Tree -- 树接口

BinaryTree -- 二叉树

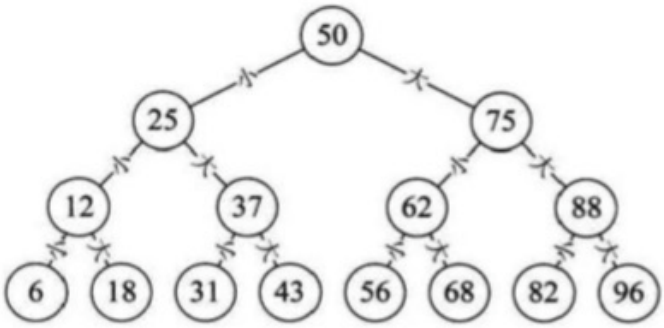
经典例子

有一个100以内的正整数，猜数字。结果是‘大了’、‘小了’和‘猜对了’。

其实最多只需要猜 7 次就可以得出答案，而不用随意猜。

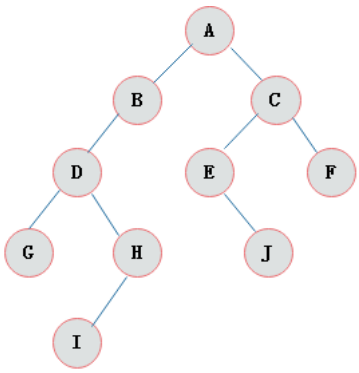
被猜数字	第一次	第二次	第三次	第四次	第五次	第六次	第七次
39	50	25	37	43	40	38	39
82	50	75	88	82			
99	50	75	88	96	98	99	
1	50	25	12	6	3	2	1

二叉树排列



定义

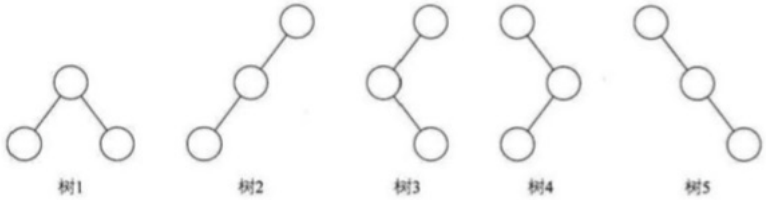
二叉树是n(n>=0)个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两棵互不相交的、分别称



特点

1. 每个结点最多有两颗子树，所以二叉树中不存在度大于2的结点
2. 左子树和右子树是有顺序的，次序不能任意颠倒
3. 即使树中某结点只有一棵子树，也要区分它是左子树还是右子树

假设有三个节点的二叉树，那么他的形态可以为以下五种：



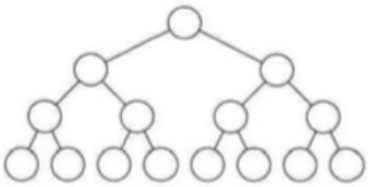
特殊二叉树

斜树

所有的结点都只有左子树的二叉树叫左斜树。
所有结点都是只有右子树的二叉树叫右斜树。
这两者统称为斜树，如上图树2和树5

满二叉树

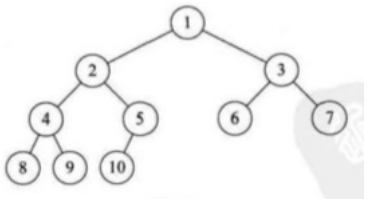
在一棵二叉树中。如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树



满二叉树的特点有：
1）叶子只能出现在最下一层。出现在其它层就不可能达成平衡。
2）非叶子结点的度一定是2。
3）在同样深度的二叉树中，满二叉树的结点个数最多，叶子数最多
4）节点数n和深度k的关系 n=2^k - 1　说明：
5）第k层上的节点数n ，那么记做n = 2^(k-1)

完全二叉树

若设二叉树的深度为h，除第 h 层外，其它各层的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树



特点：
1）叶子结点只能出现在最下层和次下层
2）最下层的叶子结点集中在树的左部
3）倒数第二层若存在叶子结点，一定在右部连续位置
4）如果结点度为1，则该结点只有左孩子，即没有右子树
5）同样结点数目的二叉树，完全二叉树深度最小

注：满二叉树一定是完全二叉树，但反过来不一定成立

性质

1）在二叉树的第i层上最多有2^{i-1} 个节点 。（i>=1）
2）二叉树中如果深度为k,那么最多有2^k-1个节点。(k>=1）
3）$n_0=n_2+1$ n_0表示度数为0的节点数，n_2表示度数为2的节点数。
4）在完全二叉树中，具有n个节点的完全二叉树的深度为$[\log_2 n]+1$，其中$[\log_2 n]$是向下取整。
5）若对含 n 个结点的完全二叉树从上到下且从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点有如下特性：

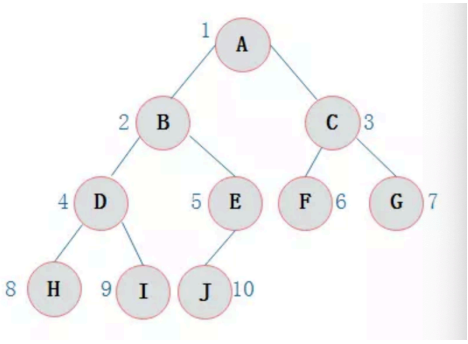
> (1) 若 i=1，则该结点是二叉树的根，无双亲, 否则，编号为 [i/2] 的结点为其双亲结点;
> (2) 若 2i>n，则该结点无左孩子， 否则，编号为 2i 的结点为其左孩子结点；
> (3) 若 2i+1>n，则该结点无右孩子结点， 否则，编号为2i+1 的结点为其右孩子结点。

存储结构

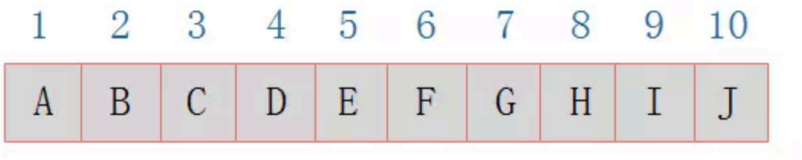
顺序存储

二叉树的顺序存储结构就是使用一维数组存储二叉树中的结点，并且结点的存储位置，就是数组的下标索引，这里分别说明

完全二叉树



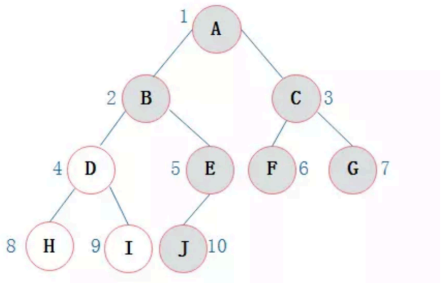
图：完全二叉树



图：顺序存储

结论：当二叉树为完全二叉树时，结点数刚好填满数组

普通二叉树

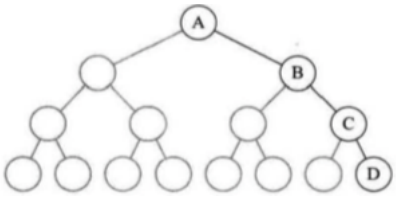


图：二叉树（浅色的表示节点不存在）

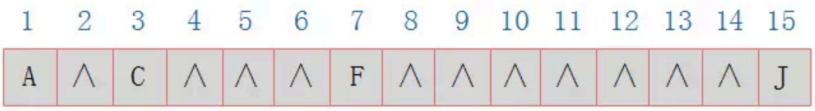
图：顺序存储(∧表示数组中此位置没有存储结点)

结论：顺序存储结构中已经出现了空间浪费的情况

斜树



图：斜树（空的表示节点不存在）



图：顺序存储(∧表示数组中此位置没有存储结点)

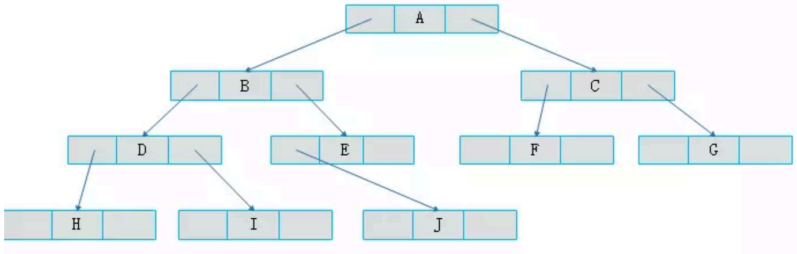
对于这种右斜树极端情况，采用顺序存储的方式是十分浪费空间的。因此，顺序存储一般适用于完全二叉树

链式存储

顺序存储不能满足二叉树的存储需求，那么考虑采用链式存储。由二叉树定义可知，二叉树的每个结点最多有两个孩子

那么节点类的代码可以定义如下：

```
static class Node{
    private Object data;
    private Node leftNode;
    private Node rightNode;
}
```

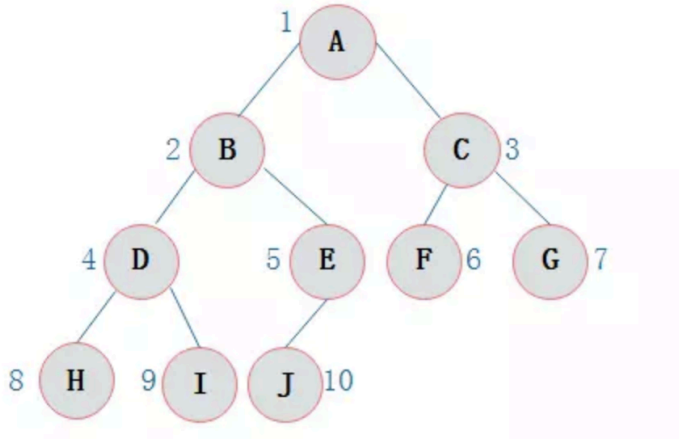


二叉树图表示

二叉树遍历

二叉树的遍历是指从二叉树的根结点出发，按照某种次序依次访问二叉树中的所有结点，使得每个结点被访问一次，且

二叉树的访问次序可以分为四种：前序遍历、中序遍历、后序遍历、层序遍历



前序遍历

前序遍历通俗的说就是从二叉树的根结点出发，当第一次到达结点时就输出结点数据，按照先向左在向右的方向访问

```
从根结点出发，则第一次到达结点A，故输出A；
继续向左访问，第一次访问结点B，故输出B；
按照同样规则，输出D，输出H；
当到达叶子结点H，返回到D，此时已经是第二次到达D，故不在输出D，进而向D右子树访问，D右子树不为空，则访问至I，第一次到达I
I为叶子结点，则返回到D，D左右子树已经访问完毕，则返回到B，进而到B右子树，第一次到达E，故输出E；
向E左子树，故输出J；
按照同样的访问规则，继续输出C、F、G；


遍历输出为：
ABDHIEJCFG
```

中序遍历

中序遍历就是从二叉树的根结点出发，当第二次到达结点时就输出结点数据，按照先向左在向右的方向访问

```
从根结点出发，则第一次到达结点A，不输出A，继续向左访问，第一次访问结点B，不输出B；继续到达D，H；
到达H，H左子树为空，则返回到H，此时第二次访问H，故输出H；
H右子树为空，则返回至D，此时第二次到达D，故输出D；
D右子树为I，第一次到达I，I的左子树为空，则返回I，此时第二次访问I，故输出I；
由D返回至B，第二次到达B，故输出B；
按照同样规则继续访问，输出J、E、A、F、C、G；

遍历输出为：
HDIBJEAFCG
```

后序遍历

后序遍历就是从二叉树的根结点出发，当第三次到达结点时就输出结点数据，按照先向左在向右的方向访问

```
从根结点出发，则第一次到达结点A，不输出A，继续向左访问，第一次访问结点B，不输出B；继续到达D，H；
```

```
到达H，H左子树为空，则返回到H，此时第二次访问H，不输出H；
H右子树为空，则返回至H，此时第三次到达H，故输出H；
由H返回至D，第二次到达D，不输出D；
继续访问至I，I左右子树均为空，故第三次访问I时，输出I；
返回至D，此时第三次到达D，故输出D；
按照同样规则继续访问，输出J、E、B、F、G、C，A；

遍历输出为：
HIDJEBFGCA
```

层序遍历

层次遍历就是按照树的层次自上而下的遍历二叉树

```

ABCDEFGHIJ
```

遍历总结

中序遍历:左子树——》根节点——》右子树
前序遍历:根节点——》左子树——》右子树
后序遍历:左子树——》右子树——》根节点

代码实现

```java
//中序遍历
public void infixOrder(Node current){
    if(current != null){
        infixOrder(current.leftChild);
        System.out.print(current.data+" ");
        infixOrder(current.rightChild);
    }
}

//前序遍历
public void preOrder(Node current){
    if(current != null){
        System.out.print(current.data+" ");
        preOrder(current.leftChild);
        preOrder(current.rightChild);
    }
}

//后序遍历
public void postOrder(Node current){
    if(current != null){
        postOrder(current.leftChild);
        postOrder(current.rightChild);
        System.out.print(current.data+" ");
    }
}

```

```java
//层序遍历,通过队列，先入先出
public void sequenceOrder(Node current){
    Queue<Node> queue = new LinkedList<>();
    while (current != null){
        System.out.print(current.data + " ");
        if (current.leftChild != null){
            queue.add(current.leftChild);
        }
        if (current.rightChild != null){
            queue.add(current.rightChild);
        }
        current = queue.poll();
    }
}
```

查找最大值和最小值

这没什么好说的，要找最小值，先找根的左节点，然后一直找这个左节点的左节点，直到找到没有左节点的节点，那么这个节一直找根节点的右节点，直到没有右节点，则就是最大值。

```java
//找到最大值
public Node findMax(){
    Node current = root;
    Node maxNode = current;
    while(current != null){
        maxNode = current;
        current = current.rightChild;
    }
    return maxNode;
}

//找到最小值
public Node findMin(){
    Node current = root;
    Node minNode = current;
    while(current != null){
        minNode = current;
        current = current.leftChild;
    }
    return minNode;
}
```

删除节点

删除节点是二叉搜索树中最复杂的操作，删除的节点有三种情况，前两种比较简单，但是第三种却很复杂

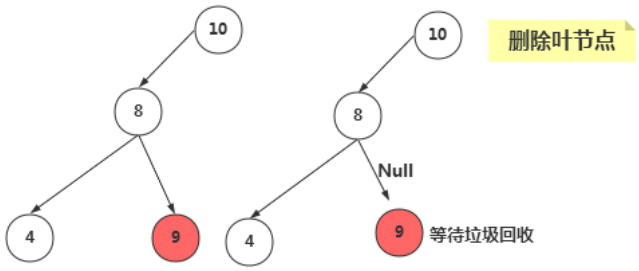
1、该节点是叶节点（没有子节点）

2、该节点有一个子节点

3、该节点有两个子节点

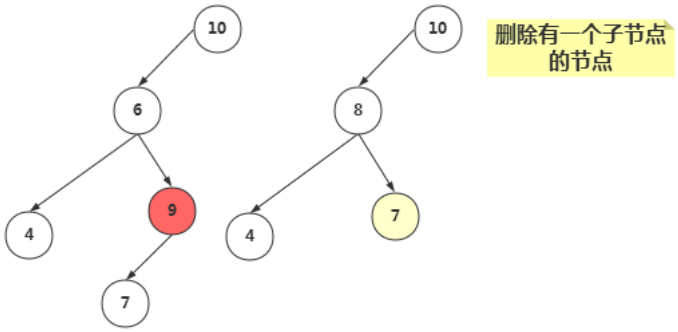
删除没有子节点的节点

要删除叶节点，只需要改变该节点的父节点引用该节点的值，即将其引用改为 null 即可。要删除的节点依然存在，但是Java语言的垃圾回收机制，我们不需要非得把节点本身删掉，一旦Java意识到程序不在与该节点有关联，就会自动把它清



删除有一个子节点的节点

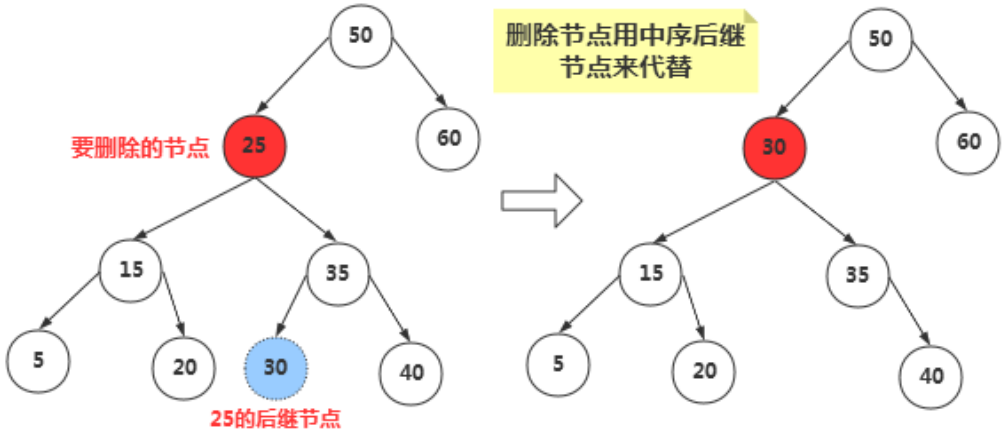
删除有一个子节点的节点，我们只需要将其父节点原本指向该节点的引用，改为指向该节点的子节点即可



删除有两个子节点的节点

当删除的节点存在两个子节点，那么删除之后，就需要被删除节点的子节点来代替被删除的节点，那么用哪一个节点来

我们知道二叉搜索树中的节点是按照关键字来进行排列的，某个节点的关键字次高节点是它的中序遍历后继节点。用后该二叉搜索树还是有序的。



后继节点也就是：比删除节点大的最小节点，或者比删除节点小的最大节点

总结

1. 如果左子树不为空，用左子树最大值替换该节点，并删除左子树中最大值节点
2. 如果右子树不为空，用右子树最小值替换该节点，并删除右子树中最小值节点
3. 如果左右子树都为空，直接删除该节点

删除真的有必要吗?

通过上面的删除分类讨论，我们发现删除其实是挺复杂的，那么其实我们可以不用真正的删除该节点，只需要在Node类
当该字段为true时，表示该节点已经删除，反正没有删除。那么我们在做比如find()等操作的时候，要先判断isDelete字段是
会改变树的结构

二叉树的效率

从前面的大部分对树的操作来看，都需要从根节点到下一层一层的查找。

一颗满树，每层节点数大概为2n-1，那么最底层的节点个数比树的其它节点数多1，因此，查找、插入或删除节点的操作大
另外四分之一的节点在倒数第二层，依次类推。

总共N层共有2n-1个节点，那么时间复杂度为O(logn),底数为2。

在有1000000 个数据项的无序数组和链表中，查找数据项平均会比较500000 次，但是在有1000000个节点的二叉树中，

有序数组可以很快的找到数据项，但是插入数据项的平均需要移动 500000 次数据项，在 1000000 个节点的二叉树中插入
20次方），在加上很短的时间来连接数据项。

同样，从 1000000 个数据项的数组中删除一个数据项平均需要移动 500000 个数据项，而在 1000000 个节点的二叉树中
来找到他，然后在花一点时间来找到它的后继节点，一点时间来断开节点以及连接后继节点。

所以，树对所有常用数据结构的操作都有很高的效率。

遍历可能不如其他操作快，但是在大型数据库中，遍历是很少使用的操作，它更常用于程序中的辅助算法来解析算术或其

完整代码

Node -- 节点类

```
1  package day6.tree.二叉树;
2
3  public class Node {
4      int data;   //节点数据
5      Node leftChild; //左子节点的引用
6      Node rightChild; //右子节点的引用
7      boolean isDelete;//表示节点是否被删除
8
9      public Node(int data){
```

```
        this.data = data;
    }
    //打印节点内容
    public void display(){
        System.out.println(data);
    }

}
```

Tree -- 树接口

```
package day6.tree.二叉树;

public interface Tree {
    //查找节点
    Node find(int key);

    //插入新节点
    boolean insert(int data);

     //中序遍历
     void infixOrder(Node current);

     //前序遍历
     void preOrder(Node current);

     //后序遍历
     void postOrder(Node current);

     //查找最大值
     Node findMax();

     //查找最小值
     Node findMin();

     //删除节点
     boolean delete(int key);


}
```

BinaryTree -- 二叉树

```
package day6.tree.二叉树;

public class BinaryTree implements Tree {
    //表示根节点
    private Node root;

    //查找节点
    public Node find(int key) {
        Node current = root;
         while(current != null){
             if(current.data > key){//当前值比查找值大，搜索左子树
                 current = current.leftChild;
             }else if(current.data < key){//当前值比查找值小，搜索右子树
```

```
                current = current.rightChild;
            }else{
                return current;
            }
        }
        return null;//遍历完整个树没找到，返回null
    }

    //插入节点
    public boolean insert(int data) {
        Node newNode = new Node(data);
        if(root == null){//当前树为空树，没有任何节点
            root = newNode;
            return true;
        }else{
            Node current = root;
            Node parentNode = null;
            while(current != null){
                parentNode = current;
                if(current.data > data){//当前值比插入值大，搜索左子节点
                    current = current.leftChild;
                    if(current == null){//左子节点为空，直接将新值插入到该节点
                        parentNode.leftChild = newNode;
                        return true;
                    }
                }else{
                    current = current.rightChild;
                    if(current == null){//右子节点为空，直接将新值插入到该节点
                        parentNode.rightChild = newNode;
                        return true;
                    }
                }
            }
        }
        return false;
    }

     //层序遍历
    public void sequenceOrder(Node current){
    Queue<Node> queue = new LinkedList<>();
    while (current != null){
        System.out.print(current.data + " ");
        if (current.leftChild != null){
            queue.add(current.leftChild);
        }
        if (current.rightChild != null){
            queue.add(current.rightChild);
        }
        current = queue.poll();
    }
}

    //中序遍历
    public void infixOrder(Node current){
```

```
        if(current != null){
            infixOrder(current.leftChild);
            System.out.print(current.data+" ");
            infixOrder(current.rightChild);
        }
    }

    //前序遍历
    public void preOrder(Node current){
        if(current != null){
            System.out.print(current.data+" ");
            infixOrder(current.leftChild);
            infixOrder(current.rightChild);
        }
    }

    //后序遍历
    public void postOrder(Node current){
        if(current != null){
            infixOrder(current.leftChild);
            infixOrder(current.rightChild);
            System.out.print(current.data+" ");
        }
    }
    //找到最大值
    public Node findMax(){
        Node current = root;
        Node maxNode = current;
        while(current != null){
            maxNode = current;
            current = current.rightChild;
        }
         return maxNode;
     }
     //找到最小值
     public Node findMin(){
         Node current = root;
         Node minNode = current;
         while(current != null){
             minNode = current;
             current = current.leftChild;
         }
         return minNode;
     }

     @Override
     public boolean delete(int key) {
         Node current = root;
         Node parent = root;
         boolean isLeftChild = false;
         //查找删除值，找不到直接返回false
         while(current.data != key){
             parent = current;
```

```
            if(current.data > key){
                isLeftChild = true;
                current = current.leftChild;
            }else{
                isLeftChild = false;
                current = current.rightChild;
            }
            if(current == null){
                return false;
            }
        }
        //如果当前节点没有子节点
        if(current.leftChild == null && current.rightChild == null){
            if(current == root){
                root = null;
            }else if(isLeftChild){
                parent.leftChild = null;
            }else{
                parent.rightChild = null;
            }
            return true;

            //当前节点有一个子节点，右子节点
        }else if(current.leftChild == null && current.rightChild != null){
            if(current == root){
                root = current.rightChild;
            }else if(isLeftChild){
                parent.leftChild = current.rightChild;
            }else{
                parent.rightChild = current.rightChild;
            }
            return true;
            //当前节点有一个子节点，左子节点
        }else if(current.leftChild != null && current.rightChild == null){
            if(current == root){
                root = current.leftChild;
            }else if(isLeftChild){
                parent.leftChild = current.leftChild;
            }else{
                parent.rightChild = current.leftChild;
            }
            return true;
        }else{
            //当前节点存在两个子节点
            Node successor = getSuccessor(current);
            if(current == root){
                root= successor;
            }else if(isLeftChild){
                parent.leftChild = successor;
            }else{
                parent.rightChild = successor;
            }
            successor.leftChild = current.leftChild;
        }
```

```
        return false;

    }

    public Node getSuccessor(Node delNode){
        Node successorParent = delNode;
        Node successor = delNode;
        Node current = delNode.rightChild;
        while(current != null){
            successorParent = successor;
            successor = current;
            current = current.leftChild;
        }
        //后继节点不是删除节点的右子节点，将后继节点替换删除节点
        if(successor != delNode.rightChild){
            successorParent.leftChild = successor.rightChild;
            successor.rightChild = delNode.rightChild;
        }

        return successor;
    }

    public static void main(String[] args) {
        BinaryTree bt = new BinaryTree();
        bt.insert(50);
        bt.insert(20);
        bt.insert(80);
        bt.insert(10);
        bt.insert(30);
        bt.insert(60);
        bt.insert(90);
        bt.insert(25);
        bt.insert(85);
        bt.insert(100);
        bt.delete(10);//删除没有子节点的节点
        bt.delete(30);//删除有一个子节点的节点
        bt.delete(80);//删除有两个子节点的节点
        System.out.println(bt.findMax().data);
        System.out.println(bt.findMin().data);
        System.out.println(bt.find(100));
        System.out.println(bt.find(200));

        bt = new BinaryTree();
        bt.insert(50);
        bt.insert(65);
        bt.insert(30);
        bt.insert(60);
        bt.insert(20);
        bt.insert(40);
        bt.insert(55);
        bt.sequenceOrder(bt.find(50));
    }

```

```
228 }
```