# 红黑树的参考资料

**参考资料：**

详细分析看了都说好

<https://www.cnblogs.com/nullllun/p/8214599.html#autoid-3-2-0>

最容易懂得红黑树

<https://blog.csdn.net/Sun_TTTT/article/details/65445754>

源代码

<https://blog.csdn.net/eson_15/article/details/51144079>

极客时间

<https://time.geekbang.org/column/article/68976>

站在巨人的肩膀上，十分感谢以上参考资料；

# 红黑树的添加节点

<https://blog.csdn.net/eson_15/article/details/51144079>

讲的非常好；

添加节点跟二叉查找树添加节点的过程大体类似；

## 1.1 添加节点过程（二叉查找树）

我们这里把二叉树查找树的添加过程，搬过来了；

1. 从根节点出发，查找待添加节点A的位置；
2. 如果A大于当前节点（记为Parent），则继续往其右孩子分支找，否则，往其左分支找；

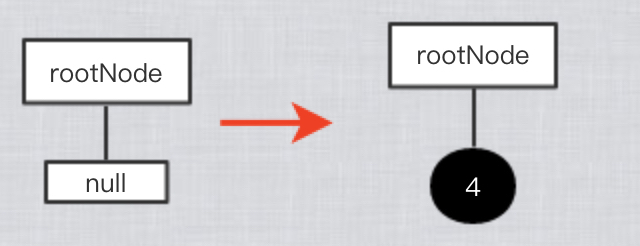
直到Parent的左或右孩子为NULL；

1. 当Parent值小于待添加节点的值，即：将A设为parent的右孩子，否则左孩子；
2. Parent为null，时，A设为根节点；

文字描述有点不好理解，我们画个图：

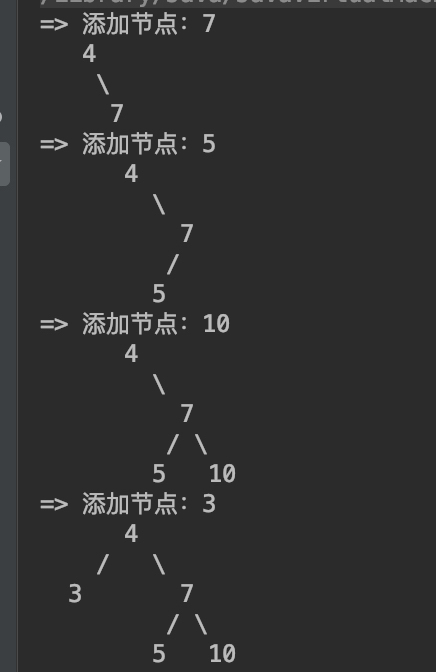
**树为空，添加节点时：**

此情况比较好理解，我们用一个rootNode根节点，初始化指向null，添加第一个节点时，便用rootNode指向她；

****

**树不为null时：**

我们就在上面的基础上，进行添加吧，比如我们添加节点(7, 5, 10, 3)；过程就不描述了，打印如下：



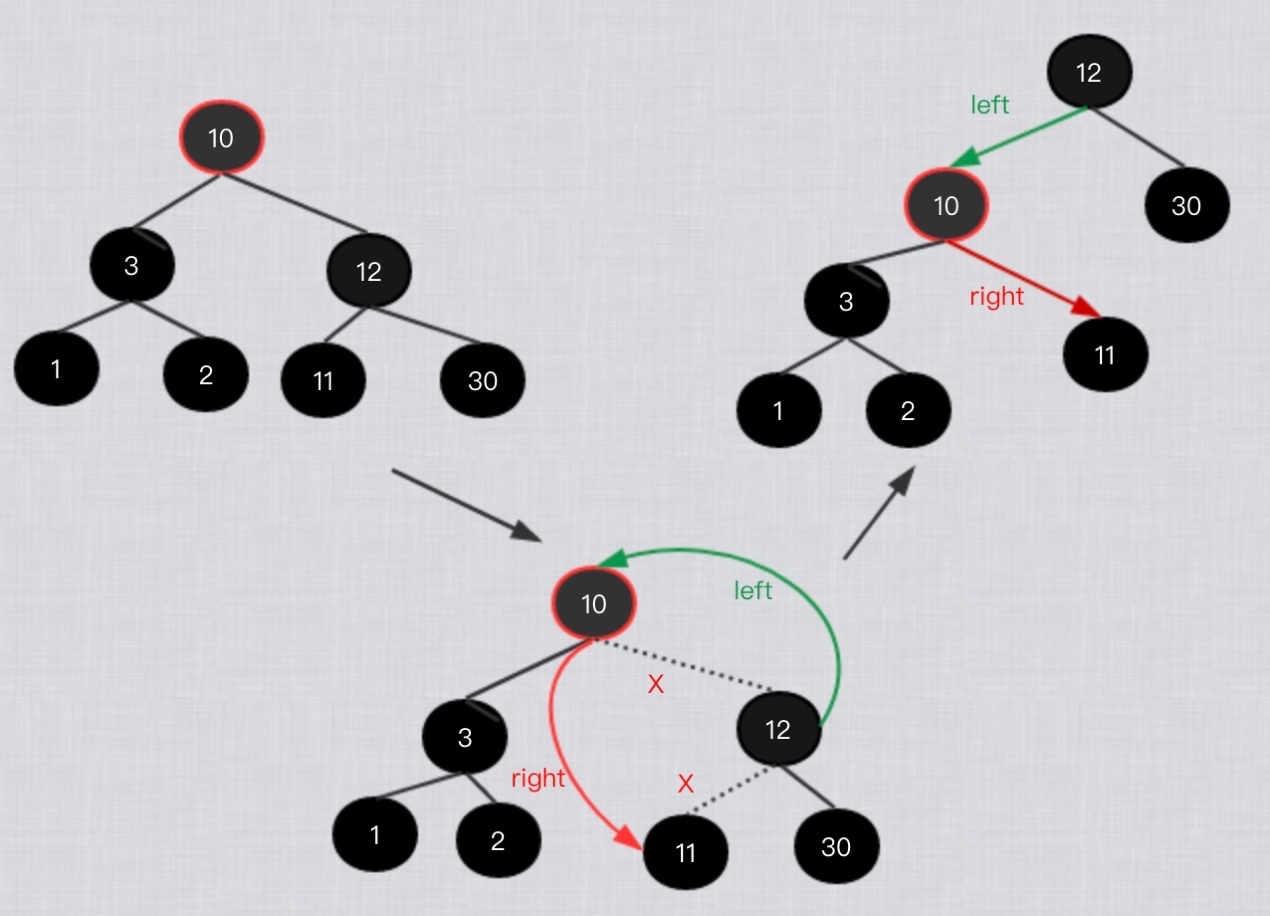
## 1.2左旋与右旋

调整之前，我们先了解一下左旋与右旋这2个神奇的操作；

**左旋：**

如下图，我们左旋节点10的过程如下：

1. 将旋转节点10变成了其右孩子12的左孩子；
2. 将右孩子12的左孩子11变成了旋转节点10的右孩子；

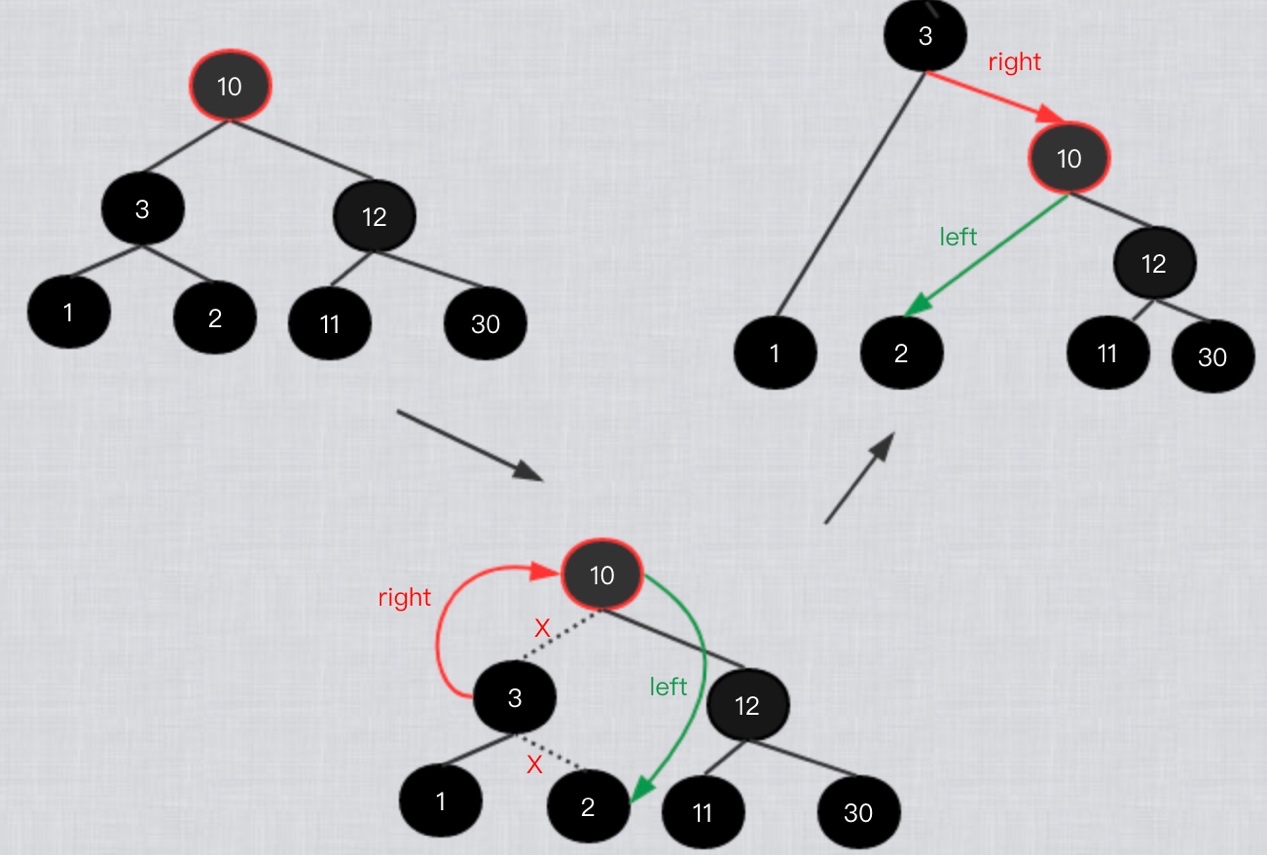


左旋时，我们要关注的是旋转节点右孩子，而不是其左孩子；

**右旋：**

如下图，仍然以10来进行，我们右旋节点10的过程如下：

1. 将旋转节点10变成了其左孩子3的右孩子；
2. 将左孩子3的右孩子2变成了旋转节点10的左孩子；

****

右旋时，我们要关注的是旋转节点的左孩子，而不是其右孩子；

**左右旋总结：**

1. 神奇的操作，经旋转后，依然保留了二叉查找树的特性（左小，右大）；
2. 从上图看，旋转只是影响其旋转节点的左或右；

## 1.3 添加后调整（难）

红黑树规定，新添加的节点必须是红色，在红黑树中添加节点的可能会破坏红黑树的特性：

1. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点；
2. 任何相邻的节点都不能同时为红色，也就是红色节点被黑色节点隔开；

所以添加节点结束后需要对树调整；

**2种特殊情况的调整：**

1. 原树为空，添加的节点为根节点，把节点变黑即可；
2. 添加节点的父节点是黑，此时添加已经结束；

**其他情况（3种），会违反红黑树的特性：**

说明：我们将刚添加进来的节点称为关注节点node，关注节点node会随着调整而发生变化，关注节点的父节点parent的兄弟节点，我们称为叔叔节点uncle；

1. 关注节点node，其叔叔节点是红色；

2. 关注节点node，其叔叔节点uncle是黑，且node是其父节点的右子节点；

3. 关注节点node，其叔叔节点uncle是黑，且node是其父节点的左子节点；

进入下面步骤之前，我们必须明确的是：

关注节点的父 一定是红色（黑色是特殊情况，上面已经处理好）；

### 1.3.1 Case 1: uncle红

关注节点node，其叔叔节点是红色；

此情况必然存在祖父节点，且祖父节点一定是黑色；

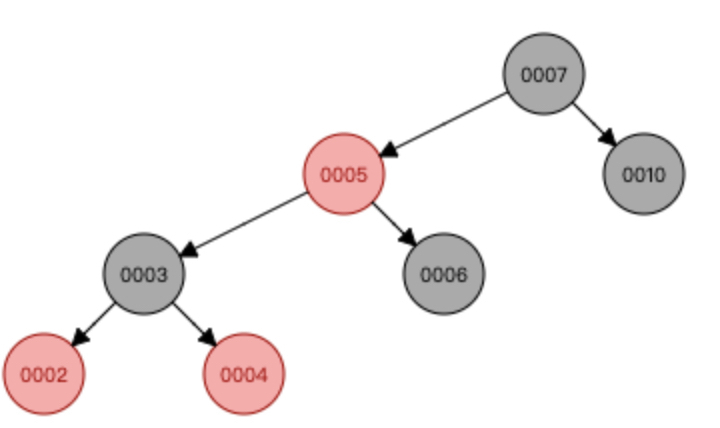
操作步骤：

1. 将关注节点node的父节点parent，叔叔节点uncle都染黑；
2. 将关注节点node的祖父节点gParent染红；
3. 将关注节点node指向祖父节点gParent；
4. gParent染红了，很有可能与其父形成连续的红色节点，此时继续走case1；
5. 跳到从case 2 或 case 3；

**模拟数据进入case3：**

我们先构建一个满足上述条件的红黑树；

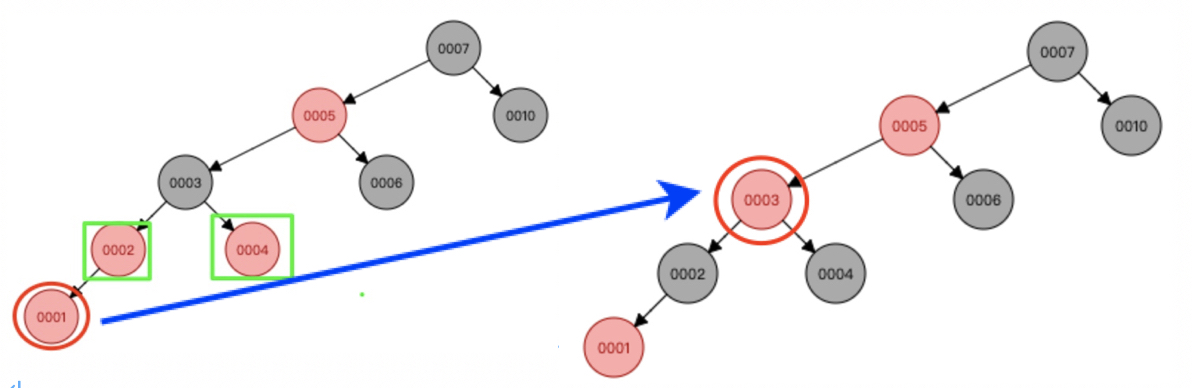
先后插入以下数据：[10,7,6,5,4,3,2]，构建出来的树如下：



我们新添加的节点1：

此时关注节点1与且其parent与uncle 都是红色；

我们变换调整如下：

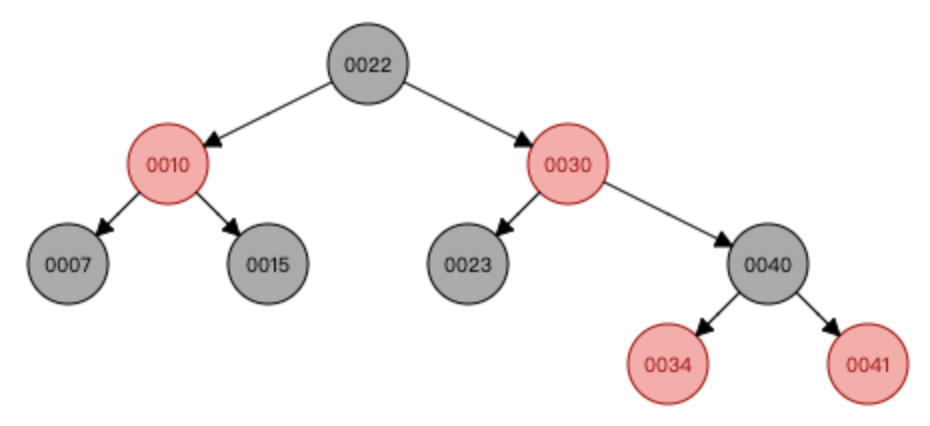


此时关注节点node是3了，且其uncle是黑色，且关注节点是父的左孩子；

那么进入到**case3**；

**模拟数据重复case1：**

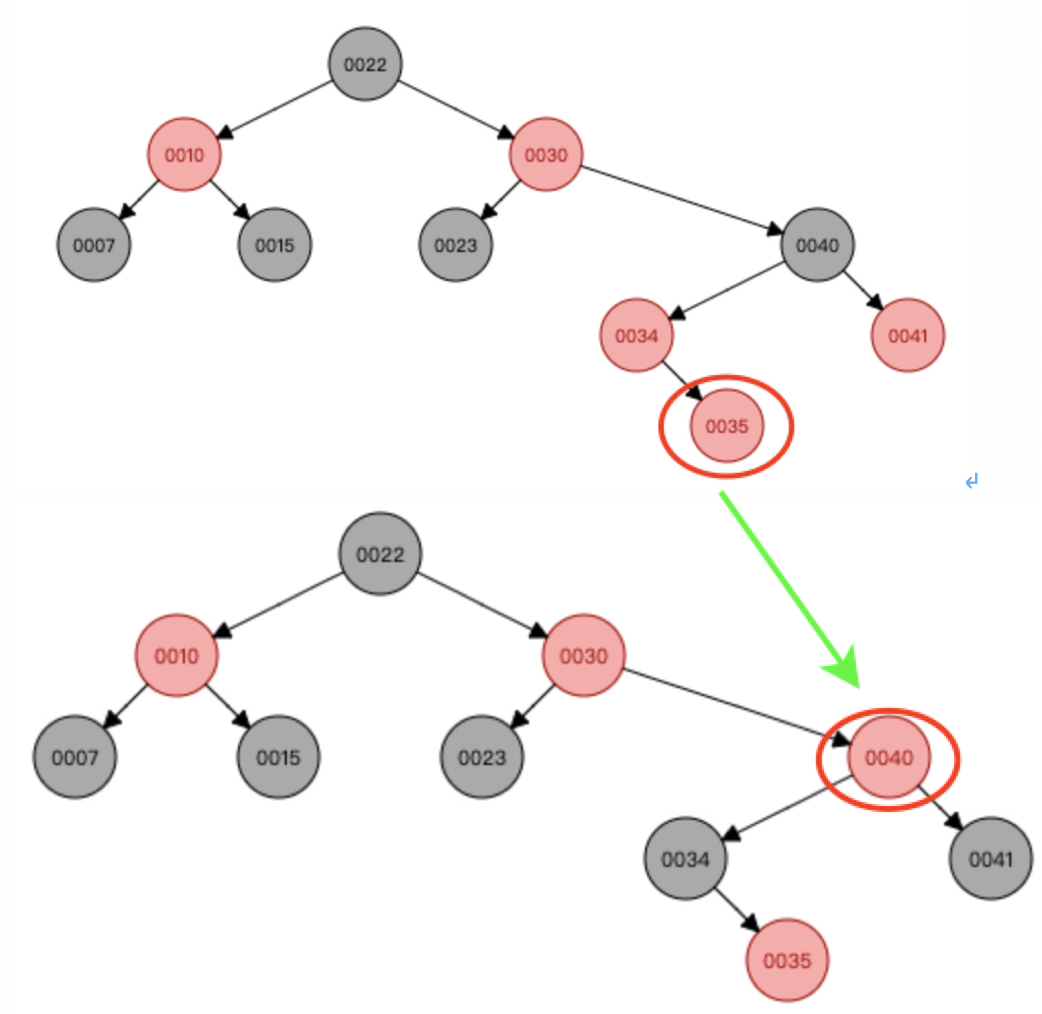
我们再次构建一个树，先后插入以下数据：[10, 7, 15, 30, 22, 23, 40, 34, 41]，构建出来的树如下：



我们新添加的节点35：

此时关注节点35与且其parent与uncle 都是红色；

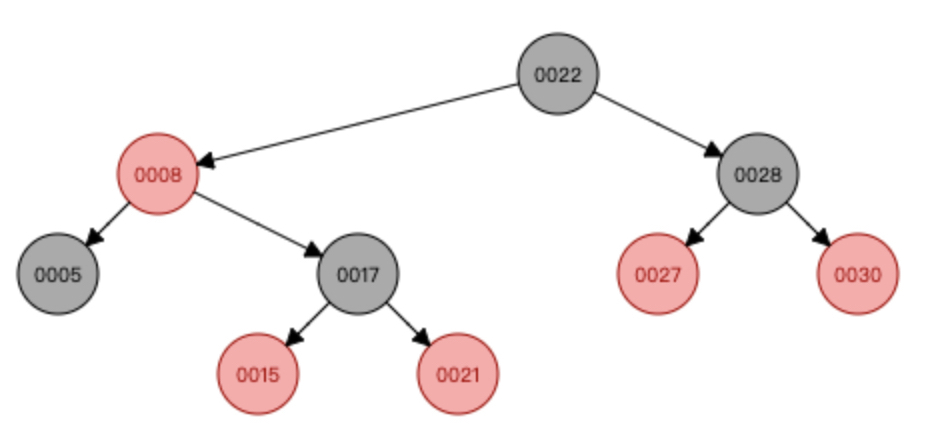
我们变换调整如下：



此时关注节点node是40了，且其uncle 红色，case 1要重新走一次；

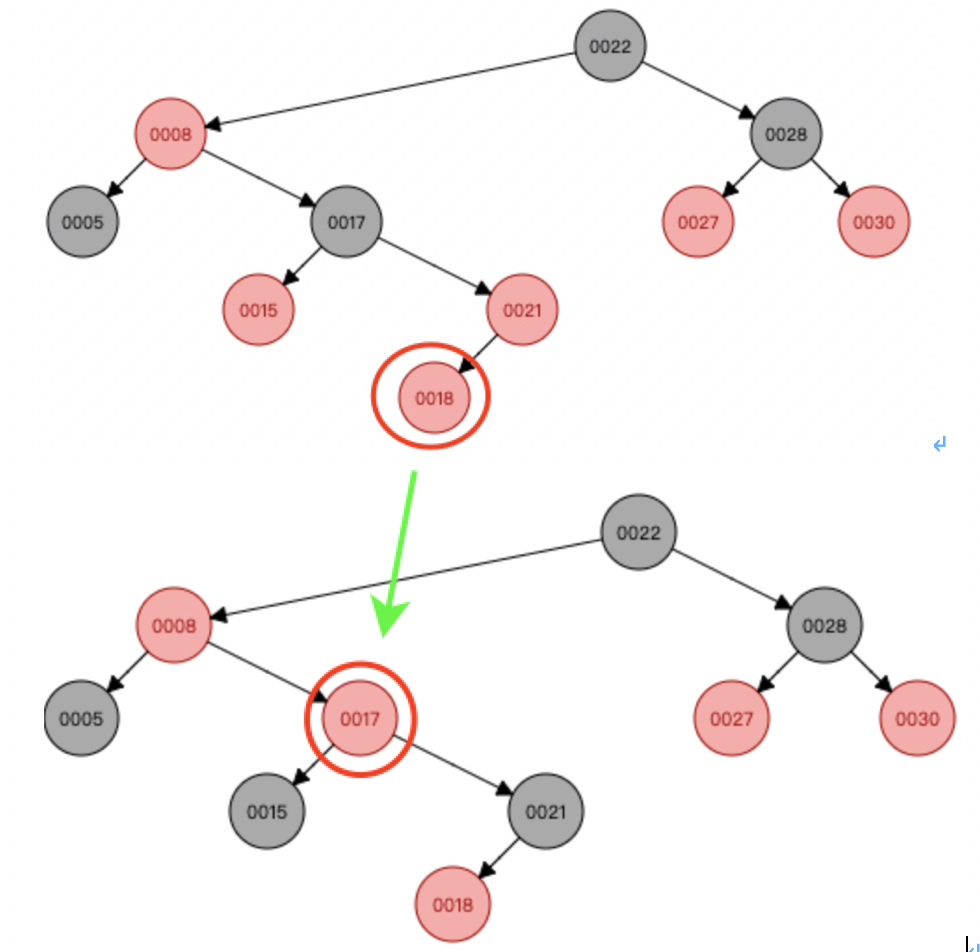
**模拟数据进入case2：**

我们构建一个树，先后插入以下数据：[]，此时树结构如下：



我们新增加节点18，此时关注节点18与且其parent与uncle 都是红色；

我们变换调整如下：



此时关注节点node是17了，且其uncle是黑色，且关注节点是父的右孩子；

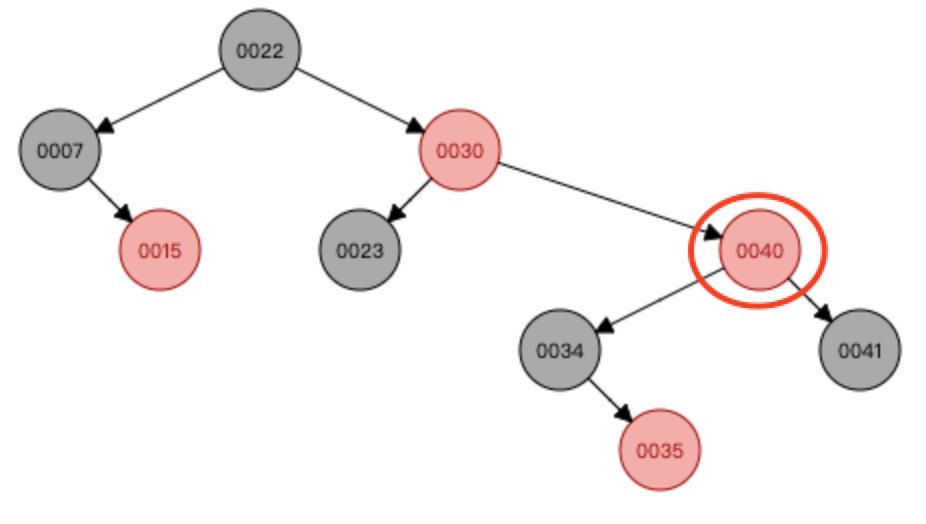
那么进入到**case2**；

### 1.3.2 Case 2: uncle 黑且A为父的右孩子

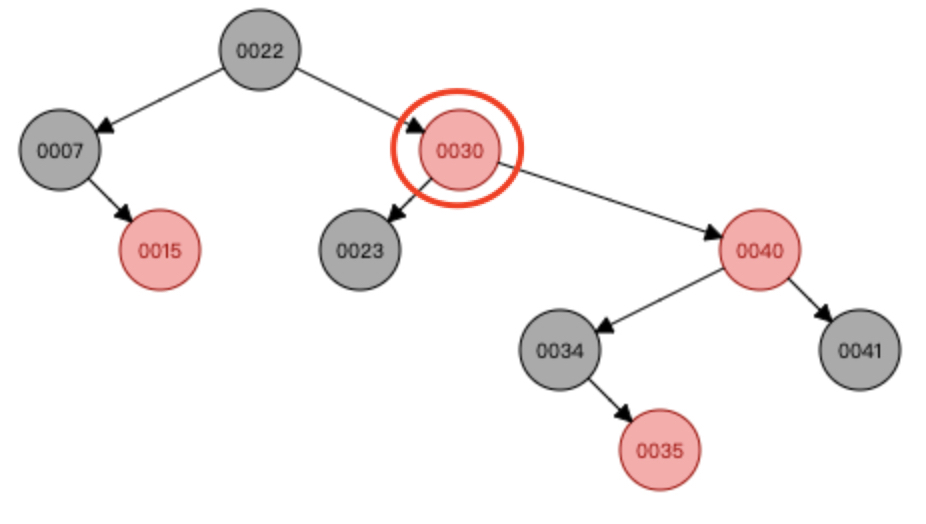
关注节点node，其叔叔节点示黑色，并且关注节点node是其父的右孩子；

操作步骤：

1. 将关注节点变成其父；
2. 围绕新的关注节点左旋；
3. 跳到case 3；



第一步，将关注节点变成其父30：



第二步，在新的关注节点处，左旋操作，如下：

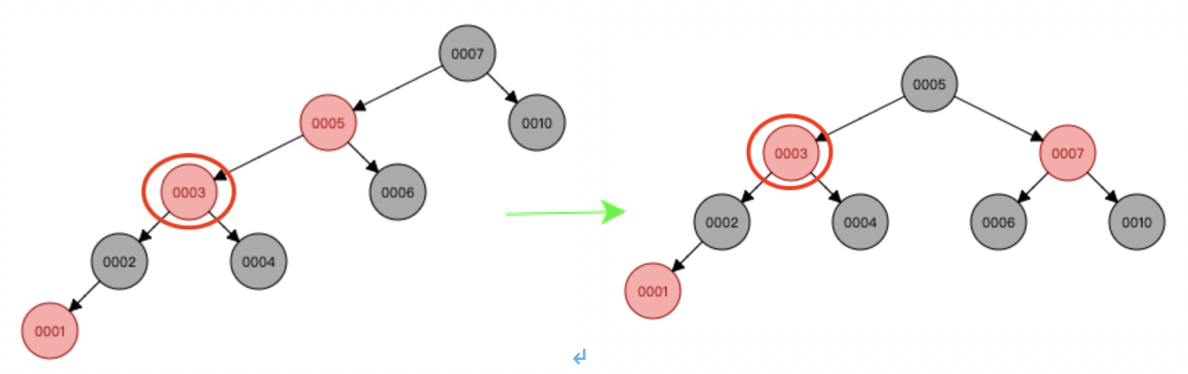
### 1.3.3 Case 3: uncle 黑且A为父的右孩子

关注节点node，其叔叔节点黑色，并且关注节点node是其父的左孩子；

操作步骤：

1. 将关注节点的父染黑，并将祖父节点染红
2. 围绕关注节点node的祖父节点右旋；
3. 根节点染黑；
4. 调整结束；

如下图示：



# 2. 红黑树的删除

红黑树的删除相对于插入节点，显得格外复杂，判断分支过多，刚接触会显得十分头晕，很容易让人放弃；

以下整理来着网络以上资料，结合自己的一些心得，希望对大家有帮助；

## 2.1 待删节点3个状态

合三为一，简化操作；简化成：

**待删节点至多只有一个孩子（左右无关）；**

### 2.1.1 待删的节点A有2个非空子节点

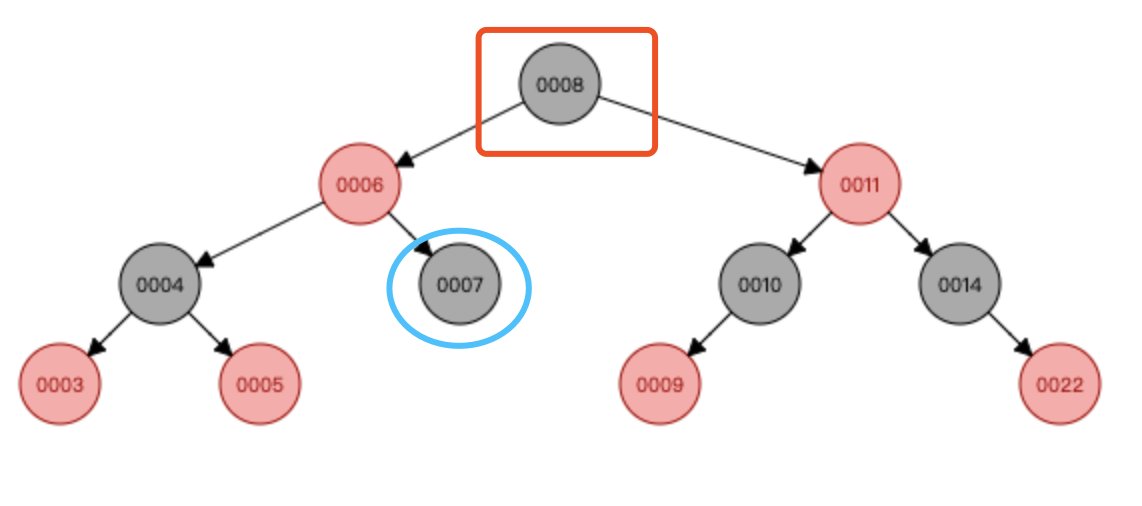
这种情况，我们不要直接删除节点A，而是通过以下操作：

1. 查找节点A前驱节点B（节点A左子树最大的节点，后继节点亦可）；
2. 将A节点的值替换成节点B的值；
3. 因为前驱节点B至多只有一个孩子节点，此时我们面对的如何删除节点B了；

（当然也可能不存在孩子）；

1. 现在待删节点变成了B，而B要么没有孩子，要么只有一个孩子；

如图：删除的是8，那么他的前驱是7；我们面对的是如何删除7这个节点；



通过前驱节点，删除变成了：待删节点（也就是前驱节点）至多只有一个子孩子；

（图示：7节点没有孩子，那么可以模拟一个黑色的null节点）

注意：此时还没有进行删除；

**但是如果要来硬的，直接删呢？（可略过）**

(注意：有点复杂，图是之前画的，使用的后继节点，懒癌上身，就不转换了)

图示说明：

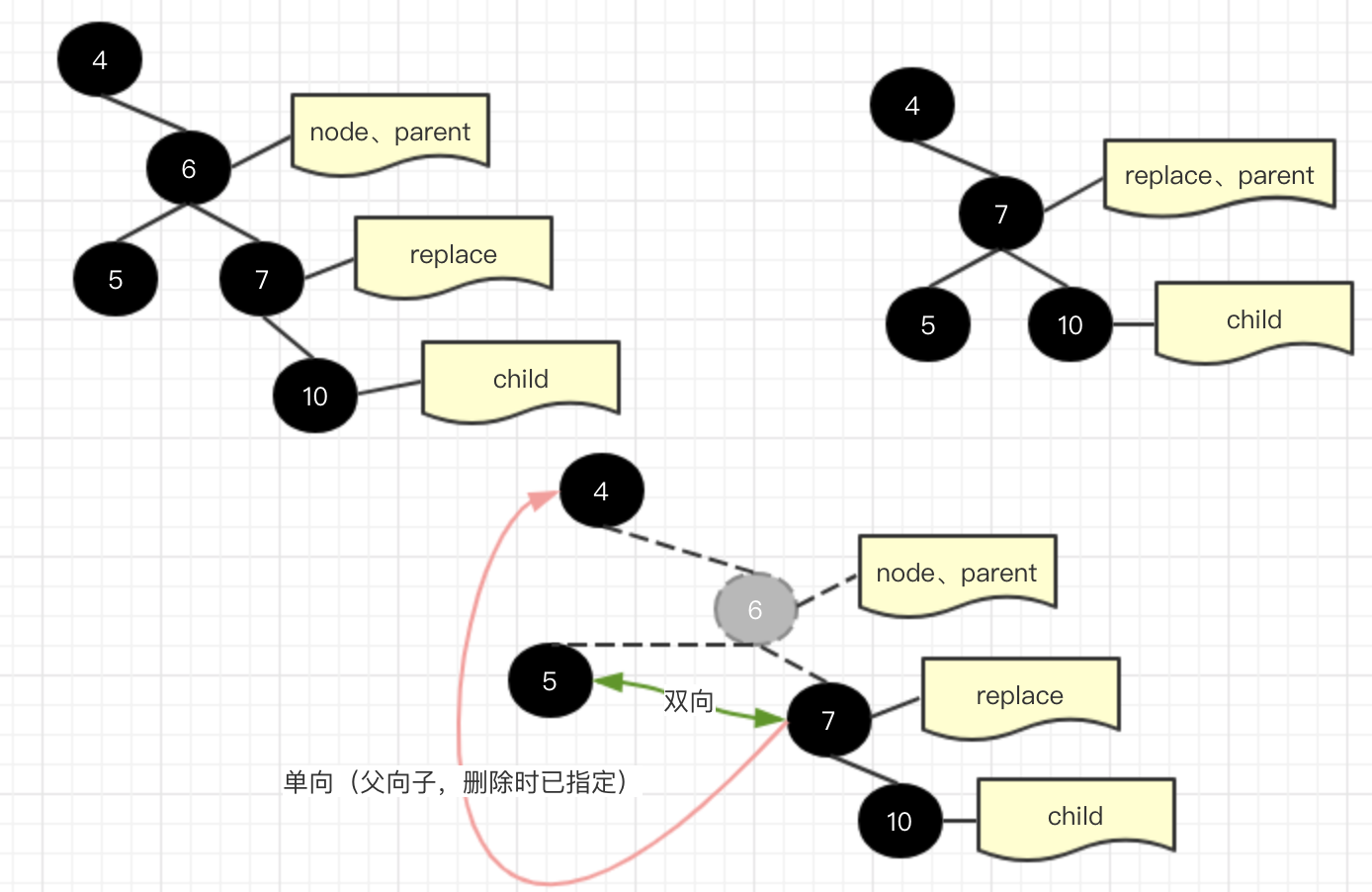
|  |  |  |
| --- | --- | --- |
| **节点** | **说明** | **备注** |
| node | 待删除节点 |  |
| replace | 后继节点 |  |
| parent | 后继节点的parent |  |
| child | 后继节点的孩子 | 只有一个孩子 |

**分为2个情况：**

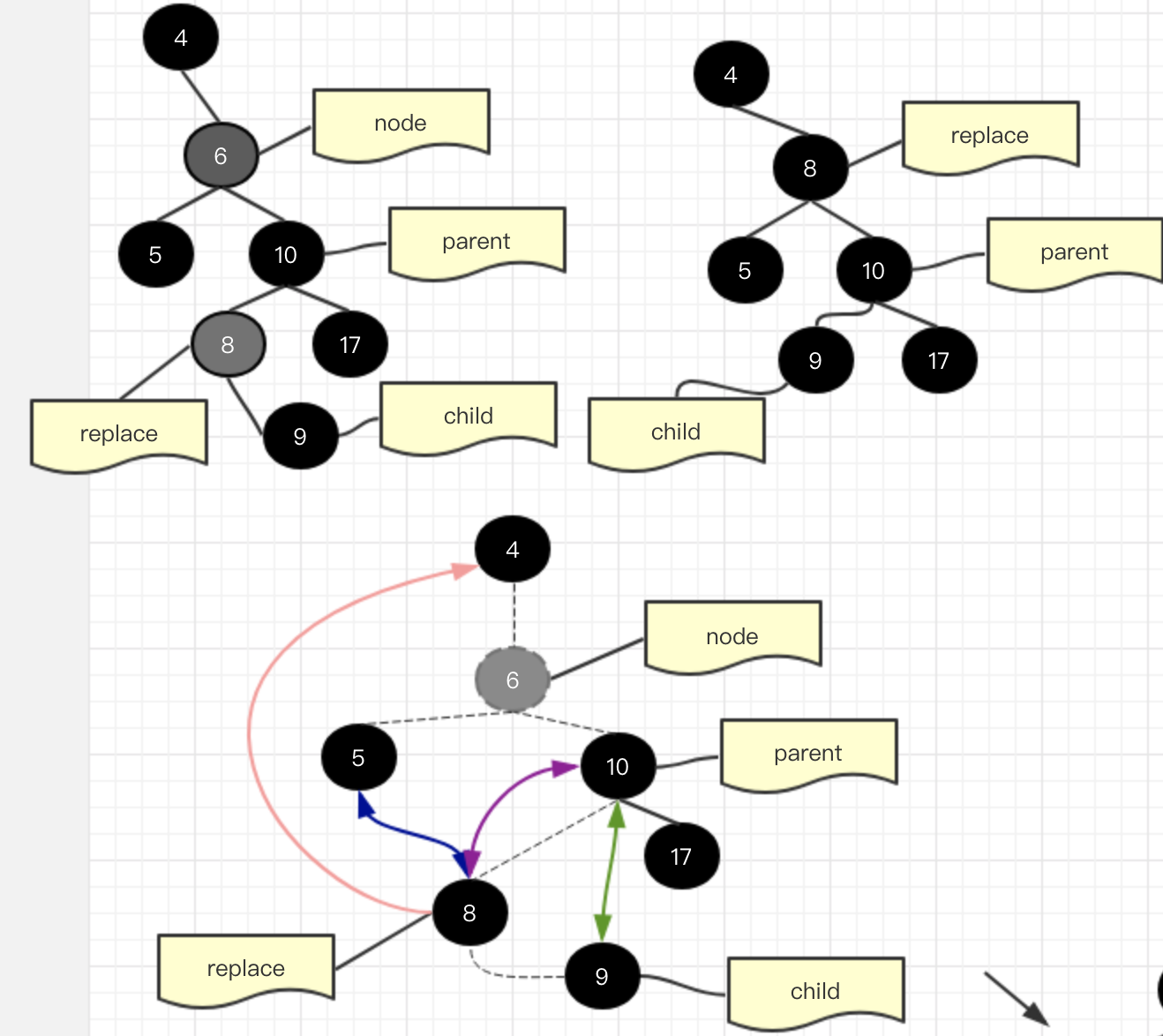
1. 后继节点是待删节点的直接子孩子：

如下，假设要删除的是6，他的后继节点是7，7的孩子是child；（7必然没有左孩子）；

删除过程如下图示；



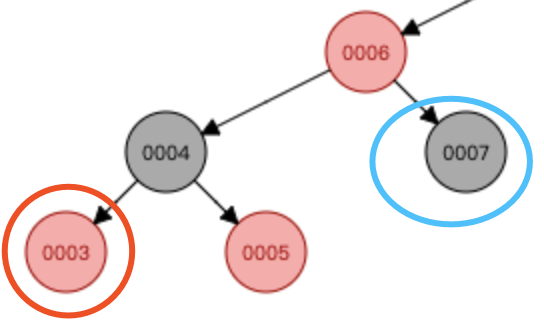
1. 如果后继节点不是待删节点的直接子孩子，如下图示：



彩色箭头为需要注意的地方，注意有些箭头是双向的，我已经晕了；

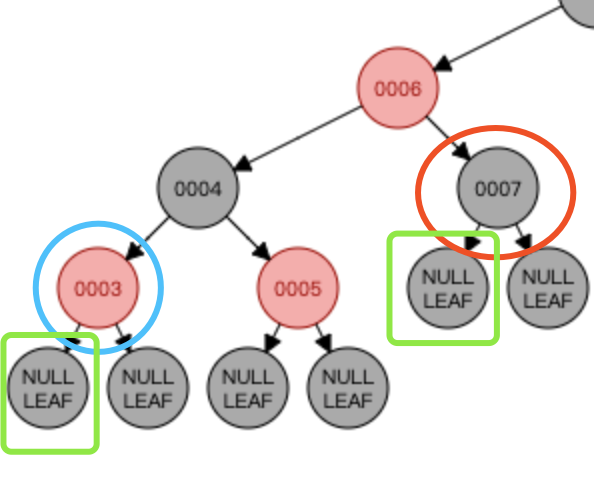
### 2.1.2 待删除的节点A没有子节点

此时待删除节点A颜色可能是红或黑；如下图：



如上图：红色节点3，黑色节点7，且他们都没有子节点；

此时，我们想象待删除的节点3或者7有一个虚拟的孩子节点B，且B是黑色的（B是左或右无关紧要）；



**通过变换：待删除的节点A也**有一个子节点B了，如上图绿色框；

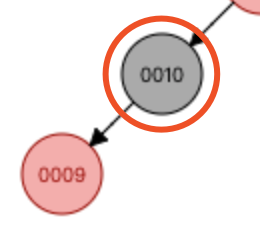
注意：此时还没有进行删除；

### 2.1.3 待删节点A只有一个子节点B

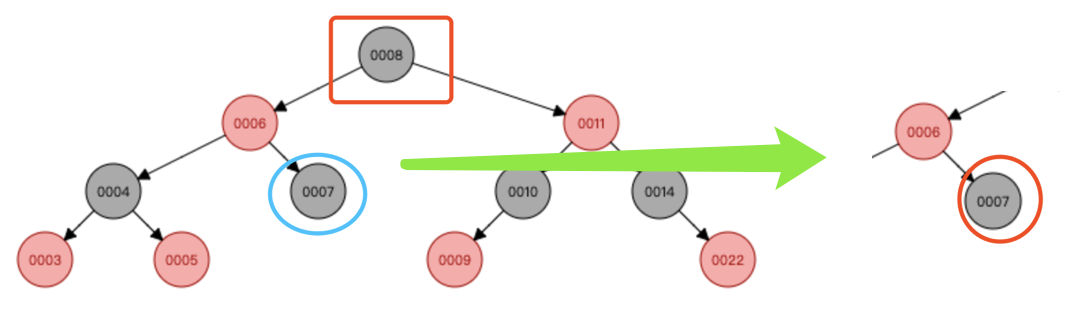
到现在所有的情况都变成了**待删节点的A，只有一个子节点B**（B有可能是虚拟节点）

汇总如下：

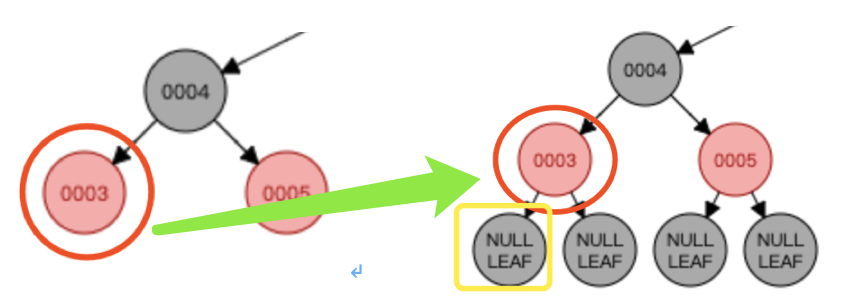
a. 待删节点只有一个节点图示如下：



b. 待删节点有2个非空子节点，需要转换一下，转换过程如下:



c. 待删节点2个空节点(虚拟节点为黑色)，转换后如下：



## 2.2 删除待删节点

有了上面的准备后，我们就可以开始删除了，删除与二叉查找树，大部分逻辑保持一致；

不一样的地方是：

1. 二叉查找树的删除，需要记录一下parent节点，但在红黑树中，节点自带parent；
2. 在红黑树中，我们也会声明一个parent节点变量，用来记录待删节点的parent；

（是否多此一举？）；

1. 在红黑树中节点中，因为多了parent成员，删除时，需要注意parent节点的衔接（类似双端链表删除节点）；

### 待删节点的转变过程

我们直接把二叉查找树，查找前驱节点那部分代码拿过来，稍微修改一下；

注意的地方是待删节点与其parent可能会发生变化（有个2个孩子时）；

示例代码：

private fun deleteNode(node: RBNode<T>) {  
var toRemove: RBNode<T> = node   // 待删除节点，待删节点有2个孩子时，   
var child: RBNode<T>?            // 待删除节点的子节点（最多一个）  
var parent = toRemove.parent    // 待删节点parent(2个孩子时，parent会改变)  
​  
 // === 搬来2叉查找树的删除过程；情况2，删除后，变成情况1或者情况3  
 if (node.left == null && node.right == null) {  
     // 1.left与right都为null  
     child = null  
 } else if (node.left != null && node.right != null) {  
     // 2.left与right都存在的情况，分为3个小步骤来操作  
     // 2-a. 获取前驱节点 replace  
     var replace = node.left  
     while (replace?.right != null) {  
       replace = replace.right  
    }  
     // 2-b.偷梁换柱  
     node.data = replace?.data!!   // 直接改变值，完成替换，链接关系不变  
     child = replace.left          // 设置前驱节点值；这里不会存在右孩子  
     toRemove = replace        // 改变待删除节点；下面就变成了删除 前驱节点 了  
​  
     // 2-c.设置parent，待删节点变化了，所以待删节点parent也跟着变化  
     parent = replace.parent  
     if (parent == toRemove) {    // 前驱节点为待删节点的直接子孩子，parent变量  
        parent = replace // 注意：parent 改变了   
    }  
  } else {  
     // 3.只存在一个孩子节点时，待删节点必是黑色，child必是红色（特性）  
      child = if (node.left != null) node.left else node.right  
  }  
 ...

### 2.2.2 删除过程

对比二叉查找树的删除过程，红黑有一点不同的是child需要设置一下parent；

示例代码：

// ======= 删除操作： toRemove  
// 说明：child：待删节点的孩子 child；

// toRemove:待删除节点；  
​  
// 待删除是根节点  
if (toRemove.parent == null) {  
  // 删除是根节点，直接删除  
  rootNode = null  
  return  
}  
​  
// 非根节点时：先直接删除节点，并设置待删节点孩子的父（注意：此时孩子只有一个）  
if (toRemove.parent?.left == toRemove) {  
  toRemove.parent?.left = child  
} else {  
  toRemove.parent?.right = child  
}  
​  
// 如果孩子不能空，设置孩子的父  
if(child != null) {  
  child.parent = toRemove.parent  
}

删除操作完成，下面进入删除图示与最复杂的变换调整；

## 2.3 删除与变换调整

准备好，我们开始吧；

变换调整是为了满足红黑树的2条重要特性：

1. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点；
2. 任何相邻的节点都不能同时为红色，也就是红色节点被黑色节点隔开；

**注意： 这里的待删节点是通过转变后的待删节点，也就是至多只有一个孩子节点的情况**，千万不要在这里混淆了，否则就没得玩了，这里不啰嗦了；

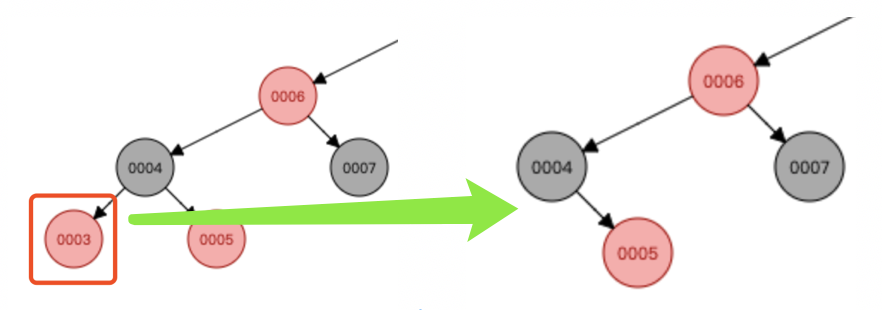
如果还晕，请重复上面的过程，直到明白了，再继续；

我们关注待删节点，待删节点分为红黑2个颜色状态；

### 2.3.1 待删节点是红色

待删节点是红色，那么直接拿该节点的孩子补空位即可（孩子必定是黑色，没有孩子时，我们想象其有一个虚拟的黑色节点）；

删除红色节点，不会违反红黑树特性；



待删节点是红色，到这里我们已经处理完毕；

如果是黑呢，继续往后走；

### 2.3.2 待删节点是黑色

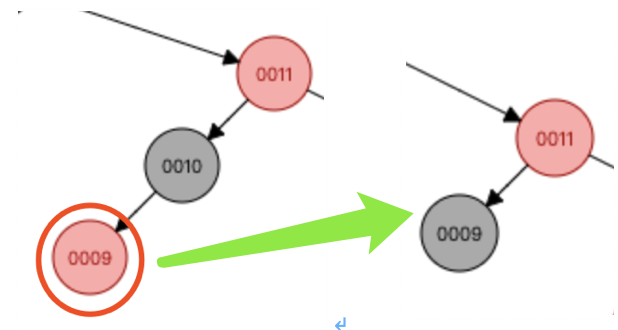
如果待删节点是黑色，删除后，破坏了（从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点），

此时我们需要对其孩子节点（记为child）进行判断（只有一个孩子）；

分为2个情况

红与黑；

1. 如果child 是红色，那么很好办，待删节点替换成child，并child颜色染成黑色，红黑树性质恢复，此时我们处理完毕；



示例代码：

/\* === Part Three: 删除后变换 \*/  
if(child != null) {  
  // 如果有child，那么toRemove必定是黑色，child必定是红色(红黑特性)，  
  // 将child直接染黑，调整结束；  
  child.isRed = false  
  return  // 后面的判断没必要了，所以加return；不加也没事  
}

1. 如果child是黑色（没有孩子时，想象黑色的虚拟节点）,我们进入最难的部分了；

继续往后走；

这里分为了好几大类，不再好细分了，所以下面单独二级标题，

## 2.4 待删节点黑色且child 是黑色（难）

到这里，待删节点已删除，我们关注的是child（child不存在时，用虚拟的黑色节点代替）；

还记得我们上面定义的parent节点么？parent的用处来了，用来拿到child的兄弟节点；

在这里我们将兄弟节点记为other节点（引用上述博客中的代码）；

此时我们需要根据兄弟节点（other）的状态来做一些变换；

我们将节点child改名成**关注节点node**；

共分为4个情况（parent颜色无需关注）：

1. 关注节点node的兄弟节点other是红色；
2. 关注节点node的兄弟节点other是黑色，且other的左右孩子都是黑色；
3. 关注节点node的兄弟节点other是黑色，且other左孩子为黑，右孩子为红；
4. 关注节点node的兄弟节点other是黑色，且other右孩子为红；

**注意：**进入以下分支之前，明确的是关注节点node（或虚拟节点）已经是黑色的；

此时千万不要纠结删除功能，删除我们已经做完了；

也不要纠结关注节点的子节点，因为不会以关注节点进行旋转操作，所以其孩子关系不会打乱；

我们关注的是节点node与其兄弟节点other（含子节点）；把什么前驱后继节点统统忘记吧；

我们现在要做的是**变换调整**；

### 2.4.1 Case1: other是红色

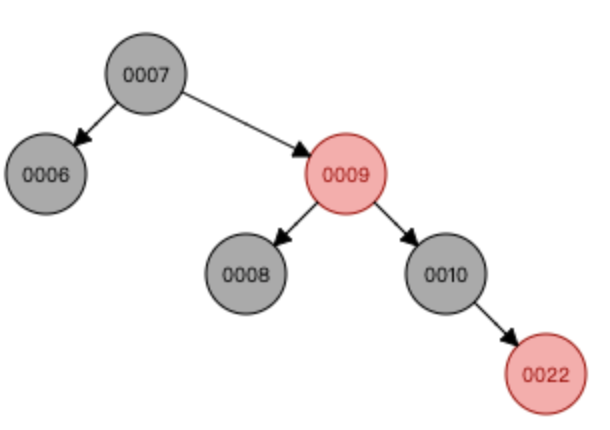
关注节点node的兄弟节点other 是红色，此时不用关注other的左右孩子；

操作步骤：

1. 将兄弟节点other染黑，父节点parent染红；
2. 以parent为中心左旋；
3. 关注节点不变，other节点改变；
4. 继续从4种情况中选择合适的规则来调整变换；

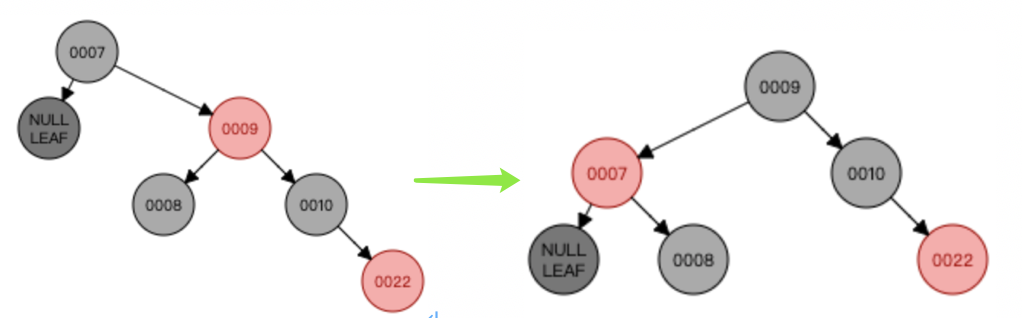
我们先构建一个满足上述条件的红黑树；

先后插入以下数据：[7,6,9,8,10,22]，构建出来的树如下：



为了方便理解，我们这里把删除功能表现出来了，即：删除节点6后，此时关注节点是虚拟的node，并且 node的兄弟节点other是红色：

变换调整如下图：



那么现在的情况是：

关注节点是黑色，且兄弟节点是黑色，且兄弟节点的左右孩子都是黑色（虚拟）；

也就是进入了下面的Case2；

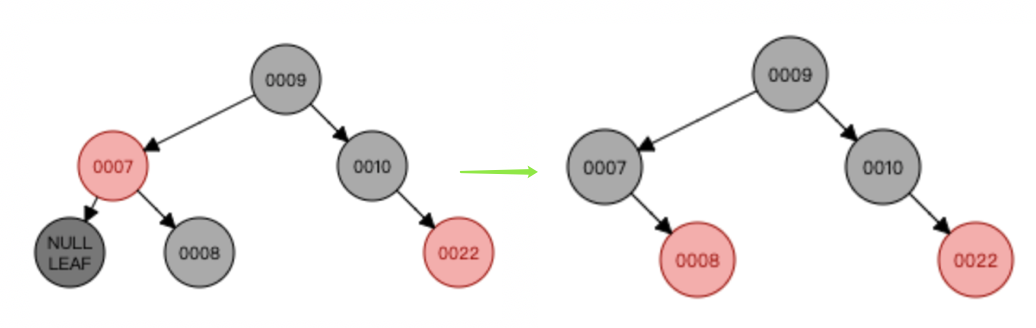
### Case2: other与2个子节点全黑

关注节点node的兄弟节点other 是黑色，且2个孩子都是黑色（没有则是2个黑色的虚拟节点）；

操作步骤：

1. 将关注节点的兄弟节点 other 染红；
2. 关注节点变成其parent，parent重新赋值；
3. 如果没完成，需要继续从4种情况中选择合适的规则来调整变换；

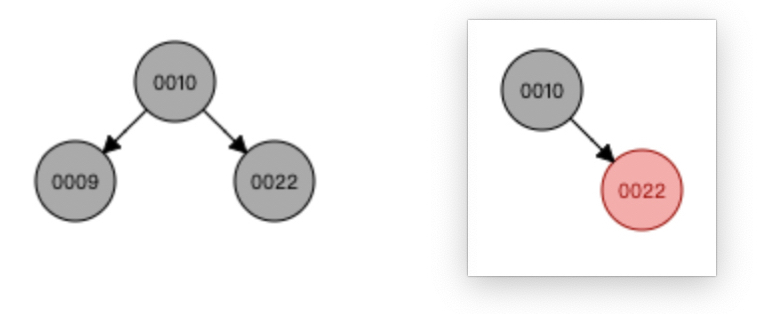
如下图图示：



上图中的parent为红，此时调整已经完成；

如果父为黑色，染红other时，经过other路径会少一个黑色；但此时父已经是根节点了

如下图示：



PS: 经过这里，其实应该已经结束调整了；不会继续走其他case了，先不纠结；

### Case3: other黑且其子左红右黑

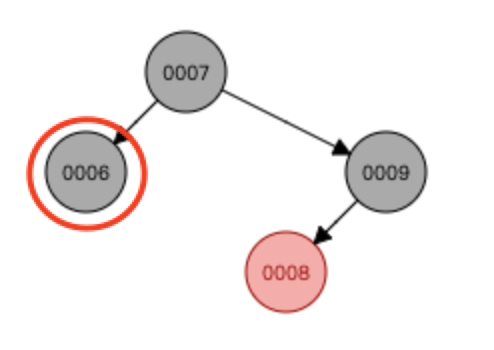
关注节点node的兄弟节点other 是黑色，且左孩子是红色，右孩子是黑色（含null）

操作规则：

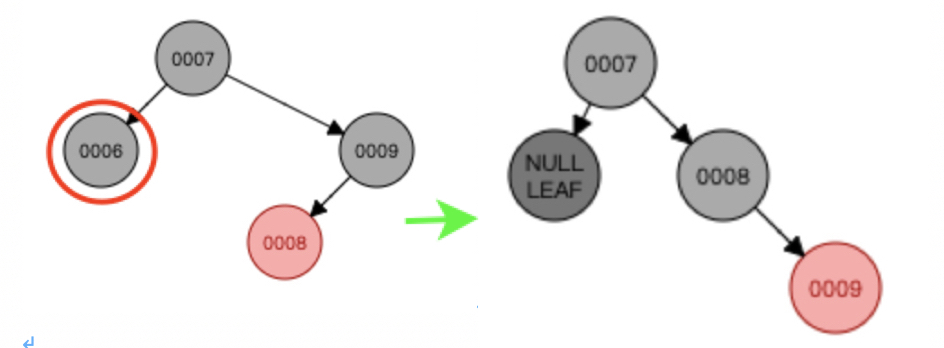
1. 将兄弟节点other的左孩子染黑，other染红（交换颜色）；
2. 围绕兄弟节点other右旋；
3. 关注节点不变，兄弟节点other重新赋值；
4. 调整后调到Case 4；

我们先构建一个满足上述条件的红黑树；

先后插入以下数据：[7,6,9,8]，构建出来的树如下：



为了方便理解，这里也把删除功能表现出来了，即：删除节点6后，此时关注节点是虚拟的node，并且 node的兄弟节点other是黑色，左孩子是红色，按规则处理如下图示：



接着进入Case 4；

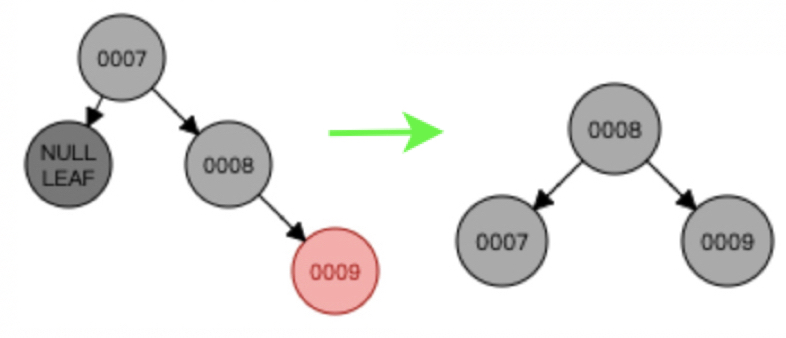
### 2.4.4 Case4: other黑且其子左黑右红

关注节点node的兄弟节点other 是黑色，且左孩子是黑色（含null），右孩子是红；

操作规则是：

1. 将 other 设为node父节点相同的颜色；
2. parent染黑，other右孩子染黑；
3. 围绕parent左旋；
4. 调整结束；

如下图图示：



## 删除节点总结

参考资料中，一些总结非常到位，这边我们也不再啰嗦了；我列出自己的一些想法：

1. 删除确实难，主要是因为关系没有理清楚，再进入调整时（2.4 时）我们必须明确关注节点现在已经是黑色的，千万不要搞错关注节点；这样2.4 就比较好理解了；
2. 我们千万不要去想象多层级时的关系。进入2.4时，请确保前面都已经理解；
3. 规则不要强行记忆，case 1与case2是成对出现的，同样case 3与case4也是成对出现的；
4. 可在 <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

进行数据模拟测试；