## 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

## 红黑树：

<https://www.cnblogs.com/yyxt/p/4983967.html>

<https://www.cnblogs.com/qingergege/p/7351659.html>

我是从以上两个博客学来，代码经过精心压缩，显得不是很多。

红黑树是平衡的BST树，具备以下特征：

* **每个节点要么是黑色，要么是红色，根节点是黑色，空节点是黑色(叶子结点的子节点)。**
* **如果一个节点是红色的，则它的子节点必须都是黑色的，反过来一个节点是黑色，它的子节点不一定是红色,这个也很重要，插入删除后调整树时候用。**
* **从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点**

这条特征确保了，没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树

在随机数据下，红黑树的时间复杂度虽然和其他平衡树同样量级，但是它的常数较小，效率是所有平衡树最高的，可实现也是很麻烦的。

定义：now代表”当前节点”，root代表根,p代表now的父亲，b代表now的兄弟

## 查找算法：

同BST树，复杂度是logn

## 插入结点的算法：

总体过程同BST树那样，插入完成之后，若插入的节点就是root把它涂成黑色。否则涂成红色，显然这么做后，可能会违反性质2，要进行修正，分类讨论

有4种情况：

### 1.父节点是黑色：

什么都不做，且递归结束，因为此时树本来就合法，不违反性质2

### 2.父节点是红色，[叔叔节点](#_关于旋转操作和一些概念解释)是红色：

(1) 将“父节点”设为黑色，如果父节点是root，就到此为止，说明now没有祖父了。  
(2) 将“叔叔节点”设为黑色。  
(3) 将“祖父节点”设为“红色”，如果祖父是root,就还是黑色。  
(4) 将“祖父节点”变成 “当前节点”，递归的进行操作，再进行分类

正确性在哪，由于此时(1)为了满足性质2，(2)和(3)是为了确保性质3

红黑树插入操作只有此操作具有连续性，执行一次后可能还需要调整，直到：到达根为止或者不满足条件为止。且有可能变成情况3或者4

### 3.父节点是红色，[叔叔节点](#_关于旋转操作和一些概念解释)是黑色，now和父亲异方向：

以now的父节点为支点旋,使得变成同方向，具体来说，若now是右儿子而p是左儿子，就要以p左旋；若now是左儿子而p是右儿子，就要以p右旋

,转换成了情况4

### 4.父节点是红色，[叔叔节点](#_关于旋转操作和一些概念解释)是黑色，now和父亲同方向：

父节点变为黑色，祖父节点变为红色，在祖父节点为支点向^f方向旋转，具体来说，若p是左儿子，就要以p的父亲为支点右旋；若p是右儿子，就要以p的父亲为支点左旋，这个操作完成后，一定会得到合法的红黑树

### 总结：

注意判断根，操作2是连续多次的，1,3,4最多执行一次。且操作2执行若干次可能转变成3或者4，而操作4执行完毕就一定是合法红黑树，道理可以根据红黑树性质想一下，比较容易。

## 删除算法概述：

总体和BST一样；但是由于删除后使树不平衡，又要修正和分类讨论，类似的，修正是个自下而上的递归操作，要不断进行分类，然后修改，把now上移，到达root为止。

到这里复习一下**普通bst树删除节点**：

1.若删除的节点是叶子，直接删除什么都不做;

2.若删除的节点只有一个儿子，删除后，把这个点的父子节点相连;

3.若删除的节点有左右两个儿子，即左右子树，应该找左右子树中的最值节点(最值节点就是左子树最大的节点 或 右子树最小的节点)把它记为v，按照bst树性质,v的儿子一定少于2(有一个儿子或者没有)，v交换和now交换信息，删除v，此时一定是前面2种去情况。这样就相当于删除了now。

**红黑树删除操作如下：**

设now是父亲的f方向儿子，用^f来表示f的相反方向。这样方便描述，不用分左右

红黑树的删除操作和普通bst删除稍微不同，红黑树应该分三步：1先寻找，再调整，最后删除。

### 第一步：

若左右儿子至少一个是空，进行下一步；否则用普通bst删除情况3那样,去寻找替代的节点，把待删除的节点变成那个点。

### 第二步：

分类讨论，进行调整，为了确保删除now之后，这棵树满足红黑树定义，这里now可能会移动，变得不再是被删除的点，所以第三步把待删除的点称为D

### 第三步：

由于第一步，D一定左右至少一个儿子是空；第二部有确保删除D后还是颗红黑树；所以现在直接删除D，如果D有一个儿子就直接把它的父亲和它的儿子相连，如果D没有儿子，直接删除就好。

## 删除时树的情况：

### now是红色节点，或者是root

什么都不做，红黑树满足性质，调整完毕

### now是黑色节点，儿子一个是红色，一个是空

很简单，把那个红色儿子变成黑色，红黑树满足性质，调整完毕

### now是黑色节点，兄弟存在且是红色

此时，父亲一定是黑色(否则就不满足红黑树性质2)

现在把父亲和兄弟的颜色互换(也就是父亲染成红色,兄弟变成黑色)，之后以父亲为支点，向f方向旋转，这么做，now的兄弟一定变成了黑色，父亲一定是红色，就变成了情况4,5,6 ，继续处理

### now是黑色节点，兄弟是黑色，兄弟的^f方向儿子是红色。

此时只是知道：^f方向儿子是红色，

但不知道：f方向儿子颜色是啥，父亲节点颜色是啥

先把兄弟的^f方向儿子染成黑色；

此时不论父亲节点是什么颜色，都把父亲和兄弟节点颜色交换，之后以父亲为支点，向f方向旋转，调整完毕

### now是黑色节点，兄弟是黑色，兄弟的f方向儿子是红色。

以兄弟节点为支点向^f方向旋转，这样变成了情况4，继续处理

### now是黑色节点，兄弟是黑色，兄弟的两儿子都是黑色，父亲节点是红色。

把父亲改成黑色，把兄弟涂成红色，这样P的所有路径黑节点数目不变且相同,

使得红黑树满足性质3，调整完毕

### now是黑色节点，兄弟是黑色，兄弟的两儿子都是黑色，父亲节点也是黑色。

先把兄弟涂成红色，由于把兄弟涂成红色又要删除now，让 P的所有路径黑节点数目相同了，但此时父亲本来就是黑色的，P的所有路径黑节点数目比原先少了一个，怎么办？此时”把兄弟涂成红色”这个操作照常不变，还要向上找，以P为now，继续分情况讨论(实际上此时2情况一定不会出现了)

### 总结：

* 红黑树的删除节点虽然麻烦，但快，只有情况7是需要往上递归，但情况7出现频率很低，在平衡二叉树里，旋转是常数复杂度操作里比较慢的，红黑树最坏需要3次旋转可以达到平衡(相比AVL树最多旋转logh次)，删除效率高是红黑树最大优势。

## 关于旋转操作和一些概念解释

**祖父节点：当前节点父节点的父节点**

**叔叔节点：当前节点祖父的另一个儿子**