# [彻底搞懂红黑树（一）](http://blog.csdn.net/haidao2009/article/details/8076970)

分类： [数据结构](http://blog.csdn.net/haidao2009/article/category/1258364)2012-10-16 15:24 3536人阅读 [评论](http://blog.csdn.net/haidao2009/article/details/8076970#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/haidao2009/article/details/8076970#report)

目录[(?)[+]](http://blog.csdn.net/haidao2009/article/details/8076970)

 红黑树和c++ 虚拟继承内存分布 几乎成了我的死敌，因为完全没用过，所以导致每次看懂了之后都忘了（也许不是真的看懂了，有些关键性的东西没理解透），这次准备把这两个难题（其实也不难）仔细看懂，然后再写一份比较详细的文档作为备忘。

首先是红黑树

## 零  八卦起源

    1972年，鲁道夫贝尔最先发明，但是他称之为“对称二叉B树”，真正的称之为“红黑树”是在1978年Leo J. Guibas 和 Robert Sedgewick的一篇论文开始的。

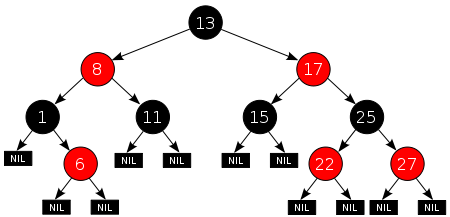
红黑树的命名和Xerox PARC也有关系，当时Robert Sedgewick在施乐做访问学者的时候，施乐正好发明出激光彩色打印机，然后觉得这个打印机打出的红色很好看，就用了红色来区分结点的不同，于是就叫红黑树了。 （注：这个数据结构是1972 由 Rudolf Bayer 发明的，叫 "symmetric binary B-tree"）。摘自[左耳朵耗子](http://weibo.com/haoel)的 微博。

## 一  红黑树的定义

算法导论里是这样定义一棵红黑树的：

      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。

    话说这些性质里面，前四个都很容易理解，但惟独第五个，我是怎么读怎么觉得拗口，原因就是对这个“子孙节点”感到疑惑，这个子孙节点是啥子玩意儿呢?所有的子、孙节点？？？ 如果当前节点是根节点，那么根据定义：子孙节点有相同的黑色节点数，那不是搞笑吗？怎么可能得到呢。后来我看了一下英文版的，发现是这个：descendant leaves，貌似理解为子孙节点也不错哈，但我查了《Introduction to Algorithms》全书，也就这么一处用了descendant leaves，其他地方都是leaves，很显然这里的descendant leaves应该是特指，不应该就泛泛的翻译为子孙节点，而应该是子孙叶节点才对，也就是树最边上的节点啦。这样理解起来也自然，也不会有歧义。然后对照红黑树的图（下图一）一比较，果然是这样。也不晓得是不是自己的语文差了，反正这里顺便鄙视下《算法导论》的翻译吧。也就是说从根节点（如下图 13根节点）到其所有的nil黑节点路径里的黑节点数就是一样的。



图一

    好吧，红黑树的性质就理解到这里。也许大家看书的话就会继续往下看了，当然我也是这样的。但是当我读第二遍的时候，我就想为什么这简简单单的5条性质就能让红黑树有如此好的性质。AVL树我还可以理解，毕竟有看得见摸得着的定义和balance代码，但红黑树我就不理解了，因为他的定义是如此的简单，也不可能看到定义就能想象它是如何调整树结构达到动态balance的，搞得我很恼火。我曾经试图搜索下 红黑树本质，但发现没有这方面的信息。o(╯□╰)o，我也只好自己思考下了。

## 二  引理

    估计有很多人对书上13.1 引理不是很在意，但是我想说，这正是理解红黑树精髓的地方之一。 这条引理也是红黑树为什么效率这么高的原因。不晓得是语文差还是啥的，我看书上的介绍也没看懂，写得太简洁了，菜鸟看不懂表示很蛋疼。后来在网上看了别人的资料，终于弄懂了。(⊙o⊙)…

下面我就仔仔细细的介绍下这条引理到底是怎么得到的。

引理：一棵有n个内结点的红黑树的高度至多为2lg(n+1)。

    这个引理怎么证明呢，这里需要一个工具  对以x为根的子树，它所包含的内部结点数至少为2^[bh(x)]-1。这里bh(x)（bh嘛，black height）被定义为结点x的黑高度，就是说，从结点x（不包括它本身）到它任一个叶结点的路径上所有黑色结点的个数。

下面用归纳法证明：

    1)若x高度为0，那么它就是一叶子结点，它确实至少包含2^0-1=0个内部结点

    2)假设x为红黑树的某一内部结点，且它高度h>0，那么它的黑高度就是bh(x)，但是它的两个孩子结点呢？这个就根据它们的颜色来判断了：

       如果x有一个红色的孩子y，那么y的黑高度bh(y)=bh(x)，看看上面对黑高度的定义你就明白了——既然它是红色的，那么它的黑高度就应该和它父亲的黑高度是一样的；

       如果x有一个黑色的孩子z，那么z的黑高度bh(z)=bh(x)-1，这个怎么解释呢，因为它自己就是个黑结点，那么在计算它的黑高度时，必须把它自己排除在外（还是根据定义），所以它是bh(x)-1。

    3)x的孩子结点所构成的子树的高度肯定小于x这颗子树，那么对于这两个孩子，不管它们颜色如何，一定满足归纳假设的是至少hb 高度为bh(x)-1。所以，对x来说，它所包含的内部结点个数“至少”为两个孩子结点所包含的内部结点数，再加上它自己，于是就为2^[bh(x)-1]-1+2^[bh(x)-1]-1+1=2^[bh(x)]-1，归纳证明完毕。

也就是说n>=2^[bh(x)]-1---------①

     把一 中红黑树性质中 4）、5）两个特性结合起来，其实我们可以得到黑节点至少是红节点的2倍。用一句话来说就是“有红必有黑，但有黑未必一定有红”。为什么这么说呢，因为从特性4）我们知道，如果有一个红结点存在，那么它的儿子结点一定是黑的，最极端的情况下，该路径上所有的结点就被红、黑两种结点给平分了那就是黑节点至少是红节点的2倍。不知这个问题我解释清楚没有，因为这是往下理解的关键。

    如果一棵红黑树的高为h，那么在这个高度上（不包括根结点本身）至少有1/2h的黑结点，再结合上面对“黑高度”的定义，我们说，红黑树根结点的黑高度至少是1/2h，好了，我们拿出公式①，设n为该红黑树所包含的内部结点数，我们得出如下结论： n>=2^(1/2h)-1。 我们把它整理整理，就得到了h<=2lg(n+1)，就是我们要证明的结论：红黑树的高度最多也就是2lg(n+1)。(⊙o⊙)

# [彻底搞懂红黑树（二）](http://blog.csdn.net/haidao2009/article/details/8077514)

分类： [数据结构](http://blog.csdn.net/haidao2009/article/category/1258364)2012-10-16 16:31 2316人阅读 [评论](http://blog.csdn.net/haidao2009/article/details/8077514#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/haidao2009/article/details/8077514#report)

[tree](http://www.csdn.net/tag/tree)[iterator](http://www.csdn.net/tag/iterator)[insert](http://www.csdn.net/tag/insert)[class](http://www.csdn.net/tag/class)[header](http://www.csdn.net/tag/header)

目录[(?)[+]](http://blog.csdn.net/haidao2009/article/details/8077514)

其实关于红黑树，[STL源码剖析---红黑树原理详解](http://blog.csdn.net/hackbuteer1/article/details/7740956) 已经写得非常好了。但套用新警察故事里的谢霆锋说的一句话：自己查，印象深一点。这里也是一样，在自己写，印象深一点。如果你要看正宗的[STL源码剖析---红黑树原理详解](http://blog.csdn.net/hackbuteer1/article/details/7740956)，那请你点击这个。这里的是D版的o(╯□╰)o  当然，我也会加一些我自己的理解，因为大神写文章都比较精简，而我这是写给我自己看的，有一点口水话加深点印象。

## 三 红黑树的插入

红黑树的节点插入默认是节点为红色的。我自己理解是，其实插入红还是黑都可以，但就要看后面的调整是否麻烦。

插入黑点，会增加路径上黑点的数目，一定会破坏性质5  
插入红点：

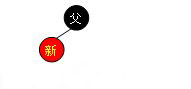
当其父节点为黑色时，不影响平衡，继续保持红黑性质  
当其父节点为红色时，可能破坏性质2（根节点是黑色的）、性质4（红色节点的子节点一定是黑色节点），需要进行修正。

因为第一篇文章已经说了，黑节点至少是红节点的两倍，这说明插入红节点OK的概率高很多（因为父节点为黑，插入子节点为红色就不会和性质冲突），这样插入就省事多了，嘿嘿，这也是红黑树为什么战胜AVL树的原因之一，插入效率高啊，节点贴上去就ok了，都不用什么左转右转调整了，多省事啊；再说，如果插入子节点为黑色，o(╯□╰)o了，黑高度变化了，得调整，如果每次都插入黑节点，都得调整，没事闲的蛋疼啊。。。

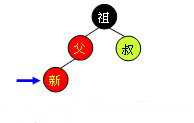
红黑树插入分一下几种情况：

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。（大神和我的理解差不多，不过别人的精简很多，我的就是口水话。）

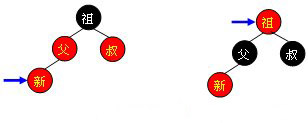


2、红父  
     如果新节点的父结点为红色，这时就需要进行一系列操作以保证整棵树红黑性质。如下图所示，由于父结点为红色，此时可以判定，祖父结点必定为黑色。这时需要根据叔父结点的颜色来决定做什么样的操作。青色结点表示颜色未知。由于有可能需要根结点到新点的路径上进行多次旋转操作，而每次进行不平衡判断的起始点（我们可将其视为新点）都不一样。所以我们在此使用一个蓝色箭头指向这个起始点，并称之为判定点。



图一

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。（注意这里是需要迭代的，有可能会调整到根节点）

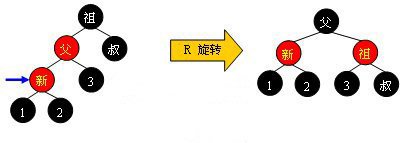


图二

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）。

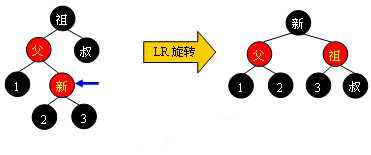
2.2 黑叔  
当叔父结点为黑色时，需要进行旋转，以下图示了所有的旋转可能：（case1 和caes 2 都是把左边较大的节点调整到上面去）

Case 1:



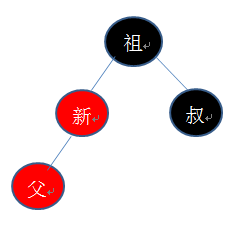
图三

Case 2:



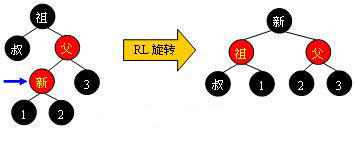
图四

其实这里case 2的图示进行了简化，就是case 2 需要L变换成case 1 的情形如图五，然后再进行R旋转，得到最后的结果。当然，下面的也一样

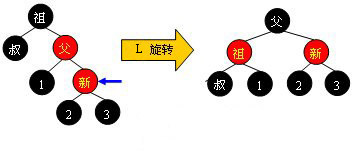


图五（图中省略了哨兵结点）

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。

      有了上面的分析，红黑树的插入就好理解多了，代码也容易读懂。不过不是很喜欢c++的代码，有时间了把linux内核的红黑树源码贴出来。这里还是贴的大神注视的红黑树的插入操作源代码：

## stl里的红黑树插入操作源码

**[cpp]** [view plaincopy](http://blog.csdn.net/haidao2009/article/details/8077514)

1. // 元素插入操作  insert\_unique()
2. // 插入新值：节点键值不允许重复，若重复则插入无效
3. // 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
4. // 第二个元素表示插入成功与否
5. **template**<**class** Key , **class** Value , **class** KeyOfValue , **class** Compare , **class** Alloc>
6. pair<**typename** rb\_tree<Key , Value , KeyOfValue , Compare , Alloc>::iterator , **bool**>
7. rb\_tree<Key , Value , KeyOfValue , Compare , Alloc>::insert\_unique(**const** Value &v)
8. {
9. rb\_tree\_node\* y = header;    // 根节点root的父节点
10. rb\_tree\_node\* x = root();    // 从根节点开始
11. **bool** comp = **true**;
12. **while**(x != 0)
13. {
14. y = x;
15. comp = key\_compare(KeyOfValue()(v) , key(x));    // v键值小于目前节点之键值？
16. x = comp ? left(x) : right(x);   // 遇“大”则往左，遇“小于或等于”则往右
17. }
18. // 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）
19. iterator j = iterator(y);     // 令迭代器j指向插入点之父节点y
20. **if**(comp)     // 如果离开while循环时comp为真（表示遇“大”，将插入于左侧）
21. {
22. **if**(j == begin())    // 如果插入点之父节点为最左节点
23. **return** pair<iterator , **bool**>(\_insert(x , y , z) , **true**);
24. **else**     // 否则（插入点之父节点不为最左节点）
25. --j;   // 调整j，回头准备测试
26. }
27. **if**(key\_compare(key(j.node) , KeyOfValue()(v) ))
28. // 新键值不与既有节点之键值重复，于是以下执行安插操作
29. **return** pair<iterator , **bool**>(\_insert(x , y , z) , **true**);
30. // 以上，x为新值插入点，y为插入点之父节点，v为新值

33. // 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
34. **return** pair<iterator , **bool**>(j , **false**);
35. }

38. // 真正地插入执行程序 \_insert()
39. **template**<**class** Key , **class** Value , **class** KeyOfValue , **class** Compare , **class** Alloc>
40. **typename**<Key , Value , KeyOfValue , Compare , Alloc>::\_insert(base\_ptr x\_ , base\_ptr y\_ , **const** Value &v)
41. {
42. // 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
43. link\_type x = (link\_type) x\_;
44. link\_type y = (link\_type) y\_;
45. link\_type z;

48. // key\_compare 是键值大小比较准则。应该会是个function object
49. **if**(y == header || x != 0 || key\_compare(KeyOfValue()(v) , key(y) ))
50. {
51. z = create\_node(v);    // 产生一个新节点
52. left(y) = z;           // 这使得当y即为header时，leftmost() = z
53. **if**(y == header)
54. {
55. root() = z;
56. rightmost() = z;
57. }
58. **else** **if**(y == leftmost())     // 如果y为最左节点
59. leftmost() = z;          // 维护leftmost()，使它永远指向最左节点
60. }
61. **else**
62. {
63. z = create\_node(v);        // 产生一个新节点
64. right(y) = z;              // 令新节点成为插入点之父节点y的右子节点
65. **if**(y == rightmost())
66. rightmost() = z;       // 维护rightmost()，使它永远指向最右节点
67. }
68. parent(z) = y;      // 设定新节点的父节点
69. left(z) = 0;        // 设定新节点的左子节点
70. right(z) = 0;       // 设定新节点的右子节点
71. // 新节点的颜色将在\_rb\_tree\_rebalance()设定（并调整）
72. \_rb\_tree\_rebalance(z , header->parent);      // 参数一为新增节点，参数二为根节点root
73. ++node\_count;       // 节点数累加
74. **return** iterator(z);  // 返回一个迭代器，指向新增节点
75. }



80. // 全局函数
81. // 重新令树形平衡（改变颜色及旋转树形）
82. // 参数一为新增节点，参数二为根节点root
83. **inline** **void** \_rb\_tree\_rebalance(\_rb\_tree\_node\_base\* x , \_rb\_tree\_node\_base\*& root)
84. {
85. x->color = \_rb\_tree\_red;    //新节点必为红
86. **while**(x != root && x->parent->color == \_rb\_tree\_red)    // 父节点为红
87. {
88. **if**(x->parent == x->parent->parent->left)      // 父节点为祖父节点之左子节点
89. {
90. \_rb\_tree\_node\_base\* y = x->parent->parent->right;    // 令y为伯父节点
91. **if**(y && y->color == \_rb\_tree\_red)    // 伯父节点存在，且为红
92. {
93. x->parent->color = \_rb\_tree\_black;           // 更改父节点为黑色
94. y->color = \_rb\_tree\_black;                   // 更改伯父节点为黑色
95. x->parent->parent->color = \_rb\_tree\_red;     // 更改祖父节点为红色
96. x = x->parent->parent;
97. }
98. **else**    // 无伯父节点，或伯父节点为黑色
99. {
100. **if**(x == x->parent->right)   // 如果新节点为父节点之右子节点
101. {
102. x = x->parent;
103. \_rb\_tree\_rotate\_left(x , root);    // 第一个参数为左旋点
104. }
105. x->parent->color = \_rb\_tree\_black;     // 改变颜色
106. x->parent->parent->color = \_rb\_tree\_red;
107. \_rb\_tree\_rotate\_right(x->parent->parent , root);    // 第一个参数为右旋点
108. }
109. }
110. **else**          // 父节点为祖父节点之右子节点
111. {
112. \_rb\_tree\_node\_base\* y = x->parent->parent->left;    // 令y为伯父节点
113. **if**(y && y->color == \_rb\_tree\_red)    // 有伯父节点，且为红
114. {
115. x->parent->color = \_rb\_tree\_black;           // 更改父节点为黑色
116. y->color = \_rb\_tree\_black;                   // 更改伯父节点为黑色
117. x->parent->parent->color = \_rb\_tree\_red;     // 更改祖父节点为红色
118. x = x->parent->parent;          // 准备继续往上层检查
119. }
120. **else**    // 无伯父节点，或伯父节点为黑色
121. {
122. **if**(x == x->parent->left)        // 如果新节点为父节点之左子节点
123. {
124. x = x->parent;
125. \_rb\_tree\_rotate\_right(x , root);    // 第一个参数为右旋点
126. }
127. x->parent->color = \_rb\_tree\_black;     // 改变颜色
128. x->parent->parent->color = \_rb\_tree\_red;
129. \_rb\_tree\_rotate\_left(x->parent->parent , root);    // 第一个参数为左旋点
130. }
131. }
132. }//while
133. root->color = \_rb\_tree\_black;    // 根节点永远为黑色
134. }



139. // 左旋函数
140. **inline** **void** \_rb\_tree\_rotate\_left(\_rb\_tree\_node\_base\* x , \_rb\_tree\_node\_base\*& root)
141. {
142. // x 为旋转点
143. \_rb\_tree\_node\_base\* y = x->right;          // 令y为旋转点的右子节点
144. x->right = y->left;
145. **if**(y->left != 0)
146. y->left->parent = x;           // 别忘了回马枪设定父节点
147. y->parent = x->parent;

150. // 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）
151. **if**(x == root)    // x为根节点
152. root = y;
153. **else** **if**(x == x->parent->left)         // x为其父节点的左子节点
154. x->parent->left = y;
155. **else**                                  // x为其父节点的右子节点
156. x->parent->right = y;
157. y->left = x;
158. x->parent = y;
159. }



164. // 右旋函数
165. **inline** **void** \_rb\_tree\_rotate\_right(\_rb\_tree\_node\_base\* x , \_rb\_tree\_node\_base\*& root)
166. {
167. // x 为旋转点
168. \_rb\_tree\_node\_base\* y = x->left;          // 令y为旋转点的左子节点
169. x->left = y->right;
170. **if**(y->right != 0)
171. y->right->parent = x;           // 别忘了回马枪设定父节点
172. y->parent = x->parent;

175. // 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）
176. **if**(x == root)
177. root = y;
178. **else** **if**(x == x->parent->right)         // x为其父节点的右子节点
179. x->parent->right = y;
180. **else**                                  // x为其父节点的左子节点
181. x->parent->left = y;
182. y->right = x;
183. x->parent = y;
184. }

## linux 源码里的红黑树插入

自认为好读多了 O(∩\_∩)O~

**[cpp]** [view plaincopy](http://blog.csdn.net/haidao2009/article/details/8077514)

1. /\*\*
2. \* node是新插入的结点，有着默认的红色。
3. \* 本函数检查是否有违背红黑树性质的地方，并进行纠正。
4. \*/
5. **void** rb\_insert\_color(rb\_node \*node, rb\_root \*root) {
6. rb\_node \*parent, \*gp;
8. /\*\*
9. \* 如果有父节点且父节点是红色，进行树的调整以保证树的性质。
10. \*/
11. **while** ((parent = rb\_parent(node)) && rb\_is\_red(parent)) {
12. gp = rb\_parent(parent);
14. **if** (parent == gp->left) {
15. /\*\*
16. \* 父节点是祖父节点左子的情况。
17. \* "else"中的情况左右相反，这里只注释"if"里的代码。
18. \*/
19. **register** rb\_node \*uncle = gp->right;
20. **if** (uncle && rb\_is\_red(uncle)) {       <span style="color:#ff0000;">//对应图一</span>
21. /\*\*
22. \* 如果有红叔，则将红叔与红父均涂黑，并将祖父节点涂红。
23. \*/
24. rb\_set\_black(parent);
25. rb\_set\_black(uncle);
26. rb\_set\_red(gp);
27. /\*\*
28. \* 现在简单路径中黑节点个数仍然平衡，但祖父变成了红色，
29. \* 我们不确定有没有造成父子均红的情况，所以需要对祖父节点进行下一轮修复。
30. \*/
31. node = gp;  <span style="color:#ff0000;">//这里要递归fix</span>
32. **continue**;
33. }
35. /\*\*
36. \* 现在是叔节点为空或为黑的情况。
37. \*/
38. **if** (node == parent->right) {
39. /\*\*
40. \* 如果新节点是父节点的右子，对父节点进行左旋。
41. \* 旋转后树仍然平衡，但新节点占了原父节点的位子。
42. \* 这两个节点交换角色后，新的父节点是红的，其左子也是红的。
43. \*/
44. \_rb\_rotate\_left(parent, root);   <span style="color:#ff0000;"> //对应图四-》图五</span>
45. **register** rb\_node \*tmp = node;
46. node = parent;
47. parent = tmp;
48. }
49. /\*\*
50. \* 此时父红左子红，树平衡但有连续红节点。
51. \*
52. \* 父涂黑，祖父涂红，再对祖父右旋，树即调整到合法状态。
53. \*/
54. rb\_set\_black(parent);     <span style="color:#ff0000;">//对应图三</span>
55. rb\_set\_red(gp);
56. \_rb\_rotate\_right(gp, root);
57. **return**;
58. } **else** {
59. **register** rb\_node \*uncle = gp->left;
60. **if** (uncle && rb\_is\_red(uncle)) {
61. rb\_set\_black(parent);
62. rb\_set\_black(uncle);
63. rb\_set\_red(gp);
64. node = gp;
65. **continue**;
66. }
68. **if** (node == parent->left) {
69. \_rb\_rotate\_right(parent, root);
70. **register** rb\_node \*tmp = node;
71. node = parent;
72. parent = tmp;
73. }
74. rb\_set\_black(parent);
75. rb\_set\_red(gp);
76. \_rb\_rotate\_left(gp, root);
77. **return**;
78. }
79. }
80. /\*\*
81. \* 若无父节点，只需将node涂黑；
82. \* 若父节点为黑，插入红节点不影响树的性质。
83. \* 循环体后直接将node涂黑，可以同时保证以上两点。
84. \*/
85. rb\_set\_black(root->node);
86. }

# [彻底搞懂红黑树（三）](http://blog.csdn.net/haidao2009/article/details/8078155)

分类： [数据结构](http://blog.csdn.net/haidao2009/article/category/1258364)2012-10-16 17:29 2133人阅读 [评论](http://blog.csdn.net/haidao2009/article/details/8078155#comments)(4) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/haidao2009/article/details/8078155#report)

[算法](http://www.csdn.net/tag/%e7%ae%97%e6%b3%95)

目录[(?)[+]](http://blog.csdn.net/haidao2009/article/details/8078155)

## 四 红黑树的插入

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。

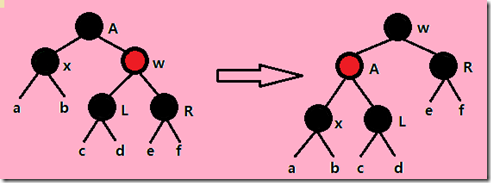
      我们回忆一下普通二叉树的节点删除方法：Z指向需要删除的节点，Y指向实质结构上被删除的结点，如果Z节点只有一个子节点或没有子节点，那么Y就是指向Z指向的节点。如果Z节点有两个子节点，那么Y指向Z节点的后继节点（其实前趋也是一样的），而Z的后继节点绝对不可能有左子树。因此，仅从结构来看，二叉树上实质被删除的节点最多只可能有一个子树。  
现在我们来看红黑性质的恢复过程：  
      ①如果Y指向的节点是个红色节点，那么直接删除掉Y以后，红黑性质不会被破坏。操作结束。  
      ②如果Y指向的节点是个黑色节点，那么就有几条红黑性质可能受到破坏了。首先是包含Y节点的所有路径，黑高度都减少了一（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。（这个很重要）

      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。

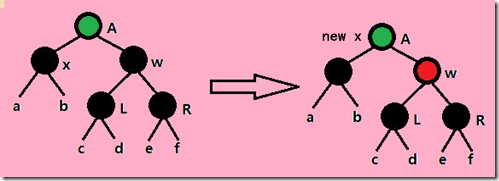
二、如果X是双黑色，那么我们希望把这种情况向上推一直推到根节点（调整树结构和颜色，X的指向新的双黑色节点，X不断向上移动），让根节点具双黑色，这时，直接把X的一层黑色去掉就行了（因为根节点被包含在所有的路径上，所以这样做所有路径同时黑高减少一，不会破坏红黑特征）。（双黑调节就是按照这种思路：无限把双黑节点上推，直到出现第四种情况）

      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。也就是说至少两层黑。其实可以说，这就是一个递归函数的递归公式，只不过情况较多而已。可以参看july的最长子字符串序列，和这种情况类似）。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。这些个节点中。  
      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。（本来x 为双黑，只要把A再涂成双黑就完事了，和我们之前说过的要把双黑往上推一样，但那样对问题根本没一点影响，还不如不做。其实旋转是可以想象到的，左边x的删除了一个黑节点，那必然导致右边重了，要左旋转）



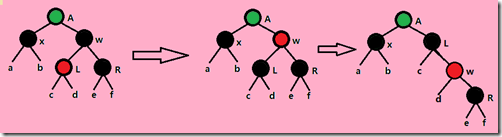
图一

    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。（因为A下面是平衡的，所以不用改动，但x要把一层黑给顶上去，那w这边就相当于多了一层黑，所以把w的黑给去掉，成为红色。至于A怎么办，那要看A本身的颜色是什么了。如下:）



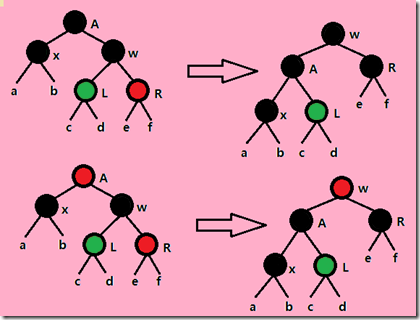
图二

      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。(L旋转后，左子树的性质是不会变的，但右子树多了个黑的w 就有问题了，所以要把w改为红色。其实case3是把问题转换为case4.)



图三

    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） （其实是相当于 x这边多了一层黑，怎么办呢，就把A点给按下去，这样x两层黑就变为 x本身黑+A的黑。A是下去了，那自然要把w给提上来，w提上来之后L就变到左边去了。L子树保持不变，原因很简单，都是两层黑+L本身。但w提上来了，可这边少了一层黑啊，所以直接把R变为黑，ok 问题解决了。）



图四

      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；

## linux 源码中红黑树的删除的修复

**[cpp]** [view plaincopy](http://blog.csdn.net/haidao2009/article/details/8078155)

1. /\*\*
2. \* 前置环境：在node与parent之前，刚刚删除了一个黑色节点。现在树很可能不平衡，
3. \* node与parent也可能红色冲突。
4. \* 本函数进行树的性质的修正，以使树恢复平衡。在一些情况下问题会转移到上一层节点，
5. \* 则须对上一层节点进行递归检查与修正。本函数中的while循环实际上实现了这种递归。
6. \*
7. \* 提示：这是红黑树里最绕的地方，看不明白可以多画画图。
8. \*/
9. **static** **void** \_rb\_erase\_color(rb\_node \*node, rb\_node \*parent, rb\_root \*root) {
10. /\*\*
11. \* other用来保存兄弟节点，这是树的修正过程中一个重要的参考节点。
12. \*/
13. rb\_node \*other;
14. /\*\*
15. \* 循环条件：node不是红节点且node不是根节点。
16. \* 解释：对于红节点或根节点，直接涂黑即可解决问题。
17. \*/
18. **while** ((!node || rb\_is\_black(node)) && node != root->node) {
19. /\*\*
20. \* 为方便程序处理各种情况，这里和节点插入一样将问题分成了左右对称的两类。
21. \* if-else两个分枝里代码逻辑完全相同，只是左右相反。所以我们只研究node是parent左子的情况。
22. \*
23. \* 在开始之前，我们先总结一下当前状态：
24. \* 1:因为删除的是黑色节点，所以node与parent都有可能是红色节点。
25. \* 2:node与parent之间少了一个黑色节点，则所有通过node的路径都少了一个黑色节点，不妨画图时用-1标出来；
26. \* 但node的兄弟节点（node一定有兄弟，可以根据删除前树的平衡性质来反推）高度并未变化，可以记作0。
27. \*
28. \* 提示：在进行旋转、涂色等操作时，可以画图观查平衡状态的变化。
29. \*/
30. **if** (parent->left == node) {  <span style="color:#ff0000;">//左边  把other换位brother更容易懂</span>
31. other = parent->right;
32. **if** (rb\_is\_red(other)) {  //case 1
33. /\*\*
34. \* 如果兄弟节点是红色，则父节点是黑色，交换父、兄节点的颜色，并对父节点进行左旋。
35. \* 旋转后，兄节点占了老的父节点位置，且和老的父节点颜色相同，所以不会向上造成颜色冲突。
36. \* 我们仍然以老的父节点为父节点来看，现在的状态是：
37. \* 父节点右子保持平衡，只有经过node的路径少了一个黑色节点。
38. \* 现在问题和之前相似，但node有了一个黑色的兄弟（还有一个红色父亲）。
39. \*/
40. rb\_set\_black(other);
41. rb\_set\_red(parent);
42. \_rb\_rotate\_left(parent, root);
43. /\*\*
44. \* other指向新的兄弟节点。other现在必然是一个黑色节点，而不会是空。这一点可以根据旋转之前树的性质反证。

**[cpp]** [view plaincopy](http://blog.csdn.net/haidao2009/article/details/8078155)

1. 递归矫正

**[cpp]** [view plaincopy](http://blog.csdn.net/haidao2009/article/details/8078155)

1. \*/
2. other = parent->right;
3. }
4. /\*\*
5. \* 此时状态：
6. \* node有黑色兄弟，父亲可能黑也可能红。

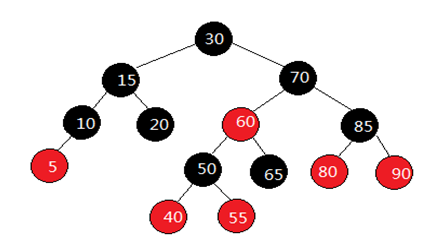
**[cpp]** [view plaincopy](http://blog.csdn.net/haidao2009/article/details/8078155)

1. \*儿子都为黑
2. \*/
3. **if** ((!other->left || rb\_is\_black(other->left)) &&
4. (!other->right || rb\_is\_black(other->right))) {
5. /\*\*
6. \* 如果other没有红色子节点，那我们就可以把other涂红，并向上转移问题。
7. \* other涂红的后果是，other分枝少了一个黑节点，与node分枝保持了平衡，但parent整体少了一个黑色节点。
8. \* 细心的人可能会发现，如果父亲是红色的，父亲与兄弟有颜色冲突，直接向上转移能纠正吗？当然是可以的。
9. \* 向上一层之后，while里的黑节点判断失败，会直接执行while后面的语句，直接将parent涂黑，则树恢复平衡。
10. \*/
11. rb\_set\_red(other);
12. node = parent;
13. parent = rb\_parent(node);
14. } **else** {
15. /\*\*
16. \* 现在黑兄弟有红子节点，父亲颜色未知。
17. \*/
18. **if** (!other->right || rb\_is\_black(other->right)) {
19. /\*\*
20. \* 如果黑兄弟右节点为空或为黑，则左节点一定是红的，我们想办法把它调整为右子为红。
21. \* 至于为什么，看后面就知了。
22. \* other->left与other交换颜色，对other进行右旋，other指向新的兄弟。根据右旋转的特点，
23. \* 可知现在other仍然是黑色，且它有了一个红色右子。同时other分枝高度不变，颜色也没有冲突。
24. \*/
25. rb\_set\_black(other->left);
26. rb\_set\_red(other);
27. \_rb\_rotate\_right(other, root);
28. other = parent->right;
29. }
30. /\*\*
31. \* 此时状态：黑兄弟有红色右子节点。
32. \* 不管parent是什么颜色，把other涂成父亲的颜色（之后旋转，other占据父亲的位置，向上没有颜色冲突），
33. \* 把父亲涂黑，把黑兄的other涂黑，这时node分枝高度可能有变化也可能没变化，other分枝多了一个黑节点。
34. \* 现在对父亲进行左旋转。旋转后的情况是右边分枝（原other右子）少了一个黑节点，重归平衡；
35. \* 左边分枝则增加了一个黑节点，也恢复了平衡。此时也没有颜色冲突
36. \*/
37. rb\_set\_color(other, rb\_color(parent));
38. rb\_set\_black(parent);
39. rb\_set\_black(other->right);
40. \_rb\_rotate\_left(parent, root);
41. /\*\*
42. \* 树已平衡，node置为根节点，并跳出循环。
43. \*  检测root节点是否黑色
44. \*/
45. node = root->node;
46. **break**;
47. }
48. } **else** {
49. other = parent->left;
50. **if** (rb\_is\_red(other)) {
51. rb\_set\_black(other);
52. rb\_set\_red(parent);
53. \_rb\_rotate\_right(parent, root);
54. other = parent->left;
55. }
56. **if** ((!other->left || rb\_is\_black(other->left)) &&
57. (!other->right || rb\_is\_black(other->right))) {
58. rb\_set\_red(other);
59. node = parent;
60. parent = rb\_parent(node);
61. } **else** {
62. **if** (!other->left || rb\_is\_black(other->left)) {
63. rb\_set\_black(other->right);
64. rb\_set\_red(other);
65. \_rb\_rotate\_left(other, root);
66. other = parent->left;
67. }
68. rb\_set\_color(other, rb\_color(parent));
69. rb\_set\_black(parent);
70. rb\_set\_black(other->left);
71. \_rb\_rotate\_right(parent, root);
72. node = root->node;
73. **break**;
74. }
75. }
76. }
77. /\*\*
78. \* 对于红节点或根节点，直接涂黑即可解决问题。
79. \*/
80. **if** (node)
81. rb\_set\_black(node);

# [STL源码剖析---红黑树原理详解上](http://blog.csdn.net/hackbuteer1/article/details/7740956)

## 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：



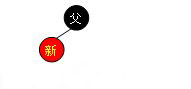
假设我们为上图分别插入节点3、8、35、75，根据二叉搜索树的规则，插入这四个节点后，我们会发现它们都破坏了红黑树的规则，因此我们必须调整树形，也就是旋转树形并改变节点的颜色。

## 二、红黑树上结点的插入

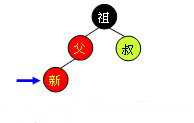
      在讨论红黑树的插入操作之前必须要明白，任何一个即将插入的新结点的初始颜色都为红色。这一点很容易理解，因为插入黑点会增加某条路径上黑结点的数目，从而导致整棵树黑高度的不平衡。但如果新结点的父结点为红色时（如下图所示），将会违反红黑树的性质：一条路径上不能出现相邻的两个红色结点。这时就需要通过一系列操作来使红黑树保持平衡。



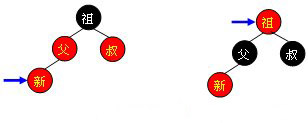
      为了清楚地表示插入操作以下在结点中使用“新”字表示一个新插入的结点；使用“父”字表示新插入点的父结点；使用“叔”字表示“父”结点的兄弟结点；使用“祖”字表示“父”结点的父结点。插入操作分为以下几种情况：  
1、黑父  
     如下图所示，如果新节点的父结点为黑色结点，那么插入一个红点将不会影响红黑树的平衡，此时插入操作完成。红黑树比AVL树优秀的地方之一在于黑父的情况比较常见，从而使红黑树需要旋转的几率相对AVL树来说会少一些。



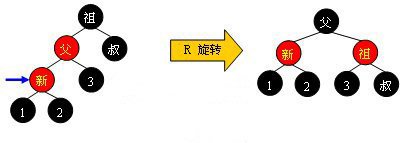
2、红父  
     如果新节点的父结点为红色，这时就需要进行一系列操作以保证整棵树红黑性质。如下图所示，由于父结点为红色，此时可以判定，祖父结点必定为黑色。这时需要根据叔父结点的颜色来决定做什么样的操作。青色结点表示颜色未知。由于有可能需要根结点到新点的路径上进行多次旋转操作，而每次进行不平衡判断的起始点（我们可将其视为新点）都不一样。所以我们在此使用一个蓝色箭头指向这个起始点，并称之为判定点。



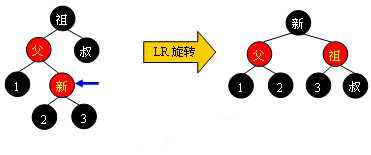
2.1 红叔  
当叔父结点为红色时，如下图所示，无需进行旋转操作，只要将父和叔结点变为黑色，将祖父结点变为红色即可。但由于祖父结点的父结点有可能为红色，从而违反红黑树性质。此时必须将祖父结点作为新的判定点继续向上（**迭代**）进行平衡操作。



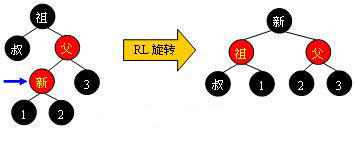
需要注意的是，无论“父节点”在“叔节点”的左边还是右边，无论“新节点”是“父节点”的左孩子还是右孩子，它们的操作都是完全一样的（其实这种情况包括4种，只需调整颜色，不需要旋转树形）。  
2.2 黑叔  
当叔父结点为黑色时，需要进行旋转，以下图示了所有的旋转可能：  
Case 1:



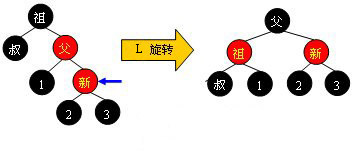
Case 2:



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 plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7740956)

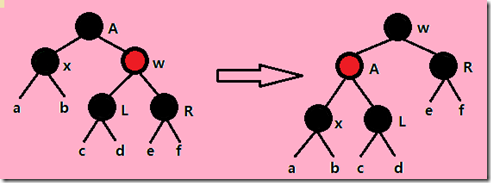
1. // 元素插入操作  insert\_unique()
2. // 插入新值：节点键值不允许重复，若重复则插入无效
3. // 注意，返回值是个pair，第一个元素是个红黑树迭代器，指向新增节点
4. // 第二个元素表示插入成功与否
5. **template**<**class** Key , **class** Value , **class** KeyOfValue , **class** Compare , **class** Alloc>
6. pair<**typename** rb\_tree<Key , Value , KeyOfValue , Compare , Alloc>::iterator , **bool**>
7. rb\_tree<Key , Value , KeyOfValue , Compare , Alloc>::insert\_unique(**const** Value &v)
8. {
9. rb\_tree\_node\* y = header;    // 根节点root的父节点
10. rb\_tree\_node\* x = root();    // 从根节点开始
11. **bool** comp = **true**;
12. **while**(x != 0)
13. {
14. y = x;
15. comp = key\_compare(KeyOfValue()(v) , key(x));    // v键值小于目前节点之键值？
16. x = comp ? left(x) : right(x);   // 遇“大”则往左，遇“小于或等于”则往右
17. }
18. // 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）
19. iterator j = iterator(y);     // 令迭代器j指向插入点之父节点y
20. **if**(comp)     // 如果离开while循环时comp为真（表示遇“大”，将插入于左侧）
21. {
22. **if**(j == begin())    // 如果插入点之父节点为最左节点
23. **return** pair<iterator , **bool**>(\_insert(x , y , z) , **true**);
24. **else**     // 否则（插入点之父节点不为最左节点）
25. --j;   // 调整j，回头准备测试
26. }
27. **if**(key\_compare(key(j.node) , KeyOfValue()(v) ))
28. // 新键值不与既有节点之键值重复，于是以下执行安插操作
29. **return** pair<iterator , **bool**>(\_insert(x , y , z) , **true**);
30. // 以上，x为新值插入点，y为插入点之父节点，v为新值
32. // 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
33. **return** pair<iterator , **bool**>(j , **false**);
34. }
36. // 真正地插入执行程序 \_insert()
37. **template**<**class** Key , **class** Value , **class** KeyOfValue , **class** Compare , **class** Alloc>
38. **typename**<Key , Value , KeyOfValue , Compare , Alloc>::\_insert(base\_ptr x\_ , base\_ptr y\_ , **const** Value &v)
39. {
40. // 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
41. link\_type x = (link\_type) x\_;
42. link\_type y = (link\_type) y\_;
43. link\_type z;
45. // key\_compare 是键值大小比较准则。应该会是个function object
46. **if**(y == header || x != 0 || key\_compare(KeyOfValue()(v) , key(y) ))
47. {
48. z = create\_node(v);    // 产生一个新节点
49. left(y) = z;           // 这使得当y即为header时，leftmost() = z
50. **if**(y == header)
51. {
52. root() = z;
53. rightmost() = z;
54. }
55. **else** **if**(y == leftmost())     // 如果y为最左节点
56. leftmost() = z;          // 维护leftmost()，使它永远指向最左节点
57. }
58. **else**
59. {
60. z = create\_node(v);        // 产生一个新节点
61. right(y) = z;              // 令新节点成为插入点之父节点y的右子节点
62. **if**(y == rightmost())
63. rightmost() = z;       // 维护rightmost()，使它永远指向最右节点
64. }
65. parent(z) = y;      // 设定新节点的父节点
66. left(z) = 0;        // 设定新节点的左子节点
67. right(z) = 0;       // 设定新节点的右子节点
68. // 新节点的颜色将在\_rb\_tree\_rebalance()设定（并调整）
69. \_rb\_tree\_rebalance(z , header->parent);      // 参数一为新增节点，参数二为根节点root
70. ++node\_count;       // 节点数累加
71. **return** iterator(z);  // 返回一个迭代器，指向新增节点
72. }

75. // 全局函数
76. // 重新令树形平衡（改变颜色及旋转树形）
77. // 参数一为新增节点，参数二为根节点root
78. **inline** **void** \_rb\_tree\_rebalance(\_rb\_tree\_node\_base\* x , \_rb\_tree\_node\_base\*& root)
79. {
80. x->color = \_rb\_tree\_red;    //新节点必为红
81. **while**(x != root && x->parent->color == \_rb\_tree\_red)    // 父节点为红
82. {
83. **if**(x->parent == x->parent->parent->left)      // 父节点为祖父节点之左子节点
84. {
85. \_rb\_tree\_node\_base\* y = x->parent->parent->right;    // 令y为伯父节点
86. **if**(y && y->color == \_rb\_tree\_red)    // 伯父节点存在，且为红
87. {
88. x->parent->color = \_rb\_tree\_black;           // 更改父节点为黑色
89. y->color = \_rb\_tree\_black;                   // 更改伯父节点为黑色
90. x->parent->parent->color = \_rb\_tree\_red;     // 更改祖父节点为红色
91. x = x->parent->parent;
92. }
93. **else**    // 无伯父节点，或伯父节点为黑色
94. {
95. **if**(x == x->parent->right)   // 如果新节点为父节点之右子节点
96. {
97. x = x->parent;
98. \_rb\_tree\_rotate\_left(x , root);    // 第一个参数为左旋点
99. }
100. x->parent->color = \_rb\_tree\_black;     // 改变颜色
101. x->parent->parent->color = \_rb\_tree\_red;
102. \_rb\_tree\_rotate\_right(x->parent->parent , root);    // 第一个参数为右旋点
103. }
104. }
105. **else**          // 父节点为祖父节点之右子节点
106. {
107. \_rb\_tree\_node\_base\* y = x->parent->parent->left;    // 令y为伯父节点
108. **if**(y && y->color == \_rb\_tree\_red)    // 有伯父节点，且为红
109. {
110. x->parent->color = \_rb\_tree\_black;           // 更改父节点为黑色
111. y->color = \_rb\_tree\_black;                   // 更改伯父节点为黑色
112. 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. }

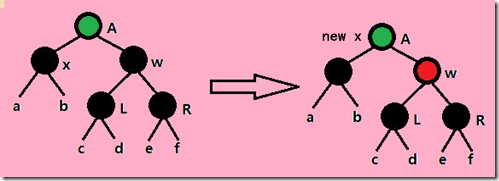
132. // 左旋函数
133. **inline** **void** \_rb\_tree\_rotate\_left(\_rb\_tree\_node\_base\* x , \_rb\_tree\_node\_base\*& root)
134. {
135. // x 为旋转点
136. \_rb\_tree\_node\_base\* y = x->right;          // 令y为旋转点的右子节点
137. x->right = y->left;
138. **if**(y->left != 0)
139. y->left->parent = x;           // 别忘了回马枪设定父节点
140. y->parent = x->parent;
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. }

154. // 右旋函数
155. **inline** **void** \_rb\_tree\_rotate\_right(\_rb\_tree\_node\_base\* x , \_rb\_tree\_node\_base\*& 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;
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. }

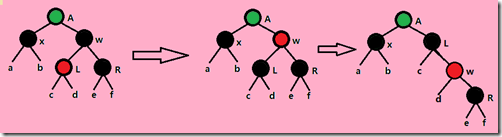
 算法导论书上给出的红黑树的性质如下，跟STL源码剖析书上面的4条性质大同小异。  
      1、每个结点或是红色的，或是黑色的  
      2、根节点是黑色的  
      3、每个叶结点（NIL）是黑色的  
      4、如果一个节点是红色的，则它的两个儿子都是黑色的。  
      5、对于每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑色结点。  
      从红黑树上删除一个节点，可以先用普通二叉搜索树的方法，将节点从红黑树上删除掉，然后再将被破坏的红黑性质进行恢复。  
      我们回忆一下普通二叉树的节点删除方法：Z指向需要删除的节点，Y指向实质结构上被删除的结点，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。



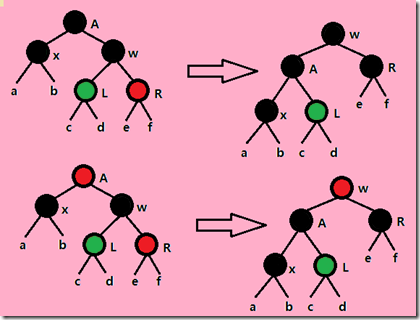
    Case2：X的兄弟节点W是黑色的，而且W的两个子节点都是黑色的。此时可以将X的一重黑色和W的黑色同时去掉，而转加给他们的父节点上，这是X就指向它的父节点了，因此此时父节点具有双重颜色了。这一重黑色节点上移。



      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 plaincopy](http://blog.csdn.net/hackbuteer1/article/details/7760584)

1. #include<iostream>
2. **using** **namespace** std;
4. // 定义节点颜色
5. **enum** COLOR
6. {
7. BLACK = 0,
8. RED
9. };
11. // 红黑树节点
12. **typedef** **struct** RB\_Tree\_Node
13. {
14. **int** key;
15. **struct** RB\_Tree\_Node \*left;
16. **struct** RB\_Tree\_Node \*right;
17. **struct** RB\_Tree\_Node \*parent;
18. unsigned **char** RB\_COLOR;
19. }RB\_Node;
21. // 红黑树，包含一个指向根节点的指针
22. **typedef** **struct** RBTree
23. {
24. RB\_Node\* root;
25. }\*RB\_Tree;
27. // 红黑树的NIL节点
28. **static** RB\_Tree\_Node NIL = {0, 0, 0, 0, BLACK};
30. #define PNIL (&NIL)   // NIL节点地址
32. **void** Init\_RBTree(RB\_Tree pTree) // 初始化一棵红黑树
33. {
34. pTree->root = PNIL;
35. }
37. // 查找最小键值节点
38. RB\_Node\* RBTREE\_MIN(RB\_Node\* pRoot)
39. {
40. **while** (PNIL != pRoot->left)
41. {
42. pRoot = pRoot->left;
43. }
44. **return** pRoot;
45. }

48. /\*
49. 15
50. /    \
51. /      \
52. /        \
53. 6          18
54. /  \       /  \
55. /    \     /    \
56. 3      7   17    20
57. /  \     \
58. /    \     \
59. 2      4     13
60. /
61. /
62. 9
63. \*/
64. // 查找指定节点的后继节点
65. RB\_Node\* RBTREE\_SUCCESSOR(RB\_Node\*  pRoot)
66. {
67. **if** (PNIL != pRoot->right)    // 查找图中6的后继节点时就调用RBTREE\_MIN函数
68. {
69. **return** RBTREE\_MIN(pRoot->right);
70. }
71. // 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）
72. RB\_Node\* pParent = pRoot->parent;
73. **while**((PNIL != pParent) && (pRoot == pParent->right))
74. {
75. pRoot = pParent;
76. pParent = pRoot->parent;
77. }
78. **return** pParent;
79. }
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
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. {
118. pDel->key = rel\_delete\_point->key;
119. }
120. **if**(rel\_delete\_point->RB\_COLOR == BLACK)
121. {
122. DeleteFixUp(pTree , delete\_point\_child);
123. }
124. **return** rel\_delete\_point;
125. }

128. /\*
129. 算法导论上的描述如下：
130. RB-DELETE-FIXUP(T, x)
131. 1 while x ≠ root[T] and color[x] = BLACK
132. 2     do if x = left[p[x]]
133. 3           then w ← right[p[x]]
134. 4                if color[w] = RED
135. 5                   then color[w] ← BLACK                           Case 1
136. 6                        color[p[x]] ← RED                          Case 1
137. 7                        LEFT-ROTATE(T, p[x])                       Case 1
138. 8                        w ← right[p[x]]                            Case 1
139. 9                if color[left[w]] = BLACK and color[right[w]] = BLACK
140. 10                   then color[w] ← RED                            Case 2
141. 11                        x p[x]                                    Case 2
142. 12                   else if color[right[w]] = BLACK
143. 13                           then color[left[w]] ← BLACK            Case 3
144. 14                                color[w] ← RED                    Case 3
145. 15                                RIGHT-ROTATE(T, w)                Case 3
146. 16                                w ← right[p[x]]                   Case 3
147. 17                         color[w] ← color[p[x]]                   Case 4
148. 18                         color[p[x]] ← BLACK                      Case 4
149. 19                         color[right[w]] ← BLACK                  Case 4
150. 20                         LEFT-ROTATE(T, p[x])                     Case 4
151. 21                         x ← root[T]                              Case 4
152. 22        else (same as then clause with "right" and "left" exchanged)
153. 23 color[x] ← BLACK
154. \*/
155. //接下来的工作，很简单，即把上述伪代码改写成c++代码即可
156. **void** DeleteFixUp(RB\_Tree pTree , RB\_Node\* node)
157. {
158. **while**(node != pTree->root && node->RB\_COLOR == BLACK)
159. {
160. **if**(node == node->parent->left)
161. {
162. RB\_Node\* brother = node->parent->right;
163. **if**(brother->RB\_COLOR==RED)   //情况1：x的兄弟w是红色的。
164. {
165. brother->RB\_COLOR = BLACK;
166. node->parent->RB\_COLOR = RED;
167. RotateLeft(node->parent);
168. }
169. **else**     //情况2：x的兄弟w是黑色的，
170. {
171. **if**(brother->left->RB\_COLOR == BLACK && brother->right->RB\_COLOR == BLACK)  //w的两个孩子都是黑色的
172. {
173. brother->RB\_COLOR = RED;
174. node = node->parent;
175. }
176. **else**
177. {
178. **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）。
179. {
180. brother->RB\_COLOR = RED;
181. brother->left->RB\_COLOR = BLACK;
182. RotateRight(brother);
183. brother = node->parent->right;      //情况3转换为情况4
184. }
185. //情况4：x的兄弟w是黑色的，且w的右孩子时红色的
186. brother->RB\_COLOR = node->parent->RB\_COLOR;
187. node->parent->RB\_COLOR = BLACK;
188. brother->right->RB\_COLOR = BLACK;
189. RotateLeft(node->parent);
190. node = pTree->root;
191. }//else
192. }//else
193. }
194. **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。
195. {
196. RB\_Node\* brother = node->parent->left;
197. **if**(brother->RB\_COLOR == RED)
198. {
199. brother->RB\_COLOR = BLACK;
200. node->parent->RB\_COLOR = RED;
201. RotateRight(node->parent);
202. }
203. **else**
204. {
205. **if**(brother->left->RB\_COLOR==BLACK && brother->right->RB\_COLOR == BLACK)
206. {
207. brother->RB\_COLOR = RED;
208. node = node->parent;
209. }
210. **else**
211. {
212. **if**(brother->left->RB\_COLOR==BLACK)
213. {
214. brother->RB\_COLOR = RED;
215. brother->right->RB\_COLOR = BLACK;
216. RotateLeft(brother);
217. brother = node->parent->left;      //情况3转换为情况4
218. }
219. brother->RB\_COLOR = node->parent->RB\_COLOR;
220. node->parent->RB\_COLOR = BLACK;
221. brother->left->RB\_COLOR = BLACK;
222. RotateRight(node->parent);
223. node = pTree->root;
224. }
225. }
226. }
227. }//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. }