数据结构

北京邮电大学 信息安全中心
武 斌 杨 榆

上章内容

上一章（串）内容：

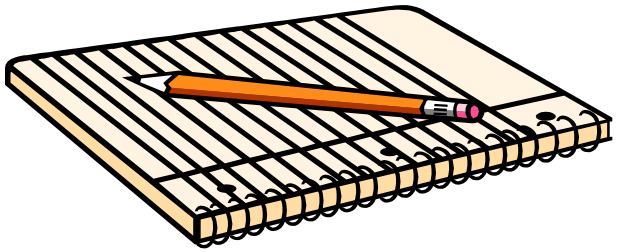
- 理解“串”类型定义中各基本操作的特点
- 能正确利用这些特点进行串的其它操作
- 理解串类型的各种存储表示方法
- 理解串匹配的各种算法

本次课程学习目标

学习完本次课程，您应该能够：

- 理解数组类型的特点及其在高级编程语言中的存储表示和实现方法
- 掌握数组在"以行为主"的存储表示中的地址计算方法
- 掌握特殊矩阵的存储压缩表示方法
- 理解稀疏矩阵的两类存储压缩方法的特点及其适用范围
- 掌握广义表的结构特点及其存储表示方法

本章课程内容（第五章 数组和广义表）



- 5.1 数组的类型定义
- 5.2 数组的顺序表示和实现
- 5.3 矩阵的压缩存储
- 5.4 广义表的定义
- 5.5 广义表的存储结构

第五章 数组和广义表

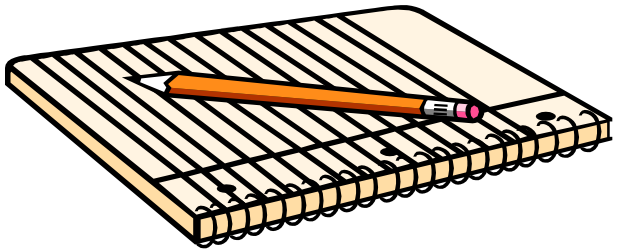
- **数组**是所有高级编程语言中都已实现的固有数据类型，因此凡学习过高级程序设计语言的读者对数组都不陌生。但它和其它诸如整数、实数等原子类型不同，它是一种结构类型。换句话说，“数组”是一种数据结构。

- C语言中的一维数组和二维数组

```
int a[10];

float a[3][5];
```

数组的类型定义



- 5.1 数组的类型定义
- 5.2 数组的顺序存储表示
- 5.3 矩阵的压缩存储
- 5.4 广义表的定义
- 5.5 广义表的存储结构

数组的类型定义

●看一个二维数组的简单情况：

$D = \{ a_{i,j} \mid 0 \le i \le m-1, 0 \le j \le n-1, \ a_{i,j} \in ElemType \}$

$R = \{ ROW, COL \}$

其中：

$ROW = \{ \langle a_{i-1,j}, a_{i,j} \rangle \mid i=1,\ldots,m-1, 0 \le j \le n-1,$

$a_{i-1,j}, a_{i,j} \in ElemType\}$ (称作“行关系”，同一列相邻两行元素间关系

$COL = \{ \langle a_{i,j-1}, a_{i,j} \rangle \mid j=1,\ldots,n-1, 0 \le i \le m-1,$

$a_{i,j-1}, a_{i,j} \in ElemType\}$ (称作“列关系”，同一行相邻两列元素间关系)

数组的类型定义

- 上述定义的二维数组中共有 m×n 个元素，每个元素 $a_{i,j}$ 同时处于“行”和“列”的两个关系中，它既在“行关系ROW”中是 $a_{i-1,j}$ (i>0)的后继，又在“列关系COL”中是 $a_{i,j-1}$ (j>0)的后继。

- 数组的类型定义如下：

ADT Array {

数据对象： $D=\{ a_{j_1,j_2,\ldots,j_i,\ldots j_N} \mid j_i = 0,\ldots, b_i-1, i=1,2,..,N,$

称 N(>0) 为数组的维数，b_i 为数组第 i 维的长度，

j_i 为数组元素的第i维下标，$a_{j_1,j_2,\ldots,j_i,\ldots j_N} \in ElemSet \}$

数据关系： $R=\{ R_1, R_2, \ldots, R_N \}$

$R_i=\{ <a_{j_1,j_2,\ldots,j_i,\ldots j_N}, a_{j_1,j_2,\ldots,j_i+1,\ldots j_N}> \mid$

$0\le j_k\le b_k-1,\ 1\le k\le N$ 且$k\neq i,\ \ 0\le j_i\le b_i-2,\ a_{j_1,\ldots,j_i,\ldots,j_N},\ a_{j_1,\ldots,j_i+1,\ldots,j_N} \in D,$ $i=2,\ldots,N \}$

数组的类型定义

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。

DestroyArray (**&**A)

初始条件：数组 A 已经存在。

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：若下标不超界，则将 e 的值赋给A中指定下标的元素。

} ADT Array

数组的类型定义

- 例如，对于如下矩阵表示的数组而言：

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \\ a_{20} & a_{21} \end{bmatrix}$$

数据对象：$D=\{a_{j_1,j_2} \mid j_i = 0,\ldots, b_i-1, i=1,2\}=\{a_{00}, a_{01}, a_{10}, a_{11}, a_{20}, a_{21}\} =\{a_{00}, a_{10}, a_{20}, a_{01}, a_{11}, a_{21}\}$数组维数N为2，第1维长度$b_1$ 为3，第2维长度b_2 为2。

数据关系：$R=\{ R_1, R_2 \}$

$R_1=\{ <a_{j_1,j_2}, a_{j_1+1,j_2}> \mid 0\le j_2\le b_2-1, 0\le j_1 \le b_1-2, a_{j_1,j_2}, a_{j_1+1,j_2} \in D\}=\{<a_{00}, a_{10}>, <a_{10}, a_{20}> ,<a_{01}, a_{11}>, <a_{11}, a_{21}>\}$

$R_2=\{ <a_{j_1,j_2}, a_{j_1,j_2+1}> \mid 0\le j_1\le b_1-1, 0\le j_2 \le b_2-2, a_{j_1,j_2}, a_{j_1,j_2+1} \in D\}=\{<a_{00}, a_{01}>, <a_{10}, a_{11}>, <a_{20}, a_{21}>\}$

数组的类型定义

- 已知数组定义如下

$$A_{2\times3}=\begin{bmatrix}a_{1,1} & a_{1,2} & a_{1,3}\\ a_{2,1} & a_{2,2} & a_{2,3}\end{bmatrix}$$

请写出该数组数据关系$R=\{R_1,R_2\}$

解：$A_{2\times3}$是一个2维数组，第1维长度为2，第2维长度为3，

$$R_1=\{\langle a_{1,1},a_{2,1}\rangle,\langle a_{1,2},a_{2,2}\rangle,\langle a_{1,3},a_{2,3}\rangle\}$$

$$R_2=\{\langle a_{1,1},a_{1,2}\rangle,\langle a_{1,2},a_{1,3}\rangle,\langle a_{2,1},a_{2,2}\rangle,\langle a_{2,2},a_{2,3}\rangle\}$$

已知$A_{2\times2\times3}=\begin{bmatrix}A_{2\times3}^{(1)} & A_{2\times3}^{(2)}\end{bmatrix}$, $A_{2\times3}^{(j)}=\begin{bmatrix}a_{j,1,1} & a_{j,1,2} & a_{j,1,3}\\ a_{j,2,1} & a_{j,2,2} & a_{j,2,3}\end{bmatrix}$, $j=1,2$

解：$A_{2\times3}^{(1)}=\begin{bmatrix}a_{1,1,1} & a_{1,1,2} & a_{1,1,3}\\ a_{1,2,1} & a_{1,2,2} & a_{1,2,3}\end{bmatrix}$, $A_{2\times3}^{(1)}=\begin{bmatrix}a_{2,1,1} & a_{2,1,2} & a_{2,1,3}\\ a_{2,2,1} & a_{2,2,2} & a_{2,2,3}\end{bmatrix}$

数组的类型定义

- 已知数组定义如下

已知$A_{2\times2\times3}=\left[A_{2\times3}^{(1)}\quad A_{2\times3}^{(2)}\right]$, $A_{2\times3}^{(j)}=\begin{bmatrix}a_{j,1,1} & a_{j,1,2} & a_{j,1,3}\\ a_{j,2,1} & a_{j,2,2} & a_{j,2,3}\end{bmatrix}$, $j=1,2$

解：$A_{2\times3}^{(1)}=\begin{bmatrix}a_{1,1,1} & a_{1,1,2} & a_{1,1,3}\\ a_{1,2,1} & a_{1,2,2} & a_{1,2,3}\end{bmatrix}$, $A_{2\times3}^{(1)}=\begin{bmatrix}a_{2,1,1} & a_{2,1,2} & a_{2,1,3}\\ a_{2,2,1} & a_{2,2,2} & a_{2,2,3}\end{bmatrix}$

$$R=\{R_1,R_2,R_3\}$$

$$R_1=\left\{\begin{matrix}\langle a_{1,1,1},a_{2,1,1}\rangle,\langle a_{1,1,2},a_{2,1,2}\rangle,\langle a_{1,1,3},a_{2,1,3}\rangle,\\ \langle a_{1,2,1},a_{2,2,1}\rangle,\langle a_{1,2,2},a_{2,2,2}\rangle,\langle a_{1,2,3},a_{2,2,3}\rangle\end{matrix}\right\}$$

$$R_2=\left\{\begin{matrix}\langle a_{1,1,1},a_{1,2,1}\rangle,\langle a_{1,1,2},a_{1,2,2}\rangle,\langle a_{1,1,3},a_{1,2,3}\rangle,\\ \langle a_{2,1,1},a_{2,2,1}\rangle,\langle a_{2,1,2},a_{2,2,2}\rangle,\langle a_{2,1,3},a_{2,2,3}\rangle\end{matrix}\right\}$$

$$R_3=\left\{\begin{matrix}\langle a_{1,1,1},a_{1,1,2}\rangle,\langle a_{1,1,2},a_{1,1,3}\rangle,\langle a_{1,2,1},a_{1,2,2}\rangle,\langle a_{1,2,2},a_{1,2,3}\rangle,\\ \langle a_{2,1,1},a_{2,1,2}\rangle,\langle a_{2,1,2},a_{2,1,3}\rangle,\langle a_{2,2,1},a_{2,2,2}\rangle,\langle a_{2,2,2},a_{2,2,3}\rangle\end{matrix}\right\}$$

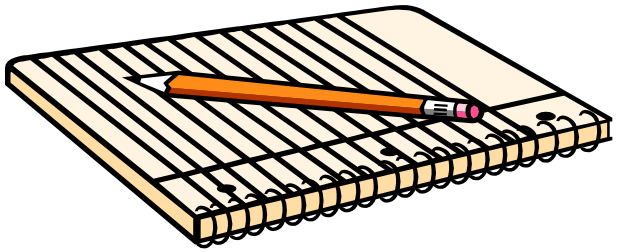
数组的类型定义

- 和线性表类似，数组中的每个元素都对应于一组下标$(j_1,j_2,\ldots j_N)$，每个下标的取值范围是$0\le j_i \le b_i-1$，称 b_i 为第 i 维的长度$(i=1,2,\ldots,N)$。因此，也可以**将数组看成是由“一组下标”和“元素值”构成的二元组的集合**。
- 每个元素$a_{j_1, j_2\cdots j_N}$ 都受着N个关系的约束。每个关系中，元素$a_{j_1, j_i\cdots j_N}$（$0<= j_i <= b_i-2$ ）都有一个直接后继元素$a_{j_1, j_i+1\cdots j_N}$。因此，就单个而言，每个关系仍然是线性关系。
- 需要**注意**的是，在此给出的数组定义中，各维下标的下限均约定为常量 0，这只是C语言的限定。在一般情况下，数组每一维的下标 j_i 的取值范围均可设置为任意值的整数。

数组的类型定义

- 数组一旦被定义，其维数(N)以及每一维的上、下界均不能再变，数组中元素之间的关系也不再改变。
- 因此，数组的基本操作除初始化和结构销毁之外，**只有通过给定的"一组下标"索引取得元素或修改元素的值**。

数组的顺序存储表示



- 5.1 数组的类型定义
- 5.2 数组的顺序存储表示
- 5.3 矩阵的压缩存储
- 5.4 广义表的定义
- 5.5 广义表的存储结构

数组的顺序存储表示

- 由于数组类型不做插入和删除的操作，因此以“**顺序映象**”作为数组的存储结构是合理选择。
- 由于数组中的数据元素之间是一个“多维”的关系，而存储地址是一维的，因此数组的顺序存储表示要解决的是一个“**如何用一维的存储地址来表示多维的关系**”的问题。
- 通常有两种映象方法：即“**以行(序)为主(序)**”的映象方法和“**以列(序)为主(序)**”的映象方法。

数组的顺序存储表示

- 例如，二维数组：

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

- 行优先顺序——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。以二维数组为例。
- 先固定第1维下标为1，枚举第2维所有下标，得到第一行元素线性序列为：$a_{11},a_{12},\ldots,a_{1n}$；
- 再固定第1维下标为2，枚举第2维所有下标，得到第二行元素线性序列为：$a_{21},a_{22},\ldots,a_{2n}$；
- 依次类推，最后固定第1维下标为m，枚举第2维所有下标，得到第m行元素线性序列为：$a_{m1},a_{m2},\ldots,a_{mn}$；

数组的顺序存储表示

●例如，二维数组：

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

- 行优先顺序——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。以二维数组为例。
- 因此，按行优先顺序存储的线性序列为：

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

- 以行序优先的元素枚举方法可描述为从第2维到第1维依次枚举。
- 在PASCAL、C语言中，数组就是按行优先顺序存储的。

数组的顺序存储表示

●例如，二维数组：

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

- 列优先顺序——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。以二维数组为例。
- 先固定第2维下标为1，枚举第1维所有下标，得到第一列元素线性序列为：$a_{11},a_{21},\ldots,a_{n1}$；
- 再固定第2维下标为2，枚举第1维所有下标，得到第二列元素线性序列为：$a_{12},a_{22},\ldots,a_{n2}$；
- 依次类推，最后固定第2维下标为n，枚举第1维所有下标，得到第n列元素线性序列为：$a_{1n},a_{2n},\ldots,a_{mn}$；

数组的顺序存储表示

●例如，二维数组：

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

- 列优先顺序——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。以二维数组为例。
- 因此，按列优先顺序存储的线性序列为：

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

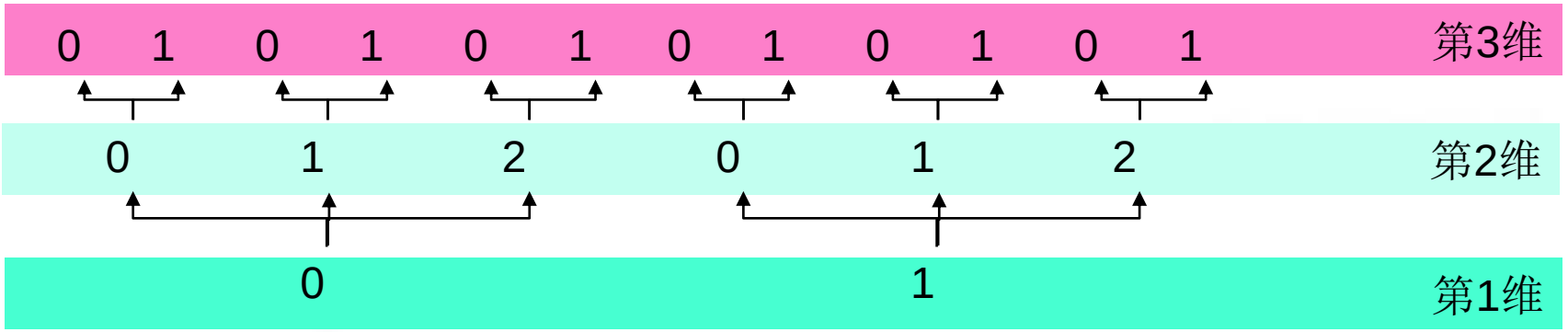
- 以列序优先的元素枚举方法可描述为从第1维到第2维依次枚举。
- 在FORTRAN语言中，数组就是按列优先顺序存储的。

数组的顺序存储表示

- 以上规则可以推广到**多维数组**的情况。对于N维数组$A_{b_1\cdots b_i\cdots b_N}=[a_{j_1\cdots j_i\cdots j_N}](0<=j_i<b_i)$而言，行优先顺序可规定为从第$j_N$维向第$j_1$维依次枚举的排序方式；列优先顺序与此相反，可规定为从第j_1维向第j_N维依次枚举的排序方式。

- 例如：三维数组$A_{2,3,2}$按行优先顺序存储的线性序列为：

$a_{000}, a_{001}, a_{010}, a_{011}, a_{020}, a_{021}, a_{100}, a_{101}, a_{110}, a_{111}, a_{120}, a_{121},$



数组的顺序存储表示

- 以上规则可以推广到**多维数组**的情况。对于N维数组$A_{b_1\cdots b_i\cdots b_N}=[a_{j_1\cdots j_i\cdots j_N}](0<=j_i<b_i)$而言，行优先顺序可规定为从第$j_N$维向第$j_1$维依次枚举的排序方式；列优先顺序与此相反，可规定为从第j_1维向第j_N维依次枚举的排序方式。

- 例如：四维数组$A_{1,3,1,2}$按行优先顺序存储的线性序列为：

a_{0000}, a_{0001}, a_{0100}, a_{0101}, a_{0200}, a_{0201},

- N维数组$A_{b_1\cdots b_i\cdots b_N}$按行优先顺序,元素下标枚举顺序示意如下：

$$j_1\begin{cases}0\longrightarrow j_2\begin{cases}0\longrightarrow \cdots j_i\begin{cases}0\longrightarrow \cdots j_N\begin{cases}0(a_{0\ldots00})\\ \cdots\\ b_N-1(a_{0\ldots0,b_{N-1}})\end{cases}\\ \cdots\\ b_i-1\end{cases}\\ \cdots\\ b_2-1\end{cases}\\ \cdots\\ b_1-1\end{cases}$$

$a_{0\ldots00},\ldots,a_{0\ldots0,b_{N-1}},a_{0\ldots1,0},\ldots,a_{0\ldots1,b_{N-1}},\ldots$

数组的顺序存储表示

- 按上述两种方式顺序存储的数组，只要知道开始元素的存放地址（即基地址），维数(N)，每维长度(b_i)，以及每个数组元素所占用的单元数，就可以根据下标($j_1,j_2,\ldots j_N$)计算出数组任意元素的存放地址。
- 可见，访问**数组中任一元素不依赖于其它元素，即顺序存储的数组是一个随机存取结构**。

数组的顺序存储表示

- 假设二维数组 R[m][n] 中每个数据元素占L个存储地址，并以 LOC(i, j) 表示下标为 (i, j) 的数据元素的存储地址，则该数组中任何一对下标 (i, j) 对应的数据元素

  - 在“以行为主”的顺序映象中的存储地址为：

第0行				第i-1行			第i行					
a_{00}	…	$a_{0,n-1}$	…	$a_{i-1,0}$	…	$a_{i-1,n-1}$	$a_{i,0}$	…	$a_{i,j}$	…	$a_{i,n-1}$	…
n个元素				n个元素			第i行$a_{i,j}$前有j个元素					

  - 可见，第0…i-1共i行存满，每行n个元素，所以有i*n个元素；第i行，第j列之前存储0…j-1共j个元素，因此$a_{i,j}$前共(i*n+j)个元素

    - **LOC (i, j) = LOC(0,0) + (i×n+j) × L**

数组的顺序存储表示

- 假设二维数组 R[m][n] 中每个数据元素占L个存储地址，并以 LOC(i, j) 表示下标为 (i, j) 的数据元素的存储地址，则该数组中任何一对下标 (i, j) 对应的数据元素

  - 在“以列为主”的顺序映象中的存储地址为：

第0列				第j-1列			第j列					
a_{00}	…	$a_{m-1,0}$	…	$a_{0,j-1}$	…	$a_{m-1,j-1}$	$a_{0,j}$	…	$a_{i,j}$	…	$a_{m-1,j}$	…
m个元素				m个元素			第j列$a_{i,j}$前有i个元素					

  - 可见，第0…j-1共j列存满，每列m个元素，所以有j*m个元素；第j列，第i列之前存储0…i-1共i个元素，因此$a_{i,j}$前共(j*m+i)个元素

    - **LOC (i, j) = LOC(0,0) + (j×m+i) ×L**

数组的顺序存储表示

- 类似地，假设三维数组 R[p][m][n] 中每个数据元素占 L 个存储地址，并以 LOC(i,j,k) 表示下标为(i,j,k) 的数据元素的存储地址，则该数组中任何一对下标(i,j,k) 对应的数据元素在“以行为主”的顺序映象中的存储地址为：
- 分析：
- 首先存放{(j_1,j_2, j_3)| j_1=0,j_2=0,…,m-1, j_3=0,…,n-1}所对应的元素，共有m×n个；
- 然后再存放{(j_1,j_2, j_3)| j_1=1,j_2=0,…,m-1, j_3=0,…,n-1}所对应的元素，也有m×n个；
- 直到{(j_1,j_2, j_3)| j_1=i-1,j_2=0,…,m-1, j_3=0,…,n-1}。
- 可见j_1=0,1,…,i-1，每个取值对应m×n个元素,共有i×m×n个元素。

数组的顺序存储表示

- 类似地，假设三维数组 R[p][m][n] 中每个数据元素占 L 个存储地址，并以 LOC(i,j,k) 表示下标为(i,j,k) 的数据元素的存储地址，则该数组中任何一对下标(i,j,k) 对应的数据元素在“以行为主”的顺序映象中的存储地址为：
- 分析：
- 接下来存放{$(j_1,j_2, j_3)| j_1=i,j_2=0, j_3=0,\ldots,n-1$}所对应的元素，共有n个；
- 再存放{$(j_1,j_2, j_3)| j_1=i,j_2=1, j_3=0,\ldots,n-1$}所对应的元素，也有n个；
- 直到{$(j_1,j_2, j_3)| j_1=i,j_2=j-1, j_3=0,\ldots,n-1$}。
- 可见$j_1=i,j_2=0,1,\ldots,j-1$，每个取值对应n个元素,共有$j\times n$个元素。
- 最后存放 {$(j_1,j_2, j_3)| j_1=i,j_2=j, j_3=0,\ldots,k-1$}对应元素，共k个。

数组的顺序存储表示

- 类似地，假设三维数组 R[p][m][n] 中每个数据元素占 L 个存储地址，并以 LOC(i,j,k) 表示下标为(i,j,k) 的数据元素的存储地址，则该数组中任何一对下标(i,j,k) 对应的数据元素在“以行为主”的顺序映象中的存储地址为：

j_1=0		j_1=i-1		j_1=i, j_2=0		j_1=i, j_2=j-1		j_1=i, j_2=j
a_{000} ⋮ $a_{0,m-1,n-1}$	⋮	$a_{i-1,0,0}$ ⋮ $a_{i-1,m-1,n-1}$	⋮	$a_{i,0,0}$ ⋮ $a_{i,0,n-1}$	⋮	$a_{i,j-1,0}$ ⋮ $a_{i,j-1,n-1}$		$a_{i,j,0}$ … $a_{i,j,k-1}$
m×n个元素		m×n个元素		n个元素		n个元素		k个元素

➔ **LOC (i,j,k) = LOC(0,0,0) + (i×m×n + j×n+k)×L**

数组的顺序存储表示

- 推广到 N 维数组，要计算LOC(j_1,j_2,..., j_N)。
- 第1维取0...j_1-1任一值时，第2到N维完全枚举有$b_2 \times ... \times b_N$个元素，共$b_2 \times ... \times b_N \times j_1$个元素；
- 固定第1维取值为j_1，第2维取0...j_2-1任一值时，第3到N维完全枚举有$b_3 \times ... \times b_N$个元素，共$b_3 \times ... \times b_N \times j_2$个元素；
- 以此类推，固定第1到N-1维取值分别为j_1...j_{N-1}，第N维取0...j_N-1共有j_N个元素；
- 则得到
  - $LOC(j_1,j_2,..., j_N) = LOC(0,0,...,0) + (b_2 \times ... \times b_N \times j_1 + b_3 \times ... \times b_N \times j_2 + ... + b_N \times j_{N-1} + j_N) \times L$
  - 可缩写成：

数组的顺序存储表示

- 推广到 N 维数组，则得到
  - $LOC(j_1,j_2,\ldots,j_N) = LOC(0,0,\ldots,0) + (b_2\times\ldots\times b_n\times j_1 + b_3\times\ldots\times b_n\times j_2 +\ldots+b_n\times j_{N-1}+j_N)\times L$
  - 可缩写成：

$$LOC(j_1,j_2,\ldots,j_N) = LOC(0,0,0) + \left(\sum_{i=1}^{N-1} j_i \prod_{k=i+1}^{N} b_k + j_N\right)\times L$$

$$LOC(j_1,j_2,\ldots,j_N) = LOC(0,0,0) + \sum_{i=1}^{N} c_i j_i$$

$$c_N = L, c_{i-1} = b_i \times c_i, 1 < i < N$$

$$c_N = L, c_{i-1} = b_i \times c_i, 1 < i < N$$

$$LOC(j_1, j_2,\ldots, j_N) = LOC(0,0,\ldots,0) + \sum_{i=1}^{N} c_i j_i$$

数组的顺序存储表示

$$LOC(j_1, j_2, ..., j_N) = LOC(0,0,...,0) + \sum_{i=1}^{N} c_i j_i$$

- 称这个地址映象公式为**N维数组的映象函数**。
- 容易看出，数组元素的存储位置是其下标的线性函数，一旦确定了数组各维的长度，c_i就是常数。
- 由于计算各个元素存储位置的时间相等，所以**存取数组中任一元素的时间**也相等。我们称具有这一特点的存储结构为**随机存储结构**。

数组的顺序存储表示

$$c_N = L, c_{i-1} = b_i \times c_i, 1 < i < N$$

$$LOC(j_1, j_2, \ldots, j_N) = LOC(0,0,\ldots,0) + \sum_{i=1}^{N} c_i j_i$$

- 例：已知4维数组$A_{3\times6\times5\times4}$起始元素存储地址LOC(0,0,0,0)，请问若以行序为主序，元素$a_{1,5,2,3}$的存储地址为多少？每个元素占用L个字节。

解：$c_4 = L, c_3 = c_4 \times b_4 = 4 \times L = 4L, c_2 = c_3 \times b_3 = 4L \times 5 = 20L, c_1 = c_2 \times b_2 = 20L \times 6 = 120L,$

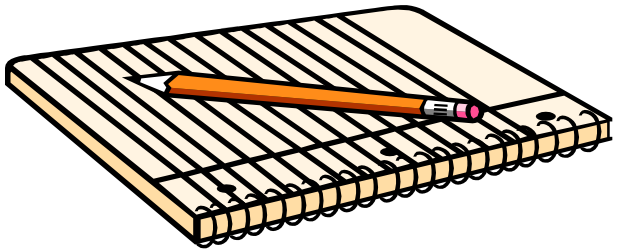
$$LOC(1,5,2,3) = LOC(0,0,0,0) + c_1 \times 1 + c_2 \times 5 + c_3 \times 2 + c_4 \times 3$$

$$= LOC(0,0,0,0) + 120L \times 1 + 20L \times 5 + 4L \times 2 + L \times 3$$

$$= LOC(0,0,0,0) + 120L + 100L + 8L + 3L$$

$$= LOC(0,0,0,0) + 231L$$

矩阵的压缩存储



- 5.1 数组的类型定义
- 5.2 数组的顺序存储表示
- 5.3 矩阵的压缩存储
- 5.4 广义表的定义
- 5.5 广义表的存储结构

矩阵的压缩存储

- 矩阵是数值程序设计中经常用到的数学模型，它是由 m 行和 n 列的数值构成（m=n 时称为方阵）。在用高级语言编制的程序中，通常用二维数组表示矩阵，它使矩阵中的每个元素都可在二维数组中找到相对应的存储位置。
- 然而在数值分析的计算中经常出现一些有下列特性的高阶矩阵，同时矩阵中有很多值相同的元或零值元，**为了节省存储空间，需要对它们进行“压缩存储”，即不存或少存这些值相同的元或零值元**。

矩阵的压缩存储

- 一、特殊矩阵的压缩存储方法

所谓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，下面我们讨论几种特殊矩阵的压缩存储。

➔ 1、对称矩阵

在一个n阶方阵A中，若元素满足下述性质：

$$a_{ij}=a_{ji} \quad 0 \leq i,j \leq n-1$$

则称A为对称矩阵；右图即为一个5阶对称矩阵。

$$\begin{bmatrix} 1 & 5 & 1 & 3 & 7 \\ 5 & 0 & 8 & 0 & 0 \\ 1 & 8 & 9 & 2 & 6 \\ 3 & 0 & 2 & 5 & 1 \\ 7 & 0 & 6 & 1 & 3 \end{bmatrix}$$

矩阵的压缩存储

- 对称矩阵中的元素关于主对角线对称，故只要存储矩阵中上三角或下三角中的元素，让每两个对称的元素共享一个存储空间，这样，能节约近一半的存储空间。不失一般性，我们按“行优先顺序”存储主对角线（包括对角线）以下的元素，其存储形式如下图所示：

a_{00}

a_{10} a_{11}

a_{20} a_{21} a_{22}

……

$a_{n-1\,0}$ $a_{n-1\,1}$ $a_{n-1\,2}$ …$a_{n-1\,n-1}$

- 在这个下三角矩阵中，第i行恰有i+1个元素，元素总数为：n(n+1)/2
- 我们按行优先顺序将这些元素存放在一个向量 $sa[0.....\frac{n(n+1)}{2}-1]$ 中。为了便于访问对称矩阵A中的元素，我们必须在a_{ij}和sa[k]之间找一个对应关系。

矩阵的压缩存储

- 若i≥j，则a_{ij}在下三角形中。 a_{ij}之前的i行（从第0行到第i-1行）一共有1+2+…+i=i(i+1)/2个元素，在第i行上， a_{ij}之前恰有j个元素（即$a_{i,0},a_{i,1},a_{i,2},\ldots,a_{i,j-1}$），因此有：
  - **$k=i\times(i+1)/2+j$　　$0\leqq k<n(n+1)/2$**
- 若i<j，则a_{ij}是在上三角矩阵中。因为$a_{ij}=a_{ji}$，所以只要交换上述对应关系式中的i和j即可得到：
  - $k=j\times(j+1)/2+i$　　$0\leqq k<n(n+1)/2$
- 令 I=max(i,j)， J=min(i,j)，则k和 i, j的对应关系可统一为：
  - $k=I\times(I+1)/2+J$　　$0\leqq k<n(n+1)/2$

矩阵的压缩存储

- 因此，a_{ij}的地址可用下列式计算：

  $LOC(a_{ij})=LOC(sa[k])$

  $=LOC(sa[0])+k\times L=LOC(sa[0])+[\ I\times\ (I+1)/2+J\ ]\ \times L$

  - 有了上述的下标交换关系，对于任意给定一组下标(i，j)，均可在sa[k]中找到矩阵元素a_{ij}，反之，对所有的k=0,1,2,… n(n+1)/2-1，都能确定sa[k]中的元素在矩阵中的位置(i,j)。由此，称sa[n(n+1)/2]为阶对称矩阵A的压缩存储，见下图：

a_{00}	a_{10}	a_{11}	a_{20}	……	$a_{n-1\ 0}$	…	$a_{n-1,n-1}$
k=0	1	2	3		n(n-1)/2	…	n(n+1)/2-1

- 例如：a_{21}和a_{12}均存储在 sa[4]中，这是因为

  - $k=I\times(I+1)/2+J=2\times(2+1)/2+1=4$

矩阵的压缩存储

➔ 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。上三角矩阵如图(a)所示，它的下三角（不包括主对角线）中的元素均为常数。下三角矩阵正好相反，它的主对角线上方均为常数，如图(b)所示。在大多数情况下，三角矩阵常数为零。

$$\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0\,n-1} \\ c & a_{11} & \cdots & a_{1\,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ c & c & \cdots & a_{n-1\,n-1} \end{pmatrix}$$

(a)上三角矩阵

$$\begin{pmatrix} a_{00} & c & \cdots & c \\ a_{10} & a_{11} & \cdots & c \\ \cdots & \cdots & \cdots & \cdots \\ a_{n-1\,0} & a_{n-1\,1} & \cdots & a_{n-1\,n-1} \end{pmatrix}$$

(b)下三角矩阵

矩阵的压缩存储

- 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**最后一个分量**中。
  - 上三角矩阵中，主对角线之上的第p行(0≤p<n)恰有n-p个元素，按行优先顺序存放上三角矩阵中的元素a_{ij}时，a_{ij}之前0…i-1共i行，对应n…n-(i-1)个元素，利用等差数列求和公式，可得，共有i(2n-i+1)/2个元素。在第i行上，a_{ii}前恰好有i-i个元素：$a_{i,i},a_{i,i+1},\ldots a_{i,j-1}$。因此，sa[k]和$a_{ij}$的对应关系是：

$$k=\begin{cases} i(2n-i+1)/2+j-i & \text{当}i\leq j \\ n(n+1)/2 & \text{当}i>j \end{cases}$$

$$\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0\,n-1} \\ c & a_{11} & \cdots & a_{1\,n-1} \\ & & \cdots & \\ c & c & \cdots & a_{n-1\,n-1} \end{pmatrix}$$

  - 下三角矩阵的存储和对称矩阵类似，sa[k]和a_{ij}对应关系是：

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

$$\begin{pmatrix} a_{00} & c & \cdots & c \\ a_{10} & a_{11} & \cdots & c \\ & & \cdots & \\ a_{n-1\,0} & a_{n-1\,1} & \cdots & a_{n-1\,n-1} \end{pmatrix}$$

矩阵的压缩存储

➔ 3、对角矩阵

对角矩阵中，所有的非零元素集中在以主对角线为中心的带状区域中，即**除了主对角线和主对角线相邻两侧的若干条对角线上的元素之外，其余元素皆为零**。下图给出了一个三对角矩阵：

$$
\begin{pmatrix}
a_{00} & a_{01} & & & & \\
a_{10} & a_{11} & a_{12} & & & \\
 & a_{21} & a_{22} & a_{23} & & \\
 & & \cdots & \cdots & \cdots & \\
 & & a_{n-2\,n-3} & a_{n-2\,n-2} & a_{n-2\,n-1} & \\
 & & & a_{n-1\,n-2} & a_{n-1\,n-1}
\end{pmatrix}
$$

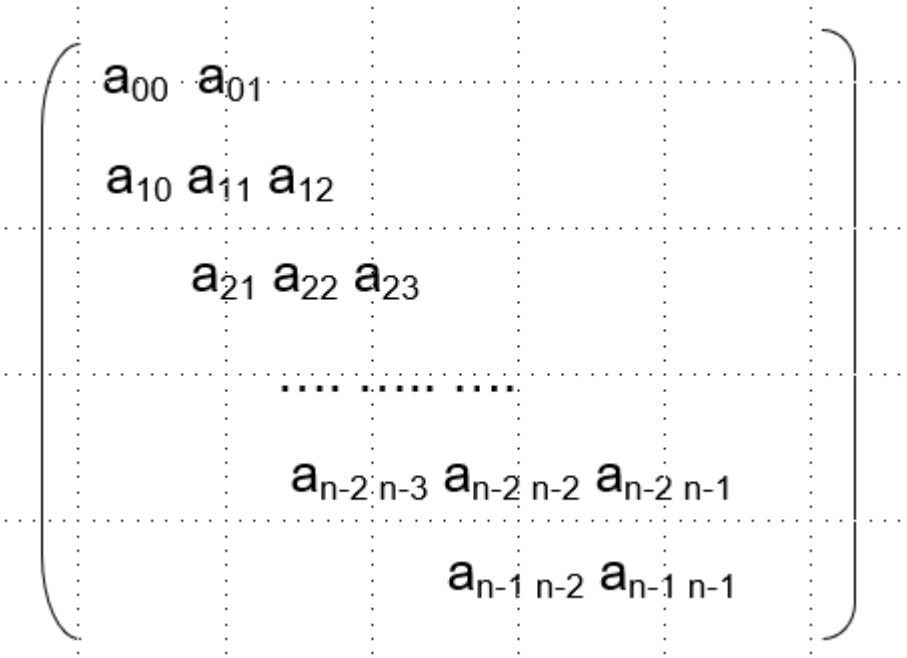
矩阵的压缩存储

- 非零元素仅出现在主对角上(a_{ii},0≦i≦n-1)，紧邻主对角线上面的那条对角线上($a_{i\ i+1}$,0≦i≦n-2)和紧邻主对角线下面的那条对角线上($a_{i+1\ i}$,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是满足下述条件的矩阵：
  - 若|i-j|>(k-1)/2，则元素a_{ij}=0。

$$\begin{pmatrix} a_{00} & a_{01} & & & & \\ a_{10} & a_{11} & a_{12} & & & \\ & a_{21} & a_{22} & a_{23} & & \\ & & \cdots & \cdots & \cdots & \\ & & & a_{n-2\,n-3} & a_{n-2\,n-2} & a_{n-2\,n-1} \\ & & & & a_{n-1\,n-2} & a_{n-1\,n-1} \end{pmatrix}$$

矩阵的压缩存储

- 对角矩阵可按行优先顺序或对角线的顺序，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。
  - 例如：在3对角矩阵里除满足条件i=0，j=0、1，或i=n-1，j=n-2、n-1 或1 ≦ i<n-1，j=i-1、i、i+1的元素a_{ij}外，其余元素都是零。
  - 对这种矩阵，我们也可按行优序为主序来存储。除第0行和第n-1行是2个元素外，每行的非零元素都要是3个，因此，需存储的元素个数为3n-2。



矩阵的压缩存储

a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	…	…	$a_{n-1\ n-2}$	$a_{n-1\ n-1}$
k=0	1	2	3	4	5	…	…	3n-2	3n-1

- 数组sa中的元素sa[k]与三对角带状矩阵中的元素a_{ij}存在一一对应关系，在a_{ij}之前有i行,第0行存2个元素，其他i-1行存3个元素，因此共有3×i-1个非零元素；
- 在第i行，有$a_{i,i-1}$ $a_{i,i}$ $a_{i,i+1}$共3个非零元素，若j为i-1,i，或i+1，则元素a_{ij}之前分别有0、1、2个非零元素，即j-i+1个非零元素，这样，
- 非零元素a_{ij}的地址为：
  - **LOC(i,j)=LOC(0,0)+[3×i-1+(j-i+1)] ×L**

    =LOC(0,0)+(2i+j) ×L
- 上例中，a_{34}对应着sa[10]， a_{21}对应着sa[5]
  - **k=2×i+j=2×3+4=10**　　**k=2×2+1=5**

矩阵的压缩存储

- 上述的各种特殊矩阵，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。

矩阵的压缩存储

●二、稀疏矩阵的压缩存储方法

如果矩阵中只有**少量的非零值元**，并且这些**非零值元在矩阵中的分布没有一定规律**，则称为**随机稀疏矩阵**，简称为稀疏矩阵。至于矩阵中究竟含多少个零值元才被称为是稀疏矩阵，目前还没有一个确切的定义，它只是一个凭人的直觉来了解的概念。

假设在 m×n 的矩阵中有 t 个非零值元，令 $\delta = \frac{t}{m \times n}$ ，称δ 为矩阵的**稀疏因子**，则通常认定δ ≤0.05的矩阵为稀疏矩阵。

矩阵的压缩存储

- 思考：如何存储稀疏矩阵中的非零值元？
  - 首先应该分析一下，如果仍然采用二维数组表示稀疏矩阵，那么它的**弊病**是什么？
  - (1)**浪费空间：**存放了大量没有用的零值元素。
  - (2)**浪费时间：**在进行矩阵的运算中进行了很多与零元素的运算。
  - 由此解决稀疏矩阵压缩存储的**目标**是：
    - 1)尽可能减少或不存储零值元；
    - 2)尽可能不作和零值元进行的运算；
    - 3)便于进行矩阵运算，即易于从一对行列号（i，j）找到矩阵的元，易于找到同一行或同一列的非零值元。

矩阵的压缩存储

- 由于压缩存储的**基本宗旨**是只存放矩阵中的非零值元，则在存储非零元的值的同时必须记下它在矩阵中的位置(i, j) ，反之，一个三元组(i, j, a_{ij})唯一确定了矩阵A中的一个非零值元，由此可以用“**数据元素为上述三元组的线性表**”表示稀疏矩阵，并且非零元在三元组线性表中是“以行为主”有序排列的。
- 相应于线性表的两种存储结构可得到稀疏矩阵的不同压缩存储方法 。

矩阵的压缩存储

●1、三元组顺序表

以顺序存储结构作为三元组线性表的存储结构，由此得到的稀疏矩阵的一种压缩存储方法，称之谓三元组顺序表。

➔ 例如表示矩阵：

$$M=\begin{bmatrix}0&0&9&0&-7&0\\0&0&0&0&0&0\\0&0&0&8&0&0\\5&0&0&0&0&2\\0&0&0&0&16&0\end{bmatrix}$$

的三元组线性表为 ((1,3,9),(1,5,-7),(3,4,8),(4,1,5),(4,6,2),(5,5,16))。

矩阵的压缩存储——三元组顺序表

●稀疏矩阵的三元组顺序表的结构定义

```
const MAXSIZE=12500;     // 假设非零元个数的最大值为12500
typedef struct {
    int i, j;                    // 该非零元的行号和列号
    ElemType e;                  // 该非零元的值
} Triple;                        // 三元组
typedef struct {
    Triple data[MAXSIZE + 1];    // 非零元三元组表，data[0] 未用
    int mu,nu,tu;          // 矩阵的行数、列数和非零元的个数
} TSMatrix;                      // 三元组顺序表
```

矩阵的压缩存储——三元组顺序

$$M=\begin{bmatrix}0&0&9&0&-7&0\\0&0&0&0&0&0\\0&0&0&8&0&0\\5&0&0&0&0&2\\0&0&0&0&16&0\end{bmatrix}$$

- 显然，在三元组顺序表中容易从给定的行列号（i，j）找到对应的矩阵元。
  - 首先按行号 i 在顺序表中进行“有序”搜索
  - 找到相同的 i 之后再按列号进行有序搜索
  - 若在三元组顺序表中找到行号和列号都和给定值相同的元素，则其中的非零元值即为所求
  - 否则为矩阵中的零元

1	3	9
1	5	-7
3	4	8
4	1	5
4	6	2
5	5	16

- 同一行的下一个非零元即为顺序表中的后继，**搜索同一列中下一个非零元稍微麻烦些**，但由于顺序表是以行号为主序有序的，则在依次搜索过程中遇到的下一个列号相同的元素即为同一列的下一个非零元。

矩阵的压缩存储——三元组顺序表

- 当以三元组顺序表表示稀疏矩阵时，是否仍然便于进行运算呢？在此以“**转置**”运算为例讨论算法。
  - ➔ 假设前面所列举的矩阵M的转置矩阵为 T，则按照矩阵转置的定义：

$$M=\begin{bmatrix}0&0&9&0&-7&0\\0&0&0&0&0&0\\0&0&0&8&0&0\\5&0&0&0&0&2\\0&0&0&0&16&0\end{bmatrix}\qquad T=\begin{bmatrix}0&0&0&5&0\\0&0&0&0&0\\9&0&0&0&0\\0&0&8&0&0\\-7&0&0&0&16\\0&0&0&2&0\end{bmatrix}$$

  - ❊ 对比M和T，T的行数、列数和非零元的个数等于M的列数、行数和非零元的个数，且T中的每个非零元和M中的非零元相比，它们的值相同，但行列号互换。

矩阵的压缩存储——三元组顺序表

- M的三元组顺序表为：

$$M=\begin{bmatrix}0&0&9&0&-7&0\\0&0&0&0&0&0\\0&0&0&8&0&0\\5&0&0&0&0&2\\0&0&0&0&16&0\end{bmatrix}$$

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

T.data ==

1	1	4	5
2	3	1	9
3	4	3	8
4	5	1	-7
5	5	5	16
6	6	4	2

- 由于三元组表中元素的顺序约定为以行序为主序，即在 T 中非零元的排列次序是以它们在 M 中的列号为主序的。因此**转置的主要操作就是要确定M中的每个非零元在T的三元组顺序表中的位序**，即**分析两个矩阵中值相同的非零元分别在 M.data 和 T.data 中的位序之间的关系是什么**。

矩阵的压缩存储——三元组顺序表

一个m×n的矩阵M，它的转置T是一个n×m的矩阵，则m[i][j]=t[j][i]，0≦i<m，0≦j<n。M.data和T.data分别是矩阵M和T的三元组顺序表实现。

简单转置算法按M.data元素固有顺序，依次交换每个三元组i和j，逐一生成T.data的各个元素。如下所示：

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

简单转置算法生成的三元组线性表序列为：

(3,1,9) (5,1,-7) (4,3,8) (1,4,5) (6,4,2) (5,5,16)

三元组在M中按行优先顺序排列，在T中，由于行列交换操作，因此三元组在T中按列优先顺序排列。即，T.data将是一个**按列优先**顺序存储的稀疏矩阵T。

要得到按行优先顺序存储的t.data，就必须重新排列三元组的顺序。

矩阵的压缩存储——三元组顺序表

另一种思路是，按m.data的**列序**转置。即按列优先顺序访问M.data元素，依次交换每个三元组i和j，逐一生成T.data的各个元素。分析可知，所得到的转置矩阵T的三元组顺序表t.data按行优先顺序存储。

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

1.扫描M.data，找出列号为1的元素转置，得到(1,4,5)；
2.扫描M.data，找出列号为2的元素转置，没有则继续；
3.扫描M.data，找出列号为3的元素转置，得到(3,19)。依次类推，则生成的三元组线性表序列为：
(1,4,5) (3,1,9) (4,3,8) (5,1,-7) (5,5,16) (6,4,2)

矩阵的压缩存储——三元组顺序表

按这种方法设计的算法，其**基本思想**是：对M中的每一列col(0≦col≦n-1)，通过从头至尾扫描三元表m.data，找出所有列号等于col的那些三元组，将它们的行号和列号互换后依次放入t.data中，即可得到T的按行优先的压缩存储表示。

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

T.data ==

1	1	4	5
2	3	1	9
3	4	3	8
4	5	1	-7
5	5	5	16
6	6	4	2

矩阵的压缩存储——三元组顺序表

算法5.1

```
Status TransposeSMatrix(TSMatrix M,TSMatrix &T)
  {//将稀疏表示的矩阵M转置，生成矩阵T
    T.mu = M.nu; T.nu=M.mu; T.tu=M.tu;
    if(T.tu){
      q=1; //初始化三元组顺序表（T.data）索引
       for(col=1; col<=M.nu; ++col) //按列枚举M
         for(p=1; p<=M.tu; ++p) //枚举M所有三元组
           if( M.data[p].j == col ) {//转置列号等于当前列的三元组
             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;
}
```

矩阵的压缩存储——三元组顺序表

- 分析这个算法，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(nu*tu)，即矩阵的列数和非零元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<=nu;++col)
    for(row=1;row<=mu;++row)
        t[col][row]=m[row][col];
```

其时间复杂度为O(nu*mu)。

- 算法5.1中，当非零元素的个数tu和mu*nu同数量级时，算法的时间复杂度为$O(mu*nu^2)$。

矩阵的压缩存储——三元组顺序表

- 三元组顺序表虽然节省了存储空间，但时间复杂度比一般矩阵转置的算法还要复杂，同时还有可能增加算法的难度。因此，此算法仅适用于tu<<mu*nu的情况。

- 下面给出另外一种称之为**快速转置**的算法，其算法思想为：
  - 预先确定矩阵M中每一列（T中每一行）的首个非零元在T.data中的位置；
  - 依次转置M.data中各个元素时，根据以上信息，将生成元素放入T.data的正确位置。

矩阵的压缩存储——三元组顺序表

- 为了预先确定矩阵M中的每一列的第一个非零元素在数组T中应有的位置，需要先求得矩阵M中的每一列中非零元素的个数。
- 因为：矩阵M中某一列的第一个非零元素在数组T中应有的位置等于前一列第一个非零元素的位置加上前列非零元素的个数。
- 为此，需要设置两个一维数组num[col]和cpot[col]
  - num[col]：M中每列非零元素的个数。
  - cpot[col]：M中每列第一个非零元素在T中的位置。

cpot[1]=1

cpot[col]=cpot[col-1]+num[col-1]　　2=<col<=m.nu

矩阵的压缩存储——三元组顺序表

例：列起始位置的确定

●M的三元组顺序表为：

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

M中第一列（T中第一行）有一个元素。

●T 的首个列非零元素序号：

●cpot[1]=1

●num[1]=1

● cpot[2]=cpot[1]+num[1]=2

➔以下表格中的三行依次为：M 的列号、M 中各列非零元的总数和M中每一列（ 即T 中每一行）第一个非零元在 T.data 中的序号：

col	1	2	3	4	5	6
num[col]	1					
cpot[col]	1	2				

矩阵的压缩存储——三元组顺序表

例：列起始位置的确定

●M的三元组顺序表为：

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

●T 的首个列非零元素序号：

●cpot[2]=2

●num[2]=0

● cpot[3]=cpot[2]+num[2]=2

M中第二列（T中第二行）有零个（非零）元素。

➔以下表格中的三行依次为：M 的列号、M 中各列非零元的总数和M中每一列（ 即T 中每一行）第一个非零元在 T.data 中的序号：

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

矩阵的压缩存储——三元组顺序表

例：列起始位置的确定

- M的三元组顺序表为：

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

M中第三列（T中第三行）有一个元素。

- T 的首个列非零元素序号：
- cpot[3]=2
- num[3]=1
- cpot[4]=cpot[3]+num[3]=3

➔以下表格中的三行依次为：M 的列号、M 中各列非零元的总数和M中每一列（ 即T 中每一行）第一个非零元在 T.data 中的序号：

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

矩阵的压缩存储——三元组顺序表

例：列起始位置的确定

- M的三元组顺序表为：

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

- T 的首个列非零元素序号：
- 依次类推
- cpot[5]=4
- num[5]=2
- cpot[6]=cpot[5]+num[5]=6
- num[6]=1

M中第五列（T中第五行）有二个元素。

➔以下表格中的三行依次为：M 的列号、M 中各列非零元的总数和M中每一列（ 即T 中每一行）第一个非零元在 T.data 中的序号：

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

矩阵的压缩存储——三元组顺序表

例：基于列起始位置的快速转置算法

●M的三元组顺序表为：

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

1.M的第1个元素（1,3,9），列号为3;
2.查表可知第3列首个非零元素应在T.data的索引为2;
3.在T.data中存入该元素(3,1,9）

●T 的三元组顺序表为：

索引	i	j	e
1			
2	3	1	9
3			
4			
5			
6			

➔以下表格中的三行依次为：M 的列号、M 中各列非零元的总数和M中每一列（ 即T 中每一行）第一个非零元在 T.data 中的序号：

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

矩阵的压缩存储——三元组顺序表

例：基于列起始位置的快速转置算法

●M的三元组顺序表为：

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

1.M的第2个元素（1,5,-7），列号为5;
2.查表可知第5列首个非零元素应在T.data的索引为4;
3.在T.data中存入该元素(5,1,-7）；
4.第5列下个非零元素存储位置自增。

●T 的三元组顺序表为：

索引	i	j	e
1			
2	3	1	9
3			
4	5	1	-7
5			
6			

➔以下表格中的三行依次为：M 的列号、M 中各列非零元的总数和M中每一列（ 即T 中每一行）第一个非零元在 T.data 中的序号：

col	1	2	3	4	5	6
num[col]	1	0	1	1	2	1
cpot[col]	1	2	2（**3**）	3	4（**5**）	6

矩阵的压缩存储——三元组顺序表

例：基于列起始位置的快速转置算法

●M的三元组顺序表为：

M.data ==

1	1	3	9
2	1	5	-7
3	3	4	8
4	4	1	5
5	4	6	2
6	5	5	16

1.依次类推转置M中其他元素，M的第6个元素（5,5,16），列号为5;
2.查表可知第5列首个非零元素应在T.data的索引为5;
3.在T.data中存入该元素(5,5,16）；

●T 的三元组顺序表为：

索引	i	j	e
1	1	4	5
2	3	1	9
3	4	3	8
4	5	1	-7
5	5	5	16
6	6	4	2

➔以下表格中的三行依次为：M 的列号、M 中各列非零元的总数和M中每一列（ 即T 中每一行）第一个非零元在 T.data 中的序号：

col	1	2	3	4	5	6
num[col]	1	0	1	1	2	1
cpot[col]	1（2）	2	2（3）	3（4）	5（6）	6（7）

矩阵的压缩存储——三元组顺序表

- **例如**：M.data 中第2个元素 (1,5,-7) 转置到T中的位序为4，这是为什么？
  - 因为该元素在M中的列号为5，行号为1，又M中前4列中非零元的总数为3。换句话说，T中前4行中只有3个非零元，而值为-7的非零元是T中第5行的第一个非零元，当然在 T.data 中应该位居第4。
  - 由此，**转置算法的操作步骤为**：
    - 1.求 M 矩阵的每一列中非零元的个数；
    - 2.确定T的每一行中第一个非零元在 T.data 中的序号；
    - 3.将 M.data 中每个元素依次转置存储到 T.data 中相应位置。

矩阵的压缩存储——三元组顺序表

● 算法 5.1

```
Status FastTransposeSMatrix(TSMatrix M, TSMatrix &T)
{
    // 采用三元组顺序表存储表示，求稀疏矩阵M的转置矩阵 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];          // 求 M 中每一列所含非零元的个数
        cpot[1] = 1; // M 第一列在T中的存储索引初始化为1
```

矩阵的压缩存储——三元组顺序表

```
        for (col=2; col<=M. nu; ++col)
              cpot[col] = cpot[col-1] + num[col-1];
        // 求T中每一行的第一个非零元在T.data中的序号
        for (p=1; p<=M.tu; ++p) {     // 转置矩阵元素
              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];
         } // for
    } // if
    return OK;
} // FastTransposeSMatrix
```

➔演示5.swf

❊上述算法的时间复杂度为***O*** **(M.nu+M.tu)**。 如果tu和mu×nu等数量级，则时间复杂度为**O(mu×nu)**

本章小结

- 通过这一章的学习，主要是了解**数组的类型定义**及其在高级语言中实现的方法。数组作为一种数据类型，它的特点是一种多维的线性结构，并只进行存取或修改某个元素的值，因此它只需要采用顺序存储结构。
- 介绍了**随机稀疏矩阵的三种表示方法**。至于在具体应用问题中采用哪一种表示法这取决于该矩阵主要进行什么样的运算。
- 从结构本身特性而言，**广义表**归属于线性结构，但实现广义表操作的算法和树的操作的算法更为相近，这正是广义表这种数据结构的特点。由于广义表是一种递归定义的线性结构，因此它兼有线性结构和层次结构的特点。

本章知识点与重点

- **知识点**

数组的类型定义、数组的存储表示、特殊矩阵的压缩存储表示方法、随机稀疏矩阵的压缩存储表示方法

- **重点和难点**

本章重点是学习数组类型的定义及其存储表示 。