第 5 章　树

到目前为止，我们已经学习了线性结构和表结构。这些数据结构一般不适合于描述具有分支结构的数据。在这种数据之间可能有祖先—后代、上级—下属、整体—部分等分支的关系。本章引入的树形结构则是以分支关系定义的层次结构，是一类重要的非线性数据结构，在计算机领域有着广泛应用。例如，在文件系统和数据库系统中，树是组织信息的重要形式之一；在编译系统中，树用来表示源程序的语法结构；在算法设计与分析中，树还是刻画程序动态性质的工具。本章讨论树、二叉树、森林，还要讨论堆与优先级队列，以及树的应用实例，即 Huffman 树。

5.1　树的基本概念

树结构广泛存在于现实世界，例如，家族的家谱、公司的组织机构、书的章节等。在计算机应用中，最为人们熟悉的就是磁盘中的文件夹，即文件目录。它包含文件和文件夹。

5.1.1　树的定义和术语

为了完整地建立有关树的基本概念，以下给出两种树的定义，即自由树（见图 5.1）和有根有序树。

自由树(free tree)。一棵自由树 T_f 可定义为一个二元组 $T_f=(V,E)$，其中 $V=\{v_1, v_2, \cdots, v_n\}$ 是由 n ($n>0$) 个元素组成的有限非空集合，称为顶点(vertex)集合，v_i($1\leqslant i\leqslant n$)称为顶点。$E=\{(v_i, v_j) \mid v_i, v_j\in V, 1\leqslant i, j\leqslant n\}$ 是由 $n-1$ 个元素组成的序对集合，称为边集合，E 中的元素 (v_i, v_j)称为边(edge)或分支(branch)。E 使得 T_f 成为一个连通图(参见图论的有关定义)。

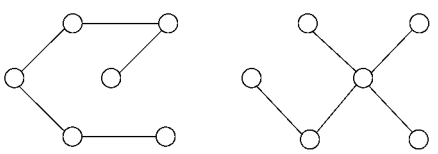


图 5.1　自由树

有关自由树的研究是图论讨论的主要内容之一，本书不讨论。本章讨论的树是有根有序树，它们的顶点统一称为结点(node)。

有根树(rooted tree)。一棵有根树 T，简称树，它是 n($n\geqslant 0$)个结点的有限集合。当 $n=0$ 时，T 称为空树；否则，T 是非空树，记作

$$T=\begin{cases}\phi, & n=0\\ \{r, T_1, T_2, \cdots, T_m\}, & n>0\end{cases}$$

其中，r 是 T 的一个特殊结点，称为根(root)。T_1, T_2, …, T_m 是除 r 之外其他结点构成的互不相交的 m($m\geqslant 0$) 个子集，每个子集也是一棵树，称为根的子树(subtree)。

每棵子树的根结点有且仅有一个直接前驱(即它的上层结点)，但可以有 0 个或多个直接后继(即它的下层结点)。m 称为 r 的分支数。

图 5.2 给出树的逻辑表示，它形如一棵倒长的树。图 5.2(a)是空树，一个结点也没有。

图 5.2(b)是只有一个根结点的树,它的子树为空。图 5.2(c)是有 13 个结点的树,其中 A 是根结点,它一般都画在树的顶部。其余结点分成 3 个互不相交的子集:$T_1=\{B, E, F, K, L\}$,$T_2=\{C, G\}$,$T_3=\{D, H, I, J, M\}$。它们都是根结点 A 的子树。再看 T_1,它的根是 B,其余结点又分成 2 个互不相交的子集 $T_{11}=\{E, K, L\}$,$T_{12}=\{F\}$,它们是 T_1 的子树。

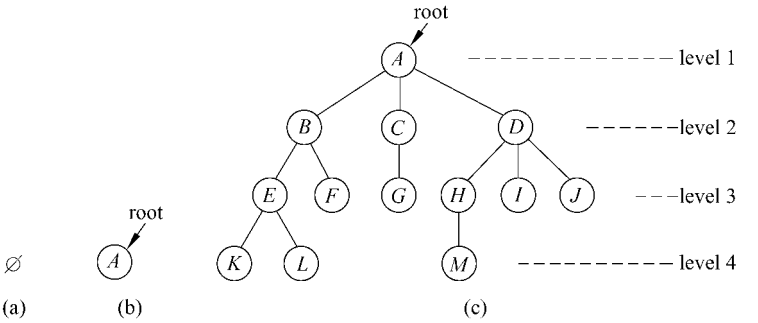
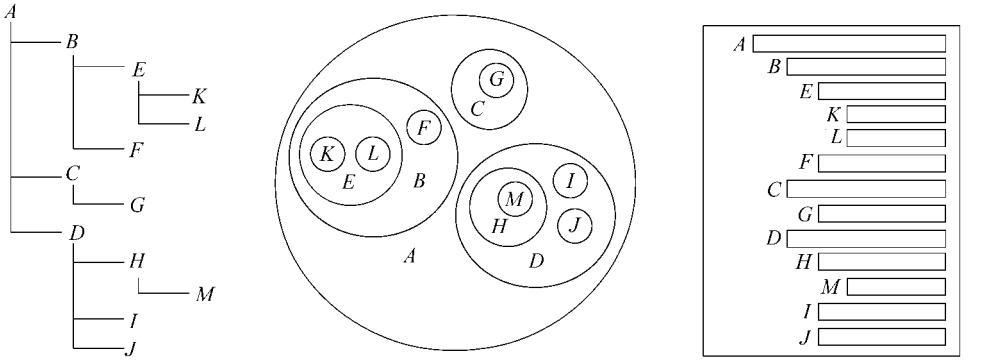


图 5.2　树的示意图

由此可知,树的定义是一个递归的定义,即树的定义中又用到了树的概念。

除了图 5.2 所描述的逻辑表示外,在计算机系统和其他领域还有几种表示方法,如图 5.3 所示。图 5.3(a)是树的目录结构表示;图 5.3(b)是树的集合文氏图表示;图 5.3(c)是树的凹入表表示;图 5.3(d) 是树的广义表表示。



(a) 目录结构表示　　(b) 集合文氏图表示　　(c) 凹入表表示

$$A(\underbrace{B(E(K,L),F)}_{T_1},\underbrace{C(G)}_{T_2},\underbrace{D(H(M),I,J)}_{T_3})$$

(d)广义表表示

图 5.3　树的其他表示方法

下面介绍有关树的一些术语,它们在本书后续章节将经常遇到。

(1) **结点**(node)。它包含数据项及指向其他结点的分支。例如在图 5.2(c) 中的树总共 13 个结点。为方便起见,每个数据项用单个字母表示。

(2) **结点的度**(degree)。结点所拥有的子树棵数。例如在图 5.2(c) 所示的树中,根 A

的度为 3，结点 E 的度为 2，结点 K，L，F，G，M，I，J 的度为 0。

(3) **叶结点**(leaf)。即度为 0 的结点，又称终端结点。例如，在图 5.2(c) 所示的树中，{K，L，F，G，M，I，J} 构成树的叶结点的集合。

(4) **分支结点**(branch)。除叶结点外的其他结点，又称非终端结点。例如在图 5.2(c) 所示的树中，A，B，C，D，E，H 就是分支结点。

(5) **子女结点**(child)。若结点 x 有子树，则子树的根结点即为结点 x 的子女。例如在图 5.2(c)所示的树中，结点 A 有 3 个子女，结点 B 有 2 个子女，结点 L 没有子女。

(6) **父结点**(parent)。若结点 x 有子女，它即为子女的父结点。例如在图 5.2(c)所示的树中，结点 B，C，D，E 有一个父结点，根结点 A 没有父结点。

(7) **兄弟结点**(sibling)。同一父结点的子女互称为兄弟。例如在图 5.2(c)所示的树中，结点 B，C，D 为兄弟，结点 E，F 也为兄弟，但结点 F，G，H 不是兄弟结点。

(8) **祖先结点**(ancestor)。从根结点到该结点所经分支上的所有结点。例如在图 5.2(c)所示的树中，结点 L 的祖先为 A，B，E。

(9) **子孙结点**(descendant)。某一结点的子女，以及这些子女的子女都是该结点的子孙。例如在图 5.2(c)所示的树中，结点 B 的子孙为 E，F，K，L。

(10) **结点所处层次**(level)。简称结点的层次，即从根到该结点所经路径上的分支条数。例如在图 5.2(c)所示的树中，根结点在第 1 层，它的子女在第 2 层。树中任一结点的层次为它的父结点的层次加 1。结点所处层次也称结点的深度。

(11) **树的深度**(depth)。树中距离根结点最远的结点所处层次即为树的深度。空树的深度为 0，只有一个根结点的树的深度为 1，图 5.2(c) 所示的树的深度为 4。

(12) **树的高度**(height)。很多数据结构教科书定义树的高度等同于树的深度，本书则从下向上定义高度。叶结点的高度为 1，非叶结点的高度等于它的子女结点高度的最大值加 1，这样可定义树的高度等于根结点的高度。高度与深度计算的方向不同，但数值相等。

(13) **树的度**(degree)。树中结点的度的最大值。例如，图 5.2(c) 所示的树的度为 3。

(14) **有序树**(ordered tree)。树中结点的各棵子树 T_1，T_2…是有次序的，即为有序树。其中，T_1 为根的第 1 棵子树，T_2 为根的第 2 棵子树……。

(15) **无序树**(unordered tree)。树中结点的各棵子树之间的次序是不重要的，可以互相交换位置。

(16) **森林**(forest)。是 $m(m\geqslant 0)$ 棵树的集合。在自然界，树与森林是两个不同的概念，但在数据结构中，它们之间的差别很小。删除一棵非空树的根结点，树就变成森林(不排除空的森林)；反之，若增加一个根结点，让森林中每棵树的根结点都变成它的子女，森林就成为一棵树。

5.1.2 树的抽象数据类型

下面给出树的抽象数据类型。利用树的抽象数据类型中提供的操作，就能实现许多应用问题的解法。

程序 5.1 树的抽象数据类型

```
class Tree {
//对象：树是由 n (≥0)个结点组成的有限集合。在类界面中的 position 是树中结点的地址，
```

```
//在顺序存储方式下是下标型，在链接存储方式下是指针型。T 是结点中存放数据的类型，要
//求所有结点的数据类型都是一致的
public:
    position Root();                          //返回根结点地址，若树为空，则返回 0
    BuildRoot(const T& value);                //建立树的根结点
    position FirstChild(position p);          //返回 p 第一个子女地址，无子女返回 0
    position NextSibling(position p);         //返回 p 下一兄弟地址，若无下一兄弟返回 0
    position Parent(position p);              //返回 p 父结点地址，若 p 为根返回 0
    T getData(position p);                    //函数返回结点 p 中存放的值
    bool InsertChild(const position p, const T &value);
    //在结点 p 下插入值为 value 的新子女，若插入失败，函数返回 false，否则返回 true
    bool DeleteChild(position p, int i);
    //删除结点 p 的第 i 个子女及其全部子孙结点，若删除失败，则函数返回 false；若删除
    //成功，则函数返回 true
    void DeleteSubTree(position t);           //删除以 t 为根结点的子树
    bool IsEmpty();                           //判断树是否为空。若空则返回 true
    void Traversal(void ( * visit)(position p));
    //遍历，visit 是用户自编的访问结点 p 数据的函数
};
```

5.2 二 叉 树

二叉树(binary tree)是树形结构的一种重要类型。

5.2.1 二叉树的定义

二叉树。这个定义也是以递归形式给出的：一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。

$$T=\begin{cases}\phi, & n=0\\ \{r,T_{\mathrm{L}},T_{\mathrm{R}}\}, & n>0\end{cases}$$

二叉树的子树 T_{L}、T_{R}仍是二叉树，到达空子树时递归的定义结束。许多基于二叉树的算法都利用了这个递归的特性。

二叉树的特点是每个结点最多有两个子女，分别称为该结点的左子女和右子女。就是说，**在二叉树中不存在度大于 2 的结点，并且二叉树的子树有左、右之分，其子树的次序不能颠倒**。因此，二叉树是分支数最大不超过 2 的有根有序树。它可能有 5 种不同的形态，如图 5.4 所示。图 5.4 (a)表示一棵空二叉树；图 5.4(b)是只有根结点的二叉树；根的左子树和右子树都是空的；图 5.4(c)是根的右子树为空的二叉树；图 5.4(d)是根的左子树为空的二叉树；

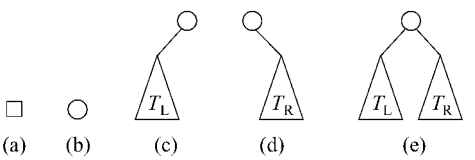


图 5.4 二叉树的 5 种不同形态

图 5.4(e)是根的两棵子树都不为空的二叉树。二叉树的任意形状都是基于这 5 种形态经过组合或嵌套而形成的。

关于树的术语对于二叉树都适用。

5.2.2 二叉树的性质

二叉树具有如下性质：

性质 5.1 在二叉树的第 $i(i\geqslant 1)$层最多有 2^{i-1}个结点。

【证明】 用归纳法：当 $i=1$ 时，非空二叉树在第 1 层只有一个根结点，$2^{1-1}=2^0=1$，结论成立；现假定对于所有的 j，$1\leqslant j<i$，结论成立，即第 j 层上至多有 2^{j-1}个结点，则在第 $i-1$ 层至多有 2^{i-2}个结点。由于二叉树每个结点最多有 2 个子女，因此第 i 层上的最大结点数为第 $i-1$ 层上最大结点数的 2 倍，即 $2\times 2^{i-2}=2^{i-1}$个结点，性质成立。

性质 5.2 深度为 $k(k\geqslant 0)$的二叉树最少有 k 个结点，最多有 2^k-1 个结点。

【证明】 因为每层最少要有 1 个结点，因此，最少结点数为 k。$k=0$ 是空二叉树的情形，此时一个结点也没有，$2^0-1=0$，结论成立。$k\geqslant 1$ 是非空二叉树情形，具有层次 $i=1,2,\cdots,k$。根据性质 5.1，第 i 层最多有 2^{i-1}个结点，则整个二叉树中所具有的最大结点数为

$$\sum_{i=1}^{k}(\text{第 } i \text{ 层上最大结点数})=\sum_{i=1}^{k}2^{i-1}=2^k-1$$

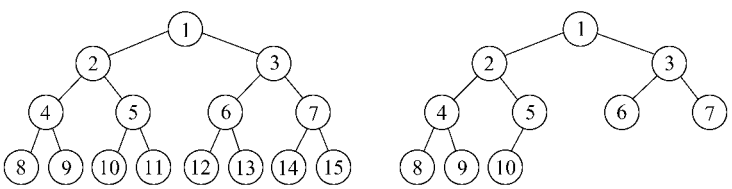
性质 5.3 对任何一棵非空二叉树，如果其叶结点数为 n_0，度为 2 的非叶结点数为 n_2，则

$$n_0=n_2+1$$

【证明】 设二叉树中度为 1 的结点数为 n_1，因为二叉树只有度为 0、度为 1 和度为 2 的结点，所以树中结点总数为 $n=n_0+n_1+n_2$。再看二叉树中边(分支)条数 e。因为二叉树中除根结点没有父结点，进入它的边数为 0 之外，其他每一结点都有一个且仅有一个父结点，进入它们的边数均为 1，故二叉树中总的边数为 $e=n-1=n_0+n_1+n_2-1$。又由于每个度为 2 的结点发出 2 条边，每个度为 1 的结点发出 1 条边，度为 0 的结点发出 0 条边，因此总的边数为 $e=2n_2+n_1$。将两个关于边的等式等同起来，有 $n_0+n_1+n_2-1=2n_2+n_1$，消去 n_1和一个 n_2，得 $n_0-1=n_2$，即 $n_0=n_2+1$，结论成立。

其他一些性质是有关某些特殊二叉树的。为此，先定义两种特殊的二叉树。

满二叉树(full binary tree)。深度为 k 的满二叉树是有 2^k-1 个结点的二叉树。在满二叉树中，每层结点都达到了最大个数。除最底层结点的度为 0 外，其他各层结点的度都为 2。图 5.5 (a)给出的就是高度为 4 的满二叉树。



(a) 满二叉树　　(b) 完全二叉树

图 5.5　两种特殊的二叉树

完全二叉树(complete binary tree)　如果一棵具有 n 个结点的深度为 k 的二叉树,它的每个结点都与深度为 k 的满二叉树中编号为 1～n 的结点一一对应,则称这棵二叉树为完全二叉树。图 5.5(b)给出的就是深度为 4 的完全二叉树。其特点:上面从第 1～$k-1$ 层的所有各层的结点数都是满的,仅最下面第 k 层或是满的,或从右向左连续缺若干结点。

性质 5.4　具有 n 个结点的完全二叉树的深度为$\lceil \log_2(n+1) \rceil$。

【证明】　由性质 5.2,深度为 k 的完全二叉树最多结点个数 $n \leqslant 2^k-1$,最少结点个数 $n > 2^{k-1}-1$,因此

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

移项得

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

取对数得

$$k-1 < \log_2(n+1) \leqslant k$$

因为 $\log_2(n+1)$ 介于 $k-1$ 和 k 之间且不等于 $k-1$,深度又只能是整数,因此有

$$k = \lceil \log_2(n+1) \rceil$$

结论成立。

注意,此性质针对完全二叉树给出。但对如图 5.6 所示的二叉树也适合。这种二叉树称为理想平衡二叉树,其第 1～$k-1$ 层也是满的,达到最大结点个数,第 k 层的结点不一定集中在最左边,可能分布在该层的各处。

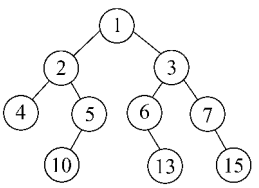


图 5.6　理想平衡二叉树

有的教科书定义 $k = \lfloor \log_2 n \rfloor + 1$,这与上述定义在 $n > 0$ 时等效,但 $n = 0$ 时不可用。

性质 5.5　如果将一棵有 n 个结点的完全二叉树自顶向下,同一层自左向右连续给结点编号 1, 2, 3, …, n,然后按此结点编号将树中各结点顺序地存放于一个一维数组中,并简称编号为 i 的结点为结点 i ($1 \leqslant i \leqslant n$),如图 5.5(b)所示。则有以下关系。

(1) 若 $i == 1$,则结点 i 为根,无父结点;若 $i > 1$,则结点 i 的父结点为结点$\lfloor i/2 \rfloor$。

(2) 若 $2 \times i <= n$,则结点 i 的左子女为结点 $2 \times i$。

(3) 若 $2 \times i + 1 <= n$,则结点 i 的右子女为结点 $2 \times i + 1$。

(4) 若结点编号 i 为奇数,且 $i != 1$,它处于右兄弟位置,则它的左兄弟为结点 $i-1$。

(5) 若结点编号 i 为偶数,且 $i != n$,它处于左兄弟位置,则它的右兄弟为结点 $i+1$。

(6) 结点 i 所在层次为$\lfloor \log_2 i \rfloor + 1$。

5.2.3　二叉树的抽象数据类型

下面给出二叉树的抽象数据类型。此定义只给出了二叉树操作的一个最小集合。可以以它为基础增加其他相关的操作。

程序 5.2　二叉树的抽象数据类型

```
class BinaryTree {                          //对象:结点的有限集合,二叉树是有序树
public:
    int Height();                           //返回树的深度或高度
```

```
    int Size();                                          //返回树中结点个数
    bool IsEmpty();                                      //判断二叉树是否为空
    BinTreeNode * Parent(BinTreeNode * current);         //求指定结点的父结点
    BinTreeNode * LeftChild(BinTreeNode * current);      //求指定结点的左子女
    BinTreeNode * RightChild(BinTreeNode * current);     //求指定结点的右子女
    bool Insert(T item);                                 //按指定规则在树中插入新元素 item
    bool Remove(T item);                                 //在树中删除元素 item
    bool Find(T item);                                   //判断 item 是否在树中
    bool getData(T& item);                               //取得结点数据,通过 item 返回
    BinTreeNode * getRoot();                             //取根
    void PreOrder(BinTreeNode * p);                      //前序遍历
    void InOrder(BinTreeNode * p);                       //中序遍历
    void PostOrder(BinTreeNode * p);                     //后序遍历
    void LevelOrder(BinTreeNode * p);                    //层次序遍历
};
```

5.3 二叉树的存储表示

二叉树的存储结构有两种：数组方式和链表方式。

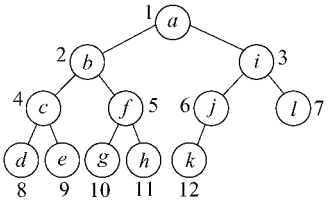
5.3.1 二叉树的数组存储表示

二叉树的数组存储表示又称二叉树的顺序存储表示。

在数据处理过程中二叉树的大小和形态不发生剧烈的动态变化的场合，适宜采用数组方式来表示二叉树的抽象数据类型。用数组方式存储二叉树的结构，就是将二叉树的数据元素存储在一组连续的存储单元之内。这种方式可以用 C ++ 的数组来描述，用数组元素的下标为索引，随机存取二叉树的结点。这种存储方案必须兼顾树形结构的特点，使各结点能够方便地定位到它的父结点、左子女和右子女。

1. 完全二叉树的数组存储表示

设有一棵完全二叉树，如图 5.7(a)所示，对它所有的结点按照层次次序自顶向下，同一层自左向右顺序编号 1，2，…，n，就得到一个结点的顺序(线性)序列。按这个线性序列，把这棵完全二叉树放在一个一维数组中，如图 5.7(b)所示。



(a) 树形表示

1	2	3	4	5	6	7	8	9	10	11	12
a	b	i	c	f	j	l	d	e	g	h	k

(b) 数组存储

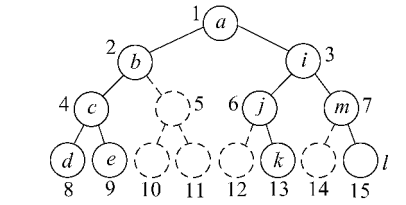
图 5.7 完全二叉树的数组表示

在数组 BinTree.data[i]中存放编号为 i 的完全二叉树的结点。采用这种方式，可以利用完全二叉树的性质 5.5，从一个结点的编号推算出它的父、子女、兄弟等结点的编号，从而找到这些结点。**这种存储表示是存储完全二叉树最简单、最省存储的存储方式。**

2. 一般二叉树的数组表示

设有一棵一般二叉树，如图 5.8(a)所示，需要将它存放在一个一维数组中。为了能够简单地找到某一个结点的上下左右的关系，也必须仿照完全二叉树那样，对二叉树的结点进行编号。然后按其编号将它放到数组中去，如图 5.8

(b)所示。在编号时,如遇到空子树,应在编号时假定有此子树进行编号,而在顺序存储时当作有此子树那样把位置留出来。这样才能反映二叉树结点之间的相互关系,由其存储位置找到它的父结点、子女结点、兄弟结点的位置,但这样做有可能会消耗大量的存储空间。例如,图 5.9 给出的单支二叉树,要求一个可存放 31 个结点的一维数组,但只在第 0,2,6,14,30 这几个位置存放有结点数据,其他大多数结点空间都空着,又不能压缩到一起,造成很大空间浪费。



(a) 树形表示

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	i	c		j	m	d	e				k		l

(b) 数组存储

图 5.8　一般二叉树的数组表示

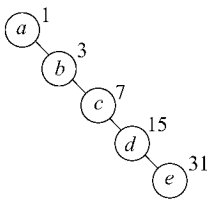
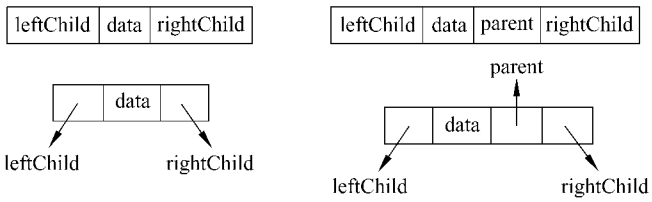


图 5.9　单支树

5.3.2　二叉树的链接存储表示

数组表示法用于完全二叉树的存储表示非常有效,但表示一般二叉树,尤其是形态剧烈变化的二叉树,存储空间的利用不是很理想。使用链接存储表示,可以克服这些缺点。

根据二叉树定义,二叉树的每个结点可以有两个分支,分别指向结点的左、右子树的根结点(如果存在)。因此,二叉树的结点至少应当包括 3 个域,分别存放结点的数据 data、左子女结点指针 leftChild 和右子女结点指针 rightChild,如图 5.10(a)所示。这种链表结构称为二叉链表。**使用这种结构,可以很方便地根据结点** leftChild **指针和** rightChild **指针找到它的左子女和右子女,但要找到它的父结点很困难**。为了便于查找任一结点的父结点,可以在结点中再增加一个父结点指针域 parent,它被称为三叉链表,如图 5.10(b)所示。

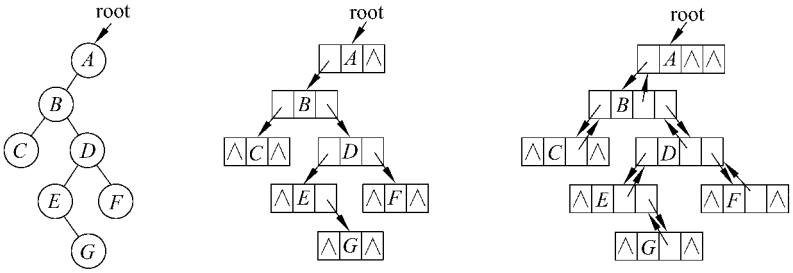


(a) 二叉链表　　(b) 三叉链表

图 5.10　二叉树的链接存储表示

整个二叉树的链表有一个表头指针,它指向二叉树的根结点,其作用是当作树的访问点。图 5.11(b)和图 5.11(c)分别是图 5.11(a)所示二叉树的二叉链表和三叉链表。根据性质 5.3 很容易验证,在含有 n 个结点的二叉链表中有 $n+1$ 个空链指针域,这是因为在所有结点的 $2n$ 个链指针域中只有 $n-1$ 个存有边信息的缘故。三叉链表则有 $n+2$ 个空链指

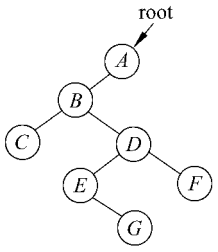
针域。



(a) 二叉树　　(b) 二叉链表　　(c) 三叉链表

图 5.11　二叉树的链接存储表示的示例

二叉链表和三叉链表可以是静态链表结构，即把链表存放在一个一维数组中，数组中每一数组元素是一个结点，它包括了 3 个域：数据域 data、左子女指针域 leftChild 和右子女指针域 rightChild。为寻找父结点方便，还可增加父指针域 parent。指针域指示另一结点在数组中的下标。也可以把链表分为两个数组存放，一个数组专存数据，另一个数组专存指针（包括 leftChild 指针、rightChild 指针和 parent 指针），如图 5.12 所示。



	data	parent	leftChild	rightChild
0	*A*	−1	1	−1
1	*B*	0	2	3
2	*C*	1	−1	−1
3	*D*	1	4	5
4	*E*	3	−1	6
5	*F*	3	−1	−1
6	*G*	4	−1	−1

图 5.12　静态链表结构

程序 5.3　利用二叉链表的二叉树类定义

```
#include <iostream.h>
#include <stdlib.h>
#include <assert.h>
template <class T>
struct BinTreeNode {                                    //二叉树结点类定义
    T data;                                             //数据域
    BinTreeNode<T> *leftChild, *rightChild;             //左子女、右子女链域
    BinTreeNode(): leftChild(NULL), rightChild(NULL) {}
    BinTreeNode(T x, BinTreeNode<T> *l = NULL, BinTreeNode<T> *r = NULL)
        : data(x), leftChild(l), rightChild(r) {}
};

template <class T>
class BinaryTree {                                      //二叉树类定义
public:
```

```
    BinaryTree(): root(NULL) {}                                    //构造函数
    BinaryTree(T value): RefValue(value), root(NULL) {}            //构造函数
    BinaryTree(BinaryTree<T>& s);                                  //复制构造函数
    ~BinaryTree() {destroy(root);}                                 //析构函数
    bool IsEmpty() {return root == NULL;}                          //判断二叉树是否为空
    BinTreeNode<T> * Parent(BinTreeNode <T> * current)             //返回父结点
        if (root == NULL || root == current) return NULL;
            else return Parent(root, current);
    int Height() {return Height(root);}                            //返回树的高度
    int Size() {return Size(root);}                                //返回树的结点数
    BinTreeNode<T> * getRoot() {return root;}                      //取根
    void setRoot(BinTreeNode<T> * p) {root = p;}                   //修改根
    BinTreeNode<T> * Search(T item)                                //搜索
        {return Search(root, item);}
    int Insert(T item) {return Insert(root, item);}                //插入新元素
    void createBinTree(T in[]) {createBinTree(in, root); }
                                                                   //从输入流读入建树
    void printBinTree() {printBinTree(root);}                      //用广义表输出二叉树
    void Traverse() {Traverse(root,1);}                            //用凹入表输出二叉树
    void createBinTree_pre(T pre[])                                //用扩展前序序列建树
        {int n = 0; createBinTree_pre(pre, root, n);}
    void PreOrder() {PreOrder(root);}                              //前序遍历
    void InOrder() {InOrder(root);}                                //中序遍历
    void PostOrder() {PostOrder(root);}                            //后序遍历
    void LevelOrder();                                             //层次序遍历
    void PreOrder_iter();                                          //非递归前序遍历
    void InOrder_iter();                                           //非递归中序遍历
    void PostOrder_iter();                                         //非递归后序遍历
    friend bool operator == (BinaryTree<T>& s, BinaryTree<T>& t);
                                                                   //重载操作:判等
protected:
    BinTreeNode<T> * root;                                         //二叉树的根指针
    T RefValue;                                                    //数据输入停止标志
    void destroy(BinTreeNode<T> * & subTree);                      //销毁子树
    void createBinTree(T in[], BinTreeNode<T> * & subTree);
                                                                   //从广义表串构建二叉树
    void createBinTree_pre(T pre[], BinTreeNode<T> * & subTree, int& n);
                                                                   //从扩展前序序列构建二叉树
    BinTreeNode<T> * Parent(BinTreeNode<T> * subTree, BinTreeNode<T> * p);
                                                                   //找父结点
    BinTreeNode<T> * Search(BinTreeNode<T> * p, T item);           //搜索
    int Insert(BinTreeNode<T> * & subTree, T item);                //插入
    BinTreeNode<T> * Copy(BinTreeNode<T> * orignode);              //复制
    int Height(BinTreeNode<T> * subTree);                          //求子树的高度
    int Size(BinTreeNode<T> * subTree);                            //求子树的结点数
```