

数据结构与算法 作业报告

第三次



姓名 曹家豪

班级 软件 2204 班

学号 2226114017

电话 13572763245

Email caojiahao@xjtu. stu. edu. cn

日期 2024-2-25

目录

| | |
|------------------------------|----|
| 实验 1 实现 BST 数据结构..... | 1 |
| 题目 | 1 |
| 数据设计 | 1 |
| 算法设计 | 3 |
| 主干代码说明 | 4 |
| 运行结果展示 | 9 |
| 总结和收获 | 12 |
| 实验 2 使用 BST 为文稿建立单词索引表 | 13 |
| 题目 | 13 |
| 数据设计 | 13 |
| 算法设计 | 14 |
| 主干代码说明 | 14 |
| 运行结果展示 | 15 |
| 总结和收获 | 17 |
| 实验 3 学会使用堆 | 17 |
| 题目 | 17 |
| 数据设计 | 18 |
| 算法设计 | 19 |
| 主干代码说明 | 20 |
| 运行结果展示 | 23 |
| 总结和收获 | 23 |
| 实验 3 学会使用堆 (3) | 24 |
| 数据设计 | 24 |
| 算法设计 | 24 |
| 主干代码说明 | 26 |
| 运行结果展示 | 28 |
| 总结和收获 | 29 |
| 附录 | 29 |

实验 1 实现 BST 数据结构

题目

二叉检索树即 BST，是利用二叉树的非线性关系，结合数据之间的大小关系进行存储的一种用于检索数据的数据结构。一般情况下，对信息进行检索时，都需要指定检索关键码 (Key)，根据该关键字找到所需要的信息（比如学生信息里关键码是学号，而姓名等信息就是该关键码对应的信息），所以当提到检索时，都会有“键值对”这个概念，用 (key, value) 表示键值和对应信息的关系。

下面的二叉检索树的 ADT 在描述时，依然使用了模板类的方法，并且在这次描述中，使用了两个模板参数 K 和 V（这表明 key 和 value 可以为不同类型的数据）。在本次任务的测试文件中，key 和 value 都是 String 类型，对模板参数运用有困难的同学，可以直接设定 key 和 value 的类型为 String。

但是请注意：任务 2 的 BST 的 key 和 value 存放的数据类型不同，因此如果在本任务中没有使用模板类构建 BST，那么在任务 2 中则需要将任务 1 中的代码进行重新撰写，这个重写过程应该也能让不使用模板类的同学发现代码复用率的下降。

数据设计

在实现二叉搜索树 (BST) 的数据结构时，我们采用 Java 的泛型机制来设计 BST 类，旨在提供一个通用且灵活的数据存储结构，支持不同类型的键 (Key) 和值 (Value)。这种设计方法不仅有助于代码复用，还能保证数据结构的严格类

型安全，从而避免运行时的类型错误。

Node 类：

每个 Node 实例包含一个键（Key key）、一个值（Value val）以及两个子节点引用（Node left, right）。这些成员变量是构建 BST 的基础。

Key：作为节点的唯一标识，Key 需要实现 Comparable 接口，以支持节点间的排序。这是实现 BST 中元素有序排列的关键，确保了 BST 的搜索、插入和删除操作能够按照预定的逻辑高效执行。

left, right：指向当前节点的左右子树，这种链接构造支持了树结构的递归定义，使得每个节点都可以视为一个独立的子树。

BST 类：

根节点引用（root）：指向 BST 的根节点，是所有操作的起点。通过递归地操作这个根节点及其子节点，BST 类实现了 BST 的所有基本功能，如插入、搜索和删除等。

关键方法：

insert(Key key, Value val)：利用键的有序性质，递归地在正确的位置插入新节点或更新现有节点的值，维护了 BST 的结构和有序性。

remove(Key key)：通过键来定位并删除相应的节点，同时确保树的连续性和有序性不被破坏，需要在删除节点后正确地重新链接其子树。

search(Key key)：依据键的有序性，递归地在树中高效查找目标键，展示了 BST 作为一种搜索树的优势。

update(Key key, Value val)：对已存在的节点进行值的更新。

printInorder()：通过中序遍历，按照键的升序打印所有节点，直观展示了 BST 键

的有序排列。

这种数据设计使 BST 能够有效地支持动态数据集合的高效操作，包括查找、插入、删除和遍历，适用于需要快速数据检索和维护有序数据集合的场景。通过泛型的使用，BST 不仅在编译时提供了类型安全保证，也提高了数据结构的复用性和适用范围，能够满足更广泛的应用需求。

算法设计

实验核心在于实现一个通用的二叉搜索树（BST），包括插入、删除、搜索和更新节点的功能。下面着重分析关键步骤的作用和实现方式：

1. **insert** 方法（在 BST 中插入一个新的键值对。如果键已存在，则更新对应的值）：

(1)递归查找插入位置：从根节点开始，根据键值的大小递归地向左或右子树进发，直到找到一个空位置插入新节点。

(2)节点插入：创建一个新的 **Node** 对象，并将其插入到找到的空位置。如果键已存在，更新该节点的值。

(3)维护树的结构：通过递归返回的方式，更新父节点的左或右子节点引用，确保树的结构完整性。

2.**remove** 方法（删除 BST 中指定键的节点，并保持 BST 的有序性不受影响）：

(1)定位节点：递归遍历树，根据键值比较结果，找到需要删除的节点。

(2)处理三种情况：

①无子节点：直接删除该节点，返回 `null` 给父节点的相应子节点引用。

②单子节点：删除该节点，并用其子节点替代自己的位置，返回给父节点的相应子节点引用。

③两个子节点：找到被删除节点的右子树中的最小节点（后继节点），用它来替换被删除节点的位置，并递归删除那个最小节点。

(3)重新链接：确保所有子树正确链接，维护 BST 的结构。

3. `search` 方法（根据给定的键，在 BST 中查找并返回相应的值）：

(1)递归搜索：从根节点开始，递归地在左或右子树中搜索给定的键。

(2)比较并决策：在每个节点处，将给定的键与节点的键进行比较，根据比较结果决定向左还是向右继续搜索。

(3)返回结果：如果找到了匹配的键，返回对应的值；如果搜索到叶子节点仍未找到，则返回 `null`。

3. `update` 方法（更新 BST 中已存在节点的值）：

(1)查找节点：使用与 `search` 方法相似的逻辑，找到具有指定键的节点。

(2)更新值：如果找到该节点，则更新其存储的值。

通过这些算法设计，BST 能够高效地支持数据动态的操作，包括数据的插入、删除、查找和更新，满足了广泛的应用需求。

主干代码说明

算法设计已经从语言角度描述了各方法的实现逻辑, 下面选取几个方法从代码角度阐述其实现细节。

Insert ():

```
1.      // 插入键值对
2.      public void insert(Key key, Value val) {
3.          root = insertRec(root, key, val);
4.      }
5.
6.      // 递归插入节点
7.      private Node insertRec(Node node, Key key, Value val) {
8.          // 如果节点为空, 创建新节点
9.          if (node == null) return new Node(key, val);
10.
11.         int cmp = key.compareTo(node.key);
12.         // 如果键小于当前节点的键, 插入左子树
13.         if (cmp < 0) node.left = insertRec(node.left, key,
            val);
14.         // 如果键大于当前节点的键, 插入右子树
15.         else if (cmp > 0) node.right = insertRec(node.right
            , key, val);
16.         // 如果键等于当前节点的键, 更新当前节点的值
17.         else node.val = val;
18.
19.         return node;
20.     }
```

remove () :

```
1.      // 删除节点并返回被删除节点的值
2.      public Value remove(Key key) {
3.          List<Node> removedNodeWrapper = new ArrayList<>(1);
4.          root = removeNode(root, key, removedNodeWrapper);
5.          return removedNodeWrapper.isEmpty() ? null : remove
            dNodeWrapper.get(0).val;
6.      }
7.
8.      // 递归删除节点
9.      private Node removeNode(Node node, Key key, List<Node>
            removedNodeWrapper) {
10.         // 如果节点为空, 返回空
11.         if (node == null) {
12.             return null;
13.         }
```

```

14.
15.         int cmp = key.compareTo(node.key);
16.         // 如果要删除的键小于当前节点的键，递归删除左子树
17.         if (cmp < 0) {
18.             node.left = removeNode(node.left, key, removedN
odeWrapper);
19.         }
20.         // 如果要删除的键大于当前节点的键，递归删除右子树
21.         else if (cmp > 0) {
22.             node.right = removeNode(node.right, key, remove
dNodeWrapper);
23.         }
24.         // 如果找到要删除的节点
25.         else {
26.             // 记录被删除的节点
27.             removedNodeWrapper.add(node);
28.
29.             // 如果左子树为空，返回右子树
30.             if (node.left == null) return node.right;
31.             // 如果右子树为空，返回左子树
32.             if (node.right == null) return node.left;
33.
34.             // 保存当前节点
35.             Node t = node;
36.             // 选取右子树中的最小节点替代当前节点
37.             node = min(t.right);
38.             // 删除右子树中的最小节点，并将其左子树连接到新节点
的右子树
39.             node.right = deleteMin(t.right);
40.             // 将当前节点的左子树连接到新节点的左子树
41.             node.left = t.left;
42.         }
43.         return node;
44.     }
45.
46.     // 找到最小节点
47.     private Node min(Node node) {
48.         while (node.left != null) node = node.left;
49.         return node;
50.     }
51.
52.     // 删除最小节点
53.     private Node deleteMin(Node node) {
54.         if (node.left == null) return node.right;
55.         node.left = deleteMin(node.left);

```



```
56.         return node;
57.     }
```

search():

```
1.         // 搜索键
2.     public Value search(Key key) {
3.         Node node = searchRec(root, key);
4.         return node == null ? null : node.val;
5.     }
6.
7.     // 递归搜索节点
8.     private Node searchRec(Node node, Key key) {
9.         // 如果节点为空, 返回空
10.        if (node == null) return null;
11.
12.        int cmp = key.compareTo(node.key);
13.        // 如果要搜索的键小于当前节点的键, 搜索左子树
14.        if (cmp < 0) return searchRec(node.left, key);
15.        // 如果要搜索的键大于当前节点的键, 搜索右子树
16.        else if (cmp > 0) return searchRec(node.right, key)
17.        ;
18.        // 如果键等于当前节点的键, 返回当前节点
19.        else return node;
20.    }
```

update():

```
1.         // 更新键对应的值
2.     public boolean update(Key key, Value val) {
3.         Node node = searchNode(root, key);
4.         // 如果节点为空, 表示未找到具有指定键的元素, 返回 false
5.         if (node == null) {
6.             return false;
7.         } else {
8.             // 找到了元素, 更新其值并返回 true
9.             node.val = val;
10.            return true;
11.        }
12.    }
13.
14.    // 递归搜索特定键的节点
15.    private Node searchNode(Node node, Key key) {
16.        // 如果节点为空, 返回空
17.        if (node == null) return null;
18.
19.        int cmp = key.compareTo(node.key);
```

```

20.      // 如果要搜索的键小于当前节点的键，搜索左子树
21.      if (cmp < 0) return searchNode(node.left, key);
22.      // 如果要搜索的键大于当前节点的键，搜索右子树
23.      else if (cmp > 0) return searchNode(node.right, key
    );
24.      // 如果键等于当前节点的键，返回当前节点
25.      else return node;
26.  }

```

size():

```

1.      // 计算节点数
2.      private int size(Node node) {
3.          if (node == null) return 0;
4.          // 递归计算左右子树的节点数，并加上当前节点
5.          return 1 + size(node.left) + size(node.right);
6.      }
7.
8.      // 计算树的高度
9.      private int height(Node node) {
10.         if (node == null) return 0;
11.         // 递归计算左右子树的高度，取较大值，并加上当前节点
12.         return 1 + Math.max(height(node.left), height(node.
    right));
13.     }

```

主程序:

```

1.      private static void processLine(String line, BST<String
    , String> bst) {
2.          if (line.startsWith("+(")) {
3.              //匹配符号（插入方法）
4.              String[] parts = line.substring(2, line.length(
    ) - 1).split(",");//去除括号
5.              String key = parts[0].substring(1, parts[0].len
    gth() - 1);
6.              String value = parts[1].substring(2, parts[1].l
    ength() - 2); // 移除引号
7.              bst.insert(key, value);
8.          } else if (line.startsWith("-(")) {
9.              //匹配方法（移除方法）
10.             String key = line.substring(3, line.length() -
    2);//去除括号
11.             String removedValue = bst.remove(key);
12.             System.out.println("remove " + (removedValue !=
    null ? "success" : "unsuccess")+"---
    "+key +" "+(removedValue != null ? removedValue : ""));

```

```

13.         } else if (line.startsWith("?(")) {
14. //匹配方法（搜索方法）
15.         String key = line.substring(3, line.length() -
16.         2); //去除括号
17.         String result = bst.search(key);
18.         System.out.println("search "+ (result != null ?
19.         "success" : "unsuccess")+"---"+key+" "+result);
20.         } else if (line.startsWith("(=")) {
21. //匹配方法（更新方法）
22.         String[] parts = line.substring(2, line.length(
23.         ) - 1).split(","); //去除括号
24.         String key = parts[0].substring(1, parts[0].len
25.         gth() - 1);
26.         String newValue = parts[1].substring(2, parts[1
27.         ].length() - 2); // 移除引号
28.         boolean updated = bst.update(key, newValue);
29.         System.out.println("update "+ (updated ? "succe
30.         ss" : "unsuccess")+"---"+ key+" "+ newValue);
31.         } else if (line.equals("#")) {
32.         bst.showStructure();
33.         }
34.     }

```

运行结果展示

```
运行: BSTTest
E:\java帮助文档\bin\java.exe "-javaagent:E:\java环境\IntelliJ IDEA Community Edition 2023.2.1\lib\idea_rt.jar=54960:E:\java环境
-----
There are 1 nodes in this BST.
The height of this BST is 1.
-----
remove unsuccessful--vertebrate
update successful--overlap overlap
remove unsuccessful--affection
search successful--persevere v. 坚忍
-----
There are 11 nodes in this BST.
The height of this BST is 5.
-----
update successful--pretentiousness pretentiousness
-----
There are 13 nodes in this BST.
The height of this BST is 5.
-----
remove successful--sill n. 基石, 门槛, 窗台
search unsuccessful--listless null
search successful--gollop v. n. 大口吞咽
update successful--sagacious sagacious
remove unsuccessful--propitiate
update successful--ideology ideology
search successful--puerile adj. 幼稚的, 儿童的
update successful--laggard laggard
search unsuccessful--penalize null
search successful--deify v. 奉为神, 崇拜
search unsuccessful--midget null
search successful--mutton n. 羊肉
update successful--conflagration conflagration
search unsuccessful--morass null
search successful--breach v. 损坏, 泄密n. 缺口
search successful--sagacious sagacious
remove successful--miscreant n. 恶棍, 歹徒
-----
There are 33 nodes in this BST.
The height of this BST is 8.
-----
search unsuccessful--guarded null
search successful--sagacious sagacious
remove unsuccessful--egalitarian
update successful--incendiary incendiary
search successful--nocturnal adj. 夜晚的, 夜间发生的
update successful--deify deify
remove successful--solemnity n. 庄严, 肃穆
search unsuccessful--tongs null
search successful--squash v. 压碎, 挤压, n. 南瓜
update successful--abate abate
-----
There are 46 nodes in this BST.
The height of this BST is 9.
-----
search unsuccessful--vaccine null
search successful--incendiary incendiary
```

```
运行: BSTTest x
The height of this BST is 9.
-----
search unsuccessful---vaccine null
search successful---incendiary incendiary
search unsuccessful---unctuous null
search successful---croak n. 蛙鸣声
search successful---gravel n. 碎石, 砂砾
update successful---coy coy
-----
There are 53 nodes in this BST.
The height of this BST is 9.
-----
search successful---behold v. 目睹, 看见
remove successful---overlap overlap
remove successful---pretentiousness pretentiousness
search successful---nocturnal adj. 夜晚的, 夜间发生的
update successful---mast mast
-----
There are 58 nodes in this BST.
The height of this BST is 9.
-----
remove successful---peruse v. 细读, 精读
remove successful---suffice v. 足够, (食物) 满足
remove successful---upstart n. 突然升官的人, 暴发户
update successful---unwieldy unwieldy
update successful---auspices auspices
update successful---sagacious sagacious
search unsuccessful---infantry null
search successful---hide n. 兽皮
remove unsuccessful---recriminate
update successful---bamboozle bamboozle
update successful---ingress ingress
remove successful---profundity n.
remove successful---flicker v. 闪烁, 摇曳
search successful---chivalry n. 骑士制度
update successful---hepatic hepatic
remove unsuccessful---reciprocity
search successful---gollop v. n. 大口吞咽
search successful---liability n. 倾向, 债务
search successful---veal n. 小牛肉
remove unsuccessful---throttle
remove successful---sagacious sagacious
remove unsuccessful---oath
search successful---impeach v. 指摘, 弹劾
remove successful---mishap n. 不幸, 坏运气
-----
There are 81 nodes in this BST.
The height of this BST is 10.
-----
remove successful---malapropism n. 字的误用
remove successful---splendid adj. 壮观的, 极好的
update successful---abate abate
search unsuccessful---turbid null
search successful---ingress ingress
search unsuccessful---whimsical null

版本控制 运行 TODO 问题 终端 服务 构建
// 构建在 750毫秒内成功完成 (1 分钟之前)
```

.....
.....
.....

```

运行: BSTTest
-----
There are 402 nodes in this BST.
The height of this BST is 17.
-----
There are 402 nodes in this BST.
The height of this BST is 17.
-----
search success---renaissance renaissance
update success---caste caste
update success---behold behold
search success---satiated v. 使饱足, 生腻
search success---stapler stapler
search success---tiff v. n. 吵嘴, 呕气
remove success---antedate (在信、文件上) 写上较早日期
search success---zephyr n. 和风, 西风
search success---exped v.
remove success---permeable adj. 可渗透的
update success---toupee toupee
-----
There are 410 nodes in this BST.
The height of this BST is 17.
-----
remove success---aperitif n. 开胃酒
search unsuccessful---crudity null
search success---icon n. 圣像, 偶像
-----
There are 412 nodes in this BST.
The height of this BST is 17.
-----
update success---orientation orientation
update success---credence credence
search success---zephyr n. 和风, 西风
remove success---supplicate v. 恳求, 乞求
search success---dynamite n. 炸药, 扣人心弦的事
remove unsuccessful---scurrilous
remove unsuccessful---regiment
search unsuccessful---ply null
search success---ellipsis n. 省略
-----
There are 417 nodes in this BST.
The height of this BST is 17.
-----
remove unsuccessful---arsonist
search unsuccessful---vandalism null
search success---royalty n. 皇室, 版税 (付给著作人的钱)
-----
There are 424 nodes in this BST.
The height of this BST is 17.
-----
remove success---gambol n. 雀跃, 嬉戏
remove unsuccessful---doting

进程已结束, 退出代码为 0

```

(经比对, 与提供正确结果一致)

总结和收获

- (1)深入理解了 BST 的工作原理和实现方式, 包括如何插入、删除、搜索和更新节点。
- (2)实现泛型版本的 BST 让我更加熟悉 Java 的泛型机制, 这在处理不

同数据类型时非常有用。(3)通过解析文件中的命令并执行相应操作,我提高了处理文件和字符串的能力

实验 2 使用 BST 为文稿建立单词索引表

题目

在使用很多文字编辑软件时,都有对所编辑的文稿进行搜索的功能。当文稿内容的数据量比较大时,检索的效率就必须要进行考虑了。利用任务 1 中构建的 BST 数据结构,可以比较有效地解决这个问题。将文稿中出现的每个单词都插入到用 BST 构建的搜索表中,记录每个单词在文章中出现的行号。

本次实验准备了一篇英文文稿,文件名为 `article.txt1`,请按照上面的样例根据该 `article.txt` 中的内容构建索引表,并将索引表的内容按照单词的字典序列输出到 `index_result.txt` 文件中。在构建 BST 中的结点数据时,请认真思考记录行号的数据类型,如果可能,尽可能使用在之前实验中自己构建的数据结构,也是对之前数据结构使用的一种验证。

数据设计

该实验用到数据结构与第一个实验 BST 基本一致,但要将 BST 的泛型具体到 `<String,StringBuilder>` 两种类型,即 Key 为 String 类型,Value 为 StringBuilder 类型,以便于更新字符串。

其次要改变 insert () 方法的部分逻辑：在更新部分，不能单纯地更换 Value 的值，而是要更新 Value，将当前行数添加到已有 Value 上。

在主程序中编写了 *buildWordIndexFromFile* 方法，传入一个 txt 文件以及 BST 对象，可以逐行提取文件文本内容，切割单词后使用 BST 的 insert 方法对其处理。

算法设计

BST 与上一题基本相同，而在 insert 部分将原本“遇到已存在 Key 的键值对进行替换”的处理方法改为“利用 StringBuilder 类的 append 方法将传入的 Value 值添加到相同 Key 的键值对的 Value 后面”。

在主测试函数中，使用了正则表达式来分割每一行的单词，以及排除文章中正常英文单词外其他成分的干扰，具体见主干代码说明。

主干代码说明

修改后的 insert ():

```
1.     private Node insertRec(Node node, String key, StringBui  
      lder val) {  
2.         if (node == null) return new Node(key, val);  
3.  
4.         int cmp = key.compareTo(node.key);  
5.         if (cmp < 0) node.left = insertRec(node.left, key,  
      val);  
6.         else if (cmp > 0) node.right = insertRec(node.right  
      , key, val);  
7.         else {  
8.             // 向已有键值对的 Value 后添加新的行数信息  
9.             node.val.append(" ");  
10.            node.val.append(val);  
11.        }  
12.
```



```
13.         return node;
14.     }
```

Ctrl+M

主测试方法：

```
1. public static void buildWordIndexFromFile(File file, WordIn
   dexBST bst) {
2.     try {
3.         BufferedReader reader = new BufferedReader(new File
   Reader(file)); // 创建文件读取器
4.         String line; // 用于存储每行文本的字符串
5.         int lineNumber = 1; // 行号初始化为1
6.         while ((line = reader.readLine()) != null) { // 逐行
   读取文件内容
7.             String[] words = line.toLowerCase().split("\\W+
   "); // 将当前行按非字母字符分割为单词数组
8.             for (String word : words) { // 遍历当前行的单词数
   组
9.                 // 过滤掉数字、日期等非单词内容
10.                if (word.matches("[a-zA-Z]+")) { // 如果单词
   只包含字母
11.                    StringBuilder temp = new StringBuilder(
   ); // 创建一个 StringBuilder 对象
12.                    temp.append(lineNumber); // 将行号添加
   到 StringBuilder 中
13.                    bst.insert(word, temp); // 将单词作为键，
   行号作为值插入二叉搜索树
14.                }
15.            }
16.            lineNumber++; // 行号自增
17.        }
18.        reader.close(); // 关闭文件读取器
19.    } catch (IOException e) { // 捕获 IOException 异常
20.        e.printStackTrace(); // 打印异常堆栈信息
21.    }
22. }
23. }
```

Ctrl+M

运行结果展示

index_result.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

[a---<2 11 19 21 44 72 76 80 83 86 87 89 90 94 95 98 98 102 110 111 112 112 113 116 121 121 123 130 133 138 144 149 153 155 157 160 162 167 171 185 186 199 200 201 21
316 320 327 344 353 353 354 356 374 377 382 398 398 405 410 417 419 419 420 423 439 451 453 467 479 480 481 483 484 487 488 490 499 508 520 530 534 548 551 551 555
673 673 674 675 689 689 689 694 697 706 707 709 713 715 720 734 734 740 742 743 753 753 765 768 771 785 796 798 801 808 809 811 811 813 814 827 833 834 837 839 842
984 985 988 988 990 991 998 1000 1000 1002 1005 1009 1012 1019 1032 1057 1059 1062 1062 1066 1075 1076 1121 1123 1124 1127 1128 1135 1140 1140 1142 1146 1157 11
1289 1289 1291 1294 1300 1305 1305 1306 1307 1314 1317 1332 1332 1333 1334 1335 1336 1337 1338 1339 1341 1360 1362 1367 1368 1369 1374 1377 1379 1381 1384 1397
1494 1498 1522 1523 1530 1538 1543 1544 1563 1567 1581 1591 1597 1598 1608 1619 1641 1650 1652 1673 1675 1675 1685 1688 1693 1696 1698 1702 1704 1707 1712 1714
1913 1919 1926 1926 1927 1932 1932 1933 1939 1951 1951 1953 1966 1997 1997 2000 2002 2004 2014 2015 2019 2032 2035 2037 2037 2055 2061 2074 2075 2082 2093 2093
2292 2293 2296 2301 2305 2305 2311 2314 2318 2320 2323 2327 2328 2337 2345 2345 2352 2353 2354 2354 2363 2367 2370 2371 2379 2385 2392 2398 2406 2410 2415 2426
2568 2578 2604 2605 2607 2616 2629 2634 2640 2651 2659 2661 2670 2675 2676 2679 2679 2683 2683 2687 2690 2692 2721 2728 2738 2752 2758 2758 2803 2820 2822 2824
2992 2993 2996 2998 3015 3026 3027 3044 3045 3048 3050 3054 3058 3060 3060 3060 3066 3073 3079 3089 3094 3094 3096 3103 3103 3104 3104 3104 3107 3111 3122 3124
3224 3225 3225 3227 3234 3244 3282 3282 3283 3283 3286 3289 3292 3296 3296 3305 3305 3314 3316 3316 3325 3327 3337 3343 3346 3348 3351 3355 3374 3386 3386 3397
3617 3620 3630 3638 3639 3661 3664 3664 3676 3678 3686 3704 3706 3706 3708 3711 3713 3720 3731 3731 3734 3738 3750 3765 3766 3767 3776 3777 3777 3778 3779 3781
3923 3923 3924 3925 3925 3926 3927 3928 3930 3930 3932 3944 3945 3948 3962 3962 3966 3967 3971 3977 3982 3983 3984 3986 3992 3992 3994 3995 3996 4004 4006 4016
4155 4155 4157 4161 4162 4166 4167 4168 4172 4174 4174 4174 4179 4196 4207 4219 4223 4227 4227 4229 4241 4242 4242 4244 4245 4251 4257 4262 4267 4269 4274 4280
4430 4430 4445 4452 4461 4461 4462 4463 4464 4466 4466 4470 4473 4475 4483 4483 4485 4492 4499 4499 4500 4504 4505 4505 4506 4519 4530 4545 4550 4551 4552 4559 4560
4706 4707 4708 4709 4712 4713 4715 4736 4740 4742 4747 4753 4753 4753 4756 4763 4767 4772 4787 4790 4791 4800 4802 4810 4811 4818 4823 4828 4833 4838 4841 4842
5014 5015 5019 5033 5035 5037 5038 5040 5041 5041 5048 5048 5049 5051 5065 5068 5074 5077 5084 5086 5086 5087 5091 5093 5096 5100 5102 5102 5103 5109 5112 5120
5252 5273 5275 5290 5299 5300 5311 5320 5321 5323 5323 5325 5326 5339 5342 5344 5359 5367 5370 5386 5389 5392 5397 5401 5402 5402 5402 5406 5406 5406 5415 5419
5524 5525 5539 5540 5540 5543 5545 5559 5562 5563 5570 5571 5573 5577 5577 5577 5581 5583 5584 5592 5593 5597 5606 5606 5608 5614 5617 5620 5622 5624 5624 5625
5717 5719 5721 5722 5723 5731 5735 5737 5737 5740 5741 5741 5741 5755 5770 5770 5771 5772 5777 5788 5790 5790 5791 5792 5795 5795 5799 5802 5806 5807 5816 5820
5954 5954 5957 5960 5964 5965 5968 5977 5983 5987 5988 5992 5996 6000 6001 6017 6018 6021 6023 6024 6026 6028 6032 6032 6033 6048 6065 6077 6077 6078 6078 6079
6234 6235 6237 6237 6240 6241 6245 6246 6246 6246 6247 6251 6253 6259 6262 6264 6275 6280 6290 6294 6302 6310 6321 6329 6330 6331 6331 6333 6335 6338 6339 6345

[aback---<25194 28913>]

[abandon---<2860 3314 3754 32802 33361 33367>]

[abandoned---<3152 3642 4166 9606 16582 17591>]

[abandons---<15317>]

[abbots---<11102>]

[abductor---<21773>]

[abdullah---<5523 8796 8837 8922 8960 9050 9311 9324>]

[abelwhite---<8685 8695 8711 8727>]

[aberdeen---<14712 19064>]

[abhor---<4957>]

[abhorrent---<9580>]

[abide---<23262>]

[abiding---<13481>]

[abilities---<29510 31795>]

[ability---<22539 22644 23831 23874 24617 25039 29659>]

[abjure---<14197>]

[able---<82 266 654 800 1745 1803 2019 2918 3221 3295 3538 3853 4130 4183 4650 4673 4680 4694 5010 5732 5948 6345 6429 6502 6629 6879 7715 7802 8035 8304 8678 9350 9860 11
18179 18557 19598 19663 20511 20959 21209 22951 23183 23757 24184 25382 25456 26083 26413 26772 26862 27300 27486 27896 28201 28202 28335 28473 29310 29432 29990 30151

[abnormal---<20970 28120>]

[abnormally---<21500>]

[aboard---<1714 8306 9280 25077 25083 25220>]

[abode---<17637>]

[abominable---<14762 25351 32601>]

[abomination---<12862>]

[aborigines---<7505>]

[abound---<13897>]

[about---<82 185 197 222 232 251 333 383 383 656 665 674 835 861 879 892 998 1090 1111 1118 1175 1276 1284 1293 1401 1463 1503 1575 1591 1685 1794 1979 2012 2049 2109 2151
4120 4252 4275 4323 4331 4337 4350 4467 4497 4566 4580 4596 4653 4665 5145 5233 5313 5535 5704 5743 5844 5897 5996 6038 6047 6100 6370 6374 6404 6443 6451 6606 6703 6795
7942 7962 8077 8117 8170 8171 8244 8245 8337 8365 8494 8502 8665 8704 8715 8752 8918 8931 9063 9107 9116 9122 9182 9189 9217 9228 9236 9351 9370 9382 9390 9428 9429 9436
11494 11551 11586 11596 11707 11850 11909 11922 11943 11963 11993 12002 12022 12023 12084 12213 12232 12313 12360 12589 12593 12713 12752 12811 12819 12906 12907 12911
14452 14550 14651 14653 14679 14724 14967 15141 15154 15233 15310 15389 15442 15525 15645 15729 15836 15903 15909 15946 15956 16063 16123 16277 16287 16300 16302 16308
18237 18293 18299 18314 18466 18560 18696 18755 18845 18943 18975 19105 19200 19309 19323 19364 19520 19661 19709 19841 19875 19977 20117 20126 20246 20332 20423 20427
21819 21902 21927 21968 21969 22055 22086 22185 22265 22361 22368 22377 22391 22401 22416 22464 22512 22758 22824 22844 22890 22932 22971 22997 22999 23006 23015 23030
23897 23908 23910 23916 23946 23948 23952 23976 24194 24331 24336 24544 24555 24708 24776 24826 25047 25127 25201 25202 25276 25321 25412 25446 25496 25501 25507 25508
27338 27425 27482 27483 27549 27608 27641 27729 27767 27861 27885 27912 28057 28076 28239 28385 28420 28421 28424 28427 28474 28484 28554 28608 28648 28754 29084 2915
30756 30813 30841 30908 30914 31143 31264 31294 31350 31493 31517 31519 31673 31707 31878 31966 32070 32102 32155 32193 32224 32351 32793 32840 32860 32871 32935 32951
[above---<537 1323 2033 2633 2846 2866 2904 3034 3175 3805 3852 3881 3911 3925 4137 4561 5419 5518 5965 5979 6236 6260 6398 6787 6947 6966 7192 7459 8653 8675 9505 10475
20332 21374 21440 21545 21915 22075 23025 23327 23387 23401 23872 25018 25162 25572 25847 25997 26893 27177 27900 28848 28945 30142 31626 32142 32452 32465 32482>]

[abroad---<1004 12319 13085>]

[abrupt---<19252 27196 28144 28162>]

[abruptly---<2497 10682 18906 29307 31690>]

[abruptness---<27994>]

[absence---<984 1838 3977 7570 9297 9319 11235 11667 13895 18715 24057 26970 30662 31459 31957 32272 32867>]

[absent---<3038 3729 3960 4147 4758 12742 27122 28152 30986>]

[absentee---<3637>]

[absentees---<21920>]

[absolute---<2633 4298 9828 11477 11556 12706 13568 14785 15189 17422 17815 17816 17830 18056 18469 19705 20046 21755 21911 26863 26934 27939 29942 31065 31072 31366 32

.....

.....

.....

```
[younger----<4156 4829 11885 18704 21111 24983 26693 26721 28515 28537 28983 29063>]
[youngest---<2864>]
[youngster---<4599 13727 19782 32699 33191>]
[youngsters---<1920>]
[your---<195 222 350 637 657 669 678 713 761 847 882 902 950 963 1103 1166 1171 1210 1238 1385 1391 1501 1502 1514 1612 1667 1667 1725 1736 1743 1744 1912 1977 3398 3413 3430 3470 3521 3531 3542 3561 3572 3719 4248 4254 4262 4292 4294 4474 4514 4532 4616 4621 4629 4806 4813 4940 5028 5039 5057 5075 5085 5093 5126 512 6066 6073 6075 6165 6300 6322 6359 6432 6472 6480 6547 6569 6601 6650 6754 6817 6840 6905 6960 7068 7206 7372 7438 7479 7609 7611 7710 7811 7894 7916 7927 797 9296 9458 9458 9474 9477 9484 9490 9515 9663 9669 9677 9680 9680 9711 9714 9715 9780 9782 9783 9818 9859 9859 9868 9897 9911 9933 9946 9998 10037 10224 10239 10782 10793 10798 10829 10830 10855 10856 10929 10930 10932 10962 10964 10988 11017 11046 11054 11060 11133 11174 11196 11235 11235 11251 11267 11272 11309 12013 12035 12062 12072 12077 12092 12094 12094 12108 12112 12118 12121 12444 12459 12521 12646 12668 12676 12680 12770 12777 12784 12784 12788 12793 12794 13678 13684 13781 13783 13809 13820 13822 14013 14041 14045 14048 14248 14535 14541 14577 14578 14597 14713 14821 14824 14968 14972 15013 15034 15052 15085 15986 16054 16071 16132 16134 16259 16270 16279 16497 16505 16510 16517 16520 16528 16530 16530 16618 16663 16686 16696 16738 16784 16804 16810 16831 16847 17408 17408 17461 17584 17703 17703 17707 17751 17754 17757 17758 17849 17858 17894 17899 17903 17904 17931 17995 18078 18178 18179 18241 18257 18305 18365 19315 19414 19417 19494 19554 19555 19578 19590 19639 19661 19837 19876 19899 19911 19912 19915 19924 19958 19978 19999 19999 20000 20000 20001 20078 20092 20454 20461 20493 20513 20523 20524 20533 20541 20567 20573 20577 20612 20769 20809 20809 20827 20840 20846 20854 20880 20881 20892 20896 20898 20905 20907 21727 21769 21783 21847 21872 22080 22149 22171 22280 22287 22289 22364 22369 22371 22377 22384 22392 22397 22407 22466 22483 22493 22493 22507 22535 22591 23325 23348 23376 23380 23464 23470 23538 23584 23585 23657 23695 23701 23728 23743 23761 23795 23831 23872 23874 23904 23905 23916 23922 23923 23931 23937 24574 24579 24584 24613 24613 24630 24690 24766 24800 24896 24920 24970 24971 24975 24984 25029 25064 25100 25282 25283 25341 25385 25387 25454 25573 25580 26803 26879 26909 26930 27002 27003 27006 27006 27012 27023 27036 27040 27041 27042 27053 27054 27369 27417 27659 27660 27660 27663 27815 27816 27816 27834 27880 27921 27941 27945 27966 27990 28008 28008 28025 28103 28246 28285 28392 28506 28657 28658 28658 28659 28686 28771 28837 28908 28911 29029 29029 29041 30247 30253 30326 30344 30346 30347 30365 30374 30428 30431 30437 30459 30525 30534 30643 30662 30689 30709 30713 30860 30888 30891 30967 30969 30972 30983 31518 31614 31657 32059 32116 32135 32138 32173 32176 32186 32441 32473 32482 32483 32486 32516 32529 32566 32573 32573 32647 32786 32790 32825 32907 32914 [yours---<174 848 1500 6053 6064 6777 7915 8478 8856 9253 9681 10625 10695 11017 11195 12332 13175 13609 14571 16207 16329 18580 18791 19044 20675 20899 2122 [yourself---<119 958 1585 1636 2731 2792 3171 3428 3470 4701 4850 4971 5335 6177 6435 6606 6898 7256 7657 7933 9650 9938 10263 10811 10829 10898 10913 10933 1 19491 19556 19575 20099 20613 20644 20891 20917 20943 22245 22471 24028 24035 24080 25583 25589 26384 26561 27563 27827 27834 27934 28028 29112 29182 29615 [yourself---<5326 7449 10935 11550 16376 16686 23496 27636>] [youth---<2964 3124 3505 5432 5653 7212 10506 10845 10847 13035 15354 15834 19207 23631 24574 26838 27908 31820 31892 32975>] [youths---<1909>] [zeal---<455 24051>] [zealand---<11901 23855 24588>] [zealous---<21809>] [zero---<14099 20665>] [zest---<16957>] [zigzag---<7102 15998>] [zion---<2988 2990>] [zum---<9522 9522>]
```

总结和收获

- (1) 学习了正则表达式的使用
- (2) 对 BST 的应用更加熟练深入

实验 3 学会使用堆

题目

堆是一棵完全二叉树，是二叉树众多应用中一个最适合用数组方式存储的树形，所以必须要 完全掌握。堆的主要用途是构建优先队列 ADT，除此之外，高效的从若干个数据中连续找到剩余元素中最小（或最大）都是堆的应用场景，比如构建 Huffman 树时，需要从候选频率中选择最小的 两个；比如最短路径 Dijkstra 算法中要从最短路径估计值数组中选取当前最小值等，所有的这些应用都因为使用了堆而如虎添翼。本任务从几个方面对堆进行实践：

- 1) 参看书中的代码，撰写一个具有 insert、delete、getMin 等方法的 Min_Heap;
- 2) 完成一个使用 1) 编写的 Min_Heap 的排序算法;
- 3) 堆是二叉树的一个应用，如果将堆的概念扩展到三叉树（树中每个结点的子结点数最多是 3 个），此时将形成三叉堆。除了树形和二叉堆不一致外，堆序的要求是完全一致的。使用完全三叉树形成堆，在 n 个结点下显然会降低整棵树的高度，但是在维持某个结点的堆序时需要比较的次数会增加（这个时候会是 3 个子结点之间进行互相比较大或小），这这也是一个权衡问题。现在要求将 1) 中实现的二叉堆改造成三叉堆，并通过 2) 中实现的排序验证这个三叉堆是否正确。

先来解决（1）（2）：

数据设计

1. 变量设计：

Heap：使用 ArrayList<Integer> 来存储堆中的元素。ArrayList 提供了动态数组的特性，使得堆的大小可以根据需要动态调整，同时也提供了便捷的索引和修改操作。

2. 方法设计：

- parent(int i)：根据当前节点的索引计算其父节点的索引，用于插入和删除操作时维护堆的性质。

- leftChild(int i): 根据当前节点的索引计算其左子节点的索引。
- rightChild(int i): 根据当前节点的索引计算其右子节点的索引。
- insert(int val): 将新元素插入堆中, 并根据堆的性质调整堆, 保持其为最小堆。
- delete(int val): 删除指定值的元素, 并根据堆的性质调整堆, 保持其为最小堆。
- extractMin(): 提取并移除堆的根节点, 然后调整堆, 使其保持最小堆的性质。
- heapify(int i): 根据指定的节点索引, 将以该节点为根的子树调整为最小堆。
- getMin(): 返回堆的根节点的值, 即最小值。
- heapSort(int[] array): 堆排序算法。利用最小堆进行排序, 将给定数组中的元素按照升序排列。

算法设计

1. insert(int val):

通过比较新元素与其父节点的值, 逐步向上移动元素, 直到满足最小堆的性质。

2. delete(int val):

- 首先标记要删除的元素为最小整数 `Integer.MIN_VALUE`。

- 通过将该元素逐步向上移动，直到满足最小堆的性质。
- 最后提取并移除堆的根节点，再进行堆调整。

3. `public int extractMin()`:

- 如果堆为空，返回最小整数。
- 如果堆中只有一个元素，直接移除并返回该元素。
- 否则，提取堆的根节点，并将堆的最后一个节点移到根节点位置，然后进行堆调整。

4. `private void heapify(int i)`:

- 获取左右子节点索引。
- 找出三者中的最小值作为根节点，并与根节点交换位置。
- 递归调用 `heapify` 方法，继续调整子树。

5. `heapSort(int[] array)`:

- 创建一个 `MinHeap` 对象，并将数组中的所有元素插入堆中。
- 依次提取最小值，并放入数组中，实现堆排序。

主干代码说明

节点索引的计算：

```
1.
2. public class MinHeap {
3.     private ArrayList<Integer> heap; // 存储堆元素
   的 ArrayList
4.
```

```

5.     public MinHeap() {
6.         heap = new ArrayList<>(); // 初始化堆
7.     }
8.
9.     // 获取父节点索引
10.    private int parent(int i) {
11.        return (i - 1) / 2; // 返回父节点索引
12.    }
13.
14.    // 获取左子节点索引
15.    private int leftChild(int i) {
16.        return 2 * i + 1; // 返回左子节点索引
17.    }
18.
19.    // 获取右子节点索引
20.    private int rightChild(int i) {
21.        return 2 * i + 2; // 返回右子节点索引
22.    }

```

插入方法：

```

1. // 插入元素
2.     public void insert(int val) {
3.         heap.add(val); // 将元素添加到堆的末尾
4.         int index = heap.size() - 1; // 获取新元素的索引
5.         // 调整堆以满足最小堆性质
6.         while (index > 0 && heap.get(parent(index)) > heap.
            get(index)) {
7.             // 如果值比父节点小，则与父节点交换位置
8.             int temp = heap.get(parent(index));
9.             heap.set(parent(index), heap.get(index));
10.            heap.set(index, temp);
11.            index = parent(index); // 更新索引为父节点索引
12.        }
13.    }
14.

```

删除方法：

```

1. // 删除元素
2.     public void delete(int val) {
3.         int index = heap.indexOf(val); // 获取要删除元素的索引
4.         if (index == -1) return; // 如果值不在堆中，则直接返回
5.         heap.set(index, Integer.MIN_VALUE); // 将值替换为最小整数

```

```

6.         while (index > 0 && heap.get(parent(index)) > heap.
           get(index)) {
7.             // 如果值比父节点小，则与父节点交换位置
8.             int temp = heap.get(parent(index));
9.             heap.set(parent(index), heap.get(index));
10.            heap.set(index, temp);
11.            index = parent(index); // 更新索引为父节点索引
12.        }
13.        extractMin(); // 删除值并调整堆
14.    }
15.
16.    // 获取并移除最小值
17.    public int extractMin() {
18.        if (heap.isEmpty()) return Integer.MIN_VALUE; // 如
           果堆为空，返回最小整数
19.        if (heap.size() == 1) return heap.remove(0); // 如果
           堆只有一个元素，直接移除并返回该元素
20.
21.        int root = heap.get(0); // 获取堆的根节点
22.        heap.set(0, heap.remove(heap.size() - 1)); // 将堆的
           最后一个节点移到根节点位置
23.        heapify(0); // 调整堆
24.
25.        return root; // 返回根节点值
26.    }
27.
28.    // 调整堆，使其满足最小堆性质
29.    private void heapify(int i) {
30.        int left = leftChild(i); // 获取左子节点索引
31.        int right = rightChild(i); // 获取右子节点索引
32.        int smallest = i; // 初始化最小值索引为当前节点索引
33.
34.        if (left < heap.size() && heap.get(left) < heap.get
           (smallest))
35.            smallest = left; // 如果左子节点存在且小于当前节
           点，则更新最小值索引为左子节点索引
36.        if (right < heap.size() && heap.get(right) < heap.g
           et(smallest))
37.            smallest = right; // 如果右子节点存在且小于当前节
           点，则更新最小值索引为右子节点索引
38.
39.        if (smallest != i) {
40.            // 如果最小值索引不等于当前节点索引，则交换值，并继
           续调整堆
41.            int temp = heap.get(i);

```



```

42.         heap.set(i, heap.get(smallest));
43.         heap.set(smallest, temp);
44.         heapify(smallest); // 递归调整堆
45.     }
46. }
47. }

```

排序算法：

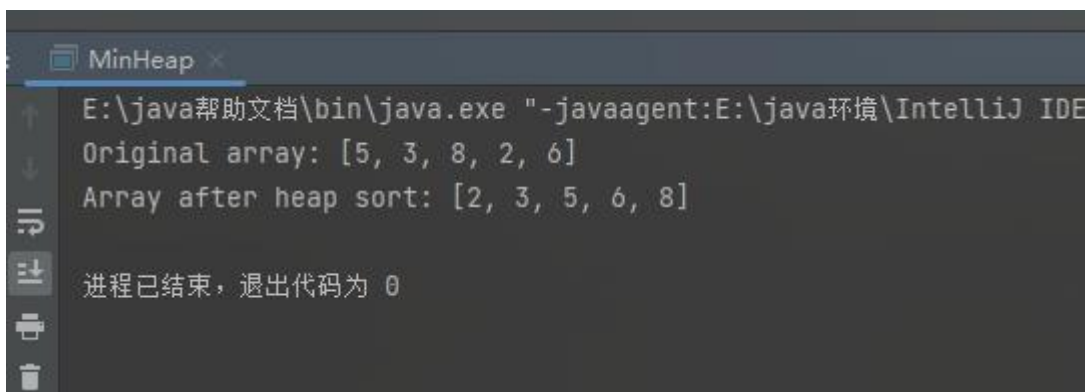
```

1. // 堆排序算法
2. public static void heapSort(int[] array) {
3.     MinHeap minHeap = new MinHeap(); // 创建一个最小堆对象
4.
5.     // 将数组中的元素插入堆中
6.     for (int num : array) {
7.         minHeap.insert(num);
8.     }
9.
10.    // 依次提取最小值，并放入数组中
11.    for (int i = 0; i < array.length; i++) {
12.        array[i] = minHeap.extractMin();
13.    }
14. }

```

Ctrl+M

运行结果展示



```

MinHeap x
E:\java帮助文档\bin\java.exe "-javaagent:E:\java环境\IntelliJ IDE
Original array: [5, 3, 8, 2, 6]
Array after heap sort: [2, 3, 5, 6, 8]
进程已结束，退出代码为 0

```

排序成功

总结和收获

(1) 加深了对数据结构中堆的理解

(2) 加强了我对递归调用的掌握。

实验 3 学会使用堆 (3)

数据设计

与二叉堆的数据、方法成员大体一致，但由于是三叉堆，需要重新定义获得子节点索引的方法，其余方法的实现逻辑有所改变，将在下面部分具体说明。

算法设计

我们将描述的重点主要落在与二叉堆实现的区别上：

首先是求子节点索引公式有所改变，其次是以下方法：

insert(int val)方法：

与二叉堆相比,插入操作的基本思路没有变化,但需要修改一些细节。

1. 将新节点插入到堆的最后一层,这一步与二叉堆相同。
2. 比较新节点的值与其父节点的值,如果新节点的值小于父节点(最小堆),则交换它们的位置。这里的区别在于使用 parent(index)方法计算父节点索引,而不是二叉堆中的 $(index-1)/2$ 公式。
3. 重复步骤2,直到新节点的值大于等于其父节点的值,或新节点成为根节点。这一步与二叉堆相同。

delete(int val)方法：

删除操作与插入操作类似,需要先找到要删除的值在堆中的索引,然后替换成一个极小值,再通过上浮操作恢复堆序。

1. 使用 heap.indexOf(val)方法查找要删除的值在堆中的索引,这与

二叉堆相同。

2. 将找到的索引位置的值替换为最小整数(Integer.MIN_VALUE), 这一步与二叉堆相同。
3. 与插入操作相反,执行上浮操作,使得极小值不断向上移动,直到找到合适的位置。这里需要修改的是比较的条件,因为三叉堆中需要与父节点比较,而不是与二叉堆中的父节点和兄弟节点比较。
4. 最后,调用 `extractMin()`方法删除堆顶元素(极小值),完成删除操作。这一步与二叉堆相同。

`extractMin()`方法:

`extractMin()`方法用于获取并移除堆顶元素,即最小值。

1. 如果堆为空,直接返回最小整数(Integer.MIN_VALUE)。这一步与二叉堆相同。
2. 如果堆中只有一个元素,直接移除并返回。这一步与二叉堆相同。
3. 将堆顶元素(`heap.get(0)`)暂存到一个变量中,然后将最后一个元素(`heap.get(heap.size()-1)`)移到堆顶位置,并从堆中移除最后一个元素。这一步与二叉堆相同。
4. 调用 `heapify(0)`方法,对堆顶元素进行下沉操作,以恢复堆序。这里的区别在于 `heapify()`方法需要修改,以处理三个子节点的比较。
5. 最后,返回之前暂存的堆顶元素值。这一步与二叉堆相同。

`heapify(int i)`方法:

`heapify()`方法用于恢复堆序,与二叉堆不同的是需要比较三个子节点,

而不是两个。

1. 获取节点 i 的三个子节点索引,使用 `firstChild(i)`、`secondChild(i)` 和 `thirdChild(i)`方法计算,而不是二叉堆中的 $2i+1$ 和 $2i+2$ 公式。
2. 将节点 i 与其三个子节点进行比较,找到最小值的索引 `smallest`。
这里需要比较三个子节点,而二叉堆只需要比较两个。
3. 如果 `smallest` 不等于 i ,说明最小值在子节点中,需要将 i 与 `smallest` 位置的值交换,并递归调用 `heapify(smallest)`方法,继续对交换后的子树进行下沉操作。
4. 如果 `smallest` 等于 i ,说明当前节点已经是最小值,无需进一步操作。

主干代码说明

获取父、子节点索引

```
1.
2.    // 获取父节点索引
3.    private int parent(int i) {
4.        return (i - 1) / 3; // 计算父节点索引
5.    }
6.
7.    // 获取第一个子节点索引
8.    private int firstChild(int i) {
9.        return 3 * i + 1; // 计算第一个子节点索引
10.    }
11.
12.    // 获取第二个子节点索引
13.    private int secondChild(int i) {
14.        return 3 * i + 2; // 计算第二个子节点索引
15.    }
16.
17.    // 获取第三个子节点索引
18.    private int thirdChild(int i) {
19.        return 3 * i + 3; // 计算第三个子节点索引
20.    }
```

插入方法：

```

1.    // 插入元素
2.    public void insert(int val) {
3.        heap.add(val); // 将元素添加到堆的末尾
4.        int index = heap.size() - 1; // 获取新元素的索引
5.        while (index > 0 && heap.get(parent(index)) > heap.
        get(index)) {
6.            // 如果值比父节点小，则与父节点交换位置
7.            int temp = heap.get(parent(index));
8.            heap.set(parent(index), heap.get(index));
9.            heap.set(index, temp);
10.           index = parent(index); // 更新索引为父节点索引
11.        }
12.    }
13.
14.    //

```

删除元素：

```

1. public void delete(int val) {
2.     int index = heap.indexOf(val); // 找到要删除元素的索引
3.     if (index == -1) return; // 如果值不在堆中，则直接返回
4.     heap.set(index, Integer.MIN_VALUE); // 将值替换为最小整数
5.     while (index > 0 && heap.get(parent(index)) > heap.
        get(index)) {
6.         // 如果值比父节点小，则与父节点交换位置
7.         int temp = heap.get(parent(index));
8.         heap.set(parent(index), heap.get(index));
9.         heap.set(index, temp);
10.        index = parent(index); // 更新索引为父节点索引
11.    }
12.    extractMin(); // 调整堆
13. }
14.
15. // 获取并移除最小值
16. public int extractMin() {
17.     if (heap.isEmpty()) return Integer.MIN_VALUE; // 如果堆为空，则返回最小整数
18.     if (heap.size() == 1) return heap.remove(0); // 如果堆中只有一个元素，则直接移除并返回该元素
19.
20.     int root = heap.get(0); // 获取堆顶元素

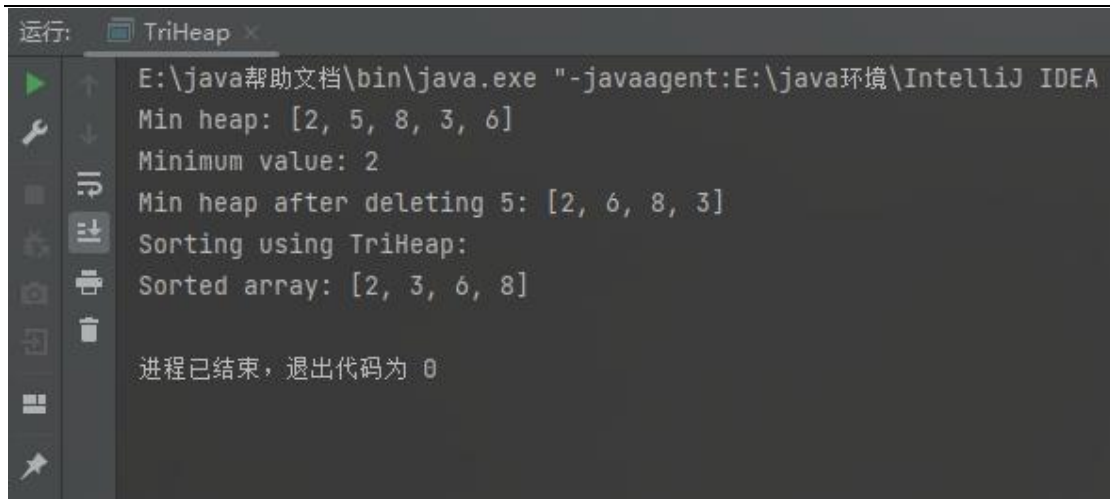
```

```

21.         heap.set(0, heap.remove(heap.size() - 1)); // 将末尾
            元素移至根节点位置
22.         heapify(0); // 调整堆
23.         return root; // 返回原堆顶元素
24.     }
25.
26.     // 调整堆，使其满足最小堆性质
27.     private void heapify(int i) {
28.         int first = firstChild(i); // 第一个子节点索引
29.         int second = secondChild(i); // 第二个子节点索引
30.         int third = thirdChild(i); // 第三个子节点索引
31.         int smallest = i; // 最小值的索引，默认为当前节点索引
32.
33.         if (first < heap.size() && heap.get(first) < heap.g
            et(smallest))
34.             smallest = first; // 如果第一个子节点存在且小于当
            前节点，则更新最小值索引
35.         if (second < heap.size() && heap.get(second) < heap
            .get(smallest))
36.             smallest = second; // 如果第二个子节点存在且小于当
            前节点，则更新最小值索引
37.         if (third < heap.size() && heap.get(third) < heap.g
            et(smallest))
38.             smallest = third; // 如果第三个子节点存在且小于当
            前节点，则更新最小值索引
39.
40.         if (smallest != i) {
41.             int temp = heap.get(i);
42.             heap.set(i, heap.get(smallest));
43.             heap.set(smallest, temp);
44.             heapify(smallest); // 递归调整堆
45.         }
46.     }
47.
48.     // 获取最小值
49.     public int getMin() {
50.         if (heap.isEmpty()) return Integer.MIN_VALUE; // 如
            果堆为空，则返回最小整数
51.         return heap.get(0); // 否则返回堆顶元素
52.     }
53.

```

运行结果展示



```
运行: TriHeap x
E:\java帮助文档\bin\java.exe "-javaagent:E:\java环境\IntelliJ IDEA
Min heap: [2, 5, 8, 3, 6]
Minimum value: 2
Min heap after deleting 5: [2, 6, 8, 3]
Sorting using TriHeap:
Sorted array: [2, 3, 6, 8]

进程已结束，退出代码为 0
```

排序正确

总结和收获

- (1) 深入理解了堆这一数据结构的原理和实现方式，并将其扩展到了三叉堆。
- (2) 学会了如何利用数组结构来表示树形结构，以及如何通过比较操作来维护堆序性质。
- (3) 通过堆排序验证了三叉堆的正确性，加深了对堆排序算法的理解。

附录

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class BST<Key extends Comparable<Key>, Value> {
5.     private Node root; // 根节点
6.
7.     private class Node {
8.         private Key key; // 键
9.         private Value val; // 值
10.        private Node left, right; // 左右子树
11.
12.        public Node(Key key, Value val) {
13.            this.key = key;
```

```

14.         this.val = val;
15.     }
16. }
17.
18. public BST() {
19. }
20.
21. // 插入键值对
22. public void insert(Key key, Value val) {
23.     root = insertRec(root, key, val);
24. }
25.
26. private Node insertRec(Node node, Key key, Value val) {
27.     if (node == null) return new Node(key, val);
28.
29.     int cmp = key.compareTo(node.key);
30.     if (cmp < 0) node.left = insertRec(node.left, key,
        val);
31.     else if (cmp > 0) node.right = insertRec(node.right
        , key, val);
32.     else node.val = val; // 更新现有的值
33.
34.     return node;
35. }
36.
37. // 删除节点并返回被删除节点的值
38. public Value remove(Key key) {
39.     List<Node> removedNodeWrapper = new ArrayList<>(1);
40.     root = removeNode(root, key, removedNodeWrapper);
41.     return removedNodeWrapper.isEmpty() ? null : remove
        dNodeWrapper.get(0).val;
42. }
43.
44. private Node removeNode(Node node, Key key, List<Node>
        removedNodeWrapper) {
45.     if (node == null) {
46.         return null;
47.     }
48.
49.     int cmp = key.compareTo(node.key);
50.     if (cmp < 0) {
51.         node.left = removeNode(node.left, key, removedN
            odeWrapper);
52.     } else if (cmp > 0) {

```



```

53.         node.right = removeNode(node.right, key, remove
           dNodeWrapper);
54.     } else {
55.         // Node found
56.         removedNodeWrapper.add(node); // Capture the re
           moved node
57.
58.         if (node.left == null) return node.right;
59.         if (node.right == null) return node.left;
60.
61.         Node t = node;
62.         node = min(t.right); // Find the smallest node
           in the right subtree
63.         node.right = deleteMin(t.right);
64.         node.left = t.left;
65.     }
66.     return node;
67. }
68.
69. private Node min(Node node) {
70.     while (node.left != null) node = node.left;
71.     return node;
72. }
73.
74. private Node deleteMin(Node node) {
75.     if (node.left == null) return node.right;
76.     node.left = deleteMin(node.left);
77.     return node;
78. }
79. // 搜索键
80. public Value search(Key key) {
81.     Node node = searchRec(root, key);
82.     return node == null ? null : node.val;
83. }
84.
85. private Node searchRec(Node node, Key key) {
86.     if (node == null) return null;
87.
88.     int cmp = key.compareTo(node.key);
89.     if (cmp < 0) return searchRec(node.left, key);
90.     else if (cmp > 0) return searchRec(node.right, key)
           ;
91.     else return node;
92. }
93. public boolean update(Key key, Value val) {

```

```

94.      Node node = searchNode(root, key);
95.      if (node == null) {
96.          // 如果没有找到具有指定键的元素
97.          return false;
98.      } else {
99.          // 找到了元素, 更新其值
100.         node.val = val;
101.         return true;
102.     }
103. }
104.
105. // 一个辅助方法来递归地搜索特定键的节点
106. private Node searchNode(Node node, Key key) {
107.     if (node == null) return null;
108.
109.     int cmp = key.compareTo(node.key);
110.     if (cmp < 0) return searchNode(node.left, key);
111.     else if (cmp > 0) return searchNode(node.right,
        key);
112.     else return node; // 找到了具有指定键的节点
113. }
114.
115. // 检查是否为空
116. public boolean isEmpty() {
117.     return root == null;
118. }
119.
120. // 清空树
121. public void clear() {
122.     root = null;
123. }
124.
125. // 打印中序遍历
126. public void printInorder() {
127.     printInorderRec(root);
128.     System.out.println();
129. }
130.
131. private void printInorderRec(Node node) {
132.     if (node != null) {
133.         printInorderRec(node.left);
134.         System.out.print(node.key + " ");
135.         printInorderRec(node.right);
136.     }
137. }

```

```

138.
139.      // 计算BST的节点数
140.      private int size(Node node) {
141.          if (node == null) return 0;
142.          return 1 + size(node.left) + size(node.right);
143.      }
144.
145.      // 计算BST的高度
146.      private int height(Node node) {
147.          if (node == null) return 0;
148.          return 1 + Math.max(height(node.left), height(node.right));
149.      }
150.
151.      // 显示结构
152.      public void showStructure() {
153.          System.out.println("-----
154.          ---");
155.          System.out.println("There are " + size(root) + "
156.          nodes in this BST.");
157.          System.out.println("The height of this BST is "
158.          + height(root) + ".");
159.          System.out.println("-----
160.          ---");
161.      }
162.  }
163.  }

```

Ctrl+M

```

1. import java.io.File;
2. import java.io.FileNotFoundException;
3. import java.util.Scanner;
4.
5. public class BSTTest {
6.     public static void main(String[] args) {
7.         BST<String, String> bst = new BST<>();
8.         String fileName = "D:\\大二作业
9.         \\homework3_testcases.txt"; // 更改为你的文件路径
10.
11.         try (Scanner scanner = new Scanner(new File(fileName))) {
12.             while (scanner.hasNextLine()) {
13.                 String line = scanner.nextLine().trim();
14.                 if (!line.isEmpty()) {
15.                     processLine(line, bst);
16.                 }
17.             }
18.         }
19.     }
20. }

```

```

16.         }
17.     } catch (FileNotFoundException e) {
18.         System.err.println("File not found: " + e.getMe
           ssage());
19.     }
20. }
21. private static void processLine(String line, BST<String
    , String> bst) {
22.     if (line.startsWith("+(")) {
23.         String[] parts = line.substring(2, line.length(
           ) - 1).split(",");
24.         String key = parts[0].substring(1, parts[0].len
           gth() - 1);
25.         String value = parts[1].substring(2, parts[1].l
           ength() - 2); // 移除引号
26.         bst.insert(key, value);
27.     } else if (line.startsWith("-(")) {
28.         String key = line.substring(3, line.length() -
           2);
29.         String removedValue = bst.remove(key);
30.         System.out.println("remove " + (removedValue !=
           null ? "success" : "unsuccess"))+"---
           "+key +" "+(removedValue != null ? removedValue : "");
31.     } else if (line.startsWith("?(")) {
32.         String key = line.substring(3, line.length() -
           2);
33.         String result = bst.search(key);
34.         System.out.println("search " + (result != null ?
           "success" : "unsuccess"))+"---"+key+" "+result);
35.     } else if (line.startsWith("(=")) {
36.         String[] parts = line.substring(2, line.length(
           ) - 1).split(",");
37.         String key = parts[0].substring(1, parts[0].len
           gth() - 1);
38.         String newValue = parts[1].substring(2, parts[1
           ].length() - 2); // 移除引号
39.         boolean updated = bst.update(key, newValue);
40.         System.out.println("update " + (updated ? "succe
           ss" : "unsuccess"))+"---"+ key+" "+ newValue);
41.     } else if (line.equals("#")) {
42.         bst.showStructure();
43.     }
44. }
45.
46. }

```

Ctrl+M

```

1. import java.io.BufferedWriter;
2. import java.io.IOException;
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. public class WordIndexBST{
7.     private Node root; // 根节点
8.
9.     private class Node {
10.         private String key; // 键
11.         private StringBuilder val; // 值
12.         private Node left, right; // 左右子树
13.
14.         public Node(String key, StringBuilder val) {
15.             this.key = key;
16.             this.val = val;
17.         }
18.     }
19.
20.     public WordIndexBST() {
21.     }
22.
23.     // 插入键值对
24.     public void insert(String key, StringBuilder val) {
25.         root = insertRec(root, key, val);
26.     }
27.
28.     private Node insertRec(Node node, String key, StringBu
29.         ilder val) {
30.         if (node == null) return new Node(key, val);
31.
32.         int cmp = key.compareTo(node.key);
33.         if (cmp < 0) node.left = insertRec(node.left, key,
34.             val);
35.         else if (cmp > 0) node.right = insertRec(node.right
36.             , key, val);
37.         else {
38.             node.val.append(" ");
39.             node.val.append(val);
40.         }
41.         return node;
42.     }

```

```

41.     public void printInorder(BufferedWriter writer) throws
        IOException {
42.         printInorderRec(root, writer);
43.         System.out.println();
44.     }
45.
46.     private void printInorderRec(Node node, BufferedWriter
        writer) throws IOException {
47.         if (node != null) {
48.             printInorderRec(node.left,writer);
49.             writer.write("[ "+node.key+"---
        "+ "<"+node.val+">"+" ]");
50.             writer.newLine();
51.             printInorderRec(node.right,writer);
52.         }
53.     }
54. }
55.

```

Ctrl+M

```

1. import java.io.*;
2.
3. public class WordIndex {
4.     public static void main(String[] args) throws IOExcepti
        on {
5.         WordIndexBST bst = new WordIndexBST();
6.
7.         // 读取存储在 D 盘 "大二作业" 目录下的文本文件
8.         String filePath = "D:\\大二作业
        \\homework3_article.txt";
9.         File file = new File(filePath);
10.        BufferedWriter writer = new BufferedWriter(new File
            Writer("D:\\大二作业\\index_result.txt"));
11.        if (file.exists() && file.isFile()) {
12.            // 读取文本文件内容并构建单词索引表
13.            buildWordIndexFromFile(file, bst);
14.        } else {
15.            System.out.println("文件不存在或者不是一个有效的文
                件。");
16.        }
17.
18.        // 打印BST 中的单词及其出现的行号。
19.        bst.printInorder(writer);
20.        writer.close();
21.    }

```

```

22.
23.    // 从文本文件中构建单词索引表的方法
24.
25.    public static void buildWordIndexFromFile(File file, WordIndexBST bst) {
26.        try {
27.            BufferedReader reader = new BufferedReader(new
                FileReader(file));
28.            String line;
29.            int lineNumber = 1;
30.            while ((line = reader.readLine()) != null) {
31.                String[] words = line.toLowerCase().split("\\W+");
32.                for (String word : words) {
33.                    // 过滤掉数字、日期等非单词内容
34.                    if (word.matches("[a-zA-Z]+")) {
35.                        StringBuilder temp = new StringBuil
                            der();
36.                        temp.append(lineNumber);
37.                        bst.insert(word, temp);
38.                    }
39.                }
40.                lineNumber++;
41.            }
42.            reader.close();
43.        } catch (IOException e) {
44.            e.printStackTrace();
45.        }
46.    }
47.}
48.

```

Ctrl+M

```

1. import java.util.ArrayList;
2.
3. public class MinHeap {
4.     private ArrayList<Integer> heap;
5.
6.     public MinHeap() {
7.         heap = new ArrayList<>();
8.     }
9.
10.    // 获取父节点索引
11.    private int parent(int i) {
12.        return (i - 1) / 2;

```

```

13.     }
14.
15.     // 获取左子节点索引
16.     private int leftChild(int i) {
17.         return 2 * i + 1;
18.     }
19.
20.     // 获取右子节点索引
21.     private int rightChild(int i) {
22.         return 2 * i + 2;
23.     }
24.
25.     // 插入元素
26.     public void insert(int val) {
27.         heap.add(val);
28.         int index = heap.size() - 1;
29.         while (index > 0 && heap.get(parent(index)) > heap.
            get(index)) {
30.             // 如果值比父节点小, 则与父节点交换位置
31.             int temp = heap.get(parent(index));
32.             heap.set(parent(index), heap.get(index));
33.             heap.set(index, temp);
34.             index = parent(index);
35.         }
36.     }
37.
38.     // 删除元素
39.     public void delete(int val) {
40.         int index = heap.indexOf(val);
41.         if (index == -1) return; // 如果值不在堆中, 则直接返回
42.         heap.set(index, Integer.MIN_VALUE); // 将值替换为最小
            整数
43.         while (index > 0 && heap.get(parent(index)) > heap.
            get(index)) {
44.             // 如果值比父节点小, 则与父节点交换位置
45.             int temp = heap.get(parent(index));
46.             heap.set(parent(index), heap.get(index));
47.             heap.set(index, temp);
48.             index = parent(index);
49.         }
50.         extractMin();
51.     }
52.
53.     // 获取并移除最小值
54.     public int extractMin() {

```



```

55.         if (heap.isEmpty()) return Integer.MIN_VALUE;
56.         if (heap.size() == 1) return heap.remove(0);
57.
58.         int root = heap.get(0);
59.         heap.set(0, heap.remove(heap.size() - 1));
60.         heapify(0);
61.
62.         return root;
63.     }
64.
65.     // 调整堆, 使其满足最小堆性质
66.     private void heapify(int i) {
67.         int left = leftChild(i);
68.         int right = rightChild(i);
69.         int smallest = i;
70.
71.         if (left < heap.size() && heap.get(left) < heap.get
            (smallest))
72.             smallest = left;
73.         if (right < heap.size() && heap.get(right) < heap.g
            et(smallest))
74.             smallest = right;
75.
76.         if (smallest != i) {
77.             int temp = heap.get(i);
78.             heap.set(i, heap.get(smallest));
79.             heap.set(smallest, temp);
80.             heapify(smallest);
81.         }
82.     }
83.
84.     // 获取最小值
85.     public int getMin() {
86.         if (heap.isEmpty()) return Integer.MIN_VALUE;
87.         return heap.get(0);
88.     }
89.
90.     // 堆排序算法
91.     public static void heapSort(int[] array) {
92.         MinHeap minHeap = new MinHeap();
93.
94.         // 将数组中的元素插入堆中
95.         for (int num : array) {
96.             minHeap.insert(num);
97.         }
    
```

```

98.
99.         // 依次提取最小值，并放入数组中
100.        for (int i = 0; i < array.length; i++) {
101.            array[i] = minHeap.extractMin();
102.        }
103.    }
104.
105.        // 示例用法
106.        public static void main(String[] args) {
107.            int[] array = {5, 3, 8, 2, 6};
108.
109.            System.out.println("Original array: " + java.util.Arrays.toString(array)); // 输出原始数组
110.
111.            heapSort(array); // 对数组进行堆排序
112.
113.            System.out.println("Array after heap sort: " + java.util.Arrays.toString(array)); // 输出排序后的数组
114.        }
115.    }
116.

```

Ctrl+M

```

1. import java.util.ArrayList;
2.
3. public class TriHeap {
4.     private ArrayList<Integer> heap;
5.
6.     public TriHeap() {
7.         heap = new ArrayList<>();
8.     }
9.
10.    // 获取父节点索引
11.    private int parent(int i) {
12.        return (i - 1) / 3;
13.    }
14.
15.    // 获取第一个子节点索引
16.    private int firstChild(int i) {
17.        return 3 * i + 1;
18.    }
19.
20.    // 获取第二个子节点索引
21.    private int secondChild(int i) {
22.        return 3 * i + 2;

```

```

23.     }
24.
25.     // 获取第三个子节点索引
26.     private int thirdChild(int i) {
27.         return 3 * i + 3;
28.     }
29.
30.     // 插入元素
31.     public void insert(int val) {
32.         heap.add(val);
33.         int index = heap.size() - 1;
34.         while (index > 0 && heap.get(parent(index)) > heap.
            get(index)) {
35.             // 如果值比父节点小, 则与父节点交换位置
36.             int temp = heap.get(parent(index));
37.             heap.set(parent(index), heap.get(index));
38.             heap.set(index, temp);
39.             index = parent(index);
40.         }
41.     }
42.
43.     // 删除元素
44.     public void delete(int val) {
45.         int index = heap.indexOf(val);
46.         if (index == -1) return; // 如果值不在堆中, 则直接返回
47.         heap.set(index, Integer.MIN_VALUE); // 将值替换为最小
            整数
48.         while (index > 0 && heap.get(parent(index)) > heap.
            get(index)) {
49.             // 如果值比父节点小, 则与父节点交换位置
50.             int temp = heap.get(parent(index));
51.             heap.set(parent(index), heap.get(index));
52.             heap.set(index, temp);
53.             index = parent(index);
54.         }
55.         extractMin();
56.     }
57.
58.     // 获取并移除最小值
59.     public int extractMin() {
60.         if (heap.isEmpty()) return Integer.MIN_VALUE;
61.         if (heap.size() == 1) return heap.remove(0);
62.
63.         int root = heap.get(0);
64.         heap.set(0, heap.remove(heap.size() - 1));

```

```

65.         heapify(0);
66.
67.         return root;
68.     }
69.
70.     // 调整堆, 使其满足最小堆性质
71.     private void heapify(int i) {
72.         int first = firstChild(i);
73.         int second = secondChild(i);
74.         int third = thirdChild(i);
75.         int smallest = i;
76.
77.         if (first < heap.size() && heap.get(first) < heap.get(smallest))
78.             smallest = first;
79.         if (second < heap.size() && heap.get(second) < heap.get(smallest))
80.             smallest = second;
81.         if (third < heap.size() && heap.get(third) < heap.get(smallest))
82.             smallest = third;
83.
84.         if (smallest != i) {
85.             int temp = heap.get(i);
86.             heap.set(i, heap.get(smallest));
87.             heap.set(smallest, temp);
88.             heapify(smallest);
89.         }
90.     }
91.
92.     // 获取最小值
93.     public int getMin() {
94.         if (heap.isEmpty()) return Integer.MIN_VALUE;
95.         return heap.get(0);
96.     }
97.
98.     // 示例用法
99.     public static void main(String[] args) {
100.         TriHeap heap = new TriHeap();
101.         heap.insert(5);
102.         heap.insert(3);
103.         heap.insert(8);
104.         heap.insert(2);
105.         heap.insert(6);
106.

```

```

107.      System.out.println("Min heap: " + heap.heap); /
        / 输出: [2, 3, 8, 5, 6]
108.      System.out.println("Minimum value: " + heap.getM
        in()); // 输出: 2
109.
110.      heap.delete(5);
111.      System.out.println("Min heap after deleting 5: "
        + heap.heap); // 输出: [2, 3, 8, 6]
112.
113.      // 使用三叉堆进行排序
114.      System.out.println("Sorting using TriHeap:");
115.      ArrayList<Integer> sorted = new ArrayList<>();
116.      while (!heap.heap.isEmpty()) {
117.          sorted.add(heap.extractMin());
118.      }
119.      System.out.println("Sorted array: " + sorted);
        // 输出: [2, 3, 6, 8]
120.    }
121.  }
122.

```

Ctrl+M