教你初步了解红黑树

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：

1. *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。
2. 本文的 github 优化版：https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md。

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。

红黑树，作为一棵二叉查找树，满足二叉查找树的一般性质。下面，来了解下 二叉查找树的一般性质。

二叉查找树

二叉查找树，也称有序二叉树（ordered binary tree），或已排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

- 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 任意节点的左、右子树也分别为二叉查找树。
- 没有键值相等的节点（no duplicate nodes）。

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，二叉查找树的一般操作的执行时间为 O(lgn)。但二叉查找树若退化成了一棵具有 n 个结点的线性链后，则这些操作最坏情况运行时间为 O(n)。

红黑树虽然本质上是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为 O(log n)。

但它是如何保证一棵 n 个结点的红黑树的高度始终保持在 logn 的呢？这就引出了**红黑树的 5 个性质：**

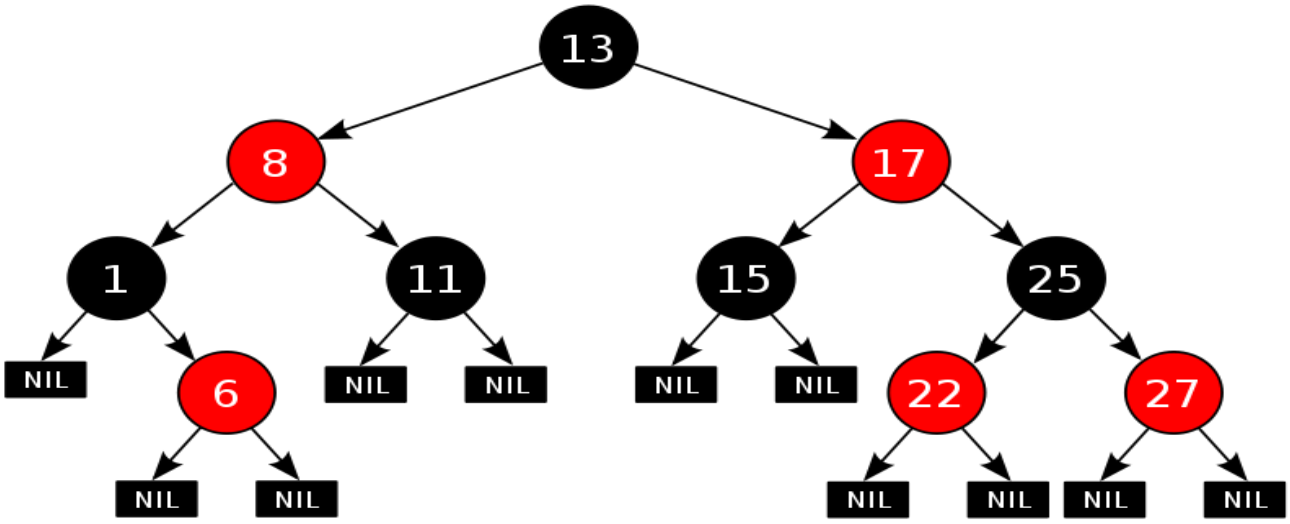
1. 每个结点要么是红的要么是黑的。
2. 根结点是黑的。
3. 每个叶结点（叶结点即指树尾端 NIL 指针或 NULL 结点）都是黑的。
4. 如果一个结点是红的，那么它的两个儿子都是黑的。

5. 对于任意结点而言，其到叶结点树尾端 NIL 指针的每条路径都包含相同数目的黑结点。

正是红黑树的这 5 条性质，使一棵 n 个结点的红黑树始终保持了 logn 的高度，从而也就解释了上面所说的“红黑树的查找、插入、删除的时间复杂度最坏为 O(log n)”这一结论成立的原因。

（注：上述第 3、5 点性质中所说的 NULL 结点，包括 wikipedia.算法导论上所认为的叶子结点即为树尾端的 NIL 指针，或者说 NULL 结点。然百度百科以及网上一些其它博文直接说的叶结点，则易引起误会，因，此叶结点非子结点）

如下图所示，即是一颗红黑树(下图引自 wikipedia：http://t.cn/hgvH1l)：



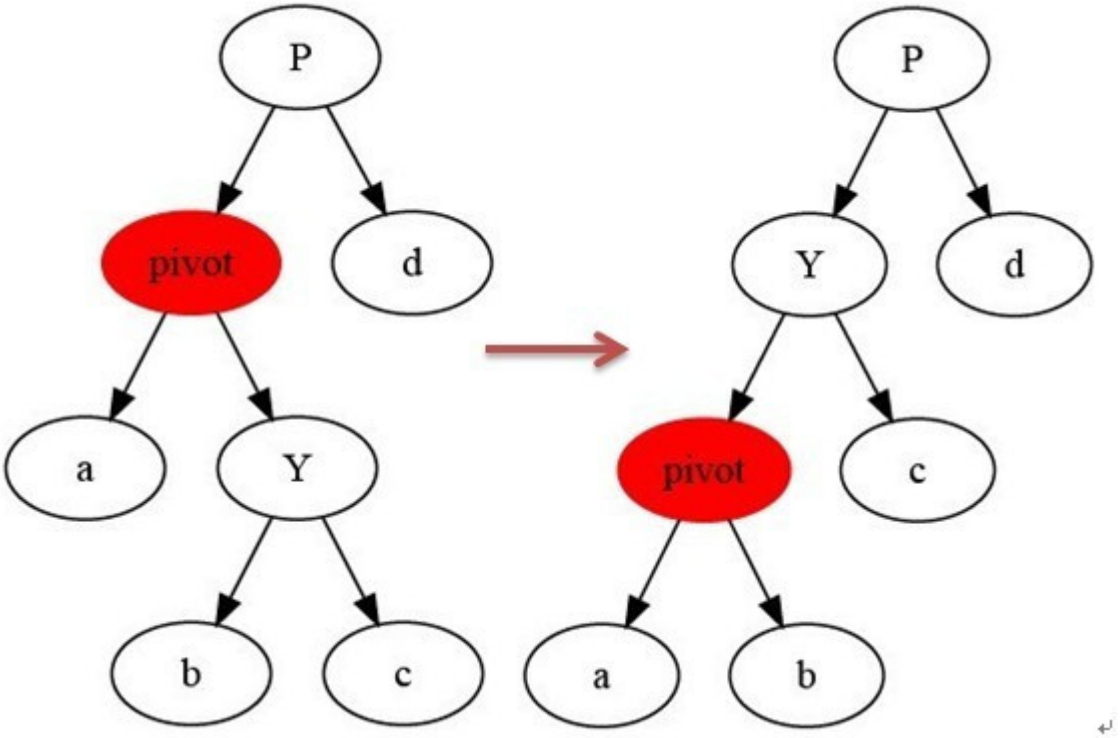
此图忽略了叶子和根部的父结点。同时，上文中我们所说的 "叶结点" 或"NULL 结点"，如上图所示，它不包含数据而只充当树在此结束的指示，这些节点在绘图中经常被省略，望看到此文后的读者朋友注意。

二、树的旋转知识

当在对红黑树进行插入和删除等操作时，对树做了修改可能会破坏红黑树的性质。为了继续保持红黑树的性质，可以通过对结点进行重新着色，以及对树进行相关的旋转操作，即通过修改树中某些结点的颜色及指针结构，来达到对红黑树进行插入或删除结点等操作后继续保持它的性质或平衡的目的。

树的旋转分为左旋和右旋，下面借助图来介绍一下左旋和右旋这两种操作。

1.左旋

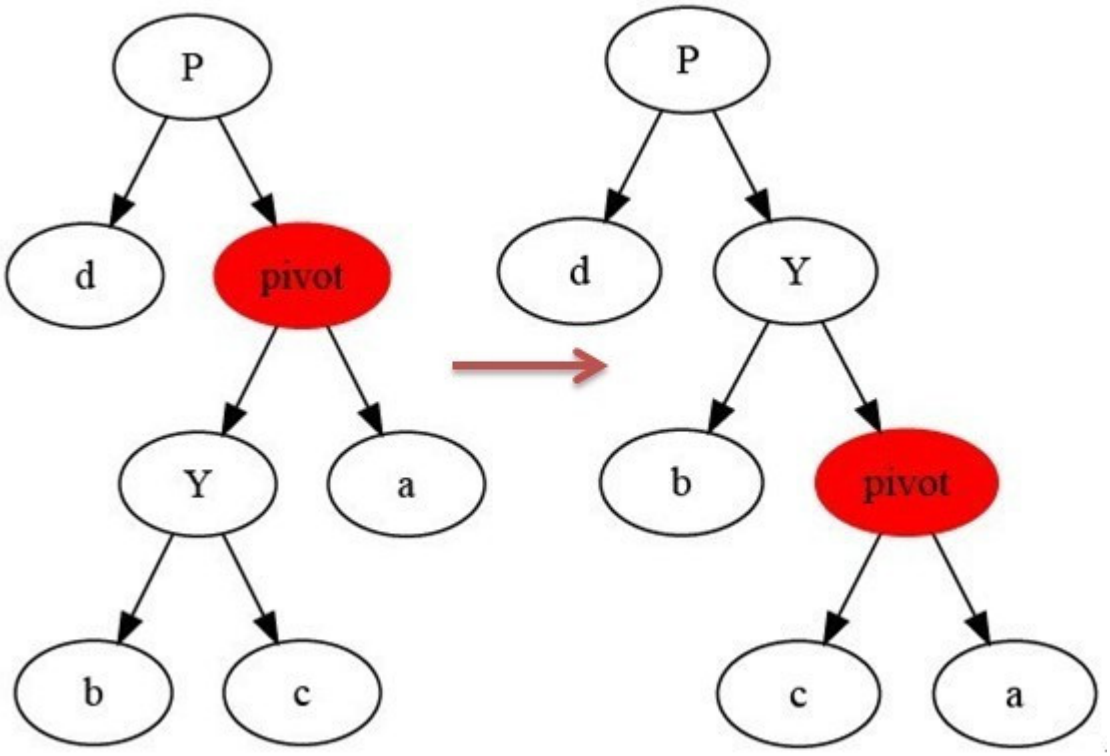


如上图所示，当在某个结点 pivot 上，做左旋操作时，我们假设它的右孩子 y 不是 NIL[T]，pivot 可以为任何不是 NIL[T]的左子结点。左旋以 pivot 到 Y 之间的链为“支轴”进行，它使 Y 成为该子树的新根，而 Y 的左孩子 b 则成为 pivot 的右孩子。

```cpp
LeftRoate(T, x)
y ← x.right                    //定义 y：y 是 x 的右孩子
x.right ← y.left                //y 的左孩子成为 x 的右孩子
if y.left ≠ T.nil
    y.left.p ← x
y.p ← x.p                      //x 的父结点成为 y 的父结点
if x.p = T.nil
    then T.root ← y
else if x = x.p.left
    then x.p.left ← y
else x.p.right ← y
y.left ← x                      //x 作为 y 的左孩子
x.p ← y
```

2.右旋

右旋与左旋差不多，再此不做详细介绍。



树在经过左旋右旋之后，树的搜索性质保持不变，但树的红黑性质则被破坏了，所以，红黑树插入和删除数据后，需要利用旋转与颜色重涂来重新恢复树的红黑性质。

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、红黑树的插入

要真正理解红黑树的插入，还得先理解二叉查找树的插入。磨刀不误砍柴工，咱们再来了解一下二叉查找树的插入和红黑树的插入。

如果要在二叉查找树中插入一个结点，首先要查找到结点要插入的位置，然后进行插入。假设插入的结点为 z 的话，插入的伪代码如下：

[cpp] view plain copy

```
1.  TREE-INSERT(T, z)
2.  y ← NIL
3.  x ← T.root
4.  while x ≠ NIL
5.      do y ←  x
6.      if z.key < x.key
7.          then x ← x.left
8.      else x ← x.right
9.  z.p ← y
10. if y == NIL
11.     then T.root ← z
12. else if z.key < y.key
13.     then y.left ← z
14. else y.right ← z
```

红黑树的插入和插入修复

现在我们了解了二叉查找树的插入，接下来，咱们便来具体了解下红黑树的插入操作。红黑树的插入相当于在二叉查找树插入的基础上，为了重新恢复平衡，继续做了插入修复操作。

假设插入的结点为 z，红黑树的插入伪代码具体如下所示：

[cpp] view plain copy

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

把上面这段红黑树的插入代码，跟之前看到的二叉查找树的插入代码比较一下可以看出，RB-INSERT(T, z)前面的第 1～13 行代码基本上就是二叉查找树的插入代码，然后第 14～16 行代码把 z 的左孩子和右孩子都赋为叶结点 nil，再把 z 结点着为红色，最后为保证红黑性质在插入操作后依然保持，调用一个辅助程序 RB-INSERT-FIXUP 来对结点进行重新着色，并旋转。

换言之，如果插入的是根结点，由于原树是空树，此情况只会违反性质 2，因此直接把此结点涂为黑色；如果插入的结点的父结点是黑色，由于此不会违反性质 2 和性质 4，红黑树没有被破坏，所以此时什么也不做。

但当遇到下述 3 种情况时又该如何调整呢？

- ● 插入修复情况 1：如果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

  ● 插入修复情况 2：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

  ● 插入修复情况 3：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

答案就是根据红黑树插入代码 RB-INSERT(T, z)最后一行调用的 RB-INSERT-FIXUP(T, z)函数所示的步骤进行操作，具体如下所示：

[cpp] view plain copy

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

下面，咱们来分别处理上述 3 种插入修复情况。

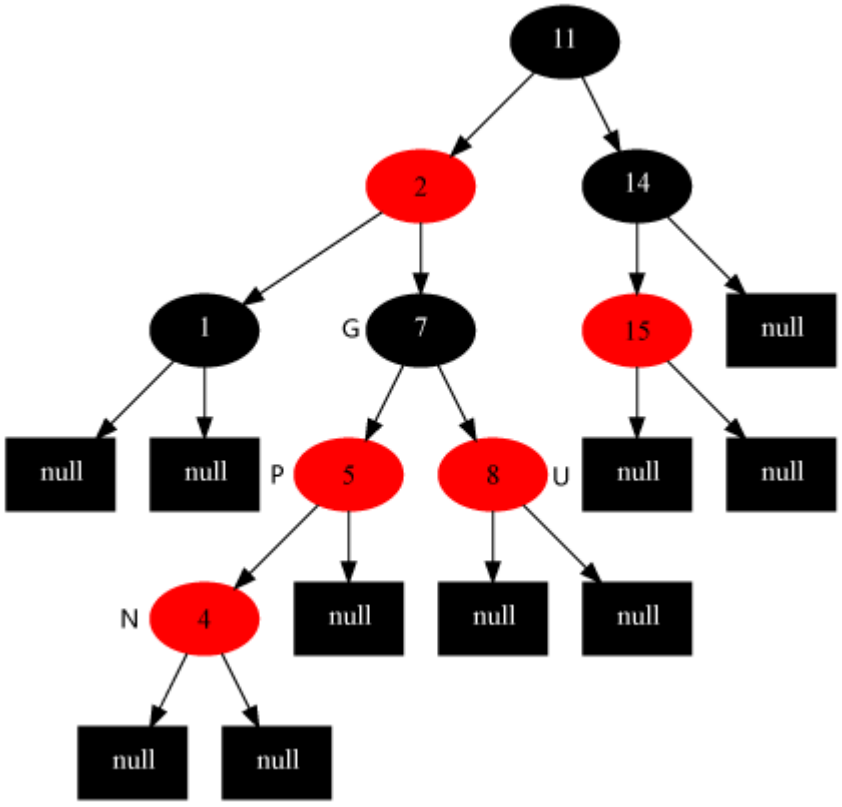
- **插入修复情况 1：当前结点的父结点是红色，祖父结点的另一个子结点（叔叔结点）是红色。**

如下代码所示：

[cpp] view plain copy

```
1.  while z.p.color == RED
2.      do if z.p == z.p.p.left
3.          then y ← z.p.p.right
4.          if y.color == RED
```

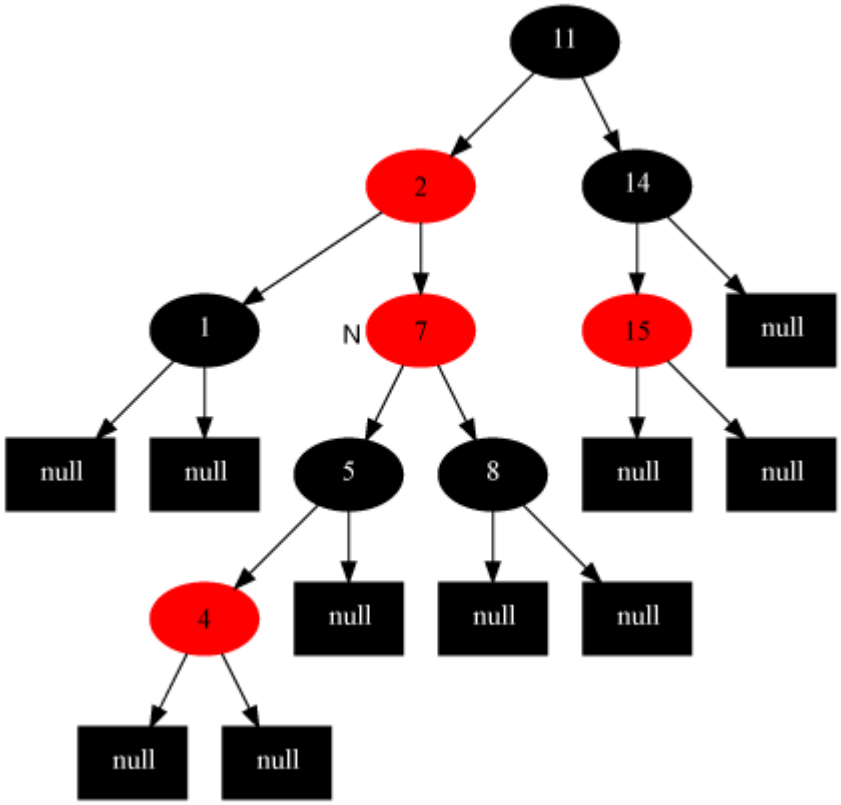
此时父结点的父结点一定存在，否则插入前就已不是红黑树。与此同时，又分为父结点是祖父结点的左孩子还是右孩子，根据对称性，我们只要解开一个方向就可以了。这里只考虑父结点为祖父左孩子的情况，如下图所示。



对此，我们的解决策略是：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。即如下代码所示：

```
[cpp] view plain copy
1.  then z.p.color ← BLACK                ▸ Case 1
2.  y.color ← BLACK                     ▸ Case 1
3.  z.p.p.color ← RED                     ▸ Case 1
4.  z ← z.p.p                           ▹ Case 1
```

所以，变化后如下图所示：



于是，插入修复情况 1 转换成了插入修复情况 2。

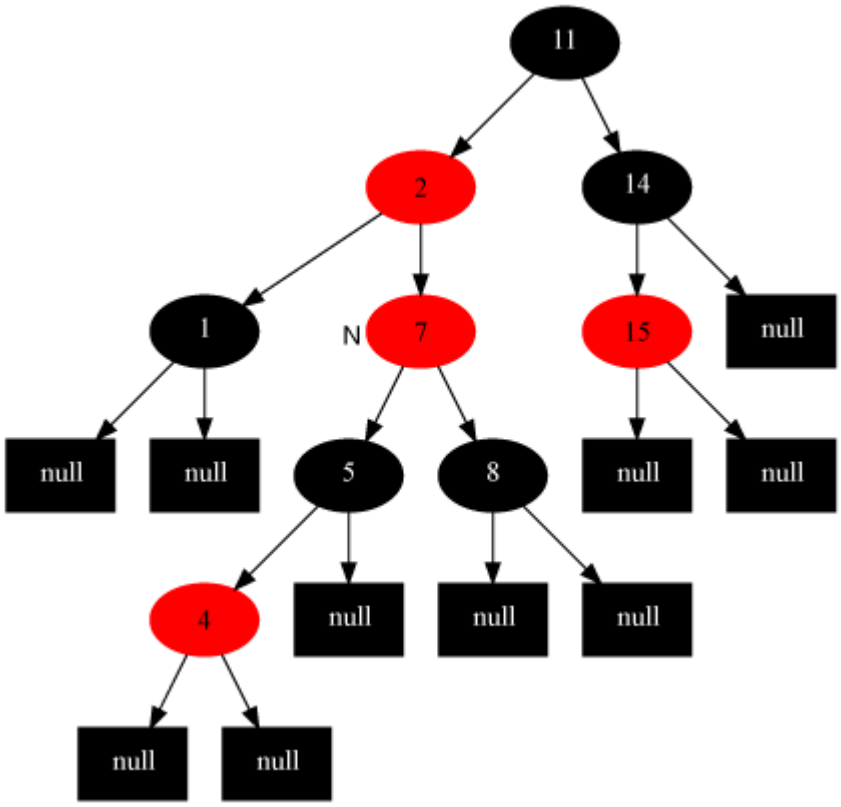
- **插入修复情况 2**：当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的右子

此时，解决对策是：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。即如下代码所示：

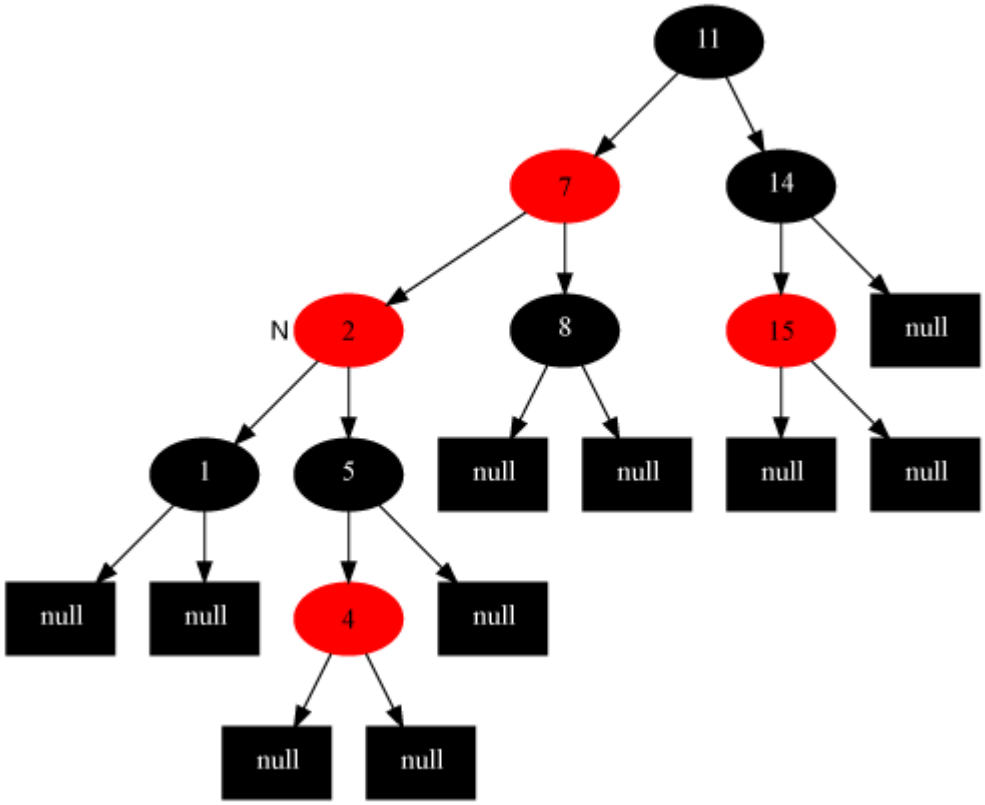
[cpp] view plain copy

```
1.  else if z == z.p.right
2.       then z ← z.p                          ▸ Case 2
3.       LEFT-ROTATE(T, z)                     ▹ Case 2
```

所以红黑树由之前的：



变化成：



从而插入修复情况 2 转换成了插入修复情况 3。

- **插入修复情况 3**：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**孩子**

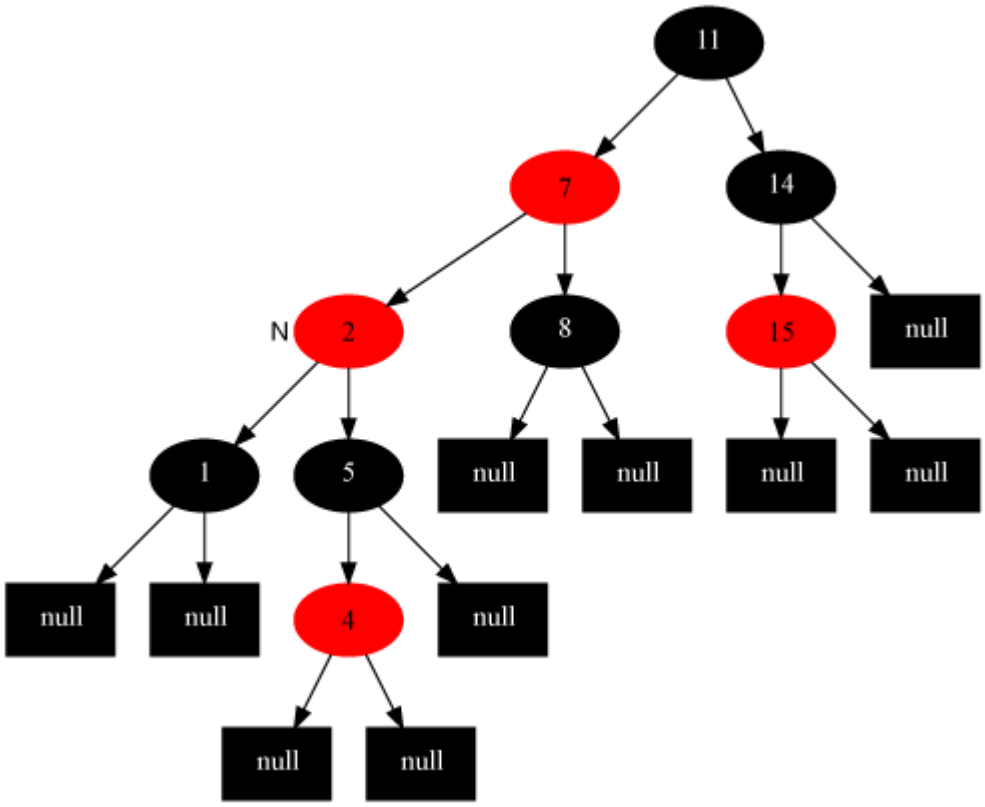
解决对策是：父节点变为黑色，祖父节点变为红色，在祖父节点为支点右旋，

操作代码为：

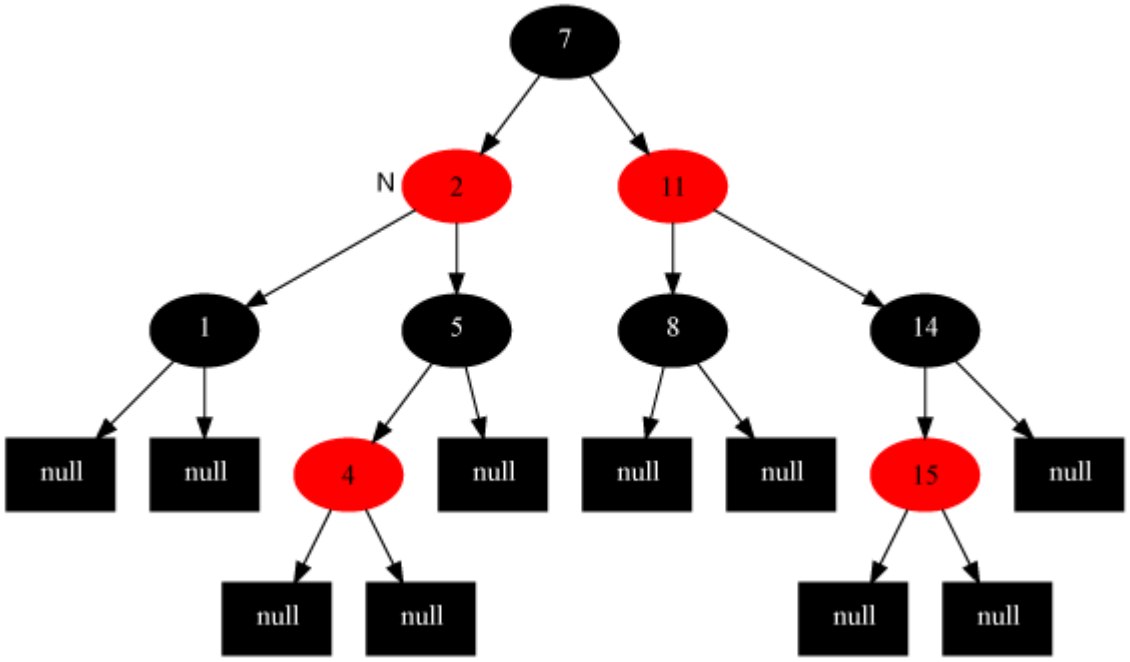
[cpp] view plain copy

```
1.  z.p.color ← BLACK                    ▹ Case 3
2.  z.p.p.color ← RED                    ▹ Case 3
3.  RIGHT-ROTATE(T, z.p.p)               ▹ Case 3
```

最后，把根结点涂为黑色，整棵红黑树便重新恢复了平衡。所以红黑树由之前的：



变化成：



「回顾：经过上面情况 3、情况 4、情况 5 等 3 种插入修复情况的操作示意图，读者自会发现，后面的情况 4、情况 5 都是针对情况 3 插入节点 4 以后，进行的一系列插入修复情况操作，不过，指向当前节点 N 指针一直在变化。所以，你可以想当然的认为：整个下来，情况 3、4、5 就是一个完整的插入修复情况的操作流程」

四、红黑树的删除

接下来，咱们最后来了解，红黑树的删除操作。

"我们删除的节点的方法与常规二叉搜索树中删除节点的方法是一样的，如果被删除的节点不是有双非空子女，则直接删除这个节点，用它的唯一子节点顶替它的位置，如果它的子节点分是空节点，那就用空节点顶替它的位置，如果它的双子全为非空，我们就把它的直接后继节点内容复制到它的位置，之后以同样的方式删除它的后继节点，它的后继节点不可能是双子非空，因此此传递过程最多只进行一次。”

二叉查找树的删除

继续讲解之前，补充说明下二叉树结点删除的几种情况，待删除的节点按照儿子的个数可以分为三种：

1. 没有儿子，即为叶结点。直接把父结点的对应儿子指针设为 NULL，删除儿子结点就 OK 了。
2. 只有一个儿子。那么把父结点的相应儿子指针指向儿子的独生子，删除儿子结点也 OK 了。
3. 有两个儿子。这是最麻烦的情况，因为你删除节点之后，还要保证满足搜索二叉树的结构。其实也比较容易，我们可以选择左儿子中的最大元素或者右儿子中的最小元素放到待删除节点的位置，就可以保证结构的不变。当然，你要记得调整子树，毕竟又出现了节点删除。习惯上大家选择左儿子中的最大元素，其实选择右儿子的最小元素也一样，没有任何差别，只是人们习惯

从左向右。这里咱们也选择左儿子的最大元素，将它放到待删结点的位置。

左儿子的最大元素其实很好找，只要顺着左儿子不断的去搜索右子树就可以了，直到找到一个没有右子树的结点。那就是最大的了。

二叉查找树的删除代码如下所示：

[cpp] view plain copy

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

红黑树的删除和删除修复

OK，回到红黑树上来，红黑树结点删除的算法实现是：

RB-DELETE(T, z) 单纯删除结点的总操作

[cpp] view plain copy

```
1.    1 if left[z] = nil[T] or right[z] = nil[T]
2.    2    then y ← z
3.    3    else y ← TREE-SUCCESSOR(z)
4.    4 if left[y] ≠ nil[T]
5.    5    then x ← left[y]
6.    6    else x ← right[y]
7.    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 ≠ 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
```

“在删除节点后，原红黑树的性质可能被改变，如果删除的是红色节点，那么原红黑树的性质依旧保持，此时不用做修正操作，如果删除的节点是黑色节点，原红黑树的性质可能会被改变，我们要对其做修正操作。那么哪些树的性质会发生变化呢，如果删除节点不是树唯一节点，那么删除节点的那一个支的到各叶节点的黑色节点数会发生变化，此时性质 5 被破坏。如果被删节点的唯一非空子节点是红色，而被删节点的父节点也是红色，那么性质 4 被破坏。如果被删节点是根节点，而它的唯一非空子节点是红色，则删除后新根节点将变成红色，违背性质 2。”

RB-DELETE-FIXUP(T, x) 恢复与保持红黑性质的工作

```
[cpp] view plain copy
 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.  13                          then color[left[w]] ← BLACK          ▷  Case 3
14.  14                               color[w] ← RED                  ▷  Case 3
15.  15                               RIGHT-
     ROTATE(T, w)                ▷  Case 3
16.  16                               w ← right[p[x]]                 ▷  Case 3
17.  17                          color[w] ← color[p[x]]               ▷  Case 4
18.  18                          color[p[x]] ← BLACK                  ▷  Case 4
19.  19                          color[right[w]] ← BLACK              ▷  Case 4
20.  20                          LEFT-
     ROTATE(T, p[x])                    ▷  Case 4
21.  21                          x ← root[T]                          ▷  Case 4
22.  22        else (same as then clause with "right" and "left" exchanged)
23.  23 color[x] ← BLACK
```

“上面的修复情况看起来有些复杂，下面我们用一个分析技巧：我们从被删节点后来顶替它的那个节点开始调整，并认为它有额外的一重黑色。这里额外一重黑色是什么意思呢，我们不是把红黑树的节点加上除红与黑的另一种颜色，这里只是一种假设，我们认为我们当前指向它，因此空有额外一种黑色，可以认为它的黑色是从它的父节点被删除后继承给它的，它现在可以容纳两种颜色，如果它原来是红色，那么现在是红+黑，如果原来是黑色，那么它现在的颜色是黑+黑。有了这重额外的黑色，原红黑树性质 5 就能保持不变。现在只要恢复其它性质就可以了，做法还是尽量向根移动和穷举所有可能性。"--saturnman。

如果是以下情况，恢复比较简单：

- a)当前节点是红+黑色
  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。
- b)当前节点是黑+黑且是根节点， 解法：什么都不做，结束。
  但如果是以下情况呢？：
- 删除修复情况 1：当前节点是黑+黑且兄弟节点为红色(此时父节点和兄弟节点的子节点分为黑)

- 删除修复情况 2：当前节点是黑加黑且兄弟是黑色且兄弟节点的两个子节点全为黑色
- 删除修复情况 3：当前节点颜色是黑+黑，兄弟节点是黑色，兄弟的左子是红色，右子是黑色
- 删除修复情况 4：当前节点颜色是黑-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

此时，我们需要调用 RB-DELETE-FIXUP(T, x)，来恢复与保持红黑性质的工作。

下面，咱们便来分别处理这 4 种删除修复情况。

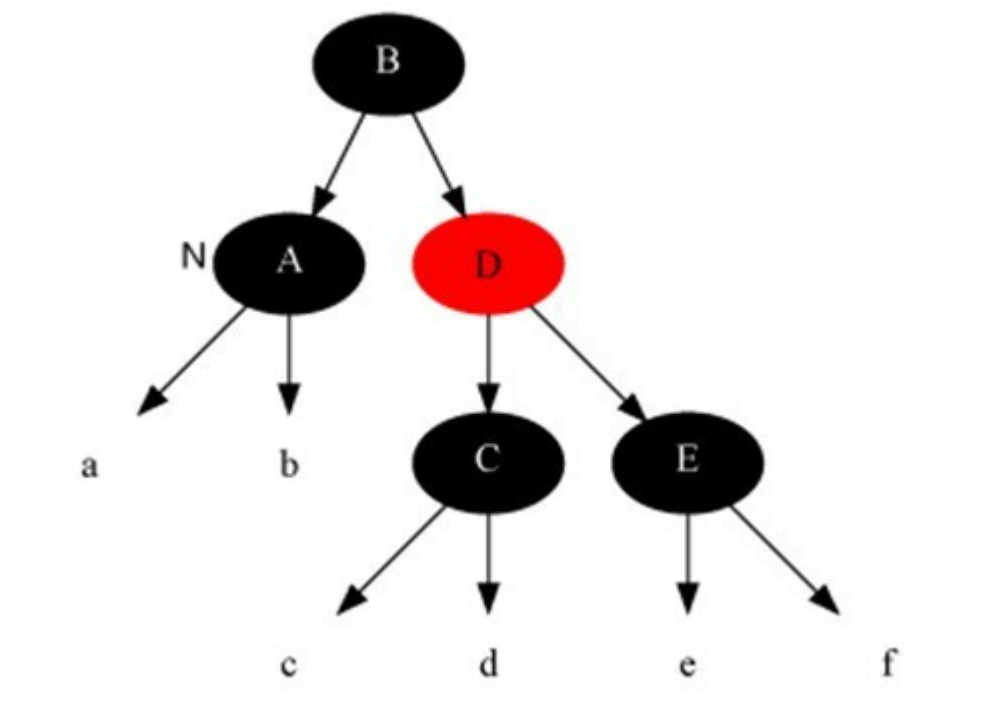
删除修复情况 1：当前节点是黑+黑且兄弟节点为红色(此时父节点和兄弟节点的子节点分为黑)。

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）。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(注：变化前，原本就未违反性质 5，只是为了把问题转化为兄弟节点为黑色的情况)。 即如下代码操作：

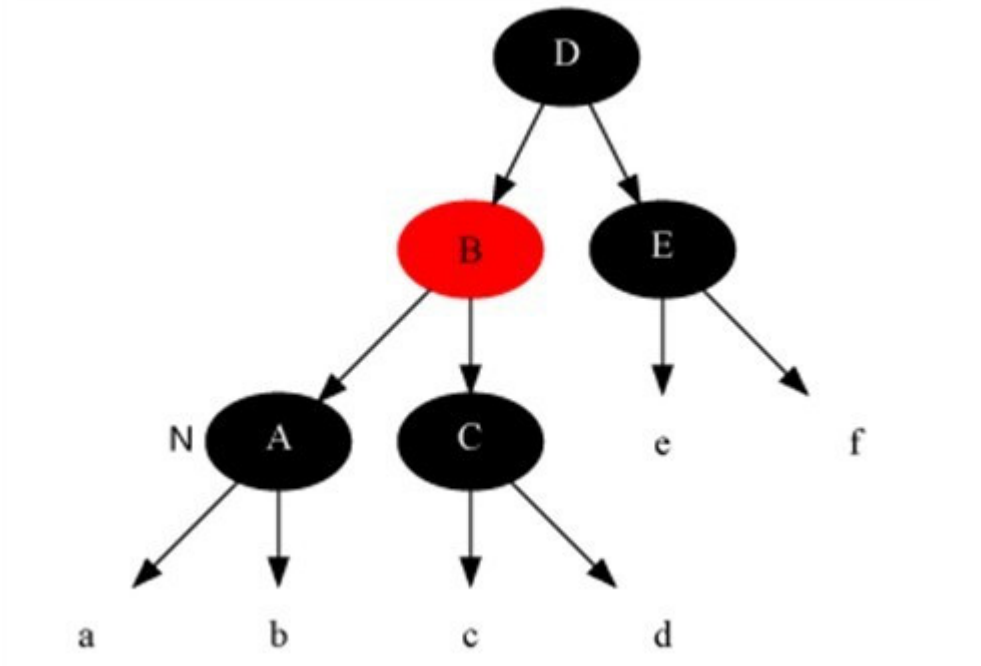
[cpp] view plain copy

```
1.  //调用 RB-DELETE-FIXUP(T, x) 的 1-8 行代码
2.    1 while x ≠ root[T] and color[x] = BLACK
3.    2     do if x = left[p[x]]
4.    3           then w ← right[p[x]]
5.    4                if color[w] = RED
6.    5                   then color[w] ← BLACK                ▹ Case 1
7.    6                        color[p[x]] ← RED               ▹ Case 1
8.    7                        LEFT-ROTATE(T, p[x])            ▹ Case 1
9.    8                        w ← right[p[x]]                 ▹ Case 1
```

变化前：



变化后:

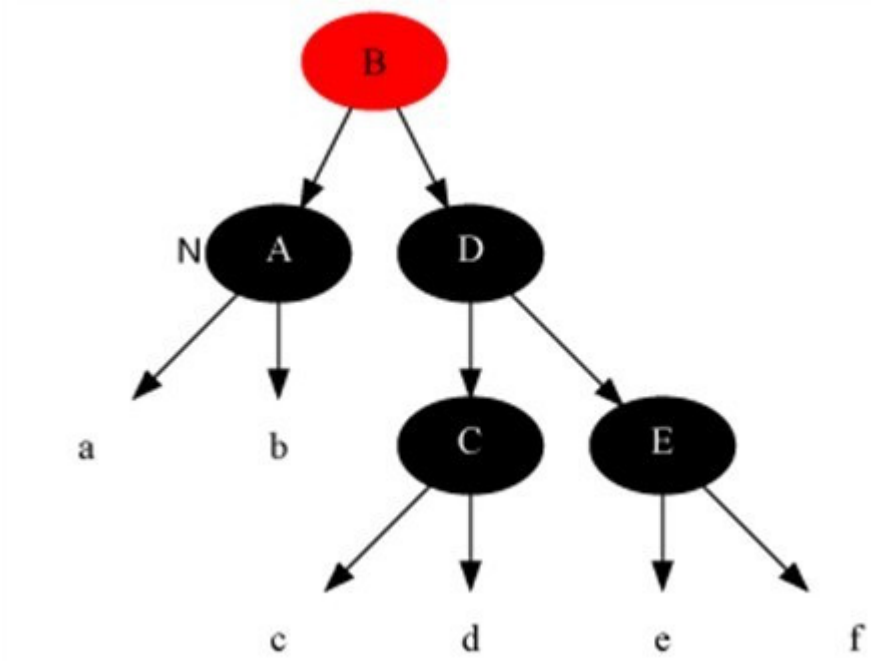


删除修复情况 2：当前节点是黑加黑且兄弟是黑色且兄弟节点的两个子节点全为黑色。

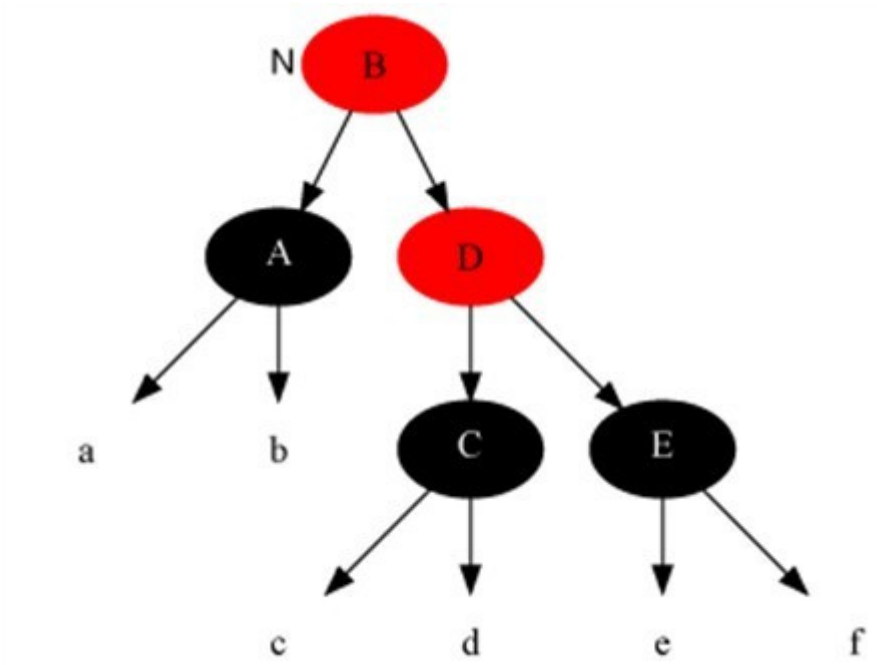
解法：把当前节点和兄弟节点中抽取一重黑色追加到父节点上，把父节点当成新的当前节点，重新进入算法。（此变换后性质 5 不变），即调用 RB-INSERT-FIXUP(T, z) 的第 9-10 行代码操作，如下：

```
[cpp] view plain copy
1.  //调用 RB-DELETE-FIXUP(T, x) 的 9-11 行代码
2.  9                 if color[left[w]] = BLACK and color[right[w]] = BLACK
3.  10                    then color[w] ← RED                              ▸  Case 2
4.  11                         x p[x]                                      ▹  Case 2
```

变化前



变化后

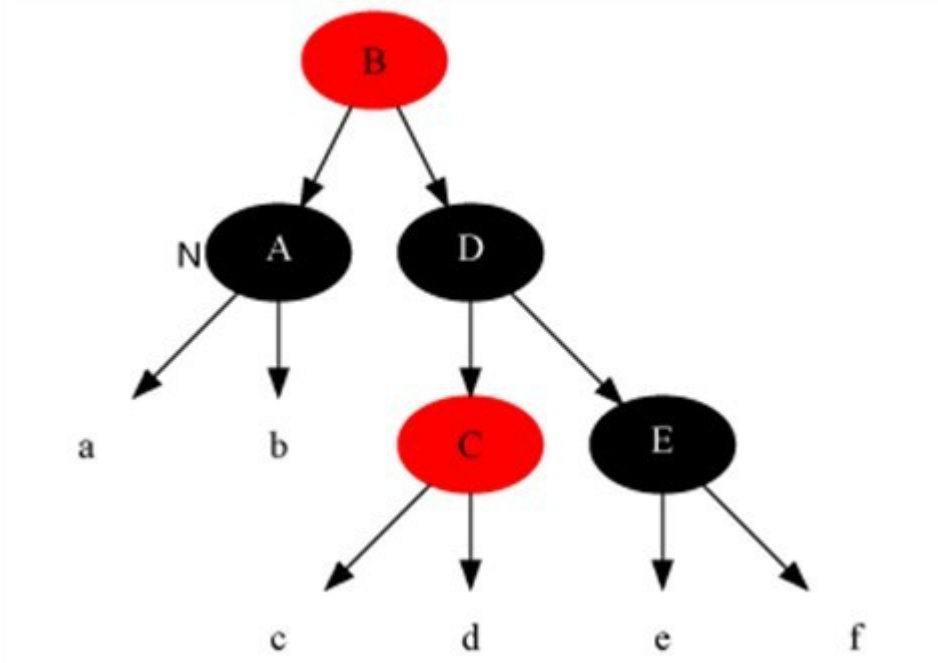


删除修复情况 3：当前节点颜色是黑+黑，兄弟节点是黑色，兄弟的左子是红色，右子是黑色。

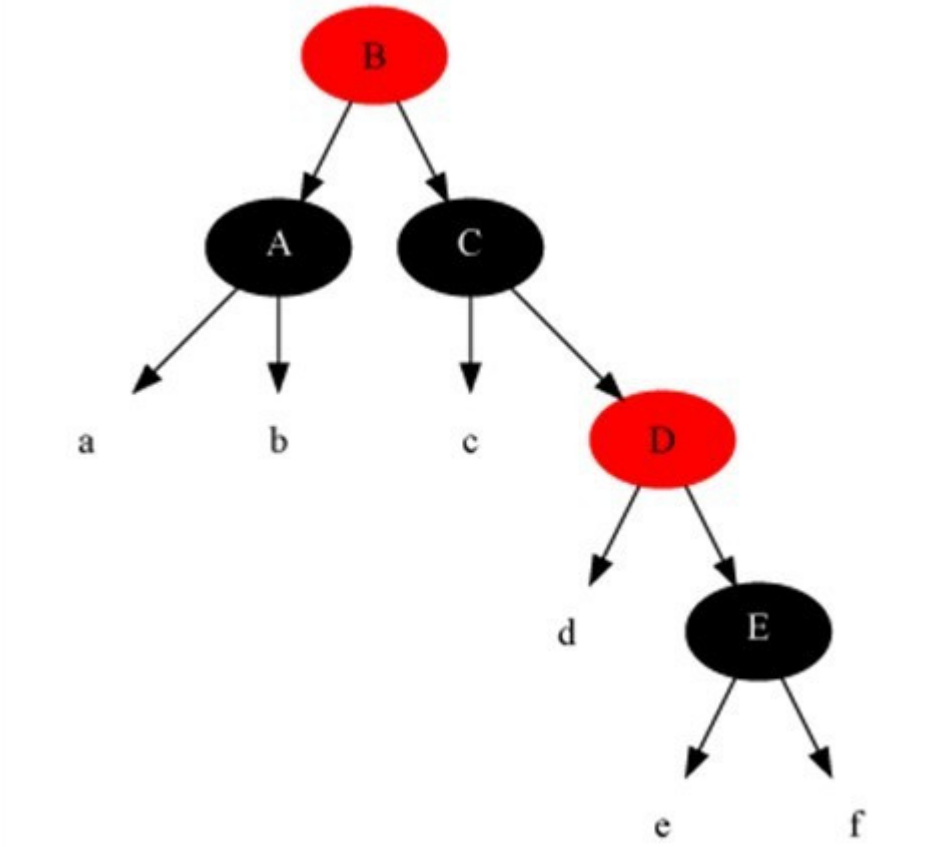
解法：把兄弟结点染红，兄弟左子节点染黑，之后再在兄弟节点为支点解右旋，之后重新进入算法。此是把当前的情况转化为情况 4，而性质 5 得以保持，即调用 RB-INSERT-FIXUP(T, z) 的第 12-16 行代码，如下所示：

```
[cpp] view plain copy
1.  //调用 RB-DELETE-FIXUP(T, x) 的第 12-16 行代码
2.  12                  else if color[right[w]] = BLACK
3.  13                          then color[left[w]] ← BLACK          ▹  Case 3
4.  14                               color[w] ← RED                  ▹  Case 3
5.  15                               RIGHT-ROTATE(T, w)              ▹  Case 3
6.  16                               w ← right[p[x]]                 ▹  Case 3
```

变化前：



变化后：

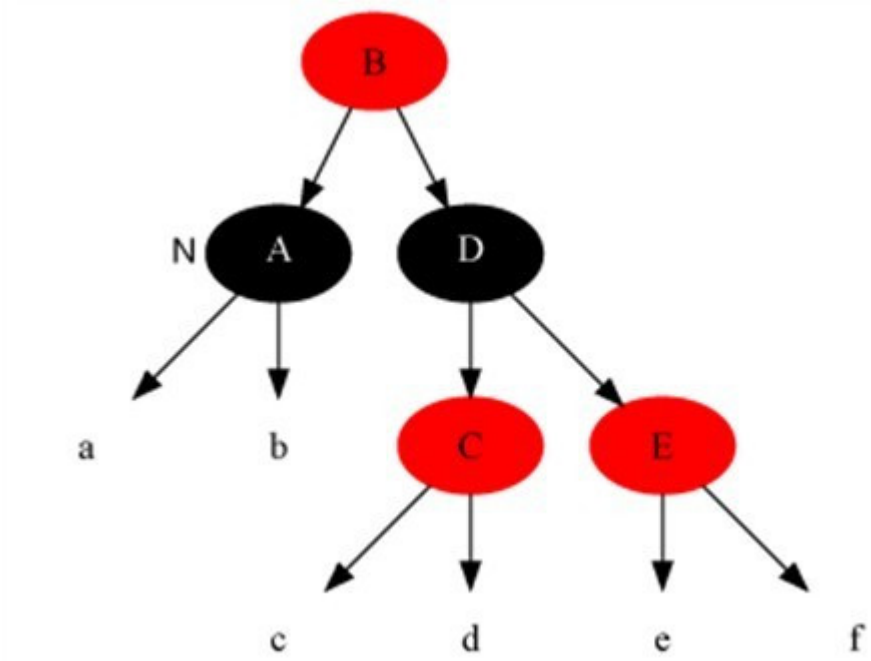


删除修复情况 4：当前节点颜色是黑-黑色，它的兄弟节点是黑色，但是兄弟节点的右子是红色，兄弟节点左子的颜色任意。

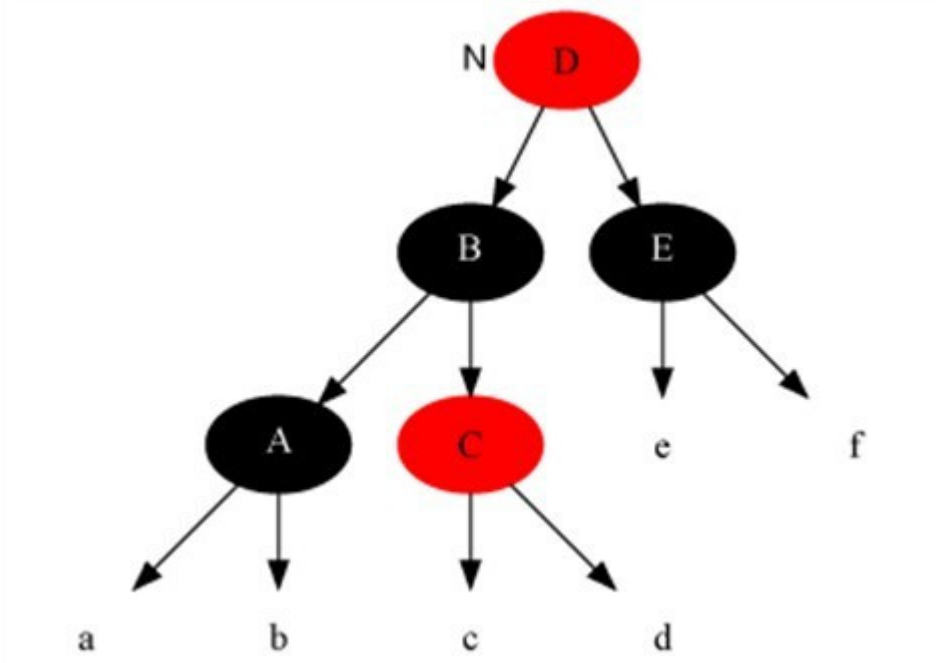
解法：把兄弟节点染成当前节点父节点的颜色，把当前节点父节点染成黑色，兄弟节点右子染成黑色，之后以当前节点的父节点为支点进行左旋，此时算法结束，红黑树所有性质调整正确，即调用 RB-INSERT-FIXUP(T, z)的第 17-21 行代码，如下所示：

```cpp
//调用 RB-DELETE-FIXUP(T, x) 的第 17-21 行代码
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
```

变化前：



变化后：



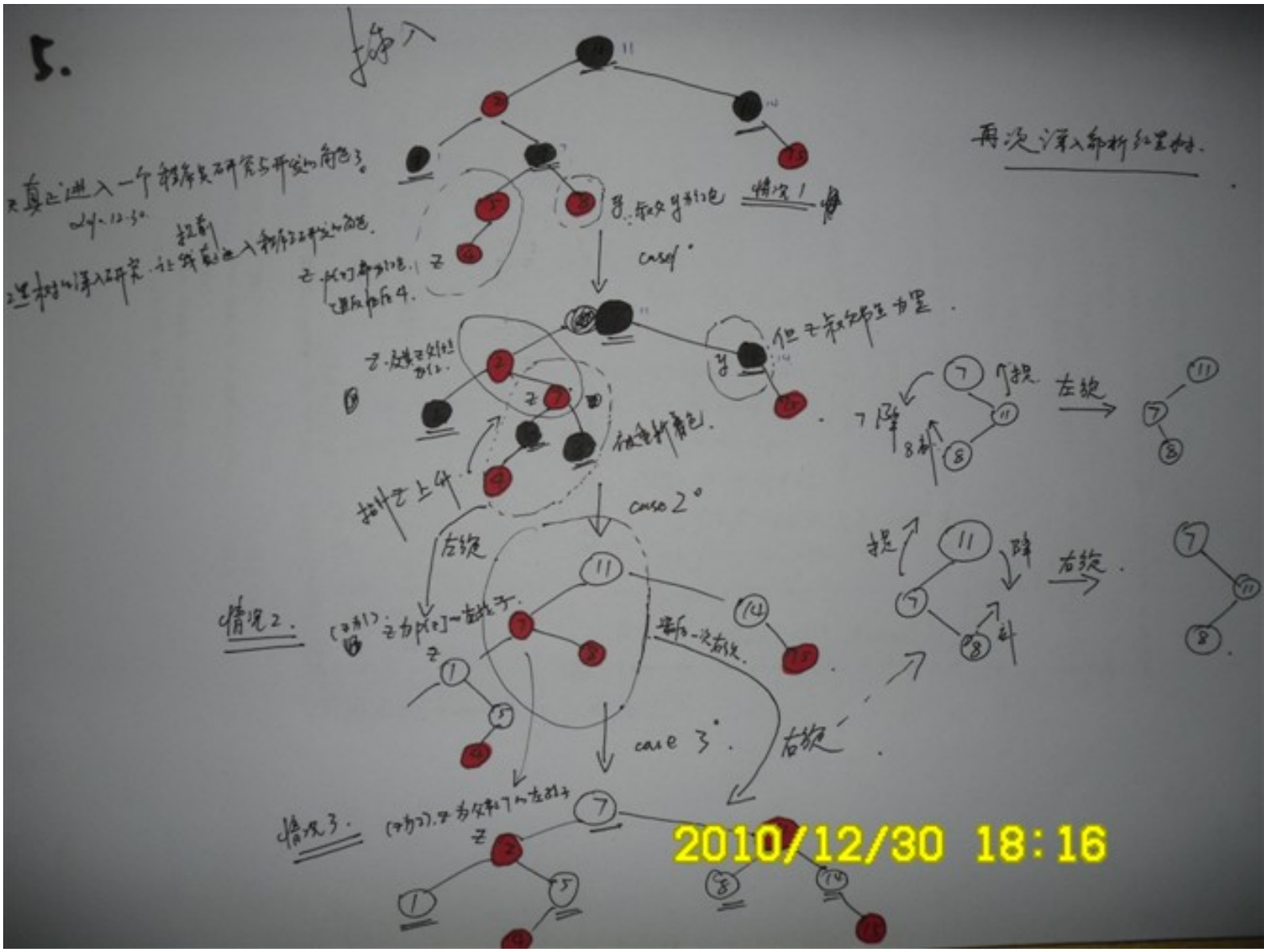
最后值得一提的是**上述删除修复的情况 1~4 都只是树的局部，并非树的整体全部，且删除修复情况 3、4 在经过上面的调整后，调整还没结束**（还得继续调整直至重新恢复平衡，只是图并没有画出来）。

后面会继续修改完善下本文，感谢关注，thanks。

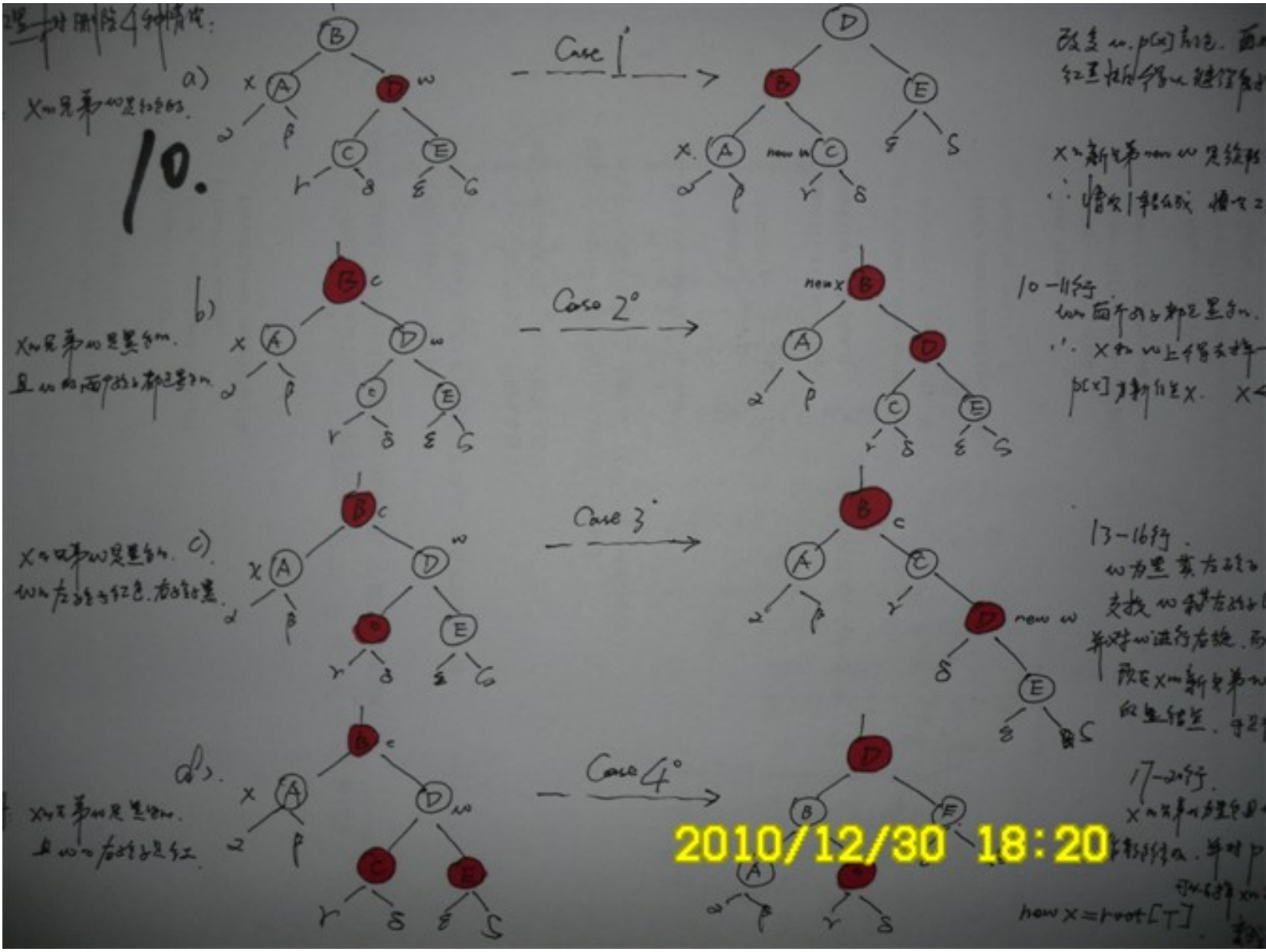
July、二零一四年九月十五日修订。

之前在学校寝室画红黑树画了好几个钟头，贴俩张图：

红黑树插入修复的 3 种情况：



红黑树删除修复的 4 种情况：



---

updated

继续请看本文的 github 优化版本：https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md，或看看这个 PPT：http://vdisk.weibo.com/s/zrFL6OXJNfNVU。另，对应新书《编程之法：面试和算法心得》3.1 节。July、二零一四年五月三日