第十二章　高级数据结构

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 （“十二五”国家级规划教材）

http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

第十二章 高级数据结构

- 12.1 多维数组
  - 多维数组的一般性质
  - 特殊的二维矩阵
  - 稀疏矩阵的实现与操作
- 12.2 广义表
- 12.3 存储管理
- 12.4 Trie 树
- 12.5 AVL树的概念与插入操作
- 12.6 AVL树的删除操作与性能分析
- 12.7 伸展树

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才分配元素个数与类型

基本概念 (续)

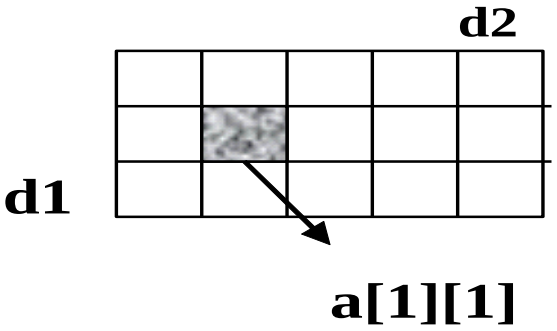
- 多维数组（Multi-array）是一般数组的扩充
- 数组的数组就组成了多维数组，可以表示为：

$$\text{ELEM } A[c_1..d_1][c_2..d_2]...[c_n..d_n]$$

- c_i 和 d_i 是各维下标的下界和上界。所以其元素个数为：

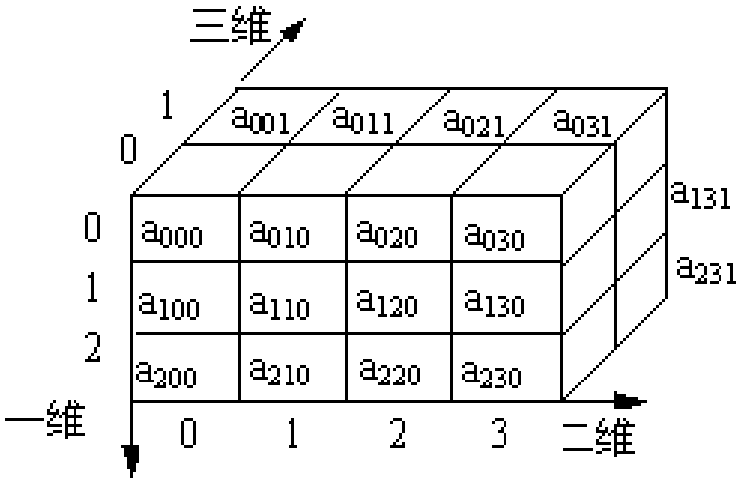
$$\prod_{i=1}^{n}(d_i - c_i + 1)$$

数组的空间结构



二维数组

第一维：c1=0，d1=2
第二维：c2=0，d2=4



三维数组

第一维：c1=0，d1=2
第二维：c2=0，d2=3
第三维：c3=0，d3=1

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
  - 以行为主序(也称为“行优先”)
  - 以列为主序(也称为“列优先”)

$$X=\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}$$

行优先存储

a_{111} a_{112} a_{113} … a_{11n} a_{11*}

a_{121} a_{122} a_{123} … a_{12n} a_{12*}

……

a_{1m1} a_{1m2} a_{1m3} … a_{1mn} a_{1m*}

a_{211} a_{212} a_{213} … a_{21n} a_{21*}

a_{221} a_{222} a_{223} … a_{22n} a_{22*}

……

a_{2m1} a_{2m2} a_{2m3} … a_{2mn} a_{2m*}

⋮

a_{k11} a_{k12} a_{k13} … a_{k1n}

a_{k21} a_{k22} a_{k23} … a_{k2n}

……

a_{km1} a_{km2} a_{km3} … a_{kmn}

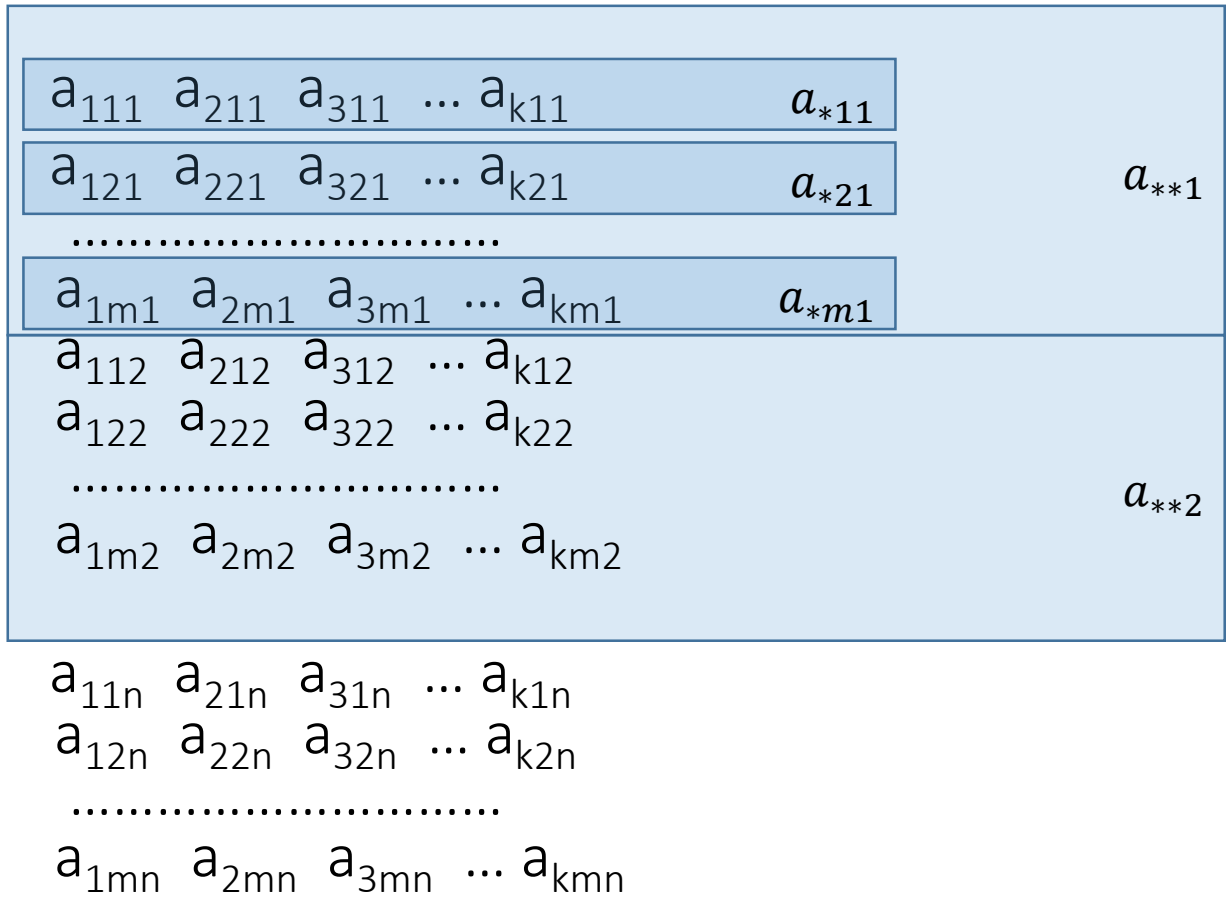
a[1..k,1..m,1..n]

典型编程语言：

Pascal、C/C++、JAVA

12.1 多维数组

列优先存储

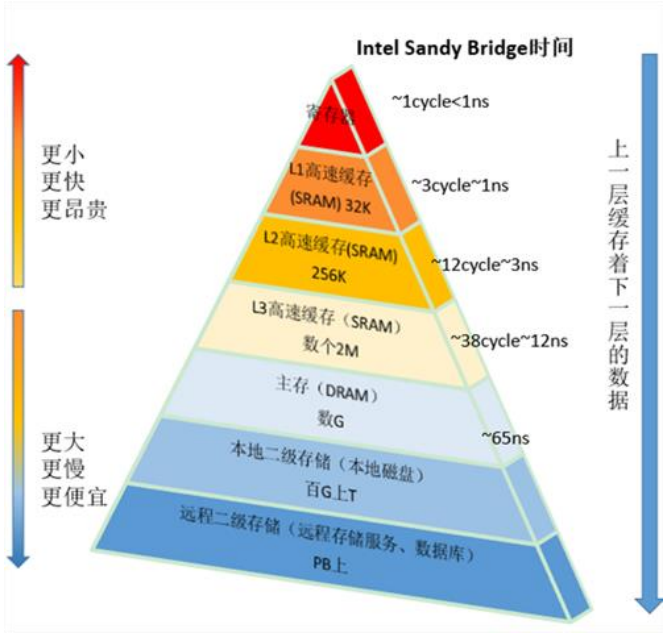


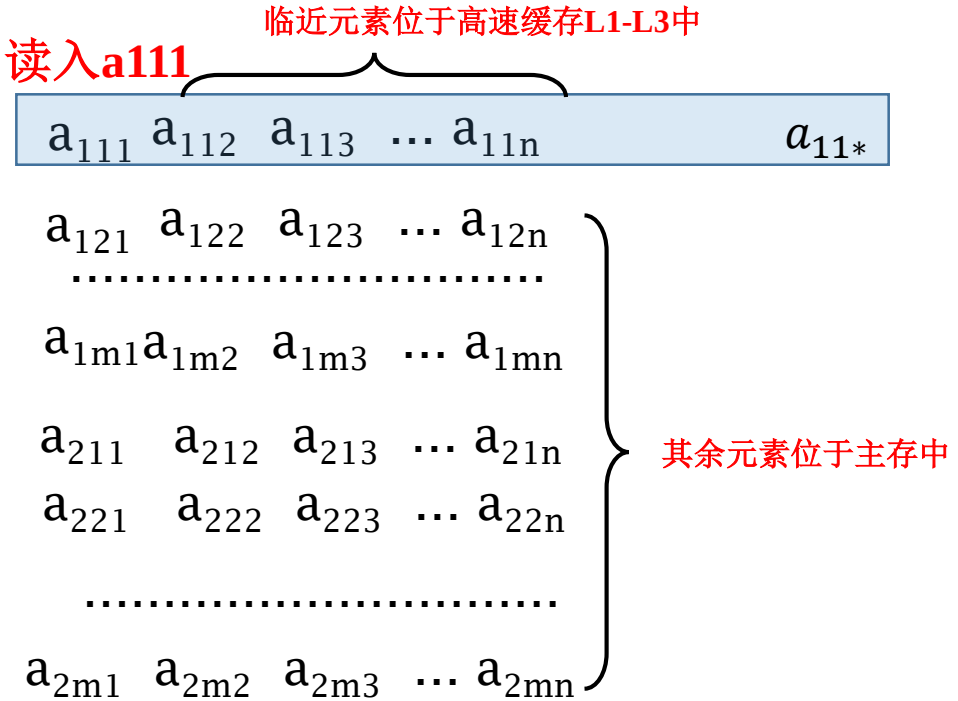
a[1..k, 1..m, 1..n]

典型编程语言：
Matlab、FORTRAN

行优先与列优先对性能的影响

- 计算机缓存系统
  - 读某个内存地址，会将该地址附近数据一次性读入，存放于缓存(L1-L3)中



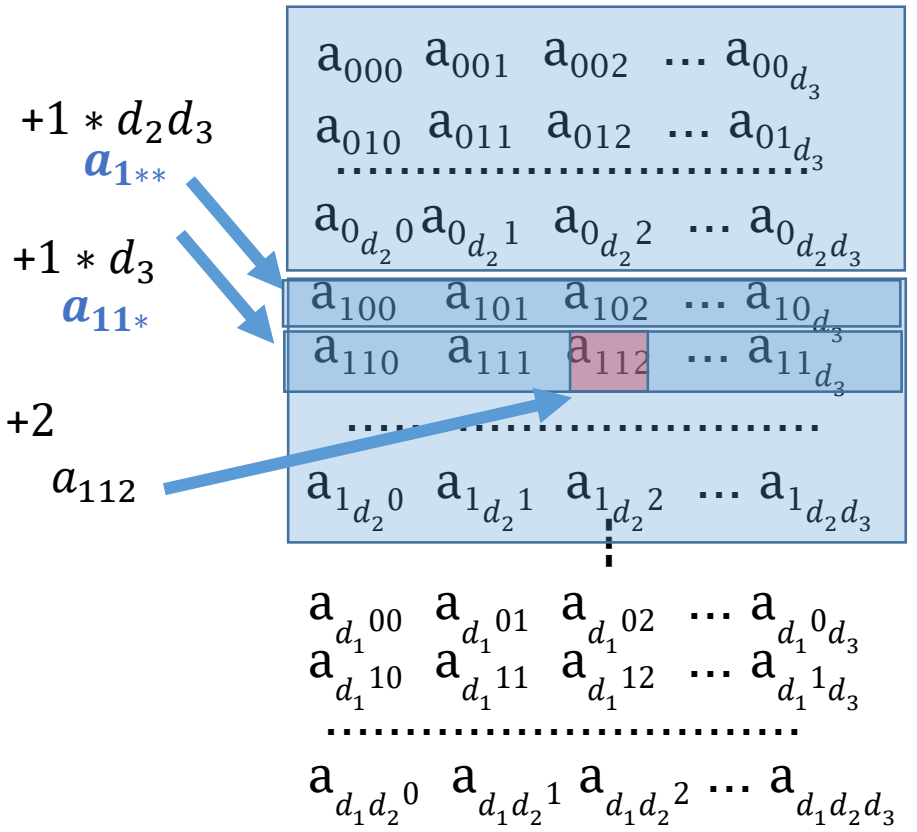


定位多维数组元素

- 多维数组ELEM A[d_1][d_2]...[d_n]，每个元素大小为d
- 采用行优先存储

$$loc(A[j_1, j_2, \ldots, j_n])$$
$$= loc(A[0,0,\ldots,0]) + d \cdot [j_1 \cdot d_2 \cdot \ldots \cdot d_n + j_2 \cdot d_3 \cdot \ldots \cdot d_n + \ldots + j_{n-1} \cdot d_n + j_n]$$
$$= loc(A[0,0,\ldots,0]) + d \cdot [\sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} d_k + j_n]$$

- 元素大小d=1，定位元素a_{112}



用数组表示特殊矩阵

- 三角矩阵：上三角、下三角
- 对称矩阵
- 对角矩阵
- 稀疏矩阵

下三角矩阵图例

- 一维数组 list[0.. $(n^2+n)/2-1$]
  - 矩阵元素 $a_{i,j}$ 与线性表相应元素的对应位置为 list[$(i^2+i)/2 + j$] (i>=j)

$$\begin{pmatrix} 0 & & & & & \\ 0 & 0 & & & & \\ 7 & 5 & 0 & & & \\ 0 & 0 & 1 & 0 & & \\ 9 & 0 & 0 & 1 & 8 & \\ 0 & 6 & 2 & 2 & 0 & 7 \end{pmatrix}$$

对称矩阵

- 元素满足性质 $a_{i,j}= a_{j,i}$，$0<= (i, j) <n$

  例如，右图的无向图相邻矩阵

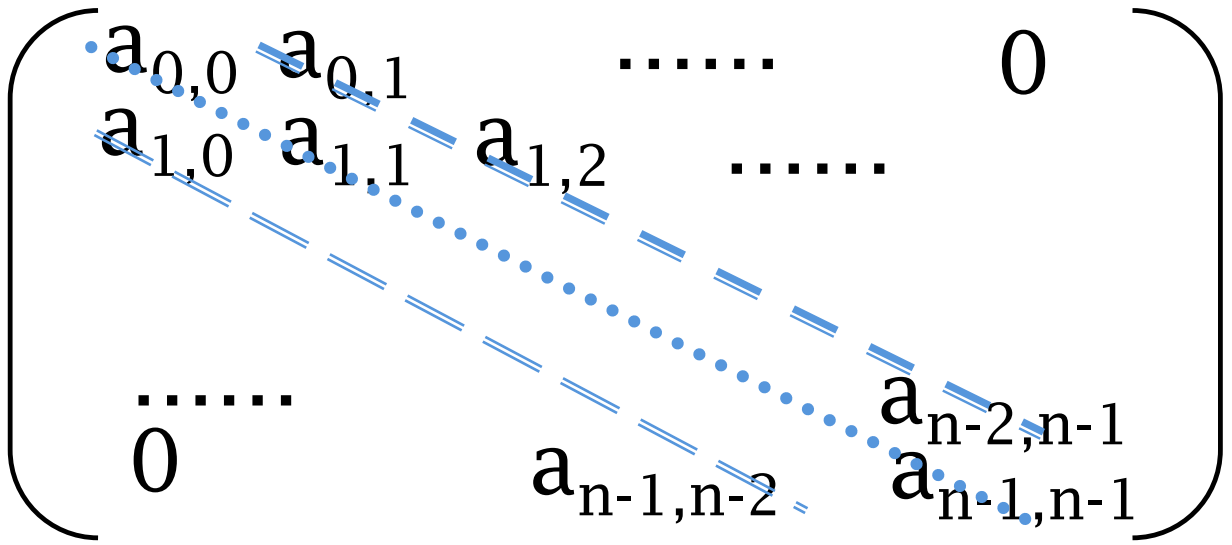
$$\begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$

- 存储其下三角的值，对称关系映射
- 存储于一维数组 sa[0..n (n+1) /2-1]
  - sa[k] 和矩阵元 $a_{i,j}$ 之间存在着一一对应的关系：

$$k = \begin{cases} j(j+1)/2 + i, \text{当} i < j \\ i(i+1)/2 + j, \text{当} i \geq j \end{cases}$$

对角矩阵

- 对角矩阵是指：所有非零元素都集中在主对角线及以它为中心的其他对角线上
- 下面是一个三对角矩阵：如果 |i-j| > 1，那么数组元素 a[i][j] = 0



稀疏矩阵

- 稀疏矩阵中的非零元素非常少，而且分布也不规律

$$A_{6\times7}=\begin{pmatrix}0&0&0&0&0&0&5\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\0&0&0&0&0&0&0\\11&0&0&0&0&0&0\\0&0&0&0&0&0&0\end{pmatrix}$$

稀疏矩阵

- 稀疏因子
  - 在 m×n 的矩阵中，有 t 个非零元素，则稀疏因子为:

$$\delta = \frac{t}{m \times n}$$

  - 当这个值小于0.05时，可以认为是稀疏矩阵
- 三元组（i, j, a_{ij}）：输入/输出常用
  - i 是该元素的行号
  - j 是该元素的列号
  - a_{ij} 是该元素的值

稀疏矩阵的作用

- 简化计算：跳过矩阵中大量的零元素

- 例：稀疏矩阵乘法

  A[c1..d1][c3..d3]，B[c3..d3][c2..d2], C[c1..d1][c2..d2]

  $C = A \times B \quad (C_{ij} = \sum_{k=c3}^{d3} A_{ik} \cdot B_{kj})$

经典矩阵乘法时间代价

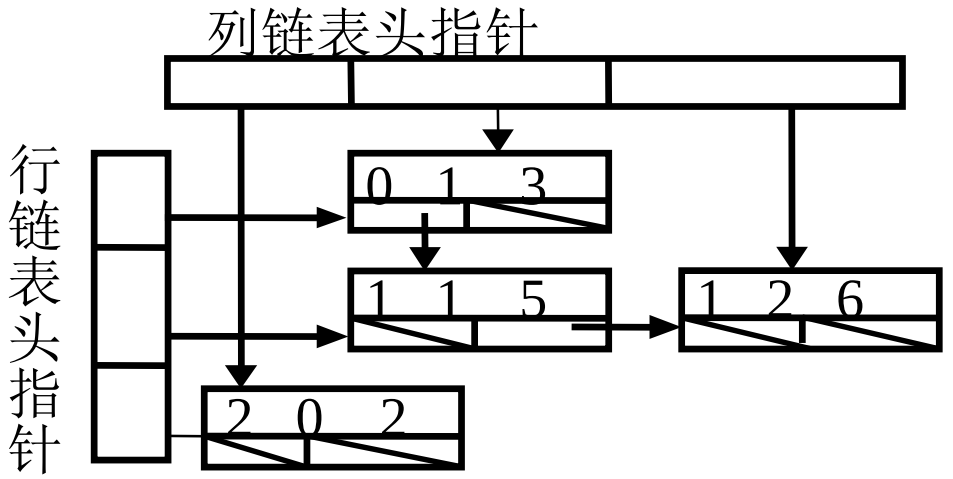
- p=d1-c1+1，m=d3-c3+1，n=d2-c2+1;
- A 为 p×m 的矩阵，B 为 m×n 的矩阵，乘得的结果 C为 p×n 的矩阵
- 经典矩阵乘法所需要的时间代价为 O (p×m×n)

```
for  (i=c1; i<=d1; i++)
    for  (j=c2; j<=d2; j++){
        sum = 0;
        for  (k=c3; k<=d3; k++)
            sum = sum + A[i,k]*B[k,j];
        C[i, j] = sum;
    }
```

稀疏矩阵的十字链表表示法

- 稀疏矩阵的表示：需要便于计算
  - 特别是处理遍历操作
- 十字链表有两组链表组成
  - 行和列的指针序列
  - 每个结点都包含两个指针：同一行的后继，同一列的后继

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 5 & 6 \\ 2 & 0 & 0 \end{bmatrix}$$

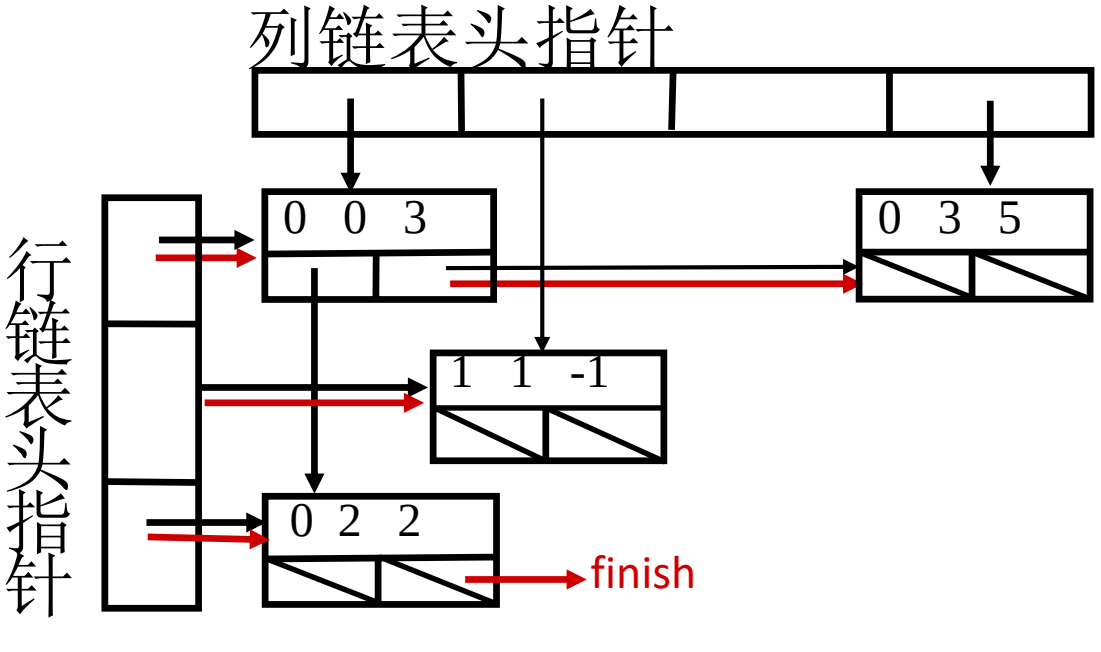


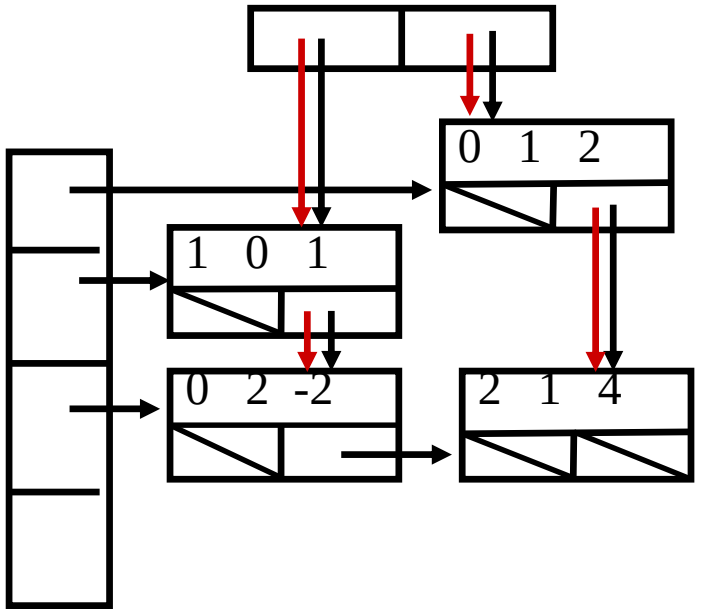
稀疏矩阵乘法

- 遍历所有（A矩阵行链表，B矩阵列链表）链表对
- 对每个（第 i 行链表，第 j 列链表）链表对
  - 第 i 行链表元素为（i, k1, v1）
  - 第 j 列链表元素为（k2, j, v2）
  - 比对行链表元素（i, k1, v1）与列链表元素（k2, j, v2）
  - 若k1 == k2，则将v1*v2加到输出矩阵的第 i 行 j 列对应元素

稀疏矩阵乘法

$$\begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{bmatrix}$$





稀疏矩阵乘法时间代价

- A为 p×m 的矩阵，B 为 m×n 的矩阵，乘得的结果 C为 p×n 的矩阵
  - 若矩阵 A 中行向量的非零元素个数最多为 t_a
  - 矩阵 B 中列向量的非零元素个数最多为 t_b
- 总执行时间降低为 O ((t_a+t_b) ×p×n)
- 经典矩阵乘法所需要的时间代价为 O (p×m×n)

稀疏矩阵的应用

- 机器学习中的稀疏矩阵
  - 自然语言处理：文本文档表示
    - *如果在语言模型中有100,000个单词，那么特征向量长度为100,000*
    - *但是对于一个简短的电子邮件来说，几乎所有的特征都是0*
  - 推荐系统：用户与物品表示

	物品1	物品2	物品3	物品4	物品5	物品6	物品7	物品8	物品9	物品10
用户1	3					5			2	
用户2			3		5			2		
用户3		1		2			5			
用户4			3					3		5
用户5	5				2					

  - 图像处理：背景像素（black pixel）

思考

- 多元多项式如何使用矩阵表示？如何使用稀疏矩阵进行存储？

一元多项式 $$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

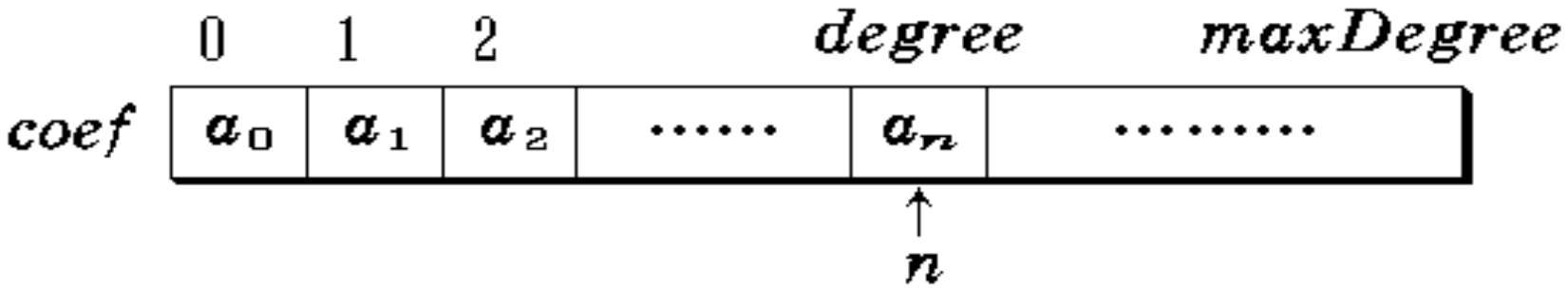
$$= \sum_{i=0}^{n} a_i x^i$$

- 如何用十字链表结构求解数独 (sudoku) 问题？

稀疏矩阵的应用

一元多项式

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i$$



思考：多元多项式的表达

- $P(x, y)=x^5y^3 + 2x^4y^3 + 3x^4y^2 + x^4y^4 + 6x^3y^4 + 2y$

	X0	X1	X2	X3	X4	X5
Y0	0	0	0	0	0	0
Y1	2	0	0	0	0	0
Y2	0	0	0	0	3	0
Y3	0	0	0	0	2	1
Y4	0	0	0	6	1	0

把P(x , y)重新写作:

$P(x, y)=((x^4+6x^3)y^4+(x^5+2x^4)y^3 +3x^4y^2+2y$

思考：多元多项式的表达

- $P(x, y, z)=x^{10}y^3z^2+2x^8y^3z^2+3x^8y^2z^2+x^4y^4z+6x^3y^4z+2yz$

把P(x , y , z)重新写作:

$P(x, y, z)=((x^{10}+2x^8)y^3+3x^8y^2)z^2+((x^4+6x^3)y^4+2y)z$

多维数组、广义表

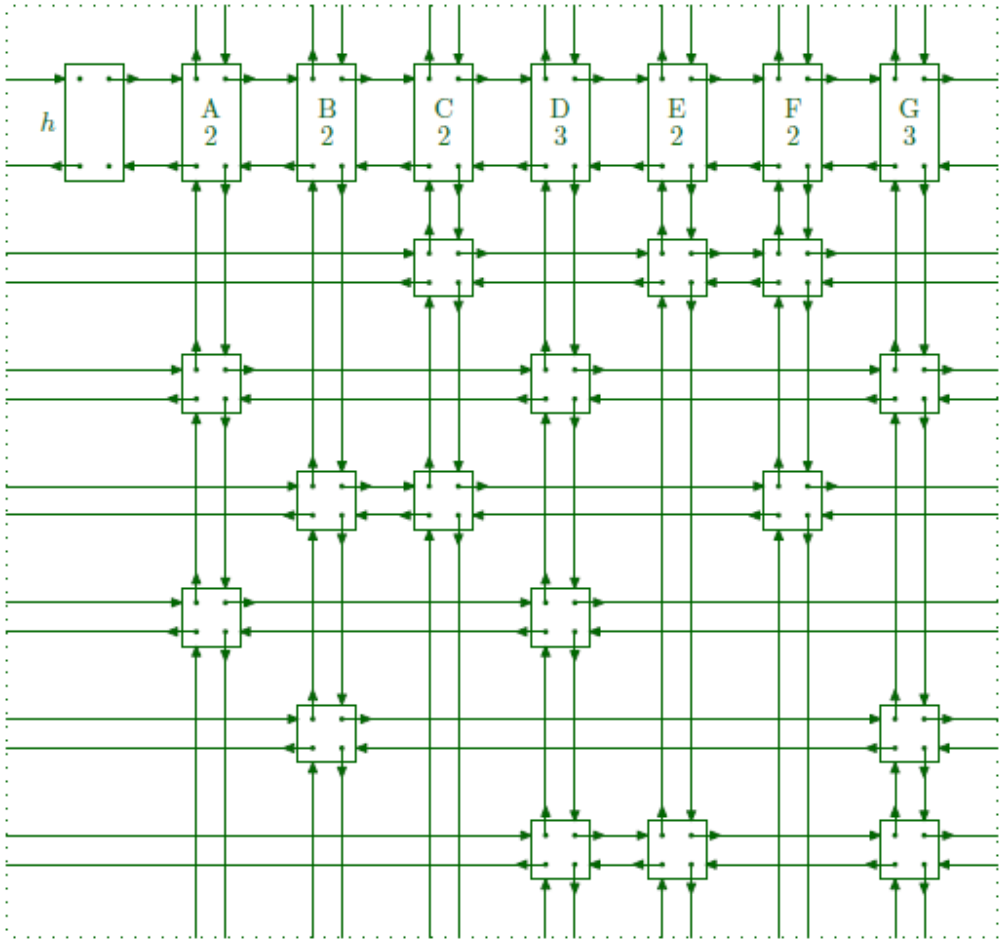
数独游戏

- Sudoku就是我们平常提到的数独。数独分为2阶、3阶、以及高阶数独。
- 最简单的是2阶，我们平常接触的是3阶，再次更高的有4阶、5阶甚至更高。

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

			14	13		6		1			9		5		8
			7			11	5		10	16		1			
			1			8	7		3				6		12
3	11	10	9		14					6				2	
			2	1		3		5					4		15
5	12					2	11			1	8		16		
		16	15				4		12			10		14	9
			10	15	12				2	13					11
4					6	12				7	2	16			
16	3		12			5		8				2	15		
		15		9	4			16						1	13
2		6					16		15		1	8			
	7				16					8		5	10	12	3
10		4				1		9	13			6			
			8		15	4		7	5			14			
15		1		10			8		6		16	7			

Dancing Links



第十二章 高级数据结构

- 12.1 多维数组
- **12.2 广义表**
  - **广义表的逻辑概念**
  - **广义表的实现与操作**
- 12.3 存储管理
- 12.4 Trie 树
- 12.5 AVL树的概念与插入操作
- 12.6 AVL树的删除操作与性能分析
- 12.7 伸展树

基本概念

- 回顾线性表
  - 由 n (n≥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
- 如果一个线性表中还包括一个或者多个子表，那就称之为广义表(Generalized Lists，也称Multi-list) 一般记作：

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

广义表的定义

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

- 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）
- 广义表的 **深度**：表中元素都化解为原子后的最大括号层数

广义表的基本操作

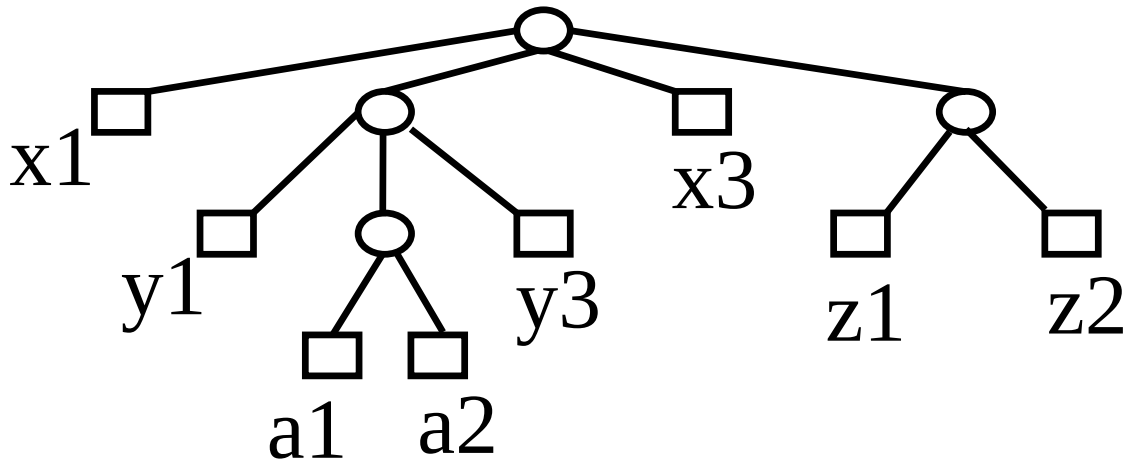
$L = (x_0, x_1, \ldots, x_i, \ldots, x_{n-1})$

- 表头 $head = x_0$
- 表尾 $tail = (x_1, \ldots, x_{n-1})$
  - 规模更小的表
- 有利于存储和实现

广义表的各种类型

- 纯表 (pure list)
  - 任何一个成员(原子、子表) 在广义表中只出现一次
  - 从根结点到任何叶结点只有一条路径

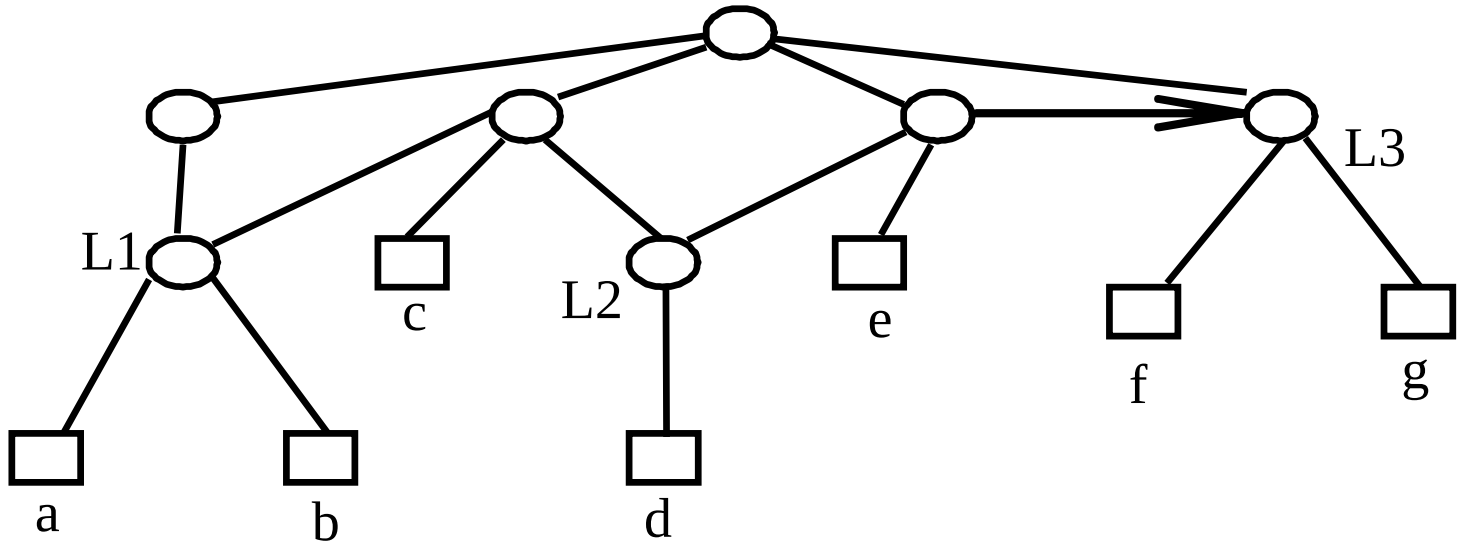
(x1, (y1 , (a1 ,a2) , y3) , x3 , (z1 ,z2))



广义表的各种类型 (续)

- 可重入表
  - 其元素 (包括原子和子表) 可能会在表中多次出现
  - 如果没有回路，则图示对应于一个 DAG
- 对子表和原子标号

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

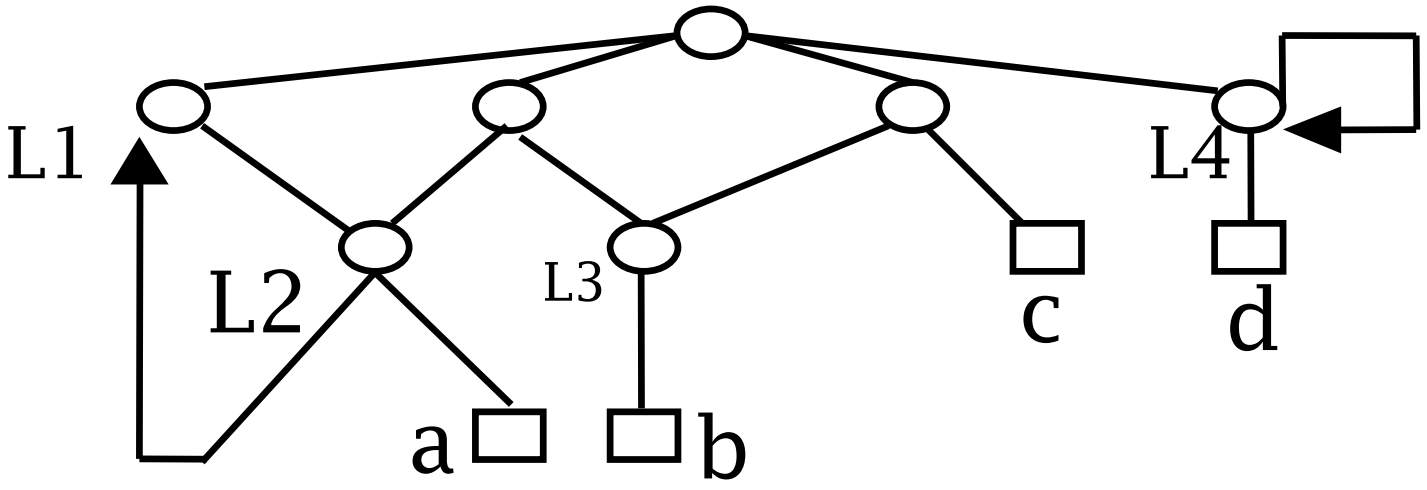


((L1: (a,b)) , (L1, c ,L2: (d)) , (L2, e,L3: (f,g)) , L3)

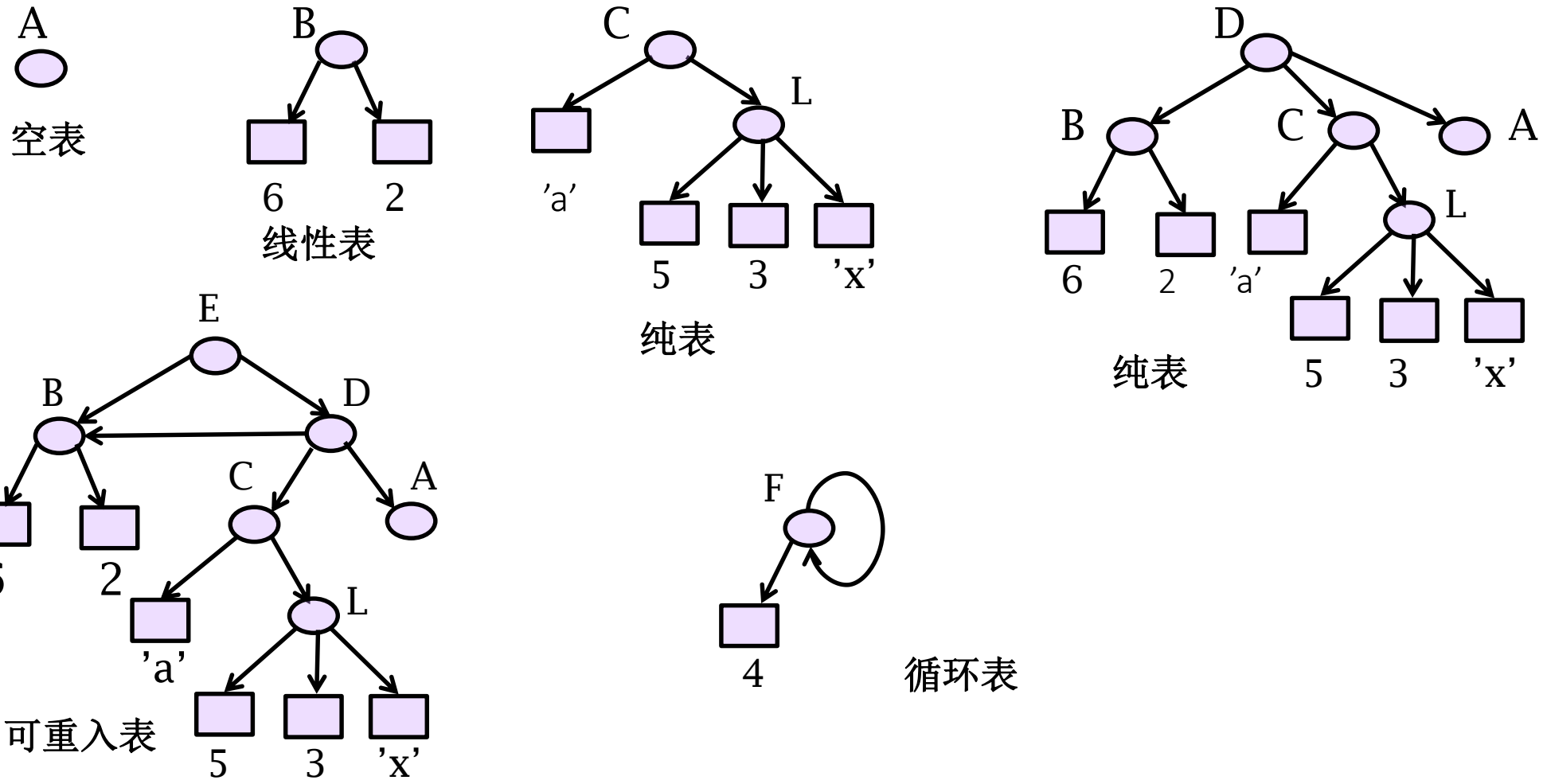
广义表的各种类型 (续)

- 循环表（递归表）：可重入表的特例
  - 包含回路
  - 循环表的深度为无穷大

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



12.2 广义表

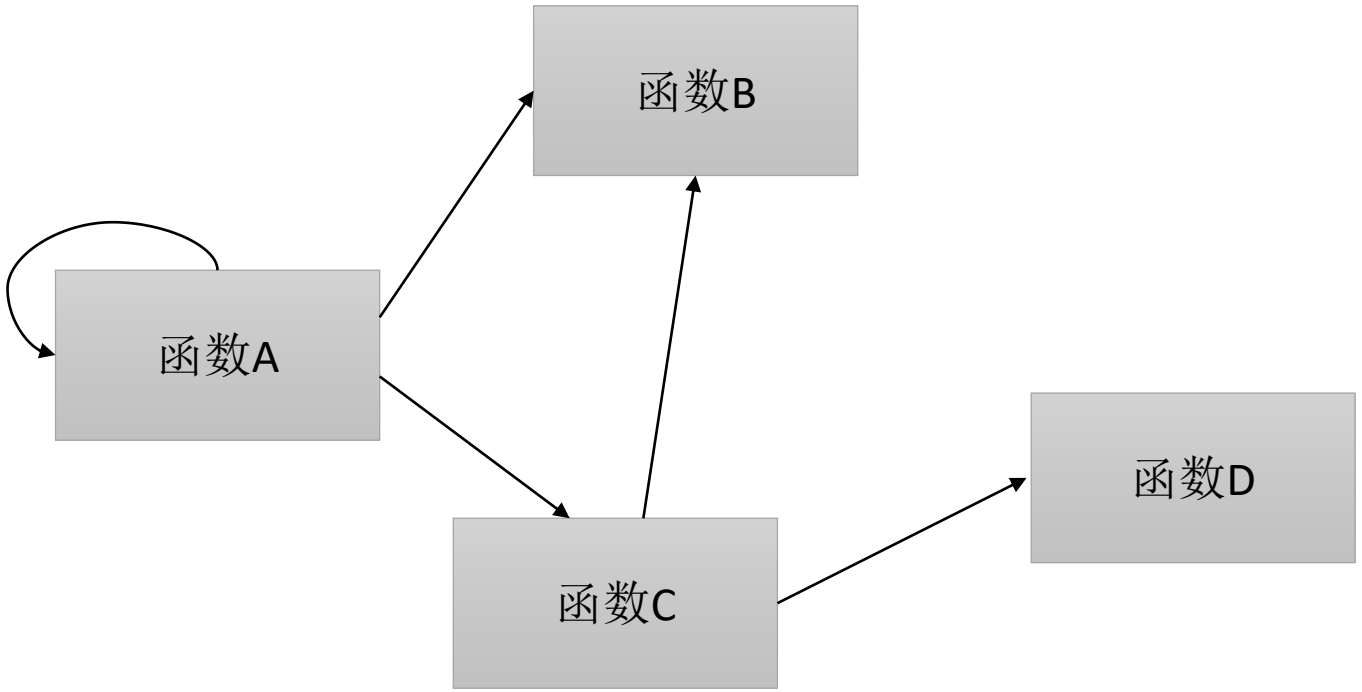

A
空表
B
6
2
线性表
C
L
'a'
5
3
'x'
纯表
D
B
C
A
L
6
2
'a'
5
3
'x'
纯表
E
B
D
C
A
6
2
L
'a'
5
3
'x'
可重入表
F
4
循环表

广义表与图

- 图⊇可重入表⊇纯表 (树) ⊇线性表
  - 广义表是树形结构、图结构的推广
- 递归表是有回路的再入表
- 线性表是树形高度为 2 的纯表

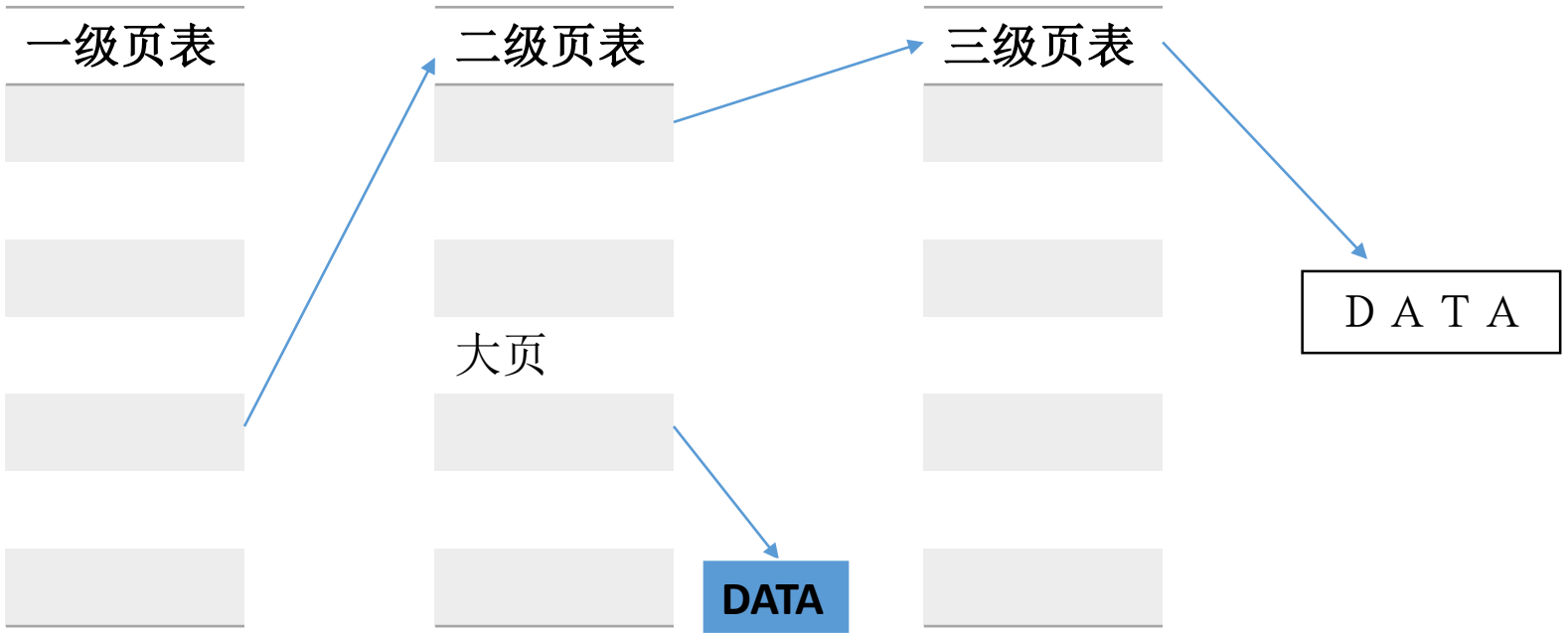
广义表应用

- 函数调用关系
- 功能块可以是原子



广义表应用

- 虚拟内存



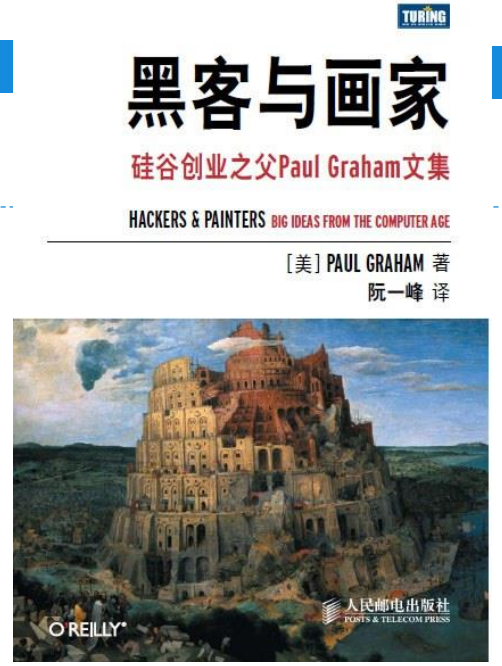
广义表应用：LISP 语言

- LISP 语言

```
(define fib (n)
  "Simple recursive Fibonacci number function"
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
```

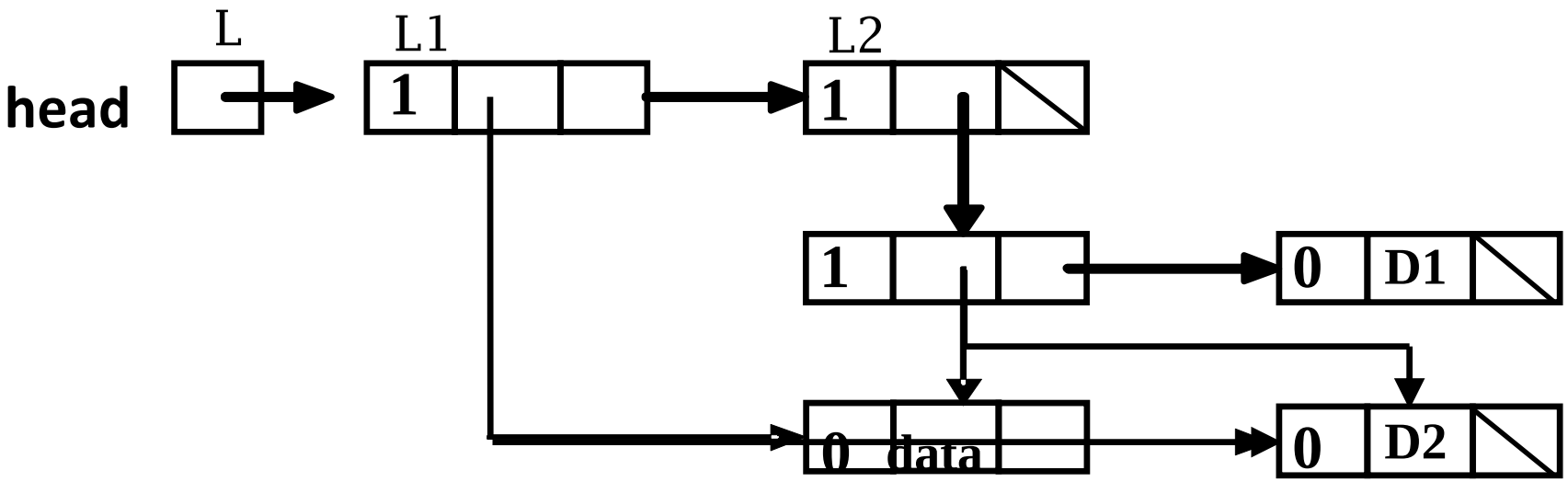
- 保罗·格雷厄姆，康奈尔大学本科，哈佛博士
  - 1995年创办 Viaweb，采用 Lisp 框架，帮助用户网上开店
  - 2005年，创建了风险投资公司 Y Combinator
- MIT SICP 采用LISP语言的变种Scheme 语言
  - UC Berkeley 的 CS61A 改用 Python

https://www2.eecs.berkeley.edu/Courses/CS61A/



例：广义表ADT

L: (L1: (data, D2) , L2: (L1, D1))



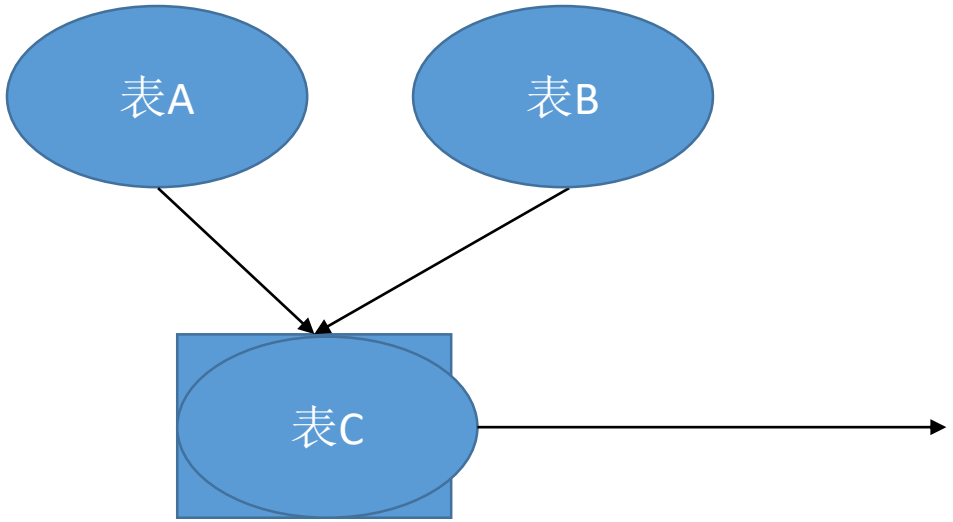
Head(L)：返回L的data或sublist字段
Tail(L)：返回L的next字段

广义表 ADT 结点定义

```
typedef  enum {ATOM=0, LIST=1} TAG;
typedef  struct GLNode {
            TAG  tag;
            union {
                        ElemType data;
                        GLNode *sublist;
            };
            GLNode *next;
};
```

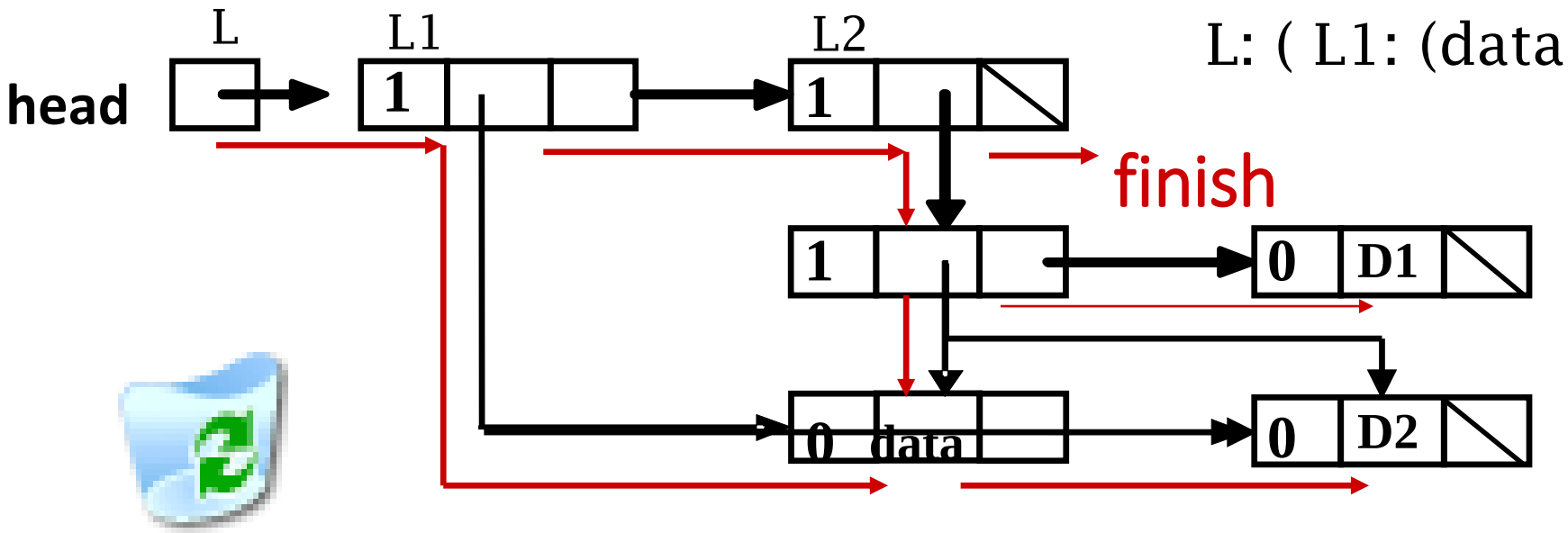
广义表存储的存储

- 如果有一个原子被多个广义表同时引用了怎么办



广义表的删除（不带表头）

- 不带头结点的广义表链
  - 在删除结点的时候会出现问题
  - 删除结点 data 就必须进行链调整



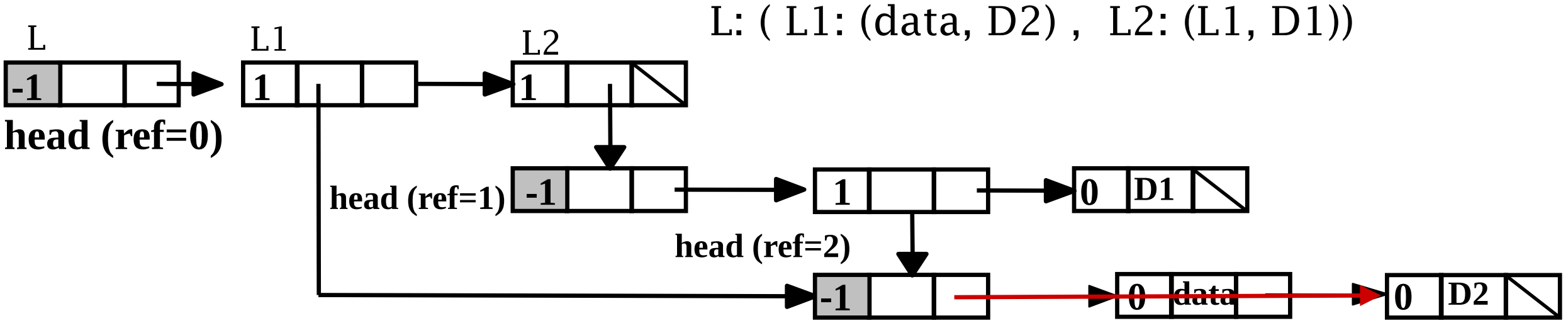
L: (L1: (data, D2) , L2: (L1, D1))

广义表的基本操作

- 基本操作：插入、删除、遍历

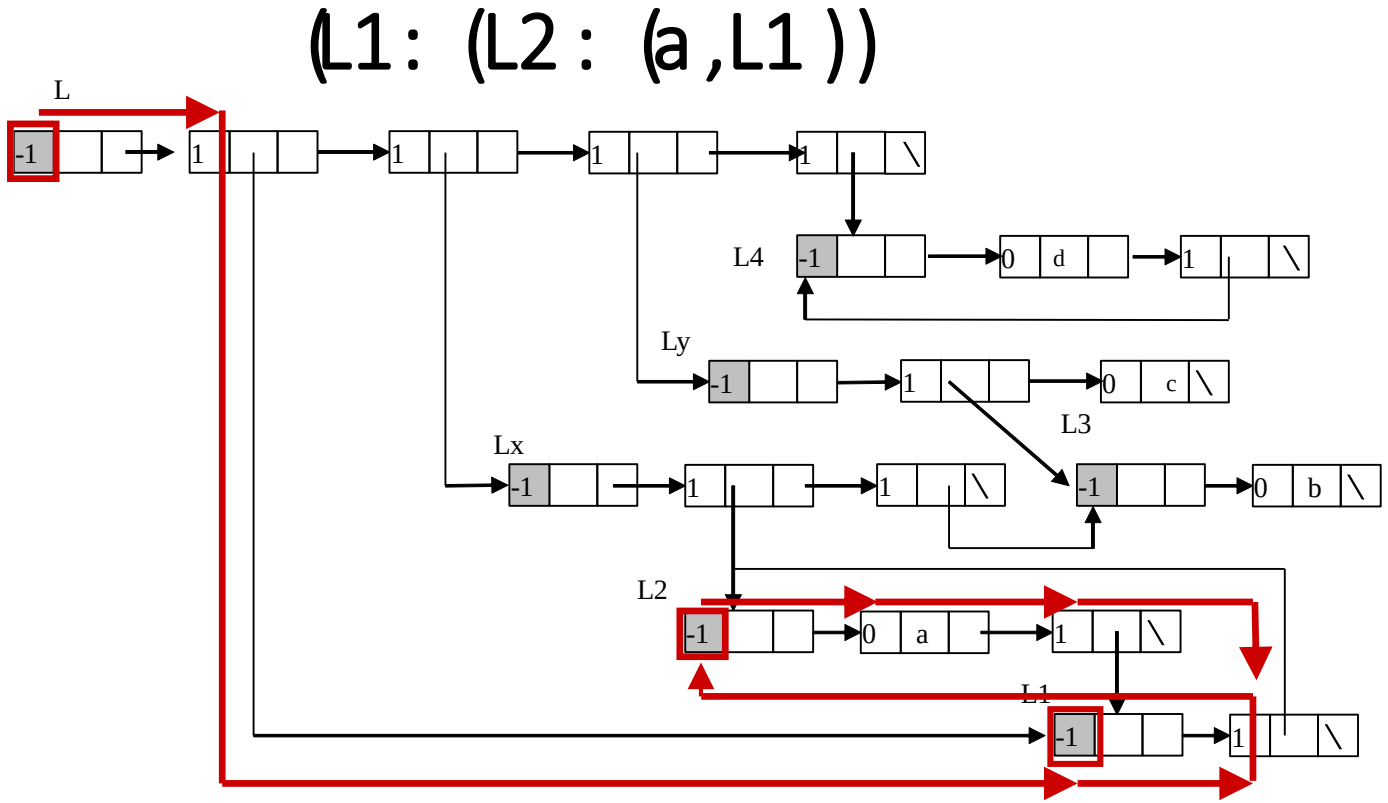
- 实现要点
  - 每个子表引入虚表头结点
  - 可极大的简化操作

广义表的存储（带虚表头结点）



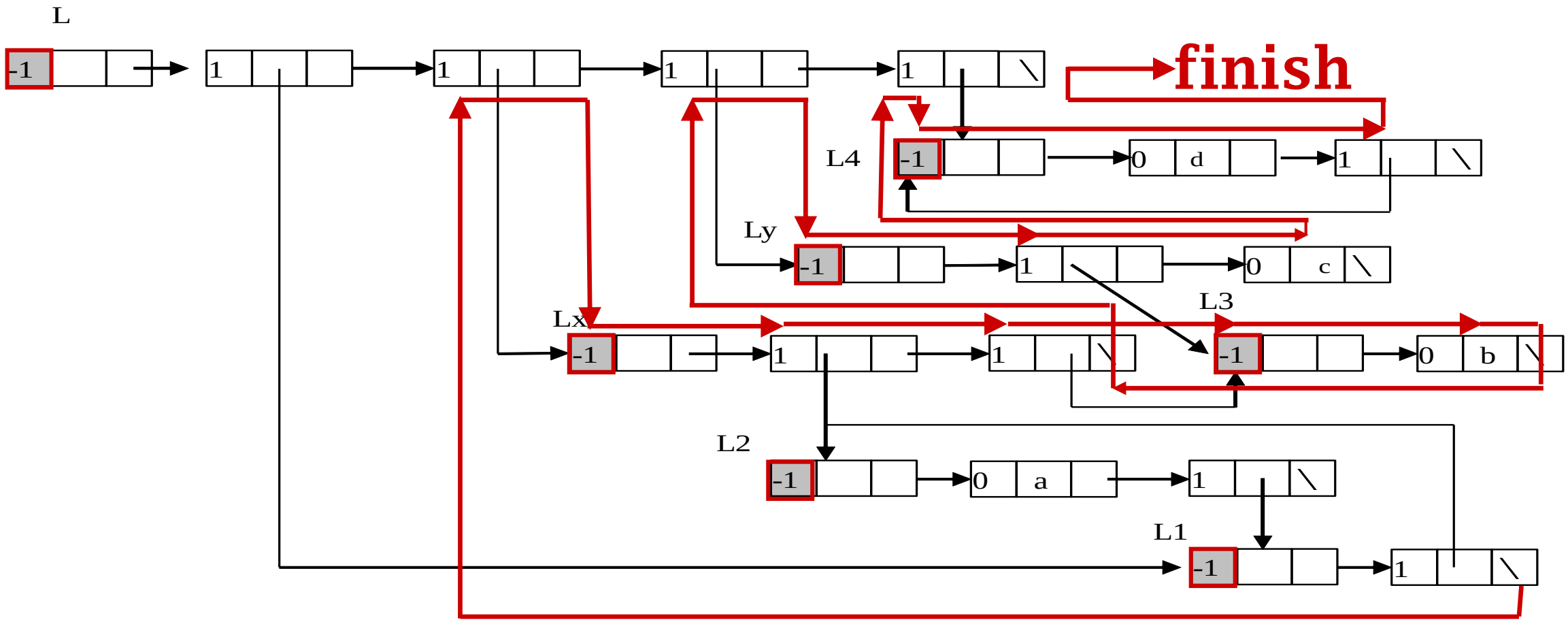
- 增加头指针，简化删除、插入操作
- 增加头指针，简化删除、插入操作
  - tag == -1表示虚表结点，标志位——图的因素
  - 重入表，尤其是循环表

对带表头结点的广义表遍历操作



12.2 广义表

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



思考

- 广义表与树、图各有什么区别与联系?
- 总结如何实现广义表遍历的算法?
  - 类似图遍历
  - 记录访问过的各个子表

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**
  - **可利用空间表与定长内存空间管理**
  - **变长空闲块分配与回收策略**
  - **失败处理机制**
- 12.4 Trie 树
- 12.5 AVL树的概念与插入操作
- 12.6 AVL树的删除操作与性能分析
- 12.7 伸展树

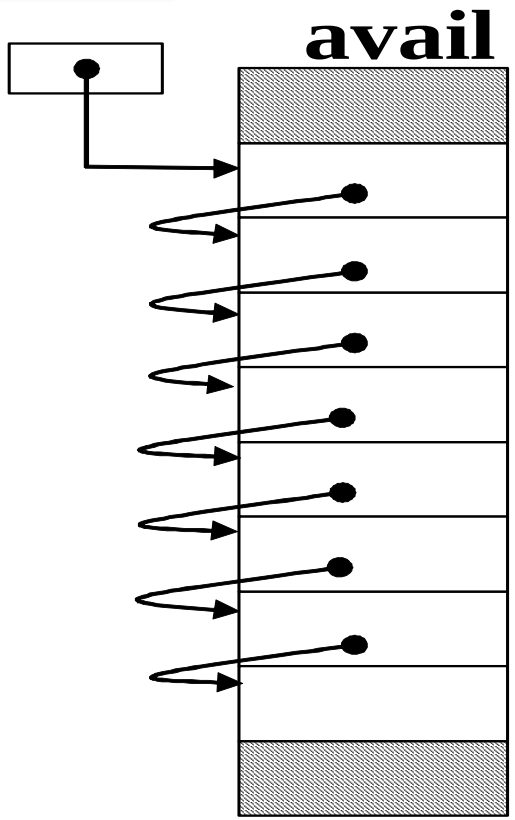
分配与回收

- 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
  - 减少可用内存空间
- 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

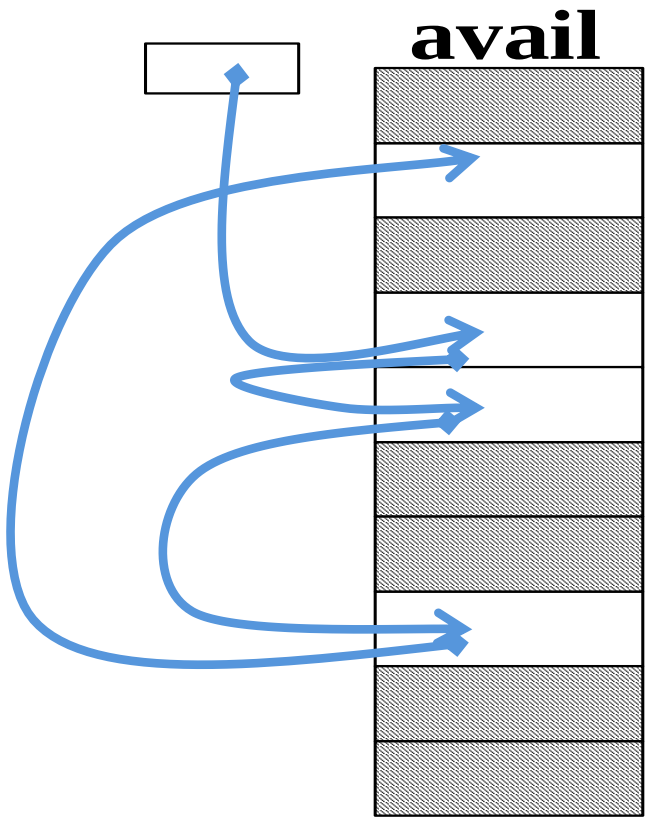
可利用空间表

- 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

12.3 存储管理



（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

程序自定义内存管理

- 许多场景下，应用程序倾向于独立进行内存管理
  - 而非向系统提出申请
- 效率优势：每次new/delete或者malloc/free都会导致系统调用
- 思想
  - 维护自己的可利用空间表(free list)，并进行管理
  - 只有在有必要的时候，才调用系统new/delete

程序自定义内存管理：数据+可利用空间表信息

```
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
};
```

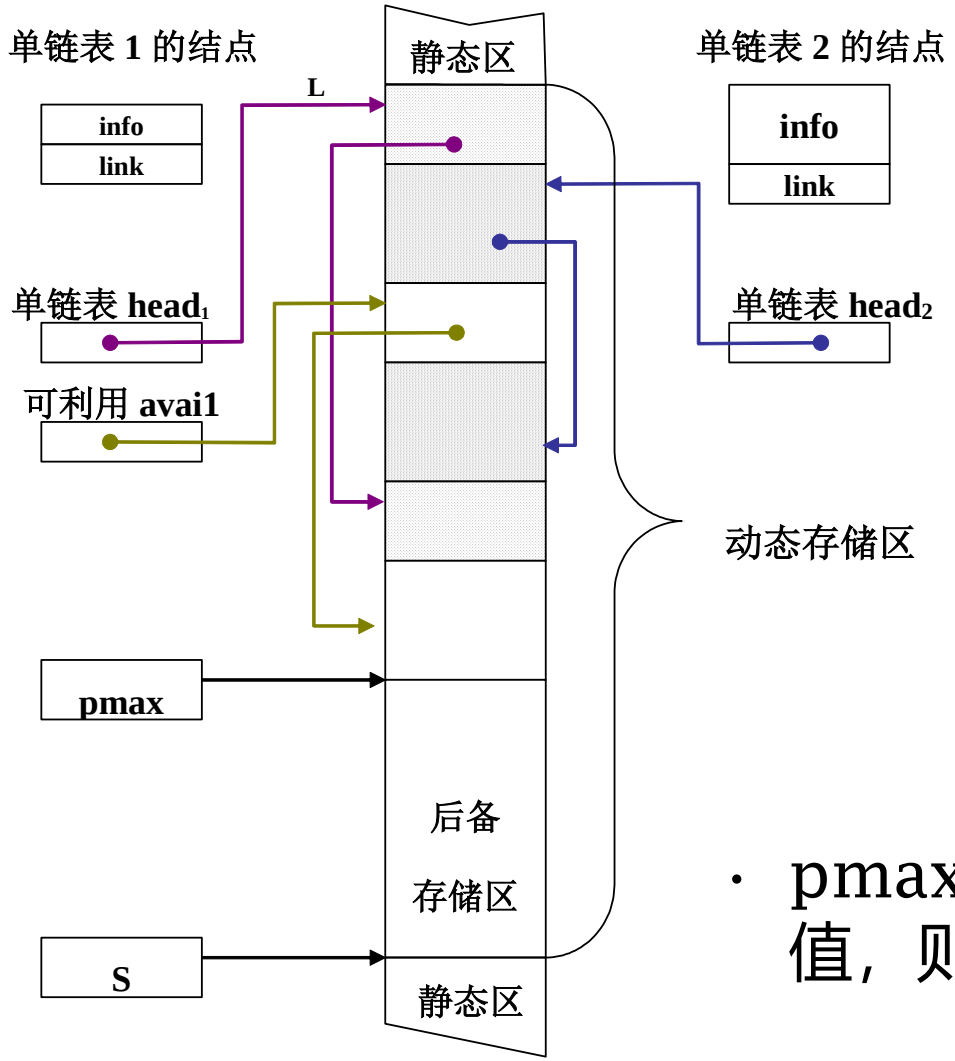
重载new运算符

```
// 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

```
// 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;
}
```

12.3 存储管理



- pmax 值已经达到或超过S值，则不能再分配空间

可利用空间表：单链表栈

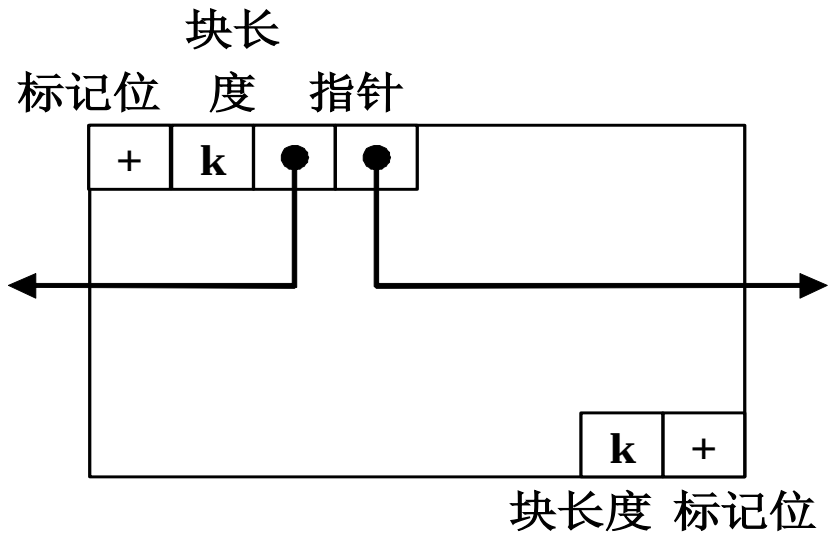
- new 即栈的删除操作
- delete 即栈的插入操作
- 直接引用系统的 new 和 delete 操作符，需要强制用 **“::new p”** 和 **“::delete p”**
  - 例如，程序运行完毕时，把 avail 所占用的空间都交还给系统 (真正释放空间)

变长内存区域分配

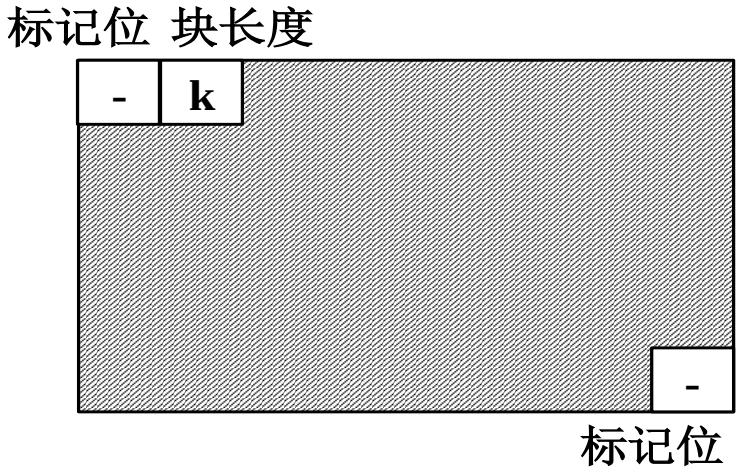
实际程序不仅仅需要申请定长数据结构

- 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

空闲块的数据结构

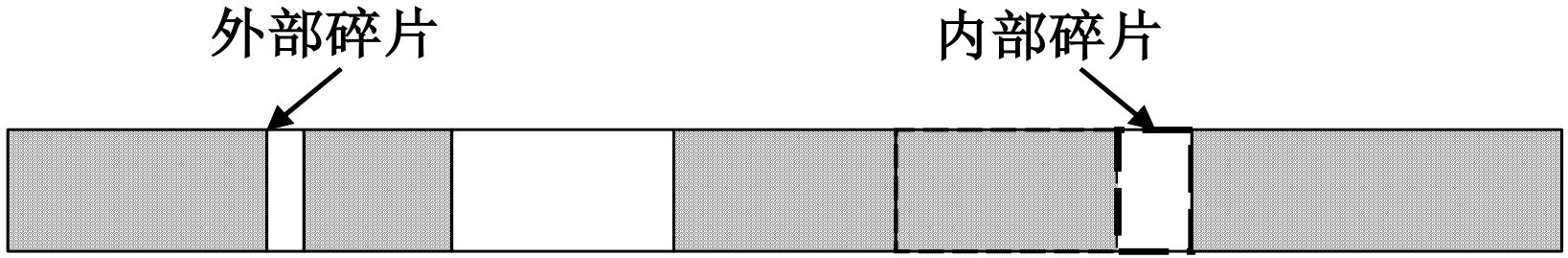


（a）空闲块的结构



（b）已分配块的结构

碎片问题



外部碎片和内部碎片

- 内部碎片：实际分配字节数多于请求字节数
  - 例：最小分配单位为1024字节，程序申请1000字节，导致24字节的内部碎片
- 外部碎片：小空闲块
  - 需要在分配与回收时进行管理，减少外部碎片

空闲块分配策略

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

空闲块分配策略

- 首先适配：返回free list里第一个满足申请字节数的块

- 问题: 三个块 1200，1000，3000
  请求序列：600，500，900，2200

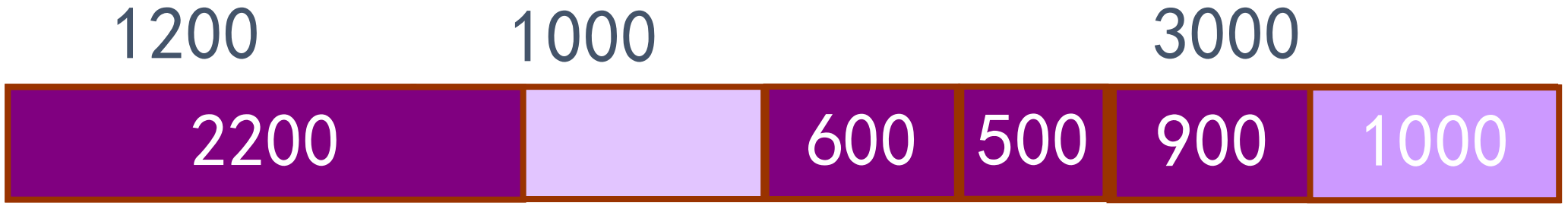
空闲块分配策略

- 最佳适配：返回free list里满足申请字节数的最小块

请求序列：600，500，900，2200

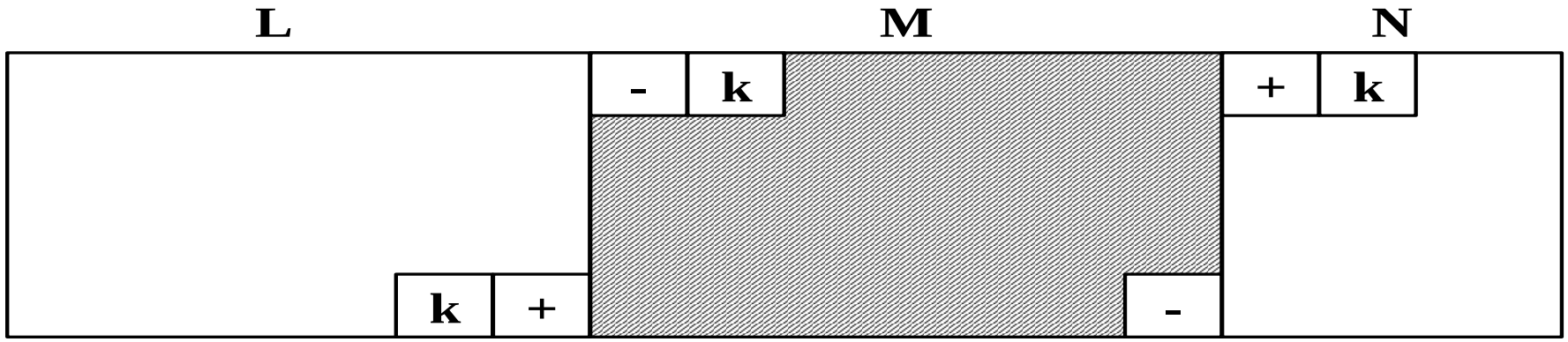
空闲块分配策略

- 最差适配：返回free list里满足申请字节数的最大块



请求序列：600，500，900，2200

回收：考虑合并相邻块



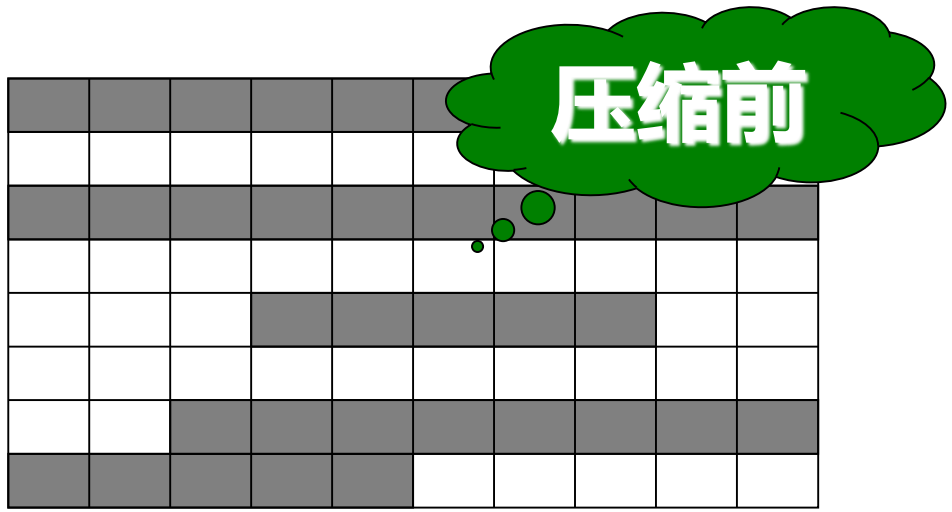
把块 M 释放回可利用空间表

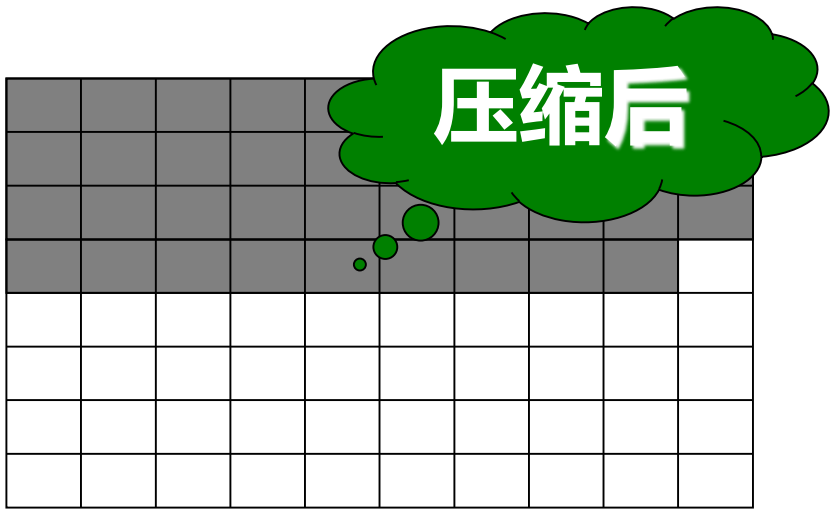
失败处理策略和无用单元回收

- 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
  - 1. 什么都不做，直接返回一个系统错误信息
  - 2. 使用失败处理策略（failure policy）来满足请求

失败处理策略：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

- 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

思考

- 比较首先适配、最佳适配和最差适配的特点
  - 哪种适配方案更容易产生外部碎片?
  - 哪种方案总体最优?

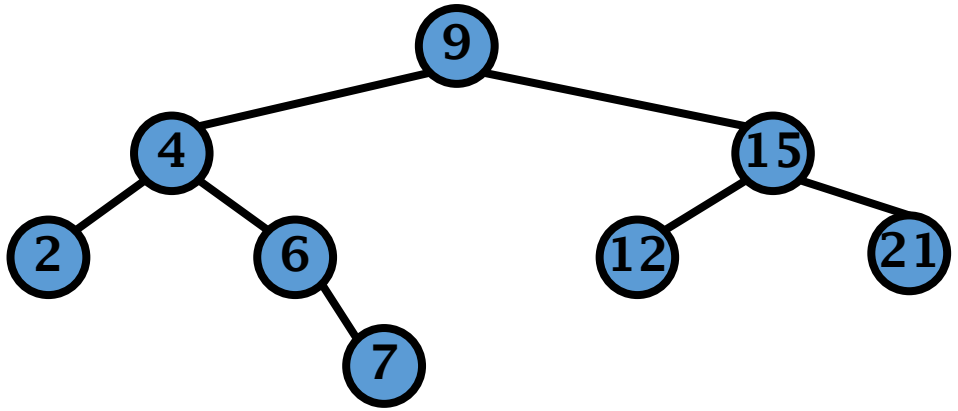
- 怎样有效地组织空闲块，使得分配时查找到合适块的效率更高?
  - 所有空闲块都组织在一个线性表中
  - 不同规模的空闲块组织在不同的链表中
  - 以空闲块的大小为 key 组织在平衡的 BST 中

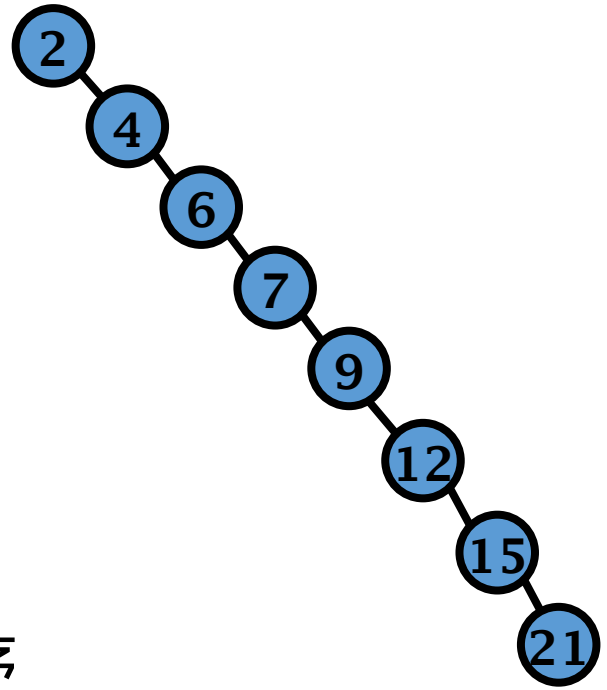
第十二章 高级数据结构

- 12.1 多维数组
- 12.2 广义表
- 12.3 存储管理
- **12.4 Trie 树**
  - **Trie树的概念与性质**
  - **后缀树与后缀数组**
- 12.5 AVL树的概念与插入操作
- 12.6 AVL树的删除操作与性能分析
- 12.7 伸展树

12.4 二叉搜索树局限性

- 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





- 二叉搜索树结构取决于插入删除顺序

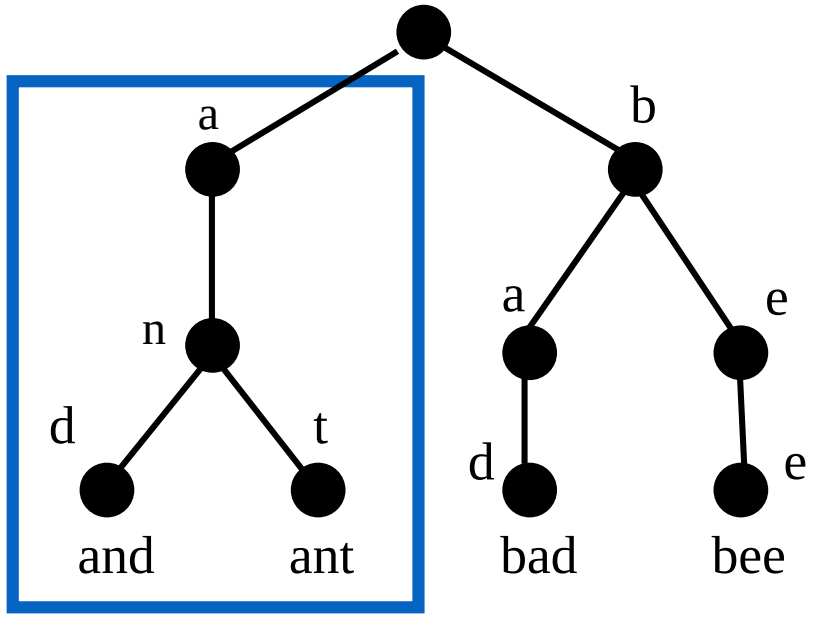
Trie 结构

- 目标：与输入顺序无关的搜索树
- 思想：关键码对象空间分解
- 主要应用
  - 信息检索 (information retrieval)，trie”这个词来源于“retrieval”
  - 自然语言大规模的英文词典
- 常见类型
  - 字符树——26叉Trie
  - 二叉Trie树
    - 编码只有0和1
    - 用每个字母（或数值）的二进制编码来代表

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}



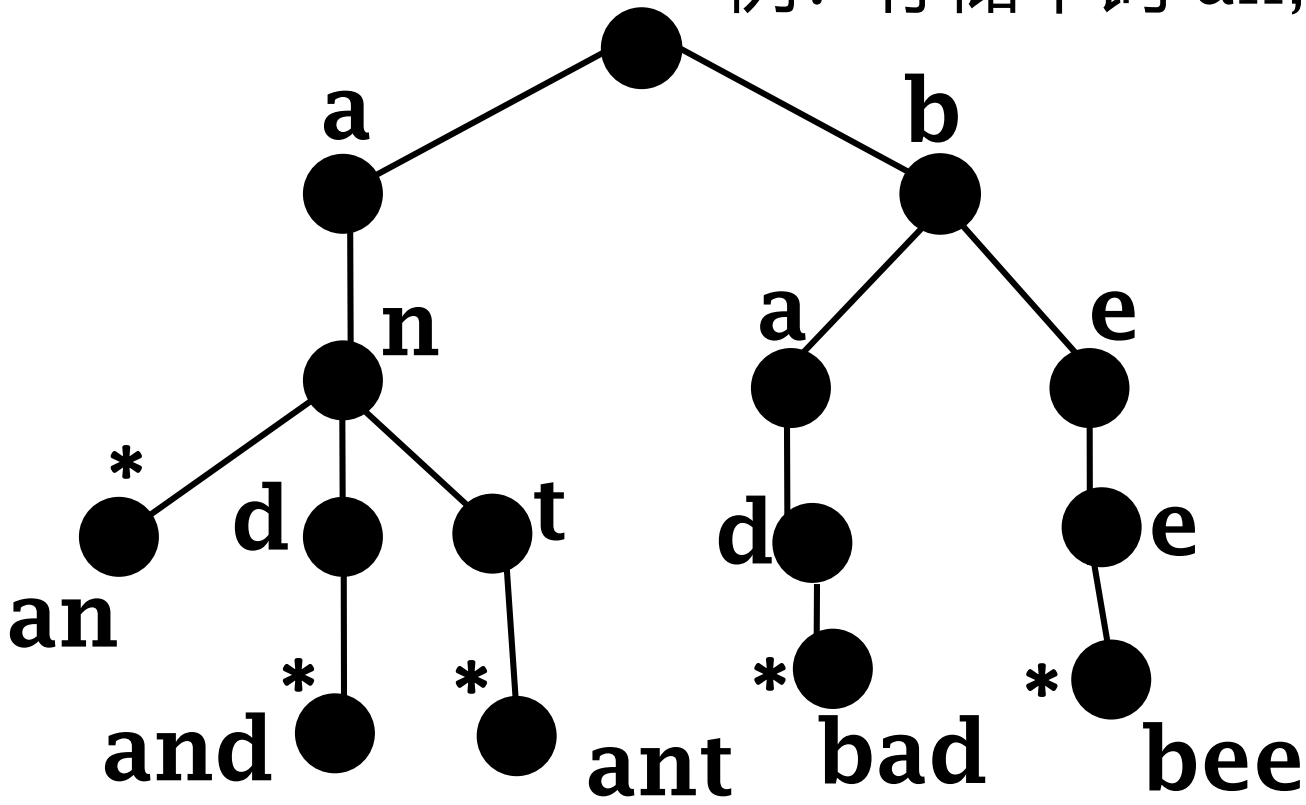
- 除根以外，每个结点表示一个字符
- 每个“根-叶”路径表示一个集合中的字符串
- 一棵子树代表具有相同前缀的关键码的集合

12.4 Trie 树

不等长的字符树，加“*”标记

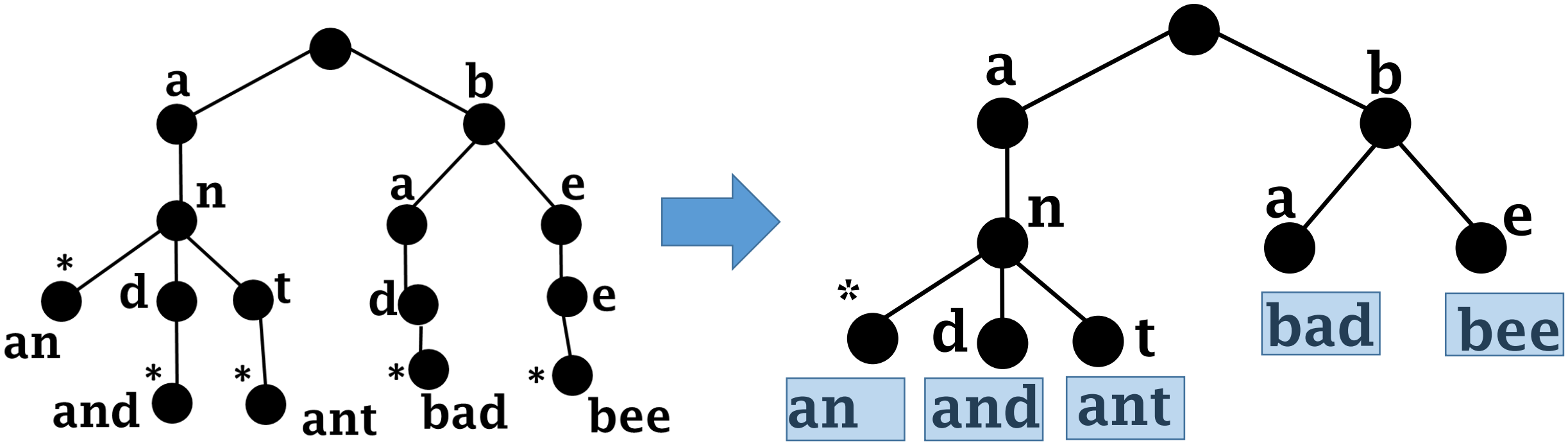
保证前缀也能被正确存储

例：存储单词 an, and, ant, bad, bee



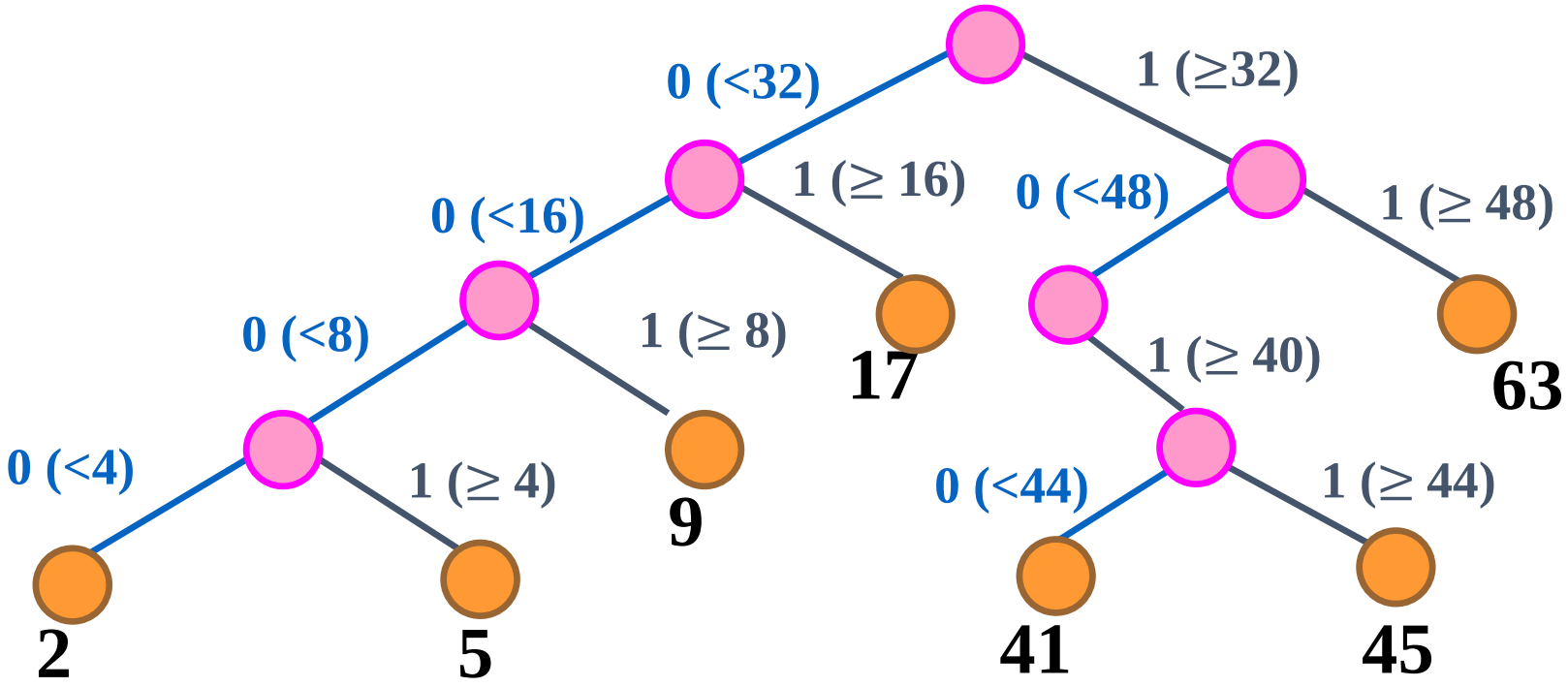
压缩靠近叶结点的单路径

存储单词 an, and, ant, bad, bee



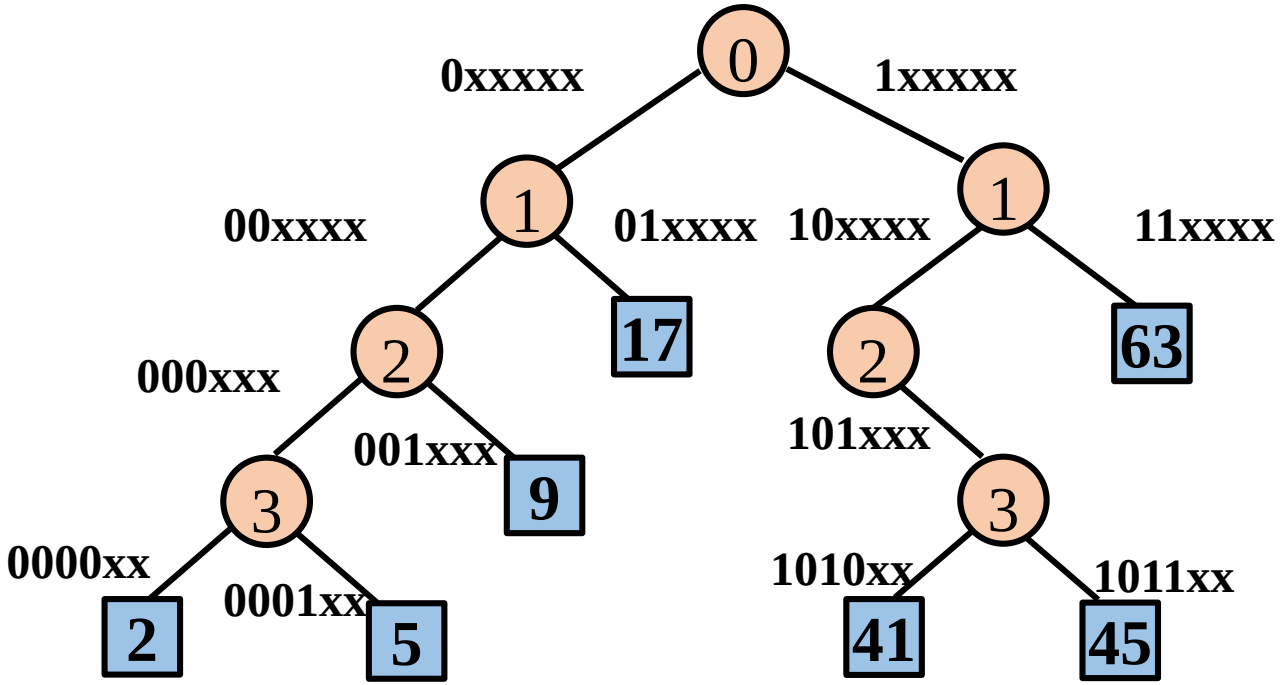
二叉 Trie 结构

元素为 2、5、9、17、41、45、63



12.4 Trie 树

二叉 Trie 结构



压缩

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

12.4 Trie 树

Patricia结构

- 该压缩技术又被称作Patricia Trie
  - Trie的变种

- 论文：“Practical Algorithm To Retrieve Information Coded In Alphanumeric”

- Patricia Trie：
  - 若一个结点是其父结点唯一儿子，则与父结点合并

Trie的性质

- 一组key形成唯一的Trie结构
- 该结构与key的插入/删除序列无关

查找命中时的时间复杂度

- 插入与查找操作的复杂度
  - 取决于插入/查找的 key 的长度+1
  - 路径压缩使得实际长度更小
  - 复杂度与 Trie 中 key的个数无关

- Trie在查找命中时有极高的效率
  - 从根结点开始较短路径

查找失败时的时间复杂度

- 查找失败时的平均复杂度
  - 假设 n 个随机 key 组成 trie
  - Key 的字符表包含 r 个字符
- 查找失败时的平均访问结点数是 $\log_r n$
  - 1 百万个 key，平均只需要访问 3-4 个结点
- 英文字符串长度5，$26^5 = 11881376$个可能 key
  - BST 至少要进行 $\log_2 26^5 = 23.5$ 次比较

空间复杂度

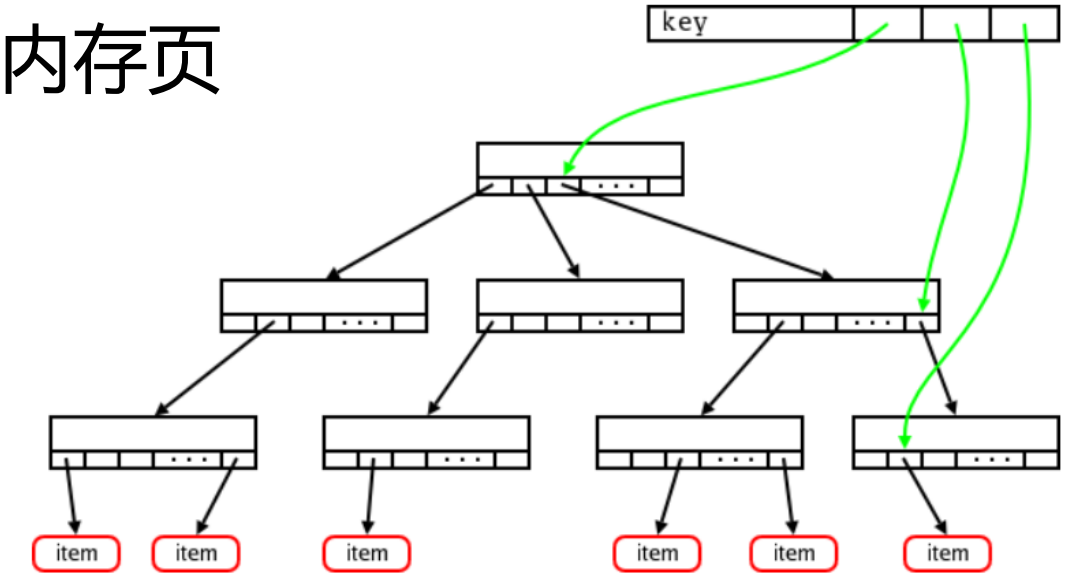
- Trie中结点数至多nw
  - n: key的总个数
  - w: 平均key长度
- 每个结点需要维护一个O(r)的表指向所有子结点，r为字符基数
- 空间开销相比于其他数据结构较大
  - 当 key 的数量较多、key较长、r 基数较大时尤为明显
- 但在承受的起空间开销时，Trie的查询速度高于其他搜索树

12.4 Trie 树

Trie树的应用

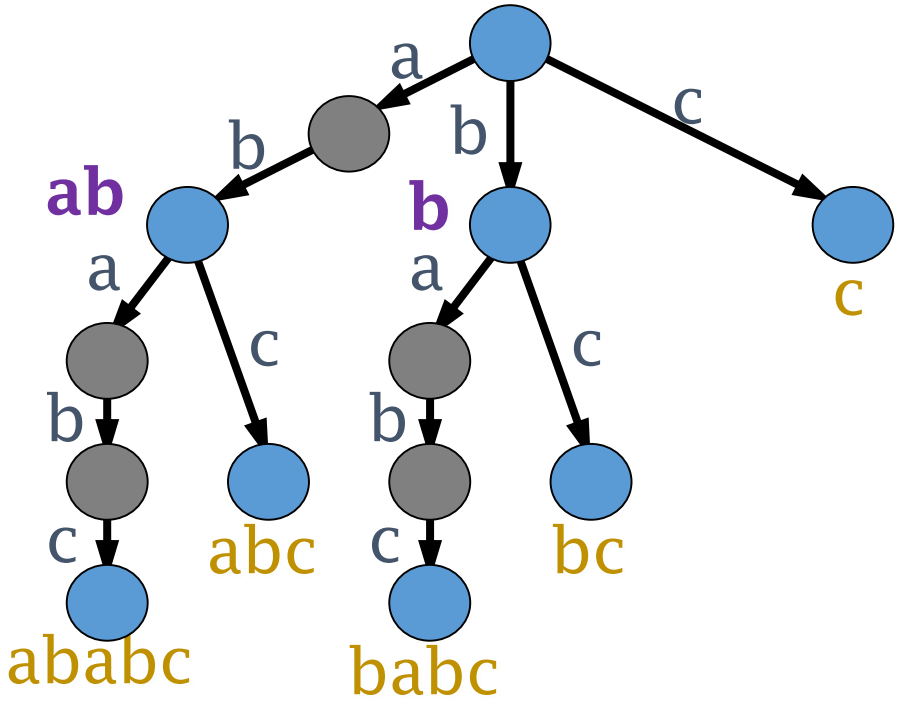
- Linux中的Trie：又称Radix树
- 重要用途：基于内存地址查找对应的内存页

```
#include <linux/radix-tree.h>

RADIX_TREE(name, gfp_mask);  /* Declare and initialize */

struct radix_tree_root my_tree;
INIT_RADIX_TREE(my_tree, gfp_mask);


int radix_tree_insert(struct radix_tree_root *tree, unsigned long key,
                      void *item);
void *radix_tree_delete(struct radix_tree_root *tree, unsigned long key);
```

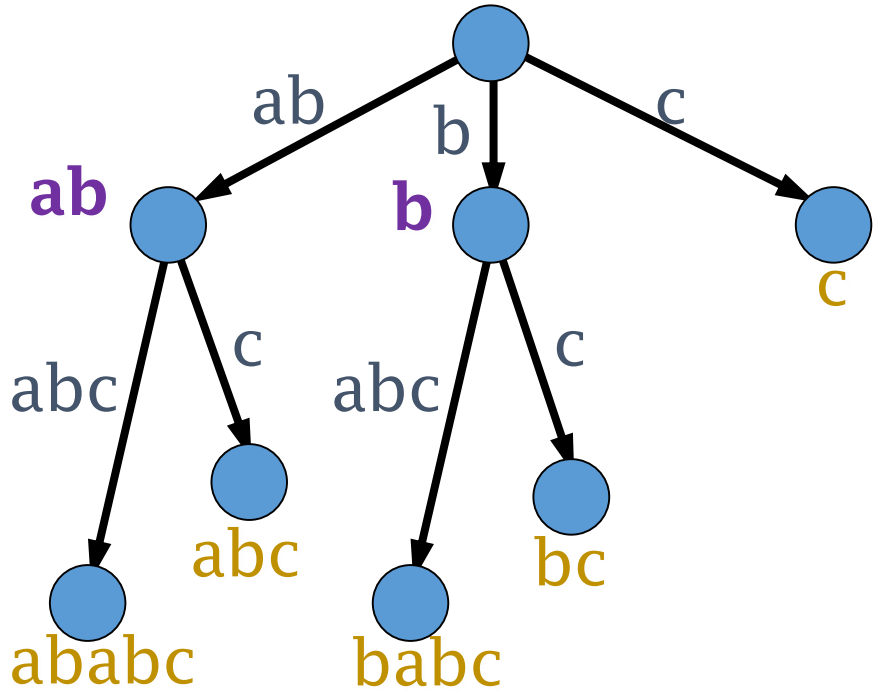


后缀树 (Suffix Trees)

ababc 后缀子串: ababc, babc, abc, bc, c
后缀树保存了一个字符串所有后缀



后缀构成的字典树（未压缩）

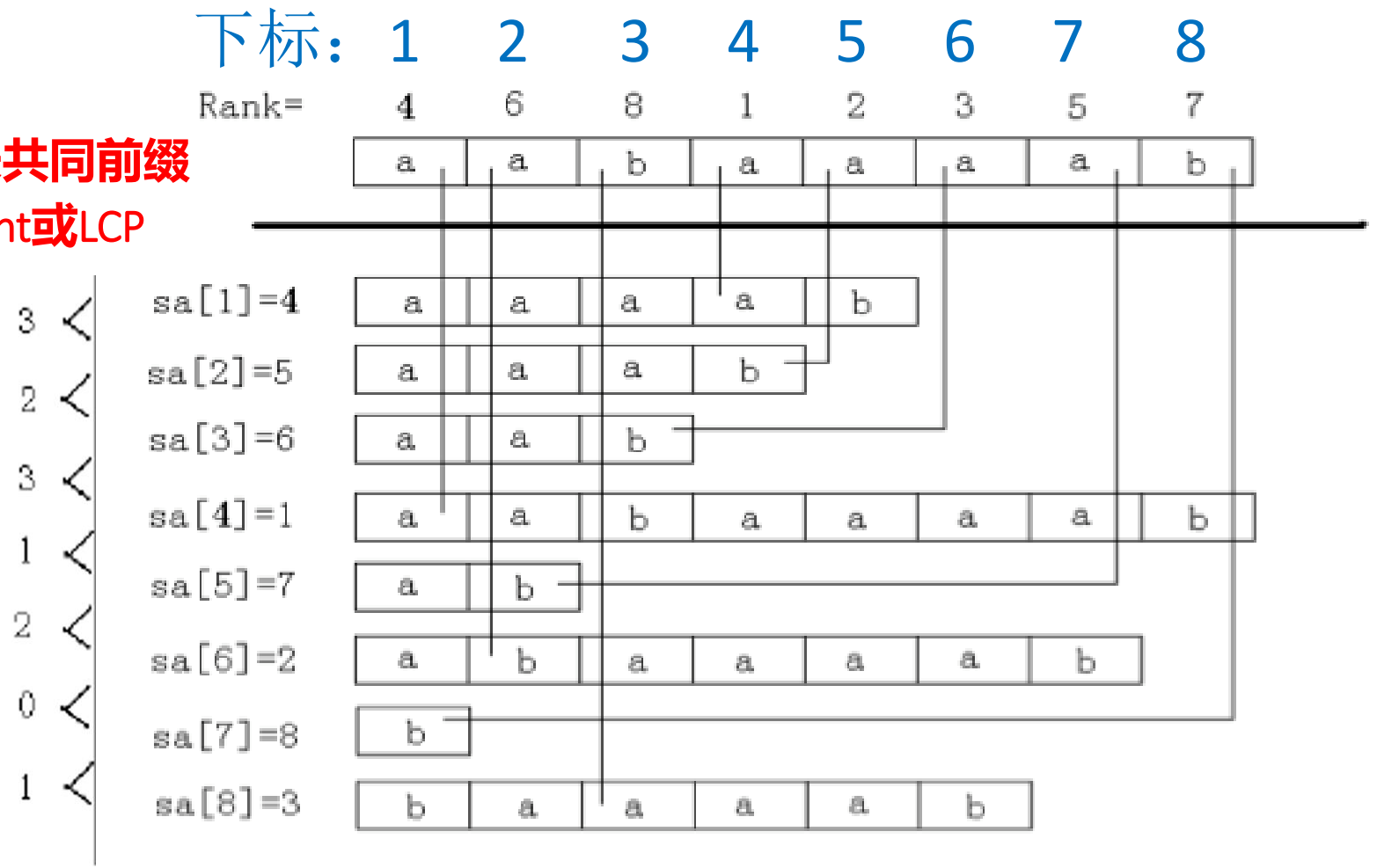


后缀树（压缩后）

后缀数组

构造：倍增算法O(nlogn)，DC3 O(n)

- Rank **你排第几?**
- **最长共同前缀**
  Height**或**LCP
- sa **第几是谁?**

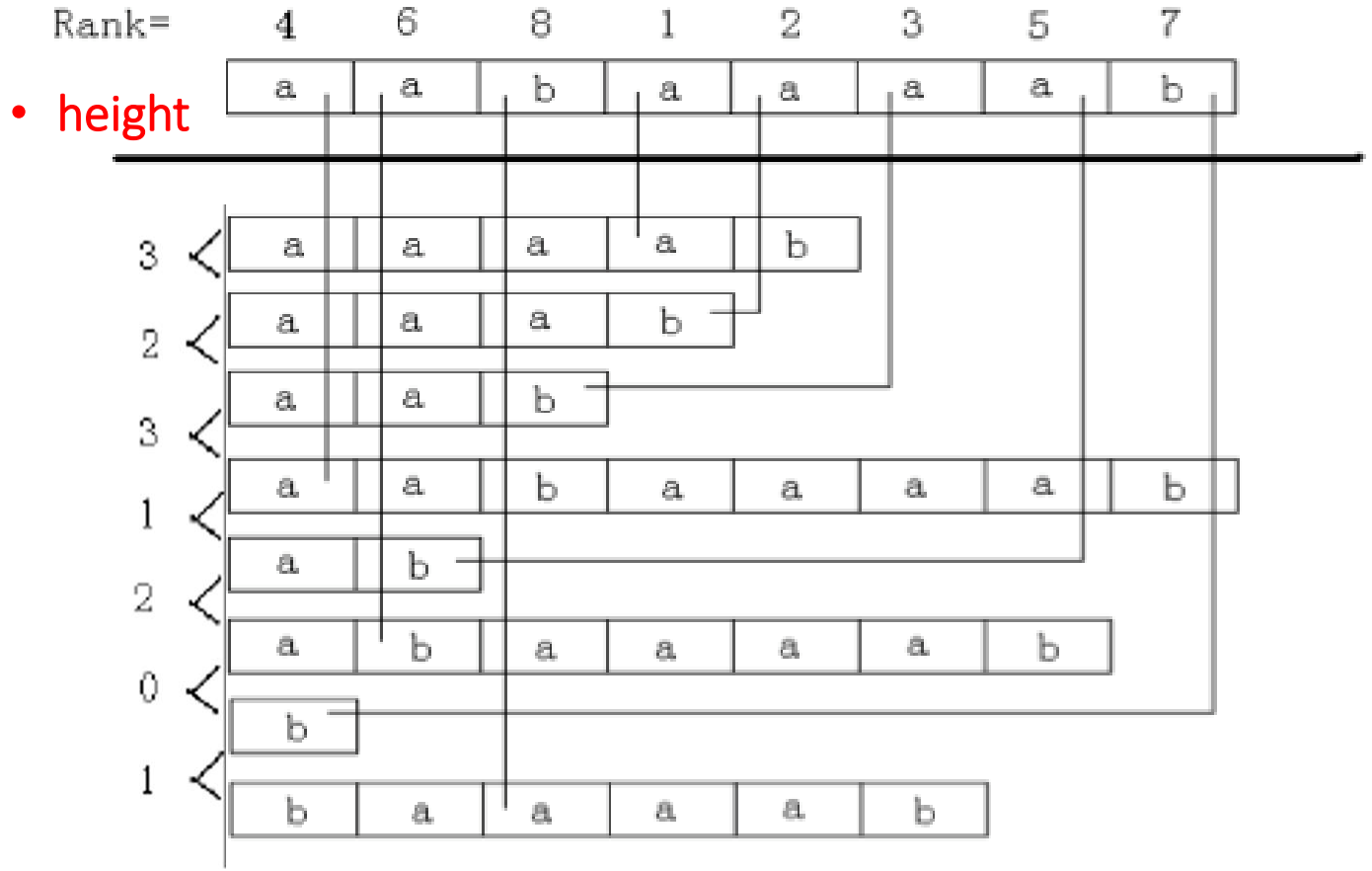


可重叠最长重复子串

- 给定一个字符串，求最长重复子串的长度，这两个子串可以重叠
- 答案：height 最大值

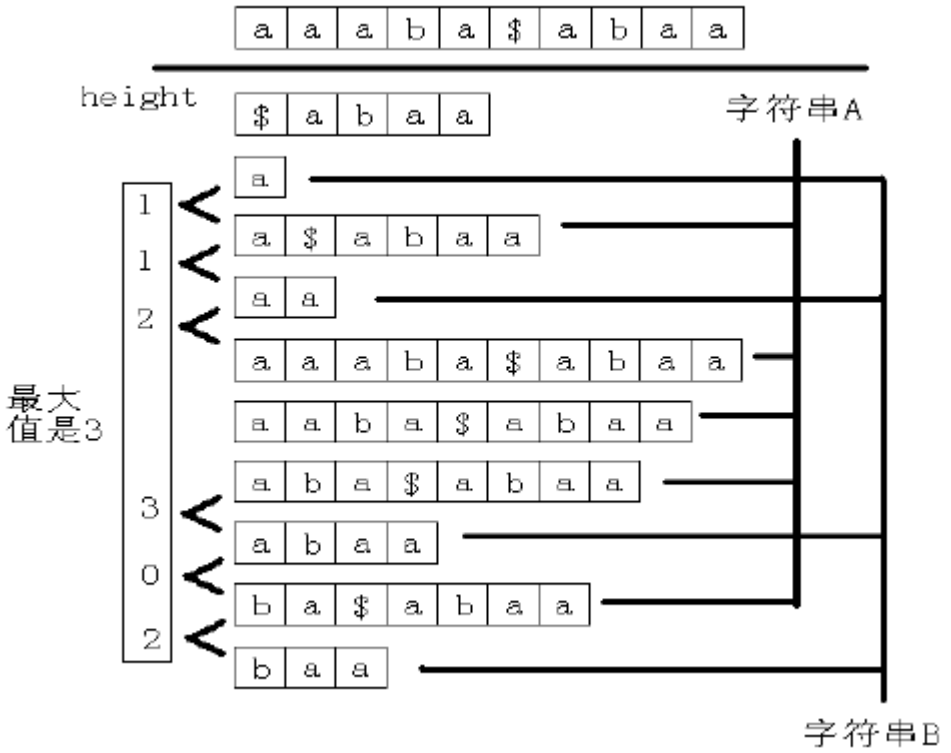
任意两个后缀的最长公共前缀都是height 数组里某一段的最小值，所以答案一定不大于height的最大值。

height数组的最大值本身就代表了一个重复子串的方案，所以答案一定不小于height的最大值。



两个字符串的最长公共子串

- 给定两个字符串A 和B，求最长公共子串。
- 将A和B连接起来，中间添上分隔符$，并求sa及height



Height的最大值？

最长公共前缀数组

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（lcp, height）

后缀5和1共享 “ALA”

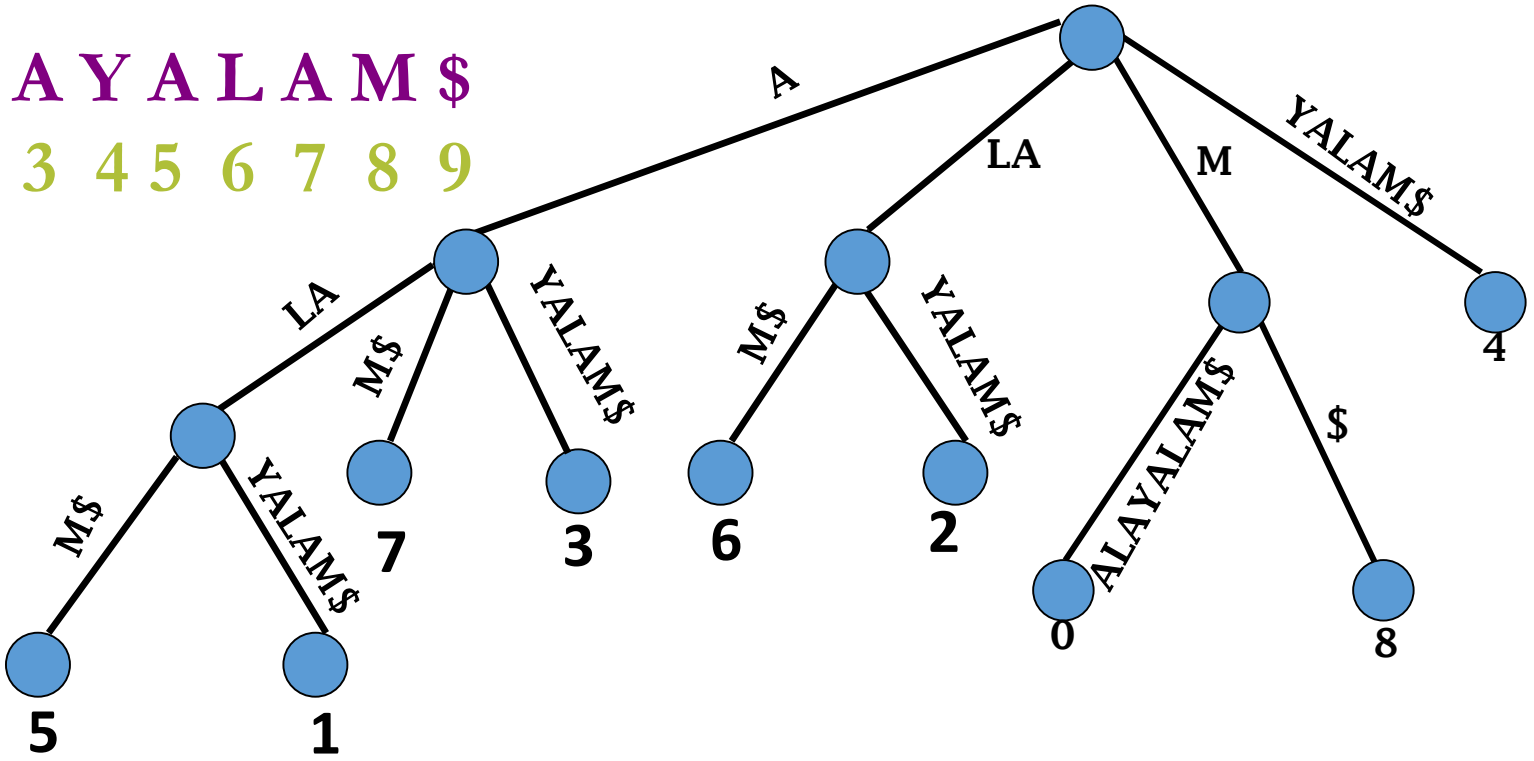
后缀1和7共享 “A”

LCP总是相邻的

从后缀数组构造后缀树

- 给定字符串A，将A的所有后缀进行排序
- 后缀数组 sa [i]：第i大的后缀在A中的起始位置

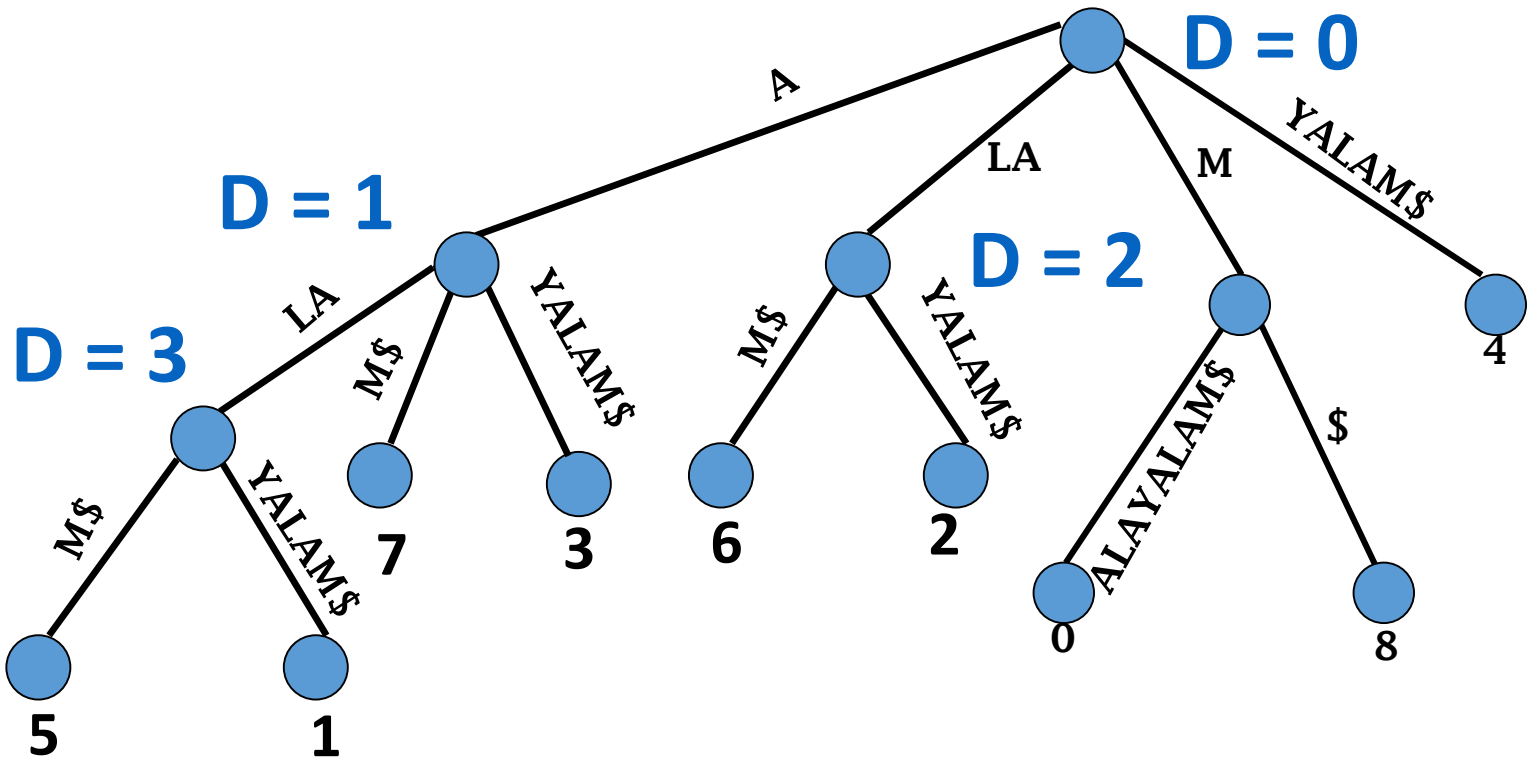
M A L A Y A L A M $
0 1 2 3 4 5 6 7 8 9



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

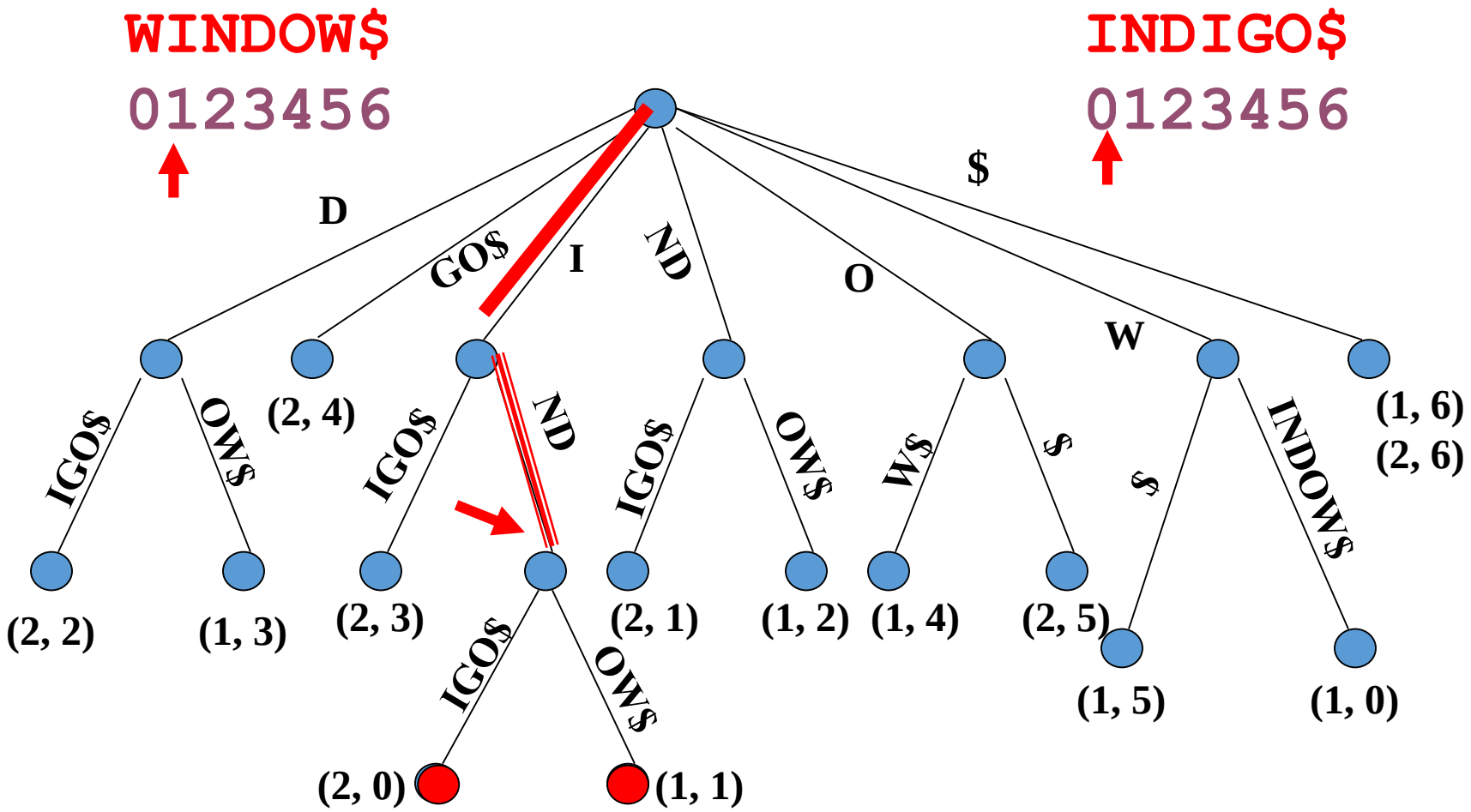
12.4 Trie 树

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



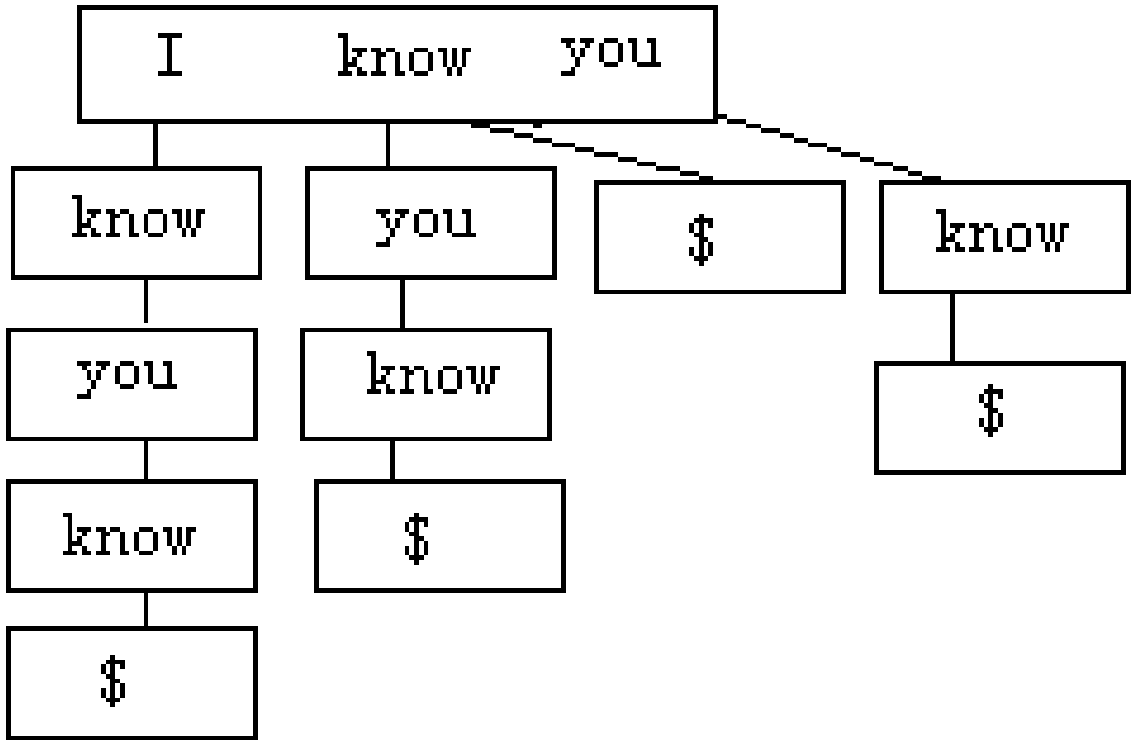
后缀数组 sa	5	1	7	3	6	2	0	8	4	9
最长公共前缀 lcp	**3**	**1**	**1**	**0**	**2**	**0**	**1**	**0**	**0**	-

GST共同前缀


WINDOW$
0123456
INDIGO$
0123456
D
GO$
I
ND
O
W
$
IGO$
OW$
(2, 4)
IGO$
ND
IGO$
OW$
W$
$
$
INDOW$
(1, 6)
(2, 6)
(2, 2)
(1, 3)
(2, 3)
(2, 1)
(1, 2)
(1, 4)
(2, 5)
(1, 5)
(1, 0)
IGO$
OW$
(2, 0)
(1, 1)

单词粒度的后缀树

- “I know you know $ ”



后缀数组的应用

- 许多字符串问题通过后缀数组或最长公共前缀数组快速计算
  - 最长重复子串（允许重叠）：找 lcp 数组中的最大值

- 海量字符串查找
  - 垃圾邮件检查：n 个关键字，m 个文本，求每一个关键字是否在某个文本中出现（文本的字串）
  - 解决方案：对文本建立后缀数组，然后对每个关键字进行二分查找

12.4 Trie 树

思考

- 中文是否适合组织字符树？是否适合 二叉 Trie 结构？

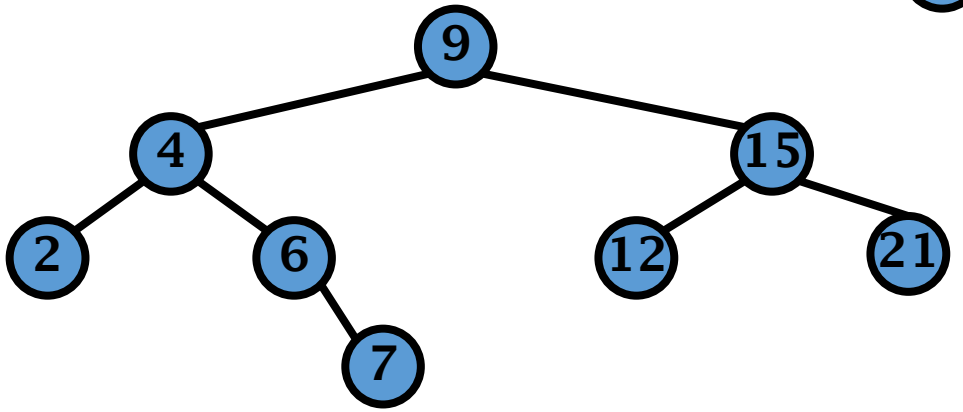
- 查阅后缀树、后缀数组的文献，思考其应用场景

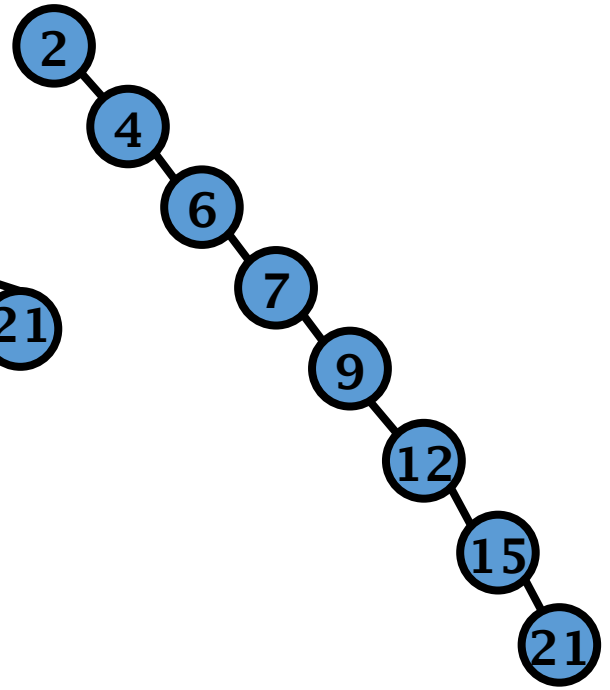
第十二章 高级数据结构

- 12.1 多维数组
- 12.2 广义表
- 12.3 存储管理
- 12.4 Trie 树
- **12.5 AVL树的概念与插入操作**
  - **AVL树的概念**
  - **AVL树的旋转操作**
  - **AVL的插入操作**
- 12.6 AVL树的删除操作与性能分析
- 12.7 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)





- 发明人：Adelson-Velskii 和 Landis
  - AVL 树，平衡的二叉搜索树
  - 始终保持O (log n) 量级
  - AVL树是最常见的一种平衡二叉树，还有其他类型的平衡二叉树变种（如：重量平衡的二叉树）

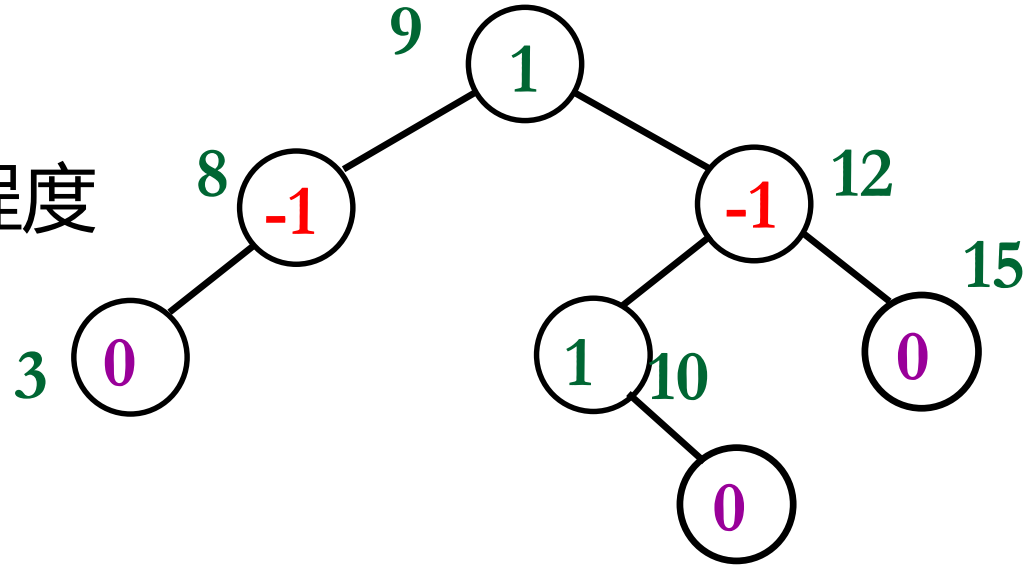
AVL树：定义

- 在普通二叉搜索树基础上，加入平衡规则
  - AVL树中的每个结点，其左子树与右子树 **高度差至多为1**
- 保证了N个结点的AVL树，高度为O(logN)

AVL 树的性质

- 可以为空
- 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

平衡因子



- 刻画二叉树中每个结点左右子树的平衡程度
- 对于结点x，其平衡因子定义为

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

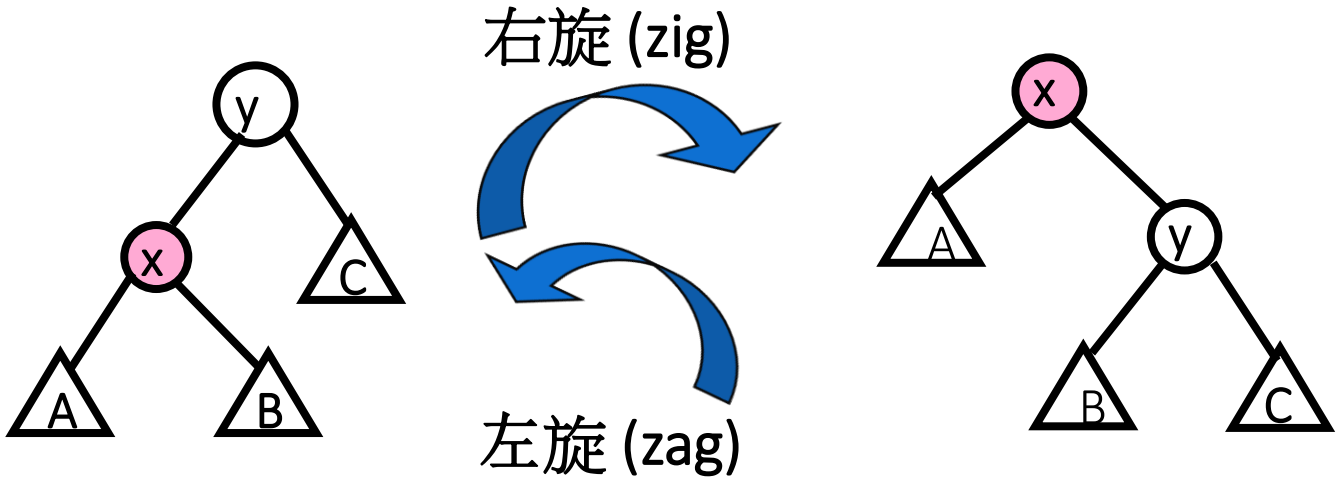
- 在一棵AVL树中，结点平衡因子取值为 0, 1, -1

AVL树的基本操作

- 查询操作：与一般二叉搜索树的查询完全一致
- 插入 & 删除：
  - 首先调用二叉搜索树的插入与删除操作
  - 然而…
    - 普通的二叉搜索树插入&删除可能破坏AVL树的平衡性质
    - 需要额外的操作保持AVL树的平衡

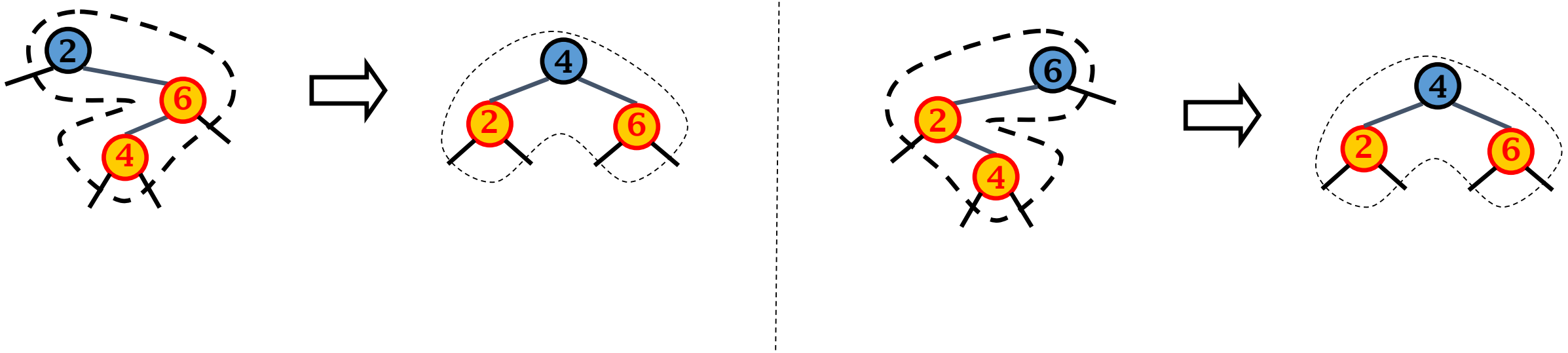
保持平衡的操作之一：单旋转

- 单旋转
  - 结点与它的父结点交换位置，保持 BST 特性



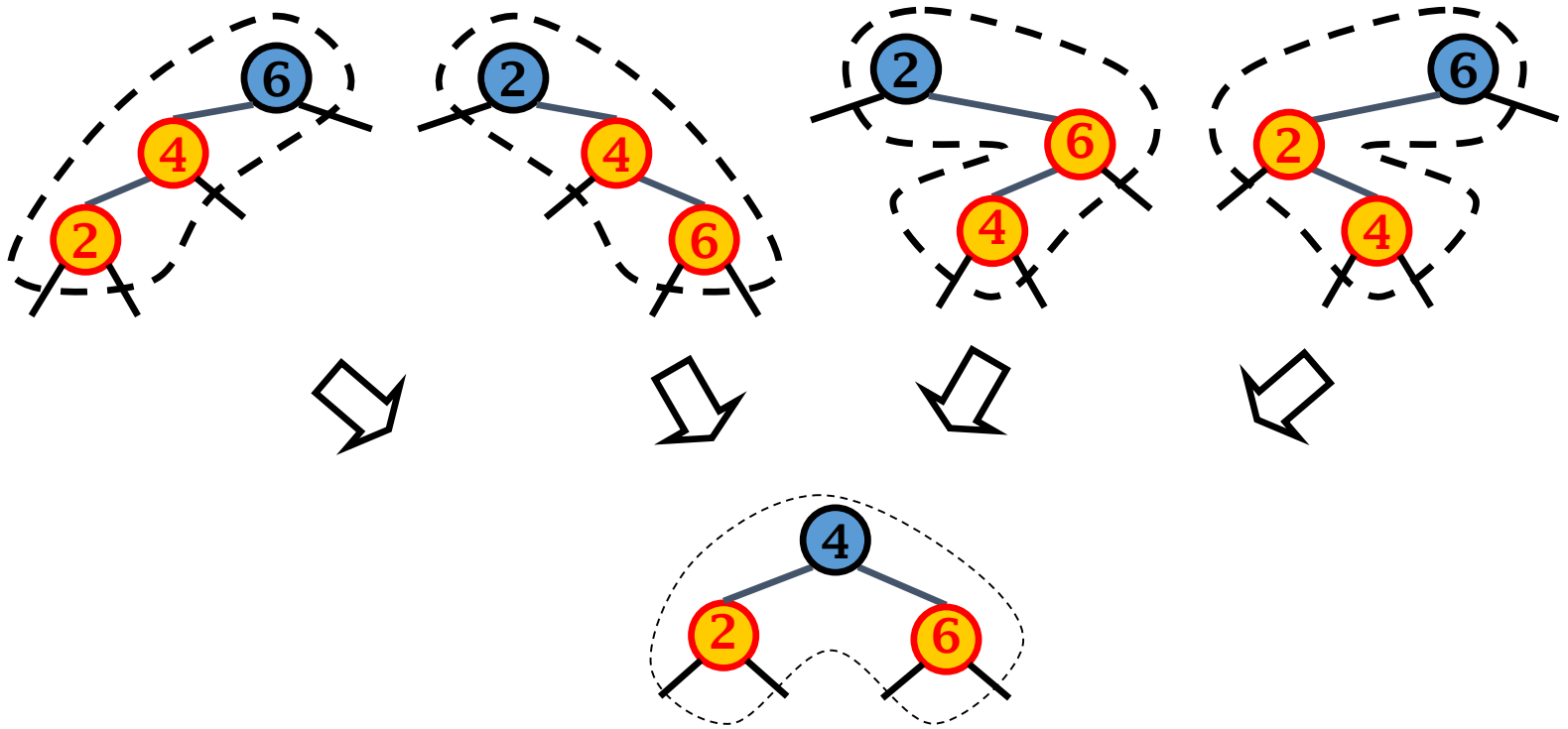
保持平衡的操作之二：双旋转

- 双旋转：
  - 结点与它的祖父结点交换位置，原祖父结点与原父亲结点分别为该结点的孩子（需保持BST特性）



保持平衡的操作小结

- 单旋和双旋：保持 BST 的中序性质

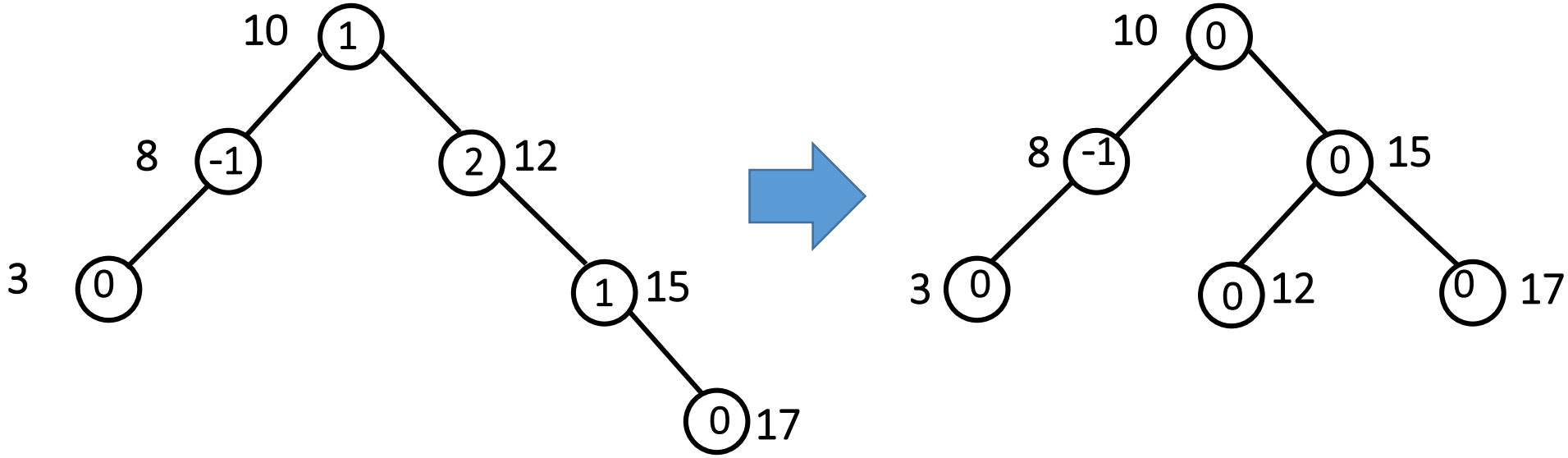


AVL 树结点的插入

- 步骤1：插入与 BST 一样
  - 新结点作叶结点
  - 只有该叶结点到根结点的路径上的结点，平衡性可能被破坏（树的高度增加了至多1）

- 步骤2：对该路径上的每一个结点x
  - 检查x两个子树高度是否增加，重新计算平衡因子
  - 当AVL性质被破坏，进行旋转操作，保持x的左右子树平衡
  - 不断遍历每个结点，直到
    - 到达根结点，或者
    - 路径上的某个结点y高度不变

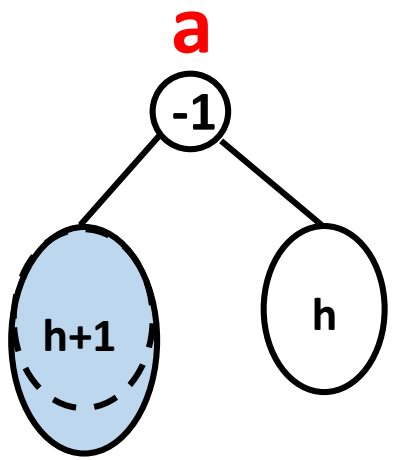
恢复平衡的例子



插入17后导致不平衡　　重新调整为平衡结构

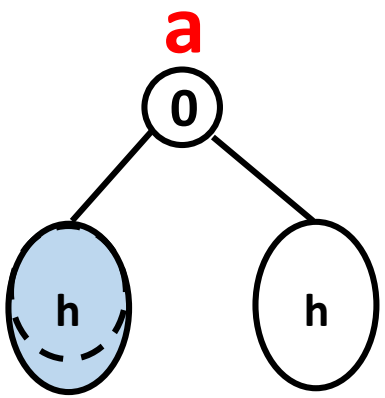
AVL 树结点的插入

- 插入新结点后，新结点的某个祖先a：平衡因子可能改变，也可能不变
  - 若a平衡因子改变，则有3种可能性



情况1：结点原先是平衡的，即bf(a)=0，插入后变成左重或者右重

解决方法：修改结点平衡因子，继续检查a的祖先结点

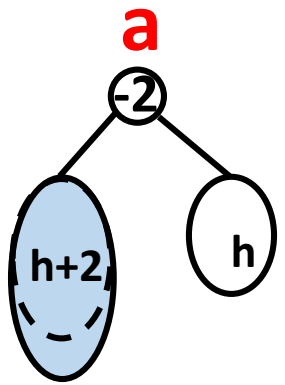


情况2：结点a原先不平衡，插入后变得平衡了

解决方法：树高不变，无需调整（对a结点及其祖先无影响）

情况3：结点a原先不平衡，插入后加重了不平衡

解决方法：对以a为根的子树进行旋转操作，使改子树恢复平衡，然后继续检查a的祖先？

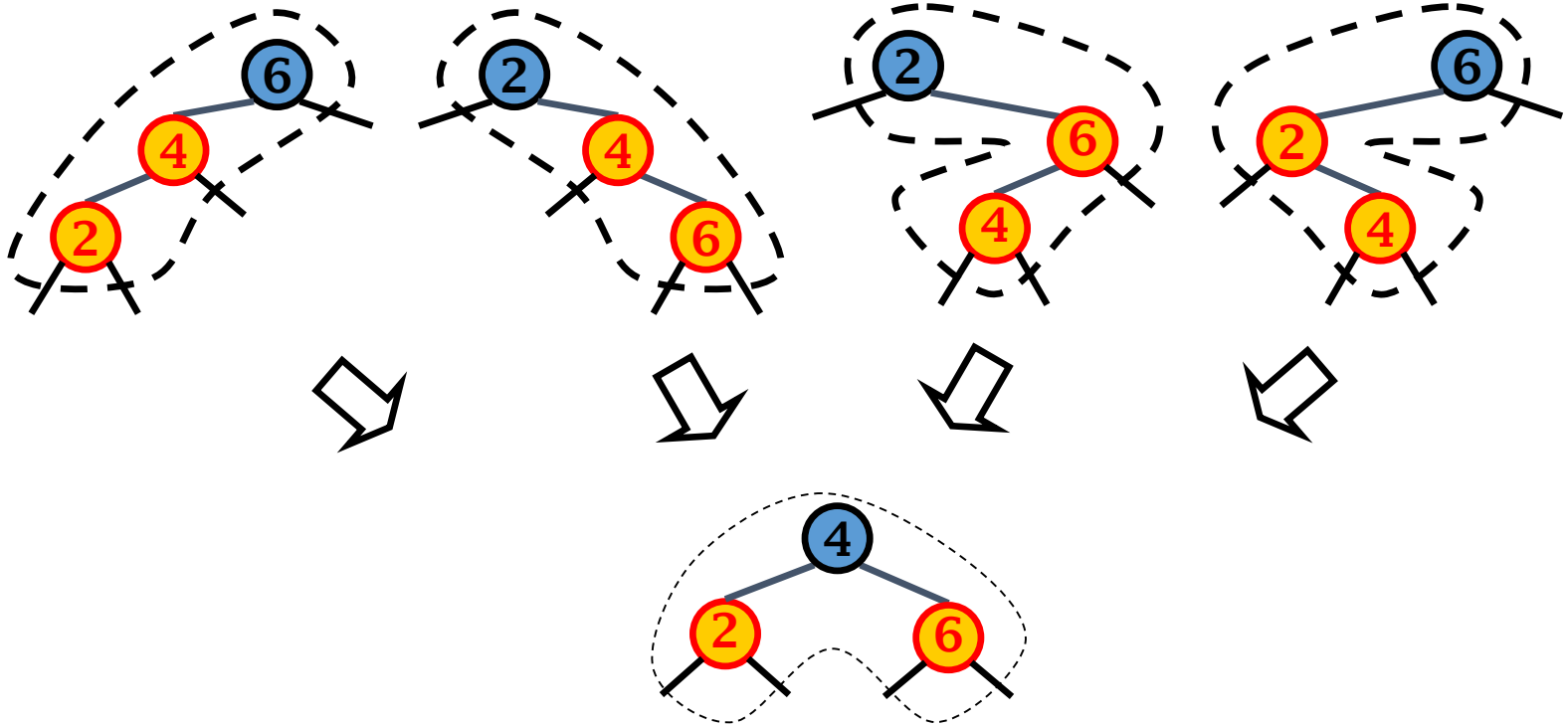


AVL 树结点的插入

- 只有情况3需要对当前子树（以a为根）进行旋转
- 又分为4种子情况
  - 子情况1（LL）：插入前bf(a)=-1，新插入结点在a的左孩子的左子树中
  - 子情况2（LR）：插入前bf(a)=-1，新插入结点在a的左孩子的右子树中
  - 子情况3（RL）：插入前bf(a)=1，新插入结点在a的右孩子的左子树中
  - 子情况4（RR）：插入前bf(a)=1，新插入结点在a的右孩子的右子树中
- 只可能存在这4种子情况
  - 思考：为什么?

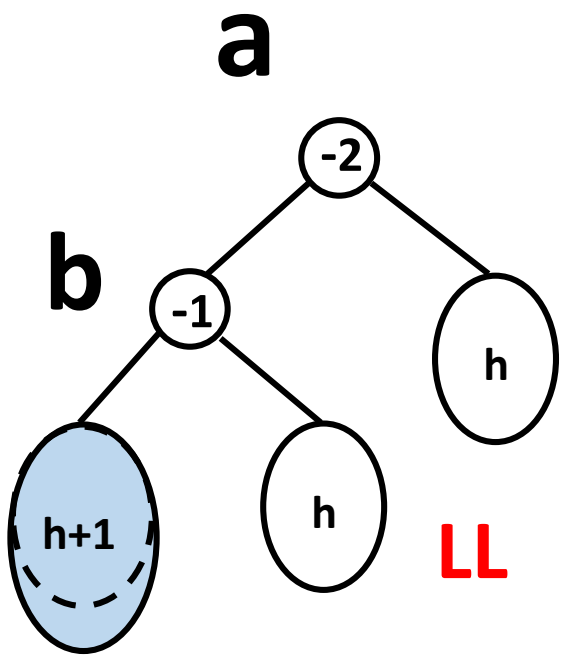
利用旋转恢复平衡

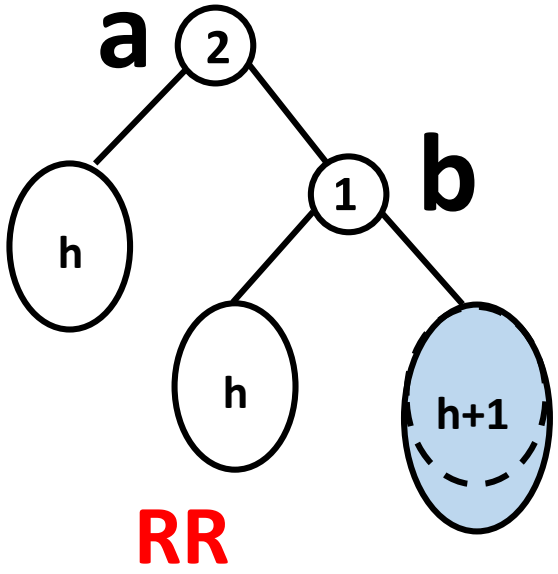
- 不同的情形采用不同的旋转方法
- 单旋转
  - 适用于LL与RR情形
- 双旋转
  - 适用于LR与RL情形



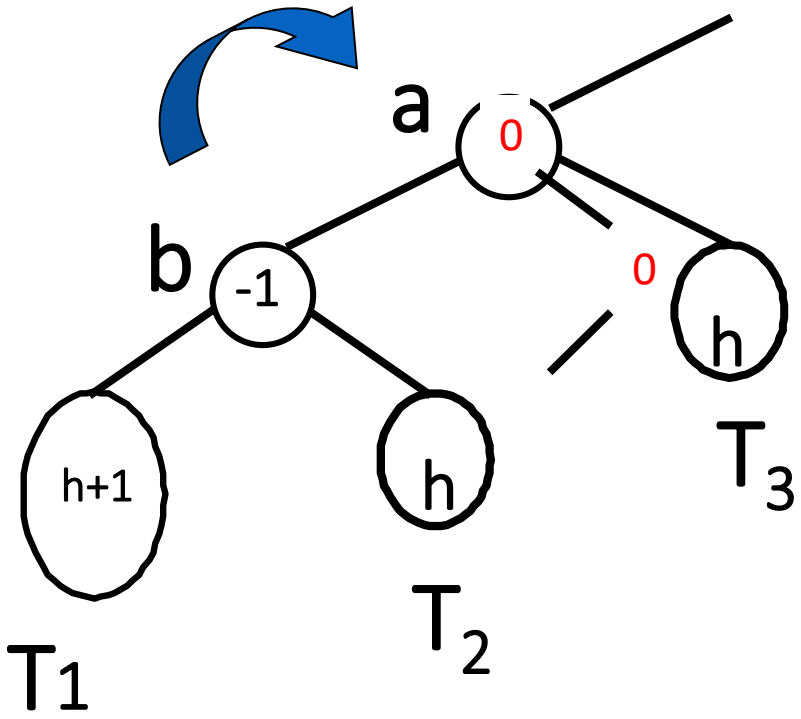
LL与RR恢复平衡：单旋转

- 结点b：结点a更高的子结点
  - 若bf(a) = -2，则b为a的左儿子
  - 若bf(a) = 2，则b为a的右儿子
- 旋转方法：单旋转交换a与b的位置



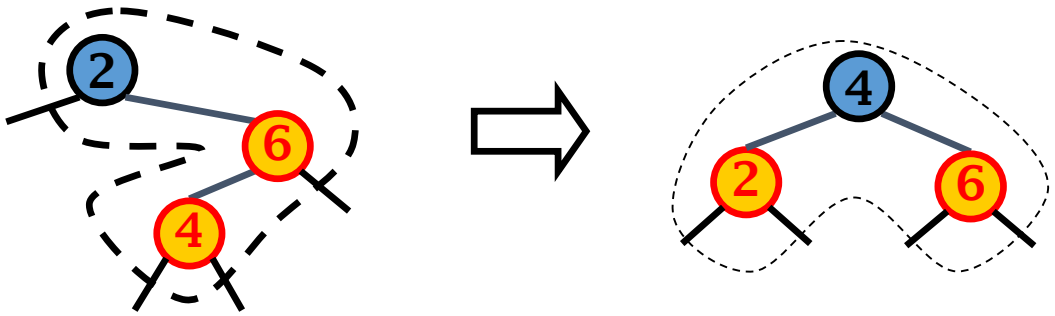


LL单旋转例子



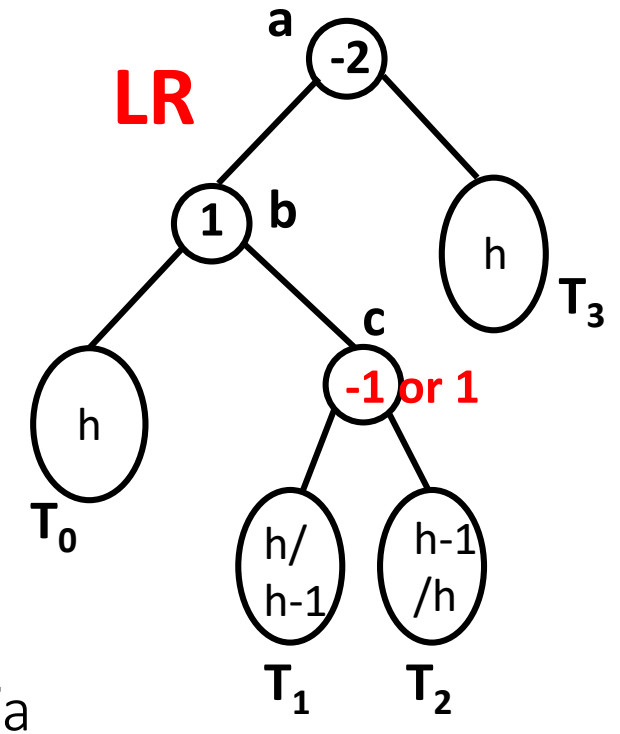
LR与RL恢复平衡：双旋转

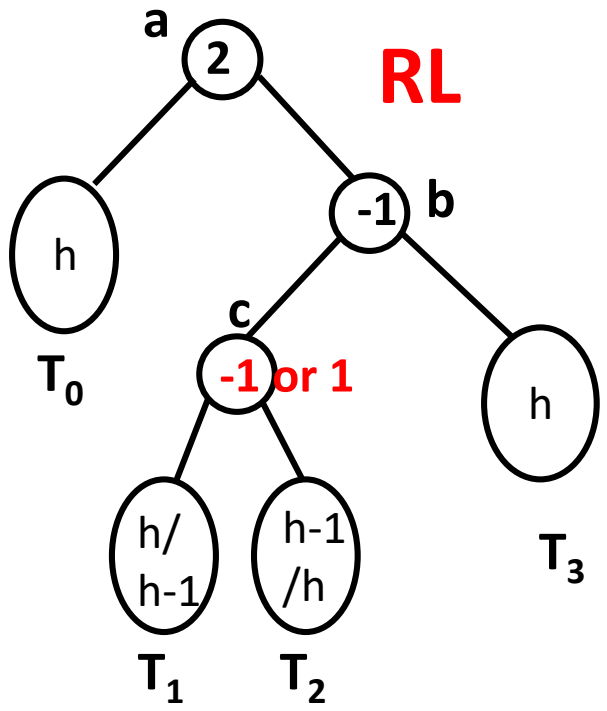
- RL 或者 LR 需要进行双旋转
  - 这两种情况是对称的
- 我们只讨论 RL 的情况
  - LR 类似地对称操作



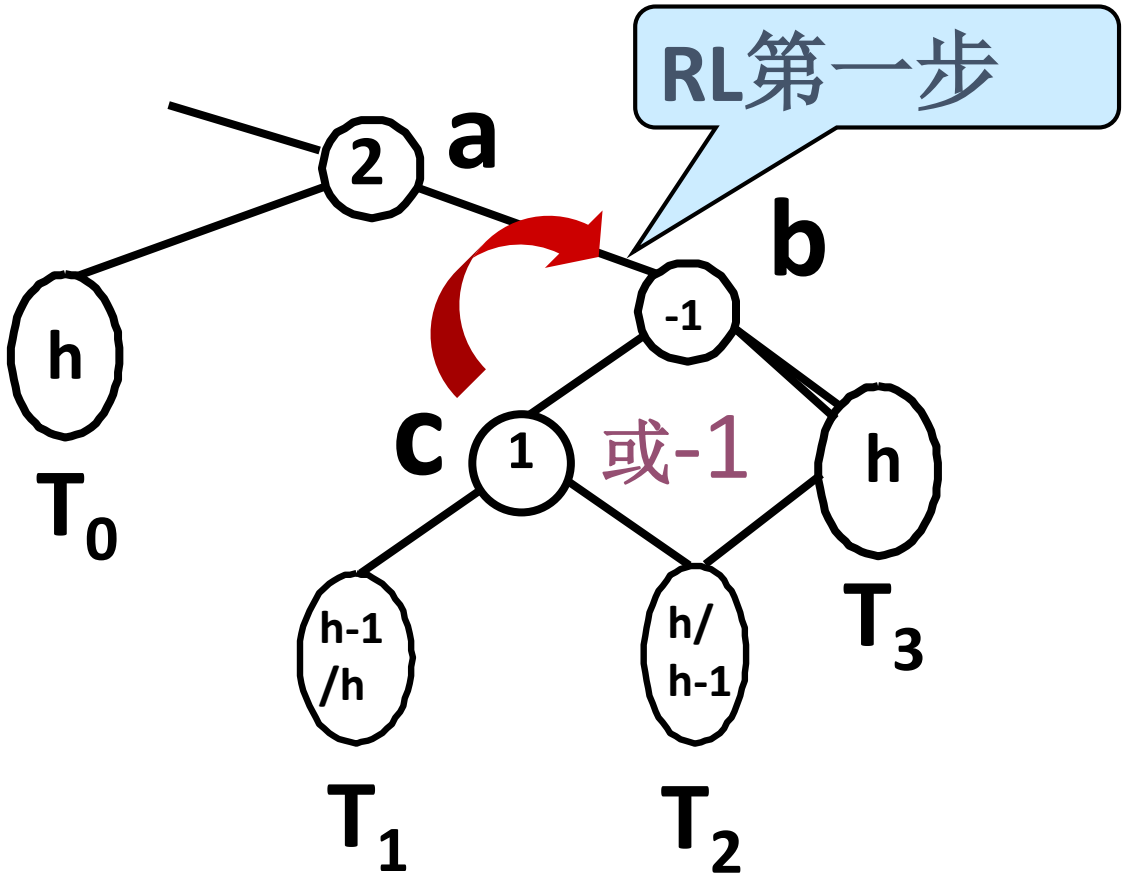
双旋转步骤

- 双旋转第1步：找到结点c，使得
  - c是b的一个子结点
  - c的BST顺序在a与b之间
  - 新插入结点一定在c为根的子树中
- 针对结点a, b, c进行双旋转
  - 本质上：先旋转b与c，然后旋转c与a





RL型双旋转第一步



插入前
a子树高h+2

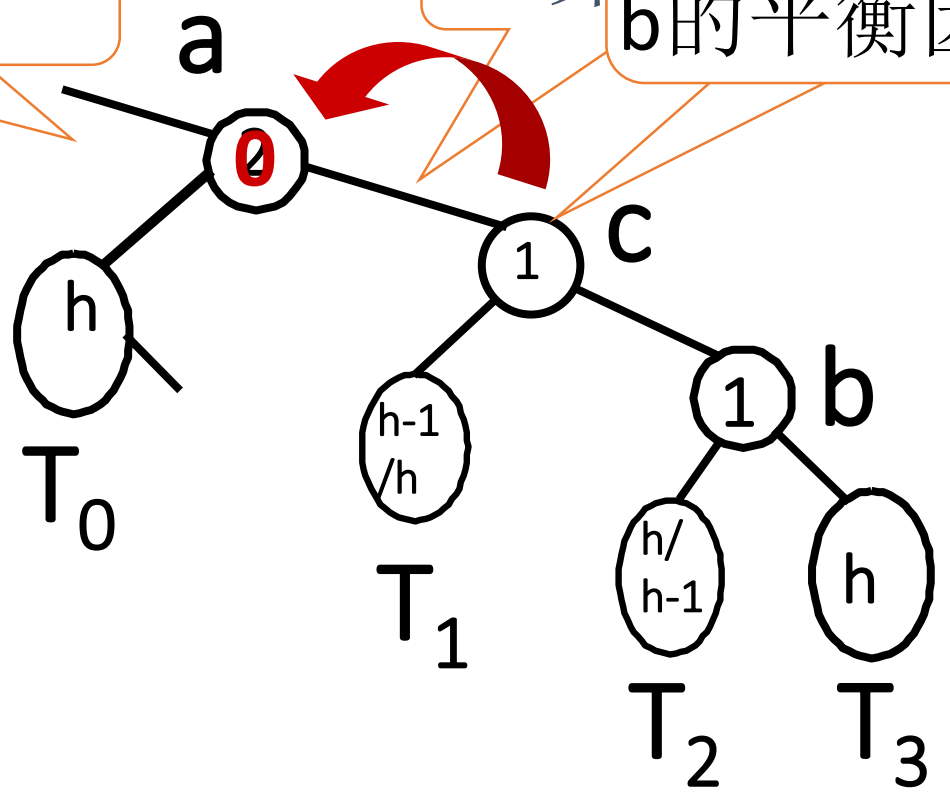
插入后
a子树高h+3

RL型双旋转第二步

中间状态
平衡因子无意义

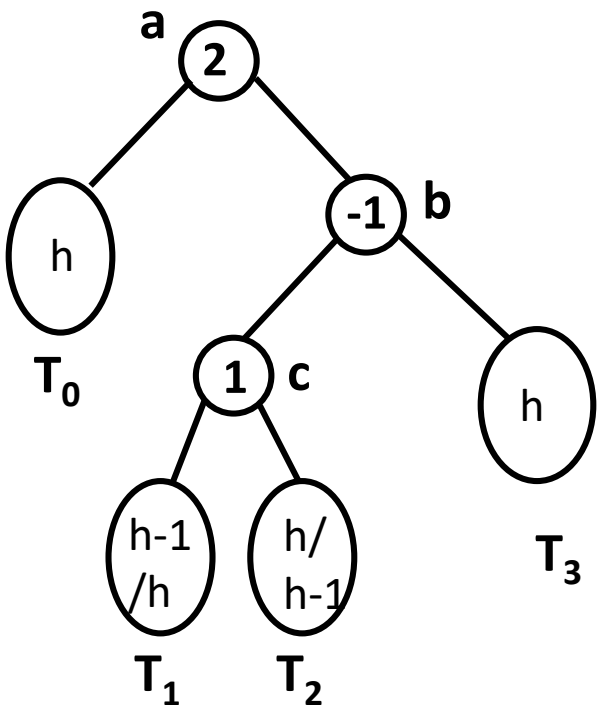
RL第

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

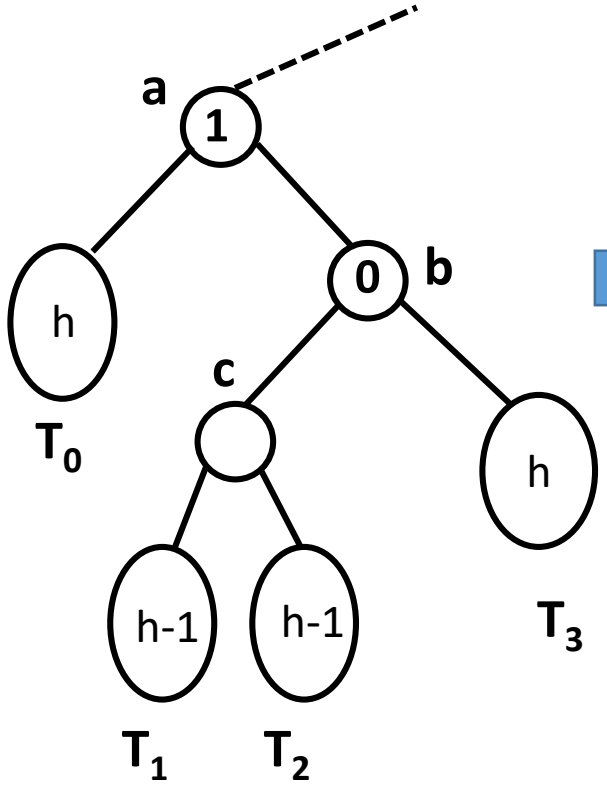
- 以 RL 型图示为例，总共有7个部分
  - 三个结点：a、b、c
  - 四棵子树 T_0、T_1、T_2、T_3
    - 加重 b 为根的子树，但是其结构其实没有变化
    - T_1、c、T_2 可以整体地看作 b 的左子树
- 目的：重新组成一个新的 AVL 结构
  - 平衡：以c作为新的根
  - 保留了中序遍历下有序的性质
    - T_0 a T_1 c T_2 b T_3



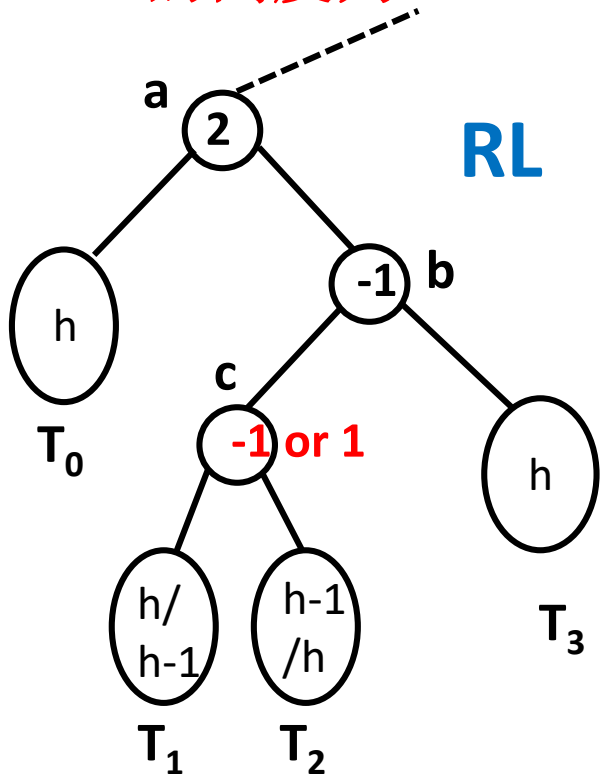
12.5 AVL树的基本概念与插入操作

旋转之后

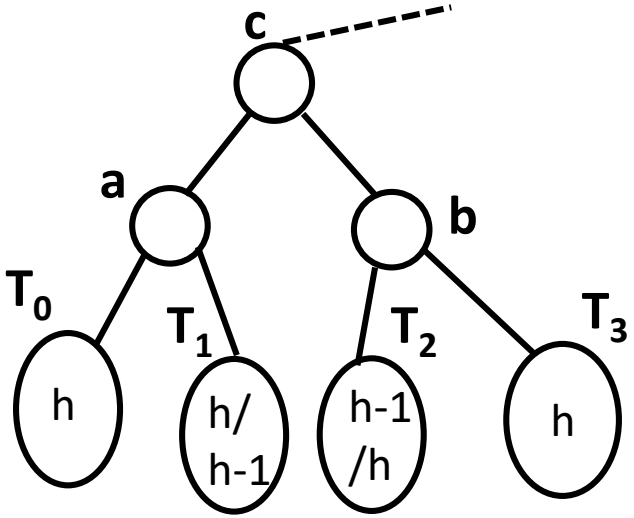
插入前:
a的高度为h+2



插入新结点，旋转之前:
a的高度为h+3

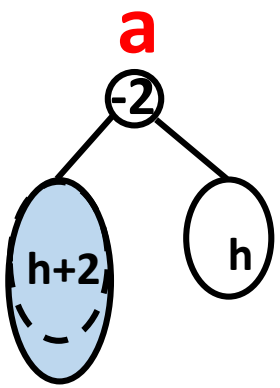


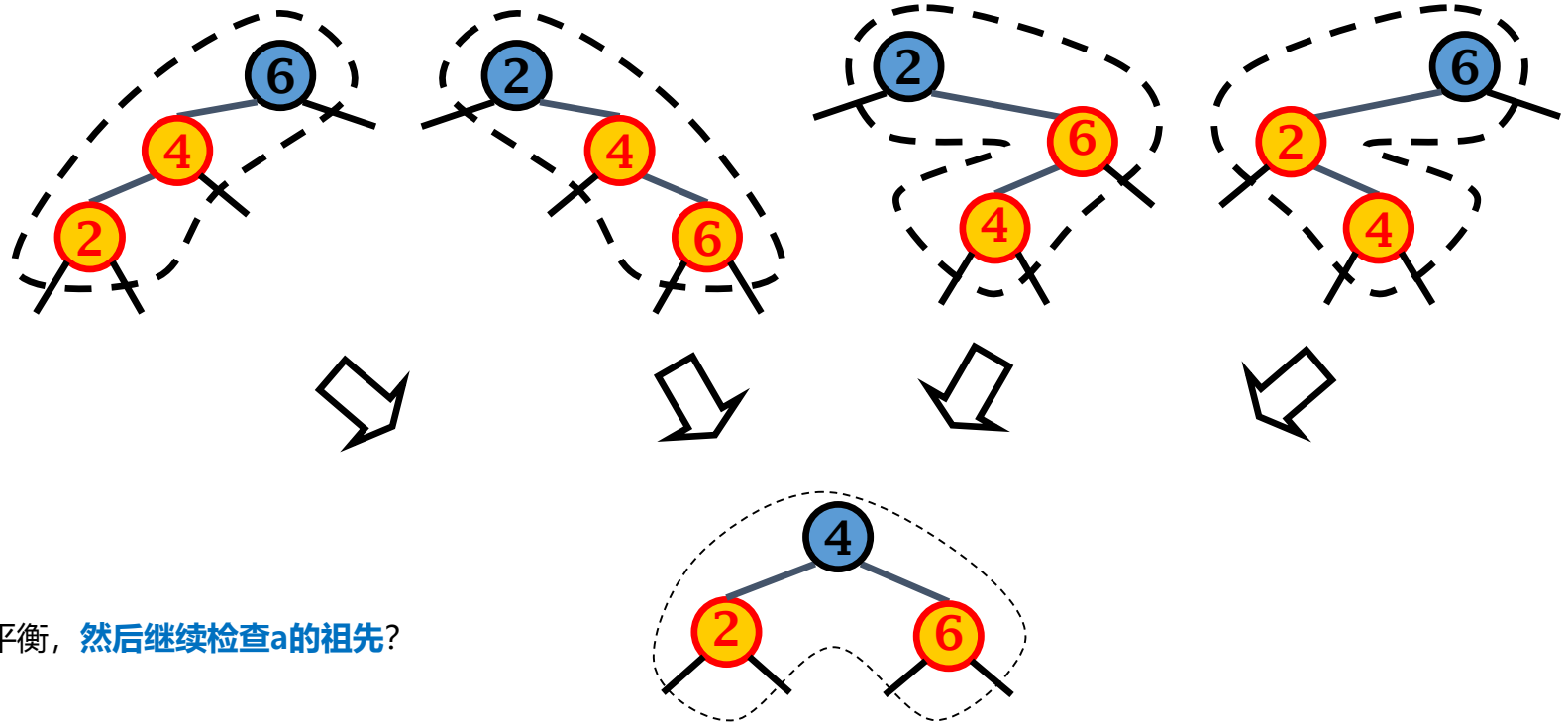
旋转之后:
新根c的高度为h+2



旋转之后

- 遗留问题



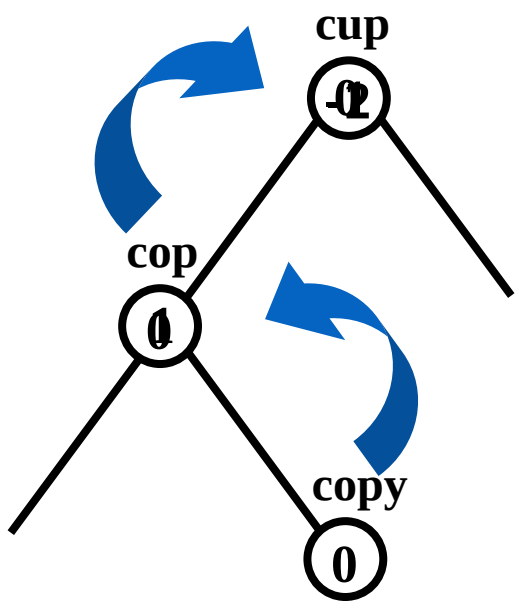


情况3：结点a原先不平衡，插入后加重了不平衡
解决方法：对以a为根的子树进行旋转操作，使改子树恢复平衡，**然后继续检查a的祖先**?

- 答案：旋转后子树高度与插入前一样，不需要进一步检查a的祖先

12.5 AVL树的基本概念与插入操作

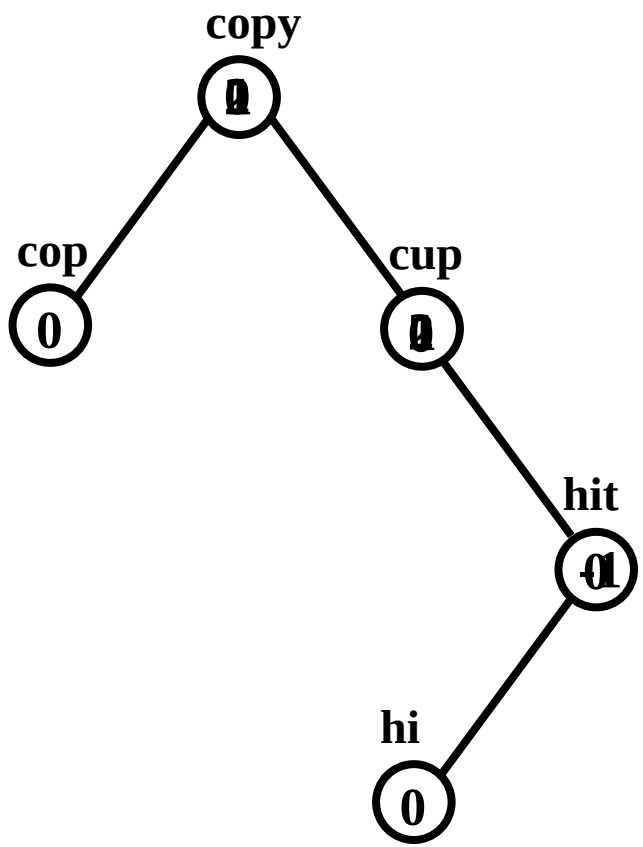
插入单词：cup，cop，copy，hit，hi，his 和 hia 后得到的 AVL 树



插入 copy 后不平衡
LR 双旋转

12.5 AVL树的基本概念与插入操作

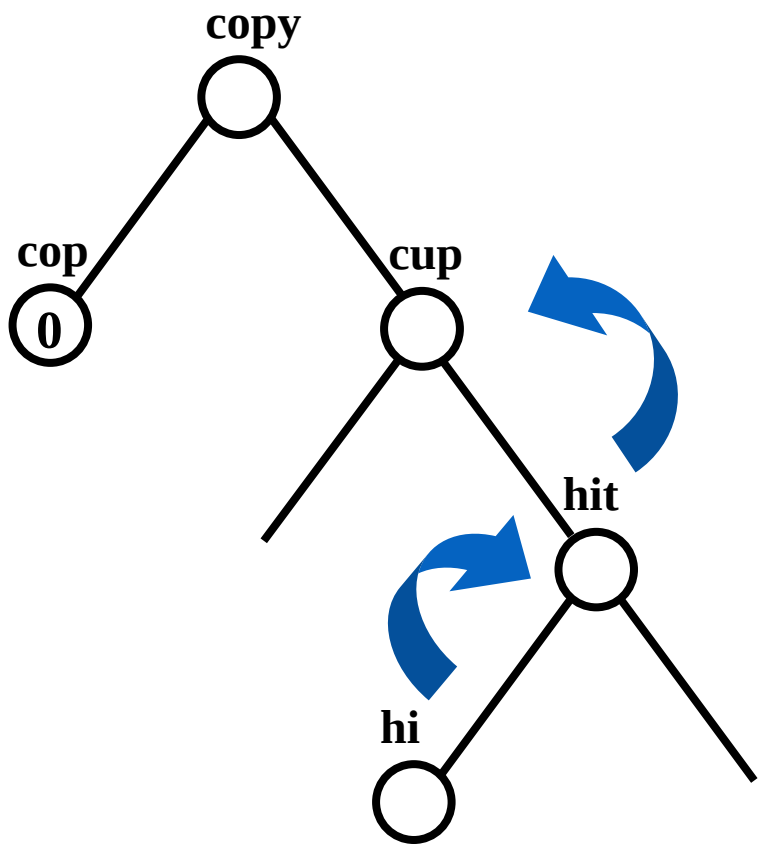
插入单词：cup，cop，copy，hit，hi，his 和 hia 后得到的 AVL 树



12.5 AVL树的基本概念与插入操作

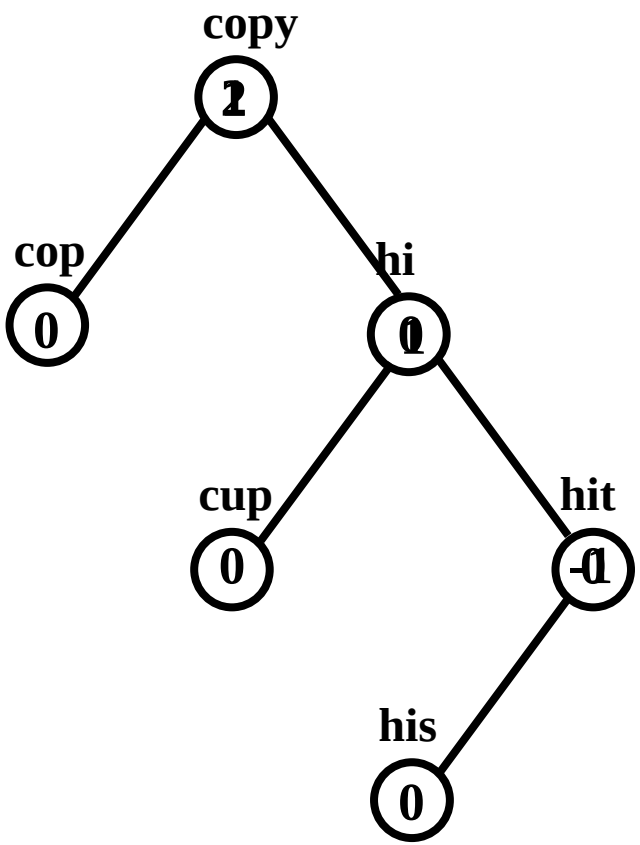
插入单词：cup，cop，copy，hit，hi，his 和 hia 后得到的 AVL 树

RL 双旋转



12.5 AVL树的基本概念与插入操作

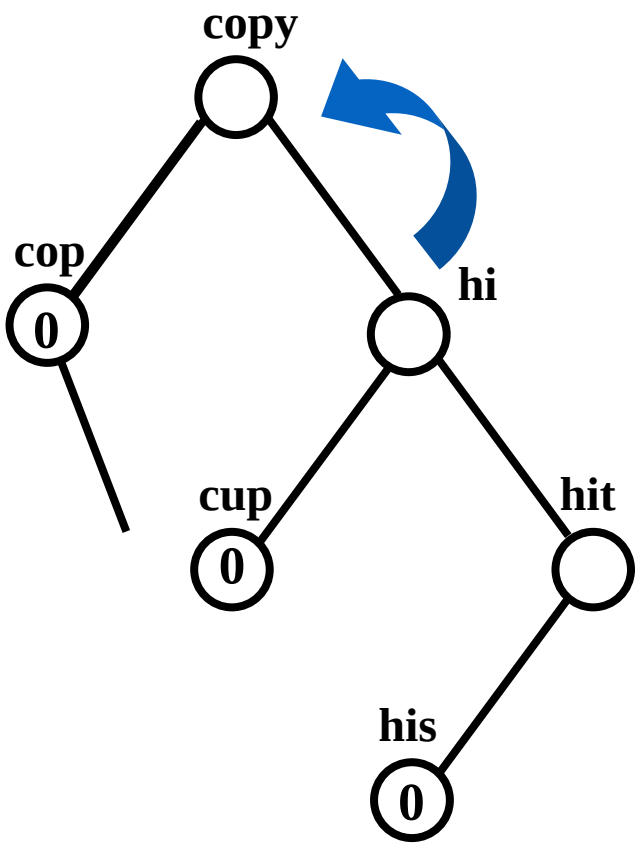
插入单词：cup，cop，copy，hit，hi，his 和 hia 后得到的 AVL 树



12.5 AVL树的基本概念与插入操作

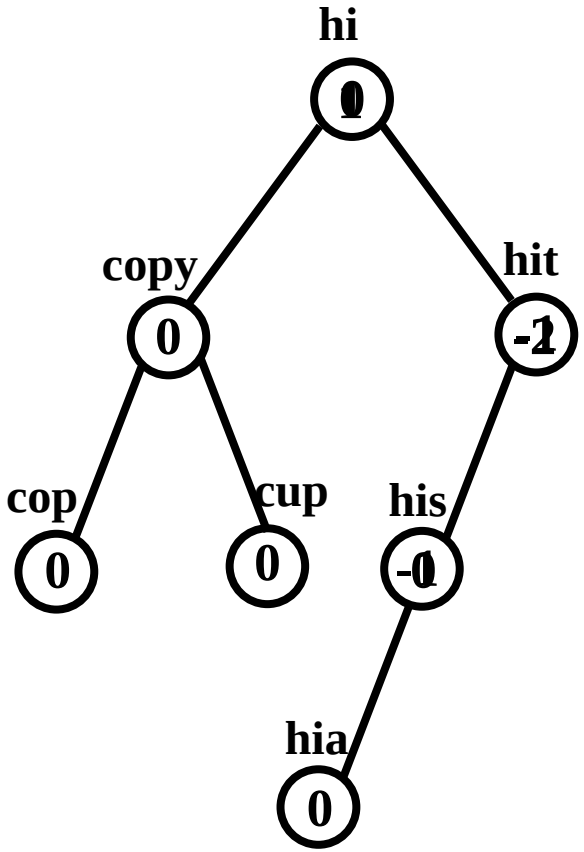
插入单词：cup，cop，copy，hit，hi，his 和 hia 后得到的 AVL 树

RR单旋转



12.5 AVL树的基本概念与插入操作

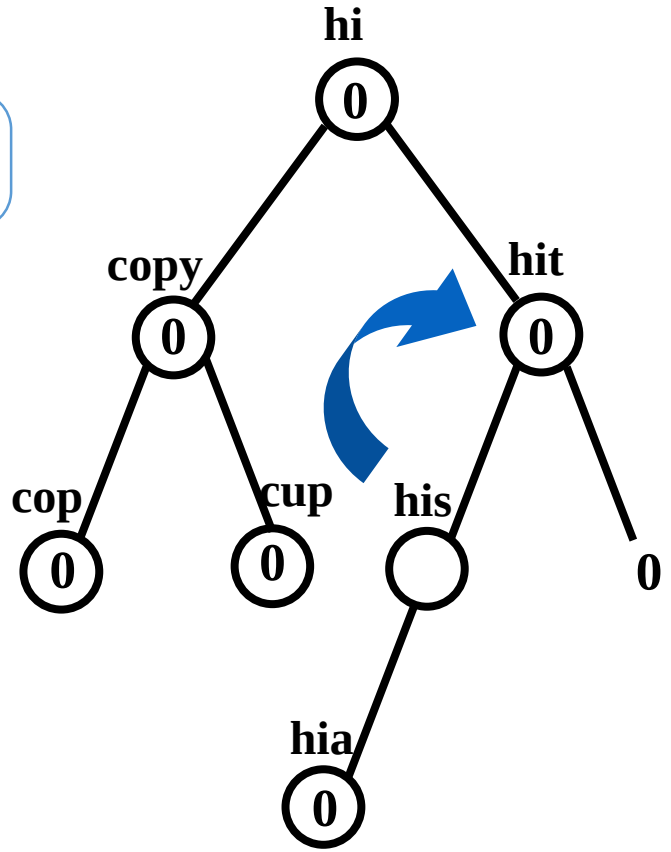
插入单词：cup，cop，copy，hit，hi，his 和 hia 后得到的 AVL 树



12.5 AVL树的基本概念与插入操作

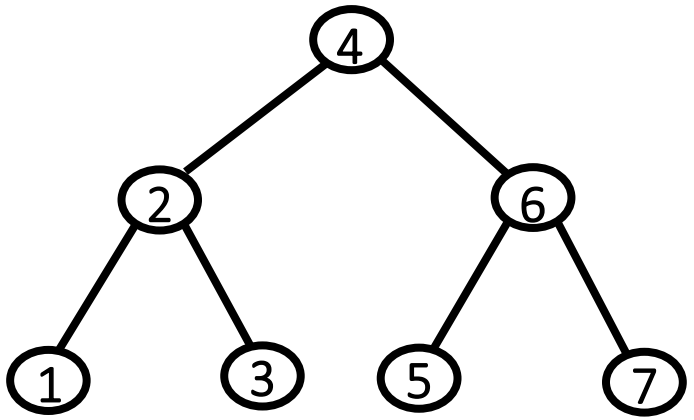
插入单词：cup，cop，copy，hit，hi，his 和 hia 后得到的 AVL 树

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。

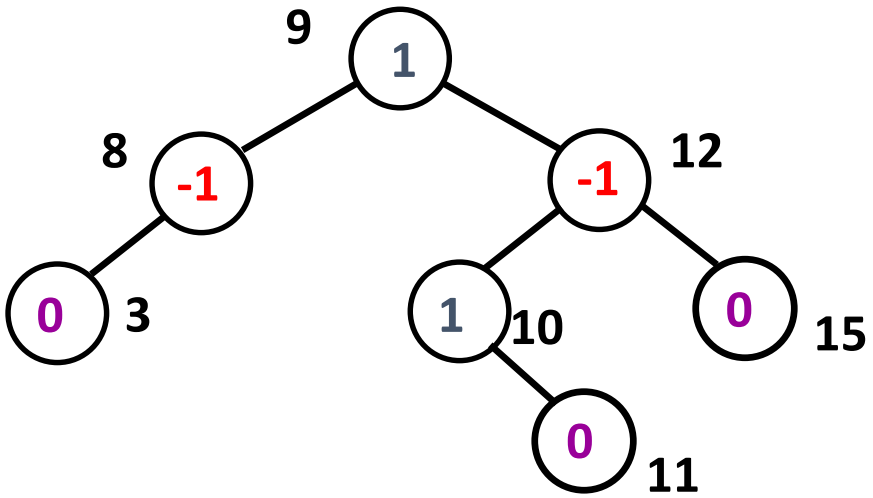


第十二章 高级数据结构

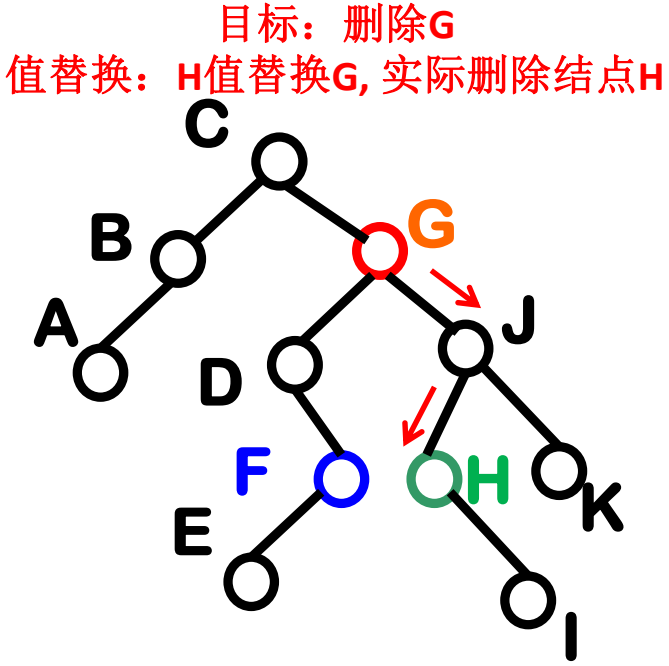
- 12.1 多维数组
- 12.2 广义表
- 12.3 存储管理
- 12.4 Trie 树
- 12.5 AVL树的概念与插入操作
- **12.6 AVL树的删除操作与性能分析**
- 12.7 伸展树

AVL 树结点的删除

- 删除是插入的逆操作
- AVL 树的删除结点的步骤与 BST 一样，但需要保证平衡性
  - AVL 树平衡因子 |bf| ≤ 1：
    - $bf\ (\mathrm{x}) = height(x_{rchild}) - height(x_{lchild})$



AVL树的结点删除

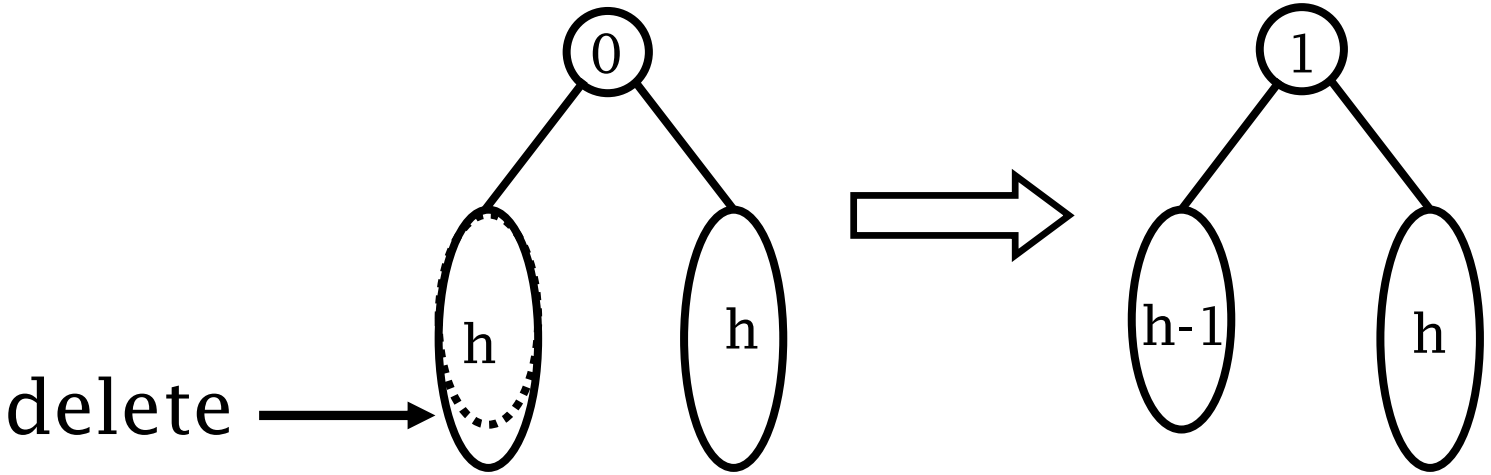


- 步骤1：像普通二叉搜索树一样删除结点
  - 注意：二叉搜索树有多种结点删除方法，我们采用值替换方法
  - 值替换方法：结点删除后，二叉树中结点高度要么不变，要么降低1

- 步骤2：检查从实际删除结点（例子中原先的H结点）到根结点路径
  - 检查结点子树高度是否降低，重新计算平衡因子
  - 若某个结点x的平衡性被破坏，通过旋转操作恢复平衡
  - 不断遍历每个结点，直到
    - 到达根结点，或者
    - 路径上的某个结点y高度不变
  - **根据平衡性的变化，有以三种情况**

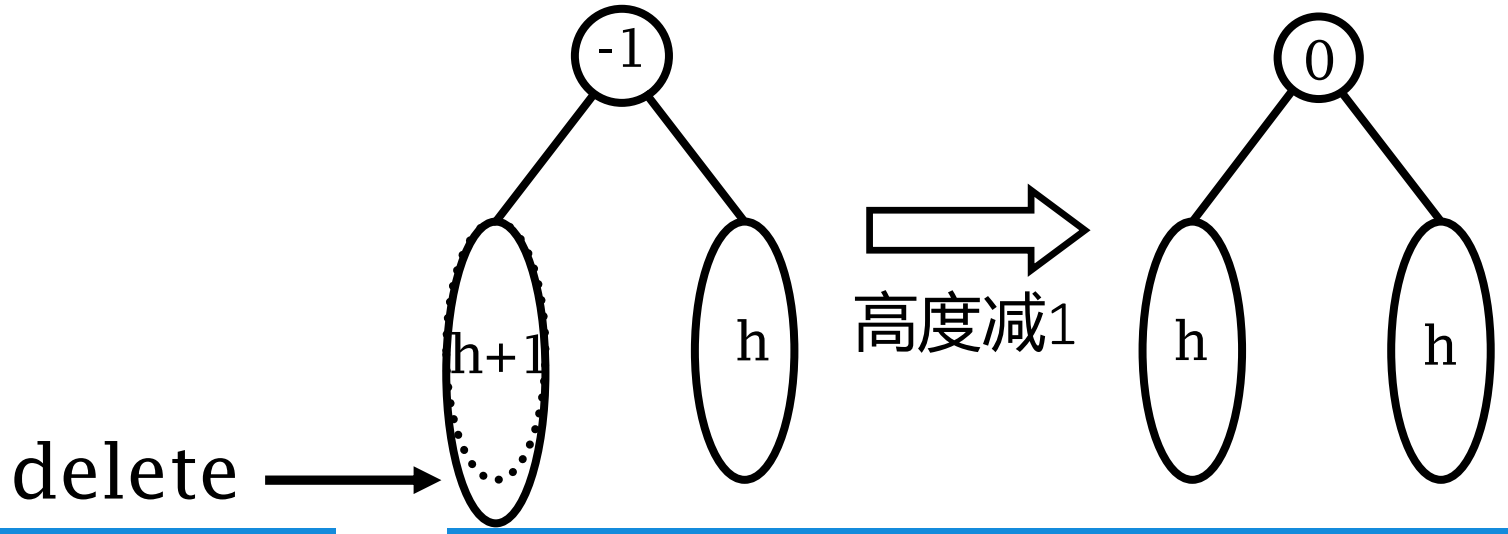
AVL树的结点删除：情况1

- 删除前：结点a 平衡因子为 bf(a)=0
  - 删除后：其左或右子树被缩短，则平衡因子该为 1 或者 -1
- 操作策略：
  - 修改结点a的平衡因子
  - 删除操作虽然引起高度变化，但变化不会影响到更高层的结点，调整可以结束



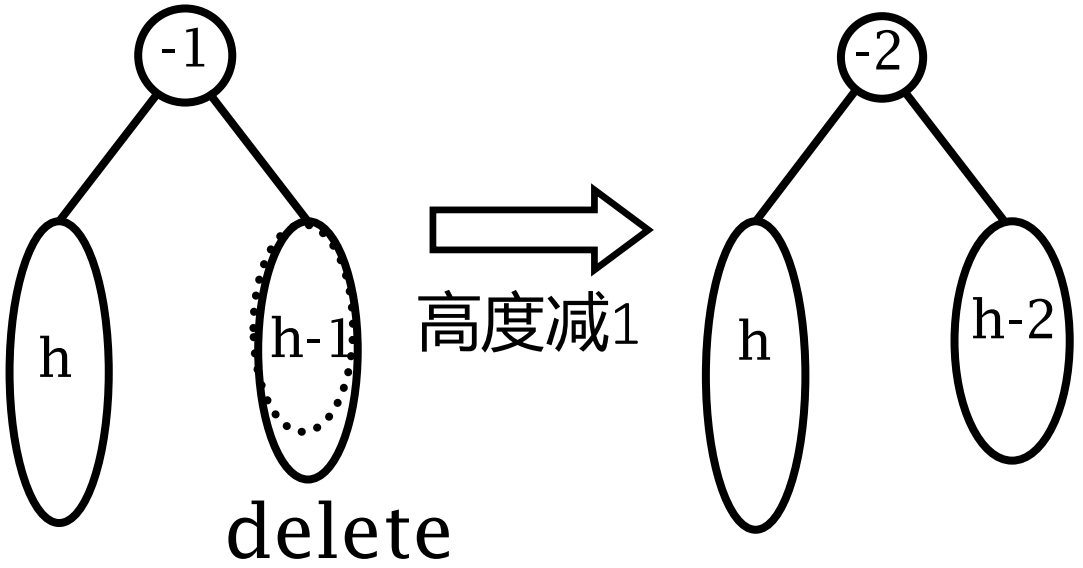
AVL 树结点的删除：情况2

- 删除前：结点 a 平衡因子不为 0，
  - 删除后：较高的子树被缩短，结点a的平衡因子修改为 0
- 操作策略：
  - 修改结点a的平衡因子
  - 需要继续向上修改（该子树高度降低，可能影响某个祖先的平衡性）



AVL 树结点的删除：情况3

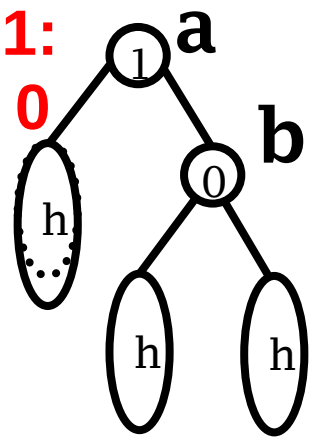
- 删除前：结点 a 平衡因子不为 0
  - 删除后：它的较矮的子树被缩短，a变的更加不平衡，破坏AVL性质
- 操作策略
  - 通过旋转使以a为根的子树恢复平衡
  - 如果旋转后高度不变，则终止



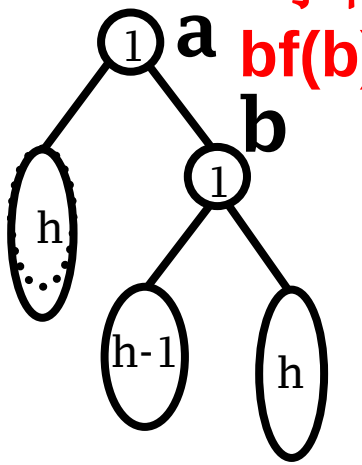
AVL树的删除：情况3的三种子情况

- a较矮的子树被缩短，看 a 较高子树根 b的情况
  - 子情况 3.1： b 的平衡因子为 0
  - 子情况 3.2： b 的平衡因子与 a 的平衡因子相同
  - 子情况 3.3： b 和 a 的平衡因子相反

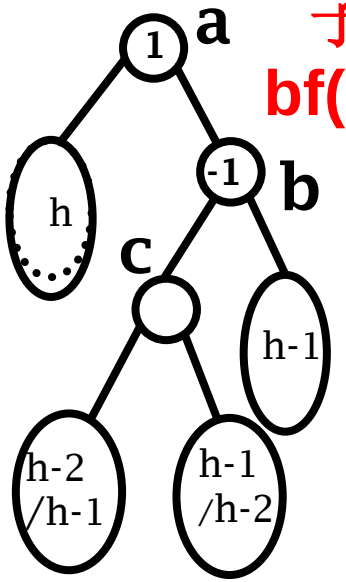
子情况3.1:
bf(b) == 0



子情况 3.2:
bf(b) == bf(a)

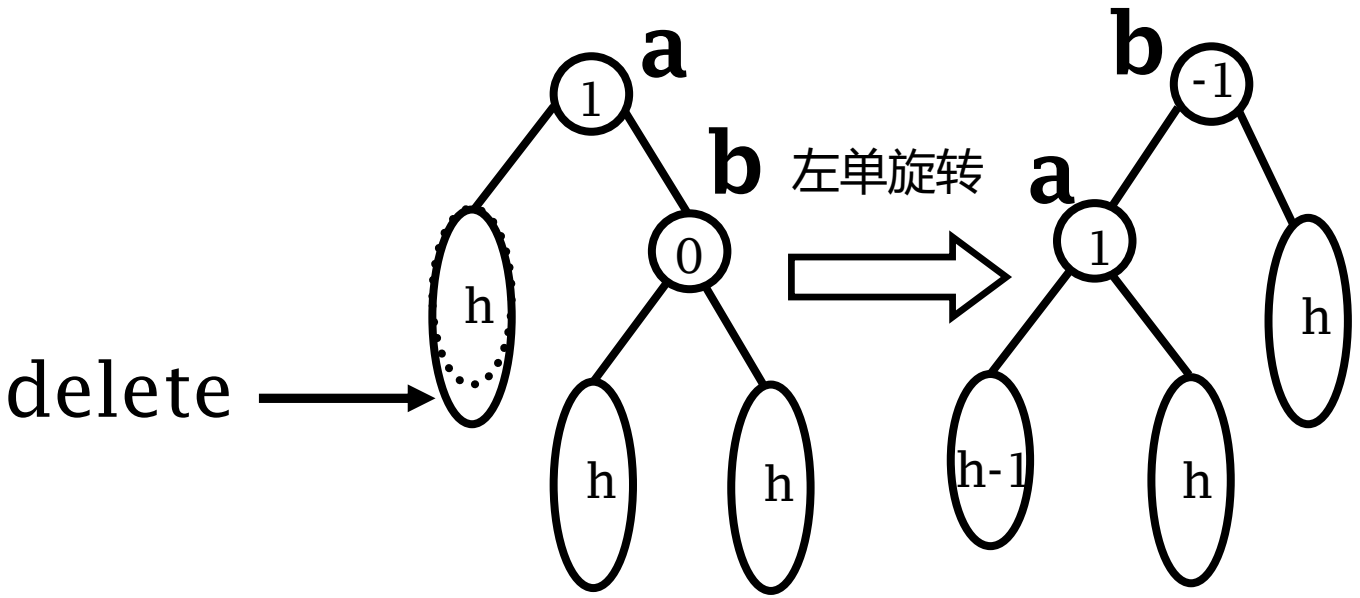


子情况 3.3:
bf(b) == -bf(a)



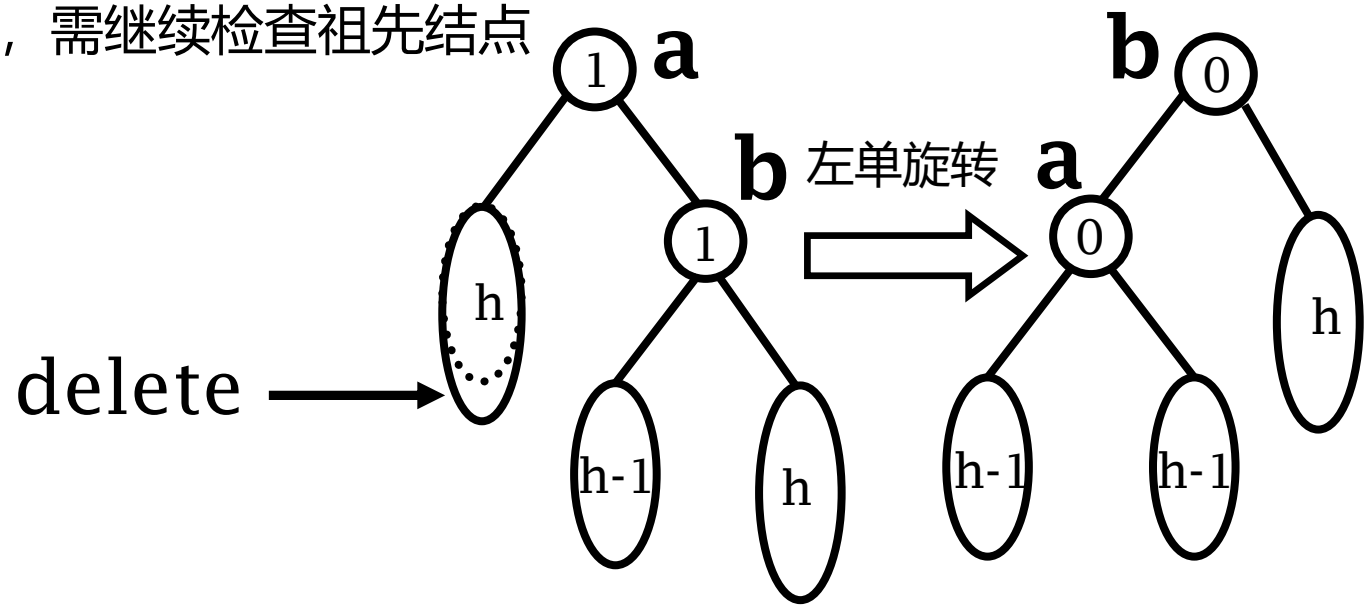
AVL 树结点的删除：子情况 3.1

- a 较矮的子树被缩短，a 较高子树根 b的 **bf(b) == 0**
- 旋转方法：b与a进行单旋转
- 旋转后：该子树高度不变，不需继续检查祖先结点



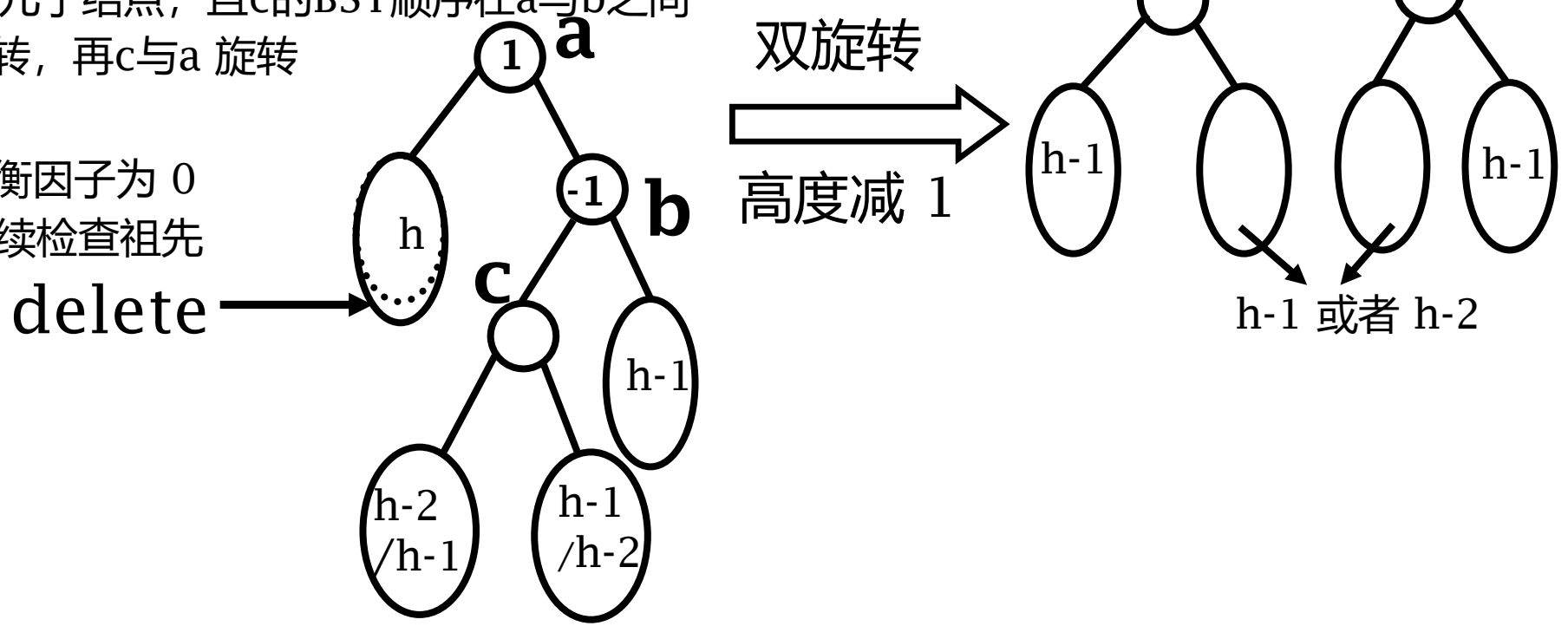
AVL 树结点的删除：子情况3.2

- a 较矮的子树被缩短，a 较高子树根 b的 **bf(b) == bf(a)**
- 旋转方法：结点b与a单旋转
- 旋转后：
  - 结点 a、b 平衡因子都变为0
  - 子树高度降1，需继续检查祖先结点



AVL 树结点的删除：子情况 3.3

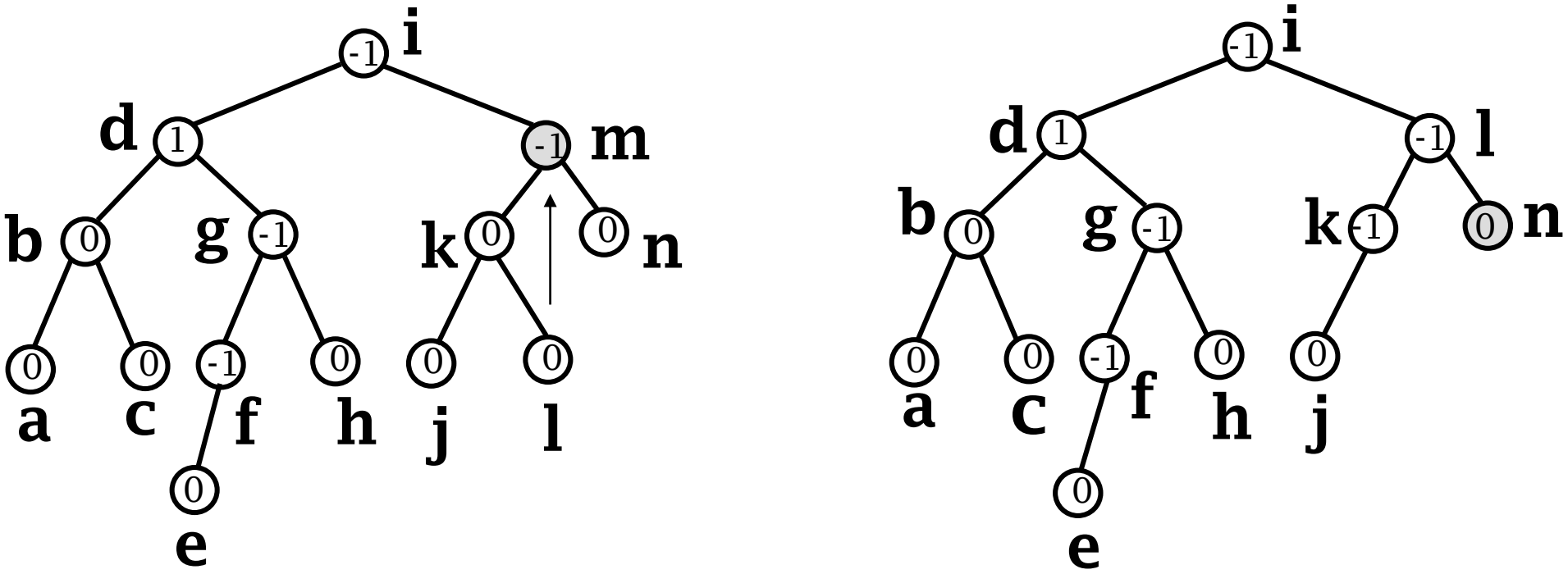
- a 较矮的子树被缩短，a 较高子树根 b的 **bf(b) == -bf(a)**
- 旋转方法：
  - 找到结点c，c为b的儿子结点，且c的BST顺序在a与b之间
  - 双旋转，先c与b 旋转，再c与a 旋转
- 旋转后
  - c为新的根结点，平衡因子为 0
  - 子树高度降1，需继续检查祖先



删除后的连续调整

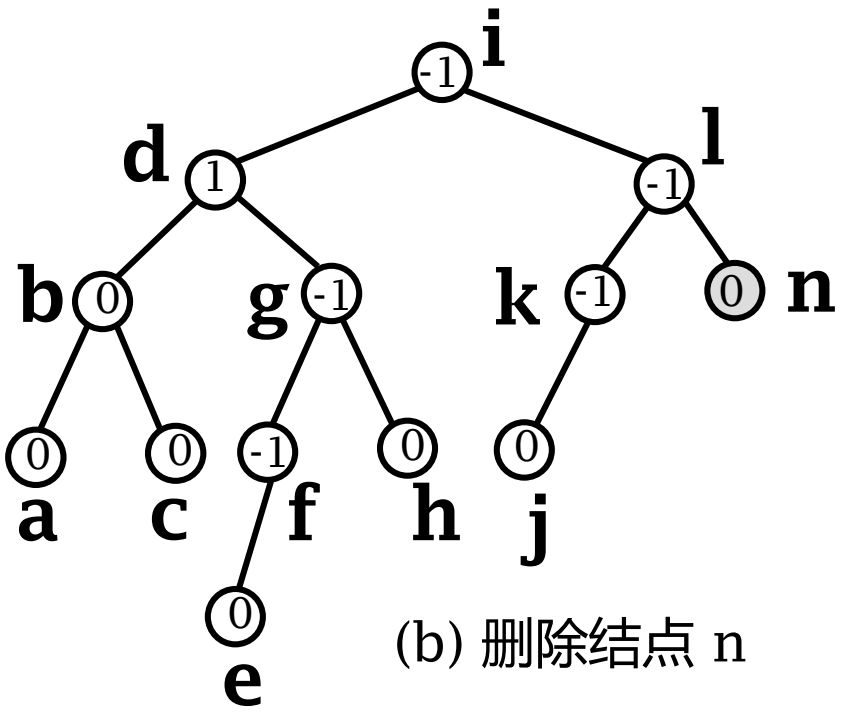
- 连续调整
  - 调整可能导致祖先结点发生新的不平衡
  - 这样的调整操作要一直进行下去，可能传递到根结点为止
- 从被删除的结点向上查找到其祖父结点
  - 然后开始单旋转或者双旋转操作
  - 旋转次数为 O (log n)
- 思考：对比插入操作
  - 插入操作：旋转后高度不变，不需继续检查祖先
  - 删除操作：旋转后可能降低高度（子情况3.2与3.3），需要继续调整

AVL 树删除的例子

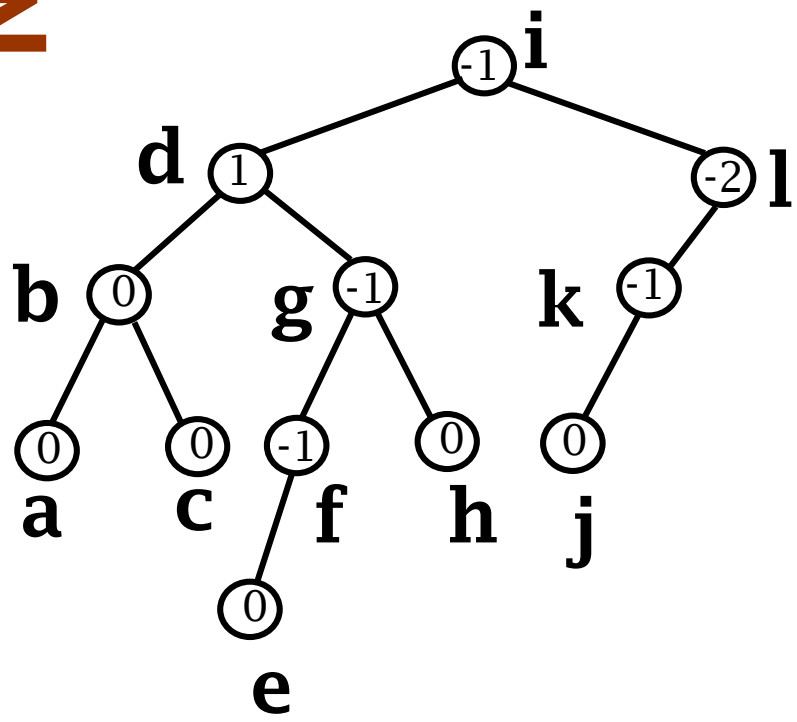


(a) 删除结点 m，则需要使用其中序前驱 l 代替（情况1）

AVL 树删除的例子

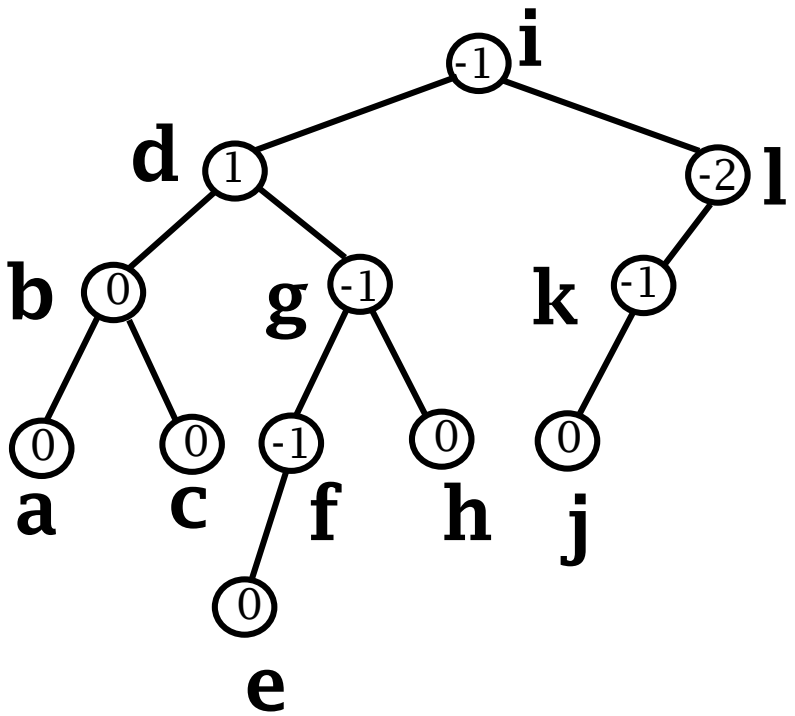


(b) 删除结点 n



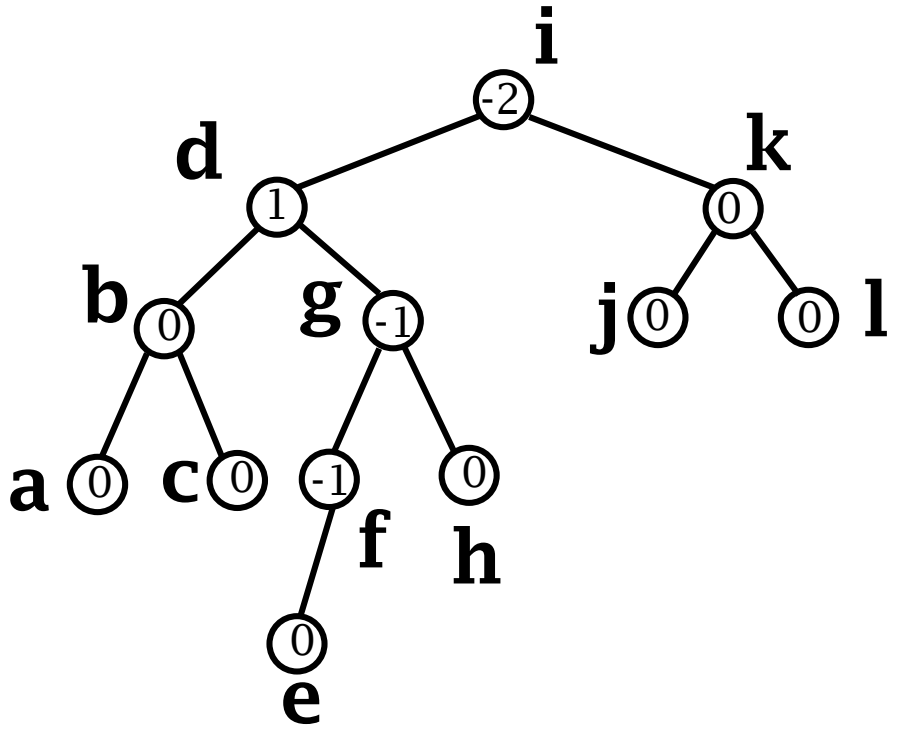
(c) 需要以 l 为根进行 LL 单旋转 (情况 3.2)

AVL 树删除的例子

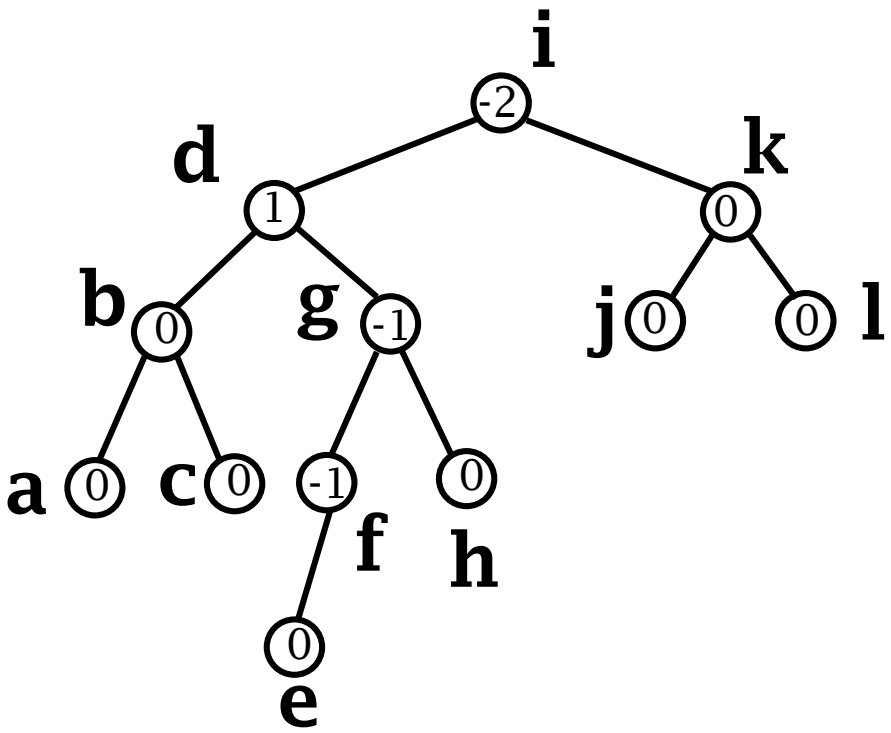


(c) 需要以 l 为根进行 LL 单旋转（情况 3.2）

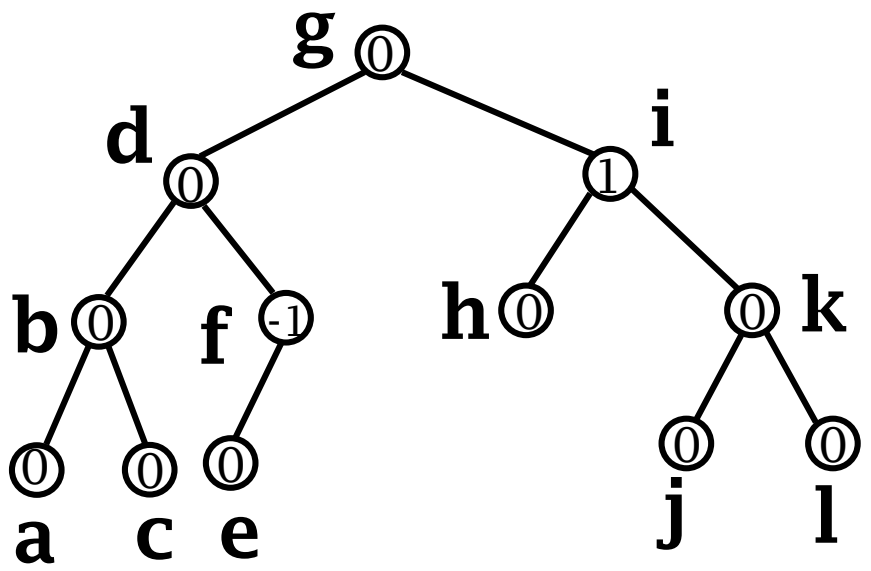
(d) LL 单旋转完毕，回溯调整父结点 i，需要以 i 为根的 LR 双旋转（情况3.3）



AVL 树删除的例子



(d) LL 单旋转完毕，回溯调整父结点 i，需要以 i 为根的 LR 双旋转（情况3.3）

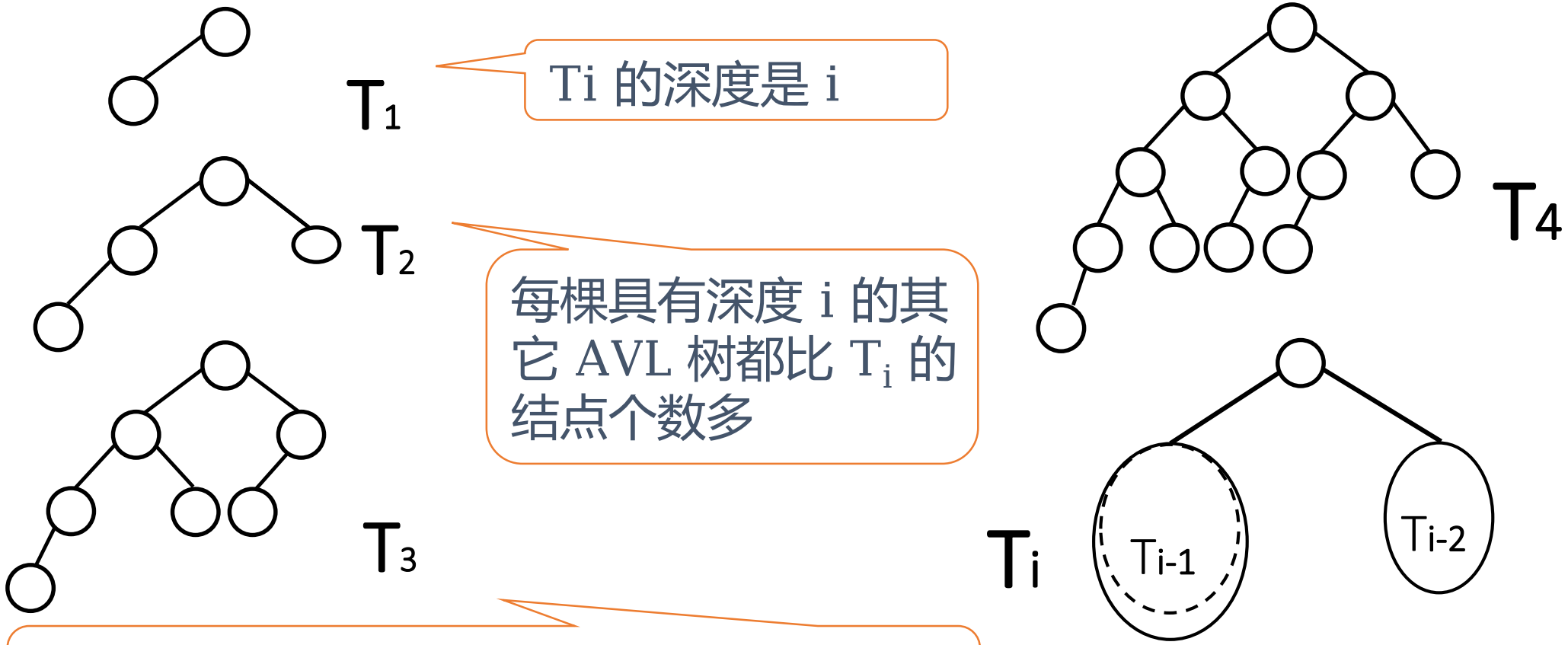


(e) 调整完毕，AVL 树重新平衡

AVL树的复杂度

- 查询、插入、删除操作的复杂度，均取决于最坏情况下AVL树的深度
- AVL树的最坏深度是什么?
- 分析方法：考虑最接近于不平衡的 AVL 树
  - 使得每个结点的平衡因子都是-1
  - 递归构造方法：构造一系列**临界** AVL 树 T_1，T_2，T_3，…

12.6 AVL树的删除操作与性能分析



Ti 的深度是 i

每棵具有深度 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

深度的证明（推理）

- 让t(h)表示深度为h的最不平衡AVL树中的结点个数
- 可看出有下列关系成立：

$$t(1) = 2$$
$$t(2) = 4$$
$$t(i) = t(i-1) + t(i-2) + 1$$

- 对于 i>2 此关系很类似于定义 Fibonacci 数的关系：

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

深度的证明 (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

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

深度的证明 (结果)

- 解出深度 i 与结点个数 t (i) 的关系

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

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

- 由换底公式 $\log_{\varphi}X = \log_2 X / \log_2 \varphi$ 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 树的深度一定是 O (log n)

AVL 树的深度

- 具有 n 个结点的 AVL 树深度一定是 O (log n)
- AVL树查询、插入、删除最坏复杂度O（log n)

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

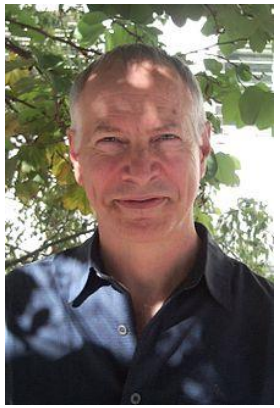
第十二章 高级数据结构

- 12.1 多维数组
- 12.2 广义表
- 12.3 存储管理
- 12.4 Trie 树
- 12.5 AVL树的概念与插入操作
- 12.6 AVL树的删除操作与性能分析
- **12.7 伸展树**

伸展树概念

- 1985年由Daniel Sleator与Robert Tarjan共同发明
- 广泛应用
  - Windows NT系统内核中记录进行信息
  - 输入法提示词：最近输入的词汇

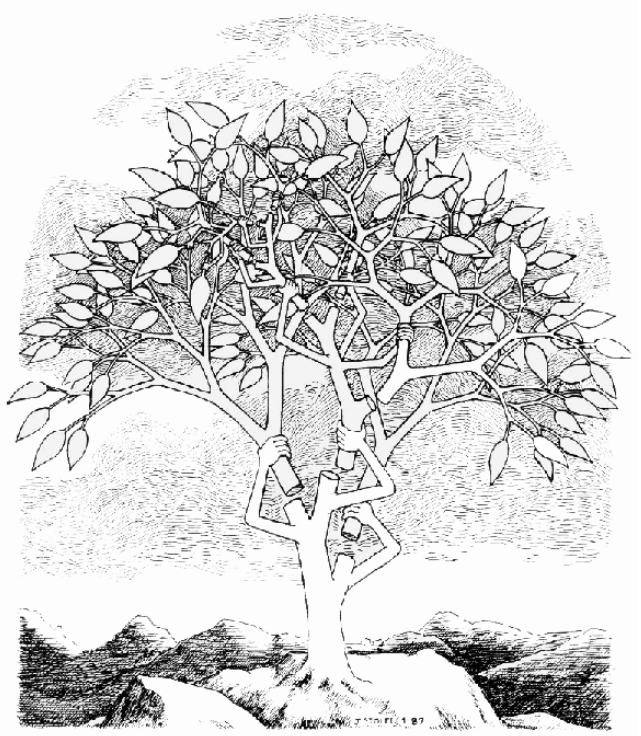
Daniel Sleator



Robert Tarjan
(1986年图灵奖得主)

伸展树概念

- 一种自组织数据结构
  - 数据随访问而调整位置
- 伸展树是改进 BST 性能的一组规则
  - 保证访问的总代价不高，达到令人满意的性能
  - 不能保证最终树高平衡



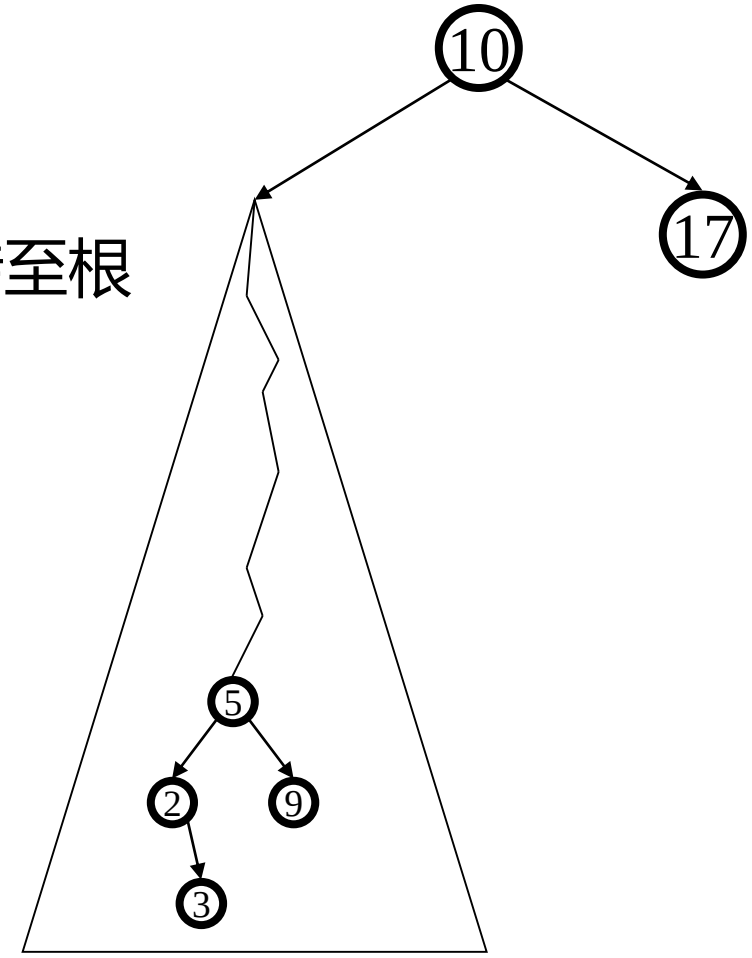
A Self-Adjusting Search Tree

伸展树核心思想

- “较为随意”的再平衡：不需要考虑高度信息
  - 与红黑树、AVL树的最大区别
- 旋转操作：无论查询、插入、删除，总是把访问结点旋转至根

- 局部性假设
  - 一个结点被访问后，很有可能未来被再次访问
  - 把访问后的结点旋转至根部，让它在未来被容易找到
  - 树根附近：大量被频繁访问的结点

- 单个操作最坏复杂度O(n)
- 一系列操作的均摊复杂度O(log n)



伸展树的操作

- 基本操作：查询、插入、删除

- 查询x或者插入x：把x旋转至根结点
- 删除x：把被删结点的父结点旋转至根结点
  - 可以是x的父，也可以是只替换x之后真正被删结点的父

- 展开(Splaying)操作：通过一系列旋转操作将某个结点x旋转到根结点
  - 每次旋转都让x在树中的位置变得更高

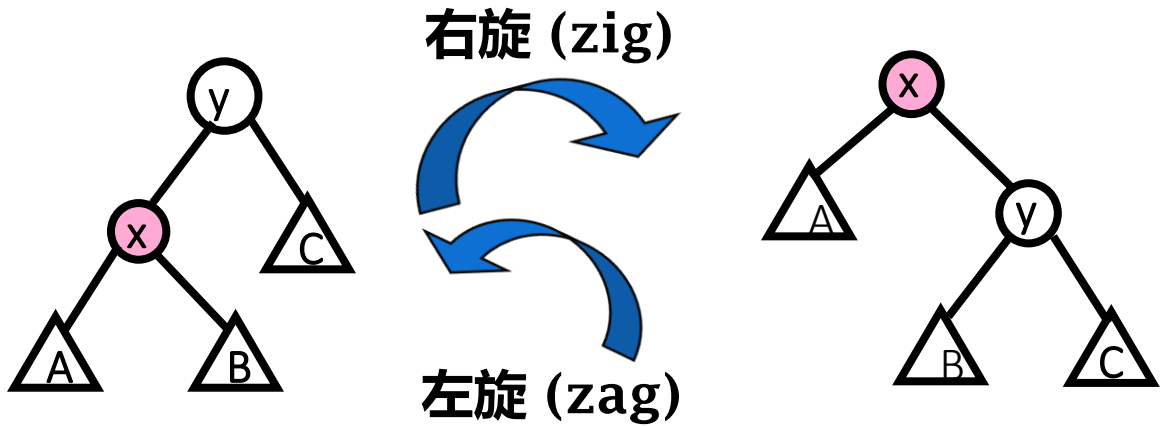
伸展树中的旋转

- 单旋转：每次将目标结点上升1层
  - 与AVL一样，交换结点与其父结点的位置
  - 在伸展树中的使用：仅当一个结点的父结点为根时使用

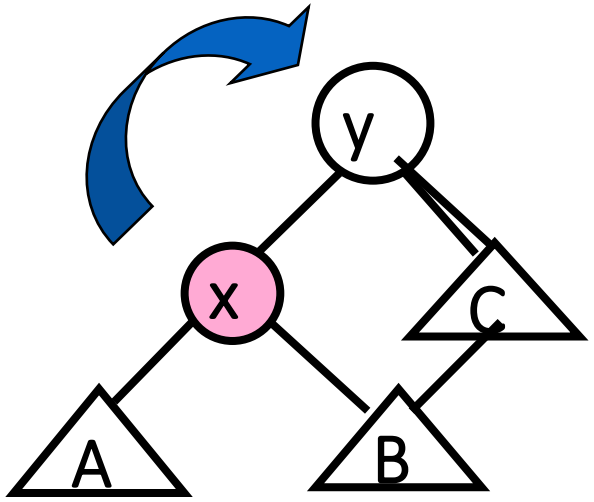
- 双旋转：每次将目标结点上升2层
  - 之字型旋转：与AVL树中一样
  - 一字型旋转：伸展树中引入

单旋转 (single rotation)

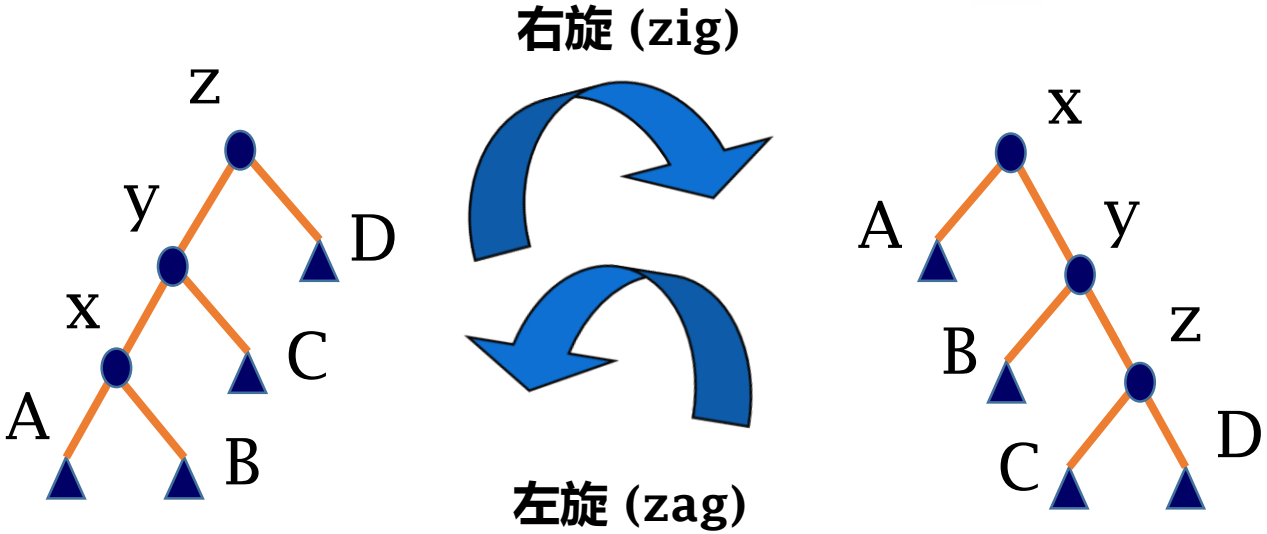
- x 是根结点y的直接子结点时
  - 把结点 x 与它的父结点y交换位置
  - 保持 BST 特性



A、B、C 代表子树，有大小顺序

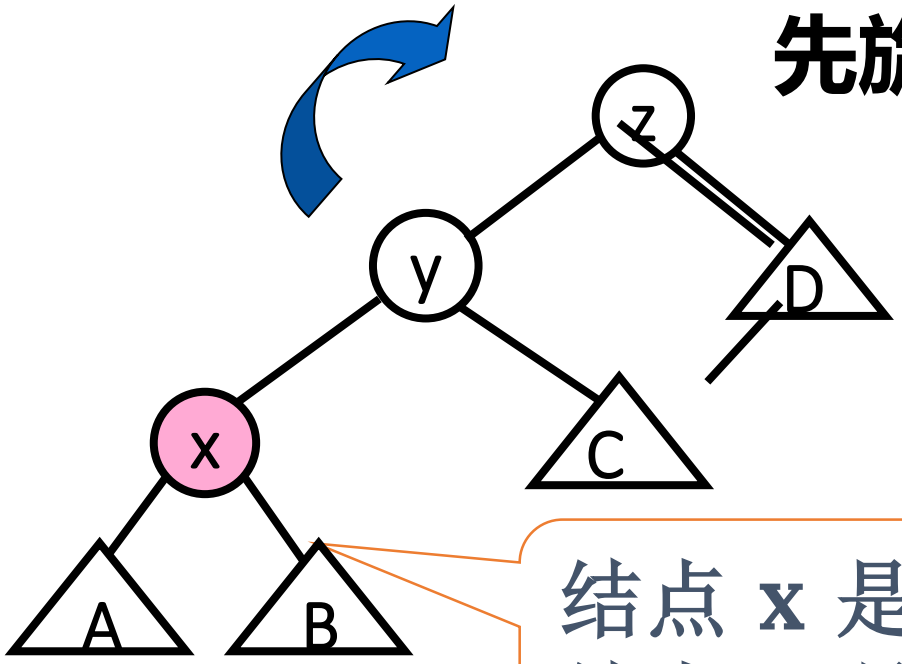


双旋转

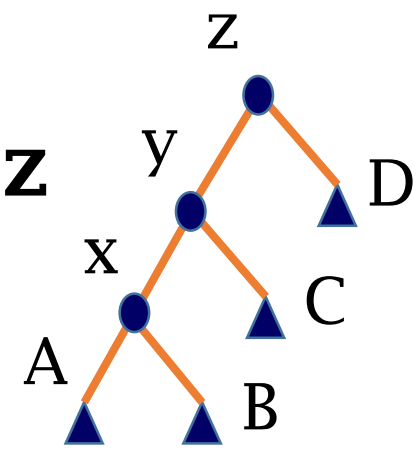


- 双旋转 (double rotation) 涉及
  - 结点 x
  - 结点 x 的父结点 (称为 y)
  - 结点 x 的祖父结点 (称为 z)
- 把结点 x 在树结构中向上移两层
- 一字形旋转 (zigzig, zagzag rotation)
  - 也称为同构调整 (homogeneous configuration)
- 之字形旋转 (zigzag, zagzig rotation)
  - 也称为异构调整 (heterogeneous configuration)

一字形旋转图示

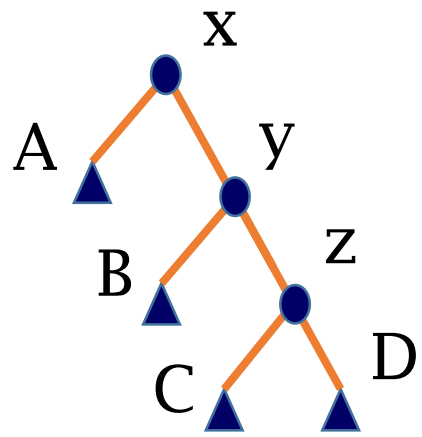


先旋转y和z



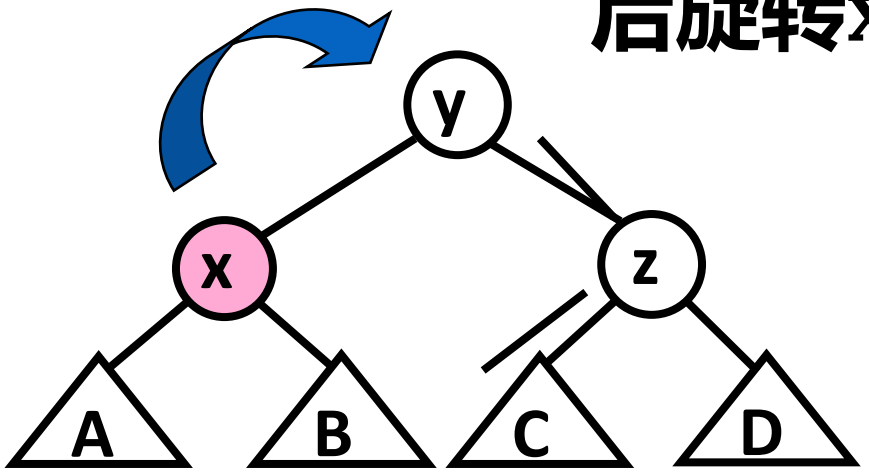
右旋 (zig)

左旋 (zag)

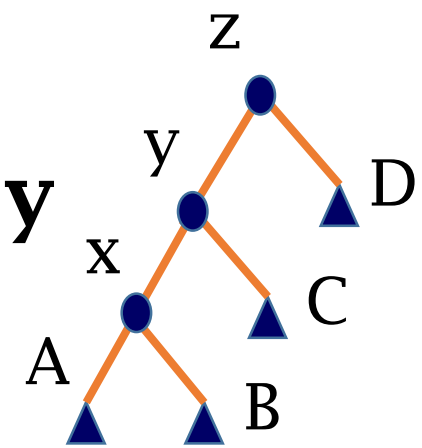


结点 x 是 y 的左子结点
结点 y 是 z 的左子结点

一字形旋转图示

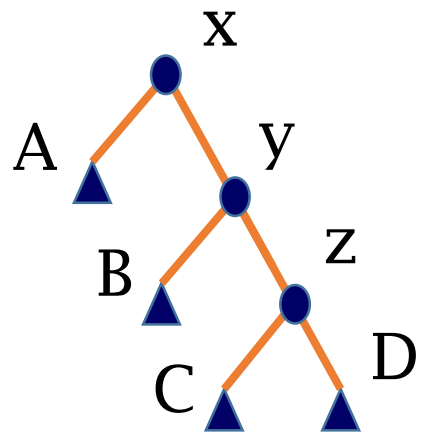


后旋转x和y

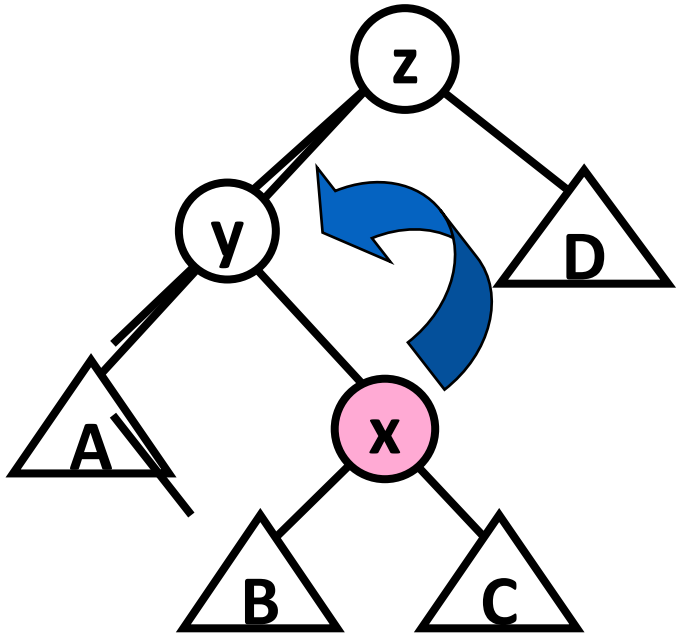


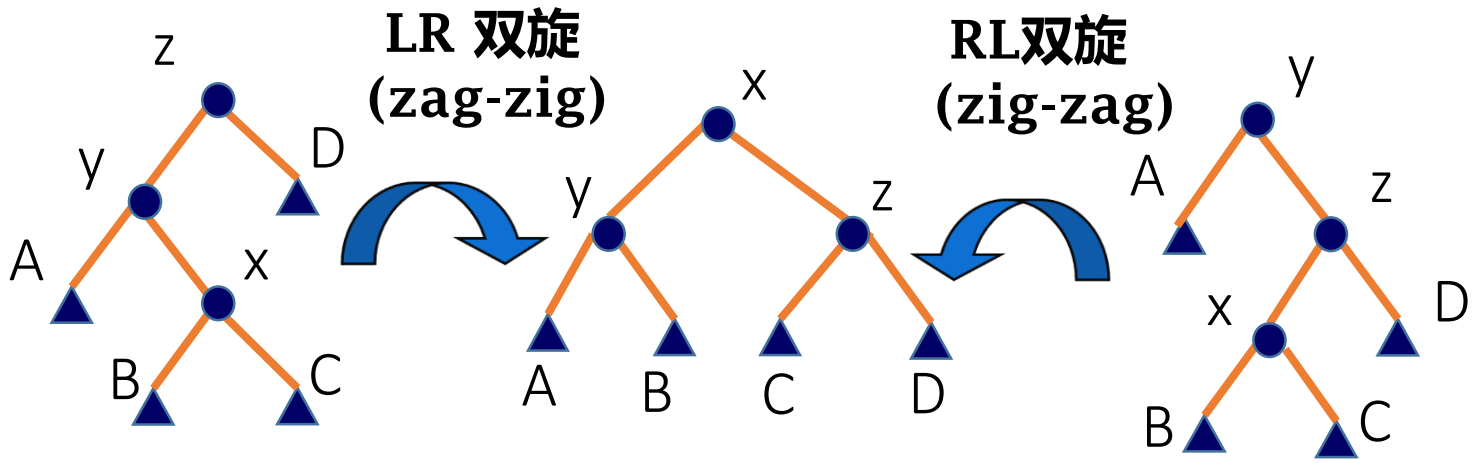
右旋 (zig)

左旋 (zag)



之字形旋转图示

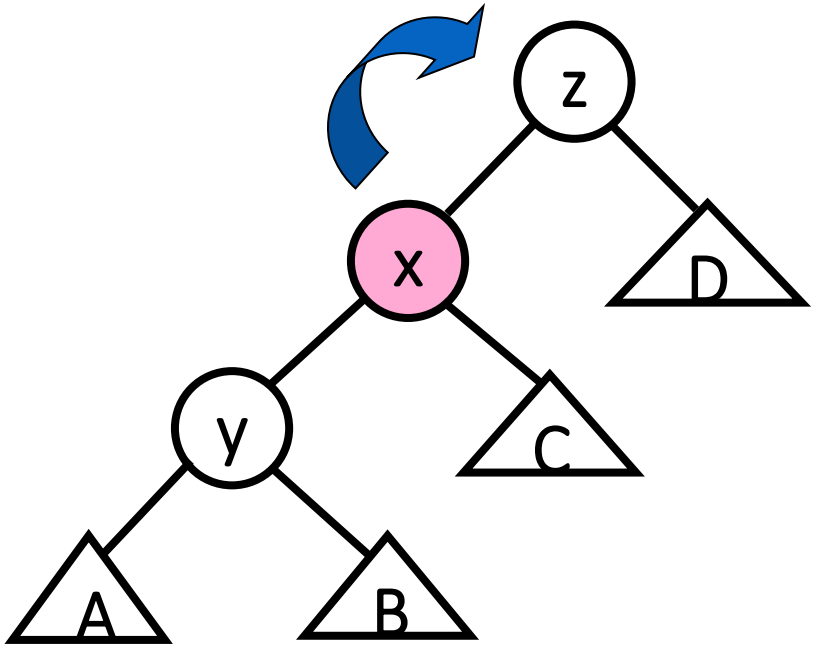


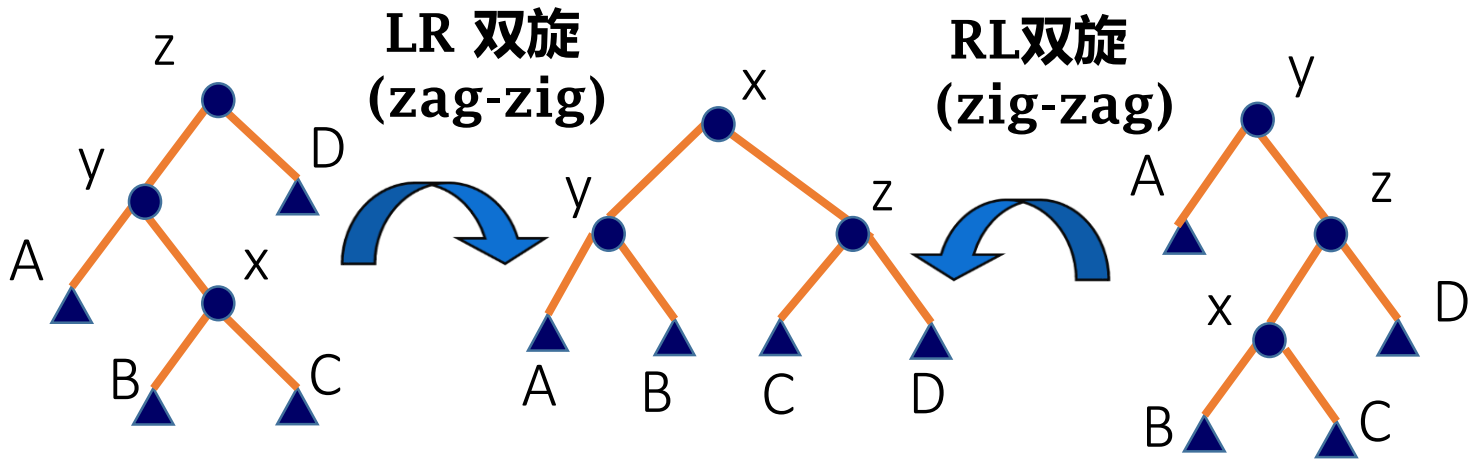


结点 x 是 y 的右子结点
结点 y 是 z 的左子结点

12.7 伸展树

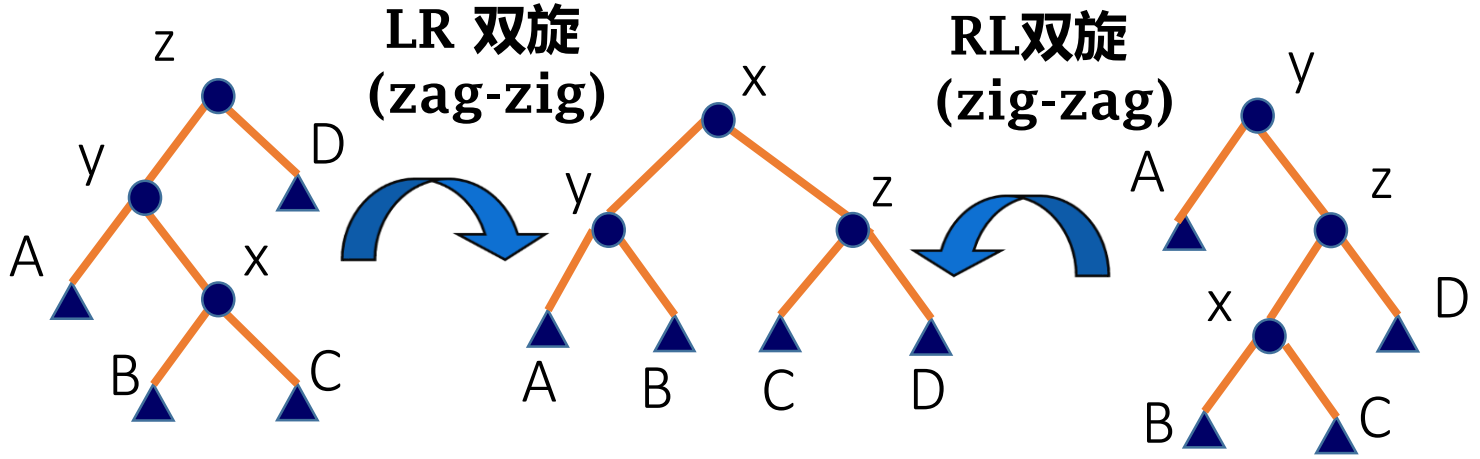
之字形旋转图示

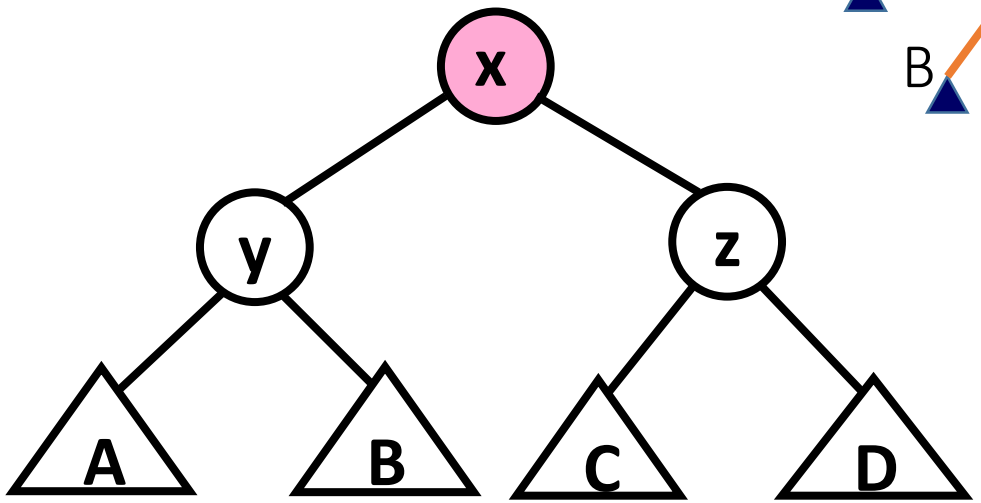




12.7 伸展树

之字形旋转图示

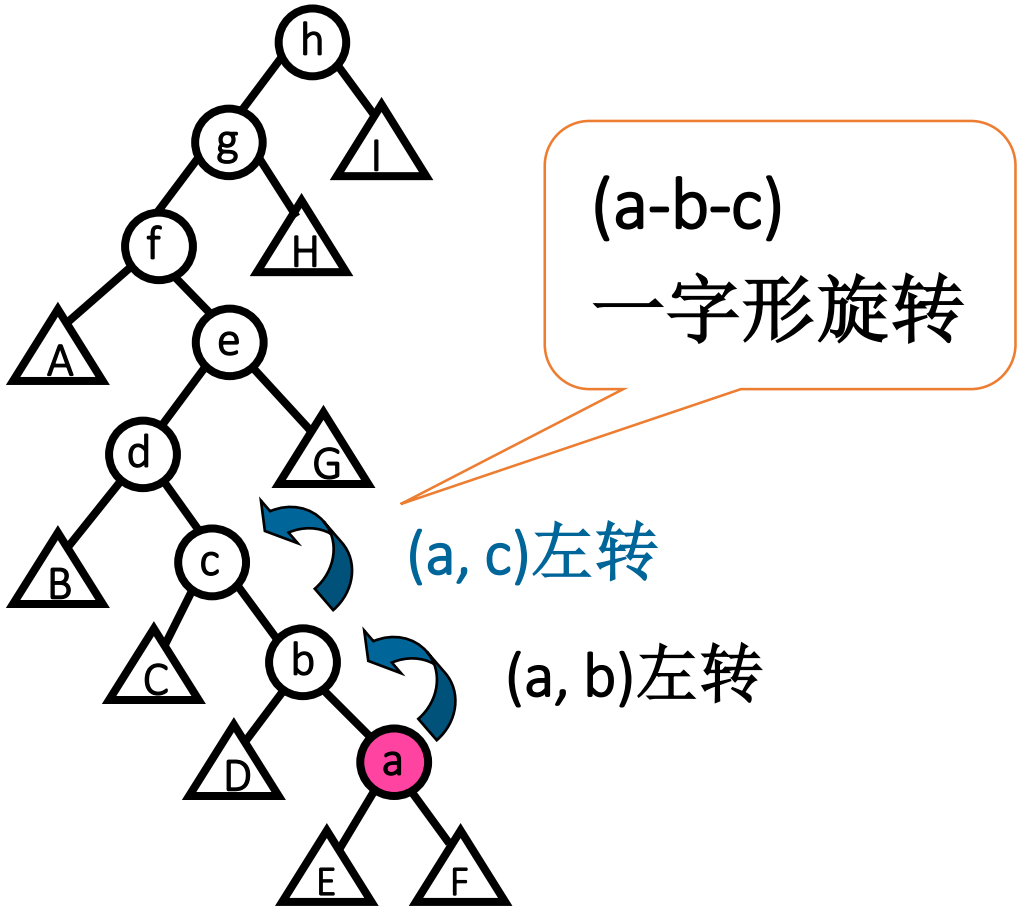




伸展树的调整过程

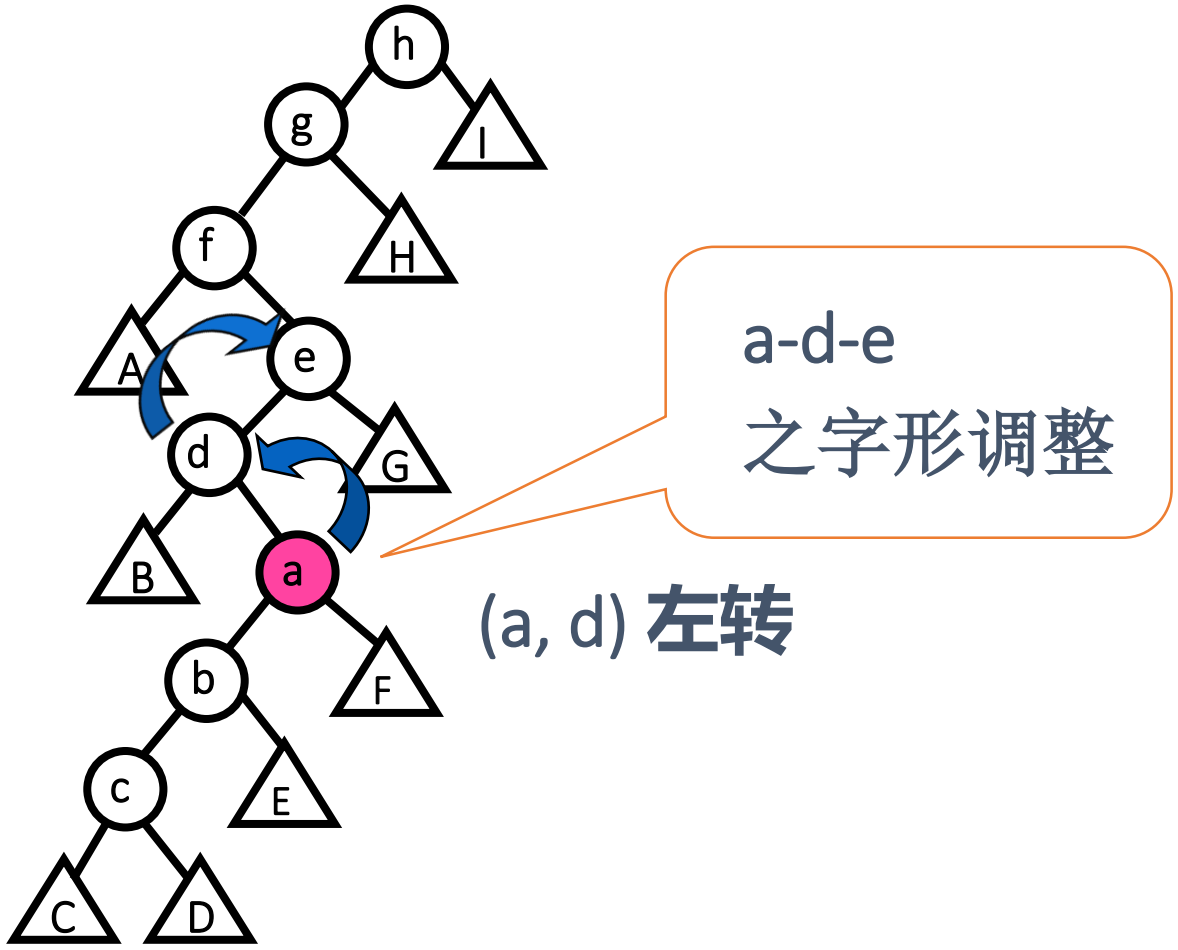
- 每次展开(splaying)操作包含
  - 一系列双旋转
    - 直到结点 x 到达根结点或者根结点的子结点
  - 如果结点x到达根结点的子结点
    - 进行一次单旋转使结点 x 成为根结点
- 这个过程趋向于使树结构重新平衡
  - 使访问最频繁的结点靠近树结构的根层
  - 从而减少访问代价

伸展树的调整过程



伸展树的调整过程

(a, e) **右转**

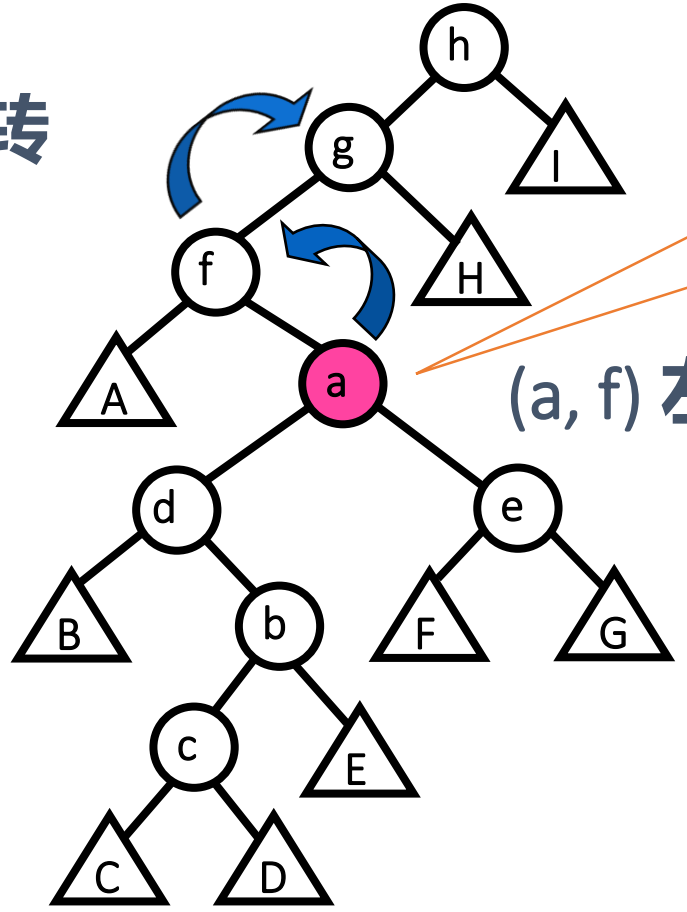


a-d-e
之字形调整

(a, d) **左转**

伸展树的调整过程

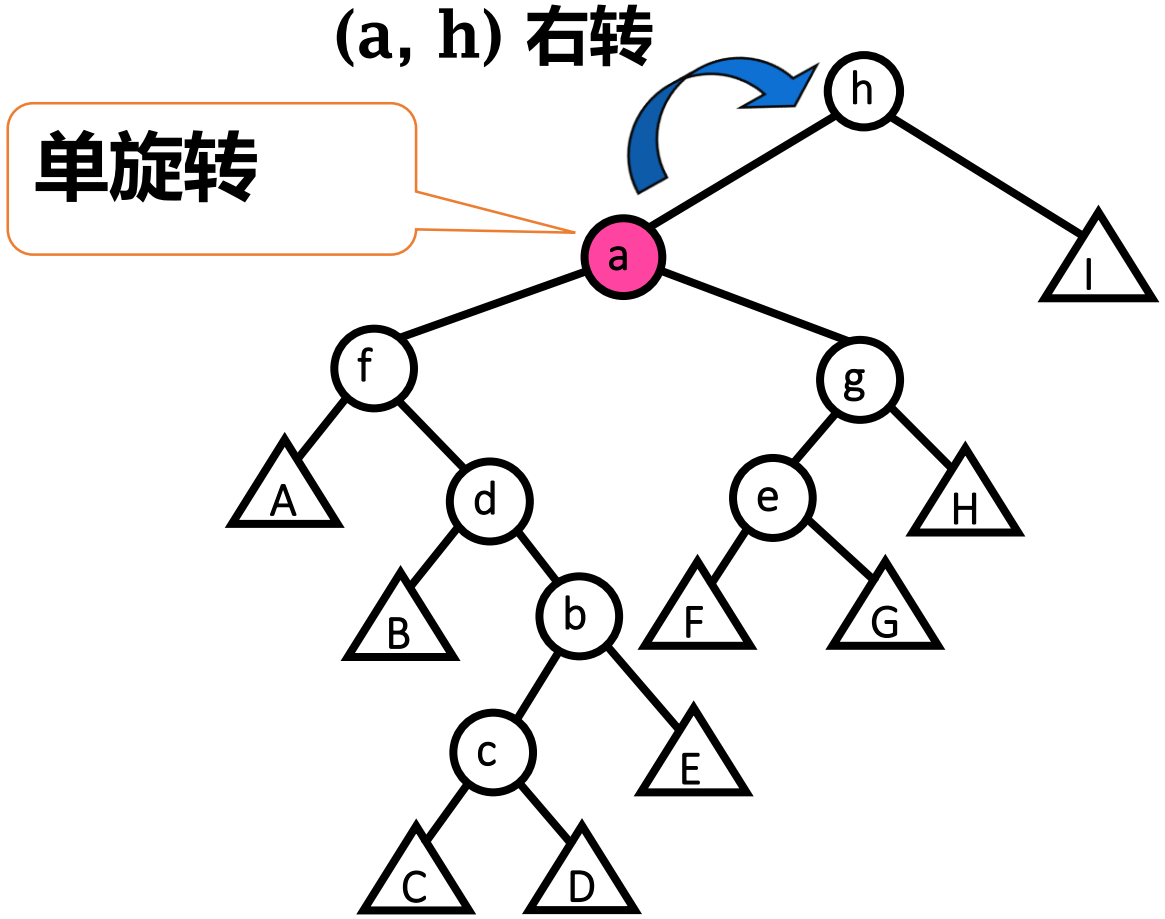
(a, g) 右转



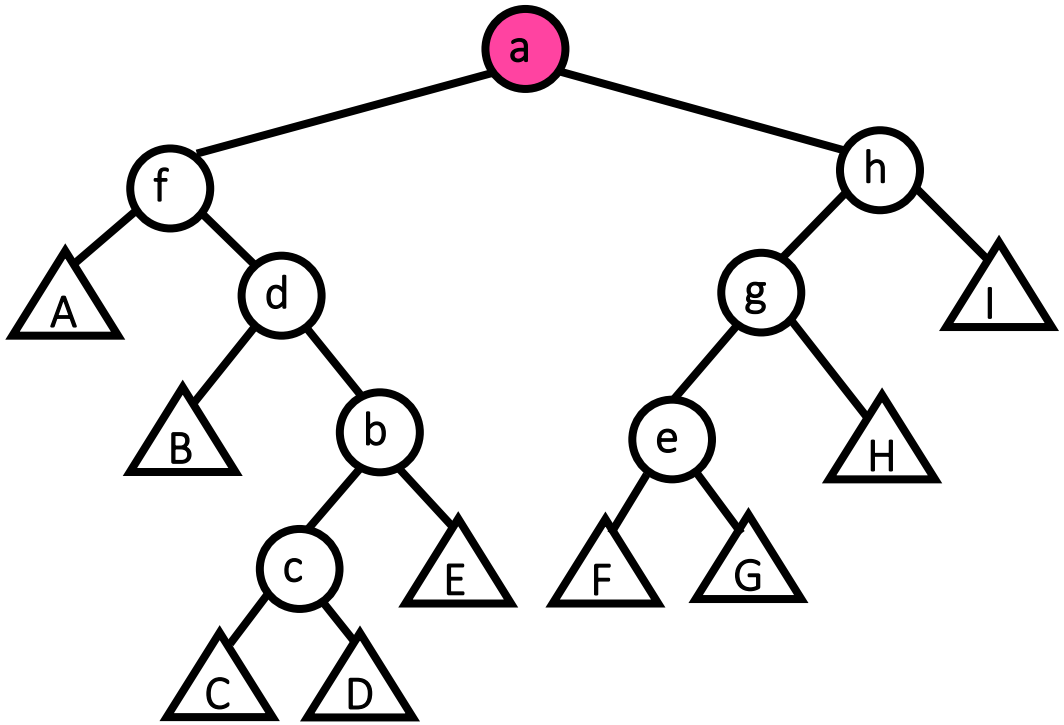
(a, f) 左转

(a-f-g)
之字形旋转

伸展树的调整过程

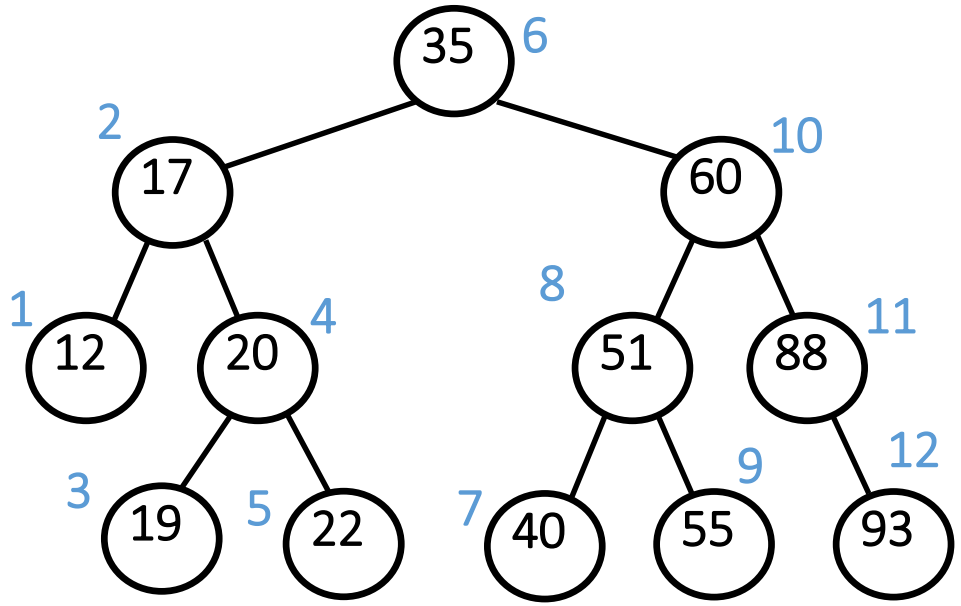


伸展树的调整过程

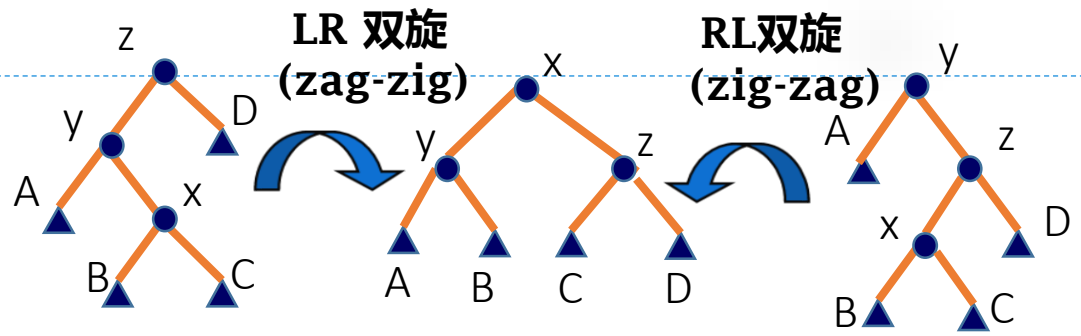


Splay 树的实现

```
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
```



12.7 伸展树

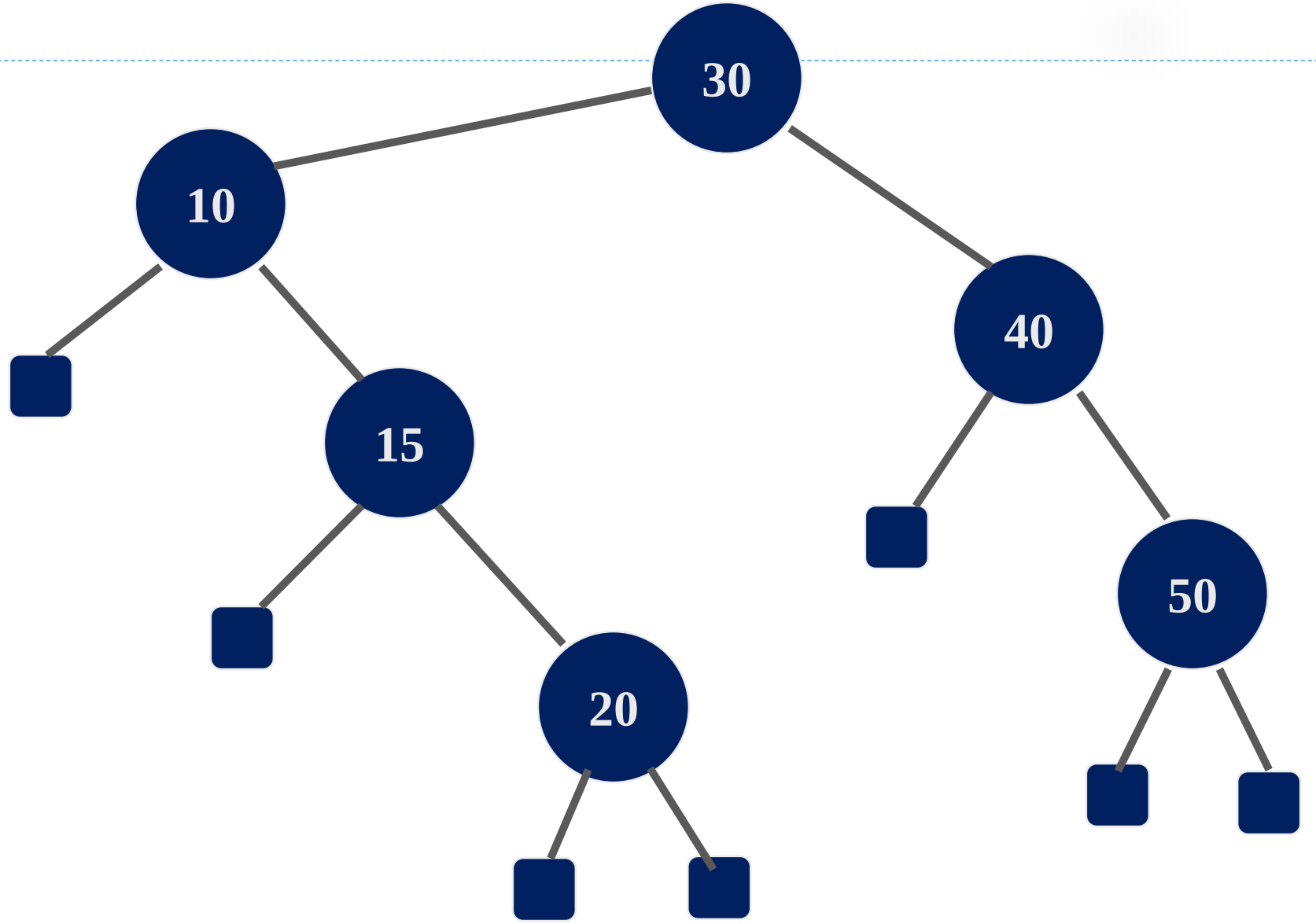



```
void Splay (TreeNode *x, TreeNode *f) {
    while (x->parent != f) {
        TreeNode *y = x-> parent, *z = y-> parent;
        if (y->parent != f)  {                  // y 不是 f 的子结点
            if (z->lchild == y) {
                if (y->lchild == x)
                { Zig(y);  Zig(x); }            // 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;
}
```

Splay 的删除

- **方法一：**
  - 值替代
  - 再从**被删的父调整**

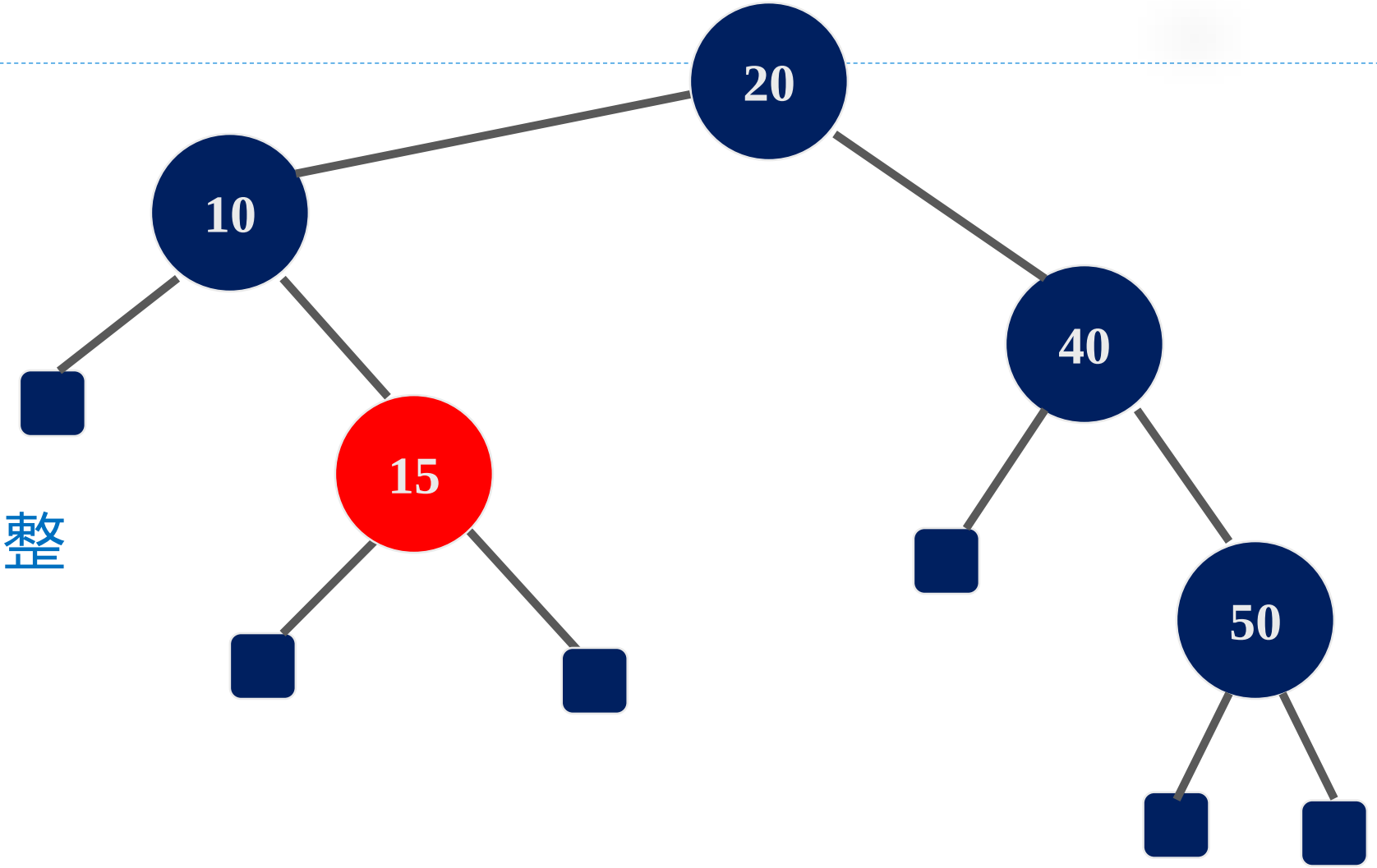
要删除 30
先用左最大20替代



Splay 的删除

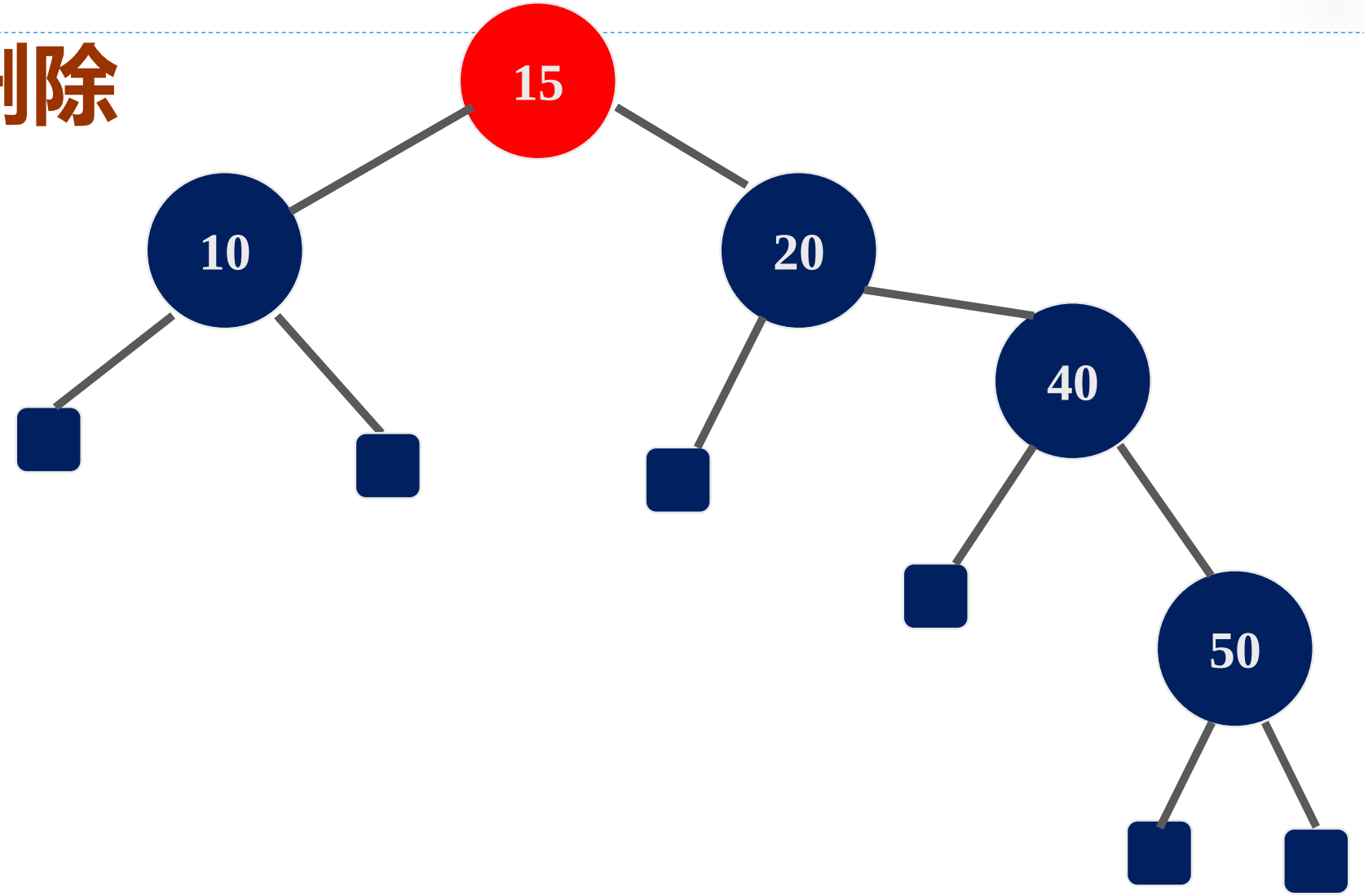
- 方法一：
  - 值替代（30->20）
  - 再从被删的父调整

被删的父15，开始调整



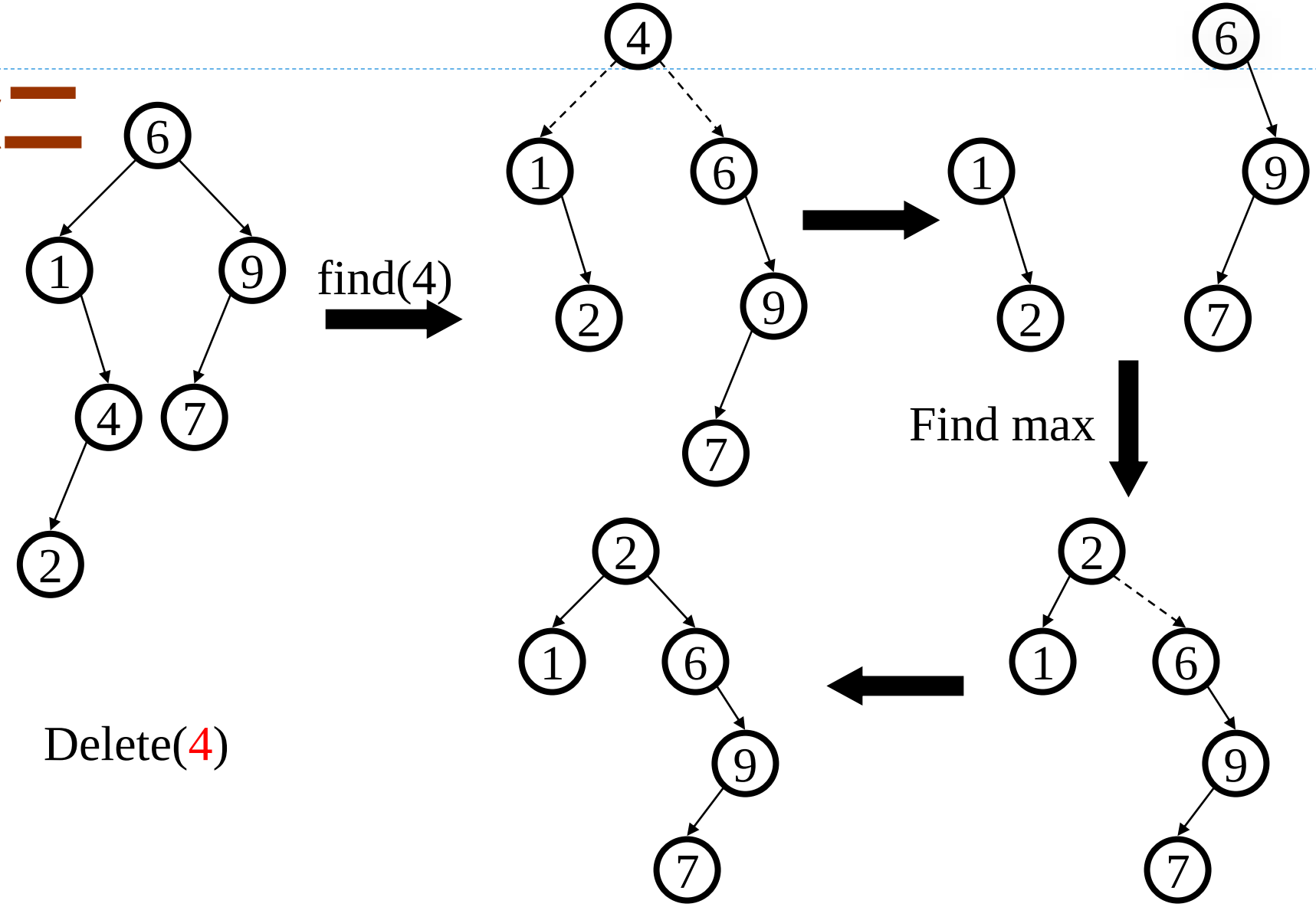
Splay 的删除

15被调到根



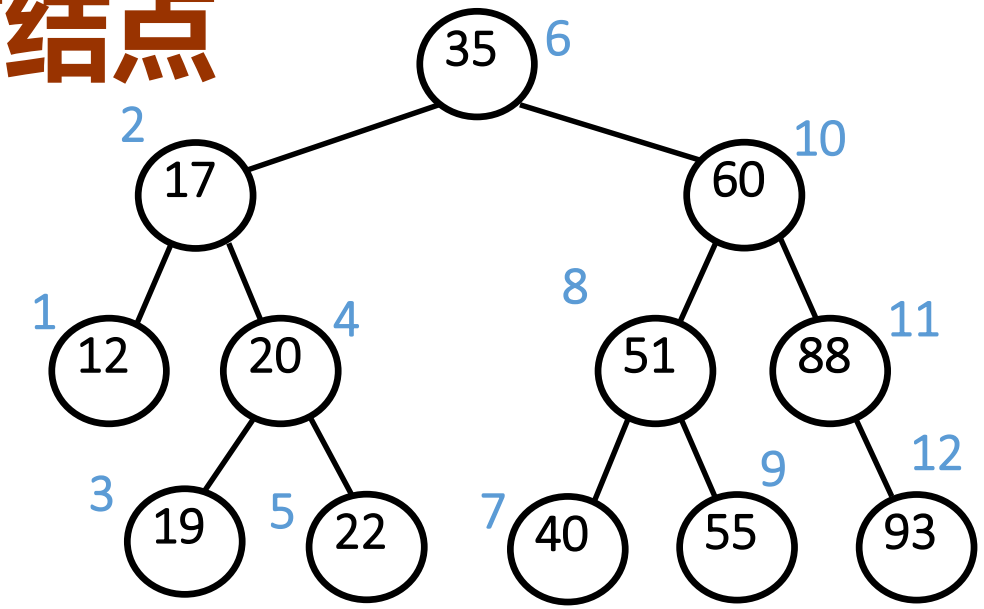
Splay 删除二

- 先旋转到根
- 删除根
- 左最大调到根
- 合并



删除大于 u 小于 v 的所有结点

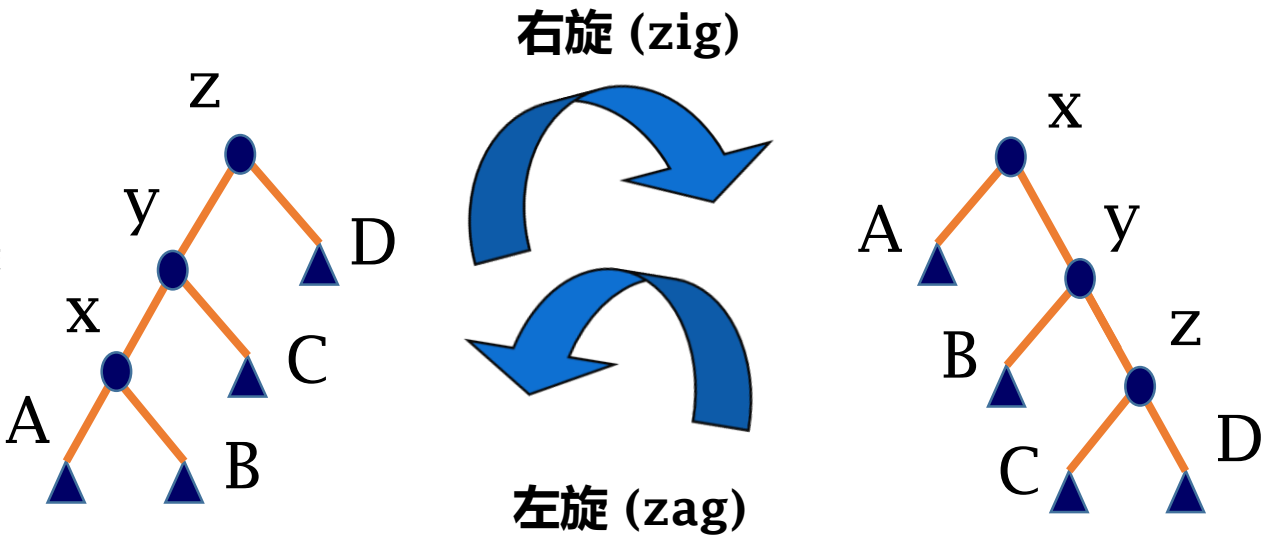
- 把 u 结点旋转到根
- 把 v 旋转为 u 的右儿子
- 删除 v 结点的左子树




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

双旋的意义

- 假设结点x，x的父亲y，y的父亲z
- 比较AVL与伸展树的之字形双旋：一样
  - AVL树中，RL与LR情形采用与伸展树同样的之字形双选
  - 旋转顺序：先交换x与y，再交换x与z
- 比较AVL与伸展树的一字型双旋：不一样
  - AVL树中LL与RR顺序：先交换x与y，再交换x与z
  - 伸展树中一字型旋转：先交换y与z，再交换x与y



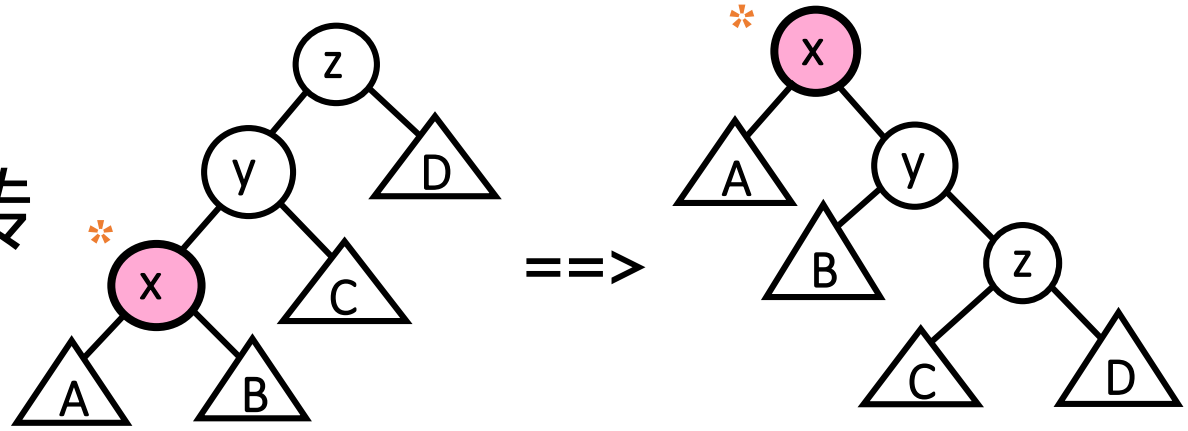
- 思考：
  - 如果伸展树中不使用一字型旋转，改用与AVL一样的旋转顺序，是否可行？
  - 如果放弃双旋，统一使用单旋，是否可行？

伸展树的效率

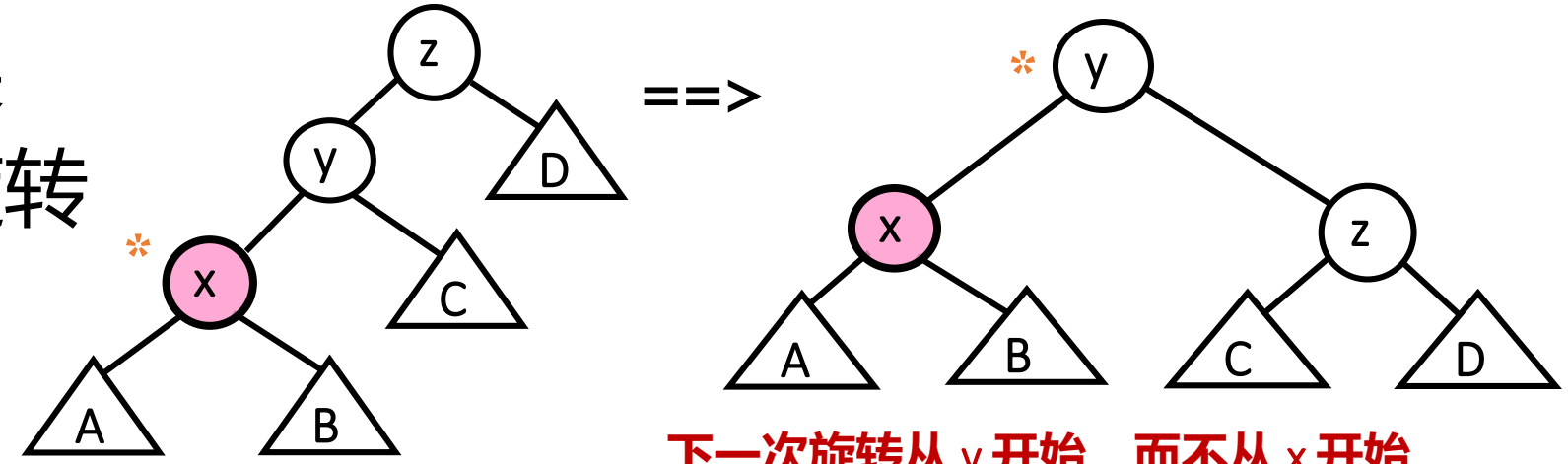
- n 个结点的伸展树
- 进行一组 m 次操作（插入、删除、查找操作），当 m>>n 时，总代价是 O(m logn)
  - 伸展树不能保证每一个单个操作是有效率的
  - 即每次访问操作的平均代价为 O(log n)

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

半伸展分析

- 半伸展树的时间复杂度在渐近意义上和伸展树是一样的。
- 但是半伸展树的执行效率和查询序列密切相关
  - 当序列比较规则的时候，半伸展树可能表现得更好
  - 但是当序列变得不规则的时候，它的调整结构速度不如伸展树快
- 同时，半伸展树的编程复杂度变高了，为了支持删除、合并等操作，需要相对复杂的处理

思考

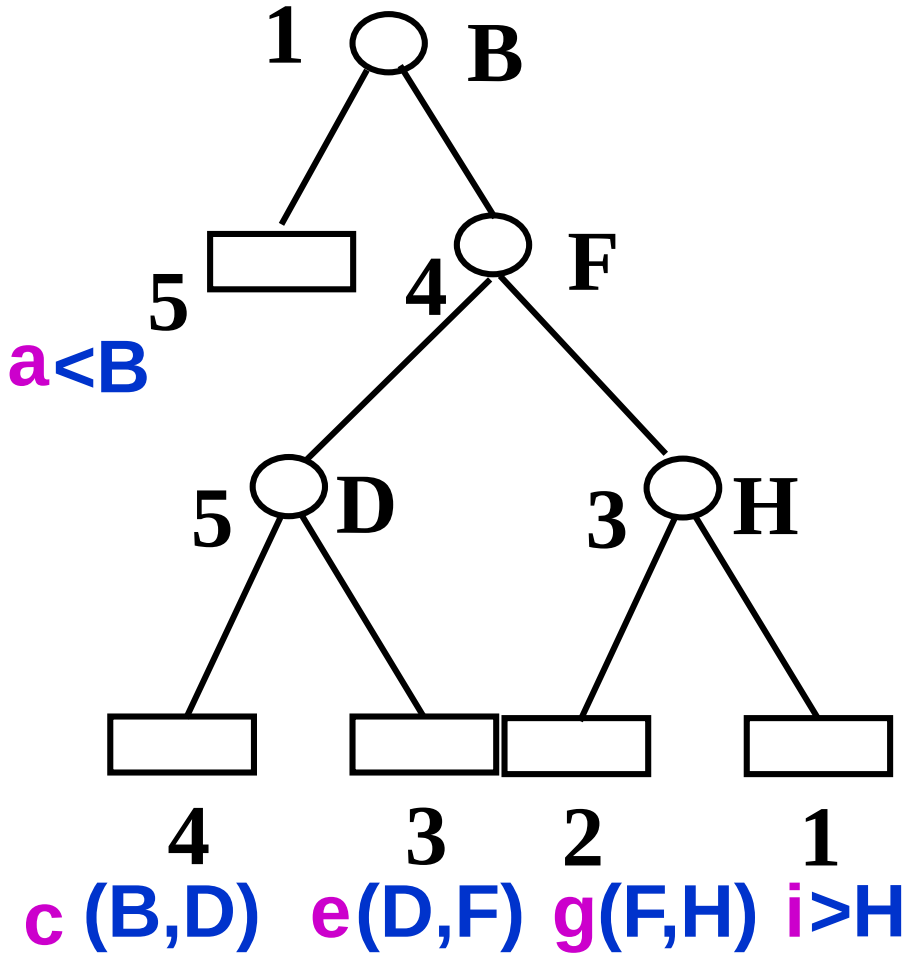
- 请调研 Splay 树的各种应用

- 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?

最佳 BST

- 包含关键码 key_{i+1}，key_{i+2}，…，key_j为内部结点($0 \le i \le j \le n$)
- 结点的权为(q_i，p_{i+1}，q_{i+1}，…，p_j，q_j)，
- 根为r(i，j)
- 开销为C(i，j)，即
- 总权为W(i，j) = $p_{i+1}+\ldots+p_j+q_i+q_{i+1}+\ldots+q_j$

$$ASL(n) = \frac{1}{W}\left[\sum_{i=1}^{n} p_i(1_i+1) + \sum_{i=0}^{n} q_1 l_i'\right]$$



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)

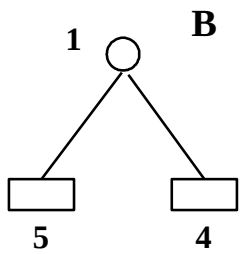
i \ j	0	1	2	3	4
0	0	10	28	43	57
1		0	12	27	40
2			0	9	19
3				0	6
4					0

C(i ,j)

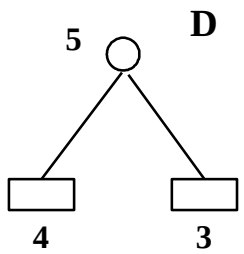
i \ j	0	1	2	3	4
0	5	10	18	21	28
1		4	12	18	22
2			3	9	3
3				3	6
4					1

W(i ,j)

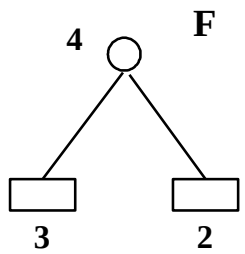
第一步



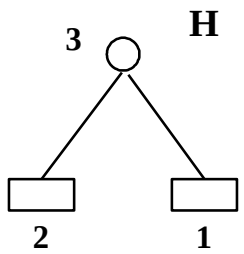
B



D



F

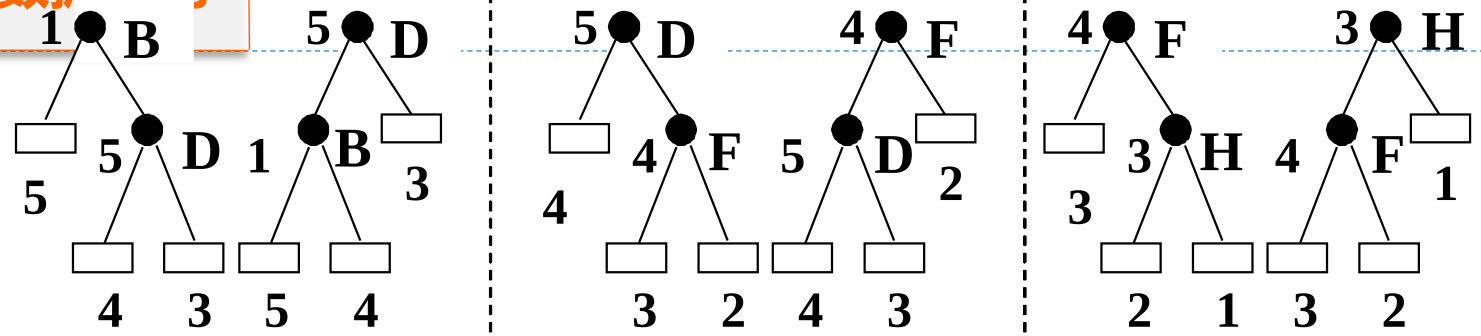


H

花费 10 12 9 6

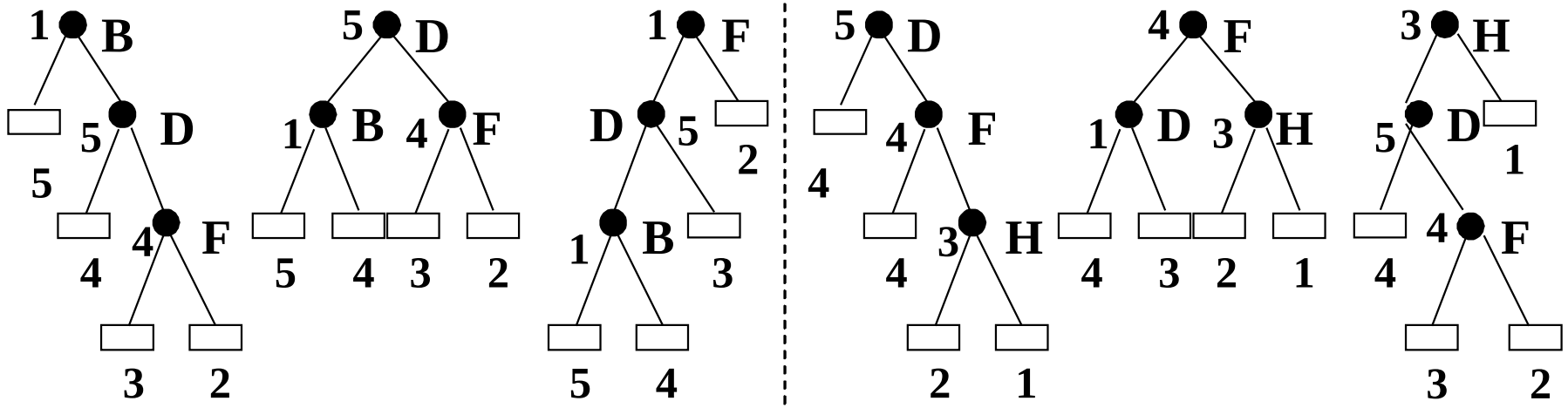
总权 10 12 9 6

第二步



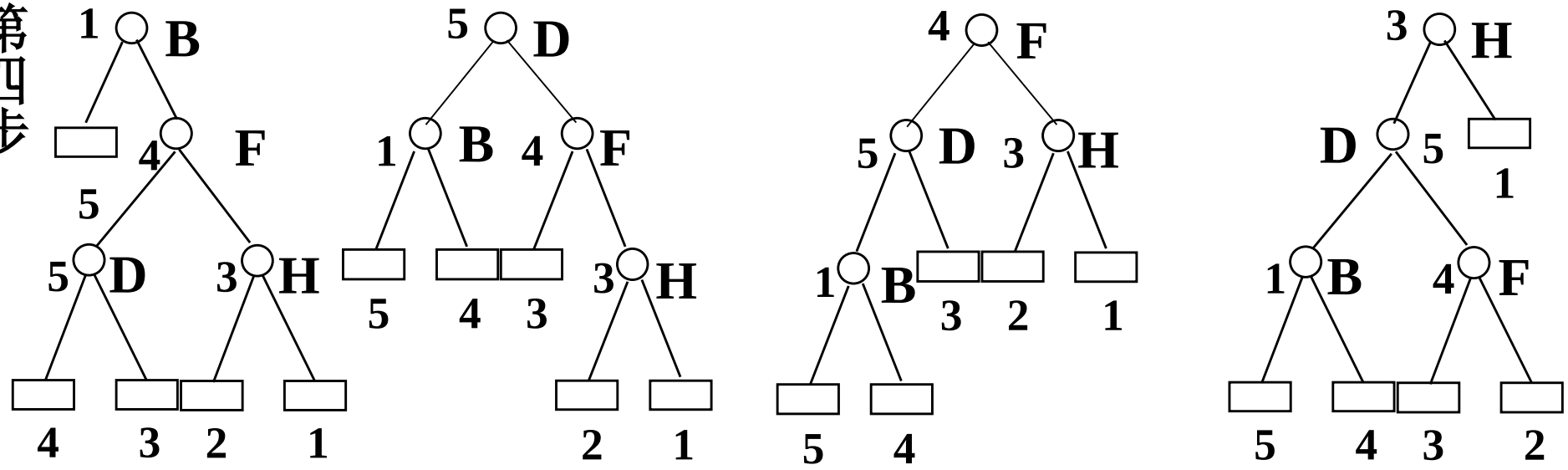
开销	30		28	27		30	
总权	18		18	18		18	
				19		22	
				13		13	

第三步



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

第四步



花费 68　57　62　71

总权 28　28　28　28

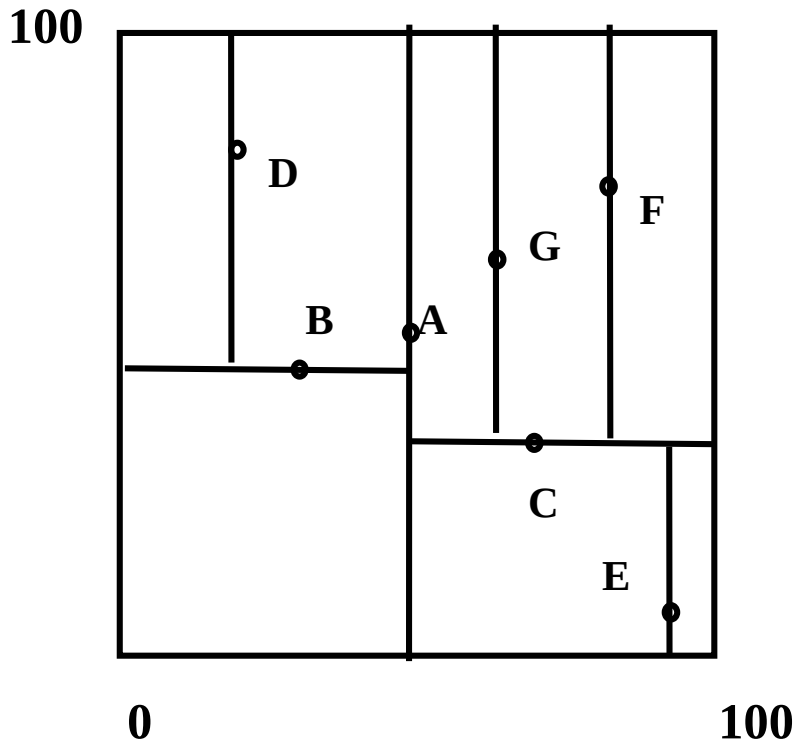
最佳二叉搜索树的动态规划

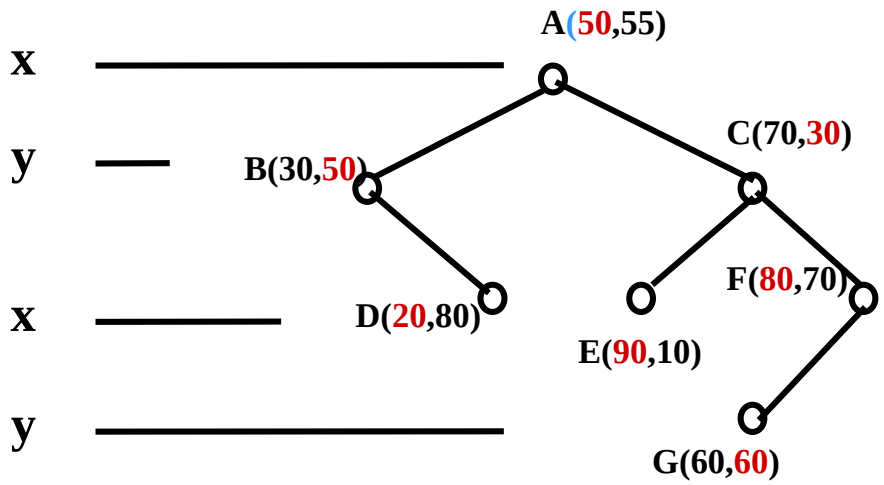
- 最佳子结构、重复子结构
  - 任何子树都是最佳二叉搜索树
- 动态规划过程
  - 第一步：构造包含1个结点的最佳二叉搜索树
    - 找t(0，1)，t(1，2)，…，t(n-1，n)
  - 第二步构造包含2个结点的最佳二叉搜索树
    - 找t(0，2)，t(1，3)，…，t(n-2，n)
  - 再构造包含3，4，…个结点的最佳二叉搜索树
  - 最后构造t(0，n)

K-D树示例

A(50,55) B(30,55) C(70,30) D(20,80)

E(90,10) F(80,70) G(60,60)



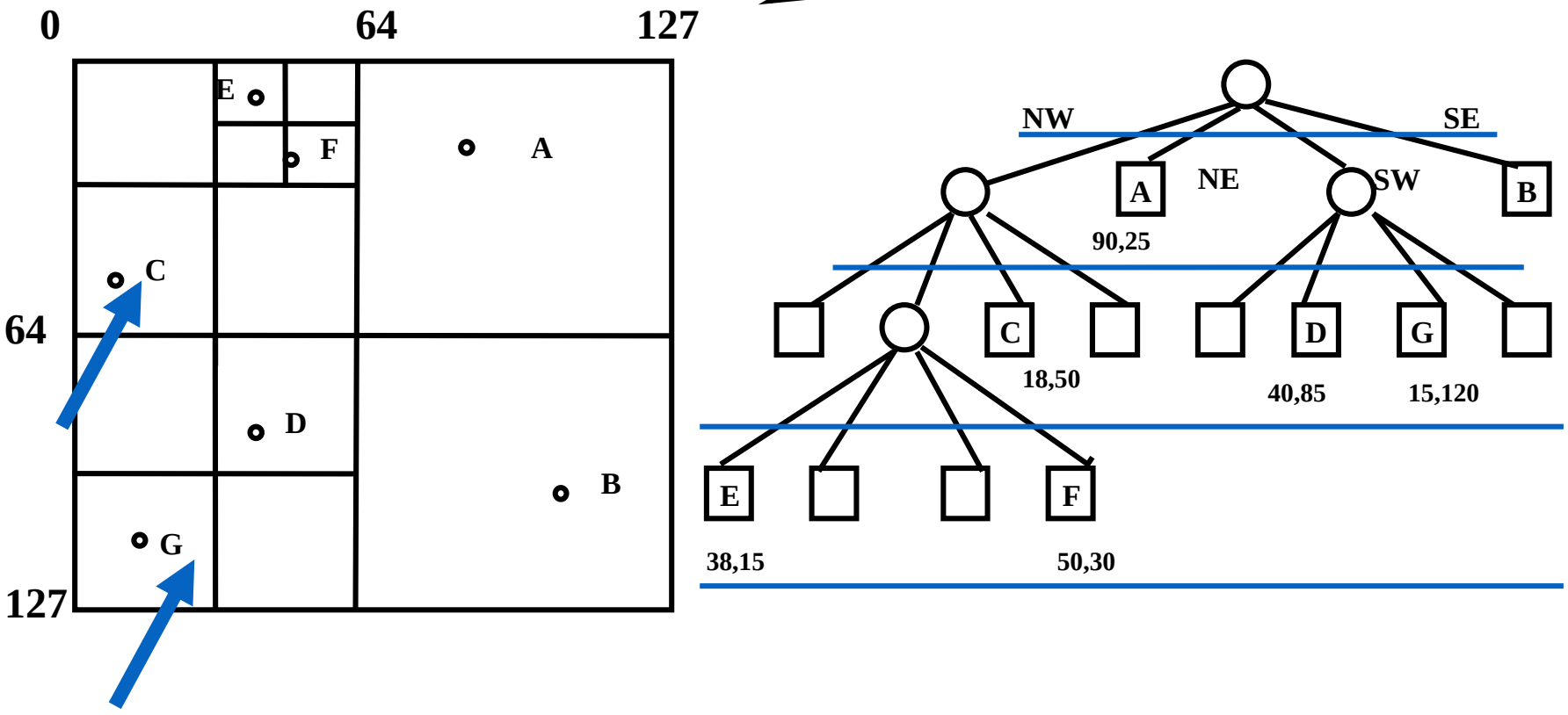


PR树的图示与划分

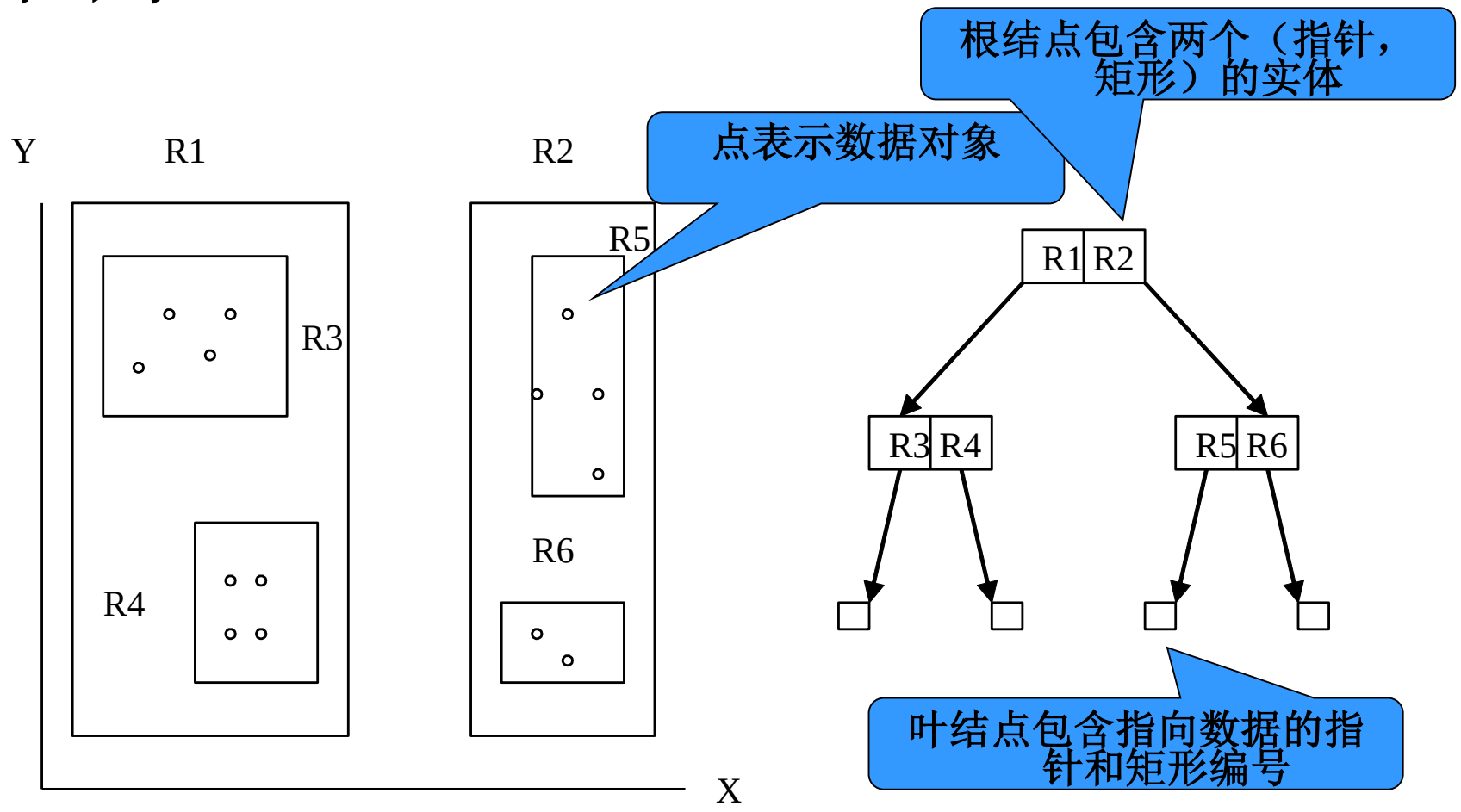
对象空间为128×128， 点A、B、C、D、E、F和G

顶层空间平分为4份：NW,NE,SW,SE

NW,SE (包含多个数据点)进一步分裂



R树的图示



左边图的矩形和右边的编号对应；右边是构建好的一棵**R**树

数据结构与算法

谢谢倾听

国家精品课“数据结构与算法”

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

https://www.icourse163.org/course/PKU-1002534001

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6。“十二五”国家级规划教材