Chapter 5 第5章 数组与广义表

前几章讨论的线性结构数据元素都是非结构的原子类型，元素的值不再分解。本章讨论的数组和广义表可以看作线性表的扩展，即表中元素本身也是一个数据结构。

5.1 数组的定义

数组是读者广泛熟知的数据类型，几乎所有的程序设计语言都会把数组作为固定的数据类型。本章以抽象数据类型的形式讨论数组的定义和实现，以及其在智能算法中的应用，以加深读者对数组的理解。

5.1.1 数组的定义和术语

数组是由下标(index)和值(value)组成的序对(index,value)的集合，也可以定义为是由**相同类型**的数据元素组成的有限序列。每个元素对应的下标都对应一组由 n($n\geqslant 1$)个线性关系构成的约束$(j_1,j_2,\cdots,j_n)$，其中每个 $j_i\in[0,b_i-1]$，b_i 是第 i 维的长度($i=1,2,\cdots,n$)。我们称这样的序列为 n 维数组。

示例：

一维数组：$(a_1,a_2,\cdots,a_n)$

二维数组：$(a_{11},\cdots,a_{1n},a_{21},\cdots,a_{2n},\cdots,a_{ij},\cdots,a_{mn})$

$1\leqslant i\leqslant m,1\leqslant j\leqslant n$

三维数组：$(a_{111},\cdots,a_{11n},a_{121},\cdots,a_{12n},\cdots,a_{ijk},\cdots,a_{mnl},\cdots,a_{mnp})$

$1\leqslant i\leqslant m,1\leqslant j\leqslant n,1\leqslant k\leqslant p$

…

(这里的 m,n,p 分别是某一维的长度，相当于定义中的 b_i)

由上可知，当 $n=1$ 时，n 维数组就退化为定长的线性表；反之，n 维数组可以看作线性表的推广。因此，我们还可以从线性表的角度来定义 n 维数组。如图 5-1 所示，二维数组 $A_{m\times n}$ 可以定义为一维数组的一维数组，或线性表的线性表。同理，n 维数组可以看作数据类型为 n−1 维数组的一维数组。

$$A_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

(a)

$$\begin{cases} A_{m\times n}=(A_1,A_2,\ldots,A_n) \\ A_i=(a_{1i},a_{2i},\ldots,a_{mi}),\ 1\leqslant i\leqslant n \end{cases}$$

(b)

$$\begin{cases} A_{m\times n}=(A_1,A_2,\ldots,A_m) \\ A_j=(A_{j1},A_{j2},\ldots,A_{jn}),\ 1\leqslant j\leqslant m \end{cases}$$

(c)

图 5-1 二维数组图例

(a)矩阵形式表示；(b)列向量的一维数组；(c)行向量的一维数组

5.1.2　数组的抽象数据类型

数组的抽象数据类型定义为

```
ADT Array{
    数据对象: j_i=0,…,b_i-1,i=1,2,…,n
              D={a_{j1j2…jn} |n>0 称为数组维数,b_i 是数组第 i 维的长度,
              j_i 是数组元素的第 i 维下标,a_{j1j2…jn} ∈ElemSet}
    数据关系: R={R1,R2,…,Rn}
              Ri={<a_{j1…ji…jn},a_{j1…ji+1…jn}> |
                        0≤j_k≤b_k-1, 1≤k≤n and k≠i,
                        0≤j_i≤b_i-2,
                        a_{j1…ji…jn},a_{j1…ji+1…jn} ∈D, i=2,…,n}
    基本操作:
        Create(&A,n,bound1,bound2,…,boundn)
            操作结果:输入合法时,构造数组 A,返回 OK
        Retrieve(&A,index1,index2,…,indexn)
            操作结果:输入合法时,给定下标,返回对应的数组元素
        Store(&A,index1,index2,…,indexn,value)
            操作结果:输入合法时,将 value 赋值给对应下标的数组元素,返回 OK
}ADT Array
```

5.2　数组的存储与实现

数组没有插入和删除操作,元素之间的位置关系不会发生变化。因此,数组在内存中使用一组连续的地址空间存储,能够实现下标运算。

5.2.1　数组的顺序存储

数组的顺序指的是在计算机中用一组连续的存储单元来实现数组的存储。从逻辑层面来看,数组因下标约束形成了多维的结构;但从物理空间来看,数组是一个一维向量,因此存在一个次序约定问题。如图 5-1 所示,二维数组有两种存储方式:以列序为主序存储,如图 5-1(b)所示;以行序为主序存储,如图 5-1(c)所示。目前,高级程序设计语言都以连续顺序空间来存储数组,其中,C、PASCAL 等按行存储,而 FORTRAN 等则按列存储。

不同的存储方式有着不同的地址计算方法,一旦确定了数组的维度、各维的长度、次序,就可以确定任意元素的存储位置。

以按行存储为例,假设每个元素占据 L 个存储单元,数组 A 的起始位置记为 LOC(0),那么对于:

二维数组 $A[b_1,b_2]$中元素 a_{ij}的位置为

$$LOC(i,j)=LOC(0,0)+(b_2\times i+j)L \tag{5-1}$$

三维数组 $A[b_1,b_2,b_3]$中元素 a_{ijk}的位置为

$$LOC(i,j,k)=LOC(0,0,0)+(b_2\times b_3\times i+b_2\times j+k)L \tag{5-2}$$

n 维数组 $A[b_1,b_2,\cdots,b_n]$中元素 $a_{j_1j_2\cdots j_n}$的位置为

$$LOC(j_1,j_2,\cdots,j_n)=LOC(0,0,\cdots,0)+(b_2\times\cdots\times b_n\times j_1+b_3\times\cdots\times b_n\times j_2+\cdots+b_n\times j_{n-1}+j_n)L \tag{5-3}$$

同理,对于按列存储有:

二维数组 $A[b_1,b_2]$中元素 a_{ij}的位置为

$$LOC(i,j)=LOC(0,0)+(i+b_1\times j)L \tag{5-4}$$

三维数组 $A[b_1,b_2,b_3]$中元素 a_{ijk}的位置为

$$LOC(i,j,k)=LOC(0,0,0)+(i+b_1\times j+b_1\times b_2\times k)L \tag{5-5}$$

n 维数组 $A[b_1,b_2,\cdots,b_n]$中元素 $a_{j_1j_2\cdots j_n}$的位置为

$$LOC(j_1,j_2,\cdots,j_n)=LOC(0,0,\cdots,0)+(j_1+b_1\times j_2+\cdots+b_1\times\cdots\times b_{n-1}\times j_n)L \tag{5-6}$$

容易看出，无论是按行存储还是按列存储，只要确定了各个维度的长度 $b_1\sim b_n$，元素位置 LOC 就是关于下标$(j_1,j_2,\cdots,j_n)$的线性函数。那么，计算各个元素存储位置的时间是相等的，也就意味着存取数组中任一元素的时间也相等。具有这一特点的存储结构一般称作**随机存储结构**。

下面是数组的顺序存储表示和实现。

```
//------ 数组的顺序表示 ------
#include <stdarg.h>                          //标准头文件,提供 va_start、va_arg、
                                             //va_end,用于存取变长参数表
#define MAX_ARRAY_DIM 8                      //假设数组最多不超过 8 维
typedef struct{
    Elemtype *base;                          //数组元素基址,由 Create 分配
    int dim;                                 //数组维数
    int *bounds;                             //数组维界基址,由 Create 分配
    int *constants;                          //数组映像函数常量基址,由 Create 分配
}Array;

//------ 数组基本操作的算法描述 ------
Status Create(Array &A, int dim, ...){
    //维数合法性判定
    if (dim < 1 || dim > MAX_ARRAY_DIM) return ERROR;
    A.dim = dim;
    //申请维界存储空间
    A.bounds = (int*)malloc(dim * sizeof(int));
    if (!A.bounds) exit(OVERFLOW);
    //记录元素总数
    int elem_sum = 1;
    //读取变长参数表,ap 为 va_list 类型
    va_start(ap, dim);                       //ap 读到的是维度 dim 之后的参数
    int i;
    for (i = 0; i < dim; i++){
        A.bounds[i] = va_arg(ap, int);
        //维界合法性判定
        if (A.bounds[i] < 0) return UNDERFLOW;
        elem_sum *= A.bounds[i];
    }
    va_end(ap);
    //申请数组元素存储空间
    A.base = (Elemtype*)malloc(elem_sum * sizeof(Elemtype));
    if (!A.base) exit(OVERFLOW);
    //申请映像函数常量的存储空间
    A.constants = (int*)malloc(dim * sizeof(int));
    if (!A.constants) exit(OVERFLOW);
    //按行存储,计算每个维度的系数
    A.constants[dim-1] = 1;
    for (i = dim-2; i >= 0; i--){
```

```
        A.constants[i] = A.bounds[i+1] * A.constants[i+1];
    }
    //如果是按列存储,则按以下方式计算
    //A.constants[0] = 1;
    //for (i = 1; i < dim; i++){
        //A.constants[i] = A.bounds[i-1] * A.constants[i-1];
    //}
    return OK;
}

Elemtype Retrieve(Array A, ...){
    //获取下标
    va_start(ap, A);                          //ap 读到的是数组 A 之后的参数
    //计算给定下标相对于基址的偏移位置
    int i, offset = 0;
    for (i = 0; i < A.dim; i++){
        index = va_arg(ap, int);
        if (ind < 0 || ind >= A.bounds[i]) return OVERFLOW;
        offset += A.constants[i] * index;
    }
    //取值
    return * (A.base + offset);
}

Status Store(Array A, e, ...){
    //获取下标
    va_start(ap, e);                          //ap 读到的是 e 之后的参数
    //计算给定下标相对于基址的偏移位置
    int i, offset = 0;
    for (i = 0; i < A.dim; i++){
        index = va_arg(ap, int);
        if (ind < 0 || ind >= A.bounds[i]) return OVERFLOW;
        offset += A.constants[i] * index;
    }
    //存值
    * (A.base + offset) = e;
    return OK;
}
```

5.2.2　数组的压缩存储

“万物皆矩阵”,矩阵是很多科学与工程问题中重点研究的数学对象。在计算机领域,我们十分关心矩阵的存储,以便于有效进行各种运算。通常,高级语言使用二维数组来存储矩阵,有的语言还会提供矩阵的运算,以方便用户使用。

然而,高阶矩阵的存储与运算开销是巨大的。对于某些比较特殊的矩阵(存在较多值相同的元素或者零元素),为了节省存储空间以及提高运算效率,可以对这类矩阵进行压缩存储。压缩存储的基本思想是:为值相同的元素分配同一存储空间,对零元素不分配空间。

这里,我们主要讨论两类矩阵的压缩存储。

1. 特殊矩阵

特殊矩阵是指值相同的元素或者零元素在分布上呈现一定规律的矩阵。

先看 n 阶对称矩阵 A:

$$a_{ij}=a_{ji}\quad 1\leqslant i,j\leqslant n$$

对于对称矩阵，我们可以为每一对对称元分配一个存储空间，如此可将 n^2 规模的元素压缩存储到 $n(n+1)/2$ 规模的空间中。不失一般性，我们以行序为主序，存储矩阵下三角部分的元素。

假设一维数组 sa[n(n+1)/2]是矩阵 A 压缩后的存储结构，那么 sa[k]和 a_{ij} 的映射关系如下：

$$k=\begin{cases}\dfrac{i(i-1)}{2}+j-1, & i\geqslant j\\ \dfrac{j(j-1)}{2}+i-1, & i<j\end{cases} \tag{5-7}$$

任意给定下标(i,j)，均可在 sa 中找到对应的 a_{ij}；给定 k=0,1,…,n(n+1)/2−1，都能确定 sa[k]在矩阵 A 中的位置(i,j)。二者一一对应，我们称 sa[n(n+1)/2]是对称矩阵 A 的压缩存储(见图 5-1)。

a_{11}	a_{21}	a_{22}	a_{31}	…	a_{ij}	…	a_{n-1}	…	a_{nn}
k= 0	1	2	3		$\frac{i(i-1)}{2}+j-1$		$\frac{n(n-1)}{2}$		$\frac{n(n+1)}{2}-1$

图 5-1 对称矩阵的压缩存储

上述压缩存储方法同样适用于三角矩阵。下(上)三角矩阵的上(下)三角部分均为零元素或者常数 c，则和对称矩阵一样，使用一维数组 sa[n(n+1)/2]，外加一个存储常数 c 的存储空间即可。

在数值分析中，还有一类特殊矩阵：对角(带状)矩阵，即非零元素集中在以主对角线为中心的带状区域内(狭义的对角矩阵是指主对角线以外的元素均为零，注意区别)，如图 5-2 所示。

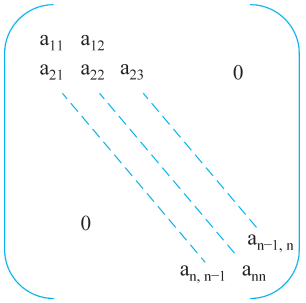

图 5-2 对角矩阵，带宽 s 为 3

和对称矩阵的压缩存储类似，假设一维数组 sa[N]是对角矩阵 A 压缩后的存储结构，那么 sa[k]和 a_{ij} 的映射关系如下：

$$k=\begin{cases}1+s\times(i-1)+(j-i) & abs(i-j)\leqslant\dfrac{s-1}{2}\\ 0 & others\end{cases} \tag{5-8}$$

其中，s 是矩阵的带宽，由上述对角矩阵的定义容易知道 s 只能为奇数。一维数组 sa 的长度 N 由矩阵维度 n 和带宽 s 共同决定：

$$N=n+2\times\sum_{i=1}^{\frac{s-1}{2}}(n-i) \tag{5-9}$$

2. 稀疏矩阵

稀疏矩阵是指非零元比较少且分布没有规律的矩阵。假设在矩阵 $A_{m\times n}$ 中有 t 个非零元，则称 δ=t/(m×n)为矩阵 A 的稀疏因子。一般地，当 δ≤0.05 时，矩阵被认为是稀疏矩阵。

稀疏矩阵的抽象数据类型定义如下：

```
ADT SparseMatrix{
    数据对象: D={aij|i=1,2,…,m;j=1,2,…,n;
                                aij∈ElemSet,m 和 n 分别为矩阵的行数和列数}
    数据关系: R={Row,Col}
              Row={<aij,ai,j+1>|1≤i≤m,1≤j≤n-1}
              Col={<aij,ai+1,j>|1≤i≤m-1,1≤j≤n}
```

```
    基本操作:
        CreateSMatrix(&M)
            操作结果:构造稀疏矩阵数组 M
        DestorySMatrix(&M)
            操作结果:销毁稀疏矩阵数组 M
        TransposeSMatrix(M,&T)
            操作结果:求稀疏矩阵 M 的转置矩阵 T
        MultSMatrix(M,N,&Q)
            操作结果:求稀疏矩阵 M,N 的乘积 Q
}ADT SparseMatrix
```

如何对稀疏矩阵进行压缩呢?

我们仍然希望减少零元素的存储,并提高运算的效率:尽可能快地找到与下标(i,j)对应的元素,以及尽可能快地找到同一行或同一列的非零元素。因此容易想到,我们只存储非零元素。考虑一个三元组(i,j,a_{ij}),它唯一确定了某元素的行列位置和值。如表 5-1 和图 5-3 所示,以(6,7)作为矩阵的行列规模,表 5-1 的三元组表分别对应图 5-3 中的矩阵 M 和其转置矩阵 T。

表 5-1　三元组表

i	j	v
1	2	12
1	3	9
3	1	−3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	−7

(a) 矩阵M的三元组

i	j	v
1	3	−3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	−7
6	3	14

(b) 矩阵T的三元组表

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

图 5-3　稀疏矩阵 M 和其转置矩阵 T

如果将每个三元组看作一个元素,那么我们可以用一维数组的顺序存储结构表示矩阵的三元组表,称为三元组顺序表。

```
//------ 稀疏矩阵的三元组顺序表存储表示 ------
#define MAXSIZE 12500                          //规定非零元个数的最大值为 12500
typedef struct{
    int i,j;                                   //非零元的行列下标
    ElemType e;                                //非零元的值
}Triple;
typedef struct{
    Triple data[MAXSIZE+1];                    //非零元三元组表,data[0]未用
    int mu, nu, tu;                            //矩阵的行数、列数以及非零元个数
}TSMatrix;
```

其中，data 域中表示非零元素的三元组是以行序为主序顺序排列的。

下面我们介绍基于该压缩结构实现稀疏矩阵的转置算法。

观察表 5-1 中(a)与(b)的差异，我们发现，实现矩阵转置，只需要做到：①交换矩阵的行列值；②交换每个三元组的 i 和 j；③按照主序(这里即行序)重新排列三元组的次序。其中，前两点容易做到，关键在于实现③。

根据转置矩阵的性质，表 5-1(b)的三元组表相对于原矩阵 M 为序列，所以我们可以按照(b)的次序依次在(a)中找到相应的三元组进行转置。为了找到 M 每列中的所有非零元素，需要对(a)从第一行开始扫描。由于(a)以 M 的行序为主序来存放非零元素，由此得到的恰好是(b)应有的顺序。算法描述如算法 5-1 所示。

```
void TransposeSMatrix(TSMatrix M, TSMatrix &T){
    T.mu = M.nu; T.nu = M.mu; T.tu = M.tu;
    if(T.tu){
        q = 1;
        //逐列扫描矩阵 M
        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;
                }
            }
        }
    }
}
```

算法 5-1

分析该算法，其主要工作集中于两重 for 循环，时间复杂度为 O(M.nu×M.tu)。我们知道，一般的基于二维数组存储的矩阵转置算法为

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

其时间复杂度为 O(M.mu×M.nu)。当非零元素的个数 M.tu 与 M.mu×M.nu 同量级时，基于三元组的算法 5-1 的复杂度将达到 $O(M.mu\times M.nu^2)$，节省了部分存储空间却显著提高了计算复杂度，因此算法 5-1 仅适用于 M.tu≪M.mu×M.nu 的情况。

更进一步，我们发现在算法 5-1 的两重循环中存在重复扫描的冗余操作。为了优化这一不足，我们希望预先得到表 5-1(a)经过转置后在(b)中恰当的位置，然后直接放入即可。

在此，需要预设两个辅助向量：num[col]表示矩阵 M 中第 col 列的非零元素个数，cpot[col]表示矩阵 M 中第 col 列的第一个非零元素在(b)中的恰当位置，显然有

$$\text{cpot}[\text{col}]=\begin{cases}1 & \text{col}=1\\ \text{cpot}[\text{col}-1]+\text{num}[\text{col}-1] & 2\leqslant \text{col}\leqslant M.nu\end{cases} \tag{5-10}$$

以图 5-3 的矩阵为例，得到的 num 和 cpot 值如表 5-2 所示。

表 5-2　矩阵 M 的向量 cpot 值

col	1	2	3	4	5	6	7
num[col]	2	2	2	1	0	1	0
cpot[col]	1	3	5	7	8	8	9

借助这两个辅助向量，我们得到一种新的转置方法，称为快速转置算法，如算法 5-2 所示。

```
void FastTransposeSMatrix(TSMatrix M, TSMatrix &T){
    T.mu = M.nu; T.nu = M.mu; T.tu = M.tu;
    if(T.tu){
        //计算每列的非零元素个数
        for(col = 0; col <= M.nu; ++col) num[col] = 0;
        for(t = 1; t <= M.tu; ++t) ++num[M.data[t].j];
        //计算每列的首个非零元素的恰当位置
        cpot[1] = 1;
        for(col = 2; col <= M.nu; ++col)
            cpot[col] = cpot[col-1] + num[col-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];                    //更新对应列的位置信息
        }
    }
}
```

算法 5-2

算法 5-2 使用了四个并列的单循环，循环次数分别为 M.nu 和 M.tu，因此时间复杂度为 O(M.nu+M.tu)。当 M.tu 与 M.mu×M.nu 同量级时，该算法的复杂度退化为 O(M.mu×M.nu)，与经典算法一致。

5.2.3　数组的链式存储

上文提到，由于数组的逻辑特性和物理特性，一般采用顺序存储结构。然而，对于多维数组，有些情况下不宜采用顺序存储。例如，使用三元组顺序表存储的稀疏矩阵，在执行加法操作时，非零元素的新增和减少会引起一维数组的变化，而这种变化可能十分频繁。在此情况下，采用链式存储结构表示三元组的线性表更为恰当。

在链表中，每个非零元素被表示为一个含有 5 个域的结点，其中，i,j,e 三个域分别表示该元素的行、列和值，余下的两个指针域分别指向同一行和同一列的下一个非零元素。理论上，这两个指针域可以选择矩阵的四个角中的任意一个，此处采用向左域 left 指向同一行左边的第一个非零元素、向上域 up 指向同一列上方的第一个非零元素，即元素由矩阵的右下角往左上方传递。每个非零元素通过指针域，既连接着行链表，又连接着列链表，故又称之为十字链表，行链表的头和列链表的头分别用两个一维数组来表示。图 5-4 展示了稀疏矩阵的十字链表。

算法 5-3 是稀疏矩阵的十字链表表示和建立十字链表的算法。

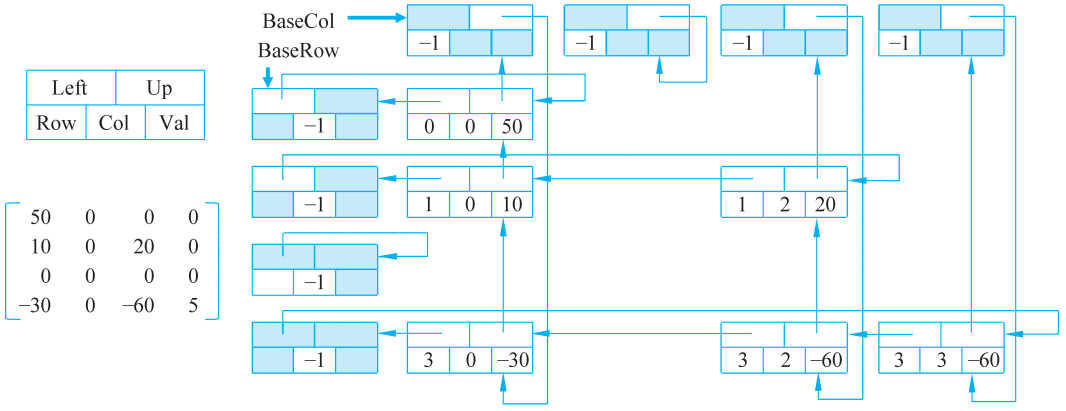

图 5-4 稀疏矩阵的十字链表

```
//------ 稀疏矩阵的十字链表存储表示 ------
typedef struct OLNode{
    int i, j;                                   //元素的行列坐标
    ElemType e;                                 //元素的值
    struct OLNode * left, * up;                 //指针域
}OLNode; * OLink;
typedef struct{
    OLink * baserow, * basecol;                 //行列链表表头指针向量基址
    int mu, nu, tu;                             //矩阵的行数、列数以及非零元素的个数
}CrossList;

void CreateSMatrix_OL(CrossList &M){
    if(M) free(M);
    //输入矩阵的行、列、非零元素的个数
    scanf(&m, &n, &t);
    M.mu = m; M.nu = n; M.tu = t;
    //初始化行列头指针向量,初始行列各链表均为空
    if(!(M.baserow = (OLink *)malloc((m+1) * sizeof(OLink))))
        exit(OVERFLOW);
    if(!(M.basecol = (OLink *)malloc((n+1) * sizeof(OLink))))
        exit(OVERFLOW);
    M.baserow[] = M.basecol[] = NULL;
    //输入矩阵的非零元素,每个非零元素都要完成行插入和列插入
    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;
        if(M.baserow[i] == NULL || M.baserow[i]->j < j{
            p->left = M.baserow[i];
            M.baserow[i] = p;
        }else{
            for(q=M.baserow[i];(q->left)&&q->left->j>j;q=q->left);
            p->left = q->left; q->left = p;
        }                                       //完成行插入
        if(M.basecol[j] == NULL || M.basecol[j]->i < i{
            p->up = M.basecol[j];
            M.basecol[j] = p;
        }else{
            for(q=M.basecol[j]; (q->up)&&q->up->i>i; q=q->up);
            p->up = q->up; q->up = p;
        }                                       //完成列插入
    }
}
```

算法 5-3

对于 m 行 n 列含有 t 个非零元素的稀疏矩阵，上述算法的执行时间为 O(t×max(m,n))，此算法在插入每个非零元素时都要查询它在行、列链表的插入位置，因此不需要考虑输入元素的顺序问题。如果输入元素的次序满足以行序为主序，那么可以简化上述算法达到 O(t)的复杂度。

思考：根据以上创建十字链表的过程，请尝试实现：①将两个指针域换成 right 和 down（或 right-up，left-down 组合），改写算法 5-3；②基于十字链表的稀疏矩阵的加法（习题 5.5）。

5.3　数组的智能算法应用

数组的用途十分广泛，在机器学习、人工智能领域，问题往往会被转换为对数组的一系列操作。人脸识别在当代社会生活中有着成熟便捷的应用，例如扫脸支付、扫脸进校门、火车站机场安检等。2DPCA（Two-Dimensional Principal Components Analysis）是人脸识别技术中的经典算法，相比于传统 PCA 将二维矩阵压缩为一维向量的做法，它直接利用人脸原始图像矩阵来进行特征提取，下面讲述该算法的大体思想。

1. 特征构造

对于原始图像矩阵 $A_{m\times d}$，我们期望通过下列线性变换构造其投影特征向量：

$$Y=AX$$

其中，X 是一个 n 维的酉列向量（unitary column vector），在此又被称作投影向量。那么如何找到最佳的投影向量 X 使得特征向量 Y 具有最好的特征提取效果呢？我们需要通过样本的散点度衡量，而散点度是通过计算特征向量 Y 的协方差矩阵的迹（trace）得到的：

$$\begin{cases} J(X)=\mathrm{tr}(S_x) \\ Y_1,Y_2,\cdots,Y_d \end{cases}$$

上式中，S_x 为特征向量 Y 的协方差矩阵，当 J(X)达到最大值时，说明特征向量 $Y_1,Y_2,\cdots,Y_d$ 是最佳的。由协方差的定义

$$\begin{aligned} S_x=E(Y-EY)(Y-EY)^{\mathrm{T}}&=E[AX-E(AX)][AX-E(AX)]^{\mathrm{T}} \\ &=E[(A-EA)X][(A-EA)X]^{\mathrm{T}} \end{aligned}$$

该协方差矩阵的迹为

$$\mathrm{tr}(S_x)=X^{\mathrm{T}}[E(A-EA)^{\mathrm{T}}(A-EA)]X$$

考虑原始图像矩阵 A 的协方差矩阵

$$G_t=\frac{1}{M}\sum_{j=1}^{M}(A_j-\overline{A})^{\mathrm{T}}(A_j-\overline{A})$$

借助一些线性代数的知识，可得到

$$J(X)=X^{\mathrm{T}}G_tX$$

因此，满足最佳投影向量的条件是

$$\begin{cases} \{X_1,X_2,\cdots,X_d\}=\mathrm{argmax}J(X) \\ X_i^{\mathrm{T}}X_j=0,i\neq j,i,j=1,2,\cdots,d \end{cases}$$

即投影向量两两正交时，我们得到 J(X)的最大值，也就找到了最佳特征向量。

至此，我们得到原始图像 A 的特征矩阵 $B_{m\times d}=[Y_1,Y_2,\cdots,Y_d]$，又被称作图像 A 的主成分（principal component）。

2. 分类方法

接下来，该算法使用一个最近邻分类器来对图像进行分类。特征矩阵的距离表示为