第 6 章　树与二叉树

教学要点

树是一种非线性数据结构，它的数据元素之间具有层次关系。在树结构中，除根结点没有前驱元素外，每个数据元素都只有一个前驱元素，但可以有零个或若干个后继元素。树结构可以分为无序树和有序树两种类型，有序树中最常用的是二叉树，其每个结点最多只有两个可分左右的子树。

本章介绍具有层次关系的非线性数据结构—树和二叉树的概念和定义，重点讨论二叉树的性质、存储结构和遍历算法，并介绍线索二叉树的定义、存储结构和遍历算法。

本章在 Visual Studio 中用名为 trees 的类库型项目实现有关数据结构的类型定义，用名为 treetest 的应用程序型项目实现对这些数据结构的测试和演示程序。

建议本章授课 6 学时，实验 6 学时。

6.1　树的定义与基本术语

生活中的很多对象之间具有层次关系，如家族成员、企业的管理部门、计算机的文件系统等。这种层次关系类似于自然界中的树，树的树根、枝杈和叶子分别对应于层次结构的起源、分支和分支终点。Windows、Linux 等操作系统的文件系统是一个树型结构的数据结构，根目录是文件（目录）树的根节点，子目录（文件夹）是树中的分支节点，文件是树的叶子结点。这些客观对象的表现形式可能多种多样，但在其对象成员的关系上，都可以用树结构来抽象描述。一个用树结构描述的家谱如图 6.1 所示。

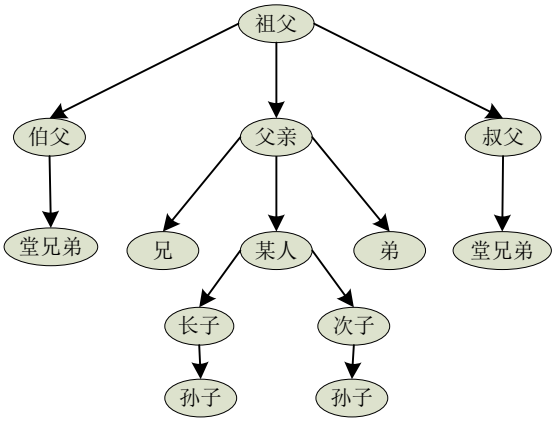


图 6.1 用树结构描述家谱

6.1.1　树的定义和表示

树（tree）可以用递归形式来定义：树 T 是由 n（$n \geqslant 0$）个结点组成的有限集合，它或者是棵空树，

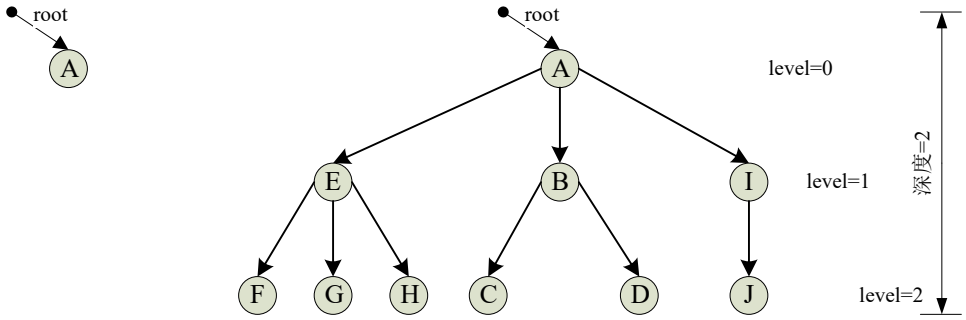
或者包含一个根结点和零或若干棵互不相交的子树。

结点个数为零，即 n=0 时称为空树；结点数 n>0 时，树 T 由一个根结点和零或若干棵互不相交的子树构成。

一颗非空树 T 具有以下特点：

- 树 T 有一个特殊的结点，它没有前驱结点，这个结点称为树的根结点（root）。
- 当树的结点数 n>1 时，根结点之外的其他结点可分为 m（m≥1）个互不相交的集合 $T_1, T_2, \cdots, T_m$，其中每个集合 T_i（1≤i≤m）具有与树 T 相同的树结构，称为子树（subtree）。每棵子树的根结点有且仅有一个直接前驱结点，但可以有零或多个直接后继结点。

图 6.2 显示了两种树的典型结构。在图 6.2 (a) 中，结点数 n=1，树中只有一个结点 A，它就是树的根结点。在图 6.2 (b) 中，结点数 n=10，A 为树的根结点，其他结点则分别在 A 的子树 T_1，T_2 和 T_3 中，其中 T_1={B，C，D}，T_2={E，F，G，F }，T_3={I，J}，子树的根分别为 B，E 和 I，可见树中每个结点都是该树中某一棵子树的根。



(a) n=1，仅有根结点的树　　(b) n=10，深度为2的树

图 6.2 树与结点

树还可以分为无序树（unorderd tree）与有序树（orderd tree）。在无序树中，结点的子树 T_1，T_2，…，T_m 之间没有次序关系。如果树中结点的子树 T_1，T_2，…，T_m 从左至右是有次序的，则称该树为有序树。通常所说的树结构指的是无序树。

若干棵互不相交的树的集合称为森林（forest）。给森林加上一个根结点就变成一棵树，将树的根结点删除就变成由子树组成的森林。

树结构可以用如图 6.2 所示的树图形表示，这种图示法比较直观，但有时显得不方便。也可以用广义表的形式表示树结构。例如，图 6.2 (b) 所示树的广义表表示形式为：A（B（C，D），E（F，G，H），I（J））。

表示树结构的广义表没有共享和递归成分，是一种纯表。广义表中的原子对应于树的叶结点，树的非叶结点则用子表结构表示。

6.1.2 树的基本术语

家谱可以用树结构来描述，反之，与树结构有关的一些基本术语也常用家族成员之间的关系来定义与说明。

1．结点（node）

结点表示树集合中的一个数据元素，例如，图 6.2 (a) 表示一棵仅有 1 个结点的树，图 6.2 (b) 表示一棵具有 10 个结点的树。树的结点一般由对应元素自身的数据和指向其子结点的指针构成。

2．子结点与父结点（child node 与 parent node）

若某结点 *N* 有子树，则子树的根结点称为结点 *N* 的子结点，又称孩子或子女结点。与子结点对应，结点 *N* 称为其子结点的父结点，又称父母或双亲结点。在一棵树中，根结点没有父结点。例如在图 6.2 (b)中，结点 *B*、*E*、*I* 是结点 A 的子树的根，所以结点 *A* 的子结点包括结点 *B*、*E*、*I*， 结点 *A* 是这些结点的父结点。结点 *A* 作为整个树的根结点，它没有父结点。

3．兄弟结点（sibling node）

同一个父结点的子结点之间是兄弟关系，它们互称为兄弟结点。例如，结点 *B*、*E*、*I* 是兄弟，结点 *C* 和 *D* 也是兄弟，但结点 *F* 和 *C* 不是兄弟结点。

4．祖先结点与后代结点（ancestor node 与 descendant node）

树中从根结点到某结点 *N* 所经过的所有结点，称作结点 *N* 的祖先结点。结点 *N* 的所有子结点，以及子结点的子结点构成结点 *N* 的后代结点。例如，结点 *B* 和 *A* 是 *C* 的祖先结点，结点 *H* 和 *J* 等则是 *A* 的后代结点。

5．结点的度和树的度（degree）

结点的度定义为结点所拥有子树的棵数，而树的度是指树中各结点度的最大值。例如图 6.2 (b) 中，结点 *A* 的度是 3，结点 *B* 的度是 2，结点 *C* 和 *D* 的度都是 0，整个树的度为 3。

6．叶子结点与分支结点（leaf node 与 branched node）

度为 0 的结点称为叶子结点，又称为终端结点。除叶子结点以外的其他结点称为分支结点，又称为非叶子结点或非终端结点。例如，结点 *C* 和 *D* 是叶子结点，*B*、*E* 和 *I* 是非叶子结点。

7．边（edge）

如果结点 *M* 是结点 *N* 的父结点，用一条线将这两个结点连接起来就构成树的一条分支，它称为连接这两个结点的边，该边可以用一个有序对<*M*，*N*>表示。例如在图 6.2 (b) 中，<*A*，*B*>和<*B*，*C*>都是树的边。

8．路径与路径长度（path 与 path length）

如果（N_1，N_2，…，N_k）是由树中的结点组成的一个序列，且<N_i，N_{i+1}>（$1 \leqslant i \leqslant k-1$）都是树的边，则该序列称为从 N_1 到 N_k 的一条路径。路径上边的数目称为该路径的长度。例如，从 *A* 到 *C* 的路径是（*A*，*B*，*C*），该路径的长度为 2。

9．结点的层次（level）和树的深度（depth）

如果根结点的层次定义为 0，它的子结点的层次则为 1，亦即，某结点的层次等于它的父结点的层次加 1，兄弟结点的层次相同。某结点的层次与从根结点到该结点的路径长度有关，树中结点的最大层次数称为树的深度（depth）或高度（height）。例如在图 6.2 (b) 中，*A* 的层次为 0，*B* 的层次为 1，*C* 的层次为 2。*C*、*F* 虽不是兄弟结点，但它们的层次相同，称为同一层上的结点；该树的深度为 2。

6.1.3　树的基本操作

树结构的基本操作有以下几种：

- Initialize：初始化。建立一个树实例并初始化它的结点集合和边的集合。
- AddNode /AddNodes：在树中设置、添加一个或若干个结点。
- Get/Set：访问。获取或设置树中的指定结点。
- Count：求树的结点个数。
- AddEdge：在树中设置、添加边，即结点之间的关联。
- Remove：删除。从树中删除一个数据结点及相关联的边。
- Contains/IndexOf：查找。在树中查找满足某种条件的结点（数据元素）。

- Traversal: 遍历。按某种次序访问树中的所有结点，并且每个结点恰好访问一次。
- Copy：复制。复制一个树。

6.2 二叉树的定义与实现

树结构可以分为无序树和有序树两种类型，有序树中最常用的是二叉树（binary tree）。

6.2.1 二叉树的定义

二叉树也可以用递归形式来定义：二叉树 BT 是由 n（$n \geqslant 0$）个结点组成的有限集合，它或者是棵空二叉树，或者包含一个根结点和两棵互不相交的子二叉树，子二叉树从左至右是有次序的，分别称为左子树和右子树。结点个数为零，即 $n=0$ 时称为空二叉树；结点数 $n>0$ 时，二叉树非空，由根结点及其两棵子二叉树构成。

从定义中可以看出，二叉树是一种特殊的树结构，树结构中定义的有关术语，如度、层次等，大都适用于二叉树。二叉树的结点最多只有两棵子树，所以二叉树的度最大为 2。但是，即使二叉树的度为 2，它与度为 2 的树在结构上也是不等价的，它们的区别在于：二叉树是一种有序树，因为二叉树中每个结点的两棵子树有左、右之分，即使只有一个非空子树，也要区分是左子树还是右子树，而普通的树结构指的是无序树。例如图 6.3 中的两棵树，如果看成是一般的树结构，则图（a）和图（b）表示同一棵树；如果看成是二叉树结构，则图（a）和图（b）表示两棵不同的二叉树。

(a) 树1　　(b) 树2

图 6.3 不同的二叉树与相同的度为 2 的树

由上述定义可知，二叉树有五种基本形态，如图 6.4 所示。

（a）表示空二叉树。

（b）表示只有一个结点（根结点）的二叉树。

（c）表示由根结点，非空的左子树和空的右子树组成的二叉树。

（d）表示由根结点，空的左子树和非空的右子树组成的二叉树。

（e）表示由根结点，非空的左子树和非空的右子树组成的二叉树。

其中，图 6.4（c）和（d）是两种不同形态的二叉树。

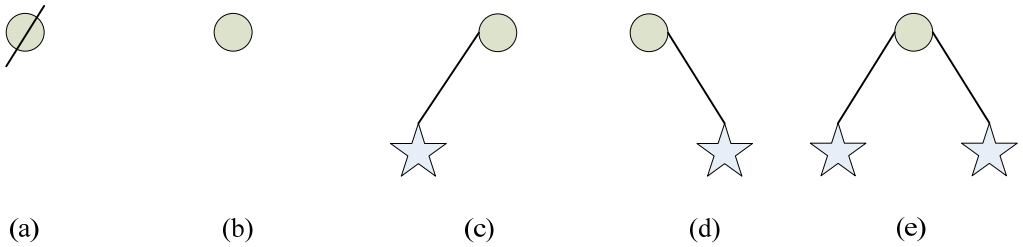
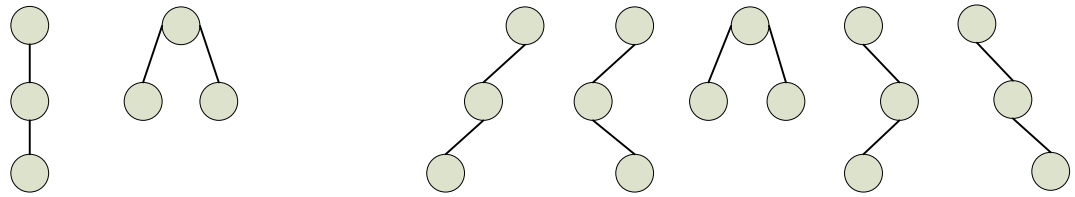


图 6.4 二叉树的基本形态

【例6.1】　画出有 3 个结点的树与二叉树的基本形态。

3 个结点的树只有如图 6.5（a）所示的 2 种基本形态；3 个结点的二叉树则可以有如图 6.5（b）所示的 5 种基本形态。



(a) 3个结点的树的2种基本形态　　(b) 3个结点的二叉树的5种基本形态

图 6.5　3 个结点的树与二叉树的基本形态

6.2.2　二叉树的性质

性质一：二叉树第 i 层的结点数目最多为 2^i（$i \geqslant 0$）。

这里，根结点的层次定义为 0，某结点的层次等于它的父结点的层次加 1。用归纳法容易证明这条性质。

当 i=0 时，根结点是 0 层上的唯一结点，故该层结点数为 $2^i=2^0=1$，命题成立。

假设命题对前 i-1（$i \geqslant 1$）层成立，即第 i-1 层上的最大结点数为 2^{i-1}。

归纳推理：根据假设，第 i-1 层上的最大结点数为 2^{i-1}；由于二叉树中每个结点的度最大为 2，故第 i 层上的最大结点数为 $2\times 2^{i-1}=2^i$。命题成立。

性质二：在深度为 k 的二叉树中，最多有 $2^{k+1}-1$ 个结点（$k \geqslant 0$）。

由性质一可知，在深度为 k 的二叉树中，最大结点数为 $\sum_{i=0}^{k} 2^i = 2^{k+1}-1$。

每一层的结点数目都达到最大值的二叉树称为满二叉树（full binary tree）。从定义可知，一棵深度为 k（$k \geqslant 0$）的满二叉树具有 2^{k+1}-1 个结点。

性质三：二叉树中，若叶子结点数为 n_0，2 度结点的数目为 n_2，则有 $n_0 = n_2+1$。

设二叉树的总结点数为 n，度为 1 的结点数为 n_1，则有

$$n = n_0 + n_1 + n_2$$

根结点不是任何结点的子结点，其他结点则会是某个结点的子结点，度为 1 的结点有 1 个子结点，

度为 2 的结点有 2 个子结点，叶子结点没有子结点，所以从二叉树的子结点数目的角度看，有一下关系：

$$n - 1 = 0 \times n_0 + 1 \times n_1 + 2 \times n_2$$

综合上述两式，可得 $n_0=n_2+1$，即二叉树中叶子结点数比度为 2 的结点数多 1。

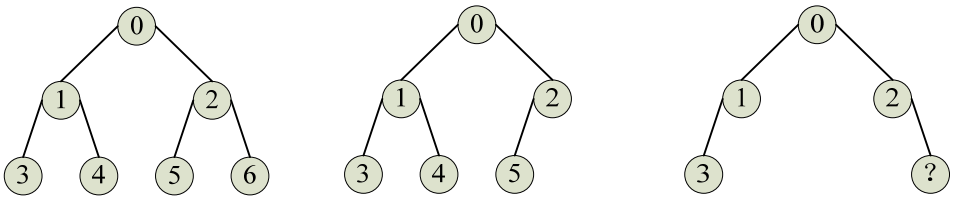
性质四：如果一棵完全二叉树有 n 个结点，则其深度 $k = \lfloor \log_2 n \rfloor$。

如前所述，深度为 k 的满二叉树具有 $2^{k+1}-1$（$k \geq 0$）个结点，我们可以对满二叉树的结点进行连续编号，并约定编号从根结点开始，自上而下，每层自左至右。一颗结点有编号的满二叉树如图 6.6（a）所示。

一棵具有 n 个结点、深度为 k 的二叉树，如果它的每个结点都与深度为 k 的满二叉树中编号为 0~n-1 的结点一一对应，则称这棵二叉树为完全二叉树（complete binary tree），如图 6.6（b）所示。

由定义可知，完全二叉树与满二叉树有相似的结构，两者之间具有下列关系：

- 满二叉树一定是完全二叉树，而完全二叉树不一定是满二叉树，它是具有满二叉树结构而不一定满的二叉树。完全二叉树只有最下面一层可以不满，其上各层都可看成满二叉树。
- 完全二叉树最下面一层的结点都集中在该层最左边的若干位置上，图 6.6（c）就不是一棵完全二叉树。
- 完全二叉树至多只有最下面两层结点的度可以小于 2。



(a) 满二叉树　　(b) 完全二叉树　　(c) 非完全二叉树

图 6.6 满二叉树与完全二叉树

性质五：若将一棵具有 n 个结点的完全二叉树的所有结点按自上而下、自左至右的顺序编号，结点编号 i 的取值范围为（0≤i≤n-1），结点编号存在下列规律：

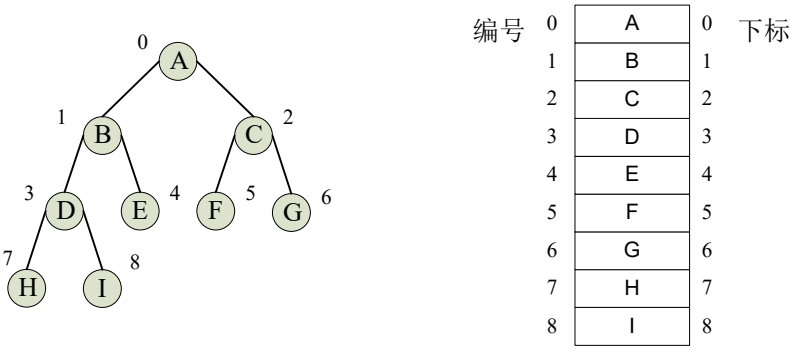
1) 若 i=0，则结点 i 为根结点，无父结点；若 $i \neq 0$，则结点 i 的父结点是编号为 $j = \left\lfloor \frac{i-1}{2} \right\rfloor$ 的结点。
2) 若 $2i+1 \leq n-1$，则结点 i 的左子结点是编号为 $2i+1$ 的结点；若 $2i+1 > n-1$，则结点 i 无左子结点。
3) 若 $2i+2 \leq n-1$，则结点 i 的右子结点是编号为 $2i+2$ 的结点；若 $2i+2 > n-1$，则结点 i 无右子结点。

6.2.3 二叉树的存储结构

在计算机中表示二叉树数据结构，可以用顺序存储结构和链式存储结构两种方式。二叉树结构具有层次关系，用链式存储结构来实现会更加灵活方便，所以一般情况下，采用链式存储结构来实现二叉树数据结构。顺序存储结构适用于完全二叉树，

1. 二叉树的顺序存储结构

完全二叉树可以用顺序存储结构实现。将完全二叉树的结点进行顺序编号，并将编号为 *i* 的结点存放在数组中下标为 *i* 的单元中。根据二叉树的性质五，对于结点 *i*，可以直接计算得到其父结点、左子结点和右子结点的位置。在图 6.7 中，一个完全二叉树的所有结点按顺序存放在一个数组中。



(a) 完全二叉树　　(b) 顺序存储完全二叉树

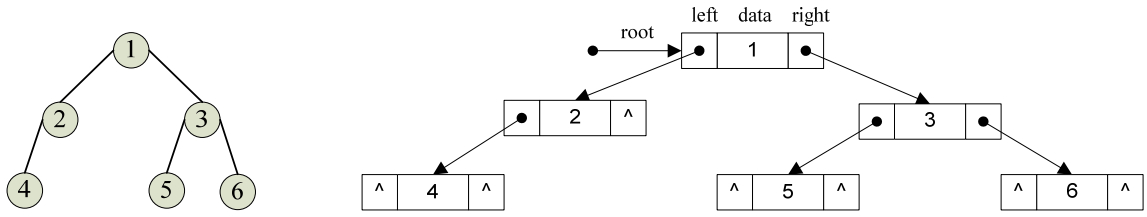
图 6.7　顺序存储结构的完全二叉树

2. 二叉树的链式存储结构

为了以链式存储结构实现二叉树，在逻辑上，二叉树的结点应有 3 个域：

- 数据域 data，表示结点的数据元素自身的内容；
- 左链域 left，指向该结点的左子结点；
- 右链域 right，指向该结点的右子结点。

二叉树的表示则需记录其根结点 root，若二叉树为空，则 `root` 置为 `null`。树中某结点的左子结点也代表该结点的左子二叉树，同理，该结点的右子结点代表它的右子二叉树。若结点的左子树为空，则其 left 链置为空值，即 `left=null`；若结点的右子树为空，则其 right 链置为空值，即 `right=null`。图 6.8 显示了一颗二叉树的链式存储结构。



(a) 二叉树　　(b) 链式存储结构

图 6.8　二叉树的链式存储结构

6.2.4　二叉树类的定义

1. 二叉树的结点类

为实现二叉树的链式存储结构，将二叉树的结点声明为 BinaryTreeNode<T>泛型类，其中有 3 个成员变量：数据域 data 表示结点的数据元素，链域 left 和 right 则分别指向左子结点和右子结点。3 个成员变量都是私有的，不能被其他类直接访问，通过添加相应的公有属性 Data，Left 和 Right 让外界访问

这些域成员。构造方法在创建一个结点时将它的值域初始化为缺省值或指定的值，而将链域 left 和 right 置为 null。

.NET Framework 和 C#编程语言 2.0 版开始增加了泛型类型，泛型利用类型参数将类型的指定推迟到声明并实例化该类对象的时候。使用泛型类型可以最大限度地重用代码、保护类型的安全以及提高性能，泛型最常见的用途是创建数据集合类，这里也用泛型定义二叉树这种数据集合类型。

```
public class BinaryTreeNode<T> {
    private T data;                                  //数据元素
    private BinaryTreeNode<T> left, right;           //指向左、右孩子结点的链

    public BinaryTreeNode(){
        left = right = null;
    }

    //构造有值结点
    public BinaryTreeNode(T d) {
        data = d;
        left = right = null;
    }

    public T Data {
        get { return data; }
        set { data = value; }
    }

    public BinaryTreeNode<T> Left {
        get { return left; }
        set { left = value; }
    }

    public BinaryTreeNode<T> Right {
        get { return right; }
        set { right = value;}
    }
}
```

2. 二叉树类

链式存储结构的二叉树用下面定义的 BinaryTree 类表示，它的成员变量 root 指向二叉树的根结点。

```
public class BinaryTree<T> {
    protected BinaryTreeNode<T> root;                //指向二叉树的根结点
    public BinaryTreeNode<T> Root {
        get { return root; }
        set { root = value; }
    }
```

```
    public BinaryTree() {                          //构造空二叉树
        root = null;
    }
}
```

上面设计的 BinaryTree 类和 BinaryTreeNode 类都声明在名字空间 DSAGL 中。

6.3　二叉树的遍历

6.3.1　二叉树遍历的过程

二叉树的遍历（traversal）操作就是按照一定规则和次序访问二叉树中的所有结点，并且每个结点仅被访问一次。通过这样一次完整的遍历操作，就按照指定的规则对二叉树中的所有结点形成一种线性次序的序列。所谓访问一个结点，可以是对该结点的数据元素进行探测、修改等操作。

二叉树的遍历过程，可以按层次的高低次序进行，即从根结点开始，逐层深入，同层从左至右依次访问结点。

二叉树是由根结点、左子树和右子树三个部分组成的，依次遍历这三个部分，便是遍历整个二叉树。若规定对子树的访问按“先左后右”的次序进行，则遍历二叉树有 3 种次序：

- 先根次序：访问根结点，遍历左子树，遍历右子树。
- 中根次序：遍历左子树，访问根结点，遍历右子树。
- 后根次序：遍历左子树，遍历右子树，访问根结点。

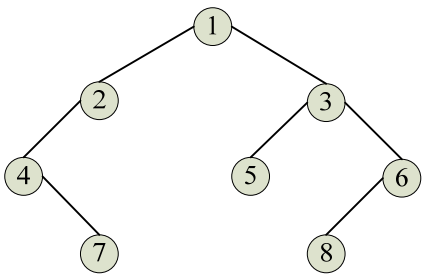
图 6.9 中显示了对二叉树进行 3 种不同次序遍历所产生的序列。以先根次序遍历二叉树为例，遍历过程如下：

若二叉树为空，则该操作为空操作，直接返回；否则从根结点开始，

1)　访问当前结点。

2)　若当前结点的左子树不空，则沿着 left 链进入该结点的左子树进行遍历操作。

3)　若当前结点的右子树不空，则沿着 right 链进入该结点的右子树进行遍历操作。



(1) 先根次序遍历序列：1 2 4 7 3 5 6 8
(2) 中根次序遍历序列：4 7 2 1 5 3 8 6
(3) 后根次序遍历序列：7 4 2 5 8 6 3 1

图 6.9　二叉树的遍历

依据二叉树的遍历规则，可以知道：根结点处在先根次序遍历序列的第一个位置，因为它是最先被访问的；而在后根次序遍历序列中，根结点是最后被访问的结点，所以根结点处在后根序列的最后一个位置；在中根次序遍历序列中，其左子树上的结点都排在根结点的前面，其右子树上的结点都排在根结点的后面。所以，先根次序或后根次序遍历序列能反映双亲与孩子结点的层次关系，中根次序遍历序列能反映兄弟结点间的左右次序。

6.3.2 二叉树遍历的递归算法

1. 先根次序遍历二叉树的递归算法

按先根次序遍历一颗二叉树的递归算法如下：
若二叉树为空，则该操作为空操作，直接返回；否则从根结点开始，
1) 访问当前结点；
2) 按先根次序遍历当前结点的左子树；
3) 按先根次序遍历当前结点的右子树。
在二叉树结点类 BinaryTreeNode 中，增加按先根次序遍历以某结点为根的二叉树的递归方法，编码如下所示。该类中定义的 ShowPreOrder 方法在控制台显示按先根次序遍历二叉树得到的结点序列的值，而 TraversalPreOrder 方法更一般化，它将各结点的值按先根次序存放在一个线性表中，其参数 sql 可以是数组或线性表 List 类型。

```
//输出本结点为根结点的二叉树，先根次序
public void ShowPreOrder() {
    Console.Write(this.Data + " ");
    BinaryTreeNode<T> q = this.Left;
    if (q != null)
        q.ShowPreOrder();
    q = this.Right;
    if (q != null)
        q.ShowPreOrder();
}

//先根次序遍历以本结点为根结点的二叉树，将各结点的值存放在表sql中
public void TraversalPreOrder(IList<T> sql) {
    sql.Add(this.Data);
    BinaryTreeNode<T> q = this.Left;
    if (q != null)
        q.TraversalPreOrder(sql);
    q = this.Right;
    if (q != null)
        q.TraversalPreOrder(sql);
}
```

2. 中根次序遍历二叉树的递归算法

按中根次序遍历一颗二叉树的递归算法如下：
若二叉树为空，则该操作为空操作，直接返回；否则从根结点开始，
1) 按中根次序遍历当前结点的左子树；
2) 访问当前结点；
3) 按中根次序遍历当前结点的右子树。

在二叉树结点类 BinaryTreeNode 中，增加按中根次序遍历以某结点为根的二叉树的递归方法，编码如下。该类的 ShowInOrder 方法在控制台显示按中根次序遍历二叉树得到的结点序列的值，而 TraversalInOrder 方法更一般化，它将各结点的值按指定的次序存放在一个数组或线性表中。

```
public void ShowInOrder() {
    BinaryTreeNode<T> q = this.Left;
    if (q != null)
        q.ShowInOrder();
    Console.Write(this.Data + " ");
    q = this.Right;
    if (q != null)
        q.ShowInOrder();
}

public void TraversalInOrder(IList<T> sql) {
    BinaryTreeNode<T> q = this.Left;
    if (q != null)
        q.TraversalInOrder(sql);
    sql.Add(this.Data);
    q = this.Right;
    if (q != null)
        q.TraversalInOrder(sql);
}
```

3. 按后根次序遍历二叉树的递归算法

按后根次序遍历一颗二叉树的递归算法如下：

若二叉树为空，则该操作为空操作，直接返回；否则从根结点开始，

1) 按后根次序遍历当前结点的左子树；
2) 按后根次序遍历当前结点的右子树；
3) 访问当前结点。

在二叉树结点类 BinaryTreeNode 中，增加按后根次序遍历以某结点为根的二叉树的递归方法，编码如下。该类的 ShowPostOrder 方法在控制台显示按后根次序遍历二叉树得到的结点序列的值，而 TraversalPostOrder 方法更一般化，它将各结点的值按指定的次序存放在一个数组或线性表中。

```
public void ShowPostOrder() {
    BinaryTreeNode<T> q = this.Left;
    if (q != null)
        q.ShowPostOrder();
    q = this.Right;
    if (q != null)
        q.ShowPostOrder();
    Console.Write(this.Data + " ");

}

public void TraversalPostOrder(IList<T> sql) {
```

```
        BinaryTreeNode<T> q = this.Left;
        if (q != null)
            q.TraversalPostOrder(sql);
        q = this.Right;
        if (q != null)
            q.TraversalPostOrder(sql);
        sql.Add(this.Data);
    }
```

4. 从根结点遍历整个二叉树

在二叉树类 BinaryTree 的定义中，增加如下的 6 个方法，每一对 Show/Traversal 方法，分别调用二叉树结点类 BinaryTreeNode 中实现的按相应次序遍历二叉树的递归方法，遍历从根结点开始的整个二叉树。

```
//先根次序遍历二叉树
public void ShowPreOrder() {
    Console.Write("先根次序:   ");
    if (root != null)
        root.ShowPreOrder();
    Console.WriteLine();
}
public List<T> TraversalPreOrder() {
    List<T> sql = new List<T>();
    if (root != null)
        root.TraversalPreOrder(sql);
    return sql;
}

//中根次序遍历二叉树
public void ShowInOrder() {
    Console.Write("中根次序:   ");
    if (root != null)
        root.ShowInOrder();
    Console.WriteLine();
}
public List<T> TraversalInOrder() {
    List<T> sql = new List<T>();
    if (root != null)
        root.TraversalInOrder(sql);
    return sql;
}
```

```
//后根次序遍历二叉树
public void ShowPostOrder() {
    Console.Write("后根次序:   ");
    if (root != null)
        root.ShowPostOrder();
    Console.WriteLine();
}
public List<T> TraversalPostOrder() {
    List<T> sql = new List<T>();
    if (root != null)
        root.TraversalPostOrder(sql);
    return sql;
}
```

【例6.2】　按先根、中根和后根次序遍历二叉树。

程序 BinaryTreeTest.cs 利用前面定义的 BinaryTree 类，先建立如图 6.9 所示的二叉树，然后以先根、中根和后根次序遍历二叉树。程序还演示了 BinaryTree 类的泛型能力，即在二叉树实例化时决定结点数据的类型。

```
using DSAGL;
class BinaryTreeTest {
   static void Main(string[] args) {
       BinaryTree<int> btree = new BinaryTree<int>();
       BinaryTreeNode<int>[] nodes = new BinaryTreeNode<int>[9];
       for (int i = 1; i <= 8; i++)
           nodes[i] = new BinaryTreeNode<int>(i);
       btree.Root = nodes[1];
       nodes[1].Left = nodes[2]; nodes[1].Right = nodes[3];
       nodes[2].Left = nodes[4];
       nodes[3].Left = nodes[5]; nodes[3].Right = nodes[6];
       nodes[4].Right = nodes[7];
       nodes[6].Left = nodes[8];
       btree.ShowPreOrder(); btree.ShowInOrder(); btree.ShowPostOrder();//显示不同的遍历序列

       BinaryTree<string> btree2 = new BinaryTree<string>();
       btree2.Root = new BinaryTreeNode<string>("大学");
       btree2.Root.Left = new BinaryTreeNode<string>("学院1");
       btree2.Root.Right = new BinaryTreeNode<string>("学院2");
       btree2.Root.Left.Left = new BinaryTreeNode<string>("C#课程");
       btree2.Root.Right.Left = new BinaryTreeNode<string>("教师1");
       btree2.Root.Right.Right = new BinaryTreeNode<string>("OS课程");
       btree2.Root.Left.Left.Right = new BinaryTreeNode<string>("学生1");
       btree2.Root.Right.Right.Left = new BinaryTreeNode<string>("教师2");
       btree2.ShowPreOrder(); btree2.ShowInOrder(); btree2.ShowPostOrder();//显示不同的遍历序列
```

```
    }
}
```

程序运行结果如下：

```
先根次序：  1 2 4 7 3 5 6 8
中根次序：  4 7 2 1 5 3 8 6
后根次序：  7 4 2 5 8 6 3 1

先根次序：  大学  学院1  C#课程  学生1  学院2  教师1  OS课程  教师2
中根次序：  C#课程  学生1  学院1  大学  教师1  学院2  教师2  OS课程
后根次序：  学生1  C#课程  学院1  教师1  教师2  OS课程  学院2  大学
```

上面以递归方式实现了二叉树的遍历操作，递归方式的思路直接清晰，但是算法的空间复杂度和时间复杂度，相比非递归方式增加了许多。

6.3.3 二叉树遍历的非递归算法

二叉树的遍历操作也可以用非递归算法实现，下面以中根次序遍历过程为例讨论二叉树遍历操作的非递归实现算法。以中根次序遍历二叉树的规则是：遍历左子树，访问根结点，遍历右子树。按照该规则，在每个结点处，先选择遍历左子树，当左子树遍历完后，必须返回到该结点，访问该结点后，再遍历右子树。但是二叉树中的任何结点均只包含指向子结点的链，而没有指向其父结点的链。在中根次序遍历的非递归实现算法中，通过设定一个栈来暂存经过的路径。

二叉树中根次序遍历的非递归算法具体过程描述如下：设置一个栈 s。设结点变量 *p*，初始指向二叉树的根结点。如果 *p* 不空或栈不空时，循环执行以下操作，直到扫描完二叉树且栈为空。

1) 如果 *p* 不空，表示扫描到一个结点，将 *p* 结点入栈（Push），进入其左子树。
2) 如果 *p* 为空并且栈不空，表示已走过一条路径，此时必须返回一步以寻找另一条路径。而要返回的结点就是栈中记录的最后一个结点，它已保存在栈顶，所以设置 *p* 指向从 s 出栈（即 p=s.Pop()）的结点，访问 *p* 结点，再进入 *p* 的右子树。

中序遍历非递归算法程序代码如下（定义在二叉树 BinaryTree 类中）：

```
//非递归中根次序遍历二叉树
public void ShowInOrderNR() {
    Stack<BinaryTreeNode<T>> s = new Stack<BinaryTreeNode<T>>(100);
    BinaryTreeNode<T> p = root;
    Console.Write("非递归中根次序：  ");
    while (p != null || s.Count != 0) {     //p非空或栈非空时
        if (p != null) {
            s.Push(p);                      //p结点入栈
            p = p.Left;                     //进入左子树
        }
        else {                              //p为空且栈非空时
            p = s.Pop();                    //p指向出栈的结点
            Console.Write(p.Data + " ");    //访问结点
            p = p.Right;                    //进入右子树
        }
```

```
            }
            Console.WriteLine();
        }
```

在上面的代码中，设计一个栈类型 s 对象，栈的元素类型是二叉树结点类 BinaryTreeNode<T>（泛型），使用泛型类型可以最大限度地重用代码、保护类型的安全以及提高性能。

对于如图 6.9 所示的二叉树，非递归中根遍历时栈中内容的变化如图 6.10 所示。

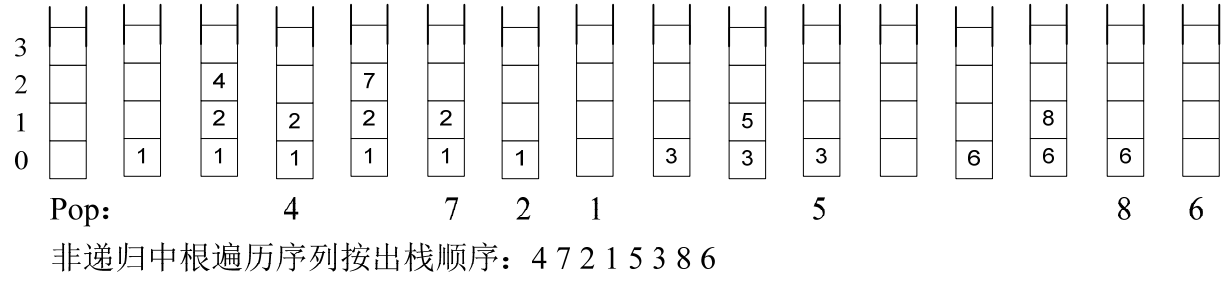


图 6.10　非递归中根遍历二叉树时栈中内容的变化

在例 6.2 中增加对实现中根次序遍历非递归算法的方法的调用，即增加下列语句：

```
btree.ShowInOrderNR();
```

得到的结果与中根次序遍历的递归算法的结果是一样的。一般而言，非递归算法的时间和空间效率都要高一些。

6.3.4　按层次遍历二叉树

二叉树也可以按结点的层次高低次序进行遍历，即从根结点开始，逐层深入，而在同一层次则从左至右依次访问结点。如图 6.9 所示的二叉树，按层次遍历规则，首先访问根结点 1，再访问根结点的孩子结点 2 和结点 3，然后应该访问结点 2 的孩子结点 4，再访问结点 3 的孩子 5 结点，依次类推，因此这颗二叉树的层次遍历序列为 1，2，3，4，5，6，7，8。

在二叉树的链式存储结构中，每个结点中保存有指向其子结点的两条链，但没有指向其他结点的链，包括同层其他结点和下一层其他结点。所以在图 6.9 中，从根结点 1 可以到达结点 2 和结点 3，而从结点 3 却无法到达下一层的结点 4，从结点 4 也无法到达同层的结点 5。要完成这些结点间的跳转，必须设立辅助的数据结构，用来指示下一个要访问的结点。如果结点 2 在结点 3 之前访问，则结点 2 的孩子结点均在结点 3 的孩子结点之前访问。因此，辅助结构应该选择具有“先进先出”特点的队列。

按层次遍历二叉树的算法具体过程描述如下：设置一个队列变量 *q*；设结点变量 *p*，初始指向二叉树的根结点。当 *p* 不空时，循环顺序执行以下操作，直至 *p* == null，循环停止：

1)　访问 *p* 结点；
2)　如果 *p* 的 left 链不空，将 *p* 结点的左子结点加入队列 q（入队操作 q.Enqueue(p.Left)）；
3)　如果 *p* 的 right 链不空，将 *p* 结点的右子结点加入队列 q（入队操作 q.Enqueue(p.Right)）；
4)　如果队列为非空，设置 *p* 指向从队列 q 出队的结点（即 p=q.Dequeue()），否则设置 *p* 指向 null。

按层次遍历二叉树算法的程序代码如下：

```
//按层次遍历二叉树
public void ShowByLevel() {
    Queue<BinaryTreeNode<T>> q = new Queue<BinaryTreeNode<T>>(100); //设立一个空队列
```

```
        BinaryTreeNode<T> p = root; Console.Write("层次遍历：  ");
        while (p != null) {
            Console.Write(p.Data + " ");
            if (p.Left != null)  q.Enqueue(p.Left);          //p的左孩子结点入队
            if (p.Right != null) q.Enqueue(p.Right);         //p的右孩子结点入队
            if (q.Count != 0)
                p = q.Dequeue();                             //当队列不空，p指向出队的结点
            else
                p = null;                                    //当队列为空，p置为null
        }
        Console.WriteLine();
    }
```

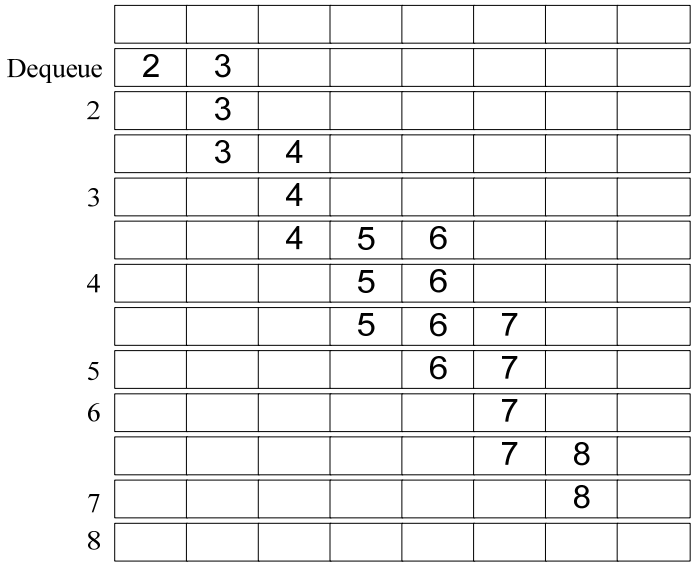
在该代码中，设计了一个队列 q，队列元素是泛型二叉树结点类 BinaryTreeNode<T>。对于如图 6.9 所示的二叉树，按层次遍历二叉树时，队列状态的变化如图 6.11 所示。

在例 6.2 中增加对实现层次遍历二叉树算法的方法的调用，即增加下列语句：

```
btree. ShowByLevel();
```

运行结果如下：

```
按层次遍历：  1 2 3 4 5 6 7 8
```



按层次遍历序列=根+出队序列，即：12345678

图 6.11 按层次遍历二叉树时队列内容的变化

6.3.5 建立二叉树

给定一定的条件，可以唯一地建立一个二叉树实例。例如对于完全二叉树，如果各结点的元素值按顺序存储在一个数组中，则可以利用二叉树的性质五，唯一地建立链式存储结构来表示这颗二叉树。

一般情况下，由于二叉树是数据元素之间具有层次关系的非线性结构，而且二叉树中每个结点的

两个子树有左右之分，这样就要求，必须满足以下两个条件，才能明确地建立一棵二叉树：

- 结点与其父结点及子结点间的层次关系是明确的。
- 兄弟结点间的左右顺序关系是明确的。

二叉树可以用广义表形式来表示，但广义表形式有时不能唯一表示一棵二叉树，原因在于它无法明确左右子树。可以定义一种特殊形式的广义表表示式来唯一描述二叉树，例如在二叉树的广义表表示式中既标明非空子树，也清楚地标明空子树，按照这样一种特殊的广义表表示式，可以唯一地建立一颗二叉树。

对于给定的一棵二叉树，遍历产生的先根、中根、后根序列是唯一的；反之，已知二叉树的一种遍历序列，并不能唯一确定一棵二叉树。因为遍历序列仅是二叉树结构在某种条件下映射成的线性序列。先根次序或后根次序反映双亲与孩子结点的层次关系，中根次序反映兄弟结点间的左右次序。所以，已知先根和中根两种遍历序列，或中根和后根两种遍历序列才能够唯一确定一棵二叉树，而已知先根和后根两种遍历序列仍无法唯一确定一棵二叉树。

1. 建立链式存储结构的完全二叉树

对于一棵其结点已经顺序存储在一个数组中的完全二叉树（如图 6.7 所示），由二叉树的性质五可知，第 0 个结点为根结点，第 i 个结点的左子结点或为第 $2i+1$ 个结点，或为空结点，它的右子结点或为第 $2i+2$ 个结点，或为空结点。

在二叉树类 BinaryTree<T>的定义中，增加静态方法 ByOneList，它的参数 t 是一个线性表或数组，用以表示顺序存储的完全二叉树结点值的序列。程序如下：

```
public static BinaryTree<T> ByOneList(IList<T> t) {
    int n = t.Count;
    BinaryTree<T> bt = new BinaryTree<T>();
    if (n == 0) {
        bt.Root = null;
        return bt;
    }
    int i, j;
    BinaryTreeNode<T>[] q = new BinaryTreeNode<T>[n];
    T v;
    for (i = 0; i < n; i++) {
        v = t[i];                                   //取编号为i的结点值
        q[i] = new BinaryTreeNode<T>(v);
    }
    for (i = 0; i < n; i++) {
        j = 2 * i + 1;
        if (j < n)
            q[i].Left = q[j];
        else
            q[i].Left = null;
        j++;
        if (j < n)
            q[i].Right = q[j];
        else
```

```
                q[i].Right = null;
            }
            bt.Root = q[0];
            return bt;
        }
```

【例6.3】　根据给定数组建立链式存储结构的完全二叉树。

程序 ByOneListTest.cs 建立链式存储结构的完全二叉树。

```
using DSAGL;
static void Main(string[] args) {
    int[] it = { 0,1,2,3,4,5,6,7};
    BinaryTree<int> btree = BinaryTree<int>.ByOneList(it);
    btree.ShowPreOrder(); btree.ShowInOrder();
    btree.ShowPostOrder(); btree.ShowByLevel();
    char[] ct = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H' };
    BinaryTree<char> btree2 = BinaryTree<char>.ByOneList(ct);
    btree2.ShowPreOrder();  btree2.ShowInOrder();
}
```

程序建立如图 6.7 所示的完全二叉树，它的运行结果如下：

```
先根次序：  0 1 3 7 4 2 5 6
中根次序：  7 3 1 4 0 5 2 6
先根次序：  A B D H E C F G
中根次序：  H D B E A F C G
```

2. 以广义表表示式建立二叉树

第五章介绍过以广义表形式可以表示树结构，但广义表形式有时不能唯一表示一棵二叉树，原因在于它无法明确左右子树。例如，广义表 A(B)没有表达出结点 B 是结点 A 的左子结点还是右子结点，A(B)表达式可以对应两棵二叉树。为了唯一地表示一棵二叉树，必须重新定义广义表的形式。

在广义表表示式中，除数据元素外还需要定义四个边界符号：

1. 空子树符 NullSubtree，如可用‘^’表示，以标明非叶子结点的空子树。
2. 左界符（起始界符）LeftDelimit，如常用‘(’表示，以标明下一层次的左（起始）边界；
3. 右界符（结束界符）RightDelimit，如常用‘)’表示，以标明下一层次的右（结束）边界。
4. 中界符 MiddleDelimit，如常用‘,’，以标明某一层次的左右子树的分界。

这样，如图 6.9 所示的二叉树用广义表形式就可以表示为：1(2(4(^,7),^),3(5,6(8,^)))。反之，给定一棵二叉树的广义表表示式，则能够唯一确定一棵二叉树。

以广义表表示式建立二叉树的算法描述为：

依次读取二叉树的广义表表示序列中的每个元素，检查其内容，如果

- 遇到有效数据值，则建立一个二叉树结点对象；扫描到下一元素，
  - 如果它为 LeftDelimit，则这个 LeftDelimit 和下一个 RightDelimit 之间是该结点的左子树与右子树，再递归调用建树算法，分别建立左、右子树，返回结点对象。
  - 如果没有遇到 LeftDelimit，表示该结点是叶子结点。
- 遇到 NullSubtree，表示空子树，返回 null 值。

在二叉树类 BinaryTree<T>的定义中， 增加静态方法 ByOneList，它的第一个参数表示顺序存储的广义表表示式，第二个参数定义广义表表示式所用的分界符。程序如下：

```
public static BinaryTree<T> ByOneList(IList<T> sList, ListFlagsStruc<T> ListFlags) {
    BinaryTree<T>.ListFlags = ListFlags;
    BinaryTree<T>.idx = 0;                                  // 初始化递归变量
    BinaryTree<T> bt = new BinaryTree<T>();
    if (sList.Count > 0)
        bt.Root = RootByOneList(sList);
    else
        bt.Root = null;
    return bt;
}

private static BinaryTreeNode<T> RootByOneList(IList<T> sList) {
    BinaryTreeNode<T> p = null;
    T nodeData = sList[idx];
    if (isData(nodeData)) {
        p = new BinaryTreeNode<T>(nodeData);                // 有效数据，建立结点
        idx++;
        nodeData = sList[idx];
        if (nodeData.Equals(ListFlags.LeftDelimit)) {
            idx++;                                          //左边界，如'('，跳过
            p.Left = RootByOneList(sList);                  //建立左子树，递归
            idx++;                                          //跳过中界符，如','
            p.Right = RootByOneList(sList);                 //建立右子树，递归
            idx++;                                          //跳过右边界，如')'
        }
    }
    if (nodeData.Equals(ListFlags.NullSubtree)) idx++; //空子树符，跳过，返回null
    return p;
}

private static bool isData(T nodeValue) {
    if (nodeValue.Equals(ListFlags.NullSubtree)) return false;
    if (nodeValue.Equals(ListFlags.LeftDelimit)) return false;
    if (nodeValue.Equals(ListFlags.RightDelimit)) return false;
    if (nodeValue.Equals(ListFlags.MiddleDelimit)) return false;
    else return true;
}
```

在二叉树类 BinaryTree<T>中， 增加静态私有成员变量 idx 的定义，表示递归处理广义表表达式的当前位置；增加私有成员变量 ListFlags，记录广义表所用的分界符：

```
private static ListFlagsStruc<T> ListFlags;
private static int idx = 0;
```

ListFlags 定义为结构类型 ListFlagsStruc<T>：

```
public struct ListFlagsStruc<T> {
    public T NullSubtree;
    public T LeftDelimit;
    public T RightDelimit;
    public T MiddleDelimit;
}
```

【例6.4】　根据给定的广义表表示式来建立一颗二叉树。

下面的程序通过提供一个广义表表示式来建立一颗二叉树。

```
static void Main(string[] args) {
    string s = "1(2(4(^,7),^),3(5,6(8,^)))";
    Console.WriteLine("Generalized List: " + s);
    ListFlagsStruc<char> ListFlags;
    ListFlags.NullSubtree = '^'; ListFlags.LeftDelimit = '(';
    ListFlags.RightDelimit = ')'; ListFlags.MiddleDelimit = ',';
    BinaryTree<char> btree = BinaryTree<char>.ByOneList(s.ToCharArray(0, s.Length), ListFlags);
    btree.ShowPreOrder(); btree.ShowInOrder();
}
```

程序建立如图 6.9 所示的二叉树，它的运行结果如下：

```
Generalized List: 1(2(4(^,7),^),3(5,6(8)))
先根次序：  1 2 4 7 3 5 6 8
中根次序：  4 7 2 1 5 3 8 6
```

3. 按先根和中根次序遍历序列建立二叉树

已知二叉树的一种遍历序列，并不能唯一确定一棵二叉树。如果已知二叉树的先根和中根两种遍历序列，或中根和后根两种遍历序列，则可唯一地确定一棵二叉树。

设二叉树的先根及中根次序遍历序列分别存储在线性表或数组 preList 和 inList 中，建立二叉树的算法描述如下：

1) 确定根元素。由先根次序遍历序列知，二叉树的根结点的值为 rootData = preList[0]。然后按值查找它在中根次序遍历序列 inList 中的位置 k：

   k = inList.IndexOf(rootData);

2) 确定根的左子树的相关序列。由中根次序知，根结点 inList[k]之前的结点在根的左子树上，根结点 inList[k]之后的结点在根的右子树上。因此，根的左子树由 k 个结点组成，它的特征是：
   - ✧ 先根序列——preList [1]，…，preList [k]。
   - ✧ 中根序列——inList [0] ，…，inList [k-1]。

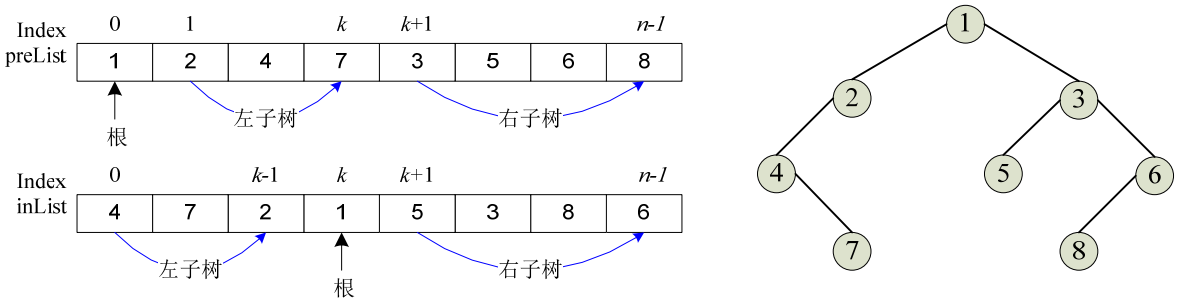
3) 根据左子树的先根序列和中根序列建立左子树，这是一种递归方式。

4) 确定根的右子树的相关序列。根的右子树由 n-k-1 个结点组成，它的特征是：
   - ✧ 先根序列——preList[k+1]，…，preList[n-1]。
   - ✧ 中根序列——inList[k+1] ，…，inList[n-1]。

其中 n= preList.Count，即已知数据序列的长度。

5)　根据右子树的先根序列和中根序列建立右子树，这也是一种递归方式。

图 6.12 显示了一个根据给定的先根和中根次序遍历序列来建立相应的二叉树的过程，假设先根次序遍历序列为 preList = { 1, 2, 4, 7, 3, 5, 6, 8}，中根次序遍历序列为 inlist = {4, 7, 2, 1, 5, 3, 8, 6}。



(a) 先根与中根遍历序列　　(b) 所建立的二叉树

图 6.12 按先根和中根次序遍历序列建立二叉树的过程

同理可证明：按中根与后根次序遍历序列可唯一确定一棵二叉树。

在二叉树类 BinaryTree<T>的定义中， 增加静态方法 ByTwoList，它的第一个参数表示先根次序遍历序列，第二个参数表示中根次序遍历序列，程序如下：

```
public static BinaryTree<T> ByTwoList(IList<T> preList, IList<T> inList) {
    BinaryTree<T> bt = new BinaryTree<T>();
    bt.Root = RootByTwoList(preList, inList);
    return bt;
}

private static BinaryTreeNode<T> RootByTwoList(IList<T> preList, IList<T> inList) {
    BinaryTreeNode<T> p = null;
    T rootData;
     int i, k, n;
    IList<T> presub = new List<T>();            // 当前子树先根序列
    IList<T> insub = new List<T>();             // 当前子树中根序列
    n = preList.Count;
    if (n > 0) {
        rootData = preList[0];                  // 当前根结点
        p = new BinaryTreeNode<T>(rootData);
        k = inList.IndexOf(rootData);           // 当前根在中根序列的位置
        Console.WriteLine("\t current root = " + rootData + "\t k=" + k);
        for (i = 0; i < k; i++)                 // 准备当前根结点的左子树先根序列
            presub.Add(preList[1 + i]);
        for (i = 0; i < k; i++)                 // 准备当前根结点的左子树中根序列
            insub.Add(inList[i]);
        p.Left = RootByTwoList(presub, insub);  // 建立当前根结点的左子树，递归
```

```
            presub.Clear();
            for (i = 0; i < n - k - 1; i++)          // 准备当前根结点的右子树先根序列
                presub.Add(preList[k + 1 + i]);
            insub.Clear();
            for (i = 0; i < n - k - 1; i++)          // 准备当前根结点的右子树中根序列
                insub.Add(inList[k + 1 + i]);
            p.Right = RootByTwoList(presub, insub); // 建立当前根结点的右子树，递归
        }
        return p;
    }
```

【例6.5】　按先根和中根次序遍历序列建立二叉树。

程序 ByTwoListTest.cs 调用 BinaryTree 类，以先根和中根次序遍历序列建立一颗二叉树。

```
using System; using DSAGL;
namespace treetest {
    class ByTwoListTest {
        static void Main(string[] args) {
            int[] prelist = { 1, 2, 4, 7, 3, 5, 6, 8};
            int[] inlist = {4, 7, 2, 1, 5, 3, 8, 6};
            BinaryTree<int> btree = BinaryTree<int>.ByTwoList(prelist, inlist);
            btree.ShowPreOrder(); btree.ShowInOrder();
        }
    }
}
```

程序建立如图 6.12（b）所示的二叉树，它的运行结果如下：

```
        current root = 1        k=3
        current root = 2        k=2
        current root = 4        k=0
        current root = 7        k=0
        current root = 3        k=1
        current root = 5        k=0
        current root = 6        k=1
        current root = 8        k=0
先根次序:   1 2 4 7 3 5 6 8
中根次序:   4 7 2 1 5 3 8 6
```

6.4　线索二叉树

从上节的讨论可知：二叉树的遍历将得到一个二叉树结点集合按一定规则排列的线性序列，在这个遍历序列中，除第一个和最后一个结点外，每个结点有且只有一个前驱结点和一个后继结点。在上

节定义的二叉树的链式存储结构中，每个结点很容易到达其左、右子结点，而不能直接到达该结点在任意一个遍历序列中的前驱或后继结点，这种信息只能在遍历的动态过程中才能得到。下面介绍的线索树结构，在不增加很多存储空间的条件下，能够存储遍历过程中得到的信息，并在后续的使用中解决直接访问前驱结点和后继结点的问题。

6.4.1　线索与线索二叉树

在二叉树中，每个结点有两个链；具有 n 个结点的二叉树总共有 $2n$ 个链。若某结点的左（右）子树为空，则左（右）链就为 *null* 值。在二叉树的所有 $2n$ 个链中，只需要 n-1 个链来指明各结点间的关系，其余 n+1 个链均为空值。在某种遍历过程中，可以利用这些空链来指明结点在该种遍历次序下的前驱和后继结点，这些指向前驱或后继结点的链称为线索。对二叉树进行遍历并加上线索的过程称为二叉树的线索化，线索化了的二叉树就构成线索二叉树，相应地按先（中、后）根次序进行线索化的二叉树称为先（中、后）序**线索二叉树**。

线索二叉树中，原先非空的链保持不变，仍然指向该结点的左、右子结点，它记录的是结点间的层次关系。原先空的左链用来指向遍历中该结点的前驱结点，原先空的右链指向后继结点，它记录的是结点间在遍历时的顺序关系。为了区别每条链是否是一个线索，可以在二叉树的结点结构中设置两个状态字段 lefttag 和 righttag，用以标记相应链的状态。

因此线索二叉树的结点结构由 5 个域构成：data, left, right, lefttag 和 righttag，其中 lefttag 和 righttag 的作用如下：

$$\text{lefttag} = \begin{cases} \text{true} & \text{left为线索，指向前驱结点} \\ \text{false} & \text{left为普通链，指向左子结点} \end{cases}$$

$$\text{righttag} = \begin{cases} \text{true} & \text{right为线索，指向后继结点} \\ \text{false} & \text{right为普通链，指向右子结点} \end{cases}$$

图 6.13 给出中序线索二叉树的示意图，图中虚线表示线索。结点 7 的前驱是 4，后继是 2。4 的 left 链为空，它的 lefttag 为 true，表明它的 left 链为线索，但它没有前驱；6 的 right 链为空，它的 righttag 为 true，表明它的 right 链为线索，但它没有后继。可见，在线索二叉树中可以利用线索直接找到结点的前驱或后继结点。

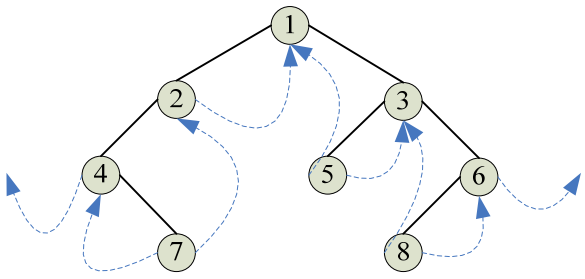


图 6.13　中序线索二叉树

6.4.2　线索二叉树类的实现

1. 定义线索二叉树结点类

下面声明的 ThreadBinaryTreeNode<T>类表示线索二叉树结点结构，其中有 5 个成员变量：data 用于表示数据元素，left 和 right 分别是指向左、右孩子结点的链，lefttag 和 righttag 为线索标记。5 个成员变量都是私有的，不能被其他类直接访问。我们通过添加相应的公有属性 Data，Left，Right，LeftTag 和 RightTag 访问这些域成员。线索二叉树结点类也是自引用的泛型类，结点的值类型在对象实例化时决定。

```
public class ThreadBinaryTreeNode<T> {
    private T data;                                  //数据元素
    private ThreadBinaryTreeNode<T> left, right;     //指向左、右孩子结点的链
    private bool lefttag;                            //左线索标志
    private bool righttag;                           //右线索标志

    public ThreadBinaryTreeNode() {
        left = right = null;
        lefttag = righttag = false;
    }

    //构造有值结点
    public ThreadBinaryTreeNode(T d) {
        data = d;
        left = right = null;
        lefttag = righttag = false;
    }
```

2. 定义线索二叉树类

下面声明的 ThreadBinaryTree 类表示线索二叉树，其中成员变量 root 指向二叉树的根结点。

```
public class ThreadBinaryTree<T> {
    private ThreadBinaryTreeNode<T> front = null;
    protected ThreadBinaryTreeNode<T> root;              //指向二叉树的根结点

    public ThreadBinaryTreeNode<T> Root {
        get { return root; }
        set { root = value; }
    }
    public ThreadBinaryTree() {                          //构造空二叉树
        root = null;
    }
    ......
}
```

6.4.3 二叉树的中序线索化

二叉树的中序线索化可以用递归方法实现，其算法描述如下：

设 *p* 指向一棵二叉树的某个结点，front 指向 *p* 的前驱结点，它的初值为 *null*。当 *p* 非空时，执行

以下操作：

- 中序线索化 *p* 结点的左子树。
- 如果 *p* 的左子树为空，设置 *p* 的 lefttag 标记为 true，它的 left 链为指向前驱结点 front 的线索。
- 如果 *p* 的右子树为空，设置 *p* 的 righttag 标记为 true
- 如果前驱结点 front 非空并且它的右链为线索，设置 front 的 right 链为指向 *p* 的线索。
- 移动 front，使 front 指向 *p*。
- 中序线索化 *p* 结点的右子树。

如果一开始让 *p* 指向二叉树的根结点 root，则上述过程线索化整个二叉树。

在线索二叉树 ThreadBinaryTree 类中，增加以下方法对二叉树进行中序线索化。

```
//中序线索化以p结点为根的子树
private void SetThreadInOrder(ThreadBinaryTreeNode<T> p) {
    if (p != null) {
        SetThreadInOrder(p.Left);           //中序线索化p的左子树
        if (p.Left == null) {               //p的左子树为空时,设置p.left为指向front的线索
            p.LeftTag = true;
            p.Left = front;
        }
        if (p.Right == null)                //p的右子树为空时
            p.RightTag = true;              //设置p.RightTag为线索的标志
        if (front != null && front.RightTag)
            front.Right = p;                //设置front.right为指向p的线索
        front = p;
        SetThreadInOrder(p.Right);          //中序线索化p的右子树
    }
}

//中序线索化二叉树
public void SetThreadInOrder() {
    front = null;
    SetThreadInOrder(root);
}
```

6.4.4　线索二叉树的遍历

对线索二叉树进行遍历，无需用上节介绍的遍历方法遍历整个二叉树，而是利用线索查找到某结点在遍历序列中的前驱或后继结点来遍历二叉树。下面以中序线索二叉树为例，讨论它的中根次序遍历和先根次序遍历算法。我们将看到，在中序线索二叉树中，可以方便地查找中根、先根和后根次序下的后继结点，因此能够以先根、中根和后根次序遍历中序线索二叉树。

1. 中序线索二叉树中查找中根次序的后继结点

根据中根次序遍历二叉树的规则，在中序线索二叉树中查找中根次序的后继结点的过程描述如下：

- 设 *p* 指向当前结点，执行以下操作：

- ✧ 如果 *p* 结点的右子树为线索，则 *p* 的 right 链为其后继结点，设置 *p* 为该结点。
- ✧ 否则说明 *p* 的右子树为非空，则 *p* 的后继结点是 *p* 的右子树上第一个中序访问的结点，即 *p* 的右孩子的最左边的子孙结点，设置 *p* 为该结点。

➢ 返回 p，作为当前结点在中根次序下的后继结点。

在线索二叉树结点类 ThreadBinaryTreeNode 中，增加 NextNodeInOrder 方法，以查找某结点在中根次序下的后继结点。

```
public ThreadBinaryTreeNode<T> NextNodeInOrder() {
    ThreadBinaryTreeNode<T>  p = this;
    if (p.RightTag)                              //右子树为空时
        p = p.Right;                             //right指向后继结点
    else {                                       //右子树非空时
        p = p.Right;                             //进入右子树
        while (!p.LeftTag)                       //找到最左边的子孙结点
            p = p.Left;
    }
    return p;
}
```

2. 中序线索二叉树的中根次序遍历

中根次序遍历中序线索二叉树可以用非递归方法实现，其算法描述如下：

➢ 寻找第一个访问结点。它是根的左子树上最左边的子孙结点，用 *p* 指向该结点。

➢ 访问 *p* 结点。

➢ 找到 *p* 的后继结点，用 *p* 指向该结点，跳转到上一步，直至 *p* 为 null，说明已访问了序列的最后一个结点。

上述步骤遍历整棵二叉树。在线索二叉树 ThreadBinaryTree 类中，增加 ShowUsingThreadInOrder 方法，它首先找到遍历序列的第一个结点，然后调用结点类中的 NextNodeInOrder 方法依次找到后继结点，在中序线索二叉树中完成中根次序遍历。

```
public void ShowUsingThreadInOrder() {
    ThreadBinaryTreeNode<T> p = root;
    if (p != null) {
        Console.Write("中根次序：  ");
        while (!p.LeftTag)
            p = p.Left;          //找到根的最左边子孙结点
        do {
            Console.Write(p.Data + " ");
            p = p.NextNodeInOrder();                  //返回p的后继结点
        } while (p != null);
        Console.WriteLine();
    }
}
```

【例6.6】　中序线索二叉树的线索化与中序遍历。

程序 ThreadBinaryTreeTest.cs 调用 ThreadBinaryTree 类，演示中序线索二叉树的线索化方法与中序

遍历方法的调用。

```
class ThreadBinaryTreeTest {
    public static void Main(string[] args) {
        ThreadBinaryTree<int> tbt = new ThreadBinaryTree<int>(); //建立空二叉树
        ThreadBinaryTreeNode<int>[] nodes = new ThreadBinaryTreeNode<int>[9];
        for (int i = 1; i <= 8; i++)                    //建立二叉树的结点
            nodes[i] = new ThreadBinaryTreeNode<int>(i);
        tbt.Root = nodes[1];                            //设置二叉树的结点结构
        nodes[1].Left = nodes[2]; nodes[1].Right = nodes[3];
        nodes[2].Left = nodes[4];
        nodes[3].Left = nodes[5]; nodes[3].Right = nodes[6];
        nodes[4].Right = nodes[7];nodes[6].Left = nodes[8];
        tbt.SetThreadInOrder();                         //中序线索化二叉树
        tbt.ShowUsingThreadInOrder();                   //中根次序遍历中序线索二叉树
        tbt.ShowUsingThreadPreOrder();                  //先根次序遍历中序线索二叉树
    }
}
```

程序建立如图 6.13 所示的二叉树，它的运行结果如下：

```
中根次序：  4 7 2 1 5 3 8 6
先根次序：  1 2 4 7 3 5 6 8
```

3. 中序线索二叉树中查找先根次序的后继结点

根据先根次序遍历二叉树的规则，在中序线索二叉树中查找先根次序的后继结点的过程描述如下：

- 设 p 指向当前结点，执行以下操作：
  - 如果 p 结点的左子树为非空，则 p 的左孩子为其后继结点，设置 p 为该结点。
  - 否则说明 p 的左子树为空，如果 p 的右子树为非空，则 p 的右孩子即为其后继结点，设置 p 为该结点。
  - 如果 p 结点的左、右子树均为空，说明它是叶子结点，则 p 的后继结点为它的中序线索祖先的右孩子，沿着右线索可以找到它，设置 p 为该结点。
- 返回 p，作为当前结点在先根次序下的后继结点。

在线索二叉树结点类 ThreadBinaryTreeNode 中，增加 NextNodePreOrder 方法，实现上述算法。

```
public ThreadBinaryTreeNode<T> NextNodePreOrder() {
    ThreadBinaryTreeNode<T> p = this;
    if (!p.LeftTag)                         //左子树非空时，左孩子就是p的后继结点
        p = p.Left;
    else {
        if (!p.RightTag)                    //左子树为空而右子树非空时，右孩子是p的后继结点
            p = p.Right;
        else {
            while (p.RightTag && p.Right!= null)   //叶子结点,后继是其中序线索祖先的右孩子
                p = p.Right;
            p = p.Right;
        }
```

```
        }
        return p;
    }
```

4. 中序线索二叉树的先根次序遍历

先根次序遍历中序线索二叉树可以用非递归方法实现，其算法描述如下：

- 寻找第一个访问结点。它是根结点，用 p 指向该结点。
- 访问 p 结点。
- 找到 p 的后继结点，用 p 指向该结点，跳转到上一步，直至 p 为 null，说明已访问了序列的最后一个结点。

上述步骤遍历整棵二叉树。在线索二叉树 ThreadBinaryTree 类中，增加 ShowUsingThreadPreOrder 方法，它将根结点作为遍历序列的第一个结点，然后调用结点类中的 NextNodePreOrder 方法依次找到后继结点，在中序线索二叉树中完成先根次序遍历。

```
    public void ShowUsingThreadPreOrder() {
        ThreadBinaryTreeNode<T> p = root;
        if (p != null) {
            Console.Write("先根次序:  ");
            do {
                Console.Write(p.Data + " ");
                p = p.NextNodePreOrder();
            } while (p != null);
            Console.WriteLine();
        }
    }
```

6.5 用二叉树表示树与森林

二叉树是一种特殊的树，它的实现相对容易，一般的树和森林实现起来相对困难一些，树和森林可以转换为二叉树进行处理。

树中的结点可能有多个子结点，所以一般的树需要用多重链表结构来实现。对于具有 n 个结点的、度为 k 的树，如果每个结点用 k 个链指向孩子结点，则一棵树总的链数为 $n \times k$，其中只有 n-1 个非空的链指向除根以外的 n-1 个结点，其余的链都是空链。在树的这种多重链表存储结构中，空链数与总链数之比为：

$$\frac{\text{空链数}}{\text{总链数}} = \frac{nk-(n-1)}{n \times k} \approx 1 - \frac{1}{k}$$

例如，当 k=20 时，空链比为 95%。由此可见，这样的多重链表存储结构的存储密度在有的情况下是很低的，常造成大量存储空间的浪费。

通常，可以用一种所谓的“孩子—兄弟”存储结构将一棵树转换成了一棵二叉树。用来表示树结构的二叉树结点有 3 个域：

1) 数据域 data，存放结点数据。
2) 左链域 child，指向该结点的第一个孩子结点。
3) 右链域 brother，指向该结点的下一个兄弟结点。

对于给定的一棵树，按照以上规则，可以得到唯一的二叉树表达式，也就是有唯一的一棵二叉树与原树结构相对应。由于树的根结点没有兄弟结点，所以相应的二叉树表达式中的根结点没有右子树。

这种形式的二叉树也可以用来表示森林，即森林可以转化成一棵二叉树来存储，其转化过程如下：

1)　将森林中的每棵树转化成二叉树。

2)　用每棵树的根结点的 brother 链将若干棵树的二叉树表示式连接成一棵单独、完整的二叉树。

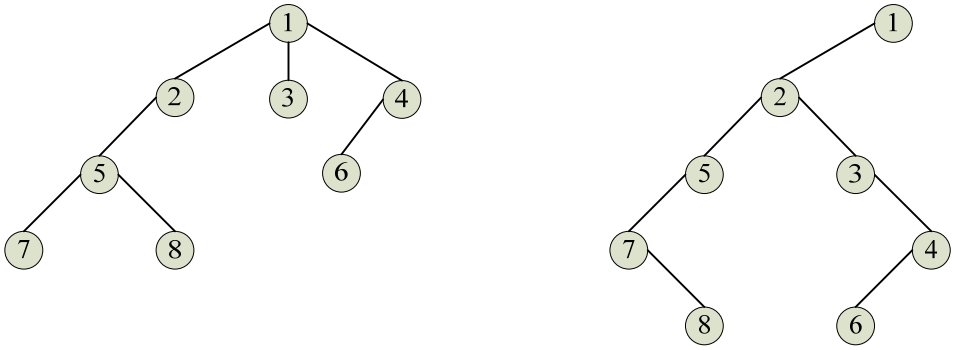
图 6.14 显示了用二叉树表示树和森林的例子，图(a)是一棵树及其对应的二叉树，图(b)是森林及其对应的二叉树。

二叉树也可以还原为树，其方法为：

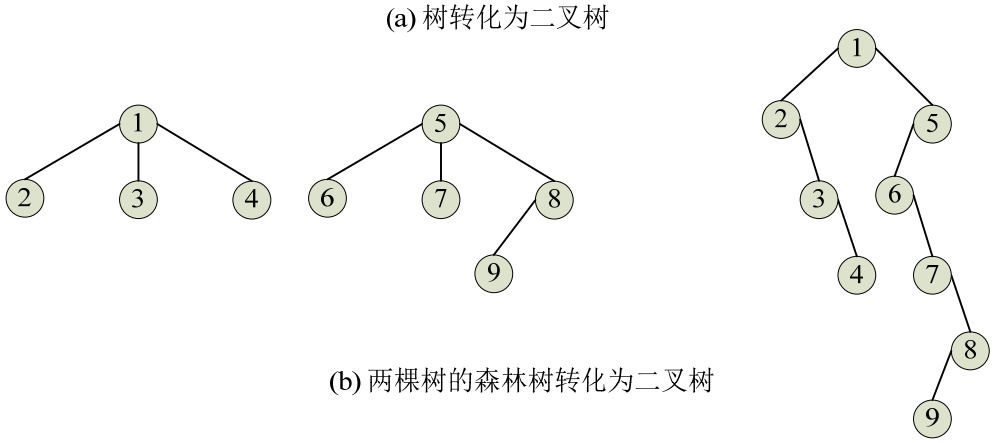
1)　删除原二叉树中所有父结点与右孩子的连线。

2)　若某结点是其父结点的左孩子，则把该结点的右孩子、右孩子的右孩子等等都与该结点的父结点用线连起来。

3)　整理所有保留的和添加的连线，使每个结点的所有子结点位于同一层次。



(a) 树转化为二叉树



(b) 两棵树的森林树转化为二叉树

图 6. 14　树和森林转化为二叉树

习题 6

6.1 填空

1)　一棵深度为 5 的满二叉树有_____个分支结点和 _____个叶子。

2)　一棵具有 257 个结点的完全二叉树，它的深度为_____。

3)　设一棵完全二叉树具有 1000 个结点，则此完全二叉树有_____个叶子结点，有_____个度为 2

的结点，有_____个结点只有非空左子树，有_____个结点只有非空右子树。

4) 设一棵满二叉树共有 2*N*-1 个结点，则它的叶结点数_____。

6.2 简述二叉树与度为 2 的树的差别。

6.3 对于如图 6.15 所示的二叉树，求先、中、后三种次序的遍历序列。

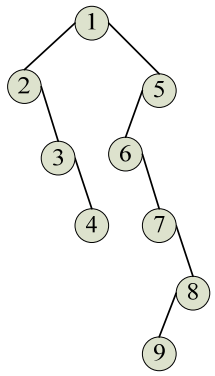


图 6.15 二叉树

6.4 什么样的非空二叉树，它的先根与后根次序遍历序列相同？

6.5 二叉树 T，已知其先根遍历是 1 2 3（数字为结点的编号，以下同），后根遍历是 3 2 1，分析二叉树 T 所有可能的中根遍历。

6.6 讨论下列关于二叉树的一些操作的实现策略：

1) 统计二叉树的结点个数。

2) 求某结点的层次。

3) 找出二叉树中值大于 *k* 的结点。

4) 输出二叉树的叶子结点。

5) 将二叉树中所有结点的左右子树相互交换。

6) 求一棵二叉树的高度。

7) 验证二叉树的性质二：$n_0=n_2+1$。

6.7 线索二叉树的建立：

1) 直接建立线索二叉树，即在建树的同时进行线索化。

2) 对图 6.15 所示的二叉树，分别以先序和中序进行线索化。

6.8 把如图 6.16 所示的树转化成二叉树。

6.9 画出与图 6.17 所示的二叉树相应的森林。

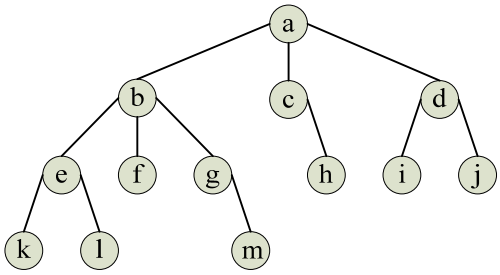


图 6.16 二叉树

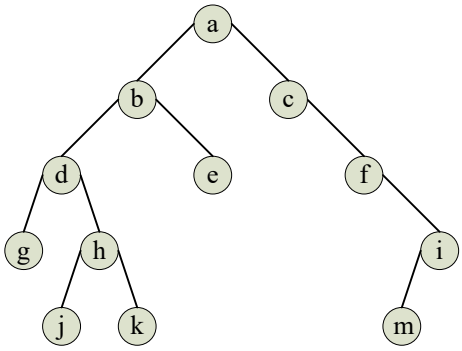


图 6.17 二叉树

6.10 解决问题的策略常用树结构来描述。有 8 枚硬币，其中恰有一枚假币，假币比真币重。现欲用一架天平称出假币，使称重的次数尽可能地少。试以树结构描述测试假币的称重策略。