第6章 数组和广义表

6.1 数 组

6.2 稀疏矩阵

6.3 广义表

6.1 数　组

6.1.1 数组的基本概念

从逻辑结构上看，一维数组A是n（$n>1$）个相同类型数据元素a_1、a_2、…、a_n构成的有限序列，其逻辑表示为：

$$A=(a_1, a_2, \ldots, a_n)$$

其中，a_i（$1\le i\le n$）表示数组A的第i个元素。

一个m行n列的二维数组A可以看作是每个数据元素都是相同类型的一维数组的一维数组。

$$A=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \cdots & & & \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \Rightarrow A=[\ A_1,\ A_2,\ \ldots,\ A_m\ ]$$

$$A_1=[\ a_{1,1},\ a_{1,2},\ \ldots,\ a_{1,n}\ ]$$

$$A_2=[\ a_{1,1},\ a_{2,2},\ \ldots,\ a_{2,n}\ ]$$

$$\cdots\cdots$$

$$A_m=[\ a_{m,1},\ a_{m,2},\ \ldots,\ a_{m,n}\ ]$$

由此看出，多维数组是线性表的推广。

数组抽象数据类型＝逻辑结构＋基本运算（运算描述）

数组的基本运算如下：

❶ Value(A,index$_1$,index$_2$,…,index$_d$)：即A(index$_1$,index$_2$,…,index$_d$)=e，元素赋值。

❷ Assign(A,e,index$_1$,index$_2$,…,index$_d$)：即e= A(index$_1$,index$_2$,…,index$_d$)，取元素值。

❸ ADisp(A,b_1,b_2,…,b_d)：输出d维数组A的所有元素值。

6.1.2 数组的存储结构

将数组的所有元素存储在一块地址连续的内存单元中，这是一种顺序存储结构。

- 数组中的数据元素数目固定。
- 数组中的所有数据元素具有相同的数据类型。
- 数组中的每个数据元素都有一组唯一的下标。
- 数组是一种随机存储结构。可随机存取数组中的任意数据元素。

一维数组：一旦a_1的存储地址LOC(a_1)确定，并假设每个数据元素占用k个存储单元，则任一数据元素a_i的存储地址LOC(a_i)就可由以下公式求出：

$$LOC(a_i)=LOC(a_1)+(i-1)*k \quad (0\le i\le n)$$

共$i-1$个元素

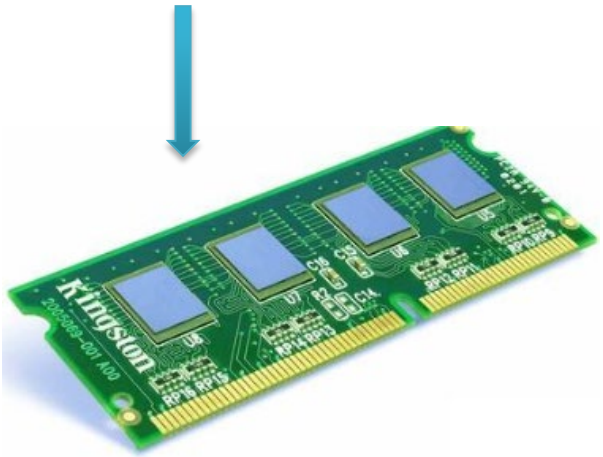
数组a：a_1 a_2 a_3 … a_{i-1} a_i … a_n

一维数组具有随机存储特性：可以在O(1)时间内找到序号为i的元素值。

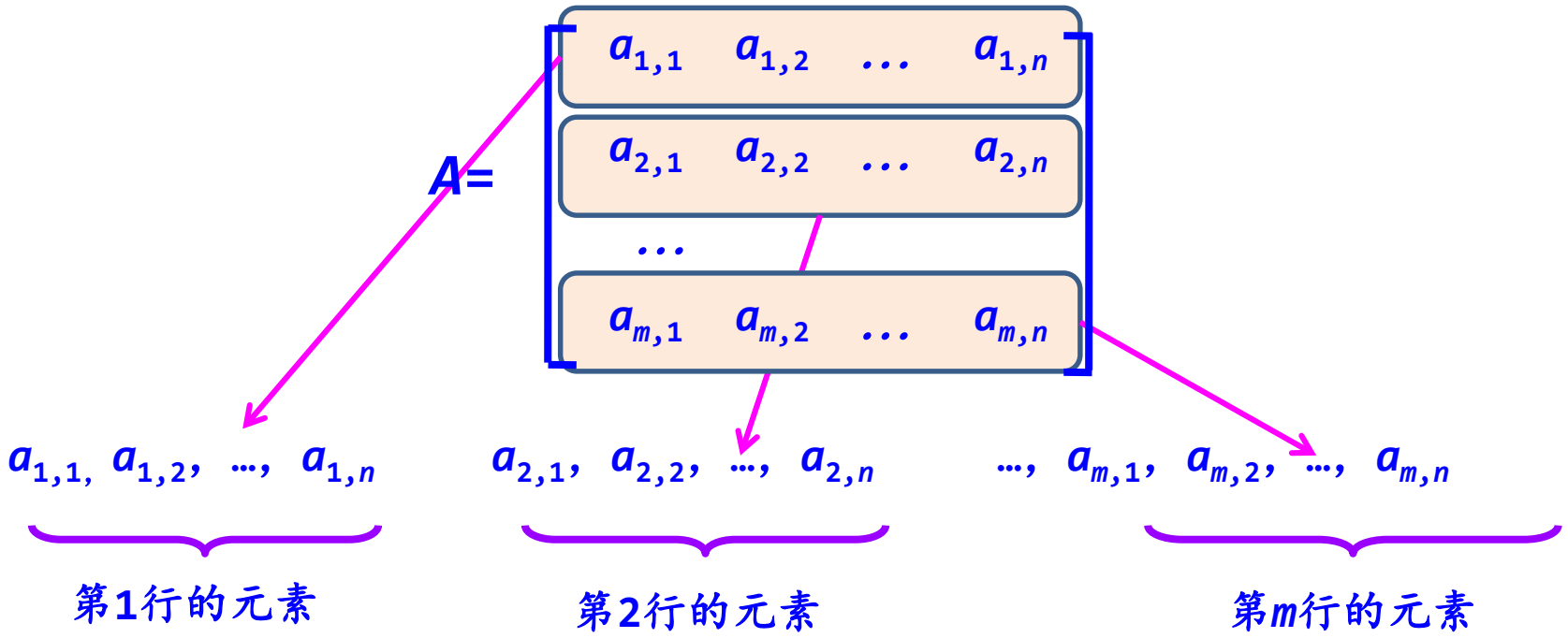
m行n列的二维数组$A_{m\times n}$，存储方式：

- 以行序为主序的存储
- 以列序为主序的存储

$$A=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \cdots & & & \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$



❶ 以行序为主序的存储方式



第1行的元素　　第i行的元素

$a_{1,1}$, $a_{1,2}$, …, $a_{1,n}$, …, $a_{i,1}$, $a_{i,2}$, …, $a_{i,j-1}$, $a_{i,j}$, …$a_{i,n}$, …

1~i-1行，每行n个元素，计$(i-1) \times n$个元素

第i行中，$a_{i,j}$元素前有j-1个元素

则$a_{i,j}$元素前共有 $(i-1)\times n+j-1$个元素

$$LOC(a_{i,j})=LOC(a_{1,1})+[(i-1)\times n+(j-1)] \times k$$

❷ 以列序为主序的存储方式

同理可推出在以列序为主序的计算机系统中有）：

$$LOC(a_{i,j})=LOC(a_{1,1})+[(j-1)\times m+(i-1)]\times k$$

二维数组采用顺序存储结构时，也具有随机存取特性。

是指给定序号i（下标），可以在O(1)的时间内找到相应的元素值。

多维数组采用顺序存储时具有随机存储特性。

6.1.3 特殊矩阵的压缩存储

特殊矩阵的主要形式有：

（1）对称矩阵

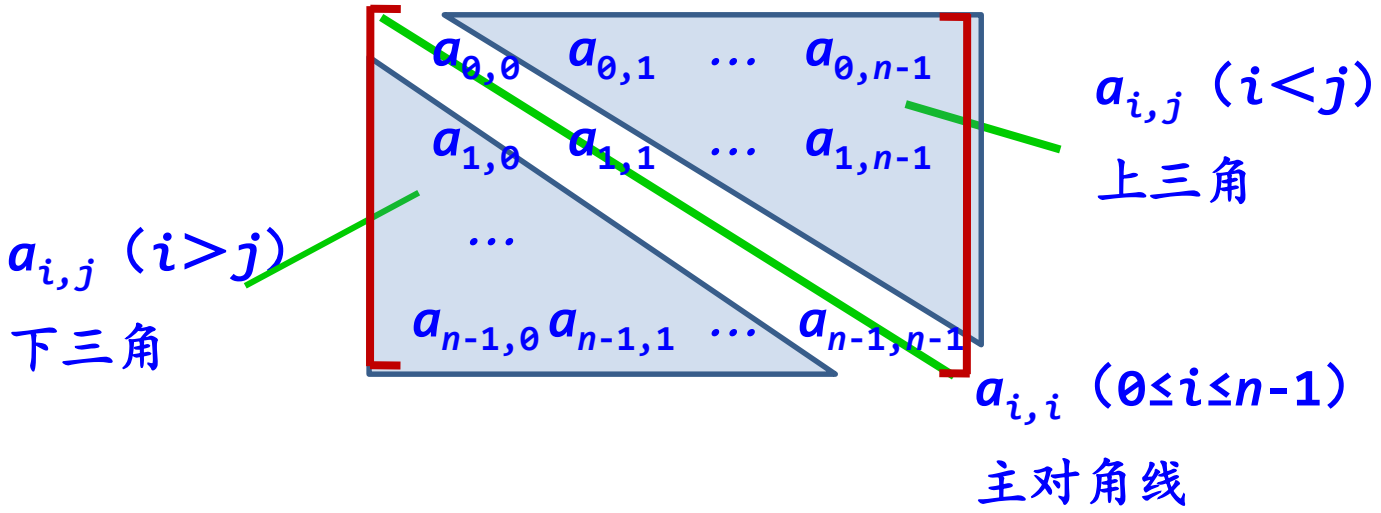
（2）上三角矩阵 / 下三角矩阵

（3）对角矩阵

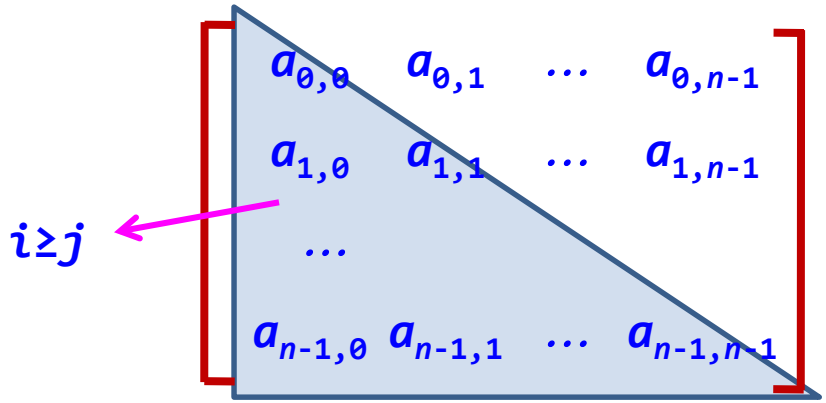
它们都是方阵，即行数和列数相同。

1 对称矩阵的压缩存储

若一个n阶方阵$A[n][n]$中的元素满足$a_{i,j}=a_{j,i}$（$0\le i, j\le n-1$），则称其为n阶对称矩阵。



以行序为主序存储其下三角+主对角线的元素。



$a_{i,j}$

下三角+主对角线

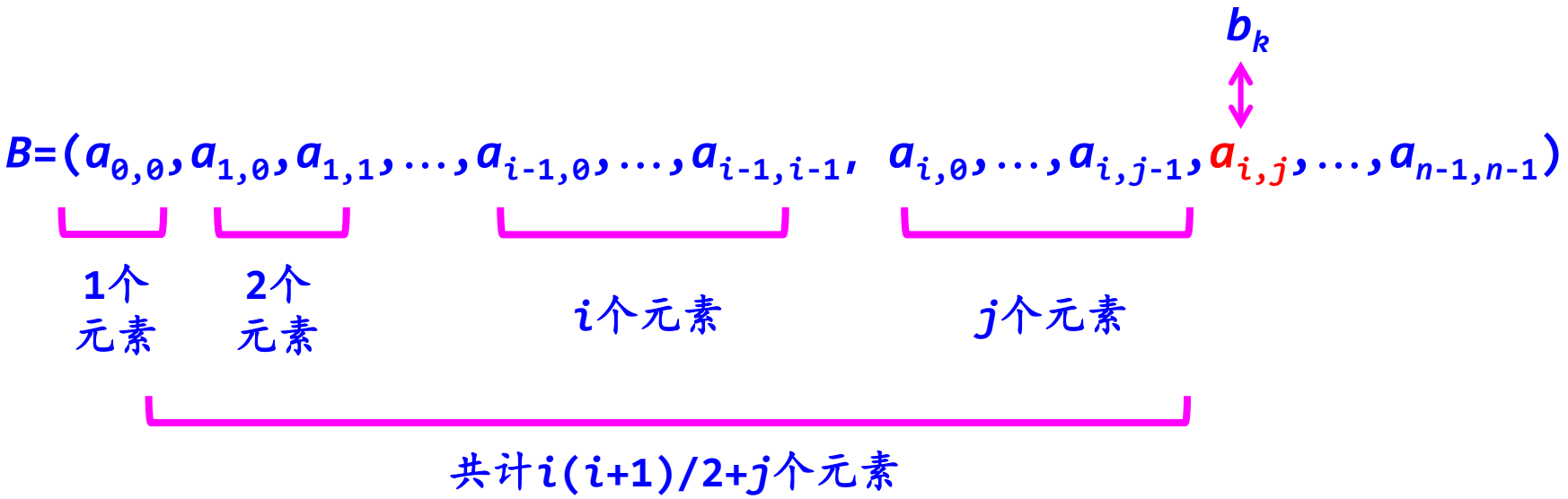
b_k

k = ?

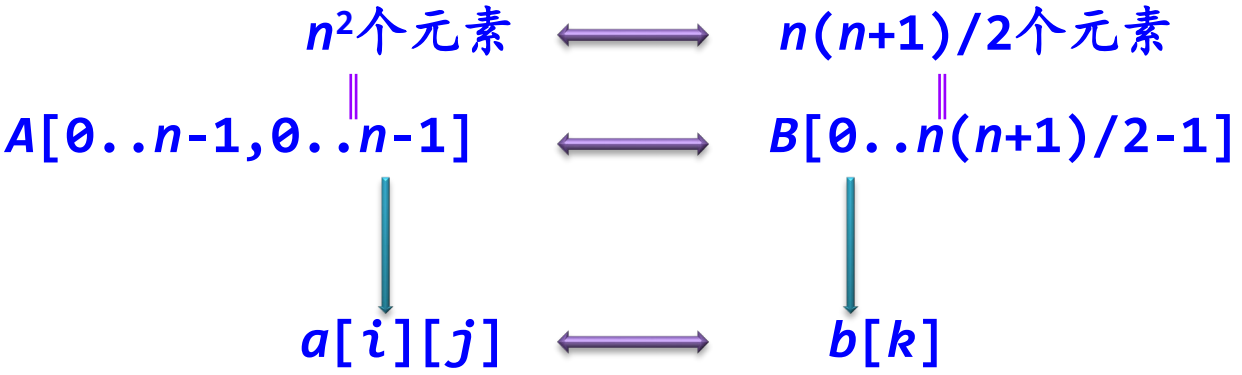
$a_{0,0}$, $a_{1,0}$, $a_{1,1}$, …, $a_{n-1,0}$, $a_{n-1,1}$, …, $a_{n-1,n-1}$

B= (b_0, b_1, b_2, ……, ……, b_s)

$n(n+1)/2$个元素



$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & \text{当 } i\geq j\text{时（下三角+主对角线的元素）}\\[2ex] \dfrac{j(j+1)}{2}+i & \text{当}i<j\text{时（}a_{i,j}=a_{j,i}\text{）}\end{cases}$$

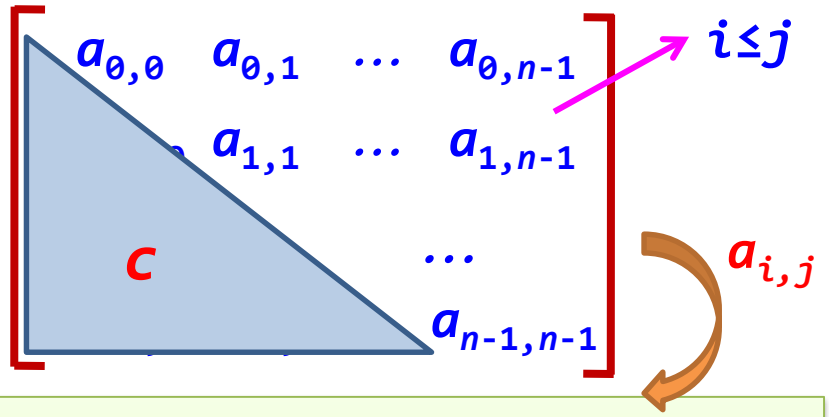


$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & \text{当 } i\geq j\text{时} \\ \dfrac{j(j+1)}{2}+i & \text{当}i<j\text{时}(a_{i,j}=a_{j,i})\end{cases}$$

对于对称矩阵A，采用一维数组B存储，并提供A的所有运算。

2 三角矩阵的压缩存储

- 上三角矩阵：



$B=(a_{0,0},a_{0,1},\ \ldots,\ a_{0,n-1},a_{1,1},\ \ldots,\ a_{i-1,n-1},$

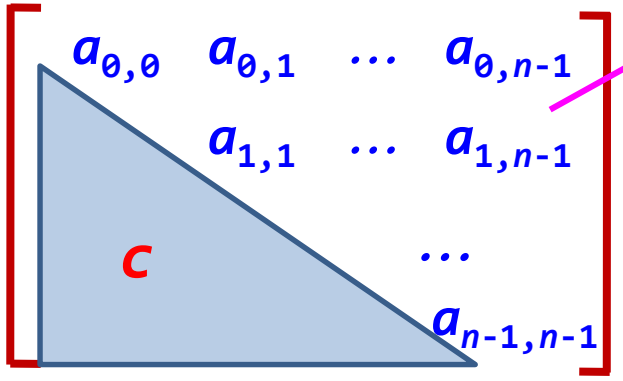
$\ldots,$

$a_{i-1,i-1},\ldots,\ a_{i-1,n-1},$

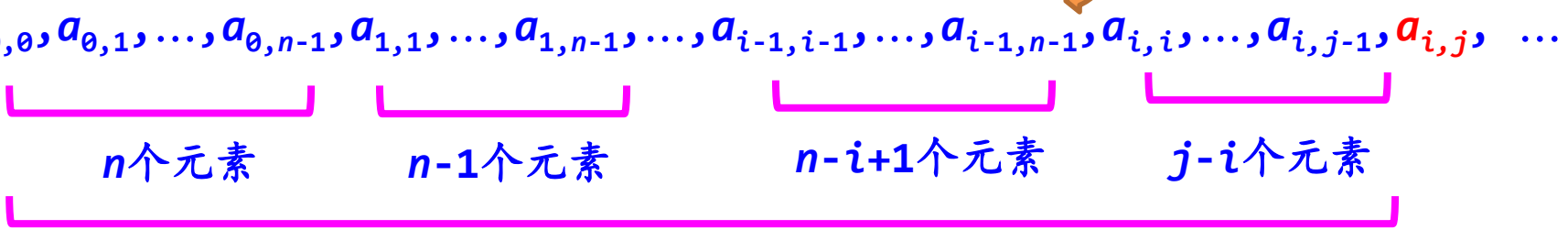
$a_{i,i}\ ,\ \ldots,a_{i,j-1},a_{i,j},\ \ldots\ )$

b_k

● 上三角矩阵：



$B=$

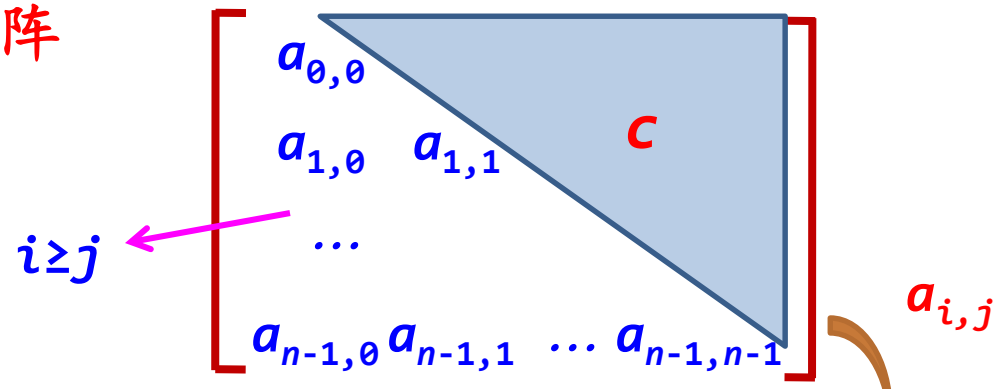


共计$i(2n-i+1)/2+j-i$个元素

$$k=\begin{cases}\dfrac{i(2n-i+1)}{2}+j-i & \text{当 } i\le j \text{ 时}\\[2ex] \dfrac{n(n+1)}{2} & \text{当 } i>j \text{ 时}\end{cases}$$

存放常量c

- 下三角矩阵



$$B=(a_{0,0},\ a_{1,0},\ a_{1,1},\ \ldots,\ a_{n-1,0},\ a_{n-1,1},\ \ldots,\ a_{n-1,n-1})$$

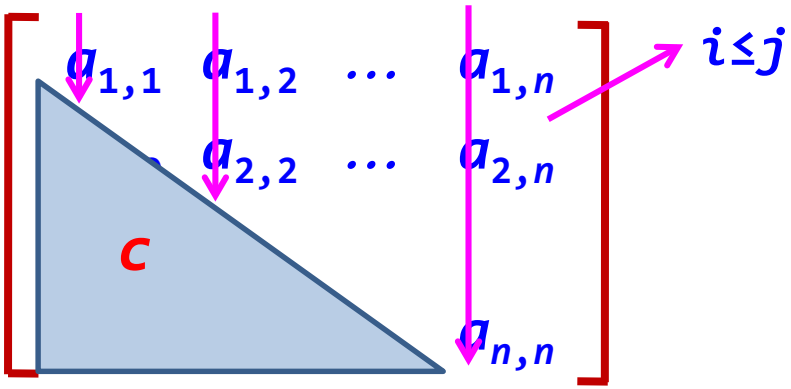
$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & \text{当 } i\geq j\text{时}\\[2ex] \dfrac{n(n+1)}{2} & \text{当 } i<j\text{时}\end{cases}$$

存放一个常量c

【例】若将n阶上三角矩阵A按列优先顺序压缩存放在一维数组$B[1..n(n+1)/2]$中，A中第一个非零元素$a_{1,1}$存于B数组的b_1中，则应存放到b_k中的非零元素$a_{i,j}$（$i \le j$）的下标i、j与k的对应关系是（　）。

A. $i(i+1)/2+j$　　B. $i(i-1)/2+j$

C. $j(j+1)/2+i$　　D. $j(j-1)/2+i$



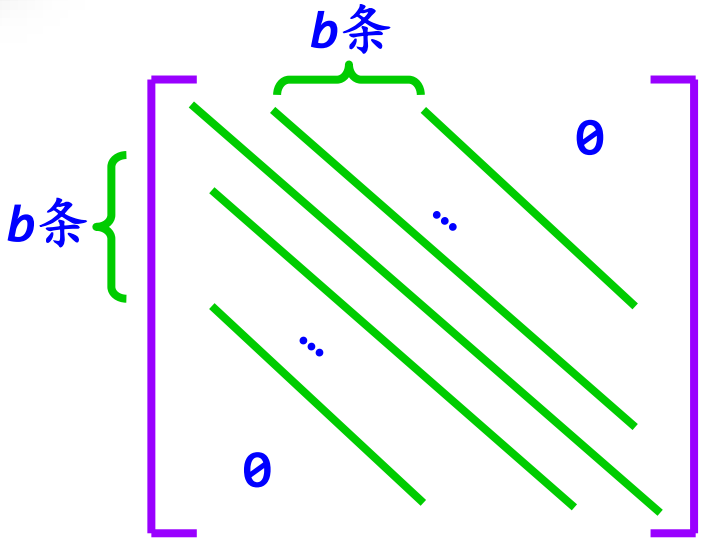
1～j-1列的元素个数：$j(j-1)/2$

第j列a_{ij}之前的元素个数：i-1

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

- 按行还是按列
- 初始下标从0还是从1开始

3 对角矩阵的压缩存储



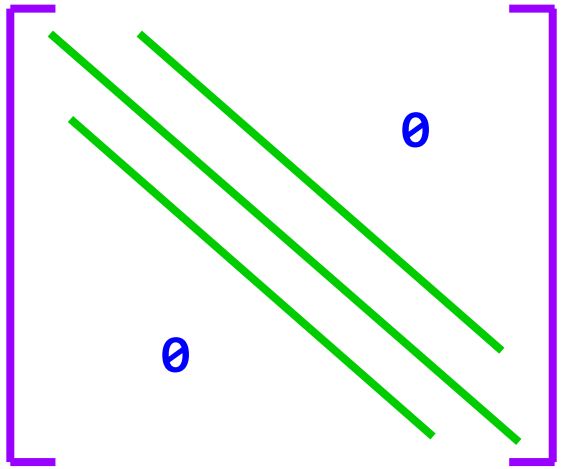
半带宽为b的对角矩阵

对角矩阵　　压缩存储

A ⟺ B

$a[i][j]$ ⟺ $b[k]$

当b=1时称为三对角矩阵



其压缩地址计算公式如下：

$$k = 2i + j$$

6.2 稀疏矩阵

稀疏矩阵的定义

一个阶数较大的矩阵中的非零元素个数s相对于矩阵元素的总个数t十分小的时候，即s<<t时，称该矩阵为稀疏矩阵。

例如一个100×100的矩阵，若其中只有100个非零元素，就可称其为稀疏矩阵。

定性的描述

稀疏矩阵和特殊矩阵的不同点：

- 特殊矩阵的特殊元素（值相同元素、常量元素）分布有规律。
- 稀疏矩阵的特殊元素（非0元素）分布没有规律。

6.2.1 稀疏矩阵的三元组表示

稀疏矩阵的压缩存储方法是只存储非零元素。

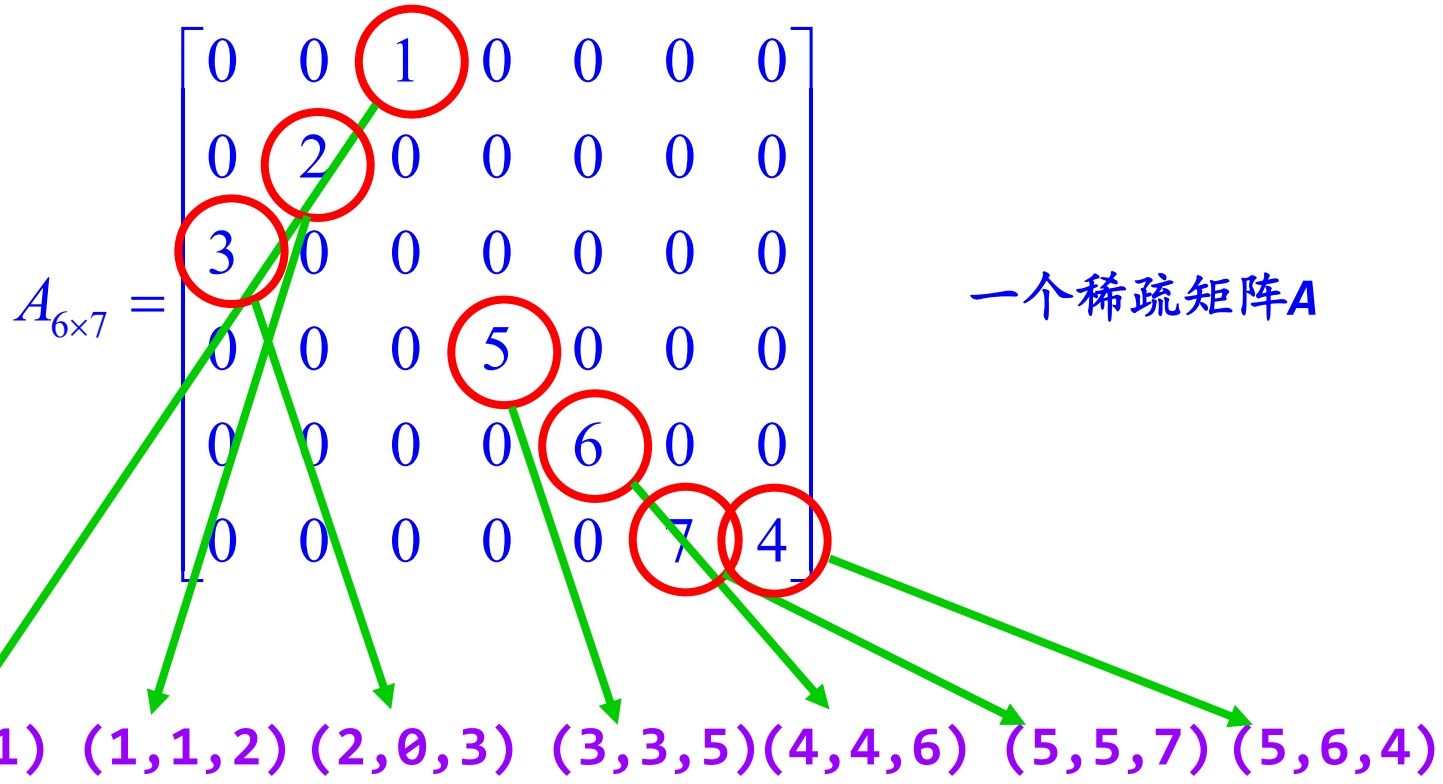
稀疏矩阵中的每一个非零元素需由一个三元组：

$$(i\ ,\ j\ ,\ a_{i,j})$$

唯一确定，稀疏矩阵中的所有非零元素构成三元组线性表。

稀疏矩阵三元组表示的演示

一个6×7阶稀疏矩阵A的三元组线性表表示



三元组线性表：

((0,2,1), (1,1,2), (2,0,3), (3,3,5), (4,4,6), (5,5,7), 5,6,4))

把稀疏矩阵的三元组线性表按顺序存储结构存储，则称为稀疏矩阵的三元组顺序表。

```
#define MaxSize  100   //矩阵中非零元素最多个数
typedef struct
{   int r;             //行号
    int c;             //列号
    ElemType d;        //元素值
} TupNode;             //三元组定义
typedef struct
{   int rows;          //行数值
    int cols;          //列数值
    int nums;          //非零元素个数
    TupNode data[MaxSize];
} TSMatrix;            //三元组顺序表定义
```

存放一个非0元素

存放整个稀疏矩阵

(1) 从一个二维矩阵创建其三元组表示

以行序方式扫描二维矩阵A，将其非零的元素插入到三元组t的后面。

$$A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

t:

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

```
void CreatMat(TSMatrix &t,ElemType A[M][N])
{  int i,j; t.rows=M; t.cols=N; t.nums=0;
  for (i=0;i<M;i++)
  {  for (j=0;j<N;j++)
       if (A[i][j]!=0)
       {  t.data[t.nums].r=i;
          t.data[t.nums].c=j;
          t.data[t.nums].d=A[i][j];
          t.nums++;
       }
   }
}
```

按行序方式扫描所有元素

只存储非零元素

（2）三元组元素赋值：A[i][j]=x

分为两种情况：①将一个非0元素修改为另一个非0值，如A[5][6]=8。

修改元素

$A_{6\times7}=$

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 4 \end{bmatrix}$$

$A_{6\times7}=$

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

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	8

②将一个0元素修改为非0值。如A[3][5]=8

增加元素

$A_{6\times7}=$

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 4 \end{bmatrix}$$

$A_{6\times7}=$

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 4 \end{bmatrix}$$

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
3	5	8
4	4	6
5	5	7
5	6	8

算法如下：

```
bool Value(TSMatrix &t,ElemType x,int i,int j)
{  int k=0,k1;
   if (i>=t.rows || j>=t.cols)
      return false;                              //失败时返回false
  while (k<t.nums && i>t.data[k].r) k++;   //查找行
     while (k<t.nums && i==t.data[k].r
              && j>t.data[k].c)   k++;     //查找列
```

在t中按行、列号查找

```
  if (t.data[k].r==i && t.data[k].c==j)    //存在这样的元素
    t.data[k].d=x;
  else                  //不存在这样的元素时插入一个元素
  {  for (k1=t.nums-1;k1>=k;k1--)
     {  t.data[k1+1].r=t.data[k1].r;
        t.data[k1+1].c=t.data[k1].c;
        t.data[k1+1].d=t.data[k1].d;
     }
     t.data[k].r=i;t.data[k].c=j;t.data[k].d=x;
     t.nums++;
  }
  return true;          //成功时返回true
}
```

修改元素

增加元素

(3) 将指定位置的元素值赋给变量　执行x=A[i][j]

先在三元组t中找到指定的位置，再将该处的元素值赋给x。

```
bool Assign(TSMatrix t,ElemType &x,int i,int j)
{  int k=0;
   if (i>=t.rows || j>=t.cols)
      return false;              //失败时返回false

   while (k<t.nums && i>t.data[k].r) k++;   //查找行
   while (k<t.nums && i==t.data[k].r
          && j>t.data[k].c)   k++;          //查找列

   if (t.data[k].r==i && t.data[k].c==j)
      x=t.data[k].d;
   else
      x=0;
   return true;                  //成功时返回true
}
```

在t中按行、列号查找

找到了非0的元素

没有找到：为0元素

(4) 输出三元组

从头到尾扫描三元组t，依次输出元素值。

```
void DispMat(TSMatrix t)
{  int i;
   if (t.nums<=0) return;
   printf("\t%d\t%d\t%d\n",t.rows,t.cols,t.nums);
   printf("  ------------------\n");
   for (i=0;i<t.nums;i++)
      printf("\t%d\t%d\t%d\n",t.data[i].r,t.data[i].c,t.data[i].d);
}
```

(5) 矩阵转置

对于一个$m \times n$的矩阵$A_{m \times n}$，其转置矩阵是一个$n \times m$的矩阵$B_{n \times m}$，满足$b_{i,j}=a_{j,i}$，其中$0 \le i \le m-1$，$0 \le j \le n-1$。

$$A_{6\times7}=\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 4 \end{bmatrix} \Rightarrow B_{7\times6}=\begin{bmatrix} 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

⇒

i	j	b_{ij}
0	2	3
1	1	2
2	0	1
3	3	5
4	4	6
5	5	7
6	5	4

一种非高效的算法：按第0、1、2、…、n-1列进行转换

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

i	j	b_{ij}
0	2	3
1	1	2
2	0	1
3	3	5
4	4	6
5	5	7
6	5	4

矩阵转置

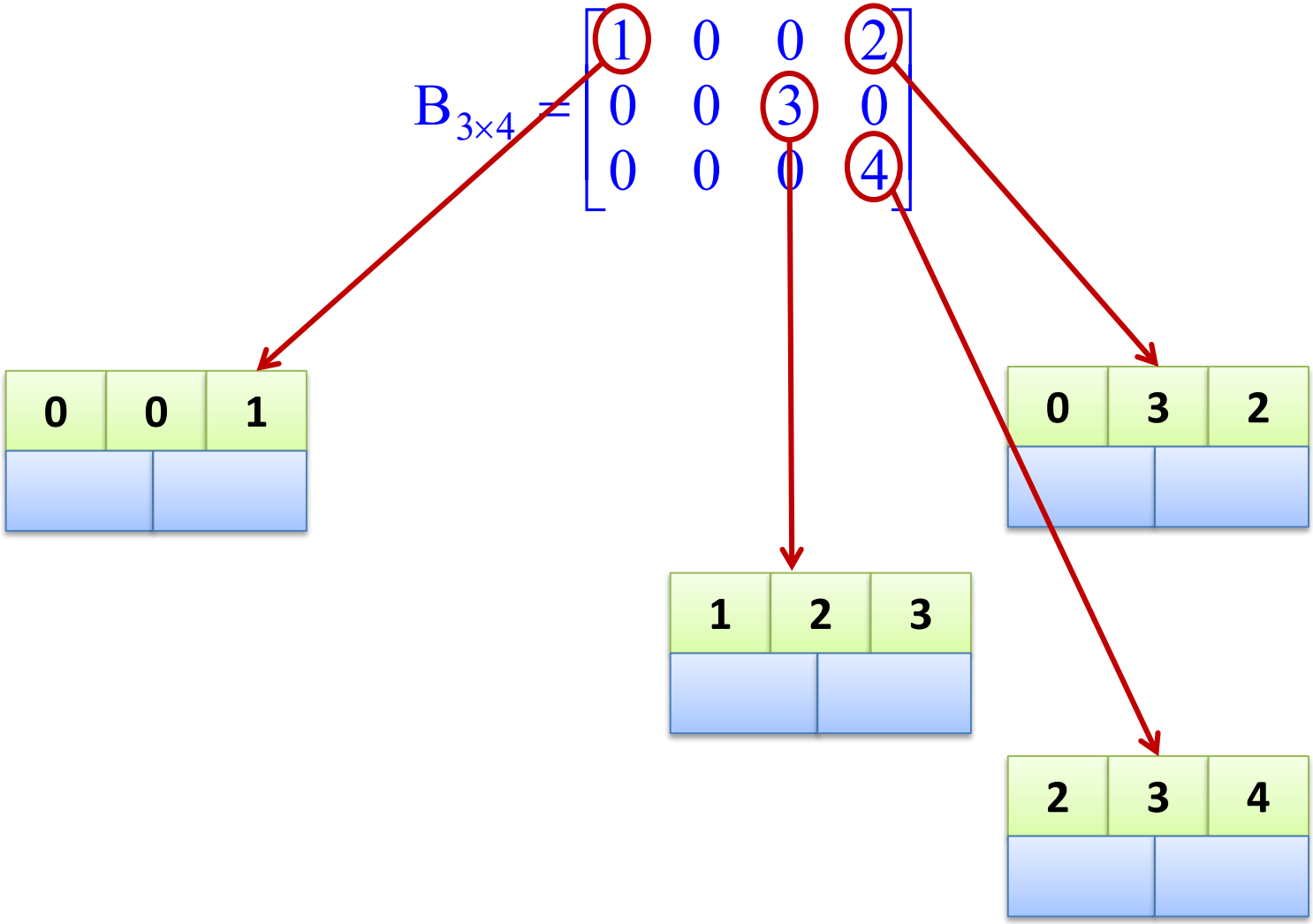
```
void TranTat(TSMatrix t,TSMatrix &tb)
{  int p,q=0,v;                          //q为tb.data的下标
   tb.rows=t.cols;  tb.cols=t.rows;  tb.nums=t.nums;
   if (t.nums!=0)                        //当存在非零元素时执行转置
   {
      for (v=0;v<t.cols;v++)             //tb.data[q]中记录以列序排列
         for (p=0;p<t.nums;p++)          //p为t.data的下标
            if (t.data[p].c==v)
            {  tb.data[q].r=t.data[p].c;
               tb.data[q].c=t.data[p].r;
               tb.data[q].d=t.data[p].d;
               q++;
            }
   }
}
```

按第0、1、2、…、$n-1$ 列进行转换

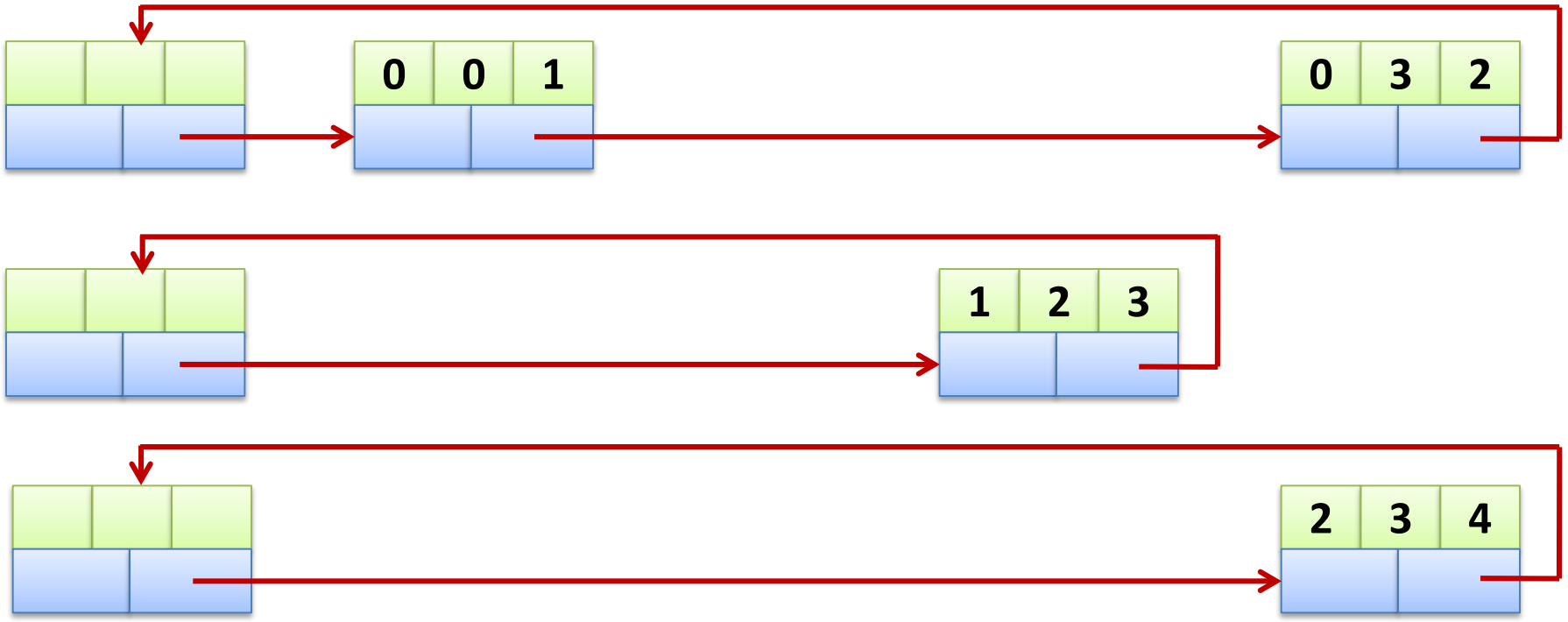
m行n列，t个非0元素，时间复杂度为$O(nt)$。

6.2.2 稀疏矩阵的十字链表表示

- 每个非零元素对应一个结点。



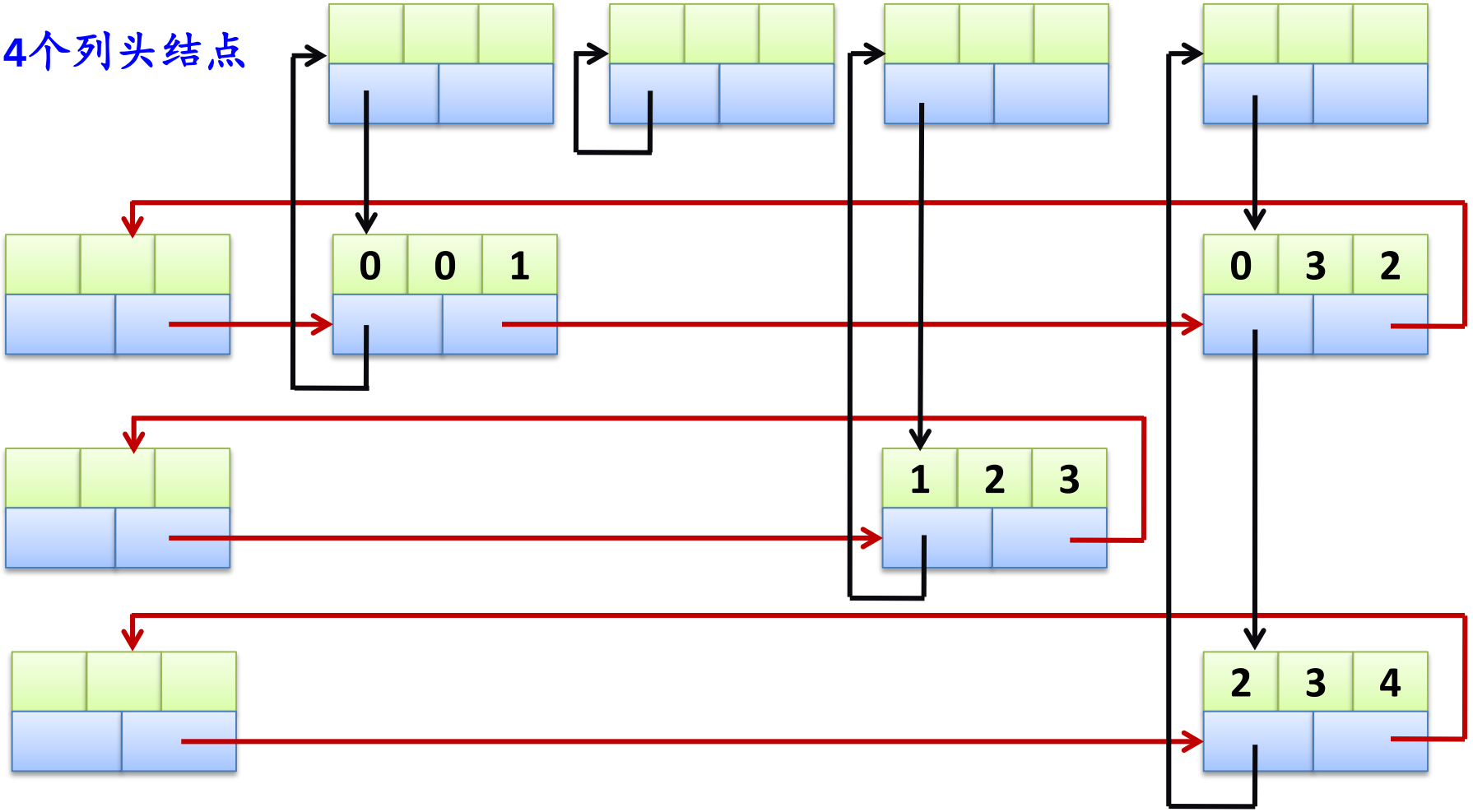
- 每行的所有结点链起来构成一个带行头结点的循环单链表。以h[i]（0≤i≤m-1）作为第i行的头结点。



3个行头结点

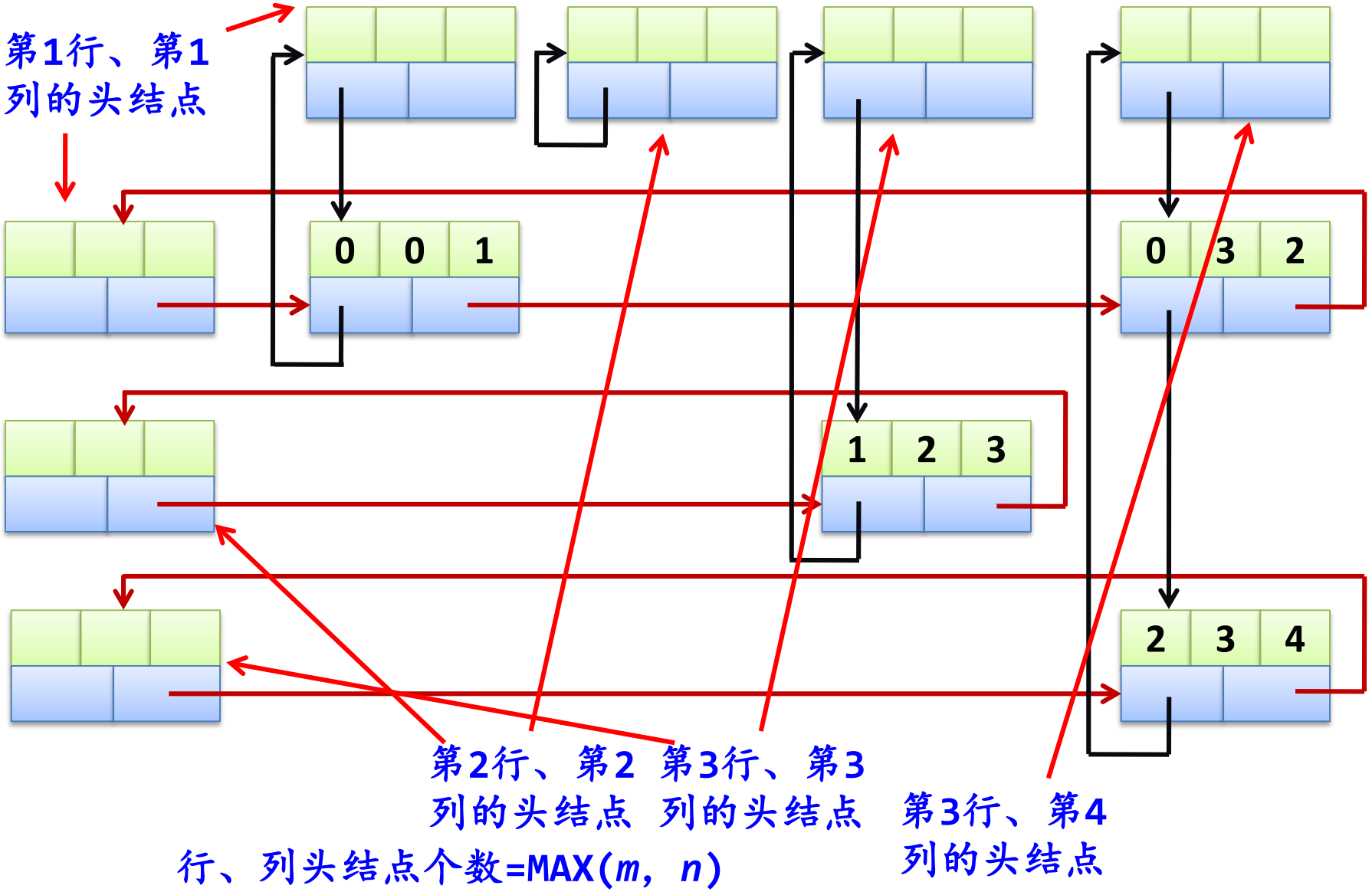
- 每列的所有结点链起来构成一个带列头结点的循环单链表。 以h[i]（0≤i≤m-1）作为第i列的头结点。

4个列头结点

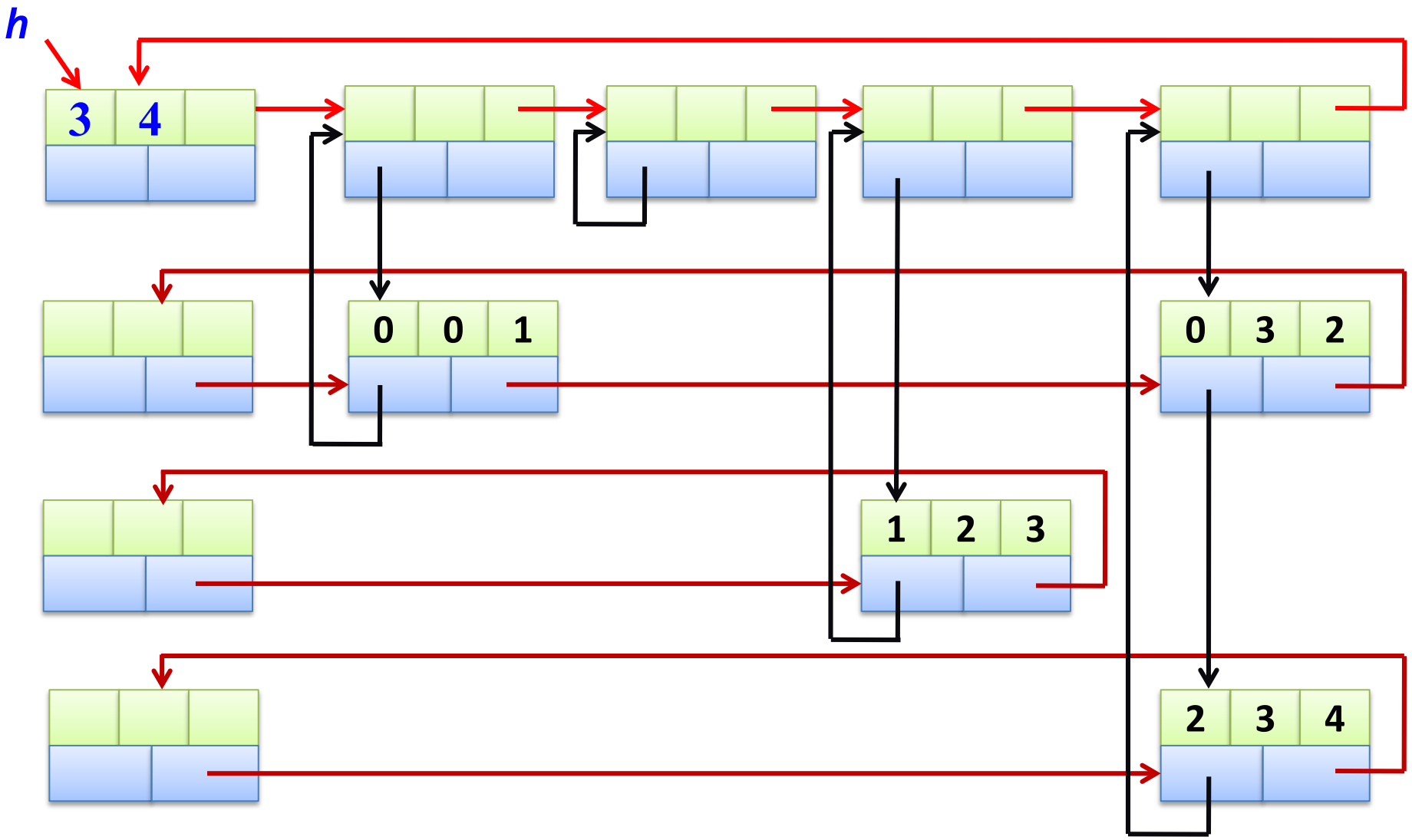


3个行头结点

行、列头结点可以共享

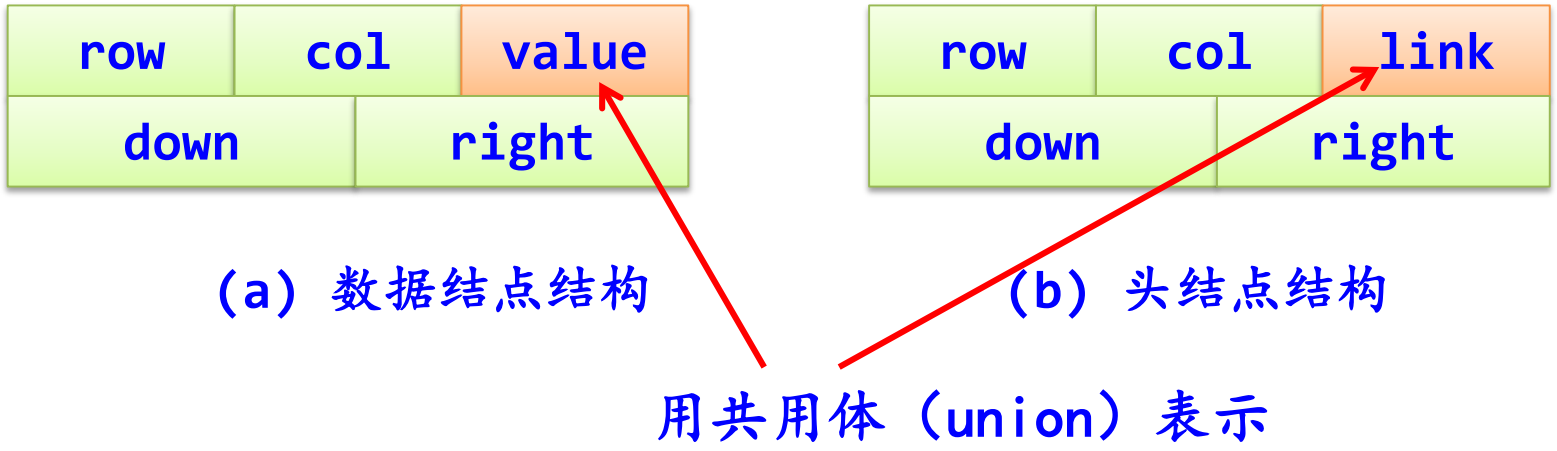

第1行、第1列的头结点
0 0 1
0 3 2
1 2 3
2 3 4
第2行、第2列的头结点
第3行、第3列的头结点
第3行、第4列的头结点
行、列头结点个数=MAX(m，n)

增加一个总头结点，并把所有行、列头结点链起来构成一个循环单链表



总的头结点个数=MAX(*m*,*n*)+1

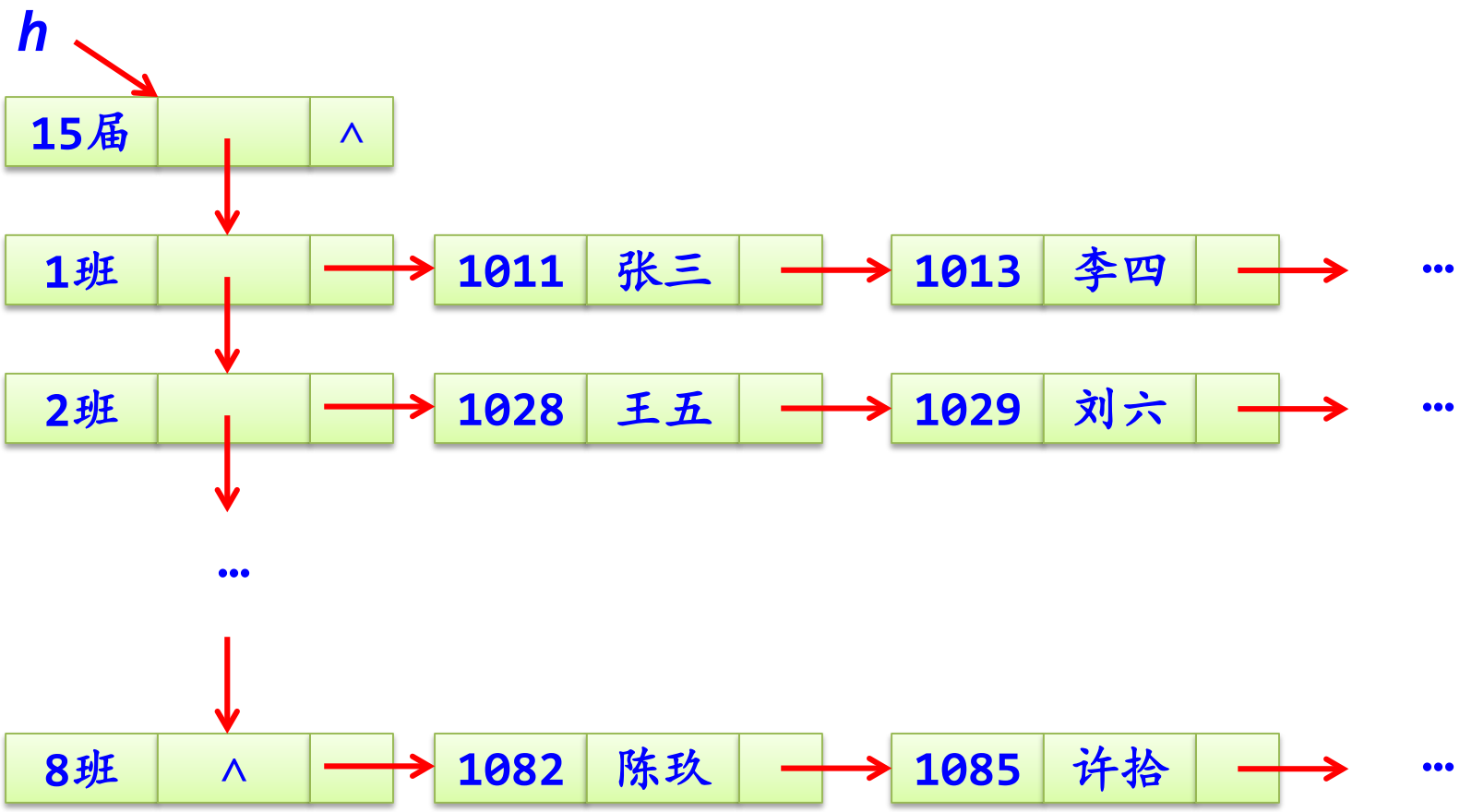
设计结点类型如下：


row
col
value
down
right
row
col
link
down
right
(a) 数据结点结构
(b) 头结点结构
用共用体（union）表示

十字链表结点结构和头结点的数据结构可定义如下：

```
#define M 3                                  //矩阵行
#define N 4                                  //矩阵列
#define Max ((M)>(N)?(M):(N))                //矩阵行列较大者
typedef struct mtxn
{   int row;                                 //行号
    int col;                                 //列号
    struct mtxn *right,*down;                //向右和向下的指针
    union                                    //共用体类型
    {   int value;
        struct mtxn *link;
    } tag;
 }  MatNode;                                 //十字链表结点类型声明
```

【例】十字链表的启示：设计存储某年级所有学生的存储结构：



通过h来唯一标识学生存储结构。

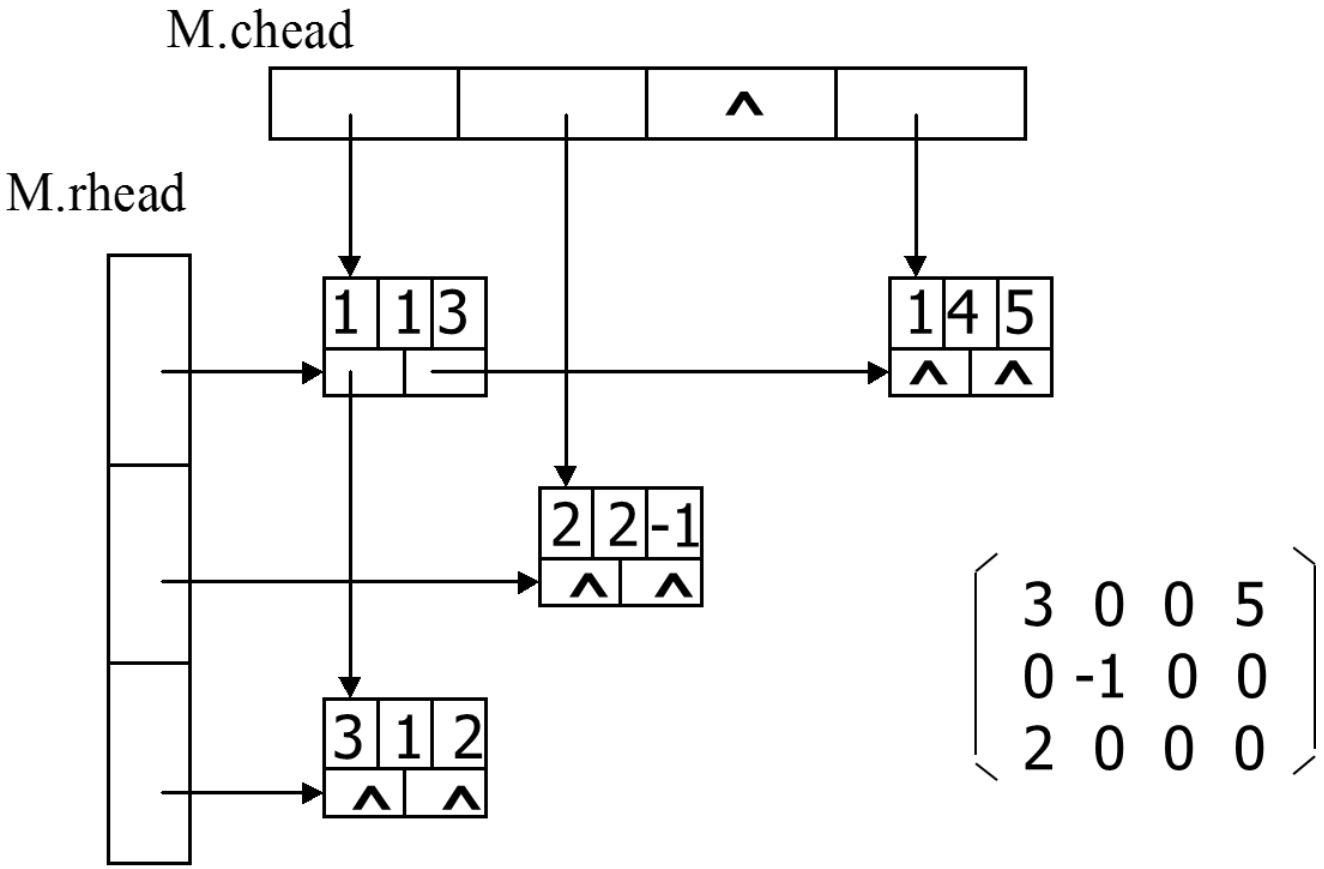
思考题

一个稀疏矩阵采用压缩后，和直接采用二维数组存储相比会失去（　）特性。

A.顺序存储　　B.随机存取

C.输入输出　　D.以上都不对

十字链表的另一种描述



6.3 广义表

6.3.1 广义表的定义

广义表是线性表的推广，是有限个元素的序列，其逻辑结构采用括号表示法表示如下：

$$GL=(a_1, a_2, \ldots, a_i, \ldots, a_n)$$

- 若$n=0$时称为空表。
- a_i为广义表的第i个元素。如果a_i属于原子类型，称之为广义表GL的原子。
- 如果a_i又是一个广义表，称之为广义表GL的子表。

广义表重要概念：

$$GL=(a_1, a_2, \ldots, a_i, \ldots, a_n)$$

- 广义表的长度定义为最外层包含元素个数。
- 广义表的深度定义为所含括弧的重数。其中，原子的深度为0，空表的深度为1 。
- 广义表GL的表头为第一个元素a_1，其余部分$(a_2, \ldots, a_n)$为GL的表尾。一个广义表的表尾始终是一个广义表。空表无表头表尾。
  - 任何一个非空广义表GL均可分解为表头head(GL) = a1和表尾tail(GL) = (a2,…,an) 两部分。
- 广义表可以是一个递归的表。一个广义表可以是自己的子表。

6.3.2 广义表的存储结构

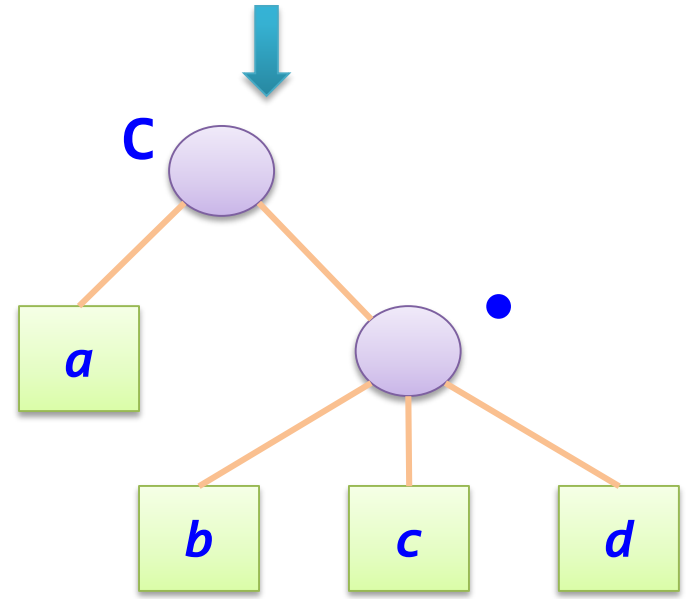
广义表的几种逻辑结构的演变，例如：

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

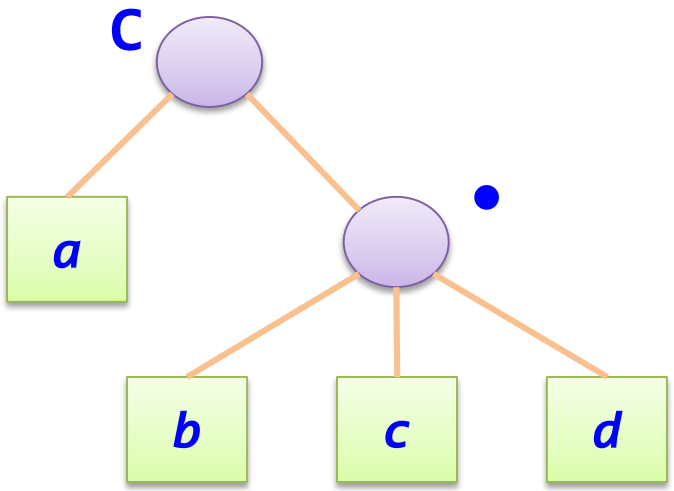
☜ 括号表示

C=(a, • (b, c, d))

☜子表加匿名“•”



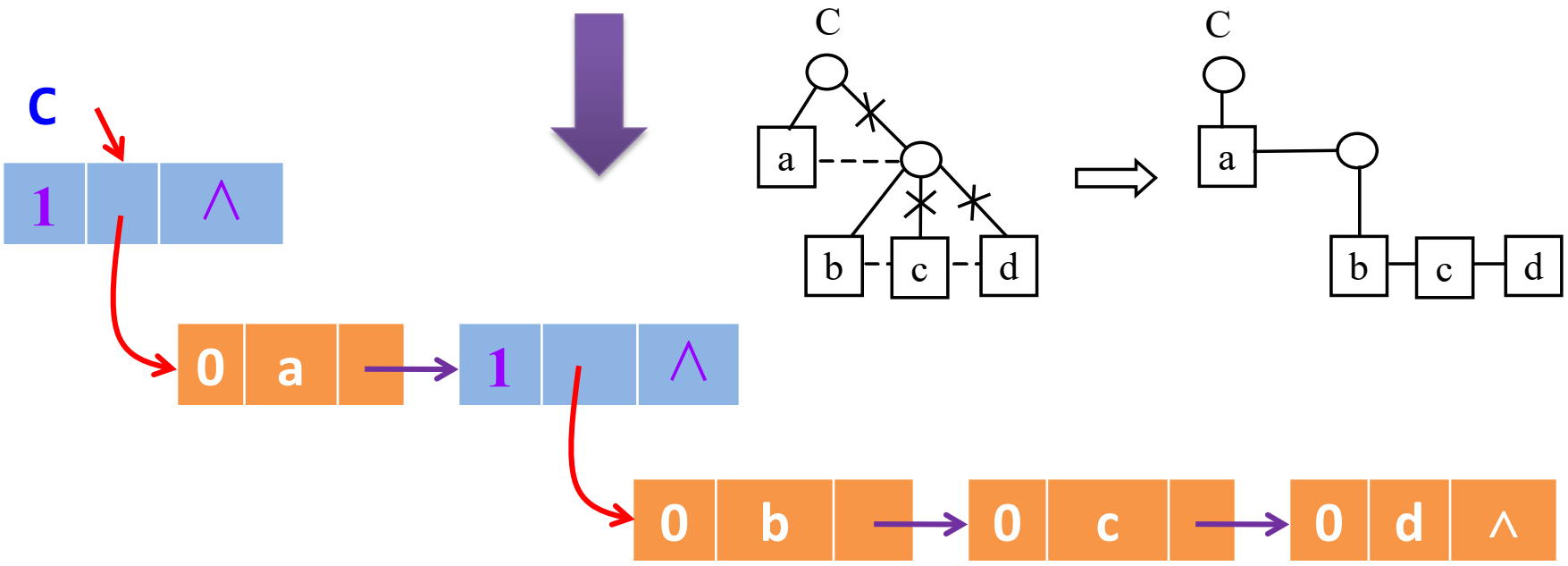
☜ 树形表示



结点类型：

tag	Sublist/data	link

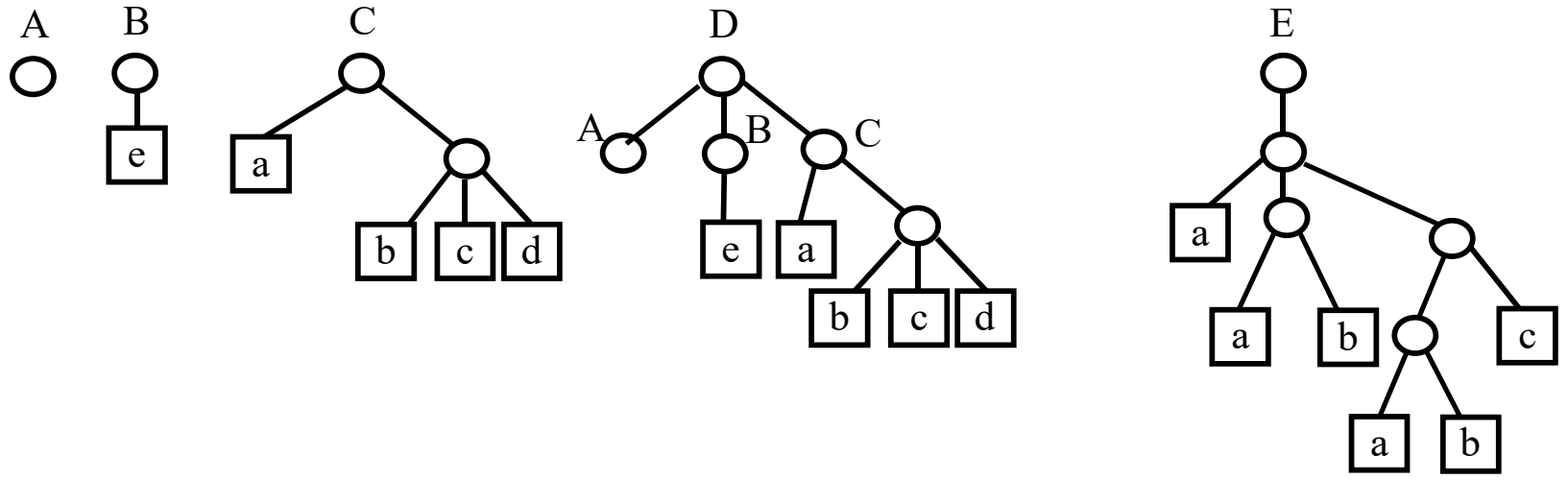
- 若tag=0，表示该结点为原子结点
- 若tag=1，表示该结点为表/子表结点



广义表

A=()　　B=(e)　　C=(a, (b,c,d))

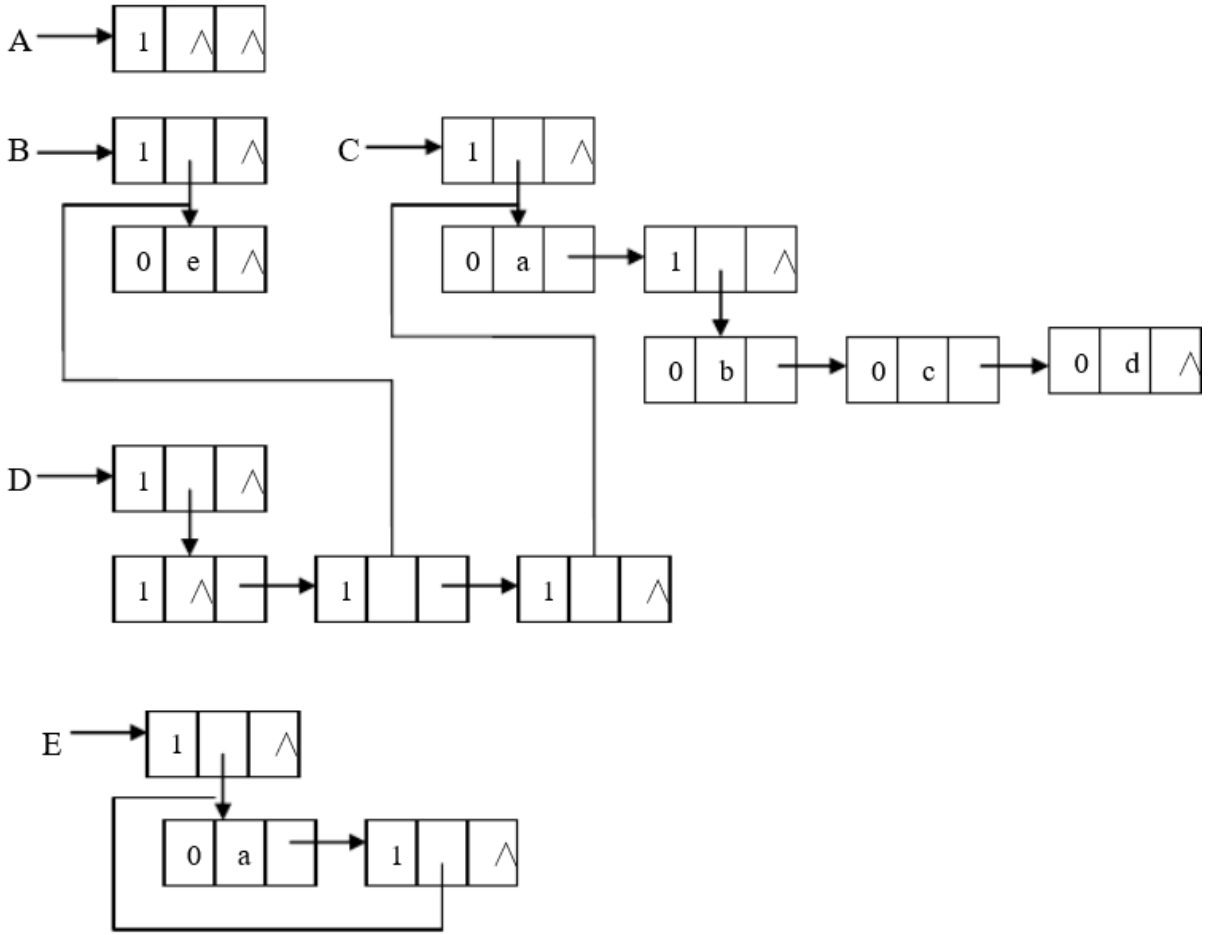
D(A,B,C)　　E((a, (a,b), ((a,b),c)))



广义表

A=()　　B=(e)　　C=(a, (b,c,d))

D(A,B,C)　　E((a, (a,b), ((a,b),c)))　　F(a, F)



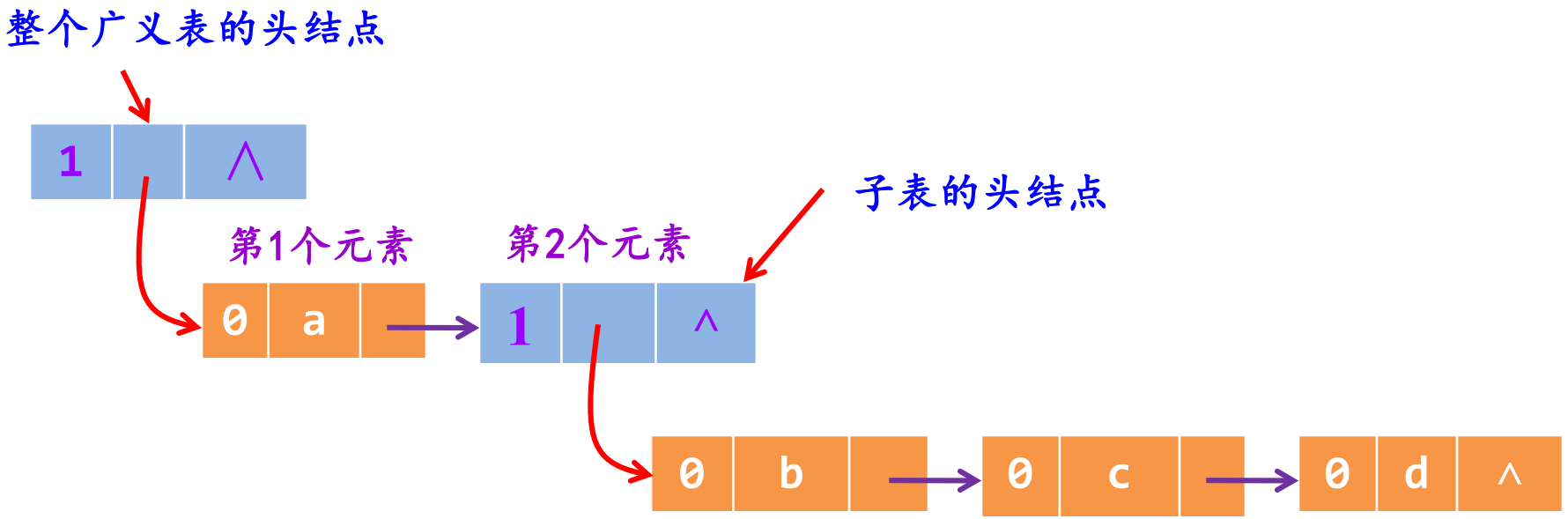
广义表的结点类型GLNode

```
typedef struct lnode
{  int tag;                              //结点类型标识
   union
   {    ElemType data;                   //存放原子值
        struct lnode *sublist;           //指向子表的指针
   } val;
   struct lnode *link;                   //指向下一个元素
} GLNode;                                //广义表的结点类型
```

6.3.3 广义表的运算

1 广义表算法设计方法

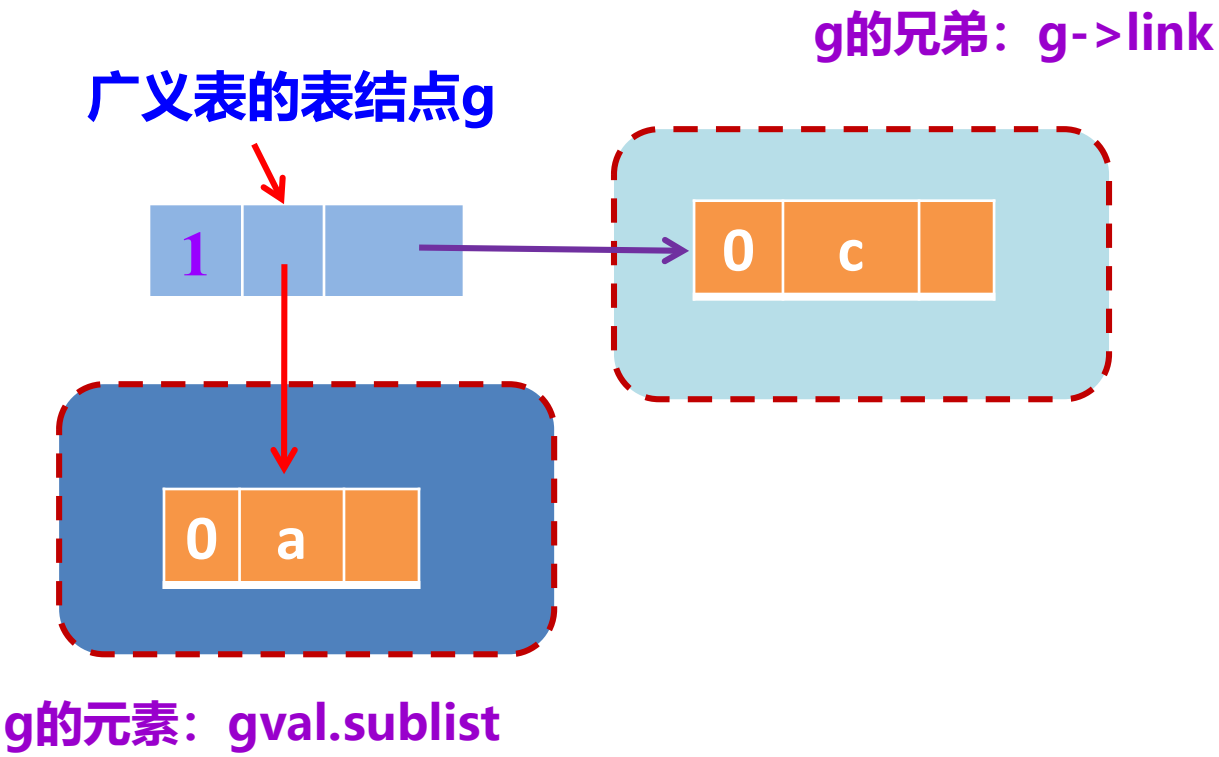
步骤1



子表的处理和整个广义表的处理是相似的。广义表递归算法的一般格式如下：

```
void fun1(GLNode *g)                    //g为广义表头结点指针
{  GLNode *g1=g->val.sublist;           //g1指向第一个元素
   while (g1!=NULL)                     //元素未处理完循环
   {  if (g1->tag==1)                   //为子表时
         fun1(g1);                      //递归处理子表
      else                              //为原子时
         原子处理语句;                  //实现原子操作
      gl=g1->link;                      //处理兄弟
   }
}
```

步骤2



兄弟的处理与整个广义表的处理是相似的；对于子表结点，其元素的处理与整个广义表的处理是相似的。从这个角度出发设计求解广义表递归算法的一般格式如下：

```
void fun2(GLNode *g)                    //g为广义表结点指针
{  if (g!=NULL)
   {    if (g->tag==1)                  //为子表时
            fun2(g->val.sublist);       //递归处理其元素
        else                            //为原子时
            原子处理语句;                //实现原子操作
        fun2(g->link);                  //递归处理其兄弟
   }
}
```

2 广义表基本算法设计

（1）求广义表的长度

在广义表中，同一层次的每个结点是通过link域链接起来的，所以可把它看做是由link域链接起来的单链表。这样，求广义表的长度就是求单链表的长度。

```
int GLLength(GLNode *g)          //求广义表g的长度
{  int n=0;                      //累计元素个数，初始值为0
   GLNode *g1;
   g1=g->val.sublist;            //g1指向广义表的第一个元素
   while (g1!=NULL)              //扫描所有元素结点
   {    n++;                     //元素个数增1
        g1=g1->link;
   }
   return n;                     //返回元素个数
}
```

（2）求广义表的深度

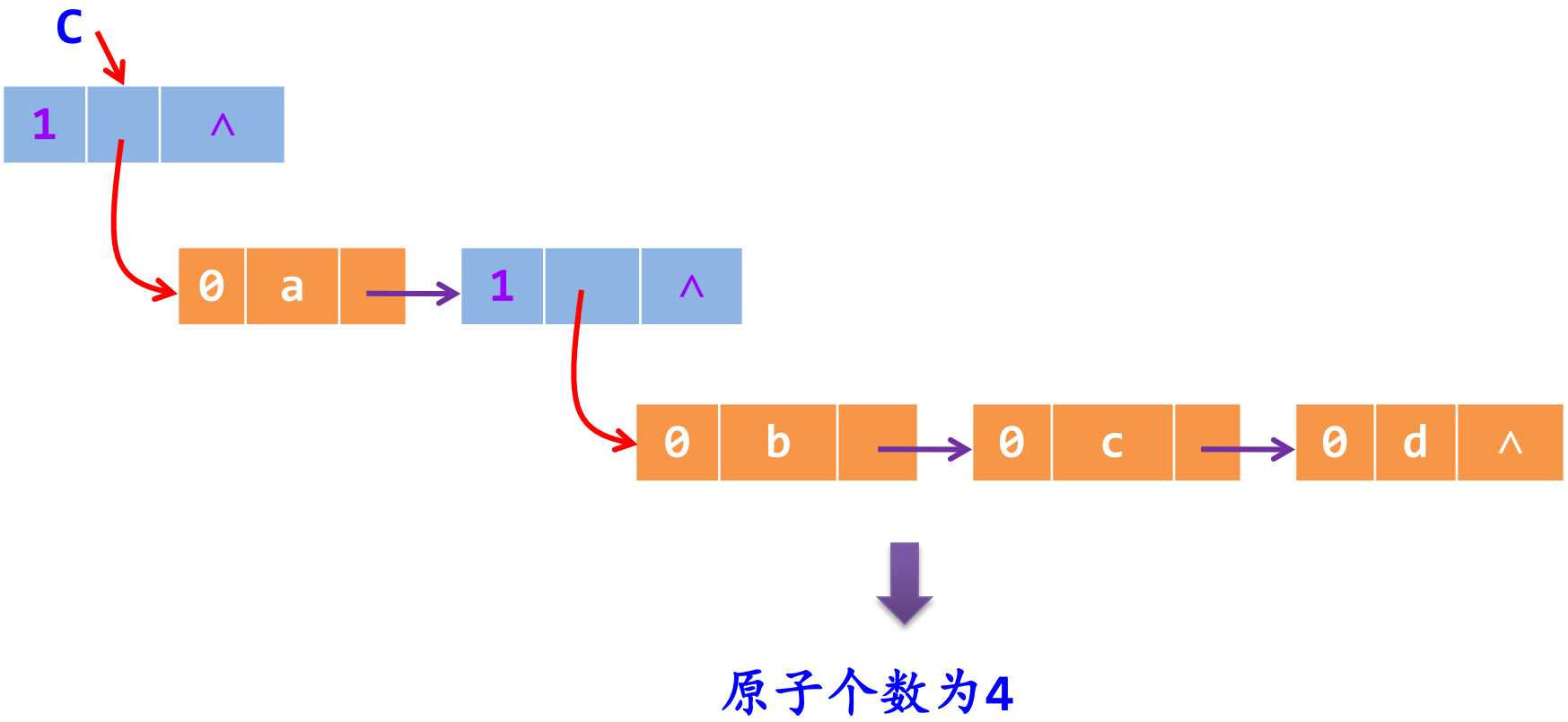
对于带头结点的广义表g，广义表深度的递归定义是它等于所有子表中表的最大深度加1。若g为原子，其深度为0。

求广义表深度的递归模型*f*()如下：

f(g)=0	g为原子
f(g)=1	g为空表
f(g)=MAX{f(subg)}+1	其他情况

```
int GLDepth(GLNode *g)                    //求广义表g的深度
{  GLNode *g1;  int maxd=0, dep;
   if (g->tag==0) return 0;               //为原子时返回0
   g1=g->val.sublist;                     //g1指向第一个元素
   if (g1==NULL)  return 1;               //为空表时返回1
   while (g1!=NULL)                       //遍历表中的每一个元素
   {  if (g1->tag==1)                     //元素为子表的情况
      {  dep=GLDepth(g1);                 //递归调用求出子表的深度
         if (dep>maxd)                    //maxd为同层子表深度的最大值
            maxd=dep;
      }
      g1=g1->link;                        //使g1指向下一个元素
   }
   return(maxd+1);                        //返回表的深度
}
```

【例6-3】对于采用链式存储结构的广义表g，设计一个算法求原子个数。



解：需要扫描广义表g中的所有结点，可以采用前面介绍的广义表算法设计方法中的两种解法来实现。

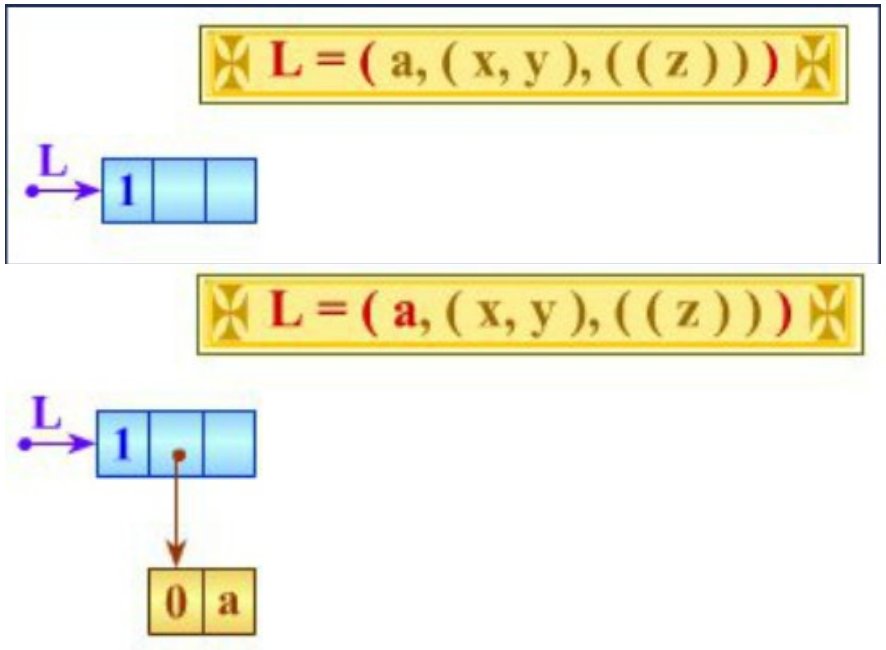
```
//采用解法1的方法
int Count1(GLNode *g)          //求广义表g的原子个数
{  int n=0;
   GLNode *g1=g->val.sublist;
   while (g1!=NULL)            //对每个元素进行循环处理
   {  if (g1->tag==0)          //为原子时
         n++;                  //原子个数增1
      else                     //为子表时
         n+=Count1(g1);        //累加元素的原子个数
      g1=g1->link;             //累加兄弟的原子个数
   }
   return n;                   //返回总原子个数
}
```

```
//采用解法2的方法
int Count2(GLNode *g)                          //求广义表g的原子个数
{  int n=0;
   if (g!=NULL)                                //对每个元素进行循环处理
   {  if (g->tag==0)                           //为原子时
         n++;                                  //原子个数增1
      else                                     //为子表时
         n+=Count2(g->val.sublist);            //累加元素的原子个数
      n+=Count2(g->link);                      //累加兄弟的原子个数
   }
   return n;                                   //返回总原子个数
}
```

广义表

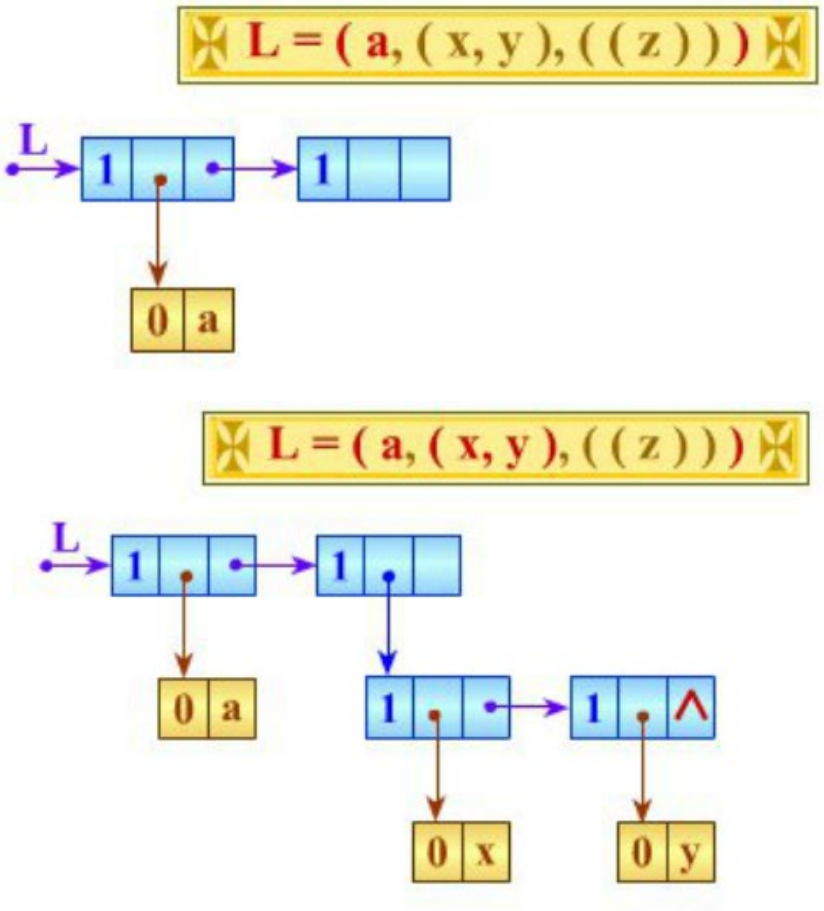
- 头尾链存储：例子 A=()

- 头尾链存储：例子 L=(a, (x,y), ((z)))



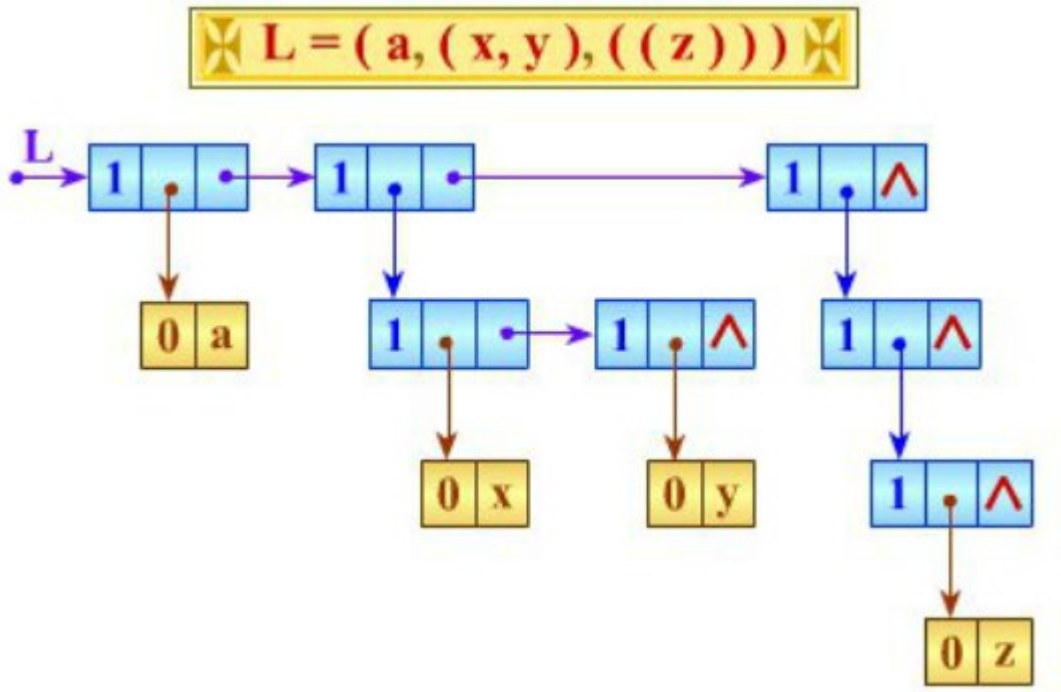
广义表

– 头尾链存储：例子 L=(a, (x,y), ((z)))



广义表

– 头尾链存储：例子 L=(a, (x,y), ((z)))



第6章小结

1 数 组

❶ 为什么说数组是线性表的推广或扩展，而不说数组就是一种线性表呢?

从逻辑结构的角度看，一维数组是一种线性表。

二维数组可以看成数组元素为一维数组的一维数组，所以二维数组是线性结构，可以看成是线性表。

但就二维数组的形状而言，它又是非线性结构，因此将二维数组看成是线性表的推广更准确。三维及以上维的数组亦如此。

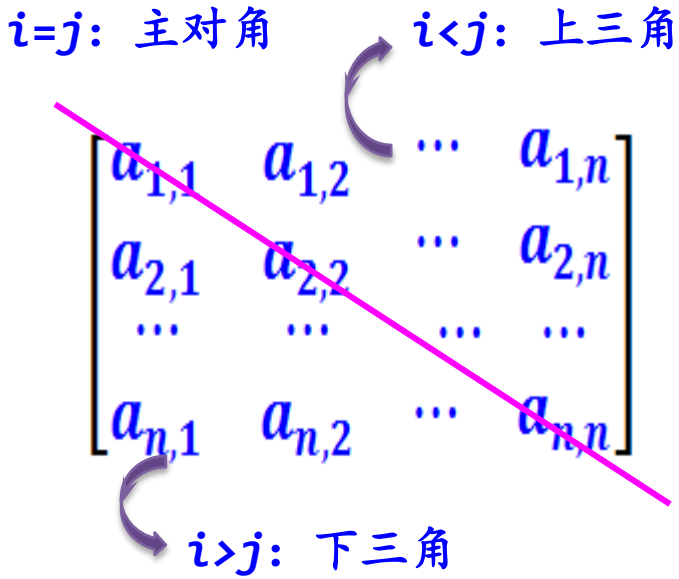
❷ 计算数组中给定元素的地址

- 数组的存储方式（按行或者按列优先存放）
- 计算给定元素的前面的元素个数s
- 每个元素的存储空间k
- 该元素地址=起始元素地址+$s \times k$

2 特 殊 矩 阵

特殊矩阵
- 对称矩阵
- 上（下）三角矩阵
- 对角矩阵

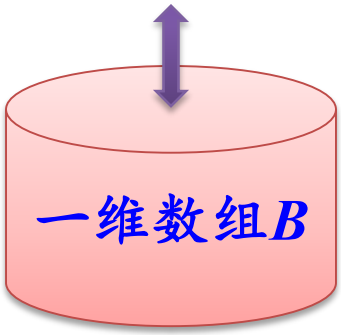
- 都是方阵
- 元素下标（i，j）可以确定元素的位置



❶ 什么是特殊矩阵的压缩存储？为什么需要压缩存储？

压缩存储：

提供二维数组的逻辑操作：$A[i][j]$



特殊矩阵采用压缩存储的目的是节省存储空间。

❷ 特殊矩阵压缩存储后具有随机存取特性吗？

❸ 在计算对称矩阵的压缩存储时应注意什么问题?

在计算对称矩阵A的压缩存储时应注意以下几点：

- 对称矩阵是按上三角还是按下三角存放。
- 对称矩阵元素是按行还是按列优先存放。
- B数组的下标从1开始还是从0开始。

3 稀 疏 矩 阵

❶ 从特殊元素分布看，稀疏矩阵和特殊矩阵相比有什么不同？

- 特殊矩阵中的特殊元素（相同元素、常数元素）分布有规律性
- 稀疏矩阵中的特殊元素（非0元素）分布没有规律性，即随机的

❷ 稀疏矩阵压缩存储后具有随机存取特性吗?

- 稀疏矩阵用十字链表作存储结构自然失去了随机存取的功能。
- 即便用三元组表的顺序存储结构，存取下标为i和j的元素时，要扫描三元组表，时间复杂度为O(t)，因此也失去了随机存取的功能。

广义表

❶ 广义表求表头、表尾?

广义表((*a*)，((*b*)，*c*)，(((*d*))))的表头是 ① ，表尾是 ② 。

答：①(*a*)　②(((*b*)，*c*)，(((*d*))))

❷ 广义表递归算法设计

先递归找子表、再移动到下一个同一层结点

——本章完——