第5章 数组和广义表

数组是一种人们非常熟悉的数据结构，几乎所有的程序设计语言都支持这种数据结构或将这种数据结构设定为语言的固有类型。数组这种数据结构可以看成是线性表的推广。

科学计算中涉及到大量的矩阵问题，在程序设计语言中一般都采用数组来存储，被描述成一个二维数组。但当矩阵规模很大且具有特殊结构(对角矩阵、三角矩阵、对称矩阵、稀疏矩阵等)，为减少程序的时间和空间需求，采用自定义的描述方式。

广义表是另一种推广形式的线性表，是一种灵活的数据结构，在许多方面有广泛的应用。

5.1 数组的定义

- **数组**是一组偶对(下标值，数据元素值)的集合。在数组中，对于一组有意义的下标，都存在一个与其对应的值。一维数组对应着一个下标值，二维数组对应着两个下标值，如此类推。
- **数组**是由n(n>1)个具有相同数据类型的数据元素a_1，a_2，…，a_n组成的有序序列，且该序列必须存储在一块地址连续的存储单元中。
  - 数组中的数据元素具有相同数据类型。
  - 数组是一种随机存取结构，给定一组下标，就可以访问与其对应的数据元素。
  - 数组中的数据元素个数是固定的。

5.1.1 数组的抽象数据类型定义

1、抽象数据类型定义

ADT Array{

数据对象：j_i= 0,1,…,b_i-1 , 1,2, …,n ;

D = { $a_{j_1 j_2 \ldots j_n}$ | n>0称为数组的维数，b_i是数组第i维的长度，j_i是数组元素第i维的下标，$a_{j_1 j_2 \ldots j_n} \in$ ElemSet }

数据关系：R = {R_1, R_2, …, R_n}

R_i={<$a_{j_1 j_2 \ldots j_i \ldots j_n}$, $a_{j_1 j_2 \ldots j_{i+1} \ldots j_n}$>|$0 \leqq j_k \leqq b_k-1$ ，$1 \leqq k \leqq n$且$k \neq i$，$0 \leqq j_i \leqq b_i-2$， $a_{j_1 j_2 \ldots j_{i+1} \ldots j_n} \in D$ }

基本操作： ……

} ADT Array

由上述定义知，n维数组中有$b_1 \times b_2 \times \ldots \times b_n$个数据元素，每个数据元素都受到n维关系的约束。

2、直观的n维数组

以二维数组为例讨论。将二维数组看成是一个定长的线性表，其每个元素又是一个定长的线性表。

设二维数组$A=(a_{ij})_{m\times n}$，则

$$A=(\alpha_1, \alpha_2, \ldots, \alpha_p) \quad (p=m\text{或}n)$$

其中每个数据元素α_j是一个列向量(线性表)：

$$\alpha_j=(a_{1j}, a_{2j}, \ldots, a_{mj}) \quad 1\leqq j\leqq n$$

或是一个行向量：

$$\alpha_i=(a_{i1}, a_{i2}, \ldots, a_{in}) \quad 1\leqq i\leqq m$$

如图5-1所示。

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

(a) 矩阵表示形式

$$A=\begin{pmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \end{bmatrix} \\ \begin{bmatrix} a_{21} & a_{22} & \cdots & a_{2n} \end{bmatrix} \\ \cdots \; \cdots \; \cdots \; \cdots \; \cdots \\ \begin{bmatrix} a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \end{pmatrix}$$

(b) 列向量的一维数组形式

$$A=\left(\begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} \;\vdots\; \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} \right)$$

(c) 行向量的一维数组形式

图5-1 二维数组图例形式

5.2 数组的顺序表示和实现

数组一般不做插入和删除操作，也就是说，数组一旦建立，结构中的元素个数和元素间的关系就不再发生变化。因此，一般都是采用顺序存储的方法来表示数组。

问题：计算机的内存结构是一维(线性)地址结构，对于多维数组，将其存放(映射)到内存一维结构时，有个次序约定问题。即必须按某种次序将数组元素排成一列序列，然后将这个线性序列存放到内存中。

二维数组是最简单的多维数组，以此为例说明多维数组存放(映射)到内存一维结构时的次序约定问题。

通常有两种顺序存储方式：

(1) **行优先顺序(Row Major Order)**：将数组元素按行排列，第i+1个行向量紧接在第i个行向量后面。对二维数组，按行优先顺序存储的线性序列为：

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

PASCAL、C是按行优先顺序存储的，如图5-2(b)示。

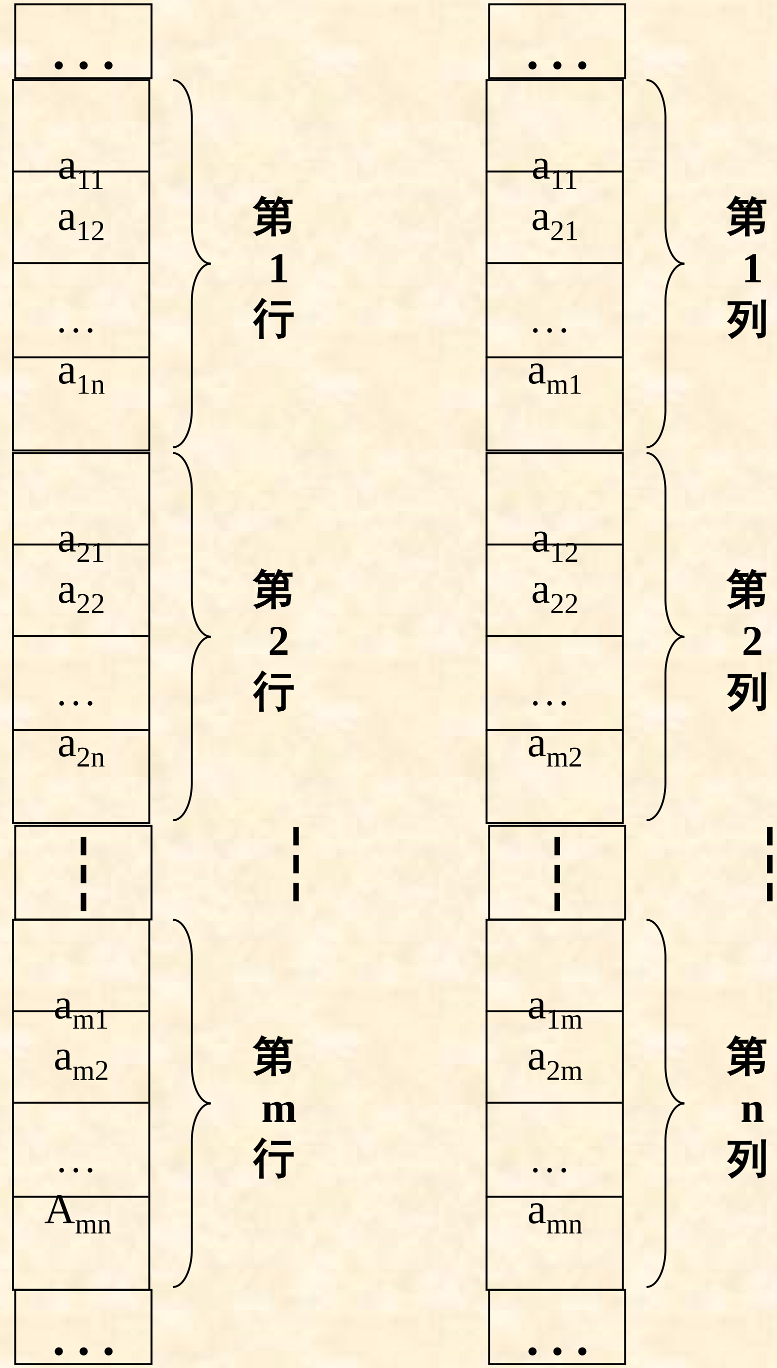
(2) **列优先顺序(Column Major Order)**：将数组元素按列向量排列，第j+1个列向量紧接在第j个列向量之后，对二维数组，按列优先顺序存储的线性序列为：

$a_{11}, a_{21}, \ldots, a_{m1}, \quad a_{12}, a_{22}, \ldots a_{m2}, \quad \ldots\ldots,$ $a_{n1}, a_{n2}, \ldots, a_{nm}$

FORTRAN是按列优先顺序存储的，如图5-2(c)

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

(a) 二维数组的表示形式



(b) 行优先顺序存储　(c) 列优先顺序存储

图5-2 二维数组及其顺序存储图例形式

设有二维数组$A=(a_{ij})_{m\times n}$，若每个元素占用的存储单元数为l(个)，$LOC[a_{11}]$表示元素a_{11}的首地址，即数组的首地址。

1、以“行优先顺序”存储

⑴ 第1行中的每个元素对应的(首)地址是：

$$LOC[a_{1j}]=LOC[a_{11}]+(j-1)\times l \qquad j=1,2, \ldots,n$$

(2) 第2行中的每个元素对应的(首)地址是：

$$LOC[a_{2j}]=LOC[a_{11}]+n\times l +(j-1)\times l \qquad j=1,2, \ldots,n$$

… … …

⑶ 第m行中的每个元素对应的(首)地址是：

$$LOC[a_{mj}]=LOC[a_{11}]+(m-1)\times n\times l +(j-1)\times l \qquad j=1,2, \ldots,n$$

由此可知，二维数组中任一元素a_{ij}的(首)地址是：

$$LOC[a_{ij}]=LOC[a_{11}]+[(i-1)\times n +(j-1)]\times l \qquad (5\text{-}1)$$

$i=1,2, \ldots,m \quad j=1,2, \ldots,n$

根据(5-1)式，对于三维数组$A=(a_{ijk})_{m\times n\times p}$，若每个元素占用的存储单元数为$l$(个)，$LOC[a_{111}]$表示元素$a_{111}$的首地址，即数组的首地址。以“行优先顺序”存储在内存中。

三维数组中任一元素a_{ijk}的(首)地址是：

$$LOC(a_{ijk})=LOC[a_{111}]+[(i-1)\times n\times p+(j-1)\times p+(k-1)]\times l \quad (5\text{-}2)$$

推而广之，对n维数组$A=(a_{j_1j_2\ldots j_n})$，若每个元素占用的存储单元数为l(个)，$LOC[a_{11\ \ldots1}]$表示元素$a_{11\ \ldots1}$的首地址。则以“行优先顺序”存储在内存中。

n维数组中任一元素$a_{j_1j_2\ldots j_n}$的(首)地址是：

$$\begin{aligned}LOC[a_{j_1j_2\ldots j_n}]=LOC[a_{11\ldots1}]&+[(b_2\times\ldots\times b_n)\times(j_1-1)\\&+(b_3\times\ldots\times b_n)\times(j_2-1)+\ldots\\&+b_n\times(j_{n-1}-1)+(j_n-1)]\times l \qquad (5\text{-}3)\end{aligned}$$

2、以“列优先顺序”存储

⑴ 第1列中的每个元素对应的(首)地址是：

$$LOC[a_{j1}]=LOC[a_{11}]+(j-1)\times l \qquad j=1,2,\ldots,m$$

(2) 第2列中的每个元素对应的(首)地址是：

$$LOC[a_{j2}]=LOC[a_{11}]+m\times l+(j-1)\times l \qquad j=1,2,\ldots,m$$

… … …

⑶ 第n列中的每个元素对应的(首)地址是：

$$LOC[a_{jn}]=LOC[a_{11}]+(n-1)\times m\times l+(j-1)\times l \qquad j=1,2,\ldots,m$$

由此可知，二维数组中任一元素a_{ij}的(首)地址是：

$$LOC[a_{ij}]=LOC[a_{11}]+[(i-1)\times m+(j-1)]\times l \qquad (5\text{-}1)$$

$i=1,2,\ldots,n \quad j=1,2,\ldots,m$

5.3 矩阵的压缩存储

在科学与工程计算问题中，矩阵是一种常用的数学对象，在高级语言编程时，通常将一个矩阵描述为一个二维数组。这样，可以对其元素进行随机存取，各种矩阵运算也非常简单。

对于高阶矩阵，若其中非零元素呈某种规律分布或者矩阵中有大量的零元素，若仍然用常规方法存储，可能存储重复的非零元素或零元素，将造成存储空间的大量浪费。对这类矩阵进行压缩存储：

- 多个相同的非零元素只分配一个存储空间；
- 零元素不分配空间。

对称矩阵

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2n} \\ & \cdots & \cdots & \cdots & \\ a_{n1} & a_{n2} & \cdots & & a_{nn} \end{bmatrix}$$

按行序为主序：

a11	a21	a22	a31	a32	……	an1	……	ann
k=0	1	2	3	4		n(n-1)/2		n(n+1)/2-1

$$k = \begin{cases} i(i-1)/2 + j - 1, & i \ge j \\ j(j-1)/2 + i - 1, & i < j \end{cases}$$

三角矩阵

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

按行序为主序：

a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	……	a_{n1}	……	a_{nn}
k=0	1	2	3	4		n(n-1)/2		n(n+1)/2-1

$$Loc(a_{ij})=Loc(a_{11})+[i(i-1)/2+(j-1)]*l$$

对角矩阵

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

按行序为主序：

a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	……	……	$a_{n(n-1)}$	a_{nn}

k=0　1　2　3　4　　　　　　3(n-1)

$$Loc(a_{ij})=Loc(a_{11})+2(i-1)+(j-1)$$

稀疏矩阵

✧定义：非零元较零元少，且分布没有一定规律的矩阵

✧压缩存储原则：只存矩阵的行列维数和每个非零元的行列下标及其值

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

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

✧稀疏矩阵的压缩存储方法

✧三元组表

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

行列下标

非零元值

	i	j	v
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

ma[0].i,ma[0].j,ma[0].v分别存放矩阵行列维数和非零元个数

三元组表所需存储单元个数为3(t+1)
其中t为非零元个数

三元组顺序表

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

带辅助行向量的二元组表

增加一个辅助数组NRA[m+1]，其物理意义是第i行第一个非零元在二元组表中的起始地址（m为行数）

显然有：

$$\begin{cases} NRA[1]=1 \\ NRA[i]=NRA[i-1]+\text{第}i-1\text{行非零元个数}(i\geq 2) \end{cases}$$

NRA[0]不用或存矩阵行数

	NRA
0	6
1	1
2	3
3	3
4	5
5	6
6	7

列下标和非零元值

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

ma

矩阵列数和非零元个数

二元组表需存储单元个数为2(t+1)+m+1

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

伪地址表示法

伪地址：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

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

伪地址 → addr　　非零元值 → v

	addr	v
0	6	7
1	2	12
2	3	9
3	15	-3
4	20	14
5	24	24
6	30	18
7	36	15
8	39	-7

ma

矩阵行列维数

伪地址表示法需存储单元个数为**2(t+1)**

求转置矩阵

- 问题描述：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
- 问题分析

  一般矩阵转置算法：

```
for(col=0;col<n;col++)
    for(row=0;row<m;row++)
        n[col][row]=m[row][col];
```

$T(n)=O(m\times n)$

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

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

ma

	i	j	v
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

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

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

mb

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

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

ma

	i	j	v
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

mb

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

✡解决思路：只要做到

①将矩阵行、列维数互换

②将每个三元组中的i和j相互调换

③重排三元组次序，使mb中元素以N的行(M的列)为主序

方法一：按M的列序转置

即按mb中三元组次序依次在ma中找到相应的三元组进行转置。为找到M中每一列所有非零元素，需对其三元组表ma从第一行起扫描一遍。由于ma中以M行序为主序，所以由此得到的恰是mb中应有的顺序。

✡算法描述：

✡算法分析：T(n)=O(M的列数n×非零元个数t)

✡若 t 与m×n同数量级，则 $T(n)=O(m\times n^2)$

	i	j	v
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

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

mb

p p p p p p p p

p p p p p p p p

k k k k k

col=1

col=2

```
Status TransposeSMatrix(TSMatrix M, TSMatrix &T) {
    T.mu = M.nu; T.nu = M.mu; T.tu = M.tu;
    if(T.tu) {
        q=1;
        for(col=1; col<=M.nu; ++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;
}
```

方法二：快速转置

即按ma中三元组次序转置，转置结果放入b中恰当位置。此法关键是要预先确定**M中每一列第一个非零元在mb中位置**，为确定这些位置，转置前应先求得**M的每一列中非零元个数。**

实现：设两个数组

num[col]：表示矩阵M中第col列中非零元个数

cpot[col]：指示M中第col列第一个非零元在mb中位置

显然有：

$$\begin{cases} cpot[1]=1; \\ cpot[col]=cpot[col-1]+num[col-1]; \quad (2 \le col \le ma[0].j) \end{cases}$$

	i	j	v
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

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

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

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
	2	4	6			9	
	3	5	7				

ma

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

mb

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

```
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];
    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  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;
}
```

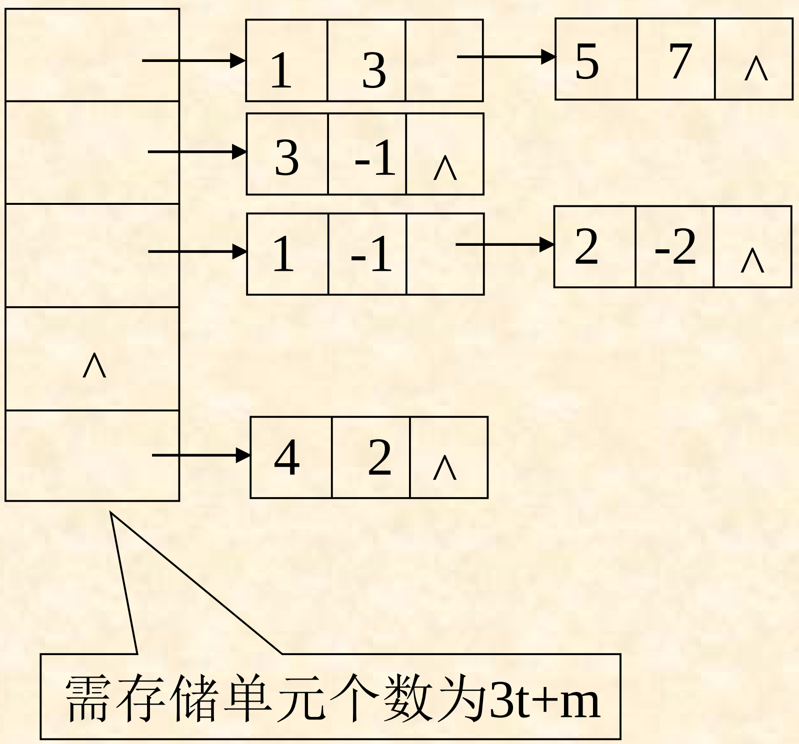
- 算法描述：
- 算法分析：T(n)=O(M的列数n+非零元个数t)
  若 t 与m×n同数量级，则T(n)=O(m×n)

链式存储结构

- ❑ 带行指针向量的单链表表示
- ✡ 每行的非零元用一个单链表存放
- ✡ 设置一个行指针数组，指向本行第一个非零元结点；若本行无非零元，则指针为空

```
typedef  struct node
{    int  col;
     int  val;
     struct node  *link;
}JD;
typedef  struct  node   *TD;
```

$$A=\begin{bmatrix} 3 & 0 & 0 & 0 & 7 \\ 0 & 0 & -1 & 0 & 0 \\ -1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix}$$



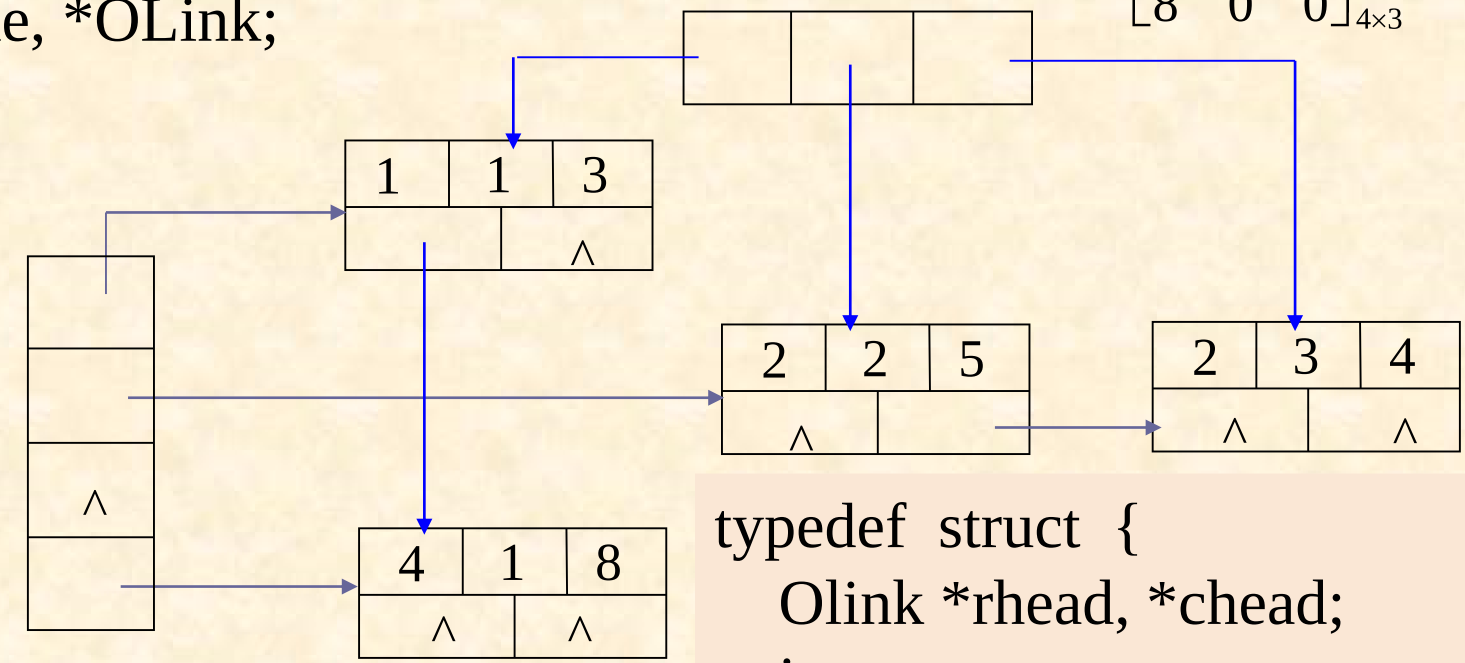
十字链表

- 设行指针数组和列指针数组，分别指向每行、列第一个非零元
- 结点定义

```
typedef  struct OLNode {
    int  i, j;
    ElemType e;
    struct  OLNode  *down, *right;
}OLNode, *OLink;
```

row	col	val
down	right	

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



```
typedef  struct  {
    Olink *rhead, *chead;
    int mu, nu, tu;
}CrossList;
```

✡从键盘接收信息建立十字链表算法
见书P104：算法5.4
任意的非零元输入先后次序
注意：行表和列表的插入操作

✡算法分析：$T(n)=O(t\times s)$
其中：t—非零元个数
$s = \max(m, n)$

m=4,n=3

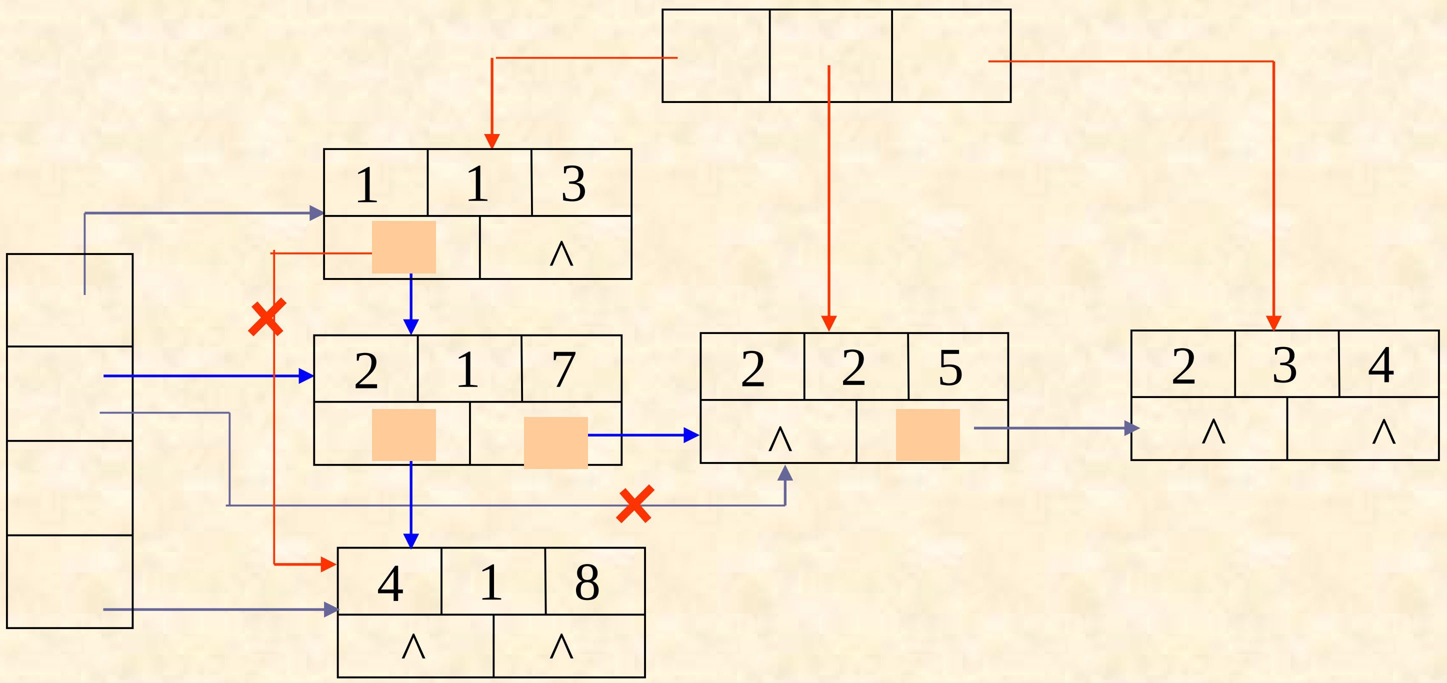
1,1,3

2,2,5

2,3,4

4,1,8

2,1,7



5.4 广义表

- 顾名思义，广义表是线性表的推广，也有人称之为列表(Lists用复数形式以示与统称的表list的区别)。
- LISP语言，把广义表示为基本的数据结构，就连程序也表示为一系列的广义表。

  $LS=(a_1, a_2, \ldots, a_n)$

  其中,LS是广义表$(a_1, a_2, \ldots, a_n)$的名称, n是它的长度。
- 在线性表的定义中， a_i $(1\le i\le n)$只限于是单个元素。而在广义表的定义中， a_i可以是单个元素，也可以是广义表，分别称为广义表LS的原子和子表。
- 习惯上，用大写字母表示广义表的名称，用小写字母表示原子。当广义表LS非空时，称第一个元素a_1为LS的表头(Head)，称其余元素组成的表为LS的表尾(Tail)。

广义表的定义

- 广义表的定义是一个递归的定义，因为在描述广义表时又用到了广义表的概念。

(1) $A=(\ )$——A 是一个空表，它的长度为零。

(2) $B=(e)$——列表 B 只有一个原子 e，B 的长度为 1。

(3) $C=(a,(b,c,d))$——列表 C 的长度为 2，两个元素分别为原子 a 和子表 (b,c,d)。

(4) $D=(A,B,C)$——列表 D 的长度为 3，三个元素都是列表。显然，将子表的值代入后，则有 $D=((\ ),(e),(a,(b,c,d)))$。

(5) $E=(a,E)$——这是一个递归的表，它的长度为 2。E 相当于一个无限的列表 $E=(a,(a,(a,\cdots)))$。

广义表的特征

(1) 列表的元素可以是子表，而子表的元素还可以是子表，…。由此，列表是一个多层次的结构，可以用图形象地表示。

(2) 列表可为其它列表所共享。

(3) 列表可以是一个递归的表，即列表也可以是其本身的一个子表。

- 根据前述对表头、表尾的定义可知：任何一个非空列表其表头可能是原子，也可能是列表．而其表尾必定为列表。
- 值得提醒的是列表()和(())不同。前者为空表．长度n＝0;后者长度n＝1，可分解得到其表头、表尾均为空表()。

广义表的存储结构

- 通常采用链式存储结构
- 每个数据元素可用一个结点表示

tag = 1	*hp*	*tp*

表结点

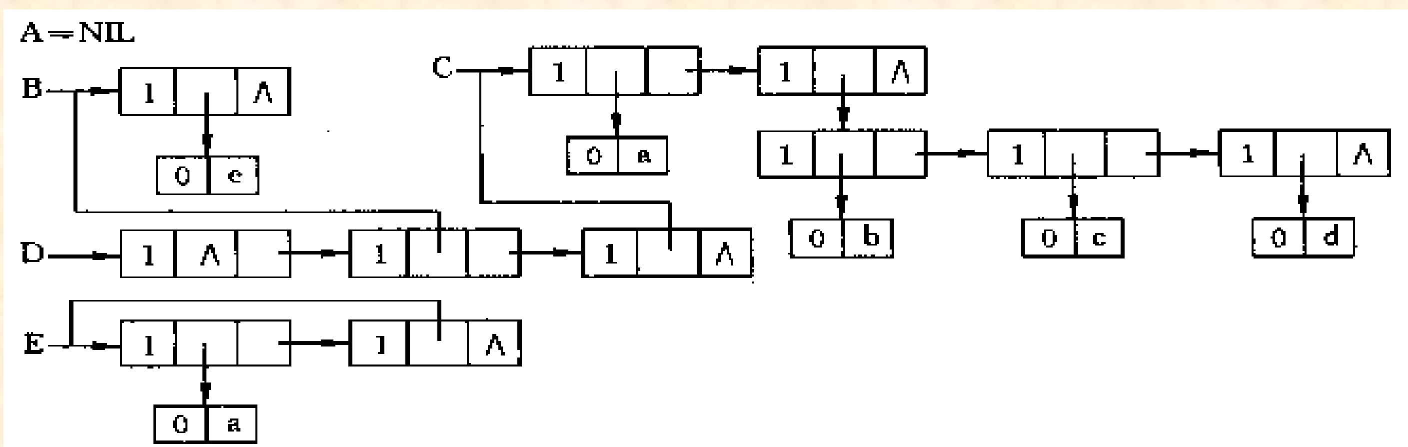
tag = 0	*atom*

原子结点

```
// ------ 广义表的头尾链表存储表示 ------
typedef enum {ATOM, LIST }ElemTag; // ATOM==0:原子,LIST==1:子表
typedef struct GLNode {
    ElemTag  tag;                 // 公共部分，用于区分原子结点和表结点
    union {                       // 原子结点和表结点的联合部分
      AtomType  atom;             // atom 是原子结点的值域，AtomType 由用户定义
      struct {struct GLNode  *hp, *tp;}ptr;
                                  // ptr 是表结点的指针域，ptr.hp 和 ptr.tp
    };                            // 分别指向表头和表尾
} *GList;                         // 广义表类型
```

广义表的存储例1

- 在这种存储结构中有几种情况：
  - (1)除空表的表头指针为空外，对任何非空列表，其表头指针均指向一个表结点，且该结点中的hp域指示列表表头(或为原子结点，或为表结点)，tp域指向列表表尾(除非表尾为空，则指针为空，否则必为表结点)；
  - (2)容易分清列表中原子和子表所在层次。
  - (3)最高层的表结点个数即为列表的长度。



$D=((\ ),(e),(a,(b,c,d)))$。 $E=(a,E)$

广义表的链式存储2

- 扩展的线性表表示

tag = 1	hp	tp

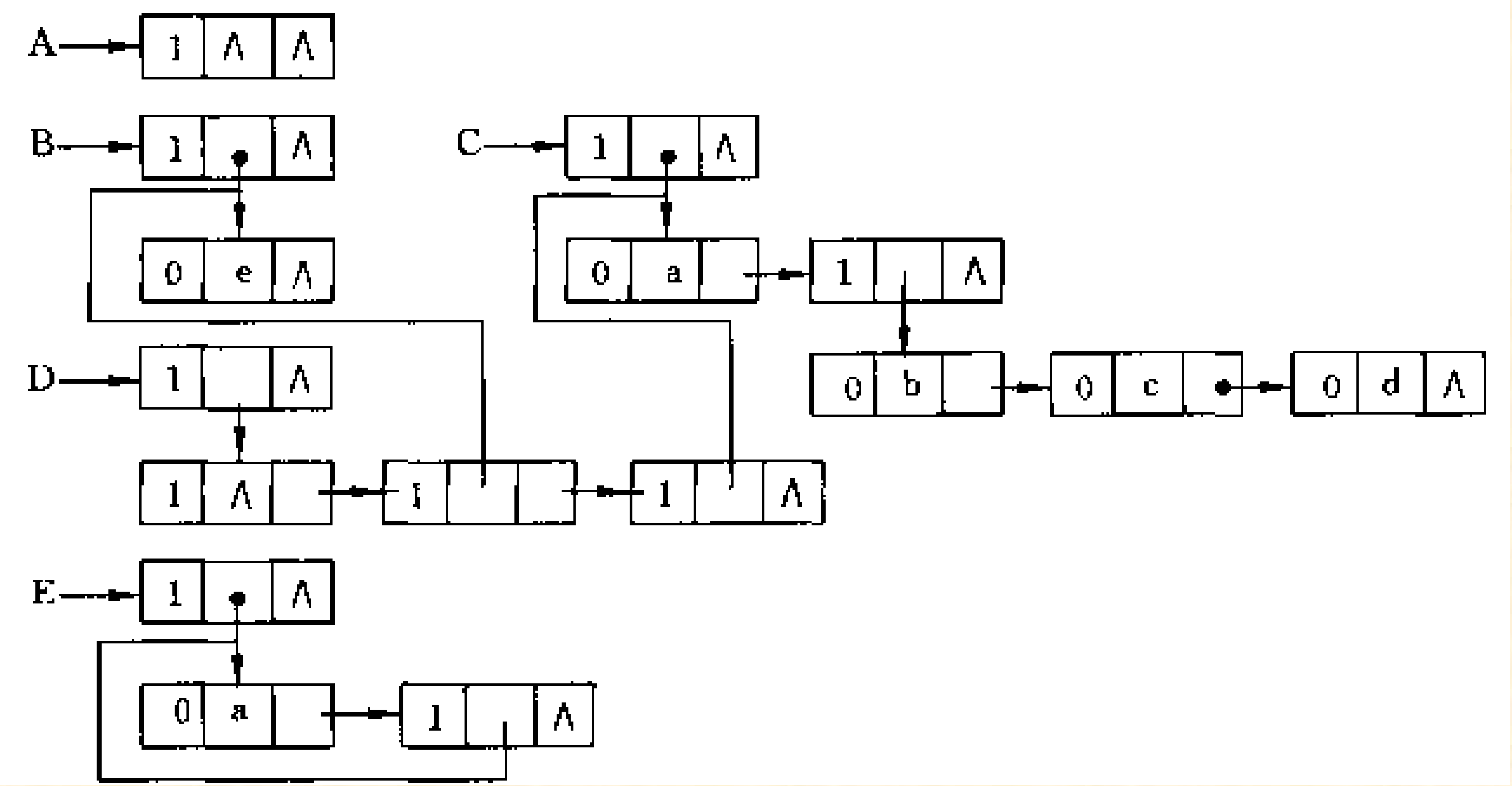
表结点

tag = 0	atom	tp

原子结点

```
// ------ 广义表的扩展线性链表存储表示 ------
typedef enum {ATOM, LIST}ElemTag; // ATOM==0:原子,LIST==1:子表
typedef struct GLNode {
    ElemTag             tag;      // 公共部分，用于区分原子结点和表结点
    union {                       // 原子结点和表结点的联合部分
        AtomType        atom;     // 原子结点的值域
        struct GLNode   *hp;      // 表结点的表头指针
    };
    struct GLNode       *tp;      // 相当于线性链表的next,指向下一个元素结点
} *GList;                         // 广义表类型 GList 是一种扩展的线性链表
```

广义表的存储例2



$D = ((\ ),(e),(a,(b,c,d)))$。

$E = (a, E)$