第5章 树和二叉树

本章和下一章分别介绍两种重要的非线性结构，即树和图。线性结构中的结点具有唯一前趋和唯一后继的关系，而非线性结构中结点之间的关系不再具有这种唯一性。其中，树形结构中结点之间的关系是前趋唯一而后继不唯一，即元素之间是一对多的关系；在图结构中结点之间的关系是前趋、后继均不唯一，因此也就无所谓前趋、后继了。直观地看，树结构既有分支关系，又有层次关系，它非常类似于自然界中的树。树形结构在现实世界中广泛存在，例如家谱、各单位的行政组织机构等都可以用树来表示。树在计算机领域中也有着广泛的应用，DOS 和 Windows 操作系统中对磁盘文件的管理就采用树形目录结构；在数据库中，树结构也是数据的重要组织形式之一。本章重点讨论二叉树的存储结构及其各种操作，并研究树和森林与二叉树的转换关系，最后介绍二叉树的一个应用实例——哈夫曼编码。

5.1 树的概念和基本操作

5.1.1 树的定义和相关术语

1. 树的定义

树(Tree)是 $n(n\geqslant 0)$个结点的有限集合。当 $n=0$ 时，称这棵树为空树；当 $n>0$ 时，该集合满足以下条件：

(1) 有且只有一个特殊的结点称为树的根(Root)，根结点没有直接前趋结点，但有零个或多个直接后继结点(这里的前趋、后继暂时沿用线性表中的概念，在树中所谓的前趋、后继其实有另外的术语)。

(2) 除根结点之外的其余 $n-1$ 个结点被分成 $m(m>0)$个互不相交的集合 T_1、T_2、…、T_m，其中每一个集合 $T_i(1\leqslant i\leqslant m)$本身又是一棵树，树 T_1、T_2、…、T_m 称为根结点的**子树**。

可以看出，在树的定义中用了递归概念，即用树来定义树。因此，树结构的算法也常常使用递归方法。

树的定义还可以形式化地描述为二元组的形式：

$$T=(D,R)$$

其中，D 为树 T 中结点的集合，R 为树中结点之间关系的集合。

当树 T 为空树时，$D=\Phi$；当树 T 不为空树时有

$$D=\{\text{Root}\}\cup D_F$$

其中，Root 为树 T 的根结点，D_F 为树 T 的根 Root 的子树集合。D_F 可由下式表示：

$$D_F=D_1\cup D_2\cup\cdots\cup D_m \quad \text{且 } D_i\cap D_j=\Phi \quad (i\neq j,1\leqslant i\leqslant m,1\leqslant j\leqslant m)$$

当树 T 中的结点个数 $n\leqslant 1$ 时，$R=\Phi$；当树 T 中的结点个数 $n>1$ 时有

$$R = \{< \text{Root}, r_i >, i = 1, 2, \cdots, m\}$$

其中,Root 为树 T 的根结点,r_i 是树 T 的根结点 Root 的子树 T_i 的根结点。

树定义的形式化主要用于树的理论描述。

图 5.1(a)是一棵具有 9 个结点的树 T,按照上述二元组的形式化描述有

T=({ A,B,C,D,E,F,G,H,I },{<A,B>,<A,C>,<B,D>,<B,E>,<B,F>,<C,G>,<E,H>,<E,I>})

其中,{ A,B,C,D,E,F,G,H,I }为树中结点的集合,{<A,B>,<A,C>,<B,D>,<B,E>, <B,F>,<C,G>,<E,H>,<E,I>}为树中结点之间关系的集合。例如,<A,B>表示 A 是 B 的直接前趋,B 是 A 直接后继,<A,B>称为树的一条分支。为了简化表示,也可以将如图 5.1(a)所示的树写成 T={ A,B,C,D,E,F,G,H,I }的形式。

在树 T 中,结点 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,其中 T_1={B,D,E,F,H,I},T_2={C,G},T_1 和 T_2 构成了结点 A 的两棵子树,T_1 和 T_2 本身也分别是一棵树。例如,子树 T_1 的根结点为 B,其余结点又分为 3 个不相交的集合,其中 T_{11}={D},T_{12}={E,H,I},T_{13}={F}。T_{11}、T_{12}和 T_{13}构成了子树 T_1 的根结点 B 的 3 棵子树。如此可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。

从树的定义和图 5.1(a)所示的示例可以看出,树具有下面两个特点:

(1) 树的根结点没有直接前趋,除根结点之外的所有结点有且只有一个直接前趋。

(2) 树中的所有结点都可以有零个或多个直接后继。

由此可知,图 5.1(b)、(c)、(d)所示的都不是树结构。

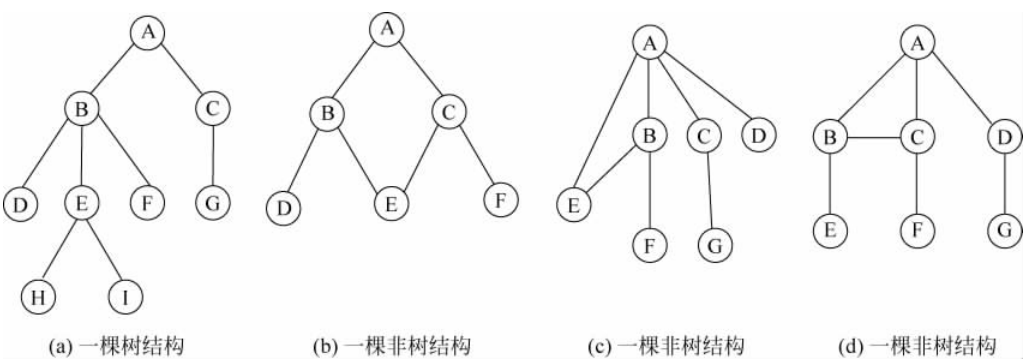


图 5.1 树结构和非树结构的示意图

2. 相关术语

(1) 结点。结点是包含一个数据元素及若干指向其他结点的分支信息的数据结构。

(2) 结点的度。结点所拥有的子树的个数称为该结点的度。

(3) 叶子结点。度为 0 的结点称为叶子结点,或者称为终端结点。

(4) 分支结点。度不为 0 的结点称为分支结点,或者称为非终端结点。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、双亲结点。树中一个结点的子树的根结点称为这个结点的孩子结点,这个结点称为它孩子结点的双亲结点。具有同一个双亲结点的孩子结点互称为兄弟结点。

(6) 路径、路径长度。设 $n_1, n_2, \cdots, n_k$ 为一棵树的结点序列,若结点 n_i 是 n_{i+1} 的双亲结点

($1 \leqslant i < k$)，则把 $n_1, n_2, \cdots, n_k$ 称为一条由 n_1 至 n_k 的路径。这条路径的长度是 $k-1$。

(7) 祖先、子孙。在树中，如果有一条路径从结点 M 到结点 N，那么 M 就称为 N 的祖先，而 N 称为 M 的子孙。

(8) 结点的层次。规定树的根结点的层数为 1，其余结点的层数等于它的双亲结点的层数加 1。

(9) 树的深度(高度)。树中所有结点的层次的最大数称为树的深度。

(10) 树的度。树中所有结点的度的最大值称为该树的度。

(11) 有序树和无序树。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，称这棵树为有序树；反之，称为无序树。

(12) 森林。$m(m \geqslant 0)$棵不相交的树的集合称为森林。自然界中树和森林是不同的概念，但在数据结构中，树和森林只有很小的差别。任何一棵树，删去根结点就变成了森林；反之，给森林增加一个统一的根结点，森林就变成了一棵树。

5.1.2 树的基本操作

树的基本操作通常有以下几种：

(1) Initiate (t)用于初始化一棵空树 t。

(2) Root (x)用于求结点 x 所在树的根结点。

(3) Parent (t,x)用于求树 t 中结点 x 的双亲结点。

(4) Child (t,x,i)用于求树 t 中结点 x 的第 i 个孩子结点。

(5) RightSibling (t,x)用于求树 t 中结点 x 右边的第一个兄弟结点(也称右兄弟结点)。

(6) Insert (t,x,i,s)用于把以 s 为根结点的树插入到树 t 中作为结点 x 的第 i 棵子树。

(7) Delete (t,x,i)用于在树 t 中删除结点 x 的第 i 棵子树。

(8) Traverse (t)是树的遍历操作，即按某种方式访问树 t 中的每个结点，且使每个结点只被访问一次。遍历操作是非线性结构中常用的基本操作，许多对树的操作都是借助该操作实现的。

5.2 二 叉 树

在进一步讨论树之前，先讨论一种简单而又非常重要的树形结构——二叉树。由于任何树都可以转换为二叉树进行处理，而二叉树又有许多好的性质，非常适合于计算机处理，因此二叉树也是数据结构研究的重点。

5.2.1 二叉树的基本概念

1. 二叉树的定义

二叉树(Binary Tree)是 n 个结点的有限集合，该集合或者为空，或者由一个称为根(Root)的结点及两个不相交的、被分别称为根结点的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。

由此定义可以看出，一棵二叉树中的每个结点只能含有 0、1 或 2 个孩子结点，而且其孩子

结点有左、右之分,位于左边的称左孩子,位于右边的称右孩子。显然,二叉树是有序的,若将其左、右孩子颠倒,就成为另一棵不同的二叉树。即使二叉树中的结点只有一棵子树,也要区分它是左子树还是右子树。

图5.2给出了二叉树的5种基本形态。图5.2(a)所示为一棵空的二叉树;图5.2(b)所示为一棵只有根结点的二叉树;图5.2(c)所示为一棵只有左子树的二叉树;图5.2(d)所示为一棵只有右子树的二叉树;图5.2(e)所示为一棵左、右子树均不为空的二叉树。

尽管二叉树的定义不同于树,在理论上二叉树和树是两个不同的概念,但两者都属于树形结构,因此,前面有关树的一些术语同样适用于二叉树。

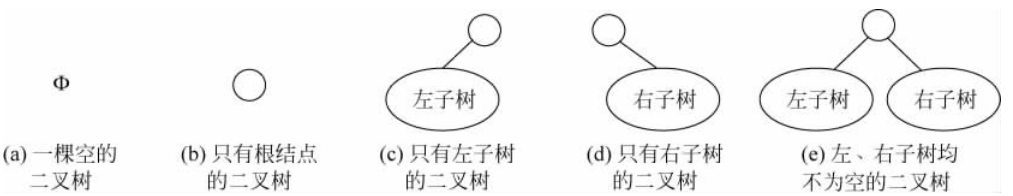


图5.2 二叉树的5种基本形态

2. 二叉树的相关概念

1) 满二叉树

在一棵二叉树中,如果所有的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,这样的一棵二叉树称为满二叉树。

如图5.3所示,(a)图就是一棵满二叉树,(b)图则不是满二叉树,因为虽然其所有结点要么是含有左、右子树的分支结点,要么是叶子结点,但由于其叶子未在同一层上,故不是满二叉树。

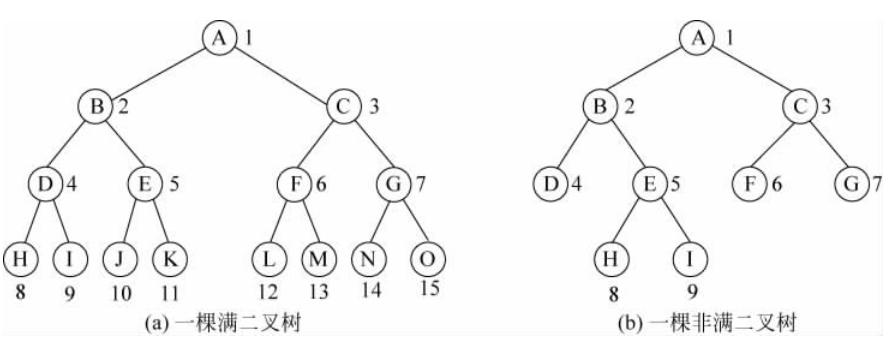


图5.3 满二叉树和非满二叉树示意图

2) 完全二叉树

一棵深度为 k 的有 n 个结点的二叉树,对树中的结点按从上至下、从左到右的顺序进行编号,如果编号为 $i(1\leqslant i\leqslant n)$ 的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同,则这棵二叉树称为完全二叉树。

完全二叉树的特点是叶子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。显然,一棵满二叉树必定是一棵完全二叉树,但完全二叉树未必是满二叉树。例如,图5.4(a)所示为一棵完全二叉树,而图5.4(b)和图5.3(b)都不是完全二叉树。

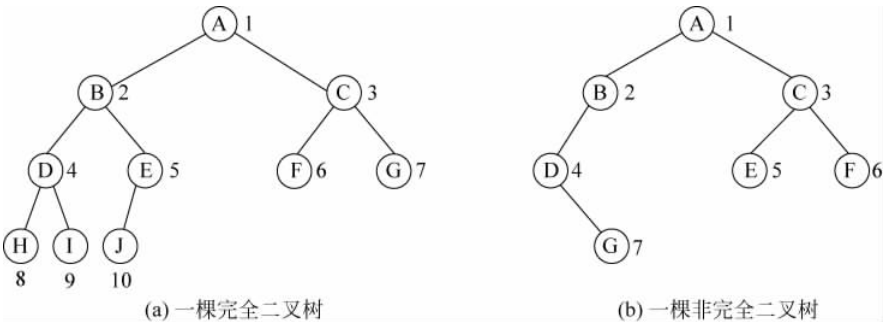


图 5.4 完全二叉树和非完全二叉树示意图

5.2.2 二叉树的主要性质

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

证明：用数学归纳法证明。

当 $i=1$ 时，$2^{i-1}=2^{1-1}=2^0=1$，第一层只有根结点，结论成立。当 $i \geqslant 1$ 时，设第 $i-1$ 层上最多有 $2^{(i-1)-1}=2^{i-2}$ 个结点，因为是二叉树，每个结点最多有两个孩子结点，所以第 i 层上最多有 $2 \cdot 2^{i-2}=2^{i-1}$ 个结点。

性质 2 一棵深度为 k 的二叉树中最多有 2^k-1 个结点。

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

$$M = \sum_{i=1}^{k} x_i \leqslant \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

性质 3 对于一棵非空二叉树，如果叶子结点数为 n_0，度数为 2 的结点数为 n_2，则有 $n_0=n_2+1$。

证明：设 n 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

$$n = n_0 + n_1 + n_2 \tag{5-1}$$

在二叉树中，除根结点外，其余结点都有唯一的一个进入分支。设 B 为二叉树中的分支数，那么有

$$B = n - 1 \tag{5-2}$$

这些分支是由度为 1 和度为 2 的结点发出的，一个度为 1 的结点发出一个分支，一个度为 2 的结点发出 2 个分支，所以有

$$B = n_1 + 2n_2 \tag{5-3}$$

综合式(5-1)、式(5-2)、式(5-3)可以得到

$$n_0 = n_2 + 1$$

性质 4 具有 n 个结点的完全二叉树的深度 k 为 $\lfloor \log_2 n \rfloor + 1$(其中，$\lfloor \log_2 n \rfloor$ 表示不大于 $\log_2 n$ 的最大整数)。

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

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

即

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

对不等式取对数,有

$$k-1 \leqslant \log_2 n < k$$

由于 k 是整数,所以有 $k=\lfloor \log_2 n \rfloor +1$。

性质5 对于具有 n 个结点的完全二叉树,如果按照从上至下和从左到右的顺序对二叉树中的所有结点从1开始顺序编号,则对于任意的编号为 i 的结点以下几点成立:

(1) 如果 $i>1$,则该结点 i 的双亲结点的编号为 $\lfloor i/2 \rfloor$;如果 $i=1$,则该结点是根结点,无双亲结点。

(2) 如果 $2i \leqslant n$,则该结点 i 的左孩子结点的编号为 $2i$;如果 $2i>n$,则该结点 i 无左孩子结点。

(3) 如果 $2i+1 \leqslant n$,则该结点 i 的右孩子结点的编号为 $2i+1$;如果 $2i+1>n$,则该结点 i 无右孩子结点。

此外,若对完全二叉树的根结点从0开始编号,则相应的结点 i 的双亲结点的编号为 $\lfloor (i-1)/2 \rfloor$,左孩子的编号为 $2i+1$,右孩子的编号为 $2i+2$。

证明:先证明结论(2)和结论(3)成立,用数学归纳法。

当 $i=1$ 时,结点是根,由完全二叉树的定义可知,若根有左孩子,则根的左孩子的编号为2,若根有右孩子,则其右孩子的编号3。于是,当 $i=1$ 时,若 $2i \leqslant n$,则根结点有左孩子,左孩子的编号为 $2=2\times 1=2i$;若 $2i>n$,则根结点无左孩子;若 $2i+1 \leqslant n$,则根结点有右孩子,右孩子的编号为 $3=2\times 1+1=2i+1$;若 $2i+1>n$,则根结点无右孩子。

由完全二叉树的定义和性质2可知,第 j 层($1 \leqslant j \leqslant \lfloor \log_2 n \rfloor$)上的结点的编号从 2^{j-1} 至 2^j-1。

当 $i=2^{j-1}$ 时,i 为第 j 层的第一个结点(最左端的结点),则第 $j+1$ 层的第一个结点就是 i 的左孩子,其编号为第 j 层最后一个结点编号加1,即 2^j,$2^j=2\times 2^{j-1}=2i$。若 i 有右孩子,则第 $j+1$ 层的第二个结点就是 i 的右孩子,其编号为 $2^j+1=2\times 2^{j-1}+1=2i+1$。

现假设 $2^{j-1} \leqslant i < 2^j-1$,结论(2)和结论(3)成立,则结点 $i+1$ 的左孩子编号就是结点 i 的右孩子编号加1,即 $(2i+1)+1=2(i+1)$,而结点 $i+1$ 的右孩子编号必为其左孩子编号加1,即 $2(i+1)+1$。

于是,若 $2(i+1) \leqslant n$,则结点 $i+1$ 有左孩子,其左孩子编号为 $2(i+1)$;若 $2(i+1)>n$,则结点 $i+1$ 无左孩子。若 $2(i+1)+1 \leqslant n$,则结点 $i+1$ 有右孩子,其右孩子的编号为 $2(i+1)+1$;若 $2(i+1)+1>n$,则结点 $i+1$ 无右孩子。结论(2)和结论(3)得证。

再证明结论(1)成立。当 $i=1$ 时,i 为根结点,显然没有双亲。因为 i 必为偶数或奇数,所以当 $i>1$ 时,若 i 为偶数,则存在某个 j 满足 $i=2j$,根据结论(2)可知,i 就是 j 的左孩子,反过来 j 就是 i 的双亲;若 i 为奇数,则存在某个 j 满足 $i=2j+1$,根据结论(3)可知,i 就是 j 的右孩子,反过来 j 是 i 的双亲。综合得知,当 $i>1$ 时,i 的双亲为 $j=\lfloor i/2 \rfloor$。结论(1)得证。

5.2.3 二叉树的存储结构与基本操作

1. 二叉树的顺序存储结构

所谓二叉树的顺序存储,就是用一组连续的存储单元存放二叉树中的结点。一般是按照二叉树中的结点从上至下、从左到右的顺序存储。这样结点在存储位置上的前趋、后继关系并不一定就是它们在逻辑上的邻接关系,然而只有通过一些方法确定某结点在逻辑上的双亲结点和孩子结点,这种存储才有意义。因此,依据二叉树的性质,完全二叉树和满二叉树采用顺

序存储比较合适，树中结点的序号可以唯一地反映出结点之间的逻辑关系，这样既能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 5.5 给出的是图 5.4(a)所示的完全二叉树的顺序存储示意图。

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

图 5.5　图 5.4(a)所示的完全二叉树的顺序存储示意图

对于一般的二叉树，如果仍按从上至下和从左到右的顺序将树中的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，只有增加一些并不存在的空结点，使之成为一棵完全二叉树的形式，然后再用一维数组顺序存储。图 5.6 给出了一棵一般二叉树改造后的完全二叉树形态(虚线表示在原二叉树中不存在的结点)及其顺序存储状态示意图。显然，这种存储对于需增加许多空结点才能将一棵二叉树改造成一棵完全二叉树的存储时会造成空间的大量浪费，不宜用顺序存储结构。最坏的情况是右单支树，如图 5.7 所示，一棵深度为 k 的右单支树只有 k 个结点，却需分配了 2^k-1 个存储单元。

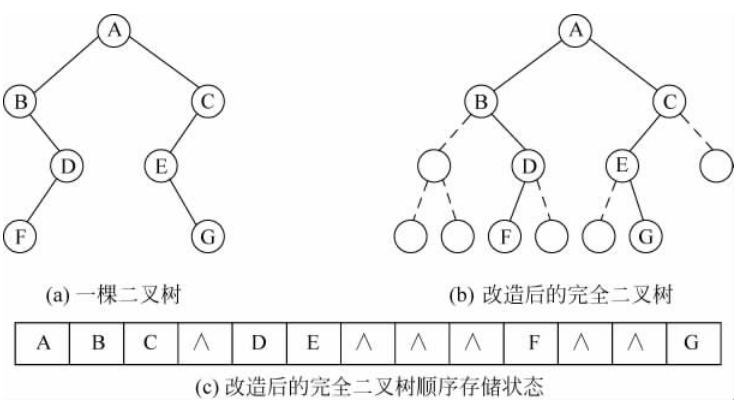


图 5.6　一般二叉树及其顺序存储示意图

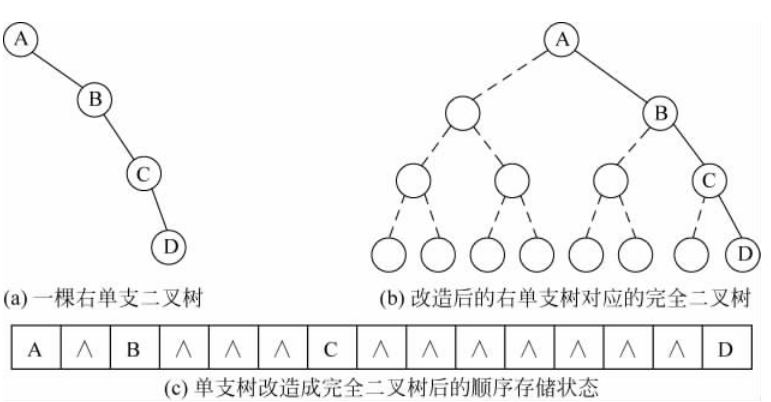


图 5.7　右单支二叉树及其顺序存储示意图

二叉树的顺序存储表示可描述为：

```
#define MAXNODE                          /* 二叉树的最大结点数 */
typedef elemtype SqBiTree[MAXNODE]       /* 0 号单元存放根结点 */
```

```
SqBiTree bt;
```

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

2. 二叉树的链式存储结构

二叉树的链式存储结构是指用链表结构来表示一棵二叉树,即用链指针来指示其元素的逻辑关系,通常有下面两种形式。

(1) 二叉链表存储。链表中的每个结点由 3 个域组成,除了数据域外还有两个指针域,分别用于给出该结点左孩子和右孩子所在的链结点的存储地址。

结点的存储结构如下:

lchild	data	rchild

其中,data 域存放结点的数据信息; 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,相应指针的域值为空(用符号 ∧ 或 NULL 表示)。

图 5.8 是一棵二叉树,图 5.9(a)给出了其二叉链表。

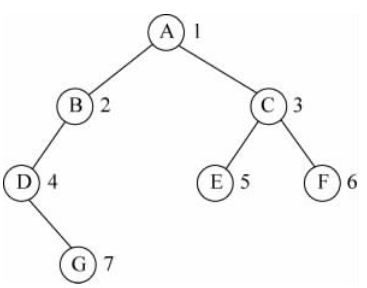


图 5.8 一棵非完全二叉树

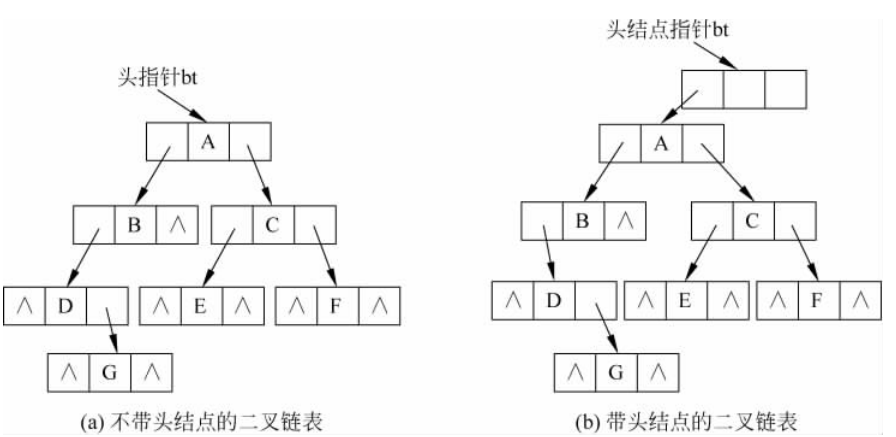


图 5.9 图 5.8 所示的二叉树的二叉链表示意图

二叉链表也可以用带头结点的方式存放,如图 5.9(b)所示。

(2) 三叉链表存储。每个结点由 4 个域组成,具体结构如下:

lchild	data	rchild	parent

其中，data、lchild 和 rchild 3 个域的意义同二叉链表结构；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而言，它增加了空间开销。

图 5.10 给出了图 5.8 所示的二叉树的三叉链表表示。

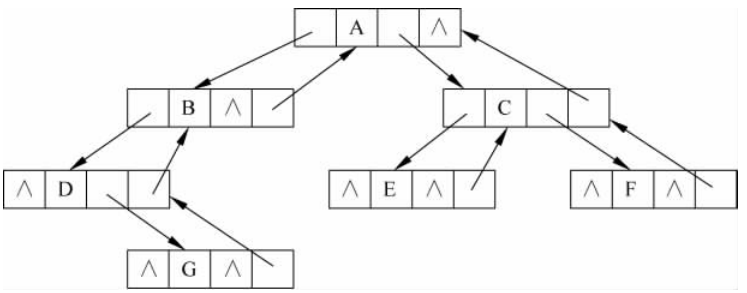


图 5.10　图 5.8 所示的二叉树的三叉链表示意图

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。因此，二叉链表是最常用的二叉树存储方式，本书后面所涉及的二叉树的链式存储结构不加特别说明都是指二叉链表结构。

二叉树的二叉链表存储表示可描述为：

```
typedef  struct  BiTNode
      {  elemtype data;
         struct BiTNode * lchild, * rchild;    /* 左、右孩子指针 */
       }  BiTNode, * BiTree;
```

即将 BiTree 定义为指向二叉链表结点结构的指针类型。

3. 二叉树的基本操作

二叉树的基本操作通常有以下几种：

(1) Initiate (bt)用于建立一棵空二叉树。

(2) Create (x,lbt,rbt)用于生成一棵以 x 为根结点的数据域信息、以二叉树 lbt 和 rbt 为左子树和右子树的二叉树。

(3) InsertL (bt,x,parent)用于将数据域信息为 x 的结点插入二叉树 bt 中作为结点 parent 的左孩子结点。如果结点 parent 原来有左孩子结点，则将其作为结点 x 的左孩子结点。

(4) InsertR (bt,x,parent)用于将数据域信息为 x 的结点插入二叉树 bt 中作为结点 parent 的右孩子结点。如果结点 parent 原来有右孩子结点，则将其作为结点 x 的右孩子结点。

(5) DeleteL (bt,parent)用于在二叉树 bt 中删除结点 parent 的左子树。

(6) DeleteR (bt,parent)用于在二叉树 bt 中删除结点 parent 的右子树。

(7) Search (bt,x)用于在二叉树 bt 中查找数据元素 x。

(8) Traverse (bt)用于按某种方式遍历二叉树 bt 中的全部结点。

5.2.4　二叉树的遍历

二叉树的遍历是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并且

只被访问一次。

遍历是二叉树中经常要用到的一种操作,因为在实际应用问题中经常需要按一定的顺序对二叉树中的每个结点逐个访问,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以使二叉树中的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,遍历操作可以使非线性结构线性化。

由二叉树的定义可知,一棵二叉树由根结点、根结点的左子树和根结点的右子树 3 个部分组成。因此,只要依次遍历这 3 个部分,就可以遍历整个二叉树。若以 D、L、R 分别表示访问根结点、遍历根结点的左子树和右子树,则二叉树的遍历方式有 6 种,即 DLR、LDR、LRD、DRL、RDL 和 RLD。由于左、右具有对称性,因此可以限定先左后右,则只有前 3 种方式,即 DLR(前序遍历或先序遍历)、LDR(中序遍历)和 LRD(后序遍历)。

1. 先序遍历

先序遍历(DLR)的递归过程是:若二叉树为空,遍历结束,否则进行以下操作:

(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。

算法 5.1 先序遍历二叉树的递归算法。

```
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 */
   }
```

对于图 5.8 所示的二叉树,按先序遍历得到的结点序列如下:

A B D G C E F

算法 5.2 先序遍历二叉树的非递归算法。

分析:先序遍历的过程是首先访问根结点,然后先序遍历根的左子树,最后先序遍历根的右子树。对于根的左子树和右子树,遍历的过程相同。如果用非递归方法,就要在遍历左子树之前先保存右子树根结点的地址(指针),以便在完成左子树的遍历之后取出右子树根结点的地址,再遍历这棵右子树。同样,在遍历左子树的左子树之前也要先保存左子树的右子树根结点的地址,以此类推。可以看出,对这些地址的保存和取出符合后进先出的原则,可设置一个辅助栈来保存这些右子树根结点的地址。为了方便编写算法,这个辅助栈保存所有经过的结点的指针,包括空的根指针和空的孩子指针。先序遍历二叉树的非递归算法如下:

```
void PreOrderNonRec(BiTree bt)
{                                             /* 先序遍历二叉树 bt 的非递归算法 */
  Stack s;                                 /* 设辅助栈的类型为 Stack ,栈中存储二叉链表结点的地址 */
  BiTree p;
  Init_Stack ( &s );                          /* 初始化栈 s */
  Push_Stack ( &s,bt );                       /* 根结点的地址 bt 入栈 s ,包括空的二叉树 */
  while(!Empty_Stack ( s )) {                 /* 栈 s 非空执行循环体 */
    p = Top_Stack ( s );                      /* 取栈顶元素 */
    while(p!= NULL) {
```

```
        Visit(p->data);
        Push_Stack (&s,p->lchild);          /* 向左走到尽头,空的左孩子指针也入栈 */
          p = Top_Stack ( s );              /* 取栈顶元素 */
    }
     Pop_Stack (&s);                        /* 空指针退栈,栈中不可能有两个连续的空指针 */
     if (!Empty_Stack ( s )) {
        p = Pop_Stack (&s);
        Push_Stack (&s,p->rchild);          /* 向右一步,右孩子的地址入栈 */
    }
 }
}
```

2. 中序遍历

中序遍历(LDR)的递归过程是：若二叉树为空,遍历结束,否则执行以下操作：

(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。

算法 5.3　中序遍历二叉树的递归算法。

```
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 */
   }
```

对于图 5.8 所示的二叉树,按中序遍历得到的结点序列如下：

D　G　B　A　E　C　F

算法 5.4　中序遍历二叉树的非递归算法。

分析：中序遍历的过程是首先中序遍历左子树,然后访问根结点,最后中序遍历根的右子树。对于根的左子树和右子树,遍历的过程相同。如果用非递归方法,就要在遍历左子树之前先保存根结点的地址(指针),以便在完成左子树的遍历之后取出根结点的地址访问根结点,然后再中序遍历右子树。同样,在中序遍历左子树的左子树之前也要先保存左子树的根结点地址,以此类推。可以看出,对这些地址的保存和取出符合后进先出的原则,可设置一个辅助栈来保存所经过的结点的地址。为了方便编写算法,栈中也保存空树的空指针。中序遍历二叉树的非递归算法如下：

```
void InOrderNonRec(BiTree bt)
{                                           /* 中序遍历二叉树 bt 的非递归算法 */
  Stack s;                                  /* 设栈类型为 Stack */
  BiTree p;
  Init_Stack ( &s );                        /* 初始化栈 s */
  Push_Stack ( &s,bt );                     /* 根结点的指针 bt 入栈 s */
  while(!Empty_Stack ( s )) {
    p = Top_Stack ( s );
    while(p!= NULL) {
        Push_Stack (&s,p->lchild);          /* 向左走到尽头,空的左孩子指针也入栈 */
        p = Top_Stack ( s );
    }
```

```
        p = Pop_Stack (&s);                          /* 空指针退栈 */
        if (!Empty_Stack ( s )) {
           p = Pop_Stack (&s);
           Visit(p->data);                           /* 访问当前根结点 */
           Push_Stack (&s,p->rchild);                /* 向右一步,右孩子的指针入栈 */
       }
    }
}
```

3. 后序遍历

后序遍历(LRD)的递归过程是：若二叉树为空,遍历结束,否则执行以下操作：

(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。

算法 5.5 后序遍历二叉树的递归算法。

```
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 */
   }
```

对于图 5.8 所示的二叉树,按后序遍历得到的结点序列如下：

G D B E F C A

算法 5.6 后序遍历二叉树的非递归算法。

分析：后序遍历的过程是首先后序遍历左子树,然后后序遍历根的右子树,最后访问根结点。如果用非递归方法,就要在遍历左子树之前先保存根结点的地址,以便在完成左子树遍历之后根据根结点的地址遍历右子树和访问根结点。对于根的左子树和根的右子树,遍历的过程相同。对这些地址的保存和使用符合后进先出的原则,可设置一个辅助栈来保存所经过的结点的地址。因为后序遍历的特点是只有在遍历了左子树和右子树之后才能访问根结点,所以为了表明子树是否被遍历过,可再设置一个辅助变量。

后序遍历二叉树的非递归算法如下：

```
void PostOrderNonRec(BiTree bt)
{                                                    /* 后序遍历二叉树 bt 的非递归算法 */
  Stack s;                                           /* 设栈类型为 Stack */
  BiTree p,q;                                        /* q 指向最近被访问过的结点,用作标志 */
  Init_Stack ( &s );                                 /* 初始化栈 s */
  p = bt;
  do {
      while(p)                                       /* 向左走到尽头,左孩子指针入栈 */
      {     Push_Stack(&s,p);
            p = p->lchild;
       }
       q = NULL;
       while(!Empty_Stack ( s )) {
            p = Top_Stack ( s );
            if(p->rchild == NULL )||(p->rchild == q )) {        /* 右子树为空或已经访问过 */
```

```
            Visit(p->data);                          /*访问当前的根结点*/
            q=p;
            Pop_Stack(&s);                           /*退栈*/
          }
          else{ /*右子树非空且未被遍历*/
              p=p->rchild;                           /*向右一步*/
              break;
          }
        }/*while*/
 }while(!Empty_Stack(s));
}
```

4. 层次遍历

二叉树的层次遍历是指从二叉树的第一层(根结点)开始从上至下逐层遍历,在同一层中则按从左到右的顺序对结点逐个访问。对于图5.8所示的二叉树,按层次遍历得到的结果序列如下:

A B C D E F G

下面讨论层次遍历的算法。

由层次遍历的定义可以推知,在进行层次遍历时,对一层结点访问完后再按照它们的访问次序对各个结点的左孩子和右孩子顺序访问,这样一层一层地进行,先遇到的结点先访问,这与队列的操作原则比较吻合。因此,在进行层次遍历时可设置一个队列结构,遍历从二叉树的根结点开始,首先将根结点指针入队列,然后从队头取出一个元素,每取一个元素执行下面两个操作:

(1) 访问该元素所指的结点;

(2) 若该元素所指结点的左、右孩子指针非空,则将该元素所指结点的非空左孩子指针和右孩子指针顺序入队。

若队列非空,重复以上过程,当队列为空时,二叉树的层次遍历结束。

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

算法5.7 二叉树的层次遍历算法。

```
void  LevelOrder(BiTree bt)                          /*层次遍历二叉树bt*/
    {  BiTree  Queue[MAXNODE];
       int front, rear;
       if(bt==NULL) return;
       front=-1;                                     /*队列初始化*/
       rear=0;
       queue[rear]=bt;                               /*根结点入队*/
       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;
              }
          }
      }
```

5.2.5 二叉树的其他操作举例

例 5.1 查找数据元素。

Search (bt,x) 在以 bt 为根结点指针的二叉树中查找数据元素 x,查找成功返回该结点的指针,查找失败返回空指针。

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

算法 5.8 二叉树的元素查找算法。

```
BiTree  Search(BiTree * bt,elemtype x)
    {                                          /* 在 bt 为根结点指针的二叉树中查找数据元素 x */
        BiTree p;
        p = bt;
        if(p -> data == x) return p;           /* 查找成功返回 */
        if( p -> lchild!= NULL)      /* 在 p -> lchild 为根结点指针的二叉树中查找数据元素 x */
          return (Search (p -> lchild, x) );
        if( p -> rchild!= NULL)      /* 在 p -> rchild 为根结点指针的二叉树中查找数据元素 x */
          return (Search (p -> rchild, x));
        return NULL;                           /* 查找失败返回 */
     }
```

例 5.2 统计出给定二叉树中叶子结点的数目。

(1) 可以利用中序递归遍历算法求二叉树中叶子结点的个数,其算法如下:

算法 5.9 利用中序遍历求二叉树的叶子结点的个数的算法。

```
void  inorder_leaf (BiTree * bt)
    {  if (bt!= NULL)
        {  inorder_leaf (bt -> lchild);
           printf (h -> data);
           if (bt -> lchild == NULL)&&(bt -> rchild == NULL) k++;
           inorder_leaf (bt -> rchild);
         }
    }
```

上面函数中的 k 是全局变量,在主程序中先置零,在调用 inorder_ leaf 后 k 值就是二叉树 bt 中叶子结点的个数。

(2) 还可以用另一种递归来实现该操作,其算法如下:

算法 5.10 求二叉树中叶子结点个数的另一种递归算法。

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

例 5.3 求二叉树的深度算法。

求深度算法同样可以通过遍历操作来实现,算法如下:

算法 5.11 求二叉树的深度算法。

```
int  treehigh (BiTree * bt)
  {  int lh,rh,h;
     if (bt == NULL) h = 0;
     else
        {  lh = treehigh(bt -> lchild);
           rh = treehigh (bt -> rchild);
           h = (1h> rh?1h: rh) + 1;
         }
     return h;
  }
```

例 5.4 创建二叉树的二叉链表存储并显示。

设创建时按二叉树带空指针的前序(即先输入根结点,再输入左子树,最后输入右子树)的次序输入结点值,结点值的类型为字符型,输出按中序遍历序列输出。

CreateBinTree (BinTree * bt)是以二叉链表为存储结构建立一棵二叉树 T 的存储,bt 为指向二叉树 T 的根结点指针的指针。设建立时的输入序列为 A B 0 D 0 0 C E 0 0 F0 0,其中,0 表示空结点,则建立二叉树的二叉链表如图 5.11 所示。

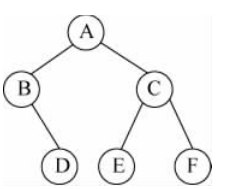


图 5.11 二叉树

InOrderOut (bt)为按中序输出二叉树 bt 的结点。

该算法的实现如下,注意在以下创建算法中,遍历算法中的 Visit(bt－>data)被读入结点、申请空间和存储的操作所代替;在输出算法中,遍历算法中的 Visit(bt－>data)被 C 语言中的格式输出语句所代替。

```
void  CreateBinTree (BinTree * T)
      {    /* 以先序序列输入结点的值,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 (BinTNode));                    /* 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 (BinTree 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);
 }
```

例 5.5 已知结点的前序序列为 ABCDEFG,中序序列为 CBEDAFG,可按上述分解求得整棵二叉树,其构造过程如图 5.12 所示。首先由前序序列得知二叉树的根为 A,则其左子树的中序序列为 CBED,右子树的中序序列为 FG。反过来得知其左子树的前序序列为 BCDE,右子树的前序序列为 FG。类似地,可由左子树的前序序列和中序序列构造得 A 的左子树,由右子树的前序序列和中序序列构造得 A 的右子树。

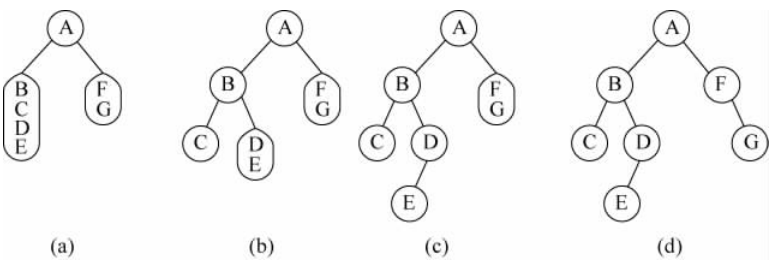


图 5.12 由前序和中序序列构造一棵二叉树的过程

上述构造过程说明了给定结点的前序序列和中序序列可确定一棵二叉树,至于它的唯一性,读者可用归纳法证明。

5.3 树和森林

5.3.1 树的存储

在计算机中,树的存储有多种方式,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,都要求存储结构不仅能存储各结点本身的数据信息,还能唯一地反映树中各结点之间的逻辑关系。下面介绍几种树的基本存储方式。

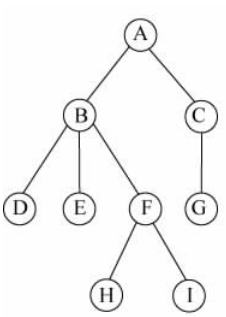
1. 双亲表示法

由树的定义可知,树中的每个结点(除根结点外)都有唯一的一个双亲结点,根据这一特性,可用一组连续的存储空间(一维数组)存储树中的各个结点,数组中的一个元素表示树中的一个结点,数组元素为结构体类型,其中包括结点本身的信息以及结点的双亲结点在数组中的序号,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的定义可描述为:

```
#define  MAXNODE  100                    /* 树中结点的最大个数 */
typedef  struct
    {  elemtype  data;
       int  parent;
     }  NodeType;
 NodeType t[MAXNODE];
```

图 5.13(a) 所示的树的双亲表示如图 5.13(b)所示,该图中用 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。



(a) 树的逻辑结构

序号	data	parent
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2
7	H	5
8	I	5

(b) 树的存储结构（双亲表示法）

图 5.13 树的双亲表示法示意图

树的双亲表示法对于实现 Parent (t,x) 操作和 Root (x) 操作很方便，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) 操作比较困难。在实际操作中，如果需要实现这些操作，可在结点结构中增加存放第一个孩子的域和存放右兄弟的域，这样就能较方便地实现上述操作了。

2. 孩子表示法

孩子表示法是将树按如图 5.14 所示的形式存储，其主体是一个与结点个数一样大小的一维数组，数组的每一个元素由两个域组成，一个域用来存放结点本身的信息，另一个用来存放指针，该指针指向由该孩子结点组成的单链表的首位置。单链表的基本结构也由两个域组成，一个存放孩子结点在一维数组中的序号，另一个是指针域，指向下一个孩子。

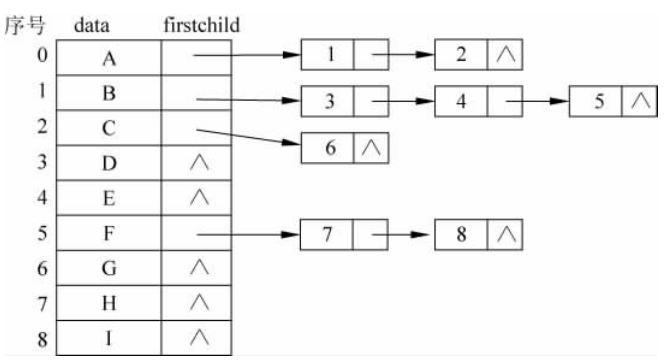


图 5.14 图 5.13 (a) 所示的树的孩子表示法示意图

显然，在孩子表示法中查找双亲比较困难，查找孩子却十分方便，故孩子表示法适用于对孩子操作多的情况。这种表示法的存储结构可描述为：

```
#define MAXNODE 100                                  /* 树中结点的最大个数 */
typedef  struct  ChildNode
      {  int  childcode;
         struct  ChildNode *nextchild;
      }
typedef struct
      {  elemtype  data;
         struct ChildNode *firstchild;
```

```
    } NodeType;
NodeType t[MAXNODE];
```

3. 孩子兄弟表示法

孩子兄弟表示法是一种常用的存储结构。其方法是：在树中，每个结点除信息域以外，再增加两个分别指向该结点的第一个孩子结点和右兄弟结点的指针。在这种存储结构下，树中结点的存储结构可描述为：

```
typedef  struct  TreeNode
        {  elemtype  data;
           struct TreeNode *firstchild;
           struct TreeNode *nextsibling;
         }  NodeType;
```

定义一棵树如下：

```
NodeType *t;
```

图 5.15 给出了图 5.13 (a) 所示的树采用孩子兄弟表示法时的存储示意图。从该图可以看出，该存储结构与二叉树的二叉链表结构非常相似，而且如果去除了字面上的含义，其实质是一样的。因此，树、森林与二叉树的转换可以方便地实现。

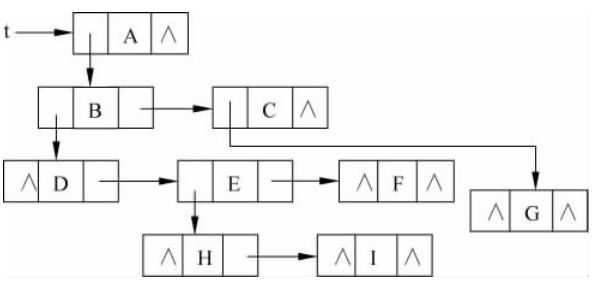


图 5.15　图 5.13(a)所示的树的孩子兄弟表示法示意图

5.3.2　树、森林与二叉树的相互转换

从树的孩子兄弟表示法可以看出，如果设定一定的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。本节讨论树和森林与二叉树之间的转换方法。

1. 将树转换为二叉树

对于一棵无序树，树中结点的各孩子结点的次序是无关紧要的，而二叉树中结点的左、右孩子结点是有区别的。为了避免发生混淆，约定树中每一个结点的孩子结点按从左到右的次序顺序编号。例如图 5.16 所示的一棵树，根结点 A 有 B、C、D 3 个孩子，可以认为结点 B 为 A 的第一个孩子结点，结点 C 为 A 的第二个孩子结点，结点 D 为 A 的第三个孩子结点。

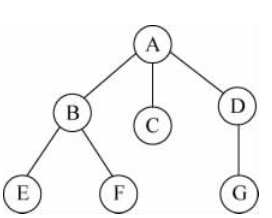


图 5.16　一棵树

将一棵树转换为二叉树的方法如下：

(1) 在树中所有相邻兄弟之间加一条连线；

(2) 对于树中的每个结点，只保留它与第一个孩子结点之间的连线，删去它与其他孩子结点之间的连线；

(3) 以树的根结点为轴心,将整棵树顺时针转动一定的角度,使之结构层次分明。

可以证明,树做这样的转换所构成的二叉树是唯一的。图 5.17 给出了图 5.16 所示的树转换为二叉树的过程示意图。

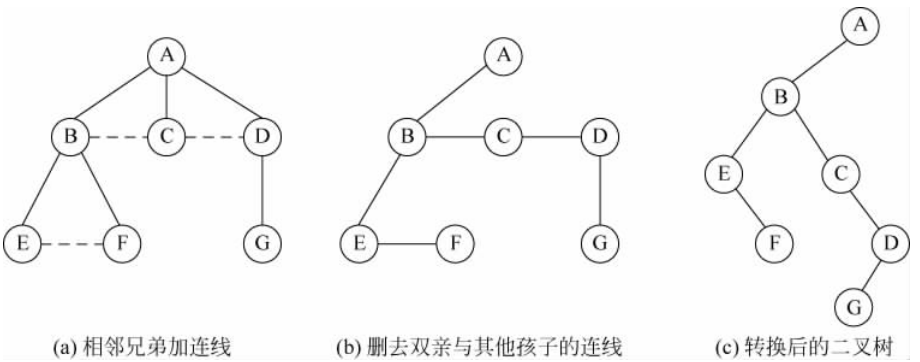


图 5.17　图 5.16 所示的树转换为二叉树的过程示意图

由上面的转换可以看出,在二叉树中,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,所以转换后的二叉树根结点的右孩子必定为空。

实际上,一棵树采用孩子兄弟表示法所建立的存储结构和它所对应的二叉树的二叉链表存储结构是完全相同的。

2. 将森林转换为二叉树

由森林的概念可知,森林是若干棵树的集合,只要将森林中各棵树的根视为兄弟,每棵树又可以用二叉树表示,这样,森林也同样可以用二叉树表示。

将森林转换为二叉树的方法如下:

(1) 将森林中的每棵树转换成相应的二叉树;

(2) 第一棵二叉树不动,从第二棵二叉树开始,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,当所有二叉树连起来以后所得到的二叉树就是由森林转换得到的二叉树。

这一方法可形式化地描述为:

如果 F=$\{T_1, T_2, \cdots, T_m\}$ 是森林,可按以下规则转换成一棵二叉树(B=(Root, LB, RB))。

(1) 若 F 为空,即 $m=0$,则 B 为空树。

(2) 若 F 非空,即 $m\neq 0$,则 B 的根 Root 即为森林中第一棵树的根 Root(T_1);B 的左子树 LB 是从 T_1 中根结点的子树森林 $F_1=\{T_{11}, T_{12}, \cdots, T_{1m1}\}$ 转换而成的二叉树;其右子树 RB 是从森林 $F'=\{T_2, T_3, \cdots, T_m\}$ 转换而成的二叉树。

图 5.18 给出了森林及其转换为二叉树的过程。

3. 将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的根结点无右分支,而森林转换成的二叉树的根结点有右分支。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分支,将一棵二叉树还原为树或森林,具体方法如下:

(1) 若某结点是其双亲的左孩子,则把该结点的右孩子、右孩子的右孩子等都与该结点的

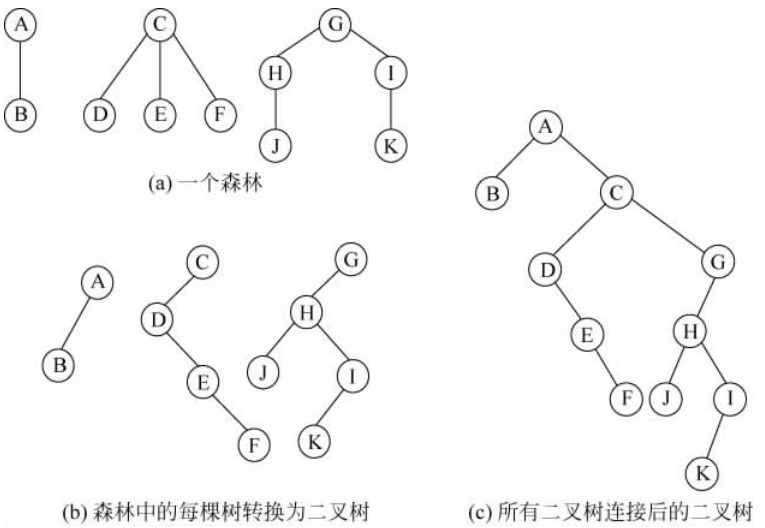


图 5.18　森林及其转换为二叉树的过程示意图

双亲结点用线连起来；

(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) 两步得到的树或森林，使之结构层次分明。

这一方法可形式化地描述为：

如果 B=(Root, LB, RB) 是一棵二叉树，可按以下规则转换成森林(F=$\{T_1, T_2, \cdots, T_m\}$)。

(1) 若 B 为空，则 F 为空。

(2) 若 B 非空，则森林中第一棵树 T_1 的根 Root (T_1) 即为 B 的根 Root；T_1 中根结点的子树森林 F_1 是由 B 的左子树 LB 转换而成的森林；F 中除 T_1 之外其余树组成的森林 $F'=\{T_2, T_3, \cdots, T_m\}$ 是由 B 的右子树 RB 转换而成的森林。

图 5.19 给出了一棵二叉树还原为森林的过程示意图。

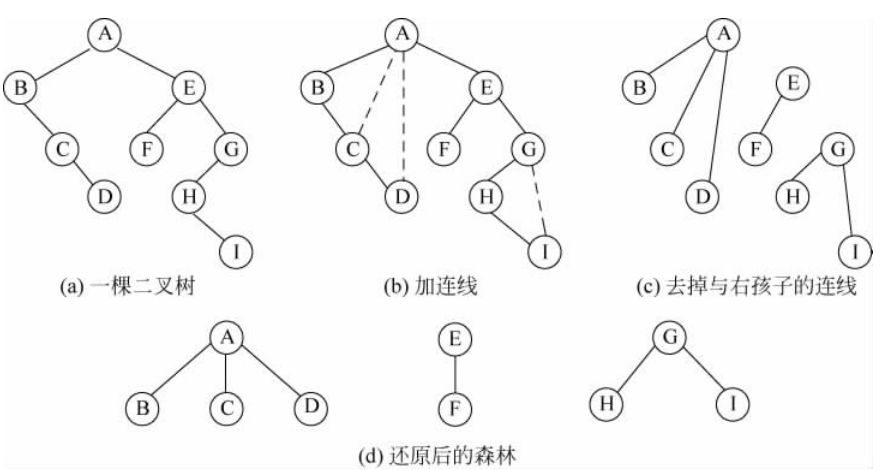


图 5.19　二叉树还原为森林的过程示意图

5.3.3 树和森林的遍历

1. 树的遍历

树的遍历通常有以下两种方式。

1）先根遍历

先根遍历的定义如下：

(1) 访问根结点；

(2) 按照从左到右的顺序先根遍历根结点的每一棵子树。

按照树的先根遍历的定义对图 5.16 所示的树进行先根遍历，得到的结果序列如下：

A B E F C D G

2）后根遍历

后根遍历的定义如下：

(1) 按照从左到右的顺序后根遍历根结点的每一棵子树；

(2) 访问根结点。

按照树的后根遍历的定义对图 5.16 所示的树进行后根遍历，得到的结果序列如下：

E F B C G D A

根据树和二叉树的转换关系以及树和二叉树的遍历定义可以推知，树的先根遍历与其转换的相应二叉树的先序遍历的结果序列相同；树的后根遍历与其转换的相应二叉树的中序遍历的结果序列相同。因此，树的遍历算法是可以采用相应二叉树的遍历算法来实现的。

2. 树的遍历算法实现

在选定树的存储结构后可按上述对应规则实现其遍历算法。例如，以孩子兄弟表示法实现树的先根遍历，算法如下：

算法 5.12 树的遍历算法。

```
void  RootFirst ( NodeType t )                  /* 先根遍历用孩子兄弟表示的树 */
    {  NodeType *p ;
       if ( t!= NULL )
        {  Visit ( t->data ) ;                  /* 访问树的根结点 */
           p = t->firstchild ;                  /* 指向根的第一个孩子结点 */
           while ( p )
              {  RootFirst ( p );               /* 访问孩子结点 */
                 p = p->nextsibling ; /* 指向下一个孩子结点,即当前结点的右兄弟结点 */
              }
        }
    }
```

当然，树的遍历算法也可以直接借助二叉树的遍历算法来实现，即通过将树转换成二叉树后进行。

3. 森林的遍历

森林的遍历有前序遍历和中序遍历两种方式。

1）前序遍历

前序遍历的定义如下：

(1) 访问森林中第一棵树的根结点；

(2) 前序遍历第一棵树的根结点的子树；

(3) 前序遍历去掉第一棵树后的子森林。

对于图 5.16(a)所示的森林进行前序遍历,得到的结果序列如下:

A B C D E F G H J I K

2) 后序遍历

后序遍历的定义如下:

(1) 后序遍历第一棵树的根结点的子树;

(2) 访问森林中第一棵树的根结点;

(3) 后序遍历去掉第一棵树后的子森林。

对于图 5.16(a)所示的森林进行后序遍历,得到的结果序列如下:

B A D E F C J H K I G

根据森林和二叉树的转换关系以及森林和二叉树的遍历定义可以推知,森林的前序遍历和后序遍历与所转换的二叉树的前序遍历和中序遍历的结果序列相同。

5.4 最优二叉树——哈夫曼树

5.4.1 哈夫曼树的基本概念

最优二叉树也称哈夫曼(Huffman)树,是指由一组带有确定权值的叶子结点构造的具有最小带权路径长度的二叉树。所谓的权值是指一个与特定结点相关的数值。

那么什么是二叉树的带权路径长度(Weighted Path Length)呢?

在前面介绍过路径和结点的路径长度的概念,二叉树的路径长度则是指由根结点到所有叶子结点的路径长度之和。如果二叉树中的所有叶子结点都具有一个特定的权值,则可以将这一概念加以推广。设二叉树具有 n 个带权值的叶子结点,那么从根结点到各个叶子结点的路径长度与该叶子结点相应的权值的乘积之和称为二叉树的带权路径长度,记为:

$$\mathrm{WPL} = \sum_{k=1}^{n} w_k \cdot L_k$$

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。对于图 5.20 所示的二叉树,它的带权路径长度值为

$$\mathrm{WPL}=2\times2+4\times2+5\times2+3\times2=28$$

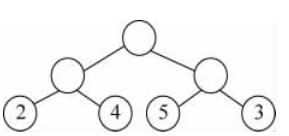
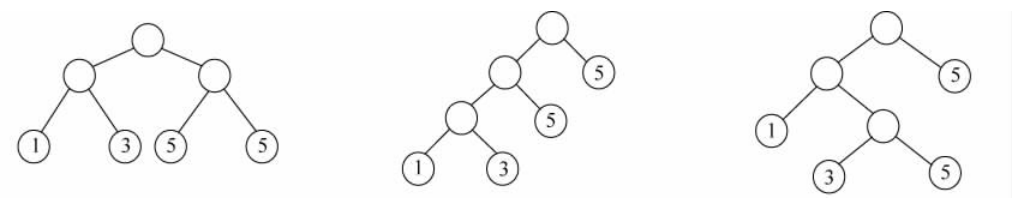


图 5.20 一棵带权二叉树

给定一组具有确定权值的叶子结点可以构造出不同的带权二叉树。例如,给出 4 个叶子结点,设其权值分别为 1、3、5、5,可以构造出形状不同的多个二叉树,这些形状不同的二叉树的带权路径长度各不相同。图 5.21 给出了其中 3 个不同形状的二叉树。



(a) 具有相同叶子结点的二叉树1 (b) 具有相同叶子结点的二叉树2 (c) 具有相同叶子结点的二叉树3

图 5.21 具有相同叶子结点和不同带权路径长度的二叉树

这3棵树的带权路径长度分别如下。

(1) 图5.21(a)：

$$WPL=1\times2+3\times2+5\times2+5\times2=32$$

(2) 图5.21(b)：

$$WPL=1\times3+3\times3+5\times2+5\times1=29$$

(3) 图5.21(c)：

$$WPL=1\times2+3\times3+5\times3+5\times1=33$$

由此可知，由相同权值的一组叶子结点所构成的二叉树有不同的形态和不同的带权路径长度，那么如何找到带权路径长度最小的二叉树(即哈夫曼树)呢？根据哈夫曼树的定义，一棵二叉树要使其WPL值最小，必须使权值较大的叶子结点靠近根结点，使权值较小的叶子结点远离根结点。哈夫曼(Huffman)依据这一特点提出了一种构造最优二叉树的方法，这种方法的基本思路如下：

(1) 由给定的n个权值$\{w_1,w_2,\cdots,w_n\}$构造n棵只有一个叶子结点的二叉树，从而得到一个二叉树的集合$F=\{T_1,T_2,\cdots,T_n\}$。

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

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

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

图5.22给出了前面提到的叶子结点权值集合为$W=\{1,3,5,5\}$的哈夫曼树的构造过程，可以计算出其带权路径长度为27。由此可知，对于同一组给定叶子结点所构造的哈夫曼树，树的形状可能不同，但带权路径长度值是相同的，并且一定是最小的。

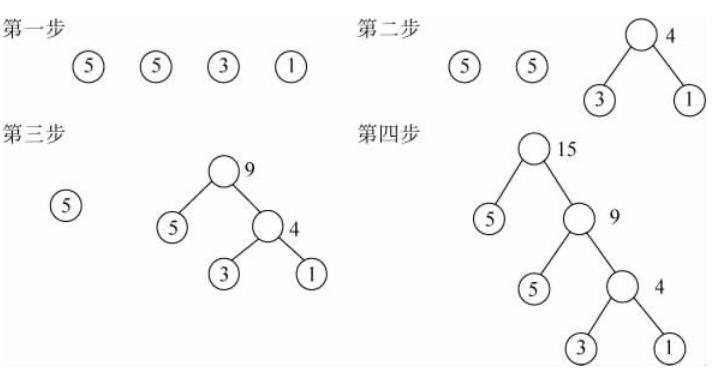


图5.22　哈夫曼树的构造过程

5.4.2　哈夫曼树的构造算法

在构造哈夫曼树时，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下:

weight	lchild	rchild	parent

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经加入要建立的哈夫曼树中,可通过 parent 域的值来确定。初始时 parent 的值为 −1,当结点加入树中时,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。

在构造哈夫曼树时,首先将由 n 个字符形成的 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路不断地将两个小子树合并为一个较大的子树,将每次构成的新子树的根结点顺序放到 HuffNode 数组中前 n 个分量的后面。

算法 5.13 哈夫曼树的构造算法。

```
#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子树 */
       {  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;
             }
         }
      /* 将找出的两棵子树合并为一棵子树 */
      ht [x1].parent = n + i; ht [x2].parent = n + i;
```

```
            ht [n + i].weight = ht [x1].weight + ht [x2].weight;
            ht [n + i].lchild = x1; ht [n + i].rchild = x2;
          }
        }
```

根据以上算法，图 5.22 建立的哈夫曼树的结果如表 5.1 和表 5.2 所示。

表 5.1 哈夫曼树的初态

	weight	parent	lchild	rchild
0	1	−1	−1	−1
1	3	−1	−1	−1
2	5	−1	−1	−1
3	5	−1	−1	−1
4	0	−1	−1	−1
5	0	−1	−1	−1
6	0	−1	−1	−1

表 5.2 哈夫曼树的终态

	weight	parent	lchild	rchild	
0	1	4	−1	−1	
1	3	4	−1	−1	*n* 个叶子结点
2	5	5	−1	−1	
3	5	6	−1	−1	
4	4	5	1	0	
5	9	6	2	4	
6	14	−1	3	5	

5.4.3 哈夫曼编码

在数据通信中，经常需要将传送的文字转换成由二进制字符 0、1 组成的二进制串，即进行符号的二进制编码。常见的 ASCII 编码就是 8 位的二进制编码，此外还有汉字国标码、电报明码等。

ASCII 编码是一种定长编码，即每个字符用相同数目的二进制位表示。为了缩短数据文件(报文)的长度，可采用不定长编码。例如，假设要传送的报文为 ABACCDA，报文中只含有 A、B、C、D 4 种字符，表 5.3 给出了 4 种不同的编码方案。若采用如表 5.3 (a) 所示的编码方案，则报文的代码为 00001000010010011100 0，长度为 21。若采用如表 5.3 (b) 所示的编码方案，则报文的代码为 00010010101100，长度为 14。这两种编码均是定长编码，码长分别为 3 和 2。若采用如表 5.3 (c)所示的编码方案，上述报文的代码为 0110010101110，长度为 13。若采用如表 5.3(d)所示的编码方案，则报文的代码为 01010010010011001，长度为 17。显然不同的编码方案，其最终形成的报文代码的总长度是不同的。那么，要使最终的报文最短，就可以借鉴哈夫曼思想，在编码时考虑字符出现的频率，让出现频率高的字符采用尽可能短的编码，让出现频率低的字符采用稍长的编码，这样构造不定长编码，则报文的代码有可能达到更短。

表 5.3 字符的 4 种不同的编码方案

(a)

字符	编码
A	000
B	010
C	100
D	111

(b)

字符	编码
A	00
B	01
C	10
D	11

(c)

字符	编码
A	0
B	110
C	10
D	111

(d)

字符	编码
A	01
B	010
C	001
D	10

因此,利用哈夫曼树来构造编码方案就是哈夫曼树的典型应用。具体方法如下:

设需要编码的字符集合为$\{d_1,d_2,\cdots,d_n\}$,它们在报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、…、d_n 作为叶子结点,以 w_1、w_2、…、w_n 作为它们的权值,构造一棵哈夫曼树,规定对哈夫曼树中的左分支赋予 0,右分支赋予 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 序列便为该叶子结点对应字符的编码,称为**哈夫曼编码**,这样的哈夫曼树称为哈夫曼编码树。

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

在建立不定长编码时,必须使任何一个字符的编码都不是另一个字符编码的前缀,这样才能保证译码的唯一性。例如表 5.1(d)的编码方案,字符 A 的编码 01 是字符 B 的编码 010 的前缀部分,这样对于代码串 0101001,既是 AAC 的代码,也是 ABA 和 BDA 的代码,因此,这样的编码不能保证译码的唯一性,称为具有二义性的译码。同时把满足“任意一个符号的编码都不是其他符号的编码的前缀”这一条件的编码称为**前缀编码**。

然而,采用哈夫曼树进行编码,不会产生上述二义性问题。因为在哈夫曼树中,每个字符结点都是叶子结点,它们不可能在根结点到其他字符结点的路径上,所以一个字符的哈夫曼编码不可能是另一个字符的哈夫曼编码的前缀,从而保证了译码的非二义性。

设 A、B、C、D 出现的频率分别为 0.4、0.3、0.2、0.1,则得到的哈夫曼树和二进制前缀编码如图 5.23 所示。

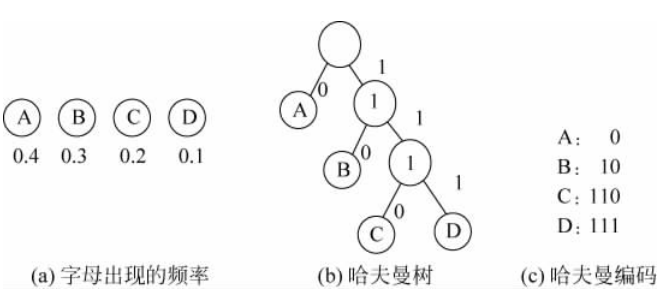


图 5.23 哈夫曼编码构造示例

可以看出,这种不定长的前缀编码能将报文唯一地无二义性地翻译成原文。当原文较长、频率很不均匀时,这种编码可使传送的报文缩短很多。当然,用户也可以在哈夫曼树中规定左分支表示“1”,右分支表示“0”,这样得到的二进制前缀编码虽然不一样,但使用效果一样。

5.4.4 哈夫曼编码的算法实现

前面已经给出了哈夫曼树的构造算法，因此，若哈夫曼编码中需要构造哈夫曼数可以调用前面的构造算法。然而，为了实现哈夫曼编码，需要定义一个编码表的存储结构。其定义如下：

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

哈夫曼编码的算法思路：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，否则为1，直至搜索到根结点为止。

算法 5.14 哈夫曼编码算法。

```
viod  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 ], &cd[ start ] );        /* 将工作区中的编码复制到编码表中 */
     }
    free ( cd );
  }
```

5.5 典型例题

例 5.6 求从二叉树的根结点到 r 结点之间的路径。

设二叉树的存储采用二叉链表方式，T 为根结点指针，r 为指定结点的指针。在利用非递归后序遍历法访问到 r 所指的结点时，可以看到栈中的所有结点均为 r 所指结点的祖先，然后依次输出栈中所有的相关结点，则可得到从 T 到 r 所指结点之间的路径。

算法 5.15 求从二叉树根结点到 r 所指结点之间的路径并依次输出这些结点。

```
void PathInBiTree( BiTree T)
```

```
{/* 非递归后序遍历,求从二叉树链表的根结点到 r 所指结点之间的路径,T 为根结点指针 */
  BiTree s[MAXSIZE];                          /* 辅助栈 */
  int i,top = 0;                              /* top 为栈顶指针,top = 0 表示空栈 */
  BiTree p,q;
  q = NULL;                                   /* q 指向最近被访问过的结点,q 初始为空指针 */
  p = T;
  while(p!= NULL || top!= 0) {
      while(p!= NULL) {                       /* 遍历左子树,使左孩子指针入栈 */
          top++;
          s[top] = p;
          p = p->lchild;
       }
      if( top > 0 ) {
          p = s[top];
          if( p->rchild == NULL || p->rchild == q ) {
                if( p == r) {/* 找到 r 所指的结点,依次显示从根结点到该结点之间的路径 */
                    for (i = 1; i <= top; i++)
                        printf(" %d,",s[i]->data );
                    top = 0 ;                 /* 强制退出循环 */
                }
                else {                        /* 未找到 r 所指的结点,继续查找 */
                    q = p;                    /* q 始终指向最近被访问过的结点 */
                    top-- ;
                    p = NULL;                 /* 跳过前面遍历左子树的循环,继续退栈 */
                }
          }
          else p = p ->rchild;                /* 向右一步 */
      }                                       /* End of if( top > 0 ) */
    }                                         /* End of while(p!= NULL || top!= 0) */
}                                             /* PathInBiTree */
```

例 5.7 中序线索二叉树及其中序遍历。

具有 n 个结点的二叉链表共有 $2n$ 个指针域,但是仅仅使用了 $n-1$ 个,还有 $n+1$ 个为空。线索二叉树就是利用这些空的指针域使二叉树的空的 lchild 域指向其直接前趋结点,使空的 rchild 域指向其直接后继结点。由于二叉树是一种非线性结构,同一个结点在不同的遍历序列中其直接前趋和直接后继可以不同,因此构建一棵线索二叉树必须要与确定的遍历方法结合起来。建立一棵中序线索二叉树就是在中序遍历这棵二叉树的过程中不断修改结点的空指针域使其指向直接前趋或直接后继的过程。

为了区分指针是指向孩子结点还是指向直接前趋或直接后继,在每个结点中增设两个标志域,即 ltag 和 rtag,取值为 0 或 1。ltag=0 表示 lchild 域指向结点的左孩子,ltag=1 表示 lchild 域指向结点的直接前趋;rtag=0 表示 rchild 域指向结点的右孩子,rtag=1 表示 rchild 域指向结点的直接后继。线索二叉树的结点结构的描述如下:

```
typedef struct ThreadNode
{
        elemtype data;
        struct  ThreadNode *lchild, *rchild;/* 左、右孩子指针域 */
        int ltag,rtag ;                     /* 左、右标志域,取值为 0 或 1 */
} ThreadNode, *PThreadBiTree;
```

```
PThreadBiTree pre;                                  /* pre 始终指向刚刚访问过的结点,全局变量 */
```

为了方便算法的实现,在根结点之前为线索二叉链表增加一个头结点(类似于带头结点的单链表中的头结点)。该头结点的 lchild 域指向二叉树的根结点,rchild 域指向所遍历的最后一个结点。

算法 5.16 建立中序线索二叉树及其中序遍历的算法。

(1) 建立中序线索二叉树。

```
void InThread (PThreadBiTree p);              /* 线索化过程 */
PThreadBiTree CreateInThreadBT (PThreadBiTree T);      /* 根据二叉链表 T 建立中序线索二叉树 */

PThreadBiTree CreateInThreadBT (PThreadBiTree T)
{ /* 中序线索化, T 指向二叉树的根结点 */
  PThreadBiTree thrt;                         /* thrt 指向头结点 */
  thrt = ( PThreadBiTree ) malloc(sizeof(ThreadNode )); /* 生成头结点 */
  thrt -> rtag = 1; thrt -> rchild = thrt;    /* 头结点的右指针回指 */
 thrt -> ltag = 0;
 if( T == NULL) thrt -> lchild = thrt;        /* 若二叉树为空,则头结点的左指针也回指 */
 else
 {
    thrt -> lchild = T;                       /* 指向二叉树的根结点 */
    pre = thrt;                               /* pre 指向头结点 */
    InTread(T);                               /* 中序线索化二叉树 T */
    pre -> rchild = thrt;                     /* 中序遍历最后一个结点的右指针域指向头结点 */
    pre -> rtag = 1 ;
    thrt -> rchild = pre;                     /* 头结点的右指针域指向中序遍历的最后一个结点 */
   }
   return thrt;                               /* 返回头结点的指针 */
} /* InOrderThreadBiTree */

void InThread (PThreadBiTree T)
{ /* 中序线索化二叉树的过程 */
  if( T!= NULL)
  {
      InTread(T -> lchild);                   /* 左子树线索化 */
      if ( T -> lchild == NULL )
      {/* 左指针域指向其直接前趋 */
          T -> ltag = 1 ;
          T -> lchild = pre;
      }
      else T -> ltag = 0 ;
      if (pre -> rchild == NULL )
      { /* 直接前趋(即 pre 所指的结点)的右指针域指向该结点 T */
          pre -> rtag = 1 ;
          pre -> rchild = T;
      }
      else pre -> rtag = 0;
      pre = T;                                /* pre 跟上,使 pre 始终指向刚刚访问过的结点 */
```

```
        InTread(T -> rchild);                    /* 右子树线索化 */
    }
} /* InThread */
```

(2) 遍历中序线索二叉树。

```
void ThreadInOrder(PThreadBiTree thrt)
{/* 中序线索二叉树的遍历,thrt 指向头结点 */
    PThreadBiTree p = thrt -> lchild;            /* p 的初值指向根结点 */
    while(p!= thrt)
    {
        while(p -> ltag == 0) p = p -> lchild;
        visit( p -> data);                       /* 访问结点 */
        while(p -> rtag == 1&& p -> rchild!= thrt)
        { /* 根据右线索访问直接后继 */
            p = p -> rchild;
            visit( p -> data);
        }
        p = p -> rchild;
    }/* while */
} /* ThreadInOrder */
```

本 章 小 结

本章主要介绍了树、森林与二叉树的定义、性质、操作和相关算法的实现,特别是二叉树的遍历算法,它们是许多二叉树应用的算法设计的基础,读者必须熟练掌握。对于树的遍历算法,由于树的先根遍历次序与对应二叉树表示的前序遍历次序一致,树的后根遍历次序与对应二叉树的中序遍历次序一致,因此可以得出树的遍历算法。

本章最后讨论的哈夫曼树是一种扩充的二叉树,即在终端结点上带有相应的权值,并使其带权路径长度最短。作为哈夫曼树的应用,引入了哈夫曼编码。通常让哈夫曼树的左分支代表编码“0”,右分支代表编码“1”,得到哈夫曼编码。这是一种不定长编码,可以有效地实现数据压缩。

本章的要点如下:

(1) 理解树、二叉树、森林的定义,尤其要清楚树与二叉树本质上的不同;掌握二叉树的性质,二叉树的顺序存储和链表存储表示,以及树的各种存储结构。同时要掌握树、森林与二叉树的相互转换方法。

(2) 熟练掌握二叉树的各种遍历算法,这是本章的重点,原因有三:一是通过遍历得到了二叉树中结点的线性序列,实现非线性结构的线性化;二是几乎所有的对二叉树的操作都可以通过遍历算法扩充来实现,遍历算法是基础;三是二叉树遍历算法的递归实现是程序设计中的重要技术,对理解递归含义,使用递归控制条件都非常重要。当然,在理解遍历应用时要注意分析应用问题对遍历顺序的要求,此外,还要掌握从二叉树遍历结果得到二叉树的方法。

(3) 掌握哈夫曼树的概念,理解哈夫曼树的构造过程与实现算法,以及解决数据压缩问题的哈夫曼编码方法。

习 题 5

5.1 单选题。

(1) 树最适合用来表示_____。

A. 有序数据元素　　B. 无序数据元素

C. 元素间具有层次关系的数据　　D. 元素间无联系的数据

(2) 在 m 叉树中,度为 0 的结点称为_____。

A. 兄弟　　B. 树叶　　C. 树根　　D. 分支结点

(3) 如果树的结点 A 有 4 个兄弟,而且 B 为 A 的双亲,则 B 的度为_____。

A. 3　　B. 4　　C. 5　　D. 1

(4) 根据二叉树的定义可知二叉树共有_____种不同的形态。

A. 4　　B. 5　　C. 6　　D. 7

(5) 由 3 个结点可以构造出_____种不同形态的二叉树。

A. 3　　B. 4　　C. 5　　D. 6

(6) 具有 20 个结点的二叉树,其深度最多为_____。

A. 4　　B. 5　　C. 6　　D. 20

(7) 高度为 h 的满二叉树的结点数是_____个。

A. $\log_2 h+1$　　B. 2^h+1　　C. 2^h-1　　D. 2^{h-1}

(8) 深度为 5 的二叉树最多有_____个结点。

A. 16　　B. 32　　C. 31　　D. 10

(9) 设一棵二叉树共有 50 个叶子结点(终端结点),则共有_____个度为 2 的结点。

A. 25　　B. 49　　C. 50　　D. 51

(10) 一颗完全二叉树中根结点的编号为 1,并且 23 号结点有左孩子但没有右孩子,则完全二叉树总共有_____个结点。

A. 24　　B. 45　　C. 46　　D. 47

(11) 二叉树的第 3 层最少有_____个结点。

A. 0　　B. 1　　C. 2　　D. 3

(12) 设 n、m 为一棵二叉树上的两个结点,在中序遍历时,n 在 m 之前的条件是_____。

A. n 在 m 右方　　B. n 是 m 祖先　　C. n 在 m 左方　　D. n 是 m 子孙

(13) 某二叉树的先序序列和后序序列正好相同,则该二叉树可能是_____的二叉树。

A. 高度大于 1 的左单支　　B. 高度大于 1 的右单支

C. 最多只有一个结点　　D. 既有左孩子又有右孩子

(14) 某二叉树的中序序列和后序序列正好相反,则该二叉树一定是_____的二叉树。

A. 空或只有一个结点　　B. 高度等于其结点数

C. 任一结点无左孩子　　D. 任一结点无右孩子

(15) 有 n 个结点的二叉树链表共有_____个空指针域。

A. $n-1$　　B. n　　C. $n+1$　　D. $n+2$

(16) 一个有 n 个叶结点的哈夫曼树具有的结点数为_____。

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

5.2　填空题。

(1) 一颗深度为5的二叉树,最少有_____个叶子结点。

(2) 一棵完全二叉树采用顺序存储结构,每个结点占4个字节,设编号为5的元素的地址为1016,且它有左孩子和右孩子,则该左孩子和右孩子的地址分别为_____和_____。

(3) 一棵完全二叉树采用顺序存储结构,若编号为 i 的元素有左孩子,则该左孩子的编号为_____。

(4) 一棵含有 $n(n>1)$ 个结点的 k 叉树,当 $k=$_____时深度最大,此最大深度为_____;当 $k=$_____时深度最小,此最小深度为_____。

(5) 深度为 k 的完全二叉树最少有_____个结点,最多有_____个结点。

(6) 已知一棵二叉树的先序遍历序列为EBADCFHGIKJ,中序遍历序列为ABCDEFGFIJK,则该二叉树的后序遍历序列为_____。

(7) 如果指针 p 指向一棵二叉树的一个结点,则判断 p 没有左孩子的逻辑表达式为_____。

(8) 在由 n 个带权叶子结点构造出的所有二叉树中,带权路径长度最小的二叉树称为_____。

(9) 在树的孩子兄弟表示法中,每个结点有两个指针域,一个指向_____,另一个指向_____ 。

(10) 树的先根遍历结果与其转换的相应二叉树的_____结果相同;树的后根遍历结果与其转换的相应二叉树的_____结果相同。

5.3　写出图5.24中树的叶子结点、非终端结点、各结点的度和树深。

5.4　分别画出含3个结点的无序树与二叉树的所有不同形态。

5.5　分别画出图5.25中所示二叉树的二叉链表、三叉链表的顺序存储结构示意图。

5.6　分别写出图5.25中所示二叉树的先序遍历、中序遍历、后序遍历的结点访问序列。

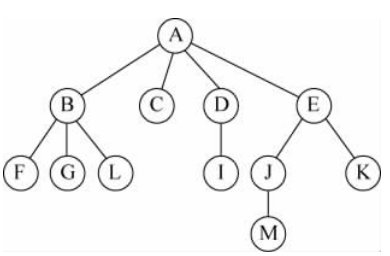


图5.24　题5.3图

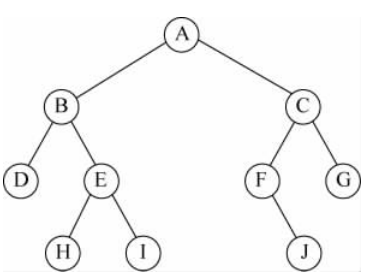


图5.25　题5.5图

5.7　试找出分别满足下列条件的所有二叉树:

(1) 先序序列和中序序列相同;

(2) 后序序列和中序序列相同;

(3) 先序序列和后序序列相同。

5.8　已知一棵二叉树的中序序列和后序序列分别为BDCEAFHG和DECBHGFA,试画出这棵二叉树。

5.9　分别写出图 5.24 中所示树的先根遍历、后根遍历和层次遍历的结点访问序列。

5.10　如果一棵树有 n_1 个度为 1 的结点，n_2 个度为 2 的结点……，n_m 个度为 m 的结点，则该树共有多少个叶子结点？

5.11　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，试求该树含有的叶子结点的数目。

5.12　编一算法求二叉树中结点的总数。

5.13　编一算法将二叉树中所有结点的左、右子树相互交换。

5.14　编一算法判别给定的二叉树是否是完全二叉树。

5.15　将图 5.26 中所示的森林转换为二叉树。

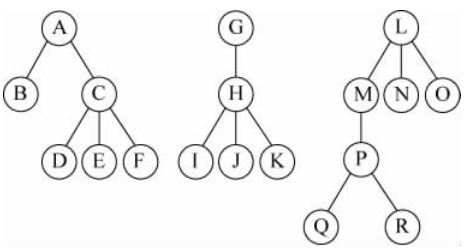


图 5.26　题 5.15 图

5.16　分别画出图 5.27 中所示二叉树对应的森林。

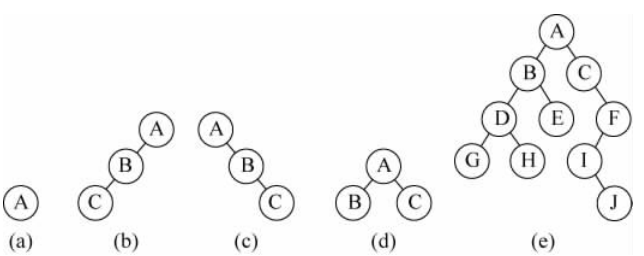


图 5.27　题 5.16 图

5.17　在以双亲链表表示法存储结构的树中，写出以下算法：

(1) 求树中结点双亲的算法；

(2) 求树中结点孩子的算法。

5.18　在以孩子链表表示法存储结构的树中，写出以下算法：

(1) 求树中结点双亲的算法；

(2) 求树中结点孩子的算法。

5.19　在以孩子兄弟链表结构存储的树中，写出求树中结点孩子的算法。

5.20　(1) 给定权值(4,3,16,9,22,10,5)，构造相应的哈夫曼树。

(2) 设上述权值分别代表 7 个字母出现的频率，试为这 7 个字母设计哈夫曼编码。