第五章

数组和广义表

数据结构中的数据元素—原子类型　　int

元素的值不会再分

— 结构类型　　struct

数组 (多维) 、广义表 ---(线性表的扩展) 。

表中数据元素本身也是一种数据结构

5.1 数组的类型定义

ADT Array {

数据对象:

$D = \{a_{j^1,j^2,\ldots,j^i,j^n} \mid j_i = 0,\ldots,b_i - 1,\ i=1,2,..,n\}$

数据关系:

$R = \{R1, R2, \ldots, Rn\}$

$Ri = \{<a_{j^1,\ldots j^i,\ldots j^n}, a_{j^1,\ldots j^i+1,\ldots j^n}> \mid 0 \le j_k \le b_k - 1,$

$1 \le k \le n$ 且 $k \ne i,\ 0 \le j_i \le b_i - 2,\ i=2,\ldots,n\}$

基本操作:

} ADT Array

5.1 数组的定义

二维数组是一个定长线性表，且它的每个数据元素也是一个定长线性表。

$$A_{m\times n} = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{m-1} \end{bmatrix}$$

$$= \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

$$A_{m\times n} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

$$= \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{m-1} \end{bmatrix}$$

$$\alpha_i = [\, a_{i0}\,,\ a_{i1}\,,\ a_{i2}\,, \ldots,\ a_{i,n-1}\,]$$

5.2 数组的顺序表示和实现

类型特点：

1) 数组一旦被定义，其维数和维界就不再改变，通常数组一般不作插入或删除操作，只是存取或修改元素，故通常采用顺序存储结构。

2) 数组是多维的结构，而存储空间是一个一维的结构。

如何将多维数组结构转换对应一组连续的存储单元？

有两种顺序映象的方式：

1) 以行序为主序（**低下标优先**）；

2) 以列序为主序（**高下标优先**）；

二维数组通常可以描述为两种形式 ：

以行序为主序： PASCAL 、 C

可以看成 $A=(\alpha_0, \alpha_1, \cdots, \alpha_{m-1})$

其中α_i 是一个行向量形式的线性表， $0\le i\le m-1$

$$\alpha_i=(a_{i0}, a_{i1}, \cdots, a_{i\,n-1})$$

$$A_{m\times n}=\begin{bmatrix} [a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1}] \\ [a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1}] \\ \vdots & \vdots & \vdots & & \vdots \\ [a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1}] \end{bmatrix}$$

以列序为主序： FORTRAN

可以看成 $A=(\alpha_0, \alpha_1, \cdots, \alpha_{n-1})$

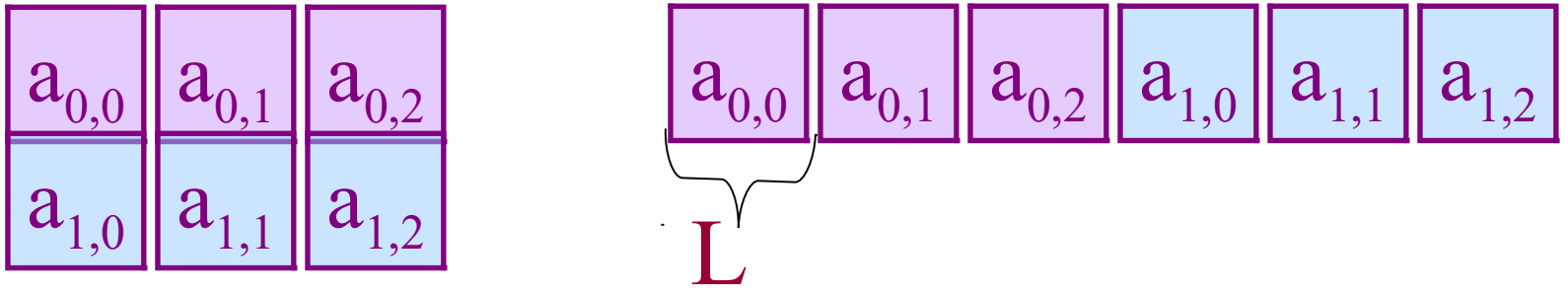
其中 α_j 是一个列向量形式的线性表， $0\le j\le n-1$

$$\alpha_j=(a_{0j}, a_{1j}, \cdots, a_{m-1j})$$

$$A_{m\times n}=\left[\begin{bmatrix}a_{00}\\a_{10}\\\vdots\\a_{m-1,0}\end{bmatrix}\begin{bmatrix}a_{01}\\a_{11}\\\vdots\\a_{m-1,1}\end{bmatrix}\begin{bmatrix}a_{02}\\a_{12}\\\vdots\\a_{m-1,2}\end{bmatrix}\cdots\begin{bmatrix}a_{0,n-1}\\a_{1,n-1}\\\vdots\\a_{m-1,n-1}\end{bmatrix}\right]$$

以“行序为主序”的存储映象

例如：



二维数组 A 中任一元素 $a_{i,j}$ 的存储位置

$$LOC(i,j) = \underline{LOC(0,0)} + (b_2 \times i + j) \times L$$

称为**基地址**或基址。

推广到一般情况，可得到 n 维数组数据元素存储位置的映象关系

$$Loc(j_1, j_2, \ldots j_n) = Loc(0,0,\ldots 0) + (b_2 \times \ldots \times b_n \times j_1 + b_3 \times \ldots \times b_n \times j_2 + \ldots + b_n \times j_{n-1} + j_n)L$$

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

其中 $c_n = L$， $c_{i-1} = b_i \times c_i$， $1 < i \le n$ 。

称为 n 维数组的映象函数。**数组元素的存储位置是其下标的线性函数**

5.3 矩阵的压缩存储

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

矩阵元素如何与存储空间对应？

通常利用二维数组来存储矩阵元素。 $b_{m\times n}$

$$a_{ij} = b_{i-1,j-1}$$

数值分析中存在某些特殊 (阶数很高) 矩阵，在矩阵中有许多值相同的元素或者零元素。

$$\begin{bmatrix} a_{11} & 0 & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ a_{m1} & 0 & \cdots & 0 \end{bmatrix}$$

用数组存储造成浪费。

为了节省存储空间，需要对这类矩阵进行压缩存储。

压缩存储是指为多个值相同的元素只分配一个存储空间；对零元素不分配空间。

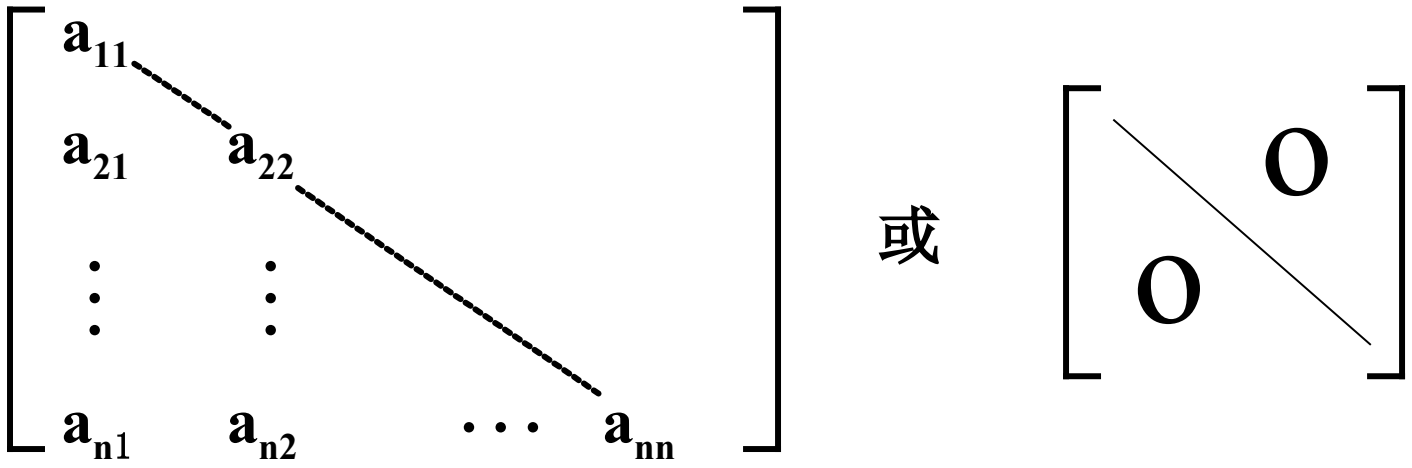
按照值相同的元素或零元素在矩阵中分布的规律：

- 对称矩阵
- 三角矩阵
- 对角矩阵
- 稀疏矩阵

n 阶对称矩阵

n 阶矩阵 A 满足： $a_{ij} = a_{ji}$

通常表示为：

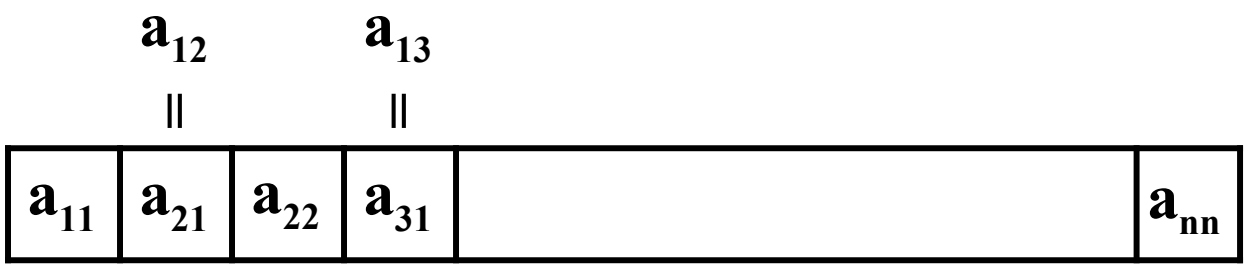


n^2 个矩阵元只需占用 $n(n+1)/2$ 存储空间

设用一维数组 sa[n(n+1)/2] 模拟存储 n 阶对称矩阵 A。

关键问题： 如何建立数组元素 sa[k] 和矩阵元素 a_{ij} 之间的一一对应关系。

$$k = \begin{cases} \dfrac{i(i-1)}{2} + j - 1 & \text{当} \quad i \geq j \\ \dfrac{j(j-1)}{2} + i - 1 & \text{当} \quad i < j \end{cases}$$

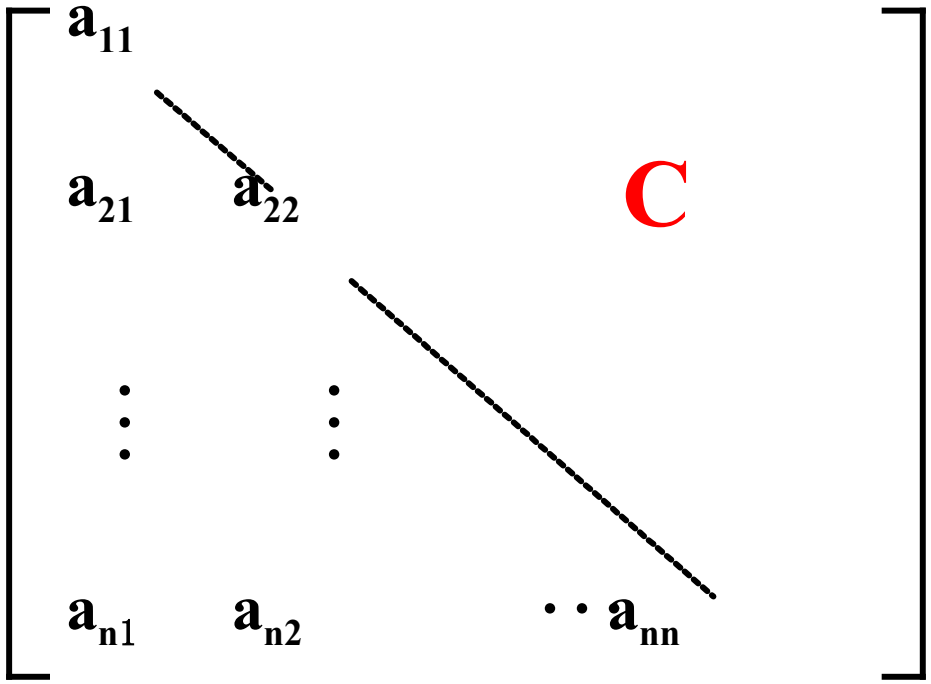


k = 0 1 2 3 $\dfrac{n(n+1)}{2} - 1$

a_{12} a_{13}

三角矩阵

所谓下 (上) 三角矩阵是指矩阵的上 (下) 三角 (不包括对角线) 中的元素均为常数 c 的 n 阶矩阵。



下三角矩阵

和对称矩阵基本一样，只是除存储其下 (上) 三角中的元之外，再加一个存储单元存放 c 。

关键问题： 如何建立数组元 sa[k] 和矩阵元 a_{ij} 之间的一一对应关系。

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

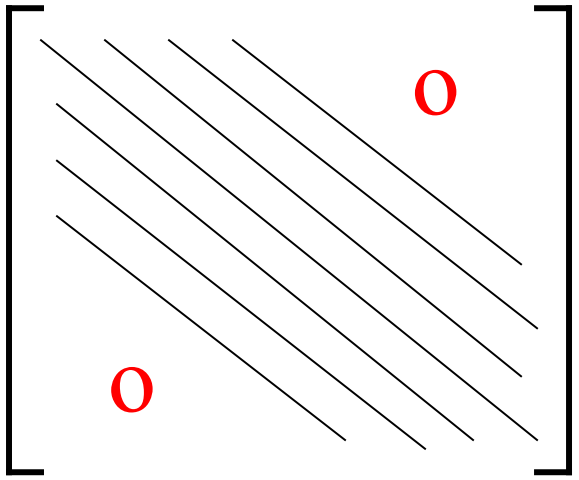
$a_{12}, a_{13} \cdots$

$\frac{n(n+1)}{2}$

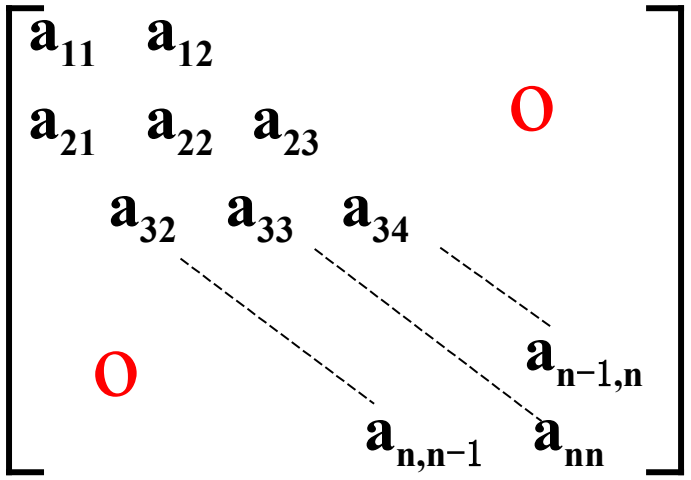
k = 0 1 2 3 $\frac{n(n+1)}{2} - 1$

对角矩阵

所有的非零元都集中在以主对角线为中心的带状区域中。



一般情况



三对角矩阵

关键问题： 如何建立数组元素 sa[k] 和矩阵元素 a_{ij} 之间的一一对应关系。

因对角数不同而不同。例三对角矩阵：

$$k = 2(i-1) + j - 1$$

$$(|i-j| \le 1)$$

a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	
k = 0	1	2	3	4	5	

稀疏矩阵

非零元很少的矩阵，且分布没有一定规律。

稀疏因子：设 $m \times n$ 的矩阵，有 t 个非零元，令

$$\delta = \frac{t}{m \times n}$$

，称 δ 为矩阵的稀疏因子。

通常认为 $\delta \leq 0.05$ 时称为稀疏矩阵。

以常规方法，即以二维数组表示

高阶的稀疏矩阵时产生的**问题：**

1) 零值元素占了很大空间；

2) 计算中进行了很多和零值的运算，

遇除法，还需判别除数是否为零；

解决问题的原则：

1) 尽可能少存或不存零值元素；

2) 尽可能减少没有实际意义的运算；

3) 操作方便；**即**：

能尽可能快地找到与下标值 (i, j) 对应的元素；

能尽可能快地找到**同一行**或**同一列**的非零值元素；

随机稀疏矩阵的压缩存储方法：

一、三元组顺序表

二、行逻辑联接的顺序表

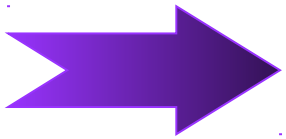
三、 十字链表

一、三元组顺序表【P97】

```
#define  MAXSIZE  12500
typedef struct {
    int  i, j;      // 该非零元的行下标和列下标
    ElemType  e;    //  该非零元的值
} Triple;                  //  三元组类型
 typedef union {
      Triple  data[MAXSIZE + 1];
      int     mu, nu, tu;
 } TSMatrix;        //  稀疏矩阵类型
```

如何求转置矩阵?

$$\begin{bmatrix} 0 & 14 & 0 & 0 & -5 \\ 0 & -7 & 0 & 0 & 0 \\ 36 & 0 & 0 & 28 & 0 \end{bmatrix}$$



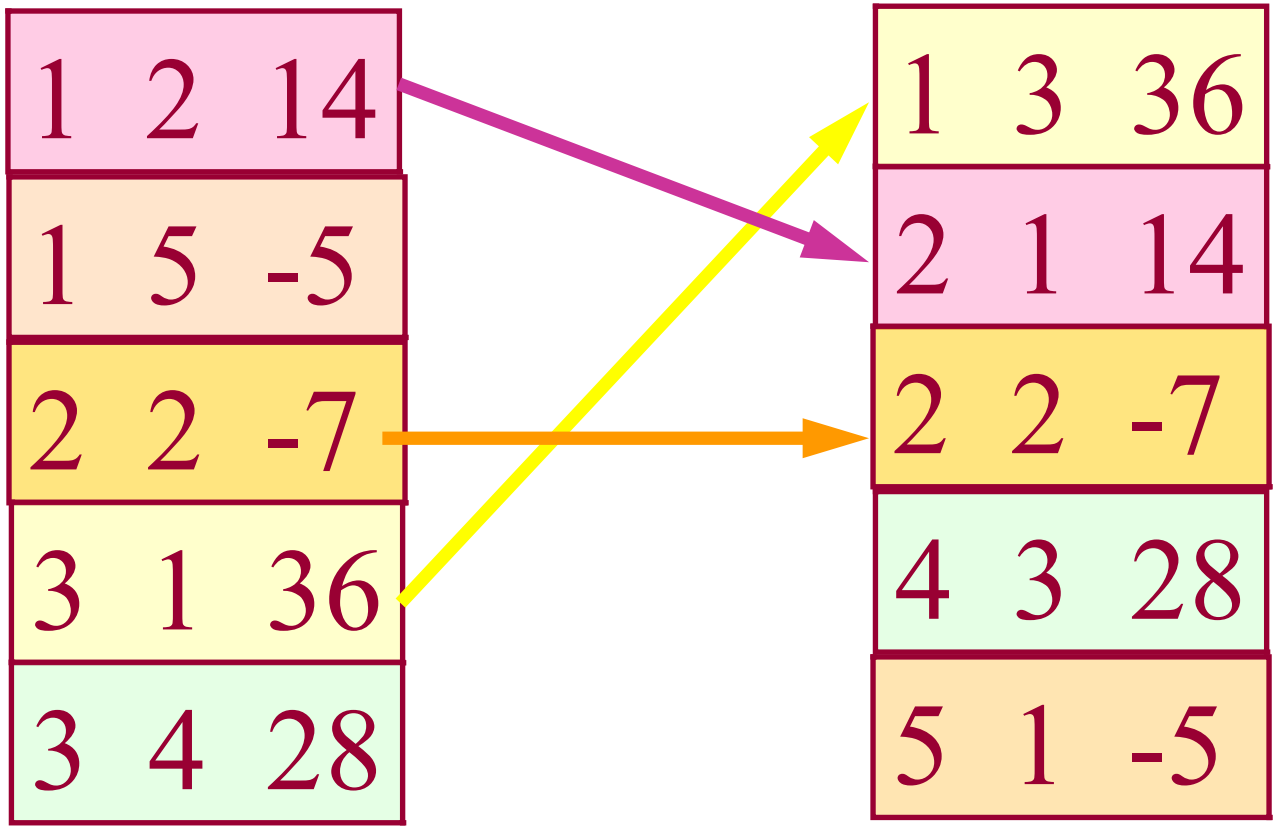
$$\begin{bmatrix} 0 & 0 & 36 \\ 14 & -7 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 28 \\ -5 & 0 & 0 \end{bmatrix}$$

用常规的二维数组表示时的算法

```
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(mu×nu)**

用“三元组”表示时如何实现?



❖稀疏矩阵转置算法思想（1）

- 设矩阵列数为 Cols，对矩阵三元组表扫描 Cols 次。第 k 次检测列号为 k 的项。
- 第 k 次扫描找寻所有列号为 k 的项，将其行号变列号、列号变行号，顺次存于转置矩阵三元组表。

【P99】

虽节省了空间，但时间复杂度高

其最坏时间复杂度为：$O(mu \times nu^2)$

❖稀疏矩阵转置算法思想（2）

- 按照矩阵 **M** 三元组的次序进行转置，并将转置后三元组放到矩阵 **T** 恰当的位置。
- 预先确定矩阵 **M** 中每一列第一个非零元的存放位置，那么转置操作时，所有的非零元都可以放到正确的位置上去。

确定转置矩阵中每一行的第一个非零元在三元组中的位置。

1	2	15
1	5	-5
2	2	-7
3	1	36
3	4	28

col	1	2	3	4	5
Num[pos]	1	2	0	1	1
Cpot[col]	**1**	**2**	**4**	**4**	**5**

附设 num 和 cpot 两个向量

```
for ( t=1; t<=M.tu; ++t )
        ++num[M.data[t].j];
cpot[1] = 1;
for (col=2; col<=M.nu; ++col)
    cpot[col] = cpot[col-1] + num[col-1];
```

```
Status FastTransposeSMatrix(TSMatrix M, TSMatrix &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];
    cpot[1] = 1;
    for (col=2; col<=M.nu; ++col)
      cpot[col] = cpot[col-1] + num[col-1];
    for (p=1; p<=M.tu; ++p) {    转置矩阵元素}
  }                                  // if
  return OK;
}                                    // FastTransposeSMatrix
```

```
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]
```

分析算法 FastTransposeSMatrix 的时间复杂度:

```
for (col=1; col<=M.nu; ++col) … …
for (t=1; t<=M.tu; ++t) … …
for (col=2; col<=M.nu; ++col) … …
for (p=1; p<=M.tu; ++p) … …
```

时间复杂度为： O(M.nu+M.tu)

二、行逻辑链接的顺序表

三元组顺序表又称**有序的双下标法**，它的特点是，非零元在表中按行序有序存储，因此**便于进行依行顺序处理的矩阵运算**。然而，若需随机存取某一行中的非零元，则需从头开始进行查找。

修改前述的稀疏矩阵的结构定义，增加一个数据成员 rpos，其值在稀疏矩阵的初始化函数中确定。

```
#define  MAXSIZE  12500
#define   MAXRC     500
typedef struct {
    Triple  data[MAXSIZE + 1];
    int     rpos[MAXRC + 1];  // 各行第一个非零元的位置表
    int     mu, nu, tu;
} RLSMatrix;   // 行逻辑链接顺序表类型
```

例如：给定一组下标，求矩阵的元素值

```
ElemType value(RLSMatrix M, int r, int c) {
  p = M.rpos[r];
  while (M.data[p].i==r &&M.data[p].j < c)
      p++;
  if (M.data[p].i==r && M.data[p].j==c)
     return M.data[p].e;
  else return 0;
} // value
```

矩阵乘法的经典算法：

```
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];
  }
```

其时间复杂度为：O(m1×n2×n1)

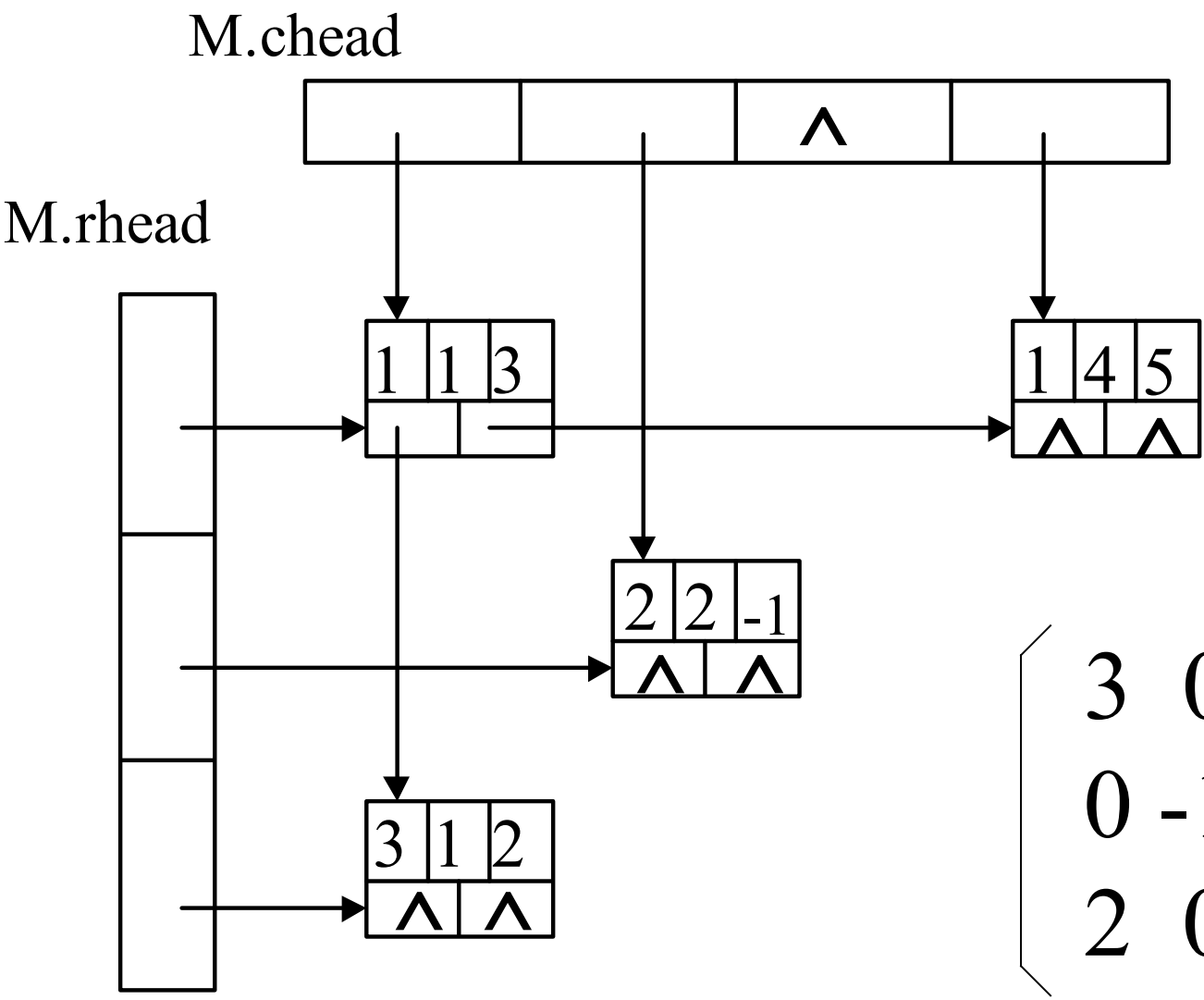
个稀疏矩阵相乘（Q=M×N）
的过程可大致描述如下：

```
初始化；
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 [nu]
        计算 Q 中第 arow 行的积并存入 ctemp[]
      将 ctemp[]  中非零元压缩存储到 Q.data ;
                    // for arow
                    // if
```

```
Status MultSMatrix
   (RLSMatrix M, RLSMatrix N, RLSMatrix &Q) {
 if (M.nu != N.mu) return ERROR;
 Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0;
 if (M.tu*N.tu != 0) {   // Q 是非零矩阵
    for (arow=1; arow<=M.mu; ++arow) {
                                  //  处理 M 的每一行
    }                              // for arow
 }                                 // if
 return OK;
}                                  // MultSMatrix
```

```
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[arow+1];++p) {
                                //对当前行中每一个非零元
处理 brow=M.data[p].j;
 if (brow < N.mu )  t = N.rpos[brow+1];
M   else  { t = N.tu+1 }
    for (q=N.rpos[brow];  q< t;  ++q) {
的    ccol = N.data[q].j;          //  乘积元素在 Q 中列号
每    ctemp[ccol] += M.data[p].e * N.data[q].e;
一   }                                    // for q
行 }                  //  求得 Q 中第 crow( =arow) 行的非零元
   for (ccol=1; ccol<=Q.nu; ++ccol) if (ctemp[ccol]) {
    if (++Q.tu > MAXSIZE) return ERROR;
    Q.data[Q.tu] = {arow, ccol, ctemp[ccol]};
   }                                    // if
```

三、 十字链表



$$\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

5.4 广义表

5.4 广义表的类型定义

5.5 广义表的表示方法

5.6 广义表操作的递归函数

广义表是**递归**定义的**线性结构**，

$$LS = (\alpha_1, \alpha_2, \cdots, \alpha_n)$$

其中：α_i 或为**单个元素** 或为**广义表**

例如： $A = ( )$

$F = (d, (e))$

$D = ((a,(b,c)), F)$

$C = (A, D, F)$

$B = (a, B) = (a, (a, (a, \cdots ,)))$

广义表是一个多层次的线性结构

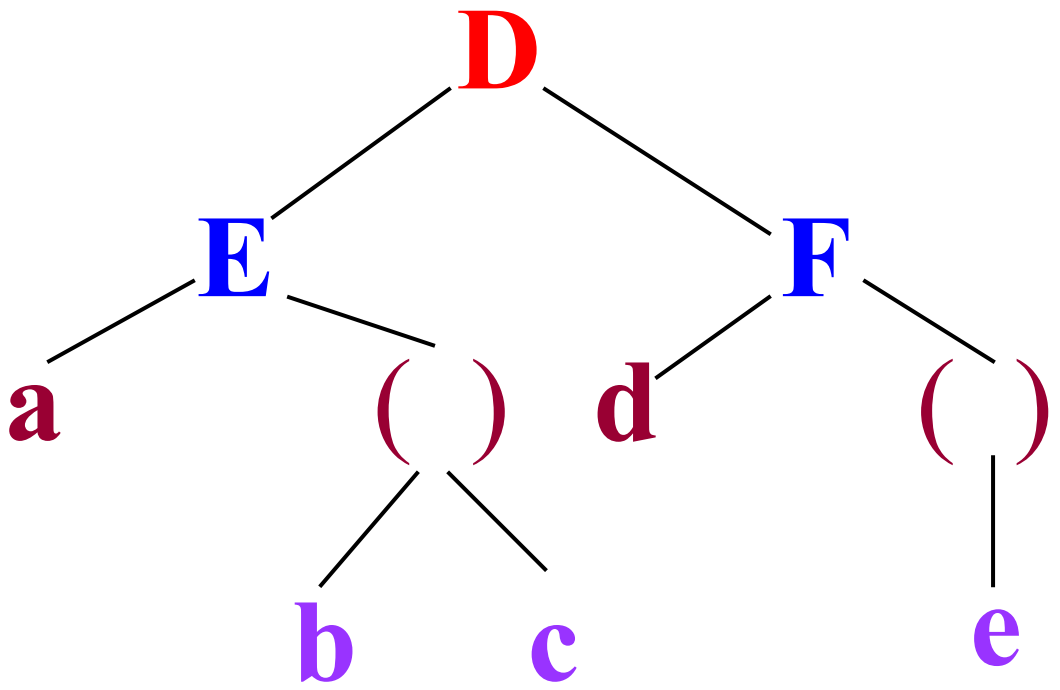
例如：

D=(E, F)

其中：

E=(a, (b, c))

F=(d, (e))



广义表 $LS = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ 的结构特点

1) 广义表中的数据元素有相对**次序**；

2) 广义表的**长度**定义为最外层包含元素个数

3) 广义表的**深度**定义为所含括弧的重数；

注意：“原子”的深度为 0；

“空表”的深度为 1。

4) 广义表可以**共享**；

) 广义表可以是一个**递归**的表；

递归表的深度是无穷值，长度是有限值

) 任何一个非空广义表 $LS = (\alpha 1, \alpha 2, \ldots, \alpha n)$

均可分解为

表头 $Head(LS) = \alpha 1$ 和

表尾 $Tail(LS) = (\alpha 2, \ldots, \alpha n)$ 两部分

例如： $D = (E, F) = ((a, (b, c)), F)$

$Head(\mathbf{D}) = E$ $Tail(\mathbf{D}) = (F)$

$Head(\mathbf{E}) = a$ $Tail(\mathbf{E}) = ((b, c))$

$Head(\mathbf{((b, c))}) = (b, c)$ $Tail(\mathbf{((b, c))}) = ()$

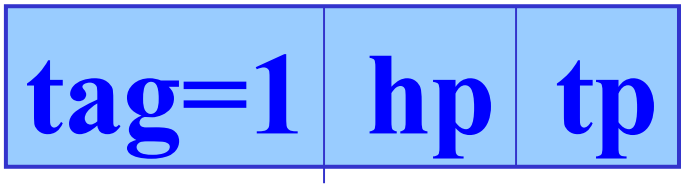
$Head(\mathbf{(b, c)}) = b$ $Tail(\mathbf{(b, c)}) = (c)$

$Head(\mathbf{(c)}) = c$ $Tail(\mathbf{(c)}) = ()$

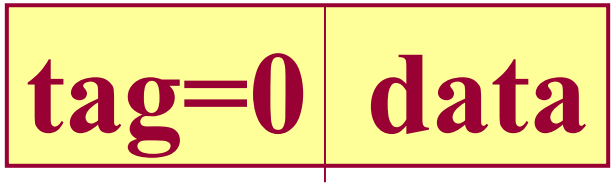
5.5 广义表的表示方法

通常采用头、尾指针的链表结构

表结点：



原子结点:

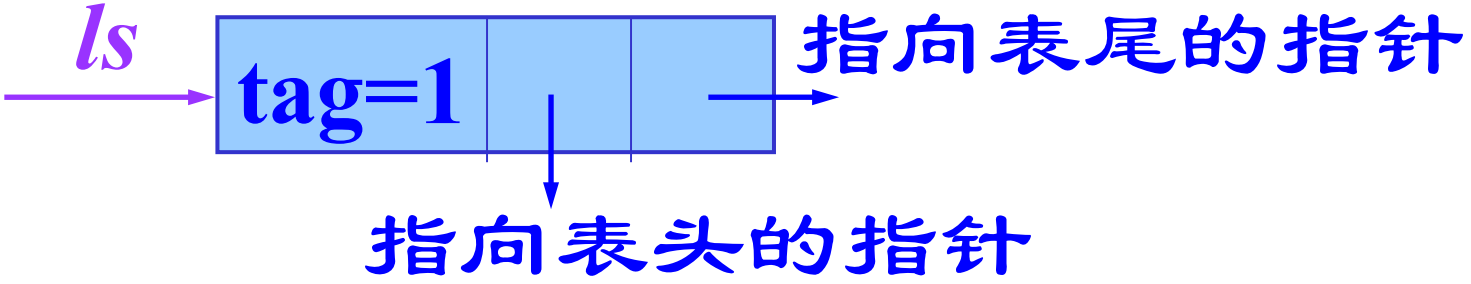


构造存储结构的两种分析方法：

1) 表头、表尾分析法：

空表　　$ls = NIL$

非空表

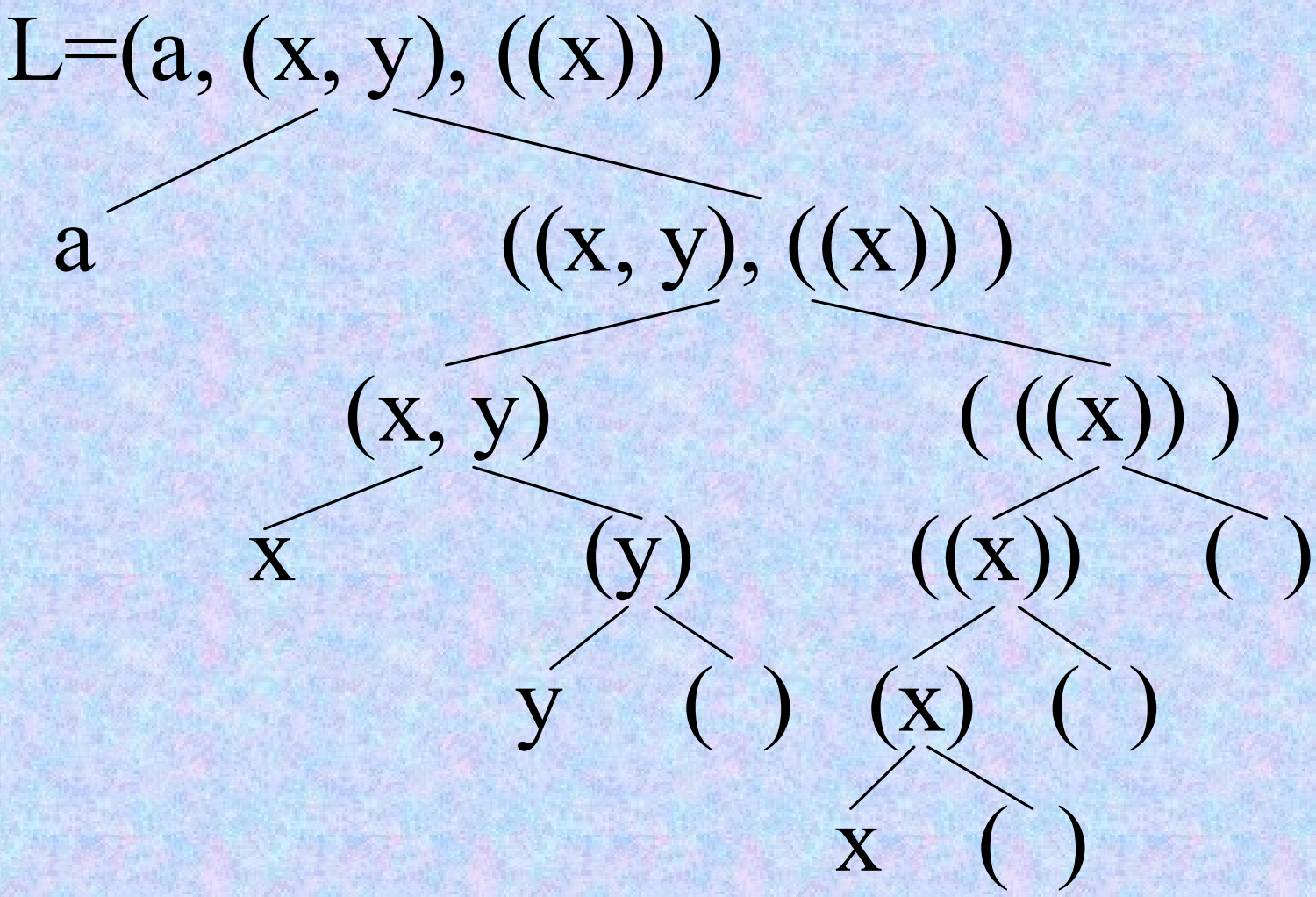


若表头为原子，则为

tag=0	data

否则，依次类推。

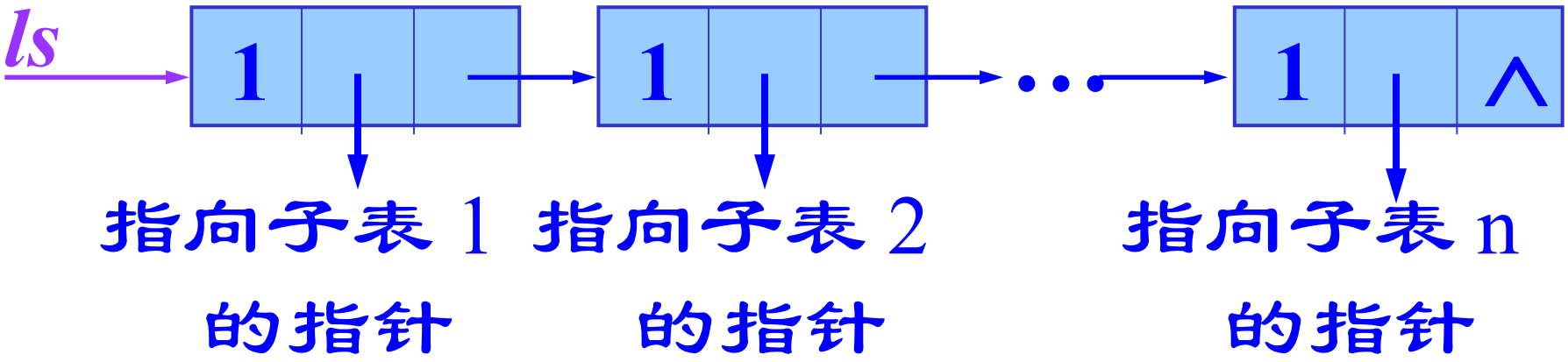
例如：



2) 子表分析法：

空表　　*ls=NIL*

非空表



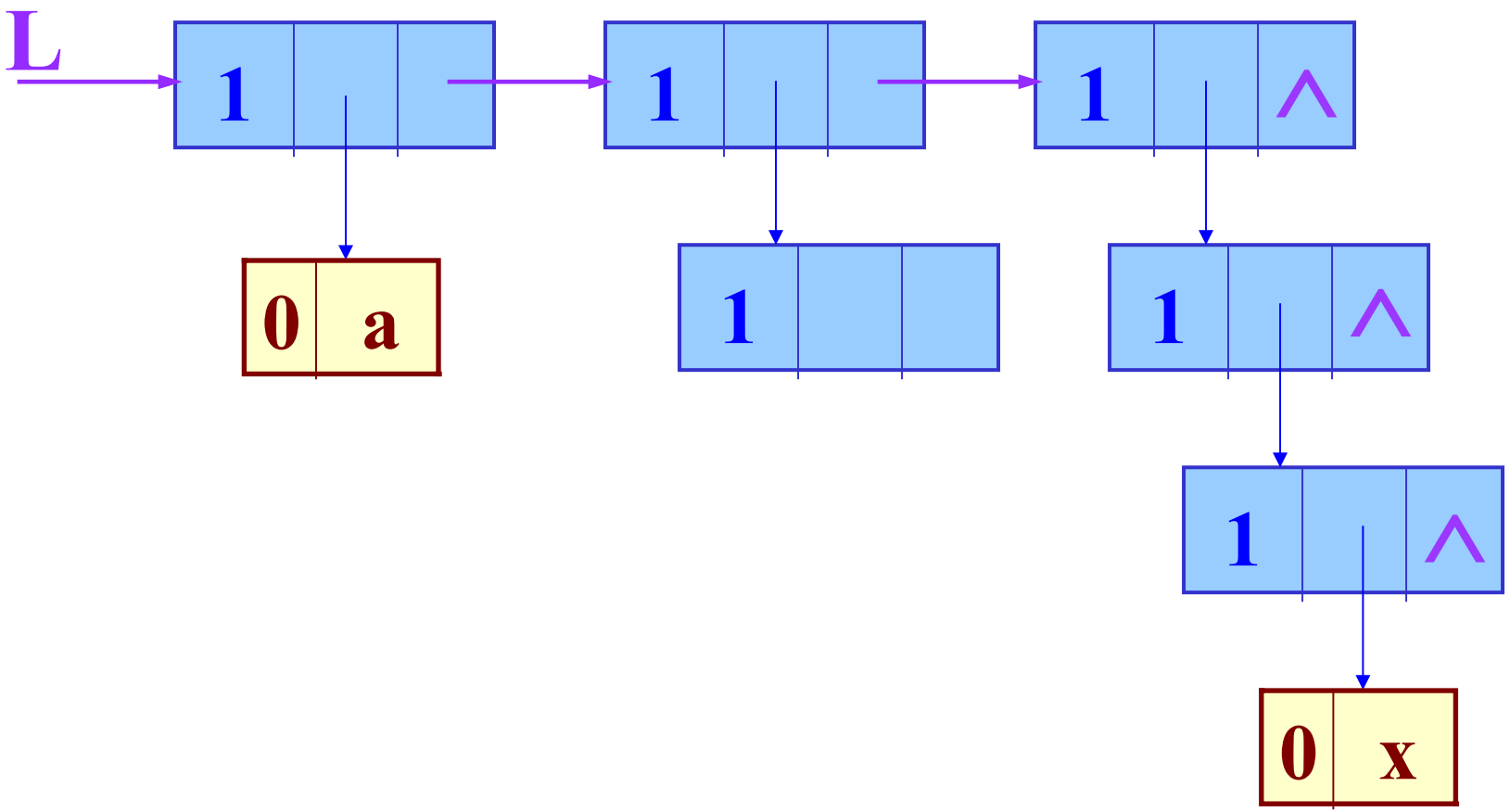
若子表为原子，则为

tag=0	data

否则，依次类推。

例如：

$L = (a, (x, y), ((x)))$



– 设有一个 n×n 的对称矩阵 A，将其上三角部分按行存放在一个一维数组 B 中，A[0][0] 存放于 B[0] 中，那么第 i 行的对角元素 A[i][i] 存放于 B 中 (　) 处。

A．(i+3)*i ／ 2

B．(i+1)*i ／ 2

C．(2n-i+1)*i ／ 2

D．(2n-i-1)*i ／ 2

1 . 已知广义表 LS ＝ ((a,b,c),(d,e,f)), 运用 head 和 tail 函数取出 LS 中原子 e 的运算是 (　　)。

A. head(tail(LS))　　B. tail(head(LS))

C. head(tail(head(tail(LS)))

D. head(tail(tail(head(LS))))

2. 广义表运算式 Tail(((a,b),(c,d))) 的操作结果是（　　）。

A. (c,d)　　B. c,d　　C. ((c,d))　　D. d

3. 广义表 L=（a，（b，c））， 进行 Tail （L）操作后的结果为（　　）。

A. c　　B. b，c　　C.（b，c）　　D.（（b，c））

4. 广义表（（a,b,c,d））的表头是（　　），表尾是（　　）。

A. a　　B.（）　　C.（a,b,c,d）　　D.（b,c,d）

5. 广义表（ a,(b,c),d,e ）的表头为（　　）。

A. a　　B. a,(b,c)　　C. (a,(b,c))　　D. (a)

6. 设广义表 L=（（ a,b,c ））， 则 L 的长度和深度分别为（ ）。

A. 1 和 1　　B. 1 和 3　　C. 1 和 2　　D. 2 和 3

7. 下面说法不正确的是 (　) 。

A. 广义表的表头总是一个广义表

B. 广义表的表尾总是一个广义表

C. 广义表难以用顺序存储结构

D. 广义表可以是一个多层次的结构