

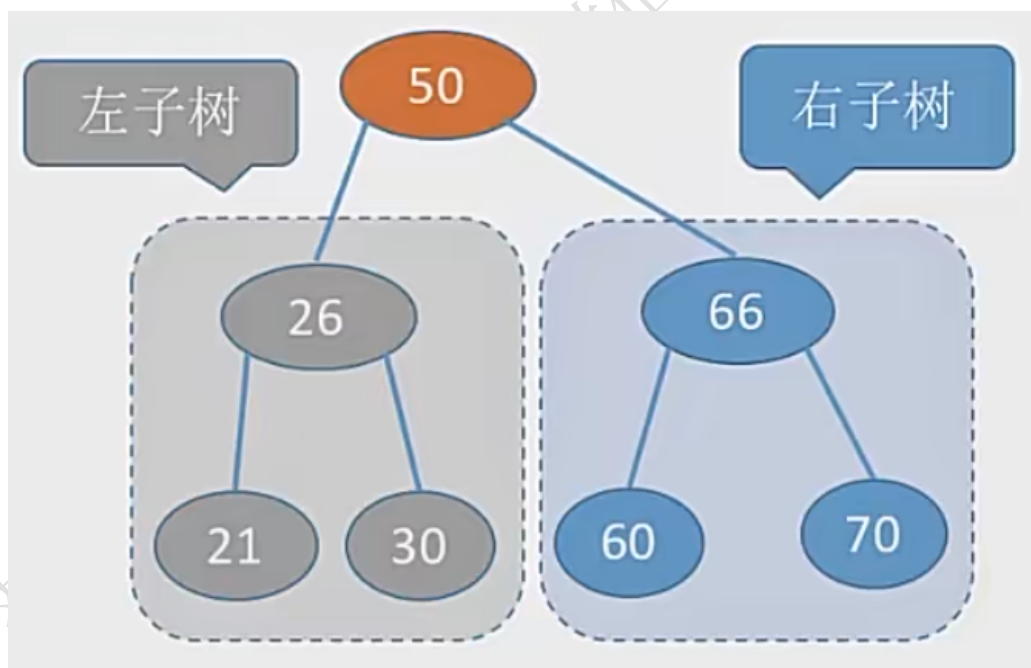
Lecture 26: 树形查找

26.1 二叉排序树

26.1.1 基本概念

基本概念

- 定义：
 - 二叉排序树，又称二叉查找树（BST，Binary Search Tree）；
 - 一棵二叉树或者是空二叉树，或者是具有如下性质的二叉树：
 - 左子树上所有结点的关键字均小于根结点的关键字；
 - 右子树上所有结点的关键字均大于根结点的关键字；
 - 左子树和右子树又各是一棵二叉排序树；
- 特性：
 - 1. 左子树结点值 < 根结点值 < 右子树结点值；
 - 2. 进行中序遍历，可以得到一个递增的有序序列；
- 图示：



26.1.2 查找

二叉排序树的查找

- 概念：
 - 若树非空，目标值与根结点的值比较；
 - 若相等，则查找成功；
 - 若小于根结点，则在左子树上查找，否则在右子树上查找；
 - 查找成功，返回结点指针；
 - 查找失败返回 NULL

- 性能:
 - 非递归实现:
 - 最坏空间复杂度 $O(1)$
 - 递归实现:
 - 最坏空间复杂度 $O(h)$

非递归实现

```
//二叉排序树结点
typedef struct BSTNode{
    int key;
    struct BSTNode *lchild,*rchild;
}BSTNode,*BSTree;
```

```
//在二叉排序树中查找值为key的结点
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->data){ //若树空或等于根结点值，则结束循环
        if(key<T->data)
            T=T->lchild; //小于，则在左子树上查找
        else
            T=T->rchild; //大于，则在右子树上查找
    }
    return T;
}
//最坏空间复杂度O(1)
```

递归实现

```
//在二叉排序树中查找值为key的结点(递归实现)
BSTNode *BSTSearch(BSTree T,int key){
    if(T==NULL)
        return NULL; //查找失败
    if(key==T->key)
        return T; //查找成功
    else if(key<T->key)
        return BSTSearch(t->lchild,key); //在左子树中找
    else
        return BSTSearch(t->rchild,key); //在右子树中找
}
```

26.1.3 插入

概念

- 实现:
 - 若原二叉排序树为空，则直接插入结点;

- 否则，若关键字 k 小于根结点值，则插入到左子树，若关键字 k 大于根结点值，则插入到右子树；

代码实现：递归实现

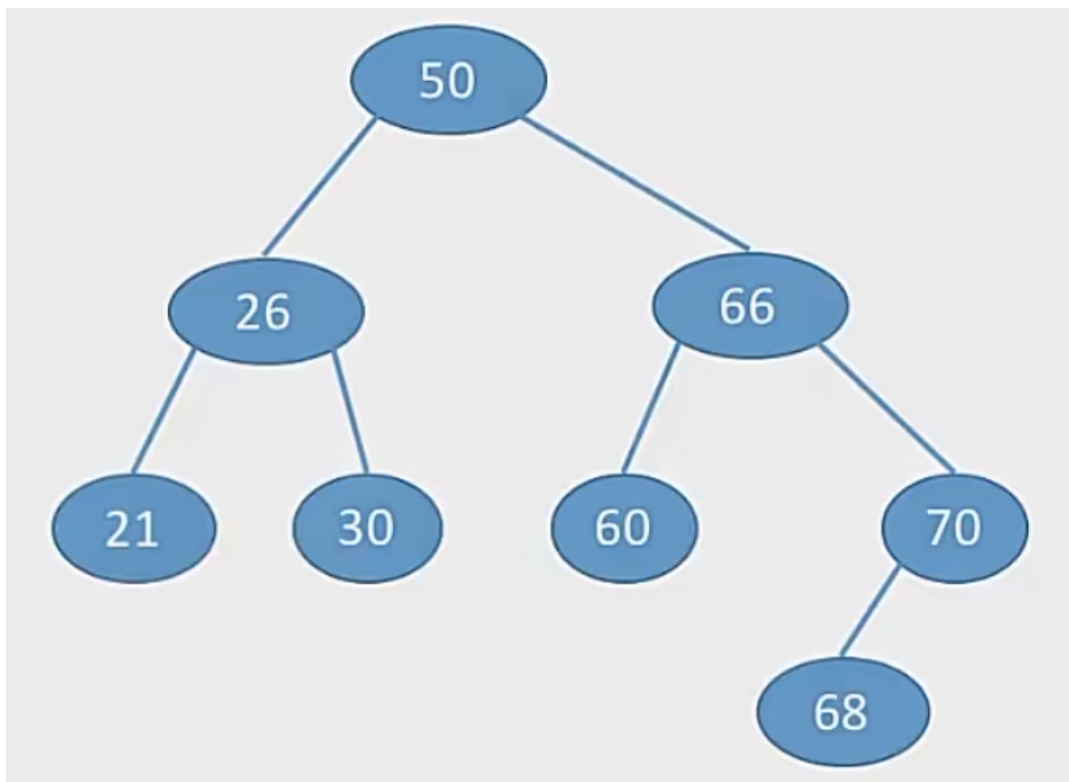
- 注意：
 - 树中存在相同关键字的结点，插入失败

```
//在二叉排序树插入关键字为k的新结点(递归实现)
int BST_Insert(BiTree &T,KeyType k){
    if(T==NULL){        //原树为空，新插入的结点为根结点
        T=(BiTree)malloc(sizeof(BSTNode));
        T->data=k;
        T->lchild=T->rchild=NULL;
        return 1;        //返回1,插入成功
    }
    else if(k==T->data)
        return 0;
    else if(k<T->key)//插入到T的左子树
        return BST_Insert(T->lchild,k);
    else                //插入到T的右子树
        return BST_Insert(T->rchild,k);
}
```

26.1.4 二叉排序树的构造

插入顺序与构造

- 概念：
 - 根据插入的顺序，构造二叉树；
- 注意：
 - 不同的插入顺序，在同样内容的情况下，可能得到不同的二叉树；
 - 不同的关键字序列，也可能得到同款二叉排序树
- 举例：
 - 插入：50, 66, 60, 26, 21, 30, 70, 68
 - 图示：



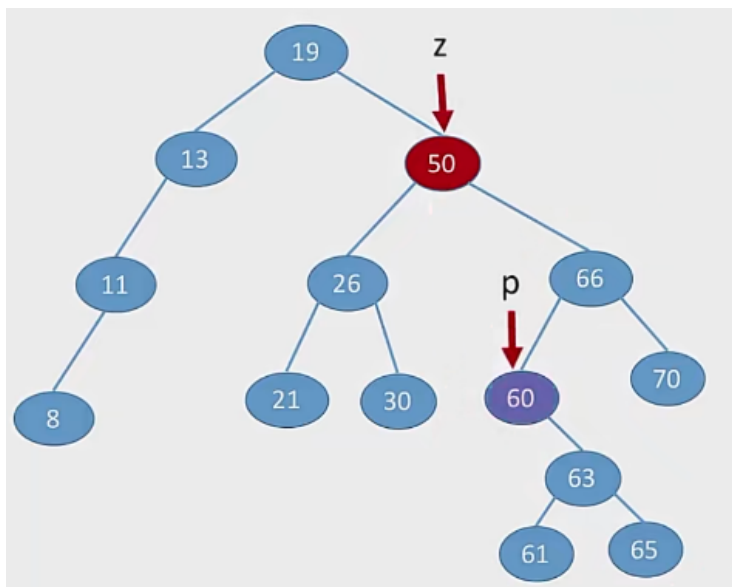
代码实现

```
//按照str[]中的关键字序列建立二叉排序树
void Creat_BST(BiTree &T,KeyType str[],int n){
    T=NULL;        //初始时T为空树
    int i=0;
    while(i<n){ //依次将每个关键字插入到二叉排序树中
        BST_Insert(T,str[i]);
        i++;
    }
}
```

26.1.5 删除

概念

- 1. 若被删除结点 z 是叶结点：
 - 则直接删除，不会破坏二叉排序树的性质。
- 2. 若结点 z 只有一棵左子树或右子树：
 - 则让 z 的子树称为 z 父结点的子树，替代 z 的位置。
- 3. 若结点 z 有左，右两棵子树：
 - 则令 z 的直接后继 (或直接前驱) 替代 z ，然后从二叉排序书中删除这个直接后继 (或直接前驱)，这样就转换成了第一或第二种情况。
 - 直接前驱：
 - 找到当前结点 z 的左子树当中，最大的那个结点，用左子树中最大的值替代 z ，也可以满足二叉排序树的特性；
- 举例：
 - 当结点有左右两个子树时，使用 z 的直接后继替代；



26.1.6 查找效率分析

概念：查找成功

- 查找长度：
 - 在查找运算中，需要对比键字的次数称为查找长度，反映了查找操作时间复杂度；
 - 若树高 h ，找到最下层的一个结点需要对比 h 次；
 - 性能：
 - 最好情况：
 - n 个结点的二叉排序树的最小高度为：

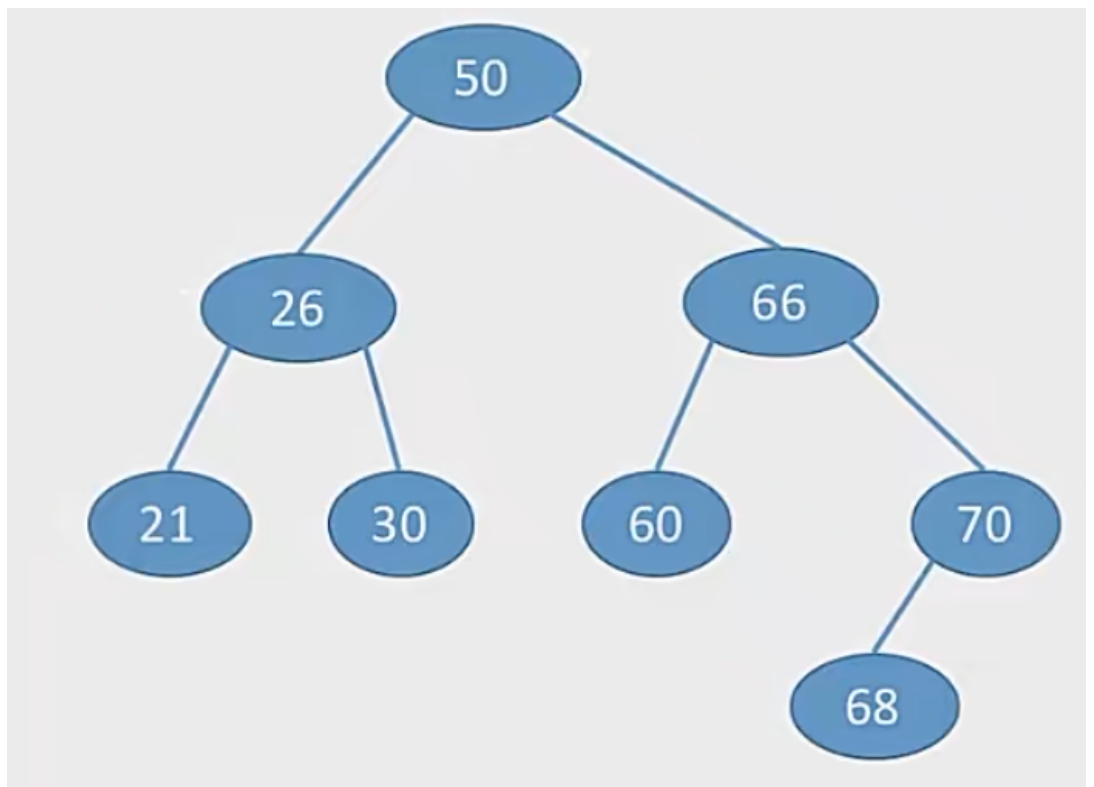
$$\text{最小高度} = \lfloor \log_2 n \rfloor + 1$$
 - 平均查找长度：

$$\text{平均查找长度} = O(\log_2 n)$$
 - 最差情况：
 - 每个结点只有一个分支：

$$\text{树高} h = \text{结点数} n$$
 - 平均查找长度：

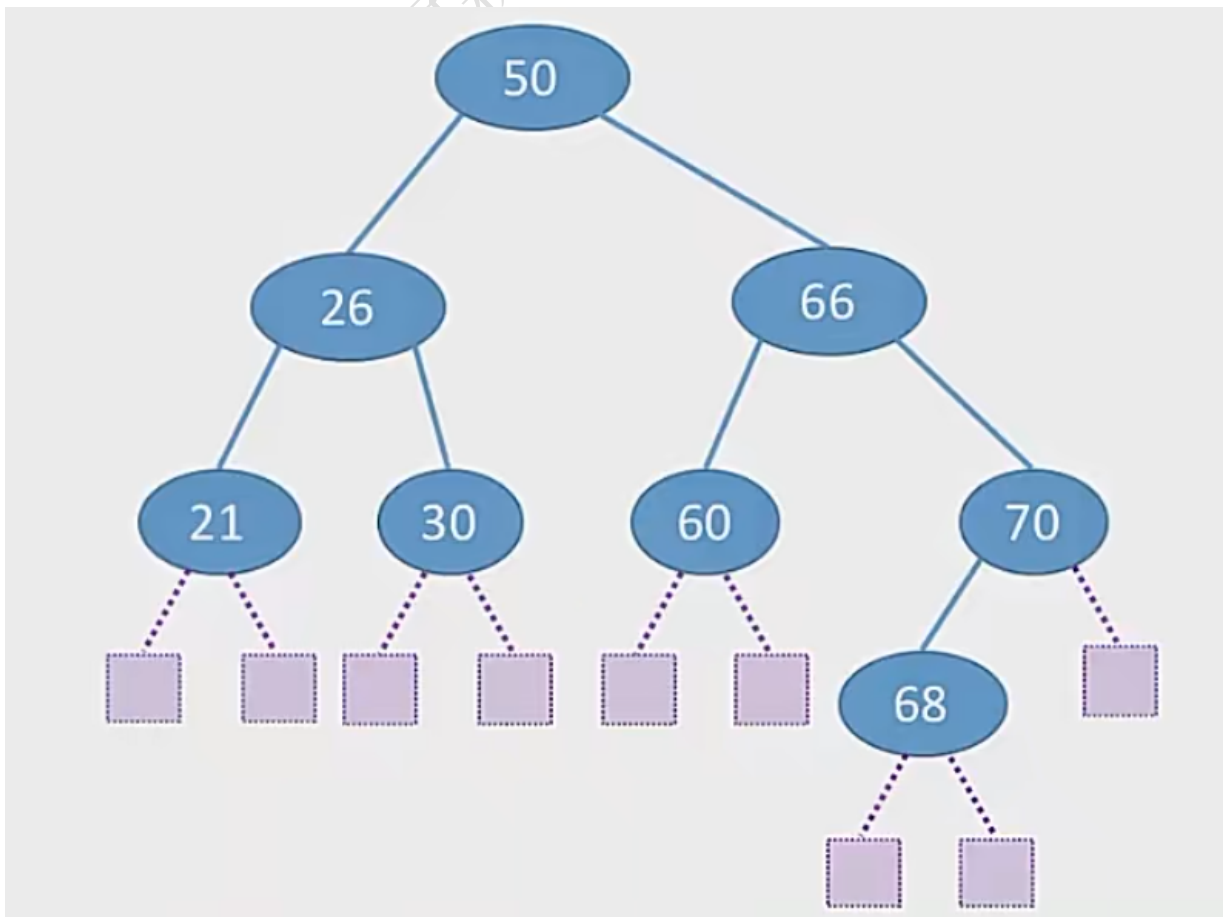
$$\text{平均查找长度} = O(n)$$
- 平均：
 - 查找成功的平均查找长度 ASL (Average search Length)；
 - 举例：
 - 图示：

$$ASL = (1 * 1 + 2 * 2 + 3 * 4 + 4 * 1) / 8 = 2.625$$



概念：查找失败

- 概念：
 - 当前树的所有结点当中，空的左右指针，都代表了一种查找失败的情况；
 - 空的左右指针所处的高度，代表了其查找失败时所付出的代价；
- 举例：
 - $ASL = (3 * 7 + 4 * 2) / 9 = 3.22$



- 公式:

•

$$ASL = \sum \frac{\text{本层高度} * \text{本层叶子数}}{\text{补上的叶子数}}$$

26.2 平衡二叉树

26.2.1 基本概念

概念

- 定义:
 - 平衡二叉树 (Balanced BinaryTree), 简称平衡树 (AVL) ;
 - 树上任一结点的左子树和右子树的**高度之差不超过 1**;
- 查找效率:
 - 假设以 n_h 表示深度为 h 的平衡树中含有的**最少节点数**;
 - 则有 $n_0 = 0, n_1 = 1, n_2 = 2$;
- 相关结论:

- 结论一: 结点数

•

$$n_h = n_{h-1} + n_{h-2} + 1$$

- 结论二: 最大深度

•

$$O(\log_2 n)$$

- 平均查找长度:

•

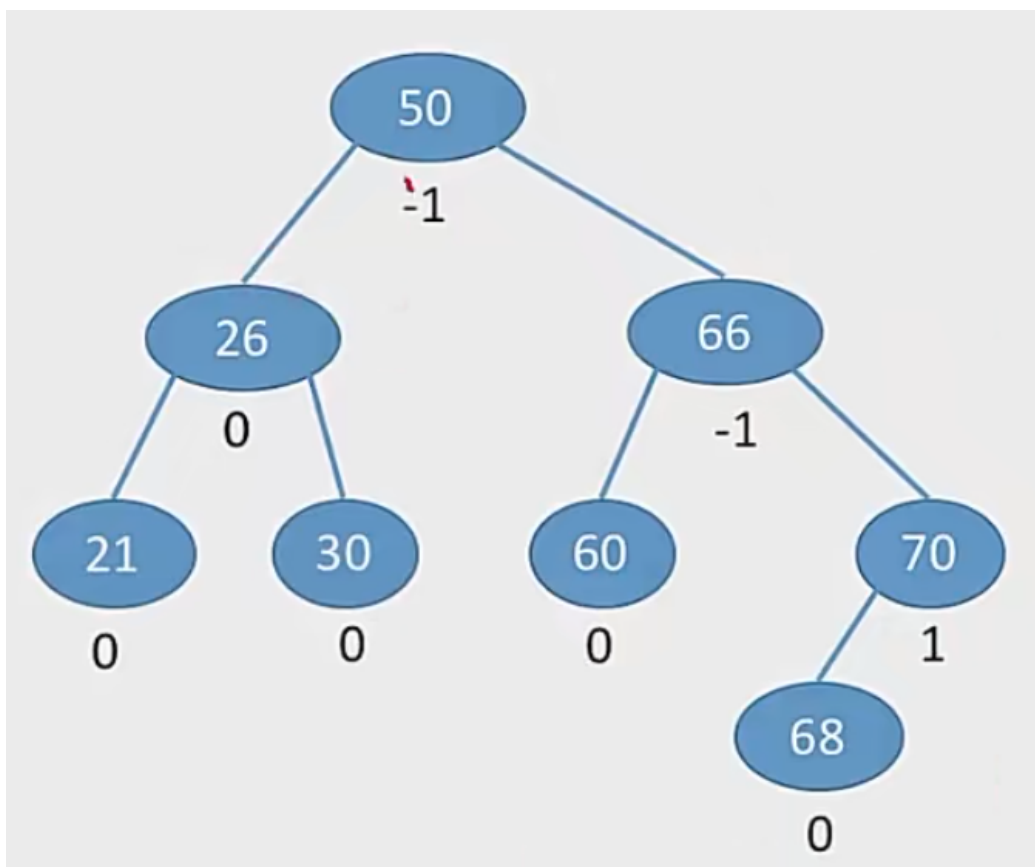
$$O(\log_2 n)$$

结点的平衡因子

- 特性:
 - 平衡二叉树结点的平衡因子的值只可能是 **-1、0 或 1**;
 - 结点的平衡因子 = 左子树高 - 右子树高
- 计算:
 - 计算平衡因子 **->**

当前节点的左子树的最大深度 - 当前节点的右子树的最大深度 = 当前节点的平衡因子

- 图示:

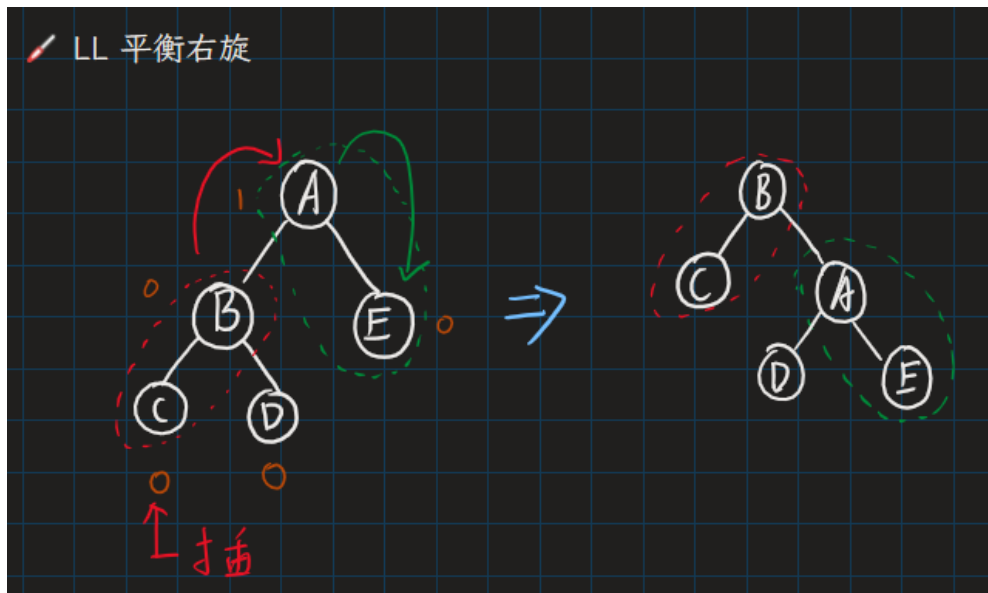


26.2.2 平衡二叉树的插入

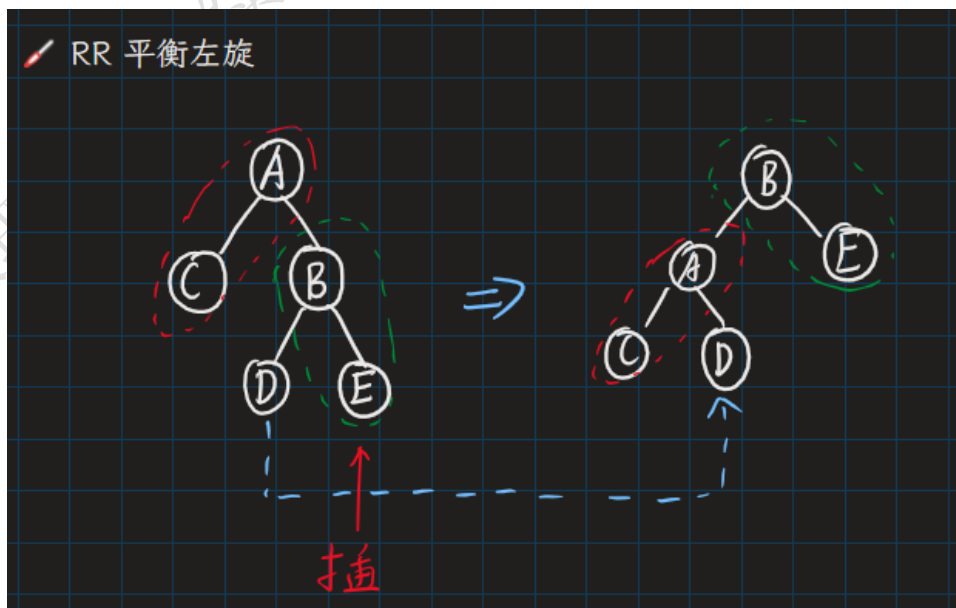
插入操作：插入步骤总结

- 对平衡二叉树的插入操作分成以下几个步骤：
- (1) 执行插入：
 - 此步骤和二叉排序树一致，根据当前插入结点的值、找到合适的插入位置；
- (2) 判断平衡因子：
 - 计算当前插入后的各个节点的平衡因子；
- (3) 判断调整：
 - 1. 如果插入之后、当前二叉树是平衡的，则无需调整；
 - 2. 如果插入之后、当前二叉树是不平衡的，则进行调整；
- (4) 调整 - 找到最小不平衡子树
 - 从插入点往回找，找到第一个不平衡的结点，以该结点为接下来进行调整的树的根；
 - 通过此操作保证：每次调整的对象都是“最小不平衡子树”；
 - 在插入操作中，只要**将最小不平衡子树调整平衡**，则其他祖先结点都会恢复平衡；
- (5) 调整 - 旋转子树
 - 通过旋转、使得最小不平衡子树恢复平衡；
 - LL：
 - 判断：
 - 在 A 的左孩子的左子树中插入导致不平衡；
 - 方法：
 - 执行向右旋转：
 - 1. 左孩子向右旋转、到达原本根结点的位置；

- 2. 根节点向右旋转、到达当前根节点的右孩子的位置；
- 3. 左孩子原本的右节点，成为原本根节点的左孩子；
- 图示：

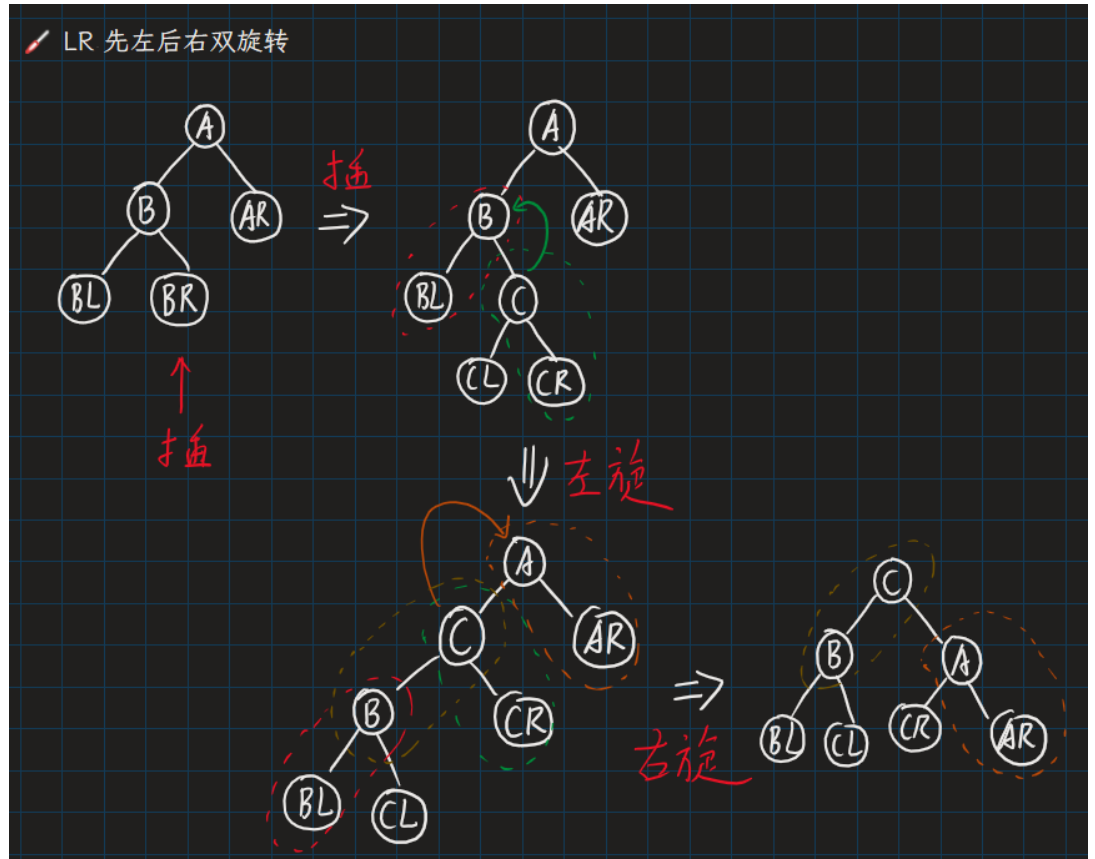


- RR:
- 判断：
 - 在 A 的右孩子的右子树中插入导致不平衡；
- 方法：
 - 执行向左旋转：
 - 1. 右孩子 (R) 向左旋转、到达原本根结点 (ROOT) 的位置；
 - 2. 根节点 (ROOT) 向左旋转、到达当前根节点 (R) 的左孩子的位置；
 - 3. 右孩子原本的左节点 (RL)，成为原本根节点 (ROOT) 的右孩子；
- 图示：



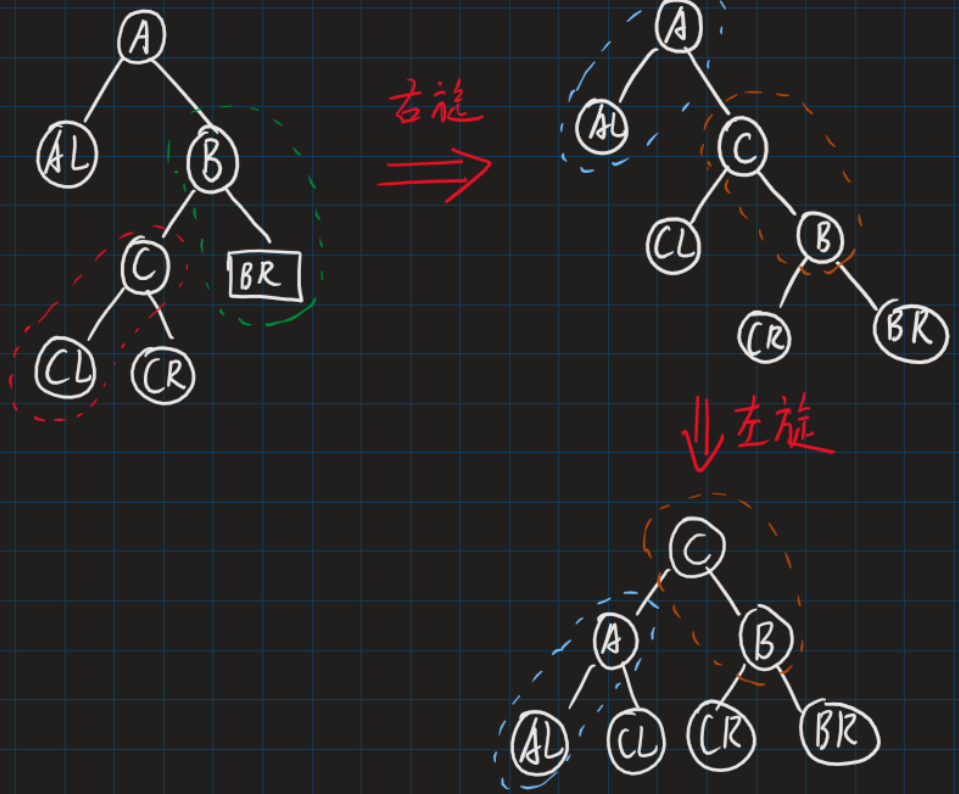
- LR:
- 判断：
 - 在 A 的左孩子的右子树中插入导致不平衡；
- 方法：

- 先以 A 的左孩子为根、执行左旋转（方法同 LL），再以 A 的根为根、执行右旋转（方法同 RR）
- 图示：



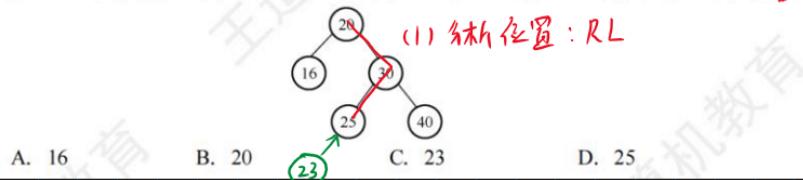
- RL:
- 判断：
 - 在 A 的右孩子的左子树中插入导致不平衡；
- 方法：
 - 先以 A 的右孩子为根、执行右旋转（方法同 RR），再以 A 的根为根、执行左旋转（方法同 LL）
- 图示：

RL 先右后左双旋转

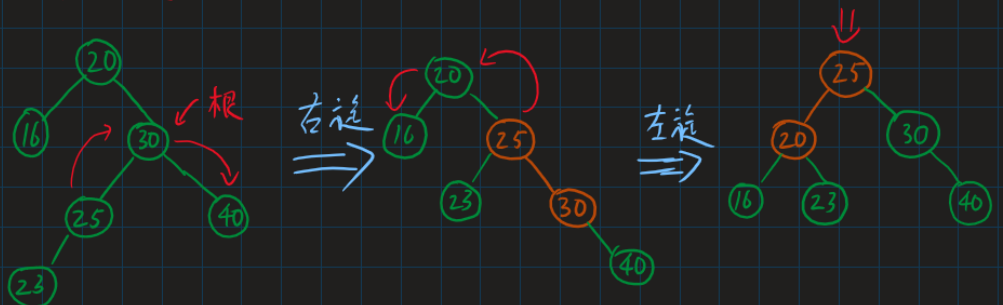


真题:

【2021 统考真题】给定平衡二叉树如下图所示，插入关键字 23 后，根中的关键字是 (D)。

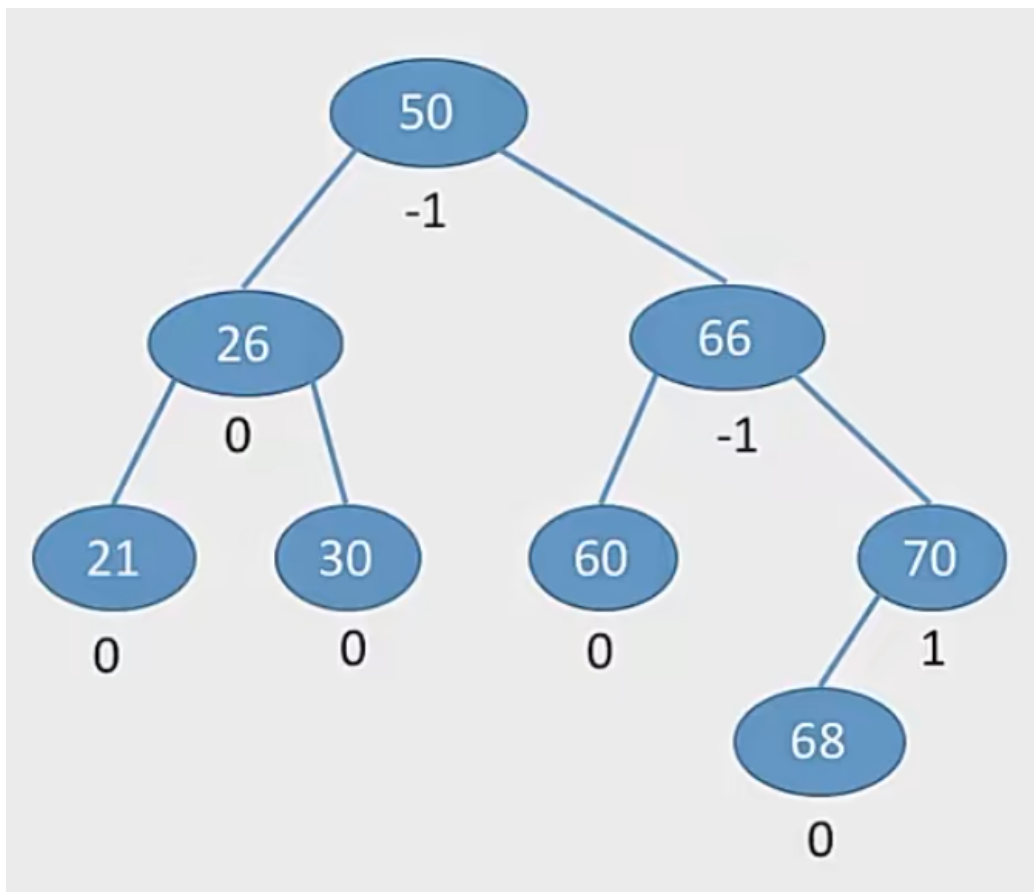


(2) 执行旋转

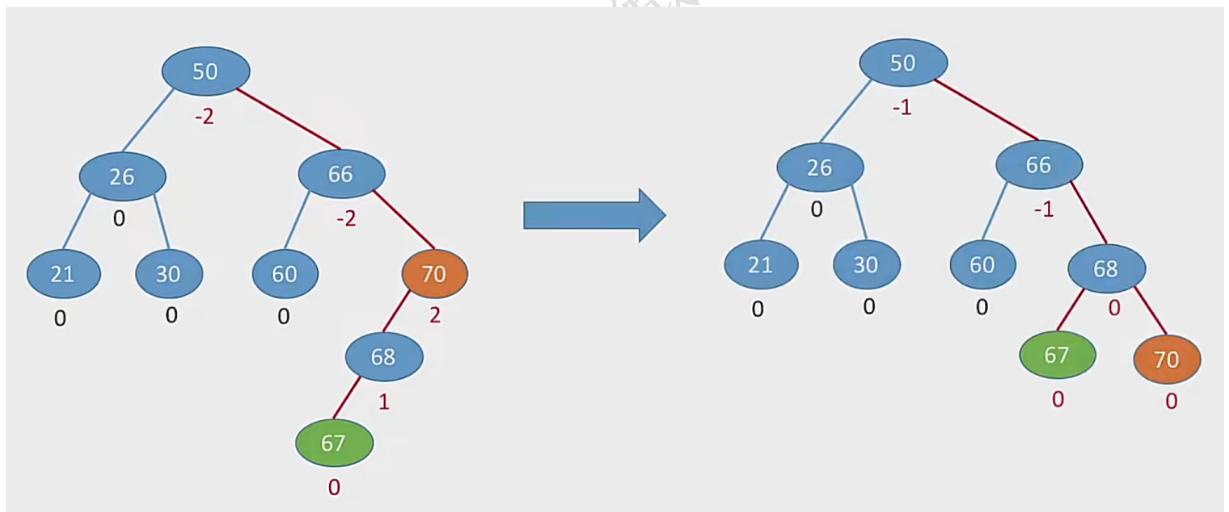


举例: 插入操作

- 插入 67;
- 原本的树:

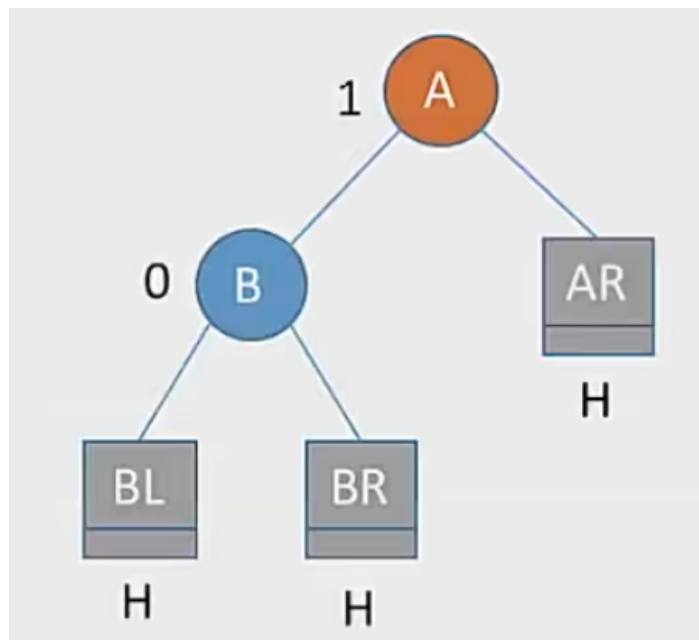


- 插入到了 68 的左子树;



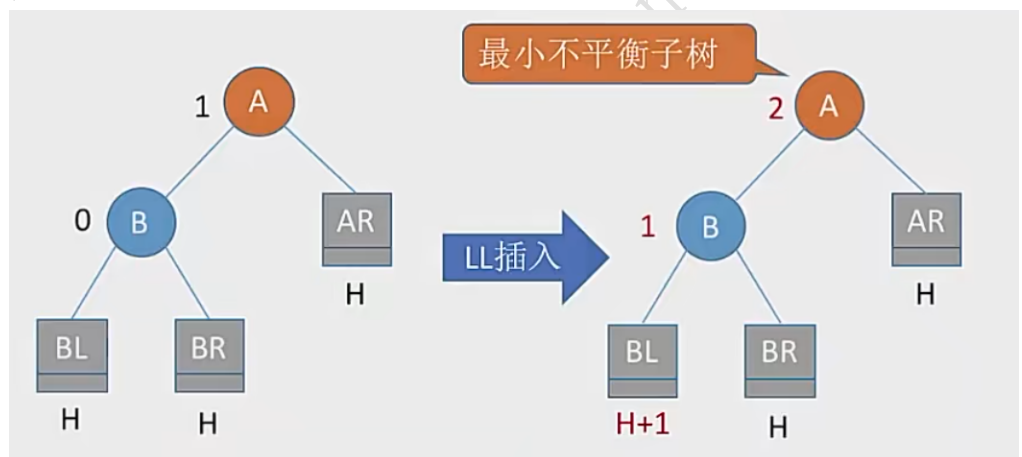
LL 分析

- 目标:
 - 1. 恢复平衡;
 - 2. 保持二叉排序树特性
- 举例:
 - 在 BL 中插入;
 - 图示:
 - 插入前:



• 图示：

• 插入后：



• 实现：将 B 右旋转

• 概念：

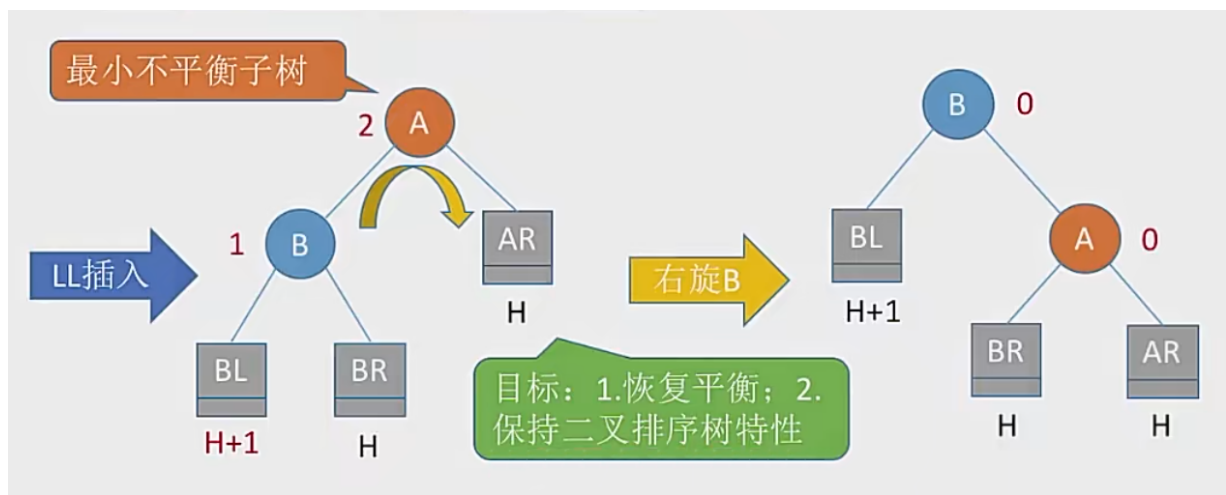
• LL 平衡旋转 (右单旋转)；

• 由于在结点 A 的左孩子 (L) 的左子树 (L) 上插入了新结点，A 的平衡因子由 1 增至 2，导致以 A 为根的子树失去平衡，需要一次向右的旋转操作；

• 操作：

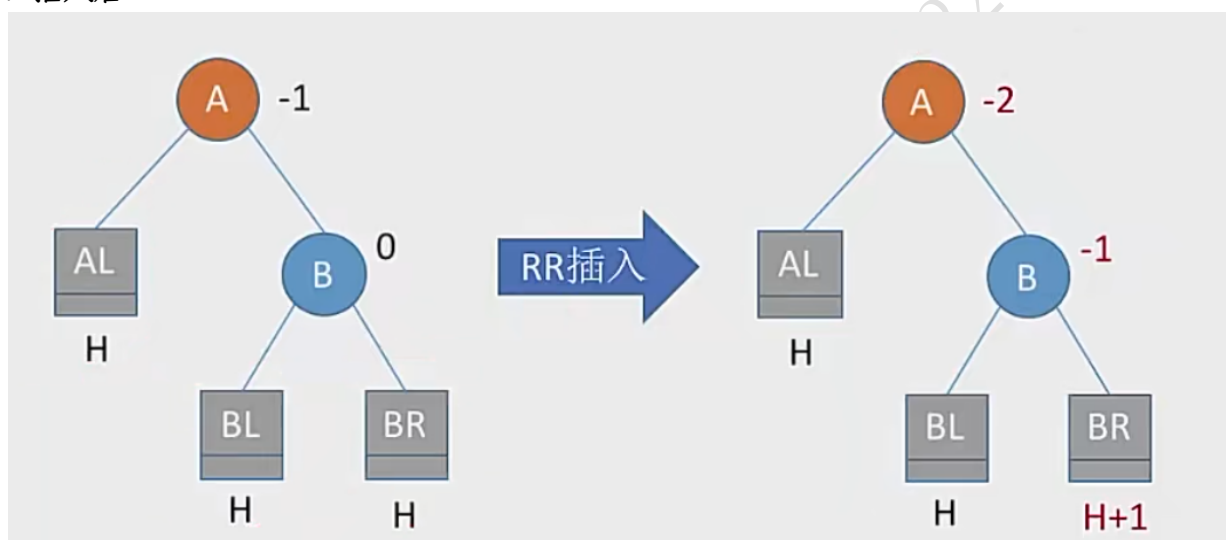
- 1. 将 A 的左孩子 B 向右上旋转代替 A 成为根结点；
- 2. 将 A 结点向右下旋转成为 B 的右子树的根结点；
- 3. B 的原右子树，则作为 A 结点的左子树；

• 图示：右旋转

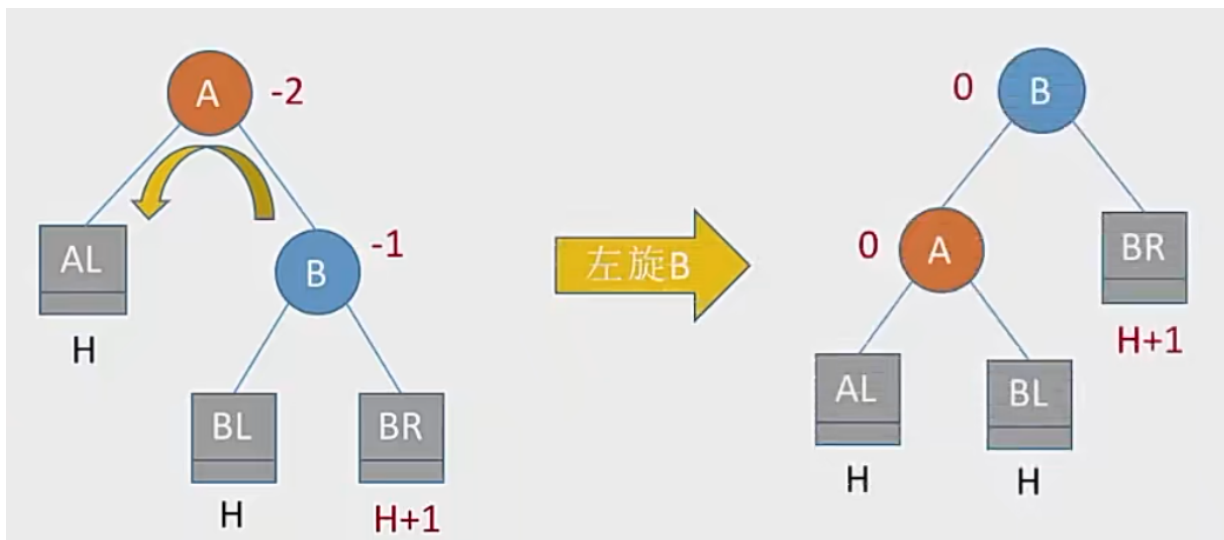


RR 分析

- 图示：插入后

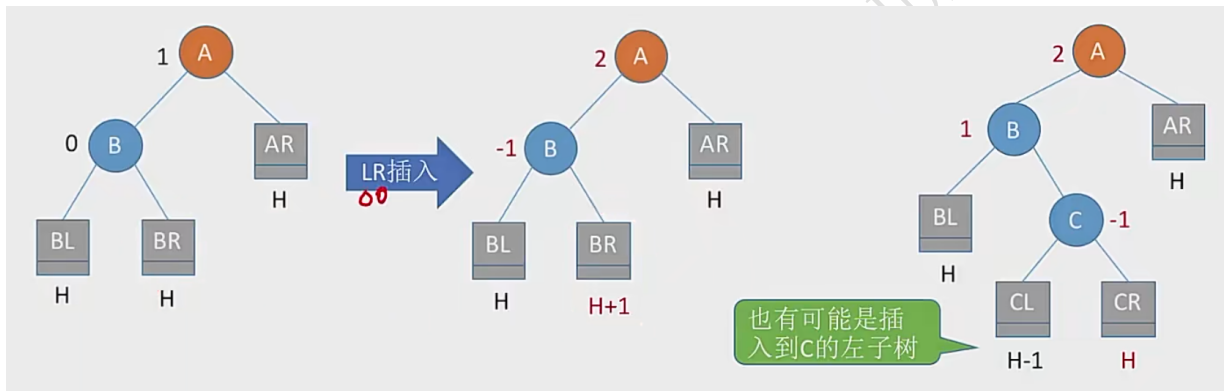


- 实现：左旋调整
 - 把 B 放到根节点，A 放到 B 的左子树；
 - AL 因为是最小的、并且小于 A，所以还是在 A 的左子树；
 - BR 因为是最大的、并且大于 B，所以还是在 B 的右子树；
 - BL 因为原本是在 B 的左子树。但因为现在 A 是 B 的左子树，BL 没有位置了，所以需要把 BL 放到 A 的右子树上；
- 图示：左旋调整

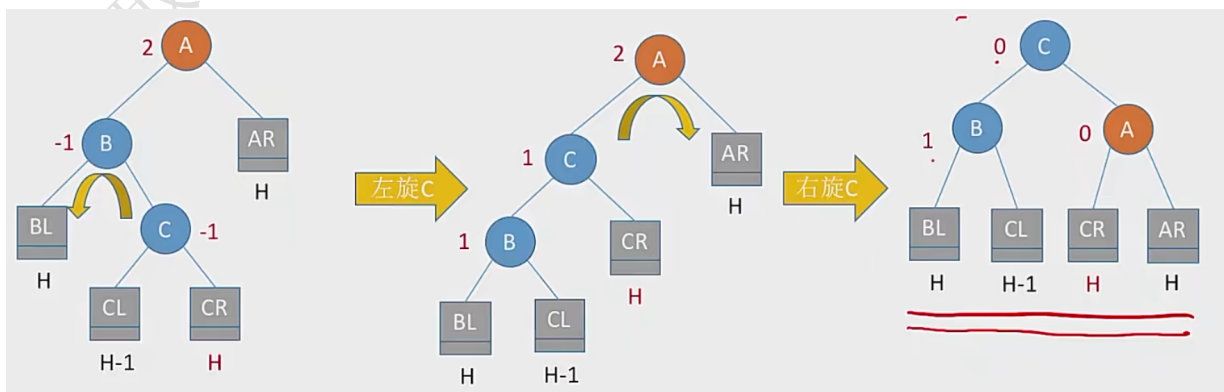


LR 分析

- 图示：插入后

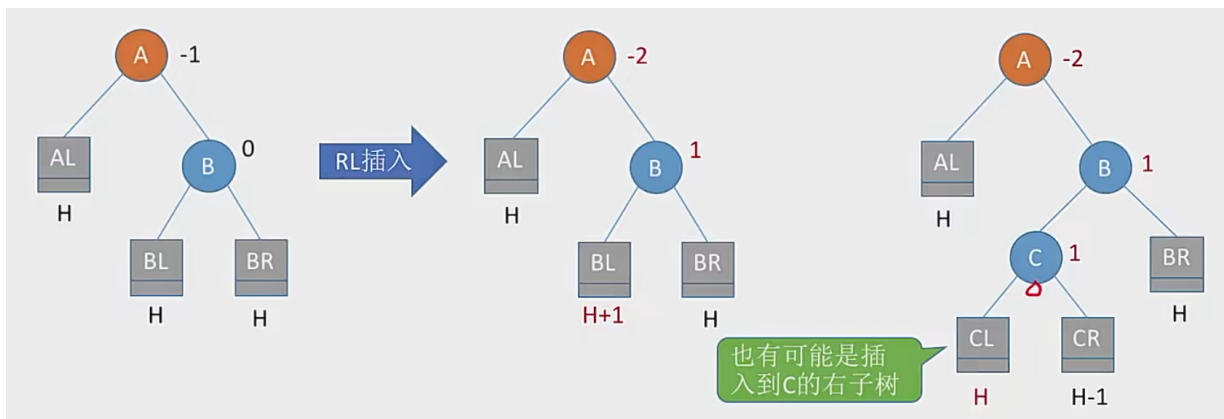


- 实现：先左后右双旋转
 - LR 平衡旋转 (先左后右双旋转);
 - 由于在 A 的左孩子 (L) 的右子树 (R) 上插入新结点, A 的平衡因子由 1 增至 2, 导致以 A 为根的子树失去平衡, 需要进行两次旋转操作, 先左旋转后右旋转;
 - 先将 A 结点的左孩子 B 的右子树的根结点 C 向左上旋转提升到 B 结点的位置;
 - 然后再把该 C 结点向右上旋转提升到 A 结点的位置;
- 图示:



RL 分析

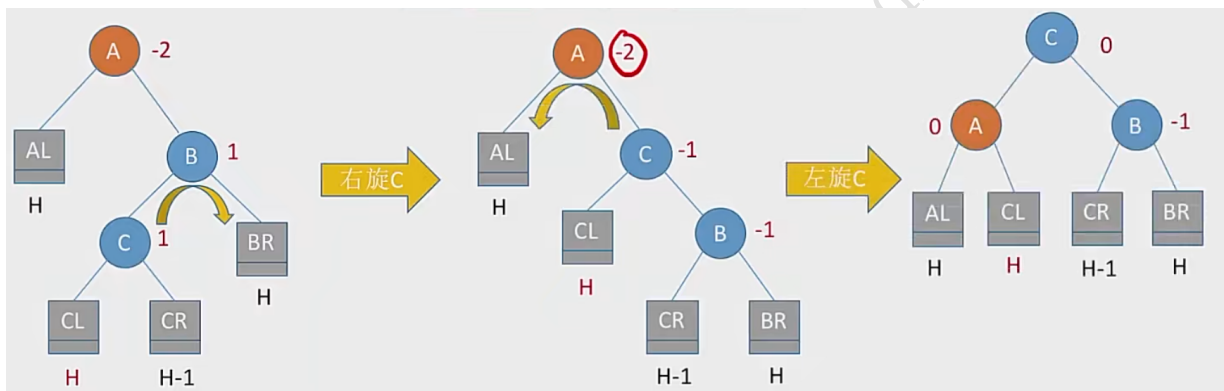
- 图示：插入后



• 实现：先右后左双旋转

- 由于在 A 的右孩子 (R) 的左子树 (L) 上插入新结点，A 的平衡因子由 -1 减至 -2，导致以 A 为根的子树失去平衡，需要进行两次旋转操作，先右旋转后左旋转；
- 先将 A 结点的右孩子 B 的左子树的根结点 C 向右上旋转提升到 B 结点的位置；
- 然后再把该 C 结点向左上旋转提升到 A 结点的位置；

• 图示：



26.2.3 总结

旋转总结

- LL:
 - 概念：在 A 的左孩子的左子树中插入导致不平衡；
 - 旋转：将 A 的左孩子 B 右上旋转，A 成为 B 的右子树；
- RR:
 - 概念：在 A 的右孩子的右子树中插入导致不平衡；
 - 旋转：将 A 的左孩子 B 左上旋转，A 成为 B 的左子树；
- LR:
 - 概念：在 A 的左孩子的右子树中插入导致不平衡；
 - 旋转一：将 A 的左孩子的右孩子向左上旋转；
 - 旋转二：然后基于第一次旋转，再进行右上旋转；
- RL:
 - 概念：在 A 的右孩子的左子树中插入导致不平衡；
 - 旋转一：将 A 的右孩子的左孩子向右上旋转；
 - 旋转二：然后基于第一次旋转，再进行左上旋转；

26.3 平衡二叉树的删除操作

概念

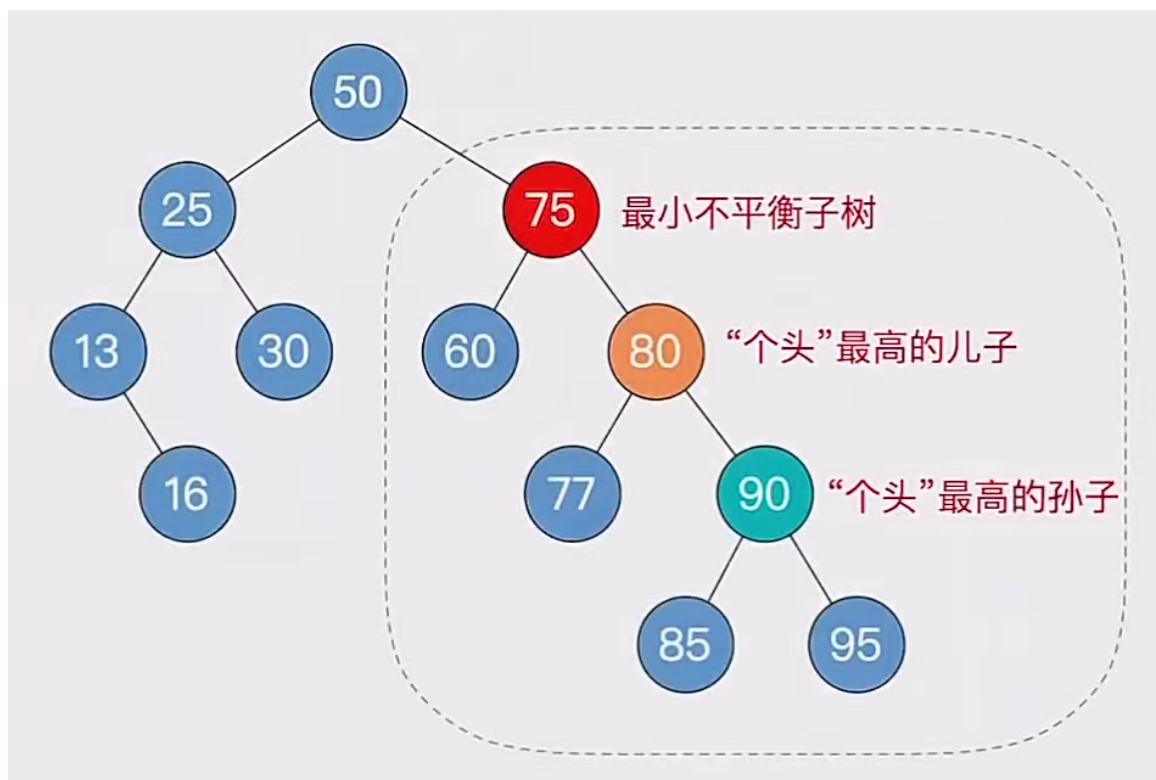
- 插入要求：
 - 1. 插入新结点后，要保持二叉排序树的特性不变 (左<中<右);
 - 2. 若插入新结点导致不平衡，则需要调整平衡;
- 删除要求：
 - 1. 删除结点后，要保持二叉排序树的特性不变 (左<中<右);
 - 2. 若删除结点导致不平衡，则需要调整平衡;

删除步骤

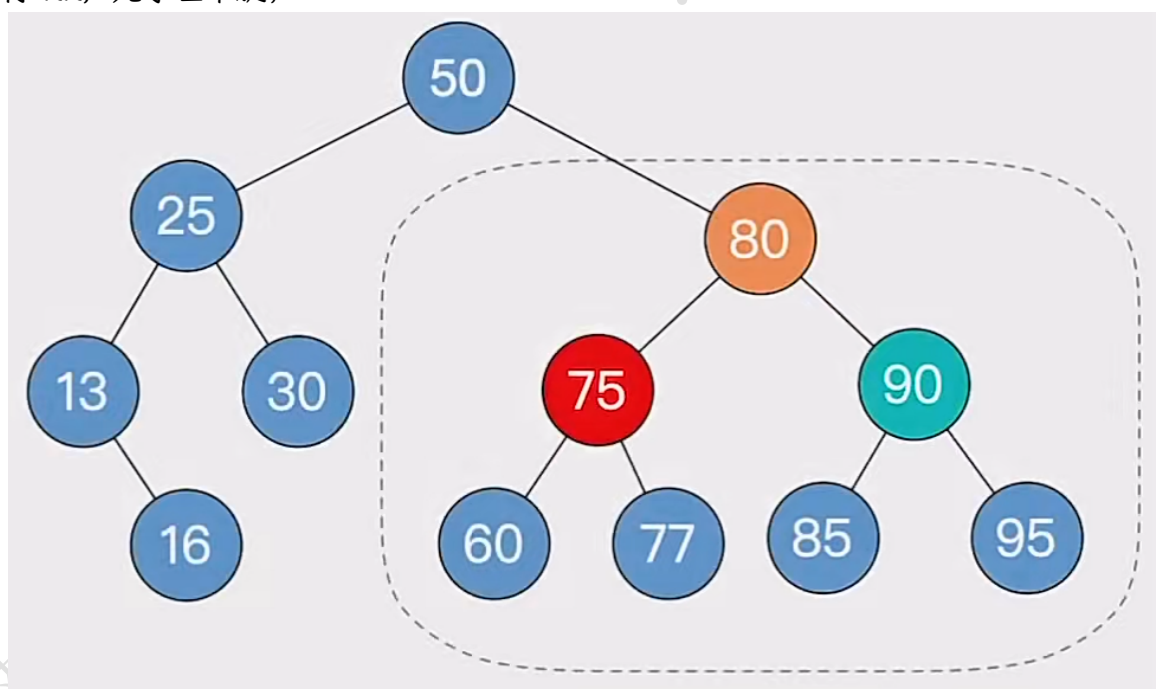
- 1. 找到删除结点，方法同二叉排序树；
 - 若删除的结点是叶子，直接删；
 - 若删除的结点只有一个子树，用子树顶替删除位置；
 - 若删除的结点有两棵子树，用**前驱或后继结点顶替**，并转换为对前驱或后继结点的删除；
 - 前驱节点：左子树中最右边（即左子树中值最大的）的结点；
- 2. 往上回溯、直到找到最小不平衡子树；
- 3. 找到最小不平衡子树下，高度最大的儿子、孙子；
- 4. 根据孙子的位置，调整平衡 (LL、RR、LR、RL)
 - 方法同插入操作
 - 孙子在 LL：儿子右单旋；
 - 孙子在 RR：儿子左单旋；
 - 孙子在 LR：孙子先左旋转，再右旋转；
 - 孙子再 RL：孙子先右旋转，再左旋转；
- 5. 如果不平衡往上传到，则重复 2 操作；
 - 以上四步都和插入操作一致，此步骤是删除和插入操作不同的地方；
 - 对最小不平衡子树的删除带来的旋转可能导致树变矮，从而导致上层祖先不平衡 (不平衡向上传递)；
 - 因此如果调整导致祖上结点不平衡，则还需往上找、找到最小不平衡子树，调整使其平衡 (方法同插入) ；

AVL 删除分析

- 情况：
 - 删除了结点 60 下的结点，此时 75 的左子树高度为 1，75 的右子树高度为 3，不平衡；
- 分析：
 - 找到最小不平衡子树下，"个头"最高的儿子、孙子；



- 孙子再 RR，儿子左单旋；



- 检查不平衡性是否上传导了；

26.4 红黑树基础概念

26.4.1 概念引入

引入

- AVL 树：
 - 红黑树得插入、查找、删除效率是一样的；
 - 但 AVL 的插入/删除很容易破坏“平衡”的特性，需要频繁调整树的形态；
 - 并且每次插入删除都需要计算平衡因子，找到最小不平衡子树，时间开销大；

- 红黑树：
 - 插入/删除很多时候不会破坏“红黑”特性，**无需频繁调整树的形态**；
 - 即便需要调整般都可以在**常数级时间**内完成；

		BST	AVL Tree	Red-Black Tree
生日		1960	1962	1972
时间复杂度	Search (查)	$O(n)$	$O(\log_2 n)$	$O(\log_2 n)$
	Insert (插)	$O(n)$	$O(\log_2 n)$	$O(\log_2 n)$
	Delete (删)	$O(n)$	$O(\log_2 n)$	$O(\log_2 n)$

使用情况分析

- 平衡二叉树：
 - 适用于以查为主、很少插入/删除的场景
- 红黑树：
 - 适用于频繁插入、删除的场景，实用性更强；

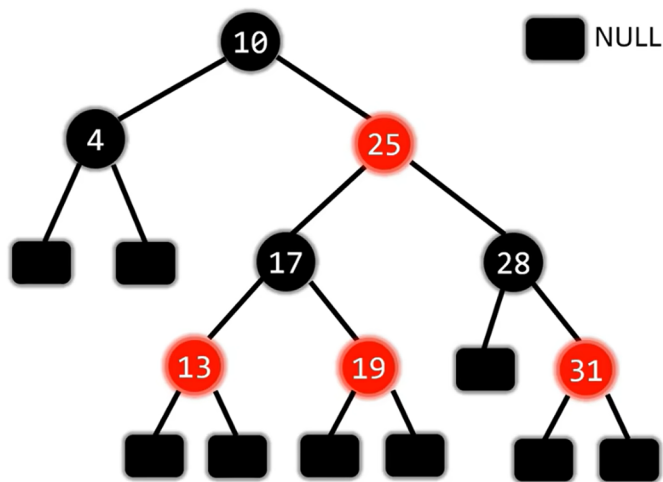
红黑树考察方向

- 红黑树的定义、性质：选择题
- 红黑树的插入/删除：要能手绘插入过程（不太可能考代码，略复杂），删除操作也比较麻烦，也许不考

26.4.2 红黑树的定义

概念

- 概念：
 - 红黑树是二叉排序树 $\rightarrow$ 左子树结点值 $\leq$ 根结点值 $\leq$ 右子树结点值；
- 性质：
 1. 每个节点是红色或黑色的；
 2. 根节点是黑色的；
 3. 叶节点（外部结点、NULL 结点、失败节点）均是黑色的；
 - 注意：叶节点是图中 **NIL** 的结点；
 4. 不存在两个相邻的红节点（即红节点的父节点和孩子节点均是黑色）；
 5. 对每个结点，从该节点到任一节点的简单路径上，所含**黑结点的数目相同**；



前提: 二叉搜索树(左 < 根 < 右)

左根右

根和叶子(NULL)都是黑色

根叶黑

不存在连续的两个红色结点

不红红

任一结点到叶所有路径黑结点数量相同

黑路同

• 口诀:

• 左根右, 根叶黑

• 左根右: 左子树结点值 ≤ 根结点值 ≤ 右子树结点值

• 根叶黑: 根节点和叶节点必须是黑色的;

• 不红红, 黑路同

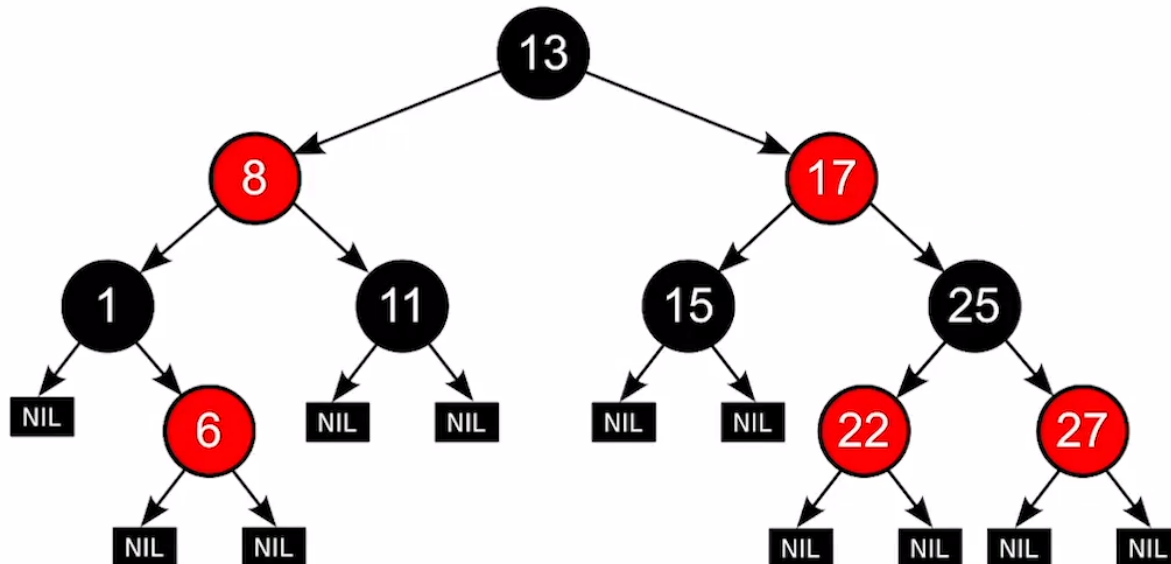
• 不红红: 不能出现连续相连的红节点;

• 黑路同: 任何一个结点出发, 到根节点经过的黑结点数量相同;

• 黑高:

• 结点的黑高 bh: 从某结点出发 (不含该结点) 到达任一空叶结点的路径上**黑结点总数**;

• 图示:



代码定义

```
struct RBnode{           //红黑树的结点定义
    int key;              //关键字的值
    RBnode* parent;      //父节点指针
    RBnode* lchild;      //左孩子指针
    RBnode* rchild;      //右孩子指针
    int color;            //结点颜色, 可用0/1黑, 红, 也可使用枚举型enum表示颜色
}
```

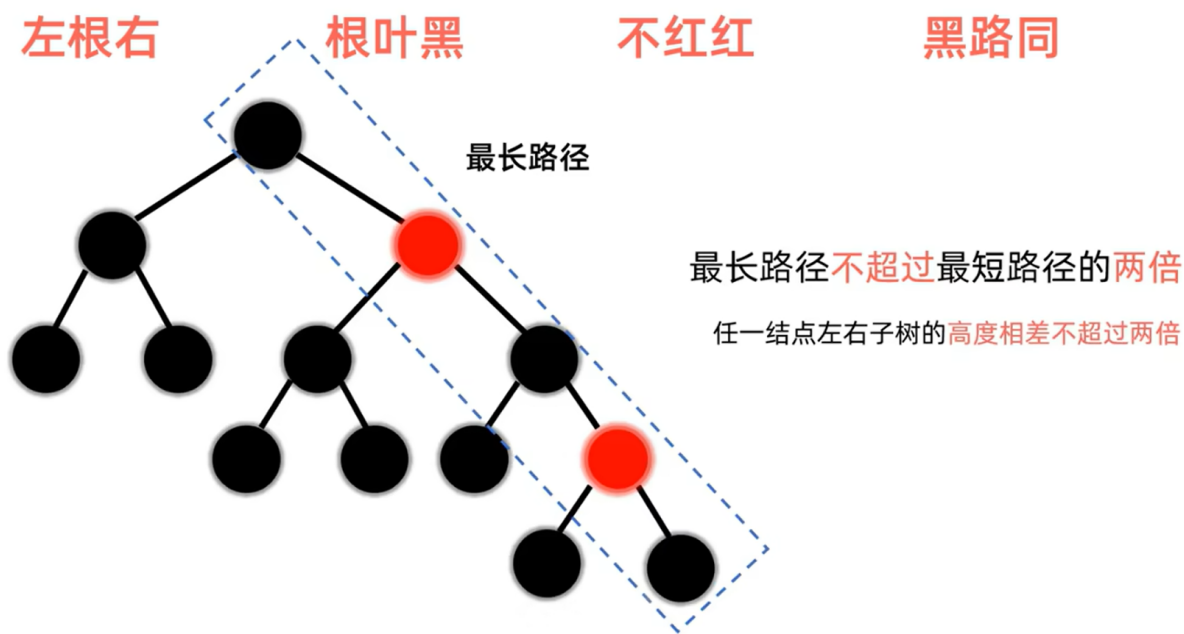
26.4.3 红黑树的性质

性质 1: 从根节点到叶节点的最长路径不大于最短路径的 2 倍；

• 概念:

- 根节点到任意叶节点的黑节点数相同，并且由于"不红红"，所以红色的结点只能穿插到黑节点中间，所以最大的数量也就是在每个黑节点之间穿插，所以不大于最短路径的 2 倍；

• 图示:



性质 2: 有 n 个内部节点的红黑树高度为 $h \leq 2\log_2(n + 1)$

- 解释: 因为从任意节点到根节点上，黑节点至少占当前结点的一半（全部都是黑红相间时），因此黑节点至少是 $2\log_2(n + 1)$
- 推论: 红黑树查找操作时间复杂度为 $O(\log_2 n)$

性质 3: 红黑树的查找操作和 AVL、BST 相同，若找到一个空叶节点，则查找失败

性质 4: 若根节点的黑高为 h ，则内部节点数最少有 $2^h - 1$ 个；

概念: AVL 树和红黑树比较

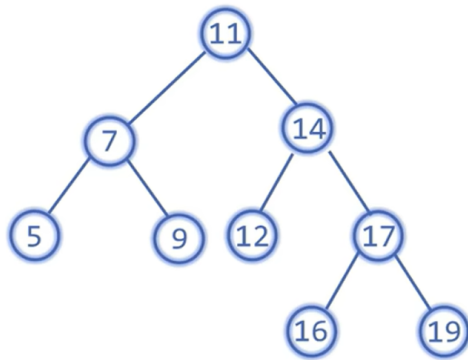
• 概念:

- 若删除和插入操作较少、则平衡二叉树更好；
- 若删除和插入操作较多、则红黑树更好；

• 图示:

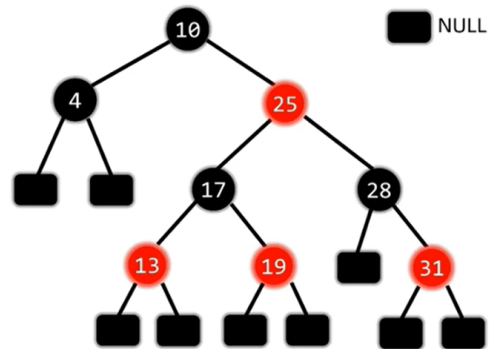
AVL树

任一结点左右子树的高度相差绝对值**不超过1**



红黑树

任一结点左右子树的高度相差**不超过两倍**



AVL树查询更高效

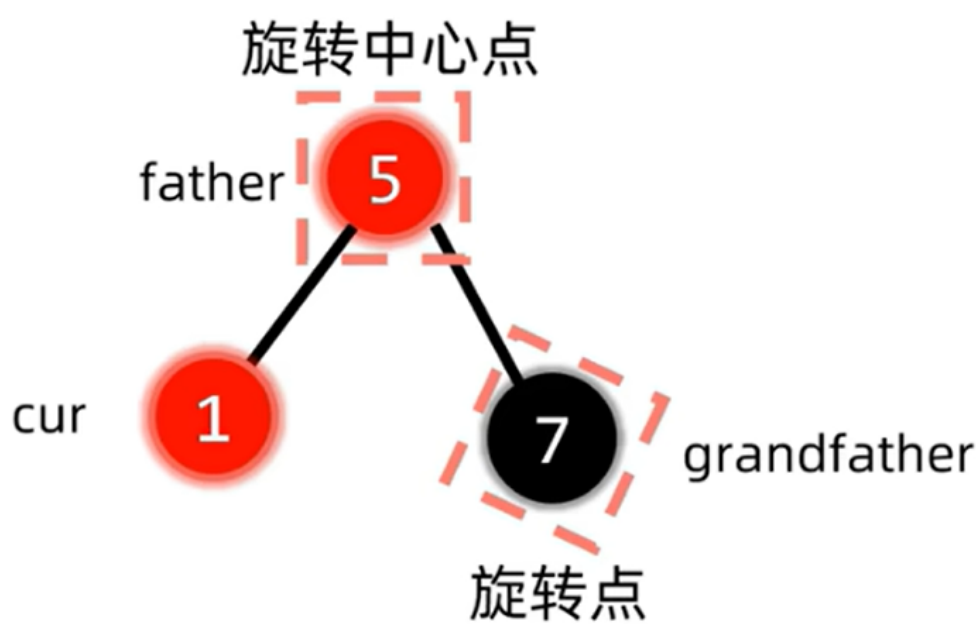
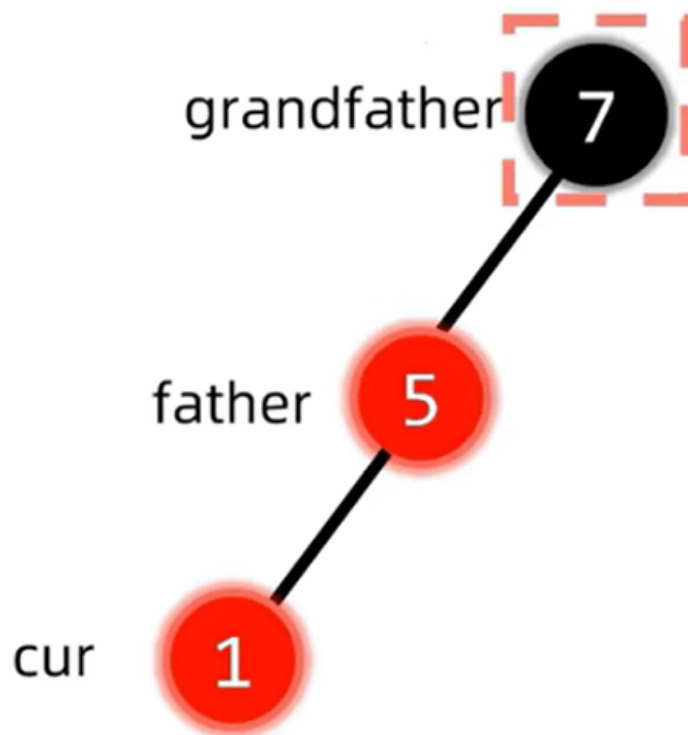
红黑树插入和删除更高效

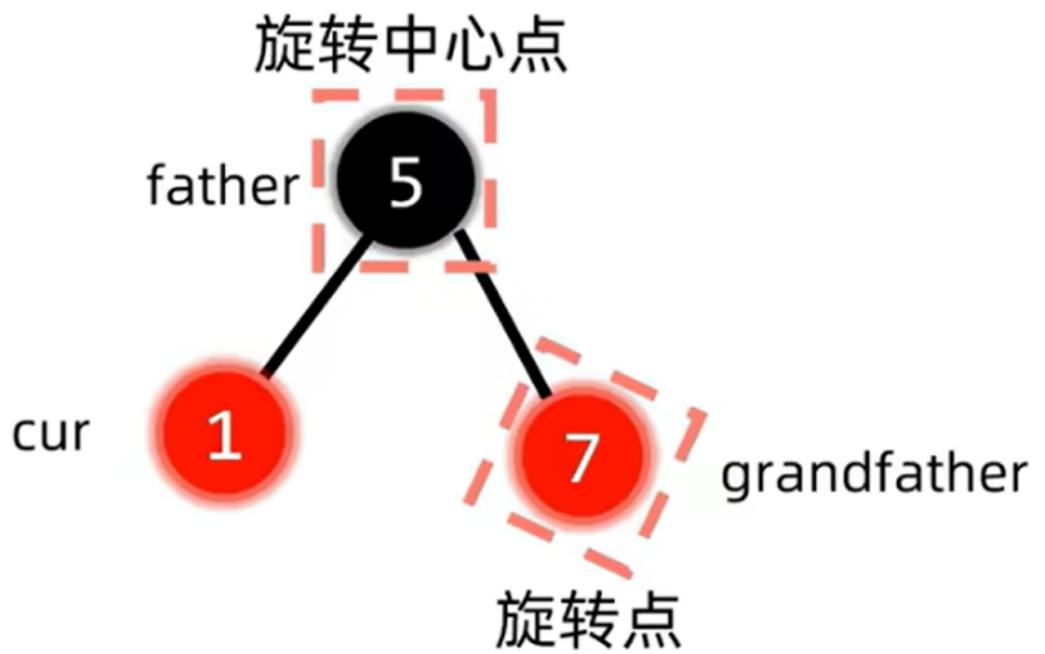
26.5 红黑树的插入与删除

26.5.1 插入方法

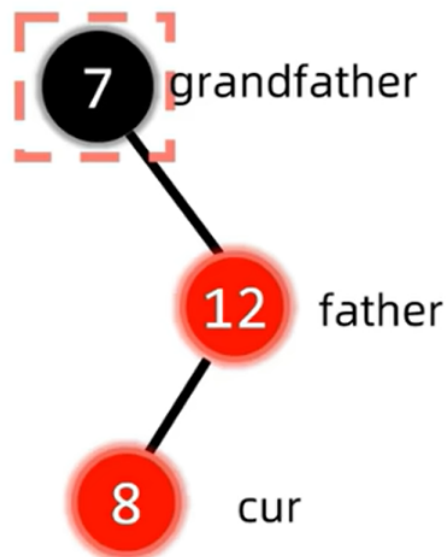
概念：插入方法

- 第一步：查找
 - 先查找，确定插入位置（原理同二叉排序树），插入新结点；
- 第二步：染色
 - 插入的新结点是根 -> 染为黑色；
 - 插入的新节点是非根 -> 染为红色
 - 因为如果插入的是黑色，则一定会破坏当前“黑路同”的条件，需要更频繁的进行调整；
 - 而如果插入的是红色、则只需要在出现“红红”时才需要调整，因此插入红色的代价更小；
- 第三步：结束或调整
 - 若插入新结点所依然满足红黑树定义（即：其父节点是黑色），则插入结束；
 - 若插入新结点后不满足红黑树定义，需要调整，使其重新满足红黑树定义；
- 第四步：调整
 - (4.1) 判断：
 - 看新结点的叔叔结点的颜色；
 - 叔叔结点：当前结点的父节点的兄弟结点；
 - (4.2.1) 如果是黑叔：旋转 + 染色
 - LL 型：右单旋，父换爷+染色



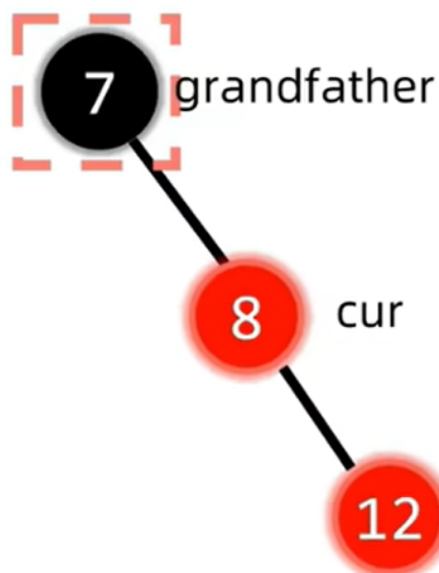


- RR 型: 左单旋, 父换爷+染色
- LR 型: 左、右双旋, 儿换爷+染色
- RL 型: 右、左双旋, 儿换爷+染色



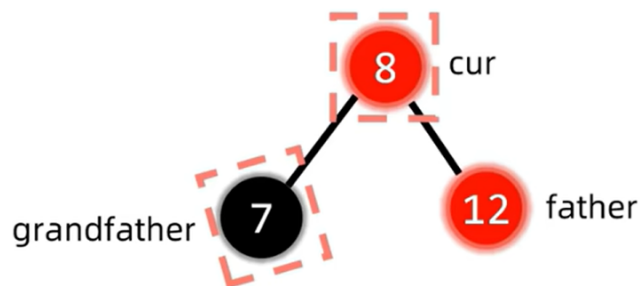
RL 型

右旋右孩子. 然后左旋



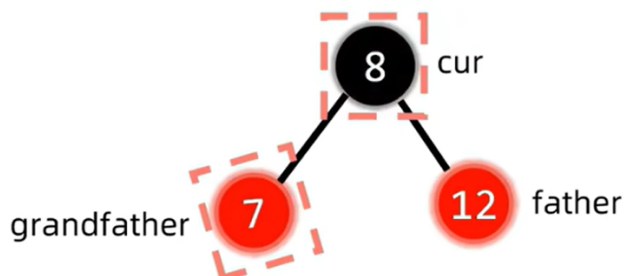
RL 型

右旋右孩子. 然后左旋



RL 型

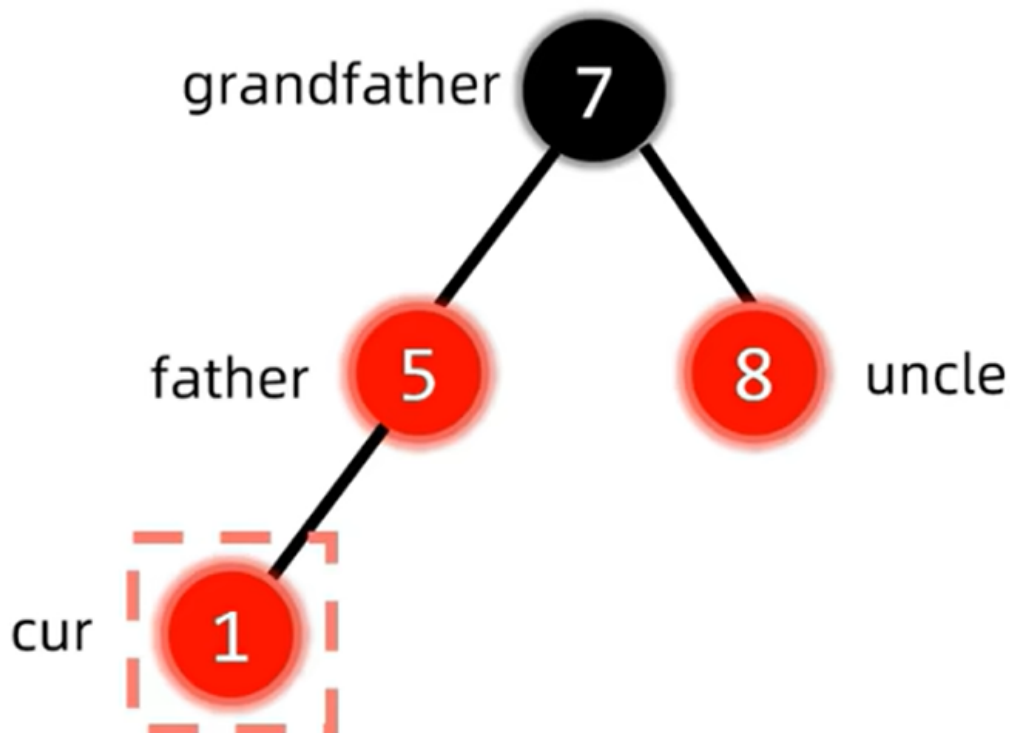
右旋右孩子. 然后左旋



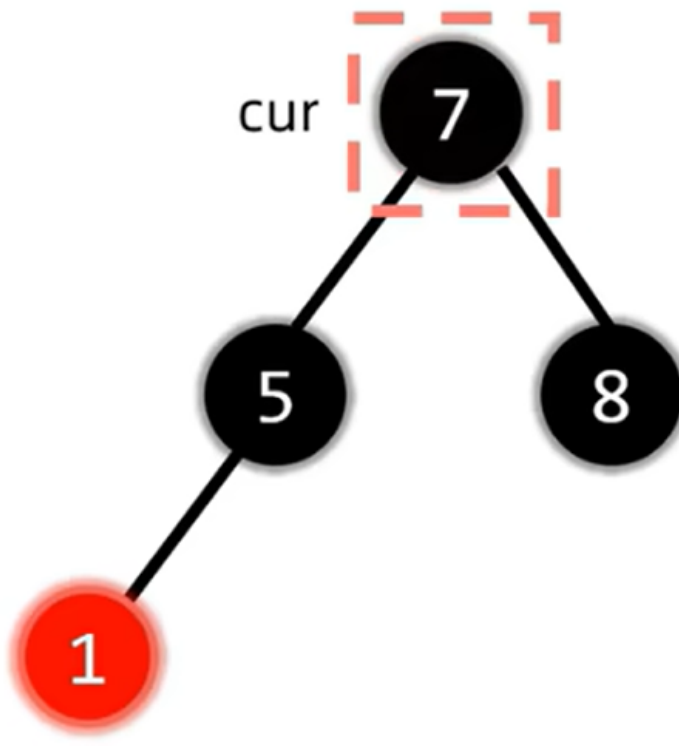
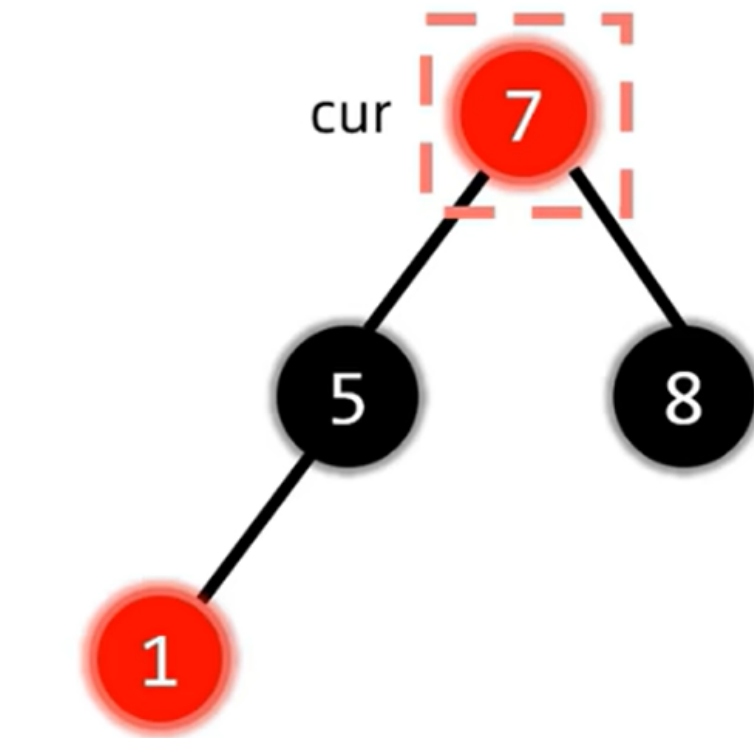
RL 型

右旋右孩子. 然后左旋

- (4.2.2) 如果是红叔: 染色 + 变新
 - 概念:
 - 叔父爷染色, 爷变为新结点, 继续往上判定是否破坏性质
 - 解释:
 - 当前的爷结点为黑色、将其染色为红色, 当前的两个红结点: 父结点和叔结点, 都染色成黑色;
 - 因为当前爷节点染色为了黑色, 其可能会破坏红黑特性。把当前爷结点看作是新节点, 往上继续调整;
 - 图示:



- 违反了: 根叶黑, 根节点染为黑色



26.5.2 插入举例

分析：红黑树插入过程

- 例题：
 - 插入序列为 17, 18, 23, 34, 27, 15
- 过程：
 - 插入 17:



•



•

• 插入 18



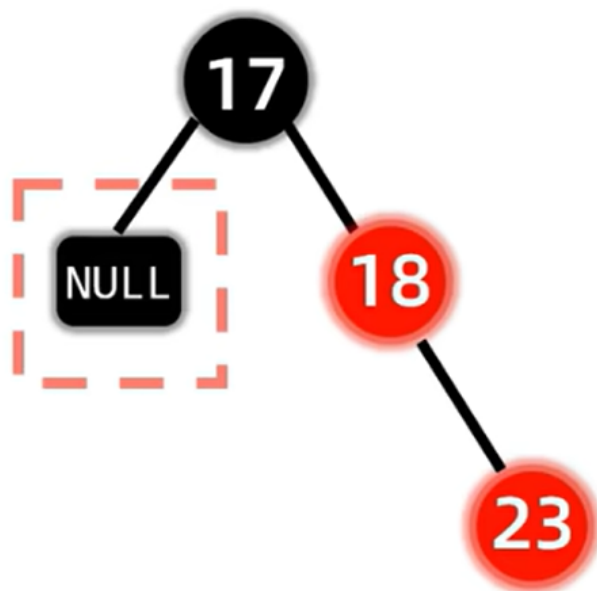
•

• 插入 23:

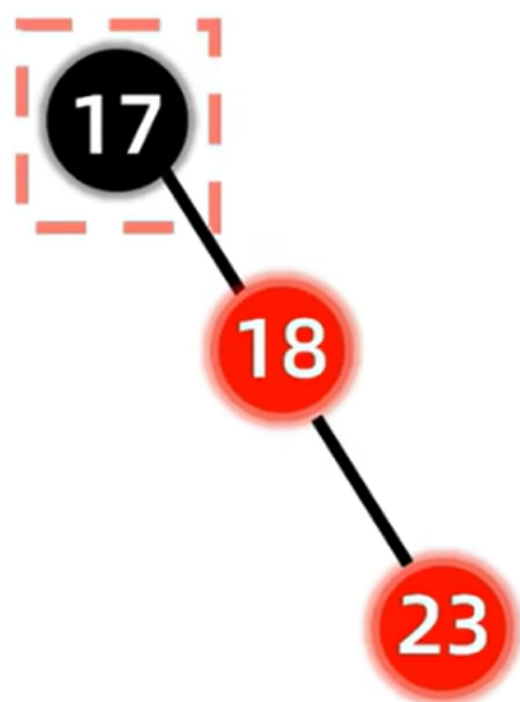


•

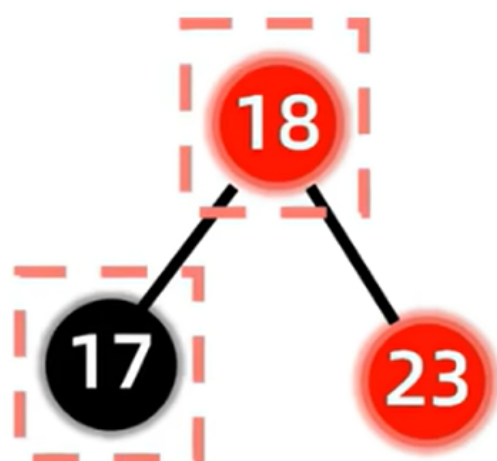
加微信: WandD2277



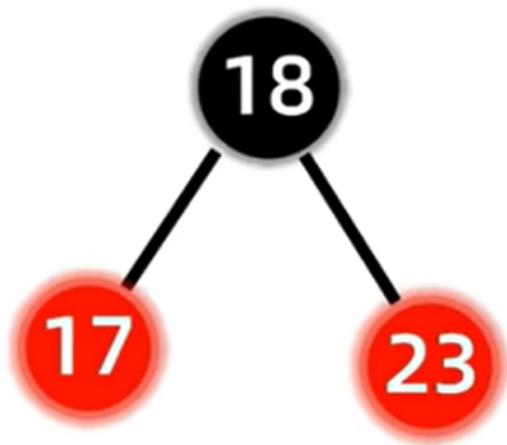
黑色叔叔



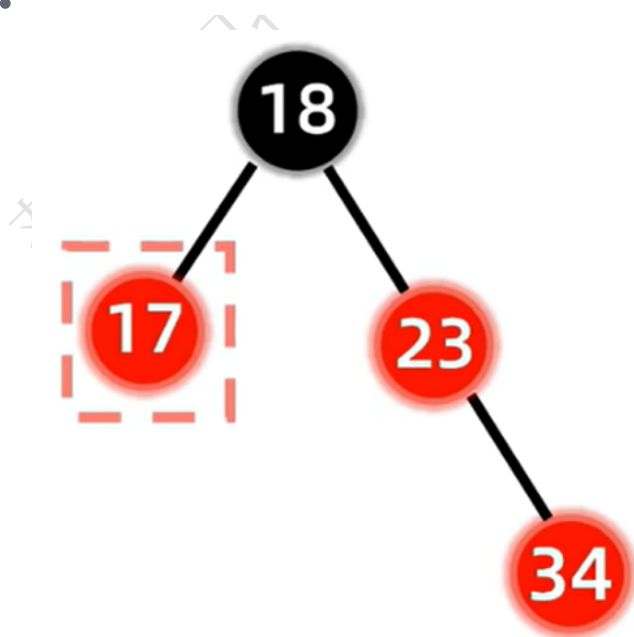
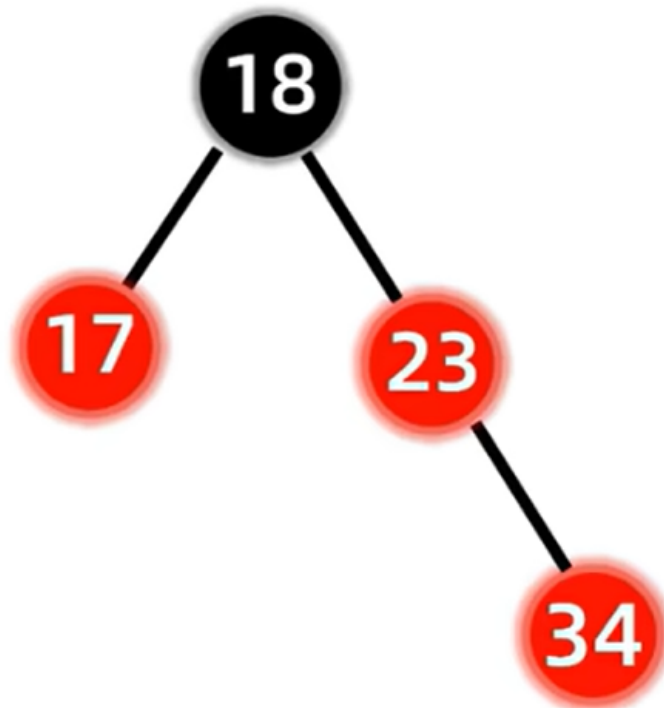
黑色叔叔
RR型
左旋



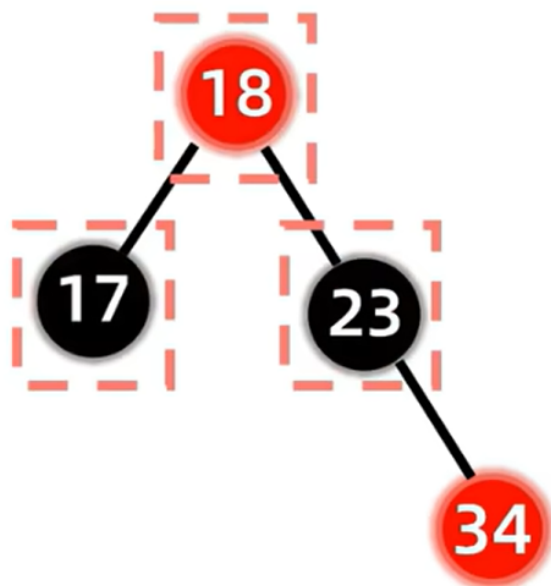
黑色叔叔
RR型
左旋
变色



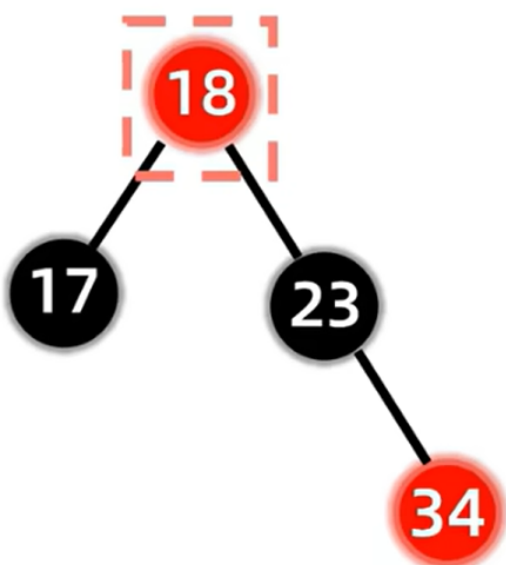
- 插入 34:



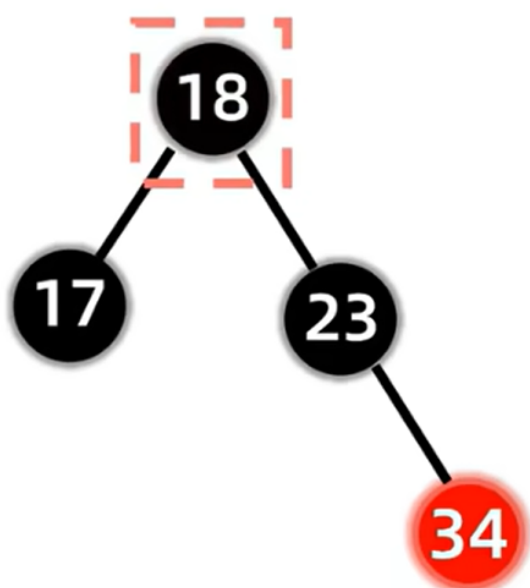
红色叔叔



红色叔叔
叔父爷变色

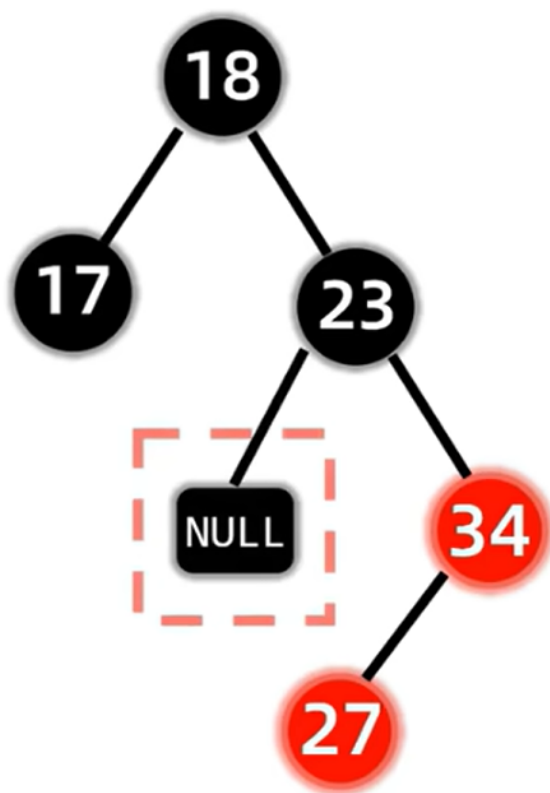
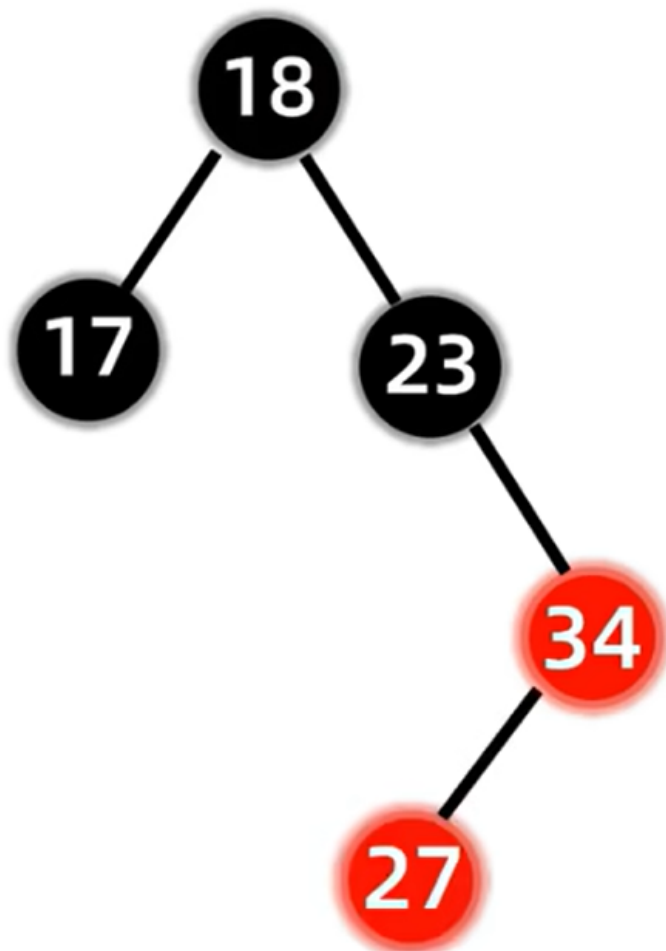


红色叔叔
叔父爷变色
爷爷变插入结点

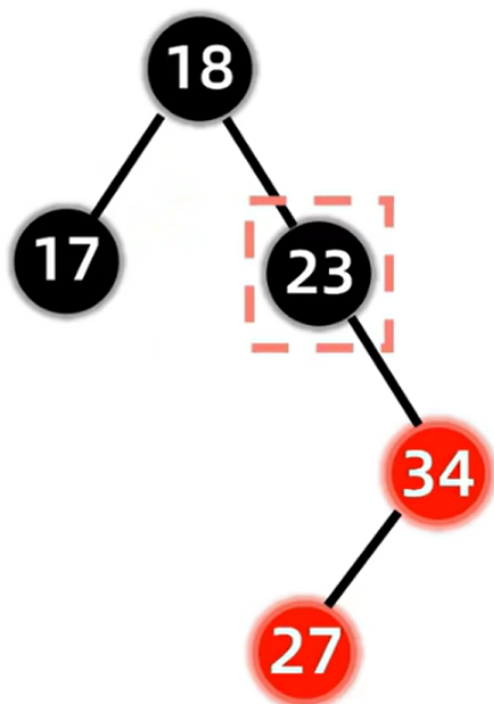


红色叔叔
叔父爷变色
爷爷变插入结点

• 插入 27:



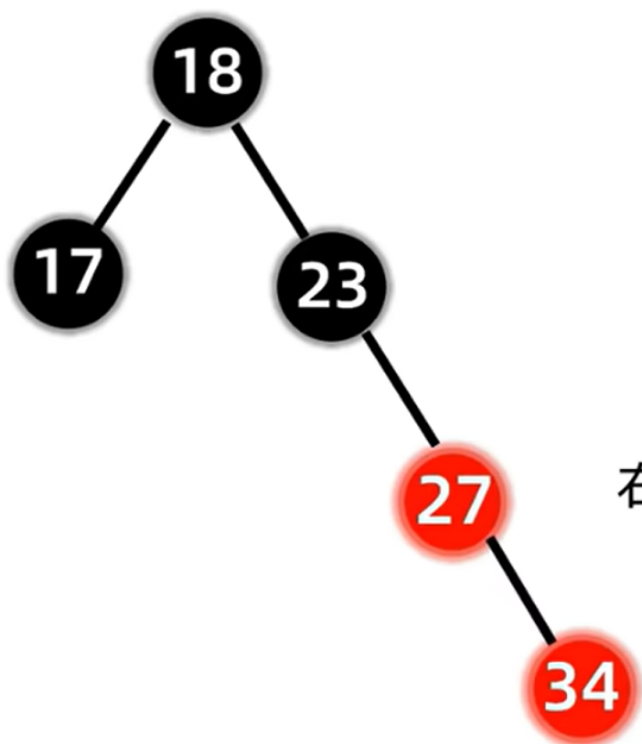
黑色叔叔



黑色叔叔

RL型

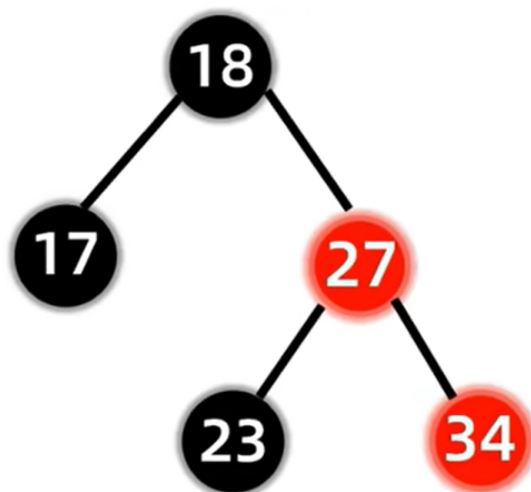
右旋右孩子, 然后左旋



黑色叔叔

RL型

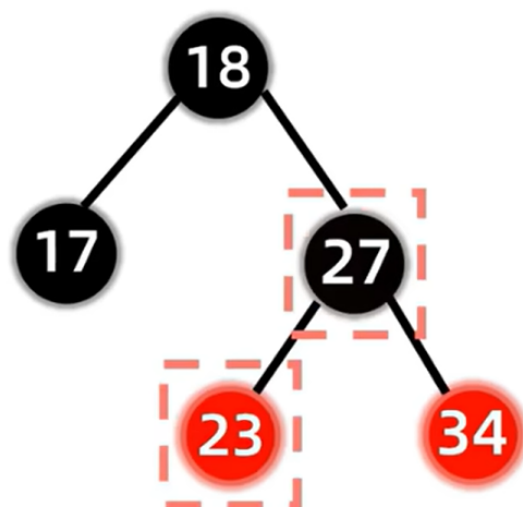
右旋右孩子, 然后左旋



黑色叔叔

RL型

右旋右孩子, 然后左旋



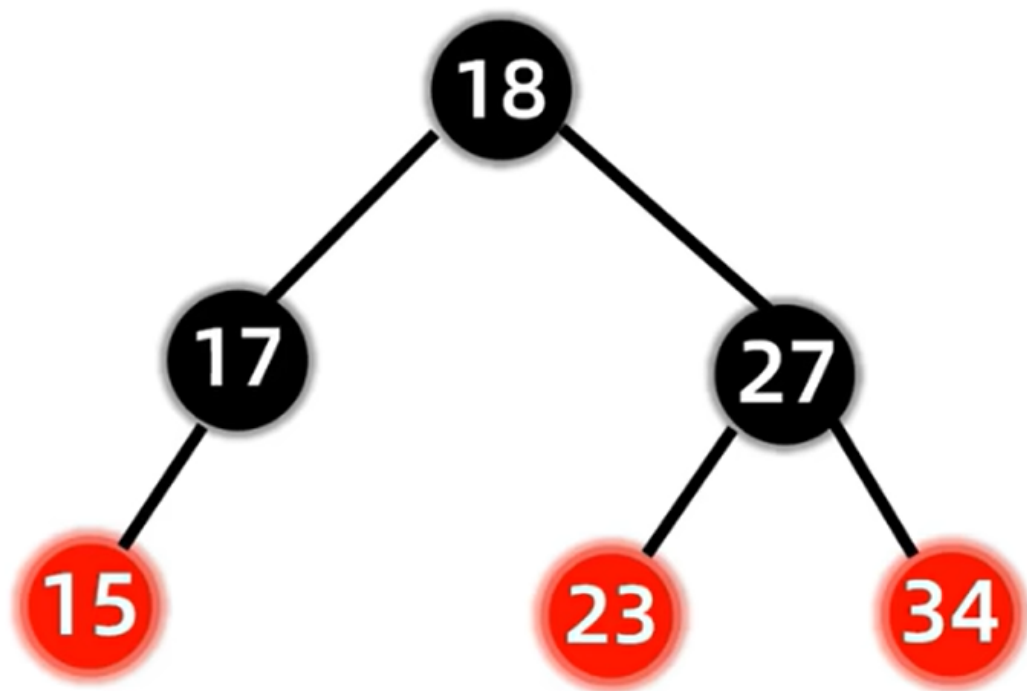
黑色叔叔

RL型

右旋右孩子, 然后左旋

变色

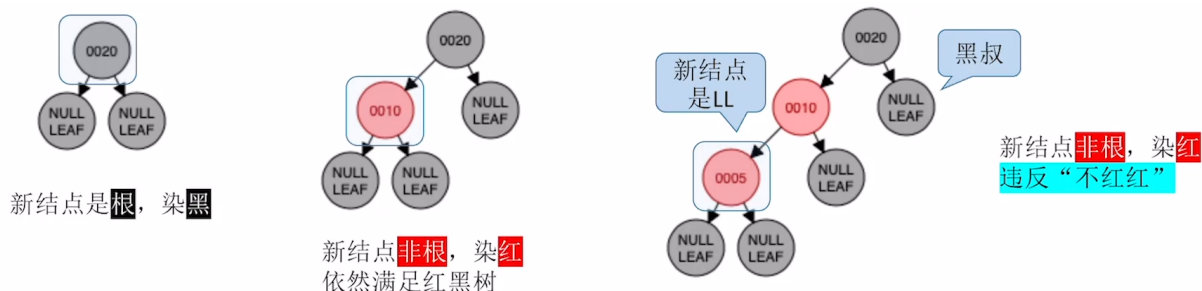
- 插入 15:



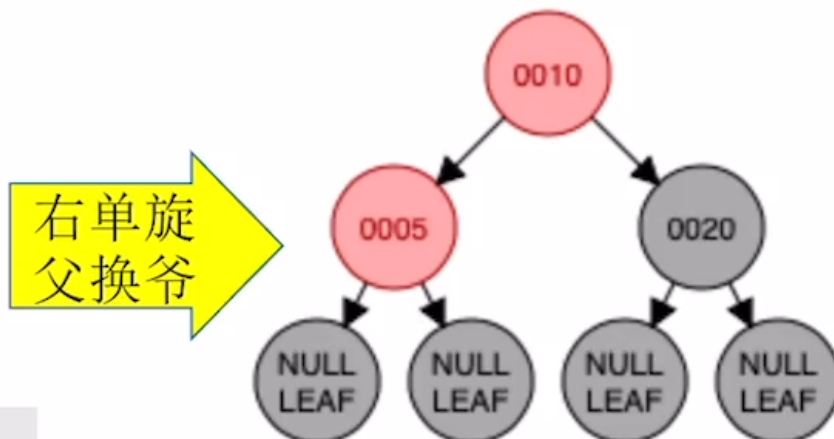
插入举例

- 数据：20, 10, 5, 30, 40, 57, 3, 2, 4, 35, 25, 18, 22, 23, 24, 19, 18

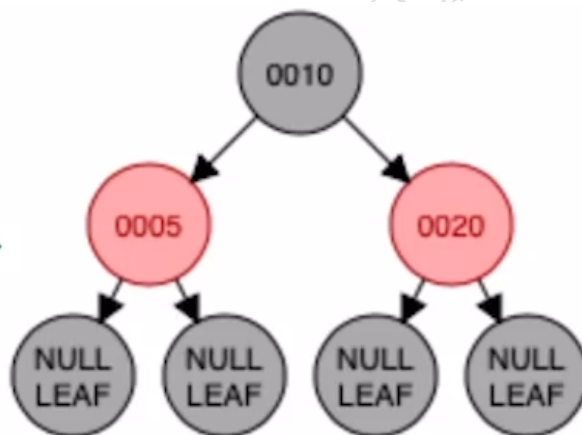
- 插入：



- 因为是 LL 型。所以：右单旋，父换爷 + 染色
 - 右单旋后：



- 然后需要染色：



26.5.3 红黑树的删除

基础性质

- 1. 红黑树删除操作的时间复杂度为 $O(\log_2 n)$;
- 2. 在红黑树中删除结点的处理方式和“二叉排序树的删除”一样;
- 3. 按【2】删除结点后， i 可能破坏“红黑树特性”，此时需要调整结点颜色、位置，使其再次满足“红黑树特性”;