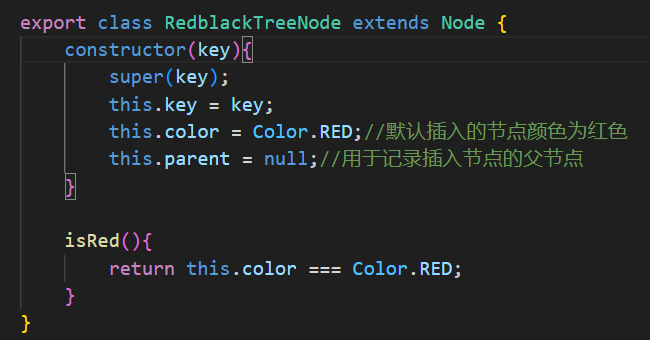
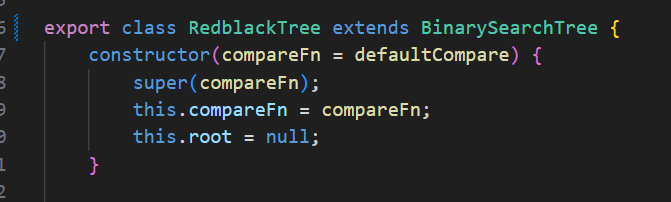
1. 构建红黑树的基础

构建红黑树，我们可以借用我们已经构建的好的二叉搜索树作为基类，之所以不用avl树的原因是:红黑树不一定是avl树。同时我们还需要单独构建红黑树的节点类，因为它的节点多了：

1. 记录节点颜色的属性
2. 记录节点父元素的指针属性
3. 一个用于检查节点是否为红色节点的方法。



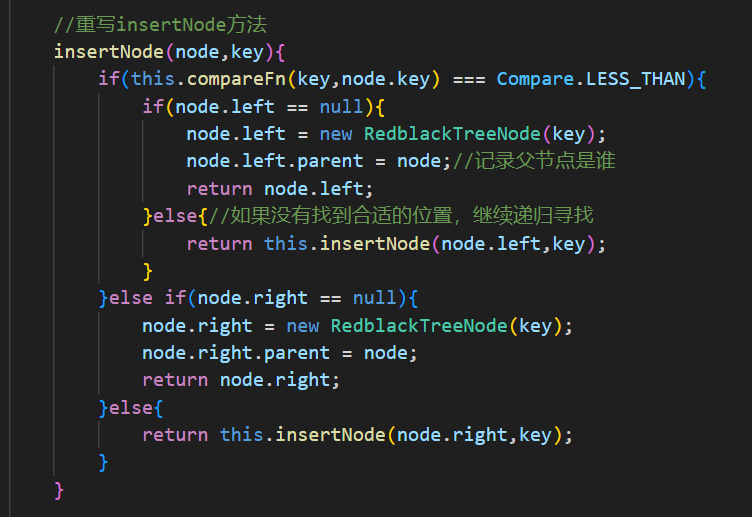
红黑树的基础类



1. 在红黑树中插入新节点

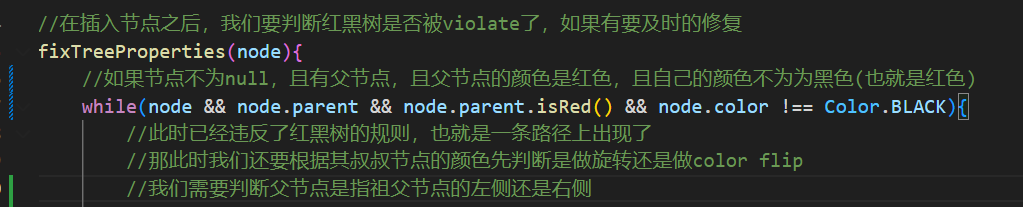


还是利用递归去找到合适的位置插入节点，这个插入节点的方法其实没什么讲究，就是按照二叉搜索树的规律去插入就行了。不过在插入节点后，我们需要检查这颗红黑树是否还合法。



1. 检查红黑树是否合法，然后根据对应的情况将红黑树变得合法。

在每次插入节点之后呢，我们需要将插入的新节点传到一个专门用于检测红黑树是否还合法的方法中。



这里提醒一点，在红黑树中有这个几个比较重要的节点

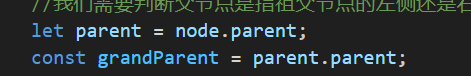
1. 父节点
2. 祖父节点
3. 叔叔节点

在插入新节点后，我们需要根据具体的情况来判断需要对这三个节点做怎么样的操作：

1. 颜色翻转
2. 旋转之后再做颜色翻转

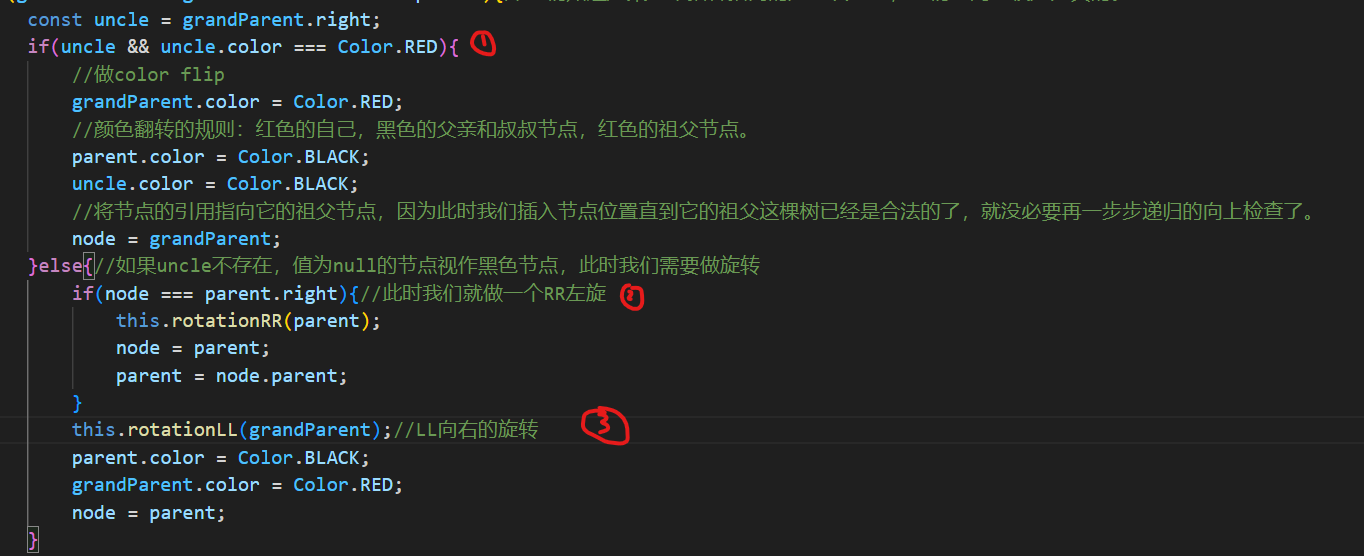
我们将这个fixTreeProperties()方法下的几种情况分解开来，一个个来分析查看，并结合简单的小例子来帮助理解：

首先不管做什么样的操作，我们都需要将插入的新节点的父亲节点和祖父节点用变量保存下来。



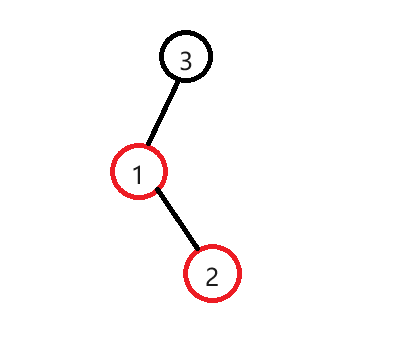
我们先来看第一种情况，就是如果新插入节点的祖父节点存在，且他的父节点是祖父节点的左侧节点。



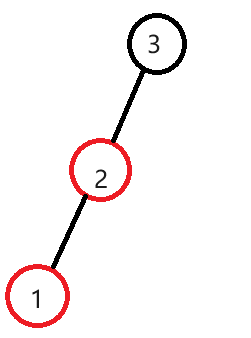


我们将它分为三种情况

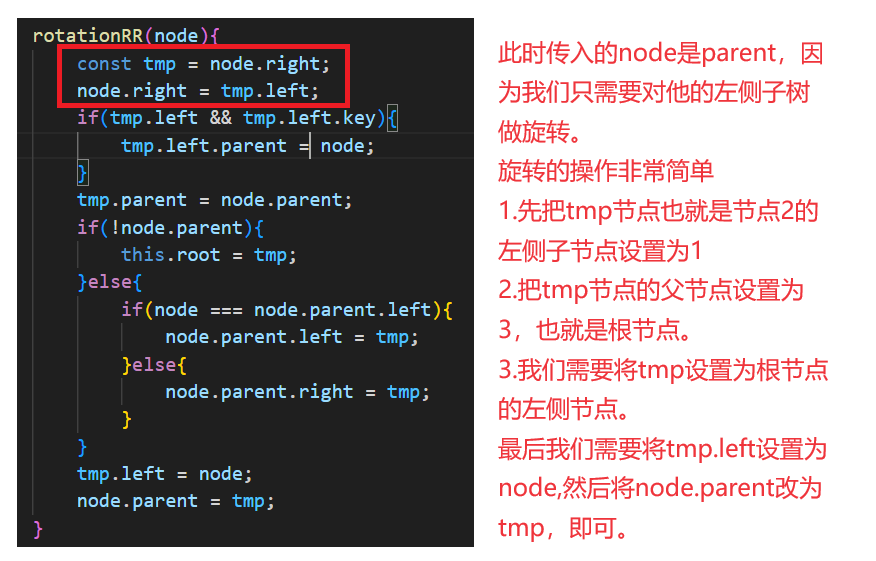
1. 叔叔节点的颜色是红色，此时我们需要做颜色翻转”color flip”此时我们就想起我们的规则”红色的爷爷节点，黑色的叔叔和父亲节点，红色的自己”。此时有可能你会说了，那假如爷爷节点恰好是红黑树的根节点呢？没关系，我们会在这个方法的最后，统一使用this.root.color = Color.BLACK来保证我们的根节点的颜色始终是黑色。
2. 如果uncle节点不存在，此时我们就需要做旋转了，那具体要做什么样的旋转呢：  
   a:如果插入的节点在父节点的右侧，我们需要做一个向左的旋转，我们来找一个简单的例子来看一下。



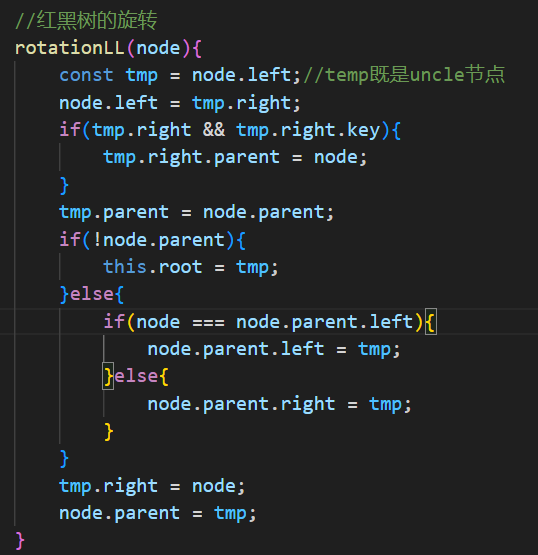
首先在这里重申一点，我现在讨论的是父节点是祖父节点的左侧节点，而新插入的节点是父节点的右侧节点，结合我们学过的avl树的知识，其实这课树呢还不算是不平衡。但是还是请你记住，但凡是树，它们的旋转操作都是一样的。不过由于这里我们学习的是红黑树这种特殊的树，我们需要考虑节点的颜色等问题，所以涉及到了稍微更加复杂一点的节点关系。不过没关系，只要我们把每一种情况都分析到位，最后再组合在一起就可以了。计算机最擅长做的事就是，帮助我们做大量重复的事情！



首先我们需要将树通过旋转变成上图所示的样子，此时我们需要对这棵树的子树做向左的旋转。

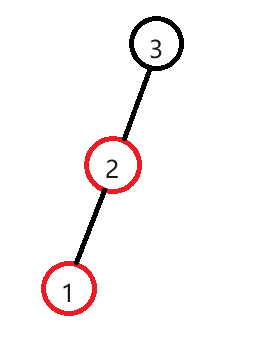


在旋转完之后，我们还需要对这个树做一个向右的旋转，但是此时传入rotationLL的节点就是grandparent节点了。



这里我就不带着一步步分析了，心细的人可以找一张纸慢慢画一画，只要细心一点就可以试出来。

接下来我们来分析一下插入的节点在父节点的左侧，且父节点是祖父节点的左侧节点的情况。我们还是先找一个最简单的例子来看一下：



对于这种情况我们知道需要对这颗树做向右的旋转操作，这个呢我们就直接将祖父节点传入交给rotationLL方法去做旋转操作即可。