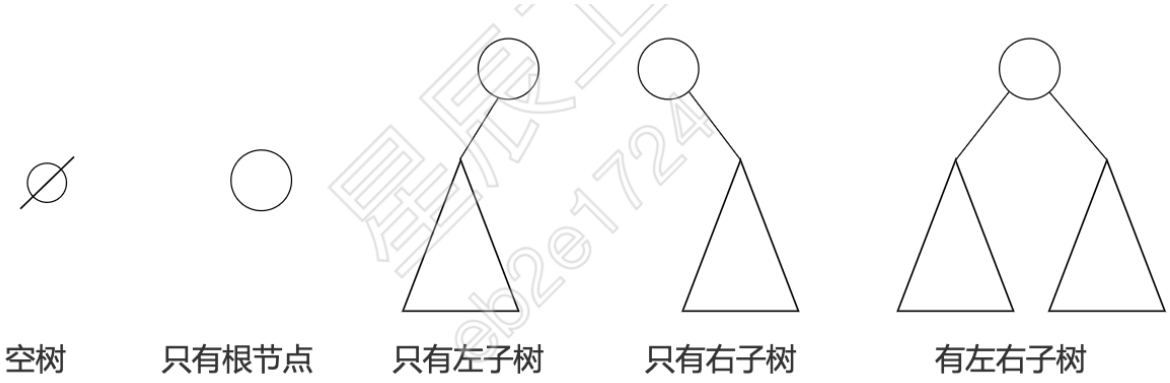
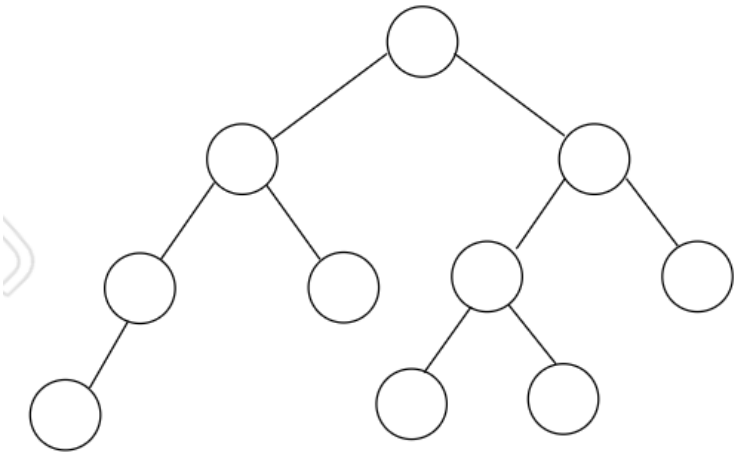
第二十二章 二叉树

1.二叉树

- 二叉树的定义
  二叉树是一种最简单且最重要的树，其特点是每个节点至多只有两棵子树 （即二叉树中不存在度大于2的节点），并且二叉树的子树有左右之分，其次序不能任意颠倒
  用递归的形式定义，二树是n（n）个节点的有限集合： 或者为空二叉树，即n=0
  或者由一个根节点和两个互不相交的被称为根的左子树和右树组成。左子树和右子树又分别是一棵二叉树
- 二叉树的基本形态
  二叉树的五种基本形态如下：



2.二叉树的性质和满二叉树的性质



1. 任何一棵树，节点的数量为n，则边的数量为n-1
2. 非空二叉树的叶子节点数等于度为2的节点数加1，也就是说 $n_0 = n_2 + 1$
   证明： 因为二叉树中所有节点的度数均不大于2，所以节点总数n应等于0度节点数n0，1度节点数n1和2度节点数n2之和：也就是

$$n = n_0 + n_1 + n_2$$

另外，1度节点有一个孩子，2度节点有两个孩子，故二叉树中孩子节点总数是

$$n_1 + 2 \times n_2$$

树中只有根节点不是任何节点的孩子，故二叉树中的节点总数又可表示为

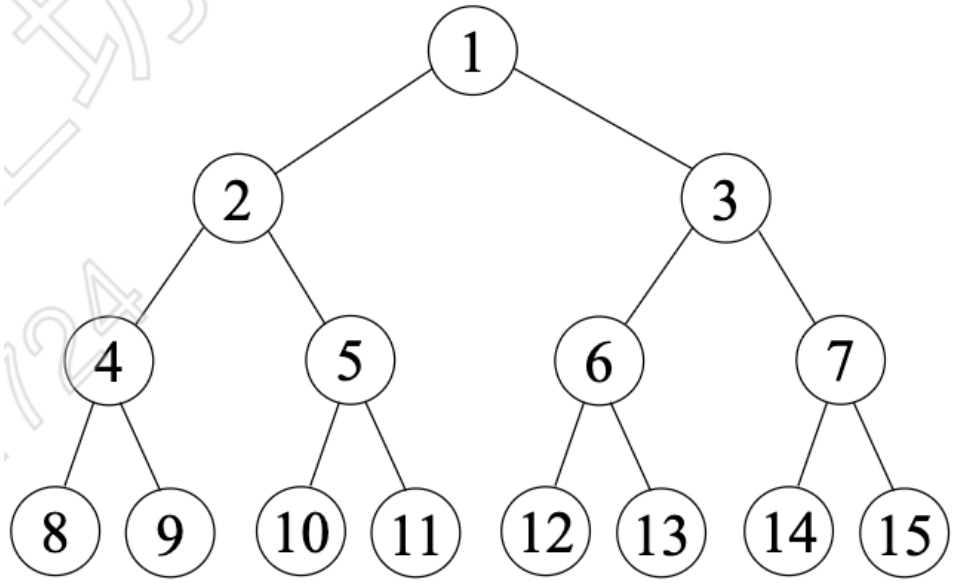
$$n = n_1 + 2 \times n_2 + 1$$

由式（1）和式（2）合并得到

$$n_0 = n_2 + 1$$

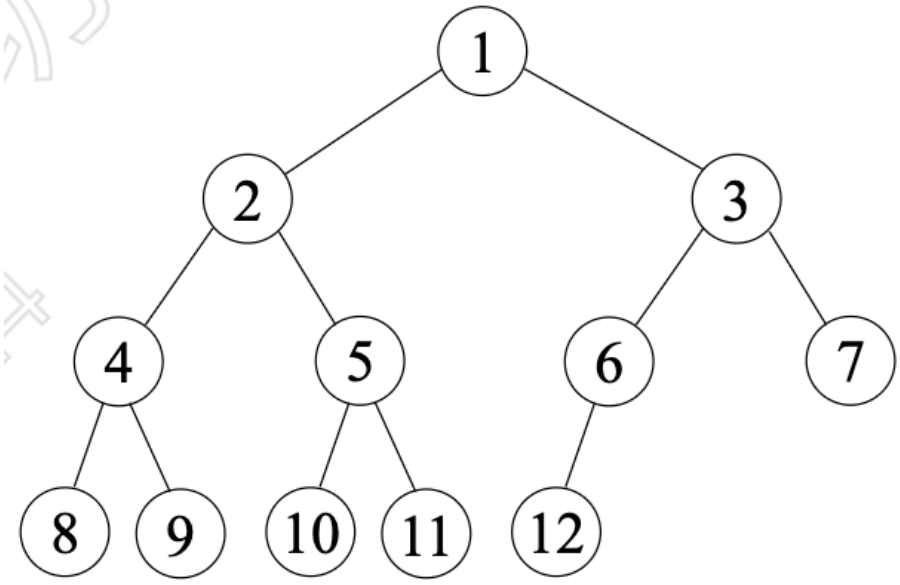
3. 非空二叉树的第k层上之多有 2^{k-1}个节点 （k>=1） 证明： 用归纳法证明：当k=1时，$2^{k-1} = 1$显然成立；
   现在假设第k-1层时命题成立，即第k-1层上最多有2^{k-2}个节点 由于二叉树的每个节点的度最多为2，故第K层上的最大节点数为第k-1层的2倍，即 $2 \times 2^{k-2} = 2^{k-1}$ 故命题成立。
4. 深度为h的二叉树至多有$2^h - 1$ 个节点 （h>=1） 证明：当每一层拥有最大节点数的时候，树中的节点最多，再根据性质三来说深度为h的二叉树的节点数至多为每一层相加。

- 满二叉树的定义：一棵有h层，且含有$2^h - 1$个节点的二叉树称为满二叉树，也叫做完美二叉树，即树中的每层都含有最多 的节点
  满二叉树的叶子节点都集中在二叉树的最下一层，并且除叶子节点之外的每个节点度数均为2
  可以对满二树按层序编号：
  编号从根节点（根节点编号为1）起，自上而下，自左向右。这样，每个节点对应一个编号。



1. 满二叉树的性质1：满二叉树中第i层的节点数2^{i-1}个
2. 满二叉树的性质2:层数为k的满二树总共的节点数量一定有$2^k - 1$个,叶节点数为2^{k-1}$
3. 满二叉树的性质3:满二叉树中不存在度为1的节点，每一个分支节点中都两棵深度相同的子树，且叶子节点都在最底层
4. 满二叉树的性质4:具有n个节点的满二叉树的层数为$log_2(n+1)$

- 完全二叉树定义： 高度为h、有n个节点的二叉树，当且仅当其每个节点都与高度为h的满二叉树中编号为1~n的节点 对应时，称为完全二叉树
  完全二叉树中除去最后一层节点后就是满二树，且最下层的叶子节点一定集中于左部连续位置



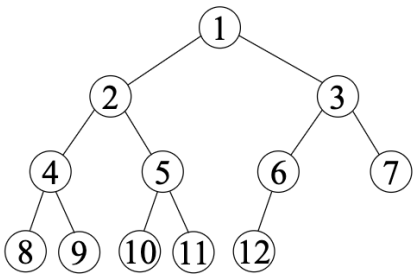
1. 完全二叉树的性质1:假定根节点从1开始编号，满二叉树的节点若有父节点，则其父节点的编号为i÷2；
   若有左子节点， 则左子节点的编号为2×i；
   若有右子节点，则右子节点的编号为2×i+1
2. 完全二叉树的性质2:如果2×i>n（总节点的个数），则节点i肯定没有左子节点 （为叶节点）；否则其左孩子的编号是2×i
3. 完全二叉树的性质3:如果2×i+1>n，则节点i肯定没有右子节点；否则右子节点的编号是2×i+1
4. 完全二叉树的性质4:n个节点的完全二叉树的深度为log_2n。

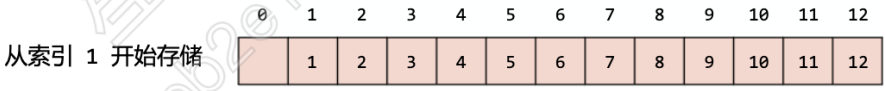
3. 二叉树的存储结构

- 顺序存储结构

  使用一组地址连续的存储单元自上而下，从左往右依次存储二叉树的节点，这样，节点 i 就存储在下标为 i - 1 的元素中。

  完全二叉树和满二叉树采用顺序存储比较合适，一般的二叉树需浪费空间存储不存在的节点。





- 链式存储结构（1） 用链表来表示一棵二叉树，即用链来指示元素的逻辑关系。通常的方法是链表中每个节点由三个域组成，数据域和左右指针域，左右指针分别用来给出该节点左孩子和右孩子所在的链节点的存储地址。 子节点表示法数组版：

```
const int M = 100;
struct Node{
    int data;  //数据域
    int left;  //指针域，指向做左子节点
    int right; //指针域，指向做右子节点
} node[M];
int root;   //根节点
```

子节点表示法指针版：

```
const int M = 100;
struct Node{
    int data;  //数据域
    Node* left;  //指针域，指向左子节点
    Node* right; //指针域，指向右子节点
};
Node* root;//根节点指针
```

- 链式存储结构（2） 用链表来表示一棵二树，即用链来指示元素的逻辑关系。通常的方法是链表中每个节点由四个域组成数据域、父节点指针域和左右指针域，左右指针分别用来存储该节点左孩子和孩子所在的链节点的存储地址。 父子节点表示法数组版：

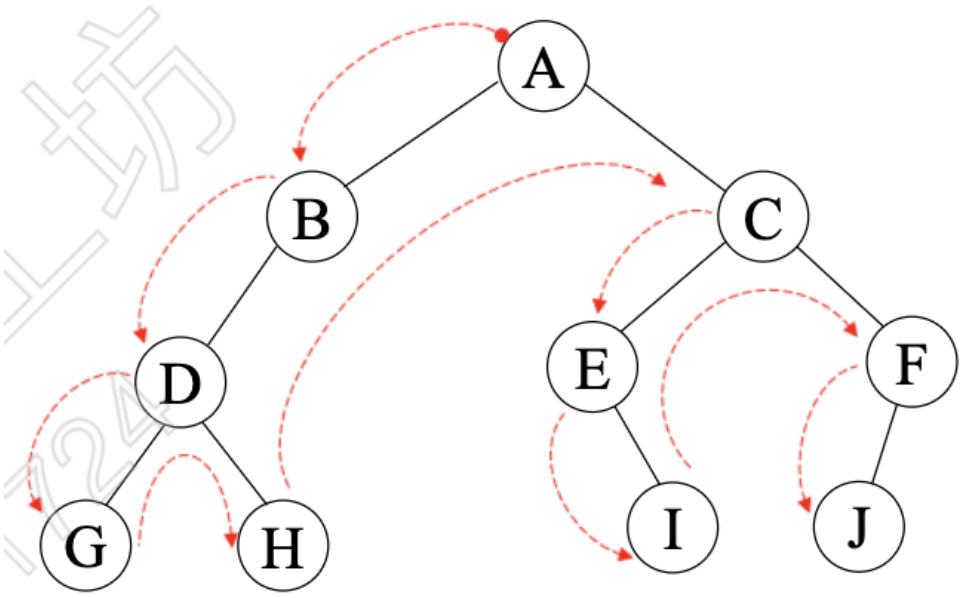
```
const int M = 100;
struct Node{
    int data;  //数据域
    int father; //指针域，指向父节点
    int left;  //指针域，指向左子节点
    int right; //指针域，指向右子节点
} node[M];
int root;   //根节点
```

父子节点表示法指针版：

```
const int M = 100;
struct Node{
    int data;  //数据域
    Node* father; //指针域，指向父节点
    Node* left;  //指针域，指向左子节点
    Node* right; //指针域，指向右子节点
};
Node* root;//根节点指针
```

4. 二叉树的遍历

- 先序遍历
- 中序遍历
- 后序遍历
- 层次遍历
- 先序遍历

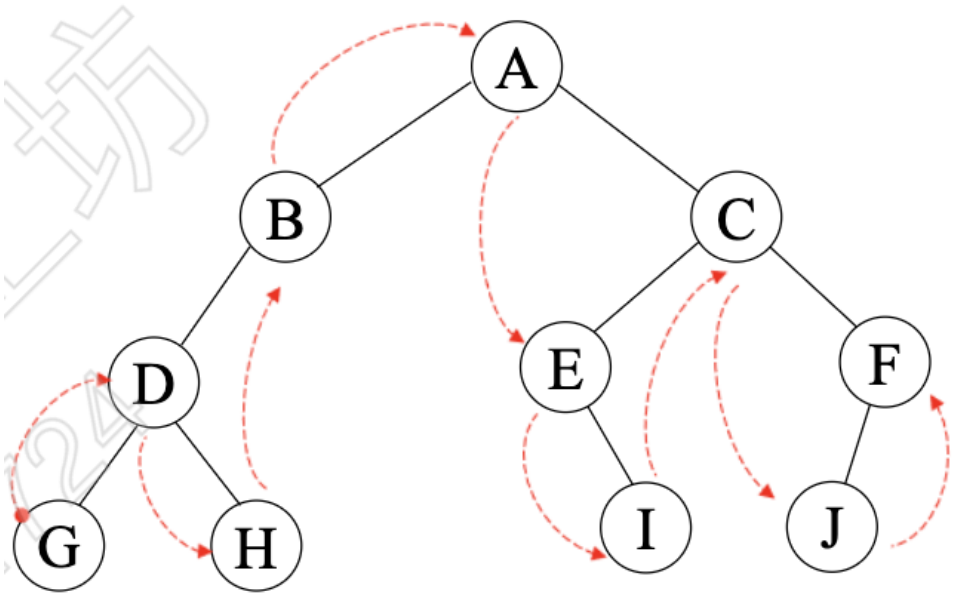


先序遍历的操作

过程如下：
若二叉树为空，则什么也不做，否则：
1）访问根节点
2）先序遍历左子树
3）先序遍历右子树
先序遍历的结果如下：
AABDGHCEIFJ 先序遍历代码：

```
void PreOrder(Node* T)
{
    if(T != NULL)
    {
        visit(T);   //访问根节点
        PreOrder(T->lchild);  //递归遍历左子树
        PreOrder(T->rchild);  //递归遍历右子树
    }
}
```

- 中序遍历



中序遍历的操作过程如下：
若二叉树为空，则什么也不做，否则：
1）中序遍历左子树
2）访问根节点
3）中序遍历右子树
中序遍历的结果：
GDHBAEICJF
中序遍历代码：

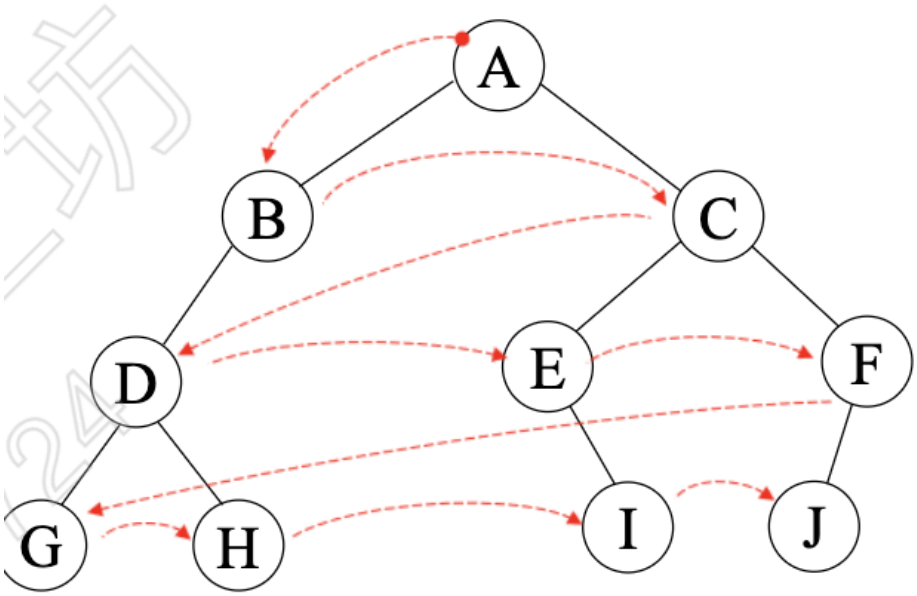
```
void InOrder(Node* T)
{
    if(T != NULL)
    {
        InOrder(T->lchild);  //递归遍历左子树
        visit(T);   //访问根节点
        InOrder(T->rchild);  //递归遍历右子树
    }
}
```

- 后序遍历
  后序遍历的操作过程如下：
  若二叉树为空，则什么也不做，否则：
  1）后序遍历左子树
  2）后序遍历右子树
  3）访问根节点
  后序遍历的结果：
  GHDBIEJFCA

后序遍历的算法如下：

```
void PostOrder(Node* T)
{
    if(T != NULL)
    {
        PostOrder(T->lchild);  //递归遍历左子树
        PostOrder(T->rchild);  //递归遍历右子树
        visit(T);   //访问根节点
    }
}
```

- 层次遍历



层次遍历的操作过程如下：
若二叉树为空，则什么也不做，否则：
1）访问根节点
2）按层次从上往下，从左往右依次遍历
层次遍历的结果（层次序列）：
ABCDEFGHIJ
层次遍历常使用队列来实现，算法如下：

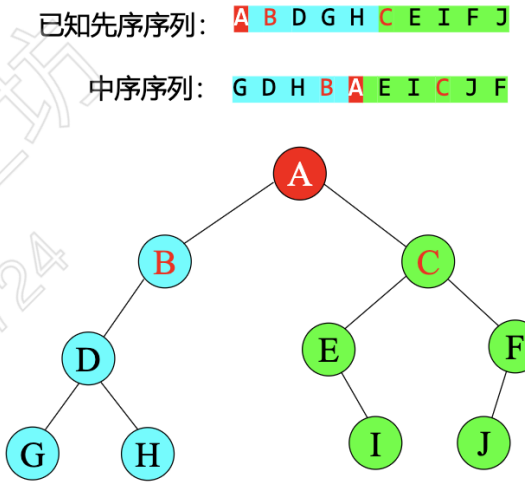
```
void LevelOrder(Node T)
{
    InitQueue(Q);   //初始化队列
    Node* p;
    EnQueue(Q,T);  //根节点入队
    while(!IsEmpty(Q))//队头不空则循环
    {
        DeQueue(Q,p);  //队头节点出队
        visit(p);   //访问出队节点
        if(p->lchild != NULL)
        {
            EnQueue(Q, p->lchild);  //左子节点不为空，则左子节点入队
        }
        if(p->rchild != NULL)
        {
            EnQueue(Q, p->rchild);//右子节点不为空，则右子节点入队
        }
    }
}
```

5.确定二叉树的结构

- 由先序序列和中序序列确定二叉树的结构
- 由后序序列和中序序列确定二叉树的结构
- 由先序序列和后序序列确定二叉树的结构
- 由层次序列和中序序列确定二叉树的结构

由先序序列和中序序列确定二叉树的结构

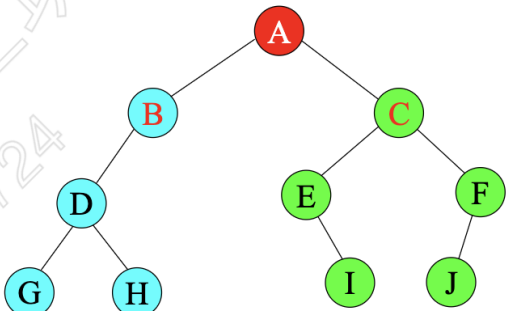
在先序序列中，第一个节点一定是二叉树的根节点；而在中序序列中，根节点必然将中序序列分割成两个子序列，前一个子序列是根节点的左子树的中序序列，后一个子序列是根节点的右子树的中序序列。
根据这两个子序列，在先序序列中找到对应的左子序列和右子序列。在先序序列中，左子序列的第一个节点是根节点的左孩子，右子序列的第一个节点是根节点的右孩子。
以此类推，进行递归，即可确定二叉树的结构。



由后序序列和中序序列确定二叉树的结构

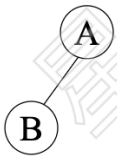
在后序序列中，最后一个节点一定是二叉树的根节点；
而在中序序列中，根节点必然将中序序列分割成两个子序列，前一个子序列是根节点的左子树的中序序列，后一个子序列是根节点的右子树的中序序列。
根据这两个子序列，在后序序列中找到对应的左子序列和右子序列。在后序序列中，左子序列的最后一个节点是根节点的左孩子，右子序列的最后一个节点是根节点的右孩子。
以此类推，进行递归，即可确定二叉树的结构。

已知后序序列：G H D B I E J F C A
中序序列：G D H B A E I C J F

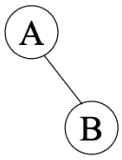


由先序序列和后序序列确定二叉树的结构

在先序遍历序列中，第一个节点一定是二叉树的根节点；
在后序遍历序列中，最后一个节点一定是二叉树的根节点；
根据这两个子序列，无法判定根节点的左右子树。
以下两棵树的对应的先序序列和后序序列完全相同，所以，仅由先序序列和后序序列无法确定二叉树的结构。



先序序列：A B
后序序列：B A

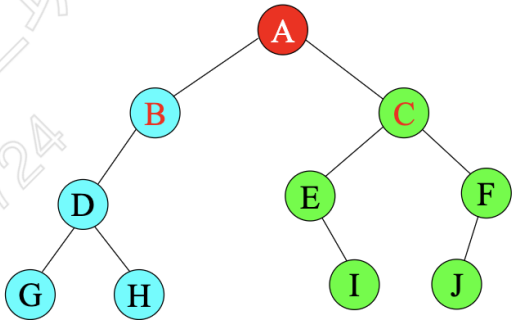


先序序列：A B
后序序列：B A

由层次序列和中序序列确定二叉树的结构

在层次序列中，第一个节点一定是二叉树的根节点；而在中序序列中，根节点必然将中序序列分割成两个子序列，前一个子序列是根节点的左子树的中序序列，后一个子序列是根节点的右子树的中序序列。
根据这两个子序列，在层次序列中找到对应的左子序列和右子序列。在层次序列中，左子序列的第一个节点是根节点的左孩子，右子序列的第一个节点是根节点的右孩子。
以此类推，进行递归，即可确定二叉树的结构。

已知层次序列：A B C D E F G H I J
中序序列：G D H B A E I C J F

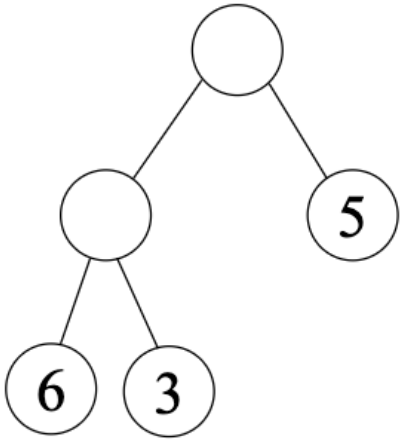


6. 哈夫曼树

树中节点常常被赋予一个表示某种意义的数值，称为该节点的权从根到任意节点的路径长度（经过的边数）与该节点权值的乘积称为该节点的带权路径长度。
树中所有叶节点的带权路径长度之和称为该树的带权路径长度，记

$$WPL=\sum_{i=1}^{n}w_i\times l_i=w_1\times l_1+\ldots+w_n\times \backslash l_n$$

w_i 是第i个叶节点所带的权值，l_i 是该叶节点到根节点的路径长度



$$WPL=6\times2+3\times2+5\times1=23$$

- 哈夫曼树

  在含有 n 个带权叶节点的二叉树中，其中带权路径长度（WPL）最小的二叉树称为哈夫曼树，也称最优二叉树。

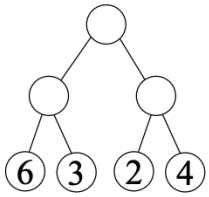


图1

$WPL=6\times2+3\times2+2\times2+4\times2=30$

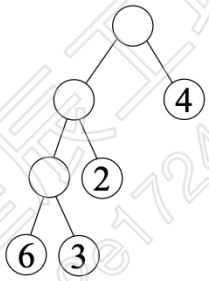


图2

$WPL=6\times3+3\times3+2\times2+4\times1=35$

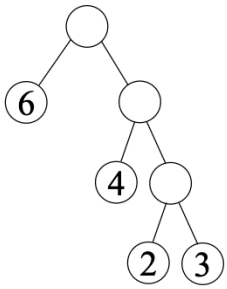


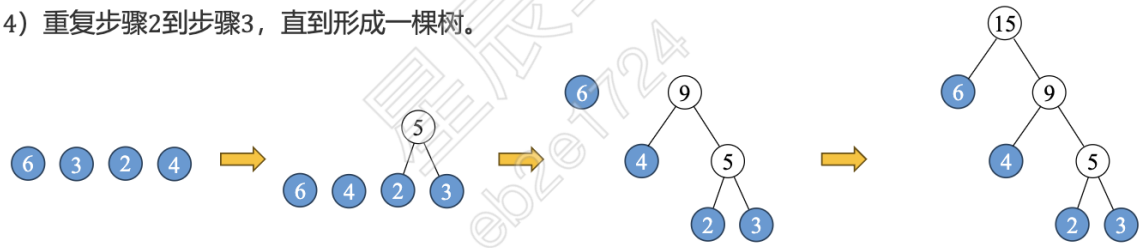
图3

$WPL=6\times1+4\times2+2\times3+3\times3=29$

其中，图 3 的WPL最小，这是一棵哈夫曼树。

- 哈夫曼的构造

  1）先把有权值的叶子节点按照从大到小的顺序进行排序。
  2）取权值最小的两个节点作为一个新节点的两个子节点（左子节点较小），新节点权值为两个子节点权值之和。
  3）用上一步构造的新节点替换它的两个子节点，再继续排序，保持从大到小排列。
  4）重复步骤2到步骤3，直到形成一棵树。



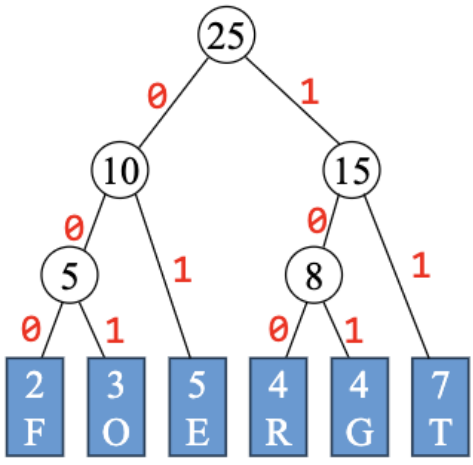
7. 哈夫曼编码

又译为霍夫曼编码、 赫夫曼编码，是一种用于无损数据压缩的编码 （权编码） 算法。由美国计算机科学家大卫·哈夫曼 （David Albert Huffman） 在1952年发明。
基于二树构建编码压缩结构，是一种经典的数据压缩算法。算法根据文本字符出现的频率，重新对字符进行编码。出现概率高的字母使用较短的编码，反之出现概率低的则使用较长的编码这便使编码之后的字符串的平均长度更低，从而达到无损压缩数据的目的。

- 哈夫曼编码
  假设我们要给一个英文单字FORGET进行哈夫曼编码 而每个英文字母出现的频率如下

字符	F	O	R	G	E	T
频率	2	3	4	4	5	7

1. 先建立哈夫曼树
   i. 将每个英文字母依照出现频率由小排到大，最小在左
   ii. 每个字母都代表-一个叶节点，比较F、O、R、G、E、T六个字母中每个字母的出现频率，将最小的两个字母频率相加合成一个新的节点，如下图所示，发现F与O的频率2、3最小，故相加2+3=5。
   iii. 比较5、R、G、E、T，发现R与G的频率4最小，故相加4+4=8
   iv. 比较5、8、E、T，发现5与E的频率5最小,故相加5+5=10
   v. 比较8、10、T，发现8与T的频率7最小，故相加8+7=15。
   vi. 最后 10、15，没有可以比较的对象，相加10+15=2.最后产生的树状图就是哈夫曼树
2. 编码
   i. 给哈夫曼树的所有左分支标记为“0”，右分支标记为“1”，如下图所示
   ii. 从树根至树叶依序记录所有字母的编码，如下图所示



字符	F	O	R	G	E	T
频率	2	3	4	4	5	7
编码	000	001	100	101	01	11

- 哈夫曼编码有以下优点：

1. 编码后的文本更节省空间
2. 任意一个编码都不会是其他编码的前缀

- 小结 效率高：哈夫曼编码是一种无损数据压缩算法，它能够有效地减少数据的大小
  适用性广：哈夫曼编码适用于各种不同类型的数据，包括文本、图像、音频和视频等

实现简单：哈夫曼编码的实现方法相对简单，不需要复杂的计算过程
压缩比高：哈夫曼编码的压缩比比其他数据压缩算法更高，能够更有效地减小数据的大小

8.二叉查找树

- 二叉查找树： 二叉查找树 （英语：BinarySearchTree），也称为二叉搜树、有序二叉树 （ordered binary tree）或排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质二叉树：

1. 若任意节点的左子树不空， 则左子树上所有节点的值均小于它的根节点的值；
2. 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值
3. 任意节点的左、右子树也分别为二叉查找树；
   二查找树相比于其他数据结构的优势在于查找、插入的时间复度较低。为O(logn)。
   二查找树是基础性数据结构，用于构建更为抽象的数据结构，集合、多重集、关联数组等

- 根据二叉排序树的定义，左子树节点值<根节点值<右子树节点值，所以对二查找树进行中序遍历，可以得到一个递增的有序序列。例如，下图所示叉查找树的中序遍历序列为1235678。!alt text
- 二叉查找树的操作-定义

```
constintM=100; //最多的节点数
struct Node{
    int data; //数据域
    int left; //左子节点指针
    int right; //右子节点指针
}node[M];
```

- 二叉查找树的操作-查找
- 二叉查找树的操作-插入节点
- 二叉查找树的操作-删除节点