

# 数据结构

## 5 数组和广义表

董洪伟 陈聪 周世兵

联系电话: 13812529213

E-mail: worldguard@163.com

# 主要内容

- 数组的类型定义
- 数组的顺序表示和实现
- 矩阵的压缩存储
  - 特殊矩阵
  - 稀疏矩阵
- 广义表的定义
- 广义表的存储结构

# 数组的类型定义

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \quad \begin{aligned} a_1 &= (a_{11}, a_{12}, \dots, a_{1n}) \\ a_2 &= (a_{21}, a_{22}, \dots, a_{2n}) \\ &\dots\dots\dots \\ a_n &= (a_{n1}, a_{n2}, \dots, a_{nn}) \end{aligned} \quad A = \begin{pmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{pmatrix}$$

$$\beta_1 = \begin{pmatrix} a_{11} \\ a_{21} \\ \dots \\ a_{n1} \end{pmatrix} \quad \beta_2 = \begin{pmatrix} a_{12} \\ a_{22} \\ \dots \\ a_{n2} \end{pmatrix} \quad \dots\dots\dots \beta_n = \begin{pmatrix} a_{1n} \\ a_{2n} \\ \dots \\ a_{nn} \end{pmatrix} \quad A = (\beta_1 \ \beta_2 \ \dots\dots \ \beta_n)$$

# 数组的类型定义：数组的特点

1) 数组 是（一组下标，值）的集合。

每一个数据元素由唯一的一组下标来标识。因此，在数组上不能做插入、删除数据元素的操作。通常在各种高级语言中数组一旦被定义，每一维的大小及上下界都不能改变。

2) 数组可视为线性表的推广：一维数组就是数据元素的一个线性表，二维数组可以看作“数据元素是一维数组”的一维数组，三维数组可以看作“数据元素是二维数组”的一维数组，依此类推。

3) 多维数组的逻辑结构不是非线型而是图型结构

4) 数组具有根据下标的读、写操作

# 数组的类型定义

ADT Array {

数据对象:  $D = \{a_{j_1 j_2 \dots j_n} \mid j_i = 0, \dots, b_i - 1, i=1, 2, \dots, n, n(>0) \text{ 称为数组的维数, } b_i \text{ 是数组第 } i \text{ 维的长度, } j_i \text{ 是数组元素的第 } i \text{ 维下标, } a_{j_1 j_2 \dots j_n} \in \text{ElemSet} \}$

数据关系:  $R = \{R_1, R_2, \dots, R_n\}$

$$R_i = \{0 \leq j_i \leq b_i - 1, 0 \leq j_k \leq b_k - 1, 1 \leq k \leq n, k \neq i\}$$

基本操作:

`InitArray(&A, n, bound1, ..., boundn);`

// 由维数  $n$  和各维长度构造相应的数组  $A$

`DestroyArray(&A);` // 销毁数组  $A$ 。

# 数组的类型定义

**Value**(A, &e, index1, ..., indexn)

//查询数组A下标为index1, ..., indexn的数组元素的值

**Assign**(&A, e, index1, ..., indexn)

//对数组A下标为index1, ..., indexn的数组元素进行赋值

} ADT Array

# 数组的顺序表示和实现

- 类型特点：
  - 1) 没有插入、删除操作，适合顺序存储；
  - 2) 数组是多维的结构，而存储空间是一个一维的结构。
- 两种顺序映象的方式：
  - 1) 以行序为主序(低下标优先)；
$$A_{00} \ A_{01} \ A_{10} \ A_{11}$$
$$A_{000}, A_{001}, A_{010}, A_{011}, A_{100}, A_{101}, \dots, A_{211}$$
  - 2) 以列序为主序(高下标优先)；
$$A_{00} \ A_{10} A_{10} A_{11}$$
$$A_{000}, A_{101}, A_{200}, A_{010}, A_{110}, A_{210}, A_{001} \dots, A_{211}$$

# 一维数组

- 一维数组在内存中的存放

- 假设NumberA的首地址为12

- 第 $i$ 个元素的地址 = 首地址 +  $i$  \* 数据类型的大小
    - 假设一个单元的大小是4个字节

下标:

0	1	2	3
5	3	11	2

值:

地址:

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

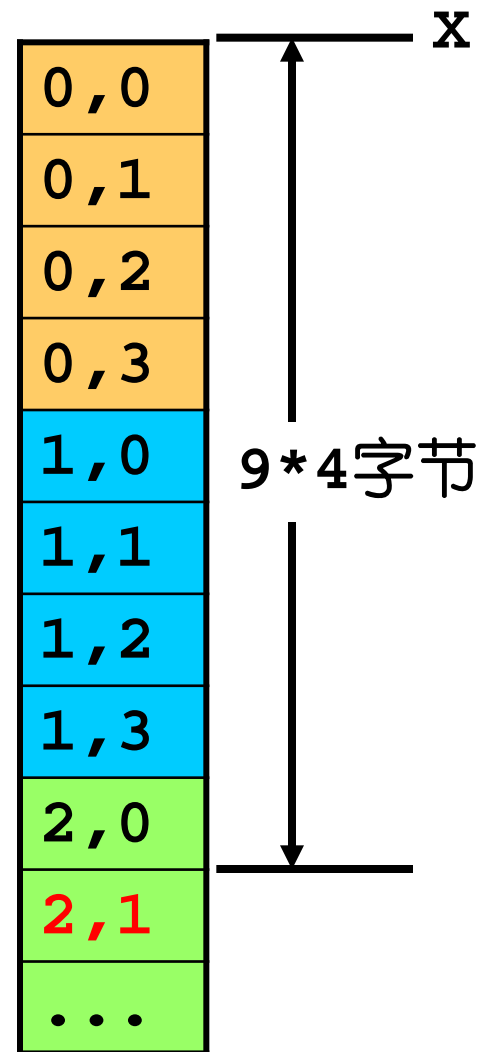


# 二维数组

- 二维数组在内存中的存放
  - 二维数组的“二维”是逻辑上的
  - 内存永远是线性编址

0, 0	0, 1	0, 2	0, 3
1, 0	1, 1	1, 2	1, 3
2, 0	2, 1	2, 2	2, 3
3, 0	3, 1	3, 2	3, 3
4, 0	4, 1	4, 2	4, 3

“二维数组”



内存中的存放方式

# 二维数组

- 按照行优先， $\text{Data}[i][j]$ 是第几个元素？
  - $\text{Data}[i][j]$ 前面有 $i$ 行， $j$ 列
  - 所以 $\text{Data}[i][j]$ 是第“ $i * \text{列数} + j$ ”个元素(从0开始)
  - 所以 $\text{Data}[i][j]$ 的地址=首地址+ $(i * \text{列数} + j) * \text{数据类型的大小}$

	0, 0	0, 1	0, 2	0, 3	
	1, 0	1, 1	1, 2	1, 3	
i	2, 0	2, 1	2, 2	2, 3	
	3, 0	3, 1	3, 2	3, 3	
	4, 0	4, 1	4, 2	4, 3	
		j			

# 二维数组

- 二维数组A中任一元素 $a_{ij}$ 在内存中的位置

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

- N维数组A中任一元素在内存中的位置

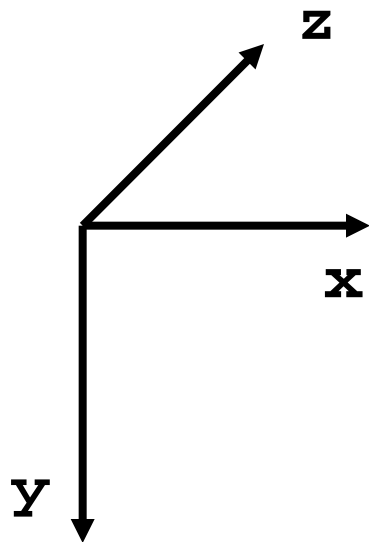
$$LOC(j_1, j_2, \dots, j_n) = LOC(0, 0, \dots, 0)$$

$$+ (b_2 \times \dots \times b_n \times j_1 + b_3 \times \dots \times b_n \times j_2$$

$$+ \dots + b_n \times j_{n-1} + j_n)L$$

$$= LOC(0, 0, \dots, 0) + \left( \sum_{i=1}^{n-1} j_i \prod_{k=i+1}^n b_k + j_n \right) L$$

# N维数组



0,0,0	0,1,0	0,2,0	0,3,0
1,0,0	1,1,0	1,2,0	1,3,0
2,0,0	2,1,0	2,2,0	2,3,0
3,0,0	3,1,0	3,2,0	3,3,0
4,0,0	4,1,0	4,2,0	4,3,0

## 下标--地址映射--n维情形

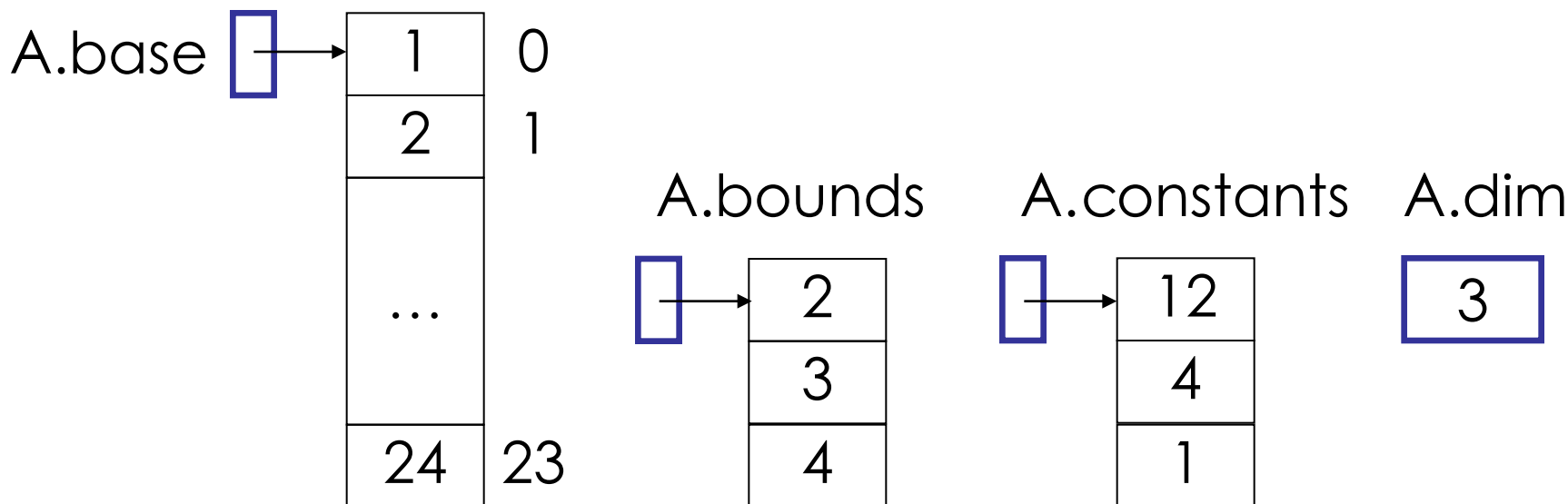
设  $n$  维数组  $A_{b_1 \times b_2 \times \dots \times b_n}$ , 则:

$$\begin{aligned} & \text{LOC}(j_1, j_2, \dots, j_n) \\ &= \text{LOC}(0, 0, \dots, 0) \\ & \quad + \underbrace{j_1 * b_2 * \dots * b_n * L}_{C_1} + \underbrace{j_2 * b_3 * \dots * b_n * L}_{C_2} + \dots + \underbrace{j_{n-1} * b_n * L}_{C_{n-1}} + \underbrace{j_n * L}_{C_n} \\ &= \text{LOC}(0, 0, \dots, 0) + \sum_{i=1}^n (c_i * j_i) \end{aligned}$$

其中,  $c_n = L$ ,  $c_{i-1} = c_i * b_i$ ,  $1 < i \leq n$ , 称求址常量。

# 数组的表示与实现


```
typedef struct{  
    ElemType    *base; //数组的基地址  
    int dim;        //维数  
    int *bounds;    //各维的界  
    int *constants; //地址函数的系数,即ci  
}Array;
```



# 数组的表示与实现

基本操作的实现：

```
bool  InitArray(Array &A,int dim, ...);  
bool  Destory(Array &A);  
bool  Value(Array &A,ElemType &e, ...);  
bool  Assign(Array &A, ElemType e, ...);
```



int b1,int b2,...



int j1,int j2, ...

## 数组的表示与实现: *InitArray*

```
bool InitArray(Array &A,int dim, ...)  
{  
    if( dim <1 || dim >= Max_Array_Dim)  
        return false; //维数非法  
    A.dim = dim;  
    A.bounds = (int *)malloc(dim*  
                             sizeof(int));  
    if( !A.bounds )  
        return ERROR; //内存分配失败
```



## 数组的表示与实现: *InitArray*

```
int  elemtotal = 1;    //元素的总数
va_list ap; va_start(ap,dim);
for( int i=0; i<dim; ++i){
    A.bounds[i] = va_arg ( ap,int);
    if(A.bounds[i] <=0) return -1;
    elemtotal *= A.bounds[i] ;
}
va_end( ap );

A.base = (ElemType *)malloc(
    elemtotal *sizeof(ElemType));
if(!A.base) return ERROR;
```

## 数组的表示与实现: *InitArray*

```
A.constants=(int *)malloc( dim *
                           sizeof(int)); //计算Ci
if (!A.constants)
    return ERROR;           //存储分配失败
A.constants[dim-1]=1;      //cn=1
for(i= dim-2;i>=0;--i) //ci=ci+1*bi+1
    A.constants[i] = A.bounds[i+1]
                    *A.constants[i+1];
return OK;
}
```

## va...实现可变的参数表

```
void va_start (va_list param, lastfix);
```

param指向被调用函数的固定参数lastfix 后的可变参数列表。

```
type va_arg (va_list param, type);
```

从param所指处取出一个类型为type的参数并返回该值。每次调用va\_arg都会改变param值使得后续的参数值能被依次取出。

```
void va_end (va_list param);
```

param列表解析的结束。

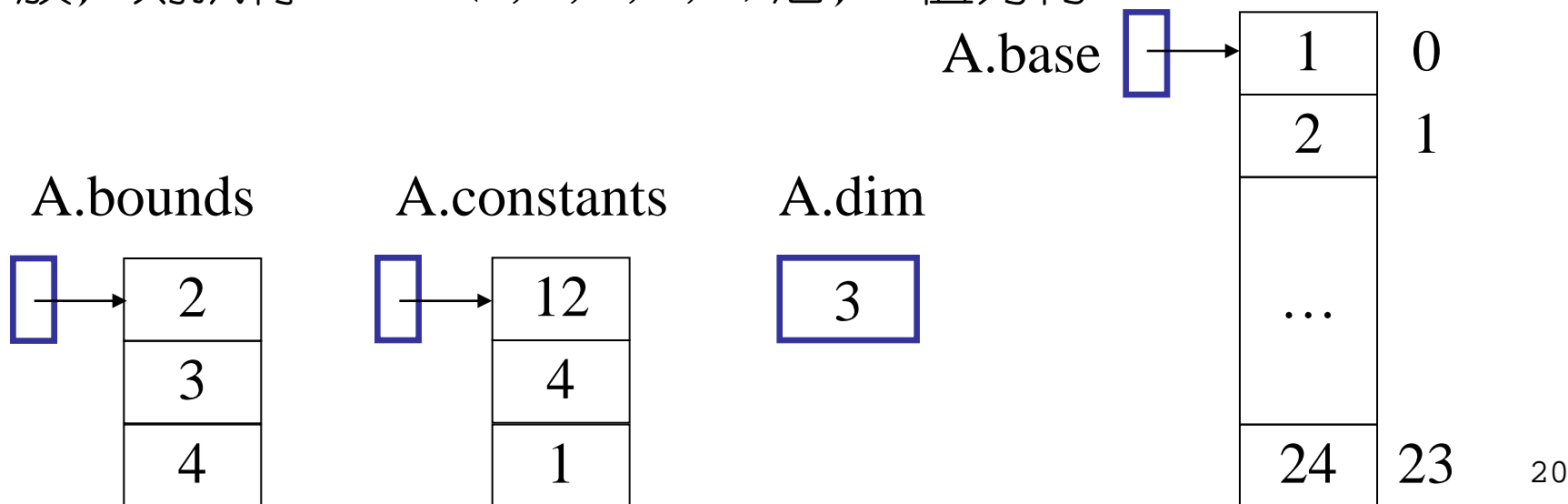
# 数组的表示与实现:取数组元素的值

## 【思路】

(1)按下标求数据元素在存储空间中的相对位置  $off = \sum_{i=1}^n (c_i * j_i)$

(2) $e = *(A.base + off)$ 。

【例】设数组  $A = (((1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12)), ((13, 14, 15, 16), (17, 18, 19, 20), (21, 22, 23, 24)))$  按行序存放，则执行  $Value(A, e, 0, 2, 1)$  后， $e$  值为何？



## 数组的表示与实现:取数组元素的值

```
bool Value(Array &A, ElemType &e, ...) {  
    int off=0; va_list ap;  
    va_start(ap, e); //使ap指向e后第一个参数  
    for(int i=0; i<A.dim; i++) {  
        int j=va_arg(ap, int); //j中保存当前下标  
        if(j<0 || j>=A.bounds[i])  
            return ERROR; //下标值非法  
        off+=A.constants[i]*j; //求ci*ji并累加  
    } //for  
    va_end(ap);  
}
```

## 数组的表示与实现:取数组元素的值

```
e=*(A.base+off); //用e返回元素值
```

```
return OK;
```

```
} //value
```

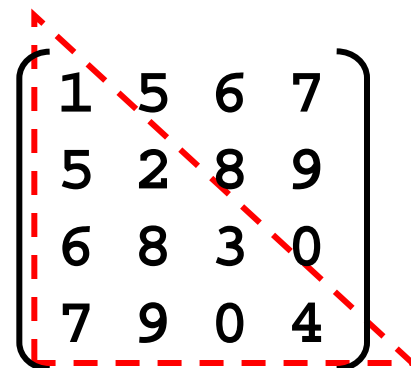
# 矩阵的压缩

- 特殊矩阵

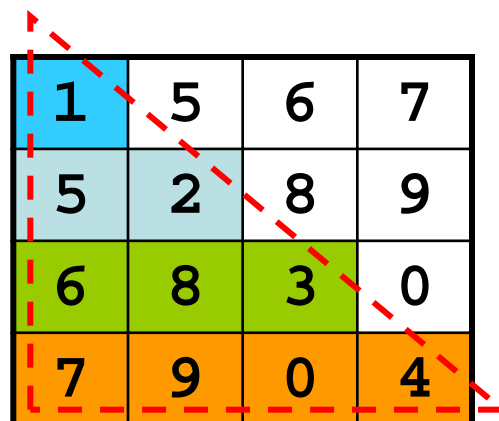
- 对称矩阵

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

- 可以将  $n^2$  个单元压缩为  $n(n+1)/2$  个:



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



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



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

- $k$  的含义：按行优先，是第  $k$  个（从 0 开始）

$$k = \begin{cases} \frac{i(i-1)}{2} + j - 1 & i \geq j \quad (\text{保存下三角}) \\ \frac{j(j-1)}{2} + i - 1 & i < j \quad (\text{保存上三角}) \end{cases}$$

$i=3$

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

$j=2$

$k=4$

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



# 矩阵的压缩

## - 公式的推导 (下三角)

- $i=3, j=2$
- 则前面有一个  $i-1$  行的完整三角形，共有元素  $(1+i-1)(i-1)/2 = i(i-1)/2$  个
- 另外，同一行，前面还有  $j-1$  个元素
- 所以， $k = i(i-1)/2 + j - 1$

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

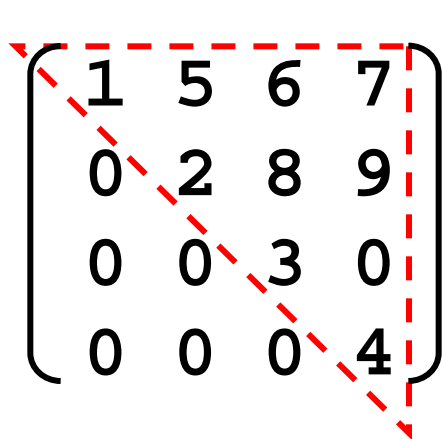
$i=3$

$j=2$

# 矩阵的压缩

## – 三角矩阵

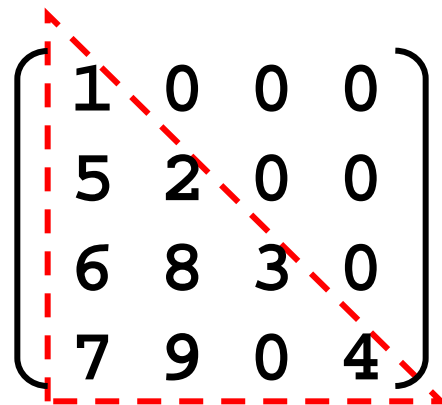
- 压缩方法和对称矩阵完全相同



A 4x4 matrix with a red dashed line from the top-left to the bottom-right, indicating the upper triangular part. The matrix is:

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

上三角



A 4x4 matrix with a red dashed line from the top-left to the bottom-right, indicating the lower triangular part. The matrix is:

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

下三角

# 矩阵的压缩

- 稀疏矩阵

$$\mathbf{A}_{9 \times 7} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

– 非零元素个数远远少于矩阵元素总数

# 矩阵的压缩

## • 稀疏矩阵的定义

- 设矩阵  $A_{m \times n}$  中有  $t$  个非零元素，若  $t$  远远小于矩阵元素的总数  $m \times n$ ，且非零元素的分布无规律，则称矩阵  $A$  为稀疏矩阵
- 为节省存储空间，应只存储非零元素
- 非零元素的分布一般没有规律，应在存储非零元素时，同时存储该非零元素的行下标  $row$ 、列下标  $col$ 、值  $value$
- 每一个非零元素由一个三元组唯一确定：
  - ( 行号  $row$ , 列号  $col$ , 值  $value$  )

# 矩阵的压缩

- 稀疏矩阵的压缩表示

$$\mathbf{A}_{9 \times 7} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

行数	列数	元素个数
↓	↓	↓
9	7	5
1	1	3
3	0	1
3	1	4
4	2	7
5	5	5
↑	↑	↑
行号	列号	元素值

# 稀疏矩阵的转置

- 三元组顺序表

```
#define MAXSIZE 12500
typedef struct{
    int i,j;           //行下标、列下标
    ElemType e;        //元素的值
}Triple;

typedef struct{
    Triple data[MAXSIZE+1];
    int mu,nu,tu;      //行数、列数、非0元素个数
}TSMatrix;
```

# 稀疏矩阵的转置

## – 矩阵的转置

- $T_{ij} = M_{ji}$

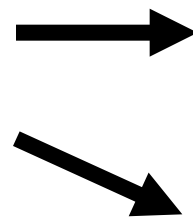
0	12	9	0	0	0
0	0	0	0	0	0
-3	0	0	0	0	14
0	0	24	0	0	0
0	18	0	0	0	0



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

- 行数和列数交换
- $i$ 、 $j$  的值相互交换
- 重排三元组之间的次序

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



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



## • 简单算法

```
Status TransposeSMatrix() {  
    T.mu = M.nu; T.nu = M.mn; T.tu = M.tu;  
    if(T.tu) {  
        q = 1;  
        for(col = 1; col <= M.nu; col ++)  
            for(p = 1; 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;  
}
```

复制总体信息

```

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

```

p扫描三元组M的  
如果p指向的记录  
的列下标=col

把M中的记录p复制  
到T中的记录q

p →

M		
i	j	v
5	6	8
1	2	12
1	3	9
3	1	-3
3	6	14

q →

T		
i	j	v
6	5	8
1	3	-3
2	1	12

col = 2

# 稀疏矩阵的转置：简单算法

- 简单算法的分析

- 稀疏矩阵转置算法复杂度 =  $O(nu \cdot tu)$
- 比较一般矩阵的转置算法：

```
for(col = 1; col <= nu; col ++)  
    for(row = 1; row <= mu; row ++)  
        T[col][row] = M[row][col];
```

- 其复杂度 =  $O(mu \cdot nu)$
- 如果 $tu$ 和 $mu \cdot nu$ 一个数量级，则稀疏矩阵转置算法复杂度= $O(mu \cdot nu^2)$
- 所以此算法只适用于 $tu \ll mu \cdot nu$ 时

# 稀疏矩阵的转置：简单算法

## • 简单算法的分析

- 简单算法的主要问题在于为了使得结果三元组表的记录按照行顺序排列，需要分别针对结果三元组表的每一行（也就是原三元组表的每一列），扫描整个三元组表，查找属于该行的所有记录
- 即对原三元组表的扫描需要反复进行，而不是只需一次
- 因此时间复杂度= $O(nu*tu)$

# 稀疏矩阵的转置：快速转置算法

- **关键：** 希望对原三元组表只需扫描一遍
- **问题：** 原三元组表中的一个三元组应该放在结果三元组表中的什么位置呢？

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



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

# 稀疏矩阵的转置：快速转置算法

- 分析：**原矩阵第1列只有一条记录，而三元组1是第一个遇到的来自第2列的非零元素，应该放在结果三元组表中的第2个位置

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

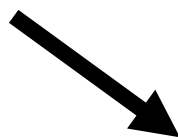


	i	j	v
0	6	5	6
1	1	3	-3
2	2	1	12
3	2	5	18
4	3	1	9
5	3	4	24
6	6	3	14

# 稀疏矩阵的转置：快速转置算法

- **类似的：** 第1列非零元素有1个，第2列非零元素有2个，而三元组2是第一个遇到的来自第3列的记录，应该放在结果三元组表中的第4个位置

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



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

# 稀疏矩阵的转置：快速转置算法

## • 算法的实现

- 数组`num[col]`：原矩阵第`col`列中，非零元素的个数
- 数组`cpot[col]`：原矩阵第`col`列中，第1个非零元素在结果三元组表中的位置
- 显然有：

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



# 稀疏矩阵的转置：快速转置算法

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

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

## • 快速转置算法

```
Status FastTransposeSMatrix() {  
    T.mu = M.nu; T.nu = M.mn; T.tu = M.tu;  
    if(T.tu) {  
        for(col = 1; col <= M.nu; col ++)  
            num[col] = 0;                //初始化num  
        for(t = 1; t <= M.tu; t ++)  
            num[M.data[t].j] ++;        //统计每列元素个数  
        //计算T中每行开始的位置  
        cpot[1] = 1;  
        for(col = 2; col <= M.nu; col ++)  
            cpot[col] = cpot[col-1] + num[col-1];  
        ...  
    }
```

## • 快速转置算法（接上页）

```
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]++;  
}  
}  
return OK;  
}
```

原三元组表只需要扫描一遍

得到当前三元组的列号

得到当前三元组的放置位置

拷贝三元组

当前列的起始位置加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]++; }

```

**p** →

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

**q** →

	i	j	v
0	6	5	6
1	1	3	-3
2	2	1	12
3			
4	3	1	9
5			
6	6	3	14

**col**  
↓

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

# 稀疏矩阵的转置：快速转置算法

## • 算法分析

- 整个算法有4个for循环
- 循环次数分别为nu、tu、nu和tu次
- 因此时间复杂度= $O(nu+tu)$
- 当tu和mu\*nu同一个数量级时,  
时间复杂度= $O(mu*nu)$
- 和一般矩阵转置算法的时间复杂度相同

# 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：**

$R1 = \{ \langle e_{i-1}, e_i \rangle \mid e_{i-1}, e_i \in D, 2 \leq i \leq n \}$

**基本操作：** 

**} ADT Glist**

- 结构的创建和销毁

**InitGLList(&L);**

**DestroyGLList(&L);**

**CreateGLList(&L, S);**

**CopyGLList(&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());**



广义表是递归定义的线性结构，

$$LS = ( \alpha_1, \alpha_2, \dots, \alpha_n )$$

其中： $\alpha_i$  或为原子 或为广义表

例如：  $A = ( )$

$B = (e)$

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

$D = (A, B, C)$

$E = (a, E) = (a, (a, (a, \dots) ) )$



广义表是一个多层次的线性结构

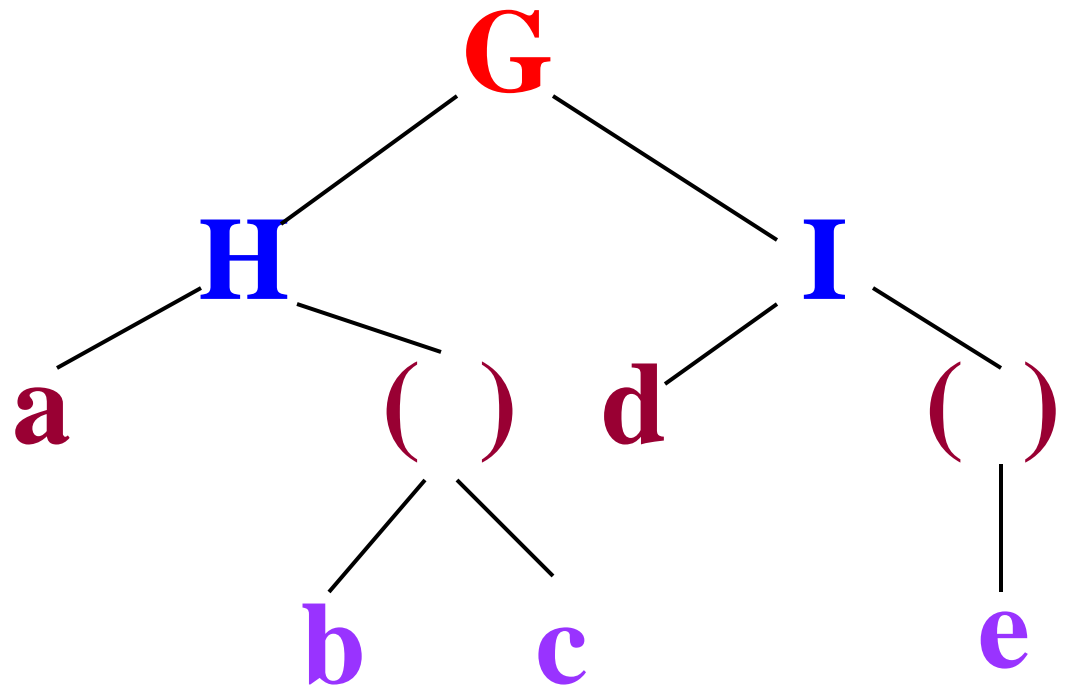
例如：

$G = (H, I)$

其中：

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

$I = (d, (e))$



## 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**定义为所含括弧的重数；  
    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。

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**  $\text{GetHead}(LS) = \alpha_1$  和

**表尾**  $\text{GetTail}(LS) = (\alpha_2, \dots, \alpha_n)$  两部分。

例如:  $D = (A, B, C) = (( ), (e), (a, (b, c, d)))$

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

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

$\text{GetHead}(((b, c))) = (b, c)$        $\text{GetTail}(((b, c))) = ( )$

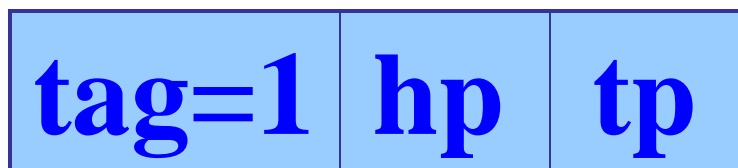
$\text{GetHead}((b, c)) = b$        $\text{GetTail}((b, c)) = (c)$

$\text{GetHead}((c)) = c$        $\text{GetTail}((c)) = ( )$

# 广义表的存储结构

## 1. 通常采用头、尾指针的链表结构

表结点:



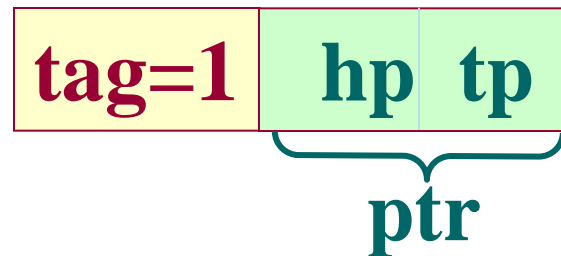
原子结点:



## 广义表的头尾链表存储表示:

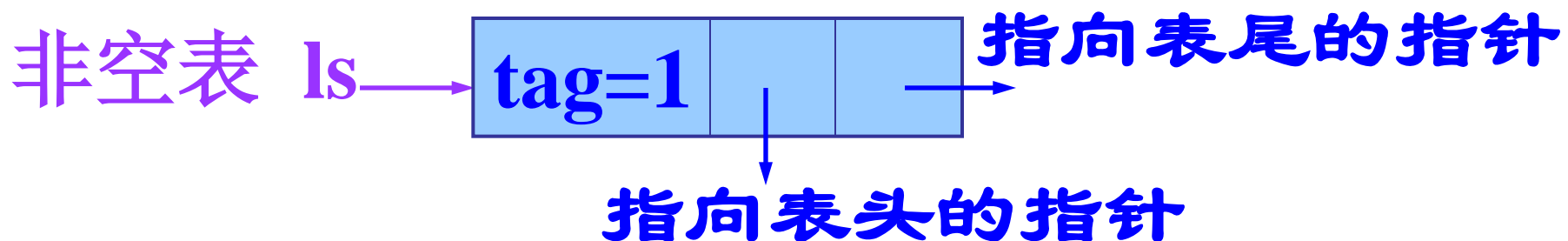
```
typedef enum {ATOM, LIST} ElemTag;  
    // ATOM==0:原子, LIST==1:子表  
typedef struct GLNode {  
    ElemTag tag; // 标志域  
    union{  
        AtomType atom;    // 原子结点的值域  
        struct {struct GLNode *hp, *tp;} ptr;  
    };  
} *GList
```

表结点



## 表头、表尾分析法：

空表  $ls = \text{NIL}$



若表头为原子，则为 

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

否则，依次类推。

# 广义表的存储结构

## 2. 扩展线性链表结构

表结点:

<b>tag=1</b>	<b>hp</b>	<b>tp</b>
--------------	-----------	-----------

原子结点:

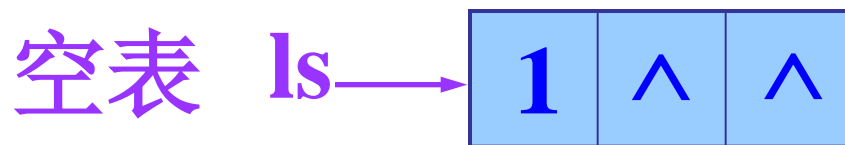
<b>tag=0</b>	<b>atom</b>	<b>tp</b>
--------------	-------------	-----------

## 广义表的扩展线性链表存储表示:

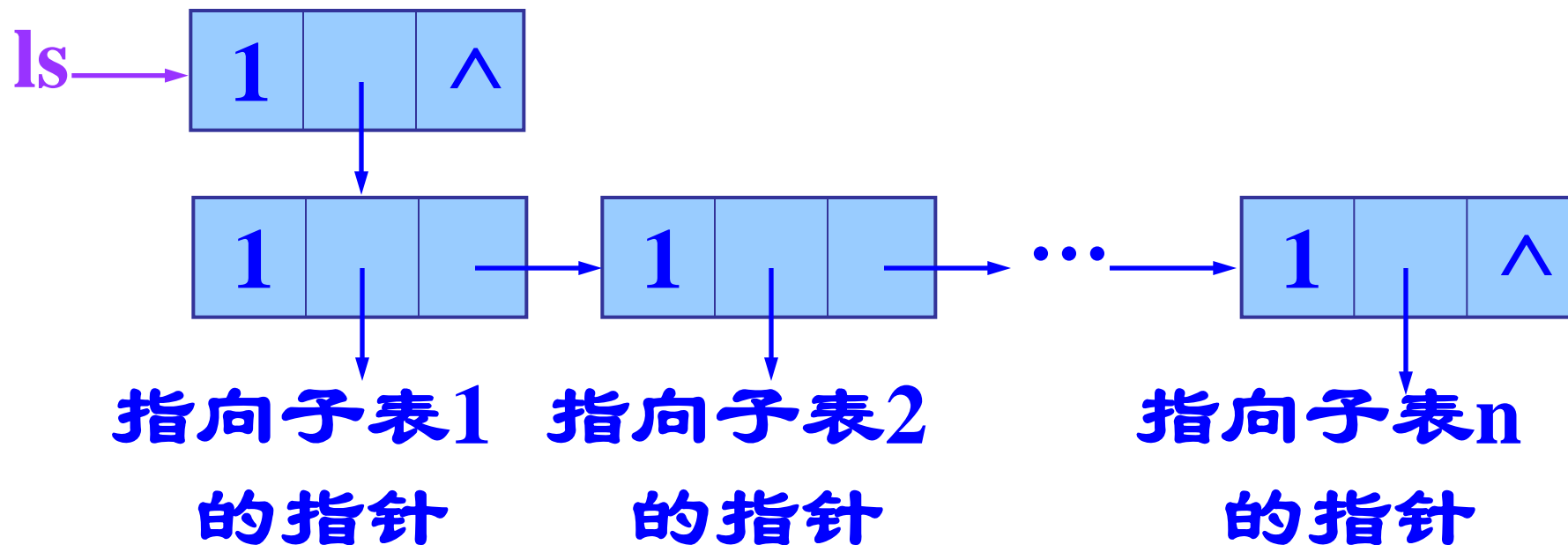
```
typedef enum {ATOM, LIST} ElemTag;  
    // ATOM==0:原子, LIST==1:子表  
typedef struct GLNode {  
    ElemTag tag; // 标志域  
    union{  
        AtomType atom; // 原子结点的值域  
        struct GLNode *hp; // 表结点的表头指针  
    };  
    struct GLNode *tp; // 指向下一个元素结点  
} *GList;
```



# 子表分析法



非空表



若子表为原子，则为



否则，依次类推。

# 本章小结

- 数组的类型定义
  - 重点掌握数组在内存中的存放规则
- 特殊矩阵的压缩
  - 对称矩阵、三角矩阵
  - 只需要保存上、下三角
  - 公式应该理解记忆
- 稀疏矩阵的压缩
  - 三元组法
  - 稀疏矩阵的转置算法

# 本章小结

- 广义表的定义
  - 掌握广义表的结构特点
- 广义表的存储结构
  - 头尾链表存储表示
  - 扩展线性链表存储表示