二分查找具有很高的查找效率，但是由于二分查找要求表中记录关于关键字有序，且不能用链表做存储结构，因为当表的插入删除操作十分的频繁的时候，为了维护表的有序性，需要移动表中很多的元素。由于移动记录会引起额外的时间开销，就会抵消二分查找的优点。**二叉排序树不仅具有二分查找的效率，同时又便于在查找表中进行记录的增加和删除操作。**

**性质**

1.若左子树不为空，则左子树上所有节点的值均小于根节点的值

2.若右子树不为空，则右子树上所有节点的值均大于根节点的值

3.左右子树同样是二叉排序树

4.对二叉排序树进行中序遍历，会得到一个关键字有序的记录序列

**内部结构**

同二叉树相同

public class BSTreeNode{ //二叉树节点结构

private Object date; //数据域

private BSTreeNode lchild; //左孩子域

private BSTreeNode rchild; //右孩子域

}

private BSTreeNode root;//根节点域

… //一些构造函数

**功能**

**1.顺序遍历（二叉树中序遍历）**

**//二叉排序树的顺序遍历等于二叉树的中序遍历**

**public void inOrderTraverse(BSTreeNode T){ //递归方法**

**if(T!=null){**

**inOrderTraverse(T.lchild);**

**System.out.print(T.date);**

**inOrderTraverse(T.rchild);**

**}**

**}**

**public void inOrderTraverse(){ //非递归方法**

**BSTreeNode T = root;**

**LinkedList<BSTreeNode> s = new LinkedList<>();**

**while(T!=null){**

**s.push(T);**

**while(!s.isEmpty()){**

**s.push(s.peek().lchild);**

**}**

**s.pop();**

**if(!s.isEmpty()){**

**T=s.pop();**

**System.out.print(T.date);**

**s.push(T.rchild);**

**}**

**}**

**}**

**2.查找**

**步骤：**

**（1）若查找树为空，则查找失败，返回-1**

**（2）若查找树非空**

**1.若给定值key等于根节点的关键字，则查找成功，结束查找过程**

**2.若给定值key小于根节点的关键字，则继续在根节点的左子树上进行**

**3.若给定值key大于根节点的关键字，则继续在根节点的右子树上进行**

**采用递归方法：**

**public Object searchBST(Object key){**

**if(key==null||root==null)//二叉排序树为空的时候返回null**

**return null;**

**else**

**return searchBST(key,root);**

**}**

**public Object searchBST(Object key, BSTreeNode root){**

**if(root.date.equals(key))//当搜索到节点**

**return root.date; //返回数据**

**if((Integer)key<(Integer)root.date){//小于当前的根节点，**

**return searchBST(key, root.lchild); //递归搜索左子树**

**}**

**if((Integer)key>(Integer)root.date){//大于当前根节点**

**return searchBST(key, root.rchild);//递归搜素右子树**

**}**

**return null;//搜索失败返回null**

**}**

**3.插入**

**假设待插入的节点的关键字为key，为了将其插入到表中，先要将它放入二叉排序树中进行查找，若查找成功，则按照二叉排序树定义，该节点已经存在，不用插入。否则，将新节点插入到表中，新插入的节点一定作为叶子节点。**

**public boolean insertBST(Object key,BSTreeNode T){ //递归添加key**

**if(root==null){ //如果树为空，建立根节点。**

**root=new BSTreeNode(key);//存入树中的root**

**return true;**

**}**

**if((Integer)T.date==(Integer)key){//如果当前根节点等于key的节点，不插入**

**return false;//返回false**

**}**

**if((Integer)key<(Integer)T.date){//如果key小于当前的根节点**

**if(T.lchild==null){ //根节点的左孩子为空**

**T.lchild=new BSTreeNode(key);//插入**

**return true;//返回结果true**

**}**

**else//当前的根节点左孩子不为空**

**return insertBST(key,T.lchild);//递归插入当前根节点左子树**

**}**

**if((Integer)key>(Integer)T.date){//如果key大于当前的根节点**

**if(T.rchild==null){//根节点的右孩子为空**

**T.rchild=new BSTreeNode(key);//直接插入**

**return true;//返回结果**

**}**

**else//右孩子不为空**

**return insertBST(key,T.rchild);//递归插入当前节点的右子树**

**}**

**return false;//没有插入，返回失败，false**

**}**

**4.删除 平均时间复杂度为O（）**

**从二叉排序树删除一个节点，要保证删除后仍然是一颗二叉排序树。**

**删除操作可以分为4中情况：**

**（1）若待删除的节点是叶子节点，则可以直接删除。若该节点同时是根节点，则删除后的二叉排序树变为空树。**

**（2）待删除的节点只有左子树，而无右子树。可以直接将其左子树的根节点代替被删除出节点的位置，即若被删除的节点为其双亲节点的左孩子，则将被删除节点的唯一的左孩子收为其双亲节点的左孩子，否则收为其双亲节点的右孩子。**

**（3）待删除的节点只有右子树，而无左子树。可以直接将其左子树的根节点代替被删除出节点的位置，即若被删除的节点为其双亲节点的左孩子，则将被删除节点的唯一的左孩子收为其双亲节点的左孩子，否则收为其双亲节点的右孩子。**

**就是，待删除节点只有左孩子或者右孩子。待删除节点的左或右孩子，在待删除节点删除的时候，将仅有的孩子添加到刚被删除节点的位置。**

**（4）待删除节点既有左子树又有右子树。可以用被删除节点在中序遍历序列下的前驱节点（或其中序遍历序列下的后继节点）代替被删除节点，同时删除用于代替的那个节点。而被删除节点在中序遍历下的前驱节点无右子树（第二种情况），被删除节点在中序遍历下的后继节点无左子树（第三种情况）。**

**用待删除节点的中序遍历前驱或者后继代替。**

**待删除节点p在中序遍历下的后继节点为其右子树的最左孩子，前驱节点为其左子树的左右孩子。**

**public Object removeBST(Object key,BSTreeNode p,BSTreeNode parent){**

**//p为当前根节点，parent为p的双亲节点**

**if(p!=null){**

**if((Integer)key<(Integer) p.date){ //key比当前根节点小的时候**

**return removeBST(key,p.lchild,p);//在左子树中搜索要删除的值**

**}**

**if((Integer)key>(Integer)p.date){ //key比当前根节点大的时候**

**return removeBST(key,p.rchild,p);//在右子树中搜索要删除的值**

**}**

**//通过上两步的递归查找，找到要删除的值为当前的节点p**

**if(p.lchild!=null&&p.rchild!=null){ //情况4，当前待删除节点p既有左孩子又有右孩子**

**//用中序的前驱或者后继代替**

**BSTreeNode innext = p.rchild; //寻找p的中序后继节点innext**

**while(innext.lchild!=null) //p的中序后继节点为其右子树的最左孩子**

**innext=innext.lchild;**

**p.date=innext.date; //用后继节点innext替换当前节点p**

**//用后继节点替换后，该节点无左子树，在其右子树中删除innext**

**return removeBST(innext.date,p.rchild,p); //递归删除innext节点**

**}else { //p的左右子树不同时为空的时候**

**if (p.rchild == null ) {//情况2，待删除节点无右子树，只有左子树**

**if (parent.lchild == p)//待删除节点是其双亲节点的左子树**

**parent.lchild = p.lchild;//用待删除节点的左子树代替**

**if (parent.rchild == p) //待删除节点是其双亲节点的右子树**

**parent.rchild = p.lchild;//用待删除节点的左子树代替**

**return p.date;**

**}**

**if (p.lchild == null ) {//情况3，待删除节点无左子树，只有右子树**

**if (parent.lchild == p)//待删除节点是其双亲节点的左子树**

**parent.lchild = p.rchild;//用待删除节点的右子树代替**

**if (parent.rchild == p) //待删除节点是其双亲节点的右子树**

**parent.rchild = p.rchild;//用待删除节点的右子树代替**

**return p.date;**

**}**

**if (p.lchild == null && p.rchild == null) {//情况1，当p既没有左子树也没有右子树的是时候，p是叶子节点**

**if (parent.lchild == p) //直接删除**

**parent.lchild = null;**

**if (parent.rchild == p) //直接删除**

**parent.rchild = null;**

**return p.date;**

**}**

**if (parent == null) {//待删除节点p双亲节点parent为null,p为根节点**

**if(p.lchild!=null)**

**root = p.lchild;**

**else**

**root = p.rchild;**

**return p.date;**

**}**

**}**

**}**

**return null;}**

**二叉排序树的查找效率与二叉树的形状有关系，对于给定序列的二叉排序树，若其左右子树均匀分部，则查找过程类似于有序表的二分查找，时间复杂度为O（），若给定序列原来有序，则建立的二叉排序树退化成为单链表，其查找效率跟顺序查找效率一样，时间复杂度为O(n).**