2.5 数组和广义表

- 数组的定义
- 数组的顺序表示
- 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

2.5.1 数组的定义

数组是我们最熟悉的数据类型，在早期的高级语言中，数组是唯一可供使用的数据类型。由于数组中各元素具有统一的类型，并且数组元素的下标一般具有固定的上界和下界，因此，数组的处理比其它复杂的结构更为简单。多维数组是向量的推广。

$$A_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n} \\ a_{21} & a_{22} & \cdots & a_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

数组的特点：
元素数目固定；
下标有界；

数组的操作：
按照下标进行读写

2.5.2 数组的顺序表示和实现

由于计算机的内存结构是一维的，因此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成一列序列，然后将这个线性序列存放在存储器中。

又由于对数组一般不做插入和删除操作，也就是说，数组一旦建立，结构中的元素个数和元素间的关系就不再发生变化。因此，一般都是采用顺序存储的方法来表示数组。

通常有两种顺序存储方式：

(1)行优先顺序——将数组元素按行排列，第i+1个行向量紧接在第i个行向量后面。以二维数组为例，按行优先顺序存储的线性序列为： $a_{11},a_{12},\ldots,a_{1n},a_{21},a_{22},\ldots a_{2n},\ldots\ldots,a_{m1},a_{m2},\ldots,a_{mn}$

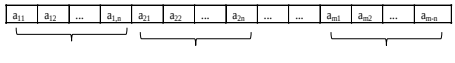
在PASCAL、C语言中，数组就是按行优先顺序存储的。

(2)列优先顺序——将数组元素按列向量排列，第j+1个列向量紧接在第j个列向量之后，A的$m\times n$个元素按列优先顺序存储的线性序列为：

$a_{11},a_{21},\ldots,a_{m1},a_{12},a_{22},\ldots a_{m2},\ldots\ldots,a_{n1},a_{n2},\ldots,a_{nm}$

在FORTRAN语言中，数组就是按列优先顺序存储的。

$$A_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n} \\ a_{21} & a_{22} & \cdots & a_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$



第1行　第2行　第m行

行优先存储

以上规则可以推广到多维数组的情况：优先顺序可规定为先排最右的下标，从右到左，最后排最左下标：列优先顺序与此相反，先排最左下标，从左向右，最后排最右下标。

按上述两种方式顺序存储的序组，只要知道开始结点的存放地址（即基地址），维数和每维的上、下界，以及每个数组元素所占用的单元数，就可以将数组元素的存放地址表示为其下标的线性函数。

例如，二维数组A_{mn}按"行优先顺序"存储在内存中，假设每个元素占用d个存储单元。元素a_{ij}的存储地址应是数组的基地址加上排在a_{ij}前面的元素所占用的单元数。因为a_{ij}位于第i行、第j列，前面i-1行一共有$(i-1)\times n$个元素，第i行上a_{ij}前面又有j-1个元素，故它前面一共有$(i-1)\times n+j-1$个元素，因此，a_{ij}的地址计算函数为：

$LOC(a_{ij})=LOC(a_{11})+[(i-1)*n+j-1]*d$

同样，三维数组A_{ijk}按"行优先顺序"存储，其地址计算函数为：

$LOC(a_{ijk})=LOC(a_{111})+[(i-1)*n*p+(j-1)*p+(k-1)]*d$

更一般的二维数组是

$A[c_1..d_1,c_2..d_2]$

因此，a_{ij}的地址计算函数为：

$LOC(a_{ij})=LOC(a_{c_1c_2})+[(i-c_1)*(d_2-c_2+1)+j-c_2)]*d$

2.5.3 特殊矩阵的压缩存储

特殊矩阵进行压缩存储：即为多个相同的非零元素只分配一个存储空间；对零元素不分配空间。

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

一个$m\times n$的矩阵。

特殊矩阵

所谓特殊矩阵就是元素值的排列具有一定规律的矩阵。常见的这类矩阵有：对称矩阵、下（上）三角矩阵、对角线矩阵等等。

$$\begin{pmatrix} 10 & 5 & 3 & 17 \\ 5 & 7 & 12 & 4 \\ 3 & 12 & 20 & 23 \\ 17 & 4 & 23 & 14 \end{pmatrix}$$

对称矩阵

下（上）三角矩阵的特点是以主对角线为界的上（下）半部分是一个固定的值，下（上）半部分的元素值没有任何规律。

$$\begin{pmatrix} 29 & 0 & 0 & 0 \\ 6 & 12 & 0 & 0 \\ 8 & 10 & 30 & 0 \\ 13 & 26 & 9 & 20 \end{pmatrix}$$

对角矩阵的特点是所有的非零元素都集中在以主对角线为中心的带状区域中。

$$\begin{pmatrix} 3 & 12 & 0 & 0 & 0 \\ 9 & 5 & 20 & 0 & 0 \\ 0 & 30 & 7 & 17 & 0 \\ 0 & 0 & 21 & 9 & -6 \\ 0 & 0 & 0 & 34 & 11 \end{pmatrix}$$

1.对称矩阵

对称矩阵的特点是$a_{ij}=a_{ji}$。一个$n\times n$的方阵，共有n^2个元素，但只需要对称矩阵中$n(n+1)/2$个元素进行存储表示。

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

顺序"存储主对角线（包括对角线）以下的元素，其存储形式如图所示：

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

a_{11}

a_{21} a_{22}

a_{31} a_{32} a_{33}

……

a_{n1} a_{n2} a_{n3} …a_{nn}

a_{11}	a_{21}	a_{22}	a_{31}	……	a_{n1}	…	a_{nn}

$$k=\begin{cases} \dfrac{i(i-1)}{2}+j-1 & \text{当} i \geq j \\ \dfrac{j(j-1)}{2}+i-1 & \text{当} i < j \end{cases}$$

k是对称矩阵位于（i，j）位置的元素在一维数组M[0..max]中的存放位置。

若$i \geq j$，则a_{ij}在下三角形中。 a_{ij}之前的i行（从第1行到第i-1行）一共有$1+2+\ldots+i=i(i+1)/2$个元素，在第i行上， a_{ij}之前恰有j个元素（即$a_{i0},a_{i1},a_{i2},\ldots,a_{ij-1}$），因此有：

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

若$i<j$，则a_{ij}是在上三角矩阵中。因为$a_{ij}=a_{ji}$，所以只要交换上述对应关系式中的i和j即可得到：

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

2、三角矩阵

以主对角线划分，三角矩阵有上三角和下三角两种。上三角矩阵如图所示，它的下三角（不包括主对角线）中的元素均为常数。下三角矩阵正好相反，它的主对角线上方均为常数。

$$\begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0\,n-1} \\ c & a_{11} & \cdots & a_{1\,n-1} \\ \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ c & c & \cdots & a_{n-1\,n-1} \end{pmatrix} \qquad \begin{pmatrix} a_{00} & c & \cdots & c \\ a_{10} & a_{11} & \cdots & c \\ \cdots\cdots & \cdots\cdots & \cdots & \cdots\cdots \\ a_{n-1\,0} & a_{n-1\,1} & \cdots & a_{n-1\,n-1} \end{pmatrix}$$

(a)上三角矩阵　　(b)下三角矩阵

3、 稀疏矩阵的压缩存储

若一个$m \times n$的矩阵含有t个非零元素，且t远远小于$m \times n$，则我们将这个矩阵称为稀疏矩阵。

$$\begin{pmatrix} 3 & 0 & 0 & 0 & 7 \\ 0 & 0 & -1 & 0 & 0 \\ -1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

三元组表示法。

矩阵中的每个元素都是由行序号和列序号唯一确定的。因此，我们需要用三项内容表示稀疏矩阵中的每个非零元素，即形式为：

(*i*,*j*,*value*)

其中，*i*表示行序号，*j*表示列序号，*value*表示非零元素的值，通常将它称为三元组。

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


```
Typedef Struct
{ int I,j;
  elementtype e;
} Triple;

Typedef Struct
{  Triple data[Max+1];
    int mu,nu,tu;
} TSMatrix
```

稀疏矩阵的转置（三元组存储）

$$\begin{pmatrix} 3 & 0 & 0 & 0 & 7 \\ 0 & 0 & -1 & 0 & 0 \\ -1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

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

$$\begin{pmatrix} 3 & 0 & -1 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 7 & 0 & 0 & 0 & 0 \end{pmatrix}$$

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

下面以矩阵的转置为例，说明在这种压缩存储结构上如何实现矩阵的运算。

一个$m\times n$的矩阵A，它的转置B是一个$n\times m$的矩阵，且a[i][j]=b[j][i]，$0 \le i \le m$，$0 \le j \le n$，即A的行是B的列，A的列是B的行。

由于A的列是B的行，因此，按a.data的列序转置，所得到的转置矩阵B的三元组表b.data必定是按行优先存放的。

转置算法

```
Void TransMatrix(TSMatrix M,TSMatrix T)
//M 和T 是矩阵的三元组表示, T 是转置矩阵
 {T.mu=M.nu; T.nu=M.mu;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=q+1;   }
           }
    }
}
```

表示三元组T的下标

按M的列值由小到大查找

分析这个算法，主要的工作是在p和col的两个循环中完成的，故算法的时间复杂度为$O(n*t)$，即矩阵的列数和非零元的个数的乘积成正比。

下面给出另外一种称之为快速转置的算法，其算法思想为：对A扫描一次，按A第二列提供的列号一次确定位置装入B的一个三元组。具体实施如下：一遍扫描先确定三元组的位置关系，二次扫描由位置关系装入三元组。可见，位置关系是此种算法的关键。

为了预先确定矩阵M中的每一列的第一个非零元素在数组B中应有的位置，需要先求得矩阵M中的每一列中非零元素的个数。因为：矩阵M中第一列的第一个非零元素在数组B中应有的位置等于前一列第一个非零元素的位置加上前列非零元素的个数。

为此，需要设置两个一维数组num[1..n]和cpot[1..n]

num[1..n]：统计M中每列非零元素的个数，num[col]的值可以由A的第二列求得。

cpot[1..n]：由递推关系得出M中的每列第一个非零元素在B中的位置。

算法通过cpot数组建立位置对应关系：

cpot[1]=1

cpot[col]=cpot[col-1]+num[col-1]

2<=cpl<=a.n

快速转置算法如下：

```
void fasttranstri(tritupletable b,tritupletable a){
 int p,q,col,k;
 int num[0..a.n],copt[0..a.n];
 b.m=a.n; b.n=a.m; b.t=a.t;
 if(b.t<=0)
    printf("a=0"\n);
 for(col=1;col<=a.u;++col)
       num[col]=0;
 for(k=1;k<=a.t;++k)
      ++num[a.data[k].j];
```



```
 cpot[1]=1;
 for(col=2;col<=a.t;++col)
     cpot[col]=cpot[col-1]+num[col-1];
 for(p=1;p<=a.t;++p){
   col=a.data[p].j;  q=cpot[col];
   b.data[q].i=a.data[p].j;
   b.data[q].j=a.data[p].i;
    b.data[q].v=a.data[p].v;    ++cpot[col];
  }
 }
}
```


2.5.4 广义表的定义

线性表的推广，示例：

LS=(a1,a2,…,an)
A=()
B=(e)
C=(a, (b,c,d))
D=(A, B, C)
E=(a, E)

广义表的元素可以是子表，子表的元素还可以是子表；

广义表是一个多层次的结构（层次性）；

一个广义表可以被其他广义表所共享（共享性）。

广义表可以是其本身的子表（递归性）。

广义表的长度：元素的数目。
广义表的表头：非空广义表中第一个元素。
广义表的表尾：除表头元素之外，其余元素构成的表。
广义表的深度：广义表中括号的重数。

	长度	表头	表尾	深度
A=()	0			1
B=(e)	1	e	()	1
C=(a,(b,c,d))	2	a	((b,c,d))	2
D=(A,B,C)	3	()	(B,C)	3
E=(a,E)	2	a	(E)	无穷大