

第5章 数组和广义表

5.1 数组的定义和运算

5.2 数组的顺序存储和实现

5.3 特殊矩阵的压缩存储

5.3.1 三角矩阵

5.3.2 带状矩阵

5.3.3 稀疏矩阵

5.4 广义表

5.5 总结与提高

5.1 数组的定义和运算

数组(Array): 由一组名字相同、下标不同的变量构成

数组是一种数据类型。从逻辑结构上看，数组可以看成是一般线性表的扩充。

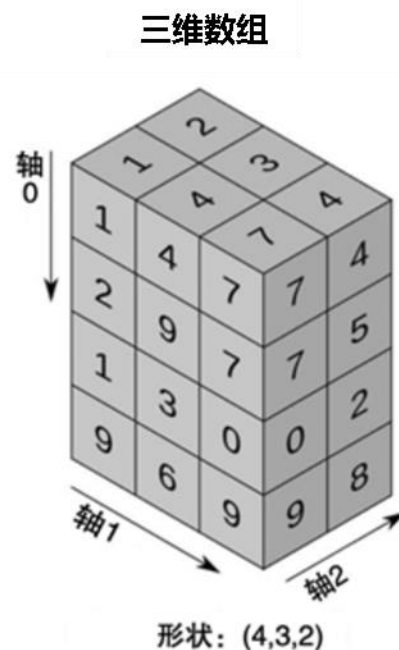
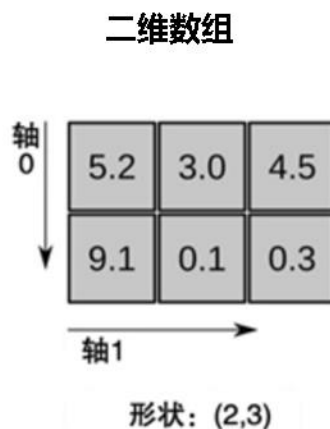
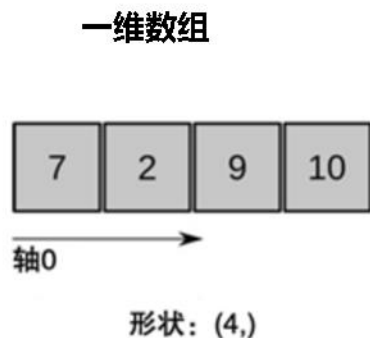
由于数组中各元素具有统一的类型，并且数组元素的下标一般具有固定的上界和下界，因此，数组的处理比其它复杂的结构更为简单。

5.1 数组的定义和运算

一维数组：1个下标， a_i 是 a_{i+1} 的直接前驱

二维数组：2个下标，每个元素 $a_{i,j}$ 受到两个关系（行关系和列关系）的约束

多维数组：n个下标，每个元素受到n个关系约束



5.1 数组的定义和运算

二维数组可以看成是线性表的线性表

$A=(\alpha_0 \ \alpha_1 \ \dots \ \alpha_j \ \dots \ \alpha_{n-1})$, 其中 α_j ($0 \leq j \leq n-1$) 本身也是一个线性表, 称为**列向量**。

$$A = (\alpha_0 \ \alpha_1 \ \dots \ \alpha_j \ \dots \ \alpha_{n-1})$$
$$A_{m \times n} = \begin{pmatrix} \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ a_{00} & a_{01} & \dots & a_{0j} & \dots & a_{0n-1} \\ a_{10} & a_{11} & \dots & a_{1j} & \dots & a_{1n-1} \\ \vdots & \vdots & & & & \\ a_{i0} & a_{i1} & \dots & a_{ij} & \dots & a_{in-1} \\ \vdots & \vdots & & & & \\ A_{m-10} & A_{m-11} & \dots & A_{m-1j} & \dots & A_{m-1n-1} \end{pmatrix}$$

矩阵 $A_{m \times n}$ 看成 n 个列向量的线性表, 即 $\alpha_j = (a_{0j}, a_{1j}, \dots, a_{m-1j})$

5.1 数组的定义和运算

还可以将数组 $A_{m \times n}$ 看成另外一个线性表:

$B=(\beta_0, \beta_1, \dots, \beta_{m-1})$, 其中 β_i ($1 \leq i \leq m-1$) 本身也是一个线性表, 称为行向量, 即: $\beta_i = (a_{i0}, a_{i2}, \dots, a_{ij}, \dots, a_{in-1})$ 。

$$A_{m \times n} = \left(\begin{array}{cccccc} a_{00} & a_{01} & \dots & a_{0j} & \dots & a_{0n-1} \\ a_{10} & a_{11} & \dots & a_{1j} & \dots & a_{1n-1} \\ \vdots & \vdots & & & & \\ a_{i0} & a_{i1} & \dots & a_{ij} & \dots & a_{in-1} \\ \vdots & \vdots & & & & \\ a_{m-10} & a_{m-11} & \dots & a_{m-1j} & \dots & a_{m-1n-1} \end{array} \right) \begin{array}{l} \parallel \\ \leftarrow \beta_0 \\ \leftarrow \beta_1 \\ \leftarrow \vdots \\ \leftarrow \beta_i \\ \leftarrow \vdots \\ \leftarrow \beta_{m-1} \end{array}$$

5.1 数组的定义和运算

数组的抽象数据类型定义 (ADT Array)

数据对象:

$D = \{ a_{j_1 j_2 \dots j_n} \mid n > 0, \text{ 称为数组的维数, } j_i \text{ 是数组的第 } i \text{ 维下标, } 1 \leq j_i \leq b_i, b_i \text{ 为数组第 } i \text{ 维的长度, } a_{j_1 j_2 \dots j_n} \in \text{ElementSet} \}$

数据关系:

$R = \{ R_1, R_2, \dots, R_n \}$

$R_i = \{ \langle a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_{i+1} \dots j_n} \rangle \mid 0 \leq j_k \leq b_k - 1, 1 \leq k \leq n \text{ 且 } k \neq i, \}$

$0 \leq j_i \leq b_i - 2, a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_{i+1} \dots j_n} \in D, i = 1, \dots, n \}$

5.1 数组的定义和运算

基本操作:

(1) **InitArray**(A,n,bound₁,...,bound_n): 若维数n和各维的长度合法, 则构造相应的数组A, 并返回TRUE;

(2) **DestroyArray** (A) : 销毁数组A;

(3) **GetValue** (A, e, index₁, ...,index_n) : 若下标合法, 用e返回数组A中由index₁, ...,index_n所指定的元素的值。

(4) **SetValue** (A, e, index₁, ...,index_n) : 若下标合法, 则将数组A中由index₁, ...,index_n所指定的元素的值置为e。

5.2 数组的顺序存储和实现

对于数组A，一旦给定其维数 n 及各维长度 b_i ($1 \leq i \leq n$)，则该数组中元素的个数是固定的，不可以对数组做插入和删除操作，不涉及移动元素操作，因此对于数组而言，采用顺序存储法比较合适。

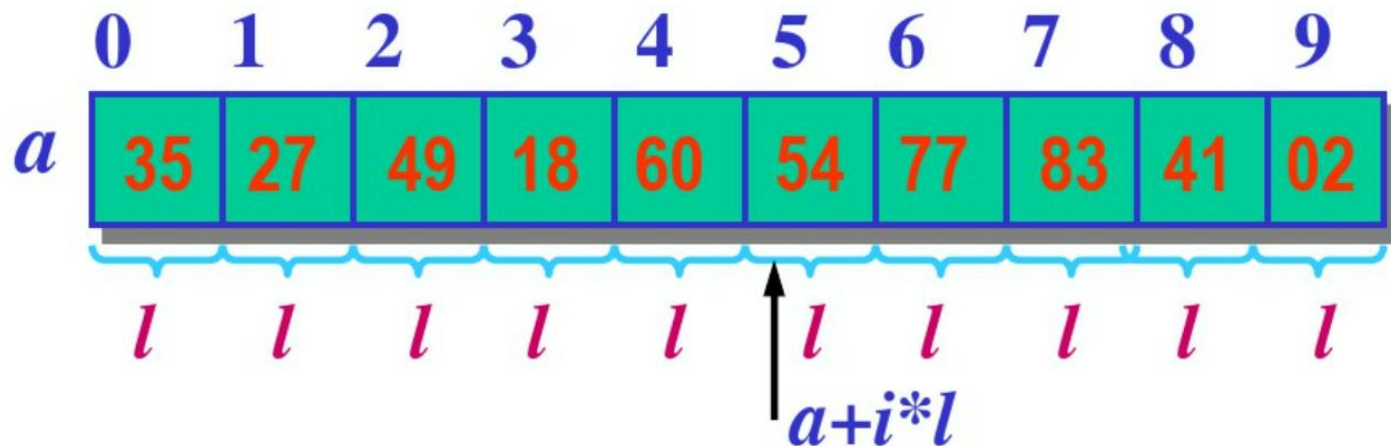
数组是多维的结构，而存储空间是一个一维的结构

- ① 开始结点的存放地址（即基地址）
- ② 维数和每维的上、下界；
- ③ 每个数组元素所占用的单元数

5.2 数组的顺序存储和实现

一维数组顺序存储

$$\text{LOC}(i) = \begin{cases} a, & i = 0 \\ \text{LOC}(i-1) + l = a + i * l, & i > 0 \end{cases}$$



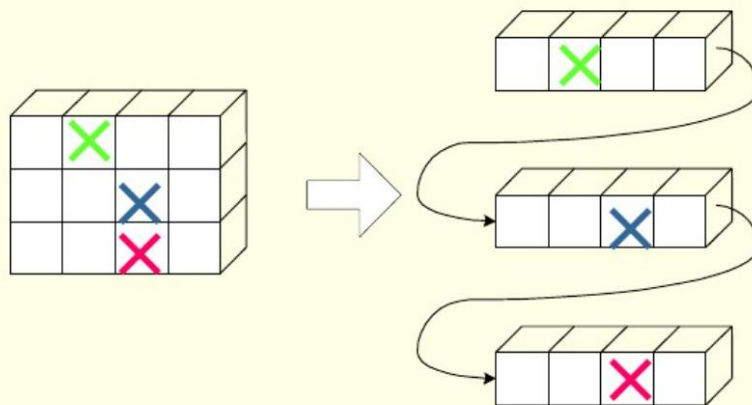
$$\text{LOC}(i) = \text{LOC}(i-1) + l = a + i * l$$

5.2 数组的顺序存储和实现

二维数组顺序存储——行存储

$$A_{m \times n} = \begin{bmatrix} [a_{11} & a_{12} & \cdots & a_{1n}] \\ [a_{21} & a_{22} & \cdots & a_{2n}] \\ \vdots & \vdots & & \vdots \\ [a_{m1} & a_{m2} & \cdots & a_{mn}] \end{bmatrix}$$

以行序为主序
(低下标优先)



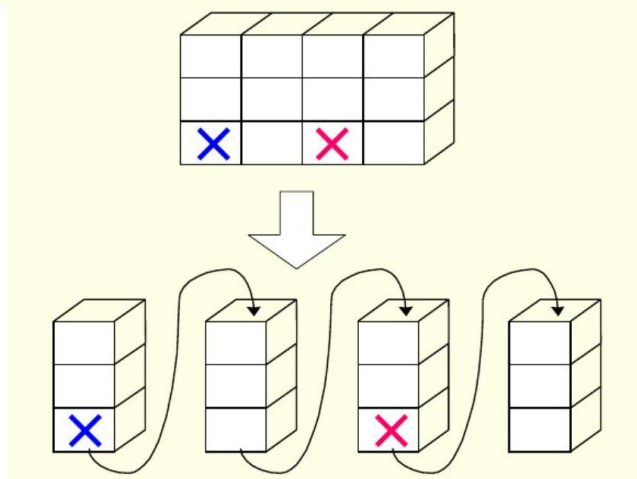
$a_{11}, a_{12}, \dots, a_{1n}, a_{21}, a_{22}, \dots, a_{2n}, \dots, a_{m1}, a_{m2}, \dots, a_{mn}$

5.2 数组的顺序存储和实现

二维数组顺序存储——列存储

$$A_{m \times n} = \begin{bmatrix} \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} & \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} & \cdots & \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} \end{bmatrix}$$

以列序为主序
(高下标优先)



$a_{11}, a_{21}, \dots, a_{m1}, a_{12}, a_{22}, \dots, a_{m2}, \dots, a_{1n}, a_{2n}, \dots, a_{mn}$

5.2 数组的顺序存储和实现

二维数组顺序存储

$a[n][m]$

$$a = \begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][m-1] \\ a[1][0] & a[1][1] & \cdots & a[1][m-1] \\ a[2][0] & a[2][1] & \cdots & a[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[n-1][0] & a[n-1][1] & \cdots & a[n-1][m-1] \end{pmatrix}$$

设数组开始存放位置 $\text{LOC}(0, 0) = a$

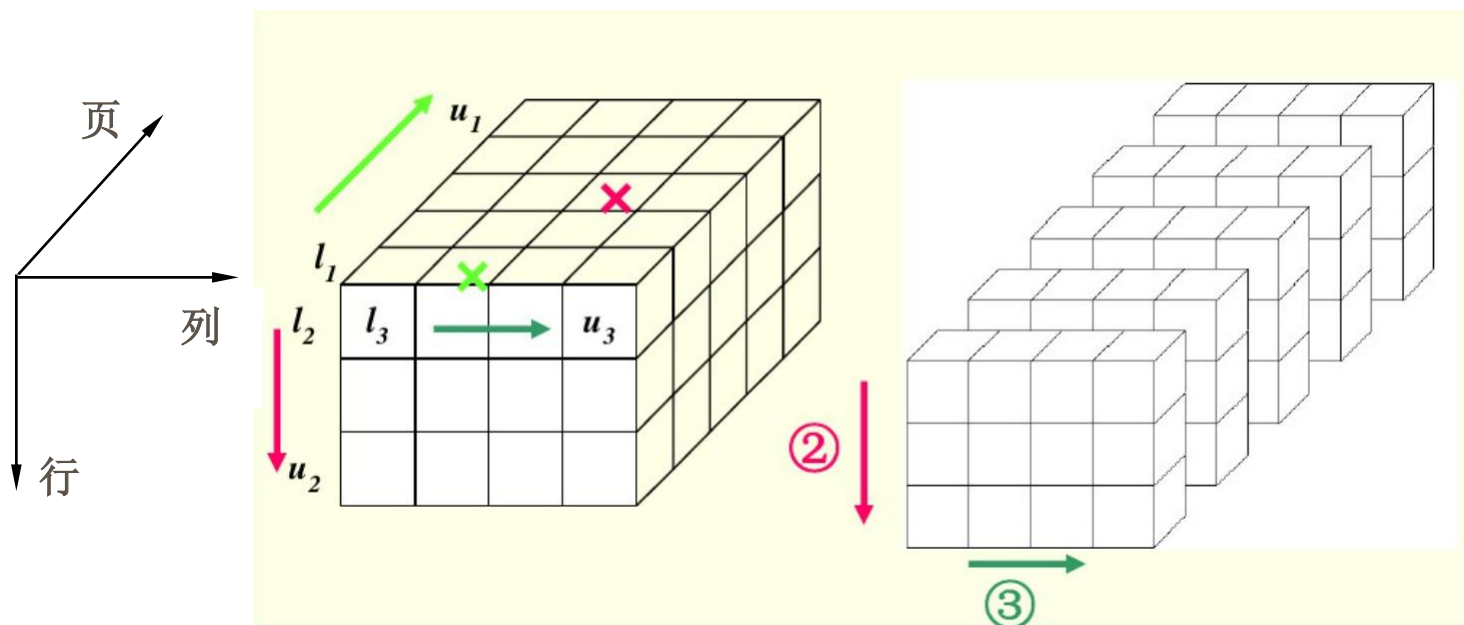
$$\text{Loc}(i, j) = \text{Loc}(0, 0) + (b_2 \times i + j)L$$

↑ ↑
地址 i维长度

5.2 数组的顺序存储和实现

三维数组顺序存储

假设有一个 $3 \times 4 \times 2$ 的三维数组A，其逻辑结构图为：



按页/行/列存放，页优先的顺序存储

5.2 数组的顺序存储和实现

三维数组顺序存储

👉 $a[m_1][m_2][m_3]$ 各维元素个数为 m_1, m_2, m_3

👉 下标为 i_1, i_2, i_3 的数组元素的存储位置:

$$\text{LOC}(i_1, i_2, i_3) = a + \underbrace{i_1 * m_2 * m_3}_{\text{前 } i_1 \text{ 页总元素个数}} + \underbrace{i_2 * m_3}_{\text{第 } i_1 \text{ 页的前 } i_2 \text{ 行总元素个数}} + i_3$$

第 i_2 行前 i_3 列
元素个数

5.2 数组的顺序存储和实现

n维数组顺序存储

$$A_{i_1 \dots i_n} \text{的起始地址} = \text{第一个元素的起始地址} + \text{该元素前面的元素个数} \times \text{单位长度}$$

$$LOC(j_1, j_2, \dots, j_n) = LOC(0, 0, \dots, 0) + \sum_{i=1}^n c_i j_i$$

其中 $c_n = L$, $c_{i-1} = b_i \times c_i$, $1 < i \leq n$ 。

称为 n 维数组的映象函数。数组元素的存储位置是其下标的线性函数。

5.2 数组的顺序存储和实现

n维数组顺序存储

```
#define MAX_ARRAY_DIM 8 //假设最大维数为8
```

```
typedef struct{  
    ELemType *base; //数组元素基址  
    int dim; //数组维数  
    int *bound; //数组各维长度信息保存区基址  
    int *constants; //数组映像函数常量的基址
```

```
}Array;
```

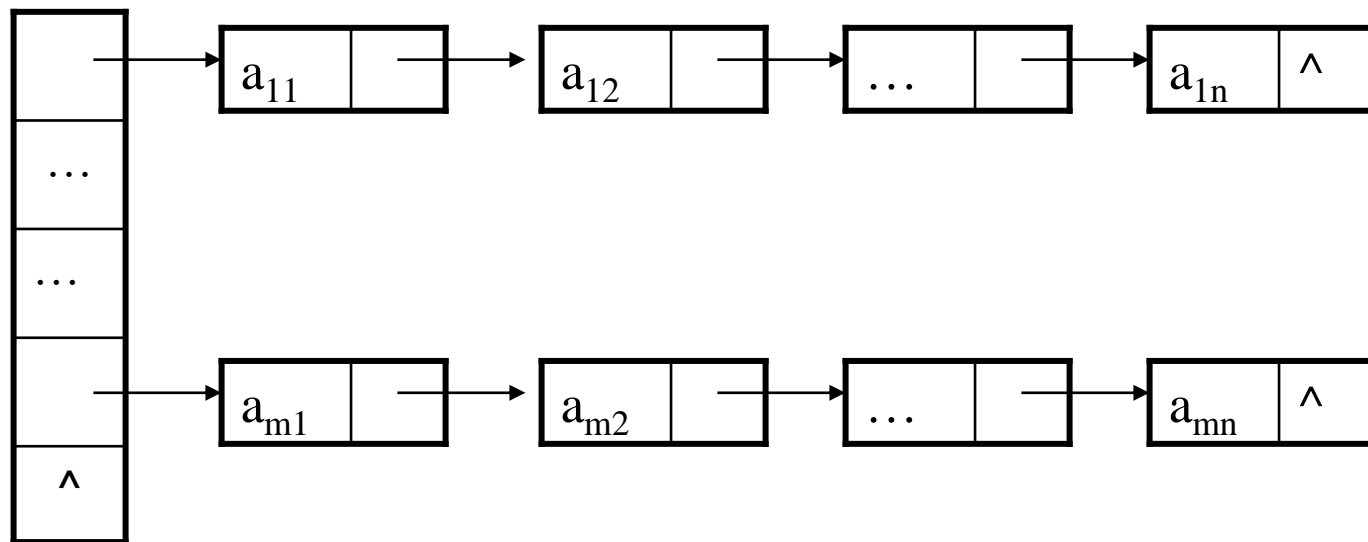
即 C_i 信息保存区

数组的基本操作函数说明（有5个）

5.2 数组的顺序存储和实现

补充：数组的链式存储方式——用带行指针向量的单链表来表示。

行指针向量



5.3 矩阵的压缩存储

- 矩阵一般可用二维数组实现，特殊矩阵采用压缩存。
- 压缩存储：为多个值相同的元只分配一个存储空间，对零元不分配空间。
- 特殊矩阵：值相同的元素或者零元素在矩阵中的分布有一定规律
- 稀疏矩阵：非零元较零元少，且分布没有一定规律的矩阵

什么样的矩阵具备压缩条件？

一些特殊矩阵，如：对称矩阵，对角矩阵，三角矩阵，稀疏矩阵等。

矩阵中非零元素的个数较少（一般小于5%）

5.3.1 特殊矩阵

对称矩阵

- 设有一个 $n \times n$ 的对称矩阵 A 。

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

- 对称矩阵中的元素关于主对角线对称, $a_{ij} = a_{ji}$, $0 \leq i, j \leq n-1$

5.3.1 特殊矩阵

- 为节约存储，只存对角线及对角线以上的元素，或者只存对角线或对角线以下的元素。前者称为**上三角矩阵**，后者称为**下三角矩阵**。

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

下三角矩阵

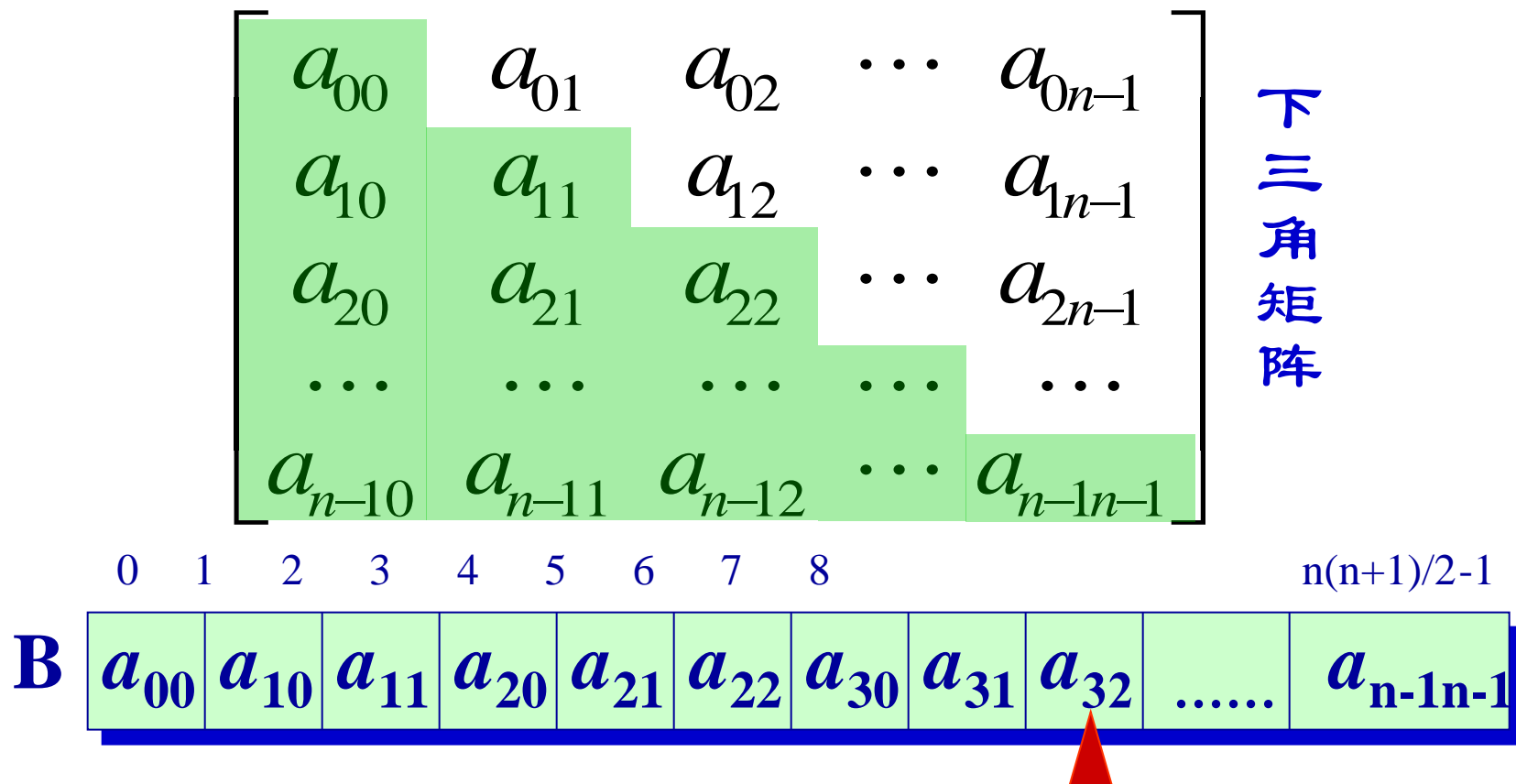
5.3.1 特殊矩阵

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

上
三
角
矩
阵

- 把它们按行存放于一个一维数组 B 中，称之为对称矩阵 A 的压缩存储方式。
- 数组 B 共有 $n + (n - 1) + \cdots + 1 = n*(n+1) / 2$ 个元素。

5.3.1 特殊矩阵



5.3.1 特殊矩阵

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1 & \text{当 } i \geq j \\ \frac{j(j-1)}{2} + i - 1 & \text{当 } i < j \end{cases} \quad \begin{matrix} 1 \leq i, j \leq n \\ \text{矩阵从1,1开始编号} \end{matrix}$$

若 $i \geq j$, 数组元素 $A[i][j]$ 在数组 B 中的存放位置为:

$$\underbrace{0 + 1 + \cdots + i - 1}_{\text{前 } i \text{ 行 (0~i-1) 元素总数}} + \underbrace{j - 1}_{\text{第 } i \text{ 行第 } j \text{ 个元素前元素个数}} = \frac{(i-1) * i}{2} + j - 1$$

前 i 行 (0~ $i-1$) 元素总数 + 第 i 行第 j 个元素前元素个数


若 $i < j$, 数组元素 $A[i][j]$ 在矩阵的上三角部分, 在数组 B 中没有存放, 可以找它的对称元素:

$$A[j][i] = j * (j - 1) / 2 + i - 1$$

5.3.1 特殊矩阵

带状矩阵

$$A = \begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & a_{n-2n-3} & a_{n-2n-2} & a_{n-2n-1} \\ 0 & 0 & 0 & 0 & a_{n-1n-2} & a_{n-1n-1} \end{bmatrix}$$

$$B \quad \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \hline a_{00} & a_{01} & a_{10} & a_{11} & a_{12} & a_{21} & a_{22} & a_{23} & \dots & a_{n-1n-2} & a_{n-1n-1} \\ \hline \end{array}$$


5.3.2 稀疏矩阵

稀疏矩阵：指矩阵中大多数元素为零的矩阵。

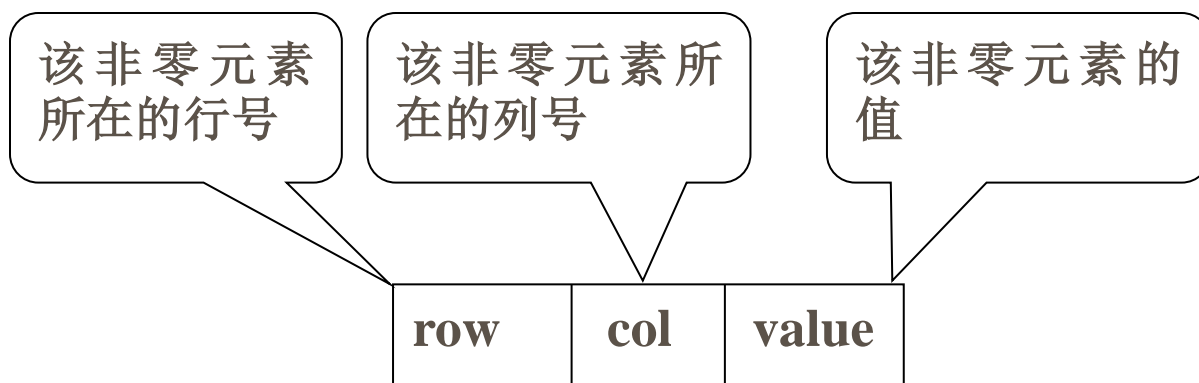
$$\mathbf{M}_{6 \times 7} = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix} \quad \mathbf{N}_{6 \times 7} = \begin{pmatrix} 0 & 0 & 3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

- 设矩阵 A 中有 t 个非零元素。令 $e = t/(m*n)$, 称 e 为矩阵的稀疏因子。
- 一般认为 $e \leq 0.05$ 时称之为稀疏矩阵。

1. 三元组表表示法

对于稀疏矩阵的压缩存储要求在存储非零元素的同时，还必须存储该非零元素在矩阵中所处的行号和列号。我们将这种存储方法叫做稀疏矩阵的三元组表示法。

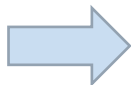
每个非零元素在一维数组中的表示形式如图所示：



1. 三元组表表示法

- 三元组顺序表又称**有序的双下标法**，它的特点是，非零元在表中按行序有序存储，因此**便于进行依行顺序处理的矩阵运算**。然而，若需随机存取某一行中的非零元，则需从头开始进行查找。

$$\begin{bmatrix} 0 & 2 & 5 \\ 0 & 3 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

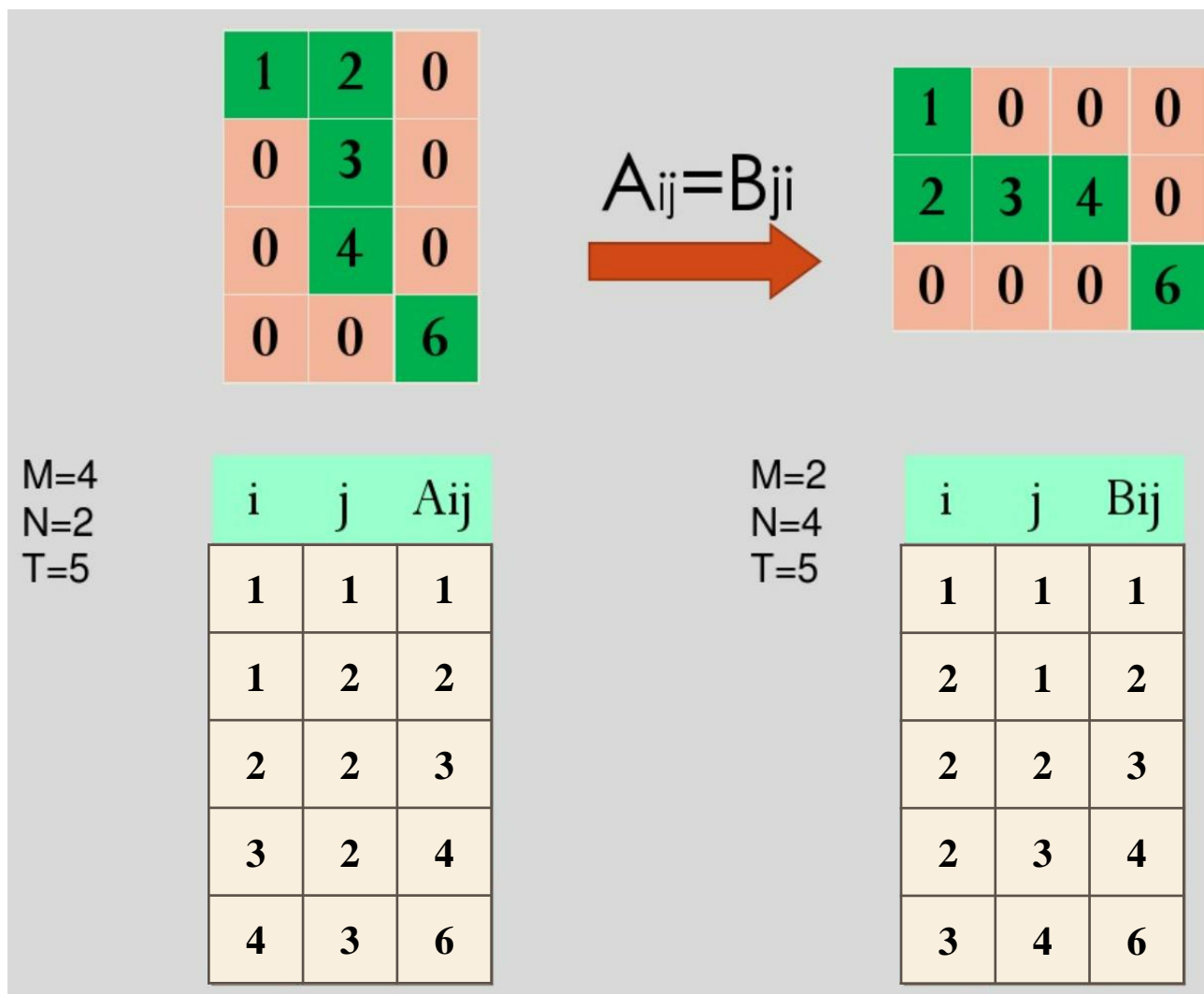


i	j	A _{ij}
1	2	2
1	3	5
2	2	3
3	2	4
4	3	6

1. 三元组表表示法

```
#define MAXSIZE 12500 /*非零元素的个数最多为12500*/  
typedef struct  
{  
    int i, j; /*该非零元素的行下标和列下标*/  
    ElemType e; /*该非零元素的值*/  
}Triple;  
typedef struct  
{  
    Triple data[MAXSIZE+1]; /* 非零元素的三元组表 。 data[0]未用*/  
    int mu,nu,tu; /*矩阵的行数、列数和非零元素的个数*/  
}TSMatrix;
```

2. 矩阵转置



2. 矩阵转置

问题：b.data是一个按列优先顺序存储的稀疏矩阵B

解决方法：重新排列B中三元组的顺序。

0	2	5
0	3	0
0	4	0
0	0	6

$$A_{ij} = B_{ji}$$

0	0	0	0
2	3	4	0
5	0	0	6

M=4
N=2
T=5

i	j	A _{ij}
1	2	2
1	3	5
2	2	3
3	2	4
4	3	6

$i \leftrightarrow j$

i	j	B _{ij}
2	1	2
3	1	5
2	2	3
2	3	4
3	4	6

M=2
N=4
T=5

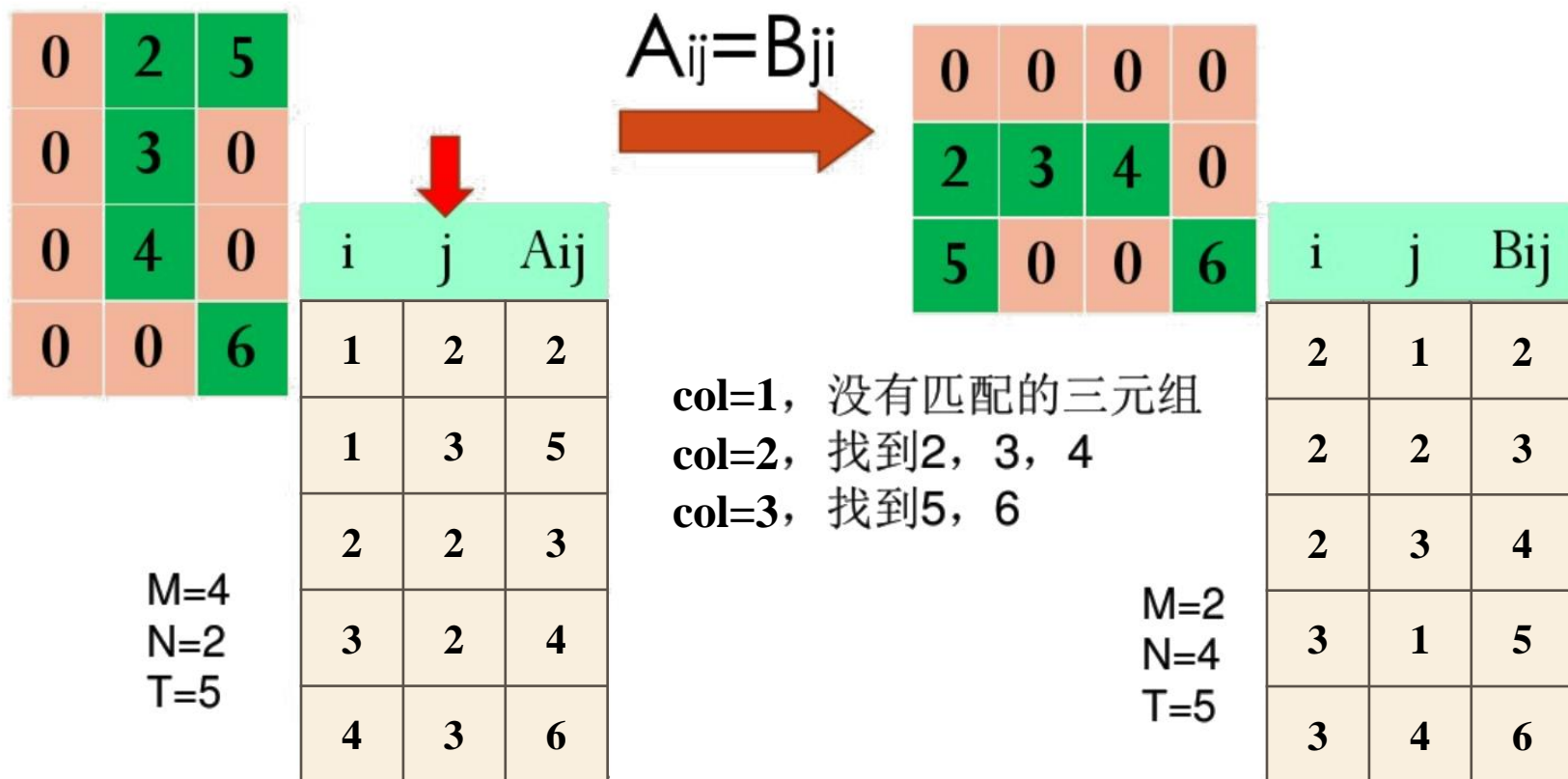
按*i*
排序

i	j	B _{ij}
2	1	2
2	2	3
2	3	4
3	1	5
3	4	6

2. 矩阵转置

B的行优先即A的列优先

对A的第col列，扫描三元组表a.data地，找出所有列号等于col的三元组，将它们的行号和列号互换后依次放入b.data中，即可得到B的按行优先的压缩存储表示



2. 矩阵转置 (5.1)

```
Status TransposeSMatrix(TSMatrix M, TSMatrix &T) {  
    int q,col,p;  
    T.mu=M.nu; T.nu=M.mu; T.tu=M.tu;  
    if (T.tu) {  
        q=1;  
        for (col=1;col<=T.mu;++col)  
            for(p=1;p<=M.tu;++p)  
                if ( M.data[p].j==col ) {  
                    T.data[q].i=M.data[p].j;  
                    T.data[q].j=M.data[p].i;  
                    T.data[q].e=M.data[p].e;  
                    ++q;  
                }  
    }  
    return OK;  
}
```

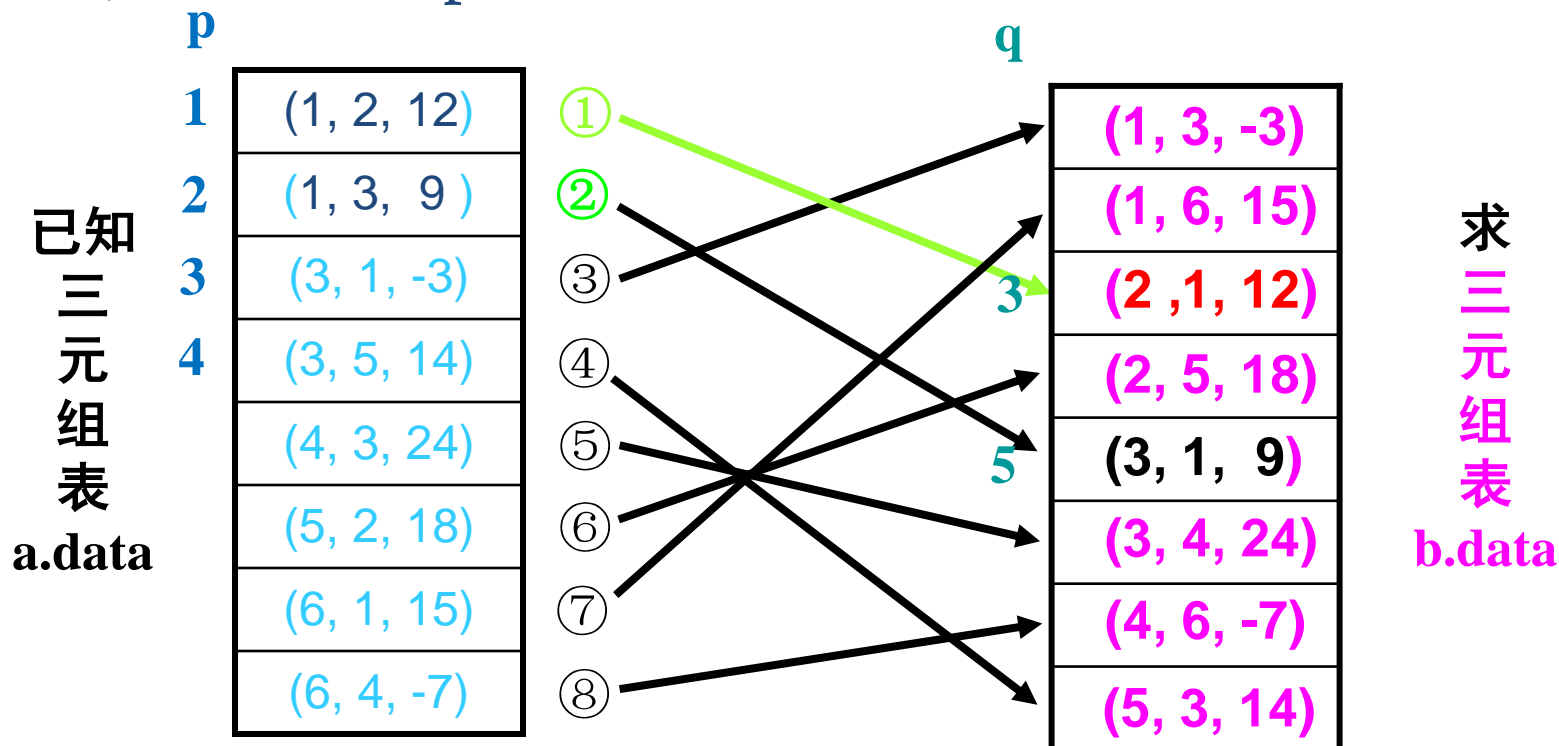
时间复杂度为 **$O(nu*tu)$**
与矩阵的列数和非零元
的个数的乘积成正比

例：若矩阵有200行，200列，100个非零元素，总共有20000次处理。

适用于 $Tu \leq mu*nu$

3、快速转置算法

思路：**依次**把a.data中的元素直接送入b.data的恰当位置上
(即M三元组的p指针(下标)不回溯)。



关键：怎样寻找b.data的“恰当”位置？

转置之前每列第一个非0元素的序号等于转置之后在以行优先存放的三元组序号。

3、快速转置算法

□ 基本思想：在A中按行序找三元组，确定该三元组在B中的位置，写入B

□ 关键问题：如何确定每个三元组在B中的位置

A中三元组在B的中位置=

每列的第一个非零元素在数组B中应有的位置

+

每一列非零元素的个数

3、快速转置算法

思路：为实现转置运算，应当按列生成 M 矩阵三元组表的**两个辅助向量**，让它携带每列的非零元素个数 **NUM[i]** 以及每列的第一个非零元素在三元组表中的位置**POS[i]** 等信息。

(1, 2, 12)
(1, 3, 9)
(3, 1, -3)
(3, 5, 14)
(4, 3, 24)
(5, 2, 18)
(6, 1, 15)
(6, 4, -7)

(3, 1, -3)
(6, 1, 15)
(1, 2, 12)
(5, 2, 18)
(1, 3, 9)
(4, 3, 24)
(6, 4, -7)
(3, 5, 14)

辅助向量的
样式：

i	1	2	3	4	5	6
NUM[i]	2	2	2	1	1	0
POS[i]	1	3	5	7	8	9

计算式： $POS[1]=1$
 $POS[i]=POS[i-1]+NUM[i-1]$

(1, 3, -3)
(1, 6, 15)
(2, 1, 12)
(2, 5, 18)
(3, 1, 9)
(3, 4, 24)
(4, 6, -7)
(5, 3, 14)

3、快速转置算法（5.2）

```
Status FastTransposeSMatrix(TSMatrix M,TSMatrix &T)
{ T.mu=M.nu; T.nu=M.mu; T.tu=M.tu;
  if (T.tu){
    for (col=1;col<=M.nu;++col) num[col]=0;
    for (t=1;t<=M.tu;++t) ++num[M.data[t].j]; //统计非0元素
    cpot[1]=1;
    //由递推关系计算cpot
    for(col=2;col<=M.nu;++col)cpot[col]=cpot[col-1]+num[col-1];
    for (p=1;p<=M.Tu;++p) {
      col=M.data[p].j; q=cpot[col];
      T.data[q].i=M.data[p].j;
      T.data[q].j=M.data[p].i;
      T.data[q].e=M.data[p].e;
      ++cpot[col]; }
  }
  return OK;
}
```

重要！修改向量内容（列坐标加1），
预备给同列的下一非零元素定位之用

3、快速转置算法（5.2）

算法5.2：算法分析

1. 与常规算法相比，附加了生成辅助向量表的工作。增开了2个长度为列长的数组(**num[]**和**cpos[]**)。
2. 从时间上，此算法用了4个并列的单循环，而且其中前3个单循环都是用来产生辅助向量表的。

for(col = 1; col <= M.nu; ++col) { }; 循环次数 = nu (列数);

for(t = 1; t <= M.tu; ++t) { }; 循环次数 = tu (非0元素个数);

for(col = 2; col <= M.nu; ++col) { }; 循环次数 = nu;

for(p = 1; p <= M.tu; ++p) { }; 循环次数 = tu;

该算法的时间复杂度 = $nu + tu + nu + tu = O(nu + tu)$

3、快速转置算法（5.2）

算法5.2：算法分析

讨论：最恶劣情况是矩阵中全为非零元素，此时 $t_u = n_u * \mu_u$ ，而此时的时间复杂度也只是 $O(\mu_u * n_u)$ ，并未超过传统转置算法的时间复杂度。

小结： 传统转置： $O(\mu_u * n_u)$ 压缩转置： $O(n_u * t_u)$

压缩快速转置： $O(n_u + t_u)$

增设辅助向量，牺牲空间效率换取时间效率。

5.3.2 稀疏矩阵

存储和实现

□ 三元组顺序表

```
#define MAXSIZE 12500 /*非零元素的个数最多为12500*/
```

```
typedef struct
```

```
{
```

```
    int i, j; /*该非零元素的行下标和列下标*/
```

```
    ElemType e; /*该非零元素的值*/
```

```
}Triple;
```

```
typedef struct
```

```
{
```

```
    Triple data[MAXSIZE+1]; /* 非零元素的三元组表 。data[0]未用*/
```

```
    int mu,nu,tu; /*矩阵的行数、列数和非零元素的个数*/
```

```
}TSMatrix;
```

row	col	value
i	j	e

□ 行逻辑链接的顺序表

□ 十字链表

5.3.2 稀疏矩阵

□ 行逻辑链接的顺序表

0	12	9	0	0	(1, 2, 12)
0	0	0	0	0	(1, 3, 9)
-3	0	0	0	14	(3, 1, -3)
0	0	24	0	0	(3, 5, 14)
0	18	0	0	0	(4, 3, 24)
15	0	0	-7	0	(5, 2, 18)
					(6, 1, 15)
					(6, 4, -7)

需知道每一行的第一个非零元在三元组表中的位置

i	1	2	3	4	5	6
NUM[i]	2	2	2	1	1	0
POS[i]	1	3	5	7	8	9

```
#define MAXSIZE 12500
#define MAXMN 500
typedef struct
{
    int i, j;
    ElemType e
}Triple;
typedef struct {
    Triple data[MAXSIZE + 1];
    int rpos[MAXMN + 1];
    int mu, nu, tu;
} RLSMatrix;
```


4、行逻辑链接的顺序表

例：给定一组下标，求矩阵的元素值

```
ElemType value(RLSMatrix M, int r, int c) {  
    p = M.rpos[r];  
    while (M.data[p].i==r && M.data[p].j < c)  
        p++;  
    if (M.data[p].i==r && M.data[p].j==c)  
        return M.data[p].e;  
    else return 0;  
} // value
```

4、行逻辑链接的顺序表

设矩阵M是 $m_1 \times n_1$ 矩阵，N是 $m_2 \times n_2$ 矩阵；若可以相乘，则必须满足矩阵M的列数 n_1 与矩阵N的行数 m_2 相等，才能得到结果矩阵 $Q=M \times N$ （一个 $m_1 \times n_2$ 的矩阵）。

数学中矩阵Q中的元素的计算方法如下：

$$Q[i][j] = \sum_{k=1}^{n1} M[i][k] \times N[k][j] \quad \text{其中：} 1 \leq i \leq m_1, 1 \leq j \leq n_2$$

$$M = \begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix} \quad Q = \begin{bmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{bmatrix}$$

4、行逻辑链接的顺序表

根据数学上矩阵相乘的原理，我们可以得到矩阵相乘的经典算法：

```
for(i=1;i<=m1;i++)
```

```
    for(j=1;j<=n2;j++)
```

```
        { Q[i][j]=0;
```

```
            for(k=1;k<=n1;k++)
```

```
                Q[i][j]= Q[i][j]+M[i][k]*N[k][j];
```

```
        }
```

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

$$\mathbf{N} = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix}$$

$$\mathbf{Q} = \begin{bmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{bmatrix}$$

4、行逻辑链接的顺序表

□ 算法分析

- 不论 $M[i][k]$, $N[k][j]$ 是否为零, 都要进行一次乘法运算

其时间复杂度为: $O(m1 \times n2 \times n1)$

□ 改进思想

采用三元组表示法, 只对非零元素做存储, 所以可以采用固定三元组a中元素 (i, k, M_{ik}) ($1 \leq i \leq m1, 1 \leq k \leq n1$), 在三元组b中找所有行号为k的的对应元素 (k, j, N_{kj}) ($1 \leq k \leq m2, 1 \leq j \leq n2$) 进行相乘、累加从而得到 $Q[i][j]$ 。

即: 以三元组a中的元素为基准, 依次求出其与三元组b的有效乘积。

4、行逻辑链接的顺序表

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

i	j	e
1	1	3
1	4	5
2	2	-1
3	1	2

$$\mathbf{N} = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix}$$

i	j	e
1	2	2
2	1	1
3	1	-2
3	2	4

$$\mathbf{Q} = \begin{bmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{bmatrix}$$

i	j	e
1	2	6
2	1	-1
3	2	4

row	1	2	3	4
num[row]	1	1	2	0
rpot[row]	1	2	3	5

$\text{rpot}(1)=1$
 $\text{rpot}[\text{row}] = \text{rpot}[\text{row} - 1] + \text{num}[\text{row} - 1]$

4、行逻辑链接的顺序表

□ 算法思想

相乘基本操作：对于三元组a中每个元素a.data[p]（p=1, 2, 3, ..., a.len），找出三元组b中所有满足条件a.data[p].col=b.data[q].row的元素b.data[q]，求得a.data[p].e与b.data[q].e的乘积，而这个乘积作为Q[i, j]的一部分，应对每个元素设一个累计和变量，其初值为0。扫描完三元组a，求得相应元素的乘积并累加到适当的累计和的变量上。

4、行逻辑链接的顺序表

□ 算法思想

Q初始化;

if Q是非零矩阵 { // 逐行求积

for (arow=1; arow<=M.mu; ++arow) { // 处理M的每一行

ctemp[] = 0; // 累加器清零

计算Q中第arow行的积并存入ctemp[] 中;

将ctemp[] 中非零元压缩存储到Q.data;

} // for arow

} // if

4、行逻辑链接的顺序表

```
int MulSMatrix(TriSparMatrix M, TriSparMatrix N, TriSparMatrix &Q){
    int arow, brow, p; int ctemp[MAXSIZE];
    if (M.nu != N.mu) return ERROR; /*返回FALSE表示求矩阵乘积失败*/
    Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0;
    if (M.tu * N.tu != 0) {
        for (arow = 1; arow <= M.mu; arow++){ /*逐行处理M*/
            for (p = 1; p <= M.nu; p++) ctemp[p] = 0; /*当前行各元素的累加器清零*/
            Q.rpos[arow] = Q.tu + 1;
            for (p=M.rpos[arow]; p<M.rpos[arow+1]; p++){ //p指向M当前行中每一个非零元素
                brow = M.data[p].col; /* M中的列号应与N中的行号相等*/
                if (brow < N.nu) t = N.rpos[brow + 1];
                else t = N.tu + 1;
                for (q = N.rpos[brow]; q < t; q++){
                    ccol = N.data[q].col; /*乘积元素在Q中列号*/
                    ctemp[ccol] += M.data[p].e * N.data[q].e;
                } /* for q */
            } /*求得Q中第crow行的非零元*/
            for (ccol = 1; ccol < Q.nu; ccol++) /*压缩存储该非零元*/
                if (ctemp[ccol]) {
                    if (++Q.tu > MAXSIZE) return ERROR;
                    Q.data[Q.tu] = {arow, ccol, ctemp[ccol]};
                } /* if */
            } /* for arow */
        } /*if*/
    }
    return OK; /*返回OK表示求矩阵乘积成功*/
}
```


4、行逻辑链接的顺序表

□ 算法分析

累加器c_{temp}初始化的时间复杂度为 $O(M.mu \times N.nu)$,
求Q的所有非零元的时间复杂度为 $O(M.tu \times N.tu / N.mu)$,
进行压缩存储的时间复杂度为 $O(M.mu \times N.nu)$,
总的时间复杂度就是 $O(M.mu \times N.nu + M.tu \times N.tu / N.mu)$ 。

若M是m行n列的稀疏矩阵, N是n行p列的稀疏矩阵,
则M中非零元的个数 $M.tu = \delta_M \times m \times n$,

N中非零元的个数 $N.tu = \delta_N \times n \times p$,

相乘算法的时间复杂度就是 $O(m \times p \times (1 + n\delta_M\delta_N))$,

当 $\delta_M < 0.05$ 和 $\delta_N < 0.05$ 及 $n < 1000$ 时,

相乘算法的时间复杂度就相当于 $O(m \times p)$ 。

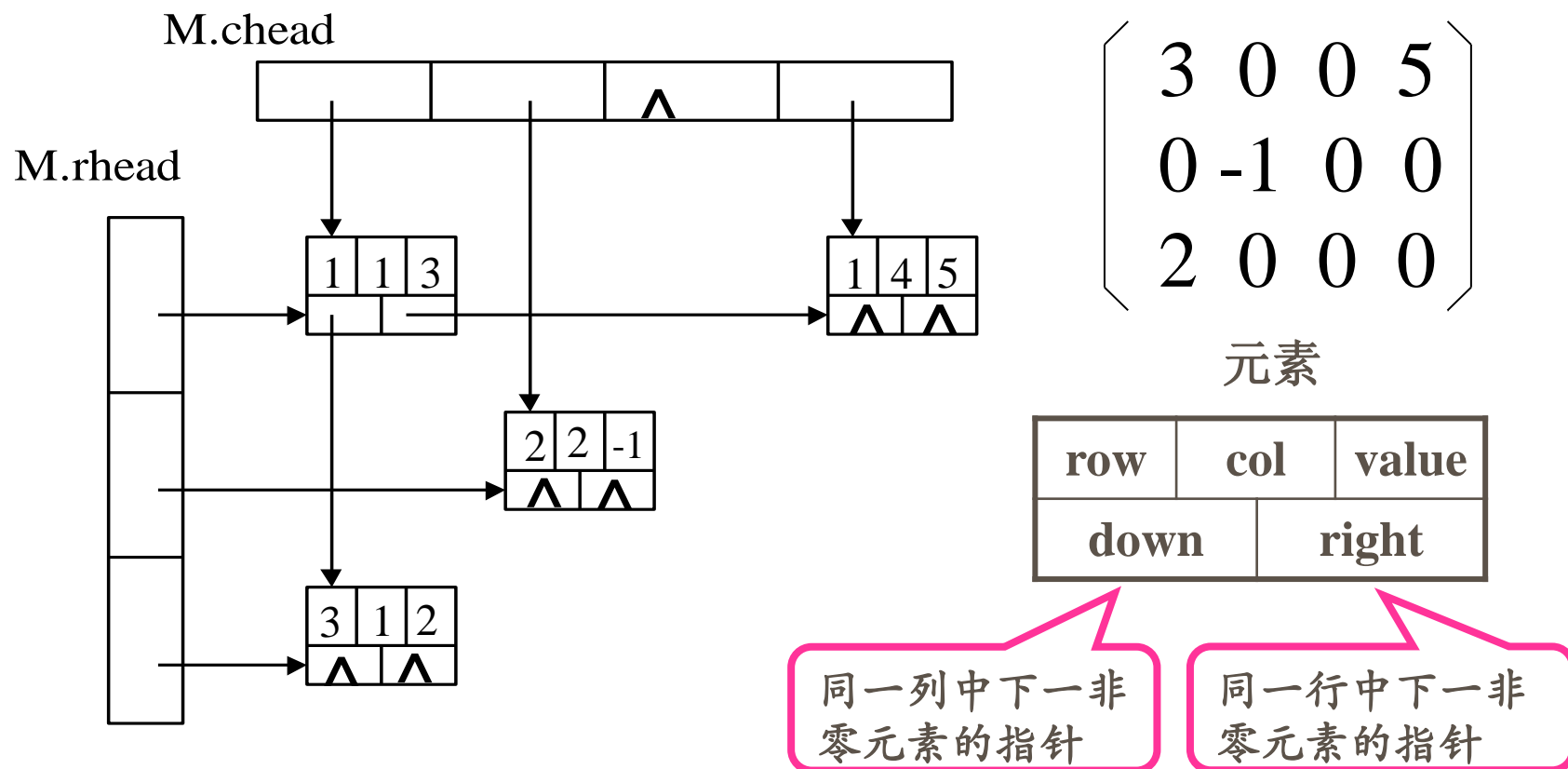
5. 稀疏矩阵的链式存储结构：十字链表

两个稀疏矩阵相乘的结果不一定是稀疏矩阵。反之，相乘的每个分量 $M[i, k] \times N[k, j]$ 不为零，但累加的结果 $Q[i, j]$ 可能是零。例如：

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

当矩阵中非零元素的个数和位置经过运算后变化较大时，就不宜采用顺序存储结构，而应采用链式存储结构来表示三元组。

5. 稀疏矩阵的链式存储结构：十字链表



优点: 它能够灵活地插入因运算而产生的新的非零元素，删除因运算而产生的新的零元素，实现矩阵的各种运算。

5. 稀疏矩阵的链式存储结构：十字链表

```
typedef struct OLNode{ //元素结点
    int i,j; //非零元的行和列下标
    ElemType e;
    struct OLNode *right,*down;
    //该非零元所在行表和列表的后继链域
}OLNode, *OLink;
typedef struct {
    OLink *rhead,*thead; //行和列链表头指针数组
    int mu,nu,tu;
}CrossList;
```

5. 稀疏矩阵的链式存储结构：十字链表

- 需要辅助结点作链表的表头，同时每个结点要增加两个指针域，所以只有在矩阵较大和较稀疏时才能起到节省空间的效果。
- 分别用两个一维数组存储行链表的头指针和列链表的头指针，可加快访问速度。

5. 稀疏矩阵的链式存储结构：十字链表

```
CreateSMatrix_OL(CrossList *M){/*采用十字链表存储结构，创建稀疏矩阵M*/
    if (M != NULL) free(M);
    scanf(&m, &n, &t); /*输入M的行数,列数和非零元素的个数*/
    M->m = m; M->n = n; M->len = t;
    If(!(M->row_head = (OLink *)malloc((m + 1)sizeof(OLink)))) exit(OVERFLOW);
    If(!(M->col_head = (OLink *)malloc((n + 1)sizeof(OLink)))) exit(OVERFLOW);
    M->row_head[] = M->col_head[] = NULL;
    for (scanf(&i, &j, &e); i != 0; scanf(&i, &j, &e)){
        if (!(p = (OLNode *)malloc(sizeof(OLNode))))exit(OVERFLOW);
        p->row = i; p->col = j; p->value = e; /*生成结点*/
        if (M->row_head[i] == NULL) M->row_head[i] = p;
        else{ /*寻找行表中的插入位置*/
            for (q = M->row_head[i]; q->right && q->right->col < j; q = q->right)
                p->right = q->right;
            q->right = p; /*完成插入*/
        }
        if (M->col_head[j] == NULL) M->col_head[j] = p;
        else{ /*寻找列表中的插入位置*/
            for (q = M->col_head[j]; q->down && q->down->row < i; q = q->down)
                p->down = q->down;
            q->down = p; /*完成插入*/
        }
    }
}
```

5. 稀疏矩阵的链式存储结构：十字链表

□ 矩阵相加

$$\begin{array}{ccc} \begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} & + & \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix} \\ A & B & A' \end{array}$$

- ① 若 $pa == NULL$ 或 $pa \rightarrow j > pb \rightarrow j$, 插入一个值为 b_{ij} , 需改变同一行中前一结点的`right` 域值, 以及同一列中前一结点的`down` 域值。
- ② 若 $pa \rightarrow j < pb \rightarrow j$, 则只要将`pa` 指针往右推进一步。
- ③ 若 $pa \rightarrow j == pb \rightarrow j$ 且 $pa \rightarrow e + pb \rightarrow e \neq 0$, 将 $a_{ij} + b_{ij}$ 的值送到`pa` 所指结点的`e` 域。
- ④ 若 $pa \rightarrow j == pb \rightarrow j$ 且 $pa \rightarrow e + pb \rightarrow e == 0$, 在A 矩阵的链表中删除`pa` 所指的结点。 , 需改变同一行中前一结点的`right` 域值, 以及同一列中前一结点的`down` 域值。

5. 稀疏矩阵的链式存储结构：十字链表

□ 元素节点

row	col	value
down		right

□ 由于行、列表头结点互相不冲突，所以可以合并起来：

row=0	col=0	next
down		right

□ 总表头结点：

行数

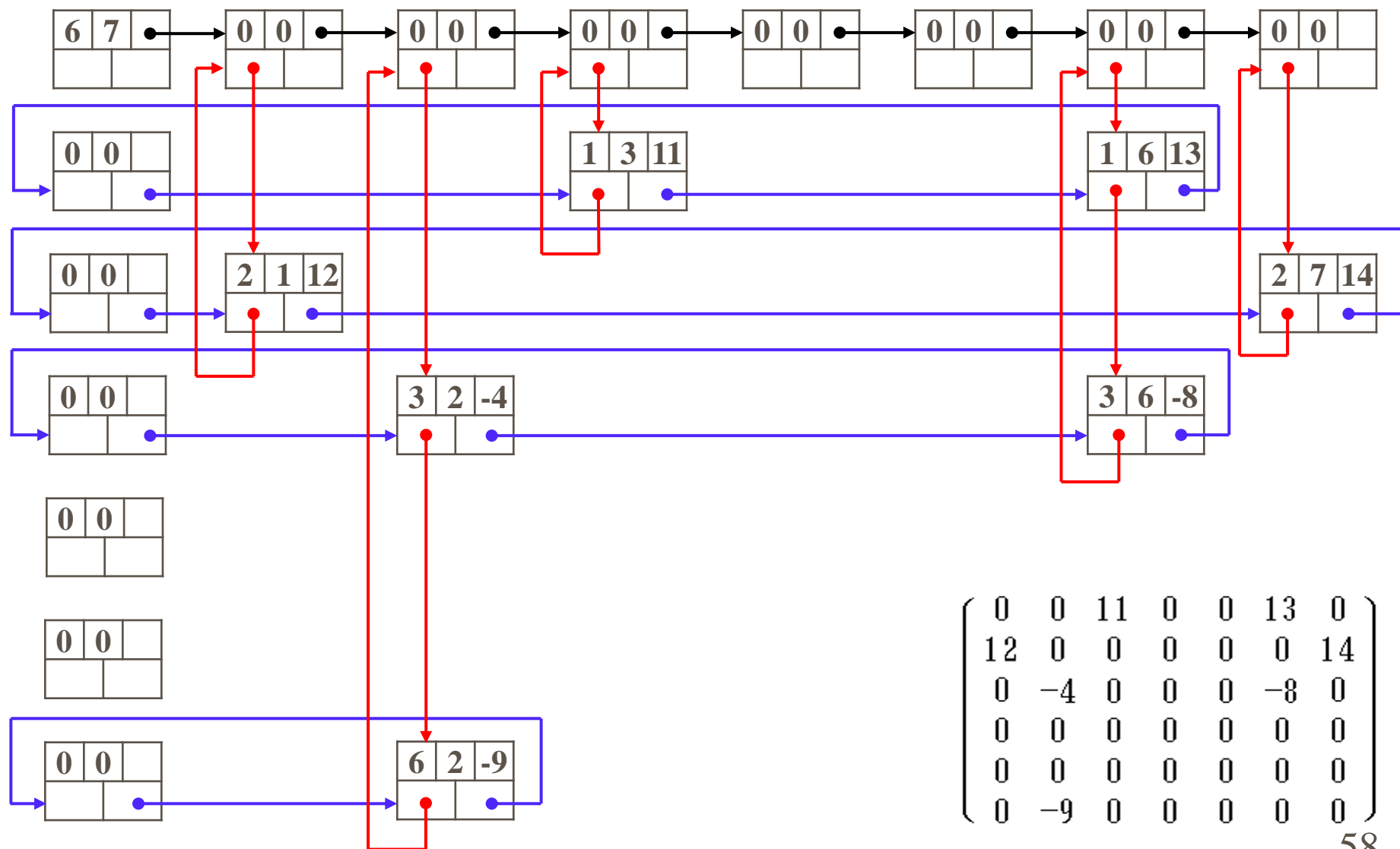
列数

row	col	next

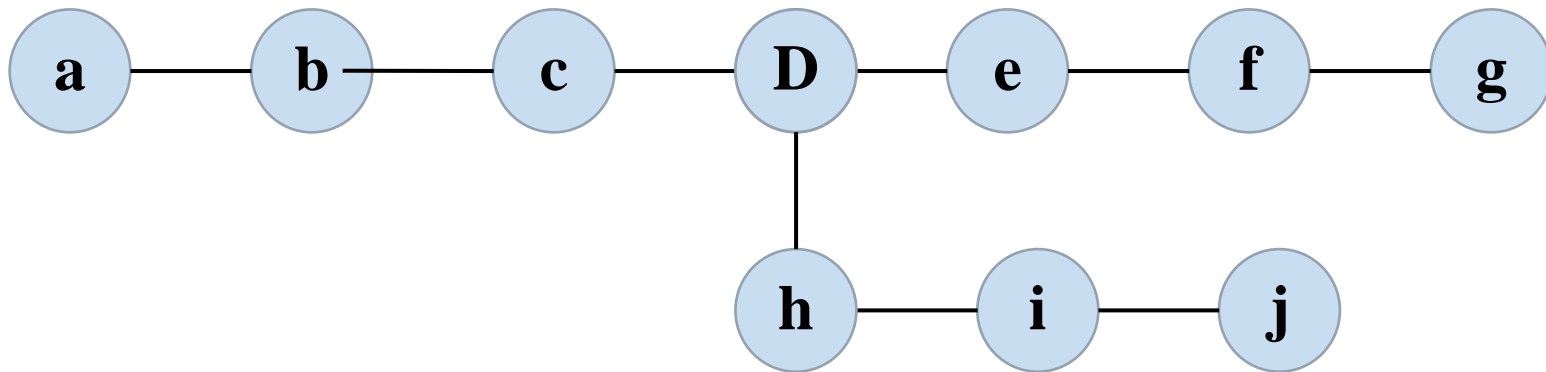
5. 稀疏矩阵的链式存储结构：十字链表

```
typedef struct matrixnode{
    int row,col;
    struct matrixnode *right,*down;
    union{
        int value;
        struct matrixnode *next;
    }tag;
}matrixnode;
typedef matrixnode *spmatrix; //指向上述结点的指针
typedef spmatrix headspmatrix[100];
// headspmatrix = 类型spmatrix+容量[100] = 类型matrixnode*+容量[100] 即指针数组
```

5. 稀疏矩阵的链式存储结构：十字链表



5.4 广义表

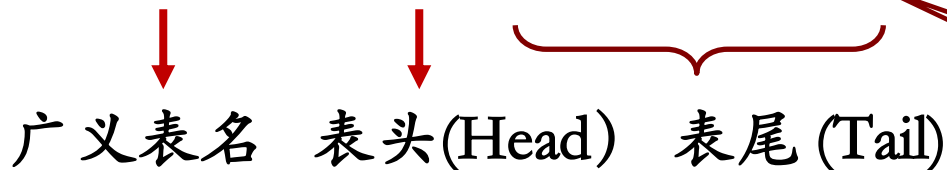


5.4 广义表

1、定义：

广义表是线性表的推广，也称为列表 (lists)

记为： $LS = (a_1, a_2, \dots, a_n)$


广义表名 表头(Head) 表尾 (Tail)

n 是表长

在广义表中约定：

- ① 第一个元素是表头，而其余元素组成的表称为表尾；
- ② 用小写字母表示原子类型，用大写字母表示列表。

5.4 广义表

例如：

- $D = ()$ 空表；其长度为零。
- $A = (a, (b, c))$ 表长度为2的广义表，其中第一个元素是单个数据a，第二个元素是一个子表 (b, c) 。
- $B = (A, A, D)$ 长度为3的广义表，其前两个元素为表A，第三个元素为空表D。
- $C = (a, C)$ 长度为2递归定义的广义表，C相当于无穷表 $C = (a, (a, (a, (...))))$ 。

$\text{head}(A) = a$ 表A的表头是a。

$\text{tail}(A) = ((b, c))$ 表A的表尾是 $((b, c))$ ，广义表的表尾一定是一个表。

5.4 广义表

- 有次序性 一个直接前驱和一个直接后继
- 有长度 =表中元素个数
- 有深度 =表中括号的重数
- 可递归 自己可以作为自己的子表
- 可共享 可以为其他广义表所共享

注意：任何一个非空表，表头可能是原子，也可能是列表；但表尾一定是列表！

5.4 广义表

例如: $D = (E, F) = ((a, (b, c)), F)$

$\text{Head}(D) = E$ $\text{Tail}(D) = (F)$

$\text{Head}(E) = a$ $\text{Tail}(E) = ((b, c))$

$\text{Head}((b, c)) = (b, c)$ $\text{Tail}((b, c)) = ()$

$\text{Head}((b, c)) = b$ $\text{Tail}((b, c)) = (c)$

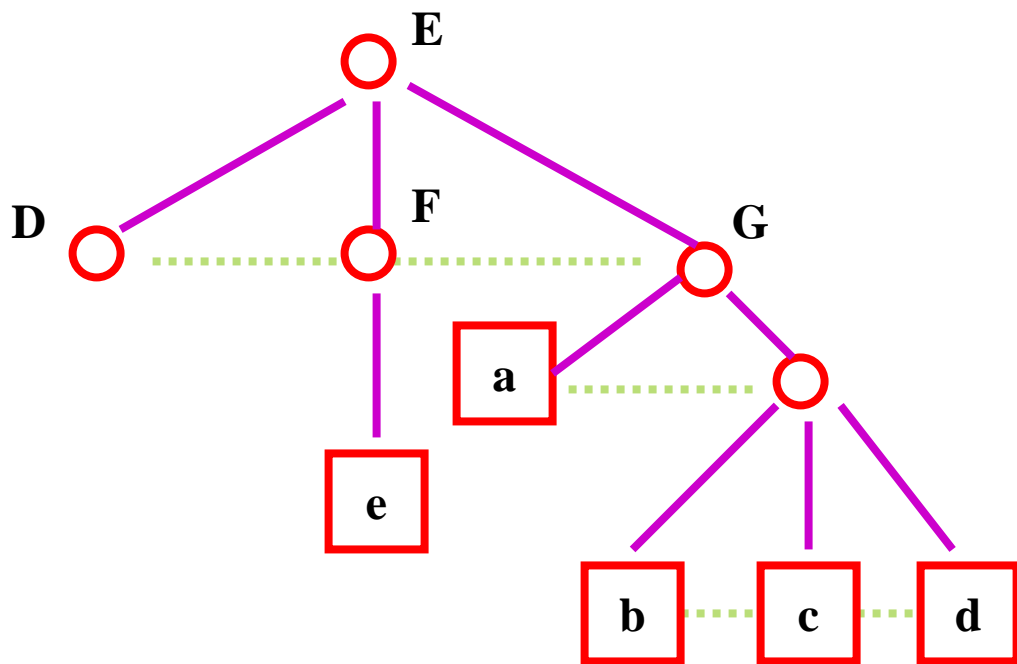
$\text{Head}(c) = c$ $\text{Tail}(c) = ()$

5.4 广义表

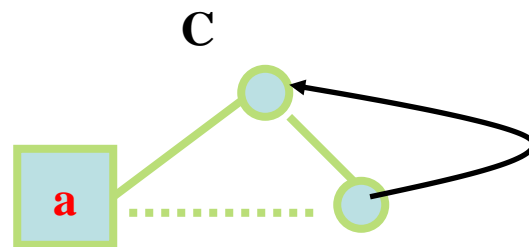
例：试用图形表示下列广义表。

(设 \bigcirc 代表子表, \boxed{e} 代表元素)

① $E=(D,F,G)=((),(e),(a, (b,c,d)))$



② $C=(a , C)$



① 的长度为3，深度为3

深度=括号的重数= 结点 \bigcirc 层数

② 的长度为2，深度为 ∞

5.4 广义表

□ 抽象数据类型

ADT Glist {

数据对象： $D = \{e_i \mid i=1,2,\dots,n; n \geq 0;$
 $e_i \in \text{AtomSet} \text{ 或 } e_i \in \text{GList},$
 $\text{AtomSet} \text{ 为某个数据对象} \}$

数据关系：

$LR = \{ \langle e_{i-1}, e_i \rangle \mid e_{i-1}, e_i \in D, 2 \leq i \leq n \}$

基本操作：

} ADT Glist

5.4 广义表

□ 抽象数据类型——基本操作

- **结构的创建和销毁**

InitGList(&L); DestroyGList(&L);
CreateGList(&L, S); CopyGList(&T, L);

- **状态函数**

GListLength(L); GListDepth(L);
GListEmpty(L); GetHead(L); GetTail(L);

- **插入和删除操作**

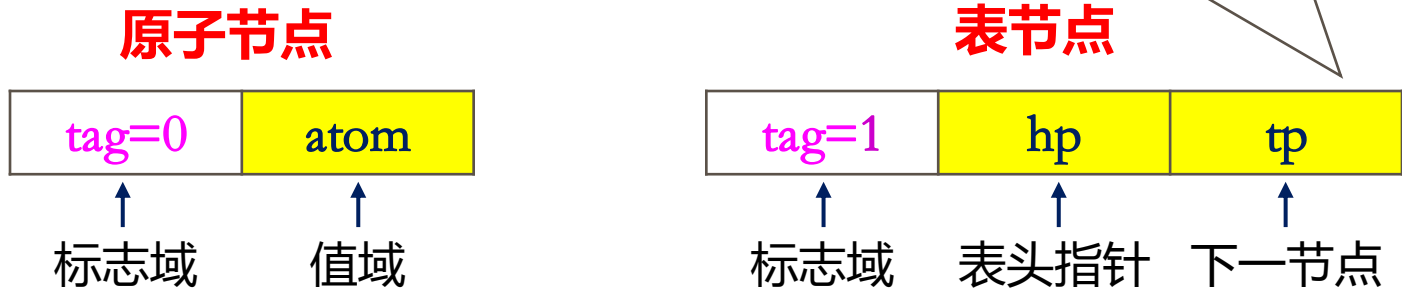
InsertFirst_GL(&L, e);
DeleteFirst_GL(&L, &e);

- **遍历**

Traverse_GL(L, Visit());

5.5 广义表存储结构

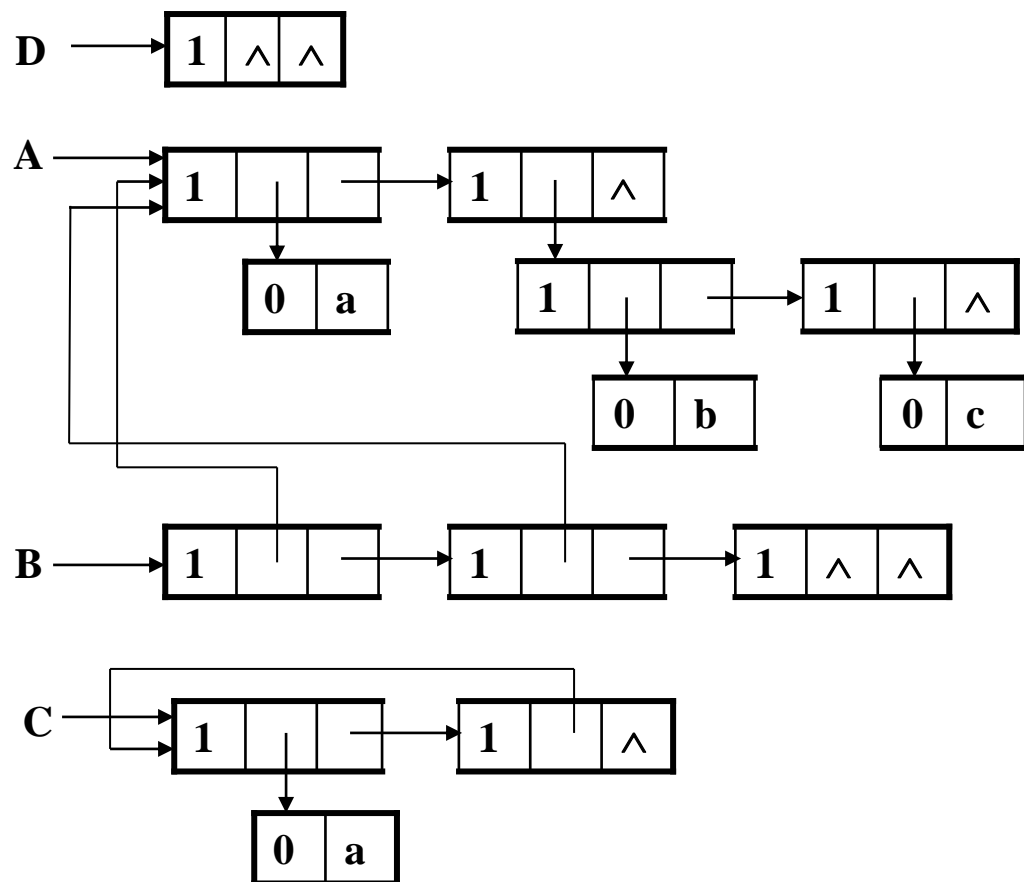
法一：



```
typedef enum{ATOM,LIST} ElemTag;
typedef struct GLNode{
    ElemTag tag;
    union {
        AtomType atom;
        struct {
            struct GLNode *hp, *tp;
        } ptr;
    };
}*GList;
```

5.5 广义表存储结构

方法一：广义表A、B、C、D的存储结构图



$D = ()$

$A = (a, (b, c))$

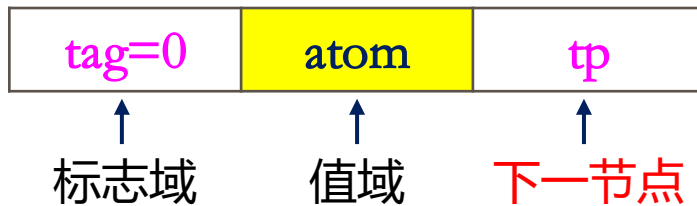
$B = (A, A, D)$

$C = (a, C)$

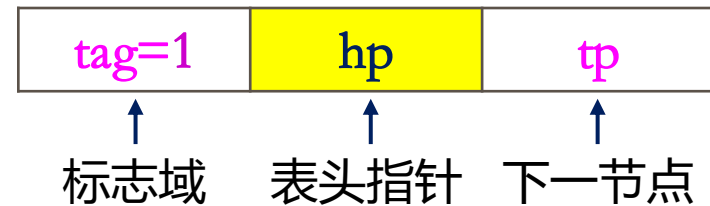
5.5 广义表存储结构

法二:

原子节点



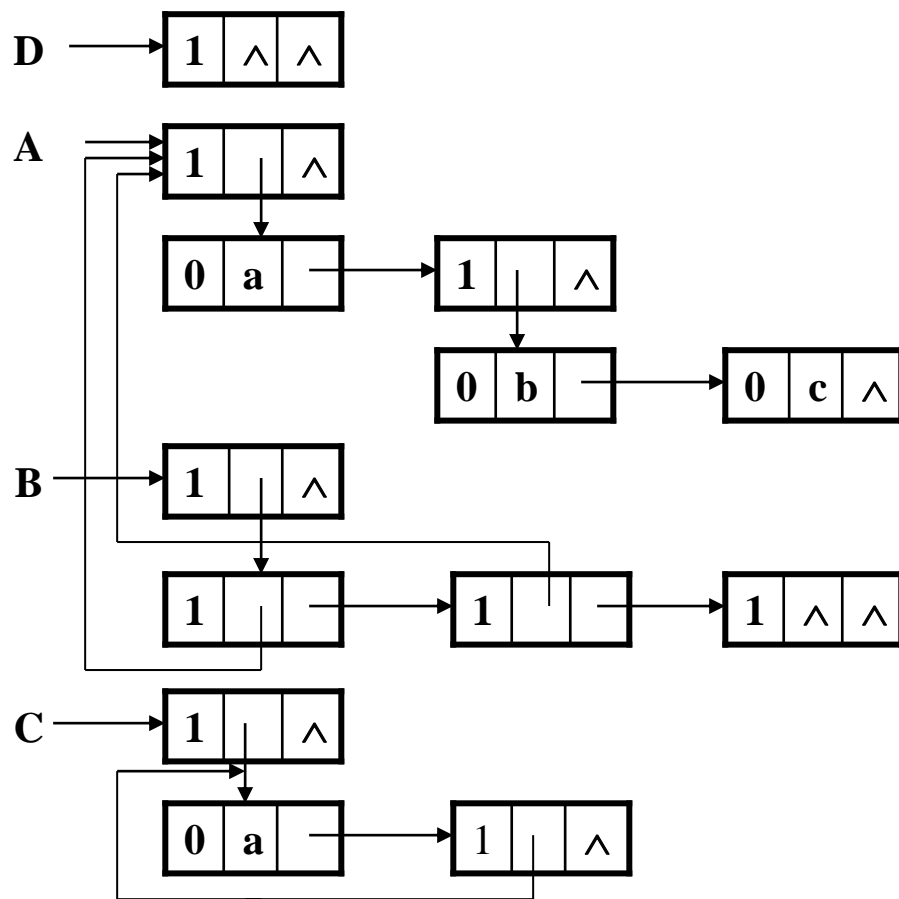
表节点



```
typedef enum{ATOM, LIST} ElemTag;
typedef struct GLNode{
    ElemTag tag;
    union {
        AtomType atom;
        struct GLNode *hp;
    };
    struct GLNode *tp;
}*GList;
```

5.5 广义表存储结构

方法二：广义表A、B、C、D、E的存储结构图（p109）



D = ()

A = (a, (b, c))

B = (A, A, D)

C = (a, C)

5.5 广义表存储结构

介绍两种特殊的基本操作

GetHead (L) —— **取表头** (可能是原子或列表)

GetTail (L) —— **取表尾** (一定是列表)

5.5 广义表存储结构

法一：

$L = (a, (a, b))$

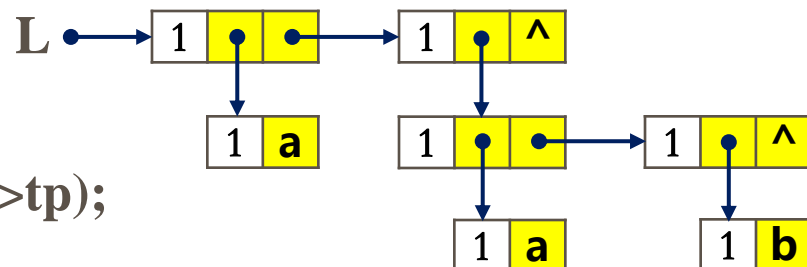
GList GetHead(**GList** L) {

if(L==NULL) return NULL;

if(L->tp->tag==ATOM) return (L->tp);

else return L->hp;

}



GList GetTail(**GList** L) {

if(L->tp!=NULL) return (L->tp);

else return NULL;

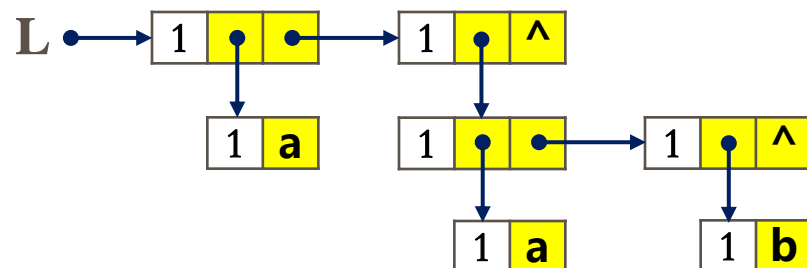
}

5.5 广义表存储结构

法一：

```
GList GetLast (GList L) {  
    GLNode *p=L->tp;  
    while(p->tp!=NULL) p=p->tp;  
    if(p->tag==ATOM) return p;  
    else return GetLast(p->hp);  
}
```

$L = (a, (a, b))$



5.5 广义表存储结构

1. **GetTail** **【(b, k, p, h)】** = (k, p, h) ;
2. **GetHead** **【((a,b), (c,d))】** = (a,b) ;
3. **GetTail** **【((a,b), (c,d))】** = ((c,d)) ;
4. **GetTail** **【GetHead **【((a,b),(c,d))】**】** = (b) ;
5. **GetTail** **【(e)】** = () ;
6. **GetHead** **【(())】** = () .
7. **GetTail** **【(())】** = () .

正在答疑
