第 3 章

CHAPTER 3

非线性数据结构

第 2 章介绍的是线性数据结构，线性数据结构中的每个元素有唯一的前驱元素和唯一的后继元素，即前驱元素和后继元素是一对一的关系。本章将介绍非线性数据结构，包括树和图，树的元素有唯一的前驱元素和多个后继元素，即前驱元素和后继元素是一对多的关系。树形结构是非常重要的一种数据结构，在实际应用中也非常广泛，它主要应用在文件系统、目录组织等大量的数据处理中。图是另一种非线性数据结构，是一种更为复杂的数据结构。在图中，数据元素之间是多对多的关系，即一个数据元素对应多个直接前驱元素和多个直接后继元素。图的应用领域十分广泛，例如工程设计、遗传学、人工智能等。

3.1 树的概念

树是由 n（n ≥ 0）个结点组成的有限集合。如果 n = 0，称为空树；如果 n > 0，则有且仅有一个特定的称之为根（Root）的结点，它只有直接后继，但没有直接前驱；当 n > 1，除根以外的其他结点划分为 m（m >0）个互不相交的有限集合 T1，T2，…，Tm，其中每个集合本身又是一棵树，并且称为根的子树（SubTree）。如图 3-1 所示。

T = { A,B,C,D,E,F,G,H,I,J }，其中 A 是根，其余结点可以划分为两个互不相交的集合，T1= { B,D,G,H,I }，T2 ={ C,E, F,G }，这两个集合本身又各是一棵树，它们是 A 的子树。对于 T2，C 是根，其余结点可以划分为两个互不相交的集合，T21 ={ E,J }，T22 ={ F }，T21，T22 是 C 的子树。

树的结构定义是一个递归的定义，即在树的定义中又用到了树的概念，它道出了树的固有特性。如图 3-2 中的子树 T1 和 T2 就是根结点 A 的子树，D,G,H,I 组成的树又是 B 结点的子树，E,J 组成的树是 C 结点的子树。

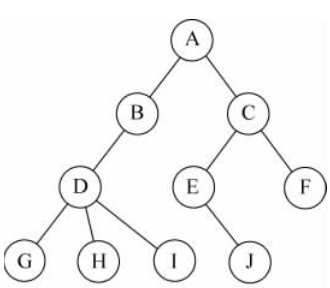


图 3-1　一般的树

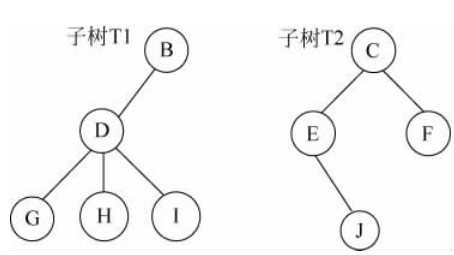


图 3-2　两棵子树

对于树的定义还需要注意两点。

(1) n>0 时,根结点是唯一的,不可能存在多个根结点。

(2) m>0 时,子树的个数没有限制,但它们一定是互不相交的。如图 3-3 中的两个结构就不符合树的定义。

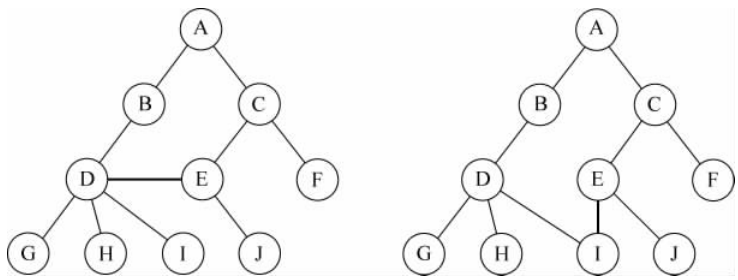


图 3-3 非树的表示

树的逻辑表示方法可以分为 4 种,分别是树形表示法、文氏图表示法、广义表表示法和凹入表示法。

(1) 树形表示法。图 3-1 就是树形表示法。树形表示法是最常见的一种表示方法,它可以直观、形象地表示出树的逻辑结构,可以清晰地反映出树中结点之间的逻辑关系。

(2) 文氏图表示法。文氏图表示法是利用数学中集合的图形化表示来描述树的逻辑关系。图 3-1 的树,可用文氏图表示如图 3-4 所示。

(3) 广义表表示法。采用广义表的形式表示树的逻辑结构,广义表的子表表示结点的子树。图 3-1 的树,利用广义表表示为(A(B(D(G,H,I)),C(E(J),F)))。

(4) 凹入表示法。凹入表示法类似于书的目录、章、节、条、款、项逐个凹入。图 3-1 的树,可用凹入表示法如图 3-5 所示。

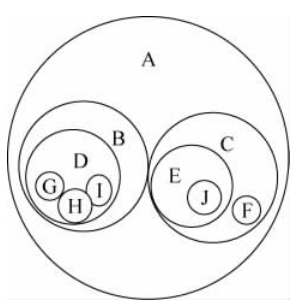


图 3-4 树的文氏图表示法

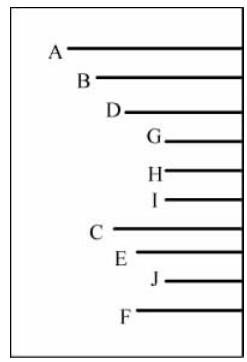


图 3-5 树的凹入表示法

下面列出树结构中的一些术语。

结点:树的结点包含一个数据元素及若干指向其子树的分支。

孩子结点:结点的子树的根称为该结点的孩子。如图 3-1 中 D 是 B 的孩子。

双亲结点:B 结点是 A 结点的孩子,则 A 结点是 B 结点的双亲。如图 3-1 中 B 是 D 的双亲。

兄弟结点：同一双亲的孩子结点。如图 3-1 中 G、H 和 I 互为兄弟。

堂兄结点：其双亲结点互为兄弟的结点。如图 3-1 中 D 的双亲结点 B 与 E 的父结点 C 互为兄弟，则称 D 与 E 互为堂兄。

祖先结点：结点的祖先是从根到该结点所经分支上的所有结点。如图 3-1 中 F 的祖先为 A、C。

子孙结点：以某结点为根的子树中的任一结点称为该结点的子孙。如图 3-1 中 A 的子孙为 B、C、D、E、F 等。

结点的度：结点拥有的子树的个数为结点的度。如图 3-1 中 A 结点的度为 2。

叶子：度为 0 的结点为叶子或者终端结点。如图 3-1 的 G，H，I，J。

树的度：树内各结点的度的最大值。如图 3-1 树的度为 3。

层次：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，以此类推。树中结点的最大层次为树的深度或高度，如图 3-1 所示的树的深度为 4。

若将树中每个结点的各子树看成是从左到右有次序的(即不能互换)，则称该树为有序树(OrderedTree)；否则称为无序树(UnorderedTree)。若不特别指明，一般讨论的树都是有序树。

森林(Forest)是 m(m≥0)棵互不相交的树的集合。对树中每个结点，其子树的集合即为森林。树和森林的概念相近。删去一棵树的根，就得到一个森林；反之，加上一个结点作树根，森林就变为一棵树。

树的基本操作主要有

(1) 初始化操作 Initate(T)：创建一棵空树 T。

(2) 销毁树操作 Destory(T)：销毁树 T。

(3) 构造树操作 Creat(T，definition)：按 definition 构造树 T，definition 给出树的定义。

(4) 清空树操作 Clear(T)：将树 T 清为空树。

(5) 求根函数 Root(T)：求树 T 的根。

(6) 插入操作 Insert(T，x，i，y)：将以 y 为根的子树插入到树 T 中作为结点 x 的第 i 棵子树。

(7) 删除操作 Delete(T，x，i)：将树 T 中结点 x 的第 i 棵子树删除。

(8) 遍历树操作 Traverse(T)：按某种次序对树 T 中的每个结点访问一次且仅一次。

(9) 求双亲操作 Parent(T，x)：若 x 是 T 的非根结点，则返回它的双亲，否则函数值返回“空”。

(10) 求右兄弟操作 RightSibling(T，X)：若 x 有右兄弟结点，则返回它的右兄弟，否则返回“空”。

(11) 求孩子操作 Child(T，x，i)：若 x 是 T 的非叶子结点，则返回它的第 i 个孩子，否则返回“空”。

3.2　二叉树

树结构中有一种应用较为广泛的树——二叉树(Binary Tree)。二叉树的度为 2，而且是一棵有序树，即树中结点的子树从左到右有次序。

3.2.1 二叉树的定义

二叉树是一个有限元素的集合，该集合或者为空、或者由一个称为根(root)的元素及两个不相交的、分别称为左子树和右子树的二叉树组成。

值得注意的是，度为 2 的有序树与二叉树是不同的。假设要在有两个孩子的结点中删除第一个孩子。在度为 2 的有序树中，删除了第一个孩子，原来第 2 个孩子改称为第 1 个孩子；二叉树中，如果删除了左孩子，右孩子依然是右孩子。

二叉树是另外一种树形结构，即不存在度大于 2 的结点，每个结点至多只有两棵子树，二叉树的子树有左右之分，左、右子树不能颠倒，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。另外，二叉树是递归结构，二叉树的定义中又用到了二叉树的概念。图 3-6 列出了二叉树的 5 种基本形态。

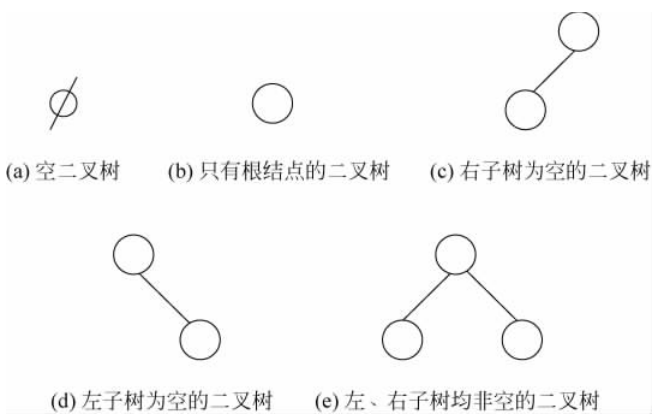


图 3-6 二叉树的 5 种基本状态

3.2.2 二叉树的主要性质

二叉树具有下列重要性质：

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

[证明用归纳法]

证明：当 i=1 时，只有根结点，$2^{i-1}=2^0=1$。

假设对所有 j，$1 \leqslant j < i$，命题成立，即第 j 层上至多有 2^{j-1} 个结点。

由归纳假设第 i−1 层上至多有 2^{i-2} 个结点。

由于二叉树的每个结点的度至多为 2，故在第 i 层上的最大结点数为第 i−1 层上的最大结点数的 2 倍，即 $2\times 2^{i-2}= 2^{i-1}$。

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

由性质 1 可见，深度为 k 的二叉树的最大结点数为

$$2^0+2^1+\cdots+2^{k-1}=2^k-1$$

(3) 性质 3：对任何一棵二叉树 T，如果叶结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0=n_2+1$。

证明：设二叉树中度为 1 的结点数为 n_1，二叉树中总结点数为 $n=n_0+n_1+n_2$

二叉树中的分支数，除根结点外，其余结点都有一个进入分支（分支进入），设 B 为二叉树中的分支总数，则有 n＝B＋1。由于这些分支都是由度为 1 和 2 的结点射出的，所以有 $B=n_1+2\times n_2$；$n=B+1=n_1+2\times n_2+1$ 得到 $n_0=n_2+1$。

完全二叉树和满二叉树是两种特殊形态的二叉树。

满二叉树指的是深度为 k 的二叉树并且有 2^k-1 个结点。这种树的特点是每一层上的结点数都达到最大结点数。如图 3-7(a)所示是一棵深度为 3 的满二叉树。

如果在深度为 k、有 n 个结点的二叉树中，各结点能够与深度为 k 的顺序编号的满二叉树从 1 到 n 标号的结点相对应，则称为完全二叉树。这种树的特点是所有的叶子结点都出现在第 k 层或 k－1 层。对任一结点，如果其右子树的最大层次为 L，则其左子树的最大层次为 L 或 L＋1。如图 3-7(b)所示的是一棵深度为 3 的完全二叉树，图 3-7(c)和(d)不是完全二叉树。

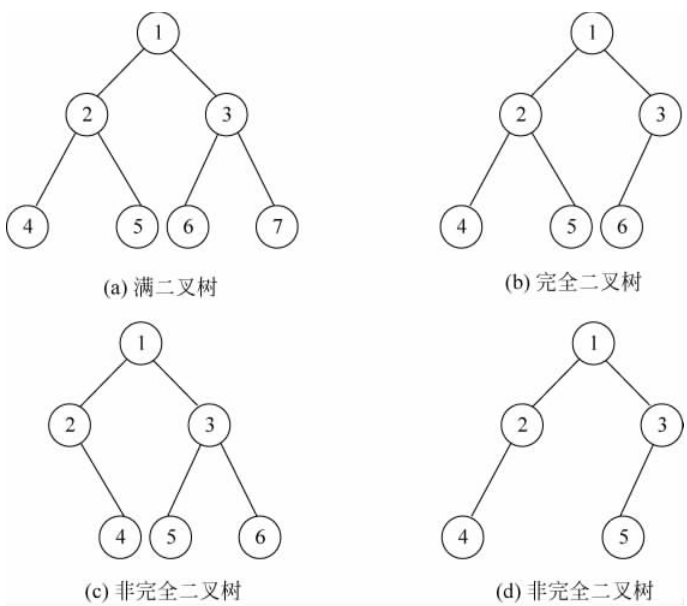


图 3-7　特殊形态的二叉树

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

证明：设完全二叉树的深度为 k，则根据性质 2 和完全二叉树的定义有 $2^{k-1}-1<n\leqslant 2^k-1$ 或 $2^{k-1}\leqslant n<2^k$。

取对数 $k-1<\log_2 n\leqslant k$，又 k 是整数，因此有 $k=\lfloor\log_2 n\rfloor+1$。

(5) 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（从第 1 层到第 $\lfloor\log_2 n\rfloor+1$ 层，每层从左到右），则对任一结点 i($1\leqslant i\leqslant n$)，有

① 如果 i＝1，则结点 i 无双亲，是二叉树的根；如果 i＞1，则其双亲是结点 $\lfloor i/2\rfloor$。

② 如果 2i＞n，则结点 i 为叶子结点，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，其右孩子是结点 2i＋1。

证明：此性质可采用数学归纳法证明。因为 1 与 2、3 是相对应的，所以只需证明 2 和 3。

当 i=1 时，根据结点编号方法可知，根的左、右孩子编号分别是 2 和 3，结论成立。假定 i−1 时结论成立，即结点 i−1 的左右孩子编号满足 lchild(i−1)=2(i−1)；rchild (i−1)=2(i−1)+1。通过完全二叉树可知，结点 i 或者与结点 i−1 同层且紧靠其右，或者结点 i−1 在某层最右端，而结点 i 在其下一层最左端。但是，无论如何，结点 i 的左孩子的编号都是紧接着结点 i−1 的右孩子的编号，故 lchild(i)=rchild(i−1)+1=2i；rchild(i)=lchild(i)+1=2i+1 命题成立。

3.2.3 二叉树的存储结构

二叉树的存储结构有两种，分别是顺序存储表示和链式存储表示。

1. 顺序存储结构

所谓顺序存储结构，就是用一组连续的存储单元存储二叉树的数据元素，结点在这个序列中的相互位置能反映出结点之间的逻辑关系。二叉树中结点之间的关系就是双亲结点与左右孩子结点间的关系。因此，必须把二叉树的所有结点安排成为一个恰当的序列。

C 语言中，这种存储形式的类型定义如下所示。

```
#define MaxTreeNodeNum 100          /* 二叉树的最大结点数 */
typedef struct {
  DataType data[MaxTreeNodeNum];  /* 0 号结点存放根结点 */
  int n;
} QBiTree;
```

通常，按照二叉树结点从上至下、从左到右的顺序存储，但这样结点在存储位置上的前驱后继关系并不一定就是它们在逻辑上的邻接关系。依据二叉树的性质，完全二叉树和满二叉树采用顺序存储比较合适，树中结点的序号可以唯一地反映出结点之间的逻辑关系，既能够最大可能地节省存储空间，又可以利用数组元素的下标值确定结点在二叉树中的位置，以及结点之间的关系。图 3-8(a)为图 3-7(b)所示的完全二叉树的顺序存储结构。对于一般的二叉树，则应将其每个结点与完全二叉树上的结点相对照，存储在一维数组的相应分量中，如图 3-7(c)所示二叉树的顺序存储结构如图 3-8(b)所示，图中以“0”表示不存在的结点。由此可见，这种顺序存储结构仅适用于完全二叉树。因为，在最坏的情况下，一个深度为 k 且只有 k 个结点的单支树(树中不存在度为 2 的结点)却需要长度为 2^k-1 的一维数组。

图 3-8 二叉树的顺序存储结构

完全二叉树的编号特点为除最下面一层外，各层都充满了结点。每一层的结点个数恰好是上一层结点个数的 2 倍。从一个结点的编号就可推得其双亲，左、右孩子，兄弟等结点

的编号。假设编号为 i 的结点是 K_i(1≤i≤n),则有

(1) 若 i>1,则 K_i 的双亲编号为⌊i/2⌋；若 i=1,则 K_i 是根结点,无双亲。

(2) 若 2i≤n,则 K_i 的左孩子编号为 2i；否则,K_i 无左孩子,即 K_i 必定是叶子。

(3) 若 2i+1≤n,则 K_i 的右孩子编号为 2i+1；否则,K_i 无右孩子。

(4) 若 i 为奇数且不为 1,则 K_i 的左兄弟编号为 i−1；否则,K_i 无左兄弟。

(5) 若 i 为偶数且小于 n,则 K_i 的右兄弟编号为 i+1；否则,K_i 无右兄弟。

顺序存储的优缺点是适合于完全二叉树,既不浪费存储空间,又能很快确定结点的存放位置,以及结点的双亲和左右孩子的存放位置。但对一般二叉树,可能造成存储空间的浪费。

2. 链式存储结构

所谓链式存储是指用链表来表示一棵二叉树,即用链来指示元素的逻辑关系。通常有下面两种形式。

1) 二叉链表存储

链表中每个结点由 3 个域组成,即数据域和两个指针域。data 域存放某结点的数据信息；lchild 与 rchild 分别存放指向左孩子和右孩子的指针。当左孩子或右孩子不存在时,相应指针域值为空(用符号∧或 NULL 表示)。结点的存储结构如图 3-9 所示。

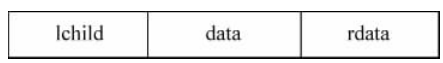


图 3-9　二叉链表存储结构

C 语言中,这种存储形式的类型定义如下所示。

```
typedef char DataType;                  /* 用户可根据具体应用定义 DataType 的实际类型 */
typedef struct bnode {
      DataType data;
      struct bnode *lchild, *rchild;    /* 左右孩子指针 */
  } BiTree;
```

图 3-10(b)给出了图 3-10(a)所示的一棵二叉树的二叉链表存储结构。链头的指针指向二叉树的根结点。容易证得,在含有 n 个结点的二叉链表中有 n+1 个空链表域。

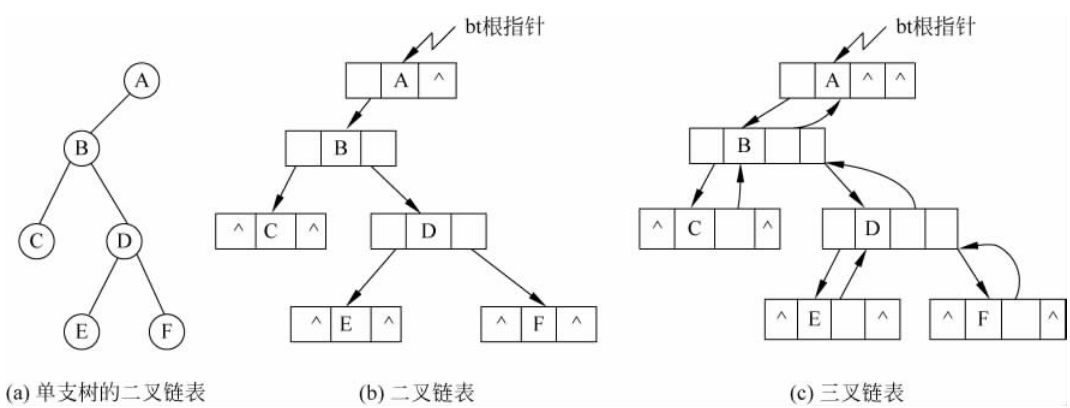


图 3-10　链表存储结构

2）三叉链表存储

每个结点由 4 个域组成，具体结构如图 3-11 所示。

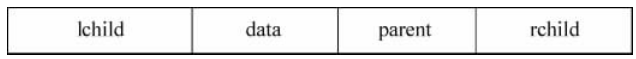


图 3-11　三叉链表存储结构

其中，data、lchild 及 rchild 三个域的意义与二叉链表结构相同；parent 域为指向该结点双亲结点的指针。这种存储结构既便于查找孩子结点，又便于查找双亲结点；但是，相对于二叉链表存储结构，它增加了空间开销。

图 3-10(c)给出了图 3-10(a)所示的一棵二叉树的三叉链表存储结构。

尽管，在二叉链表中无法由结点直接找到其双亲，但由于二叉链表结构灵活，操作方便，对于一般情况的二叉树，甚至比顺序存储结构还节省空间。例如，深度为 4 的右单支二叉树共有 4 个结点，二叉链表的 8 个指针域 3 个非空，顺序存储 2^4-1 共 15 个元素空间仅使用了 4 个。因此，二叉链表是最常用的二叉树存储方式。本书后面涉及的二叉树的链式存储结构，如不加特别说明都是指二叉链表结构。

3. 二叉树的基本操作

采用二叉链表存储结构表示的二叉树的基本操作实现如下所示。

1）二叉树的初始化操作

二叉树的初始化需要将指向二叉树的根结点指针置为空，代码如下：

```
void InitBitTree(BiTree *T)                    /*二叉树的初始化操作*/
{
    T=NULL;
}
```

2）二叉树的销毁操作

如果二叉树存在，将二叉树的存储空间释放。

```
void DestoryBitTree(BiTree *T)                 /*销毁二叉树*/
{
    if(T)                                      /*如果是非空二叉树*/
    {
        if(T->lchild)
            DestoryBitTree(T->lchild));
        if((T)->rchild)
            DestoryBitTree(T->rchild));
    free(T);
    T=NULL;
      }
}
```

3）创建二叉树操作

根据二叉树的递归定义，先生成二叉树的根结点，将元素值赋值给结点的数据域，然后递归创建左子树和右子树。其中“#”表示空。代码如下：

```
void CreateBitTree(BiTree *T)                    /*递归创建二叉树*/
{
    DataType ch;
    scanf("%c", &ch);
    if(ch=='#')
        T=NULL;
    else
    {
        T=(BiTree *)malloc(sizeof(bnode));       /*生成根结点*/
        if(!T)
          exit(-1);
        T->data=ch;
        CreateBitTree(T->lchild));               /*构造左子树*/
        CreateBitTree(T->rchild));               /*构造右子树*/
    }
}
```

4）二叉树的左插入操作

指针 p 指向二叉树 T 的某个结点，将子树 c 插入到 T 中，使 c 成为 p 指向结点的左子树，p 指向结点的原来左子树成为 c 的右子树。代码如下：

```
int InsertLeftChild(BiTree p,BiTree c)           /*二叉树的左插入操作*/
{
    if(p)                                        /*如果指针 p 不空*/
    {
        c->rchild=p->lchild;                     /*p 的原来的左子树成为 c 的右子树*/
        p->lchild=c;                             /*子树 c 作为 p 的左子树*/
        return 1;
  }
    return 0;
}
```

5）二叉树的右插入操作

指针 p 指向二叉树 T 的某个结点，将子树 c 插入到 T 中，使 c 成为 p 指向结点的右子树，p 指向结点的原来左子树成为 c 的右子树。代码如下：

```
int InsertReftChild(BiTree p,BiTree c)           /*二叉树的右插入操作*/
{
    if(p)                                        /*如果指针 p 不空*/
    {
        c->rchild=p->rchild;                     /* p 的原来的右子树成为 c 的右子树*/
        p->rchild=c;                             /*子树 c 作为 p 的右子树*/
        return 1;
    }
        return 0;
}
```

6）二叉树的左删除操作

二叉树中，指针 p 指向二叉树中的某个结点，将 p 所指向的结点的左子树删除。如果删除成功，返回 1；否则，返回 0。代码如下：

```
Int DeleteLeftChild(BiTree p)                    /* 二叉树的左删除操作 */
{
    if(p)
    {
        DestoryBiTree(p->lchild);                /* 删除 p 指向结点的左子树 */
        return 1;
}
    return 0;
}
```

7）二叉树的右删除操作

二叉树中，指针 p 指向二叉树中的某个结点，将 p 所指向的结点的右子树删除。如果删除成功，返回 1；否则，返回 0。代码如下：

```
Int DeleteRightChild(BiTree p)                   /* 二叉树的右删除操作 */
{
    if(p)
    {
        DestoryBiTree(&(p->rchild));             /* 删除 p 指向结点的右子树 */
        return 1;
}
    return 0;
}
```

8）返回二叉树的左孩子元素值基本操作

如果元素值为 e 的结点存在，并且该结点的左孩子结点存在，则将该结点的左孩子结点的元素值返回。代码如下：

```
DataType LeftChild(BiTree T, DataType e)         /* 返回二叉树的左孩子结点元素值操作 */
{
    BiTree p;                                    /* 如果二叉树非空 */
    if(T)
    {
    p = Point(T, e);                             /* p 是元素值 e 的结点的指针 */
    if(p&&p->lchild)                             /* 如果 p 不为空，且 p 的左孩子结点存在 */
      return p->lchild->data;                    /* 返回 p 的左孩子结点的元素值 */
    }
return;
}
```

9）返回二叉树的右孩子元素值基本操作

如果元素值为 e 的结点存在，并且该结点的右孩子结点存在，则将该结点的右孩子结点的元素值返回。代码如下：

```
DataType LeftChild(BiTree T, DataType e)         /* 返回二叉树的右孩子结点元素值操作 */
{
    BiTree p;                                    /* 如果二叉树非空 */
    if(T)
    {
    p = Point(T, e);                             /* p 是元素值 e 的结点的指针 */
```

```
    if(p&&p->rchild)                    /* 如果 p 不为空,且 p 的右孩子结点存在 */
      return p->rchild->data;           /* 返回 p 的右孩子结点的元素值 */
      }
    return;
}
```

3.3　二叉树的遍历

3.3.1　遍历的概念

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

这里提到的“访问”含义很广,可以是查询、修改、输出某元素的值,以及对元素做某种运算等,但要求这种访问不破坏它原来的数据结构。遍历一个线性结构很简单,只须从开始结点出发,顺序扫描每个结点。但对二叉树这样的非线性结构,每个结点可能有两个后继结点,因此需要寻找一种规律来系统访问树中的各结点。

遍历是二叉树中经常用到的一种操作。因为在实际应用问题中,常常需要按一定顺序对二叉树中的每个结点逐个进行访问,查找具有某一特点的结点,然后对这些满足条件的结点进行处理。通过一次完整的遍历,可使二叉树中结点信息由非线性排列变为某种意义上的线性序列。也就是说,遍历操作可使非线性结构线性化。

3.3.2　二叉树遍历算法

1. 二叉树的遍历方法及递归实现

由于二叉树的定义是递归的,它是由 3 个基本单元组成,即根结点、左子树和右子树。因此,遍历一棵非空二叉树的问题可以分解为 3 个子问题,即访问根结点、遍历左子树、遍历右子树,只要依次遍历这 3 部分,就可以遍历整个二叉树。由于实际问题一般都要求左子树较右子树先遍历,通常习惯先左后右的原则,放弃了先右后左的次序。于是,将根结点放在左、右子树的前、中、后,得到 3 种遍历次序,分别称为先序遍历、中序遍历和后序遍历。令 L,R,T 分别代表二叉树的左子树、右子树、根结点,则有 TLR、LTR、LRT 3 种遍历规则。

1) 先序遍历二叉树(TLR)

先序遍历的递归过程为:若二叉树为空,遍历结束。否则,

(1) 访问根结点。

(2) 先序遍历根结点的左子树。

(3) 先序遍历根结点的右子树。

先序遍历二叉树的递归算法如下:

```
void PreOrder(BiTree *bt)
{/* 先序遍历二叉树 bt */
    if (bt == NULL) return;             /* 递归调用的结束条件 */
    Visit(bt->data);                    /* 访问结点的数据域 */
    PreOrder(bt->lchild);               /* 先序递归遍历 bt 的左子树 */
    PreOrder(bt->rchild);               /* 先序递归遍历 bt 的右子树 */
}
```

2）中序遍历二叉树(LTR)

中序遍历的递归过程为：若二叉树为空，遍历结束。否则，

(1) 中序遍历根结点的左子树。

(2) 访问根结点。

(3) 中序遍历根结点的右子树。

中序遍历二叉树的递归算法如下：

```
void InOrder(BiTree *bt)
{/*中序遍历二叉树 bt*/
    if (bt==NULL) return;                    /*递归调用的结束条件*/
    InOrder(bt->lchild);                     /*中序递归遍历 bt 的左子树*/
    Visit(bt->data);                         /*访问结点的数据域*/
    InOrder(bt->rchild);                     /*中序递归遍历 bt 的右子树*/
}
```

3）后序遍历二叉树(LRT)

后序遍历的递归过程为：若二叉树为空，遍历结束。否则，

(1) 后序遍历根结点的左子树。

(2) 后序遍历根结点的右子树。

(3) 访问根结点。

后序遍历二叉树的递归算法如下：

```
void PostOrder(BiTree *bt)
{/*后序遍历二叉树 bt*/
    if (bt==NULL) return;                    /*递归调用的结束条件*/
    PostOrder(bt->lchild);                   /*后序递归遍历 bt 的左子树*/
    PostOrder(bt->rchild);                   /*后序递归遍历 bt 的右子树*/
    Visit(bt->data);                         /*访问结点的数据域*/
}
```

如图 3-12 所示的二叉树，若要先序遍历此二叉树，按访问结点的先后顺序将结点排列起来，可以得到该二叉树的先序序列 ABDGCEF。

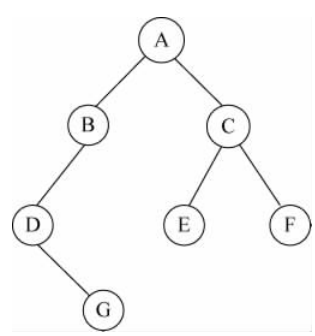


图 3-12　二叉树

类似地，中序遍历此二叉树，可以得到二叉树的中序序列 DGBAECF。

后序遍历此二叉树，可以得到二叉树的后序序列 GDBEFCA。

2. 二叉树遍历的非递归实现

先序、中序、后序遍历的非递归算法共同之处，即用栈来保存先前走过的路径，以便在访问完子树后，可以利用栈中的信息，回退到当前节点的双亲节点，进行下一步操作。

1）先序遍历二叉树

算法实现：从二叉树根结点开始，沿左子树一直走到末端(左子树为空)为止，在走的过程中，访问所遇结点，并依次把遇到的结点进栈。当左子树为空时，从栈顶退出某结点，并将

指针指向该结点的右子树。如此重复，直到栈为空或指针为空止。

先序遍历非递归算法如下：

```
void PreOrder (BiTree * t)          /* 非递归先序遍历二叉树 t,对每个元素调用 Visit 函数 */
{   PSeqStack S;
    bnode * p = t;
    S = Init_SeqStack ( );
    while ( p ||!Empty_SeqStack ( S))
        {  if ( p)                              /* 二叉树非空 */
            {  Visit (p->data) ;                /* 访问结点的数据域 */
               Push_SeqStack ( S, p);
               p = p->lchild;                   /* 遍历左子树 */
            }
        else
          {
          Pop_SeqStack (S,&p );
          p = p->rchild;                        /* 向右跨一步,以便遍历右子树 */
            }
    }
}
```

2）中序遍历二叉树

算法实现：设根指针为 p，可能有以下两种情况。

（1）若 p！＝NULL，则 p 入栈，遍历其左子树。

（2）若 p＝＝NULL，则返回。此时，

① 若栈空，则整个遍历结束；

② 否则，表明栈顶结点的左子树已遍历结束。此时，退栈，访问 p，并遍历其右子树。

中序遍历非递归算法如下：

```
void InOrder (BiTree * t )          /* 非递归中序遍历二叉树 t,对每个元素调用 Visit 函数 */
{   PSeqStack S;
    bnode * p = t;
    S = Init_SeqStack ( );
    while ( p ||!Empty_SeqStack ( S))
     {
        if ( p)                                 /* 二叉树非空 */
        {
            Push_SeqStack ( S, p);
            p = p->lchild;
        }
      else
        {
            Pop_SeqStack (S,&p );
            Visit (p->data) ;
            p = p->rchild;
        }
    }
}
```

3）后序遍历二叉树

算法实现：利用栈来实现二叉树的后序遍历要比先序和中序遍历复杂得多。后序遍历中，当搜索指针指向某一个结点时，不能马上进行访问，而先要遍历左子树，所以此结点应先进栈保存，当遍历完它的左子树后，再次回到该结点，还不能访问它，还需先遍历其右子树，所以该结点必须再次进栈，只有等它的右子树遍历完后，再次退栈时，才能访问该结点。为了区分同一结点的两次进栈，引入一个栈次数的标志，元素第 1 次进栈标志为 0，第 2 次进标志为 1，当退出的元素标志为 1 时，访问结点。

设根指针为 p，可能有以下两种情况。

(1) 若 p! ＝NULL，则 p 及标志 flag(＝0)入栈，遍历其左子树。

(2) 若 p＝＝NULL，则返回。此时，

① 若栈空，则整个遍历结束；

② 否则，表明栈顶结点的左子树或右子树已遍历结束。此时，若栈顶结点的 flag＝0，则修改为 1，并遍历其右子树；否则，访问栈顶结点并退栈，再转至(1)。

后序遍历非递归算法如下：

```
typedef struct
  {
    bnode * node;
    int flag;
  } DataType;
  void PostOrder (BiTree * t ) /* 非递归后序遍历二叉树 t,对每个元素调用 Visit 函数 */
  {
    PSeqStack S;
    DataType Sq;
    bnode * p = t;
    S = Init_SeqStack( );
    while ( p ||!Empty_SeqStack (S ))
  {
    if ( p)
      {
        Sq.flag = 0;
        Sq.node = p;
        Push_SeqStack (S, Sq);
        p = p->lchild; }
    else
      {
        Pop_SeqStack (S,&Sq);
        p = Sq.node;
      if (Sq.flag == 0)
      {
        Sq.flag = 1;
        Push_SeqStack (S,Sq);
        p= p->rchild; }
      else
      {
        Visit (p->data );
        p = NULL; }
```

```
        }
    }
}
```

3. 二叉树的层次遍历

所谓二叉树的层次遍历，就是指从二叉树的第 1 层(根结点)开始，从上到下逐层遍历，同一层中，则按照从左到右的顺序对结点逐个访问。对于图 3-12 所示的二叉树，按层次遍历所得到的结果序列为 ABCDEFG。

按层次遍历二叉树的算法描述：使用一个队列结构完成这项操作。所谓记录访问结点就是入队操作；取出记录的结点就是出队操作。

(1) 访问根结点，并将根结点入队。

(2) 当队列不空时，重复下列操作：

① 从队列退出一个结点；

② 若其有左孩子，则访问左孩子，并将其左孩子入队；

③ 若其有右孩子，则访问右孩子，并将其右孩子入队。

下面的层次遍历算法中，二叉树以二叉链表存放，一维数组 Queue[MAXNODE]用以实现队列，变量 front 和 rear 分别表示当前队首元素和队尾元素在数组中的位置。

```
void LevelOrder(BiTree *bt)
/*层次遍历二叉树 bt*/
{
    BiTree Queue[MAXNODE];
    int front,rear;
    if (bt==NULL) return;
    front=-1;
    rear=0;
    queue[rear]=bt;
    while(front!=rear)
    {
        front++;
        Visit(queue[front]->data);              /*访问队首结点的数据域*/
        if (queue[front]->lchild!=NULL)         /*将队首结点的左孩子结点入队列*/
        {
            rear++;
            queue[rear]=queue[front]->lchild;
        }
        if (queue[front]->rchild!=NULL)         /*将队首结点的右孩子结点入队列*/
        {
            rear++;
            queue[rear]=queue[front]->rchild;
        }
    }
}
```

二叉树遍历算法的时间和空间复杂度分析，由于遍历二叉树算法中的基本操作是访问结点，则不论按哪种次序进行遍历，对含有 n 个结点的二叉树，其时间复杂度均为 O(n)。所需辅助空间为遍历过程中栈的最大容量，即树的深度，最坏情况下为 n，则空间复杂度也为 O(n)。

3.3.3 二叉树遍历算法的应用

二叉树的遍历应用很广泛，本书主要通过几个例子来说明二叉树遍历的典型应用。

1. 编写求二叉树结点个数的算法

算法 1：在中序(或先序、后序)遍历算法中对遍历到的结点进行计数(count 应定义成全局变量，初值为 0)。

```
void InOrder ( BiTree *t )/*将二叉树 t 中的结点数累加到全局变量 count 中,count 的初值 0*/
{
    if (t)
      {  InOrder ( t->lchild );
         count = count + 1;
         InOrder ( t->rchild );
      }
}
```

算法 2：将一棵二叉树看成由树根、左子树和右子树 3 个部分组成，所以总的结点数是这 3 部分结点数之和，树根的结点数或者是 1 或者是 0(为空时)，而求左右子树结点数的方法和求整棵二叉树结点数的方法相同，可用递归方法。

```
int Count ( BiTree *t )
{
    int lcount,rcount;
    if (t == NULL) return 0;
    lcount = Count(t->lchild);
    rcount = Count(t->rchild);
    return lcount + rcount + 1; }
```

2. 设计算法求二叉树的高度

算法 1：使用全局变量。

```
void High( BiTree *bt )
{/*求二叉树 bt 的高度并存储到全局变量 h 中,h 初值为 0*/
   int hl;
   if(bt == NULL) h = 0;          /*bt 为空时,高度为 0*/
   else{
        High(bt->lchild);    /*求左子树的高度并存储到全局变量 h 中*/
        hl = h;
        High(bt->rchild);    /*求右子树的高度并存储到全局变量 h 中*/
        h = (hl > h? hl:h) + 1;  /*若二叉树不空,其高度应是其左右子树高度的最大值再加 1*/
       }
}
```

算法 2：使用带指针形参。

```
void High( BiTree *bt, int *h )
{/*求二叉树 bt 的高度并存储到 h 所指向的内存单元*/
   int hl, hr;
   if(bt == NULL) *h = 0;         /*bt 为空时,高度为 0*/
   else{
```

```
        High(bt->lchild,&hl);    /*求左子树的高度并存储到局部变量 hl 中*/
        High(bt->rchild,&hr);    /*求右子树的高度并存储到局部变量 hr 中*/
         *h=(hl>hr? hl:hr)+1;
                                 /*若二叉树不空,其高度应是其左右子树高度的最大值再加 1*/
       }
}
```

算法 3：通过函数值返回结点数。

```
int High( BiTree *bt )
{/*求二叉树 bt 的高度并通过函数值返回*/
   int hl,hr,h;
   if(bt==NULL) h=0;           /*bt 为空时,高度为 0*/
   else{
        hl=High(bt->lchild); /*求左子树的高度并暂时存储到局部变量 hl 中*/
        hr=High(bt->rchild); /*求右子树的高度并暂时存储到局部变量 hr 中*/
        h=(hl>h? hl:h)+1;    /*若二叉树不空,其高度应是其左右子树高度的最大值再加 1*/
       }
return h;
}
```

3. 创建二叉树的二叉链表存储结构

由二叉树的先序、中序、后序序列中的任何一个序列是不能唯一确定一棵二叉树的，原因是不能确定左右子树的大小或者说不知其子树结束的位置。针对这种情况，做如下处理。将二叉树中每个结点的空指针处再引出一个“孩子”结点，其值为特定值，如用 0 标识其指针为空。例如，要建立如图 3-12 所示的二叉树，其先序输入序列应该是 ABD0G00CE00F00。

根据二叉树的递归定义，先生成二叉树的根结点，将元素值赋值给结点的数据域，然后递归创建左子树和右子树。

```
void CreateBinTree(BiTree *T)
{/*以加入结点的先序序列输入,构造二叉链表*/
  char ch;
  scanf("\n%c",&ch);
  if (ch=='0') *T=NULL;                          /*读入 0 时,将相应结点置空*/
  else {*T=(BinTNode*)malloc(sizeof(bnode));     /*生成结点空间*/
  (*T)->data=ch;
  CreateBinTree(&(*T)->lchild);                  /*构造二叉树的左子树*/
  CreateBinTree(&(*T)->rchild);                  /*构造二叉树的右子树*/
  }
}
void InOrderOut(BiTree *T)
{/*中序遍历输出二叉树 T 的结点值*/
  if (T)
  {   InOrderOut(T->lchild);                     /*中序遍历二叉树的左子树*/
    printf("%3c",T->data);                       /*访问结点的数据*/
    InOrderOut(T->rchild);                       /*中序遍历二叉树的右子树*/
  }
}
main()
{
```

```
    BiTree bt;
    CreateBinTree(&bt);
    InOrderOut(bt);
}
```

4. 查找数据元素

下面的算法 Search(bt,x)实现在非空二叉树 bt 中查找数据元素 x。若查找成功,则返回该结点的指针;若查找失败,则返回空指针。

```
BiTree Search(BiTree * bt, int x)
{/ * 先序查找,在以 bt 为根的二叉树中值为 x 的结点是不是存在 * /
    BiTree * temp;
    if(bt == NULL) return NULL;
    if(x == bt -> data) return bt;
    temp = search(bt -> lchild, x);          / * 在 bt -> lchild 为根的二叉树中查找数据 x * /
    if(temp!= NULL) return temp;
    else return search(bt -> rchild, x);     / * 在 bt -> rchild 为根的二叉树中查找数据 x * /
}
```

5. 设计算法求二叉树每层结点的个数

算法思想:先序或者中序遍历时,都是从一个结点向它的左孩子或者右孩子移动,如果当前结点位于 L 层,则它的左孩子或者右孩子肯定是在 L+1 层。在遍历算法中给当前访问到的结点增设一个指示该结点所位于的层次变量 L,设二叉树高度为 H,定义一个全局数组 num[1…H],初始值为 0,num[i]表示第 i 层上的结点个数。

```
void Levcount (BiTree * t, int L)
{   if ( t)
    {
      Visit (t -> data); num[L]++;
      Levcount (t -> lchild, L + 1);
      Levcount (t -> rchild, L + 1);
    }
}
}
```

3.4 树和森林

树、森林和二叉树本身都是树的一种,它们之间都可以相互转换,本节将讨论树和森林的存储结构,并建立森林与二叉树的对应关系。

3.4.1 树和森林的存储结构

1. 树的存储结构

在大量的应用中,人们曾使用多种形式的存储结构来表示树。这里,介绍 3 种常用的链表结构,分别是双亲表示法、孩子表示法和孩子兄弟表示法。

1) 双亲表示法

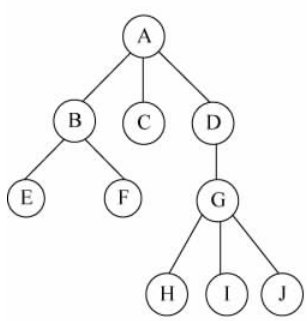
由树的定义可知,树中根结点无双亲,其他任何一个结点都只有唯一的双亲。根据这一

特性，可以以一组连续的空间存储树的结点，通过保存每个结点的双亲结点位置，表示树中结点之间的结构关系，树的这种存储方法称为双亲表示法。

双亲表示法的存储结构定义可以描述为

```
#define MaxNodeNum 100                    /* 树中结点的最大个数 */
typedef struct {                          /* 结点结构 */
   DataType data;
   int parent;
} Parentlist;
typedef struct {                          /* 树结构 */
  Parentlist elem[MaxNodeNum];
  int r, n;                               /* 双亲结点的位置和结点个数 */
} ParentTree;
```

图 3-13 所示的是一棵树及其双亲表示的存储结构。图 3-13 中用 parent 域的值为－1 表示该结点无双亲结点，即该结点是一个根结点。



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

图 3-13　树的双亲表示法

树的双亲表示法对于实现 Parent(T,x)操作和 Root(T)操作很方便，但若求某结点的孩子结点，即实现 Child(T,x,i)操作时，则需要查询整个数组。此外，这种存储方式不能反映各兄弟结点之间的关系，所以实现 RightSibling(T,x)操作也比较困难。实际中，如果需要实现这些操作，可在结点结构中增设存放第一个孩子的域和存放右兄弟的域，就能较方便地实现上述操作。

2）孩子表示法

由于每个结点可能有多棵子树，可以考虑使用多重链表，即每个结点有多个指针域，其中每个指针指向一棵子树的根结点，这种方法称为多重链表表示法。不过树的每个结点的度，也就是它的孩子个数是不同的，所以可以设计两种方案来解决。

方案 1：根据树的度 d 为每个结点设置 d 个指针域，多重链表中的结点是同构的。由于树中很多结点的度小于 d，所以链表中有很多空链域，造成存储空间浪费。不难推出，在一棵有 n 个结点，度为 d 的树中必有 n(d－1)＋1 个空链域。不过，若树的各结点度相差很小时，就意味着开辟的空间都利用了，这时缺点反而变成了优点。其结构如图 3-14 所示。

方案 2：每个结点指针域的个数等于该结点的度，专门取一个位置来存储结点指针域的个数。在结点中设置 degree 域，指出结点的度。其结构如图 3-15 所示：

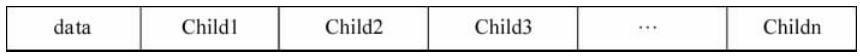


图 3-14　方案 1 结构图

图 3-15　方案 2 结构图

这种方法克服了浪费空间的缺点，对空间的利用率很高，但是各个结点的链表是不相同的结构，加上要维护结点的度的数值，运算上就会带来时间的损耗。

那么有没有更好的方法呢？既可以减少空指针的浪费，又能使结点结构相同。那就是用孩子表示法。具体办法是，把每个结点的孩子排列起来，以单链表做存储结构，则 n 个结点有 n 个孩子链表，如果是叶子结点则此单链表为空。然后 n 个头指针又组成一个线性表，采用顺序存储结构，存放进一个一维数组中。针对图 3-13 所示的树，其孩子表示法如图 3-16 所示。

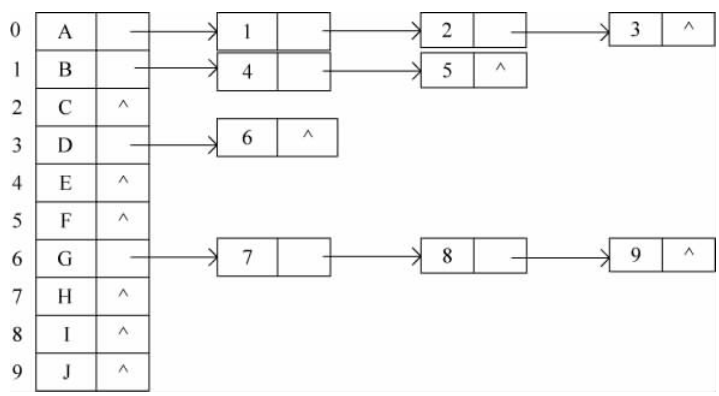


图 3-16　树的孩子表示法

树的孩子表示法使得查找已知结点的孩子结点非常容易。通过查找某结点的链表，可以找到该结点的每个孩子。但是查找双亲结点不方便，为此可以将双亲表示法与孩子表示法结合起来使用，图 3-17 就是将两者结合起来的带双亲的孩子链表。

树的孩子表示法的类型定义如下：

```
#define MAXNODE 100                          /* 树中结点的最大个数 */
struct ChildNode{                            /* 孩子结点 */
int childcode;
struct ChildNode *nextchild;
}
typedef struct {
elemtype data;
struct ChildNode *firstchild;                /* 孩子链表头指针 */
}NodeType;
NodeType t[MAXNODE];
```

3）孩子兄弟表示法

这种表示法又称为二叉树表示法，或二叉链表表示法。即以二叉链表作为树的存储结

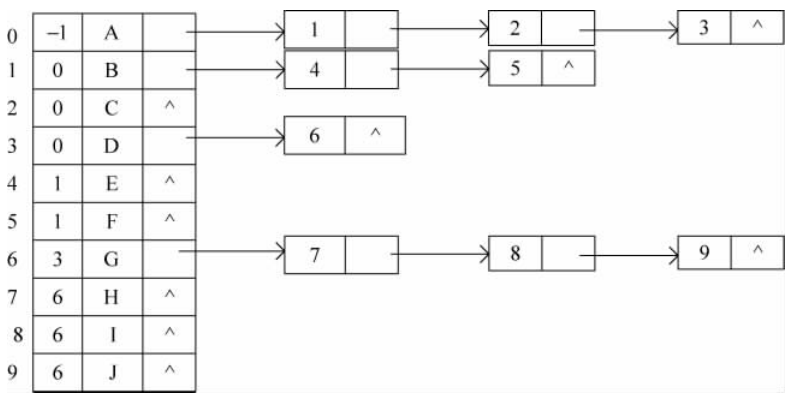


图 3-17　带双亲的孩子链表

构，链表中结点的两个域分别指向该结点的第一个孩子和下一个兄弟，分别命名为 firstchild 和 nextsibling 域。

树的孩子兄弟表示法的类型定义如下：

```
typedef struct tnode {
    DataType data;
    struct tnode *firstchild, *nextsibling;
} Tnode;
```

图 3-13 所示的树对应的孩子兄弟表示如图 3-18。利用树的孩子兄弟链表这种存储结构便于实现各种树的操作。例如，找某结点的第 i 个孩子，则只要先从左指针域中找到第 1 个孩子结点，然后沿着孩子结点的 nextsibling 域连续走 i－1 步便可找到第 i 个孩子。如增加一个 parent 域，也能方便实现求双亲的操作。

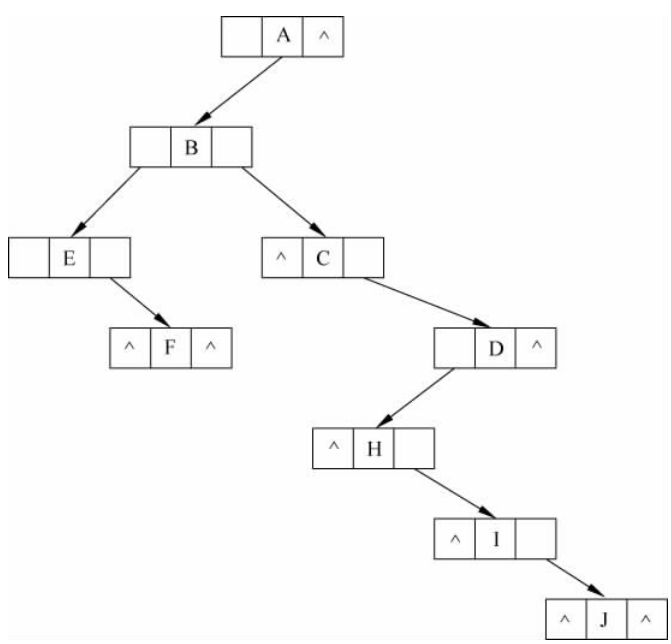


图 3-18　树的孩子兄弟表示法

2. 森林的存储结构

森林的存储方式也采用二叉链表的形式。森林实际上是多棵树结构的集合，而且在树的孩子兄弟表示法中表示每棵树的根结点的右指针必然为空。因此采用这样的方法，对于第 N+1 棵树将其根结点挂到表示第 N 棵树的根节点的右指针上即可。

3.4.2 树和森林与二叉树之间的转换

从树的孩子兄弟表示法可以看出，如果设定一定规则，就可以用二叉树结构表示树和森林。这样，对树的操作实现就可以借助二叉树存储，利用二叉树上的操作来实现。本节将讨论树和森林与二叉树之间的转换。

1. 树与二叉树的相互转换

1）树转换为二叉树

对于一棵无序树，树中结点的各孩子结点的次序无关紧要，而二叉树中结点的左、右孩子结点是有区别的。为避免发生混淆，约定树中每个结点的孩子结点按从左到右的次序编号。

将一棵树转换成二叉树的方法是

（1）加线：在兄弟之间加一连线。

（2）抹线：对每个结点，除了其左孩子外，去除其与其余孩子之间的关系。

（3）旋转：以树的根结点为轴心，将整树顺时针转 45°。

图 3-19 展示了树转换为二叉树的过程。经过这种方法转换后，对应的二叉树是唯一的。

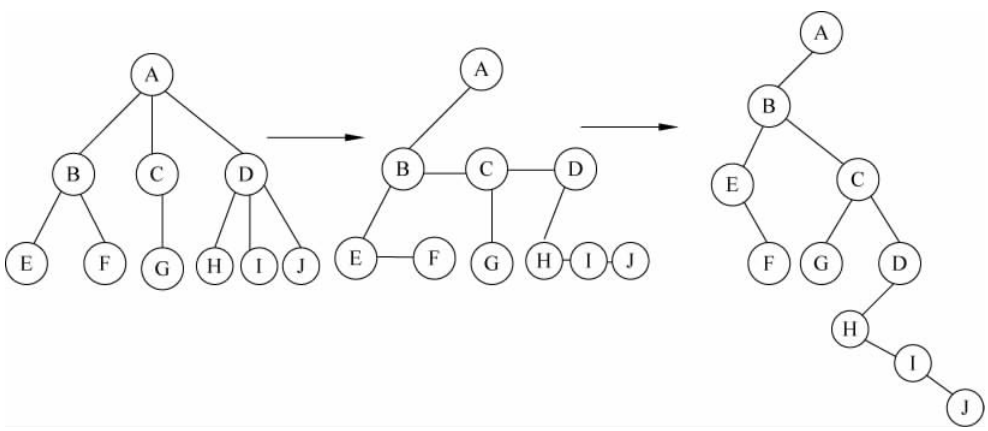


图 3-19 树转为二叉树的过程

由图 3-19 的转换可以看出，二叉树中，左分支上的各结点在原来的树中是父子关系，而右分支上的各结点在原来的树中是兄弟关系。由于树的根结点没有兄弟，所以变换后二叉树的根结点的右孩子必为空。

2）二叉树还原为树

二叉树转换为树，就是将树转换为二叉树的逆过程。树转换为二叉树，二叉树的根结点一定没有右孩子，对于一棵缺少右子树的二叉树也有唯一的一棵树与之对应。将一棵二叉树还原为树的的步骤如下：

（1）加线：若 p 结点是双亲结点的左孩子，则将 p 的右孩子，右孩子的右孩子……，沿分支找到的所有右孩子，都与 p 的双亲用线连起来。

（2）抹线：抹掉原二叉树中双亲与右孩子之间的连线。

（3）调整：将结点按层次排列，形成树结构。

图 3-20 展示了二叉树还原为树的过程。一棵没有右子树的二叉树经过这种方法还原后，对应的树也是唯一的。

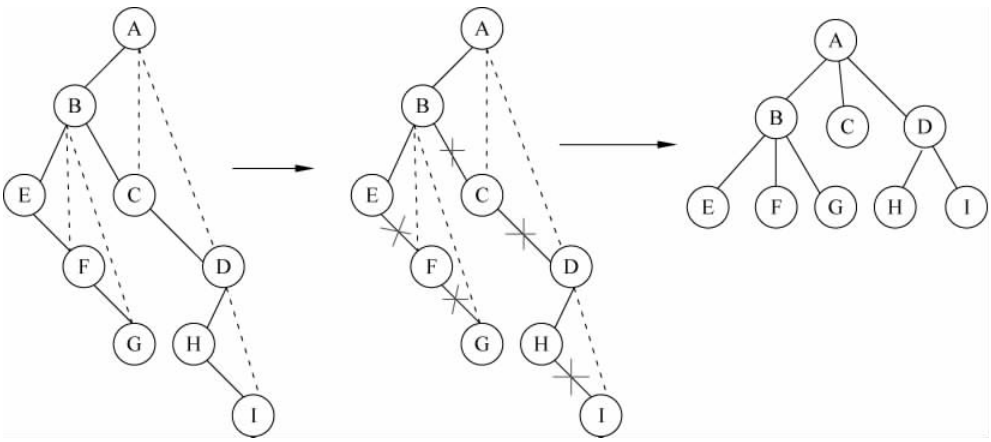


图 3-20　二叉树还原成树的过程

2. 森林与二叉树的相互转换

1）森林转换为二叉树

由森林的概念可知，森林是由若干棵树组成的集合，树可以转换为二叉树，那么森林也可以转换为二叉树。如果将森林中的每棵树转换为对应的二叉树，则再将这些二叉树按照规则转换为一棵二叉树，就实现了森林到二叉树的转换。森林转换为对应的二叉树的步骤如下：

（1）将森林中各棵树分别转换成二叉树。

（2）第 1 棵二叉树不动，从第 2 棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，当所有二叉树连在一起后，所得到的二叉树就是由森林转换得到的二叉树。

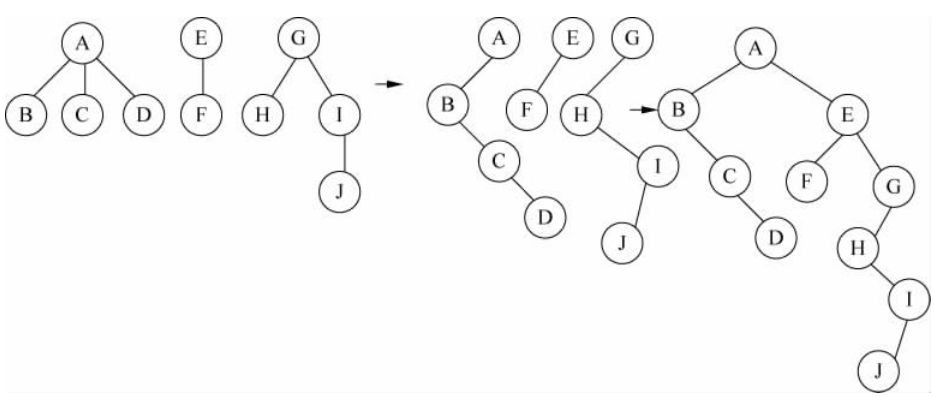


图 3-21　森林转化为二叉树的过程

2）二叉树还原为森林

将二叉树还原为森林的方法不难构造，具体还原过程如下：

（1）抹线：将二叉树中根结点与其右孩子连线，及沿右分支搜索到的所有右孩子间连线全部抹掉，使之变成孤立的二叉树。

（2）还原：将孤立的二叉树还原成树。

（3）调整：将还原后的树的根结点排列成一排。

图 3-22 展示了二叉树还原为森林的过程。一棵具有左子树和右子树的二叉树经过这种方法还原后，对应的森林是唯一的。

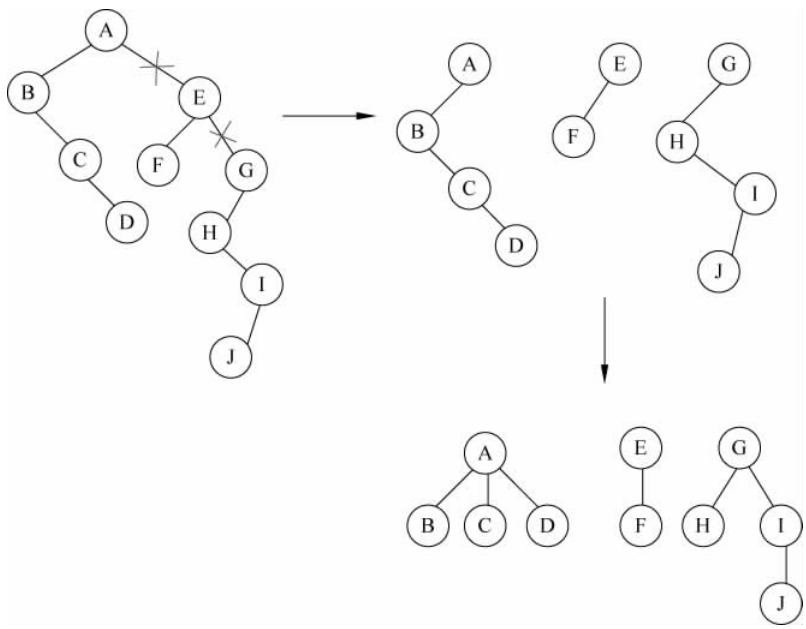


图 3-22　二叉树还原为森林的过程

3.4.3　树和森林的遍历

由树结构的定义可引出树的两种次序的遍历。一种是先根遍历树，即先访问树的根结点，然后依次先根遍历树的每棵子树；另一种是后根遍历，即先依次后根遍历每棵子树，然后访问根结点。

对图 3-1 的树进行先根遍历，可得到树的先根遍历序列为

ABDGHICEJF

若对此树进行后根遍历，则得到树的后根序列为

GHIDBJEFCA

按照森林和树相互递归的定义，可以推出森林的两种遍历方法。

1. 先序遍历森林

若森林非空，则可按下述规则遍历。

(1) 访问森林中的第一棵树的根结点。

(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子树之后剩余的树构成的森林。

2. 中序遍历森林

若森林非空,则可按下述规则遍历。

(1) 中序遍历森林中第一棵树中根结点的子树森林。

(2) 访问第一棵子树的根结点。

(3) 中序遍历除去第一棵子树之后剩余的树构成的森林。

对图 3-22 中森林进行先序遍历,可得到森林的先序序列为

```
ABCDEFGHIJ
```

若对此森林进行中序遍历,则得到森林的中序序列为

```
BCDAFEHJIG
```

由 3.4.2 节森林与二叉树之间转换的规则可知,当森林转换成二叉树时,其第一棵树的子树森林转换成左子树,剩余树的森林转换为右子树,则上述森林的先序和中序遍历即为其对应的二叉树的先序和中序遍历。由此可见,当以二叉链表作为树的存储结构时,树的先根遍历和后根遍历可借用二叉树的先序遍历和中序遍历的算法来实现。

3.5　二叉树的应用

3.5.1　哈夫曼树及其应用

哈夫曼树也称为最优二叉树。它是一种带权路径最短的树,应用非常广泛。本节主要介绍哈夫曼树的定义、哈夫曼编码及哈夫曼编码算法的实现。

1. 哈夫曼树的定义

先介绍几个基本概念和术语。

一棵树中,从一个结点往下可以达到的孩子或子孙结点之间的通路,称为路径。通路中分支的数目称为路径长度。若规定根结点的层数为 1,则从根结点到第 L 层结点的路径长度为 L－1。

若将树中结点赋给一个有着某种含义的数值,则这个数值称为该结点的权。从根结点到该结点之间的路径长度与该结点的权的乘积称为结点的带权路径长度。

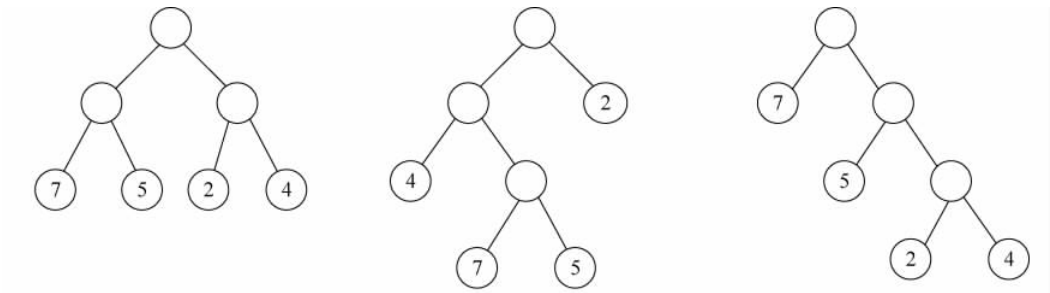
树的带权路径长度(Weighted Path Length of Tree,简记为 WPL)规定为所有叶子结点的带权路径长度之和:

$$WPL = \sum_{i=1}^{n} w_i * l_i \tag{3-1}$$

其中 n 为叶子结点数目,w_i为第 i 个叶子结点的权值,l_i为第 i 个叶子结点的路径长度。在一棵二叉树中,若树的带权路径长度达到最小,称这样的二叉树为最优二叉树,也称为哈夫曼树。

【例 3-1】 有 4 个结点,权值分别为 7,5,2,4,构造有 4 个叶子结点的二叉树。

由这 4 个叶子结点可以构造出形态不同的二叉树，如图 3-23 所示。



(a) WPL=7×2+5×2+2×2+4×2=36　(b) WPL=7×3+4×2+5×3+2×1=46　(c) WPL=7×1+5×2+2×3+4×3=35

图 3-23　具有不同带权路径长度的二叉树

从带权路径长度最小，这一角度来看，完全二叉树不一定是最优二叉树。图 3-23(c)树的 WPL 最小，此树就是哈夫曼树。由 n 个带权叶子结点所构成的二叉树中，满二叉树和完全二叉树不一定是最优二叉树。权值越大的结点离根结点越近的二叉树才是最优二叉树。

2. 哈夫曼树的建立

根据哈夫曼树的定义，一棵二叉树要使其 WPL 值最小，必须使权值越大的叶子结点越靠近根结点，而权值越小的叶子结点越远离根结点。哈夫曼依据这一特点提出了一种构造最优二叉树的方法，其基本思想如下：

(1) 根据给定的 n 个权值 $w_1, w_2, \cdots, w_n$ 构成 n 棵二叉树的森林 $F=\{T_1, T_2, \cdots, T_n\}$。其中，每棵二叉树 T_i 中都只有一个权值为 w_i 的根结点，其左右子树均空。

(2) 在森林 F 中选出两棵根结点权值最小的树(当这样的树不止两棵树时，可以从中任选两棵)，将这两棵树合并成一棵新树，为了保证新树仍是二叉树，需要增加一个新结点作为新树的根，并将所选的两棵树的根分别作为新根的左右孩子(谁左，谁右无关紧要)，将这两个孩子的权值之和作为新树根的权值。

(3) 对新的森林 F 重复(2)，直到森林 F 中只剩下一棵树为止。这棵树便是哈夫曼树。

例如，假设给定一组权值{7，5，2，4}，按照哈夫曼树构造的算法，对集合的权重构造哈夫曼树的过程如图 3-24 所示。

构造哈夫曼树时，可以设置一个结构数组 HuffNode 保存哈夫曼树中各结点的信息。根据二叉树的性质可知，具有 n 个叶子结点的哈夫曼树共有 2n－1 个结点，所以数组 HuffNode 的大小设置为 2n－1，数组元素的结构形式如图 3-25 所示。

其中，weight 域保存结点的权值，lchild 和 rchild 域分别保存该结点的左右孩子结点在数组 HuffNode 中的序号，从而建立起结点之间的关系。为了判定一个结点是否已加入到要建立的哈夫曼树中，可以通过当初 parent 的值为－1，当结点加入到树中时，该结点 parent 的值为其双亲结点在数组 HuffNode 中的序号，就不会是－1 了。

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

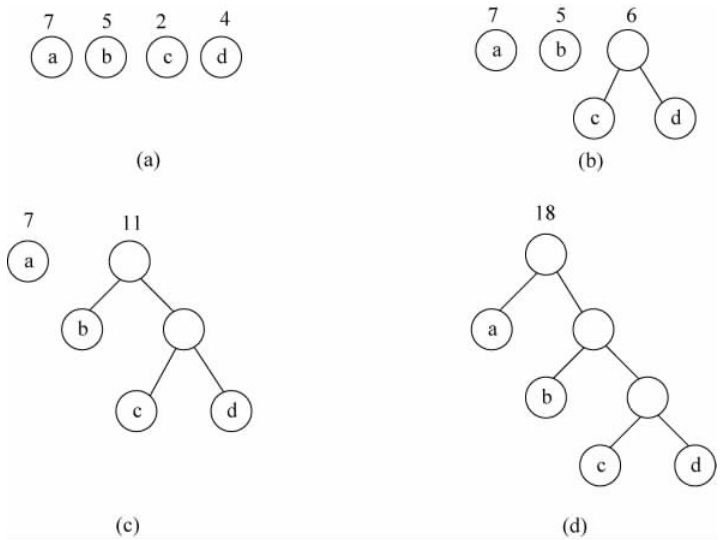


图 3-24　哈夫曼树的构造过程

weight	lchild	rchild	parent

图 3-25　数组元素的结构形式

哈夫曼树的构造算法如下：

```
#define MAXVALUE 10000                          /* 定义最大权值 */
#define MAXLEAF 30                              /* 定义哈夫曼树中叶子结点的最大个数 */
#define MAXNODE MAXLEAF * 2 - 1                 /* 定义哈夫曼树中结点的最大个数 */
typedef struct
{
    int weight;
    int parent;
    int lchild;
    int rchild;
}HNode, HuffmanTree[MAXNODE];
void CrtHuffmanTree(HuffmanTree ht, int w[], int n)  /* 数组 w[]传递 n 个权值 */
{
    int i,j,m1,m2,x1,x2;
    for(i = 0;i < 2 * n - 1;i++){               /* ht 初始化 */
    ht[i].weight = 0;
    ht[i].parent = -1;
    ht[i].lchild = -1;
    ht[i].rchild = -1;
}
for(i = 0;i < n;i++) ht[i].weight = w[i];       /* 赋予 n 个叶子结点的权值 */
for(i = 0;i < n - 1;i++)                        /* 构造哈夫曼树 */
{
    m1 = m2 = MAXVALUE;
    x1 = x2 = 0;
  for(j = 0;j < n + i;j++)                      /* 寻找权值最小和次小的两棵树 */
  {
```

```
        if(ht[j].weight < m1&&ht[j].parent == -1)
          {
            m2 = m1; x2 = x1; x1 = j;
            m1 = ht[j].weight;
          }
    else if (ht(j),weight < m2&&ht[j].parent == -1)
    {
        m2 = ht[j].weight;
        x2 = j;
      }
    }
    /* 将两棵子树合并成一棵子树 */
    ht[x1].parent = n + i; ht[x2].parent = n + i;
    ht[n + i].weight = ht[x1].weight + ht[x2].lweight;
    ht[n + i].lchild = x1; ht[n + i].rchild = x2;
      }
    }
```

3. 哈夫曼编码

哈夫曼编码常应用在数据通信中，数据传送时，需要将字符转换为二进制的编码串。例如，假设传送的电文是 ABDAACDA，电文中有 A、B、C、D 4 种字符，如果规定 A、B、C、D 的编码分别为 00、01、10、11，则上面的电文代码为 0001110000101100，总共 16 个二进制数。传送电文时，总是希望电文代码尽可能短，采用哈夫曼编码构造的电文的总长最短。

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不同，可以采用不等长的二进制编码，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。哈夫曼编码就是一种不等长的二进制编码，且哈夫曼树是一种最优二叉树，它的编码也是一种最优编码。

利用哈夫曼树编码来构造编码方案，就是哈夫曼树的典型应用。具体做法如下：

设需要编码的字符集合为 $D=\{d_1,d_2,\cdots,d_n\}$，它们在电文中出现的次数或频率集合为 $\{w_1,w_2,\cdots,w_n\}$，以 $d_1,d_2,\cdots,d_n$ 作为叶结点，$w_1,w_2,\cdots,w_n$ 作为它们的权值，构造一棵哈夫曼树，规定哈夫曼树中的左分支代表 0，右分支代表 1，则从根结点到每个叶结点所经过的路径分支组成的 0 和 1 的序列便为该结点对应字符的编码，称之为哈夫曼编码。这样的哈夫曼树亦成为哈夫曼编码树。

按照以上构造方法，字符集合为{A，B，C，D}，各个字符相应的出现次数为{4，1，1，2}，这些字符作为叶子结点构成的哈夫曼树如图 3-26 所示，字符 A 的编码为 0，字符 B 的编码为 110，字符 C 的编码为 111，字符 D 的编码为 10。

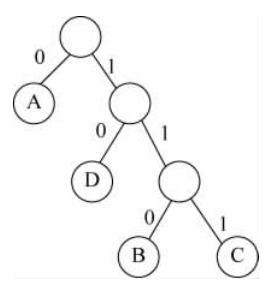


图 3-26　哈夫曼编码树

哈夫曼编码树中，树的带权路径长度的含义是各个字符的码长与其出现次数的乘积之和，也就是报文的代码总长，所以采用哈夫曼树的编码是一种能够使报文代码总长最短的不定长编码。

设计不等长编码时，必须使任何一个字符的编码都不是另外一个字符编码的前缀。例如，字符 A 的编码为 10，字符 B 的编码是 100，则字符 A 的编码就称为字符 B 编码的前缀。如果一个代

码是10010，在进行译码时，无法确定是将前两位译为A，还是将前三位译为B。但是在利用哈夫曼树进行编码时，每个编码是叶子结点的编码，一个字符是不会出现在另一个字符前面的，也就不会出现一个字符的编码是另一个字符编码的前缀编码。任何一个字符的编码都不是另一个字符编码的前缀，这种编码称为前缀编码。

哈夫曼编码的算法实现如下：

前面已经给出了哈夫曼树的构造算法，为实现哈夫曼树，自然需要定义一个编码表的存储结构。定义如下：

```
typedef struct codenode
{ char ch;                                          /*存放要表示的符号*/
  char *code;                                       /*存放相应的代码*/
}CodeNode
typedef CodeNode HuffmanCode[MAXLEAF];
```

哈夫曼编码的算法思路是在哈夫曼树中，从每个叶子结点开始，一直往上搜索，判断该结点是其双亲结点的左孩子还是右孩子，若是左孩子，则相应位置上的代码为0，否则是1，直至搜索到根结点为止。算法如下：

```
void CrtHuffmanCode(HuffmanTree ht,HuffmanCode hc,int n)
    /*从叶子结点到根，逆向搜索求每个叶子结点对应符号的哈夫曼编码*/
{
    char *cd;
    int i,c,p,start;
    cd=malloc(n*sizeof(char));                      /*为当前工作区分配空间*/
    cd[n-1]='\0';                                   /*从右到左逐位存放编码，首先存放结束符*/
    for(i=1;i<=n;i++)                               /*求n个叶子结点对应的哈夫曼编码*/
    {
      start=n-1;                                    /*编码存放的起始位置*/
    c=i;p=ht[i].parent;                             /*从叶子结点开始往上搜索*/
while(p!=0)
  {
    --start;
    if(ht[p].lchild==c) cd[start]='0'               /*左分支标0*/
      else cd[start]='1';                           /*右分支标1*/
    c=p; p=ht[p].parent;                            /*向上倒推*/
    }
  hc[i]=malloc((n-start)*sizeof(char));             /*为第i个编码分配空间*/
  scanf("%c",&(hc[i].ch));                          /*输入相应待编码字符*/
  strcpy(hc[i],&ch[start]);                         /*将工作区中编码复制到编码表中*/
}
free(cd);
}
```

3.5.2　二叉排序树

二叉排序树又称二叉查找树、二叉搜索树。它或者是一棵空树。或者是具有下列性质的二叉树：若左子树不空，则左子树上所有结点的值均小于它根结点的值；若右子树不空，则右子树上所有结点的值均大于等于它根结点的值；左、右子树也分别为二叉排序树。二叉排序树的相关算法参见4.9.1节。

例如，图 3-27 所示的二叉排序树。

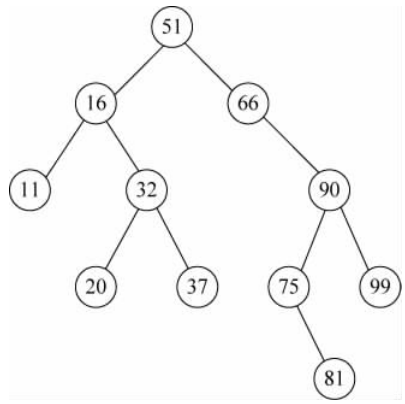


图 3-27　二叉排序树

由给定的数据序列生成二叉排序树的过程是在二叉排序树上插入结点的过程，对于一个数据序列(R_1，R_2，…，R_n)：

(1) 设 R_1 为二叉排序树的根。

(2) 若 $R_2 < R_1$，则令 R_2 为 R_1 左子树的根，否则为右子树的根。

(3) 对于 R_i，若 $R_i < R_1$，则进入左子树。否则进入右子树，继续与子树之根比较，直到找到某结点 R_k，若 $R_i < R_k$ 且 R_k 的左子树为空，则令 R_i 为 R_k 左子树的根；若 $R_i \geqslant R_k$ 且 R_k 的右子树为空，则令 R_i 为 R_k 的右子树的根。

例如，关键字序列(49，38，65，76，49，13，27，52)的插入过程如图 3-28 所示。首先，插入关键字 49，由于二叉排序树的初始状态为空树，则新生成的结点(49)应作为它的根结点；之后插入关键字 38，由于此时的二叉树不空，且 38＜49，则根据二叉排序树的定义，应插入在它的左子树上，而此时的左子树为空树，则生成的结点(38)应为左子树的根结点。同理，第 3 个关键字应插入到它的右子树上，并作为右子树的根结点，下一个关键字 76＞49，且 76＞65，则应插入成为(65)的右子树根结点……。以此类推，最后得到如图 3-28(i)所示的二叉排序树。

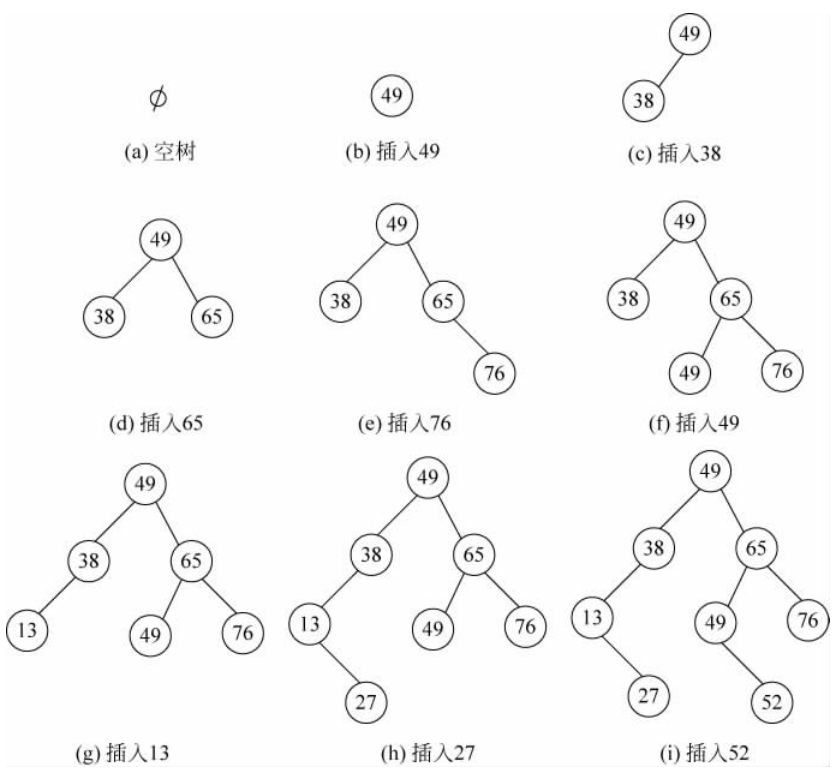


图 3-28　由关键字生序列生成二叉排序树的过程

3.6　图

图(Graph)是一种比线性表和树更为复杂的非线性数据结构。图中元素的关系既不像线性表中的元素至多只有一个直接前趋和一个直接后继，也不像树形结构中的元素具有明显的层次关系。图中元素之间的关系可以是任意的，每个元素(也称为顶点)可以具有多个直接前趋和后继，所以图可以表达数据元素之间广泛存在着的更为复杂的关系。图在语言学、逻辑学、数学、物理、化学、通信和计算机科学等领域中得到了广泛的应用。

3.6.1　图的基本概念

1. 图的定义

图(G)是一种非线性数据结构，它由两个集合 V(G)和 E(G)组成，形式上记为 G=(V，E)。其中，V(G)是顶点(Vertex)的非空有限集合，E(G)是 V(G)中任意两个顶点之间的关系集合，又称为边(Edge)的有限集合。

当 G 中的每条边有方向时，称 G 为有向图。有向边使用一对尖括号(< >)将两个顶点组成的有序对括起来，记为<起始顶点，终止顶点>。有向边又称为弧，因此弧的起始顶点就称为弧尾，终止顶点称为弧头。图 3-29 给出了一个有向图的示例，该图的顶点集和边集分别为

$V(G_1)=\{V_1, V_2, V_3, V_4\}$
$E(G_1)=\{<V_1, V_2>, <V_1, V_3>, <V_3, V_4>, <V_4, V_1>\}$

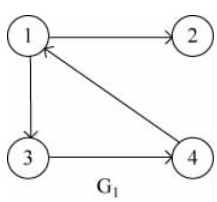


图 3-29　有向图示例

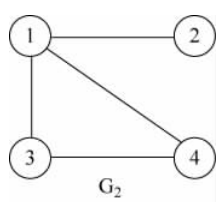


图 3-30　无向图示例

若 G 中的每条边是无方向时，称 G 为无向图。这时，两个顶点之间最多只存在一条边。无向图的一条边使用一对圆括号(( ))将两个顶点组成的无序对括起来，记为(顶点 1，顶点 2)或(顶点 2，顶点 1)。图 3-30 给出了一个无向图的示例。该图的顶点集和边集分别为

$V(G_2)=\{V_1, V_2, V_3, V_4\}$
$E(G_2)=\{(V_1, V_2), (V_1, V_3), (V_1, V_4), (V_3, V_4)\}=\{(V_2, V_1), (V_3, V_1), (V_4, V_1), (V_4, V_3)\}$

2. 基本术语

1) 完全图、稀疏图和稠密图

下面的讨论中，不考虑顶点到其自身的边，也不允许一条边在图中重复出现。在以上两条约束下，边和顶点之间存在以下关系。

(1) 对一个无向图，它的顶点数 n 和边数 e 满足 $0\leqslant e\leqslant n(n-1)/2$ 的关系。如果 $e=n(n-1)/2$，则该无向图称为完全无向图。

(2) 对一个有向图,它的顶点数 n 和边数 e 满足 $0 \leqslant e \leqslant n(n-1)$的关系。如果 $e = n(n-1)$,则称该有向图为完全有向图。

(3) 如果 $e < n\lg n$,则该图为稀疏图,否则为稠密图。

2) 子图

如果两个同类型的图 $G_1 = (V_1, E_1)$和 $G_2 = (V_2, E_2)$存在关系 $V_1 \subseteq V_2$,$E_1 \subseteq E_2$,则称 G_1 是 G_2 的子图,如图 3-31 所示。

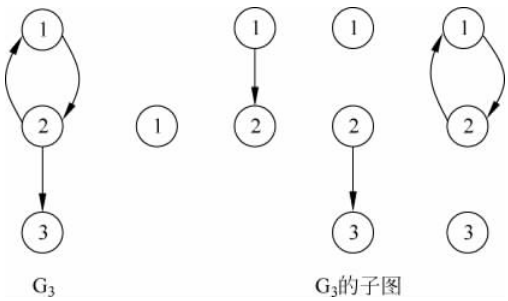


图 3-31 图与子图

3) 邻接点

无向图 G 中,若边$(V_i, V_j) \in E(G)$,则称顶点 V_i和 V_j相互邻接,两个顶点互为邻接点,并称边(V_i, V_j)关联于顶点 V_i和 V_j或称边(V_i, V_j)与顶点 V_i和 V_j相关联。例如,在图 3-30 中的顶点 1 与顶点 2、顶点 3 和顶点 4 互为邻接点,关联于顶点 1 的边是(V_1, V_2)、(V_1, V_3)和(V_1, V_4)。

有向图 G 中,若弧$<V_i, V_j> \in E(G)$,则称顶点 V_i邻接到 V_j或 V_j邻接自 V_i,并称弧$<V_i, V_j>$关联于顶点 V_i和 V_j或称弧$<V_i, V_j>$与顶点 V_i和 V_j相关联。例如,在图 3-29 中的顶点 1 邻接到顶点 2 和顶点 3,或称顶点 2 或顶点 3 邻接自顶点 1,而顶点 4 邻接到顶点 1,或称顶点 1 邻接自顶点 4。

4) 度、入度和出度

无向图中关联于某一顶点 V_i的边的数目称为 V_i的度,记为 $D(V_i)$。例如,图 3-30 中的顶点 1 的度为 3。有向图中,把以顶点 V_i为终点的边的数目称为 V_i的入度,记为 $ID(V_i)$;把以顶点 V_i为起点的边的数目称为 V_i的出度,记为 $OD(V_i)$;把顶点 V_i的度定义为该顶点的入度和出度之和。例如,图 3-29 中顶点 1 的入度为 1,出度为 2,度为 3。

如果图 G 中有 n 个顶点,e 条边,且每个顶点的度为 $D(V_i)(1 \leqslant i \leqslant n)$,则存在以下关系:

$$e = \sum_{i=1}^{n} D(v_i)/2 \qquad (3\text{-}2)$$

5) 权与网

一个图中,如果图的边或弧具有一个与它相关的数时,这个数就称为该边或弧的权,这个数常用来表示一个顶点到另一个顶点的距离或耗费。如果图中的每一条边都具有权时,这个带权图称为网络,简称为网,如图 3-32 所示。

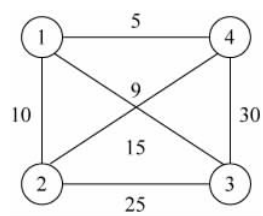


图 3-32 网(带权图)

6）路径与回路

一个图中，若从顶点 V_1 出发，沿着一些边经过顶点 V_1，V_2，…，V_{n-1} 到达顶点 V_n，则称顶点序列（V_1，V_2，…，V_{n-1}，V_n）为从 V_1 到 V_n 的一条路径。例如，图 3-30 中（V_4，V_3，V_1）是一条路径。而在图 3-29 中，（V_4，V_3，V_1）就不是一条路径。

将无权图沿路径所经过的边数，称为该路径的长度。而对有权图，取沿路径各边的权之和作为此路径的长度。图 3-31 所示的 G_3 中顶点 1 到顶点 3 的路径长度为 2。

若路径中的顶点不重复出现，则这条路径称为简单路径。起点和终点相同并且路径长度不小于 2 的简单路径称为简单回路或简单环。例如，图 3-30 的无向图中顶点序列（V_4，V_3，V_1）是一条简单路径，而（V_4，V_3，V_1，V_4）是一个简单环。

7）图的连通性

一个有向图中，若存在一个顶点 V，从该顶点有路径可到达图中其他所有顶点，则称这个有向图为有根图，V 称为该图的根。例如，图 3-29 就是一个有根图，该图的根是 V_1、V_3 和 V_4。

无向图 G 中，若顶点 V_i 和 V_j（$i \neq j$）有路径相通，则称 V_i 和 V_j 连通。如果 V(G) 中的任意两个顶点都连通，则称 G 是连通图，否则为非连通图，如图 3-33（a）所示。

无向图 G 中的极大连通子图称为 G 的连通分量。对任何连通图而言，连通分量就是其自身，对非连通图可有多个连通分量，如图 3-33（b）所示。

有向图 G 中，若从 V_i 到 V_j（$i \neq j$）、从 V_j 到 V_i 都存在路径，则称 V_i 和 V_j 强连通。若有向图 V(G) 中的任意两个顶点都是强连通的，则称该图为强连通图。有向图中的极大连通子图称作有向图的强连通分量。例如，图 3-29 中的顶点 V_1、V_3 和 V_4 是强连通的，但该有向图不是一个强连通图。

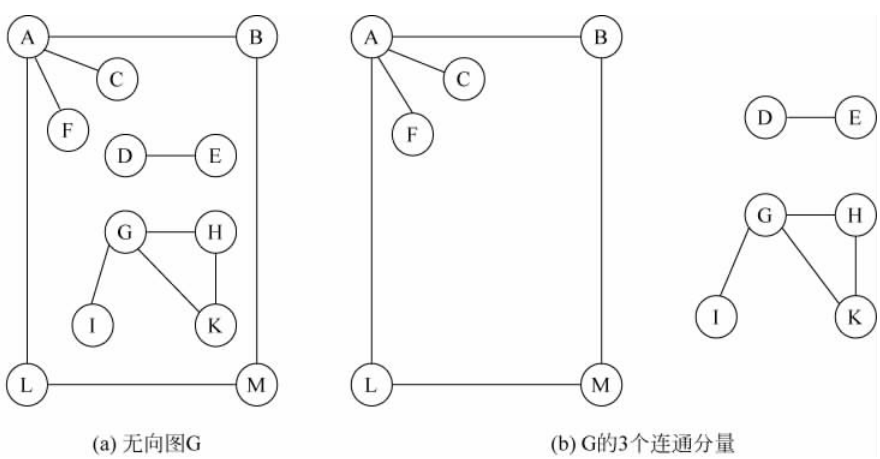


图 3-33　无向图及其连通分量

3.6.2　图的存储方法

由于图的结构复杂，任意两个顶点之间都可能存在联系，所以无法以数据元素在存储区中的物理位置来表示元素之间的关系，但仍可以借助一个二维数组中各单元的数据取值或用多重链表来表示元素间的关系。无论采用什么存储方法，都需要存储图中各顶点本身的

信息和存储顶点与顶点之间的关系。事先对图中的每个顶点进行顺序编号以后，各个顶点的数据信息即可保存在一个一维数组中，且顶点的编号与一维数组下标一一对应。因此，研究图的存储方法，主要是解决如何实现各个顶点之间关系的表示问题。

图的存储方法很多，常用的有邻接矩阵存储方法、邻接表存储方法、十字链表存储方法和多重邻接表存储方法。选择存储方法的依据取决于具体的应用和所要施加的运算。这里仅介绍邻接矩阵存储方法和邻接表存储方法。

1. 邻接矩阵存储方法

根据图的定义可知，一个图的逻辑结构分两部分，一部分是组成图的顶点的集合；另一部分是顶点之间的联系，即边或弧的集合。因此，计算机中存储图需要解决这两部分的存储表示。

邻接矩阵存储方法中，使用一个一维数组来存放图中每个顶点的数据信息，而利用一个二维数组（又称为邻接矩阵）来表示图中各顶点之间的关系。对一个有 n 个顶点的图 G 而言，将使用一个 n×n 的矩阵来表示其顶点间的关系，矩阵的每一行和每一列都顺序对应每一个顶点。矩阵中的元素 A[i, j]可按以下规则取值。

$$A[i,j]=\begin{cases}1, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\in E(G)\\ 0, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\notin E(G)\end{cases} \quad 0\leqslant i,j\leqslant n-1 \qquad (3\text{-}3)$$

一般情况下，大家不关心图中顶点的情况，若顶点编号为 1～vtxnum，设弧上或边上无权值，则使用 C 语言可以将图的存储结构简化为一个二维数组，如下所示。

```
int  adjmatrix[vtxnum][vtxnum];
```

如图 3-30 中的 G2 和图 3-31 中的 G3，其邻接矩阵分别如图 3-34 中 A_1、A_2所示。

$$A_1=\begin{bmatrix}0&1&1&1\\1&0&0&0\\1&0&0&1\\1&0&1&0\end{bmatrix} \qquad A_2=\begin{bmatrix}0&1&0\\1&0&1\\0&0&0\end{bmatrix}$$

图 3-34　图 G_2 和 G_3 的邻接矩阵

借助于邻接矩阵，可以很容易地求出图中顶点的度。

从上例可以看出，邻接矩阵有如下结论。

(1) 无向图的邻接矩阵是对称的，而有向图的邻接矩阵不一定对称。对无向图可考虑只存下三角（或上三角）元素。

(2)对于无向图，邻接矩阵第 i 行（或第 i 列）的元素之和是顶点 V_i 的度。

(3)对于有向图，邻接矩阵第 i 行元素之和为顶点 V_i 的出度；第 i 列的元素之和为顶点 V_i 的入度。

对于网络，邻接矩阵元素 A[i, j]可按以下规则取值。

$$A[i,j]=\begin{cases}W_{ij}, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\in E(G)\\ 0\text{或}\infty, & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\notin E(G)\end{cases} \quad 0\leqslant i,j\leqslant n-1 \qquad (3\text{-}4)$$

其中，W_{ij}是边(V_i, V_j)或弧$<V_i, V_j>$上的权值。

当一个图用邻接矩阵表示时，使用 C 语言编写算法可用以下数据类型说明。

```
#define n                                    /* 图的顶点数 */
#define e                                    /* 图的边数 */
typedef char vextype;                        /* 顶点的数据类型 */
typedef float adjtype;                       /* 顶点权值的数据类型 */
typedef struct {
  vextype vexs[n];                           /* 顶点数组 */
  adjtype arcs[n][n];                        /* 邻接矩阵 */
} graph;
```

下面给出了一个无向网络邻接矩阵利用上述类型说明的建立算法。

```
CREATGRAPH(graph *g)
{/* 建立无向网络 */
   int i, j, k;
   float w;
   for(i=0; i<n; i++)
    g->vexs[i]=getchar( );                   /* 读入顶点信息,建立顶点表 */
   for(i=0; i<n; i++)
    for(j=0; j<n; j++)
     g->arcs[i][j]=0;                        /* 邻接矩阵初始化 */
  for(k=0; k<e; k++) {
    scanf("%d%d%f", &i, &j, &w);             /* 读入边(Vi, Vj)上的权 w */
    g->arcs[i][j]=w;                         /* 写入邻接矩阵 */
    g->arcs[j][i]=w;}
}
```

如果要建立无向图,可在以上算法中改变 w 的类型,并使输入值为 1 即可;如果要建立有向网络,只需将写入矩阵的两个语句中的后一个语句去除即可。以上算法中,如果邻接矩阵是一个稀疏矩阵,则存在存储空间浪费现象。

该算法的执行时间是 $O(n+n^2+e)$。通常 $e<<n^2$,所以算法的时间复杂度是 $O(n^2)$。

2. 邻接表存储方法

邻接表存储方法是一种顺序存储与链式存储相结合的存储方法。它包括两个部分,一部分是链表;另一部分是向量。这种方法只考虑非零元素,所以在图中顶点很多而边很少时,可以节省存储空间。

邻接表中,对于每个顶点 V_i,使用一个具有两个域的结构体数组来存储,这个数组称为顶点表。其中一个域称作顶点域(vertex),用来存放顶点本身的数据信息;而另一个域称作指针域(link),用来存放依附于该顶点的边所组成的单链表的表头结点的存储位置。邻接于 V_i的顶点 V_j链接成的单链表称为 V_i的邻接链表。邻接链表中的每个结点最多由 3 个域构成,一是邻接点域(adjvex),用来存放与 V_i相邻接的顶点 V_j的序号 j(可以是顶点 V_j在顶点表中所占数组单元的下标);二是顶点 V_i与顶点 V_j之间边(弧)的权值(data),如果是无权图,则该项内容省略;三是链域(next),用来将邻接链表中的结点链接在一起。

邻接表的存储结构可以用 C 语言描述如下:

```
#define VTXUNM n                             /* n 为图中顶点个数的最大可能值 */
#define ETXUNM e                             /* e 为图中边数的最大可能值 */
typedef char vextype;                        /* 定义顶点数据信息类型 */
struct arcnode {                             /* 邻接链表结点 */
```

```
    int adjvex;                          /* 邻接点域 */
    float data;                          /* 权值(无权图不含此项) */
    struct arcnode *nextarc;             /* 链域 */
};
typedef struct arcnode ARCNODE;
struct headnode {
    vextype vexdata;                     /* 顶点域 */
    ARCNODE *firstarc;                   /* 指针域 */
}adjlist[VTXUNM];                        /* 顶点表 */
```

对于无向图，V_i的邻接链表中每个结点都对应与 V_i相关联的一条边，第 i 个单链表的结点个数就是此结点的度，所以将无向图的邻接链表称为边表。对于有向图，V_i的邻接链表中每个结点都对应于以 V_i为起始点射出的一条边，其邻接表中第 i 个单链表的结点个数就是此结点的出度，所以有向图的邻接链表也称为出边表。有向图还有一种逆邻接表表示法，这种方法的 V_i邻接链表中的每个结点对应于以 V_i为终点的一条边，其邻接表中第 i 个单链表的结点个数就是此结点的入度，因而称这种邻接链表为入边表。

对于图 3-35(a)的有向图和图 3-36(a)的无向图，其邻接表存储结构分别如图 3-35(b)和图 3-36(b)所示。

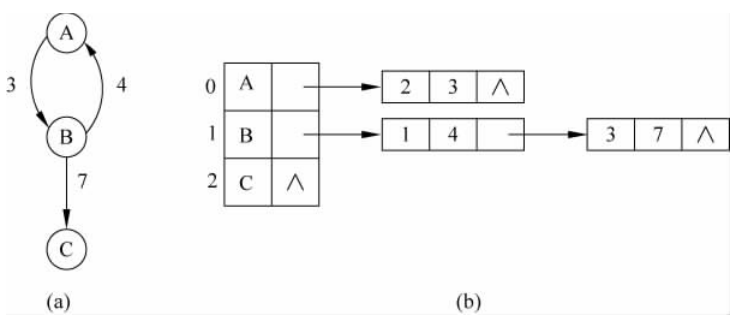


图 3-35　有向带权图及其邻接表

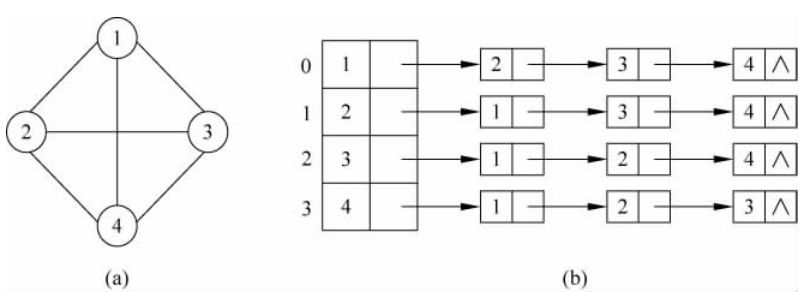


图 3-36　无向图及其邻接表

邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，但要判定任意两个顶点(V_i和 V_j)之间是否有边或弧相连，则需搜索第 i 个或第 j 个链表，因此不及邻接矩阵方便。

无论是无向图还是有向图，其邻接表的建立都比较简单，下面给出无向图邻接表的建立算法。

```
CREATADJLIST(struct headnode ga[ ])
```

```
{          /* 建立无向图的邻接表 */
    int i, j, k;
    ARCNODE *s;
    for(i=0; i<VTXUNM; i++)
    {
        ga[i].vexdata = getchar( );          /* 读入顶点信息和边表头指针初始化 */
        ga[i].firstarc = NULL;}
    for(k=0; k<ETXUNM; k++)
    {   /* 建立边表 */
        scanf("%d%d", &i, &j);               /* 读入边(Vi, Vj)的顶点序号 */
        s=malloc(sizeof(ARCNODE));           /* 生成邻接点序号为 j 的边表结点 *s */
        s->adjvex=j;
        s->nextarc=ga[i].firstarc;
        ga[i].firstarc=s;                    /* 将 *s 插入顶点 Vi 的边表头部 */
        s=malloc(sizeof(ARCNODE));           /* 生成邻接点序号为 i 的边表结点 *s */
        s->adjvex=i;
        s->nextarc=ga[j].firstarc;
        ga[j].firstarc=s;}                   /* 将 *s 插入顶点 Vj 的边表头部 */
}
```

如果要建立有向图的邻接表，则只需去除上述算法中“生成邻接点序号为 i 的边表结点 *s”部分，仅仅保留“生成邻接点序号为 j 的边表结点 *s”那一段语句组即可。若要建立网络的邻接表，只要在边表的每个结点中增加一个存储边上权值的数据域即可。该算法的执行时间是 O(n+e)。

邻接矩阵和邻接表是图中最常用的存储结构，它们各有所长，具体体现在以下几点：

(1) 一个图的邻接矩阵表示是唯一的，而其邻接表表示不唯一，这是因为邻接链表中的结点的链接次序取决于建立邻接表的算法和边的输入次序。

(2) 邻接矩阵表示中判定(V_i, V_j)或$<V_i, V_j>$是否是图中的一条边，只需判定矩阵中的第 i 行第 j 列的元素是否为零即可。而在邻接表中，需要扫描 V_i对应的邻接链表，最坏情况下需要的执行时间为 O(n)。

(3) 求图中边的数目时，使用邻接矩阵必须检测完整个矩阵之后才能确定，所消耗的时间为 $O(n^2)$；而在邻接表中，只需对每个边表中的结点个数计数便可确定。当 $e<<n^2$时，使用邻接表计算边的数目可以节省计算时间。

具体应用中选择哪种存储方法，主要考虑算法本身的特点和空间的存储密度来确定。

3.6.3　图的遍历

和树的遍历类似，从图中某一顶点出发访问图中其余的顶点，使每个顶点都被访问且仅被访问一次，这个过程就叫做图的遍历(traversing graph)。图的遍历算法是求解图的连通性、拓扑排序和关键路径等算法的基础。

图中的任一顶点都可能和其他顶点相邻接，所以图的遍历比树的遍历复杂得多。图中访问某个顶点之后，可能又会沿着某条路径回到该顶点上。为了避免对同一顶点的重复访问，在遍历图的过程，必须记下每个已访问过的顶点。为此，需要使用一个辅助数组 visited[n](n 为顶点数)来对顶点进行标识，它的初值设为 0，如果顶点 V_i被访问，则将 visited[i]置为 1，或者置为被访问时的次序号，否则保持为 0。

常用的图的遍历方法有深度优先搜索遍历和广度优先搜索遍历。下面以无向图为例进行讨论，有向图的情况与此类似。

1. 深度优先搜索遍历

对于一个图，按深度优先搜索遍历先后顺序得到的顶点序列称为该图的深度优先搜索(DFS, Depth-First Search)遍历序列，简称为 DFS 序列。图的深度优先搜索遍历类似于树的先序遍历，是树先序遍历的推广。一个图的 DFS 序列不一定唯一，它与算法、图的存储结构和初始出发点有关。当确定了有多个邻接点，并按邻接点的序号从小到大进行选择和指定初始出发点时，邻接矩阵作为存储结构得到的 DFS 序列是唯一的。使用邻接表作为存储结构时，由于图的邻接表表示不唯一，所以 DFSL 算法得到的 DFS 序列也不唯一，它取决于邻接表中边表结点的链接次序。

假设初始状态是图中所有顶点都未被访问，深度优先搜索的基本思想是：

(1) 首先访问图 G 的指定起始点 V_0，并进行标记。

(2) 从 V_0 出发，访问一个与 V_0 邻接的顶点 V_1，对 V_1 进行标记后，再从 V_1 出发，访问与 V_1 邻接且未被访问过的顶点 V_2，并进行标记。从 V_2 出发，重复上述过程，直到遇到一个所有与之邻接的顶点均被访问过的顶点为止。

(3) 沿着上述访问的次序，反向回退到尚有未被访问过的邻接点的顶点，从该顶点出发，重复步骤(2)、(3)，直到所有被访问过的顶点的邻接点都已被访问过为止；若图中尚有顶点未被访问过(非连通的情况下)，则另选图中的一个未被访问的顶点作为出发点，重复上述过程，直到图中所有顶点都被访问为止。

这种方法的特点是访问顶点的过程尽可能向纵深方向搜索，所以称为深度优先搜索遍历。显然，这种搜索方法具有递归的性质。设计具体算法时，首先要确定图的存储结构，下面以邻接表为例，讨论深度优先搜索法。

连通图 G(如图 3-37 (a)所示)的邻接表表示如图 3-37 (b)所示，以顶点 V_1 为起始点，按深度优先搜索遍历图中所有顶点，写出顶点的遍历序列。

求解过程是先访问 V_1，再访问与 V_1 邻接的 V_2，再访问 V_2 的第一个邻接点，因 V_1 已被访问过，则访问 V_2 的下一个邻接点 V_4，然后依次访问 V_8，V_5。这时，与 V_5 相邻接的顶点均已被访问，于是反向回到 V_8 去访问与 V_8 相邻接且尚未被访问的 V_6，接着访问 V_3，V_7，至此，全部顶点均被访问。相应的访问序列为 $V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow V_8 \rightarrow V_5 \rightarrow V_6 \rightarrow V_3 \rightarrow V_7$。

下面给出以邻接表作为存储结构的深度优先搜索递归算法 DFSL。

```
#define VTXUNM n                          /* n 为图中顶点个数的最大可能值 */
typedef char vextype;                     /*  定义顶点数据信息类型  */
struct arcnode {
    int adjvex;
    float data;
    struct arcnode *nextarc;
};
typedef struct arcnode ARCNODE;
struct headnode {
    vextype vexdata;
    ARCNODE *firstarc;
};
```

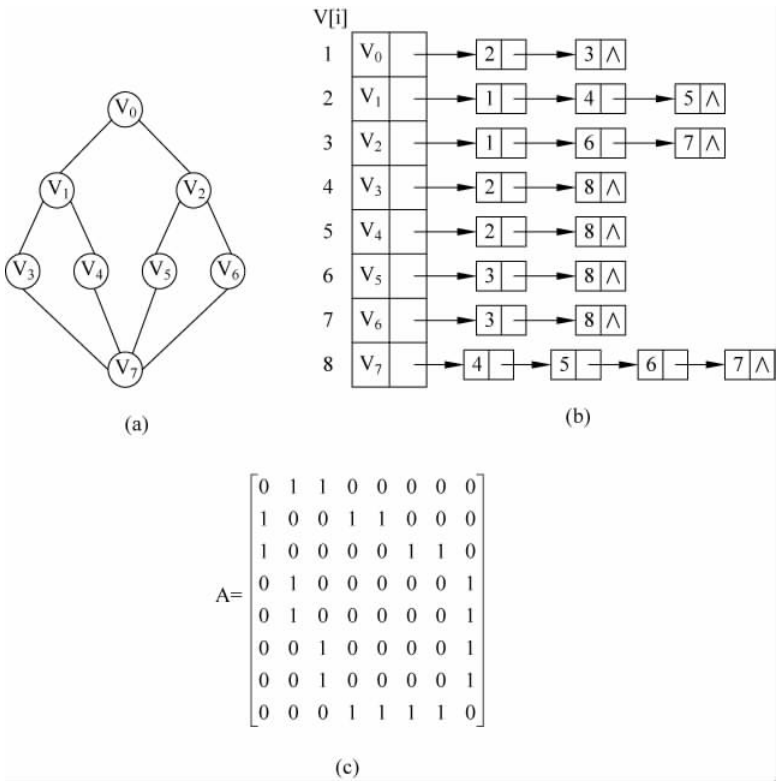


图 3-37　连通图 G、邻接表及其邻接矩阵

```
struct headnode G[VTXUNM + 1];
int visited[VTXUNM + 1];
void DFSL(struct headnode G[ ], int v) {
    ARCNODE * p;
    printf("%c->", G[v].vexdata);
    visited[v] = 1;
    p = G[v].firstarc;
    while (p!= NULL) {                          /* 当邻接点存在时 */
        if (visited[p->adjvex] == 0)
            DFSL(G, p->adjvex);
        p = p->nextarc;                         /* 找下一邻接点 */
    }
}
void traver(struct headnode G[ ]) {
    int v;
    for(v = 1;v <= VTXUNM;v++)
        visited[v] = 0;
    for(v = 1; v <= VTXUNM;v++)
        if(visited[v] == 0)DFSL(G, v);
}
```

如果采用深度优先搜索的非递归算法 DFSL，则程序如下：

```
#define VTXUNM n
```

```
void traver_ DFSL(struct headnode G[ ],int v) {
    int stack[VTXUNM];
    int top = -1;
    int i;
    ARCNODE *p;
    printf("%c->", G[v].vexdata);
    visited[v] = 1;
    top++;
    stack[top] = v;                              /*访问过的顶点进栈*/
    p = G[v].firstarc;
    while ((top!= -1)||(p!= NULL)) {
        while(p!= NULL) {
            if (visited[p->adjvex] == 1)
                p = p->nextarc;
            else {
                printf("%c->", G[p->adjvex].vexdata);
                visited[p->adjvex] = 1;
                top++;
                stack[top] = p->adjvex;
                p = G[p->adjvex].firstarc;
            }
        }
        if(top!= -1) {
            v = stack[top];
            top--;
            p = G[v].firstarc;
            p = p->nextarc;
        }
    }
}
```

因为搜索 n 个顶点的所有邻接点需要对边表各结点扫描一遍，而边表结点的数目为 2e，所以算法的时间复杂度为 O(2e+n)，空间复杂度为 O(n)。

选择邻接矩阵作为图的存储结构时，图 3-37(a)所示的连通图 G 的邻接矩阵表示如图 3-37(c)所示，其深度优先搜索遍历算法 DFSA 描述如下：

```
#define VTXUNM n                        /*n为图中顶点个数的最大可能值*/
typedef char vextype;                   /* 顶点的数据类型 */
typedef int adjtype;                    /* 顶点权值的数据类型 */
typedef struct {
  vextype vexs[VTXUNM];                 /* 顶点数组 */
  adjtype arcs[VTXUNM][VTXUNM];         /* 邻接矩阵 */
} graph;
graph g;                                /* g为全局变量 */
int visited[VTXUNM];
void DFSA (int i) {                     /* 从 Vi 出发深度优先搜索图 g,g用邻接矩阵表示 */
  int j;
  printf("node: %c\n", g.vexs[i]);      /* 访问出发点 Vi */
  visited[i] = 1;                       /* 标记 Vi 已被访问 */
  for(j = 0; j<n; j++)                  /* 依次搜索 Vi 的邻接点 */
```

```
    if ((g.arcs[i][j] == 1)&&(visited[j] == 0))
      DFSA(j);        /* 若 Vi 的邻接点 Vj 未被访问过,则从 Vj 出发进行深度优先搜索遍历 */
}                                    /* DFSA */
```

上述算法中,每进行一次 DFSA(i)的调用,for 循环中 v 的变化范围都是 0～n－1,而 DFSA(i)要被调用 n 次,所以算法的时间复杂度为 $O(n^2)$。因为是递归调用,需要使用一个长度为 n－1 的工作栈和长度为 n 的辅助数组,所以算法的空间复杂度为 O(n)。

2. 广度优先搜索遍历

对于一个图,按广度优先搜索遍历先后顺序得到的顶点序列称为该图的广度优先搜索(BFS, Breadth-First Search)遍历序列,简称为 BFS 序列。图的广度优先搜索遍历类似于树的按层次遍历。一个图的 BFS 序列不是唯一的,它与算法、图的存储结构和初始出发点有关。当确定了有多个邻接点时,按邻接点的序号从小到大进行选择和指定初始出发点后,以邻接矩阵作为存储结构得到的 BFS 序列是唯一的,而以邻接表作为存储结构得到的 BFS 序列并不唯一,它取决于邻接表中边表结点的链接次序。

假设初始状态是图中所有顶点都未被访问,BFS 方法从图中某一顶点 V_0 出发,先访问 V_0,然后访问 V_0 的各个未被访问过的邻接点,再分别从这些邻接点出发广度优先搜索遍历图,以此类推,直至图中所有已被访问的顶点的邻接点都被访问到。如果是非连通图,则选择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。

具体遍历步骤如下:

(1)访问 V_0。

(2)从 V_0 出发,依次访问 V_0 的未被访问过的邻接点 V_1,V_2,…,V_t。然后依次从 V_1,V_2,…,V_t 出发,访问各自未被访问过的邻接点。

(3)重复步骤(2),直到所有顶点的邻接点均被访问过为止。

在这种方法的遍历过程中,先被访问的顶点,其邻接点也先被访问,具有先进先出的特性,实现算法时,使用一个队列来保存每次已访问过的顶点,然后将队头顶点出列,去访问与它邻接的所有顶点,重复上述过程,直至队空。为了避免重复访问一个顶点,使用一个辅助数组 visited[n]来标记顶点的访问情况。

针对图 3-37 所示的连通图 G、邻接表及其邻接矩阵存储结构表示,假设从顶点 V_1 出发,按广度优先搜索法先访问 V_1,然后访问 V_1 的邻接点 V_2 和 V_3,再依次访问 V_2 和 V_3 的未被访问的邻接点 V_4、V_5、V_6 及 V_7,最后访问 V_4 的邻接点 V_8。遍历序列描述如下:

$$V_0 \rightarrow V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7$$

下面给出以邻接矩阵为存储结构时的广度优先搜索遍历算法 BFSA。

```
#define VTXUNM n
graph g;                             /* g 为全局变量 */
int visited[VTXUNM];
void BFSA(int k)
{                                    /* 从 Vk 出发广度优先搜索遍历图 g,g 用邻接矩阵表
示 */
    int queue[VTXUNM];
    int rear = VTXUNM - 1; front = VTXUNM - 1;  /* queue 置为空队 */
    int i, j;
    printf("%c\n", g.vexs[k]);       /* 访问出发点 Vk */
```

```
    visited[k] = 1;                    /* 标记 Vk 已被访问 */
    rear++;
    queue[rear] = k;                   /* 访问过的顶点序号入队 */
    while(front!= rear)
    {                                  /* 队非空时执行下列操作 */
        front++;
        i = queue[front]               /* 队头元素序号出队 */
        for(j = 0; j < n; j++)
            if((g.arcs[i][j] == 1)&&(visited[j]!= 1))
              {
                printf("%c\n", g.vexs[j]);          /* 访问 Vi 未被访问的邻接点 Vj */
                visited[j] = 1;
                rear++;
                queue[rear] = j;       /* 访问过的顶点入队 */
                }
    }
}                                      /* BFSA */
```

当选择邻接表作为图的存储结构时，图 3-37(a)所示的连通图 G 的广度优先搜索遍历算法 BFSL 描述如下：

```
#define VTXUNM n
void BFSL(struct headnode G[ ], int v)
struct headnode G[VIXUNM + 1];
int Visited[VIXUNM + 1];
{
    int queue[VTXUNM];
    int rear = -1; front = -1;      /* queue 置为空队 */
    int i;
    ARCNODE *p;
    printf(''%d->'', G[v].vexdata);
    visited[v] = 1;
    rear++;
    queue[rear] = v;                /* 访问过的顶点进队列 */
    while (rear!= front)
    {
        front++;
        v = queue[front];
        p = G[v].firstarc;
        while(p!= NULL) {
            if (visited[p->adjvex] == 0) {
                printf(''%d->'', G[p->adjvex].vexdata);
                visited[p->adjvex] = 1;
                rear++;
                queue[rear] = p->adjvex;
            }
            p = p->nextarc;
        }
    }
}                                   /* BFSL */
```

对于有 n 个顶点和 e 条边的连通图，BFSA 算法的 while 循环和 for 循环都需执行 n 次，所以 BFSA 算法的时间复杂度为 $O(n^2)$，同时 BFSA 算法使用了两个长度均为 n 的队列

和辅助标志数组，所以空间复杂度为 O(n)；BFSL 算法的外 while 循环要执行 n 次，而内 while 循环执行次数总计是边表结点的总个数 2e，所以 BFSL 算法的时间复杂度为 O(n+2e)；同时，BFSL 算法也使用了两个长度均为 n 的队列和辅助标志数组，所以空间复杂度为 O(n)。

3.6.4　图的应用

1. 生成树和最小生成树

图论中，树是指一个无回路存在的连通图。一个连通图 G 的生成树指的是一个包含了 G 的所有顶点的树。对于一个有 n 个顶点的连通图 G，其生成树包含了 n－1 条边，从而生成树是 G 的一个极小连通的子图。所谓极小指该子图具有连通所需的最小边数，若去掉一条边，该子图就变成了非连通图；若任意增加一条边，该子图就有回路产生。

当给定一个无向连通图 G 后，可以从 G 的任意顶点出发，作一次深度优先搜索或广度优先搜索来访问 G 中的 n 个顶点，并将顺次访问的两个顶点之间的路径记录下来。这样，G 中的 n 个顶点和从初始点出发顺次访问余下的 n－1 个顶点所经过的 n－1 条边就构成了 G 的极小连通子图，也就是 G 的一棵生成树。

通常，将深度优先搜索得到的生成树称为深度优先搜索生成树，简称为 DFS 生成树；而将广度优先搜索得到的生成树称为广度优先搜索生成树，简称为 BFS 生成树。

对于前面所给的 DFSA 和 BFSA 算法，只需在 if 语句中的 DFSA 调用语句前或 if 语句中加入将(v_i, v_j)打印出来的语句，即构成相应的生成树算法。

连通图的生成树不是唯一的，它取决于遍历方法和遍历的起始顶点。遍历方法确定后，从不同的顶点出发进行遍历，可以得到不同的生成树。对于非连通图，每个连通分量中的顶点集和遍历时走过的边一起构成一棵生成树，这些连通分量的生成树组成非连通图的生成森林。算法实现时，可通过多次调用由 DFSA 或 BFSA 构成的生成树算法求出非连通图中各连通分量对应的生成树，这些生成树构成了非连通图的生成森林。使用 DFSA 构成的生成树算法和 BFSA 构成的生成树算法，对图 3-37(a)所示连通图 G 从顶点 1 开始进行遍历得到的深度优先生成树和广度优先生成树分别如图 3-38(a)、(b)所示。

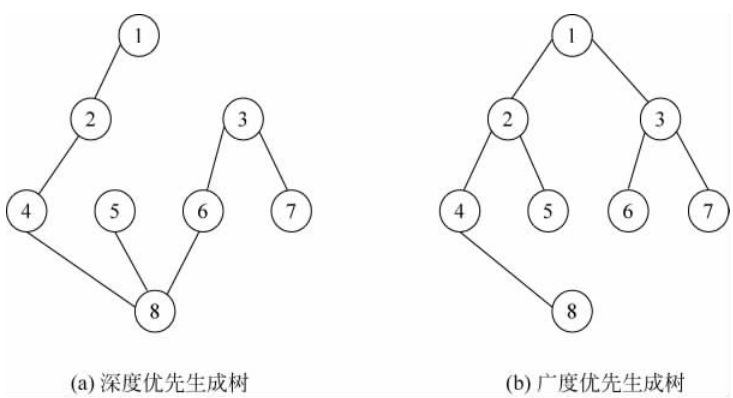


图 3-38　G 从 V_1 出发的两种生成树

图 3-39 所示为 G(图 3-33(a))的深度优先生成森林，它由 3 棵深度优先生成树组成。

对一个连通网络构造生成树时，可以得到一个带权的生成树。把生成树各边的权值总

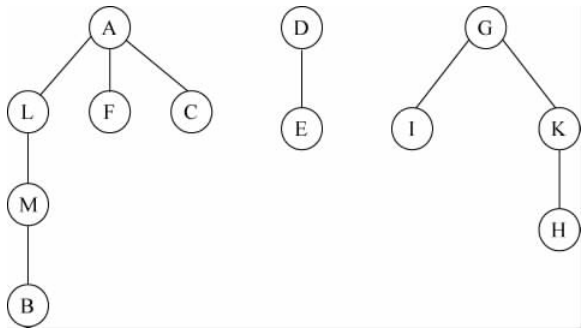


图 3-39　图 G 的深度优先生成森林

和作为生成树的权，而具有最小权值的生成树构成了连通网络的最小生成树。也就是说，构造最小生成树就是在给定 n 个顶点所对应的权矩阵（代价矩阵）的条件下，给出代价最小的生成树。

最小生成树的构造有实际应用价值。例如，要在 n 个城市之间建立通信网络，则连通 n 个城市只需 n－1 条线路。若以 n 个城市做图的顶点，n－1 条线路做图的边，则该图的生成树就是可行的建造方案。而不同城市之间建立通信线路需要一定的花费（相当于边上的权），所以对 n 个顶点的连通网可以建立许多不同的生成树，每棵生成树都可以是一个通信网，当然希望选择一个总耗费最小的生成树，即最小代价生成树。

例如，图 3-40(a)是个连通网，它的最小生成树如图 3-40(b)所示。

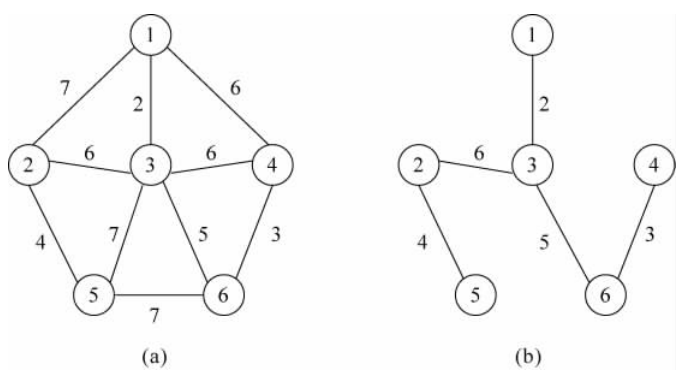


图 3-40　连通网及其最小生成树

构造最小生成树的算法有多种，大多数算法都利用了最小生成树的一个性质，简称为 MST 性质。MST 性质指出，假设 G＝(V，E)是一个连通网络，U 是 V 中的一个真子集，若存在顶点 u∈U 和顶点 v∈V－U 的边(u，v)是一条具有最小权的边，则必存在 G 的一棵最小生成树包括这条边(u，v)。

MST 性质可用反证法加以证明，假设 G 中的任何一棵最小生成树 T 都不包含(u，v)，其中 u∈U 和 v∈V－U。由于 T 是最小生成树，所以必然有一条边(u′，v′)(其中 u′∈U 和 v′∈V－U)连接两个顶点集 U 和 V－U。当(u，v)加入到 T 中时，T 中必然存在一条包含了(u，v)的回路，如图 3-41 所示。如果在 T 中保留(u，v)，去掉(u′，v′)，则得到另一棵生成树 T′。因为(u，v)的权小于(u′，v′)的权，故 T′的权小于 T 的权，这与假设矛盾，因此

MST 性质得证。

下面介绍构造最小生成树的两种常用算法，Prim(普里姆)算法和 Kruskal(克鲁斯卡尔)算法。

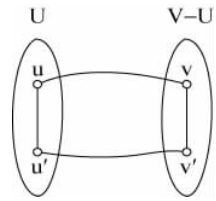


图 3-41　含有(u,v)的回路

1) Prim 算法

设 G(V, E)是有 n 个顶点的连通网络，T=(U, TE)是要构造的生成树，初始时 U={Φ}，TE={Φ}。首先，从 V 中取出一个顶点 u_0放入生成树的顶点集 U 中作为第一个顶点，此时 T=({u_0}, {Φ})；然后从 u∈U，v∈V−U 的边(u, v)中找一条代价最小的边(u*，v*)，将其放入 TE 中并将 v* 放入 U 中，此时 T=({u_0, v*}, {(u_0, v*)})；继续从 u∈U，v∈V−U 的边(u, v)中找一条代价最小的边(u*，v*)，将其放入 TE 中并将 v* 放入 U 中，直到 U=V 为止。这时 T 的 TE 中必有 n−1 条边，构成所要构造的最小生成树。

显然，Prim 算法的关键是如何找到连接 U 和 V−U 的最短边(代价最小边)来扩充 T。设当前生成树 T 中已有 k 个顶点，则 U 和 V−U 中可能存在的边数最多为 k(n−k)条，在如此多的边中寻找一条代价最小的边是困难的。

注意在相邻的寻找最小代价的边的过程中，有些操作具有重复性，所以可通过将前一次寻找所得到的最小边存储起来，然后与新找到的边进行比较，如果新找到的边比原来已找到的边短，则用新找到的边代替原有的边，否则保持不变。为此设立以下边的存储结构。

```
typedef struct {
    int fromvex, endvex;                    /* 边的起点和终点 */
    float length;                           /* 边的权值 */
} edge;
edge T[n-1];
float dist[n][n];                           /* 连通网络的带权邻接矩阵 */
```

相应的 Prim 算法描述如下：

```
Prim(int i)
{          /* i 给出选取的第一个顶点的下标,最终结果保存在 T[n-1]数组中 */
    int j, k, m, v, min, max=100000;
    float d;
    edge e;
    v=i;                                    /* 将选定顶点送入中间变量 */
  for(j=0; j<=n-2; j++)
    {   /* 构造第一个顶点 */
        T[j].fromvex=v;
        if(j>=v)
        {
            T[j].endvex=j+1;
            T[j].length=dist[v][j+1];}
        else
        {
            T[j].endvex=j;
            T[j].length=dist[v][j];}
    }
    for(k=0; k<n-1; k++) {                  /* 求第 k 条边 */
```

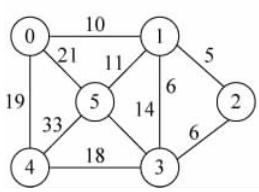
```
            min = max;
            for(j = k; j < n - 1; j++)        /* 找出最短的边并将最短边的下标记录在 m 中 */
              if(T[j].length < min) {
              min = T[j].length;
              m = j;}
            e = T[m]; T[m] = T[k]; T[k] = e; /* 将最短的边交换到 T[k]单元 */
            v = T[k].endvex;                 /* v 中存放新找到的最短边在 V - U 中的顶点 */
            for(j = k + 1; j < n - 1; j++)
            {/* 修改所存储的最小边集 */
              d = dist[v][T[j].endvex];
              if(d < T[j].length)
              {
              T[j].length = d;
              T[j].fromvex = v;}
            }
        }
    } /* Prim */
```

以上算法中构造第一个顶点所需的时间是 O(n)，求 k 条边的时间大约是

$$\sum_{k=0}^{n-2}\left(\sum_{j=k}^{n-2}O(1)+\sum_{j=k+1}^{n-2}O(1)\right)\approx 2\sum_{k=0}^{n-2}\sum_{j=k}^{n-2}O(1) \tag{3-5}$$

其中，O(1)表示某一正常数 C，所以上述公式的时间复杂度是 $O(n^2)$。

下面结合图 3-42 所示的例子来观察算法(3-4)的工作过程。设选定的第一个顶点为 2。首先将顶点值 2 写入 T[i].fromvex，并将其余顶点值写入相应的 T[i].endvex，然后从 dist 矩阵中取出第 2 行写入相应的 T[i].length 中，得到图 3-43(a)；在该图中找出具有最小权值的边(2，1)，将其交换到下标值为 0 的单元中，然后从 dist 矩阵中取出第 1 行的权值与相应的 T[i].length 作比较，若取出的权值小于相应的 T[i].length，则进行替换，否则保持不变。由于边(2，0)和(2，5)的权值大于边(1，0)和(1，5)的权值，进行相应的替换可得到图 3-43(b)；在该图中找出具有最小权值的边(2，3)，将其交换到下标值为 1 的单元中，然后从 dist 矩阵中取出第 3 行的权值与相应的 T[i].length 作比较，可见边(3，4)的权值小于边(2，4)的权值，故进行相应的替换得到图 3-43(c)；在该图中找出具有最小权值的边(1，0)，因其已在下标为 2 的单元中，故交换后仍然保持不变，然后从 dist 矩阵中取出第 0 行的权值与相应的 T[i].length 作比较，可见边(0，4)和(0，5)的权值大于边(3，4)和(1，5)的权值，故不进行替换，得到图 3-43(d)；在该图中找出具有最小权值的边(1，5)，将其交换到下标值为 3 的单元中，然后从 dist 矩阵中取出第 5 行的权值与相应的 T[i].length 作比较；因边(5，4)的权值大于边(3，4)的权值，故不替换，得到图 3-43(e)。至此整个算法结束，得出了如图 3-43(f)所示的最小生成树。



$$\begin{bmatrix} \infty & 10 & \infty & \infty & 19 & 21 \\ 10 & \infty & 5 & 6 & \infty & 11 \\ \infty & 5 & \infty & 6 & \infty & \infty \\ \infty & 6 & 6 & \infty & 18 & 14 \\ 19 & \infty & \infty & 18 & \infty & 33 \\ 21 & 11 & \infty & 14 & 33 & \infty \end{bmatrix}$$

图 3-42　一个网络及其邻接矩阵

下 标	0	1	2	3	4
fromvex	2	2	2	2	2
endvex	0	1	3	4	5
length	∞	(5)	6	∞	∞

(a) 初始化后的T数组

下 标	0	1	2	3	4
fromvex	2	1	2	2	1
endvex	1	0	3	4	5
length	5	10	(6)	∞	11

(b) 找出最短边(2,1)调整后的T数组

下 标	0	1	2	3	4
fromvex	2	2	1	3	1
endvex	1	3	0	4	5
length	5	6	(10)	18	11

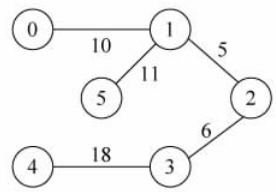
(c) 找出最短边(2,3)并调整后

下 标	0	1	2	3	4
fromvex	2	2	1	3	1
endvex	1	3	0	4	5
length	5	6	10	18	(11)

(d) 找出最短边(1,0)并调整后

下 标	0	1	2	3	4
fromvex	2	2	1	1	3
endvex	1	3	0	5	4
length	5	6	10	11	18

(e) 找出最短边(1,5)并调整后



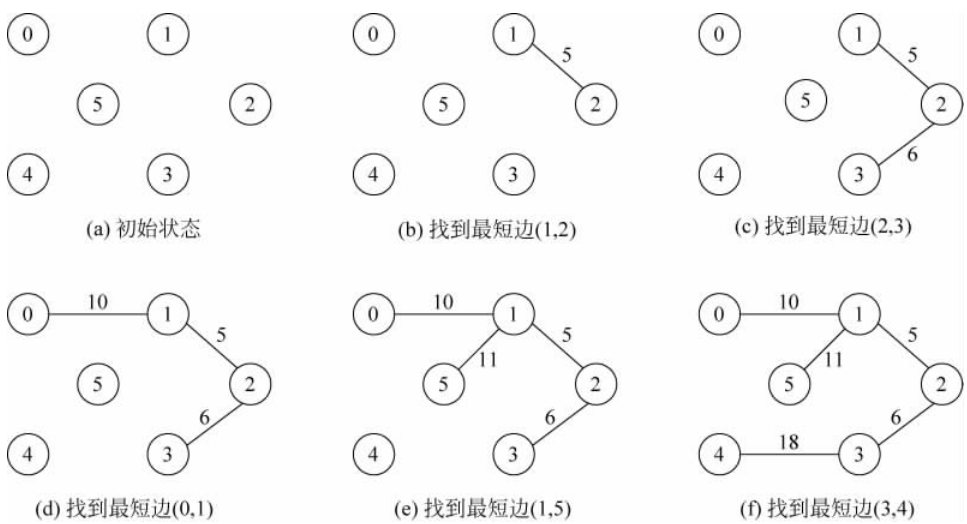
(f) 最小生成树

图 3-43　T 数组变化情况及最小生成树

2）Kruskal 算法

Kruskal 算法是从另一条途径来求网络的最小生成树。设 G＝(V，E)是一个有 n 个顶点的连通图，令最小生成树的初值状态为只有 n 个顶点而无任何边的非连通图 T＝(V，{Φ})，此时图中每个顶点自成一个连通分量。按照权值递增的顺序依次选择 E 中的边，若该边依附于 T 中两个不同的连通分量，则将此边加入 TE 中，否则舍去此边而选择下一条代价最小的边，直到 T 中所有顶点都在同一连通分量上为止。这时的 T 便是 G 的一棵最小生成树。

对于图 3-43 所示的网络，按 Kruskal 算法构造最小生成树的过程如图 3-44 所示。



(a) 初始状态　(b) 找到最短边(1,2)　(c) 找到最短边(2,3)

(d) 找到最短边(0,1)　(e) 找到最短边(1,5)　(f) 找到最短边(3,4)

图 3-44　Kruskal 算法构造最小生成树的过程

图 3-44(c)中选择最短边(2，3)时，也可以选择边(1，3)，这样所构造出的最小生成树是不同的，即最小生成树的形式不唯一，但权值的总和相同。选择了最短边(2，3)之后，在图 3-44(d)中首先选择边(1，3)，因其顶点在同一分量上，故舍去这条边而选择下一条代价最小的边。在图 3-44(f)中也首先选择边(3，5)，但因顶点 3 和 5 在同一个分量上，故舍去此边而选择下一条代价最小边(3，4)。

Kruskal 算法中，每次都要选择所有边中最短的边，若用邻接矩阵实现，则每找一条最短的边就需要对整个邻接矩阵扫描一遍。这样，整个算法复杂度太高，而使用邻接表时，由于每条边都被连接两次，这也使寻找最短边的计算时间加倍，所以采用以下的存储结构来对图中的边进行表示。

```
typedef struct {
    int fromvex, endvex;   /* 边的起点和终点 */
    float length;          /* 边的权值 */
    int sign;              /* 该边是否已选择过的标志信息 */
} edge;
edge T[e];                 /* e 为图中的边数 */
int G[n];                  /* 判断该边的两个顶点是不是在同一个分量上的数组,n 为顶点数 */
```

Kruskal 算法中，如何判定所选择的边是否在同一个分量上，是整个算法的关键和难点。为此，设置一个 G 数组，利用 G 数组的每一个单元中存放一个顶点信息的特性，通过判断两个顶点对应单元的信息是否相同来判定所选择的边是否在同一个分量上。具体算法如下：

```
Kruskal(int n, int e)
{          /* n 表示图中的顶点数目,e 表示图中的边数目 */
    int i, j, k, l, min, t;
    for(i = 0; i <= n - 1; i++)        /* 数组 G 置初值 */
      G[i] = i;
    for(i = 0; i <= e - 1; i++) {      /* 输入边信息 */
      scanf("%d%d%f", &T[i].fromvex, &T[i].endvex, &T[i].length);
      T[i].sign = 0;}
    j = 0;
    while(j < n - 1)
      {
        min = 1000;
        for(i = 0; i <= e - 1; i++)
          {                            /* 寻找最短边 */
            if(T[i].sign == 0)
            if(T[i].length < min)
            {
              k = T[i].fromvex;
              l = T[i].endvex;
              T[i].sign = l;}
      if(G[k] == G[l])T[i].sign = 2; /* 在同一分量上舍去 */
      else {
        j++;
        for(t = 0; t < n; t++)         /* 将最短边的两个顶点并入同一分量 */
          if(G[t] == l)G[t] = k;}
```

```
    }
  }
}                                    /* Kruskal */
```

如果边的信息是按权值从小到大依次存储到 T 数组中，则 Kruskal 算法的时间复杂度约为 O(e)。一般情况下，Kruskal 算法的时间复杂度约为 O(elge)，与网中的边数有关，故适合于求边稀疏网络的最小生成树；而 Prim 算法的时间复杂度为 $O(n^2)$，与网中的边数无关，适合于边稠密网络的最小生成树。

2. 最短路径

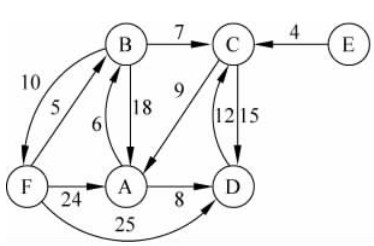
一个实际的交通网络在计算机中可用图的结构来表示。这类问题中经常考虑的问题有两个，一是两个顶点之间是否存在路径；二是在有多条路径的条件下，哪条路径最短。由于交通网络中的运输路线往往有方向性，因此将以有向网络进行讨论，无向网络的情况与此相似。讨论中，习惯上称路径的开始点为源点(Source)，路径的最后一个顶点为终点(Destination)，而最短路径意味着沿路径的各边权值之和为最小。求最短路径时，为方便起见，规定邻接矩阵中某一顶点到自身的权值为 0，即当 i=j 时，dist[i][j]=0。

最短路径问题的研究分为两种情况，一是从某个源点到其余各顶点的最短路径；二是每一对顶点之间的最短路径。

1) 从某个源点到其余各顶点的最短路径

迪杰斯特拉(Dijkstra)通过对大量的图中某个源点到其余顶点的最短路径的顶点构成集合和路径长度之间关系的研究发现，若按长度递增的次序来产生源点到其余顶点的最短路径，则当前正要生成的最短路径除终点外，其余顶点的最短路径已生成，即设 A 为源点，U 为已求得的最短路径的终点的集合(初态时为空集)，则下一条长度较长的最短路径(设它的终点为 X)或者是弧(A, X)或者是中间只经过 U 集合中的顶点，最后到达 X 的路径。例如，在图 3-45 中要生成从 F 点到其他顶点的最短路径。首先应找到最短的路径 F→B，然后找到最短的路径 F→B→C。这里除终点 C 以外，其余顶点的最短路径 F→B 已生成。

迪杰斯特拉提出的按路径长度递增次序来产生源点到各顶点的最短路径的算法思想是，对有 n 个顶点的有向连通网络 G=(V, E)，首先从 V 中取出源点 u_0放入最短路径顶点集合 U 中，这时的最短路径网络 S=($\{u_0\}$, $\{\phi\}$)；然后从 u∈U 和 v∈V−U 中找一条代价最小的边(u*, v*)加入到 S 中，此时 S=($\{u_0, v*\}$, $\{(u_0, v*)\}$)。每往 U 中增加一个顶点，都需要对 V−U 中各顶点的权值进行一次修正。如果加进 v* 作为中间顶点，使得从 u_0到其他属于 V−U 的顶点 v_i的路径比不加 v* 时最短，则修改 u_0到 v_i的权值，即以(u_0, v*)的权值加上(v*, v_i)的权值来代替原(u_0, v_i)的权值，否则不修改 u_0到 v_i的权值。接着再从权值修正后的 V−U 中选择最短的边加入 S 中，如此反复，直到 U=V 为止。



F→B	5
F→B→C	12
F→B→C→A	21
F→D	25
F→E	无路径

图 3-45　有向网络 G 和 F 到其他顶点的最短距离

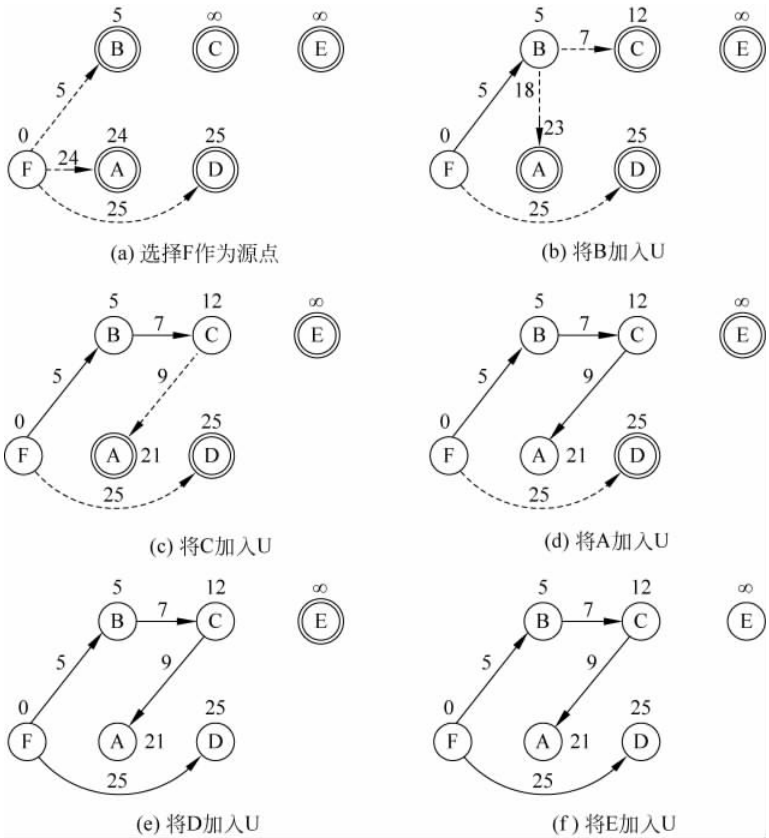


图 3-46　Dijkstra 算法求最短路径示例

对图 3-45 中的有向网络按以上算法思想处理，所求得的从源点 F 到其余顶点的最短路径的过程如图 3-46 所示。其中，单圆圈表示 U 中的顶点，而双圆圈表示 V－U 中的顶点。连接 U 中两个顶点的有向边用实线表示，连接 U 和 V－U 中两个顶点的有向边用虚线表示。圆圈旁的数字为源点到该顶点当前的距离值。

初始时，S 中只有一个源点 F，它到 V－U 中各顶点的路径如图 3-46(a)所示；选择图 3-46(a)中最小代价边(F，B)，同时由于路径(F，A)大于(F，B，A)和(F，C)大于(F，B，C)，进行相应调整可得到图 3-46(b)；选择图 3-46(b)中的最小代价边(B，C)，同时由于(F，B，A)大于(F，B，C，A)，进行相应调整可得到图 3-46(c)；选择图 3-46(c)中最小代价边(C，A)即可得到图 3-46(d)；选择图 3-46(d)中最小代价边(F，D) 即可得到图 3-46(e)D；最后选择(F，E)即可得到图 3-46(f)。

计算机上实现此算法时，需要设置一个用于存放源点到其他顶点的最短距离数组 D[n]，以便于从其中找出最短路径；因为不仅希望得到最短路径长度，而且也希望能给出最短路径具体经过那些顶点的信息，所以设置一个路径数组 p[n]，其中 p[i]表示从源点到达顶点 i 时，顶点 i 的前趋顶点；为了防止对已经生成的最短路径进行重复操作，使用一个标识数组 s[n]来记录最短路径生成情况，若 s[i]＝1 表示源点到顶点 i 的最短路径已产生，而 s[i]＝0 表示最短路径还未产生。当顶点 A，B，C，D，E，F 对应标号 0，1，2，3，4，5 时，具体算法描述如下：

```
float D[n];
int p[n], s[n];
Dijkstra(int v, float dist[][])                    /* 求源点 v 到其余顶点的最短路径及长度 */
{   int i, j, k, v1, min, max = 10000, pre;        /* Max 中的值用以表示 dist 矩阵中的值∞ */
    v1 = v;
    for( i = 0; i < n; i++)                        /* 各数组进行初始化 */
    {   D[i] = dist[v1][i];
        if( D[i] != max )  p[i] = v1 + 1;
        else               p[i] = 0;
        s[i] = 0;
    }
    s[v1] = 1;                                     /* 将源点送 U */
    for( i = 0; i < n - 1; i++)                    /* 求源点到其余顶点的最短距离 */
    {   min = 10001;                               /* min > max,以保证值为∞的顶点也能加入 U */
        for( j = 0; j < n - 1; j++)
            if ( ( !s[j] )&&(D[j]< min) )          /* 找出到源点具有最短距离的边 */
            {   min = D[j];
                k = j;
            }
        s[k] = 1;                                  /* 将找到的顶点 k 送入 U */
        for(j = 0; j < n; j++)
            if ( (!s[j])&&(D[j]> D[k] + dist[k][j]) )     /* 调整 V - U 中各顶点的距离值 */
            {   D[j] = D[k] + dist[k][j];
                p[j] = k + 1;                      /* k 是 j 的前趋 */
            }
    }                                              /* 所有顶点已扩充到 U 中 */
    for( i = 0; i < n; i++)
    {   printf(" %f %d ", D[i], i);
        pre = p[i];
        while ((pre!= 0)&&(pre!= v + 1))
        {   printf (" <- %d ", pre - 1);
            pre = p[pre - 1];
        }
        printf(" <- %d ", v);
    }
} /* Dijkstra */
```

对图 3-45 中的有向网络 G，以 F 点为源点，执行上述算法时，D、p、s 数组的变化状况如表 3-1 所示。

表 3-1　Dijkstra 算法动态执行情况

循环	U	k	D[0],…, D[5]	p[0],…, p[5]	s[0],…, s[5]
初始化	{F}	——	24 5 max 25 max 0	6 6 0 6 0 6	0 0 0 0 0 1
1	{F, B}	1	23 5 12 25 max 0	2 6 2 6 0 6	0 1 0 0 0 1
2	{F,B,C}	2	21 5 12 25 max 0	3 6 2 6 0 6	0 1 1 0 0 1
3	{F,B,C,A}	0	21 5 12 25 max 0	3 6 2 6 0 6	1 1 1 0 0 1
4	{F,B,C,A,D}	3	21 5 12 25 max 0	3 6 2 6 0 6	1 1 1 1 0 1
5	{F,B,C,A,D,E}	4	21 5 12 25 max 0	3 6 2 6 0 6	1 1 1 1 1 1

打印输出的结果为

```
21     0←2←1←5
5      1←5
12     2←1←5
25     3←5
10000  4←5
0      5←5
```

Dijkstra 算法的时间复杂度为 $O(n^2)$，占用的辅助空间是 $O(n)$。

2）每一对顶点之间的最短路径

求一个有 n 个顶点的有向网络 G=(V, E)中的每一对顶点之间的最短路径，可以依次把有向网络的每个顶点作为源点，重复执行 n 次 Dijkstra 算法，从而得到每对顶点之间的最短路径。这种方法的时间复杂度为 $O(n^3)$。弗洛伊德(Floyd)于 1962 年提出了解决这一问题的另一种算法，它形式比较简单，易于理解，而时间复杂度同样为 $O(n^3)$。

Floyd 算法根据给定有向网络的邻接矩阵 dist[n][n]求顶点 v_i到顶点 v_j的最短路径。这一算法的基本思想是假设 v_i和 v_j之间存在一条路径，但这并不一定是最短路径，试着在 v_i和 v_j之间增加一个中间顶点 v_k。

若增加 v_k后的路径(v_i, v_k, v_j) 比(v_i, v_j)短，则以新的路径代替原路径，并且修改 dist[i][j]的值为新路径的权值；若增加 v_k后的路径比(v_i, v_j)更长，则维持 dist[i][j]不变。在修改后的 dist 矩阵中，另选一个顶点作为中间顶点，重复以上操作，直到除 v_i和 v_j顶点的其余顶点都做过中间顶点为止。对初始的邻接矩阵 dist[n][n]，依次以顶点 v_1, v_2, …, v_n为中间顶点实施以上操作时，将递推地产生出一个矩阵序列 $dist^{(k)}[n][n](k=0, 1, 2, \cdots, n)$。这里初始邻接矩阵 dist[n][n]看作 $dist^{(0)}[n][n]$，它给出每一对顶点之间的直接路径的权值；$dist^{(k)}[n][n](1\leqslant k<n)$给出了中间顶点的序号不大于 k 的最短路径长度，而 $dist^{(n)}[n][n]$给出了每一对顶点之间的最短路径长度。为了给出每一对顶点之间最短路径所经过的具体路径，可用一个 path 矩阵来记录具体路径。$path^{(0)}$给出了每一对顶点之间的直接路径，$path^{(n)}$给出了每一对顶点之间的最短路径，path 矩阵中每个元素 path[i][j]所保存的值是顶点 v_i到顶点 v_j时 v_j的前趋顶点。

为了在算法中始终保持初始邻接矩阵 dist[n][n]中的元素值不变，可以设置一个 A[n][n]矩阵保存每步所求得的所有顶点对之间的当前最短路径长度。这样可给出以下算法。

```
intpath[n][n];                          /* 路径矩阵 */
Floyd(float A[ ][n], dist[ ][n])        /* A 是路径长度矩阵, dist 是有向网络 G 的带权邻接矩阵 */
{   int i, j, k, next, max = 10000;
    for (i = 0; i < n; i++)             /* 设置 A 和 path 的初值 */
        for (j = 0; j < n; j++)
        {   if (dist[i][j] != max )  path[i][j] = i + 1;          /* i 是 j 的前趋 */
            else                     path[i][j] = 0;
            A[i][j] = dist[i][j];
        }
    for (k = 0; k < n; k++)             /* 以 0, 1, …, n-1 为中间顶点执行 n 次 */
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
```

```
            if (A[i][j]>(A[i][k] + A[k][j]))
            {   A[i][j] = A[i][k] + A[k][j];/* 修改路径长度 */
                path[i][j] = path[k][j];    /* 修改路径 */
            }
    for (i = 0; i < n; i++)          /* 输出所有顶点对 i, j 之间最短路径的长度和路径 */
        for (j = 0; j < n; j++)
        {   printf ( " %f %d ", A[i][j], j);
            pre = path[i][j];
            while ((pre!= 0)&&(pre!= i + 1)) {
                printf ("<- %d ", pre - 1);
                pre = path[i][pre - 1];
            }
            printf ("<- %d\n ", i);
        }
} /* Floyd */
```

对图 3-45 中的有向网络 G 执行以上算法，矩阵 A 和 path 的变化状况如下所示。

$$A^{(0)}=\begin{bmatrix}0 & 6 & \infty & 8 & 10000 & \infty\\ 18 & 0 & 7 & \infty & \infty & 10\\ 9 & \infty & 0 & 15 & \infty & \infty\\ \infty & \infty & 12 & 0 & \infty & \infty\\ \infty & \infty & 4 & \infty & 0 & \infty\\ 24 & 5 & \infty & 25 & \infty & 0\end{bmatrix}\quad path^{(0)}=\begin{bmatrix}1 & 1 & 0 & 1 & 0 & 0\\ 2 & 2 & 2 & 0 & 0 & 2\\ 3 & 0 & 3 & 3 & 0 & 0\\ 0 & 0 & 4 & 4 & 0 & 0\\ 0 & 0 & 5 & 0 & 5 & 0\\ 6 & 6 & 0 & 6 & 0 & 6\end{bmatrix}$$

$$A^{(1)}=\begin{bmatrix}0 & 6 & \infty & 8 & \infty & \infty\\ 18 & 0 & 7 & 26 & \infty & 10\\ 9 & 15 & 0 & 15 & \infty & \infty\\ \infty & \infty & 12 & 0 & \infty & \infty\\ \infty & \infty & 4 & \infty & 0 & \infty\\ 24 & 5 & \infty & 25 & \infty & 0\end{bmatrix}\quad path^{(1)}=\begin{bmatrix}1 & 1 & 0 & 1 & 0 & 0\\ 2 & 2 & 2 & 1 & 0 & 2\\ 3 & 1 & 3 & 3 & 0 & 0\\ 0 & 0 & 4 & 4 & 0 & 0\\ 0 & 0 & 5 & 0 & 5 & 0\\ 6 & 6 & 0 & 6 & 0 & 6\end{bmatrix}$$

$$A^{(2)}=\begin{bmatrix}0 & 6 & 13 & 8 & \infty & 16\\ 18 & 0 & 7 & 26 & \infty & 10\\ 9 & 15 & 0 & 15 & \infty & 25\\ \infty & \infty & 12 & 0 & \infty & \infty\\ \infty & \infty & 4 & \infty & 0 & \infty\\ 23 & 5 & 12 & 25 & \infty & 0\end{bmatrix}\quad path^{(2)}=\begin{bmatrix}1 & 1 & 2 & 1 & 0 & 2\\ 2 & 2 & 2 & 1 & 0 & 2\\ 3 & 1 & 3 & 3 & 0 & 2\\ 0 & 0 & 4 & 4 & 0 & 0\\ 0 & 0 & 5 & 0 & 5 & 0\\ 2 & 6 & 2 & 6 & 0 & 6\end{bmatrix}$$

$$A^{(3)}=\begin{bmatrix}0 & 6 & 13 & 8 & 10000 & 16\\ 16 & 0 & 7 & 22 & \infty & 10\\ 9 & 15 & 0 & 15 & \infty & 25\\ 21 & 27 & 12 & 0 & \infty & 37\\ 13 & 19 & 4 & 19 & 0 & 29\\ 21 & 5 & 12 & 25 & \infty & 0\end{bmatrix}\quad path^{(3)}=\begin{bmatrix}1 & 1 & 2 & 1 & 0 & 2\\ 3 & 2 & 2 & 3 & 0 & 2\\ 3 & 1 & 3 & 3 & 0 & 2\\ 3 & 1 & 4 & 4 & 0 & 2\\ 3 & 1 & 5 & 3 & 5 & 2\\ 3 & 6 & 2 & 6 & 0 & 6\end{bmatrix}$$

$$A^{(4)}=\begin{bmatrix}0&6&13&8&\infty&16\\16&0&7&22&\infty&10\\9&15&0&15&\infty&25\\21&27&12&0&\infty&37\\13&19&4&19&0&29\\21&5&12&25&\infty&0\end{bmatrix}\quad \text{path}^{(4)}=\begin{bmatrix}1&1&2&1&0&2\\3&2&2&3&0&2\\3&1&3&3&0&2\\3&1&4&4&0&2\\3&1&5&3&5&2\\3&6&2&6&0&6\end{bmatrix}$$

由于 $A^{(4)}=A^{(5)}=A^{(6)}$ 和 $\text{path}^{(4)}=\text{path}^{(5)}=\text{path}^{(6)}$，所以表中省略了 $A^{(5)}$，$A^{(6)}$ 和 $\text{path}^{(5)}$，$\text{path}^{(6)}$，打印输出的结果为

```
0          0←0
6          1←0
13         2←1←0
8          3←0
10000      4←0
16         5←1←0
…          …
25         3←5
10000      4←5
0          5←5
```

3. AOV 网与拓扑排序

现实世界中，很多问题都由一系列的有序活动而构成。例如，一个工程项目的开展、一种产品的生产过程或大学期间所学专业的系列课程学习。这些活动可以是一个工程项目中的子工程、一种产品生产过程中的零部件生产或专业课程学习中的某一门课程。所有这些按一定顺序展开的活动，可以使用有向图表示。其中，顶点表示活动，顶点之间的有向边表示活动之间的先后关系，这种有向图称为顶点表示活动网络(Activity On Vertex network，简称 AOV 网)。AOV 网中的顶点可以带有权值，该权值可以表示一项活动完成所需要的时间或所需要投入的费用。AOV 网中的有向边表示了活动之间的制约关系。

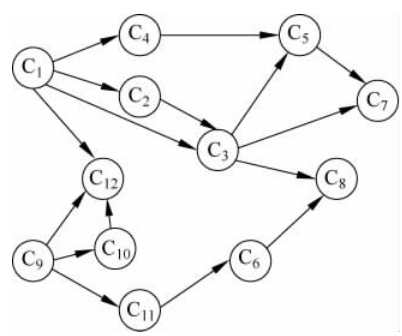


图 3-47　表示课程先后关系的 AOV 网

例如，大学本科专业的学生必须学完一系列的课程才能毕业，其中一部分课程是基础课，无须先修其他课程便可学习；另一部分课程则要求必须学完相关的基础先修课程后，才能进行学习。上述课程和课程之间关系的一个抽象表示示例如表 3-2 所示。该示例也可以用图 3-47 的 AOV 网表示，这里有向边＜C_i，C_j＞表示了课程 C_i 是课程 C_j 的先修课程。

表 3-2　专业课程设置及其关系

课程代号	课程名称	先修课程	课程代号	课程名称	先修课程
C1	课程 1	无	C7	课程 7	C3，C5
C2	课程 2	C1	C8	课程 8	C3，C6
C3	课程 3	C1，C2	C9	课程 9	无
C4	课程 4	C1	C10	课程 10	C9

续表

课程代号	课程名称	先修课程	课程代号	课程名称	先修课程
C5	课程 5	C3,C4	C11	课程 11	C9
C6	课程 6	C11	C12	课程 12	C1,C9,C10

当限制各个活动只能串行进行时，如果可以将 AOV 网中的所有顶点排列成一个线性序列 v_{i1}，v_{i2}，…，v_{in}；并且这个序列同时满足如果在 AOV 网中从顶点 v_i到顶点 v_j存在一条路径，则在线性序列中 v_i必在 v_j之前，就称这个线性序列为拓扑序列。把对 AOV 网构造拓扑序列的操作称为拓扑排序。

AOV 网的拓扑排序序列给出了各个活动按顺序完成的一种可行方案，但并非任何 AOV 网的顶点都可排成拓扑序列。当 AOV 网中存在有向环时，就无法得到该网的拓扑序列。对于实际问题，AOV 网中存在的有向环就意味着某些活动是以自己为先决条件，这显然不合理。例如，对于程序的数据流图，AOV 网中存在环就意味着程序存在一个死循环。

任何一个无环的 AOV 网中的所有顶点都可排列在一个拓扑序列里，拓扑排序的基本操作如表 3-2 所示。

(1)从网中选择一个入度为 0 的顶点并且将其输出。

(2)从网中删除此顶点及所有由它发出的边。

(3)重复上述两步，直到网中再没有入度为 0 的顶点为止。

以上操作会产生两种结果，一种结果是网中的全部顶点都被输出，整个拓扑排序完成；另一种结果是网中顶点未被全部输出，剩余顶点的入度均不为 0，此时说明网中存在有向环。

用以上操作对图 3-47 的 AOV 网拓扑排序的过程如图 3-48 所示。这里得到了一种拓扑序列 C_1，C_2，C_3，C_4，C_5，C_7，C_9，C_{10}，C_{12}，C_{11}，C_6，C_8。

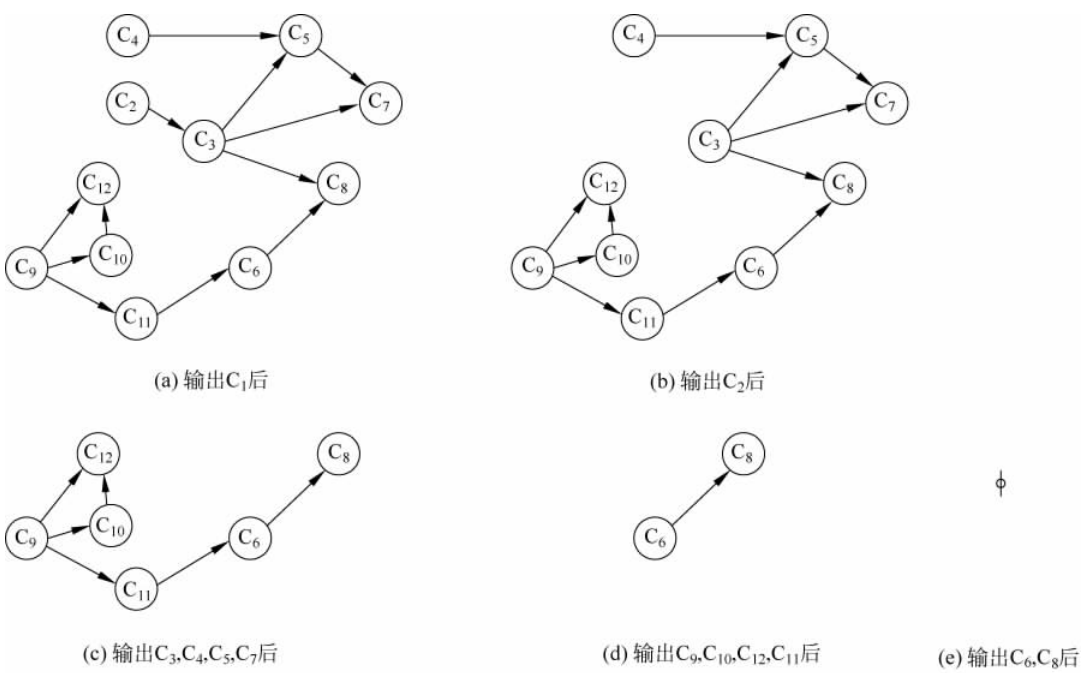


图 3-48　AOV 网拓扑排序过程

从构造拓扑序列的过程中可以看出，许多情况下，入度为 0 的顶点可能有多个，这样就可以给出多种拓扑序列。若按所给出的拓扑序列顺序进行课程学习，可保证在学习任一门课程时，这门课程的先修课程已经学过。

拓扑排序可在有向图的不同存储结构表示方法上实现。下面针对图 3-49(a)所给出的 AOV 网进行讨论。

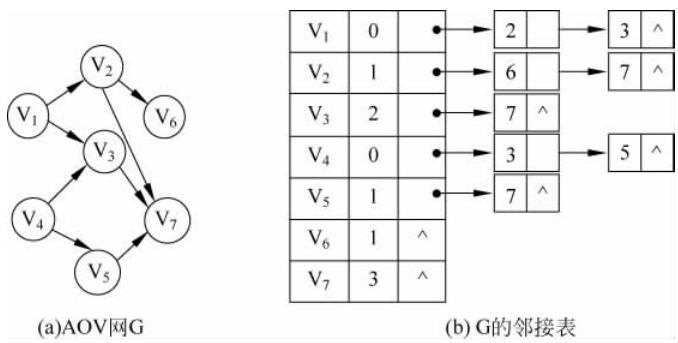


图 3-49　AOV 网 G 及其邻接表

邻接矩阵存储结构中，由于某个顶点的入度由这个顶点相对应列上的 1 的个数所确定，而它的出度由顶点所对应行上的 1 的个数所确定，所以在这种存储结构上实现拓扑排序算法的步骤是

(1) 取 1 作为第一个序号。

(2) 找一个还没有获得序号的全零元素的矩阵列，若没有则停止寻找。此时，如果矩阵中所有列都已获得了序号，则拓扑排序完成；否则，说明该有向图中有环存在。

(3) 把序号值赋给找到的列，并将该列对应的顶点输出。

(4) 将找到列所对应的行中所有为 1 的元素清零。

(5) 序号值增 1，重复执行步骤(2)～(5)。

根据以上步骤，使用一个长度为 n 的数组来存放序号值时，可以给出如下的实现算法。

```
TOPOSORTA(graph *g,int n)        /*对有 n 个顶点的有向图,使用邻接矩阵求拓扑排序*/
{   int i, j, k, t, v, D[n]=0;
    v=1;                         /* 序号变量置 1 */
    for (k=0; k<n; k++) {
        for (j=0; j<n; j++)      /*寻找全零列*/
            if (D[j]==0) {
                t=1;
                for (i=0; i<n; i++)
                    if (g->arcs[i][j]==1) {
                        t=0;
                        break;
                    }            /* 若第 j 列上有 1,则跳出循环 */
                if (t==1) {
                    m=j;
                    break;
                }                /*找到第 j 列为全 0 列*/
            }
```

```
            if ( j!= n ) {
                D[m] = v;                          /* 将新序号赋给找到的列 */
                printf (" %d\t ", g->vexs[m]); /* 将排序结果输出 */
                for (i = 0; i<n; i++)
                    g->arcs[m][i] = 0;             /*将找到的列的相应行置全 0*/
                    v++;                           /*新序号增 1*/
            }
            else break;
        }
        if( v-1<n ) printf (" \n The network has a cycle \n ");
    } /* TOPOSORTA */
```

图 3-49 中 G 的邻接矩阵应用以上算法得到的拓扑排序序列为 v_1，v_2，v_4，v_3，v_5，v_6，v_7。

利用邻接矩阵进行拓扑排序时，程序虽然简单，但效率不高，算法的时间复杂度约为 $O(n^3)$。而利用邻接表使寻找顶点入度为 0 的操作简化，从而提高拓扑排序算法的效率。

邻接表存储结构中，为了便于检查每个顶点的入度，可在顶点表中增加一个入度域(id)。此时，只需要对由 n 个元素构成的顶点表进行检查就能找出入度为 0 的顶点。为避免对每个入度为 0 的顶点重复访问，可用一个链栈来存储所有入度为 0 的顶点。进行拓扑排序前，只要对顶点表进行一次扫描，便可将所有入度为 0 的顶点都入栈。以后，每次从栈顶取出入度为 0 的顶点，并将其输出即可。

一旦排序过程中出现了新的入度为 0 的顶点，同样又将其入栈。入度为 0 的顶点出栈后，根据顶点的序号找到相应的顶点和以该顶点为起点的出边，再根据出边上的邻接点域的值使相应顶点的入度值减 1，便完成了删除所找到的入度为 0 的顶点的出边的功能。

邻接表存储结构中实现拓扑排序算法的步骤为

(1) 扫描顶点表，将入度为 0 的顶点入栈。

(2) 当栈非空时执行以下操作：

① 将栈顶顶点 v_i 的序号弹出，并输出之；

② 检查 v_i 的出边表，将每条出边表邻接点域所对应的顶点的入度域值减 1，若该顶点入度为 0，则将其入栈。

(3) 若输出的顶点数小于 n，则输出有回路，否则拓扑排序正常结束。

具体实现时，链栈无须占用额外空间，只需利用顶点表中入度域值为 0 的入度域来存放链栈的指针(即指向下一个存放链栈指针的单元的下标)，并用一个栈顶指针 top 指向该链栈的顶部即可。由此给出以下的具体算法。

```
typedef int datetype;
typedef int vextype;
typedef struct node {
    int adjvex;                          /* 邻接点域 */
    struct node *next;                   /* 链域 */
} edgenode;                              /* 边表结点 */
    typedef struct {
    vextype vertex;                      /*顶点信息 */
    int id;                              /*入度域 */
```

```
    edgenode * link;                          /* 边表头指针 */
} vexnode;                                    /* 顶点表结点 */
vexnode ga[n];
TOPOSORTB(vexnode ga[ ]) {                    /* AOV 网的邻接表 */
{   int i, j, k, m = 0, top = -1;             /* m 为输出顶点个数计数器,top 为栈指针 */
    edgenode * p;
    for (i = 0; i < n; i++)                   /* 初始化,建立入度为 0 的顶点链栈 */
        if (ga[i].id = = 0) {
            ga[i].id = top;
            top = i;
        }
    while( top!= -1 ) {                       /* 栈非空执行排序操作 */
        j = top;
        top = ga[top].id;                     /* 第 j + 1 个顶点退栈 */
        printf (" %d\t ", ga[j].vertex);      /* 输出退栈顶点 */
        m++;                                  /* 输出顶点计数 */
        p = ga[j].link;
        while(p) {                            /* 删去所有以 v_{j+1} 为起点的出边 */
            k = p->adjvex - 1;
            ga[k].id--;                       /* v_{k+1} 入度减 1 */
            if (ga[k].id = = 0) {             /* 将入度为 0 的顶点入栈 */
                ga[k].id = top;
                top = k;
            }
            p = p->next;                      /* 找 v_{j+1} 的下一条边 */
        }
    }
    if (m < n)                                /* 输出顶点数小于 n,有回路存在 */
        printf (" \n The network has a cycle\n ");
}                                             /* TOPOSORTB */
```

对于图 3-49 中的邻接表执行以上算法时，入度域的变化情况如图 3-50 所示。这时得到的拓扑序列为 v_4，v_5，v_1，v_3，v_2，v_7，v_6。

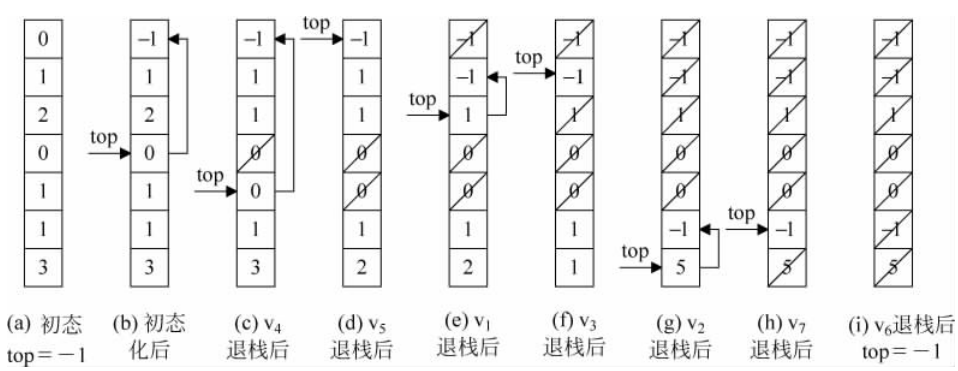


图 3-50　排序过程中入度域变化示例

对一个具有 n 个顶点，e 条边的 AOV 网来说，初始化部分执行时间是 O(n)；排序中，若 AOV 网无回路，则每个顶点入栈和出栈各一次，每个边表结点检查一次，执行时间为

O(n+e)，故总的算法时间复杂度为 O(n+e)。

3.7　小结

本章介绍了两种非线性的数据结构——树和图。

树在存储结构中占据非常重要的地位，在树形结构中树是一种具有层次特征的数据结构，二叉树是一种非常重要、简单、典型的数据结构。二叉树的 5 个性质揭示了二叉树的主要特征。二叉树的存储结构有顺序存储结构和链式存储结构两种。采用顺序存储结构可能会浪费大量的空间，因此常常利用顺序存储结构存储满二叉树和完全二叉树，而一般二叉树大多采用链式存储结构。二叉树的遍历是对二叉树进行各种操作的基础，无论递归算法还是非递归算法都要很好掌握。二叉树的遍历是一种常用的操作。二叉树的遍历分为先序遍历、中序遍历和后序遍历。二叉树的遍历过程就是将二叉树这种非线性结构转换成线性结构。

树和森林的存储有多种方法，和二叉树一样，对树和森林的遍历是对树结构操作的基础，通常有先根和后根两种遍历方法，分别对应于二叉树的先序和中序遍历，所以能利用二叉树的遍历来实现。树、森林和二叉树可以相互转化，树实现起来不是很方便，实际应用中，可以将问题转化为二叉树的相关问题加以实现。

哈夫曼树是 n 个带权叶子结点构成的带权路径长度最短的二叉树。哈夫曼树是二叉树的应用之一，要掌握哈夫曼树的建立方法及哈夫曼编码生成算法，值得注意的是哈夫曼树通常采用静态链式存储结构。

图的存储结构有 4 种，分别是邻接矩阵存储结构、邻接表存储结构、十字链表存储结构和邻接多重表存储结构。其中，最常用的是邻接矩阵存储和邻接表存储。图的遍历分为两种，分别是广度优先遍历和深度优先遍历。图的广度优先遍历类似于树的层次遍历，图的深度优先遍历类似于树的先根遍历。

构造最小生成树的算法主要有两个，分别是普里姆算法和克鲁斯卡尔算法。最短路径是一个与实际关系密切的，最短路径表示完成工程的最短工期，通常用图的顶点表示事件，弧表示活动，权值表示活动的持续时间。

树和图是数据结构中的难点，学好树和图的第一步就是要搞清楚树和图中的一些概念，然后多看算法，耐心研究算法，从多方面认真学习树和图。

3.8　习题

1. 单项选择题

(1) 树形结构的特点是任意一个结点(　　)。

A. 可以有多个直接前趋　　B. 可以有多个直接后继

C. 至少有 1 个前趋　　D. 只有一个后继

(2) 将一棵有 100 个结点的完全二叉树从根这一层开始，每一层从左到右依次对结点进行编号，根结点编号为 1，则编号为 49 的结点的左孩子的编号为(　　)。

A. 98　　B. 99　　C. 50　　D. 48

(3) 对具有 100 个结点的二叉树,若用二叉链表存储,则其指针域部分用来指向结点的左、右孩子,一共有(　　)个指针域为空。

A. 55　　B. 99　　C. 100　　D. 101

(4) 如果 T_1 是由有序树 T 转换来的二叉树,则 T 中结点的后序排列是 T_1 结点的(　　)排列。

A. 先序　　B. 后序　　C. 中序　　D. 层序

(5) 设有 13 个值,用它们组成一棵 Huffman 树,则该 Huffman 树中共有(　　)个结点。

A. 13　　B. 12　　C. 26　　D. 25

(6) 若对一棵有 20 个结点的完全二叉树按层编号,则编号为 5 的结点 x,它的双亲结点及左孩子结点的编号分别为(　　)。

A. 2,11　　B. 2,10　　C. 3,9　　D. 3,10

(7) 将一棵有 100 个结点的完全二叉树从根这一层开始,每一层从左到右依次对结点进行编号,根结点编号为 1,则编号最大的非叶结点的编号为(　　)。

A. 48　　B. 49　　C. 50　　D. 51

(8) 在有 n 个结点的二叉链表中,值为非空的链域的个数为(　　)。

A. n−1　　B. 2n−1　　C. n+1　　D. 2n+1

(9) 由 64 个结点构成的完全二叉树,其深度为(　　)。

A. 8　　B. 7　　C. 6　　D. 5

(10) 一棵含 18 个结点的二叉树的高度至少为(　　)。

A. 3　　B. 4　　C. 5　　D. 6

(11) 在一个无向图中,所有顶点的度之和等于边数的(　　)倍。

A. 1/2　　B. 1　　C. 2　　D. 4

(12) 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

(13) 设有 6 个顶点的无向图,该图至少应有(　　)条边,才能确保它是一个连通图。

A. 5　　B. 6　　C. 7　　D. 8

(14) 具有 5 个顶点的无向完全图共有(　　)条边。

A. 5　　B. 10　　C. 15　　D. 20

(15) 对于一个具有 n 个顶点的无向图,若采用邻接矩阵表示,则该矩阵的大小是(　　)。

A. n　　B. (n−1)×(n−1)　　C. n−1　　D. n×n

(16) 对于一个具有 n 个顶点和 e 条边的无向图,若采用邻接表表示,则表头向量的大小是(　　)。

A. n　　B. n+1　　C. e+1　　D. e

(17) 对于一个具有 n 个顶点和 e 条边的无向图,若采用邻接表表示,则所有邻接表中的结点总数是(　　)。

A. e/2　　B. e　　C. 2e　　D. n+e

(18) 无向图的邻接矩阵是一个(　　)。

A. 对称矩阵　　B. 零矩阵　　C. 上三角矩阵　　D. 对角矩阵

(19) 在含 n 个顶点和 e 条边的无向图的邻接矩阵中,零元素的个数为(　　)。

A. e　　B. 2e　　C. n^2-e　　D. n^2-2e

(20) 假设一个有 n 个顶点和 e 条弧的有向图用邻接表表示,则删除与某个顶点 v_i 相关的所有弧的时间复杂度是(　　)。

A. O(n)　　B. O(e)　　C. O(n+e)　　D. O(n*e)

2. 填空题

(1) 不考虑顺序的 3 个结点可构成_____种不同形态的树,_____种不同形态的二叉树。

(2) 已知某棵完全二叉树的第 4 层有 5 个结点,则该完全二叉树叶子结点的总数为_____。

(3) 已知一棵完全二叉树的第 5 层有 3 个结点,其叶子结点数是_____。

(4) 已知一棵完全二叉树中共有 768 个结点,则该树中共有_____个叶子结点。

(5) 一棵具有 110 个结点的完全二叉树,若 i=54,则结点 i 的双亲编号是_____;结点 i 的左孩子结点的编号是_____,结点 i 的右孩子结点的编号是_____。

(6) 一棵具有 48 个结点的完全二叉树,若 i=20,则结点 i 的双亲编号是_____;结点 i 的左孩子结点编号是_____,右孩子结点编号是_____。

(7) 一棵树 T 采用二叉链表存储,如果树 T 中某结点为叶子结点,则在二叉链表 BT 中所对应的结点一定_____。

(8) 已知在一棵含有 n 个结点的树中,只有度为 k 的分支结点和度为 0 的叶子结点,则该树中含有的叶子结点的数目为_____。

(9) 在有 n 个叶子结点的 Huffman 树中,总的结点数是_____。

(10) 图是一种非线性数据结构,它由两个集合 V(G)和 E(G)组成,V(G)是_____的非空有限集合,E(G)是_____的有限集合。

(11) 在无权图 G 的邻接矩阵 A 中,若(v_i,v_j)或$<v_i,v_j>$属于图 G 的边集合,则对应元素 A[i][j]等于_____。

(12) 设某无向图 G 中有 n 个顶点,用邻接矩阵 A 作为该图的存储结构,则顶点 i 和顶点 j 互为邻接点的条件是_____。

(13) 图 G 有 n 个顶点和 e 条边,以邻接表形式存储,进行深度优先搜索的时间复杂度为_____。

(14) 设无向图 G 中有 n 个顶点 e 条边,则用邻接矩阵作为图的存储结构进行深度优先或广度优先遍历时的时间复杂度为_____。

(15) 具有 n 个顶点的无向图,拥有最少的连通分量个数是_____,拥有最多的连通分量个数是_____。

(16) 图的遍历基本方法中_____是一个递归过程。

(17) n 个顶点的有向图最多有_____条弧。

(18) n 个顶点的无向图最多有_____条边。

(19) 在无向图 G 的邻接矩阵 A 中,若 A[i,j]等于 1,则 A[j,i]等于_____。

(20) 在一个具有 n 个顶点的无向图中,要连通全部顶点至少需要_____条边。

3. 判断题

(1) (　　)非线性数据结构可以顺序存储,也可以链接存储。

(2)(　　)非线性数据结构只能用链接方式才能表示其中数据元素的相互关系。

(3)(　　)完全二叉树一定是满二叉树。

(4)(　　)平衡二叉树中,任意结点左右子树的高度差(绝对值)不超过 1。

(5)(　　)若一棵二叉树的任意一个非叶子结点的度为 2,则该二叉树为满二叉树。

(6)(　　)度为 1 的有序树与度为 1 的二叉树等价。

(7)(　　)一棵树中的叶子结点数一定等于与其对应的二叉树中的叶子结点数。

(8)(　　)二叉树的先序遍历序列中,任意一个结点均排列在其孩子结点的前面。

(9)(　　)若二叉树的叶子结点数为 1,则其先序序列和后序序列一定相反。

(10)(　　)已知一棵二叉树的先序序列和后序序列,就一定能构造出该二叉树。

(11)(　　)邻接表表示法是采用链式存储结构表示图的一种方法。

(12)(　　)用邻接表表示图的方法优于用邻接矩阵表示图的方法。

(13)(　　)在边稀疏的情况下,用邻接表表示图要比用邻接矩阵节省存储空间。

(14)(　　)用邻接矩阵方法存储图比用邻接表方法存储图更容易确定图中任意两个顶点之间是否有边相连。

(15)(　　)在有向图中,逆邻接表表示法是指将原有邻接表所有数据按相反顺序重新排列的一种表示方法。

(16)(　　)在有向图中,采用逆邻接表表示法是为了便于确定顶点的入度。

(17)(　　)无向图的连通分量至少有一个。

(18)(　　)有向图的强连通分量最多有一个。

(19)(　　)简单路径是指图中所有顶点均不相同而形成的一条路径。

(20)(　　)简单回路是指一条起始点和终止点相同的简单路径所构成的回路。

4. 综合题

(1) 如图 3-51 所示的两棵二叉树,分别给出它们的顺序存储结构。

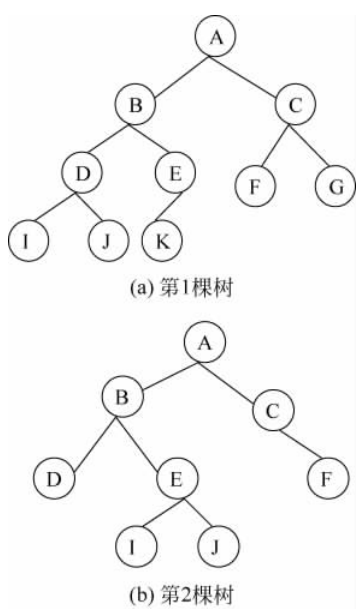


图 3-51　两棵叉树

(2) 已知一棵二叉树的中序、后序序列分别如下：

中序 D C E F B H G A K J L I M

后序 D F E C H G B K L J M I A

要求① 画出该二叉树；

② 写出该二叉树的先序序列。

(3) 将图 3-52 所示的树转换成二叉树，并写出转换后二叉树的先序、中序、后序遍历结果。

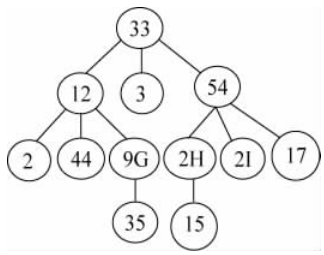


图 3-52　树

(4) 写出图 3-53 中的二叉树先序和后序遍历序列。

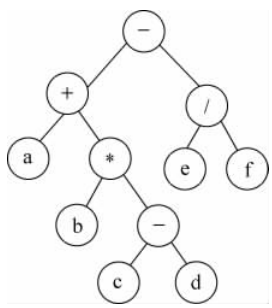


图 3-53　二叉树(1)

(5) 请画出与图 3-54 所示二叉树对应的森林。

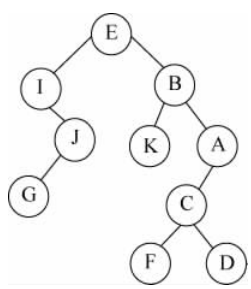


图 3-54　二叉树(2)

(6) 从空树起，依次插入关键字 40、8、90、15、62、95、12、23、56、32，构造一棵二叉排序树。

要求① 画出该二叉排序树；

② 画出删去该树中结点元素值为 90 之后的二叉排序树。

(7) 输入一个正整数序列{100,50,302,450,66,200,30,260},建立一棵二叉排序树,要求① 画出该二叉排序树;

② 画出删除结点 302 后的二叉排序树。

(8) 按给出的一组权值{4,5,7,8,11},建立一个哈夫曼树,并计算出该树的带权路径长度 WPL。

(9) 给出如图 3-55 所示无向图的邻接矩阵和邻接表。

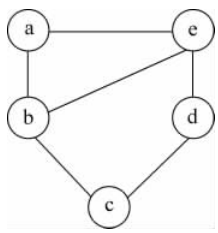


图 3-55 无向图(1)

(10) 求出图 3-56 的一棵最小生成树。

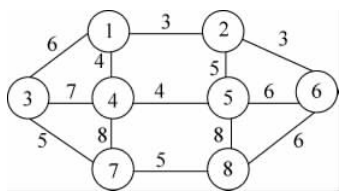


图 3-56 无向图(2)

(11) 如图 3-57 所示的有向图,请给出它的

① 每个顶点的入度和出度;

② 邻接矩阵;

③ 邻接表;

④ 强连通分量。

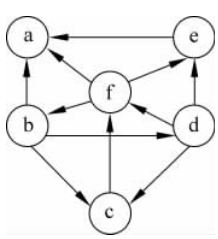


图 3-57 有向图(1)

(12) 给出有向图 3-58 的邻接矩阵、邻接表形式的存储结构,并计算出每个顶点的入度和出度。

(13) 已知图 3-59 所示,其顶点按 a、b、c、d、e、f 顺序存放在邻接表的顶点表中,请画出该图的邻接表,使得按此邻接表进行深度优先遍历时得到的顶点序列为 acbefd,进行广度优先遍历时得到的顶点序列为 acbdfe。

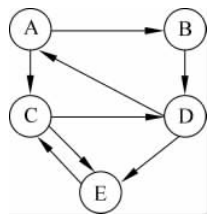


图 3-58　有向图(2)

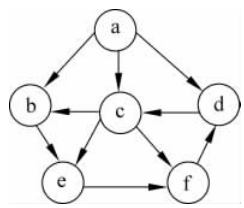


图 3-59　有向图(3)

(14) 已知一组数据序列 D={d_1, d_2, …, d_9},其数据间的关系为 R={(d_1, d_3), (d_1, d_8), (d_2, d_3), (d_2, d_4), (d_2, d_5), (d_3, d_9), (d_5, d_6), (d_8, d_9), (d_9, d_7), (d_4, d_7), (d_4, d_6)}。请画出此数据序列的逻辑结构图,并说明该图属于哪种结构。

(15) 已知数据结构的形式定义为 DS={D,S},其中

```
D={1,2,3,4},S=={R},R={<1,2>,<1,3>,<2,3>,<2,4>,<3,4>}
```

试画出此结构的图形表示。

(16) 已知图 G 的邻接表如图 3-60 所示,顶点 v_1为出发点,完成以下要求:

① 写出按深度优先搜索的顶点序列。

② 写出按广度优先搜索的顶点序列。

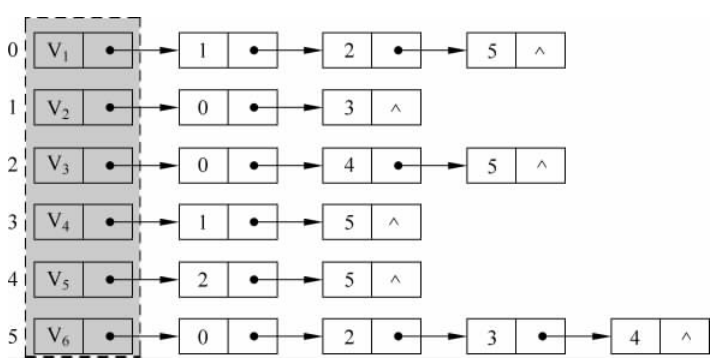


图 3-60　邻接表(1)

(17) 已知某无向图的邻接表存储结构如图 3-60 所示,要求①画出此无向图;②给出无向图的邻接矩阵表示。

(18) 已知一带权连通图 G=(V,E)的邻接表如图 3-61 所示。画出该图,并分别以 BFS 和 DFS 遍历,写出遍历序列,并画出该图的一个最小生成树。示意图 3-62 为表结点的结构

图(以 V_1 为初始点)。

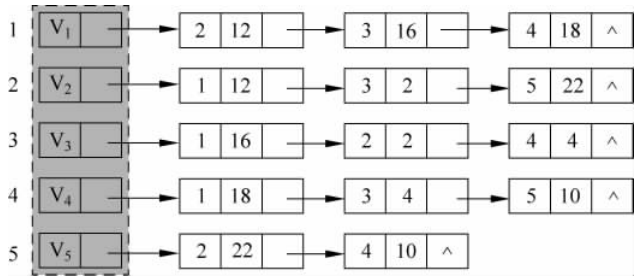


图 3-61　邻接表(2)

顶点号	边上的权值	指针

图 3-62　示意图