数据结构

北京邮电大学 网络空间安全学院

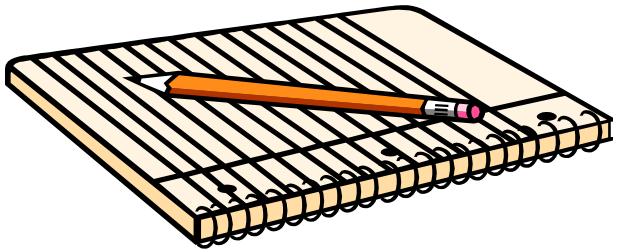
武 斌

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
- 它和其它诸如整数、实数等**原子类型**有何不同？

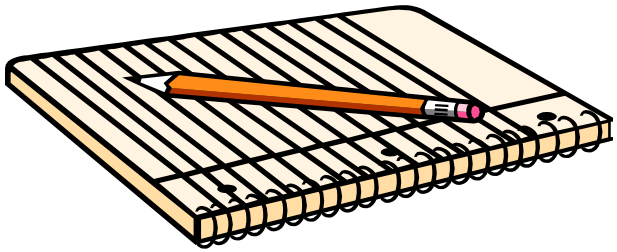
数组属于一种结构类型。换句话说，“数组”是一种数据结构。

- 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-逻辑关系

- 一维数组：顺序表
- 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：

$ROW = \{ \langle a_{i-1,j}, a_{i,j} \rangle \mid i=1,\ldots,m-2, 0 \le j \le n-1,$

$a_{i-1,j}, a_{i,j} \in ElemType\}$ (称作"行关系")

$COL = \{ \langle a_{i,j-1}, a_{i,j} \rangle \mid j=1,\ldots,n-2, 0 \le i \le m-1,$

$a_{i,j-1}, a_{i,j} \in ElemType\}$ (称作"列关系")

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

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

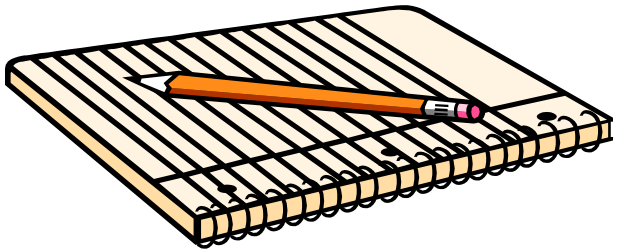
数组的类型定义

- 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**。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)和每一维的上、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-**映象**。
- 由于数组类型不作插入和删除的操作，因此只需要通过“**顺序映象**”得到它的存储结构，即**借助数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系**。

数组的顺序存储表示

$$A_{mn} = \begin{bmatrix} a_{00} & a_{01} & \cdots & \cdots & a_{0n-1} \\ a_{10} & a_{11} & \cdots & \cdots & a_{1n-1} \\ \cdots & \cdots & \cdots & \cdots & \\ a_{m-10} & a_{m-11} & \cdots & a_{m-1n-1} & \end{bmatrix}$$

a_{00}	a_{01}	a_{02}	a_{03}	……	$a_{0\ n-1}$	…	$a_{m-1,n-1}$

- 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，按行优先顺序存储的线性序列为：

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

在PASCAL、C语言中，数组就是按行优先顺序存储的。

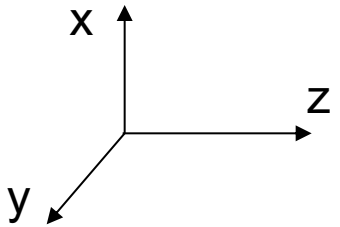
➔ 列优先顺序——将数组元素按列向量排列，第j+1个列向量紧接在第j个列向量之后，A的m×n个元素按列优先顺序存储的线性序列为：

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

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：行优先顺序可规定为先排最右的下标，从右到左，最后排最左下标；列优先顺序与此相反，先排最左下标，从左向右，最后排最右下标。



- **三维数组**：演示一、演示二
- 按上述两种方式顺序存储的数组，只要知道开始结点的存放地址（即基地址），维数和每维的上、下界，以及每个数组元素所占用的单元数，就可以将**数组元素的存放地址表示为其下标的线性函数**。
- 因此，**数组中的任一元素可以在相同的时间内存取，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

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & \cdots & a_{0n-1} \\ a_{10} & a_{11} & \cdots & \cdots & a_{1n-1} \\ \cdots & \cdots & \cdots & \cdots & \\ a_{m-10} & a_{m-11} & \cdots & & a_{m-1n-1} \end{bmatrix}$$

  - 在“以行为主”的顺序映象中的存储地址为：
    - **LOC (i, j) = LOC(0,0) + (i×n+j) × L**
  - 在“以列为主”的顺序映象中的存储地址为：
    - **LOC (i, j) = LOC(0,0) + (j×m+i) ×L**

其中 LOC(0,0) 是二维数组中第一个数据元素（下标为(0,0)）的存储地址，称为数组的 “基地址” 或“基址”。

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：
  - **LOC (i,j,k) = LOC(0,0,0) + (i×m×n + j×n+k)×L**
- 推广到 N 维数组，则得到
  - **$LOC(j_1,j_2,\ldots,j_N) = LOC(0,0,\ldots,0) + (b_2\times\ldots\times b_n\times j_1 + b_3\times\ldots\times b_n\times j_2 + \ldots + b_n\times j_{N-1}+j_N)\times L$**

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

  - 可缩写成：$\mathrm{LOC}(j_1, j_2, \ldots, j_N) = \mathrm{LOC}(0,0,\ldots,0) + \sum_{i=1}^{N} c_i j_i$

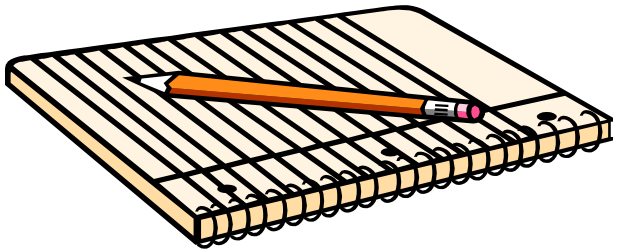
其中 $c_N = L, c_{i-1} = b_i \times c_i, 1 < i < N$

数组的顺序存储表示

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

- 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，称具有这一特点的存储结构为**随机存储结构**。

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：

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

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
- 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。

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(n-1)/2-1

矩阵的压缩存储

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。

- 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$a_{i0},a_{i1},a_{i2},\ldots,a_{ij-1}$），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(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}之前的i行一共有i(2n-i+1)/2个元素，在第i行上，a_{ij}前恰好有j-i个元素：$a_{ij}, a_{ij+1}, \ldots a_{ij-1}$。因此，sa[k]和$a_{ij}$的对应关系是：

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

  - 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：

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

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($a_{i+1\ i}$,0≦i≦n-2)。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。
- 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：在三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,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(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。

●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 $\delta = \frac{t}{m \times n}$ ，称为矩阵的**稀疏因子**，则通常认定 $\delta \leq 0.05$的矩阵为稀疏矩阵。

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

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——三元组顺序表

- 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
- 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。

((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

- 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.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

- 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，且m[i][j]=t[j][i]，0≦i≦m，0≦j≦n，即M的行是T的列，M的列是T的行。

将M转置为T，就是将M的三元组表m.data置换为表T的三元组表t.data，如果**只是简单地交换m.data中i和j的内容**，那么得到的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。

由于M的列是T的行，因此，按m.data的**列序**转置，所得到的转置矩阵T的三元组表t.data必定是按行优先存放的。

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)
  {
    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;
}
```

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$O(mu*nu^2)$。
- 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要复杂，同时还增加了算法的难度。因此，此算法仅适用于tu<<mu*nu的情况。

矩阵的压缩存储——三元组顺序表

- 如何降低时间复杂度？改变对M矩阵**按列多轮扫描**的策略。
- **快速转置**算法，其算法思想为：
  - → **首先得到转置矩阵T按行的元素结构（每行首元素在表中的位次），此结构可以由M矩阵按列统计得到。**

- 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 的三元组顺序表为：

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

- 具体实施如下：一遍扫描**先确定三元组的位置关系**，二次扫描**由位置关系装入三元组**。
- 对**M.data扫描一次**，按M.data的第二个成员参数提供的列号一次确定位置装入T.data。
- **位置关系**是此种算法的关键。**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。**

●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 的三元组顺序表为：

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

位置关系如何得到？

- 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]：统计M中每列非零元素的个数，num[col]的值可以由M.data的第二个成员参数求得。
  - cpot[col]：由递推关系得出M中的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

●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 的三元组顺序表为：

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

➔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

- 表格中的 col 指示 M 中的列号，即 T 中的行号，则 cpot 中的值可递推求得：
  - cpot[1] = 1;
  - cpot[col] = cpot[col-1]+num[col-1]
  - T中第一行的第一个非零元在 data 中的位置肯定是1，而之后其它各行的第一个非零元在 data 中的位置显然取决于它的前一行中的非零元的个数。

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 中每个元素依次复制到 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;
```

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

❊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)**

矩阵的压缩存储——行逻辑链接的顺序表

●2、行逻辑链接的顺序表

由于三元组顺序表以行序为主序存放矩阵的非零元，则为取得第 i 行的非零元，必须从第一个元素起进行搜索查询。而从上页讨论的矩阵转置算法可见，在算法过程中计算得到的 cpot[M.nu+1] 中的值实际上起到了一个**“指示矩阵中每一行的第一个非零元在三元组表中的序号”**的作用。

因此，如果在建立稀疏矩阵的三元组顺序表的同时，将这个信息固定在存储结构中，将便于随机存取稀疏矩阵中任意一行的非零元。可将 cpot 中的值视作指向每一行第一个非零元的指针，故称这种表示方法为**“行逻辑链接”**的顺序表。

矩阵的压缩存储——行逻辑链接的顺序表

●行逻辑链接的三元组顺序表的结构定义

```
const MAXRC = 500;                  // 矩阵行数和列数的最大值
const MAXSIZE=12500;                // 假设非零元个数的最大值为12500
typedef struct {
    Triple data[MAXSIZE + 1];       // 非零元三元组表，data[0] 未用
    int rpos[MAXRC + 1];            // 指示各行第一个非零元在data中的位置
    int mu, nu, tu;                 // 矩阵的行数、列数和非零元的个数
} RLSMatrix;                        // 行逻辑链接顺序表
```

矩阵的压缩存储——行逻辑链接的顺序表

- 如何建立稀疏矩阵的行逻辑链接的顺序表？

  - 首先应该输入该矩阵的行数、列数以及非零元的个数，然后依次输入各个非零元的行号、列号和它的值，并在输入每一行的第一个非零元的同时为 **rpos** 中相应分量赋值。显然，对于顺序结构应尽少进行“移动元素”的操作，因此非零元的输入次序应对行有序，且同一行的非零元按列有序。

  - 算法中需要注意的是，可能存在某一行或连续几行都没有非零元的情况，则这些行的“第一个非零元在顺序表中的位置”应该和当前输入的那个非零元所在行相同。

  - 虽然，表面上 rpos 中各分量的值只是指示每一行的第一个非零元在 data 中的位置，实际上它还隐含着一个信息，即"**第 k 行的非零元的个数=rpos[k+1]-rpos[k]**"。为了使这个公式也适用于最后一行，在 rpos 中增添一个下标为(行数+1)的分量。

矩阵的压缩存储——行逻辑链接的顺序表

- 算法 5.2

```
void Create_SM(RLSMatrix& M)
{// 以行序为主序的次序逐个输入非零元，建立稀疏矩阵的行逻辑链接顺序表
    cin >> M.mu>>M.nu>>M.tu;
    k = 1; curRow = 0;
    for (i=1; i<= M.tu; i++) {
        cin>>M.data[k].i >> M.data[k].j >> M.data[k].e;
        while (curRow < M.data[k].i)
            M.rpos[++curRow]=k;
        ++k;
    } // for
    while (curRow < M.mu+1)          // 剩余的没有非零元的行
        M.rpos[++curRow]=k;
}
```

※上述算法的控制结构只有一个for循环，显然它的时间复杂度为 ***O(M.tu)***。

矩阵的压缩存储——行逻辑链接的顺序表

- 以行逻辑链接顺序表表示时，如何进行两个矩阵相乘的运算。
  - 以二维数组表示矩阵时的矩阵相乘的乘法如下：

$$Q(i,j)=\sum_{k=1}^{n_1} M(i,k)\times N(k,j) \qquad \begin{matrix} 1\le i\le m_1 \\ 1\le j\le n_2 \end{matrix}$$

  - **void** multiplication(**double** M[m][n], **double** N[n][p], **double** Q[m][p])

```
{
   for (i=0; i<m; i++)
      for (j=0; j<p; j++){
         Q[i][j]=0;
         for (k=0; k<n; k++)
            Q[i][j]+=M[i][k]*N[k][j];
      }
}
```

矩阵的压缩存储——行逻辑链接的顺序表

- 从上述算法可见，乘积矩阵中的每个元是由 M 矩阵中的一行和 N 矩阵中一列的对应元的乘积和。如果 M[i][k] 和 N[k][j] 两者中有一个为零元，其乘积即为零。
- 由此可得如下三个结论：
  - 1）**乘积矩阵Q的非零元仅由M和N中的非零元相乘得到**，换句话说，为求得两个稀疏矩阵相乘的乘积，只需要对 M 和 N 中的非零元进行运算即可；
  - 2）从矩阵相乘的规则得知，**并非两者中每个元素都要进行彼此相乘**，对 M 中的每个非零元，只要在 N 中查找其行号等于它的列号的非零元相乘即可；
  - 3）在上述算法中，**Q 的每个元素是由 M 中的一行和N中的一列相应元素连续相乘相加得到的，但在行逻辑链接的三元组顺序表中要连续找到同一列的元素是很不方便的**，因此需要改变计算的顺序，寻找新的算法。

矩阵的压缩存储——行逻辑链接的顺序表

- 对行逻辑链接的三元组顺序表表示的稀疏矩阵，我们不易直接求得 Q 的一个非零元，但可以设法求得 Q 的“一行”非零元。因为 **Q 的一行非零元一定是由 M 的相应行的非零元得到**的，并且对Q的每个元以“累加”的方式求得。

$$Q(2,1) = M(2,0) \cdot N(0,1) + M(2,1) \cdot N(1,1) + M(2,2) \cdot N(2,1)$$

$$Q(2,2) = M(2,0) \cdot N(0,2) + M(2,1) \cdot N(1,2) + M(2,2) \cdot N(2,2)$$

$$Q(2,3) = M(2,0) \cdot N(0,3) + M(2,1) \cdot N(1,3) + M(2,2) \cdot N(2,3)$$

$$Q(2,4) = M(2,0) \cdot N(0,4) + M(2,1) \cdot N(1,4) + M(2,2) \cdot N(2,4)$$

矩阵的压缩存储——行逻辑链接的顺序表

- 具体作法为：

设累加器 ctemp 的容量为 p(p 为 Q 中列数，即 N 的列数)
初始化累加器 ctemp[ ]=0;
for (M 中第 i 行的所有非零元 M.data[p])
{
    brow=M.data[p].j;　　// 该非零元在 M 中的列号
    for (N中第 brow 行的非零元 N.data[q])
    {
        ccol= N.data[q].j;　// 该非零元在N中的列号
        ctemp[ccol]+=M.data[p].e * N.data[q].e;
    }
}

矩阵的压缩存储——行逻辑链接的顺序表

- 容易看出上述运算的结果 ctemp 中所有非零分量即为 Q 中第 i 行的所有非零元。实际上，乘积 Q 中哪些元为零哪些元为非零，并非从 M 和 N 的情况一下子就能看出来的，也只有通过上述运算的结果才能得到 Q 中一行的非零元，之后可按其列号大小依次存入 Q.data 中。

矩阵的压缩存储——行逻辑链接的顺序表

- 例如：

$$M=\begin{bmatrix}0&1&0\\2&0&-4\\-1&0&0\\0&-2&3\end{bmatrix},N=\begin{bmatrix}2&0&0&6\\0&-4&0&0\\0&-1&0&3\end{bmatrix},Q=M\times N=\begin{bmatrix}0&-4&0&0\\4&4&0&0\\-2&0&0&-6\\0&5&0&9\end{bmatrix}$$

M.data

	i	j	e
1	1	2	1
2	2	1	2
3	2	3	-4
4	3	1	-1
5	4	2	-2
6	4	3	3

p →

N.data

	i	j	e
1	1	1	2
2	1	4	6
3	2	2	-4
4	3	2	-1
5	3	4	3

q →

Q.data

	i	j	e
1	1	2	-4
2			
3			
4			
5			
6			

矩阵的压缩存储——行逻辑链接的顺序表

- 例如：对所举例子中的矩阵
  - 当 i=2 时，M 在 data 中的第一个非零元的位置 p=2，则 brow=1，矩阵 N 中有两个非零元，q=1和2。
    - 当 q=1 时，因为 N.data[q].j 为1，则M.data[p].e*N.data[q].e 应叠加到 ctemp[0] 中去，因为它是构成 Q[2][1] 的部分积。
    - 同理，当 q=2 时，因为N.data[q].j为4，则 M.data[p].e*N.data[q].e 应叠加到 ctemp[3] 中去，因为它是构成 Q[2][4] 的部分积。
  - 对 M 中第2行的另一个非零元，即 p=3，此时 brow=M.data[p].j=3，q=4和5，作如上相同处理之后，ctemp 中各分量的值依次为4,4,0,0。由此得到 Q 中第2行的两个非零元，可依次将它们放入 Q.data 中。

矩阵的压缩存储——行逻辑链接的顺序表

- ctemp

1	2	3	4
0+2×2 =4	0+(-4) ×(-1) =4	0	0+2×6 +(-4) ×3=0

- M.rpos的值为：

行号	1	2	3	4	(5)
M.rpos	1	2	4	5	7

- N.rpos的值为：

行号	1	2	3	(4)
N.rpos	1	3	4	6

矩阵的压缩存储——行逻辑链接的顺序表

- 由此，两个稀疏矩阵相乘（Q＝M ×N）的过程可大致描述如下：

```
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
```

矩阵的压缩存储——行逻辑链接的顺序表

- 其中：
  - ➔ 对矩阵Q进行初始化的操作如下：

```
Q.rows = M.rows;   // Q的行数和M相同
Q.cols = N.cols;   // Q的列数和N相同
Q.terms = 0;       // Q的非零元个数初始化为零
```

  - ➔ 求得Q中一行非零元的操作为：

```
ctemp[] = 0;       // 当前行各元素累加器清零

 for(p=M.rpos[arow];p<M.rpos[arow+1];++p)
{   // 处理 M 当前行中每一个非零元
      brow=M.data[p].j;       // 找到对应元在N中的行号
      for(q=N.rpos[brow];q<N.rpos[brow+1];++q)
      {
            ccol = N.data[q].j;     // 乘积元素在Q中列号
            ctemp[ccol]+=M.data[p].e * N.data[q].e;
      } // for q
 } // for p
```

矩阵的压缩存储——行逻辑链接的顺序表

● 算法 5.3

```
bool MultSMatrix(RLSMatrix M, RLSMatrix N, RLSMatrix &Q)
{
// 采用行逻辑链接存储表示，求矩阵乘积Q=M×N。
if (M.nu!= N.mu)   return ERROR;
Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0;
                                    // M的列数和N的行数不等，不能相乘
if (M.tu*N.tu != 0) {          // M和N中均含有非零元对矩阵Q进行初始化;
  for (arow=1; arow<=M. mu; ++arow)
  {
    //处理M(即Q)的每一行，求得 Q 中第crow( =arow)行的非零元
```

矩阵的压缩存储——行逻辑链接的顺序表

```
ctemp[] = 0;     // 当前行各元素累加器清零
Q.rpos[arow] = Q.tu+1;
for(p=M.rpos[arow];p<M.rpos[arrow+1];++p)
{   // 处理 M 当前行中每一个非零元
  brow=M.data[p].j; // 找到对应元在N中的行号
  for(q=N.rpos[brow];q<N.rpos[brow+1];++q)
  {
        ccol = N.data[q].j; // 乘积元素在Q中列号
        ctemp[ccol]+=M.data[p].e * N.data[q].e;
  } // for q
} // for p
```

矩阵的压缩存储——行逻辑链接的顺序表

```
    for (ccol=1; ccol<=Q.nu; ++ccol)   // 压缩存储该行非零元
        if (ctemp[ccol]) {
            if (++Q.tu > MAXSIZE)  return ERROR;
            Q.data[Q.tu] = (arow, ccol, ctemp[ccol]);
        } // if
    } // for arow
  } // if
  return OK;
} // MultSMatrix
```

- 该算法的时间复杂度为

- 累加器ctemp初始化的复杂度O(M.mu×N.nu)，求Q的所有非零元的复杂度O（M.tu×N.tu/N.mu），Q压缩存储的复杂度O(M.mu×N.nu)。

  O (M.mu×N.nu+M.tu×N.tu/N.mu)。

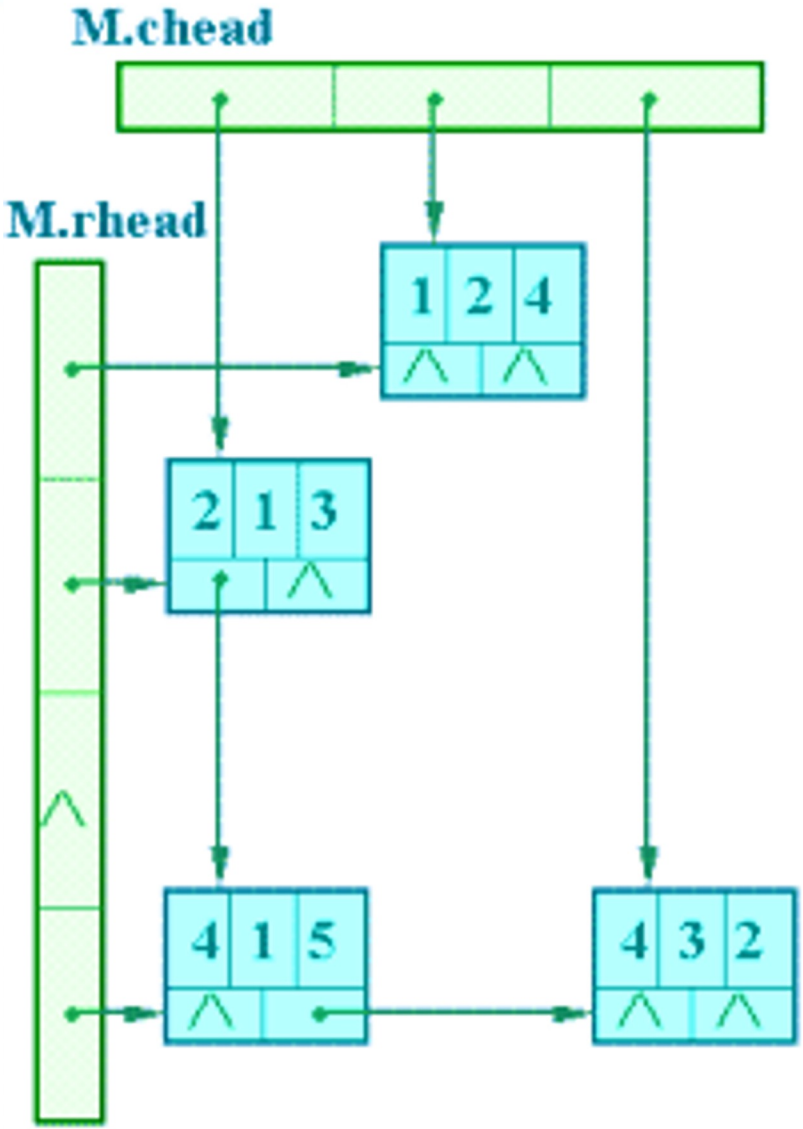
- 其中，N.tu/N.mu 表示N中每一行非零元个数的平均值。

矩阵的压缩存储——十字链表

3、十字链表

- 以上讨论的矩阵运算都不改变参与运算的矩阵本身，即在它的存储结构中没有进行插入、删除之类的操作。
- 如果矩阵运算的结果将增加或减少已知矩阵中的非零元的个数，则显然不宜采用顺序存储结构，而应以链式映象作为三元组线性表的存储结构。

矩阵的压缩存储——十字链表



矩阵的压缩存储——十字链表

➔ 稀疏矩阵的十字链表存储表示

```
typedef struct OLNode {        // 结点结构定义
    int i, j;        // 该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
} *OLink;
typedef struct {                    // 链表结构定义
    OLink *rhead, *chead;    // 行和列链表头指针向量基址由CreateSMatrix分配
    int mu, nu, tu;          // 稀疏矩阵的行数、列数和非零元个数
} CrossList;
```

矩阵的压缩存储——十字链表

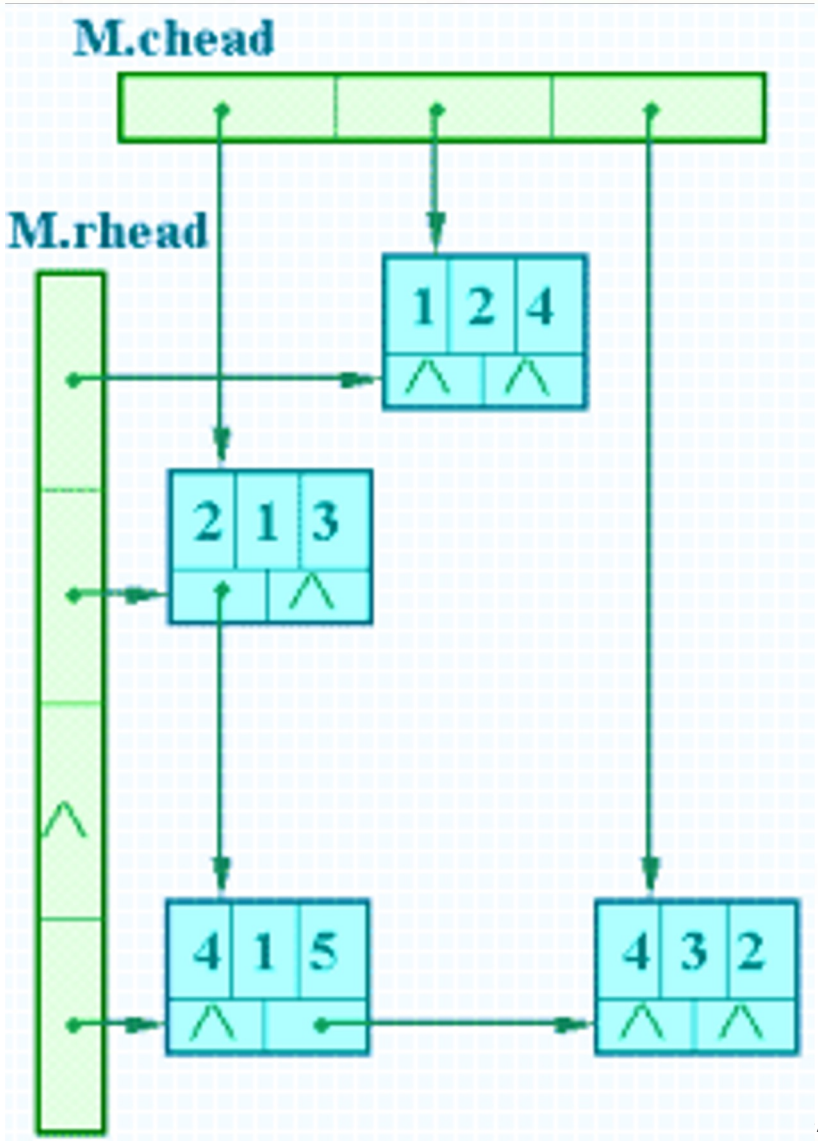
- 从上述结构定义可见，每个非零元以**含5个域的结点**表示，除了表示**非零元信息的三元组(i,j,e)**之外，添加了两个链域：一个是链接同一行下一个非零元结点的 **right 域**，另一个是链接同一列下一个非零元结点的 **down 域**。
- 每个非零元既是某个行链表中的一个结点，又是某个列链表中的一个结点，整个矩阵构成了一个十字交叉的链表，故称为**十字链表**，以两个分别存放行链表的头指针和列链表的头指针的一维数组表示之。

矩阵的压缩存储——十字链表

●例如，矩阵

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

的十字链表如图所示。



矩阵的压缩存储

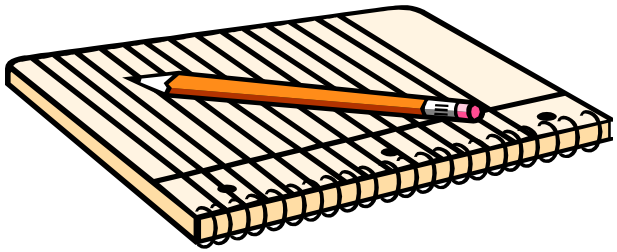
● 算法 5.4

```
Status CreateSMatrix_OL (CrossList &M)
{   // 创建稀疏矩阵M的十字链表存储结构,若存储分配失败，则返回FALSE
    cin >> M.mu >> M.nu >> M.tu;        // 输入M的行数、列数和非零元个数
    if (!(M.rhead = (OLink*)malloc((M.mu+1)*sizeof(OLink))))
        exit(OVERFLOW); // 存储分配失败
    if (!(M.chead = (OLink*)malloc((M.nu+1)*sizeof(OLink))))
        exit(OVERFLOW); // 存储分配失败
    M.rhead[ ] = M.chead[ ] = NULL ;// 初始化行列头指针向量；各行列链表为空链表
    for ( scanf(&i, &j, &e);  i!=0;  scanf(&i, &j, &e))
    {                                                  // 按任意次序输入非零元
        if (!(p = (OLNode*)malloc(sizeof(OLNode )))) exit(OVERFLOW);
          p->i = i; p->j = j; p->e = e;                // 生成结点
        if (M.rhead[i]==NULL || j < M.rhead[i]->j) {
          p->right = M.rhead[i]; M.rhead[i] = p;
        }
```

矩阵的压缩存储——十字链表

```
        else {                                  // 寻查在行表中的插入位置
            for( q = M.rhead[i]; q->right && q->right->j < j; q=q->right; );
            p->right = q->right; q ->right = p;
        }                                       // 完成行插入
        if (M.chead[j] ==NULL || i < M.chead[j]->i) {
            p->down = M.chead[j];  M.chead[j] = p;
         }
        else {                                  // 寻查在列表中的插入位置
            for( q = M.chead[j]; q->down && q->down->i < i; q = q->down; );
            p->down = q->down; q->down = p;
        }                                       // 完成列插入
    } // for
    return OK;
} // CreateSMatrix_OL
```

广义表的定义



- 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

广义表的定义

- 广义表（Lists，又称列表）是线性表的推广。在第2章中，我们把线性表定义为n>=0个元素$a_1,a_2,a_3,\ldots,a_n$的有限序列。线性表的元素仅限于原子项，原子是作为结构上不可分割的成分，它可以是一个数或一个结构，若放松对表元素的这种限制，容许它们具有其自身结构，这样就产生了广义表的概念。

- 广义表是n(n>=0)个元素$a_1,a_2,a_3,\ldots,a_n$的有限序列，其中a_i或者是原子项，或者是一个广义表。通常记作LS=（$a_1,a_2,a_3,\ldots,a_n$)。LS是广义表的名字，n为它的长度。若a_i是广义表，则称它为LS的子表。

广义表的定义

- 例1：

  D=(E，F)

  E=(a, (b,c))

  F=(d, (e))

- 例2

  A=( )

  B=(a, B)=(a,(a,(a, ……,)))

  C=(A, D, F)

广义表的定义

- 抽象数据类型广义表的定义如下：

ADT GList {

数据对象：

D＝{ e_i | i=1,2,..,n; n≥0; e_i ∈AtomSet 或 e_i ∈GList, AtomSet为某个数据对象 }

数据关系：

R1＝{< e_{i-1} , e_i >| e_{i-1} , e_i ∈D, 2≤i≤n }

广义表的定义

- 从上述定义可知，广义表LS=（$a_1,a_2,a_3,\ldots,a_n$)兼有线性结构和层次结构的特性，归纳如下：
  - 1. 广义表中的数据元素有固定的相对次序；
  - 2. 广义表的长度定义为最外层括弧中包含的数据元素个数；
  - 3. 广义表的深度定义为广义表书写形式中括弧的最大重数；
  - 4. 广义表可被其它广义表所共享；
  - 5. 广义表可以是一个递归的表，递归表的深度可以是无穷，但长度有限；
  - 6. 对于任意一个非空广义表LS=（a1,a2,a3,…,an），它的第一个数据元素被定义为广义表的“表头”，Head(LS)=a1，而由其余数据元素构成的广义表 被定义为广义表的“表尾”，Tail(LS)= (a2,a3,…,an)。

广义表的定义

- 例如：

LS = (A, D) = (( ), (E, F)) = (( ), ((a, (b, c)), F))

Head(LS) = A　　Tail(LS) = (D)

Head(D) = E　　Tail(D) =(F)

Head(E) = a　　Tail(E) = ((b,c))

Head(((b,c))) = (b,c)　　Tail (((b,c))) = ( )

Head((b,c)) = b　　Tail((b,c)) = (c)

Head((c)) = c　　Tail((c)) = ( )

广义表的定义

基本操作：

{结构初始化}

InitGList(**&**L);

操作结果：创建空的广义表 L。

CreateGList(**&**L, S);

初始条件：S是广义表的书写形式串。

操作结果：由S创建广义表 L。

CopyGList(**&**T, L);

初始条件：广义表 L 存在。

操作结果：由广义表 L 复制得到广义表 T。

{销毁结构}

DestroyGList(**&**L);

初始条件：广义表 L 存在。

操作结果：销毁广义表 L。

广义表的定义

{引用型操作}

GListLength(L);

初始条件：广义表 L 存在。

操作结果：求广义表 L 的长度，即元素个数。

GListDepth(L);

初始条件：广义表 L 存在。

操作结果：求广义表 L 的深度。

GListEmpty(L);

初始条件：广义表 L 存在。

操作结果：判定广义表 L 是否为空表。

GetHead(L);

初始条件：广义表 L 存在且非空。

操作结果：返回广义表 L 的表头。

GetTail(L);

初始条件：广义表 L 存在且非空。

操作结果：返回广义表 L 的表尾。

广义表的定义

{加工型操作}

InsertFirst_GL(&L, e);

初始条件：广义表 L 存在。

操作结果：插入元素 e 作为广义表 L 的第一个元素。

DeleteFirst_GL(&L, &e);

初始条件：广义表 L 存在。

操作结果：删除广义表 L 中第一个元素，并用 e 返回其值。

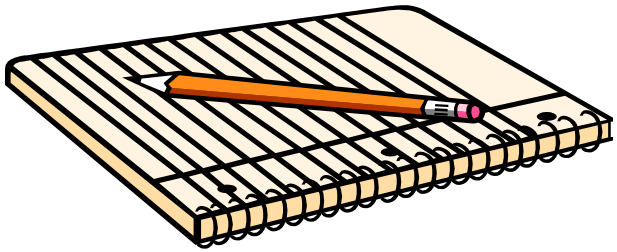
Traverse_GL(L, Visit());

初始条件：广义表 L 存在。

操作结果：遍历广义表 L，用函数 Visit 处理每个元素。

} ADT GList

广义表的存储结构



- 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

广义表的存储结构

- 由于广义表中的数据元素可以是原子，也可以是广义表，显然难以用顺序存储结构表示之，并且为了在存储结构中便于分辩原子和子表，令表示广义表的链表中的结点为“异构”结点，结点中设有一个“标志域tag”， 并约定 tag=0 表示原子结点，tag=1 表示表结点。原子结点中的 data 域存储原子，表结点中指针域的两个值分别指向表头和表尾。
- 空表：ls=Nil
- 非空表：ls指向表节点

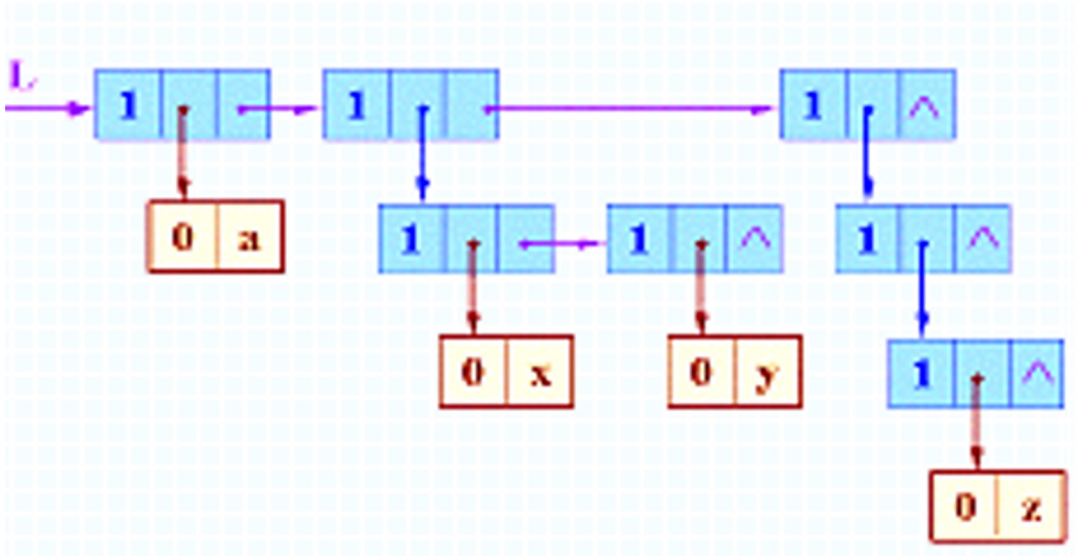
广义表的存储结构

➔ // 广义表的存储表示

```
typedef enum {ATOM=0, LIST=1} ElemTag;        // ATOM(=0)标志原子，
                                                 LIST(=1)标志子表
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 data;        // data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
```

广义表的存储结构

- 例如，广义表 L=(a,(x,y),((z))) 的存储结构如下图所示。它可由对L进行表头和表尾的分析得到。



➔ 演示

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 。