



数据结构

第五章 数组和广义表

主讲：陈锦秀

厦门大学信息学院计算机系

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

5.1 数组的定义

- 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。
- 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。例如，二维数组：

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

可以看成是由各行向量组成的向量，也可以看成是各列向量组成的向量。

5.1 数组的定义

- 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。

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

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：
 - 行优先顺序
 - 列优先顺序

5.2 数组的顺序表示和实现

- 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：

$a_{11}, a_{12}, \dots, a_{1n}, a_{21}, a_{22}, \dots, a_{2n}, \dots, a_{m1}, a_{m2}, \dots, a_{mn}$

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：

$a_{11}, a_{21}, \dots, a_{m1}, a_{12}, a_{22}, \dots, a_{m2}, \dots, a_{n1}, a_{n2}, \dots, a_{nm}$

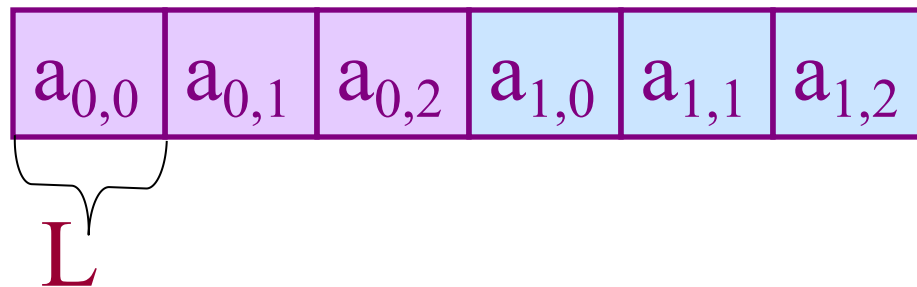
在FORTRAN语言中，数组就是按列优先顺序存储的。

5.2 数组的顺序表示和实现

- 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 $A[0..b_1, 0..b_2]$ 按“行优先顺序”存储在内存中，假设每个元素占用 L 个存储单元。

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$



元素 $a_{i,j}$ 的存储地址应是数组的基地址加上排在 $a_{i,j}$ 前面的元素所占用的单元数，即：

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

↑ 称为基地址或基址。

同样，三维数组 A_{ijk} 按“行优先顺序”存储，其地址计算函数为：

$$LOC(a_{ijk}) = LOC(a_{000}) + (i * b_2 * b_3 + j * b_3 + k) * L$$

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

$$\begin{aligned}\text{LOC}(j_1, j_2, \dots, j_n) \\ &= \text{LOC}(0, 0, \dots, 0) + (b_2 * b_3 * \dots * b_n * j_1 + b_3 * \dots * b_n * j_2 + \dots + b_n * j_{n-1} + j_n) * L \\ &= \text{LOC}(0, 0, \dots, 0) + \sum_{i=1}^n c_i j_i\end{aligned}$$

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

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。

- 上述讨论均是假设数组各维的下界是0，更一般的二维数组是 $A[c_1..b_1, c_2..b_2]$ ，这里 c_1, c_2 不一定是0。 a_{ij} 前一共有 $i-c_1$ 行，二维数组一共有 b_2-c_2+1 列，故这 $i-c_1$ 行共有 $(i-c_1)*(b_2-c_2+1)$ 个元素，第 i 行上 a_{ij} 前一共有 $j-c_2$ 个元素，因此， a_{ij} 的地址计算函数为：

$$LOC(a_{ij}) = LOC(a_{c_1c_2}) + [(i-c_1)*(b_2-c_2+1) + j-c_2] * L$$

数组的顺序存储表示:

```
#include <stdarg.h>

#define MAX_DIM 8

typedef struct {    // 数组结构
    Elemtype *base;    // 数组元素基址, 由InitArray分配
    int    dim;        // 数组维数
    int *bounds;       // 数组维界基址
    int *constants;    // 数组映象函数常量基址
} Array;    // 此结构表示的优点可以实现动态维数的数组
```

可变参数原理注解:

- 1、**va_list**: 表示可变参数列表类型, 实际上就是一个char指针
 - 2、**va_start**: 用于获取函数参数列表中可变参数的首指针(获取函数可变参数列表)
 - * **输出参数ap**(类型为**va_list**): 用于保存函数参数列表中可变参数的首指针(即, 可变参数列表)
 - * **输入参数A**: 为函数参数列表中最后一个固定参数
 - 3、**va_arg**: 用于获取当前**ap**所指的可变参数并将**ap**指针移向下一可变参数
 - * **输入参数ap**(类型为**va_list**): 可变参数列表, 指向当前正要处理的可变参数
 - * **输入参数T**: 正要处理的可变参数的类型
 - * **返回值**: 当前可变参数的值
- 在C/C++中, 默认调用方式 **cdecl** 是由调用者管理参数入栈操作, 且入栈顺序为从右至左, 入栈方向为从高地址到低地址。因此, 第1个到第n个参数被放在地址递增的堆栈里。所以, 函数参数列表中最后一个固定参数的地址加上第一个可变参数对其的偏移量就是函数的可变参数列表了 (**va_start的实现**); 当前可变参数的地址加上下一可变参数对其的偏移量的就是下一可变参数的地址了 (**va_arg的实现**)。这里提到的偏移量并不一定等于参数所占的字节数, 而是为参数所占的字节数再扩展为机器字长(**acpi_native_int**)倍数后所占的字节数 (因为入栈操作针对的是一个机器字), 这也就是为什么 **_bnd** 那么定义的原因。
- 4、**va_end**: 用于结束对可变参数的处理。实际上, **va_end** 被定义为空. 它只是为实现与 **va_start** 配对(实现代码对称和"代码自注释"功能)

可变参数原理注解:

- 对可变参数列表的处理过程一般为:
 - 1、用**va_list**定义一个可变参数列表
 - 2、用**va_start**获取函数可变参数列表
 - 3、用**va_arg**循环处理可变参数列表中的各个可变参数
 - 4、用**va_end**结束对可变参数列表的处理

数组的基本操作:

```
Status Initarray(Array &A,int dim , ...) //数组初始化
{ if (dim<1||dim>MAX_DIM) return ERROR;
  A.dim=dim;
  A.bounds=(int *)malloc(dim*sizeof(int));
  if (!A.bounds) exit(OVERFLOW);
  //若各维长度合法，则存入A.bounds，并计算数组元素总个数elemtotal
  elemtotal=1;
  va_start(ap, dim); //获取函数可变参数列表的首指针，
                     // dim为函数参数列表中最后一个固定参数
  for(i=0;i<dim;i++){
    A.bounds[i]=va_arg(ap, int);
    if (A.bounds[i]<0) return UNDERFLOW;
    elemtotal*=A.bounds[i];}
  va_end(ap); //和va_start函数配对，结束对可变参数的处理
```


数组的基本操作:

```
//分配数组元素存储空间, 并返回基址A.base
A.base=(Elemtype *)malloc(elemtotal*sizeof(elemtype));
if (!A.base) exit(overflow);
//分配数组映像函数常量基址, 即求映像函数的常数 $c_i$ ,
//并存入A.constants[i-1],  $i=1,\dots,\text{dim}$ 
A.constants=(int *)malloc(dim*sizeof(int));
if(!A.constants) exit(OVERFLOW);
A.constants[dim-1]=1; //求映象函数  $L=1$ , 指针的增减以元素的大小为单位
for(i=dim-2;i>=0;i--)
    A.constants[i]=A.bounds[i+1]*A.constants[i+1];
return OK;
}
```

数组的基本操作:

Status Destory(Array &A) //销毁数组

```
{  
    if (!A.base) return ERROR;  
    free(A.base);  
    A.base=NULL;  
    if (!A.bounds) return ERROR;  
    free(A.bounds);  
    A.bounds=NULL;  
    if (!A.constants) return ERROR;  
    free(A.constants);  
    A.constants=NULL;  
}
```

数组的基本操作:

Status Locate(Array A, va_list ap, int &off)

//求出元素在A中的相对地址off,元素各下标值存放在ap中。

```
{  
    off=0;  
    for(i=0;i<A.dim;i++)  
    {  
        ind= va_arg(ap, int);  
        if(ind<0||ind>=A.bounds[i]) return OVERFLOW;  
        off+=A.constants[i]*ind;  
    }  
    return OK;  
}
```

数组的基本操作:

Status Value(array A,Elemtype &e, ...)

//A是n维数组，e是元素变量，随后是n个下标值。

//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

```
{  
    va_start(ap, e);  
    if(result=Locate(A,ap,off)<=0) return result;  
    e=*(A.base+off);  
    return OK;  
}
```

数组的基本操作:

Status Assign(Array &A, Elemtyp e, ...)

//A是n维数组, e为元素变量, 随后是n个下标值。

//若下标不超界, 则将e的值赋给所指定的A的元素, 并返回ok。

{

va_start(ap, e);

if(result=Locate(A,ap,off)<=0) return result;

***(A.base+off) = e;**

return OK;

}

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

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：

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

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。

5.3.1 特殊矩阵——对称矩阵

- 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：

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

- 在这个下三角矩阵中，第*i*行恰有*i*个元素，元素总数为：

$$1+2+\cdots+n=n(n+1)/2$$

- 因此，我们可以按图中箭头所指的次序将这些元素存放在一个向量 $sa[0..n(n+1)/2-1]$ 中。

5.3.1 特殊矩阵——对称矩阵

- 为了便于访问对称矩阵 \mathbf{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 $i-1$ 行（从第1行到第 $i-1$ 行）一共有 $1+2+\dots+i-1=i(i-1)/2$ 个元素，在第 i 行上， a_{ij} 之前恰有 $j-1$ 个元素（即 $a_{i1}, a_{i2}, \dots, a_{ij-1}$ ），因此有：
$$k=i*(i-1)/2+j-1 \quad 0 \leq k < n(n+1)/2$$
 - 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：
$$k=j*(j-1)/2+i-1 \quad 0 \leq k < n(n+1)/2$$
- 令 $i=\max(i,j)$ ， $j=\min(i,j)$ ，则 k 和 i, j 的对应关系可统一为：

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

5.3.1 特殊矩阵——对称矩阵

- 因此， a_{ij} 的地址可用下列式计算：

$$\text{LOC}(a_{ij}) = \text{LOC}(\text{sa}[k])$$

$$= \text{LOC}(\text{sa}[0]) + k * L = \text{LOC}(\text{sa}[0]) + [i * (i-1)/2 + j-1] * L$$

- 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 $\text{sa}[k]$ 中找到矩阵元素 a_{ij} ；
- 反之，对所有的 $k=0, 1, 2, \dots, n(n-1)/2-1$ ，都能确定 $\text{sa}[k]$ 中的元素在矩阵中的位置(i, j)。

- 由此，称 $\text{sa}[n(n+1)/2]$ 为阶对称矩阵A的压缩存储，见下图：

a_{11}	a_{21}	a_{22}	a_{31}	a_{n1}	...	a_{nn}
$k=0$	1	2	3		$n(n-1)/2$...	$n(n+1)/2-1$

5.3.1 特殊矩阵——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。
- 在大多数情况下，三角矩阵常数为零。

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ c & a_{21} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ c & c & \dots & a_{nn} \end{pmatrix}$$

(a) 上三角矩阵

$$\begin{pmatrix} a_{11} & c & \dots & c \\ a_{21} & a_{22} & \dots & c \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

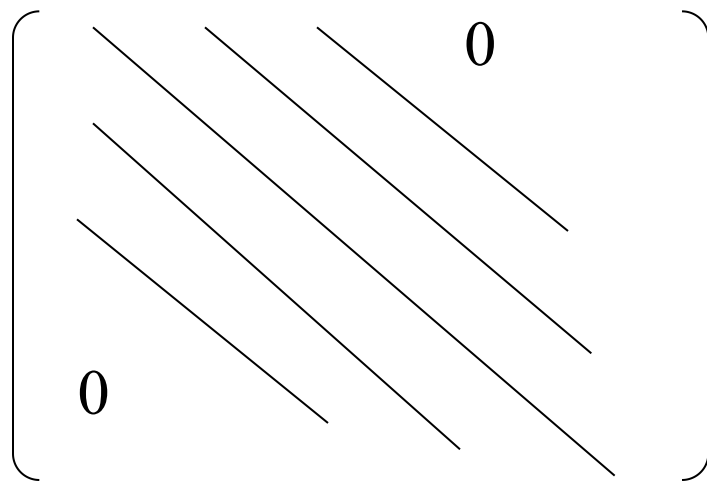
(b) 下三角矩阵

5.3.1 特殊矩阵——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。

5.3.1 特殊矩阵——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。
如P96图5.4
- 可按某个原则（或以行为主，或以对角线的顺序）将其压缩存储到一维数组上。



5.3.1 特殊矩阵

- 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 $s \leq m \times n$ ），则称A为稀疏矩阵。
- 精确点，设在矩阵A中，有t个非零元素。
令 $\delta = t / (m * n)$ ，称 δ 为矩阵的稀疏因子。
通常认为 $\delta \leq 0.05$ 时称之为稀疏矩阵。

5.3.2 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。

5.3.2 稀疏矩阵

- 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) 。
- 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。

例如，下列三元组表

$((1,2,12)(1,3,9),(3,1,-3),(3,6,14),(4,3,24),$
 $(5,2,18),(6,1,15),(6,4,-7))$

加上(6,7)这一对行、列值便可作为下列矩阵M的另一种描述。而由上述三元组表的不同表示方法可引出稀疏矩阵不同的压缩存储方法。

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

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

5.3.2 稀疏矩阵

■ 稀疏矩阵的三种压缩存储方法:

1. 三元组顺序表
2. 行逻辑链接的顺序表
3. 十字链表

5.3.2 稀疏矩阵——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 12500           //非零元个数的最大值

typedef struct{
    int i,j; //非零元的行下标和列下标
    ElemType e;
}Triple;

typedef struct{
    Triple data[MAXSIZE+1]; //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;
```

图5.5中所示的稀疏矩阵M及其对应转置矩阵T的三元组的表示分别为：

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

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

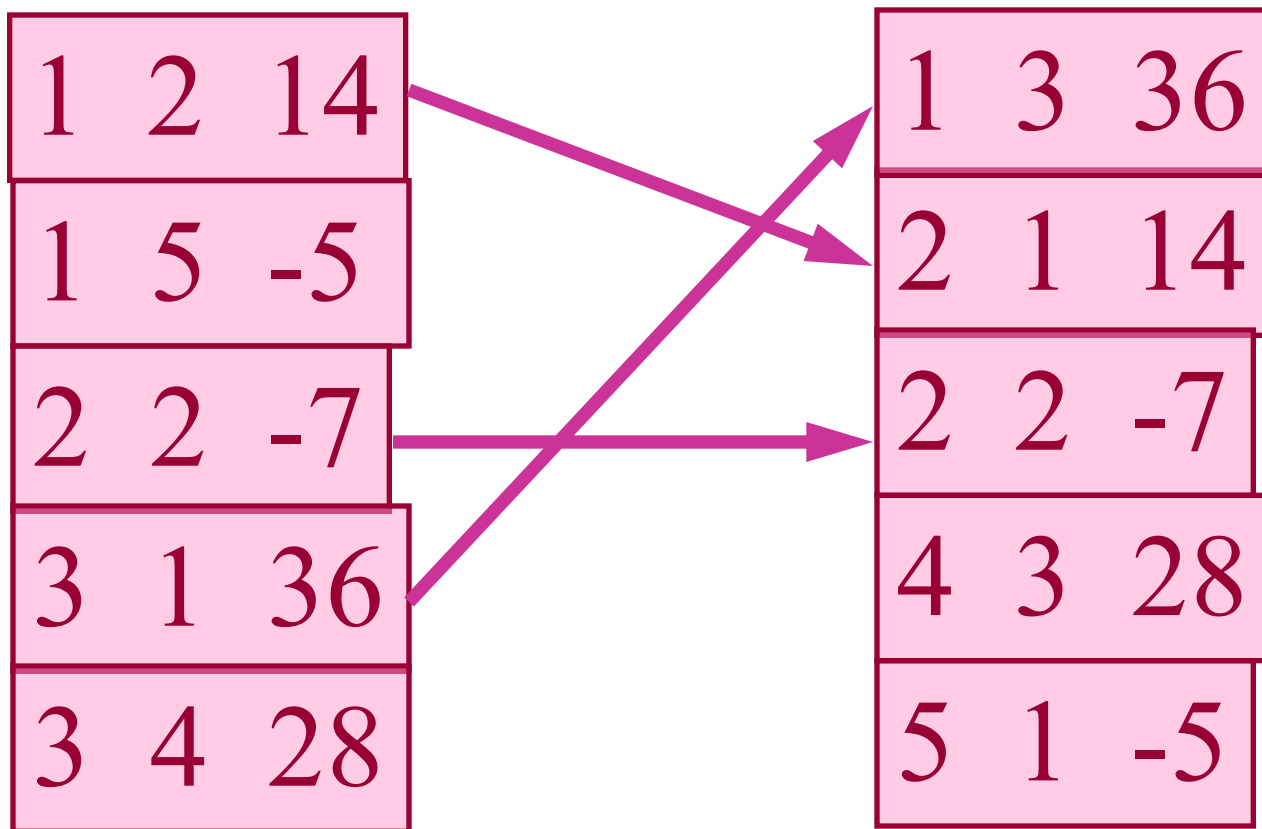
5.3.2 稀疏矩阵——三元组顺序表

- 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 $m \times n$ 的矩阵A，它的转置B是一个 $n \times m$ 的矩阵，且 $a[i][j]=b[j][i]$ ， $1 \leq i \leq m$ ， $1 \leq j \leq n$ ，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。

5.3.2 稀疏矩阵——三元组顺序表

- 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。
- **其基本思想是：**对A中的每一列 $col(1 \leq col \leq n)$ ，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**。

用“三元组”表示时如何实现？



Status TransposeSMatrix(TSMatrix M0,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;

for(col=1;col<=M.nu;col++)

for(p=0;p<=M.tu;p++)

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;

}

5.3.2 稀疏矩阵——三元组顺序表

- 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 $O(nu*tu)$ ，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=1;col<=nu;++col)
 for(row=1;row<=mu;++row)
 T[col][row]=M[row][col];
其时间复杂度为 $O(mu*nu)$ 。
- 当非零元素的个数tu和 $mu*nu$ 同数量级时，算法TransposeSMatrix的时间复杂度为 $O(mu*nu^2)$ 。

5.3.2 稀疏矩阵——三元组顺序表

- 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，此算法仅适用于非零元个数 $t \ll m * n$ 的情况。

5.3.2 稀疏矩阵——三元组顺序表

- 下面给出另外一种称之为**快速转置**的算法。
- **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中每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。

5.3.2 稀疏矩阵——三元组顺序表

- 为此，需要设置两个一维数组 $\text{num}[1..n]$ 和 $\text{cpot}[1..n]$
 - $\text{num}[1..n]$ ：统计 M 中每列非零元素的个数，例如 $\text{num}[1]$ 记录第 1 列非零元素的个数。
 - $\text{cpot}[1..n]$ ：由递推关系得出 M 中的每列第一个非零元素在 B 中的位置。
- 算法通过 cpot 数组建立位置对应关系：
$$\begin{aligned}\text{cpot}[1] &= 1 \\ \text{cpot}[\text{col}] &= \text{cpot}[\text{col}-1] + \text{num}[\text{col}-1] \\ 2 \leq \text{col} &\leq a.n\end{aligned}$$

首先应该确定每一列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

$\text{cpot}[1] = 1;$

for (col=2; col<=M.nu; ++col)

$\text{cpot}[\text{col}] = \text{cpot}[\text{col}-1] + \text{num}[\text{col}-1];$

例如：图5.4中的矩阵M和相应的三元组A可以求得num[col]和 cpot[col]的值如下：

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

第一列元素个数

第二列元素个数

第三列元素个数

A i j v

1	2	v

p

num

--	--	--	--

col=2

cpot

			...
--	--	--	-----

$q = \text{cpot}[\text{col}]$

B i j v

2	1	v

q

A的第二列第一个非
零元素在B中的位置

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){

 //求M中每一列含非零元个数

 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;

 //求第col列中的一个非零元在b.data中的序号

 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

 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];

 }

return OK;

}

5.3.2 稀疏矩阵——三元组顺序表

■ 算法分析

这个算法仅比前一个算法多用了两个辅助向量，然而其时间复杂度降为 $O(nu+tu)$ ，当 tu 和 $mu*nu$ 等数量级时，其时间复杂度为 $O(mu*nu)$ ，与经典算法的时间复杂度相同。

5.3.2 稀疏矩阵——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。

5.3.2 稀疏矩阵——行逻辑链接的顺序表

- 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：

```
typedef struct{  
    Triple data[MAXSIZE+1]; //非零元三元组表  
    int    rpos[MAXRC+1]; //各行第一个非零元的位置表  
    int    mu,nu,tu; //矩阵的行数、列数和非零元个数  
}RLSMatrix;
```

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
```

5.3.2 稀疏矩阵——行逻辑链接的顺序表

- 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：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 $O(m_1*n_1*n_2)$ 。

5.3.2 稀疏矩阵——行逻辑链接的顺序表

- 当M和N是稀疏矩阵并用三元组表存储结构时，就不能套用上述算法。假设M和N分别为：

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

则 $Q=M*N$ 为：

$$Q = \begin{pmatrix} -10 & 26 \\ -1 & 0 \\ 0 & 4 \end{pmatrix}$$

5.3.2 稀疏矩阵——行逻辑链接的顺序表

它们的三元组分别为：

i	j	e
1	1	3
1	3	5
3	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	1	-10
1	2	26
2	1	-1
3	2	4
Q.data		

5.3.2 稀疏矩阵——行逻辑链接的顺序表

- **稀疏矩阵相乘的基本思想是：**为了扫描M一次就可以实现算法，在处理M的每个元素时，求出该元素对Q的可能贡献。
 - 得到了该元素的所有贡献，也就不需要再次访问该元素了。比如对于M中的 $(1, 3, 5)$ ，这个元素对Q中的 $(1, 1)$ 位置贡献-10，对Q中的 $(1, 2)$ 位置贡献20。
- 也就是说，对于M中每个元素M，找到N中所有满足（M的列等于N的行）条件的元素，求得它们的乘积。这个乘积会对Q中的某个位置有所贡献。
- **注意到**，这个贡献只是Q中该位置最终结果的一部分。为了便于操作，为Q的每一位置各设一累加变量，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累加变量上。

5.3.2 稀疏矩阵——行逻辑链接的顺序表

(1) 由于M中的每一行，只和Q中的相应行有关。每处理M的一行，就得到Q的相应行，因此可以对M进行**逐行处理**。

——为了找到M中同一行的所有元素，可以利用M的rpos值。

(2) 为求Q的值，只需在M.data和N.data中找到相应的各对元素（即M.data中的j值和N.data中的i值相等的各对元素）相乘即可。

——为了找到N中匹配的所有数据，可以利用N的rpos值。

(3) 对Q的每一个元素设计一**累计和的变量**，其初值为零，其累加值 $Q[i][j]$ 也可能为零。因此乘积矩阵Q中的元素是否为零元，只有在求得其累加值后才能得知。因此，额外设计一中间数组temp[]，先求得累计求和的中间结果放到temp（Q的一行），然后再压缩到Q.data中去

计算 $Q=M \times N$ 的过程大致如下:

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0

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中;

}

}

Status MultSMatrix

(RLSMMatrix M, RLSMatrix N, RLSMatrix &Q)

{ //求矩阵乘积 $Q=M*N$, 采用行逻辑链接存储表示

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) { // Q是非零矩阵

for (arow=1; arow<=M.mu; ++arow) {

(处理M的每一行, 具体步骤见下一页)

} // 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
    brow=M.data[p].j;    //找到对应元在N中的行号
    if (brow < N.nu ) t = N.rpos[brow+1];
        else { t = N.tu+1 }
        for (q=N.rpos[brow]; q< t; ++q) {
            ccol = N.data[q].j;        // 乘积元素在Q中列号
            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) //压缩存储该行非零元
        if (ctemp[ccol]) {
            if (++Q.tu > MAXSIZE) return ERROR;
            Q.data[Q.tu] = {arow, ccol, ctemp[ccol]};
        } // if

```

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 $O(M.tu \times N.tu / N.mu)$,
进行压缩存储的时间复杂度为 $O(M.mu \times N.nu)$,
总的时间复杂度就是 $O(M.mu \times N.nu + M.tu \times N.tu / N.mu)$ 。

若 M 是 m 行 n 列的稀疏矩阵, N 是 n 行 p 列的稀疏矩阵,
则 M 中非零元的个数 $M.tu = \delta_M \times m \times n$,

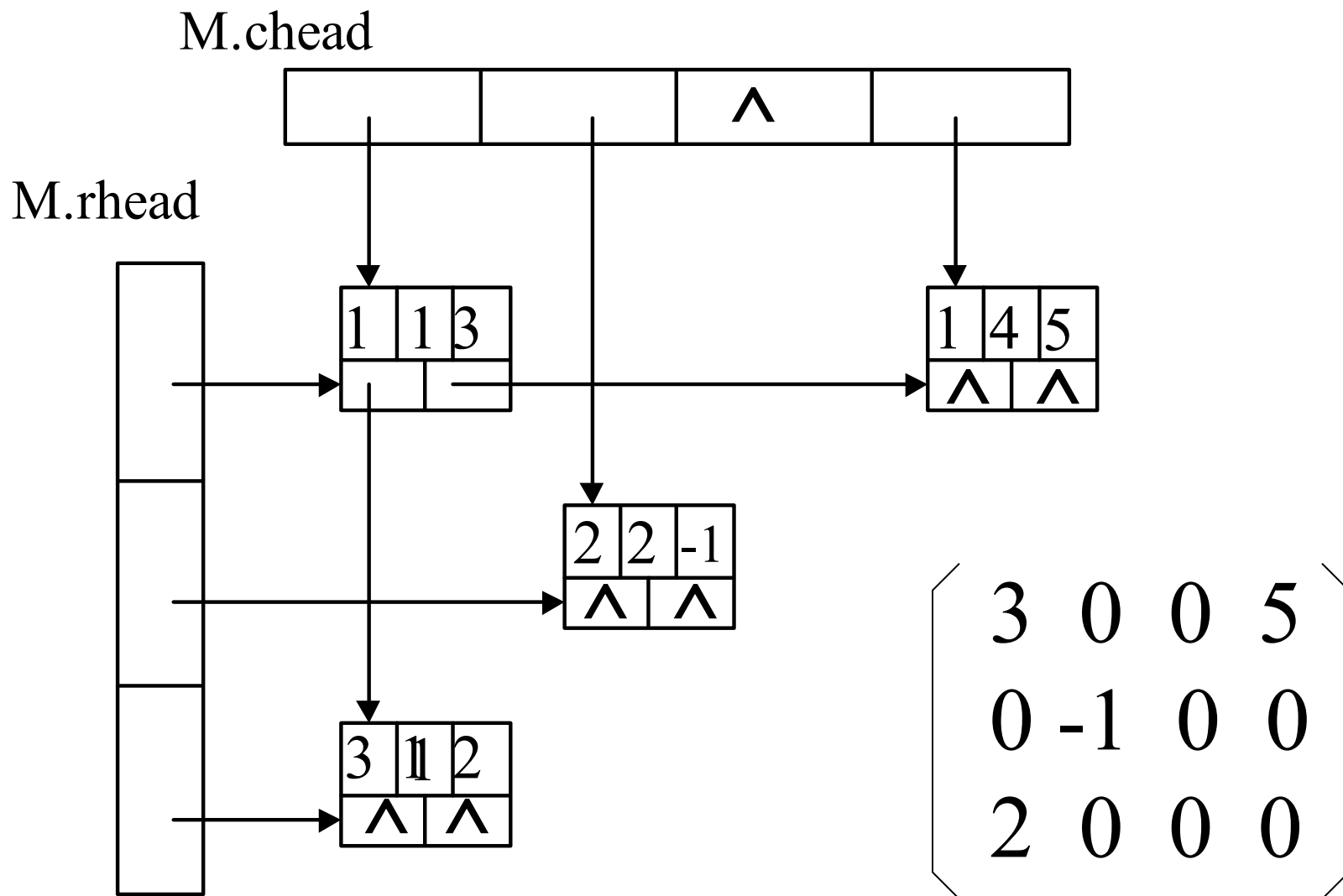
N 中非零元的个数 $N.tu = \delta_N \times n \times p$,
相乘算法的时间复杂度就是 $O(m \times p \times (1 + n \delta_M \delta_N))$,
当 $\delta_M < 0.05$ 和 $\delta_N < 0.05$ 及 $n < 1000$ 时,
相乘算法的时间复杂度就相当于 $O(m \times p)$ 。

5.3.2 稀疏矩阵——十字链表

- 当矩阵的非零元个数和位置在操作过程中变化较大时，就不宜采用顺序存储结构来表示三元组的线性表，而应采用链式存储结构的十字链表。

```
typedef struct OLNode{  
    int i,j;           //该非零元的行和列下标  
    ElemType e;        //该非零元的值  
    struct OLNode *right, *down; //行和列的后继  
}OLNode; *OLink;  
typedef struct{  
    OLink *rhead, *chead; //行和列的指针向量基址  
    int mu, nu, tu; //矩阵的行数、列数和非零元个数  
}CrossList;
```

5.3.2 稀疏矩阵——十字链表



5.3.2 稀疏矩阵——十字链表

- 算法5.4是建立十字链表的算法。
- 矩阵加法与多项式加法相似，回去自己看。

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

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 $n \geq 0$ 个元素 $a_1, a_2, a_3, \dots, a_n$ 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 $n(n \geq 0)$ 个元素 $a_1, a_2, a_3, \dots, a_n$ 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 $LS = (a_1, a_2, a_3, \dots, a_n)$ 。LS是广义表的名字， n 为它的长度，当 $n=0$ 时空表。若 a_i 是广义表，则称它为LS的子表。

5.4 广义表的定义

- 通常用圆括号将广义表括起来，用逗号分隔其中的元素。**广义表的深度**指的是广义表中括弧的重数。
- 为了区别原子和广义表，书写时用大写字母表示广义表，用小写字母表示原子。
- 若广义表LS ($n \geq 1$)非空，则 a_1 是LS的**表头**，其余元素组成的表(a_2, \dots, a_n)称为LS的**表尾**。
- 显然广义表是**递归定义**的，因为在定义广义表时又用到了广义表的概念。

5.4 广义表的定义

■ 广义表的例子如下：

- (1) $A = ()$ — A 是一个空表，其长度为零。
- (2) $B = (e)$ — 表 B 只有一个原子 e ， B 的长度为1。
- (3) $C = (a, (b, c, d))$ —— 表 C 的长度为2，两个元素分别为原子 a 和子表 (b, c, d) 。
- (4) $D = (A, B, C)$ —— 表 D 的长度为3，三个元素 A, B, C 都是广义表。显然，将子表的值代入后，则有
 $D = ((), (e), (a, (b, c, d)))$ 。
- (5) $E = (a, E)$ —— 这是一个递归的表，它的长度为2， E 相当于一个无限的广义表 $E = (a, (a, (a, (a, \dots))))$ 。

5.4 广义表的定义

从上述定义和例子可推出广义表的三个性质：

- 广义表的元素可以是子表，而子表的元素还可以是子表。由此，广义表是一个多层次的~~结构~~，可以用图形象地表示。
- 广义表可为其它表所共享。例如在上述例（4）中，广义表A，B，C为D的子表，则在D中可以不必列出子表的值，而是通过子表的名称来引用。
- 广义表可以是一个递归的表，如例（5）。

广义表是一个多层次的线性结构

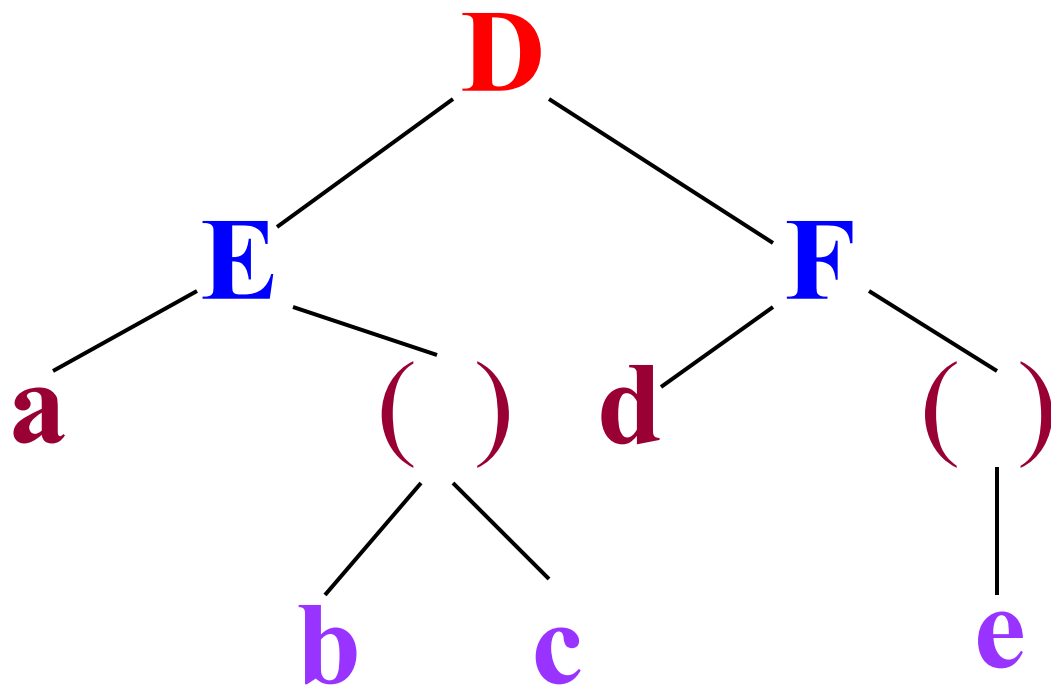
例如:

$$D=(E, F)$$

其中:

$$E=(a, (b, c))$$

$$F=(d, (e))$$



5.4 广义表的定义

- 由表头、表尾的定义可知：任何一个非空广义表其表头可能是原子，也可能是广义表，而其表尾必定是广义表。

对于 $B=(e)$ $\text{GetHead}(B)=e$, $\text{GetTail}(B)=()$

对于 $D=(A,B,C)$ $\text{GetHead}(D)=A$, $\text{GetTail}(D)=(B,C)$

由于 (B,C) 为非空广义表，则可继续分解得到：

$\text{GetHead}(B,C)=B$ $\text{GetTail}(B,C)=(C)$

- 注意广义表 $()$ 和 $(())$ 不同。前者是长度为0的空表，对其不能做求表头的和表尾的运算；而后者是长度为1的非空表（只不过该表中唯一的一个元素是空表），对其可进行分解，得到表头和表尾均为空表 $()$ 。

广义表举例:

例如: $D = (E, F) = ((a, (b, c)), F)$

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

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

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

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

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

小结：广义表 $LS = (\alpha_1, \alpha_2, \dots, \alpha_n)$ 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包括弧的重数；

注意：“原子”的深度为 0

“空表”的深度为 1

- 4) 广义表可以共享；
- 5) 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。

- 6) 任何一个非空广义表 $LS = (\alpha_1, \alpha_2, \dots, \alpha_n)$ 均可分解为表头 $\text{Head}(LS) = \alpha_1$ 和表尾 $\text{Tail}(LS) = (\alpha_2, \dots, \alpha_n)$ 两部分。

5.4 广义表的定义

ADT Glist {

数据对象: $D = \{e_i \mid i=1,2,\dots,n; n \geq 0;$
 $e_i \in \text{AtomSet} \text{ 或 } e_i \in \text{GList},$
 $\text{AtomSet} \text{ 为某个数据对象} \}$

数据关系:

$LR = \{ \langle e_{i-1}, e_i \rangle \mid e_{i-1}, e_i \in D, 2 \leq i \leq n \}$

基本操作:

} ADT Glist

基本操作

- 结构的创建和销毁

InitGList(&L); DestroyGList(&L);
CreateGList(&L, S); CopyGList(&T, L);

- 状态函数

GListLength(L); GListDepth(L);
GListEmpty(L); GetHead(L); GetTail(L);

- 插入和删除操作

InsertFirst_GL(&L, e);
DeleteFirst_GL(&L, &e);

- 遍历

Traverse_GL(L, Visit());

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

5.5 广义表的存储结构

- 广义表 $(a_1, a_2, a_3, \dots, a_n)$ 中的数据元素可以具有两种不同的结构：**原子或广义表**。
 - 因此，难以用顺序存储结构表示，通常采用**链式存储结构**，每个数据元素可用一个结点表示。
- 下面介绍两种广义表的链式存储结构。需要两种结构的结点：一种是表结点，一种是原子结点。
 1. **表结点**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；
 2. **原子结点**只需两个域：标志域和值域。

表结点

tag=1	hp	tp
-------	----	----

原子结点

tag=0	atom
-------	------

广义表的头尾链表存储表示:

```
typedef enum{ATOM, LIST} ElemTag;
```

//ATOM==0: 原子, LIST==1: 子表

```
typedef struct GLNode{
```

ElemTag tag; *//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;

//ptr是表结点的指针域,

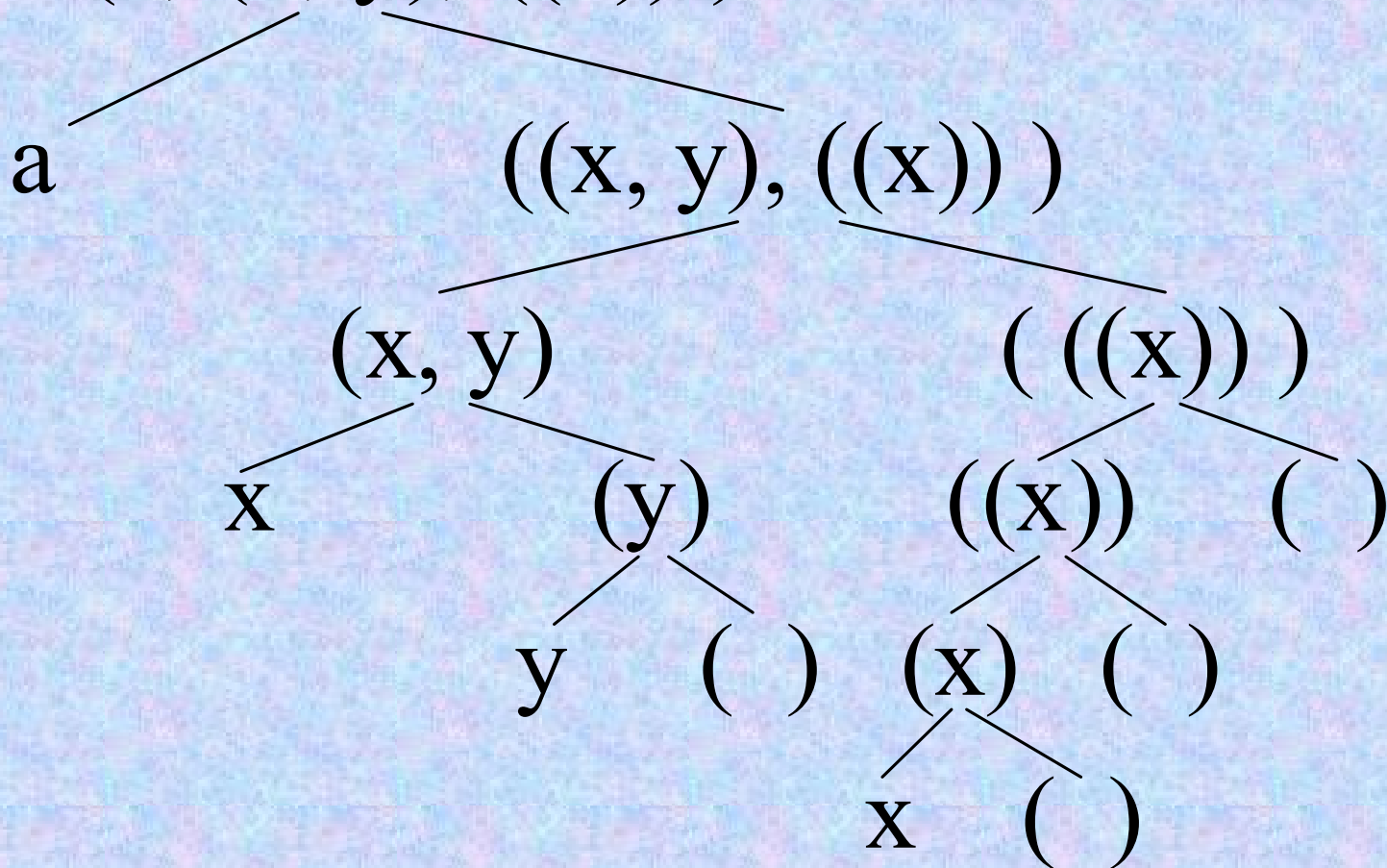
//ptr.hp和ptr.tp分别指示表头和表尾。

};

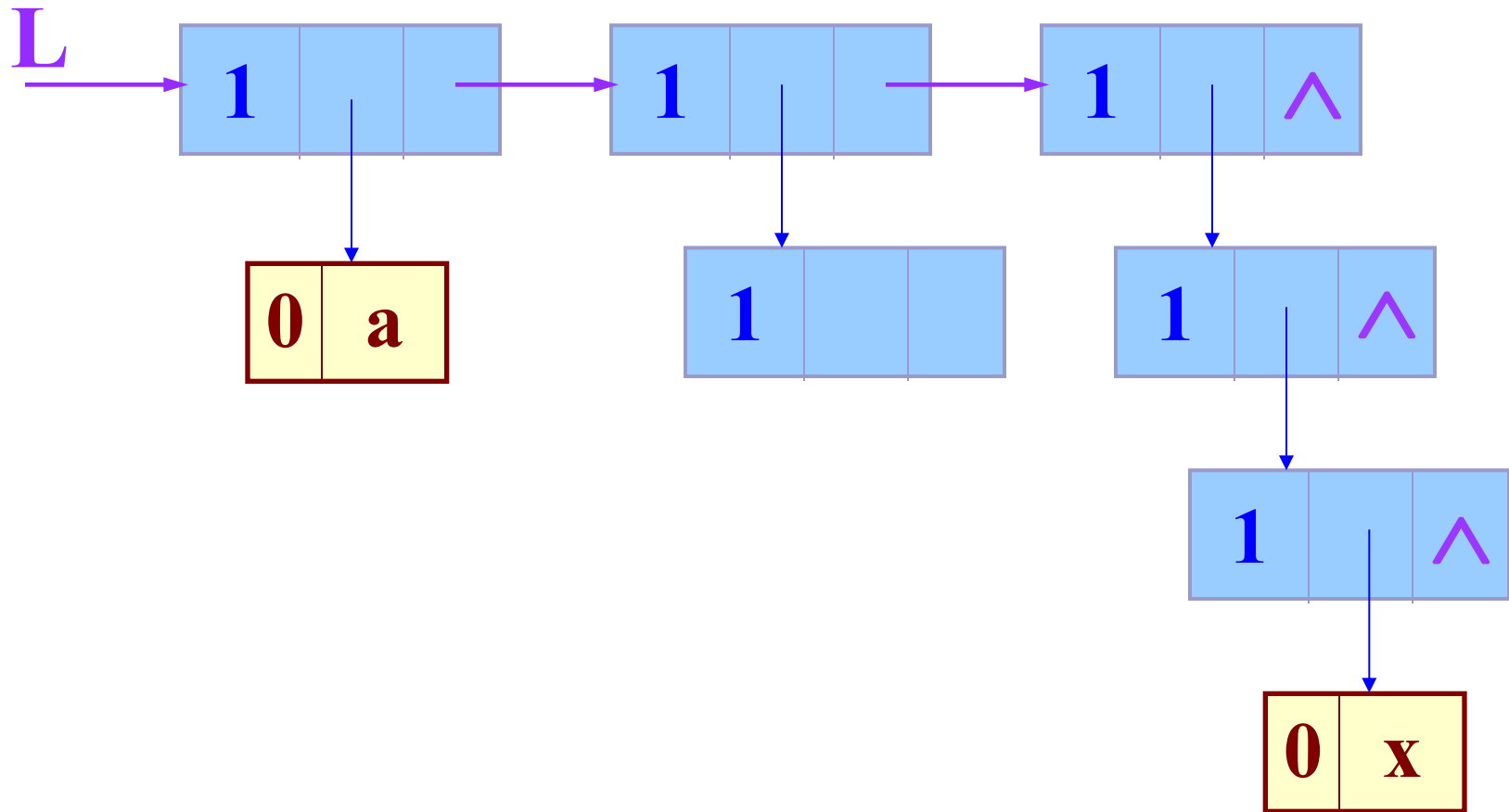
```
}*GList;
```

例如:

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



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



广义表的扩展线性链表存储表示:

表结点

tag=1	hp	tp
-------	----	----

原子结点

tag=0	atom	tp
-------	------	----

广义表的扩展线性链表存储表示:

```
typedef enum {ATOM, LIST}ElemTag;
```

```
typedef struct GLNode{
```

```
    ElemTag tag;
```

```
    union{
```

```
        AtomType atom;
```

//原子结点的值域

```
        struct GLNode *hp;
```

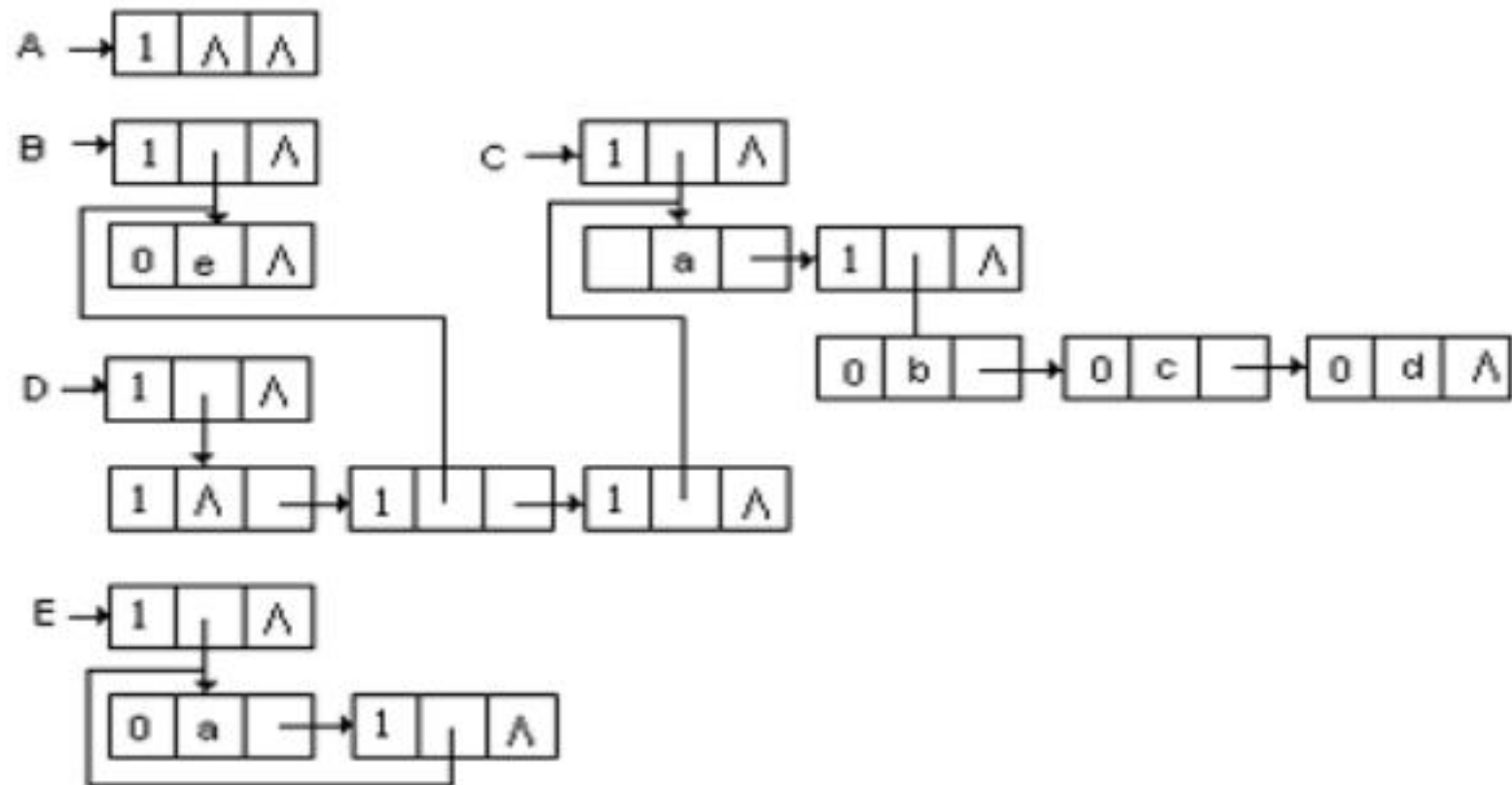
//表结点的表头指针

```
    };
```

```
    struct GLNode *tp;
```


//指向下一个元素结点

```
}*GList;
```



本章学习要点

1. 了解数组的两种存储表示方法，并掌握数组在以行为主的存储结构中的地址计算方法。
2. 掌握对特殊矩阵进行压缩存储时的下标变换公式。
3. 了解稀疏矩阵的两类压缩存储方法的特点和适用范围，领会以三元组表示稀疏矩阵时进行矩阵运算采用的处理方法。
4. 掌握广义表的结构特点及其存储表示方法，可根据自己的习惯熟练掌握任意一种结构的链表。

- 
- 习题
 - 5.7, 5.18, 5.19
 - 5.10(5)(7), 5.11(4)(6), 5.12(1), 5.13(1),