树

定义：n个节点的有限集，在任意一棵非空树中:(1) 有且仅有一个特定的称为根的节点（2）当n>1时，其余节点可以分为m个互不相交的有限集，其中每个集合本身又是一棵树并且称为根的子树。

一.二叉树：

1.定义：每个结点至多有两棵子树的树，并且子树存在左右之分，次序不可颠倒

2.性质：

1）在二叉树的第i层上至多有2的i-1次方个结点

2）深度为k的二叉树至多有2的k次方-1个结点（当相等时为完全二叉树）

3）二叉树叶子结点数为n0，度为2的结点数为n1,则n0=n2+1

4)n个结点的完全二叉树，对任一结点i

- i=1，i是二叉树的根
- 2i>n，则结点i无左孩子
- 2i+1>n,则结点i无右孩子
- 完全二叉树叶子结点数n0=[n/2](n为奇数，n=n+1)

3.遍历二叉树

a)先序遍历：

若二叉树为空，则空操作

否则，先访问根节点；先序遍历左子树，先序遍历右子树

递归方式：

```
public void preOrderRecur(Node head){
        if(head==null){
            return;
        }
        System.out.println(head.value+"");
        preOrderRecur(head.left);
        preOrderRecur(head.right);
    }
```

非递归方式：

```
public void preOrderUnRecur(Node head){
        if(head!=null){
            Stack<Node> stack=new Stack<Node>();
            stack.add(head);
            while(!stack.isEmpty()){
                head=stack.pop();
                System.out.println(head.value);
                if(head.right!=null){
                    stack.add(head.right);
                }
                if(head.left!=null){
                    stack.add(head.left);
                }
            }
        }
        System.out.println();
    }
```

b)中序遍历

二叉树为空，则空；

否则，中序遍历左子树，访问根节点，中序遍历右子树

递归方式：

```
public void inOrderRecur(Node head){
        if(head==null){
            return;
        }
        else{
```

```
            inOrderRecur(head.left);
            System.out.println(head.value);
            inOrderRecur(head.right);
        }

    }
```

非递归方式：

```
public void inOrderUnRecur(Node head){
        if(head!=null){
            Stack<Node> stack=new Stack<Node>();
            while(!stack.isEmpty()||head!=null){
                if(head!=null){
                    stack.push(head);
                    head=head.left;
                }
                else{
                    head=stack.pop();
                    System.out.println(head.value+"");
                    head=head.right;
                }
            }
        }
        System.out.println();
    }
```

c)后序遍历：

二叉树为空，则空；

后序遍历左子树，遍历右子树，访问根节点

递归方式：

```
public void posOrderRecur(Node head){
        if(head==null){
            return;
        }
        posOrderRecur(head.left);
        posOrderRecur(head.right);
        System.out.println(head.value);
    }
```

非递归方式：

//使用两个栈

```
public void posOrder(Node head){
        if(head!=null){
            Stack<Node> s1=new Stack<Node>();
            Stack<Node> s2=new Stack<Node>();
            while(!s1.isEmpty()){
                Node cur=s1.pop();
                //相当于根节点先入栈
                s2.push(cur);
                //相当于左子树后入栈
                if(cur.left!=null)
                    s1.push(cur.left);
                //右子树先入栈
                if(cur.right!=null)
                    s1.push(cur.right);
```

```
            }
            while(s2.isEmpty()){
                System.out.println(s2.pop().value);
            }
        }
    }
//使用一个栈
public void posOrder2(Node head){
        if(head!=null){
            Stack<Node> s=new Stack<Node>();
            s.push(head);
            while(!s.isEmpty()){
                Node cur=s.peek();
                //head现在代表最近一次弹出并打印的节点
                //表示左孩子和右孩子都没有处理
                if(cur.left!=null&&cur.left!=head&&cur.right!=head)
                    s.push(cur.left);
                //表示右孩子没有处理
                else if(cur.right!=null&&cur.right!=head)
                    s.push(cur.right);
                else{
                    head=s.pop();
                    System.out.println(head.value);
                }
            }
        }
```

}

二.二叉查找树（排序树）：

性质：

- 若它的左子树不为空，则左子树上所有结点值均小于它它根结点的值
- 若它的右子树不为空，则右子树上所有结点的值均大于它根结点的值
- 它的左、右子树也分别为二叉排序树
- 插入和查找的时间复杂度都为O(logN)

三.平衡二叉树：

符合二叉排序树，并且满足任何节点两个子树高度最大差为1

查找、插入和删除在平均和最坏情况下时间复杂度都为O(logN)。插入和删除可能需要通过一次或多次树旋转来重新平衡这个树。

对二叉树的平衡调整过程，主要包含四种旋转操作：LL，LR，RR，RL 。

LL型：插入点位于失去平衡点的左孩子的左子树上； RR型：插入点位于失去平衡点的右孩子的右子树上； LR型：插入点位于失去平衡点的左孩子的右子树上； RR型：插入点位于失去平衡点的右孩子的左子树上。

四.B-树与B+树

B-树是一种平衡的多路查找树，一棵m阶B-树或为空树，或为满足下列特性的m叉树：

1）树中每个结点至多有m棵子树

2）若根结点不是叶子结点，则至少有两棵子树

3）除根结点之外的非终端结点至少有m/2棵子树

4）非终端结点包含下列信息

（n,A0,K1,A1,K2...KN,AN）

n为关键字个数，关键字依次递增，并且Ai所指子树关键字均小于ki

5）所有叶子结点都出现在同一层上且不带任何信息

B+树是B-树的一种变型树，差异在于：

1）有n棵子树的结点中有n个关键字

2）所有叶子结点包含全部关键字的信息及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小而顺序链接

3）所有非终端结点可以看成索引部分，结点中仅含有其子树中的最大或最小关键字

B-树与B+树的区别：

1).B-树的关键字和记录是放在一起的，叶子节点可以看作外部节点，不包含任何信息；B+树的非叶子节点中只有关键字和指向下一个节点的索引，记录只放在叶子节点中。
2).在B-树中，越靠近根节点的记录查找时间越快，只要找到关键字即可确定记录的存在；而B+树中每个记录的查找时间基本是一样的，都需要从根节点走到叶子节点，而且在叶子节点中还要再比较关键字。从这个角度看B-树的性能好像要比B+树好，而在实际应用中却是B+树的性能要好些。因为B+树的非叶子节点不存放实际的数据，这样每个节点可容纳的元素个数比B-树多，树高比B-树小，这样带来的好处是减少磁盘访问次数。尽管B+树找到一个记录所需的比较次数要比B-树多，但是一次磁盘访问的时间相当于成百上千次内存比较的时间，因此实际中B+树的性能可能还会好些，而且B+树的叶子节点使用指针连接在一起，方便顺序遍历(例如查看一个目录下的所有文件，一个表中的所有记录等)，这也是很多数据库和文件系统使用B+树的缘故。
3）B+树支持range-query非常方便，而B树不支持。这是数据库选用B+树的最主要原因

B+树查找：从最小关键字起顺序查找；或者从根结点开始进行随机查找。在查找时如果非终端结点等于给定值，并不终止，而是继续向下直到叶子结点

B+树插入：插入在叶子结点上进行，如果叶结点中的关键码个数超过m，就必须分裂成关键码数目大致相同的两个结点，并保证上层结点中有这两个结点的最大关键码（分为三种情况：leafpage和indexpage都没满，leafpage满，leafpage和indexpage都满了，对于这三种情况做相应的处理）

B+树删除：删除在叶子结点上进行，使用填充因子控制树的删除变化，B+树的删除也仅在叶子结点进行，当叶子结点中的最大关键字被删除时，其在非终端结点中的值可以作为一个“分界关键字”存在。若因删除而使结点中关键字的个数少于m/2 （m/2结果取上界，如5/2结果为3）时，其和兄弟结点的合并过程亦和B-树类似。

B+树比B 树更适合实际应用中操作系统的文件索引和数据库索引?

1) B+树的磁盘读写代价更低

B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。 举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘快。而B+树内部结点只需要1个盘快(全部关键字都在叶结点的缘故)。当需要把数据读入内存中的时候，B-树就比B+树多一次盘块查找时间(在磁盘中就是盘片旋转的时间，对于B+树找到叶结点就可以，另外B+树可以顺序查找)。

2) B+树的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

五.红黑树

红黑树，一种二叉查找树，但在每个结点上增加一个存储位表示结点的颜色，可以是Red或Black。通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出两倍，因而是接近平衡的。

性质：节点是红色或黑色。根是黑色。 所有叶子都是黑色（叶子是NIL节点）。 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点）从任一节点到其每个叶子的所有简单路径 都包含相同数目的黑色节点。

红黑树与AVL树的比较

1）红黑树并不追求“完全平衡”，只要求部分达到平衡要求，二者插入、删除、查找的时间复杂度都为O(logN)

2)红黑树任何不平衡都会在三次旋转内解决，而AVL树可能需要多次旋转

六.哈夫曼树（最优二叉树）

n个权值构造的n个叶子结点的二叉树，带权路径长度最小的二叉树

构造过程：

1）通过给定的n个权值构成n棵二叉树的集合F,其每棵二叉树中只有一个带权为wi的根节点

1）选取两个最小权值的根节点作为左右子树构造一棵新的二叉树，且置根节点为左右子树权值之和

2）然后在权值集合中删除这两棵树，并且将新得到的二叉树加入到F中

3）重复2）和3）直至F只含一棵树为止

赫夫曼编码

将构造得到的哈夫曼树左右分别用0、1进行编码以此得到每个根节点的编码