



主要内容

第五章 数组和广义表

- 数组的定义
- 数组的顺序表示
- 矩阵的压缩存储（特殊矩阵，稀疏矩阵）
- 广义表的定义
- 广义表的存储结构

2/68



线性表——具有相同类型的数据元素的有限序列



限制插入、删除位置

栈——仅在表的一端进行插入和删除操作

队列——在一端进行插入操作，而另一端进行删除操作



限制数据元素的类型

串——数据元素为字符的有限序列



将元素类型扩充为线性表

（多维）**数组**——线性表中的数据元素可以是线性表

3/68



5.1 数组的定义

数组是我们最熟悉的数据类型，在早期的高级语言中，**数组是唯一可供使用的数据类型**。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。

4/68



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, \dots, 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。

5/68



$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1,n} \\ a_{21} & a_{22} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix}$$

数组的特点：
元素数目固定；
下标有界；

数组的操作：
按照下标进行读写

6/68

ADT Array{

数据对象: $j_i = 0, 1, \dots, b_i - 1, 1, 2, \dots, n$;

$D = \{ a_{j_1 j_2 \dots j_n} \mid n > 0 \}$ 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, $a_{j_1 j_2 \dots j_n} \in \text{ElemSet}$

数据关系: $R = \{ R_1, R_2, \dots, R_n \}$

$R_i = \{ \langle a_{j_1 j_2 \dots j_i \dots j_n}, a_{j_1 j_2 \dots j_{i+1} \dots j_n} \rangle \mid 0 \leq j_k \leq b_k - 1, 1 \leq k \leq n \text{ 且 } k \neq i, 0 \leq j_i \leq b_i - 2, a_{j_1 j_2 \dots j_i \dots j_n} \in D \}$

基本操作:

} ADT Array

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



7/68

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

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

其中每个数据元素 α_i 是一个列向量(线性表):


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

或是一个行向量:

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



8/68



$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \quad (a) \text{ 矩阵表示形式}$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \quad (b) \text{ 列向量的一维数组形式}$$

$$A = \left(\begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix}, \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix}, \dots, \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} \right) \quad (c) \text{ 行向量的一维数组形式}$$

9/68

5.2 数组的顺序表示和实现

由于对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。

计算机的内存结构是一维的, 因此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。

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。



10/68

通常有两种顺序存储方式:

(1) 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:

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

在PASCAL、C语言中, 数组就是按行优先顺序存储的。


(2) 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 $m \times n$ 个元素按列优先顺序存储的线性序列为:

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

在FORTRAN语言中, 数组就是按列优先顺序存储的。



11/68



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

a_{11}	a_{12}	...	a_{1n}	a_{21}	a_{22}	...	a_{2n}	...	a_{m1}	a_{m2}	...	a_{mn}
----------	----------	-----	----------	----------	----------	-----	----------	-----	----------	----------	-----	----------

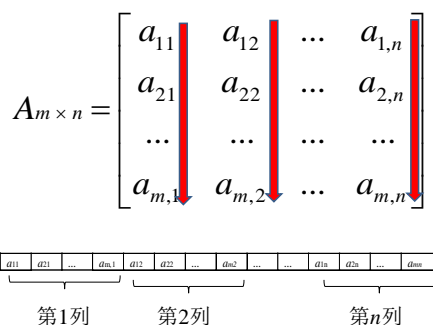
第1行

第2行

第 m 行

行优先存储

12/68



列优先存储

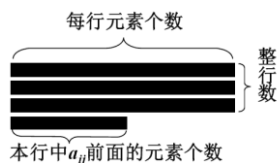
13/68

以上规则可以推广到多维数组的情况：优先顺序可规定为前排最右的下标，从右到左，最后排最左下标：列优先顺序与此相反，前排最左下标，从左向右，最后排最右下标。

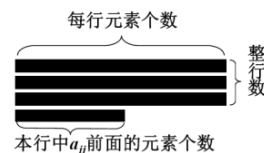
按上述两种方式顺序存储的数组，只要知道开始结点的存放地址（即基地址），维数和每维的上、下界，以及每个数组元素所占用的单元数，就可以将数组元素的存放地址表示为其下标的线性函数。

14/68

例如，二维数组 A_{mn} 按“行优先顺序”存储在内存中，假设每个元素占用 d 个存储单元。元素 a_{ij} 的存储地址应是数组的基地址加上排在 a_{ij} 前面的元素所占用的单元数。因为 a_{ij} 位于第 i 行、第 j 列，前面 $i-1$ 行一共有 $(i-1) \times n$ 个元素，第 i 行上 a_{ij} 前面又有 $j-1$ 个元素，故它前面一共有 $(i-1) \times n + j - 1$ 个元素。



15/68



因此， a_{ij} 的地址计算函数为：

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

同样，三维数组 A_{ijk} 按“行优先顺序”存储，其地址计算函数为：

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

16/68

更一般的二维数组是

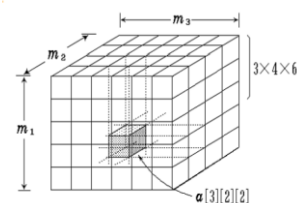
$$A[c_1..d_1, c_2..d_2]$$

因此， a_{ij} 的地址计算函数为：

$$\text{LOC}(a_{ij}) = \text{LOC}(a_{c_1 c_2}) + [(i - c_1) \times (d_2 - c_2 + 1) + j - c_2] \times d$$

17/68

n ($n > 2$) 维数组一般也采用按行优先和按列优先两种存储方法。



$$\text{Loc}(a_{ijk}) = \text{Loc}(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times c$$

18/68

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。

19/68

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：

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

则称 A 为对称矩阵。

$$A = \begin{pmatrix} 3 & 6 & 4 & 7 & 8 \\ 6 & 2 & 8 & 4 & 2 \\ 4 & 8 & 1 & 6 & 9 \\ 7 & 4 & 6 & 0 & 5 \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix} \quad A = \begin{pmatrix} 3 & c & c & c & c \\ 6 & 2 & c & c & c \\ 4 & 8 & 1 & c & c \\ 7 & 4 & 6 & 0 & c \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$

20/68

对称矩阵的特点是 $a_{ij}=a_{ji}$ 。一个 $n \times n$

的方阵，共有 n^2 个元素，但只需要对称矩阵中 $n(n+1)/2$ 个元素进行存储表示。

$$A = \begin{pmatrix} 3 & c & c & c & c \\ 6 & 2 & c & c & c \\ 4 & 8 & 1 & c & c \\ 7 & 4 & 6 & 0 & c \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix} \quad A = \begin{pmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{pmatrix}$$

21/68

假设按“**行优先顺序**”存储下三角形(包括对角线)中的元素。顺序”存储主对角线(包括对角线)以下的元素，其存储形式如图所示：

$$\begin{pmatrix} 1 & 5 & 1 & 3 & 7 \\ 5 & 0 & 8 & 0 & 0 \\ 1 & 8 & 9 & 2 & 6 \\ 3 & 0 & 2 & 5 & 1 \\ 7 & 0 & 6 & 1 & 3 \end{pmatrix} \quad \begin{matrix} a_{11} \\ a_{21} \quad a_{22} \\ a_{31} \quad a_{32} \quad a_{33} \\ \dots \\ a_{n1} \quad a_{n2} \quad a_{n3} \dots a_{nn} \end{matrix}$$

sa[0] 1 2 3

a ₁₁	a ₂₁	a ₂₂	a ₃₁	a _{n1}	...	a _{nn}
-----------------	-----------------	-----------------	-----------------	-------	-----------------	-----	-----------------

计算 a_{ij} 存储在哪里？

22/68

若 $i \geq j$ ： 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： $1+2+\dots+(i-1)=i \times (i-1)/2$

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：

$$k = i \times (i-1)/2 + j - 1 \quad i \geq j$$

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：

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

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：

$$k = \begin{cases} i \times (i-1)/2 + j - 1 & \text{当 } i \geq j \text{ 时} \\ j \times (j-1)/2 + i - 1 & \text{当 } i < j \text{ 时} \end{cases} \quad 1 \leq i, j \leq n$$

23/68

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。

$$\begin{pmatrix} a_{00} & a_{01} & \dots & a_{0 \ n-1} \\ c & a_{11} & \dots & a_{1 \ n-1} \\ \dots & \dots & \dots & \dots \\ c & c & \dots & a_{n-1 \ n-1} \end{pmatrix} \quad \begin{pmatrix} a_{00} & c & \dots & c \\ a_{10} & a_{11} & \dots & c \\ \dots & \dots & \dots & \dots \\ a_{n-1 \ 0} & a_{n-1 \ 1} & \dots & a_{n-1 \ n-1} \end{pmatrix}$$

(a)上三角矩阵

(b)下三角矩阵

24/68

三角矩阵的示例

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

其中 n 是一个大整数。理论上，我们可通过迭代法来求解方程，但是该方法太繁琐。

高斯消去法即将原方程组转化为另一个等价的方程组(该系统与原系统有相同的解)，但变换后的方程组的系数矩阵为上三角形矩阵。



25/68



$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 & a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 & a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\vdots & &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n & a'_{nn}x_n &= b'_n \end{aligned} \Rightarrow$$

用矩阵表示如下：

$$Ax = b \Rightarrow A'x = b'$$

26/68

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

(a) 上三角矩阵示例

(b) 下三角矩阵示例



27/68

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 $sa[0 \dots n(n+1)/2]$ 中，再加一个存储 c 的空间即可。



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

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

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

$$K = \begin{cases} i \times (i-1)/2 + j - 1 & \text{当 } i \geq j \text{ 时} \\ n \times (n+1)/2 & \text{当 } i < j \text{ 时} \end{cases} \quad 1 \leq j \leq n$$

28/68

3、稀疏矩阵的压缩存储

若一个 $m \times n$ 的矩阵含有 t 个非零元素，且 t 远远小于 $m \times n$ ，则我们将这个矩阵称为稀疏矩阵。

$$\begin{pmatrix} 3 & 0 & 0 & 0 & 7 \\ 0 & 0 & -1 & 0 & 0 \\ -1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$



29/68



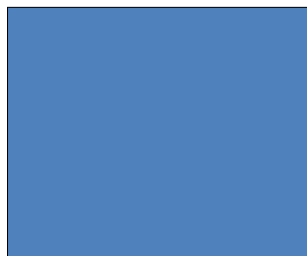
(5,5,6)

(1,1,3), (1,5,7)

(2,3,-1)

(3,1,-1), (3,2,-2)

(5,4,2)



30/68

1) 三元组表示法。

矩阵中的每个元素都是由行序号和列序号唯一确定的。因此，我们需要用三项内容表示稀疏矩阵中的每个非零元素，即形式为：

$(i, j, value)$

其中， i 表示行序号， j 表示列序号， $value$ 表示非零元素的值，通常将它称为三元组。

将非零元素按行优先，顺序存储在三元组中。

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

特点：第一列递增有序。第一列相同，第二列递增有序。

31/68

32/68

Typedef Struct

```
{ int i,j;
  elementtype e;
} Triple;
```

Typedef Struct

```
{ Triple data[Max+1];
  int mu,nu,tu;
} TSMatrix
```

33/68

稀疏矩阵的转置（三元组存储）

3	0	0	0	7
0	0	-1	0	0
-1	-2	0	0	0
0	0	0	0	0
0	0	0	2	0

3	0	-1	0	0
0	0	-2	0	0
0	-1	0	0	0
0	0	0	0	2
7	0	0	0	0

34/68

一个 $m \times n$ 的矩阵 M ，它的转置 T 是一个 $n \times m$ 的矩阵，且 $m[i][j] = t[j][i]$ ， $0 \leq i \leq m$ ，

$0 \leq j \leq n$ ，即 M 的行是 T 的列， M 的列是 T 的行。

由于三元组 M 的列是三元组 T 的行，因此，按 M 的列的列序转置，所得到的转置矩阵 T 的三元组表 T 的三元组表 T 必定是按行优先存放的。

具体方法是将 M 的列进行多次扫描，按从小到大的顺序依次复制到 T 中。

35/68

转置算法

```
Void TransMatrix(TSMatrix M, TSMatrix T)
//M 和 T 是矩阵的三元组表示, T 是转置矩阵
{ T.mu=M.nu; T.nu=M.mu; T.tu=M.tu;
  if (T.tu)
  { q=0;
    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=q+1; }
  }
}
```

表示三元组 T 的下标

按 M 的列值由小到大查找

36/68



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 $O(n*t)$ ，即矩阵的列数和非零元的个数的乘积成正比。

37/68



下面给出另外一种称之为快速转置的算法，其算法思想为：对 M 扫描一次，按 M 第二列提供的列号一次确定位置装入 T 的一个三元组。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。可见，位置关系是此种算法的关键。

38/68



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 中第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 $num[1..n]$ 和 $cpot[1..n]$

$num[1..n]$ ：统计 M 中每列非零元素的个数， $num[col]$ 的值可以由 M 的第二列求得。

39/68



$cpot[1..n]$ ：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。

算法通过 $cpot$ 数组建立位置对应关系：

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

算法的基本思想：依次读入 M 中的元素，根据 $cpot$ 数组可以获得 $M.data[j]$ 在 T 中应有的位置为 $cpot[j]$ ，进行如下操作：

```
T.data[cpot[j]].i= M.data[j].i
T.data[cpot[j]].j= M.data[j].j
T.data[cpot[j]].e= M.data[j].e
cpot[j]=cpot[j]+1
```

40/68

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

$num[1..n]$ ：统计 M 中每列非零元素的个数

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



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

读取第一个
交换行和列
计算新的位置

j	i	v
1		
2		
3	2	12
4		
5		
6		
7		
8		



$num[col]$ 和 $cpot[col]$ 的值如下：

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

41/68

42/68

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

	<i>j</i>	<i>i</i>	<i>v</i>
1			
2			
3	2	1	12
4			
5			
6			
7			
8			

读取第一个
交换行和列
计算新的位置
修改数组

num[col]和 cpot[col]的值如下：

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



43/68

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

	<i>j</i>	<i>i</i>	<i>v</i>
1			
2			
3	2	1	12
4			
5	3	1	9
6			
7			
8			

读取第二个
交换行和列
计算新的位置

num[col]和 cpot[col]的值如下：

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



44/68

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

	<i>j</i>	<i>i</i>	<i>v</i>
1			
2			
3	2	1	12
4			
5	3	1	9
6			
7			
8			

读取第二个
交换行和列
计算新的位置
修改数组

num[col]和 cpot[col]的值如下：

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



45/68

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

	<i>j</i>	<i>i</i>	<i>v</i>
1	1	3	-3
2			
3	2	1	12
4			
5	3	1	9
6			
7			
8			

读取第三个
交换行和列
计算新的位置

num[col]和 cpot[col]的值如下：

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



46/68

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

	<i>j</i>	<i>i</i>	<i>v</i>
1	1	3	-3
2			
3	2	1	12
4			
5	3	1	9
6			
7			
8			

读取第三个
交换行和列
计算新的位置
修改数组

num[col]和 cpot[col]的值如下：

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



47/68

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

	<i>j</i>	<i>i</i>	<i>v</i>
1	1	3	-3
2			
3	2	1	12
4			
5	3	1	9
6			
7			
8	6	3	14

读取第四个
交换行和列
计算新的位置

num[col]和 cpot[col]的值如下：

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



48/68

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

读取第四个
交换行和列
计算新的位置
修改数组

	<i>j</i>	<i>i</i>	<i>v</i>
1	1	3	-3
2			
3	2	1	12
4			
5	3	1	9
6			
7			
8	6	3	14

num[col]和 cpot[col]的值如下：

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

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

读取第五个
交换行和列
计算新的位置

	<i>j</i>	<i>i</i>	<i>v</i>
1	1	3	-3
2			
3	2	1	12
4			
5	3	1	9
6	3	4	24
7			
8	6	3	14

num[col]和 cpot[col]的值如下：

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

49/68

50/68

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

读取第五个
交换行和列
计算新的位置
修改数组

	<i>j</i>	<i>i</i>	<i>v</i>
1	1	3	-3
2			
3	2	1	12
4			
5	3	1	9
6	3	4	24
7			
8	6	3	14

num[col]和 cpot[col]的值如下：

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

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

读取第六个
交换行和列
计算新的位置

	<i>j</i>	<i>i</i>	<i>v</i>
1	1	3	-3
2			
3	2	1	12
4	2	5	18
5	3	1	9
6	3	4	24
7			
8	6	3	14

num[col]和 cpot[col]的值如下：

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

51/68

52/68

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

读取第六个
交换行和列
计算新的位置
修改数组

	<i>j</i>	<i>i</i>	<i>v</i>
1	1	3	-3
2			
3	2	1	12
4	2	5	18
5	3	1	9
6	3	4	24
7			
8	6	3	14

num[col]和 cpot[col]的值如下：

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

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

读取第七个
交换行和列
计算新的位置

	<i>j</i>	<i>i</i>	<i>v</i>
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			
8	6	3	14

num[col]和 cpot[col]的值如下：

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

53/68

54/68

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

	<i>j</i>	<i>i</i>	<i>v</i>
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			
8	6	3	14

读取第七个
交换行和列
计算新的位置
修改数组

num[col]和 cpot[col]的值如下：

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

55/68

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

	<i>j</i>	<i>i</i>	<i>v</i>
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

读取第八个
交换行和列
计算新的位置

num[col]和 cpot[col]的值如下：

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

56/68

快速转置算法如下：

```
void fasttransri(tritupletable b, tritupletable a){
    int p, q, col, k;
    int num[0..a.n], cpot[0..a.n];
    b.m=a.n; b.n=a.m; b.t=a.t;
    if(b.t<=0)
        printf("a=0"\n);
    for(col=1; col<=a.u; ++col)
        num[col]=0;
    for(k=1; k<=a.t; ++k)
        ++num[a.data[k].j];
```

57/68

```
cpot[1]=1;
for(col=2; col<=a.t; ++col)
    cpot[col]=cpot[col-1]+num[col-1];
for(p=1; p<=a.t; ++p){
    col=a.data[p].j; q=cpot[col];
    b.data[q].i=a.data[p].j;
    b.data[q].j=a.data[p].i;
    b.data[q].v=a.data[p].v; ++cpot[col];
}
```

58/68

4、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，结点的结构如图所示。

Row	col	value
down	right	

(a) 结点结构

rn	cn	tn
down	right	

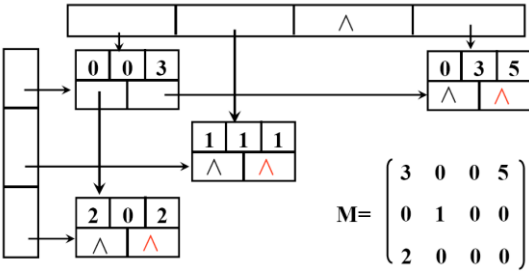
(b) 头结点结构

- 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，还可由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。

59/68

60/68

稀疏矩阵的压缩存储 ----十字链表



61/68

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。

我们把线性表定义为 $n(n \geq 0)$ 个元素 a_1, a_2, \dots, 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。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(又称为列表)：是由 $n(n \geq 0)$ 个元素组成的有穷序列： $LS = (a_1, a_2, \dots, a_n)$

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。

62/68

广义表的性质

示例：

$LS = (a_1, a_2, \dots, a_n)$
 $A = ()$
 $B = (e)$
 $C = (a, (b, c, d))$
 $D = (A, B, C)$
 $E = (a, E)$

广义表的元素可以是子表，
子表的元素还可以是子表；
广义表是一个多层次的结构（层次性）；
一个广义表可以被其他广义表所共享（共享性）。
广义表可以是其本身的子表（递归性）。



63/68

广义表的长度：元素的数目。

广义表的表头：非空广义表中第一个元素。

广义表的表尾：除表头元素之外，其余元素构成的表。

广义表的深度：广义表中括号的重数。



	长度	表头	表尾	深度
$A = ()$	0			1
$B = (e)$	1	e	$()$	1
$C = (a, (b, c, d))$	2	a	$((b, c, d))$	2
$D = (A, B, C)$	3	$()$	(B, C)	3
$E = (a, E)$	2	a	(E)	无穷大

64/68

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。

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	标志tag=1	表头指针hp	表尾指针tp
---------	------	---------	--------	--------

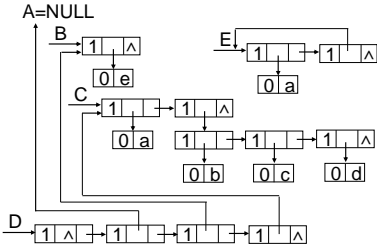
(a) 原子结点

(b) 表结点

广义表的链表结点结构示意图

65/68

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所示。



广义表的存储结构示意图

66/68



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。

67/68



总 结

- 数组的定义
- 数组的顺序表示
- 矩阵的压缩存储（特殊矩阵，稀疏矩阵）
- 广义表的定义
- 广义表的存储结构

68/68