第5章

树和二叉树

树状结构是一种典型的非线性逻辑结构，常用的树状结构包括树和二叉树。树状结构中数据元素之间为一对多的关系，也可以表示成层状关系。树状结构在现实生活中的应用十分广泛，例如族谱、计算机中的文件组织、部门结构等。

树和二叉树同属于树状结构。二叉树的子树若不为空，则严格区分左右。二叉树的物理结构包括顺序存储结构、二叉链表、三叉链表。其中，二叉链表的使用最为广泛。二叉树的操作主要有初始化、销毁、前序遍历、中序遍历、后序遍历和层次遍历等。

5.1 树的逻辑结构

5.1.1 树的基本术语

1. 树的定义

在树中一般把数据元素称为**结点**(node)。

树(tree)是 $n(n\geqslant 0)$个结点组成的有限集合。当 $n=0$ 时，称为**空树**。当 $n>0$ 时为非空树，需同时满足以下条件：

(1) 有且仅有一个特定的称为**根**(root)的结点；

(2) 当 $n>1$ 时，除了根结点之外，其余的结点可以划分成 $m(m\geqslant 1)$个互不相交的有限集合 $T_1,T_2,\cdots,T_m$，其中每个集合又是一棵树，并且称为根结点的**子树**(subtree)。

树是采用递归方式定义的。如图 5-1 所示，图 5-1(a)是一棵树，图 5-1(b)不是树，因为除了根之外的其余结点不能划分成互不相交的集合。

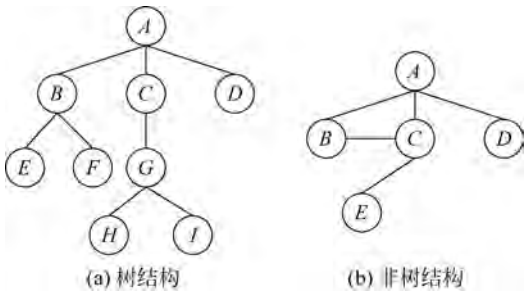


图 5-1　树结构和非树结构示例

2. 基本术语

1）结点的度、树的度

结点所具有的子树的棵数，称为结点的度(degree)。树中最大的结点的度称为树的度。例如图 5-1(a)所示的树，结点 B 的度为 2，结点 A 的度为 3，树的度为 3。

2）叶子结点、分支结点

度为 0 的结点称为叶子结点(leaf node)，也称为终端结点；度不为 0 的结点称为分支结点(branch node)，也称为非终端结点。图 5-1(a)所示的树中，结点 E、F、H、I、D 为叶子结点，其他结点为分支结点。

3）孩子结点、双亲结点

某结点子树的根结点称为该结点的孩子结点(children node)，该结点是孩子结点的双亲结点(parent node)。图 5-1(a)所示的树中，结点 B、C、D 是结点 A 的孩子结点，结点 A 是结点 B、C、D 的双亲结点。

4）兄弟结点、党兄弟结点

具有同一个双亲的结点互称为兄弟结点(brother node)，双亲互为兄弟的结点为堂兄弟结点。图 5-1(a)所示的树中，结点 B、C、D 为兄弟结点，结点 E 和 G 为堂兄弟结点。

5）路径、路径长度

如果树的结点序列 $a_1,a_2,\cdots,a_n$ 满足以下关系：结点 a_i 是结点 a_{i+1} 的双亲($1\leqslant i<n$)，即序列中相邻的两个结点，前一个是后一个的双亲，则把序列 $a_1,a_2,\cdots,a_n$ 称为结点 a_1 至结点 a_n 的路径(path)。路径上经过的分支数称为路径长度(path length)。树中任意两个结点之间如果存在路径，则路径是唯一的。图 5-1(a)所示的树中，从结点 A 到结点 I 的路径是 A、C、G、I，路径长度是 3。

6）祖先、子孙

如果从结点 x 到结点 y 存在路径，则结点 x 是结点 y 的祖先(ancestor)，结点 y 是结点 x 的子孙(descendant)。显然，一个结点的子树中所有的结点都是该结点的子孙，该结点是子树中所有结点的祖先。

7）结点的层数、树的深度(高度)

根结点所在的层为第 1 层，对其余结点，若结点在第 k 层上，则其孩子在第 $k+1$ 层上。树中所有结点的最大的层数为树的深度(depth)，也称为树的高度。图 5-1(a)所示的树的深

度为 4。

8）结点的层序编号

将树中的结点按照从上到下、从左到右的顺序从 1 开始采用连续的自然数编号，得到的编号称为结点的层序编号(level code)。图 5-1(a)所示的树中，结点 A 的层序编号为 1，结点 E 的层序编号为 5。

9）有序树、无序树

如果一棵树的子树从左到右没有顺序，交换子树的顺序，仍然是同一棵树，则称其为无序树(unordered tree)。如果一棵树的子树从左到右是有顺序的，交换子树的顺序，则和原树不再是同一棵树，则称其为有序树(ordered tree)。例如图 5-2 所示的两棵树，如果两棵树都是无序树，则两棵树是同一棵树。如果都是有序树，则不是同一棵树。“数据结构”课程中讨论的一般都是有序树。

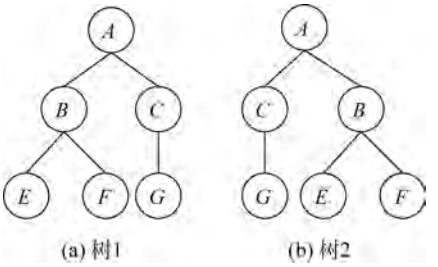


图 5-2　有序树和无序树示例

10）森林

$m(m\geqslant 0)$棵互不相交的树构成的集合称为森林(forest)。任何一棵非空树，根结点的子树可以看作一个森林。例如图 5-1(a)所示的树中，根结点 A 的 3 棵子树可以看作一个森林。

5.1.2　树的抽象数据类型定义

树的应用非常广泛，在不同的场合下的抽象数据类型不同，此处只给出最基本的操作。

树的抽象数据类型定义为：

ADT Tree{

数据对象：

$D=\{a_i \mid a_i \in \text{ElemSet}, i=1,2,\cdots,n, n\geqslant 0\}$

数据关系：

结点之间具有一对多的关系，根结点无双亲，叶子结点无孩子，其他结点只有一个双亲和多个孩子；

基本运算：

InitTree：初始化空树；

DestroyTree：销毁树；

PreOrder：先序遍历树；

PostOrder：后序遍历树；

LevelOrder：层序遍历树；

};

5.1.3　树的遍历

树的遍历指的是从根结点出发，将所有结点访问一次且仅访问一次。访问指的是对结点进行的某种抽象操作，此处简化为对结点数据的输出。树的遍历实际上是在非线性结构的基础上得到线性序列，因此访问次序尤其重要。按照访问次序，树的遍历分为前序遍历、后序遍历和层序遍历。

1. 前序遍历

树的前序遍历操作为：若树为空，则空操作返回，否则

(1) 访问根结点；

(2) 按照从左到右的顺序前序遍历根结点的每一棵子树。

例如图 5-1(a)所示的树，其前序遍历序列为 *A B E F C G H I D*，前序遍历序列中根在最前面，任意一个结点肯定出现在其子树结点的前面。

2. 后序遍历

树的后序遍历操作为：若树为空，则空操作返回，否则

(1) 按照从左到右的顺序后序遍历根结点的每一棵子树；

(2) 访问根结点。

例如图 5-1(a)所示的树，其后序遍历序列为 *E F B H I G C D A*，后序遍历序列中根在最后面，任意一个结点肯定出现在其子树结点的后面。

3. 层序遍历

树的层序遍历操作定义为按照结点从上到下、从左到右的次序访问结点，或者按照结点的层序编号访问结点。例如图 5-1(a)所示的树，其层序遍历序列为 *A B C D E F G H I*。

5.2　树的存储结构

树的存储结构需要存储结点的信息以及结点之间的逻辑关系，即结点之间的双亲与孩子的逻辑关系。

5.2.1　双亲表示法

树中的每个结点都有且仅有一个双亲结点，使用一维数组来表示树，数组的每一个元素是一个 PNode，对应树的结点，PNode 结点包括数据域 data 以及结点的双亲在一维数组

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

图 5-3　树的双亲表示法

中的下标 parent。结点在一维数组中按层序编号排序。PNode 定义如下。

```
template < class ElemType >
struct PNode{
    ElemType data;
    int parent;
};
```

例如图 5-1(a)所示的树，其对应的双亲表示法如图 5-3 所示。

在树的双亲表示法中，求已知结点的双亲非常方便，但是求结点的孩子则需要遍历一维数组。

5.2.2　孩子链表表示法

因为树中每个结点的孩子数目不确定，因此可以将每个结点所有的孩子使用单链表表示，将 n 个结点的孩子链表的头指针存储到一维数组中，n 个结点的数据域也存储到一维数组中，此一维数组称为表头数组，表头数组的每个元素为表头结点。则孩子结点和表头结点的结构如图 5-4 所示。

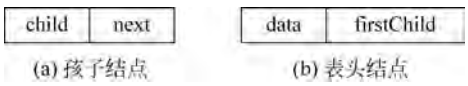


图 5-4　孩子链表表示法的结点结构

孩子结点和表头结点的结构定义如下。

```
struct CNode{
    int child;                  /* 孩子结点在表头数组中的下标 */
    CNode *next;                /* 双亲的下一个孩子 */
};
template < class ElemType >
struct HNode{
    ElemType data;              /* 数据域，存储结点的数据信息 */
    CNode *firstChild;          /* 表头结点的第一个孩子 */
};
```

例如图 5-1(a)所示的树，其孩子链表表示法如图 5-5 所示。

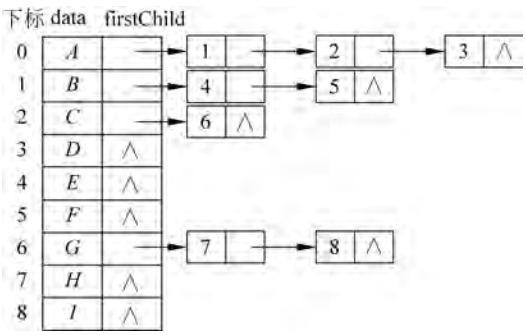


图 5-5　树的孩子链表表示法

5.2.3　孩子兄弟表示法

树的孩子兄弟表示法又称为二叉链表表示法，链表中的每个结点除了数据域外，还包括两个指针域，分别指向结点的第一个孩子和右兄弟，链表的结点结构如图 5-6 所示。

图 5-6　孩子兄弟表示法的结点结构

其中，data 为数据域，用于存储该结点的数据信息；firstChild 为指向第一个孩子结点的指针；rightSib 为指向右兄弟的指针。结点定义如下。

```
template < class ElemType >
struct CSNode{
    ElemType data;
    CSNode < ElemType > * firstChild, * rightSib;
};
```

图 5-1(a)所示的树的孩子兄弟表示法如图 5-7 所示。

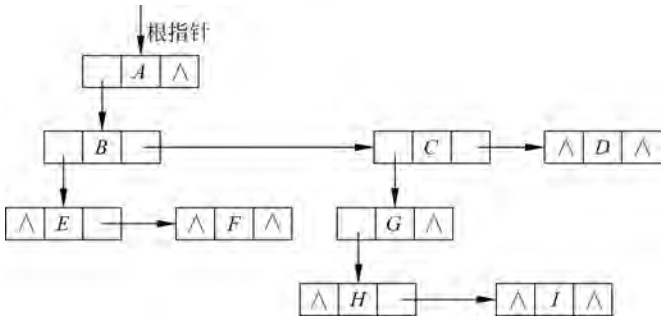


图 5-7　树的孩子兄弟表示法

5.3　二叉树的逻辑结构

二叉树(binary tree)是另一种重要的树状结构，因其结构相对简单，因此具有更广泛的应用。一般可以将树的问题转化成二叉树的问题进行处理。

5.3.1　二叉树的定义

二叉树采用递归方式定义。二叉树是 $n(n \geqslant 0)$ 个结点的有限集合。当 $n=0$ 时为空二叉树。当 $n>0$ 时，有且仅有一个称为根的结点；除了根结点之外，其余的结点可以分为互不相交的两部分，这两部分又分别是一棵二叉树，并且称为根的左子树和右子树。

二叉树和树是两种不同的树状结构。二叉树的度最大为2，当二叉树中某个结点只有一棵子树时，须严格区分是左子树还是右子树。二叉树的基本形态有五种，如图5-8所示。

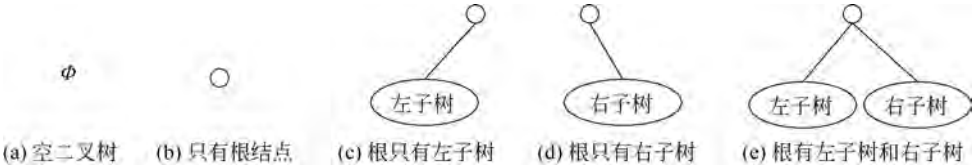


图5-8　二叉树的基本形态

斜树、满二叉树、完全二叉树是几种特殊的二叉树。

1. 斜树

斜树(oblique tree)分为**左斜树**(left oblique tree)和**右斜树**(right oblique tree)。左斜树是所有的分支结点都只有左子树的二叉树，右斜树是所有的分支结点都只有右子树的二叉树，例如，图5-9(a)为左斜树，图5-9(b)为右斜树。

斜树的每一层只有一个结点，因此斜树的结点个数与其深度相同。但是结点个数与其深度相同的二叉树不一定是斜树。

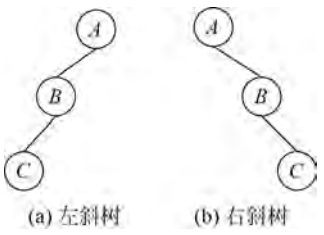


图5-9　斜树

2. 满二叉树

如果一棵二叉树的所有分支结点都存在左子树和右子树，并且所有的叶子结点都在同一层上，则此二叉树是**满二叉树**(full binary tree)。例如，图5-10(a)为满二叉树，图5-10(b)不是满二叉树，因为其叶子结点不在同一层上。

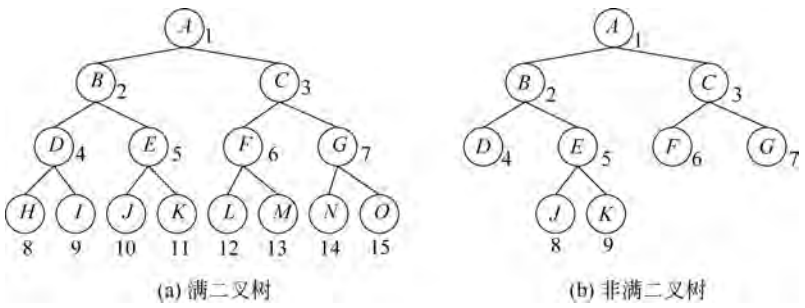


图5-10　满二叉树和非满二叉树

满二叉树的特点包括：

(1) 所有的叶子结点都在同一层；

(2) 只有度为0和度为2的结点。

3. 完全二叉树

对具有 n 个结点的二叉树进行层序编号，如果编号为 $i(1\leqslant i\leqslant n)$ 的结点与同样深度的

满二叉树中编号为 i 的结点的位置完全相同，则这棵二叉树为**完全二叉树**(complete binary tree)。例如，图 5-11(a)为完全二叉树，图 5-11(b)不是完全二叉树，因为 6 号结点的位置和满二叉树中的 6 号结点的位置不同。

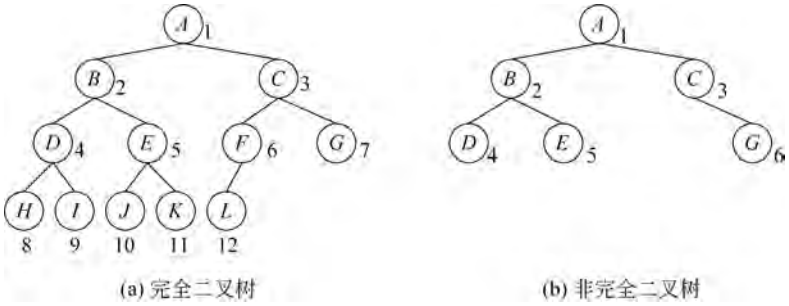


图 5-11　完全二叉树和非完全二叉树

显然，满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树。

完全二叉树的特点包括：

(1) 叶子结点只能出现在最下两层，并且最后一层的叶子集中出现在左侧位置。

(2) 最多有一个度为 1 的结点，而且该结点只有左孩子，没有右孩子。

完全二叉树也可以看作在满二叉树的最后一层从右至左连续去掉若干个结点。

5.3.2　二叉树的性质

性质 1　二叉树的第 $i(i>0)$层上最多有 2^{i-1} 个结点。

证明：可以采用数学归纳法证明。

当 $i=1$ 时，只有一个根结点，$2^{i-1}=2^0=1$，结论成立。

假设当 $i=k$ 时结论成立，即第 k 层上最多有 2^{k-1} 个结点，当第 k 层的每个结点都有左右两个孩子时，第 $k+1$ 层的结点数最多，即 $2^{k-1}\times 2=2^k$，则当 $i=k+1$ 时，结论也成立。因此，结论成立。

性质 2　深度为 $k(k>0)$的二叉树中，最多有 2^k-1 个结点，最少有 k 个结点。

证明：由性质 1 可知，二叉树的第 i 层上最多有 2^{i-1} 个结点，则深度为 k 的二叉树，只要每一层的结点数最多，则二叉树的结点总数最多。

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

深度为 k 的二叉树每层至少有一个结点，因此最少有 k 个结点。

性质 3　在一棵非空的二叉树中，如果叶子结点的个数为 n_0，度为 2 的结点的个数为 n_2，则 $n_0=n_2+1$。

证明：假设二叉树的结点总数为 n，度为 1 的结点总数为 n_1，因为二叉树中所有结点的度都小于等于 2，因此

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

假设二叉树的分支总数为 B，树状结构的分支总数等于各个结点的度数之和，因此

$$B=0\times n_0+n_1\times 1+n_2\times 2=n_1+2n_2 \tag{5-2}$$

又因为在非空的树状结构中，结点个数总是比分支个数多一个，即

$$n=B+1$$

由式(5-1)和式(5-2)可得：

$$n_0=n_2+1$$

例 5-1 具有 n 个结点的满二叉树的叶子结点和度为 2 的结点各有多少个？

因为满二叉树中没有度为 1 的结点，因此 $n_0+n_2=n$，根据二叉树的性质可知 $n_0=n_2+1$，因此 $n_0=\frac{n+1}{2}$，$n_2=\frac{n-1}{2}$。

性质 4 具有 n 个结点的完全二叉树的深度为 $\lfloor \mathrm{lb}n \rfloor+1$。

证明：假设具有 n 个结点的完全二叉树的深度为 k，完全二叉树的最后一层的结点个数最少是 1 个，最多是满二叉树的状态，如图 5-12 所示。

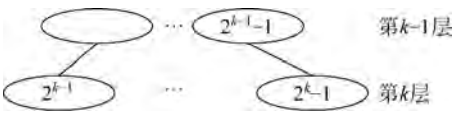


图 5-12 深度为 k 的完全二叉树的结点数范围

由图 5-12 可知下式成立：

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

对不等式取对数：

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

即

$$\mathrm{lb}n<k<\mathrm{lb}n+1$$

由于 k 取正整数，所以 $k=\lfloor \mathrm{lb}n \rfloor+1$。

性质 5 对一棵具有 n 个结点的完全二叉树从 1 开始进行层序编号，则对于任意的编号为 $i(1\leqslant i\leqslant n)$的结点(简称为结点 i)，有：

(1) 如果 $i>1$，则结点 i 的双亲的编号为 $\lfloor i/2 \rfloor$，否则 i 是根结点，无双亲。

(2) 如果 $2i\leqslant n$，则结点 i 的左孩子的编号为 $2i$，否则结点 i 无左孩子。

(3) 如果 $2i+1\leqslant n$，则结点 i 的右孩子的编号为 $2i+1$，否则结点 i 无右孩子。

证明：此性质可使用数学归纳法证明，感兴趣的读者可自行完成。

5.3.3 二叉树的抽象数据类型定义

相对于树来说，二叉树的应用更广泛，在不同的场合下的抽象数据类型定义不同，此处只给出二叉树最基本的操作。抽象数据类型定义为：

ADT BiTree{

数据对象：

$D=\{a_i \mid a_i\in \mathrm{ElemSet}, i=1,2,\cdots,n, n\geqslant 0\}$

数据关系：

结点之间具有一对多的关系，根结点无双亲，叶子结点无孩子，其他结点只有一个双亲，最多有左、右两个孩子；

基本运算：

```
        InitBiTree：初始化二叉树；
        DestroyBiTree：销毁二叉树；
        PreOrder：先序遍历二叉树；
        InOrder：中序遍历二叉树；
        PostOrder：后序遍历二叉树；
        LevelOrder：层序遍历二叉树；
};
```

5.3.4　二叉树的遍历

遍历是二叉树最基本的操作。二叉树的遍历指的是从根结点出发，将所有的结点访问一次并且仅访问一次。一棵二叉树可以划分为根、左子树、右子树三个部分。根据访问次序的不同可以将二叉树的遍历分为前序遍历、中序遍历、后序遍历和层序遍历。

1. 前序遍历

前序遍历二叉树的操作定义为：若二叉树为空，则空操作返回；否则

(1) 访问根结点。

(2) 前序访问根结点的左子树。

(3) 前序访问根结点的右子树。

显然，二叉树的前序遍历是一个递归的过程。例如图 5-13 所示的二叉树，其前序遍历序列为 $A\ B\ D\ G\ E\ C\ F$。任何一棵非空的二叉树的前序遍历序列的第一个结点是根结点。

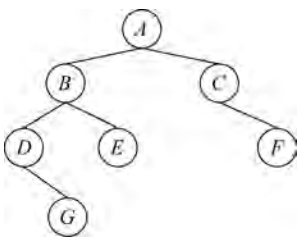


图 5-13　一棵二叉树

2. 中序遍历

中序遍历二叉树的操作定义为：若二叉树为空，则空操作返回；否则

(1) 中序遍历根结点的左子树。

(2) 访问根结点。

(3) 中序遍历根结点的右子树。

图 5-13 所示的二叉树的中序遍历序列为 $D\ G\ B\ E\ A\ C\ F$。

3. 后序遍历

后序遍历二叉树的操作定义为：若二叉树为空，则空操作返回；否则

(1) 后序遍历根结点的左子树。

(2) 后序遍历根结点的右子树。

(3) 访问根结点。

图 5-13 所示的二叉树的后序遍历序列为 $G\ D\ E\ B\ F\ C\ A$。任何一棵非空的二叉树的后序遍历序列的最后一个结点是根结点。

可见，在二叉树的前序、中序、后序遍历序列中叶子结点 G、E、F 的相对次序不变。

4. 层序遍历

层序遍历是从根结点开始，按照从上到下、从左到右的顺序遍历二叉树的结点，或者按照结点的层序编号遍历。图 5-13 所示的二叉树的层序遍历序列为 $A\ B\ C\ D\ E\ F\ G$。

5.4 二叉树的存储结构及实现

二叉树的存储结构同样需要存储两方面的内容，即二叉树的结点的数据信息以及各结点之间的逻辑关系。二叉树的存储结构包括顺序存储结构、二叉链表、三叉链表，其中二叉链表是最常用的存储结构，一般的二叉树的操作通常以二叉链表结构存储二叉树。

5.4.1 顺序存储结构

顺序存储结构指的是使用一维数组存储二叉树的结点。由于完全二叉树的结点的层序编号可以反映结点之间的逻辑关系，因此可以使用和完全二叉树的结点编号相对应的数组下标来存储结点信息，例如图 5-11(a)所示的完全二叉树可以使用图 5-14 所示的一维数组存储(数组的 0 号单元不使用)。

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

图 5-14　完全二叉树的顺序存储结构

对于一棵非完全二叉树，如果要使用顺序存储结构存储，则必须首先使用空结点将其补充成完全二叉树的形式，然后按照补充以后的结点的层序编号存储。例如图 5-15 所示的非完全二叉树补充成完全二叉树以后的形态，其对应的顺序存储结构如图 5-16 所示。

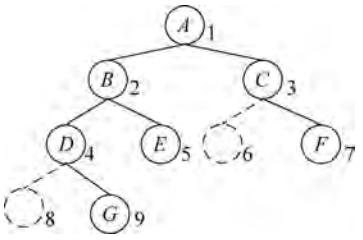


图 5-15　补充后的完全二叉树

1	2	3	4	5	6	7	8	9
A	B	C	D	E	∧	F	∧	G

图 5-16　二叉树的顺序存储结构

显然，补充的空结点会造成空间的浪费，浪费空间最严重的情况是右斜树。对于深度为 k 的右斜树，补充成完全二叉树的形态以后的结点数为 2^k-1 个，因此其顺序存储结构中的空数据单元的个数为 2^k-1-k 个。

5.4.2　二叉链表

二叉树一般采用二叉链表存储。对于二叉树的每一个结点，都分配一个二叉链表结点，结点包括数据域 data、左指针域 lchild、右指针域 rchild，其中，data 用来保存结点的数据信息，lchild 域指向结点的左孩子，rchild 域指向结点的右孩子。结点结构如图 5-17 所示。

图 5-13 所示的二叉树的二叉链表存储结构如图 5-18 所示。

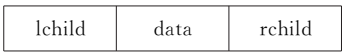


图 5-17　二叉链表的结点结构

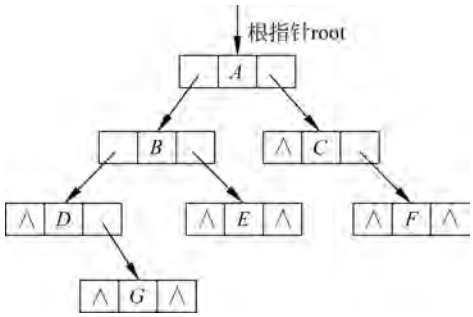


图 5-18　二叉链表存储结构

由图 5-18 可见，当二叉树的结点个数为 n 时，共分配 $2n$ 个指针，被占用的指针为 $n-1$ 个，还存在 $n+1$ 个空指针。

使用 C++语言描述二叉链表结点的结构定义如下。

```
template < class ElemType >
struct BiNode{
    ElemType data;
    BiNode < ElemType > * lchild, * rchild;
};
```

使用 C++语言中的类模板描述二叉链表如下。

```
template < class ElemType >
class BiTree{
public:
    / * 构造函数，建立一棵二叉树 * /
    BiTree() {
        root = Creat(root);
    }
    / * 析构函数，释放各结点 * /
    ~BiTree() {
        Release(root);
    }
    / * 前序遍历二叉树 * /
    void PreOrder() {
        PreOrder(root);
    }
```

```
    /*中序遍历二叉树*/
    void InOrder() {
        InOrder(root);
    }
    /*后序遍历二叉树*/
    void PostOrder() {
        PostOrder(root);
    }
    /*层序遍历二叉树*/
    void LevelOrder() {
        LevelOrder(root);
    };
private:
    BiNode<ElemType> *root;                              /*指向根结点的头指针*/
    BiNode<ElemType> *Creat(BiNode<ElemType> *bt);       /*构造函数调用*/
    void Release(BiNode<ElemType> *bt);                  /*析构函数调用*/
    void PreOrder(BiNode<ElemType> *bt);                 /*前序遍历函数调用*/
    void InOrder(BiNode<ElemType> *bt);                  /*中序遍历函数调用*/
    void PostOrder(BiNode<ElemType> *bt);                /*后序遍历函数调用*/
    void LevelOrder(BiNode<ElemType> *bt);               /*层序遍历函数调用*/
};
```

以上 BiTree 类模板中，为了避免外部对象直接访问其私有成员 root 指针，在公有的无参数方法中又调用了私有的带参数方法。

1. 前序遍历

前序遍历算法使用 C++语言描述如下。

```
template<class ElemType>
void BiTree<ElemType>::PreOrder(BiNode<ElemType> *bt) {
    if(bt == NULL)                                       /*递归调用的边界条件*/
        return;
    else {
        cout<<bt->data<<" ";                             /*访问根结点*/
        PreOrder(bt->lchild);                            /*前序递归遍历左子树*/
        PreOrder(bt->rchild);                            /*前序递归遍历右子树*/
    }
}
```

2. 中序遍历

中序遍历算法使用 C++语言描述如下。

```
template<class ElemType>
void BiTree<ElemType>::InOrder(BiNode<ElemType> *bt) {
    if(bt == NULL)
        return;                                          /*递归调用的边界条件*/
    else {
```

```
        InOrder(bt->lchild);                          /*中序递归遍历左子树*/
        cout<<bt->data<<" ";                          /*访问根结点*/
        InOrder(bt->rchild);                          /*中序递归遍历右子树*/
    }
}
```

3. 后序遍历

后序遍历算法使用C++语言描述如下。

```
template<class ElemType>
void BiTree<ElemType>::PostOrder(BiNode<ElemType> *bt) {
    if(bt == NULL)
        return;                                       /*递归调用的边界条件*/
    else {
        PostOrder(bt->lchild);                        /*后序递归遍历左子树*/
        PostOrder(bt->rchild);                        /*后序递归遍历右子树*/
        cout<<bt->data<<" ";                          /*访问根结点*/
    }
}
```

可见，二叉树递归的前序、中序、后序遍历的算法非常类似，区别主要在于访问二叉树的根、左子树、右子树的顺序不同。

4. 层序遍历

对二叉树进行层序遍历时，需要使用队列来保存结点指针，以便于寻找其左右孩子结点。使用伪代码描述二叉树的层序遍历算法如下。

1. 顺序队列Q初始化；
2. 如果二叉树不为空，则将根指针入队；
3. 当顺序队列Q不为空时循环：
   2.1 q＝队列Q的队头元素出队；
   2.2 访问q的数据域；
   2.3 若q存在左孩子，则将其左指针入队；
   2.4 若q存在右孩子，则将其右指针入队；

使用C++语言描述层序遍历算法如下。

```
template<class ElemType>
void BiTree<ElemType>::LevelOrder(BiNode<ElemType> *bt) {
    const int MaxSize = 100;
    /*采用顺序队列，假设不会发生溢出*/
    int front = -1, rear = -1;
    BiNode<ElemType> *Q[MaxSize], *q;
    if(bt == NULL)
```

```
            return;
        else {
            Q[rear++] = bt;                          /* bt 入队 */
            /* 队列非空时循环 */
            while(front != rear) {
                q = Q[front++];                      /* 队头出队 */
                cout << q->data <<" ";               /* 访问队头 */
                if(q->lchild != NULL)                /* 如果队头有左孩子,则左孩子入队 */
                    Q[rear++] = q->lchild;
                if(q->rchild != NULL)                /* 如果队头有右孩子,则右孩子入队 */
                    Q[rear++] = q->rchild;
            }
        }
}
```

5. 构造函数

要对二叉树进行遍历，必须首先在内存中创建一棵由二叉链表存储的二叉树。二叉树的一个遍历序列不能唯一地确定二叉树，例如前序遍历序列、中序遍历序列或者后序遍历序列。由二叉树得到扩展的二叉树，再由扩展的二叉树的前序遍历序列则可以唯一地确定二叉树。所谓扩展的二叉树指的是在原二叉树的末端添加虚结点，使原二叉树的所有结点的度都变为 2。扩展的结点的数据可使用不会出现在正常结点的数据表示，例如'#'。例如，图 5-19(a)为原二叉树，图 5-19(b)为扩展二叉树。扩展二叉树的前序遍历序列为 *A B # # C D # # #*。

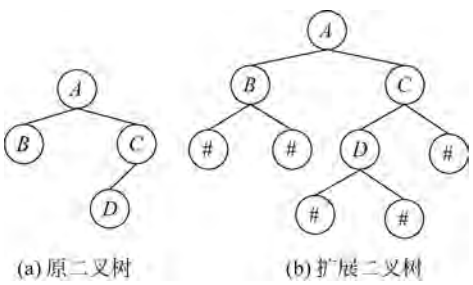


图 5-19　二叉树和扩展二叉树

使用扩展的二叉树的前序序列创建二叉链表是递归的过程。假设二叉树的结点数据类型为 char。首先输入根结点对应的字符，如果输入的是'#'，则二叉树为空二叉树，bt＝NULL；否则创建根结点，并为根结点的数据域赋值。然后递归地创建根结点的左子树和根结点的右子树。算法描述如下。

```
template <class ElemType>
BiNode<ElemType> *BiTree<ElemType>::Creat(BiNode<ElemType> *bt) {
    char ch;
    cout <<"请输入创建一棵二叉树的结点数据: "<< endl;
    cin >> ch;
    /* '#'代表空二叉树 */
    if(ch == '#')
        return NULL;
    else {
        bt = new BiNode<ElemType>;              /* 生成新结点 */
        bt->data = ch;
        bt->lchild = Creat(bt->lchild);         /* 递归创建左子树 */
        bt->rchild = Creat(bt->rchild);         /* 递归创建右子树 */
```

```
    }
    return bt;
}
```

6. 析构函数

可利用二叉树的后序遍历释放二叉链表的各结点。算法描述如下。

```
template <class ElemType>
void BiTree<ElemType>::Release(BiNode<ElemType> *bt) {
    /*按照后序遍历的顺序释放二叉树*/
    if(bt != NULL) {
        Release(bt->lchild);          /*释放左子树*/
        Release(bt->rchild);          /*释放右子树*/
        delete bt;                    /*删除根结点*/
    }
}
```

创建二叉链表及实现二叉树各种遍历的详细代码可参照 ch05\BiTree 目录下的文件。当创建的二叉树为图 5-19(a)时，扩展的前序遍历序列为 $AB\#\#CD\#\#\#$ 时(此处序列中没有空格)，运行结果如图 5-20 所示。

图 5-20　二叉链表实现的运行结果

5.4.3　三叉链表

在二叉链表中，可以从已知结点很方便地访问结点的左孩子或者右孩子，但是访问双亲结点不方便，需要通过二叉树的遍历完成。可在二叉链表结点的基础上再增加一个指向双亲结点的指针，即为三叉链表，三叉链表结点的结构定义描述如下。

```
template <class ElemType>
struct TriNode{
    ElemType data;
    TriNode<ElemType> *lchild, *rchild, *parent;
};
```

图 5-13 所示的二叉树的三叉链表存储结构如图 5-21 所示。

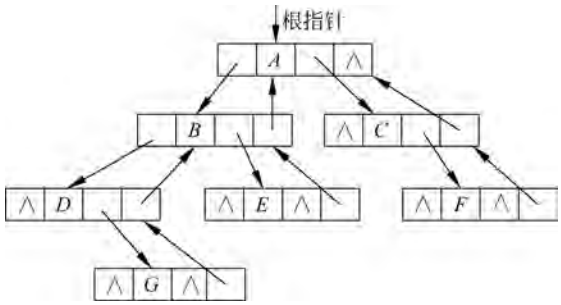


图 5-21　二叉树的三叉链表存储结构

5.5 二叉树的应用

5.5.1 非递归遍历二叉树

二叉树的递归遍历算法虽然简单，可读性好，但是递归的程序一般执行效率不高。因此，可采用非递归的方法遍历二叉树。

1. 前序非递归遍历算法

在非递归遍历算法中，关键的问题是当访问完当前结点，再访问完当前结点的左子树以后，如何再访问当前结点的右子树，因此，需要使用栈保存指向访问过的结点的指针。

假设当前指针为 bt，则按照 bt 是否为空可分为两种情况。

（1）当 bt ！= NULL 时，访问 bt-> data，bt 入栈。

（2）当 bt == NULL 时，需要判断栈的情况。如果栈为空，则遍历结束。如果栈不为空，说明当前栈顶指针的左子树为空或者栈顶指针的左子树已经访问完，栈顶出栈，继续访问其右子树。

例如图 5-19(a)所示的二叉树的前序非递归遍历的执行过程如表 5-1 所示。

表 5-1 前序非递归遍历的执行过程

步　骤	指针 bt	访 问 结 点	栈 S	说　明
1			空	初始化空栈 S
2	A	A	A	访问 A，A 入栈，找 A 的左子树
3	B	B	A B	访问 B，B 入栈，找 B 的左子树
4	NULL		A	B 出栈，找 B 的右子树
5	NULL		空	A 出栈，找 A 的右子树
6	C	C	C	访问 C，C 入栈，找 C 的左子树
7	D	D	C D	访问 D，D 入栈，找 D 的左子树
8	NULL		C	D 出栈，找 D 的右子树
9	NULL		空	C 出栈，找 C 的右子树
10	NULL		空	bt＝NULL 且栈为空，遍历结束

使用伪代码描述前序非递归遍历算法如下。

```
1. 栈 S 初始化；
2. 在 bt 不为空或栈 S 不为空时循环：
    2.1 当 bt 不为空时循环：
        2.1.1 输出 bt-> data；
```

```
        2.1.2 bt 入栈 S;
        2.1.3 继续遍历 bt 的左子树;
    2.2 如果栈 S 不为空:
        2.2.1 栈 S 栈顶出栈并赋值给 bt;
        2.2.2 遍历 bt 的右子树;
```

使用 C++语言描述的前序遍历非递归算法如下。

```
void BiTree::PreOrder(BiNode *bt) {
    SeqStack<BiNode *> S;
    while(bt != NULL || S.Empty() != 1) {
        while(bt != NULL) {
            cout << bt->data <<" ";
            S.Push(bt);
            bt = bt->lchild;
        }
        if(S.Empty() != 1) {
            bt = S.Pop();
            bt = bt->rchild;
        }
    }
}
```

此算法中使用了第 3 章的类模板 SeqStack。

2. 中序非递归遍历算法

中序非递归遍历二叉树与前序非递归遍历二叉树非常类似,区别仅在于当 bt 不为空时,在前序非递归遍历中,先访问结点,然后再入栈;而在中序非递归遍历算法中,当 bt 不为空时,先入栈,当栈顶的指针出栈时才进行访问,即:

(1) 当 bt != NULL 时,bt 入栈。

(2) 当 bt == NULL 时,如果栈为空,则遍历结束。如果栈不为空,则说明当前栈顶指针的左子树为空或者栈顶指针的左子树已经访问完,栈顶出栈,访问,并继续访问其右子树。

使用 C++语言描述中序非递归遍历算法如下。

```
void BiTree::InOrder(BiNode *bt) {
    SeqStack<BiNode *> S;
    while(bt != NULL || S.Empty() != 1) {
        while(bt != NULL) {
            S.Push(bt);
            bt = bt->lchild;
        }
        if(S.Empty() != 1) {
            bt = S.Pop();
            cout << bt->data <<" ";
            bt = bt->rchild;
        }
    }
}
```

3. 后序非递归遍历算法

后序非递归遍历算法和前序及中序非递归算法不同。在后序非递归遍历算法中，对于当前的非空指针 bt，只有从 bt 的右子树返回以后，才能对 bt 进行访问。

对于非空栈里的栈顶指针 ptr，在遍历的过程中会遇到两次当前指针 bt＝NULL。第一次 bt＝NULL 时，表示栈顶指针 ptr 的左子树为空，或者栈顶指针 ptr 的左子树已经访问完，即从 ptr 的左子树返回。第二次 bt＝NULL 时，表示栈顶指针 ptr 的右子树为空，或者栈顶指针 ptr 的右子树已经访问完，即从 ptr 的右子树返回。显然，只有当栈顶指针 ptr 第二次遇到空指针 bt 时，即从 ptr 的右子树返回时，才能使 ptr 出栈，并访问其数据域。

为了方便区分从左子树还是从右子树返回，需要对入栈的元素类型进行调整，除了 BiNode 类型的指针以外，另加标志域 flag，定义如下。

```
template<class ElemType>
struct Element{
    BiNode<ElemType> *ptr;
    int flag;
};
```

对于当前指针 bt，存在以下情况：

(1) 若 bt 不为空，则 bt 及 flag 为 1 入栈，继续访问 bt 的左子树。

(2) 若 bt 为空，则判断栈的情况。若栈为空，则遍历结束；若栈不空，则判断栈顶的 flag，如果栈顶的 flag 为 1，说明从栈顶的左子树返回，将栈顶的 flag 改为 2，继续访问栈顶的右子树；如果栈顶的 flag 为 2，则说明从栈顶的右子树返回，栈顶出栈并访问。图 5-19(a)所示的二叉树的后序非递归遍历的执行过程如表 5-2 所示，结点字符后的数字为其 flag 值。

表 5-2　后序非递归遍历的执行过程

步　骤	指针 bt	访 问 结 点	栈 S	说　明
1			空	初始化空栈 S
2	A		$A1$	将 A 带标志 1 入栈，找 A 的左子树
3	B		$A1B1$	将 B 带标志 1 入栈，找 B 的左子树
4	NULL		$A1\ B2$	将 B 的标志改为 2，找 B 的右子树
5	NULL	B	$A1$	B 出栈，访问 B
6	NULL		$A2$	将 A 的标志改为 2，找 A 的右子树
7	C		$A2\ C1$	将 C 带标志 1 入栈，找 C 的左子树
8	D		$A2\ C1\ D1$	将 D 带标志 1 入栈，找 D 的左子树
9	NULL		$A2\ C1\ D2$	将 D 的标志改为 2，找 D 的右子树
10	NULL	D	$A2\ C1$	D 出栈，访问 D
11	NULL		$A2\ C2$	将 C 的标志改为 2，找 C 的右子树
12	NULL	C	$A2$	C 出栈，访问 C
13	NULL	A	空	A 出栈，访问 A

使用伪代码描述后序非递归遍历算法如下。

```
1. 栈 S 初始化；
2. 在 bt 不为空或栈 S 不为空时循环：
   2.1 当 bt 不为空时循环：
       2.1.1 bt 及 flag＝1 入栈 S；
       2.1.2 继续遍历 bt 的左子树；
   2.2 在栈 S 不为空且栈顶的 flag ＝＝ 2 时循环：
       2.2.1 栈 S 栈顶出栈并赋值给 bt；
       2.2.2 访问 bt；
   2.3 若栈 S 不为空，则将栈顶的 flag 改为 2，并继续访问其右子树；
```

使用 C++语言描述算法如下。

```
void BiTree::PostOrder(BiNode *bt) {
    SeqStack<Element> S;
    Element e;
    /*bt 不为空或者栈不为空*/
    while(bt != NULL || S.Empty() == 0) {
        while(bt != NULL) {
            e.ptr = bt;
            e.flag = 1;
            S.Push(e);
            bt = bt->lchild;
        }
        /*栈不为空并且栈顶的 flag 为 2 时，出栈并访问*/
        while((S.Empty() == 0)&&(S.GetTop()).flag == 2) {
            e = S.Pop();
            cout<<e.ptr->data<<" ";
        }
        /*栈不为空，并且栈顶的 flag 为 1 时，将栈顶的 flag 更改为 2，并访问其右孩子*/
        if(S.Empty() == 0) {
            e = S.Pop();
            bt = e.ptr->rchild;
            e.flag = 2;
            S.Push(e);
        }
    }
}
```

二叉树非递归遍历的详细代码可参照 ch05\BiTreeNoReCur 目录下的文件，二叉树为图 5-19(a)所示的二叉树，非递归遍历的结果与递归遍历的结果相同。

5.5.2　二叉树遍历的应用

二叉树的一些常见应用是以二叉树的遍历为基础的，例如求二叉树的深度、求二叉树的结点个数、求二叉树的叶子结点个数、输出二叉树的叶子结点等，此类应用一般使用递归

算法解决。

1. 求二叉树的深度

当二叉树为空时，深度为0；当二叉树不为空时，二叉树的深度为其根的左右子树的深度中较大的值再加1。算法描述如下。

```
template<class ElemType>
int BiTree<ElemType>::Depth(BiNode<ElemType> *bt) {
    if(bt == NULL)
        return 0;
    else {
        int dep1 = Depth(bt->lchild);
        int dep2 = Depth(bt->rchild);
        return (dep1 > dep2) ? (dep1 + 1) : (dep2 + 1);
    }
}
```

2. 求二叉树的结点个数

求二叉树的结点个数时，可以利用二叉树的遍历，在遍历的过程中对结点进行计数。算法描述如下。

```
/*countNode为全局变量，初始化为0*/
template<class ElemType>
void BiTree<ElemType>::Count(BiNode<ElemType> *bt) {
    if(bt != NULL) {
        Count(bt->lchild);
        countNode++;
        Count(bt->rchild);
    }
}
```

3. 求二叉树的叶子结点个数

求二叉树的叶子结点个数与求二叉树的结点总数类似，只有结点为叶子时才计数。算法描述如下。

```
/*countLeaf为全局变量，初始化为0*/
template<class ElemType>
void BiTree<ElemType>::CountLeaf(BiNode<ElemType> *bt) {
    if(bt != NULL) {
        if(bt->lchild == NULL && bt->rchild == NULL)
            countLeaf++;
        CountLeaf(bt->lchild);
        CountLeaf(bt->rchild);
    }
}
```

4. 输出二叉树的叶子结点

算法描述如下。

```
template < class ElemType >
void BiTree < ElemType > ::PrintLeaf(BiNode < ElemType > * bt) {
    if(bt != NULL) {
        if(bt -> lchild == NULL && bt -> rchild == NULL) {
            cout << bt -> data <<" ";
        }
        PrintLeaf(bt -> lchild);
        PrintLeaf(bt -> rchild);
    }
}
```

二叉树的常见应用的详细代码可参照 ch05\BiTreeApp 目录下的文件，当构造如图 5-13 所示的二叉树时，二叉树的扩展前序序列为 $ABD\#G\#\#E\#\#C\#F\#\#$ 时，运行结果如图 5-22 所示。

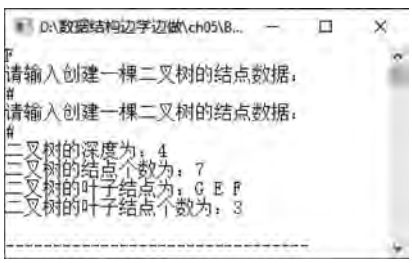


图 5-22　二叉树应用的运行结果

5.5.3　线索二叉树的构造和应用

在对二叉树进行各种操作时，有可能需要访问已知结点在某种遍历序列下的前驱结点或后继结点。含有 n 个结点的二叉树的二叉链表存储结构中仍然存在 $n+1$ 个空指针，可以利用这些空指针为访问结点的前驱或后继提供便利。如果结点没有左孩子，则左指针可以指向前驱结点；如果结点没有右孩子，则右指针可以指向后继结点。

指向前驱或后继的指针称为**线索**(thread)，将空指针更改为指向前驱或后继的过程称为**线索化**，加上线索的二叉树称为**线索二叉树**(thread binary tree)，加上线索的二叉链表称为**线索链表**(thread linked list)。

为了进一步区分指针是指向孩子还是指向前驱或后继，需要改造二叉链表的结点结构，在原有的基础上增加两个标志域 ltag 和 rtag，结构如图 5-23 所示。

ltag	lchild	data	rchild	rtag

图 5-23　线索链表的结点结构

其中，当 ltag＝0 时，lchild 指向左孩子；当 ltag＝1 时，lchild 指向前驱。当 rtag＝0 时，rchild 指向右孩子；当 rtag＝1 时，rchild 指向后继。线索链表中的结点使用 C++语言描述如下。

```
template < class ElemType >
struct ThrBiNode{
    ElemType data;
    int ltag, rtag;
    ThrNode < ElemType > * lchild, * rchild;
};
```

由于二叉树的遍历序列有四种，因此线索链表也有四种，分别是前序线索二叉链表、中序线索二叉链表、后序线索二叉链表以及层序线索二叉链表。例如图 5-13 所示的二叉树，其中序线索二叉链表如图 5-24 所示。

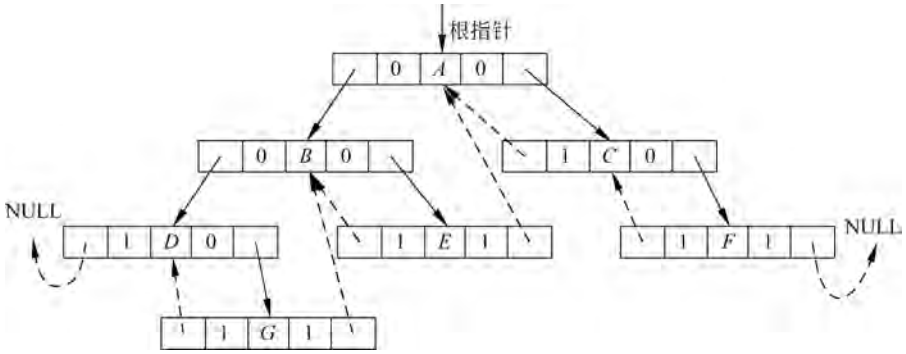


图 5-24　中序线索二叉链表

使用 C++的类模板描述中序线索二叉链表如下。

```
template < class ElemType >
class InThrBiTree{
public:
    InThrBiTree();                          / * 构造函数,建立一棵中序线索二叉树 * /
    ThrBiNode < ElemType > * Next(ThrBiNode < ElemType > * p); / * 在中序线索二叉树上查找 p 的后继 * /
    void InOrder();                         / * 在中序线索二叉树上中序遍历 * /
private:
    ThrBiNode < ElemType > * root;          / * 指向根结点的头指针 * /
    ThrBiNode < ElemType > * pre;           / * 当前根结点的前驱结点 * /
    ThrBiNode < ElemType > * Creat(ThrBiNode < ElemType > * bt); / * 构造函数调用 * /
    void ThrBiTree(ThrBiNode < ElemType > * bt);   / * 递归的中序线索化 * /
};
```

因为在二叉链表线索化的过程中要访问当前结点的前驱结点，因此为线索二叉树添加私有属性 pre 指针。

1. 构造初始的线索二叉链表

构造初始的线索二叉链表与构造初始的二叉链表相似，区别主要在于结点的左右标志域 ltag 和 rtag 赋初值为 0。算法描述如下。

```
template < class ElemType >
ThrBiNode < ElemType > * InThrBiTree < ElemType >::Creat(ThrBiNode < ElemType > * bt) {
    char ch;
    cout <<"请输入创建一棵二叉树的结点数据: "<< endl;
    cin >> ch;
    / * '#'代表空二叉树 * /
    if(ch == '#')
        return NULL;
    else {
```

```
        bt = new ThrBiNode<ElemType>;  /*生成新结点*/
        bt->data = ch;
        bt->ltag = 0;                  /*初始化标志域*/
        bt->rtag = 0;
        bt->lchild = Creat(bt->lchild); /*递归创建左子树*/
        bt->rchild = Creat(bt->rchild); /*递归创建右子树*/
    }
    return bt;
}
```

2. 中序线索化

中序线索化只能在遍历的过程中完成，因为只有在遍历的过程中，才能获取当前结点的前驱和后继信息。因为遍历是递归的过程，因此中序线索化也是递归的过程。对于当前的非空结点 bt，需要进行以下操作：

(1) 如果 bt 没有左孩子，则将其 ltag 改为 1，将 lchild 改为指向其前驱结点；

(2) 如果 bt 没有右孩子，则将其 rtag 改为 1；

(3) 如果结点 pre 的右标志为 1，则将 pre 的 rchild 指向 bt；

(4) 将 pre 更新为当前结点 bt。

使用伪代码描述中序线索化二叉链表的算法如下。

```
1. 如果二叉链表为空，则空操作返回；
2. 对 bt 的左子树建立线索；
   2.1 如果 bt 没有左孩子，则 bt-> ltag=1，bt-> lchild=pre；
   2.2 如果 bt 没有右孩子，则 bt-> rtag=1；
   2.3 如果 pre-> rtag=1，则 pre-> rchild=bt；
   2.4 pre=bt；
3. 对 bt 的右子树建立线索；
```

使用 C++语言描述算法如下。

```
template<class ElemType>
void InThrBiTree<ElemType>::ThrBiTree(ThrBiNode<ElemType> *bt) {
    if(bt == NULL)
        return;
    ThrBiTree(bt->lchild);          /*对左子树建立线索*/
    if(bt->lchild == NULL) {        /*设置 bt 的前驱线索*/
        bt->ltag = 1;
        bt->lchild = pre;
    }
    if(bt->rchild == NULL) {        /*修改 bt 的右标志域*/
        bt->rtag = 1;
    }
    if(pre != NULL && pre->rtag == 1) { /*设置 pre 的后继线索*/
```

```
            pre->rchild = bt;
        }
        pre = bt;                          /*更新 pre*/
        ThrBiTree(bt->rchild);             /*对右子树建立线索*/
    }
```

3. 构造函数

类模板 InThrBiTree 的构造函数首先创建初始的线索二叉链表，因为遍历序列的第一个结点没有前驱，因此将 pre 初始化为 NULL，然后调用中序线索化二叉树的方法。算法描述如下。

```
template<class ElemType>
InThrBiTree<ElemType>::InThrBiTree() {
    /*创建未线索化的二叉树*/
    root = Creat(root);
    /*为 pre 赋初值*/
    pre = NULL;
    /*中序线索化二叉树*/
    ThrBiTree(root);
}
```

4. 在中序线索二叉链表上求已知结点的后继

如果已经创建好中序线索二叉链表，则在其上求中序遍历的后继相对容易。对于已知结点 p，如果 p->rtag=1，则说明 p 没有右孩子，p->rchild 即为其后继。如果 p->rtag=0，则说明 p 存在右孩子，p->rchild 为其右孩子，p 的中序遍历的后继应为中序遍历 p 的右子树的第一个结点，即 p 的右子树最左下的结点。算法描述如下。

```
template<class ElemType>
ThrBiNode<ElemType> *InThrBiTree<ElemType>::Next(ThrBiNode<ElemType> *p) {
    ThrBiNode<ElemType> *q;
    if(p->rtag == 1)                   /*p 无右孩子,rchild 为线索,指向 p 的后继*/
        q = p->rchild;
    else {                                  /*p 有右孩子,后继为 p 的右子树最左下的结点*/
        q = p->rchild;
        while(q->ltag == 0)
            q = q->lchild;
    }
    return q;
}
```

5. 在中序线索二叉链表上中序遍历

在非空的中序线索二叉链表上进行中序遍历只需要找到第一个结点，然后在当前结点存在后继时，循环寻找下一个结点即可。中序遍历的第一个结点为二叉链表的最左下的结

点。算法描述如下。

```
template < class ElemType >
void InThrBiTree < ElemType >::InOrder() {
    ThrBiNode < ElemType > * p;
    if(root == NULL)
        return;
    p = root;
    while(p -> ltag == 0)              /* 查找最左下结点 */
        p = p -> lchild;
    cout << p -> data <<" ";
    while(p -> rchild != NULL) {       /* 查找下一个结点 */
        p = Next(p);
        cout << p -> data <<" ";
    }
    cout << endl;
}
```

实现线索二叉链表的详细代码可参照 ch05\InThrBiTree 目录下的文件，当构造如图 5-13 所示的二叉树时，二叉树的扩展前序序列为 $ABD\#G\#\#E\#\#C\#F\#\#$时，运行结果如图 5-25 所示。

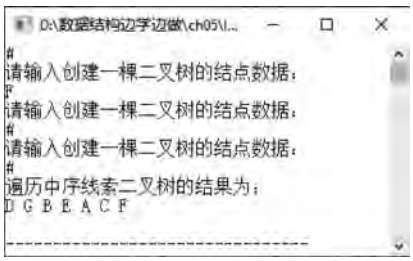


图 5-25　线索二叉链表的运行结果

5.5.4　赫夫曼树和赫夫曼编码

1. 赫夫曼树的定义

赫夫曼树(Huffman tree)也称为最优二叉树，应用非常广泛。

(1) **叶子结点的权值**：为二叉树的叶子赋予一个有意义的数值量。

(2) **二叉树的带权路径长度**(weighted path length，WPL)：假设二叉树具有 n 个带有权值的叶子结点，第 $i(1\leqslant i\leqslant n)$个叶子结点的权值为 w_i，从根结点到第 i 个叶子结点的路径长度为 l_i，则二叉树的带权路径长度为：

$$\mathrm{WPL}=\sum_{i=1}^{n}w_i l_i$$

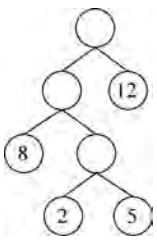


图 5-26　带权二叉树

例如图 5-26 所示的二叉树，其 WPL＝8×2＋2×3＋5×3＋12×1＝49。

(3) **赫夫曼树**：给定一组具有确定权值的叶子结点，构造出各种不同的二叉树中带权路径长度最短的二叉树称为赫夫曼树。

赫夫曼树中只有度为 0 和度为 2 的结点。另外，在赫夫曼树中通常权值越大的叶子结点离根越近，权值越小的叶子结点离根越远。

2. 赫夫曼树的构造

给定一组权值构造赫夫曼树的算法称为赫夫曼算法，其基本思想为：

（1）初始化：由给定的 n 个权值$\{w_1,w_2,\cdots,w_n\}$构造 n 棵只包含一个根结点的二叉树，记为 $F=\{T_1,T_2,\cdots,T_n\}$，第 i 棵二叉树根结点的权值为 w_i。

（2）筛选与合并：在 F 中选择两棵根结点权值最小的二叉树，分别作为左、右子树构造一棵新的二叉树，新二叉树的根结点的权值为左、右子树根结点权值的和。

（3）删除与加入：在 F 中删除作为左、右子树的二叉树，加入新构造的二叉树。

（4）重复(2)和(3)，直到 F 中只剩下一棵二叉树时，这棵二叉树即为赫夫曼树。

例 5-2 给定权值 $W=\{2,5,8,12\}$，图 5-27 给出了构造赫夫曼树的过程。

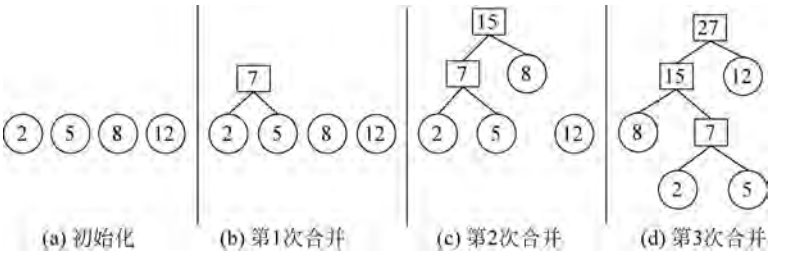


图 5-27 赫夫曼树的构造过程

由于合并二叉树时没有规定左子树或右子树，另外，有可能存在重复的权值，因此赫夫曼树是不唯一的。但是一旦确定了权值，赫夫曼树的带权路径长度是唯一的。由赫夫曼树的构造过程可知，总是由左、右子树确定根结点，因此使用三叉链表的静态链表的形式存储二叉树以实现赫夫曼算法。定义结点结构如下。

```
struct Element{
    int weight;                          /* 结点的权值 */
    int lchild, rchild, parent;          /* 结点的左孩子、右孩子、双亲在一维数组中的下标 */
}
```

给定权值数组 $w[]$及数组长度 n，构造赫夫曼树的算法描述如下。

```
void HuffmanTree(element huffTree[], int w[], int n) {
    int i, j, k;
    int m1, m2;
    /* 结点初始化 */
    for(i = 0; i < 2 * n - 1; i++) {
        huffTree[i].parent = -1;
        huffTree[i].lchild = -1;
        huffTree[i].rchild = -1;
    }
    /* 构造初始的 n 棵二叉树 */
    for(i = 0; i < n; i++) {
        huffTree[i].weight = w[i];
    }
    for(k = n; k < 2 * n - 1; k++) {
        /* 取两棵根结点权值最小的赫夫曼树 */
        /* m1 确定初值 */
        for(j = 0; j < k; j++) {
            if(huffTree[j].parent == -1) {
```

```
                m1 = j;
                break;
            }
        }
        /* m2 确定初值 */
        for(j = 0; j < k; j++) {
            if(huffTree[j].parent == -1 && j != m1) {
                m2 = j;
                break;
            }
        }
        if(huffTree[m2].weight < huffTree[m1].weight) {
            int temp = m1;
            m1 = m2;
            m2 = temp;
        }
        /* 确定根结点权值最小的两棵二叉树的下标 */
        for(j = 0; j < k; j++) {
            if(huffTree[j].parent == -1 && j != m1 && j != m2) {
                if(huffTree[j].weight < huffTree[m1].weight) {
                    m2 = m1;
                    m1 = j;
                }
                else if(huffTree[j].weight < huffTree[m2].weight) {
                    m1 = j;
                }
            }
        }
        /* 将选中的两棵二叉树合并,并将根结点信息存于下标 k 处 */
        huffTree[m1].parent = k;
        huffTree[m2].parent = k;
        huffTree[k].lchild = m1;
        huffTree[k].rchild = m2;
        huffTree[k].weight = huffTree[m1].weight + huffTree[m2].weight;
    }
}
```

构造赫夫曼树的详细代码可参照 ch05\Huffman 目录下的文件,当叶子结点的权值为 2、5、8、12 时,运行结果如图 5-28 所示。

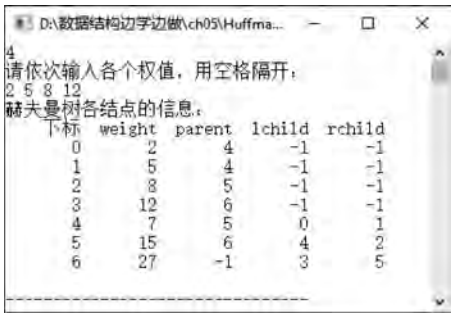


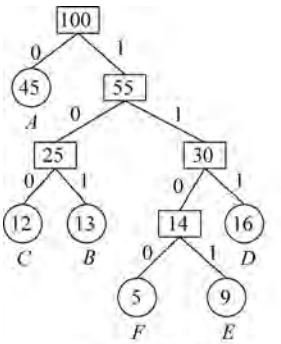
图 5-28　构造赫夫曼树的运行结果

3. 赫夫曼编码

如果一组编码中任意一个编码都不是其他编码的前缀,则称这组编码为**前缀编码**(prefix code)。前缀编码保证了解码的唯一性。等长编码指的是一组编码的长度都相等,不等长编码指的是一组编码的长度不完全相等。为了使编码总长度最短,对于不等长编码,应该使出现频次高的编码长度尽可能短,而出现频次低的编码长度可以适当加长。**赫夫曼编码**

(Huffman code)是最经济的前缀编码。利用字符出现的频次作为叶子结点的权值构造赫夫曼树，赫夫曼树中规定左分支为0，右分支为1，从根结点到叶子的路径组成的0和1序列即为叶子结点对应字符的赫夫曼编码。

例 5-3 假设对于一组字符{*A*，*B*，*C*，*D*，*E*，*F*}，使用的频次分别是{45，13，12，16，9，5}，则构造的一棵赫夫曼树如图5-29(a)所示，其对应的赫夫曼编码如图5-29(b)所示。



(a) 赫夫曼树

字符	频次	编码
A	45	0
B	13	101
C	12	100
D	16	111
E	9	1101
F	5	1100

(b) 赫夫曼编码

图 5-29 赫夫曼树和赫夫曼编码

因为赫夫曼树不唯一，所以赫夫曼编码也不唯一。

5.5.5 求二叉树的最小深度

二叉树的最小深度指的是从根结点到最近的叶子结点的距离。当二叉树为空时，返回0；当二叉树的左子树为空时，返回右子树的最小深度加1；当二叉树的右子树为空时，返回左子树的最小深度加1；当二叉树的左右子树都不为空时，返回左右子树最小深度较小的值加1。使用C++语言描述算法如下。

```
template<class ElemType>
int BiTree<ElemType>::SmallestDepth(BiNode<ElemType> *bt) {
    if(bt == NULL)
        return 0;
    if(bt->lchild == NULL)
        return SmallestDepth(bt->rchild) + 1;
    if(bt->rchild == NULL)
        return SmallestDepth(bt->lchild) + 1;
    int m = SmallestDepth(bt->lchild) + 1;
    int n = SmallestDepth(bt->rchild) + 1;
    return m<n ? m : n;
}
```

求二叉树最小深度的详细代码可参照ch05\BiTreeSmallestDepth目录下的文件。当构造如图5-13所示的二叉树时，二叉树的扩展前序序列为*ABD*##*G*###*E*###*C*#*F*##时，运行结果如图5-30所示。

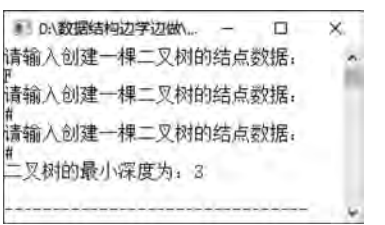


图 5-30 求二叉树最小深度的运行结果

5.5.6　判断二叉树是否是完全二叉树

要判断一棵二叉树是否是满二叉树，可以求二叉树的深度 L，再求二叉树的叶子结点的个数 N，如果满足条件 $N=2^{L-1}$，则二叉树是满二叉树，否则不是满二叉树。判断满二叉树的详细代码可参照 ch05\JudgeFullBiTree 目录下的文件。

任意一棵二叉树都可以扩充成一棵满二叉树，如果用'#'表示扩充以后的空结点，在层序遍历一棵非完全二叉树时，非空结点之前会出现空结点；而层序遍历一棵完全二叉树时，在非空结点之前不会再出现空结点，即一旦层序遍历中出现空结点，则之后再不会出现非空结点。如果在遍历二叉树时遇到空结点，整棵二叉树的遍历已经完成，则二叉树是完全二叉树；如果遇到空结点时，整棵二叉树的遍历还没有结束，即又遇到了非空结点，则二叉树不是完全二叉树。判断完全二叉树的算法使用伪代码描述如下。

```
1. 初始化空顺序队列 Q;
2. 如果 bt == NULL,返回 1;
3. bt 入队;
4. 当队列出队的指针 p 不为 NULL 时循环:
    4.1 p->lchild 入队;
    4.2 p->rchild 入队;
5. 当队列不为空时循环:
    5.1 p = 出队的指针;
    5.2 如果 p 不为空,则返回 0
6. 返回 1;
```

使用 C++语言描述算法如下。

```
template <class ElemType>
int BiTree<ElemType>::IsCompleteBiTree(BiNode<ElemType> *bt) {
    CirQueue<BiNode<ElemType> *> Q;
    BiNode<ElemType> *p;
    if(bt == NULL) return 1; /*空二叉树*/
    Q.EnQueue(bt);
    /*当出队的队头指针不为空时,将其左、右指针入队*/
    while((p = Q.DeQueue()) != NULL) {
        Q.EnQueue(p->lchild);
        Q.EnQueue(p->rchild);
    }
    /*当遇到空指针时,判断队列中是否有非空指针*/
    /*如果有,则不是完全二叉树*/
    /*没有,则为完全二叉树*/
    while(!Q.Empty()) {
        p = Q.DeQueue();
```

```
            if(p != NULL)
                return 0;
        }
        return 1;
}
```

判断完全二叉树的详细代码可参照 ch05\JudgeCompleteBiTree 目录下的文件，当构造的二叉树如图 5-19(a)所示，输入扩展的前序遍历序列为 $AB\#\#CD\#\#\#$ 时，运行结果如图 5-31 所示。

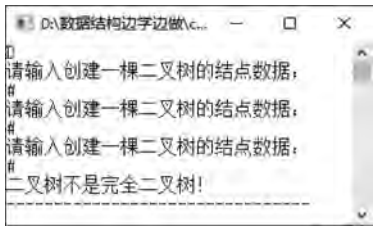


图 5-31　判断完全二叉树的运行结果 1

当构造的二叉树如图 5-32 所示时，输入的扩展的前序遍历序列为 $ABD\#\#E\#\#C\#\#$ 时，运行结果如图 5-33 所示。

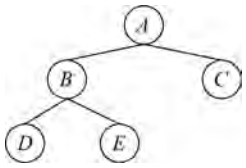


图 5-32　一棵完全二叉树示意图

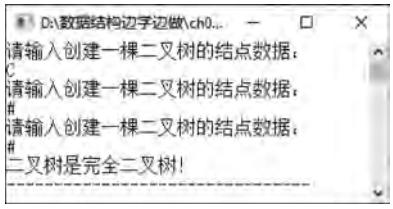


图 5-33　判断完全二叉树的运行结果 2

5.5.7　判断二叉树的结构是否对称

二叉树结构性对称指的是不考虑结点的数据值，对于二叉树中任意一个非空的结点，左、右孩子都不存在，或者左、右孩子都存在；如果结点存在子树，则左、右子树的结构也分别是结构对称的。图 5-34 所示的二叉树结构不对称，图 5-35 所示的二叉树结构对称。

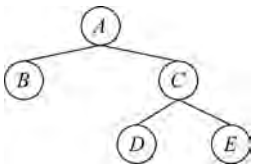


图 5-34　结构不对称的二叉树

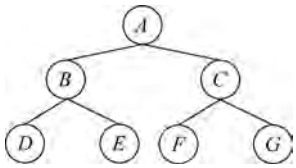


图 5-35　结构对称的二叉树

如果二叉树为空，则是结构对称的。如果二叉树不为空，则判断其左、右子树是否是结构对称的。判断一棵非空的二叉树的左、右子树是否结构对称的算法描述如下。

```
template < class ElemType >
int BiTree < ElemType >::IsCheck(BiNode < ElemType > * lchild, BiNode < ElemType > * rchild) {
    if(lchild == NULL && rchild == NULL)
        return 1;
    if(lchild == NULL || rchild == NULL)
        return 0;
```

```
    return IsCheck(lchild->lchild, rchild->rchild) && IsCheck(lchild->rchild, rchild->
lchild);
}
```

判断一棵二叉树是否结构对称的算法描述如下。

```
template<class ElemType>
int BiTree<ElemType>::IsStructureSymmetric(BiNode<ElemType> *bt) {
    if(bt == NULL)
        return 1;
    return IsCheck(bt->lchild, bt->rchild);
}
```

详细代码可参照 ch05\JudgeStructureSymmetirc 目录下的文件，当二叉树如图 5-34 所示，输入的扩展先序序列为 $AB\#\#CD\#\#E\#\#$ 时，运行结果如图 5-36 所示；当二叉树如图 5-35 所示，输入的扩展先序序列为 $ABD\#\#E\#\#CF\#\#G\#\#$ 时，运行结果如图 5-37 所示。

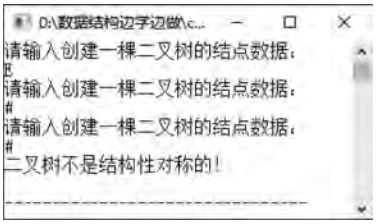


图 5-36　判断二叉树结构性对称运行结果 1

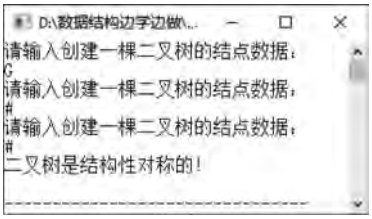


图 5-37　判断二叉树结构性对称运行结果 2

5.5.8　判断二叉树是否对称

判断二叉树是否关于中轴线对称指的是在考虑结点值的情况下，二叉树的任意一个非空结点，如果结点存在左、右孩子，则左、右孩子的值相同。并且左、右子树也都是关于中轴线对称的。判断二叉树是否对称的算法与判断二叉树是否结构性对称的算法非常类似，区别之处在于结构性对称不需要考虑结点的值，而对称需要考虑结点的值。例如，图 5-38 所示的二叉树不是对称的，因为结点 A 的左、右子树不对称。图 5-39 所示的二叉树是对称的。

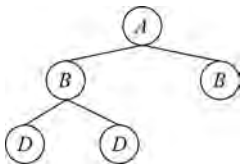


图 5-38　不对称的二叉树

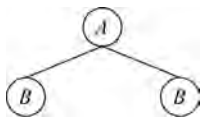


图 5-39　对称的二叉树

判断一棵非空二叉树的左右子树是否对称的算法描述如下。

```
template < class ElemType >
int BiTree < ElemType >::IsCheck(BiNode < ElemType > * lchild, BiNode < ElemType > * rchild)
{
    if(lchild == NULL && rchild == NULL)
        return 1;
    if(lchild == NULL || rchild == NULL)
        return 0;
    /* 此处与判断结构性对称不同 */
    if(lchild -> data != rchild -> data)
        return 0;
    return IsCheck(lchild -> lchild, rchild -> rchild) && IsCheck(lchild -> rchild, rchild -> lchild);
}
```

判断一棵二叉树是否对称的算法描述如下。

```
template < class ElemType >
int BiTree < ElemType >::IsSymmetric(BiNode < ElemType > * bt) {
    if(bt == NULL)
        return 1;
    return IsCheck(bt -> lchild, bt -> rchild);
}
```

详细代码可参照 ch05\ JudgeSymmetric 目录下的文件，当二叉树如图 5-38 所示，扩展的先序遍历序列为 $ABD\#\#D\#\#B\#\#$ 时，运行结果如图 5-40 所示。当二叉树如图 5-39 所示，扩展的先序遍历序列为 $AB\#\#B\#\#$ 时，运行结果如图 5-41 所示。

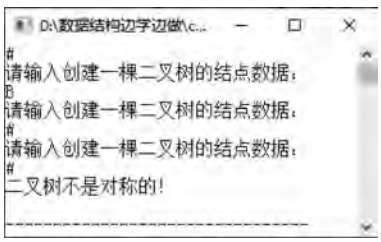


图 5-40　判断二叉树对称性的运行结果 1

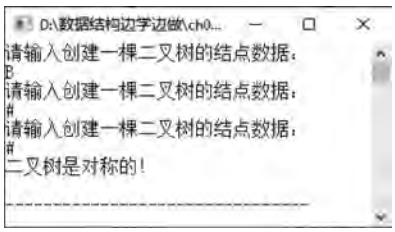


图 5-41　判断二叉树对称性的运行结果 2

5.5.9　求二叉树第 k 层的结点个数和叶子结点个数

求二叉树的第 k 层的结点个数可以采用递归的方法。

假设指向根结点的指针为 bt，算法描述如下。

1. 如果 bt 为空，或者层数 k < 1，则为空二叉树或者参数不合要求，返回 0；
2. 如果 bt 不为空，并且此时层数 k = 1，则返回 1；
3. 如果 bt 不为空，并且此时层数 k > 1，则返回 bt 左子树第 k－1 层的结点数和 bt 右子树第 k－1 层结点数的和；

使用 C++语言描述算法如下。

```
template < class ElemType >
int BiTree < ElemType >::GetNodesNumberOfKthLevel(BiNode < ElemType > * bt, int k) {
    if(bt == NULL || k < 1)
        return 0;
    if(k == 1)
        return 1;
    return GetNodesNumberOfKthLevel(bt -> lchild, k - 1) + GetNodesNumberOfKthLevel(bt ->
rchild, k - 1);
}
```

类似地，求二叉树的第 k 层的叶子结点个数也可以采用递归的方法。

假设指向根结点的指针为 bt，算法描述如下。

1. 如果 bt 为空，或者层数 k < 1，则为空二叉树或者参数不合要求，返回 0；
2. 如果 bt 不为空，并且此时层数 k = 1，则需要判断 bt 是否为叶子结点：
   2.1 如果 bt 的左右子树均为空，则返回 1；
   2.2 如果 bt 的左右子树之一不为空，则返回 0；
3. 如果 bt 不为空，并且此时层数 k > 1，则返回 bt 左子树第 k－1 层的叶子结点数和 bt 右子树第 k－1 层叶子结点数的和；

使用 C++语言描述算法如下。

```
template < class ElemType >
int BiTree < ElemType >::GetLeafNodesNumberOfKthLevel(BiNode < ElemType > * bt, int k) {
    if(bt == NULL || k < 1)
        return 0;
    if(bt != NULL) {
        /* 判断 bt 是否是叶子 */
        if(k == 1) {
            if(bt -> lchild == NULL && bt -> rchild == NULL)
                return 1;
            else
                return 0;
        }
        /* 递归求左、右子树中的叶子数的和 */
        if(k > 1) {
            return GetLeafNodesNumberOfKthLevel(bt -> lchild, k - 1) +
GetLeafNodesNumberOfKthLevel(bt -> rchild, k - 1);
        }
    }
}
```

求二叉树第 k 层结点数以及叶子结点数的详细代码可参照 ch05\ GetNodesNumberOfKthLevel 目录下的文件，当二叉树如图 5-13 所示，扩展的先序序列为 $ABD\#G\#\#E\#\#C\#F\#\#$，$k=3$ 时，运行结果如图 5-42 所示。

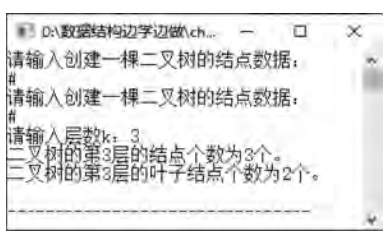


图 5-42　求二叉树第 k 层结点个数和叶子结点个数的运行结果

5.5.10 打印二叉树第 k 层的结点和叶子结点

打印二叉树第 k 层的结点和叶子结点与求二叉树第 k 层的结点个数和叶子结点个数类似。假设指向根结点的指针为 bt，打印第 k 层的结点的算法使用伪代码描述如下。

1. 如果 bt 为空，或者层数 k＜1，则为空二叉树或者参数不合要求，空操作返回；
2. 如果 bt 不为空，并且此时层数 k＝1，则输出 bt 的数据域；
3. 如果 bt 不为空，并且此时层数 k＞1，则：
   3.1 打印 bt 左子树第 k－1 层的结点；
   3.2 打印 bt 右子树第 k－1 层的结点；

使用 C++语言描述算法如下。

```
template <class ElemType>
void BiTree<ElemType>::PrintNodesOfKthLevel(BiNode<ElemType> *bt, int k) {
    if(bt == NULL || k < 1)
        return;
    if(k == 1)
        cout << bt->data <<" ";
    PrintNodesOfKthLevel(bt->lchild, k-1);
    PrintNodesOfKthLevel(bt->rchild, k-1);
}
```

打印第 k 层的叶子结点的算法使用伪代码描述如下。

1. 如果 bt 为空，或者层数 k＜1，则为空二叉树或者参数不合要求，空操作返回；
2. 如果 bt 不为空，并且此时层数 k＝1，则判断 bt 是否为叶子结点：
   2.1 如果 bt 的左右子树均为空，则输出 bt 的数据域；
3. 如果 bt 不为空，并且此时层数 k＞1，则：
   3.1 打印 bt 左子树第 k－1 层的叶子结点；
   3.2 打印 bt 右子树第 k－1 层的叶子结点；

使用 C++语言描述算法如下。

```
template <class ElemType>
void BiTree<ElemType>::PrintLeafNodesOfKthLevel(BiNode<ElemType> *bt, int k) {
    if(bt == NULL || k < 1)
        return;
    if(bt != NULL) {
        /*判断 bt 是否是叶子*/
        if(k == 1) {
            if(bt->lchild == NULL && bt->rchild == NULL)
                cout << bt->data <<" ";
```

```
            }
            else if(k > 1) {
                /* 递归打印左子树第 k - 1 层的叶子结点 */
                PrintLeafNodesOfKthLevel(bt->lchild, k - 1);
                /* 递归打印右子树第 k - 1 层的叶子结点 */
                PrintLeafNodesOfKthLevel(bt->rchild, k - 1);
            }
        }
    }
```

打印二叉树第 k 层结点以及叶子结点的详细代码可参照 ch05\PrintLeafNodesOfKthLevel 目录下的文件，当二叉树如图 5-13 所示，扩展的先序序列为 $ABD\#G\#\#E\#\#C\#F\#\#$，$k=3$ 时，运行结果如图 5-43 所示。

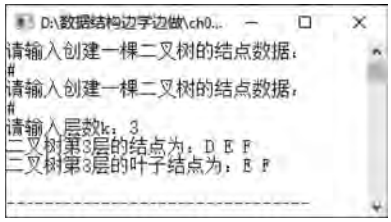


图 5-43　打印二叉树第 k 层结点个数和叶子结点个数的运行结果

5.5.11　求二叉树最大的结点距离

如果把二叉树看成图，父子结点之间的连线看作是双向的，则两个结点之间的距离定义为两个结点之间的边的条数。二叉树中结点之间的最大距离也称为二叉树的最长路径。图 5-44 所示的二叉树结点 H 和 I 的距离为 6，此距离是二叉树结点的最大的距离，也是二叉树中最长路径的长度。

在图 5-45 中，二叉树的最远的两个结点 D 和 C 之间的距离是 3。在图 5-46 中，二叉树的最远的两个结点 E 和 H 之间的距离是 5。不同之处在于图 5-45 中，二叉树的最长路径经过根结点，而图 5-46 中的二叉树的最长路径不经过根结点。

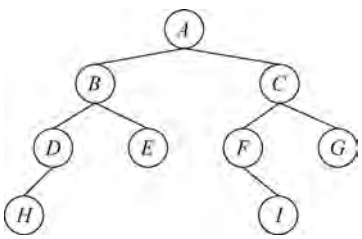


图 5-44　二叉树中相距最远的两个结点 A 和 B

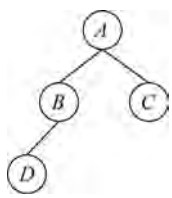


图 5-45　最长路径经过根结点

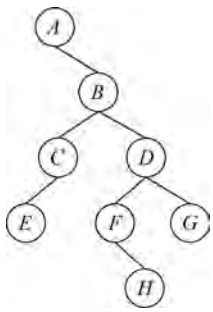


图 5-46　二叉树的最长路径不结过根结点的情况

从以上例子中可以看出，相距最远的两个结点，或者是两个叶子结点，或者是一个叶子结点到根结点。如果最长路径经过根结点，则距离最远的结点都是左、右子树中距离根结点最远的叶子结点，如图 5-44 和图 5-45 所示。如果路径不经过根结点，则肯定是根的左子树中的最大的结点距离或者是根的右子树的最大的结点距离，如图 5-46 所示，结点 E 和结点 H 之间的路径不经过根结点，同时也是根结点右子树中最大的结点距离。

可以采用递归的方法求解二叉树中最大的结点距离。由于需要记录任意一个非空结点的左子树和右子树的最大结点距离，因此将二叉链表存储的结点信息再扩充两个信息域，即 maxLeft 和 maxRight，其中 maxLeft 表示左子树的最大结点距离，maxRight 表示右子树的最大结点距离。扩展以后的二叉链表结点结构描述如下。

```
template<class ElemType>
struct BiNode{
    ElemType data;
    BiNode<ElemType> *lchild, *rchild;
    int maxLeft;                    /*左子树最大的结点距离*/
    int maxRight;                   /*右子树最大的结点距离*/
};
```

求二叉树结点的最大距离的算法描述如下。

```
/*全局变量,记录二叉树的最大的结点距离*/
int maxLen = 0;
template<class ElemType>
void BiTree<ElemType>::GetMaxPathLength(BiNode<ElemType> *bt) {
    /*二叉树为空,空操作返回*/
    if(bt == NULL) {
        return;
    }
    /*二叉树左子树为空,则左子树的最大结点距离为0*/
    if(bt->lchild == NULL) {
        bt->maxLeft = 0;
    }
    /*二叉树右子树为空,则右子树的最大结点距离为0*/
    if(bt->rchild == NULL) {
        bt->maxRight = 0;
    }
    /*如果左子树不为空,则递归寻找左子树的最大结点距离*/
    if(bt->lchild != NULL) {
        GetMaxPathLength(bt->lchild);
    }
    /*如果右子树不为空,则递归寻找右子树的最大结点距离*/
    if(bt->rchild != NULL) {
        GetMaxPathLength(bt->rchild);
    }
    /*计算左子树最大结点距离*/
    if(bt->lchild != NULL) {
        int temp = 0;
        int ll = bt->lchild->maxLeft;
        int lr = bt->lchild->maxRight;
        temp = ll > lr ? ll : lr;
        bt->maxLeft = temp + 1;
    }
    /*计算右子树最大结点距离*/
    if(bt->rchild != NULL) {
        int temp = 0;
        int rl = bt->rchild->maxLeft;
        int rr = bt->rchild->maxRight;
```

```
            temp = rl > rr ? rl : rr;
            bt->maxRight = temp + 1;
        }
        /*更新最大结点距离*/
        if(bt->maxLeft + bt->maxRight > maxLen) {
            maxLen = bt->maxLeft + bt->maxRight;
        }
    }
```

详细代码可参照 ch05\GetMaxPathLength 目录下的文件，当输入的扩展的二叉树的前序序列为 $ABDH\#\#\#E\#\#CF\#I\#\#G\#\#$ 时，对应的二叉树如图 5-44 所示，运行结果如图 5-47 所示。

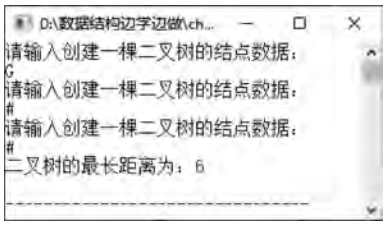


图 5-47　求二叉树最大结点距离的运行结果 1

当输入的扩展的二叉树的前序序列为 $ABD\#\#\#C\#\#$ 时，对应的二叉树如图 5-45 所示，运行结果如图 5-48 所示。

当输入的扩展的二叉树的前序序列为 $A\#BCE\#\#\#DF\#H\#\#G\#\#$ 时，对应的二叉树如图 5-46 所示，运行结果如图 5-49 所示。

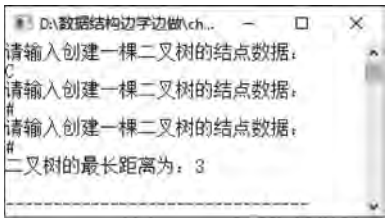


图 5-48　求二叉树最大结点距离的运行结果 2

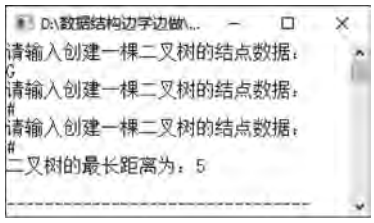


图 5-49　求二叉树最大结点距离的运行结果 3

5.5.12　由前序序列和中序序列构造二叉树

任何一棵非空二叉树的某种遍历序列都是唯一的，那么反过来呢？二叉树的一种普通的遍历序列可以唯一地确定二叉树吗？显然不能。由前文的内容可知，扩展的前序序列可以唯一地确定二叉树，但是普通的前序遍历序列不能唯一地确定二叉树。同理，普通的中序遍历序列、后序遍历序列、层序遍历序列都不能唯一地确定二叉树。

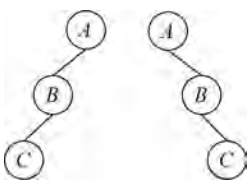


图 5-50　两棵二叉树

那么两种序列可以唯一地确定二叉树吗？例如前序序列为 $A\ B\ C$，后序序列为 $C\ B\ A$ 时，如图 5-50 所示的两棵二叉树都满足以上条件，因此由二叉树的前序序列和后序序列也不能唯一地确定二叉树。

由二叉树的前序序列和中序序列可以唯一地确定二叉树，方法如下。

(1) 前序序列的第一个结点为根；

(2) 在中序序列中找到根的位置，根把中序序列分为左、右两个部分，根之前的序列为

左子树的中序序列，根之后的序列为右子树的中序序列；

（3）在前序序列中寻找左子树的前序序列以及右子树的前序序列；

（4）由左子树的前序序列和中序序列确定左子树；

（5）由右子树的前序序列和中序序列确定右子树。

例 5-4 二叉树的前序序列为 *A B D C E F*，中序序列为 *D B A E C F*，确定二叉树。

前序序列的第一个结点为 *A*，即为二叉树的根。中序序列中 *A* 之前的 *D B* 为左子树的中序序列，*A* 之后的 *E C F* 为右子树的中序序列。前序序列中 *B D* 为左子树的前序序列，*C E F* 为右子树的前序序列。

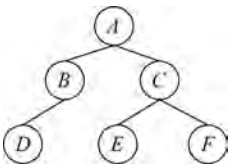


图 5-51 一棵二叉树示意图

使用左子树的前序序列（*B D*）和中序序列（*D B*）确定左子树，使用右子树的前序序列（*C E F*）和中序序列（*E C F*）确定右子树。最后确定的二叉树的结构如图 5-51 所示。

利用前序序列和中序序列构造二叉树的算法使用 C++ 语言描述如下。

```
template < class ElemType >
BiNode < ElemType > * BiTree < ElemType >::Rebuild(ElemType * preOrder, ElemType * inOrder, int n) {
    if(n == 0)
        return NULL;
    / * 获得前序遍历的第一个结点 * /
    ElemType c = preOrder[0];
    / * 创建根结点 * /
    BiNode < ElemType > * node = new BiNode < ElemType >;
    node -> data = c;
    node -> lchild = NULL;
    node -> rchild = NULL;
    int i;
    / * 在中序遍历序列中寻找根结点的位置 * /
    for(i = 0; i < n && inOrder[i] != c; i++)
        ;
    / * 左子树结点个数 * /
    int lenLeft = i;
    / * 右子树结点个数 * /
    int lenRight = n - i - 1;
    / * 左子树不为空，递归重建左子树 * /
    if(lenLeft > 0)
        node -> lchild = Rebuild(&preOrder[1], &inOrder[0], lenLeft);
    / * 右子树不为空，递归重建右子树 * /
    if(lenRight > 0)
        node -> rchild = Rebuild(&preOrder[lenLeft + 1], &inOrder[lenLeft + 1], lenRight);
    return node;
}
```

详细代码可参照 ch05\ RebuildBiTreeFromPreIn 目录下的文件，当遍历序列如例 5-3 所示时，重建的二叉树如图 5-51 所示。为了验证结果，对二叉树进行了各种遍历，运行结果如图 5-52 所示。

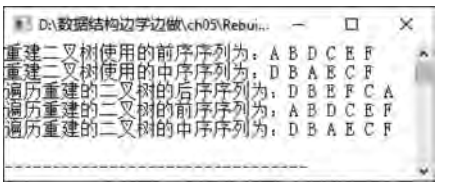


图 5-52 利用前序序列和中序序列重建二叉树的运行结果

5.5.13　由后序序列和中序序列构造二叉树

利用二叉树的后序序列和中序序列也可以唯一地确定二叉树，方法与利用前序序列和中序序列确定二叉树类似。二叉树的后序序列的最后一个结点为根结点。使用C++语言描述算法如下。

```
template<class ElemType>
BiNode<ElemType>* BiTree<ElemType>::Rebuild(ElemType *postOrder, ElemType *inOrder,
int n) {
    if(n == 0)
        return NULL;
    /*获得后序遍历的最后一个结点*/
    ElemType c = postOrder[n-1];
    /*创建根结点*/
    BiNode<ElemType> *node = new BiNode<ElemType>;
    node->data = c;
    node->lchild = NULL;
    node->rchild = NULL;
    int i;
    /*在中序遍历序列中寻找根结点的位置*/
    for(i = 0; i<n && inOrder[i] != c; i++)
        ;
    /*左子树结点个数*/
    int lenLeft = i;
    /*右子树结点个数*/
    int lenRight = n - i - 1;
    /*左子树不为空,重建左子树*/
    if(lenLeft>0)
        node->lchild = Rebuild(&postOrder[0], &inOrder[0], lenLeft);
    /*右子树不为空,重建右子树*/
    if(lenRight>0)
        node->rchild = Rebuild(&postOrder[lenLeft], &inOrder[lenLeft+1], lenRight);
    return node;
}
```

详细代码可参照 ch05\ RebuildBiTreeFromPostIn 目录下的文件，当遍历序列如例 5-3 所示时，重建的二叉树如图 5-51 所示。为了验证结果，对二叉树进行了遍历，运行结果如图 5-53 所示。

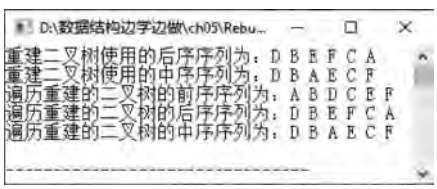


图 5-53　利用后序序列和中序序列重建二叉树的运行结果

5.5.14　求二叉树的镜像

将一棵二叉树的所有结点的左、右子树调换位置，就成了二叉树的镜像。图 5-54 所示的两棵二叉树互为镜像。

可以利用递归的方法求二叉树的镜像，使用 C++ 语言描述算法如下。

```
void BiTree<ElemType>::GetMirror(BiNode<ElemType> *bt) {
    /*二叉树为空，空操作返回*/
    if(bt == NULL)
        return;
    /*二叉树的左、右子树都为空，空操作返回*/
    if(bt->lchild == NULL && bt->rchild == NULL)
        return;
    /*交换左、右子树*/
    BiNode<ElemType> *tmp;
    tmp = bt->lchild;
    bt->lchild = bt->rchild;
    bt->rchild = tmp;
    /*递归处理左子树*/
    GetMirror(bt->lchild);
    /*递归处理右子树*/
    GetMirror(bt->rchild);
}
```

求二叉树镜像的详细代码可参照 ch05\GetBiTreeMirror 目录下的文件，当原二叉树如图 5-54(a)所示，其对应的扩展的先序遍历序列为 $AB\#D\#\#C\#\#$ 时，运行结果如图 5-55 所示。

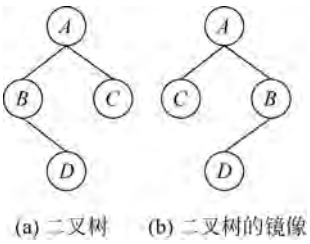


图 5-54　互为镜像的二叉树

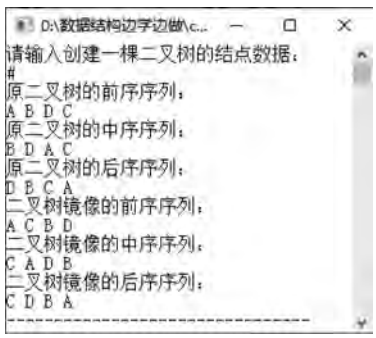


图 5-55　求二叉树镜像的运行结果

5.5.15　判断两棵二叉树是否等价

如果两棵二叉树结构相同并且相同位置上的结点的数据域也分别相同，则称两棵二叉树等价。图 5-56 所示的两棵二叉树为等价二叉树，图 5-57 所示的二叉树不是等价二叉树。

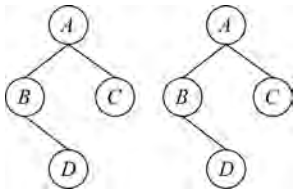


图 5-56　等价的两棵二叉树

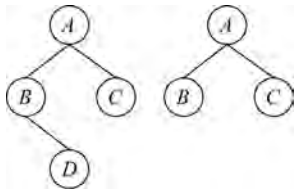


图 5-57　不等价的两棵二叉树

判断两棵二叉树是否等价的算法使用伪代码描述如下。

1. 如果两棵二叉树都为空，则返回 true；
2. 如果两棵二叉树一棵为空，另一棵非空，则返回 false；
3. 如果两棵二叉树都不为空，但根结点的数据域不相等，则返回 false；
4. 否则，递归判断两棵二叉树的左右子树是否等价。

使用 C++语言描述算法如下。

```
template<class ElemType>
int IsEqualBiTrees(BiNode<ElemType> *T1, BiNode<ElemType> *T2) {
    /*两棵二叉树都为空，返回 1 */
    if(T1 == NULL && T2 == NULL)
        return 1;
    /*两棵二叉树一棵为空，一棵不为空，返回 0 */
    if(T1 == NULL || T2 == NULL)
        return 0;
    /*两棵二叉树都不为空，并且根结点数据域不相等，返回 0 */
    if(T1->data != T2->data)
        return 0;
    /*递归判断左子树和右子树是否等价*/
    return IsEqualBiTrees(T1->lchild, T2->lchild) && IsEqualBiTrees(T1->rchild, T2->
rchild);
}
```

详细代码可参照 ch05\JudgeEqualBiTrees 目录下的文件，当两棵二叉树如图 5-56 所示时，运行结果如图 5-58 所示。当两棵二叉树如图 5-57 所示时，运行结果如图 5-59 所示。

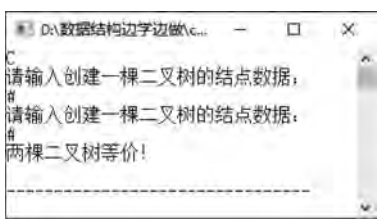


图 5-58　判断两棵二叉树是否等价的运行结果 1

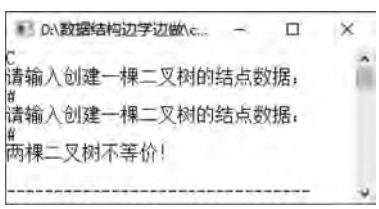


图 5-59　判断两棵二叉树是否等价的运行结果 2

5.6　树、森林与二叉树的转换

树的孩子兄弟表示法和二叉树的二叉链表表示法类似，即树的孩子兄弟表示法和二叉树的二叉链表表示法在物理结构上是相同的。如图 5-60 所示，图 5-60(a)中树的存储结构

（见图 5-60(b)）和图 5-60(c)中二叉树的存储结构（见图 5-60(d)）在内存中对应同一种状态。即给定一棵树，存在一棵唯一的二叉树与之对应。

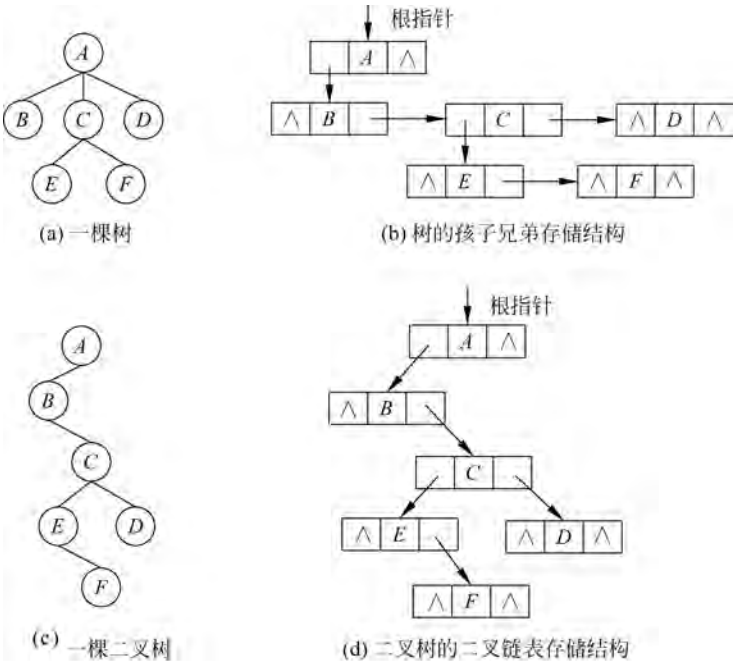


图 5-60　树与二叉树的对应关系

1. 树转换成二叉树

将一棵树转换成二叉的方法为：

（1）加线：树中所有相邻的兄弟结点之间加一条连线；

（2）去线：只保留双亲结点和第一个孩子之间的连线，删去它去其他孩子之间的连线；

（3）调整：以根结点为轴心，将水平方向的连线顺时针旋转一定的角度，使之层次分明。

例如，图 5-61 给出了一棵树转换成二叉树的过程。

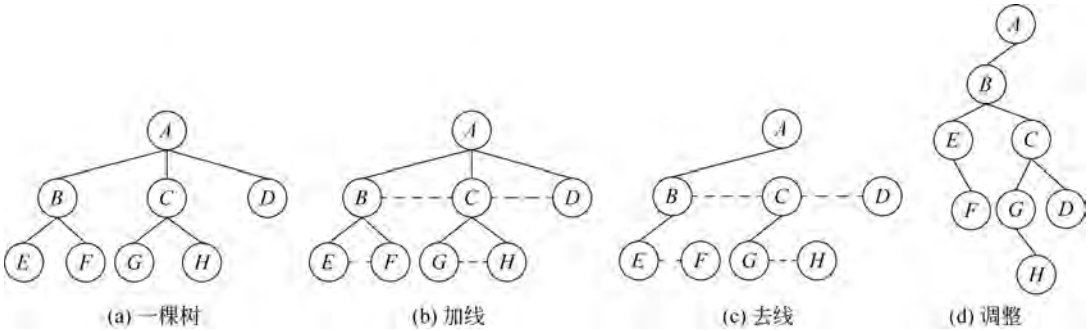


图 5-61　树转换成二叉树的过程

由树转换而来的二叉树，其根结点的右子树一定为空。因为二叉树的结点的右孩子在树中对应着其右兄弟，而树中根结点无右兄弟，因此二叉树的根结点的右子树一定为空。

根据树与二叉树的转换以及树和二叉树的遍历可知：树的前序遍历等价于二叉树的前序遍历；树的后序遍历等价于二叉树的中序遍历。

2. 森林转换成二叉树

森林转换成二叉树的方法为：

(1) 将森林中的每一棵树分别转换成二叉树；

(2) 将二叉树的根结点之间连线；

(3) 以根结点为轴心，将水平方向的连线顺时针旋转一定的角度，使之层次分明。

例如，图 5-62 给出了森林转换成二叉树的过程。

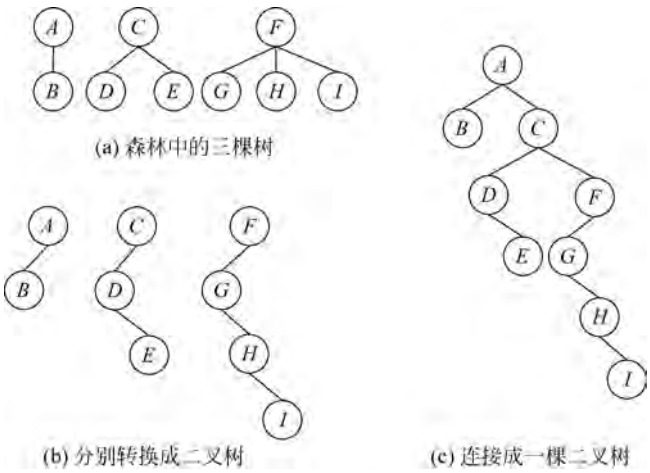


图 5-62　森林转化成二叉树的过程

3. 二叉树转换成森林或树

二叉树也可以转换成森林或树。如果二叉树的根结点无右子树，则二叉树转换成树。如果二叉树的根结点有右子树，则二叉树转换成森林。转换方法为：

(1) 加线：若结点 x 是其双亲结点 y 的左孩子，则把结点 x 的右孩子、右孩子的右孩子、……，都与结点 x 用线相连；

(2) 去线：删除原二叉树中所有双亲结点与其右孩子的连线；

(3) 调整：调整使树或森林层次分明。

例如，图 5-63 为二叉树转换成森林的过程。

4. 森林的遍历

森林的遍历包括前序遍历和后序遍历。

前序遍历森林为前序遍历森林中的每一棵树，例如对图 5-63(d)所示的森林进行前序遍历，前序序列为 $A\ B\ C\ D\ E\ F\ G\ H\ I$。

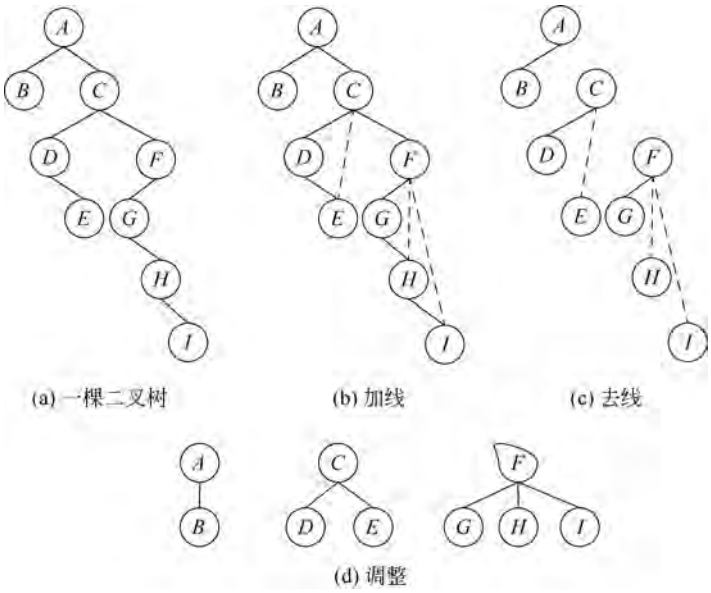


图 5-63　二叉树转换成森林的过程

后序遍历森林为后序遍历森林中的每一棵树，例如对图 5-63(d)所示的森林进行后序遍历，后序序列为 *B A D E C G H I F*。

5.7 小结

- 树状结构是应用非常广泛的非线性结构。其中，二叉树是最常用的树状结构。二叉树最常用的操作是各种遍历，包括前序遍历、中序遍历、后序遍历、层序遍历。
- 可以使用递归的算法或者非递归的算法进行遍历。除此之外，还可以在遍历的基础上进行例如求结点个数、求叶子结点个数、求二叉树的深度等操作。
- 为了充分地利用二叉链表中的空指针，可以将二叉链表进行线索化。可以方便地实现遍历中序线索二叉树。
- 判断各种特殊的二叉树也是一类重要的操作，例如完全二叉树的判断、满二叉树的判断等。
- 构造赫夫曼树是二叉树的重要应用。
- 可以根据二叉树前序遍历序列和中序遍历序列构造二叉树，也可以根据后序遍历序列和中序遍历序列构造二叉树。
- 一些扩展的操作也比较常见，例如求二叉树的最小深度、求二叉树的最大结点距离、求二叉树第 k 层的结点和叶子结点、判断两棵二叉树是否等价等操作。

习题

1. 选择题

(1) 假设树 T 的度为 4，其中度为 1、2、3、4 的结点个数分别为 4、2、1、1，则 T 中的叶子结点的个数为(　　)。

A. 5　　B. 6　　C. 7　　D. 8

(2) 若一棵二叉树中具有 10 个度为 2 的结点，5 个度为 1 的结点，则度为 0 的结点个数为(　　)。

A. 9　　B. 11　　C. 15　　D. 无法确定

(3) 已知一棵二叉树的前序遍历序列为 $A\ B\ C\ D\ E\ F$，中序遍历序列为 $C\ B\ A\ E\ D\ F$，则后序遍历序列为(　　)。

A. $C\ B\ E\ F\ D\ A$　　B. $F\ E\ D\ C\ B\ A$

C. $C\ B\ E\ D\ F\ A$　　D. 没有正确答案

(4) 二叉树的前序序列和后序列序列正好相反，则该二叉树一定是(　　)的二叉树。

A. 空或只有一个结点　　B. 深度等于其结点数

C. 任一结点无左孩子　　D. 任一结点无右孩子

(5) 深度为 h 的满 m 叉树的第 k 层有(　　)个结点。

A. m^{k-1}　　B. m^k-1　　C. m^{h-1}　　D. m^h-1

(6) 深度为 k 的完全二叉树最多有(　　)个结点。

A. $2^{k-2}+1$　　B. 2^{k-1}　　C. 2^k-1　　D. $2^{k-1}-1$

(7) 在二叉树的先序遍历序列、中序遍历序列和后序遍历序列中，所有叶子结点的相对顺序(　　)。

A. 都不相同　　B. 完全相同

C. 先序和中序相同，而与后序不同　　D. 中序和后序相同，而与先序不同

(8) 设给定权值的总数有 n 个，其赫夫曼树的结点总数为(　　)。

A. 不确定　　B. $2n$　　C. $2n+1$　　D. $2n-1$

(9) 一棵具有 124 个叶子结点的完全二叉树，最多有(　　)个结点。

A. 247　　B. 248　　C. 249　　D. 250

(10) 一个具有 1025 个结点的二叉树的深度为(　　)。

A. 11　　B. 10　　C. 11～1025　　D. 10～1024

(11) 一棵二叉树的深度为 h，所有结点的度为 0 或 2，则这棵二叉树至少有(　　)个结点。

A. $2h$　　B. $2h-1$　　C. $2h+1$　　D. $h+1$

(12) 下列存储形式中，(　　)不是树的存储结构。

A. 双亲表示法　　B. 孩子链表表示法

C. 孩子兄弟表示法　　D. 顺序存储表示法

（13）引入线索二叉树的目的是（　　）。

A. 加快查找结点的前驱或后继的速度

B. 能在二叉树中方便地进行插入或删除

C. 方便地找到双亲

D. 使二叉树的遍历结果唯一

（14）一棵赫夫曼树共有 215 个结点，对其进行赫夫曼编码，共能得到（　　）个不同的码字。

A. 107　　B. 108　　C. 214　　D. 215

（15）以下编码中，（　　）不是前缀编码。

A. 00,01,10,11　　B. 0,1,00,11

C. 0,10,110,111　　D. 1,01,000,001

（16）为 5 个使用频率不等的字符设计赫夫曼编码，不可能的方案是（　　）。

A. 000,001,010,011,1　　B. 0000,0001,001,01,1

C. 000,001,01,10,11　　D. 00,100,101,110,111

2. 填空题

（1）具有 n 个结点的二叉树，采用二叉链表存储时共有（　　）个空指针。

（2）利用树的孩子兄弟表示法，可以将树转换成（　　）。

（3）具有 n 个结点的满二叉树，其叶子结点个数为（　　）。

（4）具有 n 个结点的完全二叉树的深度为（　　）。

（5）深度为 k 的二叉树中，所含叶子的个数最多为（　　）。

（6）结点数为 101 的完全二叉树中，叶子结点的个数为（　　）。

（7）设 F 是由 T_1、T_2、T_3 三棵树组成的森林，与 F 对应的二叉树为 B，已知 T_1、T_2、T_3 的结点数分别为 n_1、n_2 和 n_3，则二叉树 B 的右子树中有（　　）个结点。

（8）已知一棵度为 3 的树有 2 个度为 1 的结点，3 个度为 2 的结点，4 个度为 3 的结点。则该树中有（　　）个叶子结点。

（9）在二叉链表中，指针 p 所指的结点为叶子的条件是（　　）。

（10）已知一棵二叉树的后序序列是 $F\ E\ G\ H\ D\ C\ B$，中序序列是 $F\ E\ B\ G\ C\ H\ D$，则它的前序序列是（　　）。

（11）树的先序遍历等价于二叉树的（　　）遍历，树的后序遍历等价于二叉树的（　　）遍历。

3. 判断题

（1）二叉树的遍历实际上是将非线性结构线性化的过程。（　　）

（2）完全二叉树一定存在度为 1 的结点。（　　）

（3）二叉树是树的特例。（　　）

（4）完全二叉树中的结点若没有左孩子，则它必为叶子。（　　）

（5）二叉树是度为 2 的树。（　　）

（6）线索二叉树中不存在空指针。（　　）

（7）可以根据前序遍历序列和后序遍历序列唯一地确定二叉树。（　　）

（8）树状结构中的数据元素之间存在一对多的逻辑关系。（　　）

（9）树和二叉树是两种不同的树状结构。（　　）

（10）顺序存储结构只适合于存储完全二叉树。（　　）

4. 问答题

（1）已知一棵满 m 叉树，设根在第 1 层，并从 1 开始自上向下分层给各个结点编号，试回答以下问题。

① 第 i 层有多少结点？

② 深度为 h 的满 m 叉树有多少个结点？

③ 编号为 k 的结点的双亲结点的编号是多少？

④ 编号为 k 的结点的第 1 个孩子的结点编号是多少？

⑤ 编号为 k 的结点在第几层？

（2）已知一棵度为 m 的树中有 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，求树中共有多少个叶子结点？

（3）已知一棵二叉树的前序遍历序列为 $A\ B\ E\ C\ D\ F\ G\ H\ I\ J$，中序遍历序列为 $E\ B\ C\ D\ A\ F\ H\ I\ G\ J$，画出这棵二叉树并写出它的后序遍历序列。

（4）已知某系统在通信联络中只可能出现 8 种字符：a、b、c、d、e、f、g、h，其概率分别为 0.05，0.29，0.07，0.08，0.14，0.23，0.03，0.11，试构造赫夫曼树并求其 WPL，并完成 8 种字符的赫夫曼编码。

（5）给定权值{8，12，4，5，26，16，9}，构造赫夫曼树，并计算其带权路径长度。

5. 算法设计题

（1）设计算法求以 root 为根指针的二叉树的最大元素的值。

（2）假设二叉树采用二叉链表存储，结点类型为 char，'#'为无效字符，设计算法求前序遍历的第 k 个结点的值。

（3）假设二叉树采用二叉链表存储，设计算法求指定结点 p 的双亲结点。