教你透彻了解红黑树

分类： 25.Red-black tree 24.data structures2010-12-29 18:36 17725人阅读 评论(105) 收藏 举报

教你透彻了解红黑树

作者：July、saturnman 2010年12月29日

本文参考：Google、算法导论、STL 源码剖析、计算机程序设计艺术。

推荐阅读：***Left-Leaning Red-Black Trees***, Dagstuhl Workshop on Data Structures, Wadern, Germany, February, 2008.
直接下载：http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf

红黑树系列，六篇文章于今日已经完成：

一、红黑树的介绍

先来看下算法导论对 R-B Tree 的介绍：
红黑树，一种二叉查找树，但在每个结点上增加一个存储位表示结点的颜色，可以是 Red 或 Black。
通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其
他路径长出俩倍，因而是接近平衡的。

前面说了，红黑树，是一种二叉查找树，既然是二叉查找树，那么它必满足二叉查找树的一般性质。
下面，在具体介绍红黑树之前，咱们先来了解下 二叉查找树的一般性质：

1.在一棵二叉查找树上，执行查找、插入、删除等操作，的时间复杂度为O（lgn）。

因为，一棵由n个结点，随机构造的二叉查找树的高度为lgn，所以顺理成章，一般操作的执行时间为O（lgn）。

//至于n个结点的二叉树高度为lgn的证明，可参考算法导论 第12章 二叉查找树 第12.4节。

2.但若是一棵具有n个结点的线性链，则此些操作最坏情况运行时间为O（n）。

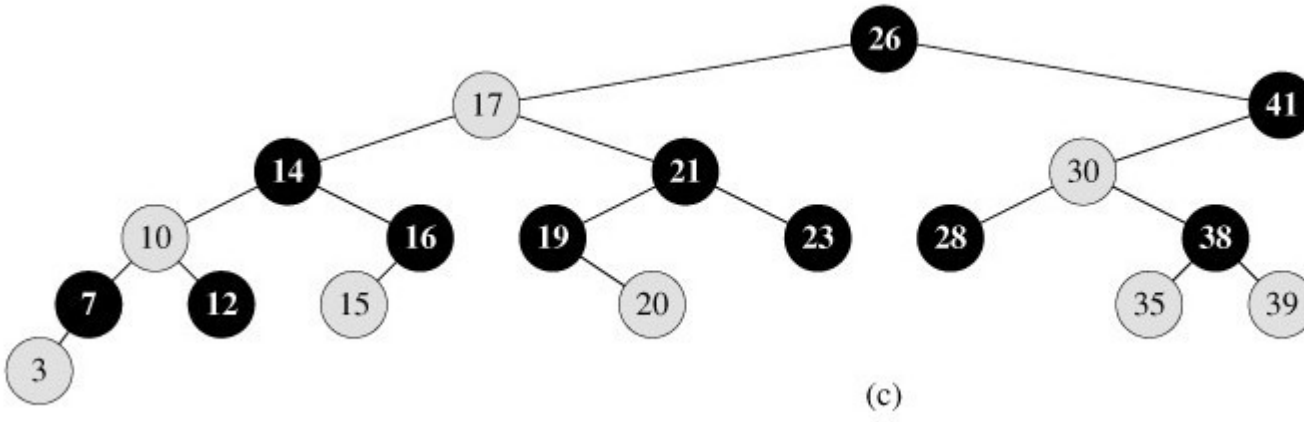
而**红黑树，能保证在最坏情况下，基本的动态几何操作的时间均为O（lgn）。**

ok，我们知道，红黑树上每个结点内含五个域，color，key，left，right，p。如果相应的指针域没有，则设为NIL。

一般的，红黑树，满足以下性质，即只有满足以下全部性质的树，我们才称之为红黑树：

1）每个结点要么是红的，要么是黑的。
2）根结点是黑的。
3）每个叶结点，即空结点（NIL）是黑的。
4）如果一个结点是红的，那么它的俩个儿子都是黑的。
5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

下图所示，即是一颗红黑树：



此图忽略了叶子和根部的父结点。

二、树的旋转知识

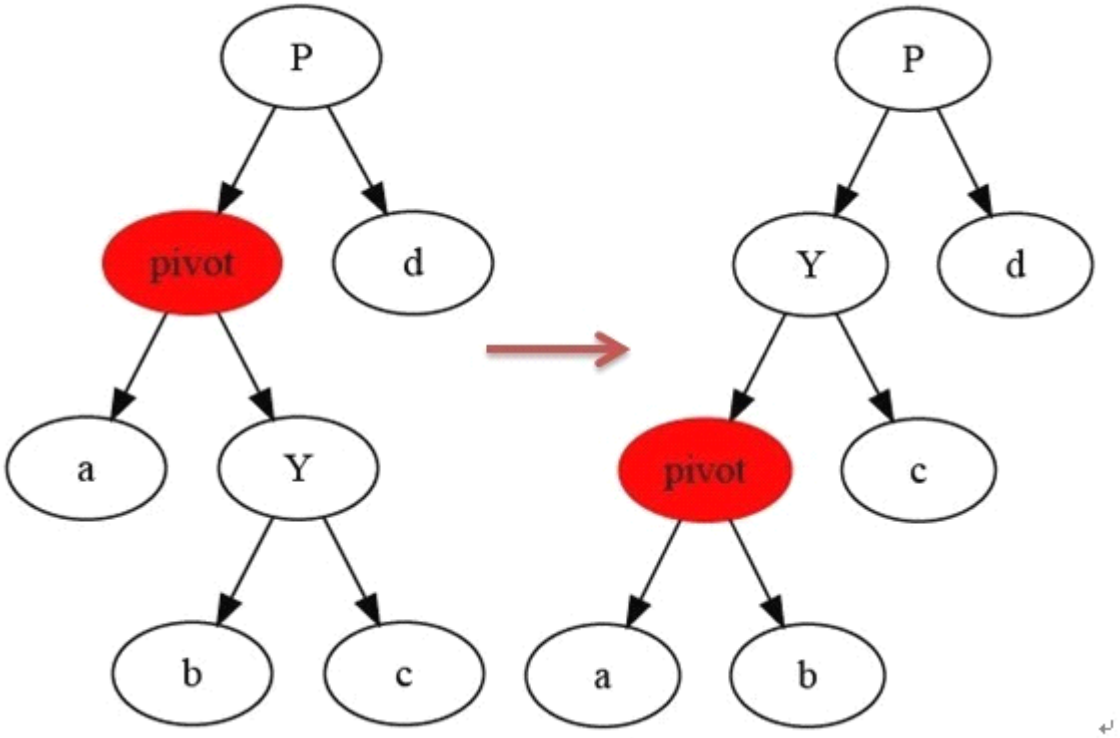
当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。

为了保持红黑树的性质，我们可以通过对树进行旋转，即修改树种某些结点的颜色及指针结构，以达到对红黑树进行

插入、删除结点等操作时，红黑树依然能保持它特有的性质（如上文所述的，五点性质）。

树的旋转，分为左旋和右旋，以下借助图来做形象的解释和介绍：

1.左旋



如上图所示：

当在某个结点 pivot 上，做左旋操作时，我们假设它的右孩子 y 不是 NIL[T]，pivot 可以为树内任意右孩子而不是 NIL[T]的结点。

左旋以 pivot 到 y 之间的链为“支轴”进行，它使 y 成为该孩子树新的根，而 y 的左孩子 b 则成为 pivot 的右孩子。

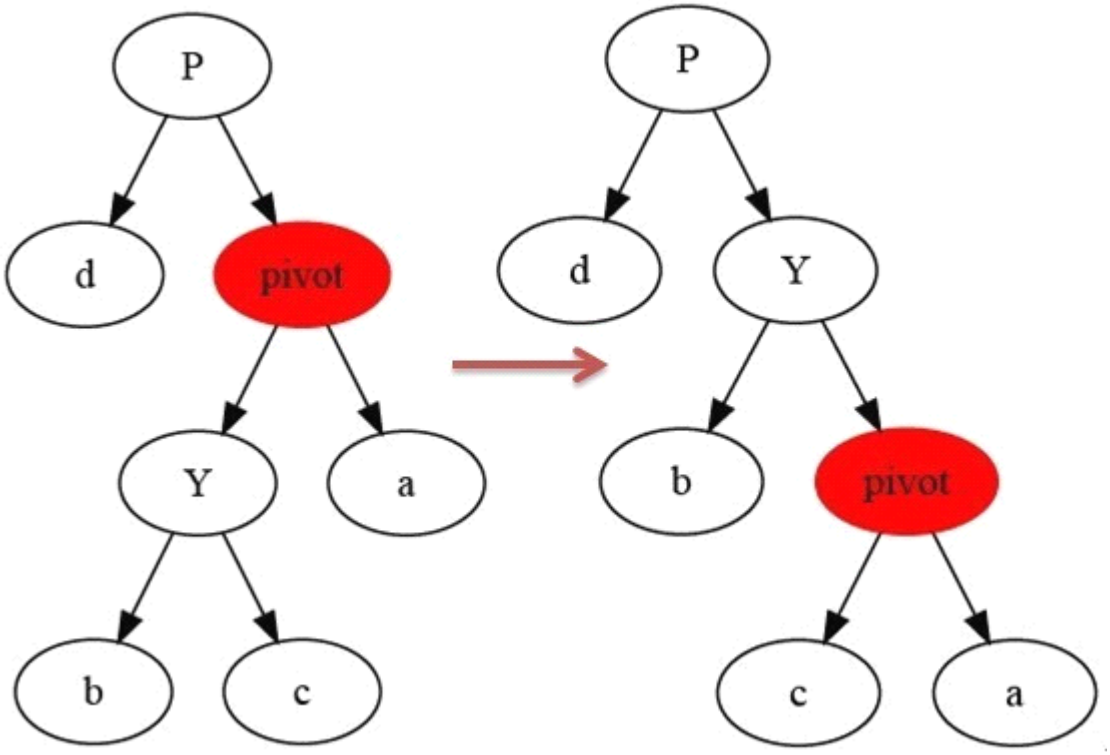
来看算法导论对此操作的算法实现（以 x 代替上述的 pivot）：

```
LEFT-ROTATE(T, x)
1   y ← right[x]     Set y.
2   right[x] ← left[y]              Turn y's left subtree into x's right subtree.

3   p[left[y]] ← x
4   p[y] ← p[x]                     Link x's parent to y.
5   if p[x] = nil[T]
6       then root[T] ← y
7       else if x = left[p[x]]
8                then left[p[x]] ← y
9                else right[p[x]] ← y
10  left[y] ← x                     Put x on y's left.
11  p[x] ← y
```

2.右旋

右旋与左旋差不多，再此不做详细介绍。



对于树的旋转，能保持不变的只有原树的搜索性质，而原树的红黑性质则不能保

持，

在红黑树的数据插入和删除后可利用旋转和颜色重涂来恢复树的红黑性质。

至于有些书如 STL 源码剖析有对双旋的描述，其实双旋只是单旋的两次应用，并无新的内容，

因此这里就不再介绍了，而且左右旋也是相互对称的，只要理解其中一种旋转就可以了。

三、红黑树插入、删除操作的具体实现

三、l、ok，接下来，咱们来具体了解红黑树的插入操作。
向一棵含有 n 个结点的红黑树插入一个新结点的操作可以在 O（lgn）时间内完成。

算法导论：

```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else if key[z] < key[y]
12                then left[y] ← z
13                else right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

咱们来具体分析下，此段代码：
RB-INSERT(T, z)，将 z 插入红黑树 T 之内。

为保证红黑性质在插入操作后依然保持，上述代码调用了一个辅助程序 RB-INSERT-FIXUP

来对结点进行重新着色，并旋转。

14 left[z] ← nil[T]
15 right[z] ← nil[T] //保持正确的树结构
第16行，将 z 着为红色，由于将 z 着为红色可能会违背某一条红黑树的性质，
所以，在第17行，调用 RB-INSERT-FIXUP（T,z）来保持红黑树的性质。

RB-INSERT-FIXUP(T, z)，如下所示：

```
 1 while color[p[z]] = RED
 2       do if p[z] = left[p[p[z]]]
 3             then y ← right[p[p[z]]]
 4                  if color[y] = RED
 5                     then color[p[z]] ← BLACK                    Case 1
 6                          color[y] ← BLACK                      Case 1
 7                          color[p[p[z]]] ← RED                  Case 1
 8                          z ← p[p[z]]                           Case 1
 9                     else if z = right[p[z]]
10                               then z ← p[z]                    Case 2
11                                    LEFT-ROTATE(T, z)           Case 2
12                               color[p[z]] ← BLACK              Case 3
13                               color[p[p[z]]] ← RED             Case 3
14                               RIGHT-ROTATE(T, p[p[z]])         Case 3
15             else (same as then clause
                          with "right" and "left" exchanged)
16 color[root[T]] ← BLACK
```

ok，参考一网友的言论，用自己的语言，再来具体解剖下上述俩段代码。
为了保证阐述清晰，我再写下红黑树的5个性质：

1）每个结点要么是红的，要么是黑的。
2）根结点是黑的。
3）每个叶结点，即空结点（NIL）是黑的。
4）如果一个结点是红的，那么它的俩个儿子都是黑的。
5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

在对红黑树进行插入操作时，我们一般总是插入红色的结点，因为这样可以在插入过程中尽量避免对树的调整。
那么，我们插入一个结点后，可能会使原树的哪些性质改变列?
由于，我们是按照二叉树的方式进行插入，因此元素的搜索性质不会改变。

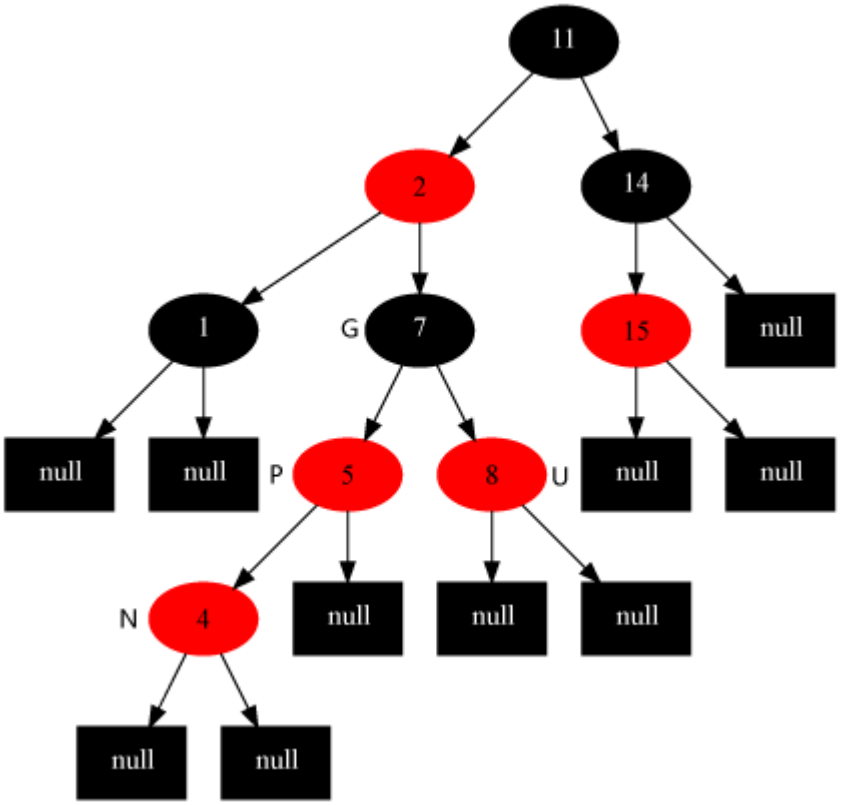
如果插入的结点是根结点，性质2会被破坏，如果插入结点的父结点是红色，则会破坏性质4。
因此，总而言之，插入一个红色结点只会破坏性质2或性质4。
我们的回复策略很简单，
其一、把出现违背红黑树性质的结点向上移，如果能移到根结点，那么很容易就能通过直接修改根结点来恢复红黑树的性质。直接通过修改根结点来恢复红黑树应满足的性质。
其二、穷举所有的可能性，之后把能归于同一类方法处理的归为同一类，不能直接处理的化归到下面的几种情况，

//注：以下情况3、4、5与上述算法导论上的代码 RB-INSERT-FIXUP(T, z)，相对应：

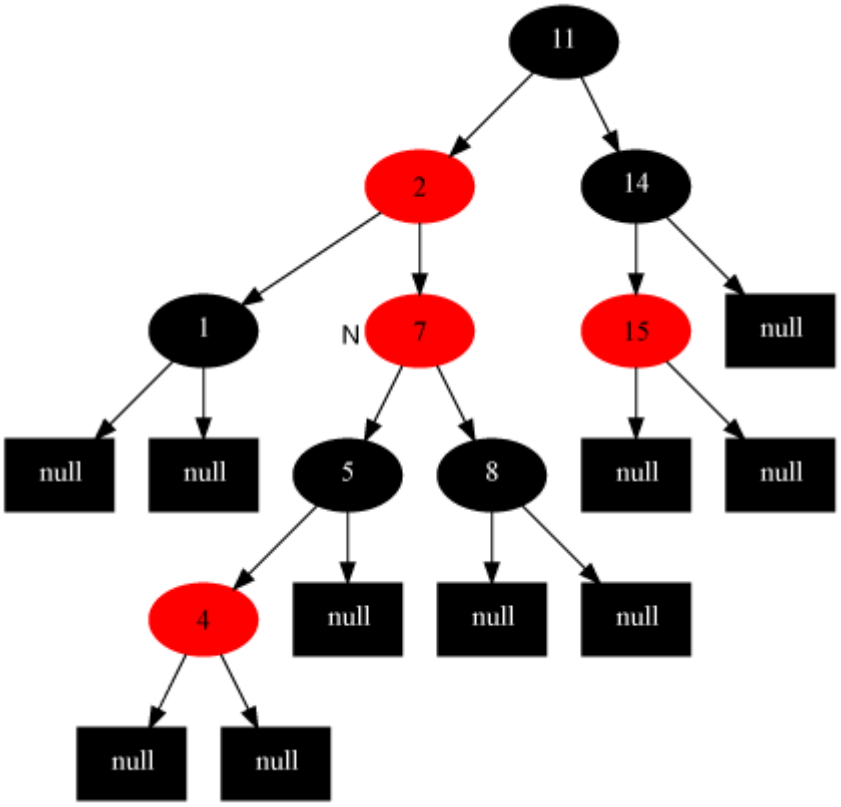
情况1：插入的是根结点。
原树是空树，此情况只会违反性质2。
对策：直接把此结点涂为黑色。
情况2：插入的结点的父结点是黑色。
此不会违反性质2和性质4，红黑树没有被破坏。
对策：什么也不做。
情况3：当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色。
此时父结点的父结点一定存在，否则插入前就已不是红黑树。
与此同时，又分为父结点是祖父结点的左子还是右子，对于对称性，我们只要解开一个方向就可以了。

在此，我们只考虑父结点为祖父左子的情况。
同时，还可以分为当前结点是其父结点的左子还是右子，但是处理方式是一样的。我们将此归为同一类。
对策：将当前节点的父节点和叔叔节点涂黑，祖父结点涂红，把当前结点指向祖父节点，从新的当前节点重新开始算法。

针对情况3，变化前（图片来源：saturnman）[插入4节点]：



变化后：

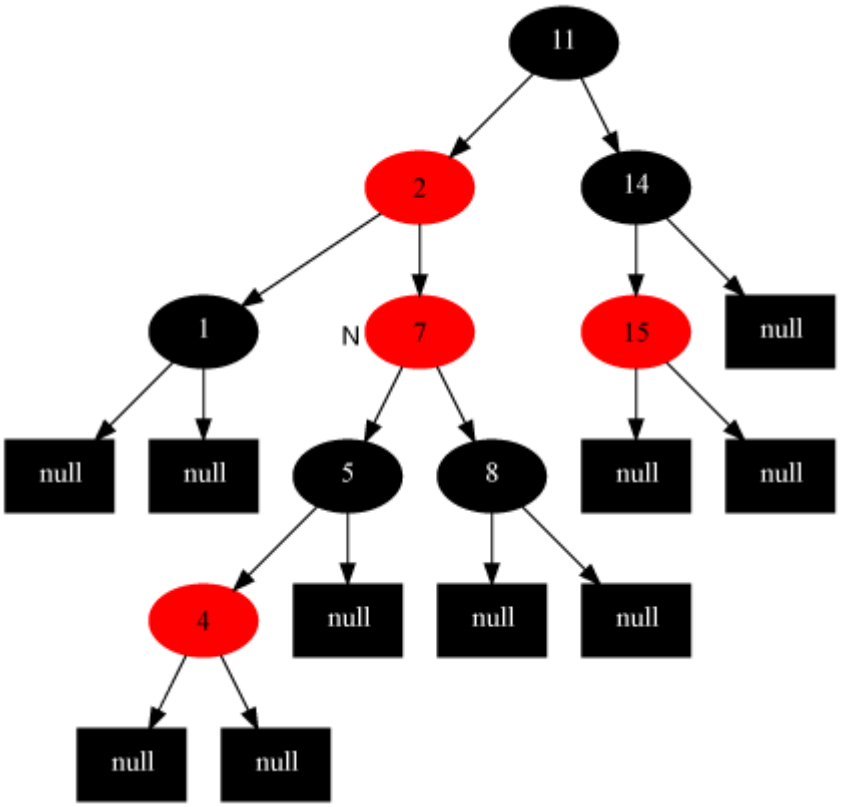


情况4：当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的右

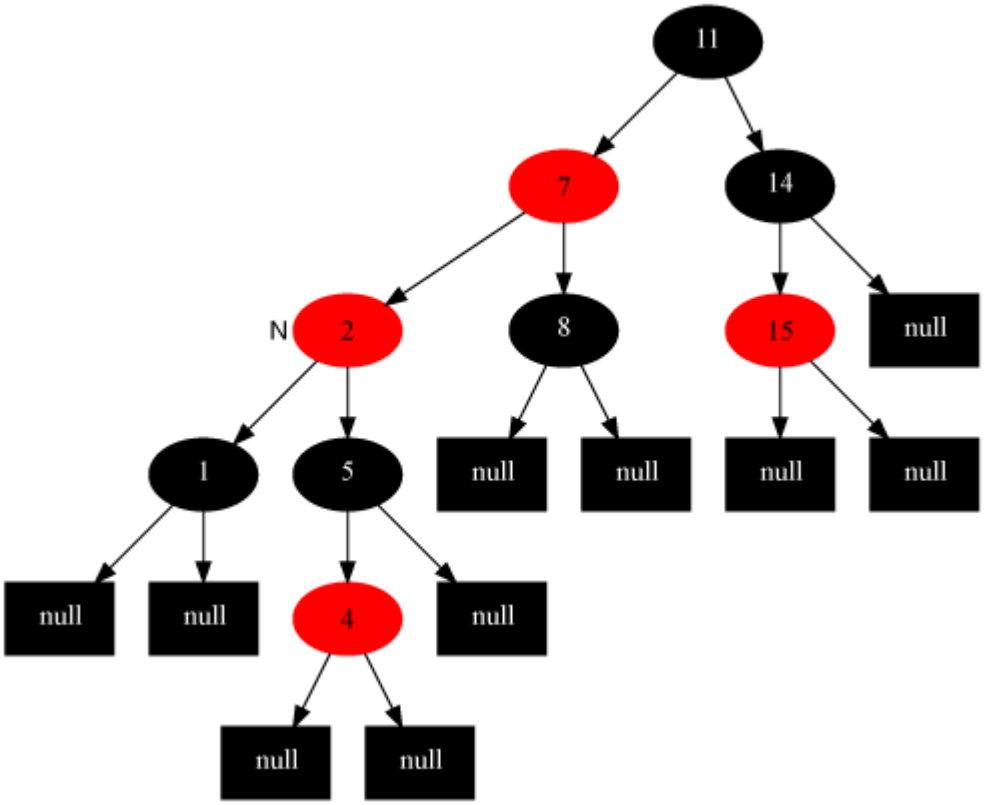
子

对策：当前节点的父节点做为新的当前节点，以新当前节点为支点左旋。

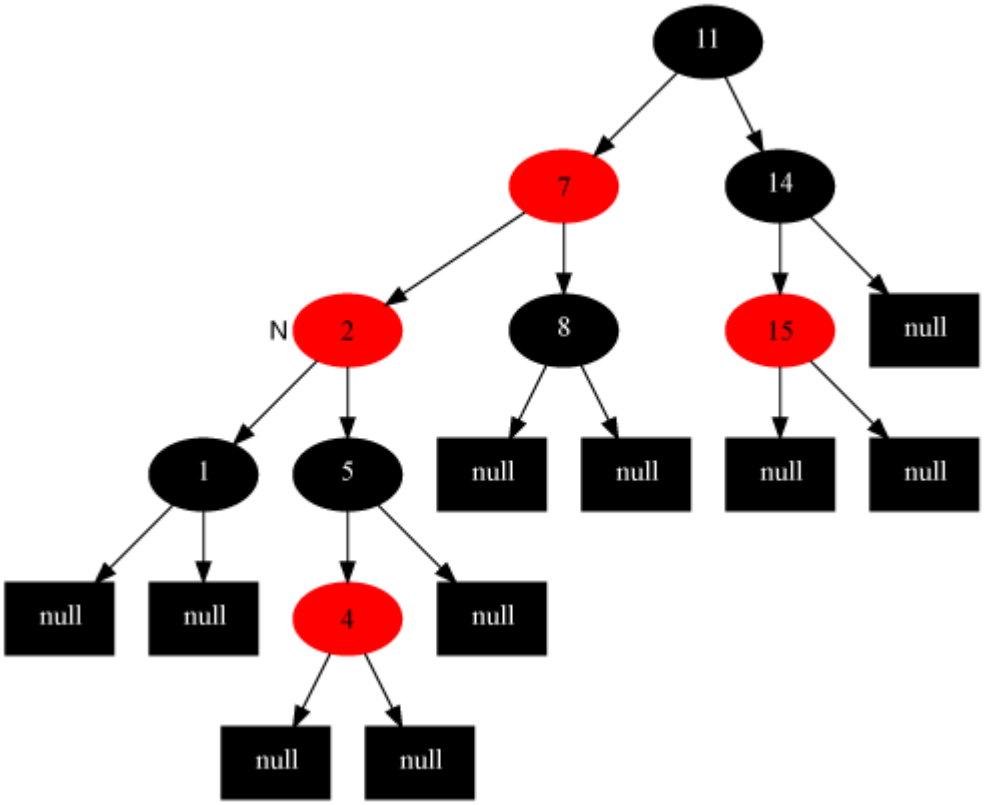
如下图所示，变化前[插入7节点]：



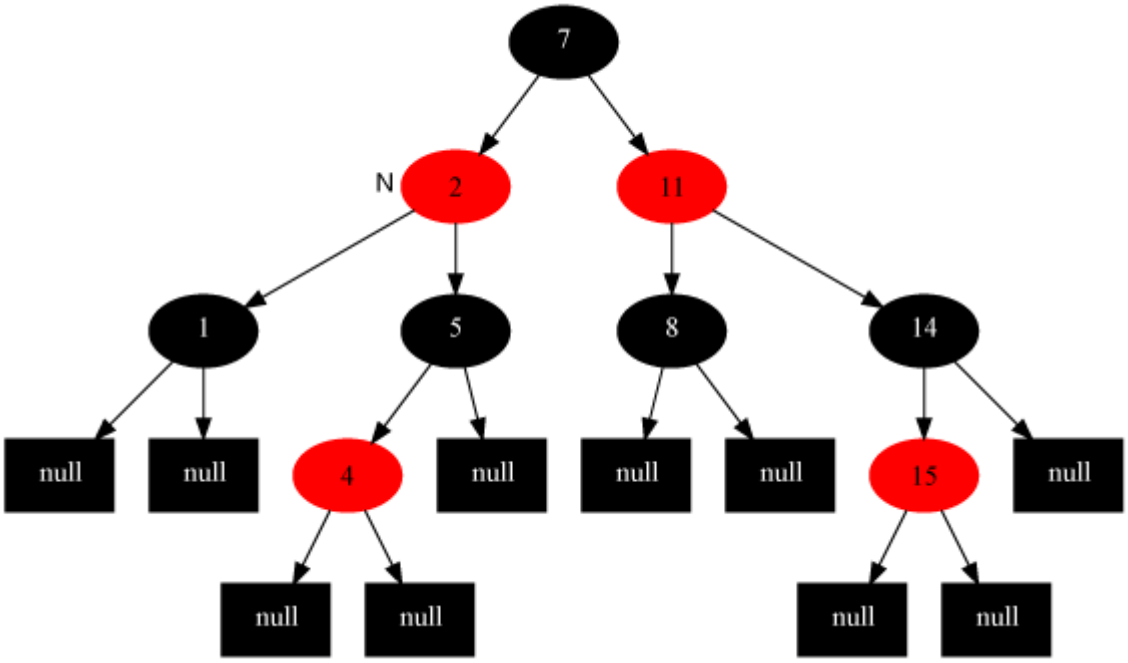
变化后：



情况5：当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的左子
解法：父节点变为黑色，祖父节点变为红色，在祖父节点为支点右旋
如下图所示[插入2节点]



变化后：



==========

三、II、ok，接下来，咱们最后来了解，红黑树的**删除**操作：

算法导论一书，给的算法实现：

RB-DELETE(T, z)　单纯删除结点的总操作

```
 1 if left[z] = nil[T] or right[z] = nil[T]
 2       then y ← z
 3       else y ← TREE-SUCCESSOR(z)
 4 if left[y] ≠ nil[T]
 5       then x ← left[y]
 6       else x ← right[y]
 7 p[x] ← p[y]
 8 if p[y] = nil[T]
 9       then root[T] ← x
10       else if y = left[p[y]]
11                 then left[p[y]] ← x
12                 else right[p[y]] ← x
13 if y 3≠ z
14       then key[z] ← key[y]
15            copy y's satellite data into z
16 if color[y] = BLACK
17       then RB-DELETE-FIXUP(T, x)
18 return y
```

RB-DELETE-FIXUP(T, x)　恢复与保持红黑性质的工作

```
 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

为了保证以下的介绍与阐述清晰，我第三次重写下红黑树的5个性质

1）每个结点要么是红的，要么是黑的。
2）根结点是黑的。
3）每个叶结点，即空结点（NIL）是黑的。
4）如果一个结点是红的，那么它的俩个儿子都是黑的。
5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

（相信，重述了3次，你应该有深刻记忆了。:D）

saturnman：
红黑树删除的几种情况：

博主提醒：
以下所有的操作，是针对红黑树已经删除结点之后，
为了恢复和保持红黑树原有的5点性质，所做的恢复工作。

前面，我已经说了，因为插入、或删除结点后，
可能会违背、或破坏红黑树的原有的性质，
所以为了使插入、或删除结点后的树依然维持为一棵新的红黑树，
那就要做俩方面的工作：
1、部分结点颜色，重新着色
2、调整部分指针的指向，即左旋、右旋。

而下面所有的文字，则是针对红黑树删除结点后，所做的修复红黑树性质的工作。

二零一一年一月七日更新。

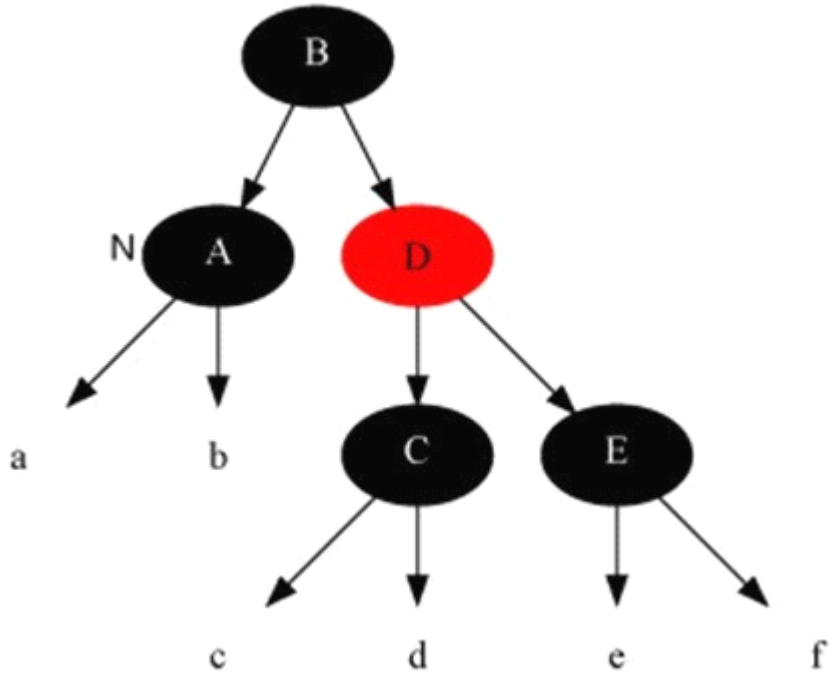
(注：以下的情况3、4、5、6，与上述算法导论之代码 RB-DELETE-FIXUP(T, x) 恢复与保持
中 case1，case2，case3，case4相对应。)
情况1：当前节点是红色
　　解法，直接把当前节点染成黑色，结束。
　　此时红黑树性质全部恢复。
情况2：当前节点是黑色且是根节点
　　解法：什么都不做，结束

情况3：当前节点是黑色，且兄弟节点为红色(此时父节点和兄弟节点的子节点分为黑)。

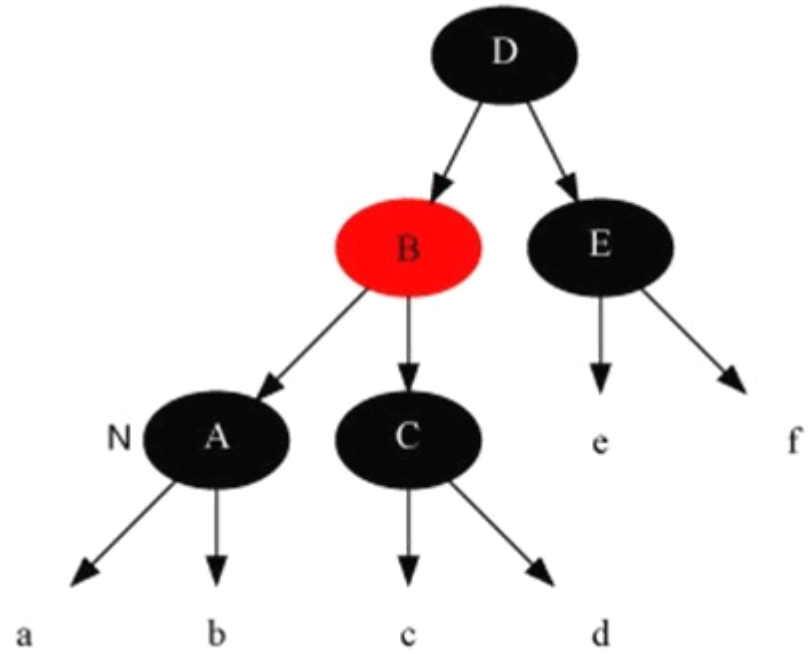
解法：把父节点染成红色，把兄弟结点染成黑色，之后重新进入算法（我们只讨论当前节点是其父节点左孩子时的情况）。

然后，针对父节点做一次左旋。此变换后原红黑树性质5不变，而把问题转化为兄弟节点为黑色的情况。

3.变化前：



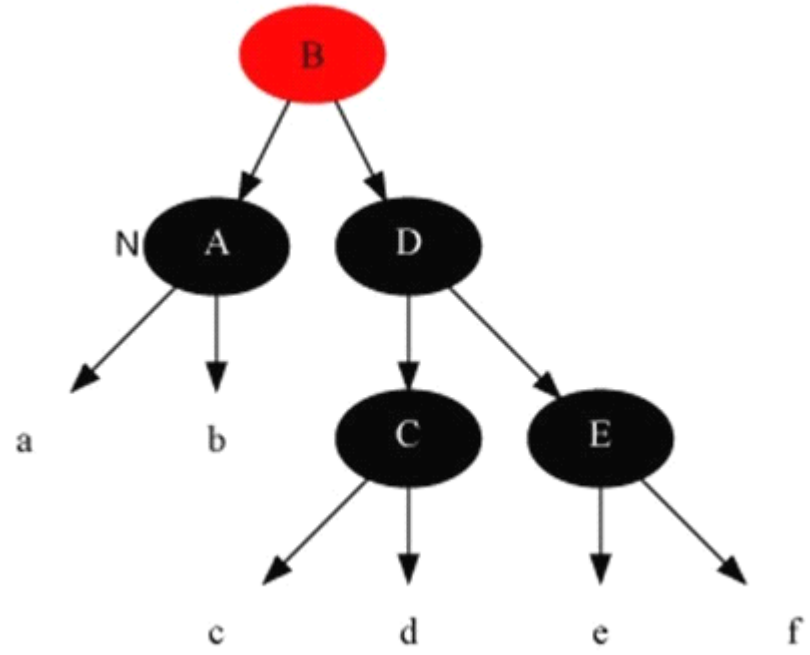
3.变化后：



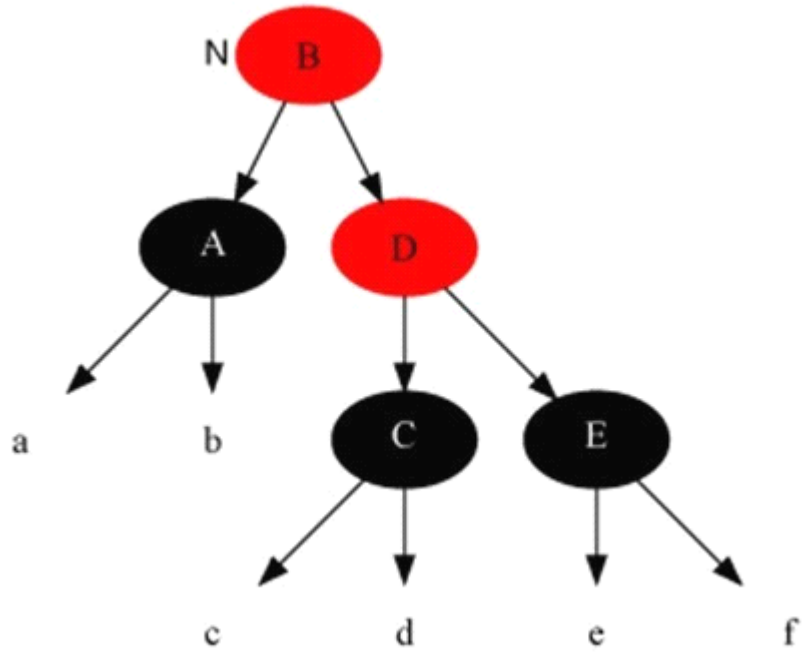
情况4：当前节点是黑色，且兄弟是黑色，且兄弟节点的两个子节点全为黑色。

解法：把当前节点和兄弟节点中抽取一重黑色追加到父节点上，把父节点当成新的当前节点，重新进入算法。（此变换后性质5不变）

4.变化前



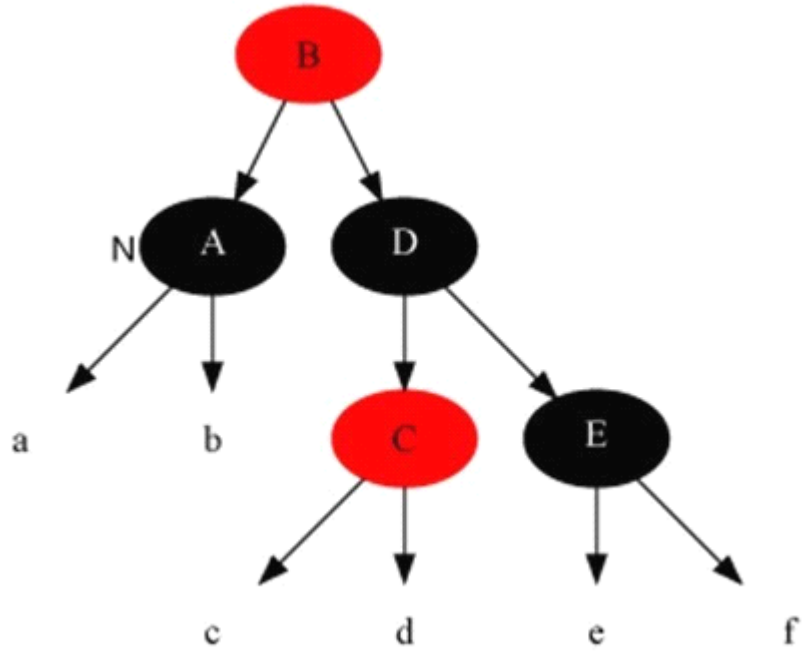
4.变化后



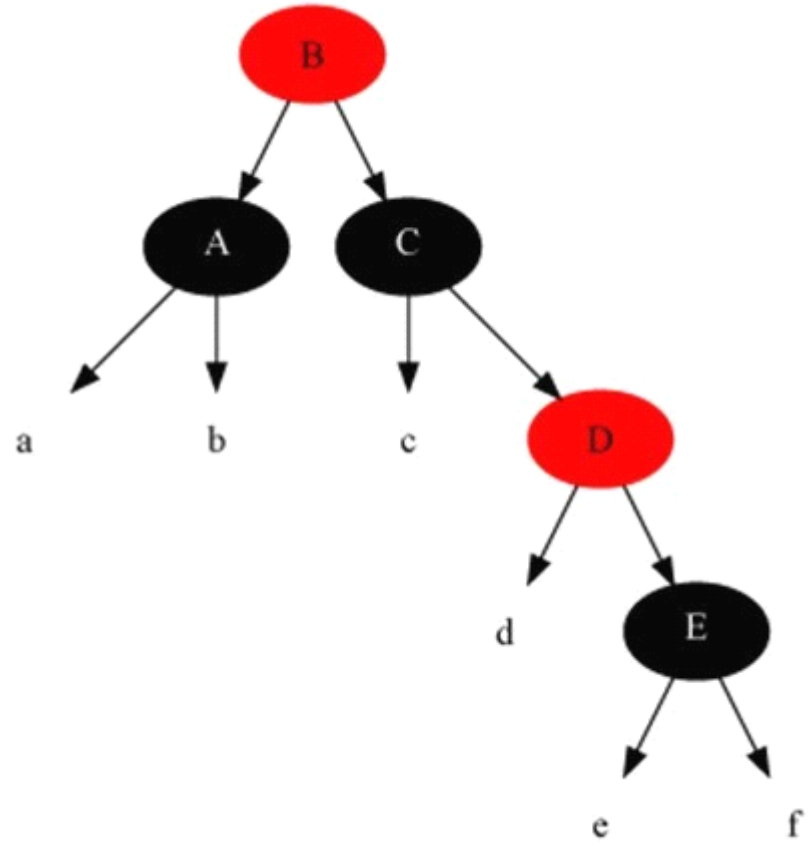
情况5：当前节点颜色是黑色，兄弟节点是黑色，兄弟的左子是红色，右子是黑色。

解法：把兄弟结点染红，兄弟左子节点染黑，之后再在兄弟节点为支点解右旋，之后重新进入算法。此是把当前的情况转化为情况6，而性质5得以保持。

5. 变化前：



5. 变化后：

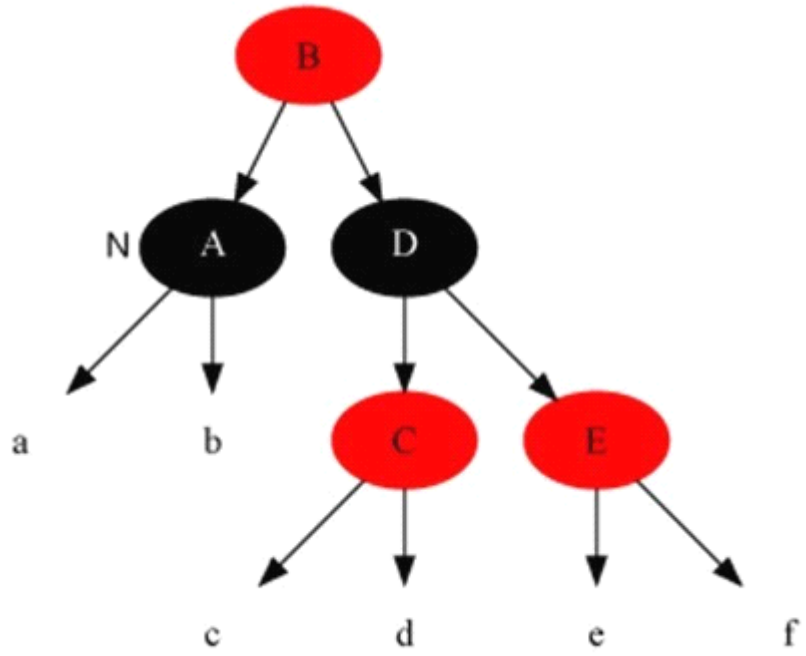


情况6：当前节点颜色是黑色，它的兄弟节点是黑色，但是兄弟节点的右子是红色，兄弟节点左子的颜色任意。

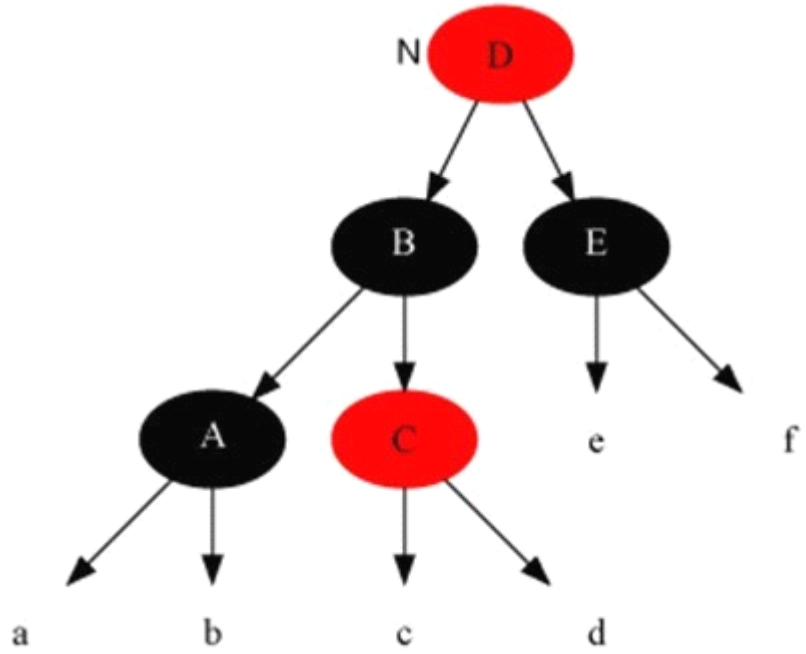
解法：把兄弟节点染成当前节点父节点的颜色，把当前节点父节点染成黑色，兄弟节点右子染成黑色，

之后以当前节点的父节点为支点进行左旋，此时算法结束，红黑树所有性质调整正确。

6. 变化前：



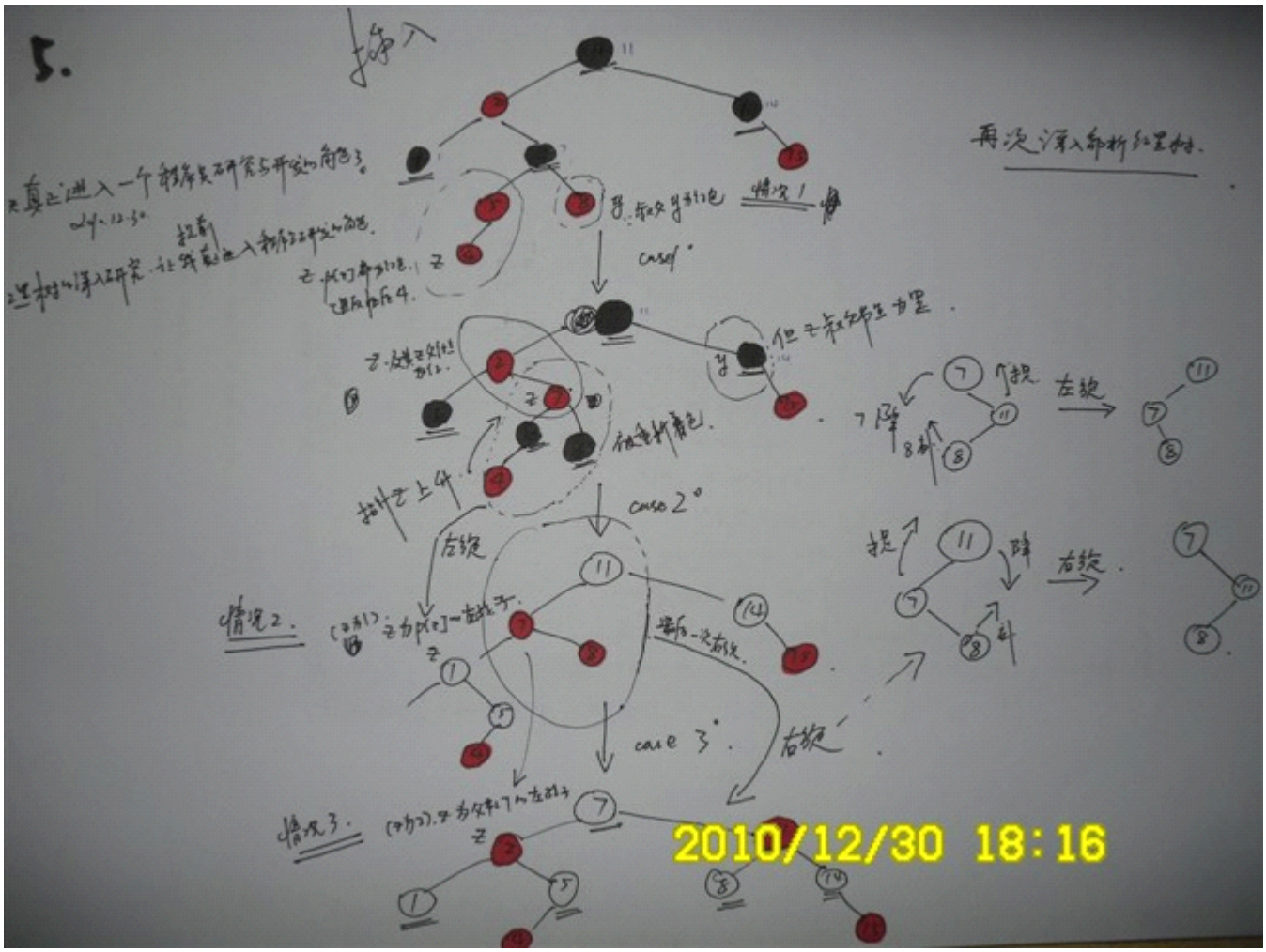
6.变化后：



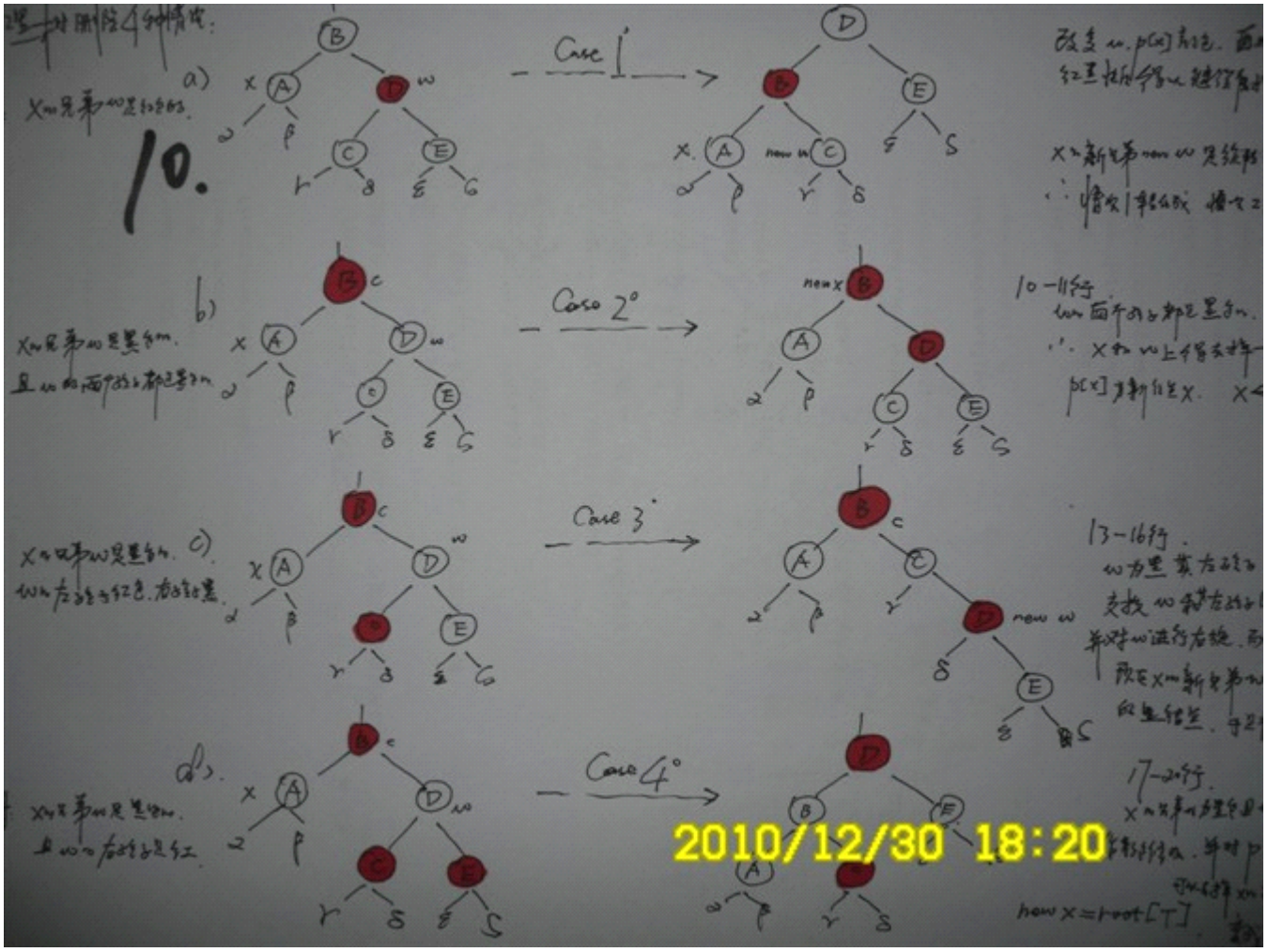
限于篇幅，不再过多赘述。更多，可参考算法导论或下文我写的第二篇文章。
完。
July、二零一零年十二月二十九日初稿。三十日凌晨修订。行文3个小时以上。

今下午画红黑树画了好几个钟头，贴俩张图：

红黑树插入的3种情况：



红黑树删除的4种情况：



ok，只贴俩张，更多，参考我写的关于红黑树的第二篇文章：
红黑树算法的层层剖析与逐步实现[推荐]
http://blog.csdn.net/v_JULY_v/archive/2010/12/31/6109153.aspx
这篇文章针对算法实现源码分十层，层层、逐层剖析，相信，更清晰易懂。
或者，saturnman的这篇文章：
http://saturnman.blog.163.com/blog/static/557611201096942038 3/。

July、二零一零年十二月三十一日、最后更新。

1、教你透彻了解红黑树
2、红黑树算法的实现与剖析
3、红黑树的 c 源码实现与剖析
4、一步一图一代码，R-B Tree
5、红黑树插入和删除结点的全程演示
6、红黑树的 c++完整实现源码

版权声明