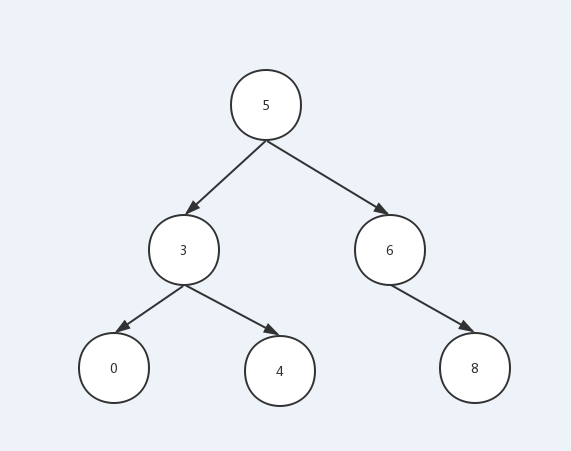
# 二叉搜索树&红黑树

大家来看以下几个结构：下图中的



二叉搜索树又叫二叉查找树，二叉排序树；它具有以下特点：

1.如果它的左子树不为空，则左子树上结点的值都小于根结点。

2.如果它的右子树不为空，则右子树上结点的值都大于根结点。

3.子树同样也要遵循以上两点

为什么又叫做二叉排序树呢？ 二叉树的遍历方式：前 中 后 层次（Mysql）

只要一颗树是二叉搜索树，那么它的中序遍历一定是有序的。左根（输出）右看右边的这颗二叉树，它的中序遍历为：左根右

左 根（输出） 右：0 3 4 5 6 8

二叉查找算法：猜数字，0~100出一个数，让你猜。每次会告诉你猜的结果是大了还是小了。

50 大了-> 51~100

小了 0~50每次一猜就可以排除一半的空间.归并排序，logn；有序的序列

二叉搜索的

5 3 6 0 4 8

插入的时候每次都是和根结点比较。一直要找到它应该插入的位置。

肯定会插在叶子结点。那么其实大家可以看到 插入其实就是查找。

增删改查

删除 是要分三种情况

1.

性能分析

查找 logn

插入：nlogn 有n个数要插入，每一个都要先查找到它的位置 就是logn 合起来就是nlogn,插入单个肯定是logn

删除：

二叉搜索的删除：

分三种情况：

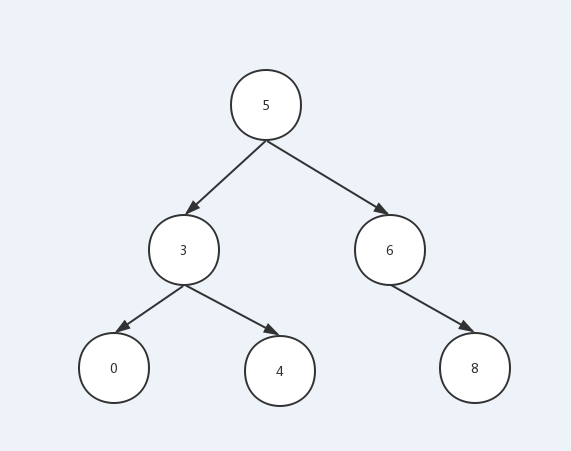
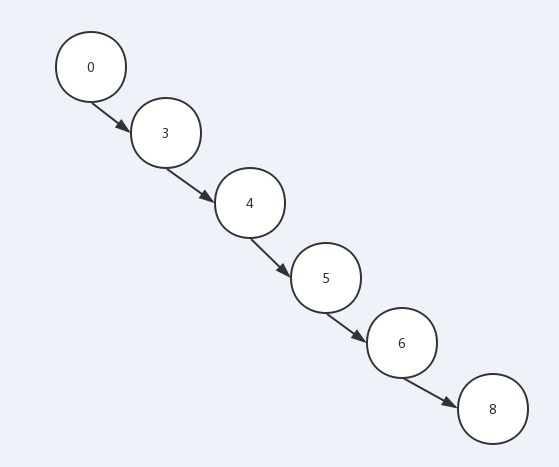
1.要删除的结点是叶子结点 ：直接删除O(1)

2.要删除的结点只有一个子树（左或者右）：删除后把子树提到原节点O(1)

3.要删除的结点有两颗子树：找后继结点，而且后继结点的左子树为空的（这样可以把这个后继节点直接放到要删除的节点上替换，后继节点的子树直接替换后继节点）

logn

二叉搜索树有哪些应用呢



O(n) for(int i = 0 ; i < n ; i++){}弊端退化成链表了

既然是搜索树，那么肯定就是用在了查找上。

我们来分析它的查找时间复杂度：

看上边两颗二叉搜索树：

他们的性能：

查找时间复杂度其实就是树的深度

O（n）表示时间复杂度：查找N次。循环了N遍

为什么（退化了）？怎么解决呢？

不要变成一个链条一样

AVL树：

红黑树和平衡二叉树：AVL属于实验室状态的，红黑树才是我们项目中用的。

红黑树：

通过上面两个图我们发现，二叉树的结构就决定了其搜索的性能，那么我们应该怎么优化呢？

因此就有了AVL树和红黑树

AVL树：平衡二叉树，它的左右子树高度之差不超过1

这样确实可以避免一条直线型的结构，但还不是我们最理想的

可以认为是理想状态，实验室。红黑树

为什么呢？

通过性能综合考虑选用：

红黑树的性质（重点）：

1.每个结点不是红色就是黑色，新增节点为红色

2.不可能有连在一起的红色结点（黑色的可以），每个叶子节点都是黑色的空节点（NIL），也就是说，叶子节点不存储数据

3.根结点都是黑色root

4.每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点；

为了满足红黑树的性质，因此出现了旋转：

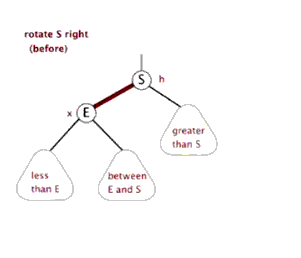
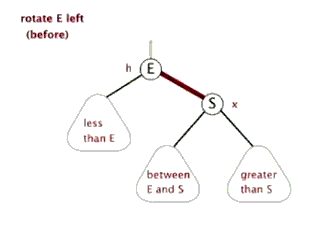
那么红黑树有几种变换呢？3

1.改变颜色:最简单 红变黑 黑变红

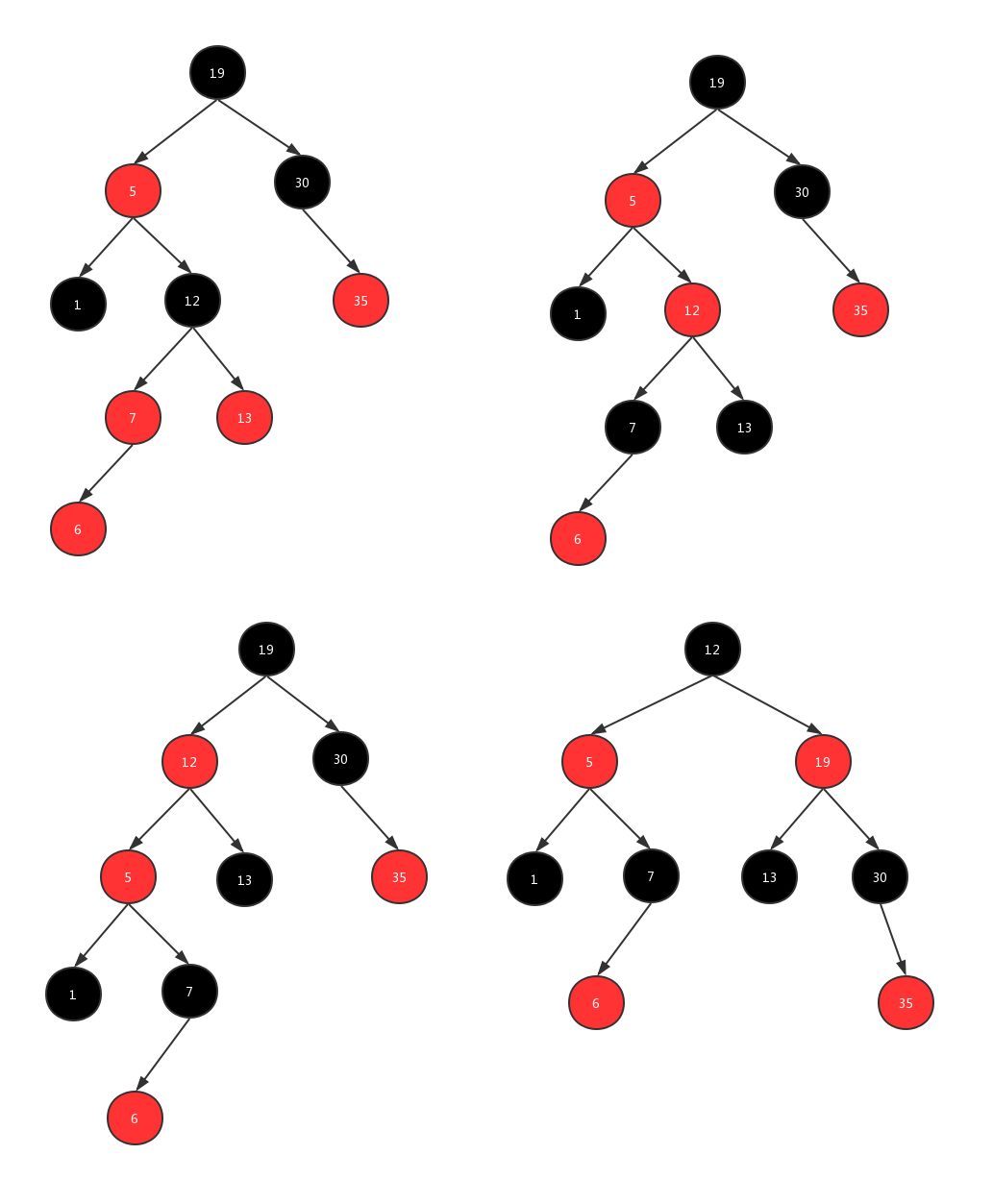
2.左旋：针对于点旋，但是点上面的子树也要跟着转。指针

3.右旋：

那么我们又该如何选择以上三种方式呢？



插入元素 6 的过程如下：



插入的时候旋转和颜色变换规则：

1.变颜色的情况：当前结点的父亲是红色，且它的祖父结点的另一个子结点也是红色。（叔叔结点）：

（1）把父节点设为黑色

（2）把叔叔也设为黑色

（3）把祖父也就是父亲的父亲设为红色（爷爷）

（4）把指针定义到祖父结点(爷爷)设为当前要操作的.

2.左旋：当前父结点是红色，叔叔是黑色的时候，且当前的结点是右子树。左旋

以父结点作为左旋。指针变换到父亲结点

3.右旋：当前父结点是红色，叔叔是黑色的时候，且当前的结点是左子树。右旋

（1）把父结点变为黑色

（2）把祖父结点变为红色 （爷爷）

（3）以祖父结点旋转（爷爷）

红黑树的删除：

原理类似二叉搜索树 详细过程见附件。。。。

红黑树的性能分析

插入 近似：nlogn

查找 logn

删除：近似logn