二叉排序树

树形数据结构：定义一种性质，维护一种性质

学习重点：如何维护这种性质的

- 名称：二叉排序树，二叉查找树
- 性质：左子树<根节点，右子树>根节点

数据结构的本质是数据定义+数据操作

```
而数据操作是维护数据的性质的
```

- 插入：每插入一个节点，肯定是到叶子节点上
- 删除：
    1. 删除叶子节点
    2. 删除出度为1的节点
    3. 删除出度为2的节点

要实现删除操作，要引入二叉查找树中前驱和后继的概念

- 前驱：值比它小的节点中最大的一个节点
    - 左子树里最靠右的
- 后继：值比它大的节点中最小的一个

所有节点的前驱都为左子树的最右节点

```
而此节点度一定为0或1
// 因为是最右节点了 则一定没有右孩子
```

- 二叉排序树与快速排序的关系

中序遍历是有序的，根节点像是快速排序里的基准值，

快排里找基准值的过程即找二叉排序树的过程

查找过程最好的时间复杂度为nlogn，最坏为 $\frac{n^2}{2}$ (排序树为一条链时退化)

思考

快速排序本质为何是二叉排序树

```
为了锻炼思维方式，思考两者联系，连成线
```

平衡二叉排序树

为了防止退化成一个链表

AVL树

任意节点子树的高度差不超过1

- BS树与AVL树的比较

相同高度的AVL树和BS树包含节点的上限相同，而下限AVL较大，因此AVL树可以保证logn的查找复杂度

AVL树提升的是下限

就像我们接受的教育，提升的是下限，而定义上限的是我们自己

旋转思维不是AVL树的专属，而是所有需要维护树的这种性质的专属

平衡调整是从底层到顶层，因此找到的是第一个失衡的节点进行旋转

左旋

- 重点是调整原来的左子树

失衡类型

LL型和RR型对称，LR和RL对称

LL：大右旋

LR：小左旋后变为LL型再大右旋

核心：旋转及其证明，不能靠记！！

SB树

通过节点数量来控制平衡

调整同样通过左旋和右旋来完成

红黑树

五个条件

1. 每个节点非黑即红
2. 根节点是黑色
3. 叶节点（NIL）是黑色
4. 如果一个节点是红色，则它的两个子节点都是黑色的
5. 从根节点出发到所有叶节点路径上，黑色节点数量相同（关键）

调整策略

1. 插入调整站在`祖父节点`看
2. 删除调整站在`父节点`看
3. 插入和删除的情况一共五种

2019-02-17 新年后第一次课

- 应该记住的：红黑树的平衡条件

- 调整策略不是记住的，而是自行推导得到的
- 正常红黑树中有一个虚拟叶节点NIL，黑色，即在红黑树中看不到。
- 最后一个条件导致一个推论：

  最长一条路径是最短那条路径长度的两倍
- 红黑树是两种信息的叠加，`数值` 和 `颜色` 没有关系。数值用来维护排序的性质，颜色用来维持平衡的性质
- 调整是维护红黑树的性质，即不改变路径上黑色节点的数目

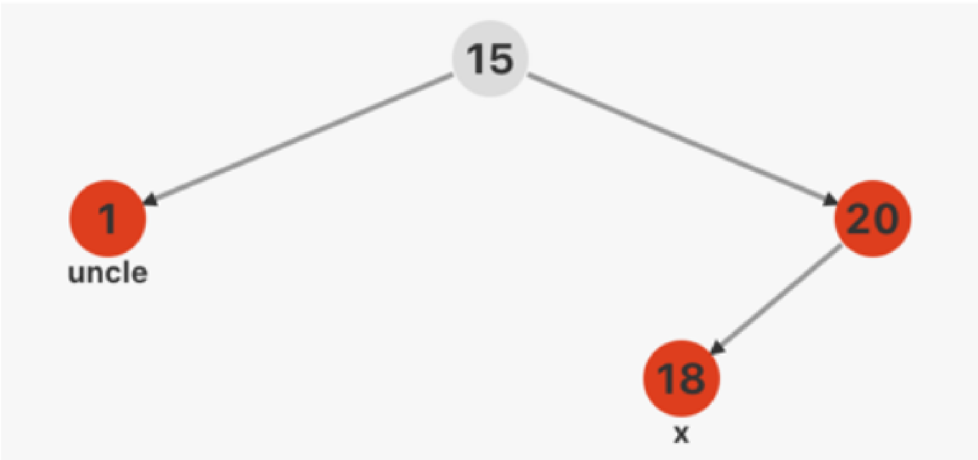
插入调整--站在祖父节点看

`调整后要保持每个路径上黑色节点数目不变`

围绕着消除两个连续的红色节点

- 新节点的颜色一定是红色
- 当父节点为黑色时不必调整，因为不会增加路径上黑色节点的数目

情况一：祖父节点另一个孩子也为红色



处理办法：1和20修改成黑色，15修改成红色（所谓的红色上顶）

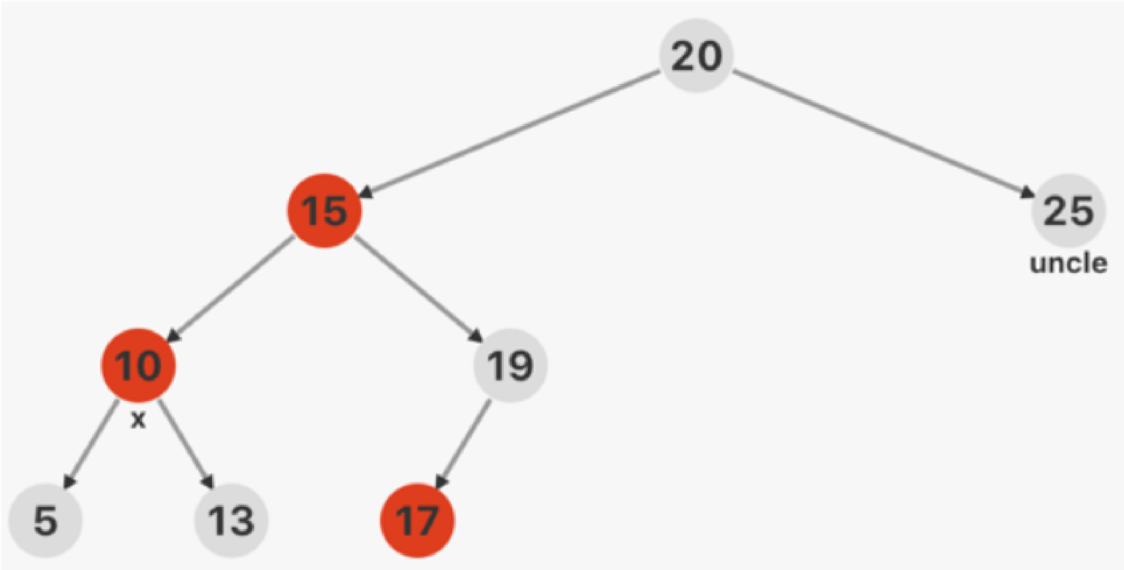
祖父节点的两个孩子都为红色

`判断发生冲突时要站在祖父节点看，回溯到父节点看发生冲突不去处理，到祖父节点才去处理冲突`

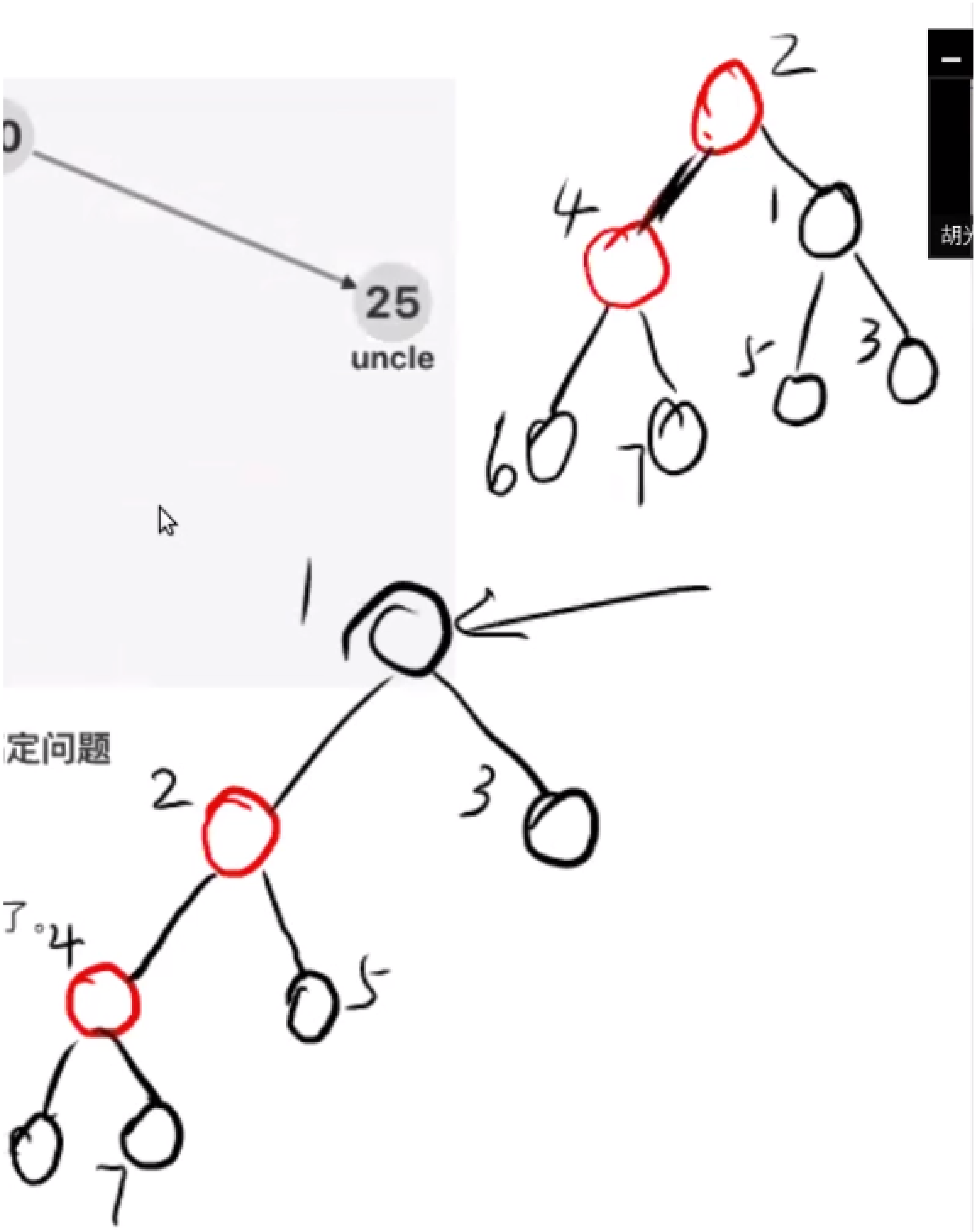
假设LL为祖父节点下的左孩子和左孩子的左孩子为红色

插入情况时，站在当前节点看，只要发生了冲突，当前节点一定为黑色

情况二：祖父节点另一个孩子为黑色



处理办法：大右（左）旋，20调整成红色，15调整成黑色，即可搞定问题

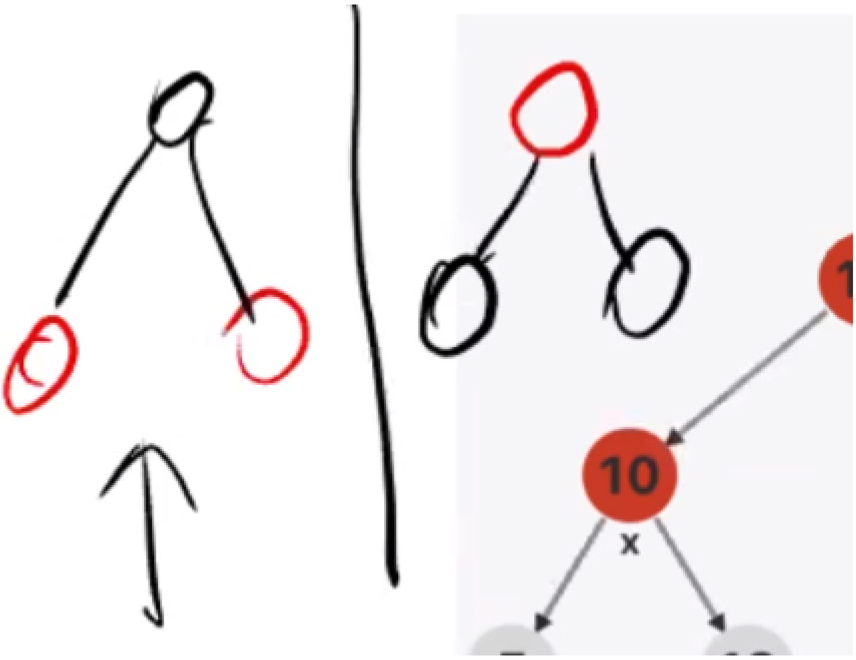


LL型：一个大右旋，戴顶小帽子即可

小帽子要保证给每条路径提供的黑色节点数相等

- 因此会导致红黑树每个人的写法不同

LR型：小左旋后变成了LL型，再大右旋，之后再调整帽子



两种调整情况都能满足功能，左边叫做`红色下沉`，右边叫做`红色上顶`。

删除调整--站在父节点看

`调整后要保持路径上黑色节点数目不变`

- 删除可分为6种情况：删除度为0的红色/黑色、删除度为1的红色/黑色、删除度为2的红色/黑色节点

1. 删除度为2的红色/黑色节点：扫前驱或后继，数值交换后删除前驱或后继即可，颜色不必交换（交换可能破坏原本的平衡）
2. 删除度为0的红色节点：可直接删除（不影响平衡）
3. 删除度为1的红色节点：可直接将它的孩子连接到它的父亲（都为黑色，因此不影响平衡）

因此接下来就只需要考虑删除度为1的黑色节点和度为0的黑色节点了，可以放在一起考虑，即以下几种情况。

度为0的黑色节点的父节点下一定有两个子孩子，兄弟节点的颜色不一定

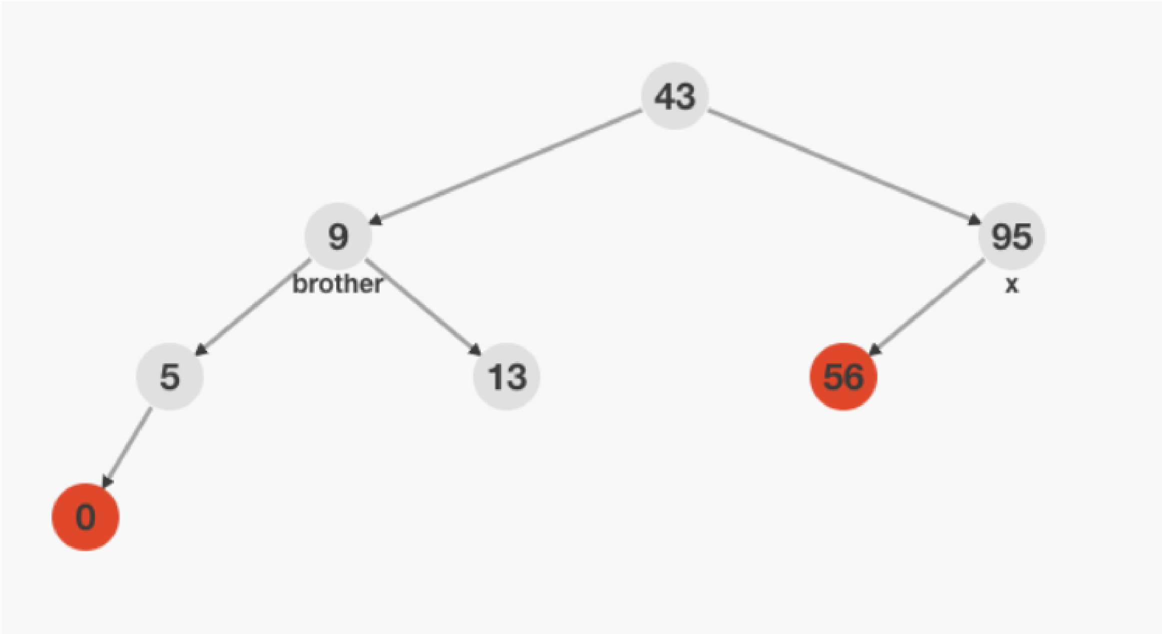
删除时一个重要概念：双重黑

围绕着消除双重黑

- 失衡：站在父节点看，存在双重黑的孩子节点
  - 父节点本身为双重黑不算失衡，站在父节点的父节点看，这种才算失衡
- 调整：消除孩子节点的双重黑即可

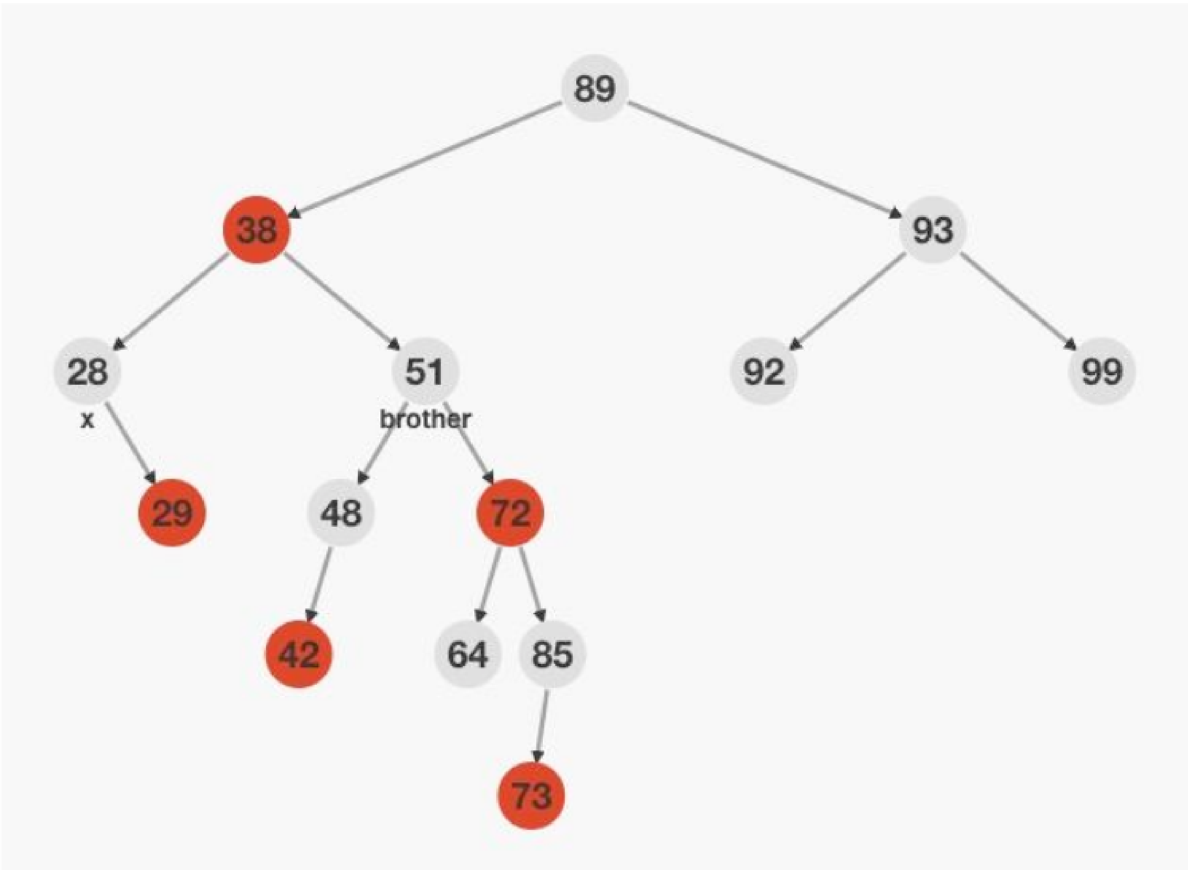
当前节点双重黑，兄弟节点是黑色的的情况

情况一：兄弟节点是黑色，兄弟两个子节点也是黑色



处理办法：brother 调整为红色，x 减少一重黑色，father 增加一重黑色

情况三：双重黑的兄弟在右侧且兄弟的右孩子为红色



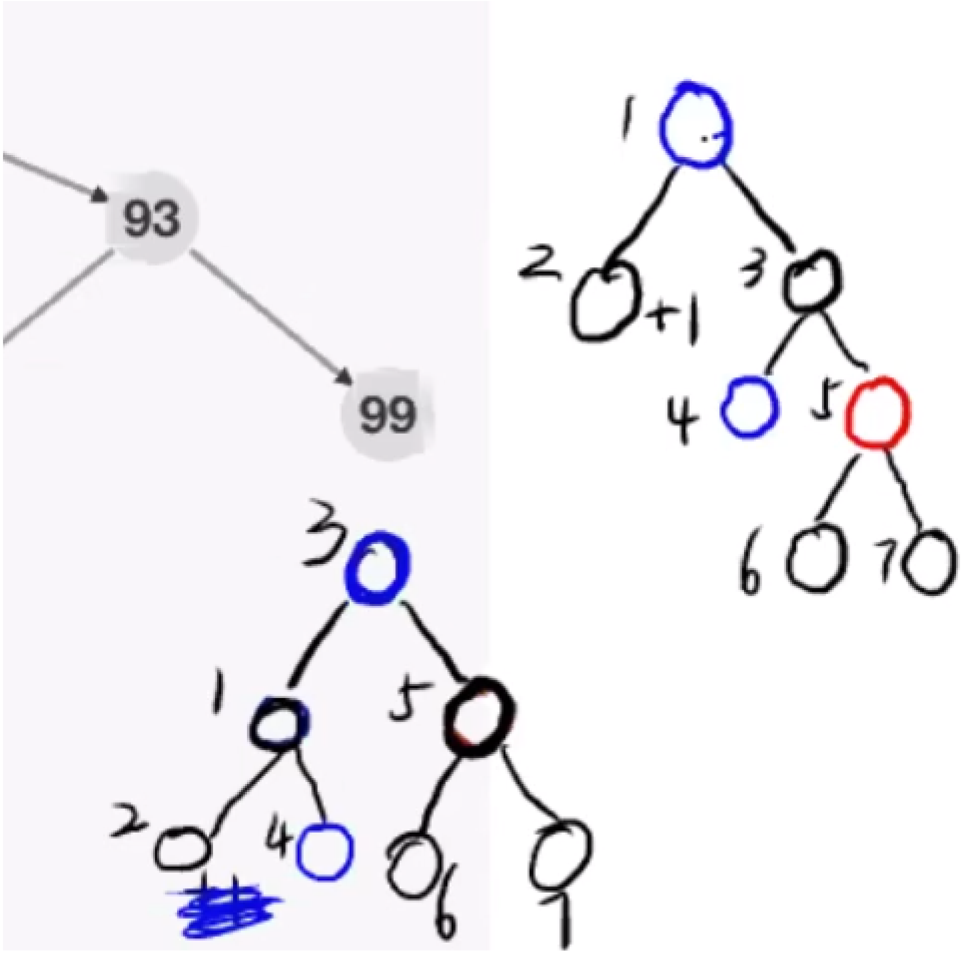
处理办法：father 左（右）旋，由于无法确定48的颜色，所以38改成黑色，51改成38的颜色，x 减少一重黑色，72改成黑色

RR型：双重黑的兄弟在右侧且兄弟的右孩子为红色

```
根节点(38)的颜色不能确定，右孩子的左孩子(48)的颜色也不能确定
```

如下为一种RR型的情况：不能确定的节点用蓝色表示

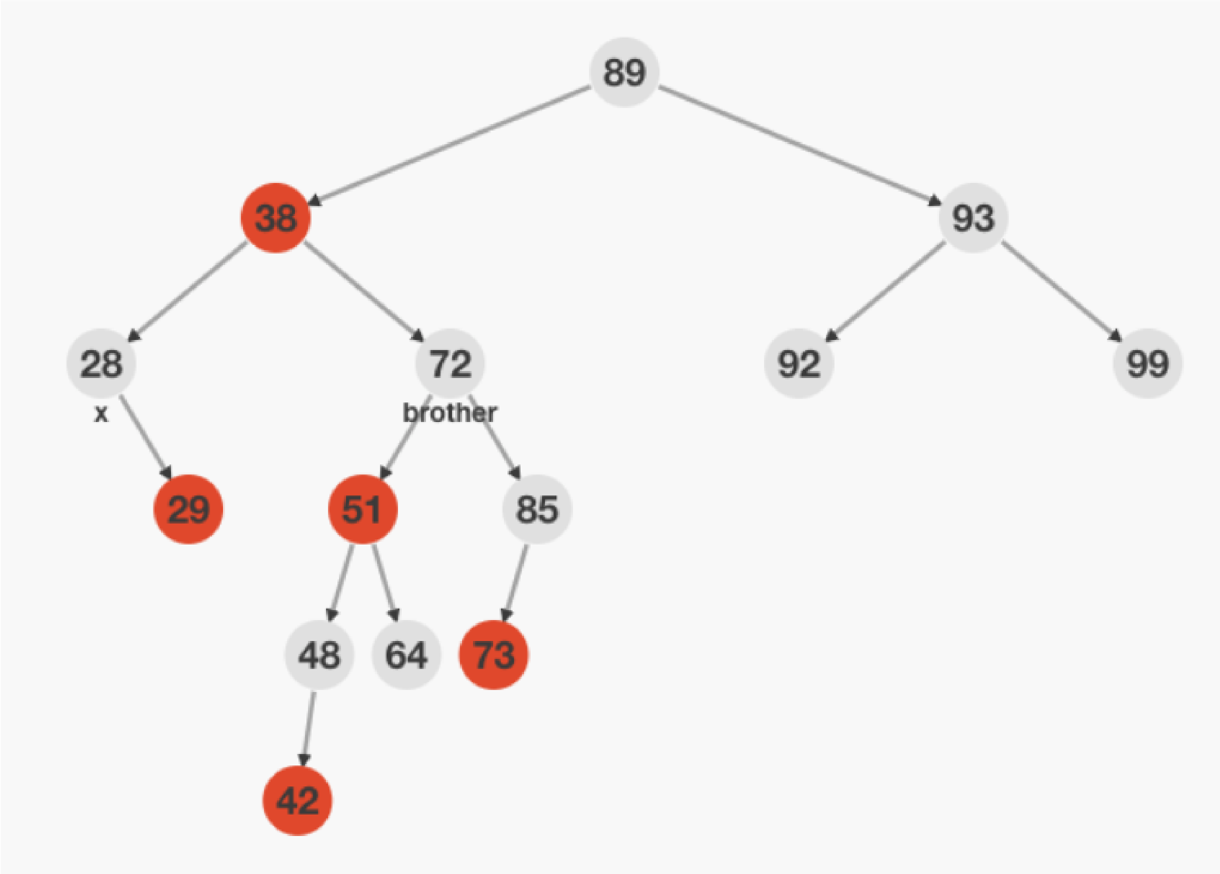
- 调整策略：先大左旋，根节点两个孩子变为黑色，根节点改为原来根节点的颜色，双重黑去掉




```
推理过程：4号节点颜色节点不确定，又因为要接在1下面，因此1号节点一定为黑色；原本4号节点往上数有一个黑色节点，而现在有两个，因此将3号节点颜色改成原来1号节点的颜色（即不确定）；这时对于左子树，4在的子树平衡了，2所在的子树多了一个黑色，因此将双重黑减少一个黑色；此时看右子树，不平衡，因为3号节点不确定，因此把5号节点改成黑色
即整个过程为：左旋后，1.根节点为原来根节点的颜色，2.双重黑减少一重黑色，3.根节点两颗子树都改为黑色
```

LL型同理

情况二：双重黑的兄弟的左孩子为红色

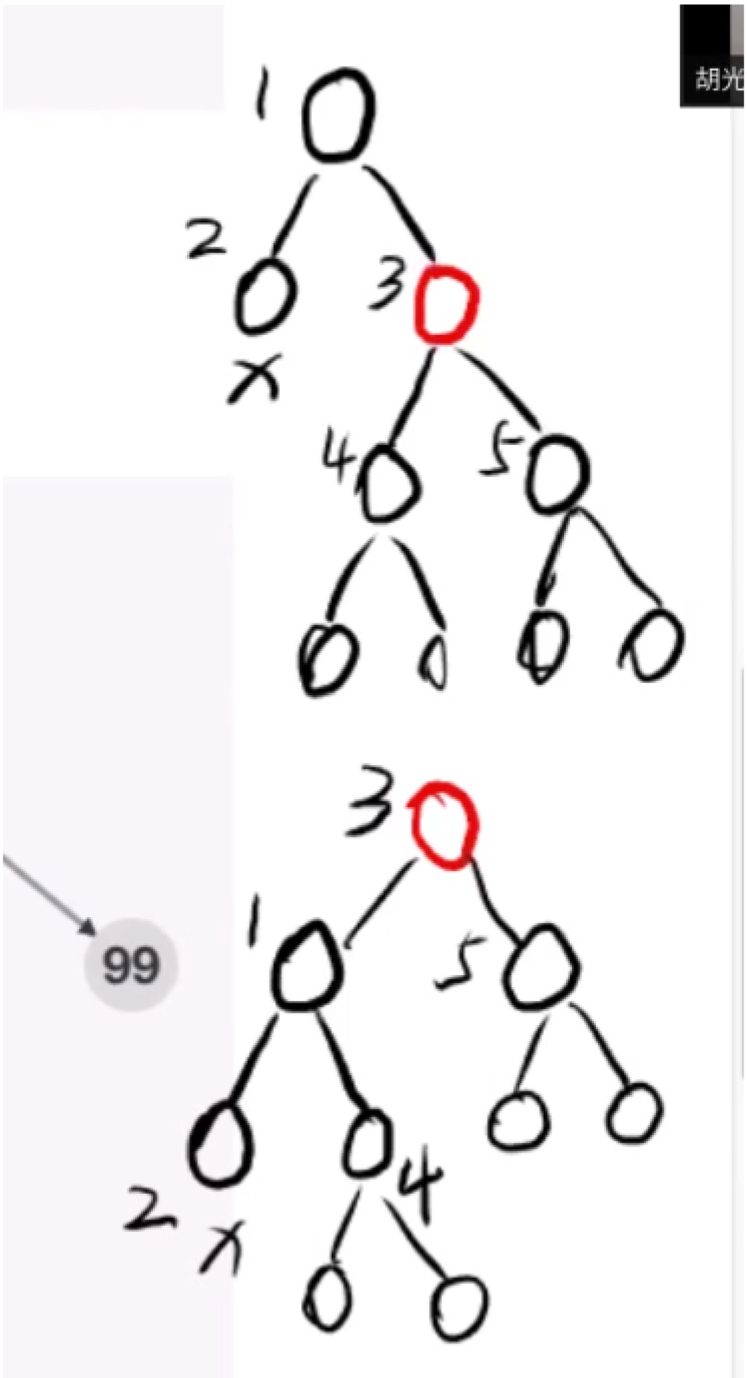


处理办法：brother 右（左）旋，51变黑，72变红，转成处理情况三

```
能确定颜色的：28、72、51、85、48、64
85一定为黑，否则就变成了RR型。51一定为红色，因为是RL型
```

RL型：小右旋，改颜色，根节点改为黑色，右孩子改为红色，即转换成了情况三RR型

兄弟节点是红色的情况



旋转让x有一个新的兄弟节点（黑色）,再递归到孩子节点去进行删除调整（因为此时双重黑的兄弟节点已经变成了黑色）

> 推理过程：进行一个大左旋后，右子树路径上少了个黑色，因此把根节点3改为黑色；改完后发现，左边路径上多了一个黑色，又因为1号节点肯定为黑色，因此将1号节点改为红色，之后递归到根节点的左孩子去调整。就转化成了情况二和情况三
> 总结：大左旋后，3改成黑色，1改成红色（和情况二的小旋转改颜色过程相似）
>
> 处在右面则一个大左旋， 处在左面则一个大右旋

插入/删除调整总结

1. 删除操作一共的情况总数8种

兄弟节点是黑色时：6种情况：子孩子全黑、子孩子红色的在右侧、子孩子红色的在左侧，又因为兄弟节点可以位于左边或右边，因此共6种。

兄弟节点是红色时：2种情况，在左侧，大右旋。在右侧，大左旋。

2. 插入调整又有8种具体的情况

因此红黑树具体一共有16种情况

性能

相对于AVL树，红黑树的旋转次数少，调整次数少

其他

STL中的set、map均基于红黑树

因此set表现为不能有重复元素，且插入set、map后输出，元素会自动排序

代码演示

一些确定性的信息在程序中不必展现，会减少代码量，比如节点的颜色

插入测试：5 1 0 3 6 4 8 9