概念

二叉查找树，也称二叉搜索树，或二叉排序树

其定义也比较简单，要么是一颗空树，要么就是具有如下性质的二叉树：

（1）若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

（2） 若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

（3） 任意节点的左、右子树也分别为二叉查找树；

（4） 没有键值相等的节点

实现

```
package day10.查找;

import day3.队列.QueueByLink;

import java.util.*;
import java.util.concurrent.ArrayBlockingQueue;

/**
 * 二叉查找树
 */
public class BinaryTreeSearch<E> {

    /**
     * 根节点
     */
    private Node root;

    static class Node<E> implements Comparable<Node> {
        private E data;
        private Node leftChild;
        private Node rightChild;

        private Comparable<Node> comparable;

        public Node(E data) {
            this(data, new Comparable<Node>() {
                @Override
                public int compareTo(Node o) {
                    if (o == null) {
```

```
                        return 1;
                    }
                    if (data.hashCode() > o.data.hashCode()) {
                        return 1;
                    } else if (data.hashCode() < o.data.hashCode()) {
                        return -1;
                    }
                    return 0;
                }
            });
        }

        public Node(E data, Comparable<Node> comparable) {
            this.data = data;
            this.comparable = comparable;
        }

        @Override
        public int compareTo(Node o) {
            return comparable.compareTo(o);
        }
    }

    public void addNode(E data) {
        Node newNode = new Node(data);
        if (root == null) {
            root = newNode;
            return;
        }
        Node current = root;
        int high = 1;
        while (current != null) {
            high++;
            if (newNode.compareTo(current) < 0) {
                if (current.leftChild == null) {
                    current.leftChild = newNode;
                    return;
                }
                current = current.leftChild;
            } else {
                if (current.rightChild == null) {
                    current.rightChild = newNode;
                    return;
                }
                current = current.rightChild;
            }
        }
    }

    //查找节点
    public Node find(E data) {
        Node newNode = new Node(data);
        Node current = root;
        while (current != null) {
```

```
            if (current.compareTo(newNode) > 0) {//当前值比查找值大，搜索左子树
                current = current.leftChild;
            } else if (current.compareTo(newNode) < 0) {//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
    }

    public Node findMax() {
        return findMax(root);
    }

    public Node findMax(Node current){
         while (current.rightChild != null) {
             current = current.rightChild;
         }
         return current;
     }

     public Node findMin() {
         return findMin(root);
     }

     public Node findMin(Node current){
         while (current.leftChild != null) {
             current = current.leftChild;
         }
         return current;
     }

     //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 + " ");
        }
    }

    //层序遍历
    public void sequenceOrder(Node current) {
        Queue<Node> queue = new ArrayBlockingQueue<>(100);
        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();
        }
    }

    /**
     * 根据目标节点及当前节点的父节点，返回数组，父节点在第一个，当前节点在第二个
     * @param data
     * @return
     */
    public Node[] findParentWithCurrent(E data){
        Node newNode = new Node(data);
        Node current = root;
        Node parent = root;
        Node result = null;
        while (current != null) {
            if (current.compareTo(newNode) > 0) {//当前值比查找值大，搜索左子树
                parent = current;
                current = current.leftChild;
            } else if (current.compareTo(newNode) < 0) {//当前值比查找值小，搜索右子树
                parent = current;
                current = current.rightChild;
            } else {
                result = current;
                break;
            }
        }
        Node[] nodes = new Node[2];
        nodes[0] = parent;
        nodes[1] = result;
        return nodes;
    }


    // 删除节点 : 减少复杂度，根节点不能删除
    public void del(E data) throws Exception {
        if (data.equals(root.data)){
            throw new Exception("根节点无法删除");
```

```
        }

        Node[] nodes = findParentWithCurrent(data);
        Node result = nodes[1];
        Node parent = nodes[0];

        if (result == null) {
            System.out.println("未找到节点");
            return;
        }

        if (result.leftChild == null && result.rightChild == null){
            if (result.equals(parent.leftChild)){
                parent.leftChild = null;
            }else {
                parent.rightChild = null;
            }
        }else if (result.leftChild != null && result.rightChild != null){
            Node successor = findMax(result.leftChild); // or findMin(result.right)
            result.data = successor.data;       // 更换值，不变化指针
            del((E) successor.data);
        }else {
            Node successor = null;
            if (result.leftChild != null){
                successor = result.leftChild;
            }else {
                successor = result.rightChild;
            }
            if (result.equals(parent.leftChild)){
                parent.leftChild = successor;
            }else {
                parent.rightChild = successor;
            }
        }
    }

    /**
     * 使用树形结构显示
     */
    public void displayTree() {
        Stack globalStack = new Stack();
        globalStack.push(root);
        int nBlank = 32;
        boolean isRowEmpty = false;
        String dot = "............................";
        System.out.println(dot + dot + dot);
        while (isRowEmpty == false) {
            Stack localStack = new Stack();
            isRowEmpty = true;
            for (int j = 0; j < nBlank; j++){
                System.out.print("-");
            }
            while (globalStack.isEmpty() == false) {
                Node temp = (Node) globalStack.pop();
```

```
                if (temp != null) {
                    System.out.print(temp.data);

                    localStack.push(temp.leftChild);
                    localStack.push(temp.rightChild);
                    if (temp.leftChild != null || temp.rightChild != null) {
                        isRowEmpty = false;

                    }
                } else {
                    System.out.print("#!");
                    localStack.push(null);
                    localStack.push(null);
                }

                //打印一些空格
                for (int j = 0; j < nBlank * 2 - 2; j++) {
                    System.out.print("*");
                }
            }


            System.out.println();
            nBlank = nBlank / 2;
            while (localStack.isEmpty() == false) {
                globalStack.push(localStack.pop());
            }
        }
        System.out.println(dot + dot + dot);

    }


    public static void main(String[] args) throws Exception {
        BinaryTreeSearch<Integer> binaryTreeSearch = new BinaryTreeSearch<>();
        binaryTreeSearch.addNode(50);
        binaryTreeSearch.addNode(30);
        binaryTreeSearch.addNode(60);
        binaryTreeSearch.addNode(20);
        binaryTreeSearch.addNode(40);
        binaryTreeSearch.addNode(65);
        binaryTreeSearch.addNode(55);
        binaryTreeSearch.addNode(56);

        binaryTreeSearch.displayTree();

        System.out.println();
        System.out.println("查找: " + binaryTreeSearch.find(40));
        System.out.println("max : " + binaryTreeSearch.findMax().data);
        System.out.println("min : " + binaryTreeSearch.findMin().data);
        System.out.println();

        // 层序遍历
        System.out.println("层序遍历");
```

```
        binaryTreeSearch.sequenceOrder(binaryTreeSearch.root);
        System.out.println();
        System.out.println();

        System.out.println("前序遍历");
        binaryTreeSearch.preOrder(binaryTreeSearch.root);
        System.out.println();
        System.out.println();

        System.out.println("后序遍历");
        binaryTreeSearch.postOrder(binaryTreeSearch.root);
        System.out.println();
        System.out.println();

        System.out.println("中序遍历");
        binaryTreeSearch.infixOrder(binaryTreeSearch.root);
        System.out.println();
        System.out.println();


        binaryTreeSearch.displayTree();
        System.out.println("删除40");
        binaryTreeSearch.del(40);
        binaryTreeSearch.displayTree();

//        System.out.println("删除55");
//        binaryTreeSearch.del(55);
//        binaryTreeSearch.displayTree();

//        System.out.println("删除60");
//        binaryTreeSearch.del(60);
//        binaryTreeSearch.displayTree();


    }

}


/* 输出

.....................................................................................
--------------------------------50**************************************************************
----------------30******************************60****************************
--------20*************40*************55*************65*************
----#!******#!******#!******#!******#!******56******#!******#!******
.....................................................................................

查找：day10.查找.BinaryTreeSearch$Node@610455d6
max : 65
min : 20

层序遍历
50 30 60 20 40 55 65 56

```

```
前序遍历
50 20 30 40 55 56 60 65

后序遍历
20 30 40 55 56 60 65 50

中序遍历
20 30 40 50 55 56 60 65

.....................................................................................
--------------------------------50**************************************************************
----------------30******************************60****************************
--------20**************40**************55**************65**************
----#!******#!******#!******#!******#!******56******#!******#!******
.....................................................................................
删除40
.....................................................................................
--------------------------------50**************************************************************
----------------30******************************60****************************
--------20**************#!**************55**************65**************
----#!******#!******#!******#!******#!******56******#!******#!******
.....................................................................................
删除60
.....................................................................................
--------------------------------50**************************************************************
----------------30******************************56****************************
--------20**************#!**************55**************65**************
.....................................................................................

*/
```

总结

1. 二叉树插入的顺序决定最终树的存储

```

插入顺序：50、65、30、60、20、40、55
结果：
--------------------------------50**************************************************************
----------------30******************************65****************************
--------20**************40**************60**************#!**************
----#!******#!******#!******#!******55******#!******#!******#!******


插入顺序：50、30、60、20、40、65、55
结果：
.....................................................................................
--------------------------------50**************************************************************
----------------30******************************60****************************
--------20**************40**************55**************65**************
.....................................................................................
```

2. 理想情况下的插入、查找时间复杂度是log2(N)，最坏情况下时间复杂度回归到线性表 O(n)