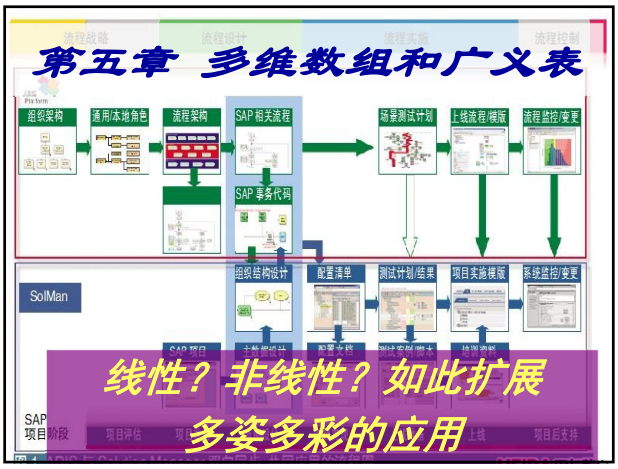


第五章 多维数组和广义表

- 数组
- 特殊矩阵的压缩存储
- 稀疏矩阵
- 广义表

数组

- 数组是一种很特殊的数据结构：
- 数组是存储结构，是语言内建的数据类型。它可以成为多种数据结构的存储表示。它的操作只有按下标“读 / 写”。
- 数组又是逻辑结构，用于问题的解决。可以有查找、定位、插入、删除等操作。
- 一维数组的数组元素为不可再分割的单元素时，是线性结构；但它的数组元素是数组时，是多维数组，是非线性结构。

一维数组

- 定义 数组是相同类型的数据元素的集合，而一维数组的每个数组元素是一个序对，由下标（index）和值（value）组成。
- 一维数组的示例

0	1	2	3	4	5	6	7	8	9
35	27	49	18	60	54	77	83	41	02

- 在高级语言中的一维数组只能按元素的下标直接存取数组元素的值。

一维数组的连续存储表示

- 设每个数组元素占据相等的 l 个存储单元，第 0 号元素的存储地址为 a，则第 i 号数组元素的存储地址 LOC(i) 为：

$$LOC(A[i]) = \begin{cases} a, & i = 0\text{时} \\ LOC(A[i-1]) + l = a + i*l, & i > 0\text{时} \end{cases}$$

	0	1	2	3	4	5	6	7	8	9
a	35	27	49	18	60	54	77	83	41	02
	l	l	l	l	l	l	l	l	l	l

$a+i*l$

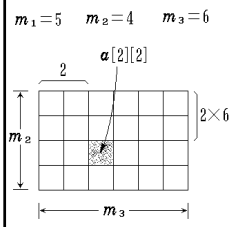
一维数组的定义和初始化

```
void main ( ) {
   int a[3] = { 3, 5, 7 }, *elem, i;              //静态数组
   for ( i = 0; i < 3; i++ )  printf ( "%d", a[i] );
   printf ( "\n" );
   elem = (int *) malloc (3*sizeof (int));  //动态数组
   for ( i = 0; i < 3; i++ ) scanf ("%d", &elem[i] );
   for ( i = 0; i < 3; i++ )
      { printf ( "%d", *elem );  elem++; }
   printf ( "\n" );
}
```

多维数组

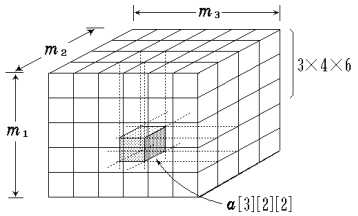
- 多维数组属于数组套数组，可以看做是一维数组的推广。多维数组的特点是每一个数据元素可以有多个直接前驱和多个直接后继。
- 例如，二维数组可以视为其每一个数组元素为一维数组的一维数组，但从整体来看，每一个数组元素同时处于两个向量（行、列），它可能有两个直接前驱，有两个直接后继。必须有两个下标（行、列）以标识该元素的位置。
- 数组元素的下标一般具有固定的下界和上界，因此它比其他复杂的非线性结构简单。

二维数组



行向量 下标 i
列向量 下标 j

三维数组



页向量 下标 i
行向量 下标 j
列向量 下标 k

二维数组的连续存储表示

- 一维数组常被称为向量（Vector）。
- 二维数组 A[m][n] 可看成是由 m 个行向量组成的向量，也可看成是由 n 个列向量组成的向量。
- 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：

```
typedef T array2[m][n];      //T为元素类型
```

等价于：

```
typedef T array1[n];         //行向量类型
typedef array1 array2[m];    //二维数组类型
```


- 同理，一个三维数组类型可以定义为其数据元素为二维数组类型的一维数组类型。
- 静态定义的数组，其维数和维界在数组定义时指定，在编译时静态分配存储空间。一旦数组空间用完则不能扩充。
- 动态定义的数组，其维界不在说明语句中显式定义，而是在程序运行中创建数组时通过动态分配语句 malloc 分配存储空间和初始化，在撤销数组时通过 free 语句动态释放。
- 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列到一个序列中。

二维数组的动态定义和初始化

- 在程序中静态定义二维数组

```
#define m 30
int A[m][m];
```

- 在程序中用动态存储分配建立的二维数组。

```
int **A;  int m = 10, n = 6, i, j;
*A = (int **) malloc (m*sizeof (int *));
for ( i = 0; i < m; i++ )
   A[i] = (int *) malloc (n*(int));
for ( i = 0; i < m; i++ )
   for ( j = 0; j < n; j++ )  scanf ("%d", &A[i][j] );
```


- 动态回收也需要分两步：

```
for ( i = 0; i < m; i++) free (A[i]);  free (A);
```

二维数组中数组元素的顺序存储

- 设有一个 n 行 m 列的二维数组，如图：

$$a=\begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][m-1] \\ a[1][0] & a[1][1] & \cdots & a[1][m-1] \\ a[2][0] & a[2][1] & \cdots & a[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[n-1][0] & a[n-1][1] & \cdots & a[n-1][m-1] \end{pmatrix}$$

- 用一维数组描述它的连续存放方式

$$a = \begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][m-1] \\ a[1][0] & a[1][1] & \cdots & a[1][m-1] \\ a[2][0] & a[2][1] & \cdots & a[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[n-1][0] & a[n-1][1] & \cdots & a[n-1][m-1] \end{pmatrix}$$

- **行优先存放（以行为主序）：**
  设数组开始存放位置为 a，每个元素占用 l 个存储单元，则$a[i][j]$的存储位置为：

  $Loc(a[i][j]) = a+(i*m+j)*l$

  其中，m 是每行元素个数，即列数。

$$a = \begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][m-1] \\ a[1][0] & a[1][1] & \cdots & a[1][m-1] \\ a[2][0] & a[2][1] & \cdots & a[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[n-1][0] & a[n-1][1] & \cdots & a[n-1][m-1] \end{pmatrix}$$

- **列优先存放（以列为主序）：**
  设数组开始存放位置为a，每个元素占用 l 个存储单元，则$a[i][j]$的存储位置为：

  $Loc(a[i][j]) = a+(j*n+i)*l$

  其中，n 是每列元素个数，即行数。

三维数组

- 各维元素个数为 m_1, m_2, m_3
- 下标为 i_1, i_2, i_3的数组元素的存储地址：
  （按页 / 行 / 列存放）

$$\&(a[i_1][i_2][i_3]) = a + (\underbrace{i_1 * m_2 * m_3}_{\text{前 } i_1 \text{ 页总元素个数}} + \underbrace{i_2 * m_3}_{\text{第 } i_1 \text{ 页前 } i_2 \text{ 行总元素个数}} + \underbrace{i_3}_{\text{第 } i_2 \text{ 行前 } i_3 \text{ 列元素个数}}) * l$$

一般情况：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：

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

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

特殊矩阵的压缩存储

- 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。
- 特殊矩阵的压缩存储主要是针对阶数很高的特殊矩阵。为节省存储空间，对可以不存储的元素，如零元素或对称元素，不再存储。
- 有三种特殊矩阵：
  - 对称矩阵
  - 三对角矩阵
  - w 对角矩阵

对称矩阵的压缩存储

- 设有一个 n×n 的矩阵 A。如果在在矩阵中，$a_{ij} = a_{ji}$，则此矩阵是对称矩阵。
- 若只保存对称矩阵的对角线和对角线以上 (下) 的元素，则称此为对称矩阵的压缩存储。

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

- 若只存对角线及对角线以上的元素，称为上三角矩阵；若只存对角线或对角线以下的元素，称之为下三角矩阵。

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

下三角矩阵

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

上三角矩阵

- 把它们按行存放于一个一维数组 B 中，称之为对称矩阵 A 的压缩存储方式。
- 数组 B 共有 n+(n-1)+…+1 = n*(n+1)/2 个元素。

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

下三角矩阵

	0	1	2	3	4	5	6	7	8		$n(n+1)/2-1$
B	a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	a_{30}	a_{31}	a_{32}	.....	a_{n-1n-1}

- 若 i≥j，数组元素a[i][j]在数组B中的存放位置为

$$1 + 2 + \cdots + i + j = (i + 1)* i / 2 + j$$

前i 行元素总数　第 i 行第 j 个元素前元素个数

- 若 i<j，数组元素 A[i][j] 在矩阵的上三角部分，在数组 B 中没有存放，可以找它的对称元素

  A[j][i] = j *(j +1) / 2 + i
- 反过来，若已知某矩阵元素位于数组 B 的第 k 个位置，可寻找满足

  i (i + 1) / 2≤ k < (i + 1)*(i + 2) / 2

  的 i，此即为该元素的行号。

  j = k - i * (i + 1) / 2

  此即为该元素的列号。
- 例，当 k = 8，3*4 / 2 = 6 ≤ k < 4*5 / 2 =10，取 i = 3。则 j = 8 – 3*4 / 2 = 2。

$$n = 4 \quad \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

上三角矩阵

	0	1	2	3	4	5	6	7	8	9
B	a_{00}	a_{01}	a_{02}	a_{03}	a_{11}	a_{12}	a_{13}	a_{22}	a_{23}	a_{33}

若 i≤j，数组元素A[i][j]在数组B中的存放位置为

$$n + (n-1) + (n-2) + \cdots + (n-i+1) + j-i$$

前 i 行元素总数　第 i 行第 j 个元素前元素个数

- 若 i≤j，数组元素A[i][j]在数组B中的存放位置为

  n + (n–1) + (n–2) + … + (n–i+1) + j–i =

  = (2*n–i+1) * i / 2 + j–i =

  = (2*n–i–1) * i / 2 + j

  若i>j，数组元素A[i][j]在矩阵的下三角部分，在数组 B 中没有存放。因此，找它的对称元素 A[j][i]。A[j][i]在数组 B 的第 (2*n–j–1) * j / 2 + i 的位置中找到。

三对角矩阵的压缩存储

$$A=\begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & a_{n-2n-3} & a_{n-2n-2} & a_{n-2n-1} \\ 0 & 0 & 0 & 0 & a_{n-1n-2} & a_{n-1n-1} \end{bmatrix}$$

	0	1	2	3	4	5	6	7	8	9	10
B	a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	…	a_{n-1n-2}	a_{n-1n-1}

- 三对角矩阵中除主对角线及在主对角线上 下最临近的两条对角线上的元素外，所有其它元素均为0。总共有$3n-2$个非零元素。
- 将三对角矩阵A中三条对角线上的元素按行存放在一维数组 B 中，且a_{00}存放于B[0]。
- 在三条对角线上的元素a_{ij}满足

  $0 \le i \le n-1,\ i-1 \le j \le i+1$
- 在一维数组 B 中 A[i][j] 在第 i 行，它前面有 $3*i-1$ 个非零元素, 在本行中第 j 列前面有 $j-i+1$ 个，所以元素 A[i][j] 在 B 中位置为 $k = 2*i+j$。

- 若已知三对角矩阵中某元素 A[i][j] 在数组 B[ ] 存放于第 k 个位置，则有

  $i=\lfloor (k+1)/3 \rfloor$

  $j = k - 2*i$
- 例如，当 $k = 8$ 时，

  $i=\lfloor (8+1)/3 \rfloor = 3,\ j = 8 - 2*3 = 2$

  当 k = 10 时，

  $i=\lfloor (10+1)/3 \rfloor = 3,\ j = 10 - 2*3 = 4$

w 对角矩阵的压缩存储

- 一个 w 对角矩阵是指主对角线两侧各有(w-1)/2条次对角线，其他位置都是零元素的矩阵，所以又称为带状矩阵（w为奇数）。

$$A=\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & & \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ & & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ & & & a_{5,3} & a_{5,4} & a_{5,5} \end{bmatrix}$$

- 非零元素 $a_{i,j}$ 的下标应满足

  $0 \le i \le n-1,\ i-(w-1)/2 \le j \le i+(w-1)/2$
- 如果把它的 w 条对角线元素按行优先方式存放到一个一维数组B中，为找到元素$a_{i,j}$ 在 B 中位置，一种简化处理是先把 w 条对角线上的元素压缩在一个 n×w 的二维数组 A' 中， 让$a'_{0,0}$存放在B_0，就可以简单地找到 $a_{i,j}$ 的存储位置了。
- 对于一个 $w = 5$ 的 w 对角矩阵，对应的 A' 矩阵如下图所示。从 $a_{i,j}$ 到$a'_{t,s}$的映射关系为：

  $t = i,\ s = j-i+(w-1)/2$
- 如$i = 0, j = 0$, 则$t = 0, s = 2$； $i = 3, j=4$, 则$t=3, s=3$。

$$A'=\begin{bmatrix} 0 & 0 & a_{0,0} & a_{0,1} & a_{0,2} \\ 0 & a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & 0 \\ a_{5,3} & a_{5,4} & a_{5,5} & 0 & 0 \end{bmatrix}$$

	0	1	2	3	4	5	6	7	8	9
B	0	0	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	0	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$

10	11	12	13	14	15	16	17	18	19
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

- 矩阵元素 $a'_{t,s}$ 在 B 中对应的存放位置 k 为 $t*w+s$，可得 w 对角矩阵元素 $a_{i,j}$在数组 B 中位置为

  $k = i*w+j-i+(w-1)/2$　　$(w = 5, (w-1)/2 = 2)$
- 例如，当 $i = 0, j = 0$ 时，$k = 0*5+0-0+2 = 2$.
- 当 $i = 2, j = 4$ 时，$k = 2*5+4-2+2 = 14$.
- 当 $i = 4, j = 2$ 时，$k = 4*5+2-4+2 = 20$.

稀疏矩阵 (Sparse Matrix)

$$A_{6\times7} = \begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

- 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 << m×n），则称 A 为稀疏矩阵。

- 设矩阵 A 中有 s 个非零元素。令 e = s/(m*n)，称 e 为矩阵的稀疏因子。
- 有人认为 e≤0.05 时称之为稀疏矩阵。
- 在存储稀疏矩阵时，为节省存储空间，应只存储非零元素。但通常非零元素的分布没有规律，故在存储非零元素时，必须记下它所在的行和列的位置 (i, j)。
- 每一个三元组 (i, j, a_{ij}) 唯一确定了矩阵A的一个非零元素。因此，稀疏矩阵可由表示非零元素的一系列三元组及其行列数唯一确定。

稀疏矩阵的顺序存储表示

- 把所有记录稀疏矩阵非零元素的三元组按行为主序的方式存储在一个称为“三元组表”的一维数组（向量）中，即为稀疏矩阵的顺序存储表示。
- 在三元组表中，行为主序，所有非零元素的三元组按行号递增的顺序排列；行号相等的按列号递增的顺序排序。
- 三元组表中三元组的个数记忆在变量Terms 中，此即矩阵中的非零元素个数。稀疏矩阵的行数和列数分别记忆在 Rows 和 Cols中。

稀疏矩阵

$$\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

对应三元组表

	行 (row)	列 (col)	值 (value)
[0]	0	3	22
[1]	0	6	15
[2]	1	1	11
[3]	1	5	17
[4]	2	3	-6
[5]	3	5	39
[6]	4	0	91
[7]	5	2	28

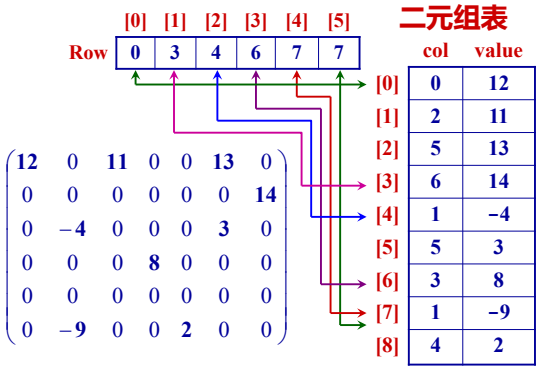
稀疏矩阵的定义

```
# define maxTerms 30      //三元组表默认大小
typedef int DataType;     //矩阵元素数据类型
typedef struct {          //三元组定义
   int row, col;          //非零元素行号/列号
   DataType value;        //非零元素的值
} Triple;

typedef struct {          //稀疏矩阵结构定义
  int Rows, Cols, Terms; //矩阵行、列、非零元素
  Triple elem[maxTerms];  //三元组表
} SparseMatrix;
```

带行指针数组的二元组表

- 稀疏矩阵的三元组表可以用带行指针数组的二元组表代替。
- 在行指针数组中元素个数与矩阵行数相等。第 *i* 个元素的下标 *i* 代表矩阵的第 *i* 行，元素的内容即为稀疏矩阵第 *i* 行的第一个非零元素在二元组表中的存放位置。
- 二元组表中每个二元组只记录非零元素的列号和元素值，且各二元组按行号递增的顺序排列。
- 与三元组表相比，省去了重复的行号。



稀疏矩阵的链接表示

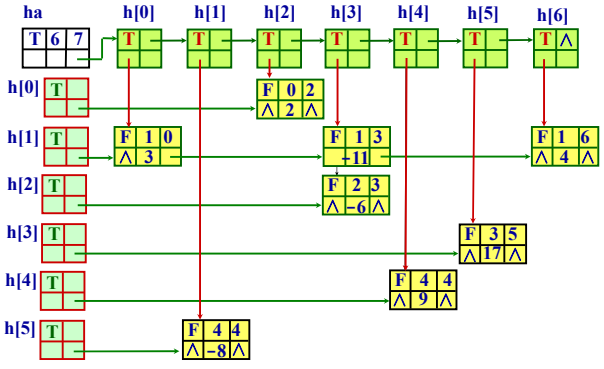
- 在执行稀疏矩阵 （+、-、*、/）操作时，稀疏矩阵的非零元素会发生动态变化，这时，使用三元组表有双重缺陷：
  (1) 不能直接访问矩阵元素；
  (2) 插入或删除时可能发生大量元素移动；
- 用稀疏矩阵的链接表示可以避免这些情况。
- 稀疏矩阵的链接表示采用十字链表：行链表与列链表十字交叉。行链表与列链表都是带头结点的单链表。用头结点表征是第几行，第几列。

- 稀疏矩阵的头结点
  - *head* = true 是头结点的标识。
  - 第 *i* 行与第 *i* 列共用一个头结点，用 *next* 链接。链表中按行列号顺序链接各行列的头结点。
  - *right* 指向该行链表首元结点的指针；*down* 指向该列链表首元结点的指针。
  - 对链表扫描从头结点开始，最后以NULL结束。

head	*next*
down	*right*

- 稀疏矩阵的元素结点的结构
  - *head* = false是稀疏矩阵元素结点的标识。
  - *row* 和 *col* 是非零元素的行 / 列号，*value* 是该非零元素的值。
  - *right* 是在行链表中指向该行下一个非零元素结点的指针；*down*是在列链表中指向该列下一个非零元素结点的指针。
  - 每一元素结点同时处于某行某列链表中。

head	*row*	*col*
down	*value*	*right*



稀疏矩阵的转置

- 一个 $m \times n$ 的矩阵 A，它的转置矩阵 B 是一个 $n \times m$ 的矩阵，且 $A[i][j] = B[j][i]$。即
  - 矩阵 A 的行成为矩阵 B 的列
  - 矩阵 A 的列成为矩阵 B 的行。
- 在稀疏矩阵的三元组表中，非零矩阵元素按行存放。当行号相同时，按列号递增的顺序存放。
- 如果稀疏矩阵的转置运算基于三元组表，则矩阵的转置要直接对相应三元组表进行转置。

稀疏矩阵

$$\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

对应三元组表

	行	列	值
[0]	0	3	22
[1]	0	6	15
[2]	1	1	11
[3]	1	5	17
[4]	2	3	-6
[5]	3	5	39
[6]	4	0	91
[7]	5	2	28

转置矩阵

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 28 \\ 22 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 17 & 0 & 39 & 0 & 0 \\ 15 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

对应三元组表

	行	列	值
[0]	0	4	91
[1]	1	1	11
[2]	2	5	28
[3]	3	0	22
[4]	3	2	-6
[5]	5	1	17
[6]	5	3	39
[7]	6	0	16

原矩阵三元组表

	行	列	值
[0]	0	3	22
[1]	0	6	15
[2]	1	1	11
[3]	1	5	17
[4]	2	3	-6
[5]	3	5	39
[6]	4	0	91
[7]	5	2	28

转置矩阵三元组表

	行	列	值
[0]	0	4	91
[1]	1	1	11
[2]	2	5	28
[3]	3	0	22
[4]	3	2	-6
[5]	5	1	17
[6]	5	3	39
[7]	6	0	16

稀疏矩阵转置算法思想

- 设矩阵列数为 Cols，对矩阵三元组表扫描Cols次。第 k 次检测列号为 k 的项。
- 第 k 次扫描找寻所有列号为 k 的项，将其行号变列号、列号变行号，连同该元素的值，顺次存于转置矩阵三元组表。
- 若设矩阵非零元素有 Terms 个，则上述二重循环执行的时间复杂性为O(Cols×Terms)。
- 若矩阵有 200 行，200 列，10,000 个非零元素，总共有 2,000,000 次处理。

稀疏矩阵的转置算法

```
#include "SparseMatrix.cpp"
void Transpose ( SparseMatrix& a, SparseMatrix& b ) {
//稀疏矩阵 a 转置，在 b 中得到结果
  int CurrentB, k, i;
  b.Rows = a.Cols;  b.Cols = a.Rows;
  b.Terms = a.Terms;
  if ( a.Terms > 0 ) {
    CurrentB = 0;              //转置三元组表存放指针
    for ( k = 0; k < a.Cols; k++ )        //按列号处理
      for ( i = 0; i < a.Terms; i++ )
                  //在三元组表中找列号为 k 的三元组
```

```
        if ( a.elem[i].col == k ) {        //第 i 项列号为 k
            b.elem[CurrentB].row = k;
            b.elem[CurrentB].col = a.elem[i].row;
            b.elem[CurrentB].value = a.elem[i].value;
            CurrentB++;                    //存放指针进1
        }
    }
}
```

- 此算法慢就慢在二重嵌套循环。若能一趟扫描过去就实现转置，运算速度将大大提高。为此，需要事先做点功课。这就是快速转置的想法。

69-49

快速转置算法

- 快速转置的想法是：对 原矩阵 a 扫描一遍，按 a 中每一元素的列号，立即确定在转置矩阵 b 三元组表中的位置，并装入它。
- 为加速转置速度，建立两个辅助数组 rowSize 和 rowStart：
  - rowSize 记录矩阵转置前各列非零元素个数，转置后就是各行非零元素个数；
  - rowStart 记录转置后各行非零元素在转置三元组表中开始存放位置。

69-50

转置矩阵三元组表

	行	列	值
[0]	0	4	91
[1]	1	1	11
[2]	2	5	28
[3]	3	0	22
[4]	3	2	-6
[5]	5	1	17
[6]	5	3	39
[7]	6	0	16

rowSize

0	1	2	3	4	5	6
1	1	1	2	0	2	1

rowStart

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

- 扫描一遍三元组表，根据某项列号j，查 rowStart 表，按 rowStart[j] 所给位置直接将该项存入转置矩阵的三元组表中。

69-51

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	语 义
rowSize	1	1	1	2	0	2	1	矩阵 A 各列非零元素个数
rowStart	0	1	2	3	5	5	7	矩阵 B 各行开始存放位置

A三元组	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)
行row	0	0	1	1	2	3	4	5
列col	3	6	1	5	3	5	0	2
值value	22	15	11	17	-6	39	91	28

69-52

稀疏矩阵的快速转置算法

```
#include "SparseMatrix.cpp"
void FastTranspos ( SparseMatrix& a, SparseMatrix& b )
{
//对稀疏矩阵 a 做快速转置, 结果放在 b 中
   int *rowSize = (int *) malloc (a.Cols*sizeof (int));
   int *rowStart = (int *) malloc (a.Cols*sizeof (int));
   int i, j;
   b.Rows = a.Cols;  b.Cols = a.Rows;
   b.Terms = a.Terms;
   if ( a.Terms > 0 ) {
```

69-53

```
for ( i = 0; i < a.Cols; i++ ) rowSize[i] = 0;
                        //统计矩阵中各列非零元素数
for ( i = 0; i < a.Terms; i++ )
   rowSize[a.elem[i].col]++;
rowStart[0] = 0;     //计算转置后各行开始位置
for ( i = 1; i < a.Cols; i++ )
   rowStart[i] = rowStart[i-1]+ rowSize[i-1];
for ( i = 0; i < a.Terms; i++ ) {     //从 a 向 b 传送
   j = rowStart[a.elem[i].col];       //转置存放位置
   b.elem[j].row = a.elem[i].col;
   b.elem[j].col = a.elem[i].row;
   b.elem[j].value = a.elem[i].value;
```

69-54

```
            rowStart[a.elem[i].col]++;
                    //修改第 j 行元素下一存放位置
        }
    }
    free ( rowSize ); free ( rowStart );
}
```

- 该算法有 4 个并列单重循环，各自的时间复杂度为$O(Cols)$, $O(Terms)$, $O(Cols)$, $O(Terms)$。总的时间复杂度为$O(\max(Cols, Terms))$。
- 若矩阵有 200 行，200 列，10,000 个非零元素,总共有 10,000 次处理。

广义表 (General Lists)

- 广义表是 n (≥0) 个表元素组成的有限序列，记作

  $LS\ (a_1, a_2, a_3, \ldots, a_n)$

- LS 是表名，a_i 是表元素，可以是表（称为子表），可以是单元素（称为原子，不可再分）。
- n为表的长度。$n = 0$ 的广义表为空表。
- $n > 0$时，表的第一个表元素称为广义表 的表头（head），除此之外，其它表元素组成的表称为广义表的表尾（tail）。

广义表的特性

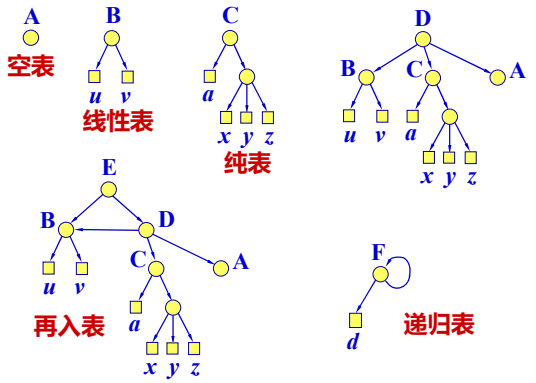
- 有次序性
- 有深度
- 可递归
- 有长度
- 可共享

A( )	A长度为0，深度为1
B(6, 2)	B长度为2，深度为1
C(‘a’, (5, 3, ‘x’))	C长度为2，深度为2
D(B, C, A)	D长度为3，深度为3
E(B, D)	E长度为? 深度为?
F(4, F)	F长度为? 深度为?

广义表的表头与表尾

- 广义表的第一个表元素即为该表的表头，除表头元素外其他表元素组成的表即为该表的表尾。

A( )	head(A) 和 tail(A) 不存在
B(6, 2)	head(B) = 6, tail(B) = (2)
C(‘a’, (5, 3, ‘x’))	head(C) =‘a’
tail(C) = ((5,3,’x’))	
head(tail(C)) = head(((5,3,’x’))) = (5,3,’x’)	
tail(tail(C)) = tail(((5,3,’x’))) = ( )	

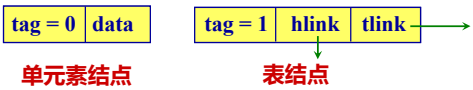


广义表的存储表示

广义表的头尾表示

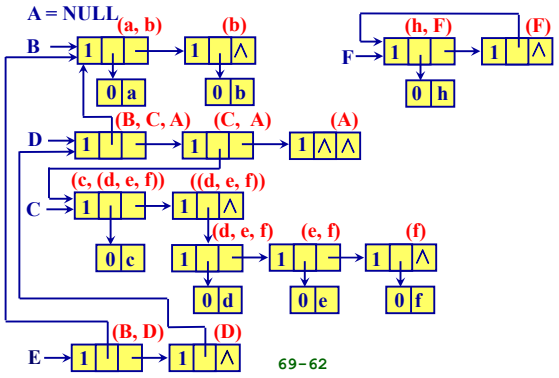
- 除空表外，广义表可以分为表头、表尾两部分。表头可以是单元素，也可以是子表，但表尾一定是子表。因此，广义表的头尾表示有两种结点：
- 表结点。包括指向表头结点的指针hlink和指向表尾结点的指针tlink；
- 单元素结点。保存单元素数据值的data域。
- 每个结点有一个标志域tag。tag = 0，表示该结点是单元素结点；tag = 1，表示该结点是表结点。

广义表的头尾表示的结点类型



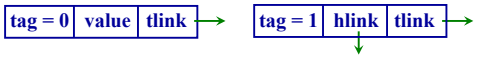
- 空表没有结点，指向空表的头指针为空。
- 非空表的头指针指向一个表结点。该结点的hlink指针指向表头元素结点，tlink指向表尾（该表尾肯定还是表）。
- 如果有多个指针指向一个表结点，则出现共享情况；如果表中某元素是子表，其 hlink 又指向该表，则出现递归情况。

广义表的头尾表示示例



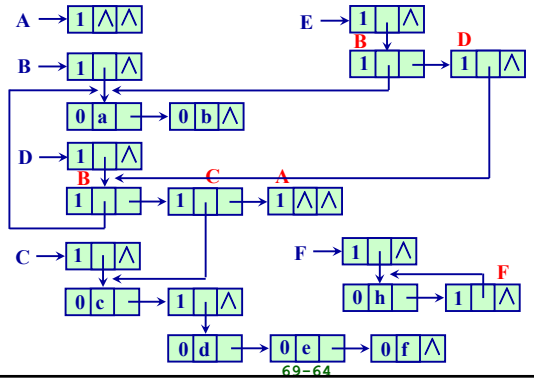
广义表的扩展线性链表表示

- 广义表的扩展线性链表表示也有两种结点：原子结点和表结点。其结点类型如下：



- 单元素结点的标志 tag = 0，value 存放元素的值，tlink 存放指向同一层下一表元素结点的指针；
- 表结点的标志 tag = 1，hlink 存放指向子表的指针 hlink，tlink 与单元素结点tlink的含义相同

广义表的扩展线性链表表示示例



- 与头尾表示相比，优点是每个广义表都有一个起到“头结点”作用的表结点，即使空表，也有一个表结点。
- 这种表示的缺点是每个对子表引用的指针没有指向子表的“头结点”，而是直接指向了广义表的表头元素结点（是第一个表元素结点），这样，造成对表头元素结点插入或删除时的困难。
- 如果某表头元素为多个表结点共享，删除它后如何找到所有共享它的结点，以修改指向这个结点的所有指针，这种表示显然无法胜任。

广义表的层次表示

- 在这种表示中有 3 种结点：原子结点、子表结点和头结点（非表头元素结点）。
- 头结点的作用是要简化插入和删除操作。如果插入或删除表头元素结点，有了头结点，就像带头结点的单链表，不必修改其他表中指向该表头的指针。
- 在表的头结点中存储该表的引用计数。如果要删除该链表，需要先查看引用计数，如果有多个链共享该链表，就不能删除它，只需将其引用计数减一。

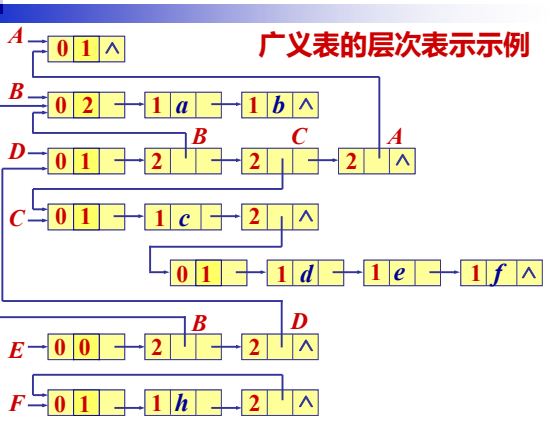
广义表的层次表示中结点的类型定义

tag	info	tlink

- 结点类型 tag：
  = 0，表头结点；= 1，原子结点；= 2，子表结点
- 信息info：tag = 0 时，存放引用计数(ref)；tag = 1 时，存放数据值(value)；tag = 2 时，存放指向子表头结点的指针(hlink)
- 尾指针tlink：tag = 0 时，指向该表第一个结点；tag ≠ 0 时，指向同一层下一个结点

69-67

广义表的层次表示示例



69-68

广义表的结点类型定义

```
typedef struct node {       //广义表结点定义
    int tag;
     // = 0为头结点，=1为原子结点，=2是子表结点
    struct node *tlink;     //指向同层下一结点的指针
    union {                 //共用体，此3个域叠压在同一空间
        int ref;            //tag=0，存放引用计数，设为整数
        char value;         //tag=1，存放数据，设为单字符
        struct node *hlink;
                            //tag=2，存放指向子表的指针
    } info;
} GenListNode, *GenList;
```

三者共享同一空间

69-69