



# 数据结构与算法

## 第十一章 高级线性表

任课教员：张 铭

<http://db.pku.edu.cn/mzhang/DS/>

mzhang@db.pku.edu.cn

北京大学信息科学与技术学院

网络与信息系统研究所

©版权所有，转载或翻印必究



# 主要内容

---

- 11.1 多维数组
- 11.2 广义表
- 11.3 存储管理技术



# 11.1 多维数组

---

- ◆ 基本概念
- ◆ 数组的空间结构
- ◆ 数组的存储
- ◆ 数组的声明
- ◆ 用数组表示特殊矩阵
- ◆ 稀疏矩阵



# 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

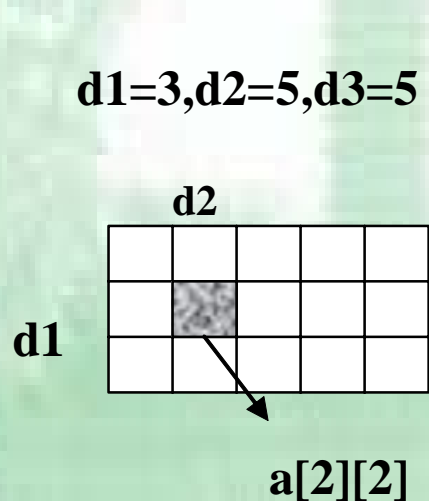


# 基本概念（续）

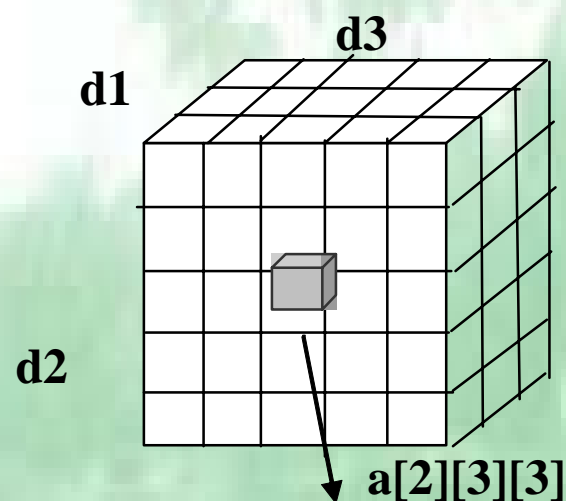
- 多维数组（Multi-array）是向量的扩充
  - 向量的向量就组成了多维数组
  - 可以表示为：
$$\text{ELEM } A[c_1..d_1][c_2..d_2]...[c_n..d_n]$$
  - $c_i$ 和 $d_i$ 是各维下标的下界和上界。所以其元素个数为：

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

# 数组的空间结构



二维数组



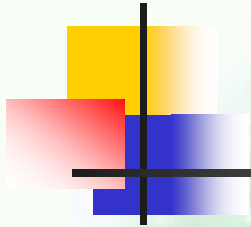
三维数组

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

# 数组的存储

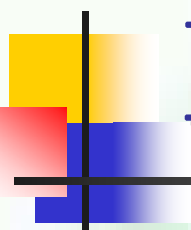
- 内存是一维的，所以数组的存储也只能是一维的
  - 以行为主序（也称为“行优先”）
  - 以列为主序（也称为“列优先”）

X=	1	2	3
	4	5	6
	7	8	9



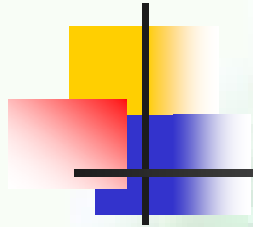
- C/C++、Pascal行优先
  - 先排最右的下标
  - 从右向左
  - 最后最左的下标
- 例如对于三维数组  
 $a[1..k, 1..m, 1..n]$  的元素  $a_{xyz}$  可以  
如下排列:





# Pascal语言的行优先存储

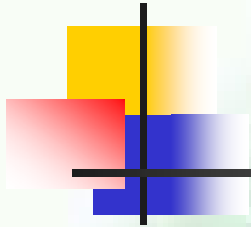
$a_{111}$	$a_{112}$	$a_{113}$	$\dots$	$a_{11n}$
$a_{121}$	$a_{122}$	$a_{123}$	$\dots$	$a_{12n}$
$\dots\dots\dots$				
$a_{1m1}$	$a_{1m2}$	$a_{1m3}$	$\dots$	$a_{1mn}$
$a_{211}$	$a_{212}$	$a_{213}$	$\dots$	$a_{21n}$
$a_{221}$	$a_{222}$	$a_{223}$	$\dots$	$a_{22n}$
$\dots\dots\dots$				
$a_{2m1}$	$a_{2m2}$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



- FORTRAN列优先
  - 先排最左的下标
  - 从左向右
  - 最后最右的下标
- 例如对于三维数组 $a[1..k, 1..m, 1..n]$ 的元素 $a_{xyz}$ 可以如下排列:

# FORTTRAN的列优先存储

$$\begin{array}{cccc}
 a_{11} & a_{21} & a_{31} & \dots & a_{k1} \\
 a_{12} & a_{22} & a_{32} & \dots & a_{k2} \\
 \dots & \dots & \dots & \dots & \dots \\
 a_{1m} & a_{2m} & a_{3m} & \dots & a_{km} \\
 a_{11} & a_{21} & a_{31} & \dots & a_{k1} \\
 a_{12} & a_{22} & a_{32} & \dots & a_{k2} \\
 \dots & \dots & \dots & \dots & \dots \\
 a_{1m} & a_{2m} & a_{3m} & \dots & a_{km} \\
 a_{11n} & a_{21n} & a_{31n} & \dots & a_{k1n} \\
 a_{12n} & a_{22n} & a_{32n} & \dots & a_{k2n} \\
 \dots & \dots & \dots & \dots & \dots \\
 a_{1mn} & a_{2mn} & a_{3mn} & \dots & a_{kmn}
 \end{array}$$



- C++ 多维数组 **ELEM**  $A[d_1][d_2] \dots [d_n];$

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



# 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  $\text{list}[(i^2+i)/2 + j] \ (i \geq j)$

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



# 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 $a_{i,j}=a_{j,i}$ ,  $0 \leq (i,j) < n$ 
  - 例如无向图的相邻矩阵
- 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 $a_{i,j}$ 之间存在着一一对应的关系:

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

# 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3对角矩阵:

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





# 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

$$\mathbf{A}_{6 \times 7} = \begin{pmatrix} 0 & 0 & 0 & 7 & 0 & 0 & 5 \\ 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 17 & 0 \\ 0 & 78 & 0 & 0 & 0 & 22 & 0 \\ 11 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 42 & 0 & 0 & 0 & 0 \end{pmatrix}$$



## ■ 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：

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

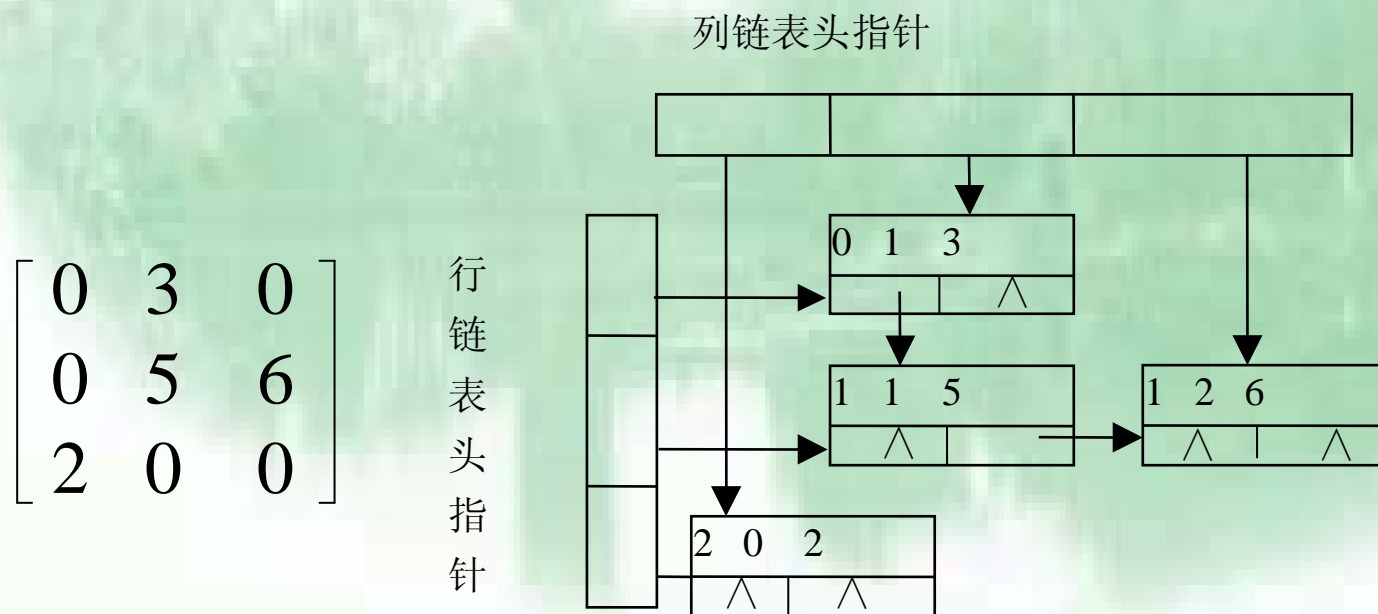
- 当这个值小于 **0.05** 时，可以认为是稀疏矩阵

## ■ 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

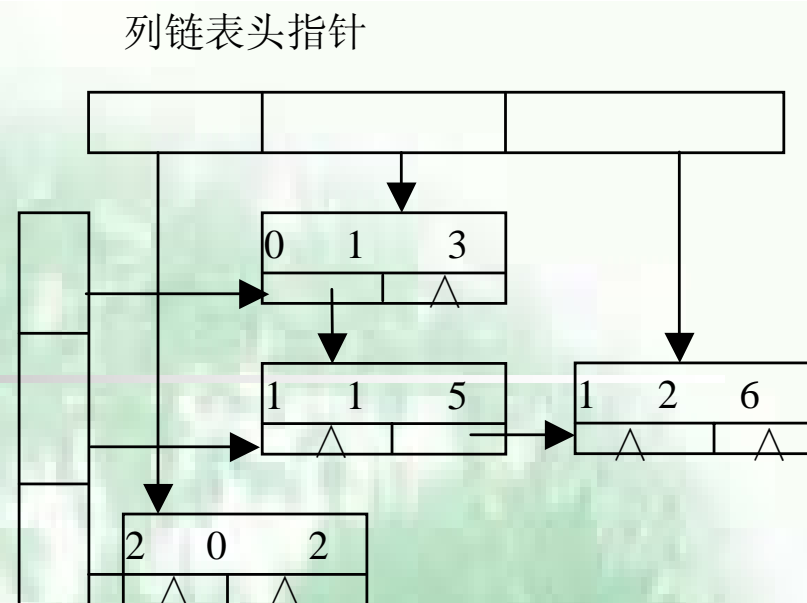
# 稀疏矩阵的十字链表

- 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



# 十字链表的建立

行链表头指针



- 建立矩阵的算法如下：
  - 首先为行头结点和列头结点申请空间，大小分别为矩阵的行列数
  - 将三元组根据情况分别加入到链表中
    - 如果三元组中的行列号错误，则退出，否则继续
    - 先处理行链表的问题
      - 如果该行头结点为空，则建立一个新的头结点，内容为该三元组
      - 如果不为空则从头结点开始查找，找到该三元组的正确位置如果该位置已经存在数据，则修改之，否则生成相应的结点插入进去
    - 类似地处理列链表头

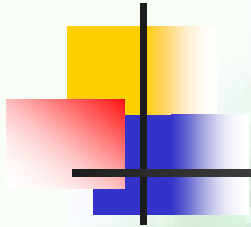


# 经典矩阵乘法

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

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

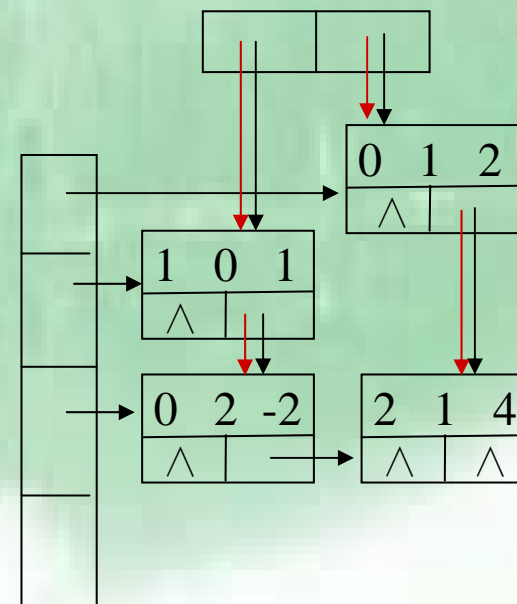
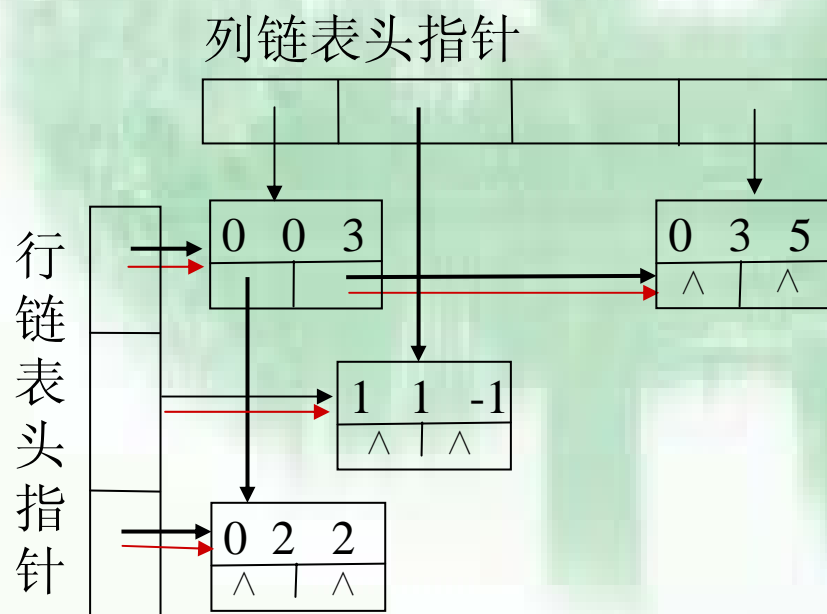
- ```
for (i=c1; i<=d1; i++)
  for (j=c2; j<=d2; j++) {
    sum = 0;
    for (k=c3; k<=d3; k++)
      sum = sum + A[i, k]*B[k, j];
    C[i, j] = sum;
  }
```



- $p = d_1 - c_1 + 1$ ,  $m = d_3 - c_3 + 1$ ,  
 $n = d_2 - c_2 + 1$ ;
- **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 $O(p \times m \times n)$

# 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 $p \times m$ 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 $O(p \times m \times n)$



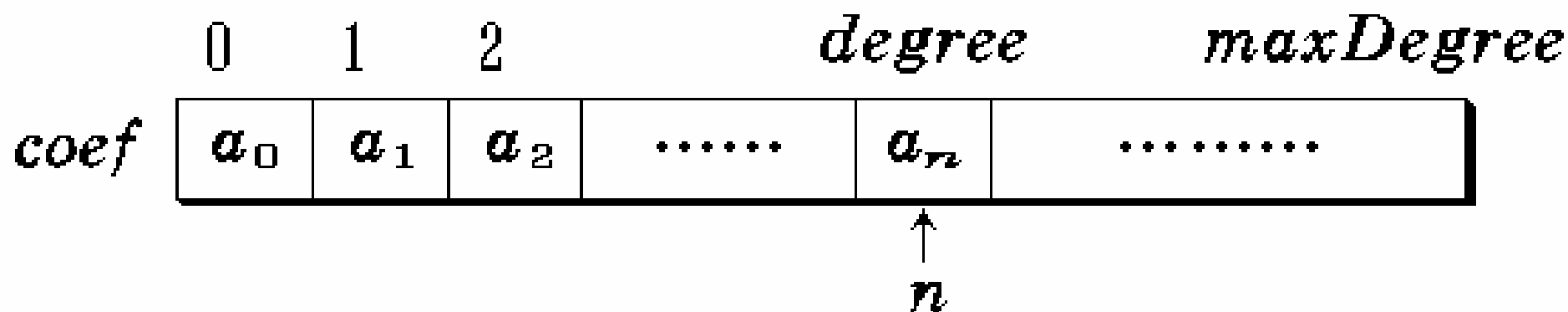


# 稀疏矩阵的应用

## ■ 一元多项式

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

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





## 11.2 广义表

---

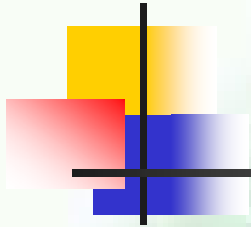
- ◆ 基本概念
- ◆ 广义表的各种类型
- ◆ 广义表的存储
- ◆ 广义表的周游算法



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



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

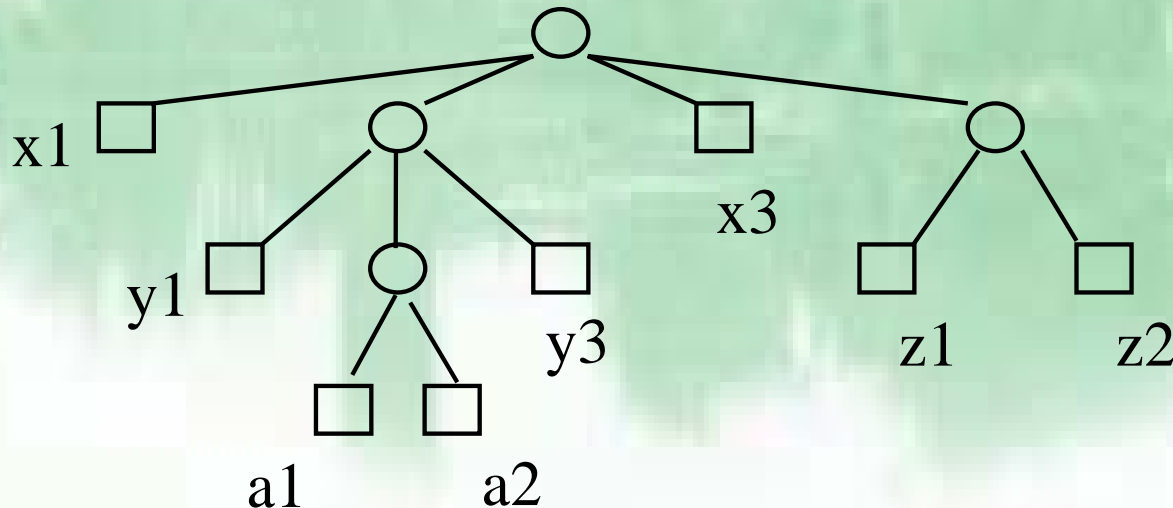
- $L$  是广义表的名称
- $n$  为长度
- 每个  $x_i$  ( $0 \leq i \leq n-1$ ) 是  $L$  的成员
  - 可以是单个元素，即原子 (atom)
  - 也可以是一个广义表，即子表 (sublist)
- 广义表的深度：表中元素都化解为原子后的括号层数

# 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
- 也就是说任何一个元素 (原子、子表) 只能在广义表中出现一次

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



# 广义表的各种类型（续）

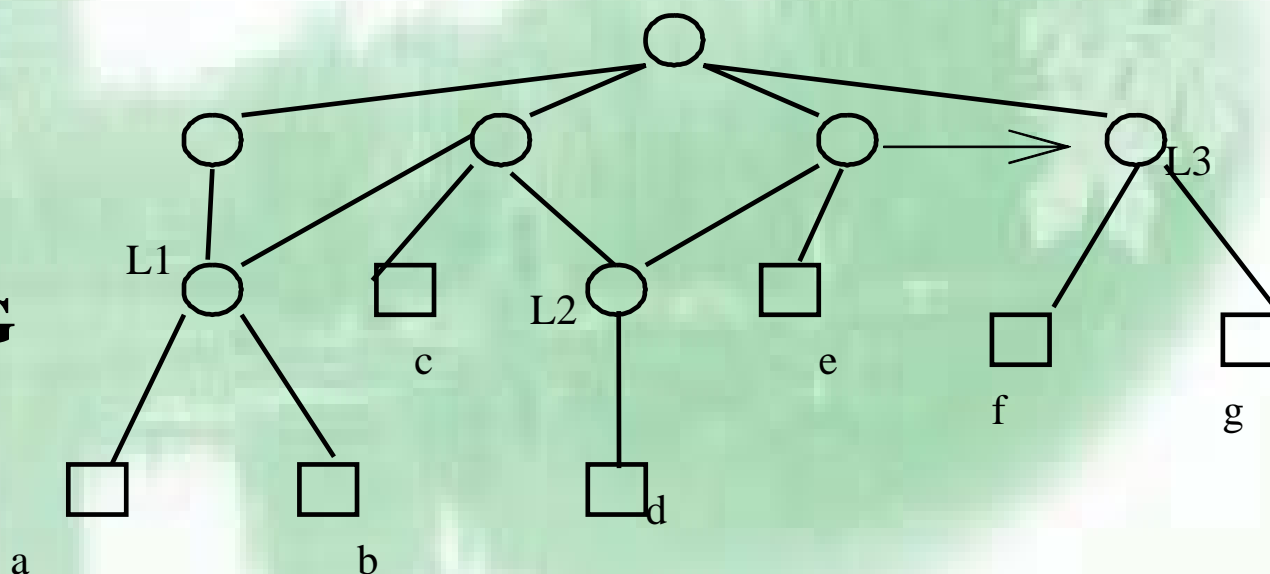
- 可重入表

- 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）

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

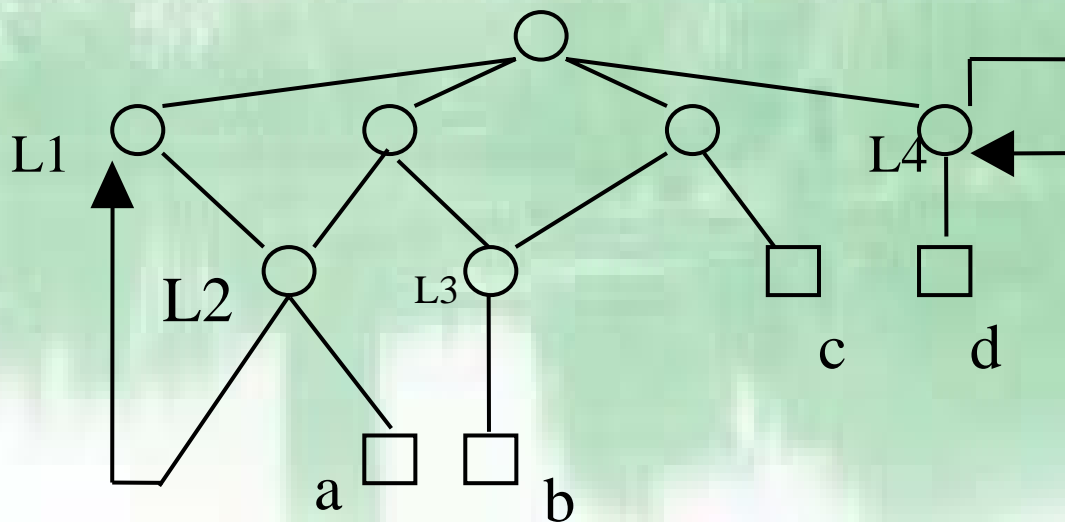


$(L1:(a,b), (L1, c, L2:(d)), (L2, e, L3:(f,g)), L3)$

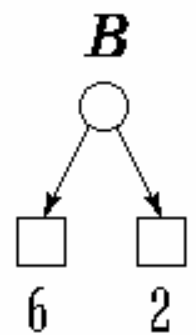
# 广义表的各种类型（续）

- 循环表
  - 包含回路
  - 循环表的深度为无穷大

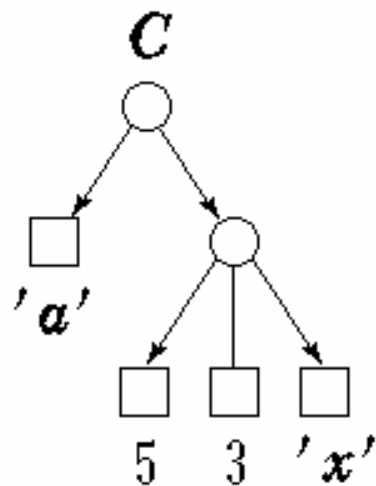
$(L1:(L2:(L1, a)), (L2, L3:(b)), (L3, c), L4:(d, L4))$



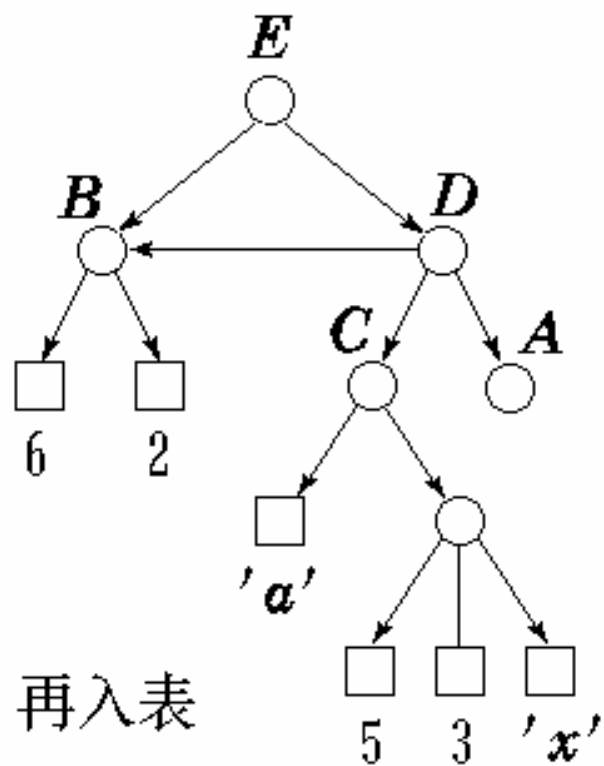
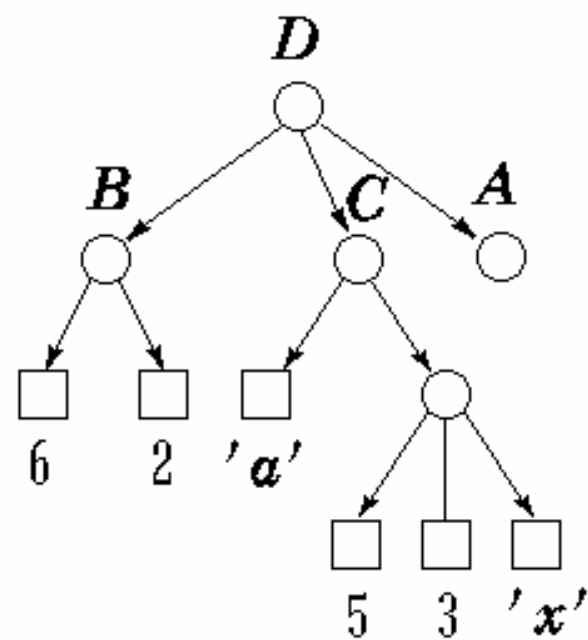
**A**  
○  
空表



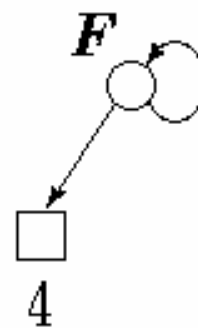
线性表



纯表

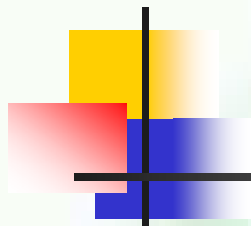


再入表



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



# 广义表的存储

- 使用数组来存储
  - 存储括号

|   |    |   |    |   |    |    |   |    |   |    |   |    |    |   |   |
|---|----|---|----|---|----|----|---|----|---|----|---|----|----|---|---|
| ( | x1 | ( | y1 | ( | a1 | a2 | ) | y3 | ) | x3 | ( | z1 | z2 | ) | ) |
|---|----|---|----|---|----|----|---|----|---|----|---|----|----|---|---|

- 可以使用链表结构存储



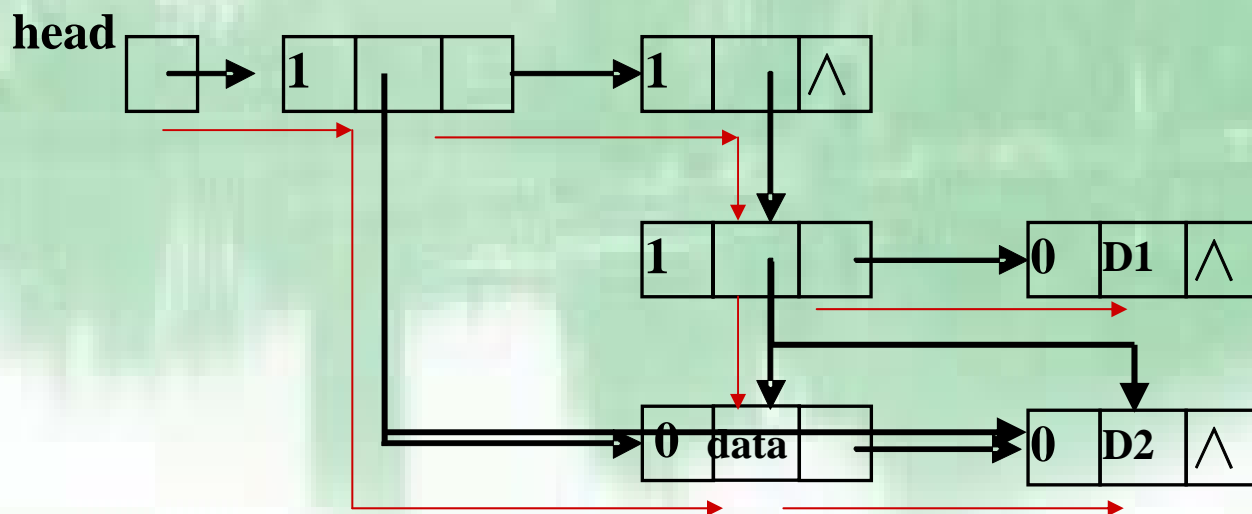
# 广义表存储ADT

---

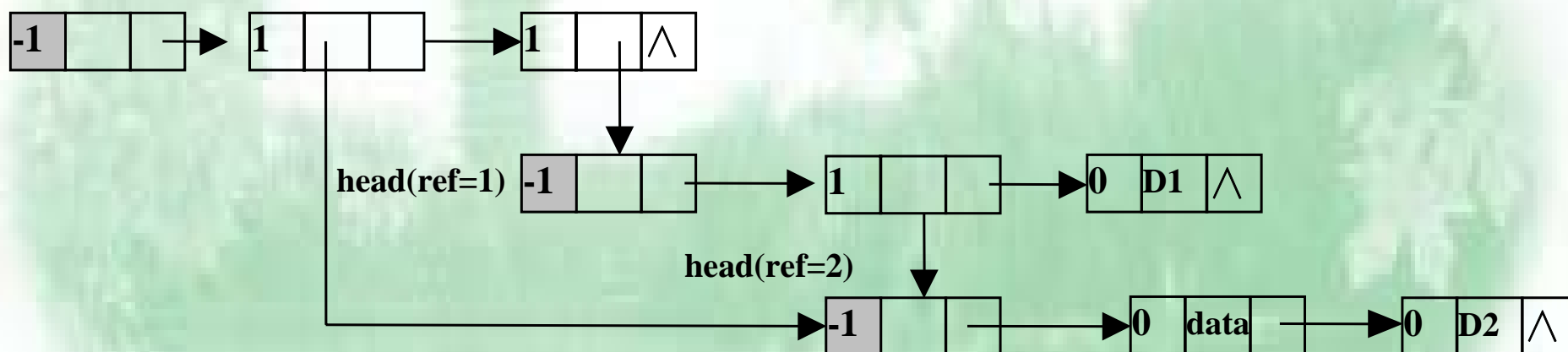
```
template <class T>
class GenListNode
{
    public:
        int type; //表示该结点是ATOM或者SubLIST
        T element; //如果是ATOM, 则存储它的值
        //如果是LIST, 则指向它的元素的首结点
        GenListNode<T> *child;
        GenListNode<T> *next; //指向下一个结点
        ... // 其他函数
};
```

# 广义表存储ADT（续）

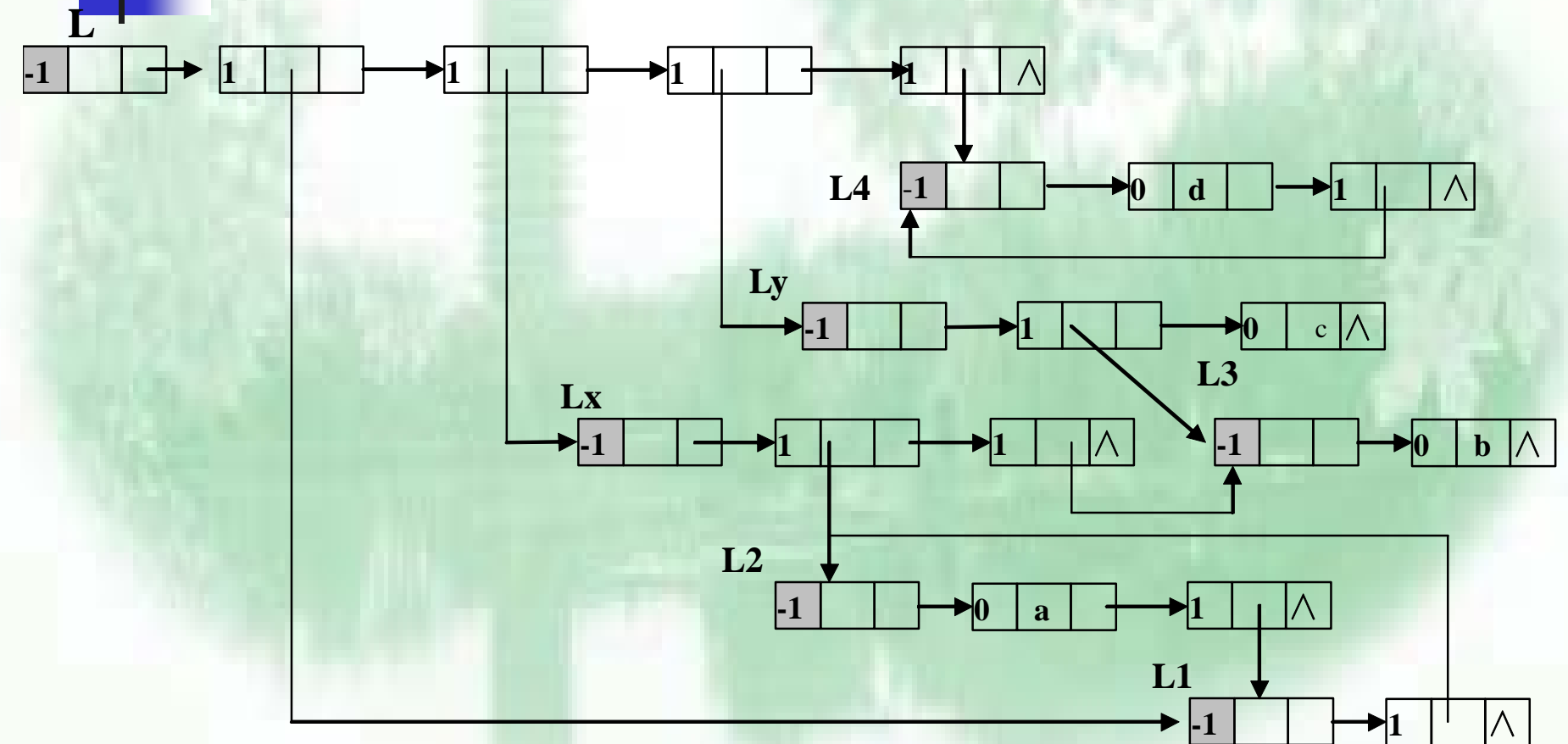
- 不带表头的广义表链
  - 在删除结点的时候会出现问题
  - 删除结点**data**就必须进行链调整

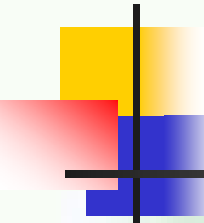


- ## head(ref=0)



- 重入表，尤其是循环表
  - mark标志位——图的因素





//改进的广义表结点类型

template <class T>

class GenListNode

{

public:

int type; //表示该结点是**ATOM** or **LIST**

struct {

int ref; //如果是表头结点则，存储该结点被引用次数

char\* Name; // 表头名称

int mark; //本子表是否被访问过

} headNode;

GenListNode<T> \*child; //如果是**LIST** ,则指向子表

T element; //如果是**ATOM**，则存储它的值

GenListNode<T> \*next; //指向下一个结点

void GenListTraversal (); //周游该结点的子孙

void GenListTraversalHelp(GenListNode<T> \*node);

.....

};

//GenList是一个原子元素类型为T的广义表  
//如果要实现能够存储多种数据类型的广义  
//表，只需要嵌套的使用它



```
template <class T>  
class GenList
```

```
{
```

```
private:
```

```
    GenListNode<T> *head;//头结点,存储头信息
```

```
    GenListNode<T> *current;//当前指针的位置
```

```
public:
```

```
    GenList(char *Name);
```

```
    ~GenList();
```

```
    void Insert(T element);//在尾部加入一个元素结点
```

```
    void Insert(GenList<T> *gl);//在尾部插入一个子表
```

```
    GenListNode<T> *GetHead();//取头结点
```

```
    GenListNode<T> *GetNext();//取当前结点的下一个
```

```
    GenListNode<T> *GetPrev();//取当前结点的前一个
```

```
    void MoveFirst();//当前指针指向head
```

```
    int Remove();//删除当前结点
```

```
    void ViewList();//周游广义表
```

```
};
```

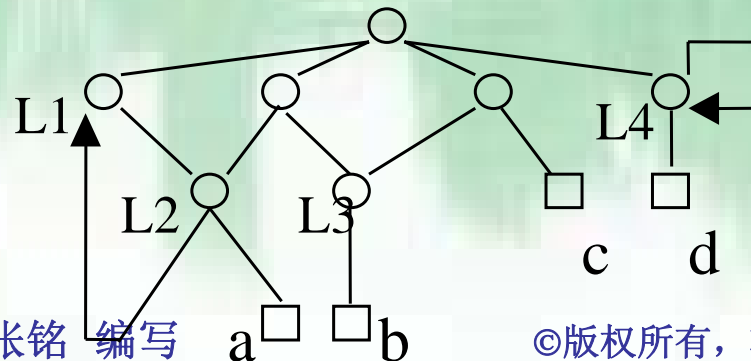


```

GenList<char*> *List=new GenList<char*>("List");
GenList<char*> *List1=new GenList<char*>("L1");
GenList<char*> *List2=new GenList<char*>("L2");
GenList<char*> *List3=new GenList<char*>("L3");
GenList<char*> *List4=new GenList<char*>("L4");
GenList<char*> *Listx=new GenList<char*>("");
GenList<char*> *Listy=new GenList<char*>("");
List3->Insert("b");
List4->Insert("d");      List4->Insert(List4);
Listy->Insert(List3);    Listy->Insert("c");
List2->Insert("a");      List2->Insert(List1);
List1->Insert(List2);
Listx->Insert(List2);    Listx->Insert(List3);
List->Insert(List1);     List->Insert(Listx);
List->Insert(Listy);     List->Insert(List4);
List->ViewList();

```

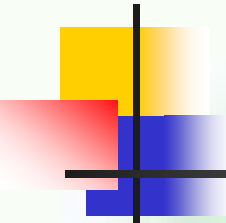
(L1:(L2:(, L1 a)), (L2, L3:(b)), (L3, c), L4:(d,L4))





# 广义表的周游算法

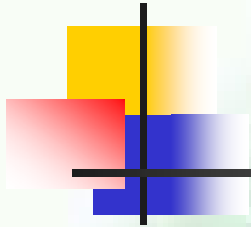
- 广义表周游的时候应该注意几个问题：
  - 相当于深度优先周游
  - 访问的时候首先进入一个子表的头结点，设置**mark**标记
  - 按照本层子结点顺序，访问广义表
    - 如果是子表结点，则准备递归地访问此子表表头结点
    - 如果是原子，则直接访问
  - 避免进入循环链中无法跳出
    - **mark**用来防止循环访问而设置的访问位
    - 实际上，表头结点才需要**mark**
- 广义表访问结束的时候
  - 应该将**mark**设置为未访问
  - 如果其他地方也引用了该链可以正常的访问到



```

template <class T>
void GenListNode<T>::GenListTraversalHelp(GenListNode<T>
    *node) {
    GenListNode<T> *p;
    node->headNode.mark=VISITED;
    cout << "(";
    for (p = node->next; p!=NULL; p=p->next) {
        //进入一个子表结点, 准备递归访问它的表头结点
        if ((p->type==LIST)&&(p->child!=NULL)) {
            cout << p->child->headNode.Name;
            if (p->child->headNode.mark == UNVISITED) {
                if (p->child->headNode.Name[0]!='\0')
                    cout << ":";
                GenListTraversalHelp(p->child);
            }
        }
        else if (p->type==ATOM) cout<<p->element;
        if ((p->next!=NULL) // &&(p->next->type!=HEAD))
            cout << ", ";
    }
    cout << ")";
}

```



```
template <class T>
void GenList<T>::ViewList()
{
    MoveToFirst();
    current->GenListTraversal ();
}
```

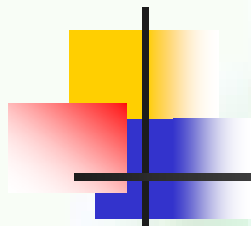
```
template <class T>
void GenListNode<T>::GenListTraversal ()
{
    GenListTraversalHelp(this);
}
```



## 11.3 存储管理技术

---

- ◆ 内存管理存在的问题
- ◆ 可利用空间表
- ◆ 存储的动态分配和回收
- ◆ 伙伴系统
- ◆ 失败处理策略和无用单元回收



- 动态内存分配
  - new和delete
- 内存管理技术
  - 链表、广义表

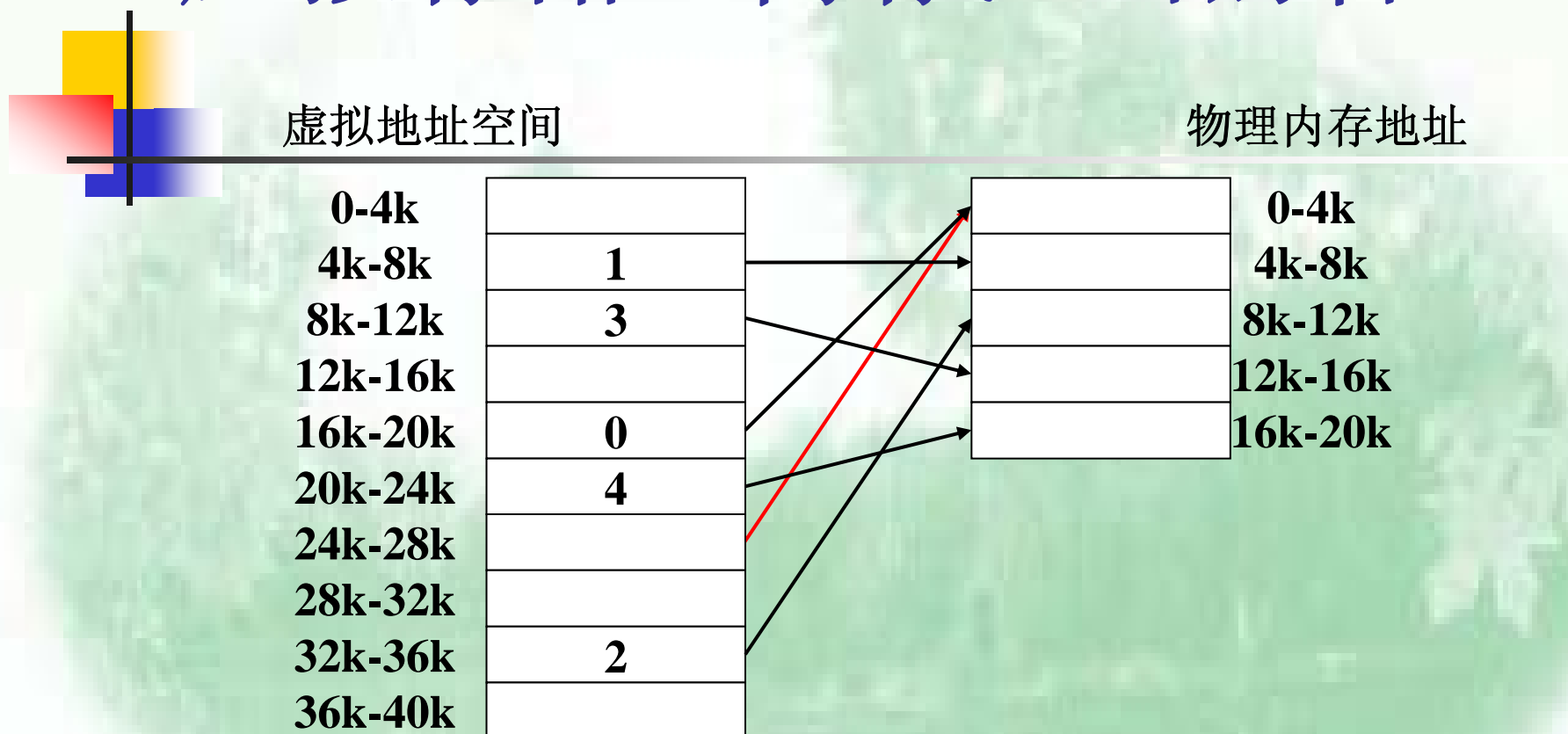


# 分配与回收

---

- 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

# 虚拟存储：内存溢出的管理



- 溢出发生后，把内存中某些数据暂存到外存
  - 选择最近不使用的那些结点

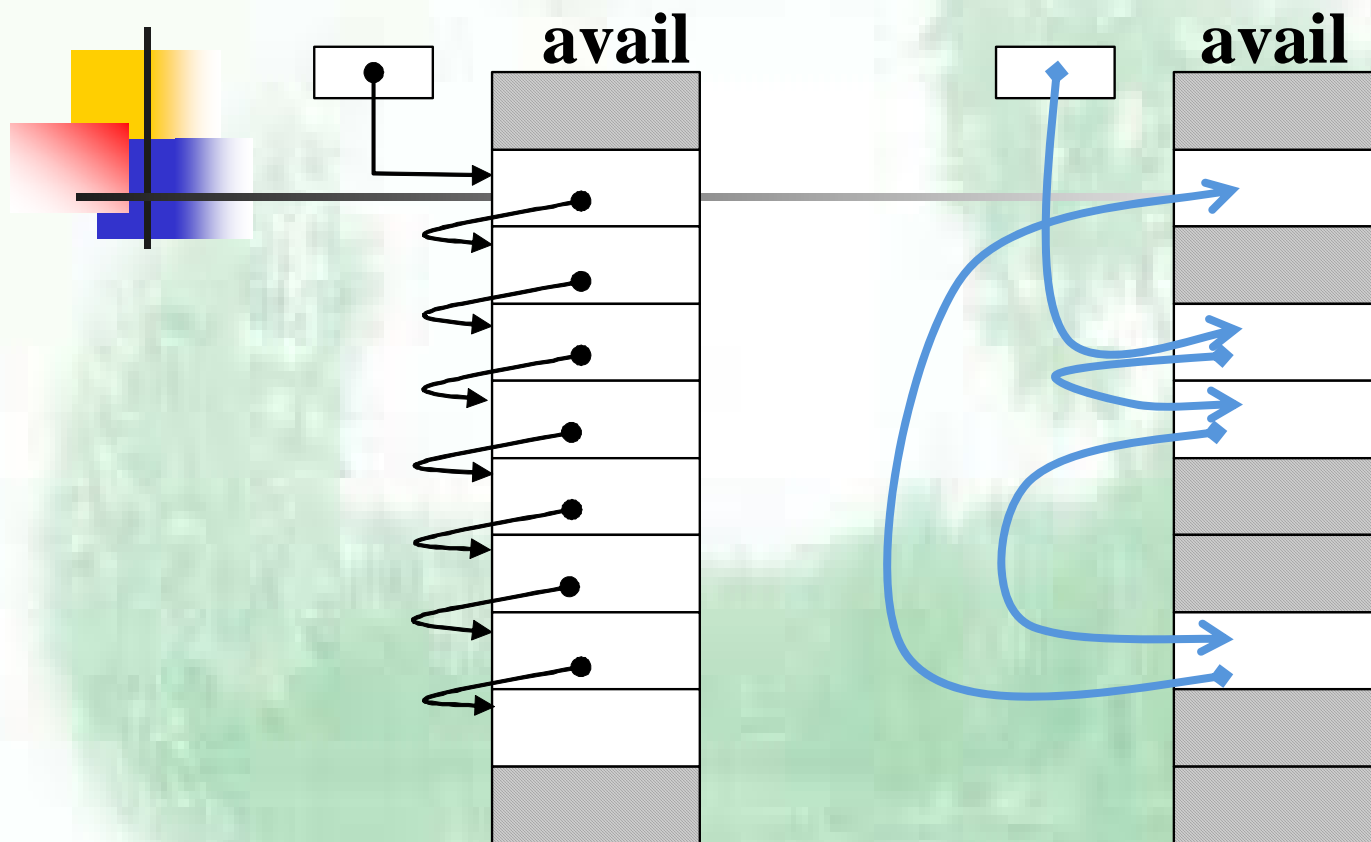




# 可利用空间表

---

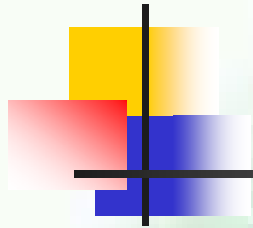
- 把存储器看成一组变长块数组
  - 一些块是已分配的
  - 链接空闲块，形成可利用空间表(**freelist**)
- 存储分配和回收
  - **new p** 从可利用空间分配
  - **delete p** 把p指向的数据块返回可利用空间表
- 空间不够，则求助于失败策略



(1) 初始状态的可利用空间表

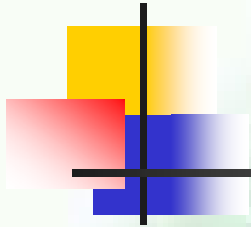
(2) 系统运行一段时间后的  
可利用空间表

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**

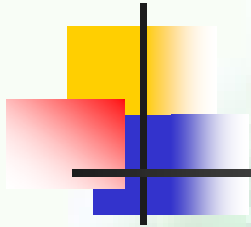
**return ::new LinkNode;//利用系统的new分配空间**

**LinkNode<Elem> \* temp = avail;       //否则从可利用  
空间表中取走一个结点**

**avail = avail->next;**

**return temp;**

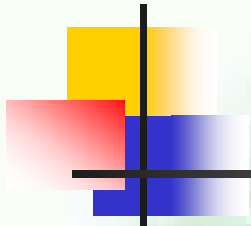
**}**



**//重载delete运算符实现**

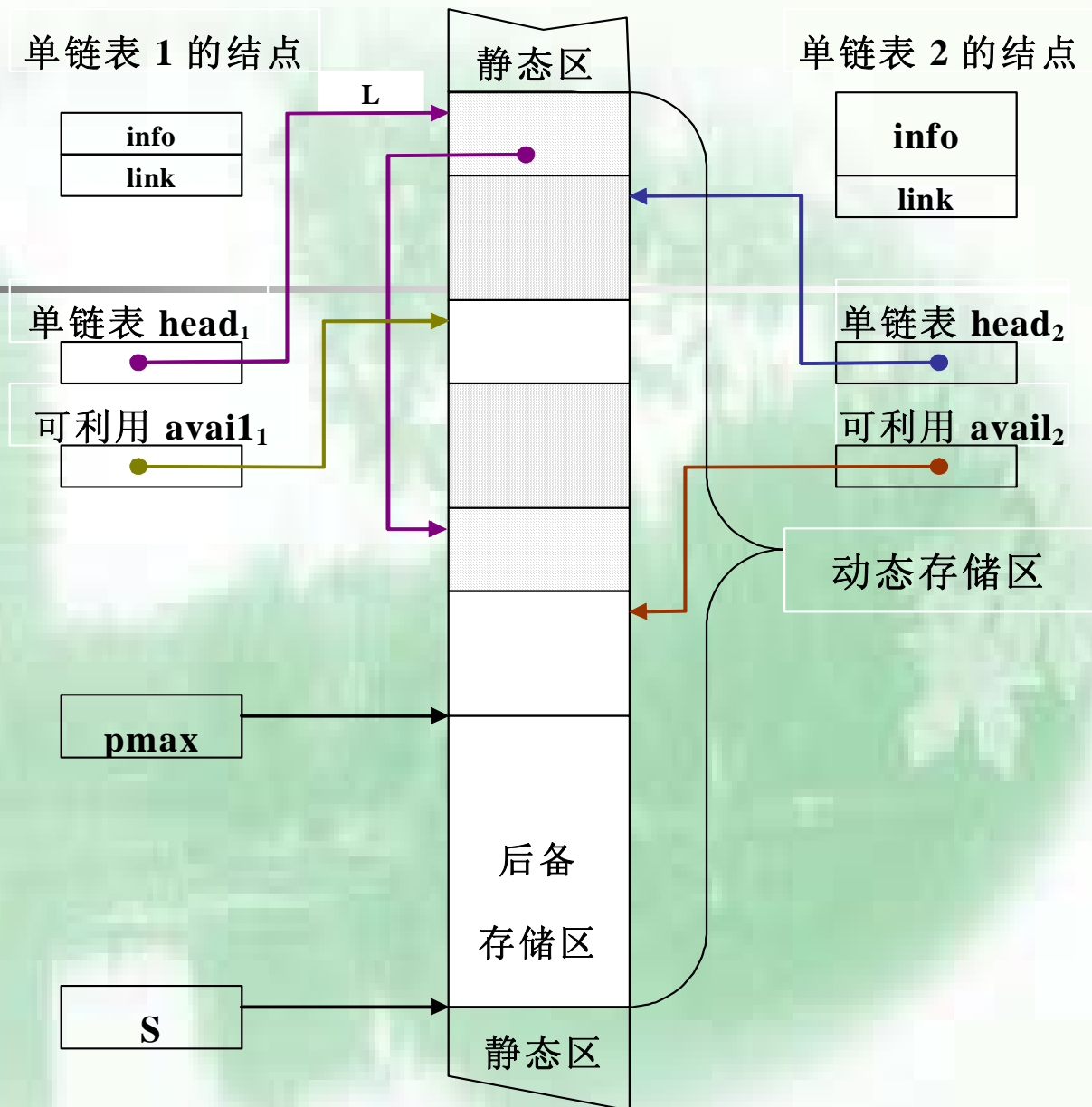
**template <class Elem>**

```
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）

- **pmax**值已经达到或超过S值，则不能再分配空间





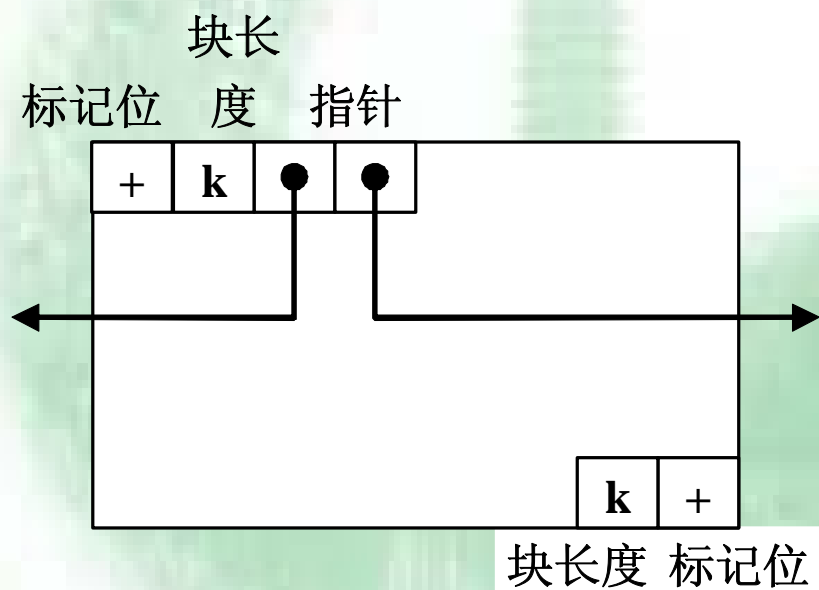
# 存储的动态分配和回收

---

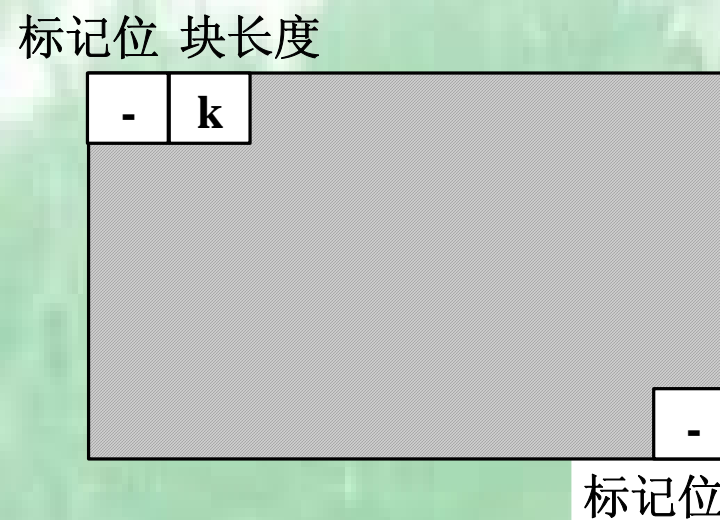
- 变长可利用块
- 分配
  - 找到其长度大于等于申请长度的结点
  - 从中截取合适的长度
- 回收
  - 考虑刚刚被删除的结点空间能否与邻接合并
  - 以便能满足后来的较大长度结点的分配请求



# 空闲块的数据结构



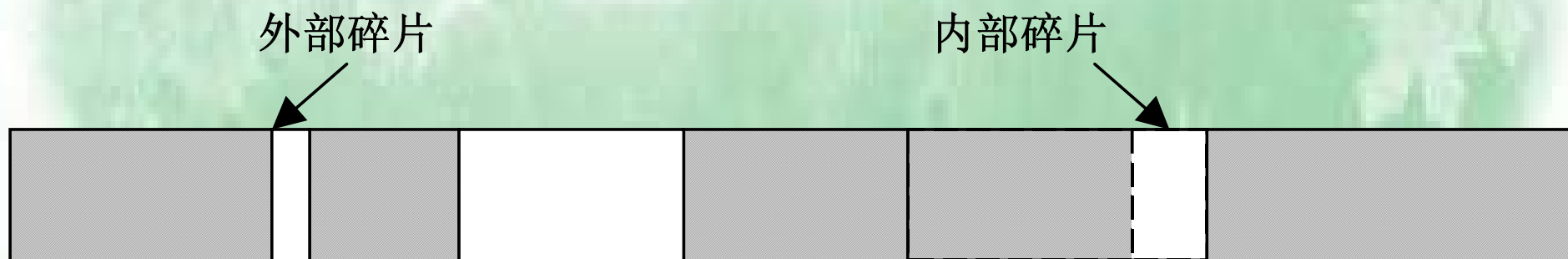
(a) 空闲块的结构



(b) 已分配块的结构

# 碎片问题

- 内部碎片：多于请求字节数的空间
- 外部碎片：小空闲块



外部碎片和内部碎片



# 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)



## 顺序适配（2）

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

1200

1000

3000

|     |     |     |     |     |      |     |
|-----|-----|-----|-----|-----|------|-----|
| 600 | 500 | 100 | 900 | 100 | 2200 | 800 |
|-----|-----|-----|-----|-----|------|-----|

# 顺序适配 (3)

## ■ 最佳适配

1200

1000

3000

|     |      |   |     |     |      |
|-----|------|---|-----|-----|------|
| 500 | 2200 | 0 | 400 | 900 | 2100 |
|-----|------|---|-----|-----|------|

请求序列: 600, 500, 900, 2200  
2200分配不了



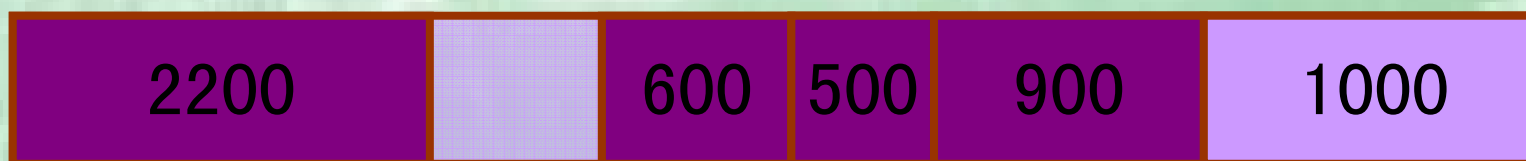
# 顺序适配（4）

## ■ 最差适配

1200

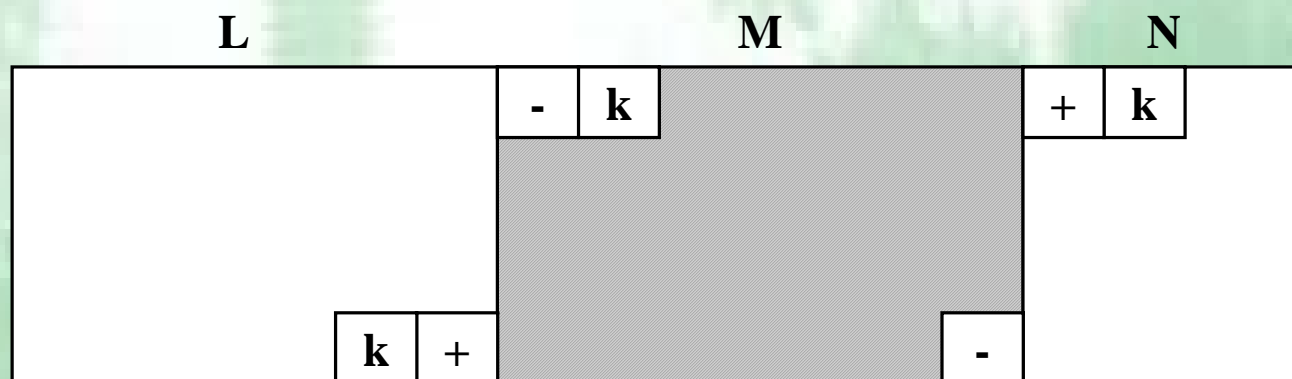
1000

3000



请求序列：600，500，900，2200  
2200分配不了

# 回收：考虑合并相邻块



把块 M 释放回可利用空间表



# 回收的策略

- 把块M释放到可利用空间表的过程：
  - 首先检查块M左邻的存储单元L，根据其标记位可以判断块L是不是空闲块。
  - 如果不是，则简单地把M插入可利用空间表。
  - 如果如图中所示，L是空闲块，则把L的长度扩展到包含M。然后，检查M的右邻块N的标记位，如果块N是空闲块，则要把N从可利用空间表中删除，同时扩展M的长度（在本例则是扩展已包含M的L的长度）。
  - 经过对块M的回收操作之后，就形成了一个包含原来的L、M、N三个块长度的大空闲块。





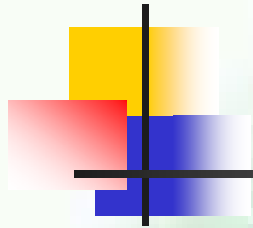
# 适配策略选择

- 很难笼统地讲这哪种适配策略最好
- 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



# 伙伴系统

- 假设存储空间的大小为 $2^M$ ， $M$ 为正整数
- 每一个空闲块和已分配块的大小都是 $2^k$ ， $k$ 小于等于 $M$ 
  - $2^k$ 大小的块首地址与它的伙伴的首地址相比，除了第 $k$ 位之外（最右为0位），所有的位都相同
- 空闲列表
  - 长度相同空闲块在一个列表中
  - 列表数目决不会超过 $M$ 个



伙伴

0000

0100

1000

伙伴

1100

|  |
|--|
|  |
|  |
|  |
|  |

## 伙伴系统的回收



# 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



# 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



# 引用计数算法

- 存储块加入一个计数字段
  - 新指针指向该存储块时，引用计数加1
  - 某指针不再指向这个块时，其引用计数减1
- 引用计数变为0，放回到可利用空间表
  - 不需要一个明显的无用单元收集阶段
- 缺点：
  - 系统的额外负担太多
  - 无用单元循环引用



# 标记/清除方法

标记/清除算法的实质就是周游广义表

- 首先清除所有的标记位
- 从变量表中的每一个变量开始，沿着指针进行深度优先搜索
  - 每遇到一个存储单元，就设置其标记位为“已访问”
- 最后访问所有存储单元，进行清理
  - 所有未标记的单元就认为是无用单元
  - 释放到可利用空间表





# Deutsch-Schorr-Waite

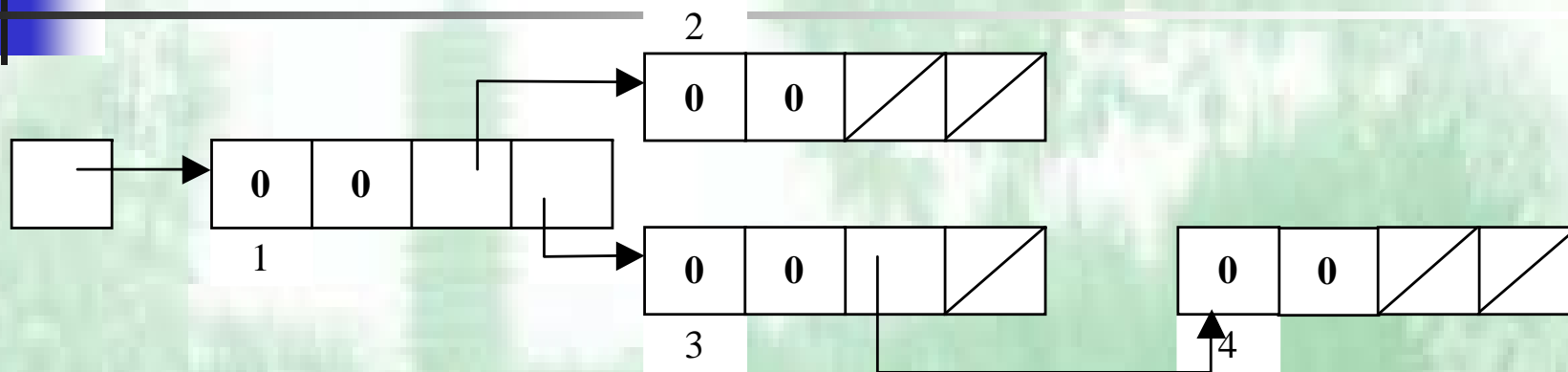
## 无用单元收集算法

---

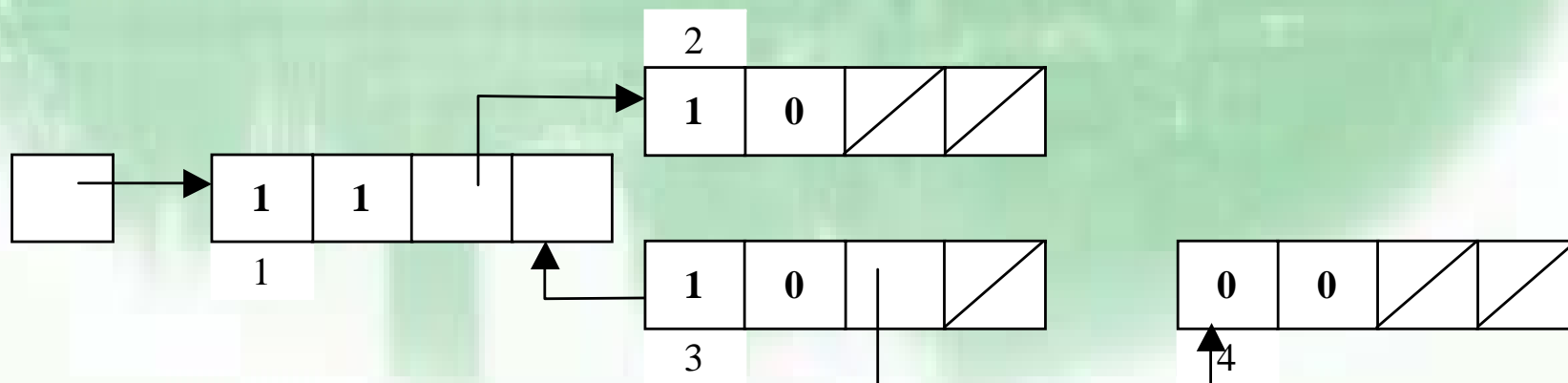
- 采用逆转链的广义表周游算法
  - 不需要额外的用于栈的空间
  - 也不需要用递归
- 借用即将深入的那个指针，把它逆转设置为指向前一步刚经过的结点
  - 不需要向栈中存储一个指针

| mark | tag | link <sub>1</sub> | link <sub>2</sub> |
|------|-----|-------------------|-------------------|
|------|-----|-------------------|-------------------|

(a) 广义表结点结构



(a) 初始广义表



(b) 处理节点 3



谢谢！

---