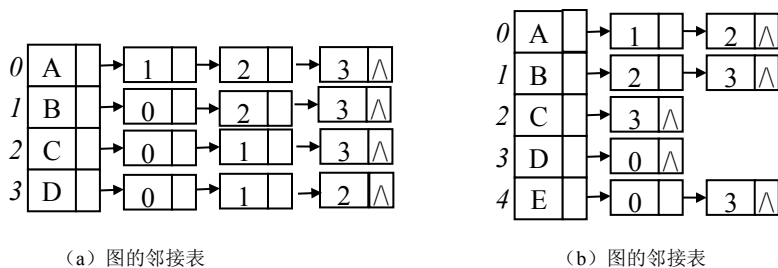


图 6-6 图的邻接表

三、逆邻接表

为了便于确定顶点的入度或以顶点 v_i 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 v_i 建立一个链接以 v_i 为头的弧的表，例如图 6-7 所示为有向图的逆邻接表。

类型定义类似邻接表。

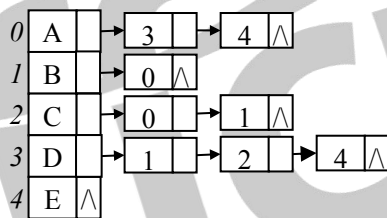


图 6-7 图的逆邻接表

第三节 图的遍历

从图中的某一顶点出发访遍图中其余顶点，且使每一个顶点仅被访问一次。这个过程就叫做图的遍历。然而，图的遍历要比树的遍历复杂得多。因为图的任一顶点都可能和其余的顶点相邻接。所以在访问了某个顶点之后，可能沿着某条路径搜索之后，又回到该顶点上。为了避免同一顶点被访问多次，在遍历图的过程中，须记下每个已访问过的顶点。为此，我们可以设一个辅助数组 $visited[0 \dots n-1]$ ，它的初始值置为“假”或者零，一旦访问了顶点 v_i ，便置 $visited[i]$ 为“真”或者为被访问时的次序号。

通常有两条遍历图的路径：深度优先搜索和广度优先搜索。它们对无向图和有向图都适用。

一、深度优先遍历

1. 遍历思想

深度优先搜索（Depth First Search）遍历类似于树的先根遍历。

深度优先搜索可从图中某个顶点 v 出发，访问此顶点，然后依次从 v 未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未被访问的顶点作为起始点，重复上述过程，直到图中所有顶点都被访问为止。

以图 6-12 (a) 中无向图 G 为例，深度优先搜索遍历图的过程如图 6-12 (b) 所示。假设从顶点 v_1 出发进行搜索，在访问了顶点 v_1 之后，选择邻接点 v_2 。因为 v_2 未曾访问，则从 v_2 出发进行搜索。依次类推，接着从 v_4 ， v_8 ， v_5 出发进行搜索。在访问了 v_5 之后，由于 v_5 的邻接点都被访问，则搜索回到 v_8 。由于同样的理由，搜索继续回到 v_4 ， v_2 直至 v_1 ，此时由于 v_1 的另一个邻接点未被访问，则搜索又从 v_1 到 v_3 ，再继续进行下去。由此，得到的顶点访问序列为：

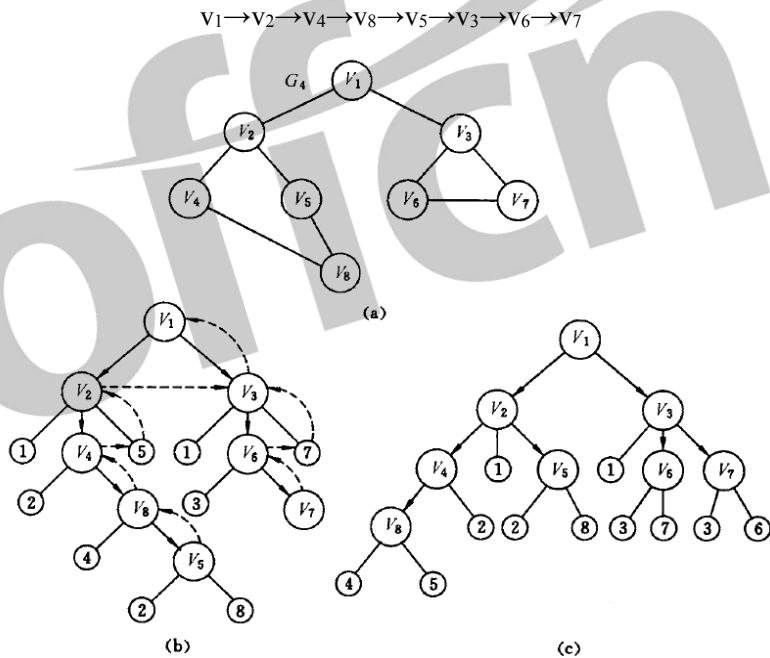


图 6-8 图的遍历

```
void DFSTraverse ( Graph G )
{
    visited [0 .. G.vexnum-1] = false;    // 初始化访问标志为未访问(false)
    for ( v=0; v<G.vexnum; v++ )
        if ( ! visited[v] ) DFS ( G, v );    // 从未被访问的顶点开始 DFS
}
```

```

void DFS ( Graph G, int v )
{
    visit ( v );  visited [v] = true;  // 访问顶点 v 并作标记
    for ( w=FirstAdjVex(G,v); w>=0; w=NextAdjVex(G,v,w) )
        if ( ! visited[w] )  DFS ( G, w );    // 分别从每个未访问的邻接点开始 DFS
}

```

其中的 FirstAdjVex(G,v)表示图 G 中顶点 v 的第一个邻接点, NextAdjVex(G,v,w)表示图 G 中顶点 v 的邻接点 w 之后 v 的下一个邻接点。

2. 算法性能分析

遍历图的过程实质上是对每个顶点查找其邻接点的过程。

深度优先搜索是一个递归的过程。需要借助一个递归工作栈, 故空间复杂度为 $O(n)$ 。其耗费的时间则取决于所采用的存储结构。当用邻接表存储图时, 其时间复杂度为 $O(n+e)$; 当采用邻接矩阵作为存储结构时, 时间复杂度是 $O(n^2)$ (因为求一个顶点的所有邻接点就是搜索邻接矩阵的一行中的 n 个数, 而顶点的个数为 n , 总共就是 n^2)。

3. 深度优先生成树

在深度遍历的过程中, 我们可以得到一棵遍历树, 称为深度优先生成树。需要注意的是, 一给定图的邻接矩阵存储表示时唯一的, 故其深度优先生成树也是唯一的, 但由于邻接表存储表示不唯一, 故其深度优先生成树也是不唯一的。

下面以图 6-13 的有向图为例, 画一棵“深度优先搜索树”:

分析: 从 A 出发, 访问 A (画圈作标记), A 的邻接点有 B 和 C (作为 A 的孩子), B 未访问, 访问 B (画圈), B 的邻接点有 E (作 B 的孩子), ..., 以此类推, 画出搜索树。深度优先搜索的过程就是沿着该搜索树先根遍历的过程。

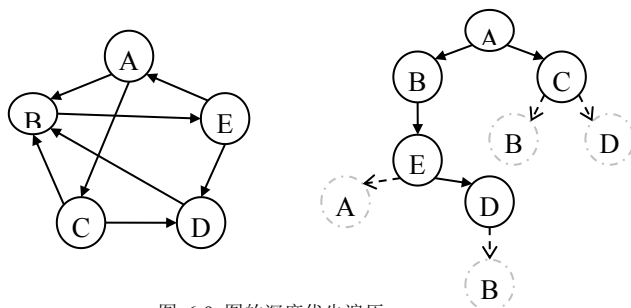


图 6-9 图的深度优先遍历

注意: 顶点的邻接点是无所谓次序的, 所以同一个图的深度优先遍历序列可能不同, 但在遍历时 (除非邻接点的次序有明确规定) 一般按照编号顺序安排邻接点的次序。

【习题演练】

1. 若从无向图的任意一个顶点出发进行一次深度优先搜索可以访问图中所有的顶点，则该图一定是（ ）图。
A. 非连通 B. 连通 C. 强连通 D. 有向
2. 用邻接表表示图进行深度优先遍历时，通常借助（ ）来实现算法。
A. 栈 B. 队列 C. 树 D. 图
3. 深度优先遍历类似于二叉树的（ ）。
A. 先序遍历 B. 中序遍历 C. 后序遍历 D. 层次遍历

二、广度优先遍历

1. 遍历思想

广度优先搜索（Depth First Search）遍历类似于树的层次遍历。

广度优先搜索是从图中某顶点 v 出发，访问 v 顶点之后依次访问 v 的各个未被访问的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作为起始点，重复以上过程，直到图中所有顶点都被访问为止。

广度优先搜索遍历图的过程是以 v 为起始点，由近至远，依次访问和 v 有路径相通且路径长度为 1, 2, ... 的顶点。

以图 6-12 (a) 中无向图 G 为例，深度优先搜索遍历图的过程如图 6-12 (c) 所示。首先访问 v_1 和 v_1 的邻接点 v_2 和 v_3 ，然后依次访问 v_2 的邻接点 v_4 和 v_5 及 v_3 的邻接点 v_6 和 v_7 ，最后访问 v_4 的邻接点 v_8 。由于这些顶点的邻接点均已被访问，并且图中所有顶点都被访问，由此完成了图的遍历。得到的顶点访问序列为：

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \rightarrow v_8$$

```
void BFSTraverse ( Graph G ){
    visited [0 .. G.vexnum-1] = false;    // 初始化访问标志为未访问(false)
    InitQueue ( Q );
    for ( v=0; v<G.vexnum; v++ )
        if ( ! visited[v] ) {
            // 从 v 出发广度优先搜索
            visit ( v );    visited [v] = true;
```

```

EnQueue ( Q, v );
while ( ! QueueEmpty(Q) ) {
    DeQueue ( Q, u );
    for ( w=FirstAdjVex(G,u); w>=0; w=NextAdjVex(G,u,w) )
        if ( ! visited[w] ) {
            visit ( w );    visited [w] = true;
            EnQueue ( Q, w );
        }
    }
}
}
}

```

2. 算法性能分析

在广度优先搜索算法中，每个顶点至多进一次队列，所以在最坏情况下，空间复杂度为 $O(n)$ 。其遍历过程实质上是通过边和弧找邻接点的过程，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，两者不同之处在于对顶点的访问顺序不同。

3. 广度优先生成树

在广度遍历的过程中，我们可以得到一棵遍历树，称为广度优先生成树。需要注意的是，一给定图的邻接矩阵存储表示时唯一的，故其深度优先生成树也是唯一的，但由于邻接表存储表示不唯一，故其深度优先生成树也是不唯一的。

下面以图 6-14 的有向图为例，画一棵“广度优先搜索树”：

分析：画“广度优先搜索树”。与深度优先搜索树类似，A 为根，其邻接点为其孩子，访问一个顶点，则扩展出其孩子。不过广度优先搜索的访问次序是对该树按层遍历的结果。

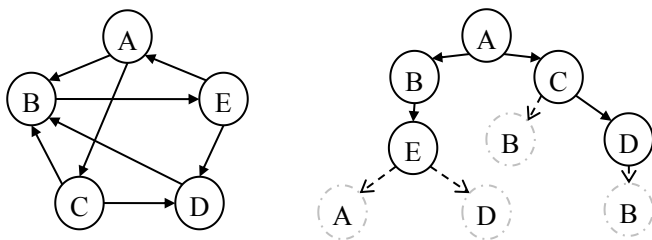


图 6-10 图的广度优先遍历

【习题演练】

1. 对有 n 个结点、 e 条边且使用邻接表存储的有向图进行广度优先遍历，其算法时间复杂度是（ ）。

- A. $O(n)$ B. $O(e)$ C. $O(n+e)$ D. $O(n*e)$
2. 用邻接表表示图进行广度优先遍历时，通常借助（ ）来实现算法。
- A. 栈 B. 队列 C. 树 D. 图
3. 广度优先遍历类似于二叉树的（ ）。
- A. 先序遍历 B. 中序遍历 C. 后序遍历 D. 层次遍历
4. 图的 BFS 生成树的树高比 DFS 生成树的树高（ ）。
- A. 小 B. 相等 C. 小或相等 D. 大或相等

三、图的连通性问题

图的遍历算法可以用来判断图的连通性。

对于无向图来说，如果无向图是连通的，则从任一结点出发，仅需一次遍历就能够访问图中所有顶点；如果无向图是非连通的，则从某一个顶点出发，一次遍历只能访问到该顶点所在连通分量的所有顶点，而对于图中其他连通分量的顶点，则无法通过这次遍历访问。对于有向图来说，若从初始点到图中的每个顶点都有路径，则能够访问到图中的所有顶点，否则不能访问到所有顶点。

故而在 $\text{BFS}\text{Traverse}()$ 或 $\text{DFS}\text{Traverse}()$ 中添加了第二个 for 循环，再选取初始点，继续进行遍历，以防止一次无法遍历图的所有顶点。对于无向图，上述两个函数调用 $\text{BFS}(G, i)$ 或 $\text{DFS}(G, i)$ 的次数等于该图的连通分量数；而对于有向图，则不是这样，因为一个连通的有向图分为强连通的和非强连通的，它的连通子图也分为强连通分量和非强连通分量，非强连通分量一次调用 $\text{BFS}(G, i)$ 或 $\text{DFS}(G, i)$ 无法访问到该连通分的所有顶点。

第四节 最小生成树

一、最小生成树

1. 最小生成树

一个连通网 $G=(V,E)$ 的生成树不同，每棵树的权（即树中所有边上的权值之和）也可能不同。我们把边的权值之和最小那棵生成树称为 G 的最小生成树。

注意：同一个连通网的最小生成树可能是不唯一的，但其代价都是最小(唯一的)。

最小生成树的边数为顶点数减 1。

构造最小生成树的算法有很多种，经典的算法有普里姆算法和克鲁斯卡尔算法。

2. 普里姆 (Prim) 算法

算法思想：

(1) 输入：一个加权连通图，其中顶点集合为 V ，边集合为 E ；

(2) 初始化： $V_{\text{new}} = \{x\}$ ，其中 x 为集合 V 中的任一节点（起始点）， $E_{\text{new}} = \{\}$ ，为空；

(3) 重复下列操作，直到 $V_{\text{new}} = V$ ：

①在集合 E 中选取权值最小的边 $\langle u, v \rangle$ ，其中 u 为集合 V_{new} 中的元素，而 v 不在 V_{new} 集合当中，并且 $v \in V$ （如果存在有多条满足前述条件即具有相同权值的边，则可任意选取其中之一）；

②将 v 加入集合 V_{new} 中，将 $\langle u, v \rangle$ 边加入集合 E_{new} 中；

(4) 输出：使用集合 V_{new} 和 E_{new} 来描述所得到的最小生成树。

普里姆算法的时间复杂度为 $O(n^2)$ ，与网中的边数无关，因此适用于求边稠密的网的最小生成树。图 6-17 为 Prim 算法构造最小生成树的过程。

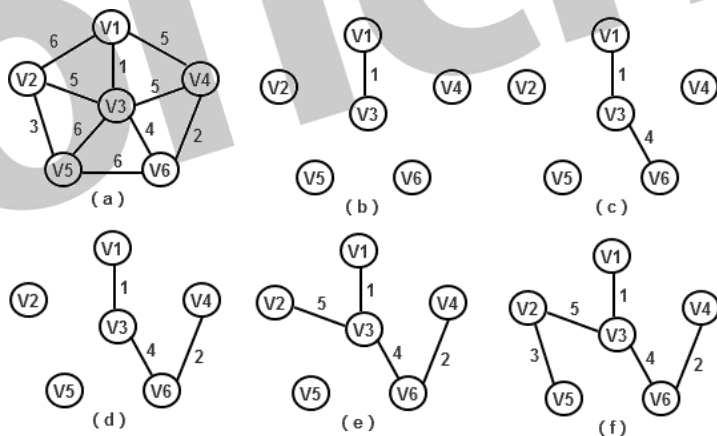


图 6-11 Prim 算法构造最小生成树

3. 克鲁斯卡尔 (Kruskal) 算法

克鲁斯卡尔算法是一种按权值的递增次序选择合适的边来构造最小生成树的方法。

假设 $N=(V,E)$ 是连通网，对应的最小生成树 $T=(V_T, E_T)$ 其算法的思想如下：

初始化： $V_T=V$ ， $E_T=\{\}$ 。即每个顶点构成一棵独立的树（即图中每个顶点自成一个连通分量）， T 此时是一个仅含 n 个顶点的森林；

循环（重复下列操作至 T 是一棵树）：按 N 的边的权值递增顺序依次从 $E-E_T$ 中选择一条边如果这条边加入 T 后不构成环（即该边依附的顶点落在 T 中不同的连通分量上），则将其加入 E_T ，否则舍弃，直到 E_T 中含有 $n-1$ 条边。

用堆存放网的边，则每次选择最小代价的边仅需 $O(\log_2 e)$ 的时间，克鲁斯卡尔算法的时间复杂度为 $O(e \log e)$ ，与网中的顶点数无关，因此适用于求边稀疏的网的最小生成树。图 6-18 为 Kruskal 算法构造最小生成树的过程。

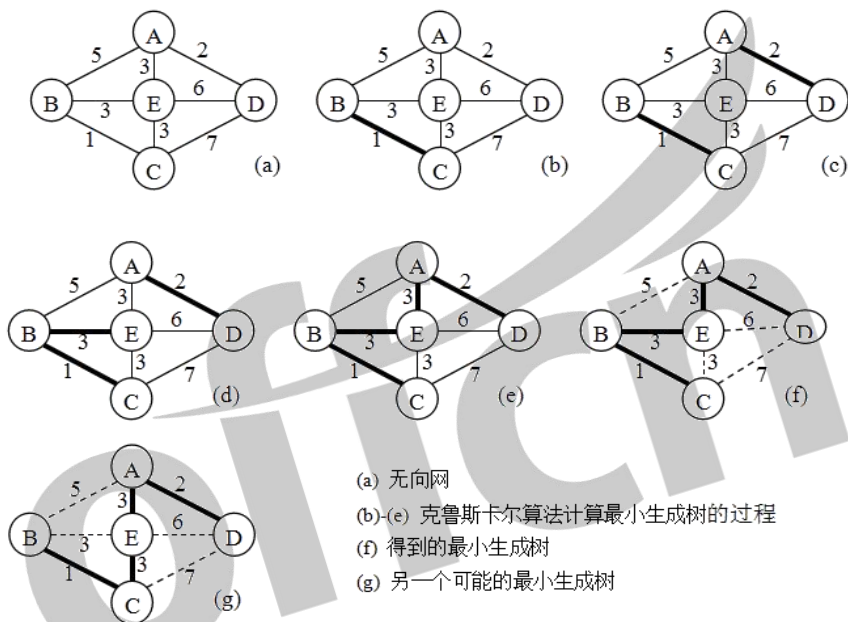


图 6-12 克鲁斯卡尔算法构造最小生成树

提示：不构成环的情况下，每次选取最小边。

4. 两种算法比较

表 6-1 普里姆算法和克鲁斯卡尔算法的比较

算法	普里姆算法	克鲁斯卡尔算法
时间复杂度	$O(n^2)$	$O(e \log e)$
	只与顶点个数 n 有关	只与边的数目 e 有关
特点	与边的数目 e 无关	与顶点个数 n 无关
	适用于稠密图	适用于稀疏图

【习题演练】

1. 下列关于最小生成树的叙述中，正确的是 ____。

- I. 最小生成树的代价唯一
 II. 所有权值最小的边一定会出现在所有的最小生成树中
 III. 使用普里姆 (Prim) 算法从不同顶点开始得到的最小生成树一定相同
 IV. 使用普里姆算法和克鲁斯卡尔 (Kruskal) 算法得到的最小生成树总不相同
- A. 仅 I B. 仅 II C. 仅 I、III D. 仅 II、IV
2. 下面 () 算法适合构造一个稠密图 G 的最小生成树。
- A. Prim 算法 B. Kruskal 算法 C. Floyd 算法 D. Dijkstra 算法
3. 已知如图 6-19 所示的无向网, 请给出最小生成树。

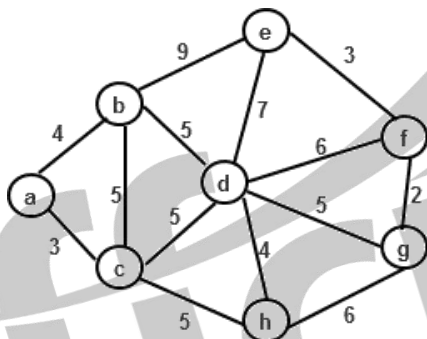


图 6-13 无向网

第五节 有向无环图及其应用

一个无环的有向图称做有向无环图, 简称 DAG 图。有向图是描述含有公共子式的表达式的有效工具, 也是描述一项工程或系统的进行过程的有效工具。

一、拓扑排序

AOV 网: 用顶点表示活动, 用弧表示活动间优先关系的有向图称为顶点表示活动的网。简称 AOV 网。在 AOV 网中, 活动 V_i 是活动 V_j 的直接前驱, 活动 V_j 是活动 V_i 的直接后继, 这种前驱和后继关系具有传递性, 且任何活动 V_i 不能以它自己作为自己的前驱或后继。

拓扑排序: 在图论中, 由一个有向无环图的顶点组成的序列, 当且仅当满足下列条件时, 称为该图的一个拓扑排序。

- (1) 每个顶点出现且只出现一次。
- (2) 若顶点 A 在序列中排在顶点 B 的前面, 则在图中不存在从顶点 B 到顶点 A 的

路径。

或者定义为：拓扑排序是对有向无环图的顶点的一种排序，它使得如果存在一条从顶点 A 到顶点 B 的路径，那么在排序中顶点 B 出现在顶点 A 的后面。每个 DAG 图都有一个或多个拓扑排字序列。

如何进行拓扑排序？解决步骤如下：

(1) 在有向图中选一个没有前驱（入度为 0）的顶点且输出之。

(2) 从图中删除该顶点和所有以它为尾的弧。

重复上述两步，直至全部顶点均已输出，或者当前图中不存在无前驱的顶点为止。

后一种情况则说明有向图中存在环。

由于输出每个顶点的同时还要删除以它开始的边，故拓扑排序的时间复杂度为 $O(n+e)$ 。

例：以下 DAG 图拓扑排序的结果是：ABCDE。

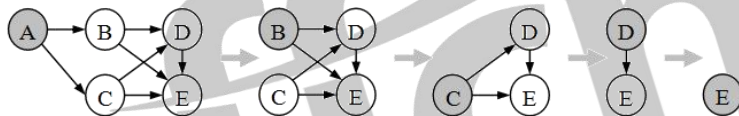


图 6-14 拓扑排序执行图

注意：拓扑排序的结果不一定是唯一的。如：ACBDE 也是以上 DAG 图的拓扑有序序列。

【习题演练】

1. 对图6-25进行拓扑排序，可以得到不同的拓扑序列的个数是 _____。

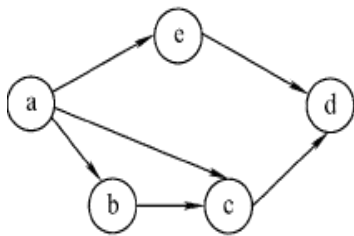


图 6-15

- | | |
|------|------|
| A. 4 | B. 3 |
| C. 2 | D. 1 |

2. 下列关于图的叙述中，正确的是_____。

I. 回路是简单路径

- II. 存储稀疏图, 用邻接矩阵比邻接表更省空间
 III. 若有向图中存在拓扑序列, 则该图不存在回路
- A. 仅 II B. 仅 I、II C. 仅 III D. 仅 I、III
3. 下面 () 方法可以判断出一个有向图是否有环。
- A. 深度优先遍历 B. 拓扑排序 C. 求最短路径 D. 求关键路径

二、关键路径

AOE 网是一个带权的有向无环图, 其中, 顶点表示事件, 弧表示活动, 权表示活动持续的时间。通常, AOE 网可用来估算工程的完成时间。在 AOE 网中, 只有一个入度为零的点 (称做源点) 和一个出度为零的点 (叫做汇点), 事件 i 发生后, 其后继活动 $a(i,*)$ 都可以开始; 只有所有先导活动 $a(*,j)$ 都结束后, 事件 j 才发生, 如图 6-16 所示。

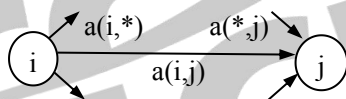


图 6-16

在 AOE 网中有些活动可以并行地进行, 所以完成工程的最短时间是从开始点到完成点的最长路径的长度 (这里所说的路径长度是指路径上各活动持续时间之和, 不是路径上弧的数目) 路径长度最长的路径叫做**关键路径**。关键路径上的所有活动都是关键活动 (即最迟开始时间等于最早开始时间的活动)。只要找到关键活动, 就找到了关键路径。

关键路径算法

假设:

事件 (顶点) i : 最早发生时间 $ve(i)$, 最晚发生时间 $vl(i)$;

活动 (边) $a(i,j)$: 最早开始时间 $e(i,j)$, 最晚开始时间 $l(i,j)$ 。

于是, 整个工程完工的时间就是终点的 earliest 发生时间。

求关键路径的算法步骤:

(a) 按拓扑有序排列顶点: 对顶点拓扑排序;

(b) 计算 $ve(j)$:

$$\begin{cases} ve(1) = 0, \\ ve(j) = \max \{ve(*) + a(*, j)\} \end{cases} \quad \text{其中 } * \text{ 为任意前驱事件;}$$

(c) 计算 $vl(i)$:

$$\begin{cases} vl(n) = ve(n), \\ vl(i) = \min\{vl(*) - a(i,*)\} \end{cases} \quad \text{其中*为任意后继事件;}$$

(d) 计算 $e(i,j)$ 和 $l(i,j)$:

$$\begin{aligned} e(i,j) &= ve(i), \\ l(i,j) &= vl(j) - a(i,j) \end{aligned}$$

(e) 结论: 工程总用时 $ve(n)$, 关键活动是 $e(i,j)=l(i,j)$ 的活动 $a(i,j)$ 。

说明:

(1) 若只求工程的总用时只要进行步骤(a)-(b)即可求得。

(2) 如何理解计算 $ve(j)$ 和 $vl(i)$ 的公式:

事件 j 在所有前驱活动都完成后发生, 其最早发生时间 $ve(j) = \max\{ve(*) + a(*,j)\}$, 即取决于最慢的前驱活动。

另一方面, 事件 i 发生后所有后继活动都可以开始了, 所以其最晚发生时间 $vl(i) = \min\{vl(*) - a(i,*)\}$, 即不耽误最慢的后继活动。



图 6-17

对于关键路径, 我们需要注意以下几点:

①关键路径上的所有活动都是关键活动, 它是决定整个工程的关键因素, 因此可通过加快关键活动来缩短整个工程的工期。但也不能任意缩短关键活动, 因为一旦缩短到一定的程度, 该关键活动可能变成非关键活动了。

②网中的关键路径并不唯一。且对于有几条关键路径的网, 只提高一条关键路径上的关键活动速度并不能缩短整个工程的工期, 只有加快那些包括在所有关键路径上的关键活动才能达到缩短工期的目的。

例: 某工程的 AOE 网如图 6-18, 求(1) 整个工程完工需要多长时间, (2) 关键路径。

说明: 图中的虚线仅表示事件的先后关系, 不代表具体活动。

分析: 按照拓扑有序排列顶点, 然后“从前往后”计算事件的最早发生时间得到总时间, 再“从后往前”计算事件的最晚发生时间, 最后计算活动的最早和最晚开始时间得到关键活动和关键路径。

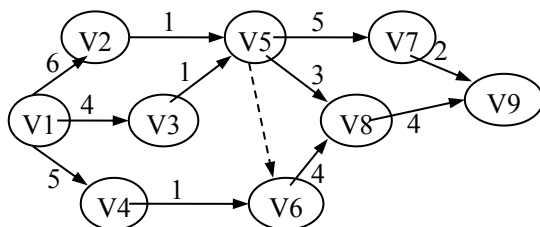


图 6-18 AOE 网

表 6-2 关键路径

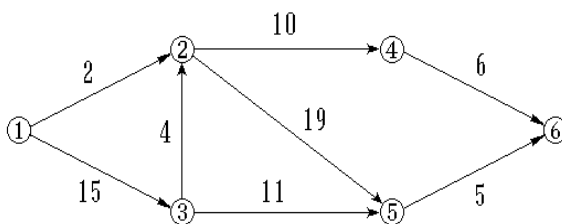
事件	最早发生时间 v_e	最晚发生时间 v_l	活动	最早开始时间 e	最晚开始时间 l
v1	0	0	a(1,2)	0	0
v2	6	6	a(1,3)	0	2
v3	4	6	a(1,4)	0	1
v4	5	6	a(2,5)	6	6
v5	7	7	a(3,5)	4	6
v6	7	7	a(4,6)	5	6
v7	12	13	a(5,6)	7	7
v8	11	11	a(5,7)	7	8
v9	15	15	a(5,8)	7	8
			a(6,8)	7	7
			a(7,9)	12	13
			a(8,9)	11	11

所以，(1)工程完工需要时间 15；(2)关键路径是 $V1 \rightarrow V2 \rightarrow V5 \rightarrow V6 \rightarrow V8 \rightarrow V9$ 。

【习题演练】

1. 对下图的 AOE-网：

- ① 求这个工程最早可能在什么时间结束；
- ② 求每个活动的最早开始时间和最迟开始时间；
- ③ 确定哪些活动是关键活动



第六节 最短路径

求解最短路径的算法通常都依赖于一种性质，也就是两点之间的最短路径也包含了路径上其他顶点间的最短路径。带权有向图 G 的最短路径问题，一般可分为两类：一是求图中某一顶点到其他各顶点的最短路径，可通过经典的 Dijkstra 算法求解；二是求每一对顶点间的最短路径，可通过 Floyd-Warshall 算法来求解。

一、迪杰斯特拉（Dijkstra）算法

求带权有向图一个顶点到其他各顶点的最短路径。该算法设置一个集合 S 记录已求得的最短路径的顶点，可用一个数组 $s[]$ 来实现，初始化为 0，当 $s[v_i]=1$ 时表示将顶点 v_i 放入 S 中，初始时把源点 v_0 放入 S 中。此外，在构造过程中还设置了两个辅助数组：

$dist[]$ ：记录了从源点 v_0 到其他各顶点最短路径长度， $dist[i]$ 初值为 $arcs[v_0][i]$ 。

$path[]$ ： $path[i]$ 表示从源点到顶点 v_i 之间的最短路径的前驱结点，在算法结束时，可根据其值追溯得到源点 v_0 到顶点 v_i 的最短路径。

假设从顶点 0 出发，即 $v_0=0$ ，集合 S 最初只包含顶点 v_0 ，邻接矩阵 $arcs$ 表示带权有向图， $arcs[i][j]$ 表示有向边 $\langle i, j \rangle$ 的权值，若不存在有向边 $\langle i, j \rangle$ ，则 $arcs[i][j]$ 为 ∞ 。Dijkstra 算法的步骤如下（不考虑对 $path$ 的操作）：

- (1) 初始化：集合 S 初始为 $\{0\}$ ， $dist[]$ 的初始值 $dist[i]=arcs[i][j]$ ， $i=1, 2, \dots, n-1$ 。
- (2) 从顶点集合 $V-S$ 中选出 v_j ，满足 $dist[j]=\min\{dist[i]|v_i \in V-S\}$ ， v_j 就是当前求得的一条从 v_0 出发的最短路径的终点，令 $S=S \cup \{j\}$ 。
- (3) 修改从 v_0 出发到集合 $V-S$ 上任一顶点 v_k 可达的最短路径长度：如果 $dist[j]+arcs[j][k]<dist[k]$ ，则令 $dist[k]=dist[j]+arcs[j][k]$ 。
- (4) 重复 (2) ~ (3) 操作共 $n-1$ 次，直到所有的顶点都包含在 S 中。

迪杰斯特拉算法求解单源点到其所有顶点的最短路径，时间复杂度也为 $O(n^2)$ 。如果要找出所有结点对之间的最短路径，则需要对每个结点运行一次迪杰斯特拉算法，即时时间复杂度为 $O(n^3)$ 。例：用迪杰斯特拉算法求图 6-19 中 A 到其余顶点的最短路径如表 6-3。

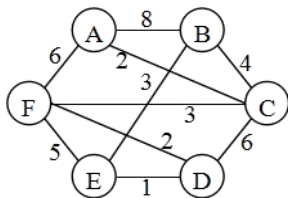


图 6-19

表 6-3 迪杰斯特拉算法

终点	第一趟	第二趟	第三趟	第四趟	第五趟
B	8 AB	6 ACB	6 ACB		
C	2 AC				
D	∞	8 ACD	7 ACFD	7 ACFD	
E	∞		10 ACFE	9 ACBE	8 ACFDE
F	6 AF	5 ACF			
S 数组	AC	ACF	ACFB	ACFBD	ACFBDE

【习题演练】

1. 如图6-20有向带权图，若采用迪杰斯特拉（Dijkstra）算法求从源点a到其他各顶点的最短路径，则得到的第一条最短路径的目标顶点是b，第二条最短路径的目标顶点是c，后续得到的其余各最短路径的目标顶点依次是_____。

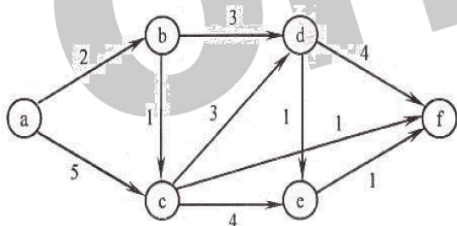


图 6-20 有向带权图

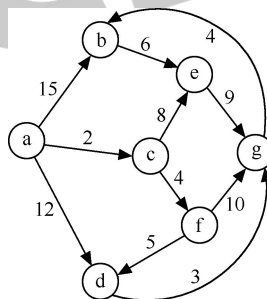


图 6-21 有向网

- A. d, e, f B. e, d, f C. f, d, e D. f, e, d

2. 有向网如图 6-21 所示，试用迪杰斯特拉算法求出从顶点 a 到其他各顶点间的最短路径。

二、弗洛伊德（Floyd）算法

求每一对顶点之间的最短路径。

求所有顶点之间的最短路径问题描述如下：已知一个各边权值均大于 0 的带权有向图，对每一对顶点 $v_i \neq v_j$ ，要求求出 v_i 与 v_j 之间的最短路径和最短路径长度。

Floyd 算法的基本思想是：递推产生一个 n 阶方阵序列 $A^{(-1)}, A^{(0)}, \dots, A^{(k)}, \dots, A^{(n-1)}$ ，其中 $A^{(k)}[i][j]$ 表示从顶点 v_i 到顶点 v_j 的路径长度， k 表示绕行第 k 个顶点的运算步骤。初始时，对于任意两个顶点 v_i 和 v_j ，若它们之间存在边，则以此边上的权值作为它们之间的最短路径长度；若它们之间不存在有向边，则以 ∞ 作为它们之间的最短路径长度。以后逐步尝试在原路径中加入顶点 k ($k=0, 1, \dots, n-1$) 作为中间顶点。如果增加中间顶点后，得到的路径比原来的路径长度减少了，则以此新路径代替原路径。算法的描述如下：

定义一个 n 阶方阵序列： $A^{(-1)}, A^{(0)}, \dots, A^{(k)}, \dots, A^{(n-1)}$ ，其中

$$A^{(-1)}[i][j] = \text{arcs}[i][j]$$

$$A^{(k)}[i][j] = \min\{A^{(k-1)}[i][j], A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\}, k=0, 1, \dots, n-1$$

其中， $A^{(0)}[i][j]$ 是从顶点 v_i 到 v_j 、中间顶点是 v_0 的最短路径的长度， $A^{(k)}[i][j]$ 是从顶点 v_i 到 v_j 、中间顶点的序号不大于 k 的最短路径的长度。Floyd 算法是一个迭代的过程，每迭代一次，在从 v_i 到 v_j 的最短路径上就多考虑了一个顶点；经过 n 次迭代后所得到的 $A^{(n-1)}[i][j]$ 就是 v_i 到 v_j 的最短路径长度，即方阵 $A^{(n-1)}$ 中就保存了任意一对顶点之间的最短路径长度。

Floyd 算法的时间复杂度为 $O(n^3)$ ，Floyd 算法同样适用于带权无向图。

下面通过实例来说明 Floyd 算法的过程，如图 6-22 所示。

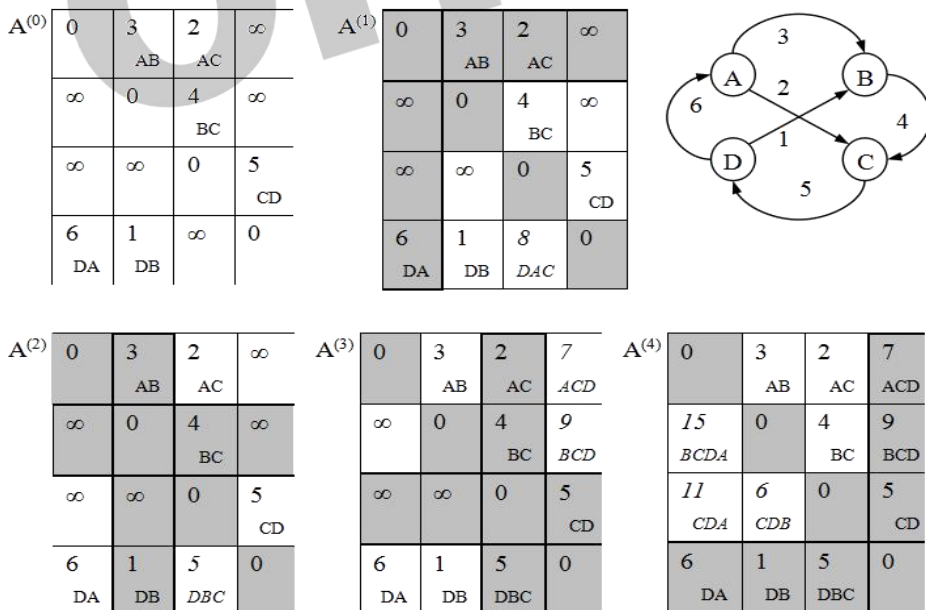


图 6-22 弗洛伊德算法图解

依次计算 $A^{(0)}, A^{(1)}, \dots, A^{(n)}$ 。 $A^{(0)}$ 为邻接矩阵，计算 $A^{(k)}$ 时， $A^{(k)}(i,j) = \min\{A^{(k-1)}(i,j),$

$A^{(k-1)}(i,k)+A^{(k-1)}(k,j)\}$ 。

技巧：计算 $A^{(k)}$ 时，第 k 行、第 k 列、对角线的元素保持不变，对其余元素，考查 $A(i,j)$ 与 $A(i,k)+A(k,j)$ （“行+列”），如果后者更小则替换 $A(i,j)$ ，同时修改路径。当不变行或不变列（即第 k 行、第 k 列）某元素为 ∞ 时，其所在的列或行元素也不变。例如：计算 $A^{(1)}$ 时， $A(2,1)=A(3,1)=\infty$ ，所以第 2、3 行都不变，而 $A(1,4)=\infty$ ，所以第 4 列也不变，这样，只剩下 $A(4,2)$ 和 $A(4,3)$ 需要计算了。

offcn

第七章 查找

【本章综述】

本章是考研命题的重点，特别是散列查找和折半查找，容易考计算题，对于散列查找，应掌握散列表的构造、冲突处理方法、查找成功和查找失败的平均查表长度、散列查找的特征和性能分析。对于折半查找，读者应掌握折半查找的过程、构造判定树、分析查找成功和查找失败的平均查找长度等；另外折半查找只能基于顺序表，因此也比较容易结合顺序表出算法设计题。B 树和 B⁺树是本章的难点，考纲仅要求了解 B⁺树的基本概念和性质，而 B 树则要求掌握插入、删除和查找的操作过程，不要求掌握算法。

【复习重点】

（一）静态查找表

1. 顺序表查找
2. 折半查找
3. 索引顺序表查找

（二）动态查找表

1. 二叉排序树和平衡二叉树
2. B 树和 B⁺树

（三）哈希表

1. 哈希表
2. 哈希函数
3. 解决冲突的方法

第一节 静态查找表

一、顺序表查找

1. 思路

从表中最后一个记录开始，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值比较相等，则查找成功，找到所查记录；反之，若直至第一个记录，其关键字和给定值比较都不相等，则表明表中没有所查记录，查找不成功。

2. 算法

【例 7-1】在数组 a 的前 n 个元素中查找 x

```
int Search ( int a[], int n, int x )
{
    for ( i=n-1; i>=0; i-- )
        if ( a[i]==x ) return i;
    return -1; // -1 表示找不到
}
```

3. 分析

(1) 平均查找长度

对于有 n 个元素的表，假定查找的元素在第 i 个位置，则找到第 i 个位置时，需进行 i 次关键字比较（从前往后找，如果是从后往前找，则需比较 n-i+1 次），一般在等概率情况下，查找成功时，平均查找长度：

$$ASL = \frac{1+2+\dots+n}{n} = \frac{n+1}{2}$$

查找失败时，如果是无序表，则从表的一端查找找到另一端，比较 n 次，平均查找长度 $ASL=n$ 。若为有序表，则查找到表中某一位置便可判定查找失败，平均查找长度

$$ASL = \frac{1+2+\dots+n+n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

(2) 判定树

判定树是一种描述查找中比较过程的直观形式，每个关键字所在层次就是其查找长度，有利于分析查找过程。顺序查找的判定树是一棵深度为 n 的单分支的树。课本上顺序查找从 a_n 开始，当然也可以从 a_1 开始。

(3) 时间复杂度

从平均查找长度看顺序查找的时间复杂度是 $O(n)$ 。

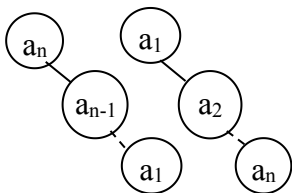


图 7-1 顺序查找的深度

【习题演练】

1. 对 n 个元素的表做顺序查找时，若查找每个元素的概率相同，则平均查找长度为（ ）。

- A. $(n-1)/2$ B. $n/2$ C. $(n+1)/2$ D. n

二、折半查找

1. 思路

待查找的表必须是有序的，先从中间位置开始比较，比较一次查找的范围便可相应的缩小一半，在缩小的范围中继续折半查找，直到查找成功或失败。

2. 算法

要熟练掌握该算法。设 $a[]$ 升序有序，有以下算法：

```
int BinarySearch ( DataType a[], int n, DataType x )
{
    low = 0; high = n-1;
    while ( low <= high ) {
        mid = ( low + high )/2;    // 折半
        if ( a[mid]==x )
            return mid;    // 找到
        else if ( x < a[mid] )    // x 位于低半区 [low..mid-1]
            high = mid - 1;
        else    // x 位于高半区 [mid+1..high]
            low = mid + 1;
    }
    return -1;    // -1 表示未找到
}
```

或者有递归版本：

```
int BinarySearch ( DataType a[], int low, int high, DataType x )
{
    if ( low > high ) return -1;    // 查找失败
    mid = (low+high)/2;    // 折半
    if ( a[mid]==x )
        return mid;    // 找到
    else if ( x < a[mid] )
        return BinarySearch ( a, low, mid-1, x );
    else
        return BinarySearch ( a, mid+1, high, x );
}
```

```
return BinarySearch(a, mid+1, high, x);
}
```

3. 分析

速度很快，要求查找表是有序的，而且随机访问（以便计算折半的下标）。所以，链表不能进行折半查找（但可以采用二叉排序树等形式进行快速的查找）。

（1）判定树

折半查找的判定树类似于完全二叉树，叶子结点所在层次之差最多为 1，其深度为 $\lfloor \log_2 n \rfloor + 1$ 。

查找过程就是走了一条从根到该结点的路径。

如：表长 $n=10$ 的有序表，

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

折半查找的判定树如下：

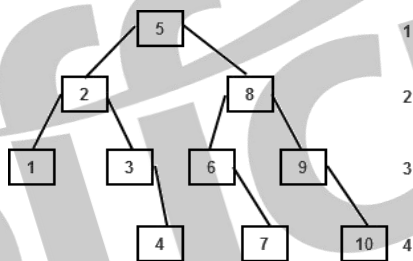


图 7-2 折半查找的判定树

（2）平均查找长度

等概率下查找成功时的平均查找长度

$$ASL = \frac{1}{n} \sum_{j=1}^h j 2^{j-1} = \frac{n+1}{n} \log_2(n+1) - 1 \approx \log_2(n+1) - 1$$

分析方法：对等概率情况，假设查找 n 次，且每个查找 1 次，共比较关键字 c 次，则平均 c/n 次。

例：表长为 $n=10$ ，平均查找长度如下。

$$ASL = \frac{3 + 2 + 3 + 4 + 1 + 3 + 4 + 2 + 3 + 4}{10} = \frac{29}{10} = 2.9$$

（3）时间复杂度：从平均查找长度看折半查找的时间复杂度是 $O(\log_2 n)$ 。

有时对需要反复查找的数据预先排序，再折半查找也是划算的。

【习题演练】

1. 对线性表进行折半查找时，要求线性表必须（ ）。

- A. 以顺序方式存储 B. 以顺序方式存储，且结点按关键字有序排列
- C. 以链式方式存储 D. 以链式方式存储，且结点按关键字有序排列
2. 采用折半查找法查找长度为 n 的线性表时，每个元素的平均查找长度为（ ）。
- A. $O(n^2)$ B. $O(n \log_2 n)$ C. $O(n)$ D. $O(\log_2 n)$

三、索引顺序表查找

1. 思路

将查找表分为若干块，块之间必须有序，块内可以是无序，将每一块中最大的关键字和第一个元素的地址放在索引表中，因而，索引表是有序的。查找时，先对索引表进行顺序或折半查找，继而确定出在哪一块中进行顺序查找。

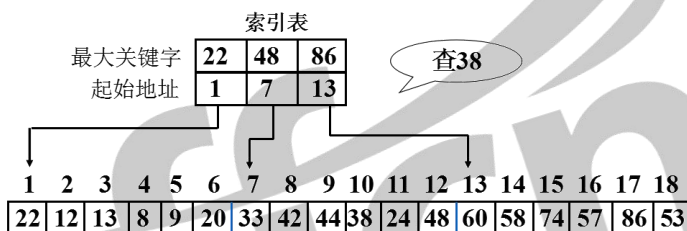


图 7-3 表及其索引表

2. 分析

索引顺序表的查找性能介于顺序查找与折半查找之间。吸取了顺序查找和折半查找各自的优点。

分块查找的**平均查找长度**=索引表中平均查找长度+块内查找的平均查找长度。

设表长为 n ，等分成 b 块，每块 s 个记录，在等概率的情况下，若在块内和索引表中均采用顺序查找，则平均查找长度为：

$$ASL = \frac{b+1}{2} + \frac{s+1}{2} = \frac{s^2 + 2s + n}{2s}$$

要使 ASL 最小，有 $s=b=\sqrt{n}$ 。

若索引表采用折半查找，块内采用顺序查找，则平均查找长度为：

$$ASL \approx \log_2(b+1) + \frac{s+1}{2}$$

【习题演练】

1. 如果要求一个线性表既能较快的查找，又能适应动态变化的要求，最好采用（ ）

查找法。

- A. 顺序查找
- B. 折半查找
- C. 分块查找
- D. 哈希查找

第二节 动态查找表

一、二叉排序树和平衡二叉树

1. 二叉排序树

二叉排序树或为空树；或者具有下列性质：若左子树不空，则左子树上所有结点值均小于根结点，若右子树不空，则右子树上所有结点值均大于根结点，其左、右子树也是二叉排序树。如果中序遍历二叉排序树，将得到递增的有序序列。

(1) 查找结点

思路：①若二叉树为空，则找不到；②若二叉树非空，先与根结点比较，相等则查找成功，若小于根则在左子树上继续查找，否则在右子树上继续查找。

递归算法：

```
BstTree BstSearchxe ( BstTree bst, DataType x )
```

```
{  
    if ( bst==NULL )  
        return NULL;  
    else if ( bst->data==x )  
        return bst;  
    else if ( x<bst->data )  
        return BstSearch ( bst->lchild, x );  
    else  
        return BstSearch ( bst->rchild, x );  
}
```

非递归算法：

```
BstTree BstSearch ( BstTree bst, DataType x )
```

```
{  
    p = bst;  
    while ( p ) {  
        if ( p->data==x ) return p;  
        else if ( x<p->data ) p = p->lchild;  
        else p = p->rchild;  
    }
```

```

    }
    return NULL; // not found
}

```

(2) 插入值

思路：先查找，若找不到则插入结点作为最后访问的叶子结点的孩子。新插入的结点总是叶子。

```

int BstInsert( BstTree &bst, DataType x)
{ //在二叉排序树 bst 中插入一个关键字为 x 的结点
    if (bst==NULL) { //原树为空，新插入的记录为根结点
        bst=(BstTree)malloc(sizeof(Bstnode));
        bst->key=x;
        bst->lchild=bst->rchild=NULL;
        return 1; //返回 1，表示成功
    }
    else if (x==bst->key) //树中存在相同关键字的结点
        return 0;
    else if(x<bst->key) //插入到的左子树中
        return BstInsert (bst->lchild,x);
    else //插入到的右子树中
        return BstInsert (bst->rchild,x);
}

```

(3) 建立二叉排序树

思路：给定关键字序列，建立二叉排序树。开始二叉树为空，第一个元素为二叉排序树的根结点，之后对读入的每一个元素，先进行查找，如果已存在，则不作任何处理，否则插入。经过一系列插入操作便建立了一棵二叉排序树。

一句话，“从空树开始，每次插入一个关键字”。

```

void CreatBst(BstTree &bst,DataType str[],int n)
{ //用关键字数组 str[]建立一个二叉排序树
    bst=NULL; //初始时树为空
    int i=0;
    while(i<n){ //依次将每个元素插入
        BstInsert (bst, str[i]);
        i++;
    }
}

```

【例 7-2】给定关键字序列{53, 45, 12, 24, 90, 45, 80}，建立二叉排序树。

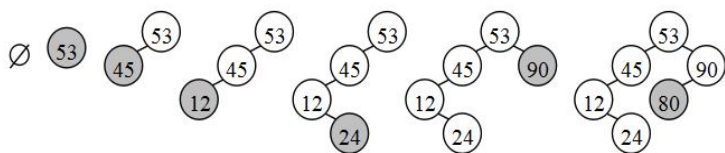


图 7-4 建立二叉排序树

(4) 删除结点

删除操作的实现有如下 3 种情况：

①如果删除的是**叶子**直接删除即可。

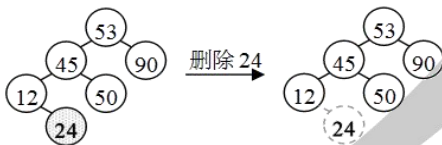


图 7-5 二叉排序树删除叶结点

②如果删除结点的**左子树或右子树为空**，“移花接木”：将左子树或右子树接到双亲上结点的位置上。

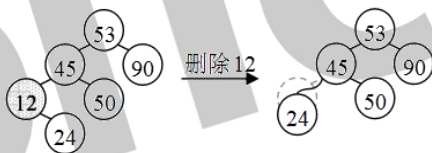


图 7-6 二叉排序树删除单分支结点

③如果删除结点的**左右子树都不空**，“偷梁换柱”：借左子树上最大的结点（直接前趋）替换被删除的结点，然后变为删除左子树最大结点的情况。（或者借用右子树上最小结点（直接后继）然后删除之）。

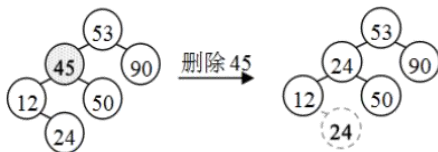


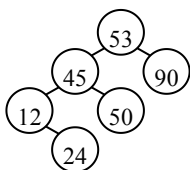
图 7-7 二叉排序树删除双分支结点

(5) ASL 分析

判定树和二叉排序树相同。结点所在的层次等于查找时比较的关键字个数。

若按照关键字有序的顺序插入结点建立二叉排序树，将得到一棵单支树，对其进行查找也退化为顺序查找，查找成功的平均查找长度为 $(1+n)/2$ 。一般情况下，它的查找过

程与二分查找类似，二叉排序树的查找的平均查找长度可达到 $O(\log_2 n)$ ，由此可知，二叉排序树的平均查找长度，主要取决于树的高度（树的形态）。



等概率情况下查找成功时

$$ASL = \frac{1}{n} \sum_{i=1}^n h_i = \frac{1+2+2+3+3+4}{6} = 2.5$$

【习题演练】

1. 对于下列关键字序列，不可能构成某二叉排序树中一条查找路径的序列是（ ）。

- A. 95, 22, 91, 24, 94, 71 B. 92, 20, 91, 34, 88, 35
C. 21, 89, 77, 29, 36, 38 D. 12, 25, 71, 68, 33, 34

2. 平衡二叉树

树上任一结点的左子树和右子树的高度差的绝对值不超过 1 的二叉树。结点左子树深度与结点右子树深度差是结点的平衡因子。平衡二叉树中各个结点的平衡因子只能是 0, 1, -1。

（1）构造平衡二叉排序树

思路：按照建立二叉排序树的方法逐个插入结点，失去平衡时作调整。

失去平衡时的调整方法：

① 确定三个代表性结点。（A 是插入路径上失去平衡的最小子树的根；B 是 A 的孩子；C 是 B 的孩子，也是新插入结点的子树）。关键是找到失去平衡的最小子树。

② 根据三个代表性结点的相对位置判断是哪种不平衡类型（LL, LR, RL, RR）。

③ 平衡化。“先摆好三个代表性结点（居中者为根），再接好其余子树（根据大小）”。

四种类型的平衡化过程：

LL 型：B 右上旋转到 A 的位置，A 变为 B 的右孩子，B 的原来的右子树变为 A 的左子树。

RR 型：B 左上旋转到 A 的位置，A 变为 B 的左孩子，B 的原来的左孩子变为 A 的右孩子。

LR 型：先将 C 左上旋转到 B 的位置，B 变为 C 的左孩子，C 原来的左孩子变为 B 的右孩子；再将 C 右上旋转到 A 的位置，A 变为 C 的右孩子，此时 C 的右孩子，将变为 A 的左孩子。

RL 型：先将 C 右上旋转到 B 的位置，B 变为 C 的右孩子，C 原来的右孩子变为 B 的左孩子；再将 C 左上旋转到 A 的位置，A 变为 C 的左孩子，此时 C 的左孩子，将变为 A 的右孩子。

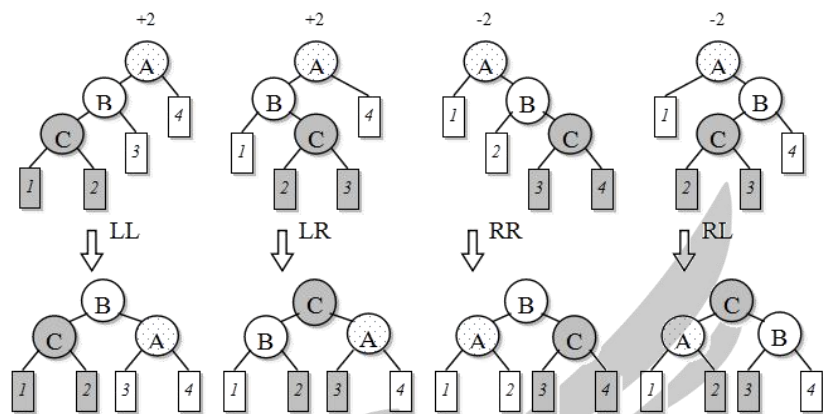


图 7-8 二叉树的平衡化

【例 7-3】给定关键字的序列{13, 24, 37, 90, 53, 40}，建立平衡二叉排序树。

注意：失去平衡时先确定失去平衡的最小子树，这是关键，然后判断类型（LL，LR，RL，RR），再平衡化处理。

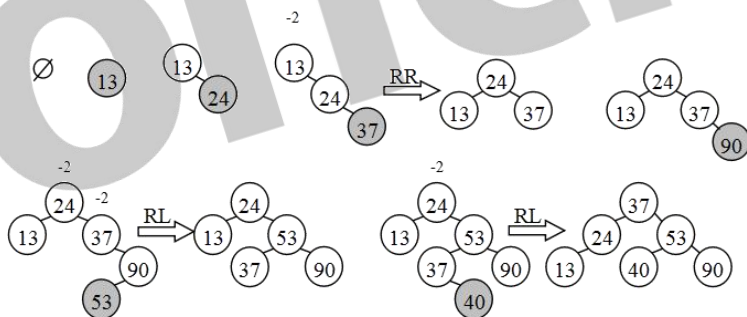


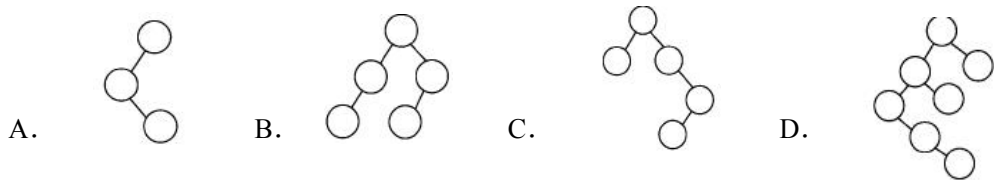
图 7-9 建立二叉平衡树

（2）平均查找长度 ASL（同二叉排序树）

在平衡二叉树排序树上进行查找的过程和二叉排序树相同，在查找过程中平均查找长度与树的高度有关。含有 n 个结点的平衡二叉树的最大高度为 $O(\log_2 n)$ ，因此，平衡二叉树排序树的平均查找长度为 $O(\log_2 n)$ 。

【习题演练】

1. 下列二叉排序树中，满足平衡二叉树定义的是（ ）。



二、B 树和 B+树

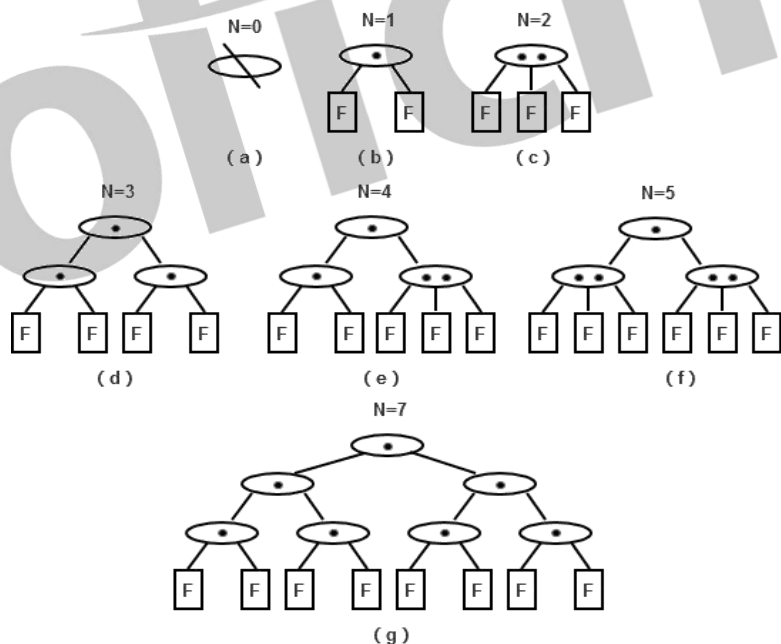
1. B 树

一棵 m 阶 B 树，或为空树，或满足：

- (1) 每个结点至多有 m 棵子树；
- (2) 若根结点不是叶子，则至少有两棵子树；
- (3) 除根之外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树；

(4) 所有非终端结点包含 n 个关键字和 $n+1$ 个指针： $(n, A_0, K_1, A_1, \dots, K_n, A_n)$ ，其中关键字满足 $K_1 < K_2 < \dots < K_n$ ，关键字的个数 $\lceil m/2 \rceil - 1 \leq n \leq m-1$ 。

(5) 所有叶子在同一层，不含信息，表示查找失败的结点。



(a)空树;(b) $N=1$;(c) $N=2$ (d) $N=3$;(e) $N=4$;(f) $N=5$;(g) $N=7$

图 7-10 不同关键字数目的 B 树（查找分析）

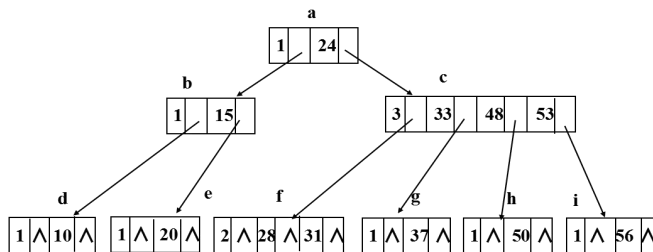


图 7-11 一棵 4 阶的 B 树（查找）

（1）B 树的查找

B 树的查找与二叉排序树的查找类似，只不过在 B 树中进行的是多路查找。B 树的查找包含两步，在 B 树中找结点和在结点内找关键字。在 B 树上查找到某个结点后，先在有序表中进行查找，若找到则查找成功，否则按照对应的指针信息到所指的子树中去查找（例如，在图 7-10 中查找到第一层的第一个结点时，若发现关键字大于或者小于 24，则在这个结点上查找失败，将根据相应的指针到结点的子树中继续查找）。当查找到叶结点时（对应的指针为空指针），则说明树中没有对应的关键字，查找失败。

在 B 树中的查找结点是在磁盘上进行的，在结点内找关键字是在内存中进行的，在磁盘上进行一次操作显然比在内存中查找一次耗时多，因此，在磁盘上进行查找的次数、即待查关键字所在结点在 B 树中所在的层次，是决定 B 树查找效率的首要因素。最坏情况下，含有 N 个关键字的 B 树的最大深度是 $l \leq \log_{\lceil m/2 \rceil} (\frac{N+1}{2}) + 1$ ，也就是说，在含有 N 个关键字的 B 树上进行查找，查找的结点树不会超过 $\log_{\lceil m/2 \rceil} (\frac{N+1}{2}) + 1$ 。

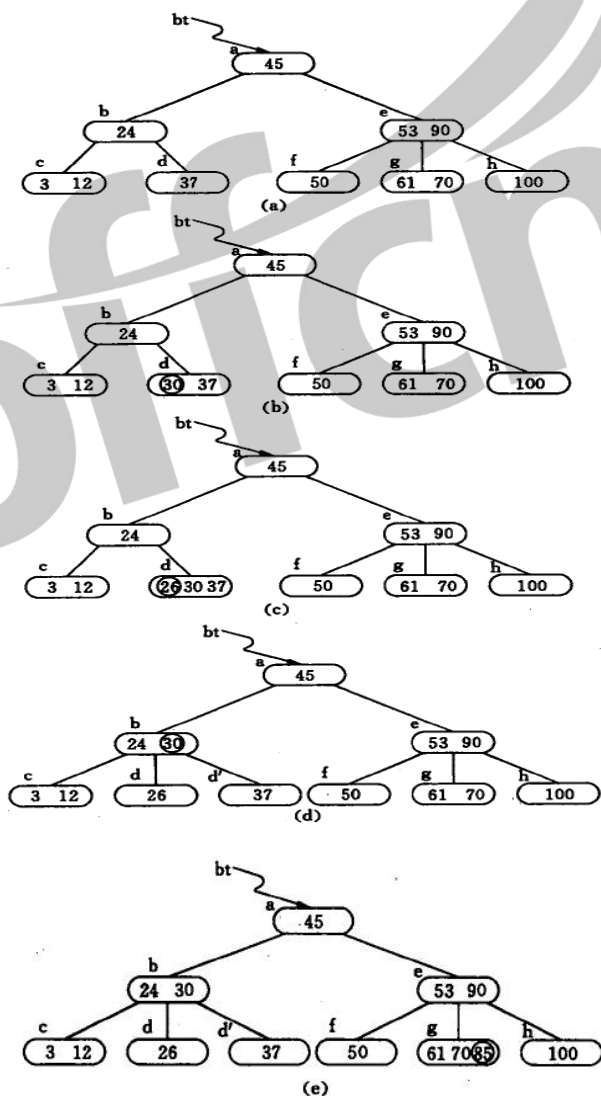
（2）B 树的插入

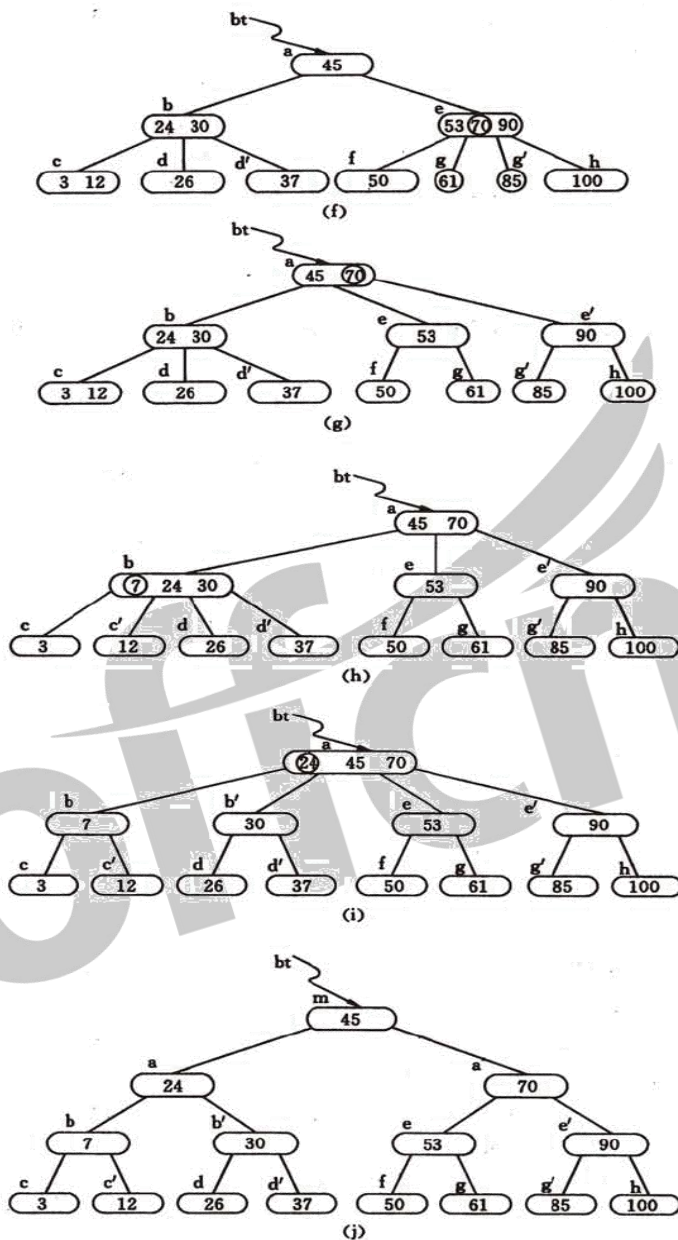
与二叉查找树的插入操作相比，B 树的插入操作要复杂得多。在二叉查找树中，仅需查找至需插入的终端结点的位置。但是，在 B 树中找到插入的位置后（最低层的非终端结点），并不能简单地将其添加到终端结点中去，因为插入操作可能会导致整棵树不再满足 B 树中定义中的要求。将关键字 key 插入到 B 过程如下：

①定位：利用 B 树查找算法，找出插入该关键字的最底层中某个非叶结点；

②插入：在 B 树中，每个非失败结点的关键字个数都在 $\lceil m/2 \rceil - 1, m - 1$ 之间。当播入后的结点关键字个数小于等于 $m - 1$ ，则可以直接插入；当插入后的结点关键字个数大于 $m - 1$ 时，则必须对结点进行分裂。

分裂的方法是：取一个新结点，将插入 key 后的原结点从中间位置将其中的关键字分为两部分，左部分包含的关键字放在原结点中，右部分包含的关键字放到新的结点中，中间位置 $\lceil m/2 \rceil$ 的结点插入到原结点的父结点中。若此时导致其父结点的关键字个数也超过了上限，则继续进行这种分裂操作，直至这个过程传到根结点为止，这样导致 B 树高度增 1。对于 $m=3$ 的 B 树，所有结点中最多有 $m-1=2$ 个关键字，在图 7-12 (b) 中插入 30 后，仍满足 B 树性质，直接插入即可；继续插入，当插入一个关键字 26 后，结点内的关键字个数超出了 $m-1$ ，如图 7-12 (c) 所示，此时必须进行结点分裂，分裂的结果如图 7-12 (d) 所示。





(a) 一棵 3 阶 B 树; (b) 插入 30 之后; (c)、(d) 插入 26 之后; (e)~(g) 插入 85 之后; (h)~(j) 插入 7 之后

图 7-12 在 B 树中进行插入 (省略叶子结点)

(3) B 树的删除

若在 B 树上删除一个关键字 key，则首先应找到该关键字所在结点，再从中删除。

若该结点为终端结点（不考虑最后一层叶子结点）：①结点中关键字数目大于 $\lceil m/2 \rceil - 1$ ，则直接删除；否则要进行“合并”结点的操作。②假若所删除关键字结点的左（右）兄弟结点中关键字个数大于 $\lceil m/2 \rceil - 1$ ，删除关键字后，将左（右）兄弟中关键字最大（小）

的移至双亲结点中，双亲结点中对应的关键字移动到被删除结点中；③假若所删除关键字结点的左（右）兄弟结点中关键字个数等于 $\lceil m/2 \rceil - 1$ ，删除关键字后，与左（右）兄弟结点及双亲结点中的关键字进行合并。

若该结点在非终端结点，①如果小于（或大于）key 的子树中关键字个数大于 $\lceil m/2 \rceil - 1$ ，则找出 key 的直接前驱（或直接后继）来代替 key，再递归的删除它的前驱（或后继）；②如果大于或小于 key 的子树中关键字个数等于 $\lceil m/2 \rceil - 1$ ，则直接删除 key，并将两个子树结点合并即可。

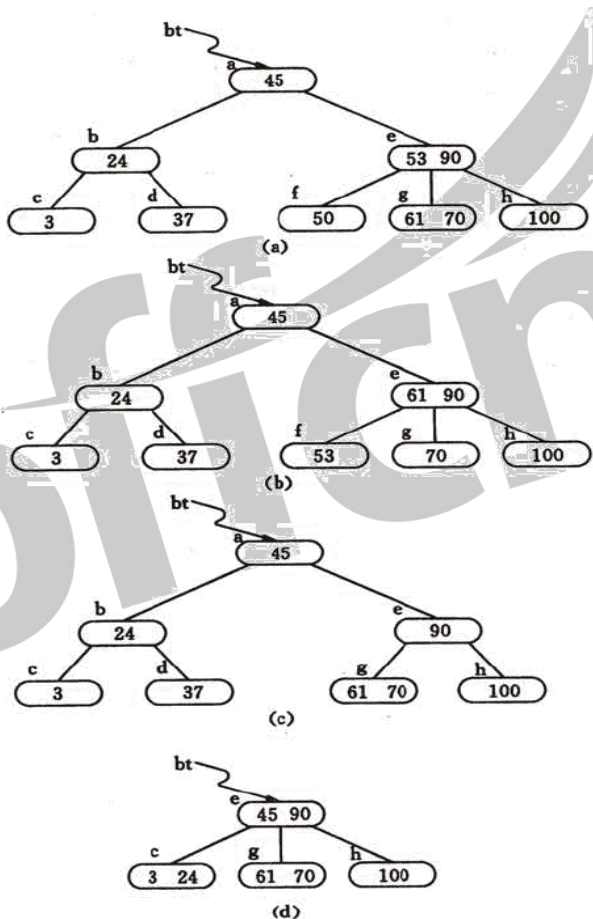


图 7-13 在 B 树中删除关键字的情形

【习题演练】

- 下列叙述中，不符合m阶B树定义要求的是（ ）。

A. 根结点最多有m棵子树
B. 所有叶结点都在同一层上

C. 各结点内关键字均升序或降序排序
D. 叶结点之间通过指针链接

2. B+树

(1) 每个分支结点最多有 m 棵子树，非叶根结点至少有两棵子树，其他每个分支至少有 $\lceil m/2 \rceil$ 棵子树。

(2) 有 n 棵子树的结点中含有 n 个关键字。

(3) 所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

(4) 所有的非终端结点可以看成是索引部分，结点中仅含有其子树（根结点）中的最大（或最小）关键字。

图 7-14 所示为一棵 3 阶的 B⁺树，通常在 B⁺树上有两个头指针，一个指向根结点，另一个指向关键字最小的叶子结点。因此，可以对 B⁺树进行两种查找运算：一种是从最小关键字起顺序查找，另一种是从根结点开始，进行随机查找。

在 B⁺树上进行随机查找、插入和删除的过程基本上与 B 树类似。只是在查找时，若非终端结点上的关键字等于给定值，并不终止，而是继续向下直到叶子结点。因此，在 B 树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。B⁺树查找的分析类似于 B 树。B⁺树的插入仅在叶子结点上进行，当结点中的关键字个数大于 m 时要分裂成两个结点，它们的双亲结点中应同时包含这两个结点中的最大关键字。B⁺树的删除也仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界关键字”存在。若因删除而使结点中关键字的个数少于 $\lceil m/2 \rceil$ 时，其与兄弟结点的合并过程亦和 B 树类似。

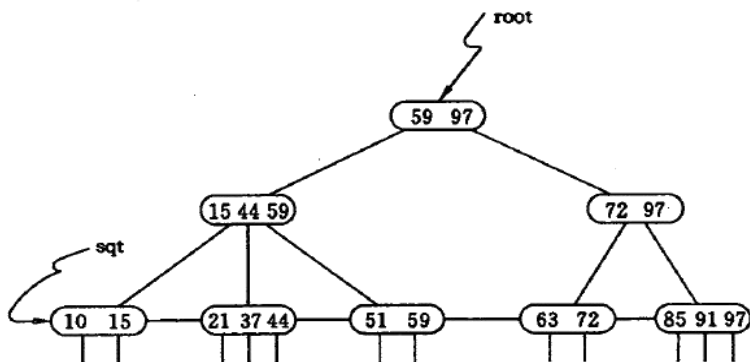


图 7-14 一棵 3 阶的 B⁺树

第三节 哈希表

一、哈希表（散列表，杂凑表）

第一、二节学习的查找方法都需要进行一系列关键字的比较，查找的效率依赖于查找过程中关键字比较的次数。理想情况是希望不经过任何比较，一次存取便可得到查找的记录。依照这个思想，设定哈希函数 $H(\text{key})$ 和处理冲突的方法，将一组关键字映射到一个有限的连续的地址集上，并以关键字在地址集中的象作为记录在表中的存储位置，这种表称为**哈希表**。

冲突： $H(\text{key}_1)=H(\text{key}_2)$ ，且 $\text{key}_1 \neq \text{key}_2$ ，称**冲突**。也就是说两个或两个以上的不同关键字映射到同一个地址，这些关键字称为同义词。散列函数应尽量减少这样的冲突，但有的时候冲突是不可避免的，所以还要设计好处理冲突的方法。

二、哈希函数

1. 除留余数法：

$H(\text{key}) = \text{key} \text{ MOD } p$ ， p 为接近但不大于表长 m 的质数。 p 越接近 m 产生冲突的可能性越小。

2. 直接定址法：

$H(\text{key})=a \times \text{key}+b$ ，取关键字或关键字的某个线性函数值为哈希地址。由该方法所得地址集合和关键字集合的大小相同。因此，对于不同的关键字不会发生冲突。但实际中能使用的情况很少。

3. 数字分析法：

假设关键字是 r 进制，并且哈希表中可能出现的关键字是已知的，则可取关键字中 r 个数码出现频率均匀的若干位组成哈希地址，或者取其中几位的叠加和舍去进位作为哈希地址。

4. 平方取中法：

取关键字平方后的中间几位为哈希地址，具体取几位要视表的长度等情况而定。适用于关键字的每一位取值都不够均匀或均小于散列地址所需要的位数。

5. 折叠法：

将关键字分割成位数相同的几部分，然后取这几部分的叠加和（舍去进位）作为哈希地址。适用于关键字位数很多，而且关键字中每一位上数字分布大致均匀时。

6. 随机数法：

选择一个随机函数，取关键字的随机函数值为它的哈希地址，即

$H(\text{key})=\text{random}(\text{key})$ 。适用于关键字长度不等时的情况。

三、解决冲突的方法

选择合适的哈希函数可以减少冲突，但是不能完全避免，因而，处理冲突是哈希表构造不可缺少的另一方面。处理冲突就是为产生冲突的关键字的记录找到另一个空的哈希地址。通常用的处理冲突的方法有以下几种：

1. 开放定址法

$H_i = (H(\text{key}) + d_i) \text{MOD } m \quad (i = 1, 2, \dots, k, (k \leq m-1))$ ，其中 $H(\text{key})$ 是哈希函数， m 为哈希表长， d_i 为增量序列，增量序列常见以下三种：

(1) 线性探测再散列： $d_i = 1, 2, 3, \dots, m-1$ 。该方法会造成大量元素在相邻的散列地址上“堆积”，降低了查找效率。

(2) 平方探测再散列： $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \dots, \pm k^2, k \leq m/2$ 。散列表长度 m 必须是一个可以表示成 $4j+3$ 的素数。该方法可以避免出现“堆积”现象，但是不能探测到表中所有元素。

(3) 伪随机探测再散列： d_i = 伪随机数序列。

(4) 双重散列法： $d_i = iHash_2(\text{key})$ ， i 是产生冲突的次数。

2. 再哈希法： $H_i = Hash_i(\text{key}), i = 1, 2, \dots, k$ ， $Hash_i(\text{key})$ 均是不同的哈希函数，这种方法不易产生“聚集”，但增加了计算的时间。

3. 链地址法：将所有关键字为同义词的记录链成单链表，该链表由散列地址唯一标识，散列地址为 i 的同义词链表的头指针放在散列表的第 i 个单元中。

4. 建立公共溢出区：包含两部分，一为基本表 $HashTable[0..m-1]$ ，每个分量存放一个记录，二为溢出表 $OverTable[0..v]$ ，所有冲突记录装入溢出表。

四、哈希表的查找及其分析

在哈希表上进行查找的过程和哈希造表的过程基本一致。给定 Key 值，根据造表时设定的哈希函数求得哈希地址，若表中此位置上没有记录，则查找不成功；否则比较关键字，若和给定值相等，则查找成功；否则根据造表时设定的处理冲突的方法找“下一地址”，直至哈希表中某个位置为“空”或者表中所填记录的关键字等于给定值时为止。

从哈希表的查找过程可见，查找过程中需和给定值进行比较的关键字的个数取决于下列三个因素：哈希函数、处理冲突的方法和哈希表的装填因子。

装填因子：表示哈希表的装满程度

$$\alpha = \frac{n}{m}$$

n 是表中记录数, m 是散列表长度。

哈希表的平均查找长度依赖于装填因子, 与 n 、 m 不直接相关。 α 越大, 产生冲突的可能性就越大, 查找时, 进行关键字比较的个数也就越多; α 越小, 反之。

【例 8-3】 已知一组关键字 {19, 14, 23, 1, 68, 20, 85, 9}, 采用哈希函数 $H(\text{key}) = \text{key} \text{ MOD } 11$, 请分别采用以下处理冲突的方法构造哈希表, 并求各自的平均查找长度。

- 1) 采用线性探测再散列;
- 2) 采用伪随机探测再散列, 伪随机函数为 $f(n) = -n$;
- 3) 采用链地址法。

思路: 开始时表为空, 依次插入关键字建立哈希表。

1) 线性探测再散列

0	1	2	3	4	5	6	7	8	9	10
9	23	1	14	68				19	20	85
3	1	2	1	3				1	1	3

key	19	14	23	1	68	20	85	9	← 关键字
H(key)	8	3	1	1	2	9	8	9	← H(key)
					2	3		9	10 ← 冲突时计算下一地址
						4		10	0 ←
查找长度	1	1	1	2	3	1	3	3	← 每个关键字的查找长度

$$ASL = (1+1+1+2+3+1+3+3)/8 = 15/8$$

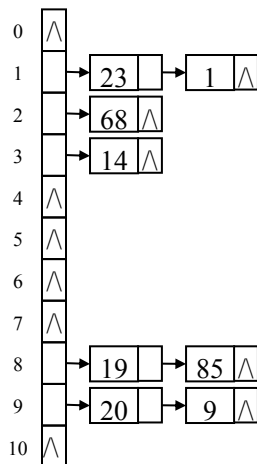
2) 伪随机探测再散列

0	1	2	3	4	5	6	7	8	9	10
1	23	68	14			9	85	19	20	
2	1	1	1			4	2	1	1	

Key	19	14	23	1	68	20	85	9
H(key)	8	3	1	1	2	9	8	9
					0		7	8
								7
								6
查找长度	1	1	1	2	1	1	2	4

$$ASL = (1+1+1+2+1+1+2+4)/8 = 13/8$$

3) 链地址法



$$ASL = (1 \times 5 + 2 \times 3) / 8 = 11/8$$

注：关键字在链表中的插入位置可以在表头或表尾，也可以在中间以便保持关键字有序。

最后，此哈希表的装填因子是 $\alpha = 8/11$ 。

【习题演练】

- 为提高散列（Hash）表的查找效率，可以采取的正确措施是（ ）。
 - 增大装填（载）因子
 - 设计冲突（碰撞）少的散列函数
 - 处理冲突（碰撞）时避免产生聚集（堆积）现象

A. 仅I B. 仅II C. 仅I、II D. 仅II、III
- 设哈希表长为 14，哈希函数是 $H(\text{key}) = \text{key} \% 11$ ，表中已有数据的关键字为 15，38，61，84 共四个，现要将关键字为 49 的元素加到表中，用二次探测法解决冲突，则放入的位置是（ ）。

A. 8 B. 3 C. 5 D. 9

第八章 内部排序

【本章综述】

堆排序（建堆、插入和调整）、快速排序（划分、过程特征）和归并排序（归并路数、归并过程）是本章重点。应深入掌握各种排序算法的思想、排序过程（能动手模拟）和特征（初态的影响、时空复杂度、稳定性、适用性等），通常以选择题的形式考查不同算法之间的对比。此外，对于一些常用排序算法的关键代码，要达到熟练编写的程度；看到某特定序列，应具有选择最优排序算法（根据排序算法特征）的能力。

【复习重点】

（一）插入排序

1. 直接插入排序
2. 折半插入排序
3. 希尔排序

（二）交换排序

1. 冒泡排序
2. 快速排序

（三）选择排序

1. 简单选择排序
2. 堆排序

（四）其他排序

1. 归并排序
2. 基数排序

（五）各种排序方法比较

第一节 插入排序

一、直接插入排序

直接插入排序是一种最简单的排序方法，它的基本操作是将一个记录插入到已排好

序的有序表中，从而得到一个新的、记录数增1的有序表。

1. 算法

```
void InsertSort(Elemtype A[],int n) {
    int i,j;
    for(i=2; i<=n; i++) //依次将 A[2]~A[n]插入到前面已排序序列
        if (A[i].key<A[i-1].key) { //若 A[i]的关键字小于其前驱，需将 A[i]插入有序表
            A[0]=A[i];           //复制哨兵，A[0]不存放元素
            for (j=i-1; A[0].key<A[j].key; --j) //从后往前查找待插入位置
                A[j+1]=A[j];           //向后挪位
            A[j+1]=A[0];           //复制到插入位置
        }
}
```

【例 8-1】 52, 49, 80, 36, 14, 58, 61 进行直接插入排序。

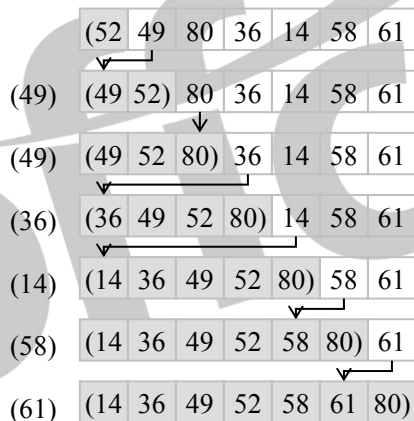


图 8-1 直接插入排序

2. 性能分析

(1) 空间效率：仅使用了常数个辅助单元，因而空间复杂度为 $O(1)$ 。

(2) 时间效率：在排序过程中，向有序子表中逐个地插入元素的操作进行了 $n-1$ 趟，每趟操作都分为比较关键字和移动元素，而比较次数和移动次数取决于待排序表的初始状态。

在最好情况下，表中元素已经有序，此时每插入一个元素，都只需比较一次而不用移动元素，因而时间复杂度为 $O(n)$ 。

在最坏情况下，表中元素顺序刚好与排序结果中元素顺序相反（逆序）时，总的比较次数达到最大：

$$\sum_{i=2}^n i = (2+n) \times (n-1) / 2$$

总的移动次数也达到最大：

$$\sum_{i=2}^n (i+1) = (3+n+1) \times (n-1) / 2$$

平均情况下，考虑待排序表中元素是随机的，此时可以取上述最好与最坏情况的平均值作为平均情况下的时间复杂度，总的比较次数与总的移动次数均约为 $n^2/4$ 。由此，直接插入排序算法的时间复杂度为 $O(n^2)$ 。

(3) 稳定性：由于每次插入元素时总是从后向前先比较再移动，所以不会出现相同元素相对位置发生变化的情况，即直接插入排序是一个稳定的排序方法。

(4) 适用性：直接插入排序算法适用于顺序存储和链式存储的线性表。当为链式存储时，可以从前往后查找指定元素的位置。注意：大部分排序算法都仅适用于顺序存储的线性表。

【习题演练】

1. 从未排序序列中依次取出元素与已排序序列中的元素进行比较，将其放入已排序序列的正确位置上的方法，这种排序方法称为（ ）。

- A. 归并排序 B. 冒泡排序 C. 插入排序 D. 选择排序

2. 对有 n 个元素的顺序表采用直接插入排序，在最坏情况下所需的比较次数是（ ）。

- A. $n-1$ B. $n+1$ C. $n/2$ D. $n(n-1)/2$

二、折半插入排序

由于插入排序的基本操作是在一个有序表中进行查找和插入，这个“查找”操作可利用“折半查找”来实现，由此进行的插入排序称之为折半插入排序。该方法先利用折半查找找到插入位置，然后统一移动插入位置之后的元素。

1. 算法

```
void InsertSort(ElemType A[],int n) {
    int i, j, low, high, mid;
    for(i=2; i<=n; i++) {    //依次将 A[2]~A[n]插入到前面已排序序列
        A[0]=A[i];          //将 A[i]暂存到 A[0]
        low=1; high=i-1;    //设置折半查找的范围
        while(low<=high) {  //折半查找
```

```
mid=(low+high)/2; //取中间点
if(A[mid].key>A[0].key)
    high=mid-1;    //查找左半子表
else low=mid+1;    //查找右半子表
}
for(j=i-1;j>=high+1; --j)
    A[j+1]=A[j];    //统一后移元素，空出插入位置
A[high+1]=A[0];    //插入操作
}
}
```

2. 性能分析

折半插入排序是稳定的排序算法。它所需附加存储空间和直接插入排序相同，从时间上比较，折半插入排序仅减少了关键字间的比较次数，约为 $O(n\log n)$ 而记录的移动次数不变。因此，折半插入排序的时间复杂度仍为 $O(n^2)$ 。

【习题演练】

1. 折半插入排序算法时间复杂度为 ()。

- A. $O(n)$ B. $O(n\log_2 n)$ C. $O(n^2)$ D. $O(n^3)$

三、希尔排序

希尔排序又称“缩小增量排序”，在时间效率上较前述几种排序方法有较大的改进。先将整个待排记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。

步骤：

- (1) 分成子序列（按照增量 d_k ）；
- (2) 对子序列排序（直接插入排序）；
- (3) 缩小增量，重复以上步骤，直到增量 $d_k=1$ 。

1. 算法

```
void ShellSort (ElemType A[],int n){
    //A[0]只是暂存单元，不是哨兵，当 j<=0 时，插入位置已到
    for(dk=n/2; dk>=1; dk=dk/2)    //步长变化
        for(i=dk+1; i<=n; ++i)
```

```

if(A[i].key<A[i-dk].key){           //需将 A[i]插入有序增量子表
    A[0]=A[i];                       //暂存在 A[0]
    for(j=i-dk; j>0&&A[0].key<A[j].key; j-=dk)
        A[j+dk]=A[j];               //记录后移，查找插入的位置
    A[j+dk]=A[0];                   //插入
} //if
}
    
```

【例 8-2】 希尔排序(52, 49, 80, 36, 14, 58, 61)。

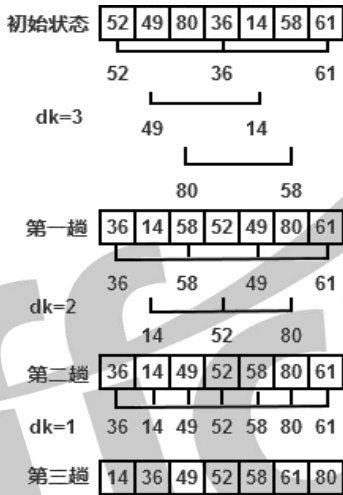


图 8-2 希尔排序

2. 性能分析

(1) 空间效率：仅使用了常数个辅助单元，因而空间复杂度为 $O(1)$ 。

(2) 时间效率：由于希尔排序的时间复杂度依赖于增量序列的函数，这涉及数学上尚未解决的难题，所以其时间复杂度分析比较困难。当 n 在某个特定范围时，希尔排序的时间复杂度约为 $O(n^{1.3})$ ，在最坏情况下希尔排序的时间复杂度为 $O(n^2)$ 。

(3) 稳定性：当相同关键字的记录被划分到不同的子表时，可能会改变它们之间的相对次序，因此，希尔排序是一个不稳定的排序方法。例如，表 $L = \{3, \underline{2}, 2\}$ ，经过一趟排序后， $L = \{2, \underline{2}, 3\}$ ，最终排序序列也是 $L = \{2, \underline{2}, 3\}$ ，显然， 2 与 $\underline{2}$ 的相对次序已经发生了变化

(4) 适用性：希尔排序算法仅适用于当线性表为顺序存储的情况。

【习题演练】

1. 对序列{15,9,7,8,20, -1,4}用希尔排序方法排序，经一趟后序列变为{15, -1,4,8,

20,9,7}则该次采用的增量是（ ）。

A. 1

B. 4

C. 3

D. 2

第二节 交换排序

一、冒泡排序

从前往后（或从后往前）依次比较相邻的两个记录的关键字，若两个记录是逆序的（即前一个记录的关键字大于后一个记录的关键字），则进行交换，直到序列比较完。也就是，首先将 $L \rightarrow A[1]$ 与 $L \rightarrow A[2]$ 的关键字进行比较，若为反序，则交换两个记录；然后比较 $L \rightarrow A[2]$ 与 $L \rightarrow A[3]$ 的关键字，依此类推，直到 $L \rightarrow A[n-1]$ 与 $L \rightarrow A[n]$ 的关键字比较后为止，称为一趟冒泡排序， $L \rightarrow A[n]$ 为关键字最大的记录。然后进行第二趟冒泡排序，对前 $n-1$ 个记录进行同样的操作。若待排序的记录有 n 个，则要经过 $n-1$ 趟冒泡排序才能使所有的记录有序。

1. 算法

```
void BubbleSort(ElemType A[],int n){
    //用冒泡排序法将序列 A 中的元素按从小到大排列
    for(i=0;i<n-1;i++){
        flag=false;           //表示本趟冒泡是否发生交换的标志
        for(j=n-1;j>i;j--)    //一趟冒泡过程
            if(A[j-1].key>A[j].key){ //若为逆序
                swap(A[j-1],A[j]);    //交换
                flag=true;
            }
        if(flag==false)
            return ;           //本趟遍历后没有发生交换，说明表已经有序
    }
}
```

2. 性能分析

（1）空间效率：仅使用了常数个辅助单元，因而空间复杂度为 $O(1)$ 。

（2）时间效率：当初始序列有序时，显然第一趟冒泡后 `flag` 依然为 `false`（本趟冒泡没有元素交换），从而直接跳出循环，比较次数为 $n-1$ ，移动次数为 0，从而最好情况下的时间复杂度为 $O(n)$ ；当初始序列为逆序时，需要进行 $n-1$ 趟排序，第 i 趟排序要进

行 $n-i$ 次关键字的比较，而且每次比较都必须移动元素 3 次来交换元素位置。这种情况

下，比较次数 = $\sum_{i=1}^{n-1} n-i = n(n-1)/2$ ，移动次数 = $\sum_{i=1}^{n-1} 3(n-i) = 3n(n-1)/2$ 。从而，最

坏情况下时间复杂度为 $O(n^2)$ ，其平均时间复杂度也为 $O(n^2)$ 。

(3) 稳定性：由于当 $i > j$ 且 $A[i].key = A[j].key$ 时，不会交换两个元素，从而冒泡排序是一个稳定的排序方法。并且每趟排序都会将一个元素放置到其最终的位置上。

【例 8-3】冒泡排序 (49, 38, 65, 97, 76, 13, 27, 49)。

49	38	38	38	38	13	13
38	49	49	49	13	27	27
65	65	65	13	27	38	38
97	76	13	27	49	49	
76	13	27	49	49		
13	27	49	65			
27	49	76				
49	97					
初始关键字	第一趟排序后	第二趟排序后	第三趟排序后	第四趟排序后	第五趟排序后	第六趟排序后

图 8-3 起泡排序

【习题演练】

1. 对 n 个不同的关键字由小到大进行冒泡排序，在下列 () 情况下比较的次数最多。

- A. 从小到大排列好的
- B. 从大到小排列好的
- C. 元素无序
- D. 元素基本有序

二、快速排序

快速排序是对起泡排序的一种改进。它的基本思想是，通过一趟排序将待排记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

1. 算法

```
void QuickSort (ElemType a[], int low, int high )
{
    if ( low < high ) { // 划分
        pivot = a[low];
        i = low; j = high;
        while ( i < j ) {
            while ( i < j && a[j] >= pivot ) j--;
            a[i] = a[j];
            while ( i < j && a[i] <= pivot ) i++;
            a[j] = a[i];
        }
        a[i] = pivot;
        // 对子序列快排
        QuickSort ( a, low, i-1);
        QuickSort ( a, i+1, high);
    }
}
```

2. 性能分析

(1) 空间效率：由于快速排序是递归的，需要借助一个递归工作栈来保存每一层递归调用的必要信息，其容量应与递归调用的最大深度一致，最好情况下为 $\lceil \log_2(n+1) \rceil$ ；最坏情况下深度为 $O(n)$ ；平均情况下，栈的深度为 $O(\log_2 n)$ 。因而空间复杂度在最坏情况下为 $O(n)$ ，平均情况下为 $O(\log_2 n)$ 。

(2) 时间效率：快速排序的运行时间与划分是否对称有关，在理想情况下，每一趟快速排序都分为两个大小均衡的序列，时间复杂度为 $O(n \log_2 n)$ 。记录本来有序时为最坏情况，时间复杂度为 $O(n^2)$ 。快速排序平均情况下运行时间与其最佳情况下的运行时间很接近，而不是接近其最坏情况下的运行时间。快速排序是所有内部排序算法中平均性能最优的排序算法。

(3) 稳定性：在划分算法中，若右端区间存在两个关键字相同，且均小于基准值的记录，则在交换到左端区间后，它们的相对位置会发生变化，即快速排序是一个不稳定的排序方法。例如，表 $L=\{3, \underline{2}, 2\}$ ，经过一趟排序后， $L=\{2, \underline{2}, 3\}$ ，最终排序序列也是 $L=\{2, \underline{2}, 3\}$ 。

【例 8-4】 {52 49 80 36 14 58 61} 快速排序。

选第 1 个记录为轴，分别从后向前，从前向后扫描记录，后面“抓大放小”（如：①②），前面“抓小放大”（如：③④），交替进行（⑤-⑦），最后将轴记录放在中间（⑧），划分成两个序列。下面是一次划分的详细步骤：

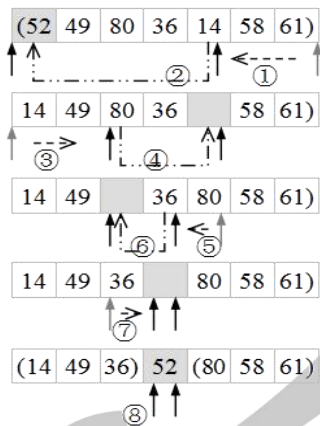


图 8-4 快速排序示例

整个快速排序过程如下：

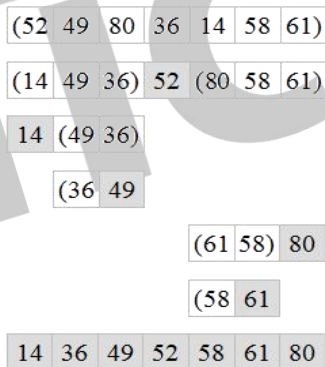


图 8-5 快速排序

【习题演练】

1. 采用递归方式对顺序表进行快速排序。下列关于递归次数的叙述中，正确的是（ ）。
- A. 递归次数与初始数据的排列次序无关
 - B. 每次划分后，先处理较长的分区可以减少递归次数
 - C. 每次划分后，先处理较短的分区可以减少递归次数
 - D. 递归次数与每次划分后得到的分区处理顺序无关

2. 若一组记录的排序码为(46, 79, 56, 38, 40, 84), 则利用快速排序的方法, 以第一个记录为基准得到的一次划分结果为()。

A. 38, 40, 46, 56, 79, 84

B. 40, 38, 46, 79, 56, 84

C. 40, 38, 46, 56, 79, 84

D. 40, 38, 46, 84, 56, 79

第三节 选择排序

选择排序的基本思想是: 每一趟在 $n-i+1$ ($i=1, 2, \dots, n-1$) 个记录中选取关键字最小的记录作为有序序列中第 i 个记录。其中最简单且为读者最熟悉的是**简单选择排序**。

一、简单选择排序

一趟简单选择排序的操作为: 通过 $n-i$ 次关键字间的比较, 从 $n-i+1$ 个记录中选出关键字最小的记录, 并和第 i ($1 \leq i \leq n$) 个记录交换之。

1. 算法

```
void SelectionSort (ElemType a[], int n)
{
    for ( i=0; i<n-1; i++) {           //一共进行 n-1 趟
        min = i;                       //记录最小元素位置
        for ( j=i+1; j<n; j++)         //在 a[i...n-1]中选择最小的元素
            if ( a[j]<a[min] )
                min=j;                 // 最小记录
        if (min!=i )
            swap(a[i], a[min]); //与第 i 个位置交换
    }
}
```

2. 性能分析

(1) 空间效率: 仅使用常数个辅助单元, 故空间效率为 $O(1)$ 。

(2) 时间效率: 从上述伪码中不难看出, 简单选择排序过程中, 元素移动的操作次数很少, 不会超过 $3(n-1)$ 次, 最好的情况是移动 0 次, 此时对应的表已经有序; 但元素间比较的次数与序列的初始状态无关, 始终是 $n(n-1)/2$ 次, 所以时间复杂度始终是 $O(n^2)$ 。

(3) 稳定性: 在第 i 趟找到最小元素后, 和第 i 个元素交换, 可能会导致第 i 个元素与其含有相同关键字元素的相对位置发生改变。例如, 表 $L=\{2, \underline{2}, 1\}$, 经过一趟排

序后, $L=\{1, \underline{2}, 2\}$, 最终排序序列也是 $L=\{1, \underline{2}, 2\}$ 。因此, 简单选择排序是一个不稳定的排序方法。

【例 8-5】 {52 49 80 36 14 58 61} 简单选择排序。

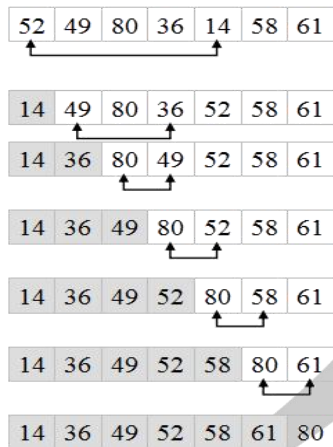


图 8-6 简单选择排序

二、堆排序

首先将待排序列建立初始堆, 输出堆顶元素 (第一个元素) 之后, 将堆底送入堆顶, 将剩余 $n-1$ 个元素又建成一个堆, 得到 n 个元素的次小值。如此反复执行, 便得到一个有序序列。

实现堆排序需要解决两个问题: 如何由一个无序序列建成一个堆; 如何在输出堆顶元素之后, 调整剩余元素为一个新堆。在排序过程中, 将 $L[1..n]$ 看成是一棵完全二叉树的顺序存储结构, 利用完全二叉树双亲和孩子结点之间的关系, 进行筛选调整。

1. 堆定义及其特点

堆的定义如下: n 个元素的序列 $\{k_1, k_2, \dots, k_n\}$ 当且仅当满足下关系时, 称之为堆。

序列 $\{K_1, K_2, \dots, K_n\}$ 满足 $K_i \leq K_{2i}$, $K_i \leq K_{2i+1}$, 称为小顶堆; 若满足 $K_i \geq K_{2i}$, $K_i \geq K_{2i+1}$, 称为大顶堆, 其中 $i=1, 2, \dots, \lfloor n/2 \rfloor$ 。

特点: 小顶堆的堆顶为最小元素, 大顶堆的堆顶为最大元素。

【例 8-6】 判断序列(12,36,24,85,47,30,53,91)是否构成堆。

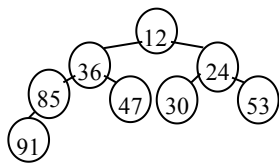


图 8-7 小顶堆

根据上图判断，该序列构成小顶堆。

2. 建立堆

堆排序的关键是构造初始堆，对初始序列建堆，就是一个反复筛选的过程。首先对第 $\lfloor n/2 \rfloor$ 个结点为根的子树筛选（对于大根堆：若根结点的关键字小于左右子女中关键字较大者，则交换），使该子树成为堆。之后依次对各结点 $(\lfloor n/2 \rfloor - 1 \sim 1)$ 为根的子树进行筛选，看该结点值是否大于其左右子结点的值，若不是，将左右子结点中较大值与之交换，交换后可能会破坏下一级的堆，于是继续采用上述方法构造下一级的堆，直到以该结点为根的子树构成堆为止。反复利用上述调整堆的方法建堆，直到根结点。图 9-8 所示，为小根堆的调整过程。

【例 8-7】把(24,85,47,53,30,91,12,36)调整成小顶堆。

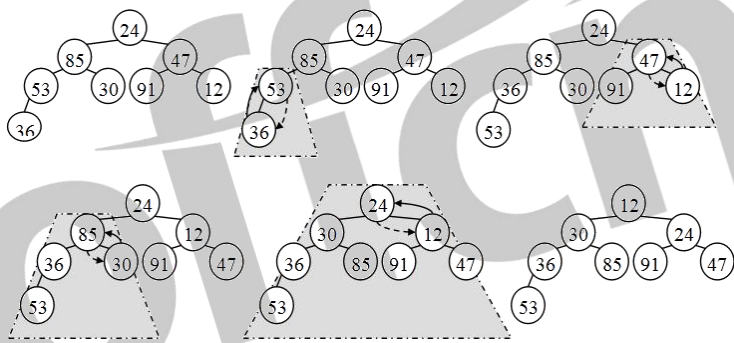


图 8-8 建立小顶堆

建立大根堆的算法：

```
void BuildMaxHeap(ElemType A[],int len){
    for(int i=len/2; i>0; i--)    //从 i=[n/2]~1，反复调整堆
        AdjustDown(A, i, len);
}

void AdjustDown(Elemtype A[], int k, int len) {
    //函数 AdjustDown 将元素 k 向下进行调整
    A[0]=A[k];                    //A[0]暂存
    for(i=2*k; i<=len; i*=2) {    //沿 key 较大的子结点向下筛选
        if(i<len&&A[i]<A[i+1])
            i++;                  //取 key 较大的子结点的下标
        if(A[0]>=A[i])    break;  //筛选结束
    }
    A[k]=A[i];                    //将 A[i]调整到双亲结点上
}
```

```

        k=i;                                //修改 k 值，以便继续向下筛选
    }
}
A[k]=A[0];                                //被筛选结点的值放入最终位置
}

```

3. 堆排序算法

```

void HeapSort(ElemType A[], int len){
    BuildMaxHeap(A, len);    //初始建堆
    for(i=len; i>1; i--){    //n-1 趟的交换和建堆过程
        Swap(A[i], A[1]);    //输出堆顶元素（和堆底元素交换）
        AdjustDown(A, 1, i-1); //整理，把剩余的 i-1 个元素整理成堆
    }
}

```

【例 8-8】(24,85,47,53,30,91,12,36)进行堆排序。

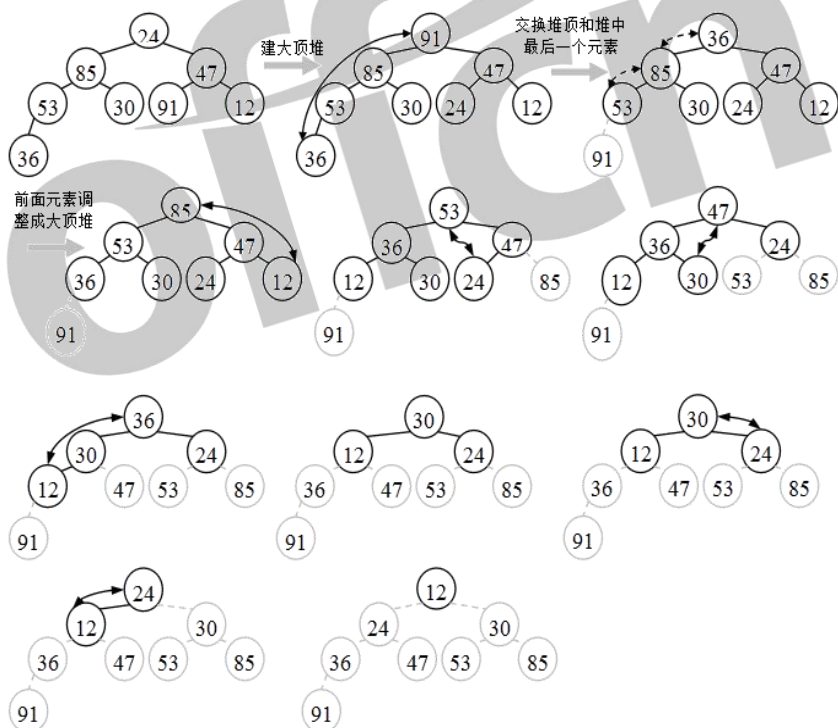


图 8-9 堆排序

整个排序步骤如下：

24	85	47	53	30	91	12	36
91	85	47	53	30	24	12	36
85	53	47	36	30	24	12	91
53	36	47	12	30	24	85	91
47	36	24	12	30	53	85	91
36	30	24	12	47	53	85	91
30	12	24	36	47	53	85	91
24	12	30	36	47	53	85	91
12	24	30	36	47	53	85	91

图 8-10 堆排序

4. 性能分析

(1) 空间效率：堆排序只需要一个记录大小的辅助空间，所以空间复杂度为 $O(1)$ 。

(2) 时间效率：建堆时间为 $O(n)$ ，之后有 $n-1$ 次向下调整操作，每次调整的时间复杂度为 $O(h)$ ，故在最好、最坏和平均情况下，堆排序的时间复杂度为 $O(n\log_2 n)$ 。

(3) 稳定性：在进行筛选时，有可能把后面相同关键字的元素调整到前面，所以堆排序算法是一种不稳定的排序方法。例如，表 $L=\{1, \underline{2}, 2\}$ ，构造初始堆时，可能将 2 交换到堆顶，此时 $L\{\underline{2}, 1, 2\}$ ，最终排序序列为 $L=\{1, 2, \underline{2}\}$ ，显然，2 与 2 的相对次序已经发生了变化。

【习题演练】

1. 堆的形状是一棵（ ）。

A. 二叉排序树 B. 满二叉树 C. 完全二叉树 D. 平衡二叉树

2. 若一组记录的排序码为 (46, 79, 56, 38, 40, 84)，则利用堆排序的方法建立的初始堆为（ ）。

A. 79, 46, 56, 38, 40, 84 B. 84, 79, 56, 38, 40, 46
C. 84, 79, 56, 46, 40, 38 D. 84, 56, 79, 40, 46, 38

3. 设待排序的关键字序列为 {12, 2, 16, 30, 28, 10, 16*, 20, 6, 18}，试写出简单选择排序，每趟排序结束后关键字序列的状态。

第四节 其他排序

一、归并排序

归并排序与上述基于交换、选择等排序的思想不一样，“归并”的含义是将两个或两个以上的有序表组合成一个新的有序表。假定待排序表含有 n 个记录，则看成是 n 个有序的子表，每个子表长度为 1，然后两两归并，再两两归并，……如重复，直到合并成一个长度为 n 的有序表为止，这种排序方法称为 2-路归并排序。

1. 算法

```
void MergeSort ( ElemType a[], int low, int high )
{
    if ( low >= high ) return;
    else {
        mid = (low+high)/2;           //从中间划分两个子序列
        MergeSort ( a, low, mid );    //对左侧子序列进行递归排序
        MergeSort ( a, mid+1, high ); //对右侧子序列进行递归排序
        Merge ( a, low, mid, high );  //归并
    }
}

void Merge (ElemType a[], int low, int mid, int high )
{//将有序序列 a[low..mid]和 a[mid+1..high]归并到
    for(int k=low; k<=high; k++)
        b[k]=a[k];                //将 a 中所有元素复制到 b 中
    for(i=low, j=mid+1, k=i; i<=mid&&j<=high; k++){
        if (b[i]<=b[j])             //比较 b 的左右两端中的元素
            a[k]=b[i++];           //将较小值复制到 a 中
        else
            a[k]=b[j++];
    }
    // 归并剩余元素
    while ( i<=mid ) a[k++] = b[i++]; //若第一个表未检测完，复制
    while ( j<=high ) a[k++] = b[j++]; //若第二个表未检测完，复制
}
```

2. 性能分析

(1) 空间效率：Merge()操作中，辅助空间刚好要占用 n 个单元，所以归并排序的空间复杂度为 $O(n)$ 。

(2) 时间效率：每一趟归并的时间复杂度为 $O(n)$ ，共需进行 $\lceil \log_2 n \rceil$ 趟归并，所以算法的时间复杂度为 $O(n \log n)$ 。

(3) 稳定性: Merge()不会改变相同关键字记录的相对次序, 归并排序是稳定的排序。

【例 8-9】对 {24, 85, 47, 53, 30, 91} 归并排序。

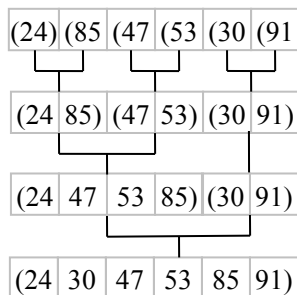


图 8-11 归并排序

【习题演练】

1. 下述几种排序方法中, 要求内存最大的是 ()。

- A. 希尔排序 B. 快速排序 C. 归并排序 D. 堆排序

二、基数排序

基数排序是和前面所述各类排序方法完全不相同的一种排序方法。基数排序是一种借助多关键字排序的思想, 采用“分配”和“收集”策略对单逻辑关键字进行排序的方法。基数排序又分为最高位优先排序和最低位优先排序。

以 r 为基数的最低位优先基数排序的过程: 假设线性表由结点序列 a_0, a_1, \dots, a_{n-1} 构成, 每个结点 a_j 的关键字由 d 元组 $(k_j^{d-1}, k_j^{d-2}, \dots, k_j^1, k_j^0)$ 组成, 其中 $0 \leq k_j^i \leq r-1$ ($0 \leq j < n, 0 \leq i \leq d-1$)。

在排序过程中, 使用 r 个队列 Q_0, Q_1, \dots, Q_{r-1} 。排序过程如下:

对 $i=0, 1, \dots, d-1$, 依次做一次“分配”和“收集”(其实就是一次稳定的排序过程)

分配: 开始时, 把 Q_0, Q_1, \dots, Q_{r-1} 各个队列置成空队列, 然后依次考察线性表中的每一个结点 a_j ($j=0, 1, \dots, n-1$), 如果 a_j 的关键字 $k_j^i = k$, 就把 a_j 放进 Q_k 队列中。

收集: 把 Q_0, Q_1, \dots, Q_{r-1} 各个队列中的结点依次首尾相接, 得到新的结点序列, 从而组成新的线性表。

1. 算法

```
void Distribute (SLCell &r, int i, ArrType &f, ArrType &e) {
    // 静态链表 L 的 r 域中记录已按 (keys[0], ..., keys[i-1]) 有序。
```

相同。
// 本算法按第 i 个关键字 $keys[i]$ 建立 RADIX 个子表, 使同一子表中记录的 $keys[i]$

$f[0 \dots RADIX-1]$ 和 $e[0 \dots RADIX-1]$ 分别指向各子表中第一个和最后一个记录。

for($j=0$; $j<Radix$; $++j$) $f[j]=0$; // 各子表初始化为空表

for($p=r[0].next$; p ; $p=r[p].next$) {

$j = ord(r[p].keys[i]);$ // ord 将记录中第 i 个关键字映射到 $[0 \dots RADIX-1]$

if ($!f[j]$) $f[j]=p$;

else $r[e[j]].next=p$;

$e[j]=p$; // 将 p 所指的结点插入第 j 个子表中

}

}

void Collect(SLCell &r, int i, ArrType f, ArrType e) {

// 本算法按 $keys[i]$ 自小至大地将 $f[0 \dots RADIX-1]$ 所指各子表依次链接成一个链表

// $e[0 \dots RADIX-1]$ 为各子表的尾指针。

for($j=0$; $!f[j]$; $j=succ(j)$); // 找第一个非空子表, $succ$ 为求后继函数

$r[0].next=f[j]$; $t=e[j]$; // $r[0].next$ 指向第一个非空子表中第一个结

点

while($j<RADIX$) {

for($j=succ(j)$; $j<RADIX-1 \ \&\& \ !f[j]$; $j=succ(j)$); // 找下一个非空子表

if($f[j]$) $\{r[t].next=f[j]$; $t=e[j]\}$; // 链接两个非空子表

}

$r[t].next=0$; // t 指向最后一个非空子表中的最后一个结点

}

void RadixSort(SLList &L) {

// L 是采用静态链表表示的顺序表。

// 对 L 作基数排序, 使得 L 成为按关键字自小到大的有序静态链表, $L.r[0]$ 为头

结点

for($i=0$; $i<L.recnum$; $++i$) $L.r[i].next=i+1$;

$L.r[L.recnum].next=0$; // 将 L 改造为静态链表

for($i=0$; $i<L.keynum$; $++i$) { // 按最低位优先依次对各关键字进行分配和收集

Distribute($L.r$, i , f , e); // 第 i 趟分配

Collect($L.r$, i , f , e); // 第 i 趟收集

}

}

【例 8-10】 {503, 087, 512, 061, 908, 170, 897, 275, 653}

每个关键字由数字 0~9 组成, 基数 $r=10$, 需使用 10 个队列。每个关键字都有 3 位数字, $d=3$, 需进行 3 趟分配和收集。

1) 先“收集”成一个链表，按最低位（个位）“分配”到 10 个链表中（0 号~9 号）：

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
170	061	512	503		275		087	908	
			653				897		

按个位顺序出队“收集”成一个链表：

(170, 061, 512, 503, 653, 275, 087, 897, 908)

2) 再按第 2 位数字（十位）“分配”到 10 个链表中：

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
503	512				653	061	170		897
908							275		

“收集”成一个链表：

(503, 908, 512, 653, 061, 170, 275, 897)

3) 按第 3 位数字（百位）“分配”到 10 个链表中：

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
061	170	275			503	653		897	908
					512				

“收集”成一个链表：

(061, 170, 275, 503, 512, 653, 897, 908)

完成排序。

2. 性能分析

(1) 空间效率：一趟排序需要的辅助存储空间为 r (r 个队列)，但以后的排序中重复使用这些队列，所以基数排序的空间复杂度为 $O(r)$ 。

(2) 时间效率：对 n 个数据进行基数排序，每个数据基数为 r ，有 d 位数字。那么，一趟分配和收集用时 $n+r$ (分配用 n ，收集用 r)，共需 d 趟，总的时间复杂度为 $O(d(n+r))$ 。它与序列的初始状态无关。

(3) 稳定性：对于基数排序算法而言，很重要一点就是按位排序时必须是稳定的。因此，这也保证了基数排序保持稳定性。基数排序是稳定的排序算法。

【习题演练】

1. 排序算法中元素的移动次数和关键字的初始排列次序无关的是 ()。

A. 直接插入排序 B. 起泡排序 C. 基数排序 D. 快速排序

第五节 各种排序方法比较

综合比较本章讨论的各种内部排序方法，大致有如下结果，如下表 8-1：

表 8-1 排序方法的比较表

排序方法	时间复杂性			空间 复杂性	稳定	特点
	平均	最好	最坏			
简单插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	是	元素少或基本有序时高效
折半插入排序	$O(n^2)$	$O(n\log_2 n)$	$O(n^2)$	$O(1)$	是	比较与移动分离开来
希尔排序				$O(1)$	否	仅适用于顺序表
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	是	子序列全局有序
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否	平均时间性能最好
简单选择排序	$O(n^2)$			$O(1)$	否	比较次数最多
堆排序	$O(n\log_2 n)$			$O(1)$	否	辅助空间少
归并排序	$O(n\log_2 n)$			$O(n)$	是	稳定的
基数排序	$O(d(n+r))$			$O(r)$	是	适合个数多关键字较小

基于关键字比较的排序方法，在最坏情况下所能达到的最好的时间复杂度是 $O(n\log_2 n)$ 。