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

# 节点平衡树SBT

是平衡的BST树，怎么保持平衡，对于每个节点，维护子节点个数s ，对于任意一个节点t，满足：

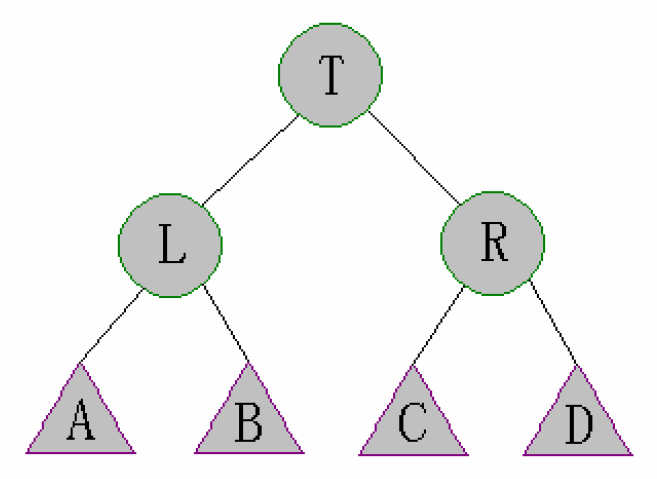
size[right[t]]>=size[left[left[t]]]

size[right[t]]>=size[right[left[t]]]

size[left[t]]>=size[left[right[t]]]

size[left[t]]>=size[right[right[t]]]

如图：



size[R]>=size[A]

size[R]>=size[B]

size[L]>=size[C]

size[L]>=size[D]

## 查找算法：

同BST树

## 插入结点的算法：

先同BST树一样插入,之后进行从插入节点向上走，循环的进行更新size值操作，且更新size后判断四种情况，哪个不满足，就通过旋转调整，且这种操作从插入的节点，向上一直要到根节点。

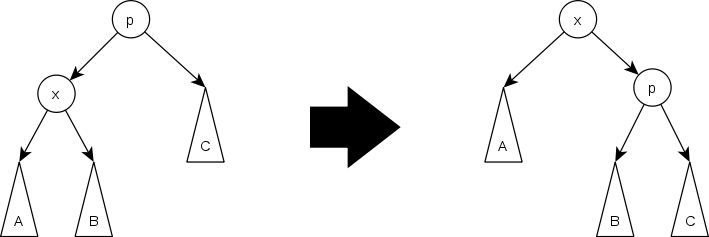
## 删除结点的算法：

BST树那样删除，不必旋转，但要更新size值

## 怎么旋转？

类似AVL树的方式，但是目的不同，这个是为了让4个条件成立，是单旋转

zig型单旋转(rotate(now，左)) ，相反方向的也类似



对于4种不合法的情况

size[R]<size[A] ,rotate(now，左)

size[R]<size[B] rotate(now->左，右) rotate(now，左)

size[L]<size[C] rotate(now，右)

size[L]<size[D] rotate(now->右，左) rotate(now，右)

注意有时候有空节点，判断比较麻烦，应该单写一个比较节点size的函数