# 第五章　数组和广义表

- 5.1 数组的定义
- 5.2 数组的顺序表示和实现
- 5.3 矩阵的压缩存储
    - 5.3.1 特殊矩阵
    - 5.3.2 稀疏矩阵

5.4 广义表的定义

5.5 广义表的存储结构

数组和广义表可看成是一种特殊的线性表，其特殊在于，表中的数组元素本身也是一种线性表。

## 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。

# 一维数组

- 定义

  相同类型的数据元素的集合。
- 一维数组的示例

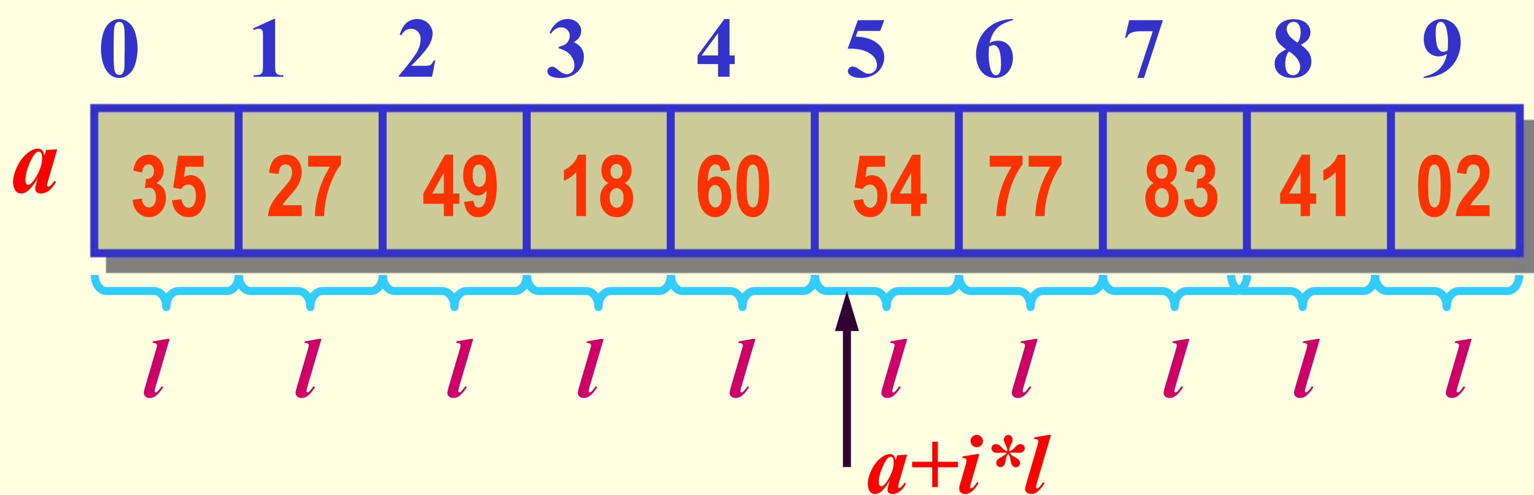
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 35 | 27 | 49 | 18 | 60 | 54 | 77 | 83 | 41 | 02 |

# 数组的定义和初始化

```
main ( ) {
    int a1[3] = { 3, 5, 7 }, *elem;
    for ( int i = 0; i < 3; i++ )
        printf ( “%d”, a1[i], “\n” );    //静态数组
    elem = malloc(..);
    for ( int i = 0; i < 3; i++ ) {
        printf ( “%d”, *elem, “\n” );  //动态数组
        elem++;
    }
}
```

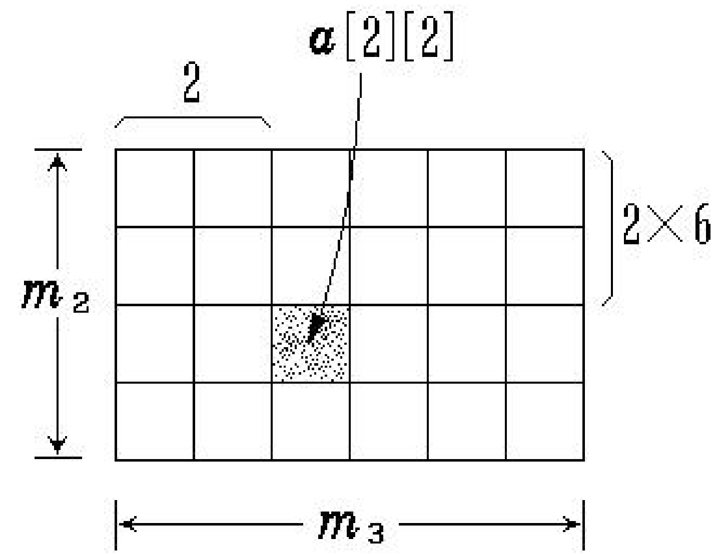
- 一维数组存储方式

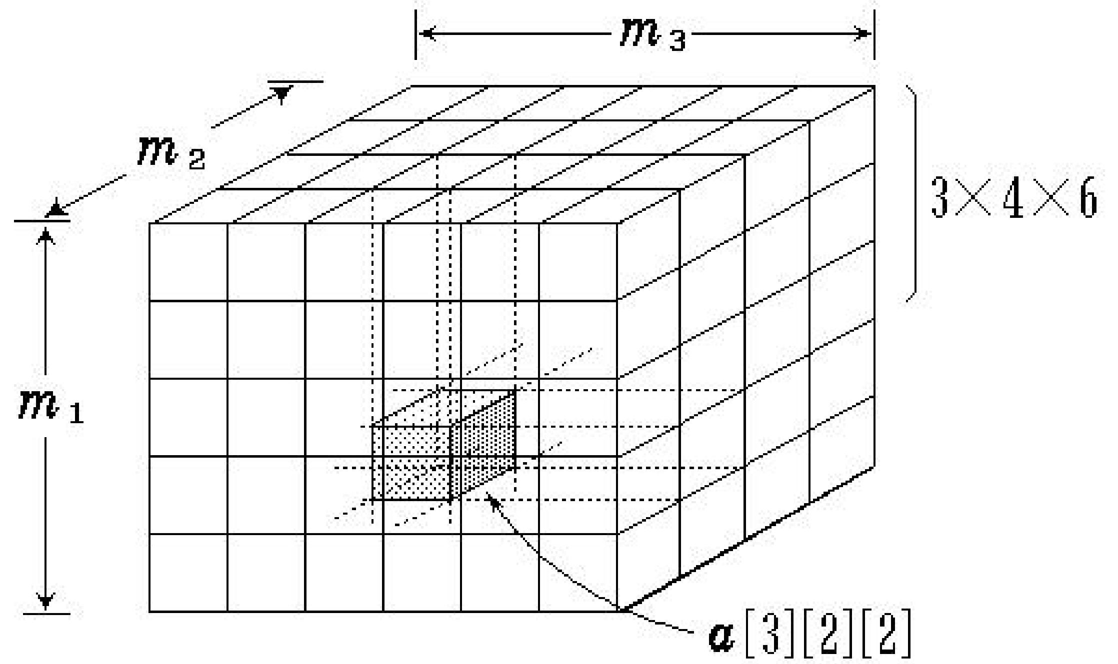
$$\mathrm{LOC}(i) = \begin{cases} a, & i = 0 \\ \mathrm{LOC}(i-1)+l = a+i*l, & i > 0 \end{cases}$$




$$\mathrm{LOC}(i) = \mathrm{LOC}(i-1)+l = a+i*l$$

# 二维数组　　三维数组




- 行向量 下标 i　　页向量 下标 i
- 列向量 下标 j　　行向量 下标 j
- 　　列向量 下标 k

- **ADT Array {**
  **数据对象：**D＝{$a_{j1,j2,\ldots,ji,\ldots,jn}$ | $j_i$=0,..., $b_i$-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，
  $a_{j1,\ldots,jn}$∈ElemSet }

- D={$a_{i,j}$ |0≤i≤m-1,0≤j≤n-1, $a_{i,j}$ ∈ ElemType}

  R={ROW,COL}

  其中：

  ROW={< $a_{i-1,j}$ , $a_{i,j}$ >| i=1,…,m-2, 0≤j≤n-1,

  $a_{i-1,j}$ , $a_{i,j}$ ∈ ElemType}

  (称作"行关系")

  COL={< $a_{i,j-1}$ , $a_{i,j}$ >| j=1,…,n-2, 0≤i≤m -1,

  $a_{i,j-1}$, $a_{i,j}$ ∈ ElemType}

  (称作"列关系")

二维数组:

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

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。

在C语言中，一个二维数组类型可以定义为其分量类型为一维数组类型的一维数组类型，也就是说，

```
typedef  elemtype  array2[m][n];
```

等价于:

```
typedef elemtype array1[n];
typedef  array1 array2[m];
```

同理，一个n维数组类型可以定义为其数据元素为n-1维数组类型的一维数组类型。

数组一旦被定义，它的维数和维界就不再改变。因此，除了结构的初始化和销毁之外，数组只有存取元素和修改元素值的操作。

- **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**

## 5.2 数组的顺序表示和实现

由于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。

又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。

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:

$a_{00}, a_{01}, \ldots, a_{0(n-1)}, a_{11}, a_{12}, \ldots a_{1(n-1)}, \ldots\ldots, a_{m-1\ 0}, a_{m-1\ 1}, \ldots, a_{m-1\ n-1}$

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*n个元素按列优先顺序存储的线性序列为:

$a_{00},a_{10},\ldots,a_{m-1\ 0},a_{01},a_{11},\ldots a_{m-1\ 1},\ldots\ldots,a_{0n-1},a_{1n-1},\ldots,a_{m-1n-1}$

在FORTRAN语言中，数组就是按列优先顺序存储的。

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。

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。

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)×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)×n+j-1个元素，因此，$a_{ij}$的地址计算函数为：

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

同样，三维数组$A_{ijk}$按“行优先顺序”存储，其地址计算函数为：

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

- 题目1：二维数组中行下标从10到20，列下标从5到10，按行优先存储，每个元素占4个存储单元，A[10][5]的存储地址是1000，则元素A[15][10]的存储地址为（）。
- 题目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，当用以列为主存放时，元素A[5，8]的存储首地址为（）。

- 题目3：一个矩阵a[0][0]开始存放，每个元素占用4个存储单元，若a[7][8]的存储地址为2732，a[13][16]的存储地址为3364，则此矩阵的存储方式为__________。

A. 只能按行优先存储　B. 只能按列优先存储

C. 按行优先存储或列优先存储

D. 以上都对

## 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。但是在矩阵中非零元素呈某种规律分布或者矩阵中出现大量的零元素的情况下，看起来存储密度仍为1，但实际上占用了许多单元去存储重复的非零元素或零元素，这对高阶矩阵会造成极大的浪费，为了节省存储空间，**我们可以对这类矩阵进行压缩存储：即为多个相同的非零元素只分配一个存储空间；对零元素不分配空间。**

## 5.3.1特殊矩阵

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。

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:

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

$a_{00}$

$a_{10}$ $a_{11}$

$a_{20}$ $a_{21}$ $a_{23}$

…………………

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

在这个下三角矩阵中，第i行恰有i+1个元素，元素总数为:

$$n(n+1)/2$$

因此，我们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$a_{ij}$和sa[k]之间找一个对应关系

若$i \geq j$，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$1+2+\ldots+i=i(i+1)/2$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:

$$k=i*(i+1)/2+j \qquad (0 \leq k<n(n+1)/2)$$

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:

$$k=j*(j+1)/2+i \qquad (0 \leq k<n(n+1)/2)$$

令$I=\max(i,j)$，$J=\min(i,j)$，则k和i, j的对应关系可统一为:

$$\mathbf{k=I*(I+1)/2+J \qquad (0 \leq k<n(n+1)/2)}$$

因此，$a_{ij}$的地址可用下列式计算:

LOC(aij)=LOC(sa[k])

=LOC(sa[0])+k*d=LOC(sa[0])+[I*(I+1)/2+J]*d

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a21和a12均存储在 sa[4]中，这是因为

k=I*(I+1)/2+J=2*(2+1)/2+1=4

- 题目4：设一个10阶对称矩阵A采用压缩存储，A[0][0]为第一个元素，其存储地址为d，每个元素占1个存储单元，则元素A[8][5]的存储地址为（ ）。

## 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，如图所示。在大多数情况下，三角矩阵常数为零。

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

(a) 上三角矩阵　　　　(b) 下三角矩阵

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($0 \leqq p<n$)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：$a_{ii}, a_{ii+1}, \ldots a_{ij-1}$。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:

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

- 题目5：设有一5阶上三角矩阵A［1..5，1..5］，现将其上三角中的元素按列优先顺序存放在一堆数组B［1..15］中。已知B［1］的地址为100，每个元素占用2个存储单元，则A［3，4］的地址为（ ）

A．116 B．118 C．120 D．122

## 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，

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

非零元素仅出现在主对角($a_{ii}$, $0 \leqq i \leqq n-1$) 上，紧邻主对角线上面的那条对角线上($a_{i\,i+1}$, $0 \leqq i \leqq n-2$)和紧邻主对角线下面的那条对角线上($a_{i+1\,i}$, $0 \leqq i \leqq n-2$)。

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。

- 题目6：设有三对角矩阵$A_{n \times n}$（从$A_{1 \times 1}$开始），将其三对角线上的元素逐行存于数组B[1…m]中，使$B_k = A_{i,j}$，求

（1）用i、j表示k的下标换算公式

（2）用k表示i、j的下标表换公式

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。

## 5.3.2 稀疏矩阵

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(即s<<m× n)，则称A为稀疏矩阵。

$$\mathbf{A}_{6\times 7}=\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

精确点，设在的矩阵A中，有s个非零元素。令 $e=s/(m*n)$，称e为矩阵的稀疏因子。通常认为 $e \leqq 0.05$ 时称之为稀疏矩阵。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。

## 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。

```
#define maxsize 10000
typedef int datatype;
typedef struct{
    int   i,j;
    datatype v;
 }triple;
typedef struct{
    triple  data[maxsize];
    int  m,n,t;
    }tripletable;
```

设A为tripletable型的结构变量，左下方稀疏矩阵的三元组的表示如下：

$$\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

A.m=6 A.n=7 A.t=8

A.data

| | 行 (row) | 列 (col) | 值 (value) |
|---|---|---|---|
| [0] | 0 | 3 | 22 |
| [1] | 0 | 6 | 15 |
| [2] | 1 | 1 | 11 |
| [3] | 1 | 5 | 17 |
| [4] | 2 | 3 | −6 |
| [5] | 3 | 5 | 39 |
| [6] | 4 | 0 | 91 |
| [7] | 5 | 2 | 28 |

下面以矩阵的转置为例，说明在这种压缩存储结构上如何实现矩阵的运算。

一个m× n的矩阵A，它的转置B是一个n× m的矩阵，且a[i][j]=b[j][i]，0≤i≤m，0≤j≤n，即A的行是B的列，A的列是B的行。

将A转置为B，就是将A的三元组表a.data置换为表B的三元组表b.data，如果只是简单地交换a.data中i和j的内容，那么得到的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的顺序。

由于A的列是B的行，因此，按a.data的列序转置，所得到的转置矩阵B的三元组表b.data必定是按行优先存放的。

按这种方法设计的算法，其基本思想是：对A中的每一列col(0≤col≤n-1)，通过从头至尾扫描三元表a.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b.data中，即可得到B的按行优先的压缩存储表示。

| i | j | v |
|---|---|---|
| 0 | 3 | 22 |
| 0 | 6 | 15 |
| 1 | 1 | 11 |
| 1 | 5 | 17 |
| 2 | 3 | -6 |
| 3 | 5 | 39 |
| 4 | 0 | 91 |
| 5 | 2 | 28 |

| i | j | v |
|---|---|---|
| 4 | 0 | 91 |
| 1 | 1 | 11 |
| 5 | 2 | 28 |
| 0 | 3 | 22 |
| 2 | 3 | -6 |
| 1 | 5 | 17 |
| 3 | 5 | 39 |
| 0 | 6 | 15 |

| i | j | v |
|---|---|---|
| 0 | 4 | 91 |
| 1 | 1 | 11 |
| 2 | 5 | 28 |
| 3 | 0 | 22 |
| 3 | 2 | -6 |
| 5 | 1 | 17 |
| 5 | 3 | 39 |
| 6 | 0 | 15 |

```
Void transmatrix(tripletable a,tripletable b)
  {
    int p, q, col;
    b.m=a.n;
    b.n=a.m;
    b.t=a.t;
    if(b.t<=0)
        printf("A=0\n");//矩阵A为零矩阵
     q=0;
```

```
for(col=1;col<=a.n;col++)
     for(p=0;p<=a.t;p++)
      if(a.data[p].j==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;
          q++;
         }
 }
```

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。而一般传统矩阵的转置算法为:

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

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

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t<<m*n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：对A扫描一次，按A第二列提供的列号一次确定位置装入B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。

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B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B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[0..n]和cpot[0..n]

num[0..n]：统计M中每列非零元素的个数，num[col]的值可以由A的第二列求得。

快速转置算法如下:

```
void fasttranstri(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.n;++col)
          num[col]=0;\\数组初始化
 for(k=1;k<=a.t;++k)
         ++num[a.data[k].j];\\统计每列非零元素个数
```

```
  cpot[1]=1;
  for(col=2;col<=a.t;++col)
      cpot[col]=cpot[col-1]+num[col-1];\\统计每列第一个非零元素的起始位置
  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];\\每列下一个非零元素的位置
   }
 }
}
```

- 算法有四个并列的单循环
- 其时间复杂度为O(n+t)，当t和m*n同数量级时，其时间复杂度为O(m*n)。和经典算法相同

## 二、行逻辑链接的三元组

有时为了方便某些矩阵运算，我们在按行优先存储的三元组中，加入一个行表来记录稀疏矩阵中每行的非零元素在三元组表中的起始位置。当将行表作为三元组表的一个新增属性加以描述时，我们就得到了稀疏矩阵的另一种顺序存储结构：行逻辑链接的三元组表。其类型描述如下：

```
#define maxrow 100
 typedef struct{
      triple data[maxsize];
      int     rpos[maxrow];
      int    n,m,t;
  }rtripletable
```

下面讨论两个稀疏矩阵相乘的例子，容易看出这种表示方法的优越性。

两个矩阵相乘的经典算法也是大家所熟悉的。

若设 $Q=M*N$

其中，M是$m_1*n_1$矩阵，N是$m_2*n_2$矩阵。

当$n_1=m_2$时有:

```
for(i=1;i<=m1;++i)
    for(j=1;j<=n2;++j){
        q[i][j]=0
        for(k=1;k<=n1;++k)
           q[i][j]+=m[i][k]*n[k][j];
      }
```

此算法的复杂度为$O(m_1*n_1*n_2)$。

- 当M和N是稀疏矩阵并用三元组表作为存储结构时，传统方法就用不上了。

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

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

M.data

| i | j | e |
|---|---|---|
| 1 | 2 | 2 |
| 2 | 1 | 1 |
| 3 | 1 | -2 |
| 3 | 2 | 4 |

N.data

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

Q.data

- 乘积矩阵Q中元素

$Q(i,j) = \text{sum}_\{k = 1, \ldots, n_1\} M(i,k) \times N(k,j), 1 \le i \le m_1, 1 \le j \le n_2$

- 在经典算法中，不论M(i,k)和N(k,j)的值是否为零，都要进行一次乘法运算，而实际上，这两者有一个值为零时，乘积也为零。因此，在对稀疏矩阵进行运算时，应免去这种无效操作，换句话说，只需在M.data和N.data中找到相应的各个元素相乘即可。
- 由此均可见，为了得到非零的乘积，只要对M.data[1…M.tu]中每个元素(i, k, M(i, k))，找到N.data中所有相应的元素(k, j, N(k,j))相乘即可。为此需在N.data中寻找矩阵N中第k行的所有非零元。

- 在稀疏矩阵的行逻辑链接的顺序表中，N.rpos为我们提供了有关信息。并且，由于rpos[row]指示矩阵N的第row行中第一个非零元在N.data中的序号，则rpos[row + 1] – 1指示矩阵N的第row行中最后一个非零元在N.data中的序号。而最后一行中最后一个非零元在N.data中的位置显然就是N.tu了。
- 总而言之，有了rpos这个数组，我们可以很方便的找到需要的行的非零元。

- 稀疏矩阵相乘的基本操作是：对于M中每一个元素M.data[p](p = 1, 2,…,M.tu)，找到N中所有满足条件M.data[p].j = N.data[q].i的元素N.data[q]，求得M.data[p].v 和 N.data[q].v的乘积。
- 而从前面的公式得知，乘积矩阵Q中每一个元素的值是个累计和，这个乘积M.data[p].v ×N.data[q].v只是Q[i][j]中的一小部分。
- 因此，为了便于操作，应对每个元素设置一个累计器变量，其初值为零，然后扫描数组M，求得相应元素的乘积并累加到适当的求累计和的变量上。

- 两个稀疏矩阵相乘的乘积不一定是稀疏矩阵。反之，即使前面公式中每个分量值M(i,k)×N(k,j)不为零，其累加值Q[i][j]也可能为零。因此乘积矩阵Q中的元素是否为非零元，只有在求得其累加和才能得知。
- 由于Q中元素的行号和M中元素的行号一致，又M中元素排列是以M的行序为主序的，由此可对Q进行逐行处理，先求得累计求和的中间结果（Q的一行），然后在压缩存储到Q.data中去。

- 由此，两个稀疏矩阵相乘（Q = M ×N）的过程可以大致描述如下：

```
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
```

```
Status MulSMatrix(RLSMatrix M, RLSMatrix N, RLSMatrix &Q){
    If(M.nu != N.mu) return ERROR;
    Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0;  //Q初始化
    if(M.tu * N.tu != 0){
        for(arow = 1; arow <= M.mu; ++arow){  //处理M的每一行
            ctemp[] = 0;                //当前行元素累加器清零
            Q.rpos[arow] = Q.tu + 1;
            if(arow < M.mu) tp = M.rpos[arow + 1];
            else tp = M.tu + 1;
            for(p = M.rpos[arow]; p < tp; ++p){
                //对当前行中每一个非零元
            }//求得Q中第crow( = arow)行的非零元
            for(ccol = 1; ccol <= Q.nu; ++ccol){
                //压缩存储该行非零单元
            }
        }
    }
return OK;
```

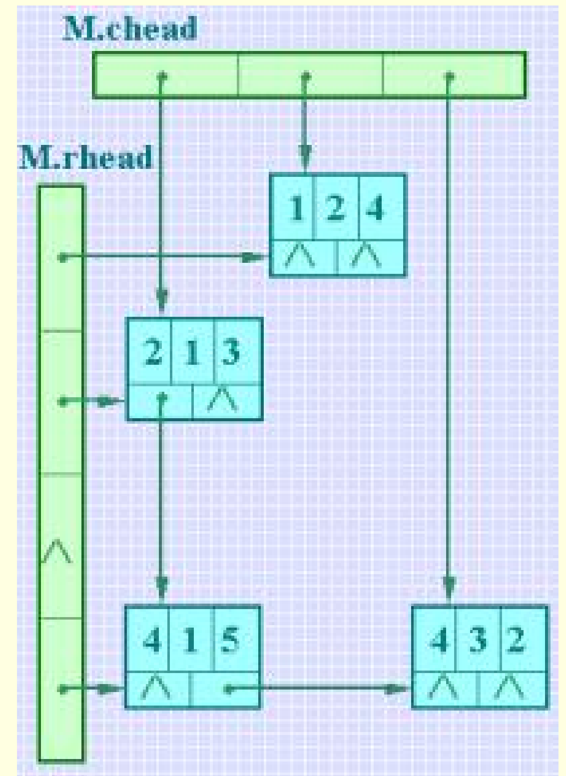
```
for(p = M.rpos[arow]; p < tp; ++p){  //对当前行中每一个非零元
        brow = M.data[p].j;
        if(brow < N.mu) t = N.rpos[brow + 1];
        else t = N.tu + 1;
        for( q = N.rpos[brow]; q < t; ++q){ //乘积元素在Q中序号
                ccol = N.data[q].j;
                ctemp[ccol] += M.data[p].e * N.data[q].e;
        }//for q
}//求得Q中第crow( = arow)行的非零元
 for (ccol = 1; ccol <= Q.nu; ++ccol){  //压缩存储该行非零单元
        if(ctemp[ccol]){
                if（++ Q.tu > MAXSIZE) return ERROR;
                Q.data[Q.tu] = (arow, ccol, ctemp[ccol])
        }//if
}//for
```

- 分析上述算法的时间复杂度：
  - 累加器ctemp初始化的时间复杂度为$O(M.mu \times N.nu)$
  - 求Q的所有非零元的时间复杂度为$O(M.tu \times N.tu/N.mu)$ //N.tu/N.mu大概是N矩阵平均每列非零元素的个数
  - 进行压缩存储的时间复杂度为$O(M.mu \times N.nu)$
  - 因此，总的时间复杂度为$O(M.mu \times N.nu + M.tu \times N.tu/N.mu)$
  - 若M是m行n列稀疏矩阵，N是n行p列的稀疏矩阵，则$M.tu = \rho_m \times m \times n$, $N.tu = \rho_n \times n \times p$。算法的时间复杂度为$O(m \times p \times (1 + n\rho_m\rho_n))$。
  - 当$\rho_m<0.05$和$\rho_n<0.05$和$n<10000$时，算法的时间复杂度为$O(m \times p)$。

# 三、十字链表

在某些运算中，如两个矩阵相加，矩阵的非零元个数和位置变化较大。需要用更灵活的方式表示矩阵

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



# 5.4 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。

- 广义表的概念 n ( ≥ 0 )个表元素组成的有
- 限序列，记作
- LS = (a0, a1, a2, …, an-1)
- LS是表名，ai是表元素，它可以是表 (称
- 为子表)，可以是数据元素(称为原子)。
- n为表的长度。n = 0 的广义表为空表。
- n > 0时，表的第一个表元素称为广义表
- 的表头(head)，除此之外，其它表元素组
- 成的表称为广义表的表尾(tail)。

# 例如

- A=( );　　//A是一个空表
- B=(e);　　//表B有一个原子
- C=(a,(b,c,d)); //两个元素为原子a和子表(b,c,d)
- D=(A,B,C);　　//有三个元素均为列表
- E=(a,E); //递归的列表

- 从上述定义可知，广义表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，因此空表的深度为1，因为一个"单原子"不是广义表，所以没有"深度"可言，但可以认为它的深度为0。

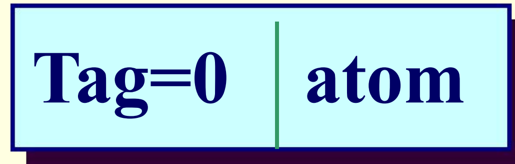
 4. 广义表可被其它广义表所共享。例如上述例子中广义表B也可以是广义表 D 中的一个子表。

 5. 广义表可以是一个"自递归"的表。例如上述例子中的广义表 E，自递归的广义表的长度是个有限值，而深度为无限值。

# 广义表存储结构

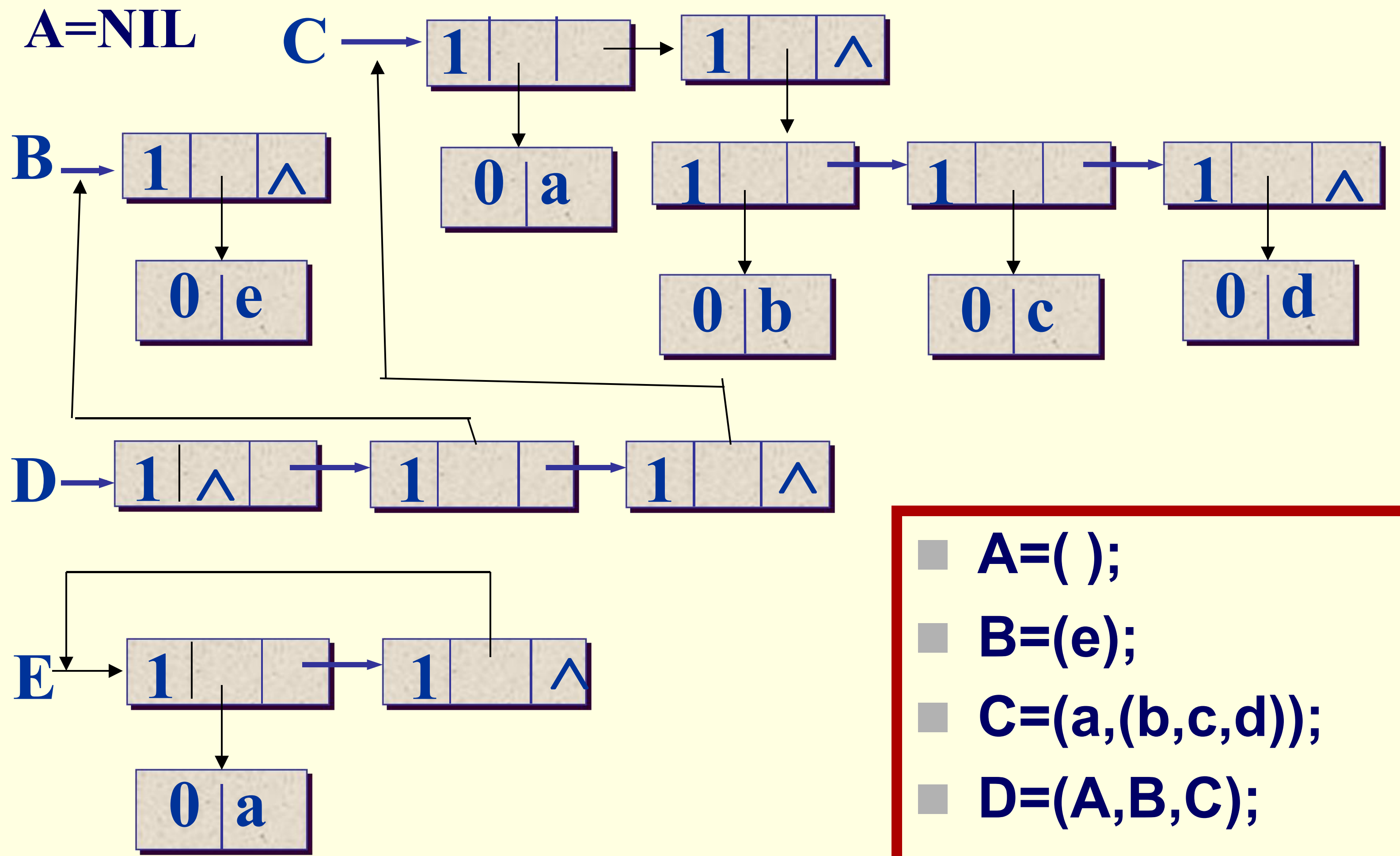
表结点

原子结点




```
typedef struct GLNode{
        int tag;
        union{
                DataType  atom;
                struct {structGLNode   *hp,*tp;} ptr;
        };
}
```

# 方法一




- A=( );
- B=(e);
- C=(a,(b,c,d));
- D=(A,B,C);
- E=(a,E);

# 方法二

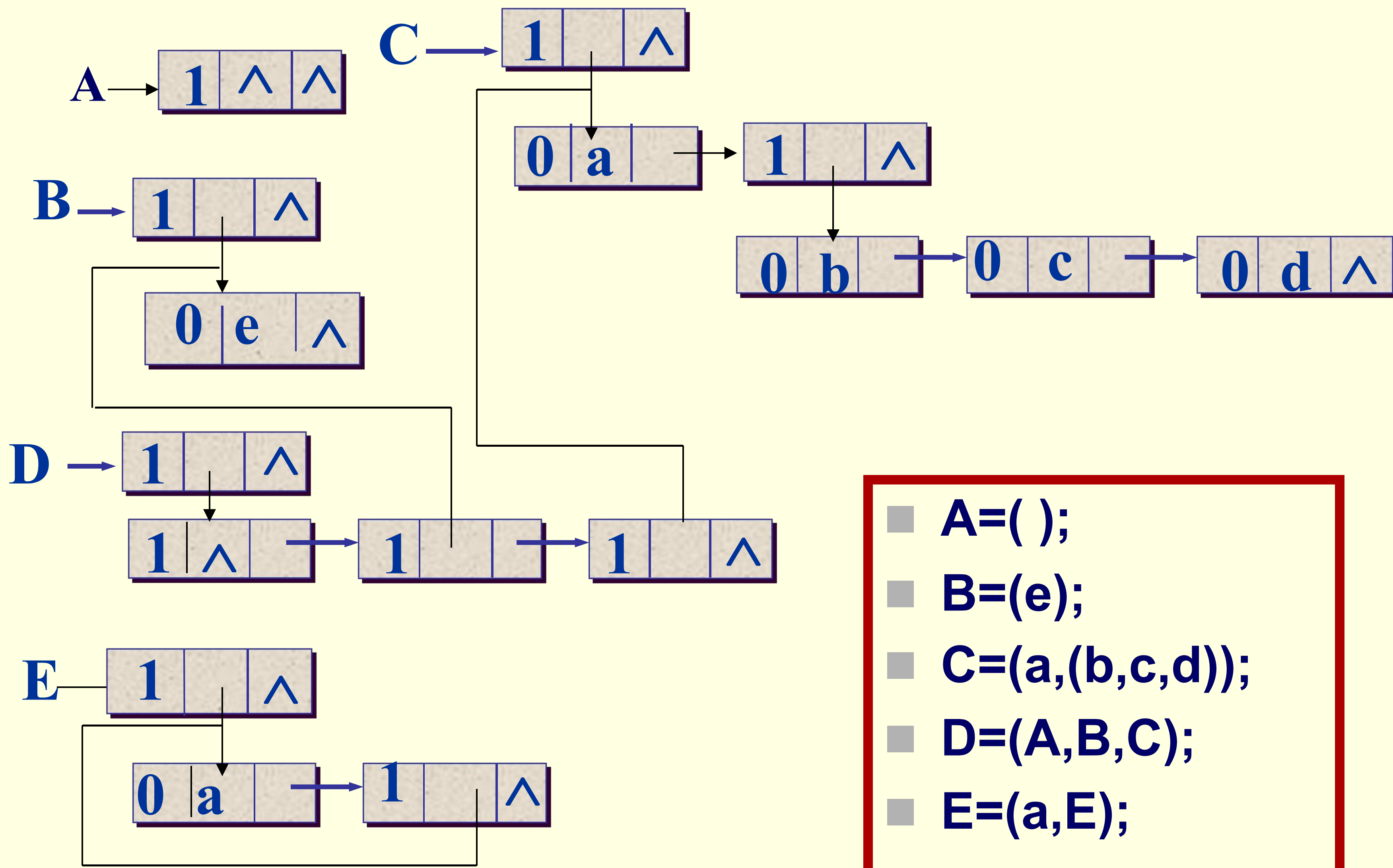
节点1

| Tag=1 | hp | tp |
| --- | --- | --- |

节点2

| Tag=0 | atom | tp |
| --- | --- | --- |

```
typedef struct GLNode{
        int tag;
        union{
                DataType  atom;
                struct GLNode   *hp;
        };
        struct GLNode *tp;
}
```


A
1 ^ ^
B
1 ^
0 e ^
C
1 ^
0 a
1 ^
0 b
0 c
0 d ^
D
1 ^
1 ^
1
1 ^
E
1 ^
0 a
1 ^
A=( );
B=(e);
C=(a,(b,c,d));
D=(A,B,C);
E=(a,E);

# 作业

- 假设有二维数组 A6×8，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）;
  (2) 数组 A 的最后一个元素 a57 的第一个字节的地址;
  (3) 按行存储时，元素 a14 的第一个字节的地址;
  (4) 按列存储时，元素 a47 的第一个字节的地址。