桂林电子科技大学

实验 3 二叉树的基本操作、算法和应用 实验报告

实验名称	二叉树的基本操作、算法和应用						
院　系	计算机与信息安全学院		专业	信息安全			
学　号	2000301708		姓名	蔡响			
实验日期	2021	年	11	月	5	日	

辅导员意见：

成绩　　辅导员签　名

一、实验目的

1. 掌握二叉树的建立和遍历
2. 掌握二叉树遍历的应用
3. 掌握线索二叉树的建立和遍历
4. 掌握hufmman编码

二、实验内容

1. 递归建立和层次遍历二叉树
2. 二叉树的深度非递归遍历
3. 统计二叉树中右孩子节点的个数
4. 线索二叉树的建立和遍历
5. 哈夫曼编码

三、实验环境

在PTA平台进行实验

四、实验要求

根据每个实训的要求完成代码提交和测评

五、实验步骤

描述算法的原理或实现流程（测评完全正确或部分正确的实训）

六、问题记录和实验总结

递归建立和层次遍历二叉树

读题：

建立二叉树的函数和层次遍历二叉树的函数

思路：

建立二叉树的时候需要考虑对应的左孩子以及右孩子的问题，由于后续是层次遍历，所以可以采用设定下标的方式进行判定对应的左孩子以及右孩子。

层次遍历主要是采用队列实现。

实现：

```
//建立二叉树中判断左孩子与右孩子
{
            p = FrontQueue_link(queue);
            if (s != NULL && p != NULL)
                if (count % 2 == 1)
                    p->leftchild = s;
                else
                    p->rightchild = s;
            if (count % 2 == 0)
                DeQueue_link(queue);
}


//层序遍历
void LevelOrder(BinTree bt)
{
    BinTree p;
    LinkQueue queue = SetNullQueue_Link();
    if (bt == NULL)
        return;
    p = bt;
    EnQueue_link(queue, p);
    while (!IsNullQueue_Link(queue))
    {
        p = FrontQueue_link(queue);
        DeQueue_link(queue);
        printf("%c ", p->data);
        if (p->leftchild != NULL)
            EnQueue_link(queue, p->leftchild);
        if (p->rightchild != NULL)
            EnQueue_link(queue, p->rightchild);
    }
}
```

非递归方式统计二叉树中右孩子结点的个数

读题：要求使用非递归的方式统计二叉树中右孩子结点的个数

思路：递归的话代码实现以及思路就会比较简单，如果是非递归统计的话，我是采用层次遍历的方法进行遍历所有的节点，通过下标来确定是否是右节点。

实现：采用层次遍历的方式进行，定义一个index用于统计该节点的下标，只要该节点下标能被2整除，则为右孩子。

二叉树的非递归遍历

读题： 本题要求采用栈实现对二叉树的非递归遍历，包括先序、中序和后序三种遍历方式。

思路： 首先需要了解先序，中序，后序遍历：

- 先序：根左右

1）从二叉树的根结点开始，将其压入栈

2）当栈不为空时，取栈顶元素作为当前树（左子树或右子树），赋值给p并弹出；否则，遍历结束

3）如果p不为空，访问p并将p的右子树、左子树（可能为空树）依次（优先访问左子树）压入栈中

4）返回 2）

- 中序：左根右与后序：左右根

1）从二叉树的根结点开始，将其赋值给p作为当前结点

2）不断将p及p的左子树压入栈，直到p为空

3）取栈顶元素并弹出，赋值给p作为当前结点，访问p

4）将p的右子树赋值给p作为当前结点

5）如果栈不为空（遍历结束） 或者 p不为空，返回2）

实现：

```
void Order(PBintree t){
    stack<pnode> s;
    PBintree p = t;
    while(p||!s.empty()){
        while(p){
            s.push(p);
            p=p->lchild?p->lchild:p->rchild;
        }                                          //循环到当前处理的点
        p=s.top();                                 //访问栈顶二叉树的根
        s.pop();
        cout<<p->n;
```

```
        if(!s.empty() && p==(s.top())->lchild)    //栈不为空，且从左子树退回
            p=(s.top())->rchild;                  //直接进入栈顶元素的右子树
        else
            p=NULL;                               //从右子树回来，退到上一层处理
    }
}
```

线索二叉树的建立和遍历

读题： 要求输入为先序序列，实现对建立中序线索二叉树和中序遍历中序线索二叉树。

思路：

输入为先序序列比较简单，采用递归中序建立的方式进行。

进中序线索二叉树的建立也比较简单，只要判断该节点是否有左孩子与右孩子即可，如果节点为null，则将对应的节点标记为1，否则保留初始化0。

输出就进行正常的中序遍历，并打印左右指针信息。

实现：

```
ClueTree CreateBiTree()
{
    char val;
    auto head = new Clue();
    cin >> val;
    if (val == '@')
    {
        return nullptr;
    }
    else if (val != '@')
    {
        head->value = val;
        head->left_flage = 0;
        head->right_flage = 0;
        head->leftchild = CreateBiTree();
        head->rightchild = CreateBiTree();
        if(!head->leftchild){
            head->left_flage = 1;
        }
        if(!head->rightchild){
            head->right_flage = 1;
        }
    }
    return head;
}
void PrintClueTree(ClueTree bt)
{
    if(!bt){
        return;
    }
    PrintClueTree(bt->leftchild);
    cout << bt->value <<"  "<<bt->left_flage<<"  "<<bt->right_flage<< endl;
    PrintClueTree(bt->rightchild);
    return;
```

```
}
```

哈夫曼编码

读题： 本题要求字符的哈夫曼编码，注意建立的哈夫曼树严格按照左小右次小的顺序，并且哈夫曼编码时严格按照左‘0’右‘1’进行编码。

思路：

输入是各个字符在通信中出现的频率。回想一下自己实现哈弗曼编码的过程，先将权重从小到大进行排序，将最小的两个权值的节点作为兄弟节点，然后将这两个点生成的权重加入权重的排序进行。。。

我们知道，我们在画哈夫曼树的时候是从下至上的，而平时树的创建是通过根节点，至上而下的，故哈夫曼树的创建不同往昔。

**

```
(1)选出哈夫曼树的某一个叶子结点。
(2)利用其双亲指针parent找到其双亲结点。
(3)利用找到的双亲结点的指针域中的lchild和rchild，判断该结点是双亲的左孩子还是右孩子。若该结点是其双亲结点的左孩子，则生成代码0；若该结点是其双亲结点的右孩子，则生成代码1。
(4)由于生成的编码与要求的编码反序，将所生成的编码反序。
(5)重复步骤(1)~(4)，直到所有结点都回溯完。
反序方法：首先将生成的编码从后向前依次存放在一个临时的一维数组中，并设一个指针start指示编码在该一维数组中的起始位置。当某个叶子结点的编码完成时，从临时的一维数组的start处将编码复制到该字符对应的bits中即可。
```

实现：

哈夫曼解码

哈夫曼解码过程:从哈夫曼树的根结点出发，依次识别电文的中的二进制编码，如果为0，则走向左孩子，否则走向右孩子，走到叶结点时，就可以得到相应的解码字符。

算法如下：

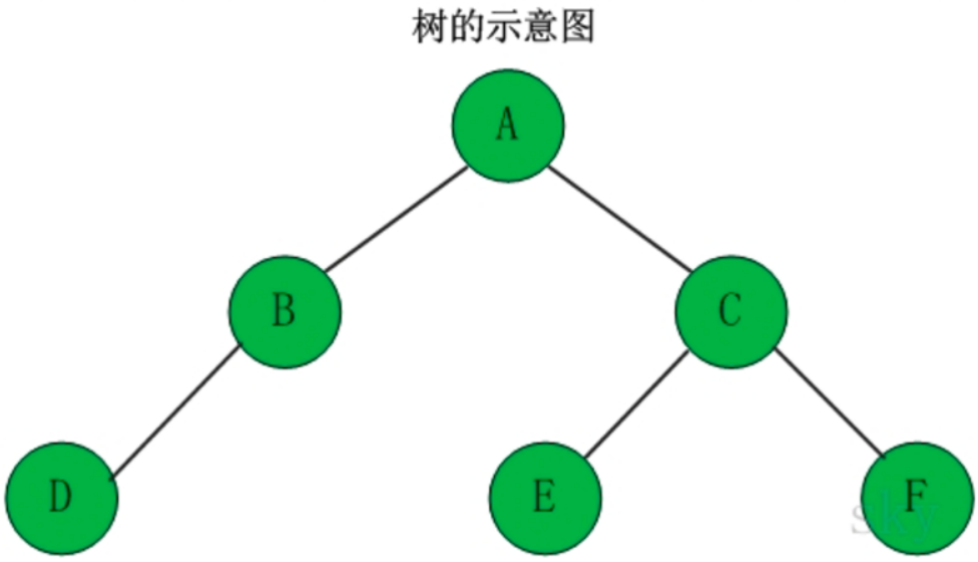
```
void CharSetHuffmanDecoding(HuffmanTree T, char* cd, int n)
  {
       int p=2*n-2;      //从根结点开始
       int i=0;
       //当要解码的字符串没有结束时
       while(cd[i]!='/0')
       {
            //当还没有到达哈夫曼树的叶子并且要解码的字符串没有结束时
           while((T[p].lchild!=0 && T[p].rchild != 0) && cd[i] != '\0')
            {
                 if(cd[i] == '0')
                 {
                    //如果是0，则叶子在左子树
                    p=T[p].lchild;
                 }
                 else
                 {
                    //如果是1，则叶子在左子树
                    p=T[p].rchild;
                 }
```

```
                i++;
        }
        //如果到达哈夫曼树的叶子时
         if(T[p].lchild == 0 && T[p].rchild == 0)
         {
             printf("%c", T[p].ch);
             p = 2*n-1;
          }
         else       //如果编号为p的结点不是叶子，那么编码有错
          {
               printf("\n解码出错! \n");
              return;
           }
    }
     printf("\n");
}
```

总结

树的定义

树是一种数据结构，它是由n（n>=1）个有限结点组成一个具有层次关系的集合。



树具有的特点有：

（1）每个结点有零个或多个子结点

（2）没有父节点的结点称为根节点

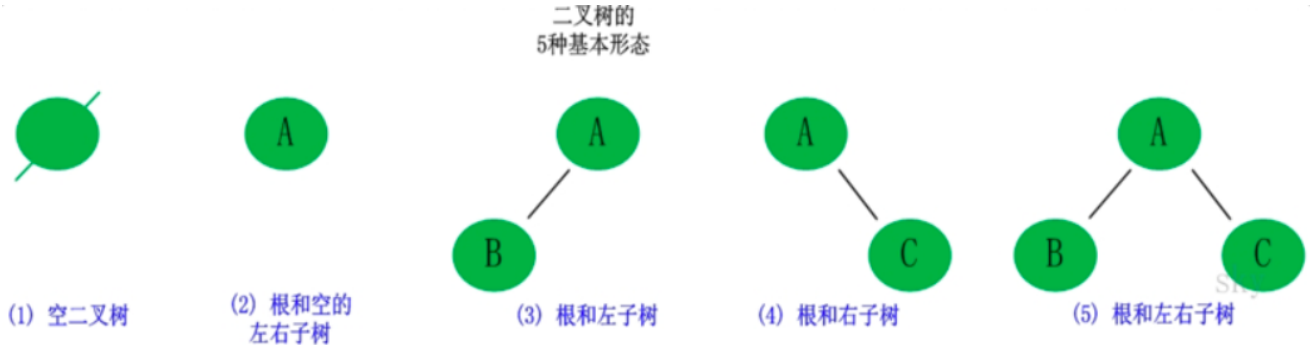
（3）每一个非根结点有且只有一个父节点

（4）除了根结点外，每个子结点可以分为多个不相交的子树。

二叉树

二叉树的定义

二叉树是每个结点最多有两个子树的树结构。它有五种基本形态：二叉树可以是空集；根可以有空的左子树或右子树；或者左、右子树皆为空。



二叉树的性质

性质1：二叉树第i层上的结点数目最多为2i-1(i>=1)

性质2：深度为k的二叉树至多有2k-1个结点（k>=1）

性质3：包含n个结点的二叉树的高度至少为(log2n)+1

性质4：在任意一棵二叉树中，若终端结点的个数为n0，度为2的结点数为n2，则n0=n2+1

二叉树的五种基本形态

- 空树
- 只有一个根结点
- 根结点只有左子树
- 根结点只有右子树
- 根结点既有左子树又有右子树

拥有特殊形态的二叉树

- 斜树:每个结点只有左结点或者每个结点只有右结点
- 满二叉树:树种每一层都含有最多的结点，对于编号ii的结点，期双亲结点为[i/2][i/2]
- 完全二叉树：每一个结点都与高度为h的满二叉树编号1−n1−n相同；如果i≤n/2i≤n/2下，则结点ii为分支结点，否则为叶子结点
- 二叉排序树：左子树均小于根结点，右子树均大于根结点
- 平衡二叉树：左右子树的深度之差不超过1

二叉树的性质

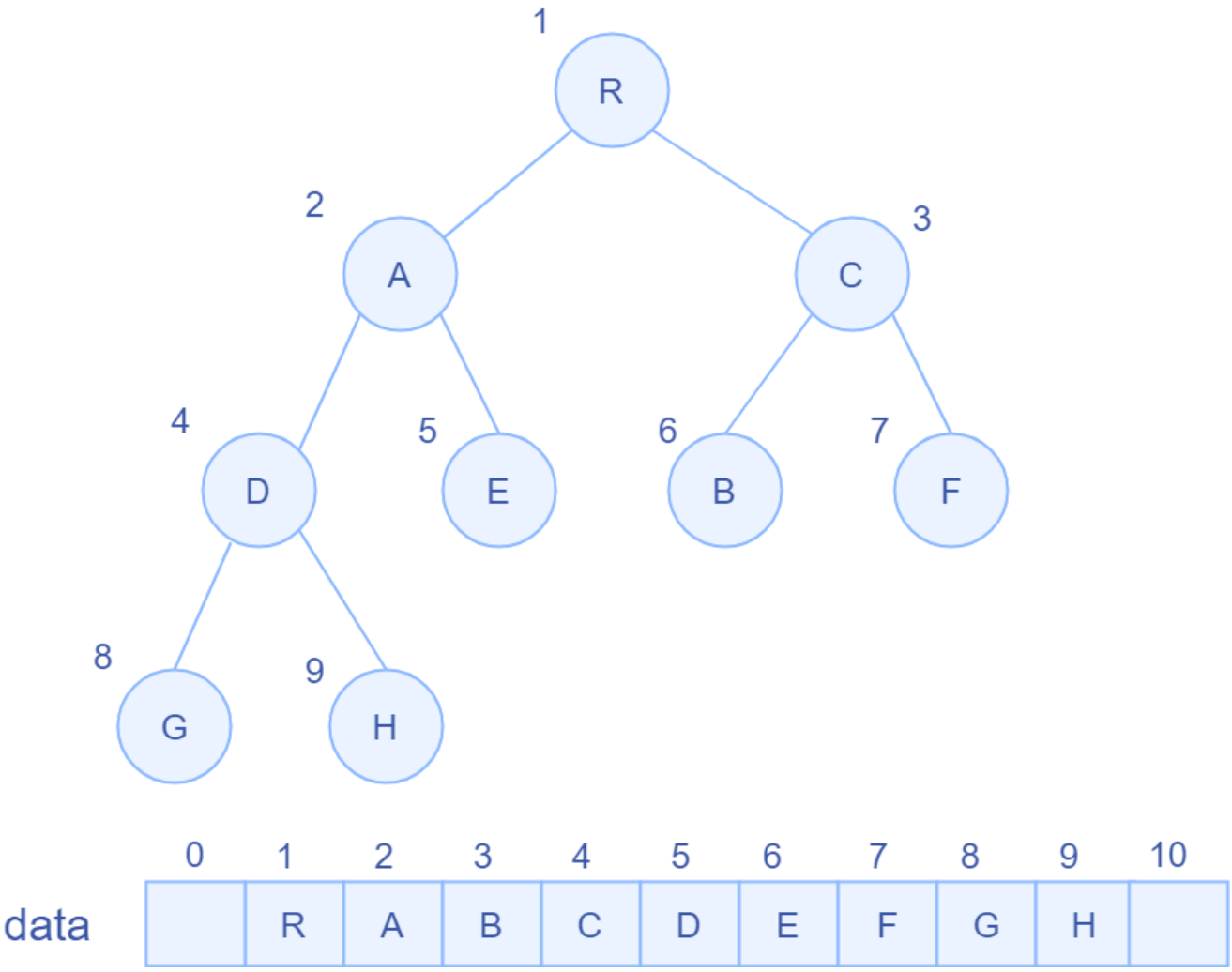
1. 非空二叉树上的叶子结点数等于度为2的结点数加一，即 n0=n2+1n0=n2+1
2. 非空二叉树上第kk层上至多有2k−12k−1个结点(k≥1)(k≥1)
3. 高度为hh的二叉树至多有2k−12k−1个结点(h≥1)(h≥1)
4. 具有nn个(n>0)(n>0)结点的完全二叉树的高度为⌈log2n+1⌉⌈log2n+1⌉或⌊log2n⌋+1⌊log2n⌋+1

二叉树的存储结构

顺序存储结构

二叉树的顺序存储结构就是用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素。

如图所示:

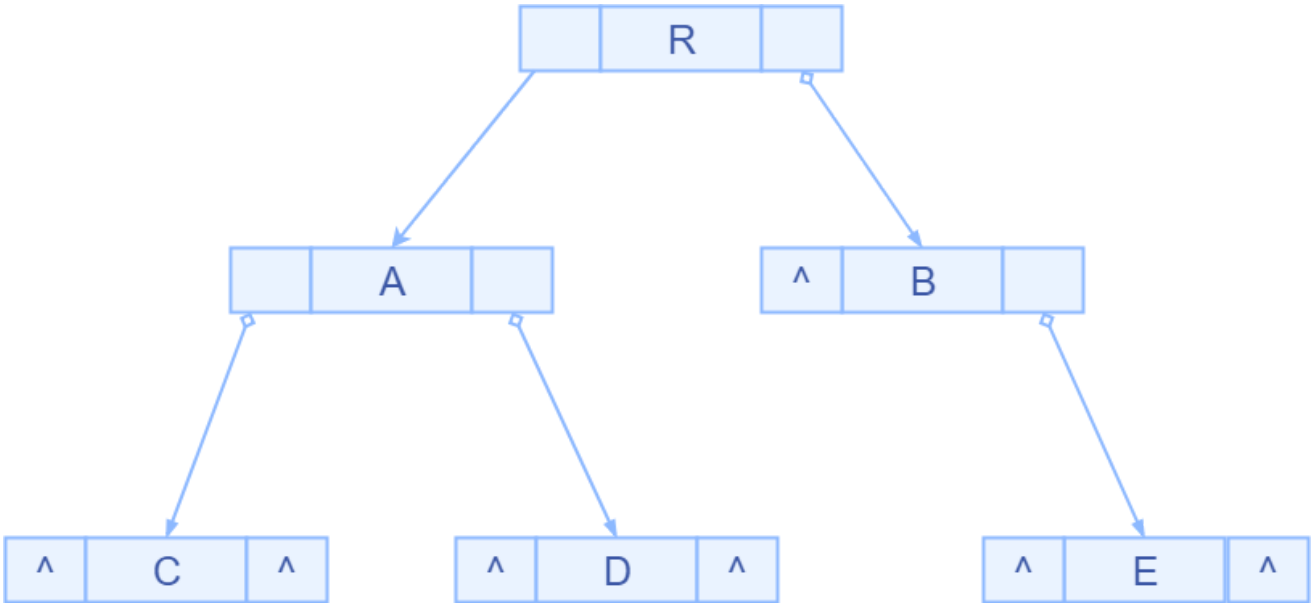


优点：适合完全二叉树和满二叉树，序号可以反映出结点之间的逻辑关系，可以节省空间
缺点：适合一般二叉树，只能添加一些空结点，空间利用率低

链式存储结构

二叉树每个结点最多两个孩子，所以设计二叉树的结点结构时考虑两个指针指向该结点的两个孩子。

如图所示:



代码如下:

```
typedef char ElemType;
typedef struct BiTNode{
    Elemtype data;
    struct BiTNode *lchild,*rchild;
}
```

二叉树的遍历

先序遍历(NLR)

过程:

1. 访问根结点
2. 先序遍历左子树
3. 先序遍历右子树

代码如下:

递归代码如下:

```
void PreOrder(BiTree T)
{
//先序遍历算法
if(T!=NULL){
    vist(T);  //访问根结点，如:printf("%c",T->data);
    PreOrder(T->lchild);  //递归遍历左子树
    PreOrder(T->rchild);  ////递归遍历右子树
    }
}
```

非递归代码如下

```
void Preorder2(BiTree T){
//先序遍历非递归算法
InitStack(S);  //需要借助一个递归栈
BiTree p=T;  //p是遍历指针
    while(p||!IsEmpty(S)){  //栈不空或p不空时循环
        if(p){  //一路向左
```

```
            visit(p);  //访问当前结点
            Push(S,p);  //入栈
            p=p->lchild;  //左孩子不空，一直向左走
            }
        else{  //出栈，并转向出栈结点的右子树，可改成if(!IsEmpty(S))
            Pop(S,p);  //栈顶元素出栈
            p=p->rchild;  //向右子树走，p赋值为当前结点的右孩子
            } // 返回while循环继续进入if-else语句
    }
}
```

中序遍历(LNRLNR)

1. 中序遍历左子树
2. 访问根结点
3. 中序遍历右子树

递归代码如下:

```
void InOrder(BiTree T)
{
//先序遍历算法
if(T!=NULL){
    InOrder(T->lchild);  //递归遍历左子树
    vist(T);  //访问根结点，如:printf("%c",T->data);
    InOrder(T->rchild);  ////递归遍历右子树
    }
}
```

非递归代码如下:

```
void Inorder2(BiTree T){
//中序遍历非递归算法
    InitStack(S);//需要借助一个递归栈
    BiTree p=T;
    while(p||!IsEmpty(S)){  //栈不空或者P不空时循环
        if(p){
            Push(S,p);
            p=p->lchild;
            }
        else{
            Pop(S,p);
            visit(p);
            p=p->rchild;
            }
        }
    }
```

后序遍历(LRNLRN)

1. 后序遍历左子树
2. 后序遍历右子树
3. 访问根结点

递归代码如下:

```
void PostOrder(BiTree T)
{
//先序遍历算法
if(T!=NULL){
    PostOrder(T->lchild);  //递归遍历左子树
    PostOrder(T->rchild);  ////递归遍历右子树
    vist(T);  //访问根结点，如:printf("%c",T->data);
    }
}
```

非递归代码如下:

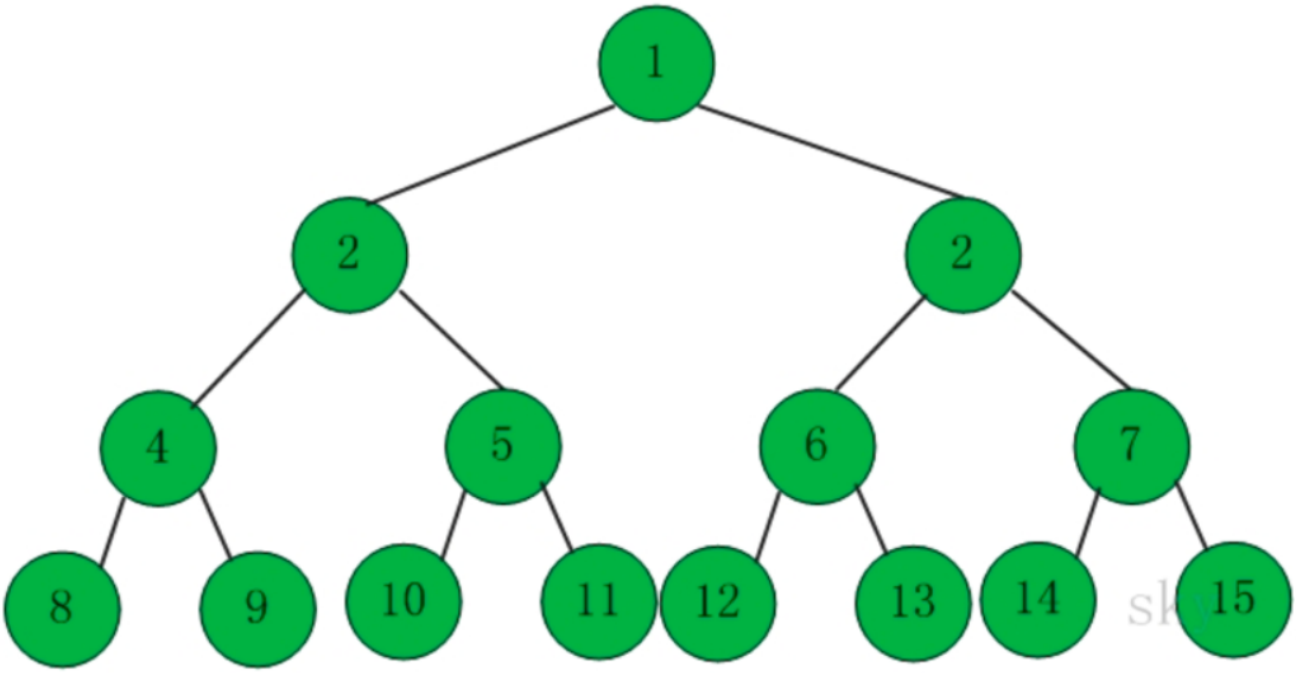
```
void PostOrder(BiTree T)){
    InitStack(S);
    BiTree p=T; //工作指针
    r=NULL;//指向最近访问过的结点，辅助指针
    while(p||!IsEmpty(S)){
        if(p){
            //1、从根结点到最左下角的左子树都入栈
            Push(S,p);
            p=p->lchild;
            }
        else{  //返回栈顶的两种情况
            GetTop(S,P);//弹出栈顶元素
            if(p->rchild&&p->rchild!=r){
                //1、右子树存在且未访问过，
                p=p->rchild;//转右
                push(S,p); //压入栈
                p=p->lchild;//走到最左
                    }
            else{
                //2、右子树已经访问或空，接下来出栈访问结点
                pop(S,p);  //将结点弹出
                visit(p->data);   //访问该结点
                  r=p;  //指针访问过的右子树根结点
                  p=NULL;//访问完之后就重置P，每次从栈中弹出一个，防止进入第一个if
                }
        }
    }
}
```

难点:要保证左孩子和右孩子都已被访问并且左孩子在右孩子前访问才能访问根结点

满二叉树

定义：高度为h，并且由2h-1个结点组成的二叉树，称为满二叉树

满二叉树的示意图



完全二叉树

定义：一棵二叉树中，只有最下面两层结点的度可以小于2，并且最下层的叶结点集中在靠左的若干位置上，这样的二叉树称为完全二叉树。

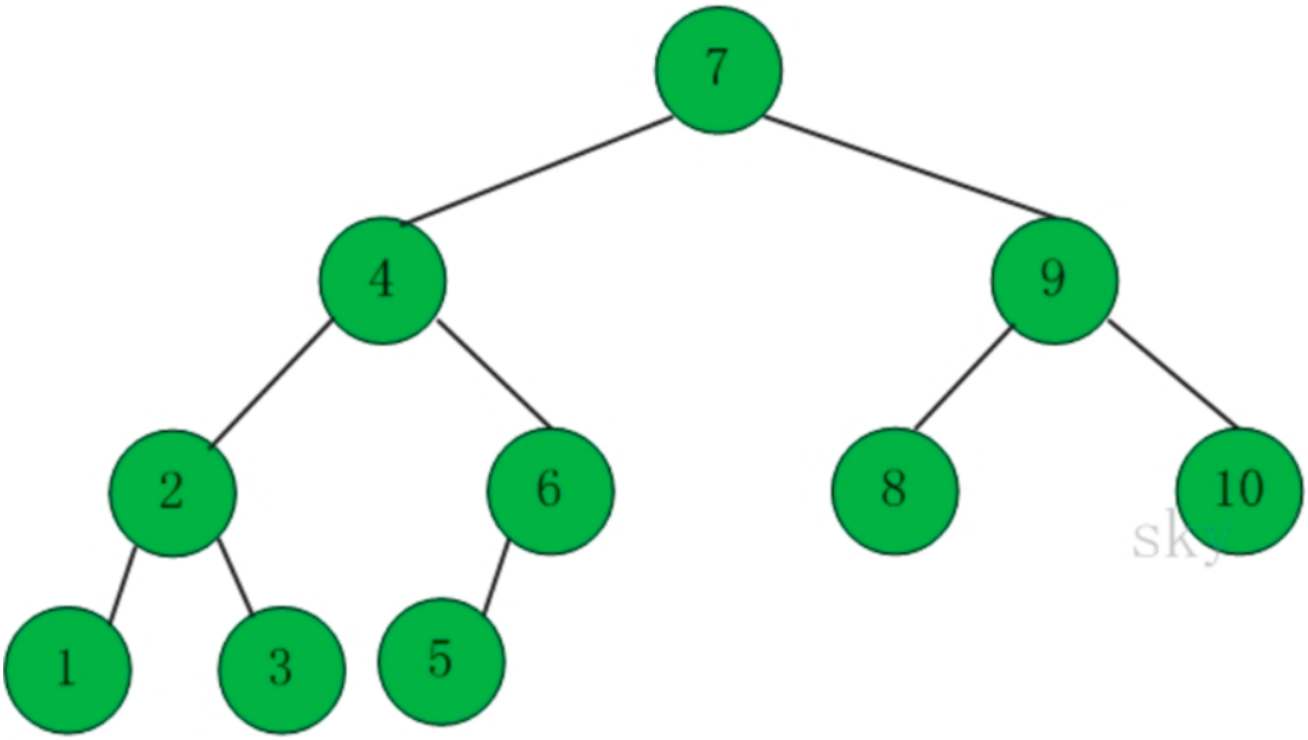
特点：叶子结点只能出现在最下层和次下层，且最下层的叶子结点集中在树的左部。显然，一棵满二叉树必定是一棵完全二叉树，而完全二叉树未必是满二叉树。

总结规律：如果一棵完全二叉树的结点总数为n，那么叶子结点等于n/2（当n为偶数时）或者(n+1)/2（当n为奇数时）

二叉查找树

定义：二叉查找树又被称为二叉搜索树。设x为二叉查找树中的一个结点，x结点包含关键字key，结点x的key值计为key[x]。如果y是x的左子树中的一个结点，则key[y]<=key[x]；如果y是x的右子树的一个结点，则key[y]>=key[x]

二叉查找树的示意图



在二叉查找树种：

（1）若任意结点的左子树不空，则左子树上所有结点的值均小于它的根结点的值。

（2）任意结点的右子树不空，则右子树上所有结点的值均大于它的根结点的值。

（3）任意结点的左、右子树也分别为二叉查找树。

（4）没有键值相等的结点。

哈夫曼编码

1.哈夫曼树又称最优二叉树，是一类带权路径长度最短的树。

对于最优二叉树，权值越大的结点越接近树的根结点，权值越小的结点越远离树的根结点。

最优二叉树的构造算法步骤：

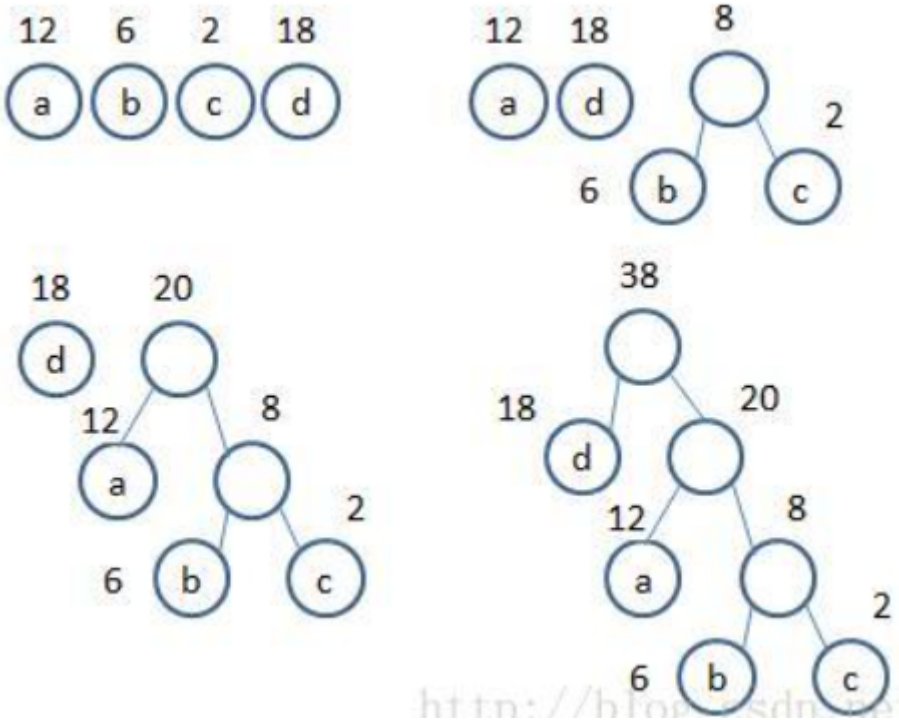
(1)根据给定的n个权值w1,w2,...,wn构成n棵二叉树森林F={T1,T2,...,Tn}，其中每一棵二叉树Ti中都只有一个权为wi的根结点，其左、右子树为空。

(2)在森林F中选出两棵根结点权值最小的树作为一棵新二叉树的左、右子树，新二叉树的根结点的权值为其左、右子树根结点的权值之和。

(3)从F中删除这两棵二叉树，同时把新二叉树加入到F中。

(4)重复步骤(2)、(3),直到F中只含有一棵树为止，此树便为最优二叉树。

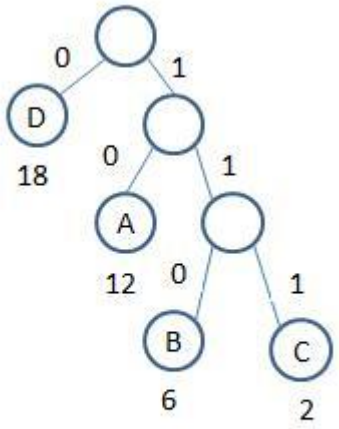
哈夫曼树的构造过程示意图如下：



哈夫曼编码是一种变长编码。其定义如下：

对于给定的字符集D={d1,d2,...,dn}及其频率分布F={w1,w2,...,wn}，用d1,d2,...,dn作为叶结点，w1,w2,...,wn作为结点的权，利用哈夫曼算法构造一棵最优二叉树，将树中每个分支结点的左分支标上"0"；右分支标上"1"，把从根到每个叶子的路径符号("0"或"1")连接起来，作为该叶子的编码。

哈夫曼编码是在哈夫曼树的基础上求出来的，其基本思想是：从叶子结点di(0<=i<n)出发，向上回溯至根结点，依次求出每个字符的编码。



A：10
B：110
C：111
D：0

(a)哈夫曼树　　(b)哈夫曼编码