第3章 数据结构

CHAPTER 3

数据结构是研究非数值计算程序设计中计算机的操作对象以及它们之间关系和运算的科学。数据结构与数学、计算机硬件和计算机软件等有着密切的关系，数据结构与算法密不可分，是操作系统、编译原理、数据库、情报检索、人工智能等学科的重要基础。

3.1 数据的逻辑结构与存储结构

3.1.1 基本概念

数据是信息的载体，是描述客观事物的数、字符以及所有能被输入计算机中并被计算机程序识别和处理的符号的集合，包括数值性数据和非数值性数据。

1. 数据元素、数据项和数据对象

数据元素是数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。一个数据元素可以由若干个数据项组成。数据项是在数据处理时不能再分割的最小单位。数据对象是性质相同的数据元素的集合。数据对象亦称为数据元素类。数据元素是数据对象的一个实例。

例如，学生张强的学籍信息集合(表)是数据元素，学生学籍信息表中的每一项，如学号、姓名、性别等各自为一个数据项。特征相同且具有共同数据项的众多学生数据可形成一个学生数据对象 student。例如：

```
student = { 张强,李兵,… }
```

任何问题中，数据元素之间都不是孤立的，它们之间存在着某种关系，数据元素之间的关系称为结构。

2. 数据结构

数据结构是互相之间存在关系的数据元素的集合。数据结构将数据按某种逻辑关系组织起来，按一定的存储表示方式把它们存储在计算机存储器中，并在这些数据上定义一个运算的集合。数据结构与数据类型和数据对象不同，它不仅要描述数据类型的数据对象，还要描述数据对象各元素之间的相互关系。

数据结构通常包括逻辑结构和存储结构。逻辑结构用于描述数据之间的逻辑关系，存储结构描述数据如何在计算机内存储。

通常，用计算机解决一个具体问题时，可分为以下步骤。

(1) 从具体问题抽象出一个适当的数学模型。

(2) 设计一个解此数学模型的算法。

(3) 编出程序,进行测试、调试,直至得到最终解答。

寻求数学模型的实质是分析问题,从中提取操作的对象,并找出这些操作对象之间含有的关系,然后用数学的语言加以描述。数值问题可以用诸如方程等描述。而非数值计算问题的数学模型则是用诸如表、树和图之类的数据结构描述。

3. 数据操作

数据操作亦称为数据运算。数据运算是数据结构的一个重要方面,对任何一种数据结构的研究都离不开对该结构上的数据运算及其实现算法的研究。最常用的数据操作有 5 种:插入、删除、修改、查找和排序。例如针对线性表,常见的基本操作如下。

(1) 线性表初始化。构造一个空的线性表。

(2) 求线性表的长度。返回线性表中所含元素的个数。

(3) 取表元。返回线性表 L 中的第 i 个元素的值或地址。

(4) 按值查找。在线性表 L 中查找值为 x 的数据元素,其结果返回在 L 中首次出现的值为 x 的元素的地址;若未找到,返回一个特殊值表示查找失败。

(5) 插入操作。在线性表 L 的第 i 个位置插入一个值为 x 的新元素。

(6) 删除操作。删除线性表 L 中序号为 i 的数据元素。

数据结构的操作定义在逻辑结构层次上,而操作的具体实现建立在存储结构基础上。每个操作的算法只有在存储结构确立之后才能实现。

图 3-1 描述了数据结构的 3 个研究内容。

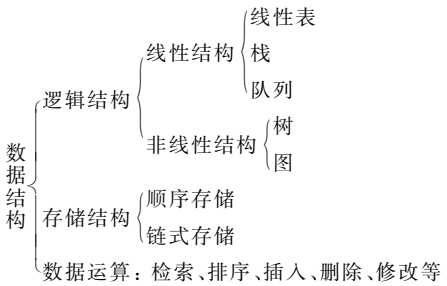


图 3-1　数据结构的研究内容

3.1.2　数据的逻辑结构

数据的逻辑结构反映数据之间的逻辑关系,是对数据之间关系的描述,可以用一个二元组来表示:$S=(D,R)$。其中 D 是有限个数据元素构成的集合,R 是有限个结点间关系的集合。数据的逻辑结构主要有线性表、树、图等形式。数据的逻辑结构和存储结构是密不可分的两方面,一个算法的设计取决于所选定的逻辑结构,而算法的实现依赖于所采用的存储结构。

1. 线性表、栈与队列

线性表是最常用、最简单的一种数据结构,其基本特点是线性表中的数据元素是有序且有限的。线性表中数据元素用结点描述,结点之间是一对一的关系。在线性表里有且仅有

一个开始结点和一个终端结点，并且所有结点最多只有一个前驱和一个后继。现实中有很多一对一的线性关系，如英文字母表、一个班中的学生关系（通过学号关联），图书馆中的书籍（通过书号关联）。

栈与队列是两种特殊的线性结构。从结构看它们与普通线性表一样，但执行数据操作时它们受特定规则限制。

栈是定义在线性结构上的抽象数据类型，其操作类似线性表操作，但元素的插入、删除和访问都必须在表的一端进行，其操作如图 3-2 所示。为形象起见，允许操作端称为栈顶（top），另一端为栈底（base），栈的特性为先进后出、后进先出。编程中嵌套函数和递归函数的调用控制、括号匹配问题、运算表达式的计算等均可用栈模拟。

队列是另一种线性表，类似日常生活中排队，队列元素的插入和删除分别在表的两端进行，如图 3-3 所示。允许插入的一端为队尾（rear），允许删除的一端为队头（front）。队列的特性为先进先出、后进后出，操作系统中的作业队列和打印任务队列、日常生活中各类排队业务等均可用队列模拟。

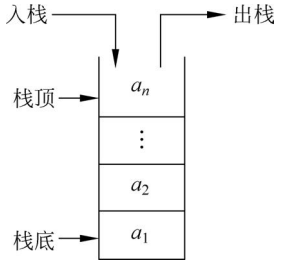


图 3-2　栈示意图

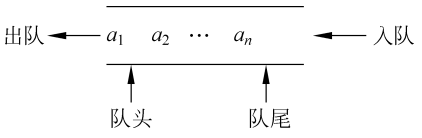


图 3-3　队列示意图

2. 树与图

树与图结构均为非线性结构。树结构中数据元素之间是一对多的关系。在树中有且仅有一个结点没有前驱，类似于树的根，称为根结点；其他结点有且仅有一个前驱。它的结构特点具有明显的层次关系。

日常生活及计算机中有很多数据关系是树结构，例如家谱、行政组织机构、源程序的语法结构、资源管理器、人机对弈问题等，如图 3-4 和图 3-5 所示。

图 3-4　资源管理器

图结构中数据元素之间是多对多的关系，图是由结点的有穷集合 V 和边的集合 E 组成。其中，为了与树结构加以区别，在图结构中常常将结点称为顶点，边是顶点的有序偶对，若两个顶点之间存在一条边，就表示这两个顶点具有相邻关系。图结构也称作网状结构。

互联网结构、教学计划编排问题、交通网络图等都可以用图结构描述，如图 3-6 和图 3-7 所示。

3.1.3　数据的存储结构

数据的逻辑结构从逻辑关系上描述数据，是独立于计算机的；数据的存储结构是逻辑结构在计算机存储器里的

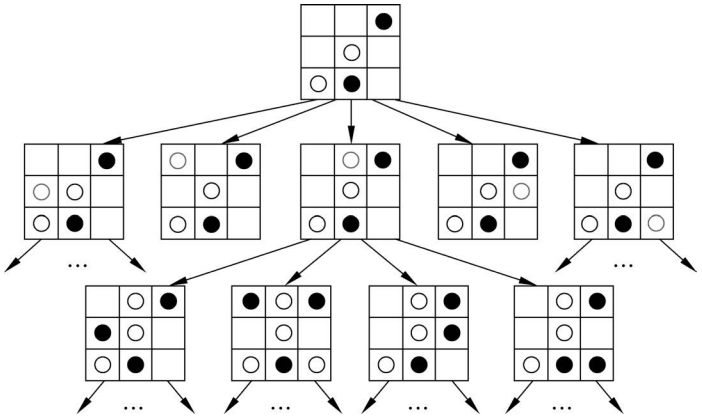


图 3-5　对弈问题

编号	课程名称	先修课
C_1	高等数学	无
C_2	计算机导论	无
C_3	离散数学	C_1
C_4	程序设计	C_1, C_2
C_5	数据结构	C_3, C_4
C_6	计算机原理	C_2, C_4
C_7	数据库原理	C_4, C_5, C_6

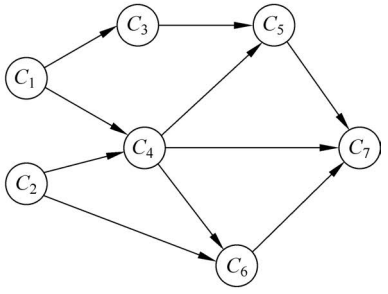


图 3-6　教学计划编排问题

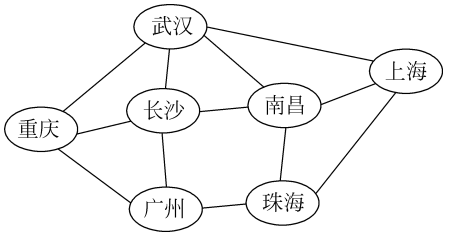


图 3-7　交通网络图

实现，是依赖于计算机的。数据的存储结构主要有顺序存储结构、链式存储结构、索引存储结构、散列存储结构 4 种，并可以根据需要组合成其他更复杂的结构。

1. 顺序存储结构

顺序存储结构是一种最基本的存储方法，是借助元素在存储器中的相对位置来表示数据元素间的逻辑关系。这种方法主要用于线性的数据结构，它把逻辑上相邻的结点存储在物理上相邻的存储单元里，结点之间的关系由存储单元的邻接关系来体现。在 C 语言中，通常借助一维数组表示顺序存储结构。

2. 链式存储结构

链式存储结构即在每一个数据元素中增加一个存放另一个元素地址的指针，用该指针来表示数据元素之间的逻辑关系，由此得到的存储表示称为链表。链表既可以表示线性结

构，也可以表示非线性结构。

链表中每一个元素称为结点。在C语言中，用结构体类型表示结点，链表由一系列结点组成，结点可以在运行时动态生成。结点所占的存储单元分为两部分：一部分存放结点本身的信息，称为数据项；另一部分存放结点的后继结点所对应的存储单元的地址，称为指针项。指针项可以包含一个或多个指针，以指向结点的一个或多个后继。

3. 索引存储结构

索引存储结构指除建立存储结点信息外，还建立附加的索引表标识结点的地址。索引存储结构的优点是检索速度快，缺点是增加了附加的索引表，会占用较多的存储空间。

4. 散列存储结构

散列存储结构又称hash存储结构，是一种将数据元素的存储位置与关键码之间建立确定对应关系的查找技术。

若数据结构中存在关键字和 K 相等的记录，则必定在 $f(K)$ 的存储位置上。由此，不须比较便可直接取得所查记录，称这个对应关系 $f(k)$ 为散列函数，按这个思想建立的表为散列表。散列技术除了可以用于查找外，还可以用于存储。

同一个逻辑结构可以用不同的存储结构存储，本章主要介绍顺序存储与链式存储。具体选择哪一种需根据数据特点及实际运算的效率来确定。

3.2 线性表

线性表是 $n(n\geqslant 0)$ 个具有相同属性的数据元素 $a_1,a_2,\cdots,a_n$ 组成的有限序列，线性表中每一个数据元素均有一个直接前驱和一个直接后继数据元素。当 $n=0$ 时，称为空表，空表不含有任何元素。

3.2.1 线性表的顺序存储和操作

1. 线性表的顺序存储

线性表的顺序存储是指在内存中把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里，用这种方法存储的线性表称为顺序表。顺序表中数据元素之间的逻辑关系以元素在计算机内物理位置相邻来表示。

存储地址	内存空间状态	逻辑地址
$\mathrm{loc}(a_1)$	a_1	1
$\mathrm{loc}(a_1)+k$	a_2	2
⋮	⋮	⋮
$\mathrm{loc}(a_1)+(i-1)k$	a_i	i
⋮	⋮	⋮
$\mathrm{loc}(a_1)+(n-1)k$	a_n	n
		空闲

图 3-8 顺序表的存储地址

由于顺序表的所有数据元素属于同一数据类型，所以每个元素在存储器中占用的空间（字节数）相同。因此，要在此结构中查找某一个元素是很方便的，只要知道顺序表首地址和每个数据元素在内存所占字节的大小就可求出第 i 个数据元素的地址，因此顺序存储结构的线性表可以随机存取其中的任意元素。

假设顺序表中有 n 个元素，每个元素占 k 个单元，第一个元素的地址 $\mathrm{loc}(a_1)$ 称为基地址，第 i 个元素的地址 $\mathrm{loc}(a_i)$ 可以通过如下公式计算：$\mathrm{loc}(a_i)=\mathrm{loc}(a_1)+(i-1)k$，如图3-8所示。

顺序存储结构在C语言中用一维数组表示，一维

数组的下标等于元素在顺序表中的序号减 1。

```
typedef struct
        { datatype data[MAXSIZE];
              int last;
          }SeqList;
```

其中：datatype 为抽象数据类型；

MAXSIZE 为线性表中最多可以存放的元素个数；

last 为最后一个元素的位置。

2. 顺序表的操作

顺序表的操作主要包括顺序表的初始化、插入及删除数据、数据的查询及求顺序表的长度等。

(1) 顺序表的初始化。

初始化即构造一个空的顺序表，为顺序表命名及分配空间。

```
SeqList * init_SeqList()
{
   SeqList * L;
   L = (SeqList * )malloc(sizeof(SeqList));
   L -> last = -1;
   return L;
}
```

(2) 插入运算。

插入运算是指在顺序表的第 $i-1$ 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。对于长度可变的顺序表，必须按可能达到的最大长度分配空间。

已知顺序表(4,9,15,28,30,30,42,51,62)，需在第 4 个元素之前插入一个元素“21”，则需要将第 9 个位置到第 4 个位置的元素依次后移一个位置，然后将“21”插入第 4 个位置，如图 3-9 所示。

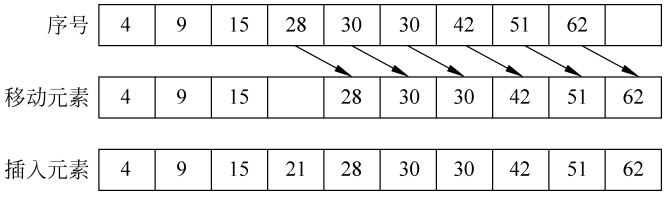


图 3-9　顺序表中插入元素

代码如下：

```
int Insert_SeqList(SeqList * L,int i,datatype x)
{
   int j;
   if(L -> last == MAXSIZE - 1)
   {
      printf("table is full!"); return( -1);
   }
   if(i < 1||i >(L -> last + 2))
   {
      printf("place is wrong!"); return(0);
   }
```

```
    for(j = L -> last;j >= i - 1;j -- )
    {
       L -> data[j + 1] = L -> data[j];
    }
    L -> data[i - 1] = x;
    L -> last++;
    return(1);
}
```

(3) 删除运算。

顺序表的删除运算是将表的第 $i(1\leqslant i\leqslant n)$个元素删除，使长度为 n 的顺序表$(e_1,\cdots,e_{i-1},e_i,e_{i+1},\cdots,e_n)$变成长度为 $n-1$ 的顺序表$(e_1,\cdots,e_{i-1},e_{i+1},\cdots,e_n)$。删除第 i 个元素$(1\leqslant i\leqslant n)$时需将第 $i+1$ 至第 n(共 $n-i$)个元素依次向前移动一个位置。

例如，删除顺序表(4,9,15,21,28,30,30,42,51,62)第 5 个元素，则需将第 6 个元素到第 10 个元素依次向前移动一个位置，如图 3-10 所示。

序号	1	2	3	4	5	6	7	8	9	10
	4	9	15	21	**28**	30	30	52	51	62
删除28后	4	9	15	21	30	30	42	51	62	

图 3-10　顺序表中删除元素

代码如下：

```
int Delete_SeqList(SeqList * L,int i)
{
   int j;
   if(i < 1||i >(L -> last + 1))
   {
      printf("this element don't exist!");
   return(0);
   }
   for(j = i;j <= L -> last;j++)
   {
      L -> data[j - 1] = L -> data[j];
   }
   L -> last -- ;
   return(1);
}
```

在顺序表中插入或删除一个数据元素时，其时间主要耗费在移动数据元素上。对于插入算法而言，设 p_i 为在第 i 个元素之前插入元素的概率，平均移动次数为

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

假设在任何位置上插入的概率相等，即 $p_i=1/(n+1)$，$i=1,2,\cdots,n+1$，平均移动次数为

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

同理，设 Q_i 为删除第 i 个元素的概率，并假设在任何位置上删除的概率相等，即 $Q_i=1/n$，$i=1,2,\cdots,n$。删除一个元素所需移动元素的平均次数为

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

由此可得，在顺序表中作插入或删除运算时，平均有一半元素需要移动，时间复杂度为$O(n)$。（时间复杂度的概念详见 4.1.3 节）

(4) 顺序表的查找操作。

在线性表中查找关键字为 x 的元素，并返回其位置。

```
int Locate_list(int s[], int n, int x)
{
   int i;
   for (i = 1; i <= n; i++)
   if (s[i] == x) return(i);
   return(0);
}
```

例 3.1　从一个有序顺序表中删除重复的元素并返回新的表长，要求空间复杂度为$O(1)$。代码如下：

```
#include <stdio.h>
typedef int ElemType;
typedef struct
{
   ElemTypedata[100];
   int length;
}SeqList;
int removeSame (SeqList &B)
{
   ElemTypee = B.data[0];
   int index = 1;
   for(int i = 1;i < B.length;++i)
   {
      if(B.data[i]!= e)
      {
         B.data[index++] = B.data[i];
         e = B.data[i];
      }
   }
   return index;
}

int main()
{
   SeqList R;
   int i;
   int A[] = {1,2,2,3,3,3,4,4,5,5};
   for(i = 0;i < sizeof(A)/4;++i)                    //顺序表初始化
   R.data[i] = A[i];
   R.length = i;
   R.length = removeSame (R);
   printf("删除前:\n");
   for(i = 0;i < sizeof(A)/4;++i)
   printf("%2d",A[i]);
   printf("\n");
   printf("删除后:\n");
   for(i = 0;i < R.length;++i)
   printf("%2d",R.data[i]);
```

```
    printf("\n");
    return 0;
}
```

显然，线性表的顺序存储具有如下优点。

（1）方法简单，各种高级语言中都有数组，容易实现。

（2）不用为表示结点间的逻辑关系而增加额外的存储开销。

（3）具有按元素序号随机访问的特点。

但线性表的顺序存储也存在以下缺点。

（1）数据元素最大个数需预先确定，使得高级程序设计语言编译系统需预先分配相应的存储空间，存储空间不便于扩充。

（2）插入与删除运算的效率很低。为了保持线性表中的数据元素顺序，在插入操作和删除操作时需移动大量数据。对于插入和删除操作频繁的线性表，将导致系统的运行速度难以提高。

3.2.2 线性表的链式存储和操作

1. 线性表的链式存储

线性表的链式存储结构又称为线性链表，就是用一组任意的存储单元（可以是不连续的）存储线性表的数据元素，每个存储单元称为结点。每个结点包含两部分内容：一部分用于存放数据元素值，称为数据域；另一部分用于存放直接前驱或直接后继结点的地址（指针），称为指针域。结点的结构示意图如图 3-11 所示。

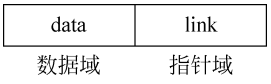
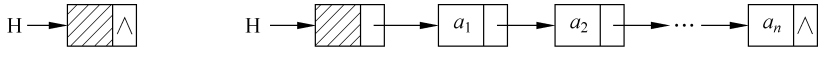


图 3-11　结点的结构示意图

链表的每个结点只有一个指针域，这种链表又称为单链表。由于单链表中每个结点的存储地址是存放在其前趋结点的指针域中，而第一个结点无前趋，因而应设一个头指针 H 指向第一个结点。同时，由于表中最后一个结点没有直接后继，则指定线性表中最后一个结点的指针域为“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，这样对于整个链表的存取必须从头指针开始。图 3-12 描述了带头结点的空单链表和单链表。



(a) 带头结点的空单链表　　(b) 带头结点的单链表

图 3-12　带头结点的空单链表和单链表

线性链表的有关术语如下。

（1）头指针。用于存放线性链表中第一个结点的存储地址。

（2）空指针。不指向任何结点。

（3）带头结点的线性链表。在第一个结点前面增加一个附加结点的线性链表，称为带头结点的线性链表。

（4）单链表。只有一个指针域的线性链表。

C 语言中用带指针的结构体类型描述结点，代码如下：

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

其中：

LNode：结构类型名；

data：用于存放元素的数据信息；

next：用于存放元素直接后继结点的地址。

该类型结构变量用于表示线性链表中的一个结点。

```
LNode * p;                                  /* p 为指向结点(结构)的指针变量 */
LinkList p;                                 /* 同 LNode * p; */
```

2. 单链表的操作

为了运算方便起见，一般用带头结点的单链表存储线性表。

(1) 单链表的创建。

动态创建单链表有头插法、尾插法两种方法。头插法是从一个空表开始，重复读入数据，生成新结点，将读入数据存放到新结点的数据域中，然后将新结点插入当前链表的表头上，直到读入结束标志为止，即每次插入的结点都作为链表的第一个结点。尾插法是将新结点插入当前链表的表尾，使其成为当前链表的尾结点。头插法建立链表算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。

例如，创建函数 create_LinkList()，实现头插法创建单链表，链表的头结点 head 作为返回值。代码如下：

```
LNode * create_LinkList(void)
{
    int data ;
    LNode * head, * p;
    head = (LNode * ) malloc( sizeof(LNode));
    head -> next = NULL;                            //创建链表的表头结点 head
    while (1)
    {
        scanf(" % d", &data) ;
        p = (LNode * )malloc(sizeof(LNode));
        p -> data = data;                           //数据域赋值
        p -> next = head -> next ; head -> next = p ; //新创建的结点总是作为第一个结点
    }
    return(head);
}
```

创建函数 create_LinkList1()，实现尾插法创建单链表函数，链表的头结点 head 作为返回值。代码如下：

```
LNode * create_LinkList1(void)
{
    int data ;
    LNode * head, * p, * q;
    head = p = (LNode * )malloc(sizeof(LNode));
```

```
    p->next = NULL;                                    //创建单链表的表头结点 head
    while (1)
    {
       scanf("%d",&data);
       q = (LNode *)malloc(sizeof(LNode));
       q->data = data;                                 //数据域赋值
       q->next = p->next; p->next = q; p = q;          //新创建的结点总是作为最后一个结点
    }
    return(head);
}
```

无论是哪种插入方法，如果要插入建立的单链表的结点是 n 个，算法的时间复杂度均为 $O(n)$。

(2) 单链表的插入运算。

链表中插入元素只需修改插入元素及其前趋元素的指针即可，操作步骤如图 3-13 所示。

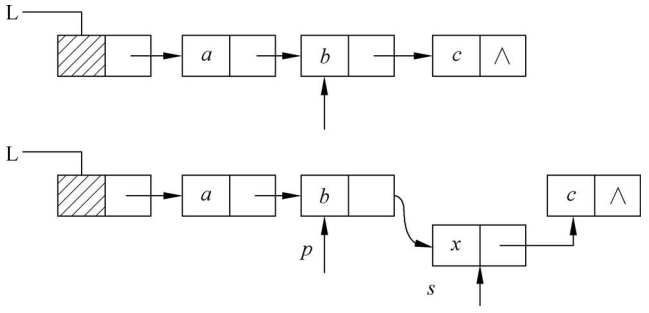


图 3-13　线性链表插入元素

创建函数 Inset()，实现在第 i 个结点前插入数据值为 x 的结点的函数。代码如下：

```
int Insert (NODE *head, int i, int x)
{
   NODE *p, *q;
   int j;
   if (i<=0) return(0);
   p = head;
   j = 1;
   while ((j<=i-1)&&(p!=NULL)) {p = p->link; j++;}
   if (p==NULL) return(0);
   q = (NODE *)malloc(sizeof(NODE));
   q->data = x; q->link = p->link; p->link = q;
   return(1);
}
```

(3) 单链表的删除运算。

链表中删除操作只需修改被删除元素前趋元素指针即可，操作步骤如图 3-14 所示。

创建函数 Delete()，实现删除指定位置(第 i 个)元素。代码如下：

```
int Delete (NODE *head, int i)
{
   NODE *p, *q;
   int j;
   if (i==0) {
            p = head; head = head->link;
```

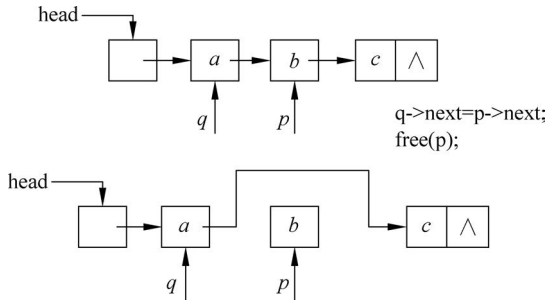


图 3-14　线性链表删除元素

```
            free(p);
            return(1);
            }
    j = 1;p = head;
    while ((j < i)&&(p -> link!= NULL)) p = p -> link;
    if (p -> link == NULL) return (0);
    q = p -> link; p -> link = q -> link;
    free(q);
    return(1);
}
```

(4) 线性链表的查找运算。

创建函数 Locate_LinkList(),实现链表中按元素值查找。代码如下:

```
Lnode * Locate_LinkList(LinkList L,datatype x)
{
    LNode * p = L -> next;
    while(p!= NULL&&p -> data!= x) p = p -> next;
    return p;
}
```

3. 循环链表的操作

循环链表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。从循环链表的任意一个结点出发都可以找到链表中的其他结点,使得表处理更加方便灵活。循环链表和单链表的差别仅在于链表中最后一个结点的指针域不为 NULL,而是指向头一个结点,成为一个由链指针链接的环,也就是算法中的循环条件不是 p 或 p－＞next 是否为空,而是它们是否等于头指针。循环链表示意图如图 3-15 所示。

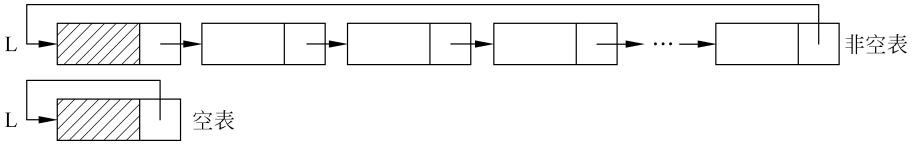


图 3-15　循环链表示意图

循环链表的操作与单链表相似。

4. 双向链表的操作

若结点中有两个指针域,一个指向直接后继,另一个指向直接前趋,这样的链表称为双向链表,如图 3-16 所示。双向链表可以克服单链表的单向性缺陷。

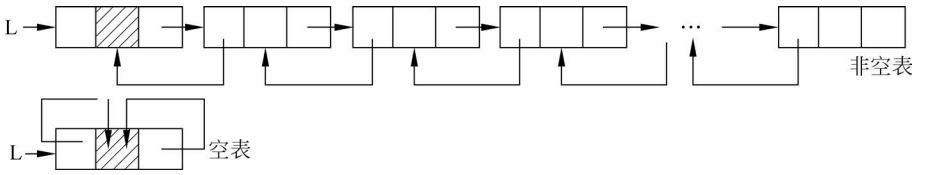


图 3-16　双向链表示意图

双向链表存储结构如下：

```
struct Double_node
{
   int data;
   struct Double_node * llink, * rlink;
};
typedef struct Double_node NODE;
```

对指向双向链表任一结点的指针 d，具有关系：d->rlink->llink=d->llink->rlink=d，即当前结点后继的前趋是自身，当前结点前趋的后继也是自身。与单链表的插入和删除操作不同的是，在双向链表中插入和删除必须同时修改两个方向上的指针域的指向。

（1）双向链表的插入操作。

在数据值为 y 的结点后，插入数据值为 x 的结点，如图 3-17 所示。

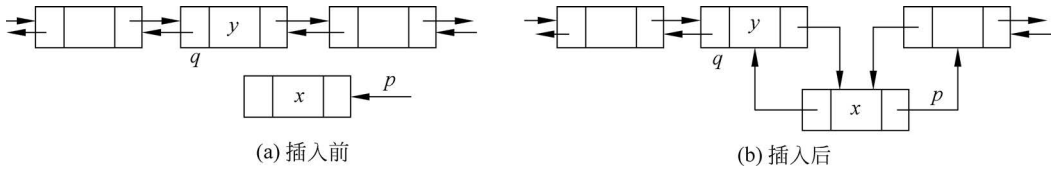


图 3-17　双向链表插入结点示意图

创建函数 Insert2()，实现双向链表插入结点操作。代码如下：

```
int Insert2(NODE * head,int x,int y)
{
   NODE * p, * q;
   q = head->rlink;
   while (q!= head && q->data!= y) q = q->rlink;
   if (q == head) return(0);
   p = (NODE * )malloc(sizeof(NODE));
   p->data = x; p->llink = q; p->rlink = q->rlink;
   (q->rlink)->llink = p;
   q->rlink = p;
   return(1)
}
```

（2）双向链表的删除操作。

在双向链表中删除数据值为 x 的结点，如图 3-18 所示。

创建函数 Delete2()，实现双向链表删除结点操作。代码如下：

```
int Delete2 (NODE * head,int x)
{
   NODE * q;
   q = head->rlink;
   while(q!= head && q->data!= x) q = q->rlink;
```

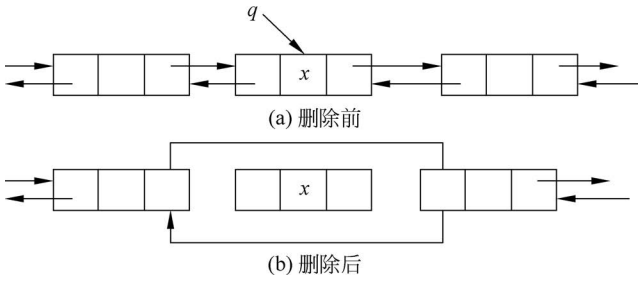


图 3-18　双向链表删除结点

```
    if(q == head) return(0);
    (q -> llink) -> rlink = q -> rlink;
    (q -> rlink) -> llink = q -> llink;
    free(q);
    return(1);
}
```

3.2.3　小结

链表的优缺点恰好与顺序表相反。在实际应用中选取何种存储结构，通常从以下几方面考虑。

1. 基于空间的考虑

顺序表的存储空间是静态分配的，在程序执行前必须明确规定它的存储规模。若线性表长度 n 变化较大，则存储规模很难预先正确估计。估计太大将造成空间浪费，估计太小又将使空间溢出机会增多。链表不用事先估计存储规模，是动态分配，只要内存空间尚有空闲，就不会产生溢出，所以当对线性表的长度或存储规模难以估计时，不宜采用顺序存储结构而宜采用动态链表作为存储结构。但链表的存储密度较低，存储密度是指一个结点中数据元素所占的存储单元和整个结点所占的存储单元之比。显然链式存储结构的存储密度是小于 1 的，顺序表的存储密度等于 1。

2. 基于时间的考虑

线性表如果主要做查找，则时间性能为 $O(1)$；而在链表中按序号访问的时间性能为 $O(n)$。所以，如果经常做的运算是按序号访问数据元素，显然顺序表优于链表。

顺序表中做插入、删除操作时，要平均移动表中一半的元素；尤其是当每个结点的信息量较大时，移动结点的时间开销就相当可观，不容忽视。在链表中的任何位置上进行插入和删除，都只需要修改指针。对于频繁进行插入和删除的线性表，宜采用链表做存储结构。若表的插入和删除主要发生在表的首尾两端，则宜采用尾指针表示的单循环链表。

3. 基于环境的考虑

顺序表容易实现，任何高级语言中都有数组类型；链表的操作是基于指针的，其使用受语言环境的限制。

总之，两种存储结构各有特点，选择哪种结构根据实际使用线性表的主要因素决定。通常较稳定的线性表选择顺序存储结构；而插入或删除操作频繁的动态性较强的线性表宜选择链式存储结构。

3.2.4 栈

1. 栈的定义

栈是限定仅在表尾进行插入或删除操作的线性表。表尾称为栈顶，表头称为栈底，不含元素的空表称为空栈。栈按照先进后出的原则存储数据，先进入的数据被压入栈底，最后进入的数据在栈顶，需要读数据的时候从栈顶开始弹出数据，即最后一个数据被第一个读出来。栈的进出操作如图 3-19 所示。

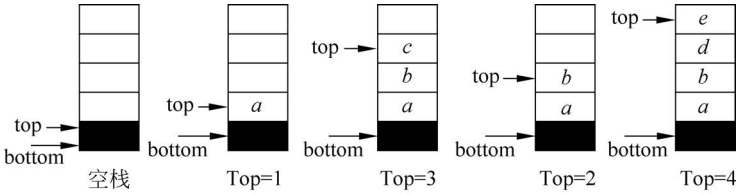


图 3-19 进出栈操作

栈的插入和删除操作分别称为进栈和出栈。进栈是将一个数据元素存放在栈顶，出栈是将栈顶元素取出。栈按存储方式可分为两种：顺序栈和链栈。

例 3.2 假设有 3 个数据元素 a，b，c，入栈顺序是 a，b，c，则它们的出栈顺序有几种可能？

出栈顺序共有如下几种。

（1）abc 顺序进栈则出栈顺序为 cba。

（2）a 进栈，a 出栈，然后 b、c 进栈，再顺序出栈，则出栈顺序为 acb。

（3）a 进栈，a 出栈；b 进栈，b 出栈；c 进栈，c 出栈；则出栈顺序为 abc。

（4）a、b 进栈，a、b 出栈，然后 c 进栈，再出栈，则出栈顺序为 bac。

（5）a、b 进栈，b 出栈；c 进栈，然后出栈，则出栈顺序为 bca。

2. 顺序栈的存储和操作

顺序栈即栈的顺序存储，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。指针 top 值随着插入和删除而变化。

在栈的操作中有两种异常状态需设法避免，即栈满时做进栈运算和栈空时做退栈运算将产生溢出。因此，入栈要先判断栈是否满，出栈要先判断栈是否空。

（1）顺序栈描述。

```
#define M 10
typedef struct
{
   int elem[M];
   int top;
} SQSTACK;
```

（2）顺序栈的操作。

① 判断栈是否为空。

```
int StackEmpty(SQSTACK stack)
{
    if (stack.top == -1) return(1);
```

```
    return(0);
}
```

② 进栈。

```
int Push(SQSTACK * stack, int x)
{
    if (stack->top == M-1) return(0);
    stack->top++; stack->elem[stack->top] = x;
    return(1);
}
```

③ 出栈。

```
int Pop(SQSTACK * stack, int * x)
{
    if (StackEmpty( * stack)) return(0);
    * x = stack->elem[stack->top]; stack->top-- ;
    return(1);
}
```

④ 取栈顶元素。

```
int GetTop(SQSTACK stack, int * x)
{
    if (stack->top == -1) return(0);
    * x = stack->elem[stack->top];
    return(1);
}
```

3. 链栈的存储和操作

栈的链式存储结构称为链栈,因为栈是运算受限的单链表,其插入和删除操作只能在表头位置上进行。因此,链栈没有必要像单链表那样附设头结点,栈顶指针 top 就是链表的头指针。

(1) 链栈初始化。

```
Stack_Node * Init_Link_Stack(void)
{
   Stack_Node * top ;
   top = (Stack_Node * )malloc(sizeof(Stack_Node )) ;
   top->next = NULL ;
   return(top) ;
}
```

(2) 进栈。

```
Status push(Stack_Node * top , ElemType e)
{
   Stack_Node * p ;
   p = (Stack_Node * )malloc(sizeof(Stack_Node)) ;
   if (!p) return ERROR;                              //申请新结点失败,返回错误标志

   p->data = e ;
   p->next = top->next;
   top->next = p;
   return OK;
}
```

（3）出栈。

```
Status pop(Stack_Node * top , ElemType * e)
{
   Stack_Node * p ;
   ElemType e ;
   if (top -> next == NULL )
   return ERROR ;                                  //栈为空,返回错误标志
   p = top -> next ; e = p -> data ;               //取栈顶元素
   top -> next = p -> next ;                       //修改栈顶指针
   free(p) ;
   return OK ;
}
```

3.2.5 队列

1. 队列的定义

队列是一种先进先出、后进后出的线性表，限定所有的插入只能在表的一端进行，允许插入的一端称为队尾(rear)，删除的一端称为队头(front)，所有的删除运算都在表的另一端队头进行。如果队列按照 $a_1,a_2,\cdots,a_n$ 顺序入队，则出队顺序同样为 $a_1,a_2,\cdots,a_n$，即先进队列的元素先出队列，后进队列的元素后出队列。插入元素通常称为入队，删除元素通常称为出队。

队列的物理存储有两种方式，顺序存储的称为顺序队列，链式存储的称为链队列。

2. 顺序队列的存储和操作

在队列的顺序存储结构中，用一组地址连续的存储单元依次存放从队头到队尾的元素，附设两个指针 front 和 rear 分别指示队头和队尾的位置，如图 3-20 所示。

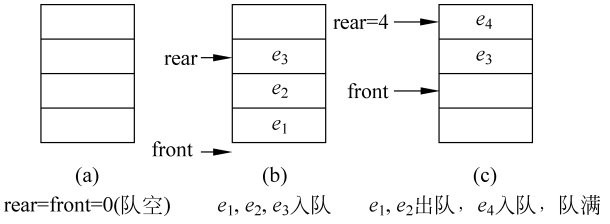


图 3-20 顺序队列操作

指针 front 和 rear 值的变化体现了队列的操作。

队列初始化：front＝rear＝0。

入队：将新元素插入 rear 所指的位置，rear 加 1。

出队：删去 front 所指的元素，front 加 1 并返回被删元素。

队列为空：front＝rear＝0。

队满：rear＝MAXSIZE－1。

（1）顺序队列的创建。

```
#define MAXSIZE 10
typedef struct
{
   elemtype elem[MAXSIZE];
```

```
    int front, rear;
} SQQUEUE;
```

(2) 判断队列是否为空。

```
int QueueEmpty (SQQUEUE q)
{
    if(q.front == 0&q.rear == 0) return(1);
    return(0);
}
```

(3) 顺序队列入队。

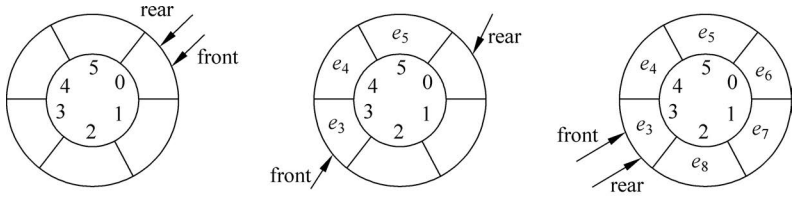
```
int AddQueue (SQQUEUE * q,elemtype e)
{
    if(q->rear == MAXSIZE - 1) return(0);
    q->rear = q->rear + 1;
    q->elem[q->rear] = e;
    return(1);
}
```

(4) 顺序队列出队。

```
int DelQueue (SQQUEUE * q, elemtype * e)
{
    if(q->front == 0&q.rear == 0) return(0);
    q->front = q->front + 1;
    * e = q->elem[q->front];
    return(1);
}
```

3. 循环队列的存储和操作

以数组方式顺序存储队列时，会发生假溢出现象，即在队列进出操作中，由于入队时尾指针向前追赶头指针；出队时头指针向前追赶尾指针，造成队列为空和队列为满时头指针与尾指针均相等，无法通过条件 rear＝MAXSIZE－1 来判读队列为“满”，为了更合理地利用空间，可将队列空间想象为一个首尾相接的圆环，这种队列称为循环队列，如图 3-21 所示。当尾指针移动到队列长度位置时，会再从 0 开始循环。



(a) 空队列　(b) 一般情况　(c) 队列满

图 3-21　循环队列

循环队列中队列为空条件依然是 front＝＝rear，队列为满可通过以下两种方式判断。

(1) 设置一个标志变量 flag，当 front＝＝rear，且 flag＝0 时为队列为空，当 front＝＝rear，且 flag＝1 时为队列为满。

(2) 另设一个标志位以区分队列是空还是满，即少用一个元素空间，当队列头指针在队

列尾指针的下一个位置上时为队列满，即(rear%MAXSIZE)+1==front。

```
//循环队列实现
#include<stdlib.h>
#include<stdio.h>
#define MAXSIZE 100                                    //最大队列长度
typedef int ElemType;
typedef struct
{
   ElemType*base;                                      //存储内存分配基地址
   int front;                                          //队列头索引
   int rear;                                           //队列尾索引
}circularQueue;
//初始化队列
InitQueue(circularQueue*q)
{
   q->base=(ElemType*)malloc((MAXSIZE)*sizeof(ElemType));
   if(!q->base)exit(0);                                //存储分配失败
   q->front=q->rear=0;
}
//入队列操作
InsertQueue(circularQueue*q,ElemType e)
{
   if((q->rear+1)%MAXSIZE==q->front) return;           //队列已满时,不执行入队操作
   q->base[q->rear]=e;                                 //将元素放入队列尾部
   q->rear=(q->rear+1)%MAXSIZE;                        /*尾部元素指向下一个空间位置,取模运算
                                                         保证了索引不越界(余数一定小于除
                                                         数)*/
}
//出队列操作
DeleteQueue(circularQueue*q,ElemType*e)
{
   if(q->front==q->rear) return;                       //空队列,直接返回
   *e=q->base[q->front];                               //头部元素出队
   q->front=(q->front+1)%MAXSIZE;
}
```

4. 链队列的存储和操作

用链表表示的队列简称为链队列。一个链队列需要两个指针分别指向队头和队尾元素，如图 3-22 所示。

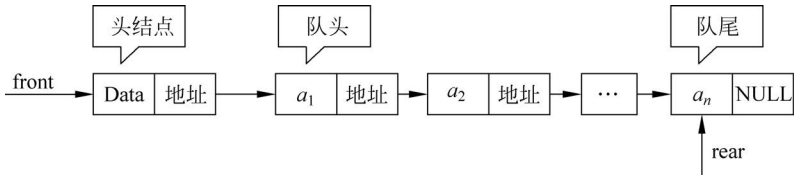


图 3-22　链队列

(1) 链队列的描述。

```
typedef struct node
{
   elemtype data;
   struct node *link;
```

```
} NODE, * NODEPTR;
typedef struct
{
   NODEPTR front, rear;
}LINKQUEUE;
```

(2) 创建空的链队列。

```
void InitQueue (LINKQUEUE * q)
{
   q -> front = NULL;
   q -> rear = NULL;
}
```

(3) 判断链列队是否为空。

```
int QueueEmpty (LINKQUEUE q)
{
   if (q.front == NULL) return(1);
   return(0);
}
```

(4) 链队列的入队。

```
void AddQueue (LINKQUEUE * q, elemtype e)
{
   NODEPTR p;
   p = (NODEPTR)malloc(sizeof(NODEPTR));
   p -> data = e; p -> link = NULL;
   if (q -> front == NULL) { q -> front = p; q -> rear = p; }
   else { q -> rear -> link = p; q -> rear = p; }
}
```

(5) 链队列的出队。

```
int DelQueue (LINKQUEUE * q, elemtype * e)
{
   NODEPTR p;
   if (q -> front == NULL) return(0);
   * e = q -> front -> data;
   p = q -> front; q -> front = p -> link;
   free(p);
   if (q -> front == NULL) q -> rear = NULL;
   return(1);
}
```

3.2.6　栈和队列的应用

栈在计算机中的应用非常广泛，许多程序语言本身就是建立于栈结构之上的，例如，在编译和运行程序的过程中，需要利用堆栈进行语法检查（如检查括号是否配对）、表达式求值、实现递归算法、数制转换与函数调用等。

队列在计算机及其网络自身内部的各种计算资源分配等场合有广泛的应用，例如，共享打印机、消息队列和广度优先搜索等，都需要借助队列结构实现合理和优化的分配。

1. 括号匹配的检验

假设表达式中包含 3 种括号：圆括号、方括号和花括号，其嵌套顺序随意，即“{([]())}”等为正确的格式，“[()]}”“([()]”或“(()}”均为不正确的格式。检验括号是否匹配可以用栈来实现：当遇到“(”“[”或“{”时进栈，遇到“}”“]”或“)”时出栈并进行匹配检验，如果出现不匹配的情况立即结束，否则继续取下一个字符。如果没有遇到不匹配的情况，最后判断栈是否为空：栈为空，括号匹配；否则不匹配。在算法的开始和结束时，栈都应该是空的。

2. 数制转换

十进制数 N 和其他 d 进制数的转换是计算机实现计算的基本问题，通常采用“除以基数取余数”方法，依次对除以基数得到的商再次求余数，这样得到的为待转换进制数从低位到高位的值，当商为 0 时转换完毕。

具体实现时使用栈暂时存放每次计算得到的余数，当算法结束时（也就是商为 0 时），从栈顶到栈底就是转换后从高位到低位的数值。

3. 表达式求值

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

一个程序设计语言应该允许设计者根据需要用表达式描述计算过程，编译器则应该能分析表达式并计算出结果。表达式的要素是运算符、操作数、界定符、算符优先级关系。例如，1＋2＊3 有“＋”“＊”两个运算符，“＊”的优先级高于“＋”，1、2、3 是操作数。界定符有括号和表达式结束符等。

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

（1）首先置操作数栈为空栈，表达式起始符“＃”为运算符栈的栈底元素。

（2）依次将表达式中的每个字符进栈，若是操作数，则进 OPND 栈；若是运算符，则和 OPTR 栈的栈顶运算符比较优先级后做相应操作，直至整个表达式求值完毕（即 OPTR 栈的栈顶元素和当前读入的字符均为“＃”）。

4. 迷宫求解

求迷宫中从入口到出口的所有路径是一个经典的程序设计问题。用计算机求解迷宫的方法是从入口出发，顺着某一方向向前探索，若能走通，则继续往前走，否则沿原路退回，换一个方向再继续探索，直到所有的通路都探索到为止。

迷宫问题求解算法中，一般将迷宫建模成图，将迷宫中的点建模为图中的点，将迷宫中相连并且相通的两点建模为图中的一条边，采用矩阵方式存储图，使用一个栈来存储访问过的顶点信息，栈中元素（即顶点信息）由顶点位置和搜索方向两部分组成，前者记载该顶点在迷宫中的位置，后者记载下一个顶点的访问方向，例如，右、下、左、上四个相连的方向。

求一条路径算法的基本思想是：假设以栈 S 记录当前路径，则栈顶中存放的是“当前路径上最后一个通道块”。

（1）若当前路径可通，则纳入“当前路径”——当前位置入栈操作，并继续朝下一个位置“探索”，即切换下一位置为当前位置，如此重复直至到达出口。

（2）若当前位置不可通，则应该顺着“来向”退回到前一通道块，然后朝着除来向之外的

其他方向继续探索。

(3) 若该通道的 4 个方向均不可走通，则应该从“当前路径”上删除该通道块——出栈操作。由于用计算机解决迷宫问题时，通常用的是“穷举求解”的方法，即从入口出发，顺着某一个方向向前探索，若能走通，则继续往前走；否则沿原路退回，换另一个方向再继续探索，直至所有可能的通路都探索完为止。为了保证在任何位置上都能沿原路退回，显然需要一个后进先出的结构保存入口到当前位置的路径。因此，在求解迷宫通路的算法中应用“栈”也就是自然而然的事了。

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

5. 共享打印机

目前，打印机提供的网络共享打印功能采用了缓冲池技术，队列就是实现这个缓冲池技术的数据结构支持。每台打印机具有一个队列(缓冲池)，用户提交打印请求被写入队列尾，当打印机空闲时，系统读取队列中第一个请求，打印并删除它。这样，利用队列的先进先出特性，就可完成打印机网络共享的先来先服务功能。

6. 消息队列

操作系统中的消息队列也是队列的应用之一，消息队列遵循先进先出的原则，发送进程将消息写入队列尾，接收进程则从队列头读取消息。

3.3　树

树是以分支关系定义的层次结构，是一类非常重要的非线性结构，其中以二叉树最为常用。用树结构描述的信息模型在客观世界普遍存在，在计算机科学及软件工程中应用十分广泛。

3.3.1　常用术语

树是 $n(n\geqslant 0)$个结点的有限集，在任意一棵非空树中存在以下特性。

(1) 有且仅有一个被称为根的结点。

(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，…，T_m，其中每一个集合本身又是一棵树，称为根的子树(递归定义)。

图 3-23 是一棵树的示意图，具有 13 个结点，A 为根结点，它有子树 B、C、D。

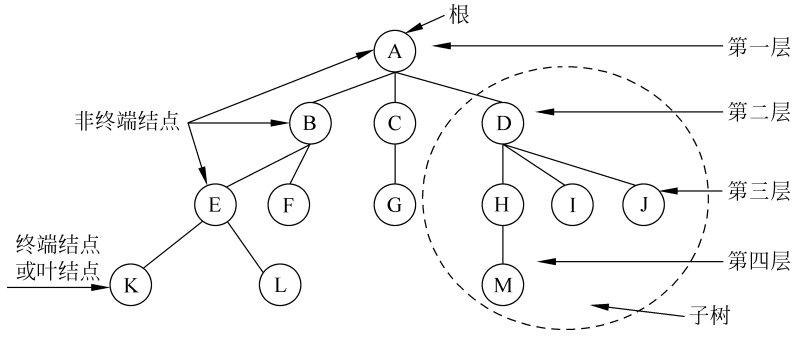


图 3-23　树的结构

关于树的常用术语如下。

（1）结点的度。每个结点的子树个数。

（2）叶子。叶子又称终端结点，是指度为 0 的结点。

（3）树的度。树中所有结点的度的最大值。

（4）结点的层次。规定根为第一层，其下面的一层为第二层，以此类推。

（5）树的深度。树中结点的最大层次数。

（6）孩子。一个结点的子树的根结点称为此结点的孩子。

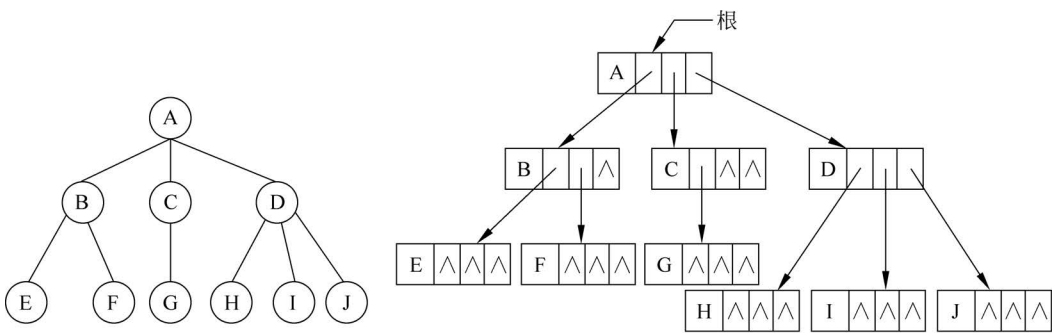
（7）双亲。若结点 1 是结点 2 的孩子，则结点 2 就被称为是结点 1 的双亲。

（8）兄弟。同一双亲的孩子之间互称兄弟。

（9）有序树。树中每个结点的各个子树从左到右依次有序（即不能互换）。

（10）森林。由 $m(m\geqslant 0)$棵互不相交的树构成的集合。

树的存储结构一般用具有多个指针域的多重链表来表示，结点中指针域的个数由树的度来决定。图 3-24(a)中树的存储结构如图 3-24(b)所示。由于树的度为 3，因此树中每个结点各具有 1 个数据域和 3 个指针域。



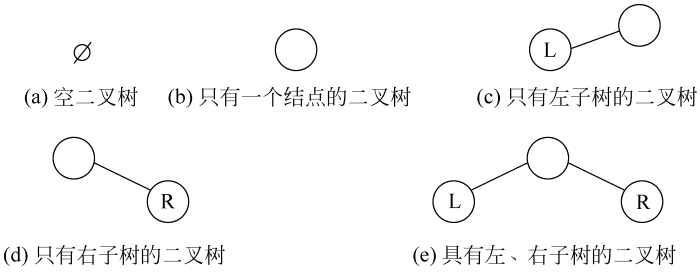
(a) 树形结构　　(b) 树的存储结构

图 3-24　树及其存储结构示意图

3.3.2　二叉树

1. 二叉树的定义

二叉树是度为 2 的有序树，即每个结点最多有两棵子树，并且二叉树的子树有左右之分，次序不能任意颠倒，即使在结点只有一棵子树的情况下也要明确指出该子树是左子树还是右子树。图 3-25 给出了二叉树的 5 种基本形态。



(a) 空二叉树　(b) 只有一个结点的二叉树　(c) 只有左子树的二叉树

(d) 只有右子树的二叉树　　(e) 具有左、右子树的二叉树

图 3-25　二叉树的 5 种基本形态

和树结构的定义类似，二叉树的定义也可以用递归形式给出。

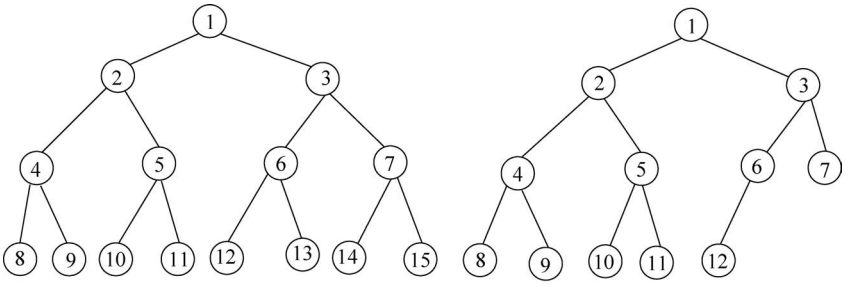
2. 二叉树的性质

性质 1：在二叉树的第 i 层上至多有 2^{i-1} 个结点($i \geqslant 1$)。

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

性质 3：对任何一棵二叉树 T，如果其终端结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

满二叉树和完全二叉树是两种特殊形态的二叉树。满二叉树是深度为 k 且具有 $2k-1$ 个结点的二叉树。如果一棵具有 n 个结点的深度为 k 的二叉树，它的每一个结点都与深度为 k 的满二叉树中编号为 1～n 的结点一一对应，则称这棵二叉树为完全二叉树，如图 3-26 所示。



(a) 满二叉树　　(b) 完全二叉树

图 3-26　两种特殊形态的二叉树

若一棵二叉树中每个结点的左、右子树的深度之差(平衡因子)均不大于 1，则称其为平衡二叉树。满二叉数、完全二叉树一定是平衡二叉树。完全二叉树具有如下性质。

性质 4：具有 n 个结点的完全二叉树的深度为 $\lfloor \log 2n \rfloor +1$，即以 2 为底 n 的对数下取整加 1。

性质 5：如果对一棵有 n 个结点的完全二叉树的结点按层序编号，则对任一结点 i($1 \leqslant i \leqslant n$)有以下几种结论。

(1) 如果 $i=1$，则结点 i 是二叉树的根，无双亲；如果 $i>1$，则双亲 PARENT(i)是结点 $i/2$。

(2) 如果 $2i>n$，则结点 i 无左孩子(结点 i 为叶子结点)；否则其左孩子 LCHILD(i)是结点 $2i$。

(3) 如果 $2i+1>n$，则结点 i 无右孩子；否则其右孩子 RCHILD(i)是结点 $2i+1$。

3. 二叉树的存储结构

二叉树可以用顺序存储结构和链式存储结构两种存储结构。

(1) 顺序存储结构。

按照顺序存储结构的定义，将一棵二叉树按完全二叉树顺序依次自上而下、自左至右存放到一个一维数组中。若该二叉树为非完全二叉树，则将相应位置空出来，使存放的结果符合完全二叉树的形状。例如，如图 3-27 所示的二叉树的顺序存储结构如表 3-1 所示。

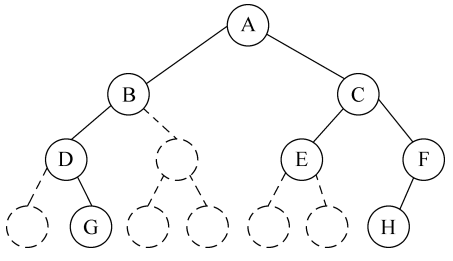


图 3-27　二叉树

表 3-1　二叉树的顺序存储结构

结点编号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
结点值	A	B	C	D	0	E	F	0	G	0	0	0	0	H	0

在顺序存储结构中，第 i 个结点的左、右孩子一定保存在第 $2i$ 及 $2i+1$ 个单元中。顺序存储的优点是容易理解，缺点是对非完全二叉树而言，大量空结点浪费存储空间。

二叉树顺序存储结构为：

```
#define MAX_TREE_SIZE 100
typedef TElemType SqBiTree[MAX_TREE_SIZE];          //TElemType 为结点数据类型
SqBiTree bt;
```

（2）链式存储结构。

在一般情况下，常用链式存储结构表示二叉树。由二叉树的定义可知，一个二叉树的结点至少保存 3 种信息：数据元素、左孩子位置、右孩子位置。对应地，链式存储二叉树的结点至少包含 3 个域：数据域、左指针域、右指针域，如图 3-28 所示。

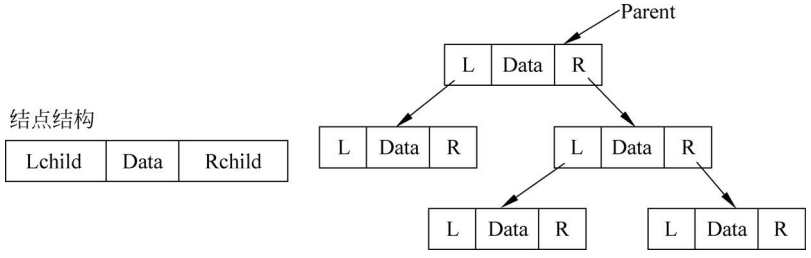


图 3-28　含有两个指针域的二叉树结点及其存储结构

二叉树链式存储结构为：

```
typedef struct treenode {
   elemtype data;
   struct treenode * lchild, * rchild;
} TREENODE, * TREENODEPTR, * BTREE
```

3.3.3　森林、树与二叉树的转换

森林是 $m(m\geqslant 0)$棵互不相交的树的集合。可以说树是森林的特例，二叉树又是树的特例。通过一定规则，森林和树可以转换为二叉树。数据结构中一般着重研究二叉树的相关性质及处理方法，通过转换，使用二叉树的一些算法去解决树和森林中的问题。

1. 树转换为二叉树

树转换为二叉树的步骤如下。

（1）在原树所有兄弟结点之间加一连线。

（2）对每个结点，除保留与其长子间的连线外，将该结点与其余孩子间的连线全部删除。

（3）以根结点为轴心，顺时针旋转 45°。

图 3-29 示范了将树转换为二叉树的步骤。

2. 森林转换为二叉树

森林转换为二叉树的步骤如下。

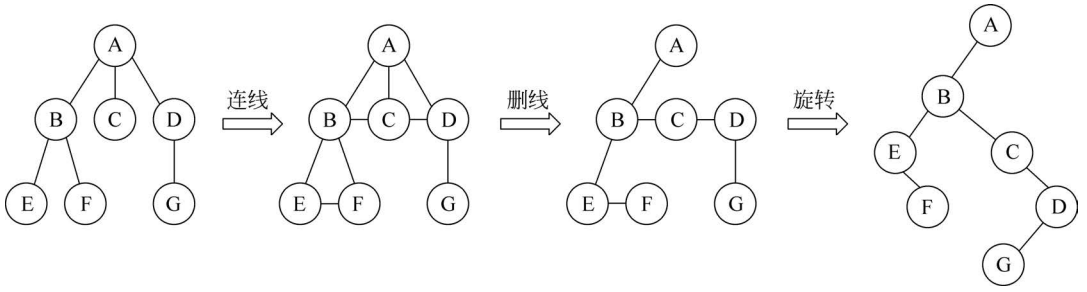


图 3-29　树转换为二叉树

(1) 在森林中所有树的根结点之间加一连线。

(2) 将森林中的每棵树转换成相应的二叉树。

(3) 以第一棵树的根结点为轴心,顺时针旋转 45°。

图 3-30 示范了将森林转换为二叉树的步骤。

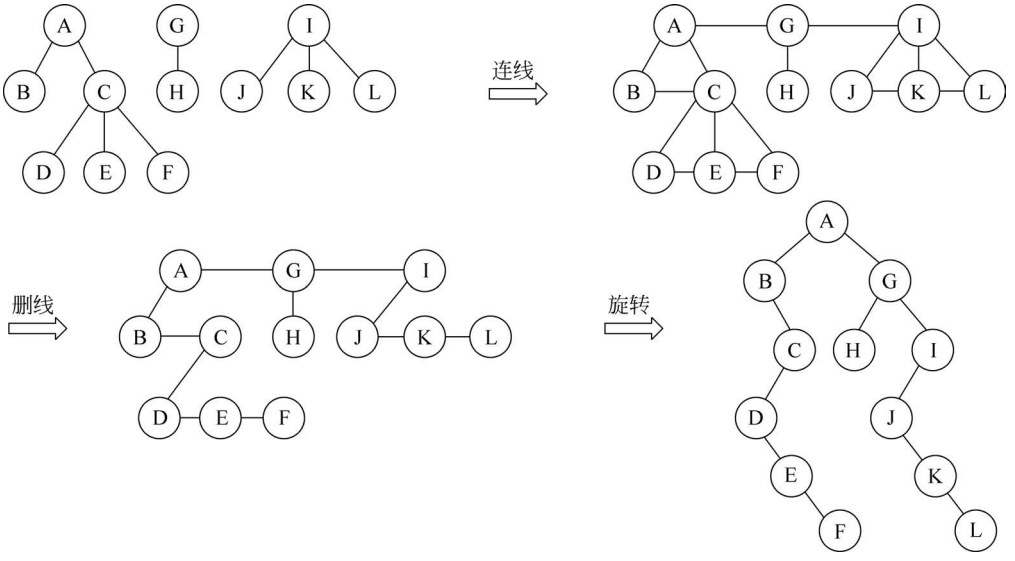


图 3-30　森林转换为二叉树

同理也可以将一个二叉树转换为对应的树或森林。

3.3.4　树的应用举例

树结构广泛应用于分类、检索、数据库及人工智能等多个方面。例如哈夫曼树常用于通信及数据传送中构造传送效率最高的二进制编码(哈夫曼编码)以及用于编程中构造平均执行时间最短的最佳判断过程等。

1. 哈夫曼树的有关概念

哈夫曼树又称为最优二叉树,是一类带权路径最短的树。设有 n 个权值$\{w_1, w_2, \cdots, w_n\}$,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,$1<i<n$,则带权路径长度最小的二叉树称作哈夫曼树。

(1) 结点间的路径长度。从树中一个结点到另一个结点之间的分支个数。

(2) 树的路径长度。从树的根结点到每一个结点的路径长度之和,记作 PL。例如,

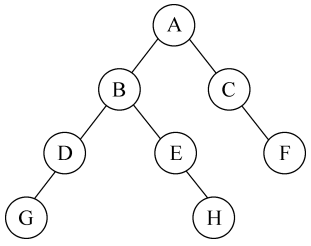


图 3-31　树的路径

图 3-31 中，结点 A 到 H 的路径长度为 3，树的路径长度为

$$PL=0+1+1+2+2+2+3+3=14$$

（3）结点的带权路径长度。从该结点到根结点之间的路径长度与结点上权值的乘积。

（4）树的带权路径长度。树中所有带权结点的路径长度之和，常记作 WPL。例如，图 3-32(a)所示的树中各结点带权路径长度为

$$A—10,B—9,C—21,D—2$$

a、b、c 三棵树的带权路径长度分别为

$$WPL=10+9+21+2=42$$

$$WPL=10+6+14+4=34$$

$$WPL=7+10+9+6=32$$

可见相同结点位于数中不同位置构成的树，其 WPL 则不同，哈夫曼树即为 WPL 值最小的树，图 3-32(c)为哈夫曼树。

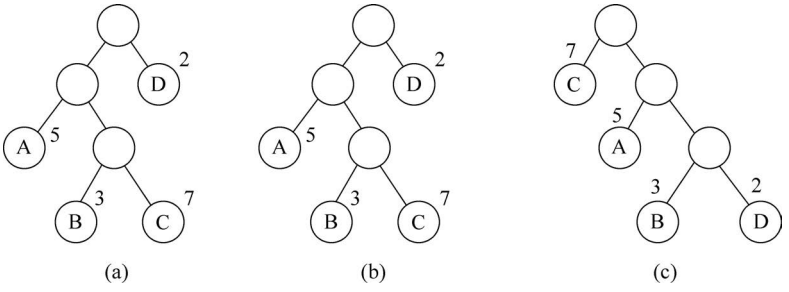


图 3-32　带权值的树

2. 哈夫曼树的构造

哈夫曼树的构造步骤如下。

（1）对于给定的 n 个权值$\{w_1,w_2,\cdots,w_n\}$，构造出具有 n 棵二叉树的森林 $F=\{T_1,T_2,\cdots,T_n\}$，其中每棵二叉树 T_i 均只有一个带有权值 w_i 的根结点。

（2）在 F 中选取根结点权值最小的两棵二叉树作为左、右子树构造一棵新的二叉树，新二叉树根结点的权值为其左、右子树根结点的权值之和。

（3）在 F 中删除这两棵树，同时将新生成的二叉树加入 F 中。

（4）重复步骤(2)和步骤(3)，直到 F 只有一棵二叉树为止，这棵二叉树就是所构造的哈夫曼树。

例如，一组结点权值为{5,3,7,2}，构造哈夫曼树的过程如图 3-33 所示。

3. 哈夫曼树的应用

（1）解决某些判定问题时的最佳判定算法。

例 3.3　针对 10 000 个学生成绩数据，按分数段分级统计。

若学生成绩分布是均匀的，如图 3-34 所示构造程序流程，读入一个 a 值平均判断：(1+2+3+4+4)×0.2=2.8(次)；输入 10 000 个数据，则需进行 28 000 次比较。

实际情况中，成绩通常是不均匀分布的，如表 3-2 所示。成绩主要集中在 70～79 分，

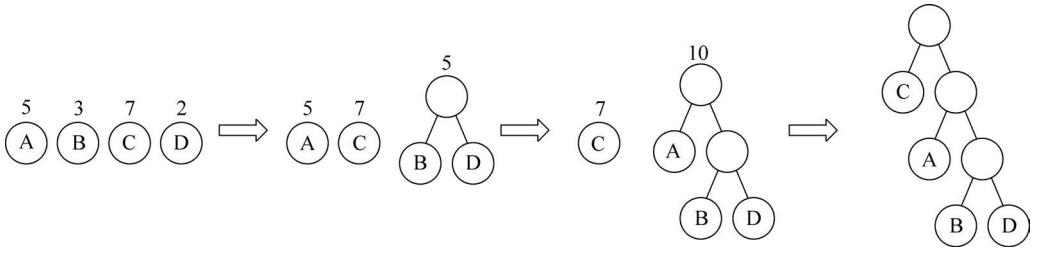


图 3-33　哈夫曼树的构造过程

80～89 分两个分数段，则需根据各分数段的分布权值构造一棵哈夫曼树，如图 3-35 所示，依此编写程序结构，输入 10 000 个数据，只需进行 20 500 次比较。

$$WPL = 0.4 \times 1 + 0.3 \times 2 + 0.15 \times 3 + (0.05 + 0.1) \times 4 = 2.05$$

表 3-2　成绩分布

分数	0～59 分	60～69 分	70～79 分	80～89 分	90～99 分
比例	0.05	0.15	0.4	0.3	0.1

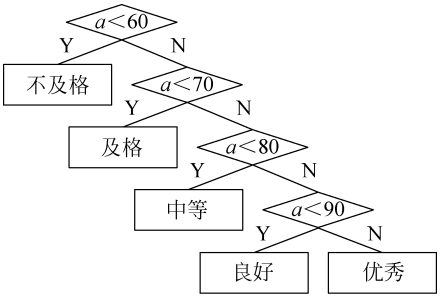


图 3-34　成绩均匀分布时的程序流程

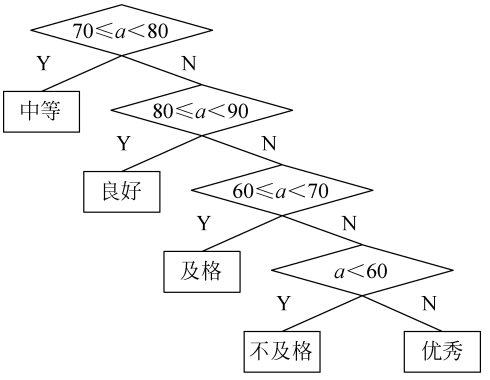


图 3-35　成绩非均匀分布时的统计哈夫曼树

(2) 哈夫曼编码。

哈夫曼编码是哈夫曼树在数据编码中的应用，即数据的最小冗余编码。作为一种最常用无损压缩编码方法，在数据压缩程序中具有非常重要的应用。

通信中，可以采用 0、1 的不同排列来表示不同的字符，称为二进制编码。若每个字符出现的频率相同，则可以采用等长的二进制编码，若频率不同，则可以采用不等长的二进编码。频率较大的采用位数较少的编码，频率较小的字符采用位数较多的编码，这样可以使字符的整体编码长度最小，这就是最小冗余编码的问题。

例如，如需传送字符串'ABACCDA'，只有 4 种字符 A、B、C、D，只需两位编码，如分别编码为 00、01、10、11，上述字符串的二进制总长度为 14 位。

在传送信息时，希望总长度尽可能短，可对每个字符进行不等长度的编码，出现频率高的字符编码尽量短。如 A、B、C、D 的编码分别为 0、00、1、01 时，上述电文长度会缩短，但可能有多种译法。

在设计不等长编码时还需注意，任一字符的编码都不能是另一个字符编码的前缀编码。例如，编码 0、00、1、01 就是前缀编码，如'0000'可能是'AAAA'、'ABA'、'BB'。

可利用二叉树来设计二进制的前缀编码，将每个字符出现的频率作为权，设计一棵哈夫曼树，左分支为0，右分支为1，就得到每个叶子结点的编码。

例 3.4 假设用于通信的电文仅由8个字母A、B、C、D、S、T、U、V组成，字母在电文中出现的频率分别为(5，29，7，8，14，23，3，11)，试为这8个字母设计哈夫曼编码。

将各字母的权值排列成结点，构造哈夫曼树，如图3-36所示，即可得到各字母的哈夫曼编码。

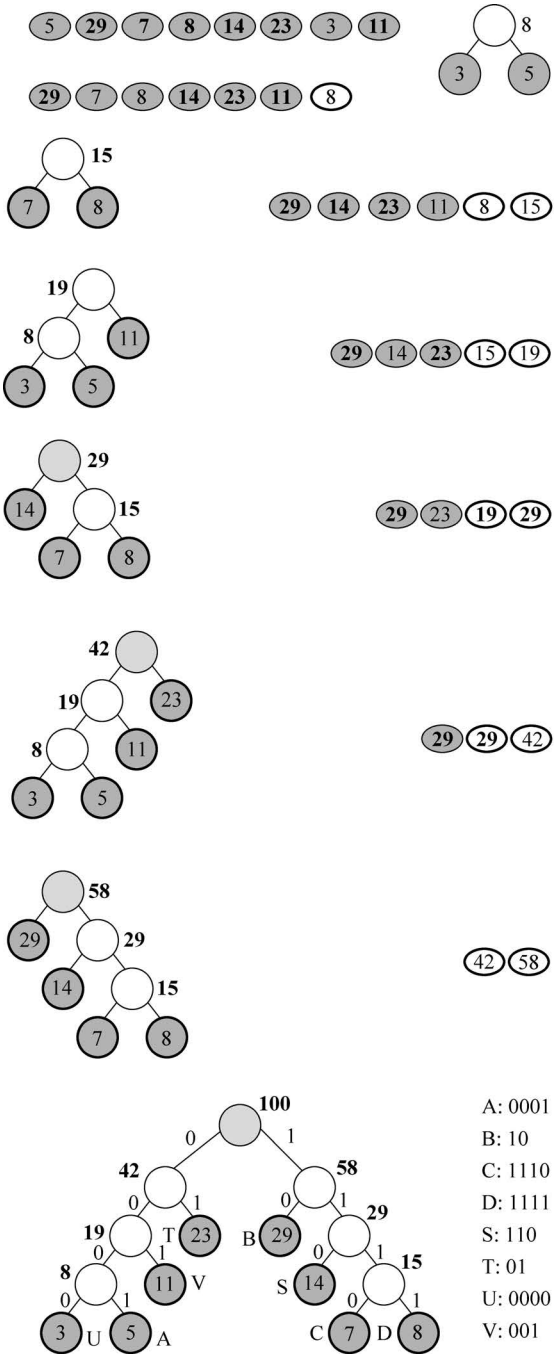


图3-36 哈夫曼树和哈夫曼编码

3.4　图

图是数据元素间为多对多关系的数据结构，在人工智能、工程、物理、化学、计算机科学等许多领域中，图结构被广泛应用。

3.4.1　常用术语

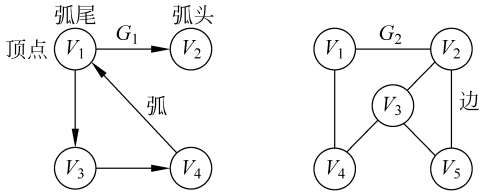
图是由顶点集 V 和顶点间的关系集合 E（边的集合）组成的一种数据结构，可以用二元组定义为

$$G=(V,E)$$

其中，V 是顶点的非空有穷集合；E 是可空的边的有穷集合。

若图中所有边的顶点对有序，则称有向图。有向图中的(V_1,V_2)和(V_2,V_1)代表不同的边，并分别用$\langle V_1,V_2\rangle$和$\langle V_2,V_1\rangle$表示，称为弧，对于弧$\langle V_1,V_2\rangle$，顶点 V_1 称为弧尾，V_2 称为弧头，如图 3-37(a)所示。

若图中所有边的顶点对无序，则称无向图。无向图中(V_1,V_2)和(V_2,V_1)代表相同的边，如图 3-37(b)所示。



(a) 有向图　　(b) 无向图

图 3-37　有向图和无向图示例

对于图 3-37(a)，有 $G_1=(V',\{A'\})$，其中 $V'=\{V_1,V_2,V_3,V_4\}$，$A'=\{\langle V_1,V_2\rangle,\langle V_1,V_3\rangle,\langle V_3,V_4\rangle,\langle V_4,V_1\rangle\}$。

图的常用术语如下。

(1) 顶点。数据元素 v_i 称为顶点。

(2) 边、弧。$P(v_i,v_j)$表示顶点 v_i 和顶点 v_j 之间的直接连线，在无向图中称为边，在有向图中称为弧。带箭头的一端称为弧头，不带箭头的一端称为弧尾。

(3) 如果用 n 表示图中顶点的数目，用 e 表示边或弧的数目，则无向图 e 取值是 $0\sim n(n-1)/2$，有向图的 e 取值是 $0\sim n(n-1)$。

(4) 完全图。完全图是指有 $n(n-1)/2$ 条边的无向图。

(5) 有向完全图。有向完全图是指有 $n(n-1)$条弧的有向图。

(6) 稀疏图。稀疏图是指有很少条边或弧的图。

(7) 稠密图。稠密图是指有很多条边或弧的图。

(8) 度。在图中，一个顶点依附的边或弧的数目称为该顶点的度。在有向图中，一个顶点依附的弧头数目称为该顶点的入度。一个顶点依附的弧尾数目称为该顶点的出度，该顶点的度等于入度和出度之和。

(9) 子图。若有两个图 G_1 和 G_2，$G_1=(V_1,E_1)$，$G_2=(V_2,E_2)$，满足如下条件：$V_2\subseteq$

V_1，$E_2 \subseteq E_1$，即 V_2 为 V_1 的子集，E_2 为 E_1 的子集，称图 G_2 为图 G_1 的子图。

（10）权和网。在图的边或弧中给出相关的数称为权。权可以代表一个顶点到另一个顶点的距离或耗费等，带权图一般称为网。

（11）连通图和非连通图。在无向图中，若从顶点 i 到顶点 j 有路径，则称顶点 i 和顶点 j 是连通的。若任意两个顶点都是连通的，则称此无向图为连通图，否则称为非连通图。

（12）连通分量。在无向图中，图的极大连通子图称为该图的连通分量。任何连通图的连通分量只有一个，即它本身，而非连通图有多个连通分量。

（13）强连通图和非强连通图。在有向图中，若从顶点 i 到顶点 j 有路径，则称顶点 i 和顶点 j 是连通的，若图中任意两个顶点都是连通的，则称此有向图为强连通图，否则称为非强连通图。

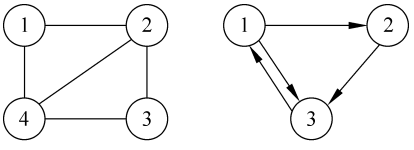
3.4.2 图的存储结构

图的存储结构有邻接矩阵、邻接表、逆邻接表以及十字链表等，其中邻接矩阵和邻接表最常用。

1. 邻接矩阵

若 $G=(V,E)$ 是一个具有 $n(n \geqslant 1)$ 个结点的图，则 G 的邻接矩阵是一个 n 阶方阵。在邻接矩阵表示中，除了存放顶点本身信息外，还用一个 n 阶方阵表示各个顶点之间的关系。方阵中只有 0 和 1 元素。若顶点 V_j 邻接于 V_i 时（从 V_i 到 V_j 有边相连），则矩阵中第 i 行第 j 列元素值为 1，否则为 0。显然，图的邻接矩阵很容易用顺序存储方式的数组存储。

图 3-38 所示无向图和有向图的邻接矩阵如图 3-39 所示。



(a) 无向图G_1　　(b) 有向图G_2

图 3-38　无向图 G_1 与有向图 G_2

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

(a) G_1的邻接矩阵　(b) G_2的邻接矩阵

图 3-39　邻接矩阵表示

邻接矩阵有如下特点。

（1）对于无向图邻接矩阵。

- 矩阵是对称的。
- 第 i 行或第 i 列 1 的个数为顶点 i 的度。
- 矩阵中值为 1 的个数的一半为图中边的数目。
- 很容易判断顶点 i 和顶点 j 之间是否有边相连（看矩阵中第 i 行 j 列的值是否为 1）。

（2）对于有向图邻接矩阵。

- 矩阵不一定是对称的。
- 第 i 行中 1 的个数为顶点 i 的出度。
- 第 i 列中 1 的个数为顶点 i 的入度。
- 矩阵中 1 的个数为图中弧的数目。
- 很容易判断顶点 i 和顶点 j 是否有弧相连。

图的邻接矩阵可以定义两个数组分别存储顶点信息(数据元素)和边或弧的信息(数据元素之间的关系),其存储结构形式定义如下:

在用邻接矩阵存储图时,除了用一个二维数组存储用于表示顶点间相邻关系的邻接矩阵外,还需用一个一维数组来存储顶点信息,还有图的顶点数、边数和权值等信息。不同类型的图定义稍有不同,例如图 3-38(a)中图的描述如下:

```
#define maxvertexnum 4                          //最大顶点数设为 4
typedef int vertextype;                         //顶点类型设为整型
typedef int edgetype;                           //边的权值设为整型
typedef struct
{
   vertextype vexs[maxvertexnum];               //顶点表
   edgeType edges[maxvertexnum][maxvertexnum];  //邻接矩阵,即边表
   int n,e;                                     //顶点数和边数
}Mgragh;                                        //Mgragh 是以邻接矩阵存储的图类型
```

2. 邻接表

邻接表是图的链式存储结构。若将每个顶点的边用一个单链表链接起来,若干个顶点可以得到若干个单链表,每个单链表都有一个头结点,所有头结点组成一个一维数组,这样的链表为邻接链表或邻接表。

在邻接表中,对图中每个顶点建立一个单链表,第 i 个单链表中的结点表示依附于顶点 V_i 的边(对有向图是以顶点 V_i 为尾的弧。若是以顶点 V_i 为头的弧,则称逆邻接表)。

邻接表中头结点及表结点结构如图 3-40 和图 3-41 所示。

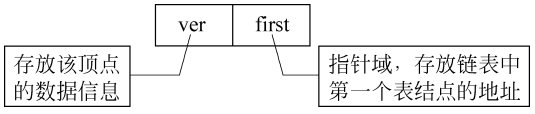


图 3-40　头结点结构

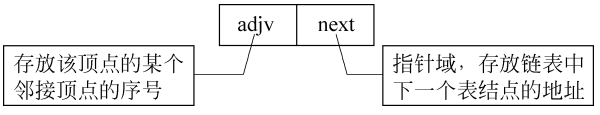
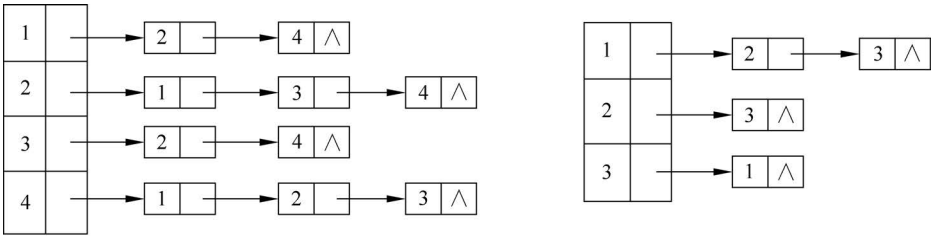


图 3-41　表结点结构

图 3-38 所示无向图和有向图的邻接表如图 3-42 所示。



(a) 无向图G_1的邻接表　　(b) 有向图G_2的邻接表

图 3-42　图的邻接表

从邻接表图示中可以得出如下结论。

（1）对于无向图的邻接表。

- 第 i 个链表中结点数目为顶点 i 的度。
- 所有链表中结点数目的一半为图中边数。
- 占用的存储单元数目为 $n+2e$。

（2）对于有向图的邻接表。

- 第 i 个链表中结点数目为顶点 i 的出度。
- 所有链表中结点数目为图中弧数。
- 占用的存储单元数目为 $n+e$。

邻接表的相关定义函数如下：

```
#define MAXVEX 100                          //最大顶点数,由用户定义
typedef char VertexType;                    //顶点类型
typedef int EdgeType;                       //边上的权值,类型由用户定义

typedef struct EdgeNode                     //边表结点
{
   int adjvex;                              //邻接点域,存储该顶点对应的下标
   EdgeType weight;                         //用于存储权值,对于非网图可以不需要
   struct EdgeNode *next;                   //链域,指向下一个邻接点
} EdgeNode;

typedef struct VextexNode                   //顶点表结点
{
   VertexType data;                         //顶点域,存储顶点信息
   EdgeNode *firstedge;                     //边表头指针
} VextexNode, AdjList[MAXVEX];

typedef struct
{
   AdjList adjList;
   int numNodes, numEdges;                  //图中当前顶点数和边数
} GraphAdjList;
```

图的邻接矩阵存储易于求顶点度（区分有/无向图）和邻接点，易判断两点间是否有弧或边相连，但不利于稀疏图的存储，因弧不存在时也要存储相应信息，且要预先分配足够大的空间。

图的邻接表存储对于稀疏图可相对节省空间，对有向图易求顶点出度与邻接点，但求入度难度较大。若只求入度可引入逆邻接表，也可结合邻接表与逆邻接表引入十字链表，对无向图易求度，但边出现两次，为方便边操作可借助多重链表。

3.4.3 图的应用举例

最短路径问题是图的一个典型的应用。例如，某一地区的一个公路网，给定了该网内的 n 个城市以及这些城市之间的相通公路的距离，能否找到城市 A 至城市 B 之间一条距离最近的通路呢？

如果将城市用点表示，城市间的公路用边表示，公路的长度作为边的权值，那么，这个问题就可归结为在图中，求点 A 到点 B 的所有路径中边的权值之和最短的那一条路径。这条路径就是两点之间的最短路径，并称路径上的第一个顶点为源点，最后一个顶点为终点。此

类问题称为单源点的最短路径问题，即给定带权有向图 $G=(V,E)$ 和源点 $v\in V$，求从 v 到 G 中其余各顶点的最短路径。

迪杰斯特拉算法是解决有向图中最短路径问题常用的解决方法。迪杰斯特拉算法的主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。首先求出长度最短的一条最短路径，然后参照它求出长度次短的一条最短路径，以此类推，直到从顶点 v 到其他各顶点的最短路径全部求出为止。

算法的基本思想如下。

(1) 设置两个顶点的集合 S 和 $T=V-S$，集合 S 中存放已找到最短路径的顶点，集合 T 存放当前还未找到最短路径的顶点。

(2) 初始状态时，集合 S 中只包含源点 v_0，然后不断从集合 T 中选取到顶点 v_0 路径长度最短的顶点 u 加入集合 S 中，集合 S 每加入一个新的顶点 u，都要修改顶点 v_0 到集合 T 中剩余顶点的最短路径长度值，集合 T 中各顶点新的最短路径长度值为原来的最短路径长度值与顶点 u 的最短路径长度值加上 u 到该顶点路径长度值中的较小值。此过程不断重复，直到集合 T 的顶点全部加入 S 中为止。

算法步骤如下。

(1) 初始化源点 A 到其他顶点的距离，若其他顶点与源点 A 无直接相连的边，则认为源点 A 到该顶点的距离为无穷大。

(2) 选择当前距离源点 A 最近的顶点 X(顶点 X 未被选择过)。

(3) 以 X 点为参照，更新源点 A 到其他未被选择过的点 M 的距离，若 A→X→M 小于 A→M 的距离，则使用新距离替换原距离；若 A→X→M 大于或等于 A→M 距离，则保持原距离不变。

(4) 重复步骤(2)、步骤(3)，直到选取完所有的点为止。