



# 数据结构与算法

# Data Structure and Algorithm

## 第5章 数组与串



# 目 录

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

5.4 广义表的定义

5.5 广义表的存储结构



## 5.1 数组 的定义

### 5.1.1 多维数组的概念

#### 1. 一维数组

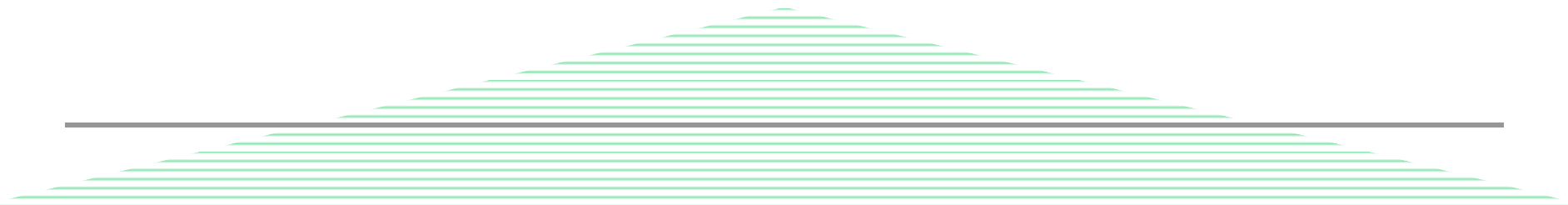
---

一维数组可以看成是一个线性表，它在计算机内是存放在一块连续的存储单元中，适合于随机查找。这在第二章的线性表的顺序存储结构中已经介绍。

#### 2. 二维数组

---


二维数组可以看成是线性表的推广。



例如，设A是一个有m行n列的二维数组，则A可以表示为：

$$A = (\alpha_1 \quad \alpha_2 \quad \cdots \quad \alpha_j \quad \cdots \quad \alpha_n)$$
$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn} \end{bmatrix}$$

矩阵 $A_{m \times n}$ 看成n个列向量的线性表



$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn} \end{bmatrix} \begin{matrix} \leftarrow \\ \leftarrow \\ \leftarrow \\ \leftarrow \\ \leftarrow \\ \leftarrow \end{matrix} \begin{matrix} \overline{\beta_1} \\ \beta_2 \\ \vdots \\ \beta_i \\ \vdots \\ \beta_m \end{matrix}$$

$$\begin{matrix} B \\ || \\ \end{matrix}$$

矩阵 $A_{m \times n}$ 看成 $m$ 个行向量的线性表



由此可知二维数组中的每一个元素最多可有二个直接前驱和两个直接后继（边界除外），故是一种典型的非线性结构。

### 3. 多维数组


同理，三维数组最多可有三个直接前驱和三个直接后继，三维以上数组可以作类似分析。因此，可以把三维以上的数组称为多维数组，多维数组可有多个直接前驱和多个直接后继，故多维数组是一种非线性结构。



## 5.2 多维数组的存储结构

由于数组一般不作插入或删除操作，也就是说，一旦建立了数组，则结构中的数组元素个数和元素之间的关系就不再发生变动，即它们的逻辑结构就固定下来了，不再发生变化。因此，采用顺序存储结构表示数组是顺理成章的事了。**本章中，仅重点讨论二维数组的存储，三维及三维以上的数组可以作类似分析。**

数组的顺序存储结构有两种：一种是按行序存储，另一种是按列序存储。



# 多维数组的顺序存储有两种形式:

## 5.2.1 行优先顺序

### 1. 存放规则

---

行优先顺序也称为低下标优先或左边下标优先于右边下标。具体实现时，按行号从小到大的顺序，先将第一行中元素全部存放好，再存放第二行元素，第三行元素，依次类推 ..... 。二维数组 $A_{m \times n}$ 以行为主的存储序列为：

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

在BASIC语言、 PASCAL语言、 C/C++语言等高级语言程序设计中，都是按行优先顺序存放的。







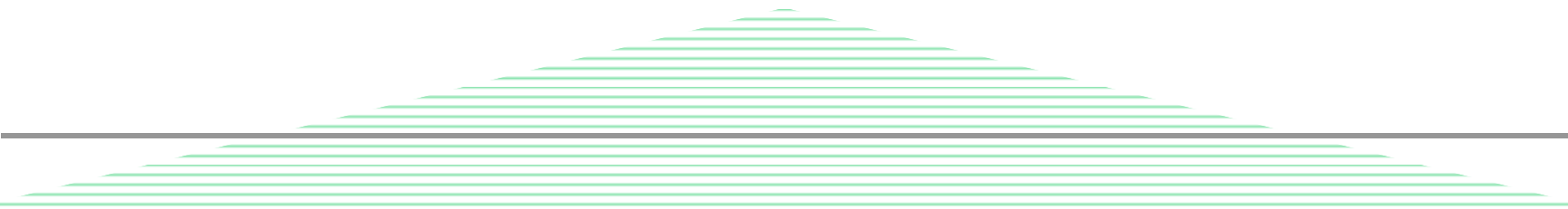
## 2. 地址计算

---

由于多维数组在内存中排列成一个线性序列，因此，若知道第一个元素的内存地址，如何求得其它元素的内存地址？我们可以将它们的地址排列看成是一个等差数列，假设每个元素占1个字节，元素 $a_{ij}$ 的存储地址应为第一个元素的地址加上排在 $a_{ij}$ 前面的元素所占用的单元数，而 $a_{ij}$ 的前面有 $i$ 行( $0 \sim i-1$ )共 $i \times n$ 个元素，而本行前面又有 $j$ 个元素，故 $a_{ij}$ 的前面一共有 $i \times n + j$ 个元素，设 $a_{00}$ 的内存地址为 $LOC(a_{00})$ ，则 $a_{ij}$ 的内存地址按等差数列计算为

$$LOC(a_{ij}) = LOC(a_{00}) + (i \times n + j) \times 1。$$

同理，三维数组 $A_{m \times n \times p}$ 按行优先存放的地址计算公式为：

$$LOC(a_{ijk}) = LOC(a_{000}) + (i \times n \times p + j \times p + k) \times 1。$$




## 5.2.2 列优先顺序


### 1. 存放规则

列优先顺序也称为高下标优先或右边下标优先于左边下标。具体实现时，按列号从小到大的顺序，先将第一列中元素全部存放好，再存放第二列元素，第三列元素，依次类推……。二维数组 $A_{m \times n}$ 以列为主的存储序列为：

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

高级语言中的FORTRAN语言就是以列序为主。





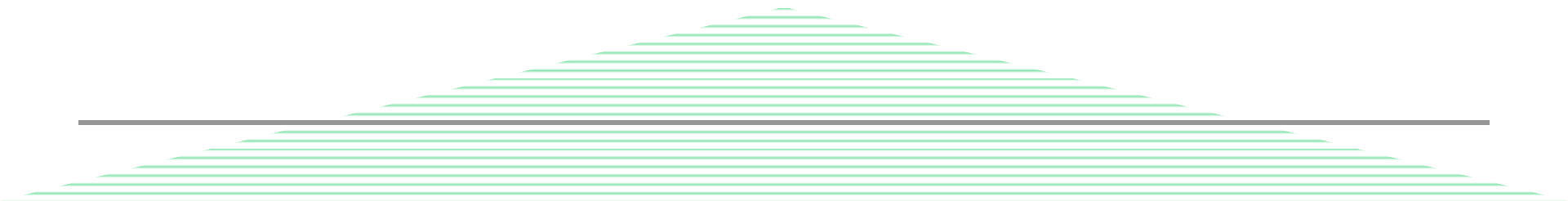
## 2. 地址计算

---

同样与行优先存放类似，若知道第一个元素的内存地址，则同样可以求得按列优先存放的某一元素 $a_{ij}$ 的地址。对二维数组有：

$$\text{LOC}(a_{ij}) = \text{LOC}(a_{00}) + (j \times m + i) \times 1$$

对三维数组有：

$$\text{LOC}(a_{ijk}) = \text{LOC}(a_{000}) + (k \times m \times n + j \times m + i) \times 1$$


## 5.3 特殊矩阵及其压缩存储

### 5.3.1 特殊矩阵

值相同的元素或零元素在矩阵中的分布有一定规律

#### 1. 对称矩阵

若一个 $n$ 阶方阵 $A$ 中元素满足下列条件：

$a_{ij}=a_{ji}$  其中  $0 \leq i, j \leq n-1$  , 则称 $A$ 为对称矩阵。

例如，下图是一个 $3 \times 3$ 的对称矩阵。

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 4 \\ 3 & 4 & 6 \end{bmatrix}$$

一个对称矩阵

## 2. 三角矩阵

### (1) 上三角矩阵

即矩阵上三角部分元素是随机的，而下三角部分元素全部相同（为某常数C）或全为0，具体形式见图 (a)。

### (2) 下三角矩阵


即矩阵的下三角部分元素是随机的，而上三角部分元素全部相同（为某常数C）或全为0，具体形式见图 (b)。

$$\begin{bmatrix} a_{00} & a_{01} & \dots & a_{0n-1} \\ c & a_{11} & \dots & a_{1n-1} \\ \dots & \dots & \dots & \dots \\ c & c & c & a_{n-1n-1} \end{bmatrix} \quad \begin{bmatrix} a_{00} & c & \dots & c \\ a_{10} & a_{11} & \dots & c \\ \dots & \dots & \dots & \dots \\ a_{n-10} & a_{n-11} & \dots & a_{n-1n-1} \end{bmatrix}$$

(a) 上三角矩阵

(b) 下三角矩阵

三角矩阵



## 5.3.2 压缩存储

### 1. 对称矩阵

若矩阵 $A_{n \times n}$ 是对称的，对称的两个元素可以共用一个存储单元，这样，原来 $n$ 阶方阵需  $n^2$ 个存储单元，若采用压缩存储，仅需  $n(n+1)/2$ 个存储单元，将近节约一半存储单元，这就是实现压缩的好处。但是，将 $n$ 阶对称方阵存放到一个向量空间 $s[0]$ 到 $s[\frac{n(n+1)}{2} - 1]$ 中，

怎样找到 $s[k]$ 与 $a[i][j]$ 的一一对称应关系呢？使我们在 $s[k]$ 中直接找到 $a[i][j]$ 。

以行优先存放分两种方式讨论：

## (1) 只存放下三角部分

$$\begin{bmatrix}
 a_{00} & & & & \\
 a_{10} & a_{11} & & & \\
 a_{20} & a_{21} & a_{22} & & \\
 \dots & \dots & \dots & \dots & \dots \\
 a_{n-10} & a_{n-11} & a_{n-12} & \dots & a_{n-1n-1}
 \end{bmatrix}
 \quad
 k = \begin{cases} i(i+1)/2 + j & \text{当 } i \geq j \\ j(j+1)/2 + i & \text{当 } i < j \end{cases}$$

(a) 一个下三角矩阵

0	1	2	3	4	5	6	7	.....	$\frac{n(n+1)}{2}$	-3	$\frac{n(n+1)}{2}$	-2	$\frac{n(n+1)}{2}$	-1
$a_{00}$	$a_{10}$	$a_{11}$	$a_{20}$	$a_{21}$	$a_{22}$	$a_{30}$	$a_{31}$	.....	$a_{n-1n-3}$		$a_{n-1n-2}$		$a_{n-1n-1}$	

(b) 下三角矩阵的压缩存储形式

## (2) 只存放上三角部分

$$\begin{bmatrix}
 a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\
 & a_{11} & a_{12} & \cdots & a_{1n-1} \\
 & & a_{22} & \cdots & a_{2n-1} \\
 \cdots & \cdots & \cdots & \cdots & \cdots \\
 & & & & a_{n-1n-1}
 \end{bmatrix}
 \quad
 k = \begin{cases}
 i*n - \frac{i(i-1)}{2} & \text{当 } i \leq j \\
 j*n - \frac{j(j-1)}{2} & \text{当 } i > j
 \end{cases}$$

(a) 一个上三角矩阵

0	1	2	3	4	5	6	7	.....	$\frac{n(n+1)}{2} - 3$	$\frac{n(n+1)}{2} - 2$	$\frac{n(n+1)}{2} - 1$
$a_{00}$	$a_{01}$	$a_{02}$	$a_{03}$	$a_{04}$	$a_{05}$	$a_{06}$	$a_{07}$	.....	$a_{n-2n-2}$	$a_{n-2n-1}$	$a_{n-1n-1}$

(b) 上三角矩阵的压缩存储形式





## 2. 三角矩阵

### (1) 下三角矩阵

下三角矩阵的压缩存放与对称矩阵用下三角形式存放类似，但必须多一个存储单元存放上三角部分元素，使用的存储单元数目为 $n(n+1)/2+1$ 。故可以将 $n \times n$ 的下三角矩阵压缩存放于只有 $n(n+1)/2+1$ 个存储单元的向量中，假设仍按行优先存放，这时 $s[k]$ 与 $a[i][j]$ 的对应关系为：


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



## (2)上三角矩阵

和下三角矩阵的存储类似，共需  $n(n+1)/2+1$  个存储单元，假设仍按行优先顺序存放，这时  $s[k]$  与  $a[i][j]$  的对应关系为：

$$k = \begin{cases} i*n - i(i-1)/2 + j - i & \text{当 } i \leq j \\ n(n+1)/2 & i > j \end{cases}$$



### 3. 对角矩阵

在一个 $n \times n$ 的三对角矩阵中，只有 $n+n-1+n-1$ 个非零元素，故只需 $3n-2$ 个存储单元即可，零元已不占用存储单元。

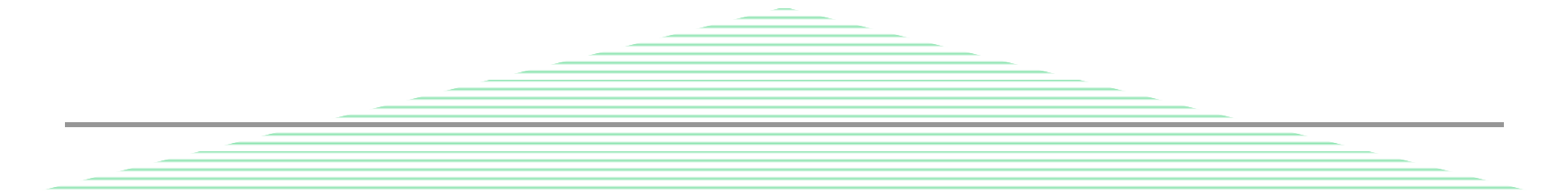
故一般情况下，可将 $n \times n$ 三对角矩阵 $A$ 压缩存放到只有 $3n-2$ 个存储单元的 $s$ 向量中，假设仍按行优先顺序存放，则：

利用三对角带状矩阵有如下特点压缩存储：

$$\begin{cases} i=1, j=1, 2; \\ 1 < i < n, j=i-1, i, i+1; \\ i=n, j=n-1, n; \end{cases}$$

时， $a_{ij}$ 非零，其它元素均为零。

思考：习题册：5.6




## 5.4 稀疏矩阵

在实际应用中，我们还经常会遇到一类矩阵：其矩阵阶数很大，非零元个数较少，零元很多，但非零元的排列没有一定规律，我们称这一类矩阵为稀疏矩阵。

$$\mathbf{M}_{6 \times 7} = \begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{N}_{7 \times 6} = \begin{bmatrix} 0 & 0 & 3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

稀疏矩阵

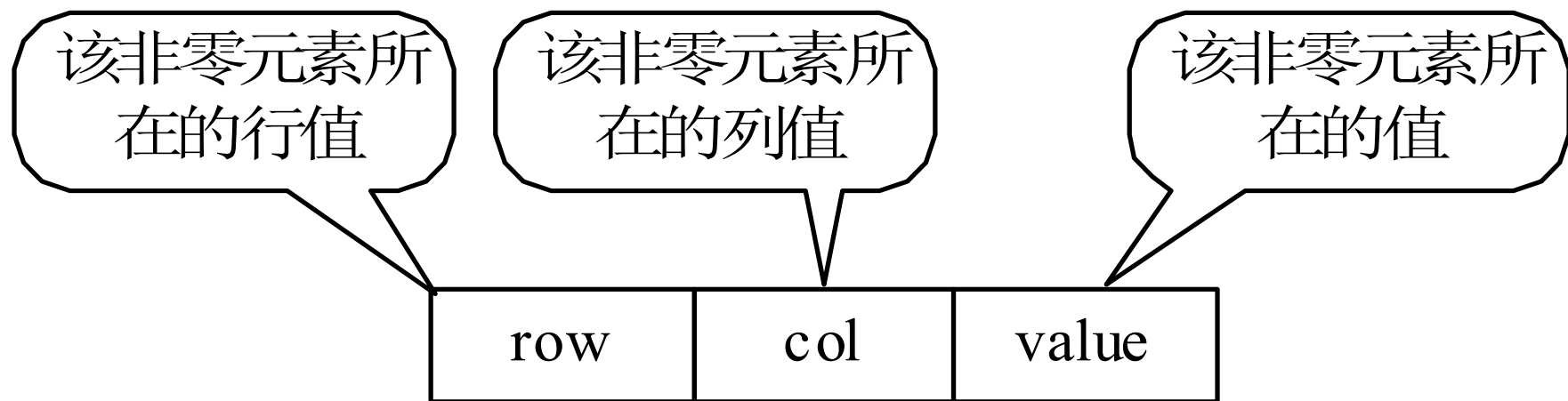


### 5.4.1 稀疏矩阵的存储


按照压缩存储的概念，要存放稀疏矩阵的元素，由于没有某种规律，除存放非零元的值外，还必须存贮适当的辅助信息，才能迅速确定一个非零元是矩阵中的哪一个位置上的元素。下面将介绍稀疏矩阵的几种存储方法及一些算法的实现。

## 1. 三元组表（矩阵的转置）

在压缩存放稀疏矩阵的非零元同时，若还存放此非零元所在的行号和列号，则称为三元组表法，即称稀疏矩阵可用三元组表进行压缩存储，但它是一种顺序存贮（按行优先顺序存放）。一个非零元有行号、列号、值，为一个三元组，整个稀疏矩阵中非零元的三元组合起来称为三元组表。



三元组的结构



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

稀疏矩阵 M

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

稀疏矩阵 N (M 的转置)

稀疏矩阵M和N的三元组表见下图。

M 的三元组表		
i	j	v
0	1	12
0	2	9
2	0	-3
2	5	14
3	2	24
4	1	18
5	0	15
5	3	-7

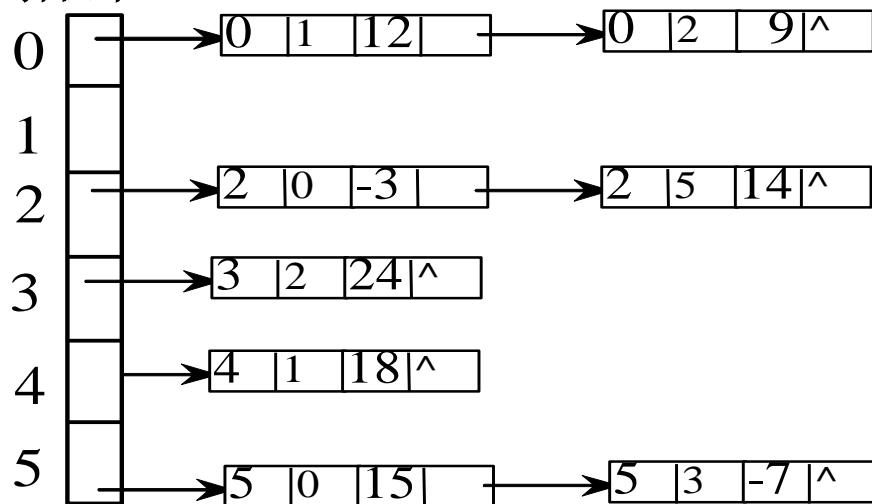
N 的三元组表		
i	j	v
0	2	-3
0	5	15
1	0	12
1	4	18
2	0	9
2	3	24
3	5	-7
5	2	14

稀疏矩阵 M 和 N 的三元组

## 2. 带行指针的链表（矩阵相乘）

把具有相同行号的非零元用一个单链表连接起来，稀疏矩阵中的若干行组成若干个单链表，合起来称为带行指针的链表。例如，稀疏矩阵M的带行指针的链表描述形式见图。

行指针



带行指针的链表





### 3. 十字链表（矩阵相加）

---

当稀疏矩阵中非零元的位置或个数经常变动时，三元组就不适合于作稀疏矩阵的存储结构，此时，采用链表作为存储结构更为恰当。

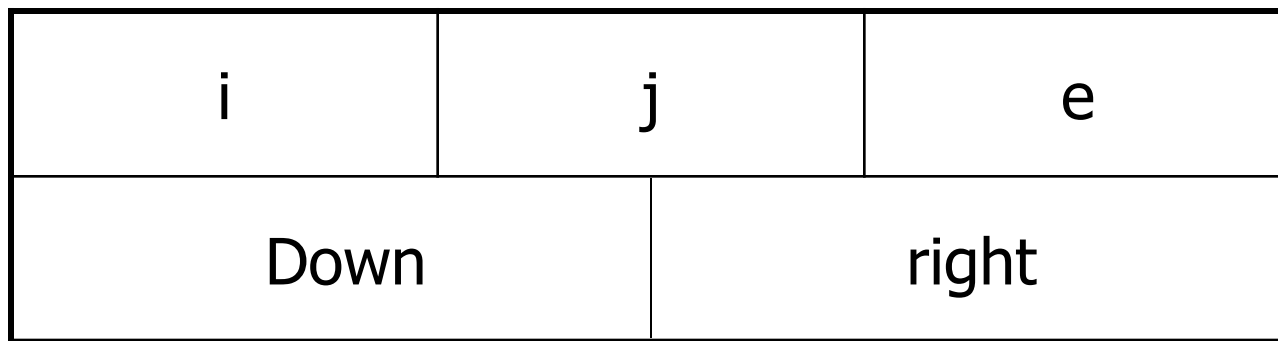
每个非零元既是第 $i$ 行循环链表中的一个结点，又是第 $j$ 列循环链表中的一个结点，相当于处在一个十字交叉路口，故称链表为十字链表。



在十字链表中，矩阵的每一个非零元素用一个结点表示，该结点除了  $(i, j, e)$  以外，还要有以下两个链域：

**right:** 用于链接同一行中的下一个非零元素；

**down:** 用于链接同一列中的下一个非零元素。

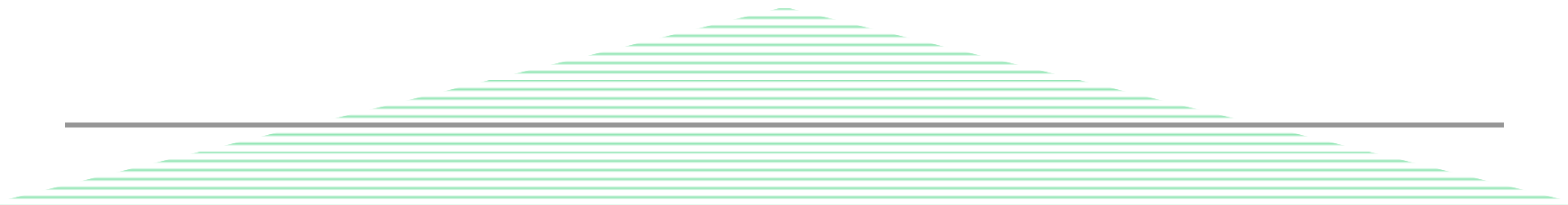


十字链表中结点的结构示意图



十字链表的结构类型说明如下：

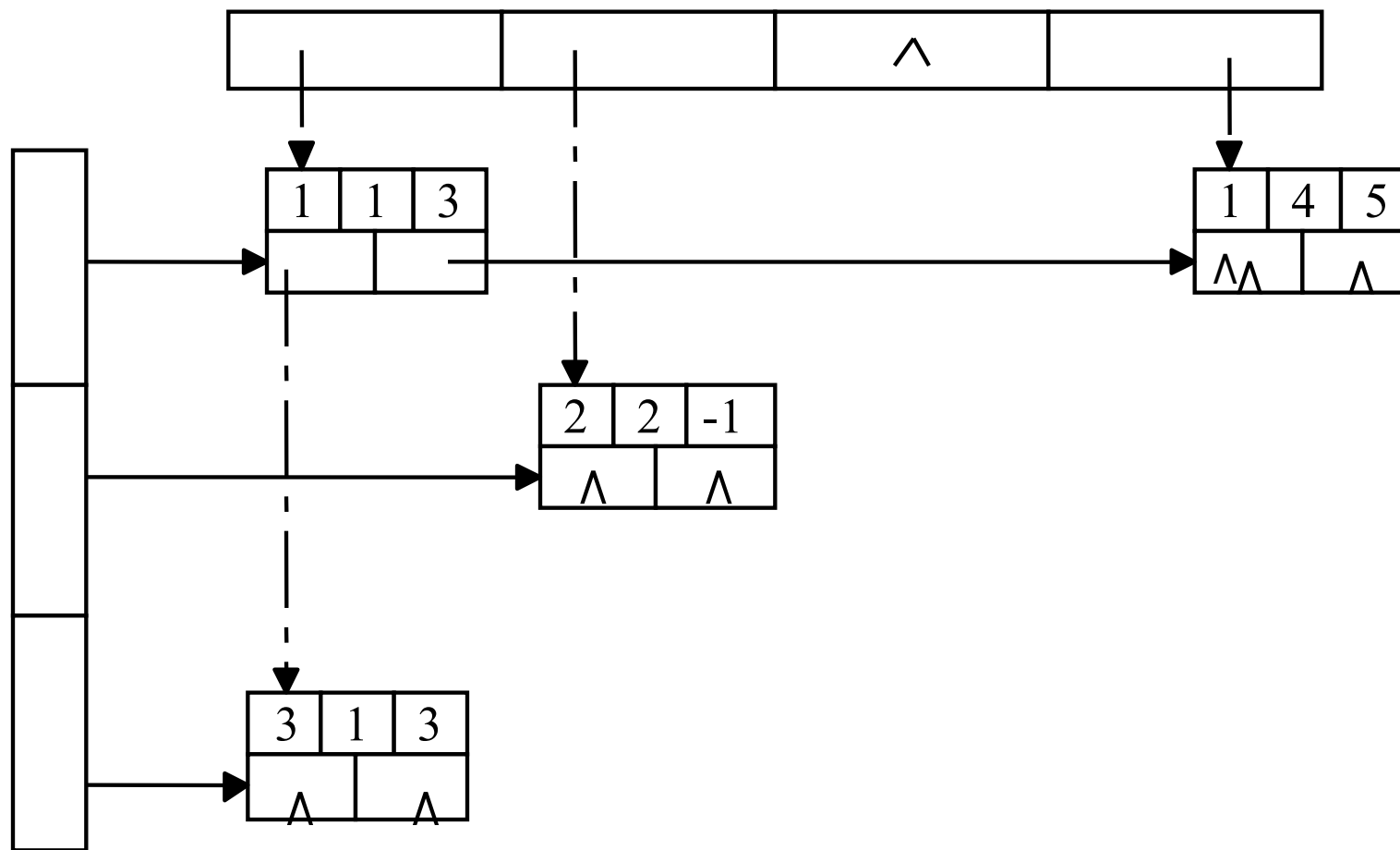
```
typedef struct OLNode {  
    int    i, j;        // 非零元素的行和列下标  
    ElementType    e;  
    struct OLNode  * right, *down; // 非零元素所在行表、列表的后继链域  
}OLNode; *OLink;  
  
typedef struct{  
    OLink * rhead, *chead; // 行、列链表的头指针向量  
    int    mu, nu, tu; /稀疏矩阵的行数、列数、非零元素的个数  
}CrossList;
```





thead

rhead



十字链表的结构



```
Status CreateSMatrix_OL ( CrossList & M) {
```

```
    // 采用十字链表存储结构， 创建稀疏矩阵M
```

```
    if(M) free(M);
```

```
    scanf(&m, &n, &t); // 输入M的行数, 列数和非零元素的个数
```

```
    M.mu=m; M.nu=n; M.tu=t;
```

```
    if(!(M.rhead=(OLink *)malloc((m+1)sizeof(OLink)))) exit(OVERFLOW);
```

```
    if(!(M.thead=(OLink *)malloc((n+1)sizeof(OLink)))) exit(OVERFLOW);
```

```
    M.rhead [ ] =M.rhead [ ] =NULL;
```

```
    // 初始化行、 列头指针向量， 各行、 列链表为空的链表
```

```
    for(scanf(&i, &j, &e); i!=0; scanf(&i, &j, &e)){
```


```
        if(!(p=(OLNode *) malloc(sizeof(OLNode)))) exit(OVERFLOW);
```

```
        p->i=i; p->j=j; p->e=e; // 生成结点
```

```
        if(M.rhead [i] ==NULL&& M.rhead [i] ->j >p->j)
```

```
            {p->right = M.rhead [i] ; M.rhead [i] =p;}
```

---



```
else{ /*寻找行表中的插入位置
```

```
    for(q=M.rhead [i] ; q->right&&q->right->j<j; q=q->right);
```

```
        p->right=q->right; q->right=p; // 完成插入
```

```
}
```

```
if(M.chead [j] ==NULL && M.rhead [j] ->i >p->i)
```

```
    { p->down = M.chead [j] ; M.chead [j] =p; }
```

```
else{ //寻找列表中的插入位置
```

```
    for(q=M.chead [j] ; q->down&&q->down->i<i; q=q->down);
```

```
        p->down=q->down; q->down=p; // 完成插入
```

```
}
```

```
return OK;
```

```
} //CreateSMatrix_OL
```


---



# 作业:

5.1、 5.5

思考: 5.6



## 5.5 广义表

### 5.5.1 基本概念

广义表的线性表的推广。线性表中的元素仅限于原子项，即不可以再分，而广义表中的元素既可以是原子项，也可以是子表(另一个线性表)。





## 1. 广义表的定义

---

广义表是 $n \geq 0$ 个元素 $a_1, a_2, \dots, a_n$ 的有限序列，其中每一个 $a_i$ 或者是原子，或者是一个子表。

广义表通常记为 $LS=(a_1, a_2, \dots, a_n)$ ，其中 $LS$ 为广义表的名字， $n$ 为广义表的长度，每一个 $a_i$ 为广义表的元素。

$a_i \in \text{Atomset}$       单个元素      原子      小写字母

$a_i \in \text{GList}$       广义表      子表      大写字母

表头      第一个元素 $a_1$

表尾      其余剩下的所有的 $(a_2, \dots, a_n)$


深度      广义表展开后所含括号的层数

---



## 2. 广义表举例

广义表	长度	深度	表头	表尾
$A = (e)$	1	1	$e$	$()$
$B = (a, (b, c), d)$	3	2	$a$	$((b, c), d)$
$C = ()$	0	1		$\wedge$
$D = (A, B, C)$	3	3	$A$	$(B, C)$
$E = (a, E)$	2		$a$	$(E)$
$F = (( ))$	1	2	$()$	$()$



## 5.5.2存储结构

由于广义表的元素类型不一定相同，因此，难以用顺序结构存储表中元素，通常采用链接存储方法来存储广义表中元素，并称之为广义链表。常见的表示方法有：


### 1. 头尾链表存储表示

原子 原子结点

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

列表 表结点

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



## 2. 扩展线性链表存储表示

---

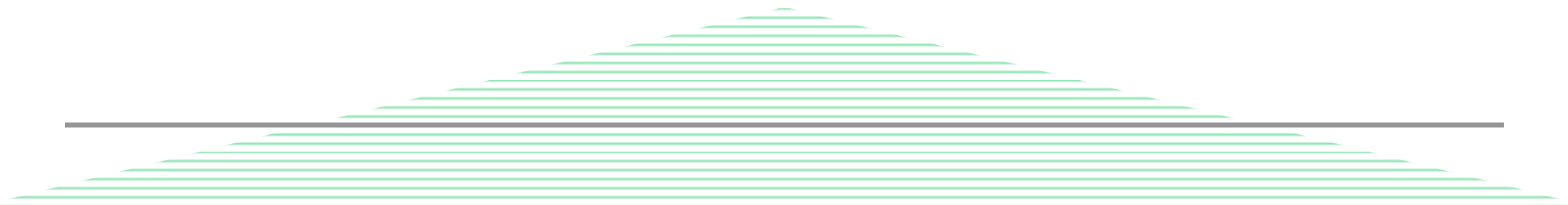
原子 原子结点

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

列表 表结点

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

---





作业:

5.10(自做), 5.12(1), 5.13(1)