# 数据结构与算法
# 实验报告

第 3 次



| | |
|---|---|
| 姓名 | 丁昌灏 |
| 班级 | 软件 72 班 |
| 学号 | 2174213743 |
| 电话 | 15327962577 |
| Email | dch0031@stu.xjtu.edu.cn |
| 日期 | 2020-11-22 |

# 目录

# 实验 1

## 1、题目

实现 BST 数据结构

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）。但是在我们后续的测试文件中，其实 key 和 value 都是 String 类型，所以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。

为了测试实现 BST 接口的数据结构是否运行正常，准备了两个文件，一个是 homework2_testcases.txt，一个是 homework2_result.txt 文件。其中，前一个包含了所有的对二叉检索树进行操作的命令内容，后一个则是执行命令文件之后的输出。

## 2、数据结构设计

为了便于输出二叉树中的结点数量，在类中定义一个辅助数据 nodesNum 来记录当前二叉树的结点数量。在进行删除操作时需要返回删除键对应的值，但是为了使删除操作递归统一，返回值为子树的根节点，于是再定义一个辅助变量存储当前删除的元素的值。


## 3、算法设计

二叉树中插入、删除、查找、更新、等操作与教材相同，本次插入算法中在插入结

点时如果键相同则更新对应的值，在这里不做赘述。

## 4、主干代码说明

插入算法：

```
public void insert(K key, V value) {
    root = inserthelp(root, key, value);
}

private BinNode<K, V> inserthelp(BinNode<K, V> root, K key, V value) {
    if (null == root) {
        if (null != key && null != value) {
            nodesNum++;
            return new BinNode<K, V>(key, value);
        } else {
            return null;
        }
    }
    if (root.key().compareTo(key) > 0) {
        root.setLeft(inserthelp(root.left(), key, value));
    } else if (root.key().compareTo(key) < 0) {
        root.setRight(inserthelp(root.right(), key, value));
    } else {
        root.setValue(value);
    }
    return root;
}
```

删除算法：

```
public V remove(K key) {
    deletedValue = null;
    root = removehelp(root, key);
    if (null != deletedValue) {
        nodesNum--;
    }
    return deletedValue;
}

private BinNode<K, V> removehelp(BinNode<K, V> root, K key) {
    if (null == key) {
        return null;
    }
    if (null == root) {
        return null;
    }
    if (root.key().compareTo(key) > 0) {
        root.setLeft(removehelp(root.left(), key));
    } else if (root.key().compareTo(key) < 0) {
        root.setRight(removehelp(root.right(), key));
    } else {
        deletedValue = (V) root.value();
        if (null == root.left() && null != root.right()) {
            root = root.right();
        } else if (null == root.right() && null != root.left()) {
            root = root.left();
        } else if (null == root.left() && null == root.right()) {
            root = null;
        } else {
            root.setKey(getmin(root.right()).key());
            root.setValue(getmin(root.right()).value());
            root.setRight(deletemin(root.right()));
        }
    }
    return root;
}
```

删除算法中用到的删除最小节点和获取最小结点：

```
private BinNode<K, V> deletemin(BinNode<K, V> root) {
    if (null == root.left()) {
        return root.right();
    } else {
        root.setLeft(deletemin(root.left()));
        return root;
    }
}

private BinNode<K, V> getmin(BinNode<K, V> root) {
    if (null == root.left()) {
        return root;
    } else {
        return getmin(root.left());
    }
}
```

查找算法：

```
public V search(K key) {
    return (V) searchhelp(root, key);
}

private V searchhelp(BinNode<K, V> root, K key) {
    if (root.key().compareTo(key) > 0) {
        if (root.left() != null) {
            return searchhelp(root.left(), key);
        } else {
            return null;
        }
    } else if (root.key().compareTo(key) < 0) {
        if (root.right() != null) {
            return searchhelp(root.right(), key);
        } else {
            return null;
        }
    } else {
        return (V) root.value();
    }
}
```

更新算法：

```
public boolean update(K key, V value) {
    return updatehelp(root, key, value);
}

public boolean updatehelp(BinNode<K, V> root, K key, V value) {
    if (null != key && null != value && null != root) {
        if (root.key().compareTo(key) > 0) {
            return updatehelp(root.left(), key, value);
        } else if (root.key().compareTo(key) < 0) {
            return updatehelp(root.right(), key, value);
        } else {
            root.setValue(value);
            return true;
        }
    } else {
        return false;
    }
}
```

判断树是否为空，以及清除 BST 的方法：

```
@Override
public boolean isEmpty() {
    return null == root;
}

@Override
public void clear() {
    root = null;
    nodesNum = 0;
}
```

打印树的结构：

```
@Override
public void showStructure(PrintWriter pw) throws IOException {
    pw.println("--------------------");
    if (isEmpty()) {
        pw.println("The BST is empty.");
    } else {
        pw.println("There are " + nodesNum + " nodes in this BST.");
        pw.println("The height of this BST is " + getHeight(root) + ".");
    }
    pw.println("--------------------");
    pw.flush();
}
```

获取树高度的方法：

```
private int getHeight(BinNode<K, V> root) {
    if (null == root) {
        return 0;
    }
    int leftHeight = getHeight(root.left());
    int rightHeight = getHeight(root.right());
    return Math.max(leftHeight, rightHeight) + 1;
}
```

按序打印树中内容：

```
@Override
public void printInorder(PrintWriter pw) throws IOException {
    if (null == root) {
        pw.println("[]");
    } else {
        printHelp(root, pw);
    }
}

private void printHelp(BinNode<K, V> root, PrintWriter pw) {
    if (null == root) {
        return;
    }
    printHelp(root.left(), pw);
    pw.println("[" + root.key() + " -- < " + root.value() + " >]");
    printHelp(root.right(), pw);
    pw.flush();
}
```

## 5、运行结果展示

测试方法正确性：

```java
public static void main(String[] args) {
    MyBST<String, String> tree = new MyBST<String, String>();
    // 测试插入
    tree.insert("37", "37");
    tree.insert("24", "24");
    tree.insert("42", "42");
    tree.insert("7", "7");
    tree.insert("32", "32");
    tree.insert("40", "40");
    tree.insert("42", "42");
    tree.insert("2", "2");
    tree.insert("120", "120");
    // 测试删除
    System.out.println(tree.remove("40"));
    System.out.println(tree.remove("32"));
    // 测试查找
    System.out.println(tree.search("24"));
    System.out.println(tree.search("32"));
    // 测试更新
    System.out.println(tree.update("24", "244"));
    System.out.println(tree.search("24"));
    System.out.println(tree.update("32", "244"));
    // 测试非空
    System.out.println(tree.isEmpty());
    try {
        tree.showStructure(new PrintWriter(System.out));
        tree.printInorder(new PrintWriter(System.out));
        // 测试清除
        tree.clear();
        System.out.println(tree.isEmpty());
    } catch (IOException e) {
```

测试结果：功能实现正确

```
40
32
24
null
true
244
false
false
--------------------
There are 6 nodes in this BST.
The height of this BST is 4.
--------------------
[120 -- < 120 >]
[2 -- < 2 >]
[24 -- < 244 >]
[37 -- < 37 >]
[42 -- < 42 >]
[7 -- < 7 >]
true
```

首先用 pdf 中的案例进行检验：

```
+( overlap , "v. （部分地）重叠" )
-( upstart )
+( sagacious , "adj. 聪明的，睿智的" )
+( vertebrate , "adj·脊椎动物" )
?( ideology )
?( overlap )
+( conflagration , "n. 建筑物或森林大火" )
+( solemnity , "n. 庄严，肃穆" )
+( pretentiousness , "n. 自命不凡" )
-( sagacious )
+( persevere , "v. 坚忍" )
-( affection )
-( vertebrate )
#
```

```
Console  Debug Shell
<terminated> TestMyBST [Java Application] D:\Java\jdk-13\bin\javaw.ex
remove unsuccess ---upstart
search unsuccess ---ideology
search success ----overlap v. （部分地）重叠
remove success ---sagacious adj. 聪明的，睿智的
remove unsuccess ---affection
remove success ---vertebrate adj·脊椎动物
--------------------
There are 5 nodes in this BST.
The height of this BST is 4.
--------------------
```

与 pdf 中相同

给出的测试用例：结果太多了，经过对比应该都相同，这里贴出前 6 个树形结果和最后 6 个树形结果，完整版见附加文件“result.txt”

前 6 个树形结果：

```
--------------------
There are 1 nodes in this BST.
The height of this BST is 1.
--------------------
```

```
--------------------
There are 11 nodes in this BST.
The height of this BST is 5.
--------------------
```

```
--------------------
There are 13 nodes in this BST.
The height of this BST is 5.
--------------------
```

```
--------------------
There are 33 nodes in this BST.
The height of this BST is 8.
--------------------
```

```
--------------------
There are 46 nodes in this BST.
The height of this BST is 9.
--------------------
```

```
--------------------
There are 53 nodes in this BST.
The height of this BST is 9.
--------------------
```

后 6 个树形结果：

```
--------------------
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.
--------------------
```

```
--------------------
There are 410 nodes in this BST.
The height of this BST is 17.
--------------------
```

```
--------------------
There are 412 nodes in this BST.
The height of this BST is 17.
--------------------
```

```
--------------------
There are 417 nodes in this BST.
The height of this BST is 17.
--------------------
```

```
--------------------
There are 424 nodes in this BST.
The height of this BST is 17.
--------------------
```

## 6、总结和收获

通过编程实现算法刚开始还是遇到了许多问题，反应了实践的重要性，在编程之前没有想到有这么多的细节需要处理。

# 实验 2

## 1、题目

使用 BST 为文稿建立单词索引表

我们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，检索的速度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。

## 2、数据结构设计

结点的键为单词，而值为存放的行数，由于会进行大量的增加，所以用线性表来存储所在的行数，即二叉树的结点类型为 BinNode<String,List<Integer>>

## 3、算法设计

文章按照行进行处理，每行一个句子按照空格进行分割，然后使用正则表达式选取分隔后字符串中我们需要的单词，建立的索引表中，每个结点的键为该单词，值为存放行号的线性表。对于某些特殊的单词，如 Mr. Dr. M.D. 建立一个语料库，对于这些单词不进行正则处理。

## 4、主干代码说明

```java
public static void main(String[] args) {
    MyBST<String, List<Integer>> index = new MyBST<>();
    File file = new File("src//com//pickupppp//task3//article.txt");
    BufferedReader br = null;
    Map<String, List<Integer>> map = new HashMap<>();
    try {
        br = new BufferedReader(new InputStreamReader(new FileInputStream(file)));
        String sentence = "";
        int row = 0;
        while ((sentence = br.readLine()) != null) {
            row++;
            sentence = sentence.trim();
            String[] words = sentence.split(" ");
            if (!sentence.equals("")) {
                for (String word : words) {
                    if (!(word.equals("Dr.") || word.equals("Mr.") || word.equals("D.M."))) {
                        Pattern p = Pattern.compile("[a-zA-Z0-9]+'?[a-zA-Z0-9]+");
                        Matcher matcher = p.matcher(word);
                        if (matcher.find()) {
                            word = matcher.group();
                        } else {
                            word = null;
                        }
                    }
                    if (null != word) {
                        if (!map.containsKey(word)) {
                            ArrayList<Integer> list = new ArrayList<>();
                            list.add(row);
                            map.put(word, list);
                        } else {
                            if (!map.get(word).contains(row)) {
                                map.get(word).add(row);
                            }
                        }
                    }
```

## 5、运行结果展示

由于结果很长，这里只截取一部分

```
[00 -- < [15395, 16116, 19790, 20769, 20869, 22581, 23941] >]
[000 -- < [11449, 11531, 13818, 14622, 15731, 16577, 19622, 19680, 19928, 20348, 20354] >]
[10 -- < [528, 13825] >]
[100 -- < [16971] >]
[1000 -- < [20566, 22580] >]
[10s -- < [14702, 17779] >]
[10th -- < [14000] >]
[11 -- < [529, 2028, 12514, 17871, 17949, 21003, 27077] >]
[117 -- < [16091, 16096] >]
[117th -- < [27147, 27589] >]
[12 -- < [530] >]
[126b -- < [23930, 23972, 23985] >]
[129 -- < [1987] >]
[12s -- < [16104] >]
[12th -- < [14001] >]
[13 -- < [1727, 1786, 1797] >]
[14 -- < [14697, 24956] >]
[140 -- < [13363] >]
[14th -- < [26116] >]
[15 -- < [2028, 10577, 24956, 25190] >]
[150 -- < [12760] >]
[16 -- < [29860] >]
[1607 -- < [25767] >]
[1642 -- < [1676] >]
[16A -- < [17636] >]
[17 -- < [11158, 23808] >]
[1840 -- < [32608] >]
```

```
14959[yield -- < [6224] >]
14960[yo -- < [6999, 9263, 9265] >]
14961[yoked -- < [2968] >]
14962[yonder -- < [8164, 8182, 16330, 22319, 23104, 25806, 28537, 32334] >]
14963[you -- < [119, 127, 133, 148, 165, 178, 188, 190, 195, 203, 219, 222, 244, 248, 251, 258,
14964[you'd -- < [16326] >]
14965[you'll -- < [2703, 2711, 2712, 2799, 4215, 4623, 6523, 6751, 16080, 16233, 17015, 21619,
14966[you're -- < [4213, 6065, 19166, 21879, 32045] >]
14967[you've -- < [2714, 2757, 14538, 16054, 25035, 25064, 26664, 27017] >]
14968[young -- < [110, 560, 1807, 2193, 2522, 2882, 2888, 2935, 3082, 3088, 3164, 3173, 3187, 3
14969[younger -- < [4156, 4829, 11885, 18704, 21111, 24983, 26693, 26721, 28515, 28537, 28983,
14970[youngest -- < [2864] >]
14971[youngster -- < [4599, 13727, 19782, 32699, 33191] >]
14972[youngsters -- < [1920] >]
14973[your -- < [195, 222, 350, 637, 657, 669, 678, 713, 761, 847, 882, 902, 950, 963, 1103, 11
14974[yours -- < [174, 1500, 6053, 6064, 6777, 7915, 8478, 8856, 9253, 9681, 10625, 10695, 1101
14975[yourself -- < [119, 958, 1585, 1636, 2731, 2792, 3171, 3428, 3470, 4701, 4850, 4971, 5335
14976[yourselves -- < [5326, 7449, 10935, 11550, 16376, 16686, 23496, 27636] >]
14977[youth -- < [2964, 3124, 3505, 5432, 5653, 7212, 10506, 10845, 10847, 13035, 15354, 15834,
14978[youths -- < [1909] >]
14979[zeal -- < [455, 24051] >]
14980[zealous -- < [21809] >]
14981[zero -- < [14099, 20665] >]
14982[zest -- < [16957] >]
14983[zigzag -- < [7102, 15998] >]
14984[zum -- < [9522] >]
14985
```

我们选择几个验证：

```
9[117 -- < [16091, 16096] >]
```

例如：

```
10[117th -- < [27147, 27589] >]
```

```
16091    "Mrs. Oakshott, 117, Brixton Road--249," read Holmes.
16096    117, Brixton Road, egg and poultry supplier.'"
27147    "The first battalion of the Royal Munsters (which is the old 117th)
27589    was the smartest man in the 117th foot.  We were in India then, in
```

---

再如

```
14971[youngster -- < [4599, 13727, 19782, 32699, 33191] >]
```

```
4599     yard when a ragged youngster asked if there was a cabby there called
13727    when he saw me first I was a youngster of twelve or so. This would be
19782    "'Oh, any old key will fit that bureau. When I was a youngster I have
32699    play cards again. It is unlikely that a youngster like Adair would at
33191    this unfortunate youngster who has thrown himself upon my
```

## 6、总结和收获

利用 BST 建立这样一个索引表看起来很酷，虽然当时实际操作时候由于文本的处理导致很心累，但是成功后感到很开心。

# 实验 3

## 1、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，所以我们必须要完全掌握。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，我们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。下面我们从几个方面对堆进行实践：

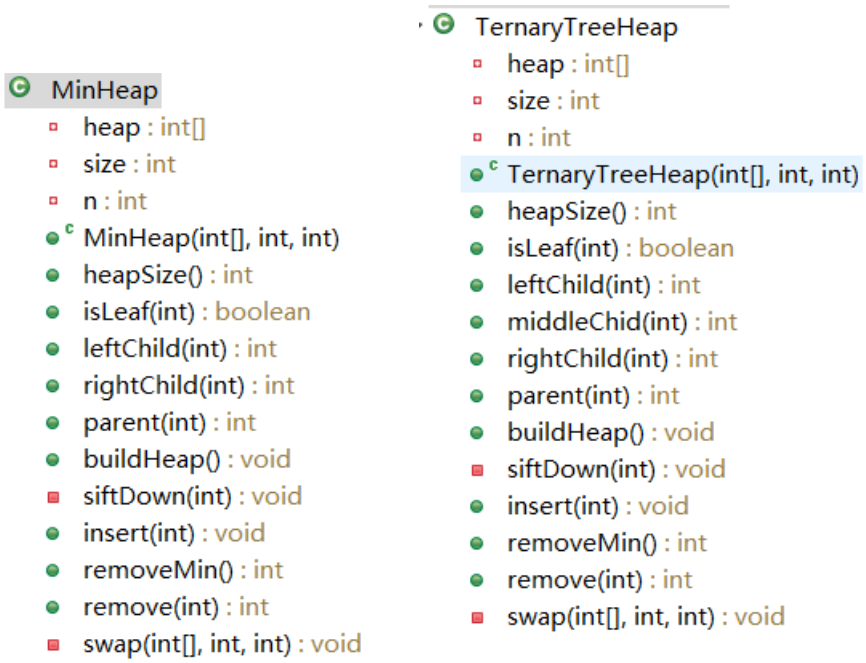
1）参看书中的代码，撰写一个具有 insert、delete、getMin 等方法的 Min_Heap；

2）完成一个使用 1）编写的 Min_Heap 的排序算法；

3）我们所学的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求最大或最小），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。

## 2、数据结构设计

在最小堆中结构与教材相同，在三叉树中多加入一个找中子树




## 3、算法设计

三叉树的最小堆构建时，输入一个数组，从非叶结点开始，依次向前，每个结点于其子节点最小值进行比较，如果大于则交换，直到所有子树均满足父节点小于等于子节点，在三叉树的堆化中需要找到最多三个子节点中的最小值，需要从左子节点开始比较两次。

基于最小堆的堆排序，只需要依次弹出堆中最小值即可。

## 4、主干代码说明

1）二叉树最小堆：

构造函数：

```
public MinHeap(int[] h, int num, int max) {
    heap = h;
    n = num;
    size = max;
    buildHeap();
}
```

查看堆大小、判断叶子结点：

```
public int heapSize() {
    return n;
}

public boolean isLeaf(int pos) {
    return (pos >= n / 2) && (pos < n);
}
```

查找左、右子结点

```
public int leftChild(int pos) {
    if (2 * pos + 1 < n) {
        return 2 * pos + 1;
    } else {
        return -1;
    }
}

public int rightChild(int pos) {
    if (2 * pos + 2 < n) {
        return 2 * pos + 2;
    } else {
        return -1;
    }
}
```

查找父节点：

```
public int parent(int pos) {
    if (pos > 0) {
        return (pos - 1) / 2;
    } else {
        return -1;
    }
}
```

堆化：

```
private void siftDown(int pos) {
    if ((pos >= 0) && (pos < n)) {
        while (!isLeaf(pos)) {
            int j = leftChild(pos);
            if ((j != -1) && (j < (n - 1)) && (heap[j] > heap[j + 1])) {
                j++;
            }
            if (heap[pos] <= heap[j]) {
                return;
            }
            swap(heap, pos, j);
            pos = j;
        }
    }
}
```

插入值、移除最小值：

```
public void insert(int val) {
    if (n < size) {
        int curr = n++;
        heap[curr] = val;
        while ((curr != 0) && (heap[curr] < heap[parent(curr)])) {
            swap(heap, curr, parent(curr));
        }
    }
}

public int removeMin() {
    if (!(n > 0)) {
        throw new IndexOutOfBoundsException();
    }
    swap(heap, 0, --n);
    if (n != 0) {
        siftDown(0);
    }
    return heap[n];
}
```

移除指定值：

```
public int remove(int pos) {
    if (!((pos >= 0) && (pos < n))) {
        throw new IndexOutOfBoundsException();
    }
    swap(heap, pos, --n);
    if (n != 0) {
        siftDown(pos);
    }
    return heap[n];
}
```

2)三叉树最小堆

三叉树与二叉树基本相同，在这里列出寻找中子结点和堆化的函数

寻找中子节点：

```
    public int middleChid(int pos) {
        if (3 * pos + 2 < n) {
            return 3 * pos + 2;
        } else {
            return -1;
        }
    }
```

堆化：

```
private void siftDown(int pos) {
    if ((pos >= 0) && (pos < n)) {
        while (!isLeaf(pos)) {
            int j = leftChild(pos);
            if ((j != -1) && (j < (n - 1))) {
                if (j < n - 2) {
                    if ((heap[j] > heap[j + 1])) {
                        j++;
                    }
                    if ((heap[j] > heap[j + 1])) {
                        j++;
                    }
                } else {
                    if ((heap[j] > heap[j + 1])) {
                        j++;
                    }

                }
            }
            if (heap[pos] <= heap[j]) {
                return;
            }
            swap(heap, pos, j);
            pos = j;
        }
    }
}
```

3）最小堆排序算法

```
public static int[] binaryTreeSort(int[] array) {
    MinHeap heap = new MinHeap(array, array.length, array.length);
    System.out.println("堆中元素为:" + heap);
    int result[] = new int[array.length];
    int i = 0;
    while (heap.heapSize() > 0) {
        result[i] = heap.removeMin();
        i++;
    }
    return result;
}

public static int[] ternaryTreeSort(int[] array) {
    TernaryTreeHeap heap = new TernaryTreeHeap(array, array.length, array.length);
    System.out.println("堆中元素为:" + heap);
    int result[] = new int[array.length];
    int i = 0;
    while (heap.heapSize() > 0) {
        result[i] = heap.removeMin();
        i++;
    }
    return result;
}
```

## 5、运行结果展示

测试最小堆代码：

```
public static void main(String[] args) {
    int[] array = { 2, 3, 1, 9, 7, 10, 8, 5, 4, 0 };
    MinHeap heap = new MinHeap(array, array.length - 1, array.length);
    System.out.println(heap);
    System.out.println(heap.isLeaf(5)); // true
    System.out.println(heap.isLeaf(4)); // false
    heap.insert(11);
    System.out.println(heap);
    heap.removeMin();
    System.out.println(heap);
    heap.remove(3);
    System.out.println(heap);
    System.out.println(heap.leftChild(1));
    System.out.println(heap.rightChild(1));
    System.out.println(heap.parent(3));
}
```

测试结果：

```
<terminated> MinHeap [Java Application] D:\Java\jdk-
[1 ,3 ,2 ,4 ,7 ,10 ,8 ,5 ,9]
true
true
[1 ,3 ,2 ,4 ,7 ,10 ,8 ,5 ,9 ,11]
[2 ,3 ,8 ,4 ,7 ,10 ,11 ,5 ,9]
[2 ,3 ,8 ,5 ,7 ,10 ,11 ,9]
3
4
```

测试结果均正确

测试三叉树代码：

```
public static void main(String[] args) {
    int[] array = { 2, 3, 1, 9, 7, 10, 8, 5, 4, 0 };
    TernaryTreeHeap heap = new TernaryTreeHeap(array, array.length - 1, array.length);
    System.out.println(heap);
    System.out.println(heap.isLeaf(5)); // true
    System.out.println(heap.isLeaf(4)); // true
    System.out.println(heap.isLeaf(3)); // true
    System.out.println(heap.isLeaf(2)); // false
    heap.insert(11);
    System.out.println(heap);
    heap.removeMin();
    System.out.println(heap);
    heap.remove(3);
    System.out.println(heap);
    System.out.println(heap.leftChild(1));
    System.out.println(heap.rightChild(1));
    System.out.println(heap.parent(3));
}
```

测试结果：测试结果均正确

```
<terminated> TernaryTreeHeap [Java Application
[1 ,3 ,2 ,9 ,7 ,10 ,8 ,5 ,4]
true
true
true
false
[1 ,3 ,2 ,9 ,7 ,10 ,8 ,5 ,4 ,11]
[2 ,3 ,4 ,9 ,7 ,10 ,8 ,5 ,11]
[2 ,3 ,4 ,11 ,7 ,10 ,8 ,5]
4
6
0
```

测试最小堆排序代码：

```
public static void main(String[] args) {
    int[] array = { 2, 4, 1, 5, 7, 10, 9, 8, 32, 77, 14 };
    int[] res = binaryTreeSort(array);
    System.out.println("排序结果为:" + Arrays.toString(res));

    int[] array2 = { 2, 4, 1, 5, 7, 10, 9, 8, 32, 77, 14 };
    int[] res2 = ternaryTreeSort(array2);
    System.out.println("排序结果为:" + Arrays.toString(res2));
}
```

运行结果：

```
<terminated> MinHeapSort [Java Application] D:\Java\jdk-13\bin\javaw.ex
堆中元素为:[1 ,4 ,2 ,5 ,7 ,10 ,9 ,8 ,32 ,77 ,14]
排序结果为:[1, 2, 4, 5, 7, 8, 9, 10, 14, 32, 77]
堆中元素为:[1 ,4 ,2 ,5 ,7 ,10 ,9 ,8 ,32 ,77 ,14]
排序结果为:[1, 2, 4, 7, 5, 8, 9, 10, 14, 32, 77]
```

堆中元素符合最小堆的排列，排序结果也正确