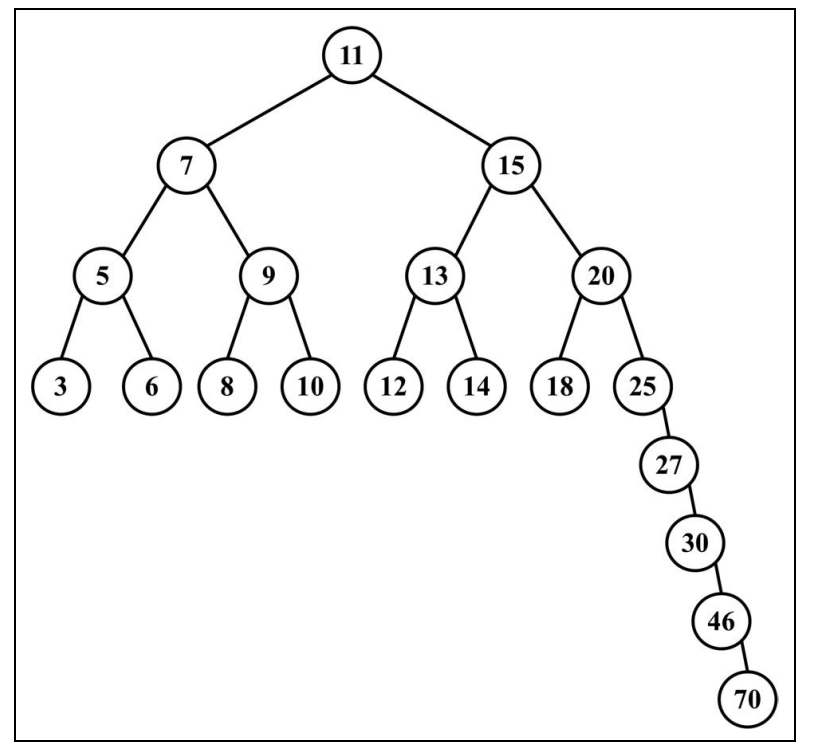
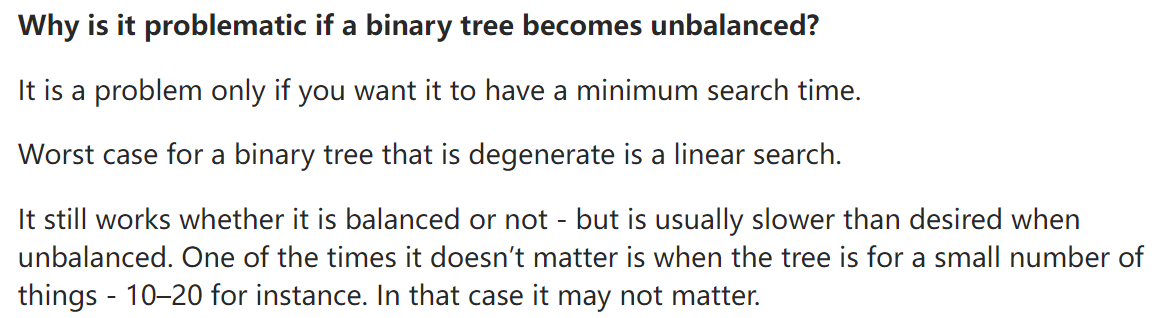
什么是AVL自平衡树？

AVL自平衡树是二叉搜索树的一种，它基于二次搜索树产生。它的产生是为了解决二叉搜索树两侧节点不平衡的。什么是两侧节点不平衡呢？请参看下图：



Avl树又叫自平衡书，他会在插入和删除节点的时候计算树的平衡因子，从而对二叉搜索树进行调整，进而达到树的平衡。

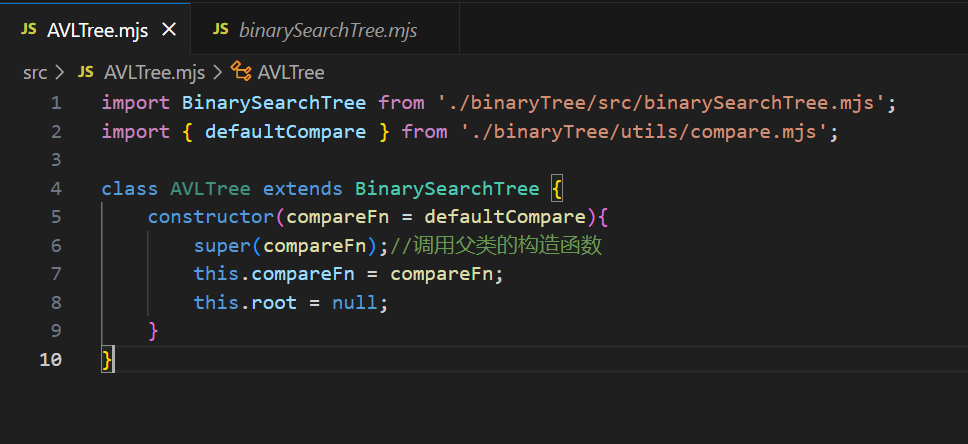
二叉搜索树为什么会产生两侧节点不平衡的问题？



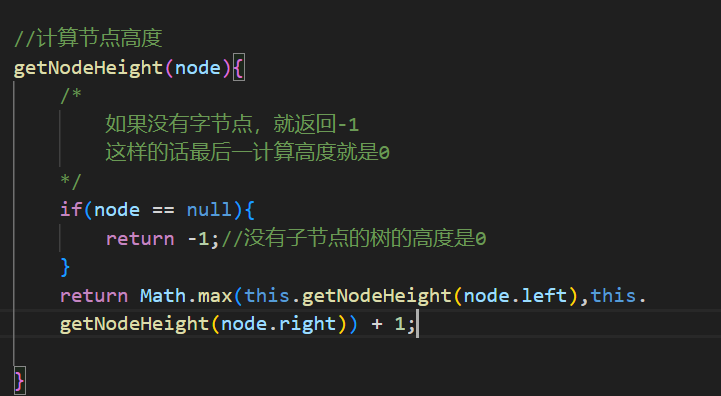
这个问题只有在你向最小化搜索时间的时候产生，最坏的情况就是二叉搜索退化为线性搜索。但不管二叉树是否平衡这个二叉树是能够正常工作的，如果你的数据量很少的话，轻微的不平衡是没有问题的，但是如果数据量庞大我们就要在插入和删除的时候就需要考虑这个不平衡的问题。

怎么样计算二叉搜索树的平衡因子？

所谓二叉搜索树的平衡因子，就是它的左侧子树的高度减去右侧子树的高度。所以想要计算出一个二叉搜索树的平衡因子就必须要先有一个用于计算二叉树高度的算法。当然在此之前我们需要构建好avl树的基础，这里我就直接把如何构建一个avl树的基础的代码贴出来了,因为本值上它是一个二叉搜索树，所以它的基础也是基于二叉搜索树构建的。

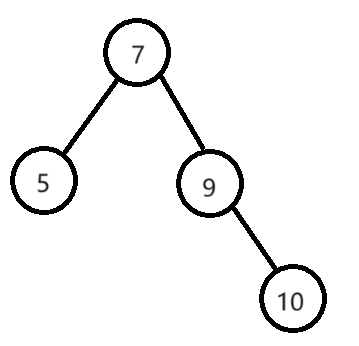


利用二叉搜索树构建一个avl自平衡树的基础



利用递归计算节点的高度

递归算法有点让人“大道以歧路多亡羊”的意思，这里我还是找一个最简单的节点来分析一下这个算法。

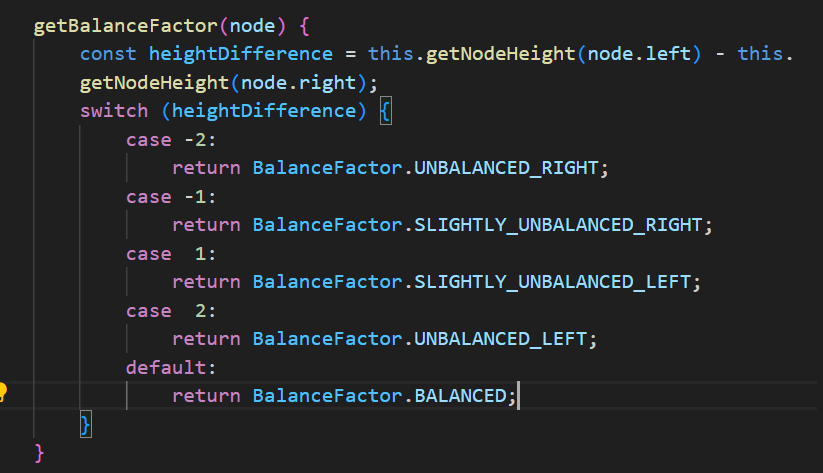


这个算法其实非常简单，你只需要知道没有字节的的节点高度为0,那个那么你就会对这个算法豁然开朗。所以对于递归算法你不能从头开始捋，你需要考虑的是递归的停止点，在这个停止点上将指定的值返回给上次函数运算。就这样一层层向上传递最终得出运算结果就行。

认识怎么样二叉搜索树才能算是不平衡？

我们知道当二叉搜索树的左右子树的高度不一致的时候，就算是“二叉搜索树不平衡”，那究竟要到什么程度才需要做avl旋转呢？我们所当左侧子树的高度减去右侧子树的高度不为1，-1，0的时候也就是两侧子树的高度差超过1的时候，我们就认为这个二叉搜索树是不平衡的，此时我们就需要通过平衡因子判断二叉搜索树需要怎么样的旋转。

认识平衡因子与对应的旋转操作之间的关系



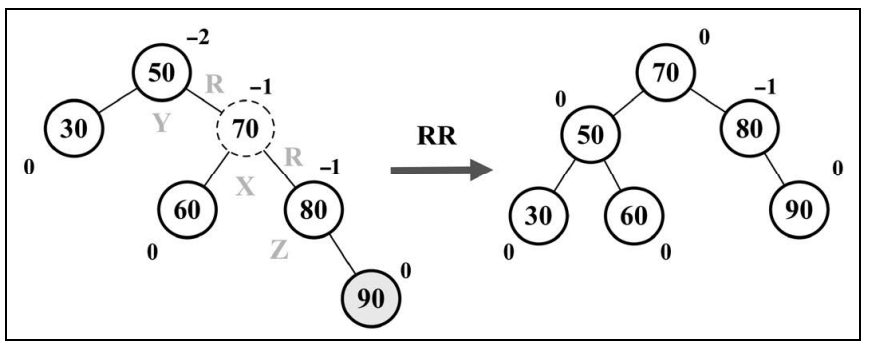
获取平衡因子的算法

通过观察上图的代码我们得知，针对平衡因子的不同我们要对二叉搜索树做的操作就不同。

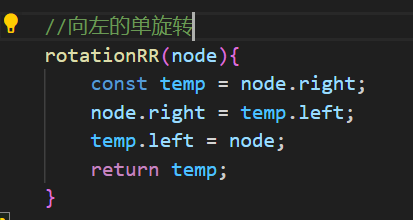
我们来一个个看看：

首先我们来看看什么是”UNBALANCED\_RIGHT”的右侧不平衡，此中情况指的是左侧子树的高度比右侧子树的高度要低2。对于这种情况我们需要对二叉搜索树做RR（向左的）单旋转。

下图是一个需要做向左的单旋转的二叉树，我们来分析一下怎么做向左的单旋转



当我们向计算平衡因子的传入值为50的节点后，通过计算我们应该能得出它的平衡因子是-2,也就是右侧太重，此时我们需要将其左旋变成上图右侧所示的部分。值得注意的是这个左旋操作的套路是固定的，无论你怎么操作，只要能实现上图所示的效果就行。

我们先来看一下具体的代码：  


针对向左的单旋转，我们需要先用一个零时变量将根节点（平衡因子为-2的节点）的右侧子节点先保存下来，因为我们要用它的右侧字节的作为旋转后的树的根节点。此时我们需要做的是：

1. 将原本的根节点的右侧节点改为临时变量存储的节点的左侧节点，也就是原本根节点的右侧子节点的左侧子节点。
2. 然后将临时变量存储的节点的左侧节点指定为原本的根节点。

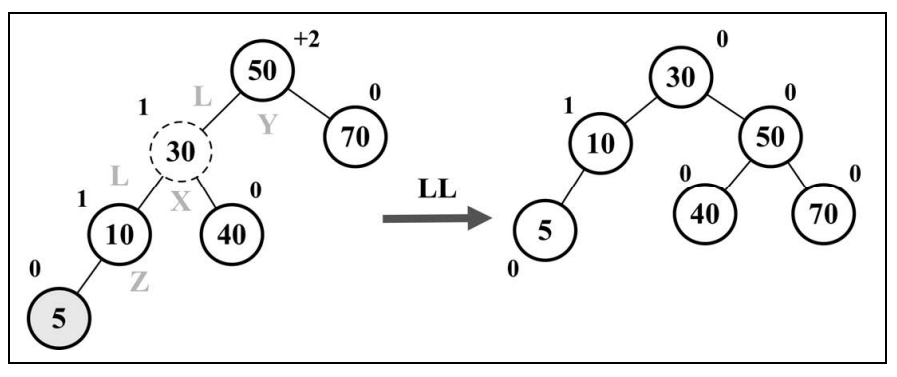
我们来一步步分析一下，首先我们已知将根节点的右侧子节点存储到临时变量上是要用这个临时变量存储的节点作为旋转后的树的根节点。

对于上面提到的第一点所做的操作，原因很简单因为临时变量存储的是原本根节点的右侧节点，对于原本根节点的右侧节点来说，它只能存放节点值比它大的节点，恰好temp.left满足。

而对于上图提到的第二点来说因为我们就是要对右侧重的树进行向左旋转，且通过第一点提到的步骤，我们已经将右侧字节的的左侧节点挪走了，此时我们就需要为它指定新的节点。那它应该是谁呢？就是我们前两步所操作的原本的根节点。

最终我们将新树的根节点return出去。

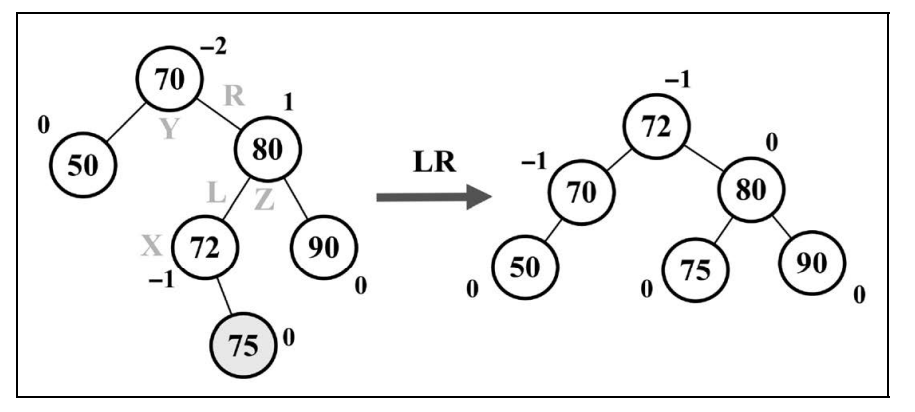
向右的单旋转



对于向右的单旋转，它的操作完全与向左的单旋转相反，你只需要对照着去做就ok了。

什么是双旋转？

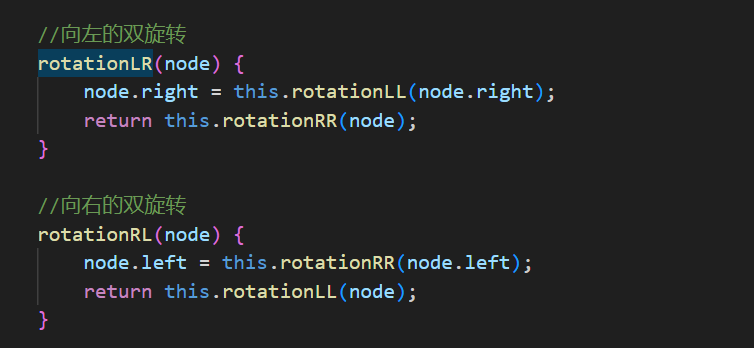
我们来看一下实际例子：



在没有插入75这个节点之前这个这个树其实是平衡的，但是一旦当我们插入了这个75节点后，根节点的平衡因子就会变成-2,也就变成了一个不平衡的树了，但要怎么样才能让它变得平衡呢？很简单，我们只需要将它的左侧节点进行一个向右的单旋转(LL),然后再对整个树做一个向左的单旋转（RR）即可。这样一算，我们就对这个不平衡的树做了两次旋转，因此叫双旋转。上图给出的树本质上是右侧重，所以它需要向左旋转，因此它也叫向左的双旋转（LR）也就是先LL(向右旋转)再RR(向左旋转)。值得注意的是，对于这种旋转操作，由于递归的特性，所以永远都是先旋子树。

Tip:一定要注意单旋转和双旋转之间的区别噢！

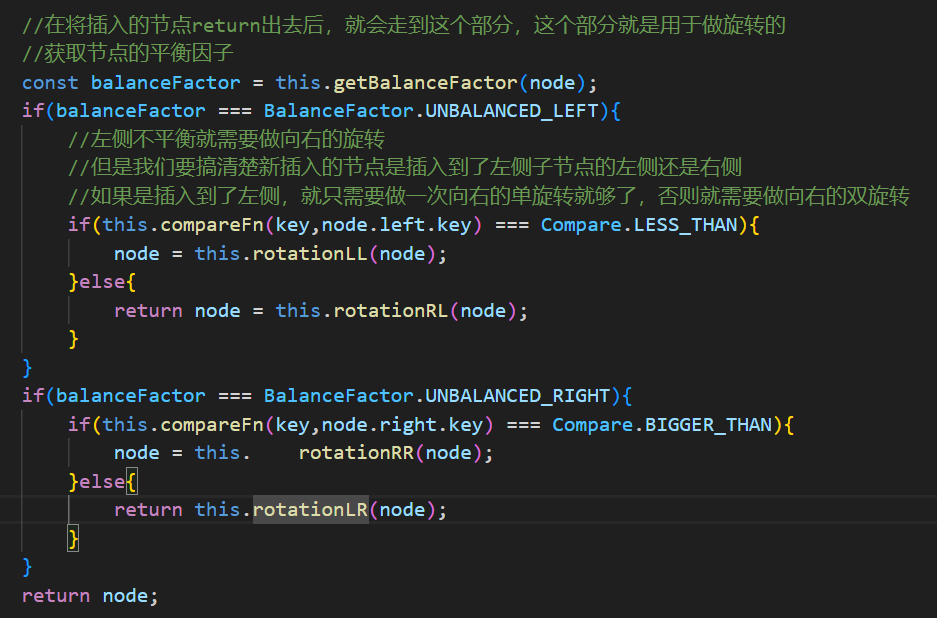
双旋转操作



向avlTree中添加节点

现在我们需要讨论如何向avl树中添加节点了，因为在插入节点之后，我们需要判断这棵树是否还平衡。如果不平衡的话，我们就要做相应的旋转操作。

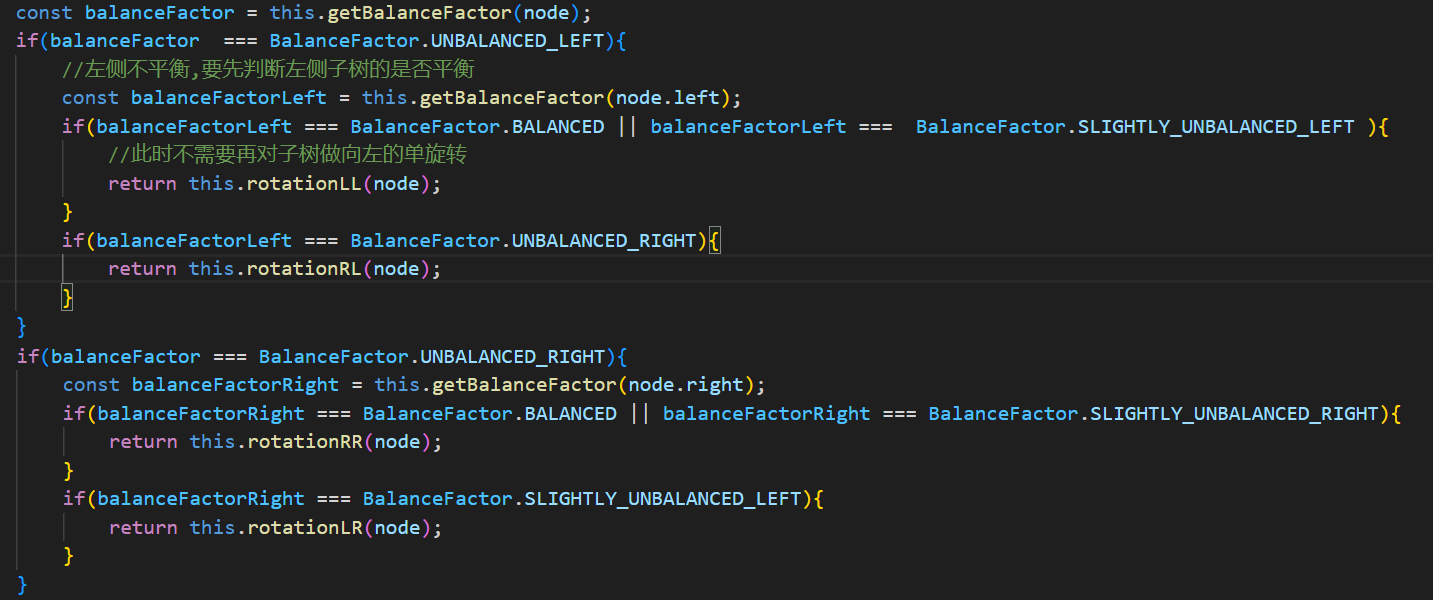
我们来看一插入节点的核心代码



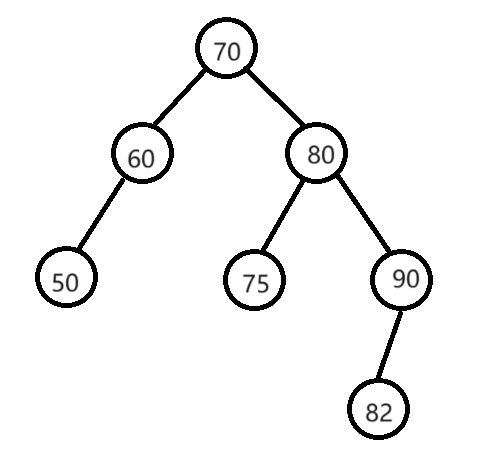
从avlTree中删除节点

在删除节点之后，我们向上递归的判断树是否平衡，但是有时候我们删除的节点它并不是一个叶子节点，有可能我们需要对删除后的节点做旋转操作。

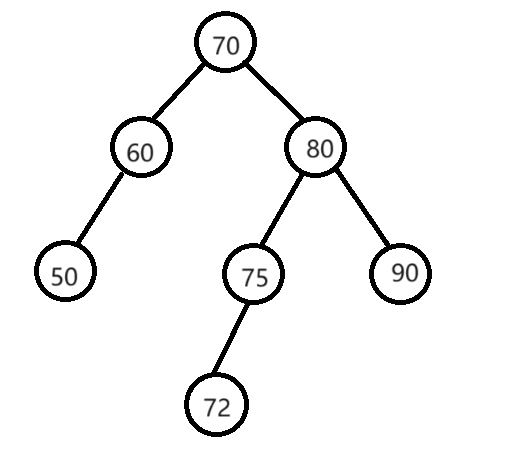
我们来看一下它的核心操作



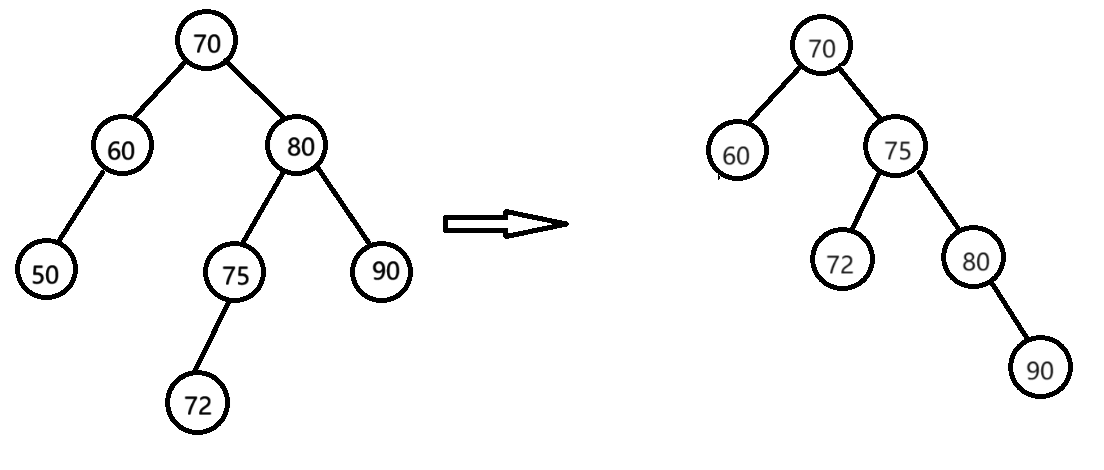
首先我们拿到了删除后的节点或者递归的向上判断删除节点的父节点的平衡因子，根据平衡因子判断是否需要旋转。以右侧不平衡为例，我们需要先判断不平衡的树的右侧子节点具体是什么不平衡，如果右侧子树只是轻微的右侧不平衡，那我们就只需要对它做一个RR。就可以了，但如果此时它右侧的子树又出现轻微的左侧不平衡的话，我们就需要对他进行向左的双旋转。

我来找一个图例方便你理解：  


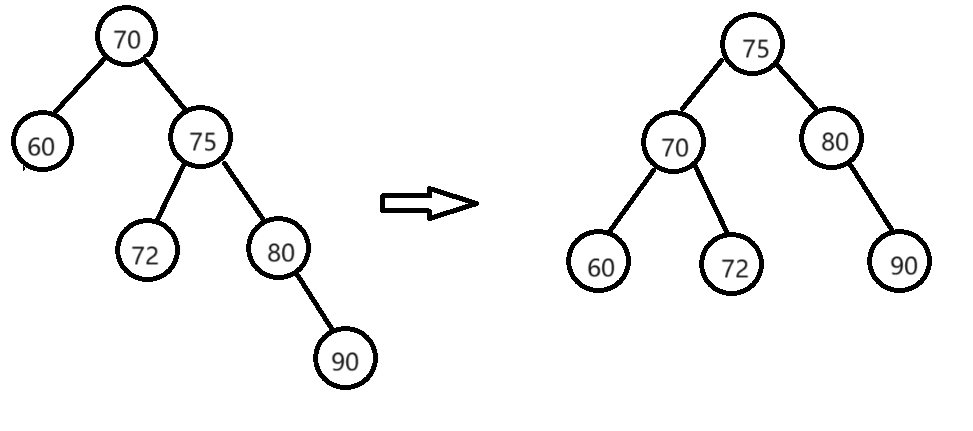
现在假设我们有上图这样的二叉树，我们现在要移除50，这个节点，当移除了50这个节点后，很明显这个课树就不平衡了。他的平衡因子是-2，也就是右侧不平衡。但对于右侧不平衡的这种情况，我们还需要判断右侧子树的状况。通过观察节点值为80的这课子树，我想你应该能很明显的看出，它属于轻微的右侧不平衡，所以它进第一个if条件，走一个向左的双旋转就可以。那现在，假设我们把82去掉，换成72，也就是如下图所示



此时我们移除key为50的节点，同样会导致整棵树变得右侧不平衡，此时我们就需要再次判断右侧子树的状况，此时很明显，是左侧轻微的不平衡，所以它会走到第二个if条件中，先对右侧子树进行一个LL右旋，在对整棵树进行一个RR左旋。



对整棵树做一个左旋RR就ok了



至此，我们已经讨论完如何使用Js构建一个基础的avl树的过程了。后续我会讨论，如何在avl树的基础上构建红黑树。