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)

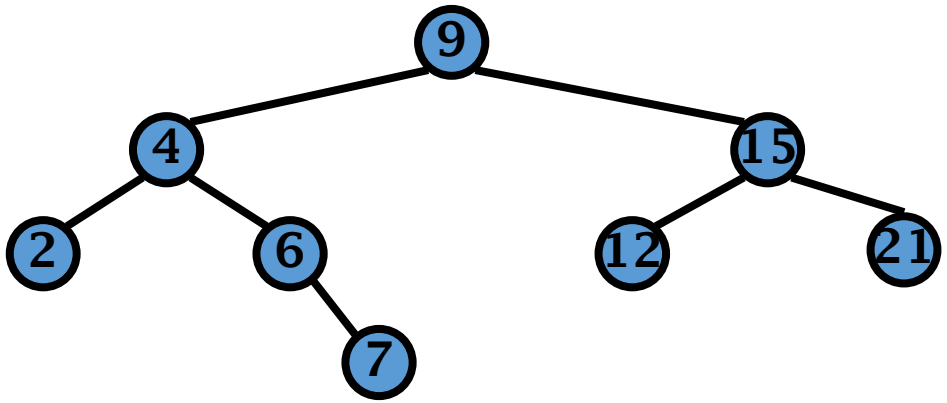
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg

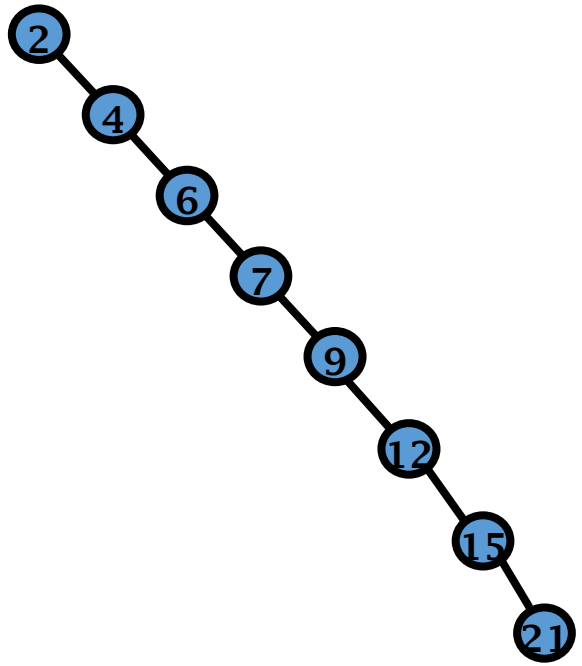
第十二章 高级数据结构

- 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

12.3 Trie 树

- 理想状况：插入、删除、查找时间代价为 O(log*n*)
- 输入 9, 4, 2, 6, 7, 15, 12, 21
- 输入 2, 4, 6, 7, 9, 12, 15, 21





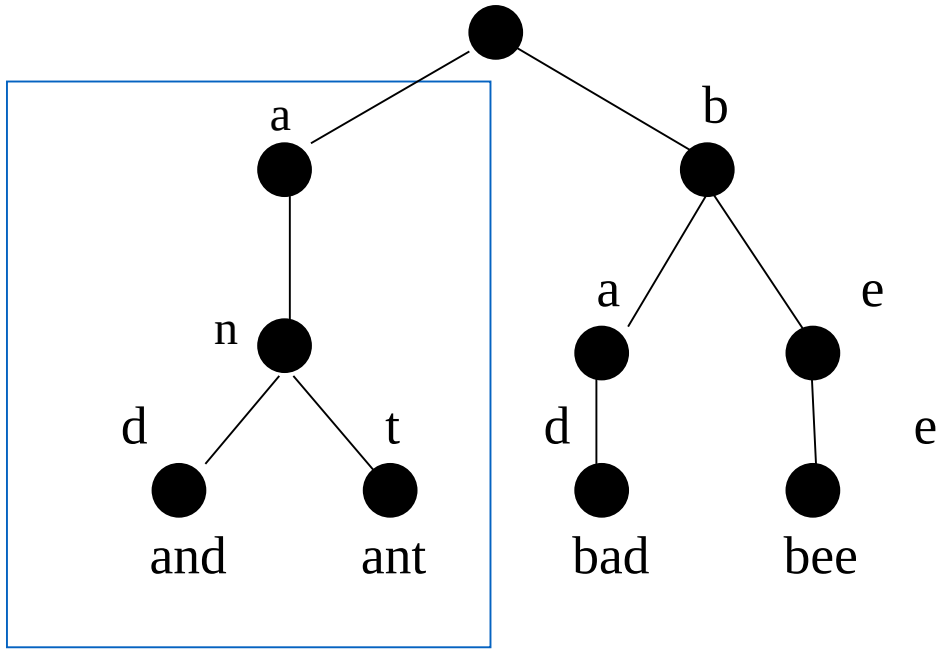
Trie 结构

- 关键码对象空间分解
- “trie”这个词来源于“retrieval”
- 主要应用
  - 信息检索 (information retrieval)
  - 自然语言大规模的英文词典
- 字符树——26叉Trie
- 二叉Trie树
  - 用每个字母（或数值）的二进制编码来代表
  - 编码只有0和1

英文字符树——26叉Trie

存单词and、ant、bad、bee

“an”子树代表相同前缀an-具有的关键码集合{and，ant}

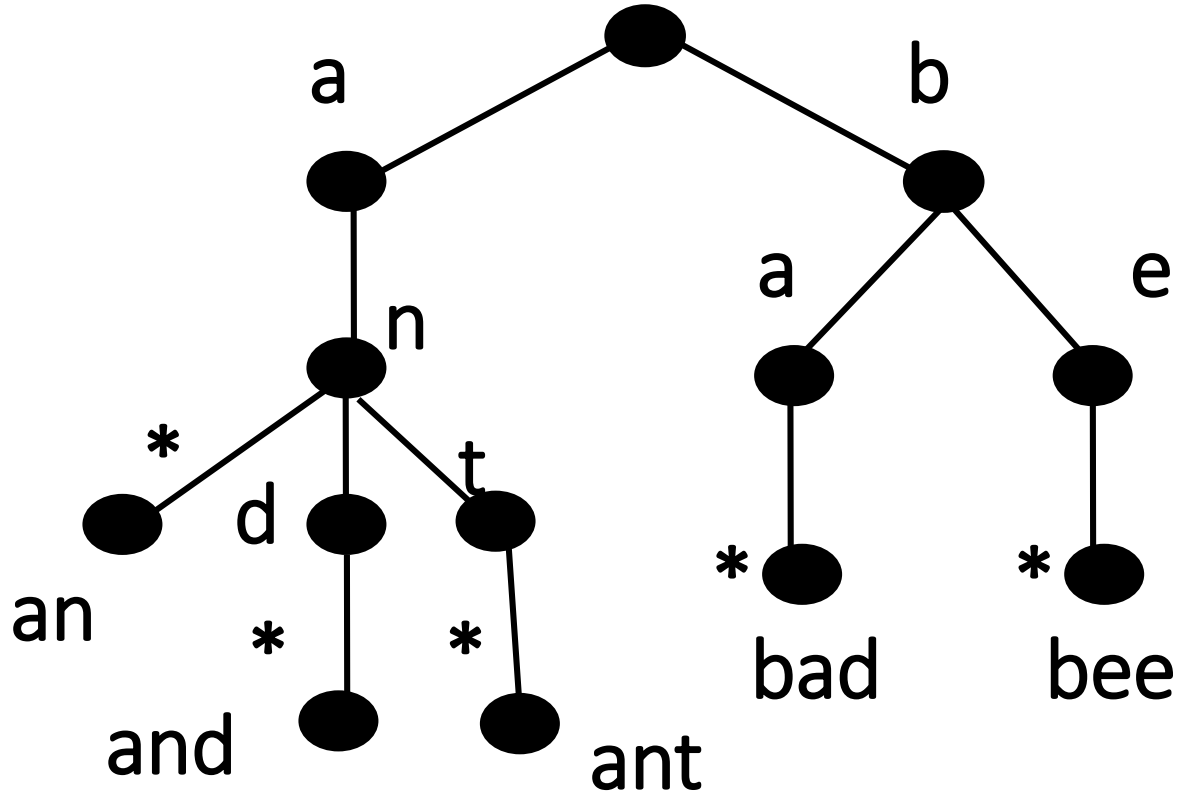


- 一棵子树代表具有相同前缀的关键码的集合

12.4 Trie 树

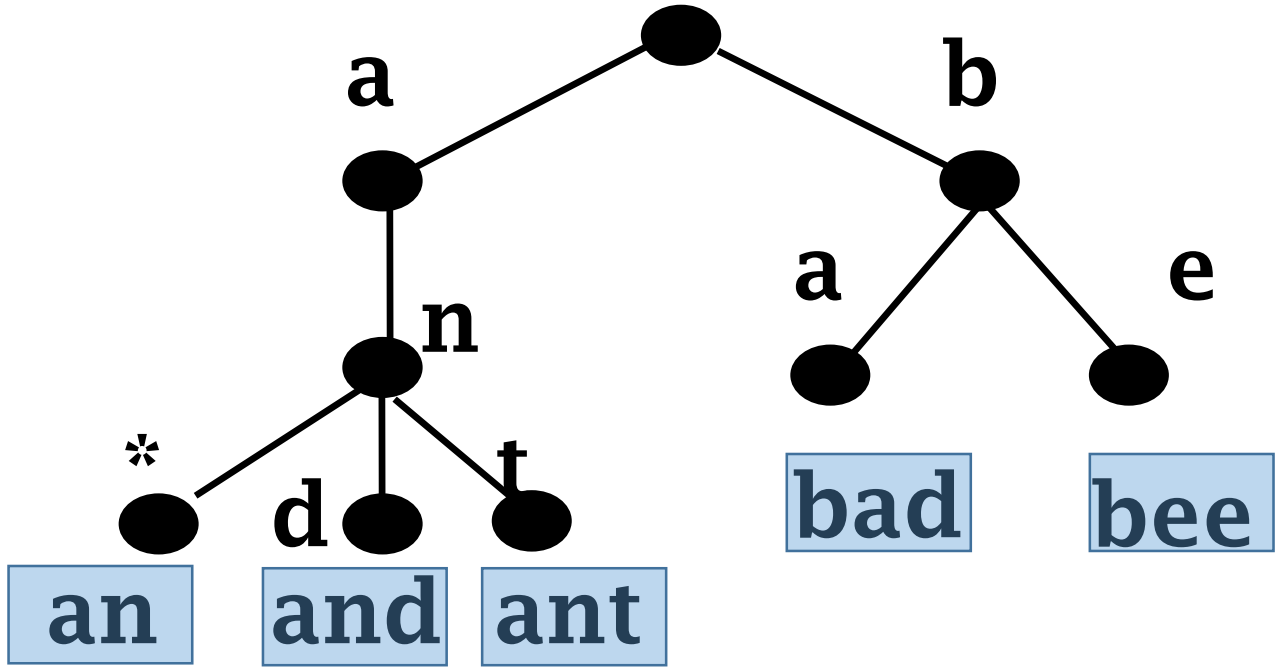
不等长的字符树，加“*”标记

存储单词 an、and ant、bad、bee



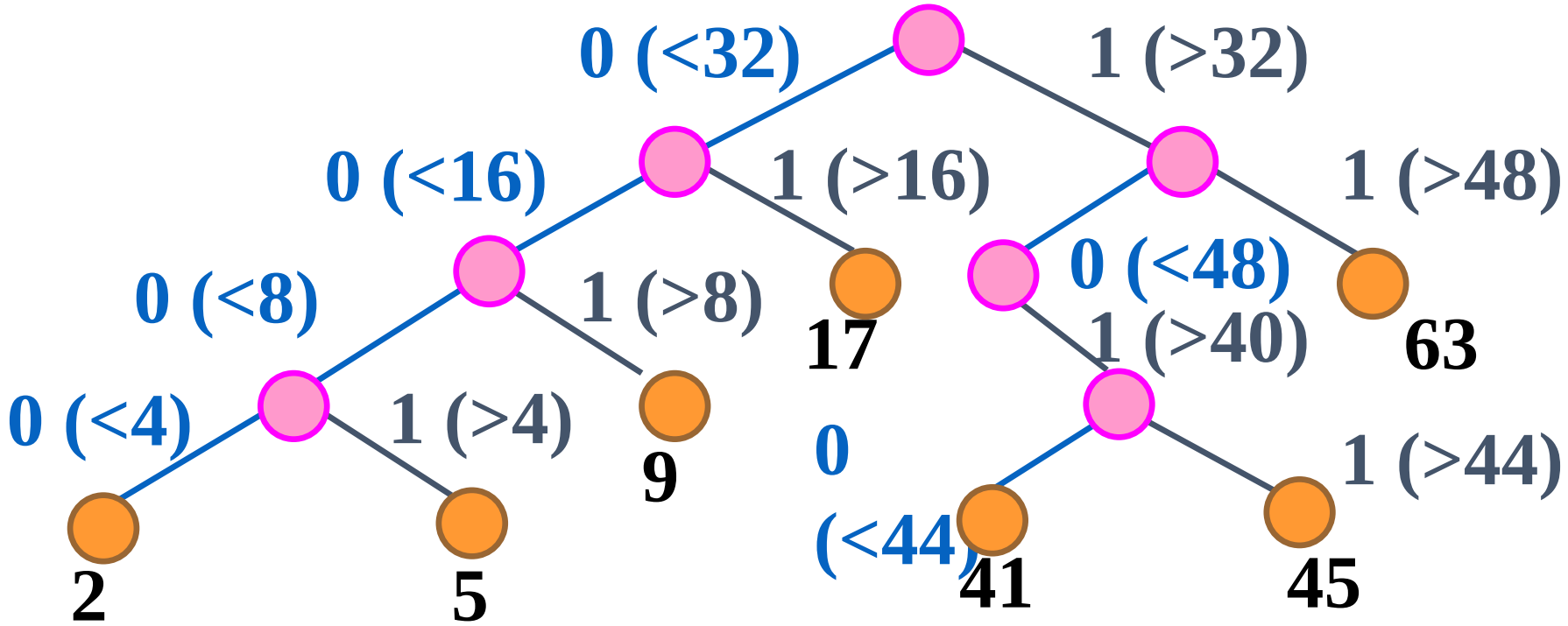
压缩靠近叶结点的单路径

存储单词 an、and、ant、bad、bee

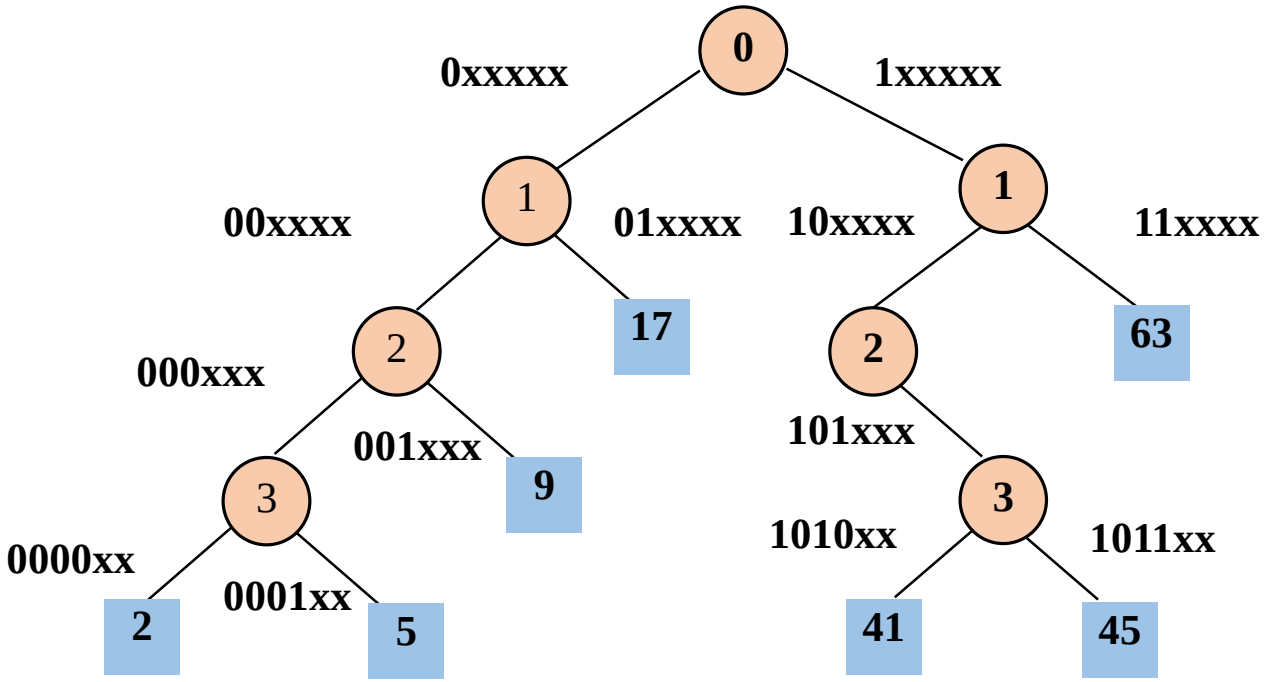


二叉Trie 结构

元素为2、5、9、17、41、45、63



PATRICIA 结构图



压缩

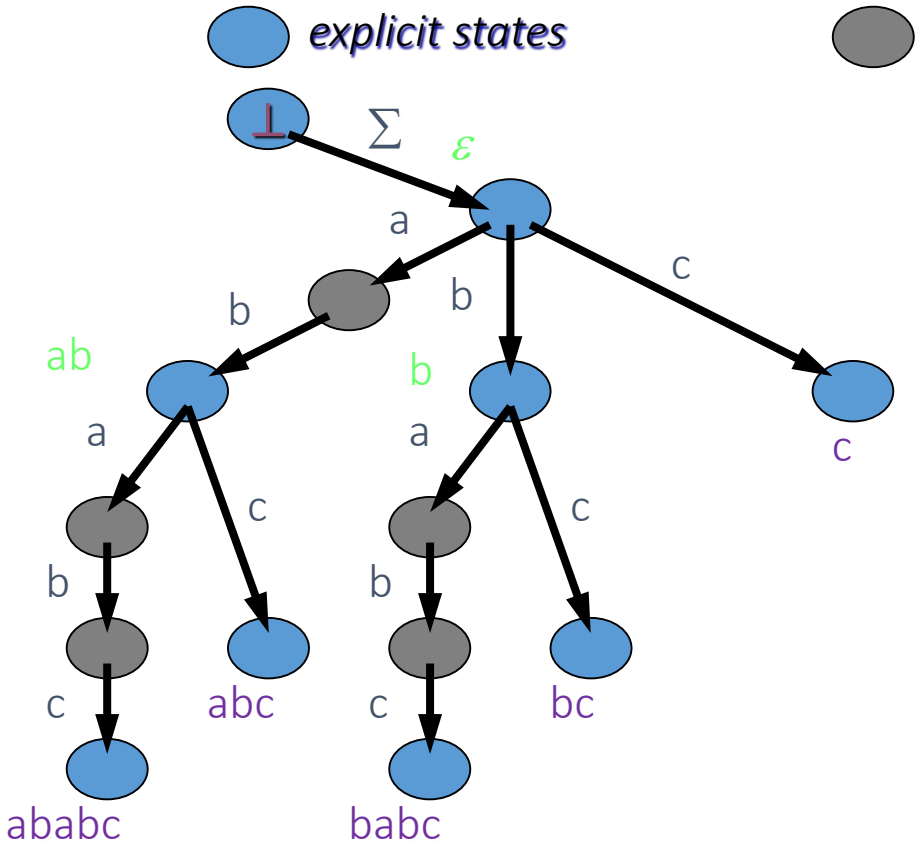
编码： 2：000010 5：000101 9：001001
17：010001 41：101001 45：101101 63：111111

PATRICIA的特点

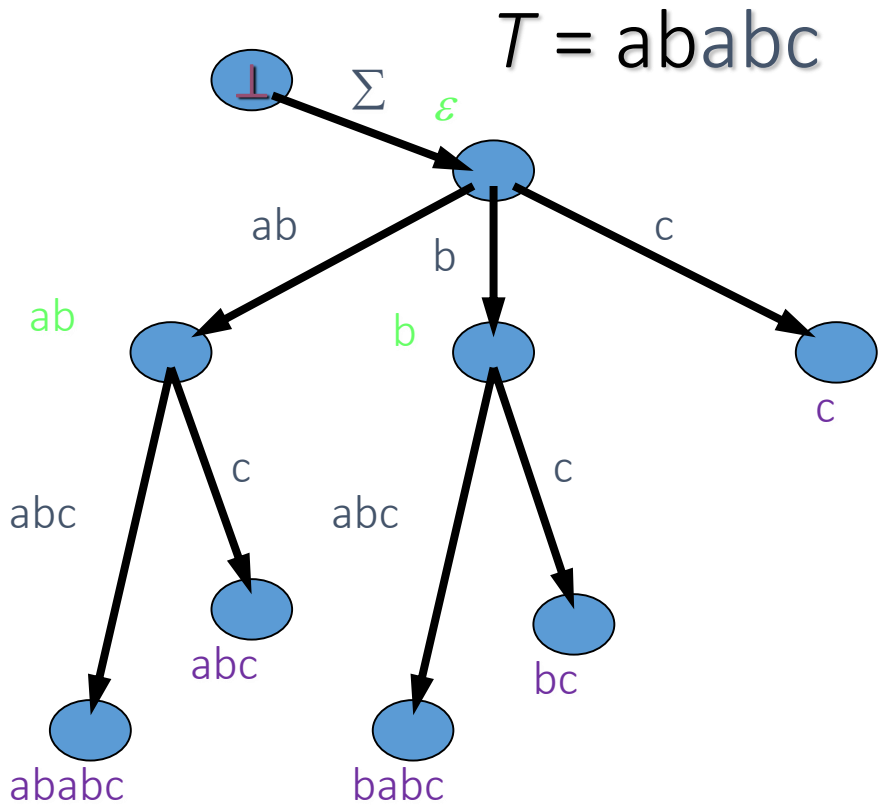
- 改进后的压缩 PATRICIA 树是满二叉树
  - 每个内部结点都代表一个位的比较
  - 必然产生两个子结点
- 一次检索不超过关键码的位个数

12.4 Trie 树

Suffix Trees



Suffix Trie



Suffix Tree

升序
ababc
abc
abc
bc
c

数组(Suffix Array)

5	ALAM$
1	ALAYALAM$
7	AM$
3	AYALAM$
6	LAM$
2	LAYALAM$
0	MALAYALAM$
8	M$
4	YALAM$
9	$

M A L A Y A L A M $

0 1 2 3 4 5 6 7 8 9

5	1	7	3	6	2	0	8	4	9

后缀数组

3	1	1	0	2	0	1	0	0	-

最长公共前缀数组

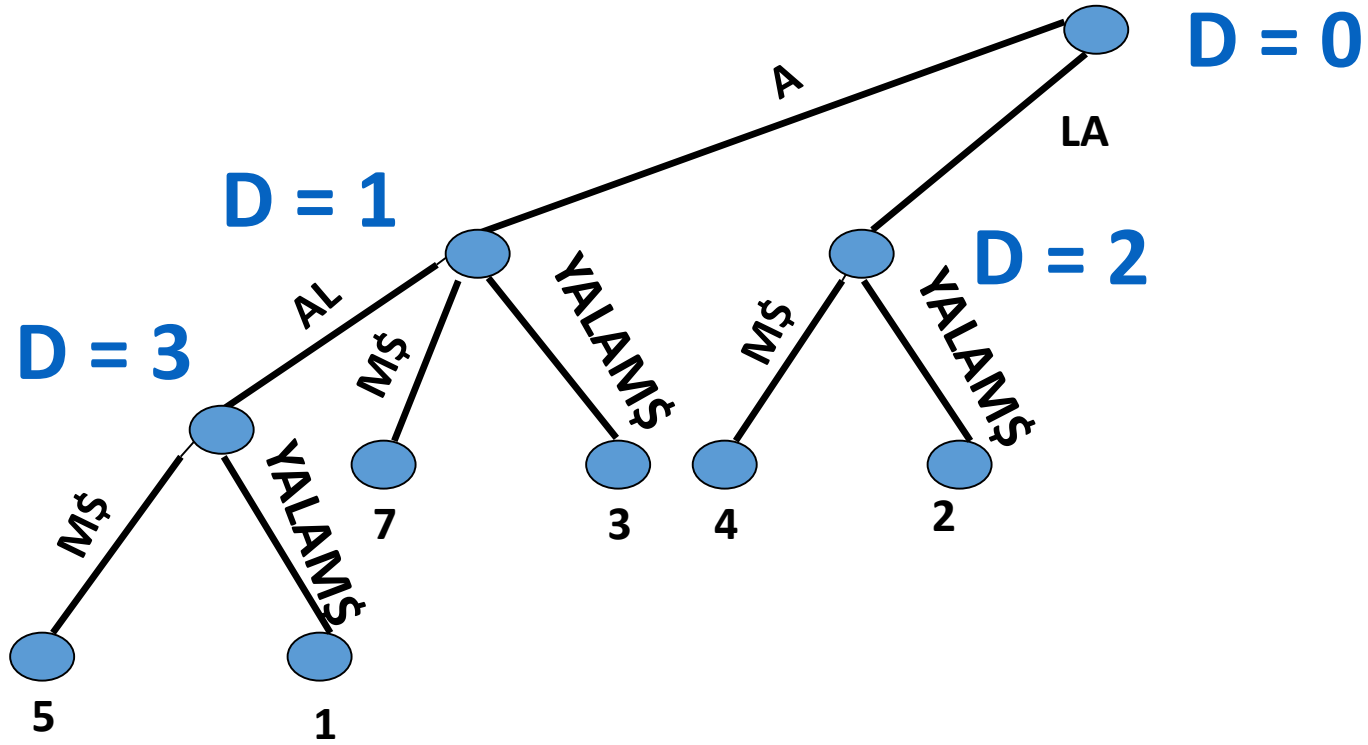
后缀5和1共享 “ALA”

后缀1和7共享 “A”

LCP总是相邻的

12.4 Trie 树

5	ALAM$
1	ALAYALAM$
7	AM$
3	AYALAM$
6	LAM$
2	LAYALAM$
0	MALAYALAM$
8	M$
4	YALAM$
9	$



SA	5	1	7	3	6	2	0	8	4	9
lcp	**3**	**1**	**1**	**0**	**2**	**0**	**1**	**0**	**0**	-

思考

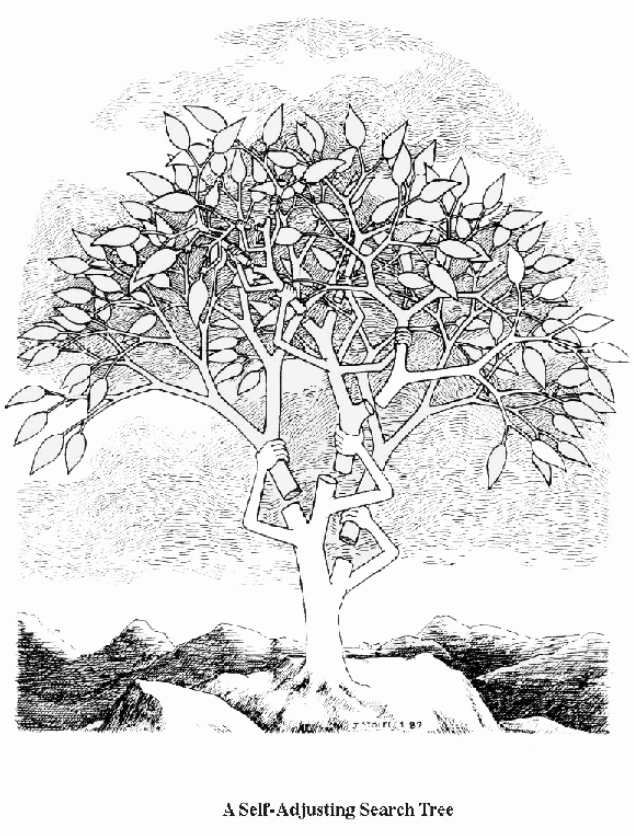
- 中文是否适合组织字符树？是否适合 PATRICIA Trie 结构？
- 查阅后缀树、后缀数组的文献，思考其应用场景。

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

12.5.2 伸展树

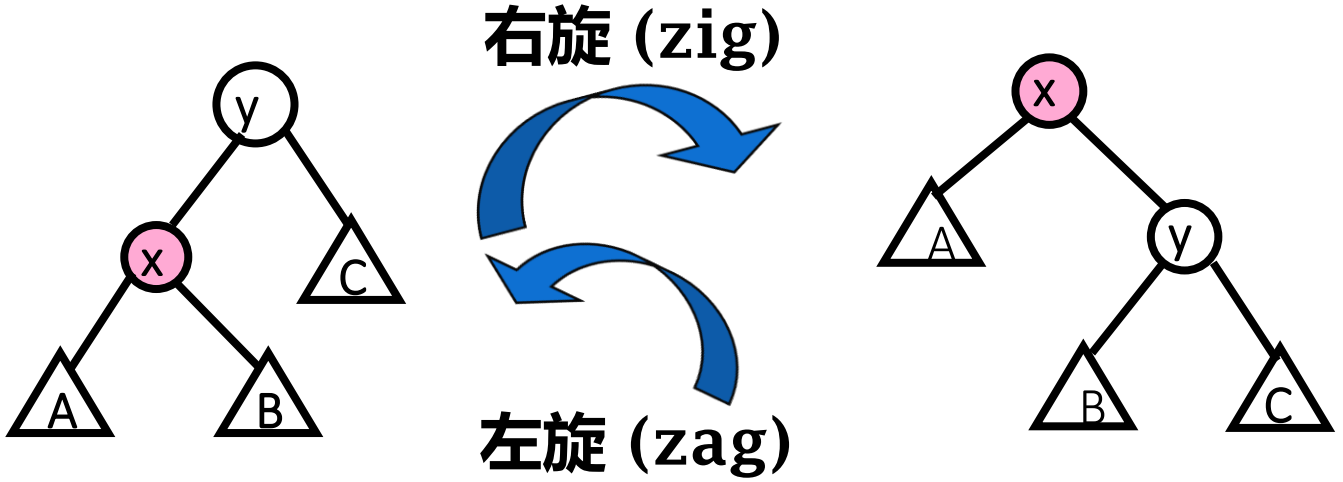
- 一种自组织数据结构
  - 数据随检索而调整位置
  - 汉字输入法的词表
- 伸展树不是一个新数据结构，而只是改进 BST 性能的一组规则
  - 保证访问的总代价不高，达到最令人满意的性能
  - 不能保证最终树高平衡



A Self-Adjusting Search Tree

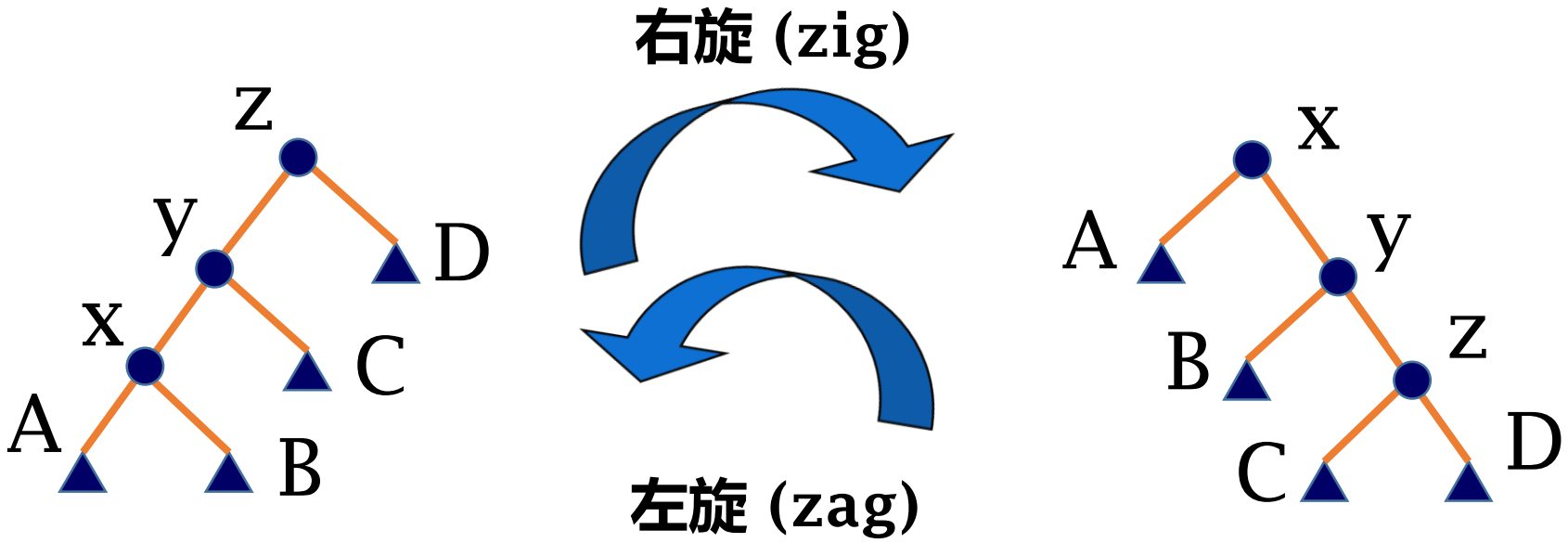
12.5.2 伸展树

- 单旋转
  - 结点与它的父结点交换位置，保持 BST 特性



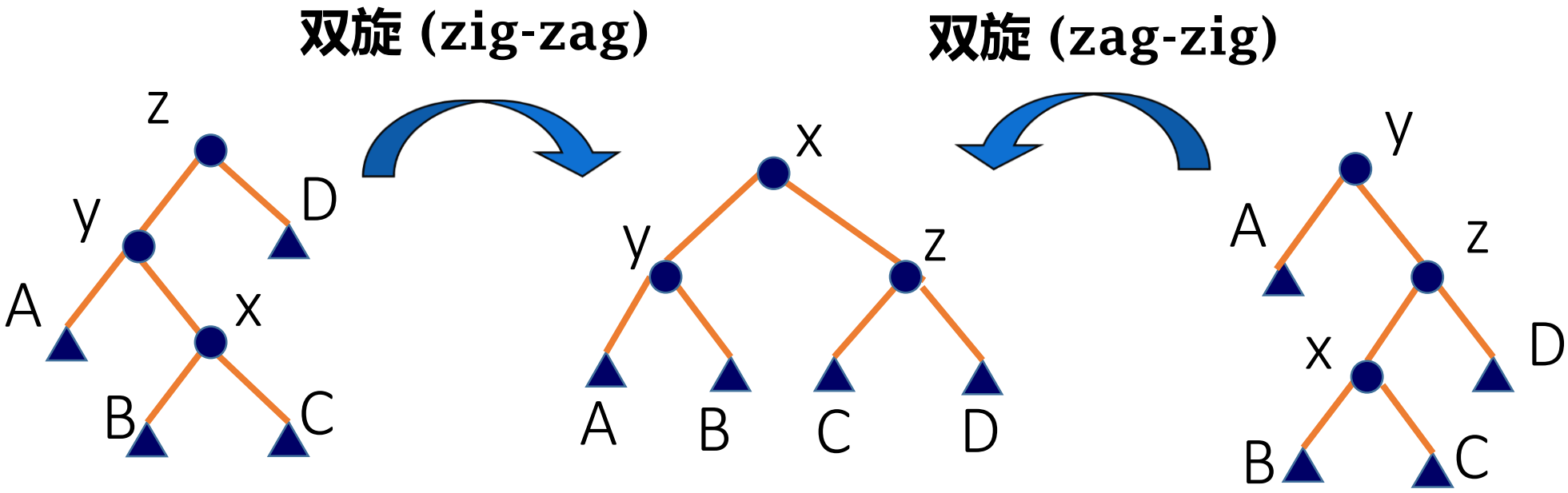
12.5.2 伸展树

- LL 和 RR 双旋：保持 BST 的中序性质



12.5.2 伸展树

- LR 和 RL 双旋：保持 BST 的中序性质



展开（splaying）

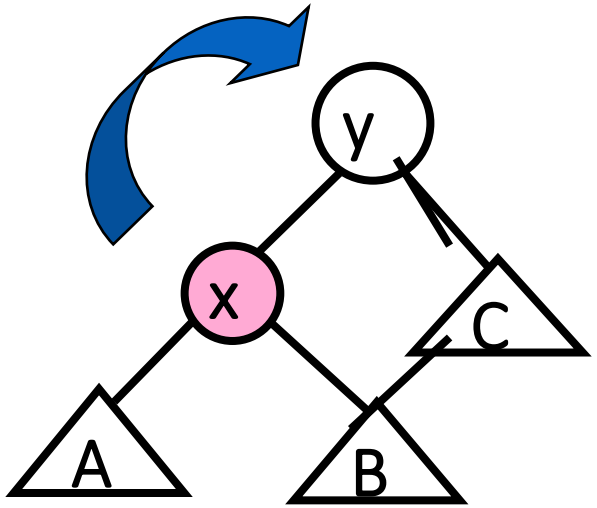
- 访问一次结点 (例如结点 x) ，完成一次称为展开的过程
  - x 被插入、检索时，把结点 x 移到 BST 的根结点
  - 删除结点 x 时，把结点 x 的父结点移到根结点
- 像在 AVL 树中一样，结点 x 的一次展开包括一组旋转（rotation）
  - 调整结点 x、父结点、祖父结点的位置
  - 把 x 移到树结构中的更高层

单旋转 (single rotation)

- x 是根结点的直接子结点时
  - 把结点 x 与它的父结点交换位置
  - 保持 BST 特性

x、y 为内部结点编号，不是值大小

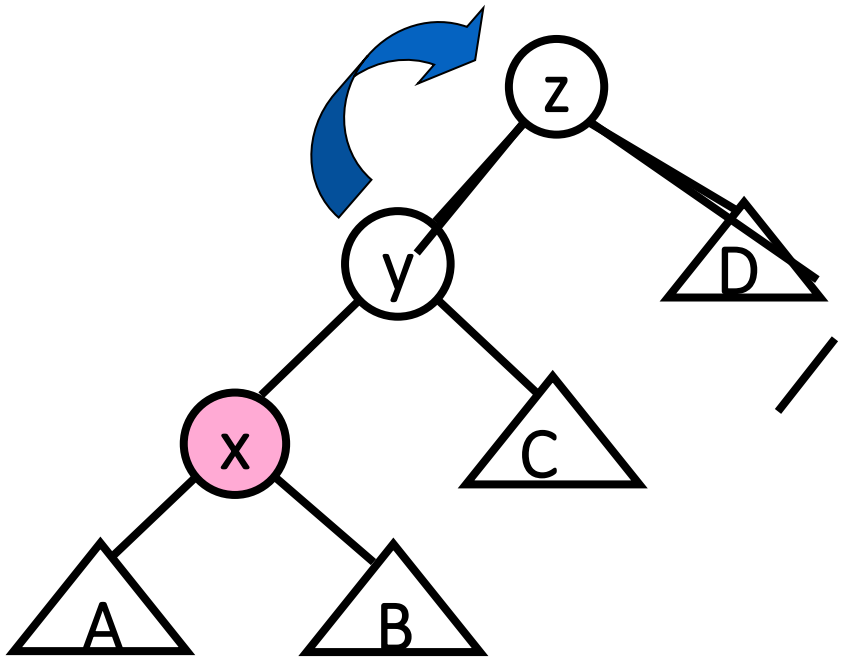
A、B、C 代表子树，有大小顺序



双旋转（double rotation）

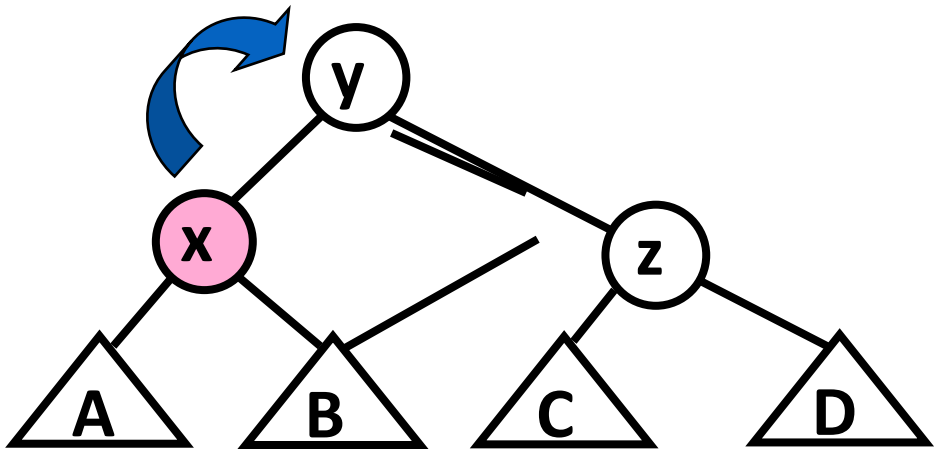
- 双旋转涉及到
  - 结点 x
  - 结点 x 的父结点 (称为 y)
  - 结点 x 的祖父结点 (称为 z)
- 把结点 x 在树结构中向上移两层
- 一字形旋转（zigzig rotation）
  - 也称为同构调整 (homogeneous configuration)
- 之字形旋转（zigzag rotation）
  - 也称为异构调整 (heterogeneous configuration)

一字形旋转图示

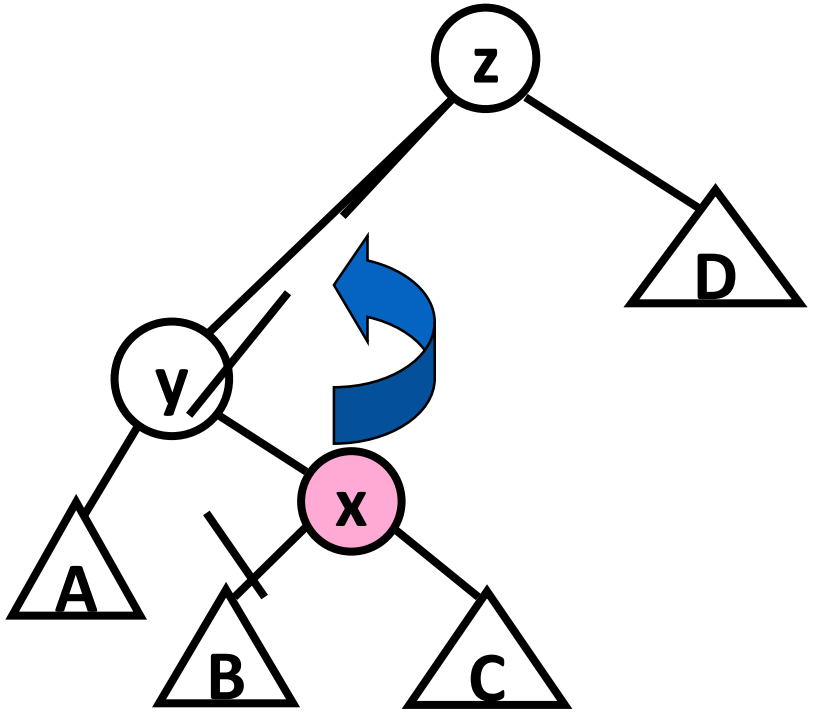


结点 x 是 y 的左子结点
结点 y 是 z 的左子结点

一字形旋转图示

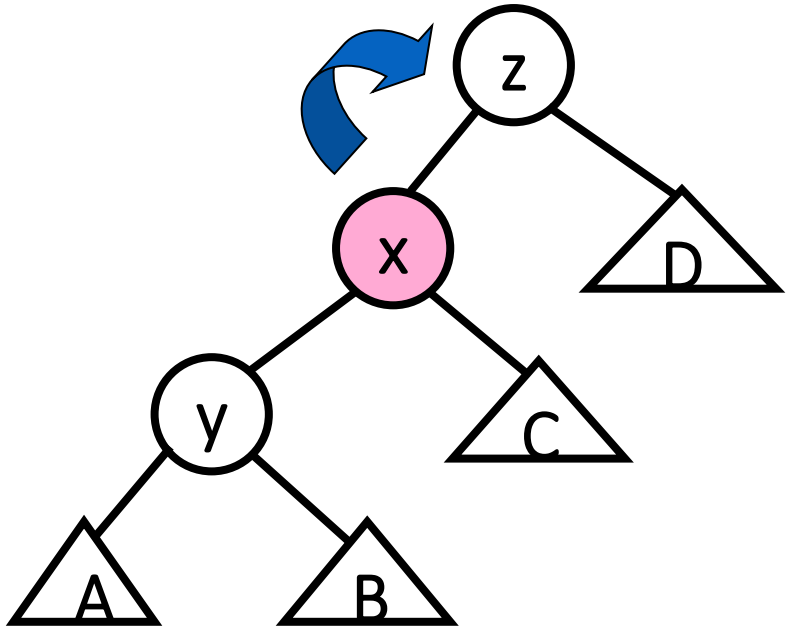


之字形旋转图示

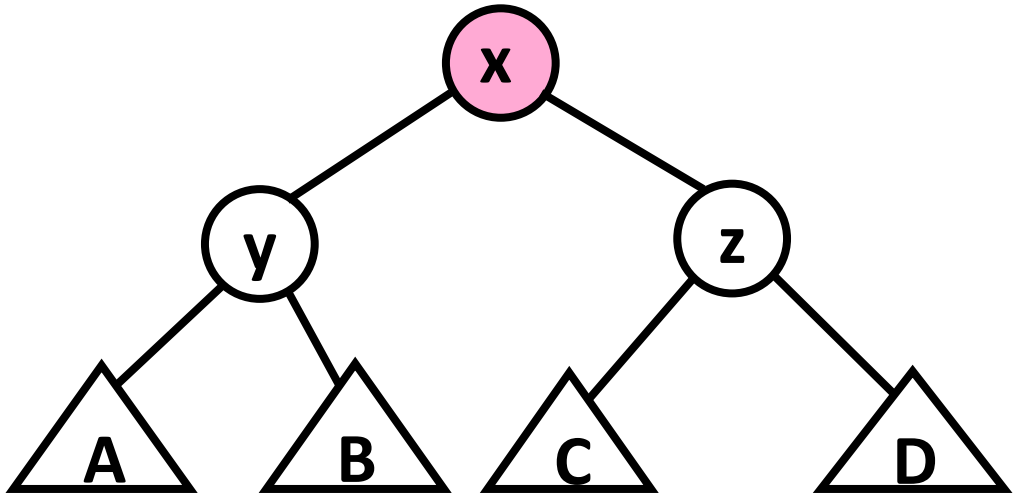


结点 x 是 y 的右子结点
结点 y 是 z 的左子结点

之字形旋转图示



之字形旋转图示



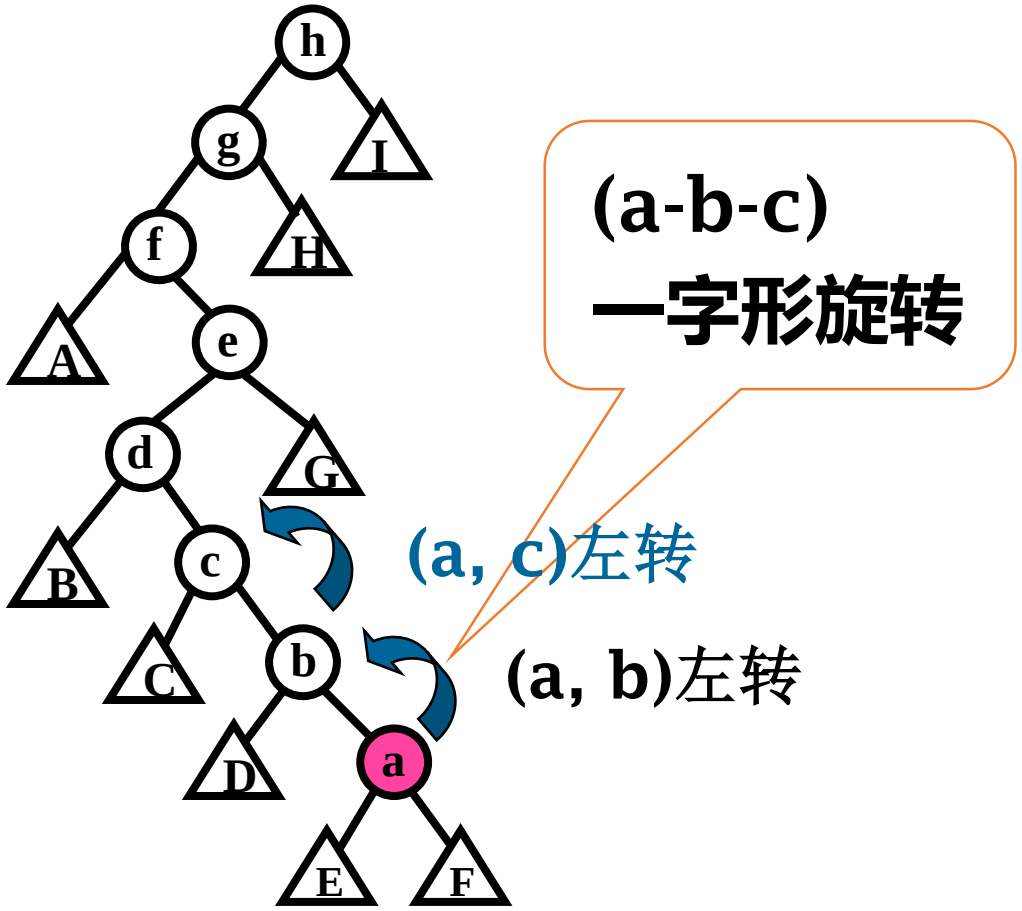
两种旋转的不同作用

- 之字形旋转
  - 把新访问的记录向根结点移动
  - 使子树结构的高度减 1
  - 趋向于使树结构更加平衡
- 一字形提升
  - 一般不会降低树结构的高度
  - 只是把新访问的记录向根结点移动

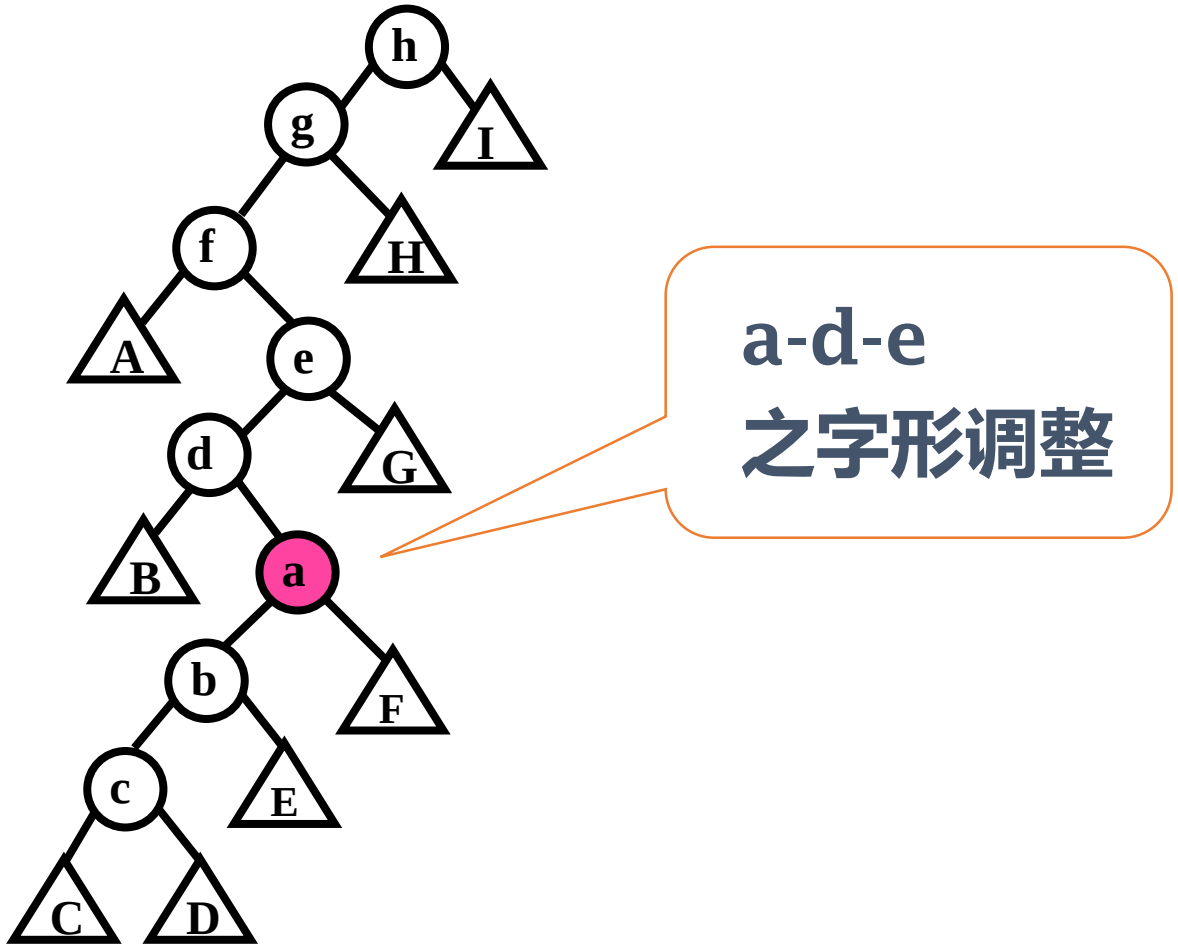
伸展树的调整过程

- 一系列双旋转
  - 直到结点 x 到达根结点或者根结点的子结点
- 如果结点x到达根结点的子结点
  - 进行一次单旋转使结点 x 成为根结点
- 这个过程趋向于使树结构重新平衡
  - 使访问最频繁的结点靠近树结构的根层
  - 从而减少访问代价

伸展树的调整过程



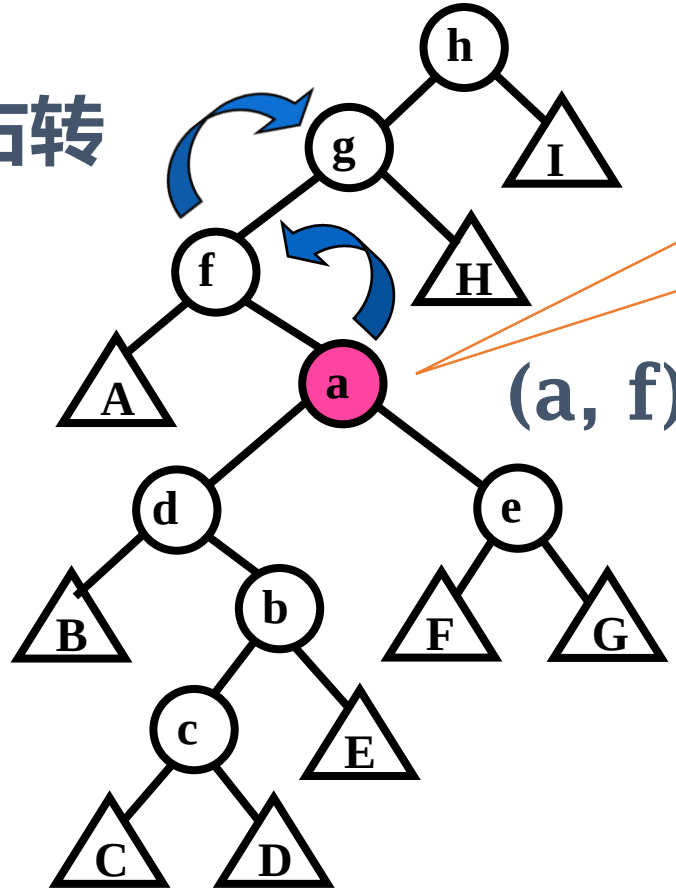
伸展树的调整过程



a-d-e
之字形调整

伸展树的调整过程

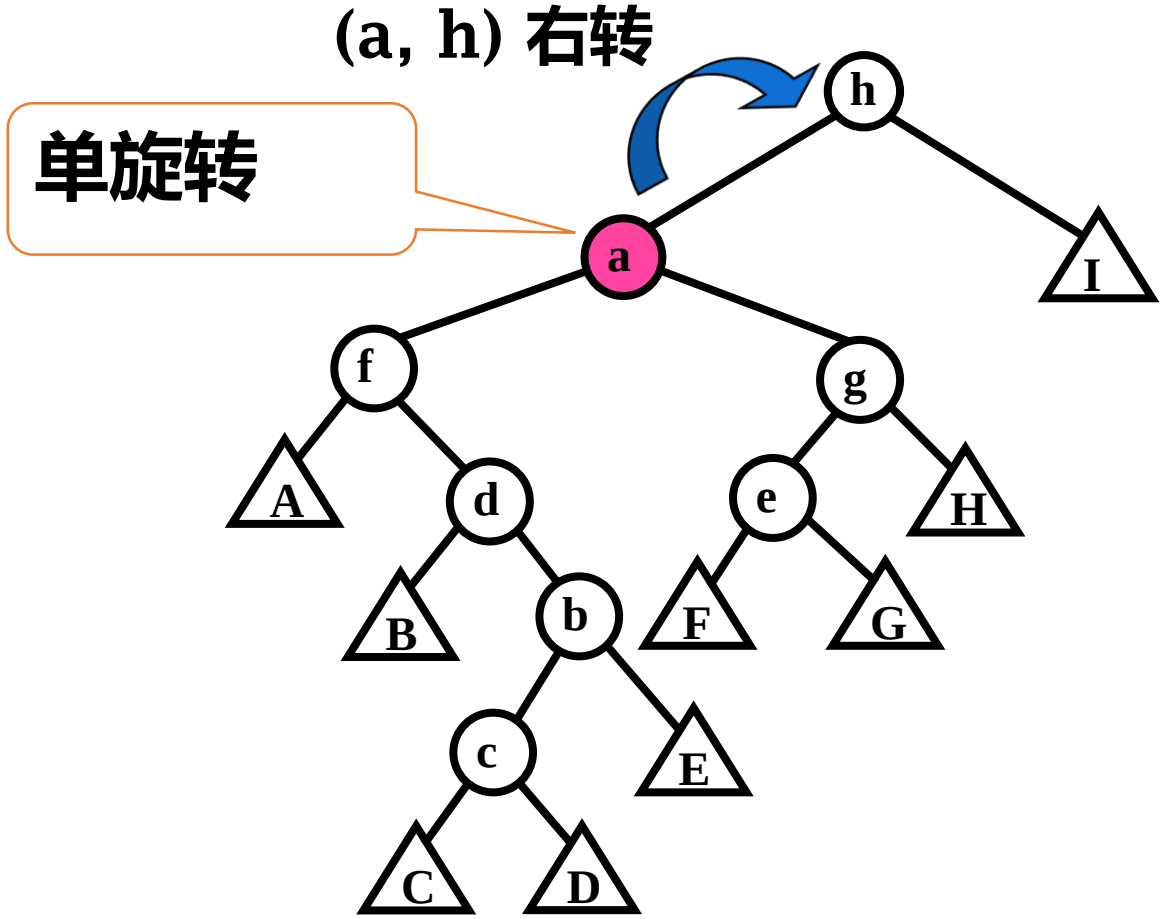
(a, g) 右转



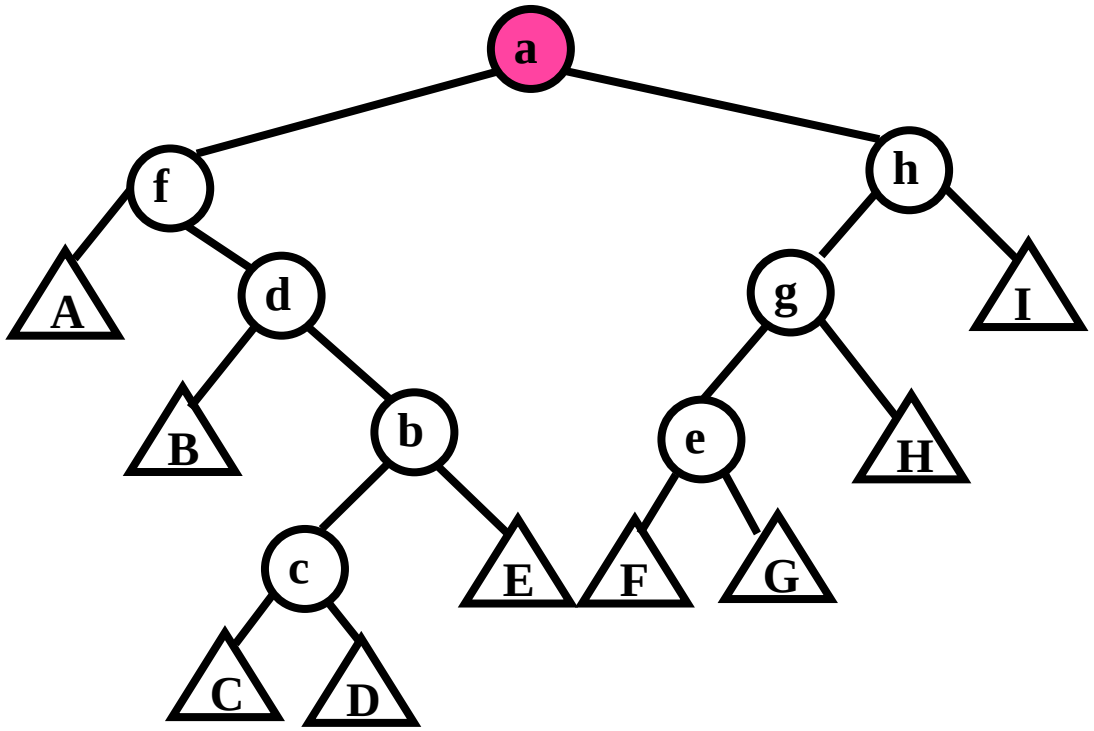
(a, f) 左转

(a-f-g)
之字形旋转

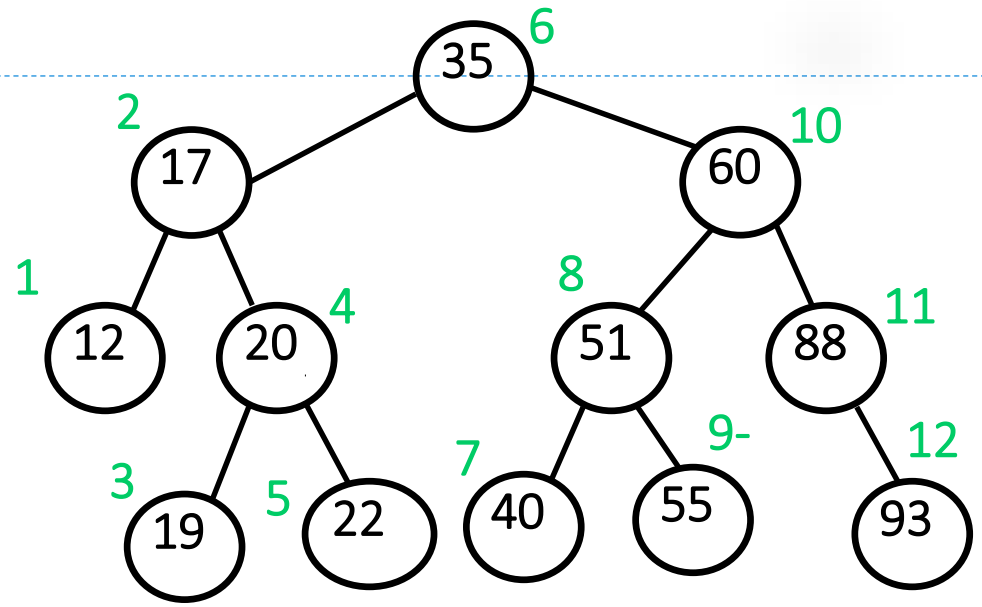
伸展树的调整过程



伸展树的调整过程



Splay 树的操作



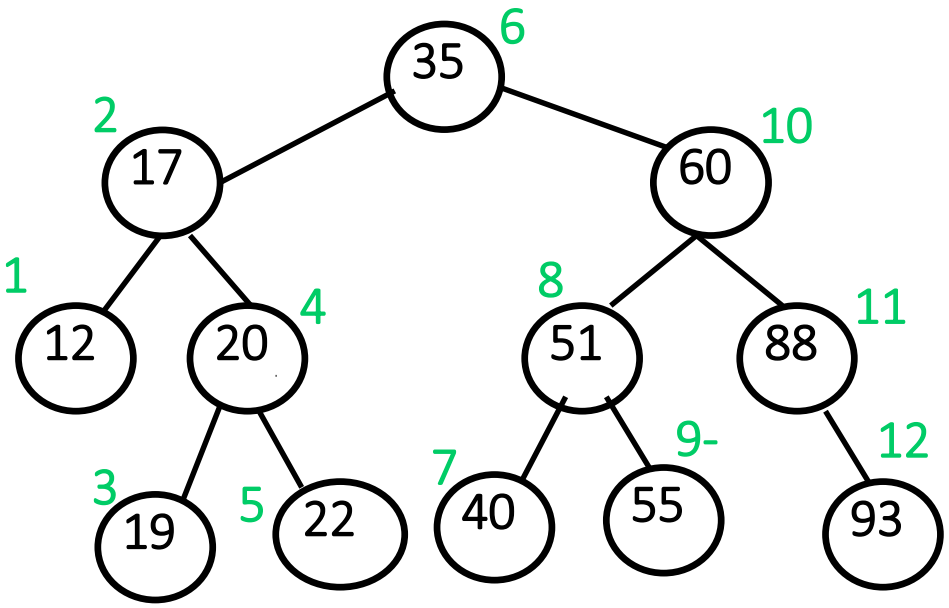

```
struct TreeNode
{
    int key;
    ELEM value;
    TreeNode *father, * left, *right;
};
```

```
Splay(TreeNode *x, TreeNode *f);    // 把 x 旋转到祖先 f 下面
Splay(x, NULL)；                    // 把 x 旋转为根
Find(int k) ;                               // 查询 k
Insert(int k)；                             // 插入值 v
Delete(TreeNode *x) ;               // 删除 x 结点
DeleteTree(TreeNode *x);            // 删除 x 子树
```

```
void Splay (TreeNode *x, TreeNode *f) {
    while (x->parent != f) {
        TreeNode *y = x-> parent, *z = y-> parent;
        if (z != NULL) {                          // 祖父不空
            if (z->lchild == y) {
                if (y->lchild == x)
                { Zig(y); Zig(x); }    // 一字型双右旋
                else { Zag(x); Zig(x); }          // x左旋上来，接着右旋
            } else {
                if (y->lchild == x)
                { Zig(x); Zag(x); }    // x右旋上来，接着左旋
                else { Zag(y); Zag(x); }          // 一字型双左旋
            }
        } else {
            if (y->lchild == x) Zig(x);    // 右单
            else Zag(x); } }                      // 左单旋
    if (x->parent == NULL) Root = x;
}
```

删除大于 u 小于 v 的所有结点

- 把 u 结点旋转到根
- 把 v 旋转为 u 的右儿子
- 删除 v 结点的左子树




```
void DeleteUV(TreeNode* rt, TreeNode* u, TreeNode* v )
{
    Splay(u, NULL);
    Splay(v, u);
    DeleteTree(v->lchild);
    v->lchild = NULL;
}
```

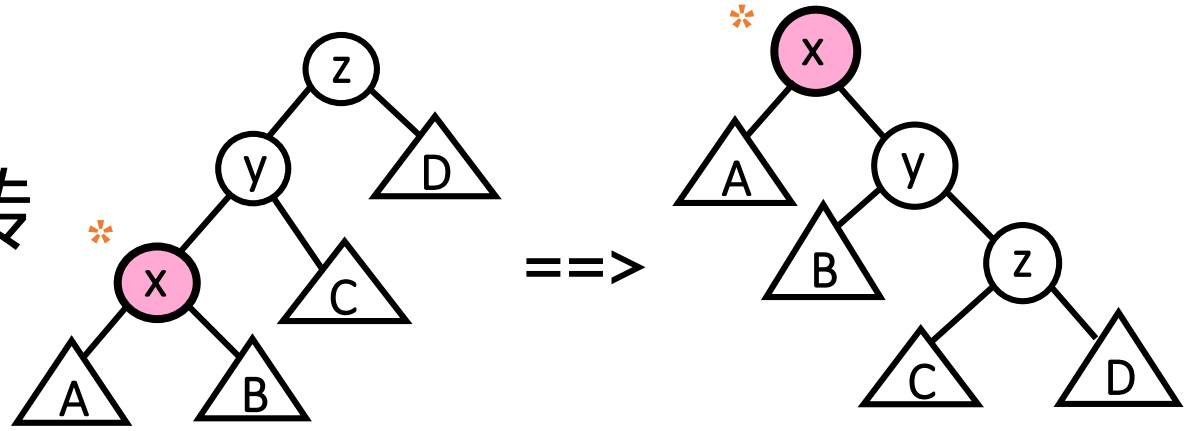
伸展树的效率

- n 个结点的伸展树
- 进行一组 m 次操作（插入、删除、查找操作），当 m≥n 时，总代价是 O(m logn)
  - 伸展树不能保证每一个单个操作是有效率的
  - 即每次访问操作的平均代价为 O(log n)
- 不要求掌握证明方法

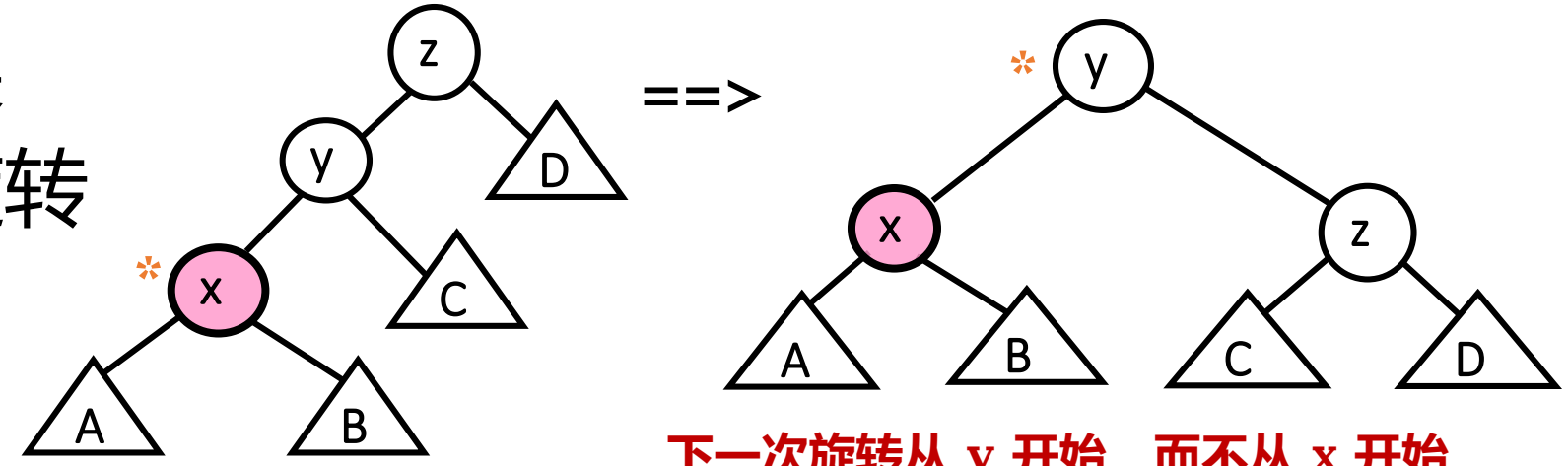
12.5 改进的二叉搜索树

思考：半伸展树

普通
一字旋转



半伸展
一字旋转



下一次旋转从 y 开始，而不从 x 开始

思考

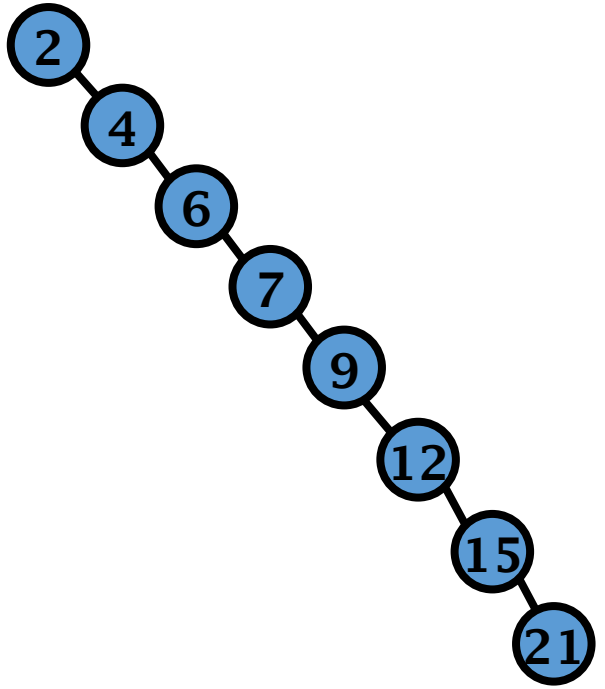
- 请调研 Splay 树的各种应用
- 红黑树、AVL 树和 Splay 树的比较
  - 它们与访问频率的关系？
  - 树形结构与输入数据的顺序关系？
  - 统计意义上哪种数据结构的性能更好？
  - 哪种数据结构最容易编写？

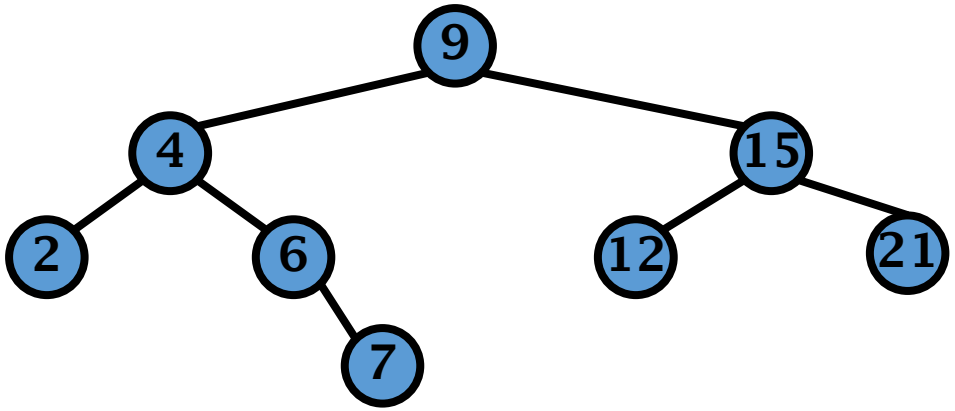
第十二章 高级数据结构

- 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
    - AVL 树的概念和插入操作
    - AVL 树的删除操作和性能分析
  - 12.5.2 伸展树

12.5.1 平衡的二叉搜索树 (AVL)

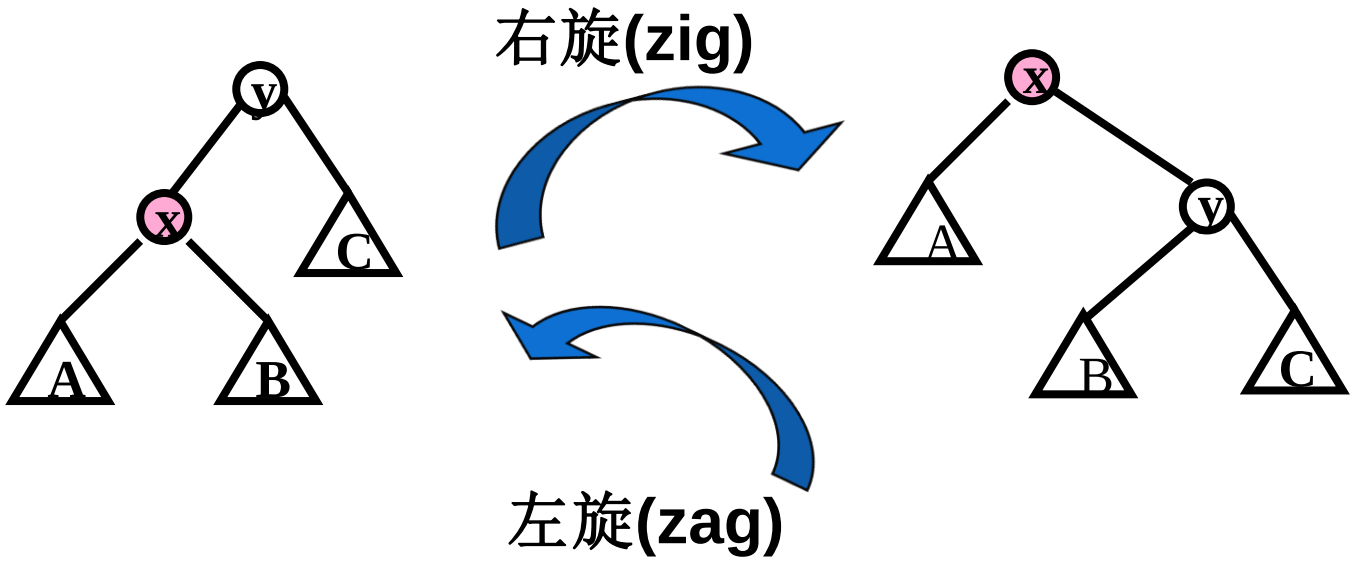
- BST受输入顺序影响
  - 最好O (log n)
  - 最坏O (n)
- Adelson-Velskii 和 Landis
  - AVL 树，平衡的二叉搜索树
  - 始终保持O (log n) 量级





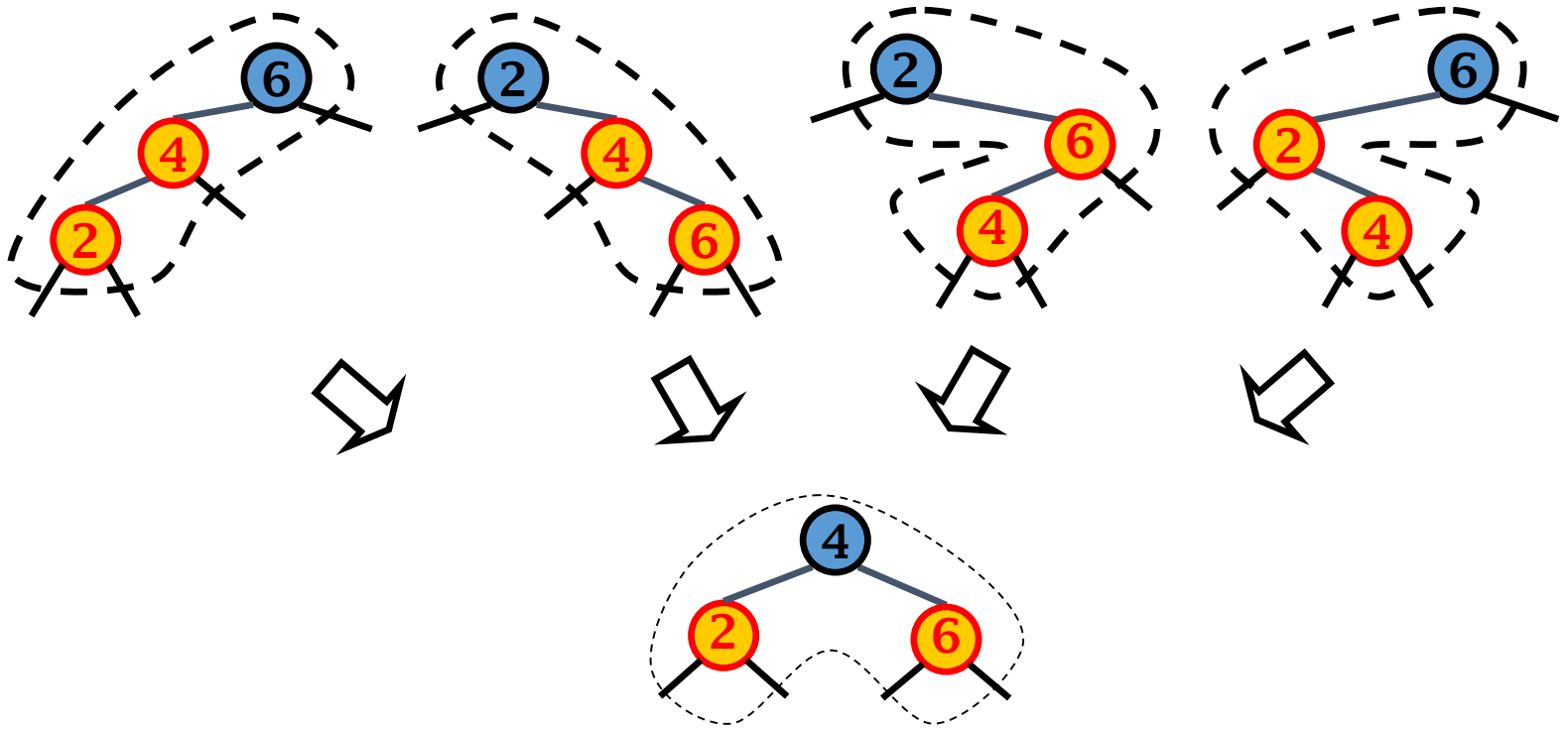
12.5.1 平衡的二叉搜索树 (AVL)

- **单旋转**
  - 结点与它的父结点交换位置，保持 BST 特性



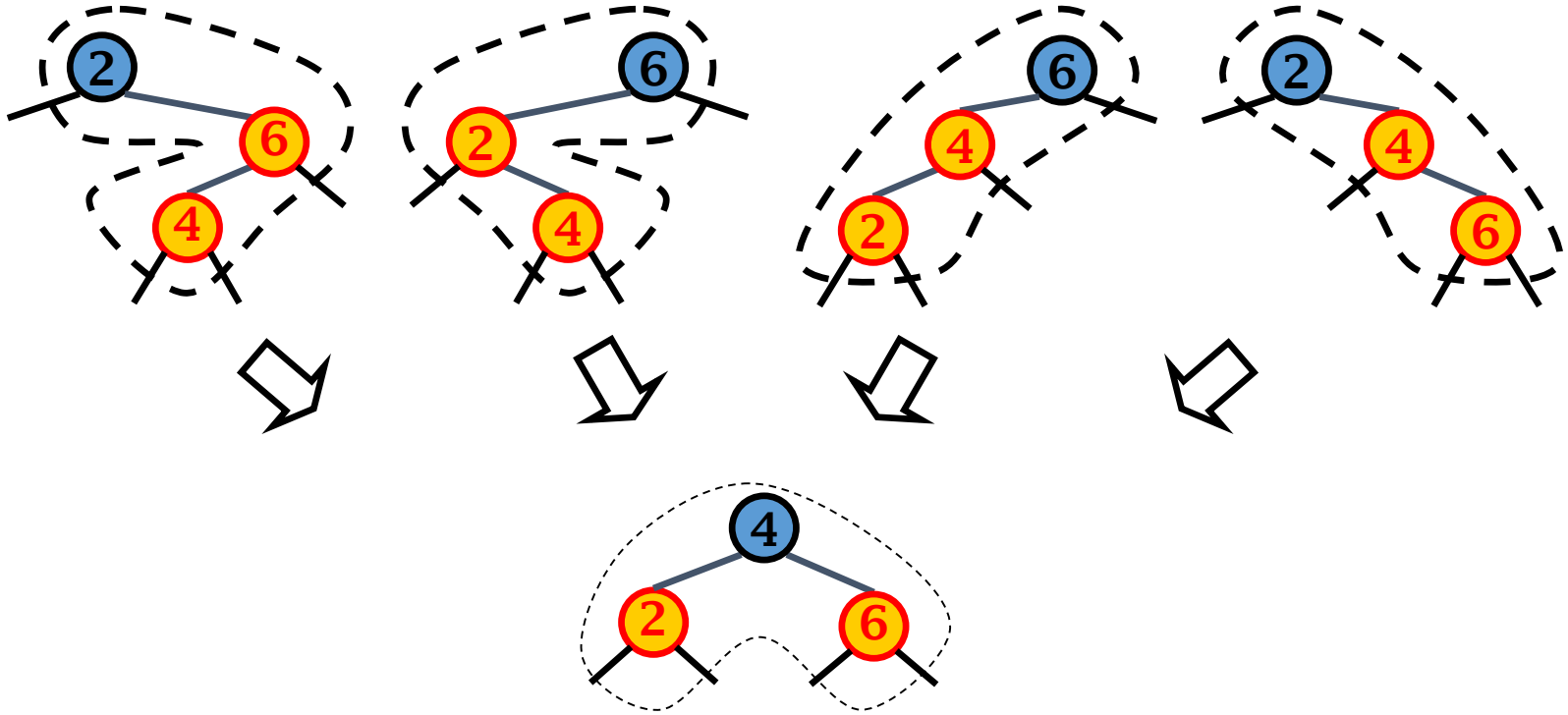
12.5.1 平衡的二叉搜索树 (AVL)

- 单旋和双旋：保持 BST 的中序性质



12.5.1 平衡的二叉搜索树 (AVL)

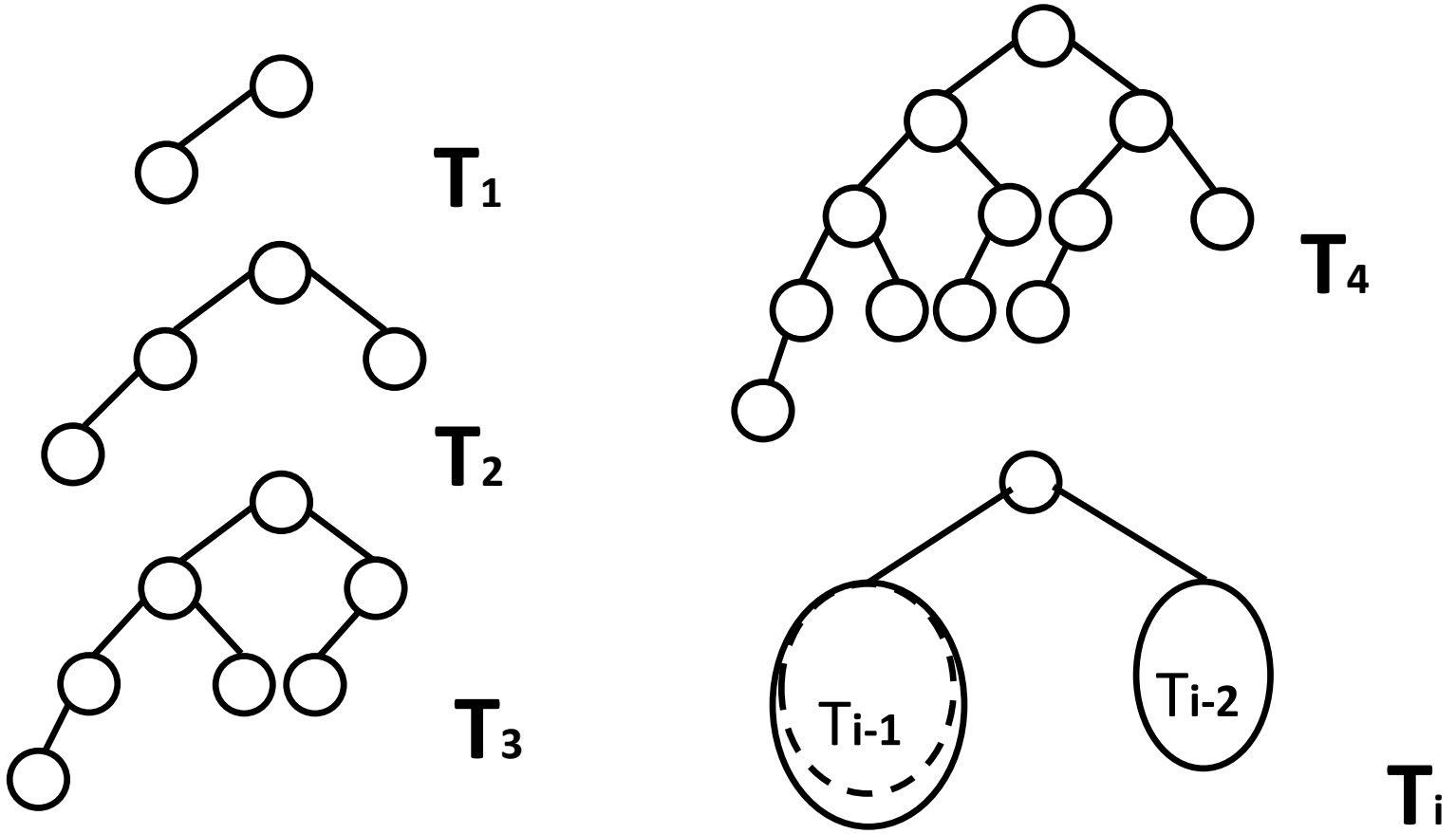
- 等价的旋转：保持 BST 的中序性质



AVL 树的性质

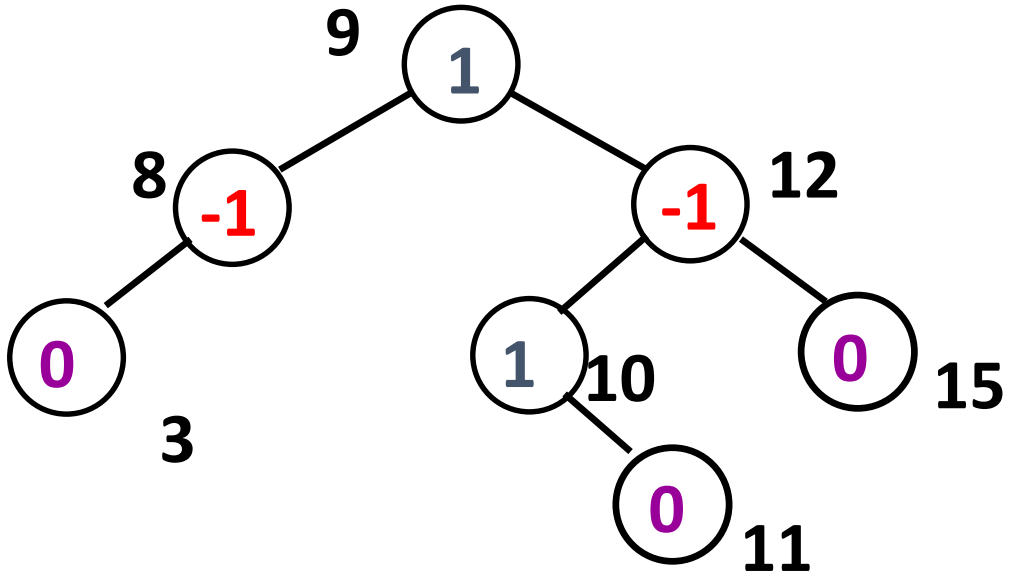
- 可以为空
- 具有 n 个结点的 AVL 树，高度为 O (log n)
- 如果 T 是一棵 AVL 树
  - 那么它的左右子树 T_L、T_R 也是 AVL 树
  - 并且 $|h_L-h_R|\leq 1$
    - h_L、h_R 是它的左右子树的高度

AVL 树举例



平衡因子

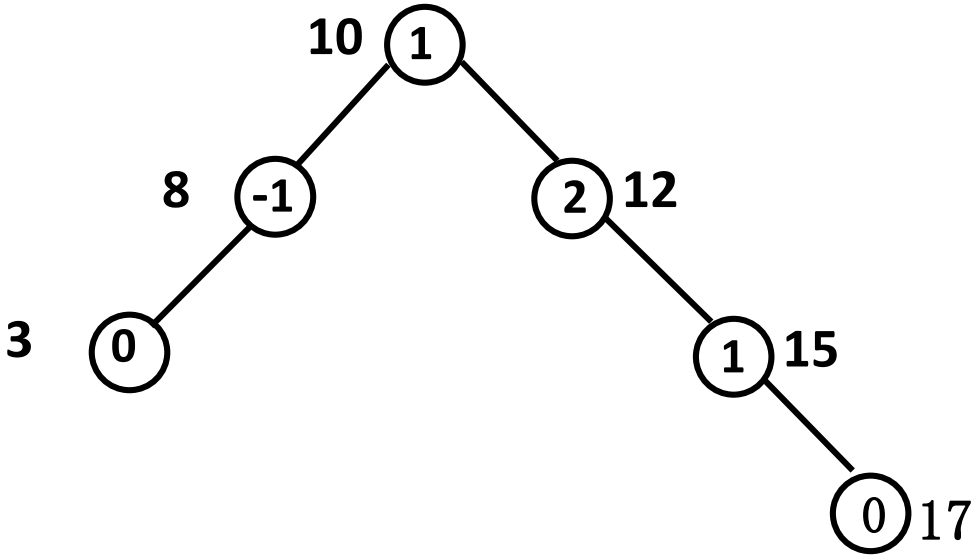
- 平衡因子，bf (x)：
  - Bf (x) = $height(x_{rchild}) - height(x_{lchild})$
- 结点平衡因子可能取值为 0，1 和 -1



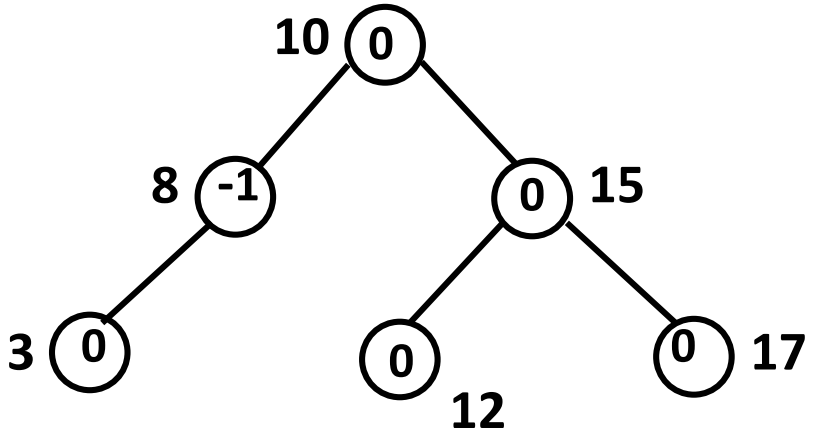
AVL 树结点的插入

- 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

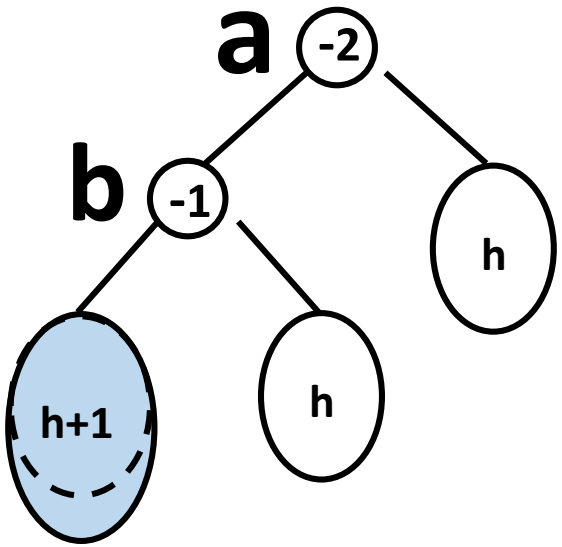
- 假设 a 离新结点 s 最近，且平衡因子绝对值不等于0
  - s (关键码为key) 要么在 a 的左子树，要么在其右子树中

- 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

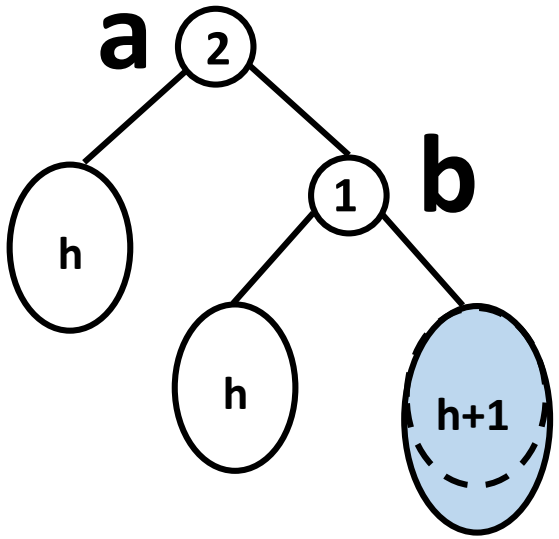
不平衡的情况

- 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

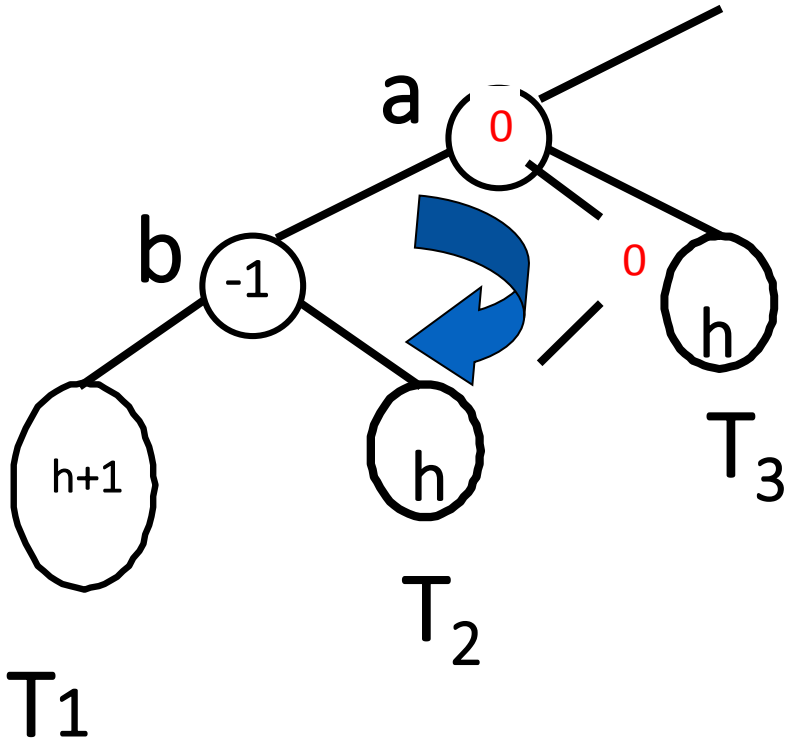


RR型

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



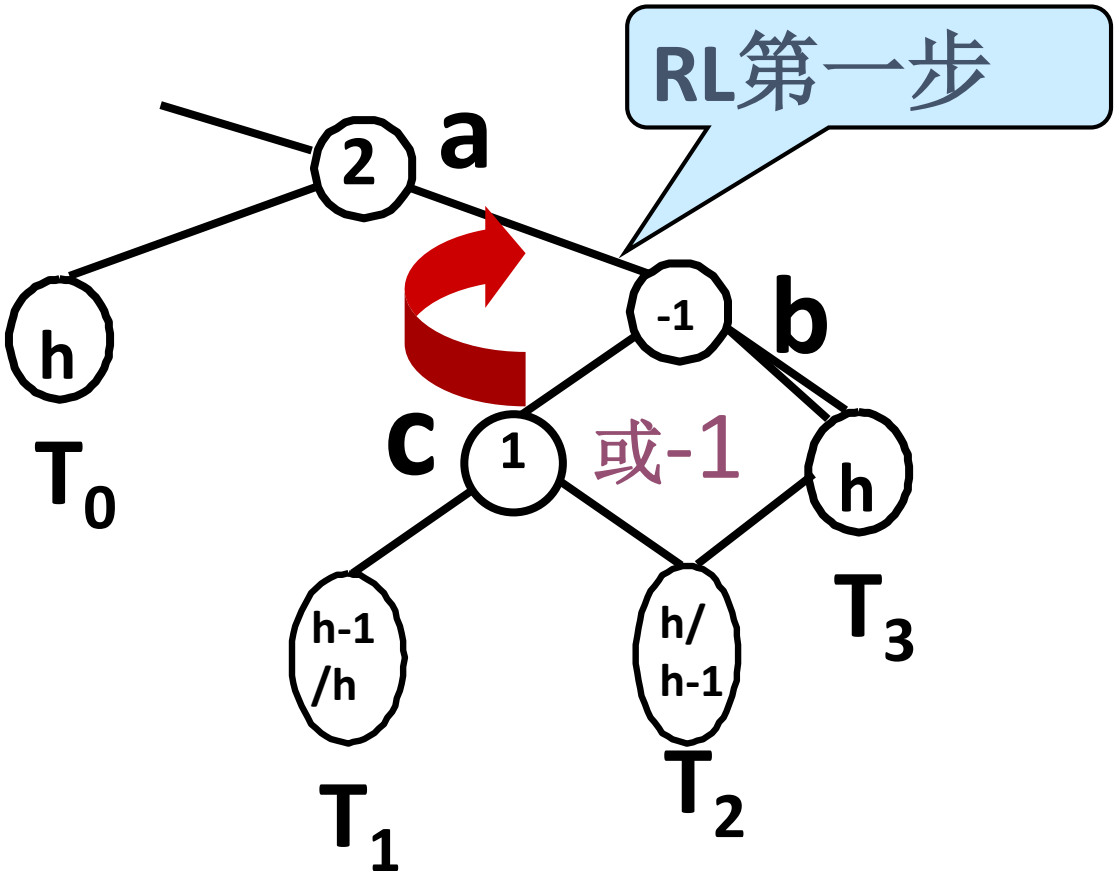
旋转运算的实质

- 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



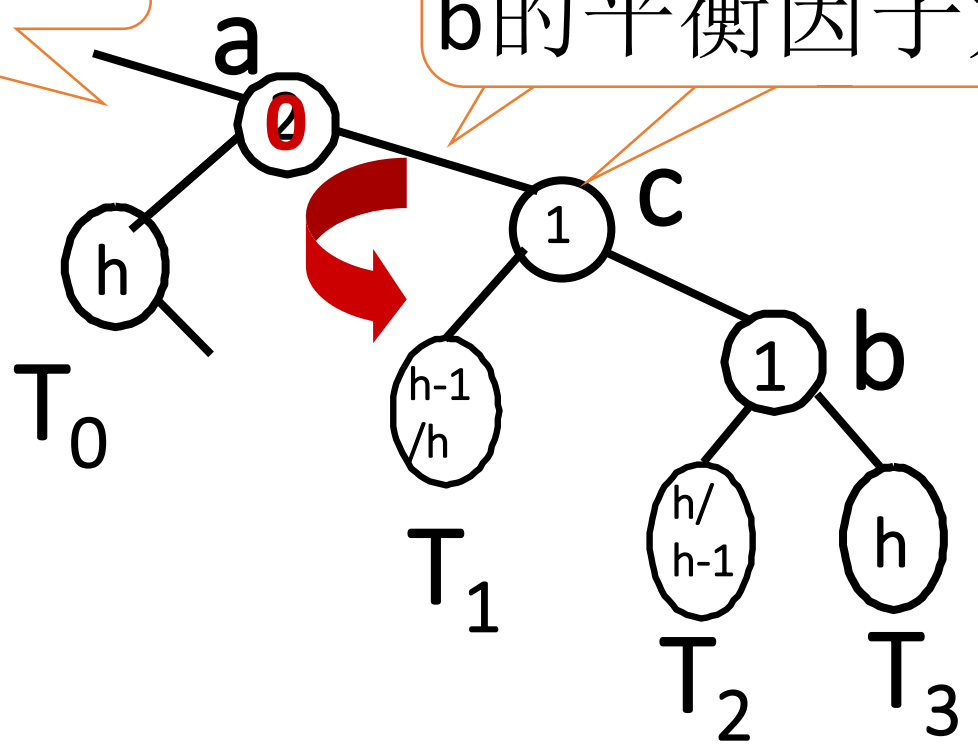
插入前
a子树高h+2

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

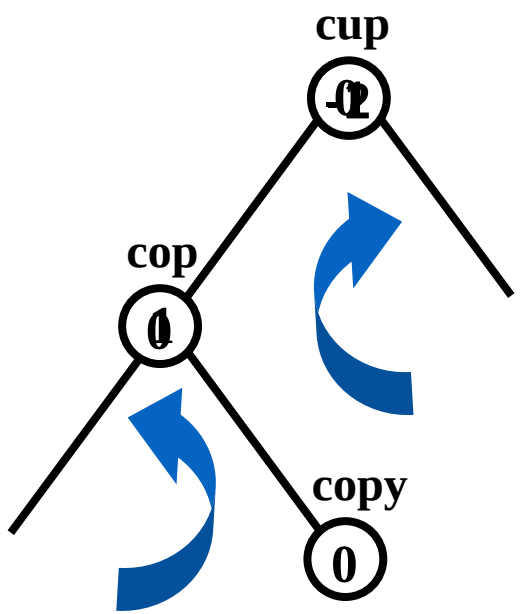


旋转运算的实质 (续)

- 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

12.5 改进的二叉搜索树

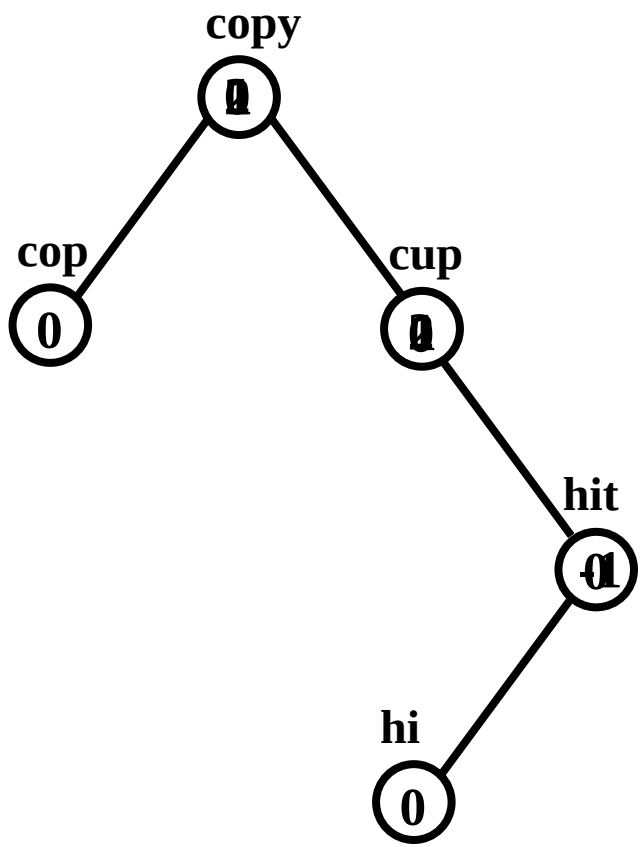
插入单词：cup ，cop， copy， hit， hi， his 和 hia 后得到的 AVL 树



插入 copy 后不平衡
LR 双旋转

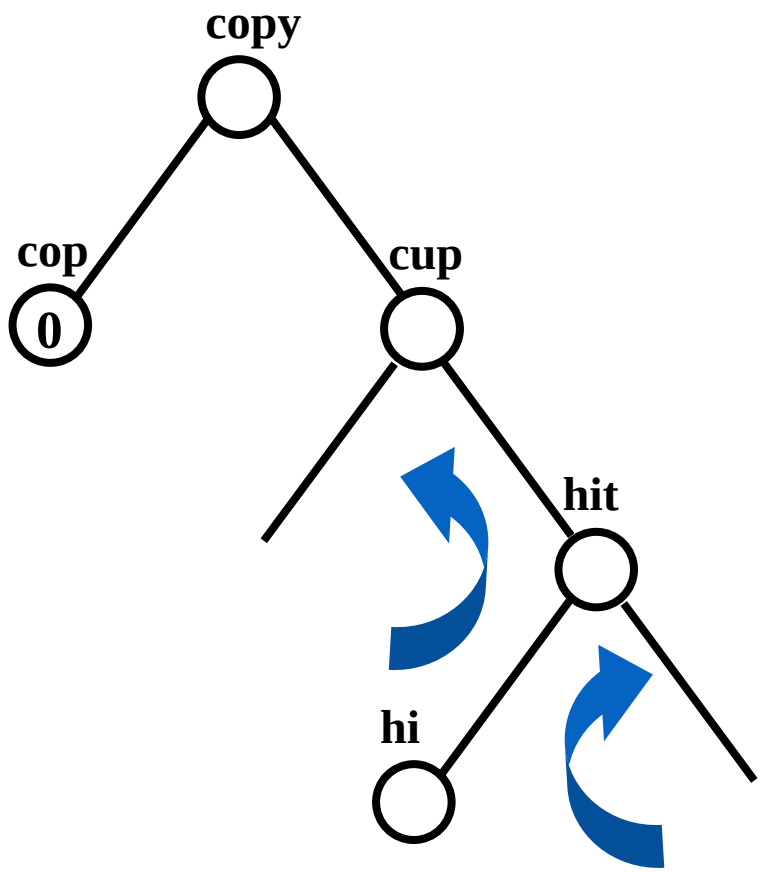
12.5 改进的二叉搜索树

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



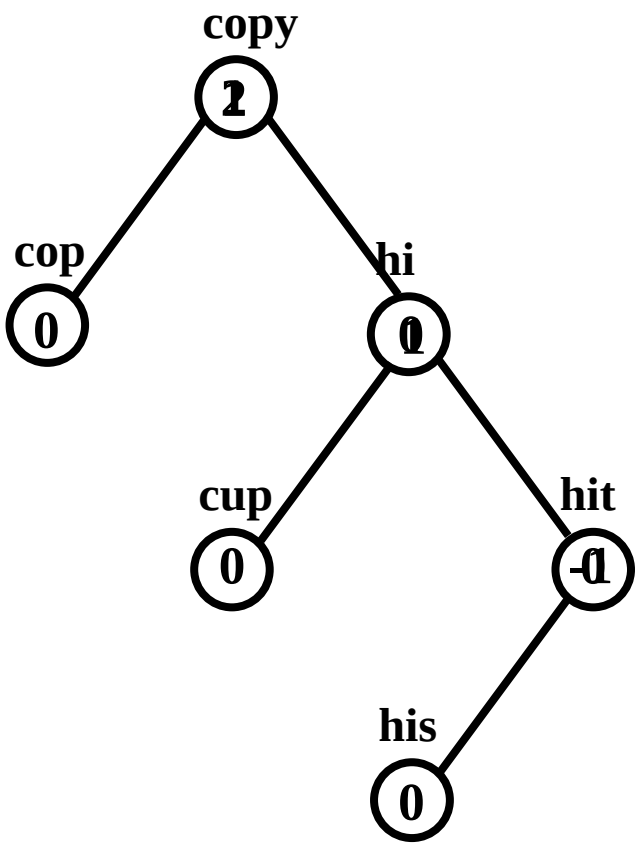
插入单词：cup ，cop， copy， hit， hi， his 和 hia 后得到的 AVL 树

RL 双旋转



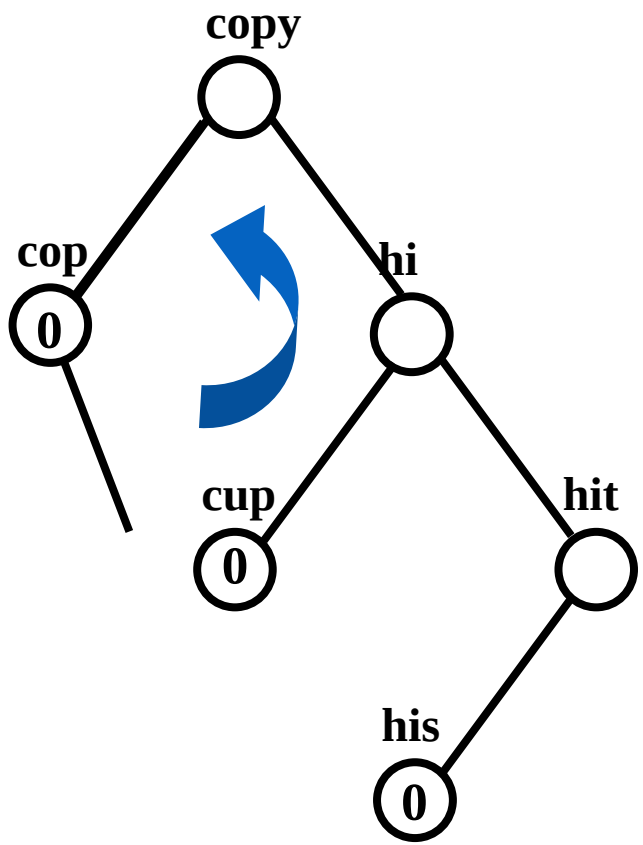
12.5 改进的二叉搜索树

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



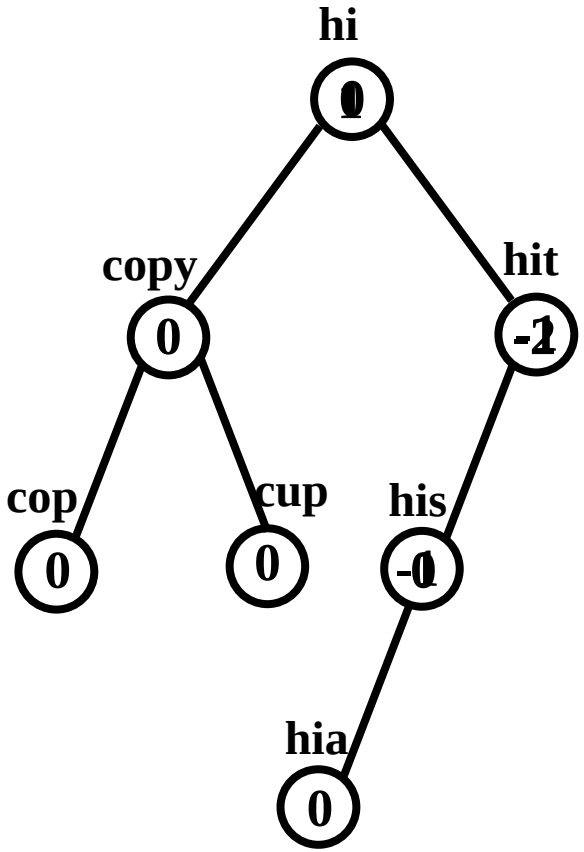
插入单词：cup ，cop， copy， hit， hi， his 和 hia 后得到的 AVL 树

RR单旋转



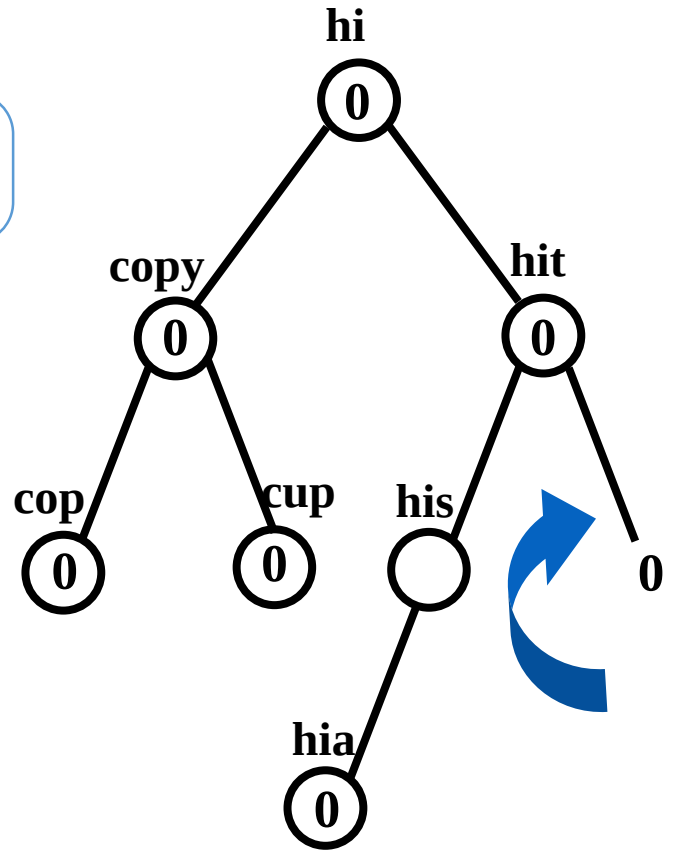
12.5 改进的二叉搜索树

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

LL单旋转



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？
- 将关键码1，2，3，…，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。

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
    - AVL 树的概念和插入操作
    - AVL 树的删除操作和性能分析
  - 12.5.2 伸展树

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

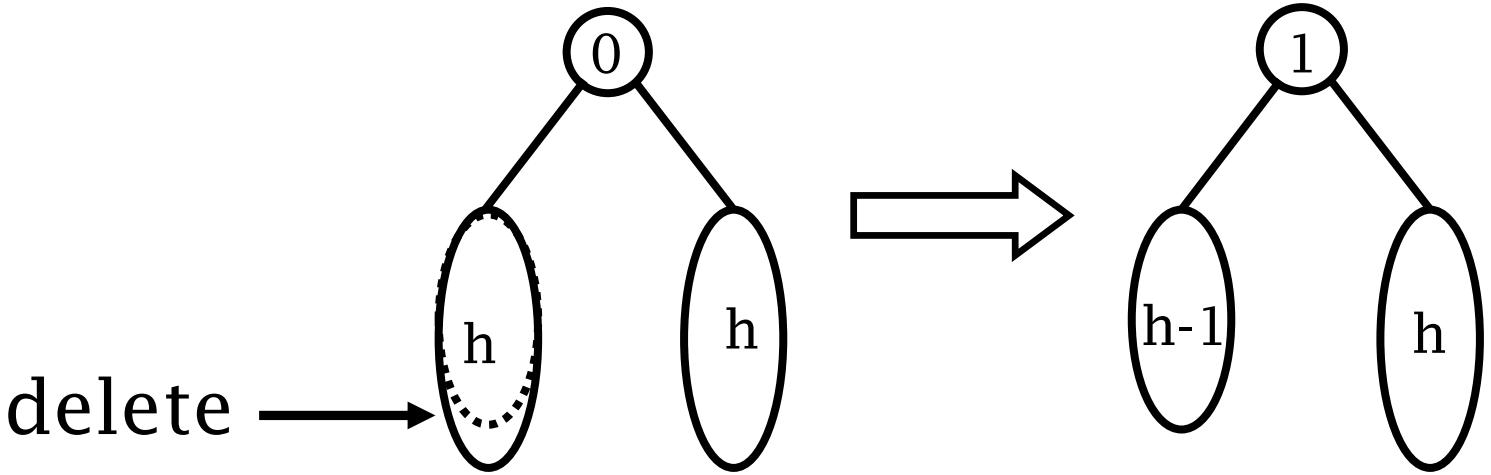
AVL结点删除后的调整

- 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

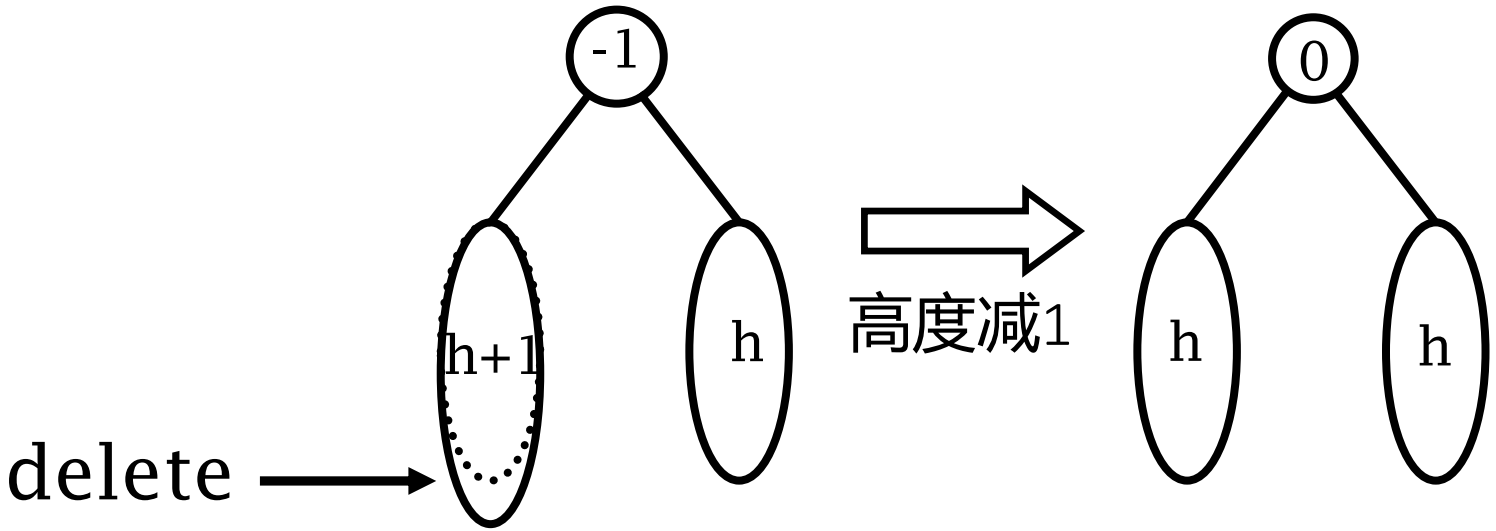
AVL 树结点的删除情况1

- 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



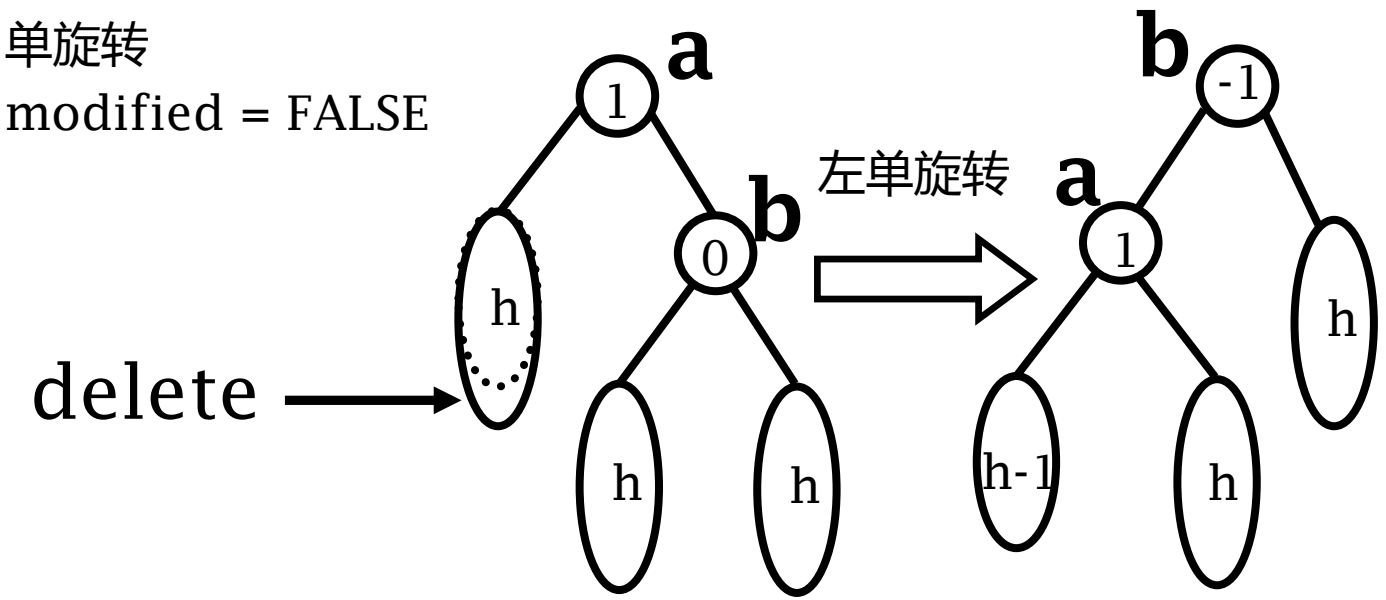
AVL 树结点的删除情况 2

- 当前结点 a 平衡因子不为 0，而较高的子树被缩短
  - 则其平衡因子修改为 0
  - modified = TRUE
  - 需要继续向上修改



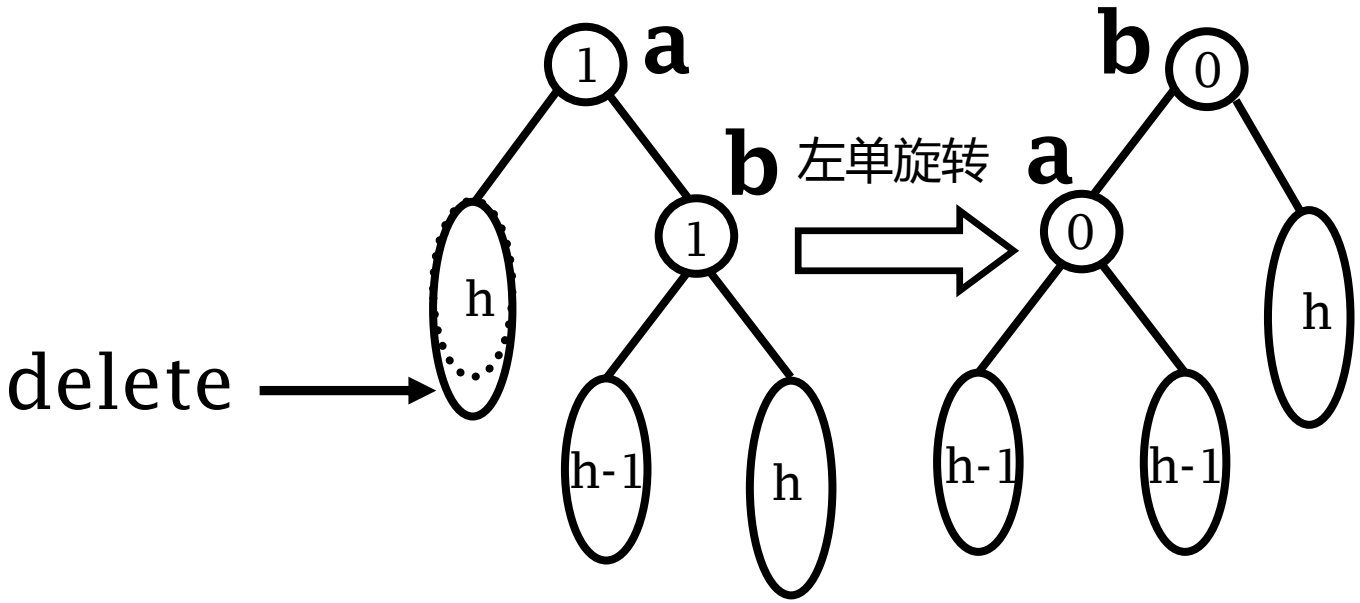
AVL 树结点的删除情况 3.1

- 当前结点 a 平衡因子不为 0，且它的较矮的子树被缩短，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



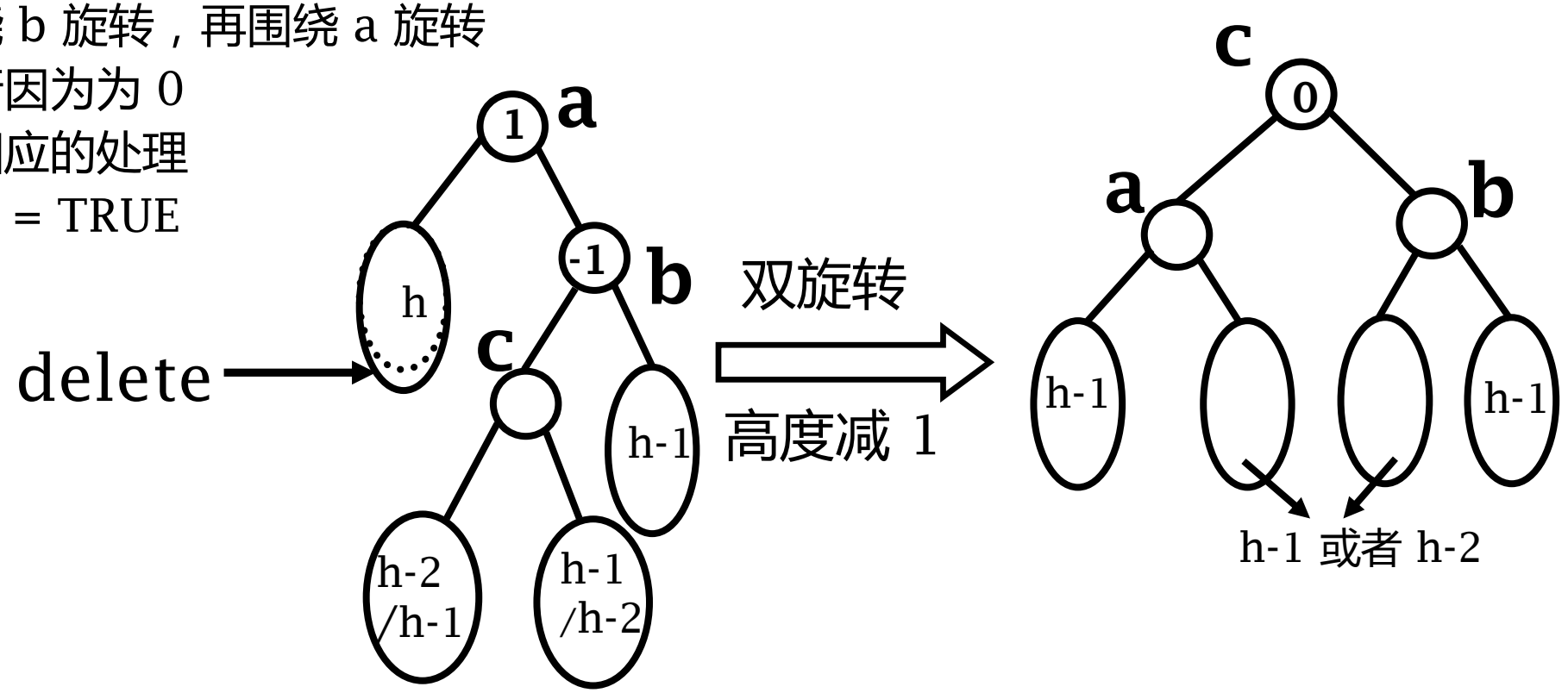
AVL 树结点的删除过程 3.2

- 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 树结点的删除情况 3.3

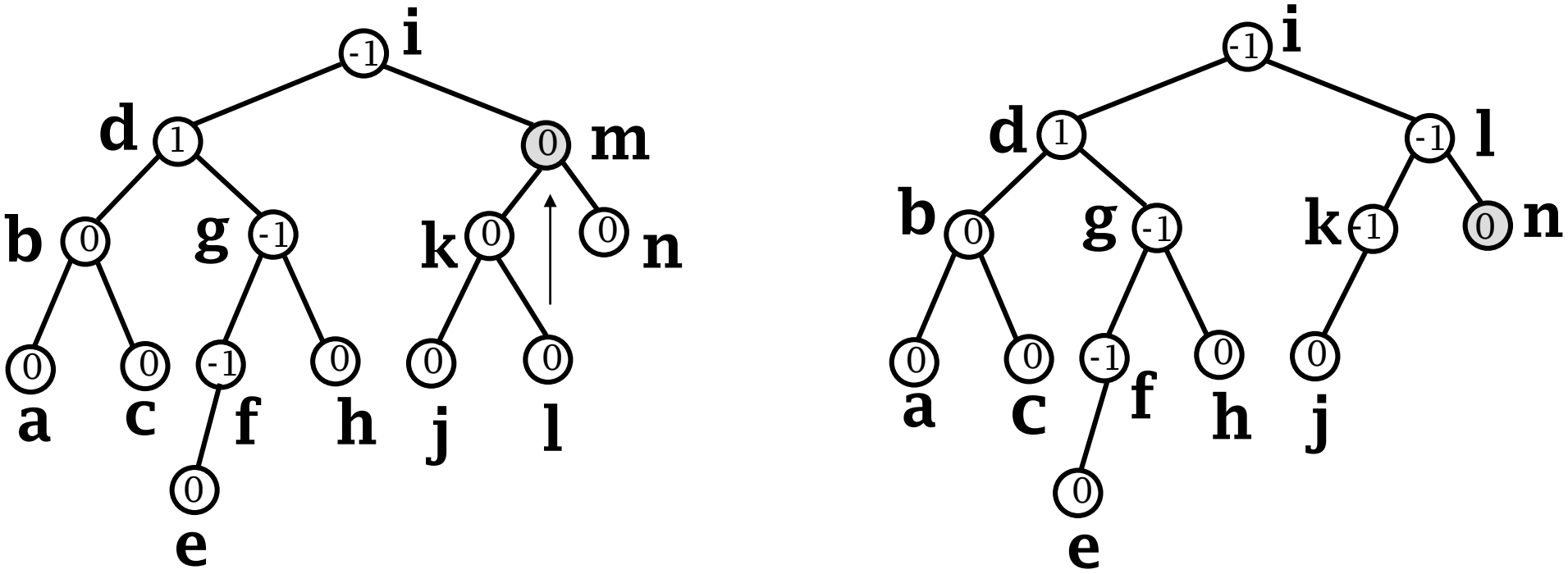
- 情况3.3：b 和 a 的平衡因子相反
  - 双旋转，先围绕 b 旋转，再围绕 a 旋转
  - 新的根结点平衡因为为 0
  - 其他结点应做相应的处理
  - 并且 modified = TRUE



删除后的连续调整

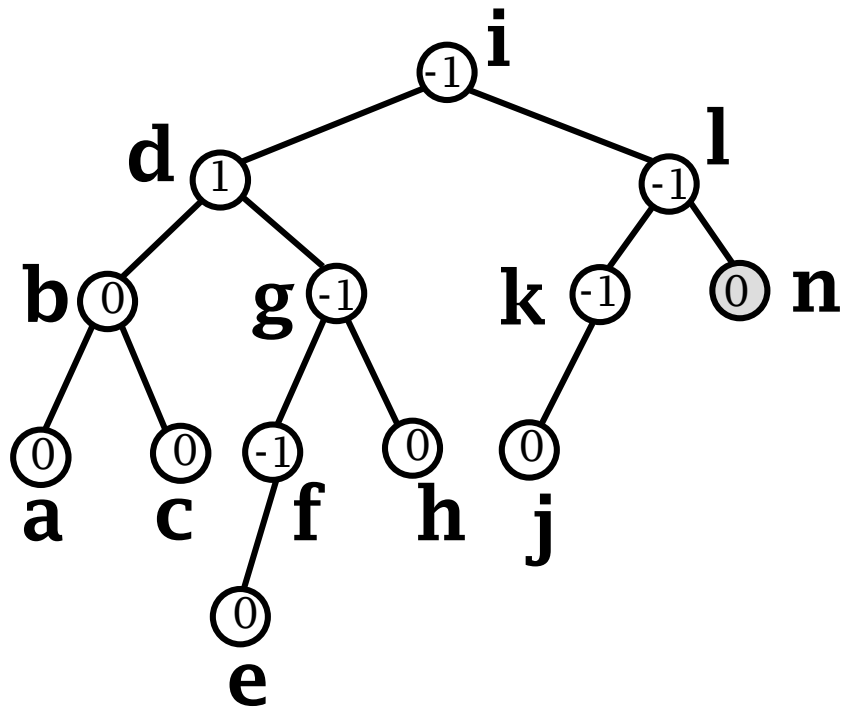
- 连续调整
  - 调整可能导致祖先结点发生新的不平衡
  - 这样的调整操作要一直进行下去，可能传递到根结点为止
- 从被删除的结点向上查找到其祖父结点
  - 然后开始单旋转或者双旋转操作
  - 旋转次数为 O (log n)

AVL 树删除的例子

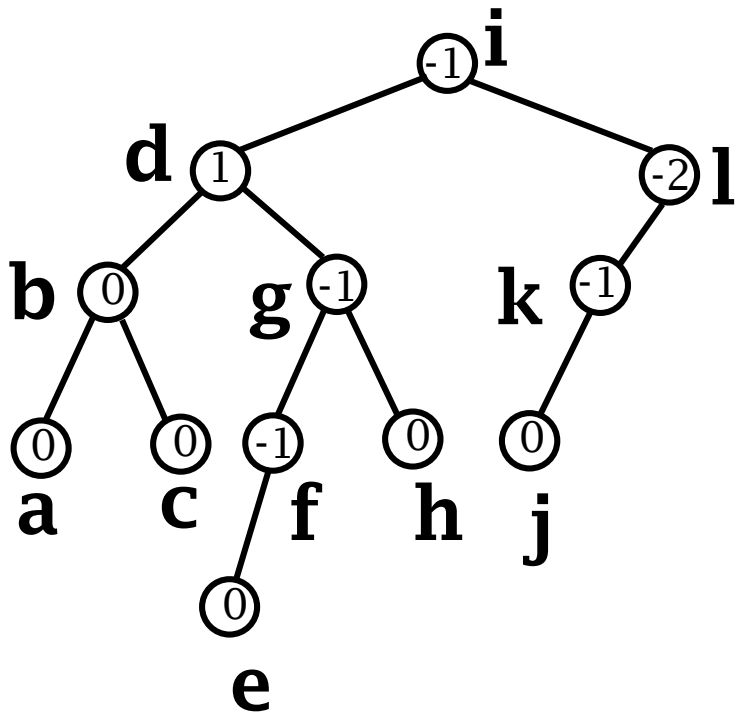


(a) 删除结点 m，则需要使用其中序前驱l代替（情况1）

AVL 树删除的例子

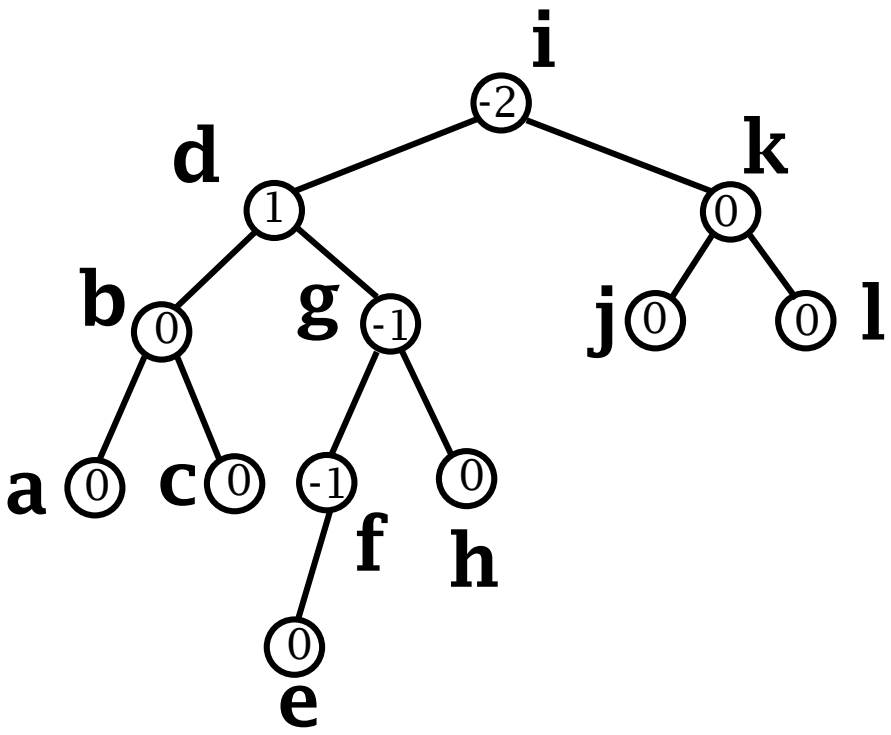


(b) 删除结点 n (情况 3.2)

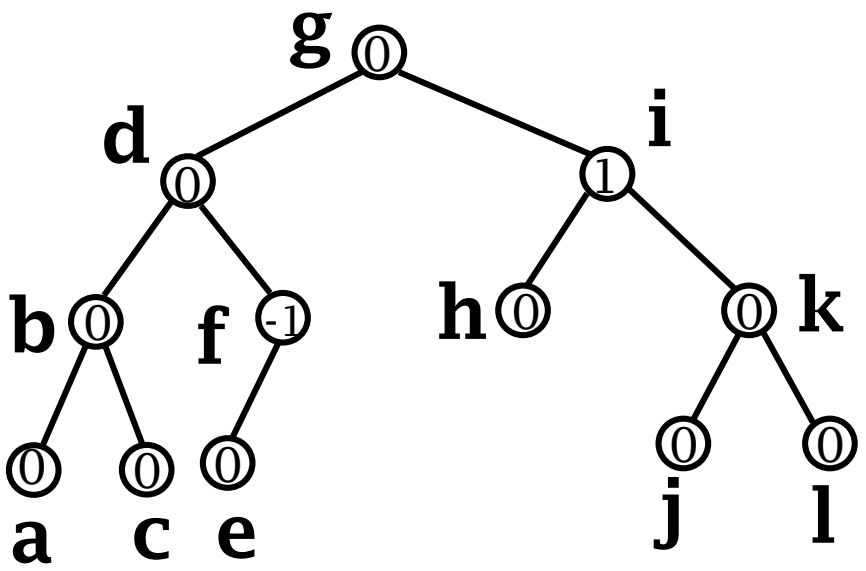


(c) 需要以 l 为根进行 LL 单旋转 (情况 3.2)

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 LR 双旋转（情况3.3）



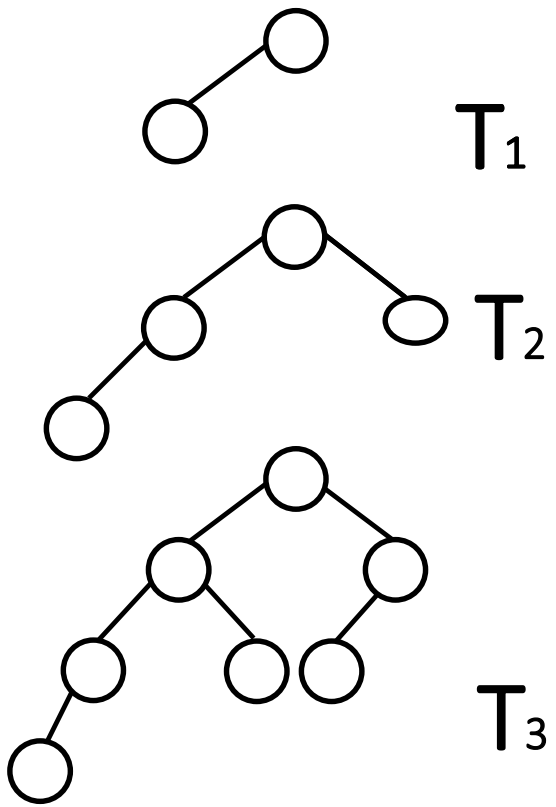
(e) 调整完毕，AVL 树重新平衡

AVL 树的高度

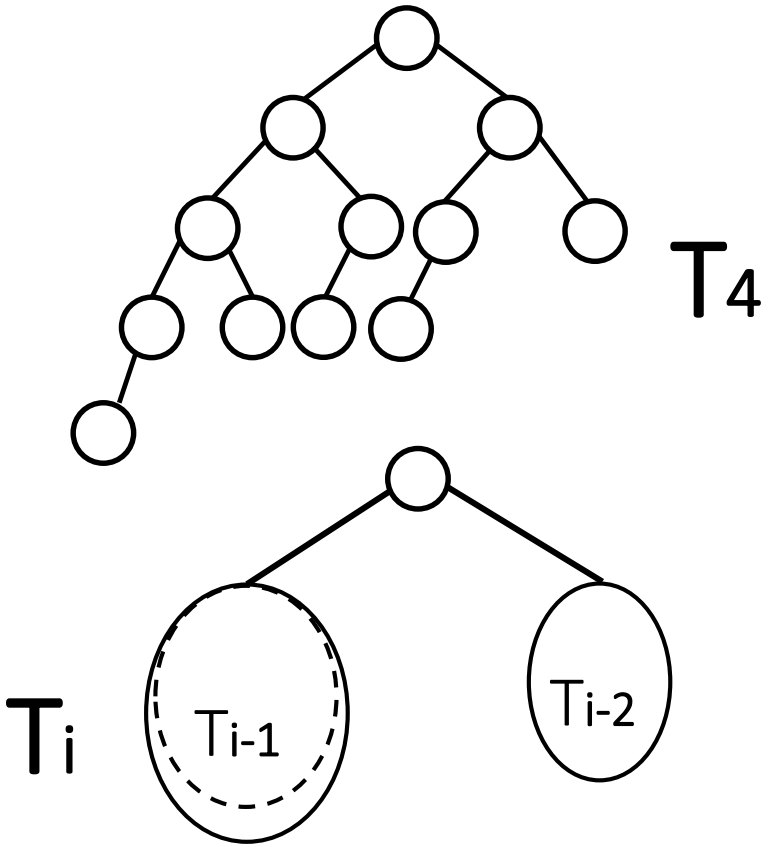
- 具有 n 个结点的 AVL 树高度一定是 O (log n)
- n 个结点的 AVL 树的最大高度不超过 $K\log_2 n$
  - 这里 K 是一个小的常数
- 最接近于不平衡的 AVL 树
  - 构造一系列 AVL 树 T_1，T_2，T_3，…。

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 T_i 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
  - t (i) = n

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

  - 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

- 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