

数据结构与算法

平衡树

主讲：赵海燕

北京大学信息科学技术学院
“数据结构与算法” 教学组

国家精品课 “数据结构与算法”
<http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/>

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6，“十一五”国家级规划教材

平衡树 (Balanced Tree)

- Definition
 - A *tree* where no *leaf* is *much farther* away from the *root* than any other leaf.
- Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

Different Balancing Schemes

- Red-black tree: also known as symmetric binary B-tree, 因结点分红、黑两色而得名 (1972年由Rudolf Bayer发明, J. Guibas 和 Robert Sedgewick 1978年的一篇论文给出Red-black tree的命名)
- AVL Tree: 得名于发明者名字的首字母缩写, Adelson-Velskii & Landis (1962)
- Top-down 2-3-4 tree: 结点分成2-、3-、4-叉而得名
- Splaying (priority tree)
-

红黑树

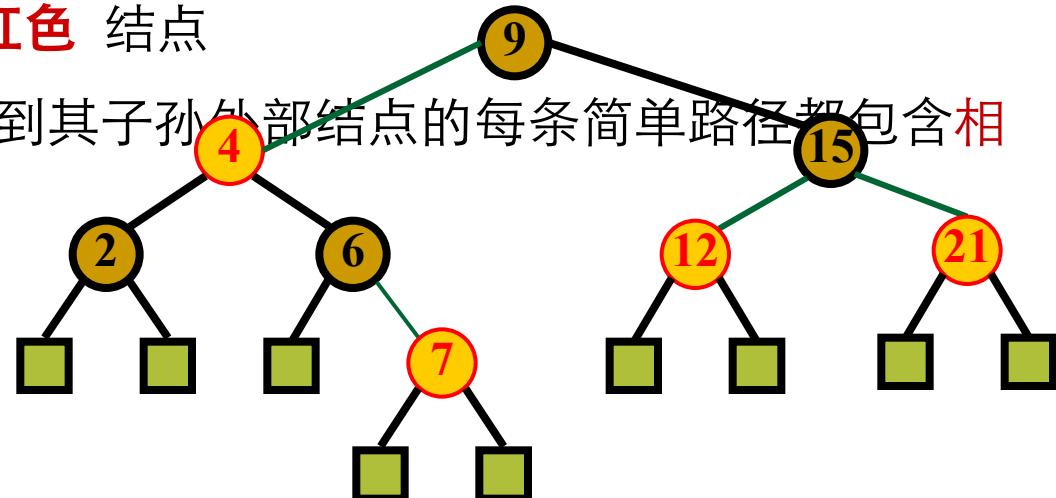
「红黑树」

- 红黑树定义
 - red-black tree, 简称RB-tree
- 红黑树相关性质
- 红黑树的操作
 - 结点插入算法
 - 结点删除算法

「红黑树」

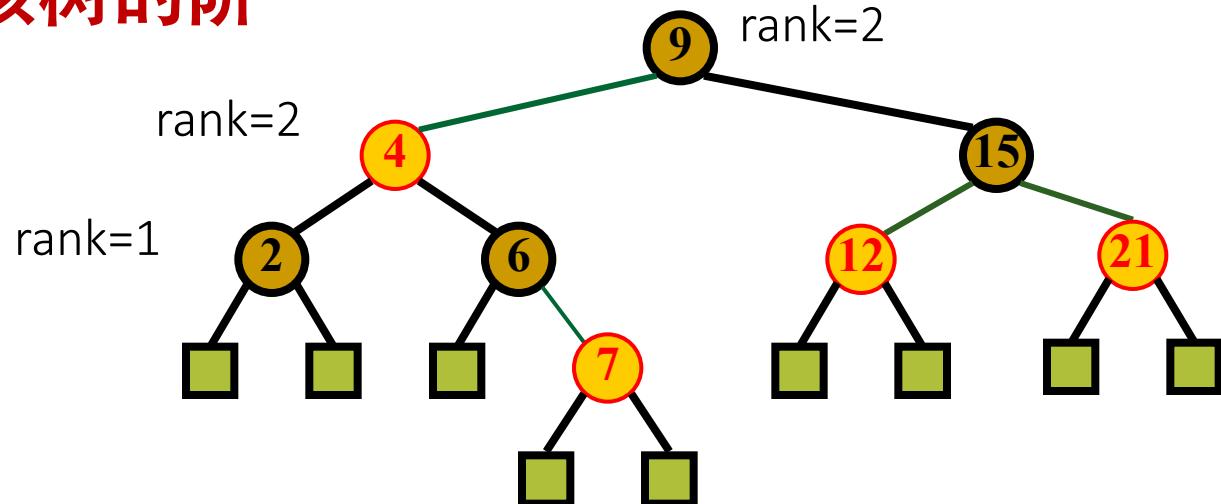
■ 平衡的**扩充**二叉搜索树

- 颜色特征：结点或 **红色** 或 **黑色**
- 根特征：根结点永远是“黑色”的；
- 外部特征：扩充外部叶结点都是空的“黑色”结点；
- 内部特征：**红色** 结点的两个子结点都是“黑色”的
 - ◆ 不允许两个连续的 **红色** 结点
- 深度特征：任何结点到其子孙外部结点的每条简单路径都包含**相同数目的“黑色”结点**

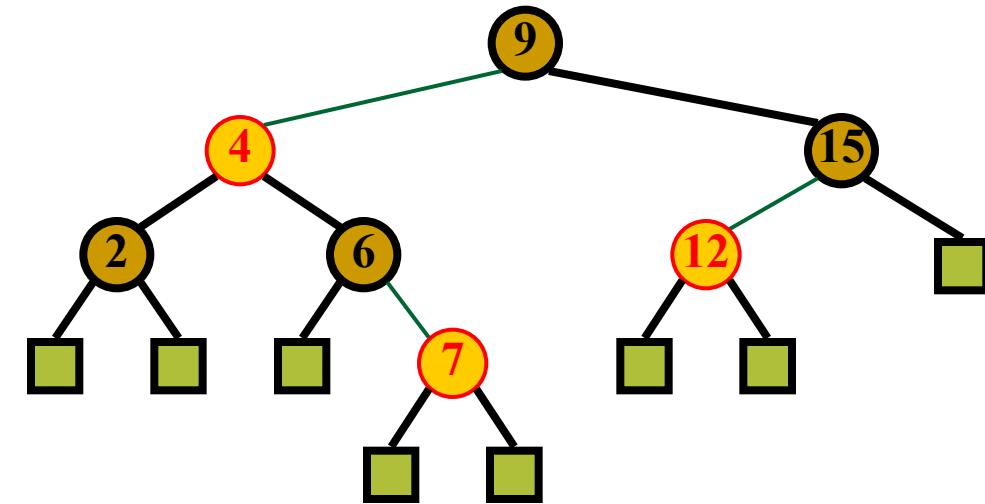


「红黑树的阶

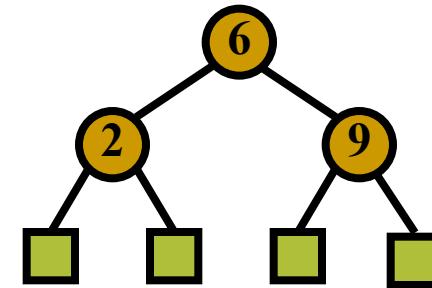
- 结点X的阶 (rank, 也称“黑色高度”)
 - 从该结点到外部结点的路径中黑色结点的数量
 - 不包括X结点本身，包括叶结点
- 外部结点的阶是零
- 根的阶称为该树的阶



「红黑树的性质」



1. 红黑树是**满二叉树**
 - 空叶结点也看作结点
2. 阶为 k 的红黑树路径长度
 - 从根到叶的**简单路径长度**: 最短是 k , 最长是 $2k$
 - 即**树高**: 最小是 $k+1$, 最高是 $2k+1$
3. 阶为 k 的红黑树的内部结点
 - **最少**是一棵**完全满二叉树**
 - 内部结点数最少是 $2^k - 1$



「红黑树的性质

4. 具有 n 个内部结点的红黑树的最大树高为
 $2 \log_2 (n+1)+1$

证明：

- 设红黑树的阶为 k , 树高为 h
- 由性质2 得 $h \leq 2k+1$
即 $k \geq (h-1) / 2$
- 由性质3 得 $n \geq 2^k - 1$
即 $n \geq 2^{(h-1)/2} - 1$
- 由此可得 $h \leq 2 \log_2 (n+1)+1$

「红黑树上的更新操作

- 包括
 - 插入
 - 删除

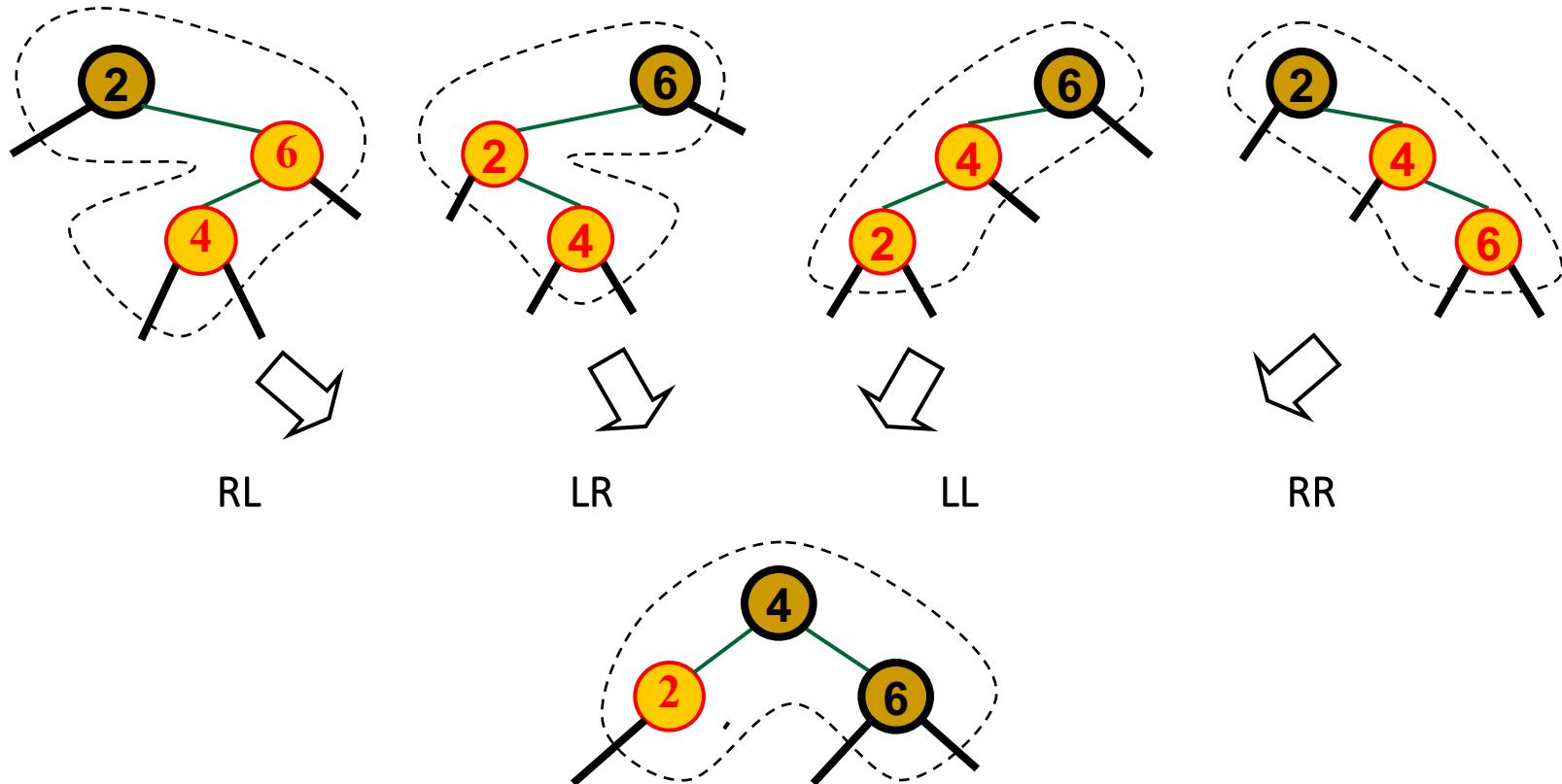
满足红黑树的定义，保持红黑树的性质

「红黑树的重构

- 一系列旋转 (rotation) 的局部操作来解决
 - LL型和 RR 型
 - ◆ 单旋转 (*single rotation*)
 - LR型和RL型
 - ◆ 双旋转 (*double rotation*)

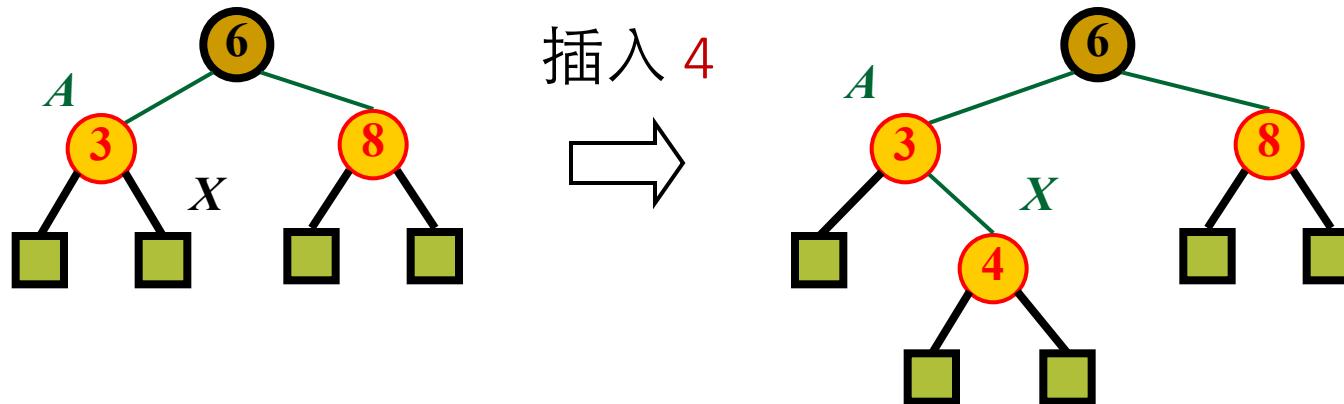
「重构的 4 种形式

- 原则：保持BST的中序性质



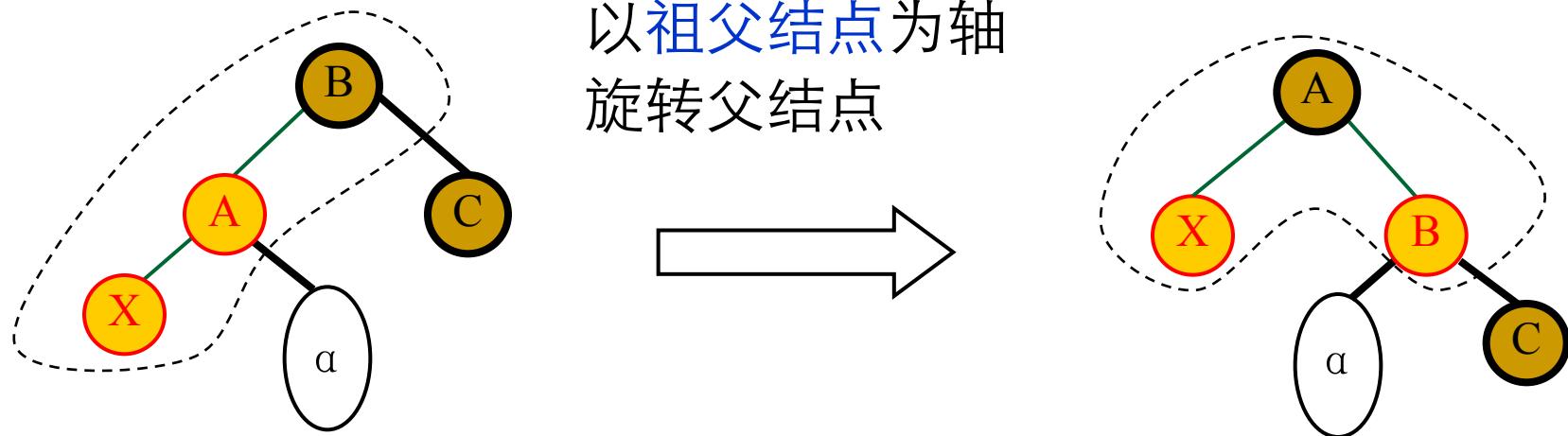
「红黑树的插入算法」

- 调用BST的插入算法
 - 把新结点着为红色 (why?)
- 根据父结点的着色
 - 若为黑色，则算法结束
 - 否则，双红调整



插入算法调整：1 重构

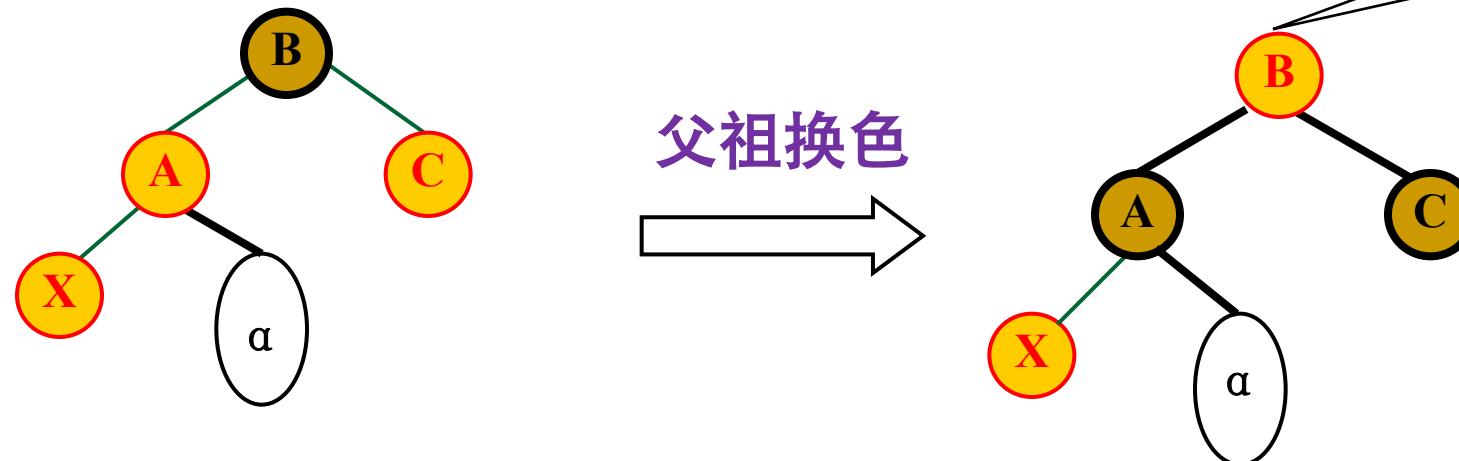
- 情况 1：新增结点X的叔父结点是黑色



- 每个结点的阶都保持原值，调整完成

插入算法调整：2 换色

- 情况2：新增结点X的叔父结点也是红色

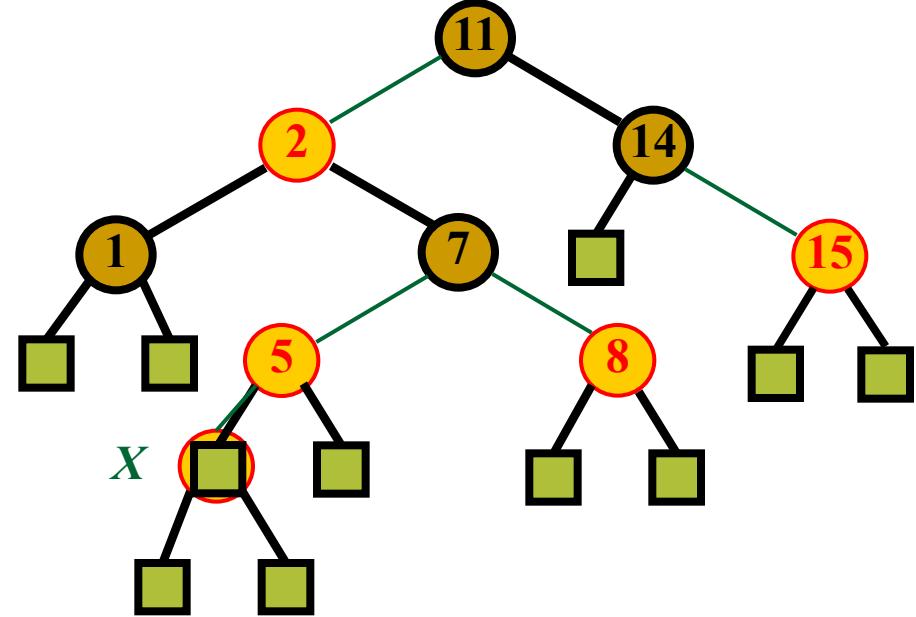


- 需要继续检查平衡

2 换色示例：插入4

■ 情况2 红红冲突

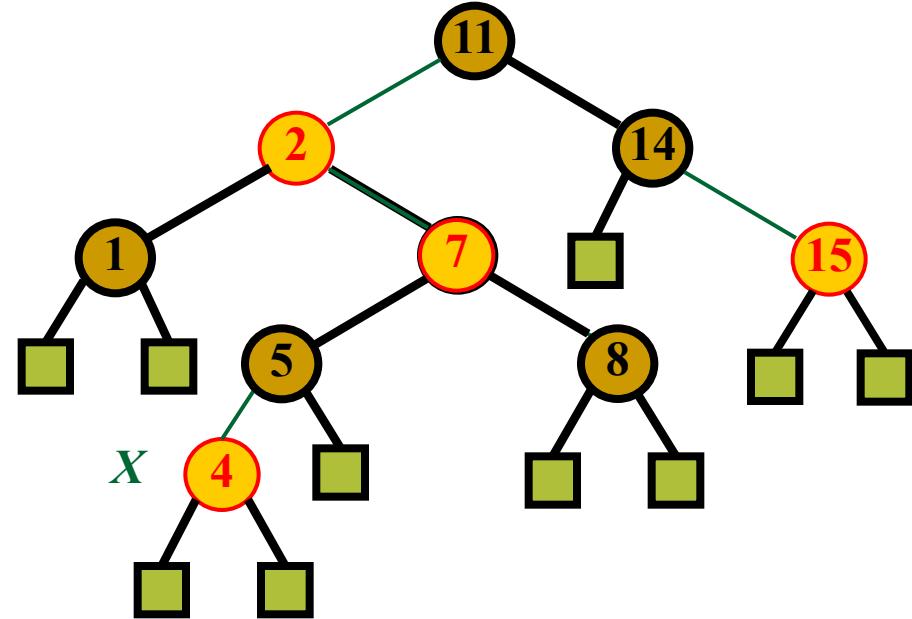
- 父和叔父也是红
- 父祖换色(color flip)



2 换色示例：插入4

■ 情况2 红红冲突

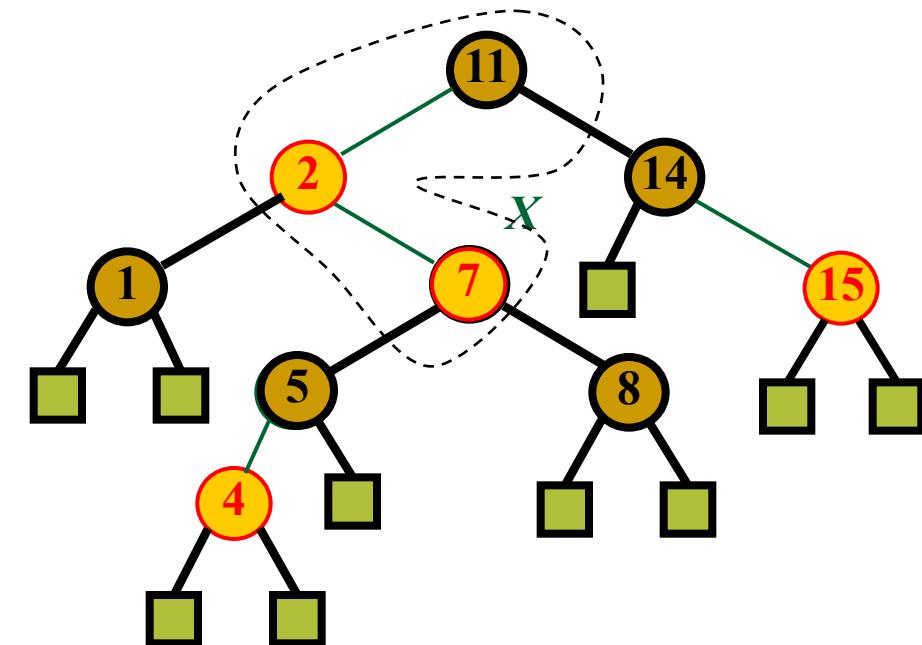
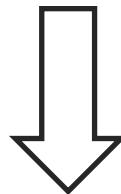
- 父和叔父也是红
- 父祖换色(color flip)



「2 换色示例：插入4

■ 情况2 红红冲突

- 父和叔父也是红
- 父祖换色(color flip)



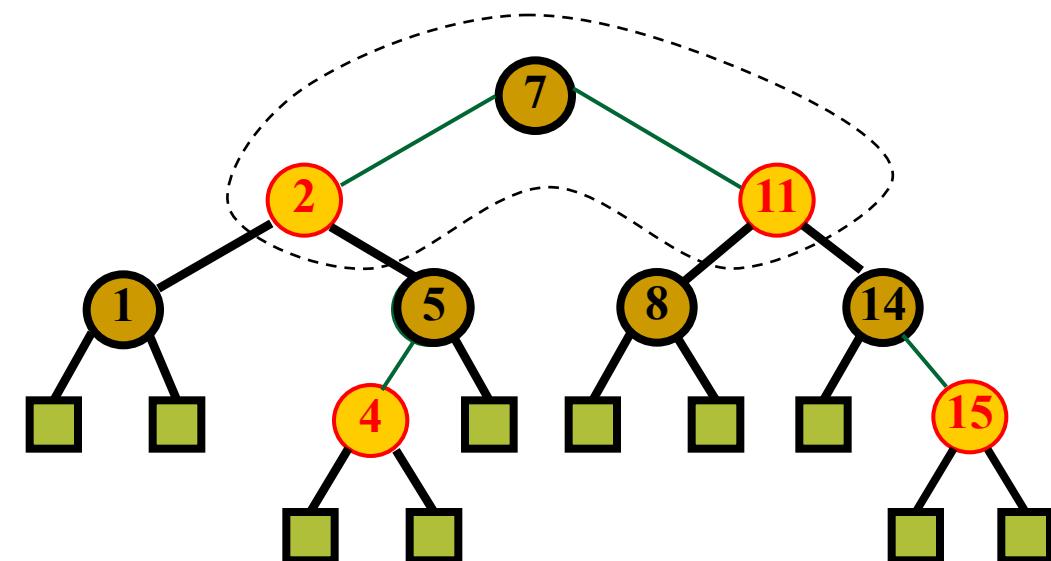
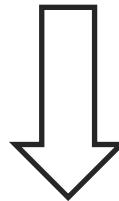
■ 情况1 红红冲突

- 叔父是黑
- 重构

「2 换色示例：插入4

■ 情况2 红红冲突

- 父和叔父也是红
- 父祖换色(color flip)



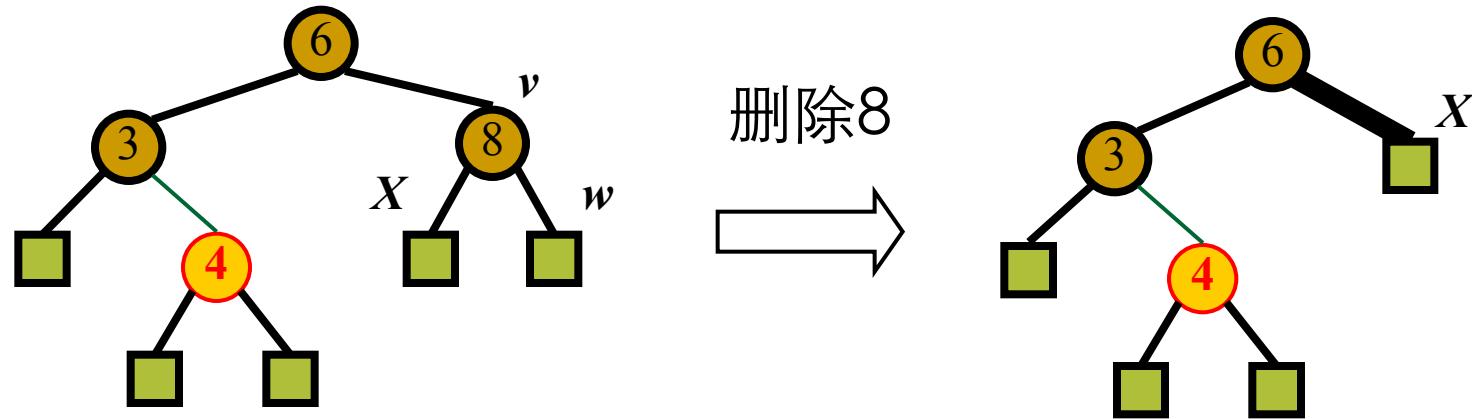
■ 情况1 红红冲突

- 叔父是黑
- 重构

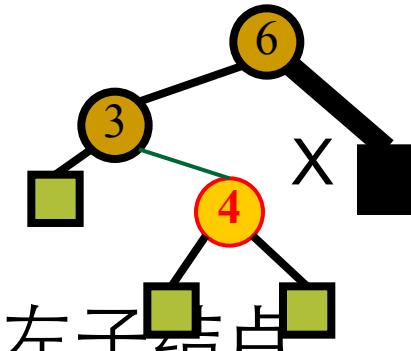
「删除算法」

- 调用 BST 的 **删除算法**
 - 若待删除的结点有一个以上的外部空指针，则直接删除
 - 否则，在右子树中找到其后继结点进行值交换（着色不变）删除
- 假设 v 是被删除的内部结点， w 是被删外部结点， x 是 w 的兄弟
 - 若 v 或者 x 是红色，则把 x 标记为黑色即可
 - 否则， x 需要标记为 双黑，根据其兄弟结点 c 进行重构调整

「红黑树删除示例」



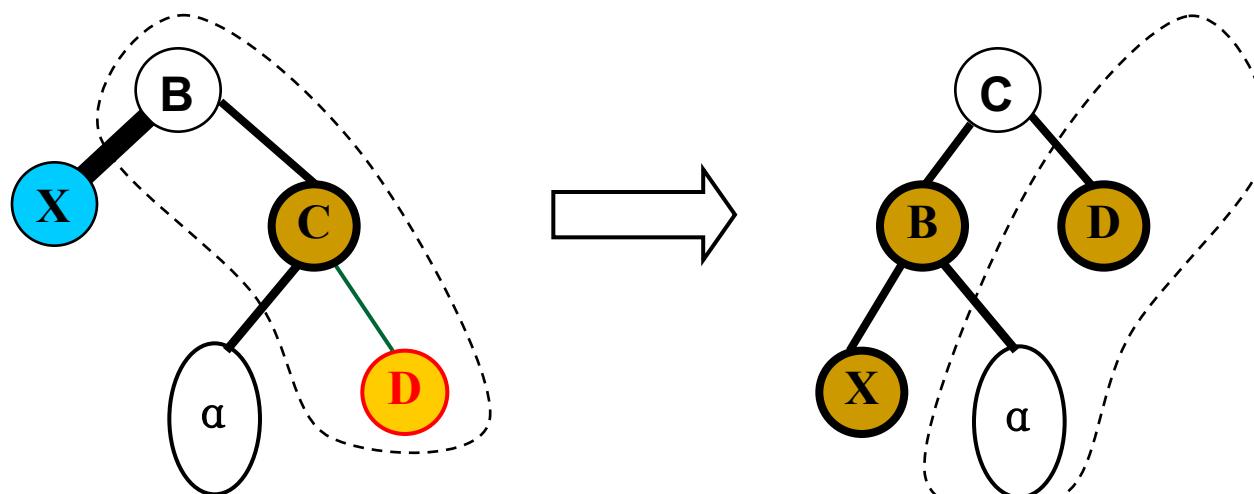
「双黑结点的调整」



- 调整有赖于双黑 X 的兄弟 C , 假设 X 是左子结点
(X 为右子结点时对称处理)
 - 情况#1: C 是黑色, 且子结点有红色
 - ◆ 重构, 完成操作
 - 情况#2: C 是黑色, 且有两个黑子结点
 - ◆ 换色
 - ◆ 父结点B原为红色, 可能需要从B继续向上调整
 - 情况#3: C 是红色
 - ◆ 转换状态
 - ◆ C 转为父结点, 调整为 情况1 或 2 继续处理

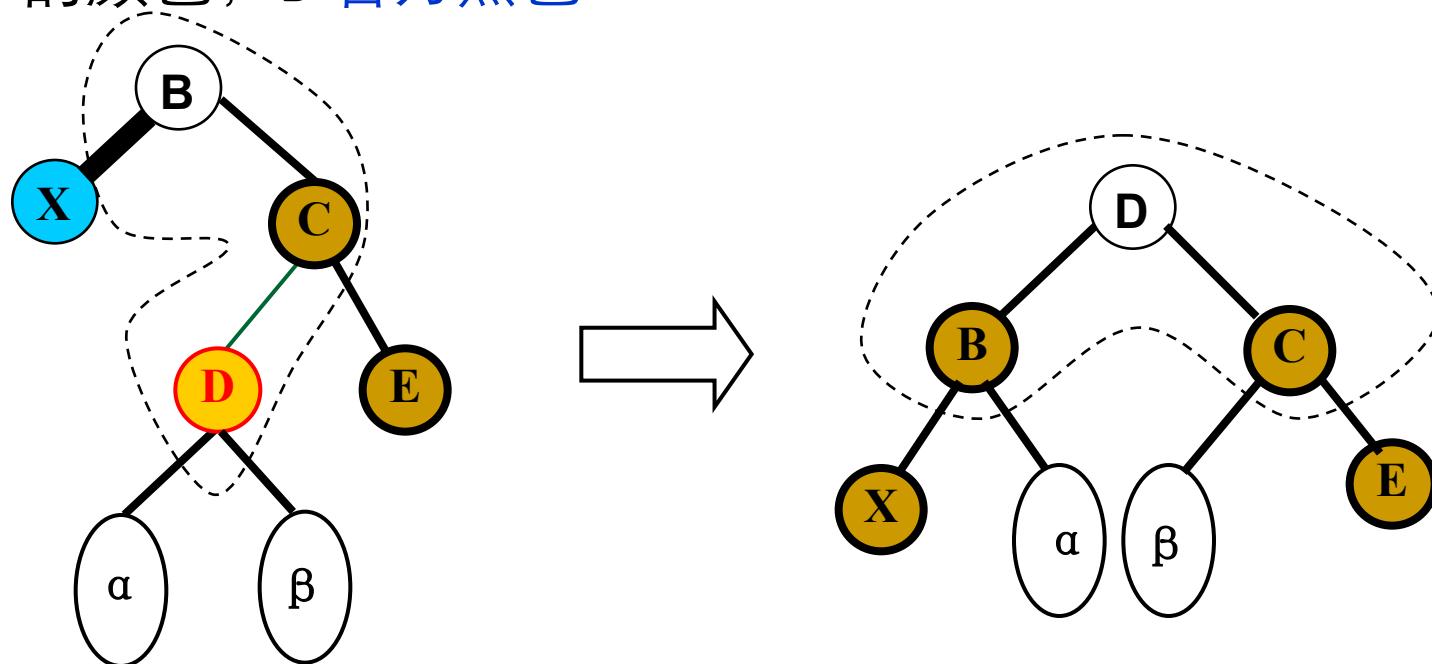
情况 1(a) 重构(侄子结点红八字)

- 兄弟 C 为黑色、且有红子结点，X 与侄子呈八字形
 - 将兄弟结点 C 提上去
 - C 继承原父结点的颜色
 - 然后将 B 着为黑色，D 着为黑色，其他颜色不变即可



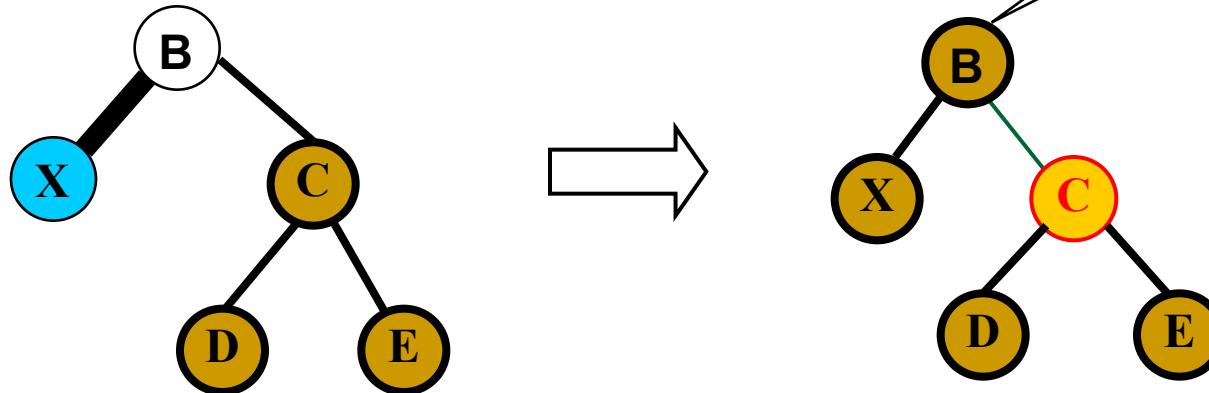
情况 1(b) 重构 (侄子结点红同边顺)

- 兄弟 C 为黑色、且有 **红** 子结点，X 与侄子呈 **同边顺**
 - 双旋：将 D 结点**旋转**为 C 结点的父结点，D **继承** 原子根 B 的颜色，B **着为**黑色



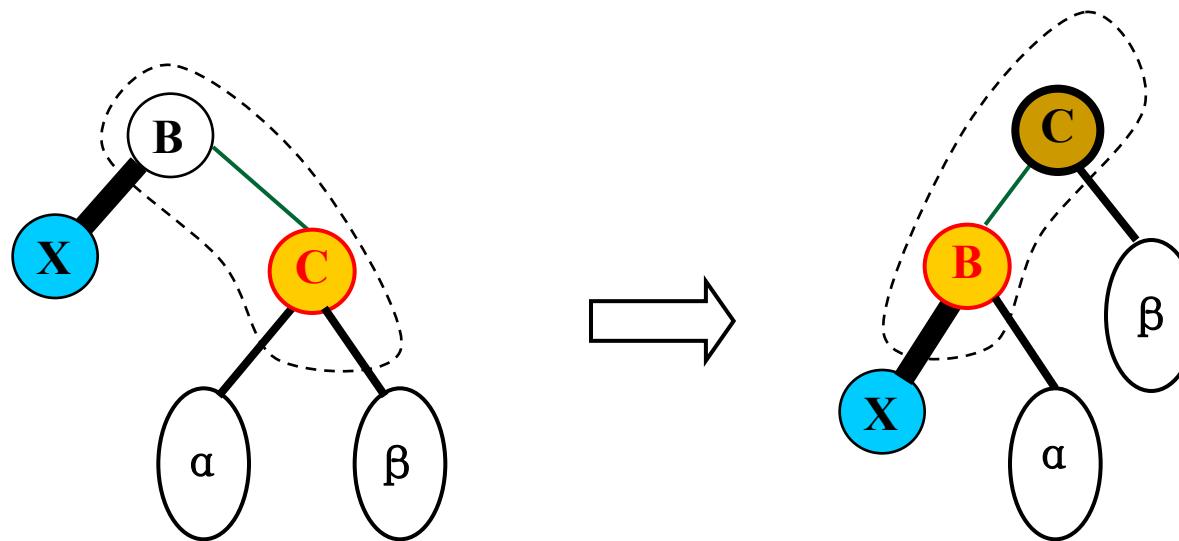
情况 2：兄弟黑且有两个黑子

- 换色：将 C 着红色，B 着黑色
 - 若 B 原为红色，则算法结束
 - 否则，对B继续作“双黑”调整



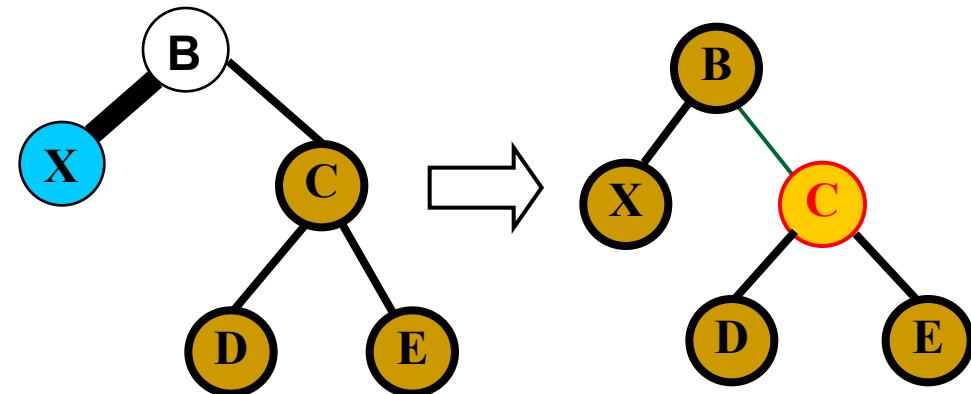
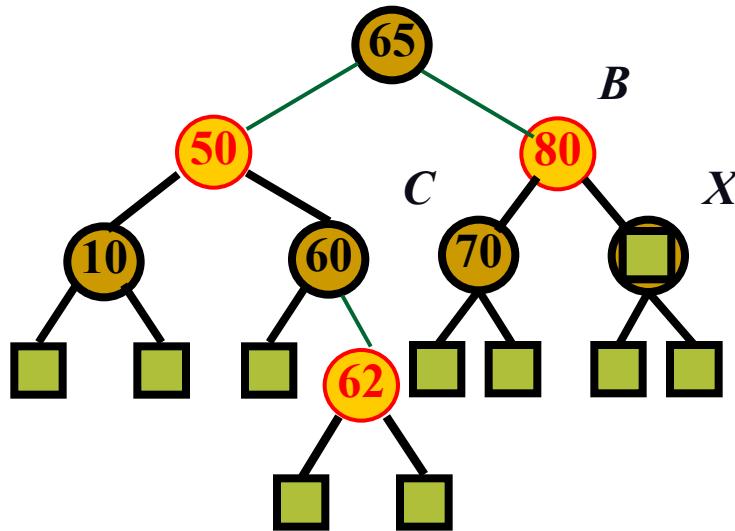
情况 3：兄弟 C 红色

- 旋转
- X 结点仍是“双黑”结点，转化为前面2种情况



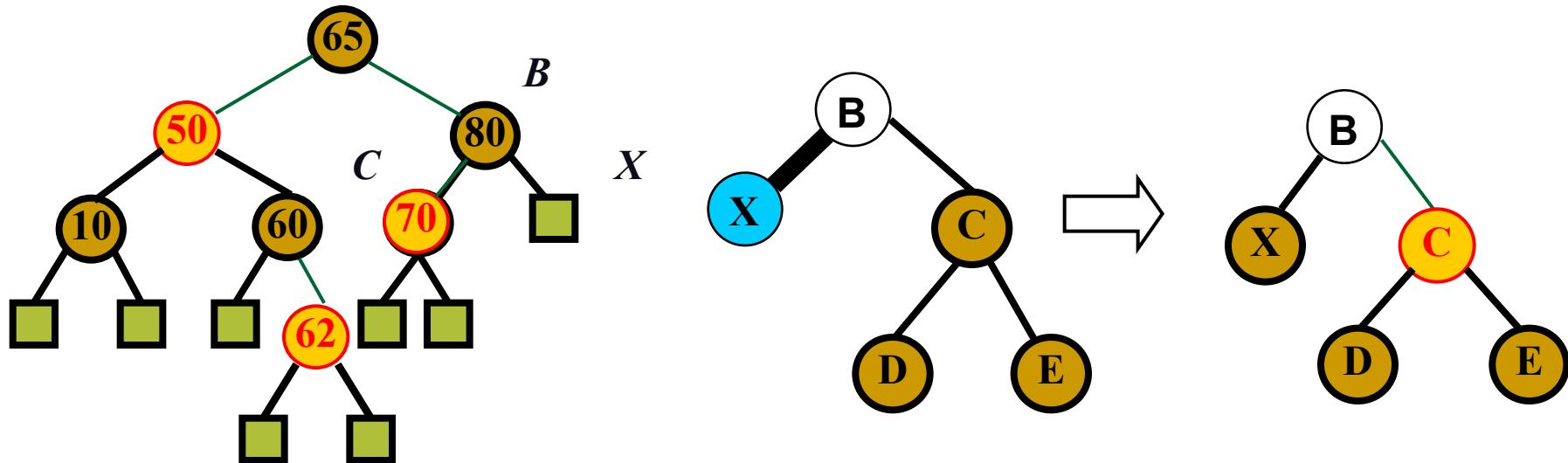
删除示例：删除90

- 当前结点变为80的 **右黑** 叶结点
- 兄弟 C 是黑色，且有两个黑色子结点 → 情况2



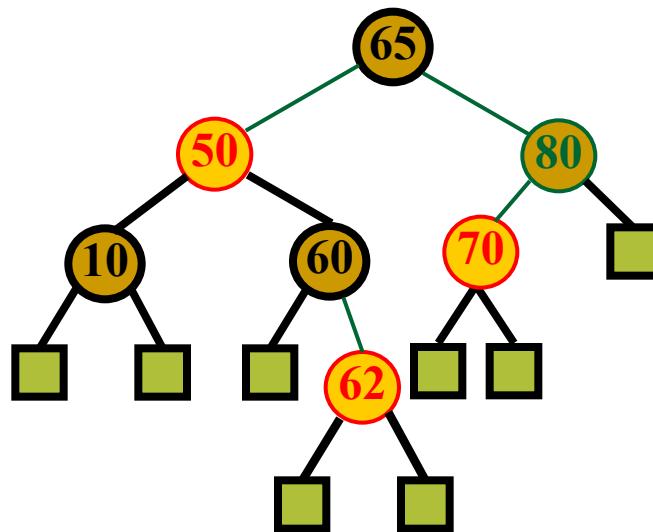
删除示例：删除90

- 当前结点变为80的右黑叶结点
- 兄弟C是黑色，且有两个黑色子结点 → 情况2
 - 换色



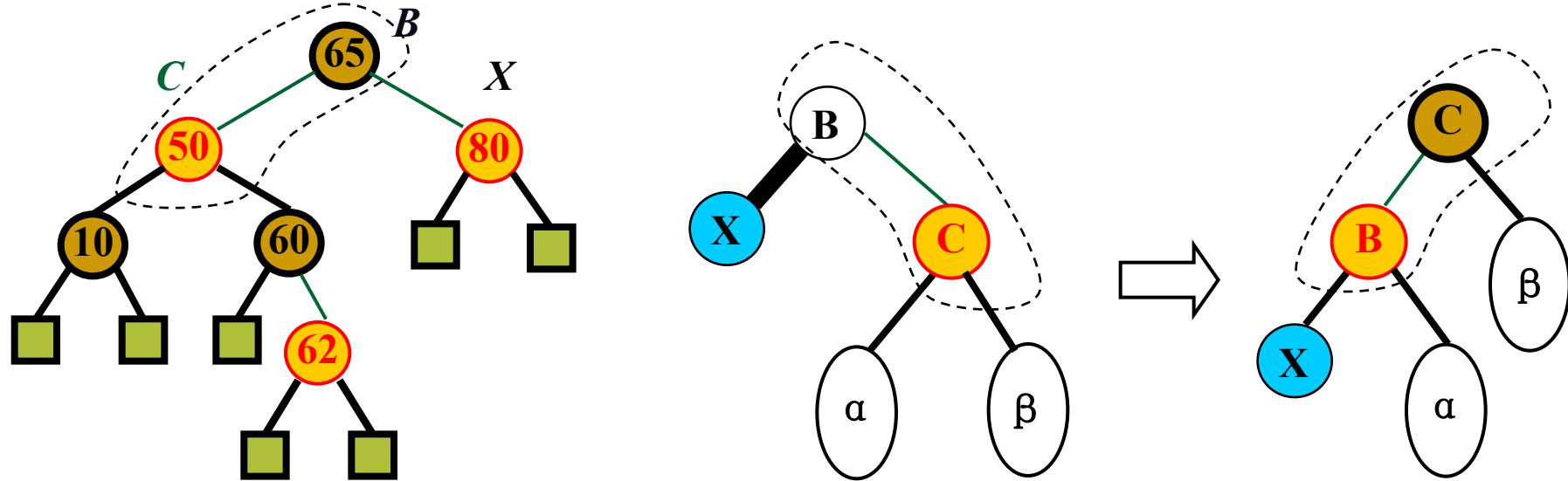
删除示例：删除70

- 红结点，不要调整



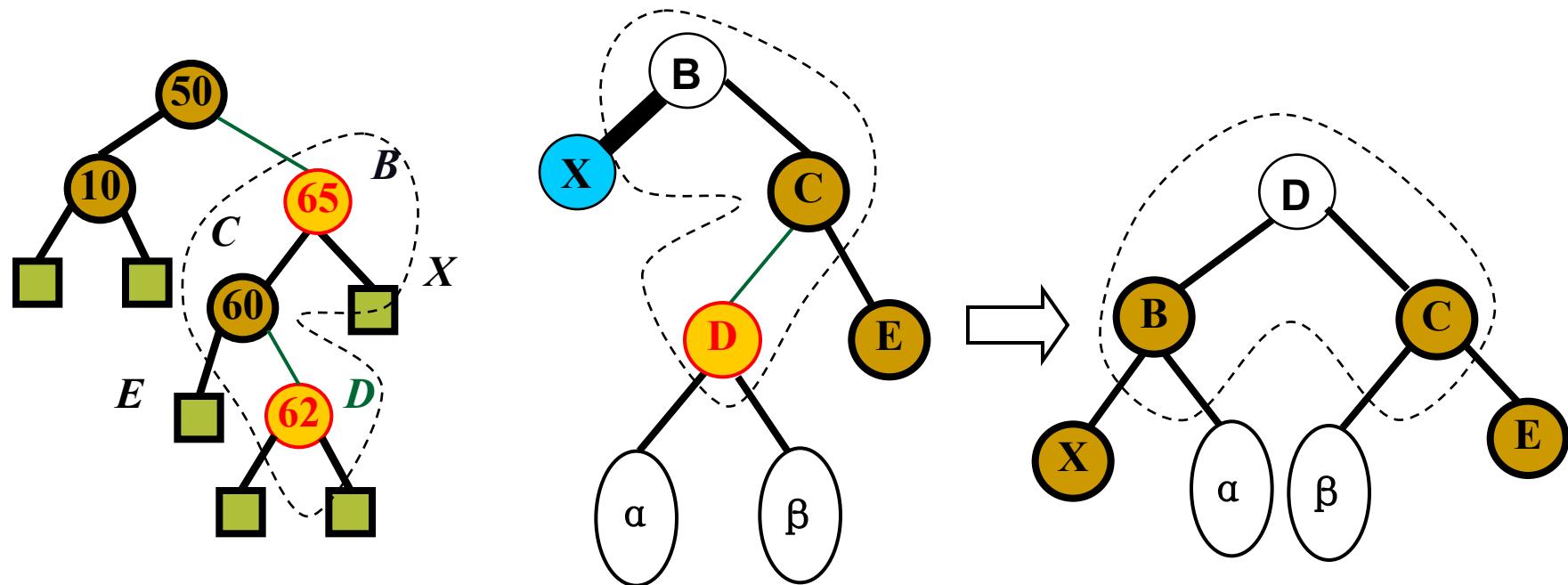
删除示例：删除80

- 当前结点 X 变为 65 的右黑叶结点
- C 是红色：情况3
 - 状态转换



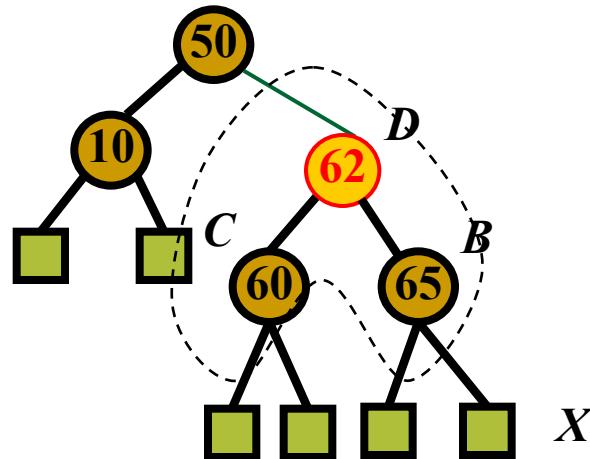
删除示例：删除80

- 转换后的兄弟C为黑色，且左黑、右红 →
情况 1(b) 重构



删除示例：删除80

- 完成调整



「删除操作时间代价」

- 平均和最差检索 $O(\log_2 n)$
 - 自底向根的方向调整
- 红黑树构造
 - (数据, 左指针, 右指针, 颜色, 父指针)
- 自顶向下的递归**插入/删除**调整方法
 - (数据, 左指针, 右指针, 颜色)
 - 非递归, 记录回溯路径

「红黑树总结」

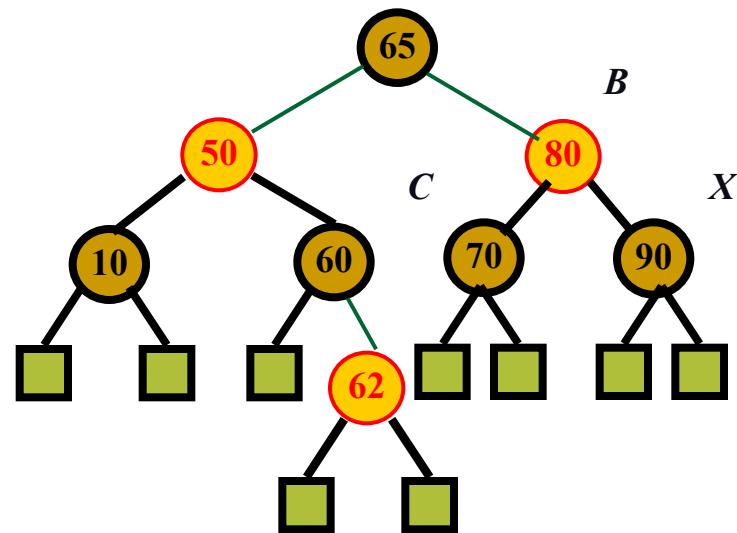
- 红黑树高度平衡
- 结点插入： **双红** 现象
 - 树重构
 - 换色、调整
- 结点删除： **双黑** 现象
 - 树重构
 - 换色（重着色）
 - 转换状态

2-3-4 树

■ 《算法导论》

- 假设将一棵红黑树的每个红结点“吸收”到其黑色父结点中，并将红结点的子女变为黑色父结点的子女（忽略关键字的变化）。当一个黑结点的所有红色子女都被吸收后
 - 其可能的度是多少？
 - ◆ 2、3、4
 - ◆ 即成为一棵2-3-4树（阶为4的B树）
 - 此结果树的叶结点深度怎样？
 - ◆ 就是RB的阶
 - ◆ 叶结点等高

The diagram illustrates a Red-Black tree transformation. A root node labeled 65 (black) has two children, both of which are red nodes labeled 50 and 80. Node 50 has two black children, 10 and 60. Node 80 also has two black children. The leaf nodes at the bottom are represented by green squares. A red circle highlights node 62, which is a red child of node 60. A green circle highlights node 60, which is a black parent of node 62. A label 'C' points to node 60, and a label 'B' points to node 80. This visualizes the process of absorbing red nodes into their black parents.



2-3-4 & RB-Tree

LLRB
Delete
Analysis

1. Represent 2-3-4 tree as a BST.
2. Use "internal" red edges for 3- and 4- nodes.



3. Require that 3-nodes be left-leaning.



2-3-4 & RB-Tree

Key Properties

- elementary BST search works
- easy-to-maintain 1-1 correspondence with 2-3-4 trees
- trees therefore have perfect black-link balance

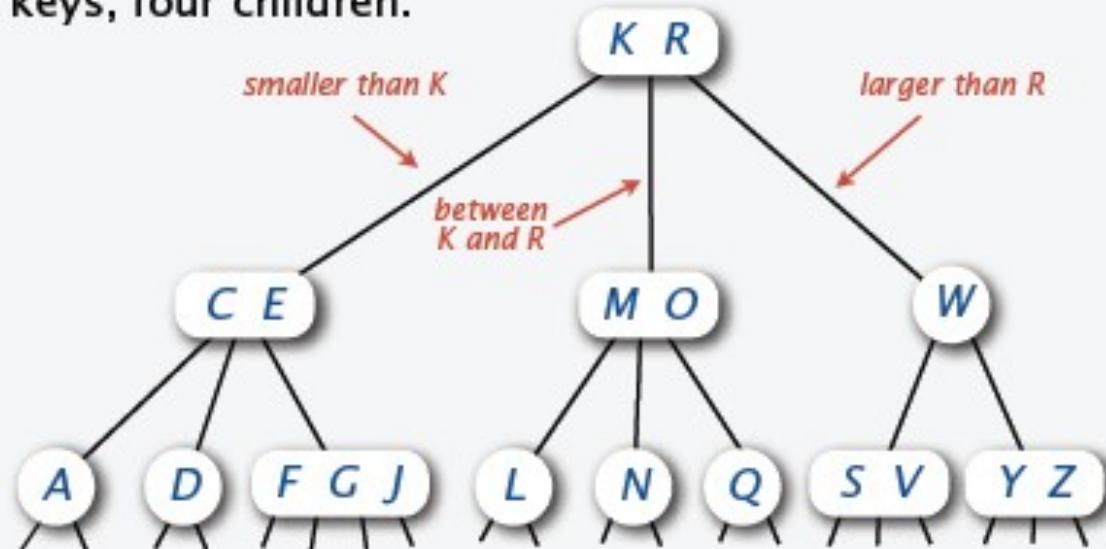


每个结点的颜色定义为 进入该点的边的颜色 → red-black tree

4阶B树

Allow 1, 2, or 3 keys per node.

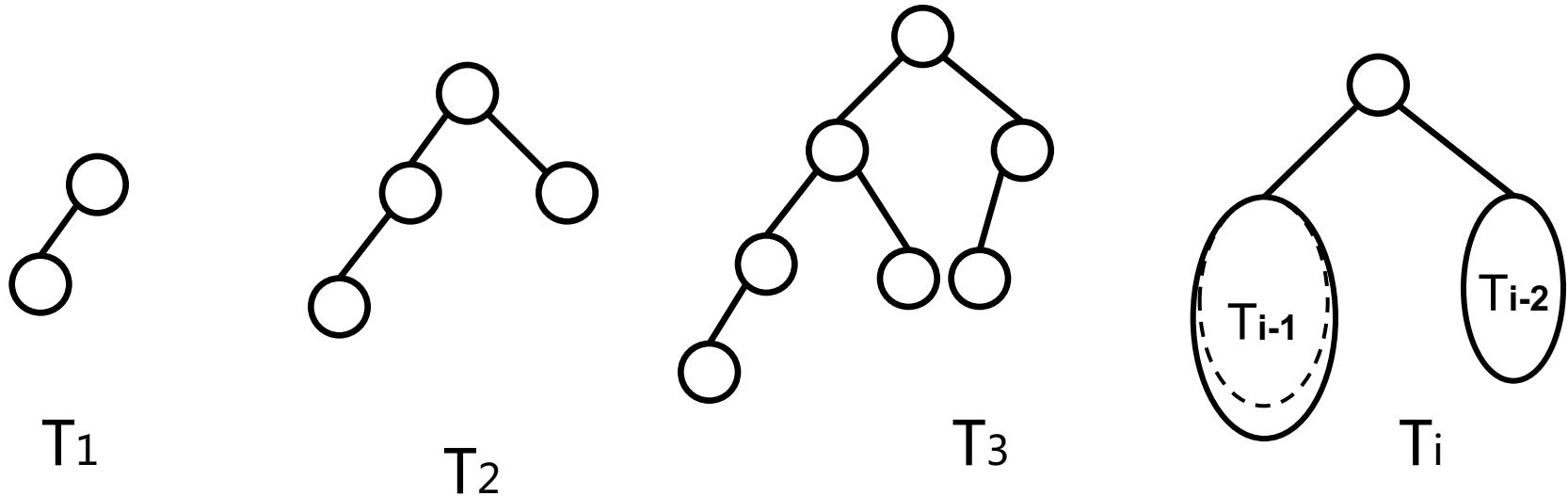
- 2-node: one key, two children.
- 3-node: two keys, three children.
- 4-node: three keys, four children.



AVL树

AVL Tree

- A (*balanced*) *binary search tree* with the additional *balance property* that, the *height* of the two subtrees (*children*) of *every* node differs by *at most one*



AVL 平衡因子

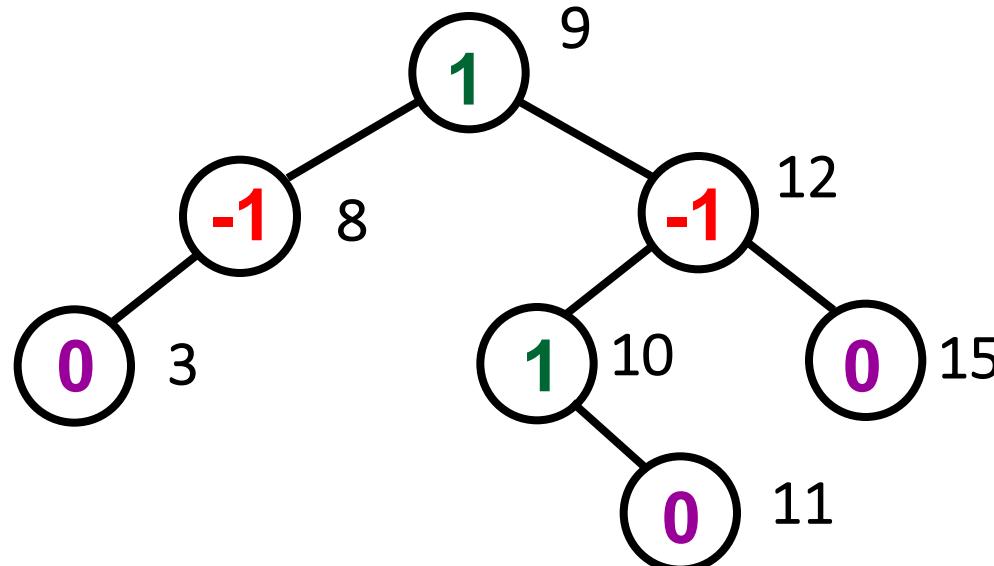
- Balance Factor $bf(x)$:

$$bf(x) = height(x_{rchild}) - height(x_{lchild})$$

i.e., $H_R - H_L$

□ 取值可为 $-1, 0, 1$

□ H_L vs H_R



「AVL树的性质

- 若 T 是一棵AVL树，则其左、右子树 T_L 、 T_R 也为 AVL树，并且 T_L 、 T_R 的高度 h_L 、 h_R 间有

$$|h_L - h_R| \leq 1$$

成立

- 一棵 空二叉树 是一个 AVL树
- 一棵具有 n 个结点的AVL树的高度为 $O(\log n)$

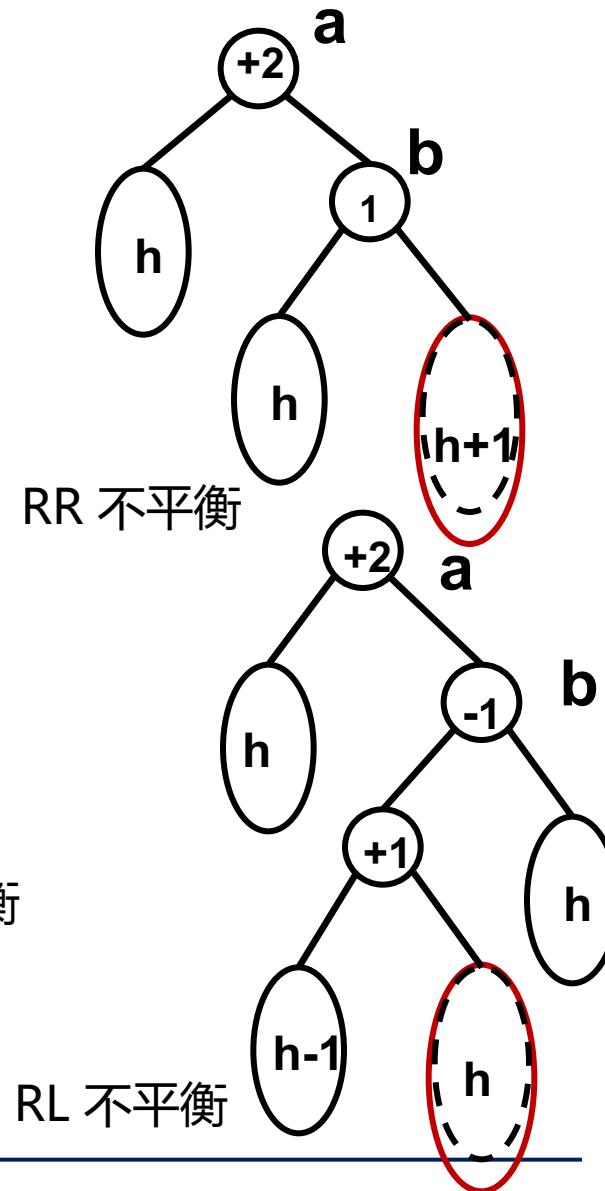
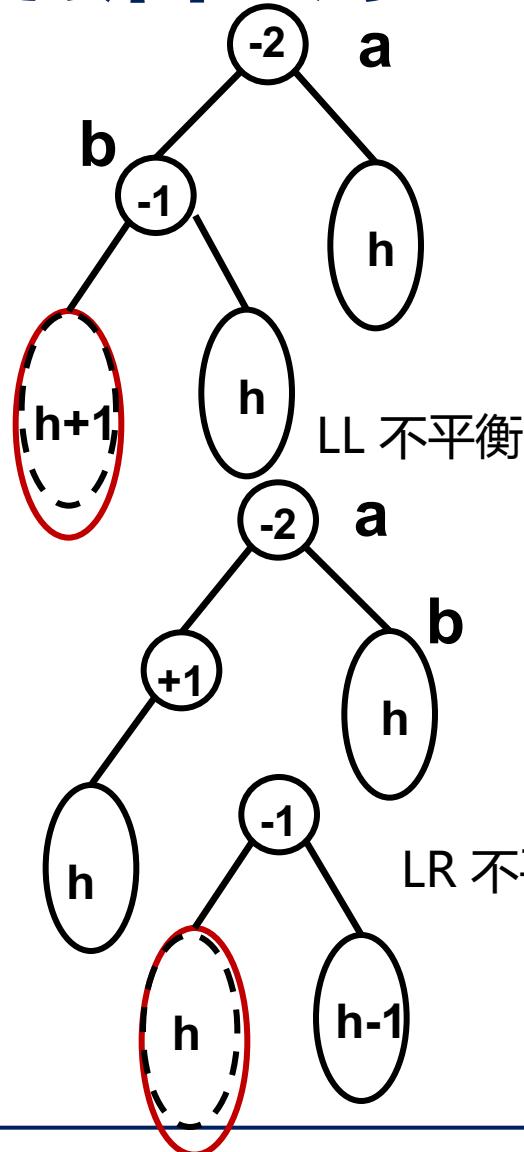
AVL树的更新和调整

- 插入和删除操作可能引起 AVL树结点 A 的 平衡因子 bf 改变而不再满足AVL的定义，分**四种**情况：
 - A本来**左重**： $A.bf = -1$ ， 插入一个结点导致其变为 **-2**
 - ◆ **LL型**： 插入到 A 的左子树的左子树
 - **左重 + 左重**， $A.bf$ 变为 **-2**
 - ◆ **LR型**： 插入到A 的左子树的右子树
 - **左重 + 右重**， $A.bf$ 变为 **-2**
 - A本来**右重**： $A.bf = 1$ ， 插入新结点使其变为 **2**
 - ◆ **RL型**： 导致不平衡的结点为 A 的右子树的左结点
 - ◆ **RR型**： 导致不平衡的结点为 A 的右子树的右结点

LL型和**RR型**是对称的， **LR型**和**RL型**是**对称的**

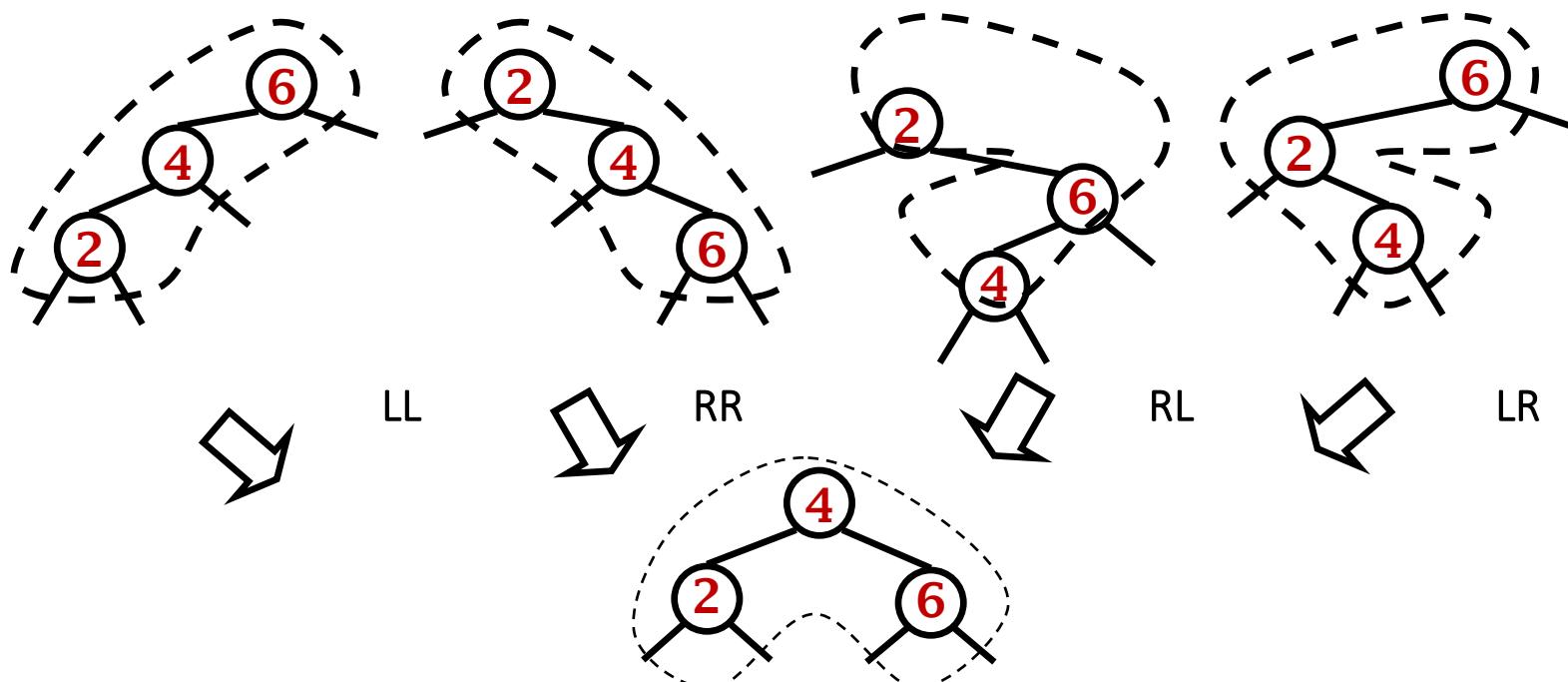
AVL树的更新和调整

- 不平衡的结点一定在根结点与新加入/删除结点之间的路径上，且其平衡因子只能是2或者-2
 - 2表示它在插入前的平衡因子是1
 - 2表示它在插入前的平衡因子是-1



AVL树的重构

- 一系列旋转 (rotation) 的局部操作来解决
 - LL型和RR型：单旋转 (single rotation)
 - LR型和RL型：双旋转 (double rotation)

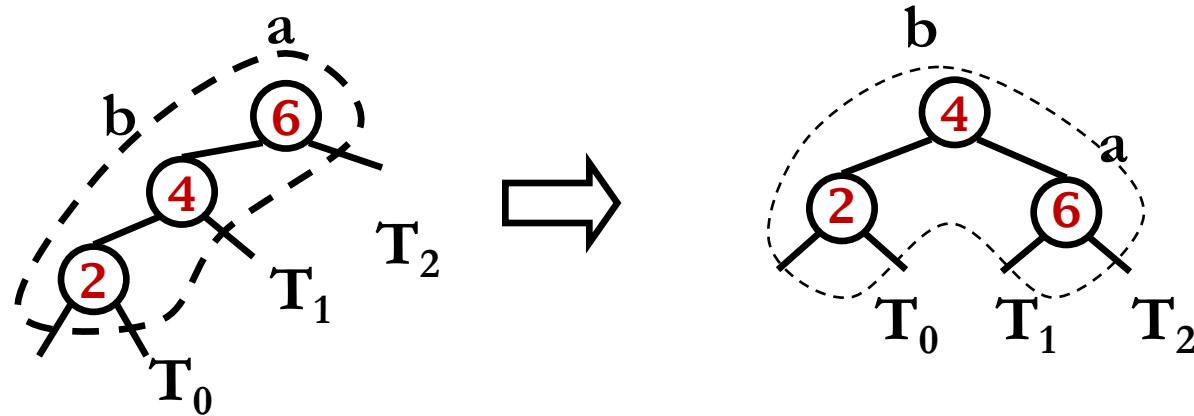


「AVL树的重构过程

1. 先从新加入/删除的结点 u 向根的方向往上搜索 BST，直到找到 距离 u 最近的不平衡祖先结点 X （平衡因子在插入/删除后变为 $2 \mid -2$ ）
 - 若直到根都未找到这样的祖先，则不做任何调整（已经平衡），返回
2. 然后根据 树的形态 进行 单旋转或双旋转

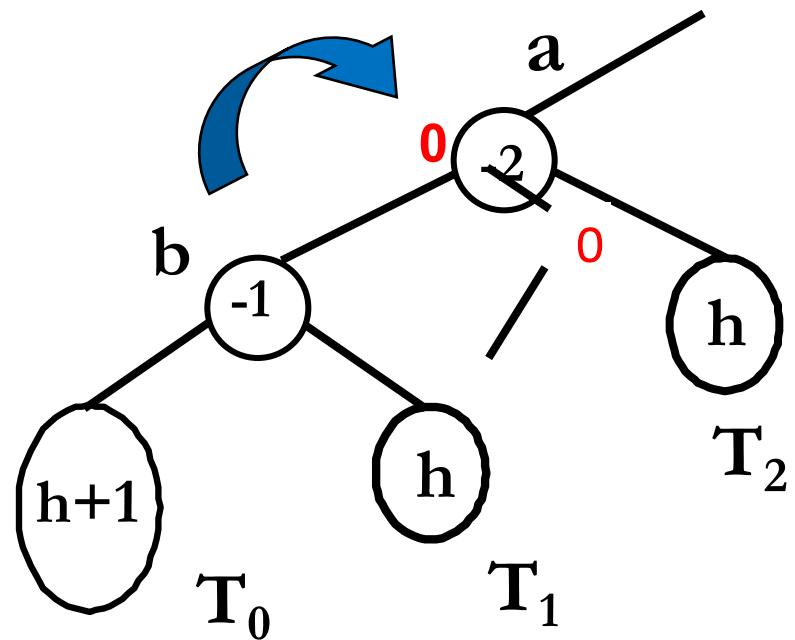
单旋转

- 调整过程只涉及到 a 和 b 两个结点；以 **LL型** 调整为例
 - b 代替 a 成为新根，a 作为 b 的右子结点，a 和 b 的子树根据 BST 的性质分别挂接到 b 和 a 结点下，保持中序周游序列为 $T_0 \ b \ T_1 \ a \ T_2$



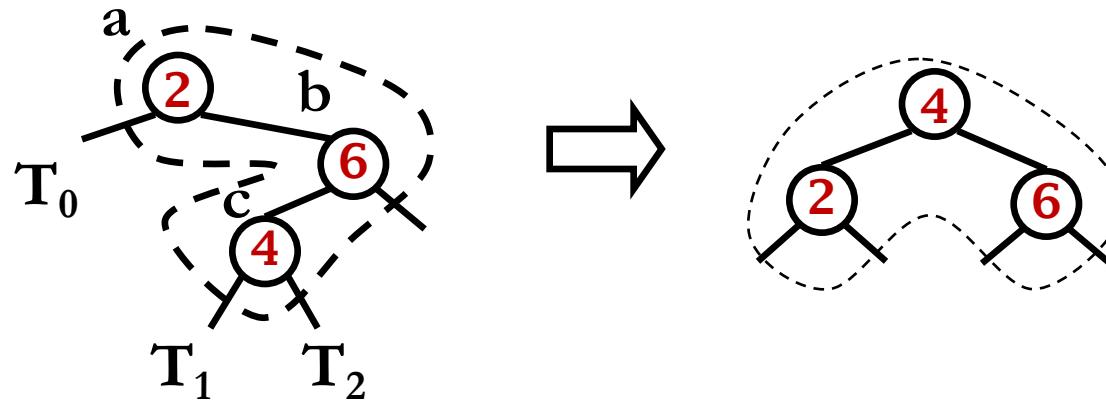
单旋转

- 以LL型调整为例



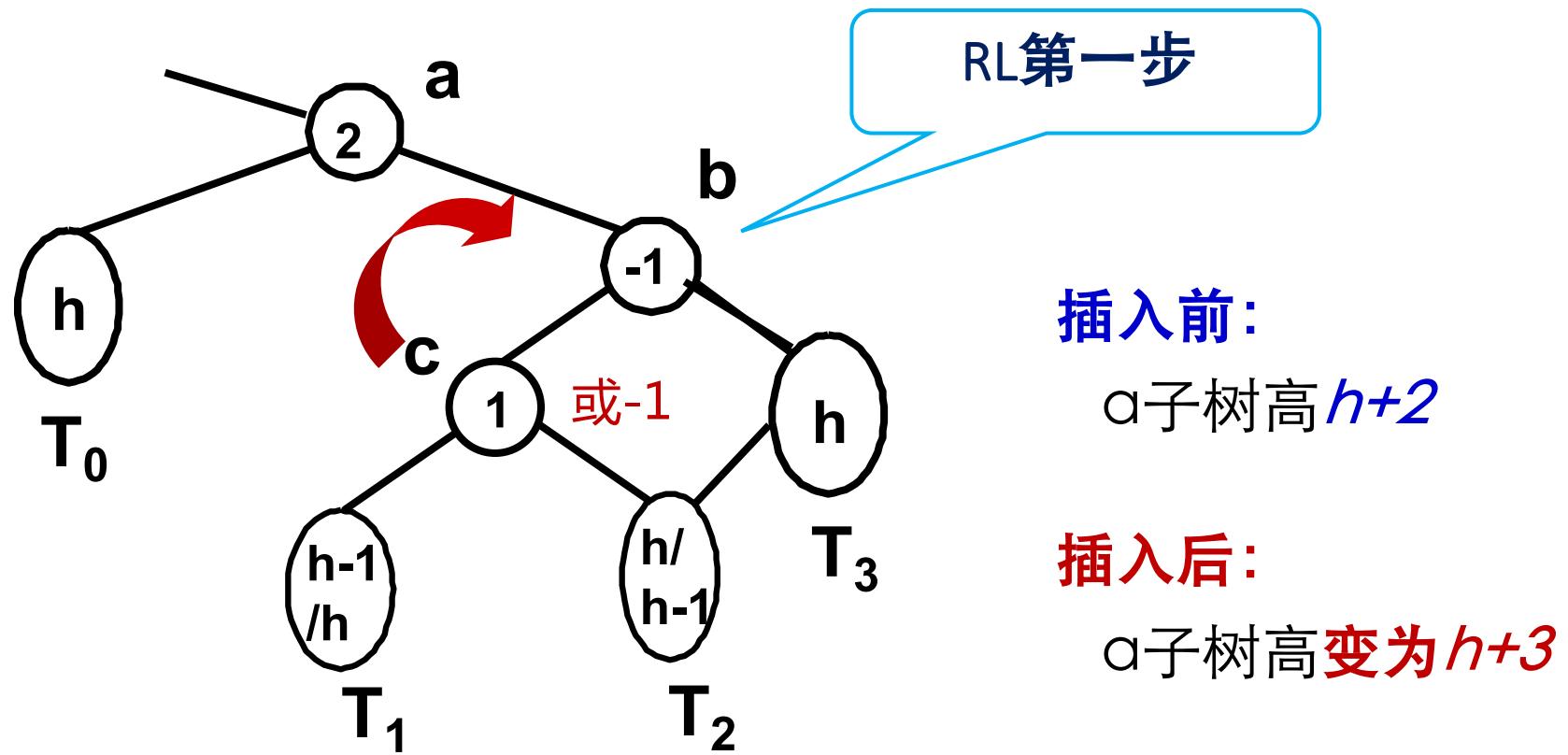
「双旋转」

- 涉及 a、b、c 结点，以 RL型 为例
 - 先作 (c, b) 右转，把 c 的右子 T_2 接到 b 上；
 - 再作 (a, c) 左转，把 c 的左子 T_1 接到 a 上



「RL型双旋转：1st Step」

- 先作(c, b) 右转，把 T_2 接到b上；

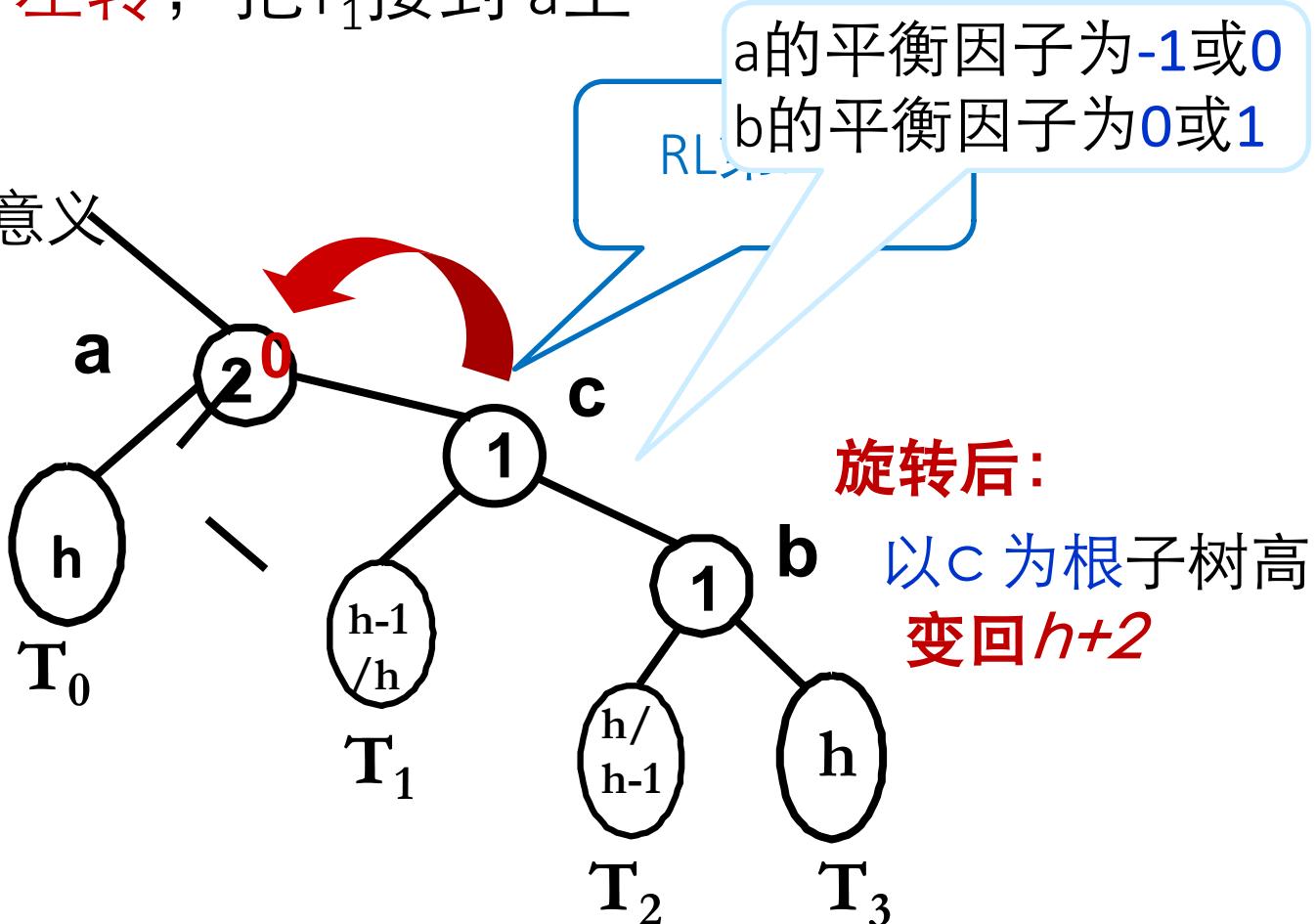


「RL型双旋转：2nd Step」

2. 再作(a, c) 左转，把 T_1 接到 a 上

中间状态

平衡因子无意义

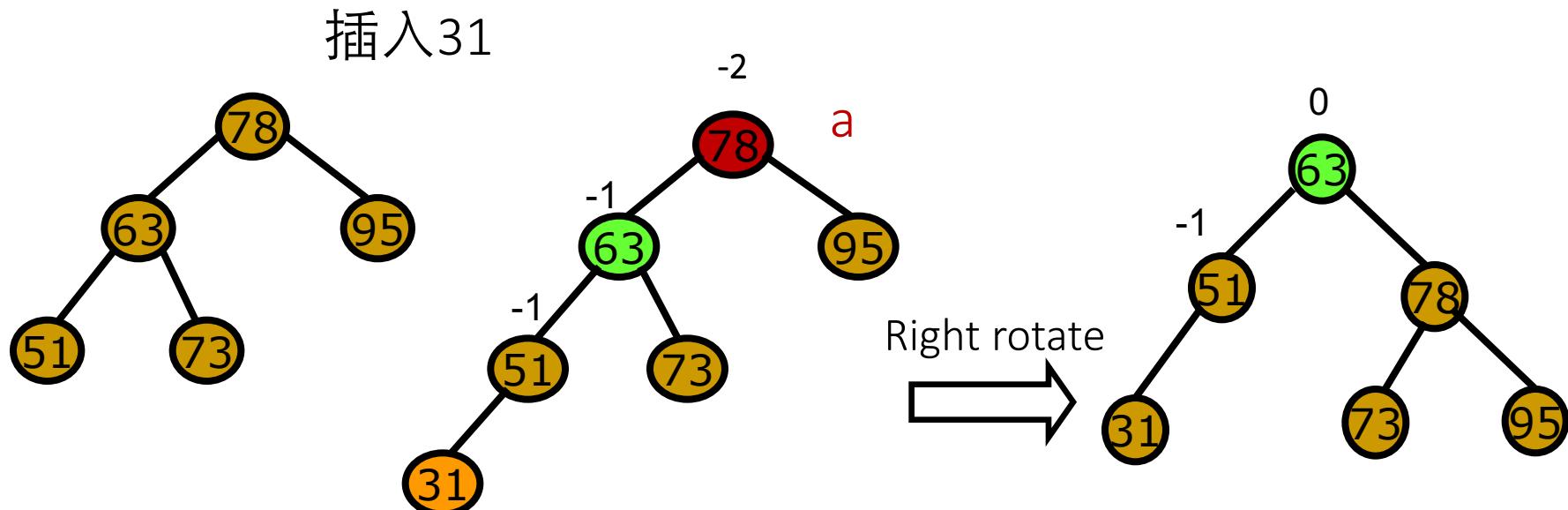


「AVL插入引发的变化」

- 插入同与BST
 - 新结点以**叶结点**身份插入
- 相应子树根结点的变化分**三类**：
 - 结点原本**平衡**，插入后成为**左重** (left-higher) 或**右重** (right-higher)，此时该结点的前驱结点的状态需改变
 - 结点原本**某一边重**，而插入后成为**平衡**的，此时前驱结点不改变状态，因为子树的高度未变
 - 结点原本**左重或右重**的，而新结点又**插入到重的一边**，此时该结点就不再满足AVL条件（称为“**危机结点**”）

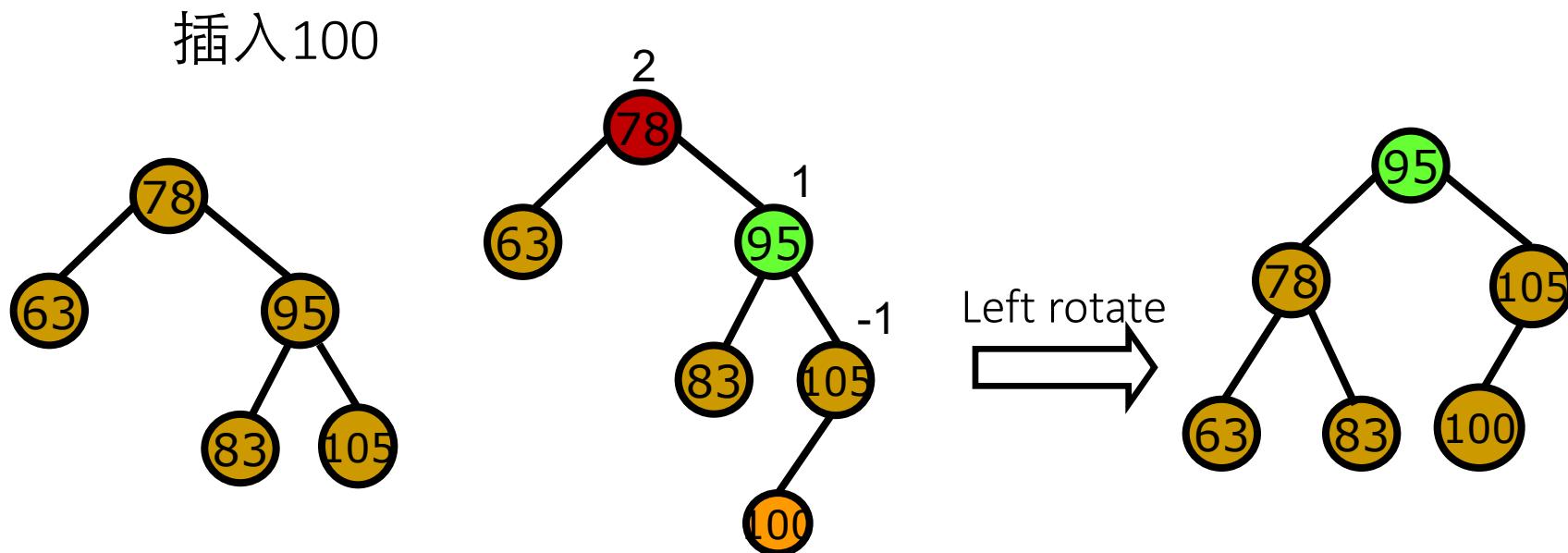
LL型示例

- 导致不平衡的结点为 a 的左子树的左子树，这时 a 的平衡因子变为-2



RR型示例

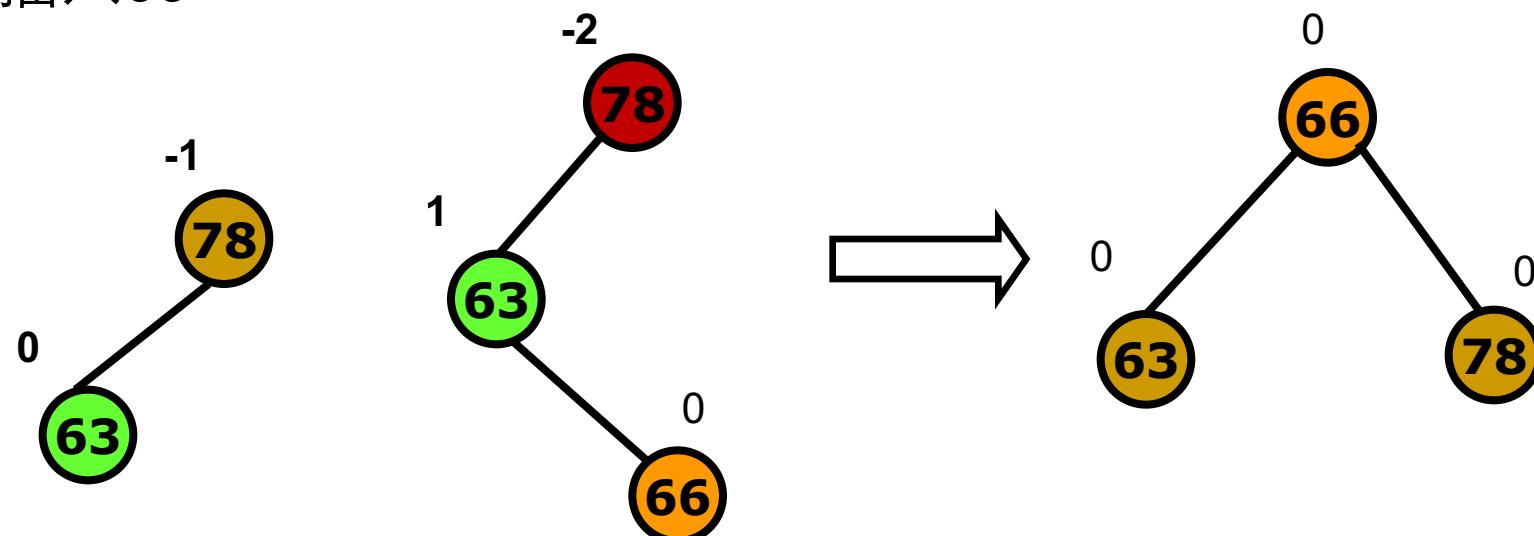
- 导致不平衡的结点为 a 的右子树的右子树，这时 a 的平衡因子变为 2



「LR型示例

- 导致不平衡的结点为a 的左子树的右子树，此时a 的平衡因子为-2

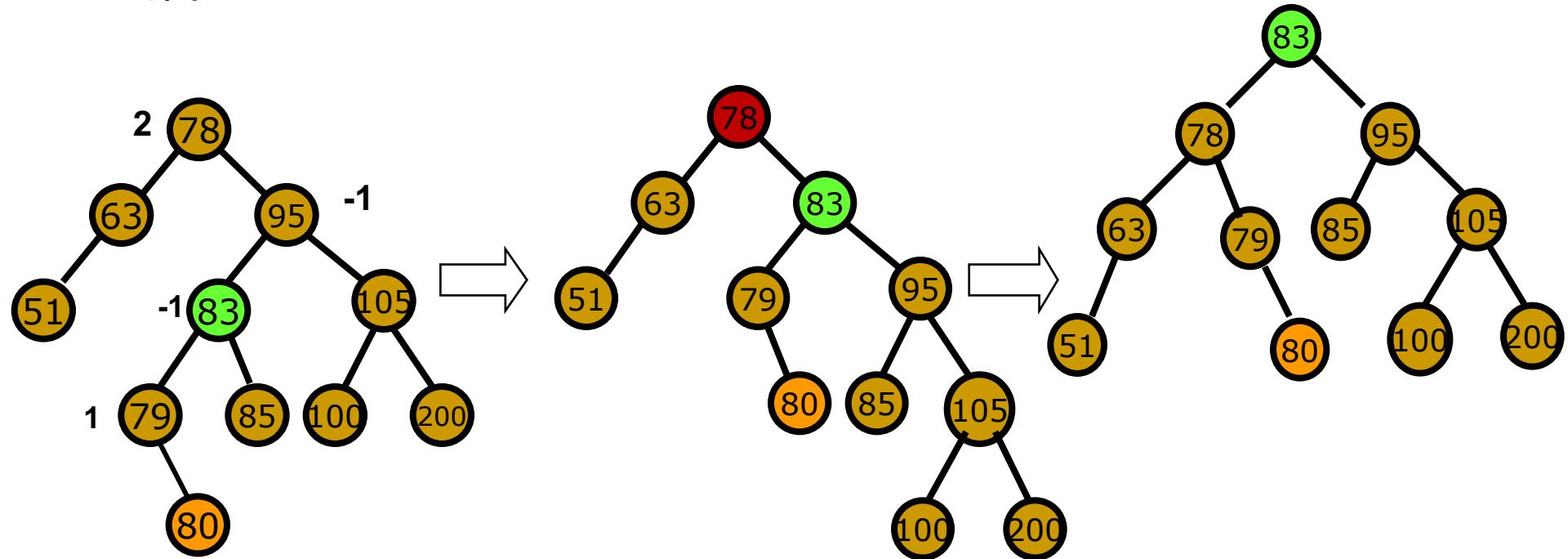
插入66



「RL型示例」

- 导致不平衡的结点为 a 的右子树的左子树，这时 a 的平衡因子为 2

插入 80



AVL插入的调整

- 局部调整
- 原来二叉树在 a 结点上面的其余部分（若还有的话）总是保持平衡的
 - 通过旋转得到的 新树 保持了原来的中序周游顺序： $T_0 \ a \ T_1 \ c \ T_2 \ b \ T_3$
 - 新的子树高度为 $h+2$ ，保持插入前子树的高度不变
 - 旋转处理中仅需改变若干指针

「AVL的删除

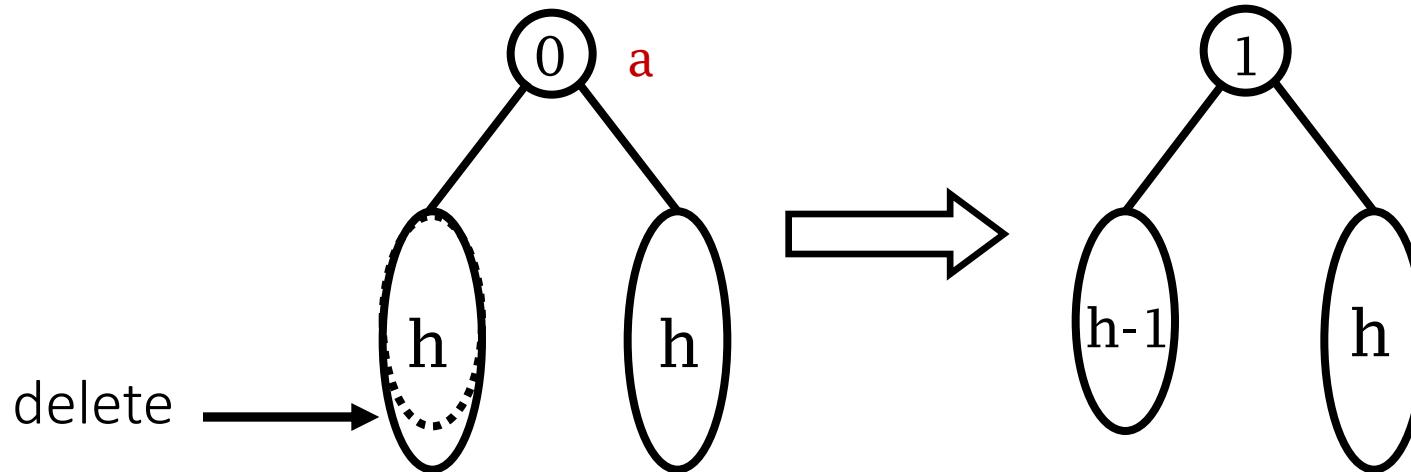
- 删除结点的操作同BST的删除
- 调整平衡因子则是插入的逆操作，但情况比插入更为复杂
- 删除会导致子树的高度及平衡因子变化，需沿着被删除结点到根结点的路径来调整这种变化
- 删除后的再平衡可能需在相应的路径上不止一处实施单旋或双旋

「AVL删除后的调整过程

- 删 除 结 点 后， 查 看 平 衡 因 子 是否 需 要 改 动
 - 若 需 要，则 改 动 之；
 - 若 不 需 改 动 则 不 必 继 续 向 上 回 溯，用 一 个 布 尔 变 量 *modified* 来 记 录 之，其 初 值 为 TRUE，当 *modified* = FALSE 时，回 溯 停 止
- 分 三 种 情 况 处 理，假 定
 - 最 下 层 的 不 平 衡 发 生 在 结 点 *a* 开 始 的 子 树 中
 - 删 除 发 生 在 *a* 的 左 子 树 中

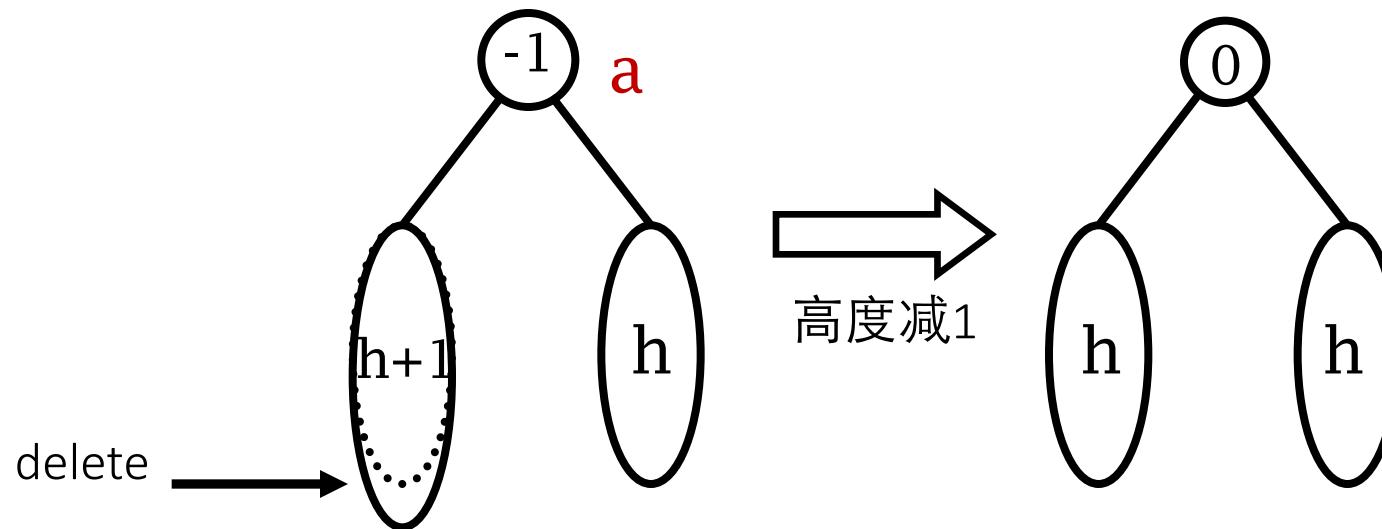
AVL的删除： case #1

- 当前结点 a 的 $bf = 0$
 - 若其左或右子树被缩短，则将其 bf 值置为 1 或 -1；
 - 置 $modified = \text{FALSE}$ ；
 - 由于以 a 为根的子树高度未变，变化不会波及上面的结点，调整结束



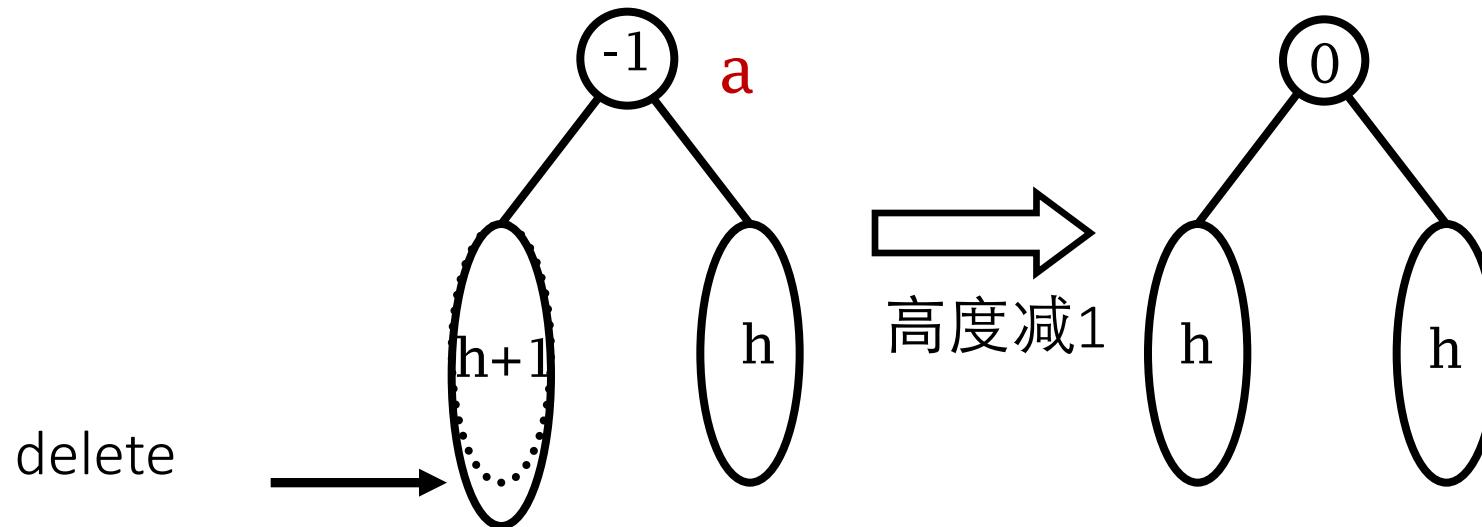
AVL 的删除： case #2

- 当前结点 a 的 $bf \neq 0$ ， 较高的子树被缩短
 - 则其平衡因子修改为 0；
 - 置 $modified = \text{TRUE}$
 - 需继续向上修改，由于以 a 为根的子树高度发生变化，可能会影响父结点的 bf 值



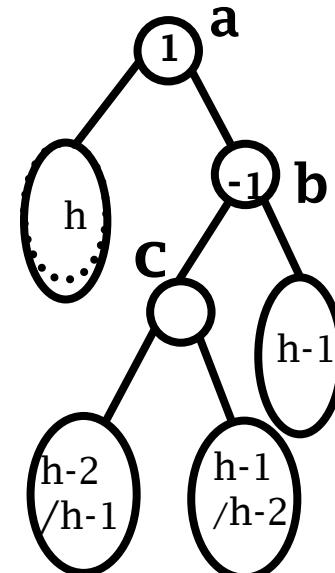
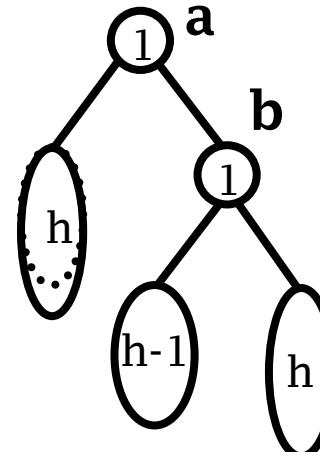
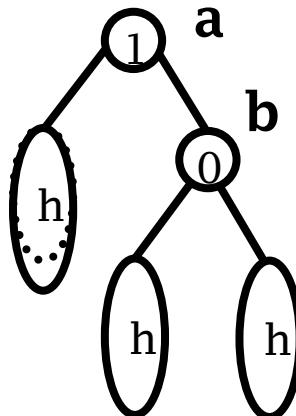
AVL 的删除： case #2

- 当前结点 a 的 $bf \neq 0$ ， 较高的子树被缩短
 - 则其平衡因子修改为 0；
 - 置 $modified = TRUE$
 - 需继续 **向上修改**，由于以 a 为根的子树高度发生变化，可能会影响父结点的 bf 值



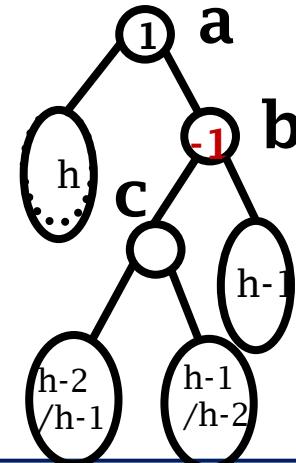
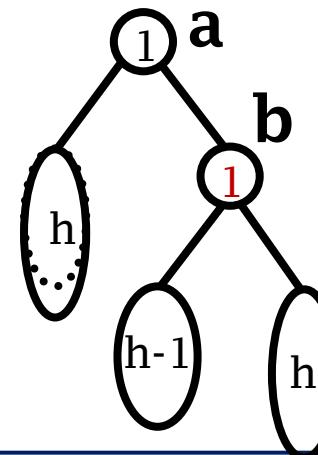
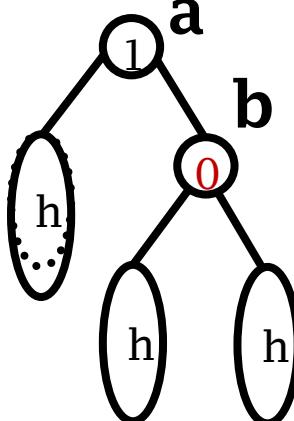
AVL 的删除： case #3

- 当前结点 **a** 的 $bf \neq 0$ ，且其较矮的子树被缩短，则结点 **a** 必然不再平衡，令其较高子树的根结点为 **b**
 - 情况 3.1: **b** 的平衡因子为 0
 - 情况 3.2: **b** 的平衡因子与 **a** 的平衡因子相同
 - 情况 3.3: **b** 和 **a** 的平衡因子相反



AVL 的删除： case #3

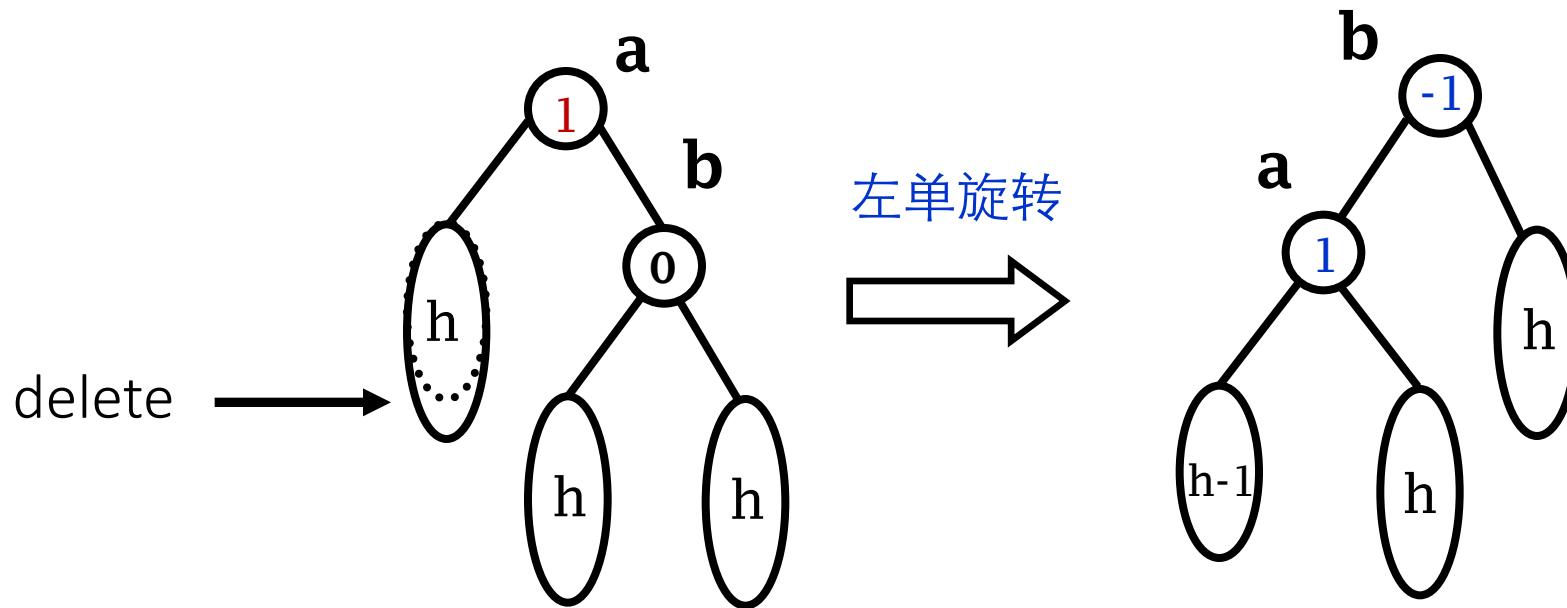
- 情况 3.1： b 的平衡因子为 0
 - 执行一个单旋来恢复结点 a 的平衡，并置 *modified = FALSE*
- 情况 3.2： b 的平衡因子与 a 的平衡因子相同
 - 执行一个单旋来恢复平衡， a 与 b 的bf 值均置为0，并置 *modified = TRUE*
- 情况 3.3： b 和 a 的平衡因子相反
 - 执行一个双旋来恢复平衡，先围绕 b 旋转，再围绕a 旋转，新的根结点的bf值变为0，其他结点做相应处理，并置 *modified = TRUE*



AVL 删除: case #3.1

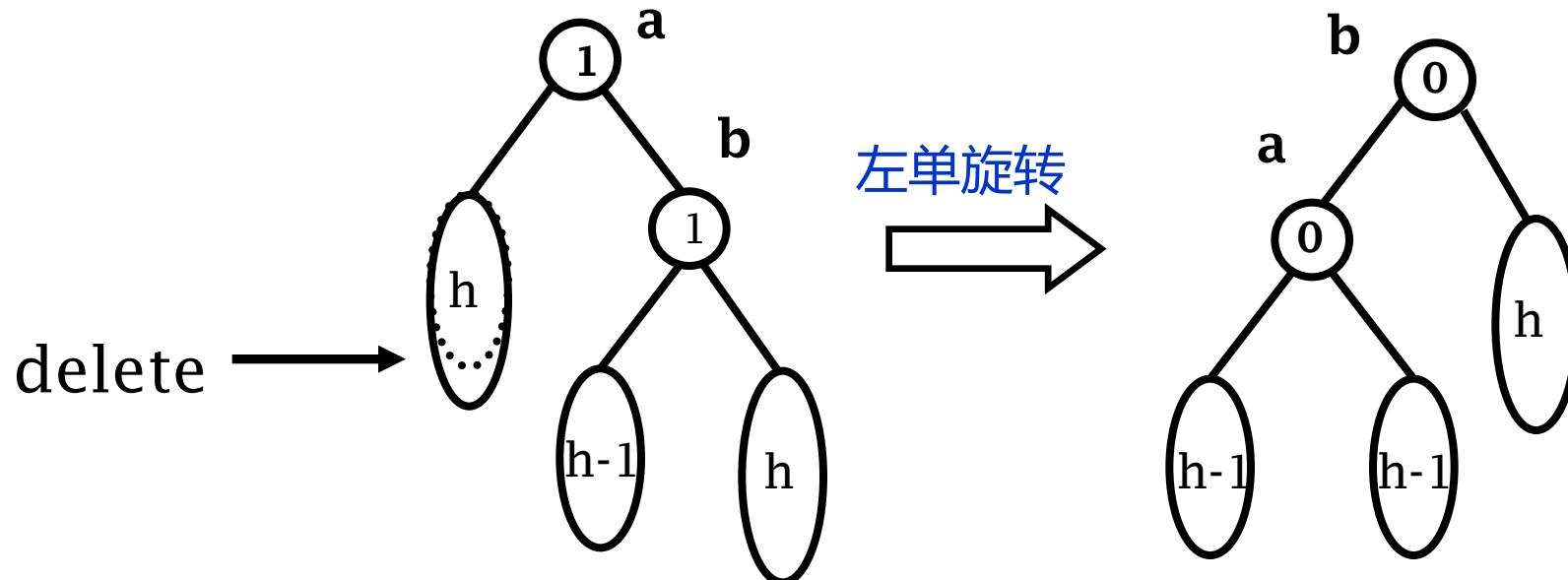
■ 情况 3.1: b 的平衡因子为 0

- 单旋转
- modified = FALSE



AVL删除: case #3.2

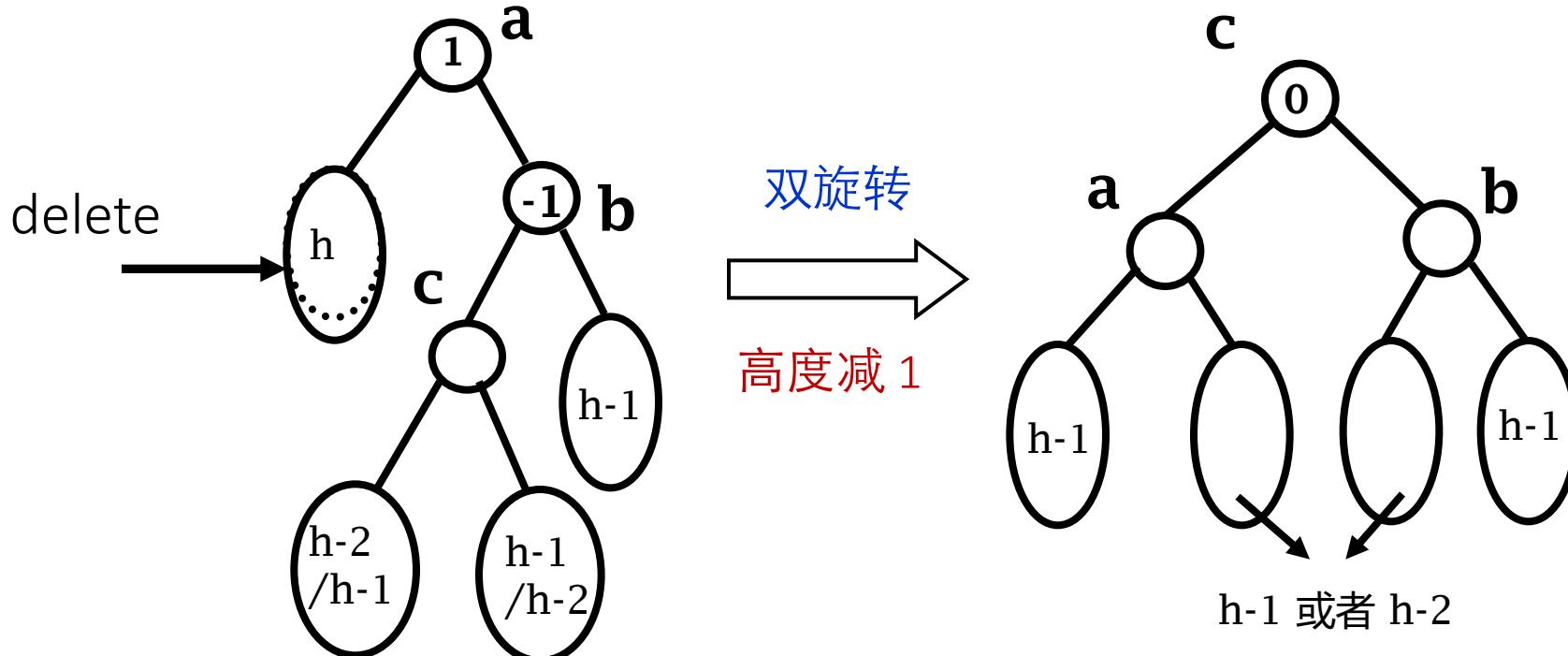
- 情况 3.2: b 的平衡因子与 a 的平衡因子相同
 - 单旋转
 - 结点 a、b 平衡因子都变为 0
 - modified =TRUE



AVL 删除: case #3.3

■ 情况 3.3: b 和 a 的平衡因子相反

- 双旋转, 先围绕 b 旋转, 再围绕 a 旋转
- 新的根结点平衡因子为 0; 其他结点应做相应的处理
- modified = TRUE



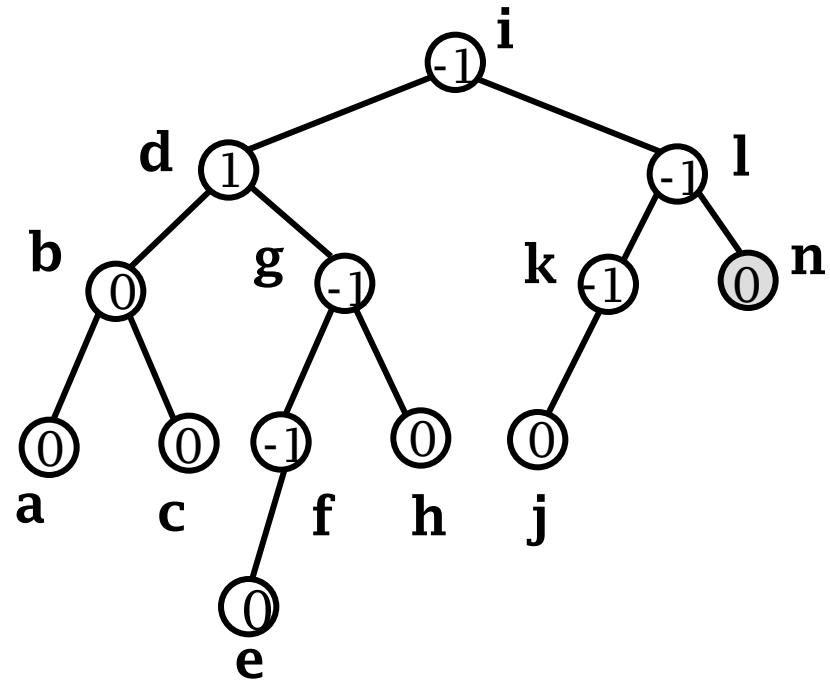
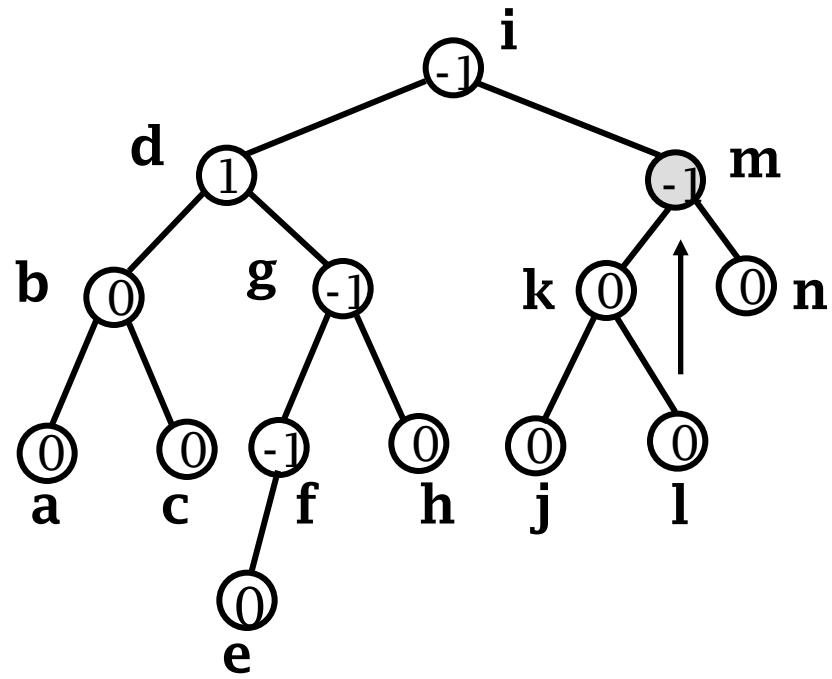
「删除后的连续调整

- 连续调整
 - 调整可能导致祖先结点发生新的不平衡
 - 这样的调整操作要持续下去，可能传递到根结点为止
- 从被删除的结点向上查找到其祖父结点
 - 然后开始单旋转或者双旋转操作
 - 旋转次数为 $O(\log n)$

AVL Deletion Case: 小结

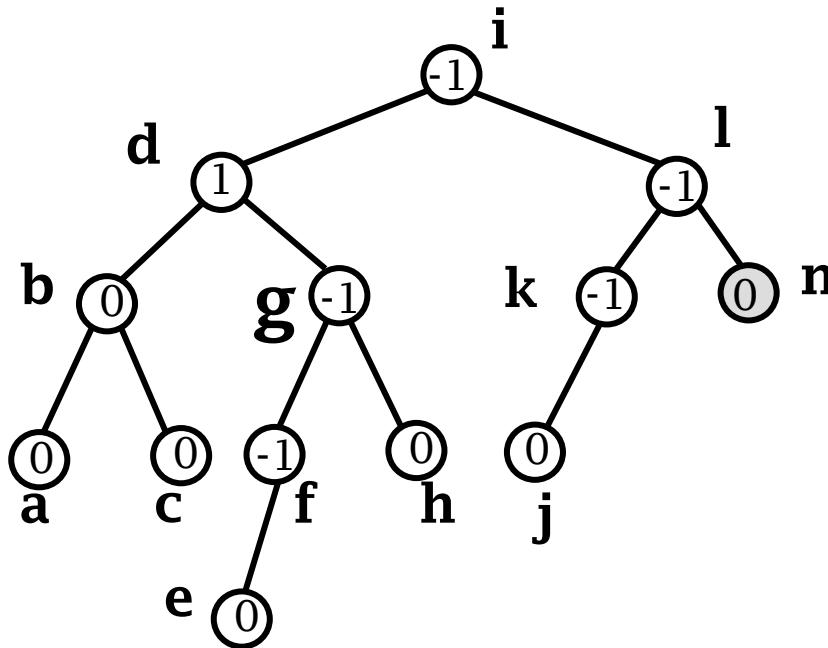
- 前述讨论均假设删除操作发生在左子树上
 - 因对称性，发生在右子树上时有类似的情况要考虑
- 另外，实现时考虑问题：
 - 实施一个删除操作时最上层树根会不会受影响，也许树根本身需要旋转
- 删除树根本身，类似二叉检索树的删除，考虑平衡问题
 -

AVL 树删除的示例

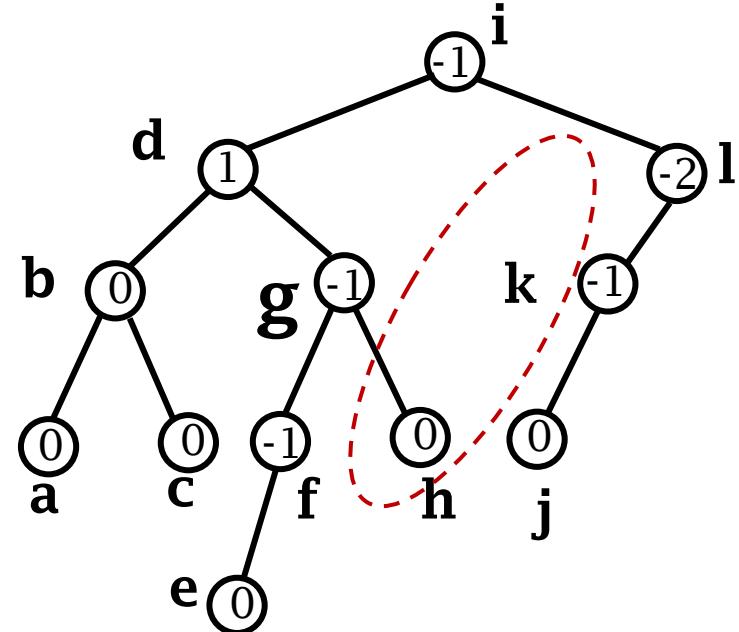


(a) 删除结点 m，则需要使用其中序前驱 | 代替（情况1）

AVL 树删除的示例

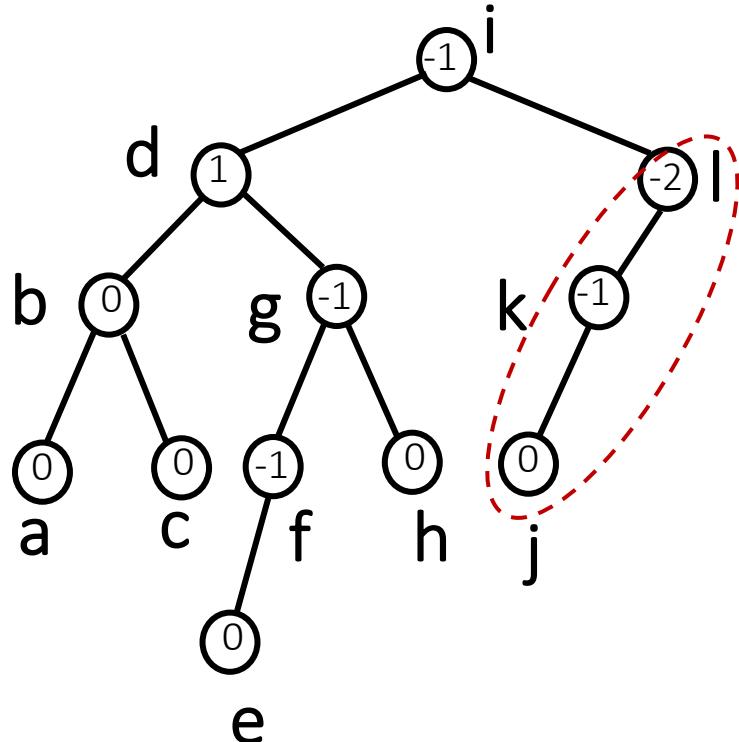


(b) 删除结点 **n** (情况 3.2)

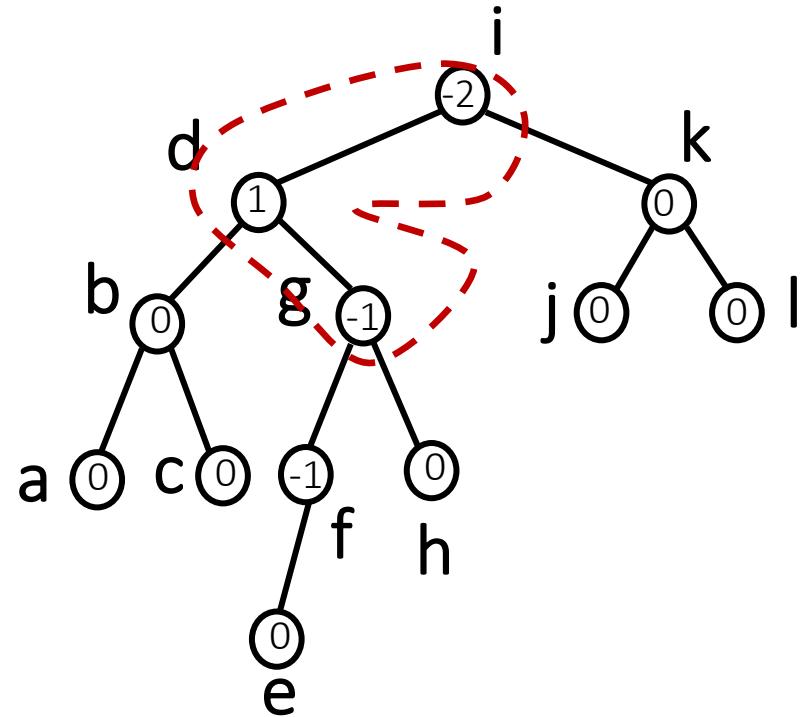


(c) 需要以 **l** 为根进行 LL 单旋转 (情况 3.2)

AVL 树删除的示例

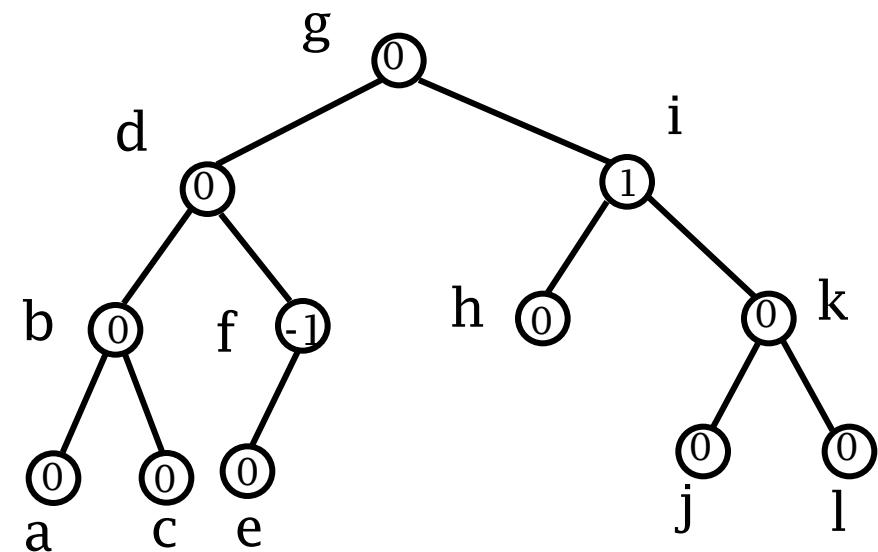
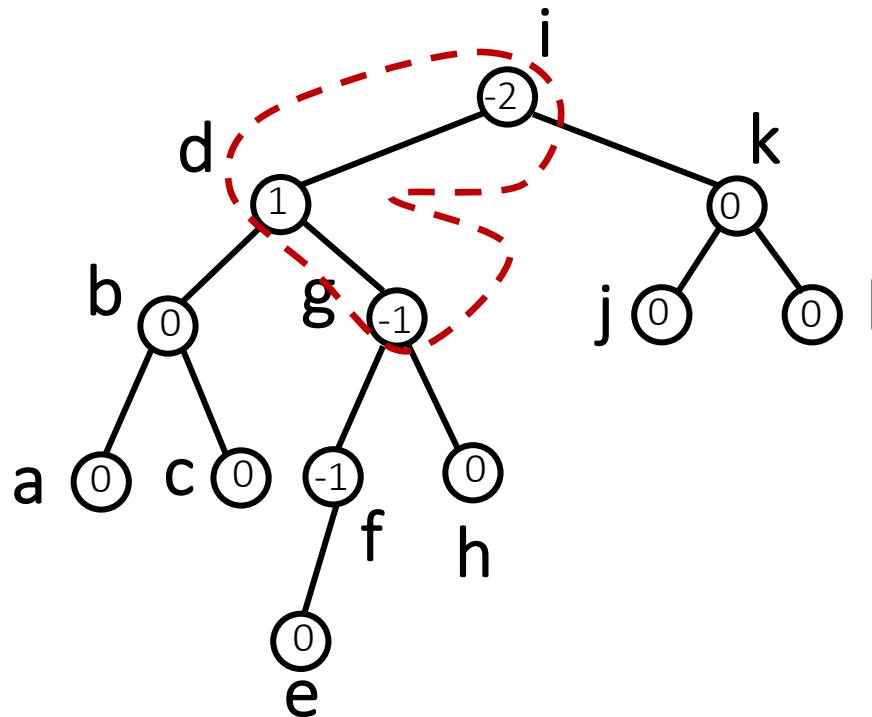


(c) 需要以 i 为根
进行 LL 单旋转 (情况 3.2)



(d) LL 单旋转完毕，上溯调整父节点 i，需要以 i 为根的 LR 双旋转 (情况3.3)

AVL 树删除的示例



(d) LL 单旋转完毕，上溯调整父节点 i，
需要以 i 为根的 LR 双旋转（情况3.3）

(e) 调整完毕，AVL 树重新平衡

AVL Tree效率分析

- 插入
 - 至多需要 1 次单旋 或 双旋
- 删除
 - 最坏需要 $1.44\log(n+2)$ 次 旋转
- 检索
 - 平均需要 $\log(n) + 0.25$
- 实验 (karlton et al., 1976)
 - 78%的删除并不破坏平衡
 - 只有53%的插入不影响树的平衡性

AVL 树

- AVL 树有效吗?
 - Search, insertion, and deletion are $O(\log n)$, where n is the number of *nodes* in the tree
- 具有 n 个结点的AVL树的高度不会超过 $1.44\log n$
 - 根据AVL的定义，最少的结点数目可由下述迭代式表示：
 - ◆ $\text{AVL}_h = \text{AVL}_{h-1} + \text{AVL}_{h-2} + 1$
 - ◆ 其中， $\text{AVL}_0 = 0$, $\text{AVL}_1 = 1$
 - ◆ 由此可得， n 个结点的AVL树的高度

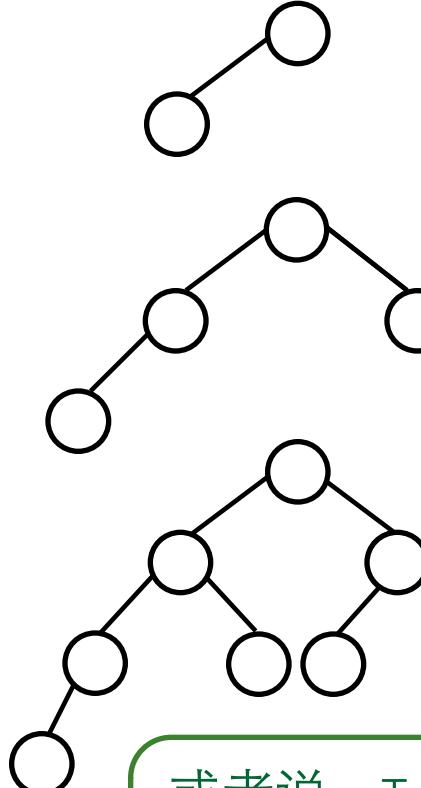
$$\log(n+1) \leq h \leq 1.44\log(n+2) - 0.328$$

- 实验表明 (Knuth 1998)，对于 n 很大时，平均的检索长度通常很接近 $\log(n)$ (约为 $\log(n) + 0.25$)

「AVL 树的深度」

- 具有 n 个结点的 AVL 树深度一定是 $O(\log n)$
- n 个结点的 AVL 树的最大深度不超过 $K \log_2 n$
 - 这里 K 是一个常数
- 最接近于不平衡的 AVL 树
 - 构造一系列 AVL 树 T_1, T_2, T_3, \dots

AVL 树的深度



T_1

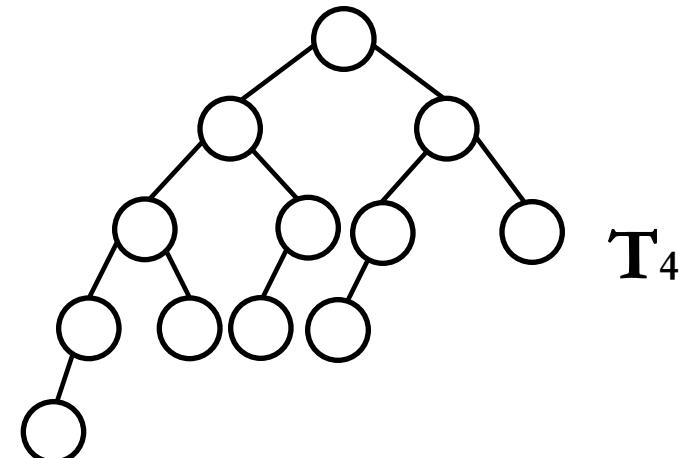
T_i 的深度是 i

T_2

每棵具有深度 i 的
其它 AVL 树都比 T_i
的结点个数多

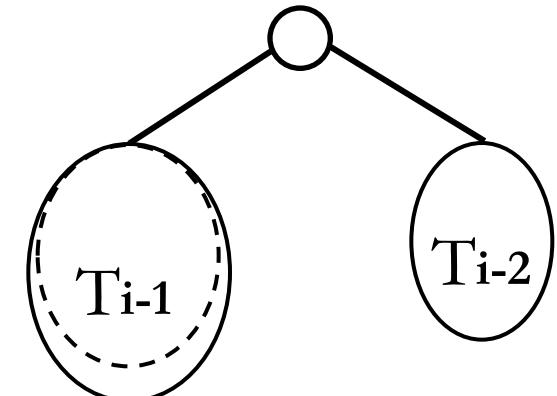
T_3

或者说， T_i 是具有同样的结点数目的所有 AVL 树中最接近不平衡状态的，删除一个结点都会引起不平衡



T_4

T_i



T_{i-1}

T_{i-2}

「深度的证明

- 可看出有下列关系成立：

$$t(1) = 2$$

$$t(2) = 4$$

$$t(i) = t(i-1) + t(i-2) + 1$$

- 对于 $i > 2$ 此关系很类似于定义 Fibonacci 数的关系：

$$F(0) = 0$$

$$F(1) = 1$$

$$F(i) = F(i-1) + F(i-2)$$

「深度的证明

- 对于 $i > l$ 仅检查序列的前几项就可有

$$t(i) = F(i+3) - 1$$

- Fibonacci 数满足渐近公式

$$F(i) = \frac{1}{\sqrt{5}} \phi^i, \text{ 这里 } \phi = \frac{1 + \sqrt{5}}{2}$$

- 由此可得近似公式

$$t(i) \approx \frac{1}{\sqrt{5}} \phi^{i+3} - 1$$

「深度的证明」

- 解出深度 i 与结点个数 $t(i)$ 的关系

$$\phi^{i+3} \approx \sqrt{5}(t(i) + 1)$$

$$i + 3 \approx \log_{\phi} \sqrt{5} + \log_{\phi} (t(i) + 1)$$

- 由换底公式 $\log_{\phi} X = \log_2 X / \log_2 \phi$ 和 $\log_2 \phi \approx 0.694$,
求出近似上限

- $t(i) = n$

$$i < \frac{3}{2} \log_2(n+1) - 1$$

- 故, n 个结点的 AVL 树的深度一定是 $O(\log n)$

「AVL 树

- AVL树适用于组织较小的、内存中的目录
- 存放在外存储器上的较大的文件
 - B树/B+树，尤其是B+树

「AVL Tree的扩展

- 允许树的高度差为 $\Delta > 1$ (Foster, 1973)，最差情况下，高度随 Δ 而增加

$$h = \begin{cases} 1.81\log(n) - 0.71, & \text{if } \Delta = 2 \\ 2.15\log(n) - 1.13, & \text{if } \Delta = 3 \end{cases}$$

- 实验表明，与单纯的AVL树($\Delta=1$)相比，平均访问结点数目增加了，但重组的数目降低了

思考

1. AVL tree: $|bf(t)| < 2$
2. Full binary tree: $bf = ?$
3. complete binary tree: $bf = ?$

思考

- 将关键码 $1, 2, 3, \dots, 2^{k-1}$ 依次插入到一棵初始为空的 AVL 树中，试证明结果是一棵高度为 k 的完全满二叉树