

经典例子

定义

特点

特殊二叉树

斜树

满二叉树

完全二叉树

性质

存储结构

顺序存储

完全二叉树

普通二叉树

斜树

链式存储

二叉树遍历

前序遍历

中序遍历

后序遍历

层序遍历

遍历总结

代码实现

查找最大值和最小值

删除节点

删除没有子节点的节点

删除有一个子节点的节点

删除有两个子节点的节点

总结

删除真的有必要吗？

二叉树的效率

完整代码

Node -- 节点类

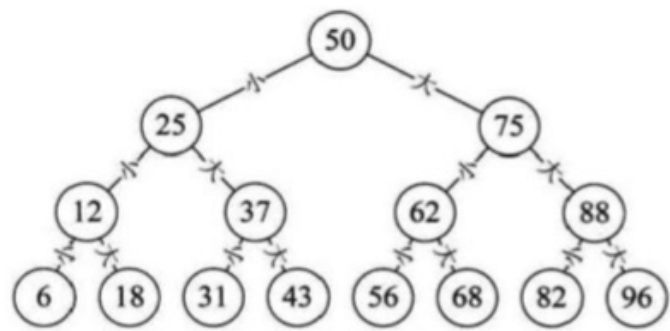
经典例子

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。

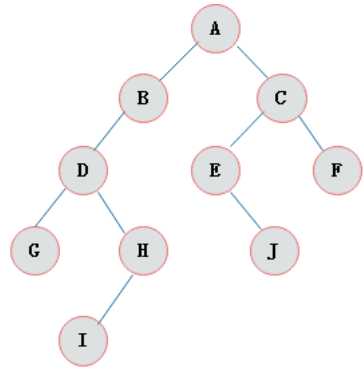
被猜数字	第一次	第二次	第三次	第四次	第五次	第六次	第七次
39	50	25	37	43	40	38	39
82	50	75	88	82			
99	50	75	88	96	98	99	
1	50	25	12	6	3	2	1

二叉树排列



定义

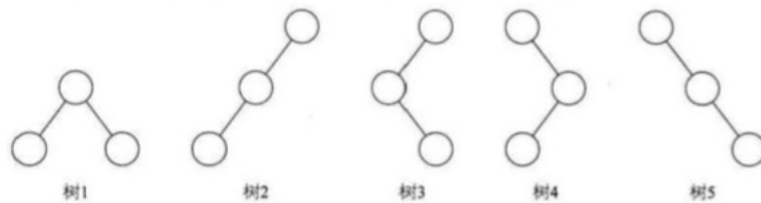
二叉树是 $n(n \geq 0)$ 个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两棵互不相交的、分别称



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

假设有三个结点的二叉树，那么他的形态可以为以下五种：



特殊二叉树

斜树

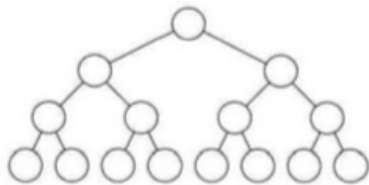
所有的结点都只有左子树的二叉树叫左斜树。

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

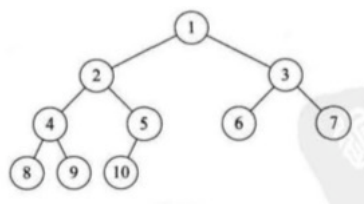


满二叉树的特点有：

- 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。 $(i \geq 1)$
- 2) 二叉树中如果深度为 k ,那么最多有 $2^k - 1$ 个节点。 $(k \geq 1)$
- 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 $\lfloor \log_2 n \rfloor + 1$ ，其中 $\lfloor \log_2 n \rfloor$ 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：

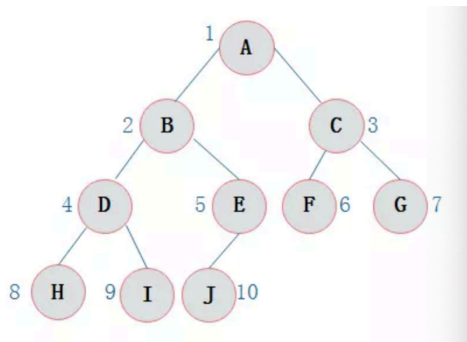
- (1) 若 $i=1$ ，则该结点是二叉树的根，无双亲，否则，编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点；
- (2) 若 $2i > n$ ，则该结点无左孩子，否则，编号为 $2i$ 的结点为其左孩子结点；
- (3) 若 $2i+1 > n$ ，则该结点无右孩子结点，否则，编号为 $2i+1$ 的结点为其右孩子结点。

存储结构

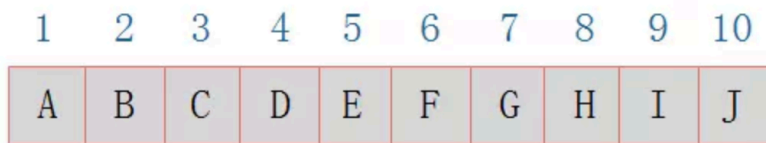
顺序存储

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

完全二叉树



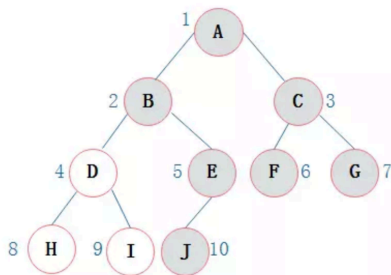
图：完全二叉树



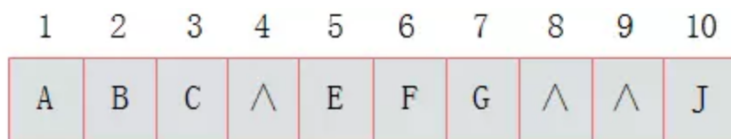
图：顺序存储

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



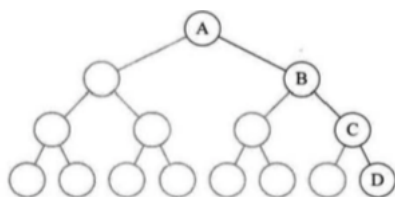
图：二叉树（浅色的表示节点不存在）



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）

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	^	C	^	^	^	F	^	^	^	^	^	^	^	J

图：顺序存储(^表示数组中此位置没有存储结点)

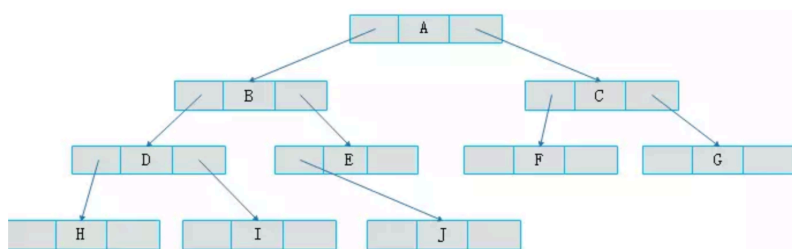
对于这种右斜树极端情况，采用顺序存储的方式是十分浪费空间的。因此，顺序存储一般适用于完全二叉树

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：

```
1 static class Node{
2     private Object data;
3     private Node leftNode;
4     private Node rightNode;
5 }
```

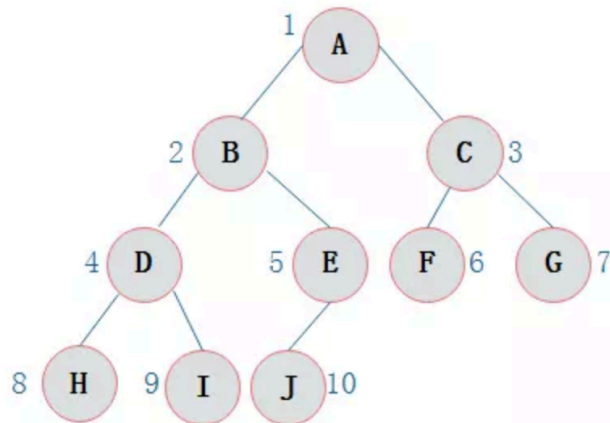


二叉树图表示

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

- 1 从根结点出发，则第一次到达结点A，故输出A；
- 2 继续向左访问，第一次访问结点B，故输出B；
- 3 按照同样规则，输出D，输出H；
- 4 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
- 5 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；
- 6 向E左子树，故输出J；
- 7 按照同样的访问规则，继续输出C、F、G；
- 8
- 9
- 10 遍历输出为：
- 11 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

- 1 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；
- 2 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；
- 3 H右子树为空，则返回至D，此时第二次到达D，故输出D；
- 4 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；
- 5 由D返回至B，第二次到达B，故输出B；
- 6 按照同样规则继续访问，输出J、E、A、F、C、G；
- 7
- 8 遍历输出为：
- 9 HDIBJEAF CG

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

- 1 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；

```
2 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;
3  H右子树为空, 则返回至H, 此时第三次到达H, 故输出H;
4  由H返回至D, 第二次到达D, 不输出D;
5  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;
6  返回至D, 此时第三次到达D, 故输出D;
7  按照同样规则继续访问, 输出J、E、B、F、G、C、A;
8
9  遍历输出为:
10  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

```
1
2  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

```
1  //中序遍历
2  public void infixOrder(Node current){
3      if(current != null){
4          infixOrder(current.leftChild);
5          System.out.print(current.data+" ");
6          infixOrder(current.rightChild);
7      }
8  }
9
10 //前序遍历
11 public void preOrder(Node current){
12     if(current != null){
13         System.out.print(current.data+" ");
14         preOrder(current.leftChild);
15         preOrder(current.rightChild);
16     }
17 }
18
19 //后序遍历
20 public void postOrder(Node current){
21     if(current != null){
22         postOrder(current.leftChild);
23         postOrder(current.rightChild);
24         System.out.print(current.data+" ");
25     }
26 }
27
```



```

28 //层序遍历,通过队列,先入先出
29 public void sequenceOrder(Node current){
30     Queue<Node> queue = new LinkedList<>();
31     while (current != null){
32         System.out.print(current.data + " ");
33         if (current.leftChild != null){
34             queue.add(current.leftChild);
35         }
36         if (current.rightChild != null){
37             queue.add(current.rightChild);
38         }
39         current = queue.poll();
40     }
41 }

```

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，那么这个节点就是最小值。要找最大值，先找根的右节点，然后一直找这个右节点的右节点，直到找到没有右节点的节点，那么这个节点就是最大值。

```

1 //找到最大值
2 public Node findMax(){
3     Node current = root;
4     Node maxNode = current;
5     while(current != null){
6         maxNode = current;
7         current = current.rightChild;
8     }
9     return maxNode;
10 }
11
12 //找到最小值
13 public Node findMin(){
14     Node current = root;
15     Node minNode = current;
16     while(current != null){
17         minNode = current;
18         current = current.leftChild;
19     }
20     return minNode;
21 }

```

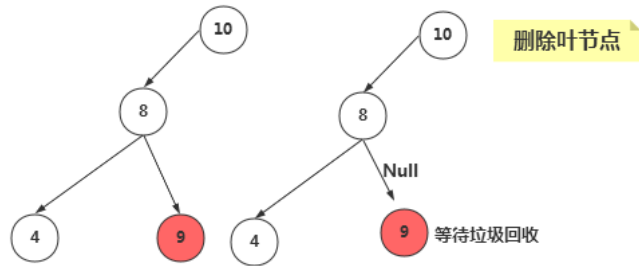
删除节点

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

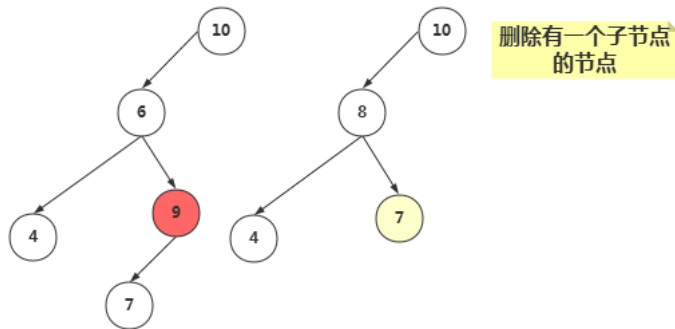
删除没有子节点的节点

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，就会自动把它清理掉。



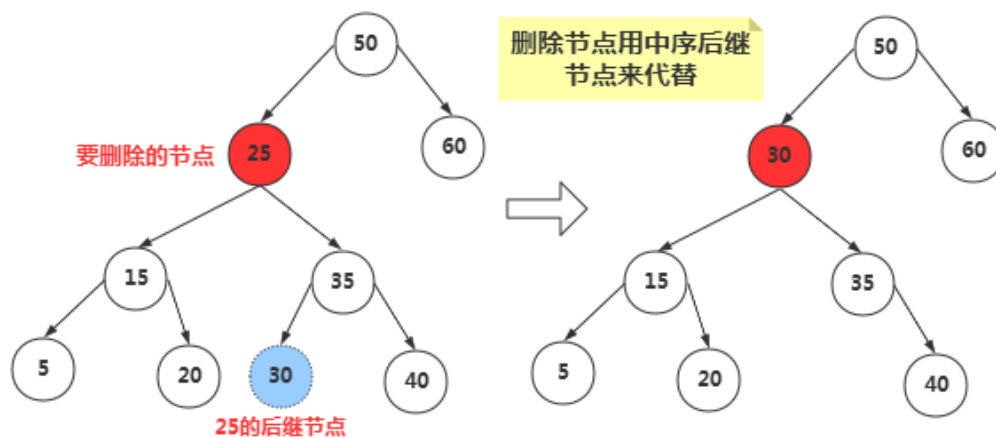
删除有一个子节点的节点

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代替呢？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。用后继节点代替该节点，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的isDelete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字段是否改变树的结构

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 2^{n-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才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 2^n-1 个节点，那么时间复杂度为 $O(\log n)$ ，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他

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){
```

```

10         this.data = data;
11     }
12     //打印节点内容
13     public void display(){
14         System.out.println(data);
15     }
16
17 }

```

Tree -- 树接口

```

1  package day6.tree.二叉树;
2
3  public interface Tree {
4      //查找节点
5      Node find(int key);
6
7      //插入新节点
8      boolean insert(int data);
9
10     //中序遍历
11     void infixOrder(Node current);
12
13     //前序遍历
14     void preOrder(Node current);
15
16     //后序遍历
17     void postOrder(Node current);
18
19     //查找最大值
20     Node findMax();
21
22     //查找最小值
23     Node findMin();
24
25     //删除节点
26     boolean delete(int key);
27
28
29 }

```

BinaryTree -- 二叉树

```

1  package day6.tree.二叉树;
2
3  public class BinaryTree implements Tree {
4      //表示根节点
5      private Node root;
6
7      //查找节点
8      public Node find(int key) {
9          Node current = root;
10         while(current != null){
11             if(current.data > key){//当前值比查找值大，搜索左子树
12                 current = current.leftChild;
13             }else if(current.data < key){//当前值比查找值小，搜索右子树

```

```

14         current = current.rightChild;
15     }else{
16         return current;
17     }
18 }
19 return null; //遍历完整棵树没找到, 返回null
20 }
21
22 //插入节点
23 public boolean insert(int data) {
24     Node newNode = new Node(data);
25     if(root == null){ //当前树为空树, 没有任何节点
26         root = newNode;
27         return true;
28     }else{
29         Node current = root;
30         Node parentNode = null;
31         while(current != null){
32             parentNode = current;
33             if(current.data > data){ //当前值比插入值大, 搜索左子节点
34                 current = current.leftChild;
35                 if(current == null){ //左子节点为空, 直接将新值插入到该节点
36                     parentNode.leftChild = newNode;
37                     return true;
38                 }
39             }else{
40                 current = current.rightChild;
41                 if(current == null){ //右子节点为空, 直接将新值插入到该节点
42                     parentNode.rightChild = newNode;
43                     return true;
44                 }
45             }
46         }
47     }
48     return false;
49 }
50
51 //层序遍历
52 public void sequenceOrder(Node current){
53     Queue<Node> queue = new LinkedList<>();
54     while (current != null){
55         System.out.print(current.data + " ");
56         if (current.leftChild != null){
57             queue.add(current.leftChild);
58         }
59         if (current.rightChild != null){
60             queue.add(current.rightChild);
61         }
62         current = queue.poll();
63     }
64 }
65
66 //中序遍历
67 public void infixOrder(Node current){

```

```

68         if(current != null){
69             infixOrder(current.leftChild);
70             System.out.print(current.data+" ");
71             infixOrder(current.rightChild);
72         }
73     }
74
75     //前序遍历
76     public void preOrder(Node current){
77         if(current != null){
78             System.out.print(current.data+" ");
79             infixOrder(current.leftChild);
80             infixOrder(current.rightChild);
81         }
82     }
83
84     //后序遍历
85     public void postOrder(Node current){
86         if(current != null){
87             infixOrder(current.leftChild);
88             infixOrder(current.rightChild);
89             System.out.print(current.data+" ");
90         }
91     }
92     //找到最大值
93     public Node findMax(){
94         Node current = root;
95         Node maxNode = current;
96         while(current != null){
97             maxNode = current;
98             current = current.rightChild;
99         }
100         return maxNode;
101     }
102     //找到最小值
103     public Node findMin(){
104         Node current = root;
105         Node minNode = current;
106         while(current != null){
107             minNode = current;
108             current = current.leftChild;
109         }
110         return minNode;
111     }
112
113     @Override
114     public boolean delete(int key) {
115         Node current = root;
116         Node parent = root;
117         boolean isLeftChild = false;
118         //查找删除值，找不到直接返回false
119         while(current.data != key){
120             parent = current;

```

```

121         if(current.data > key){
122             isLeftChild = true;
123             current = current.leftChild;
124         }else{
125             isLeftChild = false;
126             current = current.rightChild;
127         }
128         if(current == null){
129             return false;
130         }
131     }
132     //如果当前节点没有子节点
133     if(current.leftChild == null && current.rightChild == null){
134         if(current == root){
135             root = null;
136         }else if(isLeftChild){
137             parent.leftChild = null;
138         }else{
139             parent.rightChild = null;
140         }
141         return true;
142     }
143     //当前节点有一个子节点，右子节点
144     }else if(current.leftChild == null && current.rightChild != null){
145         if(current == root){
146             root = current.rightChild;
147         }else if(isLeftChild){
148             parent.leftChild = current.rightChild;
149         }else{
150             parent.rightChild = current.rightChild;
151         }
152         return true;
153     }
154     //当前节点有一个子节点，左子节点
155     }else if(current.leftChild != null && current.rightChild == null){
156         if(current == root){
157             root = current.leftChild;
158         }else if(isLeftChild){
159             parent.leftChild = current.leftChild;
160         }else{
161             parent.rightChild = current.leftChild;
162         }
163         return true;
164     }
165     }else{
166         //当前节点存在两个子节点
167         Node successor = getSuccessor(current);
168         if(current == root){
169             root = successor;
170         }else if(isLeftChild){
171             parent.leftChild = successor;
172         }else{
173             parent.rightChild = successor;
174         }
175         successor.leftChild = current.leftChild;
176     }

```

```

175         return false;
176
177     }
178
179     public Node getSuccessor(Node delNode){
180         Node successorParent = delNode;
181         Node successor = delNode;
182         Node current = delNode.rightChild;
183         while(current != null){
184             successorParent = successor;
185             successor = current;
186             current = current.leftChild;
187         }
188         //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
189         if(successor != delNode.rightChild){
190             successorParent.leftChild = successor.rightChild;
191             successor.rightChild = delNode.rightChild;
192         }
193
194         return successor;
195     }
196
197     public static void main(String[] args) {
198         BinaryTree bt = new BinaryTree();
199         bt.insert(50);
200         bt.insert(20);
201         bt.insert(80);
202         bt.insert(10);
203         bt.insert(30);
204         bt.insert(60);
205         bt.insert(90);
206         bt.insert(25);
207         bt.insert(85);
208         bt.insert(100);
209         bt.delete(10); //删除没有子节点的节点
210         bt.delete(30); //删除有一个子节点的节点
211         bt.delete(80); //删除有两个子节点的节点
212         System.out.println(bt.findMax().data);
213         System.out.println(bt.findMin().data);
214         System.out.println(bt.find(100));
215         System.out.println(bt.find(200));
216
217         bt = new BinaryTree();
218         bt.insert(50);
219         bt.insert(65);
220         bt.insert(30);
221         bt.insert(60);
222         bt.insert(20);
223         bt.insert(40);
224         bt.insert(55);
225         bt.sequenceOrder(bt.find(50));
226     }
227

```