红黑树(red-black tree)算法，附 AVL 树的比较

linux内核中的用户态地址空间管理使用了红黑树(red-black tree)这种数据结构，我想一定有许多人在这种数据结构上感到困惑，我也曾经为此查阅了许多资料以便了解红黑树的原理。最近我在一个外国网站上看到一篇讲解红黑树的文章，觉得相当不错，不敢独享，于是翻译成中文供所有内核版的弟兄们参考。由于本人水平有限，难免有出错之处，欢迎大家指正。

原文网址：http://sage.mc.yu.edu/kbeen/teaching/algorithms/resources/red-black-tree.html

加两个链结地址:
红黑树的实地使用
http://www.linuxforum.net/forum/showthreaded.php?Cat=&Board=program&Number=556347&page=0&view=collapsed&sb=5&o=31&fpart=&vc=
Splay树的介绍
http://www.linuxforum.net/forum/showflat.php?Cat=&Board=linuxK&Number=609842&page=&view=&sb=&o=&vc=1

红黑树的定义

正如在CLRS中定义的那样（译者： CLRS指的是一本著名的算法书Introduction to Algorithms，中文名应该叫算法导论，CLRS是该书作者Cormen, Leiserson, Rivest and Stein的首字母缩写），一棵红黑树是指一棵满足下述性质的二叉搜索树（BST, binary search tree）：

1. 每个结点或者为黑色或者为红色。
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，那么它的两个子节点都是黑色的（也就是说，不能有两个相邻的红色结点）。
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数量必须相同。

数据项只能存储在内部结点中（internal node）。我们所指的"叶结点"在其父结点中可能仅仅用一个NULL指针表示，但是将它也看作一个实际的结点有助于描述红黑树的插入与删除算法，叶结点一律为黑色。

定理：一棵拥有n个内部结点的红黑树的树高h<=2log(n+1)

(译者：我认为原文中的有关上述定理的证明是错误的，下面的证明方法是参考CLRS中的证明写出的。)

证明：首先定义一颗红黑树的黑高度Bh为：从这颗红黑树的根结点（但不包括这个根结点）到叶结点的路径上包含的黑色结点（注意，包括叶结点）数量。另外规定叶结点的黑高度为 0。
下面我们首先证明一颗有n个内部结点的红黑树满足n>=2^Bh-1。这可以用数学归纳法证明，施归纳于树高h。当h=0 时，这相当于是一个叶结点，黑高度Bh为 0，而内部结点数量n为 0，此时 0>=2^0-1 成立。假设树高h<=t时，n>=2^Bh-1 成立，我们记一颗树高为t+1 的红黑树的根结点的左子树的内部结点数量为nl，右子树的内部结点数量为nr，记这两颗子树的黑高度为Bh'（注意这两颗子树的黑高度必然

一样），显然这两颗子树的树高<=t，于是有nl>=2^Bh'-1 以及nr>=2^Bh'-1，将这两个不等式相加有nl+nr>=2^(Bh'+1)-2，将该不等式左右加 1，得到n>=2^(Bh'+1)-1，很显然Bh'+1>=Bh，于是前面的不等式可以变为n>=2^Bh-1，这样就证明了一颗有n个内部结点的红黑树满足n>=2^Bh-1。
下面我们完成剩余部分的证明，记红黑树树高为h。我们先证明Bh>=h/2。在任何一条从根结点到叶结点的路径上（不包括根结点，但包括叶结点），假设其结点数量为m，注意其包含的黑色结点数量即为Bh。当m为偶数时，根据性质 5 可以看出每一对儿相邻的结点至多有一个红色结点，所以有Bh>=m/2；而当m为奇数时，这条路径上除去叶结点后有偶数个结点，于是这些结点中的黑色结点数B'满足B'>=(m-1)/2，将该不等式前后加 1 得出Bh>=(m+1)/2，可以进一步得出Bh>m/2，综合m为偶数的情况可以得出Bh>=m/2，而m在最大的情况下等于树高h，因此可以证明Bh>=h/2。将Bh>=h/2 代入n>=2^Bh-1，最终得到h<=2log(n+1)。证明完毕。

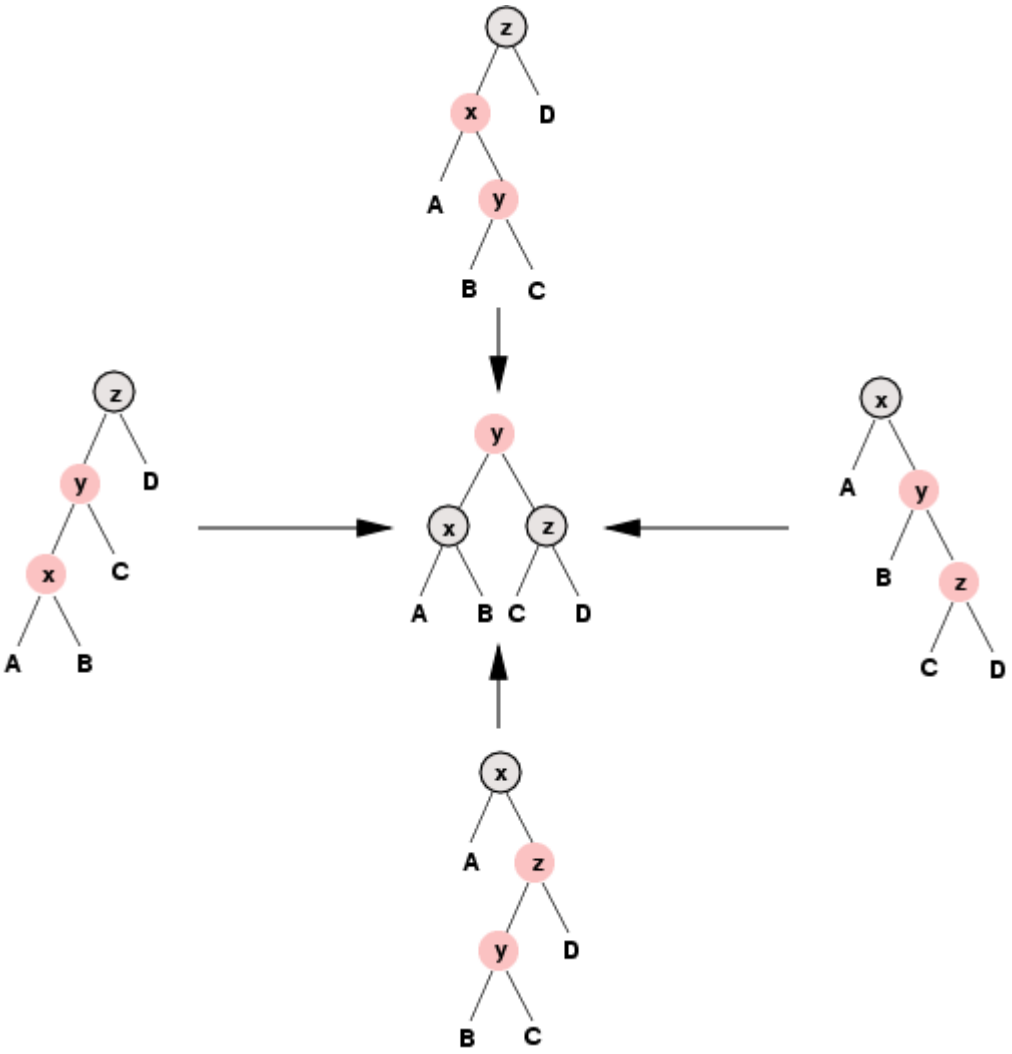
本文余下的内容将阐释如何在不破坏红黑树性质的前提下进行结点的插入与删除，以及为什么插入与删除的处理次数与树高是成比例的，或者说是O(log n)。

Okasaki插入方法

首先用与二叉搜索树一样的方法将一个结点插入到红黑树中，并且颜色为红色。(这个新结点的子结点将是叶结点，根据定义，这些叶结点是黑色的。)此时，我们将或者破坏了性质 2（根结点为黑色）或者破坏了性质 4（不能有两个相邻的红色结点）。

如果新插入的结点是根结点的话（这意味着在插入前该红黑树是空的），我们仅仅将这个结点的颜色改为黑色，插入操作就完成了。

如果性质 4 遭到了破坏，这一定是由于新插入结点的父结点也是红色造成的。由于红黑树的根结点必须是黑色的，因此新插入的结点一定会存在一个祖父结点，并且根据性质 4 这个祖父结点必然是黑色的。此时，由新插入结点的祖父结点为根的子树的结构一共有四种可能性（译者，前面这句话我没有看明白原文，我是用我的理解写出来的，如果有误请指正。），如下面的图解所示。在Okasaki插入方法中，每一种可能出现的子树都被转换为图解正中间的那种子树形式。



（A,B,C与D表示任意的子树。我们曾经说过新插入结点的子结点一定是叶结点，但很快我们就会看到上面的图解适用于更普遍的情况）。

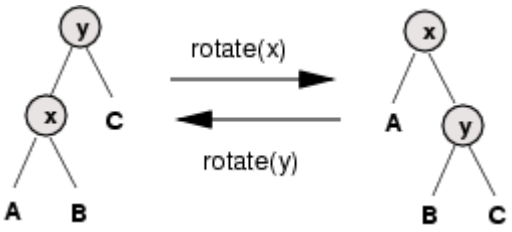
首先，请注意在变换的过程中<AxByCzD>的顺序保持不变。

另外，注意该变换不会改变从这颗子树的父结点到这颗子树中任何一个叶结点的路径中黑色结点的数量（当然前提是这颗子树有父结点）。我们再一次遇到了这样的情形：即该红黑树只有可能违反性质 2（如果y是根结点）或性质 4（如果y的父结点是红色的），但这次变换带来了一个好处，即我们现在距离红黑树的根结点靠近了两步。我们可以重复这种操作直到：或者y的父结点为黑色，在这种情况下插入操作完成；或者y成为根结点，在此情况下我们将y染为黑色后插入操作完成。（将根结点染为黑色会对每条从根结点到叶结点的路径增加相同数量的黑色结点，因此如果在染色操作之前性质 5 没有遭到破坏那么操作之后也不会。）

上述步骤保持了红黑树的性质，并且所花的时间与树高是成比例的，也即O(log n)。

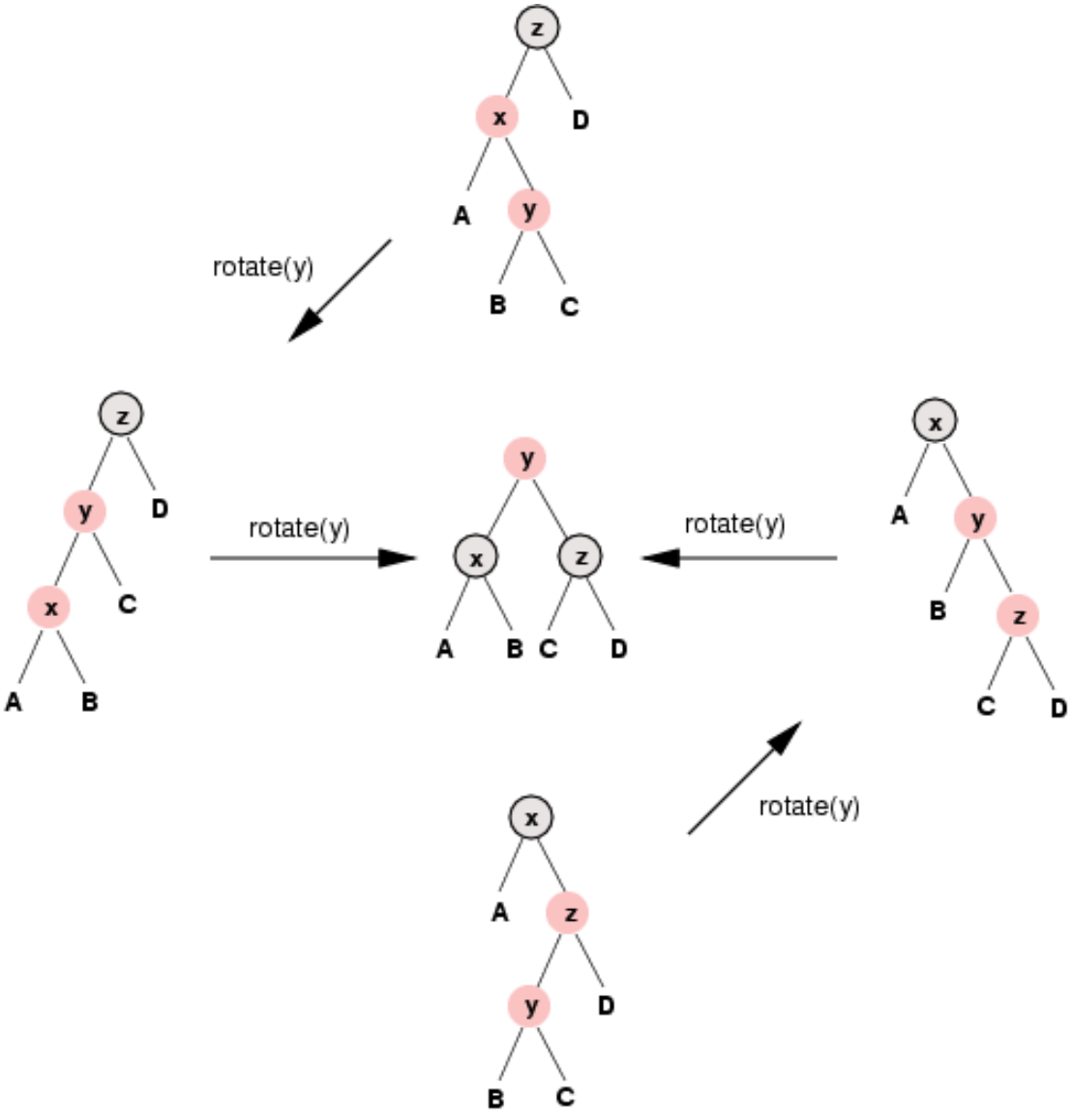
旋转

在红黑树中进行结构调整的操作常常可以用更清晰的术语"旋转"操作来表达，图解如下。



很显然，在旋转操作中<AxByC>的顺序保持不变。因此，如果操作前该树是一颗二叉搜索树，而且结构调整时只使用了旋转操作，那么调整后该树仍然是一颗二叉搜索树。在本文的余下部分，我们将仅仅使用旋转操作对树进行调整，因此我们无须再言明关于如何保持树中元素的正确排序问题。

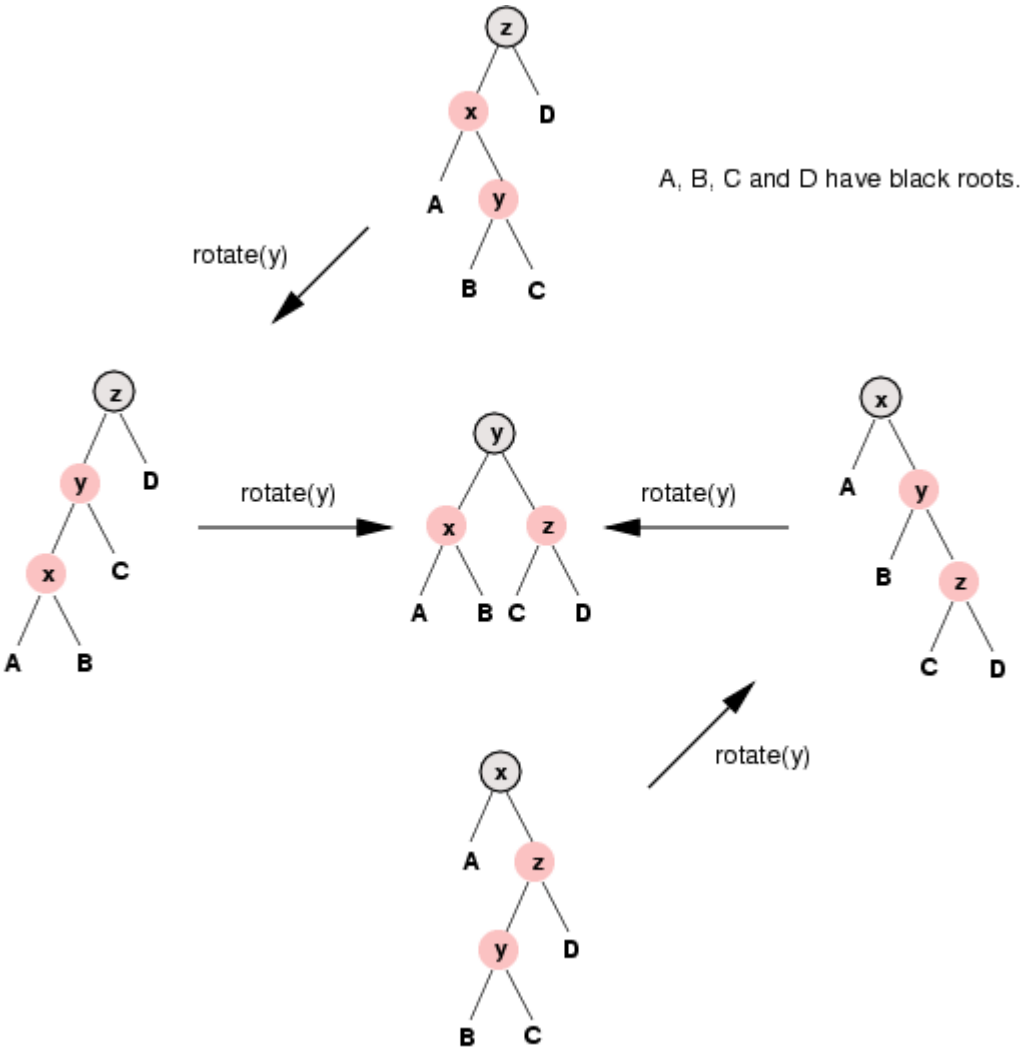
在下面的图解中，Okasaki插入方法中的变换操作被表示为一个或者两个旋转操作。



CLRS插入方法

CLRS中给出了一种比Okasaki插入方法更复杂但效率稍高的插入方法。它的时间复杂度仍然是O(log n)，但在大O中的常数要更小一些。

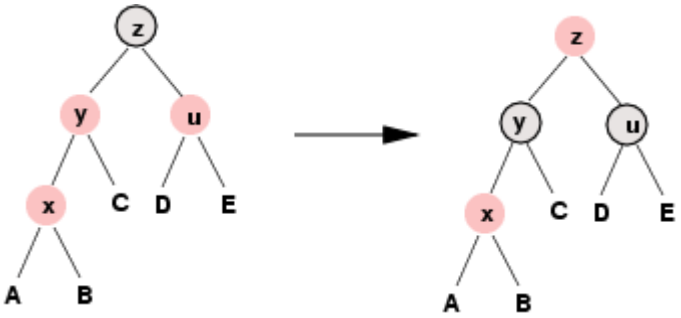
CLRS插入方法与Okasaki插入方法一样都是从标准的二叉搜索树插入操作开始的，并且将这个新插入的结点染为红色，它们的区别在于如何处理遭到破坏的性质 4（不能存在两个相邻的红色结点）。我们要根据下端红色结点的叔叔结点的颜色区分两种情况。（下端红色结点是指在一对儿红色父结点 / 红色子结点中的那个子结点。）让我们先考虑叔叔结点为黑色的情况。根据每个红色结点是其父结点的左子结点还是右子结点，这种情况可以分为四种子情况。下面的图解展示了如何调整红黑树以及如何重新染色。



在这里我们感兴趣的是上面图解中的方法与Okasaki方法的比较。它们有两点不同。第一点是关于如何对最终的子树（图解中间的那个子树）进行染色的。在Okasaki方法中，这颗子树的根结点y被染成红色而它的子结点被染成黑色，然而在CLRS方法中y被染成了黑色而它的子结点被染成了红色。将y染成黑色意味着红黑树性质 4（不能存在两个相邻的红色结点）不会在y这一点遭到破坏，因此对树的调整不需要向根结点的方向继续进行下去。在此情况下，CLRS插入方法最多需要进行两次旋转操作即可完成插入。

第二点不同是在这种情况下CLRS方法必须满足一个先决条件，即下端红色结点的叔叔结点必须是黑色的。在上面的图解中我们可以很清楚地看出，如果那个叔叔结点（即子树A或者D的根结点）是红色的，那么最终的树中将存在两个相邻的红色结点，因此这种方法不能适用于叔叔结点为红色的情况。

下面我们考虑下端红色结点的叔叔结点为红色的情况。在这种情况下我们将上端红色结点和它的兄弟结点（即下端红色结点的叔叔结点）染为黑色并且将它们的父结点染为红色。树的结构并没有进行调整。这时根据下端红色结点是其父结点的左子结点还是右子结点以及上端红色结点是其父结点的左子结点还是右子结点可以分出四种情况，但是这四种情况从本质上来说都是相同的。下面图解只描述了一种情况：



很容易看出，在这种操作的过程中从树的根结点到叶结点的路径中的黑色结点数量没有发生变化。在此操作之后，红黑数的性质只有可能在该子树的根结点同时也是整个树的根结点或者该子树的父结点是红色的情况下才会遭到破坏。换句话说，我们又将开始重复上述操作，但我们距离树的根结点又靠近了两步。照这样不断重复该步骤直到：或者(i)z的父结点为黑色，此时插入操作结束；(ii)z成为根结点，我们将它染为黑色之后插入操作结束；或者(iii)我们遇到了下端红色结点的叔叔结点为黑色的情况，这时我们只要做一或两次旋转操作即可完成插入。在最坏的情况下，我们必须对新插入的结点到根结点的路径上的每个结点进行染色操作，此时需要的操作数为O(log n)。

删除

为了从红黑树中删除一个结点，我们将从一颗标准二叉搜索树的删除操作开始（参见CLRS，第 12 章）。我们回顾一下标准二叉搜索树的删除操作的三种情况：

1. 要删除的结点没有子结点。在这种情况下，我们直接将它删除就可以了。如果这个结点是根结点，那么这颗树将成为空树；否则，将它的父结点中相应的子结点指针赋值为NULL。
2. 要删除的结点有一个子结点。与上面一样，直接将它删除。如果它是根结点，那么它的子结点变为根结点；否则，将它的父结点中相应的子结点指针赋值为被删除结点的子结点的指针。
3. 要删除的结点有两个子结点。在这种情况下，我们先找到这个结点的后继结点（successor），也就是它的右子树中最小的那个结点。然后我们将这两个结点中的数据元素互换，之后删除这个后继结点。由于这个后继结点不可能有左子结点，因此删除该后继结点的操作必然会落入上面两种情况之一。

注意，在树中被删除的结点并不一定是那个最初包含要删除的数据项的那个结点。但出于重建红黑树性质的目的，我们只关心最终被删除的那个结点。我们称这个结点为v，并称它的父结点为p(v)。

v的子结点中至少有一个为叶结点。如果v有一个非叶子结点，那么v在这颗树中的位置将被这个子结点取代；否则，它的位置将被一个叶结点取代。我们用u来表示二叉搜索树删除操作后在树中取代了v的位置的

那个结点。如果u是叶结点，那么我们可以确定它是黑色的。

如果v是红色的，那么删除操作就完成了－－－因为这种删除不会破坏红黑树的任何性质。所以，我们下面假定v是黑色的。删除了v之后，从根结点到v的所有子孙叶结点的路径将会比树中其它的从根结点到叶结点的路径拥有更少的黑色结点，这会破坏红黑树的性质 5。另外，如果p(v)与u都是红色的，那么性质 4 也会遭到破坏。但实际上我们解决性质 5 遭到破坏的方案在不用作任何额外工作的情况下就可以同时解决性质 4 遭到破坏的问题，所以从现在开始我们将集中精力考虑性质 5 的问题。

让我们在头脑中给u打上一个黑色记号（black token）。这个记号表示从根结点到这个带记号结点的所有子孙叶结点的路径上都缺少一个黑色结点（在一开始，这是由于v被删除了）。我们会将这个记号一直朝树的顶部移动直到性质 5 重新恢复。在下面的图解中用一个黑色的方块表示这个记号。如果带有这个记号的结点是黑色的，那么我们称之为双黑色结点（doubly black node）。

注意这个记号只是一个概念上的东西，在树的数据结构中并不存在物理实现。

我们要区分四种不同的情况。

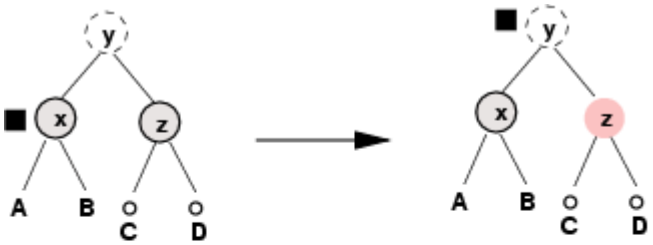
A. 如果带记号的结点是红色的或者它是树的根结点（或两者皆是），只要将它染为黑色就可以完成删除操作。注意，这样就会恢复红黑树的性质 4（不能存在两个相邻的红色结点）。而且，性质 5 也会被恢复，因为这个记号表示从根结点到该结点的所有子孙叶结点的路径需要增加一个黑色结点以便使这些路径与其它的根结点到叶结点路径所包含的黑色结点数量相同。通过将这个红色结点改变为黑色，我们就在这些缺少一个黑色结点的路径上添加了一个黑色结点。

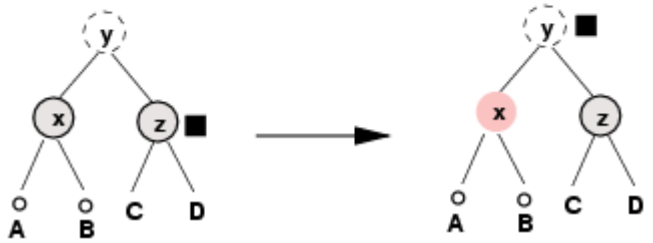
如果带记号的结点是根结点并且为黑色，那么直接将这个标记丢掉就可以了。在这种情况下，树中每条从根结点到叶结点的路径的黑色结点数量都比删除操作前少了一个，并且依旧保持住了性质 5。

在余下的情况里，我们可以假设这个带记号的结点是黑色的，并且不是根结点。

B. 如果这个双黑色结点的兄弟结点以及两个侄子结点都是黑色的，那么我们就将它的兄弟结点染为红色之后将这个记号朝树根的方向移动一步。

下面的图解展示了两种可能出现的子情况。环绕y的虚线表示在此并我们不关心y的颜色，而在A,B,C和D的上面的小圆圈表示这些子树的根结点是黑色的（译者：注意这个双黑色结点必然会有两个非叶结点的侄子结点。这是因为这个双黑色结点的记号表示从根结点到该结点的所有子孙叶结点的路径中的黑色结点数量都比其它的根结点到叶结点路径所包含的黑色结点数量少 1，而该双黑色结点本身就是一个黑色结点，因此从它的兄弟结点到其子孙叶结点的路径上的黑色结点数量必然要大于 1，我们很容易看出如果其兄弟结点的任何一个子结点为叶结点的话这一点是不可能满足的，因此这个双黑色结点的必然会有两个非叶结点的侄子结点）。

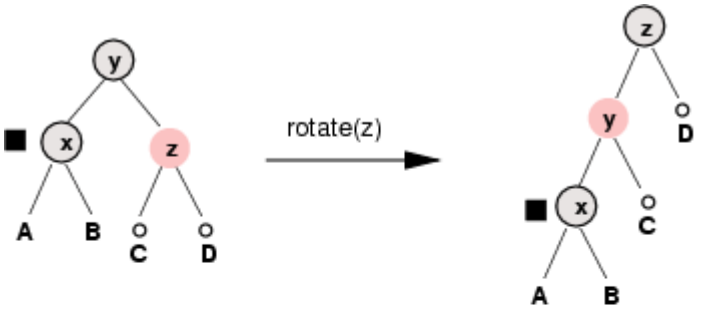


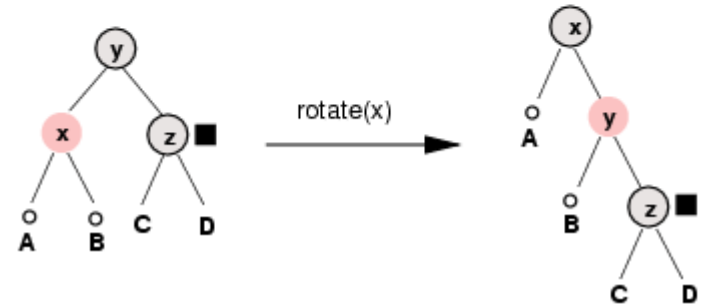


将那个兄弟结点染为红色，就会从所有到该结点的子孙叶结点的路径上去掉一个黑色结点，因此现在这些路径上的黑色结点数量与到双黑色结点的子孙叶结点的路径上的黑色结点数量一致了。我们将这个记号向上移动到y，这表明现在所有到y的子孙叶结点的路径上缺少一个黑色结点。此时问题仍然没有得到解决，但我们又向树根推进了一步。

很显然，只有带记号的结点的两个侄子结点都是黑色时才能进行上述操作，这是因为如果有一个侄子结点是红色的那么该操作会导致出现两个相邻的红色结点。

C. 如果带记号的结点的兄弟结点是红色的，那么我们就进行一次旋转操作并改变结点颜色。下面的图解展示了两种可能出现的情况：



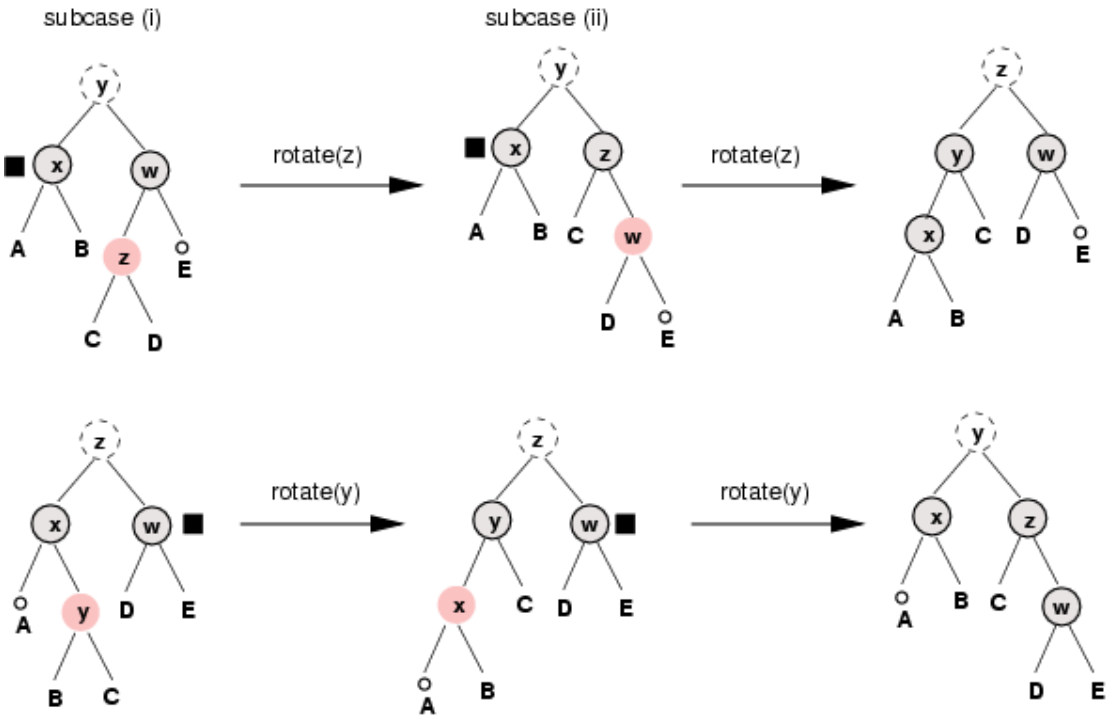


注意上面的操作并不会改变从根结点到任何叶结点路径上的黑色结点数量，并且它确保了在操作之后这个双黑色结点的兄弟结点是黑色的，这使得后续的操作或者属于情况B，或者属于情况D。

由于这个记号比起操作前离树的根结点更远了，所以看起来似乎我们向后倒退了。但请注意现在这个双黑色结点的父结点是红色的了，所以如果下一步操作属于情况B，那么这个记号将会向上移动到那个红色结点，然后我们只要将它染为黑色就完成了。此外，下面将会展示，在情况D下，我们总是能够将这个记号消耗掉从而完成删除操作。因此这种表面上的倒退现象实际上意味着删除操作就快要完成了。

D. 最终，我们遇到了双黑色结点有一个黑色兄弟结点并至少一个侄子结点是红色的情况。我们下面给出一个结点x的近侄子结点（near nephew）的定义：如果x是其父结点的左子结点，那么x的兄弟结点的左子结点为x的近侄子结点，否则x的兄弟结点的右子结点为x的近侄子结点；而另一个侄子结点则为x的远侄子结点（far nephew）。（在下面的图解中可以看出，x的近侄子结点要比它的远侄子结点距离x更近。）

现在我们会遇到两种子情况：(i)双黑色结点的远侄子结点是黑色的，在此情况下它的近侄子结点一定是红色的；(ii)远侄子结点是红色的，在此情况下它的近侄子结点可以为任何颜色。如下面的图解所示，子情况(i)可以通过一次旋转和变色转换为子情况(ii)，而在子情况(ii)下只要通过一次旋转和变色就可以完成删除操作。根据双黑色结点是其父结点的左子结点还是右子结点，下面图解中的两行显示出两种对称的形式。



在这种情况下我们生成了一个额外的黑色结点，记号被丢掉，删除操作完成。从上面图解中很容易看出，所有到带记号结点的子孙叶结点的路径上的黑色结点数量增加了 1，而其它的路径上的黑色结点数量保持不变。很显然，在此刻红黑树的任何性质都没有遭到破坏。

将上面的所有情况综合起来，我们可以看出在最坏的情况下我们必须沿着从叶结点到根结点的路径每次都执行常量次数的操作，因此删除操作的时间复杂度为O(log n)。

附　AVL 树的比较

我还一直沉浸于 2.4 的 AVL 树,殊不知,早已经是过年货了(各位新春愉快!!),2.6 已经使用 Red-Black-Tree,感叹
不是我不明白,世界变得太快,既然是 AVL 树则比较一下:
简介:
AVL 树又称高度平衡的二叉搜索树,是 1962 年由两位俄罗斯的数学家 G.M.Adel'son-Vel,sky 和 E.M.Landis 提出
的.引入二叉树的目的是为了提高二叉树的搜索的效率,减少树的平均搜索长度.为此,就必须每向二叉树插入
一个结点时调整树的结构,使得二叉树搜索保持平衡,从而可能降低树的高度,减少的平均树的搜索长度.

AVL 树的定义:
一棵 AVL 树满足以下的条件:
1>它的左子树和右子树都是 AVL 树
2>左子树和右子树的高度差不能超过 1
从条件 1 可能看出是个递归定义,如 GNU 一样.

性质:
1>一棵 n 个结点的 AVL 树的其高度保持在 0(log2(n)),不会超过 3/2log2(n+1)
2>一棵 n 个结点的 AVL 树的平均搜索长度保持在 0(log2(n)).
3>一棵 n 个结点的 AVL 树删除一个结点做平衡化旋转所需要的时间为 0(log2(n)).

从 1 这点来看红黑树是牺牲了严格的高度平衡的优越条件为代价红黑树能够以 O(log2 n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。红黑树的算法时间复杂度和 AVL 相同，但统计性能比 AVL 树更高.
看看人家怎么评价的:
AVL trees are actually easier to implement than RB trees because there are fewer cases. And AVL trees require O(1) rotations on an insertion, whereas red-black trees require O(lg n). In practice, the speed of AVL trees versus red-black trees will depend on the data that you're inserting. If your data is well distributed, so that an unbalanced binary tree would generally be acceptable (i.e. roughly in random order), but you want to handle bad cases anyway, then red-black trees will be faster because they do less unnecessary rebalancing of already acceptable data.On the other hand, if a pathological insertion order (e.g. increasing order of key) is common, then AVL trees will be faster, because the stricter balancing rule will reduce the tree's height.
Splay trees might be even faster than either RB or AVL trees,depending on your data access distribution. And if you can use a hash instead of a tree, then that'll be fastest of all.

试着翻译一下:
<pre>
由于 AVL 树种类较少所以比红黑树实际上更容易实现.而且 ALV 树在旋转插入所需要的复杂度为 0(1),而红

黑树则需要的复杂度为 0(lgn).
实际上插入 AVL 树和红黑树的速度取决于你所插入的数据.如果你的数据分布较好,则比较宜于采用 AVL 树(例如随机产生系列数),但是如果你想处理比较杂乱的情况,则红黑树是比较快的,因为红黑树对已经处理好的数据重新平衡减少了不心要的操作.另外一方面,如果是一种非寻常的插入系列比较常见(比如,插入密钥系列),则 AVL 树比较快,因为它的严格的平衡规则将会减少树的高度.
Splay 树可能比红黑树和 AVL 树还要快这也取决于你所访问的数据分布,如果你用哈希表来代替一棵树,则将所以的树还要快.
</pre>
Splay 树是什么树,我不是很清楚,我没有详细的查阅.

感受一下带来的变革

```
//--><pre>
/*
* 翻一下老皇历(2.4)
*/
struct vm_area_struct* find_vma(struct mm_struct* mm,unsigned long addr)
{
struct vm_area_struct* vma = NULL;
if(mm)
{
/*
* check the cache first.
*/
/*
* (Check hit rate is typically around 35%.)
*/
/*
* 首先查找一下最近一次访问的虚地址空间是不否是 CACHE 中
*/
vma = mm->mmap_cache;
if(!(vma && vma->vm_end > addr && vma->vm_start<addr))
{
/*
* miss hit 未命中,继续查找线性表或者是 AVL 树
*/
if(!mm->mmap_val)
{
/*
* go though the liner list
*/
vma = mm->mmap;
while(vma && vma->vm_end <= addr)
{
vma = vam->vma_next;
```

```
}
}
else
{
/*
* Then go though the AVL tree quickly
*/
struct vm_area_struct* tree = mm->mmap_avl;
vam = NULL;
for(;;)
{
if(tree == vm_avl_empty)
{
/*
* 结点为空,失败
*/
break;
}
if(tree->vm_end > addr)
{
vma = tree;
if(tree->vm_start <= addr)
{
/*
* 找到,快速退出循环
*/
break;
}
tree = tree->vm_avl_left;
}
else
{
tree = tree->vm_avl_right;
}
}
}
if(vma)
{
/*
* 查找成功,记在 CACHE 中
*/
mm->mmap_cache = vma;
}
}
```

```
}
return vma;

}

//<--</pre>

/*
* 再贴新年历(2.6)
*/
//--><pre1>
struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
{
struct vm_area_struct *vma = NULL;

if (mm) {
/* Check the cache first. */
/* (Cache hit rate is typically around 35%.) */
/*
* 首先查找一下最近一次访问的虚地址空间是不否是 CACHE 中
*/
vma = mm->mmap_cache;
if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
struct rb_node * rb_node;

/*
* miss hit 未命中,直接查找 red-black tree
*/
rb_node = mm->mm_rb.rb_node;
vma = NULL;

while (rb_node) {
struct vm_area_struct * vma_tmp;

vma_tmp = rb_entry(rb_node,
struct vm_area_struct, vm_rb);

if (vma_tmp->vm_end > addr) {
vma = vma_tmp;
if (vma_tmp->vm_start <= addr)
break;
rb_node = rb_node->rb_left;
} else
rb_node = rb_node->rb_right;
```

```
}
/*
*  查找成功,记在 CACHE 中
*/
if (vma)
mm->mmap_cache = vma;
}
}
return vma;
}

//<--</pre1>
```

在这儿只是做了一些小的方面的比较和在内核中的真正的应用,很多的地方没有分析到,还望各位同仁多多指正和拓展.