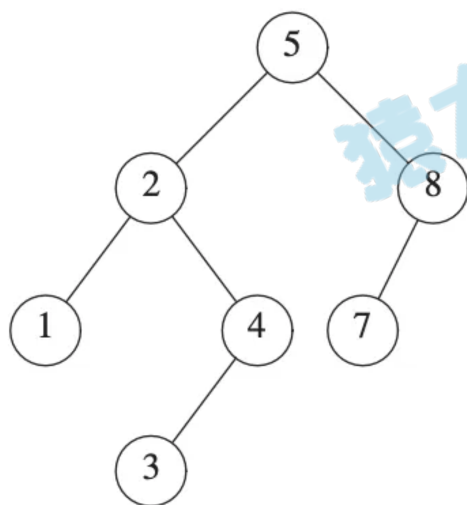
AVL树

一.简介

前面我们介绍了二叉搜索树（BST-binary search tree）,因为如果插入到二叉搜索树中的数据如果本身就是有一定顺序的，那么就会出现某个节点的子树变成了斜树，其实也就变成了链表，会导致查询效率降为O(n)，因此引出了平衡二叉搜索树，通过一个**平衡因子k**来约束二叉搜索树的高度，要求二叉搜索树的**每一个节点的左右子树的高度差不能超过k**，所以平衡二叉搜索树其实准确来说叫做**自动平衡二叉搜索树**，因为只要高度差出现了超过k，那么就需要执行相应的**平衡算法，对树进行重新组合**，使得高度差再次回归小于等于k的范围。而当k=0的时候，这样的二叉搜索树叫做**满平衡二叉搜索树**。但是大部分实际情况下，k=0是比较难实现的，但是当**k=1**的时候的平衡二叉搜索树仍然具有O(logn),而这样的平衡二叉搜索树有一个专门的名字-**AVL树**。

> The name AVL tree is coined after its inventor's names – Adelson-Velsky and Landis.

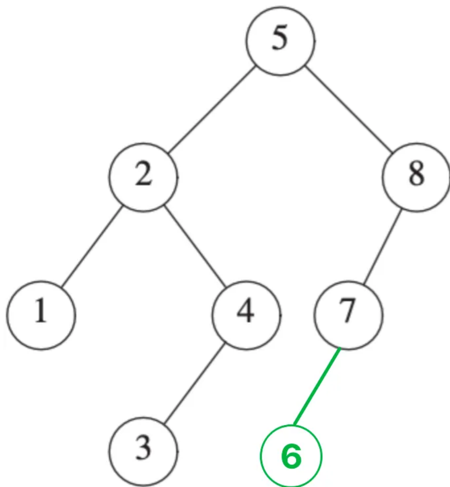
二.AVL树平衡问题引入



猿大晖@b站

对于上面的二叉搜索树，它其实是满足AVL树特性的，因为每个节点的左右子树高度差都没有超过1。

但是，如果我们插入一个节点6，那么就会导致节点8出现失衡，如下所示：



所以，我们需要通过一定的平衡算法，来修改，调整这颗树的形状，而这样的修改操作我们叫做旋转。

> When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is potentially difficult is that inserting a node could violate the AVL tree property. (For instance, inserting 6 into the AVL tree in Figure 4.32 would destroy the balance condition at the node with key 8.) If this is the case, then the property has to be restored before the insertion step is considered over. It turns out that this can always be done with a simple modification to the tree, known as a **rotation**.

上面是我摘自《Data Structures and Algorithm Analysis in C++(4th)》这本书中的一段话，我们可以看到，其实所谓的旋转**rotation**，只是这种对树结构修改的称呼，等下我们介绍完详细的修改操作后，大家就会知道，这样的操作**有点类似**于将树按照某个节点作为支点来进行旋转，但实际上，并不是完全的旋转操作，因此大家不要总是陷在这个词的含义上。

三.AVL树平衡方法

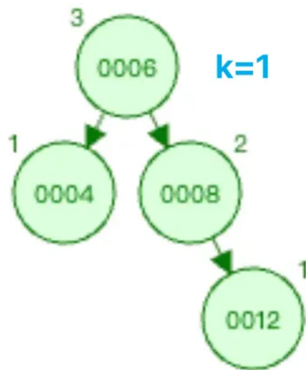
AVL树在插入或者删除的时候，如果出现了平衡因子超过1的情况，需要进行自动平衡，使树本身继续保持平衡，也就是每个节点的左右子树的高度差的绝对值小于等于1.

下面，我们将基于插入来分析下导致AVL树出现失衡的几种情况。

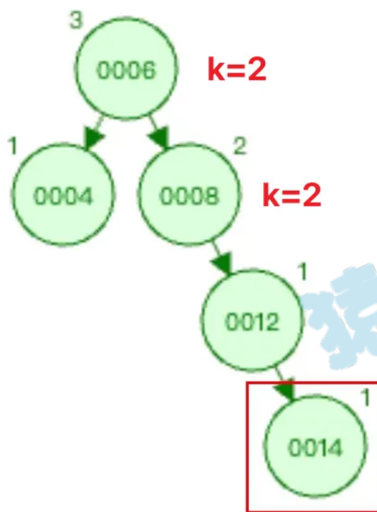
3.1 RR插入情况导致的失衡

3.1.1 简单场景

所谓RR插入导致的失衡，是指插入的节点元素是在本次出现的深度最大的失衡树的根节点的右孩子的右子树上插入的。比如下面的这种情况：



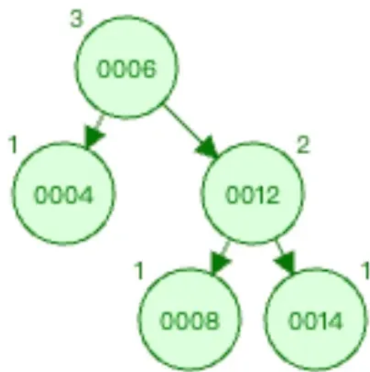
如果我们插入一个元素14，那么就会出现下面的情况：



猿大晖@b站

可以发现，6节点和8节点失衡了，并且8节点的深度大于6节点的深度。而这里插入的14是在失衡的8节点的右孩子的右子树上面，所以这就是RR插入导致失衡的情况。这时我们需要使用**左旋**的平衡方式。

所谓左旋，就是以深度最高的失衡的节点（这里也就是8）为轴，向左旋转，也就变为了这样：



此时，这颗树已经平衡了，再次满足AVL的性质。

这里我想说明一下所谓的以某个节点为轴进行旋转，我们可以将这个为轴的节点想象成一个滑轮所在的位置，二叉树的箭头指针就是滑轮的线，左旋，就是向下拉动滑轮左边的绳子，所以也就出现了上图的平衡之后的状态了。

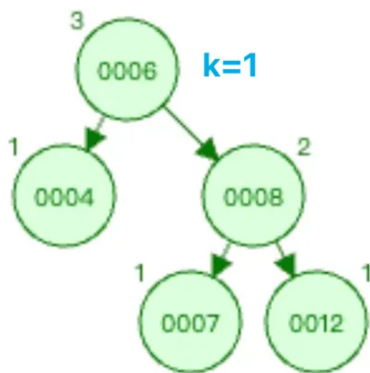
上面这是一种理解旋转的方式，我个人是比较推荐的，下面介绍另外一种：

我们还可以将失衡节点8的右孩子节点12及其子树固定，然后将8围绕节点12向左旋转（也就是逆时针旋转），然后再将节点12往上提，连接为节点6的右孩子。

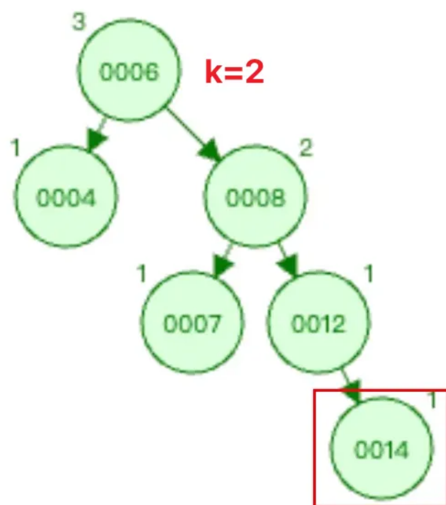
猿大晖@b站

3.1.2 需要重组的场景

比如下面的二叉树，此时它是平衡的



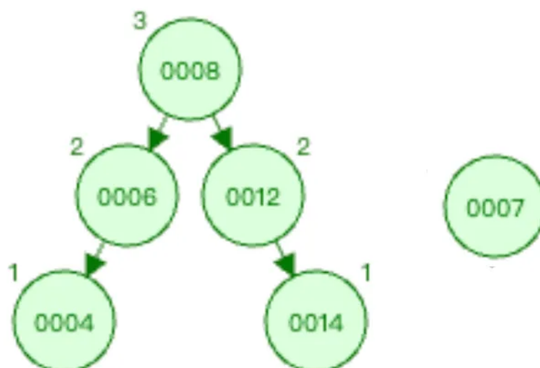
但是，如果我们插入一个元素14，也就是变为下面的样子：



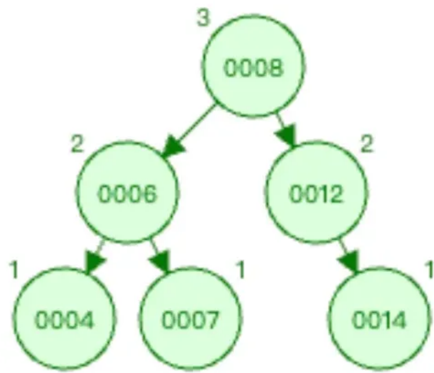
很明显节点6失衡了，这里同样的，是失衡节点6的右孩子的右子树插入的，所以也是RR插入导致失衡的情况，但是如果我们以6为轴，左旋的话，就会发现节点8仿佛多了一个7节点不知道放哪里，如果还作为8的孩子，那么节点8就变为三叉树了。所以此时，我们需要在左旋之后，再多做一个重新组合的操作。所以具体的步骤也就是：

1. 尝试以失衡的节点6为轴，左旋
2. 旋转后节点6应该变为了节点8的左孩子，但是我们发现节点8已经有了一个左孩子，于是我们需要先抛弃现有的左孩子节点7，然后再左旋转
3. 于是出现下面的场景，出现了一个孤立的节点7

猿大师官方站



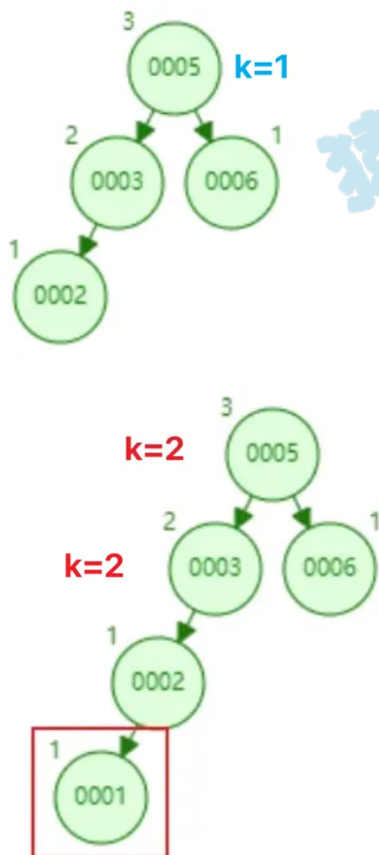
4. 因为节点6原来是有一个右孩子的，现在没了，所以刚好，将被孤立的节点7给节点6当做右孩子。也就变成了下面这样的重新平衡的状态了。



3.2 LL插入情况导致的失衡

这种情况，很明显和RR是完全一样的道理，只是RR的一种镜像操作了。我就稍微简略一点来介绍。

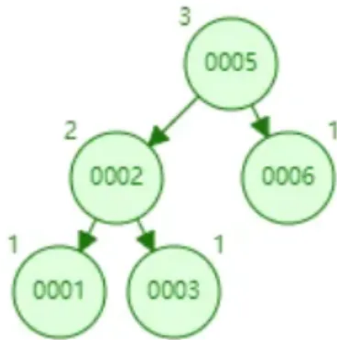
3.2.1 简单场景



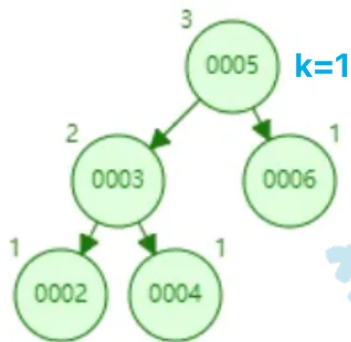
猿大晖@b站

因为是在最深的失衡节点3的左孩子的左子树上插入的，所以是LL插入导致失衡，这里我们需要进行右旋操作。

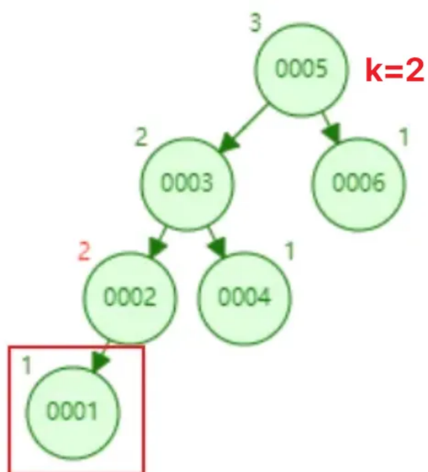
就是以深度最高的失衡的节点（这里也就是3）为轴，向右旋转。也就变为了：



3.2.2 需要重组的场景



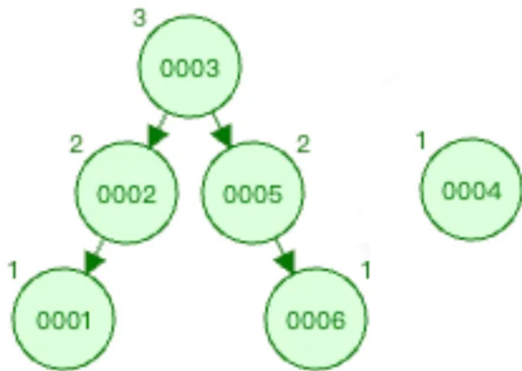
猿大晖@b站



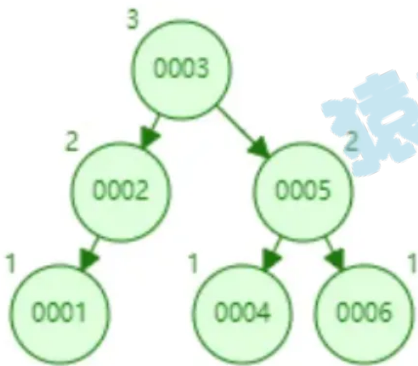
插入元素1之后，节点5失衡了，因为是在节点5的左孩子的左子树上插入的，所以也是LL插入导致失衡的情况，同样的，这里如果直接以节点5为轴右旋，那么节点3的右孩子4就无处安放了，所以需要在右旋之

后进行一次重组，也就是按照下面的步骤进行旋转操作：

1. 尝试以失衡的节点5为轴，右旋
2. 旋转后节点5应该变为了节点3的右孩子，但是我们发现节点3已经有了一个右孩子，于是我们需要先抛弃现有的右孩子节点4，然后再右旋转
3. 于是出现下面的场景，出现了一个孤立的节点4

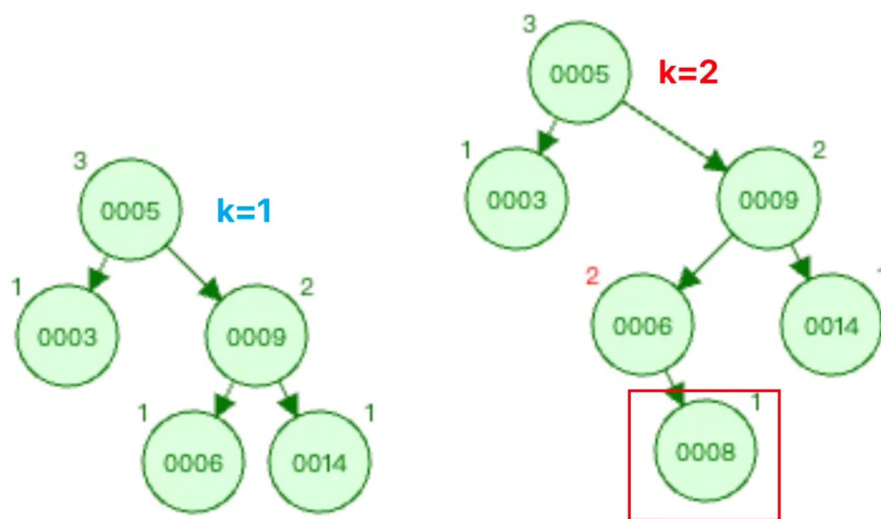


4. 因为节点5原来是有一个左孩子的，现在没了，所以刚好，将被孤立的节点4给节点5当做左孩子。也就变成了下面这样的重新平衡的状态了。



猿大晖@b站

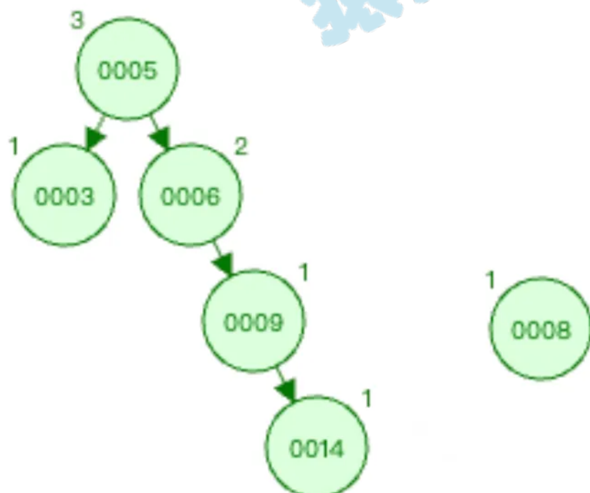
3.3 RL插入情况导致的失衡



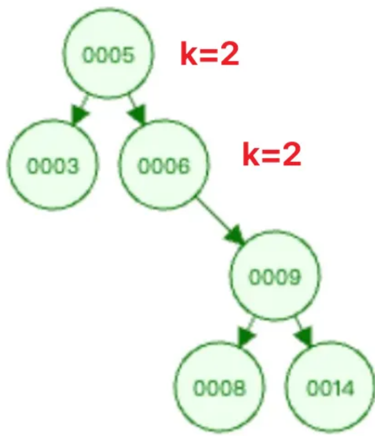
可以看到，如果我们在上面这样的AVL树上插入一个节点8，那么就会导致节点5失衡，而此时插入的节点8是在失衡的节点5的右孩子的左子树上，所以这是一种全新的失衡插入情况。我们称为RL插入情况导致的失衡。

这种情况我们需要进行两步旋转操作和重组操作才能重新平衡，具体步骤如下：

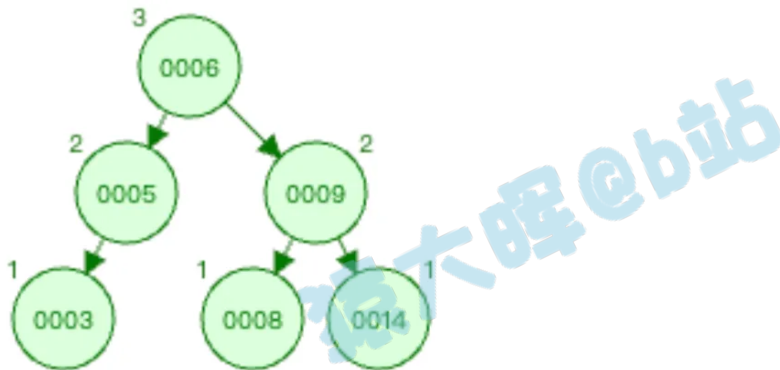
1. 先以**失衡节点5的右孩子节点9为轴**，右旋。
2. 我们发现，右旋后，节点9变为节点6的右孩子，但是实际上节点6已经有了一个右孩子节点8，所以同样的，我们需要先丢弃节点8，然后再右旋，也就是变为下面的状态：



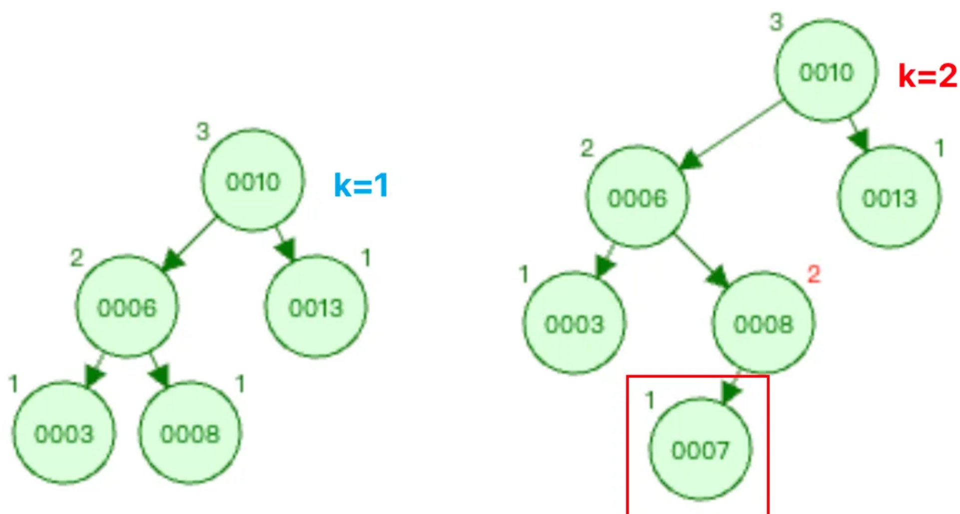
3. 然后，我们发现原来节点9是有一个左孩子6的，但是现在左孩子6造反成为了节点9的父节点，所以我们将当前成为了弃儿的节点8给节点9作为左孩子。这其实就是重组的过程，其实和前面说的都是一样的道理。也就是变为了下面的状态：



4. 到此为止，我们完成了右旋（重组也当做是包含在旋转操作中的）。但是树还是不平衡的，于是我们还需要继续做一个左旋操作，**这时要以当时失衡的节点5为轴，进行一次左旋**。因为节点6本身没有左孩子，所以本次旋转节点5可以直接作为节点6的左孩子而不需要现丢弃再重组，于是就变为下面的状态，也就重新平衡了。



3.4 LR插入情况导致的失衡

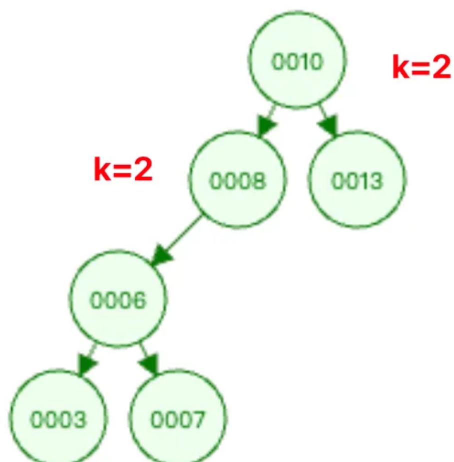


可以看到，如果我们在上面这样的AVL树上插入一个节点7，那么就会导致节点10失衡，而此时插入的节点7是在失衡的节点10的左孩子的右子树上，所以这也是一种全新的失衡插入情况。我们称为LR插入情况导致的失衡。

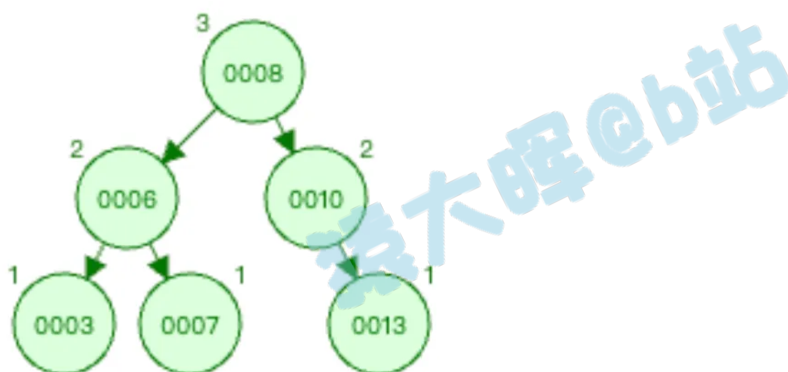
这种情况我们也需要进行两步旋转操作和重组操作才能重新平衡，具体步骤如下：

1. 先以**失衡节点10的左孩子节点6为轴**，左旋。
2. 我们发现，左旋后，节点6变为节点8的左孩子，但是实际上节点8已经有了一个左孩子节点7，所以同样的，我们需要先丢弃节点7，然后再以节点6为轴左旋，也就是变为下面的状态：

3. 然后，我们发现原来节点6是有一个右孩子8的，但是现在右孩子8造反成为了节点6的父节点，所以我们将当前成为了弃儿的节点7给节点6作为右孩子。这也就是重组的过程，和前面说的也都是一样的道理。也就是变为了下面的状态：

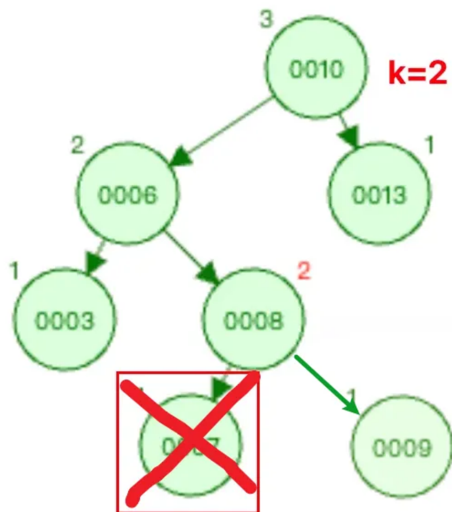


4. 到此为止，我们完成了左旋（重组也当做是包含在旋转操作中的）。但是树还是不平衡的，于是我们还需要继续做一个右旋操作，**这时要以当时失衡的节点10为轴，进行一次右旋**。因为节点8本身没有右孩子，所以本次旋转节点10可以直接作为节点8的右孩子而不需要现丢弃再重组，于是就变为下面的状态，也就重新平衡了。



3.5 补充说明

上面我们在介绍LL和RR的情况时，按照是否需要重组分了两类情况来讨论。通过上面的描述我们可以发现，其实是否需要重组，就是取决于：以节点X为轴进行左旋转，那么节点X的右孩子节点Y等于造反了，也就是节点Y势必会变为节点X的父节点，此时节点X也肯定会变为节点Y的左孩子，所以，如果节点Y本身就存在一个左孩子，那么就需要进行重组了，将这个原来的节点Y的左孩子转为节点X的右孩子。（好一个偷梁换柱，可以理解为，节点Y利用自己的左孩子来替代了自己作为节点X的右孩子，然后自己跑去篡位，变为节点X的父节点了哈哈哈，相反，如果右旋也是一样的道理了）



比如上图所示，假如在讨论LR的情况时，我们插入的不是7而是9，那么就是插在8的右孩子处，所以第一步的左旋6就可以直接作为8的左孩子不需要重组。

猿大晖@b站