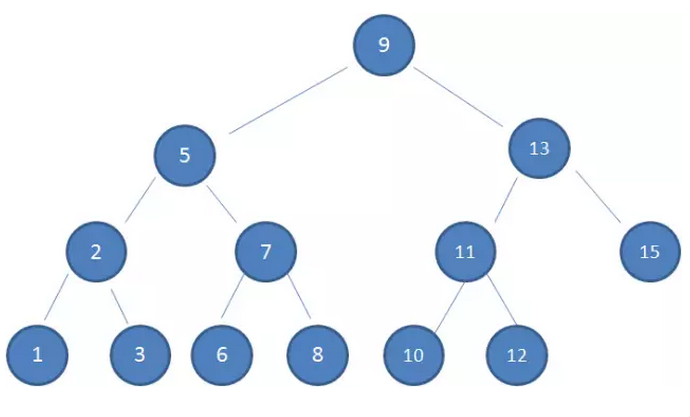
# 1. 二叉查找(搜索)树

## 1.1 定义

1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3.左、右子树也分别为二叉查找树



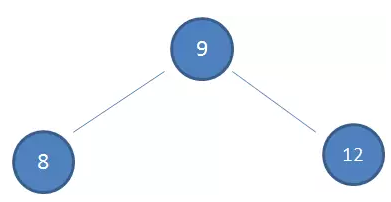
这样的结构可以帮助我们快速查找到自己想要的节点，比如我们希望查找值为12的节点，首先9<12，所以看右子树，13>12，所以看左子树，11<12所以看右子树，成功找到值为12的节点。二叉查找树，其最坏情况下的查找次数等于树的高度，这极大提高了节点的查询效率。当然了，在插入节点时，也是类似的道理。

## 1.2常用操作

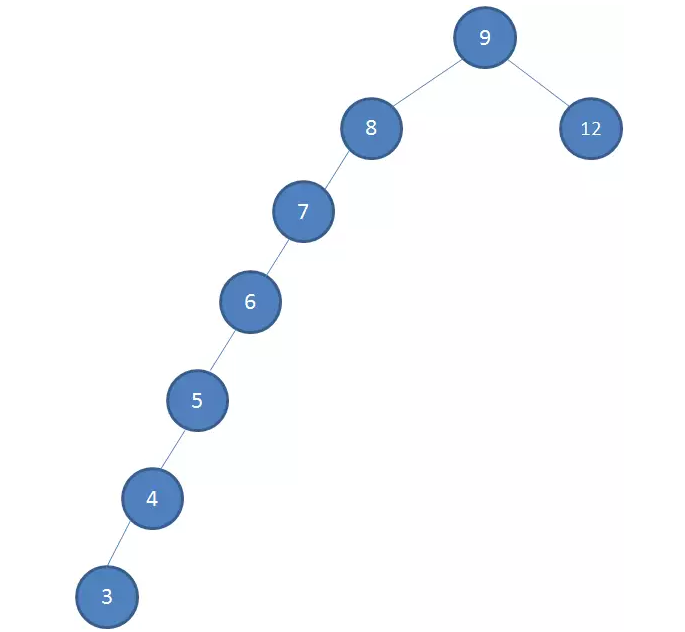
### 1.2.1遍历

树的遍历根据根节点出现的位置顺序一般分为三种：**先序遍历、中序遍历、后序遍历，**由于二叉搜索树的特殊性质，我们在选择遍历方式时，一般选择中序遍历，这样输出的遍历结果就是有序的

但是，在一些极端情况下，二叉查找树可能退化到接近链表的程度，比如下面这种情况



上图是一个8，9，12的二叉搜索树，现在向这棵树中插入3,4,5,6,7，结果如下图：



在这种情况下，二叉查找树的查找效率将大打折扣，为了解决这种极端情况下性能损耗严重的问题，二叉平衡树应运而生

# 2.平衡二叉树

二叉平衡树是一种特殊的二叉查找树，它具有以下特征：

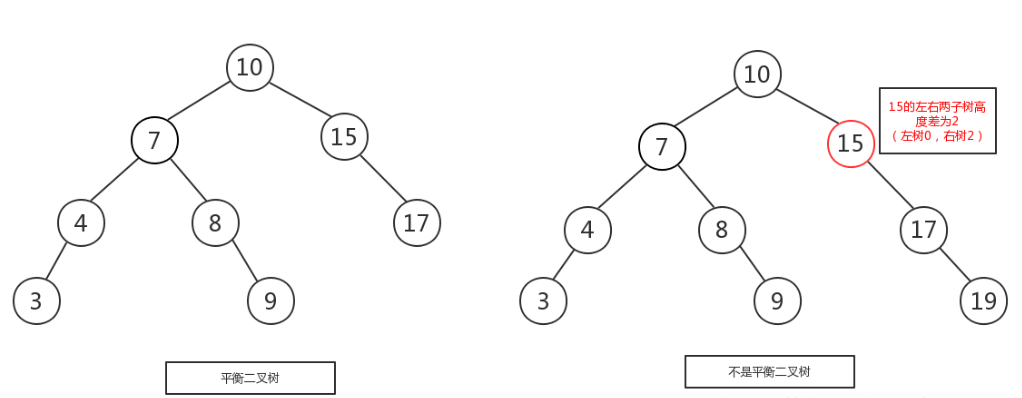
1.左子树上所有结点的值均小于或等于它的根结点的值。

2.右子树上所有结点的值均大于或等于它的根结点的值。

3. 树的左右两边的层级数相差不会大于1

4. 左、右子树也分别为二叉平衡树

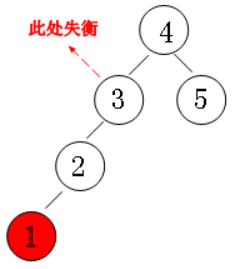
相比于二叉查找树，二叉平衡树(AVL)在左右子树的高度上做了限制，保证二者高度不会相差过大，这会使得二叉平衡树(AVL)的插入和删除操作变得更加复杂(新插入、删除节点都需要重新平衡二叉树)，但会将最好情况和最差情况的时间复杂度都维持在log2(n)



接下来我们来看一看插入结点时，如何平衡二叉平衡树。

平衡二叉树不平衡的情形可以分为以下4种：

1. 左左情况(LL)**左子树的左节点失衡**



在向树种添加1节点时，左子树的左边节点失衡。

解决方法：

1. **右右情况(RR)**
2. **左右情况(LR)**
3. **右左情况(RL)**

# 3 红黑树

首先，红黑树是一棵二叉查找树，它在每一个节点上增加了一个存储域用于表示当前节点的颜色。它满足所有二叉查找树的条件，除此之外，他还要满足以下条件：

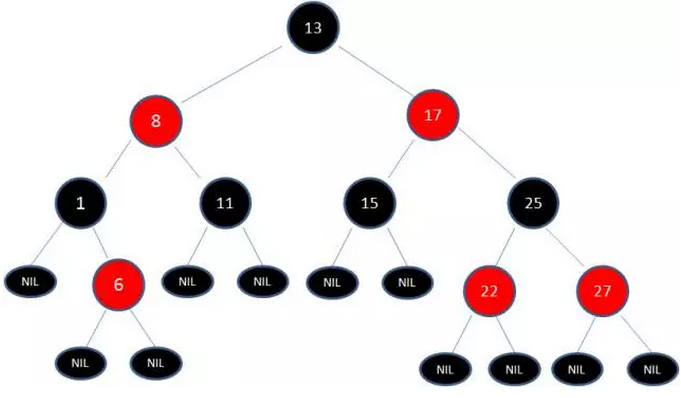
1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

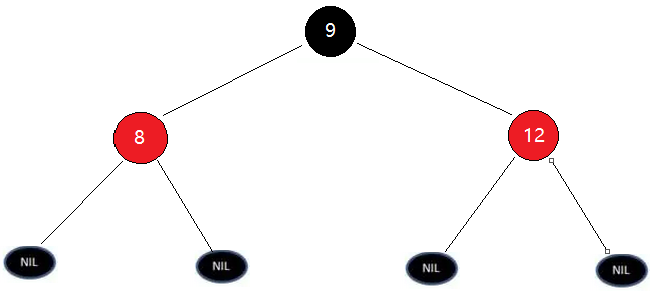
5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。



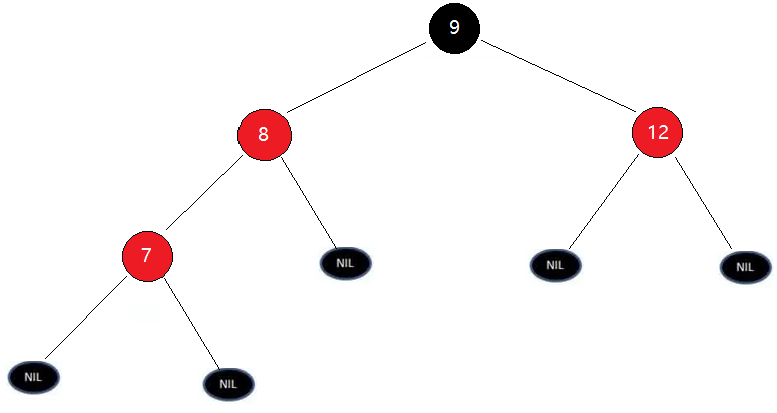
通过第四、五点的约束，我们可以知道，从根节点到叶子节点的所有路径中，最长路径是红黑交替的路径，最短路径是全黑的路径，因此**从根到叶子的所有路径中，没有哪条路径会比别的路径长出2倍。所以我们可以说红黑树是近似平衡的**

再回到我们上面的二叉搜索树的最差情况，

首先是8、9、12，然后依次向其中加入7，6，5，4，3

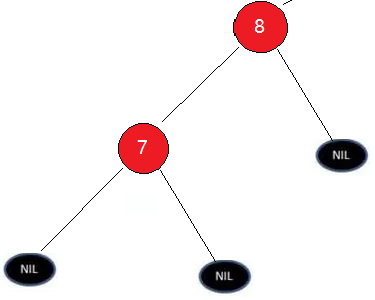
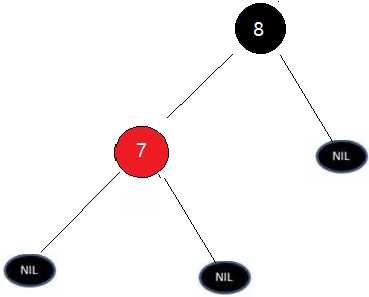


7应当在8的左边，为了保证，9经过8，7到叶子节点只有两个黑色节点(第5条约束)，因此新加入了7应当是红色的，但这又与第4条约束向矛盾(不能有两个相连的红色节点)，因此我们必须对红黑树进行调整。调整的方式有很多，主要分为变色和旋转，而旋转又可以分为左旋转和右旋转(ps:一般来说都是直接把插入的节点染成红色，这样只要满足4即可)

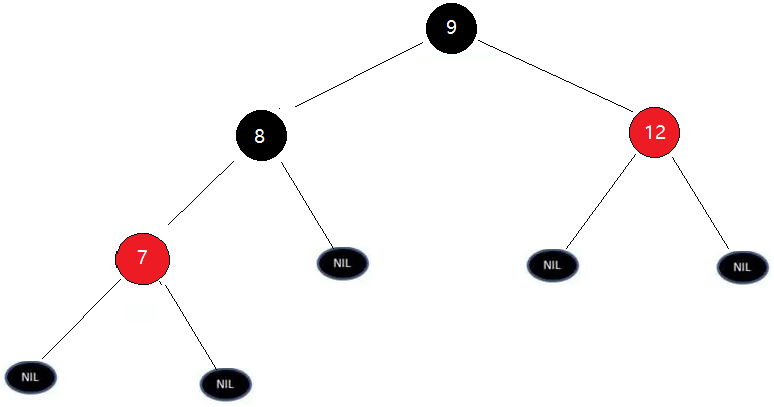
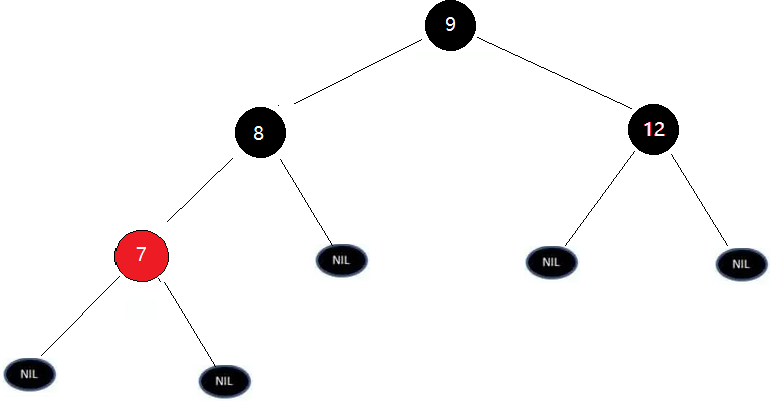


## 3.1 变色

变色是为了维护红黑树的结构，尝试将节点颜色进行转变的一种操作

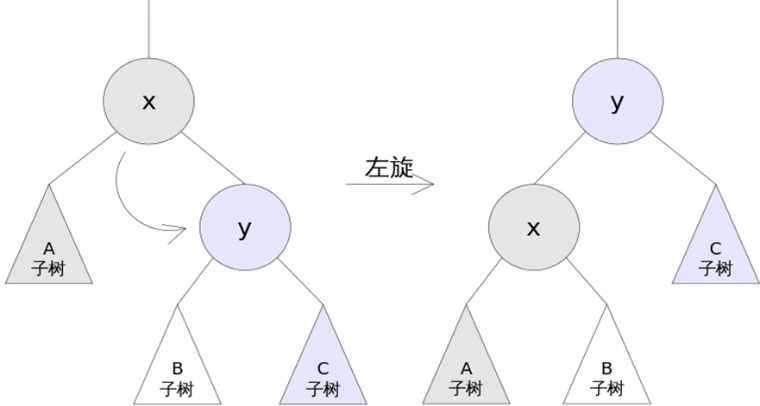
 转变为 

但是这样一来，又违背了第5条约束，从根节点到叶子节点出现了3个黑色节点，出现了连锁变化，因此还需要修改树节点的其他地方

转变为

## 3.2 左旋转

左旋转：逆时针旋转红黑树的两个节点——父节点x和右子节点y，x被y取代，y成为父节点，x成为y的左子节点，同时，y原来的左子节点将变为x的右子节点



## 3.3 右旋转

右旋转：顺时针旋转红黑树的两个节点——父节点x和左子节点y，x被y取代，y成为父节点，x成为y的右子节点，同时，y原来的右子节点将变为x的左子节点

