数据结构与算法
实验报告

第三次

姓名	代珉玥
班级	软件 001 班
学号	2205223077
电话	18585038226
Email	2040257842@qq. com
日期	2020-10-17

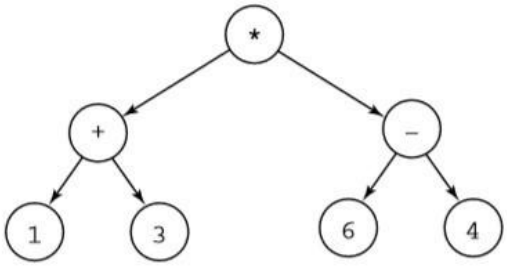
目录

实验 1

1.题目

二叉树可以用来表达一个算术表达式（当然也可以表达只要是二元运算的任何一种表达式）：叶结点储存操作数，分支结点储存操作符。括号这种运算符在二叉树中完全不需要，因为运算的先后顺序已经由二叉树的层次结构确定好了。更奇妙的是：当你对这个二叉树进行一次遍历就相当于对这个二叉树所表达的表达式完成了计算。

下面的表达式树表达了算术表达式：（1+3）*（6-4）



结合二叉树的遍历操作，对上面的表达式树进行先序遍历、中序遍历和后序遍历分别得到前缀表达式、中缀表达式和后缀表达式：

先序遍历（前缀表达式）：*+13-64

中序遍历（中缀表达式）：（（1+3）*（6-4））

后序遍历（后缀表达式）：13+64-*

一个表达式的计算也可以蕴含在表达式的遍历过程中，下面我们就一步步完成这个功能。

（中缀表达式的运算符具有优先级和结合性，为了不丢失这些信息，在中序遍历时为每一个子树都增加了一对圆括弧。）

任务 1

参照教材上的二叉树的抽象数据类型定义，为表达式树定义合适的数据成员和基础的成员方法。

任务 2

为任务 1 中创建的二叉树数据类型定义方法：readPostBinaryTree(String expression)。这个方法基于 expression 创建一棵二叉树，expression 是一个后缀算术表达式。具体的构建过程参看附录 1。

任务 3

在任务 2 的基础上实现一个成员方法：postorderResult()，该方法主要完成如下任务：

- 对表达式树进行后序遍历从而生成一个后缀表达式，结点和结点之间用空格分隔；
- 计算该表达式树，给出最终的运算结果；
  - 提示：计算表达式可以利用后序遍历的递归框架实现。
- 输出的格式如下：如果使用题头对应的那棵表达式树的话，应该为：

1 3 + 6 4 - * = 8

任务 4

为表达式树定义一个 output()函数，该函数将表达式树的层次结构以 2D 布局的方式输出。

举例：

后缀表达式为： d e a - + a b + c * *

output 函数的运行结果为：

任务 5

为表达式树定义一个 readInBinaryTree(String expression)，这个方法基于 expression 创建一棵二叉树，expression 是一个中缀算术表达式。

提示：可以使用栈转换算数表达式完成中缀转换成后缀表达式，转换的同时构建表达式树。

2. 数据结构设计

树的数据结构：

```
public class Node implements MyTree {
    private Operation operation = Operation.NULL;
    private int vulue;
    private Node leftChild;
    private Node rightChild;
    private int hight;
    private int depth;
}
```

树的接口（部分成员方法）：

```
public interface MyTree {
    boolean isLeaf();

    Node readPostBinaryTree(String expression);

    int postorderResult();

    void output();

    Node readInBinaryTree(String expression);

    boolean isNumber();

    int hight();

    void updateDepth();
}
```

枚举运算符：

```
public enum Operation {
    ADDITION,SUBTRACTION,MULTIPLICATION,DIVISION,
    LEFTBRACKET,RIGHTBRACKET,NULL
}
```

栈的数据结构：

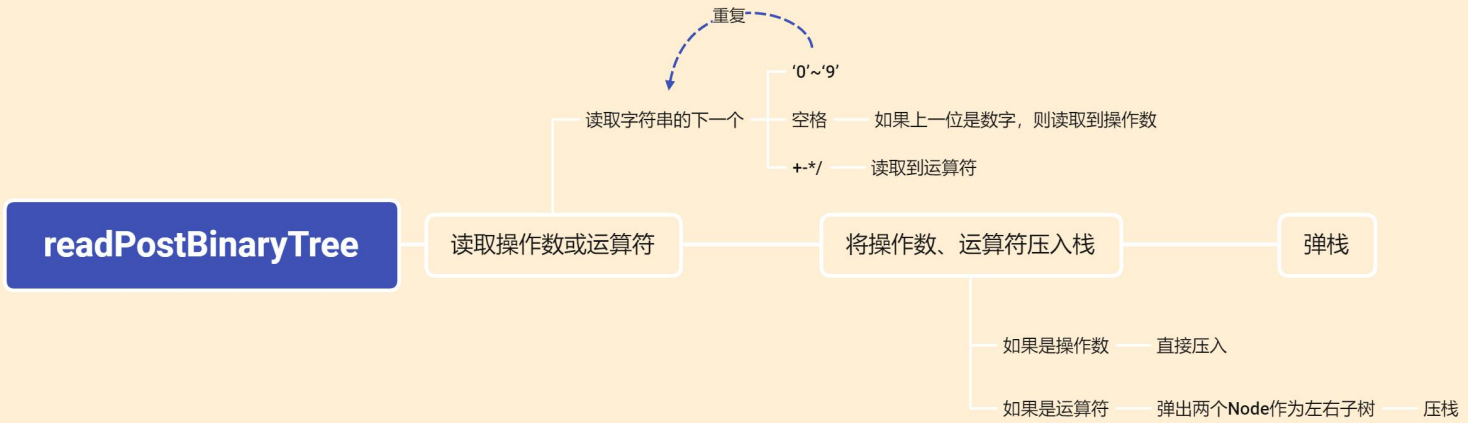
```
public class MyStack implements MyStackInterface {
    Node[] elements = new Node[100000];
    int top = 0;       //当前栈顶的上方
}
```

栈的接口（部分成员方法）：

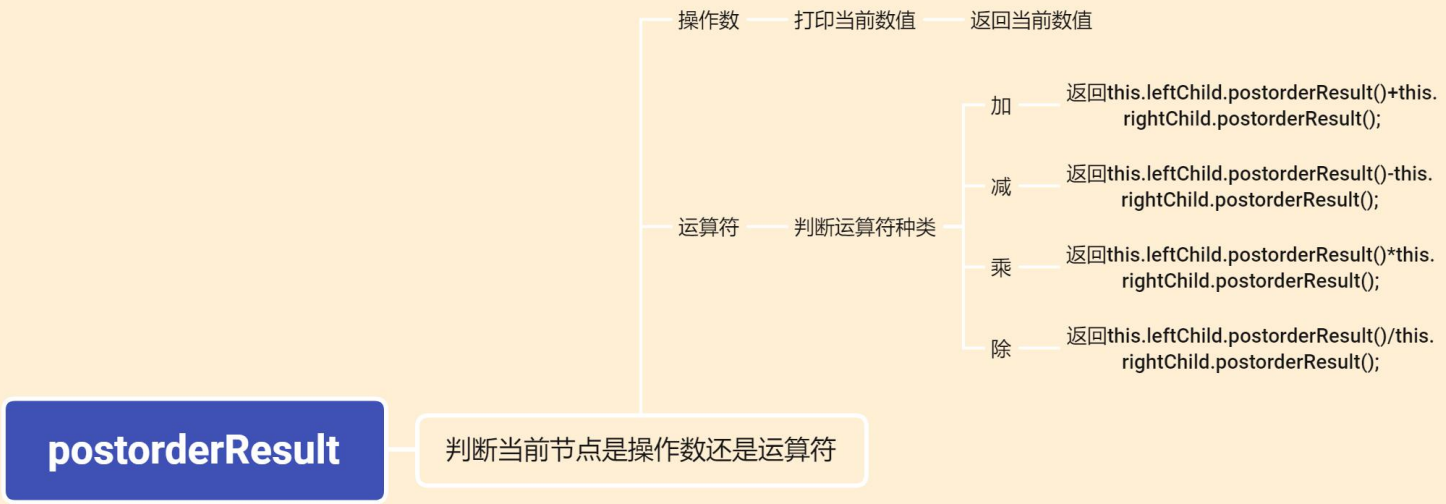
```
public interface MyStackInterface {
    boolean isFull();
    boolean isEmpty();
    Node pop();
    void push(Node node);
}
```

3. 算法设计

任务 2



任务 3



任务 4


output
创建当前树的数组储存形式
设置数组的首元素
计算数组长度
为数组剩余元素赋值
坐标为奇数
赋值为父节点的左子树
坐标为偶数
赋值为父节点的右子树
逐行打印
打印空格数
打印当前行首元素
打印空格数
未到达行尾
打印后继元素
更新空格数

任务 5


readInBinaryTree
读取操作数或运算符
重复
读取字符串的下一个
'0'~'9'
空格
如果上一位是数字，则读取到操作数
+-*/()
读取到运算符
将操作数、运算符压入栈
如果运算符栈不空
从运算符中弹出元素，并从操作数中弹出两个节点作为其左右子树，将该节点压入操作数中
弹出运算符栈
返回从操作数栈弹出的元素
如果是操作数
压入操作数栈
如果是运算符
如果是括号
如果是左括号
压入运算符栈
未弹出左括号
如果是右括号
从运算符中弹出元素，并从操作数中弹出两个节点作为其左右子树，将该节点压入操作数中
压入运算符栈
如果是+-*/
如果运算符栈空
压入运算符栈
如果运算符栈不空
如果当前运算优先级小于等于运算符栈栈顶元素
从运算符中弹出元素，并从操作数中弹出两个节点作为其左右子树，将该节点压入操作数中
压入运算符栈
如果当前运算优先级大于运算符栈栈顶元素
压入运算符栈

4. 主干代码说明

任务 2

```
public Node readPostBinaryTree(String expression) { //后缀表达式
	MyStack myStack = new MyStack();
	int tempNumber = 0;
	for (int i = 0; i < expression.length(); i++) {
		int tempCharNumber = (int) expression.charAt(i);
		if (tempCharNumber == (int)' ') {
			if ((int) expression.charAt(i-1)>=(int)'0'&&(int) expression.charAt(i-1)<=(int)'9') {
				Node tempNode = new Node(tempNumber);
				myStack.push(tempNode);
				tempNumber = 0;
			}
		} else if (tempCharNumber>=(int)'0'&&tempCharNumber<=(int)'9'){
			tempNumber*=10;
			tempNumber+=tempCharNumber-(int)'0';
		}else {
			Node tempNode = new Node((char) tempCharNumber);
			tempNode.rightChild=myStack.pop();
			tempNode.leftChild=myStack.pop();
			myStack.push(tempNode);
			tempNumber=0;
		}
	}
	return myStack.pop();
}
```

任务 3

```
public int postorderResult() {
	if (this.isNumber()) {
		System.out.print(vulue+" ");
		return vulue;
	}
	int result = 0;
	switch (operation){
		case ADDITION:
			result= this.leftChild.postorderResult()+this.rightChild.postorderResult();
			System.out.print("+ ");
			break;
		case SUBTRACTION:
			result= this.leftChild.postorderResult()-this.rightChild.postorderResult();
			System.out.print("- ");
			break;
		case MULTIPLICATION:
			result = this.leftChild.postorderResult()*this.rightChild.postorderResult();
			System.out.print("* ");
			break;
		default:
			result = this.leftChild.postorderResult()/this.rightChild.postorderResult();
```

```
                System.out.print("/ ");
        }
        return result;
}
```

任务 4

```
public void output() {
        Node[] elements = new Node[10000];
        elements[0]=this;
        int length=1;
        for (int i = 0; i <= this.hight(); i++) {
                length *= 2;
        }
        for (int i = 1; i <= length; i++) {
                if (i % 2 == 0 && elements[i / 2-1] != null) {
                        elements[i] = elements[i / 2-1].rightChild;
                }else if (i % 2 == 1 && elements[i / 2] != null) {
                        elements[i] = elements[i / 2].leftChild;
                }
        }
        int blankNum = (length%2+length)/2;
        int newlineindex = 0;
        int index = 0;
        boolean isFirst=true;
        while (index<=length-2){
                if (isFirst){
                        for (int i = 0; i < blankNum; i++) {        //打印元素前的空格
                                System.out.print(" ");
                        }
                        isFirst=false;
                }else {
                        for (int i = 0; i < blankNum * 2 - 1; i++) {        //打印元素前的空格
                                System.out.print(" ");
                        }
                }
                if (elements[index]==null){
                        System.out.print("o");
                }else {
                        System.out.print(elements[index]);
                }
                if (index == newlineindex) {
                        System.out.println();
                        newlineindex = (newlineindex + 2) * 2 - 2;
                        blankNum /= 2;
                        isFirst = true;
                }
                index++;
        }
}
```

任务 5

```
public Node readInBinaryTree(String expression) {//中缀表达式
    MyStack numberStack = new MyStack();
    MyStack operationStack = new MyStack();
    int tempNumber = 0;
    for (int i = 0; i < expression.length(); i++) {
        int tempCharNumber = (int) expression.charAt(i);
        if (tempCharNumber == (int) ' ') {
            if ((int) expression.charAt(i - 1) >= (int) '0' && (int) expression.charAt(i - 1) <= (int) '9') {
                Node tempNode = new Node(tempNumber);
                numberStack.push(tempNode);
                tempNumber = 0;
            }
        } else if (tempCharNumber >= (int) '0' && tempCharNumber <= (int) '9') {
            tempNumber *= 10;
            tempNumber += tempCharNumber - (int) '0';
        } else {
            Node tempNode = new Node((char) tempCharNumber);
            if (tempNode.operation == Operation.LEFTBRACKET) {
                operationStack.push(tempNode);
            } else if (tempNode.operation == Operation.RIGHTBRACKET) {
                Node tempNumberNode = operationStack.pop();
                while (tempNumberNode.operation != Operation.LEFTBRACKET) {
                    tempNumberNode.rightChild = numberStack.pop();
                    tempNumberNode.leftChild = numberStack.pop();
                    numberStack.push(tempNumberNode);
                    tempNumberNode = operationStack.pop();
                }
            } else {
                if (!operationStack.isEmpty()) {
                    if (tempNode.compareTo(operationStack.elements[operationStack.top - 1]) <= 0) {
                        Node tempNumberNode = operationStack.pop();
                        tempNumberNode.rightChild = numberStack.pop();
                        tempNumberNode.leftChild = numberStack.pop();
                        numberStack.push(tempNumberNode);
                    }
                }
                operationStack.push(tempNode);
            }
            tempNumber = 0;
        }
    }
    while (!operationStack.isEmpty()) {
        Node tempNumberNode = operationStack.pop();
        tempNumberNode.rightChild = numberStack.pop();
        tempNumberNode.leftChild = numberStack.pop();
        numberStack.push(tempNumberNode);
    }
    return numberStack.pop();
}
```

5. 运行结果展示

过程性结果

```
    public static void main(String[] args) {
        Node tree = new Node();
        tree = tree.readPostBinaryTree("1 3 + 6 14 - * ");
        System.out.println("= "+tree.postorderResult());
        tree.output();
        tree = tree.readInBinaryTree(" (1+3) * (6-4) ) ");
        System.out.println("= "+tree.postorderResult());
        tree.output();
    }
}
```

```
Main
D:\Javaaaa\ruanjian\bin\java.exe "-javaagent:D:\Javaaaa\ruanjian\idealIU1\IntelliJ IDEA 2021.2\lib\idea_rt
 .jar=60923:D:\Javaaaa\ruanjian\idealIU1\IntelliJ IDEA 2021.2\bin" -Dfile.encoding=UTF-8 -classpath
 D:\Javaaaa\数据结构与算法第三次作业\experiment01\out\production\experiment01 com.company.Main
1 3 + 6 14 - * = -32
    *
  +   -
 1 3 6 14
0 = 0
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "com.company.Node.hight()" because "this.leftChild" is null
    at com.company.Node.hight(Node.java:259)
    at com.company.Node.output(Node.java:156)
    at com.company.Main.main(Main.java:12)

Process finished with exit code 1
```

多打了一个小括号，且括号为中文括号。
修正后：

```
    public static void main(String[] args) {
        Node tree = new Node();
        tree = tree.readPostBinaryTree("1 3 + 6 14 - * ");
        System.out.println("= "+tree.postorderResult());
        tree.output();
        tree = tree.readInBinaryTree("(1+3)*(6-4)");
        System.out.println("= "+tree.postorderResult());
        tree.output();
    }
}
```

```
Main
D:\Javaaaa\ruanjian\bin\java.exe "-javaagent:D:\Javaaaa\ruanjian\idealIU1\IntelliJ IDEA 2021.2\lib\idea_rt
 .jar=60969:D:\Javaaaa\ruanjian\idealIU1\IntelliJ IDEA 2021.2\bin" -Dfile.encoding=UTF-8 -classpath
 D:\Javaaaa\数据结构与算法第三次作业\experiment01\out\production\experiment01 com.company.Main
1 3 + 6 14 - * = -32
    *
  +   -
 1 3 6 14
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "com.company.Node.postorderResult()" because "this
 .leftChild" is null
    at com.company.Node.postorderResult(Node.java:141)
    at com.company.Main.main(Main.java:11)

Process finished with exit code 1
```

在编程中设定了数字由空格结尾，数字的结尾应当有空格。

最终结果

```
    public static void main(String[] args) {
        Node tree = new Node();
        tree = tree.readPostBinaryTree("1 3 + 6 14 - * ");
        System.out.println("= "+tree.postorderResult());
        tree.output();
        tree = tree.readInBinaryTree("( 1 + 3 ) * ( 6 - 4 )");
        System.out.println("= "+tree.postorderResult());
        tree.output();
    }
}
```

```
Main
D:\Javaaaa\ruanjian\bin\java.exe "-javaagent:D:\Javaaaa\ruanjian\idealIU1\IntelliJ IDEA 2021.2\lib\idea_rt
 .jar=61092:D:\Javaaaa\ruanjian\idealIU1\IntelliJ IDEA 2021.2\bin" -Dfile.encoding=UTF-8 -classpath
 D:\Javaaaa\数据结构与算法第三次作业\experiment01\out\production\experiment01 com.company.Main
1 3 + 6 14 - * = -32
    *
  +   -
 1 3 6 14
1 3 + 6 4 - * = 8
    *
  +   -
 1 3 6 4

Process finished with exit code 0
```

6. 总结和收获

在编写程序时，对于输入格式有一定的要求，在测试时，应注意输入格式是否符合程序的设定；同时，应当对输入格式的限制尽可能的小，使得程序的拓展性更好。

7. 参考文献

无。

实验 2

8.题目

BST 这种数据组织形式，使得查找、删除、插入的运算时间在平均情形下会得到 O(logN)的时间性能。但是在实际的应用中，往往会存在大量有序的数据，这时 BST 会退化，如何避免这种退化造成性能上的衰退？AVL 和红黑树都是比较好的解决方案，这两种二叉树的平衡性是严格的，稳定性表现得十分出色。AVL 和红黑树都是时间效率很高的经典算法，在许多专业的应用领域（如 STL、JavaAPI）有着十分重要的地位。然而 AVL 和红黑树的编程实现难度比较大，编程复杂，故在某些 场合下(比如现场类的信息类竞赛中)不太受欢迎。本题的目的让大家掌握一种平衡性也可以得到保证的实现较简单的二叉树：Treap。

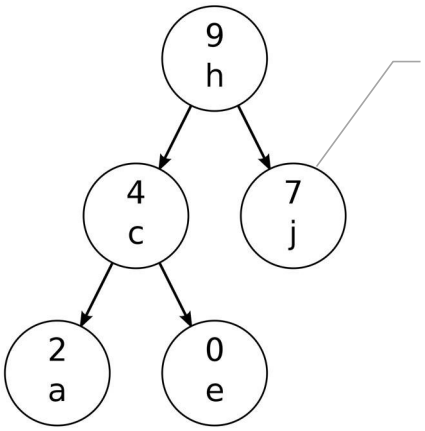
Treap 是一种平衡树，这个单词的构造选取了 Tree 的前两个字符和 Heap 的后三个字符。可以想见，Treap 把 BST 和 Heap 结合了起来。它具有 BST 在较好情形下的时间性能，引入堆就是为了帮助可能退化的 BST 维持树的平衡性，让树的高度增长速度控制在 O(logN)中。

Treap 在 BST 的基础上，为每一个结点添加了一个 priority 属性（这个值是随机生成的）。也就是说 Treap 的结点包含两个值：key 和 priority。结点中 key 值满足 BST 的性质，结点的 priority 属性满足最大值堆性质（或者最小值堆）。Treap 可以定义为满足如下性质的二叉树：

任意结点，若其左子树不空，则左子树上所有结点的 key 均小于该结点的 key，而且它的 priority 值大于等于左子树根节点的 priority；

任意结点，若其右子树不空，则右子树上所有结点的 key 均不小于该结点的 key，而且它的 priority 值大于等于右子树根节点的 priority；

任意结点的左右子树也分别为 Treap。下图所示的二叉树即为 Treap：



结点中的数字即为 Priority 属性，字符即为 key 属性，Priority 值与 key 的值没有任何关

BST 会遇到不平衡的原因是因为有序的数据会使查找的路径退化成单链，而随机序列的数据使 BST 退化是小概率事件。Treap 能够保证二叉树的平衡，根本原因就在于 Priority 值的引入使得树的结构不仅仅取决于结点的 key 值，还取决于 Priority 值。而 Priority 值是随机产生的，由于有序的随机序列是小概率事件，故 Treap 的结构是趋向于随机平衡的。

关于 Treap 的详细介绍，可以参看附件 treap.pdf。

任务 1

实现 Treap 数据结构，具有插入、删除、查找等基本操作。

任务 2

实现普通的 BST，在此基础上，按照如下表格统计 BST 和 Treap 的相应运行数据，根据这些数据,写出总结。

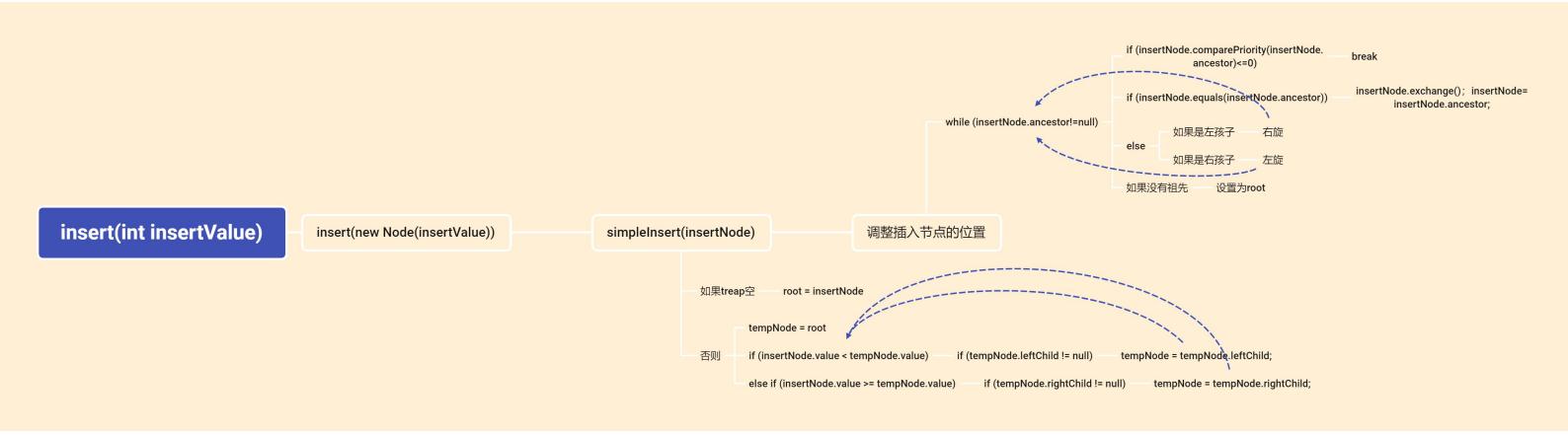
		有序序列数据（数据规模）			随机序列数据（数据规模）		
		1000	10000	100000	1000	10000	100000
BST	插入所有元素时间总计						
	树高						
	查找所有元素时间总计						
	删除所有元素时间总计						
Treap	插入所有元素时间总计						
	树高						
	查找所有元素时间总计						
	删除所有元素时间总计						

9. 数据结构设计

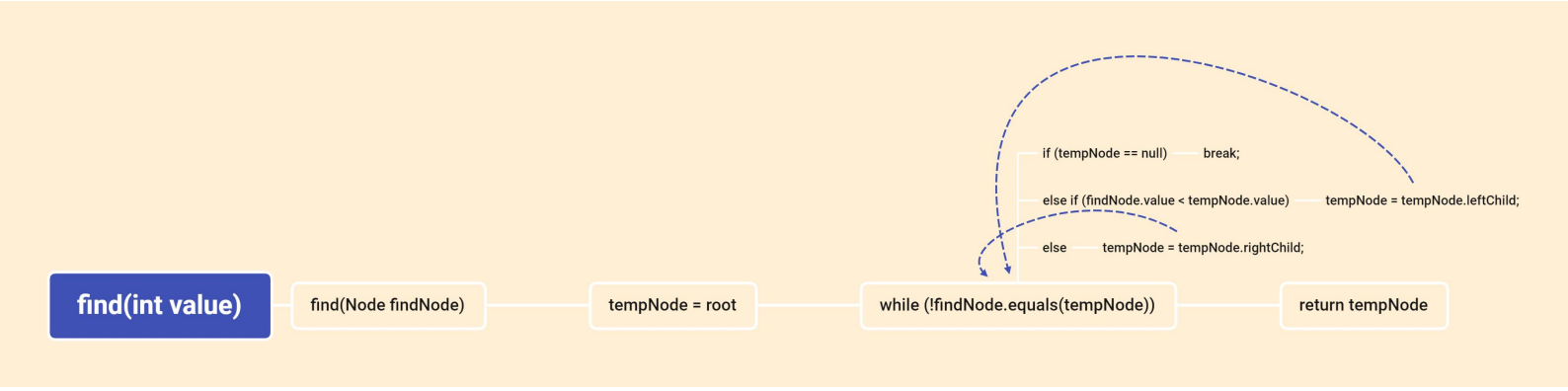
```
public interface MyTreeInterface {
    void insert(int insertValue);
    void delete(int deleteValue);
    Node find(int value);
    Node find(Node findNode);
}

public class Node {
    int value;
    int priority;
    Node ancestor;
    Node leftChild;
    Node rightChild;
    int hight;
}

public class MyBST implements MyTreeInterface{
    Node root;
}

public class MYTreap implements MyTreeInterface {
    Node root;
    }
```

10. 算法设计

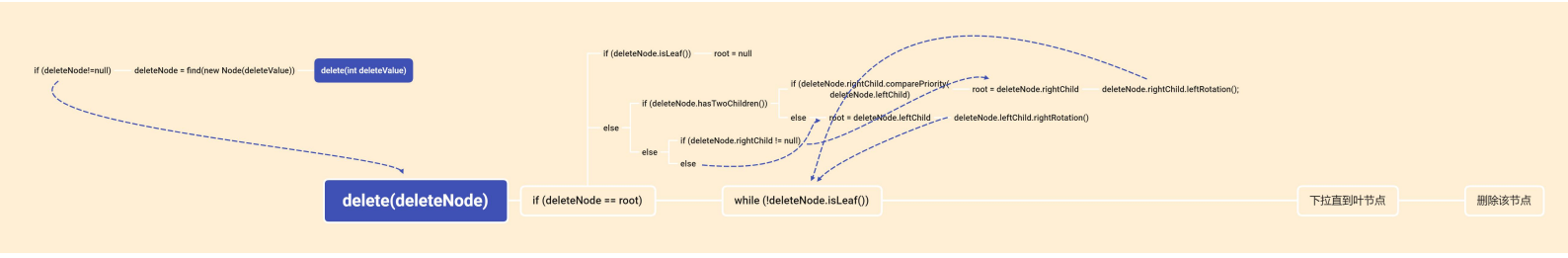
插入



查找



删除



11. 主干代码说明

插入

```
public void insert(int insertValue) {
    insert(new Node(insertValue));
}

public void insert(Node insertNode){
    simpleInsert(insertNode);
    while (insertNode.ancestor!=null){
        if (insertNode.comparePriority(insertNode.ancestor)<=0){
            break;
        }
        if (insertNode.equals(insertNode.ancestor)){
            insertNode.exchange();
            insertNode=insertNode.ancestor;
        }else {
            if (insertNode.isLeftChild()) {
                insertNode.rightRotation();
            } else if (insertNode.isRightChild()) {
                insertNode.leftRotation();
            }
        }
    }
    if (insertNode.ancestor==null){
        root = insertNode;
    }
}

public void simpleInsert(Node insertNode) {
    if (root == null) {
        root = insertNode;
        return;
    }
    Node tempNode = root;
    while (true) {
        if (insertNode.value < tempNode.value) {
            if (tempNode.leftChild != null) {
                tempNode = tempNode.leftChild;
            } else {
                tempNode.setLeftChild(insertNode);
                break;
            }
```

```
            } else if (insertNode.value >= tempNode.value) {
                if (tempNode.rightChild != null) {
                    tempNode = tempNode.rightChild;
                } else {
                    tempNode.setRightChild(insertNode);
                    break;
                }
            }
        }
    }
```

查找

```
public Node find(int value) {
    return find(new Node(value));
}


public Node find(Node findNode) {
    Node tempNode = root;
    while (!findNode.equals(tempNode)) {
        if (tempNode == null) {
            break;
        } else if (findNode.value < tempNode.value) {
            tempNode = tempNode.leftChild;
        } else {
            tempNode = tempNode.rightChild;
        }
    }
    return tempNode;
}
```

删除

```
public void delete(int deleteValue) {
    Node deleteNode = find(new Node(deleteValue));
    if (deleteNode!=null) {
        delete(deleteNode);
    }
}

public void delete(Node deleteNode) {
    if (deleteNode == root) {
```

```
            if (deleteNode.isLeaf()) {
                root = null;
            } else {
                if (deleteNode.hasTwoChildren()) {
                    if (deleteNode.rightChild.comparePriority(deleteNode.leftChild) >= 0) {
                        root = deleteNode.rightChild;
                        deleteNode.rightChild.leftRotation();
                    } else {
                        root = deleteNode.leftChild;
                        deleteNode.leftChild.rightRotation();
                    }
                } else {
                    if (deleteNode.rightChild != null) {
                        root = deleteNode.rightChild;
                        deleteNode.rightChild.leftRotation();
                    } else {
                        root = deleteNode.leftChild;
                        deleteNode.leftChild.rightRotation();
                    }
                }
                delete(deleteNode);
            }
        } else {
            while (!deleteNode.isLeaf()) {
                if (deleteNode.hasTwoChildren()) {
                    if (deleteNode.rightChild.priority - deleteNode.leftChild.priority >= 0) {
                        deleteNode.rightChild.leftRotation();
                    } else {
                        deleteNode.leftChild.rightRotation();
                    }
                } else {
                    if (deleteNode.rightChild != null) {
                        deleteNode.rightChild.leftRotation();
                    } else if (deleteNode.leftChild != null) {
                        deleteNode.leftChild.rightRotation();
                    }
                }
            }
            if (deleteNode.isLeftChild()) {
                deleteNode.ancestor.setLeftChild(null);
            } else {
                deleteNode.ancestor.setRightChild(null);
            }
        }
    }
```

12. 运行结果展示

过程性结果

未考虑删除根节点时根节点只有一个子树或没有子树的情况。

最终结果

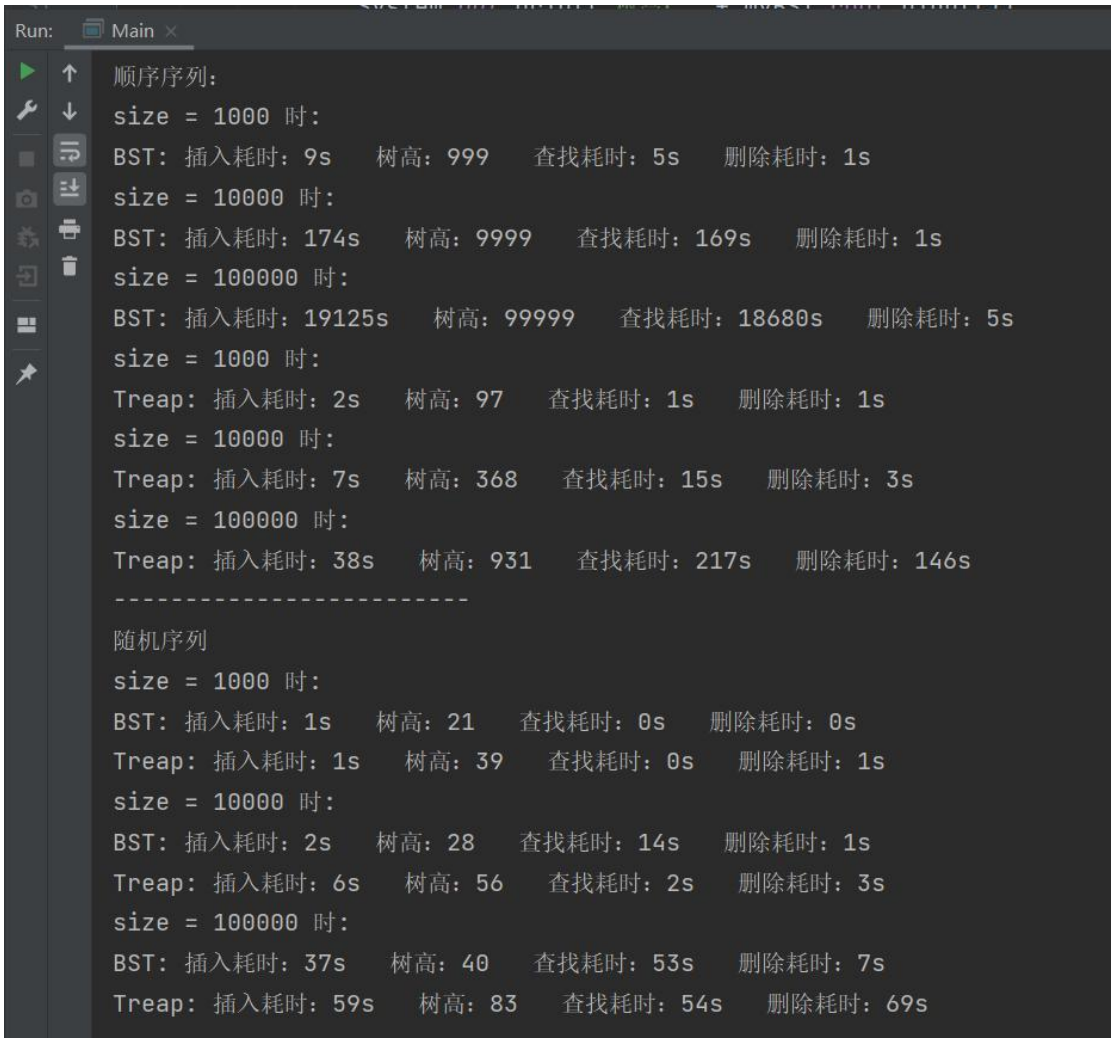
测试代码：

```
public static void main(String[] args) {
    MYTreap myTreap = new MYTreap();
    MyBST myBST = new MyBST();
    long startTime, endTime;
    int size = 1000;
    Random random = new Random();

    System.out.println("顺序序列：  ");

    for (size = 1000; size <= 100000; size *= 10) {
        System.out.println("size = " + size + "  时:");
        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myBST.insert(i);
        }
        endTime = System.currentTimeMillis();
        System.out.print("BST:  插入耗时：  " + (endTime - startTime) + "s      ");


        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myBST.find(i);
        }
        endTime = System.currentTimeMillis();
```

```
        System.out.print("查找耗时：" + (endTime - startTime) + "s      ");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myBST.delete(i);
        }
        endTime = System.currentTimeMillis();
        System.out.println("删除耗时：" + (endTime - startTime) + "s");
    }

    for (size = 1000; size <= 100000; size *= 10) {
        System.out.println("size = " + size + "  时:");
        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myTreap.insert(i);
        }
        endTime = System.currentTimeMillis();
        System.out.print("Treap: 插入耗时：" + (endTime - startTime) + "s      ");


        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myTreap.find(i);
        }
        endTime = System.currentTimeMillis();
        System.out.print("查找耗时：" + (endTime - startTime) + "s      ");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myTreap.delete(i);
        }
        endTime = System.currentTimeMillis();
        System.out.println("删除耗时：" + (endTime - startTime) + "s");
    }


    System.out.println("-------------------------\n 随机序列");

    for (size = 1000; size <= 100000; size *= 10) {
        System.out.println("size = " + size + "  时:");

        int[] elements = new int[size];
        for (int i = 0; i < size; i++) {
            elements[i] = random.nextInt(100000);
        }
```

```
        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myBST.insert(elements[i]);
        }
        endTime = System.currentTimeMillis();
        System.out.print("BST: 插入耗时: " + (endTime - startTime) + "s      ");


        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myBST.find(elements[i]);
        }
        endTime = System.currentTimeMillis();
        System.out.print("查找耗时: " + (endTime - startTime) + "s      ");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myBST.delete(elements[i]);
        }
        endTime = System.currentTimeMillis();
        System.out.println("删除耗时: " + (endTime - startTime) + "s");


        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myTreap.insert(elements[i]);
        }
        endTime = System.currentTimeMillis();
        System.out.print("Treap: 插入耗时: " + (endTime - startTime) + "s      ");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myTreap.find(elements[i]);
        }
        endTime = System.currentTimeMillis();
        System.out.print("查找耗时: " + (endTime - startTime) + "s      ");

        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            myTreap.delete(elements[i]);
        }
        endTime = System.currentTimeMillis();
        System.out.println("删除耗时: " + (endTime - startTime) + "s");
    }
}
```

结果：



		有序序列数据（数据规模）			随机序列数据（数据规模）		
		1000	10000	100000	1000	10000	100000
BST	插入所有元素时间总计	9	174	19125	1	2	37
	树高	999	9999	99999	21	28	40
	查找所有元素时间总计	5	169	18680	0	14	53
	删除所有元素时间总计	1	1	5	0	1	7
Treap	插入所有元素时间总计	2	7	38	1	6	59
	树高	97	368	931	39	56	83
	查找所有元素时间总计	1	15	217	0	2	54
	删除所有元素时间总计	1	3	146	1	3	69

13. 总结和收获

在这次实验中，测试时顺序数据和小规模数据均没有问题，而在大规模随机数据中抛出了空指针异常。经过检查之后发现，在插入和删除相同元素的时候出现了问题。在今后的程序编写中，应当注意特殊情况（例如本题中的插入和删除相同元素）。

14. 参考文献

无。