http://www.cnblogs.com/skywang12345/p/3245399.html

R-B Tree简介

R-B Tree，全称是Red-Black Tree，又称为"红黑树"，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树的特性:

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

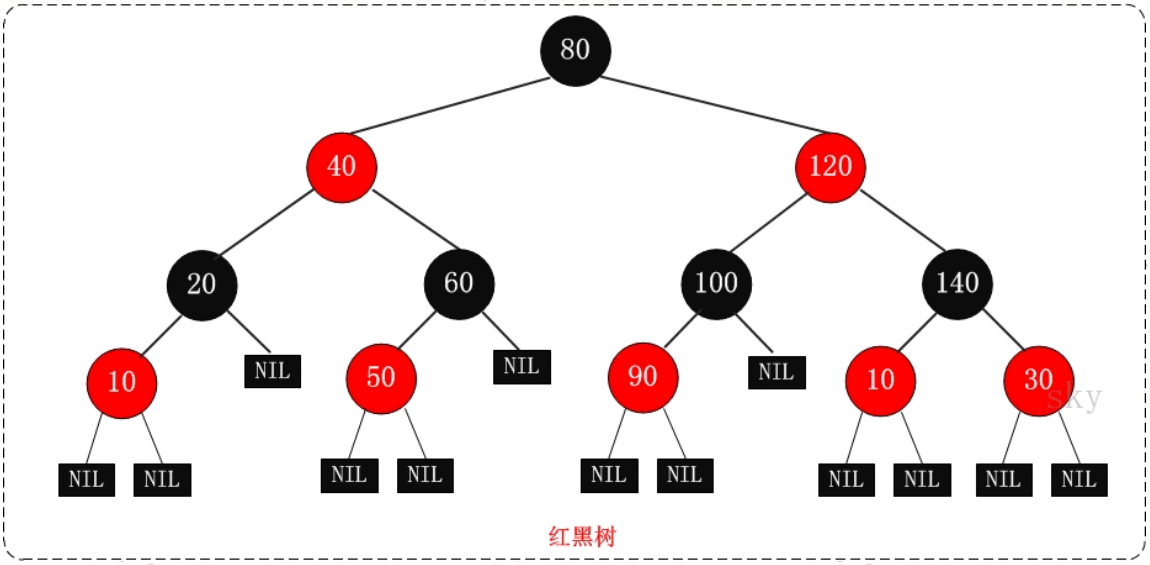
（5）从一个节点到该节点的子孙节点(NIL)的所有路径上包含相同数目的黑节点。

注意:

(01) 特性(3)中的叶子节点，是只为空(NIL或null)的节点。

(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。

红黑树示意图如下:



红黑树

红黑树的应用

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高。

例如，Java集合中的TreeSet和TreeMap，C++ STL中的set、map，以及Linux虚拟内存的管理，都是通过红黑树去实现的。

红黑树的时间复杂度和相关证明

红黑树的时间复杂度为: O(lgn)

下面通过"*数学归纳法*"对红黑树的时间复杂度进行证明。

定理：一棵含有**n**个节点的红黑树的高度至多为**2log(n+1)**.

证明：

"一棵含有n个节点的红黑树的高度至多为2log(n+1)" 的**逆否命题**是 "高度为h的红黑树，它的包含的内节点个数至少为2h/2-1个"。

我们只需要证明逆否命题，即可证明原命题为真；即只需证明 "高度为h的红黑树，它的包含的内节点个数至少为2h/2-1个"。

从某个节点x出发（不包括该节点）到达一个叶节点的任意一条路径上，黑色节点的个数称为该节点的黑高度(x's black height)，记为**bh(x)**。关于bh(x)有两点需要说明：

第1点：根据红黑树的"**特性(5)**，即*从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点*"可知，从节

点x出发到达的所有的叶节点具有相同数目的黑节点。**这也就意味着，bh(x)的值是唯一的**!

第2点：根据红黑色的"特性(4)，即*如果一个节点是红色的，则它的子节点必须是黑色的*"可知，从节点x出发达到叶节点"所经历的黑节点数目">= "所经历的红节点的数目"。假设x是根节点，则可以得出结论"**bh(x) >= h/2**"。进而，我们只需证明 "高度为h的红黑树，它的包含的黑节点个数至少为 2bh(x)-1个"即可。

到这里，我们将需要证明的定理已经由

"**一棵含有n个节点的红黑树的高度至多为2log(n+1)**"

转变成只需要证明

"**高度为h的红黑树，它的包含的内节点个数至少为 2bh(x)-1个**"。

下面通过"数学归纳法"开始论证高度为h的红黑树，它的包含的内节点个数至少为 2bh(x)-1个"。

(01) 当树的高度h=0时，

内节点个数是0，bh(x) 为0，2bh(x)-1 也为 0。显然，原命题成立。

(02) 当h>0，且树的高度为 h-1 时，它包含的节点个数至少为 2bh(x)-1-1。这个是根据(01)推断出来的!

下面，由树的高度为 h-1 的已知条件推出"树的高度为 h 时，它所包含的节点树为 2bh(x)-1"。

当树的高度为 h 时，

对于节点x(x为根节点)，其黑高度为bh(x)。

对于节点x的左右子树，它们黑高度为 bh(x) 或者 bh(x)-1。

根据(02)的已知条件，我们已知 "x的左右子树，即高度为 h-1 的节点，它包含的节点至少为 2bh(x)-1-1 个";

所以，节点x所包含的节点至少为 (2bh(x)-1-1) + (2bh(x)-1-1) + 1 = 2^bh(x)-1。即节点x所包含的节点至少为 2bh(x)-1。

因此，原命题成立。

由(01)、(02)得出，"高度为h的红黑树，它的包含的内节点个数至少为 2^bh(x)-1个"。

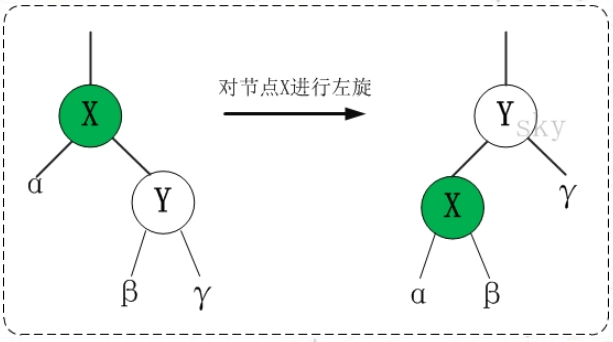
因此，"一棵含有n个节点的红黑树的高度至多为2log(n+1)"。

红黑树的基本操作(一) 左旋和右旋

红黑树的基本操作是**添加**、**删除**。在对红黑树进行添加或删除之后，都会用到旋转方法。为什么呢？道理很简单，添加或删除红黑树中的节点之后，红黑树就发生了变化，可能不满足红黑树的5条性质，也就不再是一颗红黑树了，而是一颗普通的树。而通过旋转，可以使这颗树重新成为红黑树。简单点说，旋转的目的是让树保持红黑树的特性。

旋转包括两种：**左旋** 和 **右旋**。下面分别对它们进行介绍。

1. 左旋(逆时针,自己变为右孩子的左孩子)



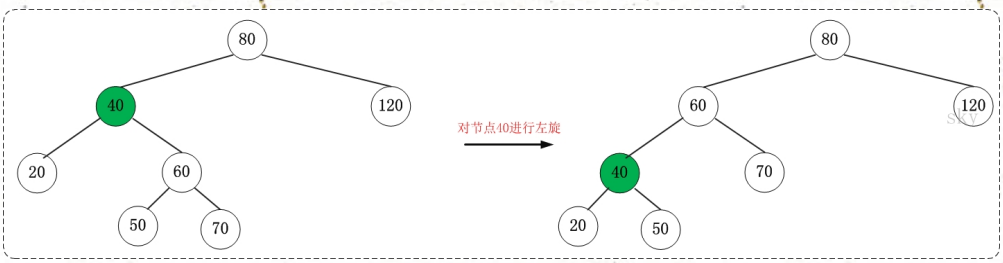
对x进行左旋，意味着"将x变成一个左节点"。

左旋的伪代码《算法导论》：参考上面的示意图和下面的伪代码，理解"红黑树T的节点x进行左旋"是如何进行的。

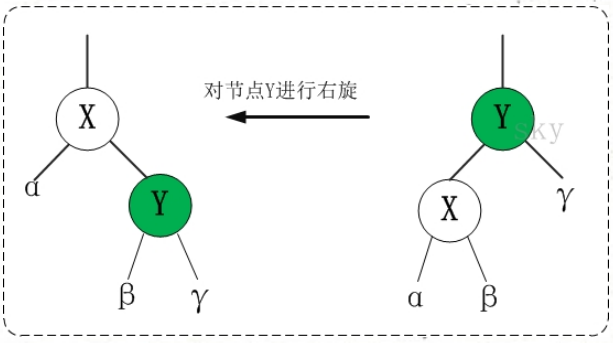
```
LEFT-ROTATE(T, x)
01  y ← right[x]              // 前提：这里假设x的右孩子为y。下面开始正式操作
02  right[x] ← left[y]        // 将 "y的左孩子" 设为 "x的右孩子"，即 将β设为x的右孩子
03  p[left[y]] ← x            // 将 "x" 设为 "y的左孩子的父亲"，即 将β的父亲设为x
04  p[y] ← p[x]               // 将 "x的父亲" 设为 "y的父亲"
05  if p[x] = nil[T]
06  then root[T] ← y                  // 情况1：如果 "x的父亲" 是空节点，则将y设为根节点
07  else if x = left[p[x]]
08            then left[p[x]] ← y     // 情况2：如果 x是它父节点的左孩子，则将y设为"x的父节点的左孩子"
09            else right[p[x]] ← y    // 情况3：(x是它父节点的右孩子) 将y设为"x的父节点的右孩子"
```

```
10  left[y] ← x               // 将 “x” 设为 “y的左孩子”
11  p[x] ← y                  // 将 “x的父节点” 设为 “y”
```

理解左旋之后，看看下面一个更鲜明的例子。你可以先不看右边的结果，自己尝试一下。



2. 右旋(顺时针,自己变为左孩子的右孩子)

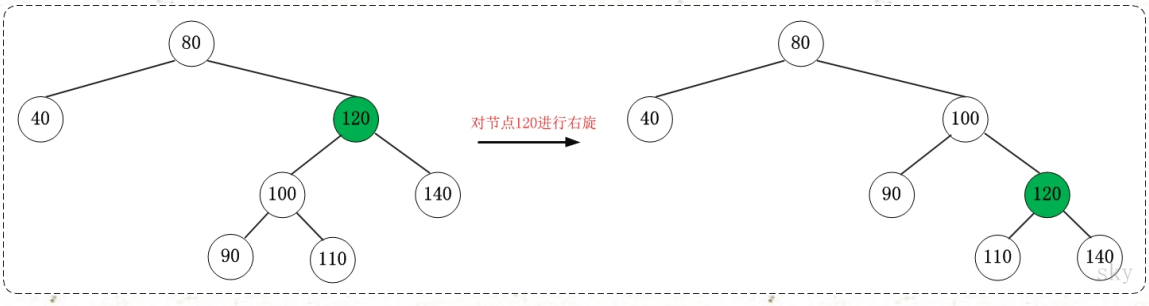


对Y进行右旋，意味着"将Y变成一个右节点"。

右旋的伪代码《算法导论》：参考上面的示意图和下面的伪代码，理解“红黑树T的节点y进行右旋”是如何进行的。 。

```
RIGHT-ROTATE(T, y)
01  x ← left[y]               // 前提：这里假设y的左孩子为x。下面开始正式操作
02  left[y] ← right[x]        // 将 “x的右孩子” 设为 “y的左孩子”，即 将β设为y的左孩子
03  p[right[x]] ← y           // 将 “y” 设为 “x的右孩子的父亲”，即 将β的父亲设为y
04  p[x] ← p[y]               // 将 “y的父亲” 设为 “x的父亲”
05  if p[y] = nil[T]
06  then root[T] ← x                  // 情况1：如果 “y的父亲” 是空节点，则将x设为根节点
07  else if y = right[p[y]]
08            then right[p[y]] ← x    // 情况2：如果 y是它父节点的右孩子，则将x设为“y的父节点的左孩子”
09            else left[p[y]] ← x     // 情况3：(y是它父节点的左孩子) 将x设为“y的父节点的左孩子”
10  right[x] ← y              // 将 “y” 设为 “x的右孩子”
11  p[y] ← x                  // 将 “y的父节点” 设为 “x”
```

理解右旋之后，看看下面一个更鲜明的例子。你可以先不看右边的结果，自己尝试一下。



旋转总结：

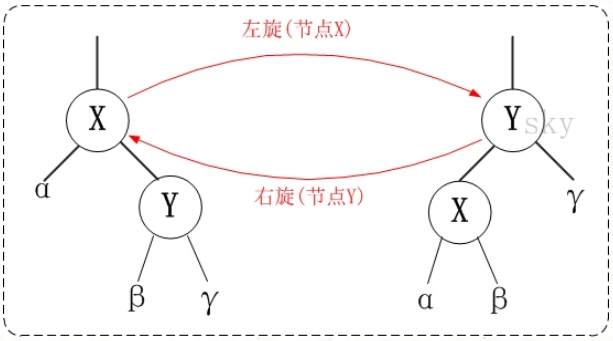
(01) 左旋 和 右旋 是相对的两个概念，原理类似。理解一个也就理解了另一个。

(02) 下面谈谈如何区分 左旋 和 右旋。

在实际应用中，若没有彻底理解 左旋 和 右旋，可能会将它们混淆。下面谈谈我对如何区分 左旋 和 右旋 的理解。

3. 区分 左旋 和 右旋

仔细观察上面"左旋"和"右旋"的示意图。我们能清晰的发现，它们是对称的。无论是左旋还是右旋，被旋转的树，在旋转前是二叉查找树，并且旋转之后仍然是一颗二叉查找树。



左旋示例图(以x为节点进行左旋)：

```
                                z
   x                           /
  / \        --(x左旋)-->      x
 y   z                         /
                              y
```

对x进行左旋，意味着，将"x的右孩子"设为"x的父亲节点"；即，将 x变成了一个左节点(x成了为z的左孩子)！。 因此，**左旋中的"左"，意味着"被旋转的节点将变成一个左节点"。**

右旋示例图(以x为节点进行右旋)：

```
                              y
   x                           \
  / \        --(右旋)-->       x
 y   z                         \
                                z
```

对x进行右旋，意味着，将"x的左孩子"设为"x的父亲节点"；即，将 x变成了一个右节点(x成了为y的右孩子)！ 因此，**右旋中的"右"，意味着"被旋转的节点将变成一个右节点"。**

红黑树的基本操作(二) 添加

将一个节点插入到红黑树中，需要执行哪些步骤呢？首先，将红黑树当作一颗二叉查找树，将节点插入；然后，将节点着色为红色；最后，通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。详细描述如下：

第一步: 将红黑树当作一颗二叉查找树，将节点插入。

红黑树本身就是一颗二叉查找树，将节点插入后，该树仍然是一颗二叉查找树。也就意味着，树的键值仍然是有序的。此外，无论是左旋还是右旋，若旋转之前这棵树是二叉查找树，旋转之后它一定还是二叉查找树。这也就意味着，任何的旋转和重新着色操作，都不会改变它仍然是一颗二叉查找树的事实。

好吧？那接下来，我们就来想方设法的旋转以及重新着色，使这颗树重新成为红黑树！

第二步：将插入的节点着色为"红色"。

为什么着色成红色，而不是黑色呢？为什么呢？在回答之前，我们需要重新温习一下红黑树的特性：

(1) 每个节点或者是黑色，或者是红色。
(2) 根节点是黑色。
(3) 每个叶子节点是黑色。 [注意：这里叶子节点，是指为空的叶子节点(Nil节点)！]
(4) 如果一个节点是红色的，则它的子节点必须是黑色的。
(5) 从一个节点到该节点的子孙节点(Nil节点)的所有路径上包含相同数目的黑节点。

将插入的节点着色为红色，不会违背"特性(5)"！少违背一条特性，就意味着我们需要处理的情况越少。接下来，就要努力的让这棵树满足其它性质即可；满足了的话，它就又是一颗红黑树了。o(∩∩)o...哈哈

第三步: 通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。

第二步中，将插入节点着色为"红色"之后，不会违背"特性(5)"。那它到底会违背哪些特性呢？

对于"特性(1)"，显然不会违背了。因为我们已经将它涂成红色了。

对于"特性(2)"，显然也不会违背。在第一步中，我们是将红黑树当作二叉查找树，然后执行的插入操作。而根据二叉

查找数的特点，插入操作不会改变根节点。所以，根节点仍然是黑色。

对于"特性(3)"，显然不会违背了。这里的叶子节点是指的空叶子节点，插入非空节点并不会对它们造成影响。

对于"特性(4)"，是有可能违背的!

那接下来，想办法使之"满足特性(4)"，就可以将树重新构造成红黑树了。

下面看看代码到底是怎样实现这三步的。

添加操作的伪代码《算法导论》

```
RB-INSERT(T, z)
01  y ← nil[T]                          // 新建节点"y"，将y设为空节点。
02  x ← root[T]                         // 设"红黑树T"的根节点为"x"
03  while x ≠ nil[T]                    // 找出要插入的节点"z"在二叉树T中的位置"y"
04      do y ← x
05         if key[z] < key[x]
06            then x ← left[x]
07            else x ← right[x]
08  p[z] ← y                            // 设置 "z的父亲" 为 "y"
09  if y = nil[T]
10     then root[T] ← z                 // 情况1：若y是空节点，则将z设为根
11     else if key[z] < key[y]
12             then left[y] ← z         // 情况2：若"z所包含的值" < "y所包含的值"，则将z设为"y的左孩子"
13             else right[y] ← z        // 情况3：("z所包含的值" >= "y所包含的值")将z设为"y的右孩子"
14  left[z] ← nil[T]                    // z的左孩子设为空
15  right[z] ← nil[T]                   // z的右孩子设为空。至此，已经完成将"节点z插入到二叉树"中了。
16  color[z] ← RED                      // 将z着色为"红色"
17  RB-INSERT-FIXUP(T, z)               // 通过RB-INSERT-FIXUP对红黑树的节点进行颜色修改以及旋转，让树T仍然是一颗红
黑树
```

结合伪代码以及为代码上面的说明，先理解RB-INSERT。理解了RB-INSERT之后，我们接着对 RB-INSERT-FIXUP的伪代码进行说明。

添加修正操作的伪代码《算法导论》

```
RB-INSERT-FIXUP(T, z)
01 while color[p[z]] = RED              // 若"当前节点(z)的父节点是红色"，则进行以下处理。
02     do if p[z] = left[p[p[z]]]             // 若"z的父节点"是"z的祖父节点的左孩子"，则进行以下处理。
03           then y ← right[p[p[z]]]             // 将y设置为"z的叔叔节点(z的祖父节点的右孩子)"
04                if color[y] = RED                    // Case 1条件：叔叔是红色
05                   then color[p[z]] ← BLACK         ▷ Case 1   //  (01) 将"父节点"设为黑色。
06                        color[y] ← BLACK            ▷ Case 1   //  (02) 将"叔叔节点"设为黑色。
07                        color[p[p[z]]] ← RED        ▷ Case 1   //  (03) 将"祖父节点"设为"红色"。
08                        z ← p[p[z]]                 ▷ Case 1   //  (04) 将"祖父节点"设为"当前节点"(红色节点)
09                   else if z = right[p[z]]                     // Case 2条件：叔叔是黑色，且当前节点是右孩子
10                           then z ← p[z]               ▷ Case 2   //  (01) 将"父节点"作为"新的当前节点"。
11                                LEFT-ROTATE(T, z)      ▷ Case 2   //  (02) 以"新的当前节点"为支点进行左旋。
12                           color[p[z]] ← BLACK         ▷ Case 3   // Case 3条件：叔叔是黑色，且当前节点是左孩
子。(01) 将"父节点"设为"黑色"。
13                           color[p[p[z]]] ← RED        ▷ Case 3   //  (02) 将"祖父节点"设为"红色"。
14                           RIGHT-ROTATE(T, p[p[z]])    ▷ Case 3   //  (03) 以"祖父节点"为支点进行右旋。
15        else (same as then clause with "right" and "left" exchanged)      // 若"z的父节点"是"z的祖父节点的
右孩子"，将上面的操作中"right"和"left"交换位置，然后依次执行。
16 color[root[T]] ← BLACK
```

根据被插入节点的父节点的情况，可以将"当节点z被着色为红色节点，并插入二叉树"划分为三种情况来处理。

① 情况说明：被插入的节点是根节点。

处理方法：直接把此节点涂为黑色。

② 情况说明：被插入的节点的父节点是黑色。

处理方法：什么也不需要做。节点被插入后，仍然是红黑树。

③ 情况说明：被插入的节点的父节点是红色。

处理方法：那么，该情况与红黑树的"特性(5)"相冲突。这种情况下，被插入节点是一定存在非空祖父节点的；进一步的讲，被插入节点也一定存在叔叔节点(即使叔叔节点为空，我们也视之为存在，空节点本身就是黑色节点)。理解这点之后，我们依据"叔叔节点的情况"，将这种情况进一步划分为3种情况(Case)。

	现象说明	处理策略
Case 1	当前节点的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。	(01) 将“父节点”设为黑色。 (02) 将“叔叔节点”设为黑色。 (03) 将“祖父节点”设为“红色”。 (04) 将“祖父节点”设为“当前节点”(红色节点)；即，之后继续对“当前节点”进行操作。
Case 2	当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子	(01) 将“父节点”作为“新的当前节点”。 (02) 以“新的当前节点”为支点进行左旋。
Case 3	当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子	(01) 将“父节点”设为“黑色”。 (02) 将“祖父节点”设为“红色”。 (03) 以“祖父节点”为支点进行右旋。

上面三种情况(Case)处理问题的核心思路都是：将红色的节点移到根节点；然后，将根节点设为黑色。下面对它们详细进行介绍。

1. (Case 1)叔叔是红色

1.1 现象说明

当前节点(即，被插入节点)的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。

1.2 处理策略

(01) 将“父节点”设为黑色。
(02) 将“叔叔节点”设为黑色。
(03) 将“祖父节点”设为“红色”。
(04) 将“祖父节点”设为“当前节点”(红色节点)；即，之后继续对“当前节点”进行操作。

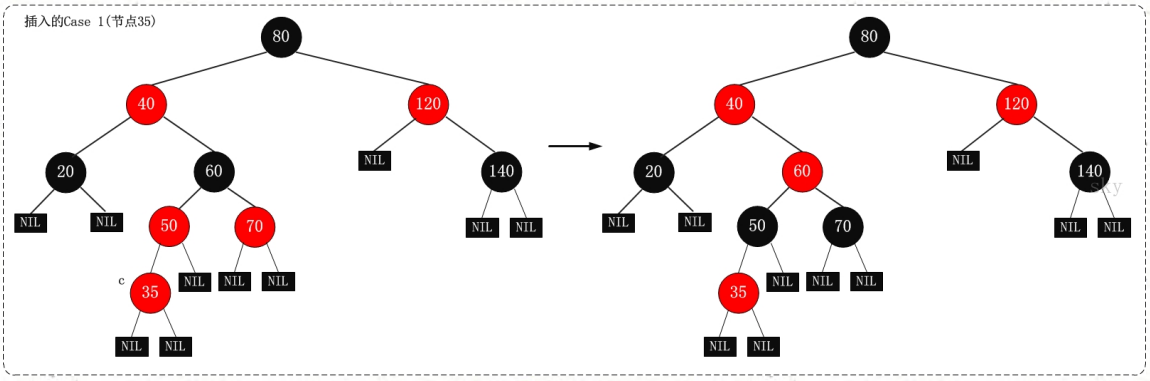
下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

“当前节点”和“父节点”都是红色，违背“特性(4)”。所以，将“父节点”设置“黑色”以解决这个问题。

但是，将“父节点”由“红色”变成“黑色”之后，违背了“特性(5)”：因为，包含“父节点”的分支的黑色节点的总数增加了1。 解决这个问题的办法是：将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”。关于这里，说明几点：第一，为什么“祖父节点”之前是黑色？这个应该很容易想明白，因为在变换操作之前，该树是红黑树，“父节点”是红色，那么“祖父节点”一定是黑色。 第二，为什么将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”；能解决“包含‘父节点’的分支的黑色节点的总数增加了1”的问题。这个道理也很简单。“包含‘父节点’的分支的黑色节点的总数增加了1” 同时也意味着 “包含‘祖父节点’的分支的黑色节点的总数增加了1”，既然这样，我们通过将“祖父节点”由“黑色”变成“红色”以解决“包含‘祖父节点’的分支的黑色节点的总数增加了1”的问题； 但是，这样处理之后又会引起另一个问题“包含‘叔叔’节点的分支的黑色节点的总数减少了1”，现在我们已知“叔叔节点”是“红色”，将“叔叔节点”设为“黑色”就能解决这个问题。 所以，将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”；就解决了该问题。

按照上面的步骤处理之后：当前节点、父节点、叔叔节点之间都不会违背红黑树特性，但祖父节点却不一定。若此时，祖父节点是根节点，直接将祖父节点设为“黑色”，那就完全解决这个问题了；若祖父节点不是根节点，那我们需要将“祖父节点”设为“新的当前节点”，接着对“新的当前节点”进行分析。

1.3 示意图



2. (Case 2)叔叔是黑色，且当前节点是右孩子

2.1 现象说明

当前节点(即，被插入节点)的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子

2.2 处理策略

(01) 将"父节点"作为"新的当前节点"。
(02) 以"新的当前节点"为支点进行左旋。

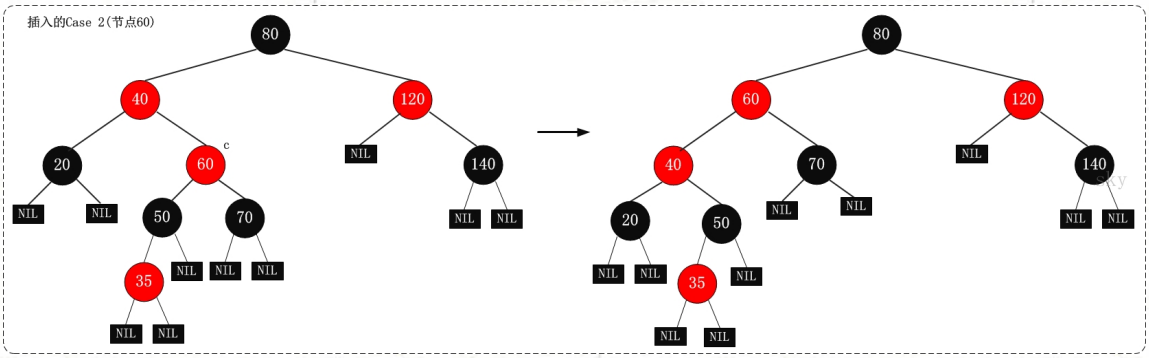
下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

首先，将"父节点"作为"新的当前节点"；接着，以"新的当前节点"为支点进行左旋。 为了便于理解，我们先说明第(02)步，再说明第(01)步；为了便于说明，我们设置"父节点"的代号为F(Father)，"当前节点"的代号为S(Son)。

为什么要"以F为支点进行左旋"呢？根据已知条件可知：S是F的右孩子。而之前我们说过，我们处理红黑树的核心思想：将红色的节点移到根节点；然后，将根节点设为黑色。既然是"将红色的节点移到根节点"，那就是说要不断的将破坏红黑树特性的红色节点上移(即向根方向移动)。 而S又是一个右孩子，因此，我们可以通过"左旋"来将S上移！

按照上面的步骤(以F为支点进行左旋)处理之后：若S变成了根节点，那么直接将其设为"黑色"，就完全解决问题了；若S不是根节点，那我们需要执行步骤(01)，即"将F设为'新的当前节点'"。那为什么不继续以S为新的当前节点继续处理，而需要以F为新的当前节点来进行处理呢？这是因为"左旋"之后，F变成了S的"子节点"，即S变成了F的父节点；而我们处理问题的时候，需要从下至上(由叶到根)方向进行处理；也就是说，必须先解决"孩子"的问题，再解决"父亲"的问题；所以，我们执行步骤(01)：将"父节点"作为"新的当前节点"。

2.2 示意图



3. (Case 3)叔叔是黑色，且当前节点是左孩子

3.1 现象说明

当前节点(即，被插入节点)的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子

3.2 处理策略

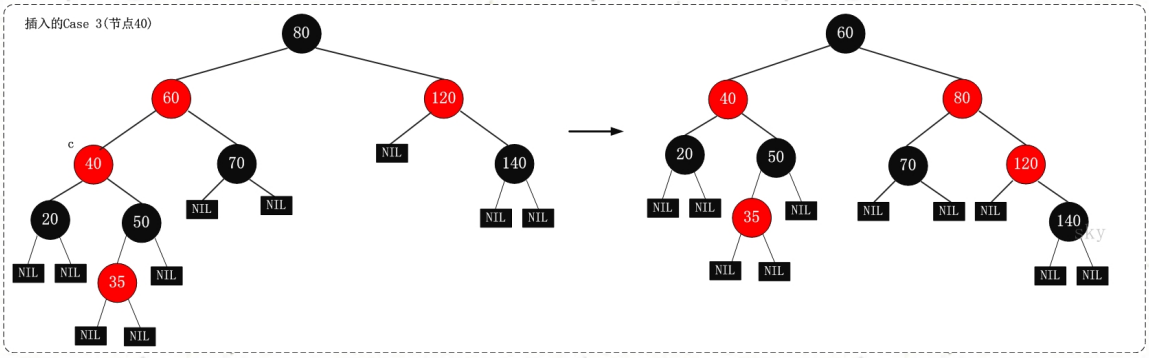
(01) 将"父节点"设为"黑色"。
(02) 将"祖父节点"设为"红色"。
(03) 以"祖父节点"为支点进行右旋。

下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

为了便于说明，我们设置"当前节点"为S(Original Son)，"兄弟节点"为B(Brother)，"叔叔节点"为U(Uncle)，"父节点"为F(Father)，祖父节点为G(Grand-Father)。

S和F都是红色，违背了红黑树的"特性(4)"，我们可以将F由"红色"变为"黑色"，就解决了"违背'特性(4)'"的问题；但却引起了其它问题：违背特性(5)，因为将F由红色改为黑色之后，所有经过F的分支的黑色节点的个数增加了1。那我们如何解决"所有经过F的分支的黑色节点的个数增加了1"的问题呢？ 我们可以通过"将G由黑色变成红色"，同时"以G为支点进行右旋"来解决。

2.3 示意图



提示：上面的进行Case 3处理之后，再将节点"120"当作当前节点，就变成了Case 2的情况。

红黑树的基本操作(三) 删除

将红黑树内的某一个节点删除。需要执行的操作依次是：首先，将红黑树当作一颗二叉查找树，将该节点从二叉查找树中删除；然后，通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。详细描述如下：

第一步：将红黑树当作一颗二叉查找树，将节点删除。

这和"删除常规二叉查找树中删除节点的方法是一样的"。分3种情况：

① 被删除节点没有儿子，即为叶节点。那么，直接将该节点删除就OK了。

② 被删除节点只有一个儿子。那么，直接删除该节点，并用该节点的唯一子节点顶替它的位置。

③ 被删除节点有两个儿子。那么，先找出它的后继节点；然后把“它的后继节点的内容”复制给“该节点的内容”；之后，删除“它的后继节点”。在这里，后继节点相当于替身，在将后继节点的内容复制给"被删除节点"之后，再将后继节点删除。这样就巧妙的将问题转换为"删除后继节点"的情况了，下面就考虑后继节点。 在"被删除节点"有两个非空子节点的情况下，它的后继节点不可能是双子非空。既然"的后继节点"不可能双子都非空，就意味着"该节点的后继节点"要么没有儿子，要么只有一个儿子。若没有儿子，则按"情况① "进行处理；若只有一个儿子，则按"情况② "进行处理。

第二步：通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。

因为"第一步"中删除节点之后，可能会违背红黑树的特性。所以需要通过"旋转和重新着色"来修正该树，使之重新成为一棵红黑树。

删除操作的伪代码《算法导论》

```
RB-DELETE(T, z)
01 if left[z] = nil[T] or right[z] = nil[T]
02    then y ← z                                  // 若“z的左孩子” 或 “z的右孩子”为空，则将“z”赋值给 “y”；
03    else y ← TREE-SUCCESSOR(z)                  // 否则，将“z的后继节点”赋值给 “y”。
04 if left[y] ≠ nil[T]
05    then x ← left[y]                            // 若“y的左孩子” 不为空，则将“y的左孩子” 赋值给 “x”；
06    else x ← right[y]                           // 否则，“y的右孩子” 赋值给 “x”。
07 p[x] ← p[y]                                    // 将“y的父节点” 设置为 “x的父节点”
08 if p[y] = nil[T]
09    then root[T] ← x                            // 情况1：若“y的父节点” 为空，则设置“x” 为 “根节点”。
10    else if y = left[p[y]]
11            then left[p[y]] ← x                 // 情况2：若“y是它父节点的左孩子”，则设置“x” 为 “y的父节点的左孩子”
12            else right[p[y]] ← x                // 情况3：若“y是它父节点的右孩子”，则设置“x” 为 “y的父节点的右孩子”
13 if y ≠ z
14    then key[z] ← key[y]                        // 若“y的值” 赋值给 “z”。注意：这里只拷贝z的值给y，而没有拷贝z的颜色！！！
15         copy y's satellite data into z
16 if color[y] = BLACK
17    then RB-DELETE-FIXUP(T, x)                  // 若“y为黑节点”，则调用
18 return y
```

结合伪代码以及为代码上面的说明，先理解RB-DELETE。理解了RB-DELETE之后，接着对 RB-DELETE-FIXUP的伪代码进行说明

```
RB-DELETE-FIXUP(T, x)
01 while x ≠ root[T] and color[x] = BLACK
02     do if x = left[p[x]]
03           then w ← right[p[x]]                                          // 若 “x”是“它父节点的左孩子”，则设置 “w”为“x的叔叔”(即x为它父节点的右孩子)
04                if color[w] = RED                                        // Case 1: x是“黑+黑”节点，x的兄弟节点是红色。(此时x的父节点和x的兄弟节点的子节点都是黑节点)。
05                   then color[w] ← BLACK                  ▷ Case 1    //    (01) 将x的兄弟节点设为“黑色”。
06                        color[p[x]] ← RED                 ▷ Case 1    //    (02) 将x的父节点设为“红色”。
07                        LEFT-ROTATE(T, p[x])              ▷ Case 1    //    (03) 对x的父节点进行左旋。
08                        w ← right[p[x]]                   ▷ Case 1    //    (04) 左旋后，重新设置x的兄弟节点。
09                if color[left[w]] = BLACK and color[right[w]] = BLACK    // Case 2: x是“黑+黑”节点，x
```

```
的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色。
10                    then color[w] ← RED                              ▷ Case 2   //   (01) 将x的兄弟节点设为"红色"。
11                         x ←  p[x]                                   ▷ Case 2   //   (02) 设置"x的父节点"为"新的x节点"。
12                    else if color[right[w]] = BLACK                                 // Case 3: x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的。
13                           then color[left[w]] ← BLACK               ▷ Case 3   //   (01) 将x兄弟节点的左孩子设为"黑色"。
14                                color[w] ← RED                       ▷ Case 3   //   (02) 将x兄弟节点设为"红色"。
15                                RIGHT-ROTATE(T, w)                   ▷ Case 3   //   (03) 对x的兄弟节点进行右旋。
16                                w ← right[p[x]]                      ▷ Case 3   //   (04) 右旋后，重新设置x的兄弟节点。
17                         color[w] ← color[p[x]]                      ▷ Case 4   // Case 4: x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的。(01) 将x父节点颜色 赋值给 x的兄弟节点。
18                         color[p[x]] ← BLACK                         ▷ Case 4   //   (02) 将x父节点设为"黑色"。
19                         color[right[w]] ← BLACK                     ▷ Case 4   //   (03) 将x兄弟节点的右子节设为"黑色"。
20                         LEFT-ROTATE(T, p[x])                        ▷ Case 4   //   (04) 对x的父节点进行左旋。
21                         x ← root[T]                                 ▷ Case 4   //   (05) 设置"x"为"根节点"。
22        else (same as then clause with "right" and "left" exchanged)            // 若 "x"是"它父节点的右孩子"，将上面的操作中"right"和"left"交换位置，然后依次执行。
23 color[x] ← BLACK
```

下面对删除函数进行分析。在分析之前，我们再次温习一下红黑树的几个特性：

(1) 每个节点或者是黑色，或者是红色。

(2) 根节点是黑色。

(3) 每个叶子节点是黑色。 [注意：这里叶子节点，是指为空的叶子节点！]

(4) 如果一个节点是红色的，则它的子节点必须是黑色的。

(5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

前面我们将"删除红黑树中的节点"大致分为两步，在第一步中"将红黑树当作一颗二叉查找树，将节点删除"后，可能违反"特性(2)、(4)、(5)"三个特性。第二步需要解决上面的三个问题，进而保持红黑树的全部特性。

为了便于分析，我们假设"x包含一个额外的黑色"(x原本的颜色还存在)，这样就不会违反"特性(5)"。为什么呢？

通过RB-DELETE算法，我们知道：删除节点y之后，x占据了原来节点y的位置。 既然删除y(y是黑色)，意味着减少一个黑色节点；那么，再在该位置上增加一个黑色即可。这样，当我们假设"x包含一个额外的黑色"，就正好弥补了"删除y所丢失的黑色节点"，也就不会违反"特性(5)"。 因此，假设"x包含一个额外的黑色"(x原本的颜色还存在)，这样就不会违反"特性(5)"。

现在，x不仅包含它原本的颜色属性，x还包含一个额外的黑色。即x的颜色属性是"红+黑"或"黑+黑"，它违反了"特性(1)"。

现在，我们面临的问题，由解决"违反了特性(2)、(4)、(5)三个特性"转换成了"解决违反特性(1)、(2)、(4)三个特性"。RB-DELETE-FIXUP需要做的就是通过算法恢复红黑树的特性(1)、(2)、(4)。RB-DELETE-FIXUP的思想是：将x所包含的额外的黑色不断沿树上移(向根方向移动)，直到出现下面的姿态：

a) x指向一个"红+黑"节点。此时，将x设为一个"黑"节点即可。

b) x指向根。此时，将x设为一个"黑"节点即可。

c) 非前面两种姿态。

将上面的姿态，可以概括为3种情况。

① 情况说明：x是"红+黑"节点。

处理方法：直接把x设为黑色，结束。此时红黑树性质全部恢复。

② 情况说明：x是"黑+黑"节点，且x是根。

处理方法：什么都不做，结束。此时红黑树性质全部恢复。

③ 情况说明：x是"黑+黑"节点，且x不是根。

处理方法：这种情况又可以划分为4种子情况。这4种子情况如下表所示：

	现象说明	处理策略
Case 1	x是"黑+黑"节点，x的兄弟节点是红色。(此时x的父节点和x的兄弟节点的子节点都是黑节点)。	(01) 将x的兄弟节点设为"黑色"。 (02) 将x的父节点设为"红色"。 (03) 对x的父节点进行左旋。 (04) 左旋后，重新设置x的兄弟节点。
Case 2	x是"黑+黑"节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色。	(01) 将x的兄弟节点设为"红色"。 (02) 设置"x的父节点"为"新的x节点"。
Case 3	x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的。	(01) 将x兄弟节点的左孩子设为"黑色"。 (02) 将x兄弟节点设为"红色"。 (03) 对x的兄弟节点进行右旋。 (04) 右旋后，重新设置x的兄弟节点。
Case 4	x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的，x的兄弟节	(01) 将x父节点颜色 赋值给 x的兄弟节点。 (02) 将x父节点设为"黑色"。 (03) 将x兄弟节点的右子节设为"黑色"。

1. (Case 1)x是"黑+黑"节点，x的兄弟节点是红色

1.1 现象说明

x是"黑+黑"节点，x的兄弟节点是红色。(此时x的父节点和x的兄弟节点的子节点都是黑节点)。

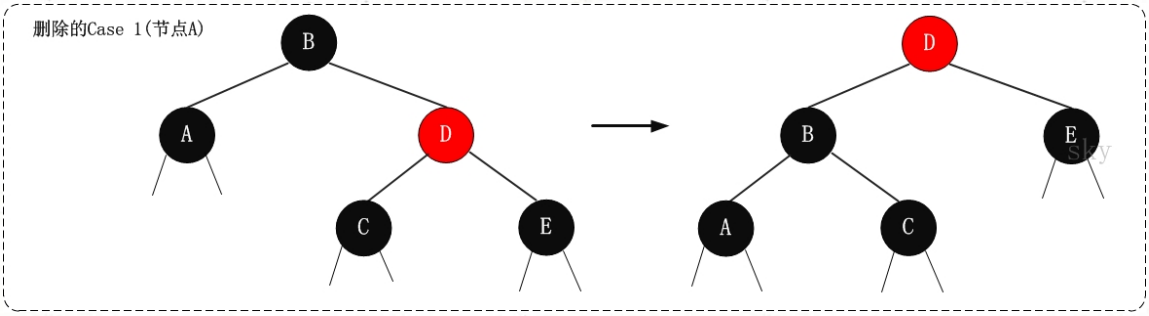
1.2 处理策略

(01) 将x的兄弟节点设为"黑色"。
(02) 将x的父节点设为"红色"。
(03) 对x的父节点进行左旋。
(04) 左旋后，重新设置x的兄弟节点。

下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

这样做的目的是将"Case 1"转换为"Case 2"、"Case 3"或"Case 4"，从而进行进一步的处理。对x的父节点进行左旋；左旋后，为了保持红黑树特性，就需要在左旋前"将x的兄弟节点设为黑色"，同时"将x的父节点设为红色"；左旋后，由于x的兄弟节点发生了变化，需要更新x的兄弟节点，从而进行后续处理。

1.3 示意图



2. (Case 2) x是"黑+黑"节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色

2.1 现象说明

x是"黑+黑"节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色。

2.2 处理策略

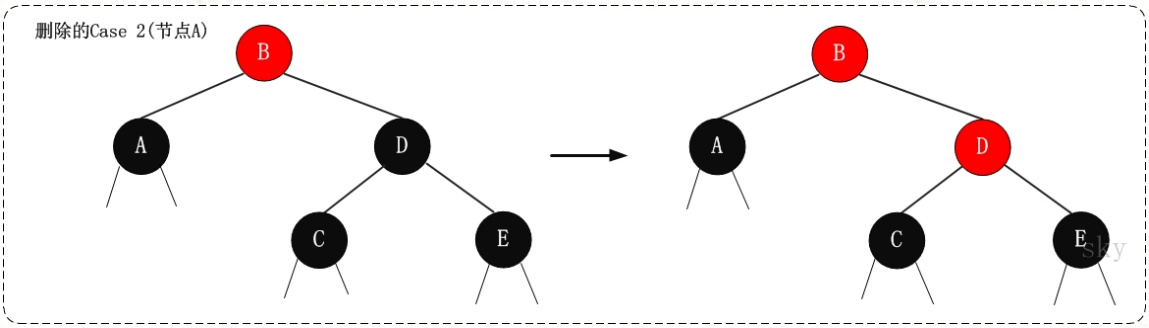
(01) 将x的兄弟节点设为"红色"。
(02) 设置"x的父节点"为"新的x节点"。

下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

这个情况的处理思想：是将"x中多余的一个黑色属性上移(往根方向移动)"。 x是"黑+黑"节点，我们将x由"黑+黑"节点 变成 "黑"节点，多余的一个"黑"属性移到x的父节点中，即x的父节点多出了一个黑属性(若x的父节点原先是"黑"，则此时变成了"黑+黑"；若x的父节点原先时"红"，则此时变成了"红+黑")。 此时，需要注意的是：所有经过x的分支中黑节点个数没变化；但是，所有经过x的兄弟节点的分支中黑色节点的个数增加了1(因为x的父节点多了一个黑色属性)！为了解决这个问题，我们需要将"所有经过x的兄弟节点的分支中黑色节点的个数减1"即可，那么就可以通过"将x的兄弟节点由黑色变成红色"来实现。

经过上面的步骤(将x的兄弟节点设为红色)，多余的一个颜色属性(黑色)已经跑到x的父节点中。我们需要将x的父节点设为"新的x节点"进行处理。若"新的x节点"是"黑+红"，直接将"新的x节点"设为黑色，即可完全解决该问题；若"新的x节点"是"黑+黑"，则需要对"新的x节点"进行进一步处理。

2.3 示意图



3. (Case 3)x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的

3.1 现象说明

x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的。

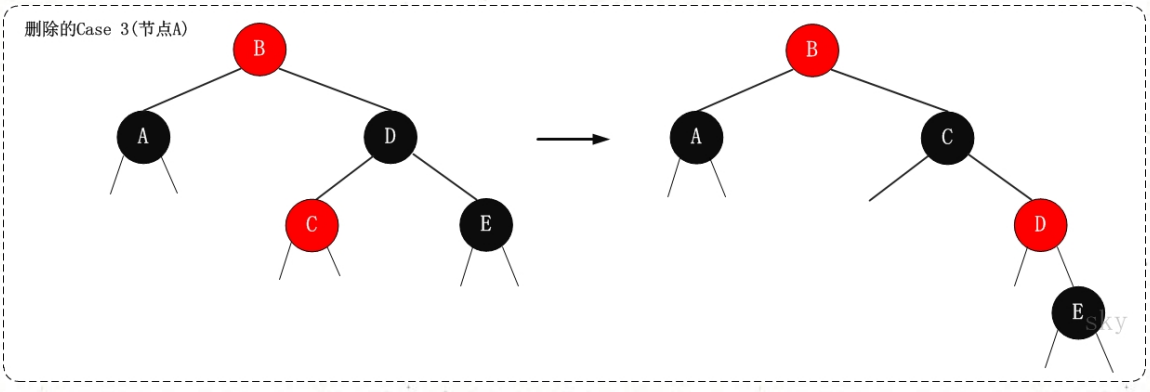
3.2 处理策略

(01) 将x兄弟节点的左孩子设为"黑色"。
(02) 将x兄弟节点设为"红色"。
(03) 对x的兄弟节点进行右旋。
(04) 右旋后，重新设置x的兄弟节点。

下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

我们处理"Case 3"的目的是为了将"Case 3"进行转换，转换成"Case 4",从而进行进一步的处理。转换的方式是对x的兄弟节点进行右旋；为了保证右旋后，它仍然是红黑树，就需要在右旋前"将x的兄弟节点的左孩子设为黑色"，同时"将x的兄弟节点设为红色"；右旋后，由于x的兄弟节点发生了变化，需要更新x的兄弟节点，从而进行后续处理。

3.3 示意图



4. (Case 4)x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的，x的兄弟节点的左孩子任意颜色

4.1 现象说明

x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的，x的兄弟节点的左孩子任意颜色。

4.2 处理策略

(01) 将x父节点颜色 赋值给 x的兄弟节点。
(02) 将x父节点设为"黑色"。
(03) 将x兄弟节点的右子节设为"黑色"。
(04) 对x的父节点进行左旋。
(05) 设置"x"为"根节点"。

下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

我们处理"Case 4"的目的是：去掉x中额外的黑色，将x变成单独的黑色。处理的方式是"：进行颜色修改，然后对x的父节点进行左旋。下面，我们来分析是如何实现的。

为了便于说明，我们设置"当前节点"为S(Original Son)，"兄弟节点"为B(Brother)，"兄弟节点的左孩子"为

BLS(Brother's Left Son)，“兄弟节点的右孩子”为BRS(Brother's Right Son)，“父节点”为F(Father)。

我们要对F进行左旋。但在左旋前，我们需要调换F和B的颜色，并设置BRS为黑色。为什么需要这里处理呢？因为左旋后，F和BLS是父子关系，而我们已知BL是红色，如果F是红色，则违背了“特性(4)”；为了解决这一问题，我们将“F设置为黑色”。 但是，F设置为黑色之后，为了保证满足“特性(5)”，即为了保证左旋之后：

第一，“同时经过根节点和S的分支的黑色节点个数不变”。

若满足“第一”，只需要S丢弃它多余的颜色即可。因为S的颜色是“黑+黑”，而左旋后“同时经过根节点和S的分支的黑色节点个数”增加了1；现在，只需将S由“黑+黑”变成单独的“黑”节点，即可满足“第一”。

第二，“同时经过根节点和BLS的分支的黑色节点数不变”。

若满足“第二”，只需要将“F的原始颜色”赋值给B即可。之前，我们已经将“F设置为黑色”(即，将B的颜色"黑色"，赋值给了F)。至此，我们算是调换了F和B的颜色。

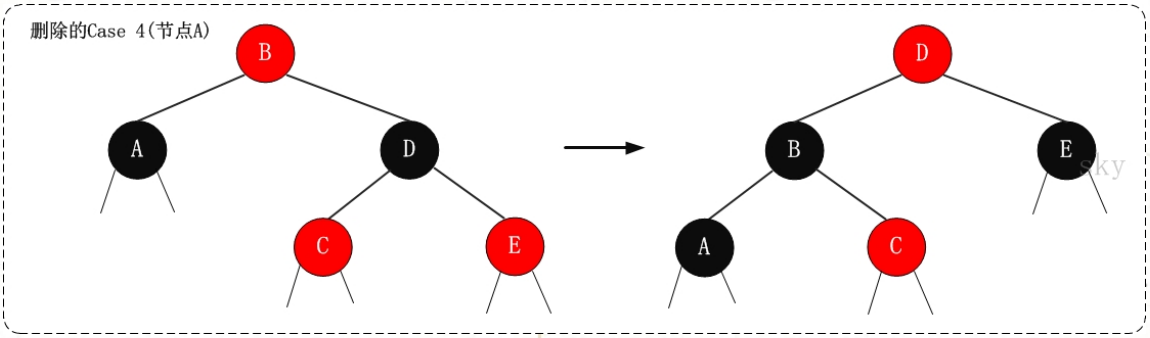
第三，“同时经过根节点和BRS的分支的黑色节点数不变”。

在“第二”已经满足的情况下，若要满足“第三”，只需要将BRS设置为“黑色”即可。

经过，上面的处理之后。红黑树的特性全部得到的满足！接着，我们将x设为根节点，就可以跳出while循环(参考伪代码)；即完成了全部处理。

至此，我们就完成了Case 4的处理。理解Case 4的核心，是了解如何“去掉当前节点额外的黑色”。

4.3 示意图



OK！至此，红黑树的理论知识差不多讲完了。后续再更新红黑树的实现代码！