红黑树原理详解

用于《C 语言常用数据结构源代码全注解－红黑树源码》

作　　者：余 强

日　　期：2009-12-4

版　　本：V1.0

协　　议：GFDL

博　　客：有水的地方就有余

院　　校：CQU

组　　织：工程信息化(嵌入式)实验室

联 系 我：yuembed@126.com

前言：

之所以要写这篇文章，第一个目的是为了各位朋友在查看我写的源代码之前有一个可以理解理论的文章，因为红黑树还是有点难的，如果不想搞懂理论，而直接看代码，那绝对是云里雾里，不知所云。第二个目的是我觉得网上虽然后不少我文章也在讲，但是我就是理解不上有点困难，在我参考了很多文章之后，认真阅读才慢慢摸透了其中的原理，所以我想用

自己的方式来表达，希望有助于各位的朋友理解。
你可以在这里下载配套源代码

红黑树由来：

它是在 1972 年由 Rudolf Bayer 发明的，他称之为"对称二叉 B 树"，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的一篇论文中获得的。它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的: 它可以在 O(log n)时间内做查找，插入和删除，这里的 n 是树中元素的数目。

红黑树性质：

1. 每个结点或红或黑。
2. 根结点为黑色。
3. 每个叶结点(实际上就是 NULL 指针)都是黑色的。
4. 如果一个结点是红色的,那么它的周边 3 个节点都是黑色的。
5. 对于每个结点,从该结点到其所有子孙叶结点的路径中所包含的黑色结点个数都一样。

讨论的前提：

1，我们只讨论往树的左边和从树的左边删除的情况，与之对称的情况一样。

2，假设我们要删除一个元素的方法都是采取删除后继节点，而非前驱节点。

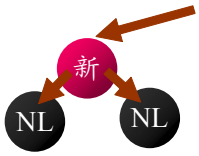
3，NL 或全黑表示黑空节点，也可以不用画出。

4，“=>”这个符号我们用来表示“变成”的意思。

一. 插入

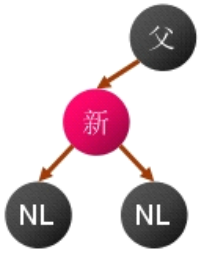
当红黑树中没有节点的时候，新节点直接涂黑就可以了。

当树中已有节点，我们就将新节点涂红，并且底下挂两个黑空节点。



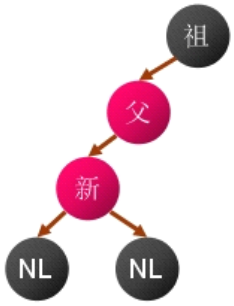
1.1 新节点的父亲为黑色

这种情况最简单，只接插入将新节点可以了，不会出现红色警戒。



1.2 新节点的父亲为红色

这种情况出现红色警戒，并且通过红色的父亲，我们可以推出一定有一个黑色的父，并且父亲不可能为树根(树根必须为黑)。



这种情况需要修复。

1.2.1 新节点的叔叔是红色。（红叔）

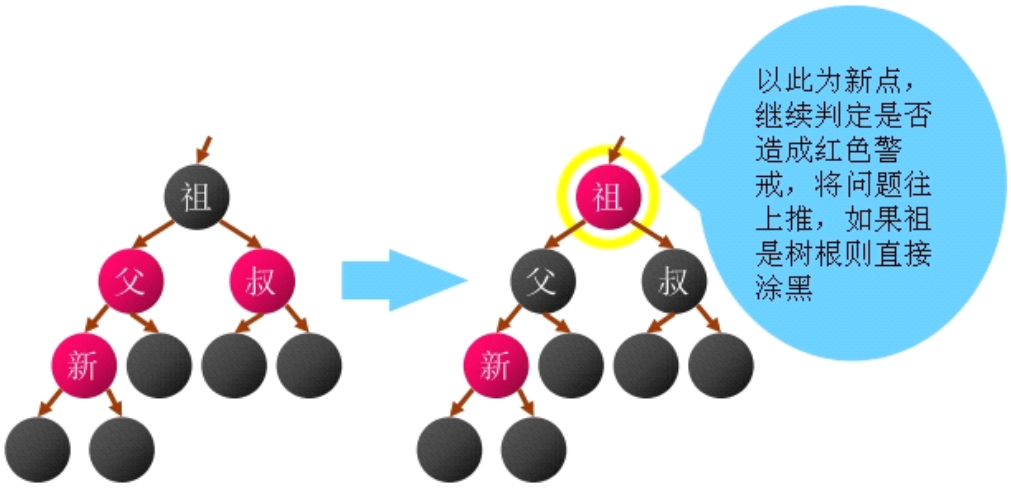


图 1.2.1-1

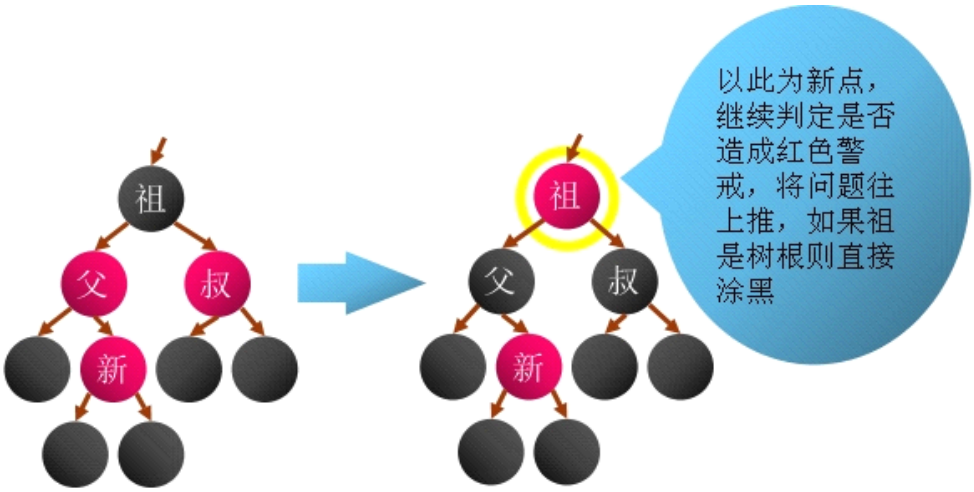


图 1.2.1-2

注解：在这种情况下，我们可以通过这样的方式来修复红黑树的性质。因为遇到红色警戒所以我们首先可以想到的就是将父亲变成黑色，但这样祖父的左子树的黑高就增加了，为了达到祖父的平衡，我们红叔变成黑色，这样祖父就平衡了。但是整个祖父这颗树的高度增高了，所以再此将祖父的颜色变成红色来保持祖父这颗树的高度不变。因为祖父变成了红色，因为往上遍历。

方法：父=>黑；叔=>黑；祖=>红；往上遍历；

1.2.2 新节点的叔叔是黑色（黑叔）

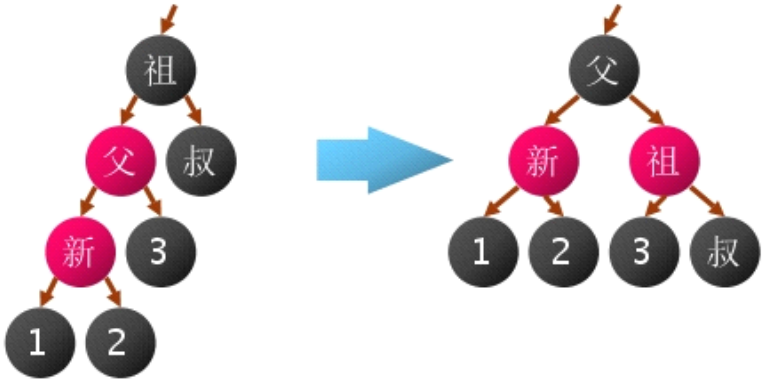


图 1.2.2-1

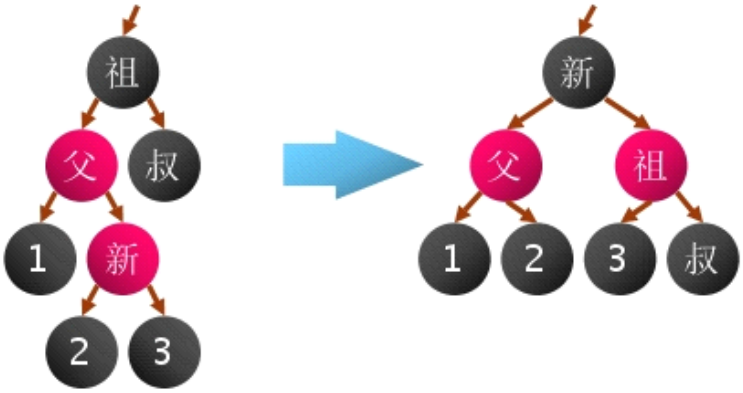


图 1.2.2-2

注解：首先可以说明的是，这种情况下我们都可以通过改变颜色和旋转的方式到达平衡，并且不再出现红色警戒，因为无需往上遍历。图 1.2.2-1：首先我们将红父变成黑色，祖父变成红色，然后对祖父进行右旋转。这样我们可以看到，整颗树的黑高不变，并且这颗树的左右子树也达到平衡。新的树根为黑色。因此无需往上遍历 。

方法：图 1.2.2-1　父=>黑；祖=>红；祖父右旋转；

图 1.2.2-2　新=>黑；祖=>红；父左旋转；祖右旋转；

插入我们就算做完了，与之对称的右边插入的情况完全一样，请自己下来分析；

二.　删除

删除是比较经典但也是最困难的一件事了，所以我们必须一步一步地理解。为了中途思想不混乱，请始终记住一点，下面我们删除的节点都已经表示的是实际要删除的后继节点了。因此可以得出一下结论。

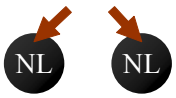
首先，可以肯定的是我们要删除的节点要么是红色，要么是黑色。

其次，既然我们要删除的结点是后继节点，那么要删除的节点的左子树一定为空。所以当删除的节点为黑色时只剩下两种情况。

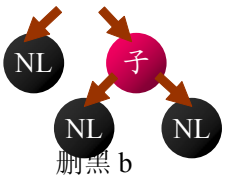
最后，如果删除的节点为红色，那么它必为叶子节点。(*自己好好想象为什么*)。

请在看下面的内容前先牢记上面的结论，这样更加方便让你理解下面的知识。

a：当删除的节点为黑色时

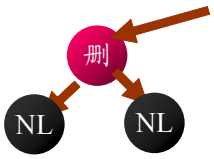


删黑 a



删黑 b

b：当删除的节点为红色时

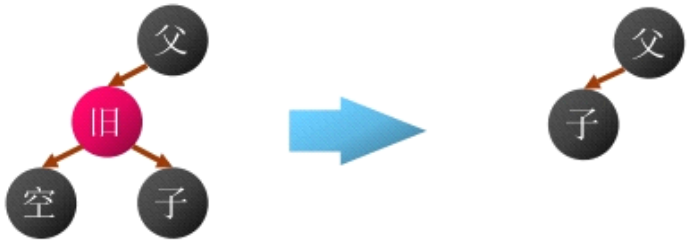


上面的几附图都是很简单的。因为你可以将空节点 

去掉不看。所就形成了要删除的节点要么有一个右孩子，要么为叶子节点。

下面我们就开始真正的删除操作了。

2.1 删除红色节点



注解：这种情况是最简单的，因为根据上面的结论我们可以推出子节点 

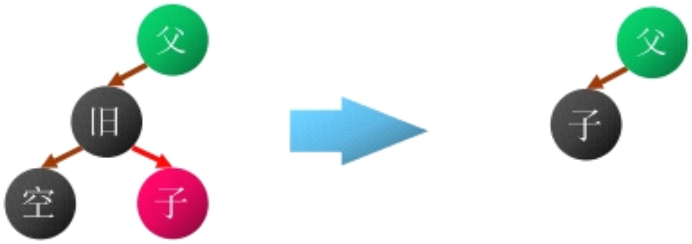
一定为空，也就是说删除的红色节点为叶子节点。只需将这个旧节点的右孩子付给父亲的左孩子就可以了，高度不变。

方法：*略*

2.2 删除黑色节点

遇到黑色节点情况就将变得复杂起来，因此我们一定要慢慢来，仔细分析。

2.2.1 当删除的节点有一个红色子孩子

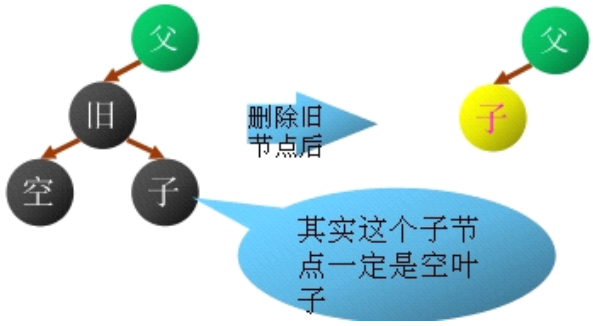


注解：这种情况其实就是上面我们分析的三种情况之一，如图"删黑 b"。这种情况是非常简单的，只需将旧节点的右孩子取代旧节点，并将子孩子的颜色变为黑色，树平衡；

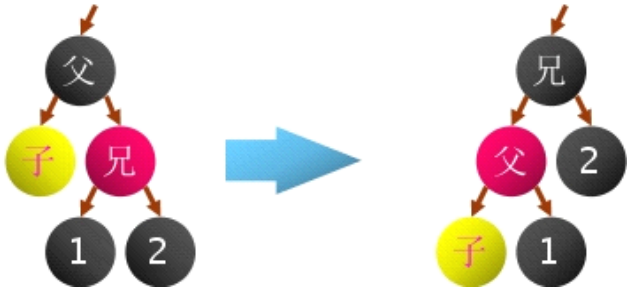
方法：子取代旧；子=>黑；

2.2.2 当删除的节点无左右孩子

这种情况其实就是上面我们分析的三种情况之一，如图"删黑 a"。我们推出子节点 子 一定为空，请务必记住这点，不然在后面你将很容易被混淆，父亲的颜色我们标记为绿色，是表示，父亲的颜色可为红或黑。黄色的子节点 子 其实就是一个黑空节点，这样方便后面分析。



a 红兄

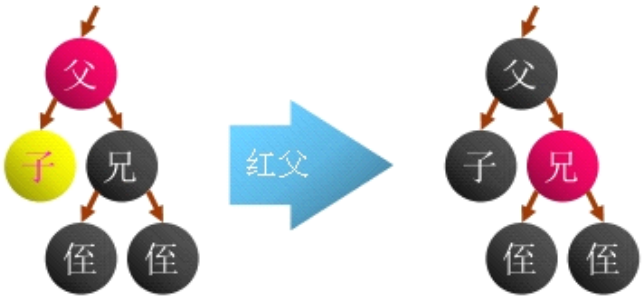


注解：当我们删除的节点拥有一个红色的兄弟时，这种情况还相对比较简单，因为兄弟为黑色，我们可以推出父亲一定为黑色。因为父节点的左树高度减一，所以我们可以通过左旋转父节点，来将节点 1 旋转到左边来恢复左边的高度。然后将父变成红，兄变成黑。这样整颗树的高度不变，父节点也平衡记住 子 子节点为空所以可以删除看，这样便于理解。其实我们也可以推出 1，2 为空，但这里没有必要。
方法：父=>红；兄=>黑；左旋转父节点；

b 黑兄

遇到黑兄就又麻烦了，我们又必须引入要删除的节点的侄子了。所以我们这里再次细分为 b1: 黑兄 双黑侄 b2: 黑兄 左黑侄右红侄 b3: 黑兄 右黑侄左红侄。可能你会问 b4 呢，双红侄的情况呢？其实双红侄的情况同属于 b2,b3 的情况，所以可以默认用 b2,b3 其中一种情况来处理就可以了。

现在我们开始了
b1 黑兄 双黑侄



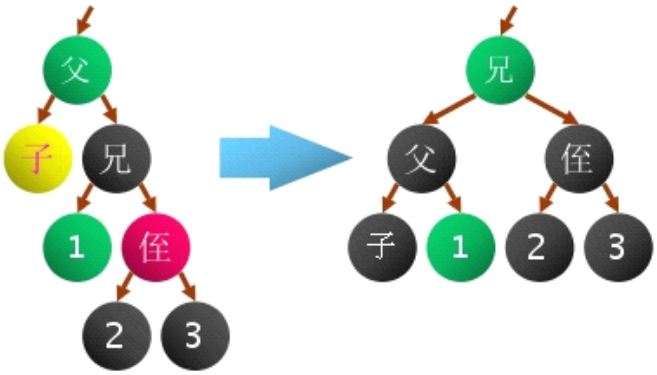
注解：我们首先可以肯定的是父节点的左子树高度减一了，所以我们必须想方设法来弥补这个缺陷。如果我们把父节点的右子树高度也减一（兄变成红色）那么父节点这颗树就可以保持平衡了，但整颗树的高度减一，所以我们可以判断，如果父亲是红色，那么我们可以通过把父亲变成黑色来弥补这一缺陷。但如果父亲是黑色呢，既然遇到到黑色了我们就只好往上遍历了。

方法：兄=>红；子=黑；红父=>黑;

往上遍历(黑父);

补充：其实子节点就是空节点，没有必要变。就算遍历上去，父节点也为黑

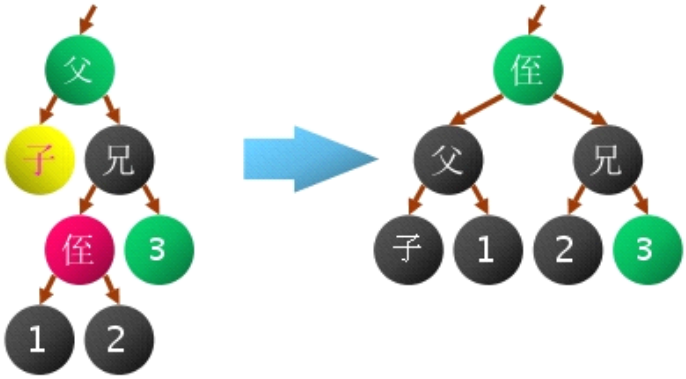
b2 黑兄 左黑侄右红侄



注解：绿色表示，颜色不影响修复的方式。依然我们可以肯定父节点的左子树高度减一，所以我们可以通过将父节点旋转下来并且变为黑色来弥补，但由于兄弟被旋转上去了，又导致右子树高度减一，但我们这有一个红侄，所以可以通过红侄变成黑色来弥补。父亲所在位置的颜色不变；(子=>*子为空*)

方法：兄=>父色；父=>黑；侄=>黑；(*子=>黑*)；左旋转父节点；

b3 黑兄 左红侄右黑侄



注解：绿色表示，颜色不影响修复的方式。依然我们可以肯定父节点的左子树高度

减一,同样我们通过父节点左旋转到左子树来弥补这一缺陷。但如果直接旋转的话，我们可以推出左右子树将不平衡(黑父)，或者两个红色节点相遇(红父)；这样将会更加的复杂，甚至不可行。因此，我们考虑首先将兄弟所在子树右旋转，然后再左旋转父子树。这样我们就可以得到旋转后的树。然后通过修改颜色来使其达到平衡，且高度不变。

方法：侄=>父色；父=>黑;右旋转兄；左旋转父；

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；

程序样列：

```
switch(uncle->color){
case RED:   /* 参考图1.2.1-2 还有一种未画出的情况 这种情况部分叔与父的位置关系 */
    (*sub_root_p)->color = BLACK;          /* 父=>黑 */
    uncle->color = BLACK;                  /* 叔=>黑 */
    grand->color = RED;                    /* 祖=>红 这种情况会往上遍历 */
    break;
case BLACK: /* 参考图1.2.2-2 */
    if(grand->right == uncle){/* 叔在右边 */
        (*sub_root_p)->right->color = BLACK;/* 新=>黑 */
        grand->color = RED;                 /* 祖=>红 */

        rotate_left(sub_root_p);          /* 父左旋转 */

        if(grand->parent == BNULL)          /* 祖父右旋转 */
            rotate_right(root_p);
        else if(grand->parent->left == grand)
            rotate_right(&(grand->parent->left));
        else
            rotate_right(&(grand->parent->right));
    }else{/* 叔在左边，未画出 */
        (*sub_root_p)->color = BLACK;        /* 父=>黑 */
        grand->color = RED;                  /* 祖=>红 */

        if(grand->parent == BNULL)           /* 祖父左旋转 */
            rotate_left(root_p);
        else if(grand->parent->left == grand)
            rotate_left(&(grand->parent->left));
        else
            rotate_left(&(grand->parent->right));
    }
    break;
}
break;
```

声明：

此文档为《C 语言常用数据结构源代码全注解-红黑树源码》所写，此文档遵循 GNU FDL 协议，你可以修改重发，但因保留原始版权信息；

此文档版本 V1.0 为初次发布，所以难面有错误的地方，如果你发现任何错误或缺点，非常感谢你发信息联系我。此文档仅供学习使用。

参考文献：

《百度百科:红黑树》 地址：http://baike.baidu.com/view/133754.htm
《C#与数据结构－树论—红黑树》
地址：http://tech.ddvip.com/2008-12/1229486123100653_7.html
《資料結構與演算法(上)》 作者：呂學一
《資料結構與演算法(上)》 红黑树大部分图片来源

版权信息：

日期：2009-12-4