RB-tree-详解

阅读更多

1 定义

1.1 节点

节点的属性

1. val：关键字
2. left：左孩子节点
3. right：右孩子节点
4. parent：父节点
5. color：颜色

节点的性质（非常重要的5条性质）

1. 每个节点或是红色的，或是黑色的
2. 根节点是黑色的
3. 每个叶节点（nil）是黑色的
4. 如果一个节点是红色的，则它的两个子节点都是黑色的
5. 对每个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点

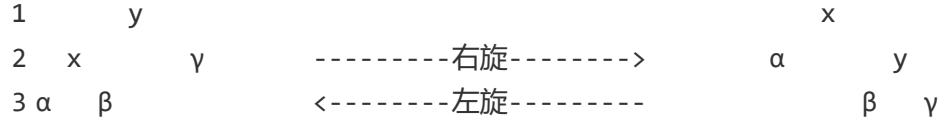
1.2 树

属性

1. nil：哨兵节点
2. root：根节点

2 基本操作

2.1 旋转




```
LEFT-ROTATE(T,x)
y=x.right
x.right=y.left
if y.left≠T.nil
```

```
5      y.left.p=x
6  y.p=x.p
7  if x.p==T.nil
8      T.root=y
9  elseif x==x.p.left
10     x.p.left=y
11 else x.p.right=y
12 y.left=x
13 x.p=y
```

```
1  RIGHT-ROTATE(T,y)
2  x=y.left
3  y.left=x.right
4  if x.right≠T.nil
5      x.right.p=y
6  x.p=y.p
7  if y.p==T.nil
8      root=x
9  elseif y==y.p.left
10     y.p.left=x
11 else y.p.right=x
12 x.right=y
13 y.p=x
```

2.2 插入

```
1  RB-INSERT(T,z)
2  y=T.nil
3  x=T.root
4  while x≠T.nil
5      y=x
6      if z.key<x.key
7          x=x.left
8      else x=x.right
9  z.p=y
10 if y==T.nil
11     T.root=z
12 elseif z.key<y.key
13     y.left=z
14 else y.right=z
15 z.left=T.nil
16 z.right=T.nil
17 z.colcor=RED
18 RB-INSERT-FIXUP(T,z)
```

插入的节点被设定为红色：那么可能会违背性质2或4，但只能是其中之一

- ①：当插入的节点是第一个节点时，此时根节点是红色，违背了性质2，但其子节点与父节点均为T.nil 是黑色，没有违反性质4
- ②：当插入的节点不是根节点，并且其父节点也为红色时，违背了性质4

2.2.1 插入纠正

纠正思路：

- 对于错误①的修正，只需要将根节点设为黑色即可
- 对于错误②的修正，由于z与其父节点均为红色，那么祖父节点必为黑色，根据z的叔节点的颜色状况以及z作为z.p的左右孩子，分三种情况讨论：

当z的父节点是祖父节点的左孩子时：（叔节点为祖父节点的右孩子）

- 情况1：z节点的父节点以及z节点的叔节点都是红色：将z节点的父节点以及叔节点置为黑色，z节点的祖父节点置为红色，继续循环z的祖父节点（z=z.p.p）（z可为z.p的左或右孩子）

```
1         z.p.p(B)                              z.p.p(R)
2     z.p(R)      y(R)     -------->      z.p(B)      y(B)
3 z(R)                                  z(R)
```

- 情况2：z节点的父节点为红色，叔节点为黑色，z为父节点的右孩子，对z的父节点做一次左旋，转为情况3：（旋转前后z所表示的关键字发生改变，但是z的祖父节点没有变）

```
1     z.p.p(B)                              z.p.p(B)
2 z.p(R)      y(B)     -------->      z(R)        y(B)
3    z(R)                           z.p(R)
```

- 情况3：z节点的父节点为红色，叔节点为黑色，z为父节点的左孩子，首先将父节点设为黑色，祖父节点设为红色，然后对祖父节点做一次右旋

```
1       z.p.p(B)                              z.p(B)
2   z.p(R)      y(B)     -------->      z(R)        z.p.p(R)
3 z(R)                                                y(B)
```

- **当z的父节点是祖父节点的右孩子时：（叔节点为祖父节点的左孩子）**：也分为三种情况，与上述三种情况镜像对称，不再赘述

下面给出**插入纠正函数**伪代码

```
1  RB-INSERT-FIXUP(T,z)
```

```
2  while z.p.color==RED//由于z.p是红色，于是z.p.p一定存在，因此访问z.p.p的任何属性者
3      if z.p==z.p.p.left
4          y=z.p.p.right
5          if y.color==RED
6              z.p.color=BLACK
7              y.color=BLACK
8              z.p.p.color=RED
9              z=z.p.p//继续循环
10         else
11             if z==z.p.right
12                 z=z.p
13                 LEFT-ROTATE(T,z)
14             z.p.color=BLACK
15             z.p.p.color=RED
16             RIGHT-ROTATE(T,z.p.p)//循环结束
17     else z.p==z.p.p.right
18         y=z.p.p.left
19         if y.color==RED
20             z.p.color=BLACK
21             y.color=BLACK
22             z.p.p.color=RED
23             z=z.p.p//继续循环
24         else
25             if z==z.p.left
26                 z=z.p
27                 RIGHT-ROTATE(T,z)
28             z.p.color=BLACK
29             z.p.p.color=RED
30             LEFT-ROTATE(T,z.p.p) //循环结束
31 T.root.color=BLACK//针对第一个插入的z，不会进入循环(性质4成立，但性质2破坏，这里纠
```

2.3 节点移植

将v为根节点的子树代替u为根节点的子树

```
1 RB-TRANSPLANT(T,u,v)
2 if u.p==T.nil
3     T.root=v
4 elseif u==u.p.left
5     u.p.left=v
6 else u.p.right=v
7 v.p=u.p//与搜索二叉树相比，这里没有判断，即使v是哨兵，也执行此句,对于移动到y位置的节
```

2.4 删除

z是被删除的节点

y是被删除的节点或者即将移动到被删除节点的节点

- 当z最多只有一个孩子时，y就是被删除的节点
- 当z有两个孩子时，y就是即将移动到被删除节点的节点

x是将要移动到y节点的节点

```
1  RB-DELETE(T,z)
2  y=z
3  y-original-color=y.color
4  if z.left==T.nil
5      x=z.right
6      RB-TRANSPLANT(T,z,z.right)
7  elseif z.right==T.nil
8      x=z.left
9      RB-TRANSPLANT(T,z,z.left)
10 else y=TREE-MINIMUM(z.right)
11     y-original-color=y.color
12     x=y.right
13     if y.p==z
14         x.p=y//使得x为哨兵节点时也成立
15     else RB-TRANSPLANT(T,y,y.right)//即使y.right是哨兵，也会指向y的父节点
16         y.right=z.right
17         y.right.p=y
18     RB-TRANSPLANT(T,z,y)
19     y.left=z.left
20     y.left.p=y
21     y.color=z.color
22 if y-original-color==BLACK
23     RB-DELETE-FIXUP(T,x)
```

此外，还有另一个版本（避免讨论，即不用讨论(y.parent==z)），两个版本等价

```
1  RB-DELETE(T,z)
2  y=z
3  y-original-color=y.color
4  if z.left==T.nil
5      x=z.right
6      RB-TRANSPLANT(T,z,z.right)
7  elseif z.right==T.nil
8      x=z.left
9      RB-TRANSPLANT(T,z,z.left)
10 else y=TREE-MINIMUM(z.right)
11     y-original-color=y.color
12     x=y.right
13     RB-TRANSPLANT(T,y,y.right)//即使y.right是哨兵，也会指向y的父节点
14     y.right=z.right
15     y.right.p=y
16     RB-TRANSPLANT(T,z,y)
```

```
17     y.left=z.left
18     y.left.p=y
19     y.color=z.color
20 if y-original-color==BLACK
21     RB-DELETE-FIXUP(T,x)
```

总结：

1. **删除最终等效为删除一个最多只有一个孩子的节点**
    - 当被删除节点z最多只有只有一个孩子，满足该条规律
    - 当被删除节点z有两个孩子，那么找到该节点z的后继节点y，**此时y节点必然最多只有一个右孩子**，于是将其右孩子y.right transplant到到y节点处以删除y节点，然后再将y节点移动到z节点处，并保持z节点原来的颜色，那么等价于删除y节点
2. 当被删除节点的颜色为红色，那么不会破坏红黑树的性质
3. 当被删除的节点是黑色，那么transplant到该节点的节点x如果是黑色，那么为了保持黑高不变的性质，x必须含有双重黑色，此时又破坏了性质1，需要进行维护矫正

红黑树性质破坏分析

- **当y-original-color为红色时：不会违反红黑树的任何性质**
    - ①：当y为被删除节点时，若y为红色，那么它的父节点为黑色，孩子节点也必为黑色，将孩子移植到该位置不会违反任何性质
    - ②：当y节点为z节点的后继时，若y为红色，那么y节点的父节点以及y节点的右子节点(可能为哨兵)必为黑色，将y.right移植到y的位置，不会违反任何性质；如果z节点是黑色的，那么删除z节点后z的任意祖先的黑高将少一，但是由于将y的颜色设为黑色，做了补偿。如果z节点是红色的，将y节点也设为红色，那么删除z节点不会违反性质5
    - 因此y-original-color为红色时，不会违反红黑树的任何性质
- **当y-original-color为黑色时：可能会违反性质2或4或5**
    - ①：如果y是根节点，而y的一个红色孩子成为新的根节点，违反了性质2

- ②：如果x和x.p是红色，违反了性质4
- ③：在树中删除或移动y将导致先前包含y的简单路径上的黑色节点少1
- 若z节点的孩子均不为T.nil，会违反性质的部分是以y的原位置为根节点的子树(包括其父节点)

2.5 删除纠正

当x是其父亲的左孩子时：x为双重黑色

- 情况1：x的兄弟节点w是红色的(w必有两个黑色的非哨兵子节点，且父亲必为黑色)。将x.p置为红色，w置为黑色，对x.p做一次左旋并更新w，即可将情况1转为234的一种

```
1         x.p(B)                                        w(B)
2 x(B)            w(R)          -------->       x.p(R)          w.r(B)
3             w.l(B)   w.r(B)               x(B)   w.l(B)
```

- 情况2：x的兄弟节点w是黑色，并且w的两个子节点都是黑色(可以是哨兵)。由于x为双重黑色，为了取消x的双重性，将x与w都去掉一层黑色属性，因此x变为单黑，w变为红色，并更新x(将双重属性赋予x的父节点)，并继续循环

```
1         x.p(?)                                        x.p(?)
2 x(B)            w(B)          -------->       x(B)            w(R)
3             w.l(B)   w.r(B)                               w.l(B)   w.r(B)
```

- 情况3：x的兄弟节点w是黑色，w的左孩子是红色，右孩子是黑色。交换w与其左孩子的颜色，对w进行右旋，并更新w，即可转为情况4

```
1         x.p(?)                                        x.p(?)
2 x(B)            w(B)          -------->      x(B)            w.l(B)
3             w.l(R)   w.r(B)                                    w(R)
4                                                                  w.r(B)
```

- 情况4：x的兄弟节点是黑色，且w的右孩子是红色。交换x与x.p的颜色，将w的右孩子置为黑色，并对x.p做一次左旋，即可退出循环

```
1         x.p(?)                                        w(?)
2 x(B)            w(B)          -------->       x.p(B)          w.r(B)
3             w.l(?)   w.r(R)               x(B)    w.l(?)
```

- **当x是其父亲的右孩子时：x为双重黑色**，也可分为四种情况，与上面4种情况镜像对称，不再赘述

```
RB-DELETE-FIXUP(T,x)
while x≠T.root and x.color ==BLACK//若x是红色的，那么将x改为黑色即可
    if x==x.p.left//x可以是哨兵，访问x.p是合法的，因为在Delete中已经设置过
        w=x.p.right
        if w.color==RED
            w.color=BLACK
            x.p.color=RED
            LEFT-ROTATE(T,x.p)
            w=x.p.right
        if w.left.color==BLACK and w.right.color==BLACK
            w.color=RED
            x=x.p
        else
            if w.right.color==BLACK
                w.left.color=BLACK
                w.color=RED
                RIGHT-ROTATE(T,w)
                w=x.p.right
            w.color=x.p.color
            x.p.color=BLACK
            w.right.color=BLACK
            LEFT-ROTATE(T,x.p)
            x=T.root
    elseif x==x.p.right
        w=x.p.left
        if w.color==RED
            w.color=BLACK
            x.p.color=RED
            RIGHT-ROTATE(T,x.p)
            w=x.p.left
        if w.left.color==BLACK and w.right.color==BLACK
            w.color=RED
            x=x.p
        else
            if w.left.color==BLACK
                w.right.color=BLACK
                w.color=RED
                LEFT-ROTATE(T,w)
                w=x.p.left
            w.color=x.p.color
            x.p.color=BLACK
            w.left.color=BLACK
            RIGHT-ROTATE(T,x.p)
            x=T.root
x.color=BLACK
```

3 Java源码

3.1 颜色枚举类型

```
1 package org.liuyehcf.algorithm.datastructure.tree.rbtree;
2
3 /**
4  * Created by HCF on 2017/4/6.
5  */
6 public enum Color {
7     BLACK,
8     RED
9 }
```

3.2 节点定义

```
1  package org.liuyehcf.algorithm.datastructure.tree.rbtree;
2
3  /**
4   * Created by HCF on 2017/4/29.
5   */
6  public class RBTreeNode {
7      int val;
8      RBTreeNode left;
9      RBTreeNode right;
10     RBTreeNode parent;
11     Color color;
12
13     RBTreeNode(int val) {
14         this.val = val;
15     }
16 }
```

3.3 RB-tree实现

```
1   package org.liuyehcf.algorithm.datastructure.tree.rbtree;
2
3   import java.util.*;
4
5   import static org.liuyehcf.algorithm.datastructure.tree.rbtree.Color.BLACK;
6   import static org.liuyehcf.algorithm.datastructure.tree.rbtree.Color.RED;
7
8   /**
9    * Created by HCF on 2017/4/6.
10  */
```

```
11
12 public class RBTree {
13     private RBTreeNode nil;
14
15     private RBTreeNode root;
16     private boolean rule5;
17
18     public RBTree() {
19         nil = new RBTreeNode(0);
20         nil.color = BLACK;
21         nil.left = nil;
22         nil.right = nil;
23         nil.parent = nil;
24
25         root = nil;
26     }
27
28     public void insert(int val) {
29         RBTreeNode x = root;
30         RBTreeNode y = nil;
31         RBTreeNode z = new RBTreeNode(val);
32         while (x != nil) {
33             y = x;
34             if (z.val < x.val) {
35                 x = x.left;
36             } else {
37                 x = x.right;
38             }
39         }
40         z.parent = y;
41         z.left = nil;
42         z.right = nil;
43         z.color = RED;
44         if (y == nil) {
45             root = z;
46         } else if (z.val < y.val) {
47             y.left = z;
48         } else {
49             y.right = z;
50         }
51         insertFix(z);
52         if (!check()) throw new RuntimeException();
53     }
54
55     private void insertFix(RBTreeNode x) {
56         while (x.parent.color == RED) {
57             if (x.parent == x.parent.parent.left) {
```

```
58                 RBTreeNode y = x.parent.parent.right;
59                 if (y.color == RED) {
60                     x.parent.color = BLACK;
61                     y.color = BLACK;
62                     x.parent.parent.color = RED;
63                     x = x.parent.parent;
64                 } else {
65                     if (x == x.parent.right) {
66                         x = x.parent;
67                         leftRotate(x);
68                     }
69                     x.parent.color = BLACK;
70                     x.parent.parent.color = RED;
71                     rightRotate(x.parent.parent);
72                 }
73             } else {
74                 RBTreeNode y = x.parent.parent.left;
75                 if (y.color == RED) {
76                     x.parent.color = BLACK;
77                     y.color = BLACK;
78                     x.parent.parent.color = RED;
79                     x = x.parent.parent;
80                 } else {
81                     if (x == x.parent.left) {
82                         x = x.parent;
83                         rightRotate(x);
84                     }
85                     x.parent.color = BLACK;
86                     x.parent.parent.color = RED;
87                     leftRotate(x.parent.parent);
88                 }
89             }
90         }
91         root.color = BLACK;
92     }
93 
94     private void leftRotate(RBTreeNode x) {
95         RBTreeNode y = x.right;
96         x.right = y.left;
97         if (y.left != nil) {
98             y.left.parent = x;
99         }
100         y.parent = x.parent;
101         if (x.parent == nil) {
102             root = y;
103         } else if (x == x.parent.left) {
104             x.parent.left = y;
```

```
105         } else {
106             x.parent.right = y;
107         }
108         y.left = x;
109         x.parent = y;
110     }
111 
112     private void rightRotate(RBTreeNode y) {
113         RBTreeNode x = y.left;
114         y.left = x.right;
115         if (x.right != nil) {
116             x.right.parent = y;
117         }
118         x.parent = y.parent;
119         if (y.parent == nil) {
120             root = x;
121         } else if (y == y.parent.left) {
122             y.parent.left = x;
123         } else {
124             y.parent.right = x;
125         }
126         x.right = y;
127         y.parent = x;
128     }
129 
130     public void insert(int[] vals) {
131         for (int val : vals) {
132             insert(val);
133         }
134     }
135 
136     public int max() {
137         RBTreeNode x = max(root);
138         if (x == nil) throw new RuntimeException();
139         return x.val;
140     }
141 
142     private RBTreeNode max(RBTreeNode x) {
143         while (x.right != nil) {
144             x = x.right;
145         }
146         return x;
147     }
148 
149     public int min() {
150         RBTreeNode x = min(root);
151         if (x == nil) throw new RuntimeException();
```

```
152         return x.val;
153     }
154 
155     private RBTreeNode min(RBTreeNode x) {
156         while (x.left != nil) {
157             x = x.left;
158         }
159         return x;
160     }
161 
162     public boolean search(int val) {
163         RBTreeNode x = search(root, val);
164         return x != nil;
165     }
166 
167     private RBTreeNode search(RBTreeNode x, int val) {
168         while (x != nil) {
169             if (x.val == val) return x;
170             else if (val < x.val) {
171                 x = x.left;
172             } else {
173                 x = x.right;
174             }
175         }
176         return nil;
177     }
178 
179     public void delete(int val) {
180         RBTreeNode z = search(root, val);
181         if (z == nil) throw new RuntimeException();
182 
183         RBTreeNode y = z;//y代表"被删除"的节点
184         RBTreeNode x = nil;//x代表移动到"被删除"节点的节点
185         Color yOriginColor = y.color;
186         if (z.left == nil) {
187             x = z.right;
188             transplant(z, z.right);
189         } else if (z.right == nil) {
190             x = z.left;
191             transplant(z, z.left);
192         } else {
193             y = min(z.right);
194             yOriginColor = y.color;
195             x = y.right;
196             transplant(y, x);
197 
198             //TODO 以下6句改为z.val=y.val也是可以的
```

```
199             y.right = z.right;
200             y.right.parent = y;
201 
202             y.left = z.left;
203             y.left.parent = y;
204 
205             transplant(z, y);
206             y.color = z.color;
207         }
208 
209         if (yOriginColor == BLACK) {
210             deleteFix(x);
211         }
212 
213         if (!check()) throw new RuntimeException();
214     }
215 
216     private void transplant(RBTreeNode u, RBTreeNode v) {
217         v.parent = u.parent;
218         if (u.parent == nil) {
219             root = v;
220         } else if (u == u.parent.left) {
221             u.parent.left = v;
222         } else {
223             u.parent.right = v;
224         }
225     }
226 
227     private void deleteFix(RBTreeNode x) {
228         while (x != root && x.color == BLACK) {
229             if (x == x.parent.left) {
230                 RBTreeNode w = x.parent.right;
231                 if (w.color == RED) {
232                     w.color = BLACK;
233                     x.parent.color = RED;
234                     leftRotate(x.parent);
235                     w = x.parent.right;
236                 }
237                 if (w.left.color == BLACK && w.right.color == BLACK) {
238                     w.color = RED;
239                     x = x.parent;
240                     //这里是可能直接退出循环的,此时x若为红色，那么x就是红色带额外
241                 } else {
242                     if (w.left.color == RED) {
243                         w.left.color = BLACK;
244                         w.color = RED;
245                         rightRotate(w);
```

```
246                     w = x.parent.right;
247                 }
248                 w.color = x.parent.color;
249                 x.parent.color = BLACK;
250                 w.right.color = BLACK;
251                 leftRotate(x.parent);
252                 x = root;
253             }
254 
255         } else {
256             RBTreeNode w = x.parent.left;
257             if (w.color == RED) {
258                 w.color = BLACK;
259                 x.parent.color = RED;
260                 rightRotate(x.parent);
261                 w = x.parent.left;
262             }
263             if (w.left.color == BLACK && w.right.color == BLACK) {
264                 w.color = RED;
265                 x = x.parent;
266                 //这里是可能直接退出循环的,此时x若为红色，那么x就是红色带额外
267             } else {
268                 if (w.right.color == RED) {
269                     w.right.color = BLACK;
270                     w.color = RED;
271                     leftRotate(w);
272                     w = x.parent.left;
273                 }
274                 w.color = x.parent.color;
275                 x.parent.color = BLACK;
276                 w.left.color = BLACK;
277                 rightRotate(x.parent);
278                 x = root;
279             }
280         }
281     }
282     x.color = BLACK;
283 }
284 
285 private boolean check() {
286     if (root.color == RED) return false;
287     if (nil.color == RED) return false;
288     if (!checkRule4(root)) return false;
289     rule5 = true;
290     checkRule5(root);
291     if (!rule5) return false;
292     return true;
```

```
293     }
294
295     private boolean checkRule4(RBTreeNode root) {
296         if (root == nil) return true;
297         if (root.color == RED &&
298                 (root.left.color == RED || root.right.color == RED))
299             return false;
300         return checkRule4(root.left) && checkRule4(root.right);
301     }
302
303     private int checkRule5(RBTreeNode root) {
304         if (root == nil) return 1;
305         int leftBlackHigh = checkRule5(root.left);
306         int rightBlackHigh = checkRule5(root.right);
307         if (leftBlackHigh != rightBlackHigh) {
308             rule5 = false;
309             return -1;
310         }
311         return leftBlackHigh + (root.color == BLACK ? 1 : 0);
312     }
313
314     public void preOrderTraverse() {
315         StringBuilder sbRecursive = new StringBuilder();
316         StringBuilder sbStack = new StringBuilder();
317         StringBuilder sbElse = new StringBuilder();
318
319         preOrderTraverseRecursive(root, sbRecursive);
320         preOrderTraverseStack(sbStack);
321         preOrderTraverseElse(sbElse);
322
323         System.out.println(sbRecursive.toString());
324         System.out.println(sbStack.toString());
325         System.out.println(sbElse.toString());
326
327         if (!sbRecursive.toString().equals(sbStack.toString()) ||
328                 !sbRecursive.toString().equals(sbElse.toString()))
329             throw new RuntimeException();
330     }
331
332     private void preOrderTraverseRecursive(RBTreeNode root, StringBuilder s
333         if (root != nil) {
334             sb.append(root.val + ", ");
335             preOrderTraverseRecursive(root.left, sb);
336             preOrderTraverseRecursive(root.right, sb);
337         }
338     }
339
```

```
340     private void preOrderTraverseStack(StringBuilder sb) {
341         LinkedList<RBTreeNode> stack = new LinkedList<RBTreeNode>();
342         RBTreeNode cur = root;
343         while (cur != nil || !stack.isEmpty()) {
344             while (cur != nil) {
345                 sb.append(cur.val + ", ");
346                 stack.push(cur);
347                 cur = cur.left;
348             }
349             if (!stack.isEmpty()) {
350                 RBTreeNode peek = stack.pop();
351                 cur = peek.right;
352             }
353         }
354     }
355 
356     private void preOrderTraverseElse(StringBuilder sb) {
357         RBTreeNode cur = root;
358         RBTreeNode pre = nil;
359         while (cur != nil) {
360             if (pre == cur.parent) {
361                 sb.append(cur.val + ", ");
362                 pre = cur;
363                 if (cur.left != nil) {
364                     cur = cur.left;
365                 } else if (cur.right != nil) {
366                     cur = cur.right;
367                 } else {
368                     cur = cur.parent;
369                 }
370             } else if (pre == cur.left) {
371                 pre = cur;
372                 if (cur.right != nil) {
373                     cur = cur.right;
374                 } else {
375                     cur = cur.parent;
376                 }
377             } else {
378                 pre = cur;
379                 cur = cur.parent;
380             }
381         }
382     }
383 
384     public void inOrderTraverse() {
385         StringBuilder sbRecursive = new StringBuilder();
386         StringBuilder sbStack = new StringBuilder();
```

```
387         StringBuilder sbElse = new StringBuilder();
388 
389         inOrderTraverseRecursive(root, sbRecursive);
390         inOrderTraverseStack(sbStack);
391         inOrderTraverseElse(sbElse);
392 
393         System.out.println(sbRecursive.toString());
394         System.out.println(sbStack.toString());
395         System.out.println(sbElse.toString());
396 
397         if (!sbRecursive.toString().equals(sbStack.toString()) ||
398                 !sbRecursive.toString().equals(sbElse.toString()))
399             throw new RuntimeException();
400     }
401 
402     private void inOrderTraverseRecursive(RBTreeNode root, StringBuilder sb
403         if (root != nil) {
404             inOrderTraverseRecursive(root.left, sb);
405             sb.append(root.val + ", ");
406             inOrderTraverseRecursive(root.right, sb);
407         }
408     }
409 
410     private void inOrderTraverseStack(StringBuilder sb) {
411         LinkedList<RBTreeNode> stack = new LinkedList<RBTreeNode>();
412         RBTreeNode cur = root;
413         while (cur != nil || !stack.isEmpty()) {
414             while (cur != nil) {
415                 stack.push(cur);
416                 cur = cur.left;
417             }
418             if (!stack.isEmpty()) {
419                 RBTreeNode peek = stack.pop();
420                 sb.append(peek.val + ", ");
421                 cur = peek.right;
422             }
423         }
424     }
425 
426     private void inOrderTraverseElse(StringBuilder sb) {
427         RBTreeNode cur = root;
428         RBTreeNode pre = nil;
429         while (cur != nil) {
430             if (pre == cur.parent) {
431                 pre = cur;
432                 if (cur.left != nil) {
433                     cur = cur.left;
```

```
434                 } else if (cur.right != nil) {
435                     sb.append(cur.val + ", ");
436                     cur = cur.right;
437                 } else {
438                     sb.append(cur.val + ", ");
439                     cur = cur.parent;
440                 }
441             } else if (pre == cur.left) {
442                 pre = cur;
443                 sb.append(cur.val + ", ");
444                 if (cur.right != nil) {
445                     cur = cur.right;
446                 } else {
447                     cur = cur.parent;
448                 }
449             } else {
450                 pre = cur;
451                 cur = cur.parent;
452             }
453         }
454     }
455 
456     public void postOrderTraverse() {
457         StringBuilder sbRecursive = new StringBuilder();
458         StringBuilder sbStack1 = new StringBuilder();
459         StringBuilder sbStack2 = new StringBuilder();
460         StringBuilder sbStack3 = new StringBuilder();
461         StringBuilder sbElse = new StringBuilder();
462 
463         postOrderTraverseRecursive(root, sbRecursive);
464         postOrderTraverseStack1(sbStack1);
465         postOrderTraverseStack2(sbStack2);
466         postOrderTraverseStack3(sbStack3);
467         postOrderTraverseElse(sbElse);
468 
469         System.out.println(sbRecursive.toString());
470         System.out.println(sbStack1.toString());
471         System.out.println(sbStack2.toString());
472         System.out.println(sbStack3.toString());
473         System.out.println(sbElse.toString());
474 
475         if (!sbRecursive.toString().equals(sbStack1.toString()) ||
476                 !sbRecursive.toString().equals(sbStack2.toString()) ||
477                 !sbRecursive.toString().equals(sbStack3.toString()) ||
478                 !sbRecursive.toString().equals(sbElse.toString()))
479             throw new RuntimeException();
480     }
```

```
481
482     private void postOrderTraverseRecursive(RBTreeNode root, StringBuilder
483         if (root != nil) {
484             postOrderTraverseRecursive(root.left, sb);
485             postOrderTraverseRecursive(root.right, sb);
486             sb.append(root.val + ", ");
487         }
488     }
489
490     private void postOrderTraverseStack1(StringBuilder sb) {
491         LinkedList<RBTreeNode> stack = new LinkedList<RBTreeNode>();
492         RBTreeNode cur = root;
493         while (cur != nil || !stack.isEmpty()) {
494             while (cur != nil) {
495                 sb.insert(0, cur.val + ", ");
496                 stack.push(cur);
497                 cur = cur.right;
498             }
499             if (!stack.isEmpty()) {
500                 RBTreeNode peek = stack.pop();
501                 cur = peek.left;
502             }
503         }
504     }
505
506     private void postOrderTraverseStack2(StringBuilder sb) {
507         LinkedList<RBTreeNode> stack = new LinkedList<RBTreeNode>();
508         RBTreeNode cur = root;
509         Map<RBTreeNode, Integer> map = new HashMap<RBTreeNode, Integer>();
510         while (cur != nil || !stack.isEmpty()) {
511             while (cur != nil) {
512                 stack.push(cur);
513                 map.put(cur, 1);
514                 cur = cur.left;
515             }
516             if (!stack.isEmpty()) {
517                 RBTreeNode peek = stack.pop();
518                 if (map.get(peek) == 2) {
519                     sb.append(peek.val + ", ");
520                     cur = nil;
521                 } else {
522                     stack.push(peek);
523                     map.put(peek, 2);
524                     cur = peek.right;
525                 }
526             }
527         }
```

```
528     }
529
530     private void postOrderTraverseStack3(StringBuilder sb) {
531         RBTreeNode pre = nil;
532         LinkedList<RBTreeNode> stack = new LinkedList<RBTreeNode>();
533         stack.push(root);
534         while (!stack.isEmpty()) {
535             RBTreeNode peek = stack.peek();
536             if (peek.left == nil && peek.right == nil || pre.parent == peek
537                 pre = peek;
538                 sb.append(peek.val + ", ");
539                 stack.pop();
540             } else {
541                 if (peek.right != nil) {
542                     stack.push(peek.right);
543                 }
544                 if (peek.left != nil) {
545                     stack.push(peek.left);
546                 }
547             }
548         }
549     }
550
551     private void postOrderTraverseElse(StringBuilder sb) {
552         RBTreeNode cur = root;
553         RBTreeNode pre = nil;
554         while (cur != nil) {
555             if (pre == cur.parent) {
556                 pre = cur;
557                 if (cur.left != nil) {
558                     cur = cur.left;
559                 } else if (cur.right != nil) {
560                     cur = cur.right;
561                 } else {
562                     sb.append(cur.val + ", ");
563                     cur = cur.parent;
564                 }
565             } else if (pre == cur.left) {
566                 pre = cur;
567                 if (cur.right != nil) {
568                     cur = cur.right;
569                 } else {
570                     sb.append(cur.val + ", ");
571                     cur = cur.parent;
572                 }
573             } else {
574                 sb.append(cur.val + ", ");
```

```
575                 pre = cur;
576                 cur = cur.parent;
577             }
578         }
579     }
580 
581     public static void main(String[] args) {
582         long start = System.currentTimeMillis();
583 
584         Random random = new Random();
585 
586         int TIMES = 10;
587 
588         while (--TIMES > 0) {
589             System.out.println("剩余测试次数: " + TIMES);
590             RBTree rbTree = new RBTree();
591 
592             int N = 10000;
593             int M = N / 2;
594 
595             Set<Integer> set = new HashSet<Integer>();
596             for (int i = 0; i < N; i++) {
597                 set.add(random.nextInt());
598             }
599 
600             List<Integer> list = new ArrayList<Integer>(set);
601             Collections.shuffle(list, random);
602             //插入N个数据
603             for (int i : list) {
604                 rbTree.insert(i);
605             }
606 
607 //rbTree.preOrderTraverse();
608 //rbTree.inOrderTraverse();
609 //rbTree.postOrderTraverse();
610 
611             //删除M个数据
612             Collections.shuffle(list, random);
613 
614             for (int i = 0; i < M; i++) {
615                 set.remove(list.get(i));
616                 rbTree.delete(list.get(i));
617             }
618 
619             //再插入M个数据
620             for (int i = 0; i < M; i++) {
621                 int k = random.nextInt();
```

```
622                 set.add(k);
623                 rbTree.insert(k);
624             }
625             list.clear();
626             list.addAll(set);
627             Collections.shuffle(list, random);
628 
629             //再删除所有元素
630             for (int i : list) {
631                 rbTree.delete(i);
632             }
633         }
634         long end = System.currentTimeMillis();
635         System.out.println("Run time: " + (end - start) / 1000 + "s");
636     }
637 }
```