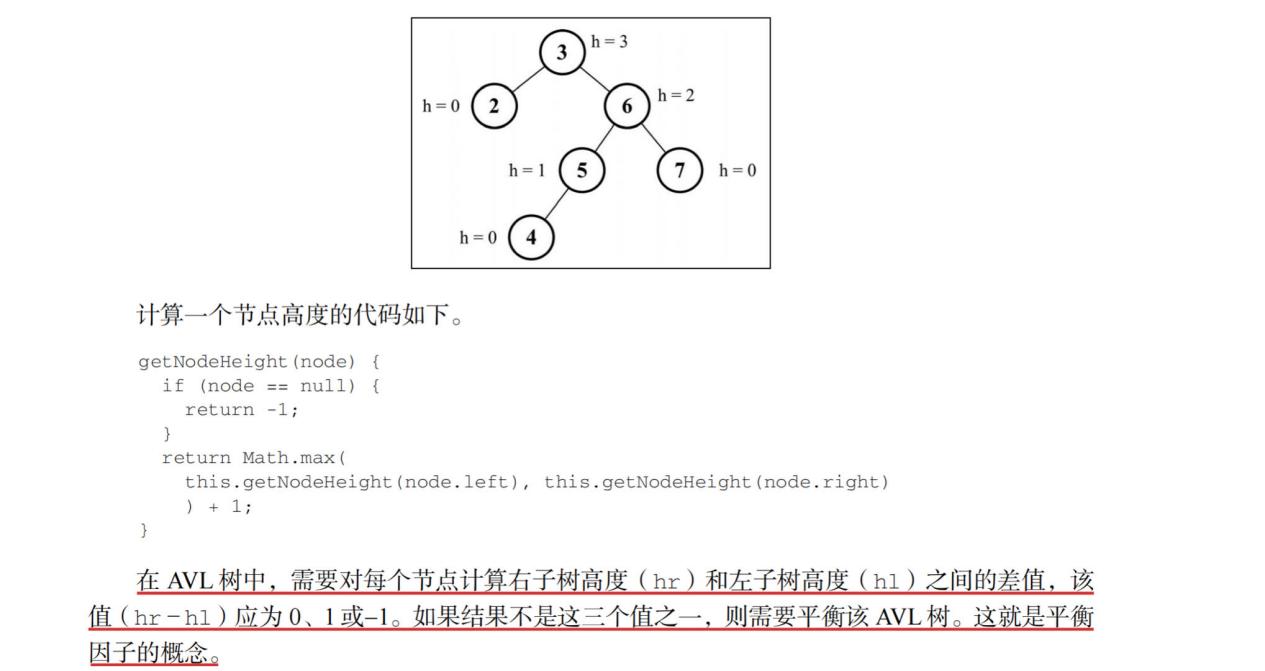
如何计算一个节点的高度？为什么要计算一个节点的高度？

同样是利用递归算法，如果这个节点为空那么它自然没有高度，之所以我返回-1是因为没有子节点的节点高度为0，这样是为了方便递归算法的实现。因为我们需要通过树的高度计算出平衡因子，通过平衡因子判断树是否需要旋转。

你能解释下图计算平衡因子的算法，为什么要这样设计吗？

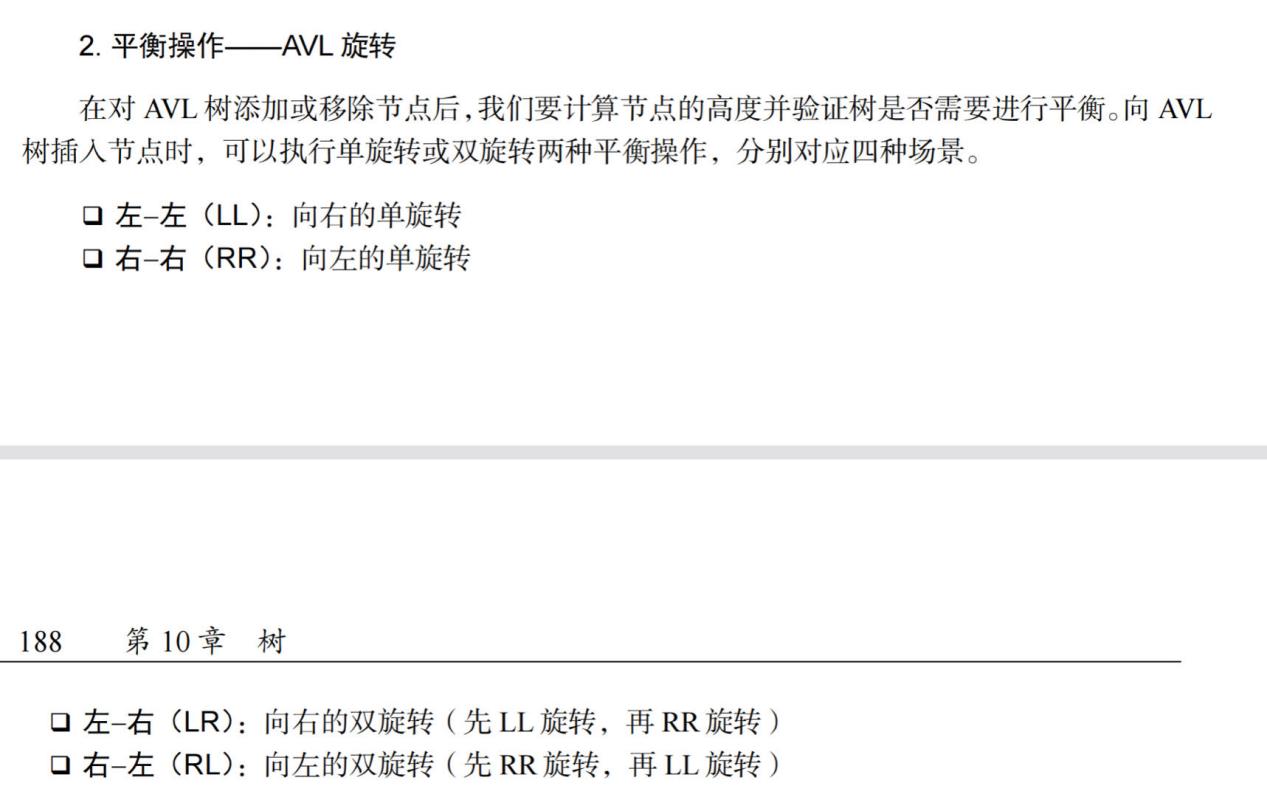


你需要重点关注我用红笔画出来的地方，它是用左侧节点的高度减去右侧节点的高估，通过判断返回值来判断树需要怎么样操作。

其中返回-2则表示右边的节点高度必左边的多2，说明这是一个不平衡的树。

2则反之。

你能说说为了让二叉搜索树实现自平衡，有点AVL旋转的操作吗？

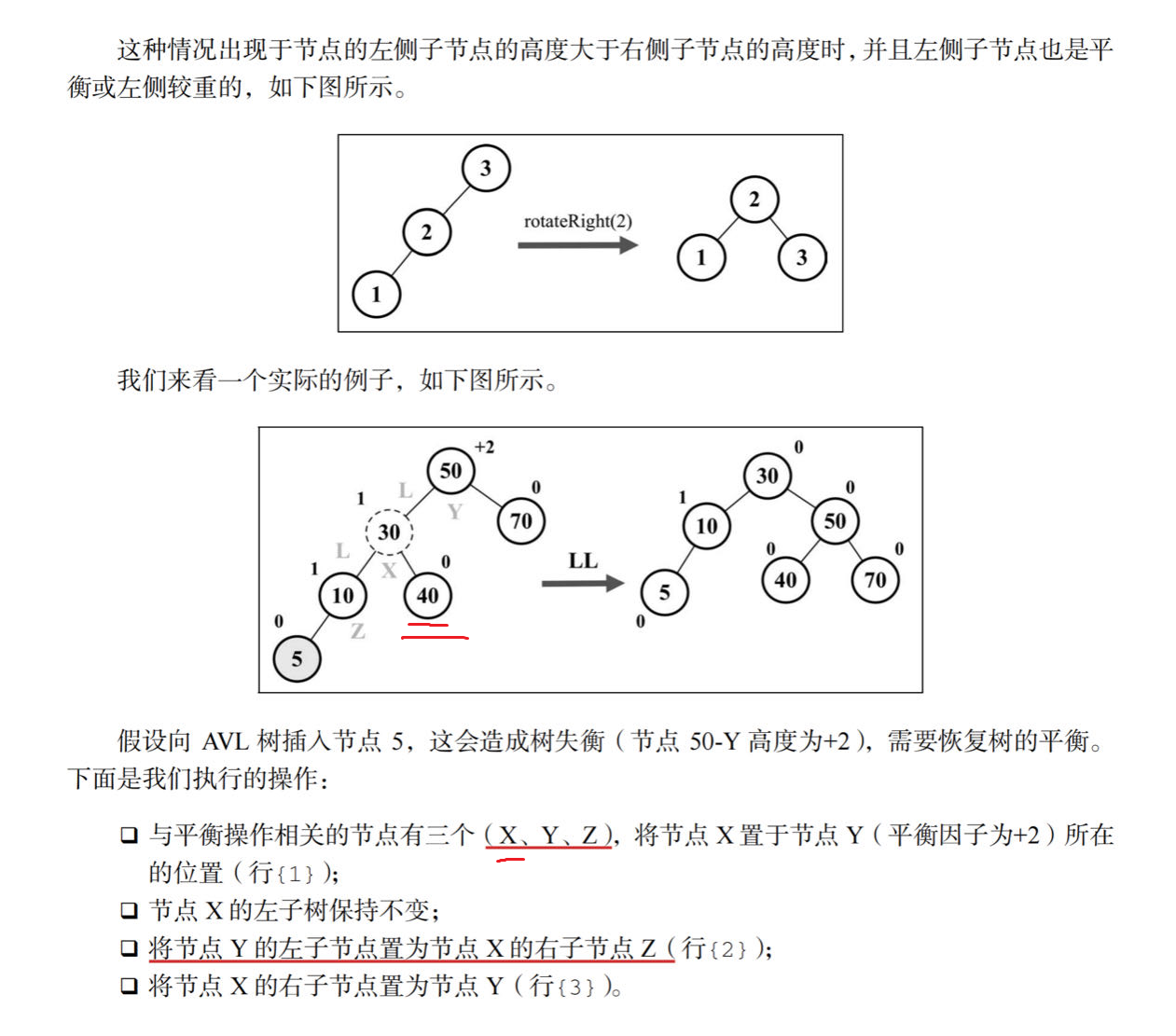


你能说说什么是二叉搜索树的LL（向右的单旋转）具体是怎么操作的吗？

在正式解释这个问题之前，我需要向你说明两点，这两点是你在学习二叉树操作的时候，必须的印在脑子里的。

1. 二叉树的一个节点最多只能有两个子节点，左侧子节点和右侧子节点。
2. 二叉搜索树的左侧节点的值比根节点小，右侧反之。

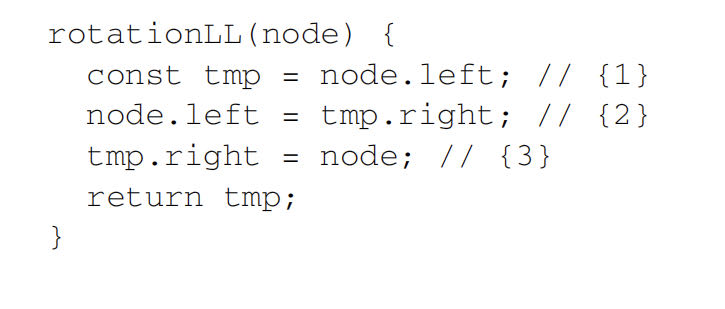
我们先来看一书上的解释



这里我来解释一下为什么LL是向右的单旋转，因为左边的高度大于右边的高度，所以是向右旋转以减轻左边的重量（高度）

在这里你可以将30看做X，因为他是X、Y、Z三个值得中间值，以它为新的根节点向右旋转。你就可以将10看做它的左侧节点，50看做它的右侧节点。但细心的你可能会发现，40属于30节点上多出来的一个节点，根据节点值40来判断，它大于新的根节点30，所以它需要放到30的右侧，50的左侧。

我们来看一下它的具体算法：

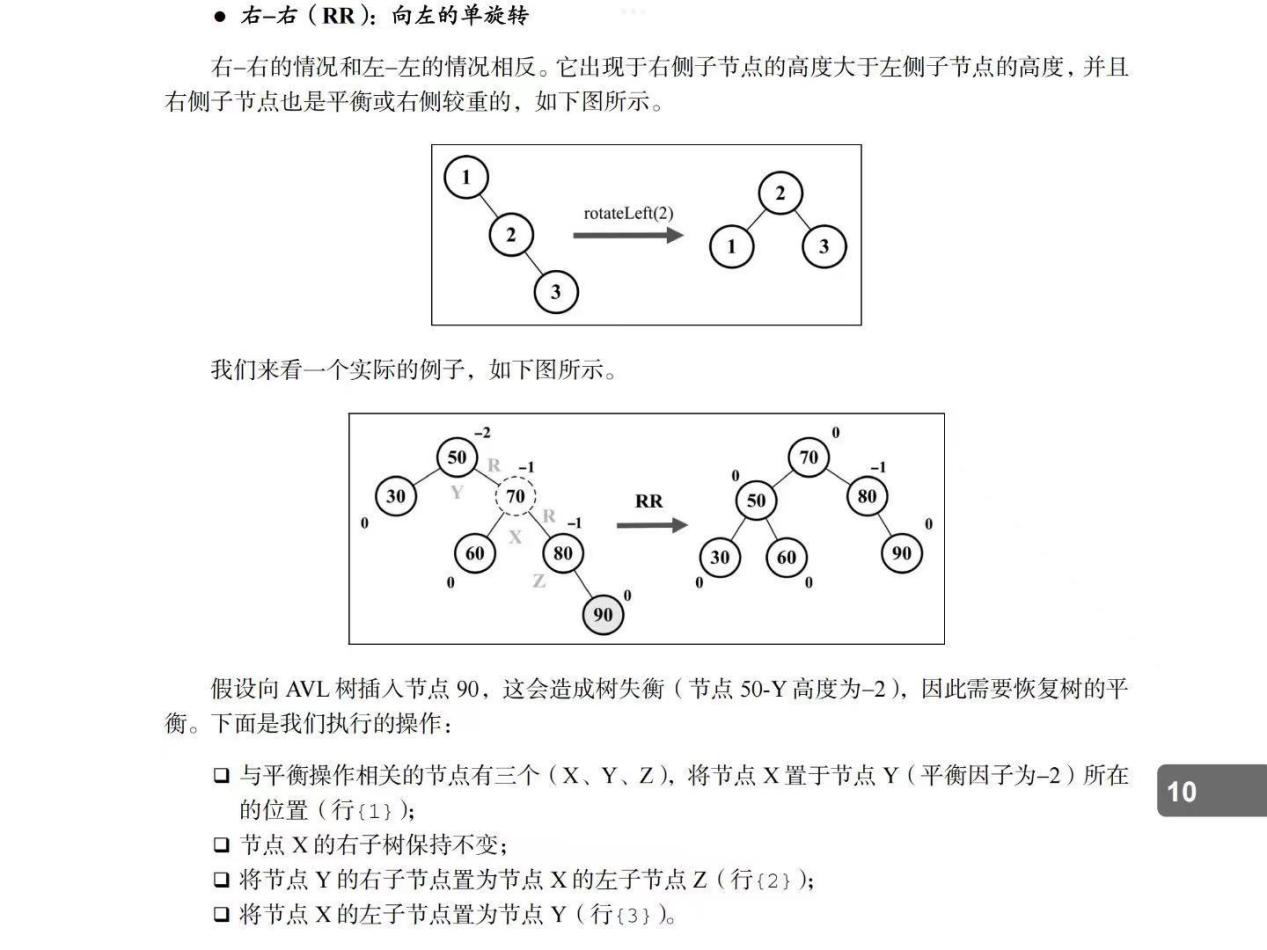


是通过平衡因子来判断具体是什么旋转，理解这类问题，你不需要向复杂了考虑，你只需要考虑一个用例就可以，只要一个想明白了，做好了，剩下的就是把这件事交给计算机重复的去做，因为计算机最擅长的就是做这个。

怎么判断一个树是需要向右还是向左旋转？

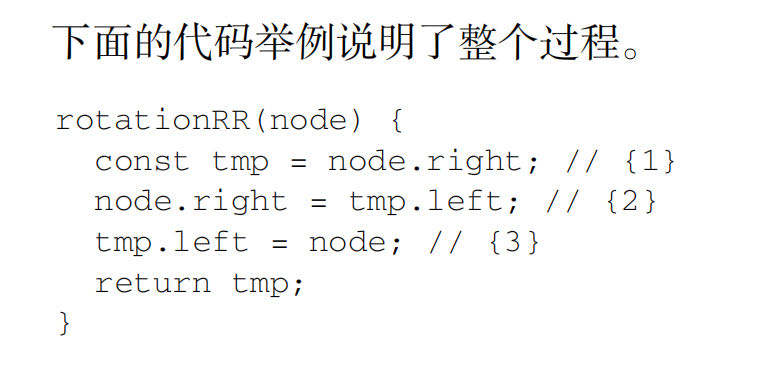
如果只是单旋转的话，还是很好判断的，你只需要要考虑左边树的高度是否大于或小于右边树的高度即可。

你能说说什么是二叉搜索树的RR（向右的单旋转）具体是怎么操作的吗？



其实单看这个图还是很好理解的，这个数需要向左旋转，70是50、70、80的中间值。左旋后需要考虑的是60这个多余的分叉放到哪里。60比70小，肯定是放在左侧，又因为60比50大，所以要放在50的右侧。

我们来看一下他的具体实现逻辑。



你能说说为什么LL是向右旋转而RR是向左旋转吗？

LL就是左边重，所以要向右旋转。RR就是右边重所以要向左旋转。

AVL自平衡树的每个节点的右侧子树和左侧子树的高度差是\_\_\_\_\_\_？

{-1,0,1}

AVL自平衡树中的每个节点都有\_\_\_\_\_\_\_\_？

Balance factor

在二叉搜索树中是否允许插入值相同的节点？

不允许

AVL自平衡树的每个节点的balance factor是怎么计算的？

就是左侧子树的高度减去右侧子树的高度。

AVL自平衡树，是在什么时候触发所谓的旋转的？

当你在AVL自平衡树中插入一个节点的时候，在节点插入完成后，它会计算这个插入节点的平衡因子。然后根据条件判断是否需要对树进行旋转平衡。

什么是二叉搜索树的左右(LR)向右的双旋转？

出现这种情况是因为，左侧节点的高度大于右侧节点的高度，并且左侧子节点的右侧比较重，在这种情况下，需要对左侧子节点先进行左旋(RR)，然后再对不平衡的节点在来一个右旋(LL)。

其中L是指LL，R是指RR。所以lR向右的双旋转的旋转顺序是先RR(子树)，再LL(完整的树)。

而且你注意因为是插入节点是递归的调用的，所以它永远不可能先旋转完整的树，在旋转子树。