第5章 树 结 构

5.1 树

5.1.1 树的基本概念

树是数据元素之间具有层次关系的非线性结构，是由 n 个结点构成的有限集合，结点数为 0 的树叫空树。树必须满足以下条件。

(1) 有且仅有一个被称为根的结点，该结点没有前驱结点。

(2) 其余结点可分为 m 个互不相交的有限集合，每个集合又构成一棵树，叫根结点的子树。

与线性结构不同，树中的数据元素具有一对多的逻辑关系，除根结点外，每个数据元素可以有多个后继但有且仅有一个前驱，反映了数据元素之间的层次关系。树是递归定义的。结点是树的基本单位，若干结点组成一棵子树，若干互不相交的子树组成一棵树。

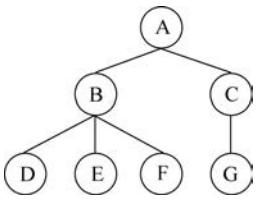
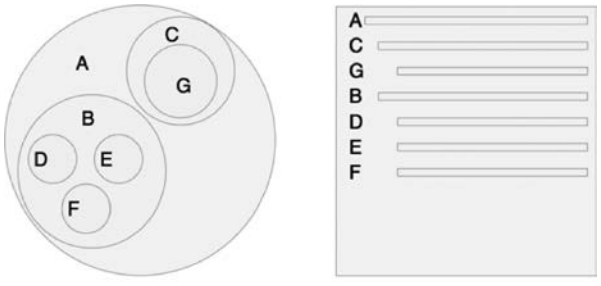


图 5.1 树的逻辑结构示意图

人们在生活中所见的家谱、Windows 的文件系统等，虽然表现形式各异，但在本质上是树结构。图 5.1 给出了树的逻辑结构示意图。

树的表示方法有多种，如树形表示法、文氏图表示法、凹入图表示法和广义表表示法等。图 5.1 所示为树形表示法，图 5.2 给出了用其他 3 种表示法对树的表示。



文氏图表示法　　凹入图表示法

A(B(D,E,F),C(G))
广义表表示法

图 5.2 树的 3 种表示方法

5.1.2 树的术语

1. 结点

树的结点就是构成树的数据元素，就是其他数据结构中存储的数据项，在树形表示法中用圆圈表示。

2. 结点的路径

结点的路径是指从根结点到该结点所经过结点的顺序排列。

3. 路径的长度

路径的长度指的是路径中包含的分支数。

4. 结点的度

结点的度指的是结点拥有的子树的数目。

5. 树的度

树的度指的是树中所有结点的度的最大值。

6. 叶结点

叶结点是树中度为 0 的结点，也叫终端结点。

7. 分支结点

分支结点是树中度不为 0 的结点，也叫非终端结点。

8. 子结点

子结点是指结点的子树的根结点，也叫孩子结点。

9. 父结点

具有子结点的结点叫该子结点的父结点，也叫双亲结点。

10. 子孙结点

子孙结点是指结点的子树中的任意结点。

11. 祖先结点

祖先结点是指结点的路径中除自身之外的所有结点。

12. 兄弟结点

兄弟结点是指和结点具有同一父结点的结点。

13. 结点的层次

树中根结点的层次为 0，其他结点的层次是父结点的层次加 1。

14. 树的深度

树的深度是指树中所有结点的层次数的最大值加 1。

15. 有序树

有序树是指树的各结点的所有子树具有次序关系，不可以改变位置。

16. 无序树

无序树是指树的各结点的所有子树之间无次序关系，可以改变位置。

17. 森林

森林是由多个互不相交的树构成的集合。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森林。

5.2 二 叉 树

5.2.1 二叉树的基本概念

1. 普通二叉树

二叉树是特殊的有序树,它也是由 n 个结点构成的有限集合。当 $n=0$ 时称为空二叉树。二叉树的每个结点最多只有两棵子树,子树也为二叉树,互不相交且有左右之分,分别称为左二叉树和右二叉树。

二叉树也是递归定义的,在树中定义的度、层次等术语同样也适用于二叉树。

2. 满二叉树

满二叉树是特殊的二叉树,它要求除叶结点外的其他结点都具有两棵子树,并且所有的叶结点都在同一层上,如图 5.3 所示。

3. 完全二叉树

完全二叉树是特殊的二叉树,若完全二叉树具有 n 个结点,且按照从上到下、从左到右的顺序将二叉树结点编号,则它要求 n 个结点与满二叉树的前 n 个结点具有完全相同的逻辑结构,如图 5.4 所示。即在完全二叉树中,只有最下层和次下层可以出现叶结点,且最下层的叶结点集中在左侧。

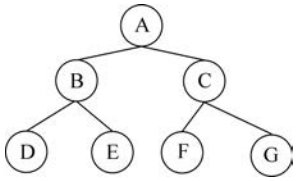


图 5.3 满二叉树

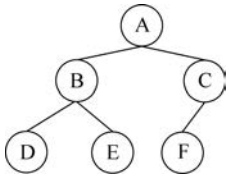


图 5.4 完全二叉树

5.2.2 二叉树的性质

性质 1:二叉树中第 i 层的结点数最多为 2^i。

证明:当 $i=0$ 时只有一个根结点,成立;假设对所有的 $k(0\leqslant k<i)$ 成立,即第 $i-1$ 层上最多有 2^{i-1} 个结点,那么由于每个结点最多有两棵子树,在第 i 层上结点数最多为 $2^{i-1}\times 2=2^i$ 个,得证。

性质 2:深度为 h 的二叉树最多有 2^h-1 个结点。

证明:由性质 1 得,深度为 h 的二叉树的结点个数最多为 $2^0+2^1+\cdots+2^{h-1}=2^h-1$,得证。

性质 3:若二叉树的叶结点的个数为 n,度为 2 的结点个数为 m,有 $n=m+1$。

证明:设二叉树中度为 1 的结点个数为 k,二叉树的结点总数为 s,有 $s=k+n+m$。又因为除根结点外每个结点都有一个进入它的分支,所以 $s-1=k+2*m$。整理后得到 $n=m+1$,得证。

性质 4：具有 n 个结点的完全二叉树，其深度为 $\lfloor \log_2 n \rfloor + 1$ 或者 $\lceil \log_2 (n+1) \rceil$。

证明：设此二叉树的深度为 h，由性质 2 可得 $2^{h-1} \leqslant n < 2^h$，两边取对数，可得 $h-1 \leqslant \log_2 n < h$，因为 h 为整数，所以 $h = \lfloor \log_2 n \rfloor + 1$，得证。

性质 5：具有 n 个结点的完全二叉树，从根结点开始自上而下、从左向右对结点从 0 开始编号。对于任意一个编号为 i 的结点：

(1) 若 $i=0$，结点为根结点，则没有父结点；若 $i>0$，则父结点的编号为 $\lfloor (i-1)/2 \rfloor$。

(2) 若 $2i+1 \geqslant n$，该结点无左孩子，否则左孩子结点的编号为 $2i+1$。

(3) 若 $2i+2 \geqslant n$，该结点无右孩子，否则右孩子结点的编号为 $2i+2$。

【例 5.1】 证明：对于任意一个满二叉树，其分支数 $B=2(n_0-1)$，其中 n_0 为终端结点数。

解：设 n_2 为度为 2 的结点，因为在满二叉树中没有度为 1 的结点，所以有

$$n = n_0 + n_2$$

设 B 为树中分支数，则

$$n = B + 1$$

所以

$$B = n_0 + n_2 - 1$$

再由二叉树的性质

$$n_0 = n_2 + 1$$

代入上式有

$$B = n_0 + n_0 - 1 - 1 = 2(n_0 - 1)$$

【例 5.2】 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，问该树中共有多少个叶结点？

解：设该树的总结点数为 n，则

$$n = n_0 + n_1 + n_2 + \cdots + n_m$$

又

$$n = \text{分支数} + 1 = 0 \times n_0 + 1 \times n_1 + 2 \times n_2 + \cdots + m \times n_m + 1$$

由上述两式可得

$$n_0 = n_2 + 2n_3 + \cdots + (m-1)n_m + 1$$

5.2.3 二叉树的存储结构

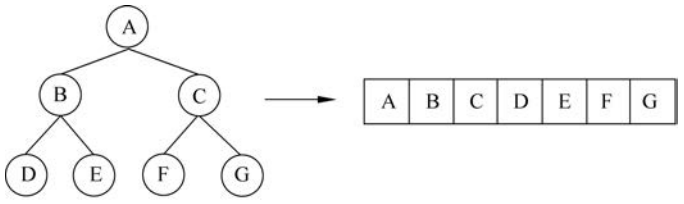
1. 二叉树的顺序存储结构

二叉树的顺序存储结构是指将二叉树的各个结点存放在一组地址连续的存储单元中，所有结点按结点序号进行顺序存储。因为二叉树为非线性结构，所以必须先将二叉树的结点排成线性序列再进行存储，实际上是对二叉树先进行一次层次遍历。二叉树的各结点间的逻辑关系由结点在线性序列中的相对位置确定。

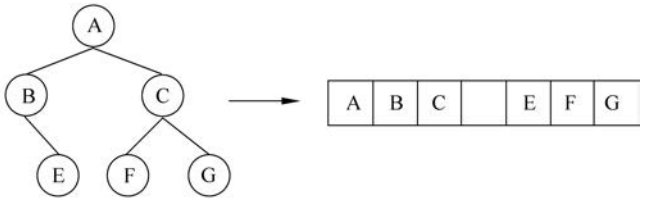
可以利用 5.2.2 节中的性质 5 将二叉树的结点排成线性序列，将结点存放在下标为对

应编号的数组元素中。为了存储非完全二叉树,需要在树中添加虚结点使其成为完全二叉树后再进行存储,这样会造成存储空间的浪费。

图 5.5 所示为二叉树的顺序存储结构示意图。



完全二叉树的顺序存储



非完全二叉树的顺序存储

图 5.5　二叉树的顺序存储结构

2. 二叉树的链式存储结构

二叉树的链式存储结构是指将二叉树的各个结点随机存放在存储空间中,二叉树的各结点间的逻辑关系由指针确定。每个结点至少要有两条链分别连接左、右孩子结点才能表达二叉树的层次关系。

根据指针域个数的不同,二叉树的链式存储结构又分为以下两种:

1) 二叉链式存储结构

二叉树的每个结点设置两个指针域和一个数据域。数据域中存放结点的值,指针域中存放左、右孩子结点的存储地址。

采用二叉链表存储二叉树,每个结点只存储了到其孩子结点的单向关系,没有存储到其父结点的关系,因此要获得父结点将花费较多的时间,需要从根结点开始在二叉树中进行查找,所花费的时间是遍历部分二叉树的时间,且与查找结点所处的位置有关。

2) 三叉链式存储结构

二叉树的每个结点设置 3 个指针域和一个数据域。数据域中存放结点的值,指针域中存放左、右孩子结点和父结点的存储地址。

图 5.6 所示为二叉链式存储和三叉链式存储的结点结构。

两种链式存储结构各有优缺点,二叉链式存储结构空间利用率高,而三叉链式存储结构既便于查找孩子结点,又便于查找父结点。在实际应用中,二叉链式存储结构更加常用,因此本书中二叉树的相关算法都是基于二叉链式存储结构设计的。

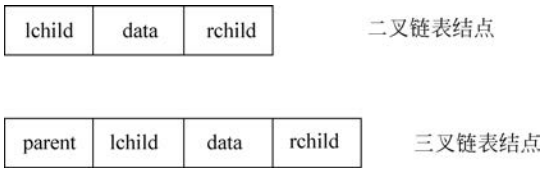


图 5.6　二叉和三叉链式存储的结点结构

3. 二叉链式存储结构的结点类的描述

```
1  package ch05;
2  public class BiTreeNode {
3  public Object data;                                    //存放结点的数据值
4  public BiTreeNode lchild,rchild;                       //存放结点的左、右孩子地址
5
6  public BiTreeNode(){
7      this(null,null,null);
8  }
9  public BiTreeNode(Object data){
10     this(data,null,null);
11 }
12 public BiTreeNode(Object data,BiTreeNode lchild,BiTreeNode rchild){
13     this.data = data;
14     this.lchild = lchild;
15     this.rchild = rchild;
16 }
17 }
```

4. 二叉树类的描述

此二叉树类基于二叉链式存储结构实现。

```
1  package ch05;
2  public class BiTree {
3  private BiTreeNode root;                               //二叉树的头结点
4  public BiTree(){
5      root = null;
6  }
7  public BiTree(BiTreeNode root){
8      this.root = root;
9  }
10 }
```

二叉树的创建操作和遍历操作比较重要，将在下面的章节中进行详细介绍。

视频讲解

5.2.4　二叉树的遍历

二叉树的遍历是指沿着某条搜索路径访问二叉树的结点，每个结点被访问的次数有且仅有一次。

1. 二叉树的遍历方法

二叉树通常可划分为 3 个部分，即根结点、左子树和右子树。根据 3 个部分的访问顺序

不同,可将二叉树的遍历方法分为以下几种。

1) 层次遍历

自上而下、从左到右依次访问每层的结点。

2) 先序遍历

先访问根结点,再先序遍历左子树,最后先序遍历右子树。又称前序遍历。

3) 中序遍历

先中序遍历左子树,再访问根结点,最后中序遍历右子树。

4) 后序遍历

先后序遍历左子树,再后序遍历右子树,最后访问根结点。

图 5.7 描述了先序遍历、中序遍历和后序遍历序列的结点排列规律。

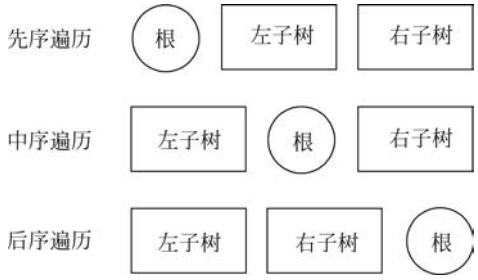


图 5.7 二叉树遍历序列的结点排列规律

2. 二叉树遍历操作实现的递归算法

【算法 5.1】 先序遍历。

```
1  public void preOrder(BiTreeNode root){
2      System.out.print(root.data + " ");          //访问根结点
3      preOrder(root.lchild);                      //先序遍历左子树
4      preOrder(root.rchild);                      //先序遍历右子树
5  }
```

【算法 5.2】 中序遍历。

```
1  public void inOrder(BiTreeNode root){
2      inOrder(root.lchild);                       //先序遍历左子树
3      System.out.print(root.data + " ");          //访问根结点
4      inOrder(root.rchild);                       //先序遍历右子树
5  }
```

【算法 5.3】 后序遍历。

```
1  public void postOrder(BiTreeNode root){
2      preOrder(root.lchild);                      //先序遍历左子树
3      preOrder(root.rchild);                      //先序遍历右子树
4      System.out.print(root.data + " ");          //访问根结点
5  }
```

3. 二叉树遍历操作实现的非递归算法

二叉树遍历操作的递归算法结构简洁,易于实现,但是在时间上开销较大,运行效率较

低。为了解决这一问题，可以将递归算法转换为非递归算法，转换方式有以下两种：

- 使用临时遍历保存中间结果，用循环结构代替递归过程；
- 利用栈保存中间结果。

二叉树遍历操作实现的非递归算法利用栈结构通过回溯访问二叉树的每个结点。

1）先序遍历

先序遍历从二叉树的根结点出发，沿着该结点的左子树向下搜索，每遇到一个结点先访问该结点，并将该结点的右子树入栈。先序遍历左子树完成后再从栈顶弹出右子树的根结点，然后采用相同的方法先序遍历右子树，直到二叉树的所有结点都被访问。先序遍历的主要步骤如下。

（1）将二叉树的根结点入栈。

（2）若栈非空，将结点从栈中弹出并访问。

（3）依次访问当前访问结点的左孩子结点，并将当前结点的右孩子结点入栈。

（4）重复步骤(2)和(3)，直到栈为空。

【算法 5.4】 先序遍历。

```
1  public void preOrder2() throws Exception{
2      BiTreeNode p = root;
3      if(p!= null){
4          LinkStack s = new LinkStack();                  //构造存储结点的栈
5          s.push(p);
6          while(!s.isEmpty()){
7              p = (BiTreeNode) s.pop();
8              System.out.print(p.data + " ");             //访问当前结点
9              while(p!= null){
10                 if(p.lchild!= null)                     //访问左孩子结点
11                     System.out.print(p.lchild.data + " ");
12                 if(p.rchild!= null)                     //将右孩子结点入栈
13                     s.push(p.rchild);
14                 p = p.lchild;
15             }
16         }
17     }
18 }
```

2）中序遍历

中序遍历从二叉树的根结点出发，沿着该结点的左子树向下搜索，每遇到一个结点就使其入栈，直到结点的左孩子结点为空。再从栈顶弹出结点并访问，然后采用相同的方法中序遍历结点的右子树，直到二叉树的所有结点都被访问。中序遍历的主要步骤如下。

（1）将二叉树的根结点入栈。

（2）若栈非空，则将栈顶结点的左孩子结点依次入栈，直到栈顶结点的左孩子结点为空。

（3）将栈顶结点弹出并访问，并使栈顶结点的右孩子结点入栈。

（4）重复步骤(2)和(3)，直到栈为空。

【算法 5.5】 中序遍历。

```
1  public void inOrder2() throws Exception{
```

```
2       BiTreeNode p = root;
3       if(p!= null){
4           LinkStack s = new LinkStack();
5           s.push(p);                                  //根结点入栈
6           while(!s.isEmpty()){
7               while(p.lchild!= null){                 //结点的左孩子结点依次入栈
8               p = p.lchild;
9               s.push(p);
10          }
11          p = (BiTreeNode)s.pop();                    //访问结点
12          System.out.print(p.data + " ");
13          if(p.rchild!= null)                         //结点的右孩子结点入栈
14              s.push(p.rchild);
15          }
16      }
17  }
```

3) 后序遍历

后序遍历从二叉树的根结点出发,沿着该结点的左子树向下搜索,每遇到一个结点需要判断其是否为第一次经过,若是则使结点入栈,后序遍历该结点的左子树,完成后再遍历该结点的右子树,最后从栈顶弹出该结点并访问。后序遍历算法的实现需要引入两个变量,一个为访问标记变量 flag,用于标记栈顶结点是否被访问,若 flag=true,则证明该结点已被访问,其左子树和右子树已经遍历完毕,可继续弹出栈顶结点,否则需要先遍历栈顶结点的右子树;另一个为结点指针 t,指向最后一个被访问的结点,查看栈顶结点的右孩子结点,证明此结点的右子树已经遍历完毕,栈顶结点可出栈并访问。后序遍历的主要步骤如下。

(1) 将二叉树的根结点入栈,t 赋值为空。

(2) 若栈非空,则将栈顶结点的左孩子结点依次入栈,直到栈顶结点的左孩子结点为空。

(3) 若栈非空,则查看栈顶结点的右孩子结点,若右孩子结点为空或者与 p 相等,则弹出栈顶结点并访问,同时使 t 指向该结点,并置 flag 为 true;否则将栈顶结点的右孩子结点入栈,并置 flag 为 false。

(4) 若 flag 为 true,重复步骤(3);否则重复步骤(2)和(3),直到栈为空。

【算法 5.6】 后序遍历。

```
1   public void postOrder2() throws Exception{
2       BiTreeNode p = root;
3       boolean flag = false;
4       BiTreeNode t = null;
5       if(p!= null){
6           LinkStack s = new LinkStack();
7           s.push(p);                                  //根结点入栈
8           while(!s.isEmpty()){
9               p = (BiTreeNode)s.peek();
10              while(p.lchild!= null){                 //将栈顶结点的左孩子结点依次入栈
11              p = p.lchild;
12              s.push(p);
13              }
```

```
            while(!s.isEmpty()&&flag){
                if(p.rchild==t||p.rchild==null){//左、右子树已经遍历完毕,访问结点
                    System.out.print(p.data+" ");
                    flag=true;
                    t=p;
                s.pop();
                }
                else{                                   //右孩子结点入栈
                    s.push(p.rchild);
                    flag=false;
                }
            }
        }
    }
}
```

4）层次遍历

层次遍历操作是从根结点出发，自上而下、从左到右依次遍历每层的结点，可以利用队列先进先出的特性进行实现。先将根结点入队，然后将队首结点出队并访问，都将其孩子结点依次入队。层次遍历的主要步骤如下。

（1）将根结点入队。

（2）若队非空，则取出队首结点并访问，将队首结点的孩子结点入队。

（3）重复执行步骤(2)，直到队为空。

【算法 5.7】 层次遍历。

```
public void order() throws Exception{
    BiTreeNode p=root;
    while(p!=null){
        LinkQueue q=new LinkQueue();
        q.offer(p);
        while(!q.isEmpty()){
            p=(BiTreeNode) q.poll();
            System.out.print(p.data+" ");
            if(p.lchild!=null)
                q.offer(p.lchild);
            if(p.rchild!=null)
                q.offer(p.rchild);
        }
    }
}
```

对于有 n 个结点的二叉树，因为每个结点都只访问一次，所以以上 4 种遍历算法的时间复杂度均为 $O(n)$。

4 种遍历算法的实现均利用了栈或队列，增加了额外的存储空间，存储空间的大小为遍历过程中栈或队列需要的最大容量。对于栈来说，其最大容量即为树的高度，在最坏情况下有 n 个结点的二叉树的高度为 n，所以其空间复杂度为 $O(n)$；对于队列来说，其最大容量为二叉树相邻两层的最大结点总数，与 n 呈线性关系，所以其空间复杂度也为 $O(n)$。

5.2.5 二叉树遍历算法的应用

二叉树的遍历操作是实现对二叉树其他操作的一个重要基础,本节介绍了二叉树遍历算法在许多应用问题中的运用。

1. 二叉树上的查找算法

二叉树上的查找是在二叉树中查找值为 x 的结点,若找到则返回该结点,否则返回空值,可以在二叉树的先序遍历过程中进行查找,主要步骤如下。

(1) 若二叉树为空,则不存在值为 x 的结点,返回空值;否则将根结点的值与 x 进行比较,若相等,则返回该结点。

(2) 若根结点的值与 x 的值不等,则在左子树中进行查找,若找到,则返回该结点。

(3) 若没有找到,则在根结点的右子树中进行查找,若找到,则返回该结点,否则返回空值。

【算法 5.8】 二叉树查找算法。

```
public BiTreeNode searchNode(BiTreeNode t,Object x){
      if(t == null)                                          //树为空
          return null;
      else{
          if(t.data.equals(x))                               //x 与根结点进行比较
              return t;
          else{
              BiTreeNode lresult = searchNode(t.lchild,x);   //在左子树中查找
              if(lresult == null)
                  return searchNode(t.rchild,x);             //在右子树中查找
              else
                  return lresult;
          }

      }
  }
```

2. 统计二叉树的结点个数的算法

二叉树的结点个数等于根结点加上左、右子树的结点的个数,可以利用二叉树的先序遍历序列,引入一个计数变量 count,count 的初值为 0,每访问根结点一次就将 count 的值加 1,其主要操作步骤如下。

(1) count 值初始化为 0。

(2) 若二叉树为空,则返回 count 值。

(3) 若二叉树非空,则 count 值加 1,统计根结点的左子树的结点个数,并将其加到 count 中;统计根结点的右子树的结点个数,并将其加到 count 中。

【算法 5.9】 统计二叉树的结点个数。

```
public int nodeCount(BiTreeNode t){
      int count = 0;
      if(t!= null)
      {
          count++;                                           //计数根结点
      count = count + nodeCount(t.lchild);                   //左子树的结点个数
```

```
        count = count + nodeCount(t.rchild);                    //右子树的结点个数
        }
        return count;
    }
```

3. 求二叉树的深度

二叉树的深度是所有结点的层次数的最大值加 1，也就是左子树和右子树的深度的最大值加 1，可以采用后序遍历的递归算法解决此问题，其主要步骤如下。

(1) 若二叉树为空，返回 0。

(2) 若二叉树非空，求左子树和右子树的深度。

(3) 比较左、右子树的深度，取最大值加 1 即为二叉树的深度。

【算法 5.10】 求二叉树的深度。

```
public int getDepth(BiTreeNode t){
        if(t == null)                                           //二叉树为空
            return 0;
        else                                                    //二叉树非空
        {
            int ldepth = getDepth(t.lchild);
            int rdepth = getDepth(t.rchild);
            if(ldepth < rdepth)
                return rdepth + 1;
            else
                return ldepth + 1;
        }
    }
```

5.2.6 二叉树的建立

二叉树遍历操作可使非线性结构的树转换成线性序列。先序遍历序列和后序遍历序列反映父结点和孩子结点间的层次关系，中序遍历序列反映兄弟结点间的左右次序关系。因为二叉树是具有层次关系的结点构成的非线性结构，并且每个结点的孩子结点具有左右次序，所以已知一种遍历序列无法唯一确定一棵二叉树，只有同时知道中序和先序遍历序列，或者同时知道中序和后序遍历序列，才能同时确定结点的层次关系和结点的左右次序，才能唯一确定一棵二叉树。

1. 由中序和先序遍历序列建立二叉树

其主要步骤如下。

(1) 取先序遍历序列的第一个结点作为根结点，序列的结点个数为 n。

(2) 在中序遍历序列中寻找根结点，其位置为 i，可确定在中序遍历序列中根结点之前的 i 个结点构成的序列为根结点的左子树中序遍历序列，根结点之后的 $n-i-1$ 个结点构成的序列为根结点的右子树中序遍历序列。

(3) 在先序遍历序列中根结点之后的 i 个结点构成的序列为根结点的左子树先序遍历序列，先序遍历序列之后的 $n-i-1$ 个结点构成的序列为根结点的右子树先序遍历序列。

(4) 对左、右子树重复步骤(1)～(3)，确定左、右子树的根结点和子树的左、右子树。

(5) 算法递归进行即可建立一棵二叉树。

假设二叉树的先序遍历序列为 ABECFG、中序遍历序列为 BEAFCG,由中序和先序遍历序列建立二叉树的过程如图 5.8 所示。

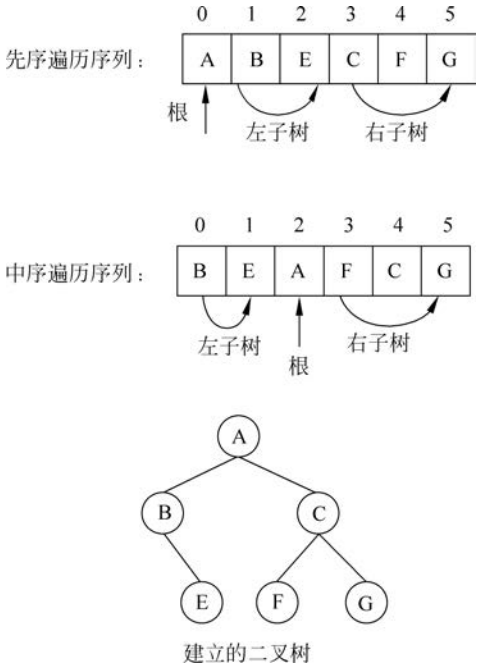


图 5.8 由中序和先序遍历序列建立二叉树

【算法 5.11】 由中序和先序遍历序列建立二叉树。

```
1  public BiTree(String preOrder,String inOrder,int pre,int in,int n){
2      if(n>0){
3          char c = preOrder.charAt(pre);                    //c 为先序序列的根结点
4          int i = 0;
5          for(;i<n;i++){                                    //i 为根结点在中序遍历序列中的位置
6              if(inOrder.charAt(i + in) == c){
7                  break;
8              }
9          }
10         root = new BiTreeNode(c);
11         root.lchild = new BiTree(preOrder,inOrder,pre + 1,in,i).root;
                                                             //递归寻找左子树的根结点
12         root.rchild = new BiTree(preOrder,inOrder,pre + i + 1,in + i + 1,n - i - 1).root;
                                                             //递归寻找右子树的根结点
13     }
14 }
```

2. 由标明空子树的先序遍历序列创建二叉树

其主要步骤如下。

(1) 从先序遍历序列中依次读取字符。

(2) 若字符为 #,则建立空子树。

(3) 建立左子树。

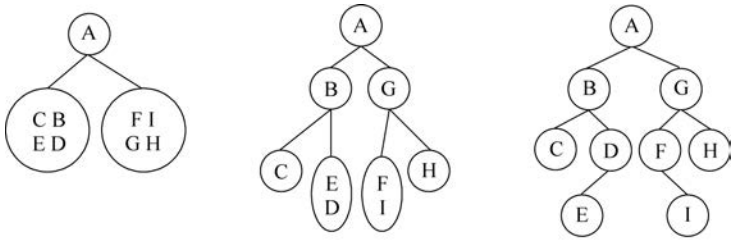
(4) 建立右子树。

【算法 5.12】 由标明空子树的先序遍历序列建立二叉树。

```
1  public BiTree(String preOrder,int i){                    //i 为常数 0
2      char c = preOrder.charAt(i);                         //取字符
3      if(c!= '#'){
4          root = new BiTreeNode(c);                        //建立根结点
5          root.lchild = new BiTree(preOrder,++i).root; //建立左子树
6          root.rchild = new BiTree(preOrder,++i).root; //建立右子树
7      }
8      else {
9          root = null;
10     }
11 }
```

【例 5.3】 已知二叉树的中序和后序序列分别为 CBEDAFIGH 和 CEDBIFHGA，试构造该二叉树。

解：二叉树的构造过程如图 5.9 所示。图(c)即为构造出的二叉树。



(a) 第一步　　(b) 第二步　　(c) 第三步

图 5.9　二叉树的构造过程

视频讲解

5.3　哈夫曼树及哈夫曼编码

由于目前常用的图像、音频、视频等多媒体信息数据量大，必须对它们采用数据压缩技术来存储和传输。数据压缩技术通过对数据进行重新编码来压缩存储，以便减少数据占用的存储空间，在使用时再进行解压缩，恢复数据的原有特性。

其压缩方法主要有有损压缩和无损压缩两种。有损压缩是指压缩过程中可能会丢失数据信息，如将 BMP 位图压缩成 JPEG 格式的图像，会有精度损失；无损压缩是指压缩存储数据的全部信息，确保解压后的数据不丢失。哈夫曼编码是数据压缩技术中的无损压缩技术。

5.3.1　哈夫曼树的基本概念

1. 结点间的路径

结点间的路径是指从一个结点到另一个结点所经过的结点序列。从根结点到 X 结点有且仅有一条路径。

2. 结点的路径长度

结点的路径长度是指从根结点到结点的路径上的边数。

3. 结点的权

结点的权是指人给结点赋予的一个具有某种实际意义的数值。

4. 结点的带权路径长度

结点的带权路径长度是指结点的权值和结点的路径长度的乘积。

5. 树的带权路径长度

树的带权路径长度是指树的叶结点的带权路径长度之和。

6. 最优二叉树

最优二叉树是指给定 n 个带有权值的结点作为叶结点构造出的具有最小带权路径长度的二叉树。最优二叉树也叫哈夫曼树。

5.3.2 哈夫曼树的构造

给定 n 个叶结点,它们的权值分别是$\{w_1,w_2,\cdots,w_n\}$,构造相应的哈夫曼树的主要步骤如下。

(1) 构造由 n 棵二叉树组成的森林,每棵二叉树只有一个根结点,根结点的权值分别为$\{w_1,w_2,\cdots,w_n\}$。

(2) 在森林中选取根结点权值最小和次小的两棵二叉树分别作为左子树和右子树去构造一棵新的二叉树,新二叉树的根结点权值为两棵子树的根结点权值之和。

(3) 将两棵二叉树从森林中删除,并将新的二叉树添加到森林中。

(4) 重复步骤(2)和(3),直到森林中只有一棵二叉树,此二叉树即为哈夫曼树。

假设给定的权值为{1,2,3,4,5},图 5.10 展示了哈夫曼树的构造过程。

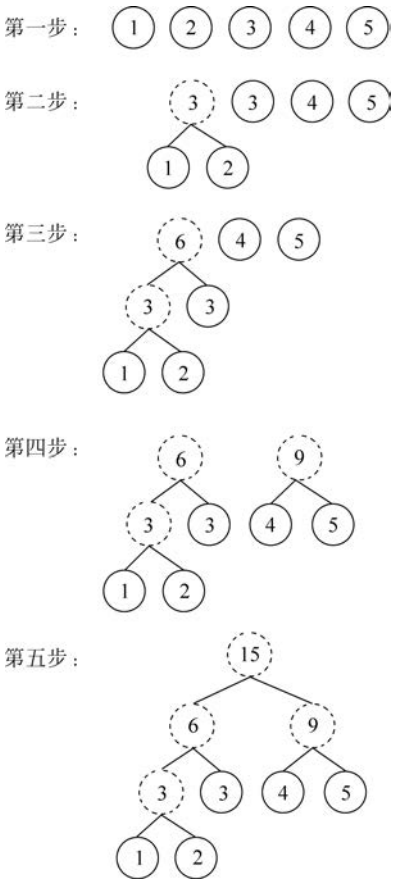


图 5.10 哈夫曼树的构造过程

【例 5.4】 对于给定的一组权值 $W=(5,2,9,11,8,3,7)$，试构造相应的哈夫曼树，并计算它的带权路径长度。

解：构造的哈夫曼树如图 5.11 所示。

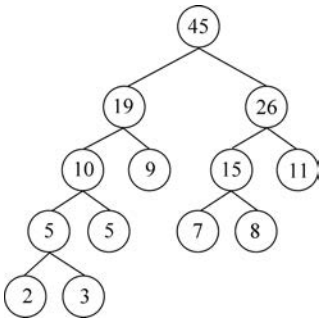


图 5.11 哈夫曼树

树的带权路径长度如下：

$$WPL=2\times4+3\times4+5\times3+7\times3+8\times3+9\times2+11\times2=120$$

5.3.3 哈夫曼编码

在传送信息时需要将信息符号转换成二进制组成的符号串，一般每个字符由一个字节或两个字节表示，即 8 或 16 个位数。为了提高存储和传输效率，需要设计对字符集进行二进制编码的规则，使得利用这种规则对信息进行编码时编码位数最小，即需要传输的信息量最小。

哈夫曼编码是一种变长的编码方案，数据的编码因其使用频率的不同而长短不一，使用频率高的数据其编码较短，使用频率低的数据其编码较长，从而使所有数据的编码总长度最短。各数据的使用频率通过在全部数据中统计重复数据的出现次数获得。

又因为在编码序列中若使用前缀相同的编码来表示不同的字符会造成二义性，额外的分隔符号会造成传输信息量的增加，为了省去不必要的分隔符号，要求每一个字符的编码都不是另一个字符的前缀，即每个字符的编码都是前缀编码。

利用哈夫曼树构造出的哈夫曼编码是一种最优前缀编码，构造的主要步骤如下。

(1) 对于具有 n 个字符的字符集，将字符的频度作为叶结点的权值，产生 n 个带权叶结点。

(2) 根据上面章节中介绍的构造哈夫曼树的方法利用 n 个叶结点构造哈夫曼树。

(3) 根据哈夫曼编码规则将哈夫曼树中的每一条左分支标记为 0、每一条右分支标记为 1，则可得到每个叶结点的哈夫曼编码。

哈夫曼编码的译码过程是构造过程的逆过程，从哈夫曼树的根结点开始对编码的每一位进行判别，如果为 0 则进入左子树，如果为 1 则进入右子树，直到到达叶结点，即译出了一个字符。

5.3.4 构造哈夫曼树和哈夫曼编码的类的描述

构造哈夫曼树需要从子结点到父结点的操作，译码时需要从父结点到子结点的操作，所以为了提高算法的效率将哈夫曼树的结点设计为三叉链式存储结构。一个数据域存储结点的权值，一个标记域 flag 标记结点是否已经加入到哈夫曼树中，3 个指针域分别存储着指向父结点、孩子结点的地址。

结点类的描述如下：

```
1  package ch05;
2  public class HuffmanNode {
3  public int weight;                          //结点的权值
4  public int flag;                            //标记是否已加入哈夫曼树
5  public HuffmanNode parent,lchild,rchild;    //父结点和孩子结点
```

```
6
7  public HuffmanNode(){
8      weight = 0;
9      flag = 0;
10     parent = lchild = rchild = null;
11 }
12 public HuffmanNode(int weight){
13     this.weight = weight;
14     flag = 0;
15     parent = lchild = rchild = null;
16 }
17 }
```

【算法 5.13】 构造哈夫曼树。

```
1  public class HuffmanTree {
2  public HuffmanTree(int[ ] w){
3      int l = w.length;                                    //字符的个数
4      int n = 2 * l - 1;                                   //哈夫曼树的结点数
5      HuffmanNode []node = new HuffmanNode[n];
6      for(int i = 0;i < l;i++){                            //构造权值为 w[i]的叶结点
7          node[i] = new HuffmanNode(w[i]);
8      }
9      for(int i = n;i > 0;i -- ){                          //构造哈夫曼树
10         HuffmanNode m1 = selectMin(node,i - 1);
11         m1.flag = 1;
12         HuffmanNode m2 = selectMin(node,i - 1);
13         m2.flag = 1;
14         node[i].lchild = m1;
15         node[i].rchild = m2;
16         node[i].weight = m1.weight + m2.weight;
17         m1.parent = node[i];
18         m2.parent = node[i];
19     }
20 }
21 public HuffmanNode selectMin(HuffmanNode[ ] node,int i){ //寻找不在树中的权值最小的点
22     HuffmanNode min = node[i];
23     for(int j = 0;j <= i;j++){
24         if(node[j].weight < min.weight&&node[j].flag == 0)
25             min = node[j];
26     }
27     return min;
28 }
```

【算法 5.14】 求哈夫曼编码。

```
1  public int[][ ] HuffmanCode(HuffmanNode[ ] node,int n){
2      int [][ ] HuffmanCode = new int[n][n];               //存储哈夫曼编码
3      for(int i = 0;i < n;i++){                            //构造 n 个字符的哈夫曼编码
```

```
            int k = n - 1;
            HuffmanNode t = node[i], p = t.parent;
            for(; p!= null; p = p.parent){
                if(p.lchild == t)
                    HuffmanCode[i][k--] = 0;
                else
                    HuffmanCode[i][k--] = 1;
            }
        }
        return HuffmanCode;
```

【例 5.5】 已知某字符串 s 中共有 8 种字符，各种字符分别出现两次、一次、4 次、5 次、7 次、3 次、4 次和 9 次，对该字符串用[0,1]进行前缀编码，问该字符串的编码至少有多少位？

解：以各字符出现的次数作为叶子结点的权值构造的哈夫曼编码树如图 5.12 所示。其带权路径长度＝2×5＋1×5＋3×4＋5×3＋9×2＋4×3＋4×3＋7×2＝98，所以该字符串的编码长度至少为 98 位。

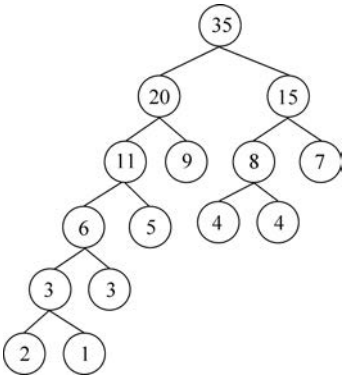


图 5.12　哈夫曼编码树

5.4　树 和 森 林

5.4.1　树的存储结构

一棵树包含各结点间的层次关系和兄弟关系，两种关系的存储结构不同。

- 树的层次关系必须采用链式存储结构存储，通过链连接父结点和孩子结点。
- 一个结点的多个孩子结点（互称兄弟结点）之间是线性关系，可以采用顺序存储结构或者链式存储结构。

1. 树的父母孩子链表

树的父母孩子链表采用顺序存储结构存储多个孩子结点，其中 children 数组存储多个孩子结点，各结点的 children 数组元素长度不同，为孩子个数。树的父母孩子链表的存储结构如图 5.13 所示。

2. 树的父母孩子兄弟链表

树的父母孩子兄弟链表采用链式存储结构存储多个孩子结点，结点结构如图 5.14 所示，其中 child 链指向一个孩子结点，sibling 链指向下一个兄弟结点。

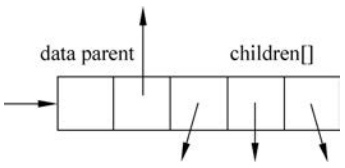


图 5.13　树的父母孩子链表的存储结构

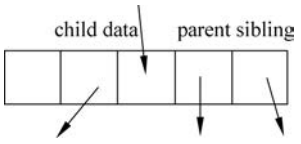


图 5.14　树的父母孩子兄弟链表的存储结构

森林也可以使用父母孩子兄弟链表进行存储,这种存储结构实际上是把一棵树转换成一棵二叉树存储。其存储规则如下。

(1) 每个结点采用 child 链指向其中一个孩子结点,多个孩子结点之间由 sibling 链连接起来,组成一条具有兄弟结点关系的单链表。

(2) 将每棵树采用树的父母孩子兄弟链表存储。

(3) 森林中的多棵树之间是兄弟关系,将这些树通过根的 sibling 链连接起来。

5.4.2 树的遍历规则

树的孩子优先遍历规则主要有两种,即先序遍历和后序遍历。树的遍历规则也是递归的。

(1) 树的先序遍历:访问根结点;按从左到右的次序遍历根的每一棵子树。

(2) 树的后序遍历:按从左到右的次序遍历根的每一棵子树;访问根结点。

树的层次遍历规则同二叉树。

5.5 实　　验

5.5.1 平衡二叉树检验

给定一个树的根结点,判断是否为平衡二叉树。

树的结点类如下:

```
class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;
    public TreeNode(int value){
        this.value = value;
    }
}
public class U5H1 {
    //给定一个二叉树,判断它是否为高度平衡的二叉树-->/平衡二叉树的子树也是平衡二叉树
    //解析:1.root 左树高度-右树高度≤1;
    //2.root 的左树是平衡点,右树也是平衡点

    public int height(TreeNode root) {
        if (root == null) return 0;
        int leftheight = height(root.left);
        int rightheight = height(root.right);

        if (leftheight >= 0 && rightheight >= 0 &&Math.abs(leftheight - rightheight) <= 1)
{
            return Math.max(leftheight,rightheight) + 1;
        }else {
            return -1;
        }
```

```
    }
    //时间复杂度 O(N)
    public boolean isBalanced(TreeNode root) {
        if (root == null) return true;
        return height(root) >= 0;
    }
}
```

5.5.2 查找最近公共祖先

给定一棵树的根结点和两个子结点，返回该两个子结点最近的公共祖先结点。

提示：用栈存储从根结点到每个结点的路径。

```
import java.util.Stack;

public class U5H2 {
    /* 孩子双亲表示法:通过链表求交点(用栈保存路径)
  1.用两个栈存储 p、q 路径(通过遍历找到根结点到指定结点的路径)
  2.求栈的大小
  3.让栈中多的元素出差值个元素
  4.开始出栈,直到栈顶元素相同,即此时是 LCA 最近公共祖先
   *
   *  */
//获取路径 root→node:
    //root 根结点,node 指定结点,stack 存储指定结点路径
    public boolean getPath(TreeNode root, TreeNode node, Stack<TreeNode> stack) {
        if (root == null || node == null) return false;   //若为空结点,则无路径
        stack.push(root);
        if (root == node) return true;                    //若根结点为 node,则找到了指定结点
//如果不是根结点,则需要在左右树寻找
        boolean flg = getPath(root.left, node, stack);    //左树寻找结点
        if (flg == true) {
            return true;
        }
        flg = getPath(root.right, node, stack);           //右树寻找结点
        if (flg == true) {
            return true;
        }
        stack.pop();                          //左右树都没找到指定结点 node,则需要弹出 pop()
        return false;
    }

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) return null;
        Stack<TreeNode> stack1 = new Stack<>();
        getPath(root, p, stack1);                         //存储 root→p 的路径到 stack1
        Stack<TreeNode> stack2 = new Stack<>();
        getPath(root, q, stack2);                         //存储 root→q 的路径到 stack2

        int size1 = stack1.size();                        //得到两个栈的大小
```

```
        int size2 = stack2.size();
        if (size1 > size2) {                                    //取两个栈差值个元素
            int size = size1 - size2;
            while (size != 0) {
                stack1.pop();           //size 大的栈 stack1 开始出元素,直至两个栈元素相同
                size--;
            }

            while (!stack1.isEmpty() && !stack2.isEmpty()) {    //两个栈都不为空
                if (stack1.peek() == stack2.peek()) { //两个栈顶元素相同,则找到公共祖先
                    return stack1.pop();
                } else {
                    stack1.pop();
                    stack2.pop();
                }
            }
        } else {
            int size = size2 - size1;
            while (size != 0) {
                stack2.pop();           //size 大的栈 stack2 开始出元素,直至两个栈元素相同
                size--;
            }
            while (!stack1.isEmpty() && !stack2.isEmpty()) {
                if (stack1.peek() == stack2.peek()) {
                    return stack1.pop();
                } else {
                    stack1.pop();
                    stack2.pop();
                }
            }
        }
        return null;
    }
}
```

视频讲解

5.5.3 由前序遍历和中序遍历构造二叉树

给出一个树的前序遍历和中序遍历,构造树并且输出树的后序遍历结果。

示例:

输入:{3,9,20,15,7}
{9,3,15,20,7}

输出:[9, 15, 7, 20, 3]

```
import java.util.ArrayList;
import java.util.List;

class U5H3 {
    public static TreeNode initTree(int[] preOrder, int pstart, int pend, int[] inOrder, int
instart, int inend) {
```

```
        if (pstart > pend || instart > inend) {
            return null;
        }
        int rootData = preOrder[pstart];
        TreeNode head = new TreeNode(rootData);
        //根据中序找到根结点所在位置(左边为左子树,右边为右子树)
        int rootIndex = findIndexInArray(inOrder, rootData, instart, inend);
        int offSet = rootIndex - instart - 1;
        //构建左子树
         TreeNode left = initTree(preOrder, pstart + 1, pstart + offSet + 1, inOrder,
instart, instart + offSet);
        //构建右子树
        TreeNode right = initTree(preOrder, pstart + offSet + 2, pend, inOrder, rootIndex + 1,
inend);
        head.left = left;
        head.right = right;
        return head;
    }
    private static int findIndexInArray(int[] inOrder, int rootData, int instart, int inend) {
        for (int i = instart; i <= inend; i++) {
            if(inOrder[i] == rootData){
                return i;
            }
        }
        return -1;
    }
    public static void main(String[] args) {
        int[] preOrder = {3,9,20,15,7};
        int[] inOrder = {9,3,15,20,7};
        TreeNode root = initTree(preOrder, 0, preOrder.length-1, inOrder, 0,inOrder.length-1);
        List<Integer> tmp = new ArrayList<>();
        System.out.println(postorderTraversal(root, tmp));
    }
    public static List<Integer> postorderTraversal(TreeNode root, List<Integer> tmp) {
        if(root == null) return tmp;
        postorderTraversal(root.left, tmp);
        postorderTraversal(root.right, tmp);
        tmp.add(root.val);
        return tmp;
    }
}
```

小　　结

(1) 树是数据元素之间具有层次关系的非线性结构,是由 n 个结点构成的有限集合。与线性结构不同,树中的数据元素具有一对多的逻辑关系。

(2) 二叉树是特殊的有序树,它也是由 n 个结点构成的有限集合。当 $n=0$ 时称为空二叉树。二叉树的每个结点最多只有两棵子树,子树也为二叉树,互不相交且有左、右之分,分

别称为左二叉树和右二叉树。

(3) 二叉树的存储结构分为两种,即顺序存储结构和链式存储结构。二叉树的顺序存储结构是指将二叉树的各个结点存放在一组地址连续的存储单元中,所有结点按结点序号进行顺序存储;二叉树的链式存储结构是指将二叉树的各个结点随机存放在存储空间中,二叉树的各结点间的逻辑关系由指针确定。

(4) 二叉树具有先序遍历、中序遍历、后序遍历和层次遍历 4 种遍历方式。

(5) 最优二叉树是指给定 n 个带有权值的结点作为叶结点构造出的具有最小带权路径长度的二叉树,也叫哈夫曼树。

(6) 哈夫曼编码是数据压缩技术中的无损压缩技术,是一种变长的编码方案,使所有数据的编码总长度最短。

习 题 5

一、选择题

1. 如果结点 A 有 3 个兄弟,B 是 A 的双亲,则结点 B 的度是(　　)。

A. 1　　B. 2　　C. 3　　D. 4

2. 设二叉树有 n 个结点,则其深度为(　　)。

A. $n-1$　　B. n　　C. $\lfloor \log_2 n \rfloor + 1$　　D. 不能确定

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

A. 空或只有一个结点　　B. 高度等于其结点数

C. 任一结点无左孩子　　D. 任一结点无右孩子

4. 线索二叉树中某结点 R 没有左孩子的充要条件是(　　)。

A. R.lchild=null　　B. R.ltag=0　　C. R.ltag=1　　D. R.rchild=null

5. 深度为 k 的完全二叉树最少有(　　)个结点、最多有(　　)个结点,具有 n 个结点的完全二叉树按层序从 1 开始编号,则编号最小的叶子结点的序号是(　　)。

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

E. 2^{k+1}　　F. $2^{k+1}-1$　　G. $2^{k-1}+1$　　H. 2^k

6. 一个高度为 h 的满二叉树共有 n 个结点,其中有 m 个叶子结点,则(　　)成立。

A. $n=h+m$　　B. $h+m=2n$　　C. $m=h-1$　　D. $n=2m-1$

7. 任何一棵二叉树的叶结点在前序、中序、后序遍历序列中的相对次序(　　)。

A. 肯定不发生改变　　B. 肯定发生改变

C. 不能确定　　D. 有时发生变化

8. 如果 T′是由有序树 T 转换而来的二叉树,那么 T 中结点的前序序列就是 T′中结点的(　　)序列,T 中结点的后序序列就是 T′中结点的(　　)序列。

A. 前序　　B. 中序　　C. 后序　　D. 层序

9. 设森林中有 4 棵树,树中结点的个数依次为 n_1、n_2、n_3、n_4,则把森林转换成二叉树后其根结点的右子树上有(　　)个结点、根结点的左子树上有(　　)个结点。

A. n_1-1　　B. n_1　　C. $n_1+n_2+n_3$　　D. $n_2+n_3+n_4$

10. 讨论树、森林和二叉树的关系目的是为了(　　)。

A. 借助二叉树上的运算方法去实现对树的一些运算

B. 将树、森林按二叉树的存储方式进行存储并利用二叉树的算法解决树的有关问题

C. 将树、森林转换成二叉树

D. 体现一种技巧，没有什么实际意义

二、填空题

1. 树是 $n(n\geqslant 0)$ 个结点的有限集合，在一棵非空树中有_____个根结点，其余结点分成 $m(m>0)$ 个_____的集合，每个集合都是根结点的子树。

2. 树中某结点的子树的个数称为该结点的_____，子树的根结点称为该结点的_____，该结点称为其子树根结点的_____。

3. 一棵二叉树的第 $i(i\geqslant 1)$ 层最多有_____个结点；一棵有 $n(n>0)$ 个结点的满二叉树共有_____个叶结点和_____个非终端结点。

4. 设高度为 h 的二叉树上只有度为 0 和度为 2 的结点，该二叉树的结点数可能达到的最大值是_____、最小值是_____。

5. 在深度为 k 的二叉树中所含叶结点的个数最多为_____。

6. 具有 100 个结点的完全二叉树的叶结点数为_____。

7. 已知一棵度为 3 的树有两个度为 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_____个叶结点。

8. 某二叉树的前序遍历序列是 ABCDEFG、中序遍历序列是 CBDAFGE，则其后序遍历序列是_____。

9. 在具有 n 个结点的二叉链表中共有_____个指针域，其中_____个指针域用于指向其左、右孩子，剩下的_____个指针域则是空的。

10. 在有 n 个叶子的哈夫曼树中叶结点总数为_____、分支结点总数为_____。

三、算法设计题

1. 设计算法求二叉树的结点个数；按前序次序打印二叉树中的叶结点；求二叉树的深度。

2. 设计算法判断一棵二叉树是否为完全二叉树。

3. 使用栈将 Tree 类中的递归遍历算法实现为非递归遍历算法。

4. 编写一个非递归算法求出二叉搜索树中的所有结点的最大值，若树为空则返回空值。

5. 编写一个算法求出一棵二叉树中叶结点的总数，参数初始指向二叉树的根结点。