从 2-3-4 树谈到红黑树

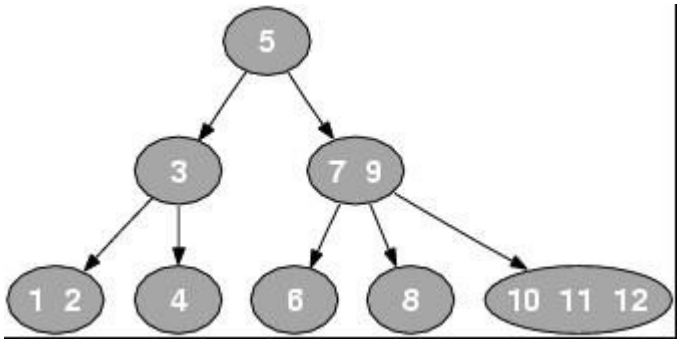
在上一篇文章--从 B 树、B+树、B*树谈到 R 树里已提到 2-3-4 树，那么本文，咱们就从 2-3-4 树开始谈起，然后谈至红黑树。因为理解了 2-3-4 树，红黑树也就没有任何问题了。同时，虽然红黑树在本 blog 已有过非常详尽的阐述。但个人此后对红黑树又有了不少新的认识，雨打风吹去，已体味到另一番意境。

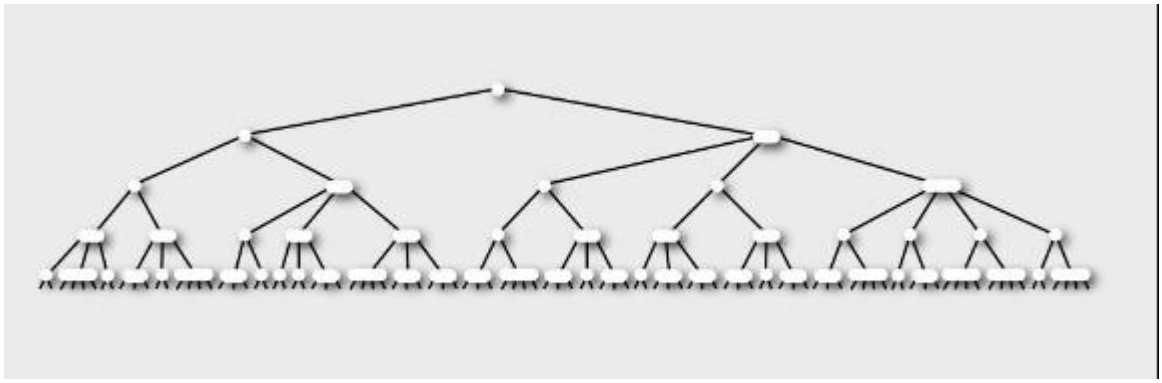
0k，本文大部分内容翻译自此文档：***Left-Leaning Red-Black Trees***, Dagstuhl Workshop on Data Structures, Wadern, Germany, February, 2008. 这个文档本人在之前介绍红黑树的文章里早已推荐过。但我相信，如果不真正完全摆在读者面前，他们是不能最大限度的体会到一个东西的价值与精彩的。

So，旅程开始，祝旅途愉快。

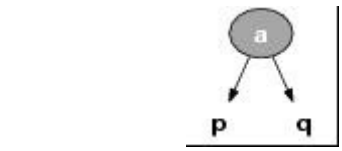
第一节、2-3-4 树

2-3-4 树在计算机科学中是阶为 4 的 B 树。根据维基百科上的介绍：大体上同 B 树一样，2-3-4 树是可以用做字典的一种自平衡数据结构。它可以在 O（log n）时间内查找、插入和删除，这里的 n 是树中元素的数目。2-3-4 树在多数编程语言中实现起来相对困难，因为在树上的操作涉及大量的特殊情况。红黑树实现起来更简单一些，所以可以用它来替代（红黑树稍后介绍）。以下就是一棵 2-3-4 树：

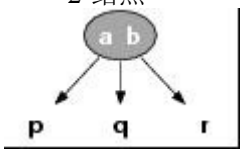




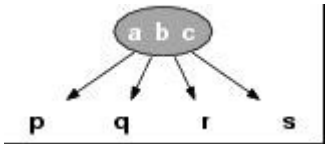
2-3-4 树把数据存储在叫做元素的单独单元中。那么请问，到底什么是 2-3-4 树呢?顾名思义，就是有 2 个子女，3 个子女，或 4 个子女的结点，这些含有 2、3、或 4 个子女的结点就构成了我们的 2-3-4 树。所以，它们组合成结点，每个结点都是下列之一：
2-节点，就是说，它包含 1 个元素和 2 个儿子，
3-节点，就是说，它包含 2 个元素和 3 个儿子，
4-节点，就是说，它包含 3 个元素和 4 个儿子 。



2-结点

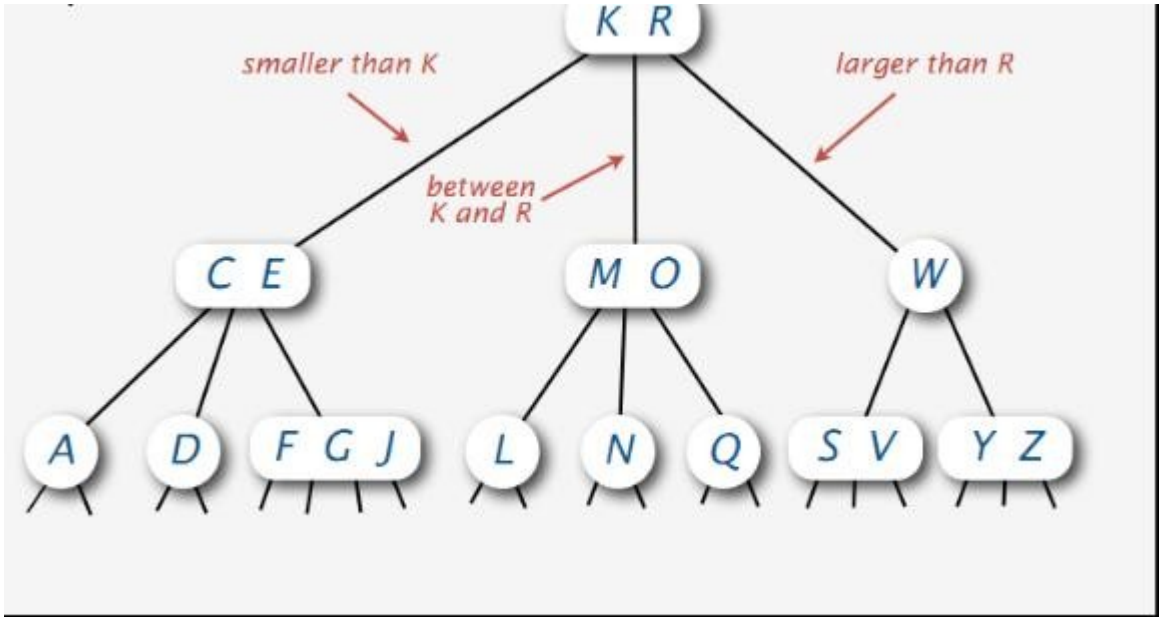


3-结点



4-结点

每个儿子都是（可能为空）一个子 2-3-4 树。根节点是其中没有父亲的那个节点；它在遍历树的时候充当起点，因为从它可以到达所有的其他节点。叶子节 点是有至少一个空儿子的节点。同 B 树一样，2-3-4 树是有序的：每个元素必须大于或等于它左边的和它的左子树中的任何其他元素。每个儿子因此成为了由它的左和右元素界定的一个区间。如下图所示（你可以看到，图中这棵 2-3-4 树是由 2-结点，3-结点，4-结点元素组成的）：



2-3-4 树是紅黑树结构的一种等同，这意味着它们是等价的数据结构。换句话说，对于每个 2-3-4 树，都存在着至少一个数据元素是相同次序的红黑树。在 2-3-4 树上的插入和删除操作也等价于在红黑树中的颜色翻转和旋转。这使得它成为理解红黑树背后的逻辑的重要工具（还是如此，红黑树稍后介绍，路得一步一步来）。

1.1、2-3-4 树的查找

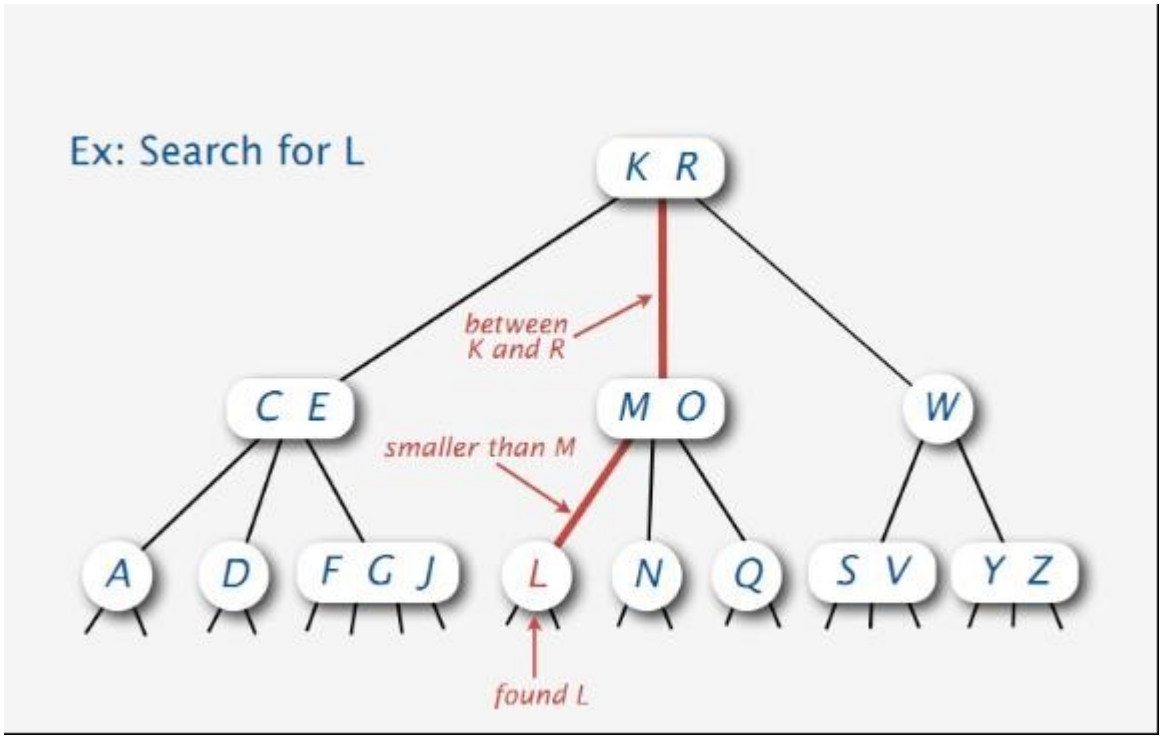
2-3-4 树中查找结点，怎么查找呢?分为以下几个步骤：

1、把要查找的结点与根结点相比较

2、根据左小右大的原则，寻找含有要查找结点的区间

3、若找到了，则直接返回该结点，否则，在其子女中继续递归寻找。

如下图所示，在下面这棵 2-3-4 树中寻找 L 结点：

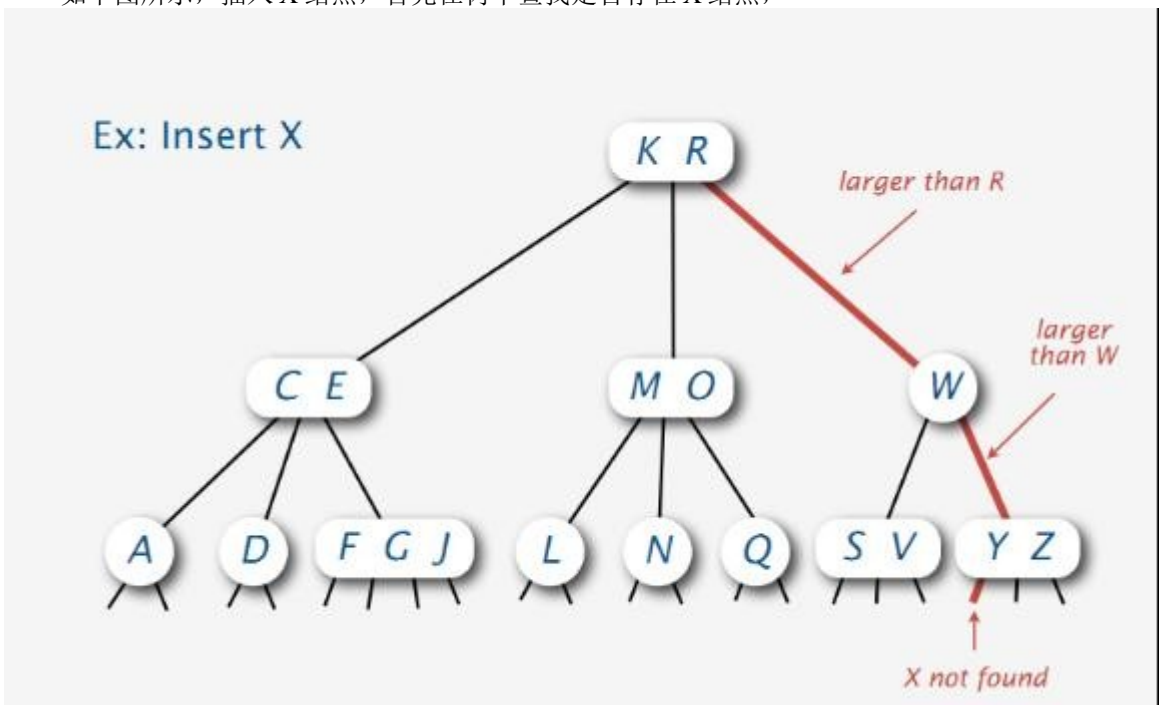


1.2、2-3-4 树的插入

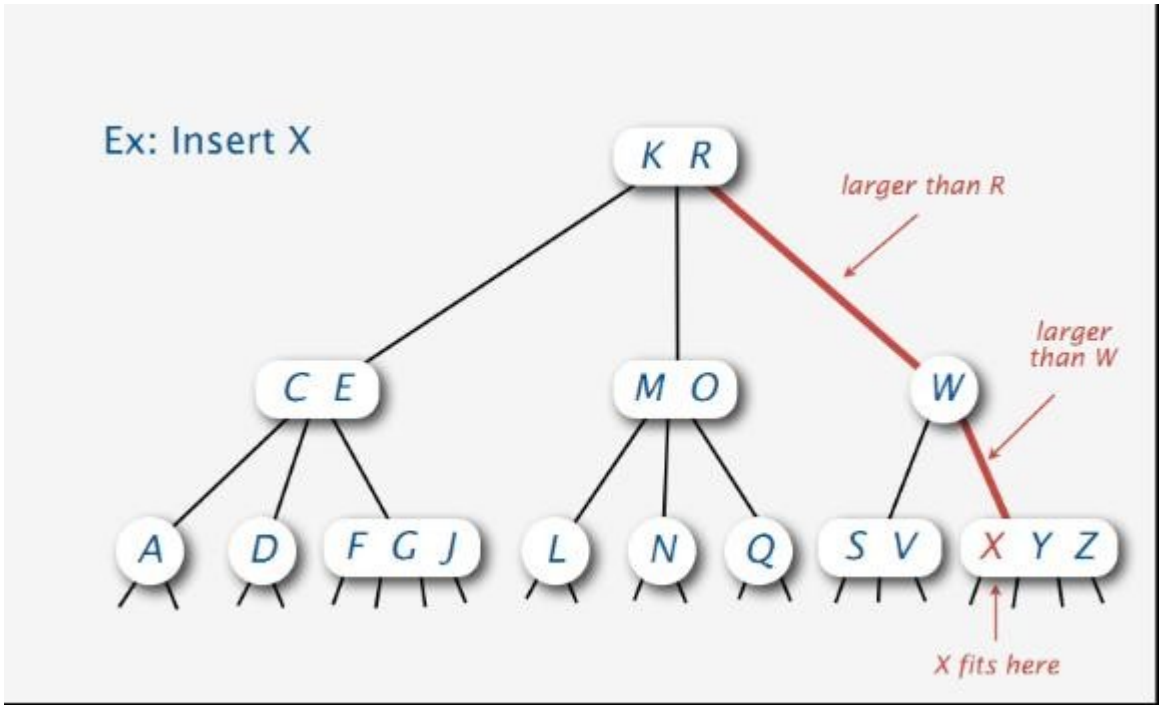
插入某个结点之前，一般我们先在 2-3-4 树中寻找是否存在该插入结点（若存在，当然也就没有必要再插入了），如果树中不存在该结点，则执行插入操作。

1.2.1、插入形式一：3-结点元素中插入结点

如下图所示，插入 X 结点，首先在树中查找是否存在 X 结点，

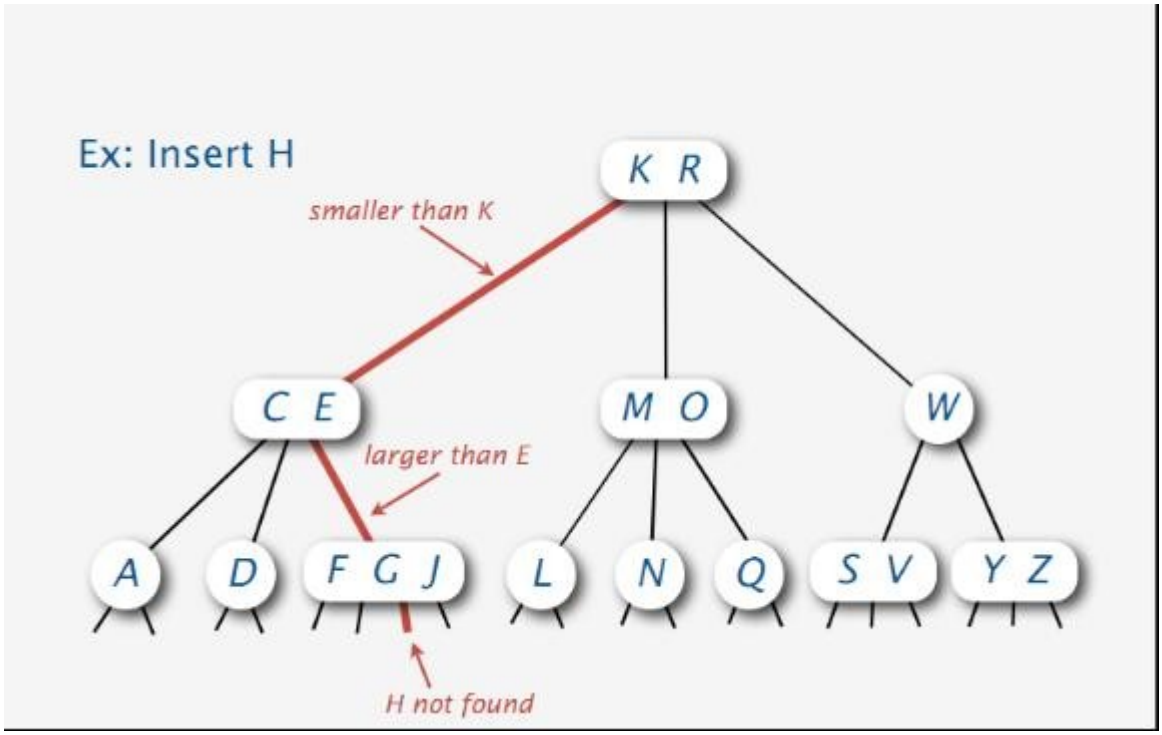


没有找到，则在含有 Y Z 的结点元素插入 X：

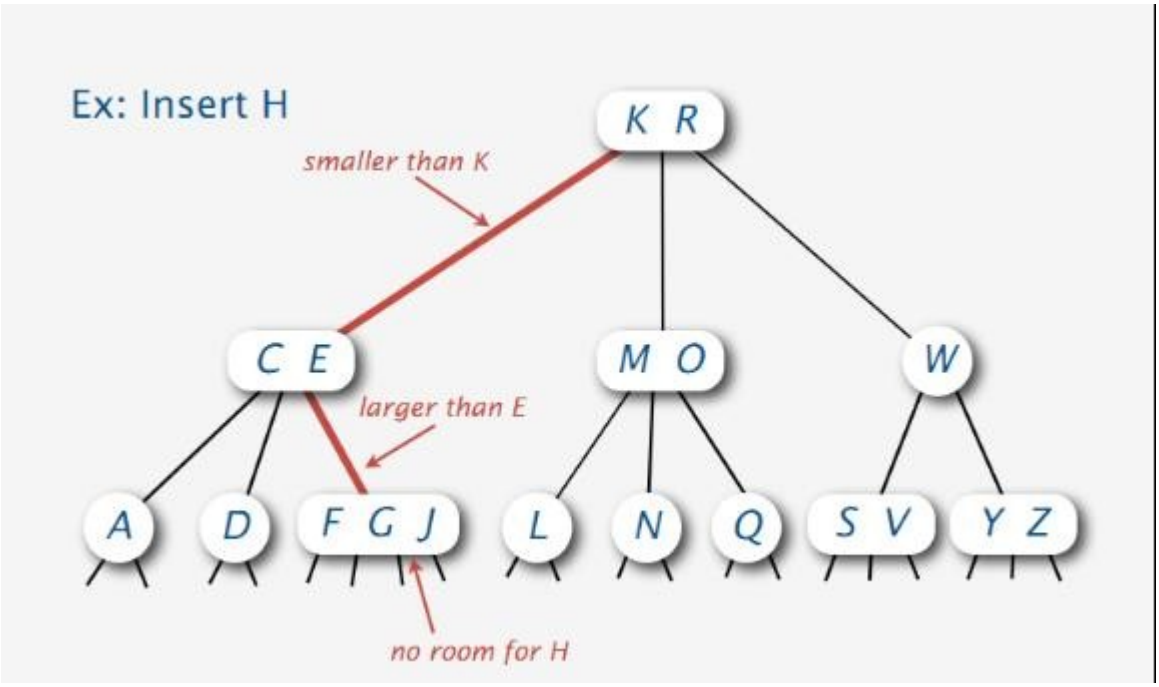


1.2.2、插入形式二：

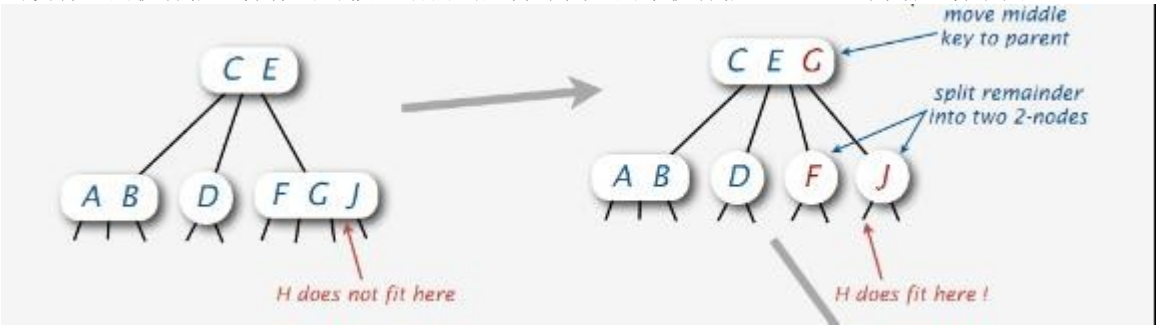
以下插入 H 结点，在 F G J 区间上发现 H 没有找到后，



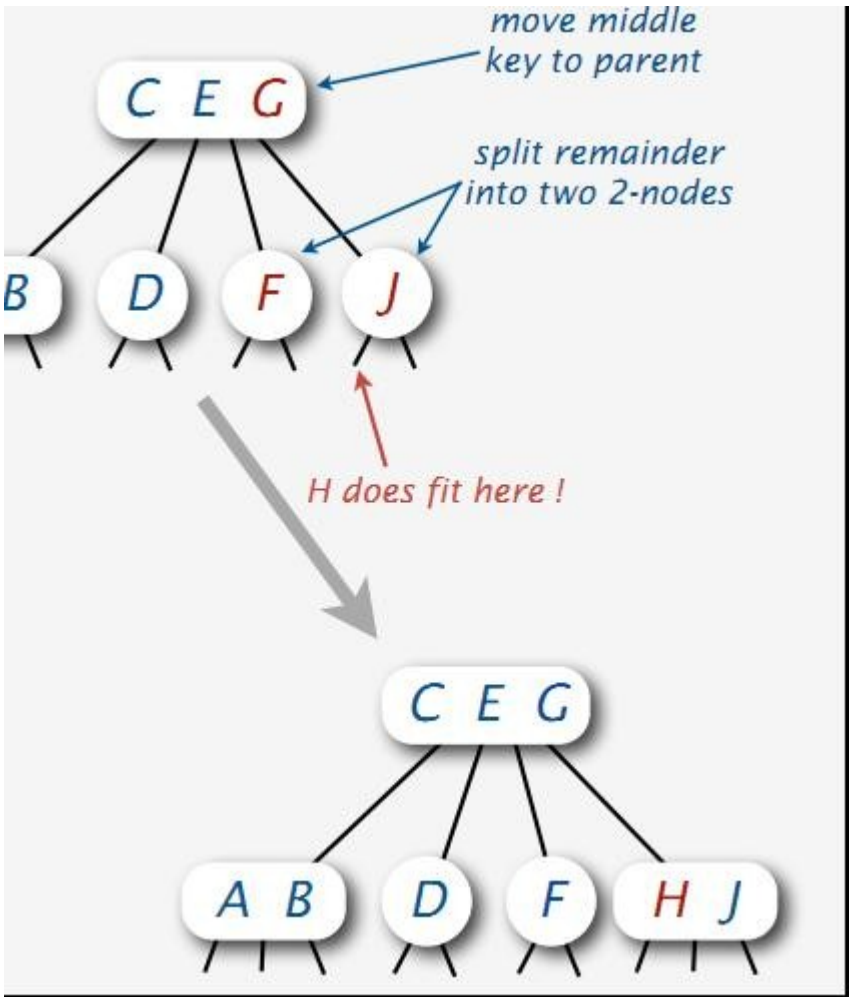
且 F G J 区间上已经没有空间来插入 H 结点了，这个时候怎么办呢?：



当没有空间执行插入操作的时候，咱们当然得寻找空间来执行插入。怎么寻找呢?看下图：



由上图，我们发现，H 在区间 F G J 上没有空间执行插入的时候，我们首先参试着让 G 元素上移至 C E 区间，组成 C E G 根结点，然后分裂 F G 区间，F 和 G 各自成为独立的元素。当我们发现，H 依然不能作为 J 的子女进行插入时，我们想到了一种折中的办法，这种办法就是，如下图所示，H 插入到 J 元素旁，成为 H J 区间，F 元素不作变动：

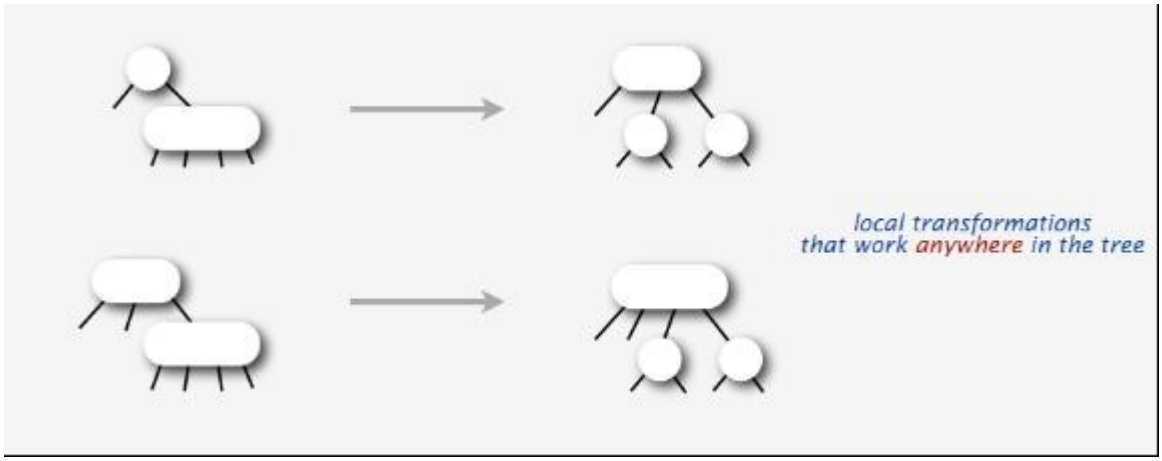


所以，当发现没有空间可执行插入结点的情况时，我们作如下对策：

1、分裂父母结点

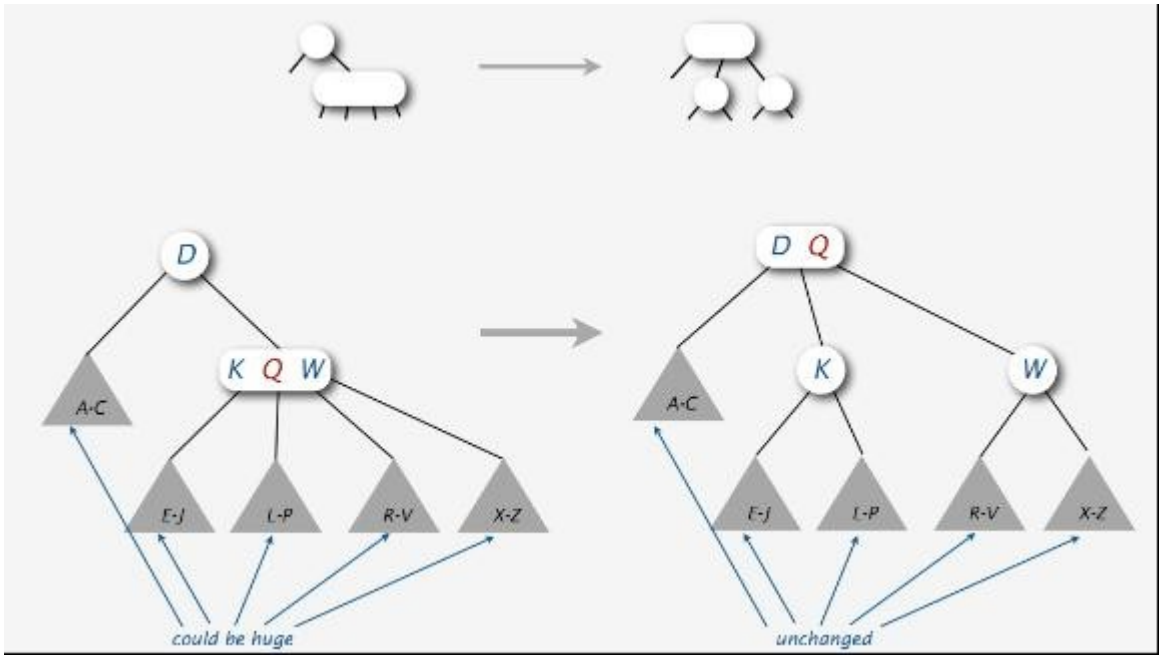
2、然后再插入结点

上述的第 1 点具体怎么分裂呢?分裂的两种情况如下图所示：

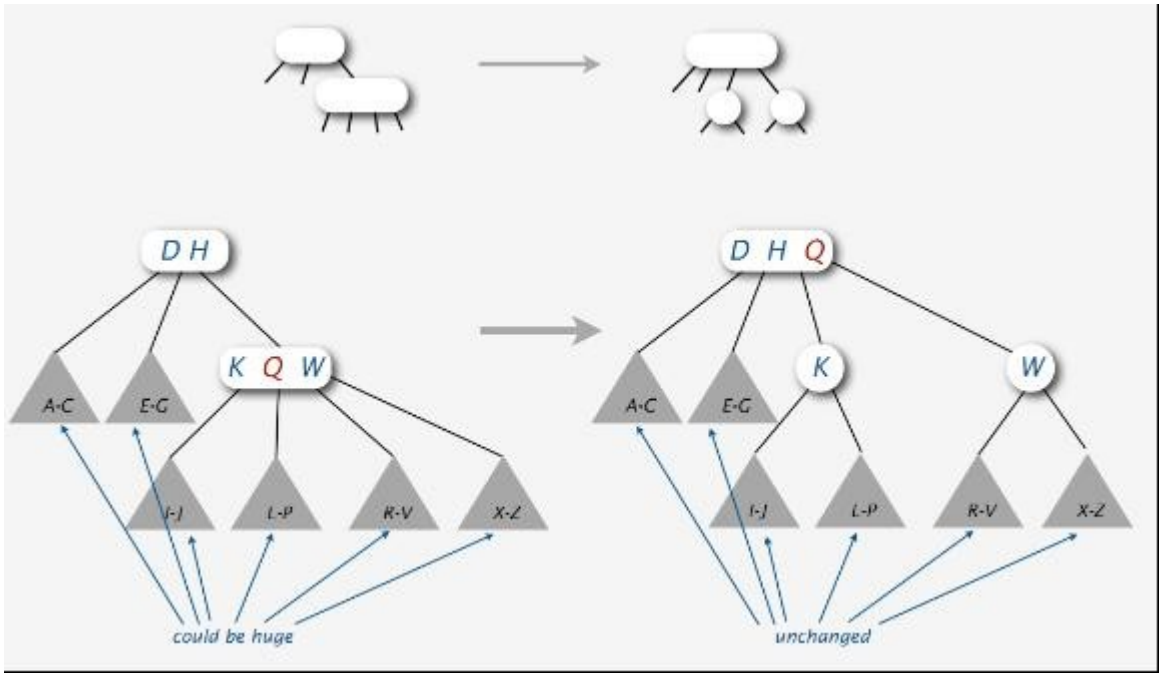


下面再具体分析下上述两种分裂情况：

分裂情况 1、如下图所示 D-K Q W，区间 K Q W 分裂，Q 上移与原根结点 D 组成新的根结点区间 D Q，K 和 W 各自分裂成独自区间，D-K Q W 最终分裂成 D Q-K-W：



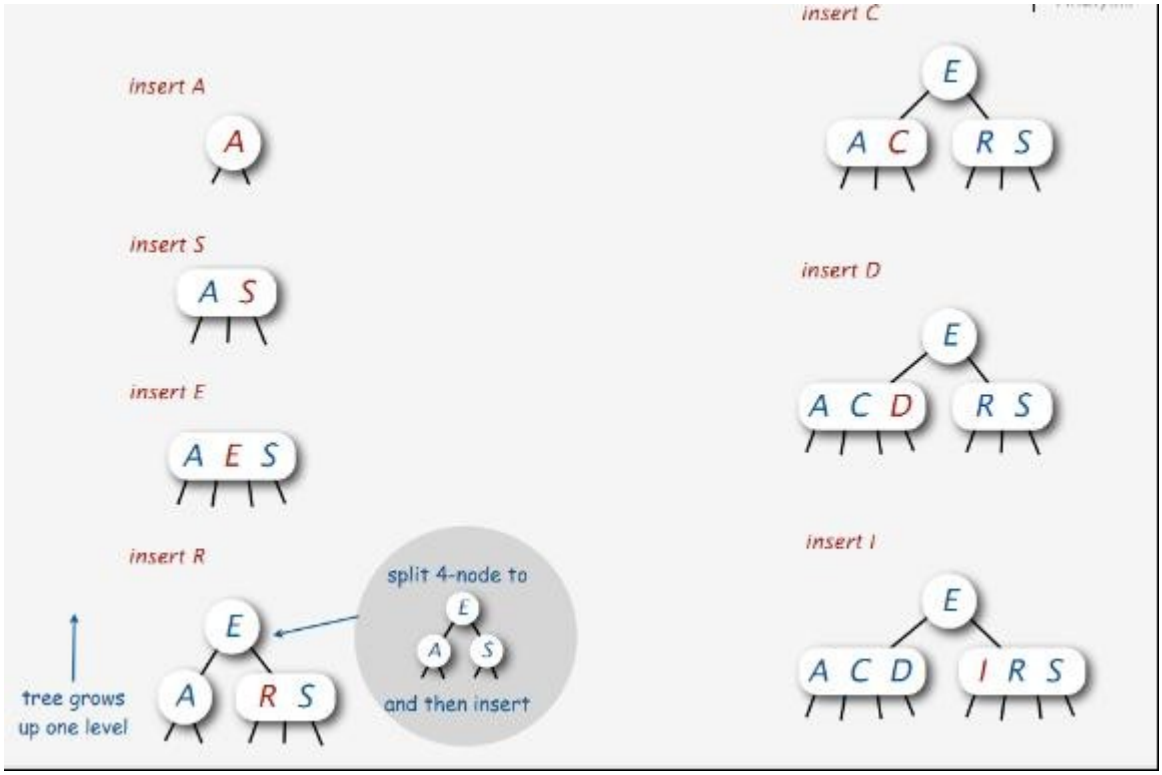
分裂情况 2、如下图所示 D H-K Q W 分裂，K Q W 区间中的 Q 元素上移与 D H 组成根元素区间 D H Q，K 和 W 分裂，最终 D H-K Q W 分裂成 D H Q-K-W：



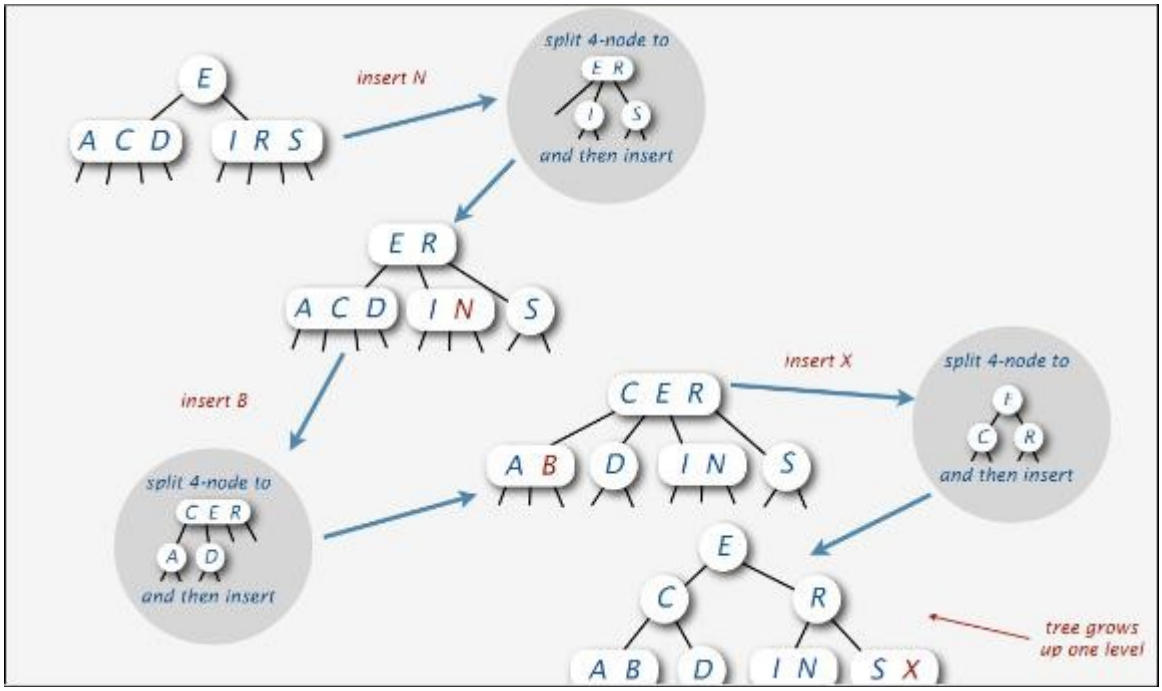
下图是逐步在 2-3-4 树中依次插入元素 A、S、E、R、C、D、I。从中你可以看到

1、当插入元素 R 时，A E S 区间分裂，E 成为新的根元素，而要插入的 R 与 S 移至一起成为 E 的右儿子。

2、当插入 C、D、I 时，都是直接找到相对应的区间，各自插入。不必啰嗦，下图已经很形象了。



但下面，继续在上述的 A、S、E、R、C、D、I 插入操作的基础上之后，再依次插入 N、B、X 各元素时，情况，就比较复杂了，但下图还是很清晰的表明了各种插入操作及相关元素的调整情况，在此不赘述：



下图所示的是在 2-3-4 树中插入结点的 insert 代码：

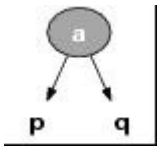
```
private void insert(Key key, Val val)        fantasy code
{
   Node x = root;
   while (x.getTheCorrectChild(key) != null)
   {
      x = x.getTheCorrectChild(key);
      if (x.is4Node()) x.split();
   }
   if      (x.is2Node()) x.make3Node(key, val);
   else if (x.is3Node()) x.make4Node(key, val);
   return x;
}
```

到此，插入情况已经阐述完，删除情况略过。接下来，咱们来分析下 2-3-4 树的平衡情况。

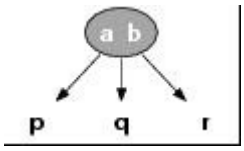
1. 3、2-3-4 树的平衡

2-3-4 树的高度为：

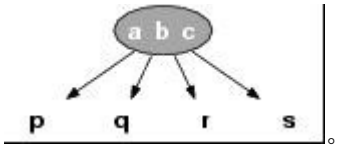
1、最坏情况，logn，树中全部都是 2 结点元素（即如第一节所述的全都是包含一个元素和 2 个儿子的结点。如此图所示，树中全部都是这样的结点：



2、最好情况，log4N=1/2logN，树中全部都是 4 结点元素（即如第一节所述的全都是包含 3 个元素和 4 个儿子的结点，如此图所示，树中全部都是这样的结点：



3、3 结点情况是中间情况，即包含 2 个元素和 3 个儿子，



。

第二节、Red-Black trees（红黑树）

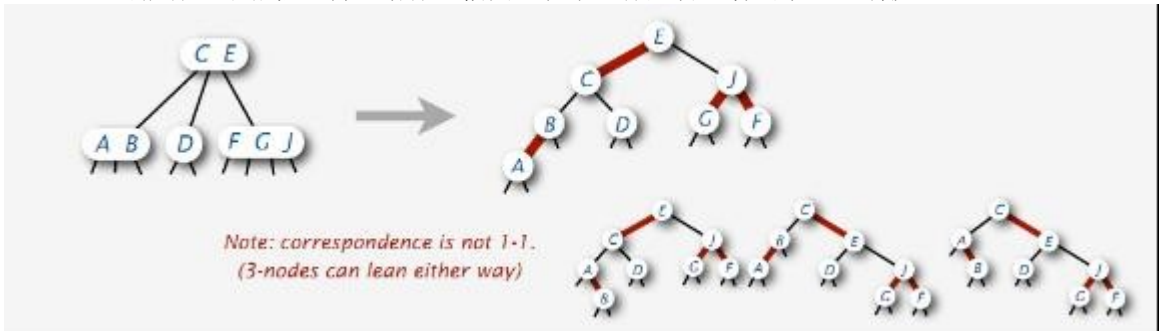
2. 1、红黑树与 2-3-4 树的相似性

上述第一节中，总是说红黑树是与 2-3-4 树等价的数据结构，下面，我来挖掘出他们的之间的相似性给你

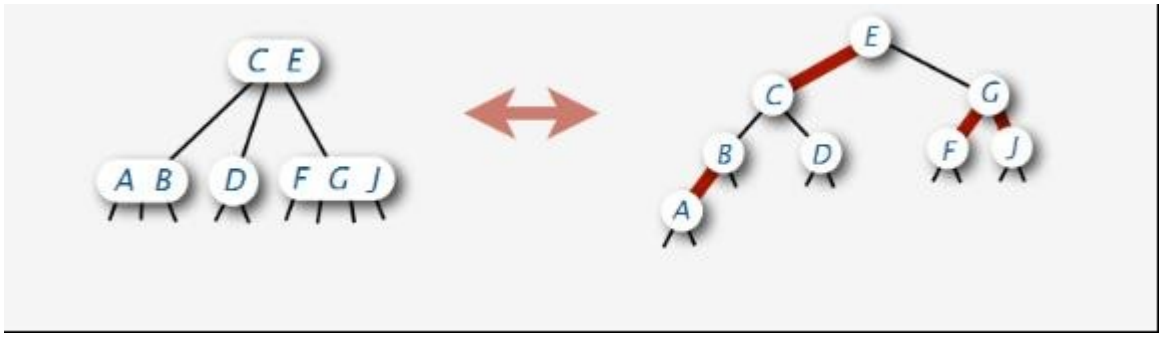
看，看看在红黑树上是否能看到 2-3-4 树的影子?ok，请看下图：

恩，yeah，不知你是否也看明白了：红黑树真的就是一种 2-3-4 树，为什么这么说呢?因为从上图中，你能轻易的看到，2-3-4 树中的 3-结点，和 4-结点分裂后，的确就是红黑树中的结点元素形式了。

0k，可能你还不相信，再送给你一幅图，从中，你是否已看出了丝毫端倪?：



在上面的图中，你可以清楚的看到图中左部分的 2-3-4 树最终能转换成一棵红黑树。不过，在上述情况下，红黑树最终将调整如下：

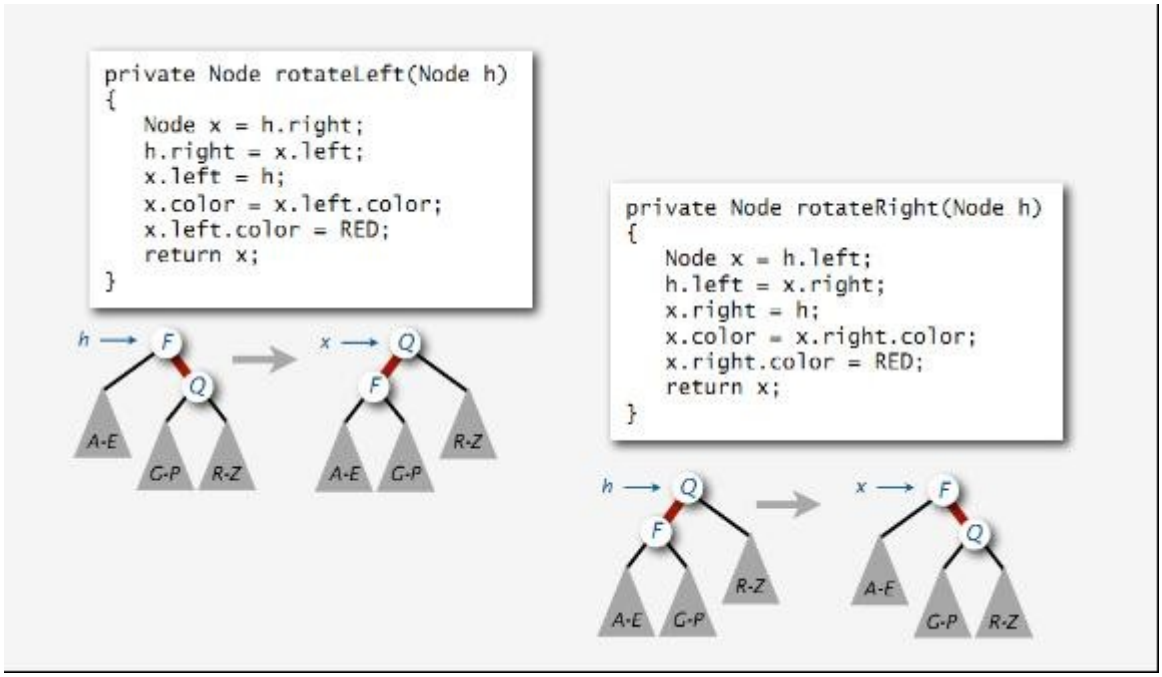


2.2、红黑树的插入操作

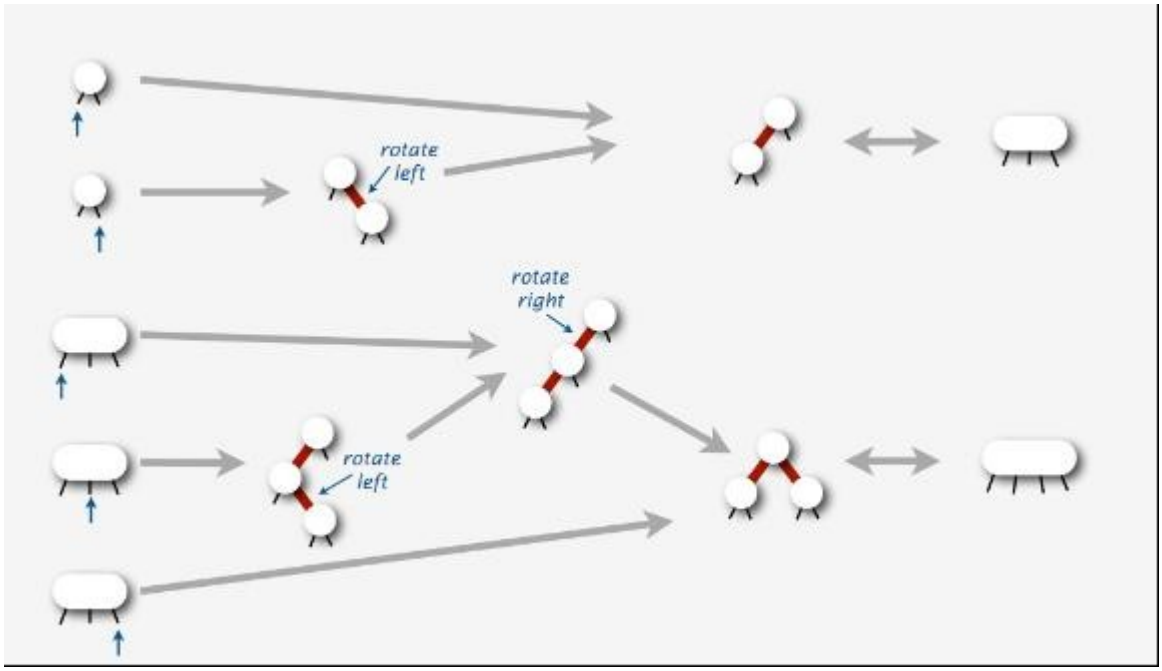
下图所示的代码是红黑树的插入操作代码：

```
private Node insert(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val);

   int cmp = key.compareTo(h.key);
   if (cmp == 0) h.val = val;   <-- associative model (no duplicate keys)
   else if (cmp < 0)
      h.left = insert(h.left, key, val);
   else
      h.right = insert(h.right, key, val);

   return h;
}
```

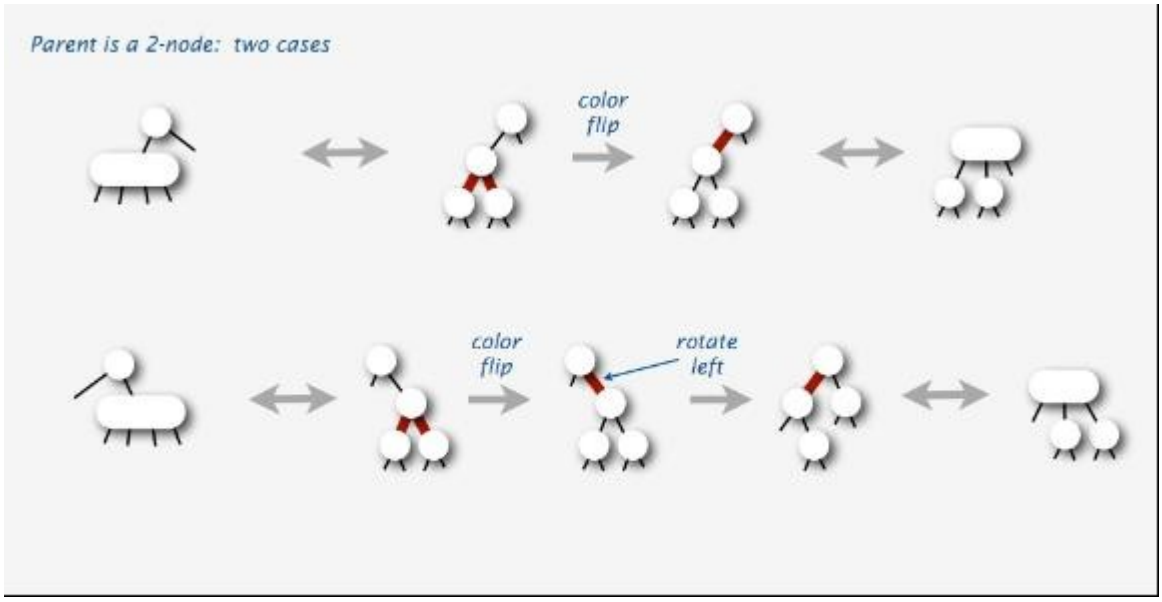
在红黑树中，有一个非常重要的基础操作，那就是左旋与右旋，在本 blog 之前的红黑树系列已经对这两种情况进行过详尽的阐述，下图是左旋和右旋各自的操作及对应的代码：



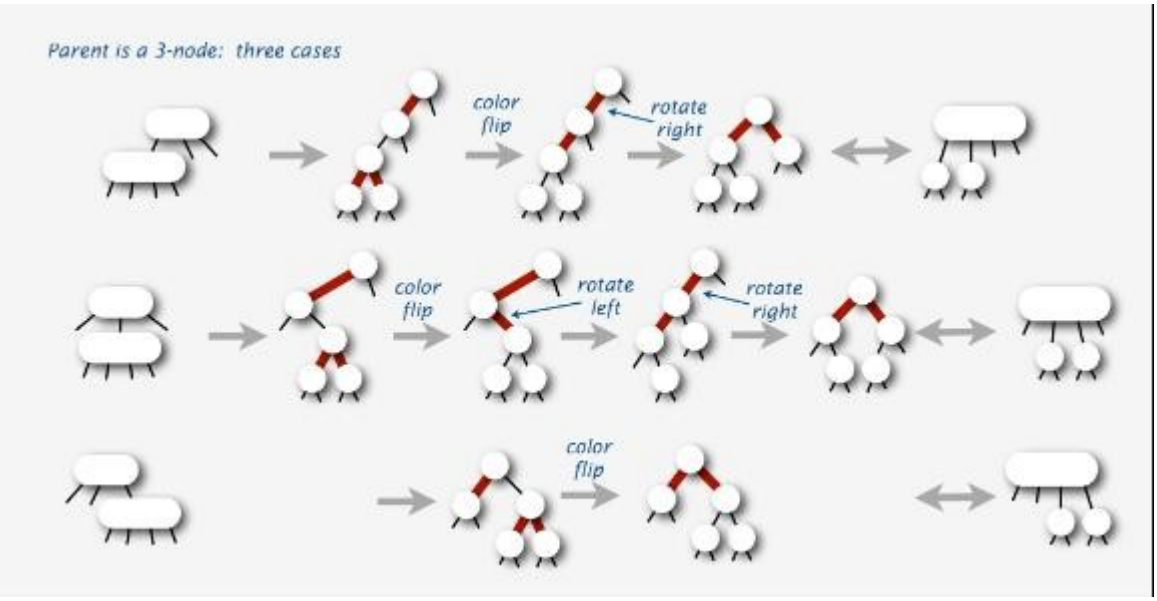
而红黑树的左旋、右旋操作情况中与 2-3-4 树对应起来的情况如下图所示：



2.2.1、红黑树的插入情况一：父母是2-结点



2.2.2、红黑树的插入情况二：父母是3-结点



其余一切类似插入操作不再阐述。关于红黑树的删除操作在本 blog 内已经有所具体的阐述，本文不再叙述。全文完。

教你透彻了解红黑树

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）。

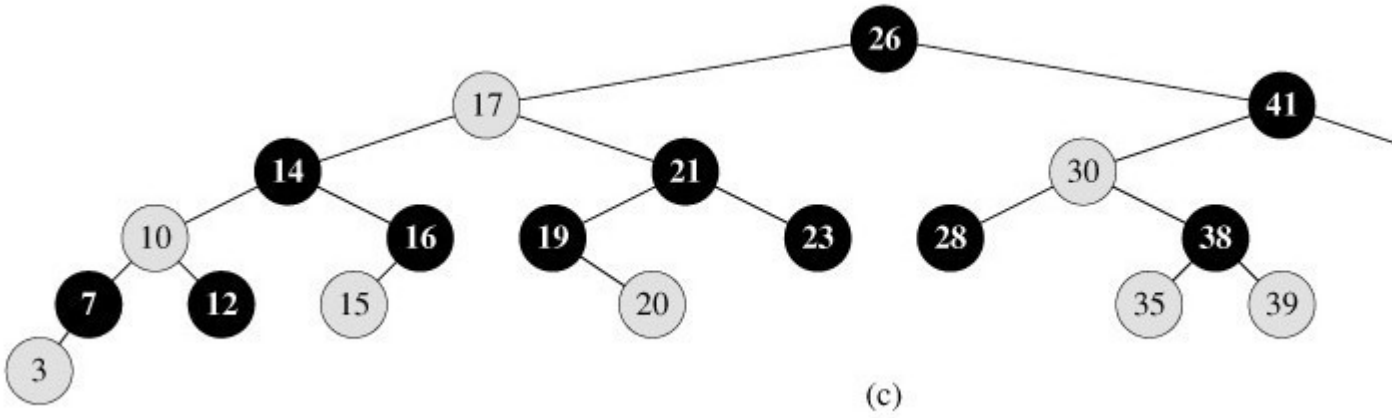
而**红黑树，能保证在最坏情况下，基本的动态几何操作的时间均为 O（lgn）。**

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

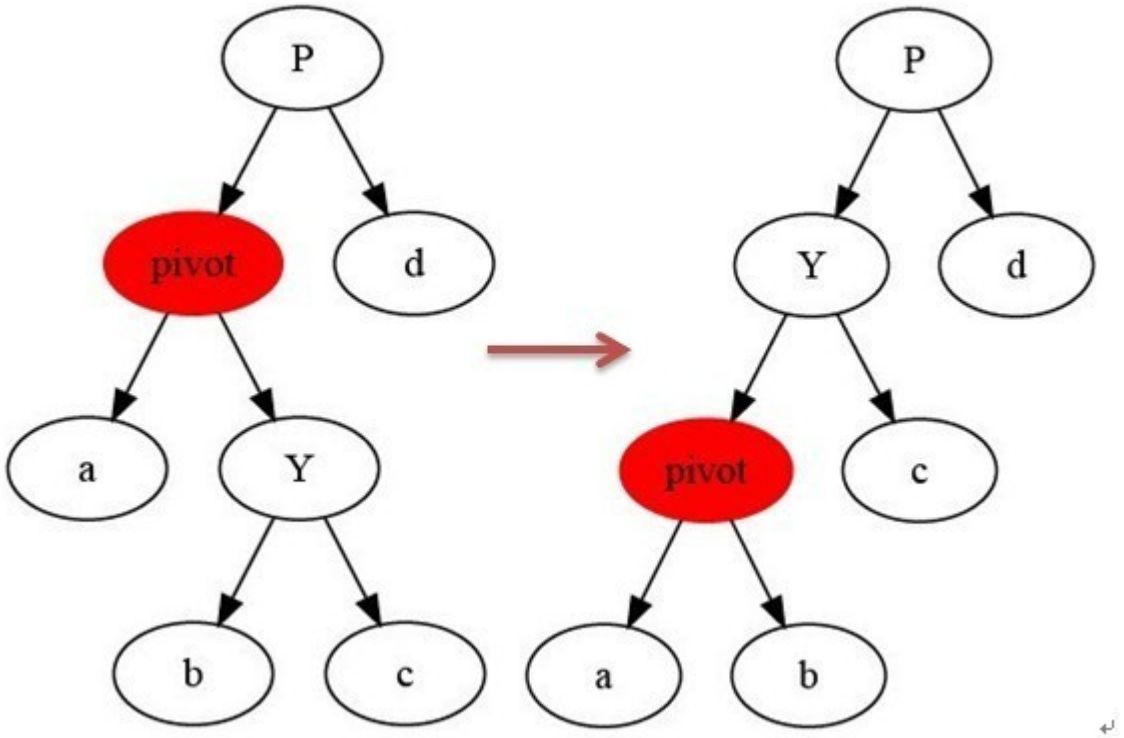
当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。

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。

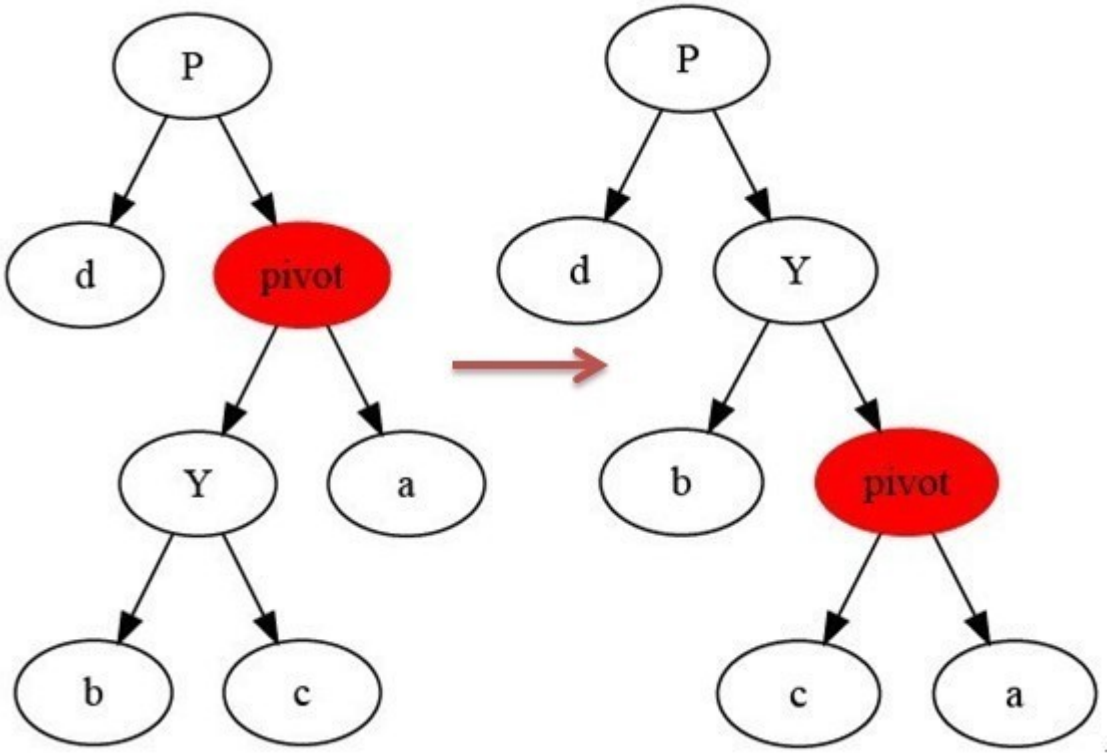
左旋以 pivot 到 y 之间的链为“支轴”进行，它使 y 成为该孩子树新的根，而 y 的左孩子 b 则成为 pivot 的右孩子。

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、I、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，

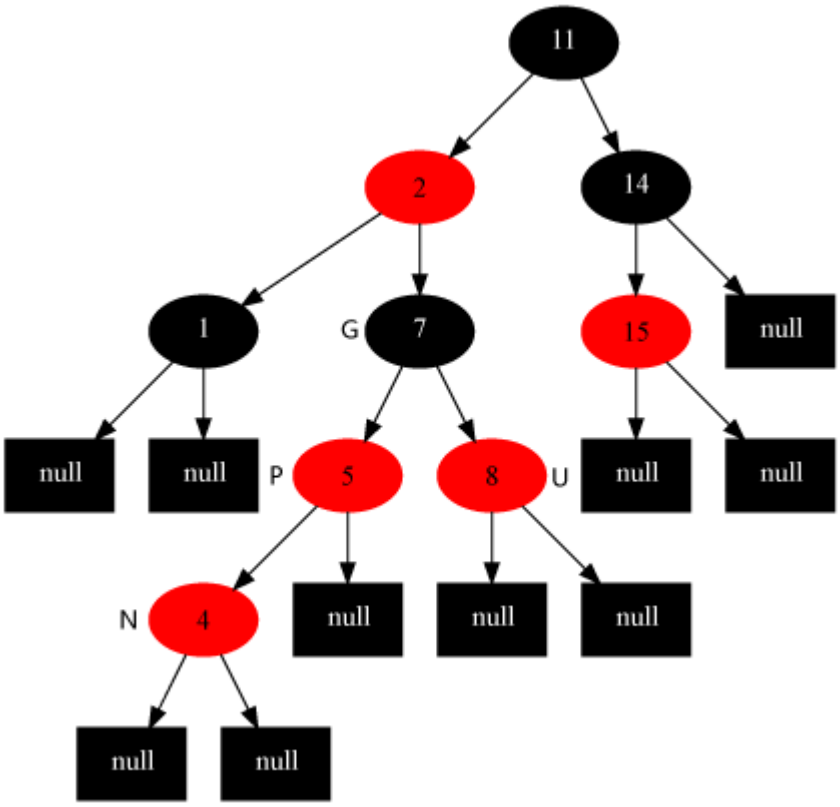
//注：以下情况 3、4、5 与上述算法导论上的代码 RB-INSERT-FIXUP(T, z)，相对应：

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。

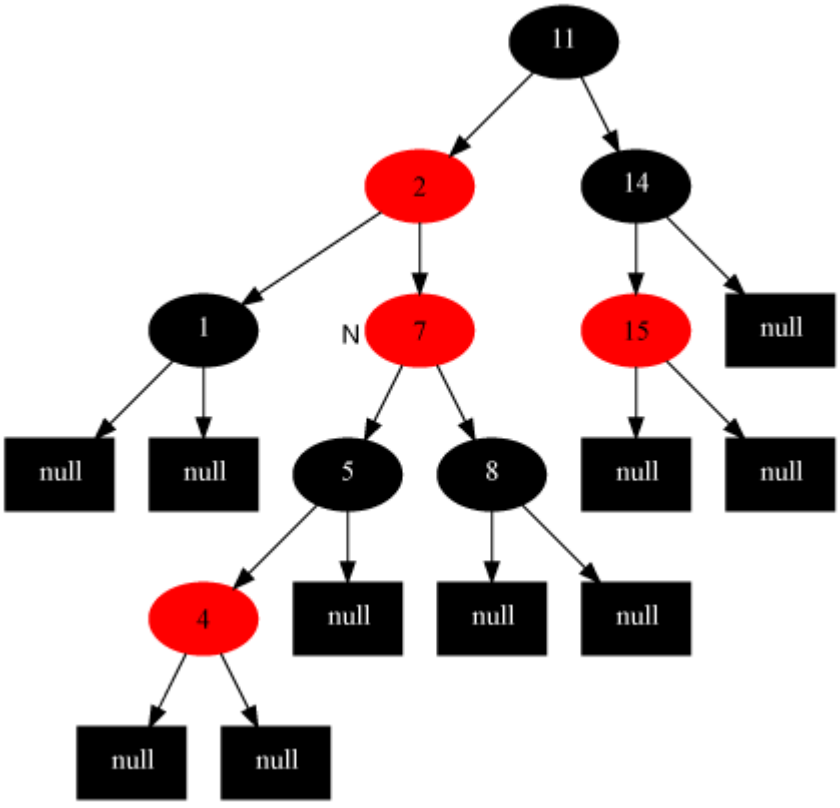
在此，我们只考虑父结点为祖父左子的情况。
同时，还可以分为当前结点是其父结点的左子还是右子，但是处理方式是一样的。我们将此归为同一类。

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]：



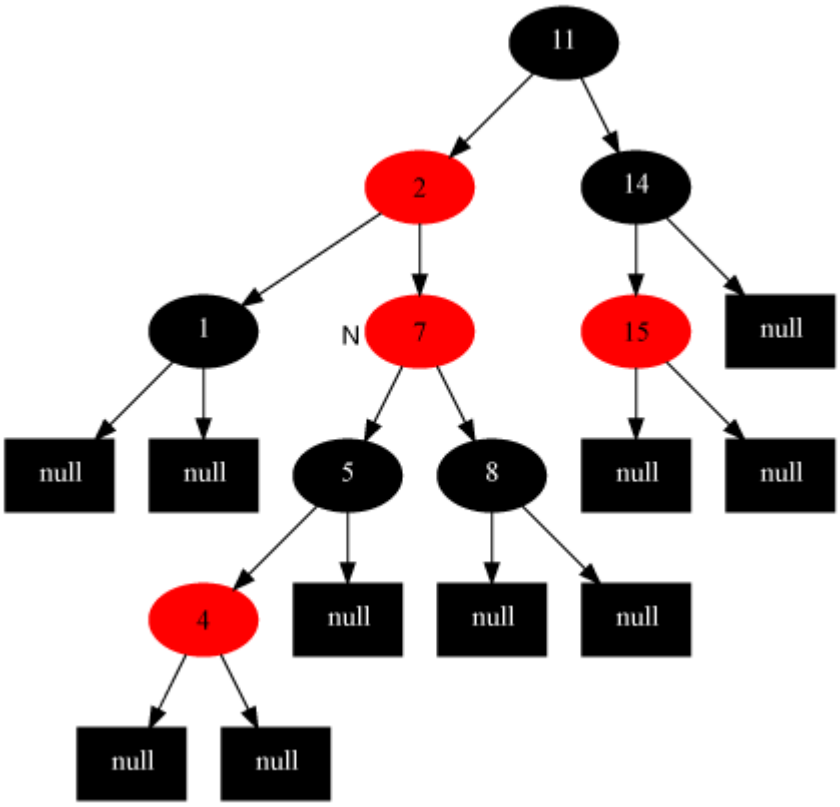
变化后：



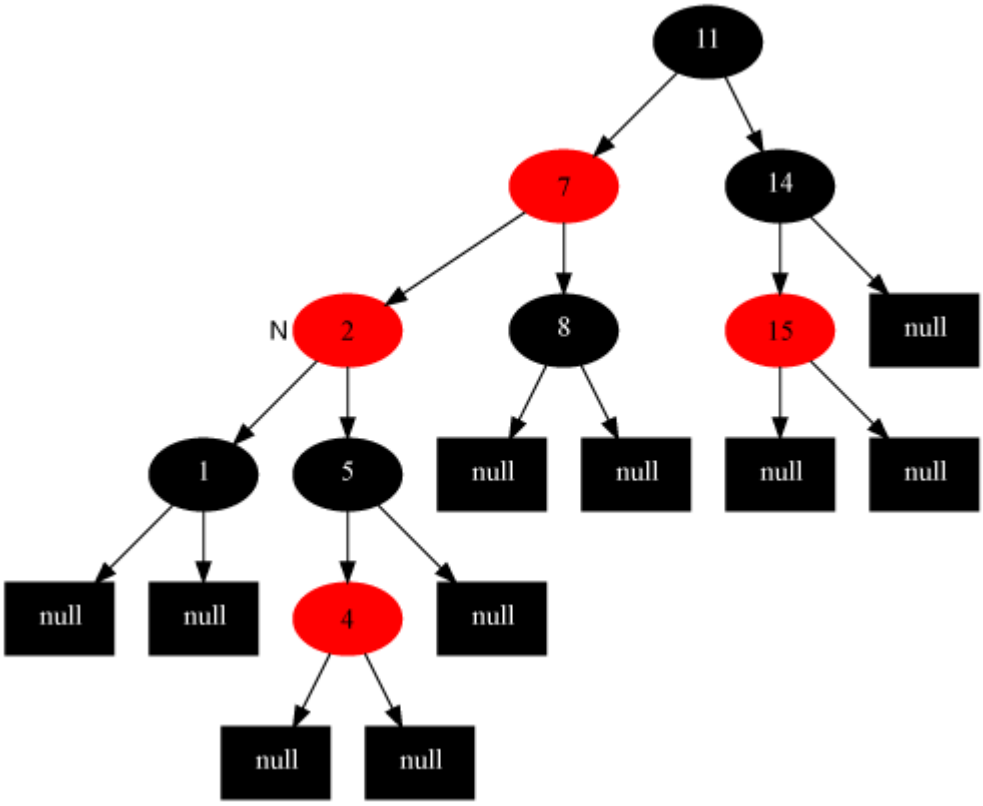
情况 4：当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的右子

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]：

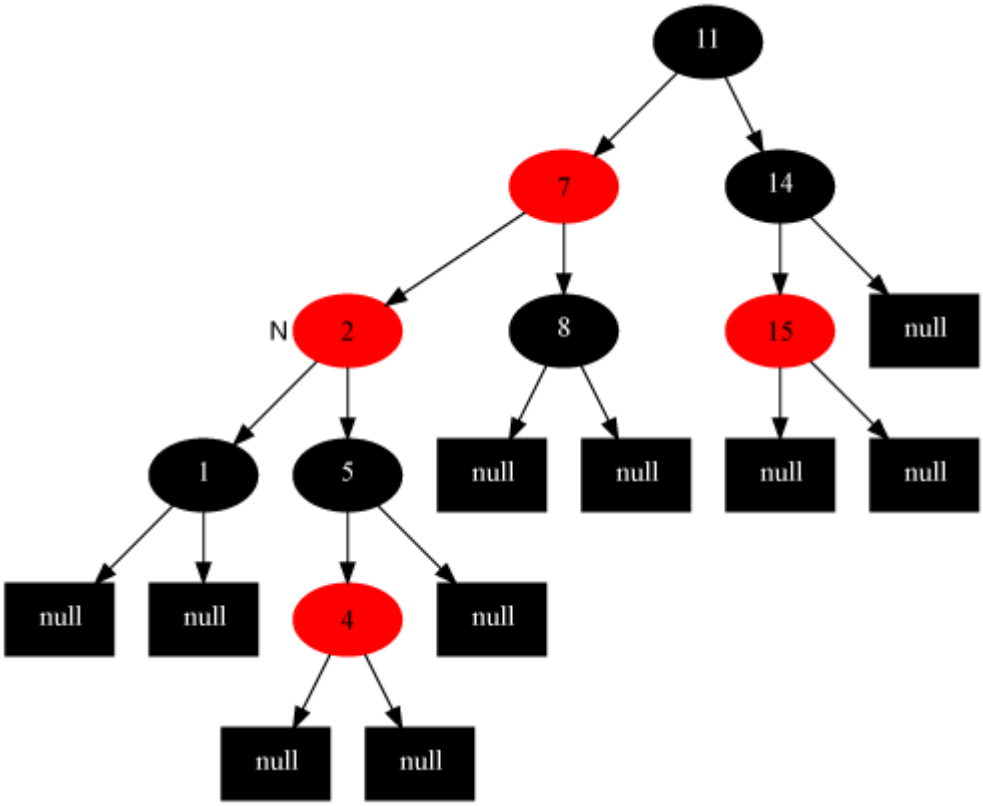


变化后：

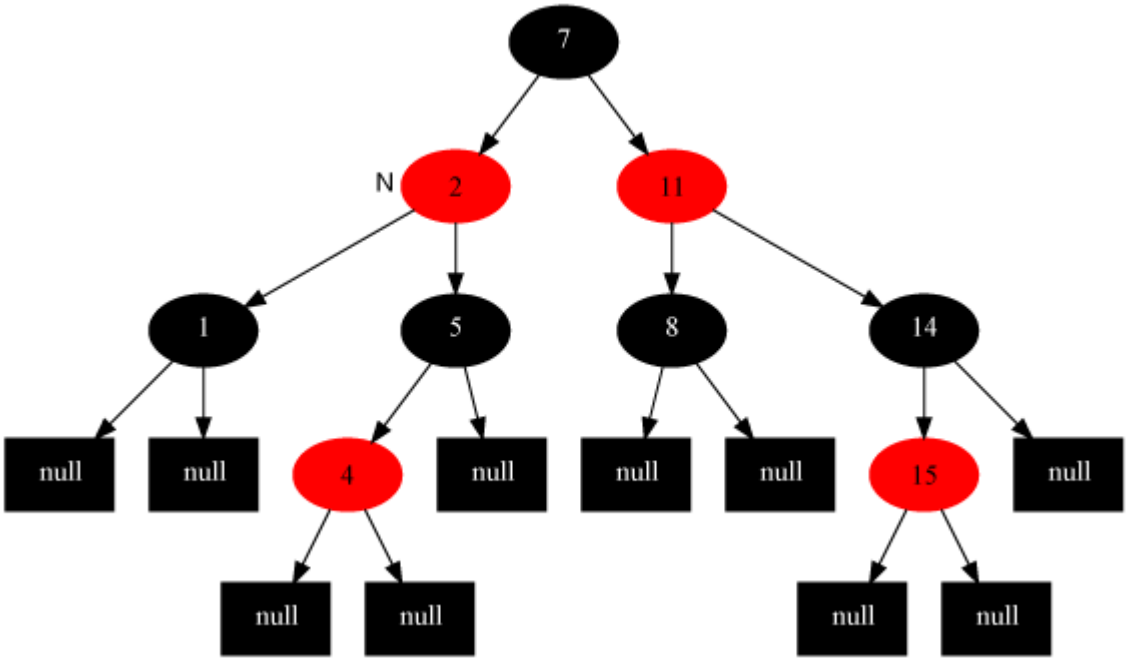


情况 5：当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的左子
解法：父节点变为黑色，祖父节点变为红色，在祖父节点为支点右旋

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。

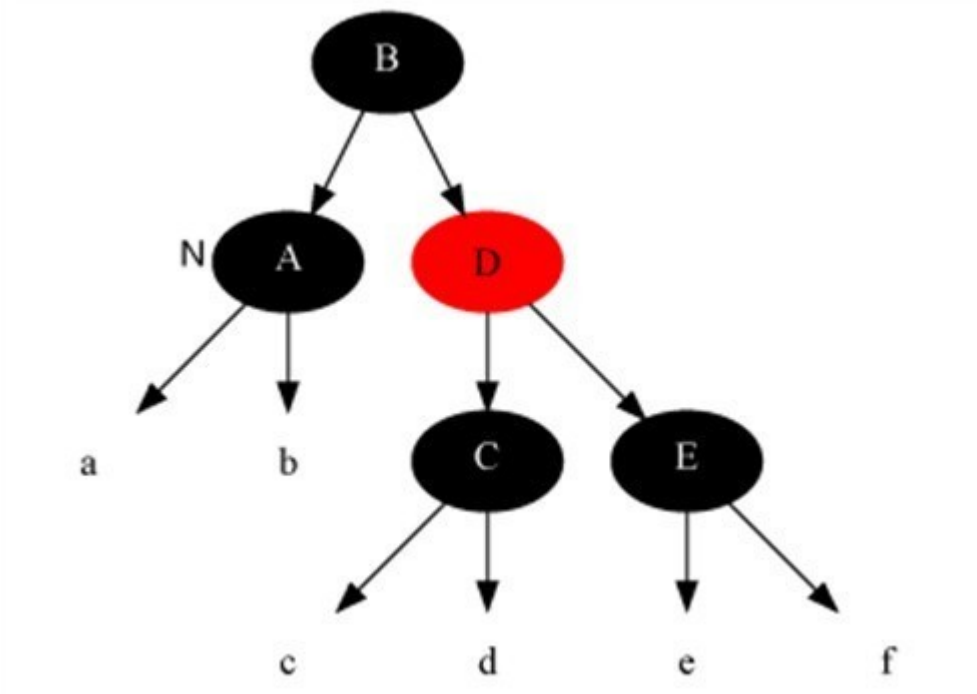
而下面所有的文字，则是针对红黑树删除结点后，所做的修复红黑树性质的工作。

二零一一年一月七日更新。

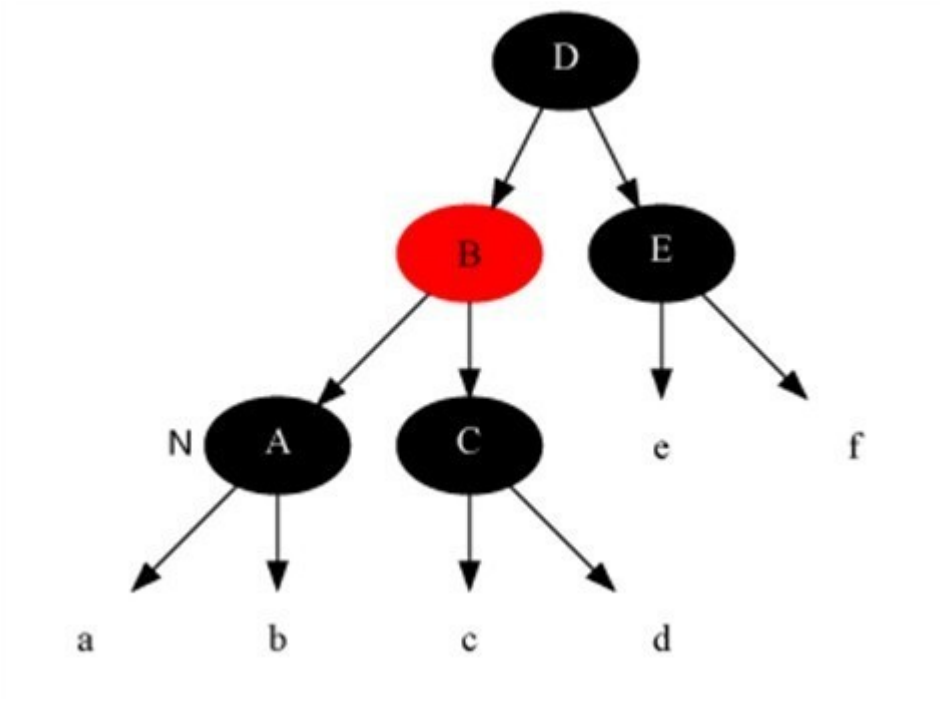
(注：以下的情况 3、4、5、6，与上述算法导论之代码 RB-DELETE-FIXUP(T, x) 恢复与保持中 case1，case2，case3，case4 相对应。)
情况 1：当前节点是红色
　解法，直接把当前节点染成黑色，结束。
　此时红黑树性质全部恢复。
情况 2：当前节点是黑色且是根节点
　解法：什么都不做，结束

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：



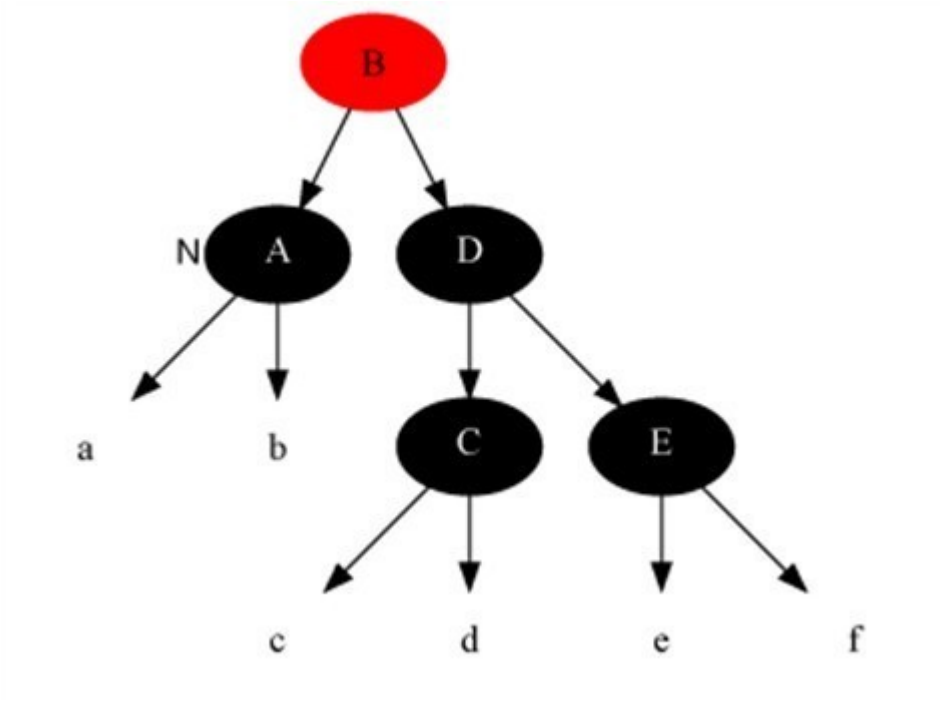
3.变化后：



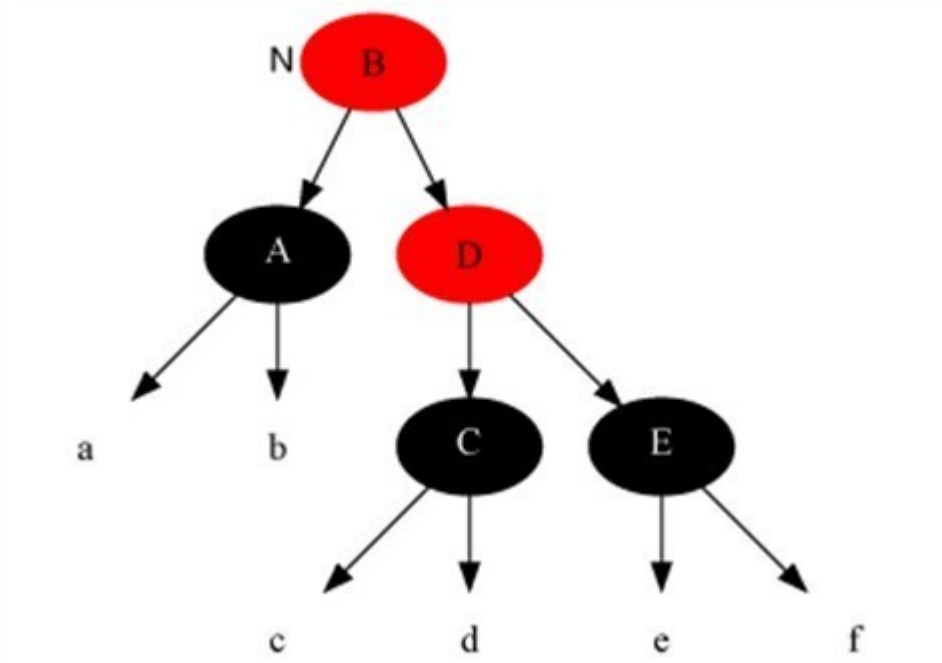
情况 4：当前节点是黑色，且兄弟是黑色，且兄弟节点的两个子节点全为黑色。

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



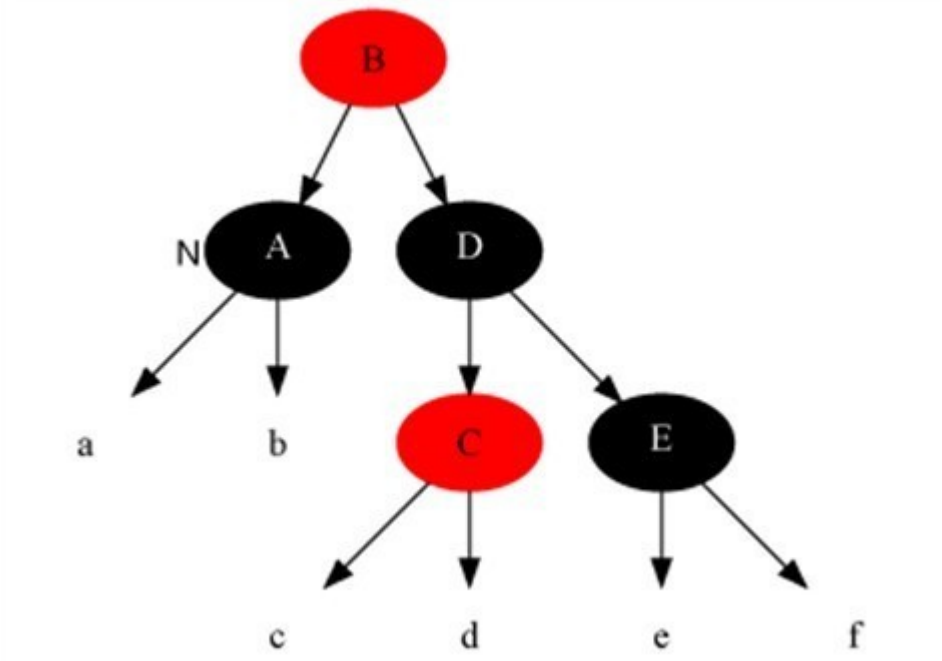
4.变化后



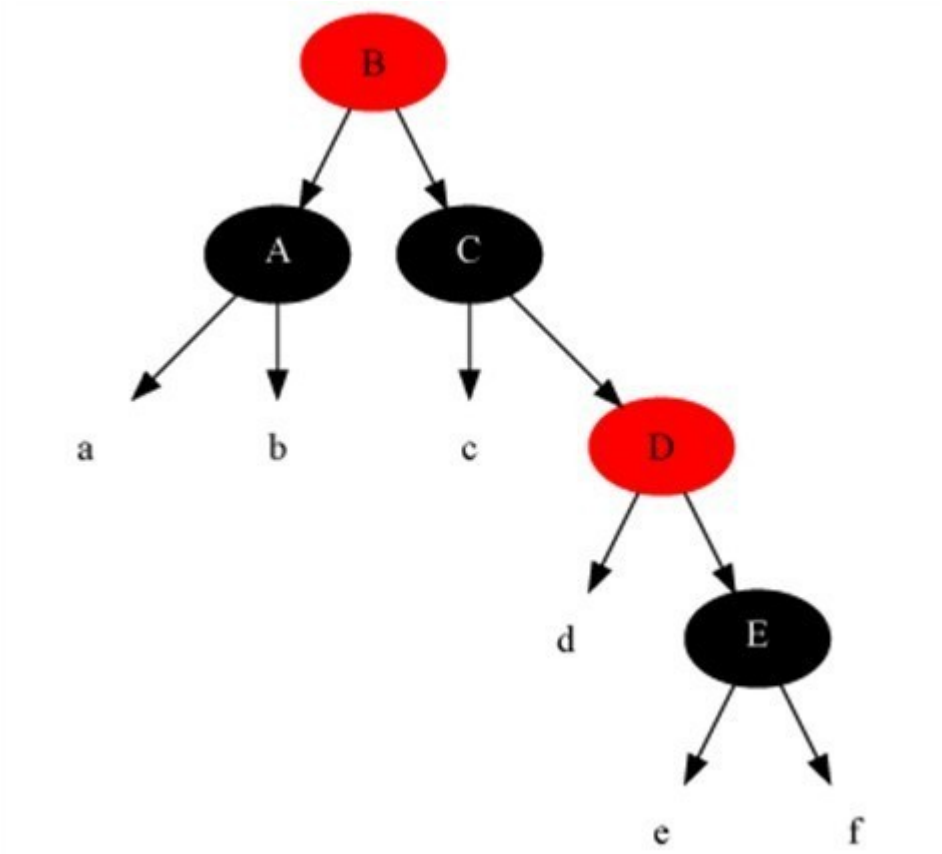
情况 5：当前节点颜色是黑色，兄弟节点是黑色，兄弟的左子是红色，右子是黑色。

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：



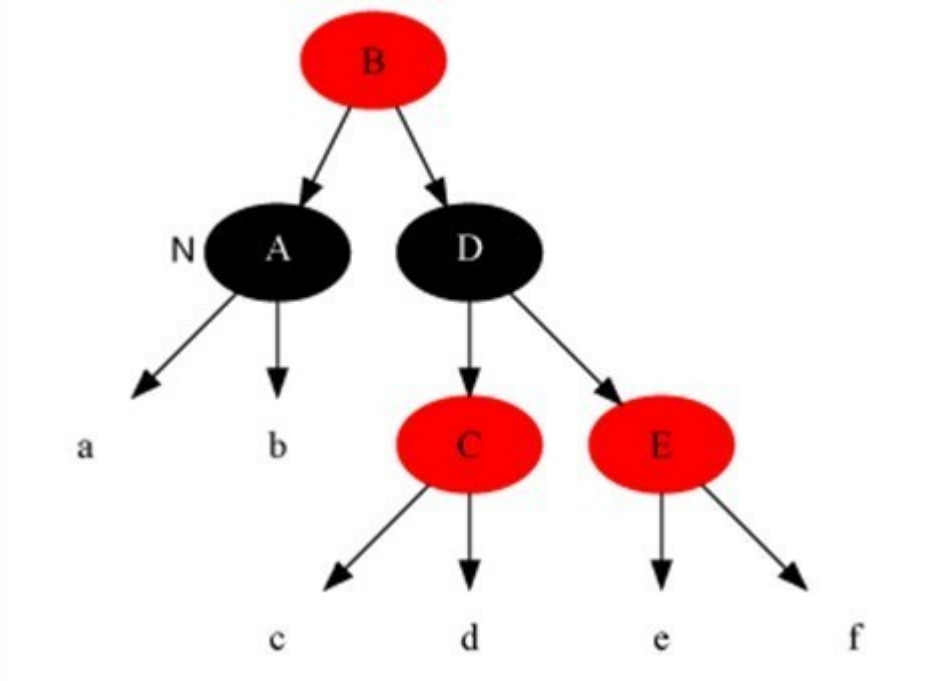
5.变化后：



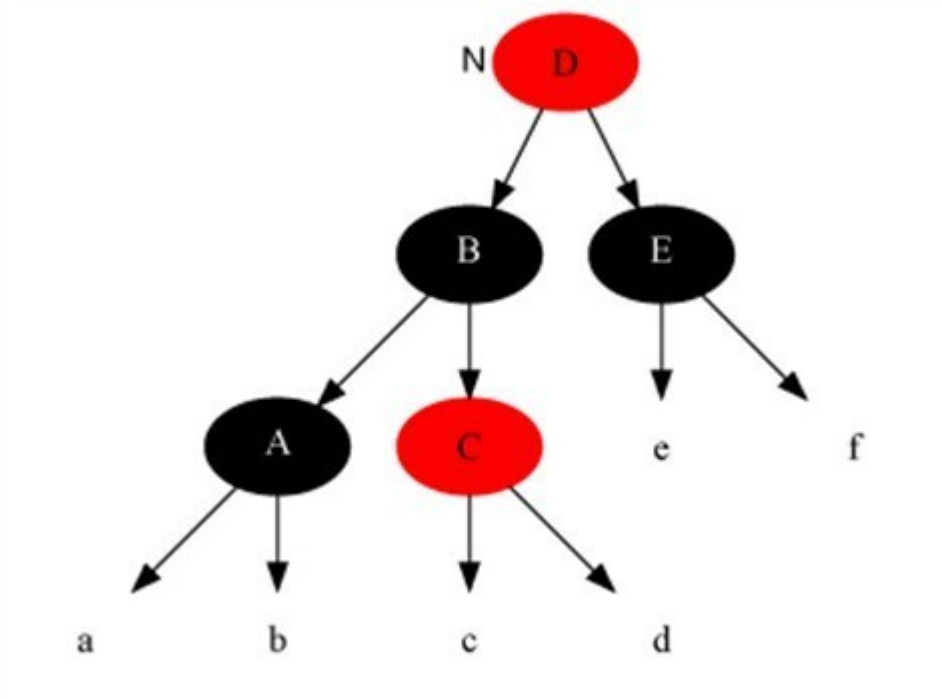
情况 6：当前节点颜色是黑色，它的兄弟节点是黑色，但是兄弟节点的右子是红色，兄弟节点左子的颜色任意。

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：

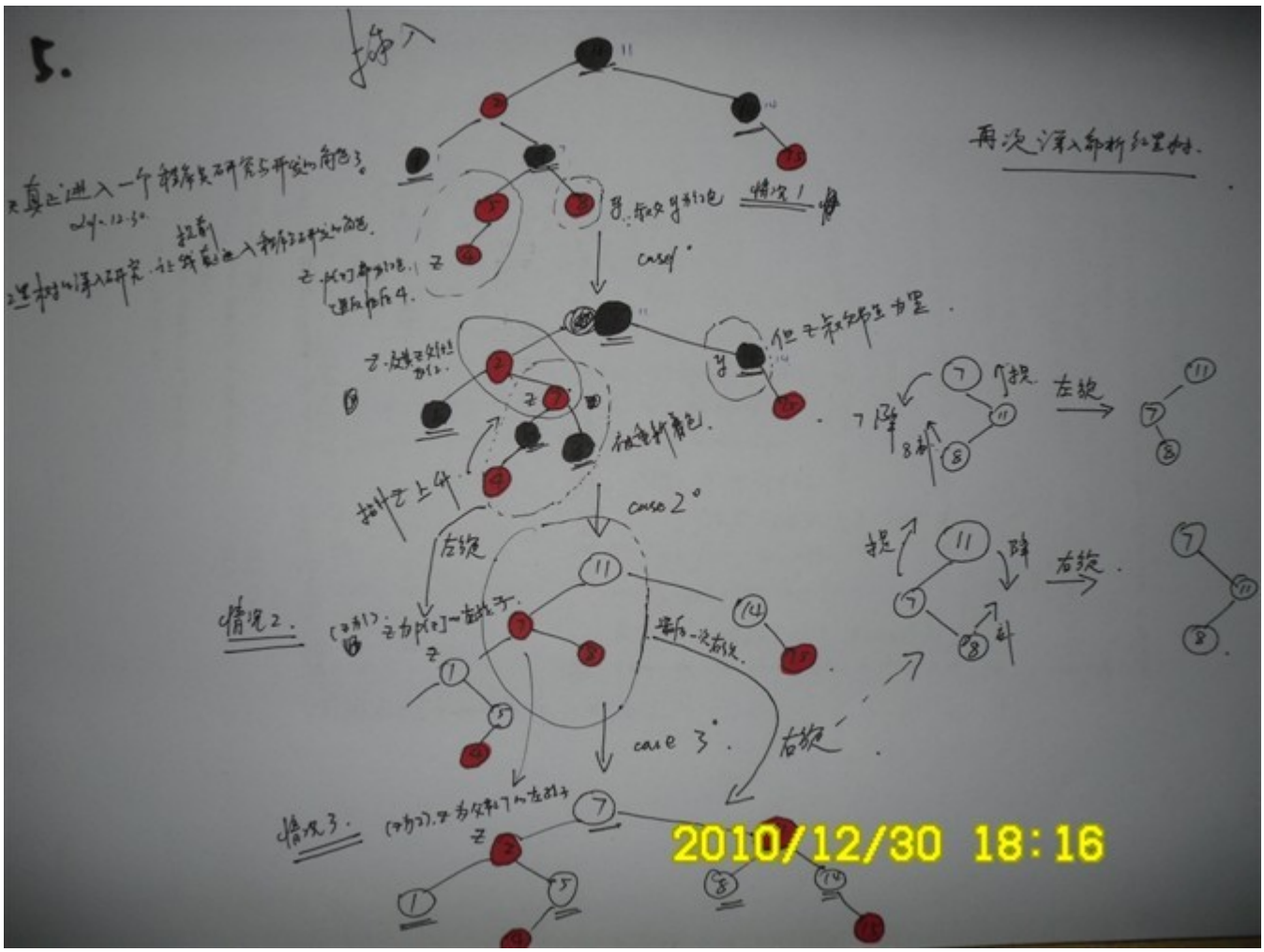


6.变化后：

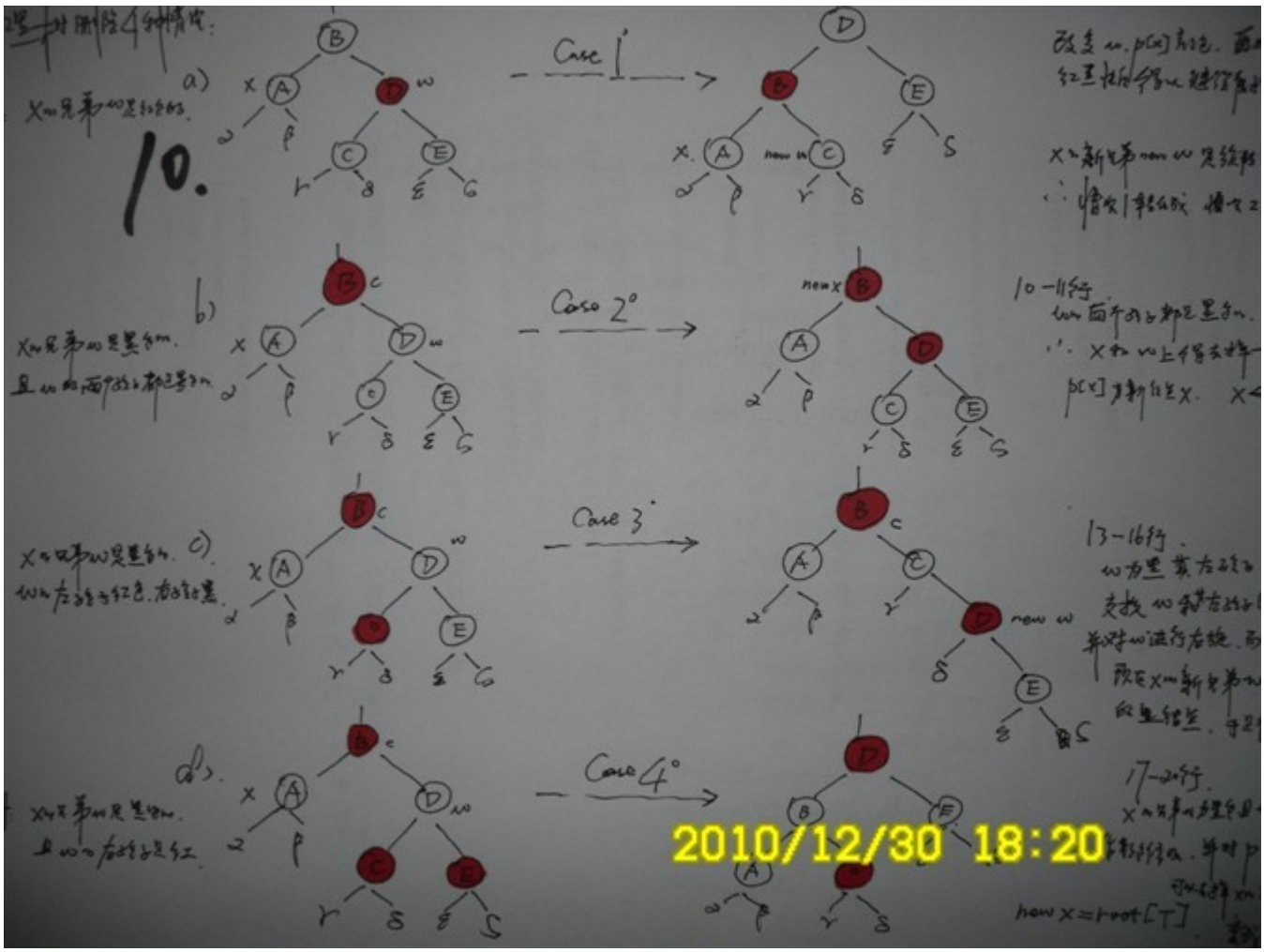


限于篇幅，不再过多赘述。更多，可参考算法导论或下文我写的第二篇文章。
完。

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

作者 July 二零一零年十二月三十一日

本文主要参考：算法导论第二版

本文主要代码：参考算法导论。

本文图片来源：个人手工画成、算法导论原书。

推荐阅读：Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的关于红黑树的一篇论文。

引言：

昨天下午画红黑树画了好几个钟头，总共 10 页纸。

特此，再深入剖析红黑树的算法实现，教你如何彻底实现红黑树算法。

经过我上一篇博文，“教你透彻了解红黑树”后，相信大家对红黑树已经有了一定的了解。
个人觉得，这个红黑树，还是比较容易懂的。
不论是插入、还是删除，不论是左旋还是右旋，最终的目的只有一个：
即保持红黑树的 5 个性质，不得违背。

再次，重述下红黑树的五个性质：
一般的，红黑树，满足一下性质，即只有满足一下性质的树，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。

抓住了红黑树的那 5 个性质，事情就好办多了。
如，
1.红黑红黑，要么是红，要么是黑；
2.根结点是黑；
3.每个叶结点是黑；
4.一个红结点，它的俩个儿子必然都是黑的；
5.每一条路径上，黑结点的数目等同。
五条性质，合起来，来句顺口溜就是：（1）红黑 （2）黑 （3）黑 （4&5）红->黑 黑。

本文所有的文字，都是参照我昨下午画的**十张纸**（即我拍的照片）与算法导论来写的。

希望，你依照此文一点一点的往下看，看懂此文后，你对红黑树的算法了解程度，一定大增不少。

ok，现在咱们来具体深入剖析红黑树的算法，并教你逐步实现此算法。

此教程分为 10 个部分，每一个部分作为一个小节。且各小节与我给的十张照片一一对应。

一、左旋与右旋

先明确一点：为什么要左旋?

因为红黑树插入或删除结点后，树的结构发生了变化，从而可能会破坏红黑树的性质。

为了维持插入、或删除结点后的树，仍然是一颗红黑树，所以有必要对树的结构做部分调整，从而恢复红黑树的原本性质。

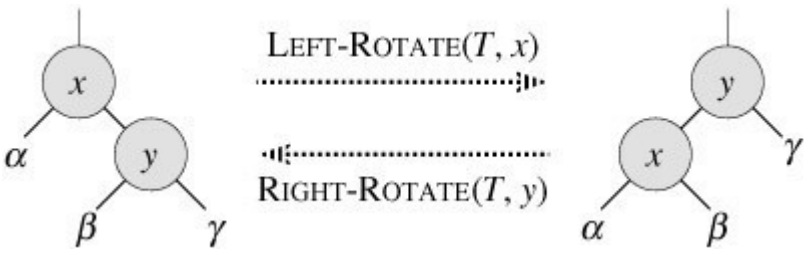
而为了恢复红黑性质而作的动作包括：

结点颜色的改变(重新着色)，和结点的调整。

这部分结点调整工作，改变指针结构，即是通过左旋或右旋而达到目的。

从而使插入、或删除结点的树重新成为一颗新的红黑树。

ok，请看下图：



如上图所示，‘找茬’

如果你看懂了上述俩幅图有什么区别时，你就知道什么是“左旋”，“右旋”。

在此，着重分析左旋算法：

左旋，如图所示（左->右），以 x->y 之间的链为“支轴”进行，

使 y 成为该新子树的根，x 成为 y 的左孩子，而 y 的左孩子则成为 x 的右孩子。

算法很简单，还有注意一点，各个结点从左往右，不论是左旋前还是左旋后，结点大小都是从小到大。

左旋代码实现，分三步（注意我给的注释）：

The pseudocode for LEFT-ROTATE assumes that right[x] ≠ nil[T] and that the root's parent is nil[T].

```
LEFT-ROTATE(T, x)
 1  y ← right[x]            ▷ Set y.
 2  right[x] ← left[y]                  //开始变化，y 的左孩子成为 x 的右孩子
 3  if left[y]  ！=nil[T]
 4  then p[left[y]] <- x
 5  p[y] <- p[x]                        //y 成为 x 的父母
 6  if p[x] = nil[T]
 7     then root[T] <- y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10              else right[p[x]] ← y
11  left[y] ← x              //x 成为 y 的左孩子（一月三日修正）
12  p[x] ← y
```

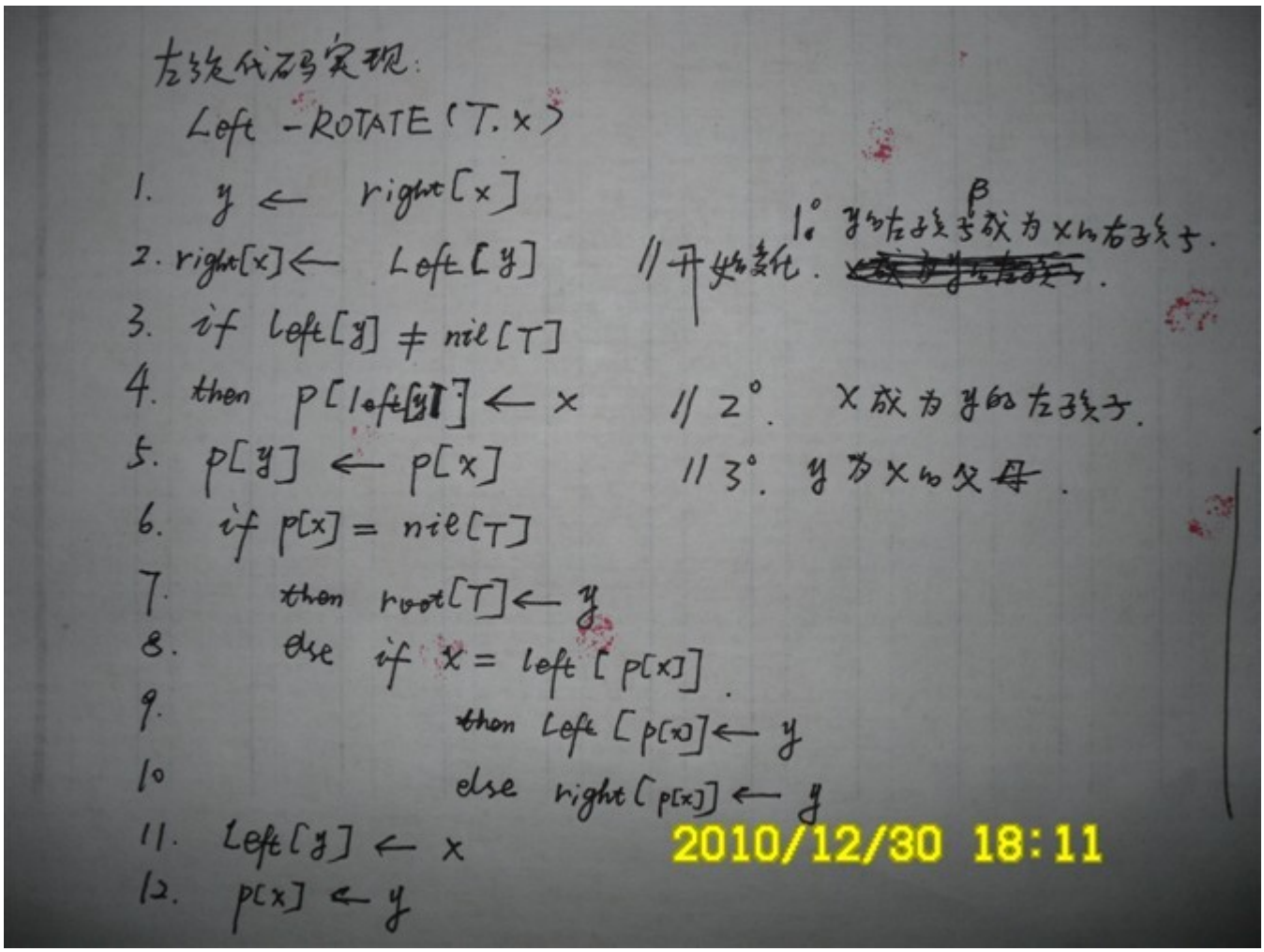
//注，此段左旋代码，原书第一版英文版与第二版中文版，有所出入。

//个人觉得，第二版更精准。所以，此段代码以第二版中文版为准。

左旋、右旋都是对称的，且都是在 O（1）时间内完成。因为旋转时只有指针被改变，而结点中的所有域都保持不变。

最后，贴出昨下午关于此右旋算法所画的图：

左旋（第 2 张图）：

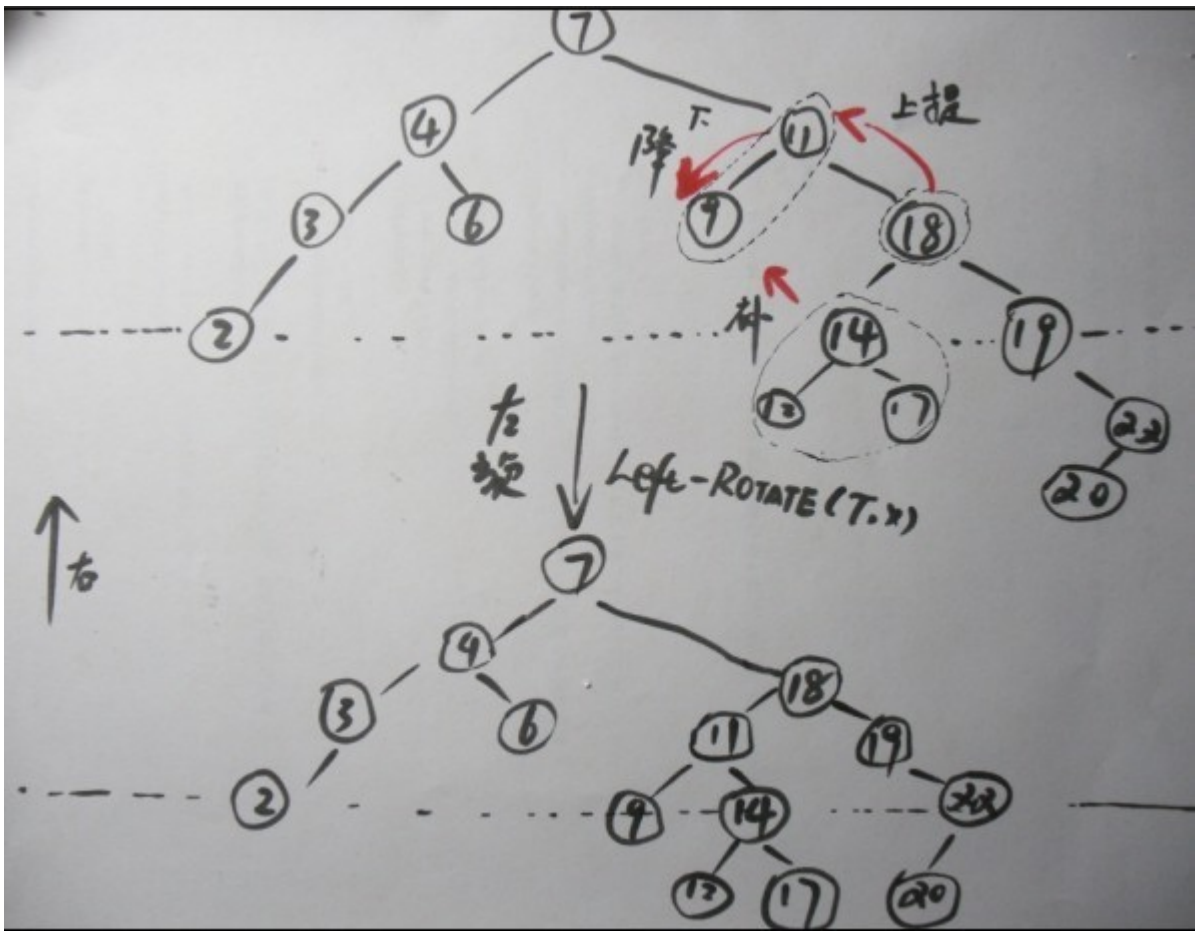


//此图有点 bug。第 4 行的注释移到第 11 行。如上述代码所示。（一月三日修正）

二、左旋的一个实例

不做过多介绍，看下副图，一目了然。

LEFT-ROTATE(T, x)的操作过程（第 3 张图）：



提醒，看下文之前，请首先务必明确，区别以下俩种操作：

1.红黑树插入、删除结点的操作

//如插入中，红黑树插入结点操作：RB-INSERT(T, z)。

2.红黑树已经插入、删除结点之后，

为了保持红黑树原有的红黑性质而做的恢复与保持红黑性质的操作。

//如插入中，为了恢复和保持原有红黑性质，所做的工作：RB-INSERT-FIXUP(T, z)。

ok，请继续。

三、红黑树的插入算法实现

```
RB-INSERT(T, z)   //注意我给的注释...
 1  y ← nil[T]                  // y 始终指向  x 的父结点。
 2  x ← root[T]                 // x 指向当前树的根结点，
 3  while x ≠ nil[T]
 4     do y ← x
 5        if key[z] < key[x]          //向左，向右..
 6           then x ← left[x]
 7           else x ← right[x]        // 为了找到合适的插入点，x 探路跟踪路径，直到 x 成为 NIL 为止。
 8  p[z] ← y          // y 置为  插入结点 z 的父结点。
 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     //此 8-13 行，置 z 相关的指针。
14  left[z] ← nil[T]
15  right[z] ← nil[T]          //设为空，
16  color[z] ← RED             //将新插入的结点 z 作为红色
17  RB-INSERT-FIXUP(T, z)   //因为将 z 着为红色，可能会违反某一红黑性质，
                                                    //所以需要调用 RB-INSERT-FIXUP(T, z)来保持红黑
性质。
```

17 行的 **RB-INSERT-FIXUP(T, z)** ，在下文会得到着重而具体的分析。

还记得，我开头说的那句话么，

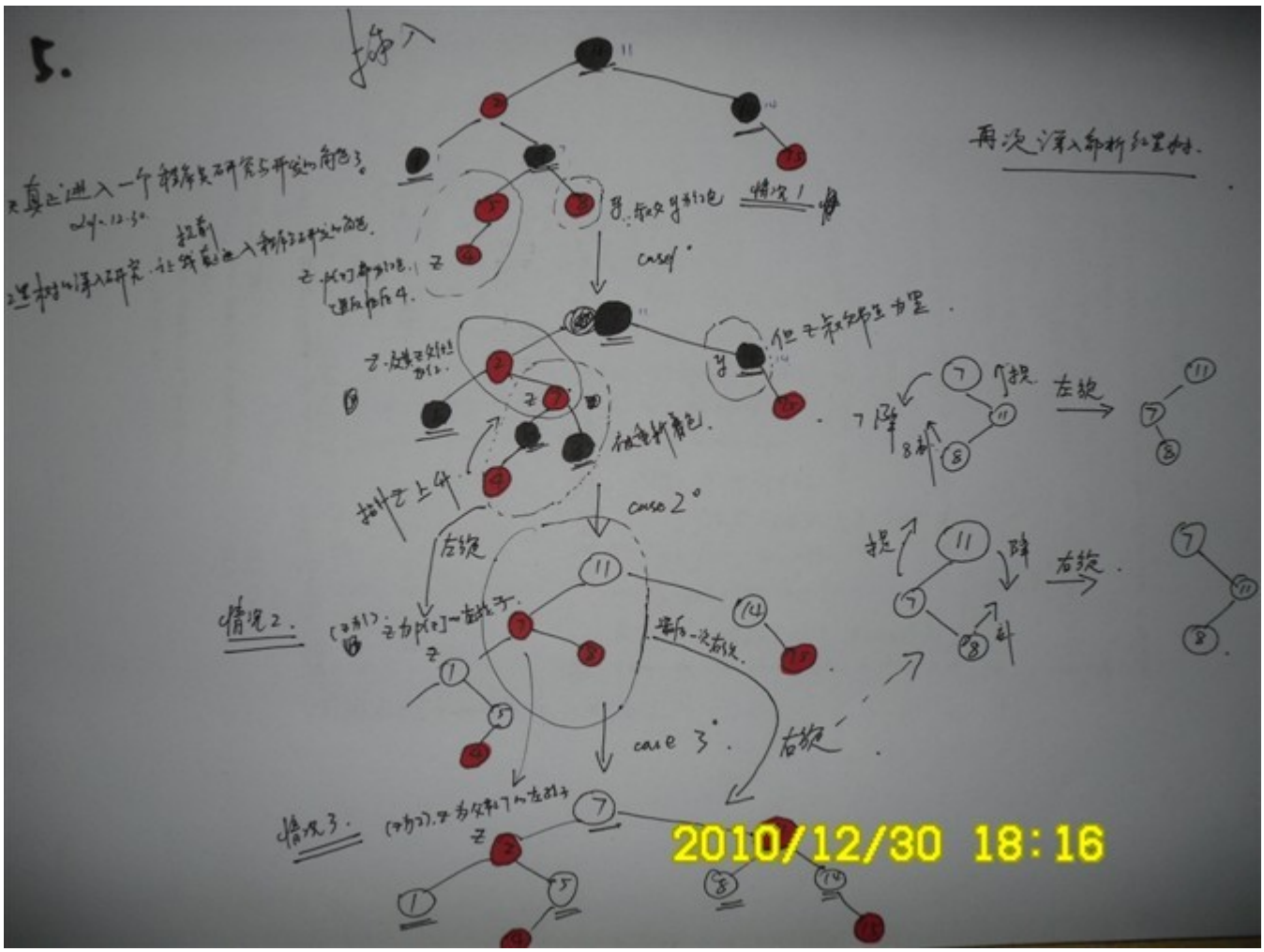
是的，时刻记住，不论是左旋还是右旋，不论是插入、还是删除，都要记得恢复和保持红黑树的 5 个性质。

四、调用 RB-INSERT-FIXUP(T, z)来保持和恢复红黑性质

```
RB-INSERT-FIXUP(T, z)
 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
```

//第 4 张图略：

五、红黑树插入的三种情况，即 RB-INSERT-FIXUP(T, z)。操作过程（第 5 张）：



//这幅图有个小小的问题，读者可能会产生误解。图中左侧所表明的情况 2、情况 3 所标的位置都要标上一点。

//请以图中的标明的 case1、case2、case3 为准。一月三日。

六、红黑树插入的第一种情况（RB-INSERT-FIXUP(T, z)代码的具体分析一）

为了保证阐述清晰，重述下 RB-INSERT-FIXUP(T, z)的源码：

```
RB-INSERT-FIXUP(T, z)
 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

//case1 表示情况 1，case2 表示情况 2，case3 表示情况 3.

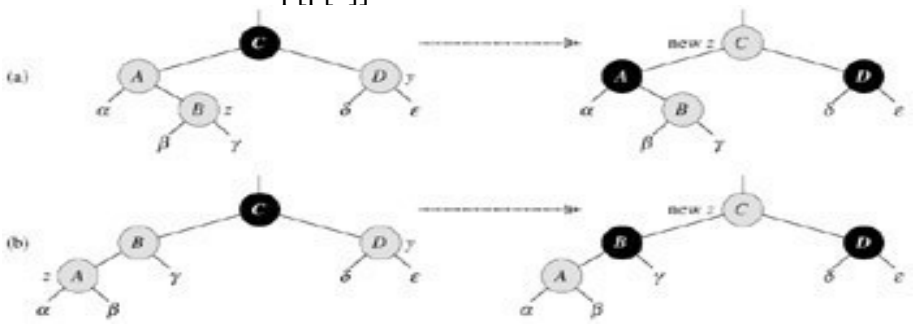
ok，如上所示，相信，你已看到了。

咱们，先来透彻分析红黑树插入的第一种情况：

插入情况 1，z 的叔叔 y 是红色的。

第一种情况，即上述代码的第 5-8 行：

```
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
```



如上图所示，a：z 为右孩子，b：z 为左孩子。

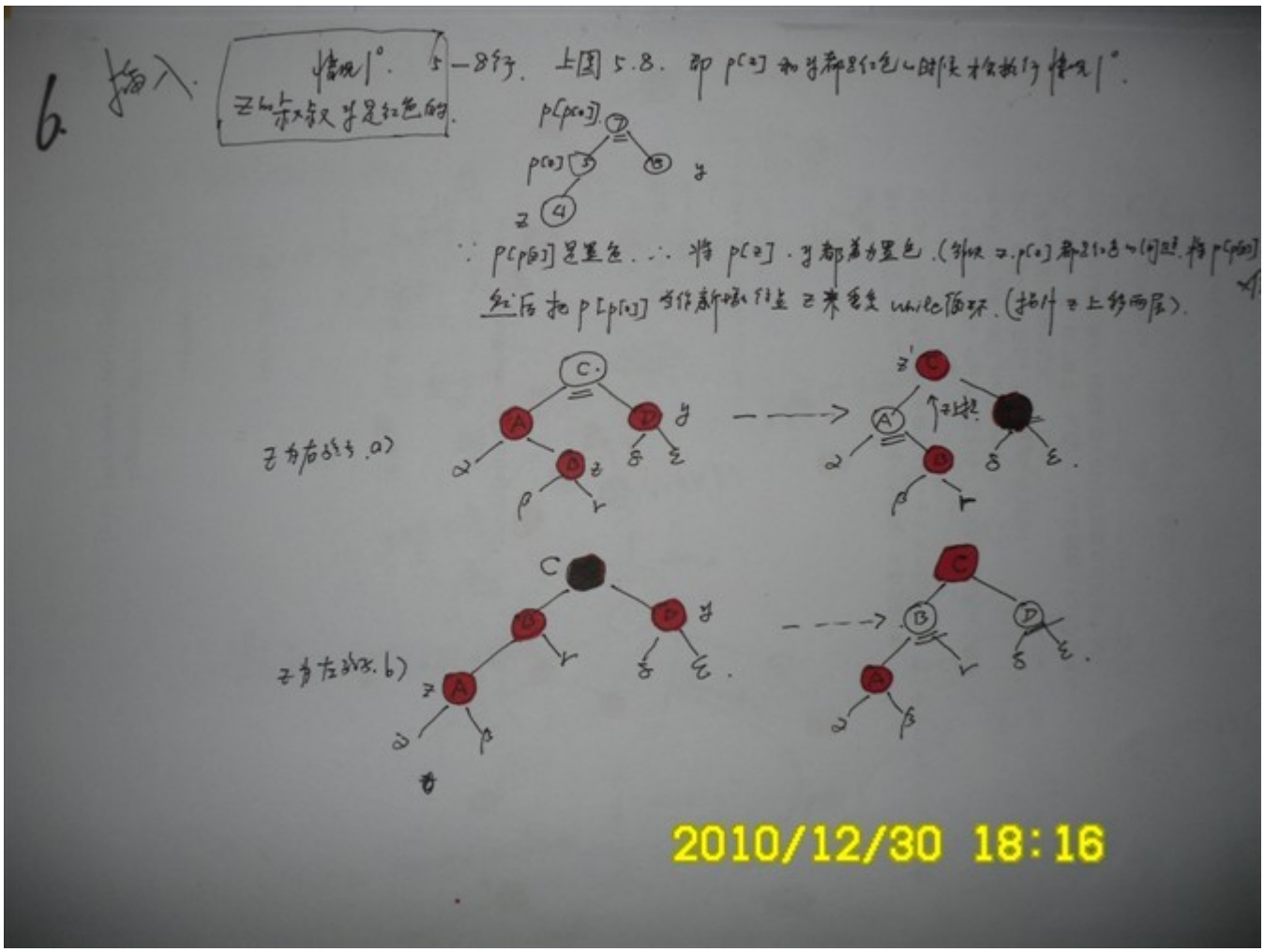
只有 p[z]和 y（上图 a 中 A 为 p[z]，D 为 z，上图 b 中，B 为 p[z]，D 为 y）都是红色的时候，才会执行此情况 1.

咱们分析下上图的 a 情况，即 z 为右孩子时

因为 p[p[z]]，即 c 是黑色，所以将 p[z]、y 都着为黑色（如上图 a 部分的右边），

此举解决 z、p[z]都是红色的问题，将 p[p[z]]着为红色，则保持了性质 5.

ok，看下我昨天画的图（第 6 张）：



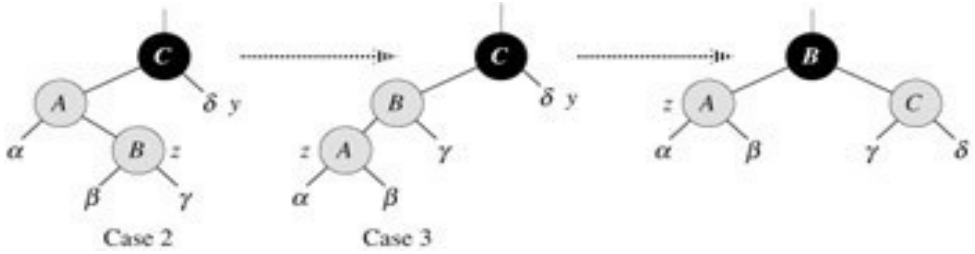
红黑树插入的第一种情况完。

七、红黑树插入的第二种、第三种情况

插入情况 2：z 的叔叔 y 是黑色的，且 z 是右孩子

插入情况 3：z 的叔叔 y 是黑色的，且 z 是左孩子

这俩种情况，是通过 z 是 p[z]的左孩子，还是右孩子区别的。



参照上图，针对情况 2，z 是她父亲的右孩子，则为了保持红黑性质，左旋则变为情况 3，此时 z 为左孩子，

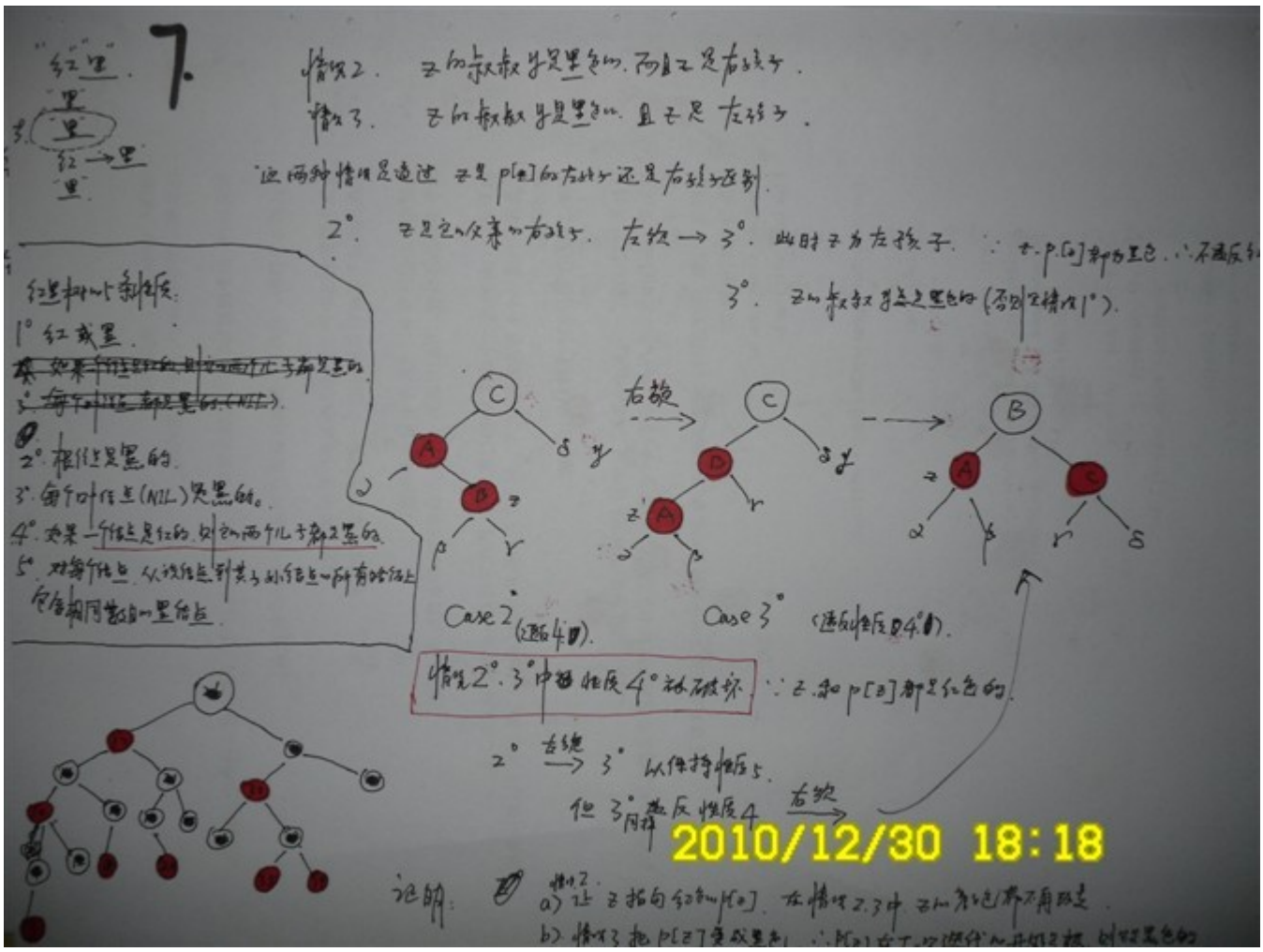
因为 z、p[z]都为黑色，所以不违反红黑性质（注，情况 3 中，z 的叔叔 y 是黑色的，否则此种情况就变成上述情况 1 了）。

ok，我们已经看出来了，情况 2，情况 3 都违反性质 4（一个红结点的俩个儿子都是黑色的）。

所以情况 2->左旋后->情况 3，此时情况 3 同样违反性质 4，所以情况 3->右旋，得到上图的最后那部分。

注，情况 2、3 都只违反性质 4，其它的性质 1、2、3、5 都不违背。

好的，最后，看下我画的图（第 7 张）：



八、接下来，进入**红黑树的删除**部分。

```
RB-DELETE(T, z)
 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                //如果 y 是黑色的，
17    then RB-DELETE-FIXUP(T, x)   //则调用 RB-DELETE-FIXUP(T, x)
18 return y                //如果 y 不是黑色，是红色的，则当 y 被删除时，红黑性质仍然得以保持。不做操作，返回。
                                                    //因为：1.树种各结点的黑高度都没有变化。2.不存在俩个相邻的红色结点。
```

//3.因为入宫 y 是红色的，就不可能是根。所以，根仍然是黑色的。

ok，第 8 张图，不必贴了。

九、红黑树删除之 4 种情况，RB-DELETE-FIXUP(T, x)之代码

```
RB-DELETE-FIXUP(T, x)
 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
```

ok，很清楚，在此，就不贴第 9 张图了。

在下文的红黑树删除的 4 种情况，详细、具体分析了上段代码。

十、红黑树删除的 4 种情况

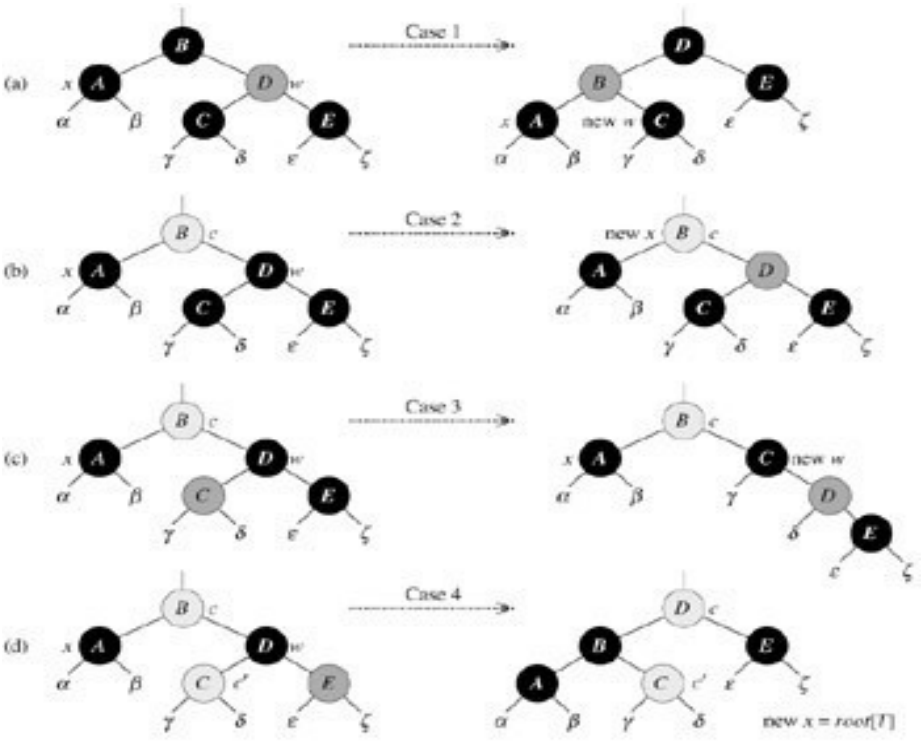
情况 1：x 的兄弟 w 是红色的。

情况 2：x 的兄弟 w 是黑色的，且 w 的俩个孩子都是黑色的。

情况 3：x 的兄弟 w 是黑色的，w 的左孩子是红色，w 的右孩子是黑色。

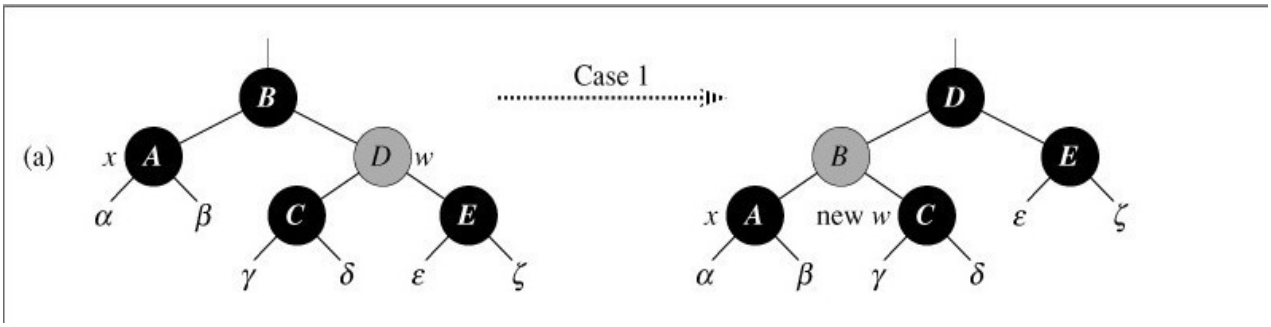
情况 4：x 的兄弟 w 是黑色的，且 w 的右孩子时红色的。

操作流程图：



ok，简单分析下，红黑树删除的 4 种情况：

针对情况 1：x 的兄弟 w 是红色的。



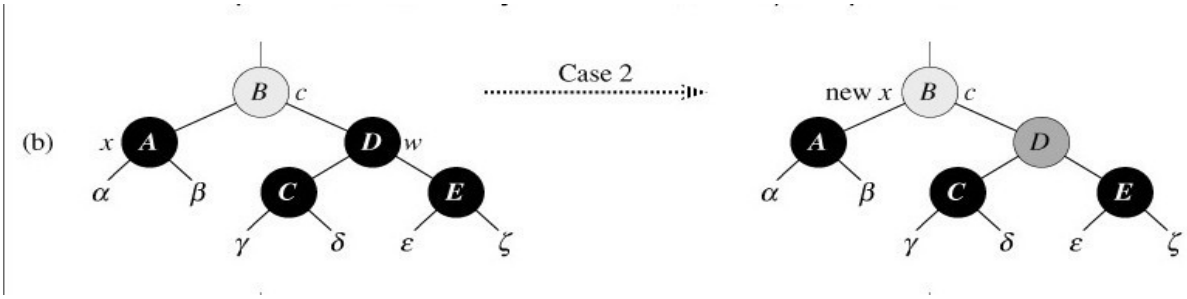

```
 5        then color[w] ← BLACK              ▷ Case 1
 6           color[p[x]] ← RED               ▷ Case 1
 7           LEFT-ROTATE(T, p[x])            ▷ Case 1
 8           w ← right[p[x]]                 ▷ Case 1
```

对策：改变 w、p[z]颜色，再对 p[x]做一次左旋，红黑性质得以继续保持。

x 的新兄弟 new w 是旋转之前 w 的某个孩子，为黑色。

所以，情况 1 转化成情况 2 或 3、4。

针对情况 2：x 的兄弟 w 是黑色的，且 w 的俩个孩子都是黑色的。




```
10        then color[w] ← RED               ▷ Case 2
```

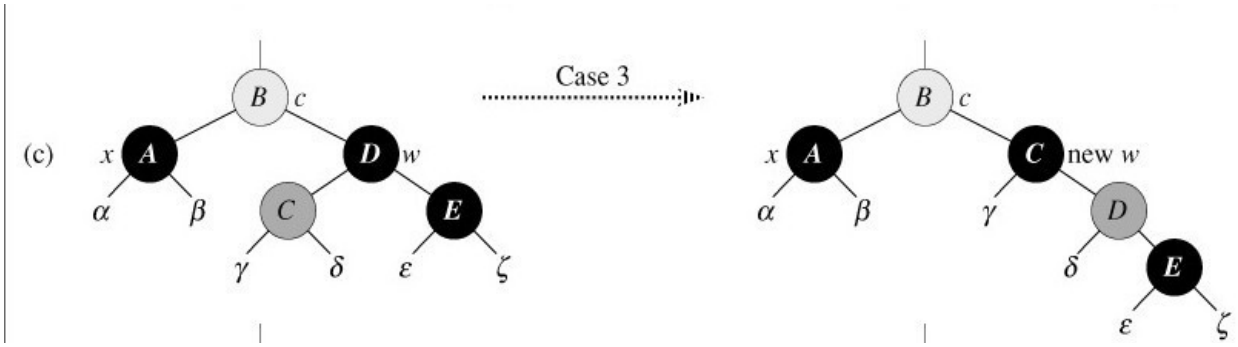
11 x <-p[x] ▷ Case 2

如图所示，**w 的俩个孩子都是黑色的，**

对策：因为 w 也是黑色的，所以 x 和 w 中得去掉一黑色，最后，w 变为红。

p[x]为新结点 x，赋给 x，x<-p[x]。

针对情况 3：x 的兄弟 w 是黑色的，w 的左孩子是红色，w 的右孩子是黑色。



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

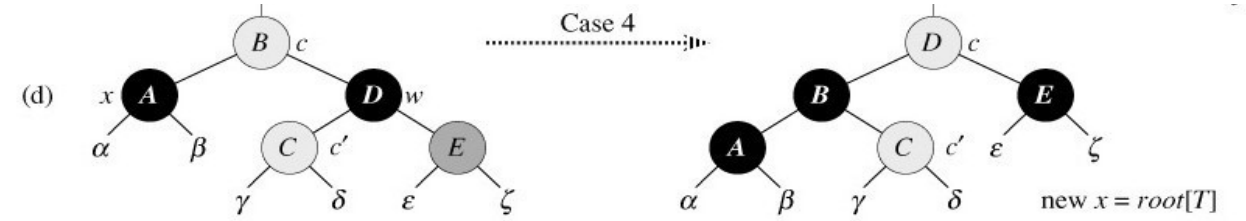
w 为黑，其左孩子为红，右孩子为黑

对策：交换 w 和和其左孩子 left[w]的颜色。 即上图的 D、C 颜色互换。:D。

并对 w 进行右旋，而红黑性质仍然得以保持。

现在 x 的新兄弟 w 是一个有红色右孩子的黑结点，于是将情况 3 转化为情况 4.

针对情况 4：x 的兄弟 w 是黑色的，且 w 的右孩子时红色的。



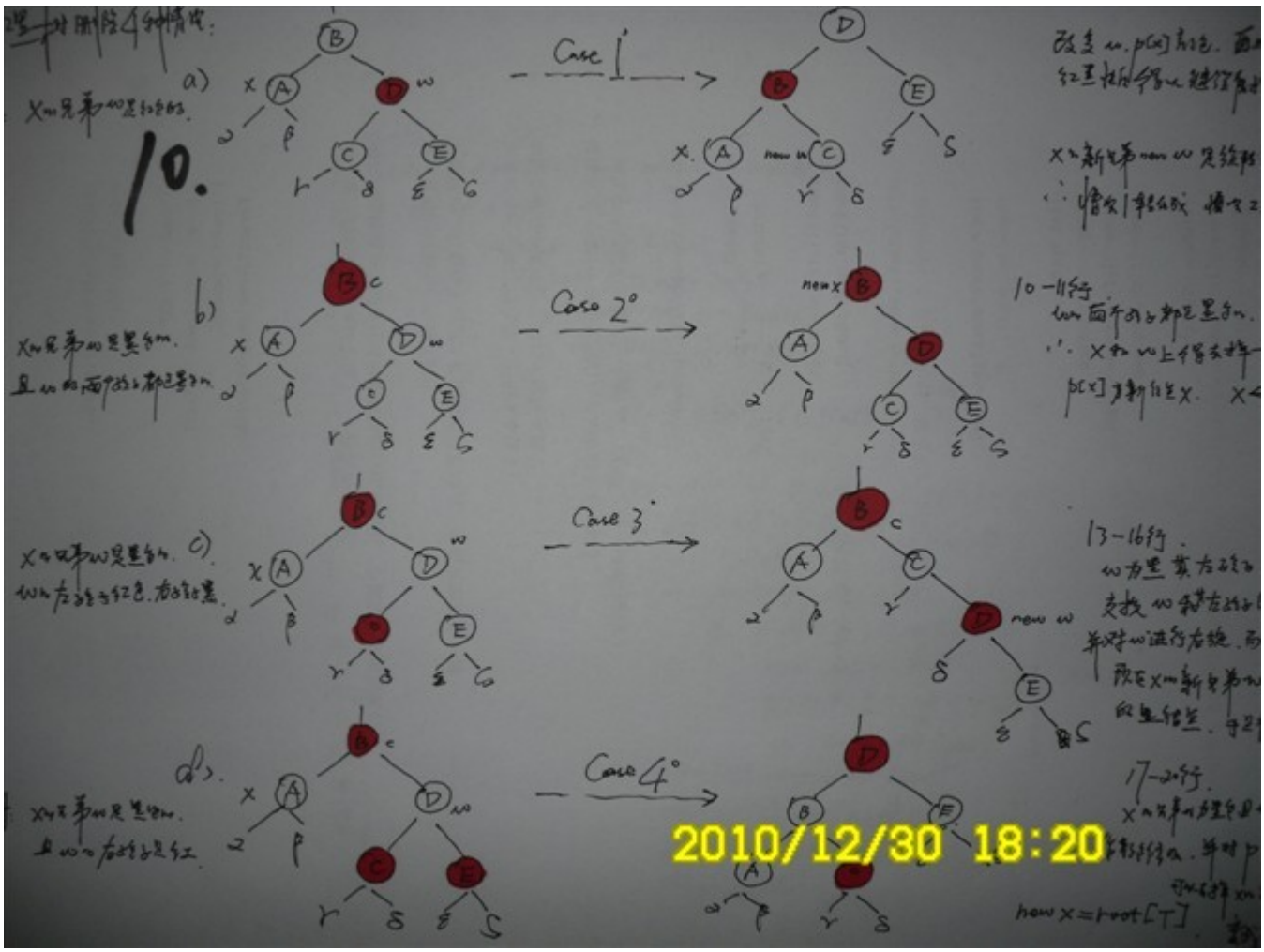
17 color[w] ← color[p[x]] ▷ Case 4
18 color[p[x]] ← BLACK ▷ Case 4
19 color[right[w]] ← BLACK ▷ Case 4
20 LEFT-ROTATE(T, p[x]) ▷ Case 4
21 x ← root[T] ▷ Case 4

x 的兄弟 w 为黑色，且 w 的右孩子为红色。

对策：做颜色修改，并对 p[x]做一次旋转，可以去掉 x 的额外黑色，来把 x 变成单独的黑色，此举不破坏红黑性质。

将 x 置为根后，循环结束。

最后，贴上最后的第 10 张图：



ok，红黑树删除的 4 中情况，分析完成。

结语：只要牢牢抓住红黑树的 5 个性质不放，而不论是树的左旋还是右旋，
不论是红黑树的插入、还是删除，都只为了保持和修复红黑树的 5 个性质而已。

红黑树 c 源码实现与剖析

作者：July、那谁　时间：二零一一年一月三日

前言：

红黑树作为一种经典而高级的数据结构，相信，已经被不少人实现过，但不是因为程序不够完善而无法运行，就是因为程序完全没有注释，初学者根本就看不懂。

此份红黑树的 c 源码最初从 linux-lib-rbtree.c 而来，后经一网友那谁（http://www.cppblog.com/converse/）用 c 写了出来。在此，向原作者表示敬意。但原来的程序存在诸多问题，且全程序没有任何一行注释。没有一行注释的程序，令程序的价值大打折扣。

所以，我特把这份源码放到了 windows xp+vc 6.0 上，一行一行的完善修正，一行一行的给它添加注释，至此，红黑树 c 带完美注释的源码，就摆在了您眼前，有不妥、不正之处，还望不吝指正。

红黑树的六篇文章：

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

---

ok，咱们开始吧。

相信，经过我前俩篇博文对红黑树的介绍，你应该对红黑树有了透彻的理解了（没看过的朋友，可事先查上面的俩篇文章，或与此文的源码剖析对应着看）。

本套源码剖析把重点放在红黑树的 3 种插入情况，与红黑树的 4 种删除情况。其余的能从略则尽量简略。

目录：

好的，咱们还是先从树的左旋、右旋代码，开始（大部分分析，直接给注释）：

view plaincopy to clipboardprint?

```
1. //一、左旋代码分析
2. /*--------------------------------------------------------
3. |   node          right
4. |   / \    ==>    / \
5. |   a  right     node  y
6. |      / \       / \
7. |      b  y      a   b    //左旋
8. --------------------------------------------------------*/
9. static rb_node_t* rb_rotate_left(rb_node_t* node, rb_node_t* root)
10.{
11.     rb_node_t* right = node->right;    //指定指针指向  right<--node->right
12.
13.     if ((node->right = right->left))
14.     {
15.         right->left->parent = node;  //好比上面的注释图，node 成为 b 的父母
16.     }
17.     right->left = node;   //node 成为 right 的左孩子
18.
19.     if ((right->parent = node->parent))
20.     {
21.         if (node == node->parent->right)
22.         {
23.             node->parent->right = right;
24.         }
25.         else
26.         {
27.             node->parent->left = right;
28.         }
29.     }
30.     else
```

```
31.     {
32.         root = right;
33.     }
34.     node->parent = right;  //right 成为 node 的父母
35.
36.     return root;
37.}
38.
39.
40.//二、右旋
41./*-----------------------------------------------------------
42.|      node           left
43.|      / \           / \
44.|   left  y   ==>    a    node
45.|   / \                / \
46.|  a   b               b   y  //右旋与左旋差不多，分析略过
47.-----------------------------------------------------------*/
48.static rb_node_t* rb_rotate_right(rb_node_t* node, rb_node_t* root)
49.{
50.     rb_node_t* left = node->left;
51.
52.     if ((node->left = left->right))
53.     {
54.         left->right->parent = node;
55.     }
56.     left->right = node;
57.
58.     if ((left->parent = node->parent))
59.     {
60.         if (node == node->parent->right)
61.         {
62.             node->parent->right = left;
63.         }
64.         else
65.         {
66.             node->parent->left = left;
67.         }
68.     }
69.     else
70.     {
71.         root = left;
72.     }
73.     node->parent = left;
74.
75.     return root;
76.}
77.
78.
79.//三、红黑树查找结点
80.//-----------------------------------------------------
81.//rb_search_auxiliary：查找
82.//rb_node_t* rb_search：返回找到的结点
```

```
83.//-----------------------------------------------------
84.static rb_node_t* rb_search_auxiliary(key_t key, rb_node_t* root, rb_node_t** save)
85.{
86.     rb_node_t *node = root, *parent = NULL;
87.     int ret;
88.
89.     while (node)
90.     {
91.          parent = node;
92.          ret = node->key - key;
93.          if (0 < ret)
94.          {
95.               node = node->left;
96.          }
97.          else if (0 > ret)
98.          {
99.               node = node->right;
100.          }
101.          else
102.          {
103.               return node;
104.          }
105.     }
106.
107.     if (save)
108.     {
109.          *save = parent;
110.     }
111.
112.     return NULL;
113.}
114.
115.//返回上述 rb_search_auxiliary 查找结果
116.rb_node_t* rb_search(key_t key, rb_node_t* root)
117.{
118.     return rb_search_auxiliary(key, root, NULL);
119.}
120.
121.
122.//四、红黑树的插入
123.//-----------------------------------------------------------
124.//红黑树的插入结点
125.rb_node_t* rb_insert(key_t key, data_t data, rb_node_t* root)
126.{
127.     rb_node_t *parent = NULL, *node;
128.
129.     parent = NULL;
130.     if ((node = rb_search_auxiliary(key, root, &parent)))  //调用 rb_search_auxiliary 找到插
   入结
131.
132.点的地方
133.     {
```

```
134.            return root;
135.        }
136.
137.        node = rb_new_node(key, data);  //分配结点
138.        node->parent = parent;
139.        node->left = node->right = NULL;
140.        node->color = RED;
141.
142.        if (parent)
143.        {
144.            if (parent->key > key)
145.            {
146.                parent->left = node;
147.            }
148.            else
149.            {
150.                parent->right = node;
151.            }
152.        }
153.        else
154.        {
155.            root = node;
156.        }
157.
158.        return rb_insert_rebalance(node, root);   //插入结点后，调用 rb_insert_rebalance 修复红
    黑树
159.
160.的性质
161.}
162.
163.
164.//五、红黑树的 3 种插入情况
165.//接下来，咱们重点分析针对红黑树插入的 3 种情况，而进行的修复工作。
166.//------------------------------------------------------------
167.//红黑树修复插入的 3 种情况
168.//为了在下面的注释中表示方便，也为了让下述代码与我的俩篇文章相对应，
169.//用 z 表示当前结点，p[z]表示父母、p[p[z]]表示祖父、y 表示叔叔。
170.//------------------------------------------------------------
171.static rb_node_t* rb_insert_rebalance(rb_node_t *node, rb_node_t *root)
172.{
173.        rb_node_t *parent, *gparent, *uncle, *tmp;  //父母 p[z]、祖父 p[p[z]]、叔叔 y、临时结
    点*tmp
174.
175.        while ((parent = node->parent) && parent->color == RED)
176.        {    //parent 为 node 的父母，且当父母的颜色为红时
177.            gparent = parent->parent;   //gparent 为祖父
178.
179.            if (parent == gparent->left)  //当祖父的左孩子即为父母时。
180.                                          //其实上述几行语句，无非就是理顺孩子、父母、祖父的
    关系。:D。
181.            {
```

```
182.                uncle = gparent->right;  //定义叔叔的概念，叔叔 y 就是父母的右孩子。
183.
184.                if (uncle && uncle->color == RED) //情况 1：z 的叔叔 y 是红色的
185.                {
186.                    uncle->color = BLACK;   //将叔叔结点 y 着为黑色
187.                    parent->color = BLACK;  //z 的父母 p[z]也着为黑色。解决 z，p[z]都是红
    色的问题。
188.                    gparent->color = RED;
189.                    node = gparent;     //将祖父当做新增结点 z，指针 z 上移俩层，且着为红色。

190.     //上述情况 1 中，只考虑了 z 作为父母的右孩子的情况。
191.                }
192.                else                //情况 2：z 的叔叔 y 是黑色的，
193.                {
194.                    if (parent->right == node)  //且 z 为右孩子
195.                    {
196.                        root = rb_rotate_left(parent, root); //左旋[结点 z，与父母结点]
197.                        tmp = parent;
198.                        parent = node;
199.                        node = tmp;     //parent 与 node 互换角色
200.                    }
201.                                //情况 3：z 的叔叔 y 是黑色的，此时 z 成为了左孩子。
202.                                    //注意，1：情况 3 是由上述情况 2 变化而来的。
203.                                    //.....2：z 的叔叔总是黑色的，否则就是情况 1 了。
204.                    parent->color = BLACK;  //z 的父母 p[z]着为黑色
205.                    gparent->color = RED;   //原祖父结点着为红色
206.                    root = rb_rotate_right(gparent, root); //右旋[结点 z，与祖父结点]
207.                }
208.            }
209.
210.            else
211.            {
212.     //这部分是特别为情况 1 中，z 作为左孩子情况，而写的。
213.                uncle = gparent->left;  //祖父的左孩子作为叔叔结点。[原理还是与上部分一样
    的]
214.                if (uncle && uncle->color == RED)  //情况 1：z 的叔叔 y 是红色的
215.                {
216.                    uncle->color = BLACK;
217.                    parent->color = BLACK;
218.                    gparent->color = RED;
219.                    node = gparent;         //同上。
220.                }
221.                else                        //情况 2：z 的叔叔 y 是黑色的，
222.                {
223.                    if (parent->left == node)  //且 z 为左孩子
224.                    {
225.                        root = rb_rotate_right(parent, root);  //以结点 parent、root 右旋
226.                        tmp = parent;
227.                        parent = node;
228.                        node = tmp;      //parent 与 node 互换角色
229.                    }
```

```
                    //经过情况 2 的变化，成为了情况 3.
                parent->color = BLACK;
                gparent->color = RED;
                root = rb_rotate_left(gparent, root);   //以结点 gparent 和 root 左旋
            }
        }
    }

    root->color = BLACK; //根结点，不论怎样，都得置为黑色。
    return root;     //返回根结点。
}


//六、红黑树的删除
//------------------------------------------------------------
//红黑树的删除结点
rb_node_t* rb_erase(key_t key, rb_node_t *root)
{
    rb_node_t *child, *parent, *old, *left, *node;
    color_t color;

    if (!(node = rb_search_auxiliary(key, root, NULL)))  //调用 rb_search_auxiliary 查找要
  删除的

结点
    {
        printf("key %d is not exist!\n");
        return root;
    }

    old = node;

    if (node->left && node->right)
    {
        node = node->right;
        while ((left = node->left) != NULL)
        {
            node = left;
        }
        child = node->right;
        parent = node->parent;
        color = node->color;

        if (child)
        {
            child->parent = parent;
        }
        if (parent)
        {
            if (parent->left == node)
            {
```

```
280.                    parent->left = child;
281.               }
282.               else
283.               {
284.                    parent->right = child;
285.               }
286.          }
287.          else
288.          {
289.               root = child;
290.          }
291.
292.          if (node->parent == old)
293.          {
294.               parent = node;
295.          }
296.
297.          node->parent = old->parent;
298.          node->color = old->color;
299.          node->right = old->right;
300.          node->left = old->left;
301.
302.          if (old->parent)
303.          {
304.               if (old->parent->left == old)
305.               {
306.                    old->parent->left = node;
307.               }
308.               else
309.               {
310.                    old->parent->right = node;
311.               }
312.          }
313.          else
314.          {
315.               root = node;
316.          }
317.
318.          old->left->parent = node;
319.          if (old->right)
320.          {
321.               old->right->parent = node;
322.          }
323.     }
324.     else
325.     {
326.          if (!node->left)
327.          {
328.               child = node->right;
329.          }
330.          else if (!node->right)
```

```
331.            {
332.                child = node->left;
333.            }
334.            parent = node->parent;
335.            color = node->color;
336.
337.            if (child)
338.            {
339.                child->parent = parent;
340.            }
341.            if (parent)
342.            {
343.                if (parent->left == node)
344.                {
345.                    parent->left = child;
346.                }
347.                else
348.                {
349.                    parent->right = child;
350.                }
351.            }
352.            else
353.            {
354.                root = child;
355.            }
356.        }
357.
358.        free(old);
359.
360.        if (color == BLACK)
361.        {
362.            root = rb_erase_rebalance(child, parent, root); //调用 rb_erase_rebalance 来恢复红
    黑树性
363.
364.质
365.        }
366.
367.        return root;
368.}
369.
370.
371.//七、红黑树的 4 种删除情况
372.//----------------------------------------------------------------
373.//红黑树修复删除的 4 种情况
374.//为了表示下述注释的方便，也为了让下述代码与我的俩篇文章相对应，
375.//x 表示要删除的结点，*other、w 表示兄弟结点，
376.//----------------------------------------------------------------
377.static rb_node_t* rb_erase_rebalance(rb_node_t *node, rb_node_t *parent, rb_node_t *roo
    t)
378.{
379.        rb_node_t *other, *o_left, *o_right;   //x 的兄弟*other，兄弟左孩子*o_left,*o_right
```

```
380.
381.     while ((!node || node->color == BLACK) && node != root)
382.     {
383.         if (parent->left == node)
384.         {
385.             other = parent->right;
386.             if (other->color == RED)   //情况 1：x 的兄弟 w 是红色的
387.             {
388.                 other->color = BLACK;
389.                 parent->color = RED;  //上俩行，改变颜色，w->黑、p[x]->红。
390.                 root = rb_rotate_left(parent, root);  //再对 p[x]做一次左旋
391.                 other = parent->right;  //x 的新兄弟 new w 是旋转之前 w 的某个孩子。其实
    就是左旋后
392.
393.的效果。
394.             }
395.             if ((!other->left || other->left->color == BLACK) &&
396.                 (!other->right || other->right->color == BLACK))
397.                             //情况 2：x 的兄弟 w 是黑色，且 w 的俩个孩子也
398.
399.都是黑色的
400.
401.             {                   //由于 w 和 w 的俩个孩子都是黑色的，则在 x 和 w 上得去掉一
    黑色，
402.                 other->color = RED;  //于是，兄弟 w 变为红色。
403.                 node = parent;    //p[x]为新结点 x
404.                 parent = node->parent;  //x<-p[x]
405.             }
406.             else                  //情况 3：x 的兄弟 w 是黑色的，
407.             {                     //且，w 的左孩子是红色，右孩子为黑色。
408.                 if (!other->right || other->right->color == BLACK)
409.                 {
410.                     if ((o_left = other->left))   //w 和其左孩子 left[w]，颜色交换。
411.                     {
412.                         o_left->color = BLACK;    //w 的左孩子变为由黑->红色
413.                     }
414.                     other->color = RED;          //w 由黑->红
415.                     root = rb_rotate_right(other, root);  //再对 w 进行右旋，从而红黑性质
    恢复。
416.                     other = parent->right;       //变化后的，父结点的右孩子，作为新的兄弟
    结点
417.
418.w。
419.                 }
420.                             //情况 4：x 的兄弟 w 是黑色的
421.
422.                 other->color = parent->color;  //把兄弟节点染成当前节点父节点的颜色。
423.                 parent->color = BLACK;  //把当前节点父节点染成黑色
424.                 if (other->right)      //且 w 的右孩子是红
425.                 {
426.                     other->right->color = BLACK;  //兄弟节点 w 右孩子染成黑色
```

```
427.                     }
428.                     root = rb_rotate_left(parent, root);  //并再做一次左旋
429.                     node = root;   //并把 x 置为根。
430.                     break;
431.                 }
432.             }
433.     //下述情况与上述情况，原理一致。分析略。
434.             else
435.             {
436.                 other = parent->left;
437.                 if (other->color == RED)
438.                 {
439.                     other->color = BLACK;
440.                     parent->color = RED;
441.                     root = rb_rotate_right(parent, root);
442.                     other = parent->left;
443.                 }
444.                 if ((!other->left || other->left->color == BLACK) &&
445.                     (!other->right || other->right->color == BLACK))
446.                 {
447.                     other->color = RED;
448.                     node = parent;
449.                     parent = node->parent;
450.                 }
451.                 else
452.                 {
453.                     if (!other->left || other->left->color == BLACK)
454.                     {
455.                         if ((o_right = other->right))
456.                         {
457.                             o_right->color = BLACK;
458.                         }
459.                         other->color = RED;
460.                         root = rb_rotate_left(other, root);
461.                         other = parent->left;
462.                     }
463.                     other->color = parent->color;
464.                     parent->color = BLACK;
465.                     if (other->left)
466.                     {
467.                         other->left->color = BLACK;
468.                     }
469.                     root = rb_rotate_right(parent, root);
470.                     node = root;
471.                     break;
472.                 }
473.             }
474.     }
475.
476.     if (node)
477.     {
```

```
478.            node->color = BLACK;  //最后将 node[上述步骤置为了根结点]，改为黑色。
479.        }
480.        return root;  //返回 root
481.}
482.
483.
484.//八、测试用例
485.//主函数
486.int main()
487.{
488.        int i, count = 100;
489.        key_t key;
490.        rb_node_t* root = NULL, *node = NULL;
491.
492.        srand(time(NULL));
493.        for (i = 1; i < count; ++i)
494.        {
495.            key = rand() % count;
496.            if ((root = rb_insert(key, i, root)))
497.            {
498.                printf("[i = %d] insert key %d success!\n", i, key);
499.            }
500.            else
501.            {
502.                printf("[i = %d] insert key %d error!\n", i, key);
503.                exit(-1);
504.            }
505.
506.            if ((node = rb_search(key, root)))
507.            {
508.                printf("[i = %d] search key %d success!\n", i, key);
509.            }
510.            else
511.            {
512.                printf("[i = %d] search key %d error!\n", i, key);
513.                exit(-1);
514.            }
515.            if (!(i % 10))
516.            {
517.                if ((root = rb_erase(key, root)))
518.                {
519.                    printf("[i = %d] erase key %d success\n", i, key);
520.                }
521.                else
522.                {
523.                    printf("[i = %d] erase key %d error\n", i, key);
524.                }
525.            }
526.        }
527.
528.        return 0;
```

529.}

ok，完。

后记：

一、欢迎任何人就此份源码，以及我的前述俩篇文章，进行讨论、提议。
但任何人，引用此份源码剖析，必须得注明作者本人 July 以及出处。
红黑树系列，已经写了三篇文章，相信，教你透彻了解红黑树的目的，应该达到了。
二、本文完整源码，请到此处下载：

http://download.csdn.net/source/2958890

红黑树的 c++完整实现源码

作者：July、saturnman。
时间：二零一一年三月二十九日。
出处：**http://blog.csdn.net/v_JULY_v**。

前言：

本人的原创作品红黑树系列文章，至此，已经写到第 5 篇了。虽然第三篇文章：**红黑树的 c 源码实现与剖析**，用 c 语言完整实现过红黑树，但个人感觉，代码还是不够清晰。特此，再奉献出一份 c++的完整实现源码，以飨读者。

此份 c++实现源码，代码紧凑了许多，也清晰了不少，同时采取 c++类实现的方式，代码也更容易维护以及重用。ok，有任何问题，欢迎指正。

版权声明

红黑树的 c++完整实现源码

本文包含红黑树 c++实现的完整源码，所有的解释都含在注释中，所有的有关红黑树的原理及各种插入、删除操作的情况，都已在本人的红黑树系列的前 4 篇文章中，一一阐述。且在此红黑树系列第五篇文章中：**红黑树从头至尾插入和删除结点的全程演示图**，把所有的插入、删除情况都一一展示尽了。

因此，有关红黑树的全部原理，请参考其它文章，重点可参考此文：**红黑树算法的实现与剖析**。因此，相关原理，本文不再赘述。

ok，以下，即是红黑树 c++实现的全部源码，先是 RBTree.h，然后是 RBTree.cpp。

RBTree.h

view plaincopy to clipboardprint?

```
1. //file RBTree.h
2. //written by saturnman，20101008。
3. //updated by July，20110329。
4. /*----------------------------------------------
5. 版权声明：
6. July 和 saturnman 对此份红黑树的 c++实现代码享有全部的版权，
7. 谢绝转载，侵权必究。
8. ----------------------------------------------*/
9. #ifndef _RB_TREE_H_
10.#define _RB_TREE_H_
11.#include<iostream>
```

```
12.#include<string>
13.#include<sstream>
14.#include<fstream>
15.using namespace std;
16.
17.template<class KEY,class U>
18.class RB_Tree
19.{
20.     private:
21.          RB_Tree(const RB_Tree& input){}
22.          const RB_Tree& operator=(const RB_Tree& input){}
23.     private:
24.          enum COLOR{RED,BLACK};
25.          class RB_Node
26.          {
27.          public:
28.               RB_Node()
29.               {
30.                    //RB_COLOR = BLACK;
31.                    right = NULL;
32.                    left = NULL;
33.                    parent = NULL;
34.               }
35.               COLOR RB_COLOR;
36.               RB_Node* right;
37.               RB_Node* left;
38.               RB_Node* parent;
39.               KEY key;
40.               U data;
41.          };
42.          public:
43.               RB_Tree()
44.               {
45.                    this->m_nullNode = new RB_Node();
46.                    this->m_root = m_nullNode;
47.                    this->m_nullNode->right = this->m_root;
48.                    this->m_nullNode->left = this->m_root;
49.                    this->m_nullNode->parent = this->m_root;
50.                    this->m_nullNode->RB_COLOR = BLACK;
51.               }
52.
53.               bool Empty()
54.               {
55.                    if(this->m_root == this->m_nullNode)
56.                    {
57.                         return true;
58.                    }
59.                    else
60.                    {
61.                         return false;
62.                    }
```

```
63.             }
64.
65.             //查找 key
66.             RB_Node* find(KEY key)
67.             {
68.                 RB_Node* index = m_root;
69.                 while(index != m_nullNode)
70.                 {
71.                     if(key<index->key)
72.                     {
73.                         index  = index->left;  //比当前的小，往左
74.                     }
75.                     else if(key>index->key)
76.                     {
77.                         index = index->right;  //比当前的大，往右
78.                     }
79.                     else
80.                     {
81.                         break;
82.                     }
83.                 }
84.                 return index;
85.             }
86.
87.             //-------------------------插入结点总操作----------------------------------
88.             //全部的工作，都在下述伪代码中：
89.             /*RB-INSERT(T, z)
90.             1  y ← nil[T]               // y 始终指向  x 的父结点。
91.             2  x ← root[T]              // x 指向当前树的根结点，
92.             3  while x ≠ nil[T]
93.             4     do y ← x
94.             5        if key[z] < key[x]          //向左，向右..
95.             6           then x ← left[x]
96.             7           else x ← right[x]   //为了找到合适的插入点，x 探路跟踪路径，直到 x 成为 NIL 为止。
97.             8  p[z] ← y         //y 置为  插入结点 z 的父结点。
98.             9  if y = nil[T]
99.             10     then root[T] ← z
100.             11     else if key[z] < key[y]
101.             12             then left[y] ← z
102.             13             else right[y] ← z     //此  8-13 行，置 z 相关的指针。
103.             14  left[z] ← nil[T]
104.             15  right[z] ← nil[T]           //设为空，
105.             16  color[z] ← RED             //将新插入的结点 z 作为红色
106.             17  RB-INSERT-FIXUP(T, z)
107.             */
108.             //因为将 z 着为红色，可能会违反某一红黑性质，
109.             //所以需要调用下面的 RB-INSERT-FIXUP(T, z)来保持红黑性质。
110.             bool Insert(KEY key,U data)
111.             {
112.                 RB_Node* insert_point = m_nullNode;
```

```
113.            RB_Node* index = m_root;
114.            while(index!=m_nullNode)
115.            {
116.                insert_point = index;
117.                if(key<index->key)
118.                {
119.                    index = index->left;
120.                }
121.                else if(key>index->key)
122.                {
123.                    index = index->right;
124.                }
125.                else
126.                {
127.                    return false;
128.                }
129.            }
130.            RB_Node* insert_node = new RB_Node();
131.            insert_node->key = key;
132.            insert_node->data = data;
133.            insert_node->RB_COLOR = RED;
134.            insert_node->right = m_nullNode;
135.            insert_node->left = m_nullNode;
136.            if(insert_point==m_nullNode) //如果插入的是一颗空树
137.            {
138.                m_root = insert_node;
139.                m_root->parent = m_nullNode;
140.                m_nullNode->left = m_root;
141.                m_nullNode->right = m_root;
142.                m_nullNode->parent = m_root;
143.            }
144.            else
145.            {
146.                if(key<insert_point->key)
147.                {
148.                    insert_point->left = insert_node;
149.                }
150.                else
151.                {
152.                    insert_point->right = insert_node;
153.                }
154.                insert_node->parent = insert_point;
155.            }
156.            InsertFixUp(insert_node);    //调用 InsertFixUp 修复红黑树性质。
157.        }
158.
159.        //----------------------插入结点性质修复-------------------------------
160.        //3 种插入情况，都在下面的伪代码中(未涉及到所有全部的插入情况)。
161.        /*
162.        RB-INSERT-FIXUP(T, z)
163.        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
*/
//然后的工作，就非常简单了，即把上述伪代码改写为下述的 c++代码：
void InsertFixUp(RB_Node* node)
{
    while(node->parent->RB_COLOR==RED)
    {
        if(node->parent==node->parent->parent->left)  //
        {
            RB_Node* uncle = node->parent->parent->right;
            if(uncle->RB_COLOR == RED)  //插入情况 1，z 的叔叔 y 是红色的。
            {
                node->parent->RB_COLOR = BLACK;
                uncle->RB_COLOR = BLACK;
                node->parent->parent->RB_COLOR = RED;
                node = node->parent->parent;
            }
            else if(uncle->RB_COLOR == BLACK )  //插入情况 2：z 的叔叔 y 是黑色的，。
            {
                if(node == node->parent->right) //且 z 是右孩子
                {
                    node = node->parent;
                    RotateLeft(node);
                }
                else            //插入情况 3：z 的叔叔 y 是黑色的，但 z 是左孩子。
                {
                    node->parent->RB_COLOR = BLACK;
                    node->parent->parent->RB_COLOR = RED;
                    RotateRight(node->parent->parent);
                }
            }
        }
        else //这部分是针对为插入情况 1 中，z 的父亲现在作为祖父的右孩子了的情况，而写的。
```

```
211.                         //15 else (same as then clause with "right" and "left" exchanged)
212.                     {
213.                         RB_Node* uncle = node->parent->parent->left;
214.                         if(uncle->RB_COLOR == RED)
215.                         {
216.                             node->parent->RB_COLOR = BLACK;
217.                             uncle->RB_COLOR = BLACK;
218.                             uncle->parent->RB_COLOR = RED;
219.                             node = node->parent->parent;
220.                         }
221.                         else if(uncle->RB_COLOR == BLACK)
222.                         {
223.                             if(node == node->parent->left)
224.                             {
225.                                 node = node->parent;
226.                                 RotateRight(node);    //与上述代码相比，左旋改为右旋
227.                             }
228.                             else
229.                             {
230.                                 node->parent->RB_COLOR = BLACK;
231.                                 node->parent->parent->RB_COLOR = RED;
232.                                 RotateLeft(node->parent->parent);  //右旋改为左旋，即
    可。
233.                             }
234.                         }
235.                     }
236.                 }
237.                 m_root->RB_COLOR = BLACK;
238.             }
239.
240.             //左旋代码实现
241.             bool RotateLeft(RB_Node* node)
242.             {
243.                 if(node==m_nullNode || node->right==m_nullNode)
244.                 {
245.                     return false;//can't rotate
246.                 }
247.                 RB_Node* lower_right = node->right;
248.                 lower_right->parent =  node->parent;
249.                 node->right=lower_right->left;
250.                 if(lower_right->left!=m_nullNode)
251.                 {
252.                     lower_right->left->parent = node;
253.                 }
254.                 if(node->parent==m_nullNode) //rotate node is root
255.                 {
256.                     m_root = lower_right;
257.                     m_nullNode->left = m_root;
258.                     m_nullNode->right= m_root;
259.                     //m_nullNode->parent = m_root;
260.                 }
```

```
261.                 else
262.                 {
263.                     if(node == node->parent->left)
264.                     {
265.                         node->parent->left = lower_right;
266.                     }
267.                     else
268.                     {
269.                         node->parent->right = lower_right;
270.                     }
271.                 }
272.                 node->parent = lower_right;
273.                 lower_right->left = node;
274.             }
275.
276.             //右旋代码实现
277.             bool RotateRight(RB_Node* node)
278.             {
279.                 if(node==m_nullNode || node->left==m_nullNode)
280.                 {
281.                     return false;//can't rotate
282.                 }
283.                 RB_Node* lower_left = node->left;
284.                 node->left = lower_left->right;
285.                 lower_left->parent = node->parent;
286.                 if(lower_left->right!=m_nullNode)
287.                 {
288.                     lower_left->right->parent = node;
289.                 }
290.                 if(node->parent == m_nullNode) //node is root
291.                 {
292.                     m_root = lower_left;
293.                     m_nullNode->left = m_root;
294.                     m_nullNode->right = m_root;
295.                     //m_nullNode->parent = m_root;
296.                 }
297.                 else
298.                 {
299.                     if(node==node->parent->right)
300.                     {
301.                         node->parent->right = lower_left;
302.                     }
303.                     else
304.                     {
305.                         node->parent->left = lower_left;
306.                     }
307.                 }
308.                 node->parent = lower_left;
309.                 lower_left->right = node;
310.             }
311.
```

```
312.            //-------------------------删除结点总操作---------------------------------
313.            //伪代码，不再贴出，详情，请参考此红黑树系列第二篇文章：
314.            //经典算法研究系列：五、红黑树算法的实现与剖析：
315.            //http://blog.csdn.net/v_JULY_v/archive/2010/12/31/6109153.aspx。
316.            bool Delete(KEY key)
317.            {
318.                RB_Node* delete_point = find(key);
319.                if(delete_point == m_nullNode)
320.                {
321.                    return false;
322.                }
323.                if(delete_point->left!=m_nullNode && delete_point->right!
   =m_nullNode)
324.                {
325.                    RB_Node* successor = InOrderSuccessor(delete_point);
326.                    delete_point->data = successor->data;
327.                    delete_point->key = successor->key;
328.                    delete_point = successor;
329.                }
330.                RB_Node* delete_point_child;
331.                if(delete_point->right!=m_nullNode)
332.                {
333.                    delete_point_child = delete_point->right;
334.                }
335.                else if(delete_point->left!=m_nullNode)
336.                {
337.                    delete_point_child = delete_point->left;
338.                }
339.                else
340.                {
341.                    delete_point_child = m_nullNode;
342.                }
343.                delete_point_child->parent = delete_point->parent;
344.                if(delete_point->parent==m_nullNode)//delete root node
345.                {
346.                    m_root = delete_point_child;
347.                    m_nullNode->parent = m_root;
348.                    m_nullNode->left = m_root;
349.                    m_nullNode->right = m_root;
350.                }
351.                else if(delete_point == delete_point->parent->right)
352.                {
353.                    delete_point->parent->right = delete_point_child;
354.                }
355.                else
356.                {
357.                    delete_point->parent->left = delete_point_child;
358.                }
359.                if(delete_point->RB_COLOR==BLACK && !
   (delete_point_child==m_nullNode && delete_point_child->parent==m_nullNode))
360.                {
```

```
361.                 DeleteFixUp(delete_point_child);
362.             }
363.             delete delete_point;
364.             return true;
365.         }
366.
367.         //---------------------删除结点性质修复----------------------------------
368.         //所有的工作，都在下述 23 行伪代码中：
369.         /*
370.         RB-DELETE-FIXUP(T, x)
371.         1 while x ≠ root[T] and color[x] = BLACK
372.         2     do if x = left[p[x]]
373.         3           then w ← right[p[x]]
374.         4                if color[w] = RED
375.         5                   then color[w] ← BLACK                    ? Case 1
376.         6                        color[p[x]] ← RED                   ? Case 1
377.         7                        LEFT-ROTATE(T, p[x])                ? Case 1
378.         8                        w ← right[p[x]]                     ? Case 1
379.         9                if color[left[w]] = BLACK and color[right[w]] = BLACK
380.         10                  then color[w] ← RED                      ? Case 2
381.         11                       x p[x]                              ? Case 2
382.         12                  else if color[right[w]] = BLACK
383.         13                            then color[left[w]] ← BLACK    ? Case 3
384.         14                                 color[w] ← RED            ? Case 3
385.         15                                 RIGHT-ROTATE(T, w)        ? Case 3
386.         16                                 w ← right[p[x]]           ? Case 3
387.         17                       color[w] ← color[p[x]]              ? Case 4
388.         18                       color[p[x]] ← BLACK                 ? Case 4
389.         19                       color[right[w]] ← BLACK             ? Case 4
390.         20                       LEFT-ROTATE(T, p[x])                ? Case 4
391.         21                       x ← root[T]                         ? Case 4
392.         22          else (same as then clause with "right" and "left" exchanged)
393.         23 color[x] ← BLACK
394.         */
395.         //接下来的工作，很简单，即把上述伪代码改写成 c++代码即可。
396.         void DeleteFixUp(RB_Node* node)
397.         {
398.             while(node!=m_root && node->RB_COLOR==BLACK)
399.             {
400.                 if(node == node->parent->left)
401.                 {
402.                     RB_Node* brother = node->parent->right;
403.                     if(brother->RB_COLOR==RED)   //情况 1：x 的兄弟 w 是红色的。
404.                     {
405.                         brother->RB_COLOR = BLACK;
406.                         node->parent->RB_COLOR = RED;
407.                         RotateLeft(node->parent);
408.                     }
409.                     else     //情况 2：x 的兄弟 w 是黑色的，
410.                     {
```

```
411.                                 if(brother->left->RB_COLOR == BLACK && brother-
    >right->RB_COLOR == BLACK)
412.                                     //且 w 的俩个孩子都是黑色的。
413.                                 {
414.                                     brother->RB_COLOR = RED;
415.                                     node = node->parent;
416.                                 }
417.                                 else if(brother->right->RB_COLOR == BLACK)
418.                                     //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）。
419.                                 {
420.                                     brother->RB_COLOR = RED;
421.                                     brother->left->RB_COLOR = BLACK;
422.                                     RotateRight(brother);
423.                                 }
424.                                 else if(brother->right->RB_COLOR == RED)
425.                                     //情况 4：x 的兄弟 w 是黑色的，且 w 的右孩子时红色的。

426.                                 {
427.                                     brother->RB_COLOR = node->parent->RB_COLOR;
428.                                     node->parent->RB_COLOR = BLACK;
429.                                     brother->right->RB_COLOR = BLACK;
430.                                     RotateLeft(node->parent);
431.                                     node = m_root;
432.                                 }
433.                             }
434.                         }
435.                         else  //下述情况针对上面的情况 1 中，node 作为右孩子而阐述的。
436.                             //22        else (same as then clause with "right" and "left" exchan
    ged)
437.                             //同样，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。
438.                         {
439.                             RB_Node* brother = node->parent->left;
440.                             if(brother->RB_COLOR == RED)
441.                             {
442.                                 brother->RB_COLOR = BLACK;
443.                                 node->parent->RB_COLOR = RED;
444.                                 RotateRight(node->parent);
445.                             }
446.                             else
447.                             {
448.                                 if(brother->left->RB_COLOR==BLACK && brother-
    >right->RB_COLOR == BLACK)
449.                                 {
450.                                     brother->RB_COLOR = RED;
451.                                     node = node->parent;
452.                                 }
453.                                 else if(brother->left->RB_COLOR==BLACK)
454.                                 {
455.                                     brother->RB_COLOR = RED;
```

```
456.                                    brother->right->RB_COLOR = BLACK;
457.                                    RotateLeft(brother);
458.                                }
459.                                else if(brother->left->RB_COLOR==RED)
460.                                {
461.                                    brother->RB_COLOR = node->parent->RB_COLOR;
462.                                    node->parent->RB_COLOR = BLACK;
463.                                    brother->left->RB_COLOR = BLACK;
464.                                    RotateRight(node->parent);
465.                                    node = m_root;
466.                                }
467.                            }
468.                        }
469.                    }
470.                    m_nullNode->parent = m_root;   //最后将 node 置为根结点，
471.                    node->RB_COLOR = BLACK;    //并改为黑色。
472.                }
473.
474.                //
475.                inline RB_Node* InOrderPredecessor(RB_Node* node)
476.                {
477.                    if(node==m_nullNode)       //null node has no predecessor
478.                    {
479.                        return m_nullNode;
480.                    }
481.                    RB_Node* result = node->left;     //get node's left child
482.                    while(result!=m_nullNode)         //try to find node's left subtree's right mo
    st node
483.                    {
484.                        if(result->right!=m_nullNode)
485.                        {
486.                            result = result->right;
487.                        }
488.                        else
489.                        {
490.                            break;
491.                        }
492.                    }          //after while loop result==null or result's right child is null
493.                    if(result==m_nullNode)
494.                    {
495.                        RB_Node* index = node->parent;
496.                        result = node;
497.                        while(index!=m_nullNode && result == index->left)
498.                        {
499.                            result = index;
500.                            index = index->parent;
501.                        }
502.                        result = index;         // first right parent or null
503.                    }
504.                    return result;
505.                }
```

```
506.
507.            //
508.            inline RB_Node* InOrderSuccessor(RB_Node* node)
509.            {
510.                if(node==m_nullNode)       //null node has no successor
511.                {
512.                    return m_nullNode;
513.                }
514.                RB_Node* result = node->right;   //get node's right node
515.                while(result!=m_nullNode)        //try to find node's right subtree's left mo
    st node
516.                {
517.                    if(result->left!=m_nullNode)
518.                    {
519.                        result = result->left;
520.                    }
521.                    else
522.                    {
523.                        break;
524.                    }
525.                }                       //after while loop result==null or result's left child is
    null
526.                if(result == m_nullNode)
527.                {
528.                    RB_Node* index = node->parent;
529.                    result = node;
530.                    while(index!=m_nullNode && result == index->right)
531.                    {
532.                        result = index;
533.                        index = index->parent;
534.                    }
535.                    result = index;        //first parent's left or null
536.                }
537.                return result;
538.            }
539.
540.            //debug
541.            void InOrderTraverse()
542.            {
543.                InOrderTraverse(m_root);
544.            }
545.            void CreateGraph(string filename)
546.            {
547.                //delete
548.            }
549.            void InOrderCreate(ofstream& file,RB_Node* node)
550.            {
551.                //delete
552.            }
553.            void InOrderTraverse(RB_Node* node)
554.            {
```

```
555.                    if(node==m_nullNode)
556.                    {
557.                        return;
558.                    }
559.                    else
560.                    {
561.                        InOrderTraverse(node->left);
562.                        cout<<node->key<<endl;
563.                        InOrderTraverse(node->right);
564.                    }
565.                }
566.                ~RB_Tree()
567.                {
568.                    clear(m_root);
569.                    delete m_nullNode;
570.                }
571.        private:
572.            // utility function for destructor to destruct object;
573.            void clear(RB_Node* node)
574.            {
575.                if(node==m_nullNode)
576.                {
577.                    return ;
578.                }
579.                else
580.                {
581.                    clear(node->left);
582.                    clear(node->right);
583.                    delete node;
584.                }
585.            }
586.        private:
587.            RB_Node *m_nullNode;
588.            RB_Node *m_root;
589.};
590.#endif /*_RB_TREE_H_*/
```

RBTree.cpp

[view plaincopy to clipboardprint?]

```
1. //file RBTree.cpp
2. //written by saturnman，20101008。
3. //updated by July，20110329。
4.
5. //所有的头文件都已补齐，现在您可以直接复制此份源码上机验证了（版权所有，侵权必究）。
6. //July、updated，2011.05.06。
7. #include<iostream>
8. #include<algorithm>
9. #include<iterator>
```

```
#include<vector>
#include<sstream>
#include"RBTree.h"    //如果.h 文件，和 cpp 文件放在一个文件里，此句去掉
using namespace std;

int main()
{
    RB_Tree<int,int> tree;
    vector<int> v;

    for(int i=0;i<20;++i)
    {
        v.push_back(i);
    }
    random_shuffle(v.begin(),v.end());
    copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
    cout<<endl;
    stringstream sstr;
    for(i=0;i<v.size();++i)
    {
        tree.Insert(v[i],i);
        cout<<"insert:"<<v[i]<<endl;   //添加结点
    }
    for(i=0;i<v.size();++i)
    {
        cout<<"Delete:"<<v[i]<<endl;
        tree.Delete(v[i]);          //删除结点
        tree.InOrderTraverse();
    }
    cout<<endl;
    tree.InOrderTraverse();
    return 0;
}
```

运行效果图（先是一一插入各结点，然后再删除所有的结点）：

```
4 19 0 17 10 12 8 11 18 1 6 16 7 14 9 3 13 2 5 15
insert:4
insert:19
insert:0
insert:17
insert:10
insert:12
insert:8
insert:11
insert:18
insert:1
insert:6
insert:16
insert:7
insert:14
insert:9
insert:3
insert:13
insert:2
insert:5
insert:15
Delete:4
0
1
搜狗拼音 半:
```

```
insert:5
insert:15
Delete:4
0
1
2
3
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Delete:19
0
搜狗拼音 半:
```

updated：

据网友鑫反馈，上述 c++源码虽说从上面的测试结果来看，没有问题。但程序还是有隐藏的 bug，下面，分两个步骤再来测试下此段源码：

1、首先在 RBTree.h 的最后里添加下述代码：

view plaincopy to clipboardprint?

```
1. public:
2.     void PrintTree()
3.     {
4.         _printNode(m_root);
5.     }
6. private:
7.     void _printNode(RB_Node *node)
8.     {
9.         if(node == NULL || node == m_nullNode) return;
10.
11.         if(node->parent == NULL || node->parent == m_nullNode){
12.             printf("root:%d\n", node->data);
13.         }else if(node->parent->left == node){
14.             printf("left:%d, parent:%d\n", node->data, node->parent->data);
15.         }else if(node->parent->right == node){
16.             printf("right:%d, parent:%d\n", node->data, node->parent->data);
17.         }
18.
19.         _printNode(node->left);
20.         _printNode(node->right);
21.     }
```

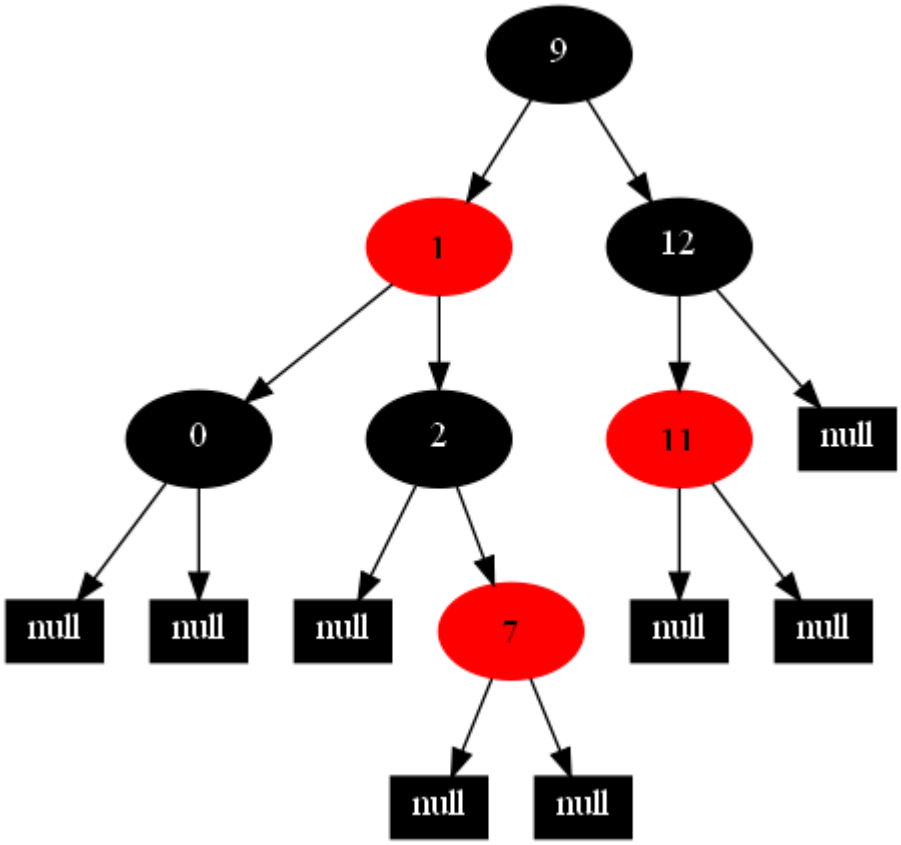
2、改写 RBTree.cpp 文件，如下：

view plaincopy to clipboardprint?

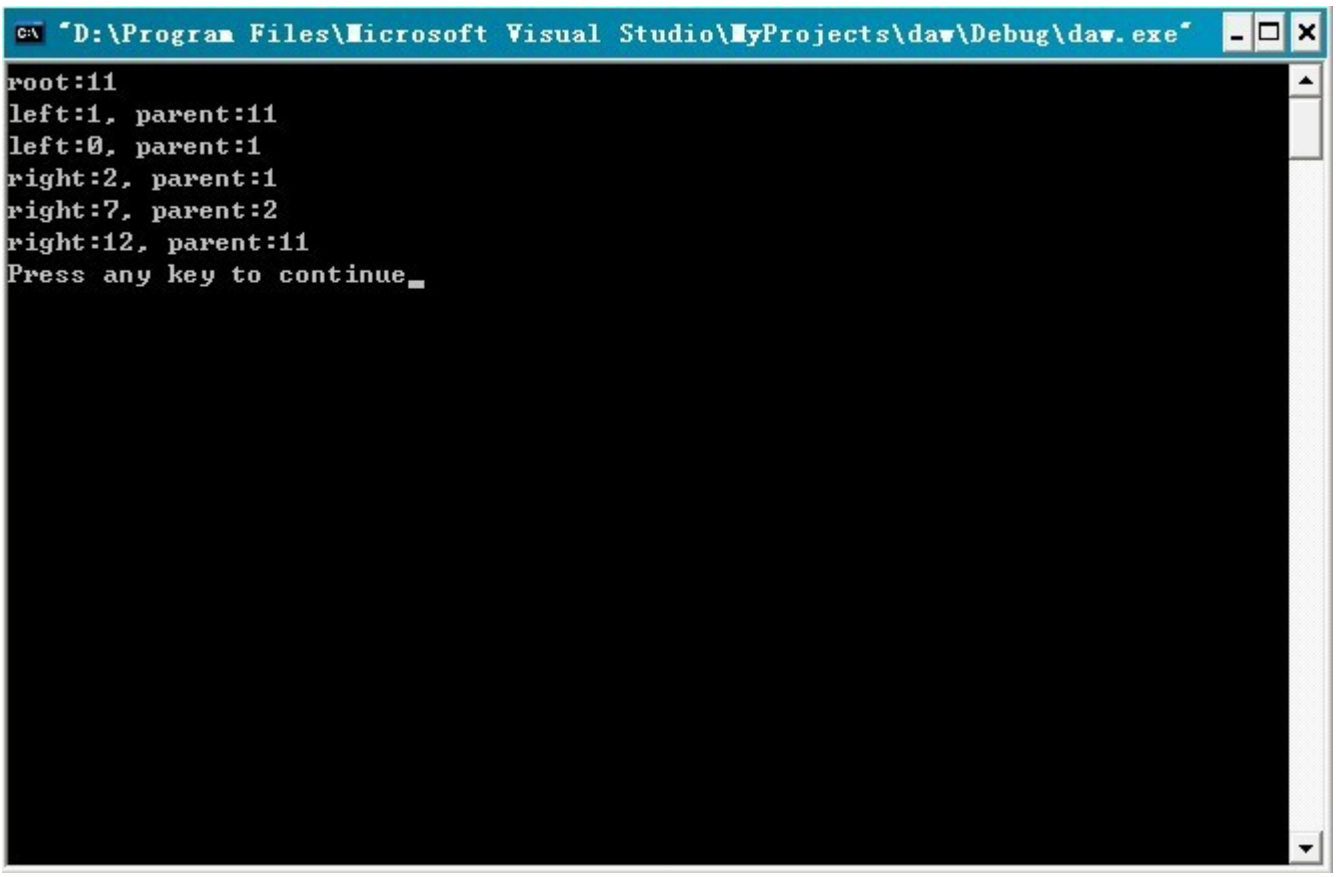
```
1. //file RBTree.cpp
2. //written by saturnman，20101008。
3. //updated by July，20110329。
4.
5. //所有的头文件都已补齐，现在您可以直接复制此份源码上机验证了（版权所有，侵权必究）。
6. //July、updated，2011.05.06。
7. #include<iostream>
8. #include<algorithm>
9. #include<iterator>
10.#include<vector>
11.#include<sstream>
12.//#include"RBTree.h"    //如果.h 文件，和 cpp 文件放在一个文件里，此句去掉
13.using namespace std;
14.
15.int main()
16.{
17.     RB_Tree<int,int> tree;
18.
19.     tree.Insert(12, 12);
20.     tree.Insert(1, 1);
21.     tree.Insert(9, 9);
22.     tree.Insert(2, 2);
23.     tree.Insert(0, 0);
24.     tree.Insert(11, 11);
25.     tree.Insert(7, 7);
26.
27.
28.     tree.Delete(9);
29.
30.     tree.PrintTree();
31.     /*vector<int> v;
32.
33.     for(int i=0;i<20;++i)
34.     {
35.          v.push_back(i);
36.     }
37.     random_shuffle(v.begin(),v.end());
38.     copy(v.begin(),v.end(),ostream_iterator<int>(cout," "));
39.     cout<<endl;
40.     stringstream sstr;
41.     for(i=0;i<v.size();++i)
42.     {
43.          tree.Insert(v[i],i);
44.          cout<<"insert:"<<v[i]<<endl;   //添加结点
45.     }
46.     for(i=0;i<v.size();++i)
47.     {
48.          cout<<"Delete:"<<v[i]<<endl;
49.          tree.Delete(v[i]);           //删除结点
50.          tree.InOrderTraverse();
```

```
51.     }
52.     cout<<endl;
53.     tree.InOrderTraverse();*/
54.     return 0;
55.}
```

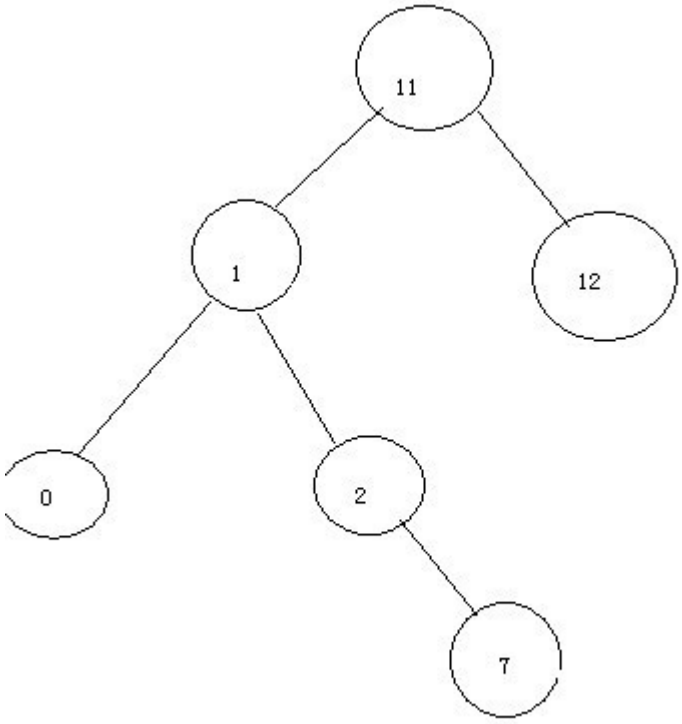
后经测试，结果，的确有误，即依次插入以下节点，12，1，9，0，2，11，7 后，红黑树变为如下：



然后删除根节点 9，经过上述程序运行后，运行结果，如下：



即上述运行结果，所对应的红黑树的状态如下（此时，红黑树已经不再平衡，存在的问题确实已经很明



显了）：

是的，如你所见，上述程序删除根节点 9 之后，正确的红黑树的状态应该为 7 代替根节点 9，7 成为新的根节点，且节点 7 着为黑色，而上述结果则是完全错误，红黑树已经完全不平衡。至此，终于发现，此 c++程序存在隐藏 bug 了。至于修正，则还得等一段时间。

说明：此程序的 bug 是经网友鑫指出的，同时，他还发现，网上不少的程序，都存在这个问题，比如这里：http://sd.csdn.net/a/20110506/297285.html 的红黑树的 flash 演示版本，也存在此类的问题。

已在原文下发表了以下评论：

> 很遗憾，经反复测试，红黑树的 flash 版本有问题（其它的暂还没发现问题）：
> http://www.cs.usfca.edu/~galles/visualization/flash.html。
> 如依次往插入这个序列，15,1,9,2,0,12,16,7,11,13,17,14，然后再删除根节点 9，严重的错误就出来了。上面的版本只是简单的一个步骤用 7 代替 9，成为根节点，然后把 7 节点着为黑色。树却没有后续调整，完全不平衡。
> 特此，把问题指出来，希望，这个红黑树的错误 flash 版本不致误导更多的人，同时，问题是朋友鑫提出的）。
> 我会记住这个问题，如果解决了，再发布在博客里。
> 后续：鑫指出：avl 树也有问题。
> July、结构之法 算法之道 博主。
> 2011.05.07。

这也让我更加的鉴定了此信念（虽说，对我的 blog 并无好处）：我的直觉是，不要轻易相信任何人的言论，任何人的文章。

希望，我可以发现潜藏在本 blog 中的任一个程序或文章里的所有 bug。任何人，有任何问题，也欢迎随时指正。再次感谢网友鑫的指正，谢谢。

同时，我会尽力修正本文程序中的 bug 的。谢谢。

updated && 修正：

本程序没有任何问题。有一点非常之重要，之前就是因为未意识到而造成上述错觉，即：**红黑树并非严格意义上的二叉查找树，它只要满足它本身的五点性质即可，不要求严格平衡**。所以，上面的例子中，12，1，9，0，2，11，7，然后删除根结点 9，只要着色适当，同样不违反红黑树的五点性质。所以，结论是，我庸人自扰了，sorry。

还是这句话，有任何问题，欢迎任何人提出或指正。

红黑树插入和删除结点的全程演示

作者：July、saturnman。
时间：二零一一年三月二十八日。
出处：**http://blog.csdn.net/v_JULY_v**。

引言：

目前国内图书市场上，抑或网上讲解红黑树的资料层次不齐，混乱不清，没有一个完整而统一的阐述。而本人的红黑树系列四篇文章（详见文末的参考文献），虽然从头至尾，讲的有根有据，层次清晰，然距离读者真正做到红黑树了然于胸，则还缺点什么。

而我们知道，即便在经典的算法导论一书上，也没有把所有的插入、删除情况一一道尽，直接导致了不少读者的迷惑，而我的红黑树系列第 4 篇文章：**一步一图一代码，一定要让你真正彻底明白红黑树**，虽然早已把所有的插入、删除情况都一一道尽了，但也缺了点东西。

缺点什么东西列?对了，缺的就是一个完完整整的，包含所有插入、删除情况全部过程的全程演示图，即缺一个例子，缺一个完整的图来从头至尾阐述这一切。

ok，本文，即以 40 幅图来全程演示此红黑树的所有插入，和删除情况。相信，一定会对您理解红黑树有所帮助。

话不絮烦，下面，本文便以此篇文章：一步一图一代码，一定要让你真正彻底明白红黑树为纲，从插入一个结点到最后插入全部结点，再到后来一个一个把结点全部删除的情况一一阐述。

由于为了有个完整统一，红黑树插入和删除情况在此合作成一篇文章。同时，由于本人的红黑树系

列的四篇文章已经把红黑树的插入、删除情况都一一详尽的阐述过 了，因此，有关红黑树的原理，本文不再赘述，只侧重于用图来一一全程演示结点的插入和删除情况。有任何问题，欢迎指正。

红黑树插入情况全过程演示

通过本人的红黑树系列第 4 篇文章，我们已经知道，**红黑树的所有插入情况有以下五种：**

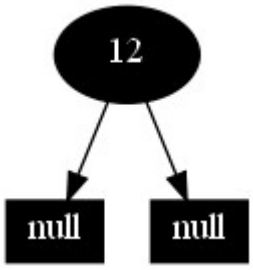
情形 1: 新节点 N 位于树的根上，没有父节点
情形 2: 新节点的父节点 P 是黑色
情形 3:父节点 P、叔叔节点 U，都为红色，
[对应第二篇文章中，的情况 1：z 的叔叔是红色的。]
情形 4: 父节点 P 是红色，叔叔节点 U 是黑色或 NIL;
插入节点 N 是其父节点 P 的右孩子，而父节点 P 又是其父节点的左孩子。
[对应我第二篇文章中，的情况 2：z 的叔叔是黑色的，且 z 是右孩子]
情形 5: 父节点 P 是红色，而叔父节点 U 是黑色或 NIL，
要插入的节点 N 是其父节点的左孩子，而父节点 P 又是其父 G 的左孩子。
[对应我第二篇文章中，情况 3：z 的叔叔是黑色的，且 z 是左孩子。]

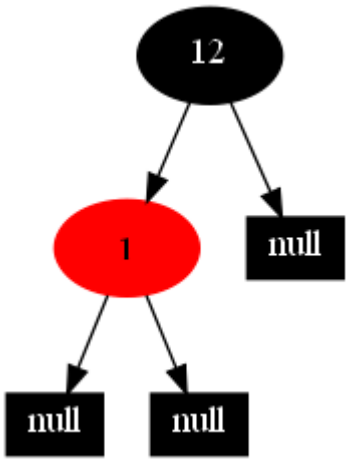
详细，可参考此红黑树系列第 4 篇文章：**一步一图一代码，一定要让你真正彻底明白红黑树**。

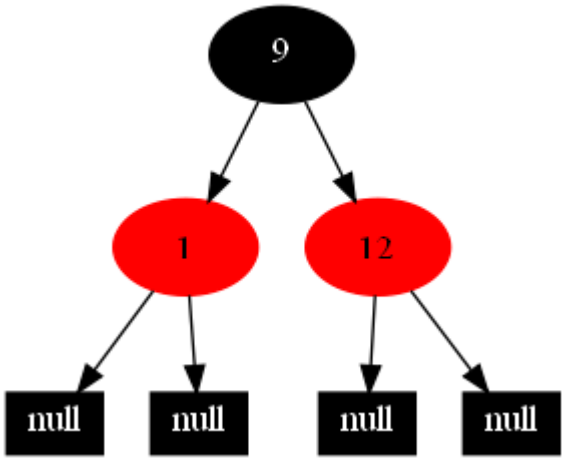
首先，各个结点插入与以上的各种插入情况，一一对应起来，如图：

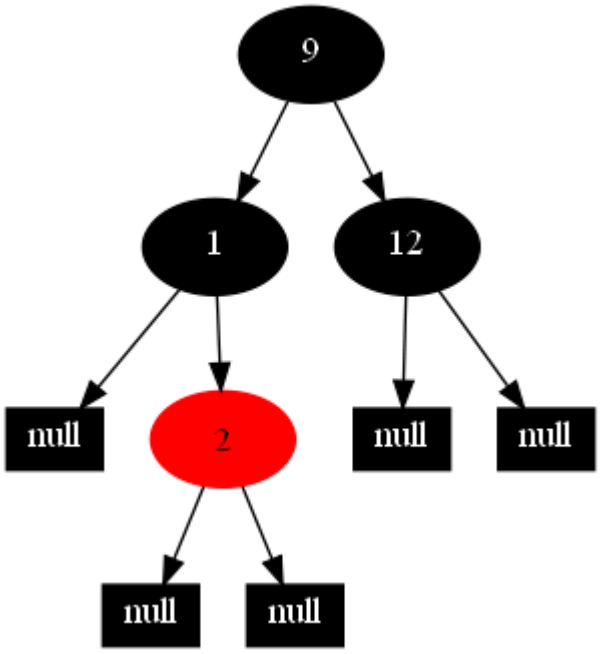
一一插入各个结点	12	1	9	2	0	11	7	19	4	15	18	5	14	13	10	16	6	3	8	17
对应的插入情况	1	2	1	3	2	2	3	2	4	2	4	3	3	5	2	3	4	2	3	4

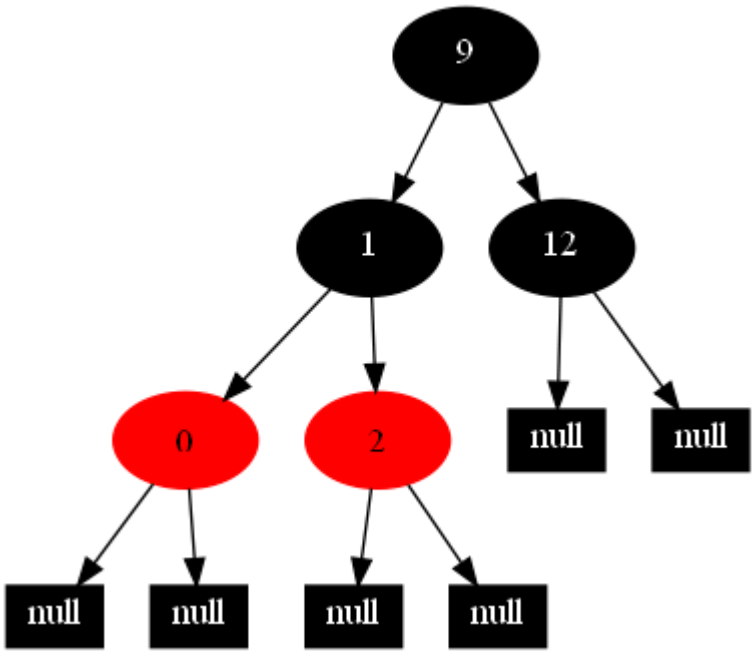
以下的 20 张图，是依次插入这些结点：12 1 9 2 0 11 7 19 4 15 18 5 14 13 10 16 6 3 8 17 的全程演示图，已经把所有的 5 种插入情况，都全部涉及到了：

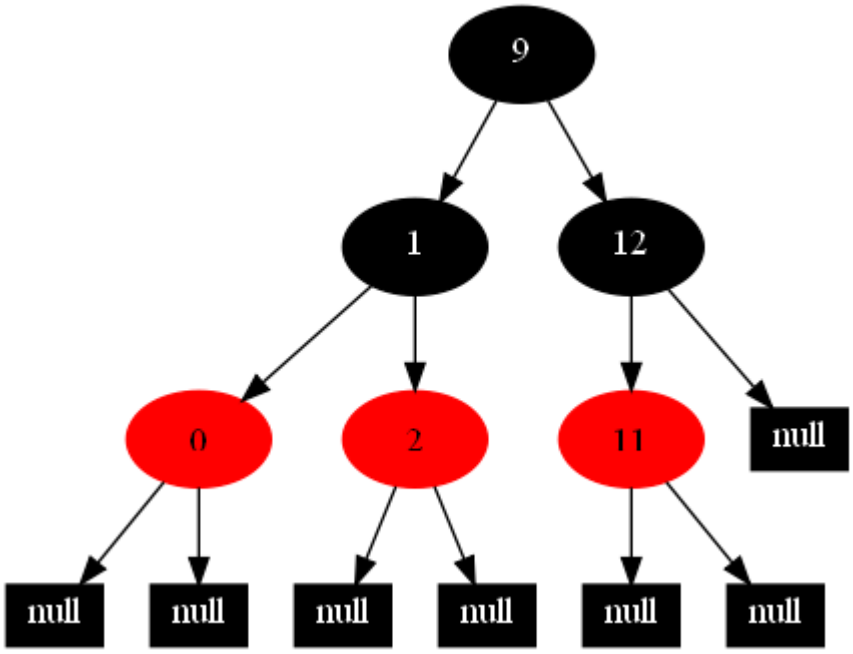


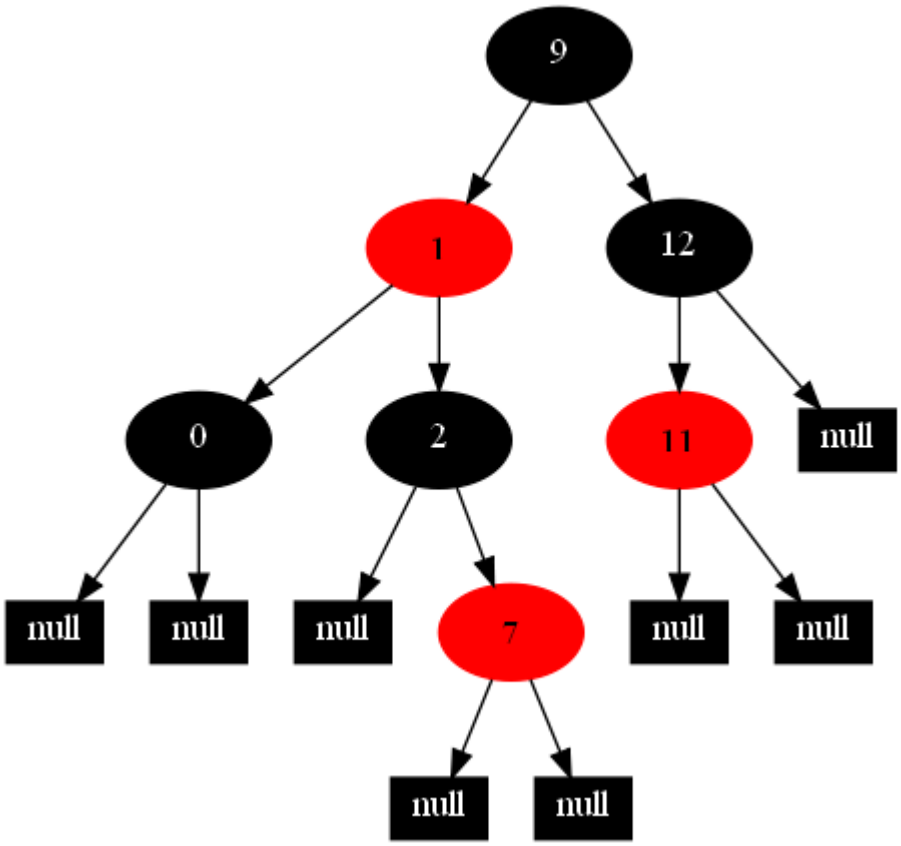

12
1
null
null
null

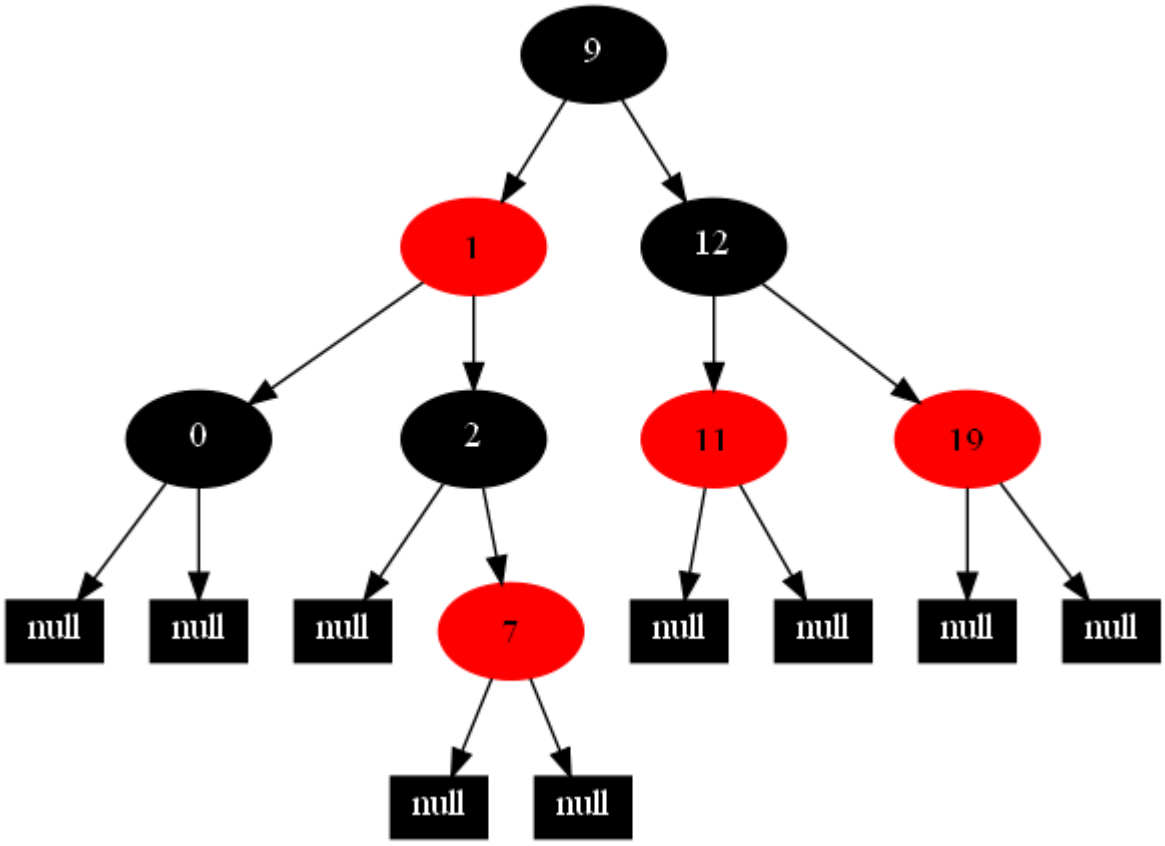

9
1
12
null
null
null
null

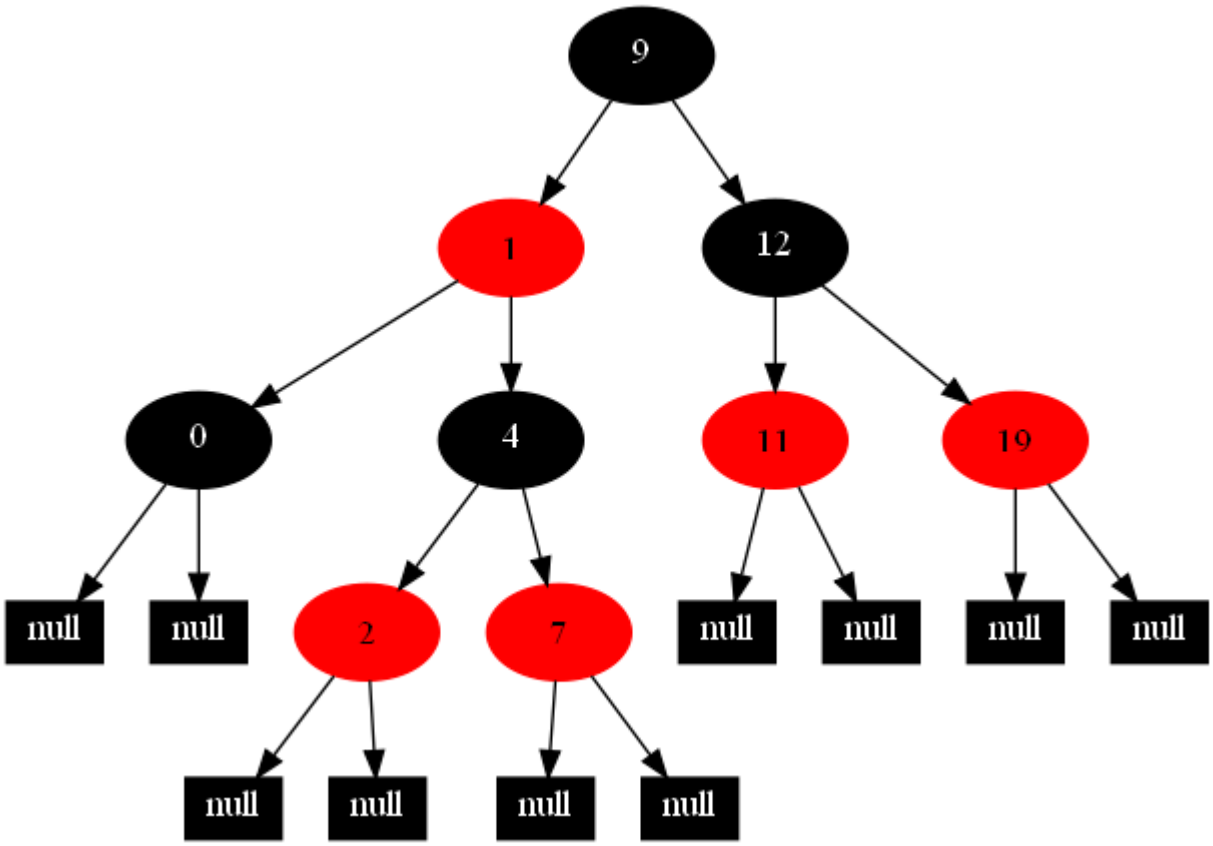

9
1
12
null
2
null
null
null
null

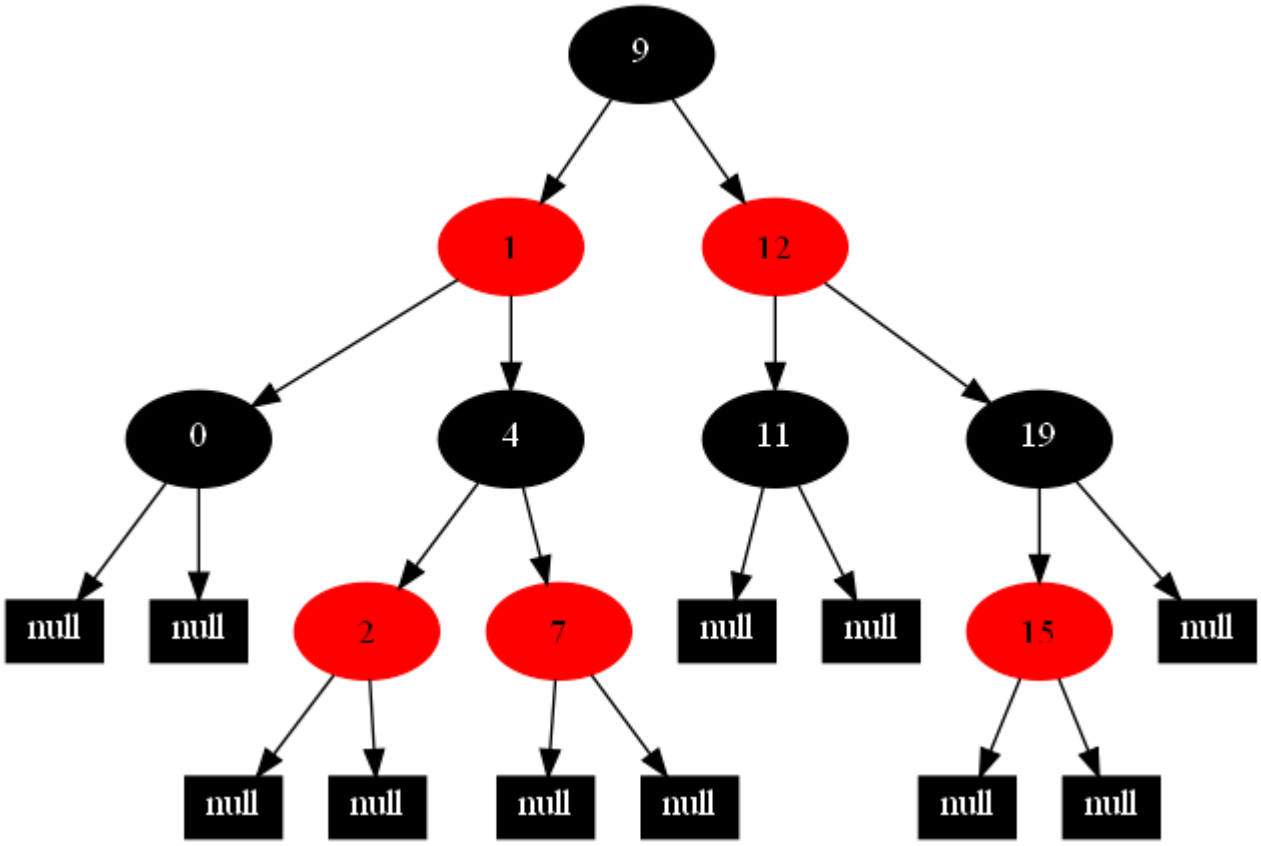

9
1
12
0
2
null
null
null
null
null
null

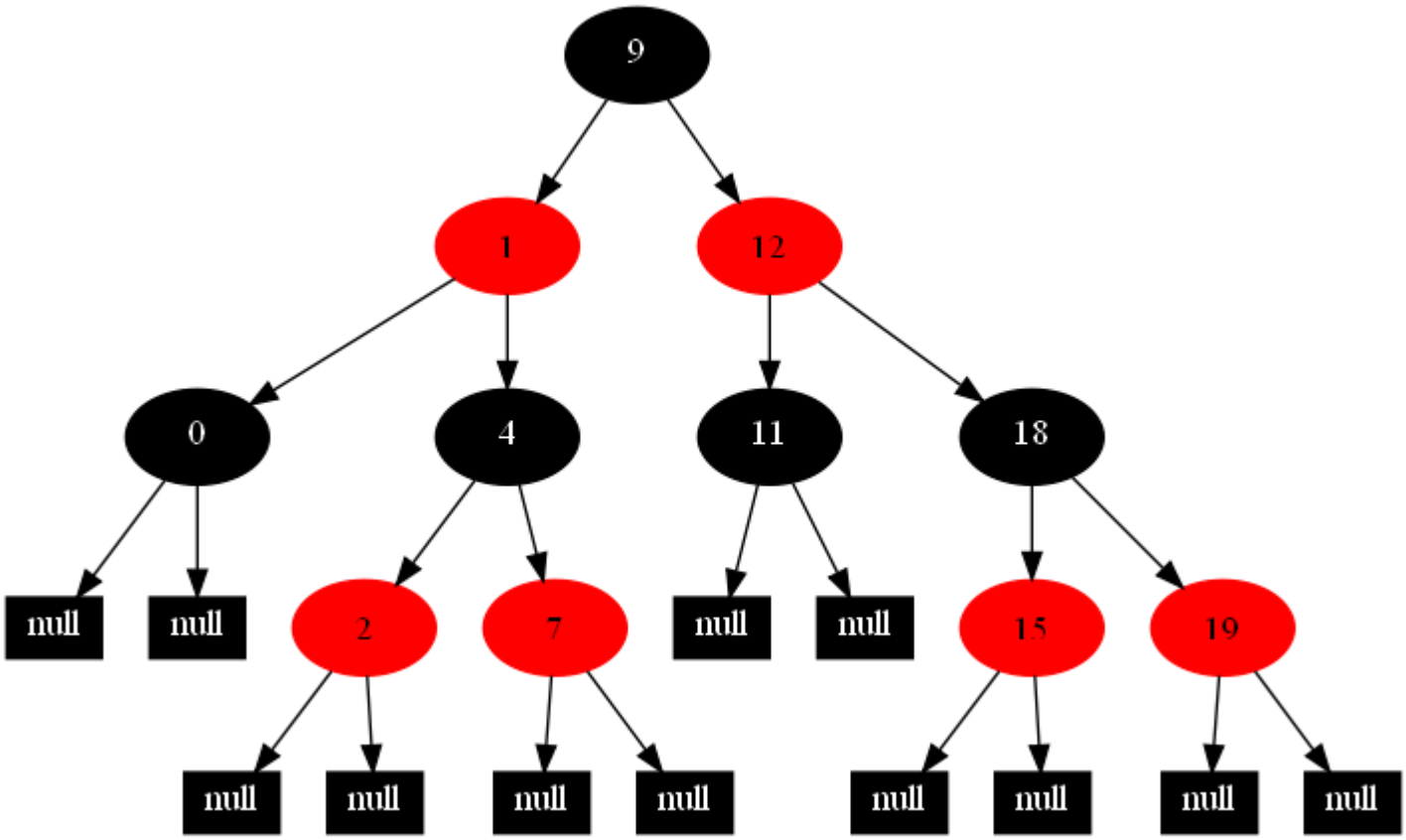

9
1
12
0
2
11
null
null
null
null
null
null
null

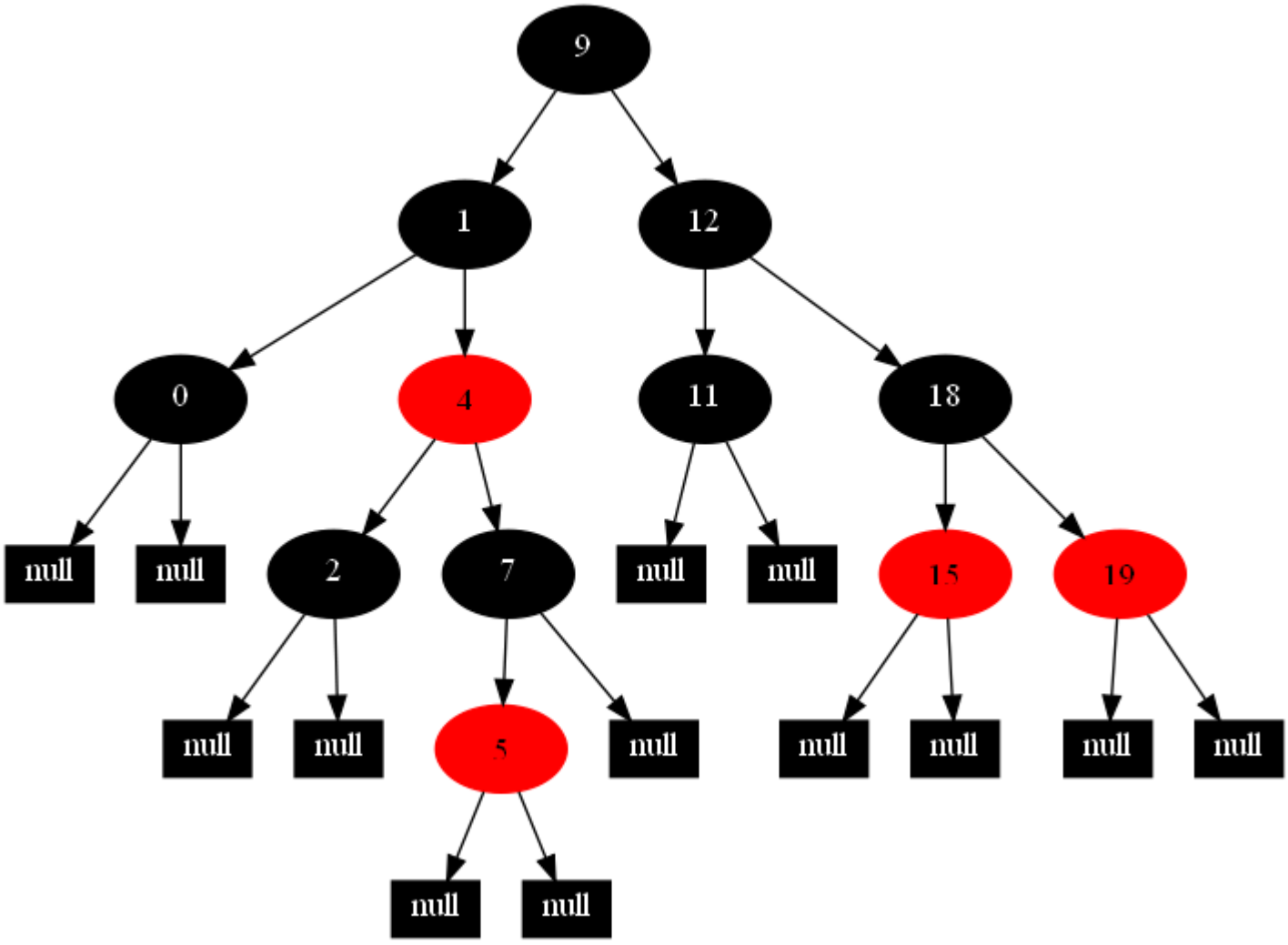

9
1
12
0
2
11
null
null
null
null
null
null
7
null
null

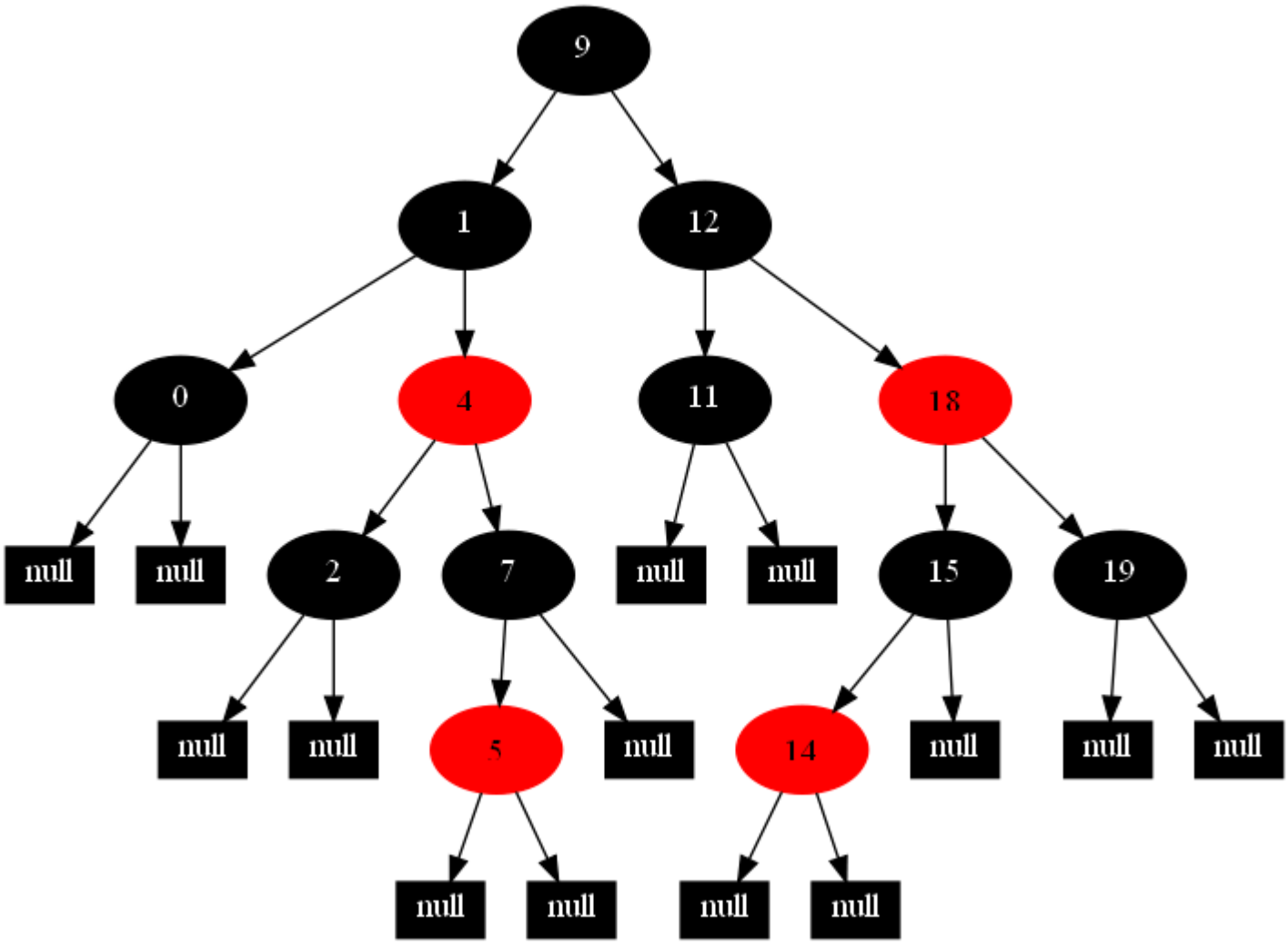

9
1
12
0
2
11
19
null
null
null
null
null
null
null
null
7
null
null

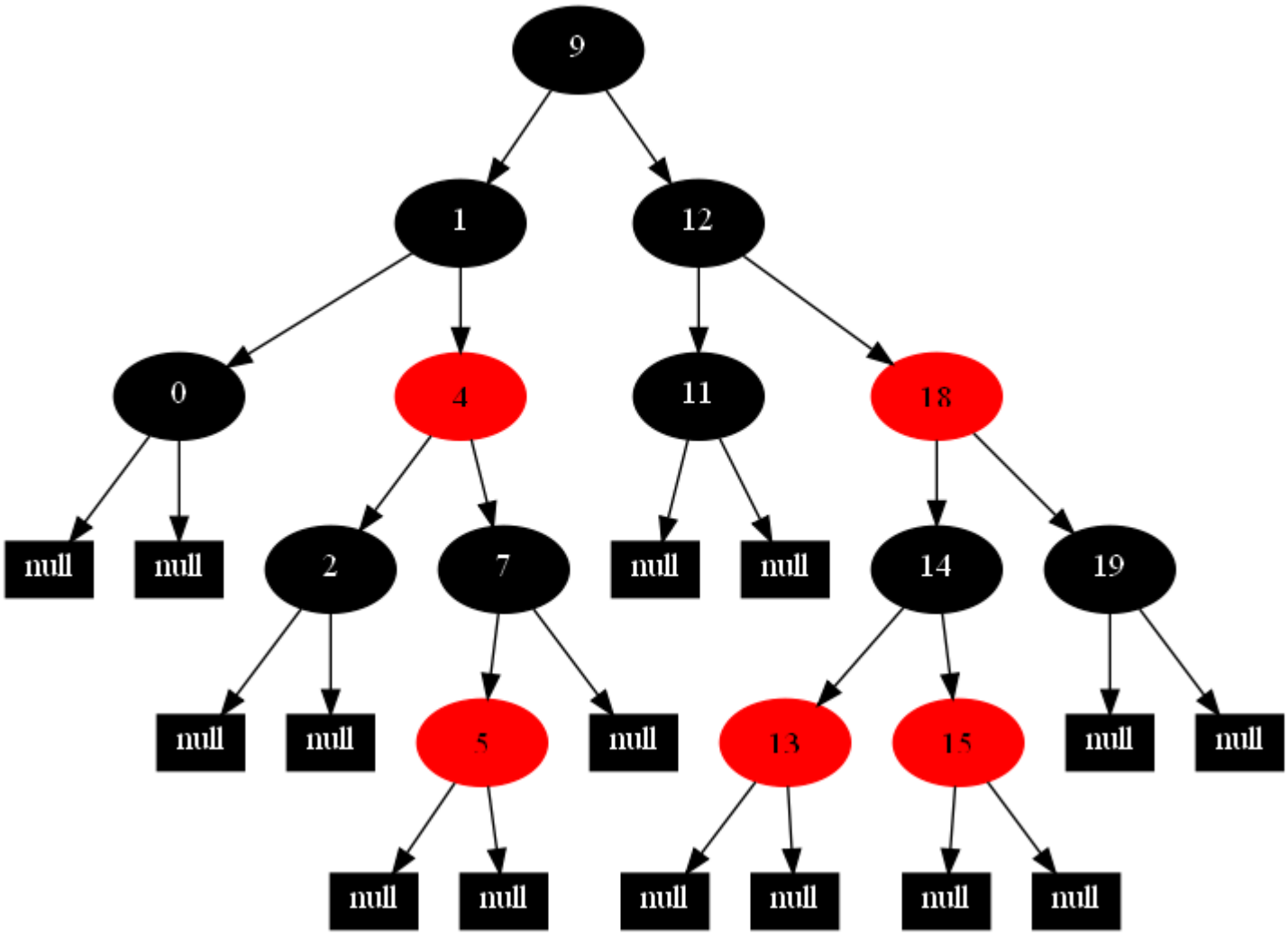

9
1
12
0
4
11
19
null
null
2
7
null
null
null
null
null
null
null
null

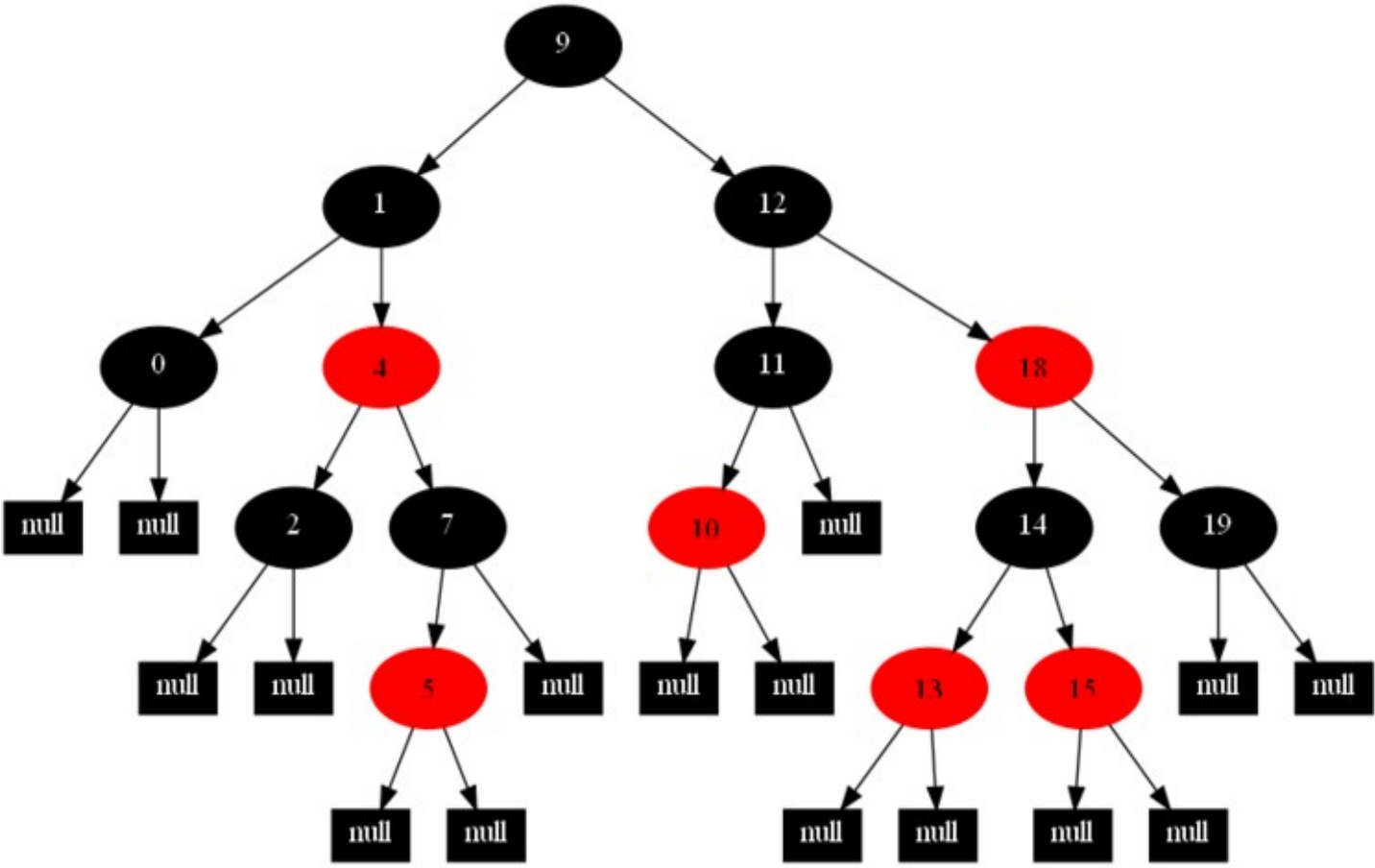

9
1
12
0
4
11
19
null
null
2
7
null
null
15
null
null
null
null
null
null
null
null

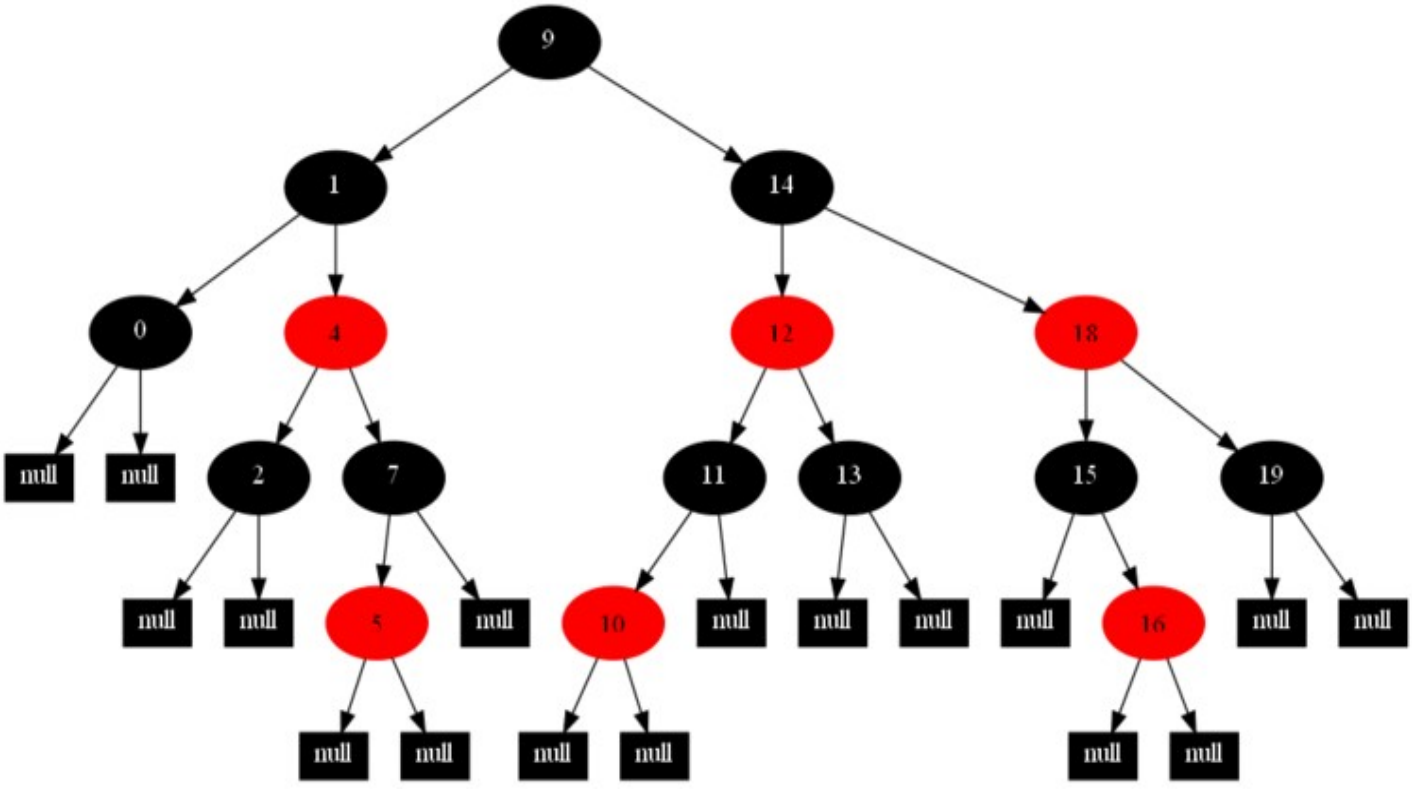

9
1
12
0
4
11
18
null
null
2
7
null
null
15
19
null
null
null
null
null
null
null
null

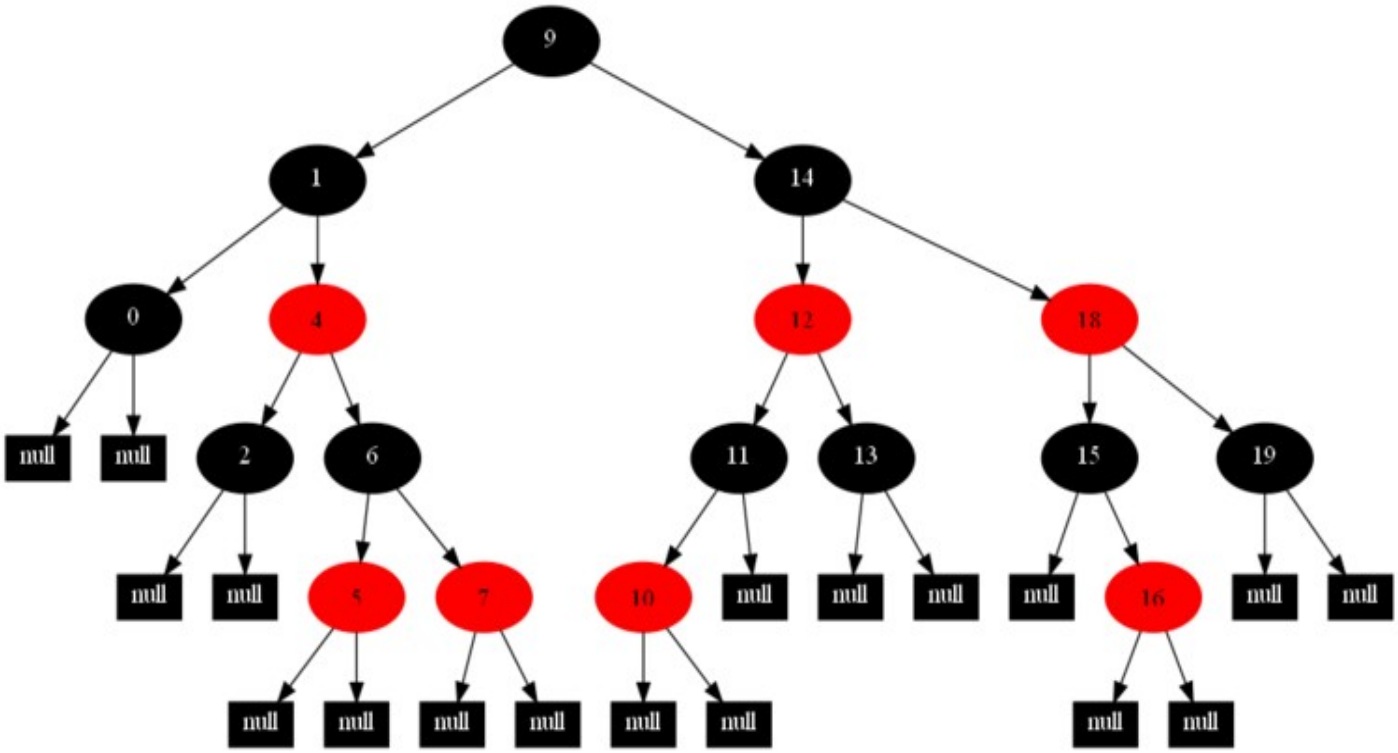

9
1
12
0
4
11
18
null
null
2
7
null
null
15
19
null
null
5
null
null
null
null
null
null
null
null

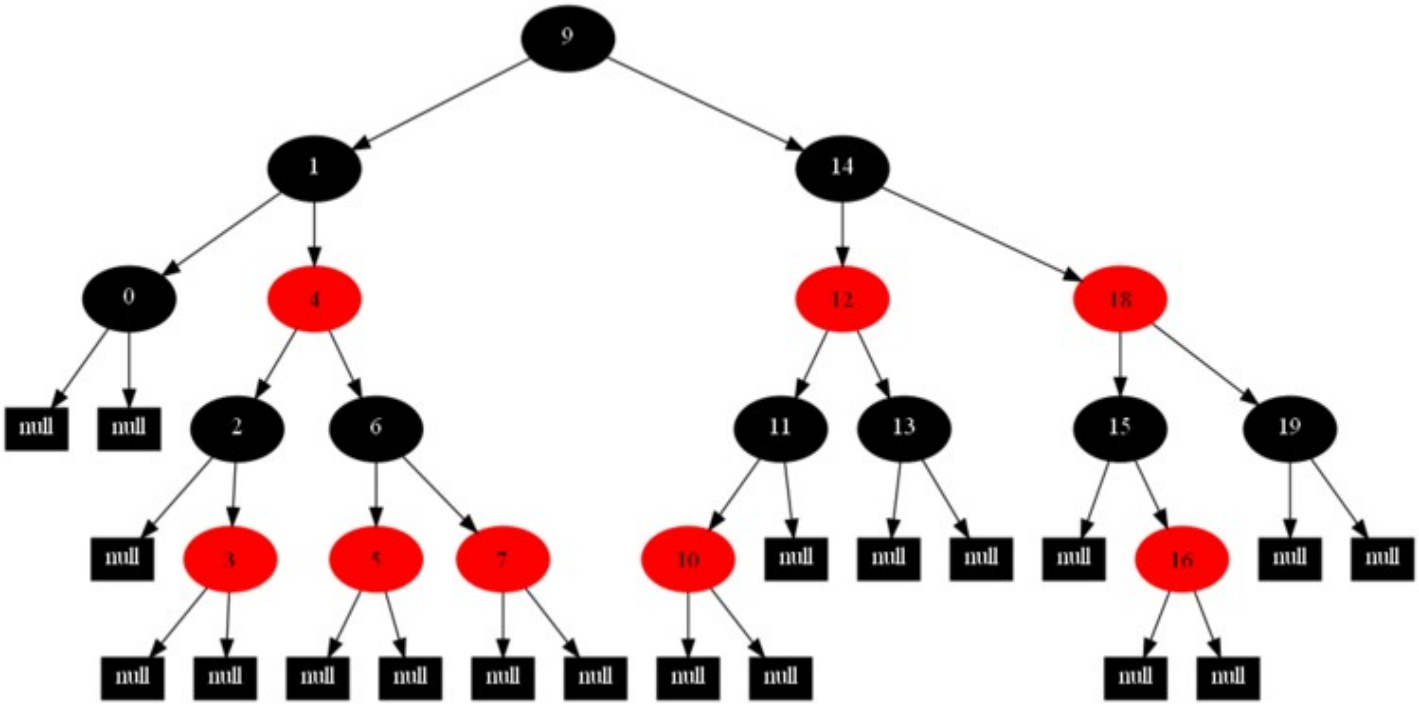

9
1
12
0
4
11
18
null
null
2
7
null
null
15
19
null
null
5
null
14
null
null
null
null
null
null
null

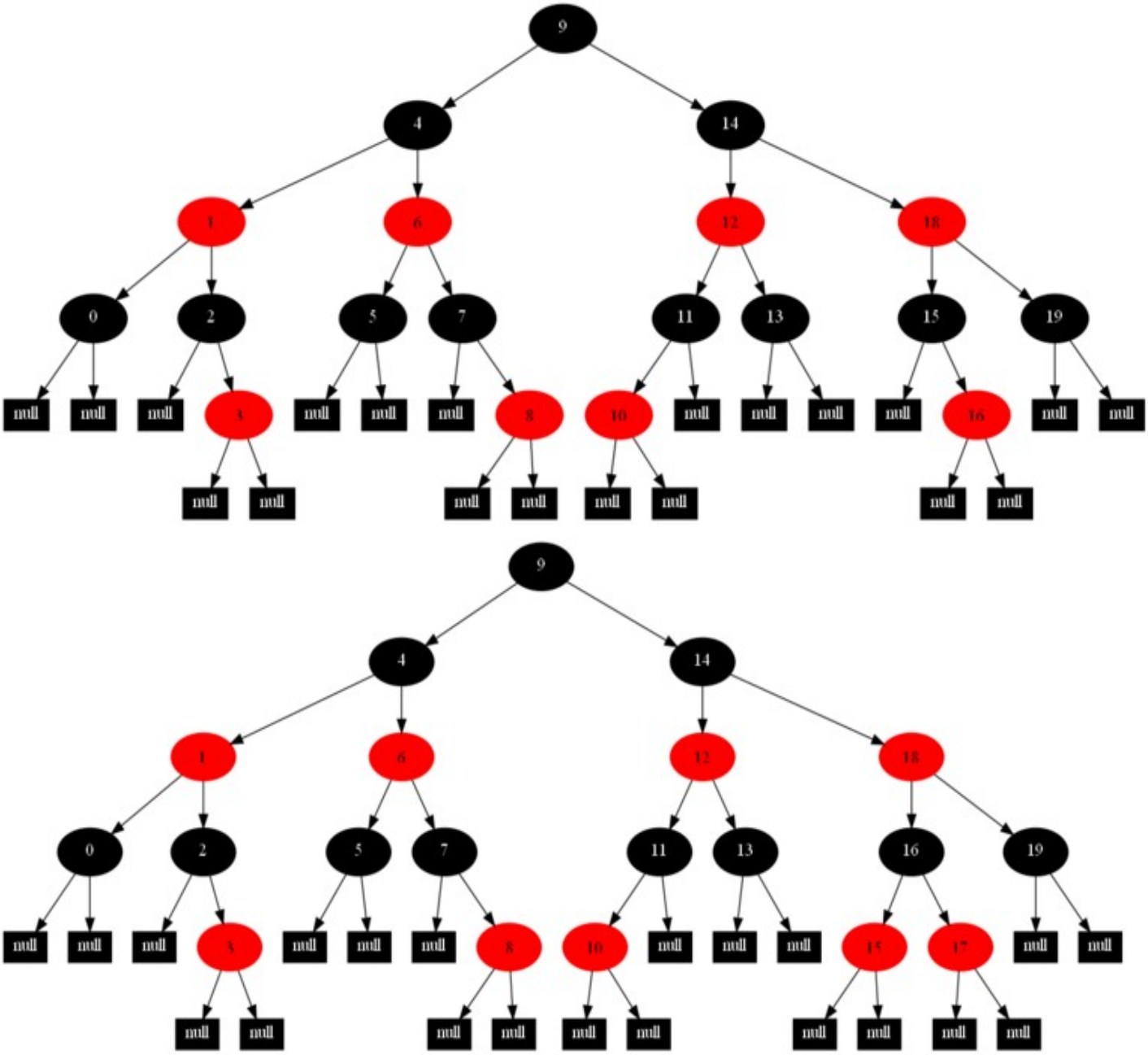

9
1
12
0
4
11
18
null
null
2
7
null
null
14
19
null
null
5
null
13
15
null
null
null
null
null
null
null
null


9
1
12
0
4
11
18
null
null
2
7
10
null
14
19
null
null
5
null
null
null
13
15
null
null
null
null
null
null


9
1
14
0
4
12
18
null
null
2
7
11
13
15
19
null
null
5
null
10
null
null
null
null
16
null
null
null
null
null
null
null
null
null
null


9
1
14
0
4
12
18
null
null
2
6
11
13
15
19
null
null
5
7
10
null
null
16
null
null
null
null
null
null
null
null
null
null


9
1
14
0
4
12
18
null
null
2
6
11
13
15
19
null
3
5
7
10
null
null
null
16
null
null
null
null
null
null
null
null
null
null
null
null
null
null



红黑树的一一插入各结点：12 1 9 2 0 11 7 19 4 15 18 5 14 13 10 16 6 3 8 17 的全程演示图完。

红黑树删除情况全过程演示

红黑树的所有删除情况，如下：

情况 1: N 是新的根。
情形 2：兄弟节点 S 是红色
[对应我第二篇文章中，情况 1：x 的兄弟 w 是红色的。]
情况 3: 兄弟节点 S 是黑色的，且 S 的俩个儿子都是黑色的。但 N 的父节点 P，是黑色。
[对应我第二篇文章中，情况 2：x 的兄弟 w 是黑色的，且兄弟 w 的俩个儿子都是黑色的。
(这里，N 的父节点 P 为黑)]
情况 4: 兄弟节点 S 是黑色的、S 的儿子也都是黑色的，但是 N 的父亲 P，是红色。
[还是对应我第二篇文章中，情况 2：x 的兄弟 w 是黑色的，且 w 的俩个孩子都是黑色的。
(这里，N 的父节点 P 为红)]
情况 5: 兄弟 S 为黑色，S 的左儿子是红色，S 的右儿子是黑色，而 N 是它父亲的左儿子。
//此种情况，最后转化到下面的情况 6。

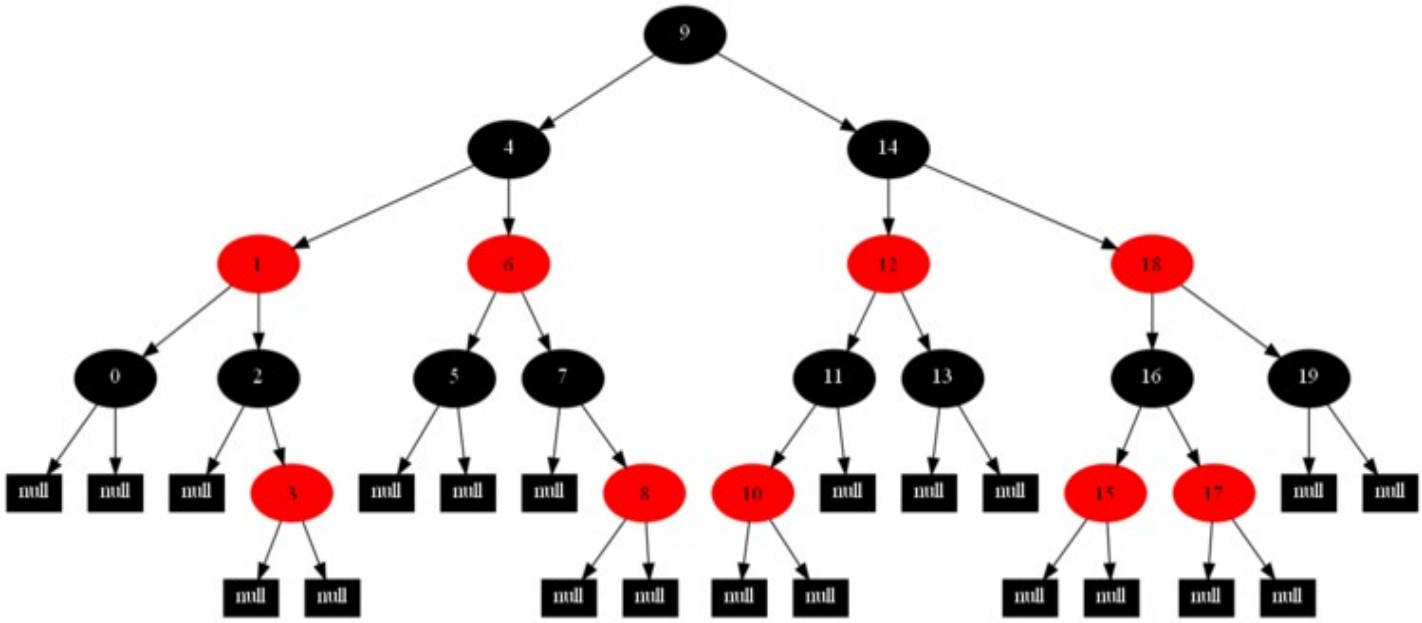
[对应我第二篇文章中，情况 3：x 的兄弟 w 是黑色的，w 的左孩子是红色，w 的右孩子是黑色。]
情况 6: 兄弟节点 S 是黑色，S 的右儿子是红色，而 N 是它父亲的左儿子。
[对应我第二篇文章中，情况 4:x 的兄弟 w 是黑色的，且 w 的右孩子时红色的。]

接下来，便是一一删除这些点 12 1 9 2 0 11 7 19 4 15 18 5 14 13 10 16 6 3 8 17 为例，即，红黑树删除情况全程演示：

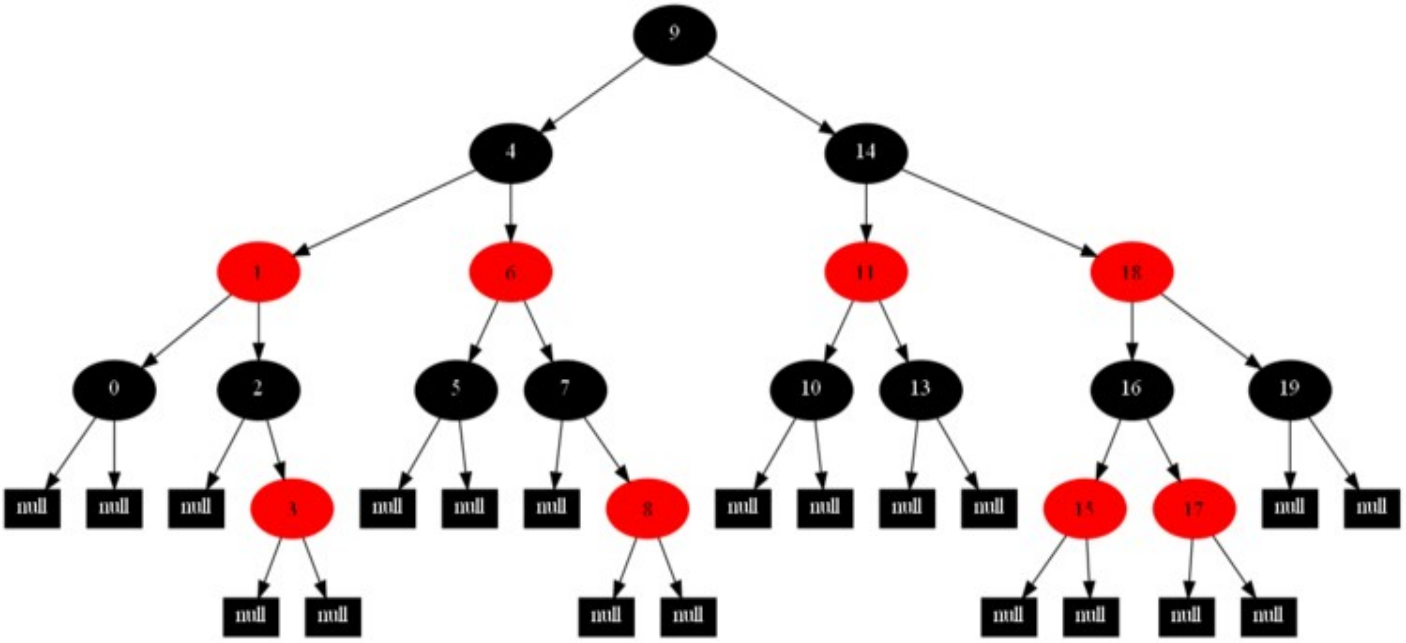
各个结点删除与以上的六种情况，一一对应起来，如图：

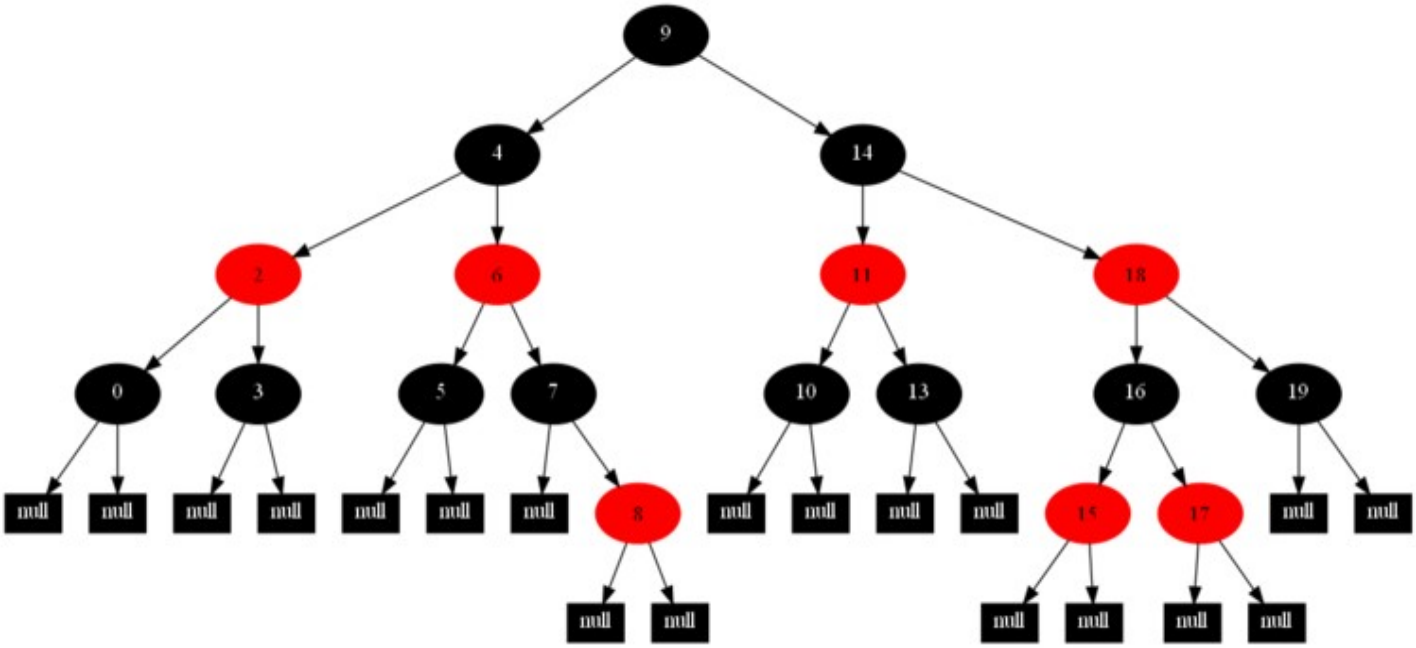
一一删除各个结点	12	1	9	2	0	11	7	19	4	15	18	5	14	13	10	16	6	3	8	17
对应的删除情况	2	2	1	2	4	2	4	6	6	5	4	3	3	3	1	1	1	3	1	1

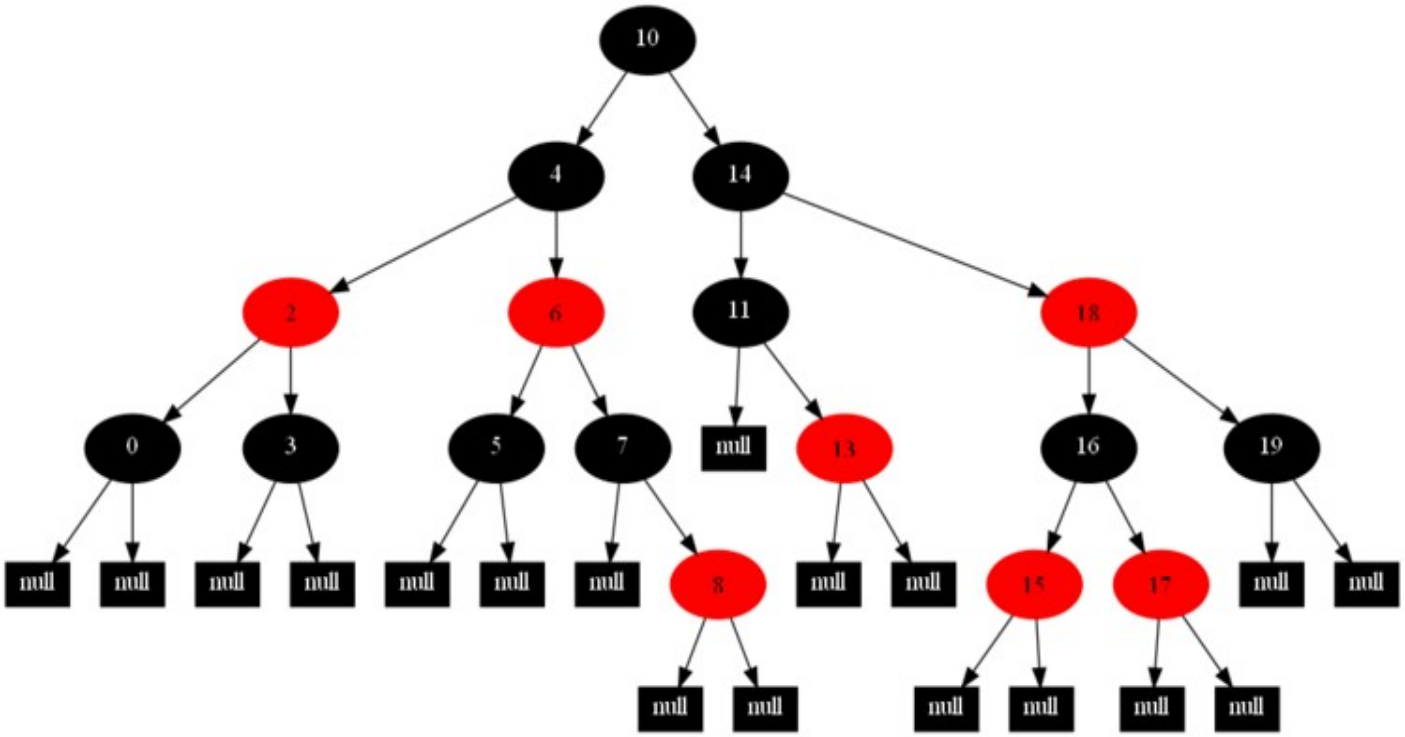
首先，插入 12 1 9 2 0 11 7 19 4 15 18 5 14 13 10 16 6 3 8 17 结点后，形成的红黑树为：

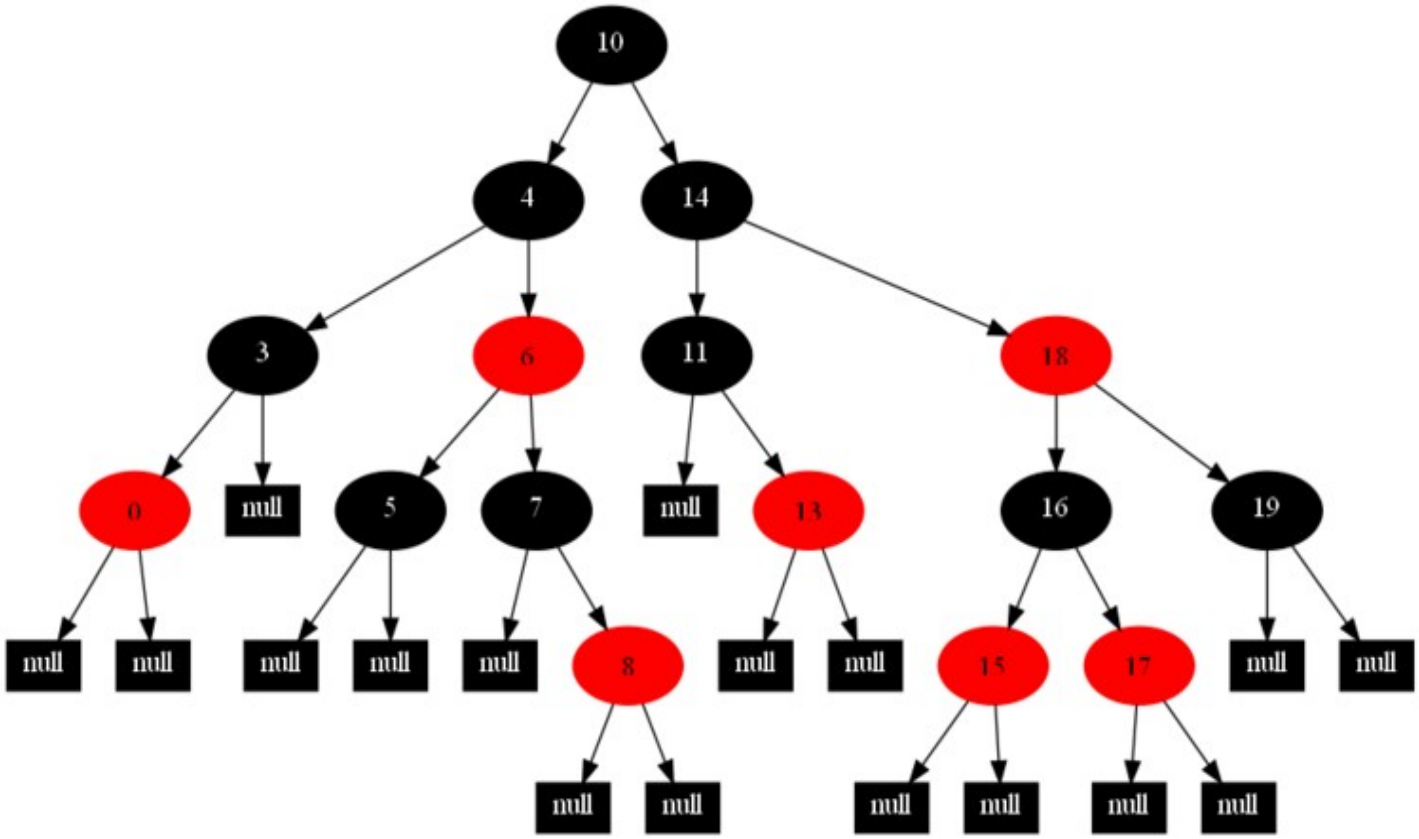


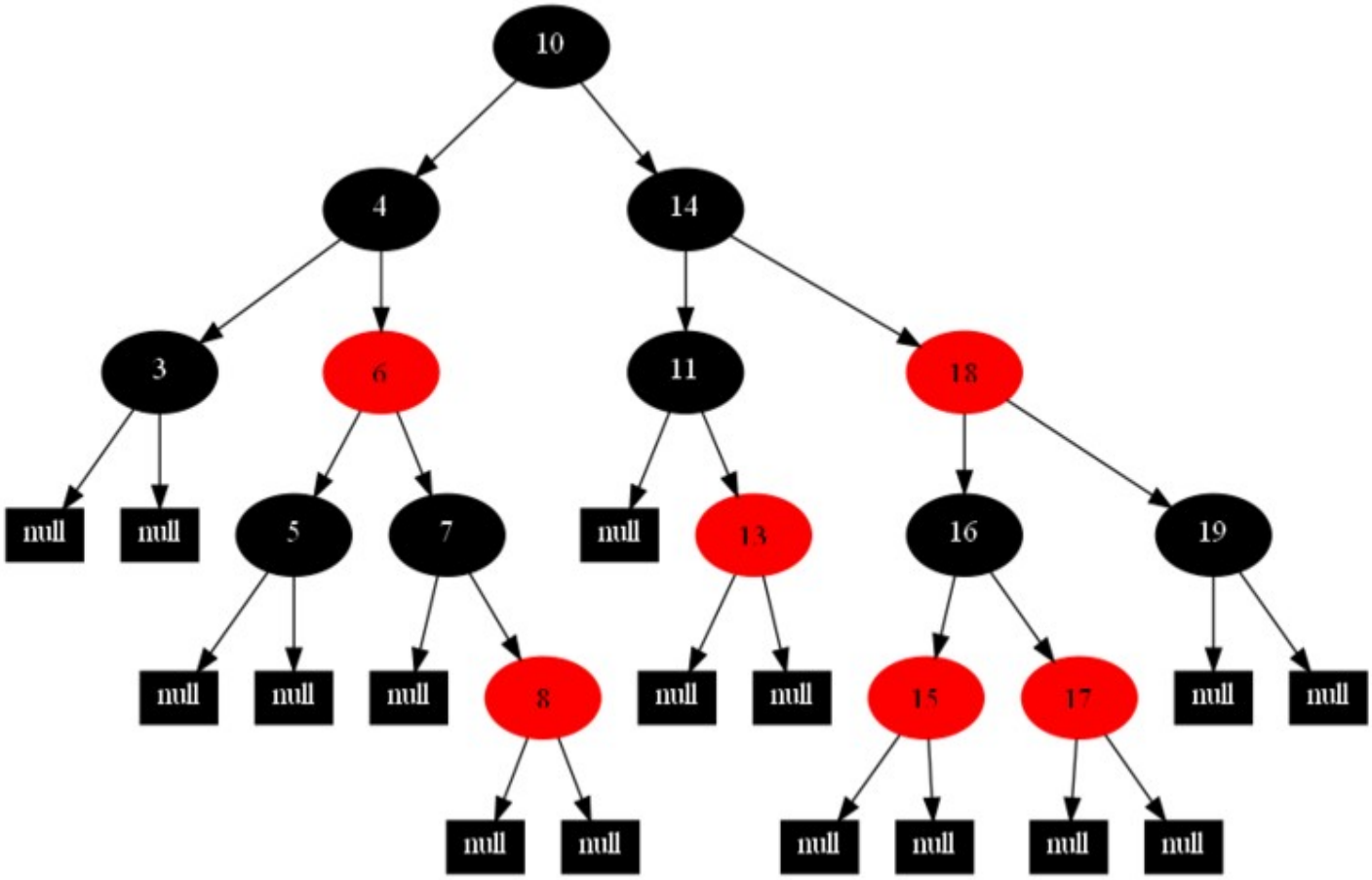
然后，以下的 20 张图，是一一删除这些结点 12 1 9 2 0 11 7 19 4 15 18 5 14 13 10 16 6 3 8 17 所得到的删除情况的全程演示图：

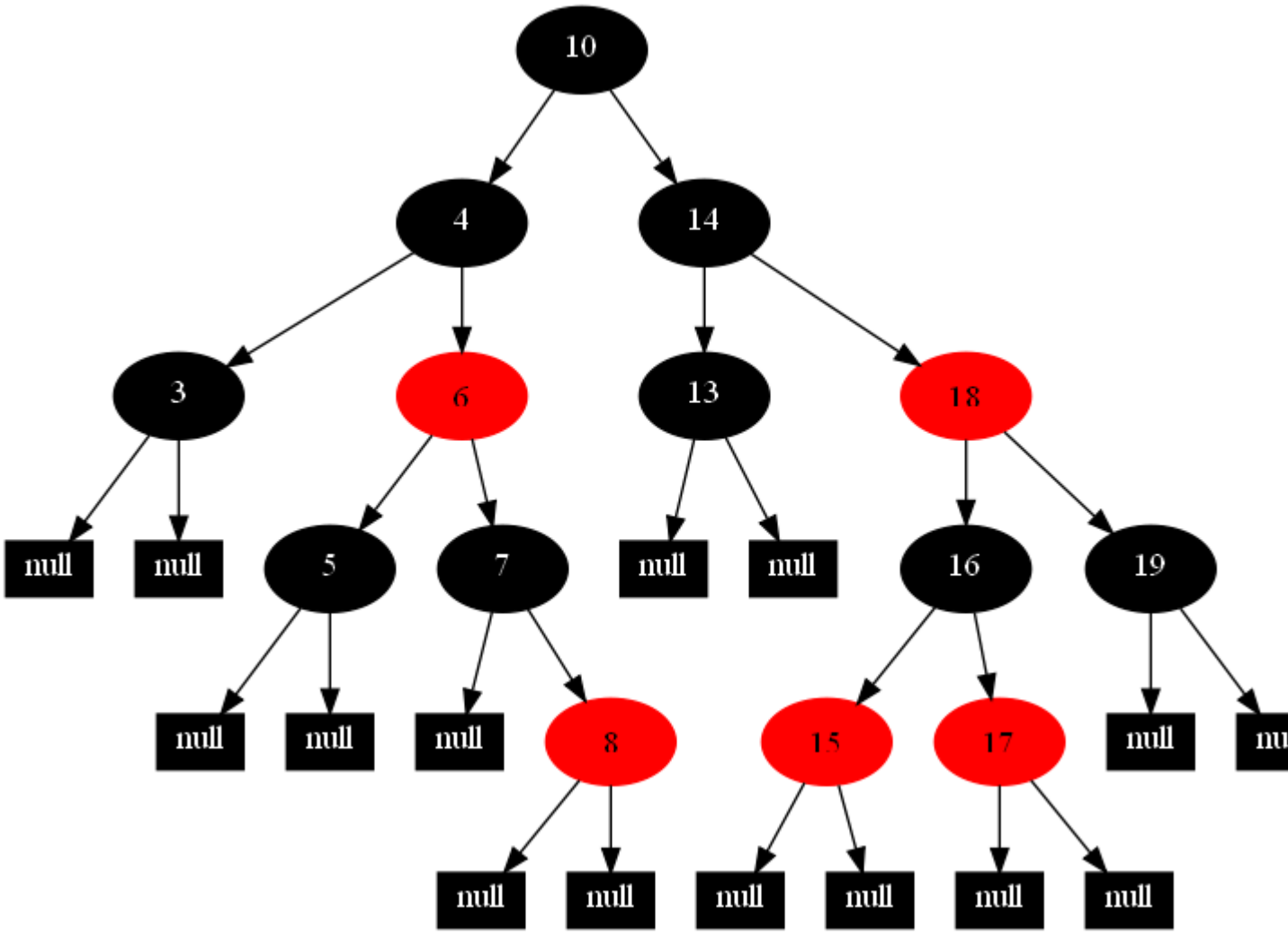

9
4
14
1
6
11
18
0
2
5
7
10
13
16
19
null
3
8
15
17

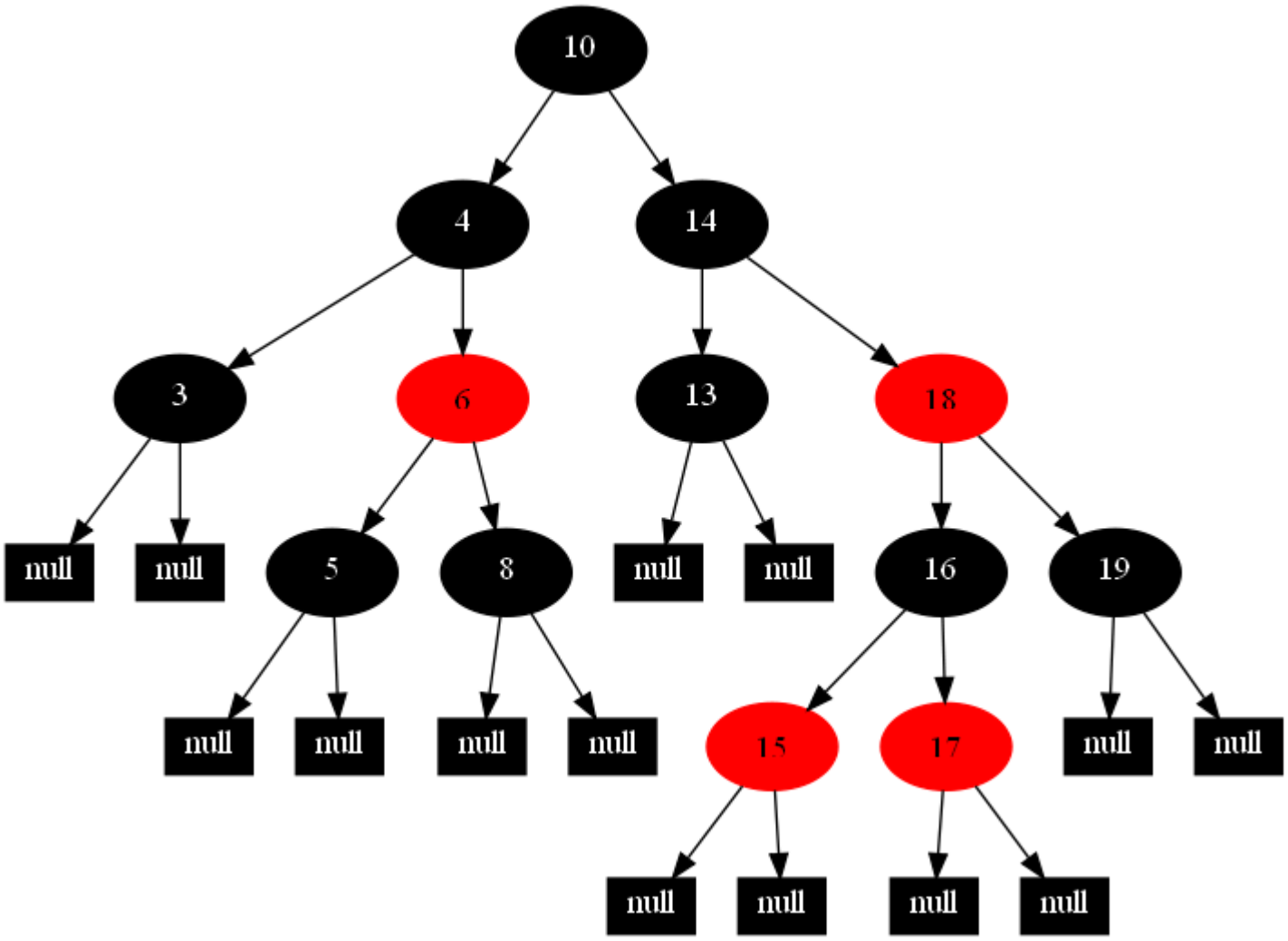

9
4
14
2
6
11
18
0
3
5
7
10
13
16
19
null
8
15
17

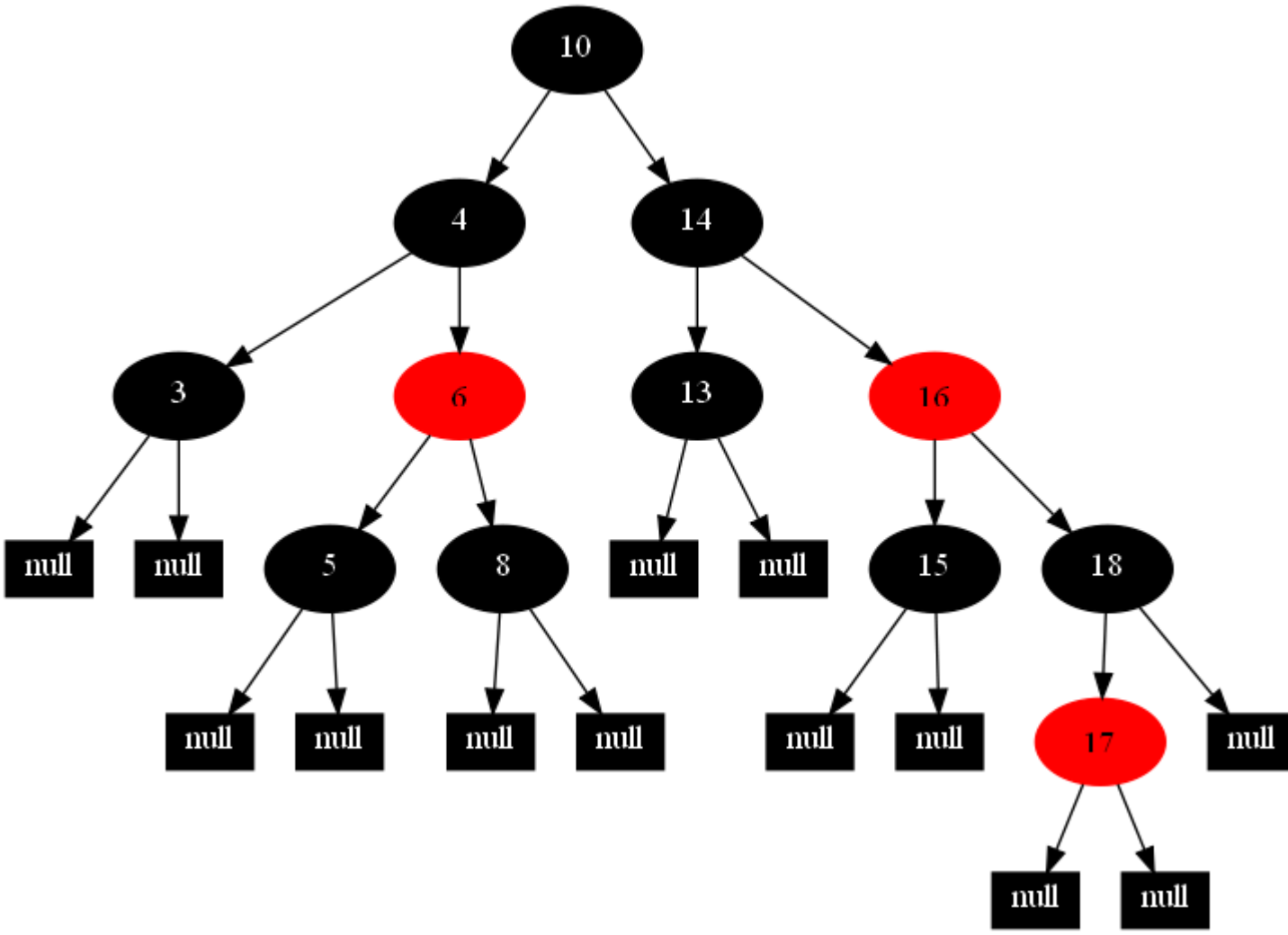

10
4
14
2
6
11
18
0
3
5
7
null
13
16
19
null
null
null
null
null
null
null
8
null
null
15
17
null
null
null
null
null
null
null
null

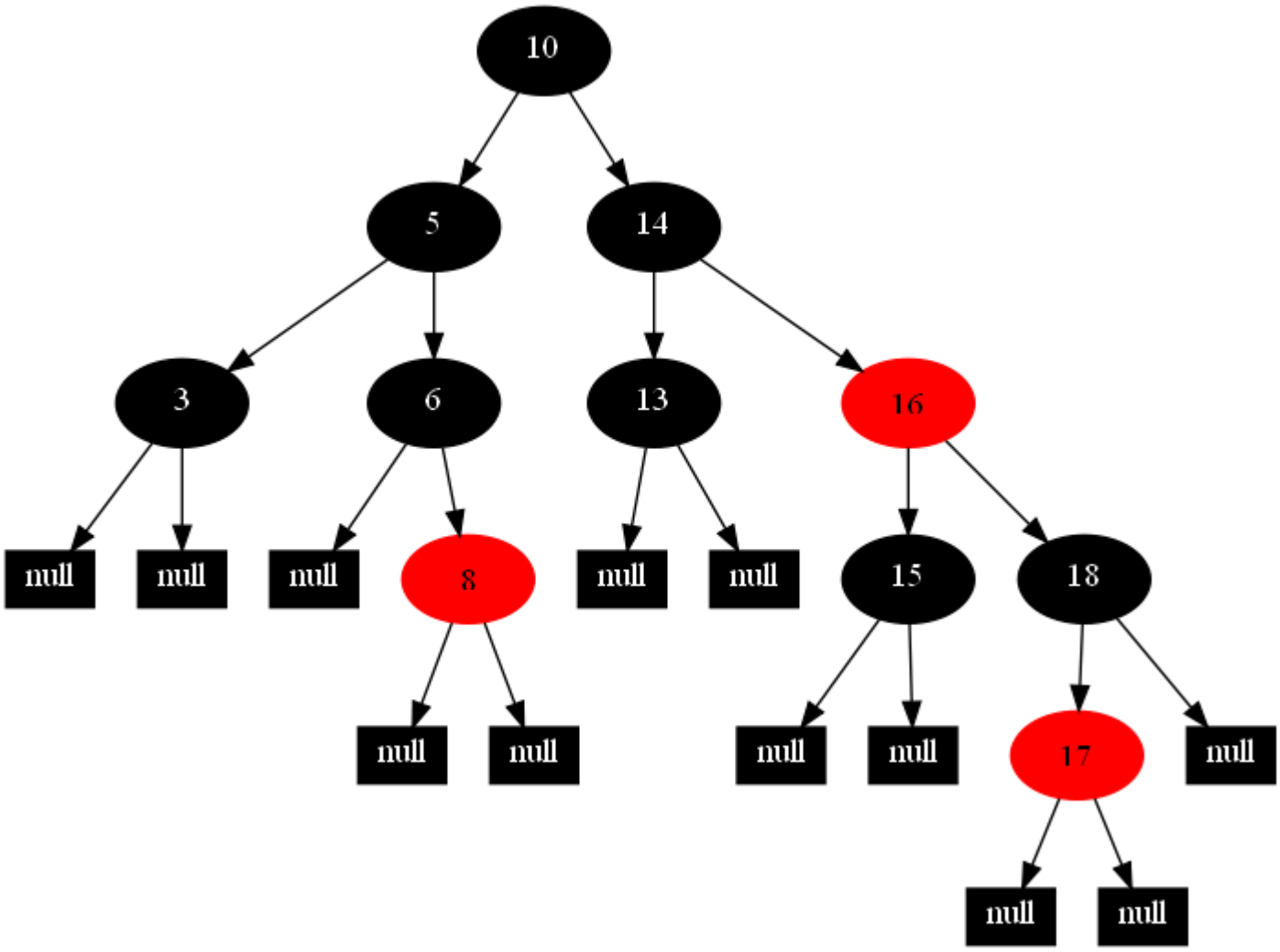

10
4
14
3
6
11
18
0
null
5
7
null
13
16
19
null
null
null
null
null
8
null
null
15
17
null
null
null
null
null
null
null
null

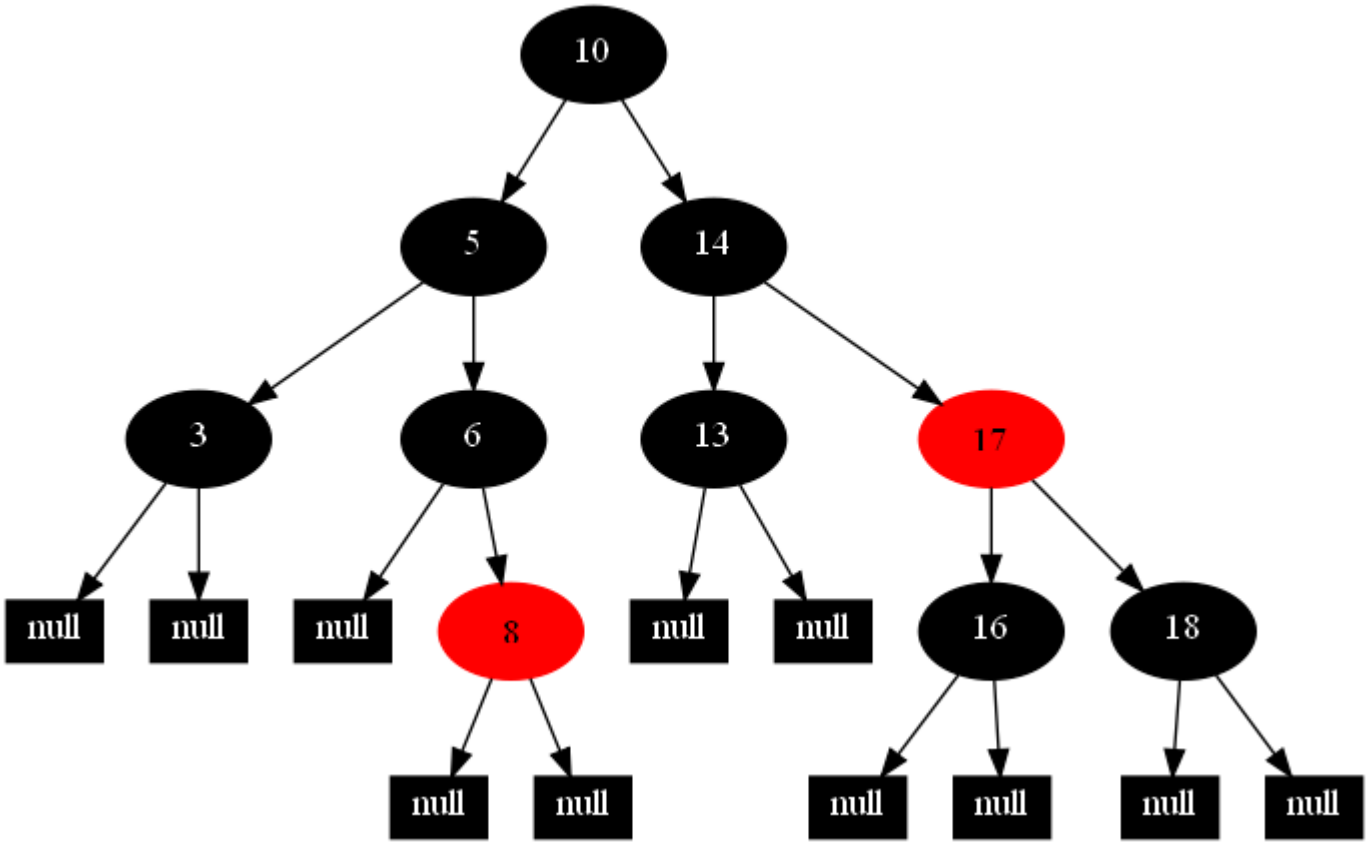

10
4
14
3
6
11
18
null
null
5
7
null
13
16
19
null
null
null
8
null
null
15
17
null
null
null
null
null
null
null
null

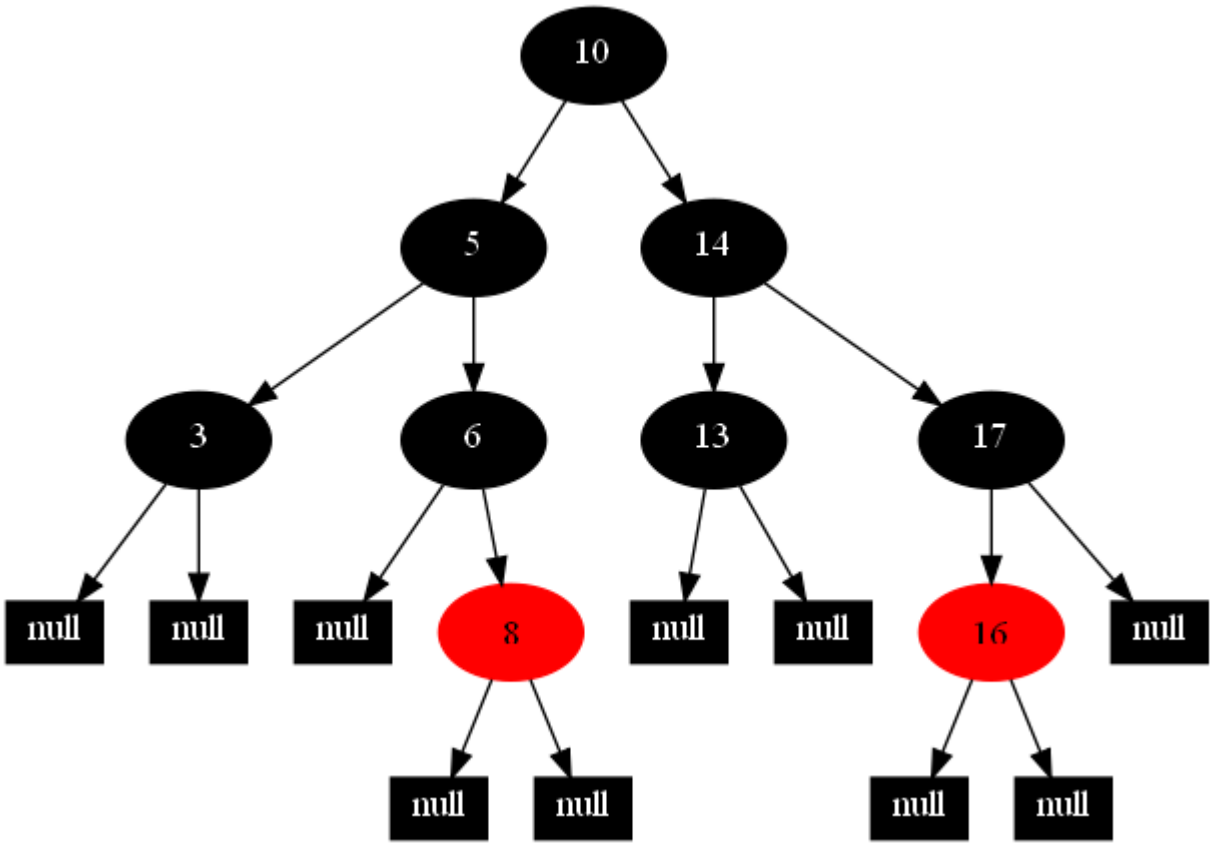

10
4
14
3
6
13
18
null
null
5
7
null
null
16
19
null
null
null
8
15
17
null
null
null
null
null
null
null
null

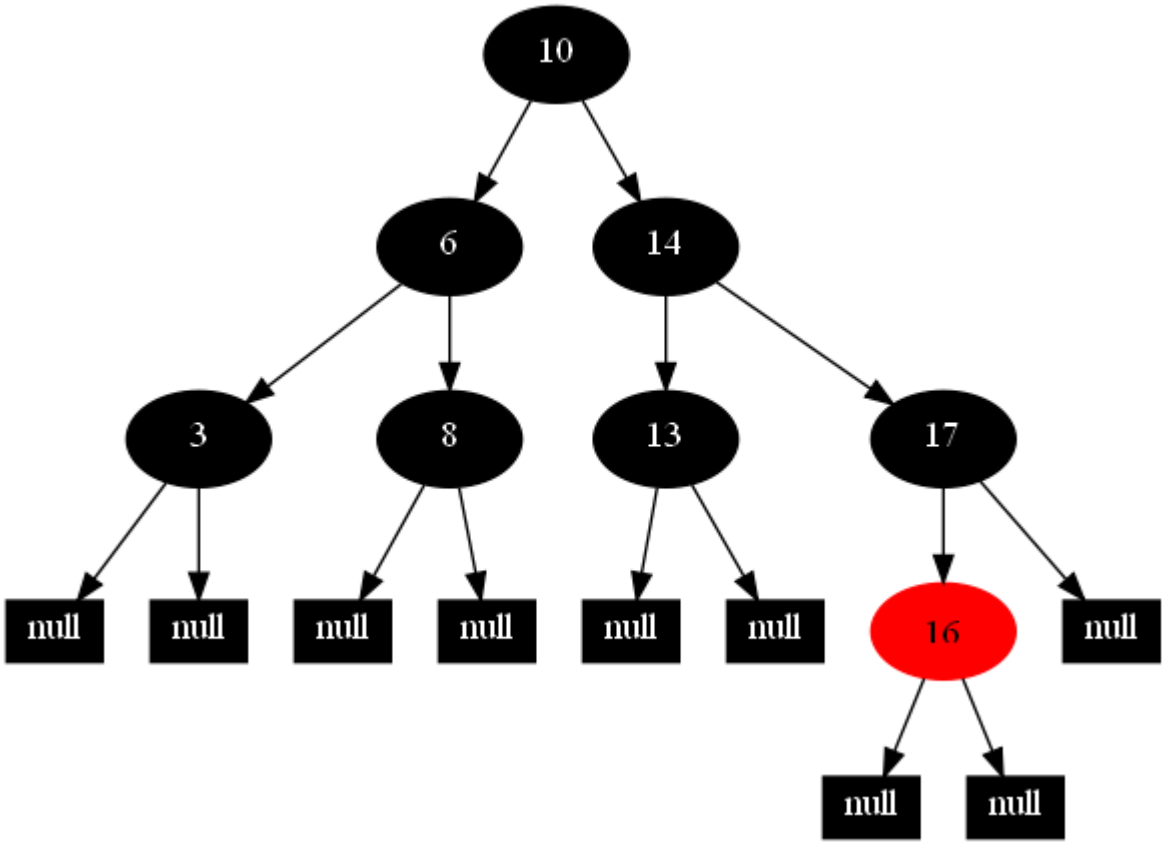

10
4
14
3
6
13
18
null
null
5
8
null
null
16
19
null
null
null
null
15
17
null
null
null
null
null
null

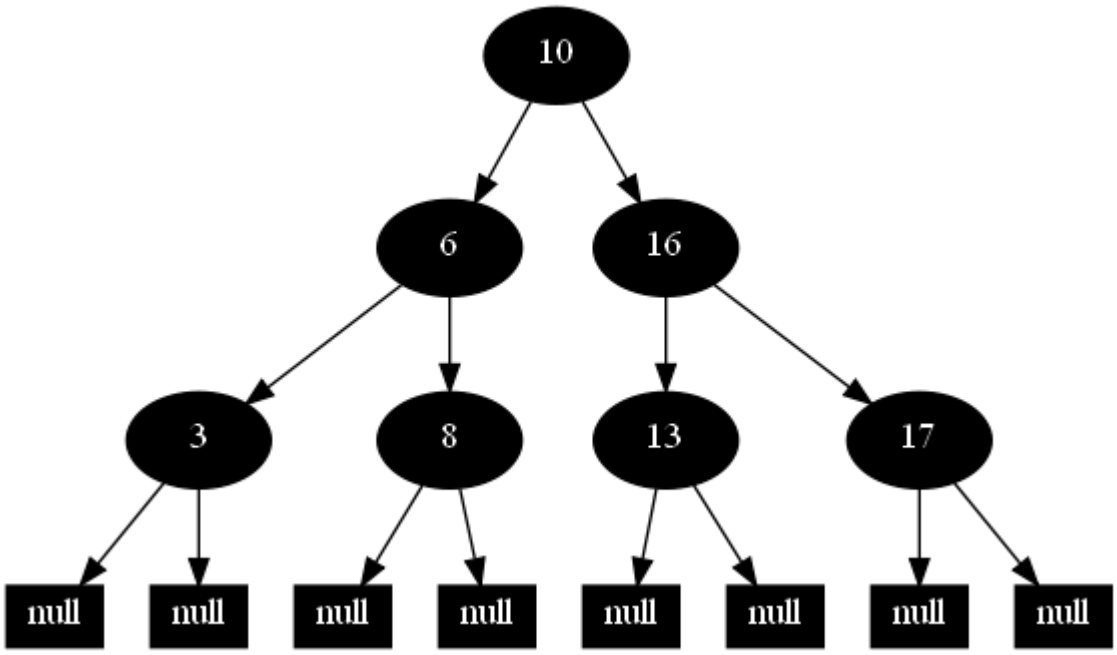

10
4
14
3
6
13
16
null
null
5
8
null
null
15
18
null
null
null
null
null
null
17
null
null

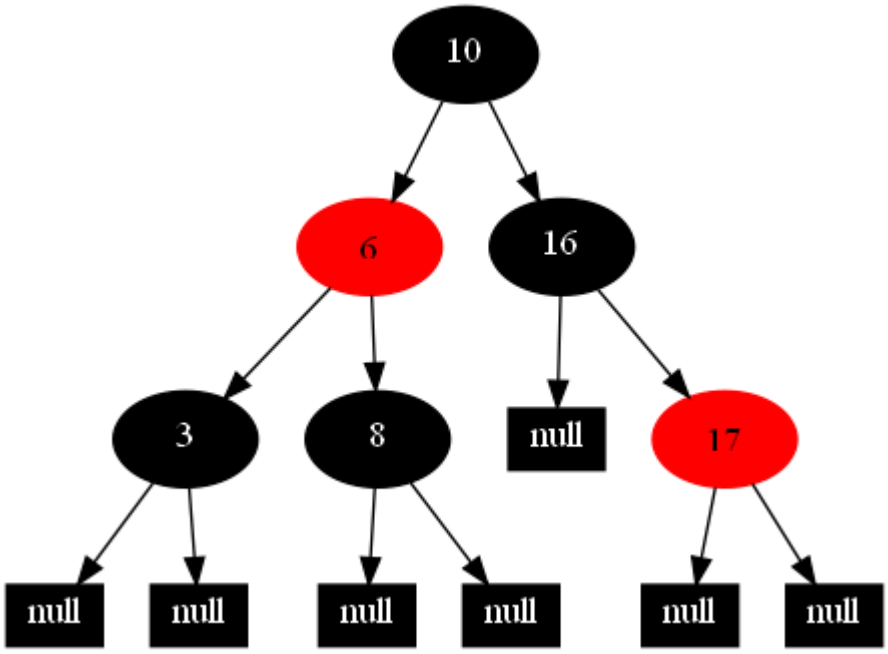

10
5
14
3
6
13
16
null
null
null
8
null
null
15
18
null
null
null
null
17
null
null
null

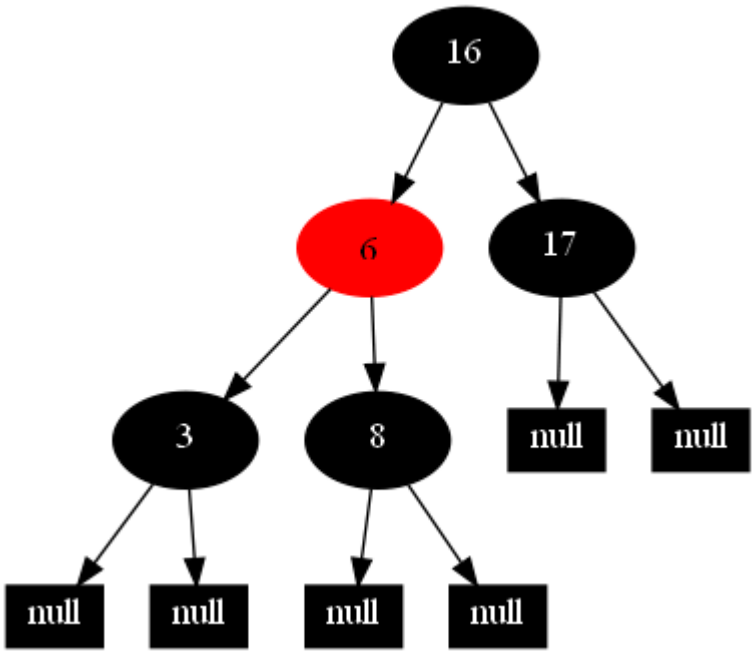

10
5
14
3
6
13
17
null
null
null
8
null
null
16
18
null
null
null
null
null
null
null
null

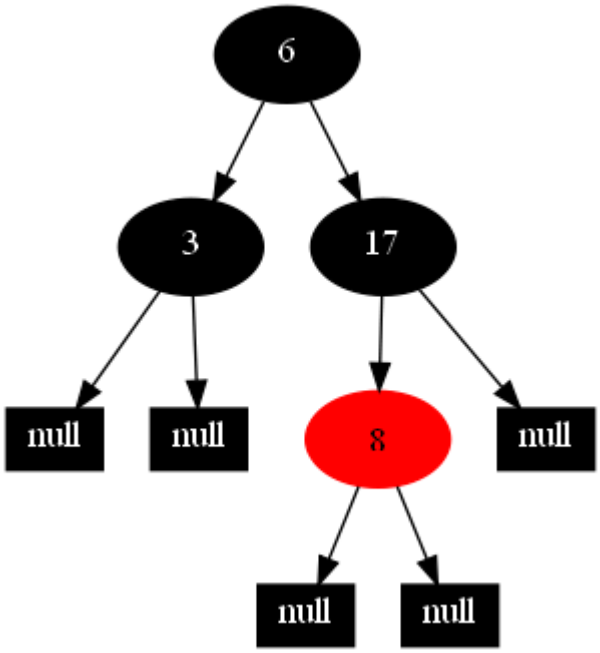

10
5
14
3
6
13
17
null
null
null
8
null
null
16
null
null
null
null
null
null

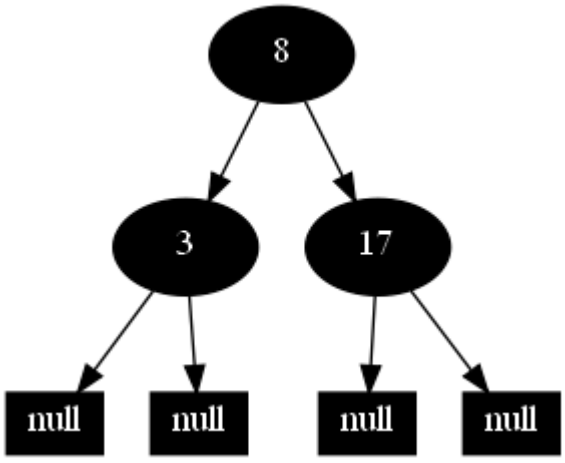

10
6
14
3
8
13
17
null
null
null
null
null
null
16
null
null
null

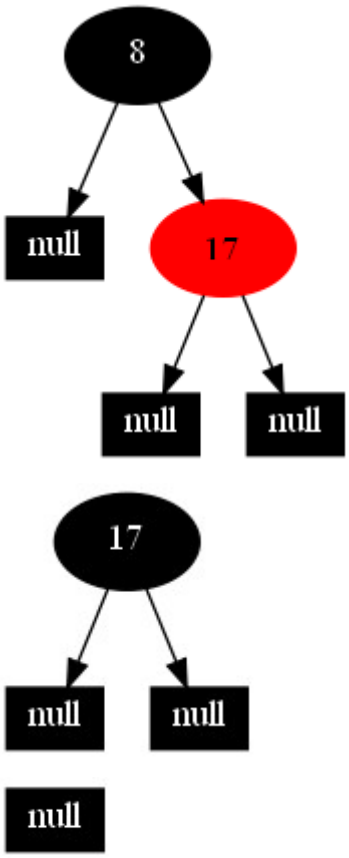

10
6
16
3
8
13
17
null
null
null
null
null
null
null
null


10
6
16
3
8
null
17
null
null
null
null
null
null


16
6
17
3
8
null
null
null
null
null
null


6
3
17
null
null
8
null
null
null


8
3
17
null
null
null
null



红黑树的一一删除各结点：12 1 9 2 0 11 7 19 4 15 18 5 14 13 10 16 6 3 8 17 的全程演示图完。

一步一图一代码，一定要让你真正彻底明白红黑树

作者：July 二零一一年一月九日

本文参考：

I、 The Art of Computer Programming Volume I
II、 Introduction to Algorithms, Second Edition
III、The Annotated STL Sources
IV、 Wikipedia
V、 Algorithms In C Third Edition

VI、 本人写的关于红黑树的前三篇文章：

第一篇：教你透彻了解红黑树：
http://blog.csdn.net/v_JULY_v/archive/2010/12/29/6105630.aspx
第二篇：红黑树算法的层层剖析与逐步实现
http://blog.csdn.net/v_JULY_v/archive/2010/12/31/6109153.aspx
第三篇：教你彻底实现红黑树：红黑树的 c 源码实现与剖析
http://blog.csdn.net/v_JULY_v/archive/2011/01/03/6114226.aspx

---

前言：

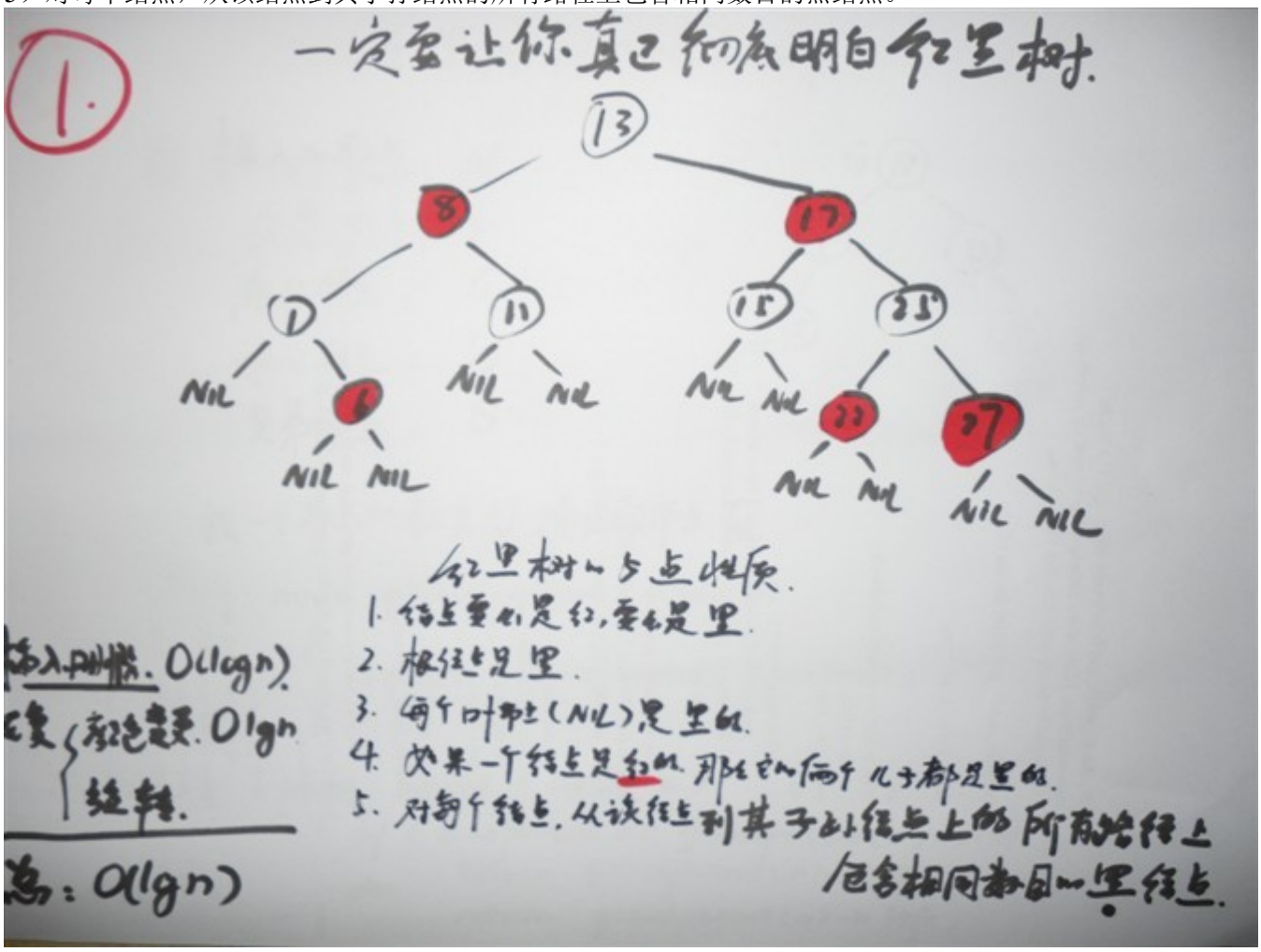
1、有读者反应，说看了我的前几篇文章，对红黑树的了解还是不够透彻。
2、我个人觉得，如果我一步一步，用图+代码来阐述各种插入、删除情况，可能会更直观易懂。
3、既然写了红黑树，那么我就一定要把它真正写好，让读者真正彻底明白红黑树。

本文相对我前面红黑树相关的 3 篇文章，主要有以下几点改进：

1.图、文字叙述、代码编写，彼此对应，明朗而清晰。

2.宏观总结，红黑树的性质与插入、删除情况的认识。
3.代码来的更直接，结合图，给你最直观的感受，彻底明白红黑树。

ok，首先，以下几点，你现在应该是要清楚明白了的：
I、红黑树的五个性质：
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。



II、红黑树插入的几种情况：
情况 1，z 的叔叔 y 是红色的。
情况 2：z 的叔叔 y 是黑色的，且 z 是右孩子
情况 3：z 的叔叔 y 是黑色的，且 z 是左孩子

III、红黑树删除的几种情况。
情况 1：x 的兄弟 w 是红色的。
情况 2：x 的兄弟 w 是黑色的，且 w 的俩个孩子都是黑色的。
情况 3：x 的兄弟 w 是黑色的，且 w 的左孩子是红色，w 的右孩子是黑色。
情况 4：x 的兄弟 w 是黑色的，且 w 的右孩子是红色的。

除此之外，还得明确一点：
IV、我们知道，红黑树插入、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。

V、并区别以下俩种操作：
1)红黑树插入、删除结点的操作，RB-INSERT(T, z)，RB-DELETE(T, z)
2).红黑树已经插入、删除结点之后，
为了保持红黑树原有的红黑性质而做的恢复与保持红黑性质的操作。
如 RB-INSERT-FIXUP(T, z)，RB-DELETE-FIXUP(T, x)

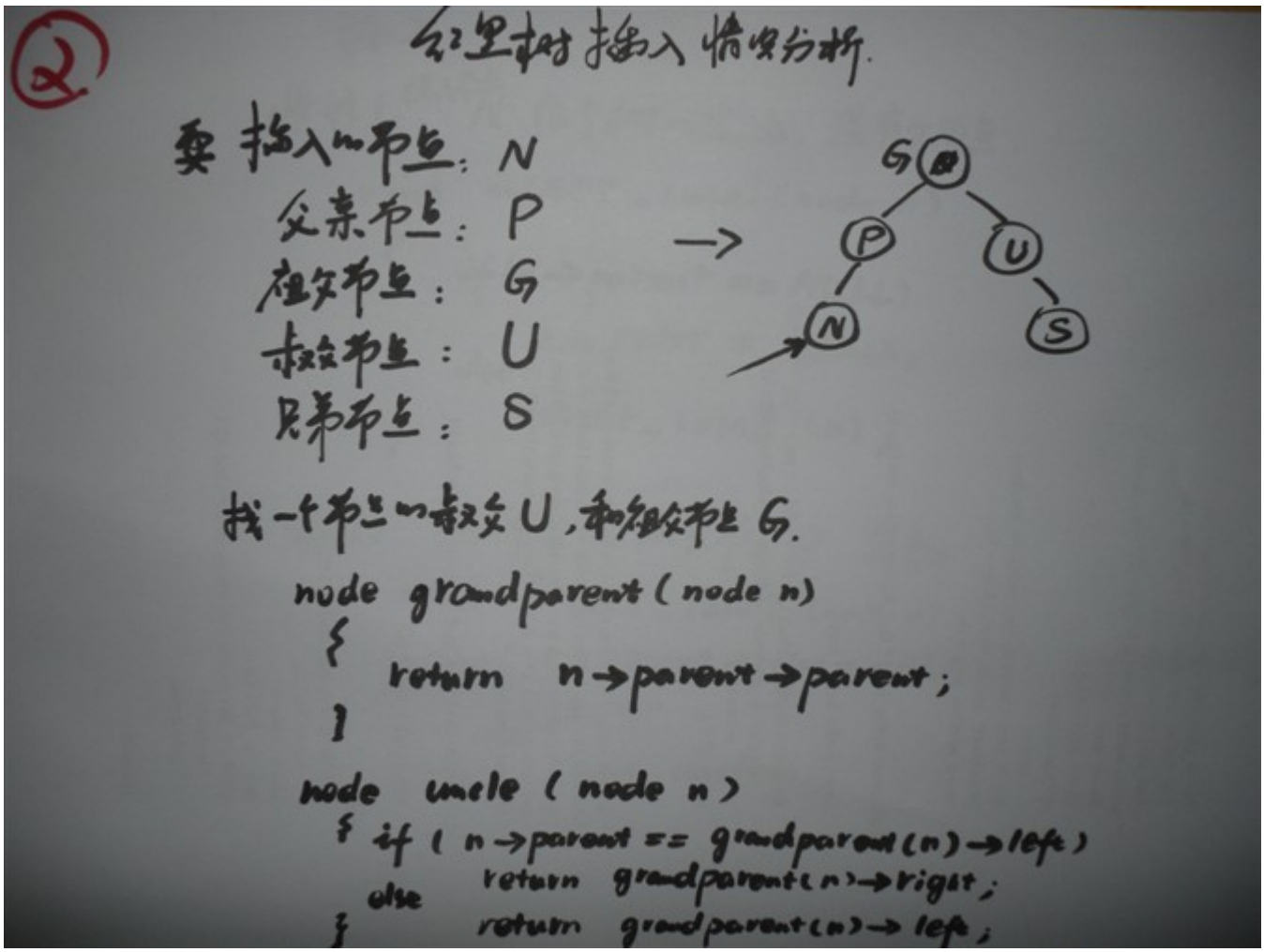
以上这 5 点，我已经在我前面的 2 篇文章，都已阐述过不少次了，希望，你现在已经透彻明了。

本文，着重图解分析红黑树插入、删除结点后为了维持红黑性质而做修复工作的各种情况。
[下文各种插入、删除的情况，与我的第二篇文章，红黑树算法的实现与剖析相对应]

ok，开始。
一、在下面的分析中，我们约定：
要插入的节点为，N
父亲节点，P
祖父节点，G
叔叔节点，U
兄弟节点，S

如下图所示，找一个节点的祖父和叔叔节点:

```
node grandparent(node n)     //祖父
{
    return n->parent->parent;
}

node uncle(node n)              //叔叔
{
    if (n->parent == grandparent(n)->left)
        return grandparent(n)->right;
    else
        return grandparent(n)->left;
}
```



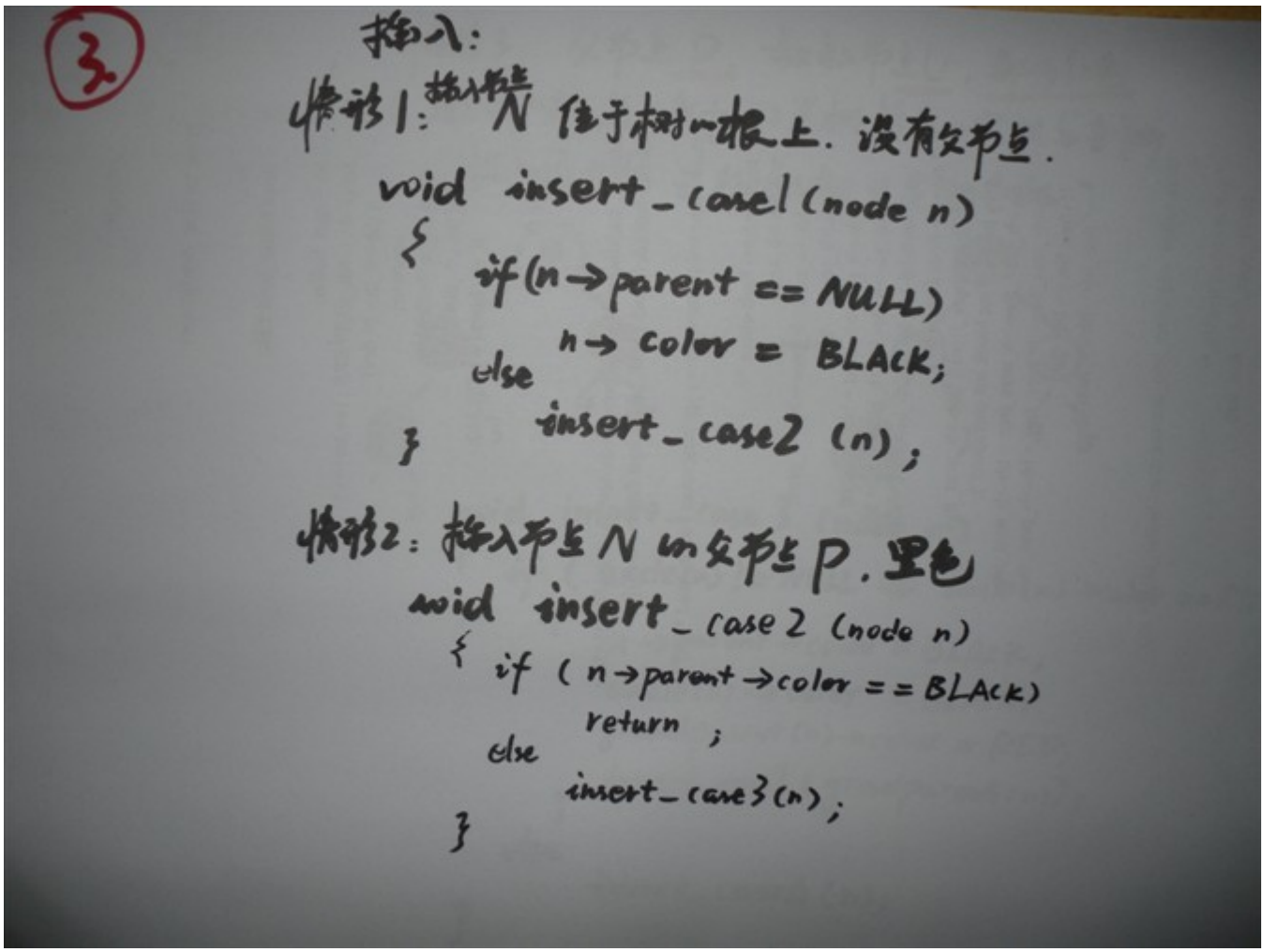
二、红黑树插入的几种情况

情形 1: 新节点 N 位于树的根上，没有父节点

```
void insert_case1(node n) {
    if (n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2(n);
}
```

情形 2: 新节点的父节点 P 是黑色

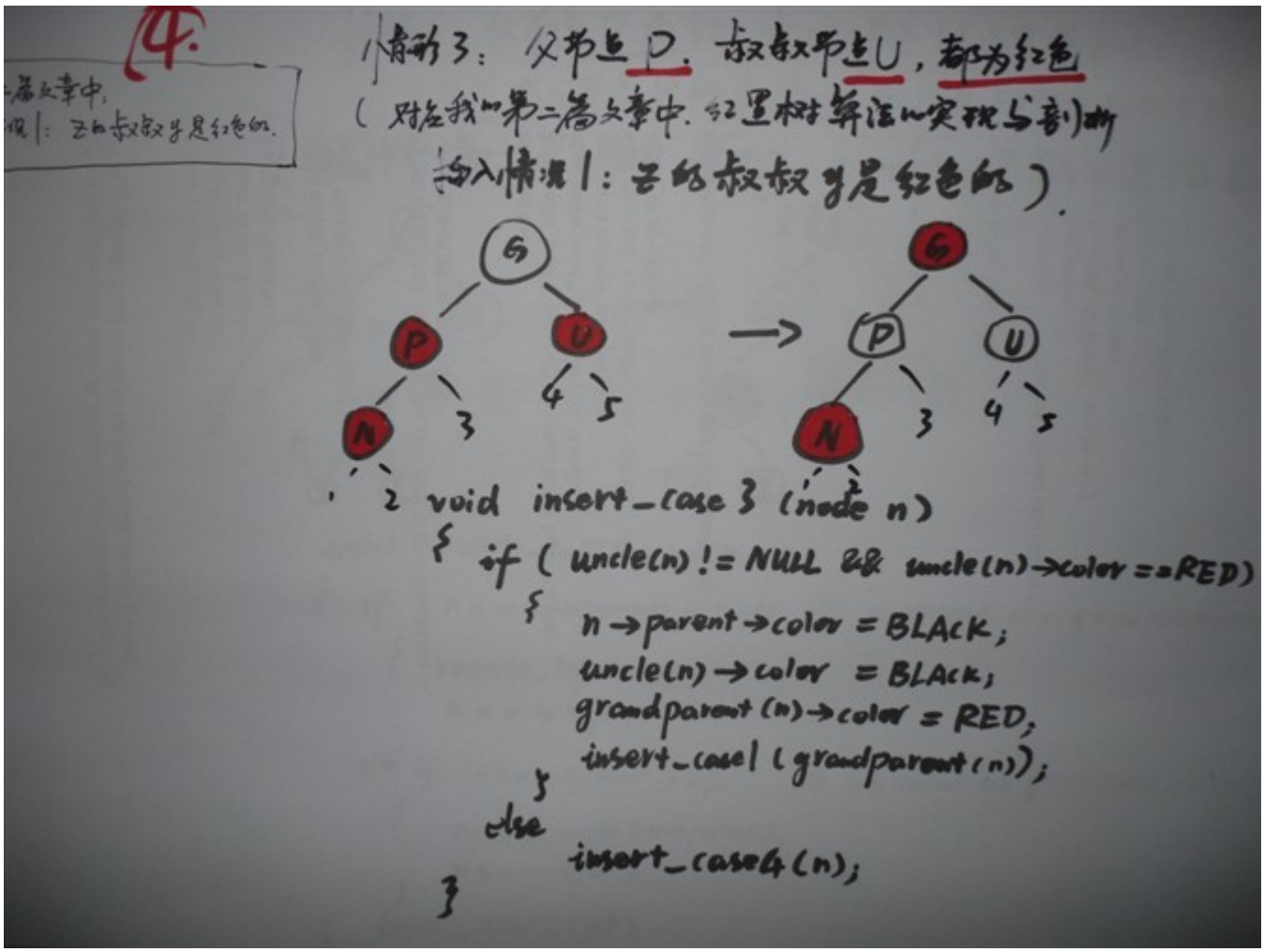
```
void insert_case2(node n) {
    if (n->parent->color == BLACK)
        return; /* 树仍旧有效 */
    else
        insert_case3(n);
}
```



情形 3:父节点 P、叔叔节点 U，都为红色，

[对应第二篇文章中，的情况 1：z 的叔叔是红色的。]

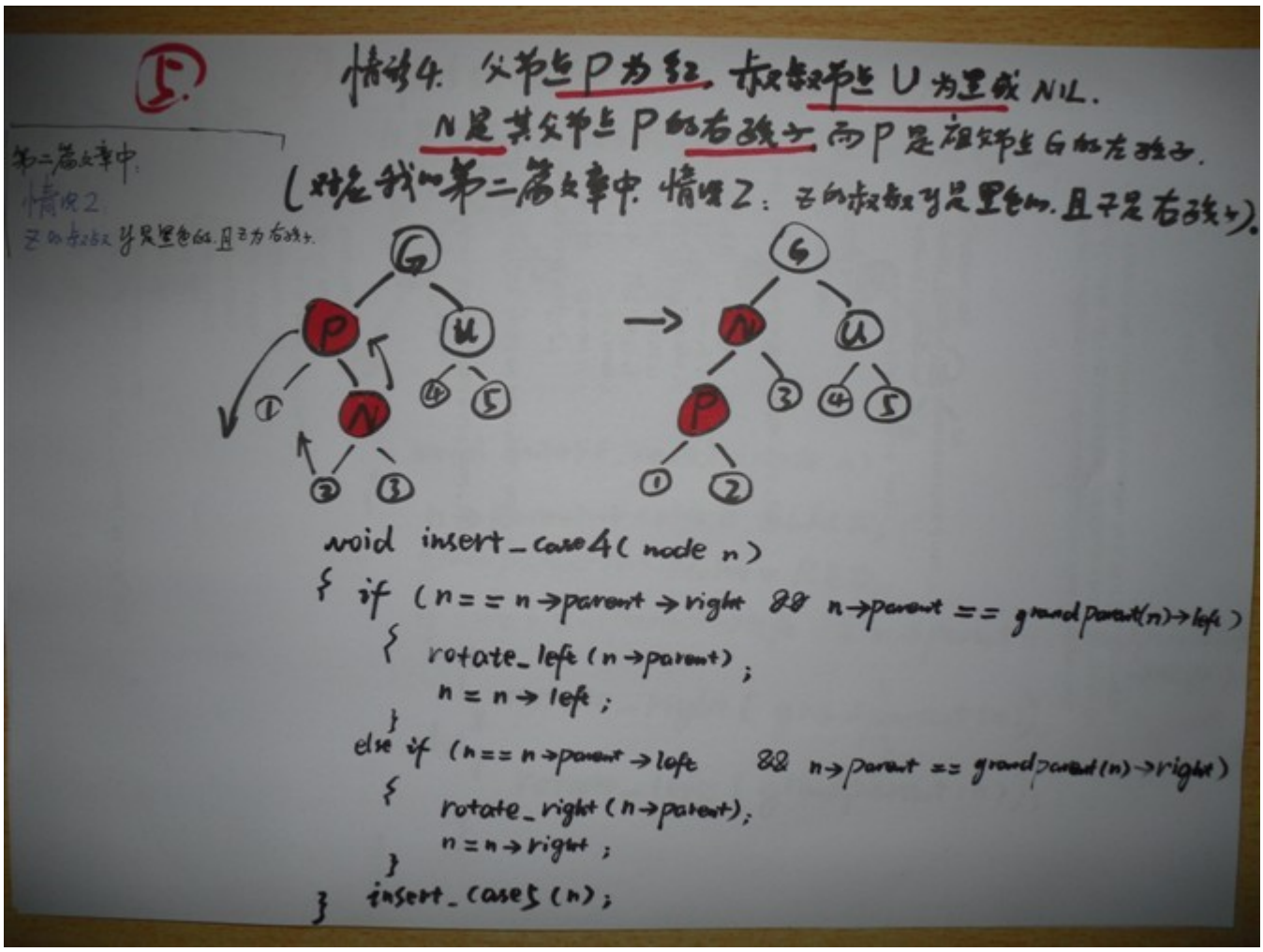
```
void insert_case3(node n) {
    if (uncle(n) != NULL && uncle(n)->color == RED) {
        n->parent->color = BLACK;
        uncle(n)->color = BLACK;
        grandparent(n)->color = RED;
        insert_case1(grandparent(n));   //因为祖父节点可能是红色的，违反性质 4，递归情形 1.
    }
    else
        insert_case4(n);   //否则，叔叔是黑色的，转到下述情形 4 处理。
```



此时新插入节点 N 做为 P 的左子节点或右子节点都属于上述**情形 3**,上图仅显示 N 做为 P 左子的情形。

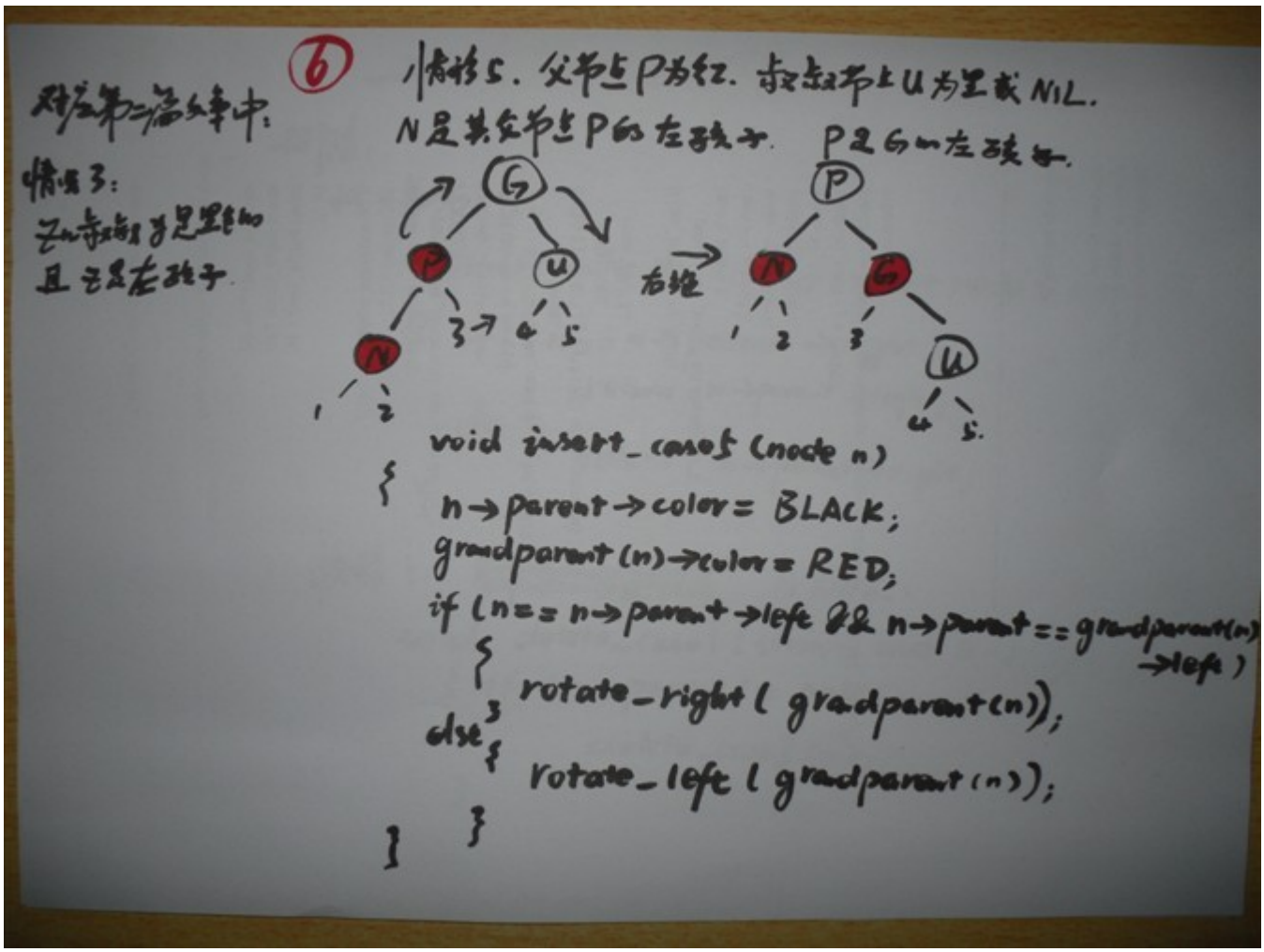
情形 4: 父节点 P 是红色，叔叔节点 U 是黑色或 NIL;
插入节点 **N 是其父节点 P 的右孩子**，而父节点 P 又是其父节点的左孩子。
[对应我第二篇文章中，的情况 2：z 的叔叔是黑色的，且 z 是右孩子]

```
void insert_case4(node n) {
    if (n == n->parent->right && n->parent == grandparent(n)->left) {
        rotate_left(n->parent);
        n = n->left;
    } else if (n == n->parent->left && n->parent == grandparent(n)->right) {
        rotate_right(n->parent);
        n = n->right;
    }
    insert_case5(n);    //转到下述情形 5 处理。
```



情形 5: 父节点 P 是红色，而叔父节点 U 是黑色或 NIL，
要插入的节点 **N 是其父节点的左孩子**，而父节点 P 又是其父 G 的左孩子。
[对应我第二篇文章中，情况 3：z 的叔叔是黑色的，且 z 是左孩子。]

```
void insert_case5(node n) {
    n->parent->color = BLACK;
    grandparent(n)->color = RED;
    if (n == n->parent->left && n->parent == grandparent(n)->left) {
        rotate_right(grandparent(n));
    } else {
        /* 反情况，N 是其父节点的右孩子，而父节点 P 又是其父 G 的右孩子 */
        rotate_left(grandparent(n));
    }
}
```



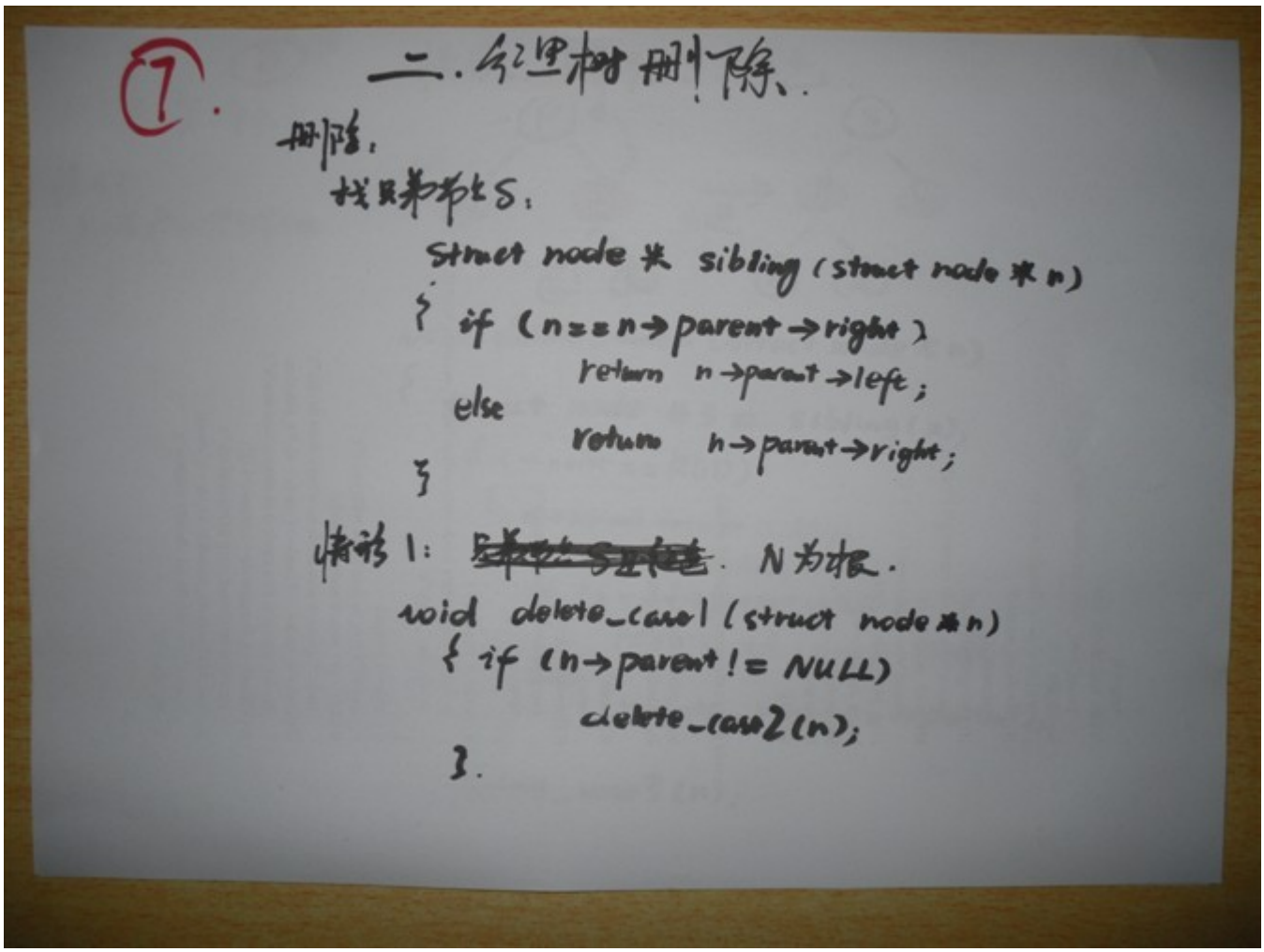
三、红黑树删除的几种情况

上文我们约定，兄弟节点设为S，我们使用下述函数找到兄弟节点:

```
struct node * sibling(struct node *n)  //找兄弟节点
{
        if (n == n->parent->left)
                return n->parent->right;
        else
                return n->parent->left;
}
```

情况 1: N 是新的根。

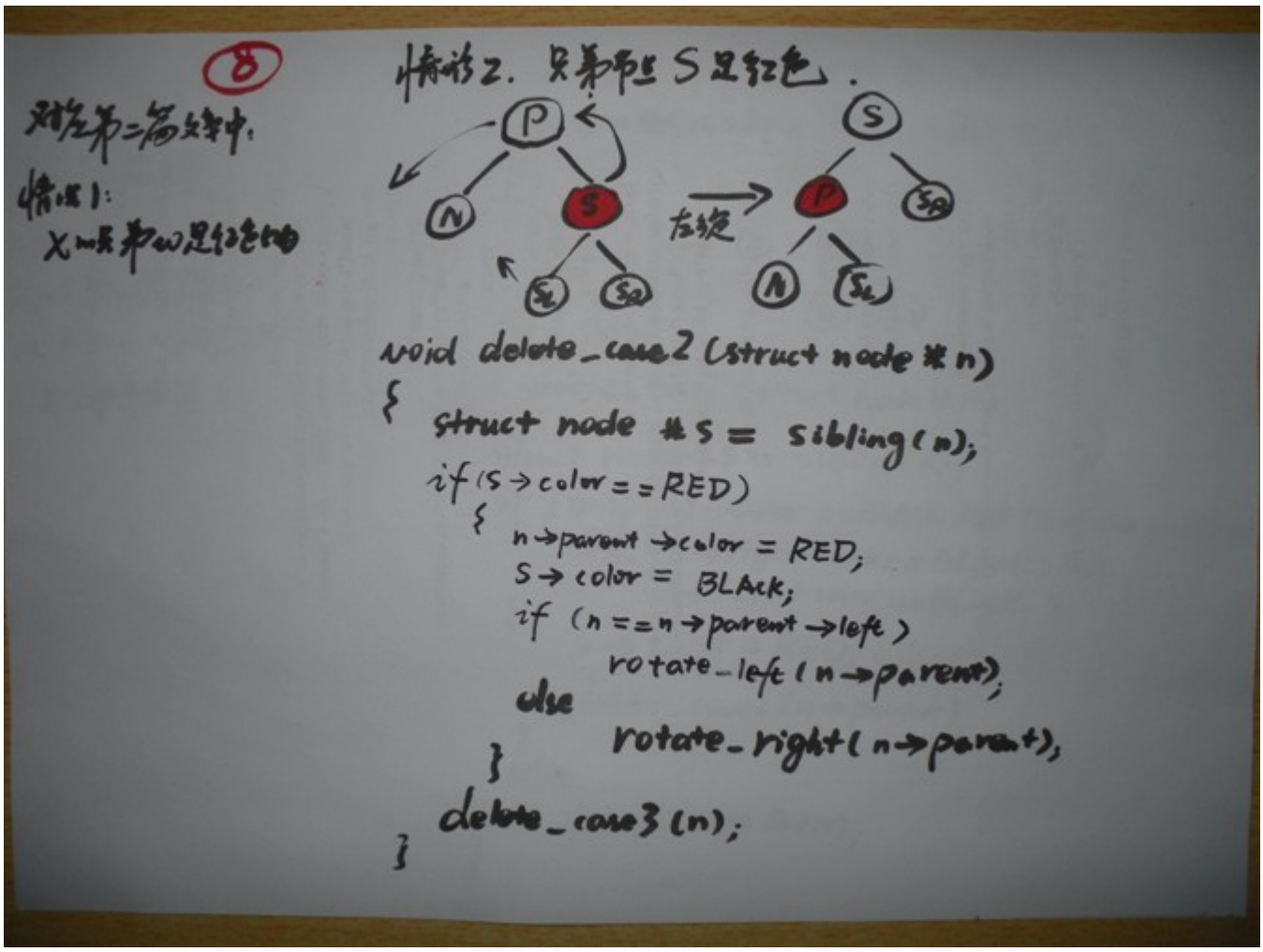
```
void
delete_case1(struct node *n)
{
        if (n->parent != NULL)
                delete_case2(n);
}
```



情形 2：兄弟节点 S 是红色

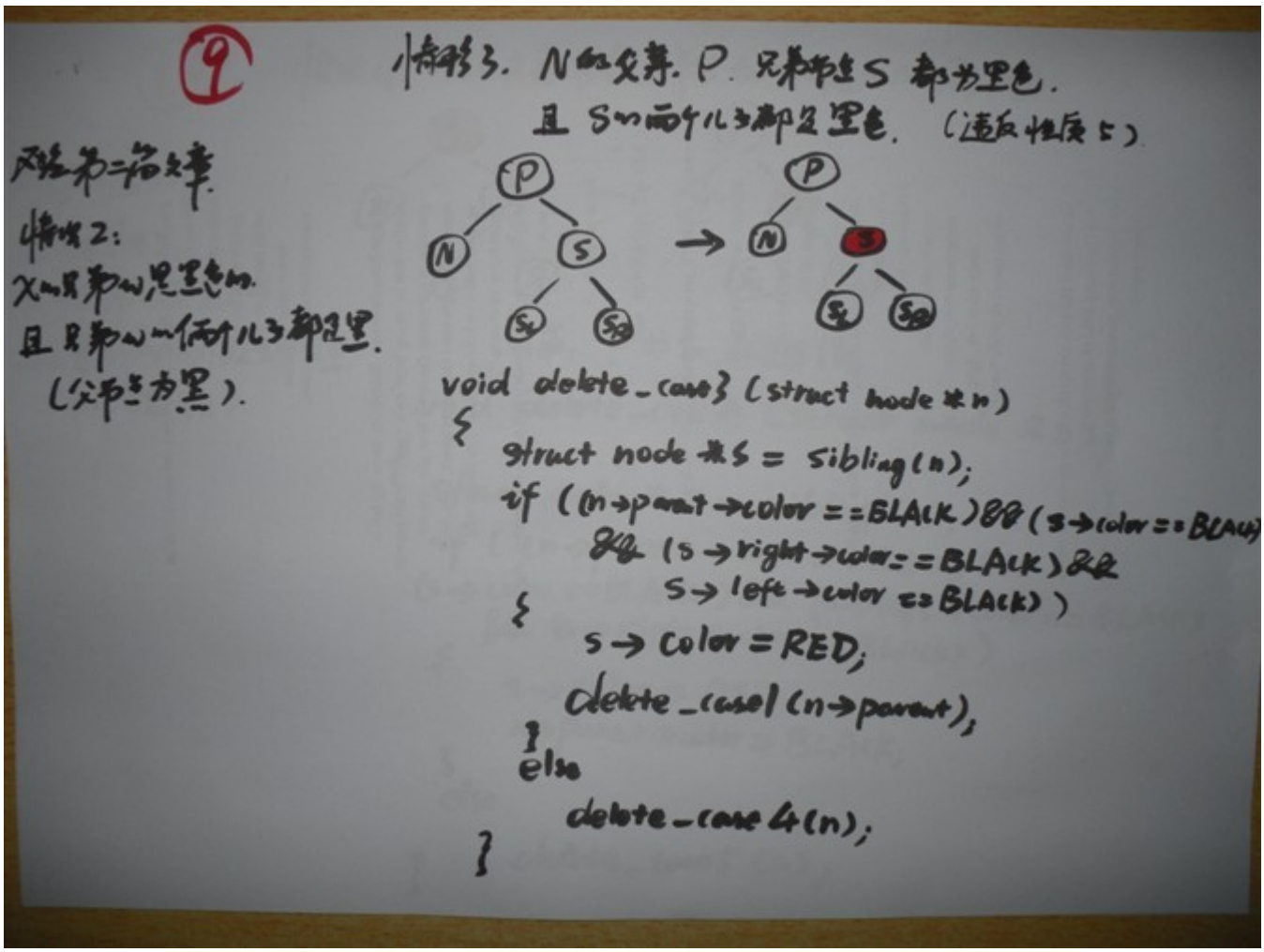
[对应我第二篇文章中，情况 1：x 的兄弟 w 是红色的。]

```
void delete_case2(struct node *n)
{
	struct node *s = sibling(n);

	if (s->color == RED) {
		n->parent->color = RED;
		s->color = BLACK;
		if (n == n->parent->left)
			rotate_left(n->parent);  //左旋
		else
			rotate_right(n->parent);
	}
	delete_case3(n);
}
```



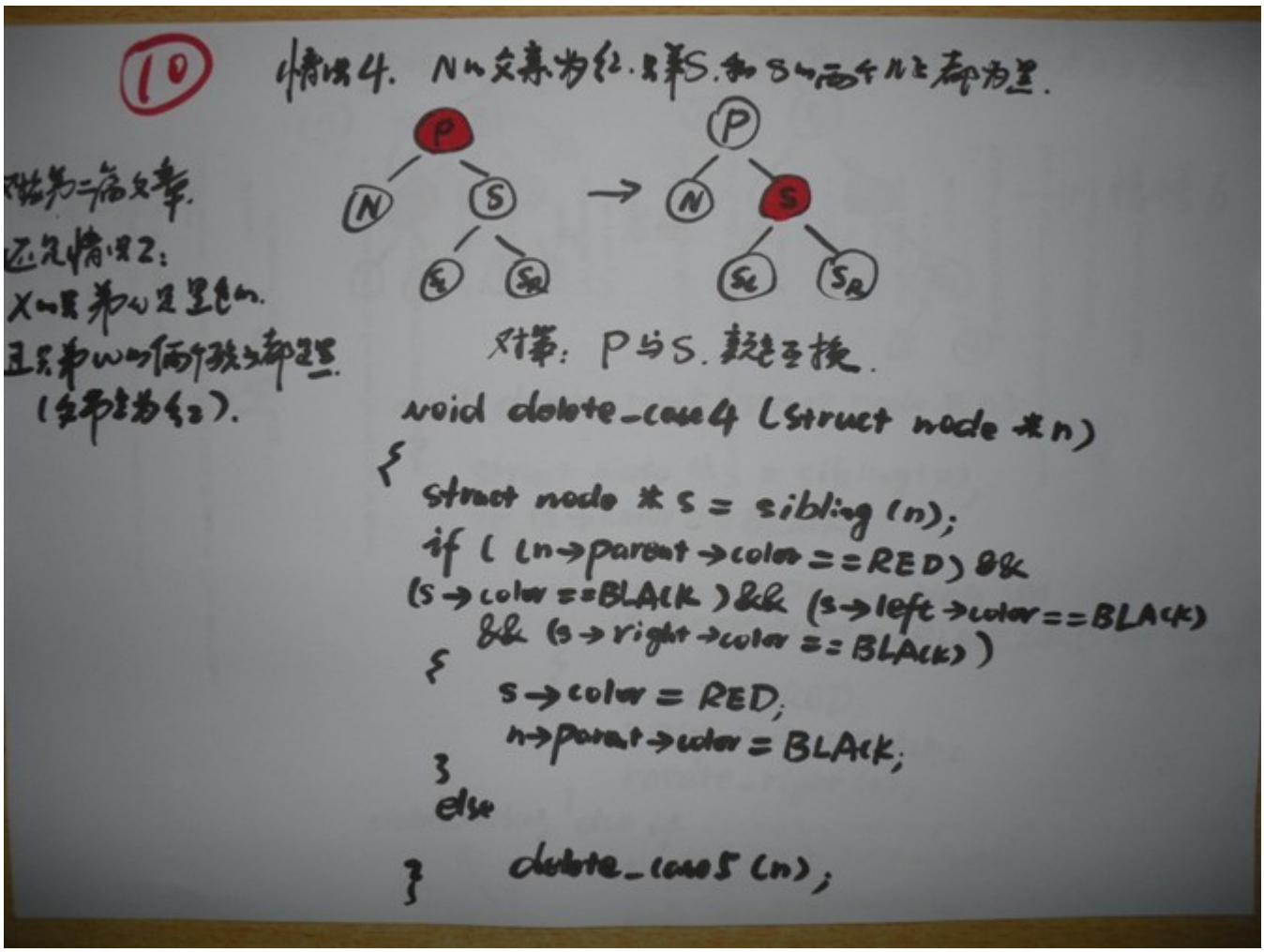
情况 3: 兄弟节点 S 是黑色的，且 S 的俩个儿子都是黑色的。但 N 的父节点 P，是黑色。
[对应我第二篇文章中，情况 2：x 的兄弟 w 是黑色的，且兄弟 w 的俩个儿子都是黑色的。
(这里，父节点 P 为黑)]

```
void delete_case3(struct node *n)
{
        struct node *s = sibling(n);

        if ((n->parent->color == BLACK) &&
           (s->color == BLACK) &&
           (s->left->color == BLACK) &&
           (s->right->color == BLACK)) {
              s->color = RED;
              delete_case1(n->parent);
        } else
              delete_case4(n);
}
```



情况 4: 兄弟节点 **S 是黑色的、S 的儿子也都是黑色的，但是 N 的父亲 P，是红色。**
[还是对应我第二篇文章中，情况 2：x 的兄弟 w 是黑色的，且 w 的俩个孩子都是黑色的。
(这里，父节点 P 为红)]

```
void delete_case4(struct node *n)
{
        struct node *s = sibling(n);

        if ((n->parent->color == RED) &&
           (s->color == BLACK) &&
           (s->left->color == BLACK) &&
           (s->right->color == BLACK)) {
              s->color = RED;
              n->parent->color = BLACK;
        } else
              delete_case5(n);
}
```



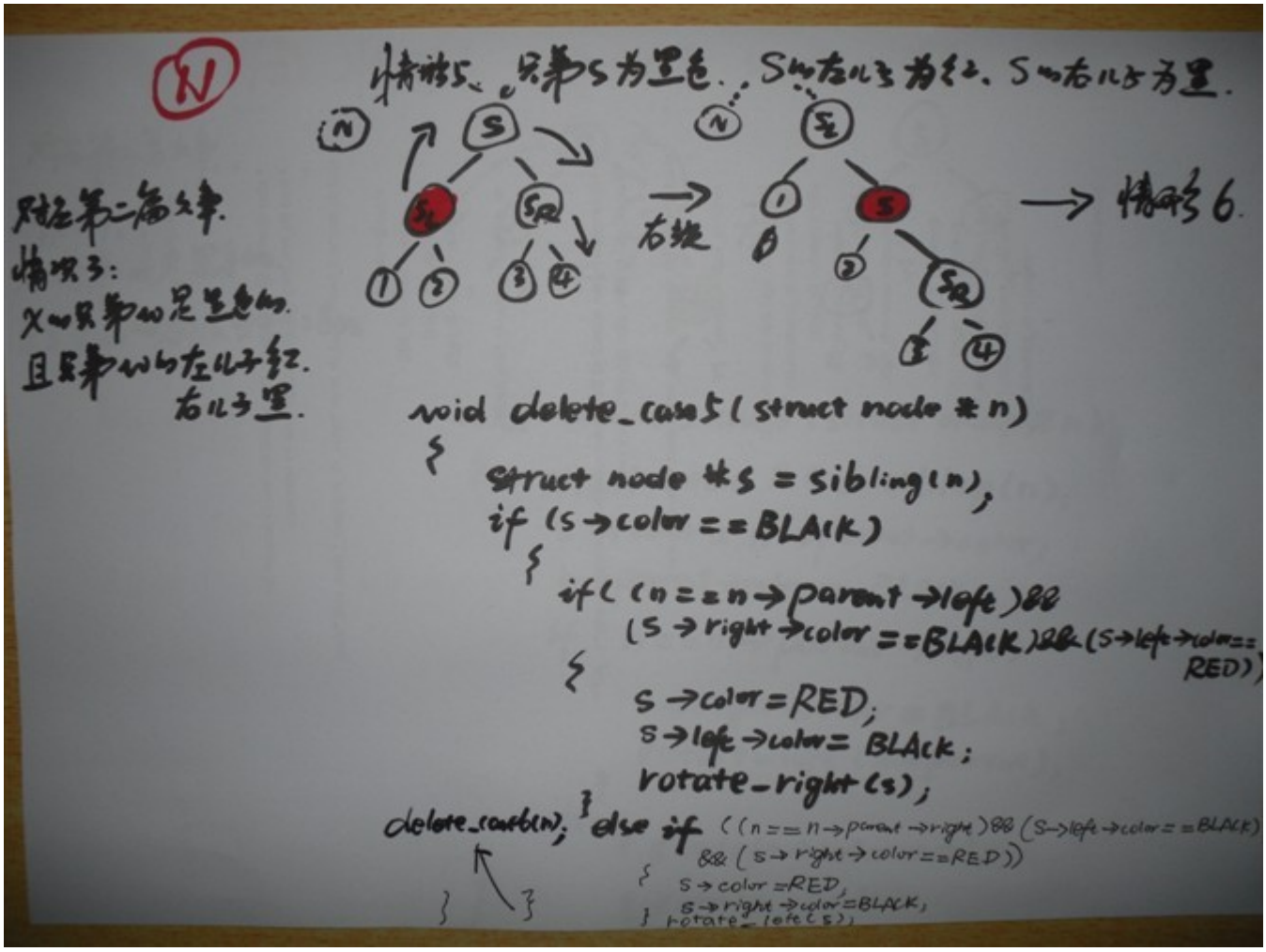
情况 5: 兄弟 S 为黑色，S 的左儿子是红色，S 的右儿子是黑色，而 N 是它父亲的左儿子。

//此种情况，最后转化到下面的情况 6。

[对应我第二篇文章中，情况 3：x 的兄弟 w 是黑色的，w 的左孩子是红色，w 的右孩子是黑色。]

```
void delete_case5(struct node *n)
{
      struct node *s = sibling(n);

      if  (s->color == BLACK)
            if ((n == n->parent->left) &&
               (s->right->color == BLACK) &&
               (s->left->color == RED)) {
                  // this last test is trivial too due to cases 2-4.
                  s->color = RED;
                  s->left->color = BLACK;
                  rotate_right(s);
            } else if ((n == n->parent->right) &&
                     (s->left->color == BLACK) &&
                     (s->right->color == RED)) {
                  // this last test is trivial too due to cases 2-4.
                  s->color = RED;
                  s->right->color = BLACK;
                  rotate_left(s);
            }
```

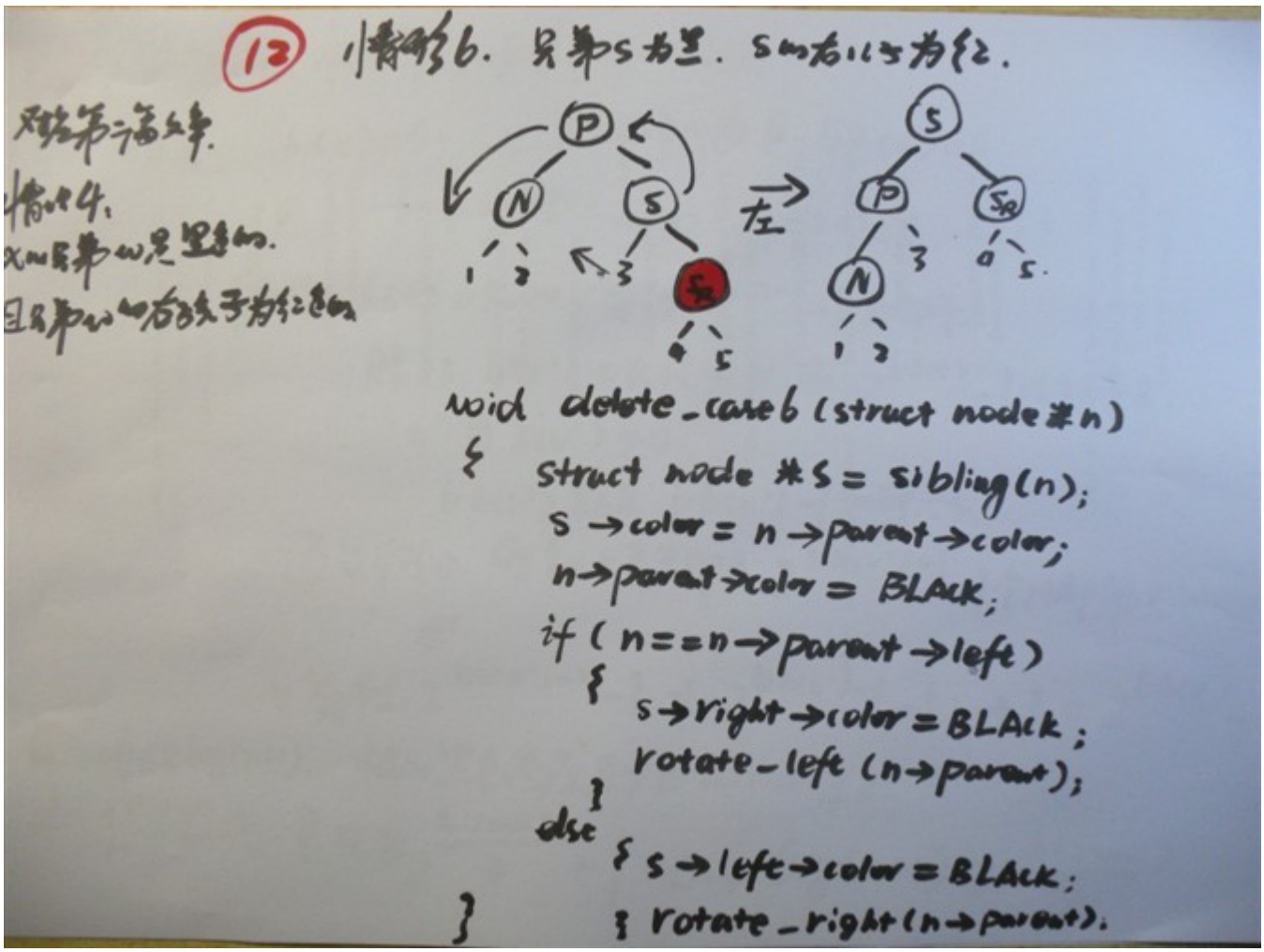
```
    }
    delete_case6(n);  //转到情况 6。
```



情况 6: 兄弟节点 S 是黑色，S 的右儿子是红色，而 N 是它父亲的左儿子。

[对应我第二篇文章中，情况 4:x 的兄弟 w 是黑色的，且 w 的右孩子时红色的。]

```
void delete_case6(struct node *n)
{
    struct node *s = sibling(n);

    s->color = n->parent->color;
    n->parent->color = BLACK;

    if (n == n->parent->left) {
        s->right->color = BLACK;
        rotate_left(n->parent);
    } else {
        s->left->color = BLACK;
        rotate_right(n->parent);
    }
}
```

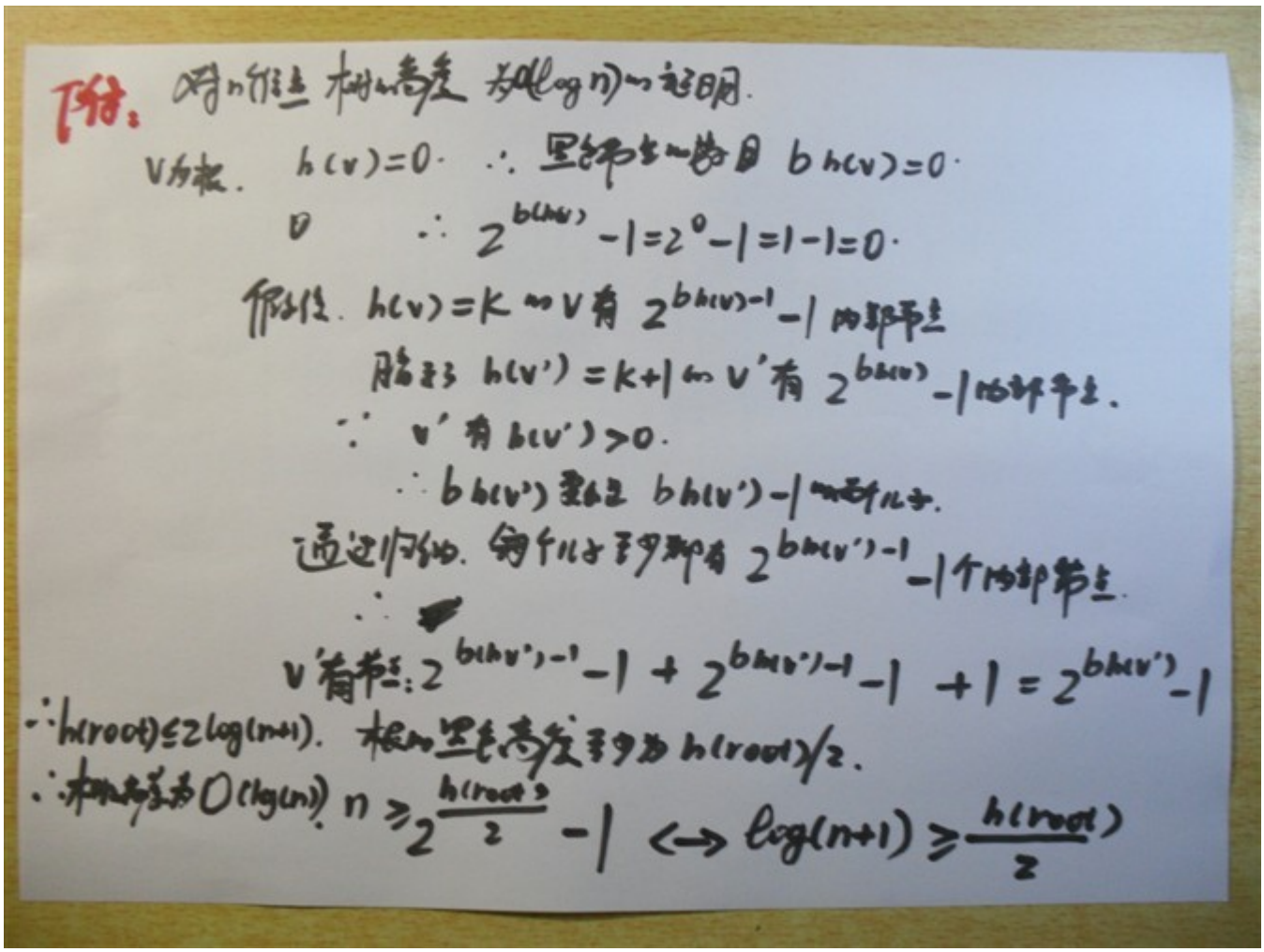


//呵呵，画这 12 张图，直接从中午画到了晚上。希望，此文能让你明白。

四、红黑树的插入、删除情况时间复杂度的分析

因为每一个红黑树也是一个特化的二叉查找树，
因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。
然而，在红黑树上进行插入操作和删除操作会导致不再符合红黑树的性质。

恢复红黑树的属性需要少量(O(log n))的颜色变更(实际是非常快速的)和
不超过三次树旋转(对于插入操作是两次)。
虽然插入和删除很复杂，但操作时间仍可以保持为 O(log n) 次。



ok，完。

后记：
此红黑树系列，前前后后，已经写了 4 篇文章，如果读者读完了这 4 篇文章，
对红黑树有一个相对之前来说，比较透彻的理解，
那么，也不枉费，我花这么多篇幅、花好几个钟头去画红黑树了。

真正理解一个数据结构、算法，最紧要的还是真正待用、实践的时候体会。
欢迎，各位，将现在、或以后学习、工作中运用此红黑树结构、算法的经验与我分享。
谢谢。:D。