



数据结构与算法 (十二)

张铭 主讲

采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008.6（“十一五”国家级规划教材）

<http://www.jpku.pku.edu.cn/pkujpku/course/sjjg>



第十二章 高级数据结构

- 12.1 多维数组
 - 12.1.1 基本概念
 - 12.1.2 数组的空间结构
 - 12.1.3 数组的存储
 - 12.1.4 数组的声明
 - 12.1.5 用数组表示特殊矩阵
 - 12.1.6 稀疏矩阵
- 12.2 广义表
- 12.3 存储管理
- 12.4 Trie 树
- 12.5 改进的二叉搜索树

12.1 多维数组

基本概念

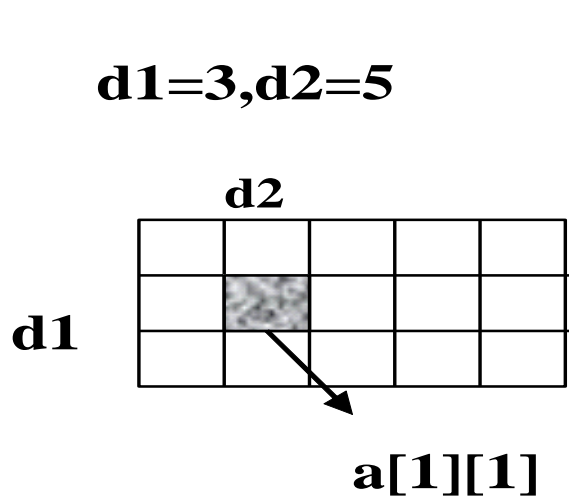
- 数组 (Array) 是数量和元素类型固定的有序序列
- 静态数组必须在定义它的时候指定其大小和类型
- 动态数组可以在程序运行才分配内存空间

基本概念 (续)

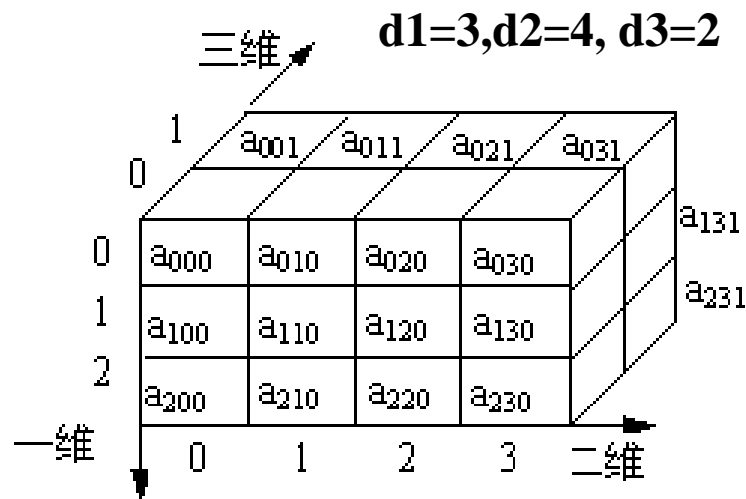
- 多维数组 (Multi-array) 是向量的扩充
- 向量的向量就组成了多维数组
- 可以表示为：
ELEM A[c₁..d₁][c₂..d₂]...[c_n..d_n]
- c_i 和 d_i 是各维下标的下界和上界。所以其元素个数为：

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

数组的空间结构



二维数组



三维数组

$d1[0..2], d2[0..3], d3[0..1]$ 分别为3个维

12.1 多维数组

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”)

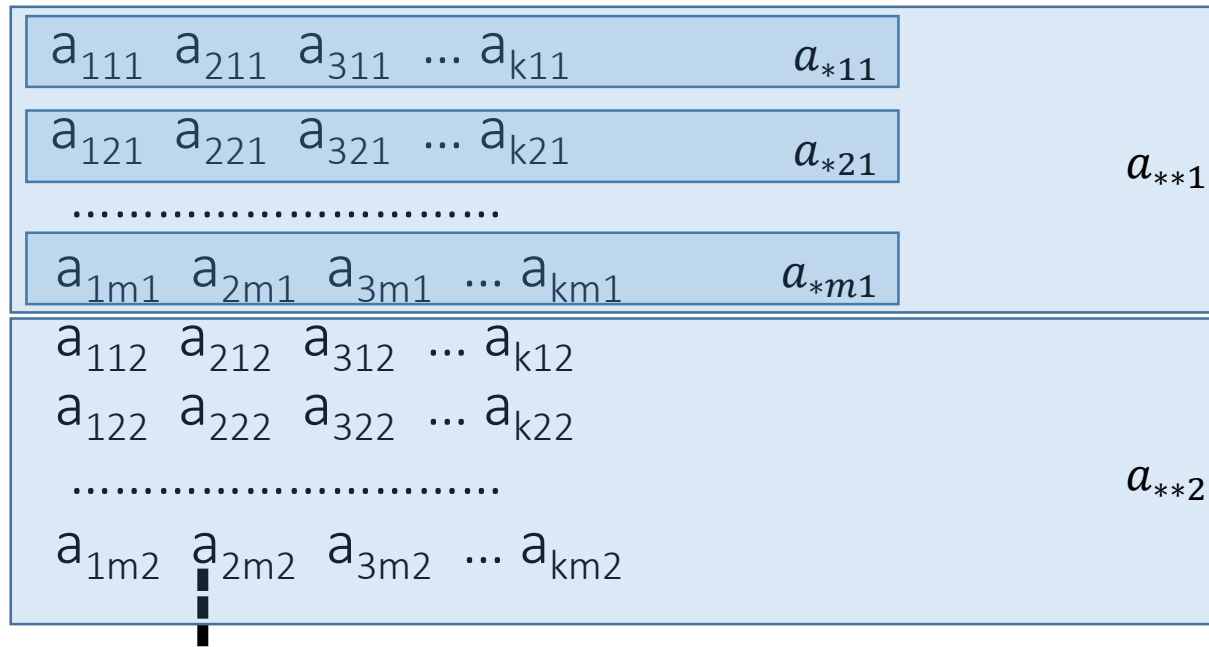
X=	1	2	3
	4	5	6
	7	8	9

12.1 多维数组

Pascal语言的行优先存储 $a[1..k, 1..m, 1..n]$

a_{111}	a_{112}	a_{113}	\dots	a_{11n}	a_{11*}
a_{121}	a_{122}	a_{123}	\dots	a_{12n}	a_{12*}
\dots					
a_{1m1}	a_{1m2}	a_{1m3}	\dots	a_{1mn}	a_{1m*}
a_{211}	a_{212}	a_{213}	\dots	a_{21n}	a_{21*}
a_{221}	a_{222}	a_{223}	\dots	a_{22n}	a_{22*}
\dots					
a_{2m1}	a_{2m2}	a_{2m3}	\dots	a_{2mn}	a_{2m*}
\vdots					
a_{k11}	a_{k12}	a_{k13}	\dots	a_{k1n}	
a_{k21}	a_{k22}	a_{k23}	\dots	a_{k2n}	
\dots					
a_{km1}	a_{km2}	a_{km3}	\dots	a_{kmn}	

FORTRAN的列优先存储 $a[1..k, 1..m, 1..n]$



$a_{11n} \ a_{21n} \ a_{31n} \ \dots \ a_{k1n}$
 $a_{12n} \ a_{22n} \ a_{32n} \ \dots \ a_{k2n}$

 $a_{1mn} \ a_{2mn} \ a_{3mn} \ \dots \ a_{kmn}$



- C++ 多维数组ELEM A[d₁][d₂]...[d_n];

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



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

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

对称矩阵

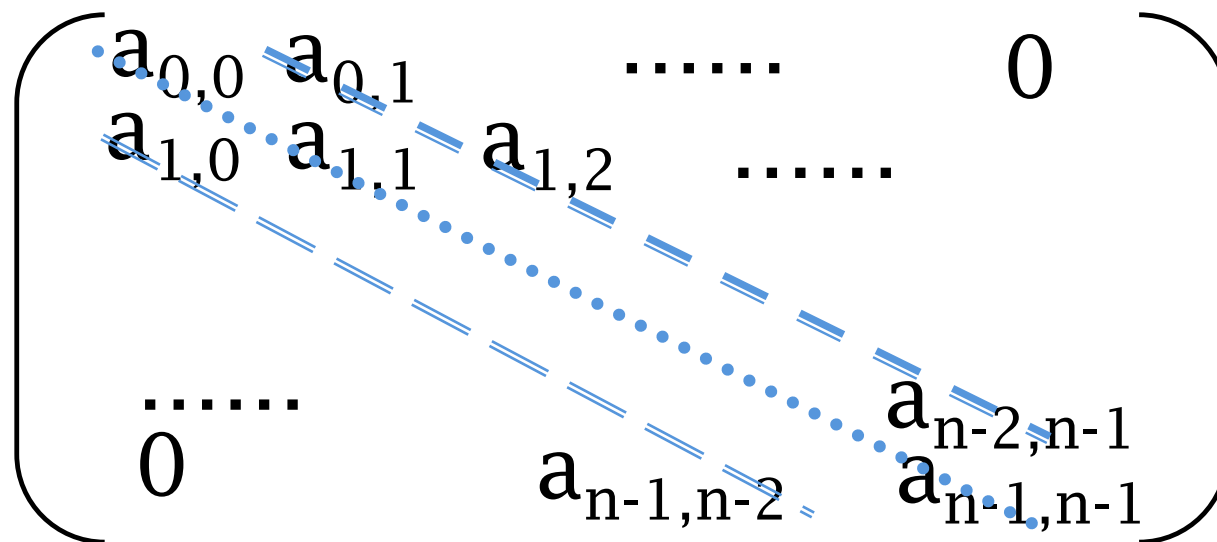
$$\begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$

- 元素满足性质 $a_{i,j} = a_{j,i}$, $0 \leq (i, j) < n$
例如, 右图的无向图相邻矩阵
- 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——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：所有的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。
- 如果 $|i-j| > 1$ ，那么数组元素 $a[i][j] = 0$ 。
 - 下面是一个三对角矩阵：


$$\begin{pmatrix} a_{0,0} & a_{0,1} & & & 0 \\ a_{1,0} & a_{1,1} & a_{1,2} & & \\ & & & \ddots & \\ 0 & & & & a_{n-2,n-1} \\ & & a_{n-1,n-2} & & a_{n-1,n-1} \end{pmatrix}$$

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

$$\mathbf{A}_{6 \times 7} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 11 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



- 稀疏因子

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

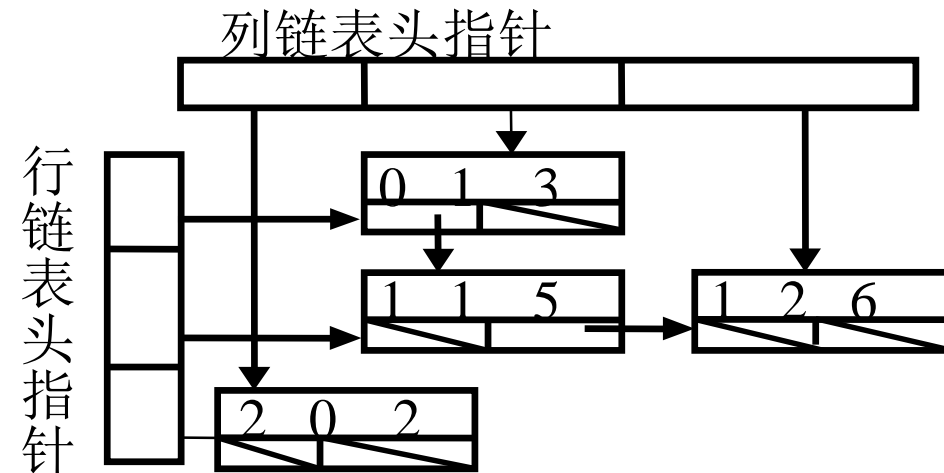
$$\delta = \frac{t}{m \times n}$$

- 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

稀疏矩阵的十字链表

- 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

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



12.1 多维数组

经典矩阵乘法

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

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

•



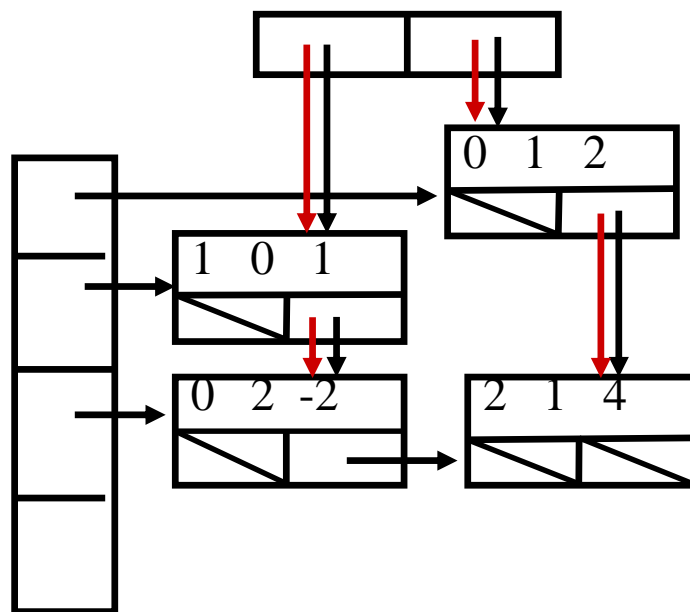
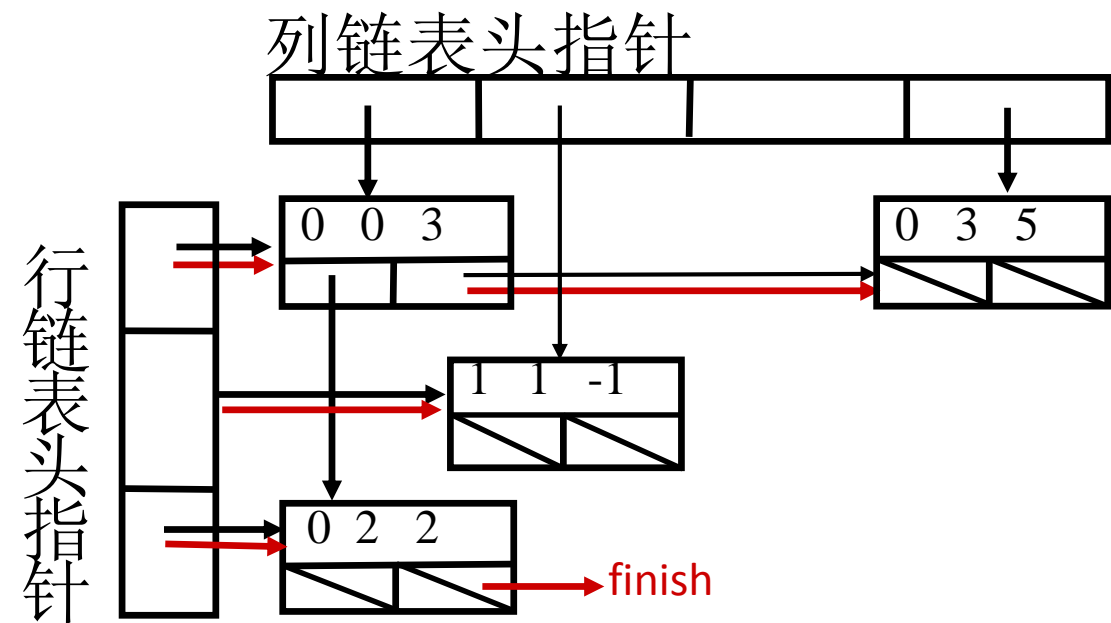
经典矩阵乘法时间代价

- $p=d_1-c_1+1$, $m=d_3-c_3+1$, $n=d_2-c_2+1$;
- A 为 $p \times m$ 的矩阵 , B 为 $m \times n$ 的矩阵 , 乘得的结果 C 为 $p \times n$ 的矩阵
- 经典矩阵乘法所需要的时间代价为 $O(p \times m \times n)$

```
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

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



稀疏矩阵乘法时间代价

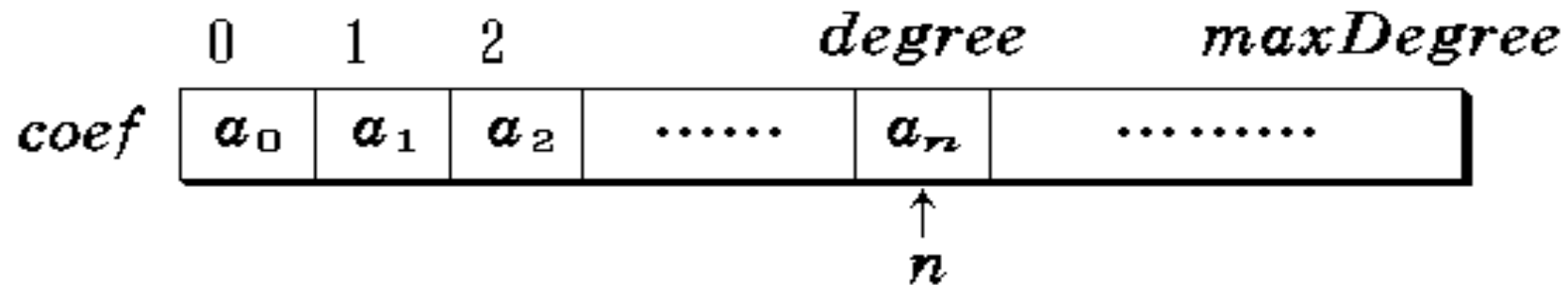
- A为 $p \times m$ 的矩阵，B 为 $m \times n$ 的矩阵，乘得的结果 C 为 $p \times n$ 的矩阵
 - 若矩阵 A 中行向量的非零元素个数最多为 t_a
 - 矩阵 B 中列向量的非零元素个数最多为 t_b
- 总执行时间降低为 $O((t_a + t_b) \times p \times n)$
- 经典矩阵乘法所需要的时间代价为 $O(p \times m \times n)$

稀疏矩阵的应用

一元多项式

$$P_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

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





第十二章 高级数据结构

- 12.1 多维数组
- 12.2 广义表
 - 基本概念
 - 广义表的各种类型
 - 广义表的存储
 - 广义表的周游算法
- 12.3 存储管理
- 12.4 Trie 树
- 12.5 改进的二叉搜索树

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

12.2 广义表和存储管理

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

- 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)
- 广义表的 **深度**：表中元素都化解为原子后的括号层数

12.2 广义表和存储管理

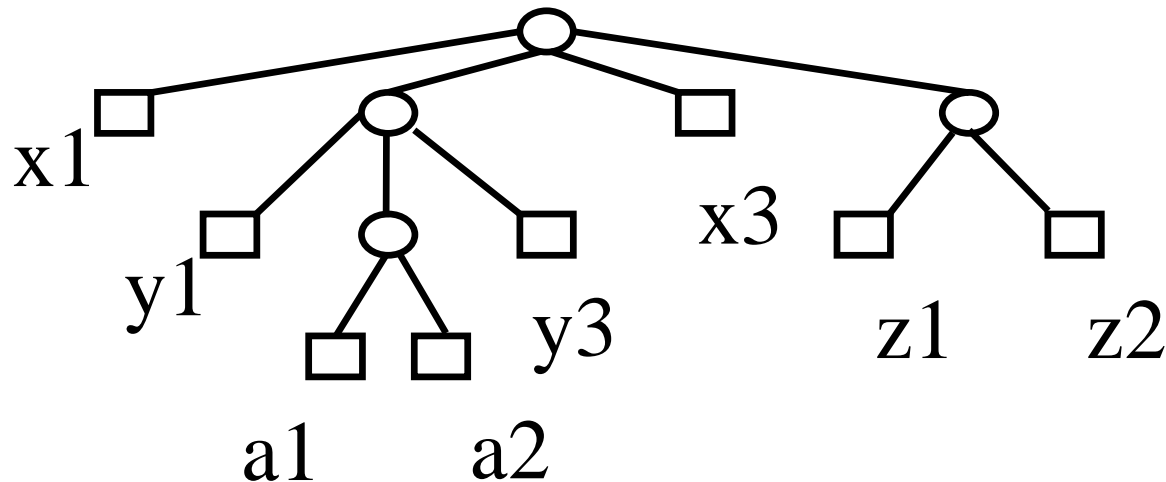
$L = (x_0, x_1, \dots, x_i, \dots, x_{n-1})$

- 表头head = x_0
- 表尾tail = (x_1, \dots, 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)
 - 从根结点到任何叶结点只有一条路径
 - 即任何一个元素 (原子、子表) 在广义表中只出现一次

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

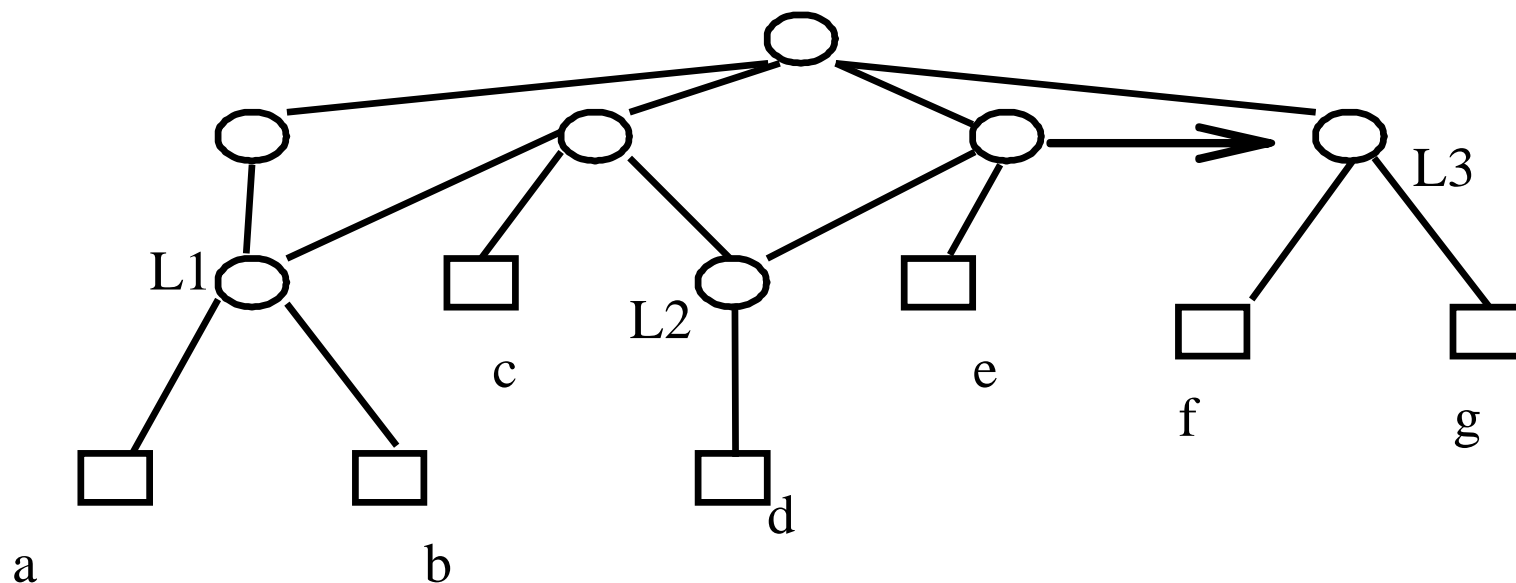


广义表的各种类型 (续)

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

特例：循环表 (即递归表)

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

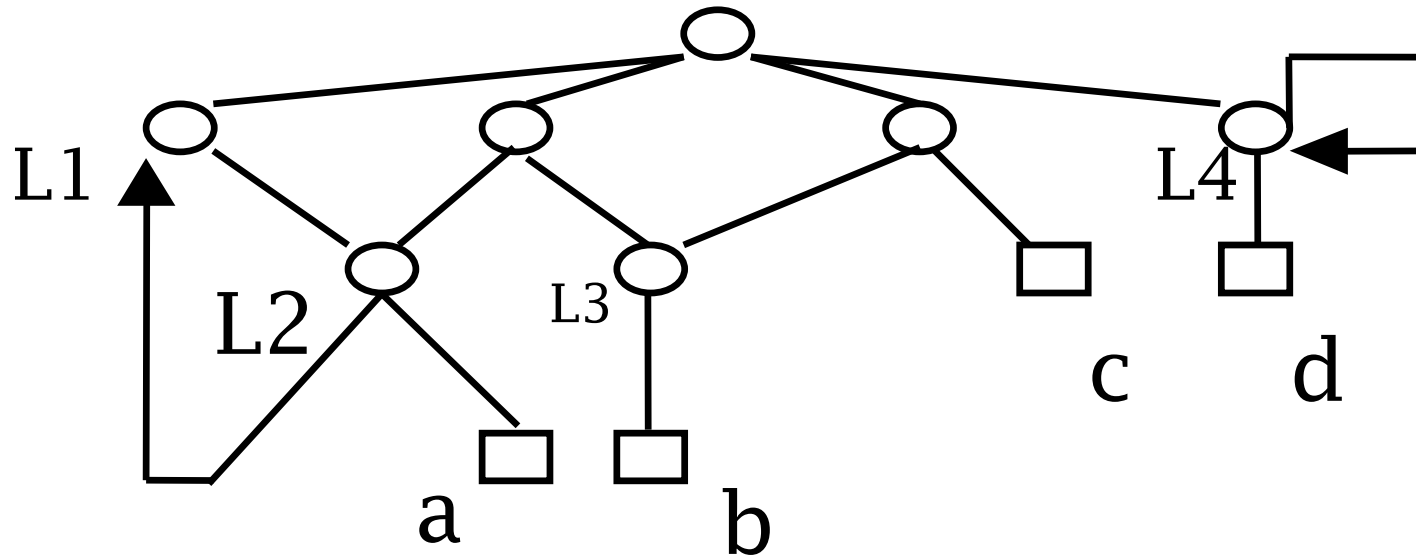


$(L1: (a,b), (L1, c, L2: (d)), (L2, e, L3: (f,g)), L3)$

广义表的各种类型 (续)

- 循环表
 - 包含回路
 - 循环表的深度为无穷大

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



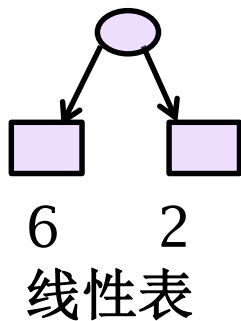


12.2 广义表和存储管理

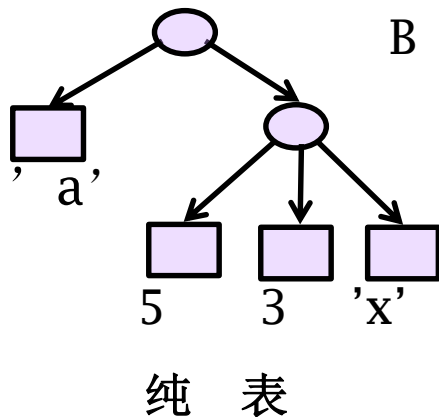
A



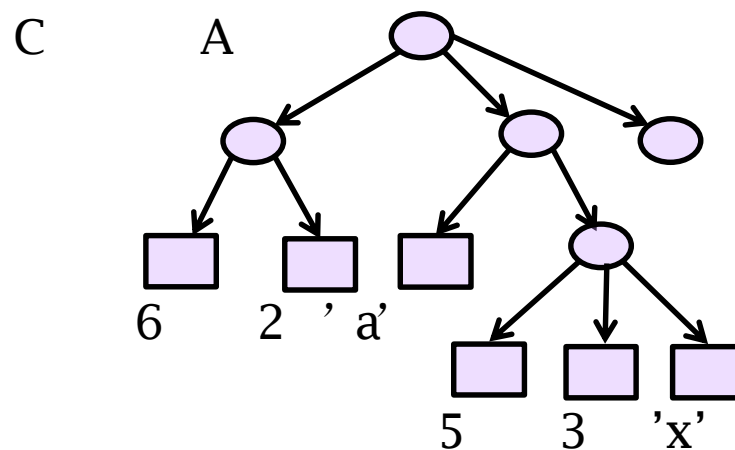
B



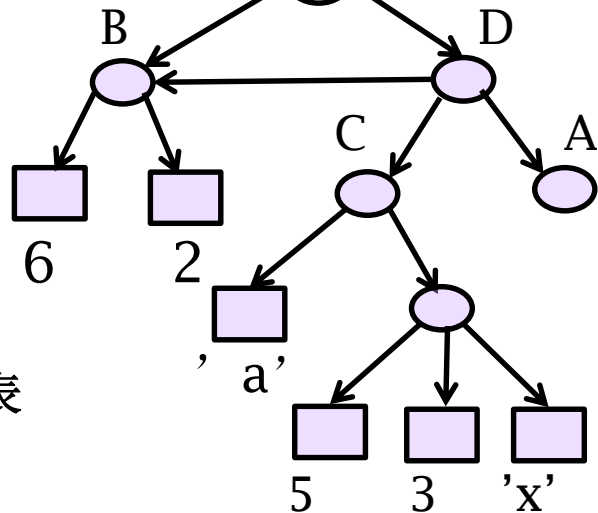
C



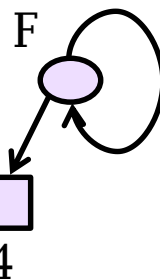
D



E



再入表



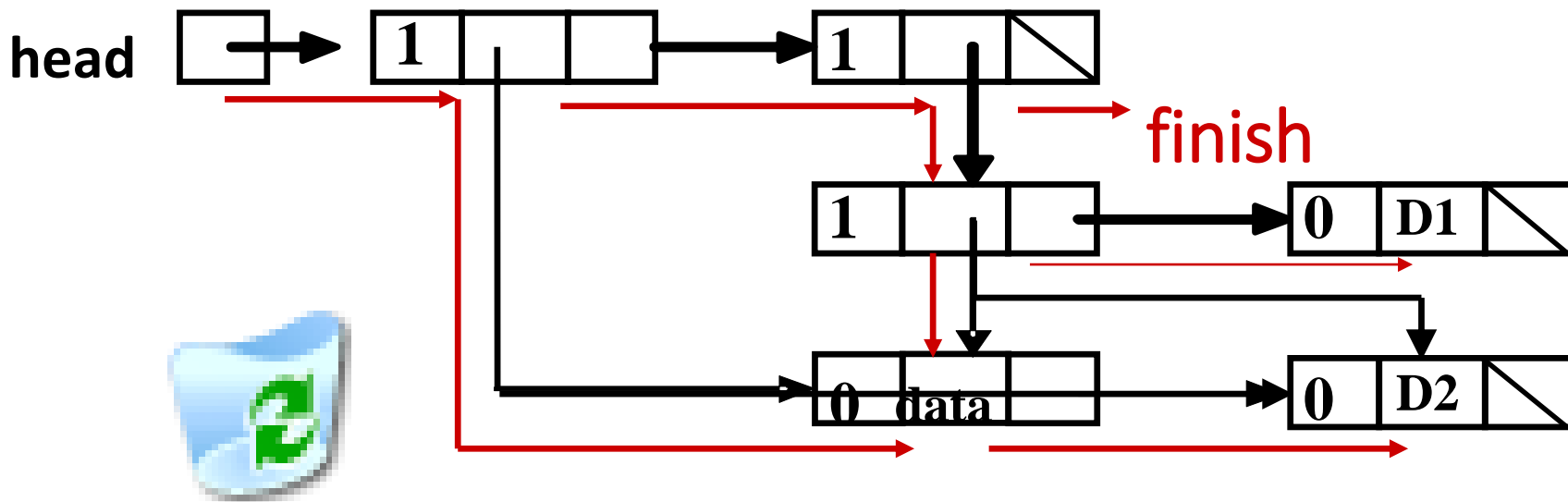
插入表



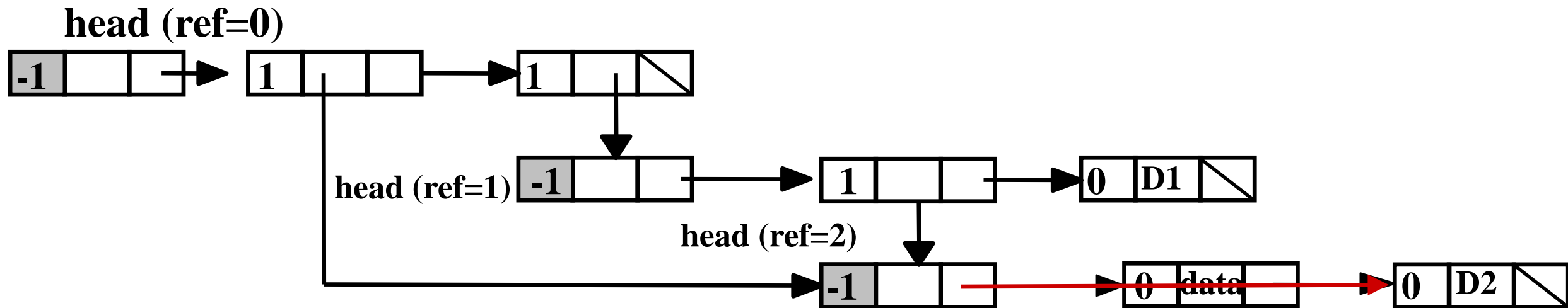
- 图 \supseteq 再入表 \supseteq 纯表 (树) \supseteq 线性表
 - 广义表是线性与树形结构的推广
- 递归表是有回路的再入表
- 广义表应用
 - 函数的调用关系
 - 内存空间的引用关系
 - LISP 语言

广义表存储ADT (续)

- 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



广义表存储ADT (续)

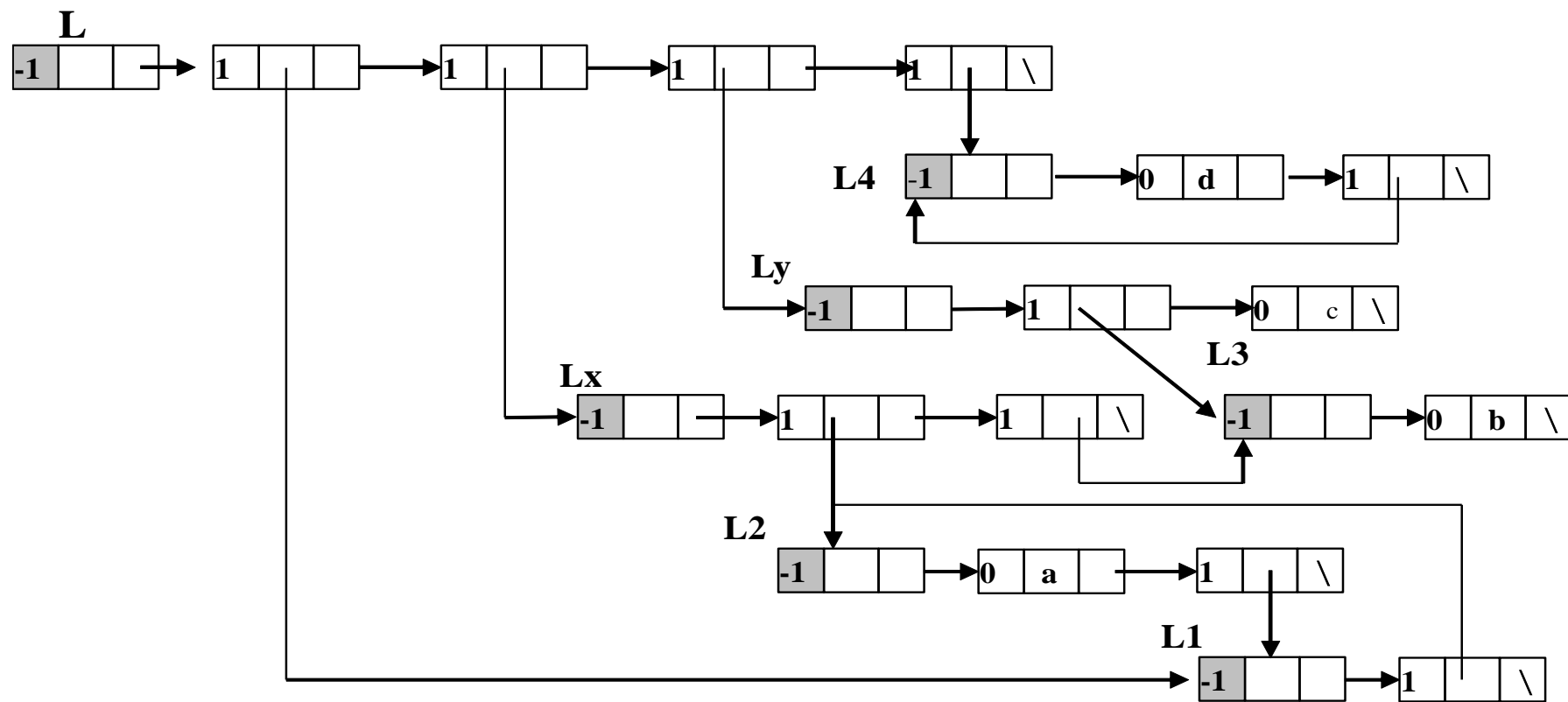


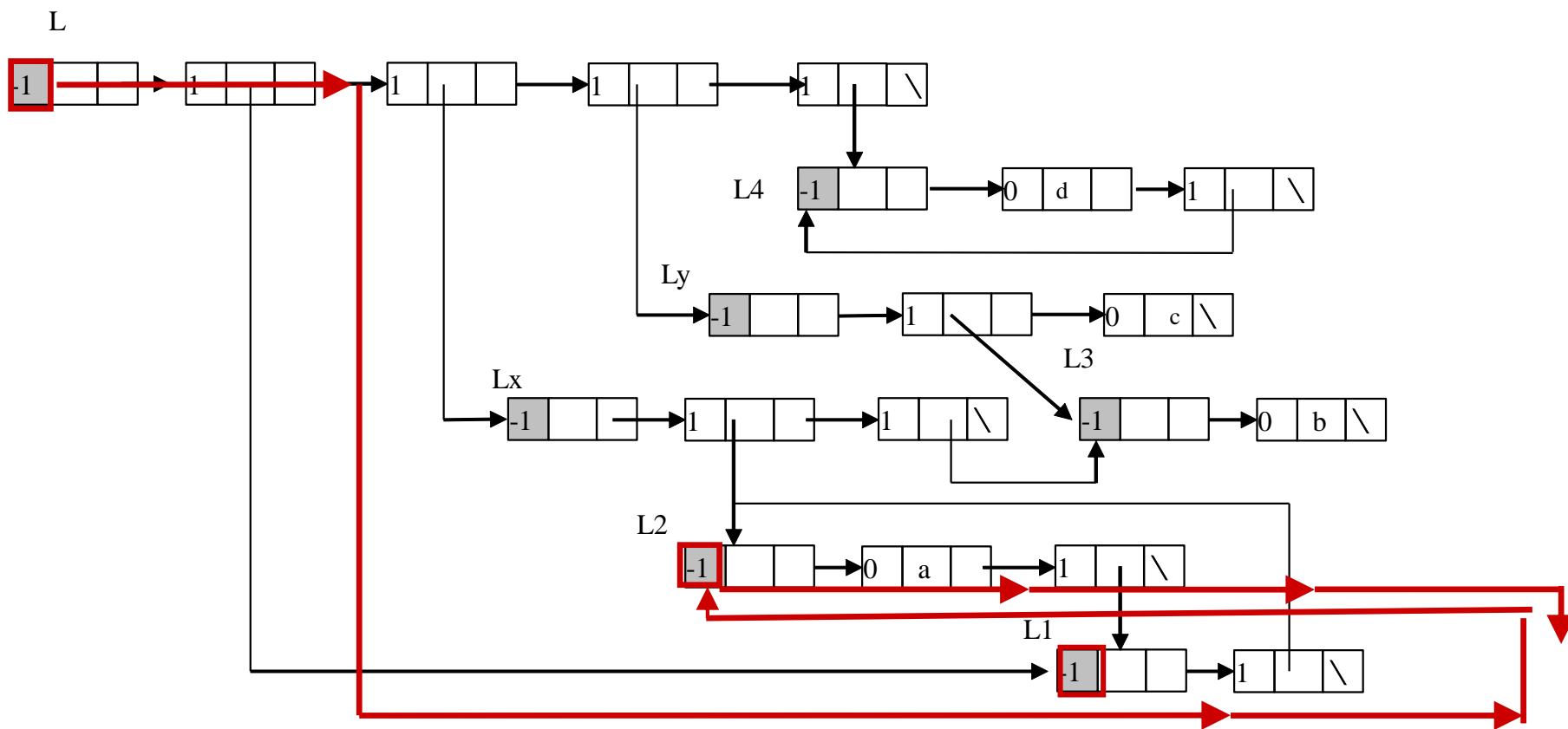
- 增加头指针，简化删除、插入操作
- 重入表，尤其是循环表
 - mark 标志位——图的因素





带表头结点的循环广义表

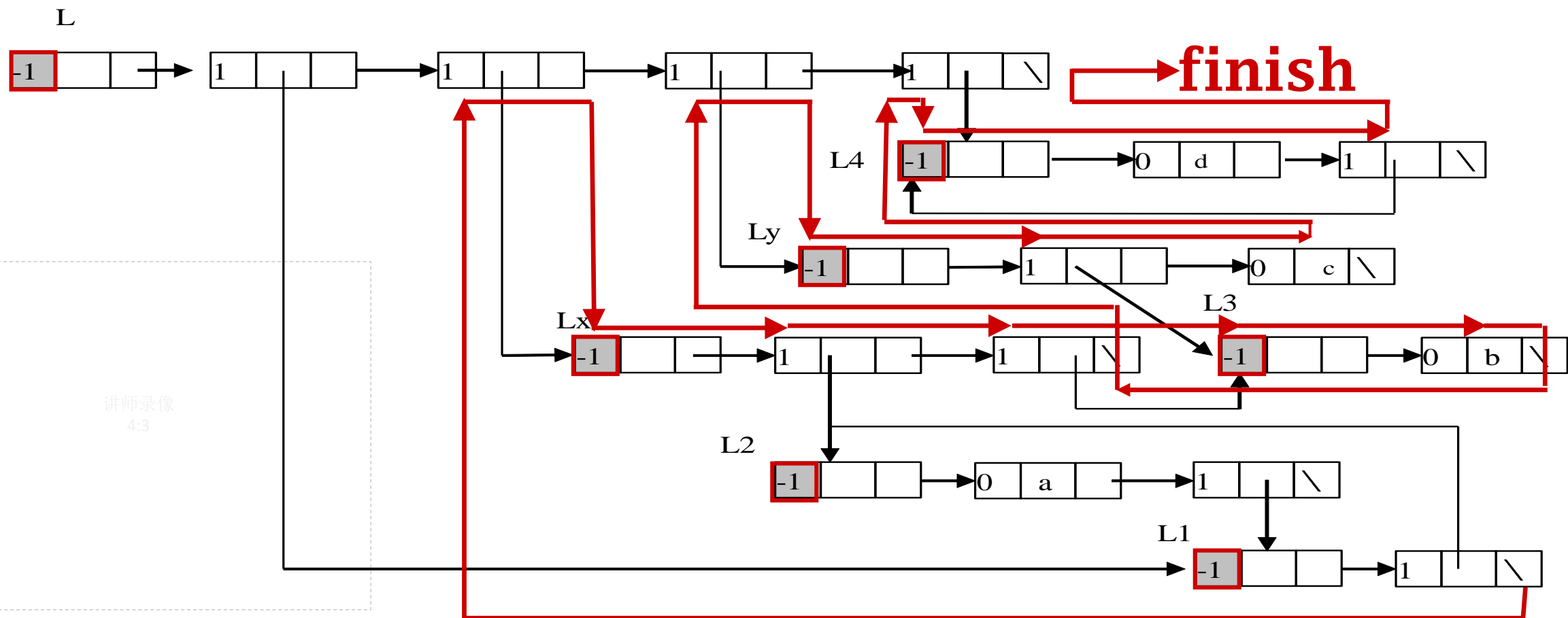


$$(L1: (L2: (a, L1)))$$




12.2 广义表和存储管理

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



讲师录像
4:3



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
 - 分配与回收
 - 可利用空间表
 - 存储的动态分配和回收
 - 失败处理策略和无用单元回收
- 12.4 Trie 树
- 12.5 改进的二叉搜索树

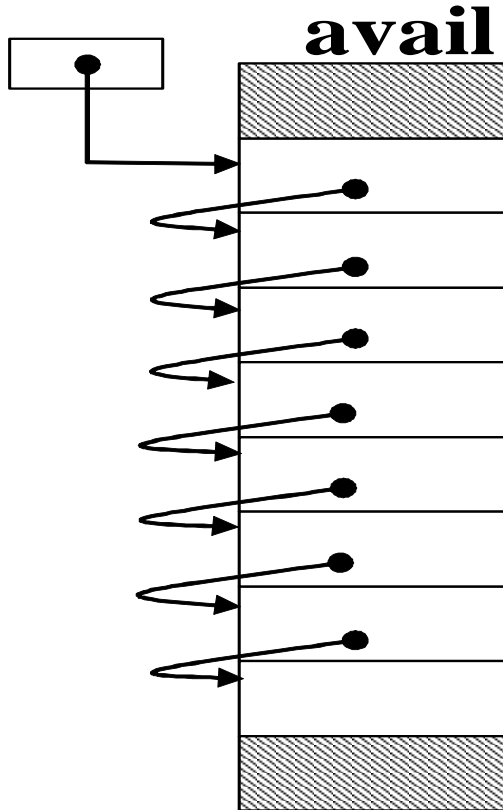
分配与回收

- 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
 - 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

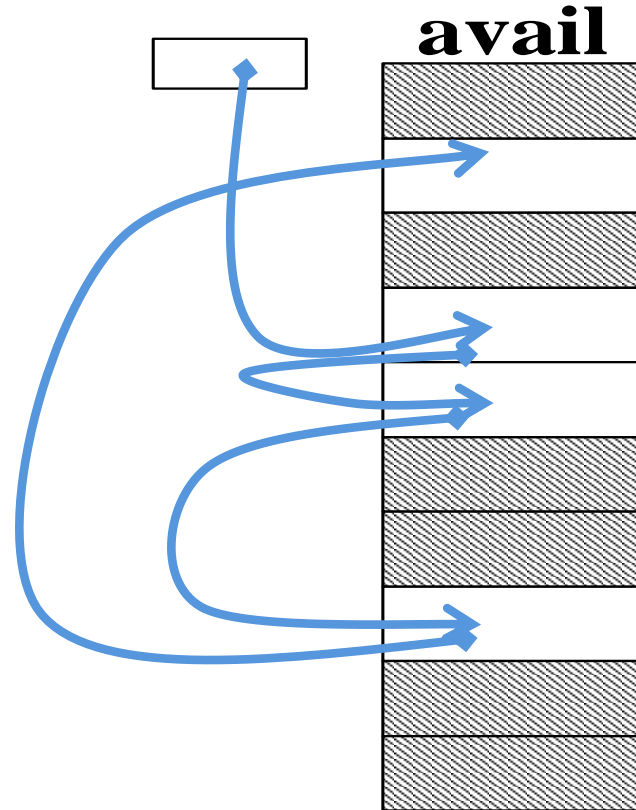


可利用空间表

- 把存储器看成一组变长块数组
 - 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
- 空间不够，则求助于失败策略



(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  
};
```




```
//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>
```

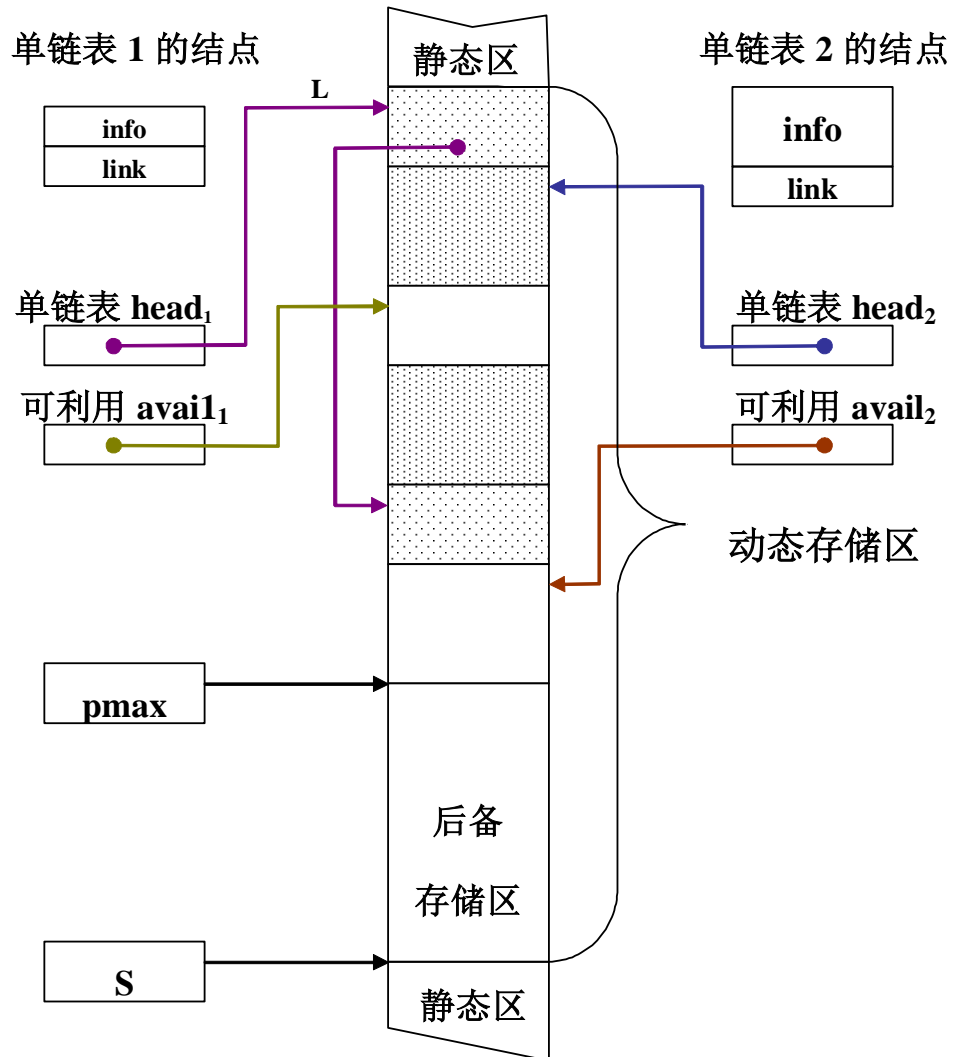
```
void LinkNode<Elem>::operator delete (void * p) {  
    ( (LinkNode<Elem> *) p) ->next = avail;  
    avail = (LinkNode<Elem> *) p;  
}
```



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）

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

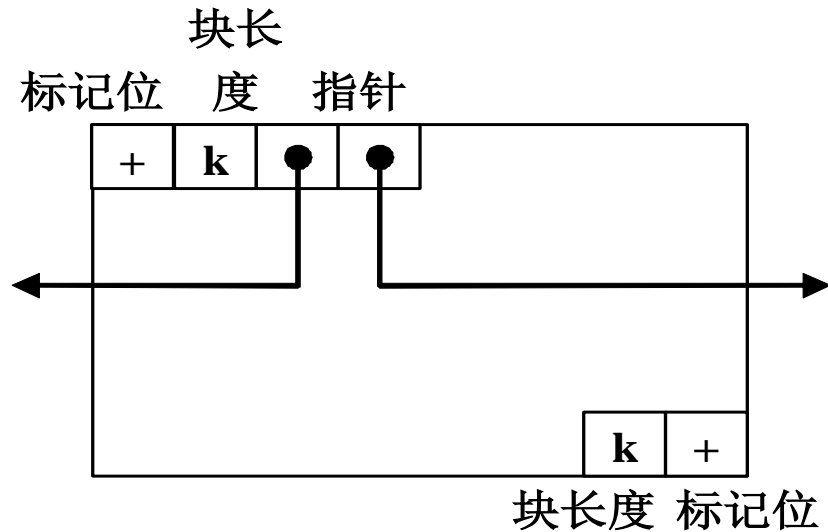


存储的动态分配和回收

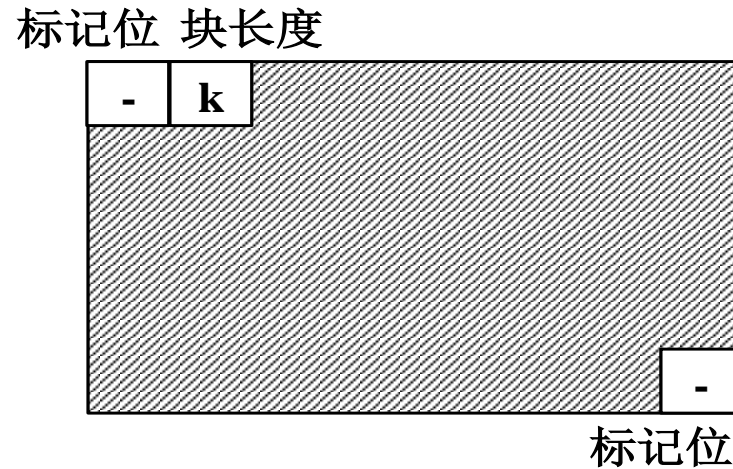
变长可利用块

- 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

空闲块的数据结构

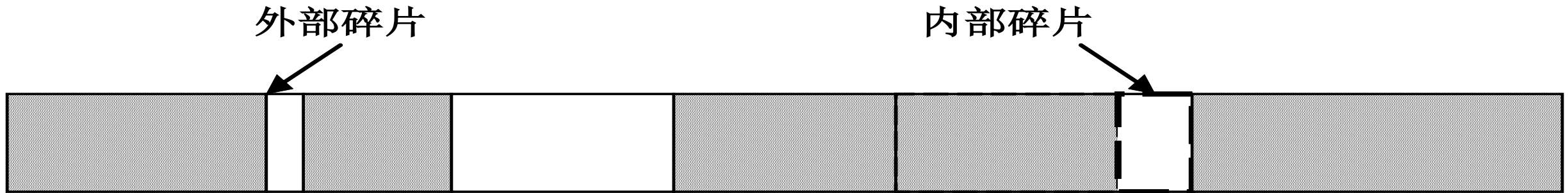


(a) 空闲块的结构



(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：多于请求字节数的空间
- 外部碎片：小空闲块



顺序适配 (sequential fit)

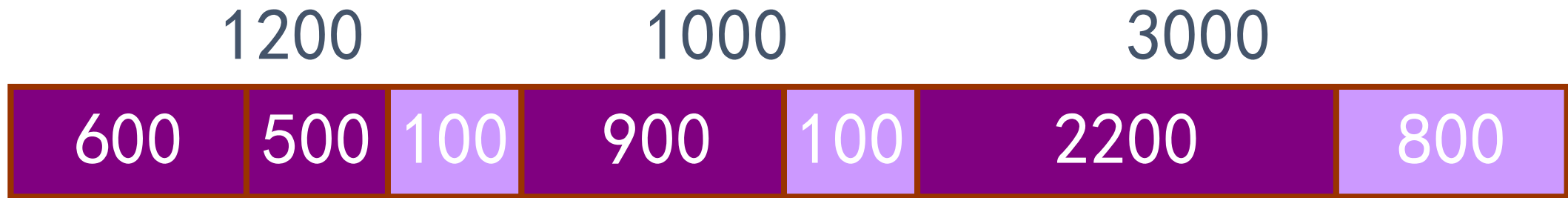
空闲块分配方法

- 常见的顺序适配方法：
 - 首先适配 (first fit)
 - 最佳适配 (best fit)
 - 最差适配 (worst fit)



12.3 存储管理

顺序适配



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



顺序适配

- 最佳适配

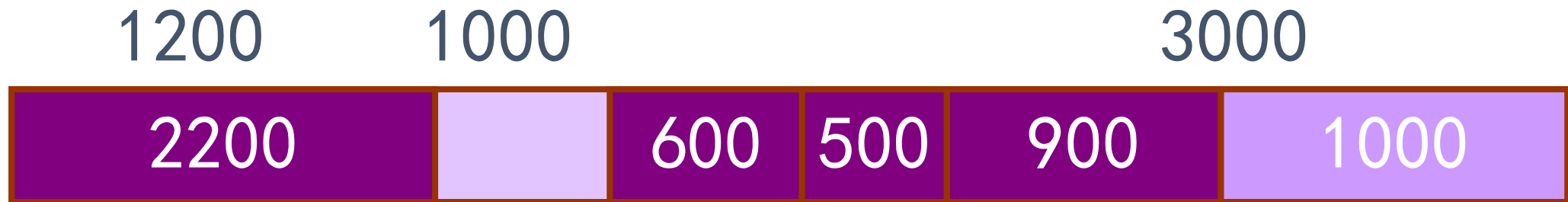


请求序列：600，500，900，2200

12.3 存储管理

顺序适配

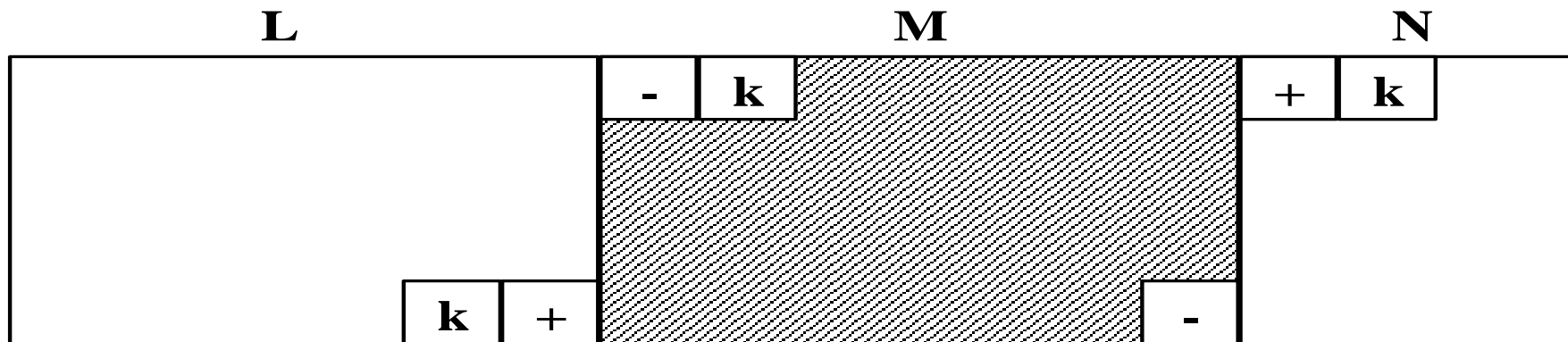
- 最差适配



为什么受伤的总是我？.....

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



适配策略选择

- 需要考虑以下因素用户的要求
 - 分配或回收效率对系统的重要性
 - 所分配空间的长度变化范围
 - 分配和回收的频率
- 在实际应用中，首先适配 **最常用**
 - 分配和回收的速度比较快
 - 支持比较随机的存储请求

很难笼统地讲这哪种适配策略最好



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：
 - 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。



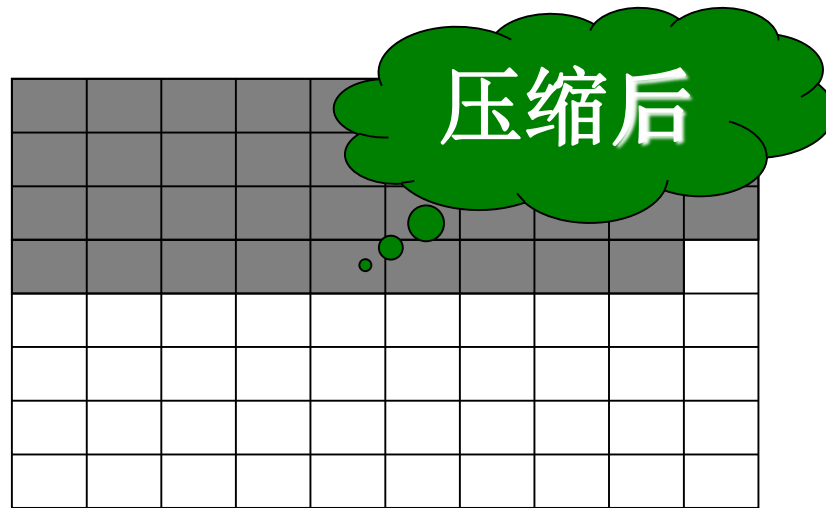
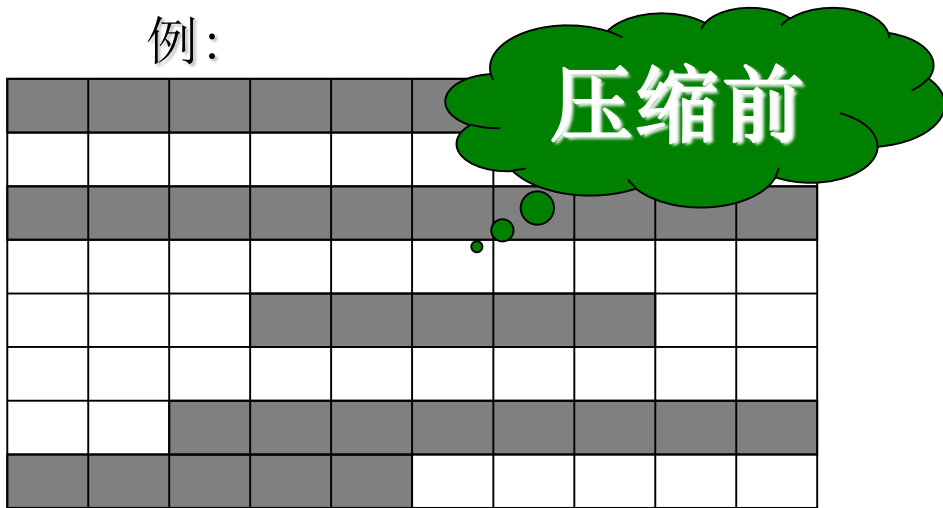
存储压缩 (compact)

- 把内存中的所有碎片集中起来
 - 组成一个大的可利用块
 - 内存碎片很多，即将产生溢出时使用
- 句柄使得存储地址相对化
 - 对存储位置的二级间接指引
 - 移动存储块位置，只需要修改句柄值
 - 不需要修改应用程序

两种存储压缩

- 一旦有用户释放存储块即进行回收压缩
- 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”

例：



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



数据结构与算法

谢谢聆听

国家精品课“数据结构与算法”

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

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6。“十一五”国家级规划教材