一、红黑树概述

红黑树和我们以前学过的 AVL 树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。不过自从红黑树出来后，AVL 树就被放到了博物馆里，据说是红黑树有更好的效率，更高的统计性能。这一点在我们了解了红黑树的实现原理后，就会有更加深切的体会。

红黑树和 AVL 树的区别在于它使用颜色来标识结点的高度，它所追求的是局部平衡而不是 AVL 树中的非常严格的平衡。学过数据结构的人应该都已经领教过 AVL 树的复杂，但 AVL 树的复杂比起红黑树来说简直是小巫见大巫，红黑树才是真正的变态级数据结构。

由于 STL 中的关联式容器默认的底层实现都是红黑树，因此红黑树对于后续学习 STL 源码还是很重要的，有必要掌握红黑树的实现原理和源码实现。

红黑树是 AVL 树的变种，红黑树通过一些着色法则确保没有一条路径会比其它路径长出两倍，因而达到接近平衡的目的。所谓红黑树，不仅是一个二叉搜索树，而且必须满足一下规则：

1、每个节点不是红色就是黑色。

2、根节点为黑色。

3、如果节点为红色，其子节点必须为黑色。

4、任意一个节点到到 NULL（树尾端）的任何路径，所含之黑色节点数必须相同。

上面的这些约束保证了这个树大致上是平衡的，这也决定了红黑树的插入、删除、查询等操作是比较快速的。 根据规则 4，新增节点必须为红色；根据规则 3，新增节点之父节点必须为黑色。当新增节点根据二叉搜索树的规则到达其插入点时，却未能符合上述条件时，就必须调整颜色并旋转树形，如下图：

假设我们为上图分别插入节点 3、8、35、75，根据二叉搜索树的规则，插入这四个节点后，我们会发现它们都破坏了红黑树的规则，因此我们必须调整树形，也就是旋转树形并改变节点的颜色。

二、红黑树上结点的插入

在讨论红黑树的插入操作之前必须要明白，任何一个即将插入的新结点的初始颜色都为红色。这一点很容易理解，因为插入黑点会增加某条路径上黑结点的数目，从而导致整棵树黑高度的不平衡。但如果新结点的父结点为红色时（如下图所示），将会违反红黑树的性质：一条路径上不能出现相邻的两个红色结点。这时就需要通过一系列操作来使红黑树保持平衡。

为了清楚地表示插入操作以下在结点中使用“新”字表示一个新插入的结点；使用“父”字表示新插入点的父结点；使用“叔”字表示“父”结点的兄弟结点；使用“祖”字表示“父”结点的父结点。插入操作分为以下几种情况：

1、黑父

如下图所示，如果新节点的父结点为黑色结点，那么插入一个红点将不会影

响红黑树的平衡，此时插入操作完成。红黑树比 AVL 树优秀的地方之一在于黑父的情况比较常见，从而使红黑树需要旋转的几率相对 AVL 树来说会少一些。

2、红父

如果新节点的父结点为红色，这时就需要进行一系列操作以保证整棵树红黑性质。如下图所示，由于父结点为红色，此时可以判定，祖父结点必定为黑色。这时需要根据叔父结点的颜色来决定做什么样的操作。青色结点表示颜色未知。由于有可能需要根结点到新点的路径上进行多次旋转操作，而每次进行不平衡判断的起始点（我们可将其视为新点）都不一样。所以我们在此使用一个蓝色箭头指向这个起始点，并称之为判定点。

2.1 红叔

当叔父结点为红色时，如下图所示，无需进行旋转操作，只要将父和叔结点变为黑色，将祖父结点变为红色即可。但由于祖父结点的父结点有可能为红色，从而违反红黑树性质。此时必须将祖父结点作为新的判定点继续向上（迭代）进行平衡操作。

需要注意的是，无论"父节点"在"叔节点"的左边还是右边，无论"新节点"是"父节点"的左孩子还是右孩子，它们的操作都是完全一样的（其实这种情况包括 4 种，只需调整颜色，不需要旋转树形）。

2.2 黑叔

当叔父结点为黑色时，需要进行旋转，以下图示了所有的旋转可能：

Case 1:

Case 2:

Case 3:

Case 4:

可以观察到，当旋转完成后，新的旋转根全部为黑色，此时不需要再向上回溯进行平衡操作，插入操作完成。需要注意，上面四张图的"叔"、"1"、"2"、"3"结点有可能为黑哨兵结点。

其实红黑树的插入操作不是很难，甚至比 AVL 树的插入操作还更简单些。红黑树的插入操作源代码如下：

[cpp] view plain copy

```cpp
// 元素插入操作  insert_unique()
// 插入新值：节点键值不允许重复，若重复则插入无效
// 注意，返回值是个 pair，第一个元素是个红黑树迭代器，指向新增节点
// 第二个元素表示插入成功与否
template<class Key , class Value , class KeyOfValue , class Compare , class Alloc>
pair<typename rb_tree<Key , Value , KeyOfValue , Compare , Alloc>::iterator , bool>
rb_tree<Key , Value , KeyOfValue , Compare , Alloc>::insert_unique(const Value &v)
{
    rb_tree_node* y = header;    // 根节点 root 的父节点
    rb_tree_node* x = root();    // 从根节点开始
    bool comp = true;
    while(x != 0)
    {
        y = x;
        comp = key_compare(KeyOfValue()(v) , key(x));    // v 键值小于目前节点之键值？
        x = comp ? left(x) : right(x);   // 遇“大”则往左，遇“小于或等于”则往右
    }
    // 离开 while 循环之后，y 所指即插入点之父节点（此时的它必为叶节点）
    iterator j = iterator(y);     // 令迭代器 j 指向插入点之父节点 y
    if(comp)     // 如果离开 while 循环时 comp 为真（表示遇“大”，将插入于左侧）
    {
        if(j == begin())    // 如果插入点之父节点为最左节点
            return pair<iterator , bool>(_insert(x , y , z) , true);
        else     // 否则（插入点之父节点不为最左节点）
            --j;   // 调整 j，回头准备测试
    }
    if(key_compare(key(j.node) , KeyOfValue()(v) ))
        // 新键值不与既有节点之键值重复，于是以下执行安插操作
        return pair<iterator , bool>(_insert(x , y , z) , true);
    // 以上，x 为新值插入点，y 为插入点之父节点，v 为新值

    // 进行至此，表示新值一定与树中键值重复，那么就不应该插入新值
    return pair<iterator , bool>(j , false);
}

// 真正地插入执行程序 _insert()
template<class Key , class Value , class KeyOfValue , class Compare , class Alloc>
```

```
typename<Key , Value , KeyOfValue , Compare , Alloc>::_insert(base_ptr x_ , base_ptr y_ , const Value &v)
{
    // 参数 x_ 为新值插入点，参数 y_为插入点之父节点，参数 v 为新值
    link_type x = (link_type) x_;
    link_type y = (link_type) y_;
    link_type z;

    // key_compare 是键值大小比较准则。应该会是个 function object
    if(y == header || x != 0 || key_compare(KeyOfValue()(v) , key(y) ))
    {
        z = create_node(v);     // 产生一个新节点
        left(y) = z;            // 这使得当 y 即为 header 时，leftmost() = z
        if(y == header)
        {
            root() = z;
            rightmost() = z;
        }
        else if(y == leftmost())      // 如果 y 为最左节点
            leftmost() = z;           // 维护 leftmost()，使它永远指向最左节点
    }
    else
    {
        z = create_node(v);         // 产生一个新节点
        right(y) = z;               // 令新节点成为插入点之父节点 y 的右子节点
        if(y == rightmost())
            rightmost() = z;        // 维护 rightmost()，使它永远指向最右节点
    }
    parent(z) = y;       // 设定新节点的父节点
    left(z) = 0;         // 设定新节点的左子节点
    right(z) = 0;        // 设定新节点的右子节点
    // 新节点的颜色将在_rb_tree_rebalance()设定（并调整）
    _rb_tree_rebalance(z , header->parent);       // 参数一为新增节点，参数二为根节点 root
    ++node_count;        // 节点数累加
    return iterator(z);  // 返回一个迭代器，指向新增节点
}


// 全局函数
// 重新令树形平衡（改变颜色及旋转树形）
// 参数一为新增节点，参数二为根节点 root
inline void _rb_tree_rebalance(_rb_tree_node_base* x , _rb_tree_node_base*& root)
```

```
{
    x->color = _rb_tree_red;    //新节点必为红
    while(x != root && x->parent->color == _rb_tree_red)    // 父节点为红
    {
        if(x->parent == x->parent->parent->left)      // 父节点为祖父节点之左子节点
        {
            _rb_tree_node_base* y = x->parent->parent->right;    // 令 y 为伯父节点
            if(y && y->color == _rb_tree_red)    // 伯父节点存在，且为红
            {
                x->parent->color = _rb_tree_black;           // 更改父节点为黑色
                y->color = _rb_tree_black;                   // 更改伯父节点为黑色
                x->parent->parent->color = _rb_tree_red;     // 更改祖父节点为红色
                x = x->parent->parent;
            }
            else    // 无伯父节点，或伯父节点为黑色
            {
                if(x == x->parent->right)   // 如果新节点为父节点之右子节点
                {
                    x = x->parent;
                    _rb_tree_rotate_left(x , root);    // 第一个参数为左旋点
                }
                 x->parent->color = _rb_tree_black;     // 改变颜色
                 x->parent->parent->color = _rb_tree_red;
                 _rb_tree_rotate_right(x->parent->parent , root);    // 第一个参数为右旋点
             }
        }
        else          // 父节点为祖父节点之右子节点
        {
            _rb_tree_node_base* y = x->parent->parent->left;    // 令 y 为伯父节点
            if(y && y->color == _rb_tree_red)    // 有伯父节点，且为红
            {
                x->parent->color = _rb_tree_black;           // 更改父节点为黑色
                y->color = _rb_tree_black;                   // 更改伯父节点为黑色
                x->parent->parent->color = _rb_tree_red;     // 更改祖父节点为红色
```

```
113.                 x = x->parent->parent;          // 准备继续往上层检查
114.             }
115.             else    // 无伯父节点，或伯父节点为黑色
116.             {
117.                 if(x == x->parent->left)        // 如果新节点为父节点之左子节点
118.                 {
119.                     x = x->parent;
120.                     _rb_tree_rotate_right(x , root);    // 第一个参数为右旋点
121.                 }
122.                 x->parent->color = _rb_tree_black;     // 改变颜色
123.                 x->parent->parent->color = _rb_tree_red;
124.                 _rb_tree_rotate_left(x->parent->parent , root);    // 第一个参数为左旋点
125.             }
126.         }
127.     }//while
128.     root->color = _rb_tree_black;    // 根节点永远为黑色
129. }
130.
131.
132. // 左旋函数
133. inline void _rb_tree_rotate_left(_rb_tree_node_base* x , _rb_tree_node_base *& root)
134. {
135.     // x 为旋转点
136.     _rb_tree_node_base* y = x->right;          // 令 y 为旋转点的右子节点
137.     x->right = y->left;
138.     if(y->left != 0)
139.         y->left->parent = x;           // 别忘了回马枪设定父节点
140.     y->parent = x->parent;
141.
142.     // 令 y 完全顶替 x 的地位（必须将 x 对其父节点的关系完全接收过来）
143.     if(x == root)    // x 为根节点
144.         root = y;
145.     else if(x == x->parent->left)         // x 为其父节点的左子节点
146.         x->parent->left = y;
147.     else                                  // x 为其父节点的右子节点
148.         x->parent->right = y;
149.     y->left = x;
150.     x->parent = y;
151. }
152.
```

```
153.
154. // 右旋函数
155. inline void _rb_tree_rotate_right(_rb_tree_node_base* x , _rb_tree_node_bas
     e*& root)
156. {
157.     // x 为旋转点
158.     _rb_tree_node_base* y = x->left;          // 令 y 为旋转点的左子节点
159.     x->left = y->right;
160.     if(y->right != 0)
161.         y->right->parent = x;            // 别忘了回马枪设定父节点
162.     y->parent = x->parent;
163.
164.     // 令 y 完全顶替 x 的地位（必须将 x 对其父节点的关系完全接收过来）
165.     if(x == root)
166.         root = y;
167.     else if(x == x->parent->right)          // x 为其父节点的右子节点
168.         x->parent->right = y;
169.     else                                    // x 为其父节点的左子节点
170.         x->parent->left = y;
171.     y->right = x;
172.     x->parent = y;
173. }
```


算法导论书上给出的红黑树的性质如下，跟 STL 源码剖析书上面的 4 条性质大同小异。

1、每个结点或是红色的，或是黑色的
2、根节点是黑色的
3、每个叶结点（NIL）是黑色的
4、如果一个节点是红色的，则它的两个儿子都是黑色的。
5、对于每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑色结点。

从红黑树上删除一个节点，可以先用普通二叉搜索树的方法，将节点从红黑树上删除掉，然后再将被破坏的红黑性质进行恢复。

我们回忆一下普通二叉树的节点删除方法：Z 指向需要删除的节点，Y 指向实质结构上被删除的结点，如果 Z 节点只有一个子节点或没有子节点，那么 Y 就是指向 Z 指向的节点。如果 Z 节点有两个子节点，那么 Y 指向 Z 节点的后继节点（其实前趋也是一样的），而 Z 的后继节点绝对不可能有左子树。因此，仅从结构来看，二叉树上实质被删除的节点最多只可能有一个子树。

现在我们来看红黑性质的恢复过程：

如果 Y 指向的节点是个红色节点，那么直接删除掉 Y 以后，红黑性质不会

被破坏。操作结束。

如果 Y 指向的节点是个黑色节点，那么就有几条红黑性质可能受到破坏了。首先是包含 Y 节点的所有路径，黑高度都减少了一（第 5 条被破坏）。其次，如果 Y 的有红色子节点，Y 又有红色的父节点，那么 Y 被删除后，就出现了两个相邻的红色节点（第 4 条被破坏）。最后，如果 Y 指向的是根节点，而 Y 的子节点又是红色的，那么 Y 被删除后，根节点就变成红色的了（第 2 条被破坏）。

其中，第 5 条被破坏是让我们比较难受的。因为这影响到了全局。这样动作就太大太复杂了。而且在这个条件下，进行其它红黑性质的恢复也很困难。所以我们首先解决这个问题：如果不改变含 Y 路径的黑高度，那么树的其它部分的黑高度就必须做出相应的变化来适应它。所以，我们想办法恢复原来含 Y 节点的路径的黑高度。做法就是：**无条件的把 Y 节点的黑色，推到它的子节点 X 上去**。（X 可能是 NIL 节点）。这样，**X 就可能具有双重黑色，或同时具有红黑两色**，也就是第 1 条性质被破坏了。

但第 1 条性质是比较容易恢复的：一、如果 X 是同时具有红黑两色，那么好办，直接把 X 涂成黑色，就行了。而且这样把所有问题都解决了。因为将 X 变为黑色，2、4 两条如果有问题的话也会得到恢复，算法结束。二、如果 X 是双黑色，那么我们希望把这种情况向上推一直推到根节点（调整树结构和颜色，X 的指向新的双黑色节点，X 不断向上移动），让根节点具双黑色，这时，直接把 X 的一层黑色去掉就行了（因为根节点被包含在所有的路径上，所以这样做所有路径同时黑高减少一，不会破坏红黑特征）。

下面就具体地分析如何恢复 1、2、4 三个可能被破坏的红黑特性：我们知道，如果 X 指向的节点是有红黑两色，或是 X 是根节点时，只需要简单的对 X 进行一些改变就行了。要对除 X 节点外的其它节点进行操作时，必定是这样的情况：X 节点是双层黑色，且 X 有父节点 P。由知可知，X 必然有兄弟节点 W，而且这个 W 节点必定有两个子节点。（因为这是原树满足红黑条件要求而自然具备的。X 为双黑色，那么 P 的另一个子节点以下一定要有至少两层的节点，否则黑色高度不可能和 X 路径一致）。所以我们就分析这些节点之间如何变形，把问题限制在比较小的范围内解决。另一个前提是：X 在一开始，肯定是树底的叶节点或是 NIL 节点，所以在递归向上的过程中，每一步都保证下一步进行时，至少 X 的子树是满足红黑特性的。因此子树的情况就可以认为是已经正确的了，这样，分析就只限制在 X 节点，X 的父节点 P 和 X 的兄弟节点 W，以及 W 的两个子节点中。

下面仅仅考虑 X 原本是黑色的情况即可。

在这种情况下，X 此时应该具有双重黑色，算法的过程就是将这多出的一重黑色向上移动，直到遇到红节点或者根节点。

接着往下分析， 会遇到 4 种情况，实际上是 8 种， 因为其中 4 种是相互对称的，这可以通过判断 X 是其父节点的右孩子还是左孩子来区分。下面我们以 X 是其父节点的左孩子的情况来分析这 4 种情况，实际上接下来的调整过程，就是要想方设法将经过 X 的所有路径上的黑色节点个数增加 1。

具体分为以下四种情况：（下面针对 x 是左儿子的情况讨论，右儿子对称）

Case1：X 的兄弟 W 是红色（想办法将其变为黑色）

由于 W 是红色的，因此其儿子节点和父节点必为黑色，只要将 W 和其父

节点的颜色对换，在对父节点进行一次左旋转，便将 W 的左子节点放到了 X 的兄弟节点上，X 的兄弟节点变成了黑色，且红黑性质不变。但还不算完，只是暂时将情况 1 转变成了下面的情况 2 或 3 或 4。

Case2：X 的兄弟节点 W 是黑色的，而且 W 的两个子节点都是黑色的。此时可以将 X 的一重黑色和 W 的黑色同时去掉，而转加给他们的父节点上，这是 X 就指向它的父节点了，因此此时父节点具有双重颜色了。这一重黑色节点上移。

如果父节点原来是红色的，现在又加一层黑色，那么 X 现在指向的这个节点就是红黑两色的，直接把 X（也就是父节点）着为黑色。问题就已经完整解决了。

如果父节点现在是双层黑色，那就以父节点为新的 X 进行向上的下一次的递归。

Case3：X 的兄弟节点 W 是黑色的，而且 W 的左子节点是红色的，右子节点是黑色的。此时通过交换 W 和其左子节点的颜色并进行一次向右旋转就可转换成下面的第四种情况。注意，原来 L 是红色的，所以 L 的子节点一定是黑色的，所以旋转中 L 节点的一个子树挂到之后着为红色的 W 节点上不会破坏红黑性质。变形后黑色高度不变。

Case4：X 的兄弟节点 W 是黑色的，而且 W 的右子节点是红色的。这种情况下，做一次左旋，W 就处于根的位置，将 W 保持为原来的根的位置的颜色，同时将 W 的两个新的儿子节点的颜色变为黑色，去掉 X 的一重黑色。这样整个问题也就得到了解决。递归结束。（在代码上，为了标识递归结束，我们把 X 指向根节点）

因此，只要按上面四种情况一直递归处理下去，X 最终总会指向根结点或一个红色结点，这时我们就可以结束递归并把问题解决了。

以上就是红黑树的节点删除全过程。

总结：

如果我们通过上面的情况画出所有的分支图，我们可以得出如下结论

插入操作：解决的是 红-红 问题

删除操作：解决的是 黑-黑 问题

即你可以从分支图中看出，需要往上遍历的情况为红红(插入)，或者为黑黑黑（删除）的情况，如果你认真分析并总结所有的情况后，并坚持下来，红黑树也就没有想象中的那么恐怖了，并且很美妙；

详细的红黑树删除节点的代码如下：

[cpp] view plain copy

```cpp
#include<iostream>
using namespace std;

// 定义节点颜色
enum COLOR
{
    BLACK = 0,
    RED
};

// 红黑树节点
typedef struct RB_Tree_Node
{
    int key;
    struct RB_Tree_Node *left;
    struct RB_Tree_Node *right;
    struct RB_Tree_Node *parent;
    unsigned char RB_COLOR;
}RB_Node;

// 红黑树，包含一个指向根节点的指针
typedef struct RBTree
{
    RB_Node* root;
}*RB_Tree;

// 红黑树的 NIL 节点
static RB_Tree_Node NIL = {0, 0, 0, 0, BLACK};

#define PNIL (&NIL)   // NIL 节点地址

```

```
void Init_RBTree(RB_Tree pTree) // 初始化一棵红黑树
{
    pTree->root = PNIL;
}

// 查找最小键值节点
RB_Node* RBTREE_MIN(RB_Node* pRoot)
{
    while (PNIL != pRoot->left)
    {
        pRoot = pRoot->left;
    }
    return pRoot;
}


/*
                          15
                        /    \
                       /      \
                      /        \
                     6          18
                    /  \       /  \
                   /    \     /    \
                  3      7   17    20
                 /  \     \
                /    \     \
               2      4     13
                           /
                          /
                         9
*/
// 查找指定节点的后继节点
RB_Node* RBTREE_SUCCESSOR(RB_Node*  pRoot)
{
    if (PNIL != pRoot->right)    // 查找图中 6 的后继节点时就调用 RBTREE_MIN 函数
    {
        return RBTREE_MIN(pRoot->right);
    }
    // 节点没有右子树的时候，进入下面的 while 循环（如查找图中 13 的后继节点时，它的后继节点是 15）
    RB_Node* pParent = pRoot->parent;
    while((PNIL != pParent) && (pRoot == pParent->right))
```

```
74.     {
75.         pRoot = pParent;
76.         pParent = pRoot->parent;
77.     }
78.     return pParent;
79. }
80.
81. // 红黑树的节点删除
82. RB_Node* Delete(RB_Tree pTree , RB_Node* pDel)
83. {
84.     RB_Node* rel_delete_point;
85.     if(pDel->left == PNIL || pDel->right == PNIL)
86.         rel_delete_point = pDel;
87.     else
88.         rel_delete_point = RBTREE_SUCCESSOR(pDel);     // 查找后继节点
89.
90.     RB_Node* delete_point_child;
91.     if(rel_delete_point->right != PNIL)
92.     {
93.         delete_point_child = rel_delete_point->right;
94.     }
95.     else if(rel_delete_point->left != PNIL)
96.     {
97.         delete_point_child = rel_delete_point->left;
98.     }
99.     else
100.     {
101.         delete_point_child = PNIL;
102.     }
103.     delete_point_child->parent = rel_delete_point->parent;
104.     if(rel_delete_point->parent == PNIL)    // 删除的节点是根节点
105.     {
106.         pTree->root = delete_point_child;
107.     }
108.     else if(rel_delete_point == rel_delete_point->parent->right)
109.     {
110.         rel_delete_point->parent->right = delete_point_child;
111.     }
112.     else
113.     {
114.         rel_delete_point->parent->left = delete_point_child;
115.     }
116.     if(pDel != rel_delete_point)
117.     {
```

```
        pDel->key = rel_delete_point->key;
    }
    if(rel_delete_point->RB_COLOR == BLACK)
    {
        DeleteFixUp(pTree , delete_point_child);
    }
    return rel_delete_point;
}


/*
算法导论上的描述如下：
RB-DELETE-FIXUP(T, x)
1 while x ≠ root[T] and color[x] = BLACK
2     do if x = left[p[x]]
3           then w ← right[p[x]]
4                if color[w] = RED
5                   then color[w] ← BLACK                          Case 1
6                        color[p[x]] ← RED                         Case 1
7                        LEFT-ROTATE(T, p[x])                      Case 1
8                        w ← right[p[x]]                           Case 1
9                if color[left[w]] = BLACK and color[right[w]] = BLACK
10                  then color[w] ← RED                            Case 2
11                       x p[x]                                    Case 2
12                  else if color[right[w]] = BLACK
13                          then color[left[w]] ← BLACK            Case 3
14                               color[w] ← RED                    Case 3
15                               RIGHT-ROTATE(T, w)                Case 3
16                               w ← right[p[x]]                   Case 3
17                       color[w] ← color[p[x]]                    Case 4
18                       color[p[x]] ← BLACK                       Case 4
```

```
19                          color[right[w]] ← BLACK                 Case 4

20                          LEFT-ROTATE(T, p[x])                    Case 4

21                          x ← root[T]                             Case 4

22        else (same as then clause with "right" and "left" exchanged)
23 color[x] ← BLACK
*/
//接下来的工作，很简单，即把上述伪代码改写成 c++代码即可
void DeleteFixUp(RB_Tree pTree , RB_Node* node)
{
    while(node != pTree->root && node->RB_COLOR == BLACK)
    {
        if(node == node->parent->left)
        {
            RB_Node* brother = node->parent->right;
            if(brother->RB_COLOR==RED)   //情况 1：x 的兄弟 w 是红色的。
            {
                brother->RB_COLOR = BLACK;
                node->parent->RB_COLOR = RED;
                RotateLeft(node->parent);
            }
            else     //情况 2：x 的兄弟 w 是黑色的，
            {
                if(brother->left->RB_COLOR == BLACK && brother->right->RB_C
OLOR == BLACK)  //w 的两个孩子都是黑色的
                {
                    brother->RB_COLOR = RED;
                    node = node->parent;
                }
                else
                {
                    if(brother->right->RB_COLOR == BLACK)   //情况 3：x 的兄弟
w 是黑色的，w 的右孩子是黑色（w 的左孩子是红色）。
                    {
                        brother->RB_COLOR = RED;
                        brother->left->RB_COLOR = BLACK;
                        RotateRight(brother);
                        brother = node->parent->right;      //情况 3 转换为情
况 4
                    }
                    //情况 4：x 的兄弟 w 是黑色的，且 w 的右孩子时红色的
                    brother->RB_COLOR = node->parent->RB_COLOR;
```

```
                    node->parent->RB_COLOR = BLACK;
                    brother->right->RB_COLOR = BLACK;
                    RotateLeft(node->parent);
                    node = pTree->root;
                }//else
            }//else
        }
        else   //同上，原理一致，只是遇到左旋改为右旋，遇到右旋改为左旋即可。其它代码不变。
        {
            RB_Node* brother = node->parent->left;
            if(brother->RB_COLOR == RED)
            {
                brother->RB_COLOR = BLACK;
                node->parent->RB_COLOR = RED;
                RotateRight(node->parent);
            }
            else
            {
                if(brother->left->RB_COLOR==BLACK && brother->right->RB_COLOR == BLACK)
                {
                    brother->RB_COLOR = RED;
                    node = node->parent;
                }
                else
                {
                    if(brother->left->RB_COLOR==BLACK)
                    {
                        brother->RB_COLOR = RED;
                        brother->right->RB_COLOR = BLACK;
                        RotateLeft(brother);
                        brother = node->parent->left;      //情况3转换为情况4
                    }
                    brother->RB_COLOR = node->parent->RB_COLOR;
                    node->parent->RB_COLOR = BLACK;
                    brother->left->RB_COLOR = BLACK;
                    RotateRight(node->parent);
                    node = pTree->root;
                }
            }
        }
    }//while
```

```
228.     node->RB_COLOR = BLACK;     //如果 X 节点原来为红色，那么直接改为黑色
229. }
```