红黑树

宋星霖

2024 年 2 月 15 日

1 结构讲解

红黑树是一种平衡二叉树排序树，其有以下几种平衡条件：

1. 每个节点非黑即红
2. 根节点是黑色
3. 叶节点（NIL）是黑色
4. 如果一个节点是红色，则它的两个子节点都是黑色的
5. 从根节点出发到所有叶节点路径上，黑色节点数量相同

例如图 1 展示的就是一颗红黑树。

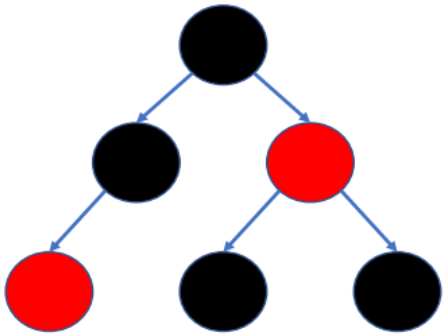


图 1: 红黑树

红黑树最重要的两条平衡条件是最后面两条性质，红黑树根据这两条性质判断是否失衡。

下面是关于红黑树平衡条件的几个问题：

Q: 红黑树中，最长路径和最短路径长度的关系？

A: 根据红黑树最后两条平衡条件，最长路径全是黑色节点，最短路径是红黑相间，所以最长是最短路径的两倍。

Q: 怎么理解条件 3 中的 NIL 节点？

A:NIL 节点就像中文中的标点符号，平时注意不到它，如果没有它，就会很麻烦。

2 插入和删除

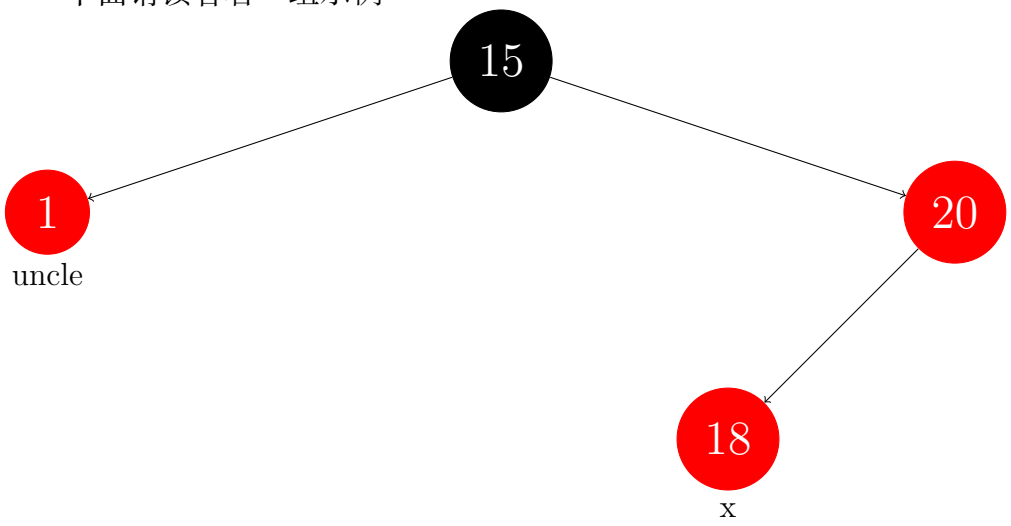
红黑树的插入和删除本质上和普通的二叉排序树没有什么不同，只不过每一次插入和删除之后需要进行插入调整和删除调整。

值得注意的是，插入调整站在祖父节点处，删除调整站在父节点处。并且，调整前后应当保证黑色节点数量不变。

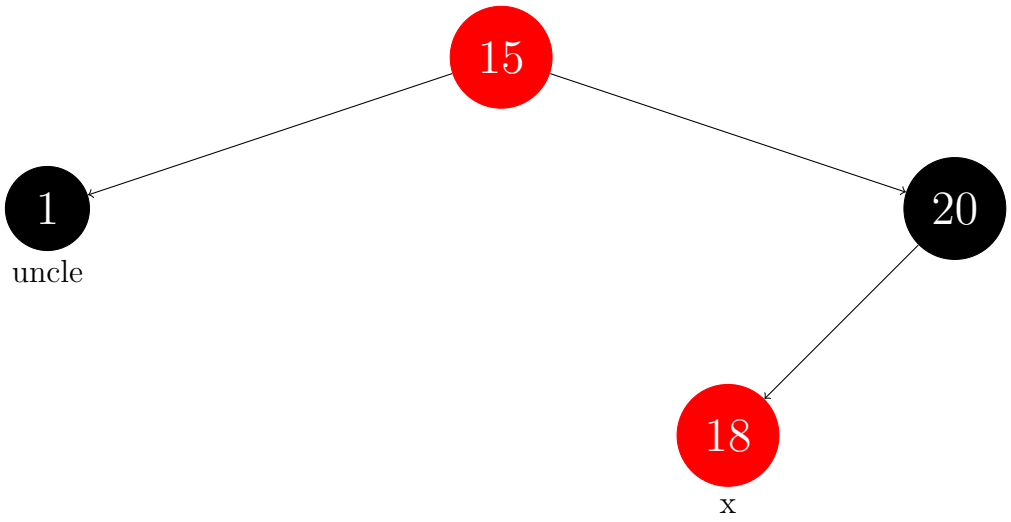
2.1 插入 [4]

在插入之前，请读者思考一个问题：新插入的节点应该是什么颜色的?
事实上，新插入的节点应该是红色的。因为根据红黑树的第 5 条平衡条件，如果插入黑色节点，这条路径上的黑色节点会多一个，必定会引发插入调整，而插入红色节点有概率不用插入调整，所以应当插入红色节点。

下面请读者看一组示例：



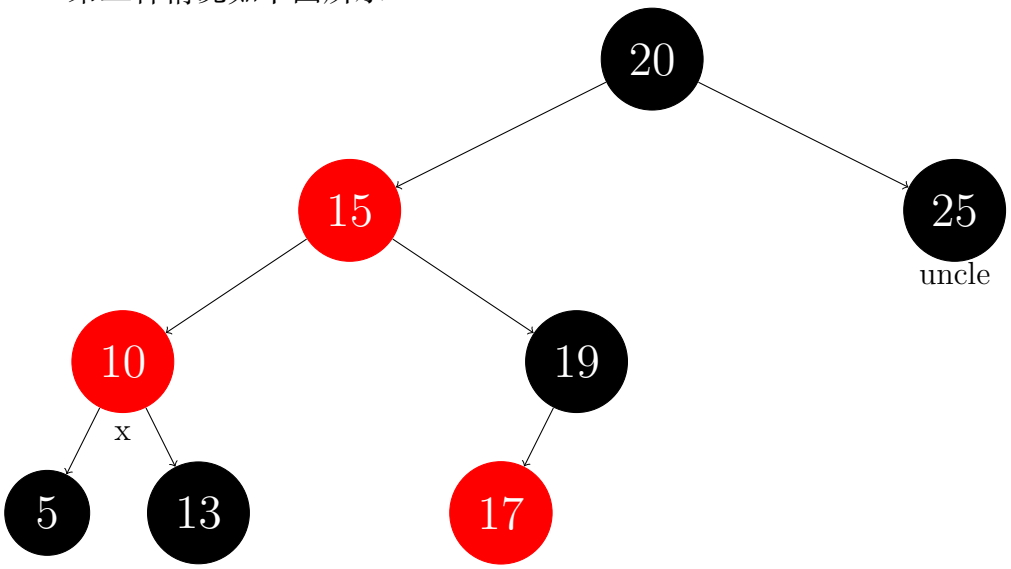
这颗红黑树违反了红黑树的第 4 条平衡性质，所以这颗二叉树需要插入调整。如果红黑树是上文中所绘制的情况，即叔父节点是红色节点，可以将祖父节点变为红色，父节点变为黑色，这种操作叫做红色上浮。调整后的红黑树如下图所示：



虽然此时祖父节点为红色，但是祖父节点不一定为整颗红黑树的根节点，即使是，也可以将

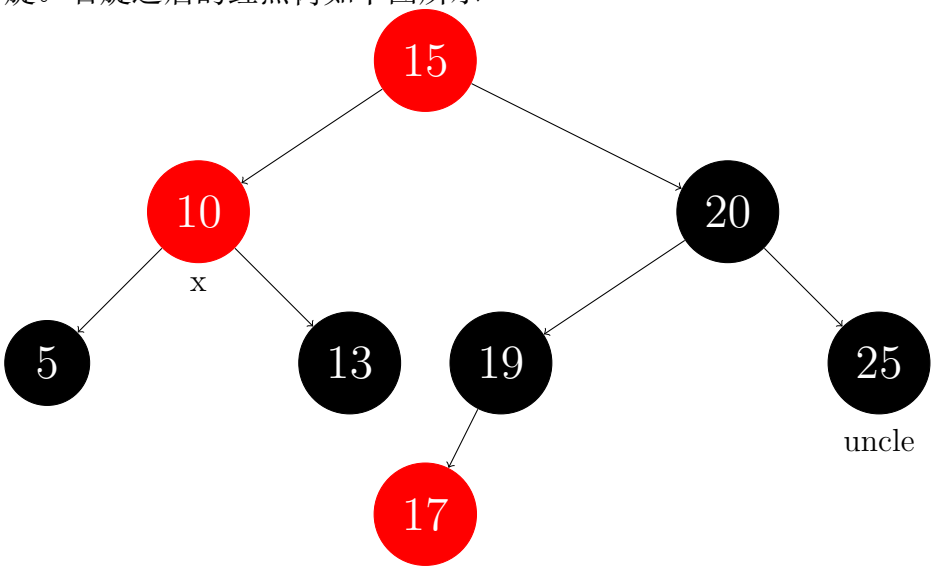
根节点手动置为黑色。

第二种情况如下图所示：



在这颗红黑树中，10 和 15 节点是连续的红色节点，违反了红黑树的第 4 条平衡性质。所以需要插入调整。

对于这种叔父节点是黑色的情况，像 AVL 树一样有 4 种子情况：祖父节点的左孩子（Left）是红色，祖父节点的左孩子的左孩子（Left）是红色 (LL)；祖父节点的左孩子（Left）是红色，祖父节点的左孩子的右孩子（Right）是红色 (LR)；祖父节点的右孩子（Right）是红色，祖父节点的右孩子的左孩子（Left）是红色 (RL)；祖父节点的右孩子（Right）是红色，祖父节点的右孩子的右孩子（Right）是红色 (RR)。对于 LL 类型，可以像 AVL 树一样，先进行右旋。右旋之后的红黑树如下图所示：



接下来，请读者思考一个问题：哪些节点的颜色是固定的？哪些节点的颜色是这颗树的特例[1]？

解答：

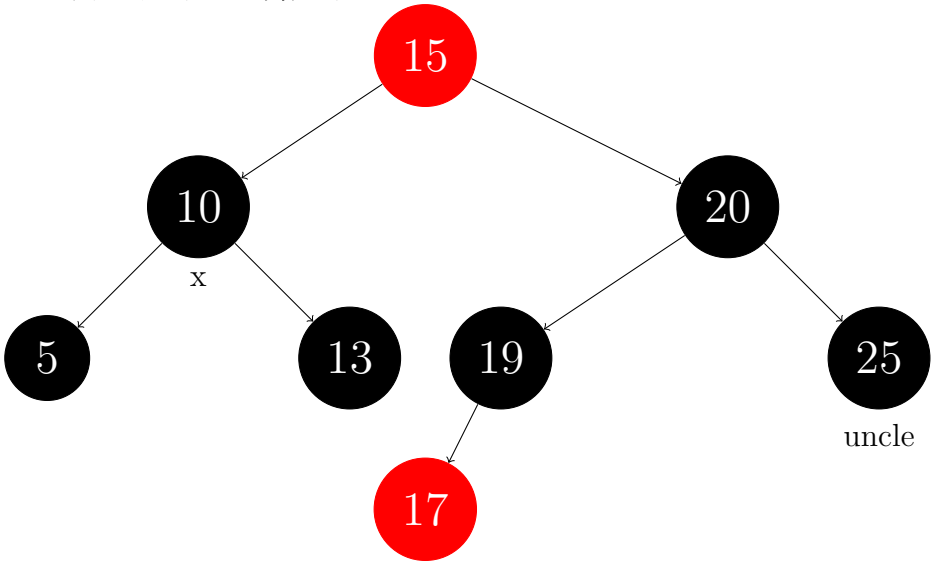
首先，根据失衡类型，10、15、25 节点的颜色是确定的。因为 15 一定是红色，所以 20，19 一定是黑色。因为 10 一定是红色，所以 5，13 一定是黑色。综上所述，20、15、25、10、5、13、19 的颜色是固定的，17 是特例。

[1]特例指这个节点的颜色在其他情况中可能会发生变化

所以，我们只能修改 20、15、25、10、5、13、19 节点。

为了使每条路线上的黑色节点数量不变，可以将 10、20 变为黑色，15 变为红色，也就是红色上浮。这样，调整前每条路径上有 2 个黑色节点，调整后每条路径上也有 2 个黑色节点。

调整后的红黑树如下图所示:



类似的，LR、RL、RR 的方法是先像 AVL 树一样进行对应的旋转，再进行红色上浮。

事实上，在最后一步也可以采用红色下沉，即父节点变为红色，祖父节点变为黑色。

2.2 删除 [2]

因为删除度为 2 的节点可以转化为删除度为 1 或度为 0 的节点，所以下面只讨论删除度为 1 或度为 0 的节点的情况。

表 1: 删除方式

颜色 / 度	红色	黑色
0	直接删除	生成双重黑节点，触发删除调整
1	不存在	删除节点，唯一子孩子变为黑色，提升唯一子树。

不存在度为 1 的红色节点是因为红色结点的左右孩子都是黑色，如果存在度为 1 的红色节点，就会使得某一条路径少一条路径。同理，度为 1 的黑色节点的唯一子孩子一定是红色，因为如果度为 1 的黑色节点的唯一子孩子是黑色，那么就会有一条路径黑色节点少一个。而删除一个度为 0 的黑色节点时，某一条路径上的黑色节点就会少一个。这个无处安放的黑色节点就会加在 NIL 身上，从而产生双重黑节点。所以，删除调整的目的就是删除双重黑节点。

下面是删除调整的几种情况：

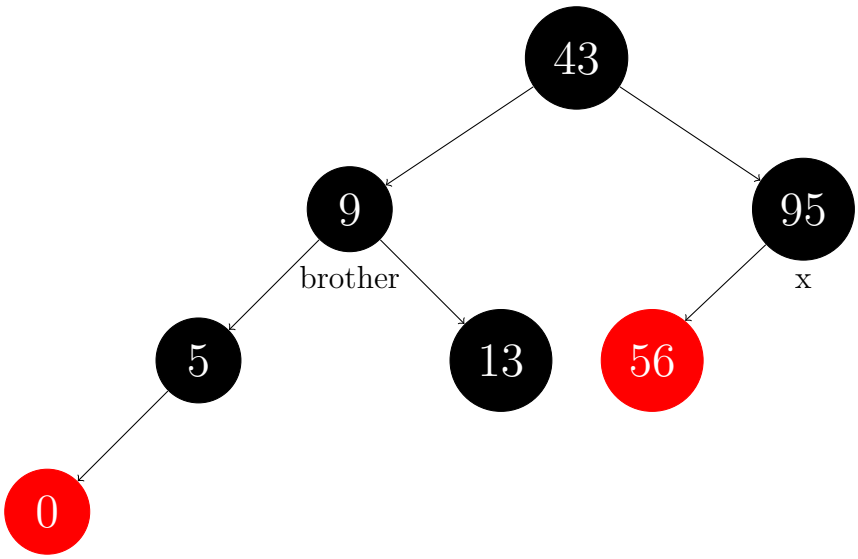
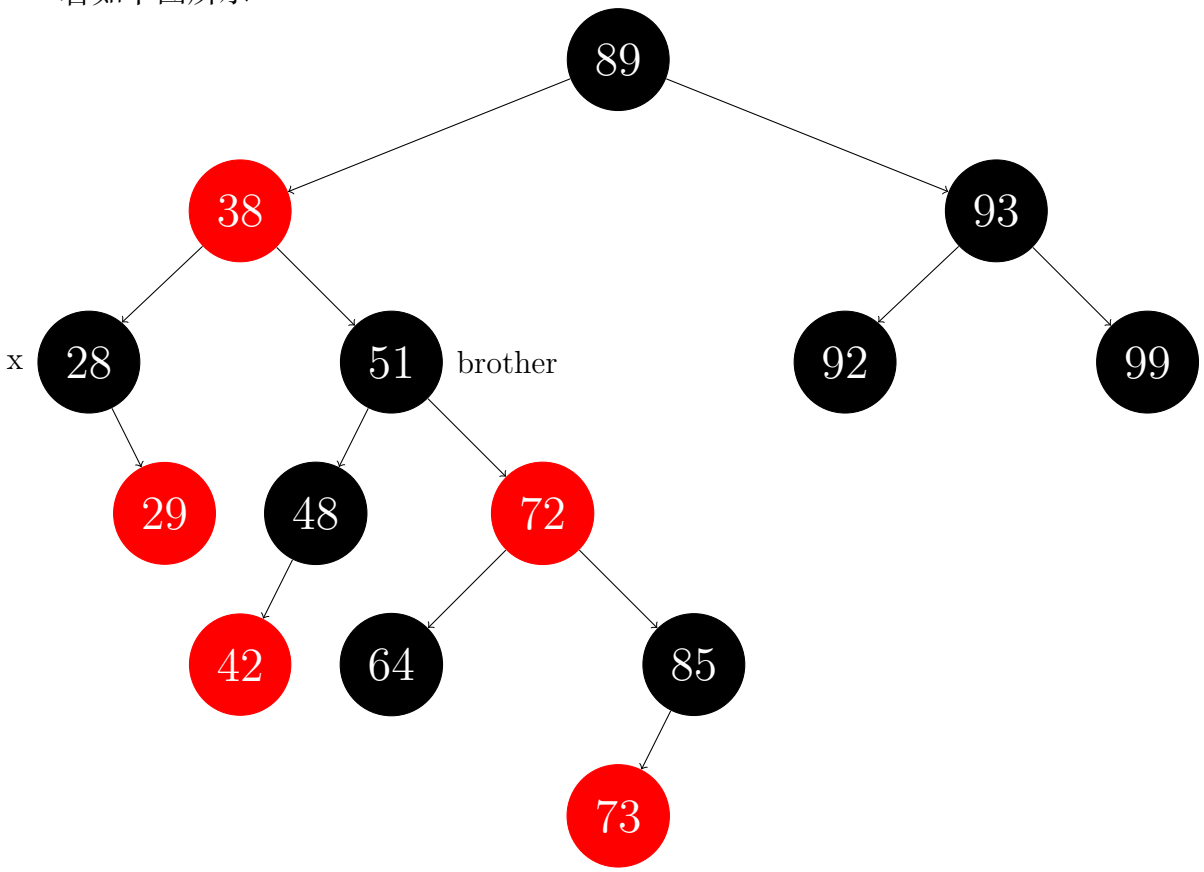


图 2: 删除调整第一种情况

如图 2 所示，若双重黑节点的兄弟节点是黑色，且没有红色子孩子则可以将父节点颜色加 1，即黑变双重黑，红变黑，兄弟节点和双重黑节点颜色减 1。

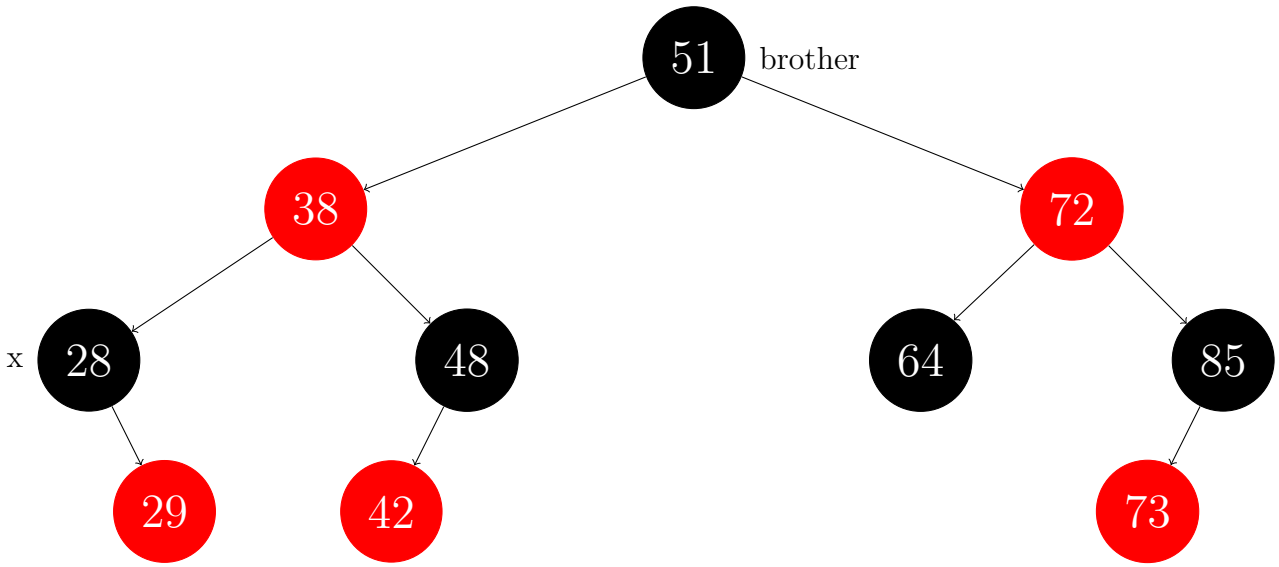
若如下图所示：



像这种兄弟节点是黑色，兄弟节点的红色子孩子在兄弟节点相对双重黑节点的同侧上的情况，我们称之为 LL（兄弟节点在双重黑节点的左侧）或 RR（兄弟节点在双重黑节点的右侧）。像这种情况，我们可以像 AVL 树一样先对 38[2]进行左旋。左旋完之后如下图所示 [3]：

[2]删除调整站在父节点处理

[3]下图根节点为父节点



接下来，请读者思考：哪些节点的颜色是确定的?

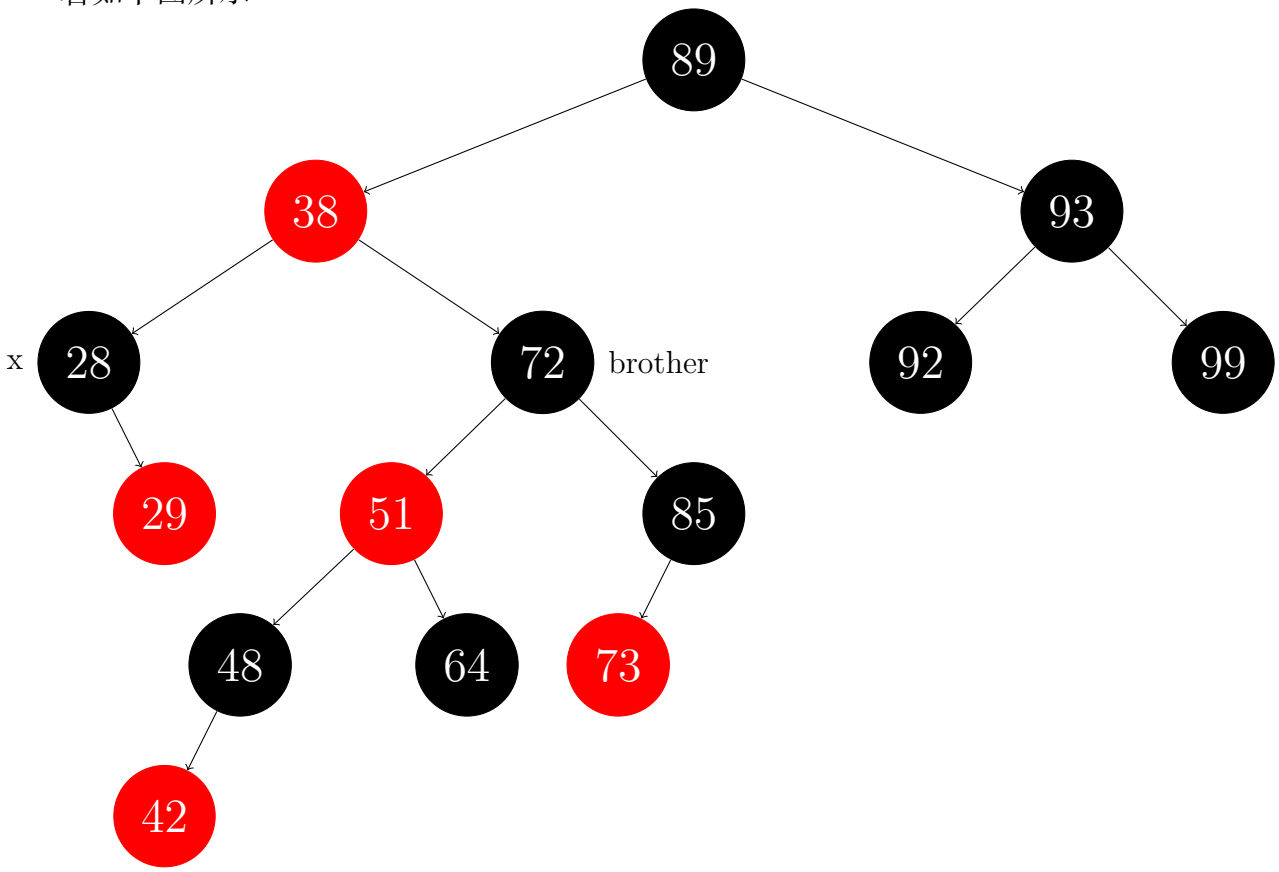
可以得到，51、28、72、64、85 的颜色是确定的。

因为 48 可能为红色，所以 38 应该改为黑色。修改后发现左侧路径上有 3 个黑色节点，所以将 51 变为红色。51 改后又发现，右侧路径上只有 1 个黑色节点，所以将 72 变为黑色。

那么，如果 38 节点原来是黑色呢? 在这种情况下，使用这种调整策略每条路径上的黑色节点会比原来少一个。所以，51 应该改成黑色。

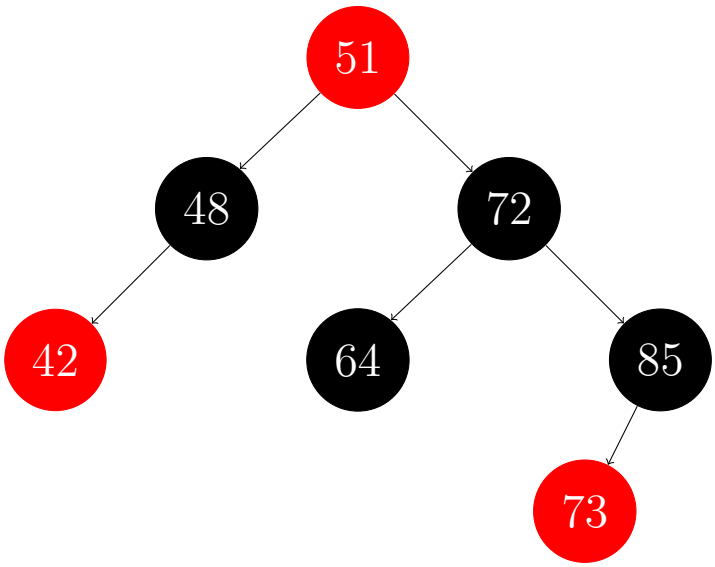
综上所述，对于 LL 和 RR 类型失衡，应当先进行左旋或右旋，再将新根节点的颜色修改成原根节点的颜色，最后将新根节点的左右孩子的颜色修改成黑色。

若如下图所示：



像这种兄弟节点是黑色，兄弟节点的红色子孩子在兄弟节点相对双重黑节点的异侧上的情况，

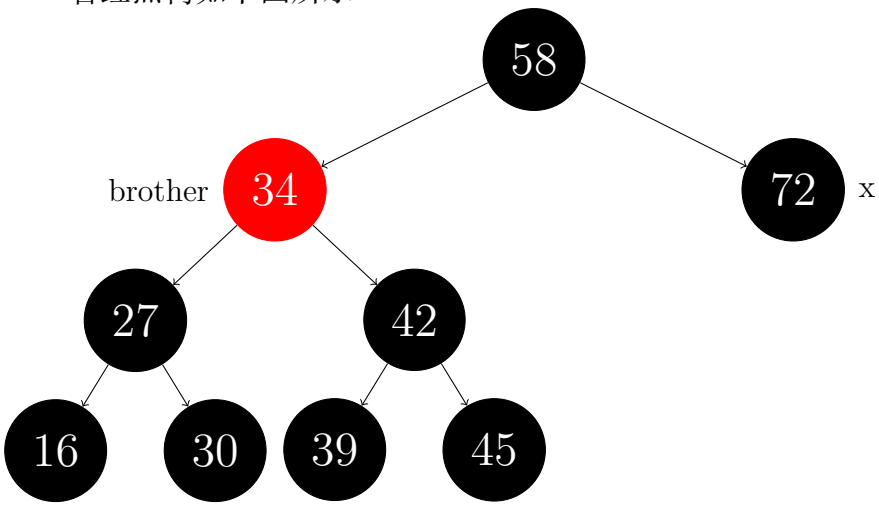
我们称之为 LR（兄弟节点在双重黑节点的左侧）或 RL（兄弟节点在双重黑节点的右侧侧）。对于这种情况，可以先进行小右旋。右旋之后的红黑树片段如下：



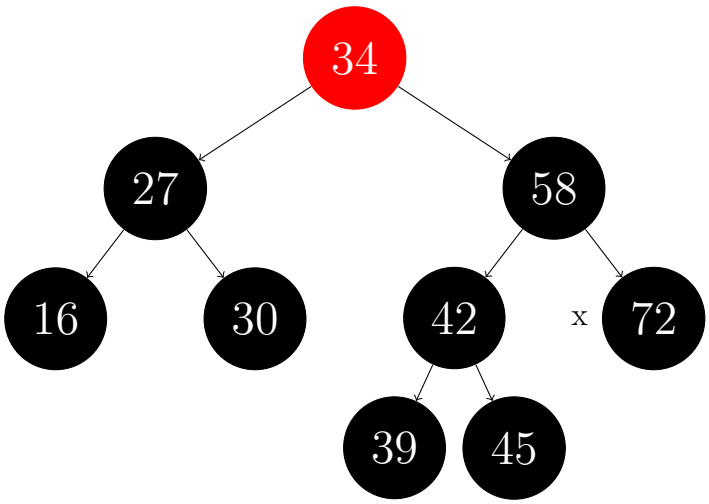
小右旋之后需要调整颜色，但是进行大左旋之后还要调整颜色，所以在代码实现时暂时不调整颜色。

综上所述，先对红黑树进行小右旋或小左旋，转化成 LL 或 RR 之后再进行大左旋或大右旋，按 LL 或 RR 的方法调整颜色。

若红黑树如下图所示：



这种情况可以先进行右旋（红色节点在根节点左边）或左旋（红色节点在根节点右边）。右旋之后的红黑树如下图所示：



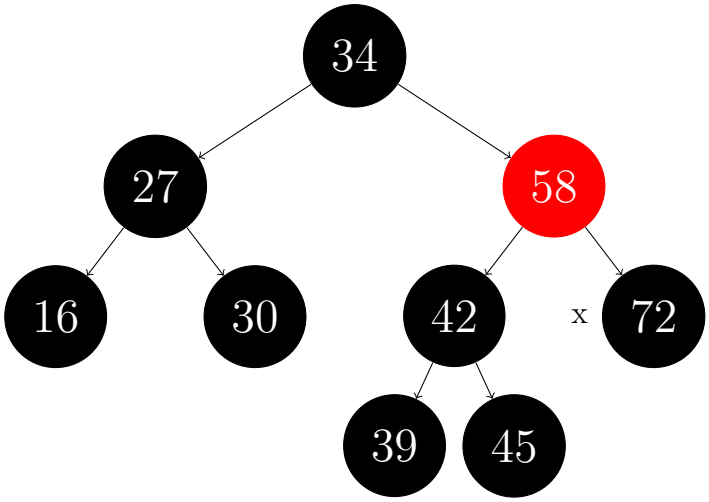
右旋之后，开始调整颜色。

首先，确定哪些节点的颜色是确定的。

可以得出，34、27、58、42、72 的颜色是确定的。

因为红黑树左边的路径黑色节点少了一个，所以将 34 变为黑色。34 改后发现，红黑树右边的路径黑色节点多了一个，所以将 58，也就是原根节点改成红色。

修改后红黑树如下：



此时，双重黑节点旁边的节点是黑色，可以按前三种情况进行调整。

3 代码演示 [3] [1]

Listing 1: RedBlackTree.cpp

```
#include <bits/stdc++.h>
#include <tools.hpp>
#define NIL (&RedBlackTree::mNIL)
#define K(n) (n->mKey)
#define L(n) (n->mLeft)
#define R(n) (n->mRight)
#define C(n) (n->mColor)
using namespace std;
const char* gStr[3] = { "red", "black", "double black" };
class RedBlackTree {
private:
    enum COLOR { RED, BLACK, DBLACK };
    class TreeNode {
    public:
        int mKey;
        COLOR mColor;
        TreeNode *mLeft, *mRight;
        TreeNode ()
            : mKey (0), mColor (RED), mLeft (NIL), mRight (NIL) {}
        TreeNode (int key)
            : mKey (key), mColor (RED), mLeft (NIL), mRight (NIL) {}
        TreeNode (int key, COLOR c)
            : mKey (key), mColor (c), mLeft (NIL), mRight (NIL) {}
        ~TreeNode () {
            if (this == NIL)
                return;
            delete mLeft;
            delete mRight;
        }
        void* operator new (size_t size) {
            return malloc (size);
        }
        void operator delete (void* ptr) {
            if (ptr == NIL)
                return;
            free (ptr);
        }
    };
    static inline bool hasRedNode (TreeNode* root) {
        return C (L (root)) == RED || C (R (root)) == RED;
    }
    static TreeNode* insertMaintain (TreeNode* root) {
        int flag = 0;
        if (C (L (root)) == RED && hasRedNode (L (root)))
            flag = 1;
        if (C (R (root)) == RED && hasRedNode (R (root)))
```

```
47              flag = 2;
48          if (flag == 0)
49              return root;
50          if (C (L (root)) == RED && C (R (root)) == RED)
51              goto upRedMaintain;
52          if (flag == 1) {
53              if (C (R (L (root))) == RED) {
54                  L (root) = leftRotate (L (root));
55              }
56              root = rightRotate (root);
57          } else {
58              if (C (L (R (root))) == RED) {
59                  R (root) = rightRotate (R (root));
60              }
61              root = leftRotate (root);
62          }
63      upRedMaintain:
64          C (root) = RED;
65          C (L (root)) = C (R (root)) = BLACK;
66          return root;
67      }
68      static TreeNode*
69      eraseMainTaim (TreeNode* root) {          // NOLINT
70          if (C (L (root)) != DBLACK && C (R (root)) != DBLACK)
71              return root;
72          // brother : red
73          if (hasRedNode (root)) {
74              root->mColor = RED;
75              if (C (L (root)) == RED) {
76                  root = rightRotate (root);
77                  R (root) = eraseMainTaim (R (root));
78              } else {
79                  root = leftRotate (root);
80                  L (root) = eraseMainTaim (L (root));
81              }
82              root->mColor = BLACK;
83          }
84          if ((C (L (root)) == DBLACK && !hasRedNode (R (root)))
85              || (C (R (root)) == DBLACK && !hasRedNode (L (root)))) {
86              root->mColor += 1;
87              C (L (root)) -= 1;
88              C (R (root)) -= 1;
89              return root;
90          };
91          if (C (R (root)) == DBLACK) {
92              C (R (root)) = BLACK;
93              if (C (L (L (root))) != RED) {
94                  L (root) = leftRotate (L (root));
95              }
96              root->mLeft->mColor = C (root);
```

```
97              root = rightRotate (root);
98          } else {
99              C (L (root)) = BLACK;
100             if (C (R (R (root))) != RED) {
101                 R (root) = rightRotate (R (root));
102             }
103             root->mRight->mColor = C (root);
104             root = leftRotate (root);
105         }
106         C (L (root)) = C (R (root)) = BLACK;
107         return root;
108     }
109     static TreeNode* leftRotate (TreeNode* root) {
110         TreeNode* ptr = root->mRight;
111         root->mRight = ptr->mLeft;
112         ptr->mLeft = root;
113         return ptr;
114     }
115     static TreeNode* rightRotate (TreeNode* root) {
116         TreeNode* ptr = root->mLeft;
117         root->mLeft = ptr->mRight;
118         ptr->mRight = root;
119         return ptr;
120     }
121     static TreeNode* predecessor (TreeNode* root) {
122         TreeNode* tmp = root->mLeft;
123         while (tmp->mRight != NIL)
124             tmp = tmp->mRight;
125         return tmp;
126     }
127     TreeNode* insertT (TreeNode* root,          // NOLINT
128                        int key) {
129         if (root == NIL)
130             return new TreeNode (key);
131         if (K (root) == key)
132             return root;
133         if (key < K (root))
134             L (root) = insertT (L (root), key);
135         else
136             R (root) = insertT (R (root), key);
137         return insertMaintain (root);
138     }
139     void outputT (TreeNode* root) {          // NOLINT
140         if (root == NIL)
141             return;
142         printf ("(%s| %d, %d, %d)\n", gStr[C (root)], K (root),
143                 K (L (root)), K (R (root)));
144         outputT (L (root));
145         outputT (R (root));
146     }
```

```
147     TreeNode* eraseT (TreeNode* root, int key) {          // NOLINT
148         if (root == NIL)
149             return root;
150         if (key < root->mKey) {
151             root->mLeft = eraseT (L (root), key);
152         } else if (key > root->mKey) {
153             root->mRight = eraseT (R (root), key);
154         } else {
155             if (root->mLeft == NIL || root->mRight == NIL) {
156                 TreeNode* tmp =
157                     root->mLeft == NIL ? root->mRight : root->mLeft;
158                 tmp->mColor += root->mColor;
159                 free (root);
160                 return tmp;
161             }
162             TreeNode* tmp = predecessor (root);
163             root->mKey = tmp->mKey;
164             root->mLeft = eraseT (root->mLeft, tmp->mKey);
165         }
166         return eraseMainTaim (root);
167     }
168     friend COLOR& operator+= (COLOR& n1, COLOR& n2) {
169         int i1 = n1, i2 = n2;
170         n1 = ( COLOR )(i1 + i2);
171         return n1;
172     }
173     friend COLOR& operator+= (COLOR& n1, int i2) {
174         int i1 = n1;
175         n1 = ( COLOR )(i1 + i2);
176         return n1;
177     }
178     friend COLOR& operator-= (COLOR& n1, int i2) {
179         int i1 = n1;
180         n1 = ( COLOR )(i1 - i2);
181         return n1;
182     }
183     static TreeNode mNIL;
184     TreeNode* mRoot;
185
186 public:
187     RedBlackTree () : mRoot (NIL) {}
188     ~RedBlackTree () {
189         delete mRoot;
190     }
191     void insert (int key) {
192         mRoot = insertT (mRoot, key);
193         C (mRoot) = BLACK;
194     }
195     void output () {
196         outputT (mRoot);
```

```
197     }
198     void erase (int key) {
199         eraseT (mRoot, key);
200         mRoot->mColor = BLACK;
201     }
202 };
203 RedBlackTree::TreeNode RedBlackTree::mNIL{ -1,          // NOLINT
204                                            BLACK };
205 int main () {
206 #define MAX_OP 10
207     RedBlackTree* t = new RedBlackTree;
208     for (int i = 0; i < MAX_OP; i++) {
209         int x = get_rand<int> (0, 100);
210         printf ("\ninsert %d to RedBlackTree : \n", x);
211         t->insert (x);
212         t->output ();
213     }
214     int x;
215     while (scanf ("%d", &x) != EOF) {
216         printf ("\nerase %d from RedBlackTree\n", x);
217         t->erase (x);
218         t->output ();
219     }
220     return 0;
221 }
```

参考文献

[1] 胡船长. 删除代码演示. https://www.bilibili.com/cheese/play/ep111482. 3

[2] 胡船长. 删除调整. https://www.bilibili.com/cheese/play/ep111481. 2.2

[3] 胡船长. 插入代码演示. https://www.bilibili.com/cheese/play/ep111480. 3

[4] 胡船长. 插入调整. https://www.bilibili.com/cheese/play/ep111479. 2.1