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：$j_i = 0,1,\ldots,b_i-1$, $1,2,\ldots,n$;

$D = \{ a_{j_1 j_2 \ldots j_n} \mid n>0$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：$R = \{R_1, R_2, \ldots, R_n\}$

$R_i=\{<a_{j_1 j_2 \ldots j_i \ldots j_n} , a_{j_1 j_2 \ldots j_{i+1} \ldots j_n}> \mid 0 \leqq j_k \leqq b_k-1$ ，$1 \leqq k \leqq n$ 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

$A=(\alpha_1, \alpha_2, \ldots, \alpha_p)$　$(p=m或n)$

其中每个数据元素α_j是一个列向量(线性表)：

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

或是一个行向量：

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

如图5-1所示。

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

(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：

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

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：

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

FORTRAN是按列优先顺序存储的，如图5-2(c)。

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

(a) 二维数组的表示形式

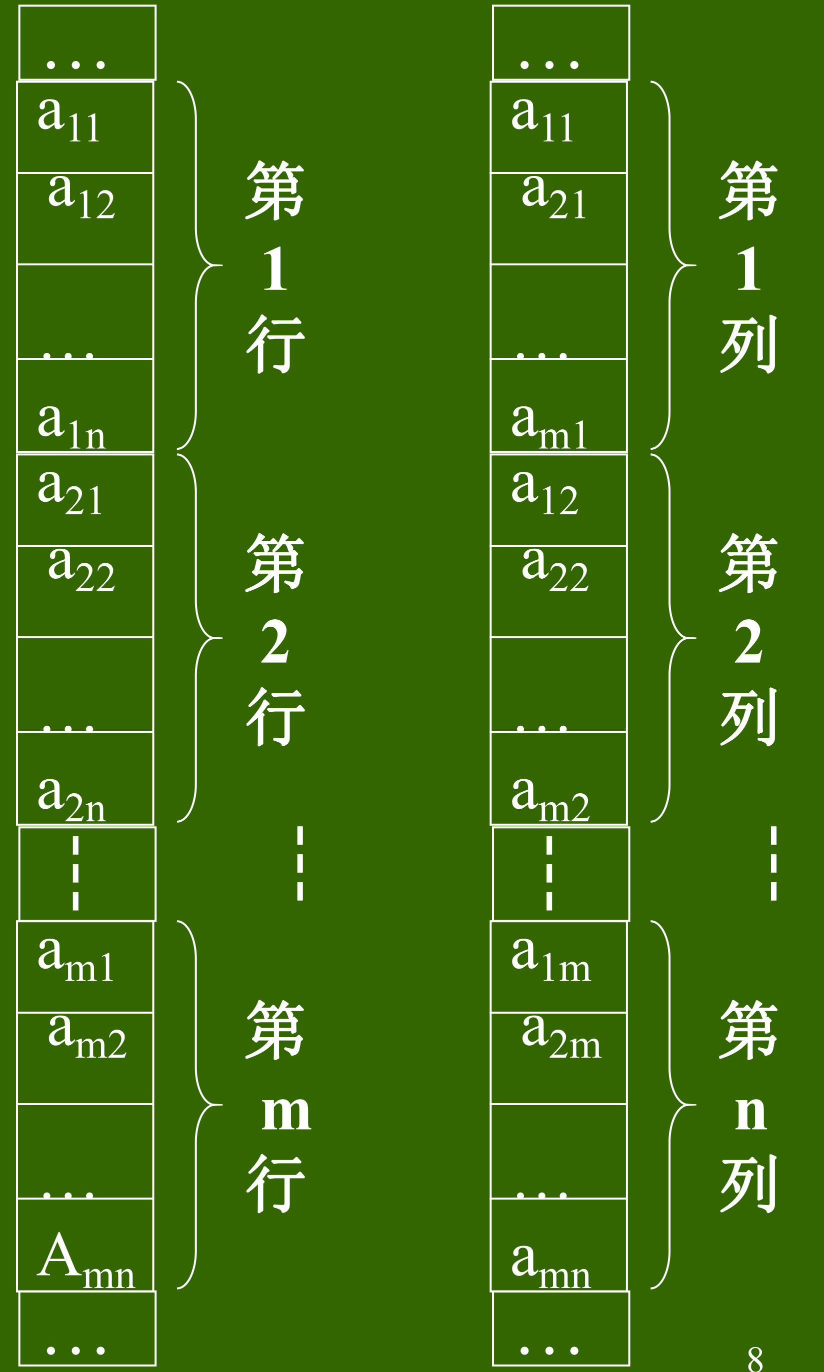


图5-2 二维数组及其顺序存储图例形式

(b) 行优先顺序存储 **(c)** 列优先顺序存储

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：

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

$j=1,2, \ldots,n$

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：

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

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$a_{j_1 j_2 \ldots j_n}$的（首）地址是：

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

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：

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

$j=1,2,\ldots,m$

由此可知，二维数组中任一元素a_{ij}的(首)地址是：

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

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

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。

5.3.1 特殊矩阵

特殊矩阵：是指非零元素或零元素的分布有一定规律的矩阵。

1 对称矩阵

若一个n阶方阵$A=(a_{ij})_{n\times n}$中的元素满足性质：

$$a_{ij}=a_{ji} \quad 1\leqq i,j\leqq n \text{且} i\neq j$$

则称A为对称矩阵，如图5-3所示。

$$A=\begin{pmatrix} 1 & 5 & 1 & 3 & 7 \\ 5 & 0 & 8 & 0 & 0 \\ 1 & 8 & 9 & 2 & 6 \\ 3 & 0 & 2 & 5 & 1 \\ 7 & 0 & 6 & 1 & 3 \end{pmatrix} \qquad A=\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

图5-3 对称矩阵示例

对称矩阵中的元素关于主对角线对称，因此，让每一对对称元素a_{ij}和$a_{ji}(i \neq j)$分配一个存储空间，则n^2个元素压缩存储到$n(n+1)/2$个存储空间，能节约近一半的存储空间。

不失一般性，假设按“行优先顺序”存储下三角形(包括对角线)中的元素。

设用一维数组(向量)sa[0…n(n+1)/2]存储n阶对称矩阵，如图5-4所示。为了便于访问，必须找出矩阵A中的元素的下标值（i,j）和向量sa[k]的下标值k之间的对应关系。

K	1	2	3	4	…		n(n-1)/2		…	n(n+1)/2
sa	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	…	a_{n1}	a_{n2}	…	a_{nn}

图5-4　对称矩阵的压缩存储示例

若$i \geqq j$：a_{ij}在下三角形中，直接保存在sa中。a_{ij}之前的$i-1$行共有元素个数：$1+2+\ldots+(i-1)=i\times(i-1)/2$

而在第i行上，a_{ij}之前恰有$j-1$个元素，因此，元素a_{ij}保存在向量sa中时的下标值k之间的对应关系是：

$$k=i\times(i-1)/2+j-1 \qquad i \geqq j$$

若$i<j$：则a_{ij}是在上三角矩阵中。因为$a_{ij}=a_{ji}$，在向量sa中保存的是a_{ji}。依上述分析可得：

$$k=j\times(j-1)/2+i-1 \qquad i<j$$

对称矩阵元素a_{ij}保存在向量sa中时的下标值k与（i,j）之间的对应关系是：

$$K=\begin{cases} i\times(i-1)/2+j-1 & \text{当}i \geqq j\text{时} \\ j\times(j-1)/2+i-1 & \text{当}i<j\text{时} \end{cases} \qquad 1 \leqq i,j \leqq n \qquad (5\text{-}4)$$

根据上述的下标对应关系，对于矩阵中的任意元素a_{ij}，均可在一维数组sa中唯一确定其位置k；反之，对所有k=1,2, …,n(n+1)/2，都能确定sa[k]中的元素在矩阵中的位置(i,j)。

称sa[0…n(n+1)/2]为n阶对称矩阵A的压缩存储。

2　三角矩阵

以主对角线划分，三角矩阵有上三角和下三角两种。

上三角矩阵的下三角（不包括主对角线）中的元素均为常数c(一般为0)。下三角矩阵正好相反，它的主对角线上方均为常数，如图5-5所示。

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ c & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \\ c & c & \dots & a_{nn} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & c & \dots & c \\ a_{21} & a_{22} & \dots & c \\ \dots & \dots & \dots & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

(a) 上三角矩阵示例　　**(b)** 下三角矩阵示例

图5-5　三角矩阵示例

三角矩阵中的重复元素c可共享一个存储空间，其余的元素正好有n(n+1)/2个，因此，三角矩阵可压缩存储到向量sa[0…n(n+1)/2]中，其中c存放在向量的第1个分量中。

上三角矩阵元素a_{ij}保存在向量sa中时的下标值k与（i,j）之间的对应关系是：

$$K=\begin{cases} i\times(i-1)/2+j-1 & \text{当} i\geqq j \text{时} \\ n\times(n+1)/2 & \text{当} i<j \text{时} \end{cases} \quad 1\leqq i,j\leqq n \qquad (5\text{-}5)$$

下三角矩阵元素$a_{i\,j}$保存在向量sa中时的下标值k与（i,j）之间的对应关系是：

$$K=\begin{cases} i\times(i-1)/2+j-1 & \text{当} i\leqq j \text{时} \\ n\times(n+1)/2 & \text{当} i>j \text{时} \end{cases} \quad 1\leqq i,j\leqq n \qquad (5\text{-}6)$$

3　对角矩阵

矩阵中，除了主对角线和主对角线上或下方若干条对角线上的元素之外，其余元素皆为零。即所有的非零元素集中在以主对角线为了中心的带状区域中，如图5-6所示。

$$
A=\begin{pmatrix}
a_{11} & a_{12} & 0 & \ldots & 0 & & \\
a_{21} & a_{22} & a_{23} & 0 & \ldots & 0 & \\
0 & a_{32} & a_{33} & a_{34} & 0 & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots & & & \\
0 & \ldots & 0 & a_{n-1\ n-2} & a_{n-1\ n-1} & a_{n-1\ n} & \\
0 & \ldots & 0 & 0 & a_{n\ n-1} & a_{n\ n} &
\end{pmatrix}
$$

图5-6　三对角矩阵示例

如上图三对角矩阵，非零元素仅出现在主对角($a_{i\ i}$,1≦i≦n)上、主对角线上的那条对角线($a_{i\ i+1}$,1≦i≦n-1)、主对角线下的那条对角线上($a_{i+1\ i}$,1≦i≦n-1)。显然，当| i-j |>1时，元素a_{ij}=0。

由此可知，一个k对角矩阵（k为奇数）A是满足下述条件：当| i-j |>(k-1)/2时，　$a_{i\ j}$=0

对角矩阵可按行优先顺序或对角线顺序，将其压缩存储到一个向量中，并且也能找到每个非零元素和向量下标的对应关系。

仍然以三对角矩阵为例讨论。

当i=1，j=1、2，或i=n， j=n-1、n或

1<i<n-1,j=i-1、i、i+1的元素a_{ij}外，其余元素都是0。

对这种矩阵，当以按“行优先顺序”存储时， 第1行和第n行是2个非零元素，其余每行的非零元素都要是3个，则需存储的元素个数为3n-2。

K	1	2	3	4	5	6	7	8	…	3n-3	3n-2
sa	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	…	$a_{n\,n-1}$	a_{nn}

图5-7 三对角矩阵的压缩存储示例

如图5-7所示三对角矩阵的压缩存储形式。数组sa中的元素sa[k]与三对角矩阵中的元素a_{ij}存在一一对应关系，在a_{ij}之前有i-1行, 共有3×i-1个非零元素，在第i行，有j-i+1个非零元素，这样，非零元素a_{ij}的地址为：

$$LOC[a_{i\,j}] = LOC[a_{11}] + [3\times i-1+(j-i+1)]\times l$$

$$= LOC[a_{11}]+(2\times i+j)\times l$$

上例中，a_{34}对应着sa[10]， k=2×i+j=2×3+4=10

称sa[0…3×n-2]是n阶三对角矩阵A的压缩存储。

上述各种特殊矩阵，其非零元素的分布都是有规律的，因此总能找到一种方法将它们压缩存储到一个向量中，并且一般都能找到矩阵中的元素与该向量的对应关系，通过这个关系，仍能对矩阵的元素进行随机存取。

5.3.2 稀疏矩阵

稀疏矩阵(Sparse Matrix)：对于稀疏矩阵，目前还没有一个确切的定义。设矩阵A是一个n×m的矩阵中有s个非零元素，设 $\delta=s/(n\times m)$，称δ为稀疏因子，如果某一矩阵的稀疏因子δ满足δ≦0.05时称为稀疏矩阵，如图5-8所示。

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 24 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

图5-8 稀疏矩阵示例

5.3.2.1 稀疏矩阵的压缩存储

对于稀疏矩阵，采用压缩存储方法时，只存储非0元素。必须存储非0元素的行下标值、列下标值、元素值。因此，一个三元组(i, j, a_{ij})唯一确定稀疏矩阵的一个非零元素。

如图5-8的稀疏矩阵A的三元组线性表为：

((1,2,12), (1,3,9), (3,1,-3), (3,8,4), (4,3,24), (5,2,18), (6,7,-7), (7,4,-6))

1 三元组顺序表

若以行序为主序，稀疏矩阵中所有非0元素的三元组，就可以得构成该稀疏矩阵的一个三元组顺序表。

1　三元组顺序表

若以行序为主序，稀疏矩阵中所有非0元素的三元组，就可以得构成该稀疏矩阵的一个三元组顺序表。相应的数据结构定义如下：

(1) 三元组结点定义

```
#define MAX_SIZE 101
typedef int elemtype ;
typedef struct
  {   int   row ;       /*  行下标  */
      int  col ;        /*  列下标  */
      elemtype value;       /*  元素值  */
  }Triple ;
```

(2) 三元组顺序表定义

```
typedef struct
  {  int  rn ;         /*  行数  */
     int  cn ;         /*  列数  */
     int  tn ;         /*  非0元素个数  */
     Triple  data[MAX_SIZE] ;
  }TMatrix ;
```

图5-8所示的稀疏矩阵及其相应的转置矩阵所对应的三元组顺序表如图5-9所示。

7	rn行数	
8	cn列数	
9	tn元素个数	
1	2	12
1	3	9
3	1	-3
3	8	4
4	3	24
4	6	2
5	2	18
6	7	-7
7	4	-6
row	col	value

(a) 原矩阵的三元组表

8	rn行数	
7	cn列数	
9	tn元素个数	
1	3	-3
2	1	12
2	5	18
3	1	9
3	4	24
4	7	-6
6	4	2
7	6	-7
8	2	4
row	col	value

(b)转置矩阵的三元组表

图5-9 稀疏矩阵及其转置矩阵的三元组顺序表

矩阵的运算包括矩阵的转置、矩阵求逆、矩阵的加减、矩阵的乘除等。在此，先讨论在这种压缩存储结构下的求矩阵的转置的运算。

一个m×n的矩阵A，它的转置B是一个n×m的矩阵，且b[i][j]=a[j][i]，0≦i≦n，0≦j≦m，即B的行是A的列，B的列是A的行。

设稀疏矩阵A是按行优先顺序压缩存储在三元组表a.data中，若仅仅是简单地交换a.data中i和j的内容，得到三元组表b.data，b.data将是一个按列优先顺序存储的稀疏矩阵B，要得到按行优先顺序存储的b.data，就必须重新排列三元组表b.data中元素的顺序。

求转置矩阵的基本算法思想是：

① 将矩阵的行、列下标值交换。即将三元组表中的行、列位置值i 、j相互交换；

② 重排三元组表中元素的顺序。即交换后仍然是按行优先顺序排序的。

方法一：

算法思想：按稀疏矩阵A的三元组表a.data中的列次序依次找到相应的三元组存入b.data中。

每找转置后矩阵的一个三元组，需从头至尾扫描整个三元组表a.data 。找到之后自然就成为按行优先的转置矩阵的压缩存储表示。

按方法一求转置矩阵的算法如下：

```
void TransMatrix(TMatrix a , TMatrix b)
  {  int p , q , col ;
     b.rn=a.cn ;  b.cn=a.rn ;  b.tn=a.tn ;
       /*   置三元组表b.data的行、列数和非0元素个数 */
     if  (b.tn==0)    printf(" The Matrix A=0\n" );
     else
       {   q=0;
          for  (col=1; col<=a.cn ; col++)
               /*  每循环一次找到转置后的一个三元组 */
          for  (p=0 ;p<a.tn ; p++)
               /*  循环次数是非0元素个数  */
```

```
            if  (a.data[p].col==col)
                {  b.data[q].row=a.data[p].col ;
                   b.data[q].col=a.data[p].row ;
                   b.data[q].value=a.data[p].value;
                   q++ ;
                }
        }
}
```

算法分析：本算法主要的工作是在p和col的两个循环中完成的，故算法的时间复杂度为O(cn×tn)，即矩阵的列数和非0元素的个数的乘积成正比。

而一般传统矩阵的转置算法为：

```
for(col=1; col<=n ;++col)
    for(row=0 ; row<=m ;++row)
        b[col][row]=a[row][col] ;
```

其时间复杂度为$O(n\times m)$。当非零元素的个数tn和$m\times n$同数量级时，算法TransMatrix的时间复杂度为$O(m\times n^2)$。

由此可见，虽然节省了存储空间，但时间复杂度却大大增加。所以上述算法只适合于稀疏矩阵中非0元素的个数tn远远小于$m\times n$的情况。

方法二(快速转置的算法)

算法思想：直接按照稀疏矩阵A的三元组表a.data的次序依次顺序转换，并将转换后的三元组放置于三元组表b.data的恰当位置。

前提：若能预先确定原矩阵A中每一列的(即B中每一行)第一个非0元素在b.data中应有的位置，则在作转置时就可直接放在b.data中恰当的位置。因此，应先求得A中每一列的非0元素个数。

附设两个辅助向量num[ ]和cpot[ ] 。

- num[col]：统计A中第col列中非0元素的个数；
- cpot[col] ：指示A中第一个非0元素在b.data中的恰当位置。

显然有位置对应关系：

$$\begin{cases} cpot[1]=1 \\ cpot[col]=cpot[col-1]+num[col-1] \qquad 2 \leqq col \leqq a.cn \end{cases}$$

例图5-8中的矩阵A和表5-9(a)的相应的三元组表可以求得num[col]和cpot[col]的值如表5-1：

表5-1　num[col]和cpot[col]的值表

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

快速转置算法如下：

```
void  FastTransMatrix(TMatrix a, TMatrix b)
 {    int p , q , col , k ;
      int num[MAX_SIZE] , copt[MAX_SIZE] ;
      b.rn=a.cn ; b.cn=a.rn ; b.tn=a.tn ;
          /*   置三元组表b.data的行、列数和非0元素个数   */
      if  (b.tn==0)    printf(“ The Matrix A=0\n” ) ;
      else
          {  for (col=1 ; col<=a.cn ; ++col)    num[col]=0 ;
               /*  向量num[]初始化为0   */
             for (k=1 ; k<=a.tn ; ++k)
                 ++num[ a.data[k].col] ;
                   /*   求原矩阵中每一列非0元素个数    */
```

```
for (cpot[0]=1, col=2 ; col<=a.cn ; ++col)
      cpot[col]=cpot[col-1]+num[col-1] ;
        /*  求第col列中第一个非0元在b.data中的序号  */
for (p=1 ; p<=a.tn ; ++p)
    {   col=a.data[p].col ;  q=cpot[col] ;
        b.data[q].row=a.data[p].col ;
        b.data[q].col=a.data[p].row ;
         b.data[q].value=a.data[p].value ;
         ++cpot[col] ;      /*至关重要!!当本列中   */
     }
  }
}
```

2　行逻辑链接的三元组顺序表

将上述方法二中的辅助向量cpot[ ]固定在稀疏矩阵的三元组表中，用来指示“行”的信息。得到另一种顺序存储结构：行逻辑链接的三元组顺序表。其类型描述如下：

```
#define MAX_ROW 100
typedef struct
  { Triple data[MAX_SIZE] ;    /*  非0元素的三元组表  */
    int     rpos[MAX_ROW];    /* 各行第一个非0位置表 */
    int    rn ,cn , tn ;      /*  矩阵的行、列数和非0元个数  */
  }RLSMatrix ;
```

稀疏矩阵的乘法

设有两个矩阵：$A=(a_{ij})_{m\times n}$， $B=(b_{ij})_{n\times p}$

则： $C=(c_{ij})_{m\times p}$ 其中 $c_{ij}=\sum a_{ik}\times b_{kj}$

$1\leqq k\leqq n$， $1\leqq i\leqq m$， $1\leqq j\leqq p$

经典算法是三重循环：

```
for  ( i=1 ; i<=m ; ++i)
    for ( j=1 ; j<=p ; ++j)
        {   c[i][j]=0 ;
            for ( k=1 ; k<=n ; ++k)
                c[i][j]= c[i][j]+a[i][k]×b[k][j];
        }
```

此算法的复杂度为$O(m\times n\times p)$。

设有两个稀疏矩阵$A=(a_{ij})_{m\times n}$， $B=(b_{ij})_{n\times p}$， 其存储结构采用行逻辑链接的三元组顺序表。

算法思想：对于A中的每个元素a.data[p](p=1, 2, … , a.tn)，找到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].value×b.data[q].value，该乘积是c_{ij}中的一部分。求得所有这样的乘积并累加求和就能得到c_{ij}。

为得到非0的乘积，只要对a.data[1…a.tn] 中每个元素(i，k，a_{ik})(1≦i≦a.rn，1≦k≦a.cn) ，找到b.data中所有相应的元素(k，j，b_{kj})(1≦k≦b.rn，1≦j≦b.cn)相乘即可。则必须知道矩阵B中第k行的所有非0元素，而b.rpos[ ]向量中提供了相应的信息。

b.rpos[row]指示了矩阵B的第row行中第一个非0元素在b.data[ ]中的位置(序号)，显然，b.rpos[row+1]-1指示了第row行中最后一个非0元素在b.data[ ]中的位置(序号)。最后一行中最后一个非0元素在b.data[ ]中的位置显然就是b.tn 。

两个稀疏矩阵相乘的算法如下：

void MultsMatrix(RLSMatrix a, RLSMatrix b, RLSMatrix c)

```
    /* 求矩阵A 、B的积C=A×B，采用行逻辑链接的顺序表 */
  {   elemtype  ctemp[Max_Size] ;
      int  p , q , arow , ccol , brow , t ;
      if  (a.cn!=b.rn)   {   printf("Error\n") ; exit(0);  }
```

```
else
  {  c.rn=a.rn ; c.cn=b. n ; c.tn=0 ;    /* 初始化C  */
   if (a.tn*b.tn!=0)       /* C  是非零矩阵  */
   {   for (arow=1 ; arow<=a.rn ; ++arow)
        {   ctemp[arow]=0 ;  /*  当前行累加器清零   */
            c.rpos[arow]=c.tn+1; p=a.rops[arow];
            for ( ; p<a.rpos[arow+1];++p)
               /* 对第arow行的每一个非0元素   */
              {   brow=a.data[p].col ;
                   /*   找到元素在b.data[]中的行号  */
                  if (brow<b.cn) t=( b.rpos[brow+1];
                  else t=b.tn+1 ;
```

```
        for (q=b.rpos[brow] ; q<t ; ++q)
            {  ccol=b.data[q].col ;
                /*  积元素在c中的列号  */
    ctemp[ccol]+=a.data[p].value*b.data[q].value ;
            }
    }    /* 求出c中第arow行中的非0元素   */
for (ccol=1 ; ccol<=c.cn ; ++ccol)
    if ( ctemp[ccol] !=0 )
      {   if ( ++c.tn>MAX_SIZE)
            {   printf("Error\n") ; exit(0);   }
        else
```

```
        c.data[c.tn]=(arow , ccol , ctemp[ccol]) ;
          }
        }
    }
}
```

3　十字链表

对于稀疏矩阵，当非0元素的个数和位置在操作过程中变化较大时，采用链式存储结构表示比三元组的线性表更方便。

矩阵中非0元素的结点所含的域有：行、列、值、行指针(指向同一行的下一个非0元)、列指针(指向同一列的下一个非0元)。其次，十字交叉链表还有一个头结点，结点的结构如图5-10所示。

<table>
<tr><td>row</td><td>col</td><td>value</td></tr>
<tr><td colspan="2">down</td><td>right</td></tr>
</table>

(a)　结点结构

<table>
<tr><td>rn</td><td>cn</td><td>tn</td></tr>
<tr><td colspan="2">down</td><td>right</td></tr>
</table>

(b)　头结点结构

图5-10　十字链表结点结构

由定义知，稀疏矩阵中同一行的非0元素的由right指针域链接成一个行链表， 由down指针域链接成一个列链表。则每个非0元素既是某个行链表中的一个结点，同时又是某个列链表中的一个结点，所有的非0元素构成一个十字交叉的链表。称为十字链表。

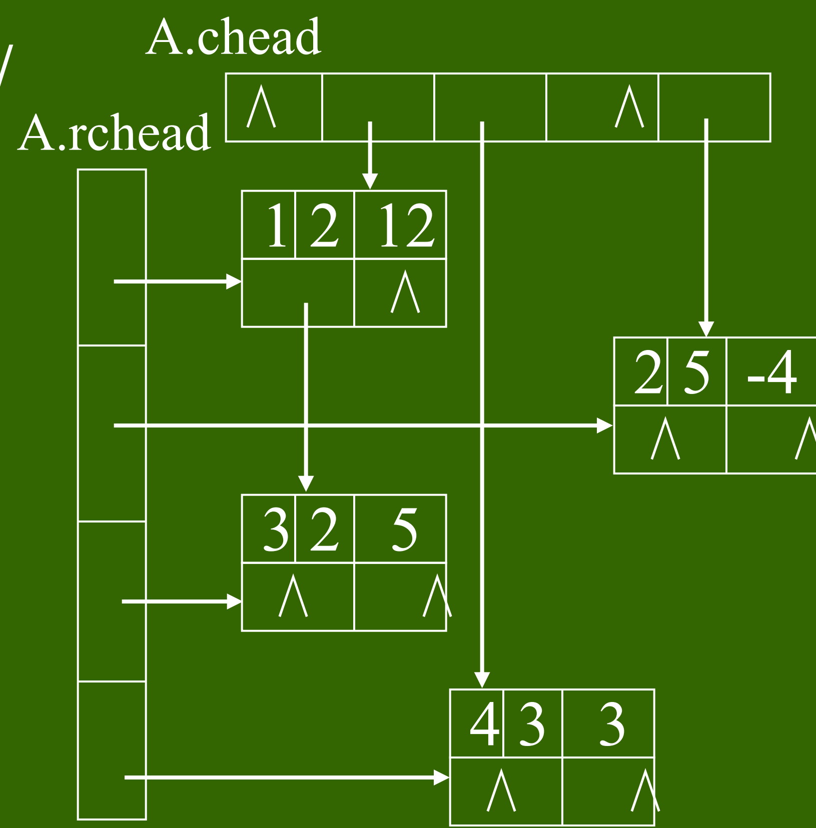
此外，还可用两个一维数组分别存储行链表的头指针和列链表的头指针。对于图5-11(a)的稀疏矩阵A，对应的十字交叉链表如图5-11(b)所示，结点的描述如下：

```
typedef struct  Clnode
  {   int  row , col ;  /*  行号和列号  */
      elemtype value ;   /*  元素值  */
      struct  Clnode  *down , *right ;
  }OLNode ;  /*  非0元素结点  */
```

```
typedef struct  Clnode
  {   int   rn;        /*  矩阵的行数  */
      int   cn;        /*  矩阵的列数  */
      int   tn;        /*  非0元素总数  */
      OLNode *rhead ;
      OLNode *chead ;
  } CrossList ;
```

$$A=\begin{pmatrix} 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{pmatrix}$$

(a) 稀疏矩阵



(b) 稀疏矩阵的十字交叉链表

图5-11 稀疏矩阵及其十字交叉链表

5.4 广义表

广义表是线性表的推广和扩充，在人工智能领域中应用十分广泛。

在第2章中，我们把线性表定义为n(n≧0)个元素a_1, a_2 ,…, a_n的有穷序列，该序列中的所有元素具有相同的数据类型且只能是原子项(Atom)。所谓原子项可以是一个数或一个结构，是指结构上不可再分的。若放松对元素的这种限制，容许它们具有其自身结构，就产生了广义表的概念。

广义表(Lists，又称为列表)：是由n(n ≧0)个元素组成的有穷序列： LS=(a_1，a_2，…，a_n)

其中a_i或者是原子项，或者是一个广义表。LS是广义表的名字，n为它的长度。若a_i是广义表，则称为LS的子表。

习惯上：原子用小写字母，子表用大写字母。

若广义表LS非空时：

- a_1(表中第一个元素)称为表头；
- 其余元素组成的子表称为表尾；(a_2，a_3，…，a_n)
- 广义表中所包含的元素(包括原子和子表)的个数称为表的长 度。
- 广义表中括号的最大层数称为表深 (度)。

有关广义表的这些概念的例子如表5-2所示。

表5-2 广义表及其示例

广义表	表长n	表深h
A=()	0	0
B=(e)	1	1
C=(a,(b,c,d))	2	2
D=(A,B,C)	3	3
E=(a,E)	2	∞
F=(())	1	2

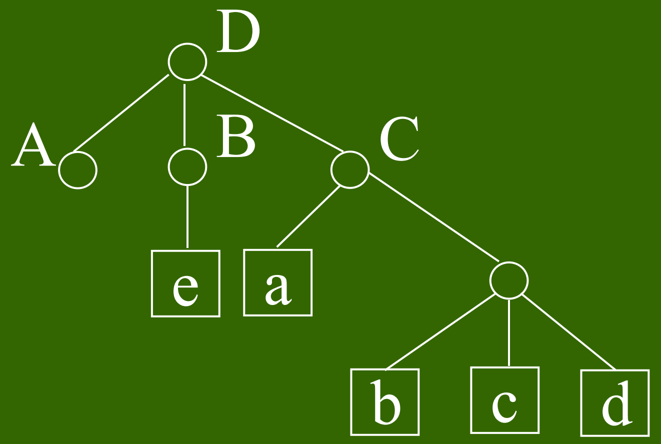


图5-12 广义表的图形表示

广义表的重要结论：

⑴ 广义表的元素可以是原子，也可以是子表，子表的元素又可以是子表， …。即广义表是一个多层次的结构。

表**5-2**中的广义表**D**的图形表示如图**5-12**所示。

(2) 广义表可以被其它广义表所共享，也可以共享其它广义表。广义表共享其它广义表时通过表名引用。

(3) 广义表本身可以是一个递归表。

(4) 根据对表头、表尾的定义，任何一个非空广义表的表头可以是原子，也可以是子表， 而表尾必定是广义表。

5.4.1 广义表的存储结构

由于广义表中的数据元素具有不同的结构，通常用链式存储结构表示，每个数据元素用一个结点表示。因此，广义表中就有两类结点：

- 一类是表结点，用来表示广义表项，由标志域，表头指针域，表尾指针域组成;
- 另一类是原子结点，用来表示原子项，由标志域，原子的值域组成。如图5-13所示。

只要广义表非空，都是由表头和表尾组成。即一个确定的表头和表尾就唯一确定一个广义表。

标志tag=0	原子的值

(a) 原子结点

标志tag=1	表头指针hp	表尾指针tp

(b) 表结点

图5-13 广义表的链表结点结构示意图

相应的数据结构定义如下：

```
typedef struct GLNode
  {  int   tag ;     /*  标志域，为1：表结点；为0 ：原子结点   */
     union
       {  elemtype value;     /* 原子结点的值域  */
          struct
            {  struct GLNode  *hp , *tp ;
             }ptr ;   /*  ptr和atom两成员共用    */
        }Gdata ;
  } GLNode ;       /* 广义表结点类型  */
```

例： 对A=()，B=(e)，C=(a, (b, c, d))，D=(A, B, C)，E=(a, E)的广义表的存储结构如图5-14所示。

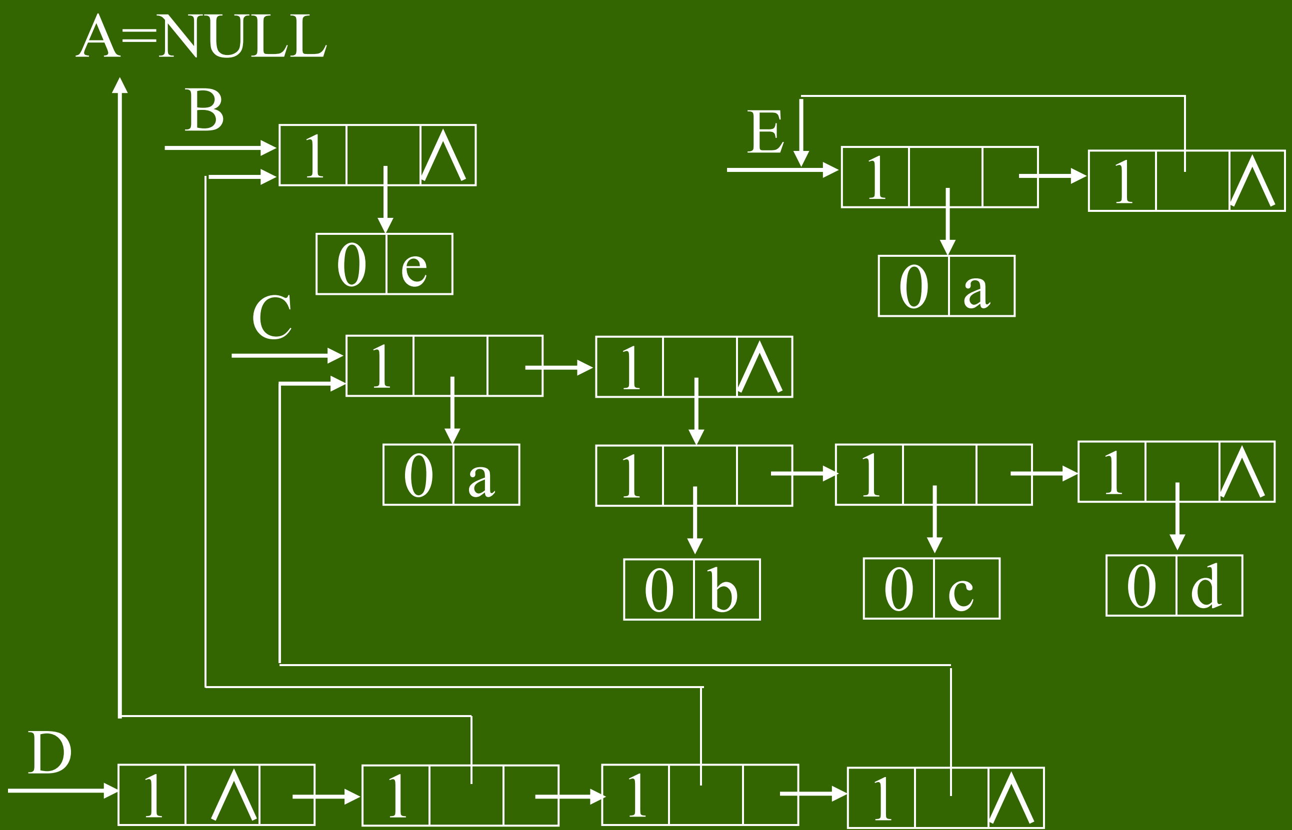


图5-14 广义表的存储结构示意图

对于上述存储结构，有如下几个特点：

(1) 若广义表为空，表头指针为空；否则，表头指针总是指向一个表结点，其中hp指向广义表的表头结点（或为原子结点，或为表结点），tp指向广义表的表尾(表尾为空时，指针为空，否则必为表结点)。

(2) 这种结构求广义表的长度、深度、表头、表尾的操作十分方便。

(3) 表结点太多，造成空间浪费。也可用图5-15所示的结点结构。

tag=0	原子的值	表尾指针tp

(a) 原子结点

tag=1	表头指针hp	表尾指针tp

(b) 表结点

图5-15 广义表的链表结点结构示意图

习 题 五

⑴ 什么是广义表？请简述广义表与线性表的区别？

⑵ 一个广义表是(a, (a, b), d, e, (a, (i, j), k))，请画出该广义表的链式存储结构。

⑶ 设有二维数组a[6][8]，每个元素占相邻的4个字节，存储器按字节编址，已知a的起始地址是1000，试计算：

① 数组a的最后一个元素a[5][7]起始地址；

② 按行序优先时，元素a[4][6]起始地址；

③ 按行序优先时，元素a[4][6]起始地址。

(4) 设A和B是稀疏矩阵，都以三元组作为存储结构，请写出矩阵相加的算法，其结果存放在三元组表C中，并分析时间复杂度。

(5) 设有稀疏矩阵B如下图所示，请画出该稀疏矩阵的三元组表和十字链表存储结构。

$$A=\begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 5 & 0 \\ 0 & 0 & -3 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$