第五章 数组和广义表

第五章　数组和广义表

前4章介绍的数据结构共同特点：

（1）都属于线性数据结构；

（2）每种数据结构中的数据元素，都作为原子数据，不再进行分解；

本章讨论的两种数据结构：数组和广义表，其共同特点是：

1）从逻辑结构上看它们，可看成是线性结构的一种扩展；

2）数据元素本身也是一个数据结构；

一　数组的概念

数组是由一组个数固定，类型相同的数据元素组成阵列。

以二维数组为例：二维数组中的每个元素都受两个线性关系的约束

即行关系和列关系，在每个关系中，每个元素a_{ij}都有且仅有一个直接前趋，都有且仅有一个直接后继。

$$A_{m\times n}=\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0\ n-1} \\ a_{10} & a_{11} & \cdots & a_{1\ n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1\ 0} & a_{m-1\ 1} & \cdots & a_{m-1\ n-1} \end{pmatrix}$$

在行关系中

a_{ij}直接前趋是 a_{ij-1}

a_{ij}直接后继是 a_{ij+1}

在列关系中

a_{ij}直接前趋是 a_{i-1j}

a_{ij}直接后继是 a_{i+1j}

二维数组也可看作这样的线性表：其每一个数据元素也是一个线性表

$A=(\alpha_0,\alpha_1,\alpha_2,\alpha_3,\alpha_4,\ldots\ldots,\alpha_p)$

其中每一个数据元素 α_j 是一个列向量的线性表

$\alpha_j=(a_{0j},a_{1j},a_{2j},a_{3j},\ldots\ldots,a_{m-1j})$

或 α_i 是一个行向量的线性表

$\alpha_i=(a_{i0},a_{i1},a_{i2},a_{i3},\ldots\ldots,a_{in-1})$

二、数组的基本操作

1　读元素操作

2　写元素操作

操作方法根据其存储结构决定

5．2　数组的顺序存贮结构

一维数组在内存中的存放很简单，只要顺序存放在连续的内存单元即可。

二维数组，如何用顺序结构表示？内存地址是一维的，而数组是二维的，要将二维数组挤入一维的地址中，有两个策略：

以行为主序（C语言使用）

以列为主序

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

设A是一个具有m 行n列的元素的二维数组:

$$A_{m\times n}=\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & \cdots & a_{1n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-10} & a_{m-11} & \cdots & a_{m-1n-1} \end{pmatrix}$$

以行为主序的方式:

0	1	.....	n-1	n	n+1	.....	2n-1	.....	mn-1
a_{00}	a_{01}	.....	a_{0n-1}	a_{10}	a_{11}	.....	a_{1n-1}	.....	a_{m-1n-1}

以列为主序的方式:

0	1	.....	m-1	m	m+1	.....	2m-1	.....	nm-1
a_{00}	a_{10}	.....	a_{m-10}	a_{01}	a_{11}	.....	a_{m-11}	.....	a_{m-1n-1}

数组元素存储地址的计算

假设二维数组A每个元素占用s 个存储单元， $Loc(a_{ij})$为元素a_{ij}的存储地址，$Loc(a_{00})$ 是a_{00}存储位置, 也是二维数组A的基址。

若以行序为主序的方式存储二维数组，则元素a_{ij} 的存储位置可由下式确定： $Loc(a_{ij}) = Loc(a_{00}) + (n \times i + j) \times s$

若以列序为主序的方式存储二维数组，则元素a_{ij} 的存储位置可由下式确定： $Loc(a_{ij}) = Loc(a_{00}) + (m \times j + i) \times s$

一般在程序设计过程中，一维数组和二维数组使用较普遍，超过二维以上的多维数组使用相对较少，对于高维数组的顺序存储方法，可以将二维的情形加以推广便能够得到。

5.3 矩阵的压缩存储

一　特殊矩阵的压缩存储

二　稀疏矩阵的压缩存储

1　三元组表的存储结构

2　十字链表的存储结构

5．3　矩阵的压缩存储

矩阵是许多科学与工程计算问题中常常涉及到的一种运算对象。一个m行n列的矩阵是一平面阵列，有m×n个元素。可以对矩阵作加、减、乘等运算。

只有少数程序设计语言提供了矩阵运算。通常程序员是用二维数组存储矩阵。由于这种存储方法可以随机地访问矩阵的每个元素，因而能较为容易地实现矩阵的各种运算。

$$A_{m\times n}=\begin{pmatrix} a_{00} & a_{01} & & a_{0n-1} \\ a_{10} & a_{11} & & a_{1n-1} \\ & & & \\ & & & \\ a_{m-10} & a_{m-11} & & a_{m-1n-1} \end{pmatrix}$$

5．3　矩阵的压缩存储

应用中常遇到一些阶数很高的矩阵，矩阵中有许多值相同的元素或零元素。二维数组存储矩阵会浪费很多的存储单元。

例如，设一个1000 × 1000的矩阵中有800个非零元素，若用二维数组存储需要10^6个存储单元。因此，需要使用高效的存储方法，减少数据的存储量，即对原矩阵，根据数据分布特征进行压缩存储。

本章将讨论两类矩阵的压缩存储：

1　特殊矩阵的压缩存储

2　稀疏矩阵的压缩存储

$$A_{m\times n}=\begin{pmatrix} a_{00} & a_{01} & & a_{0n-1} \\ a_{10} & a_{11} & & a_{1n-1} \\ & & & \\ & & & \\ a_{m-10} & a_{m-11} & & a_{m-1n-1} \end{pmatrix}$$

一　特殊矩阵

值相同元素或者零元素分布有一定规律的矩阵称为特殊矩阵

例　对称矩阵、上（下）三角矩阵都是特殊矩阵

$$\begin{pmatrix} a_{00} & a_{10} & a_{20} & \cdots & a_{n-10} \\ a_{10} & a_{11} & a_{21} & & \\ & & & & \\ a_{n-10} & a_{n-11} & & & a_{n-1\ n-1} \end{pmatrix} \qquad \begin{pmatrix} a_{00} & a_{01} & & & a_{0n-1} \\ 0 & a_{11} & & & a_{1n-1} \\ & & & & \\ 0 & 0 & & 0 & a_{n-1\ n-1} \end{pmatrix}$$

5．3　矩阵的压缩存储

特殊矩阵压缩存储　(以对称矩阵为例)

对称矩阵是满足下面条件的n 阶矩阵: $a_{ij}= a_{ji}$　　$0\le i, j\le n-1$

$$\begin{pmatrix} a_{00} & a_{01} & & a_{0n-1} \\ a_{10} & a_{11} & & a_{1n-1} \\ & & & \\ a_{n-10} & a_{n-11} & & a_{n-1n-1} \end{pmatrix} \qquad \begin{pmatrix} a_{00} & & & \\ a_{10} & a_{11} & & \\ a_{20} & a_{21} & a_{22} & \\ & & & \\ a_{n-10} & a_{n-11} & & a_{n-1n-1} \end{pmatrix}$$

a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	.....	a_{n-10}	a_{n-11}	.....	a_{n-1n-1}		
k= 0	1	2	3	4	5					n(n+1)/2-1		

对称矩阵元素可以只存储下三角部分，共需 n(n+1)/2 个单元的空间

(三角矩阵的存储方式类似）

5．3　矩阵的压缩存储

以一维数组sa[ ]作为n 阶对称矩阵A的存储结构，A中任意一元素 a_{ij} 与它的存储位置 sa[k] 之间存在着如下对应关系：

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

例如，a_{53} 在 sa[ ]中的存储位置是：k=5*(5+1)/2+3=18

sa[18]= a_{53}

压缩存储的对称矩阵的取值算法

```
int get_M(int i, int j)
{ if(i>=j) return(sa[i*(i+1)/2+j])
    else return(sa[j*(j+1)/2+i]);
}
```

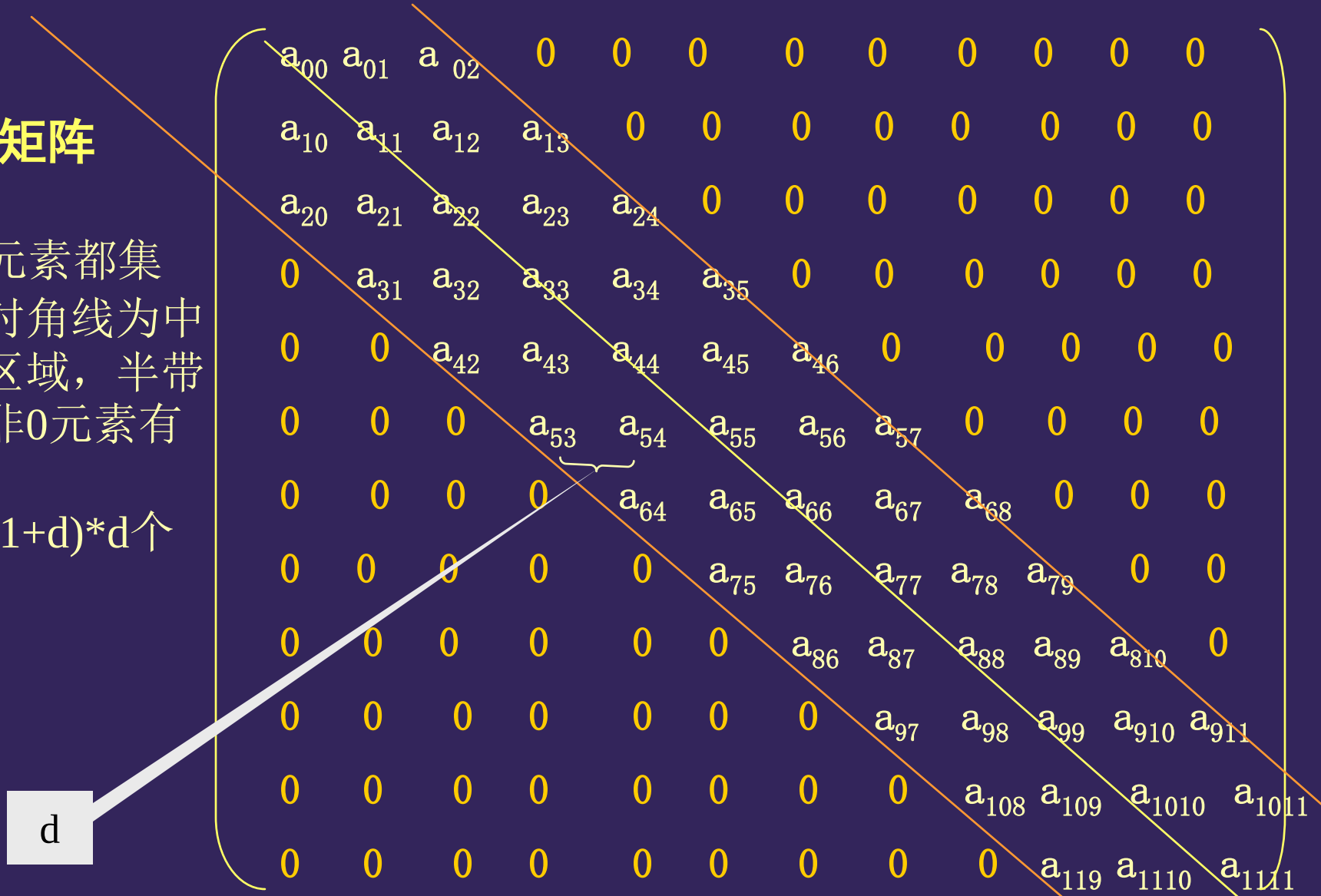
压缩存储的对称矩阵的 赋值算法

```
void assign_M(int i, int j, int value)
{ if(i>=j) sa[i*(i+1)/2+j]=value;
    else sa[j*(j+1)/2+i]=value;
}
```

带状矩阵

所有非0元素都集中在以主对角线为中心的带状区域，半带宽为d时，非0元素有

(2d+1)*n-(1+d)*d个

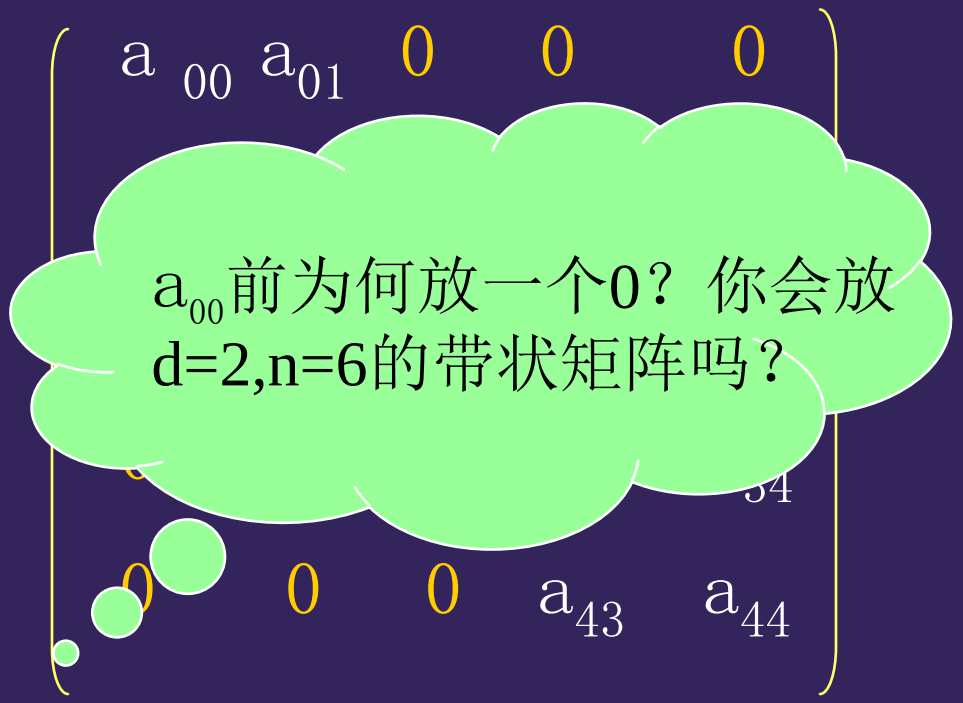


为计算方便，认为每一行都有2d+1个非0元素，若少则用0补足，所以，存放矩阵的数组sa[ ]有

n(2d+1) 个元素

数组元素sa[k]与矩阵元素a_{ij}之间有关系

k= i*(2d+1)+d+(j-i)



K=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	a_{43}	a_{44}	0	

压缩存储的带状矩阵的取值算法

```
int get_Md(int i, int j)
{ if(abs(i-j)<=d）  return(sa[i*(2*d+1)+d+(j-i)]);
    else return(0);
}
```

压缩存储的 带状矩阵的 赋值算法

```
void assign_Md(int i, int j, int value)
{ if(abs(i-j)<=d）  sa[i*(i+1)/2+j]=value;
}
```

二　稀疏矩阵

1　什么是稀疏矩阵

如何进行稀疏矩阵的压缩存储？

有较多值相同元素或较多零元素，且[illegible]分布没有一定规律的矩阵称为稀疏矩阵。

例

$$A = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

A有42（6×7）个元素

有8个非零元素

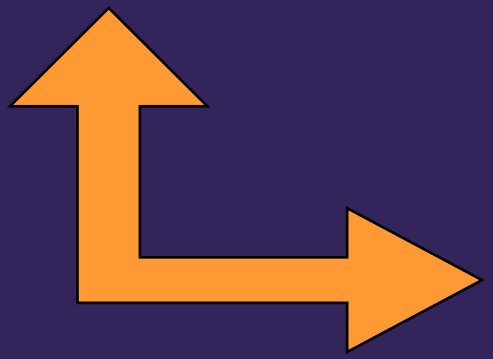
2 稀疏矩阵的压缩存储（只讨论有较多零元素矩阵的压缩存储）

表示非零元的三元组

1）三元组表（**i, j, a_{ij})

A=((0,1,12), (0,2,9), (2,0,-3),(2,5,14),
(3,2,24), (4,1,18), (5,0,15), (5,3,-7))

加上行、列数6，7



$$A = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

2）三元组顺序表

假设以顺序存储结构来表示三元组表，则可得稀疏矩阵的一种压缩存储方式——我们称之为三元组顺序表。

稀疏矩阵的三元组顺序表的类型定义

用于存储非零元三元组的结构

```
struct node
{ int  row,col;          // 非零元的行下标和列下标
       int value;          //非零元值
}; typedef struct node NODE;
NODE ma[MAX];
```

ma[0]用于存储矩阵行数、列数、非零元个数

A的三元组顺序表图示

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

	row	col	value
0	6	7	8
1	0	1	12
2	0	2	9
3	2	0	-3
4	2	5	14
5	3	2	24
6	4	1	18
7	5	0	15
8	5	3	-7

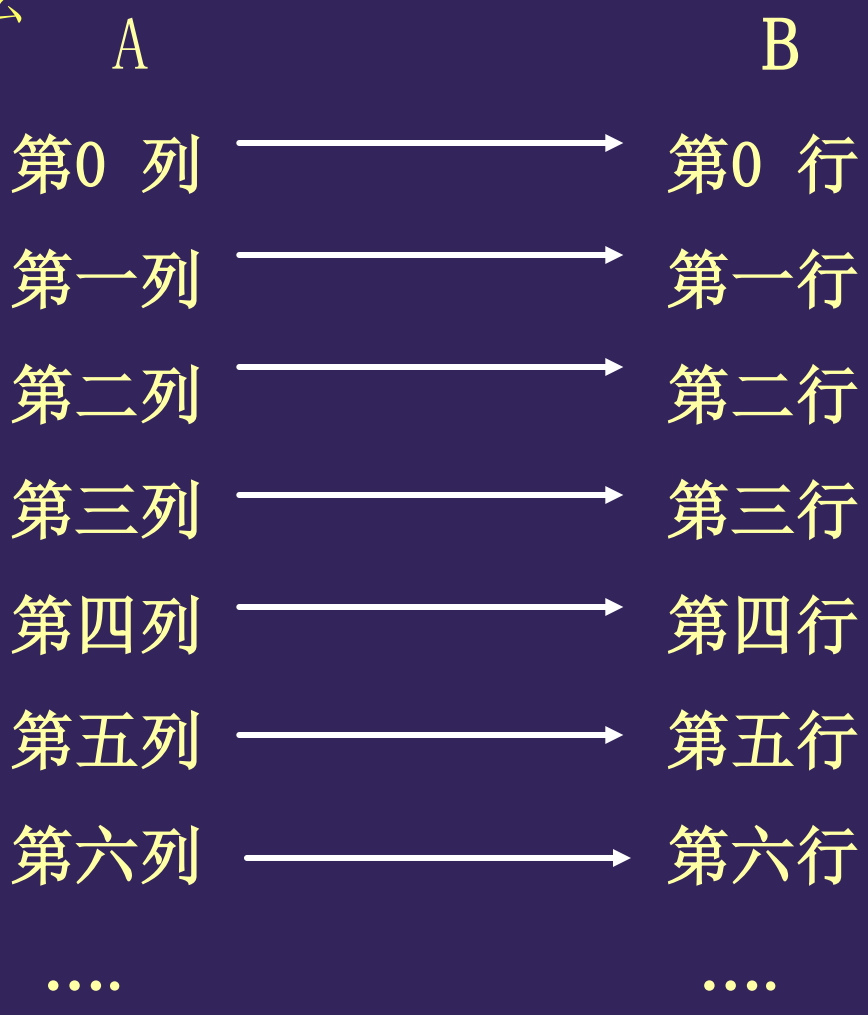
例如 **ma[1].row=0,　ma[1].col=1,　ma[1].value=12**

3） 转置运算算法

转置运算是一种最常用的矩阵运算。对于一个 m 行 n 列的矩阵 **A**，它的转置矩阵 **B** 是一个n行m列的矩阵。例如，下图中的矩阵A和B互为转置矩阵。

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix} \qquad B=\begin{pmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

转置运算算法



矩阵 A

	row	col	value
0	6	7	8
1	0	1	12
2	0	2	9
3	2	0	-3
4	2	5	14
5	3	2	24
6	4	1	18
7	5	0	15
8	5	3	-7

矩阵 B

	row	colv	alue
0	7	6	8
1	0	2	-3
2	0	5	15
3	1	0	12
4	1	4	18
5	2	0	9
6	2	3	24
7	3	5	-7
8	5	2	14

分析：

（1）将矩阵的行列数的值交换

（2）将每一个三元组的 i 和 j相互调换

（3）重排三元组之间的次序

转置运算算法

按照A的列序来进行转换的基本思想

对 ma[ ] 从头至尾扫描：

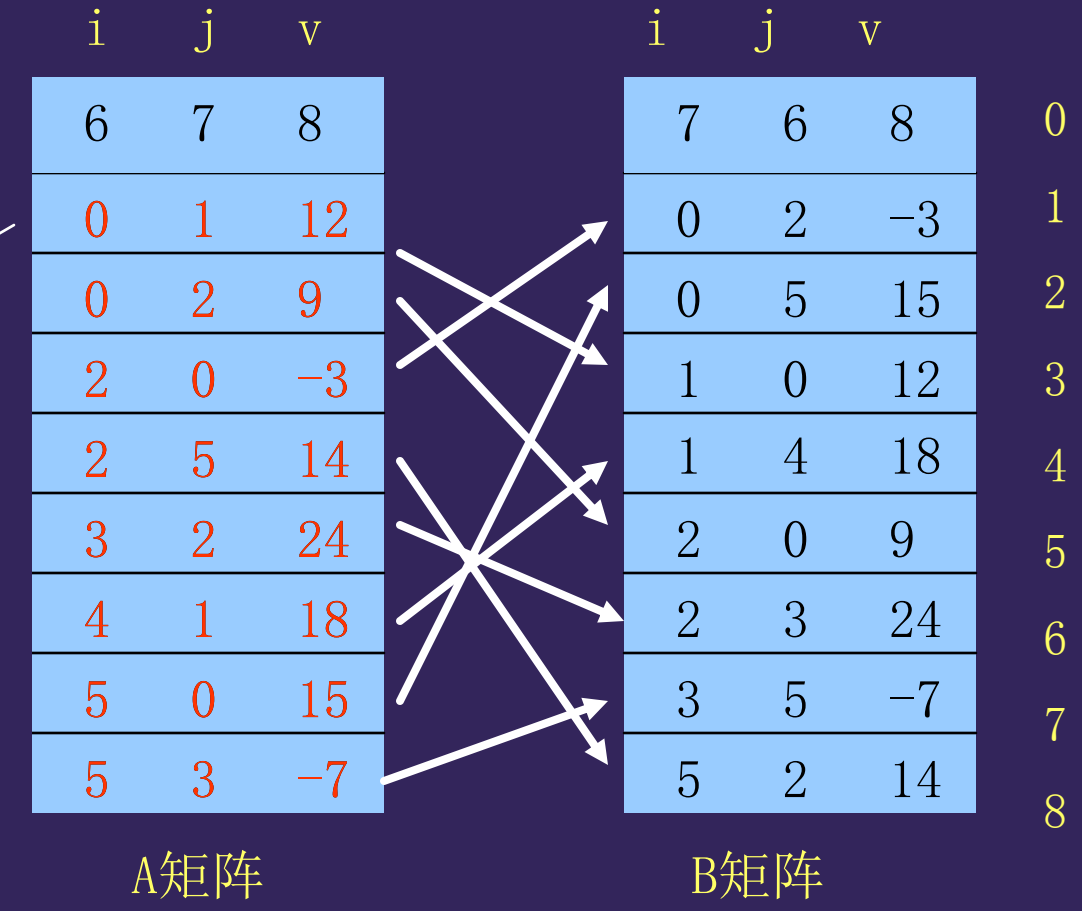
第一次扫描时，将 ma[ ]中列号为0的所有元组交换行列值后，依次赋值到 mb[ ]中

第二次扫描时，将 ma[ ]中列号为1的所有元组交换行列值后，依次赋值到 mb[ ] 中

依此类推，直至将 ma[ ]的所有三元组赋值到 mb[ ]中

转置运算算法图示

第六次扫描查找第5列元素



对A六次扫描完成转置运算

转置算法:采用三元组表存储表示，求稀疏矩阵A的转置矩阵B

```
int transpose (NODE ma[ ], NODE mb[ ])
  {  int i, j, k;
     if( ma[0].value= =0)return(0);
    mb[0].row=ma[0].col; mb[0].col=ma[0].row;
     mb[0].value=ma[0].value; k=1;  // k为当前三元组在mb[ ]存储位置(下标)
        for (i=0; i<ma[0].col;  i++)     // 找 ma[ ]中第  i  列所有非0元素
             for (j=1;  j<=ma[0].value;  j++)      //j为扫描ma[ ]的“指示器”
                                                   //j“指向”三元组称为当前三元组
                 if (ma[j].col= =i)
                  { mb[k].row =ma[j].col; mb[k].col=ma[j].row;
                    mb[k].value=ma[j].value; k++;}
      return(1);
}                                  \\ 算法5.7
```

时间复杂度分析

算法的基本操作为将 ma[ ] 中的三元组赋值到 mb[ ]，是在两个循环中完成的，故算法的时间复杂度为 **O(n × t)**，其中**n** 为A矩阵的列数, **t**为非0元素个数。

当非零元的个数 **t** 和矩阵元素个数m×n 同数量级时，即 **t ≈m ×n**，转置运算算法的时间复杂度为**O**（ n ×m × n）。由此可见：在这种情况下，用三元组顺序表存储矩阵，虽然可能节省了存储空间，但时间复杂度提高了，因此算法仅适于 **t <<** m × n 的情况。

该算法效率不高的原因是：对为实现 A 到 B 的转置，该算法对 ma[ ] 进行了多次扫描。其特点是：以B矩阵的三元组为中心，在 A 矩阵的三元组中通盘查找合适的结点置入 mb[k] 中

能否在对 ma[ ] 一次扫描的过程中，完成 A 到 B 的转置？

下面介绍的转置运算算法称为 **快速转置算法**

方法是：以A矩阵的三元组为中心，依次取出 ma[ ] 中的每一个三元组，交换行列后，直接将其写入mb[ ]合适的位置中

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

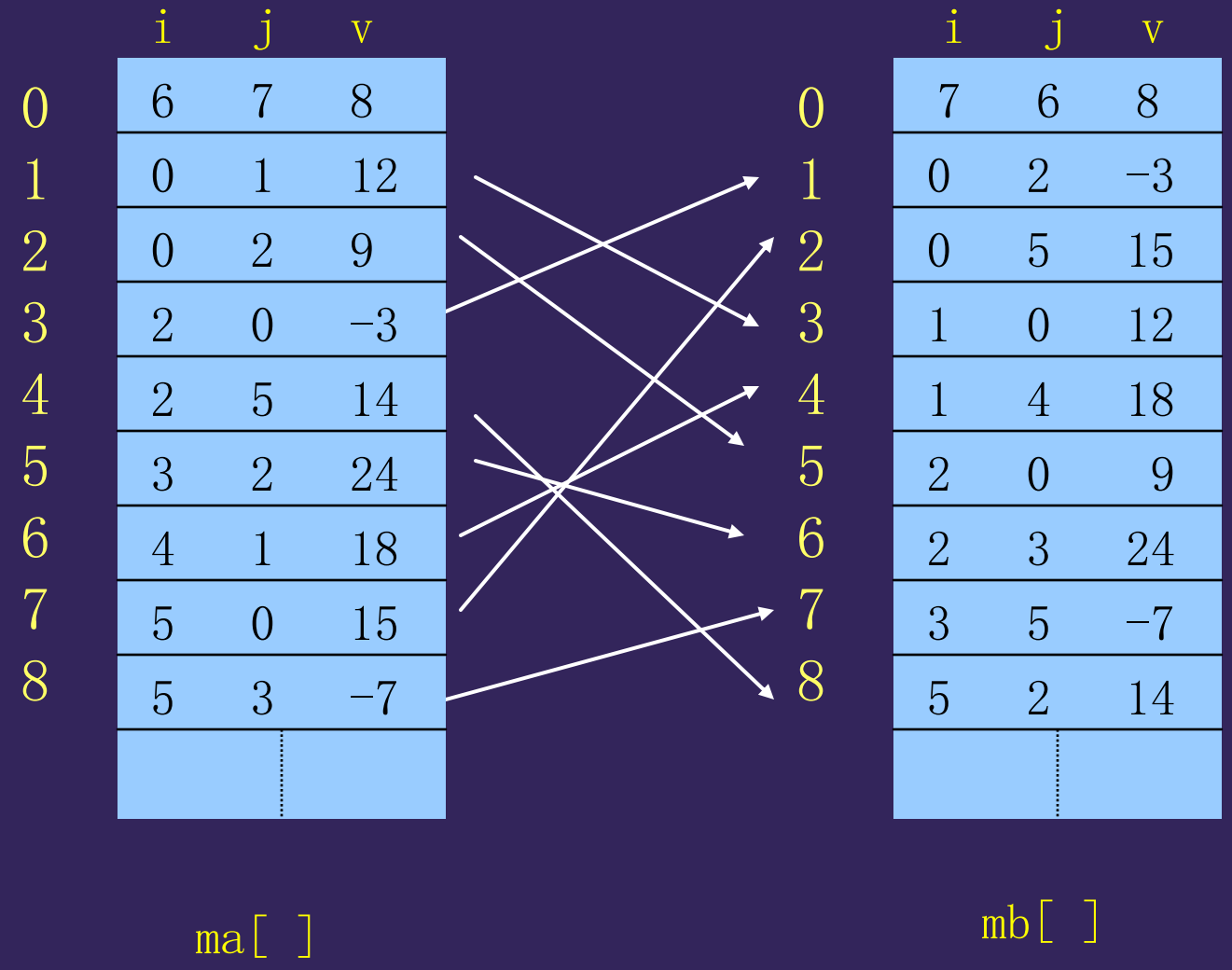
如果能先求得A各列
第一个非零元三元组在 mb[ ] 中的位置
就能在对ma[ ]一次扫描的过程中，
完成A到B 的转置

	row	col	value
0	6	7	8
1	0	1	12
2	0	2	9
3	2	0	-3
4	2	5	14
5	3	2	24
6	4	1	18
7	5	0	15
8	5	3	-7

row	col	value
7	6	8
0	2	-3
0	5	15
1	0	12
1	4	18
2	0	9
2	3	24
3	5	-7
5	2	14

各列第一个非零元三元组按先前求出的位置，放至mb[ ]中

各列后续的非零元三元组，顺序放到对应列元素的后面



引入辅助数组 **num[ ]**、**pos[ ]**

num[col]：存储 A 矩阵第 col 列非零元个数

pos[col]：存储第 col 列第一个非零元三元组在 mb[ ] 中的位置

pos[col] 的计算方法：

$$\begin{cases} pos[0]=1 \\ pos[j]=pos[j-1]+num[j-1] \qquad 1 \le j \le n-1 \end{cases}$$

例 矩阵 A

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

快速转置算法主要步骤：

1　　求 A 中各列非零元个数num[ ];
2　　求 A 中各列第一个非零元在 mb[ ] 中的下标 pos[ ];
3　　对 ma[ ] 进行一次扫描， 遇到 col 列的第一个非零元三元组时，按 pos[col]的位置，将其放至 mb[ ] 中，当再次遇到 col 列的非零元三元组时，只须顺序放到 col 列元素的后面;

求各列非零元个数

求各列第1个非零元在mb[]中位置

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

第0列第一个非零元在mb[]中的位置

第5 列第一个非零元在mb[]中的位置

扫描ma[ ]实现A到B的转置

	i	j	v
1	0	1	12
2	0	2	9
3	2	0	-3
4	2	5	14
5	3	2	24
6	4	1	18
7	5	0	15
8	5	3	-7

A 矩阵

	i	j	v
1	0	2	-3
2	0	5	15
3	1	0	12
4	1	4	18
5	2	0	9
6	2	3	24
7	3	5	-7
8	5	2	14

B 矩阵

快速转置算法图示

快速转置算法

```
int transpose1 (NODE ma[ ], NODE mb[ ] )
{   int i, j, k,  num[MAX], pos[MAX];
    if (ma[0].value= =0)return(0);
    mb[0].row=ma[0].col; mb[0].col=ma[0].row;
    mb[0].value=ma[0].value;
    for (i=0; i<ma[0].col;  i++) num[i]=0;
    for (i=1;  i<=ma[0].value;  i++) num[ma[i].col]++; // 统计第i 列非0元个数
    for( pos[0]=1, i=1; i<ma[0].col; i++)
          pos[i]=pos[i-1]+num[i-1];              //计算 pos[ ] 数组
    for (i=1; i<=ma[0].value; i++)               // 快速转置
      { j=ma[i].col;  k=pos[j];  mb[k].row =ma[i].col;
        mb[k].col=ma[i].row; mb[k].value=ma[i].value; pos[j]++;}
     return(1);
}
```

时间复杂度分析

该算法利用两个辅助数组num[ ]、pos[ ]，实现了对 ma[ ]一次扫描完成A 到 B 的转置。

从时间上看，算法中有四个并列的单循环，循环次数分别为n、t、n和 t，因而总的时间复杂度为O（n+t)，在A的非零元个数 t 和m×n 同等数量级时，其时间复杂度为O(m×n)，均比转置算法5.7的时间复杂度小。

空间复杂度分析

两个辅助数组占用n+n个单元，空间复杂度为O(n)。

十字链表的存储结构

以三元组表示的稀疏矩阵，在运算中，若非0元素的位置发生变化，会引起数组元素的频繁移动。为解决这个问题，采用十字链表的存储结构

在十字链表中，表示非0元素的结点除了三元组，还有两个指针域：

向下域（down）　链接同一列下一个非0元素

向右域(right)　链接同一行下一个非0元素

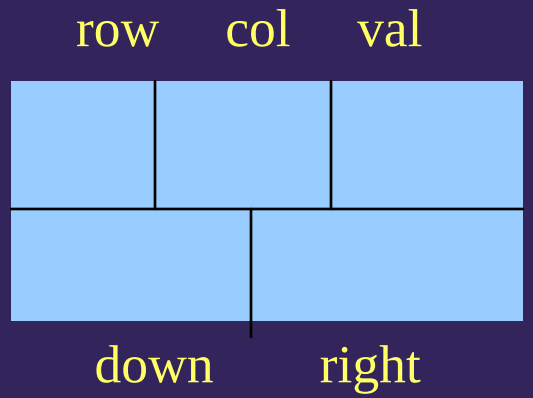
稀疏矩阵中同一行的非0元素结点通过向右域，链接成一个带头结点的行循环链表

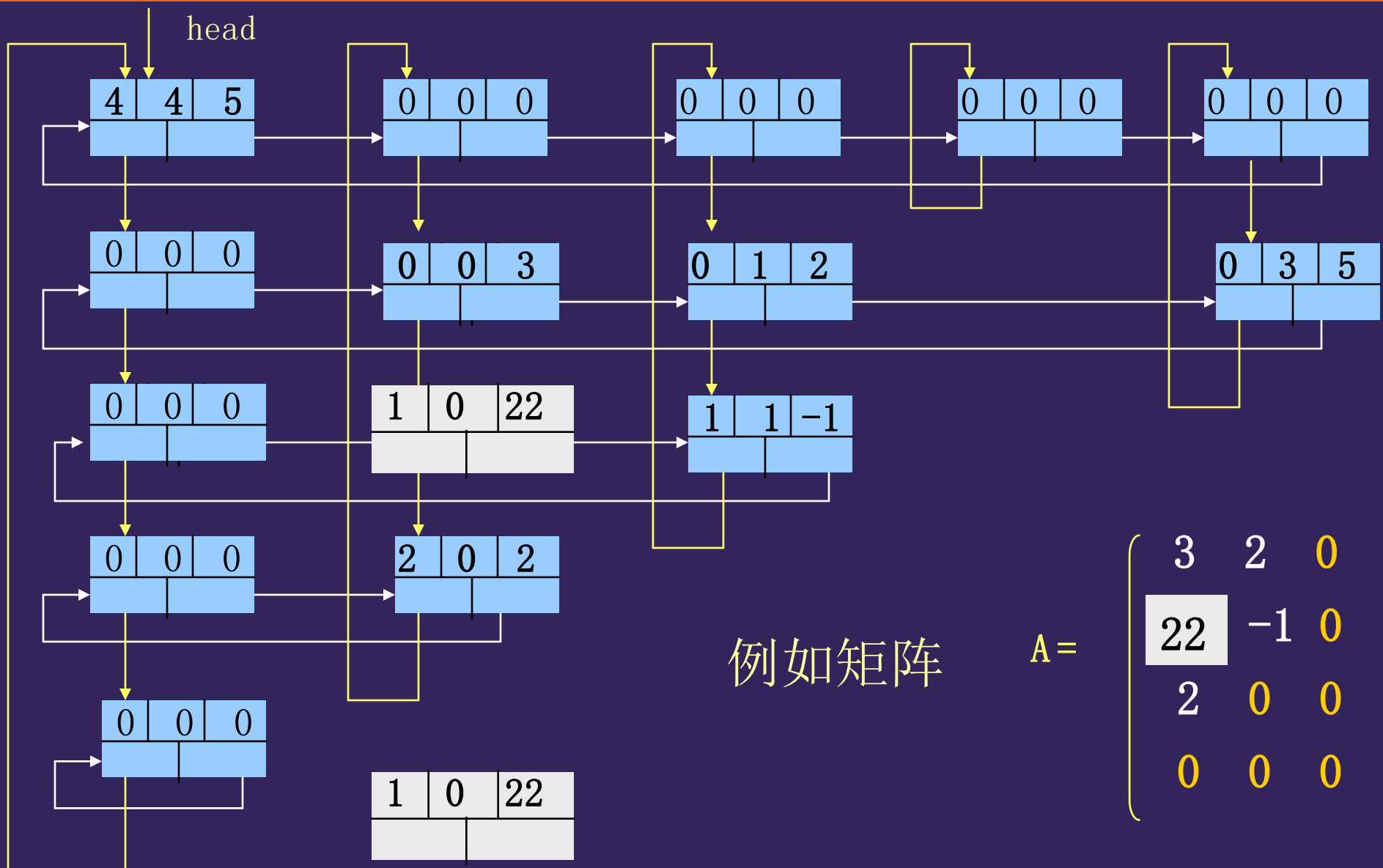
同一列的非0元素结点通过向下域，链接成一个带头结点的列循环链表

十字链表的存储结构

结点的数据结构如下：

```
struct node
{  int row, col, val;
   struct node *down, *right;
};
typedef struct node NODE;
```



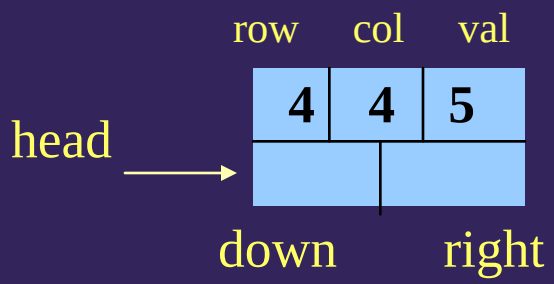


例如矩阵　A=

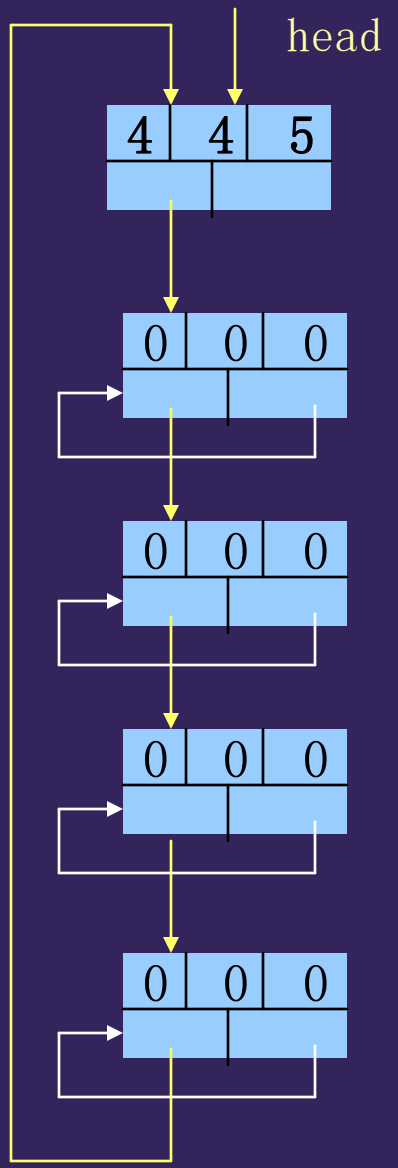
$$\begin{pmatrix} 3 & 2 & 0 & 5 \\ 22 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

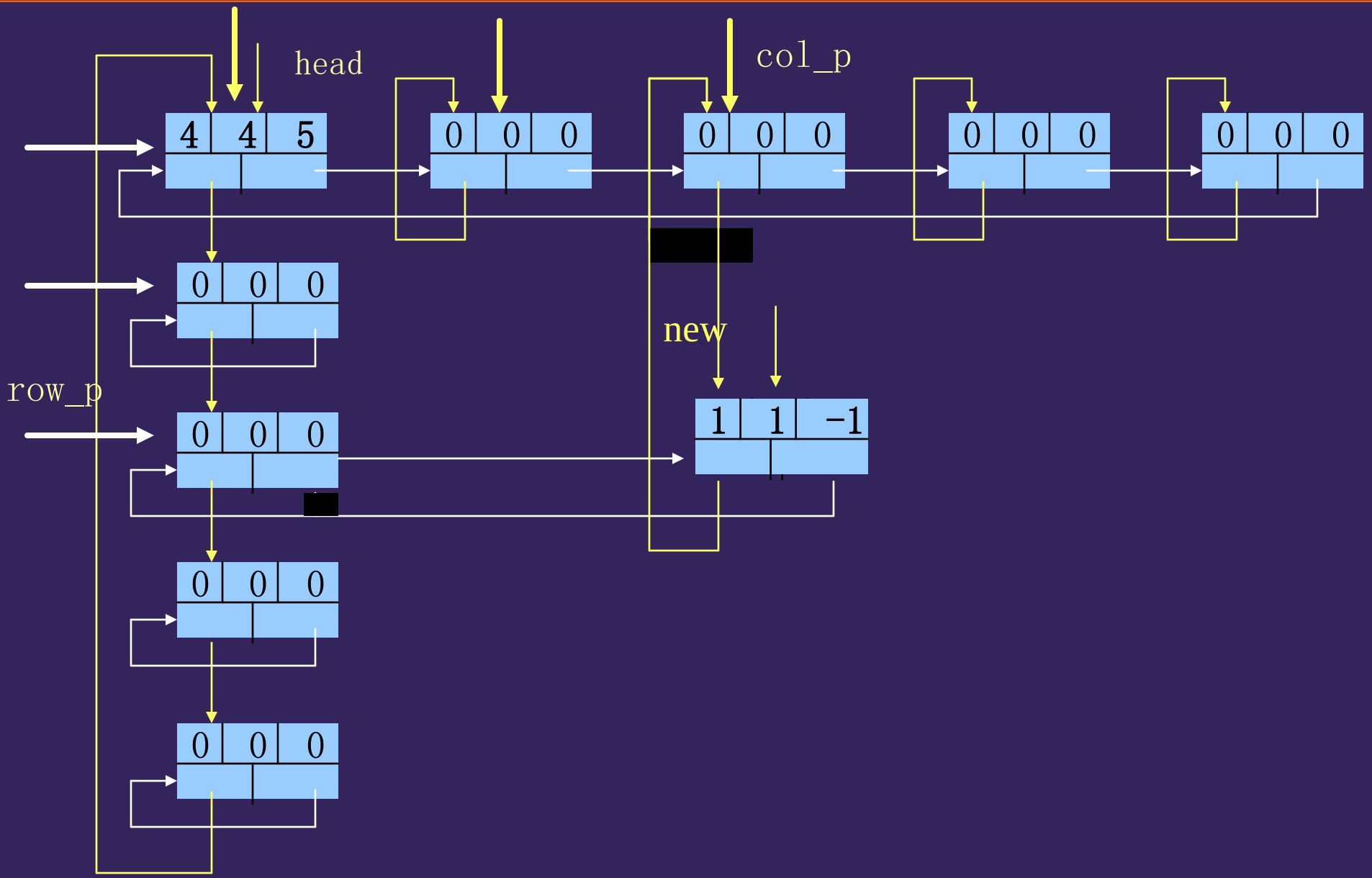
建立十字链表算法5.9

```
NODE *create()
  { NODE *head,*new,*pre,*p,*row_p,*col_p;
    int i,count;
    head=(NODE *)malloc(sizeof(NODE));
    scanf("%d,%d,%d\n",&head->row,&head->col,&head->val);
```



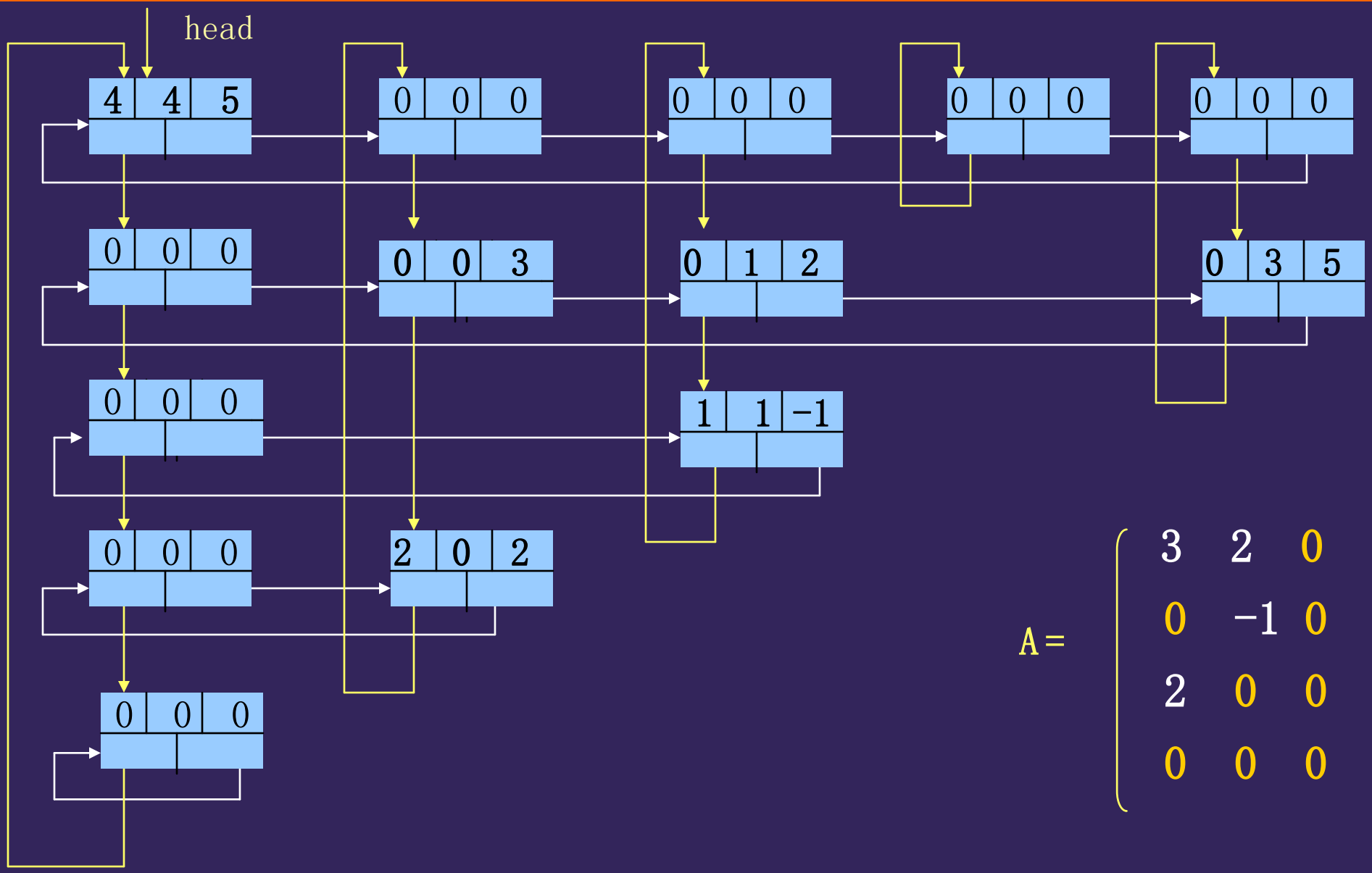
```
for(pre=head,i=0;i<head->row;i++)
  {p=(NODE *)malloc(sizeof(NODE));
    p->val=p->row=p->col=0;
    p->right=p;pre->down=p;pre=p;
   } p->down=head;
  for(pre=head,i=0;i<head->col;i++)
   { p=(NODE *)malloc(sizeof(NODE));
      p->val=p->row=p->col=0;
      p->down=p;pre->right=p;pre=p;
    } p->right=head;
```




head
col_p
4 4 5
0 0 0
0 0 0
0 0 0
0 0 0
new
row_p
1 1 -1
0 0 0
0 0 0
0 0 0
0 0 0

```
count=0;
while(count<head->val)
 { count++;  new=(NODE *)malloc(sizeof(NODE));
    scanf("%d,%d,%d\n",&new->row,&new->col,&new->val);
     for(row_p=head,i=0;i<=new->row;i++)
         row_p=row_p->down;p=row_p;
         while(p->right!=row_p&&p->right->col<new->col)
             p=p->right;
          new->right=p->right;p->right=new;
```

```
for(col_p=head,i=0;i<=new->col;i++)
    col_p=col_p->right;p=col_p;
 while(p->down!=col_p&&p->down->row<new->row)
    p=p->down;
   new->down=p->down;p->down=new;
}
return(head);
}
```



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

小 结

1 矩阵压缩存储是指为多个值相同的元素分配一个存储空间，对零元素不分配存储空间；

2 特殊矩阵的压缩存储是根据元素的分布规律，确定元素的存储位置与元素在矩阵中的位置的对应关系；

3 稀疏矩阵的压缩存储除了要保存非零元素的值外，还要保存非零元素在矩阵中的位置；

5.4 广义表

5.3.1 广义表的概念

5.3.2 广义表的存储结构

5.3.2 广义表的基本操作

5.4.1 广义表的概念

1 什么是广义表

广义表也称为列表，是线性表的一种扩展，也是数据元素的有限序列。记作：$LS= (d_0, d_1, d_2, \ldots\ldots d_{n-1})$。其中$d_i$既可以是单个元素，也可以是广义表。

说明

1）广义表的定义是一个递归定义，因为在描述广义表时又用到了广义表；

2）在线性表中数据元素是单个元素，而在广义表中， 元素可以是单个元素, 称为单元素(原子)，也可以是广义表，称为广义表的子表；

3）n 是广义表长度；

4）下面是一些广义表的例子；

A = ( ) 空表，表长为0；

B = (a,(b,c,d)) B的表长为2，两个元素分别为 a 和子表（b,c,d);

C = (e) C中只有一个元素e，表长为1；

D = (A,B,C,f) D 的表长为4，它的前三个元素 A，B，C 广义表,第四个是单元素；

E=(a ,E) 递归表.

5）若广义表不空，则可分成表头和表尾，反之，一对表头和表尾可唯一确定广义表

对非空广义表：称第一个元素为L的表头，其余元素组成的表称为LS的表尾；

B = (a,(b,c,d)) 表头： a 表尾 ((b,c,d))

即 HEAD（B）=a, TAIL(B)=((b,c,d)),

C = (e) 表头： e 表尾 ( )

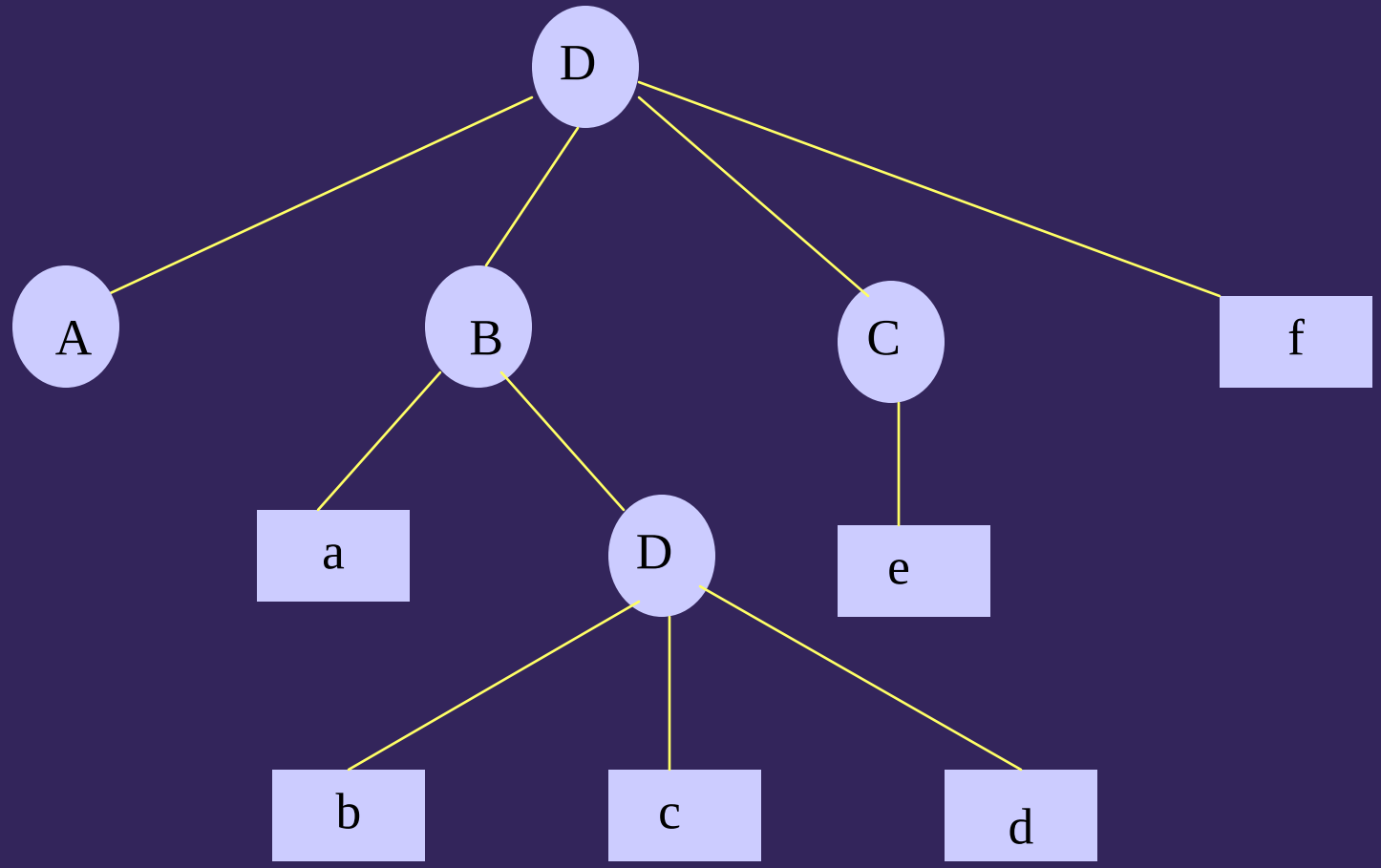
D = (A,B,C,f) 表头： A 表尾 (B,C,f)

运算可以嵌套，如：

HEAD（TAIL（B））=b, TAIL(TAIL(B))=(c, d) 。

5．4 广义表

广义表的元素之间除了存在次序关系外，还存在层次关系。如：



2 广义表的基本操作

1） 创建广义表L;
2） 销毁广义表L;
3） 已有广义表L，由L复制得到广义表T;
4） 求广义表L的长度;
5） 求广义表L的深度;
6） 判广义表L是否为空;
7） 取广义表L的表头;
8） 取广义表L的表尾;
9） 在L中插入元素作为L的第i个元素;
10）删除广义表L的第i个元素;
11）遍历广义表L，用函数 traverse( )处理每个元素;

5.4.2 广义表的存储结构

由于广义表中数据元素可以具有不同结构，故难以用顺序结构表示广义表。通常采用链表存储方式

如何设定链表结点？广义表中的数据元素可能为单元素(原子)或子表，由此需要两种结点：一种是表结点，用以表示广义表；一种是单元素结点，用以表示单元素（原子）。

单元素结点

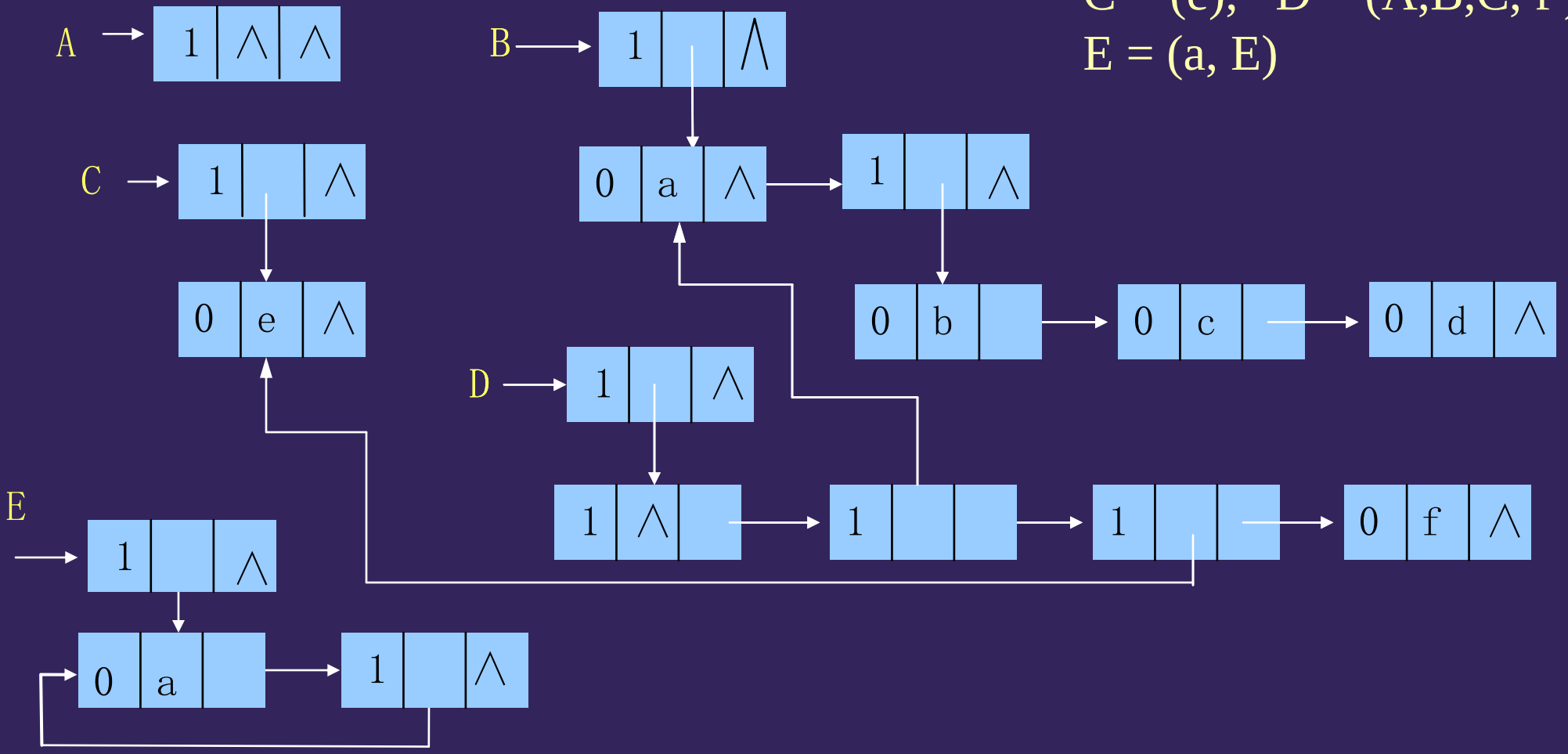
表结点

链表结点的类型定义如下：

```
struct  node
{   int  tag;          //标志域：用于区分原子结点和表结点
        union
        {  int  val;                        //原子结点的值域
           struct  node  *child;              //表结点的子表指针域，
         }content;
       struct  node  *next;      //    指向下一个元素的指针
 };typedef struct node NODE;
```

广义表的存储结构示例

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



5.3.4 广义表的基本操作的递归算法

广义表是递归结构，所以广义表的许多操作可以用递归算法实现，

1 求广义表的深度 int depth(NODE *head)

广义表 L= (α_1, α_2。。。α_n) 的深度=广义表中括号重数

例 A=（）B = (e) C = (a,(b,c,d)) D = (A,B,C)

depth(A)=1 depth(B)=1 depth(C)=2 depth(D)=3

广义表的深度=1+MAX(depth(α_i))

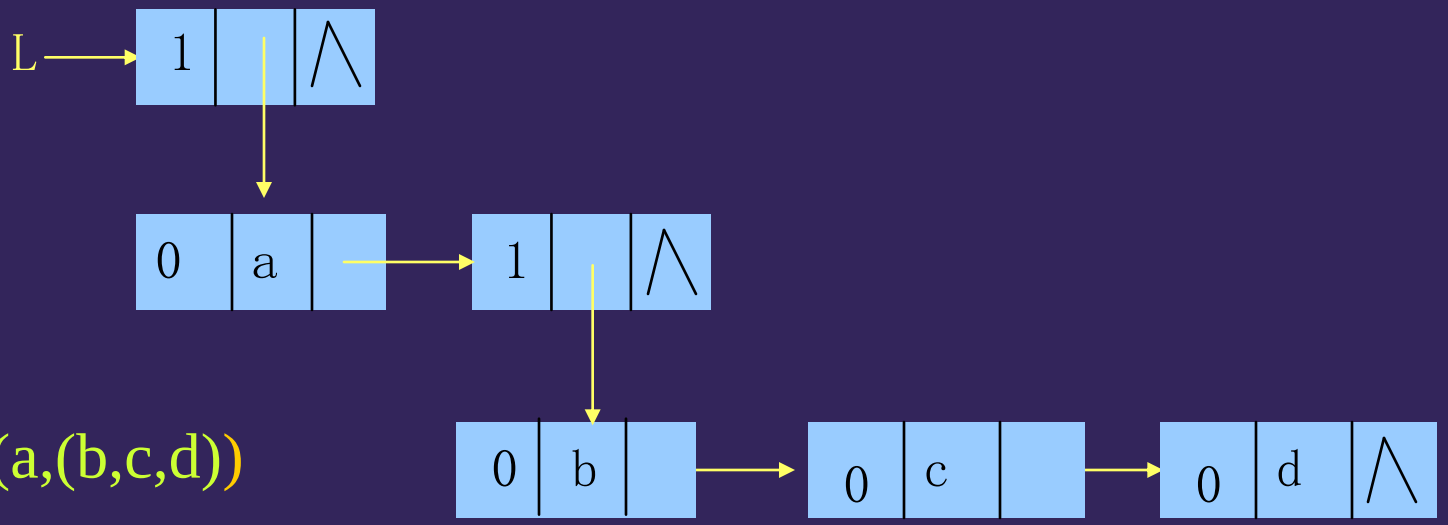
depth(α_i)的递归定义：

基本项：L 为空表，depth(α_i)=1

L 为原子结点指针，depth(α_i)=0

递归项：α_i 为非空表，

depth(L)= 1+MAX(depth (α_i))



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

求广义表L的深度。

```
int depth (NODE *head)
{  NODE  *p; int max, deth1;
     p=head;max=0;                          //空表深度为1
 while (p!=NULL)

  { if(p->tag= =0) depth1=0;       //原子深度为0
    else depth1=depth(p->content.child);     //求以p->content.child

       if(depth1>max)max=depth1;            //为头指针的子表深度
         p=p->next;
       }
         return(max+1);     //非空表的深度是各元素的深度的最大值加1

  }
```

建立广义表算法

```
NODE *create( )
 { NODE *p; char ch;
  ch=str[i];i++;
  switch(ch)
  { case '(':
      p=(NODE *)malloc(sizeof(NODE));p->tag=1;
      p->content.child=create( );
      p->next=create( );break;
```

```
case 'a': case 'b': case 'c': case 'd':case 'e':
     p=(NODE *)malloc(sizeof(NODE));p->tag=0;
     p->content.val=ch;
     p->next=create();break;
    case ',': p=create();break;
    case ')': case '\0':
    p=NULL;   }
    return(p);
  }
```

遍历广义表算法

```
void traverse(NODE *head)
{  NODE *p;
   for(p=head;p!=NULL;p=p->next)
   if(p->tag==1)
     {  printf("("); traverse(p->content.child);
        printf("\b),");  }
     else printf("%c,",p->content.val);
  }
```

小 结

1 广义表是数据元素的有限序列。其数据元素

可以单个元素，也可以是广义表；

2 广义表，通常采用链式存储结构

3 分解非空广义表的方法：

将非空广义表分解为表头和表尾两部分；

第五章结束