第12章 高级数据结构

宋国杰

北京大学信息科学技术学院

内容

- 多维数组
- 广义表
- Trie结构
- 改进的二叉搜索树

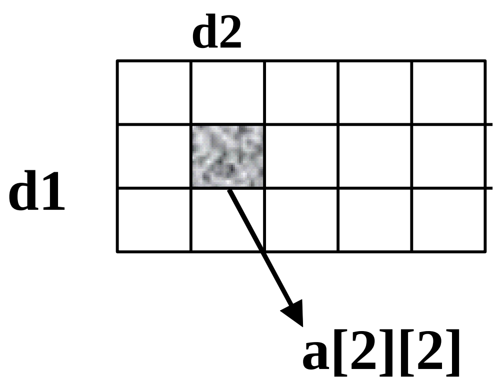
基本概念

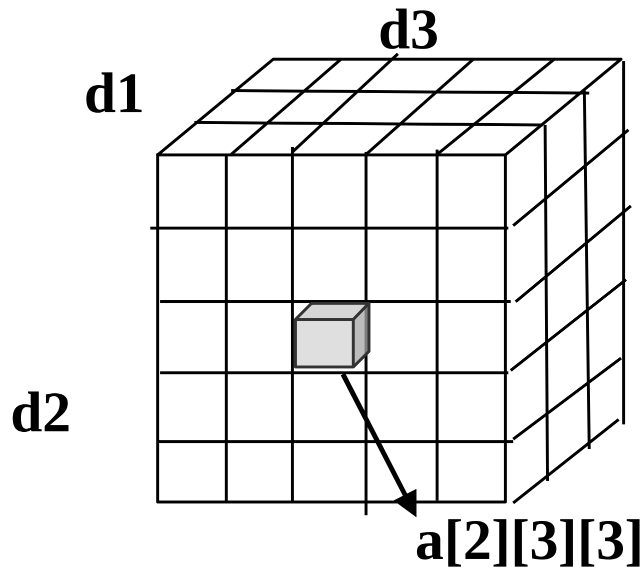
- 数组（Array）是数量和元素类型固定的有序序列
  - 静态数组必须在定义它的时候指定其大小和类型
  - 动态数组可以在程序运行时才分配空间
- 多维数组（Multi-array）是向量的扩充
  - 向量的向量就组成了多维数组
  - 可以表示为：**ELEM A[c_1..d_1][c_2..d_2]…[c_n..d_n]**
    - ✓ c_i和d_i是各维下标的下界和上界。
    - ✓ 所以，其元素个数为：

$$\prod_{i=1}^{n}(d_i - c_i + 1)$$

数组的空间结构

d1=3,d2=5,d3=5





二维数组　　　　三维数组

d1[1..3], d2[1..5], d3[1..5]分别为3个维

数组的存储

- 内存是一维的，所以数组的存储也只能是一维的
- **行优先顺序**——将数组元素按行排列
  - 在PASCAL、C语言中，数组就是按行优先顺序存储的。
- **列优先顺序**——将数组元素按列向量排列
  - 在FORTRAN语言中，数组就是按列优先顺序存储的。
- 推广到多维数组的情况：
  - 行优先顺序：先排最右下标，从右到左，最后排最左下标
  - 列优先顺序：先排最左下标，从左向右，最后排最右下标

行优先存储(Pascal)

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	…	a_{1n}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	…	a_{2n}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	…	a_{3n}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	…	a_{4n}
…	…	…	…	…	…	…
a_{m1}	a_{m2}	a_{m3}	a_{m4}	a_{m5}	…	a_{mn}

列优先存储(FORTRAN)

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	…	a_{1n}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	…	a_{2n}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	…	a_{3n}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	…	a_{4n}
…	…	…	…	…	…	…
a_{m1}	a_{m2}	a_{m3}	a_{m4}	a_{m5}	…	a_{mn}

下标地址计算基本原理

$A_{i1\ldots in}$的起始地址

$=$ 第一个元素的起始地址 $+$ 该元素前面的元素个数 $\times$ 单位长度

- 假设数组各维的下界是0，按“行优先顺序”存储，假设每个元素占用d个存储单元。
- 二维数组A_{mn}， a_{ij}的地址计算函数为：

$$LOC(a_{ij})=LOC(a_{00})+[\, i*n + j \,]*d$$

- 三维数组A_{mnp}，a_{ijk}的地址计算函数为：

$$LOC(a_{ijk})=LOC(a_{000})+[\, \underline{i*n*p} + \underline{j*p} + \underline{k} \,] * d$$

C语言下标地址的计算

- **C++多维数组ELEM A[d_0][d_1]…[d_{n-1}];**

$$
\begin{aligned}
loc(A[j_0,\ldots,j_{n-1}]) &= loc(A[0,\ldots,0]) \\
&+ d*[j_0*d_1*\ldots*d_{n-1} \\
&\qquad + j_1*d_2*\ldots*d_{n-1}+\ldots \\
&\qquad + j_{n-2}*d_{n-1} \\
&\qquad + j_{n-1}] \\
&= loc(A[0,\ldots,0]) + d*[\sum_{i=0}^{n-2} j_i \prod_{k=i+1}^{n-1} d_k + j_{n-1}]
\end{aligned}
$$

特殊矩阵

- 矩阵描述为二维数组，其元素可以随机访问
- 特殊矩阵
  - 非零元素呈某种规律分布或者矩阵中有大量的零元素
  - 仍用数组存放，会造成极大浪费，尤其是高阶矩阵时
- 为节省空间，可对这类矩阵进行压缩存储

对称矩阵

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

在一个n阶方阵A中，若元素满足下述性质：$a_{ij}=a_{ji}$ $0 \leqq i,j \leqq n-1$，则称A为对称矩阵。

特征：元素关于主对角线对称

压缩存储方法：只存矩阵中上三角或下三角中的元素。

所需空间：

$$N=\sum_{i=0}^{n-1}(i+1)=\frac{n(n+1)}{2}$$

三角矩阵

1	2	3	4	5
0	3	4	5	6
0	0	5	6	7
0	0	0	7	8
0	0	0	0	9

1	0	0	0	0
2	3	0	0	0
3	6	5	0	0
4	7	9	7	0
5	8	1	2	9

1	2	3	4	5
4	3	4	5	6
4	4	5	6	7
4	4	4	7	8
4	4	4	4	9

1	4	4	4	4
2	3	4	4	4
3	6	5	4	4
4	7	9	7	4
5	8	1	2	9

- **分布特征**
  - 上三角矩阵中，主对角线的下三角中的元素均为常数。常为零
  - 下三角矩阵正好相反
- **压缩方法**
  - 只存上(下)三角阵中上(下)三角中的元素
  - 常数c共享一个存储空间
- **所需空间**

$$N = \frac{n(n+1)}{2} + 1$$

稀疏矩阵

1	2	0	0	5
0	3	0	0	0
0	4	0	0	0
0	0	6	0	0
0	0	0	8	0

- **分布特征**
  - 只有少量非零元素，且非零元素的分布没有规律
- **压缩方法**
  - 只存非零元素
- **所需空间**
  - 与非零元素的个数和存储方式有关

特殊矩阵的压缩存储

- **矩阵类型**
  - 对称矩阵，三角矩阵、稀疏矩阵
- **压缩思想**
  - 只存有用的元素
  - 二维数组改为用一维数组来存放

关键问题

- 如何确定一维数组的大小？ **= 所需空间**
- 如何确定矩阵元素在一维数组中的位置？从而保证对矩阵元素的随机存取

A_{ij}的位置 = **该元素前的元素个数**

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

1
2
3
3
4
5
4
5
6
7
…

1．对称矩阵

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

如何确定一维数组的大小？

$$N=\frac{n(n+1)}{2}$$

设：存放下三角阵中的元素，

则：如何确定元素Aij在一维数组中的位置？

$$Loc(A_{ij})=\begin{cases}\dfrac{i\times(i+1)}{2}+j & \text{当}\, i>=j,\ A_{ij}\text{在下三角阵中}\\[2ex] \dfrac{j\times(j+1)}{2}+i & \text{当}\, i<j,\ A_{ij}\text{在上三角阵中}\\ & (\text{根据}\,A_{ij}=A_{ji})\end{cases}$$

1
2
3
3
4
5
4
5
6
7
…

2. 三角矩阵

1	4	4	4	4
2	3	4	4	4
3	6	5	4	4
4	7	9	7	4
5	8	1	2	9

如何确定一维数组的大小？

$$N = \frac{n(n+1)}{2} + 1$$

设：在下三角阵中，

则：如何确定元素Aij在一维数组中的位置？

1
2
3
3
6
5
…
…
4

$$Loc(A_{ij}) = \begin{cases} \frac{i\times(i+1)}{2} + j, \text{当} i >= j, \text{即下三角阵中的元素} \\ \frac{n\times(n+1)}{2}, \quad \text{当} i < j, \text{即下三角阵中的常数} \end{cases}$$

稀疏矩阵的压缩存储

- 顺序存储：三元组表
- 链式存储：十字链表

1.三元组表存稀疏矩阵

1	2	0	0	5
0	3	0	0	0
0	4	0	0	0
0	0	6	0	0
0	0	0	8	0

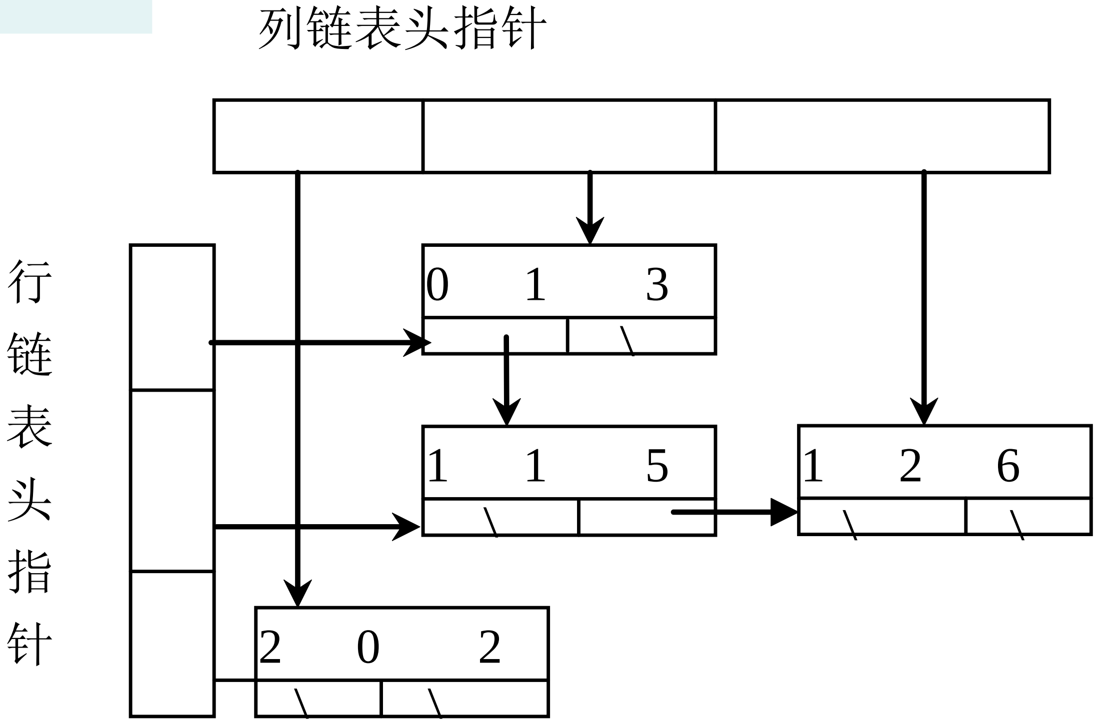
M=5
N=5
T=7

i	j	Aij
0	0	1
0	1	2
0	4	5
1	1	3
2	1	4
3	2	6
4	3	8

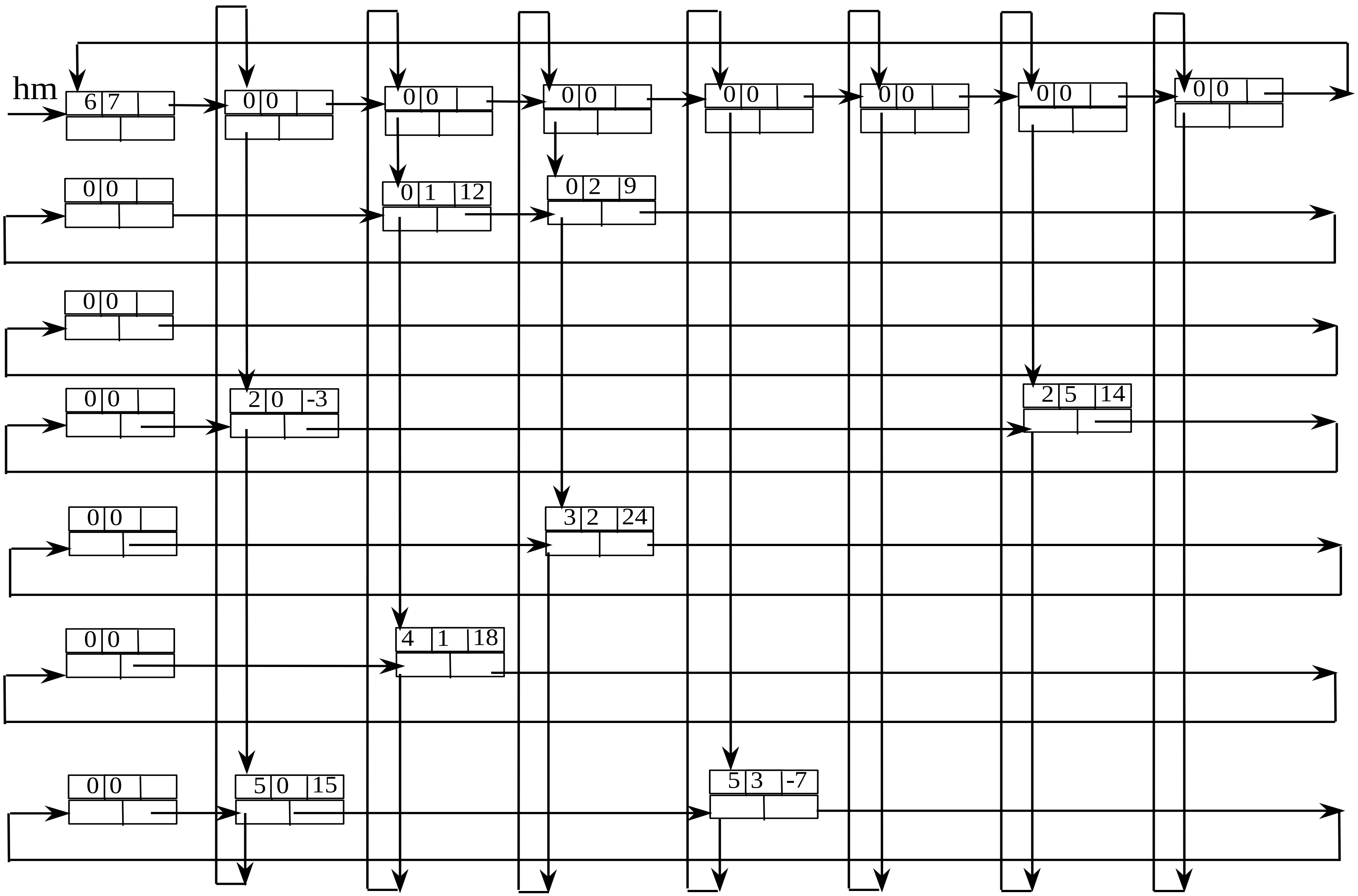
- 考虑：
  - 只存非零元素
  - 一个非零元素的必需信息有：
    (i, j, a_{ij})
  - 记录一个稀疏矩阵的必需信息有：
    行数M，列数N，非零元素个数T

2. 十字链表

i	j	Val/Next
列指针col		行指针row



稀疏矩阵的十字链表


hm
6 7
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 1 12
0 2 9
2 0 -3
2 5 14
3 2 24
4 1 18
5 0 15
5 3 -7

12.2.1 广义表

- 基本概念
- 广义表的类型
- 广义表的存储

1、基本概念

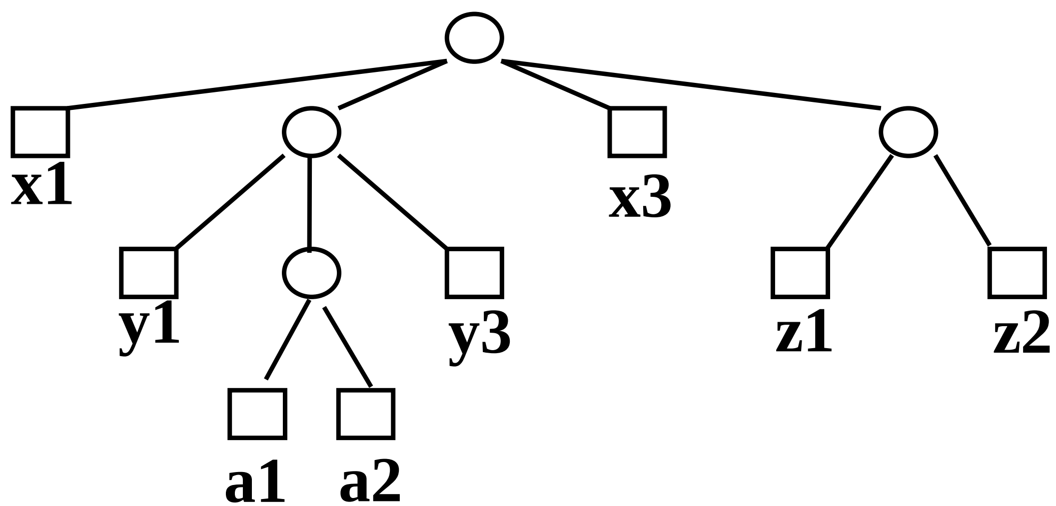
- **回顾：线性表**
  - 由n（n≥0）个数据元素组成的有限序列
  - 线性表的每个元素都具有相同的数据类型
- **广义表（Generalized Lists，也称Multi-list)**
  - 如果一个线性表中还包括一个或者多个子表
  - 一般记作：L＝（x_0，x_1，…，x_i，…，x_{n-1}）

L = （x_0，x_1，…，x_i，…，x_{n-1}）

- L是广义表的名称，n为长度
- 每个x_i（$0 \le i \le n-1$）是L的成员
  - 可以是单个元素，即原子（atom)
  - 也可以是一个广义表，即子表（sublist）
- **广义表深度**：表中元素都化解为原子后的括号层数
- 表头head = x_0，表尾tail = （x_1，…，x_{n-1}）
  - **表头是一个元素，表尾是除了表头的表**

2、广义表的类型

- 纯表（pure list）
  - 从根结点到任何叶结点只有一条路径
  - 任何元素（原子、子表）只能在广义表中出现一次

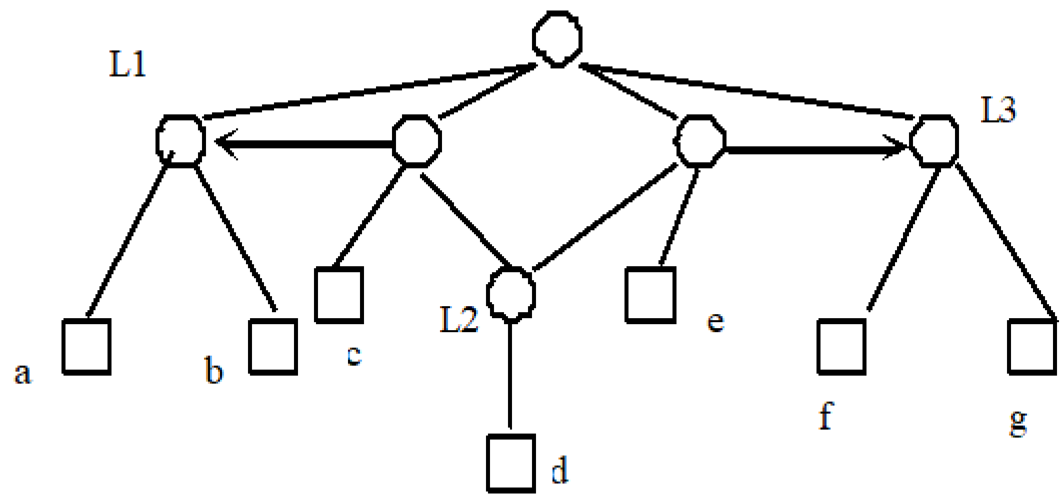


(x1, (y1 ,(a1 ,a2), y3), x3 ,(z1 ,z2))

(L1:(a, b), (L1, c, L2: (d)), (L2, e, L3: (f, g)), L3)

- **可重入表（再入表）**
  - 表中元素(包括原子和子表)可能多次出现
  - **对应一个DAG**
- 对子表和原子标号

((a, b), ((a,b),c,(d)), ((d), e, (f, g)) , (f ,g))

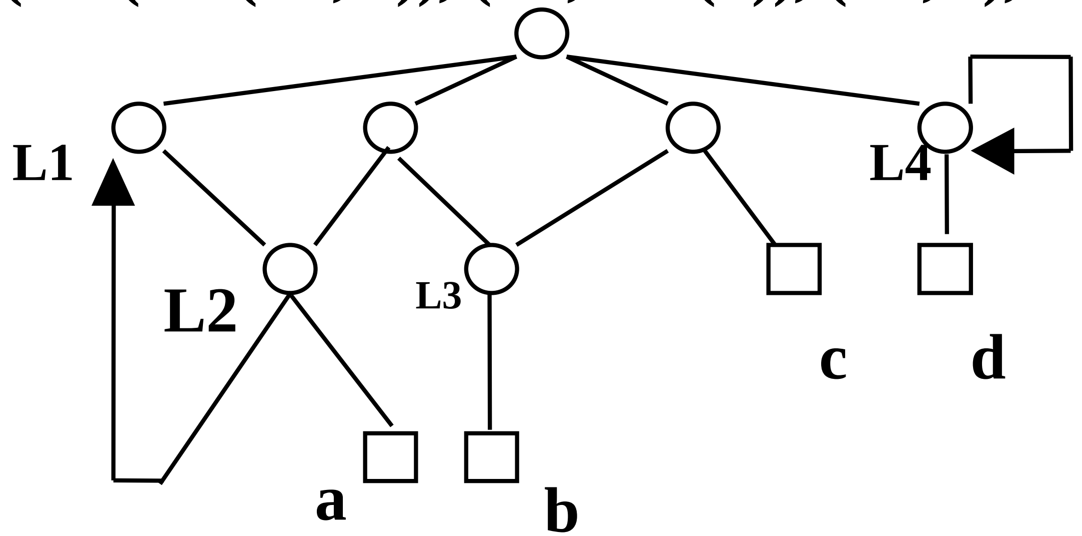


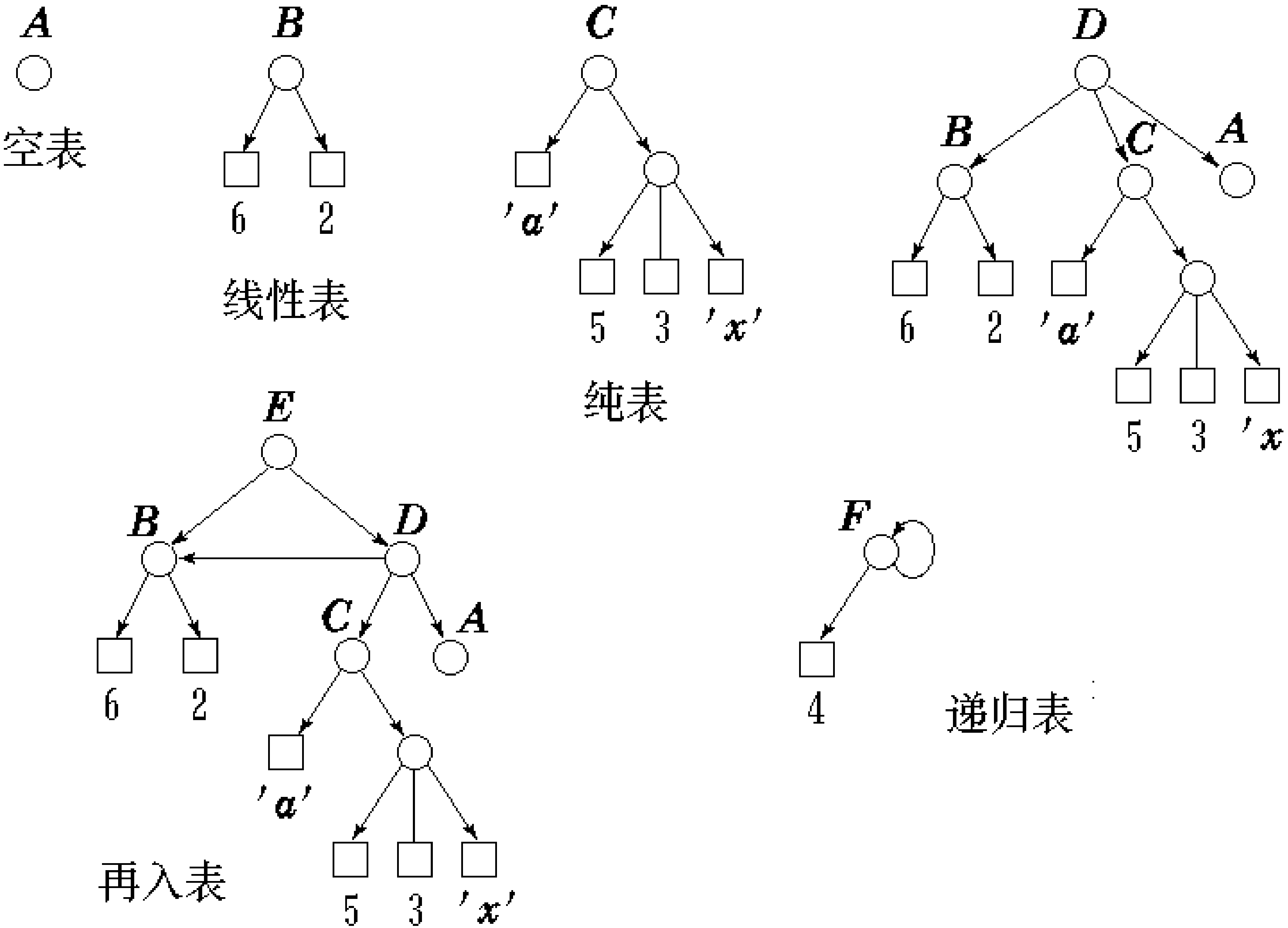
没有标志箭头的边都指向下方

循环表

- 包含回路，循环表深度为无穷大

(L1:(L2:(L1, a)), (L2, L3:(b)), (L3, c), L4:(d,L4))




A
空表
B
6 2
线性表
C
'a'
5 3 'x'
纯表
D
B C A
6 2 'a'
5 3 'x'
E
B D
C A
6 2
'a'
5 3 'x'
再入表
F
4
递归表

- 图⊇再入表⊇纯表（树）⊇线性表
  - 广义表是线性与树形结构的推广
- 递归表是有回路的再入表
- 广义表应用
  - 函数的调用关系
  - 内存空间的引用关系
  - LISP语言

3、广义表存储ADT

- typedef enum {ATOM, LIST} TAG;

  // ATOM = 0：单元素；LIST = 1：子表

- typedef struct GLNode{

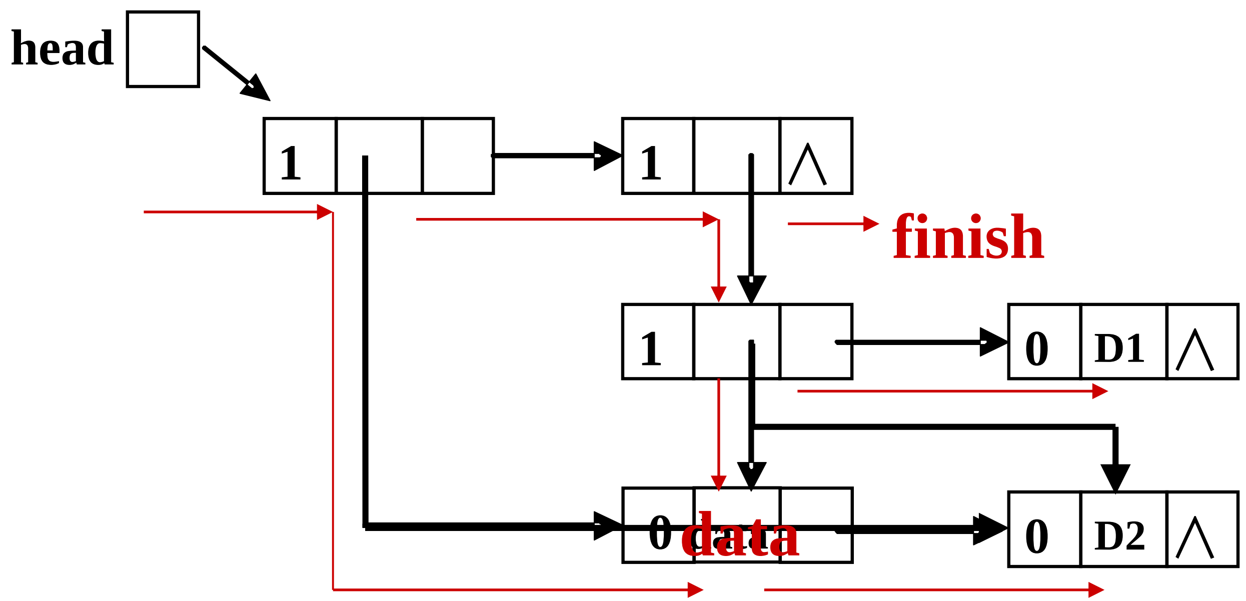
```
typedef struct GLNode{
        TAG  tag;
        union {
                ElemType data;
                GLNode *sublist;        // 子表的头指针
        };
        GLNode *next;                   // 后继结点
};
```

TAG= 0 或 1	data / sublist	next

广义表存储ADT（续）

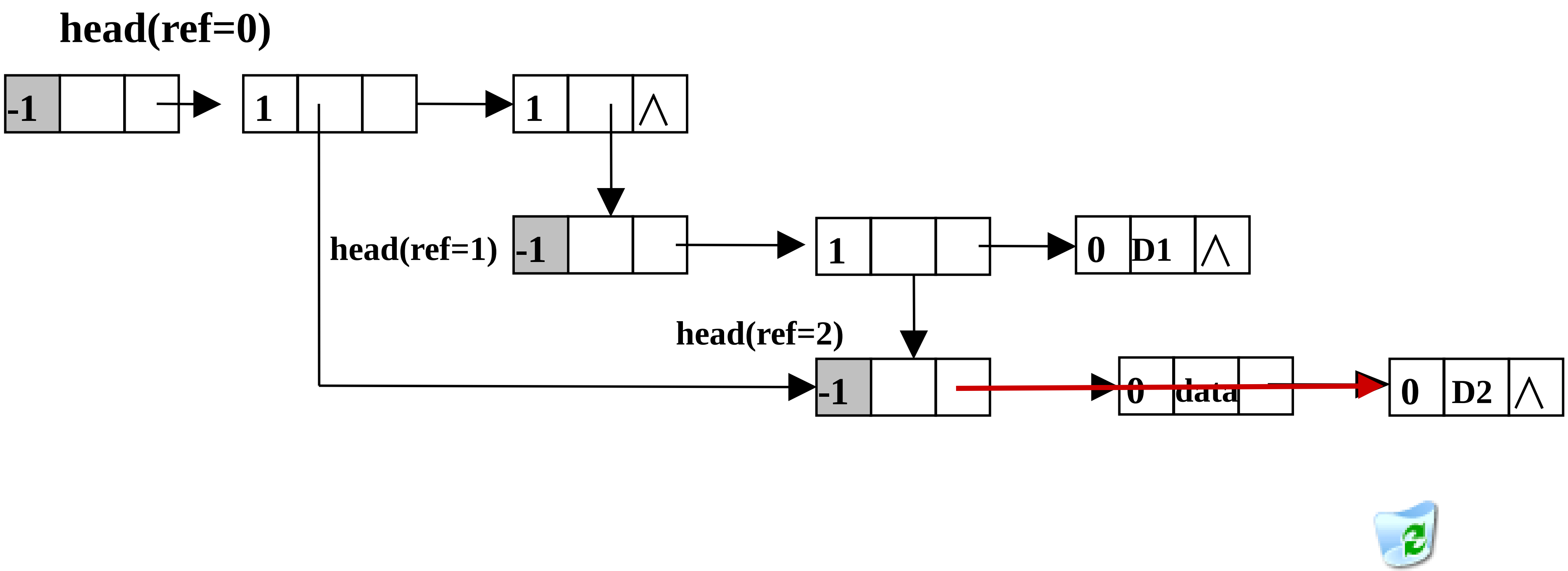
➢ 不带头结点的广义表

- 删除结点时会出现问题
- 删除结点data必须进行链调整

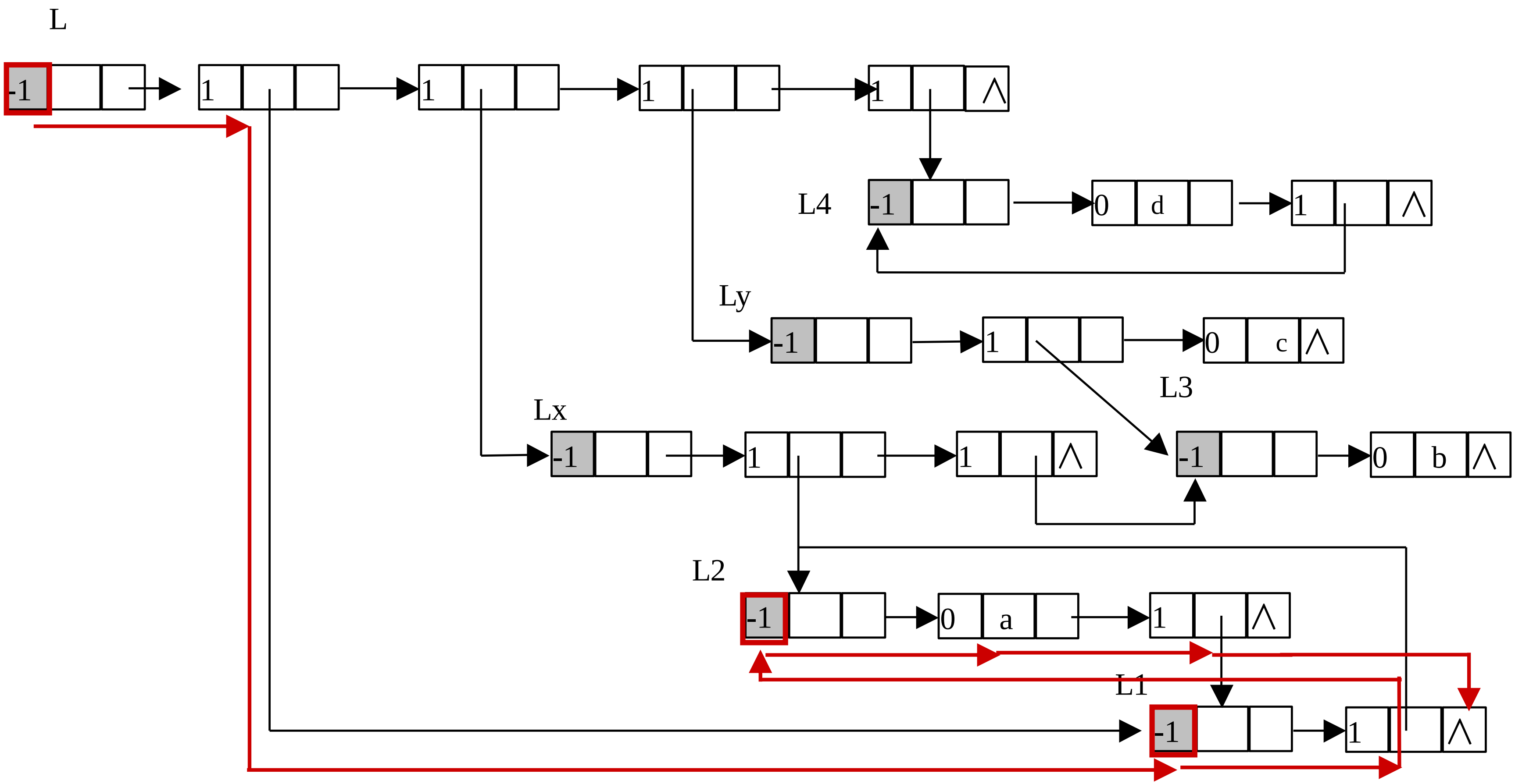


广义表存储ADT（续）

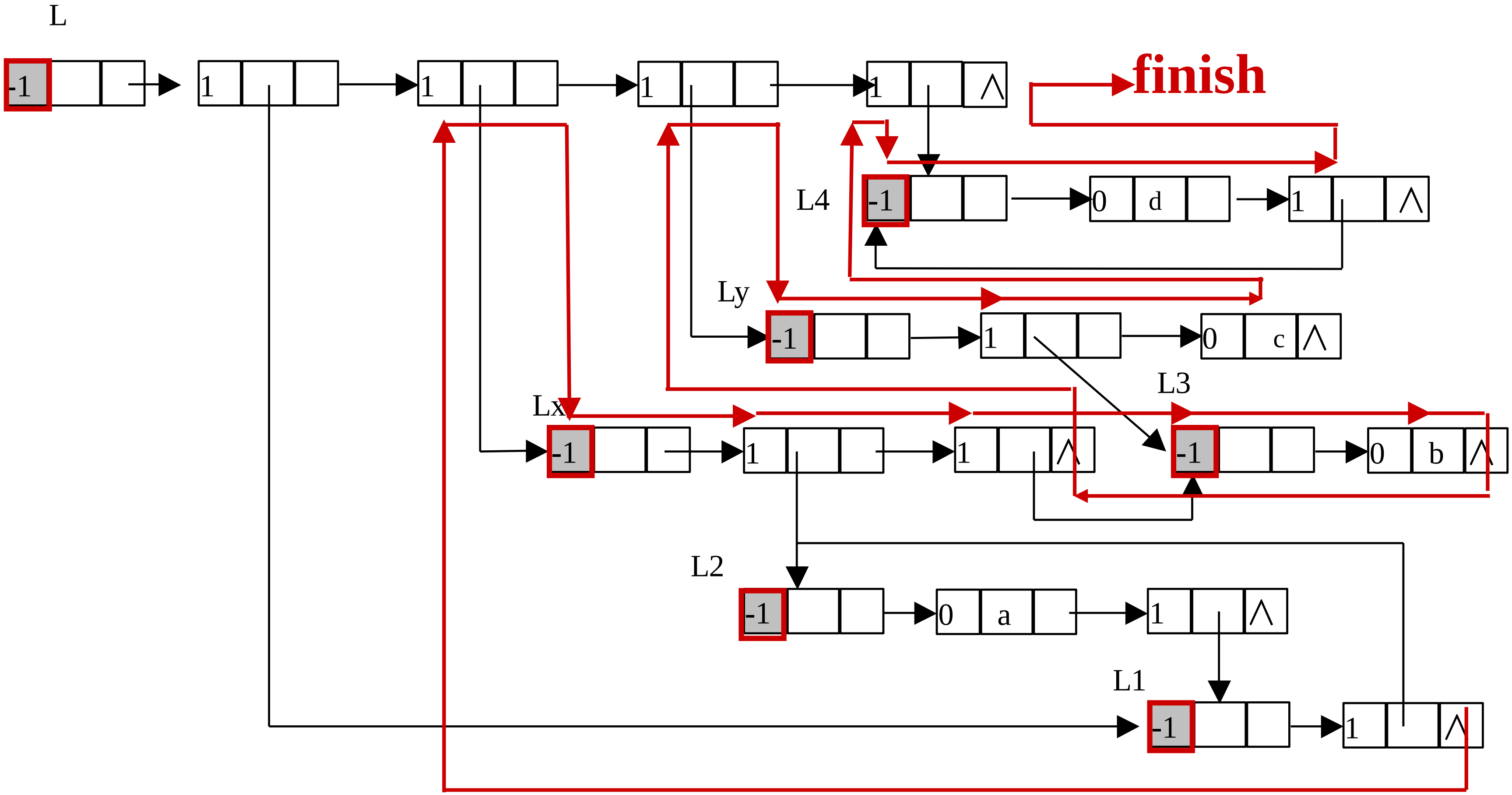
- 增加头指针，简化删除、插入操作



- **带表头结点的循环广义表**



➢ 带表头结点的循环广义表



12.3 Trie 结构

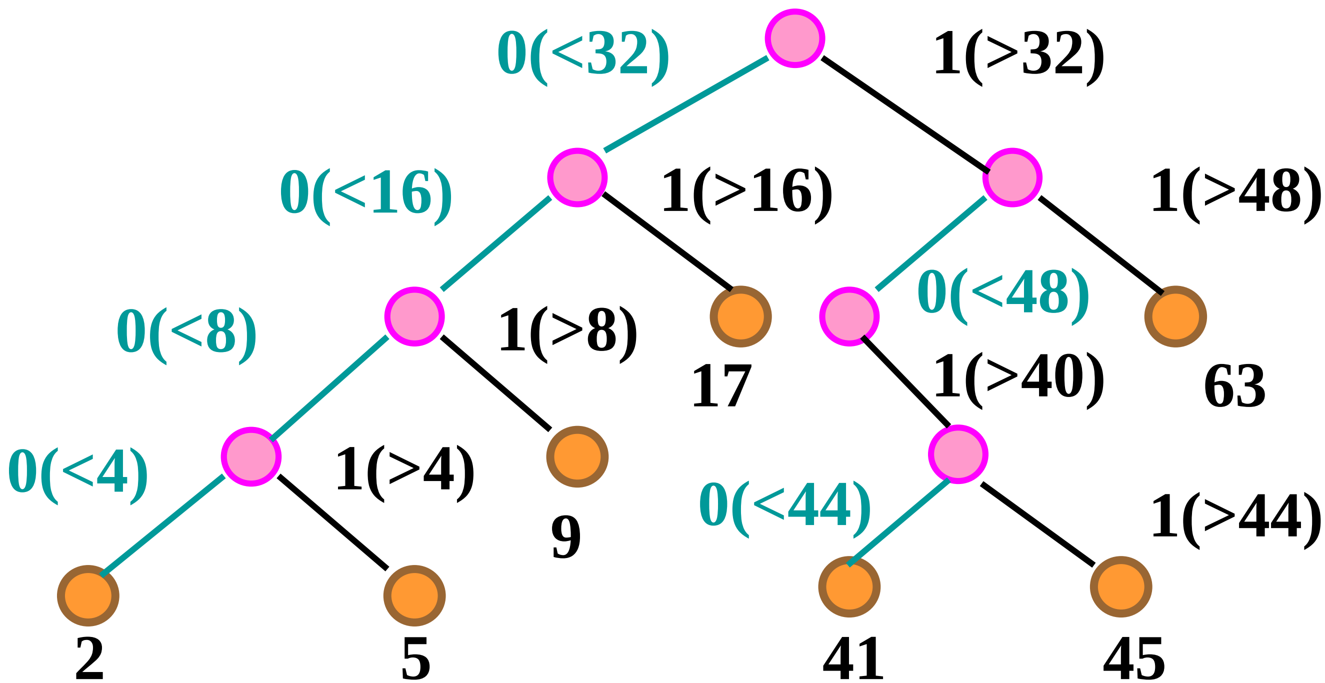
- **BST**是一类基于对象空间分解的数据结构
  - 关键码范围分解由关键码值决定
  - 与值插入的次序紧密相关
- 若要消除**BST**的不平衡，根本办法是改变关键码的划分方式

- **关键码空间分解：** 对关键码范围进行均分
- 基于关键码分解的数据结构，叫做Trie
  - 又称**前缀树或字典树**，是一种有序树，键通常是字符串
- Trie树基于两个原则
  - （1）关键码集合固定
  - （2）对结点分层标记

二叉Trie结构

- 在关键码范围内对树结构中每一结点预定义划分位置

元素为2、5、9、17、41、45、63

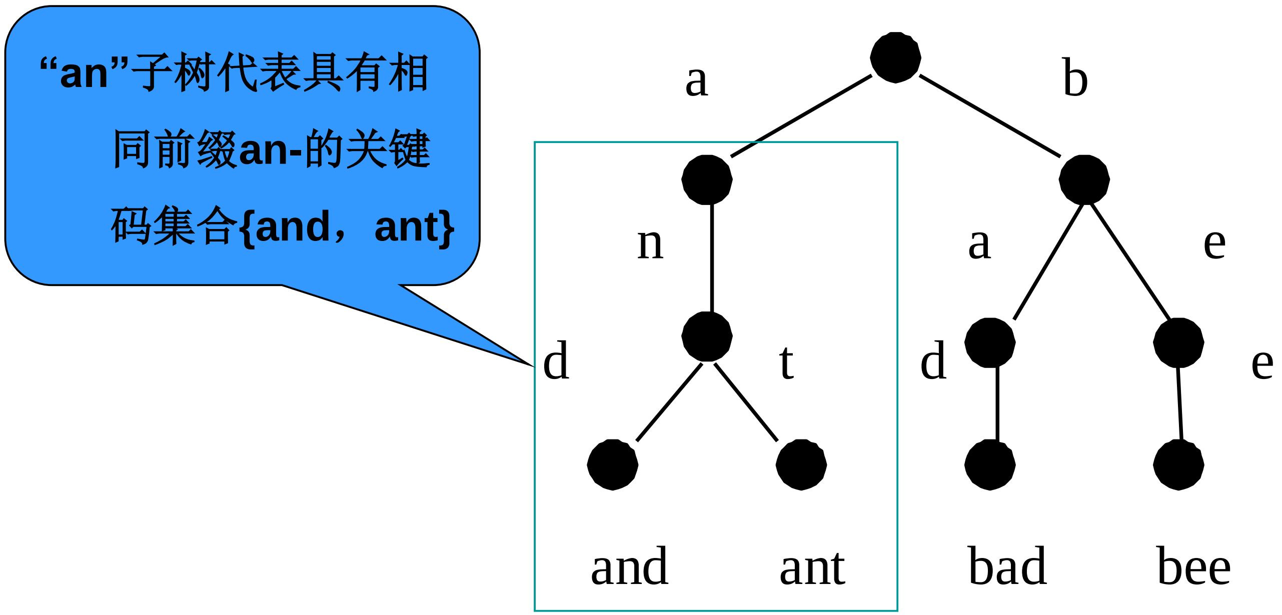


Trie的基本性质

- 根节点不包含字符
- 除根节点外每一个节点都只包含一个字符
- 从根节点到某一节点路径上经过的字符连接起来，为该节点对应的字符串
- 每个节点的所有子节点包含的字符都不相同

英文字符树——26叉Trie，前缀树

存单词 and、ant、bad、bee



- 一棵子树代表具有相同前缀的关键码的集合

在Trie树上进行查找

- **在Trie树中查找单词 *aba***

1. 在trie树上进行检索总是始于根结点
2. 取得要查找关键词的第一个字母（例如 ***a*** ），并根据该字母选择对应的子树并转到该子树继续进行检索
3. 在相应的子树上，取得要查找关键词的第二个字母（例如 ***b***），并进一步选择对应的子树进行检索
4. ...
5. 在某个结点处，关键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找

关于Trie树

- 在trie树中查找一个关键字的效率和树中包含的结点数无关，而取决于组成关键字的字符数。
  - 二叉查找树的查找效率和树中结点数有关$O(\log_2 n)$。
- 如果要查找的关键字可以分解成字符序列且不是很长，利用trie树查找速度优于二叉查找树。
  - 如：若关键字长度最大是5，则利用trie树，利用5次比较可以从$26^5＝11881376$个可能的关键字中检索出指定的关键字。而利用二叉查找树至少要进行$\log_2 26^5=23.5$次比较。

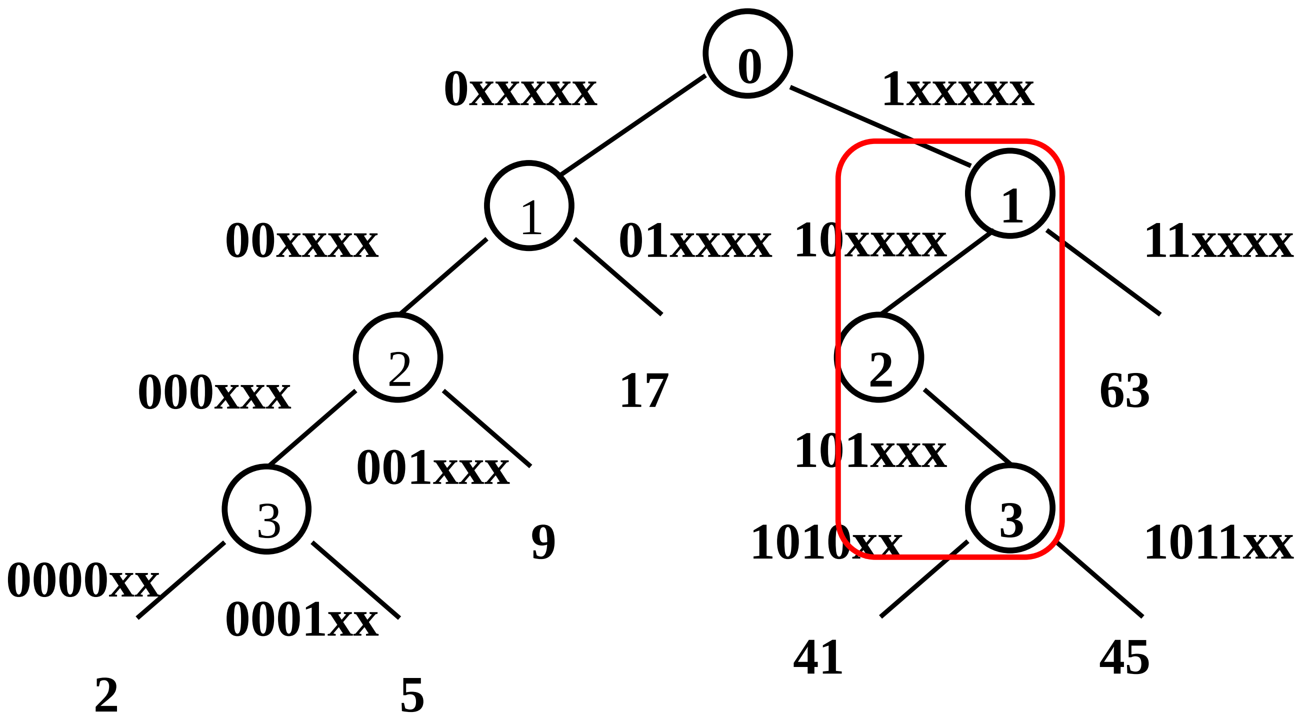
Trie字符树的特点

- Trie结构不是平衡的
  - 例如，用它存取英文单词时，t子树下的分支比z子树下的分支多出很多（以t开头的单词比以z开头的单词多得多）
  - 且每次有26个分支因子使得树结构过于庞大，不便于检索
- 应用场景
  - 词频统计、前缀匹配、排序等

PATRICIA Tree

- 简称PAT-tree
- D.Morrision发明的Trie结构变体
  - 根据关键码二进制位的编码来划分
  - 二叉Trie树

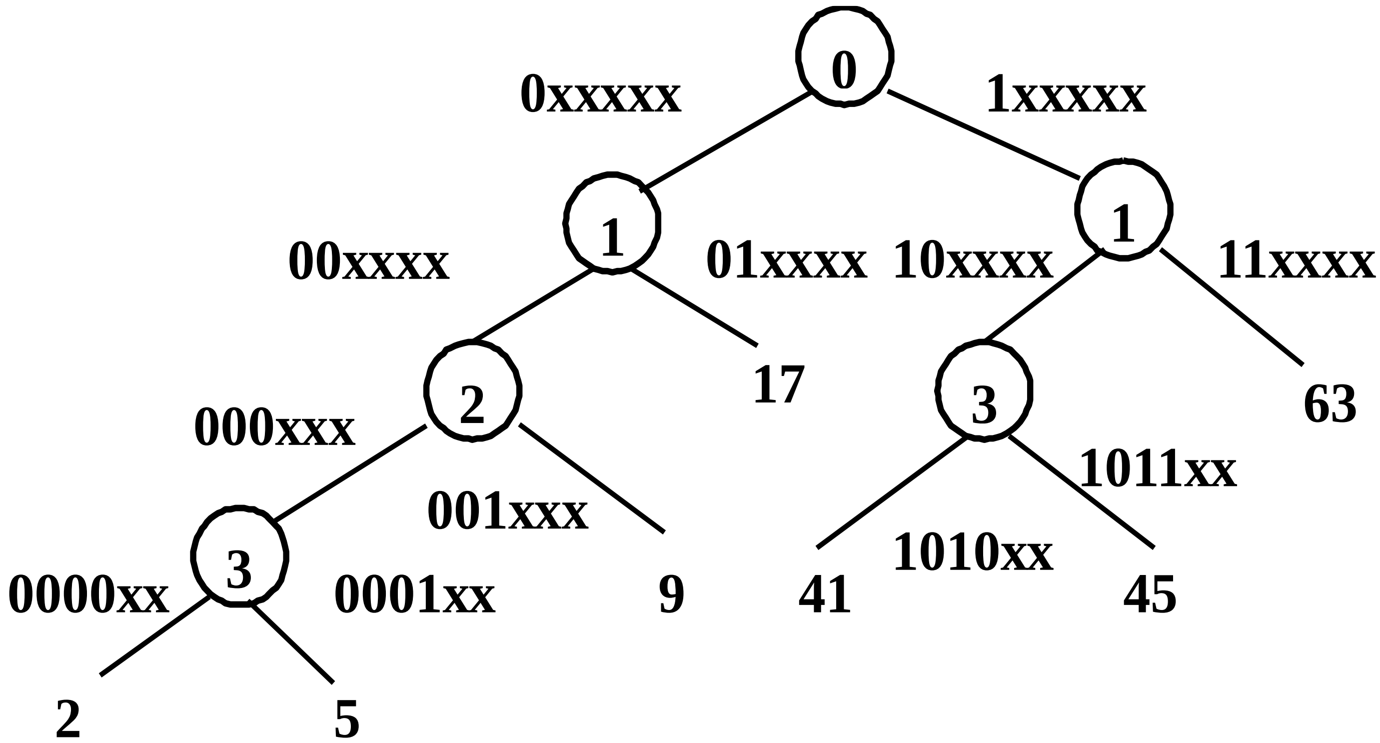
PATRICIA 结构图



编码： 2：000010 5：000101 9：001001

17：010001 41：101001 45：101101 63：111111

压缩PATRICIA 结构



编码：2：000010　 5：000101　9：001001

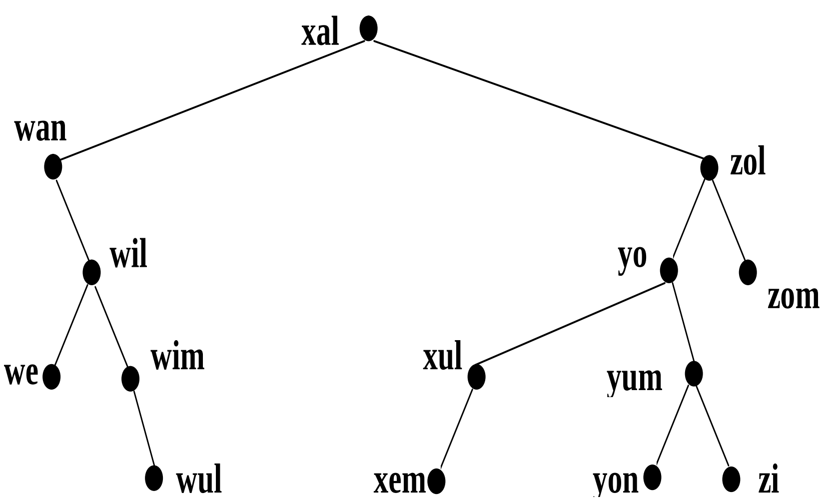
17：010001　41：101001　45：101101　63：111111

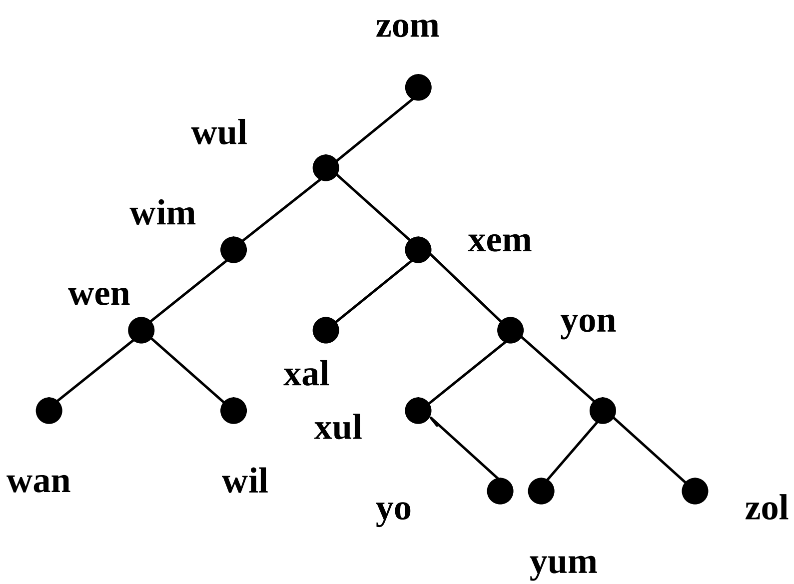
12.4 二叉搜索树的扩展

1. 最佳BST树—基于用户访问习惯
2. 平衡BST树—基于树高平衡约束
3. 伸展树—基于用户动态访问特征

12.4.1 最佳BST树

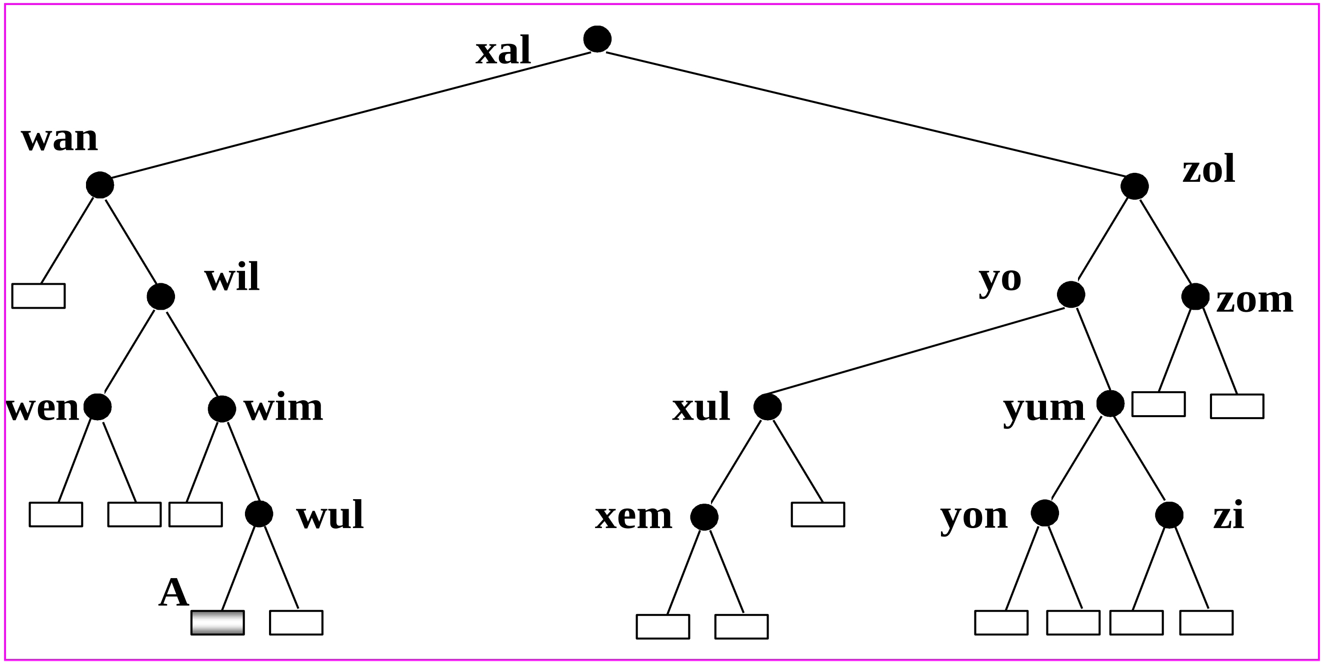
- 同一个关键码集合，插入次序不同，所构成的二叉搜索树也不同





- 一个拥有n个关键码的集合，关键码可以有n!种不同的排列法
  - 是否正好可以构成n!棵二叉搜索树？
- 其实不然，不同排列所构成的BST树有可能相同
  - 例如，{2，1，3}和{2，3，1}
- 可以证明，这n!种排列中， 只有 $\boxed{C_{2n}^{n}-C_{2n}^{n-1}=\frac{1}{n+1}C_{2n}^{n}}$（组合数学中称之为Catalan数)种前序排列能够构成二叉搜索树。
- 如何评价上述BST树的效率呢？

扩充二叉搜索树



- 每个外部结点代表其值处于原来二叉搜索树的两个相邻结点关键码值之间的可能关键码的集合

是

BST树效率的度量

- **成功检索**
  - 比较的次数就是关键码所在的层数加1（<u>根为第0层</u>）
- **不成功检索**
  - 比较次数就等于被检索关键码所属的那个<u>外部结点</u>的层数
- **平均比较次数**

p_i：检索第i个内部结点的的频率

q_i：检索关键码处于第i和第$i+1$内部结点间的频率

$$ASL(n) = \frac{1}{W}\left[\sum_{i=1}^{n} p_i(l_i + 1) + \sum_{i=0}^{n} q_i l'_i\right]$$

$$W = \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i$$

l_i：第i个内部结点的层数

l'_i：第i个外部结点的层数

最佳BST树

- 最佳BST树
  - 根据用户访问频率，寻找平均比较次数ASL(n)最小的二叉搜索树
- 如何构造最佳BST树？
- 两种情况
  - 结点检索概率均等情况
  - 结点检索概率不同情况

1、结点等概率访问

- 结点检索概率相等

$$\frac{p_1}{W}=\frac{p_2}{W}=\cdots=\frac{p_n}{W}=\frac{q_0}{W}=\frac{q_1}{W}=\cdots=\frac{q_n}{W}=\frac{1}{2n+1}$$

- 则有

$$\mathrm{ASL}(n)=\frac{1}{2n+1}\left(\sum_{i=1}^{n}(l_i+1)+\sum_{i=0}^{n}l'_i\right)=\frac{1}{2n+1}\left(\sum_{i=1}^{n}l_i+n+\sum_{i=0}^{n}l'_i\right)$$

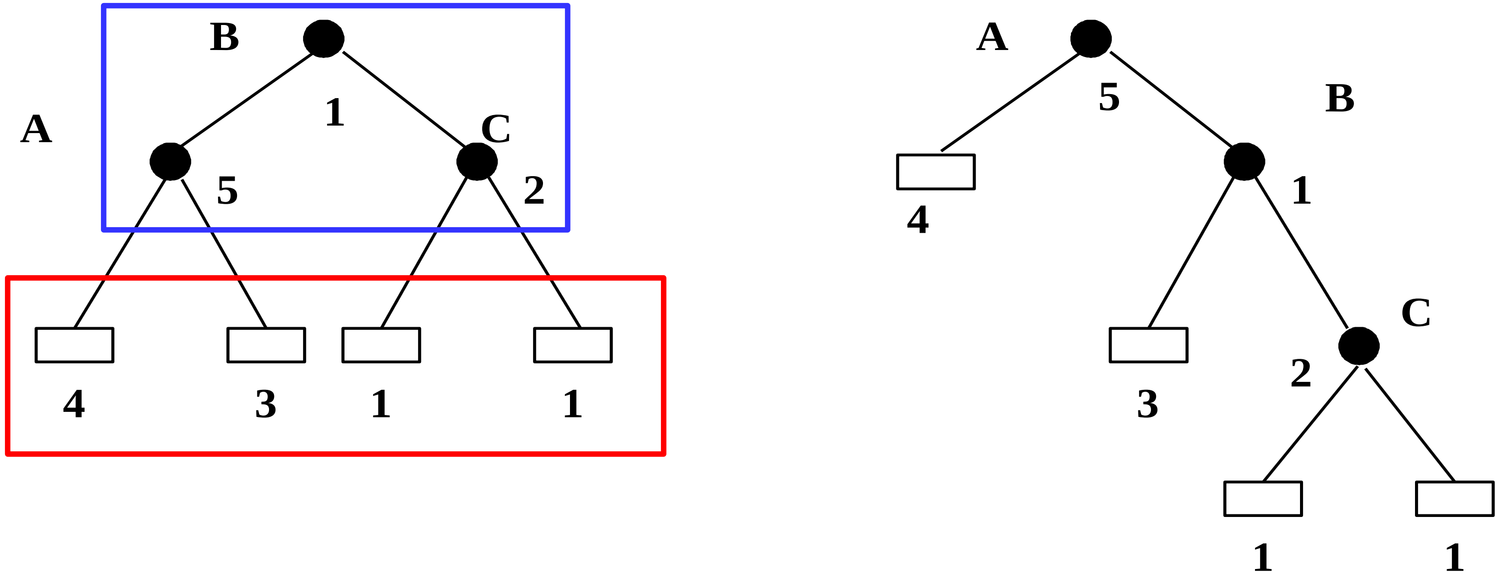
$$=\frac{1}{2n+1}(I+n+E)\;=\frac{2I+3n}{2n+1}$$

E=I+2n

ASL(n) 最小，就是要

内部路径长度I最小

2、结点不等概率访问



$$ASL(n) = \frac{33}{17} \qquad ASL(n) = \frac{29}{17}$$

➢ **问题定义**：给定有序关键码集合{key_1, key_2, …, key_n}，及权重集合{p_1, p_2, …, p_n, q_0, q_1, …, q_n}，如何构造最佳BST（ASL(n)值最小）？

最佳BST树的构造方法

- 最佳BST树性质：**其任何子树都是最佳二叉搜索树**
- 利用这一性质，可以从底层逐步构造最佳BST树
  - 先构造包含1个内部结点的最佳二叉搜索树
  - 再构造包含2个结点的最佳二叉搜索树，…，
  - 直到把所有的结点都包含进去。
- **较大的最佳BST树由较小最佳BST树构造而成**
- **动态规划的基本思想**

若干定义

- 树定义：t(i，j)
  - 包含的内部结点关键码为：(key_{i+1}，key_{i+2}，…，key_j)
  - 内部结点和外部节点的权：(q_i，p_{i+1}，q_{i+1}，…，p_j，q_j)
- 查询代价：C(i，j)

$$\sum_{x=i+1}^{j} p_x(l_x+1) \quad \sum_{x=i}^{j} q_x l'_x$$

- 权的总和：W(i，j)

$$W(i, j) = p_{i+1} + \ldots + p_j + q_i + q_{i+1} + \ldots + q_j$$

动态规划构造法

动态规划过程

- 第1步：构造包含1个关键码的最佳二叉搜索树
  - ✓找t（0，1)，t（1，2)，…，t（n-1，n)
- 第2步：构造包含2个关键码的最佳二叉搜索树
  - ✓找t（0，2)，t（1，3)，…，t（n-2，n)
- 再构造包含3，4，…个关键码的最佳二叉搜索树
- 最后构造包含n个关键码的t（0，n）

对于子树 T(i, j)

- 以key_k为根
  - 左子树包含key_{i+1}，…，key_{k-1}
    - ✓C(i, k-1)已求出
  - 右子树包含key_{k+1}，key_{k+2}，…，key_j
    - ✓C(k, j)已求出
- $C(i, j) = W(i, j) + \min_{i<k\leq j} (C(i, k-1) + C(k, j))$

例子

- 给出关键码集合

$$\left\langle \begin{matrix} B, & D, & F, & H \\ key_1, & key_2, & key_3, & key_4 \end{matrix} \right\rangle$$

- 及权的序列

$$\left\langle \begin{matrix} 1, & 5, & 4, & ,3, & 5, & 4, & 3, & 2, & 1 \\ p_1, & p_2, & p_3, & p_4, & q_0, & q_1, & q_2, & q_3, & q_4 \end{matrix} \right\rangle$$

- 构造最佳二叉搜索树

例子(续)

i \ j	0	1	2	3	4
0	0	1	2	2	2
1		0	2	2	3
2			0	3	3
3				0	4
4					0

r(i ,j)

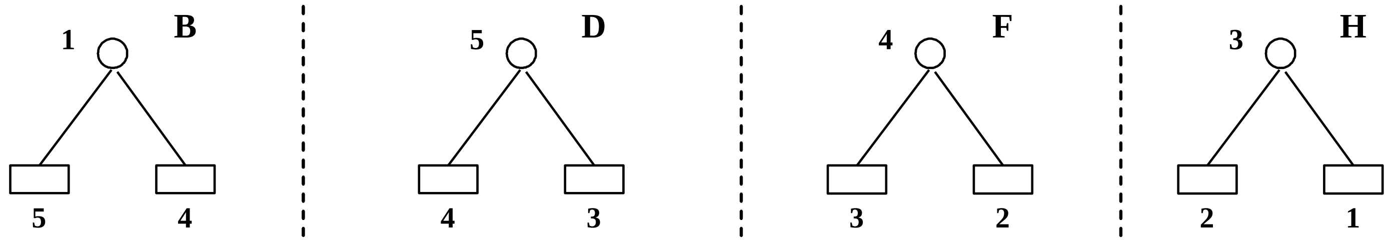
i \ j	0	1	2	3	4
0	0	10	28	43	57
1		0	12	27	40
2			0	9	19
3				0	6
4					0

C(i ,j)

i \ j	0	1	2	3	4
0	5	10	18	21	28
1		4	12	18	22
2			3	9	3
3				3	6
4					1

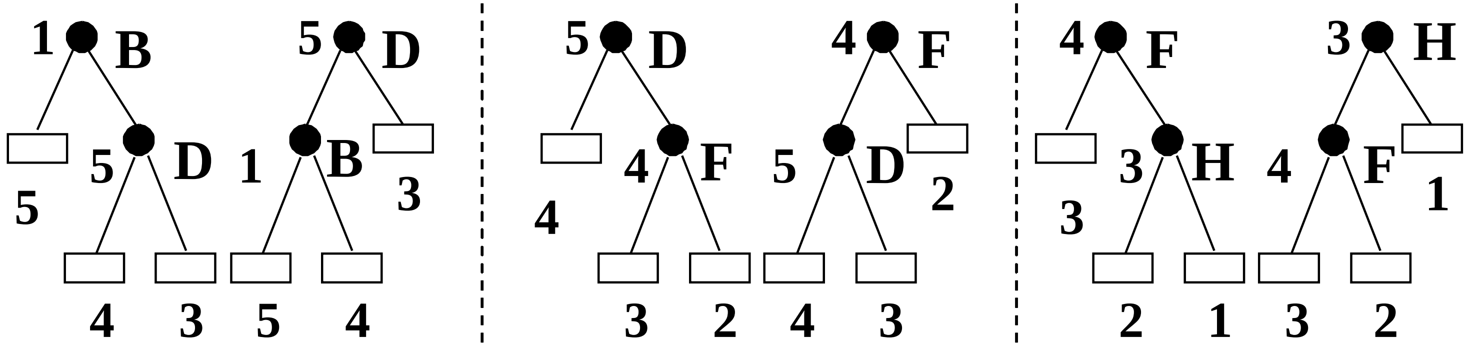
W(i ,j)

第一步



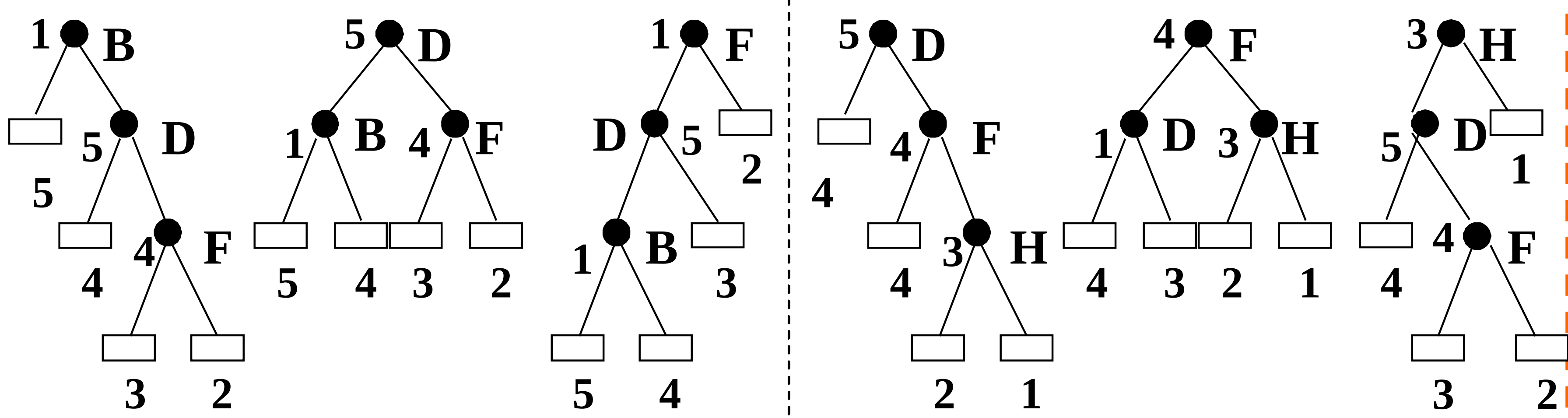
花费	$\boxed{10}$	$\boxed{12}$	$\boxed{9}$	$\boxed{6}$
总权	10	12	9	6

第二步



开销	30	28	27	30	19	22
总权	18	18	18	18	13	13

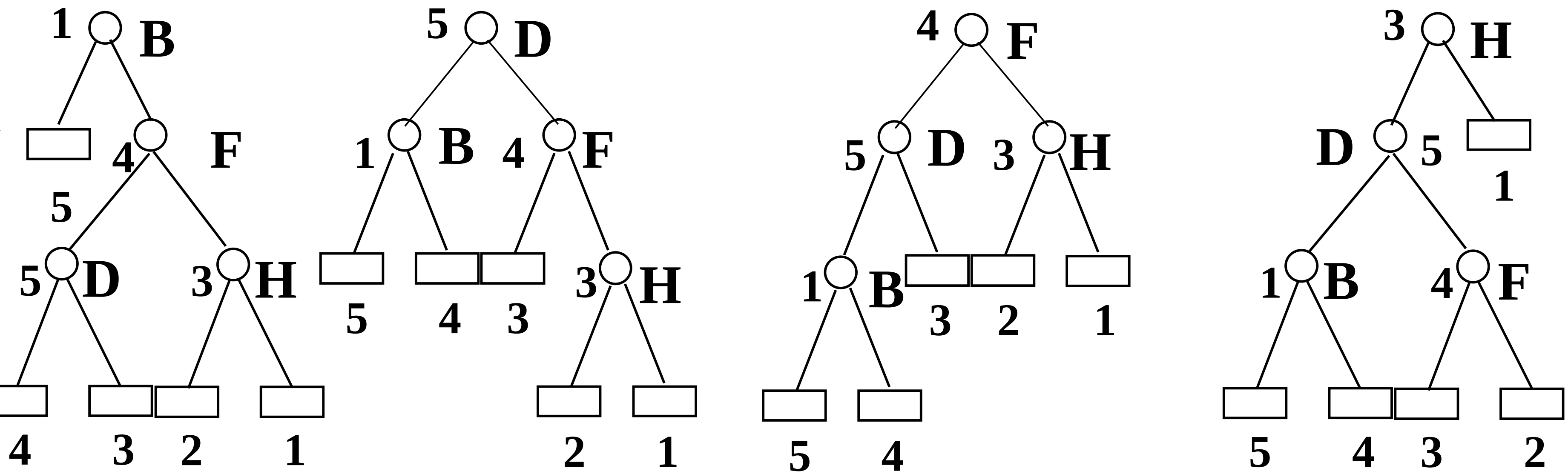
第三步



花费	51	43	52	41	40	49
总权	24	24	24	22	22	24

最佳二叉搜索树 t(i, j)

第四步



花费	68	57	62	71
总权	28	28	28	28

如何动态地保持最佳?

- 根据关键码集合及检索概率，可以构造出最佳二叉检索树
- 问题： 静态，经过若干次插入、删除后可能会失去平衡，检索性能变坏
- 如何动态保持一棵二叉检索树的平衡，从而有较高的检索效率

平衡树技术

12.4.2 平衡二叉搜索树(AVL)

- **Adelson-Velskii和Landis发明了AVL树，是一种平衡的二叉搜索树**
- **AVL树的性质**
  - 空二叉树是一个AVL树；
  - 如果T是一棵AVL树，那么其左右子树T_L、T_R也是AVL树，并且$|h_R-h_L| \le 1$， h_L、h_R是其左右子树的高度；
  - 如果T是一棵具有n个结点的AVL树，其高为O(log *n*)。

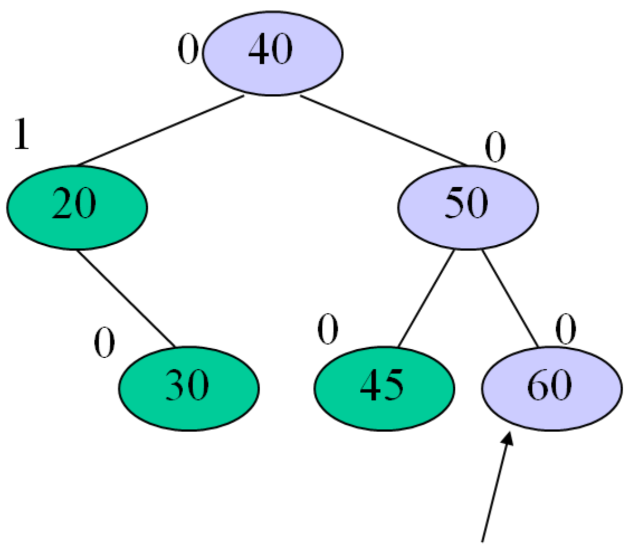
平衡因子bf（balance factor）

- ***bf*(*x*)**：表示结点*x*的平衡因子
  - 定义：***bf*(*x*) = *x*的右子树高度 – *x*的左子树高度**
  - ***bf*(*x*) 取值：{0, 1, -1}**
  - **AVL树的ASL: O($\log_2 n$)**
- **AVL树的平衡调整**
  - **结点插入**
  - **结点删除**

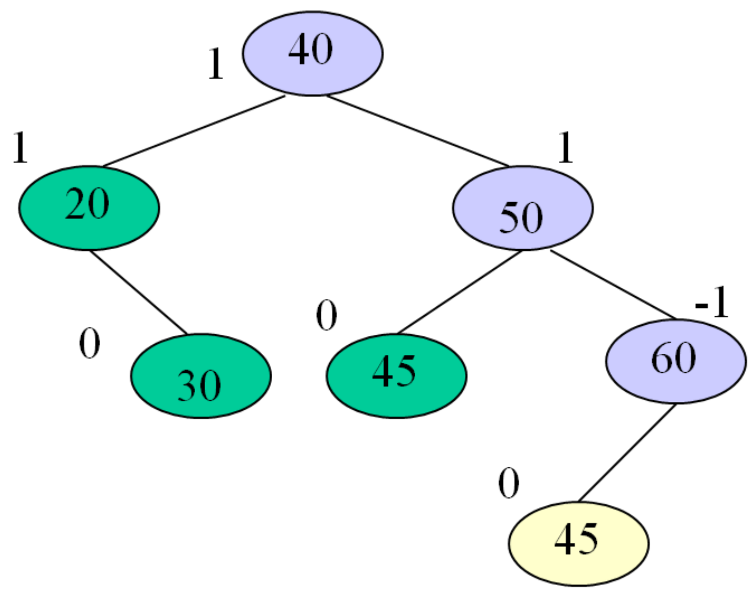
1．AVL树结点的插入

- 与BST树类似，执行失败的查找，确定插入位置插入
- 根据平衡因子决定是否调整
  - **第一情况**：原来树平衡，插入新结点后发生偏重但未失衡

插入55前

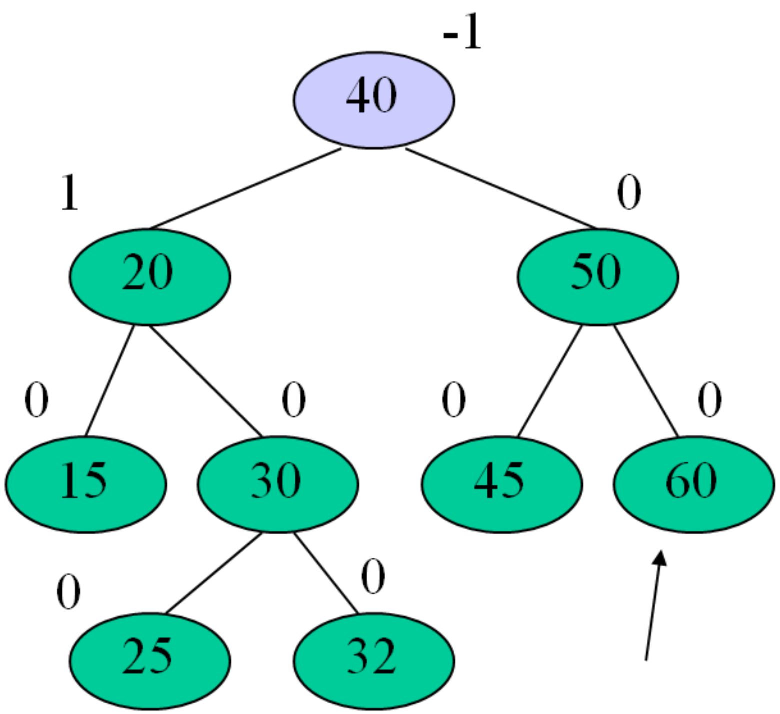


插入55后

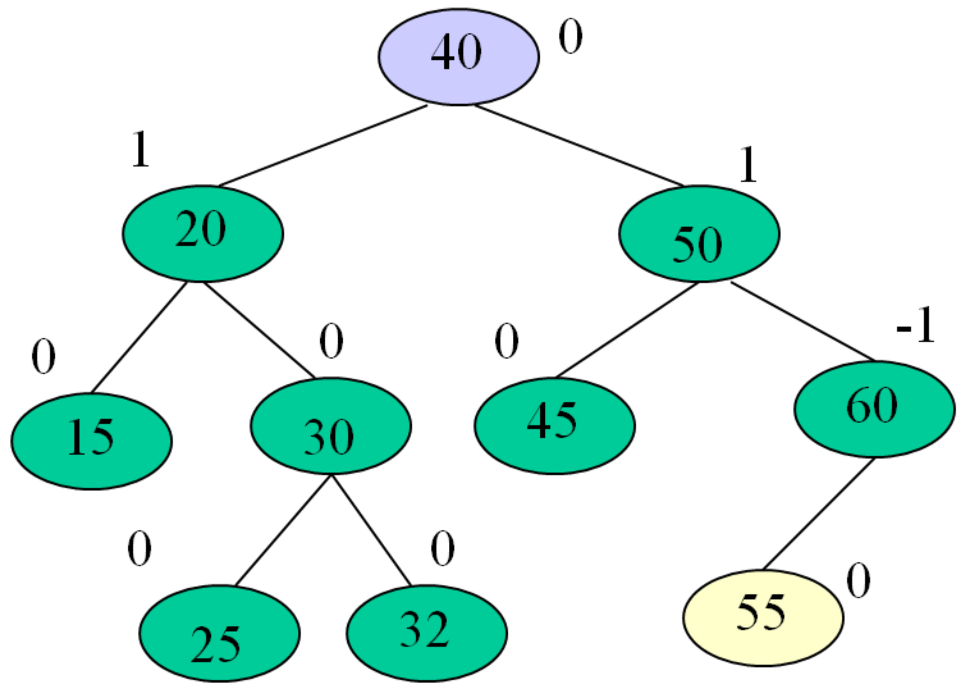


- **第二情况**：路径上原来各结点是有偏重的，插入新结点后，反而使这些结点恢复平衡。

插入55前

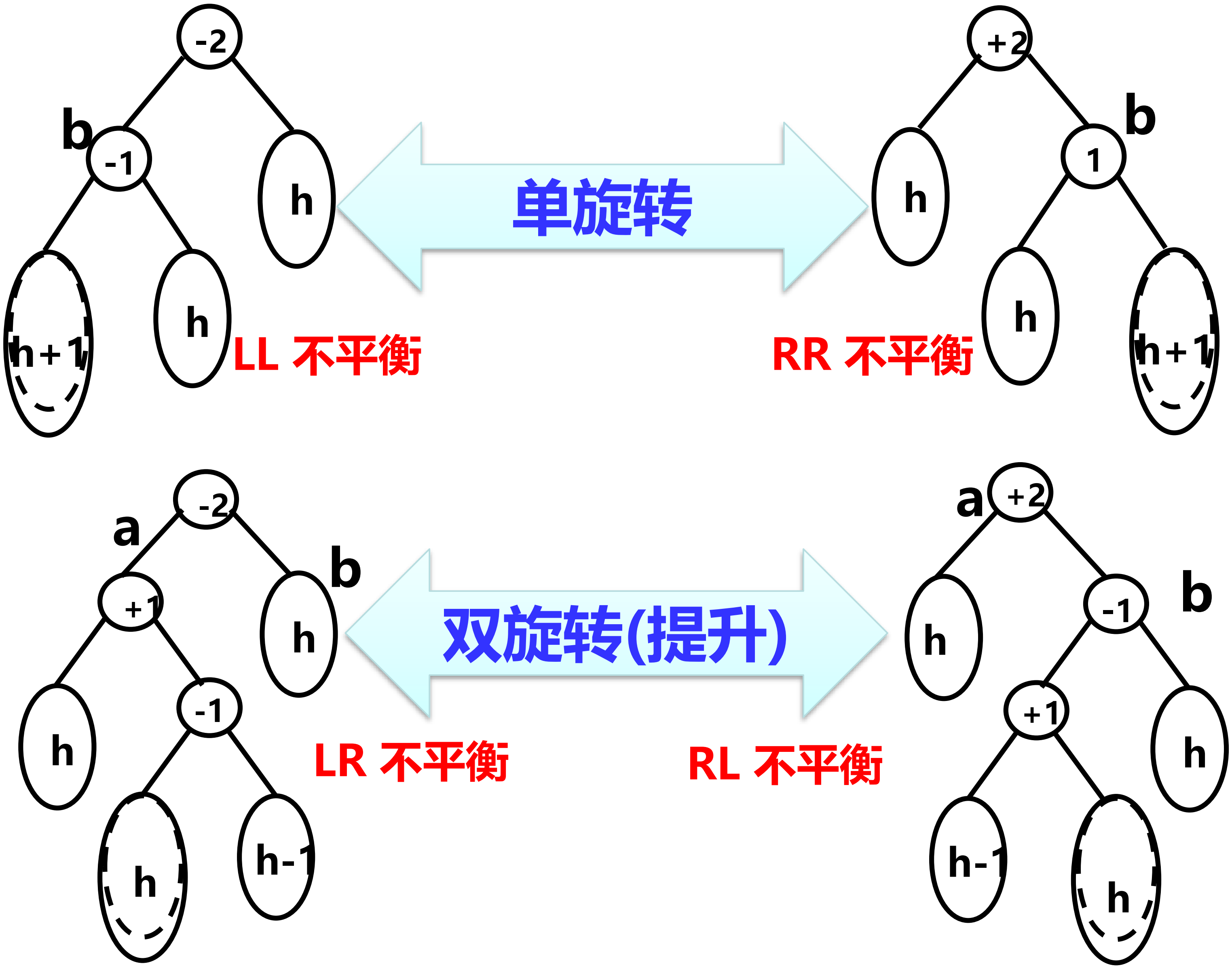


插入55后



- **第三种情况**：插入结点后失衡，执行平衡操作
- 分四种情况（设A为根节点）
  - **LL型**：A的左子树的左子树导致失衡，A的平衡因子为-2
  - **LR型**：A的左子树的右子树导致失衡，A的平衡因子为-2
  - **RL型**：A的右子树的左子树导致失衡，A的平衡因子为2
  - **RR型**：A的右子树的右子树导致失衡，A的平衡因子为2
- **失衡结点一定在根结点与新加入结点间的路径上**，并且其平衡因子只能是2或者-2（之前是1或者-1）

四种不平衡情况



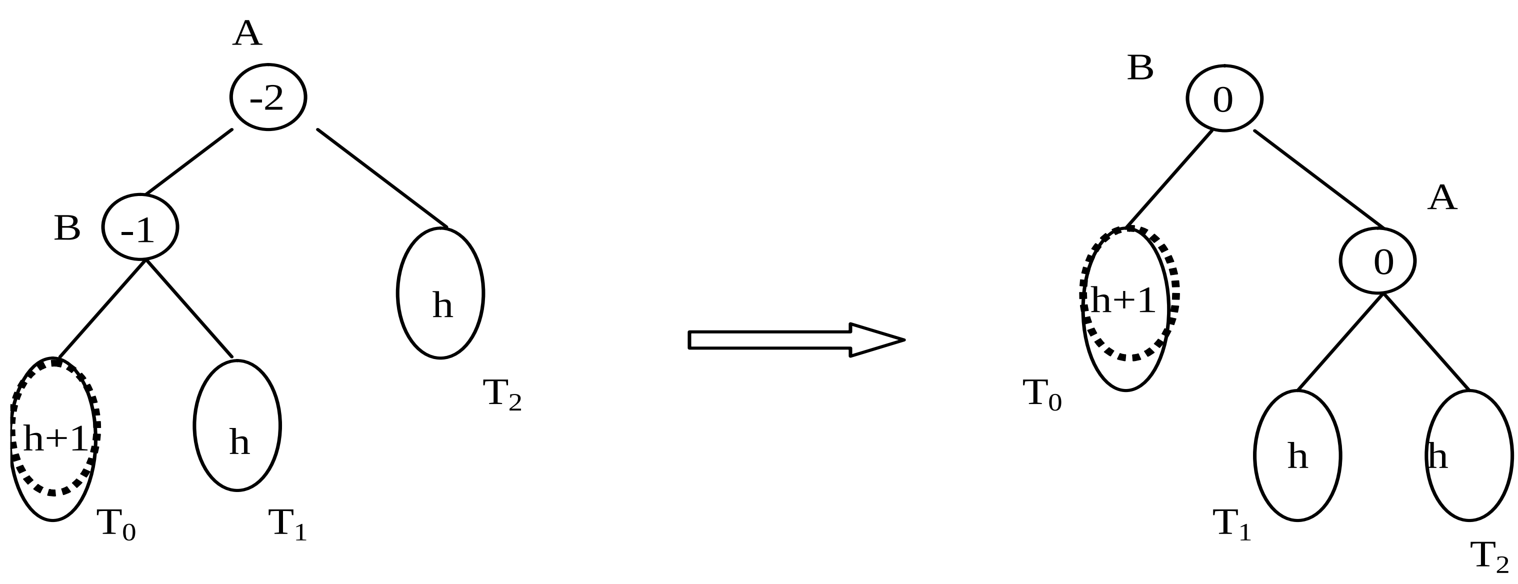
-2
b
-1
h
h+1
h
单旋转
LL 不平衡
+2
b
1
h
h
h+1
RR 不平衡
a
-2
b
+1
h
h
-1
h
h-1
双旋转(提升)
LR 不平衡
RL 不平衡
a
+2
b
-1
h
+1
h
h-1
h

AVL树的重构

- **从新加入的结点 *u* 向根方向往上搜索，直到找到距离u最近的不平衡祖先结点 *A*（平衡因子为2或者-2）**
- 令**A**结点指向结点**u**的路径上的第一个子结点为**B**，第二个子结点为**C**
  - **重构操作发生在“A->B->C”结点上**
- 然后根据失衡形态进行**单旋**或者**双旋操作**

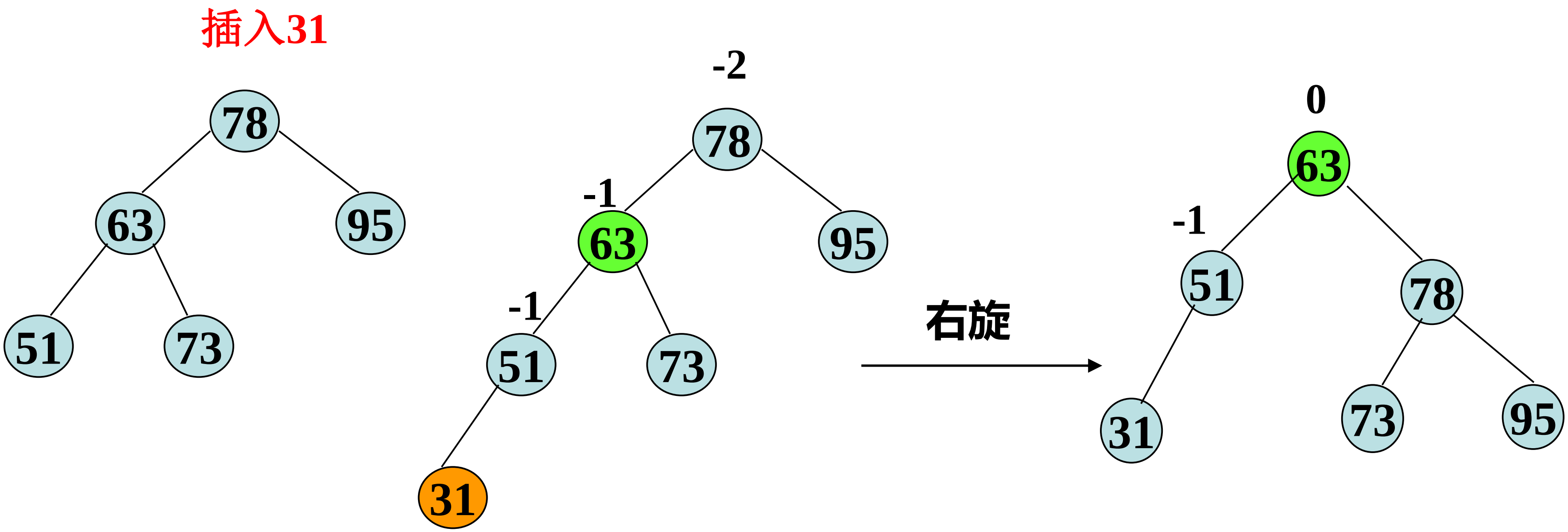
1、单旋转

- **调整过程仅涉及A和B两个结点**（以LL型调整为例）
  - 令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$\underline{T_0\ B\ T_1\ A\ T_2}$



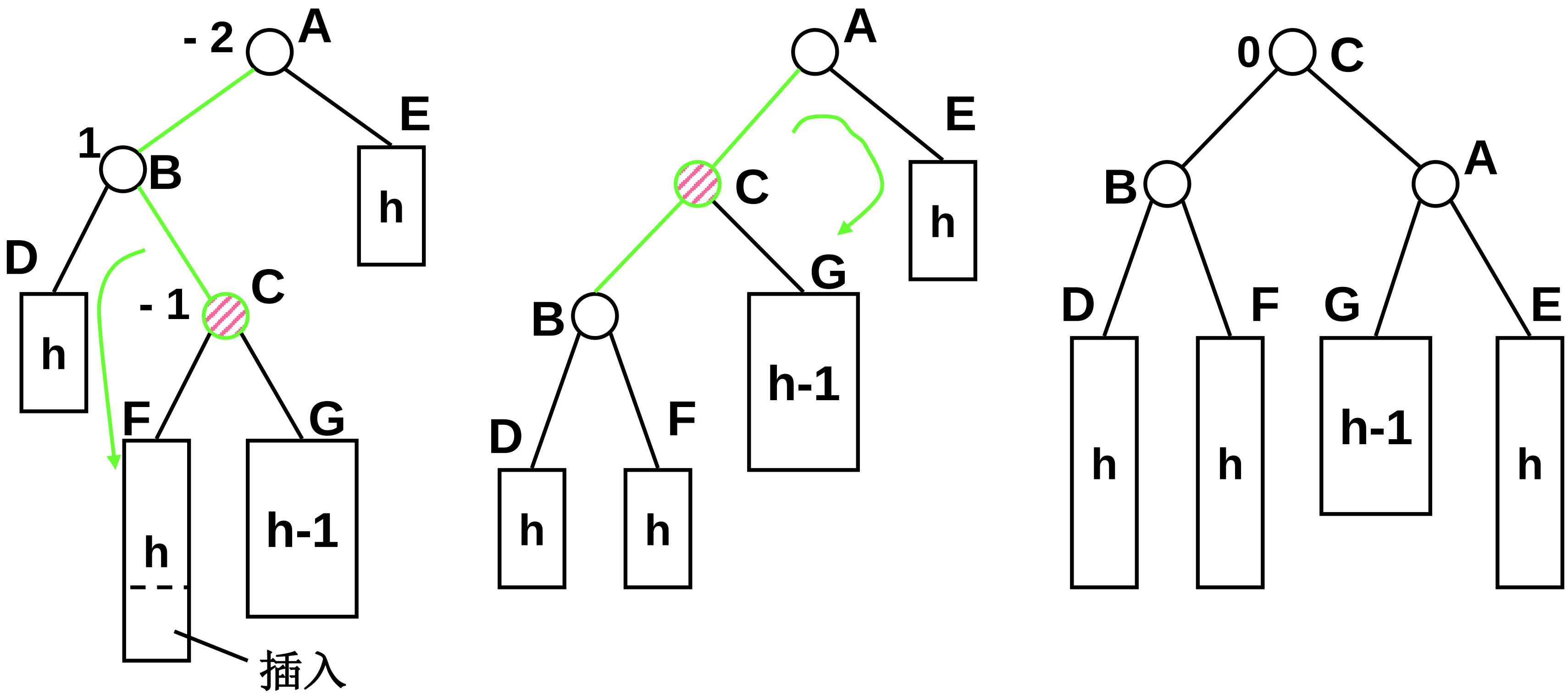
单旋转：LL型示例

- 导致不平衡的结点为A的左子树的左子树，这时A的平衡因子为-2



2、双旋转（提升中间节点）

- **调整过程涉及ABC三个结点**（以LR型调整为例）



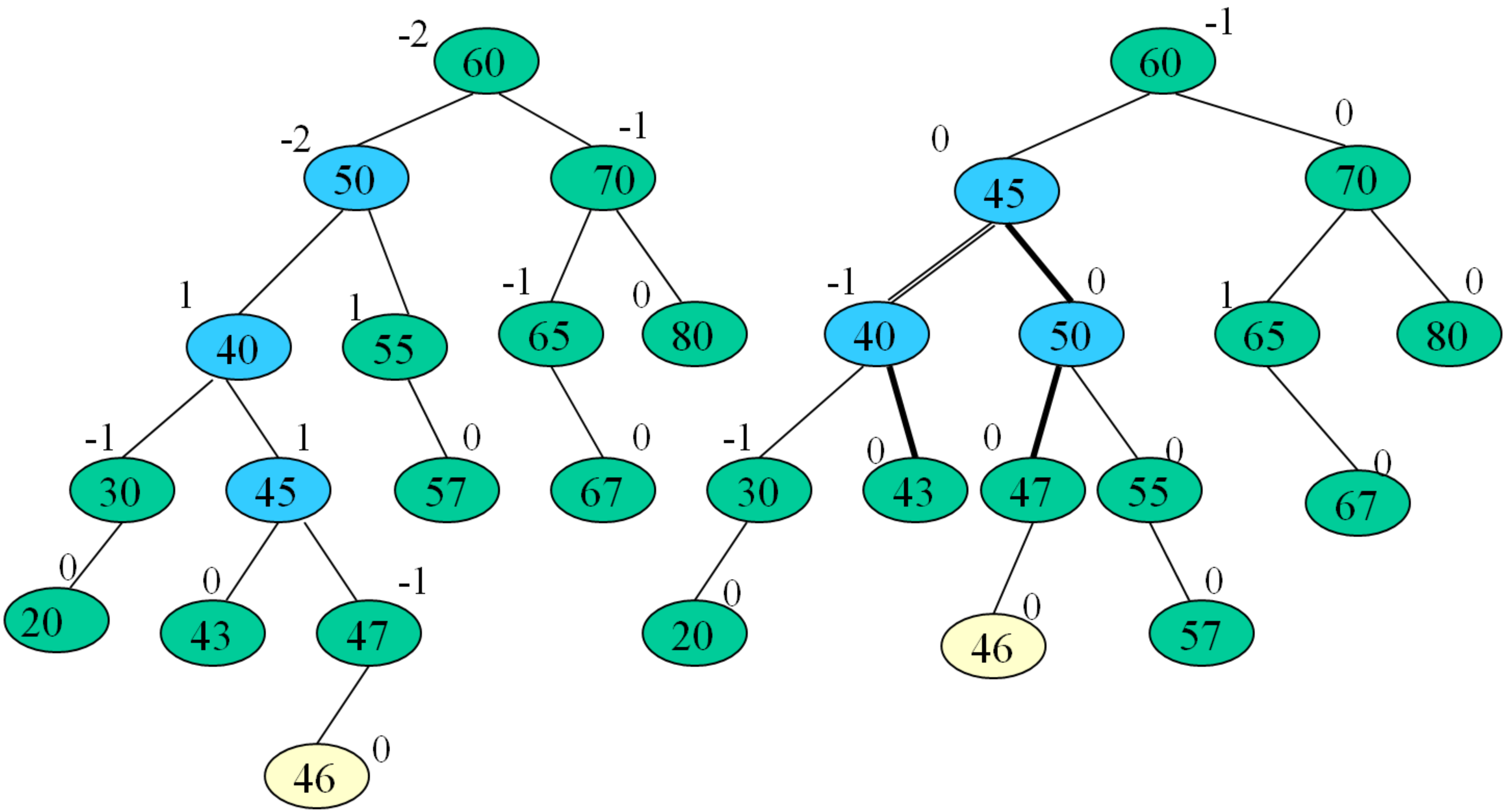
(a) F子树插入结点高度变为h

(b)绕C，将B逆时针转后

(c)绕C，将A顺时针转后

双旋转：LR型示例

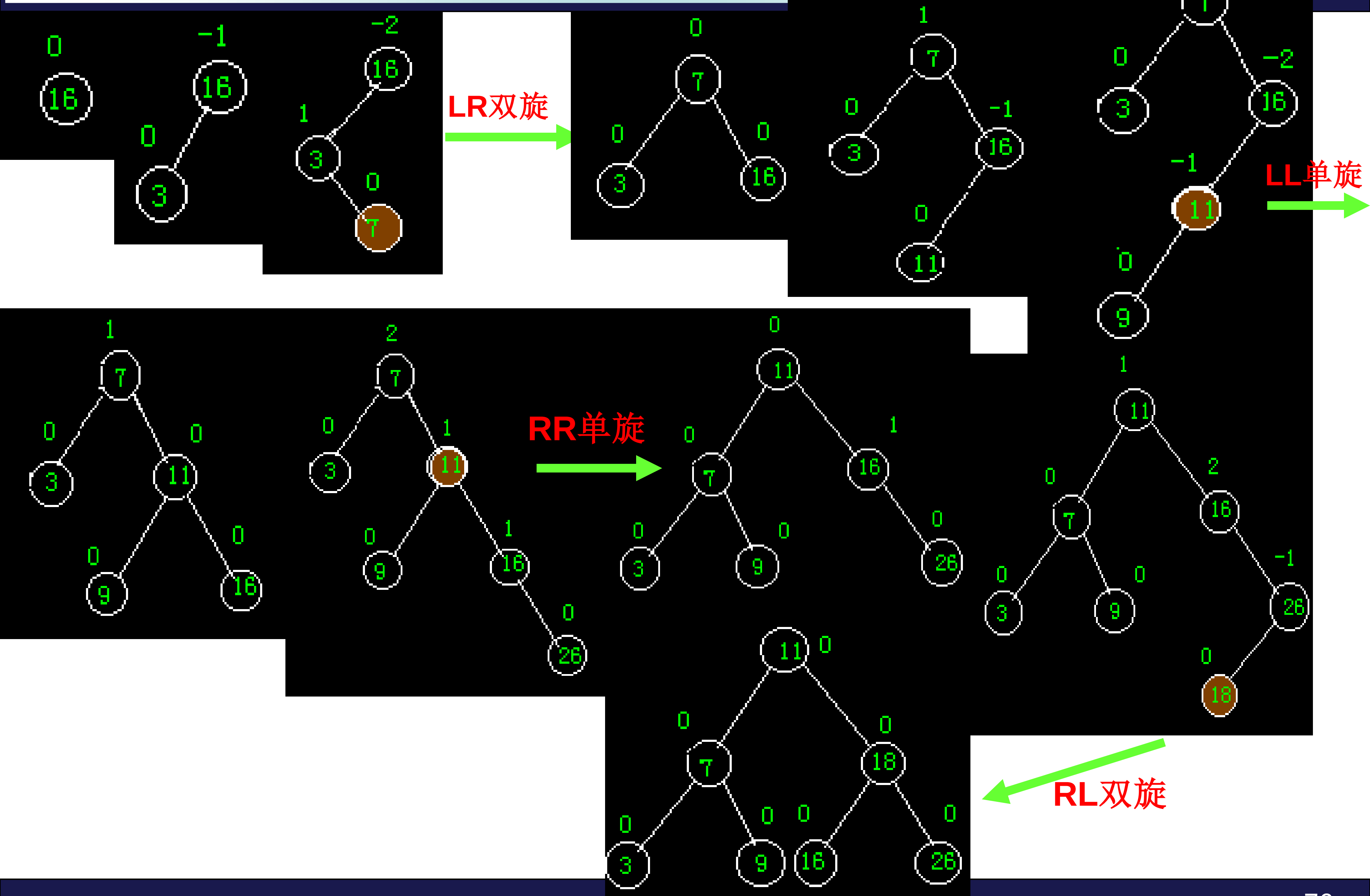
- 插入结点46



AVL树的建立

- 从空开始不断插入一组关键码序列，构成AVL树
- 每插入一个结点后就应判断从该结点到根的路径上有无结点发生不平衡
- 如有不平衡问题，利用旋转方法进行树的调整，使之平衡化
- 建AVL树过程是不断插入结点和必要时进行平衡化的过程

建树过程：依次输入16,3,7,11,9,26,18



2．AVL树结点的删除

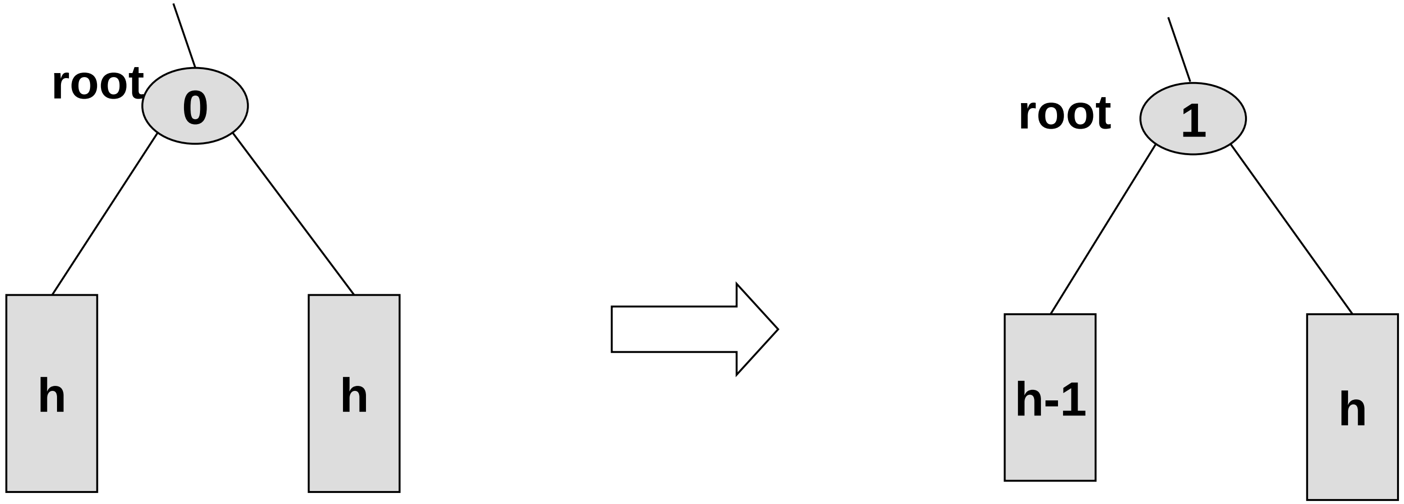
- 删除操作同BST删除：**与后继交换再删除**
- 删除会导致树高及平衡因子变化，需要**沿着被删除结点到根结点的路径**来调整这种变化。
- **删除操作比插入操作复杂**
  - 删除后的再平衡可能需要在相应的路径上的**不止一次实施单旋或双旋**

AVL删除后的调整

- 删除结点后，如果导致**AVL**树失衡，需要执行类似**LL**、**RR**、**LR**和**RL**操作进行平衡调整
- 如平衡调整后不导致新的失衡，则不必继续向上回溯
  - 用布尔变量***modified***来标记（初值为**TRUE**），当为**FALSE**时，停止回溯
- 分情况处理，假定
  - 最下层不平衡发生在结点**root开始的子树中**
  - 删除操作发生在**root**的左子树中

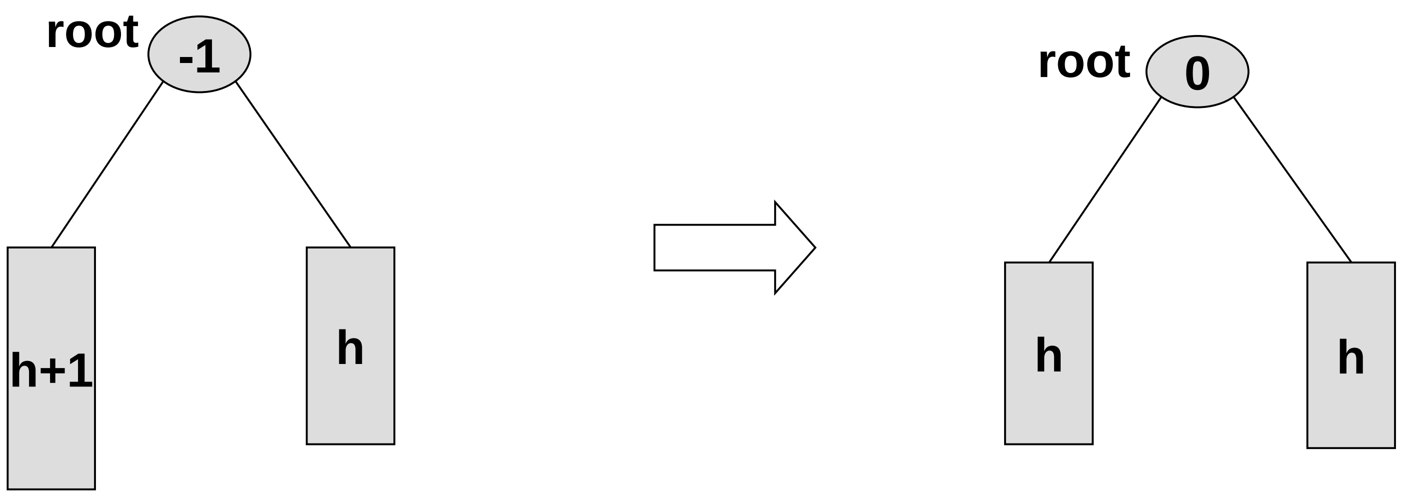
删除操作：Case 1

- *bf*(root)=0：若其左或右子树被缩短，则将其bf值置为1或-1，同时置modified = FALSE
- 由于以root 为根的子树高度未变，故不影响祖先结点的bf，调整结束



删除操作：Case 2

- *bf*(root)≠0，**且较高子树被缩短**：bf值变为0，修改 modified =TRUE，需继续向上修改
  - 由于以root为根的子树高度减1，可能影响父结点的bf值



删除操作：Case 3

- *bf*(root)≠0，**且较低子树被缩短**： root失衡（设较高子树为right ），有如下三种情况

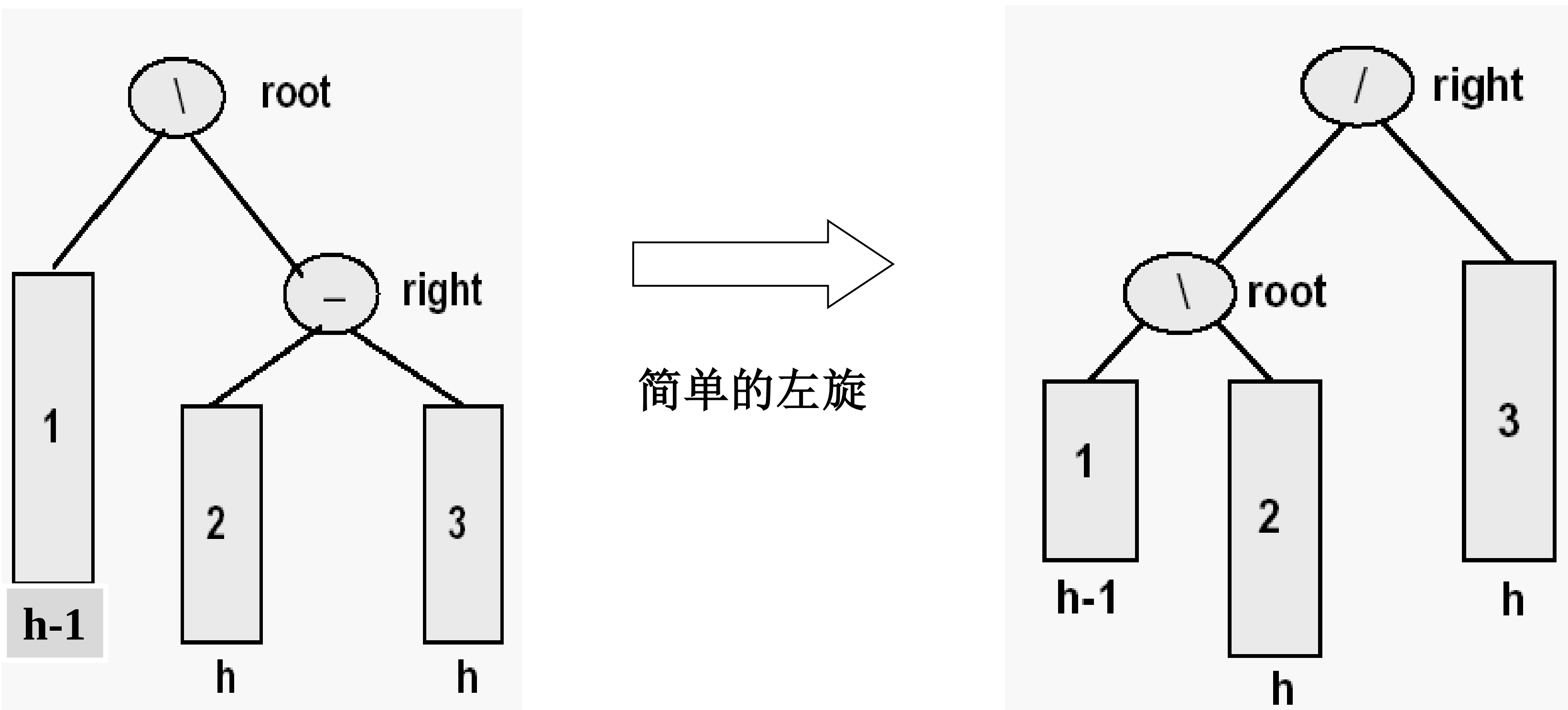
I. **Case 3.1: bf(right)=0**
   - 执行单旋恢复root平衡，*modified = FALSE*

II. **Case 3.2: bf(right)与bf(root)相同**
   - 执行单旋来恢复平衡，*modified = TRUE*

III. **Case 3.3: bf(right)与bf(root)不同**
   - 执行双旋来恢复平衡，*modified = TRUE*

删除操作: Case 3.1

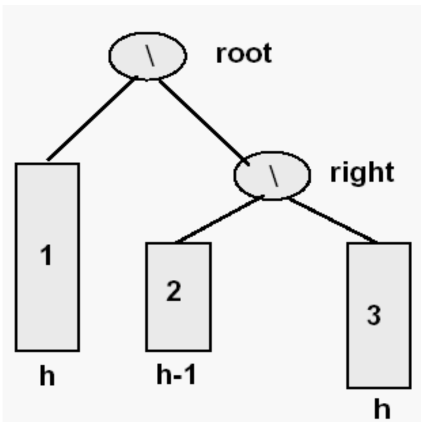
假定root的右子树是平衡的：bf(right)=0



modified = FALSE

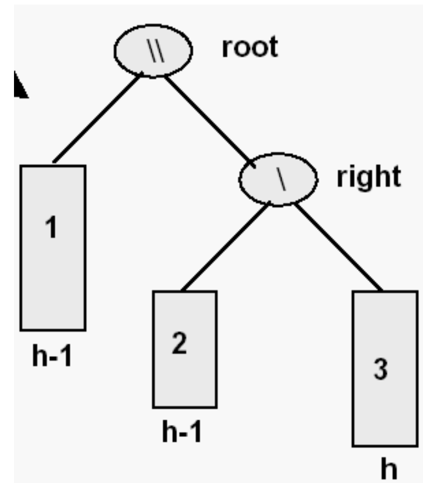
删除操作: Case 3.2

bf(right)与bf(root)相同



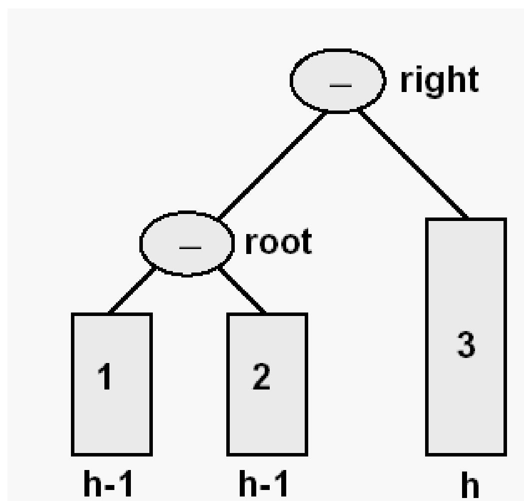
删除前的子树

树高=h+2



删除后的子树

树高=h+1

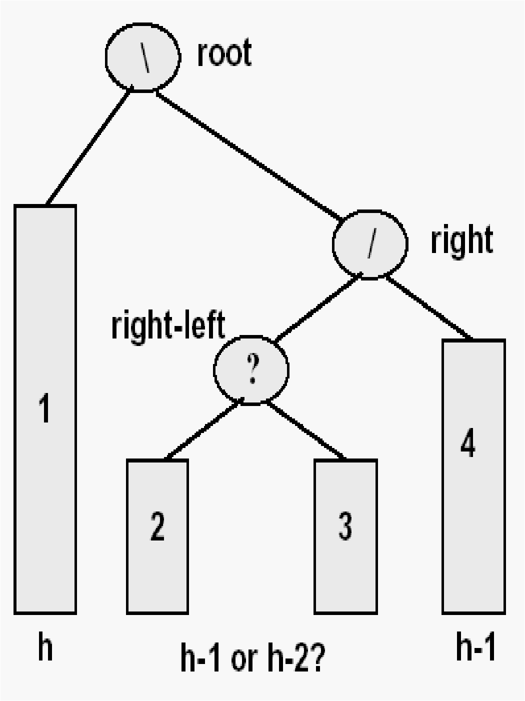


简单的左旋可恢复平衡

modified = TRUE

删除操作: Case 3.3

bf(right)与bf(root)不同，面临如下情形：



从root 的左子树中删除一个结点将导致root更加右重

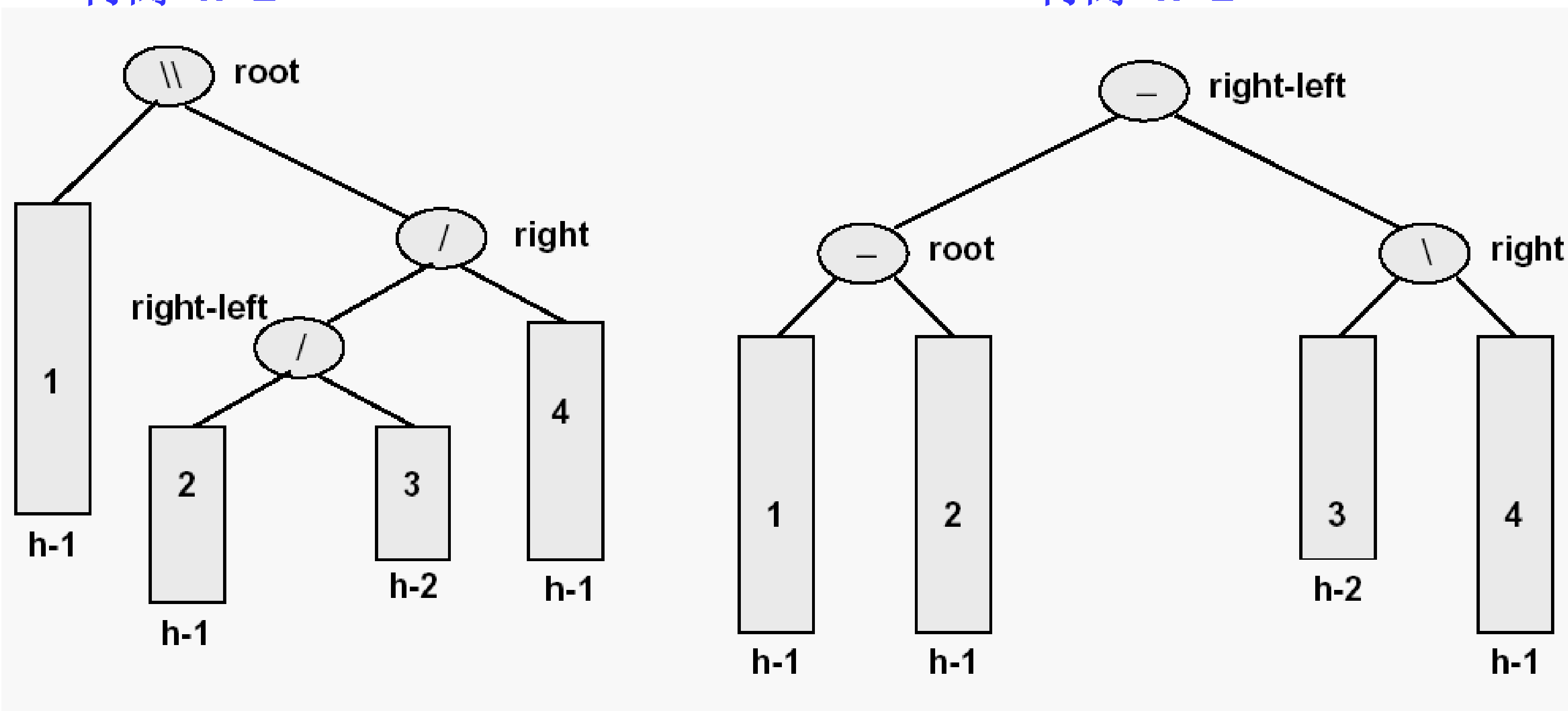
平衡的修复通过对right 实施一个右旋、之后对root 实施一个左旋即可

但需注意的是各平衡因子，与right-left原来的平衡因子有关 ……

删除操作: Case 3.3.1

如果right-left子树也是左重的，可得如下结果：

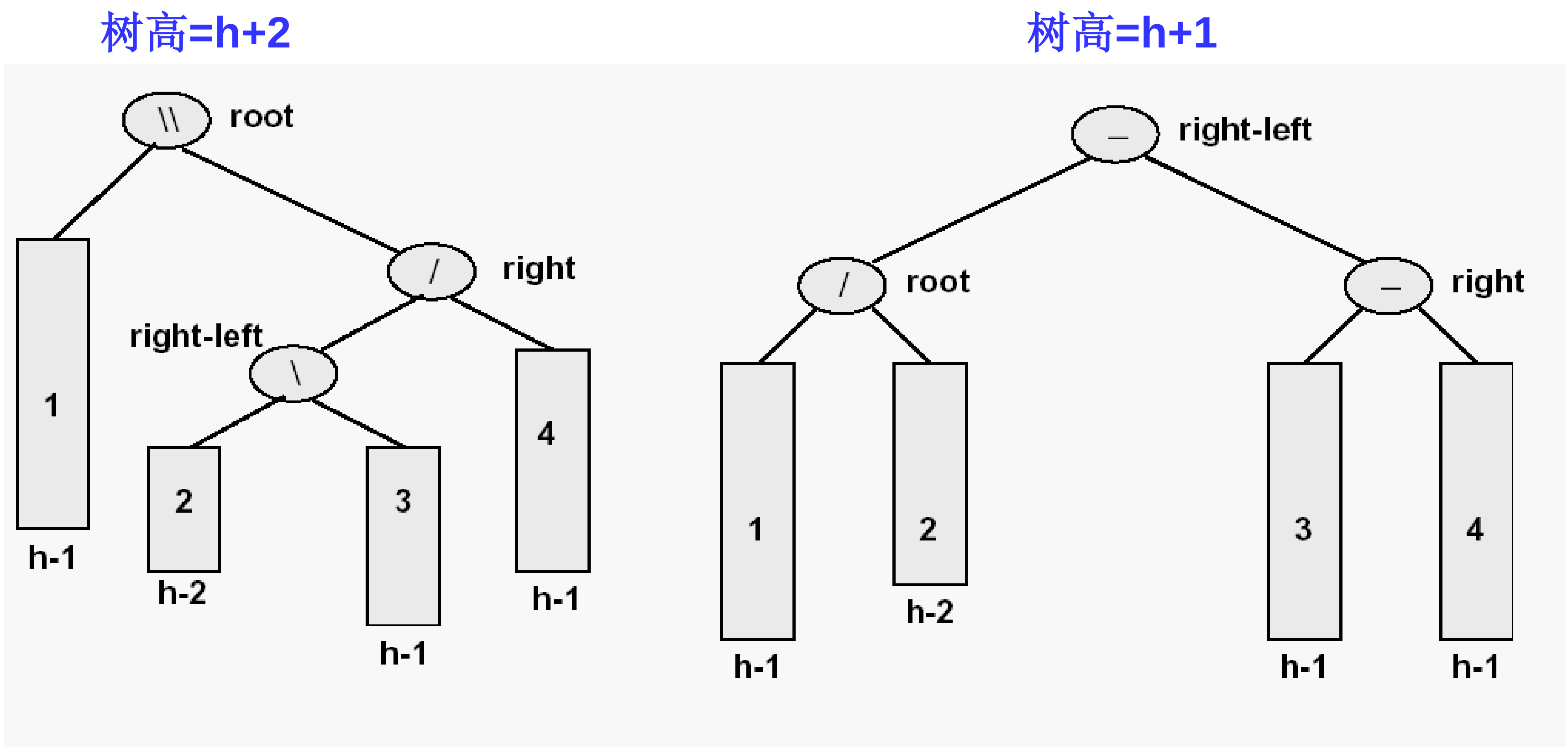
树高=h+2 树高=h+1



modified = TRUE

删除操作: Case 3.3.2

如果right-left子树是右重的，可得到如下结果：



modified = TRUE

AVL 删除: case 3.3.3

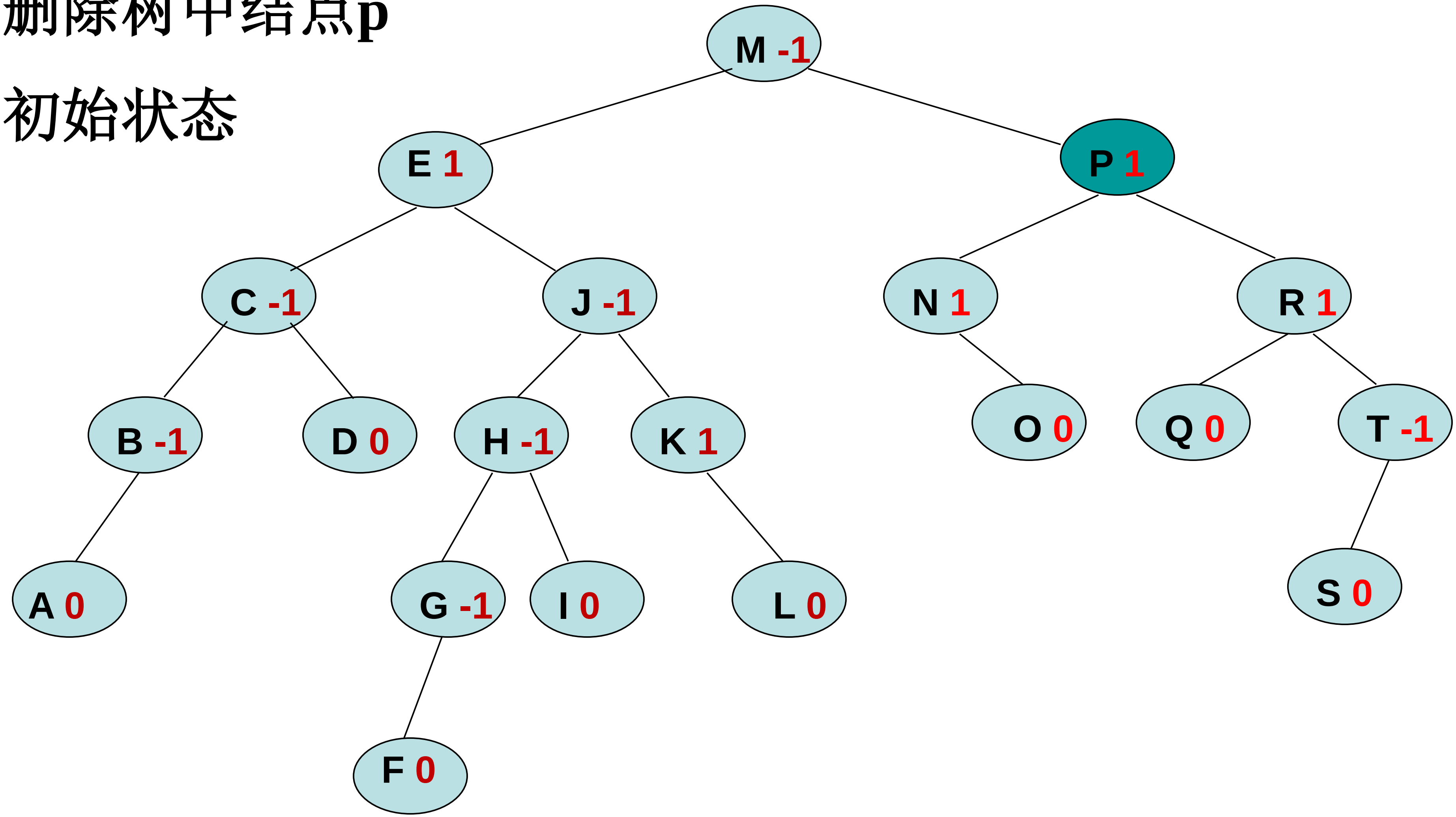
- 如果right-left 子树是平衡的，则可以得到一棵三个结点（即，root, right, right-left）的平衡因子皆为0的子树 *modified = TRUE*

- 前述讨论均假设删除操作发生在左子树上，因为对称的缘故，发生在右子树上时有类似的情况要考虑

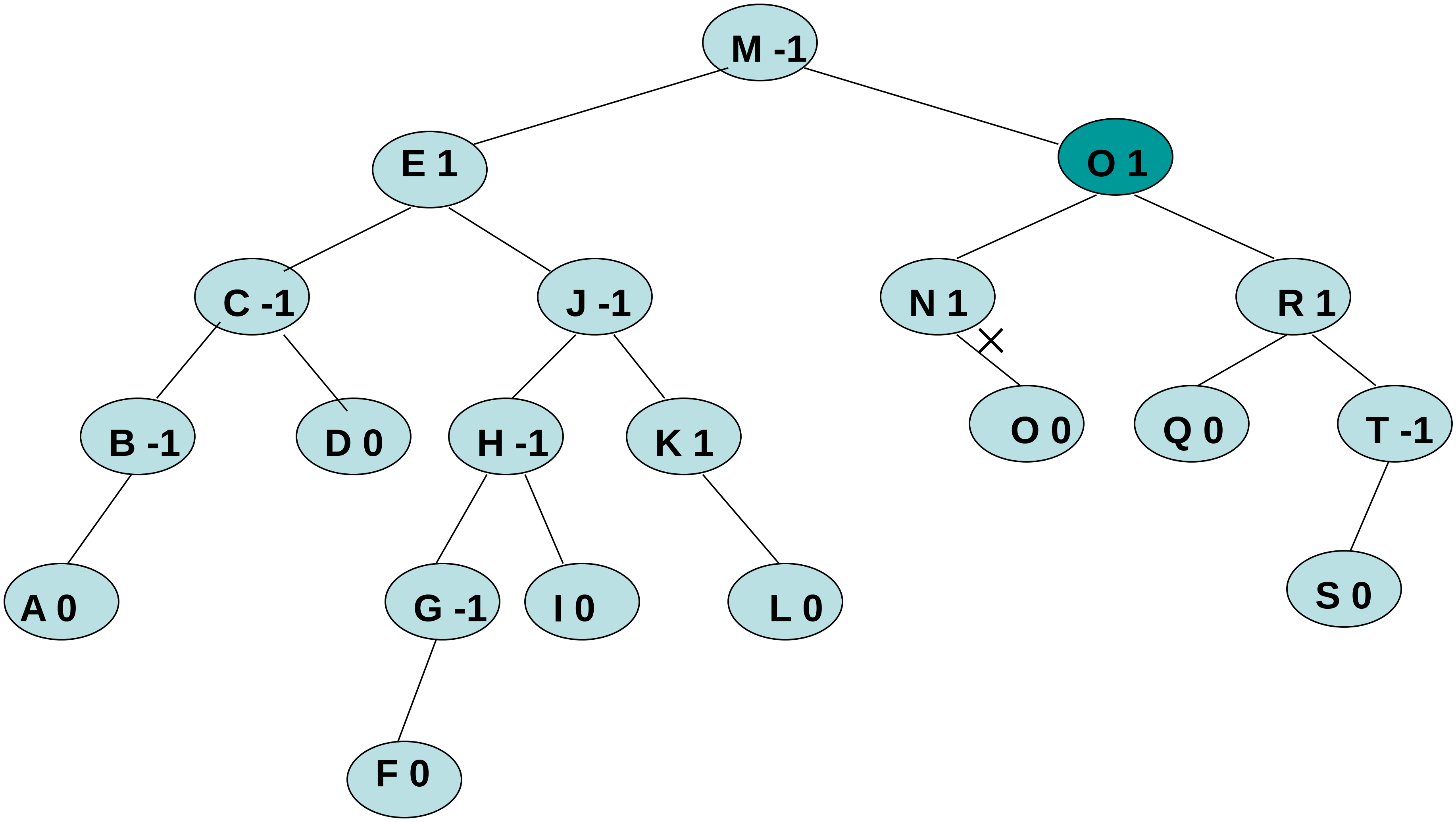
AVL树结点删除的示例

删除树中结点p

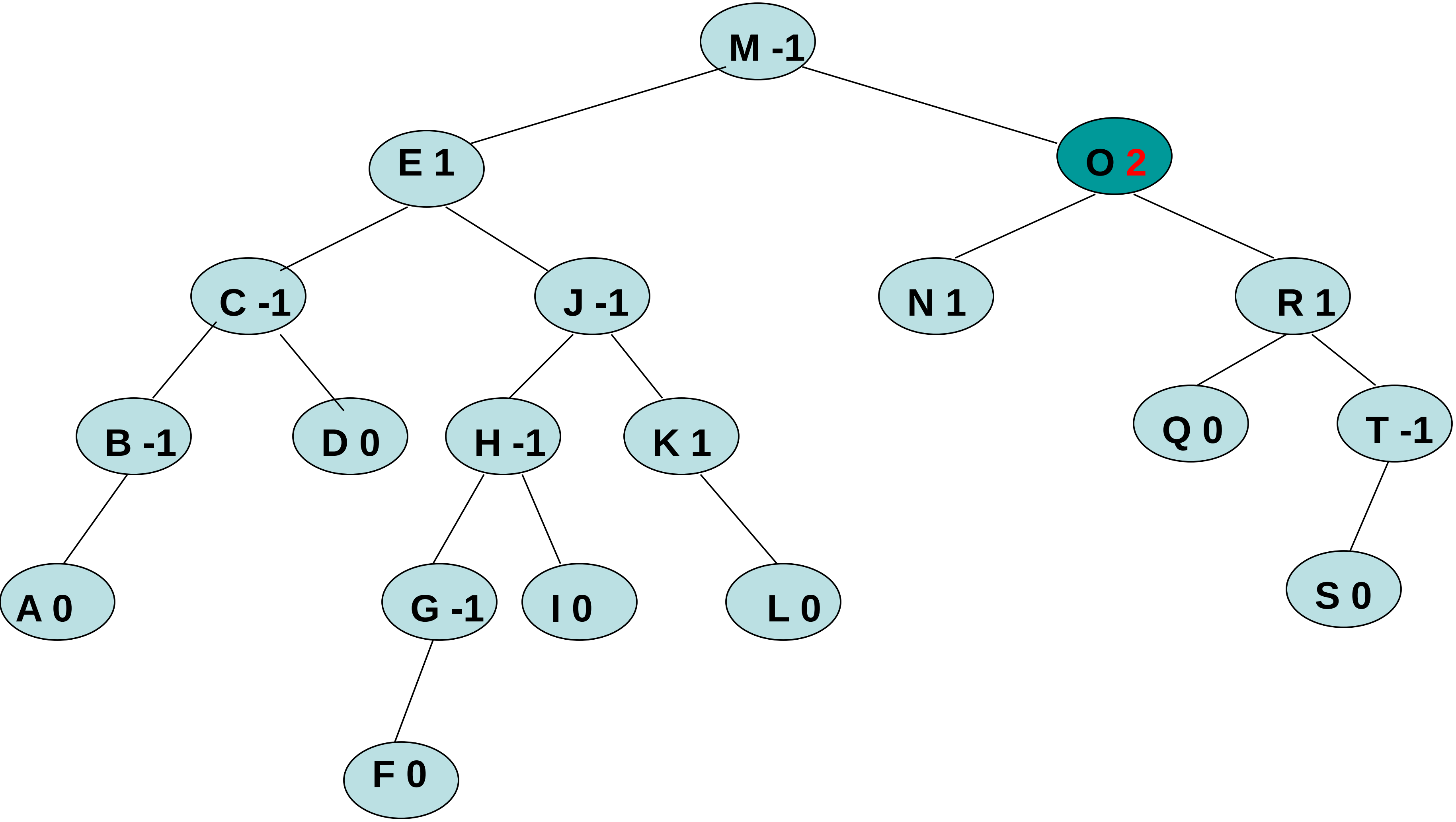
初始状态



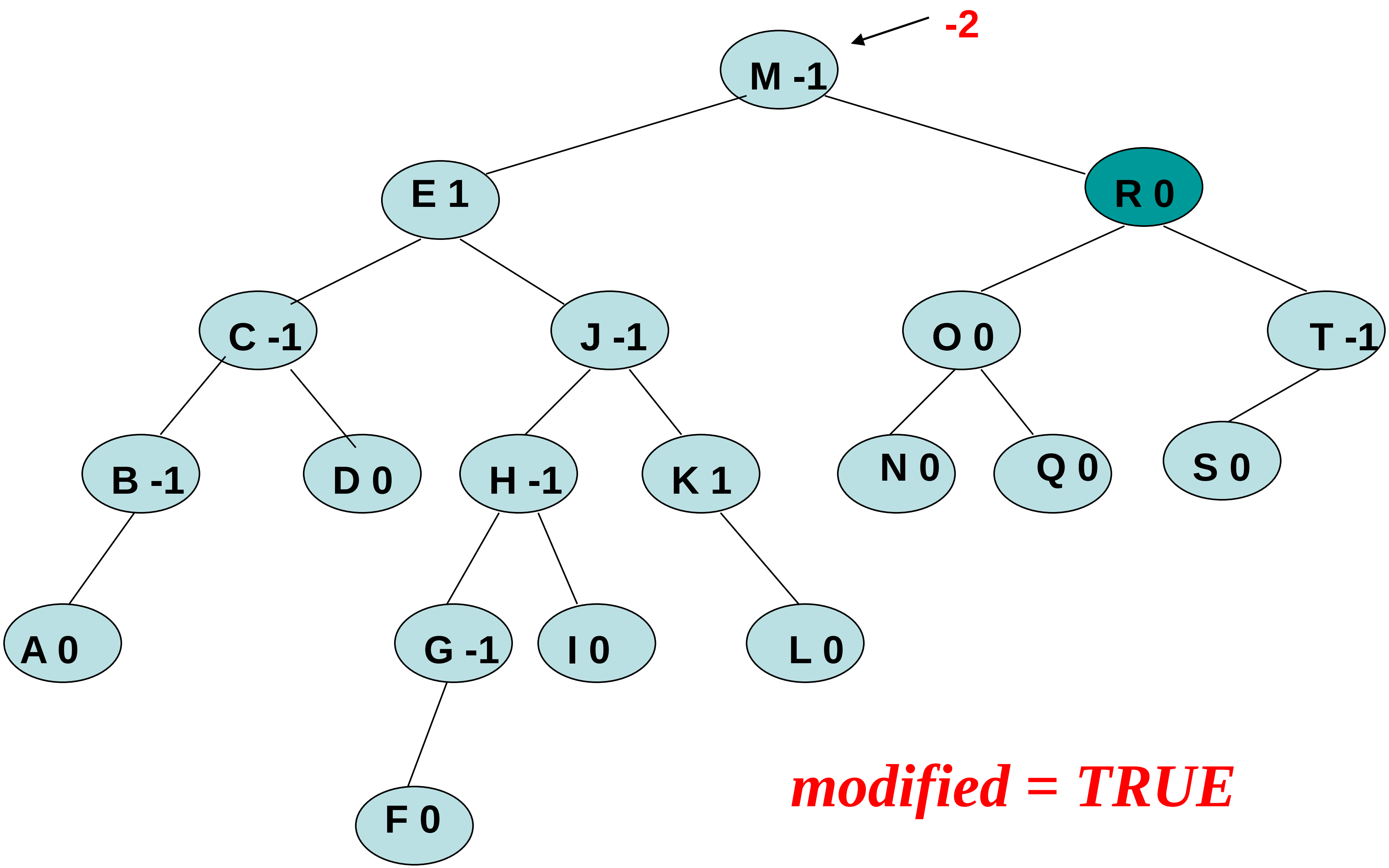
删除树中结点P，用O替换P



删除后状态，继续RR型旋转

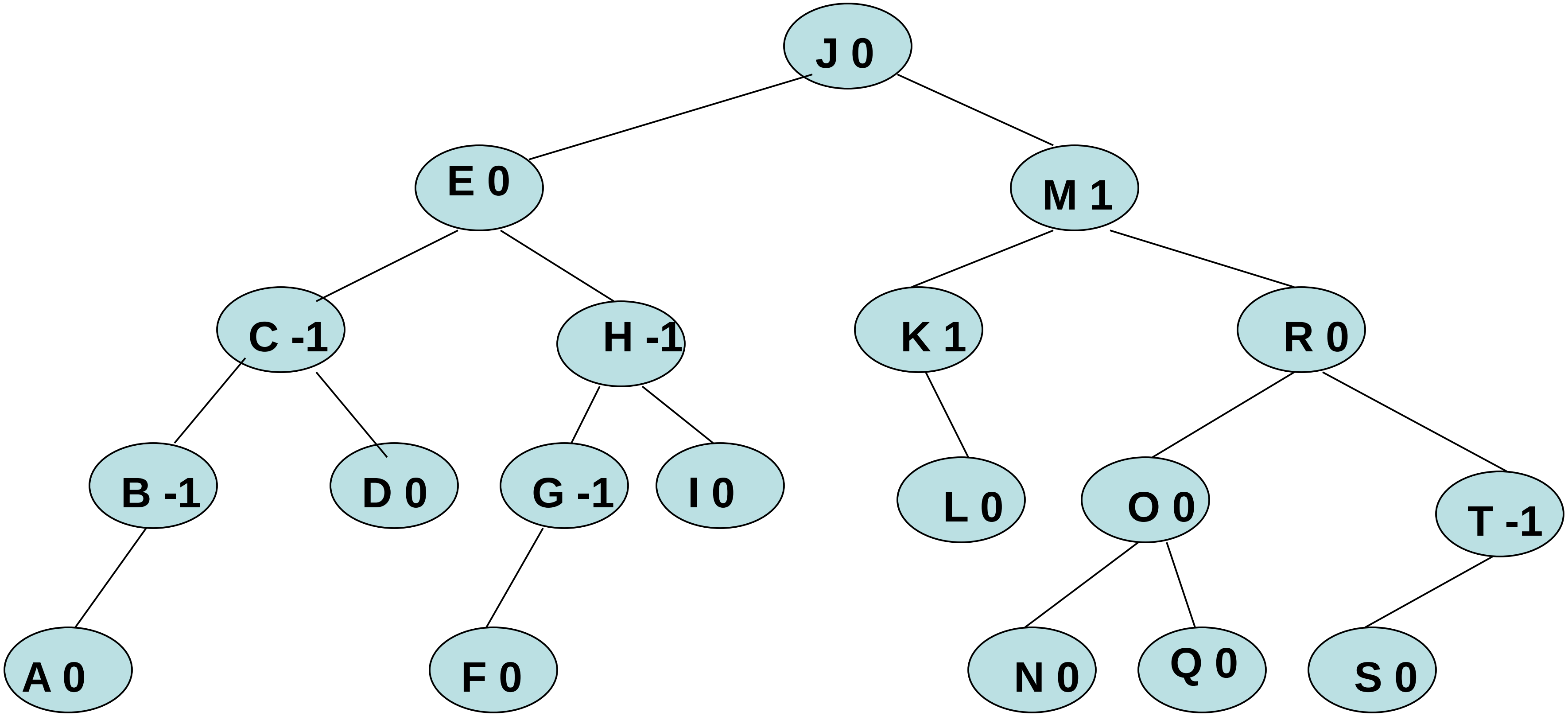


调整后路径上又出现失衡结点



modified = TRUE

LR型旋转调整



AVL Tree的扩展

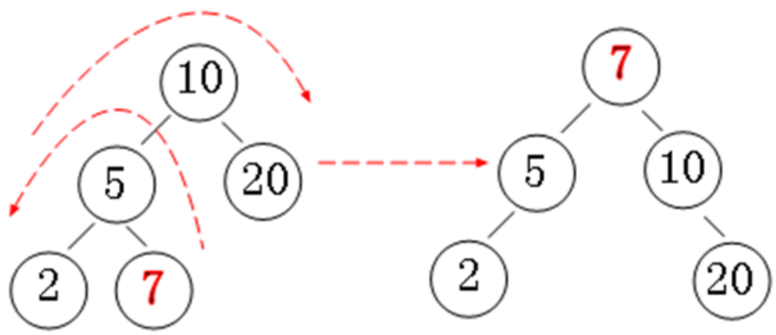
- 允许树的高度差为 Δ >1 (Foster, 1973)，最差情况下，高度随Δ 而增加

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

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

12.4.3 伸展树 (Splaying Tree)

- 数据访问的“二八规则”
  - 80%的人只会用到20%的数据
  - “QQ输入法”，字很多，或许只有20%常用
- 一种自组织数据结构
  - 数据随检索而调整位置
  - 例如：汉字输入法的词表



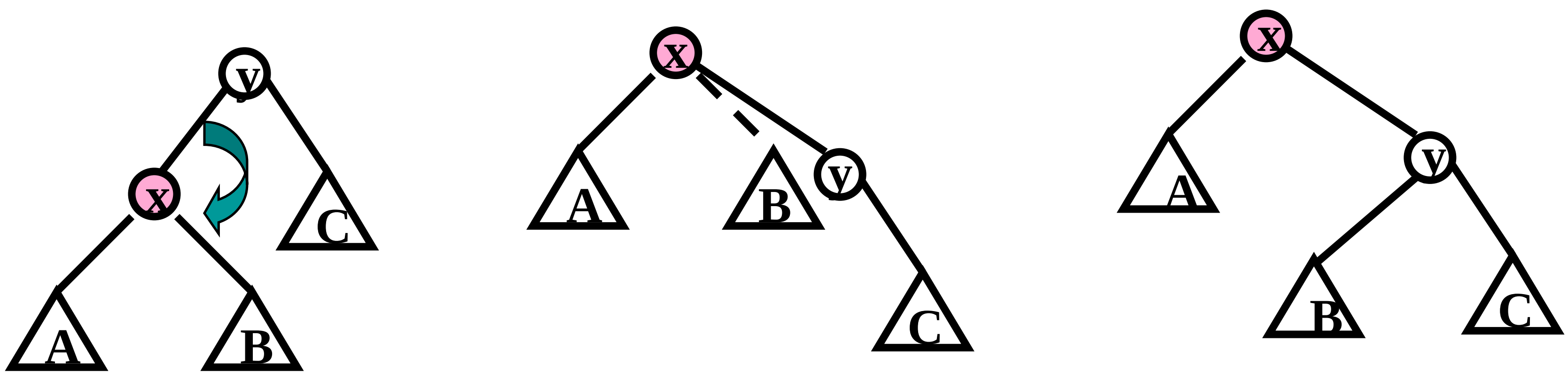
- 伸展树不是一个新数据结构，而只是<u>改进**BST**性能的一组规则</u>
  - 保证访问的总代价不高
  - 不能保证树高平衡

展开(splaying)

- 访问一次结点*x*，完成一次展开过程
  - **插入检索*x*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**
- 结点*x*的展开包括一组旋转
  - 调整**结点*x***、**父结点**、**祖父结点**，把*x*旋到更高层
  - 旋转分：单旋和双旋

单旋转

- 当被访问结点*x*是根结点的子结点时，单旋转
  - 伸展树的单旋转与AVL树的单旋转是一样的
  - 结点提升一层

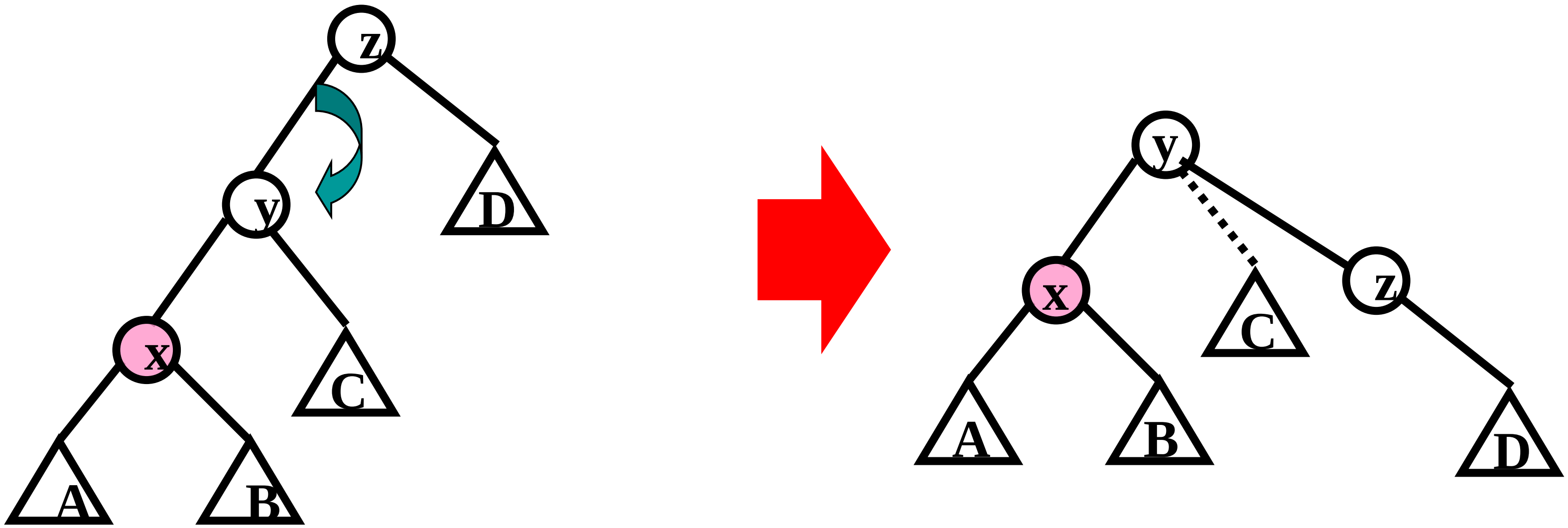


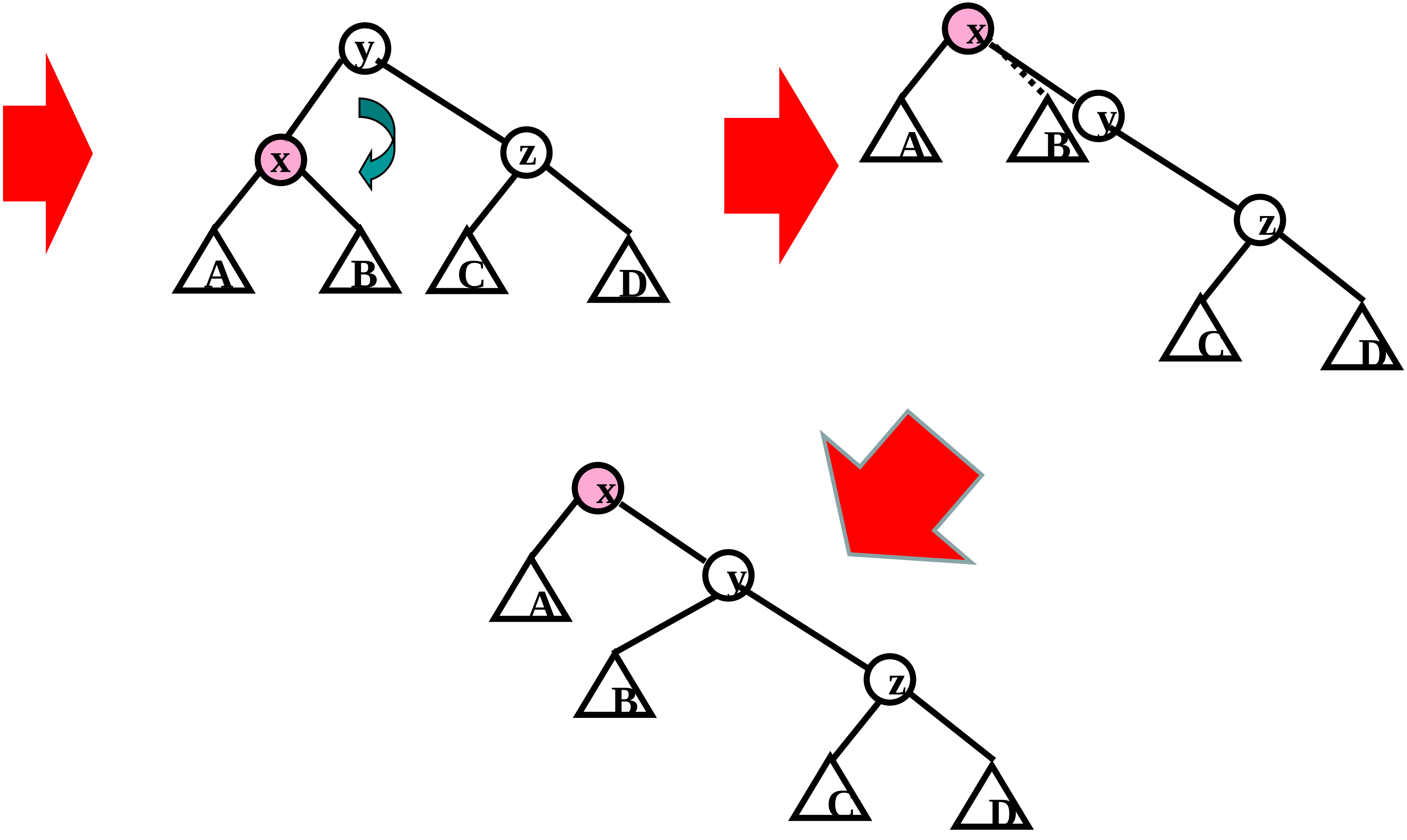
双旋转

- 伸展树需要两种类型的双旋转
  - 一字形旋转，也称为同构调整
  - 之字形旋转，也称为异构调整
- 双旋转涉及
  - **结点*x*、*x*的父结点 *y*、*x*的祖父结点*z***
  - 把结点*x*在树结构中向上移两层

一字形旋转

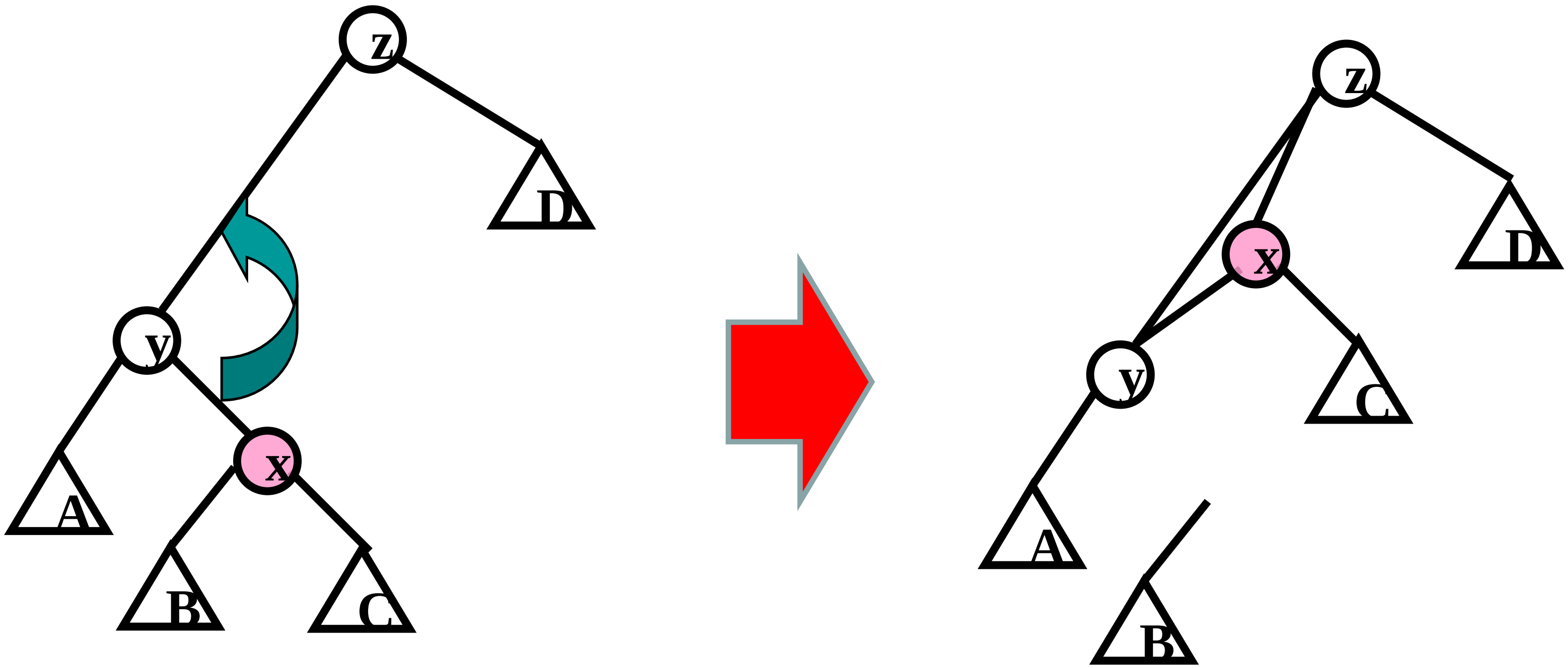
- *x*、*y*、*z*是一顺的
  - 左顺：*x*是*y*的左子结点，*y*是z的左子结点
  - 右顺：*x*是*y*的右子结点，*y*是z的右子结点

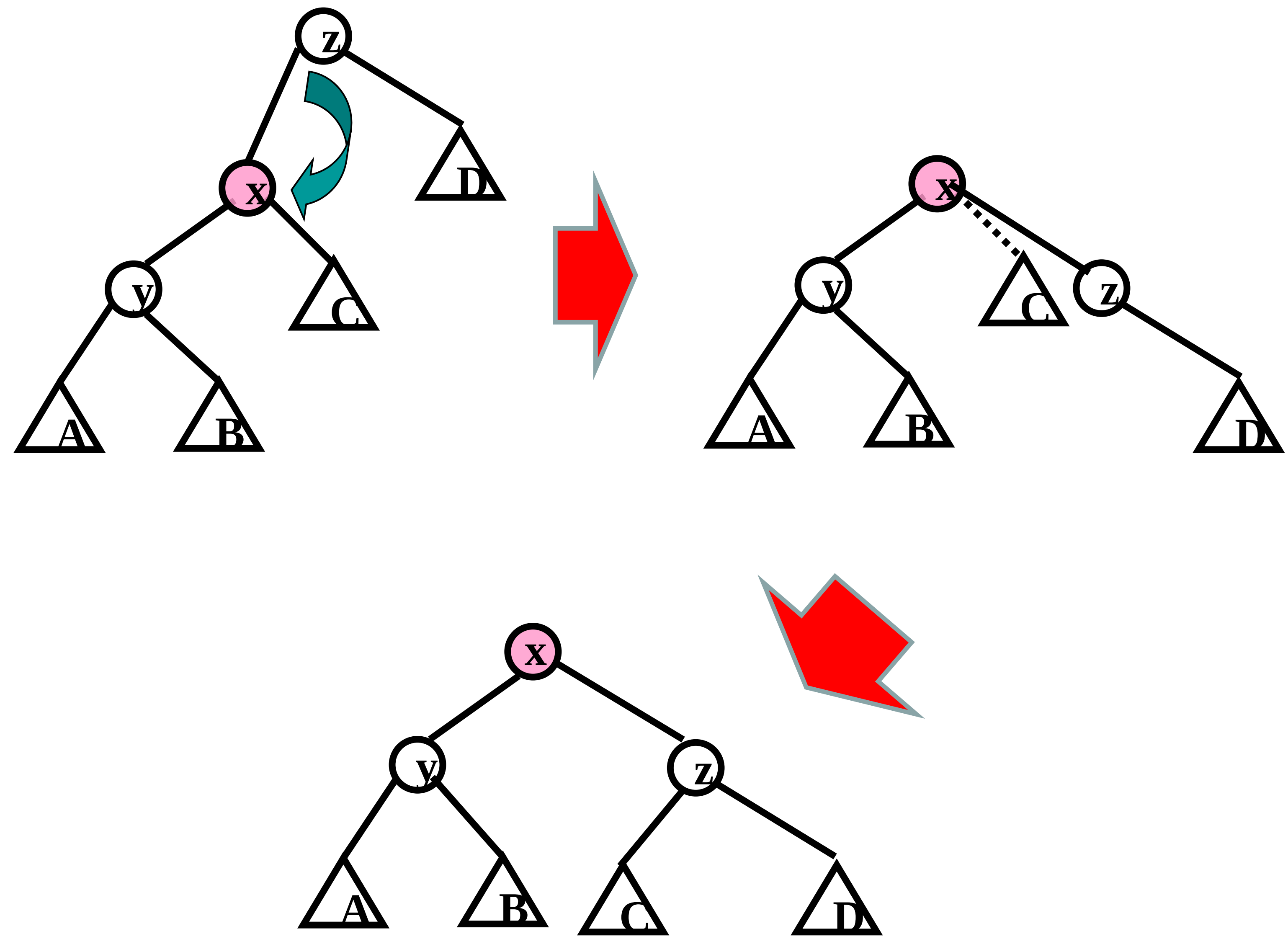



y
x
z
A
B
C
D
x
y
A
B
z
C
D
x
A
y
B
z
C
D

之字形旋转

- 出现以下两种情况之一时，会发生之字形旋转：
  - *x*是*y*的左子结点，*y*是*z*的右子结点
  - *x*是*y*的右子结点，*y*是*z*的左子结点




z
x
y
A
B
C
D
x
y
C
z
A
B
D
x
y
z
A
B
C
D

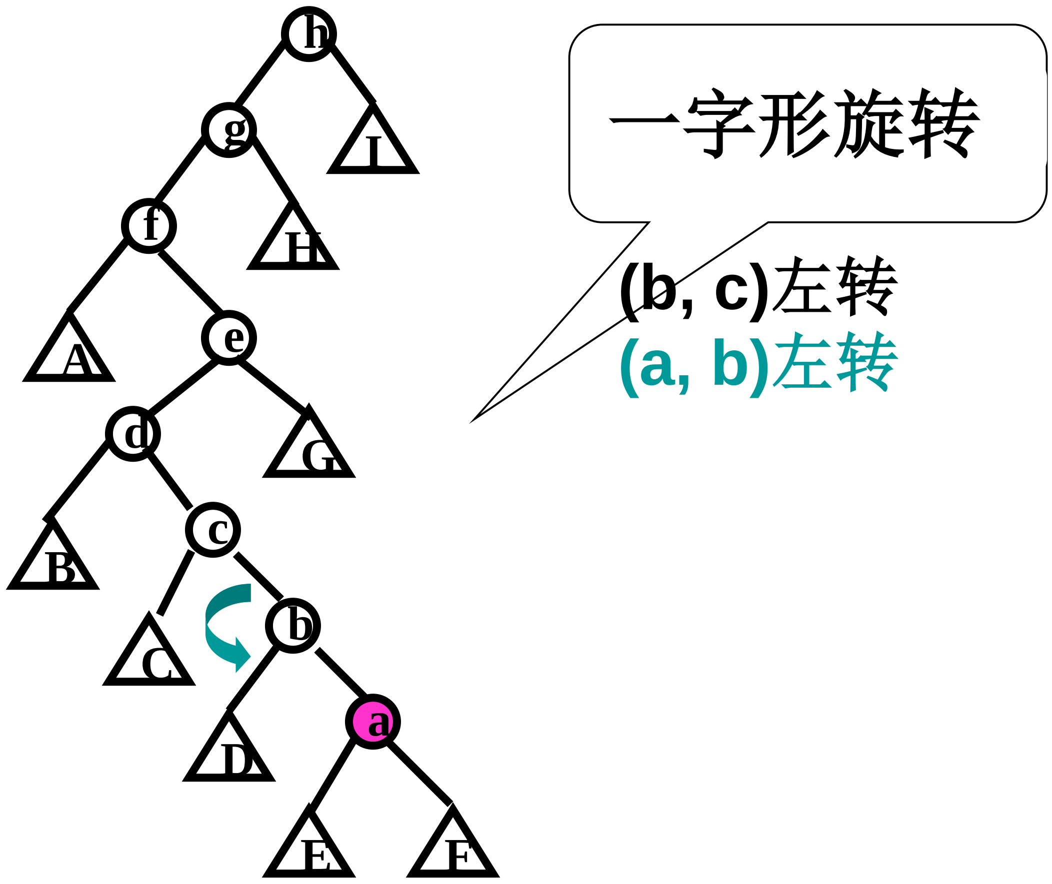
两种旋转的不同作用

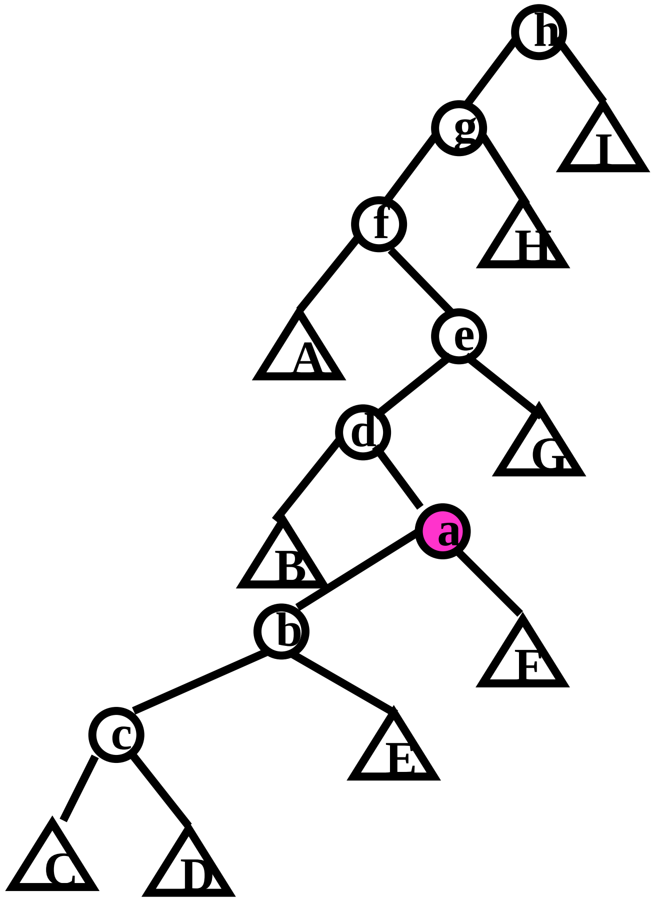
- **之字形旋转**
  - 把新访问的记录向根结点移动
  - 使子树结构的高度减1
  - 趋向于使树结构更加平衡
- **一字形提升**
  - 一般不会降低树结构的高度
  - 只是把新访问的记录向根结点移动

伸展树的调整过程

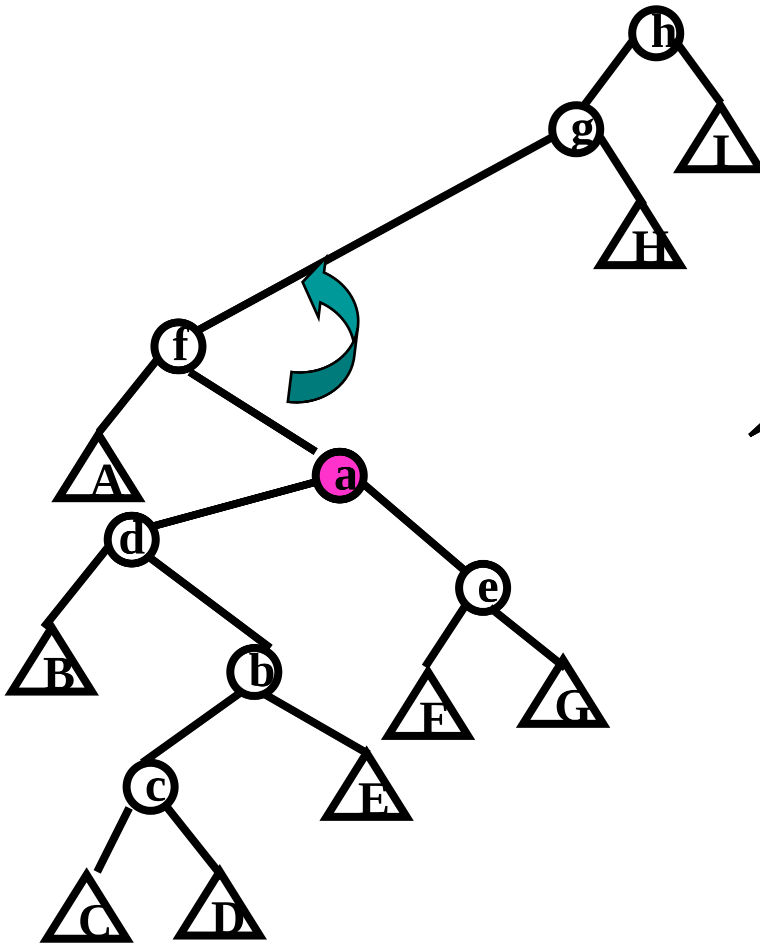
- 一系列双旋转
  - 直到结点x到达根结点或者根结点的子结点
- 如果结点x到达根结点的子结点
  - 进行一次单旋转使结点x成为根结点
- 这个过程趋向于使树结构重新平衡
  - 使访问最频繁的结点靠近树结构的根层
  - 从而减少访问代价

伸展树的调整过程示例





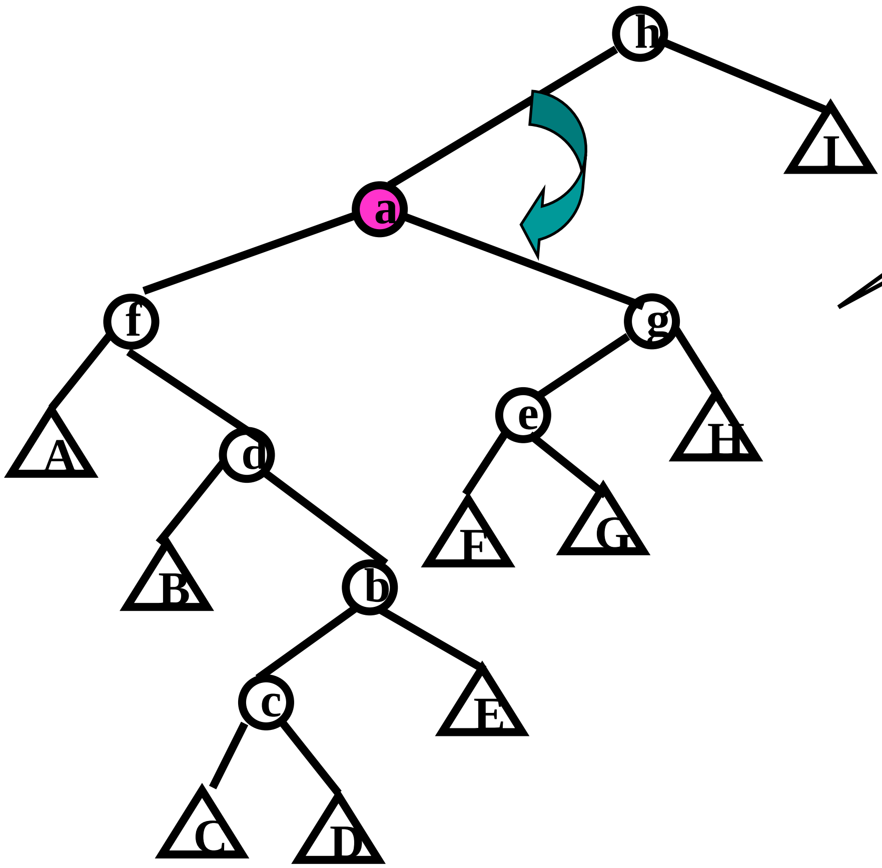
之字形调整



之字形旋转

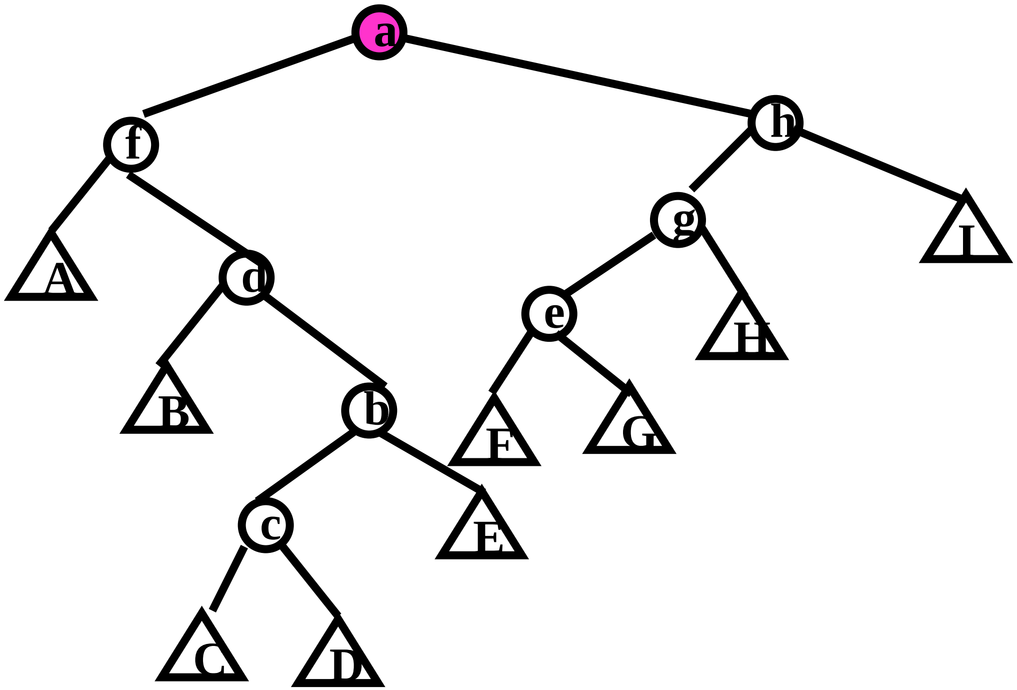
(a, f)左转

(a, g)右转



单旋转

(a, h)右转



(a, h)右转

伸展树的效率

- 伸展树不能保证每单个操作是有效率的
  - 即某次访问操作的代价为O(n)
- 能够保证m次操作总共需要O(mlogn)时间
  - 即每次访问操作的平均代价为O(log n)

几种平衡机制比较

- AVL树要求完全平衡
  - AVL树结构与访问频率无关，只与插入、删除的顺序有关
- 伸展树与操作频率相关
  - 根据插入、删除、检索等动态地调整
- RB-Tree局部平衡
  - 统计性能好于AVL树
  - 增删记录算法性能好、易实现

期末考试

- 时间：2019.01.05 上午

- 地点：

新年快乐！ 考试顺利！ 前程似锦！