第五章　数组和广义表

本章讨论的两种数据结构：数组和广义表，其共同特点是：

1）从逻辑结构上看它们，可看成是线性结构的一种扩展；

2）数据元素本身也是一个数据结构；

本章讨论三个方面的内容：

数组、 矩阵的压缩存储、广义表

5. 1 数组的定义

数组的概念

数组是我们十分熟悉的，几乎所有的程序设计语言都包含数组。本书在讨论各种数据结构的顺序存储分配时，也都是借用一维数组来描述它们的存储结构。

数组是由一组个数固定，类型相同的数据元素组成阵列。

由于数组中各元素具有统一的类型，并且数组元素的下标一般具有固定的上界和下界，因此，数组的处理比其它复杂的结构更为简单。多维数组是向量的推广。例如，二维数组：

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

可以看成是由m个行向量组成的向量，也可以看成是n个列向量组成的向量。

n维数组中的每个元素都受n个线性关系的约束，即在每个关系中，每个元素a_{ij}都有且仅有一个直接前趋，都有且仅有一个直接后继。

以二维数组为例：二维数组中的每个元素都受两个线性关系的约束。

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

在行关系中

a_{ij}直接前趋是 a_{ij-1}

a_{ij}直接后继是 a_{ij+1}

在列关系中

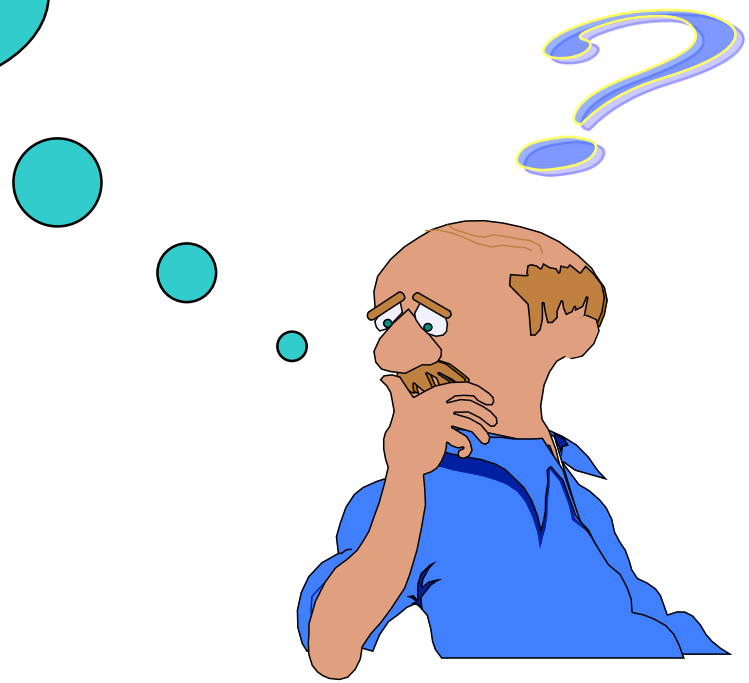
a_{ij}直接前趋是 a_{i-1j}

a_{ij}直接后继是 a_{i+1j}

数据元素之间在每个关系中具有线性特性，但在整个结构中呈非线性关系。

5．2 数组的顺序表示和实现

数组在计算机的内存空间中是如何存储？



数组的顺序存储结构

一般来说，数组一旦定义，其元素的个数和元素之间的相互关系不再发生变化，即对数组一般不进行插入和删除操作。因此，数组采用顺序存储结构是十分自然的事情。

计算机的内存空间是一个一维结构，而二维以上的数组是多维结构。用一维内存来表示多维数组，就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在存储器中。 因此，用一组连续的存储单元存放数组元素，就有次序约定的问题。

通常有两种顺序存储方式：

(1)以行为主序的方式(行优先顺序)——将数组元素按行优先关系排列，第i+1个行向量紧接在第i个行向量后面。

在PASCAL、 COBOL、C及扩展BASIC 中，数组就是按行优先顺序存储的。

(2)以列序为主序的方式（列优先顺序）——将数组元素按列优先关系排列，第j+1个列向量紧接在第j个列向量之后。

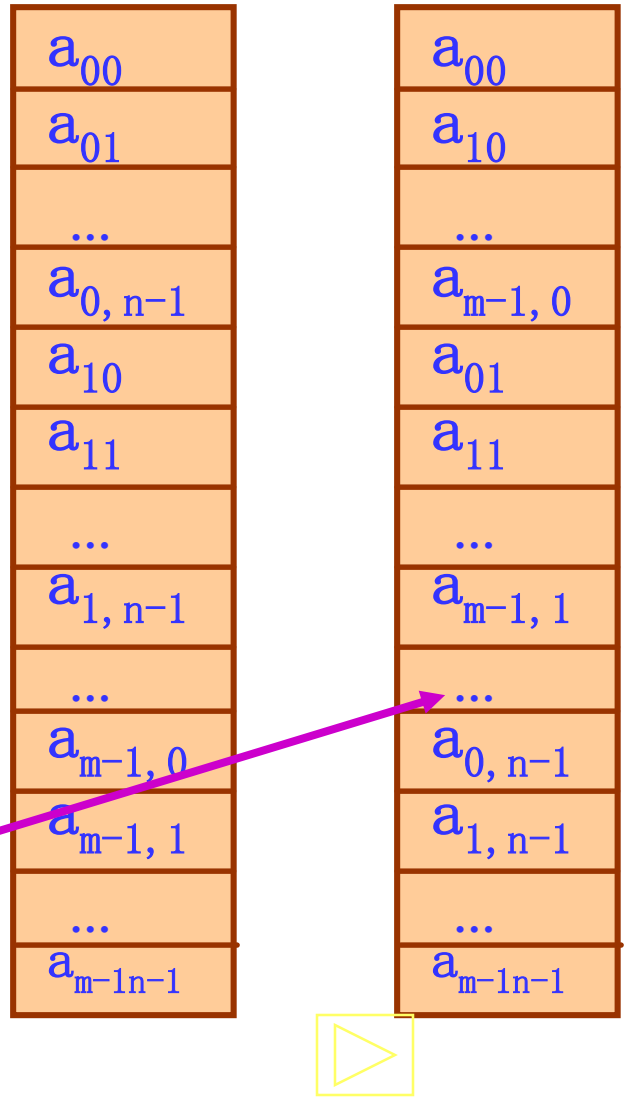
在FORTRAN语言中，数组就是按列优先顺序存储的。

设A是一个具有m 行n列的元素的二维数组（借助矩阵形式给出比较直观）如下：

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

以行为主序的方式：

以列为主序的方式：



数组元素存储地址的计算

无论采用哪种存储方式，
确定了存储映象的首地址，
数组中任意元素的存储地
址是不是都可以确定？

当然是的，你看…

数组元素存储地址的计算

假设二维数组A_{mn}每个元素占用 L 个存储单元， $Loc(a_{ij})$ 为元素a_{ij} 的存储地址，$Loc(a_{00})$ 是a_{00}存储位置，也是二维数组A的基址。

若以行序为主序的方式存储二维数组，则元素a_{ij} 的存储位置可由下式确定：

$$Loc(a_{ij}) = Loc(a_{00}) + (n \times i + j) \times L$$

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

若以列序为主序的方式存储二维数组，则元素a_{ij}的存储位置可由下式确定：

$$Loc(a_{ij}) = Loc(a_{00}) + (m \times j + i) \times L$$

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

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。

1、设有数组A[i,j]，数组的每个元素长度为3字节，i的值为1 到8 ，j的值为1 到10，数组从内存首地址BA开始顺序存放，当用以列为主存放时，元素A[5，8]的存储首地址为(　　)。

A. BA+141　　B. BA+180　　C. BA+222　　D. BA+225

2、设有数组A[i,j]，数组的每个元素长度为3字节，i的值为1到8 ，j的值为1 到10，数组从内存首地址BA开始顺序存放，当用以行为主存放时，元素A[8，5]的存储首地址为(　　)。

A. BA+141　B. BA+180　　C. BA+222　　D. BA+225

3、二维数组M的元素是4个字符(每个字符占一个存储单元)组成的串，行下标i的范围从0到4，列下标j的范围从0到5，M按行存储时元素M[3][5]的起始地址与M按列存储时元素_____的起始地址相同。

A. M[2][4]　　B. M[3][4]　　C. M[3][5]　　D. M[4][4]

4、数组A中，每个元素的长度为3个字节，行下标i的范围从1到8，列下标j的范围从1到10，从首地址SA开始连续存放在存储器内，存放该数组至少需要的单元数是 _____。

A. 80　　B. 100　　C. 240　　D. 270

小 结

1　n维数组中的每个元素都受n个线性关系的约束，在每个关系中，每个元素 a_{ij}都有且仅有一个直接前趋，都有且仅有一个直接后继。

2　二维数组有两种顺序存储方式：一种是以行为主序的方式，另一种是以列序为主序的方式。

5. 3　矩阵的压缩存储

基本内容

1　　特殊矩阵的压缩存储 ；

2　　稀疏矩阵的压缩存储；

3　　稀疏矩阵在三元组顺序表存储方式下，矩阵的运算的实现；

采用“压缩存储”：

1. 为多个相同的非零元素只分配一个存储空间；
2. 对零元素不分配空间。

考虑特殊情况：

1、矩阵中非零元素呈某种规律分布

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

2、矩阵中出现大量的零元素

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 7 & 0 & 0 & 0 & 6 \end{pmatrix}$$

5.3.1特殊矩阵

所谓特殊矩阵是指值相同的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。

如右图便是一个5阶对称矩阵。

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

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

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：

$$\sum_{i=0}^{n-1}(i+1)=\frac{n(n+1)}{2}$$

因此，我们可以按图中箭头所指的次序将这些元素存放在一个向量sa[0..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$a_{i\ j}$和sa[k]之间找一个对应关系。

若i≧j，则$a_{i\ j}$在下三角形中。 $a_{i\ j}$之前的i行（从第0行到第i-1行）一共有

$$1+2+\cdots+i=\frac{i(i+1)}{2}$$

个元素，在第i行上， $a_{i\ j}$之前恰有j个元素（$a_{i0}, a_{i1}, a_{i2}, \ldots, a_{i\ j-1}$）
因此有：

$$k=\frac{i\times(i+1)}{2}+j$$

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：

$$k=\frac{j\times(j+1)}{2}+i \qquad 0\le k\le \frac{n(n+1)}{2}-1$$

A中任意一元素 a_{ij}与它的存储位置之间存在着如下对应关系：

$$k=\begin{cases}\dfrac{i\times(i+1)}{2}+j & \text{当 } i\ge j\\[2ex] \dfrac{j\times(j+1)}{2}+i & \text{当 } i<j\end{cases}$$

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n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*(i+1)/2+j=2*(2+1)/2+1=4

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。

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

(a) 上三角矩阵

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

(b) 下三角矩阵

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，**

5.3.2 稀疏矩阵

1 什么是稀疏矩阵

简单说，**设矩阵A中有s个非零元素，若s远远小于矩阵元素的总数，且零元素的分布没有规律**，则称A为稀疏矩阵。

$$M = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

M有42（6×7）个元素

有8个非零元素

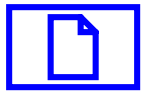
在存储稀疏矩阵时，为了节省存储单元，很自然地想到使用压缩存储方法。但由于非零元素的分布一般是没有规律的，因此在存储非零元素的同时，还必须同时记下它所在的行和列的位置（i，j）。反之，一个三元组（i，j，a_{ij}）唯一确定了矩阵A的一个非零元。因此，稀疏矩阵可由表示非零元的三元组及其行列数唯一确定。

$$M = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

例如，下列三元组表

((1, 2, 12) (1, 3, 9), (3, 1, 3), (3, 6, 14), (4, 3, 24), (5, 2, 18), (6, 1, 15), (6, 4, -7)) 加上 (6, 7) 这一对行、列值便可作为下列矩阵M的另一种描述。

$$M = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$



而由上述三元组表的不同表示方法可引出稀疏矩阵不同的压缩存储方法。

一、三元组顺序表

假设以顺序存储结构来表示三元组表，则可得到稀疏矩阵的一种压缩存储方法——三元顺序表。

稀疏矩阵的三元组顺序表的类型定义

```
#define MAXSIZE                          零元个数的最大值为12500
Typedef struct {
        int          i,
        ElemTy
}Triple;

Typedef  union {
    Triple    data[MAXSIZE+1];  //用于存储非零元三元组表, data[0]未用
       int        mu,nu,tu;                 //矩阵的行数、列数和非零元个数
}TSMatrix;
```

Triple：是包含三个域的结构类型，其变量用于存储矩阵的非零元三元

data：一维数组，用于存储矩阵的非零元三元组表

设M是 TSMatrix 类型的结构变量，则M有四个域，其中M.data用于存储矩阵M的三元组表，在此我们约定，M.data域中非零元三元组是以行序为主序顺序存储的。

$$M=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

M的三元组顺序表图示

M.data

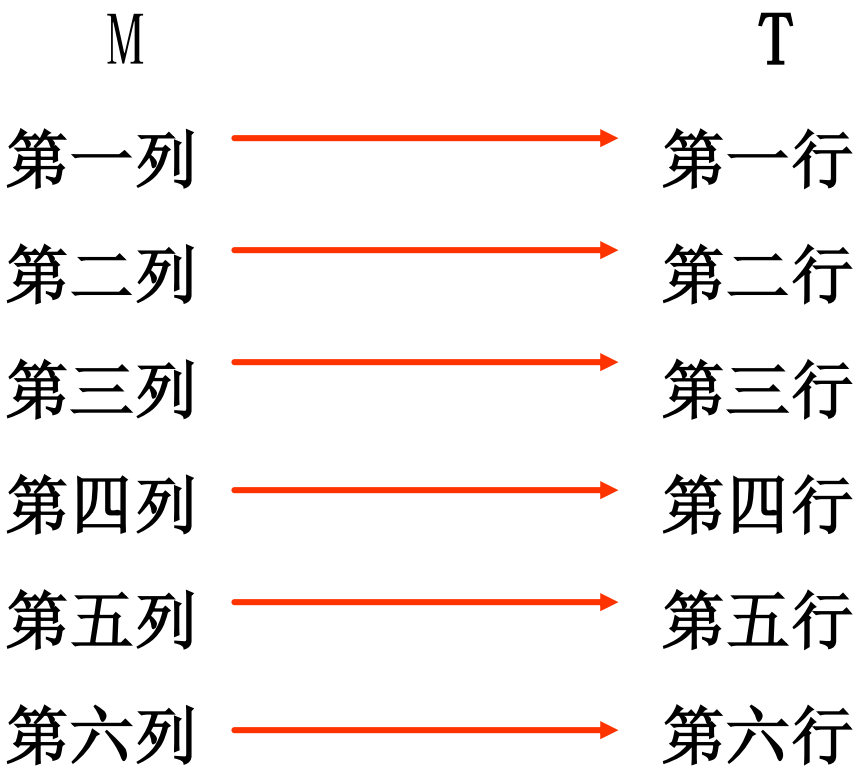
	i	j	e
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

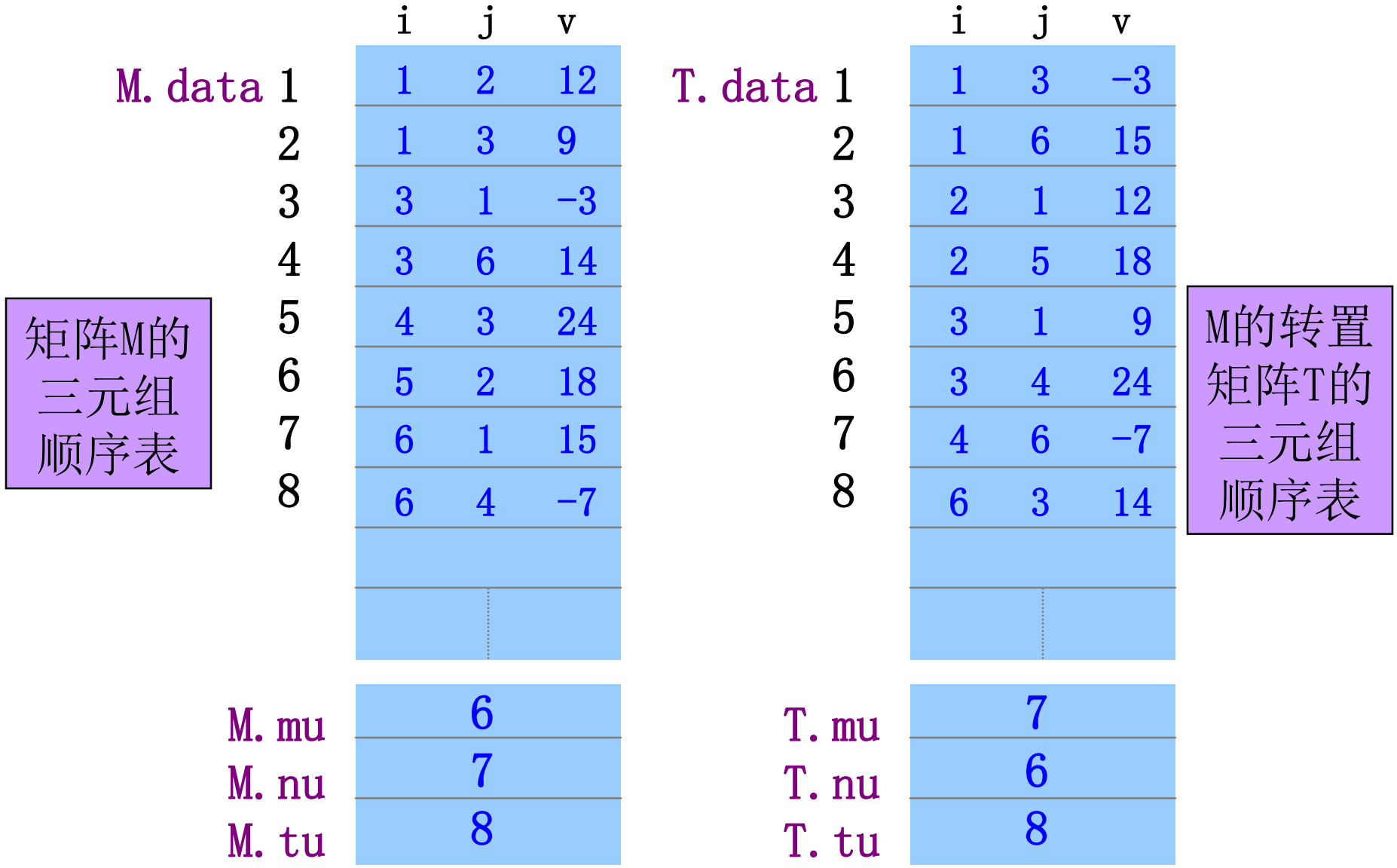
M.mu	6
M.nu	7
M.tu	8

转置运算算法

转置运算是一种最简单的矩阵运算。对于一个m行 n列的矩阵M，它的转置矩阵T是一个n行m列的矩阵。例如，下图中的矩阵M和T互为转置矩阵。

$$M=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix} \qquad T=\begin{pmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$


M
T
第一列
第一行
第二列
第二行
第三列
第三行
第四列
第四行
第五列
第五行
第六列
第六行


i j v
M.data 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
矩阵M的三元组顺序表
M.mu 6
M.nu 7
M.tu 8
i j v
T.data 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
M的转置矩阵T的三元组顺序表
T.mu 7
T.nu 6
T.tu 8

1) 转置运算算法TransposeSMatrix(TSMatrix M, TSMatrix &T)

基本思想:

对M.data从头至尾扫描:

第一次扫描时，将M.data中列号为1的三元组赋值到T.data中，

第二次扫描时，将M.data中列号为2的三元组赋值到T.data中，

依此类推，直至将M.data所有三元组赋值到T.data中

转置运算算法

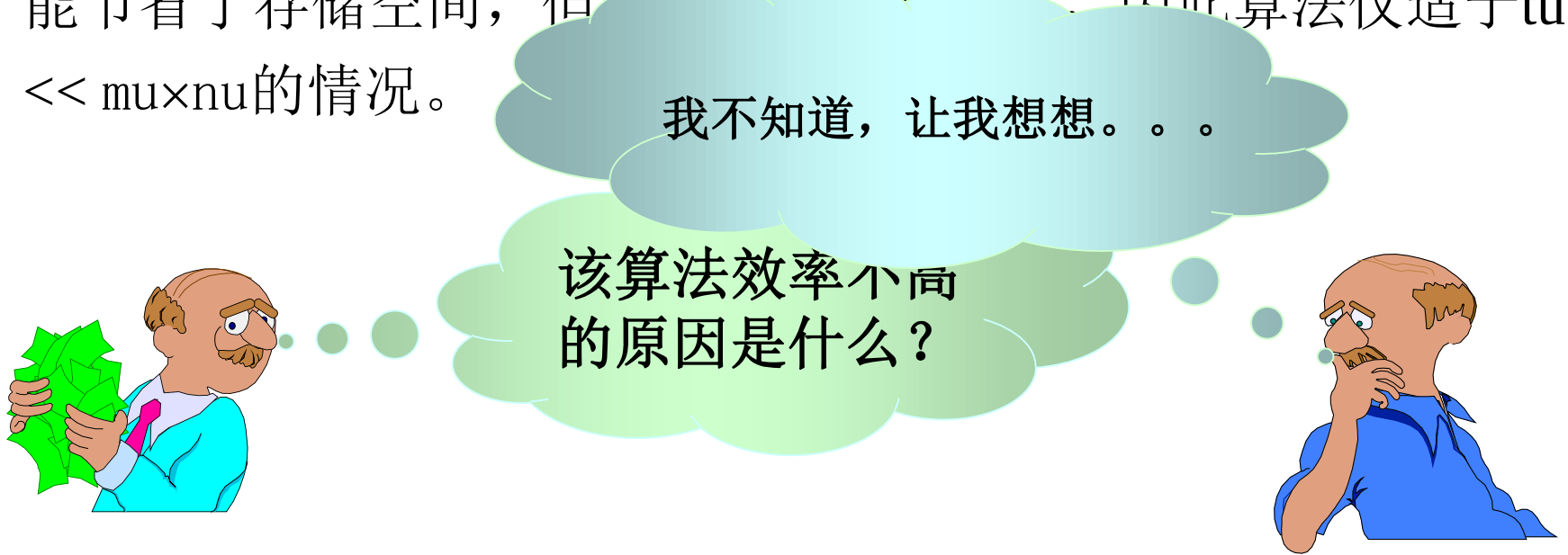
```
Status TransposeSMatrix(TSMatrix M, TSMatrix &T) {
  //采用三元组表存储表示，求稀疏矩阵M的转置矩阵T
  T.mu=M.nu;  T.nu=M.mu;  T.tu=M.tu;
  if (T.tu) {
    q=1;              // q为当前三元组在T.data[ ]存储位置(下标)
    for (col=1; col<=M.nu;  ++col)
      for (p=1;  p<=M.tu;  ++p)     //p为扫描M.data[ ]的“指示器”
                                    //p“指向”三元组称为当前三元
        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;
}//  TransposeSMtrix
```

算法演示

时间复杂度分析

算法的基本操作为将M.data中的三元组赋值到T.data，是在两个循环中完成的，故算法的时间复杂度为**O(nu×tu)**

我们知道，若用二维数组存储矩阵，转置算法的时间复杂度为O（ mu×nu） 。当非零元的个数tu和矩阵元素个数mu×nu 同数量级时，转置运算算法1的时间复杂度为O（ mu×nu × nu）。由此可见：在这种情况下，用三元组顺序表存储矩阵，虽然可能节省了存储空间，但[illegible]因此算法仅适于tu << mu×nu的情况。

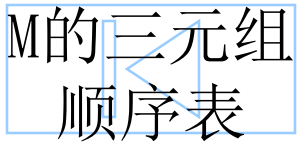


该算法效率不高的原因是：为实现M到T的转置，该算法对M.data进行了多次扫描。能否在对M.data一次扫描的过程中，完成M到T 的转置？

快速转置算法

分析

在M.data中，M的各列非零元三元组是以行为主序存储的，故M的各列非零元对应的三元组存储位置不是“连续”的。然而，M的各列非零元对应的三元组在T中的存储位置是“连续”的。



如果能先求得M各列第一个非零元三元组在T.data中的位置，就能在对M.data一次扫描的过程中，完成M到T 的转置：

对M.data一次扫描时， 首先遇到各列的第一个非零元三元组，可按先前求出的位置，将其放至T.data中，当再次遇到各列的非零元三元组时，只须顺序放到对应列元素的后面。

辅助向量num[ ]、cpos[ ]

为先求得M各列第一个非零元三元组在T.data中的位置。引入两个辅助向量num[ ]、cpot[ ]：

num[col]：存储第col列非零元个数

cpot[col]：存储第col列第一个非零元三元组在T.data中的位置

cpot[col]的计算方法：

$$\begin{cases} cpot[1]=1 \\ cpot[col]=cpot[col-1]+num[col-1] & 2 < co \end{cases}$$

例 矩阵M

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

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

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

各列第一个非零元三元组按先前求出的位置，放至T.data中

各列后续的非零元三元组，顺序放到对应列元素的后面

M.data

	i	j	v
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

T.data

	i	j	v
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

M.mu	6
M.nu	7
M.tu	8

T.mu	7
T.nu	6
T.tu	8

快速转置算法主要步骤：

1　　求M中各列非零元个数num[ ];

2　　求M中各列第一个非零元在T.data中的下标cpot[ ];

3　　对M.data进行一次扫描， 遇到col列的第一个非零元三元组时，按cpot[col]的位置，将其放至T.data中，当再次遇到col列的非零元三元组时，只须顺序放到col列元素的后面;

```
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中每一列非零元个数
      cpot[1]=1;
       //求第 col列中第一个非零元在T.data中的序号
       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];
       }//for
      }//if
  return OK;
}//FastTransposeSMatrix
```

算法演示

时间复杂度分析

该算法利用两个辅助向量num[ ]、cpos[ ]，实现了对M.data一次扫描完成M到T 的转置。

从时间上看，算法中有四个并列的单循环，循环次数分别为nu、tu、nu和tu，因而总的时间复杂度为O（nu+tu)，在M的非零元个数tu和mu×nu 同等数量级时，其时间复杂度为O(mu×nu)，和转置算法5.1的时间复杂度相同。由此可见，**只有当tu<< mu×nu时，使用快速转置算法才有意义。**

小 结

1 **矩阵压缩存储**是指为多个值相同的元素分配一个存储空间，对零元素不分配存储空间；

2 **特殊矩阵的压缩存储**是根据元素的分布规律，确定元素的存储位置与元素在矩阵中的位置的对应关系；

3 **稀疏矩阵的压缩存储**除了要保存非零元素的值外，还要保存非零元素在矩阵中的位置；

5．4　广义表

基本内容

1　广义表的概念；

2　广义表的基本操作；

广义表的概念

1　什么是广义表

广义表也称为**列表**，是线性表的一种扩展，也是数据元素的有限序列。

记作：$LS=(\alpha_1, \alpha_2, \ldots, \alpha_n)$。

其中α_i其可以是单个元素，也可以是广义表。

1）LS是**广义表的名称**，n 是**广义表长度**；

2）　广义表的定义是一个**递归定义**，因为在描述广义表时又用到了广义表；

3）在线性表中数据元素是单个元素，而在广义表中，　元素可以是单个元素称为**原子**，也可以是广义表，称为**广义表的子表**；

4)广义表例：

A = （ ） 空表，表长为0；

B = (e) B中只有一个原子e，表长为1；

C = (a, (b, c, d)) C的表长为2，两个元素分别为原子a 和子表

B = (e) 表头： e 表尾 ( )

C = (a,(b,c,d)) 表头： a 表尾 ((b,c,d))

D = (A,B,C) 表头： A 表尾 (B,C)

是

广

E = （ a, E ） 递归表，长度为2

5） 对非空广义表LS：

称第一个元素α_1为LS的表头；

其余元素组成的表$(\alpha_2, \ldots, \alpha_n)$称为LS的表尾；

2　广义表的基本操作

1）创建空的广义表L；
2）销毁广义表L；
3）已有广义表L，由L复制得到广义表T；
4）求广义表L的长度；
5）求广义表L的深度；
6）判广义表L是否为空；
7）取广义表L的表头；
8）取广义表L的表尾；
9）在L中插入元素作为L的第一个元素；
10）删除广义表L的第一个元素，并e用返回其值；
11）遍历广义表L，用函数visit( )处理每个元素；

小 结

1　广义表是数据元素的有限序列。其数据元素可以单个元素，也可以是广义表；

2　若广义表不空，则可分成表头和表尾，反之，一对表头和表尾可唯一确定广义表；

第五章　　习 题

1、 P32 5.1

2、 P33 5.10 1） 2） 3） 4） 5）

求各列非零元个数

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

求各列第个非零元b中位置

扫描M.data实现M到T的转置

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

M.data

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

T.data

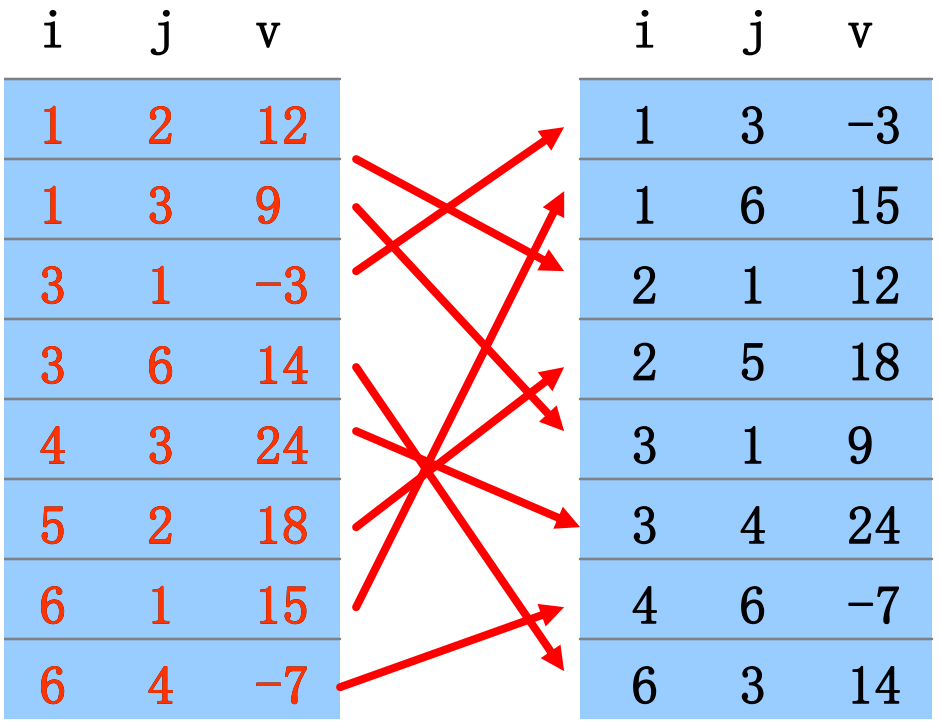
1 2 3 4 5 6 7 8

返回算法

快速转置算法图示

转置运算算法图示

第六次扫描查找
第6列元素



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

M.data

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

T.data

对M六次扫描完成转置运算

返回算法