



# 第12章

# 高级数据结构

邹磊

北京大学计算机科学技术研究所

`zoulei@pku.edu.cn`

- 多维数组
- 广义表
- Trie结构
- 改进的二叉搜索树

➤ 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

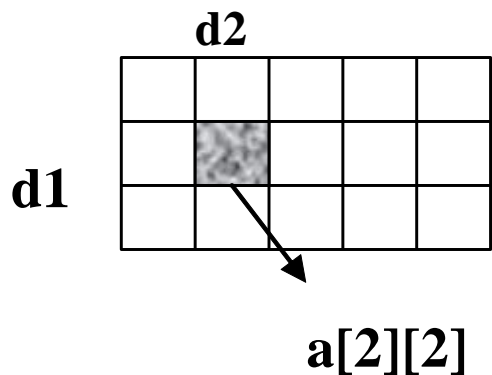
- 向量的向量就组成了多维数组
- 可以表示为：ELEM  $A[c_1..d_1][c_2..d_2]\cdots[c_n..d_n]$ 
  - ✓  $c_i$ 和 $d_i$ 是各维下标的下界和上界。
  - ✓ 所以其元素个数为：

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

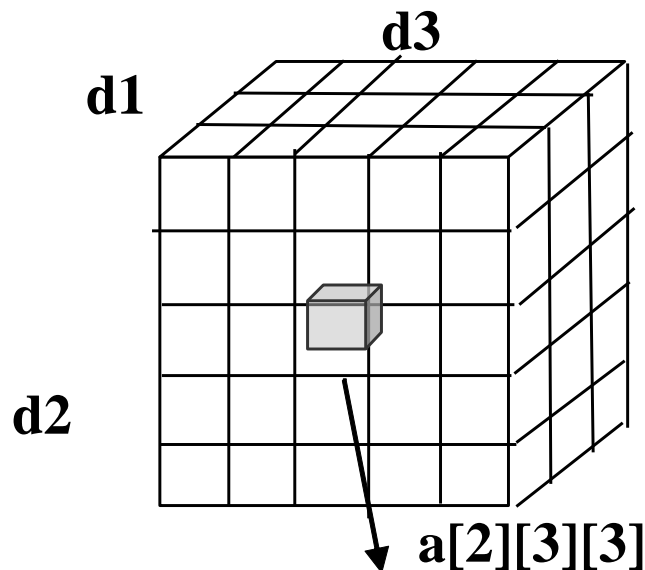
# 数组的空间结构



$d1=3, d2=5, d3=5$



二维数组



三维数组

$d1[1..3], d2[1..5], d3[1..5]$  分别为3个维

- 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)



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

# 下标地址计算基本原理



$A_{i1...in}$ 的起始地址

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



- 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:

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

- 三维数组 $A_{mnp}$ ,  $a_{ijk}$ 的地址计算函数为:

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

# C语言下标地址的计算



► C++多维数组**ELEM**  $A[d_0][d_1] \dots [d_{n-1}]$ ;

$$loc(A[j_0, \dots, j_{n-1}]) = loc(A[0, \dots, 0])$$

$$\begin{aligned} &+ d * [j_0 * d_1 * \dots * d_{n-1} \\ &\quad + j_1 * d_2 * \dots * d_{n-1} + \dots \\ &\quad + j_{n-2} * d_{n-1} \\ &\quad + j_{n-1}] \end{aligned}$$

$$= loc(A[0, \dots, 0]) + d * \left[ \sum_{i=0}^{n-2} j_i \prod_{k=i+1}^{n-1} d_k + j_{n-1} \right]$$

- 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

# 常见的特殊矩阵

## 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： $a_{ij}=a_{ji}$   $0 \leq i, j \leq n-1$ ，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

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

## ➤ 分布特征

- ➡ 只有少量非零元素，且非零元素的分布没有规律

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

## ➤ 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

## ➤ 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 $A_{ij}$ 在一维数组中的位置？

$$Loc(A_{ij}) = \begin{cases} \frac{i \times (i+1)}{2} + j & \text{当 } i \geq j, A_{ij} \text{ 在下三角阵中} \\ \frac{j \times (j+1)}{2} + i & \text{当 } i < j, A_{ij} \text{ 在上三角阵中} \end{cases}$$

(根据  $A_{ij} = A_{ji}$ )



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 $A_{ij}$ 在一维数组中的位置？

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

1
2
3
3
6
5
...
...
4

# 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	$A_{ij}$
0	0	1
0	1	2
0	4	5
1	1	3
2	1	4
3	2	6
4	3	8

➤ 考虑:

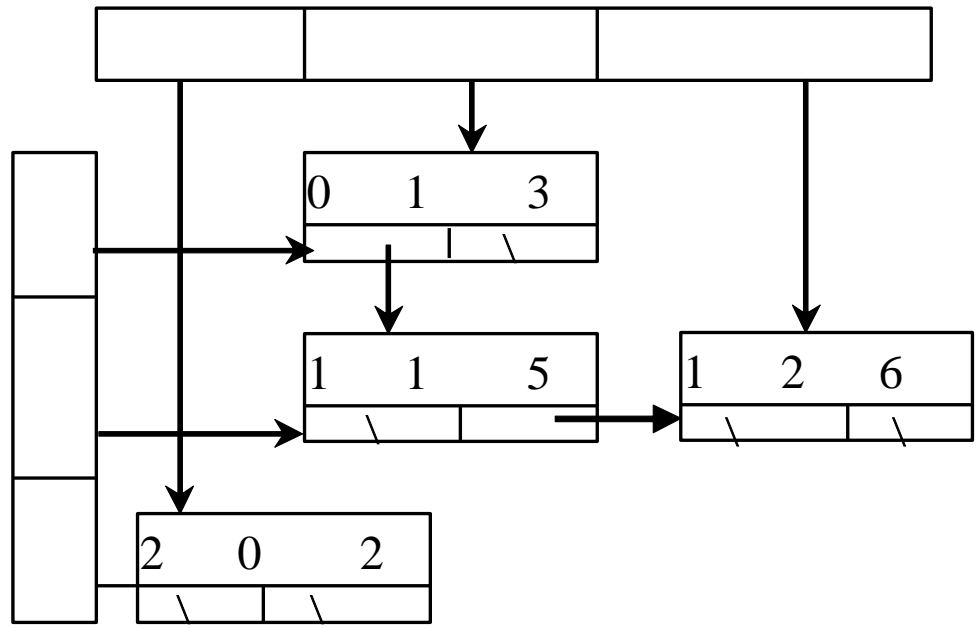
- ➡ 只存非零元素
- ➡ 一个非零元素的必需信息有:  
 $(i, j, a_{ij})$
- ➡ 记录一个稀疏矩阵的必需信息有:  
 行数 $M$ , 列数 $N$ , 非零元素个数 $T$

# 2. 十字链表

i	j	Val/Next
列指针col		行指针row

为链表头指针

行链表头指针



稀疏矩阵的十字链表

# 3. CSR (压缩稀疏矩阵表示)

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

Row Pointer  
Array

0	3	4	5	6
---	---	---	---	---

Column Index  
Array

0	1	4	1	1	2	3
---	---	---	---	---	---	---

Value  
Array

1	2	5	3	4	6	8
---	---	---	---	---	---	---

# 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 $n$  ( $n \geq 0$ ) 个数据元素组成的有限有序序列
- ➡ 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:  $L = (x_0, x_1, \dots, x_i, \dots, x_{n-1})$



$$L = (x_0, x_1, \dots, x_i, \dots, x_{n-1})$$



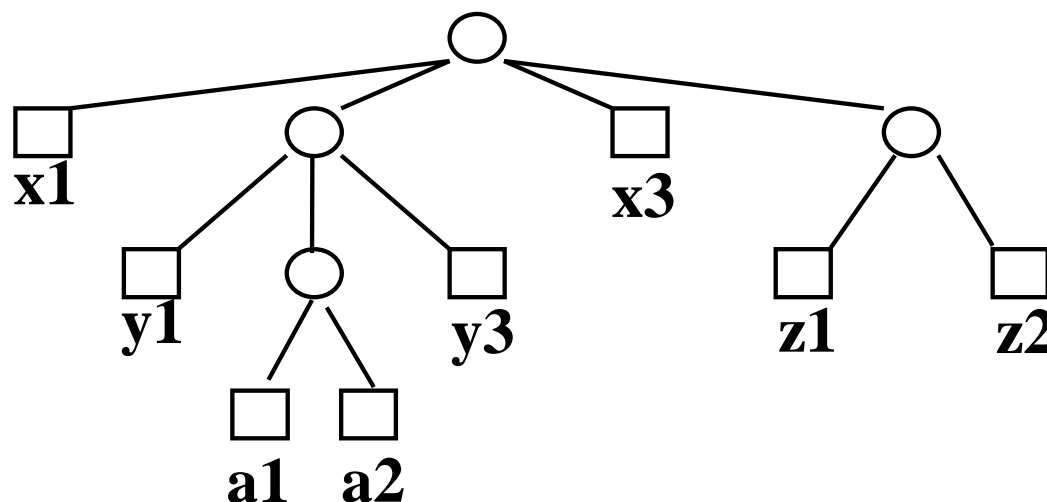
- L是广义表的名称，n为长度
- 每个 $x_i$  ( $0 \leq i \leq n-1$ ) 是L的成员
  - ➡ 可以是单个元素，即原子 (atom)
  - ➡ 也可以是一个广义表，即子表 (sublist)
- 广义表深度：表中元素都化解为原子后的括号层数
- 表头head =  $x_0$ , 表尾tail =  $(x_1, \dots, x_{n-1})$ 
  - ➡ 表头是一个元素，表尾是除了表头的表

## 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))**

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

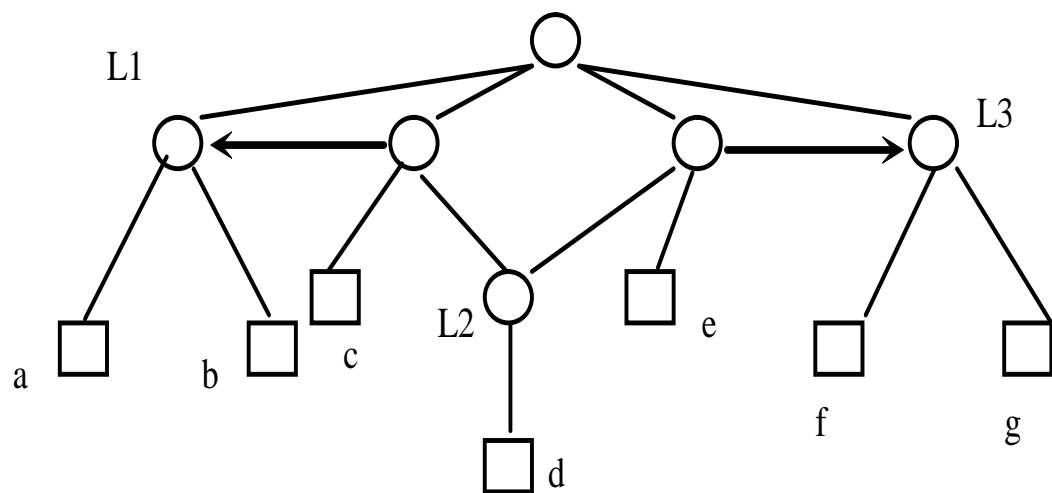
## ➤ 可重入表（再入表）

➡ 其元素(包括原子和子表)可能在表中多次出现

➡ 对应一个DAG

## ➤ 对子表和原子标号

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

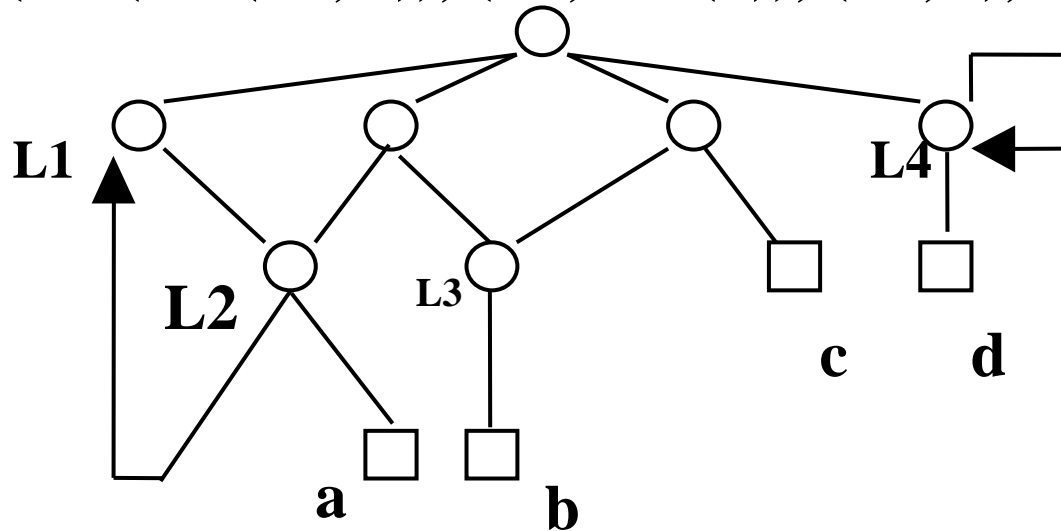


没有标志箭头的边都指向下方

## ➤ 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))**

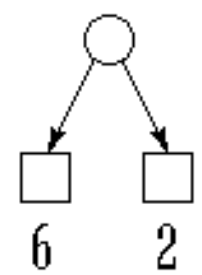


**A**



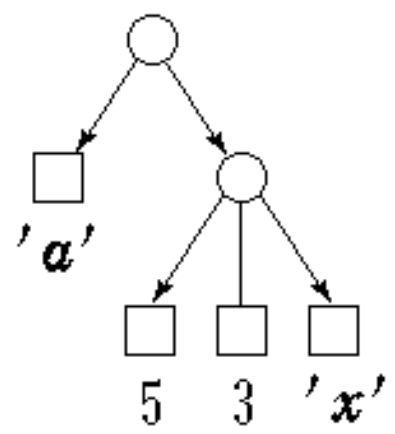
空表

**B**



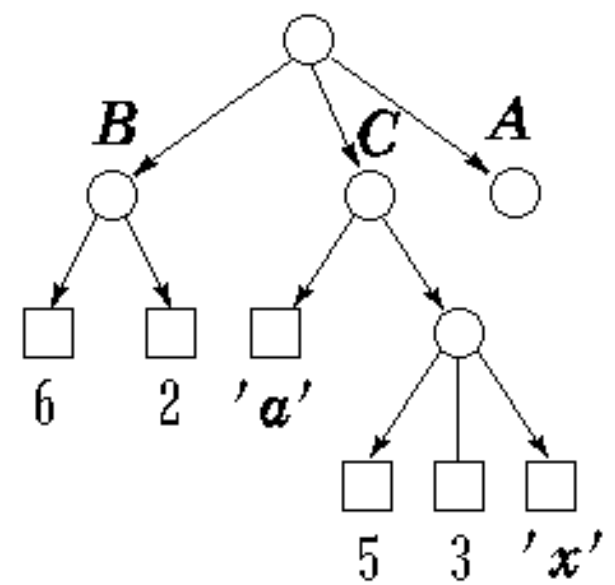
线性表

**C**

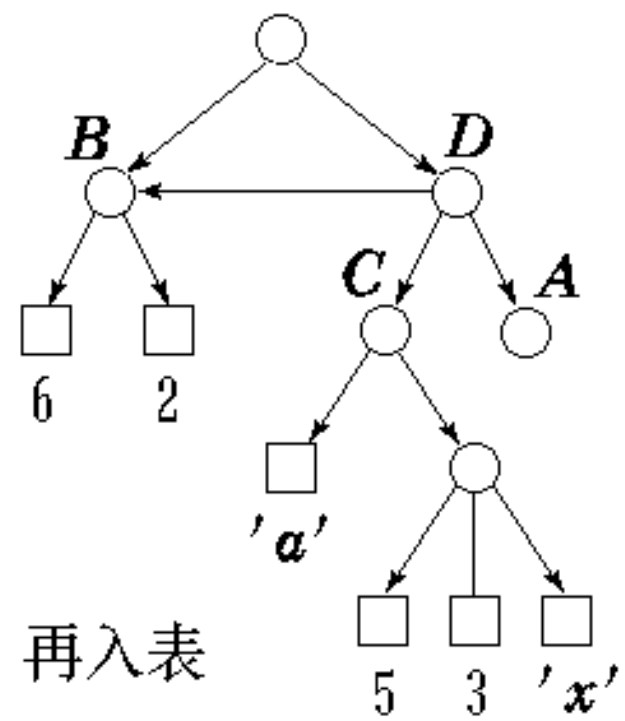


纯表

**D**

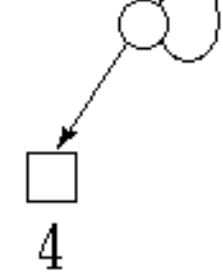


**E**



再入表

**F**



递归表

➤ 图 $\supseteq$ 再入表 $\supseteq$ 纯表(树) $\supseteq$ 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{

    TAG tag;

    union {

        ElemType data;

        GLNode \*sublist;                   // 子表的头指针

    };

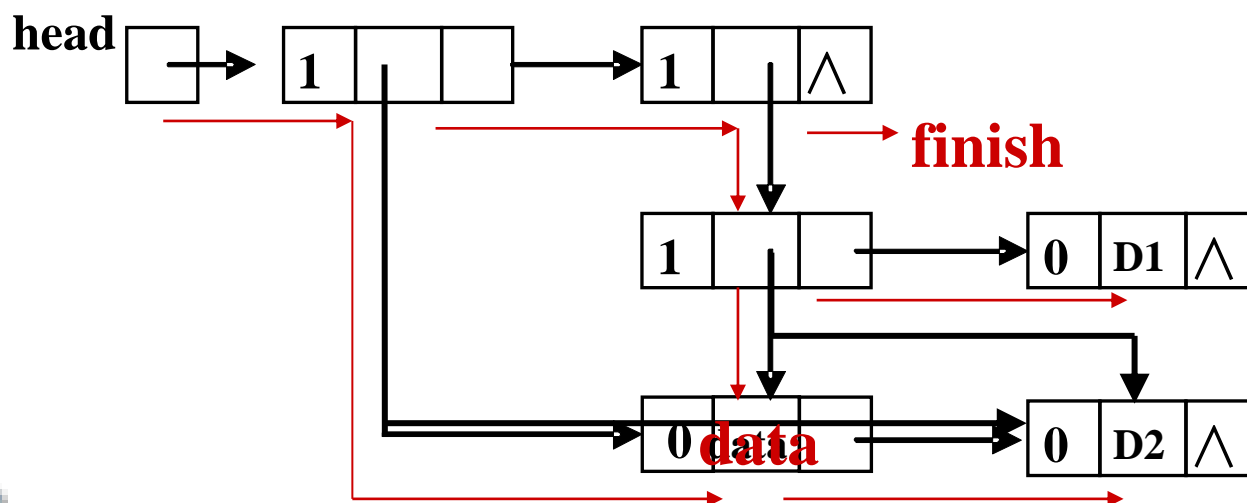
    GLNode \*next;                                 // 后继结点

};

TAG= 0 或 1	data / sublist	next
------------	----------------	------

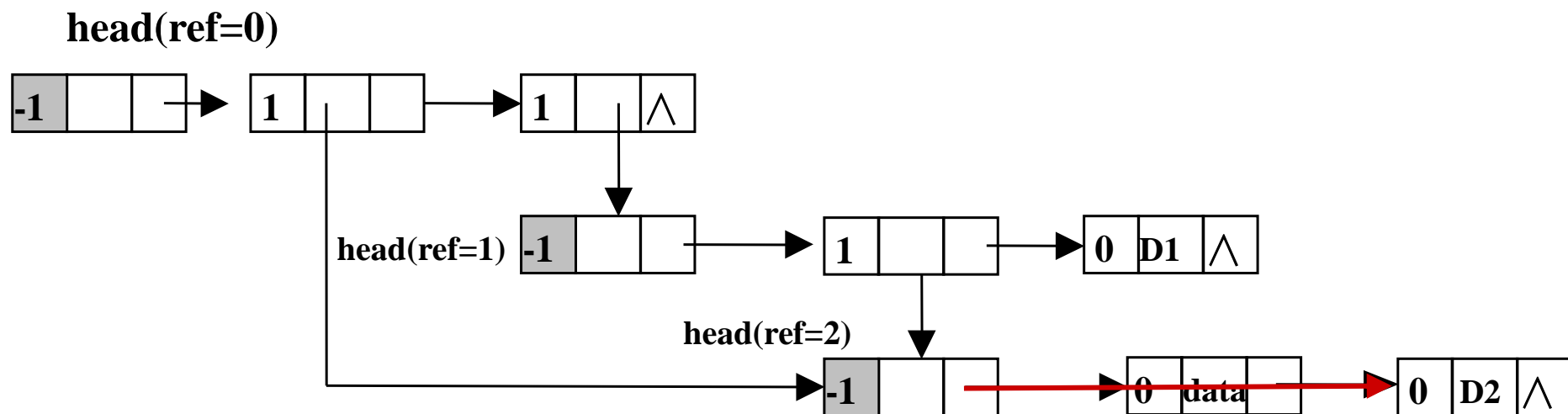
## ➤ 不带头结点的广义表

- ➡ 删除结点时会出现问题
- ➡ 删除结点data必须进行链调整

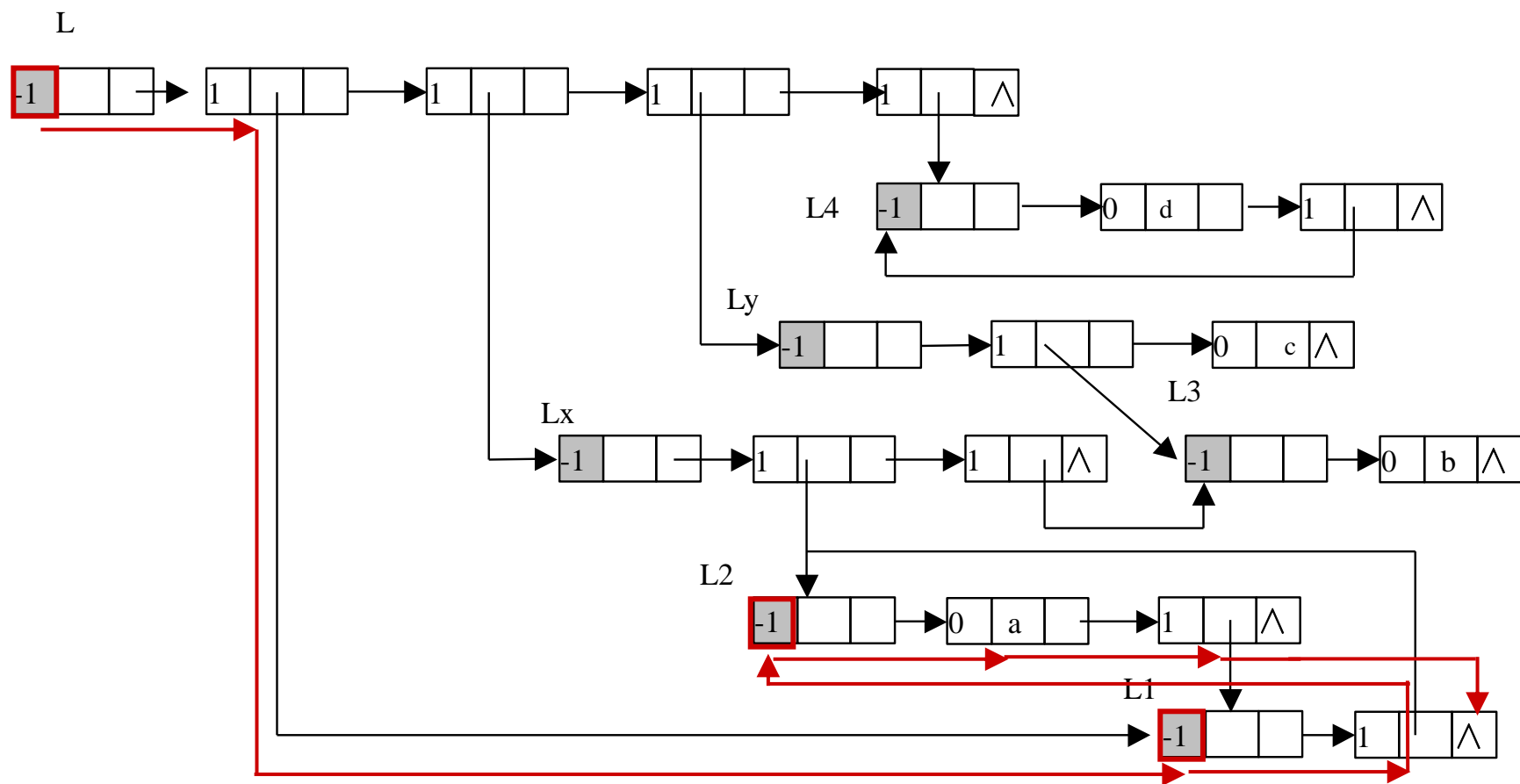




## ➤ 增加头指针，简化删除、插入操作



## ➤ 带表头结点的循环广义表





- 内存管理最基本的问题

- ➡ 分配存储空间

- ➡ 回收被“释放”的存储空间

- 碎片问题

- ➡ 存储的压缩

- 无用单元收集

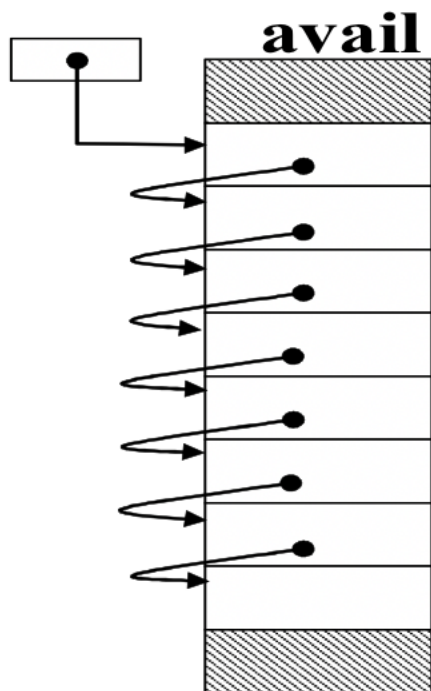
- ➡ 无用单元：可以回收而没有回收的空间

- ➡ 内存泄漏 (memory leak)

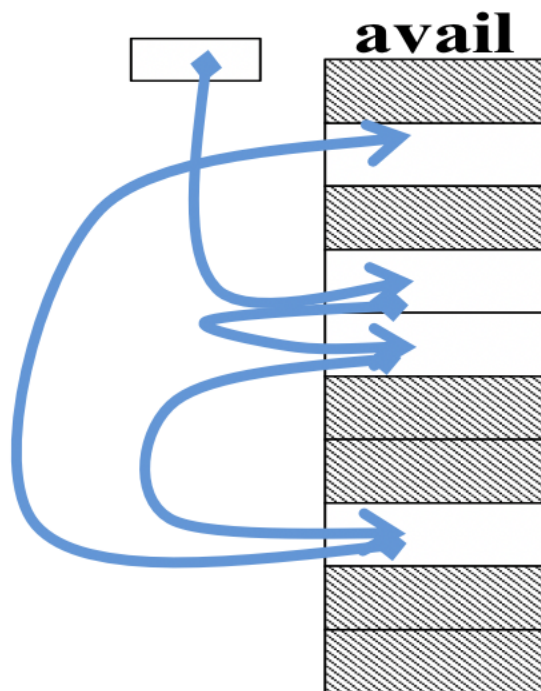
- ✓ 程序员忘记 `delete` 已经不再使用的指针

- 把存储器看成一组定长块组成的数组
  - ➡ 一些块是已分配的
  - ➡ 链接空闲块，形成可利用空间表 (freelist)
- 存储分配和回收
  - ➡ new p 从可利用空间分配
  - ➡ delete p 把 p 指向的数据块返回可利用空间表

# 可利用空间表



(1) 初始状态的可利用空间表



(2) 系统运行一段时间后的  
可利用空间表

结点等长的可利用空间表

# 可利用空间表的函数重载



```
template <class Elem> class LinkNode{  
private:  
    static LinkNode *avail;           // 可利用空间表头指针  
public:  
    Elem value;                       // 结点值  
    LinkNode * next;                  // 指向下一结点的指针  
    LinkNode (const Elem & val, LinkNode * p) ;  
    LinkNode (LinkNode * p = NULL) ;  // 构造函数  
    void * operator new (size_t) ;    // 重载new运算符  
    void operator delete (void * p) ; // 重载delete运算符  
};
```

# 可利用空间表的函数重载



// 重载new运算符实现

```
template <class Elem>
```

```
void * LinkNode<Elem>::operator new (size_t) {
```

```
    if (avail == NULL)                                // 可利用空间表为空
```

```
        return ::new LinkNode;    // 利用系统的new分配空间
```

```
    LinkNode<Elem> * temp = avail;
```

```
                                // 从可利用空间表中分配
```

```
    avail = avail->next;
```

```
    return temp;
```

```
}
```



// 重载delete运算符实现

```
template <class Elem>
```

```
void LinkNode<Elem>::operator delete (void * p) {
```

```
    ( (LinkNode<Elem> *) p) ->next = avail;
```

```
    avail = (LinkNode<Elem> *) p;
```

```
}
```



- new 即栈的删除操作
- delete 即栈的插入操作
- 直接引用系统的 new 和 delete 操作符，需要强制用 “::new p” 和 “::delete p”
  - 例如，程序运行完毕时，把 avail 所占用的空间都交还给系统（真正释放空间）

## 变长可利用块

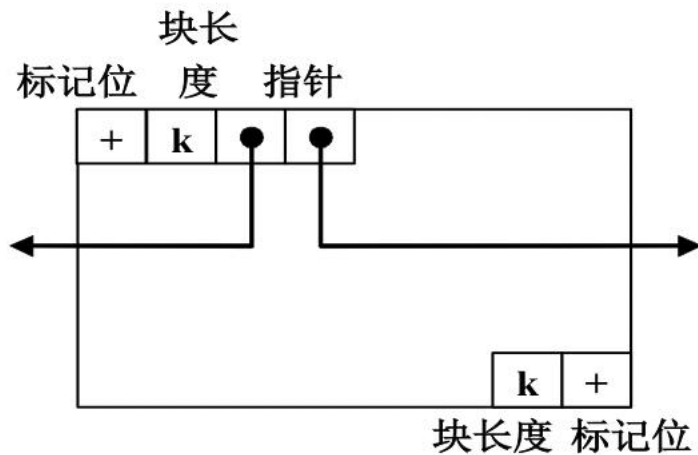
### ➤ 分配

- ➡ 找到其长度大于等于申请长度的结点
- ➡ 从中截取合适的长度

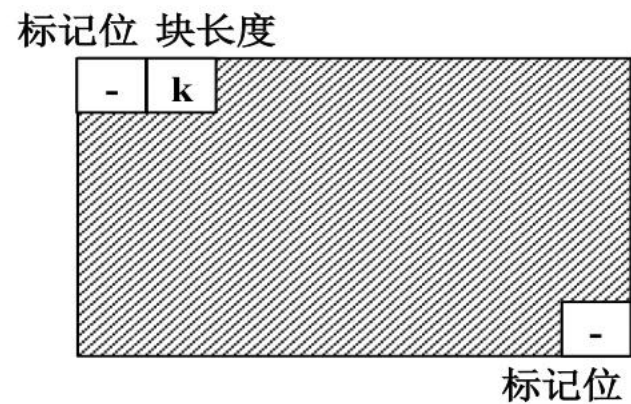
### ➤ 回收

- ➡ 考虑刚刚被删除的结点空间能否与邻接合并
- ➡ 以便能满足后来的较大长度结点的分配请求

# 存储的动态分配和回收



(a) 空闲块的结构



(b) 已分配块的结构

为了分配一个大小为 $N$ 的空闲块，找到长度为 $K$ 的，并且 $K$ 大于等于 $N$ 的空闲块。

第一种分配：将上述整个空闲分配，则会造成“内部碎片”；

第二种分配：将该空闲块中的 $N$ 个字节分配出去，同时将剩下的 $K-N$ 个字节作为新的空闲块，则会造成“外部碎片”

# 空闲块的顺序适配 (sequential fit)

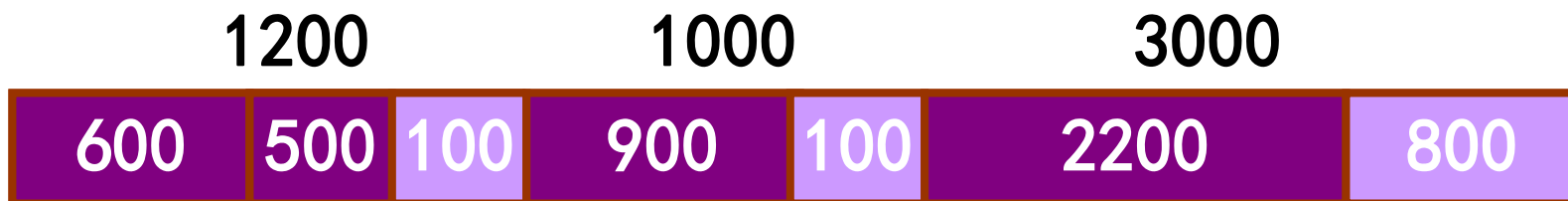


- 首先适配 (first fit)
- 最佳适配 (best fit)
- 最差适配 (worst fit)

# 顺序适配 (sequential fit)



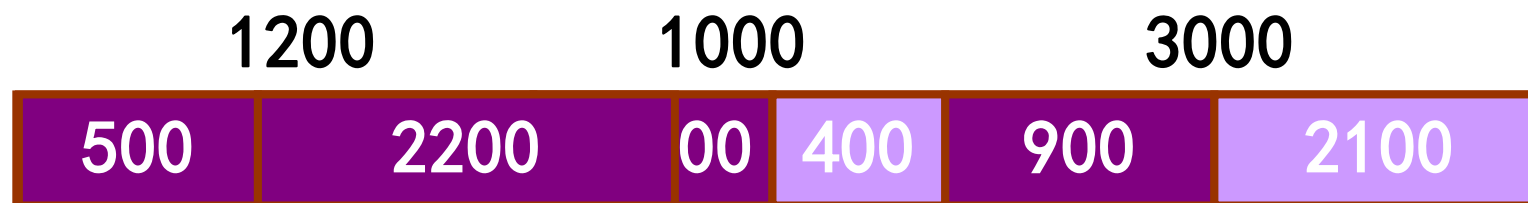
- 首先适配:



➤ 问题: 三个块 1200, 1000, 3000

请求序列: 600, 500, 900, 2200

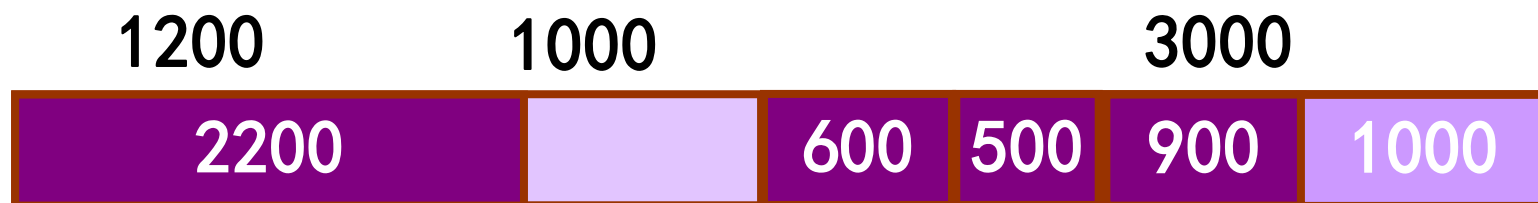
## ➤ 最佳适配



请求序列：600，500，900，2200

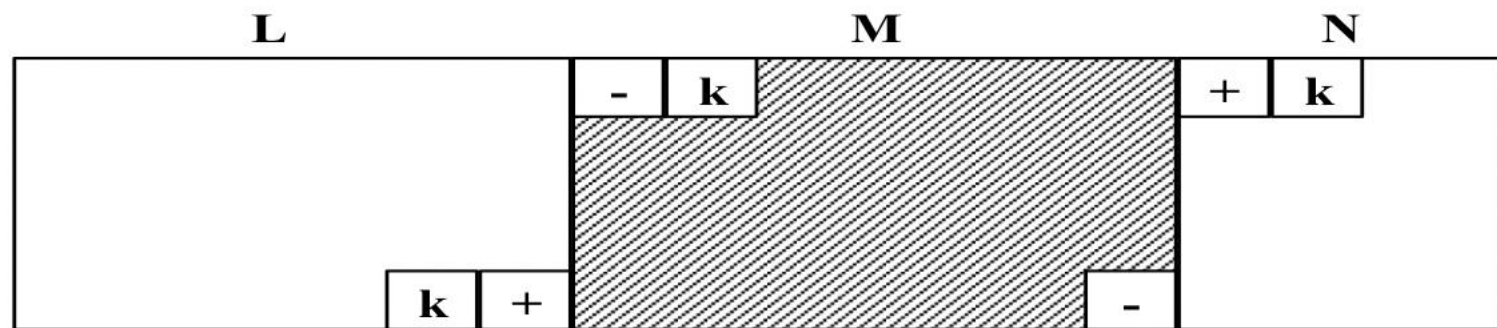


## ➤ 最差适配



请求序列：600，500，900，2200

# 回收：考虑合并相邻块

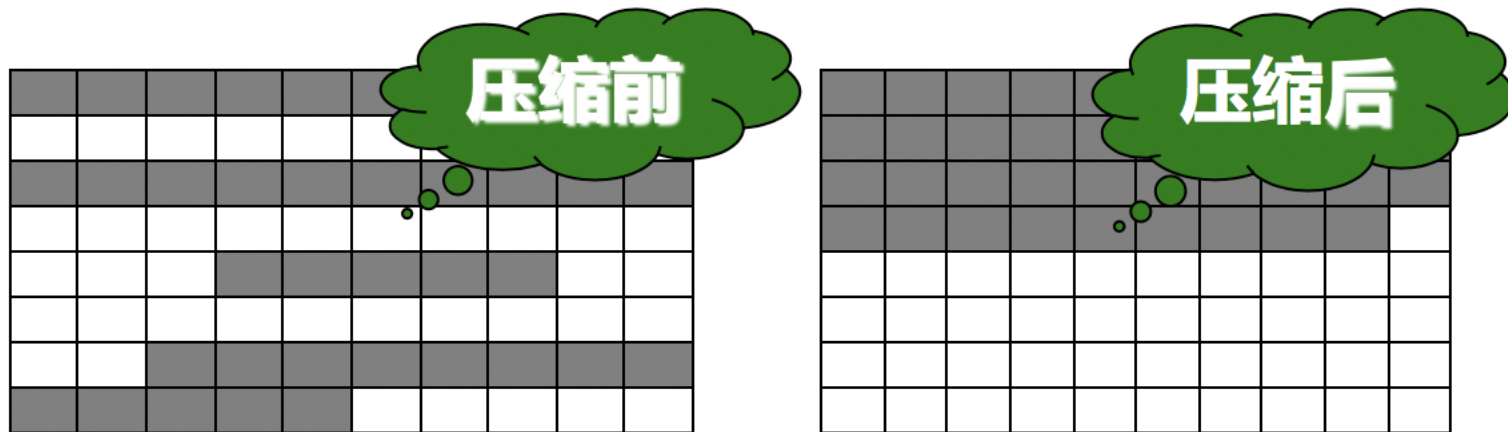


把块 M 释放回可利用空间表

- 如果遇到因内存不足而无法满足一个存储请求，存储管理器可以有两种行为
  - 一是什么都不做，直接返回一个系统错误信息
  - 二是使用失败处理策略（failure policy）来满足请求

# 存储压缩 (compact)

- 在可利用空间不够分配或在进行无用单元的收集时进行“存储压缩”



- 无用单元收集：最彻底的失败处理策略
  - ➡ 普查内存，标记把那些不属于任何链的结点
  - ➡ 将它们收集到可利用空间表中
  - ➡ 回收过程通常还可与存储压缩一起进行

## 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

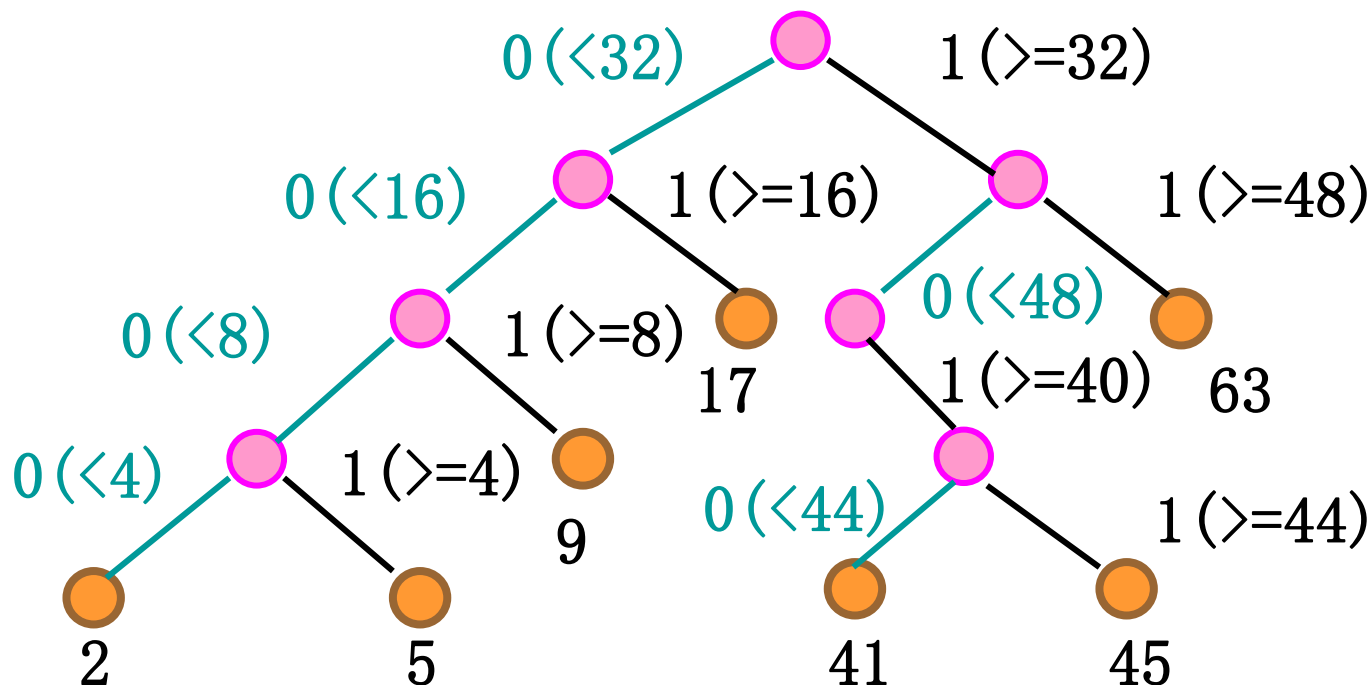
- 关键码空间分解：对关键码范围进行均分
- 基于关键码分解的数据结构，叫做Trie
  - ➡ 又称前缀树或字典树，是一种有序树，键通常是字符串
- Trie树基于两个原则
  - ➡ (1) 关键码集合固定
  - ➡ (2) 对结点分层标记

# 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





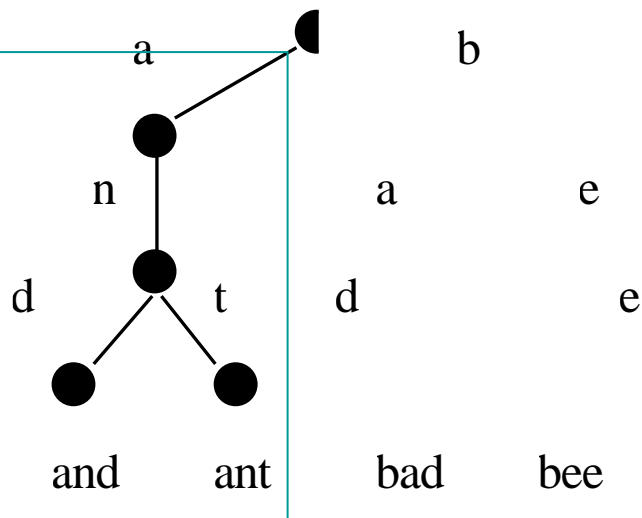
- 根节点不包含字符
- 除根节点外每一个节点都只包含一个字符
- 从根节点到某一节点路径上经过的字符连接起来，  
为该节点对应的字符串
- 每个节点的所有子节点包含的字符都不相同

# 英文字符树——26叉Trie，前缀树



“an”子树代表具有相同前缀an-的关键码集合{and, ant}

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。

- 在trie树中查找一个关键字的时间和树中包含的结点数无关，而取决于组成关键字的字符数。
  - ➡ 二叉查找树的查找时间和树中的结点数有关 $O(\log_2 n)$ 。
- 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

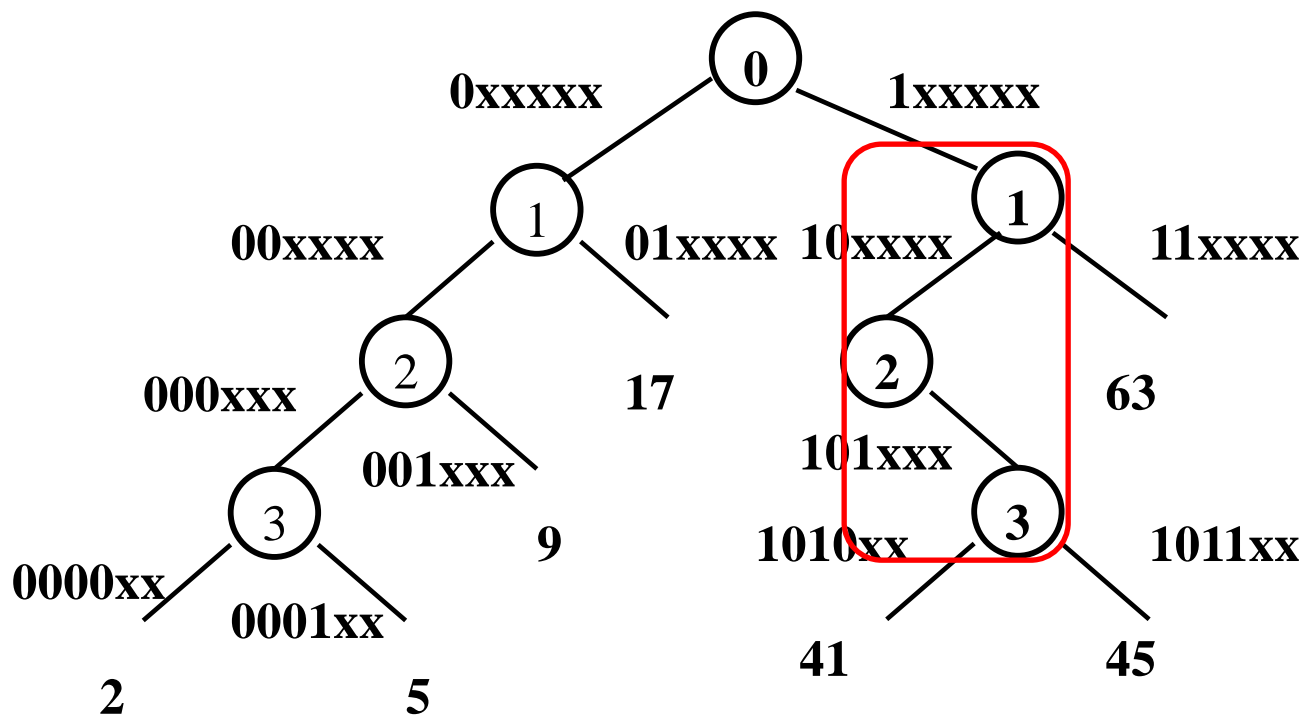
- ➡ 例如，用它存取英文单词时，t子树下的分支比z子树下的分支多出很多（以t开头的单词比以z开头的单词多得多）
- ➡ 且每次有26个分支因子使得树结构过于庞大，不便于检索

## ➤ 应用场景

- ➡ 词频统计、前缀匹配、排序等

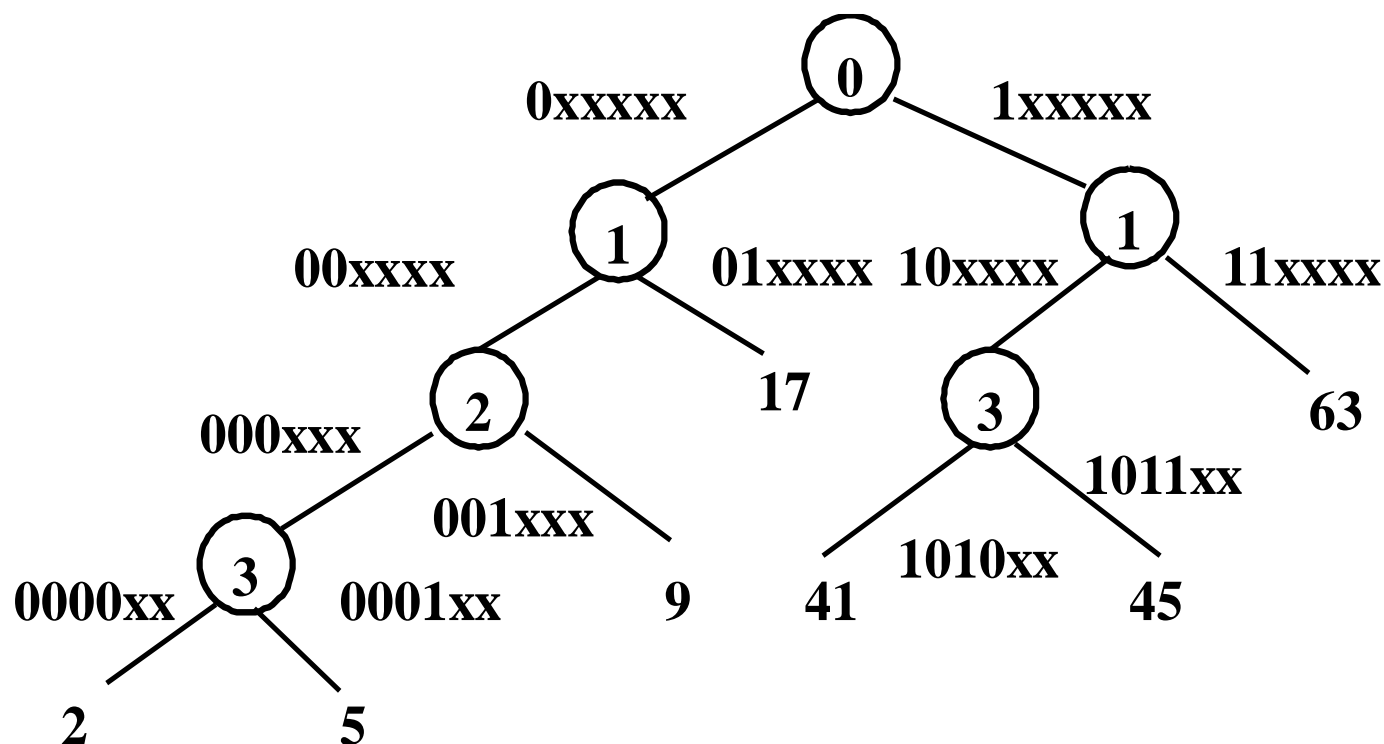
- 简称PAT-tree
- D. Morrison发明的Trie结构变体
  - ➡ 根据关键码二进制位的编码来划分
  - ➡ 二叉Trie树

# 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

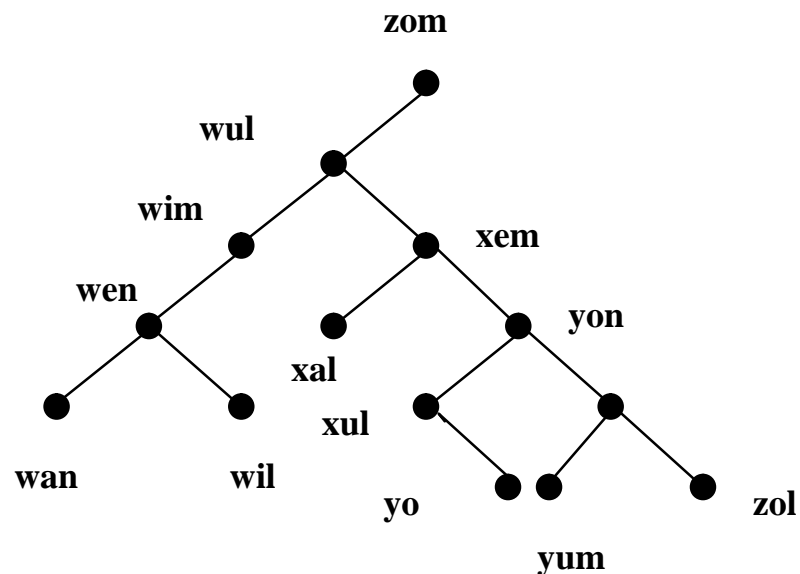
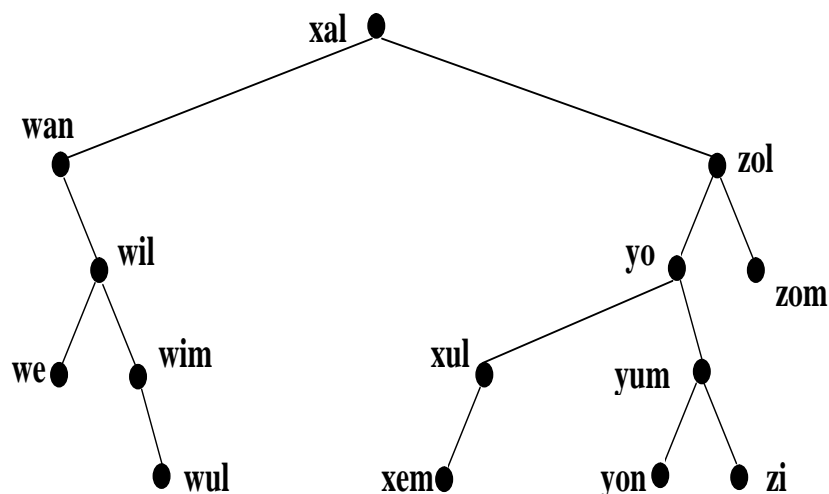


- 最佳BST树—基于用户访问习惯
- 平衡BST树—基于树高平衡约束
- 伸展树—基于用户动态访问特征

# 12.4.1 最佳BST树

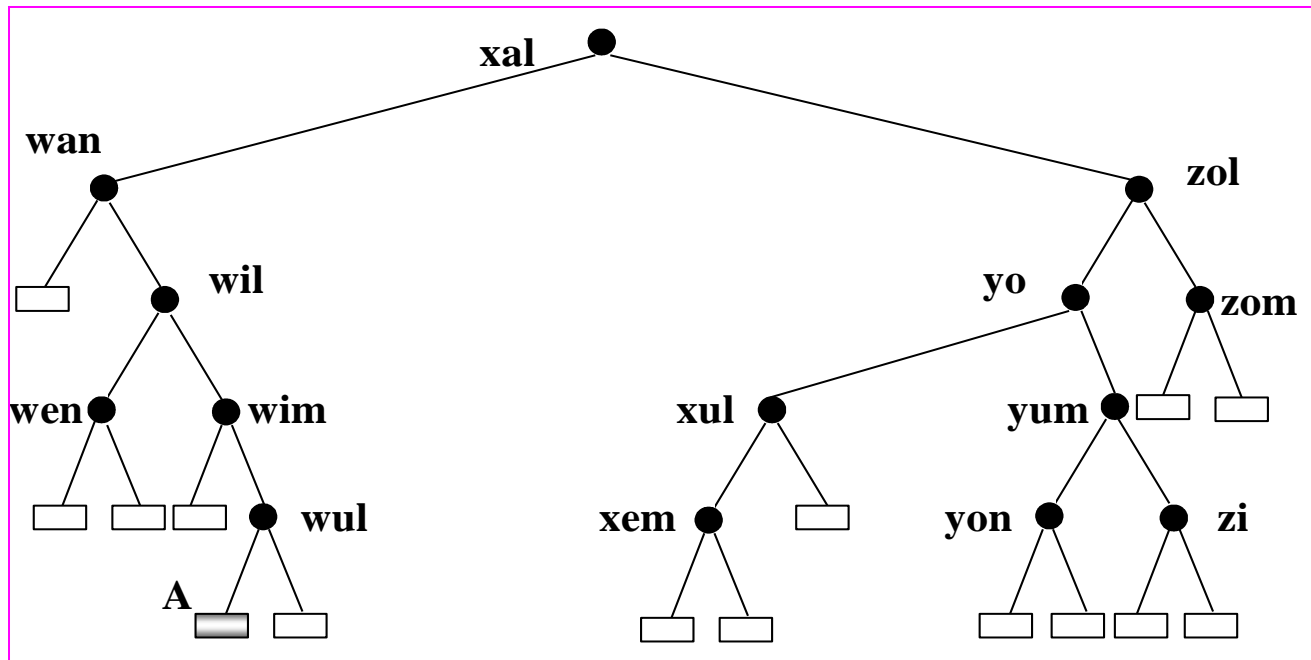


- 同一个关键码集合，插入次序不同，所构成的二叉搜索树也不同



- 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  $C_{2n}^n - C_{2n}^{n-1} = \frac{1}{n+1} C_{2n}^n$   
(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 (根为第0层)

## ➤ 不成功检索

- ➡ 比较次数就等于被检索关键码所属的那个外部结点的层数

## ➤ 平均比较次数

$p_i$ : 检索第  $i$  个内部结点的频率

$q_i$ : 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

➡ 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

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

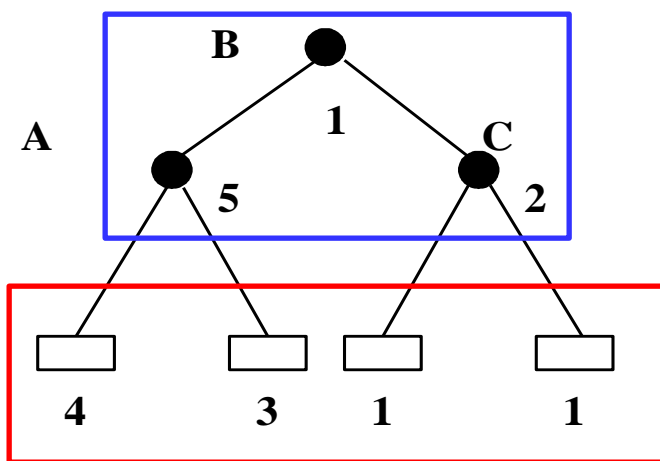
## ➤ 则有

$$\begin{aligned} \text{ASL}(n) &= \frac{1}{2n+1} \left( \sum_{i=1}^n (l_i + 1) + \sum_{i=0}^n l'_i \right) = \frac{1}{2n+1} \left( \sum_{i=1}^n l_i + n + \sum_{i=0}^n l'_i \right) \\ &= \frac{1}{2n+1} (I + n + E) = \frac{2I + 3n}{2n+1} \end{aligned}$$

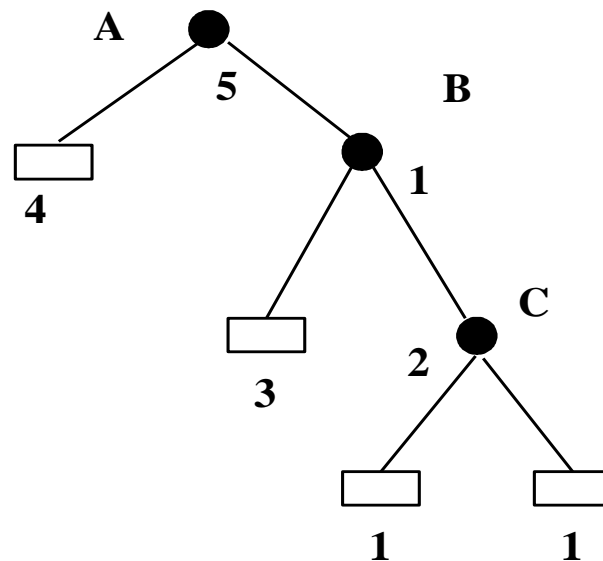
  
 $E = I + 2n$

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



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



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

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



- 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

➤ 树根定义:  $r(i, j)$

➤ 包含的内部结点关键码为:  $(key_{i+1}, key_{i+2}, \dots, key_j)$

➤ 内部结点和空树叶的权为:  $(q_i, p_{i+1}, q_{i+1}, \dots, p_j, q_j)$

➤ 查询代价定义:  $C(i, j)$

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

➤ 权的总和

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

## ► 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  $t(0, 1)$ ,  $t(1, 2)$ ,  $\dots$ ,  $t(n-1, n)$

► 第2步：构造包含2个关键码的最佳二叉搜索树

✓ 找  $t(0, 2)$ ,  $t(1, 3)$ ,  $\dots$ ,  $t(n-2, n)$

► 再构造包含3, 4,  $\dots$ 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  $\text{key}_k$  为根

➡ 左子树包含  $\text{key}_{i+1}, \dots, \text{key}_{k-1}$

✓  $C(i, k-1)$  已求出

➡ 右子树包含  $\text{key}_{k+1}, \text{key}_{k+2}, \dots, \text{key}_j$

✓  $C(k, j)$  已求出

➤  $C(i, j) = W(i, j) + \min_{i < k \leq j} (C(i, k-1) + C(k, j))$

➤ 给出关键码集合

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

➤ 及权的序列

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

➤ 构造最佳二叉搜索树

# 例子(续)



$i \backslash j$	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

$r(i, j)$

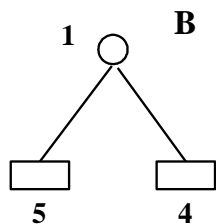
$i \backslash j$	0	1	2	3	4
0	0	10	28	43	57
1		0	12	27	40
2			0	9	19
3				0	6
4					0

$C(i, j)$

$i \backslash j$	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

$W(i, j)$

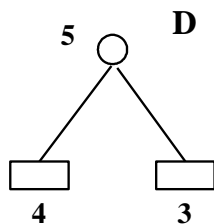
第一步



花费  
总权

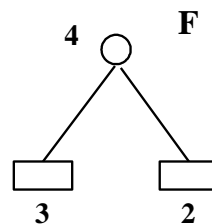
10

10



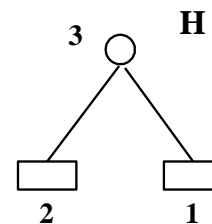
12

12



9

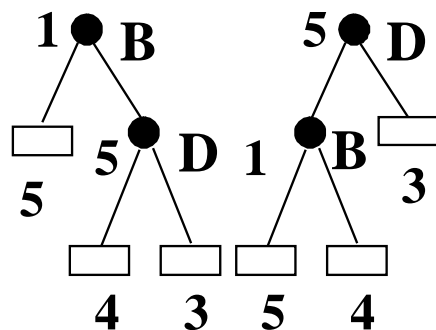
9



6

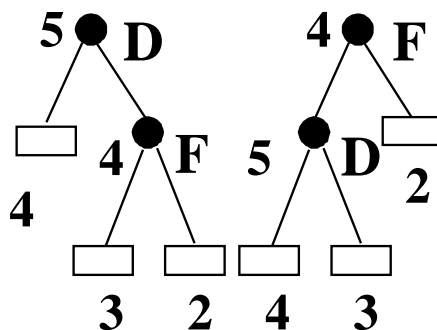
6

第二步



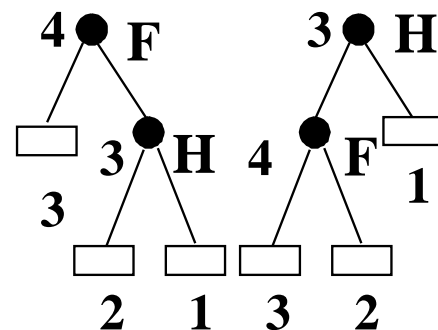
开销 30  
总权 18

**28**  
18



**27**  
18

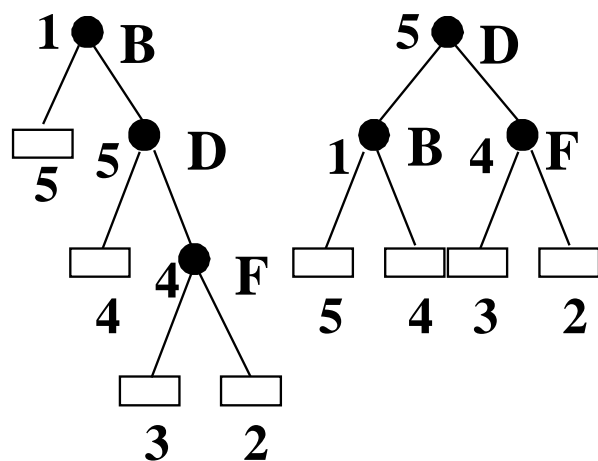
30  
18



**19**  
13

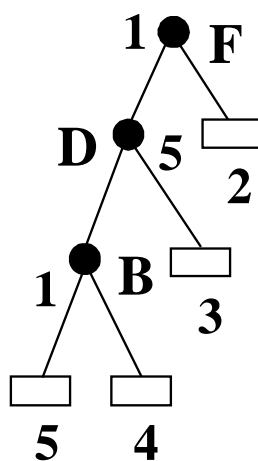
22  
13

第三步

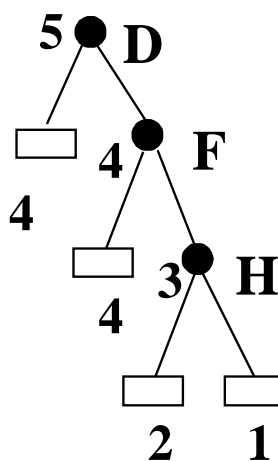


花费 51  
总权 24

**43**  
**24**



52  
24



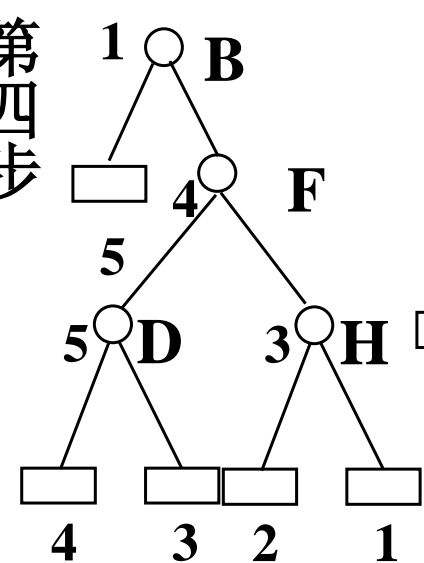
41  
22

**40**  
22

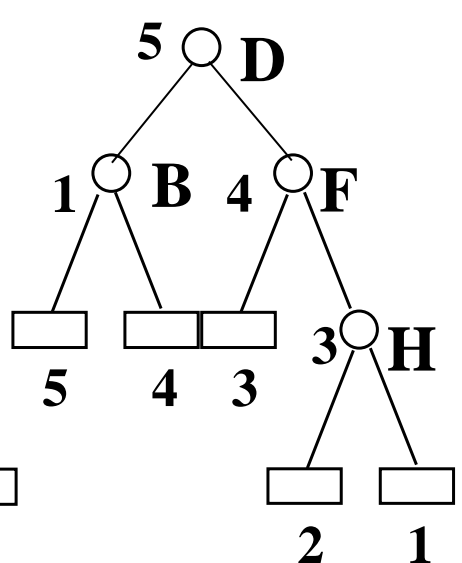
49  
22

# 最佳二叉搜索树 $t(i, j)$

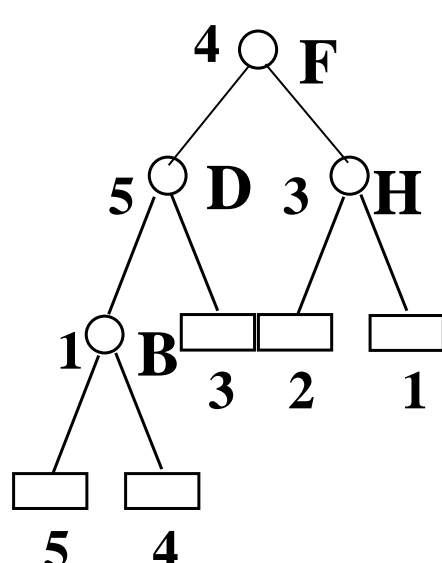
第四步



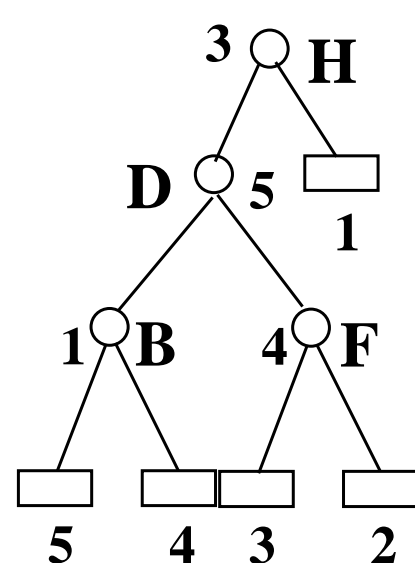
花费 68  
总权 28



57  
28



62  
28



71  
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 $|h_R - h_L| \leq 1$ ， $h_L$ 、 $h_R$ 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 $O(\log n)$ 。

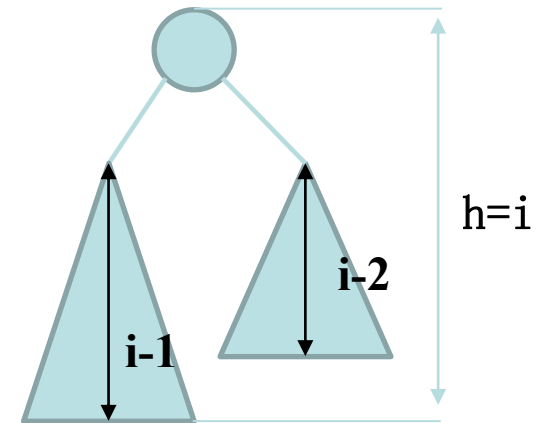
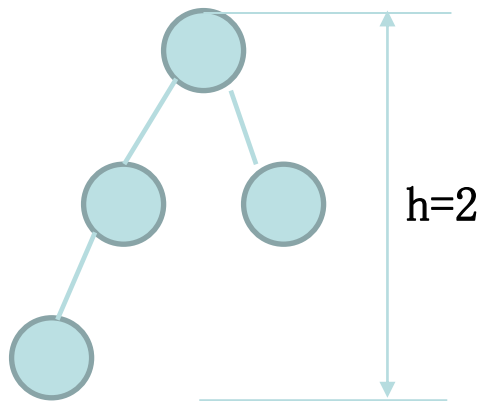
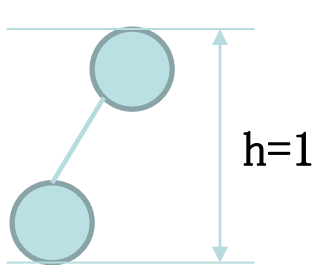
# 12.4.2 平衡二叉搜索树 (AVL)



定理：如果T是一棵具有n个结点的AVL树，其高为 $O(\log n)$ 。

证明：用 $t(i)$ 表示高度为i的AVL树，其节点数目的最少值。

显然  $t(1)=2$ ;  $t(2)=4$



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

## 12.4.2 平衡二叉搜索树 (AVL)



定理：如果T是一棵具有n个结点的AVL树，其高为 $O(\log n)$ 。

证明(续)：

考虑 Fibonacci数列  $F(i) = F(i-1) + F(i-2)$

可以用数学归纳法证明  $t(i) = F(i+3) - 1$

根据Fibonacci的渐进表示  $F(i) = f^i / \sqrt{5}$ , 其中  $f = (1 + \sqrt{5}) / 2$

可得：  $t(i) = (f^{i+3} / \sqrt{5}) - 1$

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

注意到 $t(i)$ 表示结点个数，因此上面的结论是对于一个有n个结点的AVL树，其高度不超过 $1.5 \log(n + 1)$

# 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

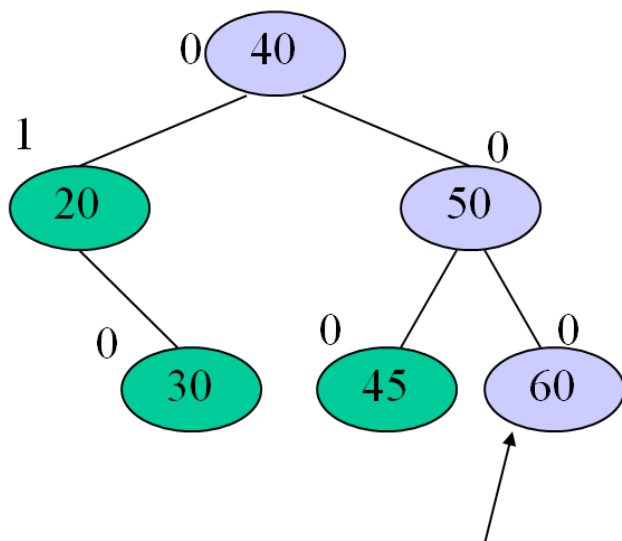


➤ 与BST树类似，执行失败的查找，确定插入位置插入

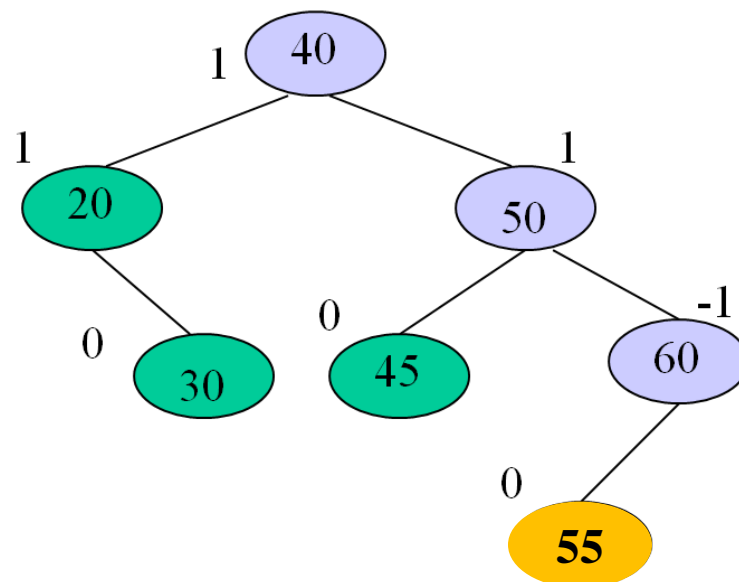
➤ 根据平衡因子决定是否调整

➤ **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

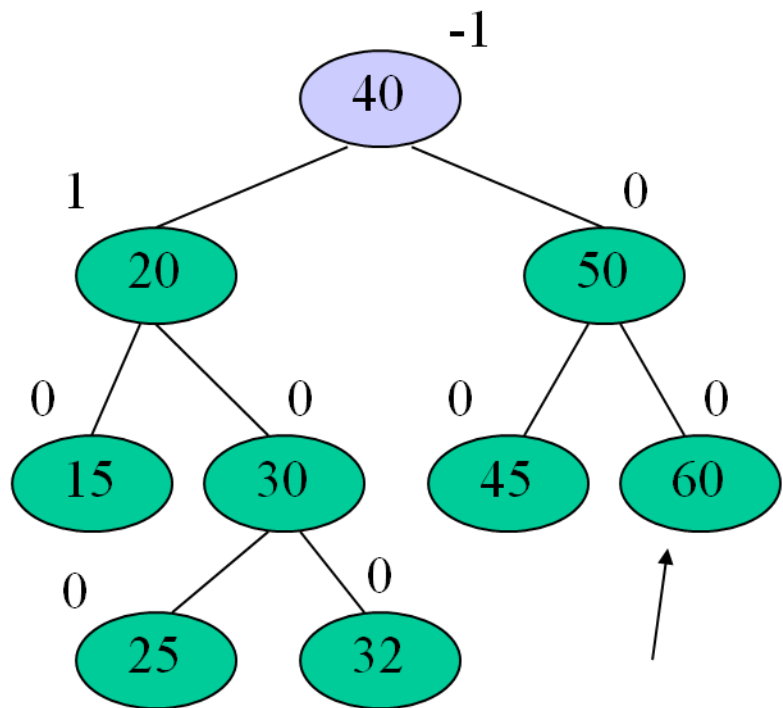


插入55后

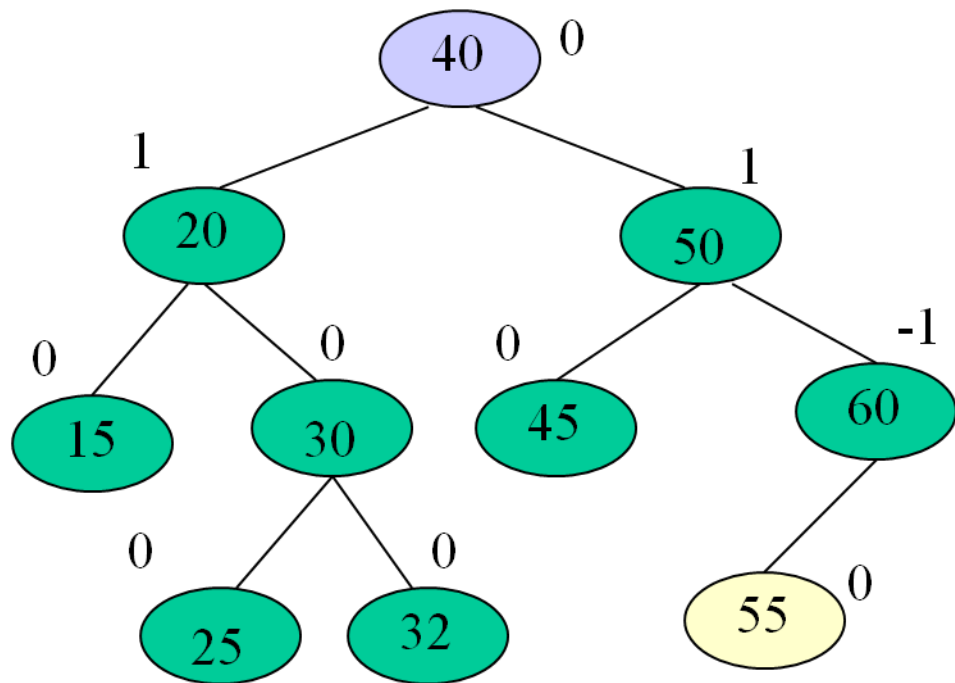


➤ **第二情况：** 路径上原来各结点是有偏重的，插入新结点后，反而使这些结点恢复平衡。

**插入55前**



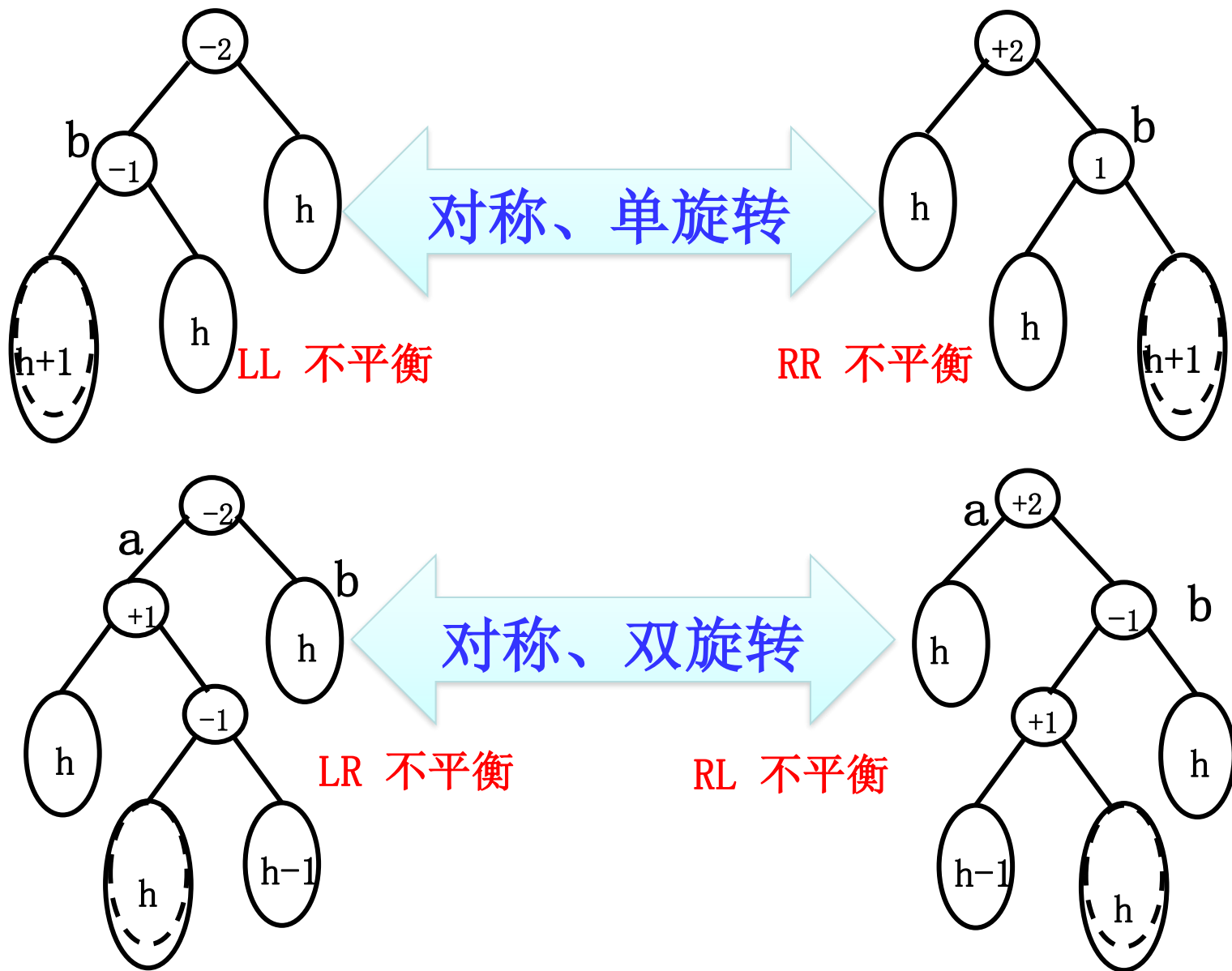
**插入55后**



- **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为失衡节点）
  - ➡ **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



- 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“ $A \rightarrow B \rightarrow C$ ”结点上
- 然后根据失衡形态进行单旋或者双旋操作

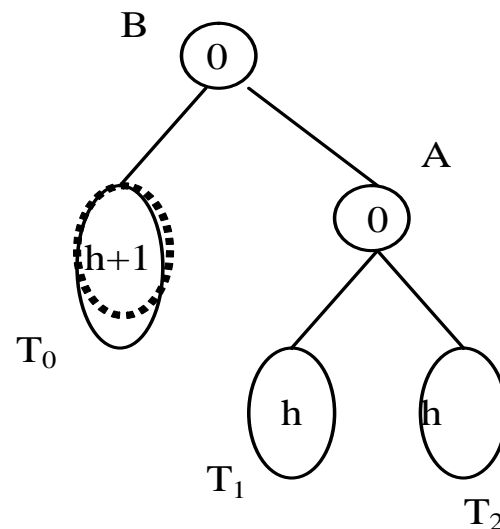
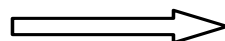
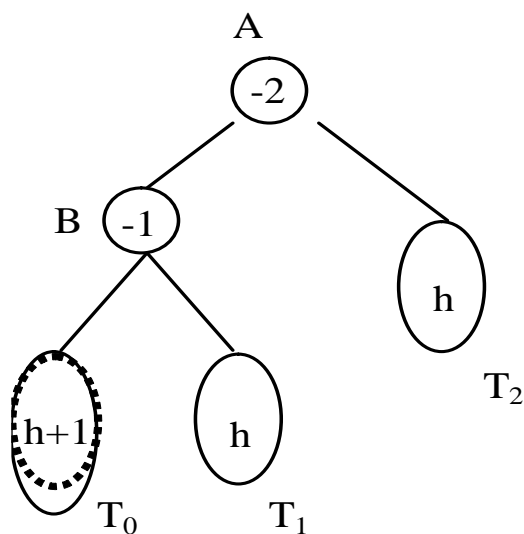
# 1、单旋转



➤ 调整过程仅涉及A和B两个结点（以LL型调整为例）

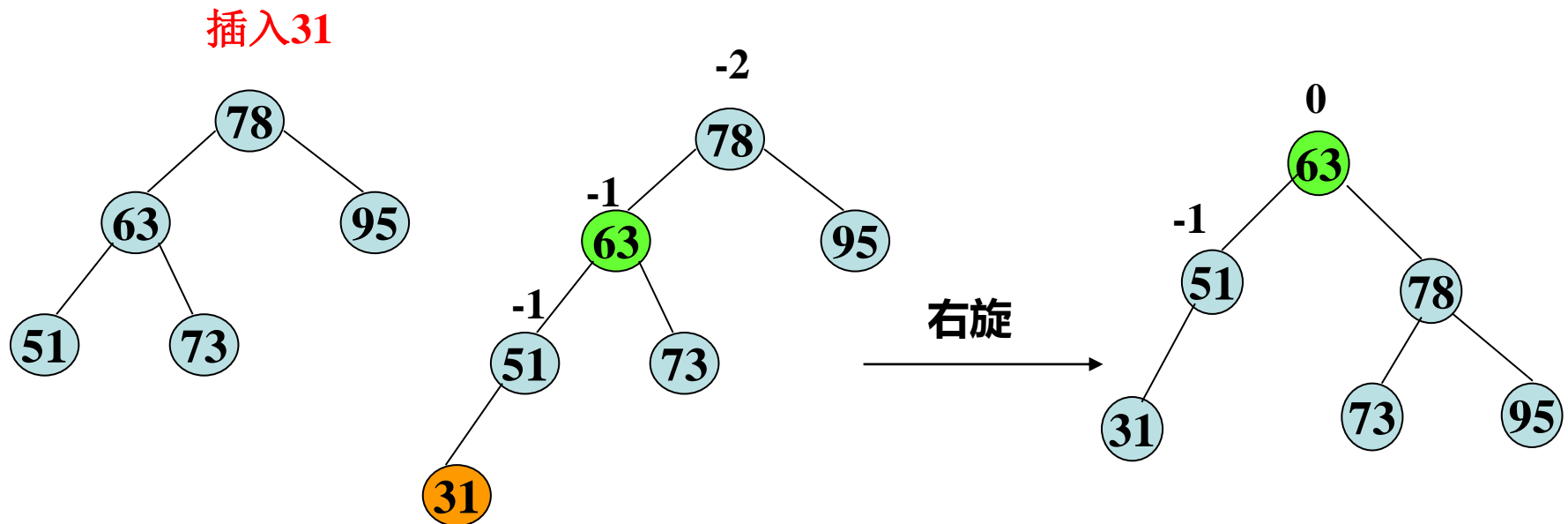
➡ 令B代替A成为新根，A作为B的右子结点，A和B的子树根据BST的性质分别挂接到B和A结点下，保持中序周游序列为

$T_0$  B  $T_1$  A  $T_2$



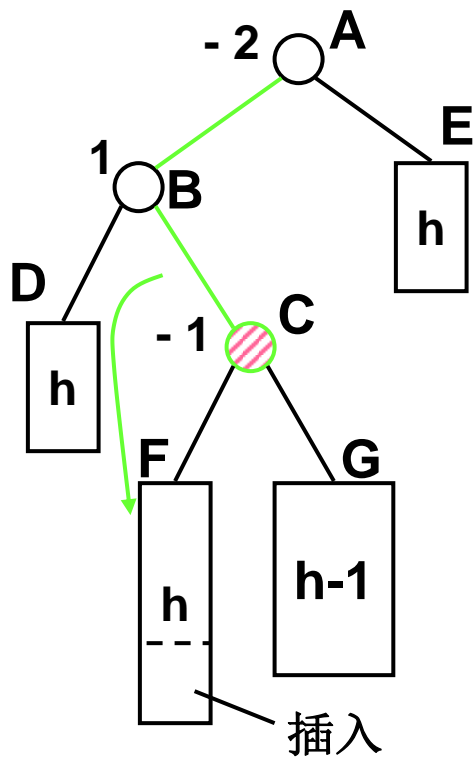
# 单旋转：LL型示例

- 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

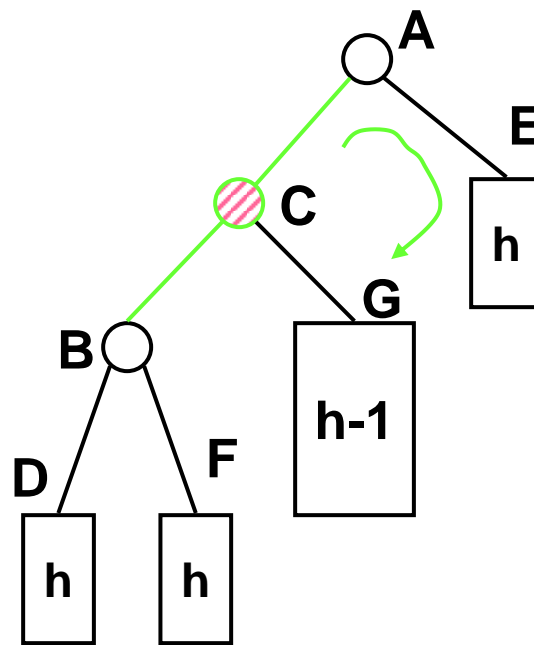


## 2、双旋转（提升中间节点）

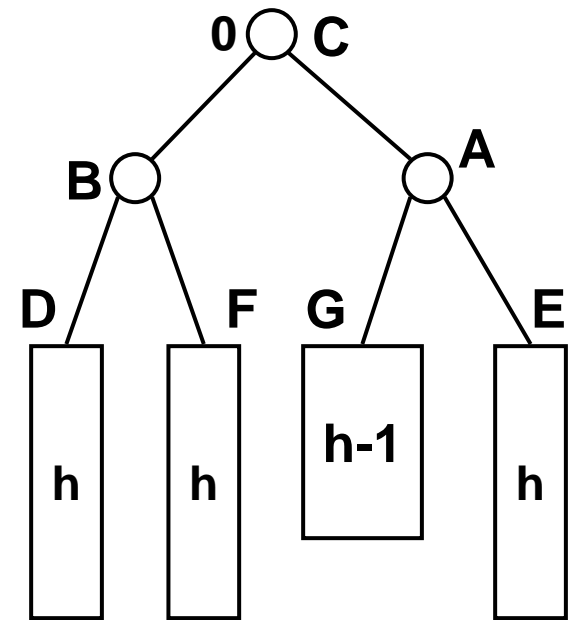
➤ 调整过程涉及ABC三个结点（以LR型调整为例）



(a) F子树插入结点  
高度变为h



(b) 绕C，将B  
逆时针转后

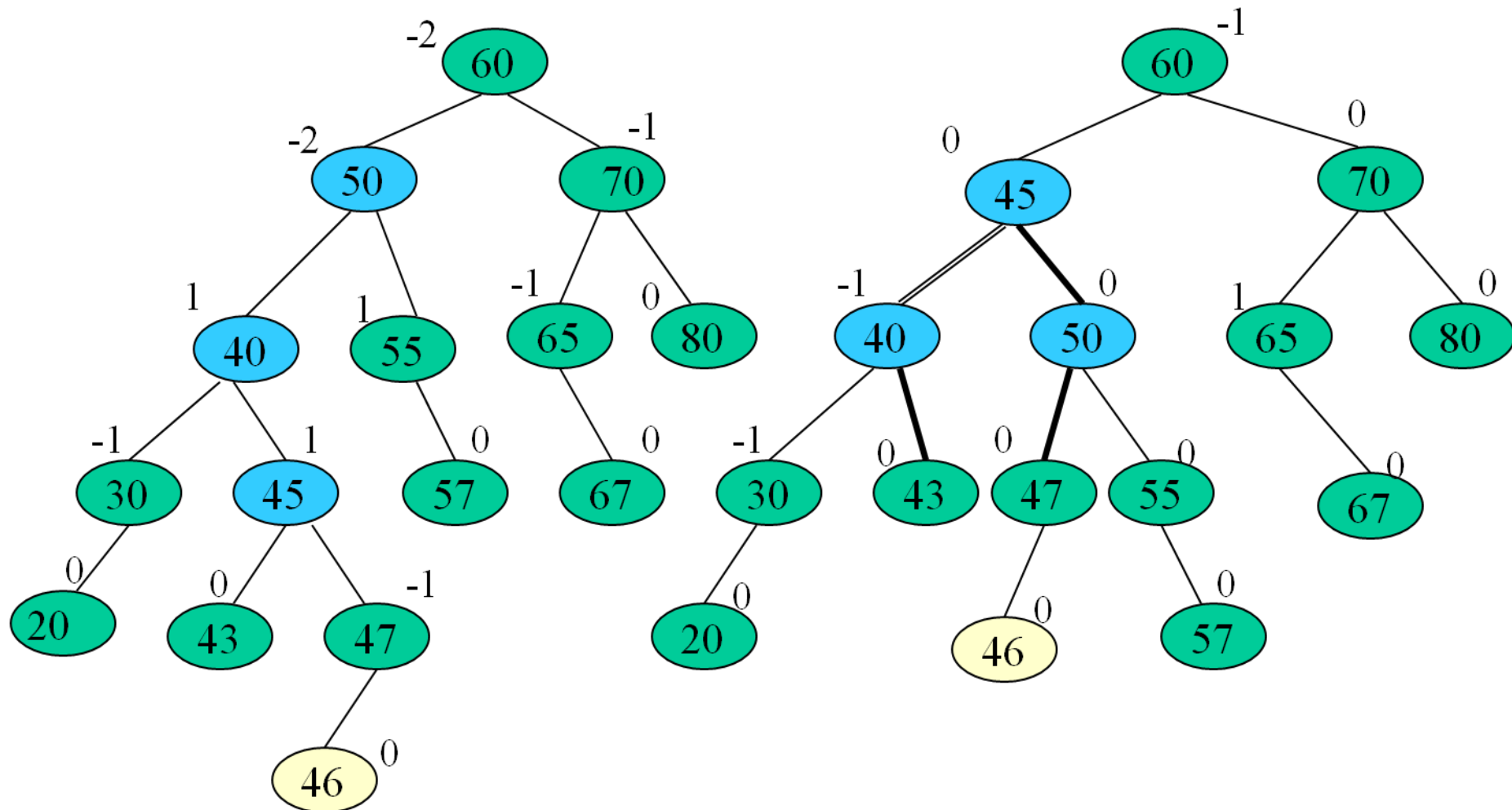


(c) 绕C，将A  
顺时针转后

# 双旋转：LR型示例

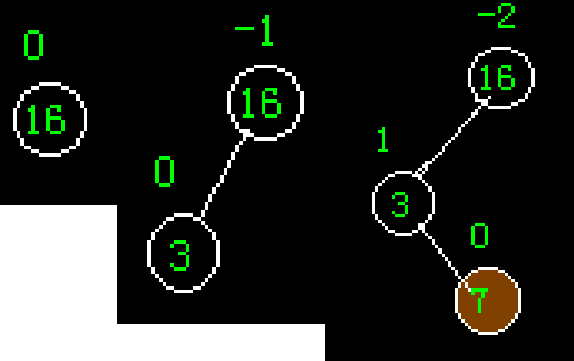


► 插入结点46

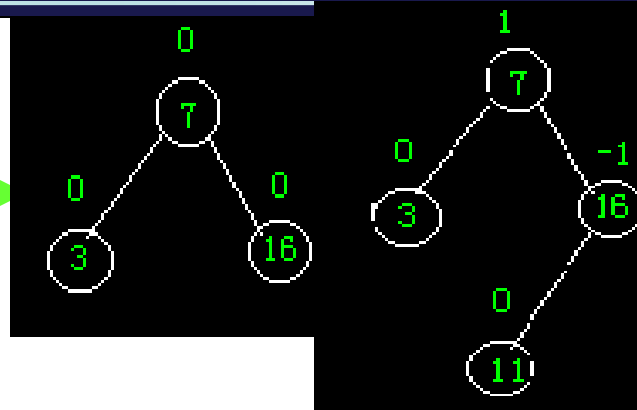


- 从空开始不断插入一组关键码序列，构成AVL树
- 每插入一个结点后就应判断从该结点到根的路径上  
有无结点发生不平衡
- 如有不平衡问题，利用旋转方法进行树的调整，使  
之平衡化
- 建AVL树过程是不断插入结点和必要时进行平衡化  
的过程

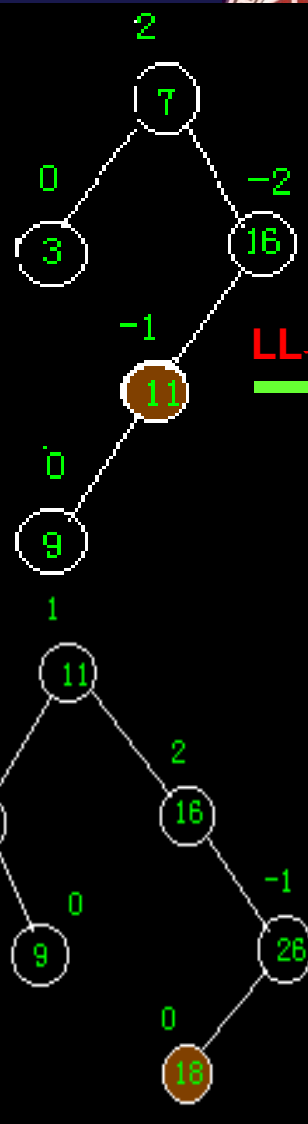
# 建树过程：依次输入16, 3, 7, 11, 9, 26, 18



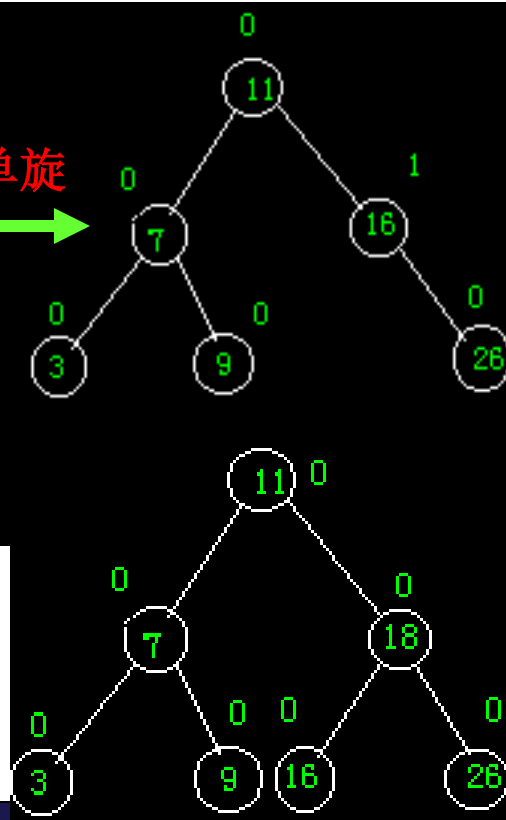
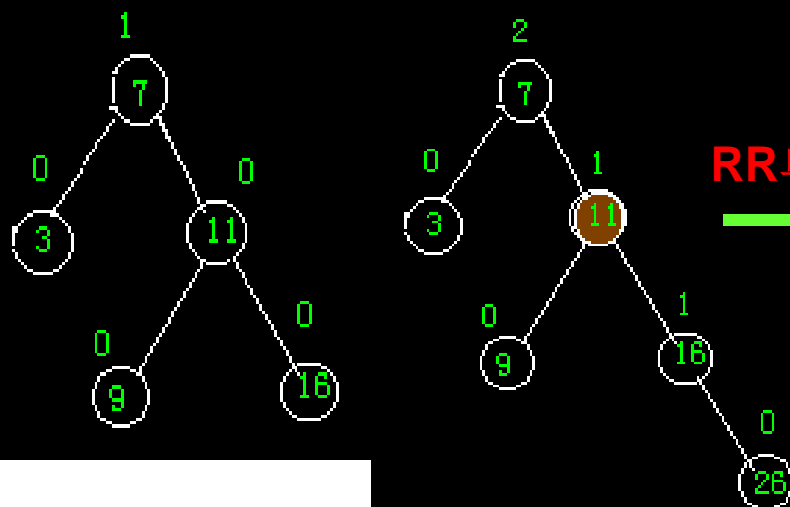
LR双旋



LL单旋



RR单旋



RL双旋





## 2. AVL树结点的删除

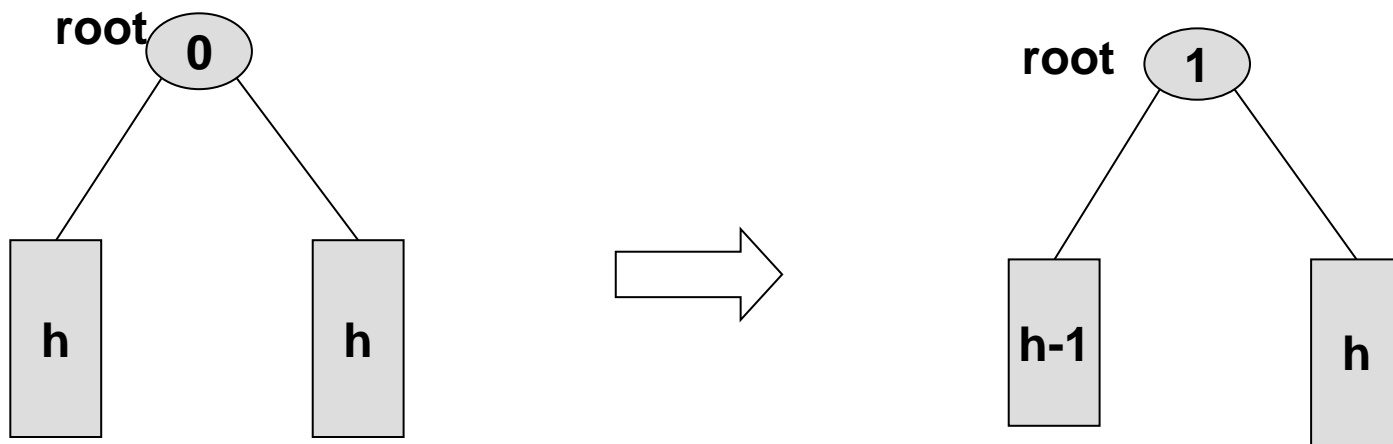
- 删除操作同BST删除：与后继交换再删除
- 删除会导致树高及平衡因子变化，需要沿着被删除结点到根结点的路径来调整这种变化。  
但删除比插入复杂
  - ➡ 删除后的再平衡可能需要在相应的路径上的不止一次实施单旋或双旋

- 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



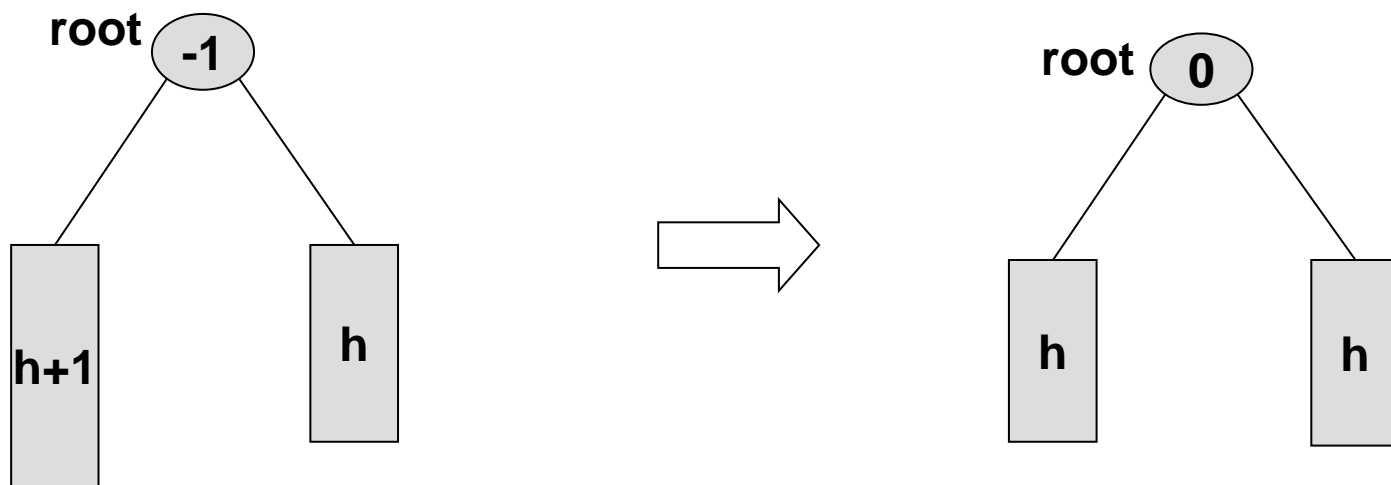
- $bf(\text{root})=0$ : 若其左或右子树被缩短, 则将其bf值置为1或-1, 同时置  $\text{modified} = \text{FALSE}$
- 由于以  $\text{root}$  为根的子树高度未变, 故不影响祖先结点的bf, 调整结束



# 删除操作: Case 2



- $bf(\text{root}) \neq 0$ , 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



➤  $bf(\text{root}) \neq 0$ , 且较低子树被缩短:  $\text{root}$ 失衡 (设较高子树为 $\text{right}$ ), 有如下三种情况

I. Case 3.1:  $bf(\text{right})=0$

➤ 执行单旋恢复 $\text{root}$ 平衡,  $\text{modified} = \text{FALSE}$

II. Case 3.2:  $bf(\text{right})$ 与 $bf(\text{root})$ 相同

➤ 执行单旋来恢复平衡,  $\text{modified} = \text{TRUE}$

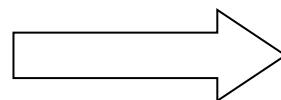
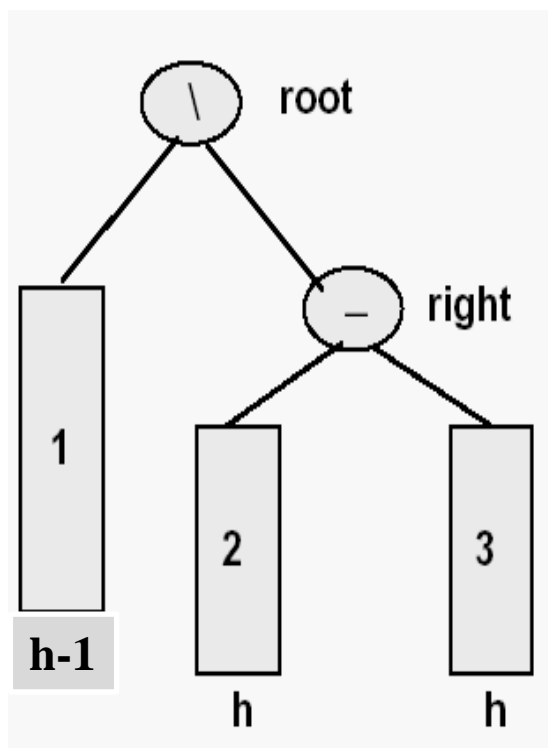
III. Case 3.3:  $bf(\text{right})$ 与 $bf(\text{root})$ 不同

➤ 执行双旋来恢复平衡,  $\text{modified} = \text{TRUE}$

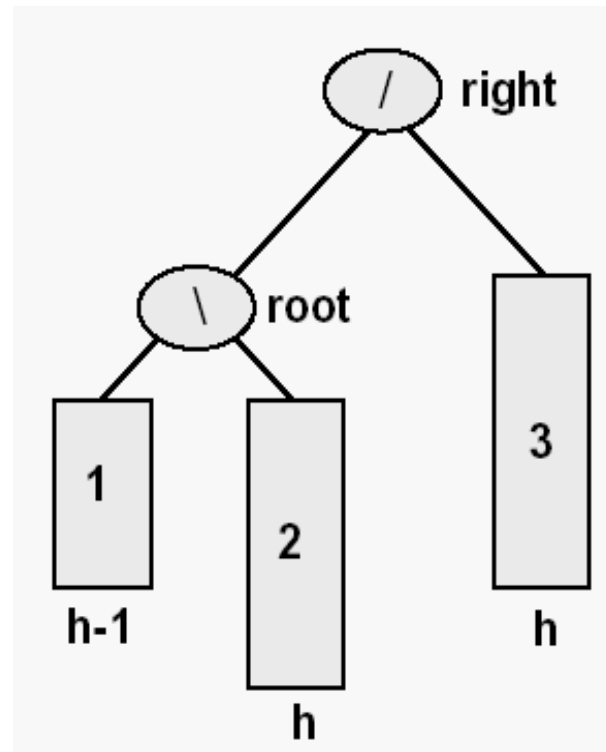
# 删除操作: Case 3.1



假定root的右子树是平衡的:  $bf(right) = 0$



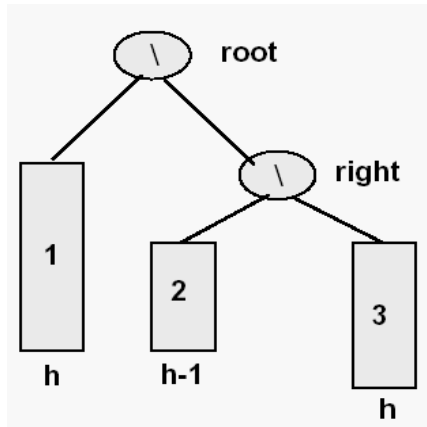
简单的左旋



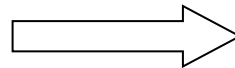
***modified = FALSE***

# 删除操作: Case 3.2

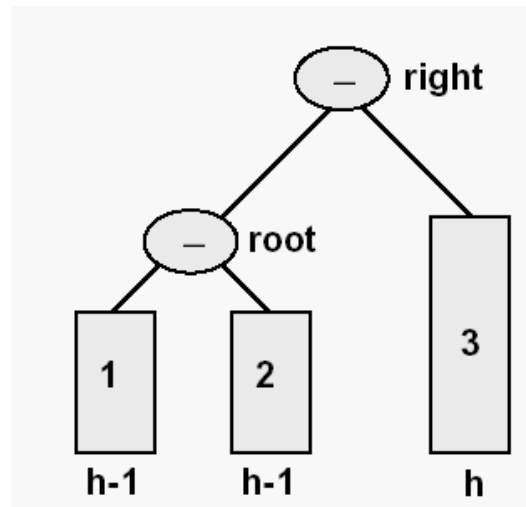
$bf(right)$  与  $bf(root)$  相同 树高= $h+2$



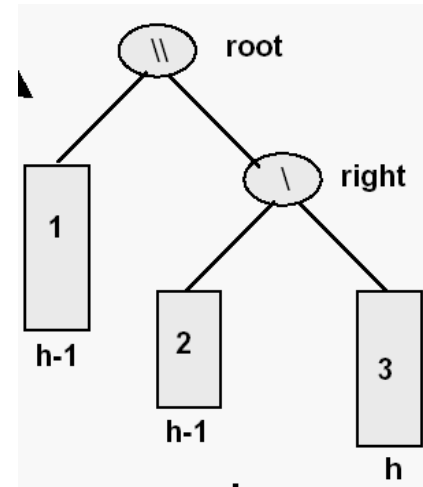
删除前的子树



树高= $h+1$



*modified = TRUE*



删除后的子树

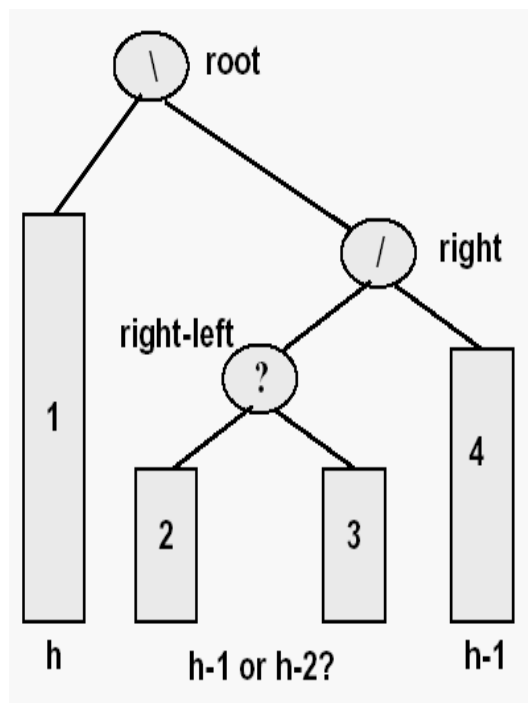


简单的左旋可恢复平衡

# 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 .....

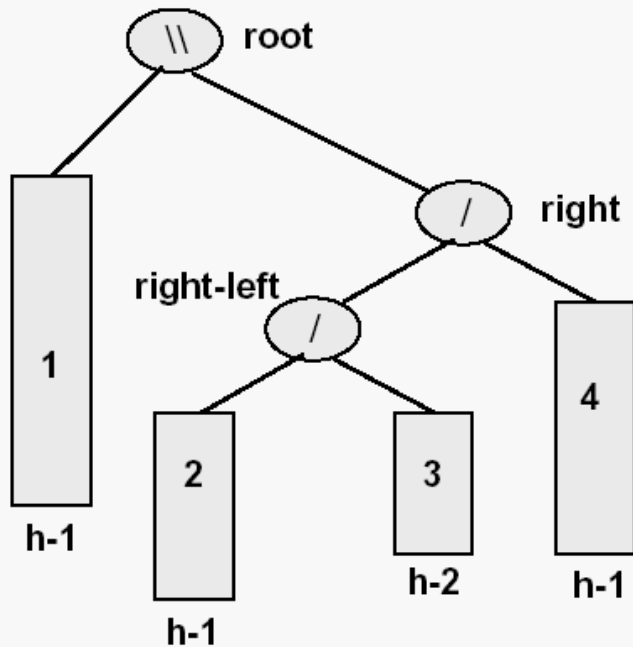


# 删除操作: Case 3.3.1

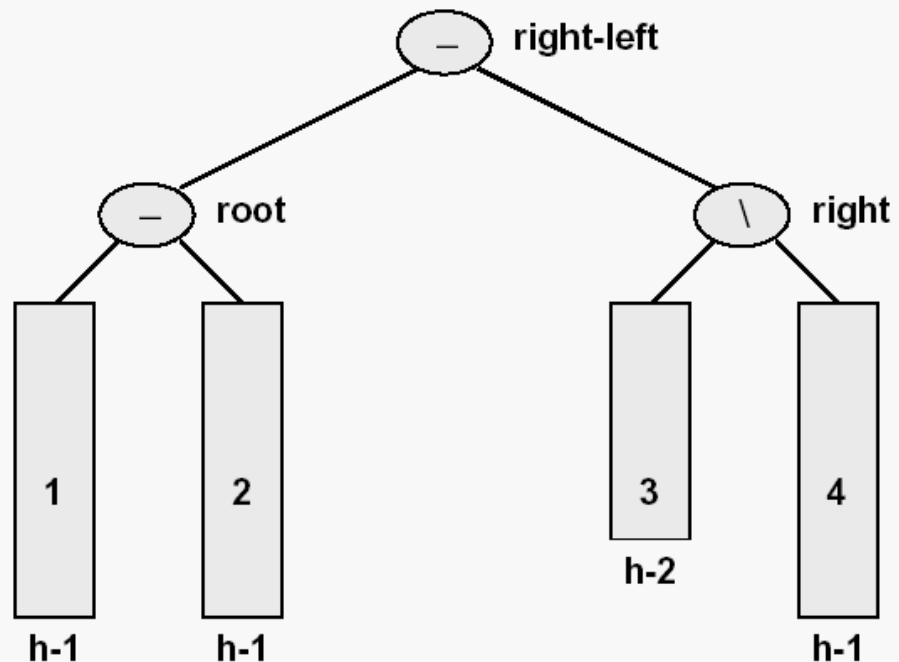


如果right-left子树也是左重的，可得如下结果：

树高= $h+2$



树高= $h+1$

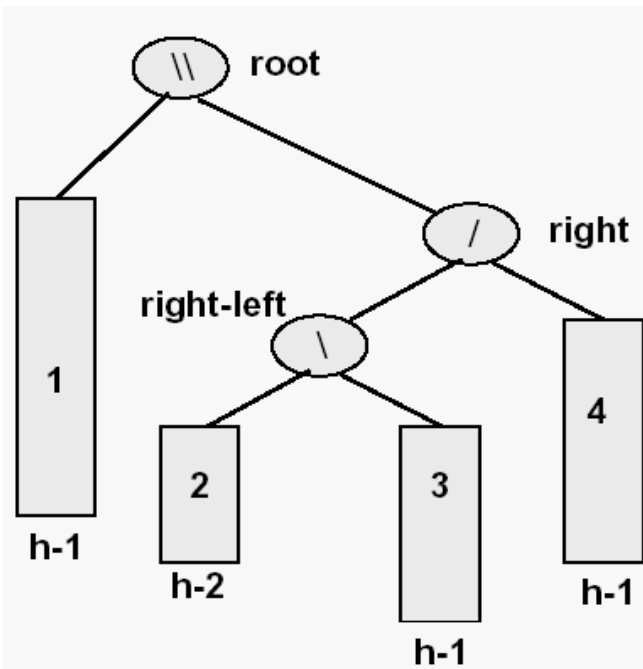


*modified = TRUE*

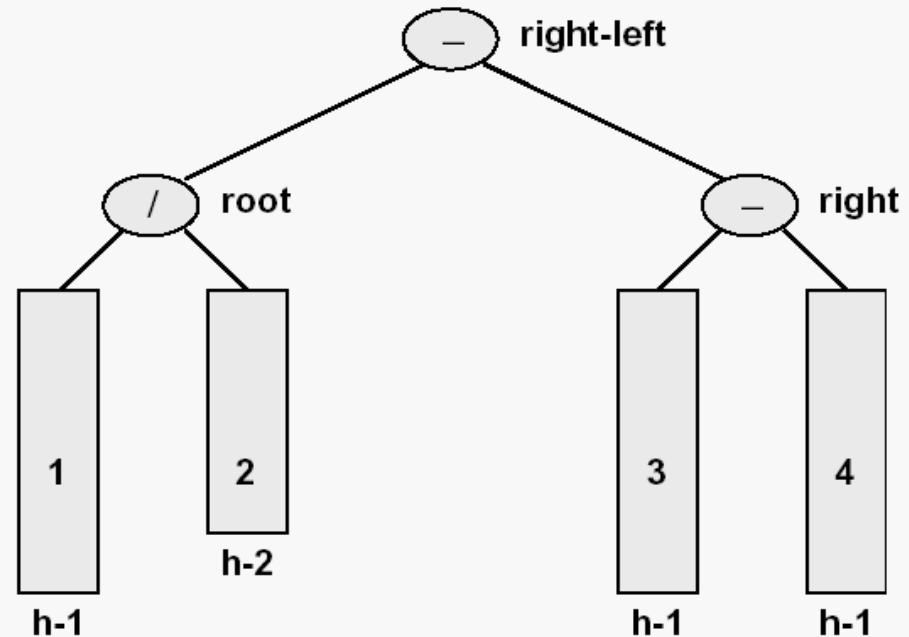
# 删除操作: Case 3.3.2

如果right-left子树是**右重**的，可得到如下结果：

树高= $h+2$



树高= $h+1$



*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*

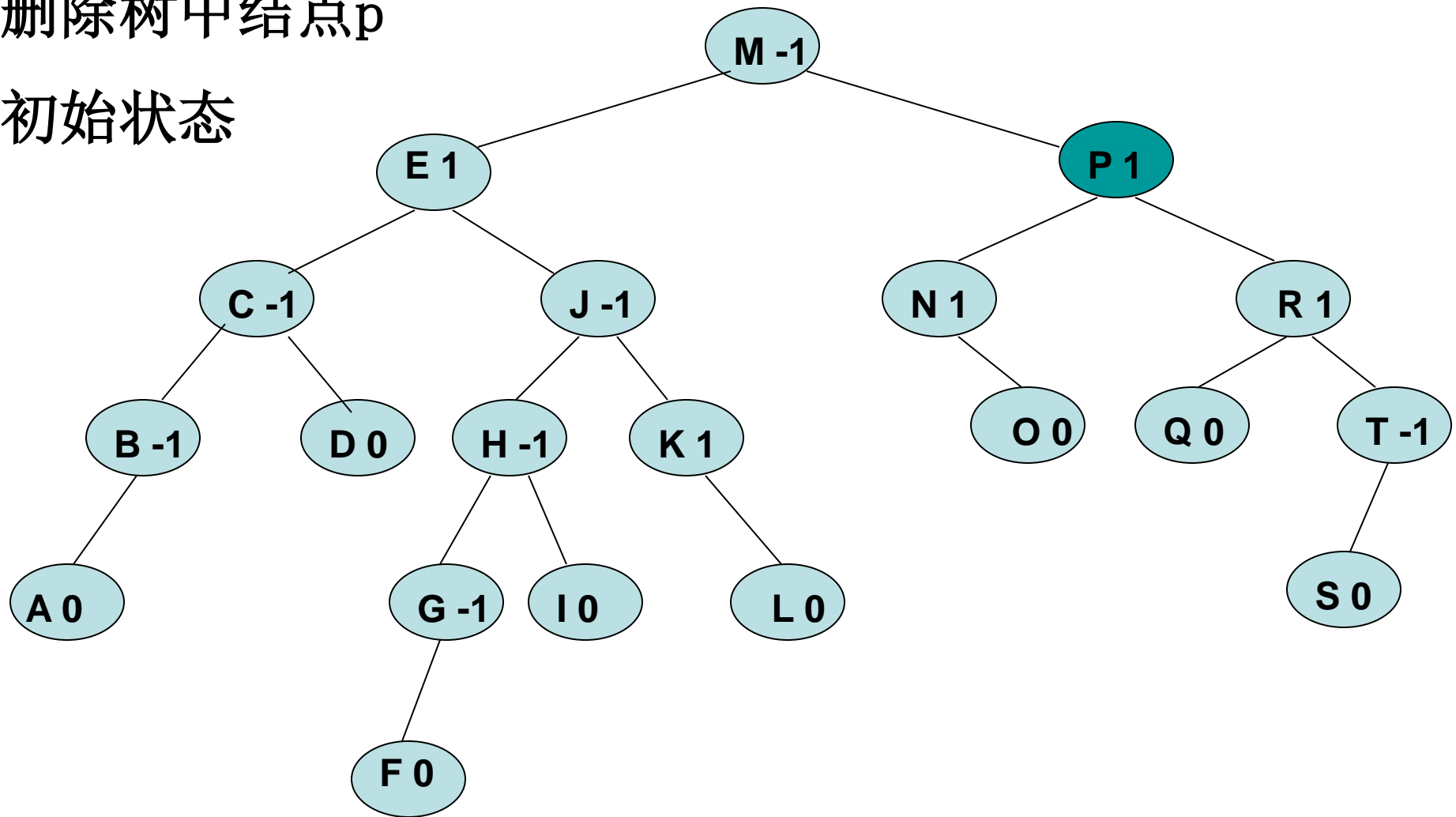
- 前述讨论均假设删除操作发生在左子树上，因为对称的缘故，发生在右子树上时有类似的情况要考虑

# AVL树结点删除的示例

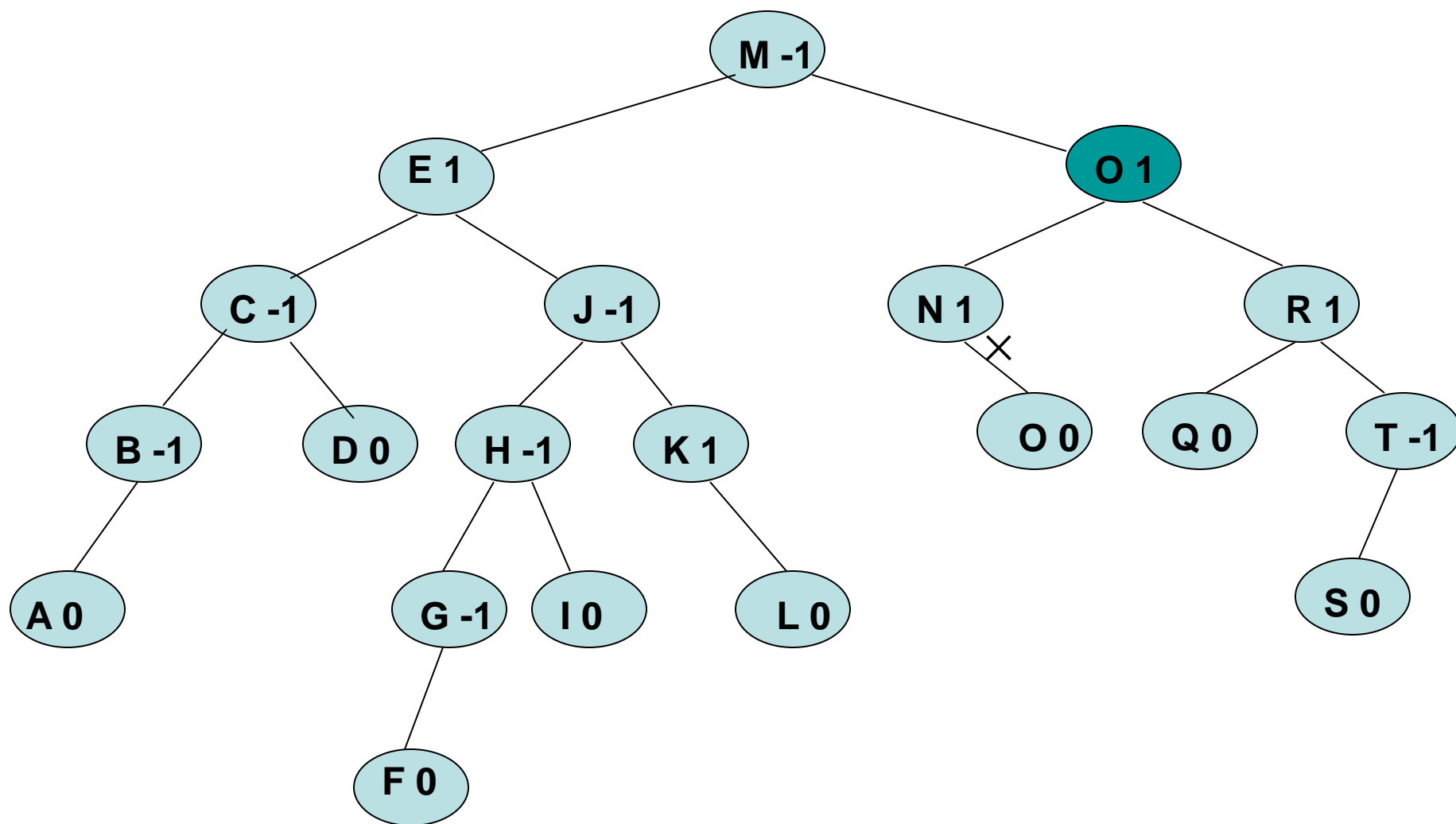


删除树中结点p

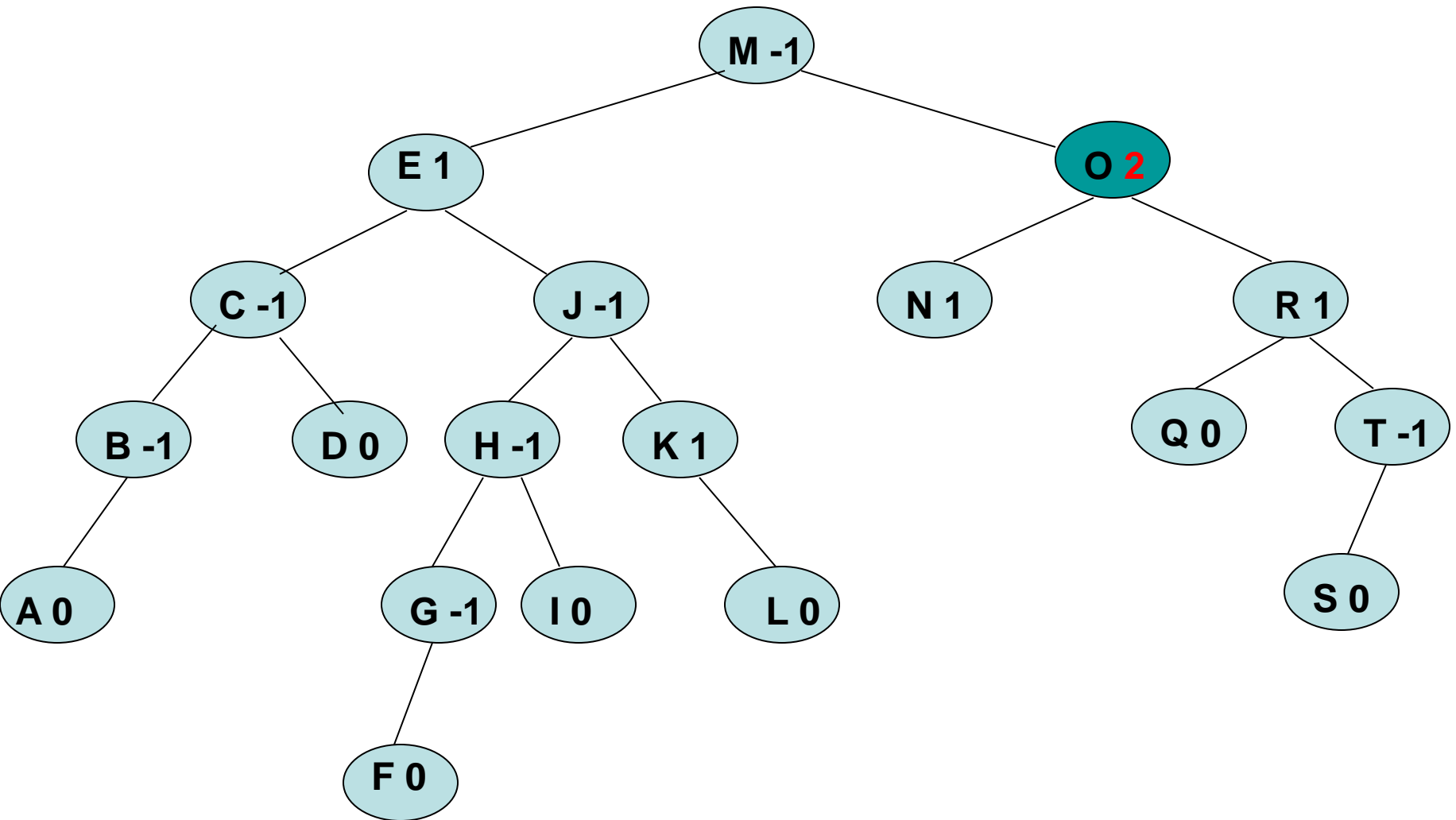
初始状态



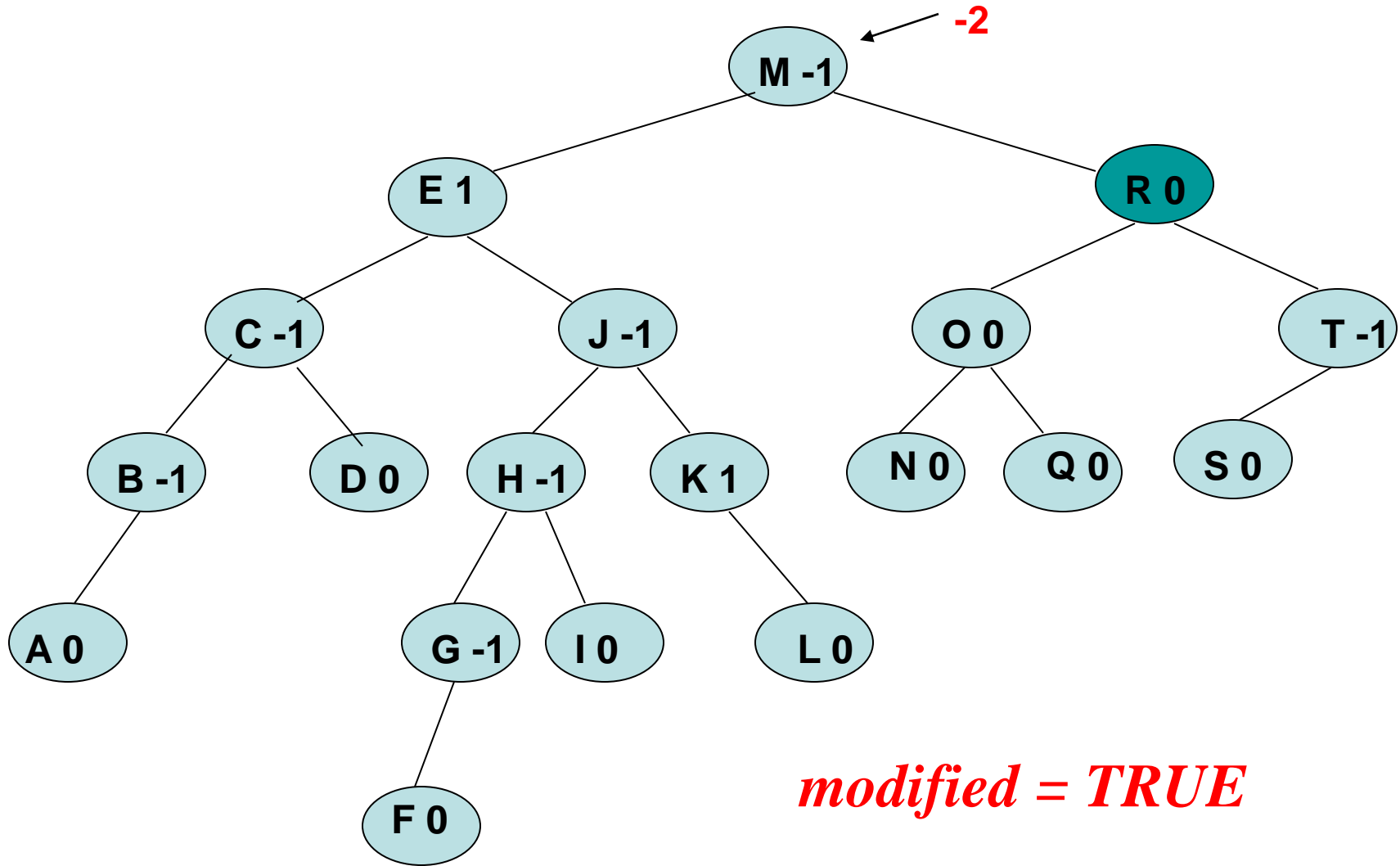
# 删除树中结点P，用0替换P



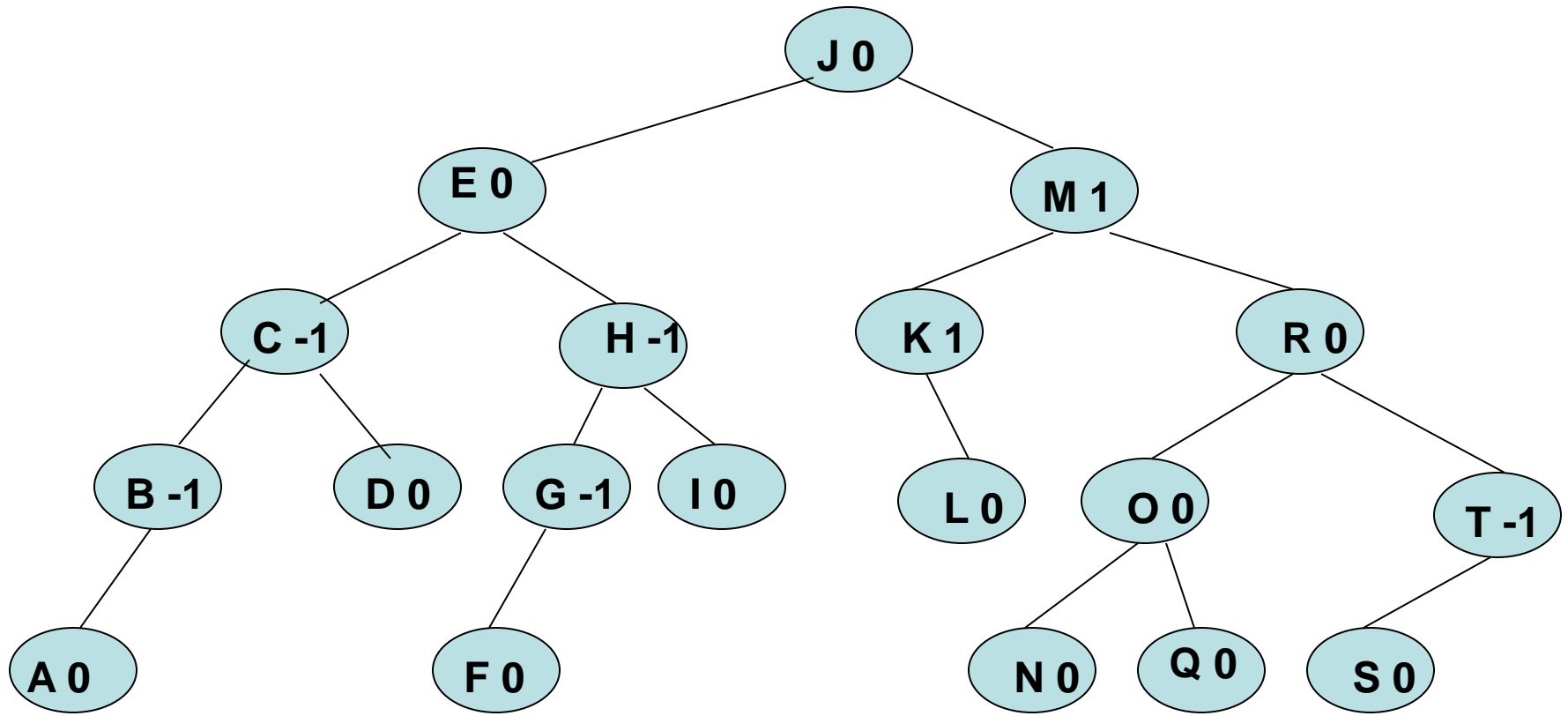
# 删除后状态，继续RR型旋转



# 调整后路径上又出现失衡结点



# LR型旋转调整





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

# 12.4.3 伸展树 (Splaying Tree)

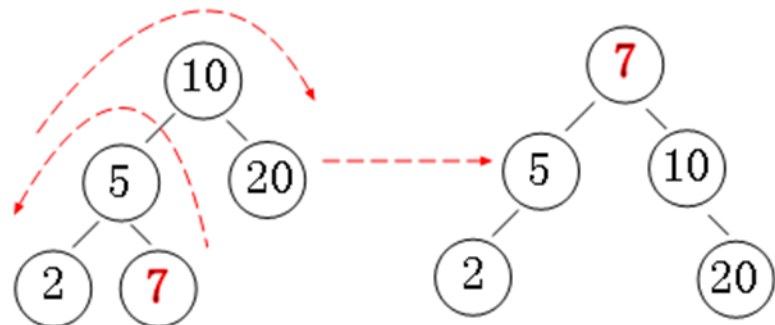


## ➤ 数据访问的“二八规则”

- ➡ 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，而只是改进BST性能的一组规则

- ➡ 保证访问的总代价不高
- ➡ 不能保证树高平衡

➤ 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

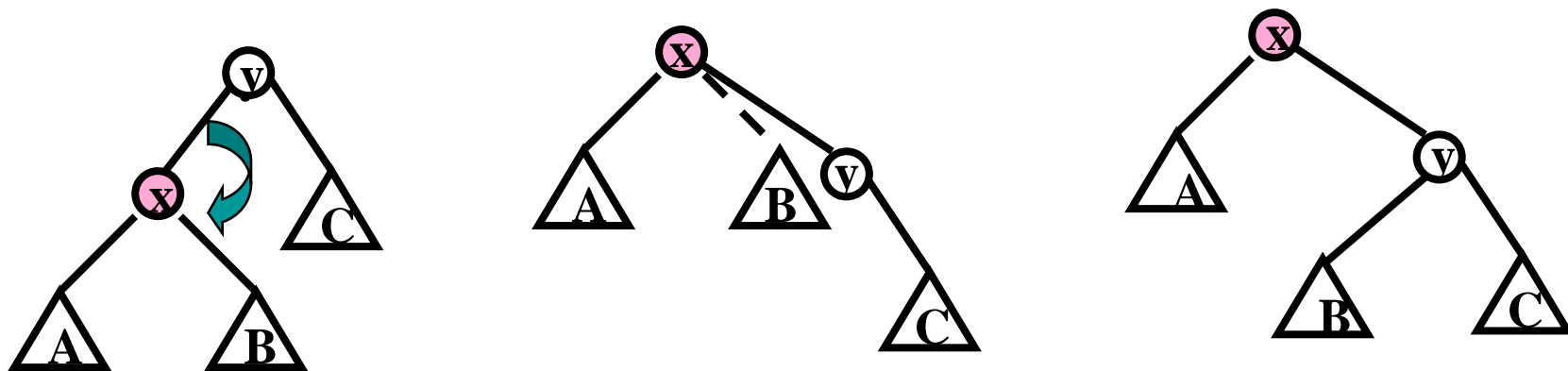
➤ 结点  $x$  的展开包括一组旋转

- ➡ 调整结点  $x$ 、父结点、祖父结点，把  $x$  旋转到更高层
- ➡ 旋转分：单旋转和双旋转

➤ 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



## ► 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

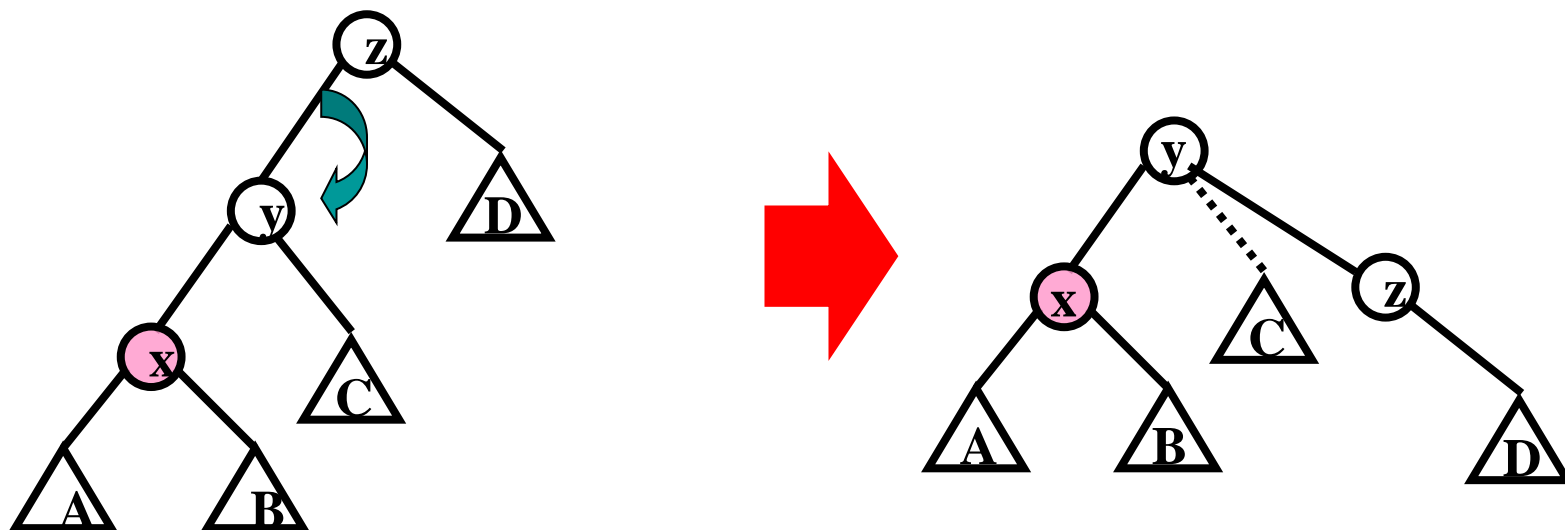
# 一字形旋转

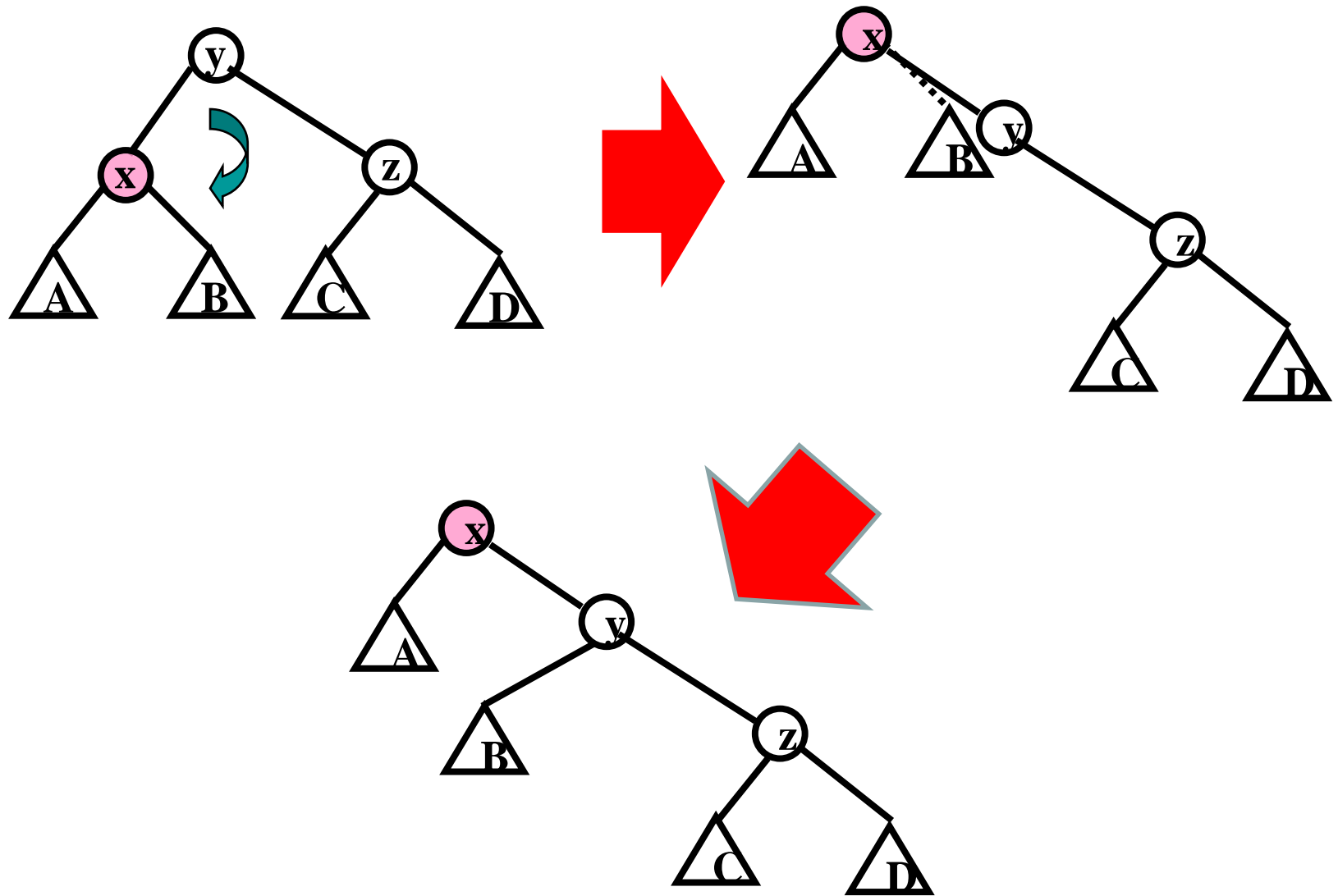


►  $x$ 、 $y$ 、 $z$ 是一顺的

► 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







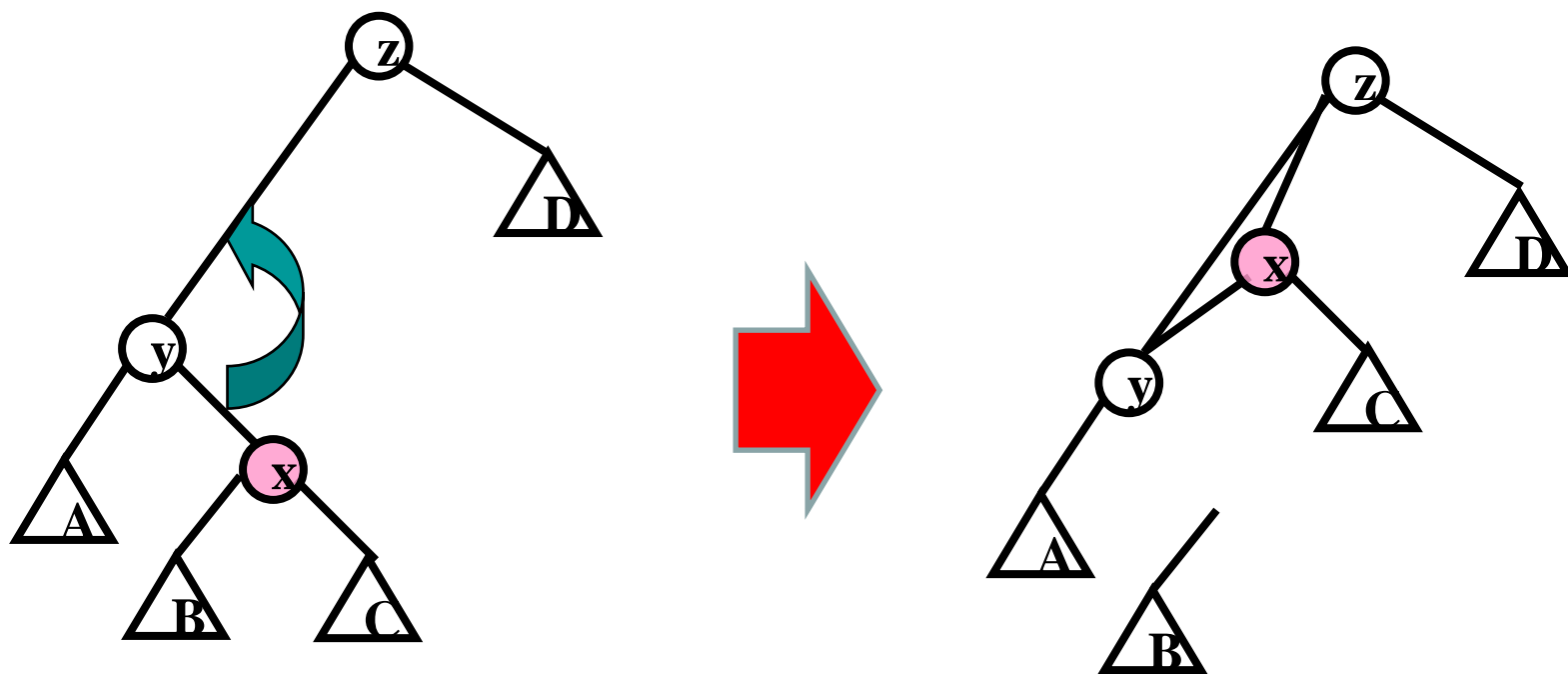
# 之字形旋转

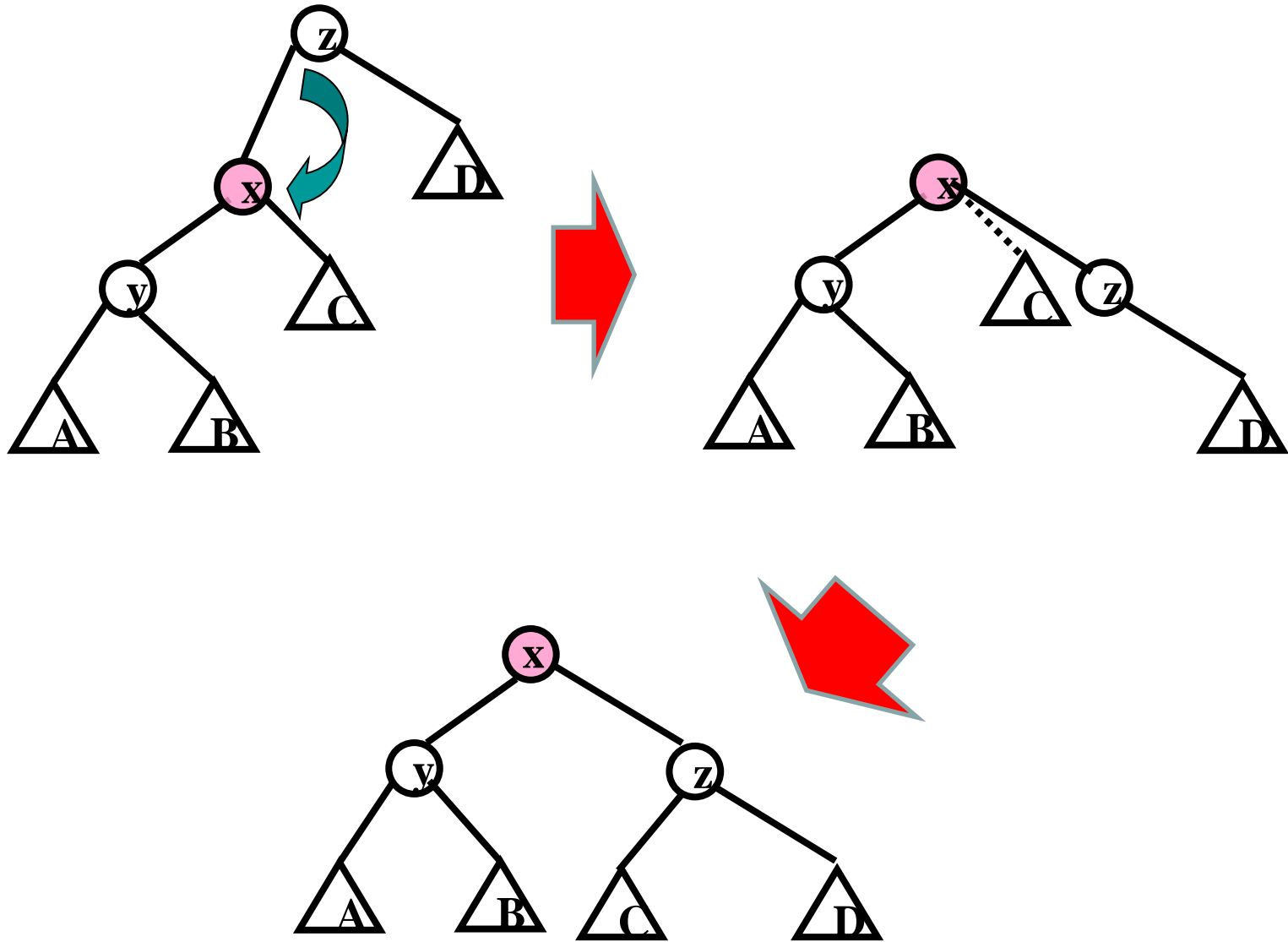


➤ 出现以下两种情况之一时，会发生之字形旋转：

➤  $x$ 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





## ➤ 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

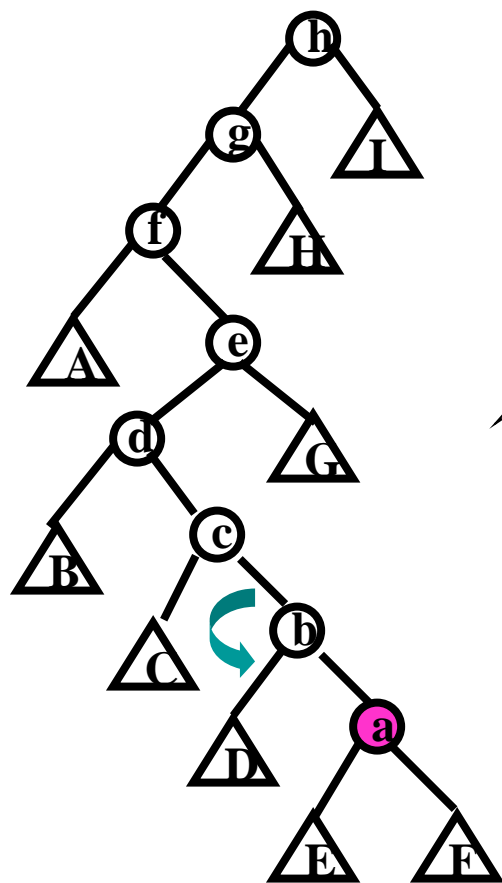
- ➡ 进行一次单旋转使结点 $x$ 成为根结点

## ➤ 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

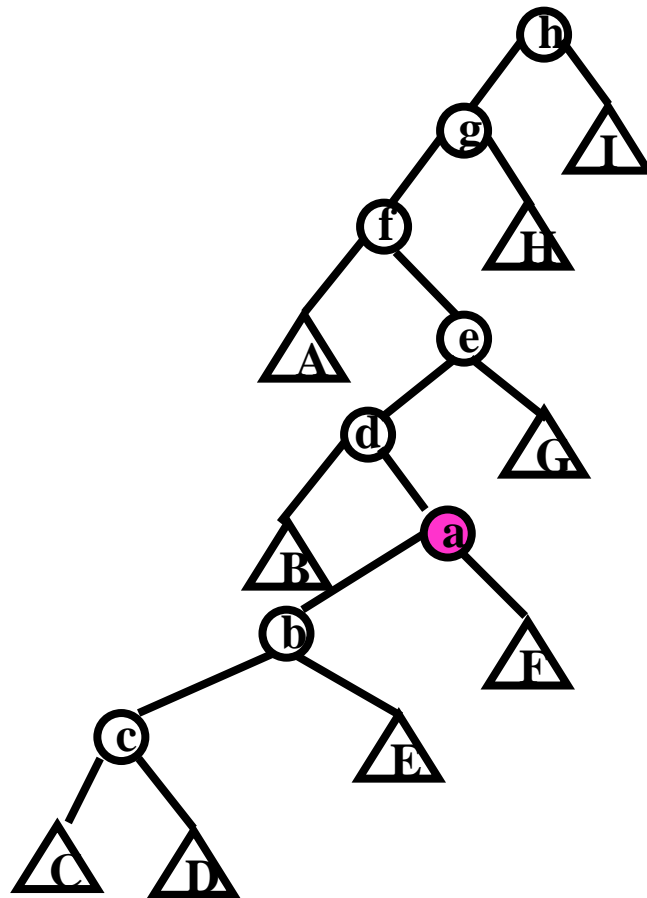
- ➡ 从而减少访问代价

# 伸展树的调整过程示例

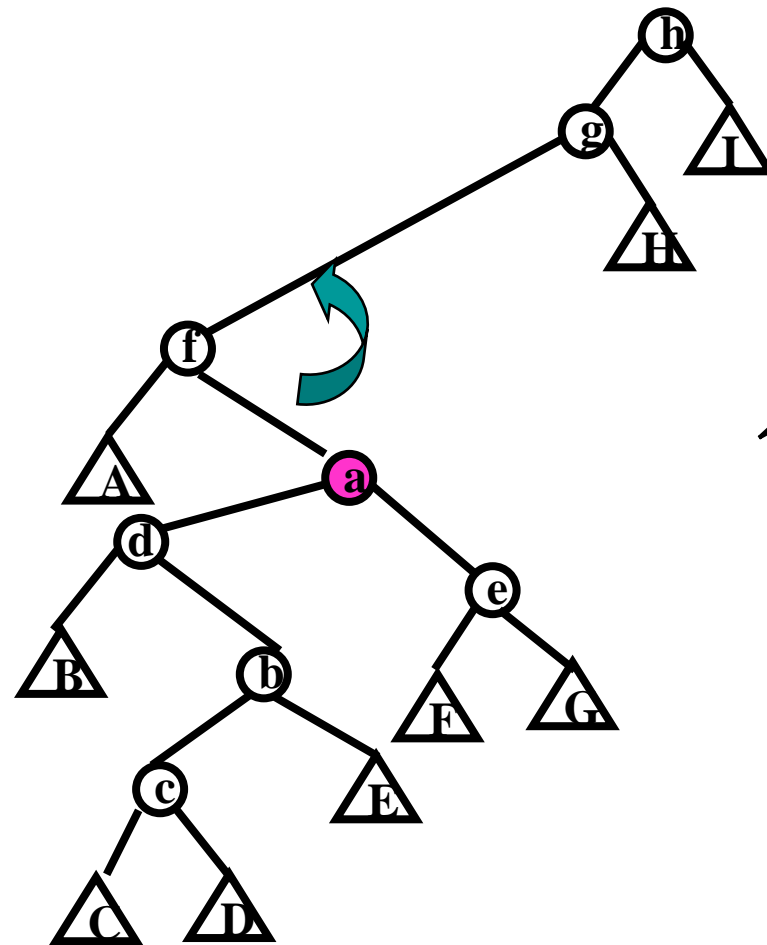


一字形旋转

(b, c) 左转  
(a, b) 左转



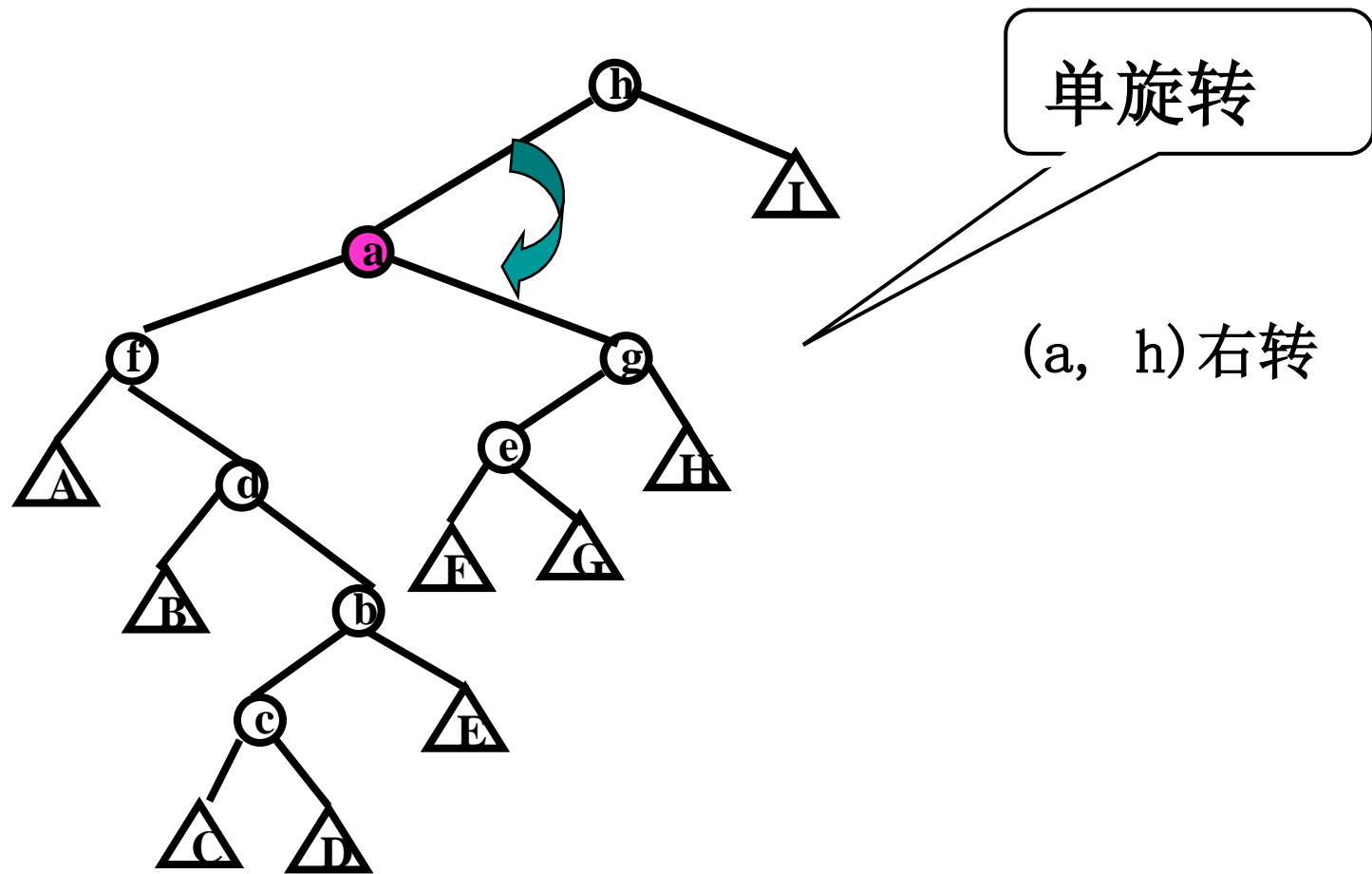
之字形调整



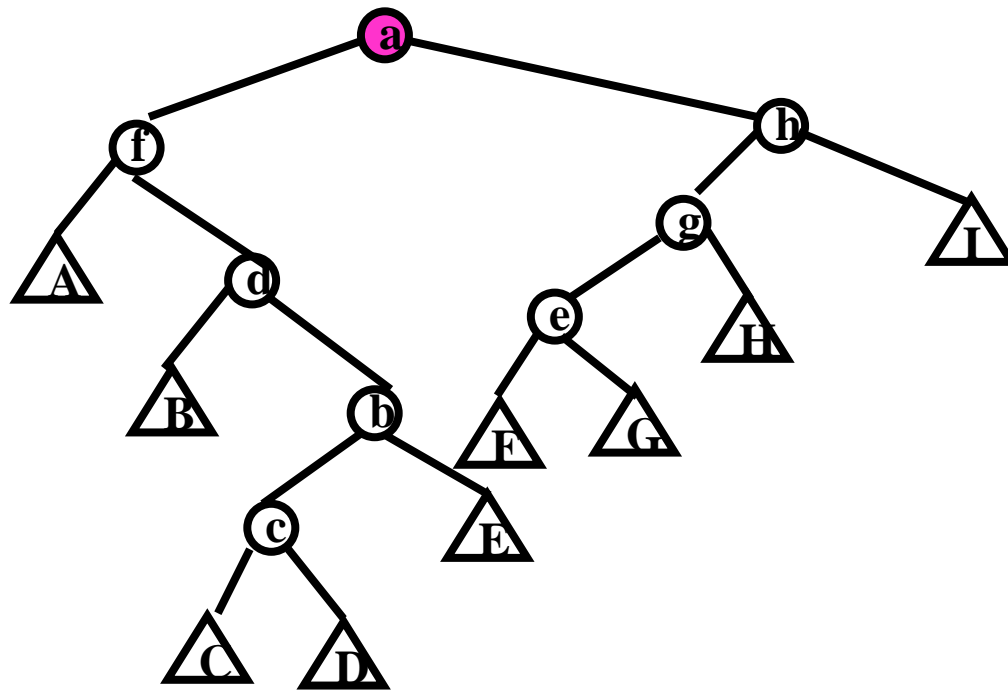
之字形旋转

(a, f) 左转

(a, g) 右转







(a, h) 右转

- 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 $O(m \log n)$ 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

# 基于伸展树的区间操作

## ✧1) 区间提取

给定一个数列 $[n_1, n_2, \dots, n_m]$ , 假设我们需要提取区间 $[n_a, n_b]$ 的子序列。

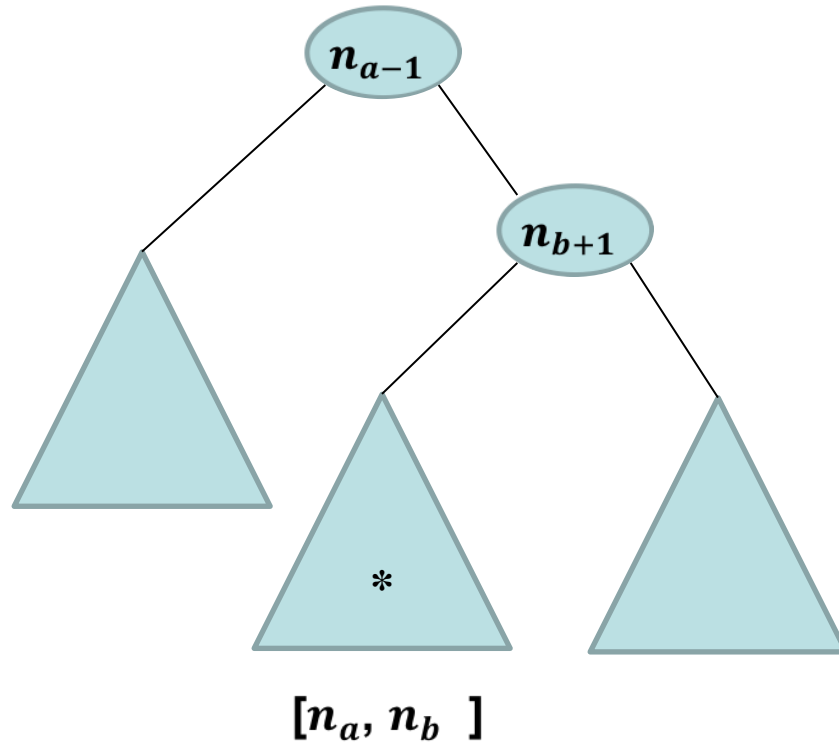
步骤:

Step 1: 将位置 $n_a$ 前面的元素转到树根;

Step 2: 将位置 $n_b$ 后面一个元素转到树根的右儿子;

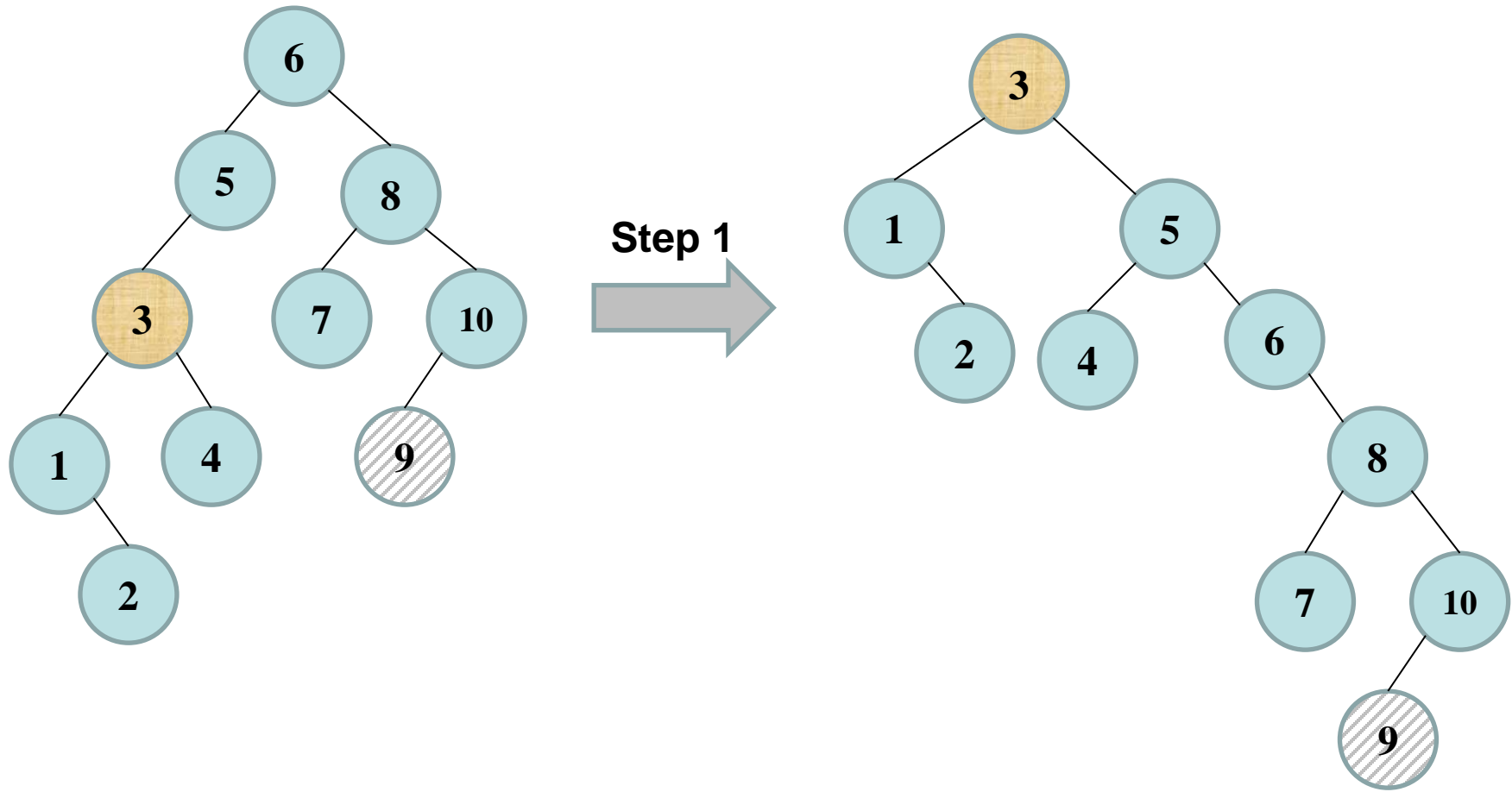
Step 3: 则树根右儿子的左子树就是需要提取的区间。

# 基于伸展树的区间操作

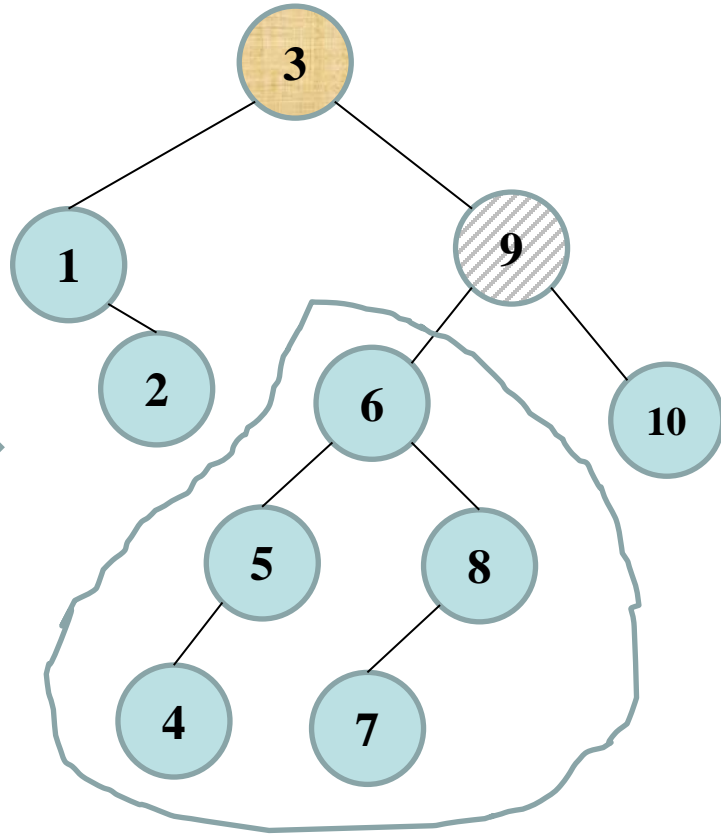


例如:  $A=[1,2,3,4,5,6,7,8,9,10]$   
提取  $[4---8]$

# 基于伸展树的区间操作



# 基于伸展树的区间操作



# 基于伸展树的区间操作

## ✧2) 区间删除

给定一个数列  $[n_1, n_2, \dots, n_m]$ , 假设我们删除区间  $[n_a, n_b]$  子序列。

步骤:

Step 1: 利用区间提取方法找到区间所对应的子树;

Step 2: 删除对应的子树





# 基于伸展树的区间操作

## ✧3) 区间插入

给定一个数列 $A=[n_1, n_2, \dots, n_m]$ 和对应的伸展树, 假设我们在 $n_a$ 和 $n_{a+1}$ 之间插入一个新序列 $B$ 。

Step 1: 把 $n_a$ 转到树根;

Step 2: 把 $n_{a+1}$ 转到树根的右儿子;

Step 3: 将待插入序列 $B$  构建成一棵伸展树。

Step 4: 把新构建的伸展树插入到树根的右儿子的左子树。

# 基于伸展树的区间操作

## ✧3) 区间翻转

给定一个数列 $A=[n_1, n_2, \dots, n_m]$ 和对应的伸展树, 假设我们在 $n_a$ 和 $n_b$ 之间插入一个子序列做一个前后翻转。

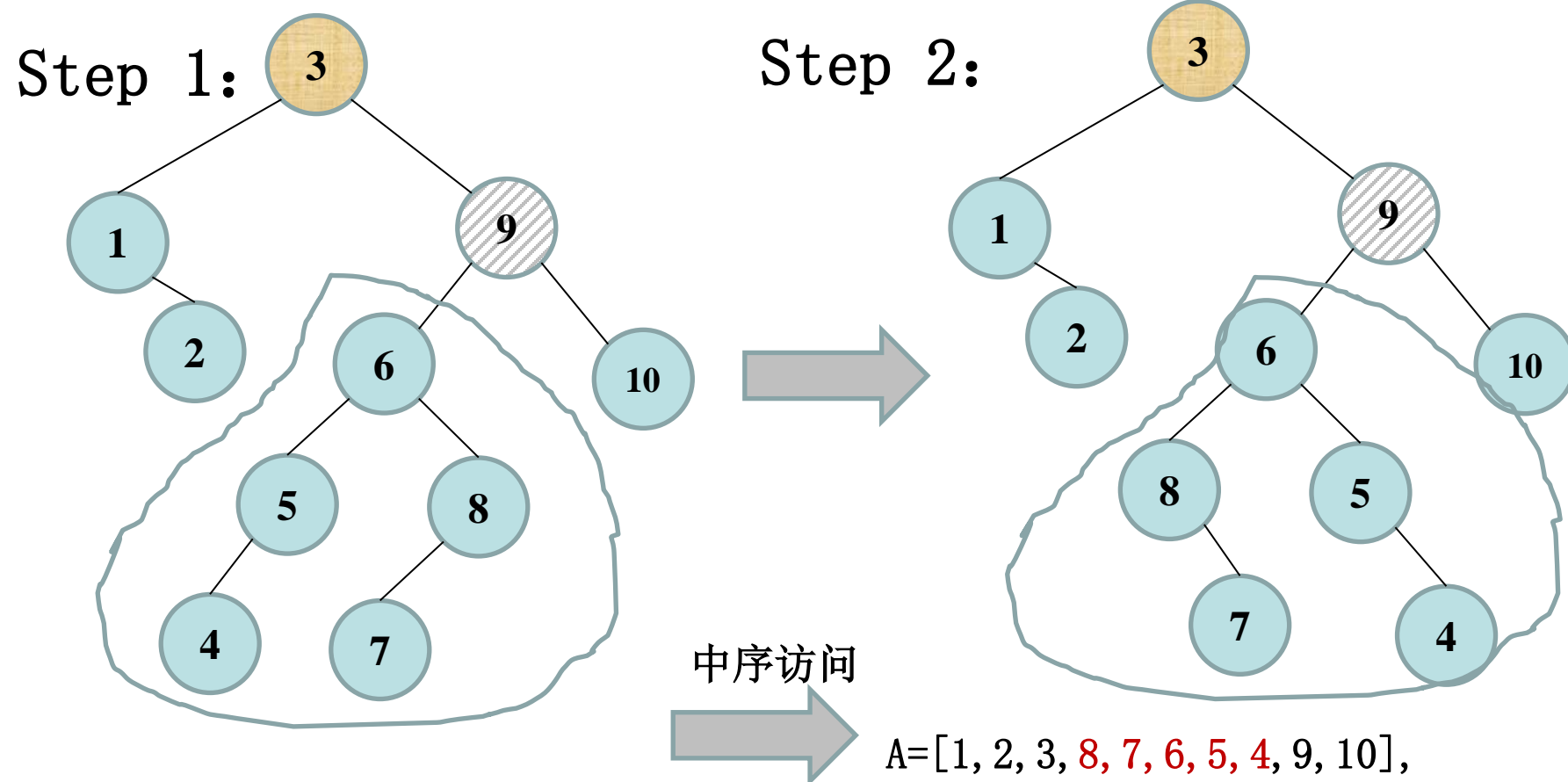
Step 1: 按照区间提取算法, 定位 $[n_a, n_b]$ 所对应的子树;

Step 2: 对此区间中的每个结点, 交换它的左右儿子。

# 基于伸展树的区间操作

## (3) 区间翻转

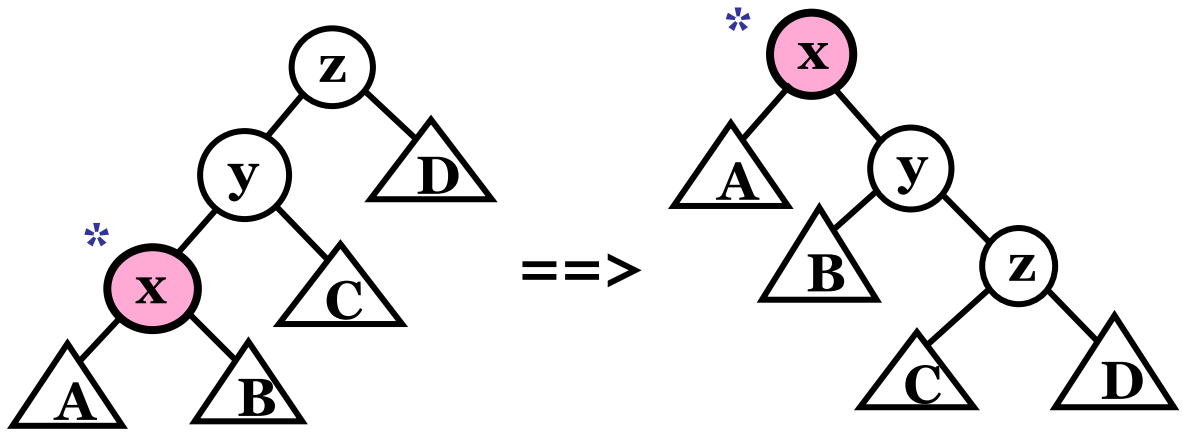
例如:  $A=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ , 翻转 $A[4-8]$ 部分。



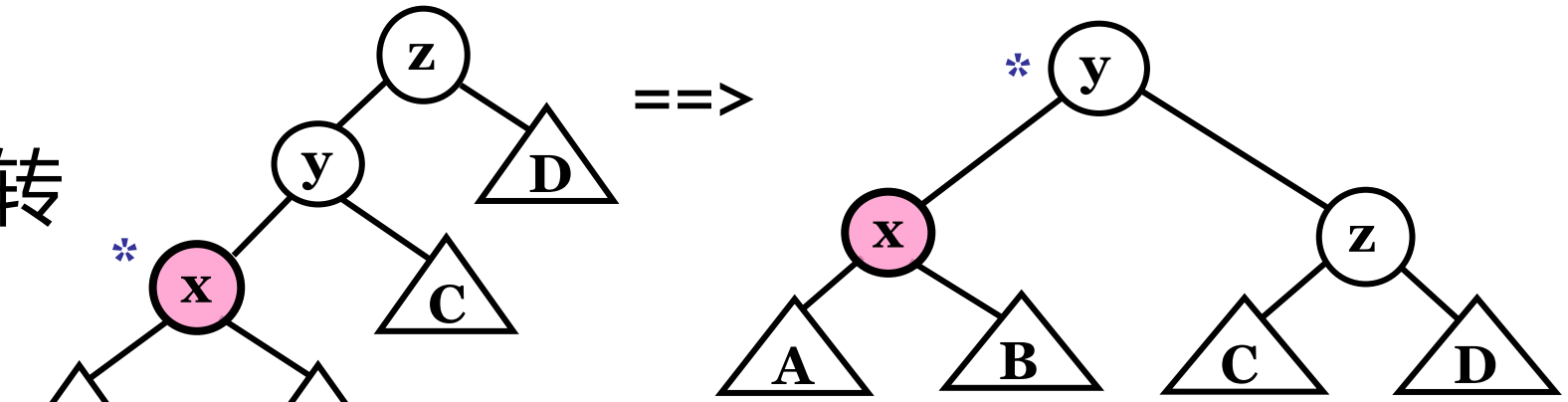
# 半伸展树

下一轮的伸展，当前结点上移到父结点，父结点位置开始旋转，即半伸展不需要把检索结点移动到根。

普通  
一字旋转



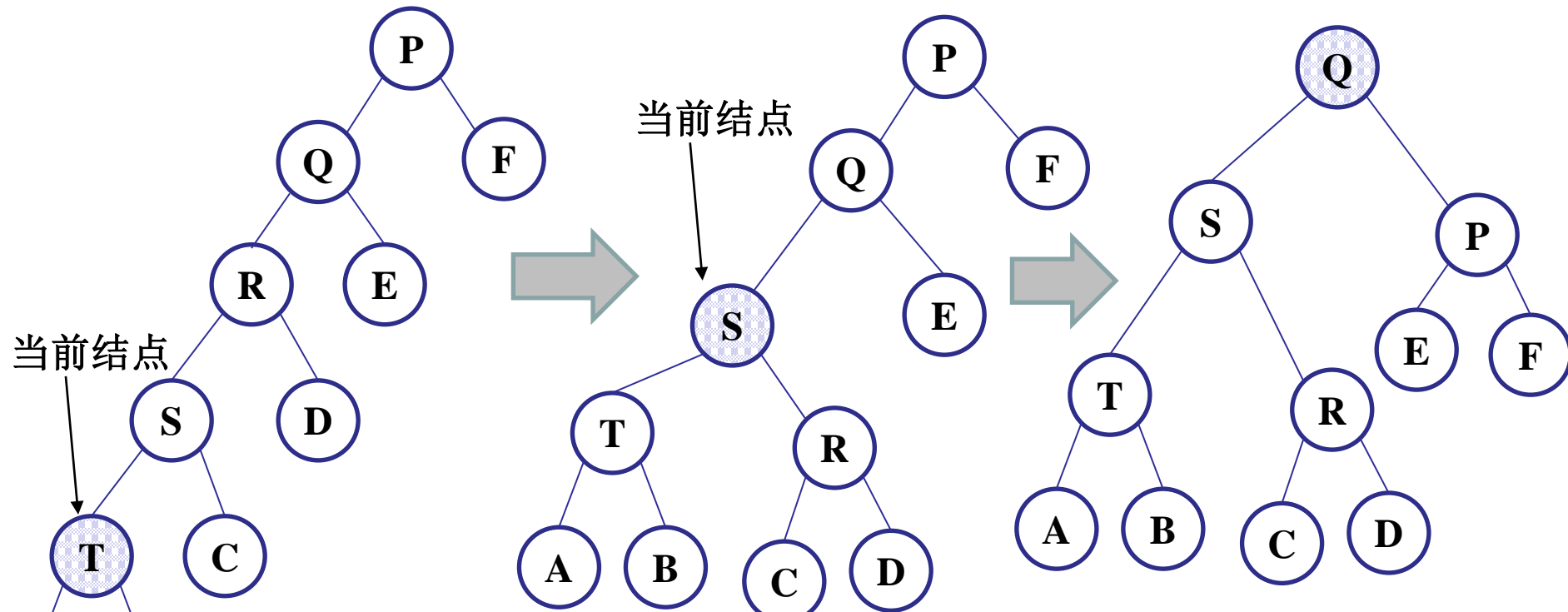
半伸展  
一字旋转



下一次旋转从 y 开始，而不从 x 开始

# 半伸展树

下一轮的伸展，当前结点上移到父结点，父结点位置开始旋转，即半伸展不需要把检索结点移动到根。

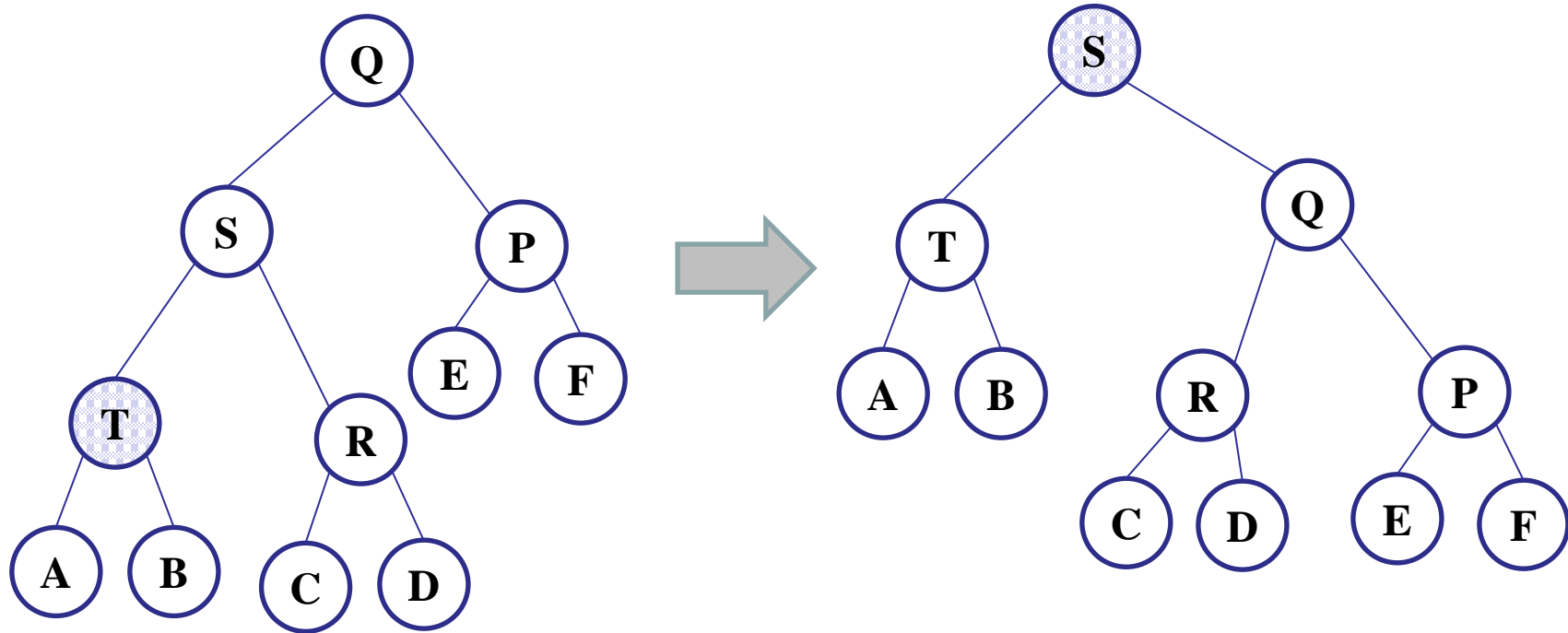


第一次访问T

# 半伸展树



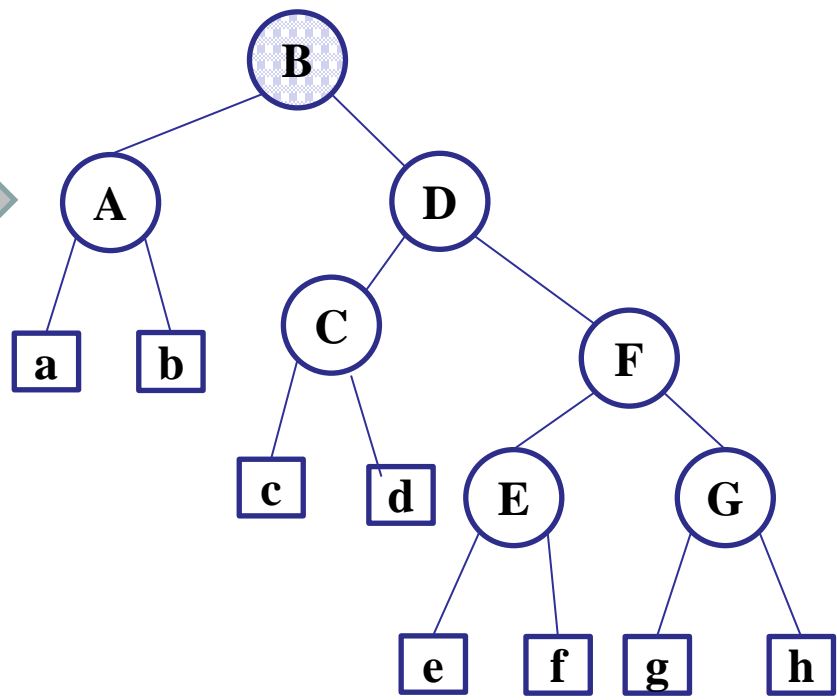
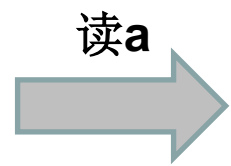
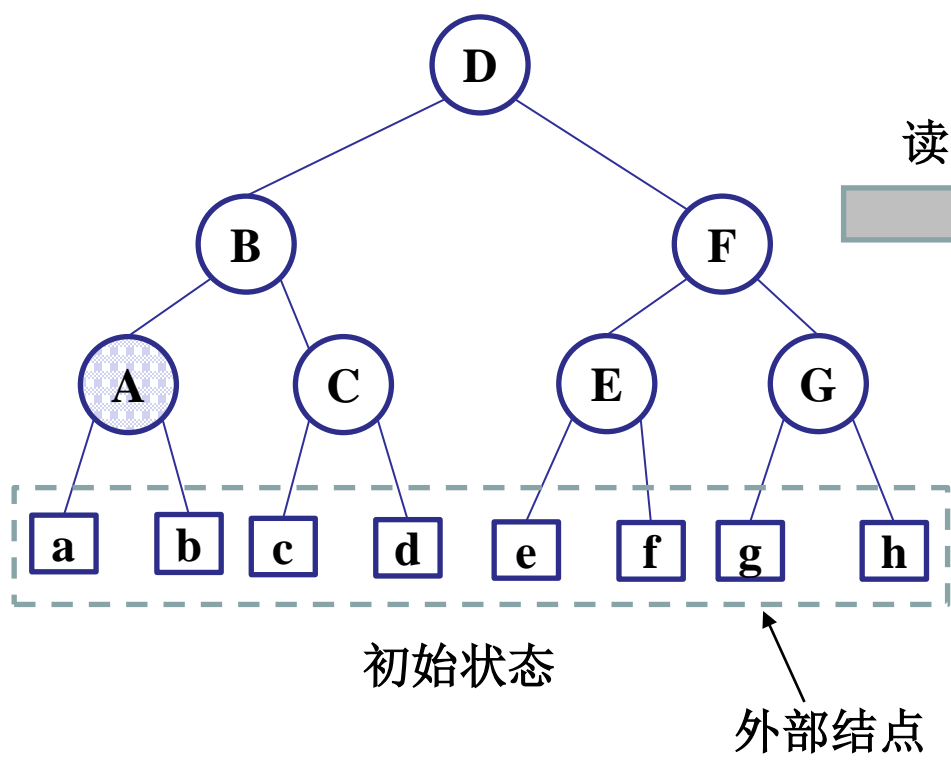
下一轮的伸展，当前结点上移到父结点，父结点位置开始旋转，即半伸展不需要把检索结点移动到根。



第二次访问T

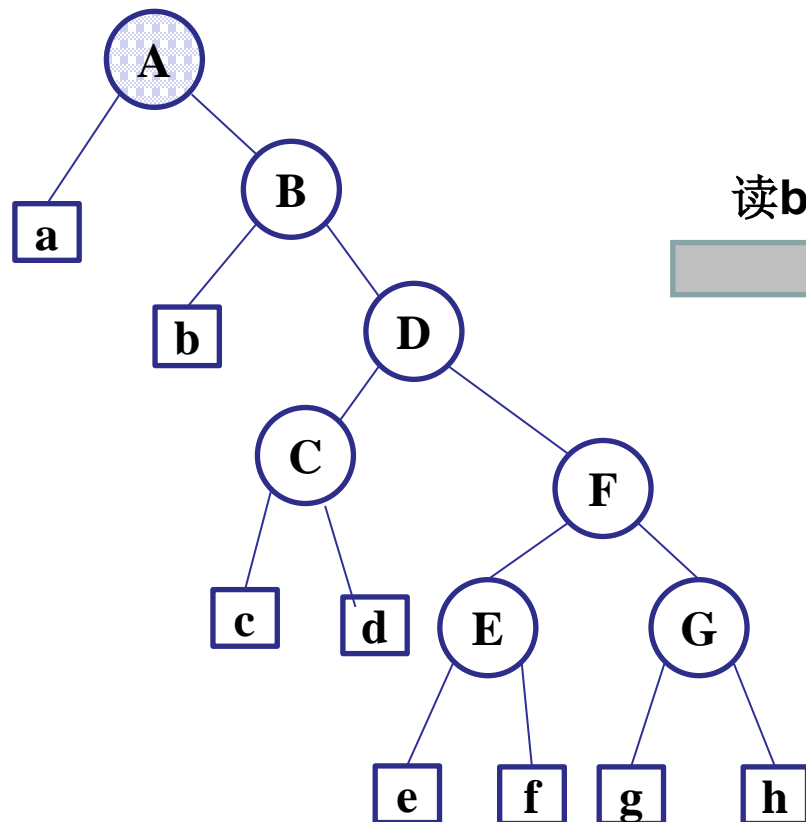
# 半伸展树

如果记录信息存储在外部结点，内部结点只作为索引结构；检索到某个外部结点时，其内部父结点为当前结点，然后开始调整。

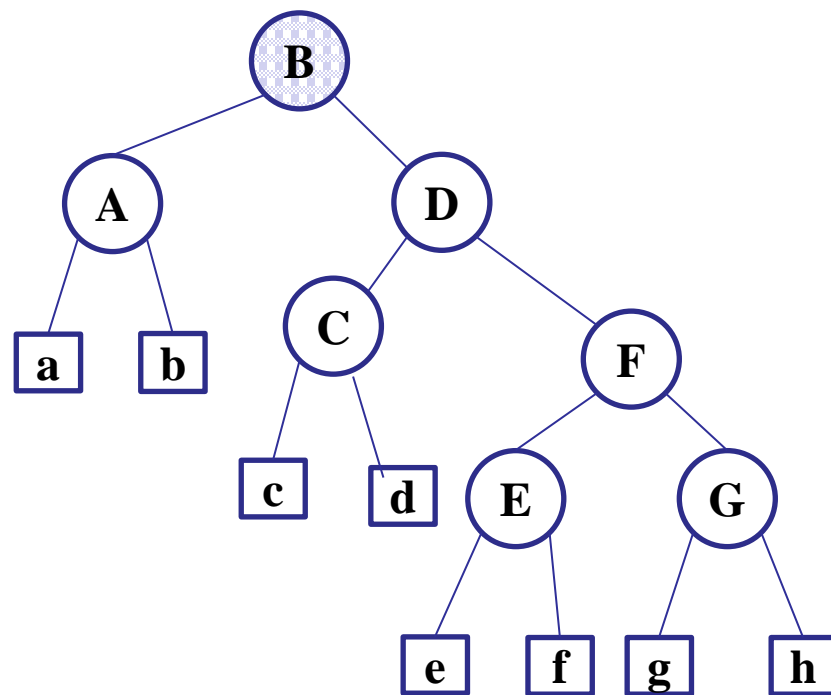
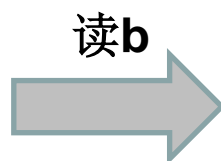


(a) 读a后

如果记录信息存储在外部结点，内部结点只作为索引结构；检索到某个外部结点时，其内部父结点为当前结点，然后开始调整。



(b) 继续读a后

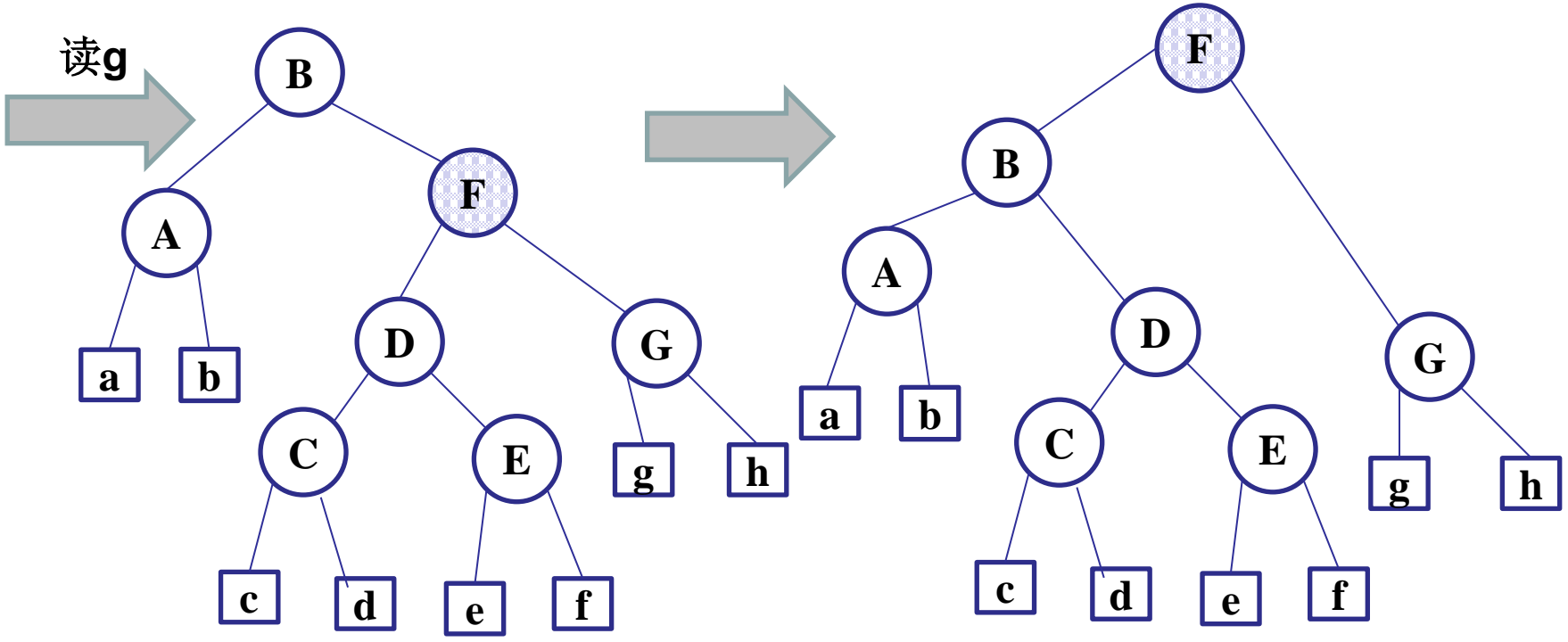


(c) 读b后



# 半伸展树

如果记录信息存储在外部结点，内部结点只作为索引结构；检索到某个外部结点时，其内部父结点为当前结点，然后开始调整。



中间状态

(d) 读g后

**Suffix Trie (后缀Trie):** 给定一个字符串T, 构建一棵面向T的所有后缀子串的Trie树。为了区分不同子串, 在每个子串后面都添加一个“\$”作为串尾!

T	b	a	n	a	n	a
	0	1	2	3	4	5

(1) 给定一个模式串P, 问P是否在T的子串。

例如: P=“ana”

(2) 给定一个模式串P, 问P是否是T的后缀子串。

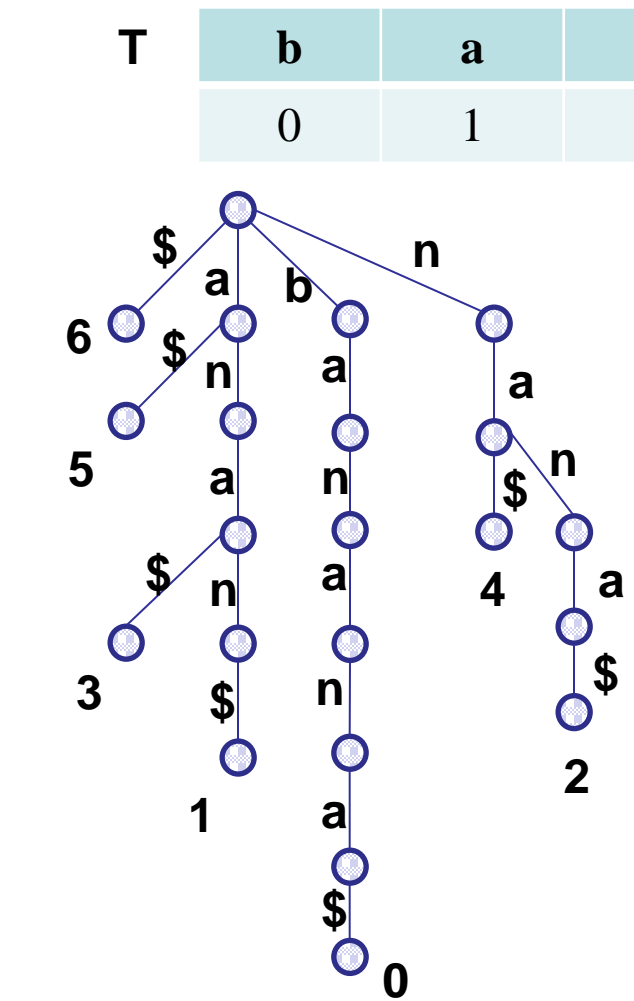
例如: P= “ana” , “anan”

(3) 给定一个模式串P, 问P在T串中出现的次数; 和它们出现的所有位置。

(4) 在T中找到最长的重复子串。

# 后缀树

**Suffix Trie (后缀Trie):** 给定一个字符串T, 构建一棵面向T的所有后缀子串的Trie树。为了区分不同子串, 在每个子串后面都添加一个“\$”作为串尾!



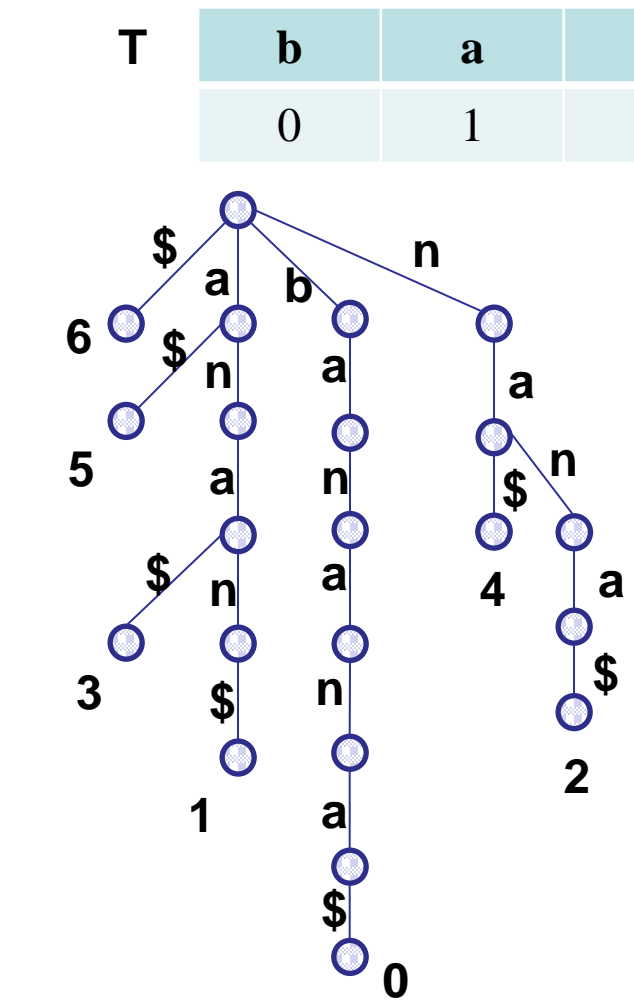
(1) 给定一个模式串P, 问P是否是T的子串。

例如: P="ana"

类似于之前的前缀树 (Trie) 中的判定前缀的方法:  
从root开始, 按照模式串P的字符, 逐层访问, 如果P中的所有字符都可以在这棵树中访问到 (不一定最终达到叶子节点), 则P是T的子串; 否则不是。

# 后缀树

**Suffix Trie (后缀Trie):** 给定一个字符串T, 构建一棵面向T的所有后缀子串的Trie树。为了区分不同子串, 在每个子串后面都添加一个“\$”作为串尾!

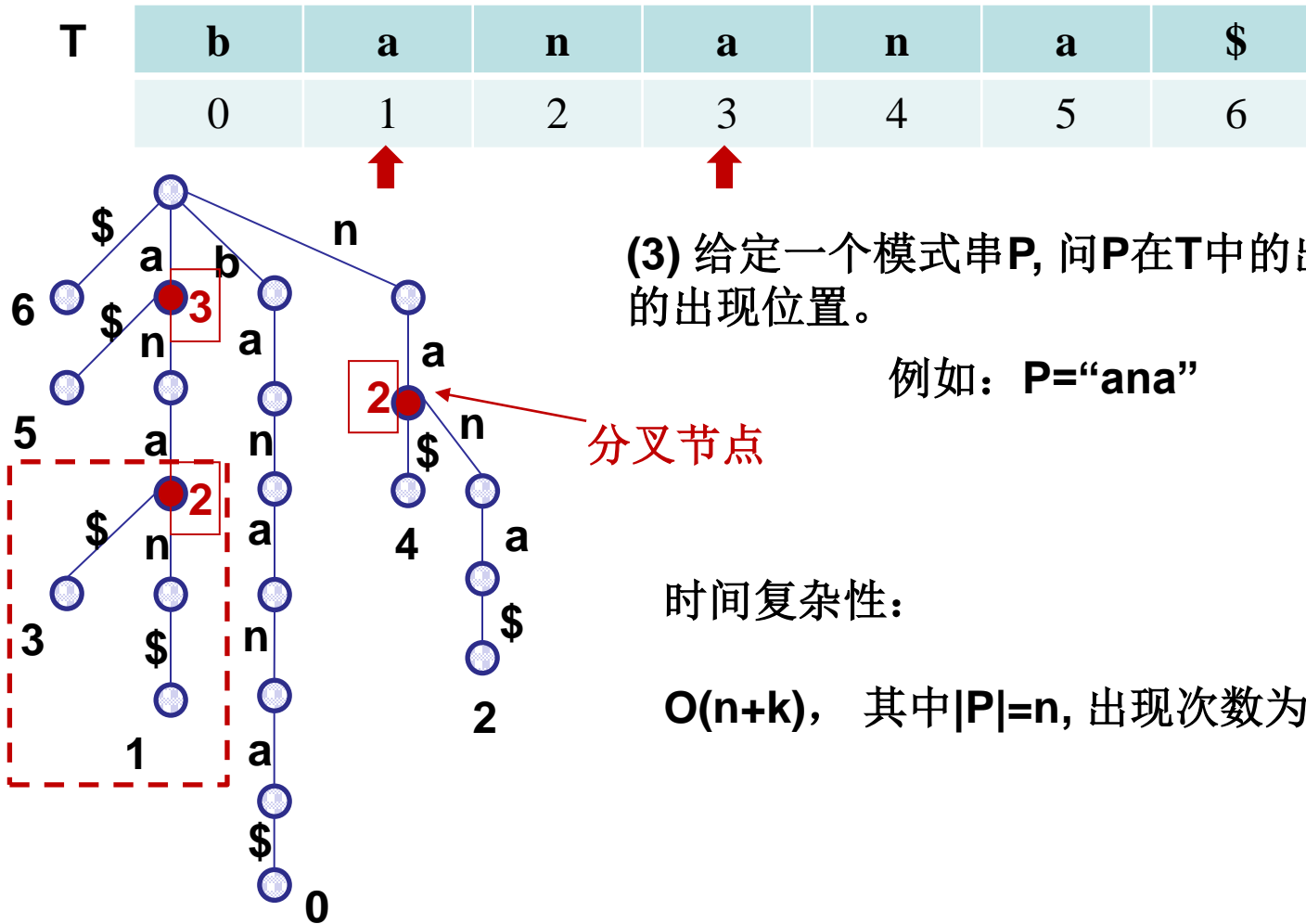


(2) 给定一个模式串P, 问P是否是T的**后缀**子串。

例如: P="ana", P= "anan"

类似于之前的前缀树 (Trie) 中的判定前缀的方法:  
**给P末尾添加 "\$"**, 然后从root开始, 按照模式树P的字符, 逐层访问, 直到访问到叶子节点, 则可以判定P是T的后缀子串; 否则不是。

**Suffix Trie**（后缀Trie）： 给定一个字符串T, 构建一棵面向T的所有后缀子串的Trie树。为了区分不同子串，在每个子串后面都添加一个“\$”作为串尾！



**(3) 给定一个模式串P, 问P在T中的出现次数, 和全部的出现位置。**

例如:  $P = \text{"ana"}$

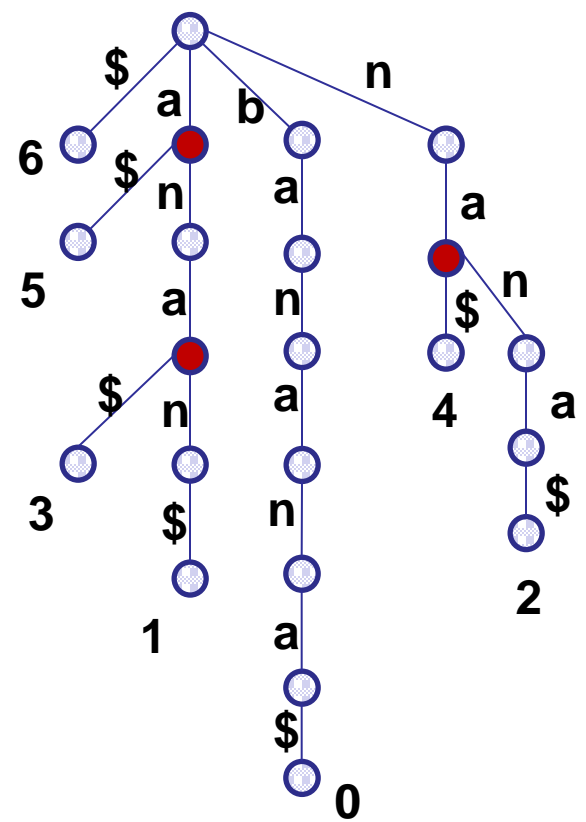
### 时间复杂性:

**$O(n+k)$ , 其中 $|P|=n$ , 出现次数为 $k$ 。**

# 后缀树

**Suffix Trie (后缀Trie):** 给定一个字符串T, 构建一棵面向T的所有后缀子串的Trie树。为了区分不同子串, 在每个子串后面都添加一个“\$”作为串尾!

T	b	a	n	a	n	a	\$
	0	1	2	3	4	5	6

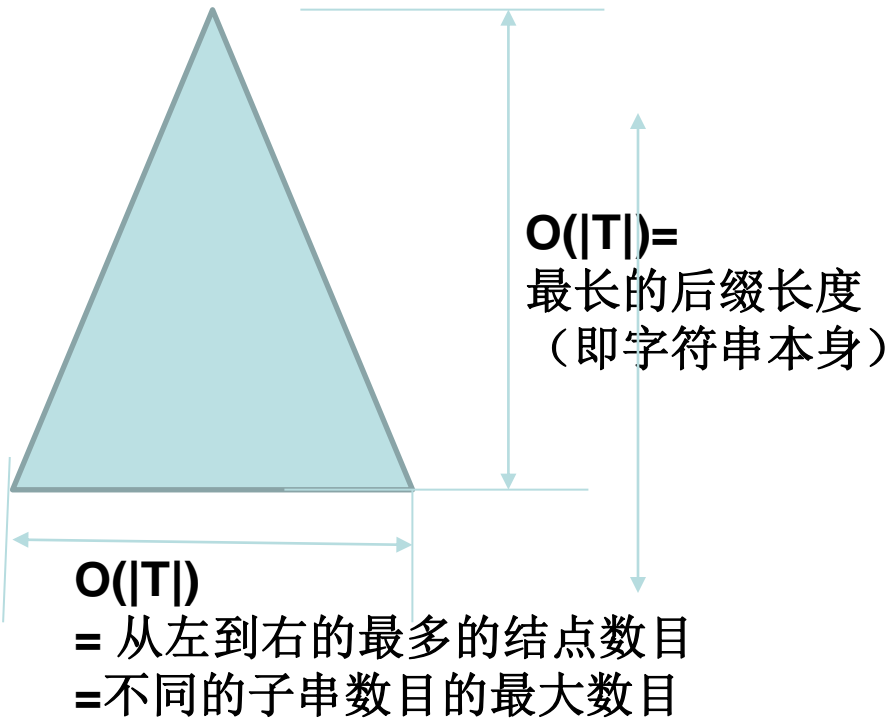


(4) 找到T中最长重复子串。

找到后缀Trie中, 最深的分叉节点。

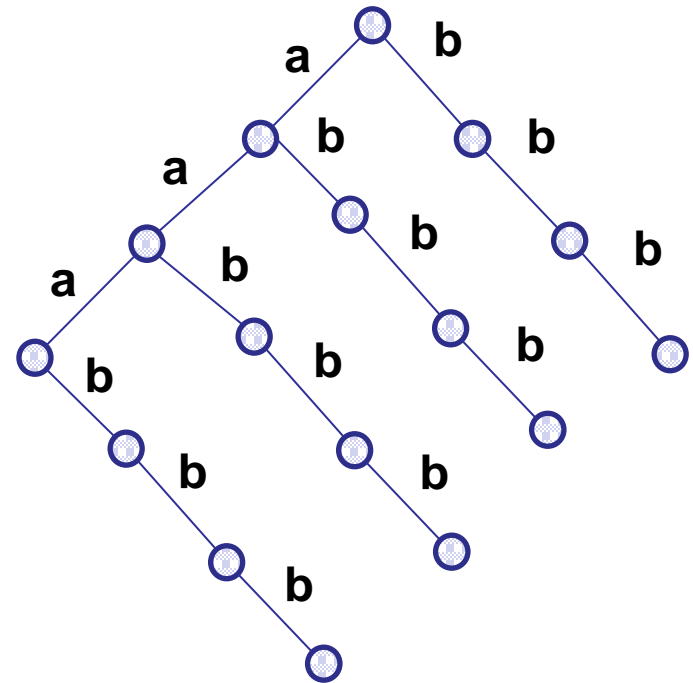
# 后缀树

## Suffix Trie (后缀Trie) 的空间大小



所以总的空间代价是：  
 $O(|T|^2)$

例子:  $T = aaabbb$



## Suffix Trie (后缀Trie) 的空间大小

### Suffix Trie $\rightarrow$ Suffix Tree

方法1: 将没有分叉的路径压缩成一条边, 边上的标签是对应的字符串;  
压缩以后的**Suffix Tree**, 设  $m = |T| - 1$  :

- (1) 有多少个叶子节点:  $m$
- (2) 有多少个非叶子节点:  $\leq m - 1$

证明: 设**Suffix Tree**中的节点数目为 $x$ , 边的数目为 $y$   
很显然叶子节点数目为  $m$ , 非叶子节点数目为  $x - m$

$$y = x - 1;$$

$$y \geq 2(x - m)$$

$$\Rightarrow x \leq 2m - 1$$

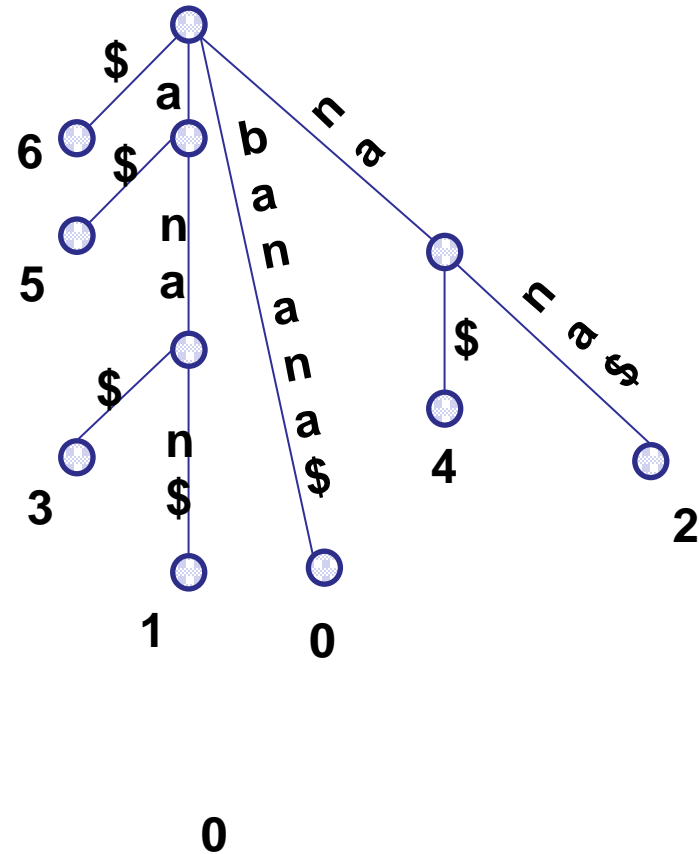
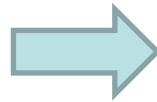
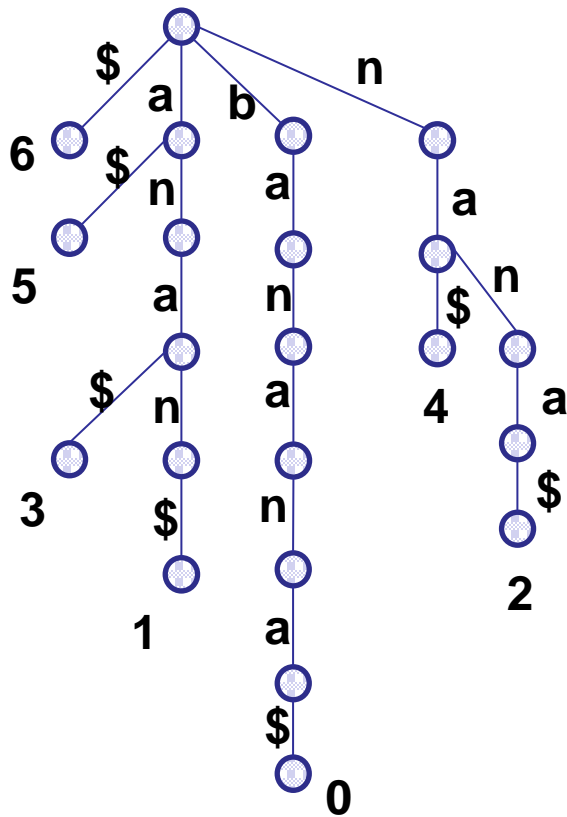
$$\text{所以 } (x - m) \leq m - 1$$



# 后缀树

## Suffix Trie (后缀Trie) 的空间大小

Suffix Trie → Suffix Tree

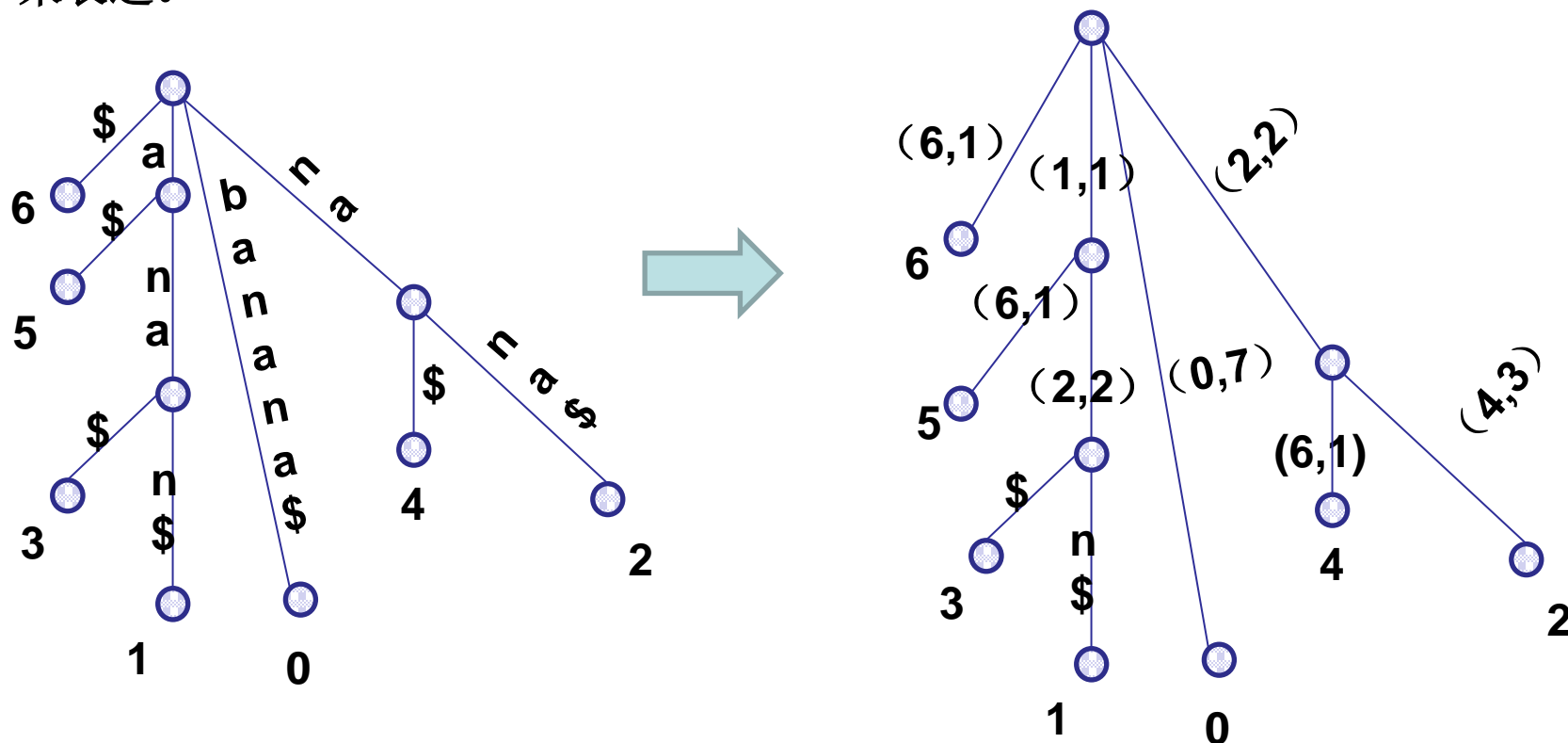


# 后缀树

## Suffix Trie (后缀Trie) 的空间大小

### Suffix Trie → Suffix Tree

方法2: 存储字符串T本身, 每条边上的label, 按照 (offset, length) 的方式来表达。



# 后缀树

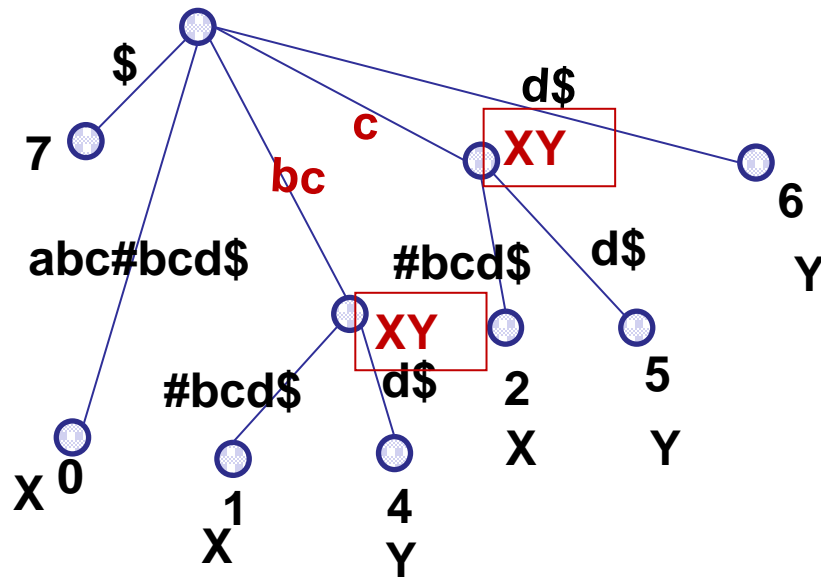
**Suffix Tree** 的应用：找两个字符串**X**和**Y**的公共子串

**X= abc**

**Y= bcd**

构建一个新的字符串 **X#Y\$ = a b c # b c d \$**

构建这个字符串的后缀树



A decorative graphic consisting of overlapping green, blue, and yellow squares with a black crosshair.

# 再见…

---

联系信息:

电子邮件: [zoulei@pku.edu.cn](mailto:zoulei@pku.edu.cn)

电 话: 82529643