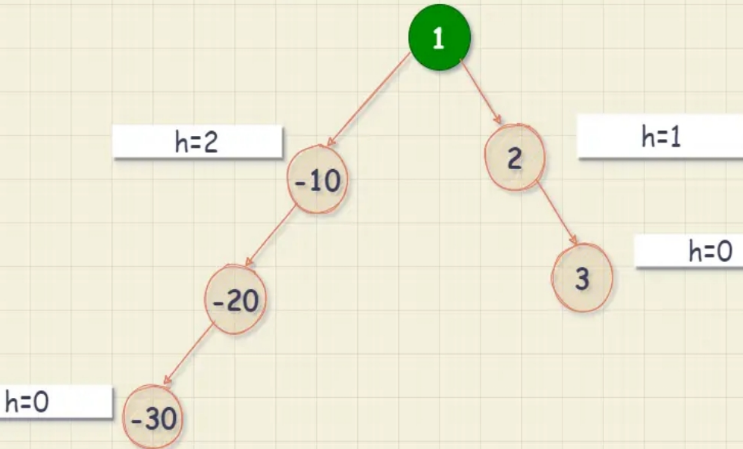
## AVL树

**1.术语**

**1.1 树高**：

计算节点高度公式： ;

左图中，**节点1**的高度：

;

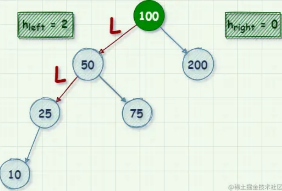
1.2**失衡类型**

在AVL树的定义中**左右子树高度差**时就破坏了AVL树的平衡，属于失衡节点；

在AVL中总共有4种类型的失衡：

|  |  |
| --- | --- |
| * LL型 | * LR型 |
|  |  |
| * RR型 | * RL型 |
|  |  |

1.3 **如何判断失衡节类型**

以LL 型来讲如何确定节点是否失衡？如何判断失衡的节点属于哪种类型的失衡？

  1.3.1: 判断是否失衡

100节点的左分支高度：，右分支高度： ;左右分支高度差2，因此100节点左右分支失衡；

1.3.2: 确定失衡类型:

  以100节点的视角来看，左分支的高度大于右分支，属于：L 型; 继续观察100节点的左节点：50，再来看50的左右分支高度：50节点的左分支高度大于右分支，因此是 ： L 型；这2个组合在一起就是 ：LL型了；其他三种类型都是一样判断方法；

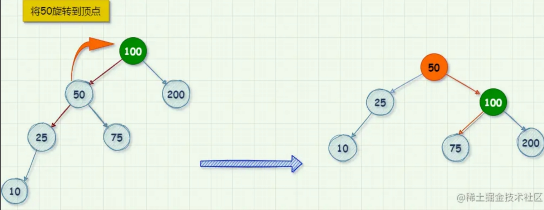
1.4 **如何调节失衡节点**

那么如何让左右平衡呢？**通过让高度大的分支向高度小的分支旋转，使高度大的分支节点减 1，高度小分支的节点加 1；这样就可以让左右分支重新平衡。**

上面四种类型以旋转次数来划分又可以分为：**单旋转，双旋转**；

* 单旋转：旋转1次可以让失衡节点恢复平衡；LL, RR；
* 双旋转：旋转2次可以让失衡节点恢复平衡；LR, RL;

**1.4.1 单旋转以LL为例：**

****

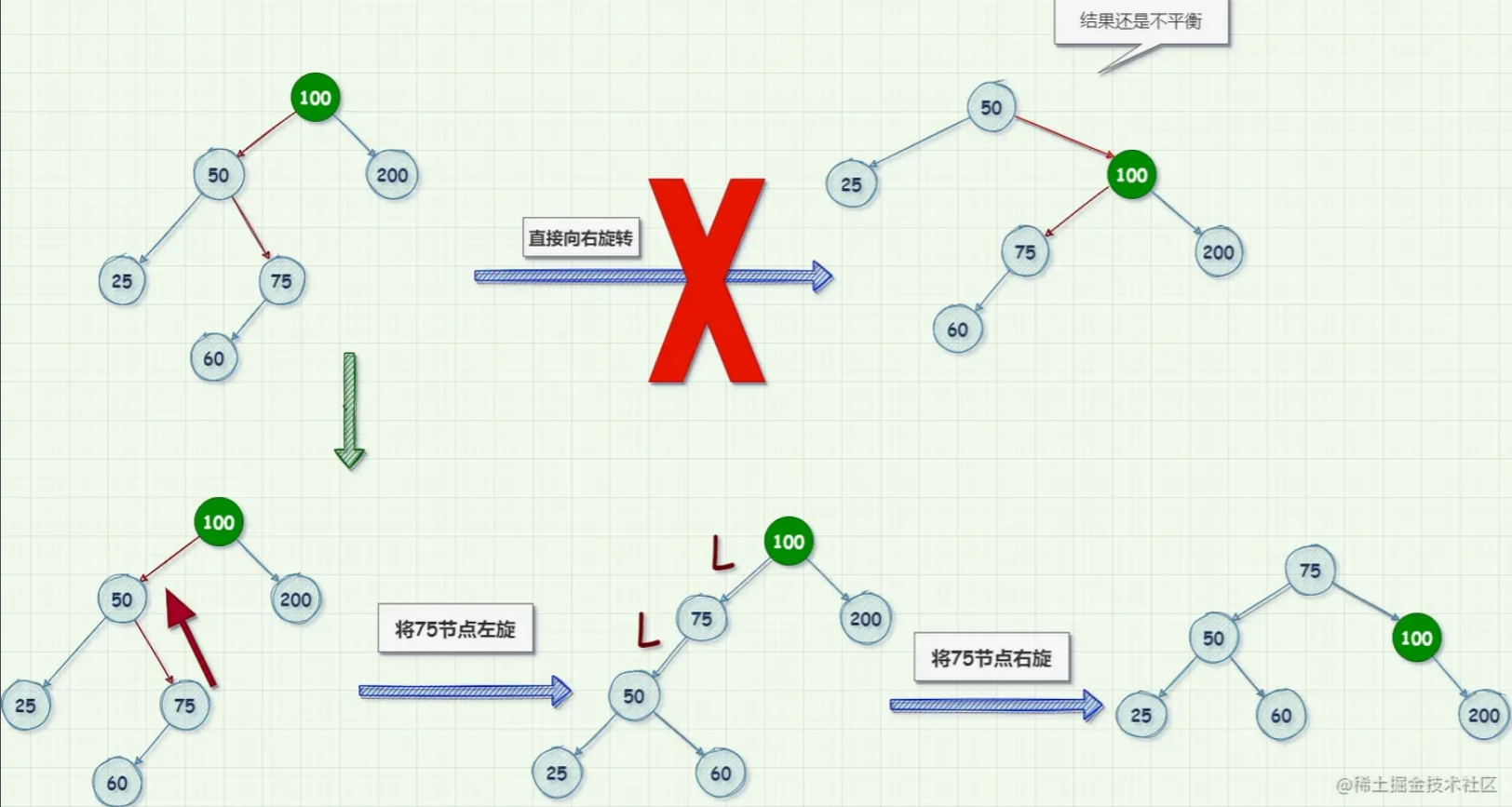
 在旋转时要注意2个节点：50，100之间的关系变化；

* 旋转之前： 100是50的父节点；50是100的左节点；
* 旋转之后：
  + 对于50节点来讲：50是100的父节点，50的右节点是100；
  + 对于100节点来讲：100的父节点是50，100的左节点是 旋转之前50的右节点75 ；
  + 对于75节点来讲：需要更新父节点，将父节点更改为 100；
  + 对于100的父节点来讲：100的父节点parent要更新子节点，将指向100的指针指向 50；

这就是单旋其实不难，只需要看图就能想清楚节点之间的关系变化；提到的75节点，100的父节点；这些都是围绕50，100节点来看的，因为 50，100节点的位置关系发生了变化75以及100的parent节点才随之改变；如果还是不太清楚建议手动画一下，在画图的过程中就明白了；



**1.4.2 双旋转以LR为例：**

****

对于LR型失衡如何调整：

* 首先,将LR型通过左旋 🡪 转换成LL型；
* LL型-通过右旋🡪使节点平衡；

  对于RL型，就先将RL -> RR ;再对RR进行旋转；通过上面对LL，LR旋转的分析可以看出只要弄清楚了LL 型其余三个就都明白了，逻辑都一样；

# 2. AVL树代码

insert过程和二叉搜索树相比多了一条：新加入Node节点之后，判断该节点的父节点的左右分支是否是平衡的（左右分支高度差不超过1）? 图示

描述已自动生成

  如何判断呢？在Node节点中加入高度属性:;要判断一个节点的左右分支是否失衡，就可以直接获取左右子节点的高度计算出高度差来判断；在确认了节点左右子节点高度失衡之后，接着利用高度差来判断出节点的失衡类型；

 无论是判断节点是否失衡或者是失衡的类型都是依据节**点左右子节点高度差**来判断的，因此节点高度是关键。

**2.1那如何更新每个节点的高度？**

每个节点在刚插入AVL时都是在叶子节点，因此每个新插入的节点高度都是0；而新插入的节点直接影响了它parent节点的高度；我们看一个例子：

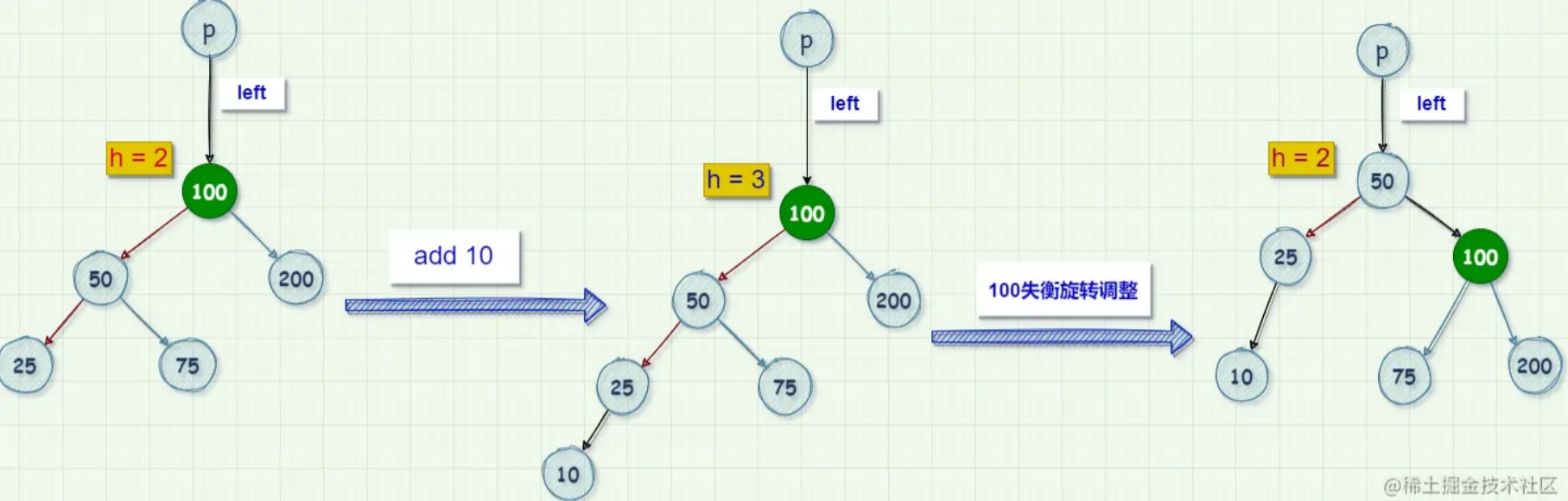


在上面的两个例子中第一个例子高度更新完25节点高度时，继续向上更新50节点的高度，50节点根据左右子节点高度计算出新高度与之前没有变化，因此50节点高度不变，这个时候没有必要继续向上更新节点高度；

**补充**： 在递归到最后一层执行插入时，函数参数 node,即为新节点的parent 节点。insert(struct node\* node, int x): 所以，执行插入后，首先更新 parent高度。 递归返回段，时再更新上一层高度，直到回退到根节点。

第二个例子：更新完25时，继续更新50，发现50节点高度也更新了就继续向上更新50的parent节点高度。

  通过这2个例子可以看出当更新到某个节点时，节点高度不变这个时候就不用继续向上更新高度了；如果节点高度更新就继续向上更新直到更新到root节点为止；

为什么失衡节点只需要一次调节就结束，而不用继续向上检查？

在100节点失衡之前：节点p指向100的left分支高度是 2；在100失衡调节完之后 P的left指向50，这个分支的高度仍然是 2；就是说100失衡并调节之后，对于p节点的left分支来说高度是没有改变的。因此在100失衡并调节之后不用继续向上调节了；

补充：插入后，如何找失平衡节点？

1. 假设插入的新节点为 **w**
2. 从**W** 开始，向上遍历，找到第1 个失衡节点（假设为**Z**节点）
3. 从**W**到**Z**的路径上找到, **Z**的子节点**Y**
4. 从**W**到**Z**的路径上找到, **X**是 **Z**的孙子节点，

例子： 原始二叉树：

[Insertion in an AVL Tree - GeeksforGeeks](https://www.geeksforgeeks.org/insertion-in-an-avl-tree/)

|  |  |
| --- | --- |
| 1. 原始二叉树 | 1. 🡪根据BST插入key=3的节点， 相关于 **W** 2. 🡪递归返回，向上更新节点高度 |
|  |  |
| 1. 从新节点向上遍历，找到第1 个平衡因子 <-1,即，失衡节点： 从 ③ 🡪 ④ 🡪 ⑤ 🡪 ⑩ 2. **⑩**作为左转**支点** 相当于是 **Z** | 1. 在③🡪⑩路径上找 **⑩** 的子节点 **⑤** 2. 在③🡪⑩路径上找 **⑩** 的子节点④ |
|  | **struct** node \***rightRotate**(**struct** node \*y) //Y是支点的节点，也就是不平衡的那个节点。相当于节点 10 {  **struct** node \*x = y->left; //x 相当 5  **struct** node \*T2 = x->right; //T2 相当 8  // Perform rotation  x->right = y;  y->left = T2;  // Update heights  y->height = max(height(y->left),  height(y->right)) + 1;  x->height = max(height(x->left),  height(x->right)) + 1;  // Return new root  **return** x;  } |

|  |  |
| --- | --- |
| **Left Ratation** |  |
|  |  |

|  |
| --- |
| LR型： 找到失衡节点node，将它左子树 node->left 左转 🡪 将它自己右转 |
|  |

|  |
| --- |
| RL型： 找到失衡节点node，将它右子树 node->right 右转 🡪 将它自己左转 |
|  |