

# Array

## 概念

- 数组：由一组名字相同、下标不同的变量构成
- 特点：
  - 各个元素有统一的类型
  - 下标有固定的上界、下界；定义后，维数、维界不改变
  - 基本操作简单
    - 初始化、销毁、存取元素、改变元素值
- 二维数组： $A_{m \times n}$ : m行，n列
- !!! N 维数组的特点：n个下标，可以看作由若干个  $n-1$  维数组组成的线性表

### N维数组的数据类型定义

$n\_ARRAY = (D, R)$  其中：

数据对象： $D = \{a_{j_1, j_2, \dots, j_n} \mid j_i \text{ 为数组元素的第 } i \text{ 维下标}, a_{j_1, j_2, \dots, j_n} \in \text{Elemset}\}$

数据关系： $R = \{R_1, R_2, \dots, R_n\}$

$$R_i = \{ \langle a_{j_1, j_2, \dots, j_i, \dots, j_n}, a_{j_1, j_2, \dots, j_{i+1}, \dots, j_n} \rangle \mid a_{j_1, j_2, \dots, j_i, \dots, j_n}, a_{j_1, j_2, \dots, j_{i+1}, \dots, j_n} \in D \}$$

基本操作：构造数组、销毁数组、读数组元素、写数组元素

## 数组的顺序存储表示和实现

- 按照次序将数组元素排成一列序列
  - 例如：按列存储、按行存储……
- 注意：
  - 若规定好了次序，则数组中任意一个元素的存放地址便有规律可寻，可形成地址计算公式；
  - 约定的次序不同，则计算元素地址的公式也有所不同；
  - C和PASCAL中一般采用行优先顺序；FORTRAN采用列优先。
- 求地址：
  - 开始结点
  - 维数、每个维的上下界
  - 每个元素占的单元数

补充：计算二维数组元素地址的通式  
设一般的二维数组是 $A[c_1..d_1, c_2..d_2]$ ，这里 $c_1, c_2$ 不一定是0或1

则行优先存储时的地址公式为：

$$LOC(a_{ij}) = LOC(a_{c_1, c_2}) + [(i-c_1)*(d_2-c_2+1) + (j-c_2)] * L$$

数组基址

$a_{ij}$ 之前的行数

总列数，即第2维长度

$a_{ij}$ 本行前面的元素个数

单个元素长度

二维数组列优先存储的通式为：

$$LOC(a_{ij}) = LOC(a_{c_1, c_2}) + [(j-c_2)*(d_1-c_1+1) + (i-c_1)] * L$$

激活 Wi  
转到“设置”

## 练习题：求地址

**【例】(软考题)**：一个二维数组A，行下标的范围是1到6，列下标的范围是0到7，每个数组元素用相邻的6个字节存储，存储器按字节编址。那么，这个数组的体积是\_\_\_\_\_个字节。

行下标从1到6，则共6行；

列下标从0到7，则共8列；

每个占6字节，则共： $(6-1+1) * (7-0+1) * 6 = 288$

**【例】**已知二维数组 $A_{m,m}$ 按行存储的元素地址公式是：

$Loc(a_{ij}) = Loc(a_{11}) + [(i-1)*m + (j-1)] * K$ ，请问按列存储的公式相同吗？

$$Loc(a_{ij}) = Loc(a_{11}) + [(j-1)*m + (i-1)] * K$$

**【例】(00年计算机系考研题)**：设数组 $a[1..60, 1..70]$ 的基地址为2048，每个元素占2个存储单元，若以列序为主序顺序存储，则元素 $a[32,58]$ 的存储地址为\_\_\_\_\_。

$$\begin{aligned} Loc(a[32][58]) &= Loc(a[1][1]) + 2 * [60 * (58-1) + (32-1)] \\ &= 2048 + 2 * (3420 + 31) \\ &= 2048 + 6902 \\ &= 8950 \end{aligned}$$

## 三维数组的地址求法

已知： $A[x_0 \dots x_n][y_0 \dots y_n][z_0 \dots z_n]$

已知： $A[x][y][z]$ 的地址 $base$ ，且单位长度 $t$

求 $A[i][j][k]$

$$A[i][j][k] = base + t * [(i-x) * (x_n - x_0 + 1) * (y_n - y_0 + 1) * (z_n - z_0 + 1) + (j-y) * (z_n - z_0 + 1) + (k-z)]$$

## 顺序存储表示

```
#define MAX_ARRAY_DIM 8 // 假设维数8
typedef struct{
    ElemtType *base; // 数组元素基址
    int dim; // 数组维数
    int *bound; // 数组各维长度信息保存区基址
    int *constants; // 数组映像函数常量的基址
}Array;
```

## 补充：数组的链式存储方式

- 用带行指针向量的单链表来表示。

## 矩阵的压缩存储

- 压缩存储
  - 若多个数据元素的 值 都相同，则只分配一个元素值的存储空间，且零元素不分配存储空间
- 特点：对称、对角、三角、稀疏矩阵
- 稀疏矩阵：非零元素的个数较少（一般小于 5%）

## 稀疏矩阵的压缩存储

- 将每个非零元素用一个 3元组  $(i, j, a_{ij})$  里表示
  - $i$ : 行下标
  - $j$ : 列下标
  - $a_{ij}$ : 元素值

### 法一：线性表

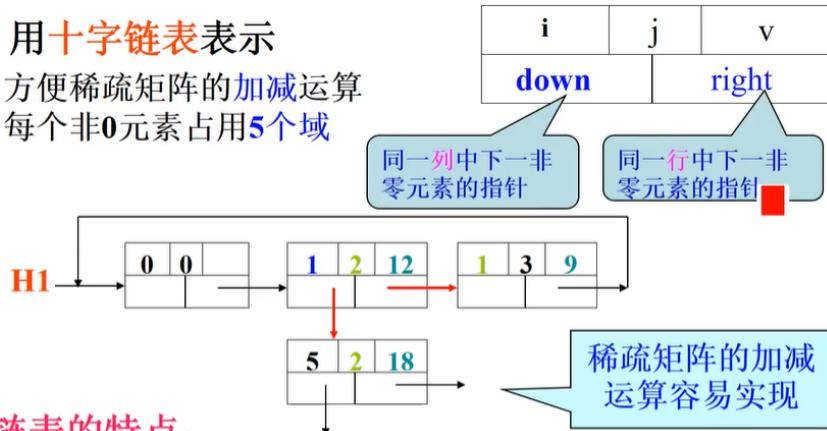
法1：用线性表表示：

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

### 法二：十字链表

## 法2：用十字链表表示

用途：方便稀疏矩阵的加减运算  
方法：每个非0元素占用5个域



## 十字链表的特点：

- ① 每行非零元素链接成带表头结点的循环链表；
  - ② 每列非零元素也链接成带表头结点的循环链表。
- 则每个非零元素既是行循环链表中的一个结点；又是列循环链表中的一个结点，即呈十字链状。

激活 Win

## 法三：三元数组矩阵

### 法3：用三元组矩阵表示：

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

0	12	9	0	0	0
0	0	0	0	0	0
-3	0	0	0	0	14
0	0	24	0	0	0
0	18	0	0	0	0
15	0	0	-7	0	0

注意：为更可靠描述，通常再加一行“总体”信息：即总行数、总列数、非零元素总个数

稀疏矩阵压缩存储的缺点：

将失去随机存取功能！

激活 V  
转到设置

- 针对具体问题、逻辑关系，选择是否压缩、如何压缩
- 失去随机存取的功能：如何解决？

## 法四：带辅助向量的三元组

- 用途：更高效地访问
- 方法：增加2个辅助向量
  - 记录 每行 非0个数：NUM[i]
  - 记录 每行 第一个非0元素在 三元组 中的行号：POS[i]

```
pos[1]=1;
pos[i]=pos[i-1] + num[i-1];
```

**法4：用带辅助向量的三元组表示。**

用途：便于高效访问稀疏矩阵中任一非零元素。

方法：增加2个辅助向量：

- ① 记录每行非0元素个数，用NUM(i)表示；
- ② 记录稀疏矩阵中每行第一个非0元素在三元组中的行号，用POS(i)表示。

i	1	2	3	4	5	6
NUM(i)	2	0	2	1	1	2
POS(i)	1	3	3	5	6	7

POS(i)如何计算？

$$POS(1)=1$$

$$POS(i)=POS(i-1)+NUM(i-1)$$

0	12	9	0	0	0
0	0	0	0	0	0
-3	0	0	0	14	0
0	0	24	0	0	0
0	18	0	0	0	0
15	0	0	-7	0	0

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

激活 W

## 稀疏矩阵的操作

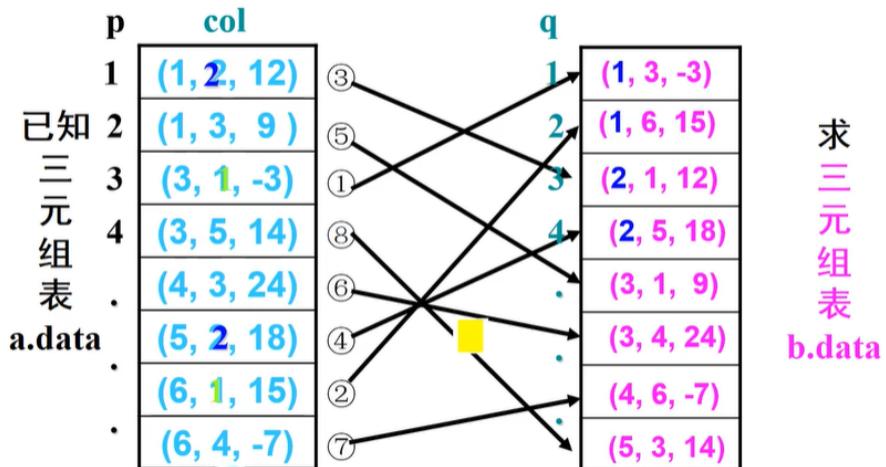
### 矩阵转置

- 已知：原三元组
- 求解：转置后的三元组
- 步骤
  1. 每个元素行下标、列下标互换
  2. T总行数(mu)和总列数(nu)交换
  3. 重排三元组内个元素顺序

### 法1.压缩转置

#### 方法1：压缩转置

思路：反复扫描a表（记为a.data）中的列序，从j=1~n依次进行转置。

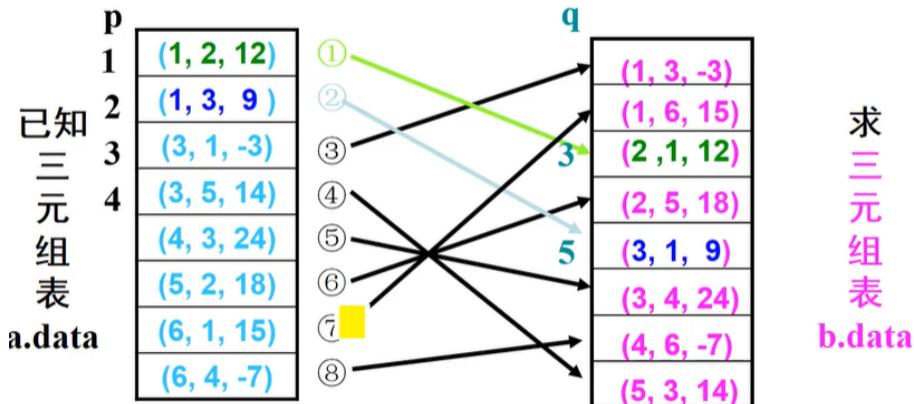


每个元素的列分量表示为：**a.data[p].j**

## 法2. 快速压缩转置

### 方法2 快速转置

思路：依次把a. data中的元素直接送入b. data的恰当位置上（即M三元组的p指针不回溯）。



### 设计思路：辅助向量

#### 设计思路：

如果能预知M矩阵每一列（即T的每一行）的非零元素个数，  
又能很快得知第一个非零元素在b. data中的位置，  
则扫描a.data时便可以将每个元素准确定位（因已知若干参考点）



请注意a. data特征：每列首个非零元素必定先被扫描到。

技巧：为实现转置运算，应当按列生成 M 矩阵三元组表的  
两个辅助向量，让它携带每列的非零元素个数 NUM(i)  
以及每列的第一个非零元素在三元组表中的位置 POS(i)  
等信息。

辅助向量的样式：

i	1	2	3	4	5	6
NUM(i)	2	2	2	1	1	0
POS(i)	1	3	5	7	8	9

激活-Wir

令：  
 $\begin{cases} \text{M矩阵中的列变量用} \text{col} \text{表示;} \\ \text{num[ col ]：存放M中第} \text{col} \text{列中非0元素个数} \\ \text{cpos[ col ]：存放M中第} \text{col} \text{列的第一个非0元素的位置} \end{cases}$   
 (即b. data中待计算的“恰当”位置所需参考点)

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

计算式： $cpos(1) = 1$   
 $cpos[ col ] = cpos[ col-1 ] + num[ col-1 ]$

讨论：求出按列优先的辅助向量后，如何实现快速转置？

由a. data中每个元素的列信息，可以直接从辅助向量cpos[col]中查出在b. data中的“基准”位置，进而得到当前元素之位置。

p/t	col
1	(1, 2, 12)
2	(1, 3, 9)
3	(3, 1, -3)
4	(3, 5, 14)
.	(4, 3, 24)
.	(5, 2, 18)
.	(6, 1, 15)
.	(6, 4, -7)

激活  
转到“设置”  
7

## 分析时间复杂度

快速转置算法的效率分析：

- 与常规算法相比，附加了生成辅助向量表的工作。增开了2个长度为列长的数组(num[]和cpos[])。
- 从时间上，此算法用了4个并列的单循环，而且其中前3个单循环都是用来产生辅助向量表的。

```
for(col = 1; col <= M.nu; col++){}; 循环次数 = nu (列数) ;
for( i = 1 <= M.tu; i ++ ){}; 循环次数 = tu (非0元素个数) ;
for(col = 2; col <= M.nu; col++){}; 循环次数 = nu;
for( p = 1; p <= M.tu ; p ++ ){}; 循环次数 = tu;
```

该算法的时间复杂度 =  $nu + tu + nu + tu = O(nu + tu)$

讨论：最恶劣情况是矩阵中全为非零元素，此时 $tu = nu * mu$ 而此时的时间复杂度也只是 $O(mu * nu)$ ，并未超过传统转置算法的时间复杂度。

小结：传统转置： $O(mu * nu)$  压缩转置： $O(mu * tu)$

压缩快速转置： $O(nu + tu)$

稀疏矩阵相乘的算法略，  
见教材P101-103

增设辅助向量，牺牲空间  
效率换取时间效率。

激活 Wi  
转到“设置”  
9

## 数组和广义表( Arrays & Lists )

- 特点：特殊的线性表
- 元素的值，并非原子类型，可再分解
- 所有元素仍属于同一类型

# 广义表的定义

## 5.4 广义表的定义

### 1、定义：

广义表是线性表的推广，也称为列表（lists）

记为：  $LS = ( a_1, a_2, \dots, a_n )$

广义表名 表头(Head) 表尾 (Tail)

n是表长

在广义表中约定：

①用小写字母表示原子类型，用大写字母表示列表。

②第一个元素是表头，而其余元素组成的表称为表尾；

特别提示：任何一个非空表，表头可能是原子，也可能是列表；但表尾一定是列表！

激活 Wi  
转到“设置”

11

- 小写字母：原子类型
- 大写字母：列表
- 第一个元素：表头
- 其余元素组成的表：表尾

### 广义表与线性表区别

- 广义表：元素既可以是原子，也可以是列表
- 线性表：每个元素都是原子类型，且相同

### 特点

- 次序性：前驱、后继
- 长度：表中个数
- 深度：括号的重数
- 递归：自己作为自己的子表
- 共享：略

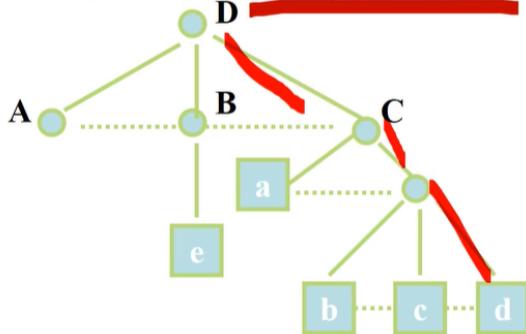
### 例1：求广义表的长度

- 1)  $A = ()$   $n=0$ , A是空表
- 2)  $B = (e)$   $n=1$ , 表中元素e是小写，原子
- 3)  $C = (a, (b, c, d))$   $n=2$ , a为原子, (b, c, d) 为子表
- 4)  $D = (A, B, C)$   $n=3$ , 3个元素都是子表
- 5)  $E = (a, E)$   $n=2$ , a为原子, E为子表

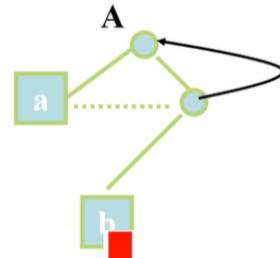
### 例2：求深度

**例2：**试用图形表示下列广义表。  
 (设●代表子表, ■代表元素)

$$\textcircled{1} \quad D = (A, B, C) = (( ), (e), (a, (b, c, d)))$$



$$\textcircled{2} \quad A = (a, (b, A))$$



①的长度为3，  
 深度为3  
 深度=括号的重数=●结点的层数

②的长度为2，  
 深度为 $\infty$

### 例3：GetHead, GetTail

**例：**求下列广义表操作的结果 (严题集5.10②)

$$1. \text{GetTail } [(b, k, p, h)] = \underline{(k, p, h)};$$

$$2. \text{GetHead } [(a, b), (c, d)] = \underline{(a, b)};$$

$$3. \text{GetTail } [(a, b), \underline{(c, d)}] = \underline{((c, d))};$$

$$4. \text{GetTail } [\text{GetHead } [(a, b), (c, d)]] = \underline{(b)};$$

$$5. \text{GetTail } [(e)] = \underline{();} \quad ; \quad \boxed{(a, b)}$$

$$6. \text{GetHead } [( )] = \underline{()}. \quad \blacksquare$$

$$7. \text{GetTail } [( )] = \underline{()}. \quad \blacksquare$$

• 备注：

- Head指的第一个元素
- Tail为除第一个以外的部分，组成的列表（看第3题）

## 广义表的存储结构

## 5.5 广义表的存储结构

```
// ----- 广义表的头尾链表存储表示 ----- //
typedef enum {ATOM, LIST} Elemtag; // ATOM==0: 原子; LIST==1: 表
typedef struct GLNode {
    Elemtag tag;           // 公共部分, 用于区分原子节点和表节点
    union {                // 原子节点和表节点的联合部分
        AtomType atom;     // atop是原子节点的值域, AtomType由用户定义
        struct {
            struct GLNode *hp, *tp;
        }ptr;               // ptr是表节点的指针域, ptr.hp和ptr.tp分别指向表头和表尾
    };
} *GList
```

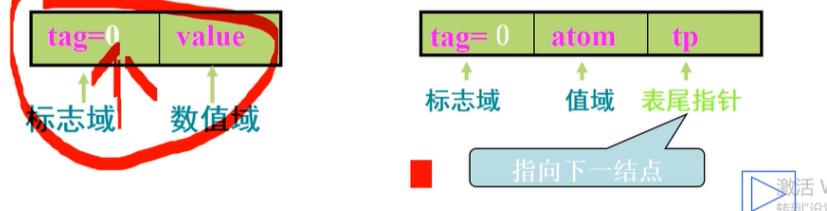
- 链表

由于广义表的元素可以是不同结构（原子或列表），难以用顺序存储结构表示，通常用链式结构，每个元素用一个结点表示。

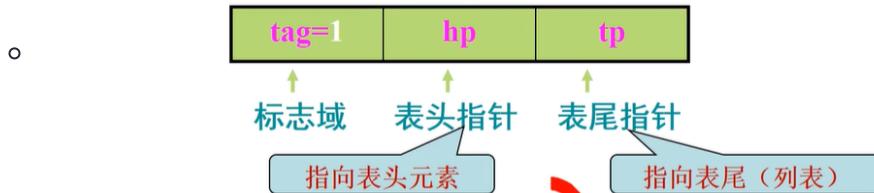
注意：列表的“元素”还可以是列表，所以结点可能有三种形式

1. 原子结点：表示原子，可设2个域或3个域，依习惯而选。

- 法1：标志域，数值域
- 法2：标志域、值域、表尾指针



2. 表结点：表示列表，若表不空，则可分解为表头和表尾，用3个域表示：标志域，表头指针，表尾指针。



- 例：

例：①  $A = ()$      $A = \text{NULL}$      $\rightarrow$

