



## Data Structure & Algorithm Analysis

# Array

---

**Zibin Zheng ( 郑子彬 )**

**School of Data and Computer Science , SYSU**

**<http://www.inpluslab.com>**

课程主页: <http://inpluslab.sysu.edu.cn/dsa2016/>

---

**串**——零个或多个字符组成的有限序列



将元素类型限制为字符

**线性表**——具有相同类型的数据元素的有限序列。



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

# 数组与广义表

---

数组是一种人们非常熟悉的数据结构，几乎所有的程序设计语言都支持这种数据结构或将这种数据结构设定为语言的固有类型。

**数组**这种数据结构可以看成**是线性表的推广**。

科学计算中涉及到大量的矩阵问题，在程序设计语言中一般都采用数组来存储，被描述成一个二维数组。但**当矩阵规模很大且具有特殊结构**(对角矩阵、三角矩阵、对称矩阵、稀疏矩阵等)，为减少程序的时间和空间需求，**采用自定义的描述方式**。

**广义表**是另一种推广形式的线性表，是一种灵活的数据结构，在许多方面有广泛的应用。

# 数组的定义

---

数组是由一组**类型相同**的数据元素构成的**有序**集合，每个数据元素称为一个数组元素（简称为元素），每个元素受 $n(n \geq 1)$ 个**线性关系**的约束，每个元素在 $n$ 个线性关系中的序号 $i_1$ 、 $i_2$ 、...、 $i_n$ 称为该元素的下标，并称该数组为  $n$  维数组。

## 数组的特点

- 元素本身可以具有某种结构，属于同一数据类型；
- 数组是一个具有固定格式和数量的数据集合。

# 数组的定义

---

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

例如，元素 $a_{22}$ 受两个线性关系的约束，在行上有一个行前驱 $a_{21}$ 和一个行后继 $a_{23}$ ，在列上有一个列前驱 $a_{12}$ 和一个列后继 $a_{32}$

## 数组——线性表的推广

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



$$A = (A_1, A_2, \dots, A_n)$$

其中:


$$A_i = (a_{1i}, a_{2i}, \dots, a_{mi}) \\ (1 \leq i \leq n)$$

二维数组是数据元素为线性表的线性表


# 数组的基本操作

① 在数组中插入（或删除）一个元素有意义吗？

将元素  $x$  插入  
到数组中第1行第2列。

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


删除数组中  
第1行第2列元素。

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


## 数组的基本操作

---

- (1) 存取：给定一组下标，读出对应的数组元素；
- (2) 修改：给定一组下标，存储或修改与其相对应的数组元素。

存取和修改操作本质上只对应一种操作——**寻址**

① **数组应该采用何种方式存储？**

数组没有插入和删除操作，所以，不用预留空间，适合采用顺序存储。



# 数组的定义

- 抽象数据类型  
ADT Array{

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

$D = \{ a_{j_1 j_2 \dots j_n} \mid n > 0 \text{ 称为数组的维数, } b_i \text{ 是数组第 } i \text{ 维的长度, } j_i \text{ 是数组元素第 } i \text{ 维的下标, } a_{j_1 j_2 \dots j_n} \in \text{ElemSet} \}$

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

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

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。

# 数组的定义

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

(a) 矩阵表示形式

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

(b) 列向量的一维数组形式

$$A = \left( \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} \dots \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} \right)$$

(c) 行向量的一维数组形式

二维数组图例形式

# 数组的定义

---

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

数组的特点：

- 元素数目固定；
- 下标有界；

数组的操作：

- 照下标进行读写

# 数组的顺序表示和实现

---

- 数组一般不做插入和删除操作，也就是说，数组一旦建立，结构中的元素个数和元素间的关系就不再发生变化。因此，一般都是采用顺序存储的方法来表示数组。
- **问题**：计算机的内存结构是一维(线性)地址结构，对于多维数组，将其存放(映射)到内存一维结构时，有个次序约定问题。即必须按某种次序将数组元素排成一系列列，然后将这个线性序列存放在内存中。
- 二维数组是最简单的多维数组，以此为例说明多维数组存放(映射)到内存一维结构时的次序约定问题。

# 数组的顺序表示和实现

---

通常有两种顺序存储方式：

- (1) **行优先顺序(Row Major Order)** ——将数组元素按行排列，第 $i+1$ 个行向量紧接在第 $i$ 个行向量后面。以二维数组为例，按行优先顺序存储的线性序列为： $a_{11}, a_{12}, \dots, a_{1n}, a_{21}, a_{22}, \dots, a_{2n}, \dots, a_{m1}, a_{m2}, \dots, a_{mn}$

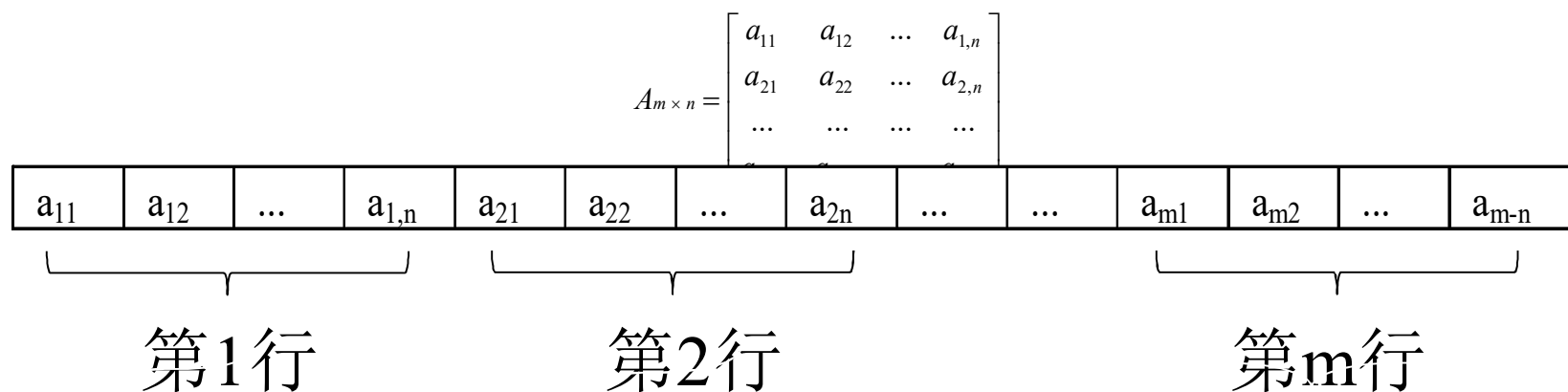
在PASCAL、C语言中，数组就是按行优先顺序存储的。

- (2) **列优先顺序(Column Major Order)** ——将数组元素按列向量排列，第 $j+1$ 个列向量紧接在第 $j$ 个列向量之后， $A$ 的 $m \times n$ 个元素按列优先顺序存储的线性序列为：

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

在FORTRAN语言中，数组就是按列优先顺序存储的。

# 数组的顺序表示和实现



行优先存储

# 数组的顺序表示和实现

---

- 以上规则可以推广到多维数组的情况：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。
- 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。

# 数组的顺序表示和实现

设有二维数组 $A=(a_{ij})_{m \times n}$ ，若每个元素占用的存储单元数为 $l$ (个)， $LOC[a_{11}]$ 表示元素 $a_{11}$ 的首地址，即**数组的首地址**。

- 以“**行优先顺序**”存储

第1行中的每个元素对应的(首)地址是：

$$LOC[a_{1j}] = LOC[a_{11}] + (j-1) \times l \quad j = 1, 2, \dots, n$$

第2行中的每个元素对应的(首)地址是：

$$LOC[a_{2j}] = LOC[a_{11}] + n \times l + (j-1) \times l \quad j = 1, 2, \dots, n$$

... ..

第 $m$ 行中的每个元素对应的(首)地址是：

$$LOC[a_{mj}] = LOC[a_{11}] + (m-1) \times n \times l + (j-1) \times l \quad j = 1, 2, \dots, n$$



# 数组的顺序表示和实现

---

由此可知，二维数组中任一元素 $a_{ij}$ 的(首)地址是：

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

$$i=1,2,\dots,m \quad j=1,2,\dots,n$$

根据(1)式，对于三维数组 $A=(a_{ijk})_{m \times n \times p}$ ，若每个元素占用的存储单元数为 $l$ (个)， $\text{LOC}[a_{111}]$ 表示元素 $a_{111}$ 的首地址，即数组的首地址。以“行优先顺序”存储在内存中。

三维数组中任一元素 $a_{ijk}$ 的(首)地址是：

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

# 数组的顺序表示和实现

推而广之，对 $n$ 维数组 $A=(a_{j_1 j_2 \dots j_n})$ ，若每个元素占用的存储单元数为 $l$ (个)， $LOC[a_{11} \dots 1]$ 表示元素 $a_{11} \dots 1$ 的首地址。则以“**行优先顺序**”存储在内存中。

$n$ 维数组中任一元素  $a_{j_1 j_2 \dots j_n}$  的(首)地址是：

$$\begin{aligned} LOC[a_{j_1 j_2 \dots j_n}] = & LOC[a_{11} \dots 1] + [(b_2 \times \dots \times b_n) \times (j_1 - 1) \\ & + (b_3 \times \dots \times b_n) \times (j_2 - 1) + \dots \\ & + b_n \times (j_{n-1} - 1) + (j_n - 1)] \times l \end{aligned}$$

# 数组的顺序表示和实现

- 以“列优先顺序”存储

第1列中的每个元素对应的(首)地址是：

$$\text{LOC}[a_{j1}] = \text{LOC}[a_{11}] + (j - 1) \times l \quad j = 1, 2, \dots, m$$

第2列中的每个元素对应的(首)地址是：

$$\text{LOC}[a_{j2}] = \text{LOC}[a_{11}] + m \times l + (j-1) \times l \quad j = 1, 2, \dots, m$$

... ..

第n列中的每个元素对应的(首)地址是：

$$\text{LOC}[a_{jn}] = \text{LOC}[a_{11}] + (n - 1) \times m \times l + (j - 1) \times l \quad j = 1, 2, \dots, m$$

由此可知，二维数组中任一元素 $a_{ij}$ 的(首)地址是：

$$\begin{aligned} \text{LOC}[a_{ij}] &= \text{LOC}[a_{11}] + [(i - 1) \times m + (j - 1)] \times l \\ i &= 1, 2, \dots, n \quad j = 1, 2, \dots, m \end{aligned}$$

# 数组的顺序表示和实现

---

- 假设二维数组A的规模为6X8，每个元素用相邻的6个字节存储，存储器按字节编址。已知A的起始存储位置（基地址）为1000，计算：

(1) 数组A的体积（即存储量）

$$6 \times 8 \times 6 = 288$$

(2) 数组A的最后一个元素 $a_{68}$ 的第一个字节的地址

$$1000 + 288 - 6 = 1282$$

(3) 按行存储时，元素 $a_{25}$ 的第一个字节的地址

$$1000 + (1 \times 8 + 4) \times 6 = 1072$$

(4) 按列存储时，元素 $a_{58}$ 的第一个字节的地址

$$1000 + (7 \times 6 + 4) \times 6 = 1276$$

# 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，在高级语言编制程序时，简单而又自然的方法，就是将一个矩阵描述为一个二维数组。
- 矩阵在这种存储表示之下，可以对其元素进行随机存取，各种矩阵运算也非常简单，并且存储的密度为1。

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \quad \text{一个 } m \times n \text{ 的矩阵。}$$

# 矩阵的压缩存储

---

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。

# 矩阵的压缩存储

---

- 特殊矩阵
- 所谓特殊矩阵就是指非零元素或零元素的分布具有一定规律的矩阵。
- 常见的这类矩阵有：对称矩阵、下（上）三角矩阵、对角线矩阵等等。

# 矩阵的压缩存储--对称矩阵

- 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，如下图所示。

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

对称矩阵示例



# 矩阵的压缩存储--对称矩阵

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

不失一般性，假设按“行优先顺序”存储下三角形(包括对角线)中的元素。

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

K	1	2	3	4	...	n(n-1)/2		...	n(n+1)/2		
sa	a <sub>11</sub>	a <sub>21</sub>	a <sub>22</sub>	a <sub>31</sub>	a <sub>32</sub>	a <sub>33</sub>	...	a <sub>n1</sub>	a <sub>n2</sub>	...	a <sub>nn</sub>

对称矩阵的压缩存储示例

# 矩阵的压缩存储--对称矩阵

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：

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

## 矩阵的压缩存储--对称矩阵

---

根据上述的下标对应关系，对于矩阵中的任意元素 $a_{ij}$ ，均可在一维数组sa中唯一确定其位置k；反之，对所有：

$$k=1, 2, \dots, n(n+1)/2,$$

都能确定sa[k]中的元素在矩阵中的位置(i,j)。

称sa[0...n(n+1)/2]为n阶对称矩阵A的压缩存储。

# 矩阵的压缩存储--三角矩阵

- 以主对角线划分，三角矩阵有上三角和下三角两种。

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

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

(a) 上三角矩阵示例

$$\begin{pmatrix} a_{11} & c & \dots & c \\ a_{21} & a_{22} & \dots & c \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

(b) 下三角矩阵示例

三角矩阵示例

# 矩阵的压缩存储--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。
- 下三角矩阵元素  $a_{ij}$  保存在向量  $sa$  中时的下标值  $k$  与  $(i,j)$  之间的对应关系是：

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

- 上三角矩阵元素  $a_{ij}$  保存在向量  $sa$  中时的下标值  $k$  与  $(i,j)$  之间的对应关系是：
- $$K = \begin{cases} i \times (i-1) / 2 + j - 1 & \text{当 } i \leq j \text{ 时} \\ n \times (n+1) / 2 & \text{当 } i > j \text{ 时} \end{cases} \quad 1 \leq i, j \leq n$$

## 矩阵的压缩存储--对角矩阵

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

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

三对角矩阵示例

# 矩阵的压缩存储--对角矩阵

- 如上图三对角矩阵，非零元素仅出现在主对角( $a_{ii}, 1 \leq i \leq n$ )上、主对角线上的那条对角线( $a_{i, i+1}, 1 \leq i \leq n-1$ )、主对角线下的那条对角线上( $a_{i+1, i}, 1 \leq i \leq n-1$ )。显然，当 $|i-j| > 1$ 时，元素 $a_{ij} = 0$ 。
- 由此可知，一个 $k$ 对角矩阵( $k$ 为奇数) $A$ 是满足下述条件：当 $|i-j| > (k-1)/2$ 时， $a_{ij} = 0$

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

三对角矩阵示例

# 矩阵的压缩存储--对角矩阵

- 对角矩阵可按行优先顺序或对角线顺序，将其压缩存储到一个向量中，并且也能找到每个非零元素和向量下标的对应关系。
- 仍然以三对角矩阵为例讨论。

当 $i=1, j=1, 2$ ，或 $i=n, j=n-1, n$  或

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

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

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

三对角矩阵的压缩存储示例



# 矩阵的压缩存储--对角矩阵

- 如前图所示三对角矩阵的压缩存储形式。数组sa中的元素sa[k]与三对角矩阵中的元素 $a_{ij}$ 存在一一对应关系，在 $a_{ij}$ 之前有 $i-1$ 行，共有 $3 \times (i-1) - 1$ 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：

$$\begin{aligned}\text{LOC}[a_{ij}] &= \text{LOC}[a_{11}] + [3 \times (i-1) - 1 + (j - i + 1)] \times l \\ &= \text{LOC}[a_{11}] + (2 \times i + j - 3) \times l\end{aligned}$$

- 上例中， $a_{34}$ 对应着sa[7]， $k = 2 \times i + j - 3 = 2 \times 3 + 4 - 3 = 7$ ，称sa[0...3×n-2]是n阶三对角矩阵A的压缩存储。
- 上述各种特殊矩阵，其非零元素的分布都是有规律的，因此总能找到一种方法将它们压缩存储到一个向量中，并且一般都能找到矩阵中的元素与该向量的对应关系，通过这个关系，仍能对矩阵的元素进行随机存取。

# 稀疏矩阵

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

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

稀疏矩阵示例

# 稀疏矩阵

- 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。

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

稀疏矩阵示例



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

# 稀疏矩阵

---

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

三元组结点定义

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

三元组顺序表定义

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

# 稀疏矩阵

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

稀疏矩阵示例



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

↑   ↑   ↑  
row col value

原矩阵的三元组表

# 稀疏矩阵

---

- 矩阵的运算包括矩阵的转置、矩阵求逆、矩阵的加减、矩阵的乘除等。在此，先讨论在这种压缩存储结构下的求矩阵的转置的运算。
- 一个 $m \times n$ 的矩阵A，它的转置B是一个 $n \times m$ 的矩阵，且 $b[i][j] = a[j][i]$ ， $0 \leq i \leq n$ ， $0 \leq j \leq m$ ，即B的行是A的列，B的列是A的行。
- 设稀疏矩阵A是按行优先顺序压缩存储在三元组表a.data中，若仅仅是简单地交换a.data中i和j的内容，得到三元组表b.data，b.data将是一个按列优先顺序存储的稀疏矩阵B，要得到按行优先顺序存储的b.data，就必须重新排列三元组表b.data中元素的顺序。

# 稀疏矩阵的转置

---

求转置矩阵的基本算法思想是：

- ① 将矩阵的行、列下标值交换。即将三元组表中的行、列位置值 $i$ 、 $j$ 相互交换；
- ② 重排三元组表中元素的顺序。即交换后仍然是按行优先顺序排序的。

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

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

# 稀疏矩阵的转置

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

(a) 原矩阵的三元组表

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

(b) 转置矩阵的三元组表

稀疏矩阵及其转置矩阵的三元组顺序表



# 稀疏矩阵的转置

- 转置矩阵的算法如下：

```
void TransMatrix(TMatrix a, TMatrix b)
```

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

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

# 稀疏矩阵的转置

---

一般传统矩阵的转置算法为：

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

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

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

# 稀疏矩阵的快速转置

---

## • 快速转置的算法

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

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

附设两个辅助向量num[ ]和cpot[ ]。

- ◆ num[col]：统计A中第col列中非0元素的个数
- ◆ cpot[col]：指示A中相应列第一个非0元素在b.data中的位置

# 稀疏矩阵的快速转置

- 显然有位置对应关系：

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

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

稀疏矩阵示例

num[col]和cpot[col]的值表

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

# 十字链表

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

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

row	col	value
down		right

(a) 结点结构

rn	cn	tn
down		right

(b) 头结点结构

十字链表结点结构

# 十字链表

---

- 由定义知，稀疏矩阵中同一行的非0元素的由right指针域链接成一个行链表，由down指针域链接成一个列链表。则每个非0元素既是某个行链表中的一个结点，同时又是某个列链表中的一个结点，所有的非0元素构成一个十字交叉的链表，称为十字链表。
- 此外，还可用两个一维数组分别存储行链表的头指针和列链表的头指针。

# 十字链表

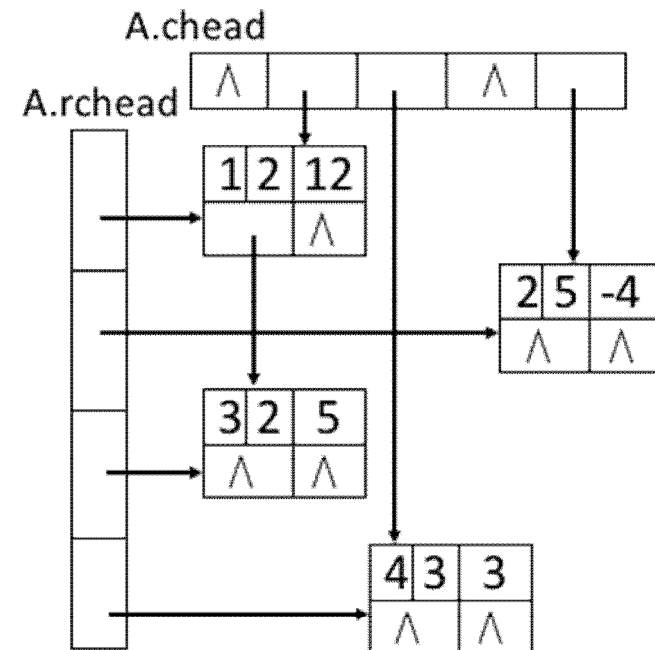
十字链表结点的描述如下：

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

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

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

(a) 稀疏矩阵



(b) 稀疏矩阵的十字交叉链表

# 广义表

---

- 广义表是线性表的推广和扩充，在人工智能领域中应用十分广泛。
- 在前面的内容中，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(Atom)。所谓原子项可以是一个数或一个结构，是指结构上不可再分的。若放松对元素的这种限制，容许它们具有其自身结构，就产生了广义表的概念。
- 广义表(Lists，又称为列表)：是由 $n(n \geq 0)$ 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。



# 广义表的定义

---

- 线性表的推广，表中的元素可以是表。

- $LS = ( a_1, a_2, \dots, a_n )$

- $A = ()$

- $B = ( e )$

- $C = ( a, ( b, c, d ) )$

- $D = ( A, B, C )$

- $E = ( a, E )$

习惯上：原子用**小写字母**，  
子表用**大写字母**。

# 广义表的定义

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。

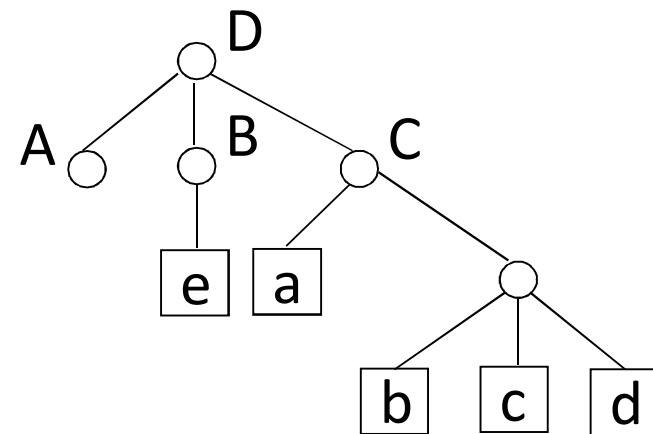
广义表的深度：广义表中括号的重数。

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

# 广义表的定义

## 广义表及其示例

广 义 表	表长n	表深h
$A=()$	0	0
$B=(e)$	1	1
$C=(a,(b,c,d))$	2	2
$D=(A,B,C)$	3	3



广义表的图形表示

# 广义表的定义

---

## 广义表的重要结论：

- 广义表的元素可以是原子，也可以是子表，子表的元素又可以是子表，…。即广义表是一个多层次的结构。
- 广义表可以被其它广义表所共享，也可以共享其它广义表。广义表共享其它广义表时通过表名引用。
- 广义表本身可以是一个递归表。
- 根据对表头、表尾的定义，任何一个非空广义表的表头可以是原子，也可以是子表，而表尾必定是广义表。

# 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
---------	------

(a) 原子结点

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

(b) 表结点

广义表的链表结点结构示意图

# 广义表的存储结构

---

- 相应的数据结构定义如下：

```
typedef struct GLNode
```

```
{ int tag; /* 标志域，为1：表结点;为0：原子结点 */
```

```
    union
```

```
        { elemtype value; /* 原子结点的值域 */
```

```
          struct {struct GLNode *hp, *tp; } ptr;
```

```
    } Gdata;
```

```
} GLNode; /* 广义表结点类型 */
```

# 广义表的存储结构

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。

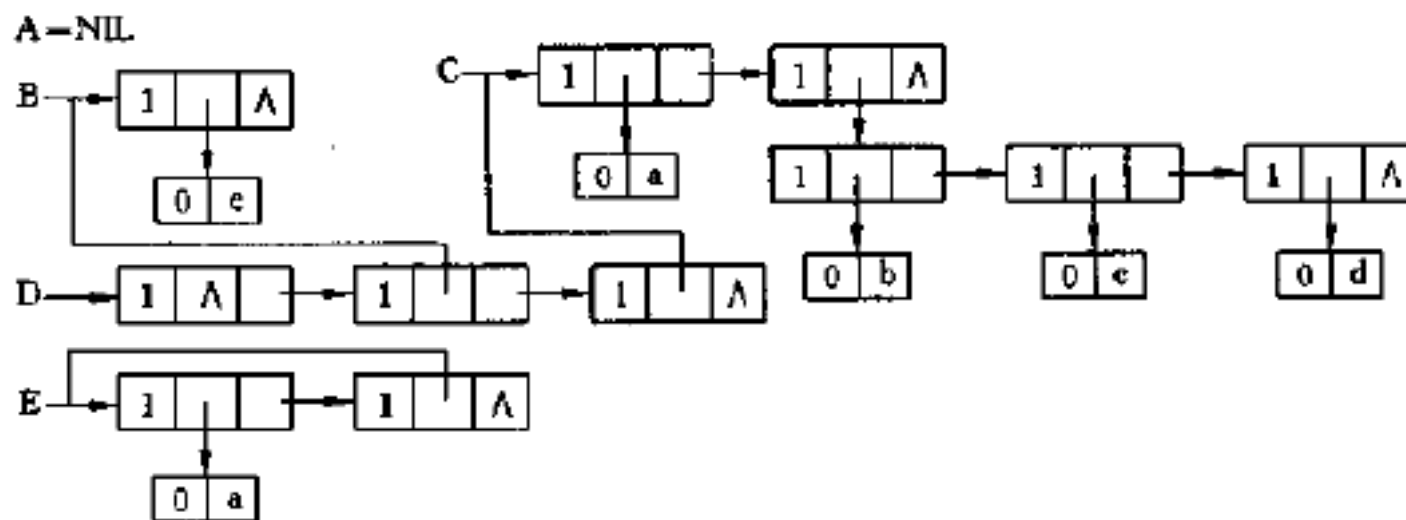


图 5:9 广义表的存储结构示例

# 广义表的存储结构

---

- 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；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。



---

# 谢谢！

