

概念
实现
总结

概念

二叉查找树，也称二叉搜索树，或二叉排序树

其定义也比较简单，要么是一颗空树，要么就是具有如下性质的二叉树：

- (1) 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- (2) 若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- (3) 任意节点的左、右子树也分别为二叉查找树；
- (4) 没有键值相等的节点

实现

```
1 package day10.查找;
2
3 import day3.队列.QueueByLink;
4
5 import java.util.*;
6 import java.util.concurrent.ArrayBlockingQueue;
7
8 /**
9  * 二叉查找树
10  */
11 public class BinaryTreeSearch<E> {
12
13     /**
14      * 根节点
15      */
16     private Node root;
17
18     static class Node<E> implements Comparable<Node> {
19         private E data;
20         private Node leftChild;
21         private Node rightChild;
22
23         private Comparable<Node> comparable;
24
25         public Node(E data) {
26             this(data, new Comparable<Node>() {
27                 @Override
28                 public int compareTo(Node o) {
29                     if (o == null) {
```

```

30         return 1;
31     }
32     if (data.hashCode() > o.data.hashCode()) {
33         return 1;
34     } else if (data.hashCode() < o.data.hashCode()) {
35         return -1;
36     }
37     return 0;
38 }
39 });
40 }
41
42 public Node(E data, Comparable<Node> comparable) {
43     this.data = data;
44     this.comparable = comparable;
45 }
46
47 @Override
48 public int compareTo(Node o) {
49     return comparable.compareTo(o);
50 }
51 }
52
53 public void addNode(E data) {
54     Node newNode = new Node(data);
55     if (root == null) {
56         root = newNode;
57         return;
58     }
59     Node current = root;
60     int high = 1;
61     while (current != null) {
62         high++;
63         if (newNode.compareTo(current) < 0) {
64             if (current.leftChild == null) {
65                 current.leftChild = newNode;
66                 return;
67             }
68             current = current.leftChild;
69         } else {
70             if (current.rightChild == null) {
71                 current.rightChild = newNode;
72                 return;
73             }
74             current = current.rightChild;
75         }
76     }
77 }
78
79 //查找节点
80 public Node find(E data) {
81     Node newNode = new Node(data);
82     Node current = root;
83     while (current != null) {

```

```

84         if (current.compareTo(newNode) > 0) { //当前值比查找值大，搜索左子树
85             current = current.leftChild;
86         } else if (current.compareTo(newNode) < 0) { //当前值比查找值小，搜索右子树
87             current = current.rightChild;
88         } else {
89             return current;
90         }
91     }
92     return null; //遍历完整个树没找到，返回null
93 }
94
95 public Node findMax() {
96     return findMax(root);
97 }
98
99 public Node findMax(Node current){
100     while (current.rightChild != null) {
101         current = current.rightChild;
102     }
103     return current;
104 }
105
106 public Node findMin() {
107     return findMin(root);
108 }
109
110 public Node findMin(Node current){
111     while (current.leftChild != null) {
112         current = current.leftChild;
113     }
114     return current;
115 }
116
117 //中序遍历
118 public void infixOrder(Node current) {
119     if (current != null) {
120         infixOrder(current.leftChild);
121         System.out.print(current.data + " ");
122         infixOrder(current.rightChild);
123     }
124 }
125
126 //前序遍历
127 public void preOrder(Node current) {
128     if (current != null) {
129         System.out.print(current.data + " ");
130         infixOrder(current.leftChild);
131         infixOrder(current.rightChild);
132     }
133 }
134
135 //后序遍历
136 public void postOrder(Node current) {
137     if (current != null) {

```

```

138         infixOrder(current.leftChild);
139         infixOrder(current.rightChild);
140         System.out.print(current.data + " ");
141     }
142 }
143
144 //层序遍历
145 public void sequenceOrder(Node current) {
146     Queue<Node> queue = new ArrayBlockingQueue<>(100);
147     while (current != null) {
148         System.out.print(current.data + " ");
149         if (current.leftChild != null) {
150             queue.add(current.leftChild);
151         }
152         if (current.rightChild != null) {
153             queue.add(current.rightChild);
154         }
155         current = queue.poll();
156     }
157 }
158
159 /**
160  * 根据目标节点及当前节点的父节点，返回数组，父节点在第一个，当前节点在第二个
161  * @param data
162  * @return
163  */
164 public Node[] findParentWithCurrent(E data){
165     Node newNode = new Node(data);
166     Node current = root;
167     Node parent = root;
168     Node result = null;
169     while (current != null) {
170         if (current.compareTo(newNode) > 0) { //当前值比查找值大，搜索左子树
171             parent = current;
172             current = current.leftChild;
173         } else if (current.compareTo(newNode) < 0) { //当前值比查找值小，搜索右子树
174             parent = current;
175             current = current.rightChild;
176         } else {
177             result = current;
178             break;
179         }
180     }
181     Node[] nodes = new Node[2];
182     nodes[0] = parent;
183     nodes[1] = result;
184     return nodes;
185 }
186
187
188 // 删除节点：减少复杂度，根节点不能删除
189 public void del(E data) throws Exception {
190     if (data.equals(root.data)){
191         throw new Exception("根节点无法删除");

```

```

192     }
193
194     Node[] nodes = findParentWithCurrent(data);
195     Node result = nodes[1];
196     Node parent = nodes[0];
197
198     if (result == null) {
199         System.out.println("未找到节点");
200         return;
201     }
202
203     if (result.leftChild == null && result.rightChild == null){
204         if (result.equals(parent.leftChild)){
205             parent.leftChild = null;
206         }else {
207             parent.rightChild = null;
208         }
209     }else if (result.leftChild != null && result.rightChild != null){
210         Node successor = findMax(result.leftChild); // or findMin(result.right)
211         result.data = successor.data;          // 更换值，不变化指针
212         del((E) successor.data);
213     }else {
214         Node successor = null;
215         if (result.leftChild != null){
216             successor = result.leftChild;
217         }else {
218             successor = result.rightChild;
219         }
220         if (result.equals(parent.leftChild)){
221             parent.leftChild = successor;
222         }else {
223             parent.rightChild = successor;
224         }
225     }
226 }
227
228 /**
229  * 使用树形结构显示
230  */
231 public void displayTree() {
232     Stack globalStack = new Stack();
233     globalStack.push(root);
234     int nBlank = 32;
235     boolean isRowEmpty = false;
236     String dot = ".....";
237     System.out.println(dot + dot + dot);
238     while (isRowEmpty == false) {
239         Stack localStack = new Stack();
240         isRowEmpty = true;
241         for (int j = 0; j < nBlank; j++){
242             System.out.print("-");
243         }
244         while (globalStack.isEmpty() == false) {
245             Node temp = (Node) globalStack.pop();

```

```

246         if (temp != null) {
247             System.out.print(temp.data);
248
249             localStack.push(temp.leftChild);
250             localStack.push(temp.rightChild);
251             if (temp.leftChild != null || temp.rightChild != null) {
252                 isRowEmpty = false;
253
254             }
255         } else {
256             System.out.print("#!");
257             localStack.push(null);
258             localStack.push(null);
259         }
260
261         //打印一些空格
262         for (int j = 0; j < nBlank * 2 - 2; j++) {
263             System.out.print(" ");
264         }
265     }
266
267
268     System.out.println();
269     nBlank = nBlank / 2;
270     while (localStack.isEmpty() == false) {
271         globalStack.push(localStack.pop());
272     }
273 }
274 System.out.println(dot + dot + dot);
275
276 }
277
278
279 public static void main(String[] args) throws Exception {
280     BinaryTreeSearch<Integer> binaryTreeSearch = new BinaryTreeSearch<>();
281     binaryTreeSearch.addNode(50);
282     binaryTreeSearch.addNode(30);
283     binaryTreeSearch.addNode(60);
284     binaryTreeSearch.addNode(20);
285     binaryTreeSearch.addNode(40);
286     binaryTreeSearch.addNode(65);
287     binaryTreeSearch.addNode(55);
288     binaryTreeSearch.addNode(56);
289
290     binaryTreeSearch.displayTree();
291
292     System.out.println();
293     System.out.println("查找: " + binaryTreeSearch.find(40));
294     System.out.println("max : " + binaryTreeSearch.findMax().data);
295     System.out.println("min : " + binaryTreeSearch.findMin().data);
296     System.out.println();
297
298     // 层序遍历
299     System.out.println("层序遍历");

```

```

300     binaryTreeSearch.sequenceOrder(binaryTreeSearch.root);
301     System.out.println();
302     System.out.println();
303
304     System.out.println("前序遍历");
305     binaryTreeSearch.preOrder(binaryTreeSearch.root);
306     System.out.println();
307     System.out.println();
308
309     System.out.println("后序遍历");
310     binaryTreeSearch.postOrder(binaryTreeSearch.root);
311     System.out.println();
312     System.out.println();
313
314     System.out.println("中序遍历");
315     binaryTreeSearch.infixOrder(binaryTreeSearch.root);
316     System.out.println();
317     System.out.println();
318
319
320     binaryTreeSearch.displayTree();
321     System.out.println("删除40");
322     binaryTreeSearch.del(40);
323     binaryTreeSearch.displayTree();
324
325     //      System.out.println("删除55");
326     //      binaryTreeSearch.del(55);
327     //      binaryTreeSearch.displayTree();
328
329     //      System.out.println("删除60");
330     //      binaryTreeSearch.del(60);
331     //      binaryTreeSearch.displayTree();
332
333
334 }
335
336 }
337
338
339 /* 输出
340
341 .....
342 -----50*****
343 -----30*****60*****
344 -----20*****40*****55*****65*****
345 ---#!*****#!*****#!*****#!*****#!*****56*****#!*****#!*****
346 .....
347
348 查找: day10.查找.BinaryTreeSearch$Node@610455d6
349 max : 65
350 min : 20
351
352 层序遍历
353 50 30 60 20 40 55 65 56
354

```

```

355 前序遍历
356 50 20 30 40 55 56 60 65
357
358 后序遍历
359 20 30 40 55 56 60 65 50
360
361 中序遍历
362 20 30 40 50 55 56 60 65
363
364 .....
365 -----50*****
366 -----30*****60*****
367 -----20*****40*****55*****65*****
368 ---#!*****#!*****#!*****#!*****56*****#!*****#!*****
369 .....
370 删除40
371 .....
372 -----50*****
373 -----30*****60*****
374 -----20*****#!*****55*****65*****
375 ---#!*****#!*****#!*****#!*****56*****#!*****#!*****
376 .....
377 删除60
378 .....
379 -----50*****
380 -----30*****56*****
381 -----20*****#!*****55*****65*****
382 .....
383
384 */

```

总结

1. 二叉树插入的顺序决定最终树的存储

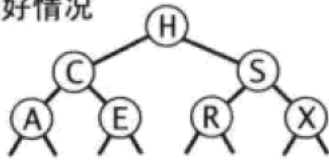
```

1
2
3 插入顺序: 50、65、30、60、20、40、55
4 结果:
5 -----50*****
6 -----30*****65*****
7 -----20*****40*****60*****#!*****
8 ---#!*****#!*****#!*****55*****#!*****#!*****
9
10
11
12 插入顺序: 50、30、60、20、40、65、55
13 结果:
14 .....
15 -----50*****
16 -----30*****60*****
17 -----20*****40*****55*****65*****
18 .....

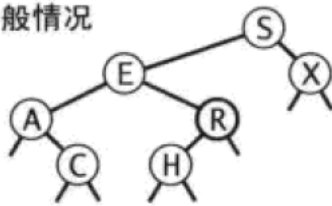
```


2. 理想情况下的插入、查找时间复杂度是 $\log_2(N)$ ，最坏情况下时间复杂度回归到线性表 $O(n)$

最好情况



一般情况



最坏情况

