第3章 非线性数据结构

在计算机科学中，除了前面章节介绍的线性数据结构之外，还存在许多线性数据结构描述不了的问题，如数据元素之间的层次关系、分支关系等，这些复杂关系需要靠非线性数据结构来进行描述。树和图就是两种最广泛使用的非线性结构，本章将介绍树和图的定义以及基本操作。

3.1 树与二叉树

3.1.1 树的基本概念

1. 树的定义

在数据结构中，树的定义最常用的方式是一种递归定义。**树（Tree）**是包含 $n(n \geqslant 0)$ 个结点的有穷集合。$N=0$ 时称为空树，否则任何一个非空树都满足如下条件：

（1）有且仅有一个特定结点被称为根结点（Root）；

（2）除根结点之外的其余数据元素被分为 $m(m \geqslant 0)$ 个互不相交的集合 $T_1, T_2, \cdots, T_{m-1}$，其中每一个集合本身也是一棵树，被称作子树。

很显然，在上面树的定义当中又用到了树的概念，因此这是一个递归定义，它显示了树的固有特性，这在本章后续的算法当中会有充分的体现。

图 3-1 是包含了 9 个结点的树，即 $T\{A, B, C, \cdots, H, I\}$，其中 A 为树 T 的根结点，除根结点 A 之外的其余结点分为两个不相交的集合：$T_1\{B, D, E, F, H, I\}$ 和 $T_2\{C, G\}$，T_1 和 T_2 本身也是一棵树，分别称为根结点 A 的两棵子树，而两棵子树 T_1 和 T_2 的根结点分别为 B 和 C。其余结点又分为三个不相交的集合：$T_{11}\{D\}$，$T_{12}\{E, H, I\}$ 和 $T_{13}\{F\}$。T_{11}，T_{12} 和 T_{13} 又构成了 T_1 子树的三棵子树。如此继续向下分为更小的子树，直到每棵子树只有一个根结点为止。

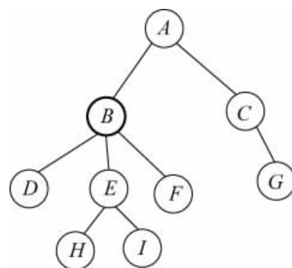


图 3-1 树形结构示意图

下面结合图 3-1 给出树的一些基本术语，通过这些术语能够更好的理解树形结构。

2. 树的基本术语

（1）**结点的度**：一个结点具有的子树的个数称为该结点的度。图 3-1 中，该树根结点 A 的度是 2，结点 B 的度是 3，而结点 D 没有子树，因此结点 D 的度为 0。

（2）**叶子**：度为 0 的结点称为叶子或终端结点。图 3-1 中树的叶子结点是 D、H、I、F、G。

（3）**树的度**：树中所有结点度的最大值称为树的度。图3-1中树的度为3。

（4）**分支结点**：与叶子相对应，树中度大于0的结点称为分支结点，分支结点有时也称非终端结点。一棵树的结点除叶子结点外，其余的都是分支结点。

（5）**孩子结点**：树中某结点子树的根称为该结点的孩子结点。

（6）**双亲结点**：与孩子相对应，如果一个结点存在孩子结点，则这个结点就称为孩子结点的双亲。双亲结点有时也称父亲结点，图3-1中A结点的孩子结点是B和C，同时结点A又是B和C的双亲结点。

（7）**兄弟结点**：具有同一双亲的结点互称为兄弟结点。图3-1中结点D、E、F是兄弟结点。

（8）**堂兄弟结点**：双亲在同一层的结点互称为堂兄弟结点。图3-1中结点D和G是堂兄弟结点。

（9）**子孙结点**：一个结点的所有子树中的结点称为该结点的子孙结点。

（10）**结点层数**：规定树的根结点层数为1，其孩子结点为第2层。以此类推，结点层数等于其双亲结点加1，由此可得到每个结点的层数。

（11）**树的深度**：树中所有结点的最大层数称为树的深度。如图3-1所示，该树的深度为4。

（12）**有序树**：如果一棵树中结点的各子树从左到右是有次序的，即若交换了某结点各子树的相对位置，则构成不同的树，则称这棵树为有序树。

（13）**森林**：有限棵不相交的树的集合称为森林。

通常情况下，在计算机中表示一棵树时，本身就隐含了一种确定的相对次序，因此后面本书中所讨论的树默认都是有序树。

3.1.2 二叉树及其性质

在介绍树形结构的算法之前，先来学习一种特殊形态的树——二叉树。二叉树是树形结构的一个重要类型，许多实际问题可以抽象出来的数据结构往往满足二叉树的形式，且二叉树和普通树之间也存在相互转换的关系。

1. 二叉树的定义

二叉树是特殊形态的树，因此其定义可以参照普通树的定义，顾名思义与普通树的最主要区别在于二叉树中每个结点的儿子至多有两个，即每个结点最多有两个向下的分叉，并由此而得名，其定义如下。

二叉树（**Binary Tree**）是个有限元素的集合，该集合或者为空，或由一个根结点和两棵互不相交的、分别被称为左子树和右子树的二叉树组成。当集合为空时，称该二叉树为空二叉树。

很显然这也是一个递归定义，由定义可得到二叉树的两个基本特点，具体如下。

（1）二叉树中不存在度大于2的结点，即每个结点至多有两个孩子。

（2）二叉树是有序的，即若将其左、右子树颠倒，就成为另一棵不同的二叉树。即使树中结点只有一棵子树，也要区分它是左子树还是右子树。因此二叉树具有五种基本形态，如图3-2所示。

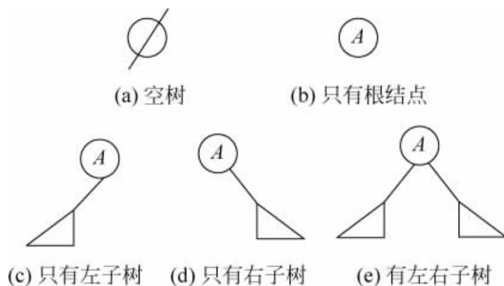


图 3-2　二叉树的 5 种基本形态

思考：根据树和二叉树的定义，读者可以自己试着画出具有 3 个结点的树和二叉树的形态。

2. 二叉树的性质

【性质 3.1】 一棵非空二叉树的第 i 层结点数最多为 $2^{i-1}(i\geqslant 1)$。

证明：该性质可由数学归纳法，根据二叉树的定义来证明。

【性质 3.2】 一棵深度为 k 的二叉树中，最多具有 $2^k-1(k\geqslant 1)$个结点。

证明：在深度为 k 的二叉树中，只有当每层的结点数都为最大值时，树的总结点数才为最大值，因此根据性质 3.1 可以得到深度为 k 的二叉树中结点数最多为：

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

【性质 3.3】 在一棵非空二叉树中，如果其叶子结点数为 n_0，度数为 2 的结点为 n_2，则有：

$$n_0=n_2+1$$

证明：

(1) 设 M 为二叉树的结点总数，n_1 为二叉树中度为 1 的结点数，则有：$M=n_0+n_1+n_2$；

(2) 此外，根据二叉树的定义，度为 1 的结点有 1 个孩子，度为 2 的结点有 2 个孩子，因此二叉树中的孩子结点数为 n_1+2n_2；

(3) 而在二叉树中，只有根结点不是任何结点的孩子，所以二叉树的总结点树又可表示为 $M=n_1+2n_2+1$。

由此得到 $M=n_0+n_1+n_2=n_1+2n_2+1$，最终推导出 $n_0=n_2+1$。

【性质 3.4】 具有 n 个结点的完全二叉树的深度为$\lfloor \text{lb}n \rfloor+1$。

在证明性质 3.4 之前先来介绍两种特殊形态的二叉树——完全二叉树和满二叉树。

一棵深度为 k 且有 2^k-1 个结点的二叉树称为**满二叉树**，即该满二叉树中的结点数为最大值。图 3-3(a)给出了一棵深度为 4 的满二叉树，这种二叉树的特点是所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上。

可以对满二叉树的结点自上而下、自左而右进行连续编号，约定编号从根结点起，编号结果如图 3-3(b)所示。由此可引出完全二叉树的定义如下：深度为 k 的，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时，称为**完全二叉树**。图 3-4 给出了一个深度为 4 的完全二叉树。

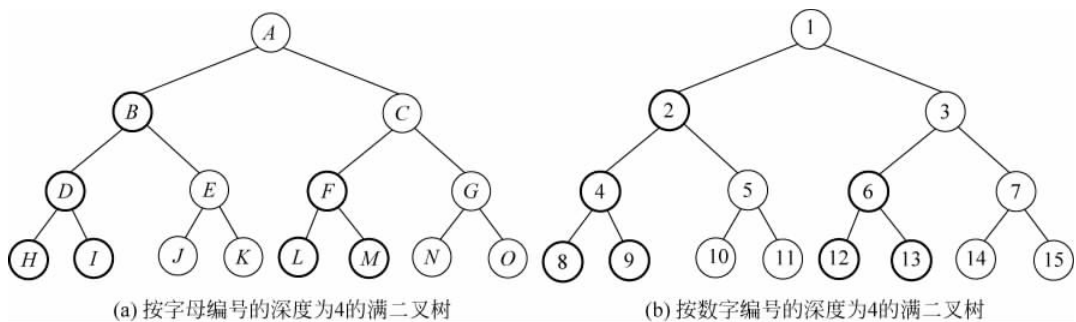


图 3-3　深度为 4 的满二叉树

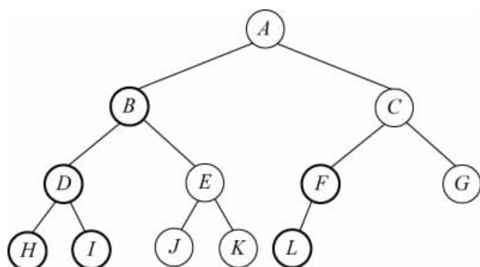


图 3-4　深度为 4 的完全二叉树

由此可以总结出完全二叉树和满二叉树的特点如下。

(1) 完全二叉树的叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。

(2) 一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。

下面证明性质 3.4：设一棵完全二叉树的深度为 k，结点个数为 n，则根据完全二叉树的定义和性质 3.2 可知，结点的个数 n 要位于深度为 $k-1$ 层和 k 层的最大结点数之间，即

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

对不等式两边同取对数可得

$$k-1 \leqslant \mathrm{lb}n < k$$

又因为 $k-1$ 和 k 是相邻的两个整数，所以

$$k-1 = \lfloor \mathrm{lb}n \rfloor$$

由此性质 3.4 得证，即 $k=\lfloor \mathrm{lb}n \rfloor+1$。

【性质 3.5】 对于具有 n 个结点的完全二叉树，如果按照从上至下，同一层按照从左到右的顺序对二叉树的所有结点从 1 开始顺序编号，则对于任意的序号为 i 的结点，有如下性质。

(1) 如果 $i=1$，则结点 i 是二叉树的根，该结点无双亲；如果 $i>1$，则序号为 i 的结点的双亲结点序号为 $\lfloor i/2 \rfloor$。

(2) 如果 $2i>n$，则结点 i 无左孩子，否则其左孩子结点的序号为 $2i$。

(3) 如果 $2i+1>n$，则结点 i 无右孩子，否则其右孩子结点序号为 $2i+1$。

此性质可采用数学归纳法证明。

3.1.3 二叉树的存储结构

二叉树的存储方式主要有顺序存储和链式存储两种方式,下面进行详细介绍。

1. 二叉树的顺序存储

所谓二叉树的顺序存储,就是用一组连续的存储单元,按照从上至下、从左到右的顺序依次存放二叉树中的结点。因此,依据二叉树的性质,完全二叉树和满二叉树采用顺序存储比较合适,树中结点的序号可以唯一地反映出结点之间的逻辑关系。图 3-4 中的完全二叉树的顺序存储结构如图 3-5 所示。

1	2	3	4	5	6	7	8	9	10	11	12
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*

图 3-5　完全二叉树的顺序存储结构

从图 3-5 中可以出,完全二叉树的顺序存储结构能够很好地反映出二叉树结构的线性序列,这样既能够最大地节省存储空间,又可以利用数组元素的下标值确定结点在二叉树中的位置,以及结点之间的关系。

但是对于一般的二叉树而言,如果仍按从上至下和从左到右的顺序将树中的结点顺序存储在一维数组中,则数组元素下标之间的关系不能够反映二叉树中结点之间的逻辑关系。这时,需要对二叉树进行改造,增添一些空结点,使之成为一棵形式上的完全二叉树,然后再用一维数组顺序存储,空结点在存储时用 0 表示,如图 3-6 所示。

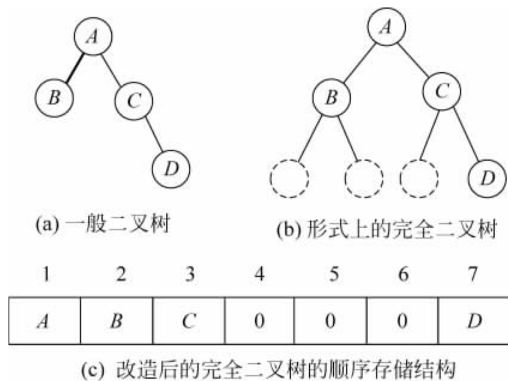


图 3-6　一般二叉树的顺序存储结构

显然,这种存储方式会造成存储空间的大量浪费,尤其是图 3-6(a)中的右单支树,需要补足的空结点非常多,因此对于一般的二叉树而言,采用顺序的存储方式不太合适。下面介绍二叉树的链式存储结构。

2. 二叉树的链式存储

所谓二叉树的链式存储是指用链表来表示一棵二叉树,根据二叉树的定义,二叉树中每个结点最多拥有左右两个孩子,因此在构建链表节点时,每个结点由三个域组成:数据域、左孩子指针域和右孩子指针域,这种形式在数据结构中又称为**二叉链表**。结点结构如图 3-7 所示。

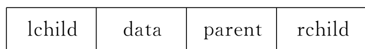
lchild	data	rchild

图 3-7　结点结构 1

其中，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。二叉树的二叉链表的 C 语言描述如下。

```
typedef struct BiTNode{
      elemtype data;
      struct BiTNode * lchild; * rchild;
      }BiTNode, * BiTree;
```

在上述的二叉链表中，可以比较方便地从某一个结点出发找到它的子结点，但是没法找到其父结点，也就是说二叉链表是单方向的。有时为了便于对链表中的结点进行查找，在结点结构中增加一个指向其父结点的指针域，这种结构又称**三叉链表**。结点结构如图 3-8 所示。



lchild	data	parent	rchild

图 3-8　结点结构 2

图 3-9 给出了图 3-6(a)所示的二叉树的二叉链表和三叉链表存储结构，图 3-9 中 T 表示该二叉树根结点的指针。

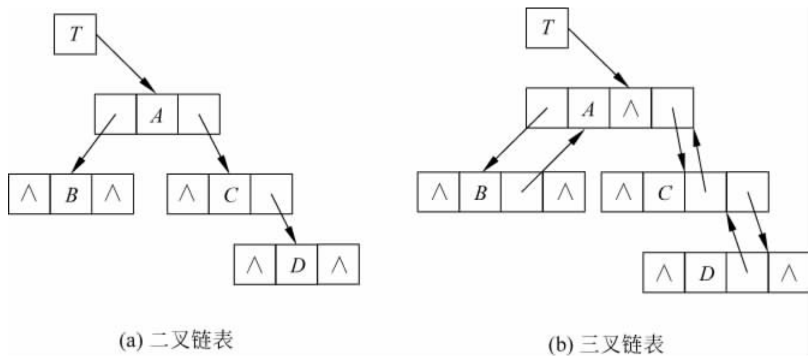


图 3-9　图 3-6 中二叉树的链式存储结构

3.1.4　二叉树的遍历方法

二叉树的遍历是指按照一定次序访问二叉树中的所有结点，且每个结点仅被访问一次。遍历是二叉树中经常要用到的一种操作，通过一次完整的遍历，可使二叉树中结点由非线性序列变为某种意义上的线性排列。

由二叉树的定义可知，一棵二叉树由根结点、左子树和右子树三部分组成，因此，只要依次遍历这三部分，就可以遍历整个二叉树，常用的遍历方式主要有先序遍历、中序遍历和后序遍历。如果用 D 操作表示访问根结点，用 L 操作表示遍历根结点的左子树，用 R 操作表示遍历根结点的右子树，则三种常用的遍历方式可以表示为：先序遍历(DLR)、中序遍历(LDR)和后序遍历(LRD)。从中我们可以看出，这三种遍历方式的共同点在于都是先访问左子树，再访问右子树，即 L 操作总是在 R 操作之前，而不同的地方在于 D 操作位于不同的位置。下面给出三种遍历方式的递归算法思路，该思路是基于二叉树的递归定义给出的。

1. 先序遍历(DLR)

若二叉树为空，遍历结束，否则按如下步骤进行操作：

① 访问根结点的数据域；

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 C 语言递归算法如下。

算法 3-1：先序遍历二叉树的递归算法

```
void Preorder(BiTNode *bt)
{
    if(bt != NULL)
    {
        printf("%d", bt->data);              /* 访问根结点的数据域 */
        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(LDR)

若二叉树为空，遍历结束，否则按如下步骤进行操作：

① 中序遍历根结点的左子树；

② 访问根结点的数据域；

③ 中序遍历根结点的右子树。

中序遍历二叉树的 C 语言递归算法如下。

算法 3-2：中序遍历二叉树的递归算法

```
void Inorder (BiTNode *bt)
{
    if(bt != NULL)
    {
        Inorder(bt->lchild);                 /* 中序递归遍历 bt 的左子树 */
        printf("%d", bt->data);              /* 访问根结点的数据域 */
        Inorder(bt->rchild);                 /* 中序递归遍历 bt 的右子树 */
    }
}
```

3. 后序遍历(LRD)

若二叉树为空，遍历结束，否则按如下步骤进行操作：

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 访问根结点的数据域。

后序遍历二叉树的 C 语言递归算法如下。

算法 3-3：后序遍历二叉树的递归算法

```
void Posorder(BiTNode *bt)
{
    if(bt != NULL)
    {
        Posorder(bt->lchild);                /* 后序递归遍历 bt 的左子树 */
        Posorder(bt->rchild);                /* 后序递归遍历 bt 的右子树 */
```

```
        printf("%d", bt->data);                /* 访问根结点的数据域 */
    }
}
```

4. 二叉树遍历的非递归算法

上面介绍的二叉树先序遍历、中序遍历和后序遍历三种算法都是递归算法。这种递归算法程序简洁,易于实现,但是并非所有程序设计语言都允许递归,另外递归算法可读性一般不好,执行效率也不高,因此下面将分析二叉树遍历的非递归算法。

分析上述三种遍历算法可以发现,如果将访问根结点的 D 操作,即把三个算法中的 printf 语句去掉,那么三个算法的结构是完全一致的。也就是说对二叉树进行先序、中序和后序遍历都是从根结点开始的,且在遍历过程中经过结点的路线是一样的,只是访问的时机不同而已。先序遍历是在深入时遇到结点就访问,中序遍历是在从左子树返回时遇到结点访问,后序遍历是在从右子树返回时遇到结点访问。遍历过程中,返回结点的顺序与深入结点的顺序相反,既后深入先返回,正好符合栈结构后进先出的特点。因此,可以用栈来帮助实现这一遍历路线。其过程如下:在沿左子树深入时,深入一个结点入栈一个结点,若为先序遍历,则在入栈之前访问之,当沿左分支深入不下去时,则返回,即从堆栈中弹出前面压入的结点;若为中序遍历,则此时访问该结点,然后从该结点的右子树继续深入;若为后序遍历,则将此结点再次入栈,然后从该结点的右子树继续深入,与前面类同,仍为深入一个结点入栈一个结点,深入不下去再返回,直到第二次从栈里弹出该结点,才访问之。

下面以先序遍历为例给出非递归算法。该算法中,二叉树以二叉链表存放,一维数组 stack[MAXNODE]用于实现栈,变量 top 用来表示当前栈顶的位置,算法的具体步骤如下:

(1) 当树非空时,将指针 p 指向根结点,p 为当前结点指针;

(2) 先访问当前结点 p,并将 p 压入栈 stack 中;

(3) 令 p 指向其左孩子;

(4) 重复执行步骤(2)和(3),直到 p 为空为止;

(5) 从栈 stack 中弹出栈顶元素,将 p 指向此元素的右孩子;

(6) 重复执行步骤(2)～(5),直到 p 为空并且栈也为空;

(7) 遍历结束。

算法 3-4:先序遍历二叉树的非递归算法

```
void NRPreOrder(BiTree bt)                     /* 非递归先序遍历二叉树 */
{
        BiTree stack[MAXNODE], p;
        int top;
        if(bt == NULL) return;
        top = 0;
        p = bt;
        while(!(p == NULL&&top == 0))
        {
            while(p!= NULL)
          {
                printf("%d", p->data);         /* 访问结点的数据域 */
                 if(top < MAXNODE - 1)         /* 将当前指针 p 压栈 */
```

```
                {   stack[top] = p;
                    top++;
            }
                else
                {
                    printf("栈溢出");
                return;
                }
                p = p -> lchild;               /* 指针指向 p 的左孩子 */
            }
            if(top <= 0) return;               /* 栈空时结束 */
            else
            {
                top -- ;
                p = stack[top];                /* 从栈中弹出栈顶元素 */
                p = p -> rchild;               /* 指针指向 p 的右孩子结点 */
            }
        }
}
```

3.1.5 树的存储结构和遍历

对于一棵普通的树而言，既可以采用顺序存储结构，也可以采用链式存储结构，但无论采用何种存储方式，除了准确存储各结点本身的数据信息外，还要唯一反映树中各结点之间的逻辑关系，常见的树的存储表示方法有双亲表示法、孩子表示法、孩子兄弟表示法等。在此仅介绍一种常用的链式存储结构——孩子兄弟表示法，这种方法又称树的二叉链表表示法。

1. 树的二叉链表表示法

在该二叉链表中，每个结点除其信息域外，还包含两个指针域，分别指向该结点的第一个孩子结点和下一个兄弟(右邻兄弟)结点。图 3-10 给出了图 3-1 所示的树的二叉链表存储结构。

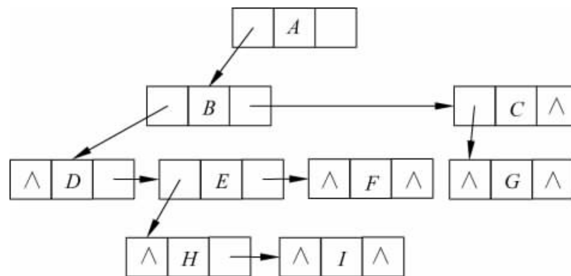


图 3-10　图 3-1 中树的二叉链表存储结构

2. 树的遍历

对普通树的遍历只有先序和后序两种，但没有中序遍历之说，中序遍历只有二叉树才有。

(1) 先根遍历：首先访问根结点，然后按照从左到右的顺序先根遍历根结点的每一棵

子树。

(2) 后根遍历：首先按照从左到右的顺序后根遍历结点的每一棵子树，然后访问根结点。

3.1.6 树、森林与二叉树

由分析树和二叉树的二叉链表表示法可知，树和二叉树的存储表示结构本质上是一致的，两者都是用二叉链表作为其存储结构，只是解释不同而已。因此我们能够以二叉链表作为媒介推导出树与二叉树之间的一个对应关系，也就是说，给定一棵树，可以找到唯一的一棵二叉树与之对应。下面给出树与二叉树之间的这种对应关系。

1. 树转化为二叉树

将树转化成二叉树就是将树首先用兄弟孩子链表表示法进行存储，然后根据该二叉链表存储结构推导出对应的二叉树。具体步骤如下。

(1) 加线：亲兄弟之间加一虚连线。

(2) 抹线：抹掉(除最左一个孩子外)该结点到其余孩子之间的连线。

(3) 旋转：新加上去的虚线改实线且均向右斜(rchild)，原有的连线均向左斜(lchild)。

图 3-11 给出了一棵普通树转化为二叉树的具体步骤。

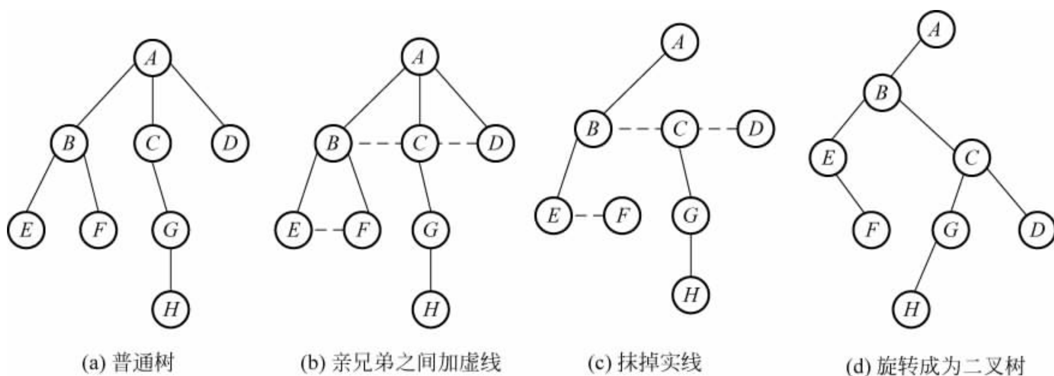


图 3-11 普通树转化为二叉树的具体步骤

由上面的转化过程可以看出，在最终得到的二叉树中，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，因此变换后的二叉树根结点的右孩子必然为空。

根据树与二叉树表的转换关系以及树与二叉树遍历的操作定义可知，树的遍历序列与由树转换成的二叉树的遍历序列之间有如下对应关系：

(1) 树的先序遍历⇔二叉树的先序遍历；

(2) 树的后序遍历⇔二叉树的后序遍历。

2. 森林转换为二叉树

顾名思义，森林指的是树的有限集合。森林中每棵树又可以用二叉树表示，然后将森林中各棵树的根视为兄弟，这样，森林也同样可以用二叉树表示。森林转换为二叉树的方法如下：

(1) 将各棵树分别转换为二叉树。

（2）由于变换后的二叉树根结点的右孩子必然为空，因此按森林中树的次序，依次将后一棵二叉树作为前一棵二叉树根结点的右子树连接在一起形成最终的二叉树。

森林转换成对应的二叉树的具体过程如图 3-12 所示。

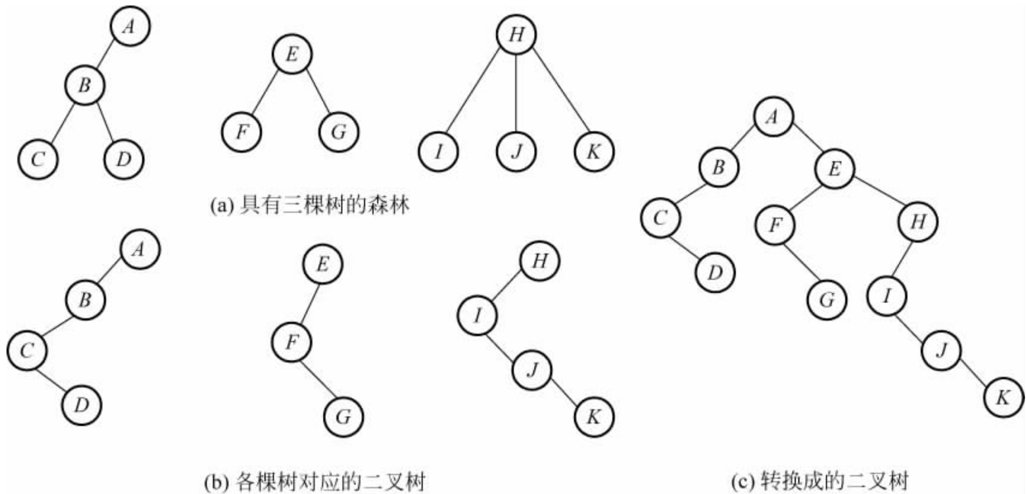


图 3-12　森林转换成对应的二叉树的具体过程

3. 二叉树转换为树和森林

树和森林都可以转换为二叉树，二者不同的是树转换成的二叉树，其根结点无右孩子，而森林转换后的二叉树，其根结点有右孩子。显然这一转换过程是可逆的，即可以依据二叉树的根结点有无右孩子，将一棵二叉树还原为树或森林。具体过程如下：

（1）若某结点 x 是其双亲 y 的左孩子，则把结点 x 的右孩子，右孩子的右孩子……都与该结点的双亲结点 y 用线连起来。

（2）删去原二叉树中所有的双亲结点与右孩子结点的连线。

（3）整理由（1），（2）两步所得到的树或森林，使之结构层次分明。

3.1.7　哈夫曼树及其应用

1. 哈夫曼树的定义及构成方法

最优二叉树，也称哈父曼（Haffman）树，是指对于一组带有确定权值的终端结点，构造具有最小带权路径长度的二叉树。该树在数据编码等算法中有着非常广泛的应用。

从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支数称为这两个结点之间的路径长度。对于一整棵树而言，从根结点到树中每一个结点的路径长度之和称为树**的路径长度**。根据该定义可知在结点数目相同的二叉树中，完全二叉树具有最小的路径长度。

在二叉树中还可以为结点赋以权值，称为带权二叉树。如果二叉树中的叶子结点都具有一定的权值，则可以把树的路径长路加以推广，将从根结点到各个叶子结点的路径长度与相应叶子结点权值的乘积之和叫做二叉树的**带权路径长度**，记为：

$$\mathrm{WPL}=\sum_{k=1}^{n}W_kL_k$$

其中，W_k 为第 k 个叶子结点的权值，L_k 为第 k 个叶子结点的路径长度。

【例 3-1】 图 3-13 中三棵二叉树的叶子结点的权值分别为 2，4，6，8，求其带权路径长度。

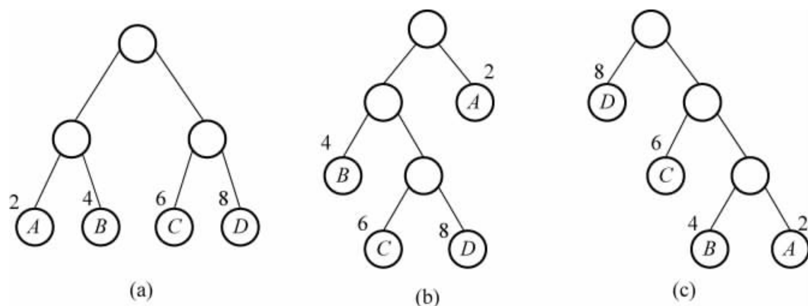


图 3-13　具有不同带权路径长度的二叉树

解：根据带权路径长度的定义，可算得图 3-13 中的三棵树的带权路径长度分别为：

(a) WPL＝2＊2＋4＊2＋6＊2＋8＊2＝40

(b) WPL＝4＊2＋6＊3＋8＊3＋2＊1＝52

(c) WPL＝8＊1＋6＊2＋4＊3＋2＊3＝38

分析上述结果，图 3-13(a)树是完全二叉树，但是带权路径最小的树是图 3-13(c)，因此带权路径长度最小的二叉树不一定是完全二叉树。观察图 3-13(c)中树的特点可以发现，在带权路径长度最小的二叉树中，权值越大的终端结点离二叉树的根越近。而哈夫曼树就是这种带权路径长度最小的二叉树，下面介绍哈夫曼树的构造方法。其基本思想如下。

(1) 设给定一组权值为$\{W_1, W_2, \cdots, W_n\}$的结点，据此构成包含 n 棵二叉树的森林 $F=\{T_1, T_2, \cdots, T_n\}$，$F$ 中的每棵二叉树 T_i 只有一个带权为 W_i 的根结点($i=1,2,\cdots,n$)，其左右子树皆为空。

(2) 在森林 F 中选取两棵根结点的权值最小和次小的二叉树作为左、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) 在森林 F 中删除这两棵权值最小和次小的二叉树，同时将新生成的二叉树加入森林 F 中。

(4) 重复步骤(2)和(3)，直到森林 F 中只有一棵二叉树为止，此树便是哈夫曼树。

例如，给定一组权值{1，3，5，7}，图 3-14 给出了构造相应哈夫曼树的过程。

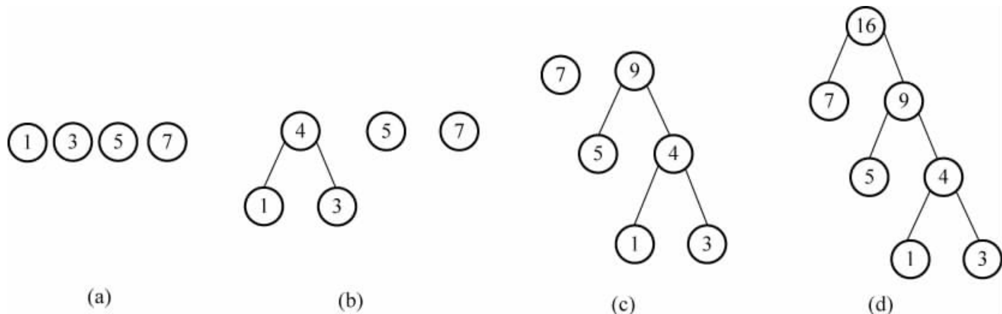


图 3-14　哈夫曼树的构造过程

2. 哈夫曼树的应用

哈夫曼树的应用很广，在不同的应用中叶子结点的权值可以有不同的解释。哈夫曼树应用到信息编码中，权值可看成是某个符号出现的频率；应用到判定过程中，权值可看成是某类数据出现的频率；应用到排序过程中，权值可看成是已排好次序而待合并的序列的长度等。其中最典型的就是在编码技术上的应用，利用哈夫曼树可以得到平均长度最短的编码。本书以数据通信中电文的传送为例来分析说明。

在数据通信中，经常需要将传送的文字转换成由二进制 0 和 1 组成的码字，称为编码。这些二进制编码往往是针对英文字符进行的，而字符集中的每个字符使用的频率是不等的，如果能让使用频率较高的字符的编码尽可能短，这样就可以缩短整个信息通信过程中所需传送的二进制编码序列的长度，从而达到节省通信资源的目的。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 $W=\{w_1, w_2, \cdots, w_n\}$，以 $d_1, d_2, \cdots, d_n$ 作为叶子结点，$w_1, w_2, \cdots, w_n$ 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子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，称为哈夫曼编码。

【例 3-2】 给出一个字符文本：huff full luff lluhf，请给出该文本的哈夫曼编码。

解：分析该文本得到字符集 $D=\{h, u, l, f\}$，对应的权值分别为 $W=\{2, 4, 5, 6\}$，由此可得到哈夫曼树和哈夫曼编码，如图 3-15 所示。

这种编码的优点是：对于给出的文本，其编码长度是最短的；任一字符的编码均不可能是另一字符编码的开始部分（前缀）。这样，两个字符之间就不需要分隔符，但是，两个词之间仍需要留空格，以起到分隔作用。其缺点是：每个字符的编码长度不相等，译码时较困难。

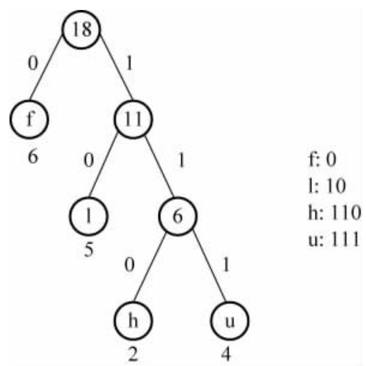


图 3-15　哈夫曼树和对应的哈夫曼编码

哈夫曼树不但可以用来编码，还可以用来解码，其解码过程正好与编码过程相反。其解码过程为：从哈夫曼树的根结点出发，依次识别电文中的二进制编码，如果为 0，则走向左孩子，否则走向其右孩子，走到叶子结点时，就可以得到相应的解码字符。具体的解码过程读者可参阅相关参考书，这里不再做深入介绍。

3.2　图

图形结构是一种比树形结构更复杂的非线性数据结构。在树形结构中，结点中具有分支层次关系，每一层上的结点只能和上一层中的最多一个结点相关，但可能和下一层的多个结点相关。而在图形结构中，结点之间的联系是任意的，每个结点都可以与其他的结点相联系。因此，图形结构被用于描述各种复杂的数据对象，应用极为广泛，特别是近年来发展迅速，已渗入到诸如语言学、逻辑学、物理、化学、电信工程、计算机、数学及其他分支中。

3.2.1 图的逻辑定义

下面介绍图(Graph)的定义及其相关术语。

1. 图

图 G 是由两个集合 $V(G)$ 和 $E(G)$ 组成的,记为 $G=(V, E)$,其中,$V(G)$ 是顶点的非空有限集,$E(G)$ 是边的有限集合,是顶点的无序对或有序对,反映的是顶点之间的关系。根据边 E 的不同,图又可分为无向图和有向图。

2. 无向图和边

如果图中每条边都是顶点的无序对,则称此图为无向图。无向边用圆括号括起的两个相关顶点来表示。图 3-16(a)所示的 G_1 为无向图,在该图中,集合 $V_1=\{v_1, v_2, v_3, v_4\}$,集合 $E_1=\{(v_1, v_2),(v_2, v_3),(v_3, v_4),(v_1, v_4)\}$。对于无向图而言,$(v_1, v_2)$ 与 (v_2, v_1) 表示同一条边。

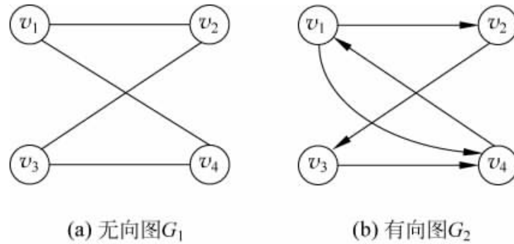


图 3-16　图形结构示例

3. 有向图和弧

如果图中每条边都是顶点的有序对,即每条边都用箭头表明了方向,则此图为有向图。有向图中的边也称弧,用尖括号括起一对顶点表示。图 3-16(b)所示的 G_2 为有向图,在该图中,$V_2=\{v_1, v_2, v_3, v_4\}$,集合 $E_2=\{<v_1,v_2>, <v_1, v_4>, <v_2,v_3>, <v_3,v_4>, <v_4,v_1>\}$。对于有向图而言,$<v_1, v_4>$ 与 $<v_4,v_1>$ 表示不同的弧。

4. 子图

对于图 $G=(V, E)$,$G'=(V', E')$,若存在 V' 是 V 的子集,且 E' 是 E 的子集,则称图 G' 是 G 的一个子图。图 3-17 分别给出了 G_2 的三个子图。

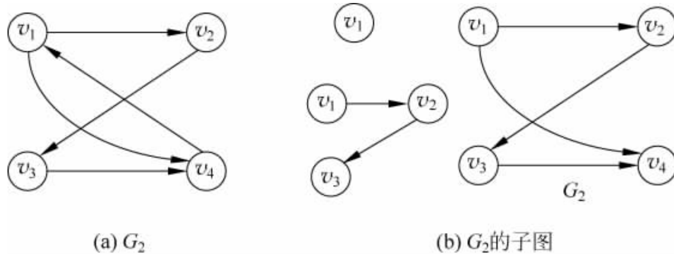


图 3-17　图与子图

5. 带权图

与边或者弧有关的数据信息称为权(Weight)。在实际应用中,可以为权值赋以某种具体含义。例如,在一个反映城市交通线路的图中,边上的权值可以表示该线路的长度或者等

级；对于一个电子电路图，边上的权值可以表示两个端点之间的电阻、电流或电压值等。边上带权的图称为网(Network)，如图 3-18 所示。

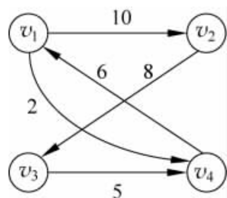


图 3-18　边上带有权值的网

6. 顶点的度

在无向图中，顶点的度就是和该顶点相关联的边的数目，通常记为 TD(v)，图 3-16 中 G_1 图中 TD(v_1)＝2。在有向图中，顶点的度具有入度与出度的区别。顶点 v 的入度是指以顶点为终点的弧的数目。记为 ID(v)；顶点 v 出度是指以顶点 v 为始点的弧的数目，记为 OD(v)。该顶点的度 ID 则是此顶点的入度与出度之和，即 TD(v)＝ID(v)＋OD(v)。如图 3-16 中的 G_2，ID(v_1)＝1，OD(v_1)＝2，TD(v_1)＝ID(v_1)＋OD(v_1)＝3。

7. 路径

在无向图中，从顶点 v_i 到顶点 v_j 的路径(Path)是指一个顶点序列 v_i，v_{p1}，v_{p2}，…，v_{pk}，v_j。其中，(v_i，v_{p1})，(v_{p1}，v_{p2})，…，(v_{pk}，v_j)分别为图中的边，且 $1\leqslant k\leqslant n$。如果 G 是有向图，则路径也是有向的。路径上边的数目称为路径长度。

8. 连通图与强连通图

在无向图中，如果从一个顶点 v_i 到另一个顶点 v_j($i\neq j$)存在路径，则称顶点 v_i 和 v_j 是连通的。如果无向图中任意两个顶点都是连通的，则称该图是连通图。在有向图中，若每一对顶点 v_i 和 v_j 之间都存在 v_i 到 v_j 及 v_j 到 v_i 的路径，则称此图为强连通图。

9. 邻接点

在无向图中，如果边(v_i，v_j)∈E，则 v_i 和 v 互为邻接点，即 v_i 是 v_j 的邻接点，v_j 也是 v_i 的邻接点。在有向图中，如果弧< v_i，v_j >∈E，则 v_j 是 v_i 的邻接点。称 v_i 邻接到 v_j，或顶点 v_j 邻接自顶点 v_i。

3.2.2 图的存储结构

图是一种复杂的数据结构，主要表现在顶点之间的逻辑关系错综复杂。从图的定义可知，一个图的信息包括两部分，即图中顶点的信息以及顶点之间的联系(边或者弧的信息)。因此无论采用什么样的存储结构，都要完整、准确地反映这两方面的信息。所以，图存储的方法很多，需要根据具体图形和所要做的运算选取适当的存储结构。这里只讨论两种最常用的表示方法：**邻接矩阵**和**邻接表**。

1. 邻接矩阵

所谓邻接矩阵储存结构就是用一个一维数组存放图中所有顶点的信息；用一个二维数组来存放数据元素之间关系的信息(即边或弧)，这个二维数组称为邻接矩阵。所以这种存储形式的关键在于二维数组的构建，下面介绍该邻接矩阵的构建方法。

设图 G＝(V，E)，具有 n 个顶点，即 V＝{v_0，v_1，…，v_{n-1}}，表示 G 中各顶点相邻关系的邻接矩阵是一个 $n\times n$ 的矩阵 $\boldsymbol{A}$，表示如下：

$$\boldsymbol{A}[i,j]=\begin{cases}1 & 若(v_i,v_j)\ 或 < v_i,v_j > 是\ E(G)\ 中的边\\ 0 & 若(v_i,v_j)\ 或 < v_i,v_j > 不是\ E(G)\ 中的边\end{cases}$$

图 3-19 给出了图 3-16 中无向图和有向图的邻接矩阵 $\boldsymbol{A}_1$ 和 $\boldsymbol{A}_2$。

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

图 3-19　图 3-16 中 G_1 和 G_2 的邻接矩阵

从图的邻接矩阵存储方法容易看出这种表示具有以下特点。

(1) 无向图的邻接矩阵一定是一个对称矩阵。因此，在具体存放邻接矩阵时只需存放上三角(或下三角)矩阵元素即可。

(2) 对于无向图，邻接矩阵的第 i 行(或第 i 列)非零元素(或非∞元素)的个数正好是第 i 个顶点的度 TD(v_i)。

(3) 对于有向图，邻接矩阵的第 i 行(或第 i 列)非零元素(或非∞元素)的个数正好是第 i 个顶点的出度 OD(v_i)[或入度 ID(v_i)]。

(4) 用邻接矩阵方法存储图，很容易确定图中任意两个顶点之间是否有边相连；但是要确定图中有多少边，则必须按行、按列对每个元素进行检测，所花费时间代价很大；同时，使用两个数组分别存储图的顶点信息和关系信息，空间消耗也比较大，这是用邻接矩阵存储图的局限性。

若 G 是网，则邻接矩阵可定义为：

$$A[i,j]=\begin{cases}W_{ij} & \text{若}(v_i,v_j)\text{ 或}<v_i,v_j>\text{是 }E(G)\text{ 中的边}\\ 0\text{ 或 }\infty & \text{若}(v_i,v_j)\text{ 或}<v_i,v_j>\text{不是 }E(G)\text{ 中的边}\end{cases}$$

其中，W_{ij} 表示边(v_i,v_j)或$<v_i,v_j>$上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

2. 邻接表

邻接表是一种顺序分配和链式分配相结合的存储结构。它包括两个部分：一部分是链表；另一部分是向量。在邻接表中，对图中每个顶点建立一个单链表，第 i 个单链表中的结点包含了顶点 v_i 的所有邻接顶点。

在邻接表中一般首先为每个顶点都附设一个顶点结点，作为链表的起始结点，顶点结点的结构如图 3-20(a)所示。而每个链表除了顶点结点外，还需要普通的链表结点，普通的链表结点结构如图 3-20(b)所示。

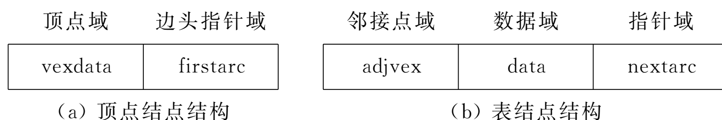


图 3-20　邻接表的结点结构

图 3-20 中，顶点结点由顶点域(vertex，该域给出的是顶点的信息)和指向第一个邻接点的指针域(firstarc)构成；另一种是表结点，它由邻接点域(adjvex，给出的是邻接点在图中的位置)和指向下一条邻接边的指针域(next)构成。对于网的边表需要再增设一个存储边上权值的数据域(data，该域存储权值信息，如果权值为 1，该域可以省略)。因此，图 3-16 中的无向图 G_1 和有向图 G_2 的邻接表存储结构如图 3-21 所示。

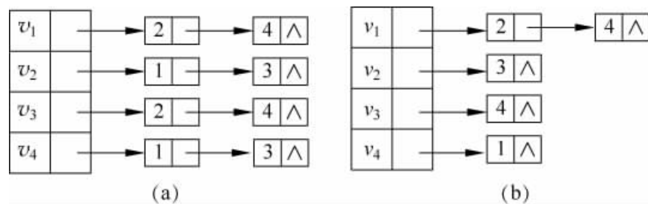


图 3-21　图 3-16 中 G_1 和 G_2 的邻接表

从邻接表的存储结构中可以发现，若无向图中有 n 个顶点、e 条边，则它的邻接表中含有 n 个顶点结点和 $2e$ 个表结点。在无向图的邻接表中，顶点 v_1 的度恰为第 i 个链接表中的结点数；而在有向图中，第 i 个链接表中的结点个数只是顶点 v_1 的出度，而要求入度，必须遍历整个邻接表。在邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，若要判定任意两个顶点之间是否有边或弧相连，则需搜索第 i 个或第 j 个链表，这一点不及邻接矩阵方便。

3.2.3　图的遍历方法

与树的遍历操作功能相似，图的遍历是指从图中的任一顶点出发，访问图中所有顶点，且保证每个顶点仅访问一次。图的遍历是图的一种基本操作，图的许多其他操作都是建立在遍历操作的基础之上的。

由于图本身结构的复杂性，因此图的遍历操作也较复杂，主要表现在以下几个方面。

(1) 在图中，没有一个确定的首结点，任意一个顶点都可以作为遍历的第一个结点。

(2) 在图中，一个顶点可以和其他多个顶点相连，当这样的顶点访问过后，存在如何选取下一个要访问的顶点的问题。

(3) 在图结构中，如果有回路存在，那么一个顶点被访问之后，有可能沿回路又回到该顶点，因此需要着重考虑如何保证每个顶点仅访问一次。

(4) 在非连通图中，从一个顶点出发，只能够访问它所在的连通分量上的所有顶点，因此，还需考虑如何选取下一个出发点以访问图中其余的连通分量。

图的遍历通常有深度优先搜索(Depth_First Search, DFS)和广度优先搜索(Breadth_First Search,BFS)两种方式，下面分别介绍。

1. 深度优先搜索

深度优先搜索遍历类似于树的先序遍历，是树的先序遍历的推广，其基本思想为假设初始状态是图中所有顶点未曾被访问，则深度优先所搜可从图中某个顶点 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径的顶点都被访问，然后重复上述过程，直到图中所有顶点都被访问到为止。具体的遍历步骤如下。

(1) 访问图 G 的指定起始点 v_1。

(2) 从 v_1 出发，访问一个与 v_1 邻接的顶点 w_1 后，再从 w_1 出发，访问与 w_1 邻接且未被访问过的顶点 w_2。从 w_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) 沿着刚才访问的次序，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；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

【例 3-3】 已知图的邻接表存储结构如图 3-22 所示，以顶点 v_1 为始点，按深度优先搜索遍历该图中，写出顶点的遍历序列。

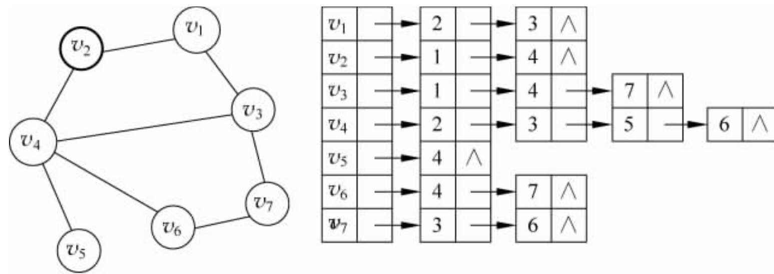


图 3-22 图的邻接表及其深度优先搜索序列

解：先访问 v_1，再访问 v_1 的第一个邻接点 v_2，再访问 v_2 的第一个邻接点 v_4，再访问 v_4 的第一个邻接点 v_2，因 v_2 已被访问过，则访问 v_4 的下一个邻接点 v_3，然后依次访问 v_7，v_6。这时，与 v_6 相邻接的顶点均已被访问，于是反向回到 v_4 去访问与 v_4 相邻接且尚未被访问的 v_5，至此，全部顶点均被访问。相应的访问序列为：$v_1 \to v_2 \to v_4 \to v_3 \to v_7 \to v_6 \to v_5$。

显然，这是一个递归的过程，需要递归遍历当前访问顶点的所有邻接点。而遍历规定，每个顶点只能被访问一次，因此，为了在遍历过程中便于区分顶点是否已被访问，需附设访问标志数组 visited[n]，其初值为 0，一旦某个顶点被访问，则其相应的分量置为 1。下面给出图的深度优先搜索的 C 语言算法，该算法假设图采用的是邻接表的存储结构。

算法 3-5：以图的邻接表为存储结构的深度优先搜索算法（该算法分为两部分，一部分是深度优先搜索算法 DFS 函数；另一部分是遍历函数 DFSTraver，在遍历函数中调用了 DFS 函数）

```
#define  VTXUNM   10                              /* 假设图中顶点个数的最大可能值为 10 */
    struct arcnode
    {                                             /* 定义表结点结构 */
        int adjvex;
        float data;
        struct arcnode *nextarc;
    } arcnode;
    typedef struct arcnode ARCNODE;
    struct headnode
    {                                             /* 定义顶点结点结构 */
        int vexdata;
        ARCNODE *firstarc;
    } headnode;
    struct headnode G[VTXUNM];
    int visited[VTXUNM];
    void dfs (struct headnode G[], int v)
    {
```

```
        struct arcnode *p;
        printf("%d->", G[v].vexdata);          /* 访问顶点 v */
        visited[v] = 1;                        /* 标记顶点 v 已经被访问 */
        p = G[v].firstarc;                     /* 读取顶点 v 的第一个邻接点的指针 */
        while (p!= NULL)                       /* 当邻接点存在时 */
        {
            if (visited[p->adjvex] == 0)
                dfs(G, p->adjvex);
            p = p->nextarc;                    /* 找下一个邻接点 */
        }
    }
    void DFSTraver(struct headnode G[])
    {
        int v;
        for(v = 0;v < VTXUNM; v++)             /* 初始化访问标志数组 */
            visited[v] = 0;
        for(v = 0; v < VTXUNM; v++)
            if ( visited[v] == 0)        /* 如果顶点 v 未被访问,则从顶点 v 出发开始遍历 */
                dfs(G, v);
    }
```

分析上述算法，在遍历时，对图中每个顶点至多调用一次 DFS 函数，一旦某个顶点被标记成已被访问，就不再从它出发进行搜索。因此，深度优先搜索遍历图的过程实质上是对每个顶点查找其邻接点的过程，耗费的时间则取决于所采用的存储结构。当用邻接矩阵存储结构时，查找每个顶点的邻接点所需时间为 $O(n^2)$，其中 n 为图中顶点数；而当采用邻接表存储结构时，找邻接点所需时间为 $O(e)$，其中 e 为无向图中的边数或有向图的弧数。

2. 广度优先搜索

广度优先搜索遍历类似于树的按层次遍历的过程。假设从图中某顶点 v 出发，在访问了 v 之后，依次访问 v 的各个未曾访问过和邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作为起始点，重复上述过程，直至图中所有顶点都被访问到为止。广度优先搜索遍历的具体步骤如下。

(1) 访问图 G 的指定起始点 v_1。

(2) 从 v_1 出发，依次访问 v_1 的未被访问过的邻接点 $w_1, w_2, \cdots, w_i$；然后依次从 $w_1, w_2, \cdots, w_i$ 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。

按照上述思想和步骤，图 3-22 中的邻接表存储结构按照广度优先搜索遍历的访问序列为：$v_1 \to v_2 \to v_3 \to v_4 \to v_7 \to v_5 \to v_6$。

与深度优先搜索类似，在广度优先搜索遍历也需要一个访问标志数组 visited[n]。同时，为了保证顺次访问每个顶点，广度优先搜索遍历还需要附设一个访问次序队列，用来存储已经被访问的顶点的次序，以便按照已被访问过的顶点次序先后访问它们未被访问的邻接点。广度优先遍历的非递归过程如算法 3.6 所示。

算法 3-6：广度优先搜索遍历的非递归算法

```
void BFSTraver(struct headnode G[ ], int v)
{                    /* 按广度优先非递归遍历图 G，使用辅助队列 Q 和访问标志数组 visited */
    int queue[VTXUNM];
    int rear = VTXUNM - 1; front = VTXUNM - 1;
    int i;
    ARCNODE *p;
    printf("%d->", G[v].vexdata);
    visited[v] = 1;
    rear = (rear + 1) % VTXUNM;
    queue[rear] = v;                          /* 访问过的顶点进队列 */
    while (rear!= front)
    {
        front = (front + 1) % VTXUNM;
        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 = (rear + 1) % VTXUNM;
                queue[rear] = p->adjvex;
            }
            p = p->nextarc;
        }
    }
}
```

分析上述算法，每个顶点至多进一次队列，因此广度优先搜索遍历图的过程实质是通过边或弧找邻接点的过程，因此广度优先遍历图的时间复杂度和深度优先搜索遍历相同，两者不同之处仅仅在于对顶点访问的顺序不同。

3.2.4 图的连通性与最小生成树

前面介绍的图的遍历算法是解决图的应用问题的基础，本节将介绍利用图的遍历算法来判断一个图的连通性问题，重点讨论无向图的连通性、有向图的连通性、由图得到其生成树或生成森林以及连通图的最小生成树(Minimum Spanning Tree，MST)问题。

1. 图的连通性

根据图的连通性的定义，对于连通图，仅需从图中任一顶点出发，进行深度优先搜索或广度优先搜索(只需调用一次 DFS 或 BFS 算法)，便可访问到图中所有顶点；而对非连通图，则需从多个顶点出发进行搜索(需调用多次 DFS 或 BFS 算法)。每次从一个新的顶点出发进行搜索过程中得到的顶点访问序列恰为其各个连通分量中的顶点集。图 3-23 中的无向图 G_{21} 是由两个连通分量构成的非连通图，按照深度优先搜索遍历需要从顶点 v_1 和 v_4 出发，调用两次 DFS 算法。

对于有向图而言，深度优先搜索是求其强连通分量的一个有效方法，求有向图的强连通

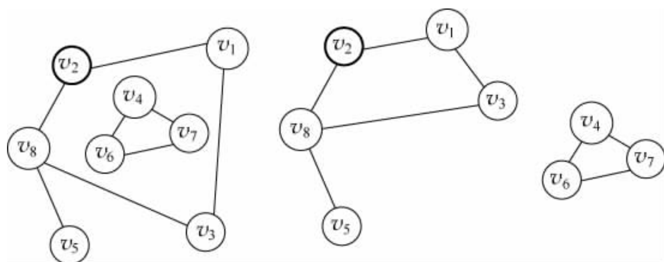


图 3-23 非连通无向图及其连通分量

分量的算法主要有 Kosaraju 算法、Gabow 算法和 Tarjan 算法，其中 Kosaraju 算法要对原图和逆图都进行一次 DFS，另外两种只要一次 DFS 就可以，具体的算法思想读者可以参阅相关文献，本书不再做详细介绍。

2. 生成树和生成森林

生成树：若图是连通图，从图中的某一个顶点出发进行遍历时，可以系统地访问图中所有顶点，此时图中所有的顶点加上遍历时经过的边所构成的子图，称为连通图的生成树。并且称由深度优先搜索得到的生成树为深度优先生成树；由广度优先搜索得到的生成树为广度优先生成树。由此可以得到图 3-22 中的图的深度优先生成树和广度优先生成树如图 3-24 所示。

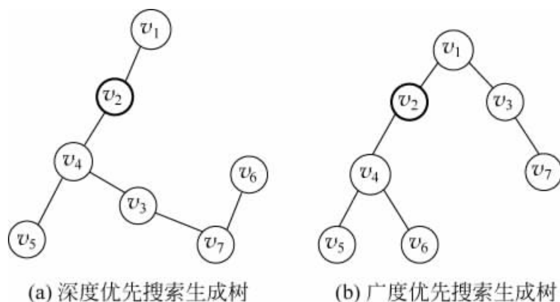


图 3-24 图 3-22 中所示图的生成树

对于非连通图，遍历过程会得到多个连通分量，每个连通分量中的顶点集和遍历时走过的边一起构成若干棵生成树，这些生成树组成非连通图的生成森林。

生成树和生成森林具有如下特点：

(1) 有 n 个顶点的连通图，其生成树有 $n-1$ 条边；

(2) 生成树中不含回路；

(3) 对于给定的图，其生成树或者生成森林不是唯一的，因为起点不同，访问的路径也不同。

3. 最小生成树

最后我们来讨论如何在一个带有权值的无向图中找出其最小生成树。生成树各边上的权值之和，称为生成树的代价，使代价最小的生成树，称为最小代价生成树，简称最小生成树。图 3-25 给出了一个最小生成树的示例。

最小生成树在实际生活中应用非常广泛。例如，要用最少的电线给一个房子安装电路，

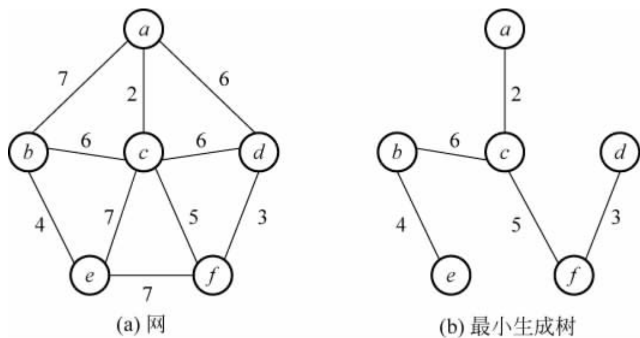


图 3-25　最小生成树示例

求解电路安装图的过程就是一个求解最小生成树的过程；再如要建 n 个城市之间的通信联络网，若以 n 个城市做图的顶点，$n-1$ 条线路做图的边，每条线路建造需付出一定代价（相当于边上的权），那么对 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，我们当然希望选择一个总耗费最小的生成树就是一个可行的方案。

最小生成树具有如下最基本的性质：设 $G=(V, E)$ 是一个连通图，U 是顶点集 V 的一个真子集，若 (u, v) 是一条具有最小权值的边，其中 $u \in U$，$v \in V-U$，则一定存在 G 的一棵最小生成树包含边 (u, v)。

而构造最小生成树的原则如下。

（1）尽可能选权值小的边，但不构成回路。

（2）在网中选 $n-1$ 条边以连接网中的 n 个顶点。

通常求最小生成树的算法有两种：Prim 算法和 Kruskal 算法，这里不再介绍，有兴趣的读者可参阅相关参考书。

习　　题

3-1　试分别画出具有 3 个结点的树和 3 个结点的二叉树的所有不同形态。

3-2　已知二叉树如图 3-26 所示，试用顺序存储和二叉链表两种方式给出存储结构，并分别给出该二叉树的先序、中序和后序遍历序列。

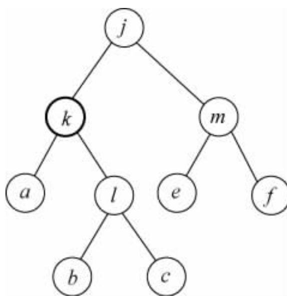


图 3-26　题 3-2 图

3-3　现有以下按先序和中序遍历二叉树的结果，问这样能否唯一地确定这棵二叉树的形状？为什么？如果能请画出该二叉树。如果给出的是先序和后序遍历能否唯一确定该二叉树？

先序：$ABCDEFGHI$

中序：$BCAEDGHFI$

3-4　假设二叉树采用二叉链表作为存储结构，编写计算二叉树中叶子（终端）结点数目的递归算法。

3-5　假设二叉树采用二叉链表作为存储结构，试编写一个判断该二叉树是否为完全二叉树的算法。

3-6　请画出图3-27中三棵树所对应的二叉树，并给出该森林所对应的二叉树，并给出该二叉树的先序和后序遍历序列。

3-7　什么叫哈夫曼树？给出一组权值{6,9,2,5,4,8}，请建立一棵哈夫曼树，并给出按照该哈夫曼树得到的哈夫曼编码。

3-8　有向图如题图3-28所示，请给出该图的邻接矩阵和邻接表，并求出各个顶点的度。

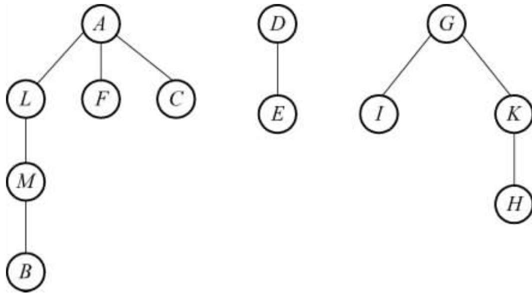


图3-27　题3-6图

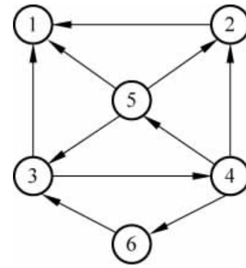


图3-28　题3-8图

3-9　已知某无向图的邻接矩阵为

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

，请画出该无向图，给出各个顶点的度，并画出该图的邻接表，分别给出从第一个结点出发的广度和深度优先生成树。