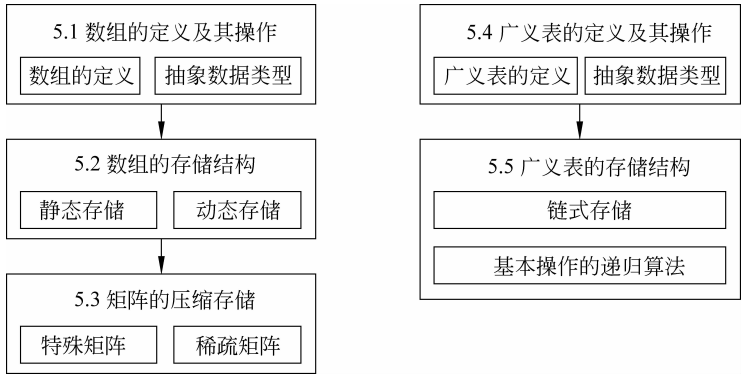
第 5 章　数组和广义表

在线性表 $L=(a_0,a_1,\cdots,a_{n-1})$中，数据元素 a_i 是无结构的(称为原子或单元素)，即 a_i 本身不再是一个数据结构。本章的数组和广义表是线性结构的推广。在这种结构中，元素本身可以又是一个数据结构。本章讨论多维数组的表示、矩阵压缩存储、广义表的表示和相关算法等问题。本章各小节之间关系如下图：



5.1　数组的定义及其操作

5.1.1　数组的定义

数组(Array)是 n(n≥1)维相同类型数据元素构成的有限序列，而且是存储在一块地址连续的空间中。

在众多的算法语言中，如 Fortran、Basic、Pascal 和 C 语言，都有数组类型。前面第 2 至第 4 章中线性结构的顺序存储表示，接触到的都是一维数组。对于一个一维数组，我们已经知道是可以进行随机存取的结构，后面谈到数组地址计算时我们将知道，对于二维乃至多维数组也是如此。本节将以 C 语言为例讨论数组的描述、存储映像、地址计算等问题。

对于一维数组已经很熟悉了，下面来看看二维数组。设二维数组：

$$A = A[d_1][d_2] = \begin{bmatrix} a_{00} & \cdots & a_{0j} & \cdots & a_{0,d_2-1} \\ \vdots & & \vdots & & \vdots \\ a_{i0} & \cdots & a_{ij} & \cdots & a_{i,d_2-1} \\ \vdots & & \vdots & & \vdots \\ a_{d_1-1,0} & \cdots & a_{d_1-1,j} & \cdots & a_{d_1-1,d_2-1} \end{bmatrix}$$

其中：d_1 为数组的行数，d_2 为数组的列数；a_{ij} 为数组中第 i 行、第 j 列的数据元素，$0 \leqslant i \leqslant d_1-1$，$0 \leqslant j \leqslant d_2-1$；元素个数为 $d_1 * d_2$。

二维数组的逻辑结构可以表示为：

$$A^{(2)} = (D, R)$$

其中 D 和 R 分别为数组中数据元素集和关系集：

$D = \{a_{ij} \mid a_{ij} \in \text{datatype}, 0 \leqslant i \leqslant d_1-1, 0 \leqslant j \leqslant d_2-1\}$

$R = \{\text{Row}, \text{Col}\}$

行关系 $\text{Row} = \{< a_{ij}, a_{ij+1} > \mid a_{ij}, a_{ij+1} \in D, 0 \leqslant i \leqslant d_1-1, 0 \leqslant j \leqslant d_2-2\}$

列关系 $\text{Col} = \{< a_{ij}, a_{i+1j} > \mid a_{ij}, a_{i+1j} \in D, 0 \leqslant i \leqslant d_1-2, 0 \leqslant j \leqslant d_2-1\}$

关系集 Row 和 Col 表明：除数组 $A^{(2)}$ 周边元素外的其他任一个元素 a_{ij}，有两个直接前驱 a_{i-1j}，a_{ij-1}，和两个直接后继 a_{i+1j}，a_{ij+1}（周边元素的前驱或后继可不足两个）。n 维数组也可按上述方法类似定义。

还可以用如下形式描述二维数组（设 $d_1=m$，$d_2=n$）：

$$\text{二维数组 } A^{(2)} = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & \cdots & a_{1n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-10} & a_{m-11} & \cdots & a_{m-1n-1} \end{pmatrix} \begin{matrix} -A_0^{(1)} \\ -A_1^{(1)} \\ \\ -A_{m-1}^{(1)} \end{matrix}$$

$$\begin{matrix} | & | & & | \\ A_0^{(1)} & A_1^{(1)} & \cdots & A_{n-1}^{(1)} \end{matrix}$$

故 $A^{(2)} = (A_0^{(1)}, A_1^{(1)}, \cdots, A_{m-1}^{(1)})$（或 $A^{(2)} = (A_0^{(1)}, A_1^{(1)}, \cdots, A_{n-1}^{(1)})$），它形式上变为一个线性表，只不过是其中每个元素 $A_i^{(1)}(0 \leqslant i < m-1) = (a_{i0} \cdots a_{in-1})$ 又是一个线性表而已。三维或者多维数组可以以此类推。

所以说，多维数组是线性表的推广，而线性表是多维数组的特例。

5.1.2 数组的抽象数据类型

数组的基本操作比较简单，因为大多的程序设计语言中数组一旦生成，其元素的存储空间一般就固定下来，故数组的操作不包括插入和删除这样的操作。于是可以定义如下的数组抽象数据类型：

```
ADT Array{
    数据元素集：D={a_{j1j2…jn} | a_{j1j2…jn} ∈ datatype, j_i=0,…,b_i-1 其中 i=1,2,…,n}
        n(n>0)称为数组维数，b_i 是数组第 i 维长度，j_i 是数组元素第 i 维下标。
    数据关系集：R={R_1, R_2, …, R_n}
```

$R_i=\{<a_{j1\cdots ji\cdots jn},a_{j1\cdots ji+1\cdots jn}>\mid 0\leqslant j_k\leqslant b_k-1,1\leqslant k\leqslant n$ 且 $k\neq i,0\leqslant j_i\leqslant b_i-2,a_{j1\cdots ji\cdots jn},a_{j1\cdots ji+1\cdots jn}\in D,i=1,2,\cdots,n\}$

基本操作集：P

ArrayInit(&A,n,d_1,d_2,…,d_n)

操作结果：若维数 n 和各维长度合法，则生成一个 n 维数组 A[d_1][d_2]…[d_n](C 语言中，$1\leqslant n\leqslant 8$)。

ArrayDestroy(&A)

初始条件：数组 A 存在。

操作结果：撤销数组 A。

ArrayGet(A,i_1,…,i_n,&e)

初始条件：数组 A 存在，e∈datatype。

操作结果：若各下标合法，则将数组元素 A[i_1][i_2],…,[i_n]的值传给变量 e。

ArrayAssign(&A,i_1,…,i_n,e)

初始条件：数组 A 存在，e∈datatype。

操作结果：若各下标合法，则将变量 e 的值传给数组元素 A[i_1][i_2],…,[i_n]。

}ADT Array;

其中后面两条是对数组元素随机存取的操作。

5.2　数组的存储结构

5.2.1　数组的静态存储方式

1. 数组的静态存储映像

依据数组的定义，数组的存储方式只能是顺序的。而由于计算机的存储空间是一维的(或线性的)，所以存储数组时，要将多维数组中的元素按某种次序映像到一维存储空间，即解决“降维”问题。

在 Basic、Pascal 和 C 等语言中，是按低维下标优先变化(或按行优先)的方式存储数组中的元素。如在 C 语言中，二维数组的映像如图 5.1 所示。但在 Fortran 语言中，数组元素是按高维下标优先变化(或按列优先)的方式存储数组中的元素。

2. 静态数组元素的地址计算

为了实现数组的随机存取，也就是通过下标直接找到相应地址，就需要建立一个数组的存储位置与下标之间的函数关系。以 C 语言为例。设数组元素的起始地址为 b，每个元素占用 L 个单元(元素所占单元量由元素的类型而定)，元素 a 的地址用 Loc(a)表示。

1) 一维数组

设有如图 5.2 所示的一维数组。

由图 5.2 可知：

$Loc(a_0)=b;$

$Loc(a_1)=b+L;$

⋮

$Loc(a_i)=b+i*L;$

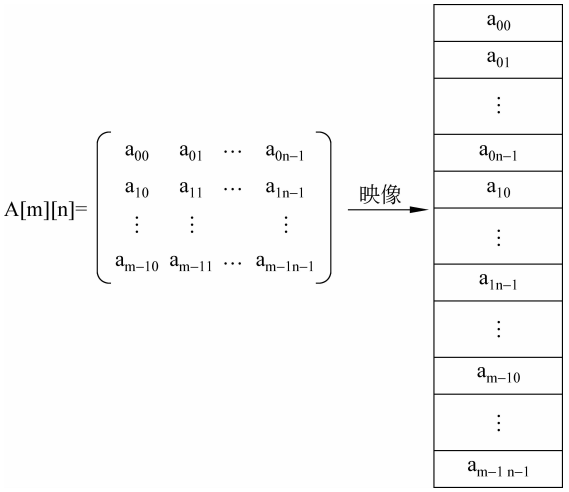


图 5.1 数组映像

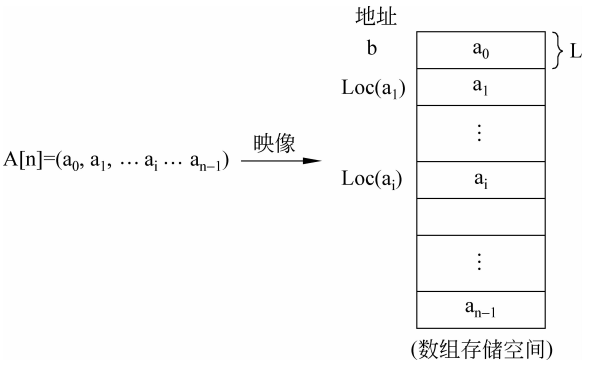


图 5.2 一维数组

即 A[n]中任一元素 a_i 的地址＝(起始地址 b)＋(a_i 前的元素个数 i)＊L。

2) 二维数组

设有如图 5.3 所示的二维数组。

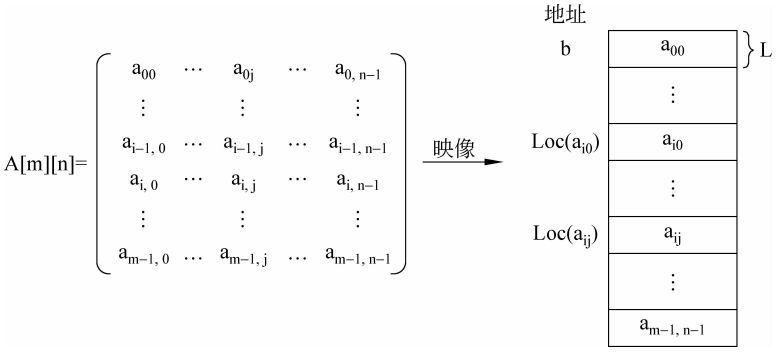


图 5.3 二维数组

由图 5.3 知：

$Loc(a_{00})=b;$

$Loc(a_{i0})=b+(a_{i0}$前的元素个数$)*L$

$=b+(i*n)*L;$

$Loc(a_{ij})=b+(a_{ij}$前的元素个数$)*L$

$=b+(i*n+j)*L;$

例 5-1　设二维数组 A[7][8]，起始地址 b=1000，每个元素所占单元量 L=3，则：

$$Loc(a_{5,6})=1000+(5*8+6)*3=1138$$

3）三维数组

设有如图 5.4 所示的三维数组。

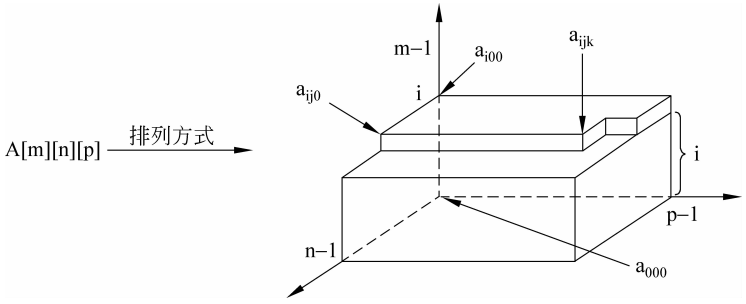


图 5.4　三维数组

由图 5.4 可知：

$Loc(a_{000})=b;$

$Loc(a_{i00})=b+(i*n*p)*L;$

$Loc(a_{ij0})=b+(i*n*p+j*p)*L;$

$Loc(a_{ijk})=b+(i*n*p+j*p+k)*L;$

可以看出，$i*n*p$、$i*n*p+j*p$、$i*n*p+j*p+k$ 分别为 a_{i00}、a_{ij0}、a_{ijk}前的元素个数。

4）n 维数组

从以上的地址公式推导中得出这样一条规律：任意维数组中任一元素的地址为起始地址加上该元素前的元素个数乘以元素单元量。

设 n 维数组 $A[u_1][u_2]\cdots[u_n]$，其中任一元素 $a_{i1\cdots in}$，其地址为：

$$\begin{aligned}Loc(a_{i1i2\cdots in}) &= b+(i_1*u_2*u_3*\cdots*u_n+i_2*u_3*u_4*\cdots\\ &\quad *u_n+\cdots+i_{n-1}*u_n+i_n)*L\\ &= b+\left(\sum_{j=1}^{n-1} i_j \prod_{k=j+1}^{n} u_k+i_n\right)*L\end{aligned}$$

元素按“列优先”方式存储时，地址计算方法类似，此处不再赘述。

有了数组元素的地址计算公式，给出相应参数后，能够直接求出任一元素的地址，然后按地址存取相应元素，故对任意维数组的存取都是随机存取。

5.2.2 数组的动态存储方式

前面讲的数组，其存储空间是在算法执行前就分配的，属于静态分配。这种存储方式对于一些运行之前还不能确定长度的数组，必须要定义足够大的空间以避免溢出。但在某些资源紧张的系统中，这样做就很不实际，于是就用到了数组的动态存储方式。

下面以生成 A[3][4][5][6]为例讨论以下下几点：

(1) 目标是要得到一个以 base 为基址的数组存储空间，本例中总共元素个数是 etotal=3×4×5×6=360。

(2) 各维数下标存入辅助向量 bound，将 bound[i]记为 k_i，在本例中 $k_0=3$，$k_1=4$，$k_2=5$，$k_3=6$。

(3) 令函数映像 $C_i=k_{i+1}\times C_{i+1}$，$C_{n-1}=1$，$0\leqslant i\leqslant n-2$，并将各值存入 const。对本例 $C_3=1$，$C_2=k_3\times C_3=6$，$C_1=k_2\times C_2=30$，$C_0=k_1\times C_1=120$。

(4) 取数组某元素的相对地址。如本例中 a_{1234} 的相对地址 $i=C_0\times 1+C_1\times 2+C_2\times 3+C_3\times 4=202$。

(5) 取元素的绝对地址(实际地址)，绝对地址＝A. base＋相对地址(本例的绝对地址＝A. base＋202)。明确了思路之后，我们就可以写出动态数组的生成及操作算法。

定义数组类型：

```
#difine MAX_DIM 8                                    //最大维数
typedef struct
{   datatype *base;                                  //数组基址
    int dim;                                         //数组维数
    int *bound;                                      //辅助向量 bound
    int *const;                                      //辅助向量 const
}array;
```

1. 数组生成算法

```
int Setarray(array*A,int n,int dim[])                //生成 n 维数组 A 的算法,其中
                                                     //dim[]存放各维数的长度
{   int i,etotal;
    if(n<1||n>MAX_DIM) return (0);                   //非法维数
    (*A).dim=n;                                      //存入维数
    (*A).bound=(int*)malloc(n*sizeof(int));          //生成 bound 的空间
    if(!(*A).bound) return(0);                       //内存分配失败
    etotal=1;
    for(i=0;i<n;i++)
    {   if(dim[i]<0) return(0);                      //非法维长度
        (*A).bound[i]=dim[i];                        //各维下标
        etotal*=dim[i];                              //统计元素个数
    }
    (*A).base=(datatype*)malloc(etotal*sizeof(datatype));      //生成数组空间
    if(!(*A).base) return(0);
```

```
    (*A).const=(int*)malloc(n*sizeof(int));  //生成 const 空间
    if(!(*A).const) return(0);
    (*A).const[n-1]=1;
    for(i=n-2;i>=0;i--)
        (*A).const[i]=(*A).bound[i+1]*(*A).const[i+1];          //给 const 赋值
    return(1);
}
```

2. 求元素相对地址的算法

```
int Index(array A,int d[],int *i)              //求元素相对地址的 i,d[]存放元素下标
{   int j;
    *i=0;
    for(j=0;j<A.dim;j++)
    {   if(d[j]<0||d[j]>A.bound[j]) return(0);//下标越界
        i+=A.const[j]*d[j];                    //求相对地址
    }
    return(1);
}
```

3. 取数组元素地址的算法

```
int Aget(array A,int d[],datatype*e)           //求数组元素地址送 e,d[]中存放下标
{   int k,i;
    k=Index(A,d,&i);                           //取相对地址
    if(k==0) return(0);                        //地址获取失败(非法下标)
    e=A.base+i;                                //取元素地址
    return(1);
}
```

5.3　矩阵的压缩存储

多维数组中最常用的就是二维数组，因为它可以表示一种数学和工程上常用的数学对象——矩阵。有的程序语言甚至还专门提供了矩阵运算的函数。然而对于数学上的一些高阶矩阵来说，如果用常规的方法存储会占用过多的空间。

本节介绍一些矩阵的压缩存储算法，对于一些特定的矩阵，可以有效地压缩存储空间。压缩存储的主要原则就是：对多个值相同的元素只存储其中之一，对 0 元素甚至不分配存储空间。

5.3.1　特殊矩阵的压缩存储

这里的特殊矩阵，指的是值相同的元素或 0 元素在矩阵中的分布遵循一定规律的矩阵。

1. 对称矩阵

设 n 阶方阵(矩阵中元素序号约定从 1 起,以下同):

$$A_{n\times n}=\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & \vdots & \ddots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

若满足 $a_{ij}=a_{ji}$,$(1\leqslant i,j\leqslant n)$,则称 $A_{n\times n}$ 为对称矩阵。

显然,因 a_{ij} 与 a_{ji} 对称相等,二者只需分配一个存储单元,即只存储矩阵中包括主对角线的下三角(或上三角)元素。于是 $A_{n\times n}$ 所需的存储单元数为 $n(n+1)/2$,而不压缩存储需要 n^2 个存储单元。当 n 很大时,几乎能压缩原存储空间的一半。

具体做法是:设置一个一维数组 S[n(n+1)/2+1]作为 $A_{n\times n}$ 的存储空间,且按行的次序存放 $A_{n\times n}$ 中包括主对角线的下三角元素,如图 5.5 所示。其中 a_{ij} 存入 S[k]单元,下标(i,j)与 k 的关系为:

$$k=\begin{cases} i(i-1)/2+j, & \text{当 } i\geqslant j \text{ 时} \\ j(j-1)/2+i, & \text{当 } i<j \text{ 时} \end{cases}$$

当 i≥j 时,k 实际上是矩阵下三角元素 a_{ij} 按行排列的序号。即 a_{11} 的序号为 1,a_{21} 的序号为 2,…a_{ij} 的序号为 k。所以 a_{ij} 的序号 k 为 $a_{11}\sim a_{ij}$ 的元素个数,即 i(i−1)/2+j。另当 i<j 时,即对上三角元素 a_{ij},因 $a_{ij}=a_{ji}$,所以 a_{ij} 的序号=a_{ji} 的序号=j(j−1)/2+i。于是:

$$a_{ij}=\begin{cases} S[i(i-1)/2+j], & \text{当 } i\geqslant j \text{ 时} \\ S[j(j-1)/2+i], & \text{当 } i<j \text{ 时} \end{cases}$$

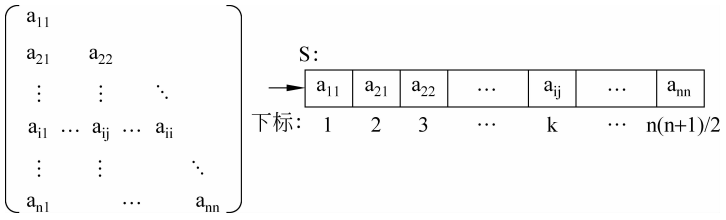


图 5.5 对称矩阵压缩

2. 三角矩阵

设有矩阵:

$$A_{n\times n}=\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ \vdots & \vdots & \ddots & \\ a_{n1} & \cdots & & a_{nn} \end{pmatrix}$$

(下三角矩阵)

或

$$A_{n\times n}=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ & a_{22} & \cdots & a_{2n} \\ & & \ddots & \vdots \\ & & & a_{nn} \end{pmatrix}$$

(上三角矩阵)

称其为下三角矩阵或上三角矩阵，其中 C 为一个常数。

显然，对于下三角矩阵，类似于对称矩阵的压缩存储，即只存储包括主对角线的下三角元素。而当 i<j 时，a_{ij}取 C 即可。对于上三角矩阵，压缩方法如图 5.6 所示。

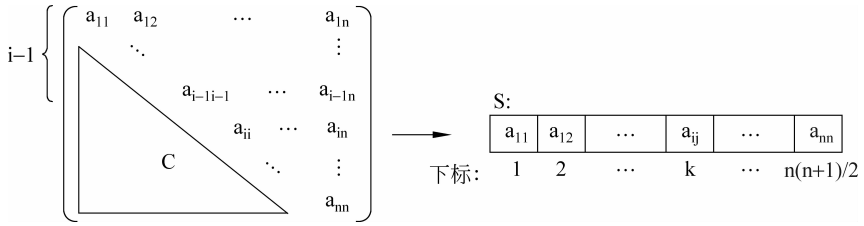


图 5.6　上三角压缩矩阵

同理，当 i≤j 时，数组下标 k 实际上是矩阵的上三角元素 a_{ij}按行排列的序号，即从 a_{11}～a_{ij}的元素个数。故：

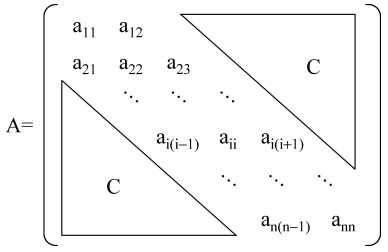
$$k=(i-1)*n-(i-1)(i-2)/2+(j-i+1)=(i-1)(2n-i)/2+j \quad (i\leqslant j)$$

于是

$$a_{ij}=\begin{cases}S[(i-1)(2n-i)/2+j], & \text{当 } i\leqslant j \text{ 时}\\ C, & \text{当 } i>j \text{ 时}\end{cases}$$

3. 对角线矩阵

设包括主对角线的三对角线矩阵：



按行顺序压缩于 S 中，如图 5.7 所示。

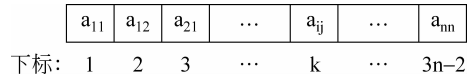


图 5.7　对角线矩阵压缩

元素 a_{ij}的下标(i,j)与数组 S 中下标 k 之间的关系为：

$$k=\begin{cases}3(i-1), & \text{当 } i=j+1 \text{ 时}\\ 3i-2, & \text{当 } i=j \text{ 时}\\ 3i-1, & \text{当 } i+1=j \text{ 时}\end{cases}$$

归纳为：k=2i+j−2(当 i=j+1,i=j,i+1=j)时。

于是

$$a_{ij}=\begin{cases}S[2i+j-2], & \text{当 } i=j+1,i=j,i+1=j \text{ 时}\\ C, & \text{其他}\end{cases}$$

5.3.2 稀疏矩阵的压缩存储

特殊矩阵中同值元素的分布有一定的规律可循，而有的矩阵，0 元素很多（如同一个画面上有几个亮点，其余全是空白），但分布无规律，称这类矩阵为稀疏矩阵。

例 5-2 设一个 6×7 的矩阵如下：

$$A_{6\times7}=\begin{pmatrix}0&1&0&2&0&0&0\\0&0&0&0&0&0&0\\3&0&0&0&0&4&0\\0&0&5&0&0&0&0\\0&6&0&0&0&0&0\\7&0&0&8&0&0&0\end{pmatrix}$$

则 $A_{6\times7}$ 可以视为一个稀疏矩阵。对于矩阵 $A_{m\times n}$，设非 0 元素个数为 t，若 $\delta=t/(m*n)\leqslant 0.2$，则可以将其视为稀疏矩阵。显然，为节省存储空间，须对这类矩阵压缩存储空间，原则是只存储非 0 元素。一般有“三元组表”和“十字链表”的压缩存储方法。

1. 三元组表

三元组为(i,j,v)，其中 i,j 分别为非 0 元素所在的行号和列号，v 存放该非 0 元素的数值。以行优先的顺序将稀疏矩阵中非 0 元素以三元组存入一数组，即所谓的三元组表。对例 5-2 中 $A_{6\times7}$ 的三元组表如图 5.8 所示。设每个元素占 16 个字节，若不压缩存储，需要 6×7×16＝672(字节)，而压缩存储时，i,j 为整型，故共需 2×16＋8×16＝160(字节)。

	i	j	v
1	1	2	1
2	1	4	2
3	3	1	3
4	3	6	4
5	4	3	5
6	5	2	6
7	6	1	7
8	6	4	8

图 5.8 三元组表

三元组表存储结构的描述：

```
#define maxsize 64                  //最大非 0 元个数
typedef Struct                      //三元组类型
{   int i,j;
    datatype v;
}tritype;                           //三元组说明符
typedef Struct
{   tritype data[maxsize+1];        //三元组表存储空间
    int mu,nu,tu;                   //原稀疏矩阵的行、列号和非 0 元素个数
}Tsmtype, *Tsmlink;                 //三元组表说明符
```

若说明：“Tsmlink A;A＝(Tsmlink)malloc(sizeof(Tsmtype));”，则指针变量 A 指向一个如图 5.9 所示的三元组表。稀疏矩阵的行、列号和非 0 元素个数分别为 A->mu、A->nu 和 A->tu。

然而，稀疏矩阵的压缩存储会给矩阵运算带来一些不便，算法要复杂些。这里的运算指求矩阵的转置、两矩阵相加和相乘等。我们只讨论矩阵的转置的算法。