

## 第 6 章 树形结构

树形结构是元素结点之间有分支和层次关系的结构，它类似于自然界的树。树形结构是一种非线性结构，在客观世界中，有许多事物本身就呈现树形结构，如家族关系，部门机构设置等。另一方面，一些解决实际问题的算法，也常常借助树结构解决。本章介绍树结构有关的基本内容并列举一些简单应用实例。关于它的进一步应用，在后面几章中还要介绍。

### § 6.1 树结构的基本概念

#### § 6.1.1 树结构的定义

##### (一) 非递归定义

**树结构** (Tree Structures) 是二元组  $(D, R)$ ，其中， $D$  是  $n$  个数据元素的有穷集合 ( $n \geq 0$ ) (数据元素称为结点)， $R$  是  $D$  上的一个关系。 $n=0$  时，称为空树； $n>0$  时，它满足以下条件：

- 有且仅有一个结点  $d_0 \in D$ ，满足：不存在任何  $d \in D$ ，使  $\langle d, d_0 \rangle \in R$ 。我们称它为树的**根(Root)**。
- 除根结点  $d_0$  外， $D$  上每个结点  $d$  (若有的话)，总存在一个唯一的结点  $d' \in D$ ， $d \neq d'$ ，使得  $\langle d, d' \rangle \in R$ 。

**例 6-1** 设有数据结构  $T=(D, R)$ ，其中，

$D=\{a,b,c,d,e,f,g\}$

$R=\{r\}$

$r=\{\langle a,b \rangle, \langle a,c \rangle, \langle a,d \rangle, \langle c,e \rangle, \langle c,f \rangle, \langle f,g \rangle\}$

则容易验证，该结构是树形结构， $a$  为根。其图形表示如

图 6-0。

从上面定义可知，树结构中有一个特殊结点，称为根，它的特殊点在于无前趋。其它结点有且仅有一个前趋。每个结点可以有多个后继，但必有一些无后继的结点 (否则  $D$  为

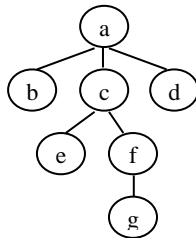


图 6-1 一棵树

无限集合或有结点的前趋不唯一)。

下面根据上面的定义给出其他几个概念。

**路径、通路：**如果存在一个结点序列  $d_{i_1}, d_{i_2}, \dots, d_{i_k}$ ，使得  $\langle d_{i_{j-1}}, d_{i_j} \rangle \in R, j = 2, \dots, k$ ,

则称其为从  $d_{i_1}$  到  $d_{i_k}$  的一条**路径或通路(Path)**。也称从  $d_{i_1}$  到  $d_{i_k}$  有**通路**。通路记为

$$(d_{i_1}, d_{i_2}, \dots, d_{i_k})$$


其中的结点个数称为**通路长**。

有的文献将通路长定义为通路中的边的个数。

例如，图 6-0 所示图中，下列结点序列就是几条通路：

(a, b), (a, c, e), (a, b, f), (c, f, g), ...

**子树：**若对上面的树  $(D, R)$ ，有  $D' \subseteq D, R' \subseteq R$  (且  $R'$  中的各成员也都是  $R$  的成员的子集)， $R'$  是  $D'$  上的关系，则如果  $\langle D', R' \rangle$  满足上面的树的定义，且不存在任何  $d \in D$  和  $d' \in D'$ ，使  $\langle d', d \rangle \in R$ ，则称其为  $\langle D, R \rangle$  的**子树**。若  $\langle D', R' \rangle$  的根为  $x$ ，且  $\langle d, x \rangle \in R, d \in D$ ，则称  $\langle D', R' \rangle$  为  $d$  的**子树**。若对某结点  $d$ ，不存  $x \in D$  使在  $\langle d, x \rangle \in R$ ，则称  $d$  的子树为空 (空子树)。

从这个定义看出，若某子树的根是  $x$ ，则该子树包括从  $x$  出发的所有通路。条件“且不存在任何  $d \in D$  和  $d' \in D'$ ，使  $\langle d', d \rangle \in R$ ”的目的是限制通路没有通到“底”的情况，即该条件指出，对子树中每个结点，该子树也包含了它的所有后继。

例如，图 6-0 所示图中，下列二元组都是子树：

a 的子树：  $(D_1, R_1), D_1 = \{b\}, R_1 = \{\}$ ，根为 b；

a 的子树：  $(D_2, R_2), D_2 = \{c, e, f, g\}, R_2 = \{\langle c, e \rangle, \langle c, f \rangle, \langle f, g \rangle\}$ ，根为 c

a 的子树：  $(D_3, R_3), D_3 = \{d\}, R_3 = \{\}$ ，根为 d

c 的子树：  $(D_4, R_4), D_4 = \{e\}, R_4 = \{\}$ ，根为 e

c 的子树：  $(D_5, R_5), D_5 = \{f, g\}, R_5 = \{\langle f, g \rangle\}$ ，根为 f

f 的子树：  $(D_6, R_6), D_6 = \{g\}, R_6 = \{\}$ ，根为 g

b, d, e, g 无子树，或说它们的子树为空。

**n 叉树：**若各结点的后继个数最大为  $n$ ，则称该树为  $n$  叉树。例如，图 6-0 给出的树是个 3 叉树。

**有序树 (Sequential Tree)：**若二元组  $(D, R)$  中的关系集合  $R$  中含有  $n$  个关系集合 ( $n$  为各结点的最大后继数目)，即  $R = \{r_1, r_2, \dots, r_n\}$ ，且每个关系集合  $r_i$  都不与其他关系集合相交，则称其为有序树。

有序树实质上是后继有序的树，即每个结点的  $n$  个后继次序相关，不同的次序排列，属于不同的结构。

若一棵树不是有序的，则称为**无序树**。

在树的定义中，并没有对二元组 $(D, R)$ 中的关系集合 $R$ 进行其他规定。事实上， $R$ 中可包含一个关系，这种情况下， $(D, R)$ 中只有一种关系，对应于树，表示不区分树的各个子树（无序树）。 $R$ 中包含多个关系时表示有序树。

例如，图 6-0 给出的树是个 3 叉树，但  $R$  中只含一个集合  $r$ ，所以是无序树。若我们重新定义  $R$  (这里是重组合  $R$ ):

$$R = \{r_1, r_2, r_3\}$$

$$r_1 = \{ \langle a, b \rangle, \langle c, e \rangle \}$$

$$r_2 = \{ \langle a, c \rangle, \langle c, f \rangle, \langle f, g \rangle \}$$

$$r_3 = \{ \langle a, d \rangle \}$$

则  $\{D, R\}$  为三叉有序树， $r_1, r_2, r_3$  分别表示第一、第二、第三叉。

简言之，树满足“单前趋，多后继，根连通”。

由于这里定义的树是指定根的，故我们在强调这点时称这种树为**有根树**。

## (二) 树的递归定义

树也可以递归地定义为：**树**是具有  $n$  个结点的有限集合  $T$  (这里  $n \geq 0$ )，若  $n=0$  则称为空树，否则，它满足：

- 有一个特殊结点  $d_0$ ，我们称它为根。
- 若  $T - \{d_0\}$  非空，则  $T - \{d_0\}$  可分成  $m$  个 ( $m > 0$ ) 不相交的集合  $T_1, T_2, \dots, T_m$ ，而且这些集合中的每一个又都是满足本定义的树，称作  $T$  的**子树**。若  $T - \{d_0\}$  为空，称  $T$  无子树（或子树为空）。

从该定义知，只有一个元素的集合是一棵树，该元素就是根，它无子树。如果集合中元素个数大于 1，则它至少含一棵子树。

**例 6-2** 设有集合  $T = \{a, b, c, d, e, f, g\}$ ，且

$T - \{a\}$  可分成下列子集：

$$T_1 = \{b\}$$

$$T_2 = \{c, e, f, g\}$$

$$T_3 = \{d\}$$

$T_1 - \{b\}$  为空集，满足树的定义。

$T_2 - \{c\}$  可分为下列集合：

$$T_{21} = \{e\}$$

$$T_{22} = \{f, g\}$$

$T_{21} - \{e\}$  为空集，满足树的定义；

$T_{22} - \{f\}$  可分为下列集合：

$$T_{221} = \{g\}$$

$T_{221} - \{g\}$  为空集，满足树的定义。

因此，由树的定义知，集合  $T$  是树。容易看出，这里的树  $T$  就是图 6-0 中的树。

可以证明，前面给出的两种定义是等价的。

此定义是用子集关系表达结点间的前趋/后继关系。在上面的定义中， $T$  的子集  $T_1, T_2, \dots, T_m$  的根都是  $T$  的根的后继。

同样，也可以给出路径的定义：

若  $T_1, T_2, \dots, T_k$  是子树，它们的根分别是  $a_1, a_2, \dots, a_k$ ， $T_2$  是  $T_1$  的子树， $T_3$  是  $T_2$  的子树， $\dots$ ， $T_k$  是  $T_{k-1}$  的子树，则称  $a_1$  到  $a_k$  有**路径（通路）**，记为：

$$\langle a_1, a_2, \dots, a_k \rangle$$

## § 6.1.2 基本术语

这里给出一些未在定义中给出的但很常用的术语。所给定义以树的非递归定义为准。

- 树枝（分枝）(Branch)：结点之间的二元关系（序偶）。
- 入度(In Degree)：x 的前驱的个数称为 x 的入度。显然，在树中，除根的入度为 0 外，其他结点的入度均为 1。
- 度(Degree)/出度(Out Degree)：结点拥有的子树（后继）的个数称为该结点的度，也称为出度。
- 叶子(Leaf)：无后继（子树）的结点称为叶子。
- 分枝结点(Branch Nodes)：有后继的结点称为分枝结点。
- 儿子(Sons)：结点 x 的子树的根称为 x 的儿子。
- 双亲（父亲）(Parents)：结点 x 的前趋结点称为 x 的双亲。
- 祖先(Anccestor)：结点 x 的父亲称为该结点的祖先；结点 x 的父亲的祖先也称 x 的祖先。
- 后代(Descendants)：结点 x 的儿子称 x 的后代，结点 x 的儿子的后代也称 x 的后代。
- 兄弟(Brothers)：同父亲的结点称为兄弟结点。
- 堂兄弟(Cousin)：父亲是兄弟关系或堂兄弟关系的结点称为堂兄弟结点。
- 层次（层号）(Level)：根为第 1 层，对任何其它结点，若它父亲为第 k 层结点，则它为第(k+1)层结点(层号为 k+1)。
- 深度(depth)：树中的具有最大层号的结点的层号，称为树的深度/高度。
- 结点按层编号（层序编号）：将树中结点按从上（层）到下（层）、同层从左到右的次序排成一个线性序列，依次给它们编以连续的自然数。
- 祖辈（上层）(Forerunner)：层号比结点 x 小的结点，称为 x 的祖辈（上层）
- 后辈（下层）(Latecomes)：层号比结点 x 大的结点，称为 x 的后辈（下层）
- 森林(Forest)：相互不相交的树的集合称为森林。

• 同构(Isomorphism): 若对两棵树, 通过对结点适当地重命, 就可以使两棵树完全相等 (结点对应相等, 对应结点的相关关系也对应相等), 则称这两棵树同构。

**例 6-3** 具有 3 个结点的不同构的有序树共 5 种, 如图 6-0。

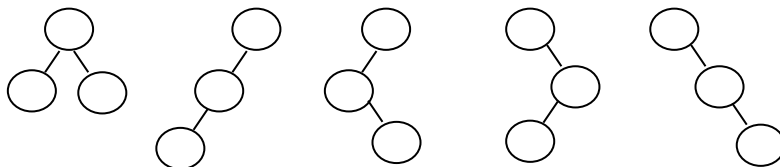


图 6-2 结点总数为 3 的不同构的有序树

## § 6.2 二叉树的概念

二叉树是一种特殊的树, 特别适合于计算机处理。后面将会看到, 任何树都可以转化为二叉树进行处理。所以二叉树是我们研究的重点。

### § 6.2.1 二叉树的基本概念

若有序树中每个结点的子树数目不超过 2, 则称该树为二叉树。二叉树是树的特例, 空集仍称为空 (二叉) 树。

在树的非递归定义中限制结点的后继个数不超过 2, 或在树的递归定义中将  $m$  的上限定为 2 即可得到二叉树的严格定义。

在二叉树中, 通常分别称第 1 与第 2 子树为左子树和右子树。

### § 6.2.2 几种特殊二叉树

下面介绍几种特殊结构的二叉树, 它们在实际中有许多应用。这几种特殊二叉树是逻辑结构上的特殊二叉树, 在后面的章节中, 我们还将介绍几种语义上的特殊二叉树。

#### (一) 满二叉树.

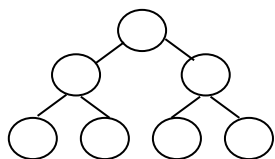
只含度为 0 与度为 2 的结点, 且度为 0 的结点只出现在最后一层的二叉树称为满二叉树。

显然, 满二叉树中除最后一层外, 其他各层上结点都有 2 个结点。空树和只含一个结点的树也是满二叉树。实例见图 6-0 (a)。

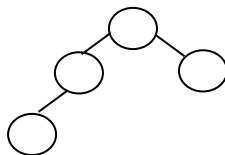
(二) 顺序二叉树 (注：其他很多书中都称其为完全二叉树，为防误解，是否统一？？？  
 --答：完全二叉树在这里另有所指，不能再用。我们认为其他书的那种叫法不妥，没有区分顺序二叉树和完全二叉树)

对任一棵满二叉树，从它的最后一层的最右结点起，按从下到上、从右到左的次序，去掉若干个结点后所成的树称为**顺序二叉树**。

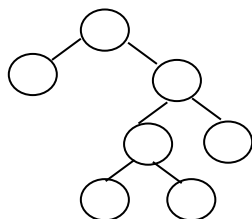
显然，顺序二叉树中可含度为 1 的结点，但它只出现在倒数第 2 层上的最右边。空树和只含一个结点的树也是顺序二叉树。实例见图 6-0 (b)。



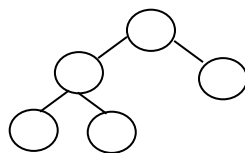
(a) 满二叉树



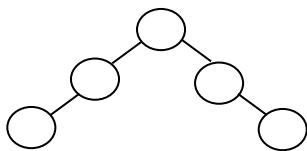
(b) 顺序二叉树



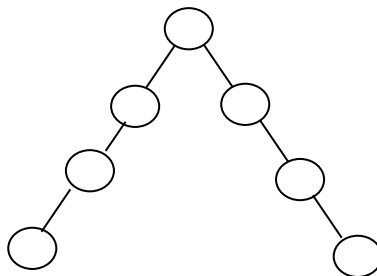
(c) 完全二叉树



(d) 顺序、完全二叉树



(e) 平衡二叉树



(f) 一般二叉树(非平衡)

图 6-3 特殊二叉树示例

(三) 完全二叉树

不含度为 1 的结点的二叉树称**完全二叉树**。

显然，与满二叉树的不同是，完全二叉树的度为 0 的结点可出现在任一层上。实例见


图 6-0 (c)(d)。

显然，空树和只含一个结点的树也是完全二叉树。

#### (四) 高度平衡二叉树

若树中任一结点的任两个子树的高度之差不超过 1，则称这种树为**高度平衡树**，特别是对二叉树，则为**高度平衡二叉树**。

显然，空树和只含一个结点的树也是平衡树二叉树。实例见图 6-0 (e)

 有的教材将我们这里定义的顺序二叉树叫作完全二叉树，而对我们这里的完全二叉树，没有明确的叫法。

显然，有下面的结论：

- 满二叉树、顺序二叉树均为高度平衡二叉树。
- 满二叉树是顺序二叉树。
- 满二叉树是完全二叉树。

因此，对顺序二叉树或完全二叉树或平衡二叉树成立的结论（定理），对满二叉树也成立。对高度平衡二叉树成立的结论（定理），对顺序二叉树也成立。

### § 6.2.3 二叉树的基本性质

为了更好地了解二叉树结构，我们这里介绍二叉树的一些与数据结构有关的重要性质。

**定理 6-1：**满二叉树第  $i$  层上恰好有  $2^{i-1}$  个结点( $i \geq 1$ )。

证：使用归纳法。 $i=1$  时，结论显然成立。设  $i=k$  时结论成立，则考虑  $i=k+1$  的情形。由于  $(k+1)$  层上结点是  $k$  层上结点的儿子，而且满二叉树每个非叶子结点恰好有两个儿子，故  $(k+1)$  层上结点数为  $k$  层上结点个数的 2 倍，即

$$2 \cdot 2^{k-1} = 2^k = 2^{(k+1)-1}.$$

这表明， $i=k+1$  时结论也成立。由归纳法原理，结论对任意的  $k$  都成立，证毕。

**定理 6-2：**二叉树的第  $i$  层上结点个数不超过  $2^{i-1}$  ( $i \geq 1$ )。

事实上，这是**定理 6-1**的直接推论，因为任何二叉树，只有满二叉树的结点最多。

**定理 6-3：**深度为  $k$  的满二叉树有  $2^k-1$  个结点。

证：结点总数 =  $\sum_{i=1}^k$  (第  $i$  层上结点总数)

$$\begin{aligned}
 &= \sum_{i=1}^k 2^{i-1} \quad (\text{定理 6-1}) \\
 &= \frac{2^{1-1}(2^k - 1)}{2 - 1} \\
 &= 2^k - 1.
 \end{aligned}$$

证毕。

**定理 6-4:** 深度为  $k$  的二叉树，至多有  $(2^k - 1)$  个结点。

此乃定理 6-3 的直接推论。

**定理 6-5:** 对任一棵二叉树，有

$$\begin{aligned}
 n_0 &= n_2 + 1 \\
 n &= 2n_0 + n_1 - 1
 \end{aligned}$$

这里， $n_0$ 、 $n_1$  和  $n_2$  分别为度为 0、1 和 2 的结点的数目， $n$  表示结点总数。

证：显然，

$$n = n_0 + n_1 + n_2$$

另一方面，由于二叉树除根外每个结点恰好有一个前趋，所以，非根结点恰与分枝一一对应，故有

$$n = B + 1$$

这里， $B$  为分枝数目。又分枝是由度为 1 和 2 的结点射出的，故有

$$B = n_1 + 2n_2$$

结合上面三式，即可导出  $n_0 = n_2 + 1$  与  $n = 2n_0 + n_1 - 1$

**定理 6-6:** 具有  $n$  个结点的顺序二叉树的深度为  $\lceil \log_2 n \rceil + 1$ （这里，符号  $\lceil x \rceil$  表示不大于  $x$  的最大整数）。

证：设  $k$  为顺序二叉树的深度，由定理 6-4 知

$$n \leq 2^k - 1$$

由于这里  $n$  与  $2^k$  均为整数，故  $n < 2^k$ ，从而

$$k > \log_2 n \quad \dots\dots\dots(a)$$

另一方面，由于是顺序二叉树，去掉最后一层后必为满二叉树，故  $(k-1)$  层以上结点总数为  $2^{k-1} - 1$ ，因此有  $n > 2^{k-1} - 1$ 。由于  $n$  与  $2^{k-1}$  均为整数，为  $2^{k-1} - 1$  加一后，有可能与  $n$  相等，但不会变得大于  $n$ ，故  $n \geq 2^{k-1}$ ，即

$$k \leq \log_2 n + 1 \quad \dots\dots\dots(b)$$

现在，我们已得到 (a) 与 (b) 两式，即

$$\log_2 n < k \leq \log_2 n + 1$$



由于  $k$  为整数, 故必有  $k = [\log_2 n] + 1$  (见图 6-0)

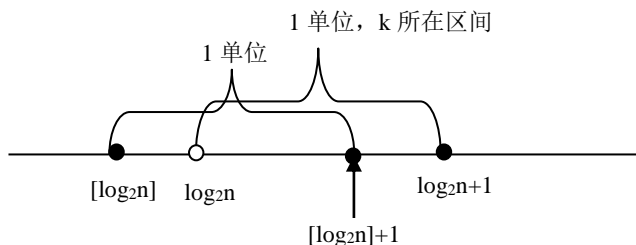


图 6-4  $\log_2 n < k \leq \log_2 n + 1$  示意图

**定理 6-7:** 若对一棵顺序二叉树的结点按层序编号 (即从根所在层起, 依次从层号较小的层到层号较大的层、同层从左到右, 给各结点依次编以连续的号码), 则对任一结点  $i$  ( $1 \leq i \leq n$ ,  $n$  为结点数目, 1 为根的编号), 有

- (a) 若  $i=1$ , 则结点  $i$  是根, 无父亲, 若  $i>1$ , 则其父亲的编号为  $[i/2]$ ;
- (b) 若  $2i > n$ , 则结点  $i$  无左儿子 (从而也无右儿, 为叶子), 否则  $i$  的左儿子的编号为  $2i$ 。
- (c) 若  $2i+1 > n$ , 则结点  $i$  无右孩子, 否则它的右孩子结点的编号为  $2i+1$ 。

证: 先证(b)和(c), 用归纳法。当  $i=1$  时, 结论是显然的。现设  $i=k$  时结论(b)成立, 考虑  $i=k+1$  的情形。若  $k$  结点无左儿子或无右儿子, 则  $(k+1)$  亦必为叶子 (否则就不是顺序二叉树了), 故证明(b)时无需考虑此情况, 而只需考虑  $k$  有两个儿子的情况。先考虑  $k$  与  $k+1$  在同一层上, 则由顺序二叉树的结构知, 若  $(k+1)$  有儿子, 则必然出现在下一层上  $k$  的两个儿子的紧右边, 由于  $k$  的两个儿子的编号为  $2k$  与  $2k+1$  (归纳假设), 故  $(k+1)$  若有左儿子, 则编号为  $(2k+1)+1$  即  $2(k+1)$ , 这表示  $i=k+1$  时结论成立; 若  $k+1$  有右儿子 (当然也有左儿子), 则编号为  $(2k+1)+2$ , 即  $2(k+1)+1$ , 这表明(iii)在  $k+1$  时仍成立。再考虑  $k$  与  $k+1$  不在同一层的情况, 此时,  $k$  必在它所在层的最右, 而  $k+1$  必在下一层的最左, 由于顺序二叉树编号是编完上层最右结点时从下层最左结点起编号, 所以若  $k+1$  有儿子, 则编号必然为  $(2i+1)+1$  与  $(2i+1)+2$ , 这与  $k$  与  $k+1$  位于同层的情况相同。

至于(a), 则可由(b)与(c)的式子变换而来, 证毕。

#### § 6.2.4 二叉树的遍历的概念

##### (一) 基本概念

**遍历**就是无重复无遗漏的访问。对于树的遍历, 是指按一定方式访问树中的结点, 使得每个结点恰好被访问一次。

这里，**访问**是指读、写、修改或其它种类的加工处理。不失一般性，我们在此将访问看作是输出。访问方式是指访问次序。我们假定执行访问的机构是单处理机的，任何时刻只能对一个目标进行访问，所以，访问的轨迹是被访问结点的线性序列，即遍历的结果是树中各结点按某种次序的一个线性序列。通过遍历，非线性结构被映射成了线性结构。

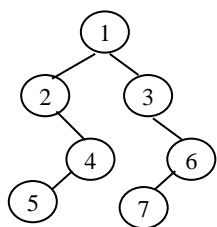
二叉树的遍历方式有前序遍历、中序遍历、后序遍历与层序遍历四种，现分述如下。

## (二) 前序遍历

二叉树前序遍历定义为：

若树为空，则不作任何动作，返回，否则，先访问根结点，然后按前序方式分别遍历根的左子树和右子树。

简言之，前序遍历是按“根—左子树—右子树”的次序遍历二叉树。显然，这是个递归定义，下面的中序遍历和后序遍历也是按递归方式定义的。前序遍历实例见图 6-0。



前序遍历结果：1 2 4 5 3 6 7

中序遍历结果：2 5 4 1 3 7 6

后序遍历结果：5 4 2 7 6 3 1

层序遍历结果：1 2 3 4 6 5 7

图 6-5 二叉树遍历示例

## (三) 中序遍历

二叉树中序遍历定义为：

若树为空，则不作任何动作，返回，否则，先按中序方式遍历根的左子树，然后访问根，最后再按中序方式遍历根的右子树。

简言之，中序遍历是按“左子树—根—右子树”的次序遍历树。其实例见图 6-0。

## (四) 后序遍历

二叉树后序遍历定义为：

若树为空，则不作任何动作，返回，否则，先分别按后序遍历方式遍历根的左子树与右子树，然后访问根。

简言之，后序遍历是按“左子树—右子树—根”的次序遍历二叉树。其实例见图 6-0。

在上列三种方式的定义中，采用了递归描述方式，这种描述方式简洁而准确。但注意

这种描述必须有始基。在上面的定义中，始基就是“若树为空，则不做任何动作”，若无此句，所定义的遍历将是个无限动作。

### (五) 层序遍历

层序遍历是按树的层序编号的次序访问各结点，即按从上层到下层，同层内从左到右的次序访问结点。实例见图 6-0。

遍历是树结构的一种重要的操作（我们在后面还要给出一般树的遍历的概念），许多对树的操作，都是基于遍历的。另外，遍历也是树的一种串行化，即将树中结点按某种方式线性输出。

## § 6.3 二叉树的存储结构

二叉树存储结构应能体现二叉树的逻辑关系，即单前驱多后继的关系。在具体的应用中，可能要求从任一结点能直接访问到它的后继（即儿子），或直接访问到它的前驱（父亲），或同时直接访问父亲和儿子。所以，存储结构的设计，要按这些要求进行。

### § 6.3.1 顺序存储结构

#### (一) 顺序二叉树的顺序存储结构

这种存储结构是按结点的层序编号的次序，将结点存储在一片连续存储区域内。

由**定理 6-7**知，对顺序二叉树，若已知结点的层序编号，则可推算出它的父亲和儿子的编号，所以，在这种存储结构中，很容易根据结点的相对地址计算出它的父亲和儿子的相对地址，方法是：

$x$  的相对地址  $\rightarrow x$  的编号  $\rightarrow x$  的父亲/儿子的编号 (性质 7)  $\rightarrow x$  的父亲/儿子的相对地址。

至于结点的相对地址与编号之间的换算，有下列关系：

结点相对地址 = (结点编号 - 1)  $\times$  每个结点所占单元数目

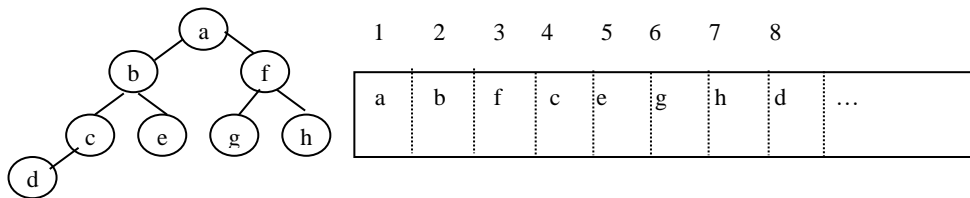


图 6-6 顺序二叉树的顺序存储

图 6-0 给出了这种存储结构的示例。这种存储方式，逻辑结构用存储次序体现，故若不进行插入与删除操作，是一种很经济的存储方式。

## (二) 一般二叉树的顺序存储

由于定理 6-7 只对顺序二叉树（包括满二叉树）成立，所以，对一般的二叉树而言，若要利用定理 6-7 访问一个结点的父亲与儿子，则需在该二叉树中补设一些虚结点，使它成为一棵顺序二叉树，然后对结点按层序编号。这样处理后，再按顺序二叉树的顺序存储方式存储。这种带虚结点的树，虚结点也要对应存储位置，但要设立虚结点标志。

这种存储方式中，实结点是不连续出现的，所以，若虚结点对应的存储位置不能被利用，则是一种很大的浪费（虚结点数目可能很大），图 6-0 给出了这种存储结构的示例。

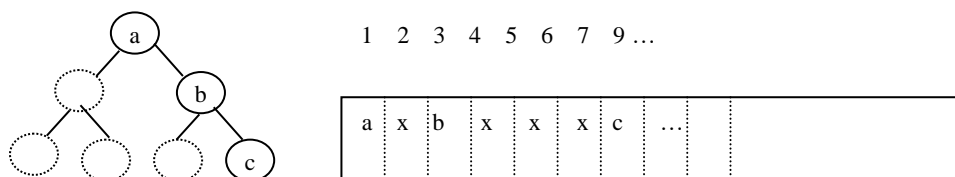


图 6-7 一般二叉树的连续存储(x 表示虚结点的位置)

## § 6.3.2 链式存储

二叉树一般多采用链式存储。这种存储方式的基本思想是，令每个树结点对应一个链结点，链结点除存放与树结点有关的信息外，还要根据具体应用的需要设置指示父亲、儿子的指针。根据指针设置情况，存储方式分为一指针式、二指针式和三指针式。

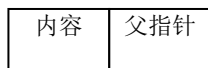
### (一) 一指针式

这种方法是，为每个结点只设立指向其前驱的指针。由于每个结点的前驱是唯一的，故每个结点只需设一个前驱指针。在树中，前驱称为父亲，所以这种方式也称父指针式。显然，这种存储方式下，已知某结点的指针，很容易找到它的父亲，但要找到它的儿子，需从叶子起搜索，很耗时。另一问题是，虽可知道儿子，但不能区分是左儿还是右儿。为了解决该问题，可在结点上设立左右儿标志位。因此，一指针式存储在存储了左右儿标志的情况下，是关系完备的（即存储了全部关系信息）。

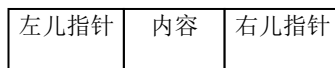
显然，对一指针结构，只有知道每个叶子的地址，才能访问到一棵树中的每个结点。因此，需要将一棵树中各个叶子的地址记录下来。为此，对每棵树，设立一个数组，将各叶子地址分别保存在数组的各个元素中。

---

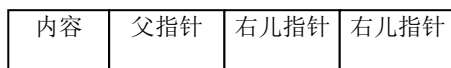
一指针式的结点结构如图 6-0(a)，示例如图 6-0(b)。



(a) 一指针结点

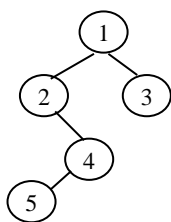


(b) 二指针结点

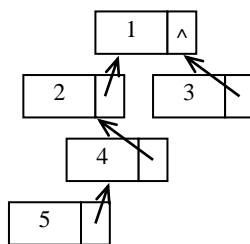


(c) 三指针结点

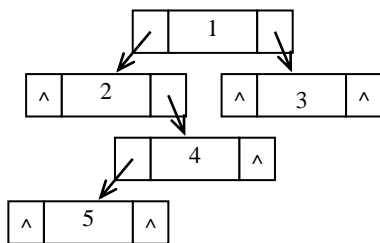
图 6-9 二叉树链式存储方式的结点结构



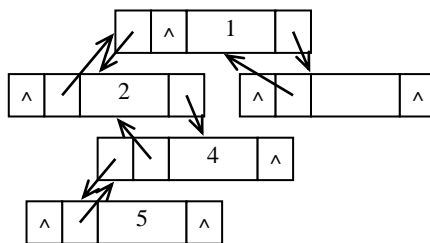
(a) 二叉树



(b) (a)树的一指针存储



(c) (a)树的二指针存储



(d) (a)树的三指针存储

图 6-8 二叉树链式存储结构示例

## (二) 二指针式

这种方法是，为每个结点只设立指向其后继的指针。由于二叉树每个结点的后继最多有两个，故每个结点需设二个后继指针，分别称为左儿指针和右儿指针。

显然，在这种储存方式下，从根出发可以访问到所有结点，所以，记录下根的地址后，就可以访问到各个元素。因此，二指针式在已知根的情况下，是关系完备的。

虽然已知某结点的指针，很容易找到它的儿子，但要找到它的父亲，一般需从根起搜索，很耗时。二指针式的结点结构如图 6-0 (b)，示例如图 6-0 (c)。

### (三) 三指针式

三指针式是一指针和二指针的结合，即每个结点分别设立三个指针，分别指向其前驱和两个后继。三指针式同时具有一指针和二指针的优点，当然是通过存储空间换来的。三指针式的结点结构如图 6-0 (c)，示例如图 6-0 (d)。

显然，三指针式在关系存储方面有冗余（我们前面已指出，二指针是关系完备的）。

与普通链式存储一样，二叉树的链式存储也既可以是“静态”的，也可以是“动态”的。不过，由于动态链式更多地隐蔽了存储管理细节，使我们能用更多的精力考虑其他问题，所以，我们下面一般以动态为主。当然，也将适当介绍静态方法。

### (四) 存储结构的高级语言描述

下面给出二叉树结点（三指针）的数据部分的 C++ 描述。关于它的完整对象描述，将在后面给出。

```
template <class TElem> //对树结点内容使用可变类型
struct TBinTreeNode
{
    TElem info; //树结点内容，类型为可变类型（类模板）
    TBinTreeNode *father; //父指针
    TBinTreeNode *lc, *rc; //左儿指针、右儿指针
};
```

## § 6.4 二叉树对象模型

下面将二叉树看作一个独立的对象，给出它的属性和方法（基本操作）接口描述。

二叉树对象主要涉及两个类：树结点类 TBinTreeNode0 和树类 TBinTree。下面分别介绍。

## § 6.4.1 二叉树结点对象

### (一) 抽象结点

结点类 `TBinTreeNode0` 是对二叉树的元素结点的抽象，它规定了针对结点本身的基本操作接口，使各种具体的结点结构，都可从它派生。在 C++ 中，用纯虚类表示。

从逻辑关系上讲，每个二叉树元素除应包含相应的内容外，还应包含表示关系的信息：儿子和父亲指示器。由于元素内容可通过类属（参数化类型）表示，所以，在纯虚类中可直接定义元素内容，但对关系，就不易这样处理，我们这里只给出它们的访问接口，具体的结构要根据具体的存储方式确定。

这里的关于元素结点的抽象操作的设置，我们采用最小化的原则，即只设置针对结点本身的内容和关系的操作，对涉及全局的操作，放到“树类”中。

下面是二叉树元素结点的抽象结构的定义。其中各成分从其名称可自明。

```
template <class TElem>
class TBinTreeNode0
{
public:
    TElem info; //树结点内容，使用可变类型

    virtual TElem& GetElem(){return info;}; //读元素内容
    virtual TBinTreeNode0* SetElem(TElem *e){info=*e; return this;}; //置元素值
    virtual TBinTreeNode0* GetSon(char sonNo)=0; //读元素的儿子指针
    virtual TBinTreeNode0* SetSon(char sonNo, TBinTreeNode0* pSon)=0; //置儿子指针

    virtual TBinTreeNode0* GetFather()=0; //读父指针
    virtual TBinTreeNode0* SetFather(char sonNo, TBinTreeNode0* f)=0; //置父指针

    char IsLeaf() {return GetSon(1)==NULL && GetSon(2)==NULL;}; //判断是否为叶子
};
```

### (二) 动态三指针结点

从上面给出的抽象类 `TBinTreeNode0`，可派生出各种具体存储结构的二叉树结点。这里，我们给出一种存储结构：动态三指针链结构 `TBinTreeNode`，它适合于使用堆空间（通过 C++ 的 `new`、`malloc` 等建立）。各成份在程序中作了注释，不再单独解释。

```
template <class TElem>
class TBinTreeNode :public TBinTreeNode0<TElem>
{
```



```

public:
TBinTreeNode<TElem> *lc, *rc, *father; //左儿指针、右儿指针、父指针

TBinTreeNode(){lc=rc=father=NULL;}; //初始时，将各指针置为空

virtual TBinTreeNode0<TElem>* GetSon(char sonNo)
    {if (sonNo==1) return lc; else return rc;};
    //读出指定的儿子指针。sonNo 为 1 时表示左儿，为 2 时表示右儿，下同
virtual TBinTreeNode0<TElem>* SetSon(char sonNo, TBinTreeNode0<TElem>* pSon)
    {sonNo==1? lc=
        (TBinTreeNode<TElem>*)pSon: rc=(TBinTreeNode<TElem>*)pSon; return pSon;
    };//将 pSon 设置为本对象的儿子 (sonNo=1 时置为左儿，否则置为右儿)
virtual TBinTreeNode0<TElem>* GetFather(){return father;}; //读父指针
virtual TBinTreeNode0<TElem>* SetFather(TBinTreeNode0<TElem>* f);
    { father=(TBinTreeNode<TElem>*)f;}; //把 f 置为本对象的父亲
};

```

该类是从 TBinTreeNode0 派生而来，实现了 TBinTreeNode0 中规定的操作。在后面的讨论中，除非特别声明，我都以该类为例。

### § 6.4.2 二叉树对象

为了方便对整棵树进行操作，设立二叉树类 TBinTree。该类用于封装具体的二叉树实体及其相关操作。

TBinTree 中包含的树实体是由一个类型为 TBinTreeNode0 的根指针 root 所指出。由于在 C++ 中，指向基类的指针，可以直接指向该基类的派生类，而且，我们已在树结点类中通过基本操作抽象隐蔽了结点的结构，所以，该树类对各种具体的树结点都是兼容的。下面是该类的 C++ 描述。

```

enum TTraverseMode {PreOrder, InOrder, PostOrder, LevelOrder};
    //定义一个枚举类型，表示树的几种遍历方式
template <class TElem>
class TBinTree    //树类
{
public:
    TBinTreeNode0<TElem> *root; //指向所代表的树的根
    long numNodes; //树结点总数

```

```

TBinTree();
~TBinTree();

virtual TBinTreeNode0<TElem>* GetRoot(){return root;};
virtual void SetRoot(TBinTreeNode0<TElem>* rt){root=rt;};

virtual long GetLevel(TBinTreeNode0<TElem>* pNode);
virtual long GetHeight(TBinTreeNode0<TElem>* pNode);
virtual long GetNumSubNodes(TBinTreeNode0<TElem>* pNode);
virtual long GetNumSubNodes(void);
virtual long GetNumLeaves(TBinTreeNode0<TElem>* pNode);

virtual long Cluster(TBinTreeNode0<TElem>* pNode,TElem **e,TTraverseMode tm);
virtual long Cluster(TBinTreeNode0<TElem>* pNode,
    TBinTreeNode0<TElem> **pe,TTraverseMode tm);
virtual long Cluster2(TBinTreeNode0<TElem>* pNode,
    TBinTreeNode0<TElem> **pe,TTraverseMode tm);
virtual long ClusterDescendants(TBinTreeNode0<TElem>* pNode,
    TBinTreeNode0<TElem> **pe, TTraverseMode tm = PreOrder,
    int startLevel=1, int endLevel=-1);
virtual long ClusterAncestors(TBinTreeNode0<TElem>* pNode,
    TBinTreeNode0<TElem> **pe);
virtual long ClusterAncestors(TBinTreeNode0<TElem>* pNode,TElem **e);
virtual long ClusterAncestors(TElem &e, TBinTreeNode0<TElem>** pNodes);
virtual long ClusterJuniors(TBinTreeNode0<TElem>* pNode,
    TBinTreeNode0<TElem> **pe,TTraverseMode travMode=PreOrder,
    int startLevel=1, int endLevel=-1);
virtual long ClusterSeniors(TBinTreeNode0<TElem>* pNode,
    TBinTreeNode0<TElem> **pe, TTraverseMode travMode=PreOrder,
    int startLevel=1, int endLevel=-1);
virtual TBinTreeNode0<TElem> *DeleteSubTree(
    TBinTreeNode0<TElem>* pNode,char sonNo);

/*下面这些在注释中的函数，这里不讨论其具体实现
virtual long ClusterDescendants(TBinTreeNode0<TElem>* pNode,
    TElem **es,TTraverseMode tm = PreOrder, int startLevel=1, int endLevel=-1);
virtual long ClusterDescendants(TElem &e ,TBinTreeNode0<TElem> **pe,

```

```

        TTraverseMode tm = PreOrder, int startLevel=1, int endLevel=-1);
virtual long ClusterDescendants(TElem &e ,TElem **es,
        TTraverseMode tm = PreOrder, int startLevel=1, int endLevel=-1);

virtual long ClusterJuniors(TBinTreeNode0<TElem>* pNode,
        TElem **es, TTraverseMode travMode=PreOrder, int startLevel=1, int endLevel=-1);
virtual long ClusterJuniors(TElem& e, TBinTreeNode0<TElem> **pe,
        TTraverseMode travMode=PreOrder, int startLevel=1, int endLevel=-1);
virtual long ClusterJuniors(TElem& e, TElem **es,
        TTraverseMode travMode=PreOrder, int startLevel=1, int endLevel=-1);

virtual long ClusterSeniors(TBinTreeNode0<TElem>* pNode,
        TElem **es, TTraverseMode travMode=PreOrder, int startLevel=1, int endLevel=-1);
virtual long ClusterSeniors(TElem& e, TElem **es,
        TTraverseMode travMode=PreOrder, int startLevel=1, int endLevel=-1);
virtual long ClusterSeniors(TElem& e, TBinTreeNode0<TElem> **pe,
        TTraverseMode travMode=PreOrder, int startLevel=1, int endLevel=-1);
virtual TBinTreeNode0<TElem> *Locate(TBinTreeNode0<TElem>* rt, TElem &e,
        TTraverseMode travMode=PreOrder, long sn=1);
*/
virtual long ReleaseSubs(TBinTreeNode0<TElem>* pNode);
virtual TBinTreeNode0<TElem> *Locate(TBinTreeNode0<TElem>* rt, TElem *e);

virtual TBinTreeNode0<TElem> *GListToTree(long *gListExp,
        TElem *es, long numElem);
virtual TBinTreeNode0<TElem> *PreOrderExToTree(TElem *nodes, int numElem);
        TBinTreeNode0<TElem> *PreOrderExToTree(TElem *nodes);
virtual long TreeToGList(TBinTreeNode0<TElem> *rt, TElem *e);

virtual void Print(TBinTreeNode0<TElem> *rt, char mode);
};

```

下面对其中主要成员函数做一简单介绍。

**TBinTree():** 构造函数，充当初始化操作，其主要任务是置 root 为空，表示当前包含的树为空树。

**~TBinTree():** 析构函数，通过调用 ReleaseSubs 将树的各个结点所占空间释放。

**GetRoot():** 返回树根指针。

**SetRoot(TBinTreeNode0<TElem>\* rt):** 设置根指针。

☞ 以上两个操作用于隐蔽 root。

**GetLevel(TBinTreeNode0<TElem>\* pNode):** 返回结点 pNode 在以 root 为根的树中的层号。定义根的层号为 1。

**GetHeight(TBinTreeNode0<TElem>\* pNode):** 返回以 pNode 为根的树（或子树，下同）的高度（深度）。定义根的高度为 1，空树的高度为 0。

**GetNumSubNodes(TBinTreeNode0<TElem>\* pNode):** 返回以 pNode 为根的树中的结点总数。

**GetNumLeaves(TBinTreeNode0<TElem>\* pNode):** 返回以 pNode 为根的树中的叶子结点总数。

**Cluster(TBinTreeNode0<TElem>\* pNode, TElem \*\*e, TTraverseMode tm):** 串行聚集。将以 pNode 为根的树中的各结点的内容的地址，按 tm 遍历次序存入 e 中，并返回树 pNode 中结点总数。该函数实质上实现的是常规的遍历操作，但这里采取将遍历到的结点的指针（遍历结果）存入指定的数组中的方式，要比指定“访问”函数更具通用性。

**Cluster(TBinTreeNode0<TElem>\* pNode, TBinTreeNode0<TElem>\* \*\*pe, TTraverseMode tm):** 与上面的 Cluster 类似，不同点是存储（返回）各结点的指针。

**Cluster2(...):** 与上面参数对应的 Cluster 相同，只是这里给出另一种实现方法（非递归）。

**ClusterDescendants(TBinTreeNode0<TElem>\* pNode, TBinTreeNode0<TElem>\* \*\*pe, TTraverseMode tm = PreOrder, int startLevel=1, int endLevel=-1):** 与上面的 Cluster 类似，不同的是，这里只遍历以 pNode 为根的树的 startLevel 层到 endLevel 层之间的结点（设 pNode 为第一层）。另一个 ClusterDescendants 也类似，只是存储遍历到的结点的内容的地址。startLevel 和 endLevel 为正数时，层号相对于 pNode（其层号设为 1），为负数时，表示“倒数”，即最深一层为-1，其上一层为-2，余类推。

**Virtual long ClusterAncestors(TBinTreeNode0<TElem>\* pNode, TBinTreeNode0<TElem>\* \*\*pe):** 聚集祖先。将结点 pNode 的各祖先结点的指针，按前序次序存入数组 pe，并返回其数目。注意，这里 pNode 的“祖先”是指 pNode “直系”前代，即从根到 pNode 的路径上的结点。另一个 ClusterAncestors() 类似，不同的是存储结点的内容的地址。

**virtual long ClusterJuniors(TBinTreeNode0<TElem>\* pNode, TBinTreeNode0<TElem>\* \*\*pe, TTraverseMode travMode=PreOrder, int startLevel=1, int endLevel=-1):** 后辈聚集。将 pNode 的第 startLevel 层到第 endLevel 层的后辈结点，按遍历次序 tm 存入 pe。这里，层号 startLevel 和 endLevel 是相对于 pNode 结点（其为第 1 层），它们取负数时含义同前解释。注意，这里，x “后辈”是指在（层次方面）所有位于 x 之下的结点，包括非直系结点。

**ClusterSeniors(TBinTreeNode0<TElem>\* pNode, TBinTreeNode0<TElem>\* \*\*pe, TTraverseMode travMode=PreOrder, int startLevel=1, int endLevel=-1):**

前辈聚集。将 pNode 的第 startLevel 层到第 endLevel 层的前辈结点, 按遍历次序 tm 存入 pe。这里, 层号 startLevel 和 endLevel 是相对于 pNode 结点, pNode 层号为 1, pNode 上一层为 2, 余类推。startLevel 和 endLevel 取负数时, 表示层号从总根 (pNode 所在树的根) 算起。

**DeleteSubTree(TBinTreeNode0<TElem>\* pNode, char sonNo):** 删除结点。将 pNode 的儿子号为 sonNo 的子树, 连根一同摘取下来, 并返回其根。

☞ 由于我们这里的二叉树没有具体的语义, 所以, 对树的删除操作, 只能这样处理。对具有具体含义的二叉树 (如后面要介绍的二叉排序树), 可单独删除一个结点 (不是整棵子树), 而将被删结点的儿子做适当“转让”。

**Locate(TBinTreeNode0<TElem>\* rt, TElem \*e):** 按内容定位。在以 rt 为根的树中, 查找元素值为 e 的结点, 返回其指针。找不到时, 返回 NULL。这里的查找方式是前序, 即按前序遍历的次序查找 e。若值为 e 的结点存在多个, 返回其第一个出现。

**TBinTreeNode0<TElem> \*Locate(TBinTreeNode0<TElem>\* rt, TElem &e, TTraverseMode travMode=PreOrder, long sn=1):** 按内容定位。在以 rt 为根的树中, 按指定的遍历次序, 查找元素值为 e 的第 sn 个出现的结点, 返回其指针。找不到时, 返回 NULL。这里, sn 相对于各个值为 e 的结点, 其可正可负, 若值为 e 的结点有 m 个, 则 sn=1 表示查找其第 1 个出现的 (按遍历次序) 值为 e 的结点, sn=-1 表示查找最后一个值为 e 的结点, 余类推。

**InPreToTree(TElem \*pa, TElem \*ia, long p1, long p2, long i1, long i2):** 串行化结果的还原。根据二叉树的中序遍历结果和前序遍历结果, 创建二叉树结构, 返回所创建的树的根。数组 pa 和 ia 分别存放二叉树的前序遍历结果和中序遍历结果 (结点内容), p1 和 p2 分别指出 pa 中的元素的起点和终点下标, i1 和 i2 分别指出 ia 中的元素的起点和终点下标。p1, p2, i1, i2 的设置, 主要是为了递归的实现。若不是为了递归算法的实现, 则可隐含起点为某个值 (如 0), 而只指出数组中元素个数。

遍历实质上是串行化, 但任何单独的遍历结果都是不可逆的 (即不能唯一的决定二叉树), 前序和后序结果的结合, 也不可逆, 而中序和前序结合, 或中序和后序的结合, 都是可逆的。该问题我们在后面还要进一步讨论。

也可以设置和实现由中序和后序遍历结果产生二叉树的操作, 方法与这里的类似, 具体留作练习。

**GListToTree(long InPreToTree(long \*gListExp, TElem \*es, long numElem):** 串行化结果的还原。根据二叉树的广义表表示, 创建二叉树, 返回所创建的树的根。数组 gListExp 中存放二叉树的广义表表达式, 数组 es 存放树结点的内容。numElem 表示待创建的二叉树的元素个数。gListExp 中, 结点用编号表示, 它所对应的结点内容在 es[k] 中, 这里, k 表示某结点在 gListExp 中的编号。

**PreOrderExToTree(TElem \*nodes, int numElem):** 串行化结果的还原。根据二叉树的前序遍历结果的扩充, 创建二叉树, 返回其根。numElem 表示待创建的二叉树的元素个数。

**PreOrderExToTree(TElem \*nodes, int numElem):** 类似于前, 只是采用递归算法。

**TreeToGList(TBinTreeNode0<TElem> \*rt, TElem \*e):** 广义表串行化。将以 rt 为根的二出叉树中各结点的内容的地址, 按广义表形式输出到 e, 返回 g 广义表的元素个数。

关于这些基本操作的具体实现, 我们放到下面几节介绍。

## § 6.5 二叉树的遍历操作的实现

这里的操作以二指针存储结构为例 (同样适合于三指针结构)。

### § 6.5.1 前序遍历操作的实现

#### (一) 前序递归算法

由于二叉树的前序遍历的定义是按递归方式描述的, 所以它的算法的描述采用递归方式是自然且简洁的, 具体的 C++ 语言描述见下面的程序

```
template <class TElem>
long TBinTree<TElem>::
ClusterPre(TBinTreeNode0<TElem>* pNode, TElem ** e, long& cnt)
{//前序遍历以 pNode 为根的二叉树, 将遍历到的结点的指针存放在指针数组 e 中,
//将遍历到的结点的个数存放在 cnt, 并将其作为返回值。调用该函数前, 实参 cnt 应为 0 值。
if (pNode==NULL) return 0;
e[cnt++]=&(pNode->GetElem()); //访问根结点
ClusterPre(pNode->GetSon(1), e, cnt); //前序遍历根的左子树
ClusterPre(pNode->GetSon(2), e, cnt); //前序遍历根的右子树
return cnt; //返回遍历到的结点的数目
}
```

为了指示遍历到的结点的存放位置, 以及返回遍历到的结点的数目, 在 ClusterPre 中设立了引用参数 cnt。也可以用 C/C++ 的静态(static)变量实现。因为静态变量分配在共享区, 其作用类似于全局变量, 只是对其他函数不可见。由于是递归程序, 所以使用普通局部变量达不到共享目的。但是, 使用静态变量会导致副作用, 因为下一次直接调用 ClusterPre 时, 静态变量仍保持当前值, 而没有初始化为 0。

在 ClusterPre 中增加的参数 cnt 不属于函数接口, 只是为了函数的实现而引入的, 所以参数 cnt 的出现, 破坏接口的“友好性”。为解决该问题, 我们对 ClusterPre 进行“包装”:

```
template <class TElem>
long TBinTree<TElem>::
```

```

ClusterPre(TBinTreeNode0<TElem>* pNode, TElem ** e)
{
    long cnt=0;
    return ClusterPre(pNode, e, cnt);
}

```

这样，我们就可避免直接调用 ClusterPre(pNode, e, cnt)。

🔊 这种“包装”方法很常用，特别是对递归程序。

## (二) 递归算法的跟踪

为了便于对递归算法的理解，这里用证明树的形式给出递归算法对图 6-0 所示树的执行过程的示意图（图 6-0）。

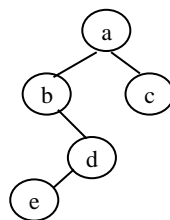


图 6-10 一棵二叉数

在图 6-0 中，用 C(x)表示前序遍历算法，它表示前序遍历以 x 为根的子树，用 P(x)表示输出/访问结点 x。方框中左边括号中的数字表示子程序调用序号。某方框的各个下属方框表示该方

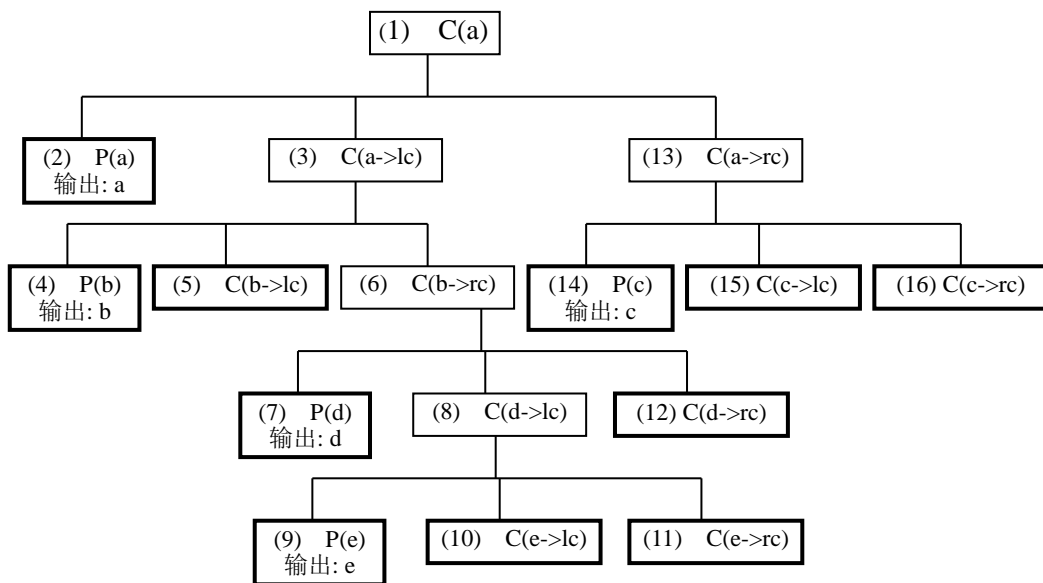


图 6-11 图 6-10 所示二叉树的前序遍历递归算法执行过程

框对应的子程序的执行的各个主要步骤（一次子程序调用）。

从图中知，在步骤 2、4、7、9、14 中分别输出了结点 a、b、d、e、c，这正是前序遍历结果。

### (三) 前序遍历非递归算法

一般而言，递归算法的效率比非递归低。对遍历算法，如果不采用高级语言的递归机制，遍历算法描述要复杂一些。由前序遍历的定义知，遍历某树时，是从根起，顺左分枝往下搜索，每搜索到一个结点，就访问该结点，直到遇到没有左分枝的结点为止。然后，返回到已搜索过的最近一个有右儿子的且该右儿子尚未被访问过的结点处，按同样的方法顺着它的右分枝往下搜索（访问），如此一直进行，直到所有结点均已处理完。

在访问完某结点后，不能立即放弃它，因为遍历完它的左子树后，要从它的右儿子起遍历，所以在访问完某结点后，应保存它的指针，以便以后能从它找到它的右儿子。由于较先访问到的结点较后才重新利用，故应将访问到的结点按栈的方式保存。具体的实现程序如下。

```
template <class TElem>
long TBinTree<TElem>::
ClusterPre2(TBinTreeNode0<TElem>* pNode, TBinTreeNode0<TElem> **pNodes)
{//前序遍历非递归程序。前序遍历以 pNode 为根的树，将遍历到的结点的指针存于 pNodes
//返回遍历到的结点的数目
long cnt=0;
long nNodes;
long top=0;
TBinTreeNode<TElem>** sk;
TBinTreeNode<TElem>* p;

nNodes=GetHeight(pNode); //调用成员函数求树高，以确定栈的大小
if (nNodes<=0) return 0;

sk =new (nothrow) TBinTreeNode<TElem> *[nNodes+1]; //分配栈空间，栈元素为结点指针
if (sk==NULL) throw TExceptComm(3);

p =(TBinTreeNode<TElem> *) pNode; //令 p 开始时指向根
while (p!=NULL || top!=0)
{
    if (p!=NULL)
    {
        pNodes[cnt++]= p; //访问元素（存入 pNodes）
```



```

    top++; sk[top]=p; //p 进栈
    p = (TBinTreeNode<TElem> *)p->GetSon(1); //令 p 指向 p 的左儿子
}
else
{
    //已走到左分枝的底，返回到父亲或其他祖辈的右儿子（结点）处
    //若该结点无右儿子，则 p=NULL，从而在下次循环时，右进入该分枝，继续回退
    p = sk[top]; top--; //退栈
    p = (TBinTreeNode<TElem> *)p->GetSon(2); //令 p 指向原栈顶元素的右儿子
}
} //while
delete[] sk; //释放所分配的栈空间
return cnt; //返回所访问到的结点数目
}

```

为了容易理解该算法，下面我们展示该算法的执行过程，如表 6-1 所示。表中的步骤  $i$  是指第  $i$  次循环刚刚执行完，而第  $(i+1)$  次尚未开始。

上一小节和这里分别针对递归算法和非递归算法，给出了跟踪执行过程的示例。虽然是具体例子，但它们的方法具有普遍意义。当一个算法很难理解或出了难以发现的错误，都可用这些方法试验。

**时空复杂度分析：**该算法每循环一次， $p$  指向一个结点或空目标，所指向的空目标是无左儿子或无右儿子的结点的空链域，因此所指向的空目标的次数为  $(n+1)$ ， $n$  为结点个数，故循环次数为  $n+(n+1)=2n+1$ ，从而算法的时间复杂度为  $O(n)$ 。由于每个结点恰好进栈 1 次，所以栈空间最大需求量是  $n$ （当树蜕化为线性链时达到最大）。

表 6-1 二叉树非递归算法执行过程

步骤	P 所指	栈中内容	访问到的结点	说明
0	A	空		进入循环前
1	B	a	a	a 进栈，p 指向 a 的左儿(b)
2	^	a b	b	b 进栈，p 指向 b 的左儿(空)
3	D	a		b 出栈，p 指向 b 的右儿(d)
4	E	a d	d	d 进栈，p 指向 d 的左儿(e)
5	^	a d e	e	e 进栈，p 指向 e 的左儿(空)
6	^	a d		e 出栈，p 指向 e 的右儿(空)
7	^	a		d 出栈，p 指向 d 的右儿(空)
8	C	空		a 出栈，p 指向 a 的右儿(c)
9	^	c	c	c 进栈，p 指向 c 的左儿(空)
10	^	空		c 出栈，p 指向 c 的右儿(空)
11				结束

## § 6.5.2 中序遍历操作的实现

### (一) 递归算法

中序遍历的递归算法也是很直接的，具体描述见下面的程序。该程序与前面介绍的前序遍历递归程序类似，主要不同是，该程序中，访问结点的操作出现在遍历左子树和遍历右子树的两个操作之间。

```
template <class TElem>
long TBinTree<TElem>::
ClusterIn(TBinTreeNode0<TElem>* pNode, TElem ** e, long& cnt)
{
    if (pNode==NULL) return 0;
    ClusterIn(pNode->GetSon(1), e, cnt);
    e[cnt++]=&(pNode->GetElem()); //访问根结点
    ClusterIn(pNode->GetSon(2), e, cnt);
    return cnt;
}

template <class TElem>
ClusterIn(TBinTreeNode0<TElem>* pNode, TElem ** e)
{
    long cnt=0;
    return ClusterIn(pNode, e, cnt);
}
```

### (二) 非递归算法

中序遍历非递归算法与前序遍历非递归很相似。在前序遍历中，每搜索到一个结点，就访问它，并将它推进栈。但在中序遍历中，搜索到结点时并不能立即访问，只是将它推进栈，等到左分枝搜索完后再从栈中弹出并访问之。由此可知，中序遍历的非递归算法，只需在前序遍历非递归程序的基础上，将访问结点语句调到出栈语句后即可。具体描述见下面的程序。

```
template <class TElem>
long TBinTree<TElem>::
ClusterIn2(TBinTreeNode0<TElem>* pNode, TBinTreeNode0<TElem> **pNodes)
{
```

```
long cnt=0;
long nNodes;
long top=0;
TBinTreeNode<TElem>** sk;
TBinTreeNode<TElem>* p;

nNodes=GetHeight(pNode);
if (nNodes<=0) return 0;

sk =new TBinTreeNode<TElem> *[nNodes+1];
if (sk==NULL) throw TExcepComm(3);

p =(TBinTreeNode<TElem> *) pNode;
while (p!=NULL || top!=0)
{
    if (p!=NULL)
    {
        top++; sk[top]=p;
        p = (TBinTreeNode<TElem> *)p->GetSon(1);
    }
    else
    {
        p = sk[top]; top--;
        pNodes[cnt++]= p;
        p = (TBinTreeNode<TElem> *) p->GetSon(2);
    }
} //while
delete[] sk;
return cnt;
}
```

### § 6.5.3 后序遍历操作的实现

#### (一) 递归算法

后序遍历递归算法同样很简单,与前序遍历也很类似,不同的是,将访问元素的操作移到遍历左子树和遍历右子树的后面。具体实现见下面的程序。

```
template <class TElem>
long TBinTree<TElem>::
ClusterPost(TBinTreeNode0<TElem>* pNode, TElem ** e, long & cnt)
{
    if (pNode==NULL) return 0;
    ClusterPost(pNode->GetSon(1), e, cnt);
    ClusterPost(pNode->GetSon(2), e, cnt);
    e[cnt++]=&(pNode->GetElem()); //访问元素
    return cnt;
}

template <class TElem>
long TBinTree<TElem>::
ClusterPost(TBinTreeNode0<TElem>* pNode, TElem ** e, long & cnt)
{
    long cnt=0;
    ClusterPost(pNode, e, cnt);
}
```

## (二) 非递归算法\*

后序遍历的非递归算法要复杂一些。由于后序遍历最后访问根结点，所以对任一结点，应先沿它的左分枝往下搜索，每搜索到一个结点就进栈，直到遇到一个无左分枝结点为止，此时，若该结点无右子树，则直接访问该结点，否则从该结点的右子树的根起，按同样的方法沿左分枝处理，处理完右分枝后，才访问该结点。因此，若每搜索到一个结点就将其进栈，则对任一结点，当处理完它的左分枝时（此时它位于栈顶）不能将它弹出栈，而应继续处理它的右分枝，直到右分枝处理完后（此时它又位于栈顶）才能出栈并访问它。因此，在后序遍历中，每一结点都有两次出现在栈顶（不包括刚进栈时的情况），这两种情况的含意与处理方法为：

- 结点 p 第 1 次出现在栈顶：  
含意：p 的左子树处理完（遍历完），右子树尚未遍历。  
操作：不出栈，利用它找到它的右儿子，遍历它的右子树。
- 结点 p 第 2 次出现在栈顶：  
含意：p 的右子树已处理（遍历）完（左子树亦遍历完）。  
操作：出栈，并访问它。

为了区分是第几次出现在栈顶，需为每个进栈的结点设个标志域 `isFirst`，刚进栈时，

将它置为 1 (“真”), 当它第 1 次出现在栈顶后, 将它置为 0 (“假”)。具体的算法描述见下面的程序。

```
template <class TElem>
long TBinTree<TElem>::
Cluster2(TBinTreeNode0<TElem>* pNode, TBinTreeNode0<TElem> **pNodes)
{//后序遍历非递归程序。遍历以 pNode 为根的树, 将遍历到的结点的指针存入 pNdes
//返回遍历到的结点的数目
long cnt=0;
long nNodes;
long top=0;
TBinTreeNode<TElem>* p;

if (pNode==NULL) return 0;
nNodes=GetHeight(pNode); //求树高
if (nNodes<=0) return 0;
struct TPostTraverseStackNode //定义栈元素类型
{
    TBinTreeNode<TElem> *pNode; //结点指针
    char isFirst; //标志, 为 1 时表示 pNode 第一次出现在栈顶
};

TPostTraverseStackNode *sk2;

sk2 =new TPostTraverseStackNode [nNodes+1]; //申请栈空间
if (sk2==NULL) throw TExcepComm(3);

p =(TBinTreeNode<TElem> *) pNode;
while (p!=NULL || top!=0)
{
    if (p!=NULL)
    { //p 不空时顺着左分枝走
        top++;
        sk2[top].pNode=p;
        sk2[top].isFirst = 1;
        p = (TBinTreeNode<TElem> *)p->GetSon(1); //令 p 指向它的左儿子
    }
    else //p 空, 走到了左分枝的底
```

```
if (sk2[top].isFirst)
{
    //当栈顶元素是第一次出现在栈顶时
    sk2[top].isFirst = 0; //将标志置为“非第一次”
    p = (TBinTreeNode<TElem> *) (sk2[top].pNode->GetSon(2)); //p 指向栈顶元素的右儿子
}
else
{
    //当栈顶元素是第二次出现在栈顶时
    p = sk2[top].pNode; //出栈到 p
    top--;
    pNodes[cnt++] = p; //访问 p
    p=NULL; //置 p 为空，以强制下次循环进入"if (p!=NULL)"的 else 分枝
}
} //while
delete [] sk2; //释放栈空间

return cnt; //返回所访问到的结点个数
}
```

至于二叉树的层序遍历，易使用非递归算法。具体实现时需使用队列存放结点（利用队列的 FIFO 特点）。具体的实现程序，留作练习。

比较一下我们这里给出的递归算法和非递归算法，前面已指出递归算法的速度一般没有非递归快，那么，它们的空间需求又如何？

表面看上去，递归算法不象非递归算法那样使用批量数据空间（这里是栈），但是，由于递归的执行，需反复调用自己，这如同调用其他函数。每调用一次，系统就在系统的运行栈中存放函数的参数、临时变量等，若在函数中又调用函数，则运行栈就继续增长，只有函数彻底执行完，才释放它所占用的栈空间。因此，递归算法不但不节省空间，而且一般比非递归更浪费空间。

## § 6.6 二叉树的解析表示与存储结构之间的转化

二叉树的解析表示，是指二叉树的表达式或文字信息表示，是串行化结果，这里，串行化是指将结点形成一个序列。这种表示应该是无二义的。解析表示与存储结构的相互转化具有很大实用价值。例如，当需要将二叉树存储到永久存储器上（即文件中）时就需要这种转化：存储时需将存储结构转化为解析表示，而恢复时需将解析表示转化为存储结构。

本节以几种常见的解析表示为例，介绍转化算法。

### § 6.6.1 双遍历结果转化为树

在这里，我们将树的前序、中序、后序遍历序列看作是二叉树的一种解析表示，那么，上节所介绍的遍历算法就是将二叉树存储结构转化为解析表示的算法。这里我们考虑这个问题的逆，即如何将解析表示转化为存储结构（二指针式）。

由上节知，给定一棵二叉树可产生唯一的前序遍历结果、中序遍历结果、后序遍历结果与层序遍历结果，那么反之亦成立吗？即，可由某种遍历结果唯一地确定一棵二叉树吗？请看一个实例，图 6-0 给出了二棵不同构的有序二叉树，但它们分别有着相同的前序遍历序列和后序遍历序列：

前序：1 2 4 3

后序：4 2 3 1

这表示，不能由前序遍历结果单独确定一棵二叉树，也不能由后序遍历结果单独确定一棵二叉树。同样方法可以证明，单由中序遍历结果或层序遍历结果亦不能确定二叉树。

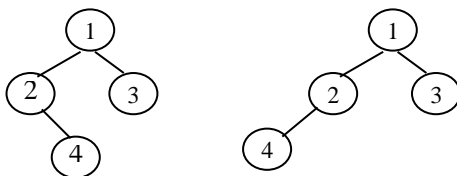


图 6-12 两棵不同构的有序二叉树

那么，若干种遍历结果结合起来能否唯一地确定二叉树？从图 6-0 已看到，两棵不同的树，却对应着相同的前序序列和相同的后序序列！因此，由前序遍历结果和后序遍历结果的结合也不能单独确定一棵二叉树。

事实上，可由中序遍历结果与前序遍历结果，或者中序遍历结果与后序遍历结果唯一地确定一棵二叉树。下面考虑由中序遍历结果与前序遍历结果确定二叉树的情形。

由中序遍历序列与前序遍历序列确定对应的二叉树的基本思想是，用前序序列识别根，用中序序列区分左右子树。具体构造方法是，从前序遍历序列中找出第 1 个结点，其必为待构造的二叉树的根，然后在中序遍历序列中查找该结点，则它（在中序序列中）的左边为左子树结点（左边无结点时，左子树为空），右边为右子树结点（右边无结点时，右子树为空），这样，可在前序遍历序列中找出左子树结点与右子树结点的分界点（如图 6-0 所示）。因此，在前序与中序遍历序列中就找到了根的左子树结点序列与右子树结点序列，从而，可用相同的方法再处理左子树结点序列与右子树结点序列。左右子树处理完后，整个二叉树也就确定了。

**例 6-4** 设某二叉树的前序与中序遍历序列分别为

前序：1 2 4 6 3 5 7 8

中序：2 6 4 1 3 7 5 8

则相应的二叉树建立过程如图 6-0(a)~(e)所示。

下面考虑转化的算法实现。

上面给出的由中序与前序遍历序列确定相应二叉树的过程是个递归过程，故用递归算法较为自然。为了实现递归，在递归过程的参数中，应设置表示当前待确立的子树的中序与前序序列的范围。具体程序如下。对该程序中的两次递归调用的参数的设置的理解，请参阅图 6-0。

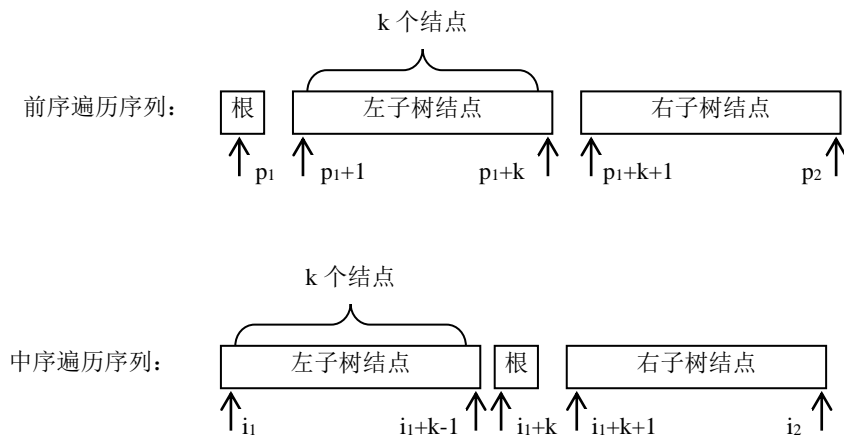
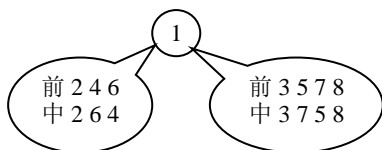
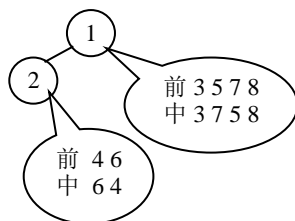


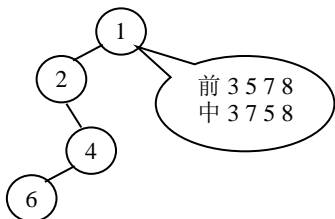
图 6-13 由中序和前序确定二叉树



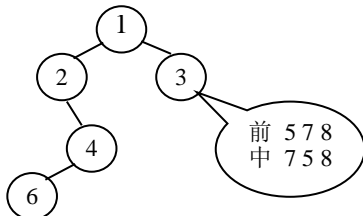
(a) 分出根(1)及其左右子树的前/中序序列



(b) 确定 1 的左子树：根为 2，  
无左子树，右子树前中序序列

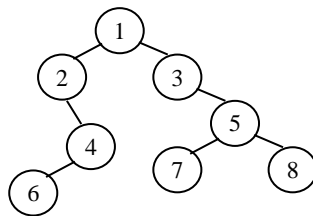


(c) 确定 2 的右子树：根为 4  
左子树为 6，右子树空



(d) 确定 1 的右子树：根为 3，  
3 无右子树





(e) 确定 3 的右子树：根为 5，  
左右子树分别为 7 和 8

图 6-14 利用中序与前序遍历序列确定二叉树示例

```

template <class TElem>
TBinTreeNode0<TElem> *TBinTree<TElem>::
InPreToTree(TElem *pa, TElem *ia, long p1, long p2, long i1, long i2)
{ //根据前序序列 pa[p1]~pa[p2]和中序序列 ia[i1]~ia[i2], 创建对应的二叉树结构, 返回其根指针
  long k;
  TBinTreeNode<TElem> *p;

  if (i1>i2) return NULL; //递归始基, i1>i2 时表示当前序列代表空树

  k=0;
  while (pa[p1]!=ia[i1+k]) k++; //找左右子树的分界点 k,中序中 i1 起的 k 个元素为左子树

  p= new(nothrow) TBinTreeNode<TElem>; //新生成一个结点
  p->SetElem(&pa[p1]); //将 p1 所对元素做为新结点的值
  p->SetSon(1, InPreToTree(pa, ia, p1+1, p1+k, i1, i1+k-1) ); //构造 p 的左子树
  p->SetSon(2, InPreToTree(pa, ia, p1+k+1, p2, i1+k+1, i2) ); //构造 p 的右子树

  //下面建立起 p 与它的儿子的父亲链, 对纯二指针结构, 不需要这两句
  if (p->GetSon(1)!=NULL) p->GetSon(1)->SetFather(1,p);
  if (p->GetSon(2)!=NULL) p->GetSon(2)->SetFather(2,p);

  return p; //返回所创建的树的根
}

```

## § 6.6.2 根据广义表表示创建树

### (一) 二叉树的广义表表示

一棵以  $r$  为根的二叉树的广义表表示（广义表表达式）定义为如下形式：

(R)

其中， $R$  可递归地定义为：

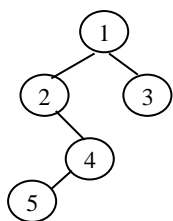
- a) 若  $r$  为空树，则  $R$  的形式为星号，即 “\*”；
- b) 若  $r$  是叶子，则  $R$  的形式为： $rN$
- c) 若  $r$  是非叶子结点，则  $R$  的形式为： $rN(rL, rR)$

其中， $rN$  为结点  $r$  的标识， $rL$  和  $rR$  分别为  $r$  的左右子树的广义表表示。

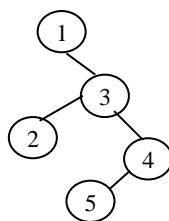
二叉树要明确区分左右子树（有序树），而且，可以只有右子树而无左子树，所以，对空子树，也应有相应表示法。

关于广义表，我们在其他章节中还要专门介绍，这里是介绍如何使用它表示二叉树。

图 6-0 给出了几个表示实例。



广义表： $(1(2(*,4(5,*)),3))$



广义表： $(1(*,3(2,4(5,*)))$

图 6-15 二叉树的广义表表示

### (二) 根据广义表创建树的非递归算法

这里介绍由二叉树的广义表表示创建二指针式的存储结构的非递归算法。为了突出主题，我们假定广义表表达式已被规格化到一维整型数组中（可存储字符）中，无语法错误，每个数组元素存放一个广义表表达式符号（或结点标识），各符号之间无空格存储，表达式中树结点用整数表示（称结点标识），而实际的结点值存放在另一个一维数组中，结点的整数标识就是结点值在该数组中的下标。在实际使用中，则需构造一个简单的规格化程序，使广义表表达式符合该要求。

由广义表表示法的性质知，我们按从左到右的顺序读广义表表达式，当读到一个表示

结点的符号时, 应建立相应的内存结点, 但由于它的子树的信息尚未读出, 故此时不能确定它的左右链域, 但它的父亲已被读出并建立, 故可将它链到它的父亲结点上。也就是说, 结点的儿子链域是滞后确定的, 因此, 应将当前建立的结点的指针保存起来, 以备以后确定它的左右链域。假定读到一个结点时, 产生它的内存结点后, 就将内存结点指针推入栈, 而它的左右子树均处理完后弹出栈, 则需要在遇到逗号或右括号时退栈, 遇到结点时进栈, 而对左括号不作处理。具体算法见下面的程序。

```
template <class TElem>
TBinTreeNode0<TElem> *TBinTree<TElem>::
GListToTree(long *gListExp, TElem *es, long numElem)
{//根据二叉树的广义表表达式 gListExp 和 es, 在内存创建对应的二叉树, 返回其根
//一维数组 gListExp 存放广义表表达式, numElem 是表达式长度。
//在表达式中, 每个树元素用一个整数(编号)表示, 全体树元素对应连续自然数 1~n,
//这里 n 为树元素个数。编号为 i 的元素的值存放在一维数组 es 的 es[i]中
TBinTreeNode<TElem> *p, *rt, **s;
long top, i;

if (numElem<2) return NULL; //表达式长度不足 2 时非法

s = new(nothrow) TBinTreeNode<TElem> *[numElem+1]; //申请栈空间
if (s==NULL) throw TExcepComm(3);

p = new(nothrow) TBinTreeNode<TElem>; //申请一个树结点, 做为根
if (p==NULL)
{
delete [] s; //切记, 返回前释放所申请的工作空间
throw TExcepComm(3);
}

top=0; i=1;
rt = p;
p->SetElem(&es[gListExp[i]]); //为根结点置值
s[++top] = p; //根进栈
do
{
i++; //为了顺序读下个表达式符号
if (gListExp[i]=='(') continue; //忽略左括号
if (gListExp[i]==',' || gListExp[i]==')') top--; //遇逗号或右括号时进栈
```

```

else
{
    if (gListExp[i]=='*') p = NULL; //遇"*"时置 p 为 NULL，表示当前结点是空结点
    else
    { //读到的是表示元素的符号
        p = new(nothrow) TBinTreeNode<TElem>; //申请树结点
        if (p==NULL)
        {
            delete [] s; //切记，返回前释放所申请的工作空间
            throw TExcepComm(3);
        }
        p->SetElem(&es[gListExp[i]]); //为所申请的树结点置值
    } // if (gListExp[i]=='*')
    //下面将 p 链到它的父亲儿子链上
    if (gListExp[i-1] == '(' )
    {
        s[top]->SetSon(1, p); //p 为左儿子
        if (p!=NULL) p->SetFather(1, s[top]); //置父链
    }
    else
    {
        s[top]->SetSon(2, p); //p 为右儿子
        if (p!=NULL) p->SetFather(2, s[top]); //置父链
    }

    s[++top] = p; //当前生成的结点 p 进栈
    } // if (gListExp[i]=='*' || gListExp[i]==')') ...else
} while (top!=0);

delete [] s;
return rt;
}

```

### (三) 广义表建树的递归算法

上面给出的是广义表建树的非递归程序，这里我们考虑递归实现。考查一下我们这里的二叉树广义表表达式，其表元素有三种：空子树符号“\*”，叶子结点符号（其后不带左

圆括号)与子树的广义表表达式。所以,在算法中,应分别考虑这三种情况,即先读出根结点名称(设为 $r$ ),为其申请一个树结点,然后读它的左右子树,若子树的根结点为“\*”,表示该子树为空,则将 $r$ 的相应链域置为空;若 $r$ 的子树为叶子,则为其分配一个树结点,并将 $r$ 的相应链域置为所分配结点的地址;若 $r$ 的子树不是叶子,则通过递归调用建立该子树,然后将子树的根指针作为 $r$ 的根结点的相应链域的值。不论何种情况,每处理完一个表元素,应跳过分隔符,使下次的处理(读取)位置为下个表元素的开头,具体算法描述这里省略。

### § 6.6.3 根据存储结构创建广义表\*

现在讨论上节的逆问题:从二叉树的存储结构创建对应的广义表表达式。

考查一下二叉树的广义表表达式,就会发现,表达式中的结点是按前序遍历次序排列的,只是中间插入了分隔符(括号、逗号及星号),所以由存储结构生成表达式,是个前序遍历问题,只是要在遍历中,除输出结点编码外,还要输出适当的分隔符。

设 $p$ 为某树(子树)的根指针,则输出以它为根的二叉树(子树)的广义表的过程为。为了能递归描述,所输出的广义表表达式的最外层不加圆括号。

```
TreeToGList(p)
{
    if (p=空) 输出星号"*";
    else
    {
        输出 p 的编码;
        if (p 不为叶子)
        {
            输出左圆括号 "(";
            TreeToGList (p 的左子树);
            输出逗号 ",";
            TreeToGList (p 的右子树);
            输出右圆括号 ";";
        }
    }
}
```

下面考虑它的 C 语言实现。假定广义表表达式输出到一个字符数组(字符串)中,每个数组元素存放一个广义表表达式符号。具体程序如下。


```
template <class TElem>
long TBinTree<TElem>::TreeToGList(TBinTreeNode0<TElem> *rt, TElem *e)
```

```

{
    static long cnt=0; //所用静态变量 cnt，用于在递归中记下已输出的符号数目。
                        //但更好方法是将 cnt 放到参数中，这样可避免副作用！

    if (rt==NULL) e[cnt++]='*'; //对空（子）树，用星号表示
    else
    {
        e[cnt++] = rt->GetElem(); //输出元素值
        if (!rt->IsLeaf())
        { //处理非叶子
            e[cnt++] = '(';
            TreeToGList(rt->GetSon(1), e); //输出左子树
            e[cnt++] = ',';
            TreeToGList(rt->GetSon(2), e); //输出右子树
            e[cnt++] = ')';
        }
    }
    return cnt;
}

```

 请思考，如果不使用静态变量 cnt，程序要如何修改？

### § 6.6.4 前序扩展序列创建树\*

我们已知道，单独根据二叉树的一种遍历序列是不能唯一确定二叉树的。但通过在遍历序列中加适当信息就可唯一地确定二叉树了。前面给出的二叉树广义表表达式就是一个例子。现在我们考察如何给二叉树前序遍历序列添加其他种类信息，使其能唯一确定二叉树。

二叉树前序遍历序列之所以不能单独确定二叉树，是因为单靠它不能区分树的左右子树。我们可以按下列方式使其能区分左右子树，每个（非空）结点，都写出它的左右子树，若子树为空，则用特殊符号代表（这里用\*）。例如，图 6-0 的两棵二叉树，它们的扩展序列分别为

1 2 \* 4 5 \* \* \* 3 \* \*

1 \* 3 2 \* \* 4 5 \* \* \*

下面给出程序实现。有两个程序，分别为非递归和递归程序。程序中，用-1 代表空标志。进一步的说明，见程序中的注释。

```

template <class TElem>
TBinTreeNode0<TElem> *TBinTree<TElem>::

```

```
PreOrderExToTree(TElem *nodes, int numElem)
{
    //非递归建树：根据二叉树前序扩展序列（存于一维数组 nodes 中，该数组共 numElem 个元素），
    //返回所创建的树的根
    int i, top;
    TBinTreeNode<TElem> *p, *rt;
    struct TPreOrderExToTreeStackRec //工作栈元素类型
    {
        TBinTreeNode<TElem> *pNode; //指针
        char tag; // “处理” 标志：0--左右子树均未处理；1--左子树已处理完
    };

    TPreOrderExToTreeStackRec *sk;
    if (nodes[0]==-1) return NULL;

    sk = new(nothrow) TPreOrderExToTreeStackRec[numElem]; //申请栈空间
    if (sk==NULL) throw TExcepComm(3);

    rt = new TBinTreeNode<TElem>; //创建树根
    if (rt==NULL) goto ClearUp;

    rt->SetElem(&nodes[0]); //为根结点置值

    top=0;
    top++; sk[top].pNode=rt; //根结点进栈
    sk[top].tag=0; //置根结点的处理标志为“未处理”，表示它的左右子树均未建立

    i=1;
    do
    {
        if (nodes[i]==-1) p=NULL; //读到空标志，置 p 为空，以代表空子树
        else
        {
            //读到结点
            p = new (nothrow) TBinTreeNode<TElem>; //申请新结点
            if (p==NULL) goto ClearUp;
            p->SetElem(&nodes[i]); //为新结点置值
        }
        i++;
    }
```

```

    if (!sk[top].tag)
    { //当前栈顶元素是“未处理”结点时
        sk[top].pNode->SetSon(1,p); //将新结点 p 做为当前栈顶的左儿子
        sk[top].tag=1; //置 p 为“左子树已处理”标志
    }
    else //当前栈顶元素是“左子树已处理”结点时
    {
        sk[top].pNode->SetSon(2,p); //将新结点 p 做为当前栈顶的右儿子
        top--; //当前栈顶的左右子树已处理完，出栈
    }
    if (p!=NULL)
    {top++; sk[top].pNode=p; //新结点 p 进栈
      sk[top].tag=0; //置“未处理”标志
    }

} while (top!=0);

return rt; //返回树根指针

ClearUp: //程序非正常时转到这里，清理现场
    delete [] sk;
    throw TExcepComm(3);
}

```

```

template <class TElem>
TBinTreeNode0<TElem> *TBinTree<TElem>::PreOrderExToTree(TElem *nodes)
{ //递归建树，根据一维数组 nodes 中的前序扩展序列，在内存创建二叉树，返回其根
  //在前序扩展序列正确的情况下，不需要指出 nodes 中的元素个数
  TBinTreeNode<TElem> *rt;
  static long start=0;
  //静态变量 start 用于指示 nodes 中当前被处理的结点，其每次递归调用都在上次的基础上累加

  if (nodes[start]==-1)
  { //空子树标志，返回 NULL 表示创建了空子树
      start++;
      return NULL;
  }
}

```



```

rt = new TBinTreeNode<TElem>; //创建一个树结点，做为当前子树的根
rt->SetElem(&nodes[start]); //为树结点置值
start++;
rt->SetSon(1,PreOrderExToTree(nodes)); //为 rt 创建左子树
rt->SetSon(2,PreOrderExToTree(nodes)); //为 rt 创建右子树

return rt; //返回当前所创建的子树的根
}

```

## § 6.7 二叉树的线索化

### § 6.7.1 线索化的概念

二叉树是非线性结构，树中的结点可有多个后继。但若按某种方式遍历，遍历结果序列就变为线性结构，每个结点就有了唯一的前趋与后继。有时需知道二叉树结点在某种遍历方式下的前趋与后继。这就是线索化的目的。

一棵具有  $n$  个结点的二叉树，有  $(n+1)$  个空链域，即  $2n_0 + n_1$ ，占总链域数目  $(2n)$  的  $1/2$  多。这些空链域有时可用来存放一些有用的信息。这里，我们考虑用空链域存放结点在某种遍历方式下的前趋或后继指针。

规定对任一结点，用空左链域存放它的前趋的指针，而用空右链域存放它的后继的指针。若某链域不空，则不存储这种前趋与后继指针。这样，既保持了原树的结构，又利用空链表示了部分前趋与后继关系。我们称这种使树中结点空链域存放前趋或后继信息的过程为**线索化**（Threaded），被线索化了的树为**线索树**。

由于树的遍历方式有四种，故有四种意义下的前趋与后继。相应有四种线索化方法和线索树：前序线索化/树、中序线索化/树、后序线索化/树及层序线索化/树。

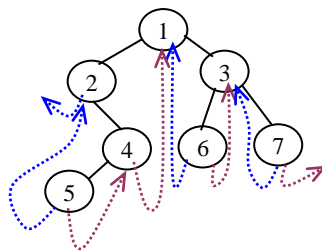


图 6-16 中序线索树  
中序：2 5 4 1 6 3 7

**例 6-5** 中序线索化树。如图 6-0 所示。图中虚线表示线索化（即前趋或后继指针），其中，浅（蓝）虚线表示前趋，深（红）虚线表示后继（颜色仅针对电子版）。

在线索化树中，由于原来的空链也被置为指向前趋或后继的指针，所以，对每个结点，必需设一个标志，来说明它的左链和右链指向的是树结构信息（如儿子）还是线索化信息（前趋还是后继）。为节省存储空间，可用两个二进位表示，其定义为

- 00: 左右链均为树结构信息;
- 01: 左为树结构信息, 右为线索;
- 10: 左为线索, 右为树结构信息;
- 11: 左右均为线索;

该标志可以与其它域共用存储单元 (以节省空间), 也可单独用一个存储单元表示。这里为简明, 用一个单独的存储单元 (在 C 中是字符型量) 表示。

因此, 需改写二叉树二指针存储结构的描述, 增加关于线索的标志:

```
struct TBinTreeNodeThread
{
    TElem info;
    struct TBinTreeNode *lc, *rc;
    unsigned char threadTag;
    ... ..
};
```

按面向对象观点, 该类/结构应是前面介绍的 TBinTreeNode 的派生类:

```
struct TBinTreeNodeThread : public TBinTreeNode
{
    unsigned char threadTag;
    ... ..
};
```

## § 6.7.2 线索化算法

线索化过程实质上是一种特殊的遍历过程。在线索化中, “访问” 结点就是要检查结点的左右链是否为空, 若空, 则令其指向 (遍历意义下的) 前趋或后继。在遍历中, 访问某结点时, 由于它的前趋 (遍历结果意义下的前驱, 下同) 已被访问过 (最近一次访问过), 所以若左链不空, 则可令其指向它的前趋, 但由于它的后继 (遍历结果意义下的后继, 下同) 尚未访问到, 所以它的右链不能进行线索化, 而要等到下次访问后才能进行线索化, 即右链的线索化要滞后一步进行。为了实现右链的滞后一步线索化, 设立一个指针, 令它指向上次访问到的结点。

下面以中序线索化为例, 给出线索化算法的 C 语言描述。这是个递归程序, 由于每次递归都需要知道最近一次 (即紧上一次) 访问的结点, 所以设置 pre 记下最近一次访问到的结点的指针。pre 的初值必须设为 NULL。由于是递归程序, pre 不能作为该程序中的普通临时变量。这里, 我们将 pre 作为该函数的参数。将其作为函数参数, 仅仅是为了让其对该函数具有 “全局” 变量的作用。

```
long InThread(TBinTreeNode *rt, TBinTreeNode **pre)
{//线索化以 rt 为根的树, 返回树结点个数
```

```

//在程序中, *pre 一直指向当前处理的结点的前驱。初始调用该函数时, 置*pre 为空,
//表示第一个结点无前驱。之所以将 pre 说明为指针的指针, 是为了能在程序中改变 pre 的值
long cnt=0;
if (rt==NULL) return 0;
cnt = InThread(rt->lc, pre); //中序线索化根的左子树
if (rt->lc==NULL)
{
    //rt 的左链空, 存入线索
    rt->lc = *pre; //令 rt 的左链存放 rt 的前驱
    rt->threadTag=0x10; //置左标志: 存放线索
}
else    rt->threadTag = 0x00; //置左标志: 存放左儿

if (*pre!=NULL) //第一次调用该函数时, *pre 为空, 表示 rt 是遍历序列中第一个结点, 无前驱
    if (*pre->rc==NULL)
    {
        //线索化*pre 的右链
        *pre->rc=rt; //令 rt 的左链存放 rt 的前驱
        *pre->threadTag=*pre->threadTag | 0x01; //置 pre 的右标志: 存放 pre 的后继
        //由于该标志已在前面设置过左标志, 故为了不破坏其, 应使用“或”
    }
    else *pre->threadTag=*pre->threadTag | 0x00; //置 pre 的右标志: 存放 pre 的右儿

*pre = rt; //rt 已处理, 接下来的操作, 当前结点就是 rt 的后继, 故*pre 应置为 rt
cnt = cnt + InThread(rt->rc, pre); //中序线索化根的右子树

return cnt+1; //返回处理过的结点数目
}

```

上面的函数, 由于参数中设了一个在逻辑上并不需要 (只在实现中需要) 的参数 pre, 使得函数调用不自然。为解决该问题, 我们再定义一个函数, 它是上面函数的“包装”, 但参数中去掉了只在实现中使用的部分。该函数如下:

```

long InThread(TBinTreeNodeThread *rt)
{
    TBinTreeNodeThread *pre=NULL;
    return InThread(rt, &pre);
}

```

至于其他遍历方式下的线索化, 方法类似, 这里就不作介绍, 留作练习。

## § 6.8 树与森林

前面已给出树与森林的概念，但只是重点介绍了二叉树，本节开始讨论树与森林操作、存储及其与二叉树的转化等问题。

### § 6.8.1 树与森林的遍历

#### (一) 树的遍历

由于树不象二叉树那样，根位于二棵子树的中间，故一般无“中序”一说。树一般只有先根和后根及层序三种遍历方法。关于层序遍历，与二叉树相同。

##### 1. 先根遍历

先根遍历以 root 为根的树（子树）的规则为：

- a) 若 root 为空，则无动作，结束；
- b) 若 root 不空，则先访问根结点，然后按此规则（先根），按各子树的次序（若为无序树，则次序任意）分别遍历 root 的各子树。

##### 2. 后根遍历

后根遍历以 root 为根的树（子树）的规则为：

- a) 若 root 为空，则无动作，结束；
- b) 若 root 不空，则按后根遍历规则，按各子树的次序（若为无序树，则次序任意）分别遍历 root 的各子树，然后访问根。

由上面的定义知，若为无序树，则各子树的访问/遍历次序是随意的，因此，无序树的先根和后根遍历结果不唯一。但在一般情况下，我们假定按各子树的（在图中，或在存储结构中，或在形式定义中）出现次序遍历。

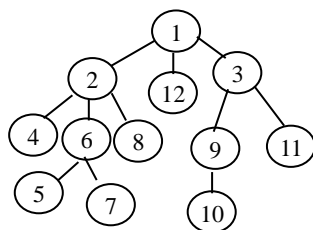


图 6-17 一棵 3 叉树

**例 6-6** 对图 6-0 所示树的遍历结果为：

先根： 1 2 4 6 5 7 8 12 3 9 10 11

后根： 4 5 7 6 8 2 12 10 9 11 3 1

#### (二) 森林的遍历

森林可有三种遍历方式：前序遍历、后序遍历及层序遍历。层序遍历与二叉树相同。


## 1. 前序遍历

- a) 若森林为空，则无动作，返回。
- b) 若森林非空，则
  - 先访问森林中的第 1 棵子树的根结点；
  - 前序遍历第 1 棵子树的根的各子树构成的森林；
  - 前序遍历除去第 1 棵子树后剩余子树构成的森林。

## 2. 后序遍历

- a) 若森林为空，则无动作，返回。
- b) 若森林不空，则
  - 后序遍历森林中第 1 棵子树的根结点的各子树构成的森林。
  - 访问第 1 棵子树的根。
  - 后序遍历除去第 1 棵子树后剩余子树构成的森林。

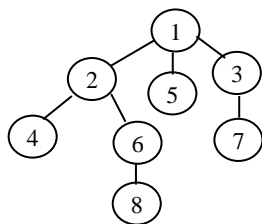
在上面的定义中，对各子树的遍历次序，我们仍然规定，若为有序树，则按子树的次序遍历，否则遍历子树的次序任意。

 一些文献将我们这里定义的森林的后序遍历，称做“前序”或“中序”遍历。由于森林比较特殊，这几种称谓都有道理，但本教程中，按我们这里的称谓。

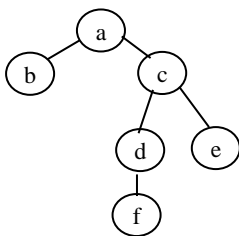
**例 6-7** 对图 6-0 所示森林的遍历结果为：

前序：1 2 4 6 8 5 3 7   a b c d f e   g h i k j

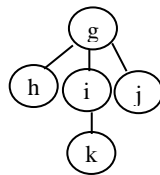
后序：4 8 6 2 5 7 3 1   b f d e c a   h k i j g



(a) 树 a



(b) 树 b



(c) 树 c

图 6-18 一个森林

## § 6.8.2 树/森林与二叉树之间的转化

### (一) 基本转化方法

与树相比，二叉树的存储与操作实现要简单一些。若能将树确定地转化为二叉树，并可确定地将转化后的结果还原为原树，则对树的处理就转化为对二叉树的处理了。讨论树与二叉树间的转化，主要目的就在于此。

这里先考虑将树转化为二叉树。其基本思想是，对每一树结点，都将它转化为一个二叉树结点，它的第 1 个儿子作为它在二叉树中的左儿子，而将它的第二个儿子做为第一个儿子的右儿子，第三个儿子做为第二个儿子的右儿子，…，余类推。图 6-0 给出了这种转化方法的几个例子。

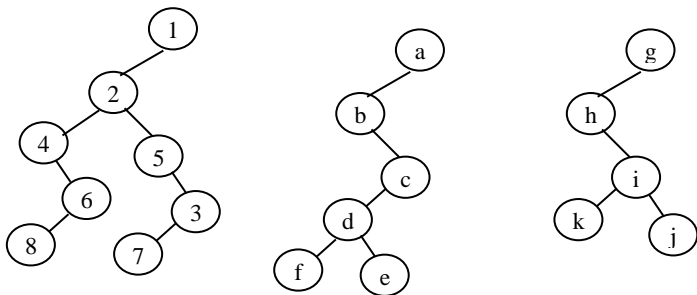


图 6-19 图 6-18 所示的三棵树的二叉树转化形式

从转化方法看，转化后的二叉树的根肯定无右儿子，而且，树中的叶子在二叉树中无左儿子，若其（树中的叶子）在树中无兄弟，或有兄弟，但排行最后（兄弟），则转化后仍为叶子。

将二叉树复原为树也是简单的。具体方法是，对任一个二叉树结点，将它的左儿子作为第 1 个儿子，而将左儿子的右分枝上各结点依次作为它的第 2、第 3，…第  $n$  个儿子。

可以将上述转化方法推广到森林。设森林中有  $n$  棵树 ( $n > 1$ )，先按上述方法分别将这  $n$  棵树转化为  $n$  棵二叉树，然后，将这  $n$  棵二叉树的根相连即可。根相连的方法是，将第  $k$  棵二叉树作为第  $(k-1)$  棵二叉树的根的右子树， $k=2, 3, \dots, n$

### (二) 转化方法的形式描述

下面给出二叉树与森林之间的转化方法的严格描述。由于树是森林的特例，所以这里的描述包含了树的情况。

#### 1. 森林转化为二叉树

设  $F=\{T_1, T_2, \dots, T_m\}$  是森林，则可按如下规则将其转化为一棵二叉树  $B=(\text{root}, \text{LB}, \text{RB})$ ，其中， $\text{root}$  为根， $\text{LB}$  和  $\text{RB}$  分别为左右子树：

- 若  $F$  为空, 即  $m=0$ , 则  $B$  为空树;
- 若  $F$  非空, 则  $B$  的根  $root$  即为  $F$  中第 1 棵子树的根,  $B$  的左子树  $LB$  是由森林  $F_1=\{T_{11}, T_{12}, \dots, T_{1k}\}$  转化来的二叉树, 这里  $F_1$  是由  $T_1$  的各子树构成的森林; 而  $B$  的右子树  $RB$  是由森林  $F_2=\{T_2, T_3, \dots, T_m\}$  转化来的二叉树。

## 2. 二叉树转化为森林

设  $B=(root, LB, RB)$  是一棵由森林转化来的二叉树, 则可按如下规则将它还原为森林  $F=\{T_1, T_2, \dots, T_m\}$ 。

- 若  $B$  为空, 则  $F$  为空。
- 若  $B$  非空, 则  $F$  中第 1 棵树  $T_1$  的根即为  $B$  的根  $root$ ;  $T_1$  中根结点的子树森林  $F_1$  是由  $B$  的左子树  $LB$  转化而成的森林;  $F$  中除  $T_1$  之外的其他树组成的森林  $F'=\{T_2, T_3, \dots, T_m\}$  是由  $B$  的右子树  $RB$  转化而成的森林。

关于转化的实现, 在选定了存储结构后可按上列规则写出相应的算法。

显然, 树的遍历结果与它对应的二叉树 (由树转化成的二叉树) 有如下对应关系:

树的先根遍历  $\iff$  转化二叉树的前序遍历

树的后根遍历  $\iff$  转化二叉树的中序遍历

另外, 森林的前序与后序遍历分别与它对应的二叉树的前序和中序遍历结果相同。

## § 6.8.3 树的存储结构

根据上面的讨论, 树与二叉树有着一一对应的转化方式, 所以, 可以按二叉树的存储方式存储树。在许多情况下, 我们都可以这样处理。但是, 这样处理也存在一些问题。例如, 由于将树转化为二叉树后, 原来的父子关系就不对应二叉树中的父子关系了, 所以, 在二叉树中识别原树中的父子关系就比较困难、费时。其次, 转化成的二叉树一般空链域较多, 存储利用率较低。所以, 有时需要直接存储树。但是, 树的特点是结点的子树的个数变化范围较大, 所以给存储结构带来了复杂性。这里就介绍几种常见的方法。

### (一) 双亲表示法 (一指针法)

与二叉树的一指针法相同, 即每个树结点设置指向父结点的指针。

这种存储法的特点是与结点结构和子树数目无关。由某结点出发, 容易找到它的祖先, 但不能搜索到后代, 即只表示了前趋关系, 而无后继关系。而且在由前趋关系推导后继关系时, 也不是完备的 (无法区分子树的次序)。

## (二) 多指针式

与二叉树的二指针式类似，即每个结点设立  $n$  个指向儿子的指针， $n$  为各结点的最大子树个数。

这种方式虽然是完备的，但由于结点子树个数变化范围大，所以会造成很大的存储浪费。

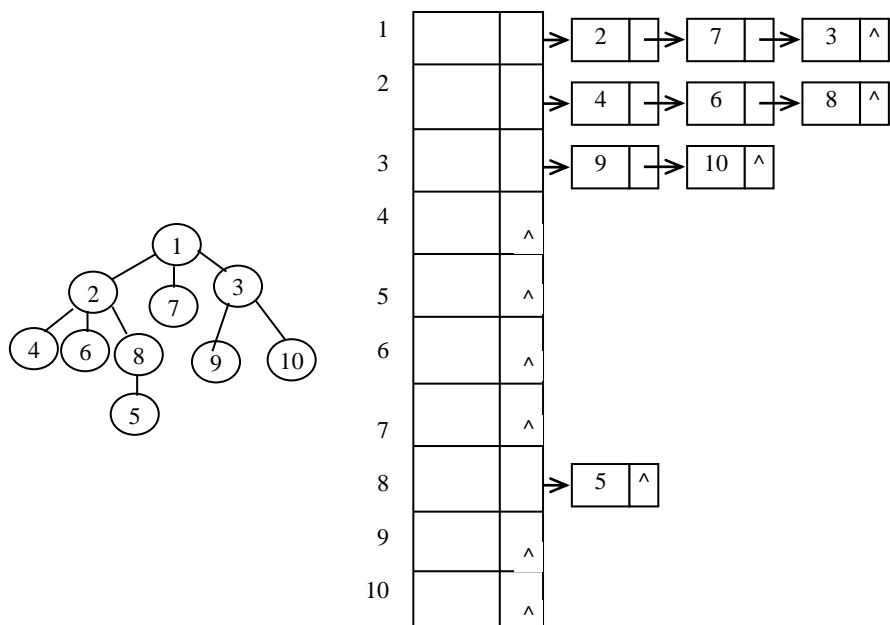


图 6-20 树的邻接表存储结构示例

## (三) 邻接表法

### 1. 存储方法

这是一种用多个单链表表示树的方法，具体存储方式为：

- 为每个树元素结点分别设一个结点（称为“树结点”）：其结构为：

元素内容	链头指针
------	------

其中，“元素内容”存放树元素结点的内容；“链头指针”指向该结点对应的“兄弟链表”的首结点。

当然，在该结点内，还可设立有关树结点的其他信息，如父指针，儿子个数等。

• 将各“树结点”集中放在一个一维数组中，该数组称为邻接表的“头数组”，是邻接表的代表。也可以不采用数组，而是在“树结点”中设立链指针，使各树结点链为一条链（称为“结点链”）。



• 对每个树结点（记为  $p$ ），设一个单链表（也称儿子链表，其中的每个链结点分别代表从  $p$  出发的一条边（即一个二元关系， $p$  是关系的前件，记它的各后件分别为  $p_i, i=1, \dots, n$ ，而  $n$  为  $p$  的儿子个数），称其为“边结点”，其结构为：

元素索引	链指针
------	-----

其中，“元素索引”用于存放该边结点对应的儿子结点（记其为  $pc$ ，它的父亲是  $p$ ）的位置信息。对头数组的情况（ $p$  的儿子结点为数组元素），“元素索引”为  $pc$  在“头数组”中的下标。通过该索引，可在头数组中找到对应“树结点”的位置。“链指针”为指向单链表中下个结点的指针。若不采用数组（采用“结点链”），则“元素索引”为一指针，指向“结点链”中对应结点。

显然，链在同一链中的各链结点代表的边所对应的关系的后件（树结点），是兄弟关系。也就是讲，对任一树结点，它的各儿子均逻辑上在它对应的单链表中，而同一单链表中各链结点对应的关系后件互为兄弟。

图 6-0 给出了这种存储法的一个例子。

## 2. 高级语言描述

下面考虑这种存储结构的描述（数据部分）。

该结构有两种结点：树结点和链结点（边结点），只需对它们进行描述。

现假定各树结点组成一个数组，则链结点的 C/C++ 描述为：

```
struct TTreeLinkNode
{
    long nodeId; //树元素索引
    TTreeLinkNode *next; //链指针
};
```

树结点的 C++ 描述为：

```
template <class TElem>
struct TTreeNodeAdj
{
    TElem info; //树元素内容
    TTreeLinkNode *firstSon; //链头指针，实质上指向它与它的第一个儿子对应的边
    long fatherId; //父亲索引，即父亲元素结点在数组中的位置
    long numSons; //儿子数目
};
```

#### (四) 孩子兄弟法

##### 1. 存储方法

又称**二叉链表法**。这种方法与邻接表法有些类似类似。具体地，孩子兄弟法可陈述为：

- 对树中每一结点，设立一个如下结构的结点：

元素内容	大儿子指针	下个兄弟指针
------	-------	--------

- 对树中每一结点，设立一个单链表，将它的各儿子结点通过“下个兄弟指针”链在一起，并用它的“大儿子指针”指向该链的首结点。

易知，这种方式实质上是二叉树的二指针方式，其中“大儿子指针”和“下个兄弟指针”分别相当于左儿指针和右儿指针，而且，若先将树按前面的方法转化为二叉树，再按二叉树的二指针方式存储其，则存储结构就是这里的孩子兄弟存储结构。

图 6-0 给出了图 6-0 中的树的孩子兄弟存储结构

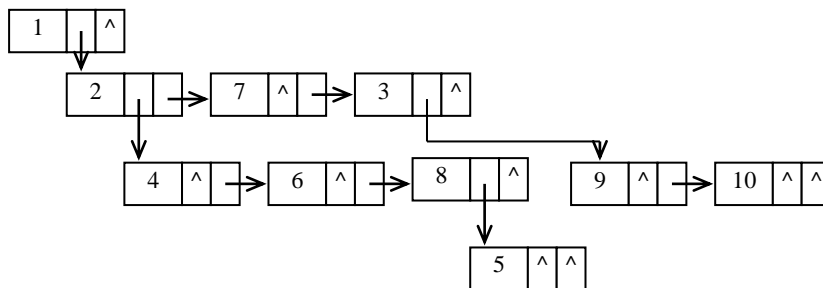


图 6-21 图 6-20 所示树的孩子兄弟表示法

##### 2. 高级语言描述

孩子兄弟链中只含一种结点，从纯结构（不考虑面向对象）的 C++描述为：

```

template <class TElem>
struct TTreeNodeBinLink
{
    TElem info; //元素内容
    TTreeNodeBinLink *firstSon; //大儿子指针
    TTreeNodeBinLink *nextBrother; //下个兄弟指针
    TTreeNodeBinLink *father; //父亲指针（可不设立）
};
  
```

除了上面介绍的几种存储方式外，广义表的存储方法也可用于树，这是因为树总可以

表示为广义表。

## § 6.9 树对象模型\*

与二叉树类似，我们将树看作一个独立的对象，给出它的属性和方法（基本操作）接口描述。树对象也主要涉及两个类：树结点类 `TTreeNode0` 和树类 `TTree`。下面分别介绍。

### § 6.9.1 树结点对象

#### (一) 抽象结点

树结点类 `TTreeNode0` 是对树的元素结点的抽象，它规定了针对结点本身的基本操作接口，使各种具体的结点结构，都可从它派生。在 C++ 中，用纯虚类表示。

在树抽象结点中，通过类属（参数化类型）定义元素内容。对关系，由于考虑到通用性，我们这里只给出它们的访问接口，具体的结构要根据具体的存储方式确定。

这里的关于树结点的抽象操作的设置，同样采用最小化的原则，即只设置在逻辑上针对结点本身的内容和关系的操作，对涉及全局的操作，放到“树类”中。

下面是树元素结点的抽象结构的定义。其中各成分从其名称可自明。

```
template <class TElem>
class TTreeNode0
{
public:
    TElem info; //树元素内容

    virtual TElem& GetElem(){return info;};
    virtual TTreeNode0* SetElem(TElem *e){info=*e; return this;};
    virtual TTreeNode0* GetSon(char sonNo)=0;
    virtual TTreeNode0* SetSon(char sonNo, TTreeNode0* pSon)=0;

    virtual TTreeNode0* GetFirstSon(void)=0;
    virtual TTreeNode0* GetNextSon(void)=0;

    virtual TTreeNode0* GetFather()=0;
    virtual TTreeNode0* SetFather(char sonNo, TTreeNode0* f)=0;

    char IsLeaf() {return GetSon(1)==NULL && GetSon(2)==NULL;};
```

```
};
```

事实上，该类形式上与二叉树类很类似，但其针对具体的存储结构的实现有较大区别，特别是 `GetSon(sonNO)` 操作，由于一般不可能在每个结点中设立完全的儿子信息，所以，它的实现一般需要进行查找。

另外，在该类中增加了下列操作：

**GetFirstSon(void):** 返回本结点的第一个儿子。

**GetNextSon(void):** 返回本结点的下一个儿子。该操作的“下一个”针对最近一次的 `GetFirstSon()`（当最近未执行 `GetNextSon()` 时），或最近一次的 `GetNextSon()`（当最近执行的是 `GetNextSon()` 时）。

设立这两个操作主要是为了支持快速地依次访问各个儿子。

## (二) 具体存储结构下的树结点类

上节介绍的具体存储中的元素结点，如邻接表中的 `TTreeNodeAdj`，孩子兄弟链中的 `TTreeNodeBinLink`，都应该是这里的 `TTreeNode0` 的派生类。这些派生类的具体描述，这里就不给出了，留作练习。

## § 6.9.2 树类

为了方便对整棵树进行操作，设立树类 `TTree`。该类用于封装具体的树实体及其相关操作。

`TTree` 中包含的树实体是由一个类型为 `TTreeNode0` 的根指针 `root` 所指出的树结构。由于在 C++ 中，指向基类的指针，可以直接指向相应基类的的派生类，而且，我们已在树结点类中通过基本操作抽象隐蔽了结点的结构，所以，该树类对各种具体的树结点都是兼容的。下面是该类的 C++ 描述。

```
enum TTraverseMode {PreOrder, InOrder, PostOrder, LevelOrder};

template <class TElem>
class TTree //树类
{
public:
    TTreeNode0<TElem> *root; //指向该类所代表的树的根结点
    long numNodes; //树结点个数

    TTree();
    ~TTree();
```

```

virtual TTreeNode0<TElem>* GetRoot(){return root;};
virtual void SetRoot(TTreeNode0<TElem>* rt){root=rt;};

virtual long GetLevel(TTreeNode0<TElem>* pNode);
virtual long GetHeight(TTreeNode0<TElem>* pNode);
virtual long GetNumSubNodes(TTreeNode0<TElem>* pNode);
virtual long GetNumSubNodes(void);
virtual long GetNumLeaves(TTreeNode0<TElem>* pNode);

virtual long Cluster(TTreeNode0<TElem>* pNode,TElem **e,TTraverseMode tm);
virtual long Cluster(TTreeNode0<TElem>* pNode,
                    TTreeNode0<TElem> **pe,TTraverseMode tm);
virtual long Cluster2(TTreeNode0<TElem>* pNode,
                    TTreeNode0<TElem> **pe,TTraverseMode tm);
virtual long ClusterDescendants(TTreeNode0<TElem>* pNode,
                             TTreeNode0<TElem> **pe, TTraverseMode tm = PreOrder,
                             int startLevel=1, int endLevel=-1);
virtual long ClusterAncestors(TTreeNode0<TElem>* pNode,
                             TTreeNode0<TElem> **pe);
virtual long ClusterAncestors(TTreeNode0<TElem>* pNode,TElem **e);
virtual long ClusterAncestors(TElem &e, TTreeNode0<TElem>** pNodes);
virtual long ClusterJuniors(TTreeNode0<TElem>* pNode,
                           TTreeNode0<TElem> **pe,TTraverseMode travMode=PreOrder,
                           int startLevel=1, int endLevel=-1);
virtual long ClusterSeniors(TTreeNode0<TElem>* pNode,
                           TTreeNode0<TElem> **pe, TTraverseMode travMode=PreOrder,
                           int startLevel=1, int endLevel=-1);
virtual TTreeNode0<TElem> *DeleteSubTree(TTreeNode0<TElem>* pNode,
                                         char sonNo);

/*
virtual long ClusterDescendants(TTreeNode0<TElem>* pNode,
                             TElem **es,TTraverseMode tm = PreOrder,
                             int startLevel=1, int endLevel=-1);
virtual long ClusterDescendants(TElem &e ,TTreeNode0<TElem> **pe,
                             TTraverseMode tm = PreOrder,
                             int startLevel=1, int endLevel=-1);
virtual long ClusterDescendants(TElem &e ,TElem **es,

```

```

        TTraverseMode tm = PreOrder,
        int startLevel=1, int endLevel=-1);

virtual long ClusterJuniors(TTreeNode0<TElem>* pNode,
        TElem **es, TTraverseMode travMode=PreOrder,
        int startLevel=1, int endLevel=-1);
virtual long ClusterJuniors(TElem& e, TTreeNode0<TElem> **pe,
        TTraverseMode travMode=PreOrder,
        int startLevel=1, int endLevel=-1);
virtual long ClusterJuniors(TElem& e, TElem **es,
        TTraverseMode travMode=PreOrder,
        int startLevel=1, int endLevel=-1);
virtual long ClusterSeniors(TTreeNode0<TElem>* pNode,
        TElem **es, TTraverseMode travMode=PreOrder,
        int startLevel=1, int endLevel=-1);
virtual long ClusterSeniors(TElem& e, TElem **es,
        TTraverseMode travMode=PreOrder,
        int startLevel=1, int endLevel=-1);
virtual long ClusterSeniors(TElem& e, TTreeNode0<TElem> **pe,
        TTraverseMode travMode=PreOrder,
        int startLevel=1, int endLevel=-1);
virtual TTreeNode0<TElem> *Locate(TTreeNode0<TElem>* rt, TElem &e,
        TTraverseMode travMode=PreOrder, long sn=1);
*/
virtual long ReleaseSubs(TTreeNode0<TElem>* pNode);
virtual TTreeNode0<TElem> *Locate(TTreeNode0<TElem>* rt, TElem *e);
virtual void Print(TTreeNode0<TElem> *rt, char mode);
};

```

该类的成员函数形式与二叉树类类似，其含义也类似，这里就不具体解释了。有关操作的实现，这里也不作介绍。

## § 6.10 树的应用示例—哈夫曼树

前面介绍的树，都是抽象意义的，与树的具体语义无关。在具体应用时，树都有具体

的语义,即树结点的含义、父子关系的含义等,都有具体的意义。这里,我们以一种应用很广的二叉树——哈夫曼树为例,说明二叉树的一种具体应用。另外,在这个例子中,我们也介绍用静态存储空间存储树的方法。从算法设计角度看, Huffman 树的生成算法也是一种典型的贪心法,因此这也作为我们后面即将要展开介绍的算法设计的前序。

在应用方面,哈夫曼树可用于构造最优编码,用在信息传输、数据压缩等方面,此外,哈夫曼树还可用于优化分枝构造。

### § 6.10.1 哈夫曼树的基本概念

关于哈夫曼(Huffman)树的定义,涉及到通路、权等概念,有关通路,我们在前面已定义过。下面给出其他概念。

**权(Weight):** 权是赋予某个实体的一个量,是对实体的某个/些属性的数值化描述。在数据结构中,实体有结点(元素)和边(关系)两大类,所以对应有**结点权**和**边权**。具体边权或结点权代表什么意义,由具体情况决定。

**通路、通路长:** 定义同前。

**树通路长:** 从根到各个叶子结点的各条通路的长度之和。

**树加权通路长:** 从根到各个叶子结点的各条通路的长度,分别乘以各自叶子的权,然后相加,所得结果称为树加权通路长。设树有  $n$  个叶子,它们的权分别为  $w_1, w_2, \dots, w_n$ , 从根到它们的通路的长度分别为  $p_1, p_2, \dots, p_n$ , 则该树的加权通路长为:

$$\sum_{i=1}^n w_i p_i$$

例如,图 6-0 中的二叉树(叶子中的数值为权)的加权通路长为:

$$3*1 + 5*8 + 4*6 + 3*2 + 4*4 + 4*3 = 101$$

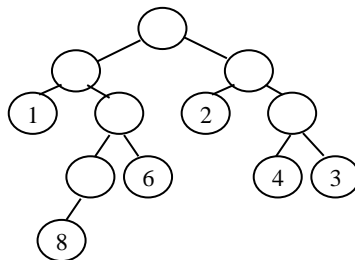


图 6-22 一棵完全二叉树

**定理 6-8** 给定  $n$  个结点,可以构造出无数多棵以这  $n$  个结点为叶子的二叉树。

该定理是显然的。但严格证明,要使用数学归纳法。首先,可用归纳法证明,给定  $n$  个结点,总可以构造出一棵以这  $n$  个结点为叶子的二叉树,然后,可以在这棵树的根上加一个结点,使该结点为原根的父亲,而该结点成为新的根,这样,原树加深了一层,但叶子数目没变,且仍然是原来的那  $n$  个叶子。这个过程可以无休止地进行,所以可以产生任意多棵不同二叉树(但叶子都相同)。

**哈夫曼树:** 给定  $n$  个值,那么,可以构造出任意多棵具有  $n$  个叶子且叶权分别为这  $n$  个给定值的二叉树(定理 6-8),而其中加权通路长最小的那棵,就称为哈夫曼树。哈夫曼树也称**最优二叉树**。因为这种树最早由哈夫曼(Huffman)研究,所以称为哈夫曼树。

**定理 6-9** 哈夫曼树是完全二叉树。

证：用反证法。设二叉树  $T$  是哈夫曼树，但不是完全二叉树，则  $T$  中至少有一个结点的度为 1(只有一个儿子)，记其为  $x$ 。那么，我们可以这样消去该结点  $x$ ：将该结点删除，将它的儿子（只有一个）转移给它的父亲，则  $T$  中从根到中间经过  $x$  到达各叶子的那些通路，其长度分别降低了 1，从而，删除  $x$  后，树  $T$  的加权通路长减小了，而叶子保持不变，这与原  $T$  是哈夫曼树的假设相矛盾。

该定理指出了哈夫曼树的一个必要条件。因此，我们在寻找（构造）哈夫曼树时，就可以只考虑完全二叉树。

**定理 6-10** 完全二叉树的结点总数为  $2n-1$ ，这里， $n$  为叶子总数。

证：设完全二叉树结点总数为  $m$ ，则  $m = n_2 + n$ ，这里  $n_2$  为度为 2 的结点数目。另一方面，每个非根结点恰有一个前驱（边），故有  $m = B + 1$ ，这里， $B$  为边的数目。又因为边是由度为 2 的结点射出的（后继），故有  $B = 2n_2$ ，结合得到的三个式子，即有  $m = 2n - 1$ 。

### § 6.10.2 哈夫曼树构造算法

这里，我们假定已知  $n$  个权值  $W = \{w_1, w_2, \dots, w_n\}$ ，构造一棵具有  $n$  个叶子且叶子权分别对应这  $n$  个值的哈夫曼树。具体步骤如下。

1.[初试化] 构造一个森林  $F = \{w_1, w_2, \dots, w_n\}$ ，它有  $n$  棵树，每棵树只有一个结点，且其权分别为给定的  $n$  个权值。

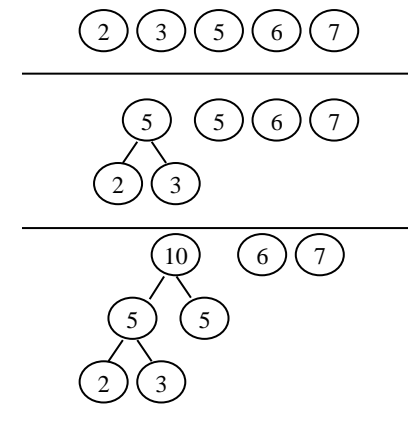
2.[找最小树] 从  $F$  中找两棵根权最小的树，分别以它们为左右子树，构造一棵新树，并令新树的根的权为这两棵根权之和。

3.[删除] 从  $F$  中将所找到的两棵最小树删除

4.[加入] 将新构造的树加入  $F$

5.[判断] 若  $F$  中只剩一棵树，则结束，其即为所求，否则，转 2。

**例 6-8** 设  $W = \{2, 3, 5, 6, 7\}$ ，则构造对应的哈夫曼树的过程见图 6-0。





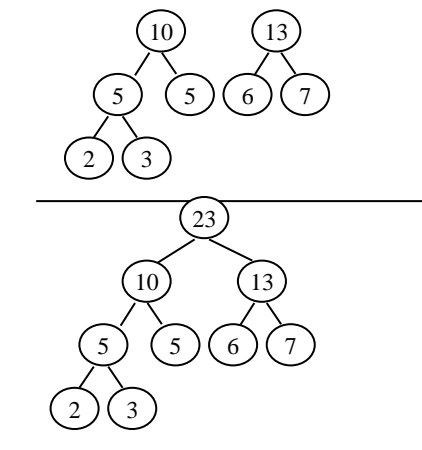


图 6-23 构造哈夫曼树

从上面的构造方法知，该构造过程一定是可终止的，因为在每趟处理中， $F$  都是删除两棵树，增加一棵树，即每趟后， $F$  中减少一棵树，经  $(n-1)$  趟后， $F$  就只剩一棵树了。这里， $n$  为权值个数。另外，直观上讲，先选择权较小的，所以权较小的结点，被放置在树的较深层，而权较大的，离根较近，这样，计算加权通路长时，自然会取得较小的值，这就是贪心法(Greedy)。关于该构造算法的正确性严格证明，Knoth 文献中已给出，这里就不介绍了。

### § 6.10.3 哈夫曼树构造算法的实现

这里，我们讨论前面给出的哈夫曼树构造算法的实现。

#### (一) 数据结构的存储考虑

哈夫曼树是一种二叉树，其当然可采用前面给出的存储方法。但构造哈夫曼树的目的一般是求得某种最优化的方案，如哈夫曼码、逻辑判断方案等。所以，数据结构存储方式的选取，一般不需顾及通用性，只求解决对应的问题。我们这里给出一种用一维数组存储哈夫曼树的方法。

该方法实质是树的三指针静态结构。每个结点的结构为

(权值，父亲序号，左儿子序号，右儿序号)

各结点存储在一维数组中。0 号位置不使用，从 1 号位置起存储。设哈夫曼树有  $n$  个叶子，则由前面的定理知，其结点总数为  $2n-1$ 。叶子集中存储在前面部分，即  $1 \sim n$  的位置，

而 $(n+1) \sim (2n-1)$ 存储其余结点。用 0 表示空指针。图 6-0 给出了上例（图 6-0）所建的哈夫曼树的这种存储结构。

序号	权	父亲	左儿	右儿
0				
1	2	6	0	0
2	3	6	0	0
3	5	7	0	0
4	6	8	0	0
5	7	8	0	0
6	5	7	1	2
7	10	9	3	6
8	13	9	4	5
9	23	0	7	8

图 6-24 哈夫曼树存储结构示例

## (二) 结构描述

这种存储结构下，树结点的 C++描述为：

```
struct THuffmanNode
{
    float weight; //权值
    long father, lc, rc; //父、左儿、右儿指示器
};
```

至于整棵哈夫曼树，是元素类型为 THuffmanNode 的一维数组。

构造哈夫曼树的已知条件是存放着各个权值的一个一维浮点数组，其定义形式为

```
float W[n+1]
```

这里  $n$  表示元素（叶子）个数。

## (三) 程序

我们设哈夫曼树构造子程序原型为

```
int HuffmanGen(float *W, int n, THuffmanNode *ht)
```

这里， $W$  存放为已知的  $n$  个权值， $ht$  为存放所生成的哈夫曼树的一维数组。

程序开始时，应先初始化  $ht$ ，使其  $1 \sim n$  号元素存放各个叶子（相当于初始森林）。此时，它们都没有父亲，也没有儿子。具体的初始化  $ht$  的程序片断为：

```
for (i=1; i<=n; i++)
```

```

{
    ht[i].weight = W[i]; //置叶权
    ht[i].father = 0; //将父指示器及左儿和右儿指示器置为“空”
    ht[i].lc = ht[i].rc = 0;
}

```

完成初始化后,就可以进入一个循环过程,依次生成(即填写)ht 的 $(n+1) \sim (2n-1)$ 号结点,也就是实施哈夫曼树构造算法中的步骤 2~5。但注意,这里并没有真正去删除结点,只是若某棵树被选中,则将其父亲置了非空值,相当于删除标记(事实上,哈夫曼树构造算法中。“删除”是“转让”,即将被删除的树作为其他树的子树)。

每次循环,所做的具体工作是,从当前森林中(即无父亲的结点中)查找两棵根权最小的树,将它们的根权的和存入 ht 的当前最前面的空闲元素中(相当于创建一个树结点),并置相应的儿子和父亲指示器。下面是具体的程序片段:

```

for (i=n+1; i<2*n; i++)
{
    //下面的函数从 ht[1]~ht[i-1]中找两个无父亲的权最小的结点,将其序号存入 s1 和 s2
    Selecte2Small(ht, i-1, &s1, &s2);
    //下面生成(填写)一棵树,以 ht[i]为根,以 ht[s1]和 ht[s2]为左右子树
    ht[i].weight = ht[s1].weight+ ht[s2].weight; //置根权
    ht[i].father= 0; //根无父亲
    ht[i].lc = s1; //置儿子指示器
    ht[i].rc = s2;

    ht[s1].father = i; //将当前找到的两棵最小树置为 ht[i]的儿子
    ht[s2].father = i;
};

```

这里用到一个子程序 Selecte2Small(), 下面是它的实现。

```

int Selecte2Small(THuffmanNode *ht, int k, int *s1, int *s2)
{ //从 ht[1]~ht[k]中找两个无父亲的权最小的结点,将其序号分别存入 s1 和 s2
    int i, j;

    if (k<2) return -1;
    *s1=0; *s2=0;
    for (j=1; j<=k; j++) //令*s1 和*s2 分别指向 ht 中最前两个无父亲的元素
    {
        if (ht[j].father != 0 ) continue;

```

```

    if (*s1==0) *s1=j;
    else { *s2=j; break; }
}
if (*s1==0 || *s2==0) return -1; //这种情况表示 ht 中最多有一棵树

if (ht[*s1].weight > ht[*s2].weight)
{ //令 s1 和 s2 分别持有 ht 中最前面两个无父亲的根权最小者中第一和第二小的下标
    i = *s1; *s1=*s2; *s2 = i;
}

for (i=j+1; i<=k; i++) //注意，这里的 j 是上面的循环中确定的 j
{
    if (ht[i].father != 0 ) continue;
    if (ht[i].weight < ht[*s1].weight ) // ht[i]为最小
    {
        *s2 = *s1;
        *s1 = i;
    }
    else
        if (ht[i].weight < ht[*s2].weight) *s2 = i; // ht[i]为第二小
} //for
return 0;
}

```

该程序开始时，首先找出 ht 中最前面的两个无父亲的结点，然后令\*s1 和\*s2 分别等于这两个结点中最小和最大者的下标。接下来从 j(第一个无父亲结点)的后面起，扫描 ht 中的元素，每扫描到一个元素，将其与当前两个最小 ht[\*s1].weight 和 ht[\*s2].weight 比较。若扫描到的元素 ht[i].weight 比 ht[\*s1].weight 小，则，ht[i].weight 作为新的最小者，而 ht[\*s1].weight 退居为新的第 2 小；若扫描到的元素 ht[i].weight 大于 ht[\*s1].weight 但小于 ht[\*s2].weight，则，ht[\*s1].weight 保持最小者地位，而 ht[i].weight 作为新的第二小；其他情况下\*s1 和\*s2 不变。

#### § 6.10.4 哈夫曼判定树

哈夫曼树是具有相同叶子个数的二叉树中“最小”的一棵，因此，凡涉及根据给定叶子（带权）求其“规模最小”的二叉树的问题，都可归结到哈夫曼树。哈夫曼逻辑判定树和哈夫曼编码是两种典型的应用，这里先介绍哈夫曼判定树，下小节介绍哈夫曼编码。

下面举例说明。

设某工厂的某种产品按某种测度（属性）分等级，如表 6-2 所表。

表 6-2 某种产品按某种测度（属性）的分等级及其出现

测度 $f$	0—20	21—40	51—60	71—90	91—100
等级 $g$	E	D	C	B	A
出现概率 $p(\%)$	2	5	13	70	10

表中“出现概率”是指对应等级的产品的出现概率。

在表 6-2 的基础上确定产品的等级，可通过多次逻辑条件判断进行，其可用高级语言的 if 语句多条嵌套描述，if 的判定过程相当于一棵二叉树。不同的判别方式，对应不同的二叉树。图 6-0 给出了对应两种不同的判定方式的两棵二叉树。图中，菱形结点表示条件判断，圆型结点（叶子）表示结论。结点的左分支表示判断结果为“真”，右分支表示“假”。

表面看上去，似乎图 6-0 左图对应的判定方式效率高（每个判断式都是简单的“小于

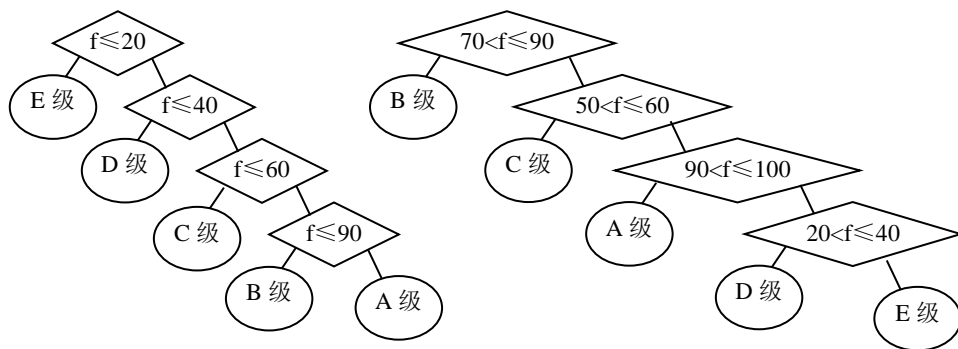


图 6-25 两棵不同的判定树

等于”判断），但分析一下每种等级的出现概率，E 级只有 2%，在实际中极少出现，而 B 级出现概率最大，因此，在右面的判定方式中，一次“命中”的机会很大，余下的分支很少需要判断，因此，右面的判定方式应该效率最高。

事实上，若把等级的出现概率看作叶子的权，则上述判定方式的选择，是个构造哈夫曼树的问题。图 6-0 中右图对应的是一棵哈夫曼树。

### § 6.10.5 哈夫曼编码与数据压缩

#### (一) 基本概念

目前，在用电子方式处理符号时，先对符号用二进制编码。例如，在计算机中使用的英文字符的 ASCII 编码就 8 位二进制编码，这是一种定长编码，即每个字符用相同数目的二进制位编码。

为了缩短数据文件（报文）长度，也采用不定长编码。其基本思想是，给使用频度较高的字符编以较短的编码。这是数据压缩技术的最基本的思想。

如何给数据文件中的字符编以不定长编码，使各种数据文件平均最短呢？这也是个哈夫曼树问题。先介绍几个概念。

**前缀码：**如果在一个编码系统中，任一个编码都不是另外其他任何编码的前缀（最左子串），则称该编码系统中的编码是前缀码。

例如，下面一组编码：

01, 001, 010, 100, 110

就不是前缀码，因为 01 是 010 的前缀。若去掉 01 或 010 则就是前缀码。

显然，若是前缀码，则在电文中，各字符对应的编码之间不需要分隔符。如果不是前缀码，则若不使用分隔符，会产生二义性。

**哈夫曼码：**对一棵具有  $n$  个叶子的哈夫曼树，若对树中的每个左分支赋予 0，右分支赋予 1，则从根到每个叶子的通路上，各分支的赋值分别构成一个二进制串，该种二进制串就称为哈夫曼码。

例如，设图 6-0 是一棵哈夫曼树，则各叶子的哈夫曼码为：

15: 00

8: 0100

2: 0101

10: 011

40: 10

25: 11

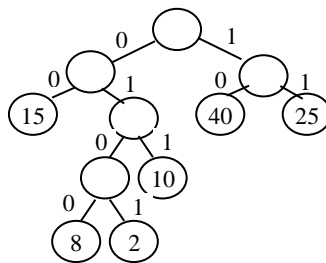


图 6-26 哈夫曼编码树

**定理 6-11** 哈夫曼码是前缀码。

证：哈夫曼码是根到叶子路径上的边的编码（0 或 1）的序列，也就等价边序列，而由树的特点知，若路径 A 是另一条路径 B 的最左部分，则 B 经过了 A，因此，A 的终点不是叶子。而哈夫曼编码都对应终点为叶子的路径，所以，任一哈夫曼码都不会与任意其他哈夫曼码的前部分完全重叠，因此哈夫曼码是前缀码。


**定理 6-12** 哈夫曼码是最短前缀码，即，对于  $n$  个字符，分别以它们的使用频度为叶子权，构造哈夫曼树，则该树对应的哈夫曼码，能使各种报文（由这  $n$  种字符构成的文本）对应的二进制串平均最短。

证：由于哈夫曼码对应于叶权为各字符使用频度的哈夫曼树，因此，该树为加权（频度）通路最小的树，即  $\sum_{i=1}^n w_i p_i$  最小，这里， $w_i$  是第  $i$  个字符的使用频度，而  $p_i$  是第  $i$  个字符的编码长度，这正是度量报文的平均长度的式子。证毕。

由哈夫曼树的构造方法看，使用频度较高的字符，对应的编码较短。这也直观地说明本定理。

## (二) 数据文件的压缩/解压


显然，由于哈夫曼编码是最短前缀码，所以，对任一数据文件，若对其中的字符采用哈夫曼编码，则所得文件的长度，比采用任何其他编码得到的文件的长度都短。根据找个性质，我们可以实现文件压缩。

 使用哈夫曼编码压缩文件，是通过缩短编码的长度实现数据量的减少的。在实际中，往往是多种压缩策略的结合，才能取得很大的压缩率。例如，还可以对文件中多个连续重复的字符，只存储一个字符和它的个数。

压缩时，先分析文件中的各种字符的出现频度，然后以这些频度值为依据，为文件中每种字符生成哈夫曼编码，最后，将文件中各字符都换成对应的哈夫曼编码。

解压时，也是按上述方法先生成哈夫曼编码，然后将文件中各字符分别替换为原来的编码。

显然，为了能解压，在压缩时，应保存各种字符的原编码与哈夫曼编码的对照表。具体程序的实现，留作练习。

 具体编程实现压缩时要注意，字符编码的存储，要直接按二进制方式存储，例如，“0101”应当作 4 个二进制符号存储，占 4 个二进制位。

## 本章小结

树是非线性结构中最重要的一种。本章中我们介绍的是有根树，它是一种严格的层次结构：除根没有前驱外，每个元素有唯一的前驱，但后继个数不限（在有限数范围内）。二叉树是后继个数最多为 2 的树，是本章的重点。

二叉树可以采用一指针、二指针、三指针方式存储。对顺序二叉树，也可以采用顺序存储法（不存储关系）。对一般的树，由于结点的后继个数变化范围较大，所以一般都采用一指针、邻接表或二叉链表（孩子兄弟链）存储。

关于树的基本操作，重点是按父子（祖孙）关系访问（遍历）结点。这里，我们通过几种 `Cluster` 操作概括了这些操作。例如，`Cluster()` 遍历所有后代，`ClusterDescendants()` 遍历某个范围内的后代，`ClusterAncestors()` 遍历某结点的祖先，`ClusterSeniors()` 遍历某范围内的前辈（包括非直系），`ClusterJuniors()` 遍历某范围内的后辈（包括非直系），等等。

树的串行化也是非常重要的。如果要将内存中的树结构持久保存（也称持久化，即保存到外部设备上），则首先需要将内存中的树结构转化为/表示为某种表达式（串行化），然后将代表树的表达式保存到外部存储器。需要读到内存处理时，要将表达式还原为对应的树结构（逆串行化）。我们这里给出了几种典型的串行化算法。

有时候，需要将一般树转化为二叉树后进行处理。我们这里也给出了相应的转化方法。

## 习 题

注：下面各题中，如无特别说明，假定二叉树的存储结构为二指针式。

1. 设有一棵叉树，它的中序和前序遍历结果如下，请画出该二叉树。

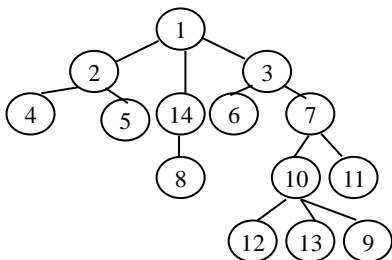
中序：1 4 3 5 6 2

后序：4 6 5 3 2 1

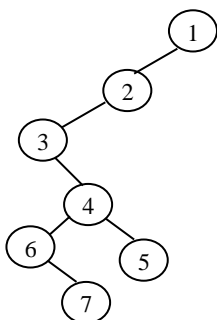
2. 请画出具有四个结点的各种不同构的有序二叉树。

3. 设一棵顺序二叉树具有 10 个结点，请计算其中叶子结点的数目。

4. 设有如下所示的一棵树，请将其转化为二叉树。



5. 设下列二叉树是有某棵树转化而来，请画出其对应的原树。



5. 请写出二指针存储结构的二叉树的层序遍历程序。

6. 设顺序二叉树按顺序存储结构存储（从上到下，从左到右的次序），请分别写出前序、中序、后序和层序遍历程序。



7. 编写程序, 判断二叉树是否是完全二叉树。
8. 编写程序, 判断二叉树是否是顺序二叉树。
9. 编写程序, 判断二叉树是否是高度平衡二叉树。
10. 编写非递归程序, 求出二叉树的结点总数。
11. 编写非递归程序, 求出二叉树的高度。
12. 编写非递归程序, 求出二叉树的叶子结点总数。
13. 编写程序 (非递归与递归), 复制二叉树。
14. 编写程序, 将二指针存储结构的二叉树修改为三指针存储结构 (设树结点中以留有父亲指针值)
15. 编写程序, 按各叶子的“叶子链”建立树结构。某叶子的“叶子链”, 是指从根到该叶子的路径。
16. 编写程序, 将树结构输出为 (各叶子的) “叶子链”。
17. 编写一个由二叉树生成前序扩展序列 (见§6.6.4) 的程序。
18. 编写前序和后序线索化的递归程序。
19. 编写前序和后序线索化的非递归程序。
20. 编写层序线索化程序。
21. 假定无序树用边集合表示, 请编写程序, 将其转化为邻接矩阵。这里边集合是指一个一维数组, 其每个元素表示树的一条边。
22. 设树用邻接表存储, 请编写程序, 将它转化为二叉树/
23. 设一组权值为{10, 20, 15, 32, 40, 60, 26, 18}, 请写出对应的哈夫曼树。
24. 假定二叉树按动态三指针存储, 编写哈夫曼树构造程序。
25. 编写由已知的哈夫曼树求其各叶子对应的哈夫曼编码的程序。
26. 编写一个基于哈夫曼树的数据文件压缩/解压程序。