BST树，是二叉搜索树：

二叉查找树（Binary Search Tree），（又：[二叉搜索树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%90%9C%E7%B4%A2%E6%A0%91)，二叉排序树）它或者是一棵空树，或者是具有下列性质的[二叉树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91)： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为[二叉排序树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91)。

* 每个点的值d都不同，因此若插入时候某个点的值在树中存在，则插入失败
* 左边的d一定小于当前点，右边点一定大于当前点
* 中序遍历可得到有序序列

# AVL平衡树

是平衡的BST树，插入删除节点时，一旦发现不平衡了马上旋转操作让他平衡。要有变量记录平衡因子,AVL树插入删除原则是，插删后再旋转。

## 查找算法：

同BST树

## 插入结点的算法：

先同普通bst树那样插入，在插入完成后,改变平衡因子，如果发现不平衡，要经过多次旋转使树平衡，这个旋转时间是log(n)级别的，总复杂度是logn(logn的旋转操作)，AVL树的插入删除后调整规则是一样的，后面说

## 删除结点的算法：

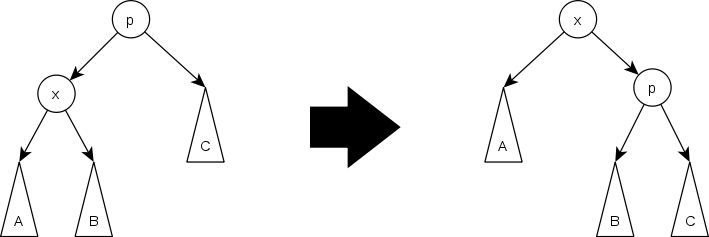
总体和BST一样；但是由于删除使树不平衡，又要转，在被删除的节点开始递归回溯的旋转树，类似插入操作。

总复杂度是logn (logn的旋转操作)

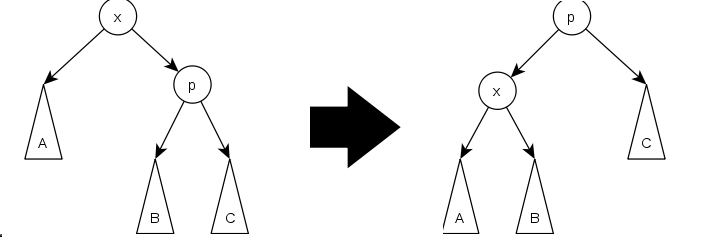
AVL树的插入删除后调整规则是一样的，后面说

## 调整算法：

定义一种基础操作，rot(now,fp)代表以now为支点向fp方向旋转，fp有2种取值，fp==1代表向右旋转，fp==0代表向左旋转：如下所示：



上图是rot(P,1)



上图是rot(X,0)

对于每个节点记录高度h,h代表从子树的最深叶子节点到这个节点的最短路径上的节点数。

不管插入还是删除算法，是一样的调整思路，都是从(插入/删除)的节点起，这是个子下而上的递归的过程，一直检查到根，对于这条路径上的每个节点，发现某个点左右不平衡，就要旋转调整平衡了。所谓” 左右不平衡”,指的是平衡度绝对值大于1

设平衡度difh=左子节点的h - 左子节点的h

平衡度不需要单独记录，通过h计算即可

对于某个点now，检查它的abs(difh)是否大于1，如果abs(difh)>1：看now的较高的儿子的节点,设它是s1

看s1的较高的儿子的节点,设它是s2

设s1是now的fp方向的儿子

又分为如下两种情况：

### s1和s2相对于它们的父亲同方向

则以s1的父亲(就是节点now)为支点向fp^1方向旋转,旋转后不要忘了刷新now和s1的h属性

### s1和s2相对于它们的父亲异方向

则以s2的父亲(就是节点s1)为支点向fp方向旋转,旋转后不要忘了刷新s1和s2的h属性，之后一定变成了情况1，继续执行。