第5章 树和二叉树

树形结构是一种常用的非线性结构，数据元素之间具有一对多的层次关系，常用于描述各种具有层次关系的数据对象，例如行政组织结构、文件目录等。本章主要讨论二叉树的存储结构、基本操作的实现及其应用。

5.1 树的定义和基本操作

5.1.1 树的定义和基本术语

1. 树的定义

树(tree)是 n(n≥0) 个结点的有限集。在任意一棵非空树中：

(1) 有且只有一个称为根(root)的结点。

(2) 除根之外的其余 n−1 个结点被分为 m(m≥0) 个互不相交的有限集，其中每个集合本身又是一棵树，称为根结点的子树(sub_tree)。

可以看出，树的定义是递归的，即在树的定义中又用到了树的概念。递归定义和递归操作在树和二叉树中的应用是比较广泛的，应注意领会递归的实质。

2. 树的表示法

树的表示法有树形表示法、文氏图表示法、广义表表示法和凹入表示法，如图 5.1 所示。

其中，树形表示法是最常用的表示方法。在用树形表示法表示的树中，边的数目(或称分支数，用 e 表示)恰好比结点的数目(用 n 表示)少一个，即 e=n−1。这是树形结构最重要的一个结论。

3. 树的基本术语

(1) 结点：在树中，通常将数据元素称为结点。从计算机的角度来划分，可以分为终端结点和非终端结点；以树的特征划分，可以分为根结点、分支结点和叶子结点；用族谱的关系划分，可以分为双亲结点和孩子结点、祖先结点和子孙结点、兄弟结点和堂兄弟结点。

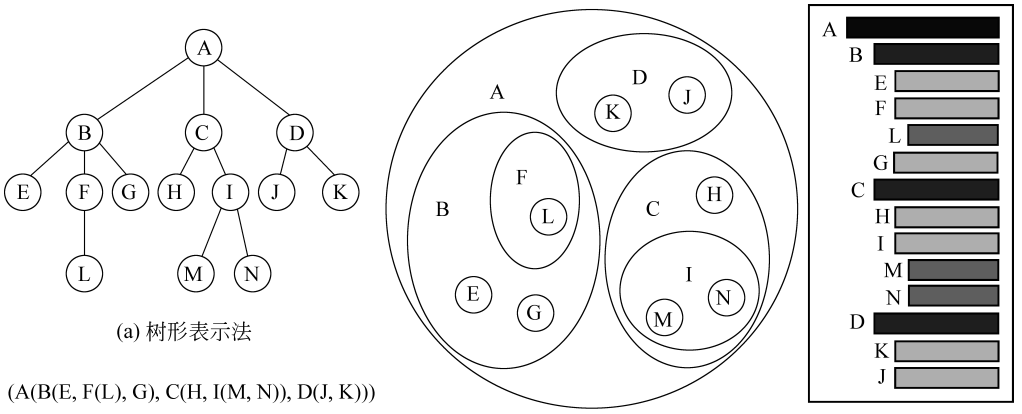


图 5.1 树的几种表示法

(2) **度**：分为结点的度和树的度两种。**结点的度**是指与该结点相连接的孩子结点的数目。**树的度**是指树中所有结点的度的最大值。

(3) **深度**：树是一种分层结构，根结点作为第一层，其余**结点的深度(或称层次)为其双亲结点的层数加 1**。**树的深度**(或称**层数**)是指树中所有结点的层次的最大值。

(4) **有序树与无序树**：如果将树中结点的各棵子树看成是从左到右有次序的(即不能互换)，则称该树为有序树，否则称该树为无序树。

(5) **有向树与无向树**：如果树的每个分支都是有方向的，则称该树为有向树，否则称该树为无向树。

(6) **n 元树**：树的度为 n 的有向树。

(7) **位置树**：指树中每个结点的孩子结点的位置不能被改变(改变则不是原树)的有向树。例如，某结点可能没有第一个孩子结点，但却可能有第二个和第三个孩子结点。

(8) **m 叉树**：指树的度为 m 的有向位置树，即 m 元位置树。

(9) **森林**：指 m(m≥0) 棵互不相交的树的集合。对于树中的每个结点，其子树的集合就是森林，因此森林和树是密切相关的。森林中的树也可以有顺序关系和位置关系。

5.1.2 树的基本操作

树的基本操作通常有以下几种。

(1) 初始化操作 InitTree(T)，用于建立一棵空树 T。

(2) 清空操作 CleTree(T)，用于释放树 T 占用的存储空间。

(3) 判空操作 EmpTree(T)，用于判断树 T 是否为空树。

(4) 求深度操作 DepTree(T)，用于计算树 T 的深度。

(5) 插入操作 InsChild(T,p,i,c)，用于在树 T 中插入子树 c，使其成为 p 指向的结点的第 i 棵子树。

(6) 删除操作 DelChild(T,p,i)，用于删除树 T 中 p 所指结点的第 i 棵子树。

(7) 遍历操作 TraTree(T)，用于按某种次序对树 T 中的所有结点进行访问，且每个

结点仅访问一次。

二叉树是一种最简单的树形结构，它有许多好的性质，且任何树都可以转化为二叉树。因此，下面重点研究二叉树的性质、存储结构及其应用。

5.2　二　叉　树

5.2.1　二叉树的定义和基本操作

1. 二叉树的定义

二叉树(binary tree)是一种特殊的有向树，也称二元位置树，它的特点是每个结点至多有两棵子树，即二叉树中的每个结点至多有两个孩子结点，且每个孩子结点都有各自的位置关系。或者说，二叉树的子树有左右之分，其次序不能任意颠倒。具体地，**二叉树**或者为空，或者由一个根结点及两棵不相交的、分别称为左子树和右子树的二叉树组成。

根据二叉树的定义，二叉树有 5 种形态，如图 5.2 所示。

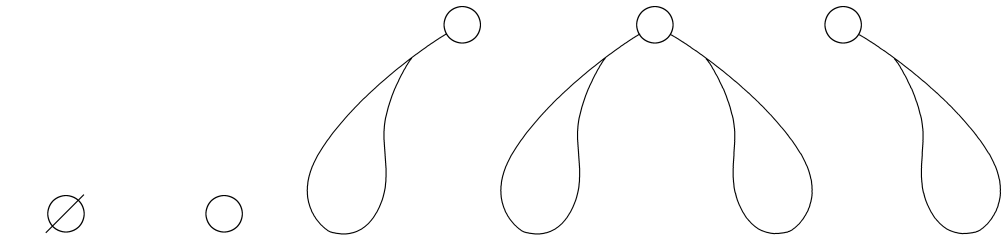


图 5.2　二叉树的 5 种基本形态

【例 5.1】 列举出只有两个结点、三个结点的二叉树的所有形态。

(1) 只有两个结点的二叉树的形态有两种，如图 5.3 所示。

图 5.3　只有两个结点的二叉树的所有形态

(2) 只有三个结点的二叉树的形态有五种，如图 5.4 所示。

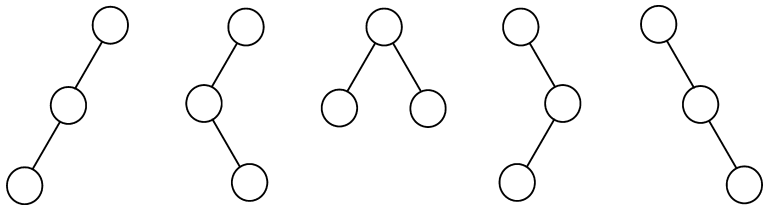


图 5.4　只有三个结点的二叉树的所有形态

满二叉树和完全二叉树是两种特殊形态的二叉树,在实际应用中经常用到。

满二叉树是除叶子结点外的任何结点均有两个孩子结点,且所有叶子结点都在同一层的二叉树。这种二叉树的特点是每一层上的结点数目都是最大值。图 5.5(a)是一棵深度为 4 的满二叉树。

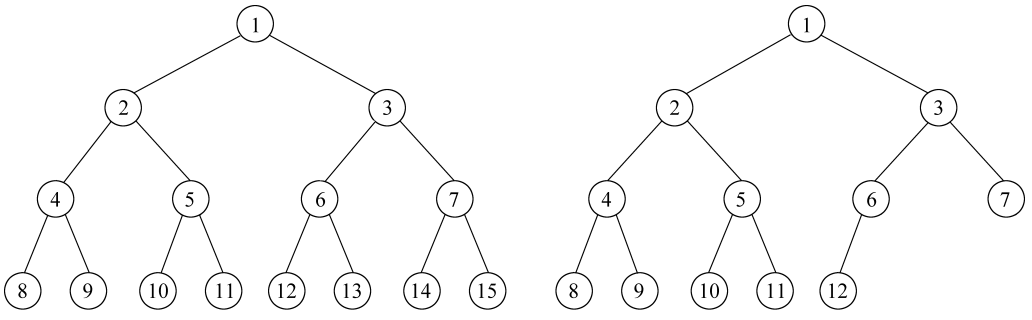


图 5.5 满二叉树与完全二叉树

完全二叉树是除去最底层结点后的二叉树是一棵满二叉树,且最底层结点均靠左对齐的二叉树。在这里,靠左对齐的含义是左边是满的,即没有空隙再放入任何一个结点。图 5.5(b)是一棵深度为 4 的完全二叉树。实质上,满二叉树是完全二叉树的一个特例。

2. 二叉树的基本操作

二叉树的基本操作通常有以下几种。

(1) 创建操作 CreBiTree(bt),用于建立一棵二叉树 bt。

(2) 先序遍历操作 PreOrder(bt),用于先序遍历二叉树 bt。

(3) 中序遍历操作 InOrder(bt),用于中序遍历二叉树 bt。

(4) 后序遍历操作 PostOrder(bt),用于后序遍历二叉树 bt。

(5) 查找操作 Search(bt,x,p),用于在二叉树 bt 中查找值为 x 的结点,并通过 p 带回该结点的指针。

(6) 插入操作 InsTree(bt,p,x),用于在二叉树 bt 中 p 指向的结点位置插入一个值为 x 的结点。

(7) 删除操作 DelTree(bt,p),用于在二叉树 bt 中删除 p 指向的结点。

5.2.2 二叉树的性质

二叉树有许多性质,也就是说,二叉树的理论基础较强,应用也较为广泛,下面依次进行讨论。

性质 1:在二叉树的第 i(i≥1) 层上至多有 2^{i-1} 个结点。

该性质的证明利用数学归纳法很容易实现,留给读者自行思考。

性质 2:深度为 k(k≥1) 的二叉树上至多有 2^k-1 个结点。

该性质的证明直接利用性质 1 即可,留给读者自行思考。

性质 3:在任意一棵二叉树中,叶子结点的数目(用 n_0 表示)总是比度为 2 的结点的

数目(用 n_2 表示)多一个,即 $n_0=n_2+1$。

证明:设二叉树的结点总数为 n,度为 1 的结点数为 n_1,则有

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

在二叉树中,除根结点之外的每个结点都有唯一的一个分支进入。设分支总数为 e,则有

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

由于这些分支或者是度为 1 的结点射出的,或者是度为 2 的结点射出的,所以有

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

由式(5-2)和式(5-3)可以得到

$$n=n_1+2n_2+1 \tag{5-4}$$

由式(5-1)和式(5-4)可以得到

$$n_0=n_2+1$$

证毕。

性质 4:具有 n 个结点的完全二叉树的深度为$\lfloor \log_2 n \rfloor+1$。

证明:设具有 n 个结点的完全二叉树的深度为 k,则由性质 2 可知

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

即

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

取以 2 为底的对数,得

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

因为 $\log_2 n$ 处于两个连续的整数 k−1 和 k 之间,所以 $k-1=\lfloor \log_2 n \rfloor$,即

$$k=\lfloor \log_2 n \rfloor+1$$

证毕。

性质 5:具有 n 个结点的完全二叉树,若按照从上至下、从左至右的顺序对二叉树中的所有结点从 1 开始顺序编号,则对于序号为 i(1≤i≤n)的结点,有:

(1) 其双亲结点的编号为$\lfloor i/2 \rfloor$(1<i≤n)。

(2) 其左孩子结点的编号为 2i(1≤i≤n/2)。

(3) 其右孩子结点的编号为 2i+1(1≤i≤(n−1)/2)。

先利用数学归纳法证明(2)和(3),再从(2)和(3)推出(1),具体证明过程留给读者自行思考。

性质 6:在有 n 个结点的完全二叉树中,度为 1 的结点数为(n+1) %2。

证明:设完全二叉树中的分支数为 e,则根据树的定义可知 e=n−1。

若 n 为偶数,则分支数 e 为奇数,根据完全二叉树的定义可知,在完全二叉树中仅有一个度为 1 的结点。

同理,若 n 为奇数,则分支数 e 为偶数,在完全二叉树中没有度为 1 的结点。

证毕。

性质 7:具有 n 个结点的完全二叉树,若按照从上至下、从左至右的顺序对二叉树中的所有结点从 1 开始顺序编号,则完全二叉树中编号大于$\lfloor n/2 \rfloor$的结点均为叶子结点。

证明：设完全二叉树中度为 0 的结点数为 n_0，度为 1 的结点数为 n_1，度为 2 的结点数为 n_2。

(1) 当 n 为偶数时，由性质 6 可知

$$n_1 = 1 \tag{5-5}$$

由性质 3 可知

$$n_0 = n_2 + 1 \tag{5-6}$$

由式(5-5)和式(5-6)可以得到

$$n_0 = n_2 + n_1 \tag{5-7}$$

又因结点总数为

$$n = n_0 + n_1 + n_2 \tag{5-8}$$

由式(5-7)和式(5-8)可以得到

$$n_0 = n_1 + n_2 = \lfloor n/2 \rfloor$$

即，当 n 为偶数时，编号大于$\lfloor n/2 \rfloor$的结点均为叶子结点。

(2) 当 n 为奇数时，由性质 6 可知

$$n_1 = 0 \tag{5-9}$$

又因结点总数为

$$n = n_0 + n_1 + n_2 \tag{5-10}$$

由式(5-9)和式(5-10)可以得到

$$n = n_0 + n_2 \tag{5-11}$$

由性质 3 可知

$$n_0 = n_2 + 1 \tag{5-12}$$

由式(5-11)和式(5-12)可以得到

$$n_1 + n_2 = (n-1)/2 = \lfloor n/2 \rfloor$$

即，当 n 为奇数时，编号大于 n/2 的结点均为叶子结点。

证毕。

在此需要注意的是，一般二叉树的性质可用于完全二叉树，但完全二叉树的性质不能用于一般二叉树。

【例 5.2】 已知一棵完全二叉树中有 234 个结点，试问：

(1) 树的高度是多少？

(2) 第 7 层和第 8 层上各有多少个结点？

(3) 树中有多少个叶子结点？有多少个度为 2 的结点？有多少个度为 1 的结点？

本例的分析如下。

(1) 由性质 4 可知，该完全二叉树的高度(k)为

$$k = \lfloor \log_2 234 \rfloor + 1 = \log_2 2^7 + 1 = 8$$

(2) 由性质 1 可知，第 7 层上的结点数为 $2^{7-1} = 2^6 = 64$(个)。

由性质 2 可知，第 8 层上的结点数为 $234 - (2^7 - 1) = 107$(个)。

(3) 由性质 7 可知，树中叶子结点的个数为 $234 - \lfloor 234/2 \rfloor = 117$(个)。

由性质 2 可知，度为 2 的结点数为 $117 - 1 = 116$(个)。

由性质 6 可知，度为 1 的结点数为 1(个)。

5.2.3　二叉树的遍历

二叉树的遍历是二叉树最基本的操作，通过遍历可以完成二叉树的很多操作。**二叉树遍历**是指按照某种次序访问二叉树中的每个结点，且每个结点仅被访问一次。在这里，访问的含义比较广泛。访问并非一定是输出结点数据，还可能是查看结点属性值、更新结点数据值、增加或删除结点等，这些都可以看成是访问。不失一般性，在此将以输出结点值为例研究二叉树的遍历过程。

第 2 章中介绍的线性表的遍历比较简单，只需要从头至尾扫描一遍即可。由于二叉树描述的数据元素之间的关系是一对多的关系，因此很难从头至尾一遍扫描完。从二叉树的定义可以看出，一棵二叉树由根结点、左子树和右子树三部分组成。因此，只要依次遍历这 3 部分，即可完成整个二叉树的遍历。假如以 L、D、R 分别表示遍历左子树、访问根结点和遍历右子树，则有 LDR、LRD、DLR、DRL、RLD、RDL 六种遍历方式。在遍历时，常常规定“先左子树(L)，后右子树(R)”的顺序，因此遍历二叉树就有 DLR、LDR 和 LRD 这 3 种方式。根据根结点在遍历时的访问顺序，分别称以上 3 种方式为二叉树的**先(根)序遍历**、**中(根)序遍历**和**后(根)序遍历**。

由图 5.6 可以看出，从根结点出发，逆时针沿着二叉树外缘移动，每个结点都分别从左侧、正下方和右侧途经 3 次。访问的时机不同，就是不同的遍历。若结点访问均是第 1 次途经进行的，则是先序遍历；若结点访问均是第 2 次途经进行的，则是中序遍历；若结点访问均是第 3 次途经进行的，则是后序遍历。图 5.6 中的“▭”表示空指针。

【例 5.3】　已知一棵二叉树如图 5.7 所示，写出其先序、中序和后序遍历序列。

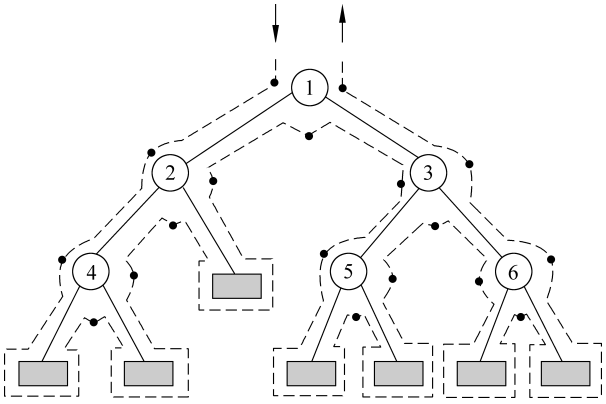


图 5.6　二叉树的 3 种遍历访问结点的时机

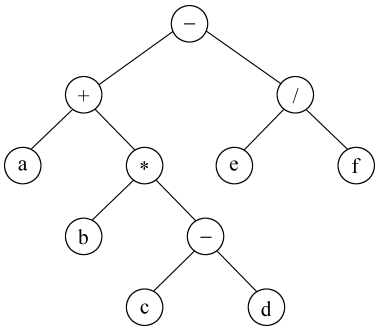


图 5.7　例 5.3 的二叉树

先序遍历序列为 − + a * b − cd/ef。

中序遍历序列为 a + b * c − d − e/f。

后序遍历序列为 abcd − * + ef/ − 。

【例 5.4】　已知一棵二叉树的先序遍历序列和中序遍历序列分别为 KAFGIBEDHJC 和 AGFKEDBHJIC，试画出这棵二叉树。

分析：先序遍历序列中的第一个元素一定是(子)树的根结点，在中序遍历序列中，排在这个根结点前面的结点一定是其左子树上的结点，排在这个根结点后面的结点一定是其右子树上的结点。如此依次作用于各子序列，就可以画出整个二叉树的树形，而且是唯一的，如图 5.8 所示。

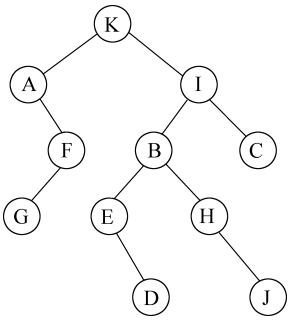


图 5.8 例 5.4 的结果

【例 5.5】 已知一棵二叉树的后序遍历序列和中序遍历序列分别为 FBCGIEJDAH 和 BFGCHIEADJ，试画出这棵二叉树。

分析：后序遍历序列中的最后一个元素一定是(子)树的根结点，在中序遍历序列中，排在这个根结点前面的结点一定是其左子树上的结点，排在这个根结点后面的结点一定是其右子树上的结点。如此依次作用于各子序列，就可以画出整个二叉树的树形，而且是唯一的，如图 5.9 所示。

【例 5.6】 已知一棵二叉树的先序遍历序列和后序遍历序列分别为 AB 和 BA，试画出这棵二叉树。

分析：由先序遍历序列可知，A 一定是根结点，综合两种遍历序列可知，B 既可能是 A 的左孩子结点，也可能是 A 的右孩子结点，所以这棵树的形状是不确定的，如图 5.10 所示。

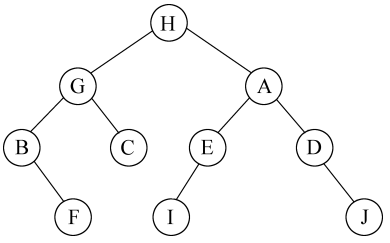


图 5.9 例 5.5 的结果

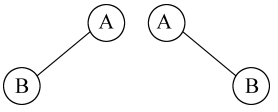
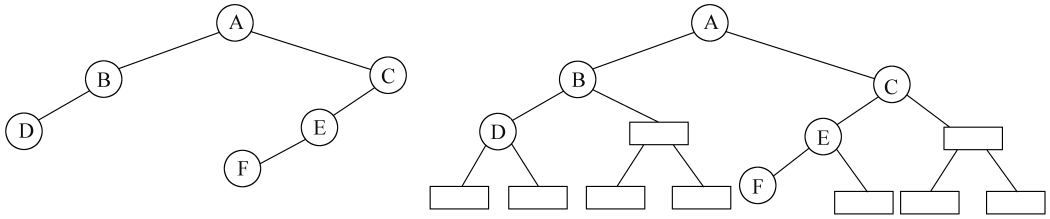


图 5.10 例 5.6 的结果

与线性表类似，二叉树也有顺序存储和链式存储两种存储结构，所不同的是，这两种存储结构不是表示数据的线性关系，而是表示一种层次关系。

5.2.4 二叉树的顺序存储结构

将二叉树上的结点值按从上至下、从左至右的顺序存储到一个线性结构(通常为数组)中，这种存储方式称为二叉树的顺序存储结构。但是，为了方便计算，并不能简单地将各个结点顺序存放到数组的各个单元中，而必须增加一些**虚结点**，使之变成满二叉树的树形。这样处理的目的是能够明确表示出树中各个结点之间的相互关系，给操作带来便利。例如，对于图 5.11(a)所示的二叉树，增加虚结点后的满二叉树如图 5.11(b)所示，二叉树的顺序存储示意图如图 5.11(c)所示。其中，数组中下标为 0 的单元可用于存放满二叉树中结点的总数或二叉树的深度(图 5.11(c)中存放的是深度为 4 的满二叉树的结点数)，而虚结点可以用一个特殊的标志识别。



(a) 二叉树　　(b) 增加虚结点后的满二叉树

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	A	B	C	D	0	E	0	0	0	0	0	F	0	0	0

(c) 二叉树顺序存储示意图

图 5.11　二叉树及其顺序存储示意图

二叉树的顺序存储结构类型说明为

```
#define MAXSIZE 100                          /* 存储空间最大容量 */
typedef char ElemType;                       /* 结点值类型 */
#define VirNode '0'                          /* 虚结点值 */
typedef ElemType SeqTree[MAXSIZE];
/* SeqTree[0]单元存放结点数,通常存放对应的满二叉树的结点数 */
```

下面介绍顺序存储二叉树基本操作的实现。

1. 建立二叉树

算法思路：将结点值(包括虚结点)按层次依次存入相应单元,下标为 0 的单元存放对应的满二叉树的结点数。

```
void CreBiTree(SeqTree bt,int n)             /* n 为真实结点总数 */
{ int i=1,j,m=0;
  while(m<n)
  { for(j=i;j<2*i;j++)                       /* 按层次输入,虚结点值一起输入 */
    { scanf("%c",bt+j);
      if(bt[j]!=VirNode) m++;
    }
    i=2*i;
  }
  bt[0]=i-1;                                 /* 0 号单元存放满二叉树的结点总数 */
}
```

2. 层次遍历

算法思路：从下标(编号)为 1 的单元开始,若是虚结点,则输出一个“*”号,否则输出结点值,输出一层后换行。重复此操作,直到下标(编号)大于 bt[0]为止。

```
void LevelTree(SeqTree bt)                   /* 按满二叉树遍历(输出) */
{ int i=1,j;
  while(i<=bt[0])                            /* 按层扫描 */
```

```
  { for(j=i;j<2*i;j++)                    /*扫描第 i 层结点*/
      if(bt[j]==VirNode) printf("*");     /*若是虚结点,则输出一个"*"号*/
      else printf("%c",bt[j]);
    printf("\n");
    i=2*i;                                /*跳到下一层的第一个结点*/
  }
}
```

3. 先序递归遍历

算法思路：若根结点编号不大于满二叉树的结点数，且为实结点，则进行下列操作：

(1) 访问根结点；

(2) 先序递归遍历根的左子树；

(3) 先序递归遍历根的右子树。

```
void PreOrder(SeqTree bt,int i)           /*i 是根结点存储位置(下标或编号)*/
{ if(i<=bt[0])
  { if(bt[i]!=VirNode)
    { printf("%4c",bt[i]);               /*访问根结点*/
      PreOrder(bt,2*i);                 /*先序遍历根结点的左子树*/
      PreOrder(bt,2*i+1);               /*先序遍历根结点的右子树*/
    }
  }
}
```

4. 中序递归遍历

算法思路：若根结点编号不大于满二叉树的结点数，且为实结点，则进行下列操作：

(1) 中序递归遍历根的左子树；

(2) 访问根结点；

(3) 中序递归遍历根的右子树。

```
void InOrder(SeqTree bt,int i)            /*i 是根结点存储位置(下标或编号)*/
{ if(i<=bt[0])
  { if(bt[i]!=VirNode)
    { InOrder(bt,2*i);                  /*中序遍历根结点的左子树*/
      printf("%4c",bt[i]);              /*访问根结点*/
      InOrder(bt,2*i+1);                /*中序遍历根结点的右子树*/
    }
  }
}
```

5. 后序递归遍历

算法思路：若根结点编号不大于满二叉树的结点数，且为实结点，则进行下列操作：

(1) 后序递归遍历根的左子树；

(2) 后序递归遍历根的右子树；