人工智能学院

Data Structures and Algorithms

Multi-Dimensional Array

Preview

1. Definition
2. Implementation

Definition

◆ 数组的定义及特点：

定义：数组是由类型相同的数据元素构成的有序集合，每个元素称为数组元素，每个元素受n(n>=1)个线性关系的约束，可以通过下标访问该数据元素。

特点：数组可以看成是线性表的推广，其特点是结构中的元素本身可以是具有某种结构的数据， 但属于同一数据类型。简单来说：数组结构固定，数组行列数不可变、数据元素同构。

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

◆ 栈的运算：

给定一组下标，存取、修改相应的数据元素；一般不做插入、删除操作

Definition

ADT Array {

数据对象：

$j_i = 0,\ldots,b_i - 1,\ \ i=1,2,..,n$

D = {$a_{j1,j2,\ldots,jn}$ | n>0称为数组的维数, bi是数组第i维的长度，

j_i是数组元素第i维的下标，$a_{j1,j2,\ldots,jn}$属于ElemSet}

数据关系：

R = {R1, R2, ..., Rn}

Ri = {<aj1 ,... ji ,... jn , aj1 , ...ji +1, ...jn > |

$0 \le j_k \le b_k - 1,\ 1 \le k \le n$ 且$k \neq i$,

$0 \le j_i \le b_i - 2,\ i=2,\ldots,n$ }

} ADT Array

Definition

基本操作：

InitArray(*A, n, bound1, ..., boundn)

操作结果：若维数 n 和各维长度合法，
则构造相应的数组A，并返回OK

DestroyArray(*A)

操作结果：销毁数组A

Value(A, *e, index1, ..., indexn)

初始条件：A是n维数组，e为元素变量，
随后是n个下标值。

操作结果：若各下标不超界，则e赋值为
所指定的A 的元素值，并返回OK

Assign(*A, e, index1, ..., indexn)

初始条件：A是n维数组，e为元素变量，
随后是n个下标值。

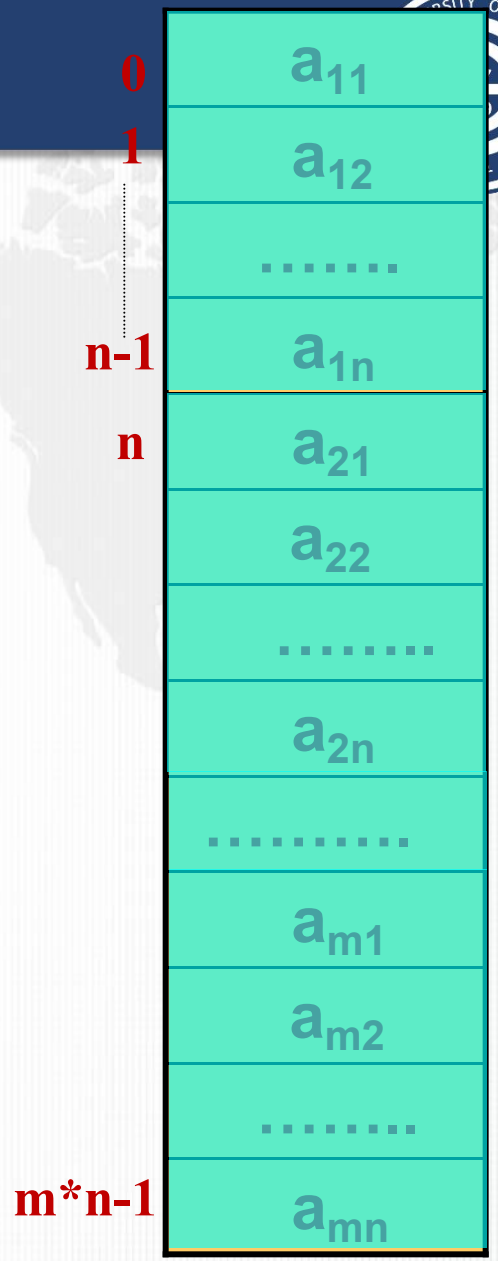
操作结果：若各下标不超界，则e值赋给 所指定的A
的元素，并返回OK

Definition

◆ 次序约定：

- 以行序为主序：BASIC、PASCAL、C

$$\begin{array}{llll} a_{11} & a_{12} & \cdots\cdots & a_{1n} \\ a_{21} & a_{22} & \cdots\cdots & a_{2n} \\ & & \cdots\cdots\cdots\cdots & \\ a_{m1} & a_{m2} & \cdots\cdots & a_{mn} \end{array}$$

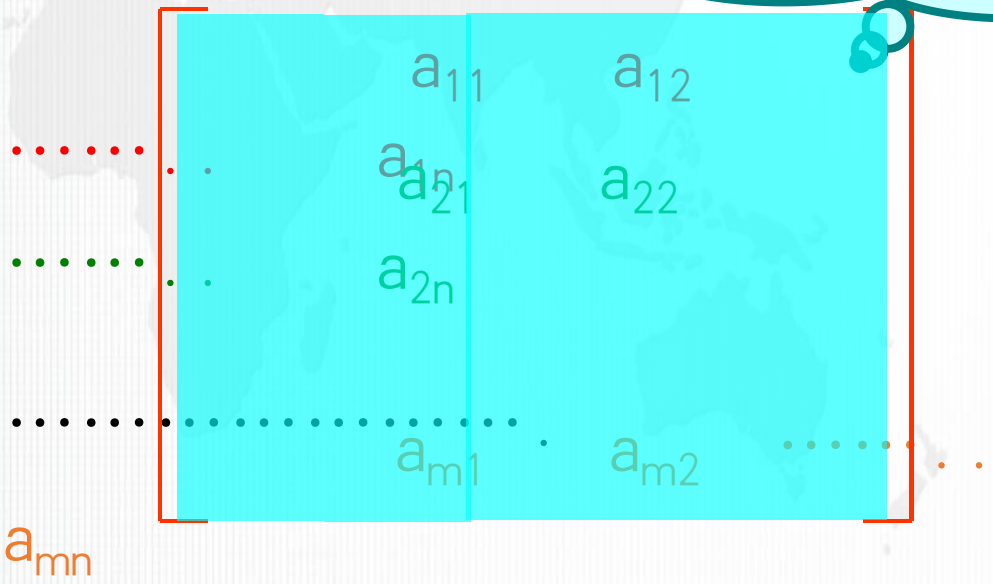


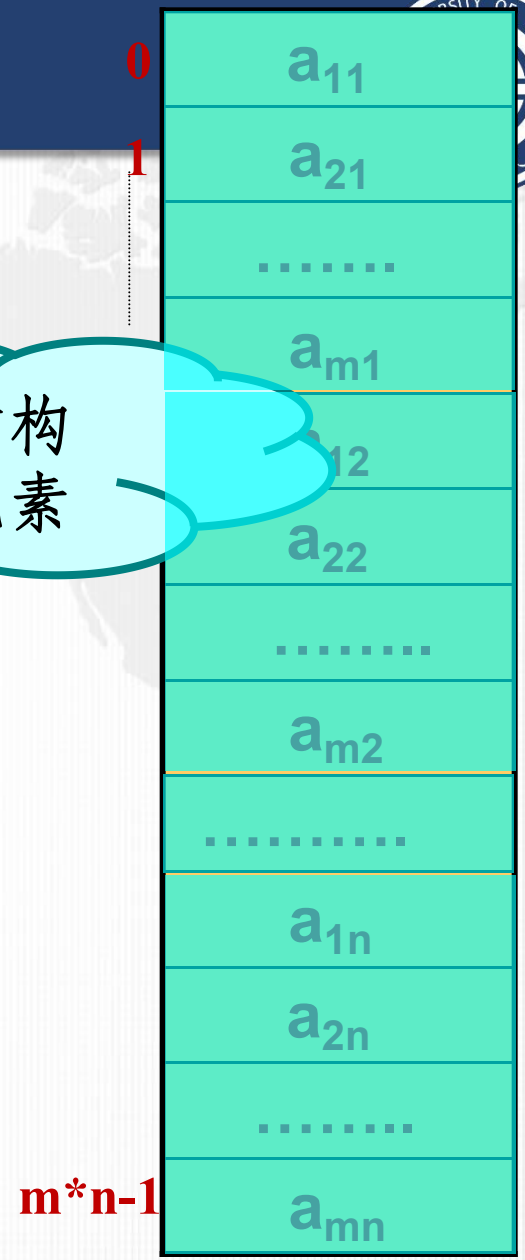
$$Loc(a_{ij}) = Loc(a_{11}) + [(i-1)n + (j-1)] * L$$

Definition

- 次序约定：
  - 以行序为主序：BASIC、PASCAL、C
  - 以列序为主序：FORTRAN

数组的顺序存储结构
可随机存取任一元素





$$Loc(a_{ij})=Loc(a_{11})+[(j-1)m+(i-1)]*L$$

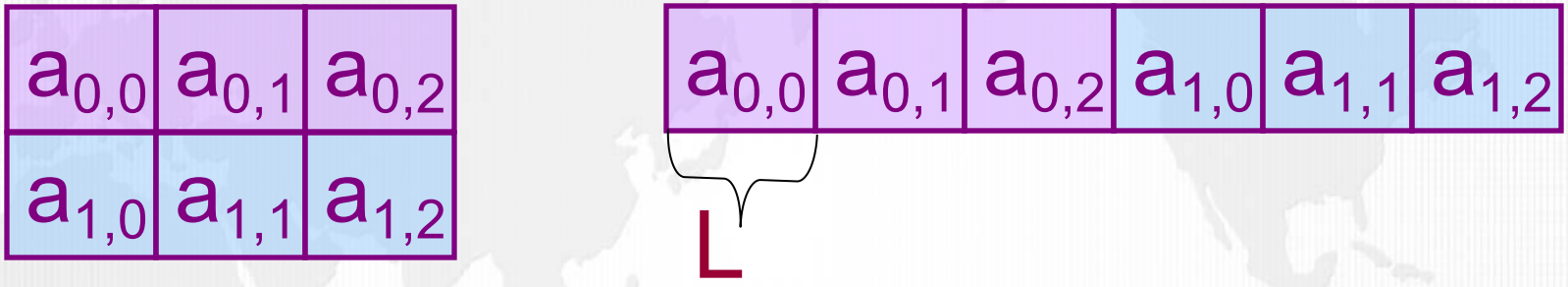
Implementation

◆ 顺序存储结构结构体：

```
5   #define MAX_ARRAY_DIM 8
6
7
8   typedef int ElemType;
9   struct Array {
10      ElemType *base;    // 数组元素基址
11      int dim;           // 数组维数
12      int *bounds;       // 数组维界基址
13      int *constants;    // 常数基址
14  } ;
15  typedef struct Array Array;
```

Implementation

◆ 以“行序为主序”的存储映象：



二维数组A中任一元素$a_{i,j}$的存储位置

$$LOC(i,j) = LOC(0,0) + (b_2 \times i + j) \times L$$

推广到一般情况，可得到 n 维数组数据元素存储位置的映象关系

$$LOC(j_1, j_2, ..., j_n) = LOC(0,0,...,0) + (b_2 \times ... \times b_n \times j_1 + b_3 \times ... \times b_n \times j_2 + ...)$$

Implementation

- 数组应用——矩阵的压缩存储：

  在编写程序时往往都是二维数组表示矩阵，然而在数值分析中经常出现一些阶数很高的的矩阵同时在距震中有很多值相同的元素，或者是零元素，为了节省空间，可以对这类矩阵进行压缩存储，所谓的压缩存储就是，多个值相同的元之分配一个存储空间，对零元不分配空间。

- 矩阵的压缩存储基本思想：

  1) 值相同的元素分配一个存储空间；
  2) 零元素不分配空间

压缩存储后，若能得到元素aij的存储地址，则仍具有随机存取功能。

Implementation

- 压缩矩阵分类：

  假如值相同的元素或者零元素在矩阵中的分配有一定的规律，则我们称此类矩阵为特殊矩阵。

  1）n阶对称矩阵：满足Aij = Aji 1<=i,j<=n;

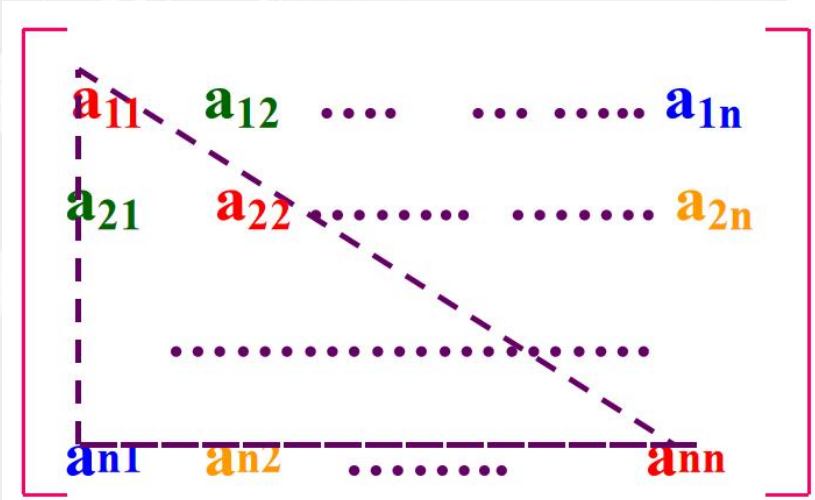
  2）稀疏矩阵：非零元较零元少，且分布没有规律。

  3）上下三角阵：即有上半部（或下半部）元素均为c或0，其余半部随机分布

  4）对角阵和三对角阵： 对角矩阵所有非零元素集中在以主对角线为中心的带状区域。三对角矩阵以主对角线为中心的三条带状区域有元素，其余位置为0。

Implementation

◆ 对称矩阵：



按行序为主序

a_{11}
a_{21}
a_{22}
a_{31}
a_{32}
a_{33}
……
a_{n1}
……
a_{nn}

$a_{ij} = a_{ji}$

压缩存储元素个数：$1+2+\ldots+n=\frac{n(n+1)}{2}$

$$Loc(a_{ij}) = Loc(a_{11}) + \left[\frac{i(i-1)}{2} + (j-1)\right] * L \quad (i \geq j)$$

$$Loc(a_{ij}) = Loc(a_{11}) + \left[\frac{j(j-1)}{2} + (i-1)\right] * L \quad (i < j)$$

Implementation

◆ 三角矩阵：

$$\begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ & & & & 0 \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

按行序为主序

a_{11}
a_{21}
a_{22}
a_{31}
a_{32}
a_{33}
……
a_{n1}
……
a_{nn}

压缩存储元素个数：$1+2+\ldots+n=\frac{n(n+1)}{2}$

$Loc(a_{ij})= Loc(a_{11})+[\frac{i(i-1)}{2}+(j-1)]*L \quad (i\geq j)$

$a_{ij}= 0 \quad (i<j)$

Implementation

◆ 对角矩阵：

$$\begin{bmatrix} a_{11} & a_{12} & 0 & \cdots\cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 \cdots\cdots & 0 \\ 0 & a_{32} & a_{33} & a_{34} \cdots & 0 \\ & \cdots\cdots & & & \\ 0 & 0 \cdots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & \cdots & \cdots & a_{n,n-1} \quad a_{nn} \end{bmatrix}$$

按行序为主序

a_{11}
a_{12}
a_{21}
a_{22}
a_{23}
a_{32}
a_{33}
a_{34}
……
$a_{n\,n-1}$
a_{nn}

压缩存储元素个数：$(n-2)\times 3+4=3n-2$

$Loc(a_{ij})=Loc(a_{11})+[\,2(i-1)+(j-1)\,]*L$ （$|i-j|\leqslant 1$）

$a_{ij}=0$ 其它

Implementation

- **稀疏矩阵：**

  非零元较零元少，且分布没有一定规律的矩阵

  假设 m 行 n 列的矩阵含 t 个非零元素

  通常认为 δ ≤0.05 的矩阵为稀疏矩阵

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

稀疏因子

- **矩阵的压缩存储基本思想：**

  1) 值相同的元素分配一个存储空间；

  2) 零元素不分配空间

M由{(1,2,12), (1,3,9), (3,1,-3),
(3,6,14), (4,3,24),(5,2,18),
(6,1,15), (6,4,-7) }
和矩阵维数（6,7）唯一确定

$$M = \begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}_{6\times7}$$

Implementation

- 三元组表：

```
4
5   # define MAX_SIZE 100
6   typedef struct {
7       int row_index, col_index;
8       int length;
9   } Triple;
10  typedef struct {
11      Triple data[MAX_SIZE + 1];
12      int rows, cols, elements;
13  } TSMatrix;
14
```

Implementation

◆ 三元组表：

行列下标　　非零元值

	i	j	e
0	6	7	8
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

ma.data

TSMatrix ma;

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

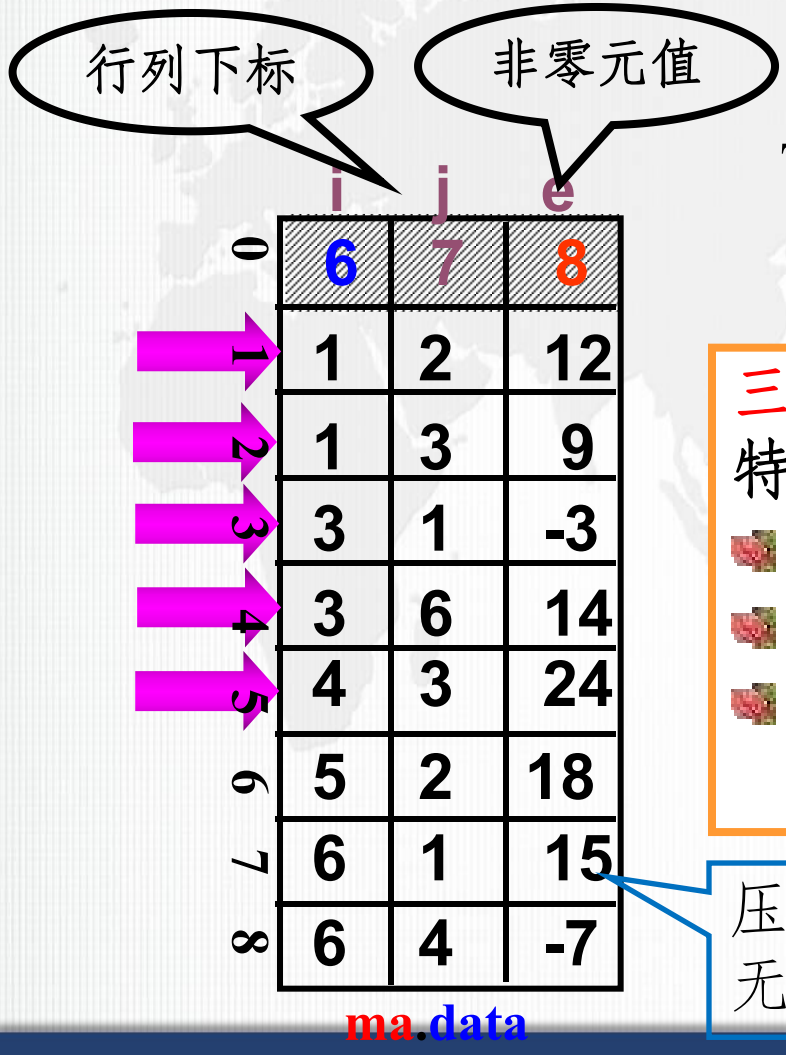
data[0]号单元未用
或存放矩阵行列维数和非零元个数

矩阵行数：　**ma.cols=6**
矩阵列数：　**ma.rows=7**
非零元个数：　**ma.elements=8**

三元组表所需存储单元个数为3(ma.elements+1)

Implementation

◆ 三元组表：



TSMatrix ma;

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

三元组顺序表又称有序的双下标法

特点：

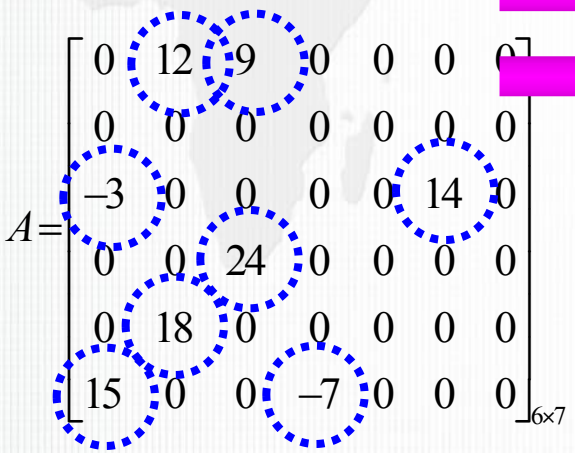
- 非零元在表中按行序有序存储
- 便于进行依行顺序处理的矩阵运算
- 若需存取某一行中的非零元，需从头开始查找。

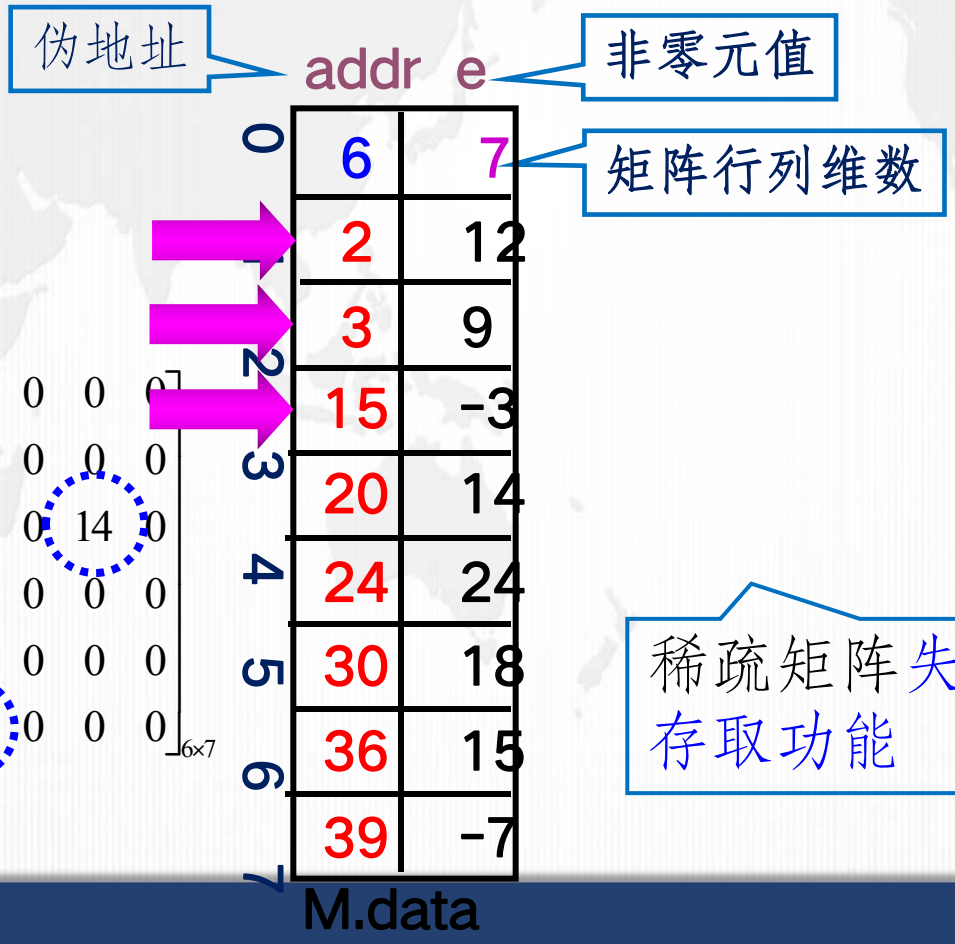
压缩存储后，元素a_{ij}的存储位置与其下标无关，而取决于之前的非零元个数

Implementation

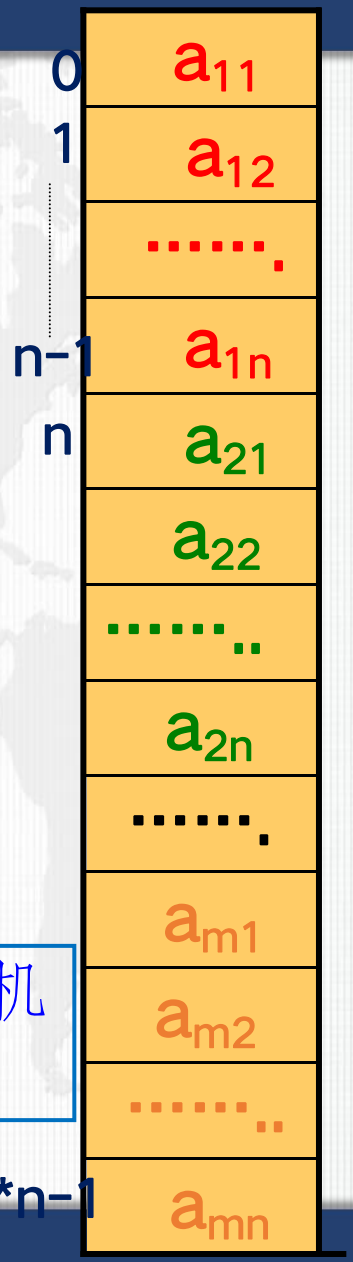
- 伪地址表示：

本元素在矩阵中（包括零元素在内）按行优先顺序的相对位置。





稀疏矩阵失去随机存取功能



Implementation

- 三元组表示压缩矩阵转置：

  已知一个稀疏矩阵的三元组表，求该矩阵转置矩阵的三元组表。

- 一般转置算法：

$$M_{mu\times nu}=\begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1nu} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2nu} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{mu1} & a_{mu2} & \cdots & \cdots & a_{munu} \end{bmatrix} \Rightarrow T_{nu\times mu}=\begin{bmatrix} a'_{11} & a'_{12} & \cdots & \cdots & a'_{1mu} \\ a'_{21} & a'_{22} & \cdots & \cdots & a'_{2mu} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a'_{nu1} & a'_{nu2} & \cdots & \cdots & a'_{numu} \end{bmatrix}$$

```
for(col=0;col<nu;col++)
    for(row=0;row<mu;row++)
        T[col][row]=M[row][col];
```

T(n)=O(mu×nu)

Implementation

- 三元组表示压缩矩阵转置：

  已知一个稀疏矩阵的三元组表，求该矩阵转置矩阵的三元组表。

- 三元组转置算法：

  ①将矩阵行、列维数互换，非零元个数不变

  ②将每个三元组中的i和j相互调换，非零元值不变

  ③重排次序，使T.data中元素以T的行(M的列)为主序

$$M=\begin{bmatrix}0&12&9&0&0&0&0\\0&0&0&0&0&0&0\\-3&0&0&0&0&14&0\\0&0&24&0&0&0&0\\0&18&0&0&0&0&0\\15&0&0&-7&0&0&0\end{bmatrix}_{6\times7}$$

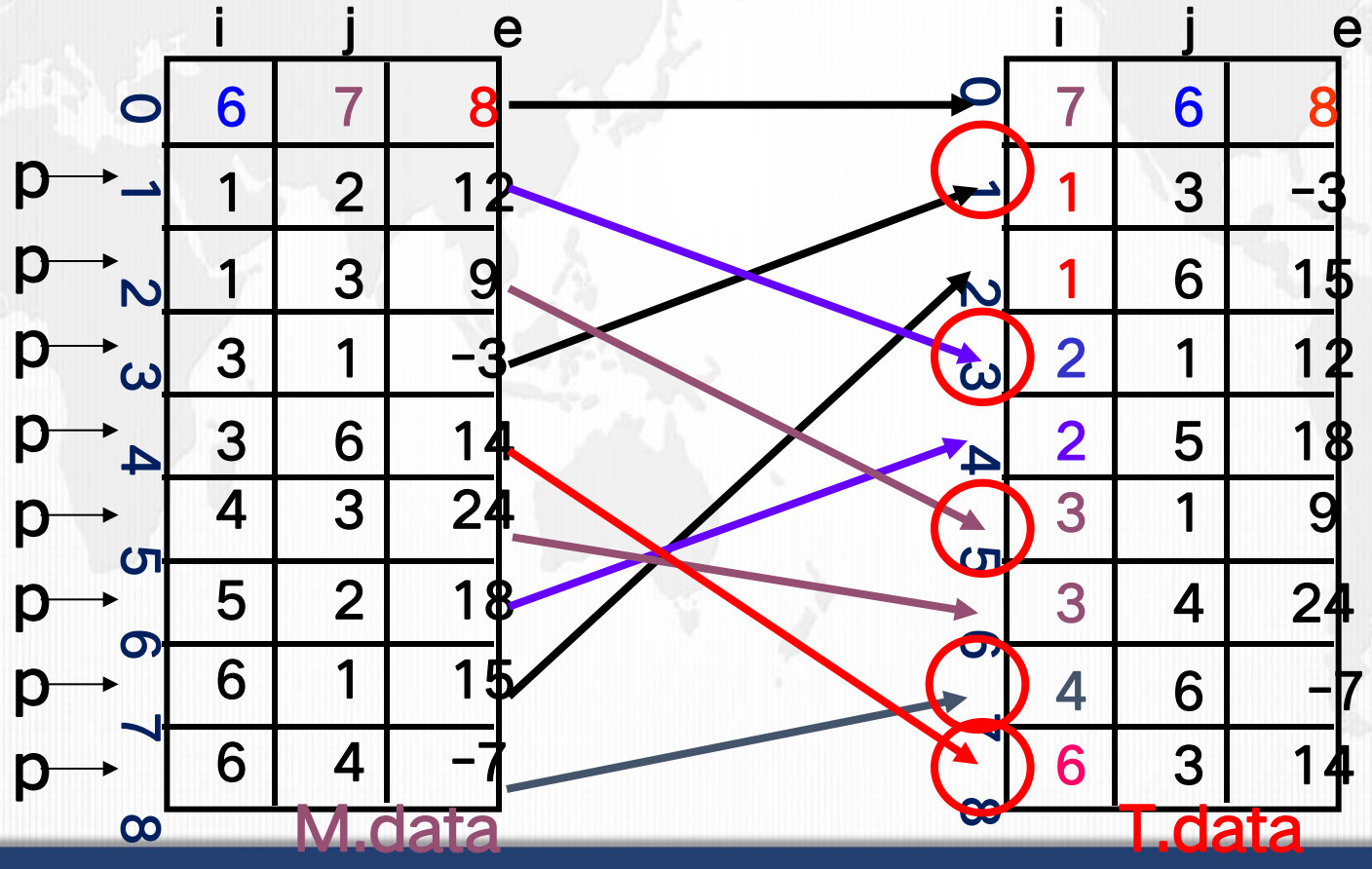
$$T=\begin{bmatrix}0&0&-3&0&0&15\\12&0&0&0&18&0\\9&0&0&24&0&0\\0&0&0&0&0&-7\\0&0&0&0&0&0\\0&0&14&0&0&0\\0&0&0&0&0&0\end{bmatrix}_{7\times6}$$

Implementation

- 快速转置算法：

1.按M.data中三元组次序转置，结果放入T.data中恰当位置

2.需要预先确定M中每一列第一个非零元在T.data中位置，



Implementation

◆ 快速转置算法：

1.按M.data中三元组次序转置，结果放入T.data中恰当位置
2.需要预先确定M中每一列第一个非零元在T.data中位置，为确定这些位置，应先求得M的每一列中非零元个数

实现：设两个数组

num[col]：矩阵M中第col列中非零元个数

cpot[col]：矩阵M中第col列第一个非零元在T.data中位置

显然有：

$$\begin{cases} cpot[1]=1; \\ cpot[col]= cpot[col-1]+ num[col-1]; \quad (2\le col\le M.nu) \end{cases}$$

Implementation

- 快速转置算法：

实现：设两个数组

num[col]：矩阵M中第col列中非零元个数

cpot[col]：矩阵M中第col列第一个非零元在T.data中位置

显然有：

$$\begin{cases} cpot[1]=1; \\ cpot[col]= cpot[col-1]+ num[col-1]; \quad (2\le col\le M.nu) \end{cases}$$

i j v

M.data

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

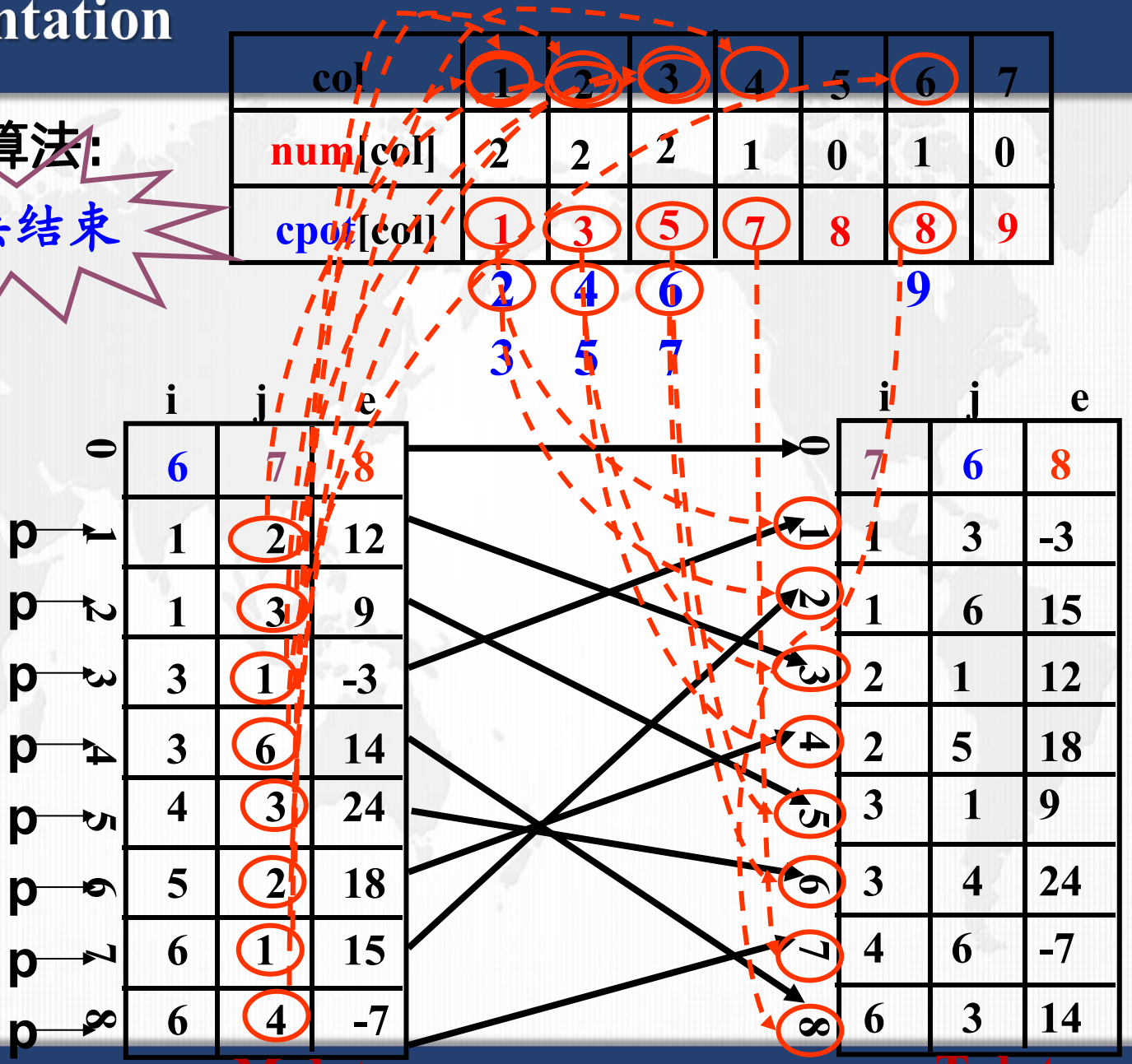
col	1	2	3	4	5	6	7
num[col]	2	2	2	1	0	1	0
cpot[col]	1	3	5	7	8	8	9

Implementation

- 快速转置算法：

算法结束

col	1	2	3	4	5	6	7
num[col]	2	2	2	1	0	1	0
cpot[col]	1	3	5	7	8	8	9



Implementation

```
Status fast_transpos ( TSMatrix M, TSMatrix *T )
{  int col,p,k , num[N],cpot[N];
   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(p=1; p<=M.tu; p++) num[M.data[p].j ]++;
      cpot[0]=0; cpot[1]=1;
      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;   k=cpot[col];
            T.data[k].i=M.data[p].i;
            T.data[k].j=M.data[p].j;
            T.data[k].e=M.data[p].e;
        }
   }
   return OK;
}
```

设置矩阵信息

扫描M的三元组，取其所在列

num[]清零

设置num[]

M各列第1个非零元位置

扫描M的三元组，逐一转置

转置

	i	j	e
0	6	7	8
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

p→

M.data

T(n)=O(M.nu+M.tu)

若M.tu与M.mu × M.nu同数量级

则 T(n)=O(M.mu × M.nu)

Implementation

- 链式存储：

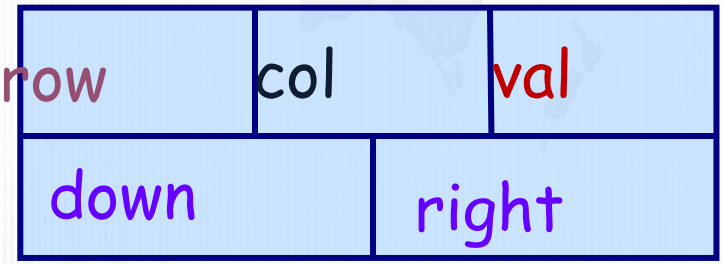
  引入链式存储的原因：

  ①用三元组表存储稀疏矩阵，在单纯的存储和做类似转置之类的运算时可以节约存储空间，且运算速度较快;

  ②但当进行矩阵相加等运算时，稀疏矩阵的非零元位置和个数都会发生变化。使用三元组表必然会引起数组元素　的大量移动。

- 十字链表：

  每个非零元用含有五个域的结点表示（非零元的所在行、列、值，及同行、同列的下一个非零元）



Implementation

- **十字链表：**

  每个非零元用含有五个域的结点表示（非零元的所在行、列、值，及同行、同列的下一个非零元）

结点结构

```
typedef struct OLNode
{    int  row, col;                 //非零元所在行、列
     ElemType  val;                 //非零元的值
     struct OLNode  *right, *down;
                                    //同行、同列的下一个非零元
}OLNode,* OLink ;
```

十字链表结构

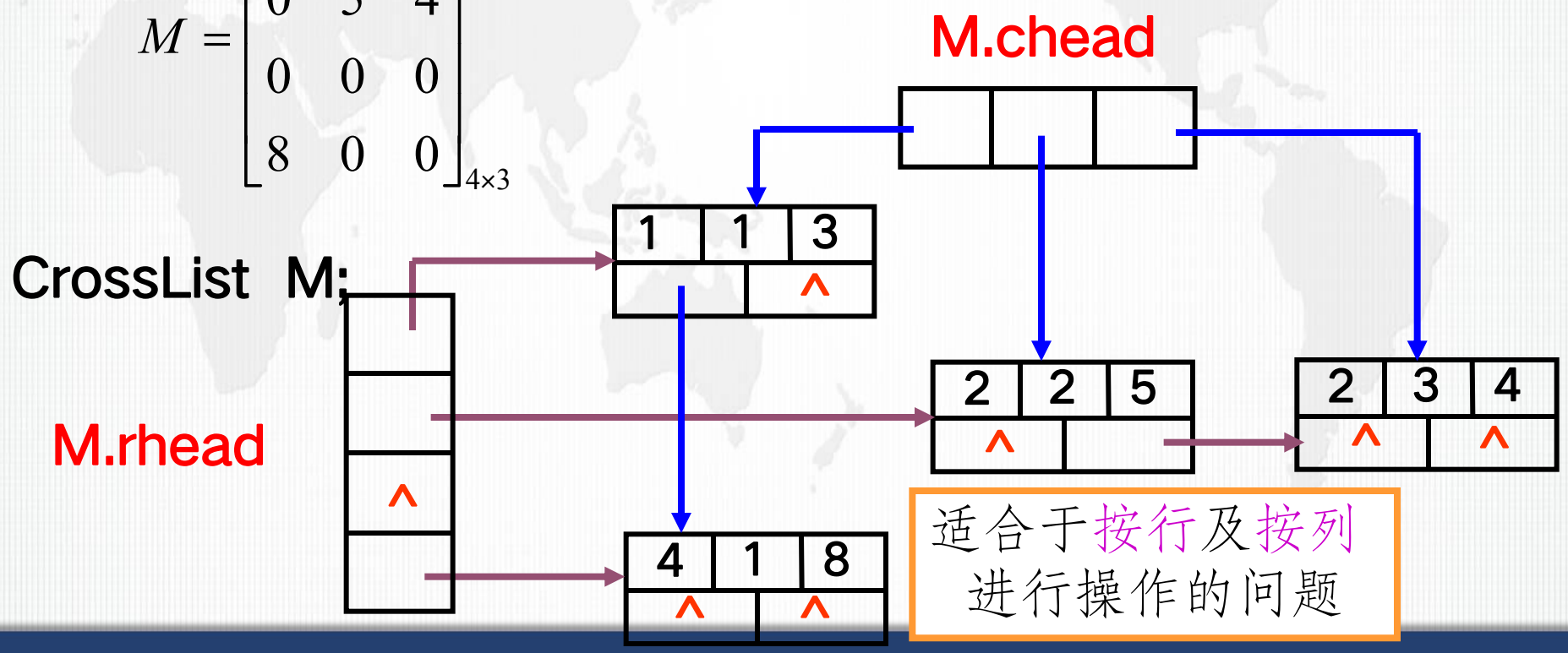
```
typedef struct
{  OLink  rhead[M],chead[N]; //行、列指针数组
     int  mu, nu, tu;               //行、列数及非零元个数
}CrossList;
```

Implementation

- 十字链表：

  每个非零元用含有五个域的结点表示（非零元的所在行、列、值，及同行、同列的下一个非零元）

$$M=\begin{bmatrix}3&0&0\\0&5&4\\0&0&0\\8&0&0\end{bmatrix}_{4\times3}$$



Homework

- **Happy Holidays**

Thanks!

See you in the next session!