第五章 数组和广义表

本章学习另外两种特殊的线性表，数组和广义表。数组是一个数据元素的集合，元素之间具有线性关系，但是，元素可以参与多个线性关系（前面讲的都是参与一个）；广义表是一个数据元素的集合，元素之间具有线性关系，但其前驱后继可以是一般元素，也可以是一个表。

第五章　数组和广义表

5.1　数组

5.2　矩阵的压缩存储

5.3　广义表

5.1 数 组

基本内容

1 数组的概念及基本操作

2 数组的顺序存贮结构

学习要点

1 了解数组的逻辑结构

2 了解数组的两种存储结构；

3 掌握数组在以行为主序的方式存储时，数组元素地址的计算方法；

一 数组的概念

数组是我们十分熟悉的一种数据类型，几乎所有的程序设计语言都包含数组。本书在讨论各种数据结构的顺序存储分配时，也都是借用一维数组来描述它们的存储结构。这一章将从数据结构的角度，简单讨论数组的逻辑结构及其存储方式，另外书上还给出了数组的基本操作算法，如：数组的初始化，读取指定下标的数组元素算法，给指定下标的数组元素赋值等操作算法；

注:数组的基本操作算法,不是本课程要求掌握的内容

数组(Array)： 简单说是数据类型相同的一组数据元素的有序集合。元素在集合中的序是由一组称做“下标”的值确定的，一个数据元素称为一个数组元素。

一个数据元素的位置由多个下标值确定，即参与多个线性关系，每个关系上都有前驱、后继！即在每个关系中，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。

数组的维数： 确定一个数据元素在集合中位置的下标的个数。

注意:(1)数组中每个数据元素受多个线性关系制约，元素在每个线性关系上都有前驱、后继。(2)一个N 维数组N可以看作一个线性表，其每个数据元素是一个N -1 维的数组。

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

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

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}

此处未考虑第一个和最后一个元素

数组的基本操作

1 初始化操作 InitArray(&A, n, bound1, …, boundn)

功能：参数合法，构造数组A，并返回OK；

2 销毁操作 DestroyArray(&A) 功能：销毁数组A

3 读元素操作 Value(A, &e, index1, …, indexn)

功能：若指定下标不越界，读指定下标的元素，用e返回，并返回OK；

4 写元素操作 Assign(A, e, index1, …, indexn)

功能：若指定下标不越界，将e赋值给A指定的下标元素，并返回OK；

二 数组的顺序存贮结构

一般来说，数组一旦定义，其元素的个数和元素之间的相互关系不再发生变化，即对数组一般不进行插入和删除操作。因此，数组采用顺序存储结构是十分自然的事情

计算机的内存空间是一个一维结构，而二维以上的数组是多维结构。因此，用一组连续的存储单元存放数组元素，就有次序约定的问题。以二维数组为例，它有两种方式：一种是以行为主序的方式，另一种是以列序为主序的方式。行为主序方式是先存储数组的第一行元素，再存储第二行元素…；而列序为主序方式是先存储数组的第一列元素，再存储第二列元素…；在众多的程序设计语言中，以行序为主序方式的有PASCAL、COBOL、C及扩展BASIC等，以列序为主序方式的有FORTRAN语言。

设A是一个具有m 行n列的元素的二维数组（借助矩阵形式给出比较直观）如下：

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

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-10}	a_{m-11}	.....	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_{0n-1}	a_{1n-1}	……	a_{m-1n-1}	……	

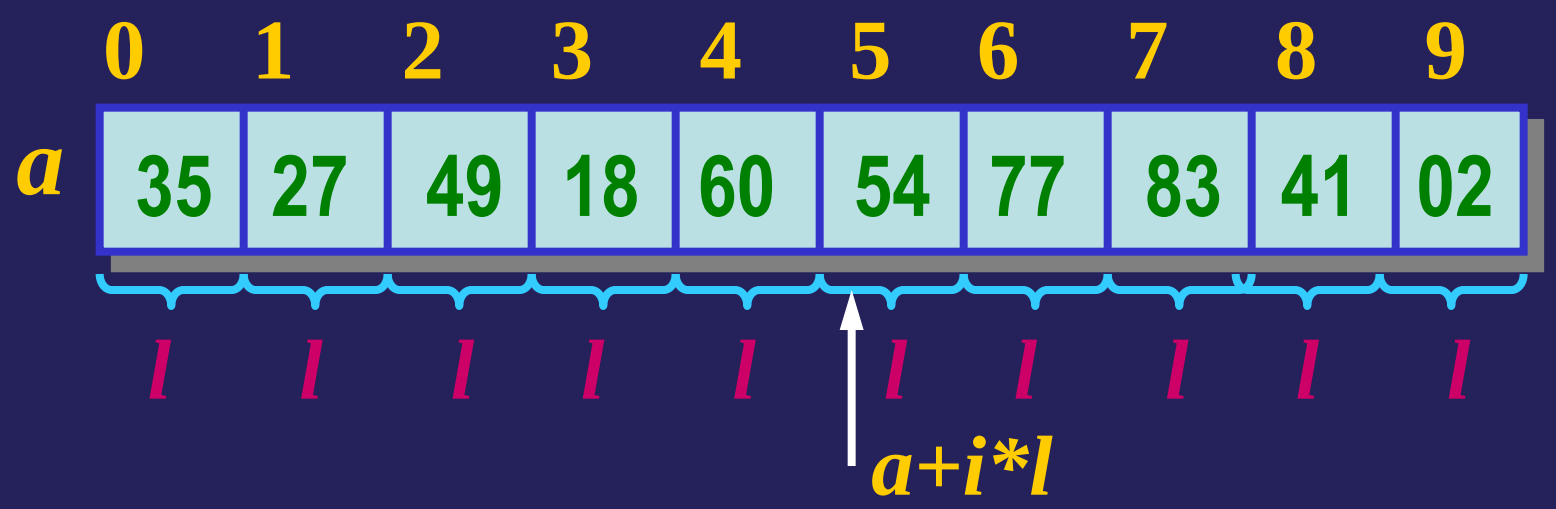
$$\begin{pmatrix} a_{00} & a_{01} & & a_{0n-1} \\ a_{10} & a_{11} & & a_{1n-1} \\ & & & \\ a_{m-1\ 0} & a_{m-1\ 1} & & a_{m-1n-1} \end{pmatrix}$$

数组元素存储地址的计算 $A_{m\times n}$

无论采用哪种存储方式，一旦确定了存储映象的首地址。数组中任意元素的存储地址都可以确定。

一维数组存储方式

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

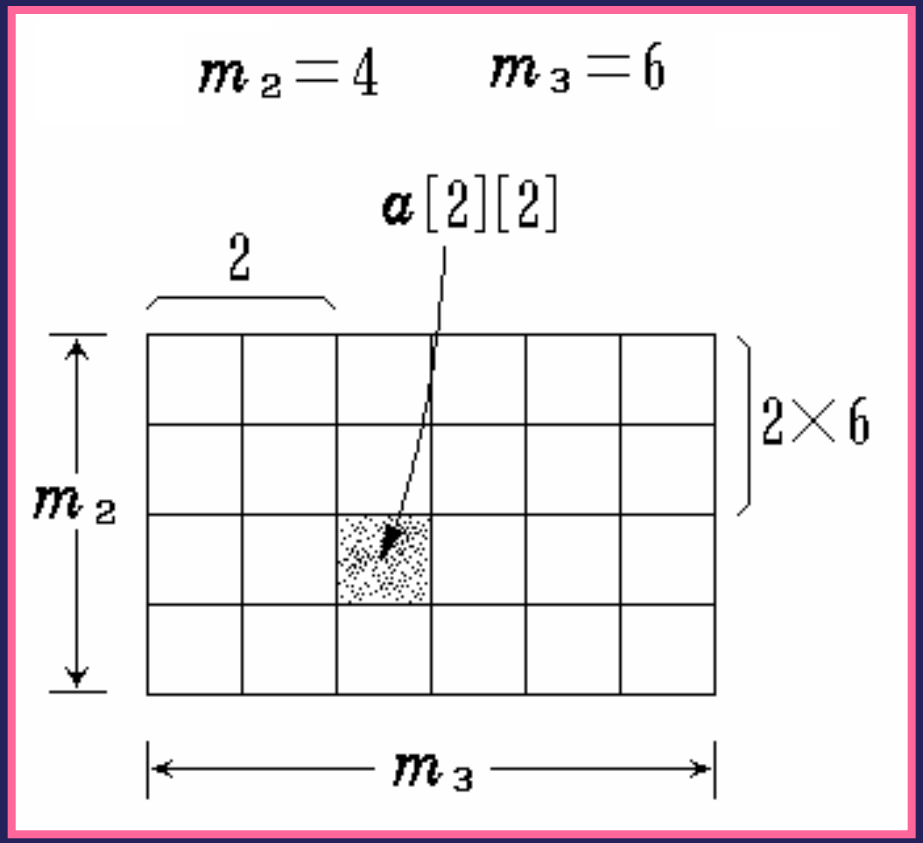


$$\mathrm{LOC}(i) = \mathrm{LOC}(i-1)+l = a+i*l$$

二维数组

假设二维数组A每个元素占用 **k** 个存储单元， **Loc(aij)** 为元素**aij** 的存储地址，**Loc(a00)** 是**a00**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**aij** 的存储位置可由下式确定：



Loc(aij) = Loc(a00) + （n× i+j) ×k

行向量	下表i
列向量	下表j

若以*列序为主序*的方式存储二维数组，则元素**aij** 的存储位置可由下式确定：

Loc(aij) = Loc(a00) + （m× j+i) ×k

三维数组

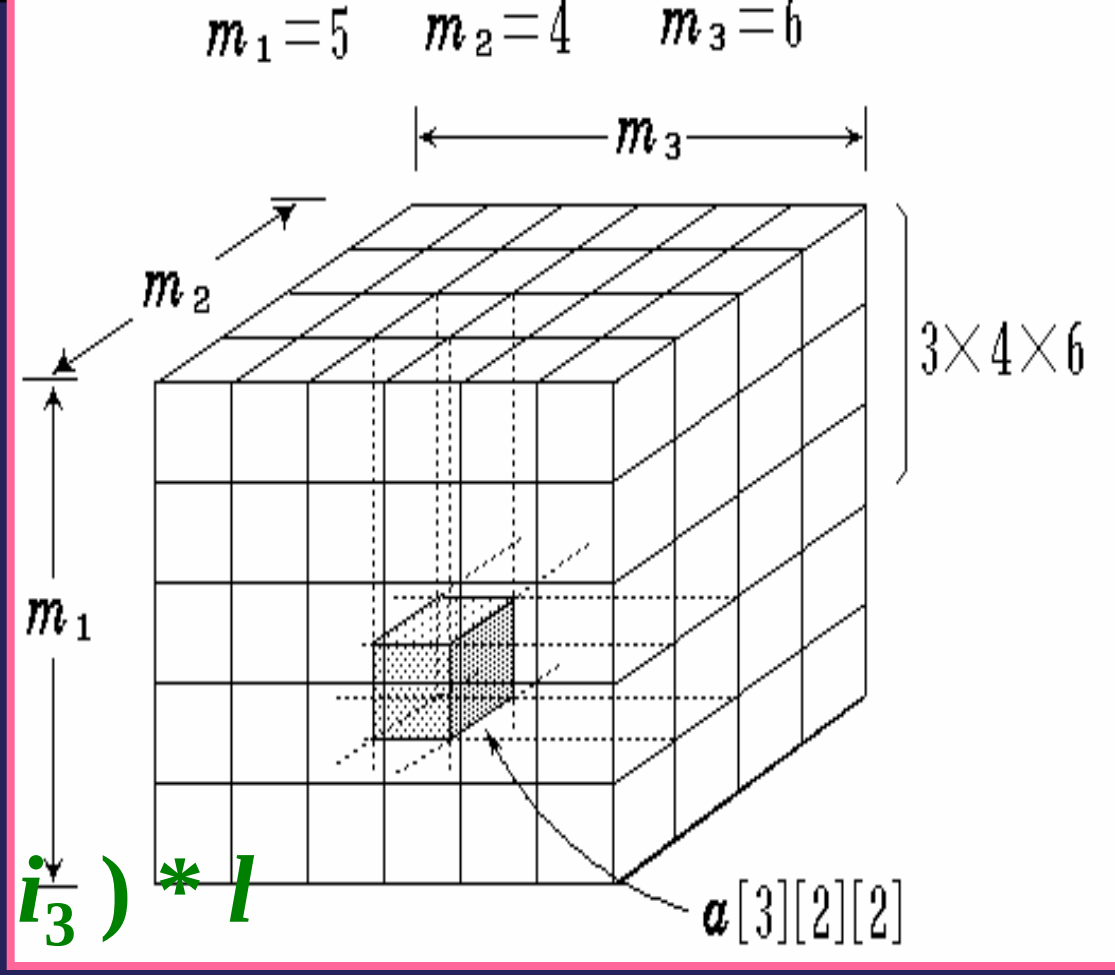
各维元素个数为 **m1, m2, m3 .**下标为 **i1, i2, i3**的数组元素的存储地址：

（按页/行/列存放）

$$LOC(i_1, i_2, i_3) = a + (i_1 * m_2 * m_3 + i_2 * m_3 + i_3) * l$$

前i_1页总元素个数

第i_1页的前i_2行总元素个数



页向量 下标**i**
行向量 下标**j**
列向量 下标**k**

n 维数组

各维元素个数为 $m_1, m_2, m_3, \ldots, m_n$

下标为 $i_1, i_2, i_3, \ldots, i_n$ 的数组元素的存储地址：

$$LOC(i_1, i_2, \ldots, i_n) = a + (i_1 * m_2 * m_3 * \ldots * m_n + i_2 * m_3 * m_4 * \ldots * m_n + \ldots\ldots + i_{n-1} * m_n + i_n) * l$$

$$= a + \left(\sum_{j=1}^{n-1} i_j * \prod_{k=j+1}^{n} m_k + i_n \right) * l$$

顺序存储的特点:

随机存取，即存取任何元素花的时间相同。

实现:

根据数组说明，得到数组的元素个数，即元素类型（每个元素占用空间），然后，分配连续空间，空间的首地址存储在数组名中。

小结

- 数组中的每个元素都受n个线性关系的约束，在每个关系中，每个元素 aij都有且仅有一个直接前趋，都有且仅有一个直接后继。
- 二维数组有两种方式：一种是以行为主序的方式，另一种是以列序为主序的方式。

5.2 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

基本内容

1　特殊矩阵的压缩存储；

2　稀疏矩阵的压缩存储；

3　稀疏矩阵在三元组顺序表存储方式下，矩阵的运算的实现；

学习要点

1　了解矩阵的两种压缩存储方法及适用范围;

2　了解在三元组顺序表存储方式下，实现矩阵运算的方法；

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。这里我们感兴趣的不是矩阵本身，而是矩阵在计算机内的有效存储方法和对应的各种矩阵运算的算法。

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。

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

然而应用中常遇到一些阶数很高的矩阵，并且矩阵中有许多值相同的元素或零元素，用二维数组存储矩阵会浪费很多的存储单元存放实际上可以不必存放的元素。为了节省存储空间，对这类矩阵可以压缩存储。所谓*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。

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，而压缩存储只需800个存储单元。

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

一　特殊矩阵

1　什么是特殊矩阵

具有某种特性的矩阵,如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

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

对称矩阵(转置矩阵):　m[i][j]=m[j][i]
对角矩阵:　当i!=j时m[i][j]=0
上三角矩阵:　当i>=j时 m[i][j]=0或常数C
下三角矩阵:　当i<=j时m[i][j]=0或常数C
带状矩阵(带宽为K):当|i-j|>k时m[i][j]=0

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

(转置矩阵)

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{matrix}$$

(对角矩阵)

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

(上三角矩阵)

m[i][j]=m[j][i]

当i!=j时
m[i][j]=0

当i>=j时
m[i][j]=0或常数C

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

(下三角矩阵)

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

(转置矩阵)
(带状矩阵)

$$\begin{matrix} 1 & 2 & 3 & 0 & 0 \\ 2 & 3 & 4 & 5 & 0 \\ 3 & 4 & 5 & 6 & 7 \\ 0 & 5 & 6 & 7 & 8 \\ 0 & 0 & 7 & 8 & 9 \end{matrix}$$

(转置矩阵)

k=2
(带状矩阵)

当i<j时
m[i][j]=0或常数C

当|i-j|>k时m[i][j]=0

2　特殊矩阵压缩存储

特殊矩阵可以采用二维数组同样的存储方式，占用空间维m * n，但是，由于其特殊性，空间效率不高(存储了许多零或相同的值)，为此，它们一般采用特殊的存储方式（相同值只存放一个，零元素不存储）

1	0	0	0
2	3	0	0
4	5	6	0
7	8	9	10

原来

1	0	0	0	2	3	0	0	4	5	6	0	7	8	9	10

现在

1	2	3	4	5	6	7	8	9	10

A 矩阵越大,空间节省越多!!

B 有行主序,列主序之分!!

下面以对称矩阵为例，讨论特殊矩阵的压缩存储. 对称矩阵是满足下面条件的n 阶矩阵

$$a_{ij} = a_{ji} \quad 1 \le i,j \le n$$

存储元素数: $1+2+\ldots+n = n(n+1)/2$

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

设用一维数组S[n(n+1)/2]存储n阶对称对称矩阵A的下三角(包括主对角线元素)的元素。不失一般性，我们以行序为主序方式存储对称矩阵下三角元素。

0	1	2	3	4	5					n(n+1)/2-1					
a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	……	a_{m1}	a_{m2}	……	a_{mm}					

A中任意一元素 a_{ij}与它的存储位置之间存在着如下对应关系：

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

对任意一组下标值（i,j），均可以在S[ ]找到矩阵 a_{ij}，反之，对所有 k=0,2…n(n+1)/2-1,都能确定S[k]在矩阵中的位置（i ,j ）因此，称S[ ]为n阶对称矩阵A的压缩存储。

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

例 对称矩阵

$$A = \begin{pmatrix} 4 & 5 & 3 & 2 & 1 \\ 5 & 2 & 1 & 5 & 6 \\ 3 & 1 & 3 & 2 & 7 \\ 2 & 5 & 2 & 8 & 9 \\ 1 & 6 & 7 & 9 & 5 \end{pmatrix} \qquad \underline{A} = \begin{pmatrix} 4 & & & & \\ 5 & 2 & & 0 & \\ 3 & 1 & 3 & & \\ 2 & 5 & 2 & 8 & \\ 1 & 6 & 7 & 9 & 5 \end{pmatrix}$$

n = 5, 1+2+3+4+5 = 5*(5+1)/2 = 15

一维数组SA[0..15]作为数组A的存储结构:

SA=(4 5 2 3 1 3 2 5 2 8 1 6 7 9 5)

如: a[5,3] = a[3,5] = 7

k = 5(5-1)/2 + (3-1) = 12

故:sa[12] = 7

二 稀疏矩阵

1 什么是稀疏矩阵

有较多值相同元素或较多零元素，且值相同元素或者零元素分布没有一定规律的矩阵称为稀疏矩阵

例

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

M有42（6×7）个元素

有8个非零元素

m*n 的矩阵中，非零元有t 个，令k＝t/(m*n)，则k为矩阵稀疏因子，当k< = 0 .05 时为稀疏矩阵

分析：

根据稀疏矩阵的特点，如果存储所有元素，显然存放了很多0，空间利用率低！我们可以考虑只存放非0 元素，但如果仅仅存放一个元素值是不行的，因为不能确定它在矩阵中的位置，所以必须存储其下标。

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）

如何进行稀疏矩阵的压缩存储？

稀疏矩阵的压缩存储有多种方法，本课程主要介绍三元组顺序表这种存储方式。

1）三元组表

按照压缩存储的概念，只须存储稀疏矩阵的非零元素。但稀疏矩阵零元素分布没有一定规律，因此，除了存储非零元素的值之外，还必须同时记下它所在行和列号；反之，一个三元组（**i,j,e**），能唯一确定矩阵的一个非零元。由此，稀疏矩阵可由非零元的三元组表及其行数、列数唯一确定。

例如有稀疏矩阵：

$$\begin{matrix} 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{matrix}$$

可抽象为线性表：

按行： ((1,3,1), (1,5 2), (3,1,1), (4,3,1), (4,5,-1))

按列： ((3,1,1), (1,3,1), (4,3,1), (1,5,2), (4,5,-1))

线性表的存储我们已经介绍了,有顺序和链式两种,
所以稀疏矩阵也有两种存储形式!

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

```
#define MAXSIZE 12500
//假设非零元个数的最大值为12500
Typrdef struct {
        int          i,j;      // 非零元的行下标和列下标
        ElemType  e;           //非零元
}Triple;
Typedef union {
      Triple   data[MAXSIZE+1];
 //用于存储非零元三元组表, data[0]未用
      int        mu,nu,tu;
 //矩阵的行数、列数和非零元个数
}TSMatrix;
```

Triple： 是包含三个域的结构类型，其变量用于存储矩阵的非零元三元组

i, j： 两个整数域，用于存储 非零元的行下标和列下标

e： 用于存储 非零元的值

TSMatrix： 是包含四个域的结构类型，

data： 一维数组，用于存储矩阵的非零元三元组表

mu, nu, tu： 三个整数域，分别存储矩阵的行数、列数和非零元个数

设M是 **TSMatrix** 类型的结构变量，则M有四个域，其中M.data用于存储矩阵M的三元组表，在此我们约定,M.data域中非零元三元组是以行序为主序顺序存储的。

设M是TSMatrix 类型的结构变量，则M有四个域，其中M.data用于存储矩阵M的三元组表，如右图所示

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

M的三元组顺序表图示

		i	j	e
M.data	0	1	2	12
	1	1	3	9
	2	3	1	-3
	3	3	6	14
	4	4	3	24
	5	5	2	18
	6	6	1	15
	7	6	4	-7
	8			

M.mu	6
M.nu	7
M.tu	8

3 转置运算算法

对于一个m行 n列的矩阵M，它的转置矩阵T是一个n行m列的矩阵。

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

M		T
第一列	→	第一行
第二列	→	第二行
第三列	→	第三行
第四列	→	第四行
第五列	→	第五行
第六列	→	第六行

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

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

矩阵M的三元组顺序表

M.data

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

M.mu	6
M.nu	7
M.tu	8

M的转置矩阵T的三元组顺序表

T.data

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

T.mu	7
T.nu	6
T.tu	8

1) 转置运算算法

TransposeSMatrix(TSMatrix M, TSMatrix &T)

基本思想

对M.data从头至尾扫描：

第一次扫描时，将M.data中列号为1的三元组赋值到T.data中，

第二次扫描时，将M.data中列号为2的三元组赋值到T.data中，

依此类推，直至将M.data所有三元组赋值到T.data中

M.data

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

M.mu	6
M.nu	7
M.tu	8

T.data

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

T.mu	7
T.nu	6
T.tu	8

转置运算算法

```
Status TransposeSMatrix(TSMatrix M, TSMatrix &T)  {
//采用三元组表存储表示，求稀疏矩阵M的转置矩阵T
    T.mu=M.nu;  T.nu=M.mu;  T.tu=M.tu;
    if (T.tu)  {
        q=1;              // q为当前三元组在T.data[ ]存储位置(下标)
        for (col=1; col<=M.nu;  ++col)
         for (p=1;  p<=M.tu;  ++p) //p为扫描M.data[ ]的“指示器”
                                   //p“指向”三元组称为当前三元组
              if (M.data[p].j= =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; ++q;}
   }
   return OK;   } //  TransposeSMtrix
```

转置运算算法图示

第六次扫描查找第6列元素

第二次扫描结束

M.data

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

T.data

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

对M六次扫描完成转置运算

时间复杂度分析

算法的基本操作为将**M.data**中的三元组赋值到T.data，是在两个循环中完成的，故算法的时间复杂度为**O(nu×tu)**

我们知道，若用二维数组存储矩阵，转置算法的时间复杂度为**O**（mu×nu）。当非零元的个数**tu**和矩阵元素个数mu×nu 同数量级时，转置运算的时间复杂度为**O(**mu×nu × **nu)**。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**tu** << mu×nu的情况。

该算法效率不高的原因是：对为实现M到T的转置，该算法对**M.data**进行了多次扫描。能否在对**M.data**一次扫描的过程中，完成**M**到**T** 的转置？

快速转置算法

下面介绍转置运算算法称为快速转置算法

分析 在M.data中， M的各列非零元对应的三元组是以行为主序存储的，故M的各列非零元对应的三元组存储位置不是“连续”的。然而，M的各列非零元三元组的存储顺序却与各列非零元在M中的顺序一致。

如：M的第一列非零元素是 -3、15，它们对应的三元组在M.data 中的存储顺序也是（3，1，-3）在前（6，1，15）后。

M.data

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

M.mu	6
M.nu	7
M.tu	8

如果能先求得M各列第一个非零元三元组在T.data中的位置，就能在对M.data一次扫描的过程中，完成M到T 的转置：

对M.data一次扫描时， 首先遇到各列的第一个非零元三元组，可按先前求出的位置，将其放至T.data中，当再次遇到各列的非零元三元组时，只须顺序放到对应列元素的后面。

M.data

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

M.mu	6
M.nu	7
M.tu	8

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

M的各列非零元三元组
的存储顺序与各列
非零元在M中的顺序一致

M.data

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

M.mu	6
M.nu	7
M.tu	8

各列第一个非零元三元组按先前求出的位置，放至T.data中

各列后续的非零元三元组，顺序放到对应列元素的后面

M.data

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

M.mu	6
M.nu	7
M.tu	8

T.data

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

T.mu	7
T.nu	6
T.tu	8

辅助向量num[ ]、cpos[ ]

为先求得M各列第一个非零元三元组在T.data中的位置。引入两个辅助向量num[ ]、cpos[ ]：

num[col]：存储第col列非零元个数

cpos[col]：存储第col列第一个非零元三元组在T.data中的位置

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

例 矩阵M

cpos[col]的计算方法：

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

$$\begin{cases} cpos[1]=1 \\ cpos[col]=cpos[col-1]+num[col-1] \end{cases}$$

2 <= col<= n

快速转置算法主要步骤:

求M中各列非零元个数num[ ];

1 求M中各列第一个非零元在T.data中的下标cpos[ ];

3 对M.data进行一次扫描， 遇到col列的第一个非零元三元组时，按cpos[col]的位置，将其放至T.data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;

```
Status FastTransposeSMatrix(TSMatrix M, TSMatrix &T) {
  //采用三元组顺序表存储表示，求稀疏矩阵M的转置矩阵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];        //求M中每一
列非零元个数,求第 col列中第一个非零元在T.data中的序号
cpot[1]=1;
for(col=2; col<=M.tu;  ++col)  cpot[col]=cpot[col-1]+num[col-1];
   for(p=1; p<M.tu;  ++p) {
       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];
   } //for
 } //if       return OK;  }   //FastTransposeSMatrix
```

求各列非零元个数

求各列第1个非零元在b中位置

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

第1列第一个非零元在b中的位置

第4 列第一个非零元在b中的位置

扫描M.data实现M到T的转置

M.data

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

T.data

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

快速转置算法图示

时间复杂度分析

该算法利用两个辅助向量num[ ] 、cpos[ ]，实现了对M.data一次扫描完成M到T 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mu、tu、t和tu，因而总的时间复杂度为O（mu+tu)，在M的非零元个数tu和mu×nu 同等数量级时，其时间复杂度为O(mu×nu），和转置算法5.1的时间复杂度相同。由此可见，只有当tu<< mu×nu时，使用快速转置算法才有意义。

4) 稀疏矩阵三元组链式存储-----十字链表:

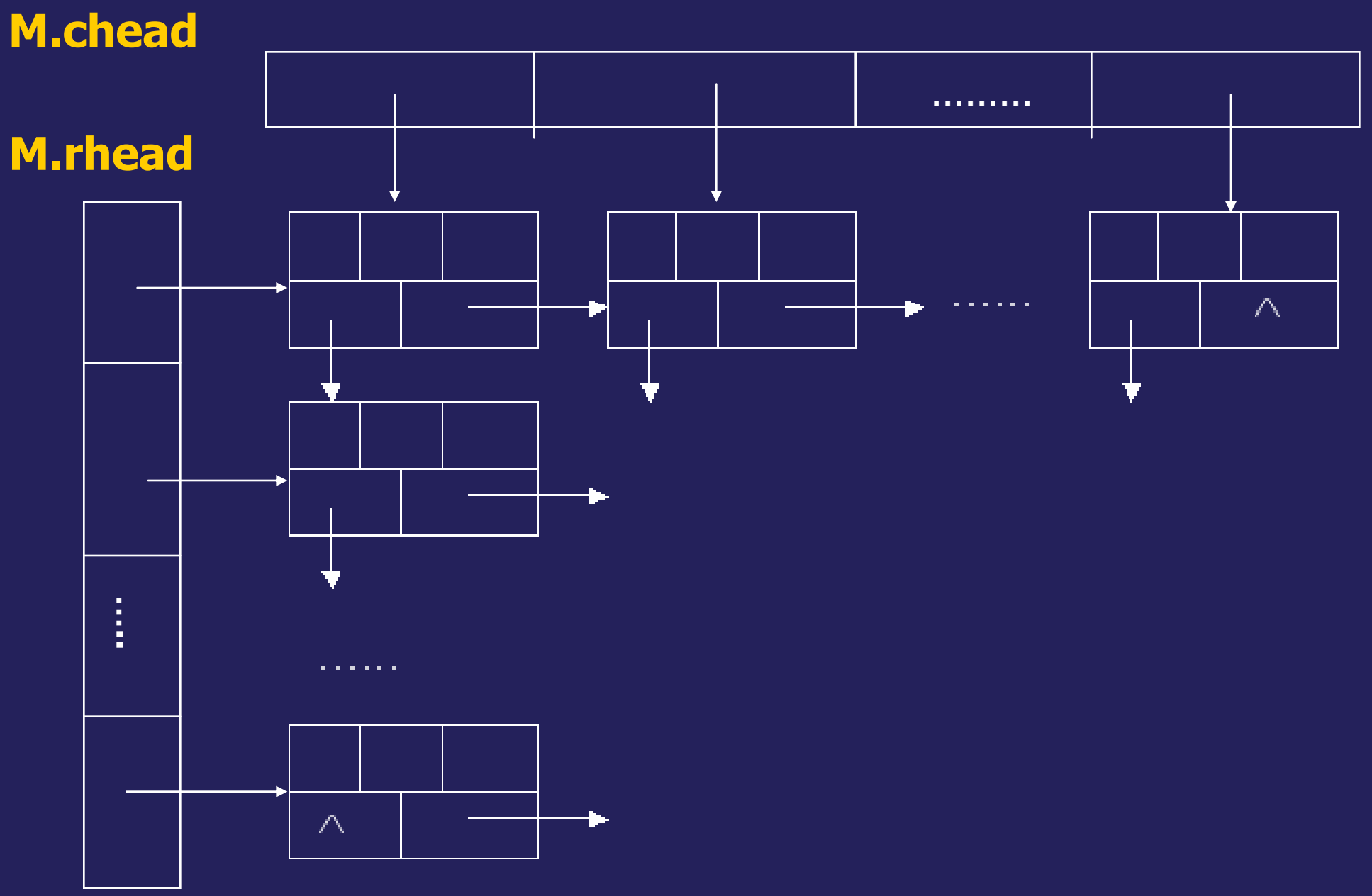
每个三元组用一个如下的结点存储：

row	col	val
down		right

row , col ： 非**0**元素的行、列值
val ： 非**0** 元素的值
down ： 指向同列的下一个非**O** 元素结点
right ： 指向同行的下一个非**O** 元素结点

稀疏矩阵的十字链表存储表示

```
Typedef struct OLNode{
    int i,j;     //该非零元的行和列下标
    ElemType e;
Struct OLNode *right,*down;
        //该非零元所在行表和列表的后继链域
}OLNode; *OLink;
Typedef struct{
    OLink *rhead,*chead;
        //行和列链表头指针向量基址
     int mu,nu,tu;
        //稀疏矩阵的行数,列数和非零元个数
}CrossList;
```


M.chead
M.rhead

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零元素的值外，还要保存零元素在矩阵中的位置；

5.3 广义表

5.3.1 广义表的概念

5.3.2 广义表的存储结构

基本内容

1 广义表的概念；

2 广义表的基本操作；

3 广义表的存储结构；

学习要点

了解广义表的结构特征及存储方法；

5.3.1 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= (α_1, α_2。。α_n)。其中α_i其可以是单个元素，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，元素可是以单个元素称为原子，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 = (e) B中只有一个元素e，表长为1;
C = (a,(b,c,d)) C的表长为2，两个元素分别为 a 和子表（b,c,d);
D = (A,B,C) D 的表长为3，它的三个元素 A，B，C 广义表;

5）广义表的术语:

长度：元素个数n （注意子表是一个数据元素）;
单元素：一般用小写字母表示;
子表:一般用大写字母表示;
表头:当广义表LS 非空时，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;

深度:简单说就是表的嵌套层次，定义为:

LS = (dl , d2 ， … ， dn)

depth (LS) =

max{depth(dl), depth(d2)， … ， depth(dn) } + l

$$\text{depth}(di) = \begin{cases} 0 & di \text{ 是单元素} \\ l & di \text{ 是空表} \end{cases}$$

例如: B = (e)　　表头：e　　表尾 ( )

C = (a,(b,c,d))　表头：a　　表尾 ((b,c,d))

D = (A,B,C)　　表头：A　　表尾 (B,C)

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

depth(B)=max{0}+1

例:求下列广义表的长度和深度.

A=()　　长度0　　深度1

B=(e)　　长度1　　深度1

C=(a,(b,c,d))　　长度2　　深度2

D=(A,B,C)　　长度3　　深度3

depth(C)=max{0,1}+1

depth(D)=max{1,1,2}+1

2　广义表的基本操作

1）创建空的广义表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的第一个元素；
10）删除广义表L的第一个元素，并e用返回其值；
11）遍历广义表L，用函数visit( )处理每个元素；

5.3.2 广义表的存储结构

由于广义表中的数据元素可以具有不同的类型，（或是原子，可是广义表）因此难以用顺序存储结构表示，通常采用链式存储结构，一种是首尾链表，另一种是扩展线性链表，本课程只介绍首尾链表存储方式。

如何设定首尾链表结点的结构？由于广义表中的数据元素可能为原子或列表，由此需要两种结构的结点：一种是*表结点*，用以表示广义表；一种是*原子结点*，用以表示原子。从上节得知：若广义表不空，则可分成表头和表尾，反之，一对确定的表头和表尾可唯一确定广义表。由此，一个表结点可由三个域组成：标志域、指示表头的指针域和指示表尾的指针域；而原子结点只需两个域：标志域和值域：

tag	ptr	
1	hp	tp

表结点

tag	atom
0	

原子结点

其中 atom： 是原子结点的值域

hp:是指向表头元素结点(单元素?子表?)

tp:是指向表尾元素结点(单元素?子表?)

tag:是标志域，区分是什么结点（单元素?子表?）

首尾链表结点的类型定义如下：

```
Typedef  enum{ ATOM,LIST}ElemTag;
//ATOM==0:原子,LIST==1: 列表
Typedef struct  GLNode {
        ElemTag  tag;      //标志域：用于区分原子结点和表结点
         union {
            AtomType  atom;                    //原子结点的值域
            struct  {struct  GLNode  *hp,*tp;}ptr;
    //表结点的指针域，由两个指针域构成：ptr.hp指向表头，
// ptr.tp指向表尾
        };
}*Glist;
```

tag	ptr	
1	hp	tp

表结点

tag	atom
0	

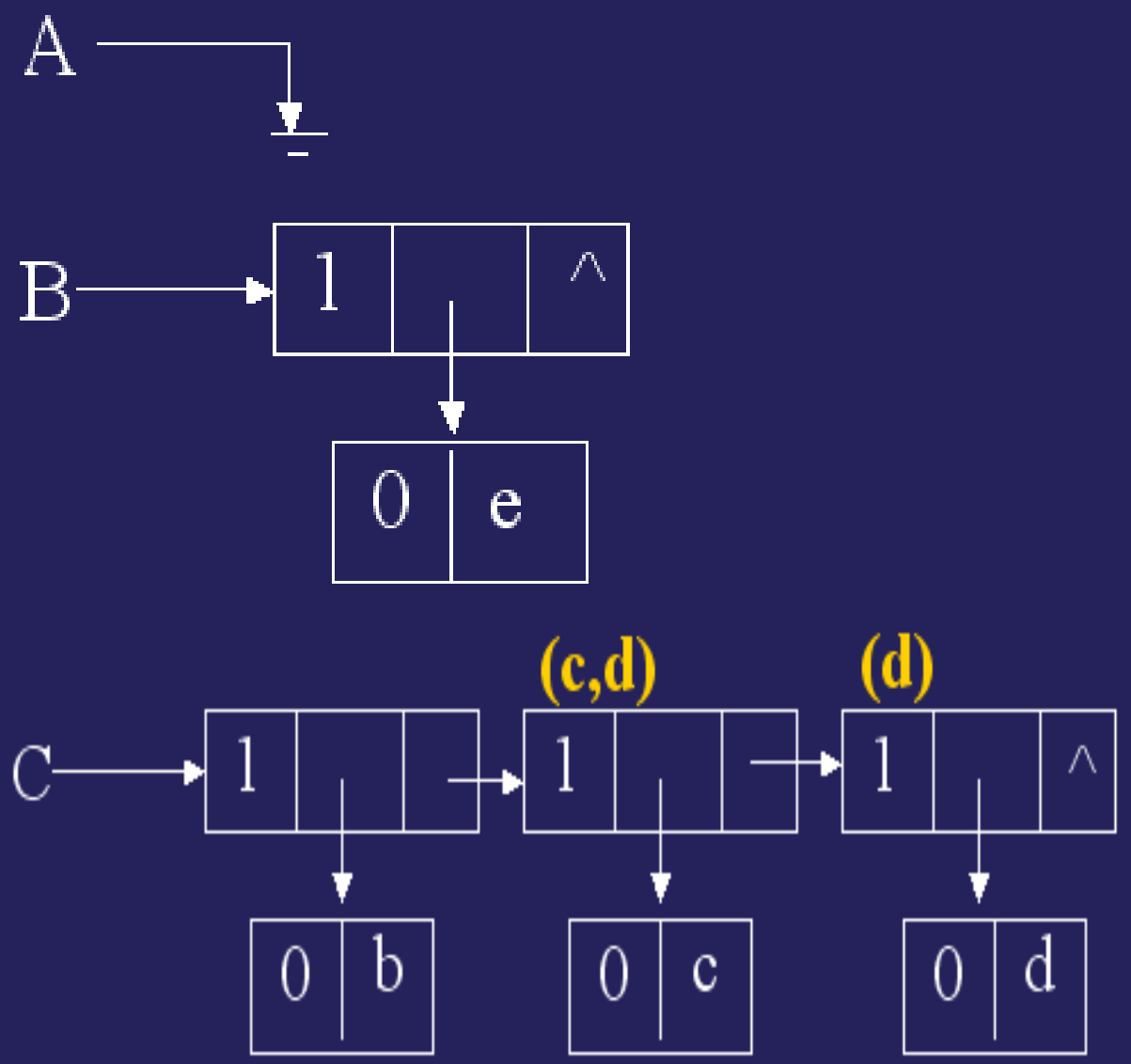
原子结点

广义表的存储结构示例

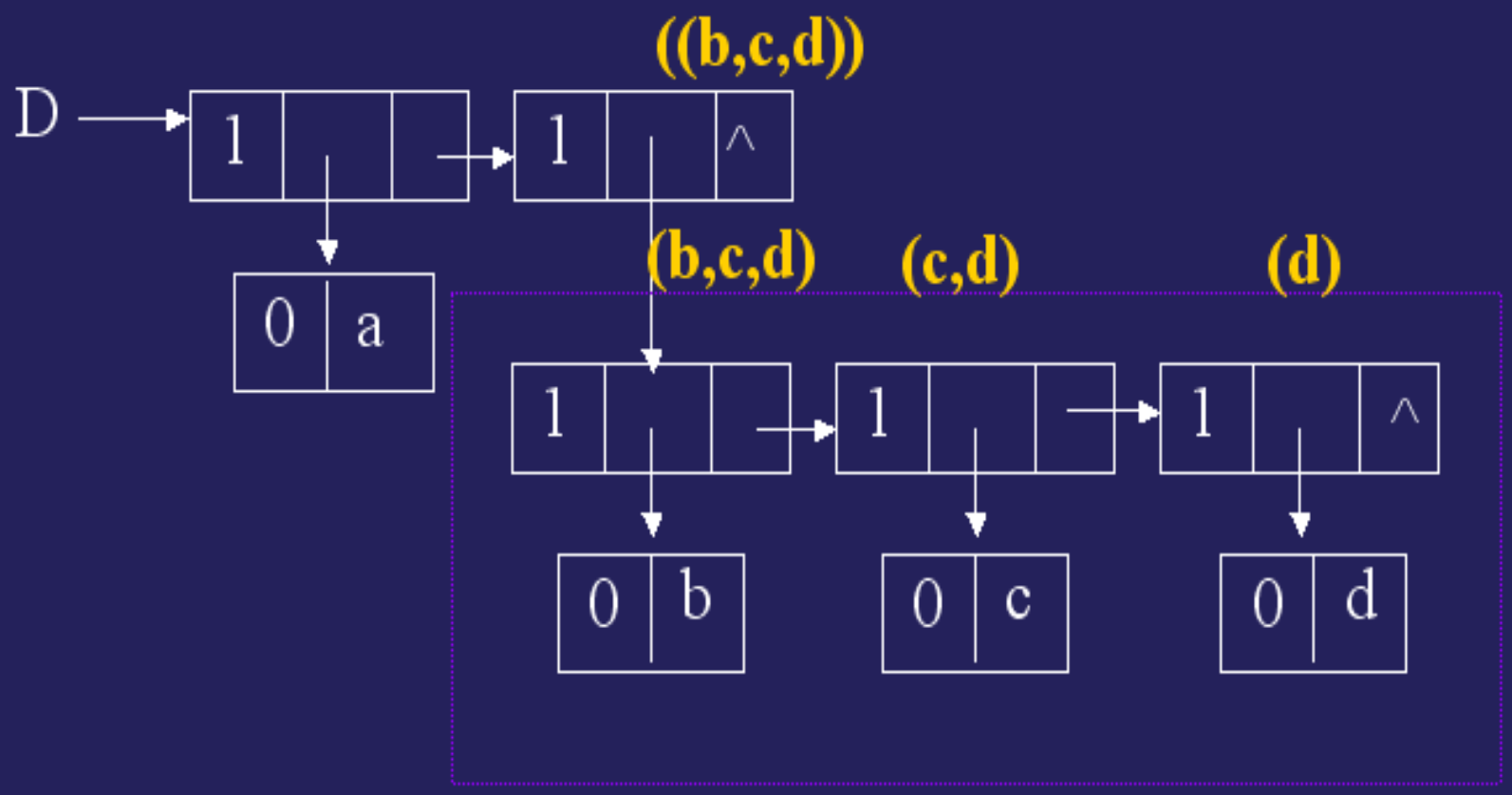
A=()

B=(e)

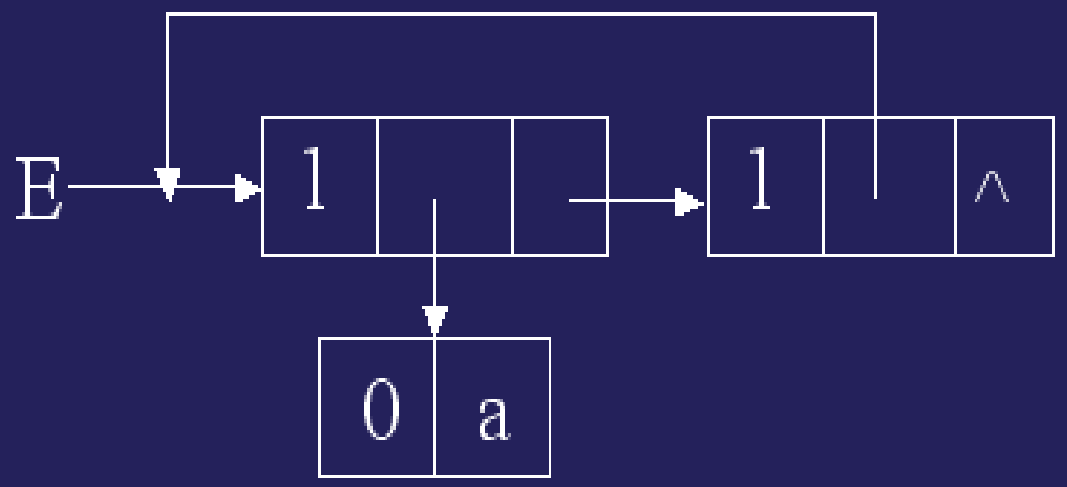
C=(b,c,d)



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



E=(a,(E))



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　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；

3　广义表，通常采用链式存储结构，，本课程只介绍首尾链表存储方式。链表中有两种的结点：一种是表结点，用以表示广义表；一种是原子结点，用以表示原子；