数据结构与算法 (十二)

张铭 主讲

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

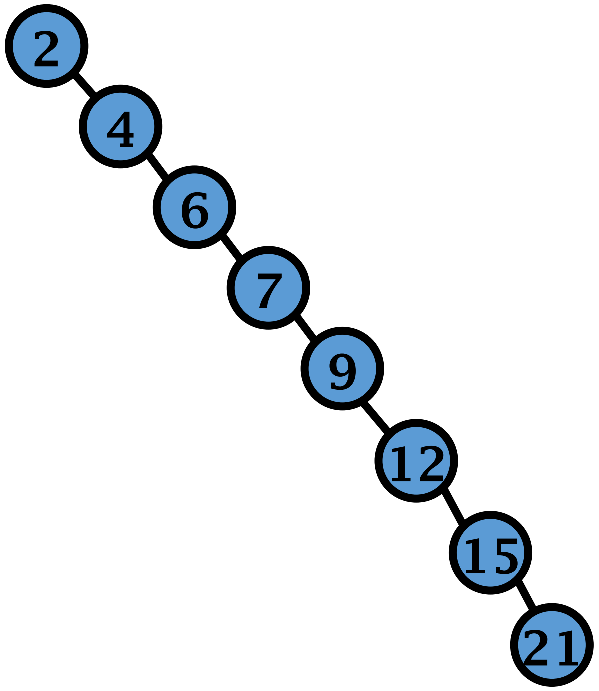
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg

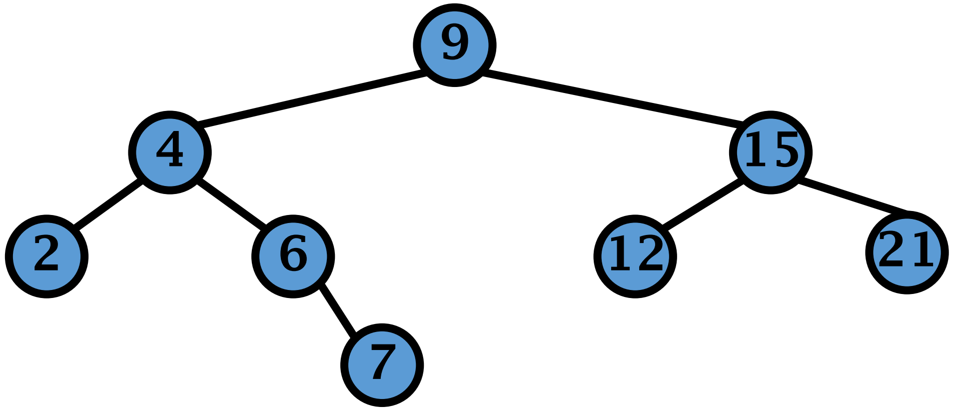
第十二章 高级数据结构

- 12.1 多维数组
- 12.2 广义表
- 12.3 存储管理
- 12.4 Trie 树
- 12.5 改进的二叉搜索树
  - 12.5.1 平衡的二叉搜索树
  - 12.5.2 伸展树

12.5.1 平衡的二叉搜索树 (AVL)

- BST受输入顺序影响
  - 最好O (log n)
  - 最坏O (n)
- Adelson-Velskii 和 Landis
  - AVL 树，平衡的二叉搜索树
  - 始终保持O (log n) 量级

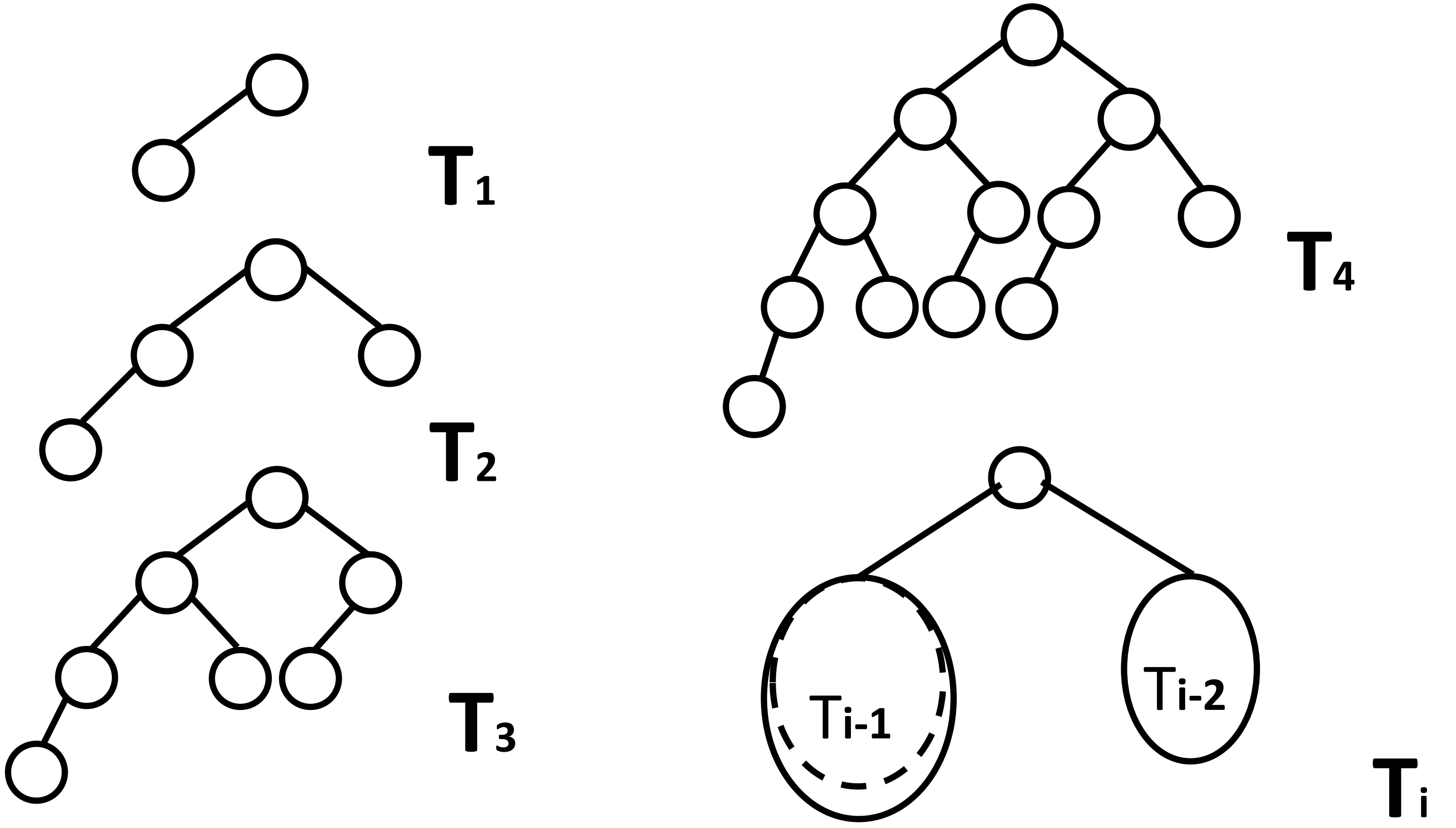




AVL 树的性质

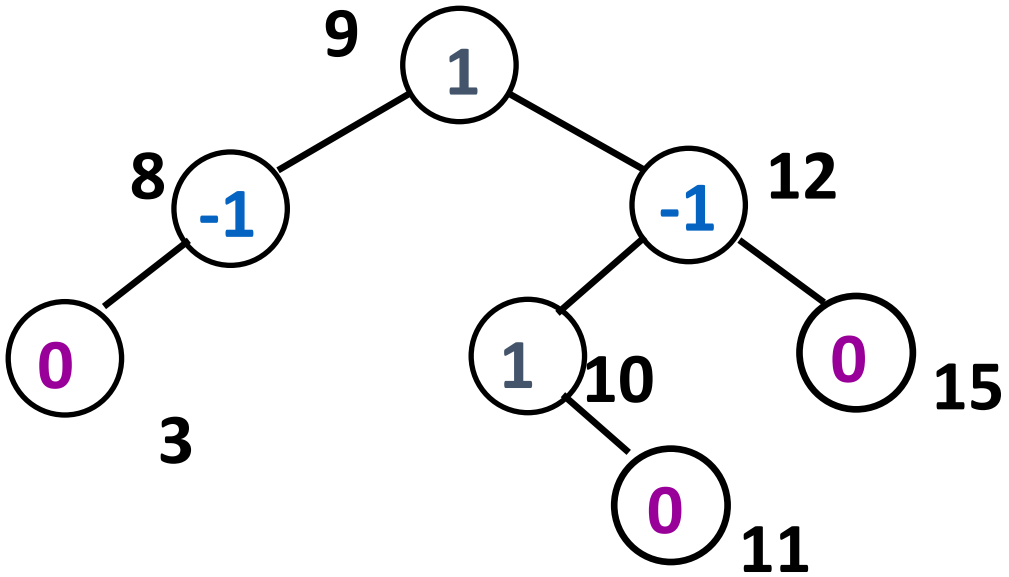
- 可以为空
- 具有 n 个结点的 AVL 树，高度为 O (log n)
- 如果 T 是一棵 AVL 树
  - 那么它的左右子树 T_L、T_R 也是 AVL 树
  - 并且 $|h_L - h_R| \le 1$
    - h_L、h_R 是它的左右子树的高度

AVL 树举例



平衡因子

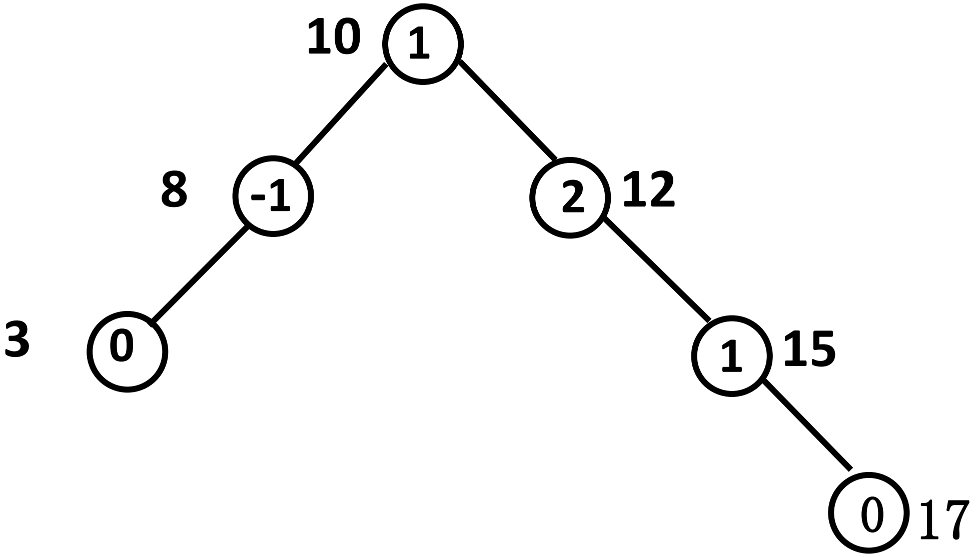
- 平衡因子，*bf* （x）：
  - *Bf* （x）= $height(x_{lchild}) - height(x_{rchild})$
- 结点平衡因子可能取值为 0，1 和 -1



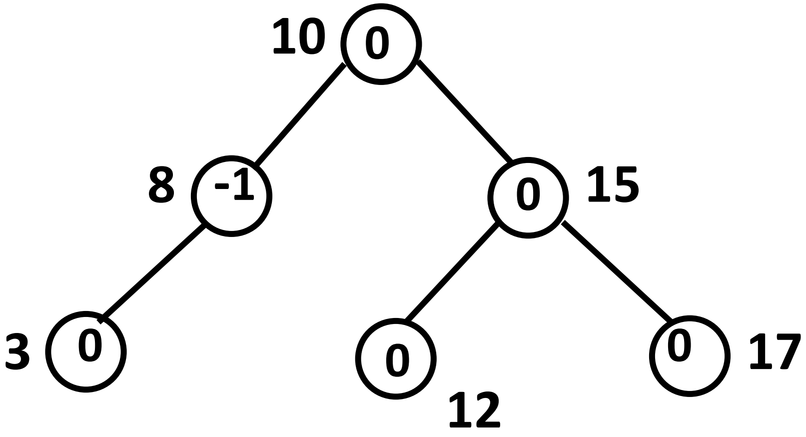
AVL 树结点的插入

- 插入与 BST 一样：新结点作叶结点
- 调整后的状态
  - 结点原来是平衡的，现在成为左重或右重的
    - 修改相应前驱结点的平衡因子
  - 结点原来是某一边重的，而现在成为平衡的了
    - 树的高度未变，不修改
  - 结点原来就是左重或右重的，又加到重的一边
    - 不平衡
    - “危急结点”

恢复平衡



插入17后导致不平衡



重新调整为平衡结构

12.5 改进的二叉搜索树

- 不平衡情况发生在插入新结点后
- BST 把新结点插入到叶结点
- 假设 a 是离插入结点最近，且平衡因子绝对值不等于0的结点
  - 新插入的关键码为 key 的结点 s 要么在它的左子树中，要么在其右子树中
  - 假设插入在右边，原平衡因子
    - (1) a->bf = -1
    - (2) a->bf = 0
    - (3) a->bf = +1

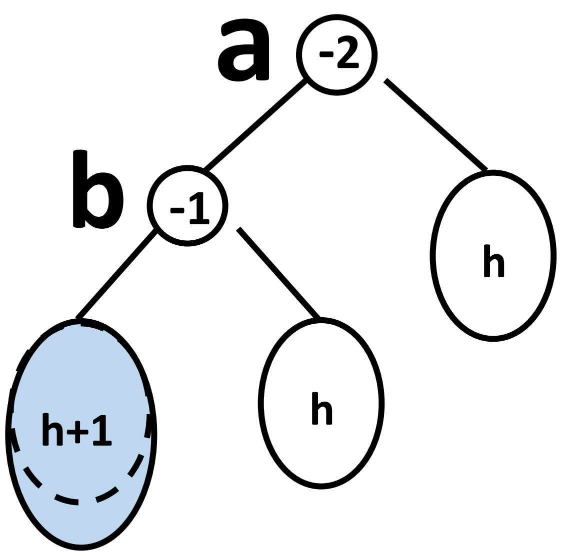
12.5 改进的二叉搜索树

- 假设 a 离新结点 s 最近，且平衡因子绝对值不等于0
  - s (关键码为key) 要么在 a 的左子树，要么在其右子树中
- 假设在右边，因为从 s 到 a 的路径上（除 s 和 a 以外）结点都要从原 bf=0 变为 |bf|=+1，对于结点 a
  - 1. a->bf = -1，则 a->bf = 0， a 子树高度不变
  - 2. a->bf = 0， 则 a->bf = +1，a 子树树高改变
    - 由a的定义 (a->bf ≠ 0) ，可知 a 是根
  - 3. a->bf = +1，则 a->bf = +2，需要调整

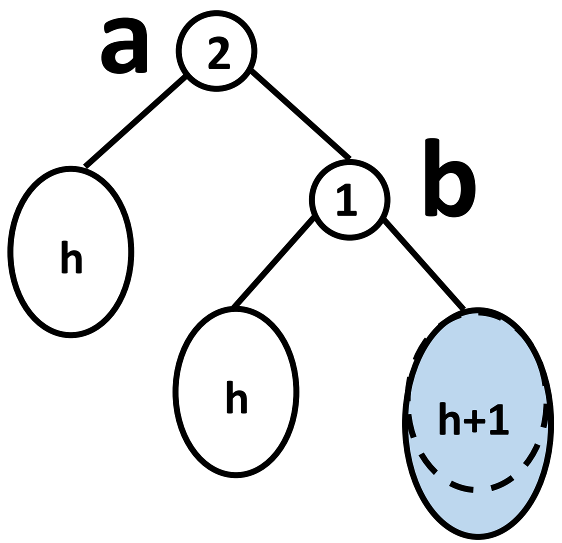
不平衡的情况

- AVL 树任意结点 a 的平衡因子只能是0，1，-1
- a 本来左重，a.bf= = -1，插入一个结点导致 a.bf 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
- 类似地， a.bf= = 1，插入新结点使得 a.bf 变为2
  - RR 型：导致不平衡的结点为 a 的右子树的右结点
  - RL 型：导致不平衡的结点为 a 的右子树的左结点

不平衡的图示



LL型

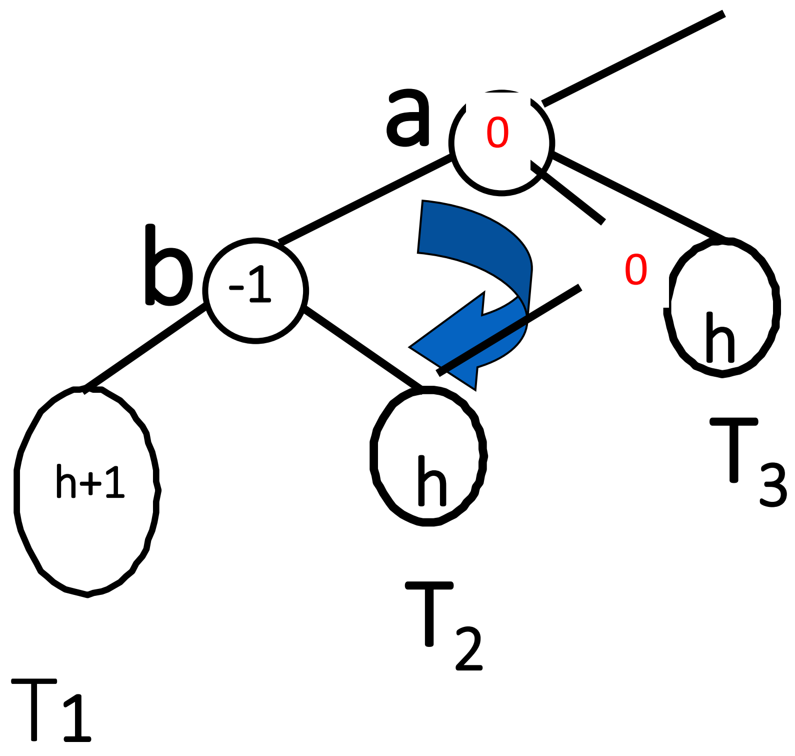


RR型

不平衡情况总结

- LL 型和 RR 型是对称的，
  LR 型和 RL 型是对称的
- 不平衡的结点一定在根结点与新加入结点之间的路径上
- 它的平衡因子只能是 2 或者 -2
  - 如果是 2， 它在插入前的平衡因子是1
  - 如果是 -2，它在插入前的平衡因子是 -1

LL单旋转



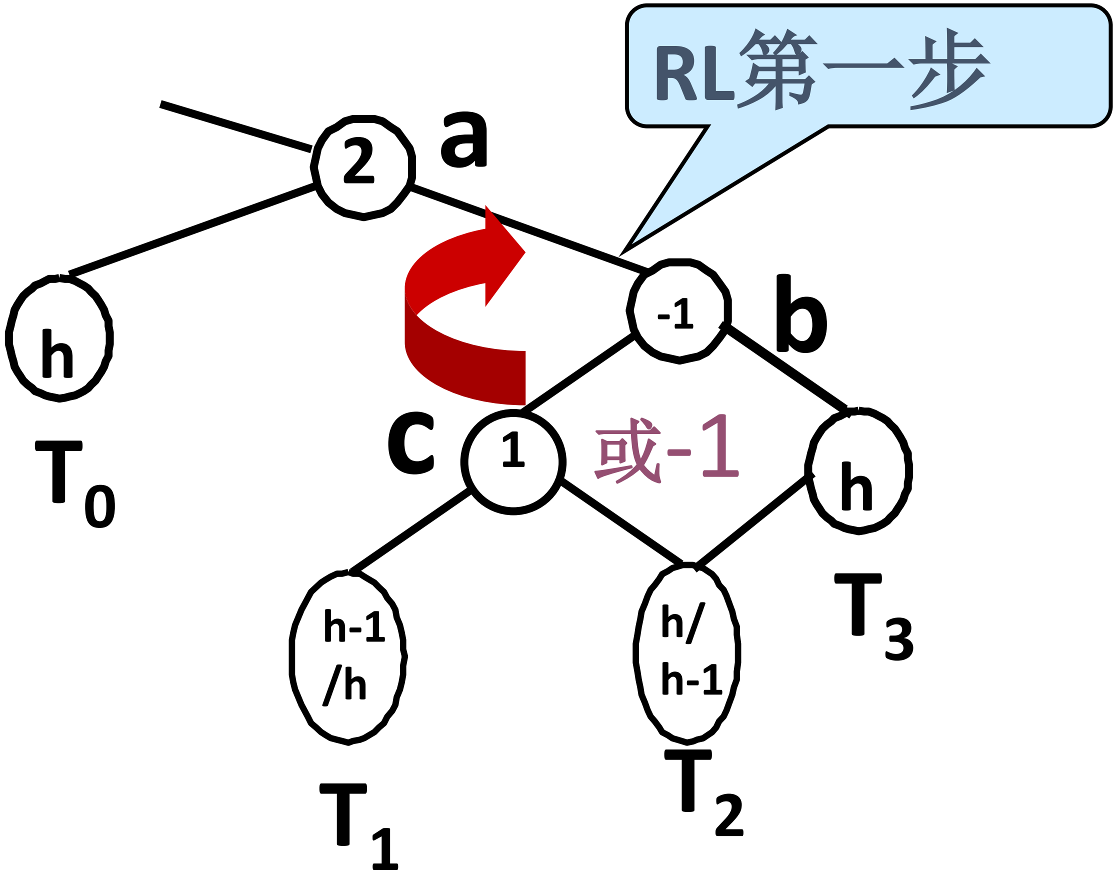
旋转运算的实质

- 以RR型图示为例，总共有7个部分
  - 三个结点：a、b、c
  - 四棵子树 T_0、T_1、T_2、T_3
    - 加重 c 为根的子树，但是其结构其实没有变化
    - T_2、c、T_3 可以整体地看作 b 的右子树
- 目的：重新组成一个新的 AVL 结构
  - 平衡
  - 保留了中序周游的性质
    - T_0 a T_1 b T_2 c T_3

双旋转

- RL 或者 LR 需要进行双旋转
  - 这两种情况是对称的
- 我们只讨论 RL 的情况
  - LR 是一样的

RL型双旋转第一步



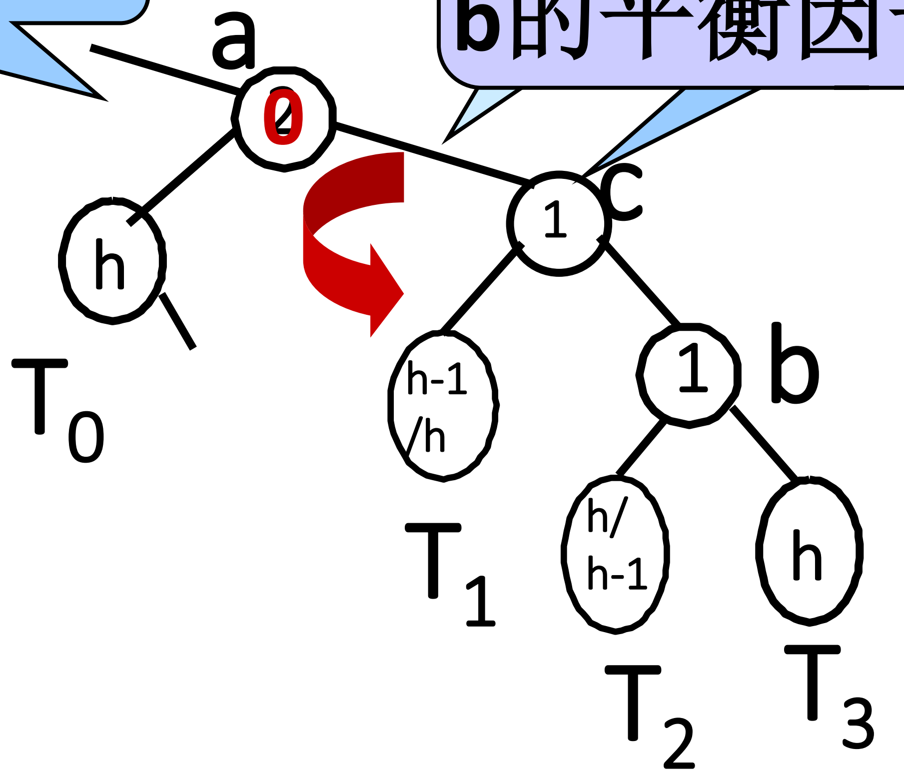
插入前
a子树高h+2

插入后
a子树高h+3

RL型双旋转第二步

中间状态
平衡因子无意义

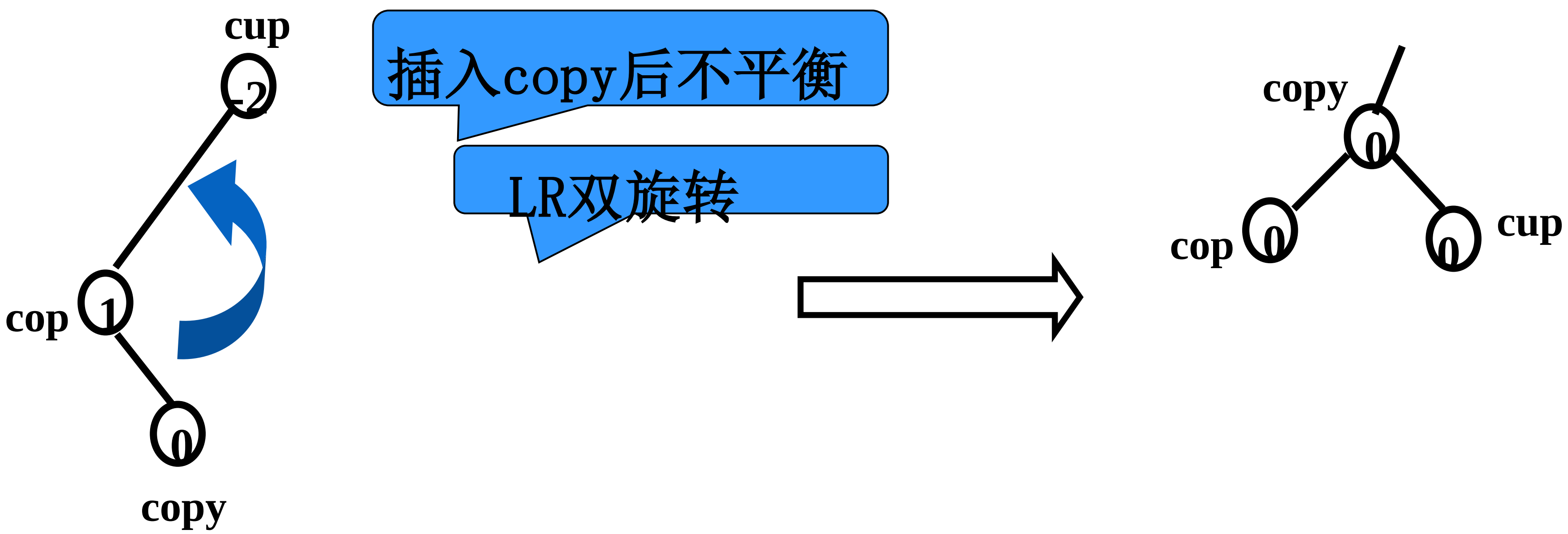
a的平衡因子为-1或0
b的平衡因子为0或1



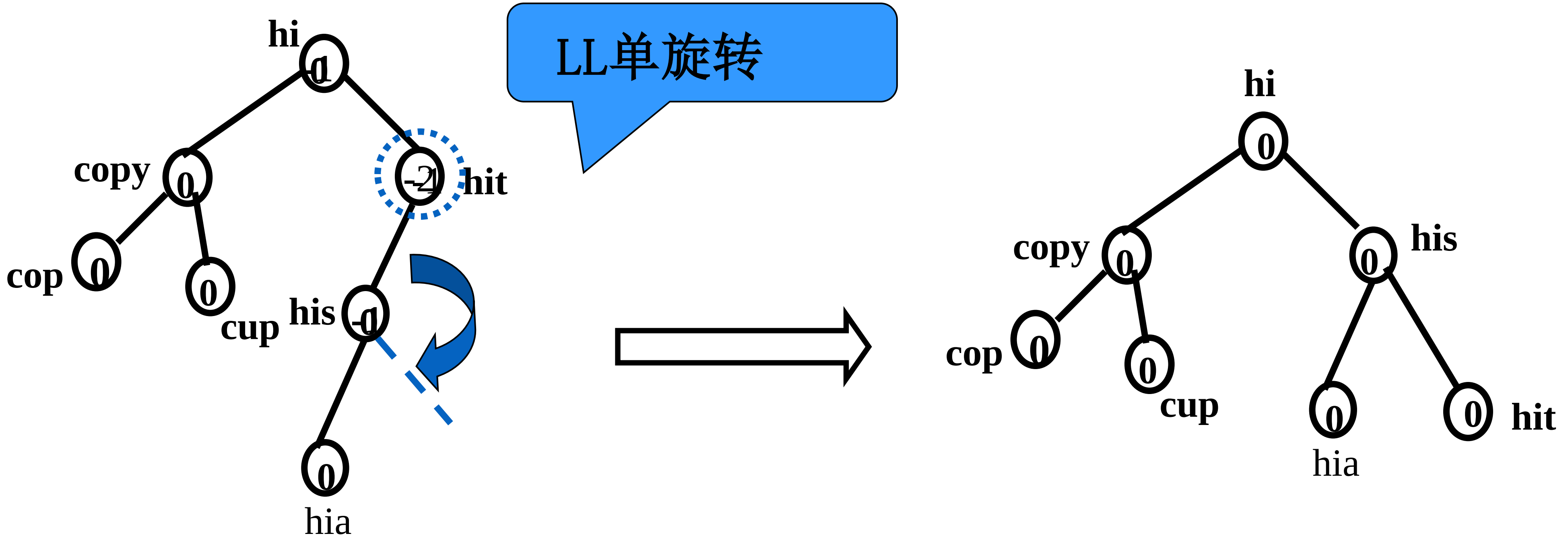
旋转运算的实质（续）

- 把树做任何一种旋转（RR、RL、LL、LR）
- 新树保持了原来的中序周游顺序
- 旋转处理中仅需改变少数指针
- 而且新的子树高度为 h+2，保持插入前子树的高度不变
- 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

插入单词：cup ，cop，copy， hit， hi， his和 hia后得到的AVL树



插入单词：cup ，cop， copy， hit， hi， his和 hia后得到的AVL树



AVL 树结点的删除

- 删除是插入的逆操作。从删除结点的意义上来说，AVL 树的删除操作与 BST 一样
- AVL 树的删除是比较复杂过程，下面具体讨论一下删除的过程
- 由于情况较多，所以图示每种情况只列举了一种例子，其他都是类似的

AVL 树结点的删除

- 具体删除过程请参考 BST 结点的删除
- 如果被删除结点 a 没有子结点→直接删除 a
- 如果 a 有一个子结点
  - 用子结点的内容代替 a 的内容，然后删除子结点
- 如果 a 有两个子结点
  - 那么则要找到 a 在中序周游下的前驱结点 b (b 的右子树必定为空)
  - 用 b 的内容代替 a，并且删除结点 b (如果 b 的左子树不空，则该左子树代替代替原来 b 的位置)。

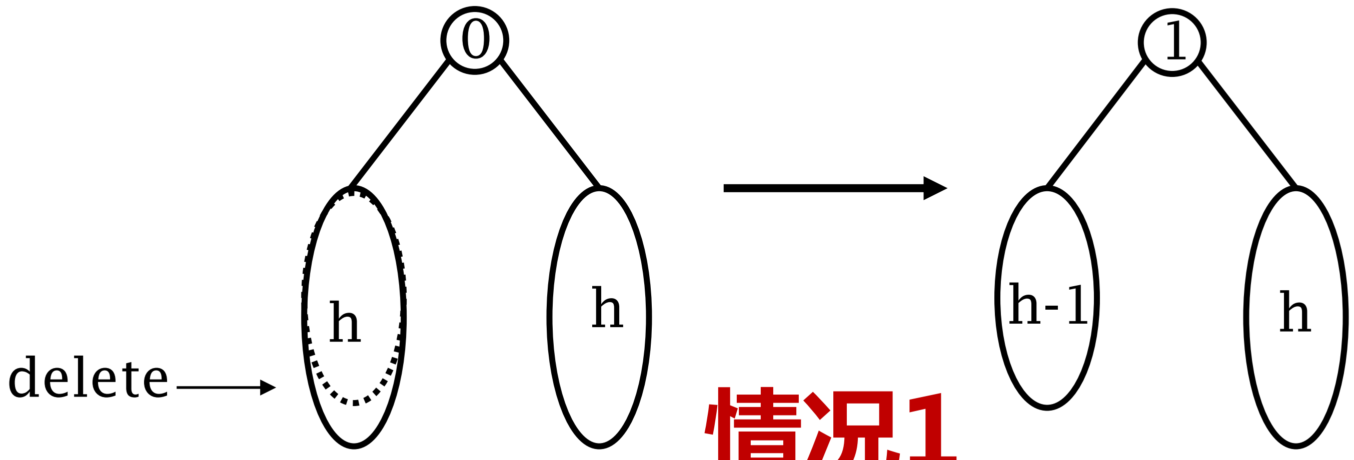
AVL结点删除后的调整

- AVL 树调整的需要
  - 删除结点后可能导致子树的高度以及平衡因子的变化
  - 沿着被删除结点到根结点的路径来调整这种变化
- 需要改动平衡因子
  - 则修改之
- 如果发现不需要修改则不必继续向上回溯
  - 布尔变量 modified 来标记，其初值为 TRUE
  - 当 modified=FALSE 时，回溯停止

有以下三种情况

AVL 树结点的删除过程（续）

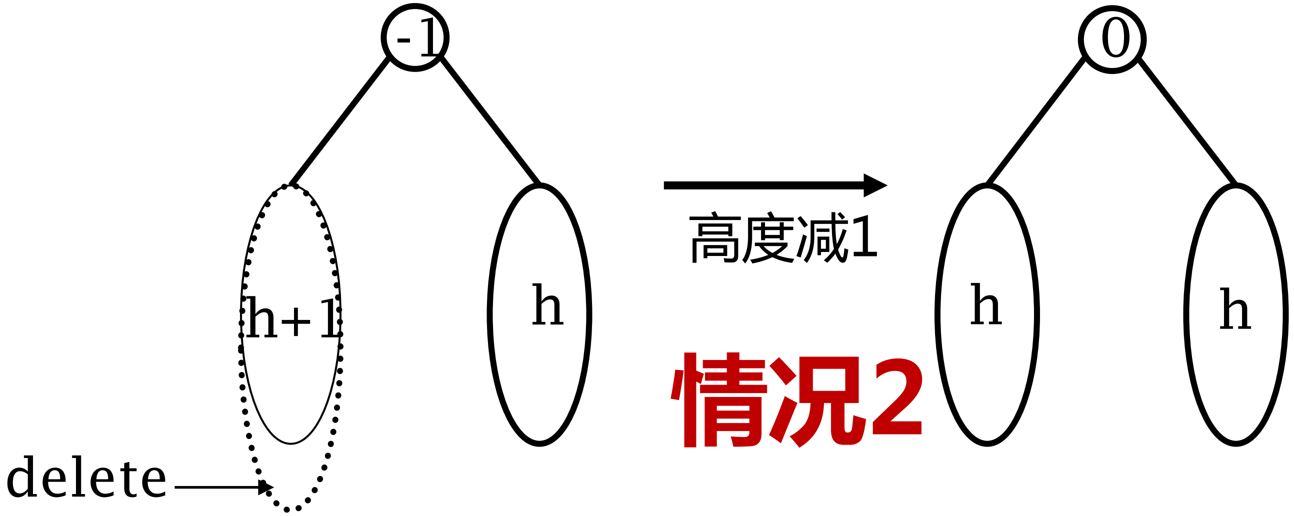
- 第一种情况当前结点a平衡因子为0
  - 其左或右子树被缩短，则平衡因子该为1或者-1
  - modified=FALSE
  - 变化不会影响到上面的结点，调整可以结束



情况1

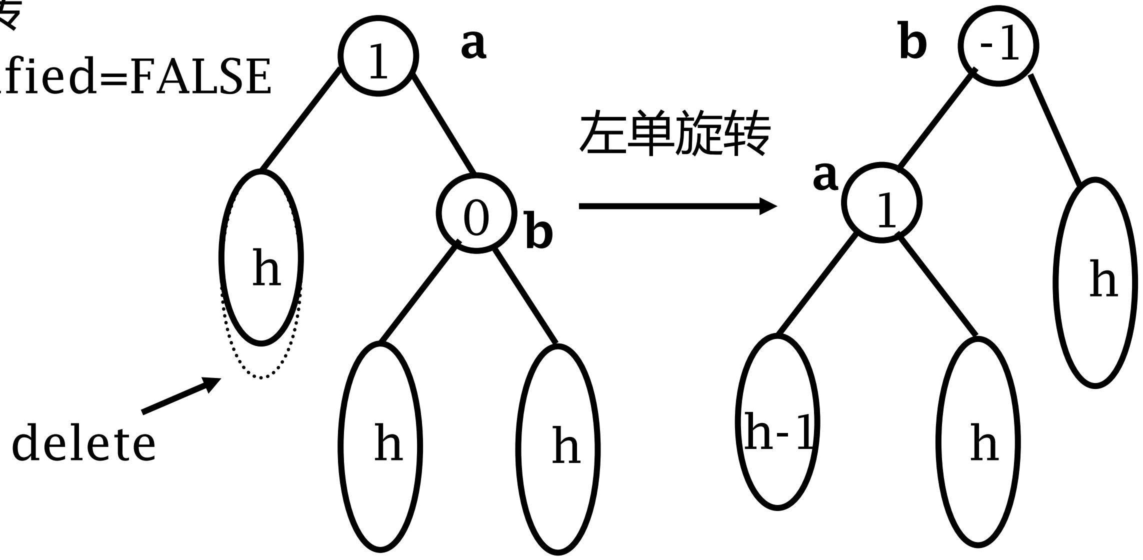
AVL 树结点的删除过程（续）

- 第二种情况是当前结点a平衡因子不为 0，但是其较高的子树被缩短
  - 则其平衡因子修改为 0
  - Modified = TRUE
  - 需要继续向上修改



AVL 树结点的删除过程（续）

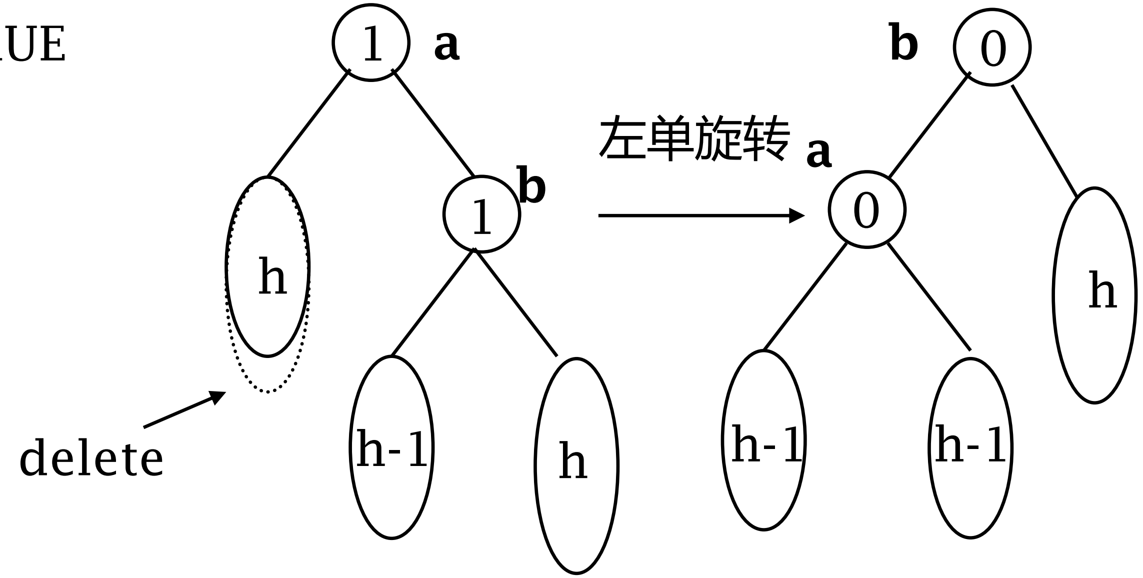
- 第三种情况是当前结点 a 平衡因子不为 0，且它的较低的子树被缩短，结点 a 必然不再平衡
- 假设其较高子树的根结点为 b，出现下面三种情况
  - 情况 3.1： b 的平衡因子为 0
    - 单旋转
    - modified=FALSE



情况3.1

AVL 树结点的删除过程（续）

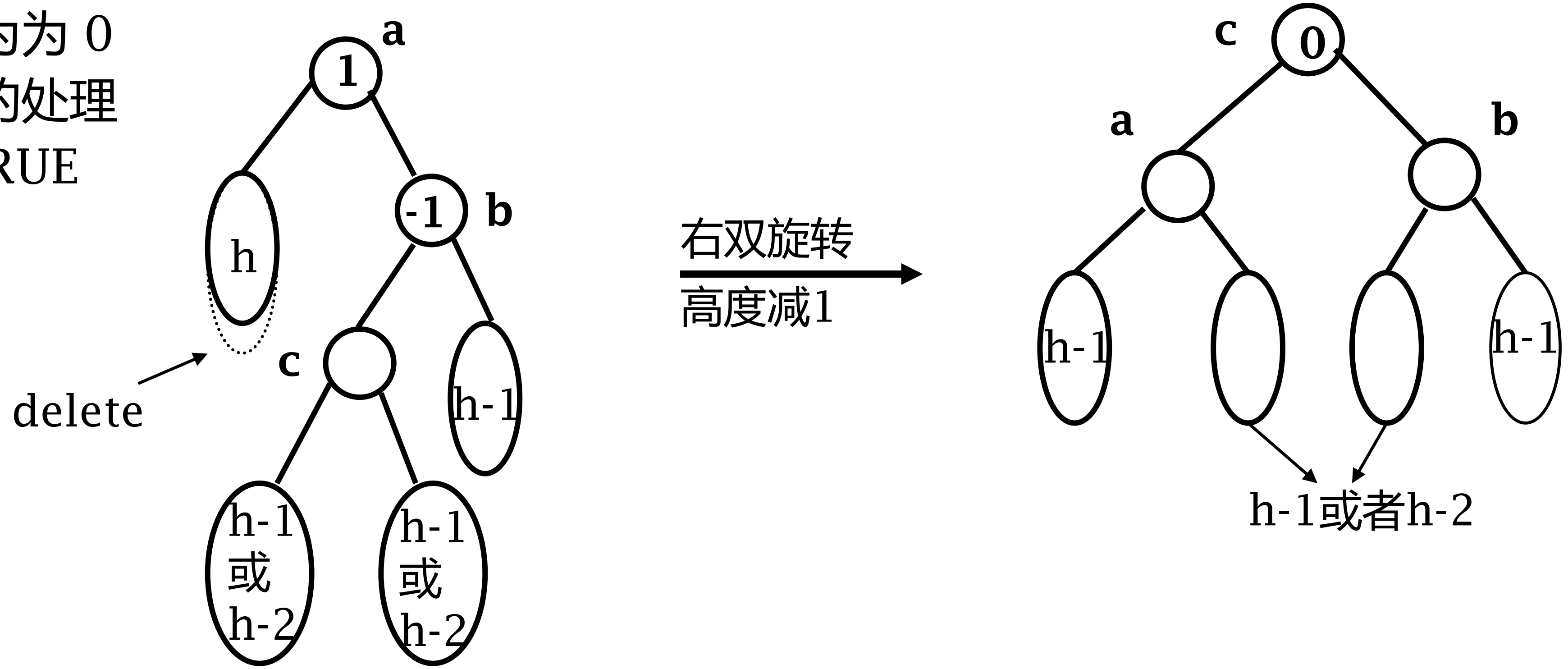
- 第三种情况是当前结点 a 平衡因子不为 0，且它的较低的子树被缩短，结点 a 必然不再平衡
  - 情况3.2：b 的平衡因子与 a 的平衡因子相同
    - 单旋转
    - 结点 a、b 平衡因子都变为0
    - modified=TRUE



情况3.2

AVL 树结点的删除过程（续）

- 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
  - 新的根结点平衡因为为 0
  - 其他结点应做相应的处理
  - 并且 modified=TRUE

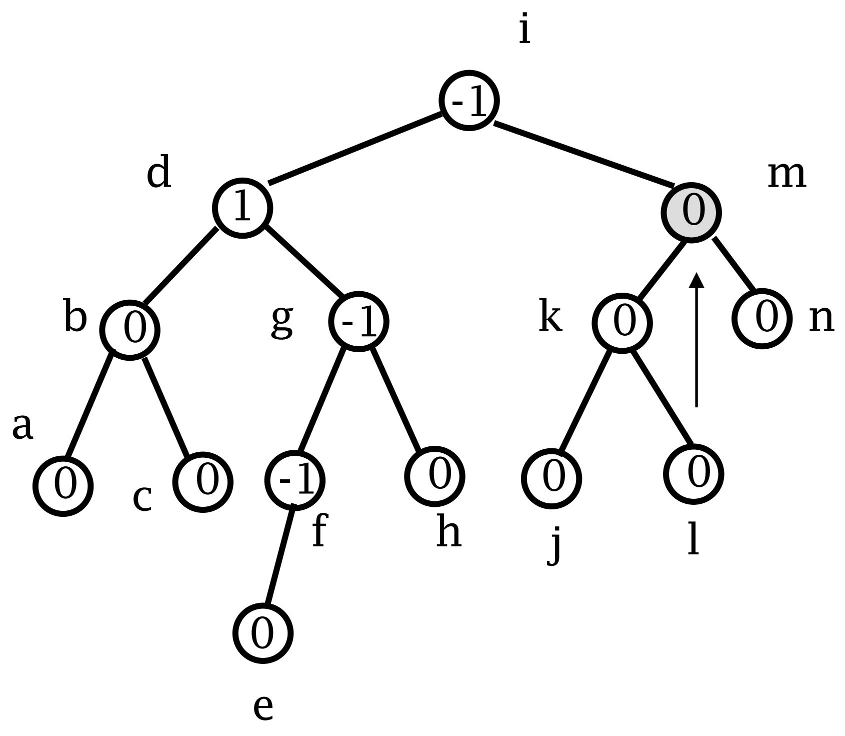


情况3.3

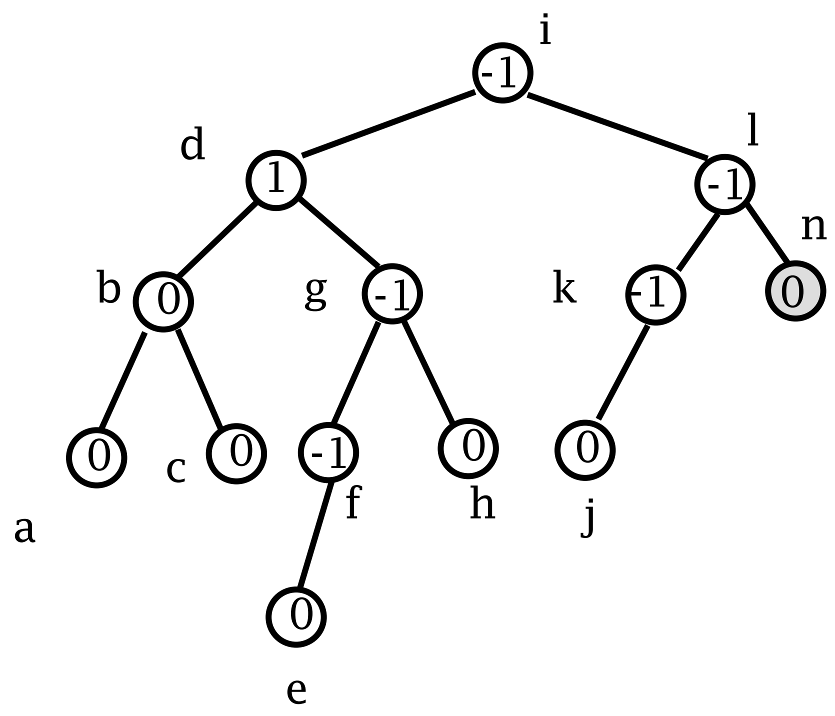
删除后的连续调整

- 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)
- 连续调整
  - 调整可能导致祖先结点发生新的不平衡
  - 这样的调整操作要一直进行下去，可能传递到根结点为止

AVL 树删除的例子

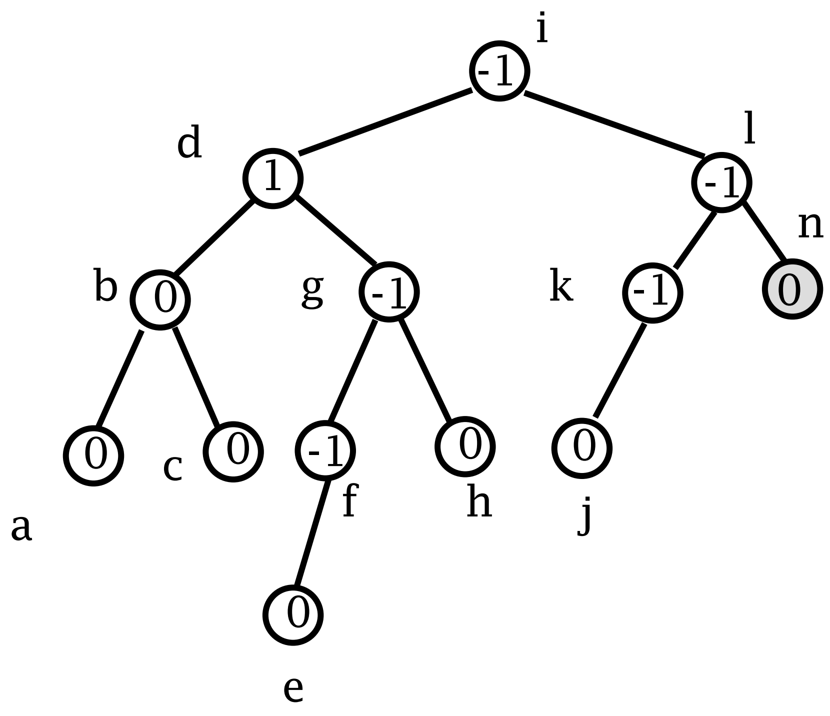


(a) 删除结点m，则需要使用其中序前驱l代替
（情况1）

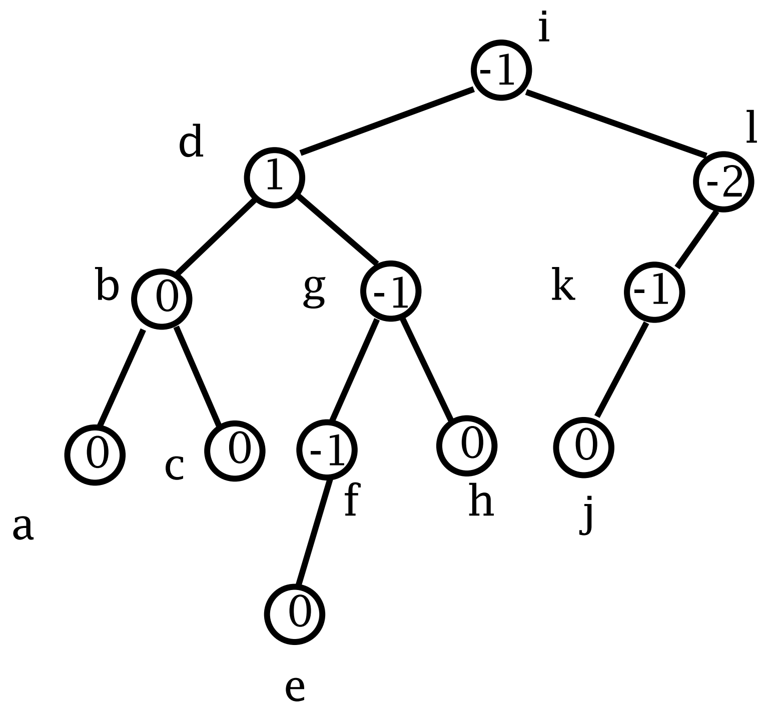


(b) 删除结点n
（情况3.2）

AVL 树删除的例子

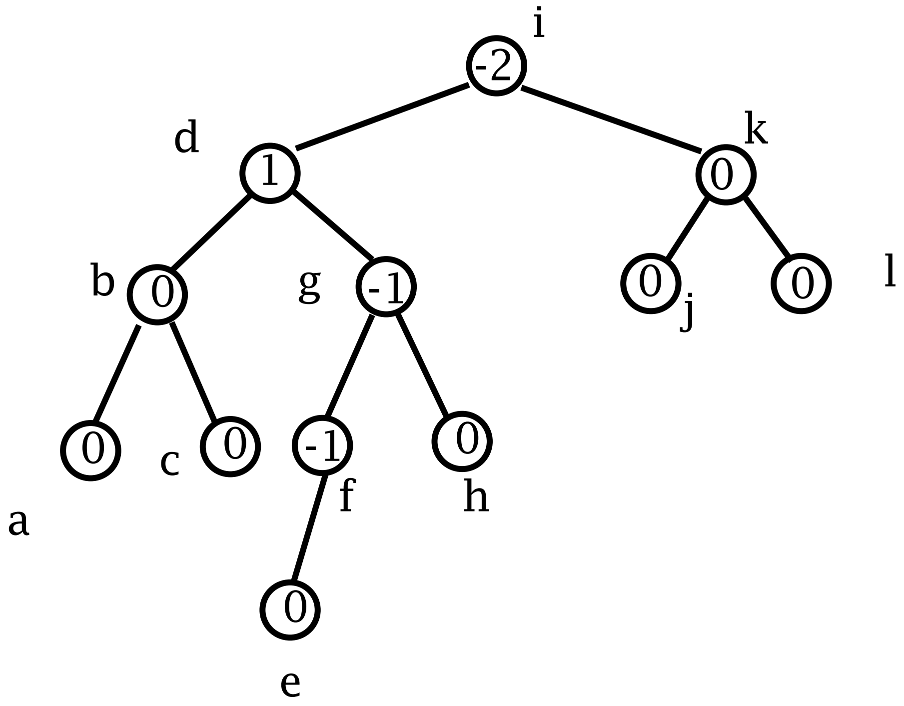


(b) 删除结点n
（情况3.2）

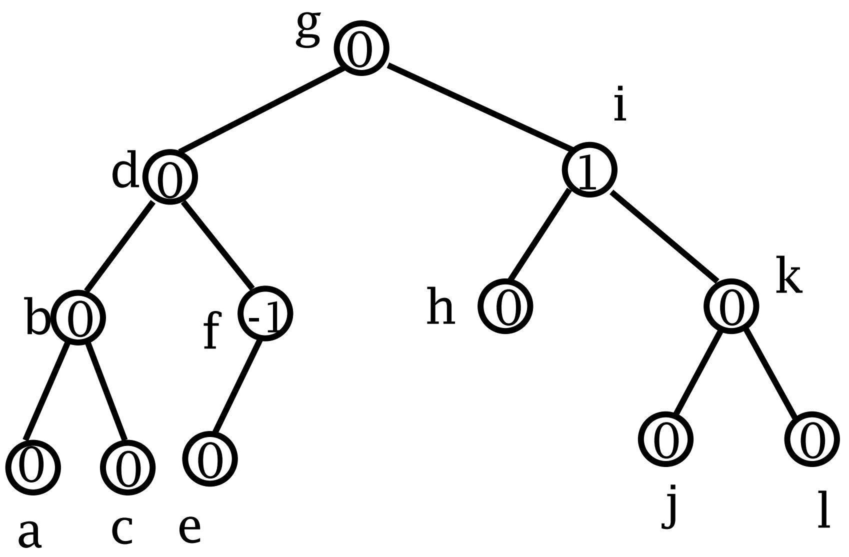


(c) 需要以l为根进行LL单旋转
（情况3.2）

AVL 树删除的例子



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



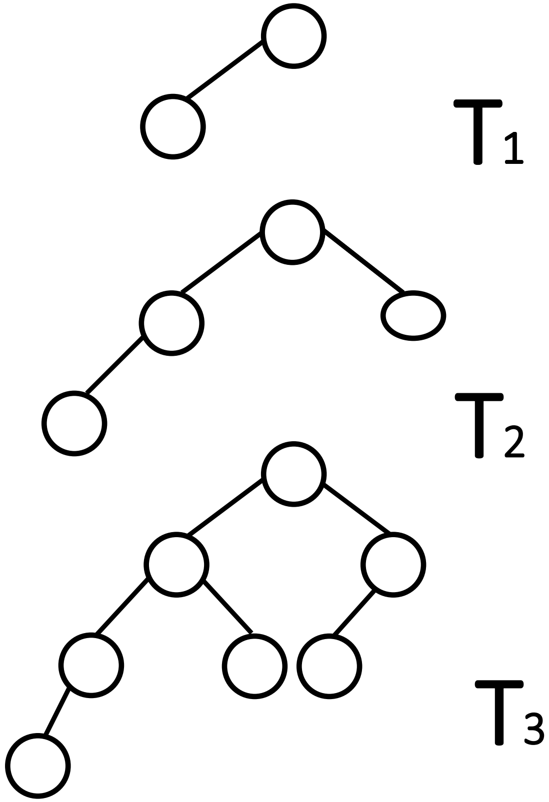
(e) 调整完毕，AVL树重新平衡

AVL 树的高度

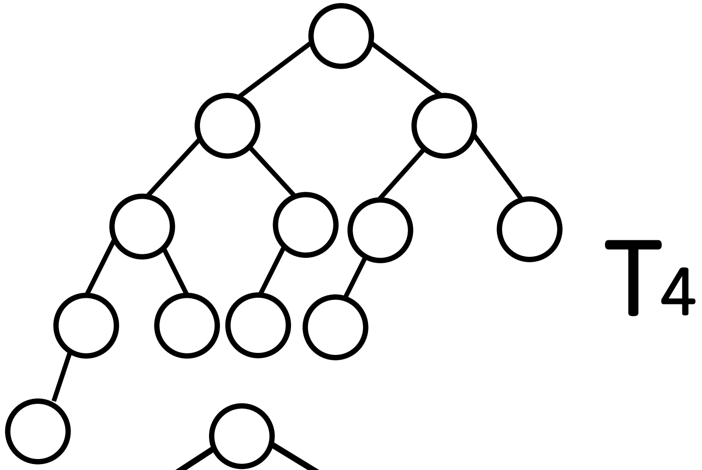
- 具有 n 个结点的 AVL 树高度一定是 O (log n)
- n 个结点的 AVL 树的最大高度不超过 K$\log_2 n$
    - 这里 K 是一个小的常数
- 最接近于不平衡的 AVL 树
    - 构造一系列 AVL 树 T_1，T_2，T_3，… 。

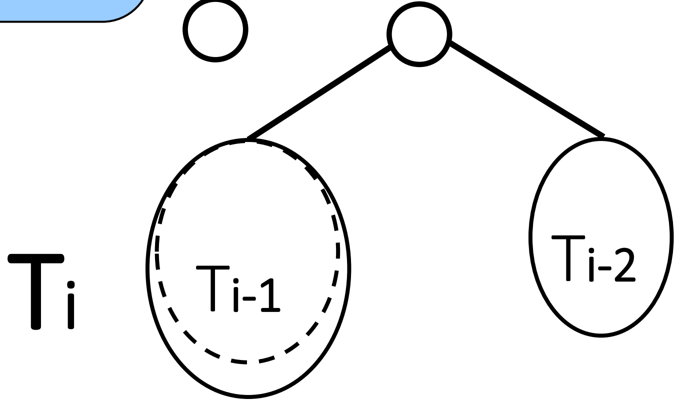
12.5 改进的二叉搜索树

Ti的高度是i



每棵具有高度i的其它AVL树都比Ti的结点个数多





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，删除一个结点都会不平衡

高度的证明（推理）

- 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 数的那些关系：

$$F(0) = 0$$
$$F(1) = 1$$
$$F(i) = F(i-1) + F(i-2)$$

高度的证明（推理续）

- 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

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

高度的证明（结果）

- 解出高度 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 $\log_{\varphi}X=\log_2 X/\log_2\varphi$ 和 $\log_2\varphi\approx 0.694$ ，求出近似上限

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

  - t (i) =n
  - 所以 n 个结点的 AVL 树的高度一定是 O (log *n*)

AVL 树的效率

- 检索、插入和删除效率都是 $O(\log_2 n)$
  - 具有 n 个结点的 AVL 树的高度一定是 $O(\log n)$
- 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

思考

- 是否可以修改 AVL 树平衡因子的定义，例如允许高度差为 2？
- 对比红黑树、AVL 树的平衡策略，哪个更好？
  - 最差情况下的树高
  - 统计意义下的操作效率
  - 代码的易写、易维护

数据结构与算法

谢谢聆听

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