# 第5章 数组和广义表

数组是线性表的推广形式，广泛用于日常生活中，如表1.1表示的表格就是一个典型的二维数组，数组中的某个元素根据行和列的位置可能出现行的前驱和后继，或者是列的前驱和后继，因而一个元素可能有多个前驱和多个后继。

广义表是一种较为复杂的线性结构，它们的逻辑特征为一个数据元素可能被递归地定义。

数组和广义表是一种复杂的非线性结构，它们的逻辑特征是：一个数据元素可能有多个直接前驱和多个直接后继。数组和广义表大量用于计算机领域，是一种重要的数据存储和表现形式。

## 5.1 数组的基本概念

### 5.1.1 数组的概念

**数组**(array)是由一组类型相同的数据元素构造而成的。它的每个元素由一个值和一组下标确定。

**1. 一维数组**

如果数组元素只含有一个下标，这样的数组称为一维数组。如果把数据元素的下标顺序换成线性表中的序号，则一维数组就是一个**线性表**。下面就是一个一维数组。

$$A=(a_1,a_2,\cdots,a_n)$$

**2. 二维数组**

如果数组的每一个元素都含有两个下标，这样的数组称为二维数组，也称为矩阵(matrix)。

如图5.1所示，$\mathbf{A}_{mn}$ 就是一个m行n列的矩阵，可以用一个二维数组来表示，它的每一个元素 $a_{ij}$ 由一组下标来确定，但是需要说明的是，在算法设计中，数组、列表等线性表的下标均从0开始，为了便于理解，图中均从1开始。这个二维数组还可以进一步表示成如下的列向量形式。

$$\mathbf{A}_{mn}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

图5.1 m×n阶矩阵

$$\mathbf{A}=\begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_m \end{bmatrix}$$

其中，$\mathbf{A}_i$ 为行向量，$\mathbf{A}_i=(a_{i1},a_{i2},\cdots,a_{in})(1\leqslant i\leqslant m)$。或者把图5.1表示成如下行向量形式：

$$\mathbf{A}=[\mathbf{A}_1 \quad \mathbf{A}_2 \quad \cdots \quad \mathbf{A}_n]$$

此时，$\mathbf{A}_j$ 为列向量，形式为

$$\mathbf{A}_j=\begin{bmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{bmatrix} \quad (1 \leqslant j \leqslant m)$$

经过这样处理，一个二维数组就蜕化成了一维数组，因而成为线性表，所以处理二维数组的基础是线性结构。

数组一旦被定义，它的维数就不再改变。在二维数组中进行元素的插入和删除操作将会移动大量的数据，因此运算的效率很低，一般不进行这类运算。数组通常只有两种基本运算：**存取给定位置上的数据元素**和**修改给定位置上的数据元素值**。

**3. 三维数组**

三维数组 $\mathbf{A}_{mnp}$ 可视为以二维数组为数据元素的向量。三维数组中的每个元素 $a_{ijk}$ 都属于三个向量。图 5.2 是一个三维数组的示意，可以帮助理解三维数组的空间结构。

其中，i(1≤i≤m)表示行，j(1≤j≤n)表示列，k(1≤k≤p)表示二维数组的个数。三维数组是由 p 个 m 行 n 列的二维数组组成的。

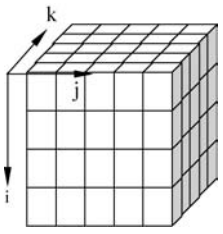

图 5.2　4×5×5 三维数组

## 5.1.2　数组的顺序存储结构

由于计算机的内存结构是一维的，用一维结构来表示多维数组，就必须按某种次序将数组元素排成一维线性序列，然后将这个线性序列存放在存储器中。

在数组操作中，一般不做插入和删除操作，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，一般采用顺序存储结构来表示数组。

数组的顺序存储结构指的是用一组连续的存储单元依次存放数组元素。一维数组的存储结构在 2.2 节中讨论过，下面介绍二维数组和三维数组的顺序存储结构。

要将二维数组的元素按顺序结构进行存储，就必须要按某种次序将元素排成一个线性序列。通常用行优先和列优先两种存储方式来存储二维数组的元素。

**1. 行优先顺序**

将数组元素按行向量排列，第 i+1 个行向量紧接在第 i 个行向量后面。在 PASCAL、C 语言中，数组就是按行优先顺序存储的。此时，图 5.1 中的矩阵元素按下列方式存储：

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

**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。在 FORTRAN 语言中，数组就是按列优先顺序存储的。此时，图 5.1 中的元素按下列方式存储：

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

不论是行优先还是列优先，都是把二维数组元素依次映射到一维内存空间上，若用 k 表示一维内存空间的地址(1≤k≤m×n)，很容易得到二维数组元素 $a_{ij}$ 与 k 的对应关系。设二维数组 $\mathbf{A}_{mn}$ 按行优先顺序存储在内存中，假设每个元素占 d 个存储单元，则 $a_{ij}$ 的地址计算函数为

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

对于Python语言来说,数组**A**[m][n]的两个下标的下界均为0,上界分别为m-1、n-1,每个数据元素占d个存储单元,二维数组中任一元素a[i][j]的存储位置可由下列公式确定。

$$LOC(a[i][j]) = LOC(a[0][0]) + (n \times i + j) \times d$$

其中,LOC(a[0][0])是$a_{00}$的存储位置,它是该二维数组的起始地址,在内存中地址为1。LOC(a[i][j])是$a_{ij}$的存储位置,在内存中地址为k($1 \leqslant k \leqslant m \times n$)。这个式子确定了Python语言的二维数组元素的位置和下标的关系。

**3. 数组元素的地址计算公式**

1) 二维数组

设m×n的二维数组行下标为1..m,下界为1,上界为m;列下标为1..n,下界为1,上界为n。

(1) 按行优先顺序存储的二维数组$A_{mn}$地址计算公式。

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

(2) 按列优先顺序存储的二维数组$A_{mn}$地址计算公式。

$$LOC(a_{ij}) = LOC(a_{11}) + [(j-1) \times m + i - 1] \times d$$

2) 三维数组

设m×n×p的二维数组行下标为1..m,下界为1,上界为m;列下标为1..n,下界为1,上界为n;第三维(高)下标为1..p,下界为1,上界为p。

(1) 按行优先顺序存储的三维数组$A_{mnp}$地址计算公式。

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

(2) 按列优先顺序存储的二维数组$A_{mnp}$地址计算公式。

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

### 5.1.3 特殊矩阵的压缩存储

在科学与工程计算问题中,矩阵是一种常用的数学对象,在用高级语言编程时,简单而又自然的方法就是将一个矩阵描述为一个二维数组。矩阵在这种存储表示之下,可以对其元素进行随机存取,各种矩阵运算也非常简单。但是在矩阵中非零元素呈某种规律分布或者矩阵中出现大量的零元素的情况下,用了许多单元去存储重复的非零元素或零元素,这对于高阶矩阵的存储会造成极大的空间浪费,为了节省存储空间,可以对这类矩阵进行压缩存储,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。

**1. 几种特殊的矩阵**

1) 对称矩阵

在一个n阶(n行n列)方阵**A**中,若元素满足下述性质:

$$a_{ij} = a_{ji} \quad (0 \leqslant i,j \leqslant n-1)$$

则称**A**为对称矩阵。如图5.3所示,对称矩阵沿主对角线折叠时,元素相同。

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。上三角矩阵的下三角(不包括主对角线)中的元素均为0。下三角矩阵正好相反,它的主对角线上方均为0。图5.4给出了上

三角和下三角矩阵的图示。

$$\begin{bmatrix} 1 & 3 & -1 & 0 & 0 \\ 3 & 5 & 0 & 2 & 1 \\ -1 & 0 & 0 & 6 & 8 \\ 0 & 2 & 6 & 7 & 0 \\ 0 & 1 & 8 & 0 & 2 \end{bmatrix}$$

图 5.3　对称矩阵

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \qquad \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

(a) 上三角矩阵　　(b) 下三角矩阵

图 5.4　三角矩阵

3）稀疏矩阵

设矩阵 $\mathbf{A}_{mn}$ 中有 s 个非零元素，若 $s \ll m \times n$，则称 **A** 为稀疏矩阵(sparse matrix)。图 5.5 给出了一个稀疏矩阵，它的非零元素的个数为 8，而矩阵的大小为 36。

稀疏矩阵中由于非零元素的个数远远小于矩阵元素的个数，若存储时仍然采用二维数组来存储，对于高阶矩阵而言，则将造成巨大的存储空间浪费，其空间效率低下，因而需要寻找其他存储结构来存储稀疏矩阵的元素。

4）三对角矩阵

非零元素沿矩阵的主对角线按照图 5.6 分布的称为三对角矩阵。

$$\mathbf{M} = \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & 15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

图 5.5　稀疏矩阵

$$\begin{bmatrix} * & * & & & & 0 \\ * & * & * & & & \\ & * & * & * & & \\ & & * & * & * & \\ 0 & & & * & * & \end{bmatrix}$$

图 5.6　三对角矩阵

**2. 特殊矩阵的存储压缩**

对于顺序表示数组的方法，当数组具有完整的矩阵结构时(即多维数组元素 $a_{i_1 i_2 \cdots i_n}$ 的各下标在各自独立的范围 $1 \leqslant i_1 \leqslant d_1$，$1 \leqslant i_2 \leqslant d_2$，…，$1 \leqslant i_n \leqslant d_n$ 内改变时，大部分元素值不为零)，一般是很适合的。但对于上面提到的几种情况，仍然采用数组来存储矩阵元素就会导致大量的存储空间浪费。为了节省存储空间，可以对这些矩阵进行压缩存储。

这些特殊矩阵的共同特点是非零元素的分布很有规律，从而可将其压缩到一维数组中，并能找到每个非零元素在一维数组中的对应位置。

对于 n 阶上三角形和下三角形矩阵，按以行序为主序的原则将矩阵的所有非零元素压缩存储到一个一维数组 M[1.. n(n+1)/2]中，则 M[k]和矩阵非零元素 $a_{ij}$ 之间存在一一对应的关系。

上三角形矩阵：$k=(2n-i+1)\times(i-1)/2+(j-i+1)$　　$(i \leqslant j)$

下三角形矩阵：$k=i\times(i-1)/2+j$　　$(i \geqslant j)$

对于三对角矩阵，按以行序为主序的原则将矩阵的所有非零元素压缩存储到一个一维数组 M[1..n-2]中，则 M[k]和矩阵中非零元素 $a_{ij}$ 之间存在一一对应的关系：

$$k = 2 \times i + j - 2$$

下面给出了三对角矩阵的压缩存储的实例。设有矩阵如图 5.7 所示。

$$\begin{matrix} a_{11} & a_{12} & \cdots & & & \\ a_{21} & a_{22} & a_{23} & \cdots & & \\ & a_{32} & a_{33} & \cdots & & \\ & & a_{43} & \cdots & & \\ & & & \cdots & a_{(n-1)n} & \\ & & & & a_{n(n-1)} & a_{nn} \end{matrix}$$

图 5.7　三对角矩阵

则压缩存储如图 5.8 所示。

| | $a_{11}$ | $a_{12}$ | $a_{21}$ | $a_{22}$ | $a_{23}$ | … | $a_{ij}$ | … | $a_{nn}$ |
|---|---|---|---|---|---|---|---|---|---|
| k | 1 | 2 | 3 | 4 | | … | 2i+j−2 | | 3n−2 |

图 5.8　三对角矩阵的压缩存储

## 5.2　稀 疏 矩 阵

在实际应用中,往往会遇到稀疏矩阵。式(5.1)所示的矩阵 **M** 和它的转置矩阵 **N**(见式(5.2))在 36 个元素中只有 6 个非零元素,按稀疏矩阵处理。

$$\mathbf{M}=\begin{bmatrix} 15 & 0 & 0 & 0 & 0 & 15 \\ 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \tag{5.1}$$

$$\mathbf{N}=\begin{bmatrix} 15 & 0 & 0 & 0 & 91 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 & 0 & 28 \\ 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.2}$$

按照压缩存储概念,只需存储稀疏矩阵的非零元素。但是,为了实现矩阵的各种运算,除了存储非零元素的值外,还必须同时记下它所在的行和列。这样一个三元组$(i,j,a_{ij})$便能唯一地确定矩阵中的一个非零元素,其中,i,j 分别表示非零元素的行号和列号,$a_{ij}$ 表示第 i 行,第 j 列的非零元素的值。下列 6 个三元组表示了式(5.1)中矩阵 **M** 的 6 个非零元素:

(1,1,15)(1,6,15) (2,3,3) (3,4,6) (5,1,91) (6,3,28)

其中,第一个数表示行号 i,第二个数表示列号 j,第三个数表示在第 i 行第 j 列的非零元素值 $a_{ij}$,这样就构成了所谓的三元组$(i,j,a_{ij})$,并由此三元组唯一确定稀疏矩阵的存储结构。

若以某种方式(以行为主或以列为主的顺序)将 6 个三元组排列起来,再加上一个表示矩阵 **M** 的行数、列数及非零元素的个数的特殊的三元组(6,6,6),则所形成的表就能唯一地确定稀疏矩阵。稀疏矩阵的压缩存储使矩阵的随机存取变得困难。

稀疏矩阵进行压缩存储通常有两类方法:顺序存储和链式存储。这里只讨论顺序存储。

将表示稀疏矩阵的非零元素的三元组按行优先(或列优先)的顺序排列(跳过零元素),并依次存放在向量中,这种稀疏矩阵的顺序存储结构称为三元组表。对于式(5.1),按照行优先的顺序排列,它的三元组表如图 5.9 所示。

| i | j | v |
|---|---|---|
| 1 | 1 | 15 |
| 1 | 6 | 15 |
| 2 | 3 | 3 |
| 3 | 4 | 6 |
| 5 | 1 | 91 |
| 6 | 3 | 28 |

图 5.9　三元组表

以下讨论中,均假定三元组是按行优先顺序排列的。

**1. 类型定义**

为了运算方便,将矩阵的总行数、总列数及非零元素的总数均作为三元组表的属性进行描述。其类型描述为:

**定义 5.1**　三元组的结点类型

```
class Triple(object):
    #初始化三元组
    def __init__(self):
        self.listrow = []                    #三元组的行,由外部获取
        self.listcol = []                    #三元组的列,由外部获取
        self.liste = []                      #三元组的值,由外部获取
```

**2. 转置运算**

一个 m×n 的矩阵 **A**,它的转置矩阵 **B** 是一个 n×m 的矩阵,且

$$\mathbf{A}[i][j]=\mathbf{B}[j][i] \quad (0\leqslant i<m,0\leqslant j<n)$$

即 **A** 的行是 **B** 的列,**A** 的列是 **B** 的行。

1) 三元组表表示的矩阵转置的思想方法

(1) 根据矩阵 **A** 的行数、列数和非零元总数确定矩阵 **B** 的列数、行数和非零元总数。

(2) 当三元组表非空(矩阵 **A** 的非零元不为 0)时,将 **A** 的三元组表 a.data 转置为 **B** 的三元组表 b.data。

2) 三元组表实现矩阵转置

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。

按这种方法设计的算法,其基本思想是:对 **A** 中的每一列 col(0≤col≤a->n−1),通过从头至尾扫描三元组表 a.data,找出所有列号等于 col 的那些三元组,将它们的行号和列号互换后依次放入 b.data 中,即可得到 **B** 的按行优先的压缩存储表示。

根据上述方法得到三元组矩阵转置算法如算法 5.1 所示。

**算法 5.1**　三元组实现矩阵转置算法

```
def TransMatrix(self):
```

```
        TripleTable_A = Triple()
        Rows = len(self.listcol)
        Cols = len(self.listrow)
        Rows_A = Cols
        Cols_A = Rows
        if len(self.liste) == 0:
            print("当前三元组表为空表!")
            return
        else:
            print("转置后的三元组为: \ni   j   e\n{")
            for i in  range(Rows):
                TripleTable_A.listrow.append(self.listcol[i])
                TripleTable_A.listcol.append(self.listrow[i])
                TripleTable_A.liste.append(self.liste[i])
                TripleTable_A.PrintTriple()
            print("}")                              #在输出时、"{ }"可以不使用
```

该算法的时间主要耗费在原三元组表的遍历循环上。

若 **A** 的列数为 n,非零元素个数为 t,则执行时间复杂度为 O(n×t),即与 **A** 的列数和非零元素个数的乘积成正比。

通常用二维数组表示矩阵时,其转置算法的执行时间复杂度是 O(m×n),它正比于行数和列数的乘积。

由于非零元素个数一般远远大于行数,因此上述稀疏矩阵转置算法的时间复杂度大于通常的转置算法的时间复杂度。

## 5.3 数组的应用

下面介绍数组的应用——迷宫问题。

所谓迷宫问题,就是把一只老鼠放进一个无盖的大箱内,箱内设置若干隔板,使老鼠走动的方向受到阻碍,看其如何找到一条通道,走出大箱。

现用二维数组 maze[m][n]来模拟迷宫,数组元素为 0 表示此路可通,数组元素为 1 表示此路不通。不失一般性,设迷宫入口为 maze[1][1],出口为 maze[m][n],且 maze[1][1]=0,maze[m][n]=0。

现要求设计算法找一条从迷宫入口到迷宫出口的通道,如图 5.10 所示,maze[6][7]表示一个 6×7 的迷宫。

求解迷宫问题的基本思想是将迷宫的入点(1,1)作为第一个出发点,向四周搜索可通行的位置,形成第一层新的出发点,然后对第一层中各个位置再分别向四周搜索可通行的位置,形成第二层新的出发点,…,如此进行下去直至到达迷宫的出口点(m,n)为止。

Maze[6][7]

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |

图 5.10　6×7 的迷宫

为了避免多次检测是否走到边缘,将迷宫四周各镶上一条取值均为 1 的边,相当于在迷宫周围加上一圈不通过的墙。由此,表示迷宫的二维数组应为 maze[m+2][n+2],如图 5.11 所示。由此,表示迷宫的二维数组应为 maze[8][9]。这样,在迷宫任一位置(x,y)(1≤x≤8,1≤y≤9)上都有 4 个

可以搜索的方位，设当前的坐标位置为(x,y)，则递归往左表示坐标为(x－1,y)，往右表示坐标为(x＋1,y)，往上表示坐标为(x,y＋1)，往下表示坐标为(x,y－1)。

为使问题简化，使用递归方法来解决。其算法的基本思想是将迷宫的入点(6,7)作为第一个出发点，向四周搜索可通行的位置，形成第一层新的出发点，然后对第一层中各个位置再分别向四周搜索可通行的位置，形成第二层新的出发点，…，如此进行下去直至到达迷宫的出口点(1,1)为止。

Maze[8][9]

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

图 5.11　图 5.9 外圈加 1 后的迷宫

如果有路可走，则设置 maze[x][y]＝0；如果是不可走的路，则设置 maze[x][y]＝1；如果是已经走过的路，设置 maze[x][y]＝2。

**算法 5.2**　迷宫初始化

```
#应用递归求迷宫问题; 数字 0 表示是可走的路; 数字 1 表示是墙壁,不可走的路
#数字 2 表示是走过的路
from numpy import *                          #导入 Numpy 库,以便进行矩阵相关运算
maze = [  [1, 1, 1, 1, 1, 1, 1, 1, 1,],      #初始化迷宫的数组
          [1, 0, 1, 0, 1, 0, 0, 0, 1,],
          [1, 0, 1, 0, 1, 0, 1, 1, 1,],
          [1, 0, 1, 0, 1, 1, 1, 1, 1,],
          [1, 0, 0, 0, 1, 0, 0, 0, 1,],
          [1, 1, 1, 0, 0, 0, 1, 0, 1,],
          [1, 0, 1, 1, 1, 1, 1, 0, 1,],
          [1, 1, 1, 1, 1, 1, 1, 1, 1 ]]
```

**算法 5.3**　走迷宫的递归函数

```
#走迷宫的递归函数
def find_path(x,y):
    if x == 1 and y == 1:
       maze[x][y] = 2
       return 1
    else:
        if maze[x][y] == 0:
            maze[x][y] = 2
            if (find_path(x - 1,y) + find_path(x + 1,y)
                 + find_path(x,y - 1) + find_path(x,y + 1)> 0):
                return 1
            else:
                maze[x][y] = 0
                return 0
        else:
            return 0
```

**算法 5.4**　主函数

```
#主函数: 用递归的方法在数组迷宫找出口
print("当前的迷宫构造为: ",mat(maze))          #利用矩阵的形式将迷宫构造输出
```

```
x = int(input("请输入迷宫入口横坐标:"))
y = int(input("请输入迷宫入口纵坐标:"))
print("迷宫的路径如下: ")
find_path(x,y)
for i in range(9):                          #迷宫的总列数
    for j in range(8):                      #迷宫的总行数
        if maze[i][j] == 2:                 #迷宫可以通行
            maze[i][j] = 'Y'
        else:                               #迷宫不可以通行
            maze[i][j] = 'N'
print(mat(maze))                            #利用矩阵的形式将迷宫路径输出
```

## 5.4 广 义 表

广义表是线性表的推广,它是递归的数据结构。

### 5.4.1 广义表的定义

广义表是 n(n≥0)个元素的有限序列,记作

$$A=(a_1,a_2,\cdots,a_n)$$

其中,A 是广义表的名称,n 是它的长度,$a_i$(1≤i≤n)或者是单个数据元素,或者是一个广义表。显然,广义表的定义是一个递归的定义,广义表中可以包含广义表。按照惯例,用英文大写字母表示广义表的名称,用小写字母表示数据元素。广义表 A 中的某个元素 $a_i$ 是一个数据元素时,称其为 A 的一个原子;当其不是一个数据元素时,则称其为广义表 A 的子表。

当广义表 A 非空时,称第一个元素 $a_1$ 为 A 的**表头**(head),称其余元素组成的表($a_2$,$a_3$,…,$a_n$)为 A 的**表尾**(tail)。

**例 5.1** 广义表示例

(1) A=(),A 是一个空表,其长度为零。

(2) B=(e),列表 B 只有一个原子 e,B 的长度为 1。

(3) C=(a,(b,c,d)),列表 C 的长度为 2,两个元素分别为原子 a 和子表(b,c,d)。

(4) D=(A,B,C),列表 D 的长度为 3,3 个元素都是列表。

(5) E=(a,E),这是一个递归的表,其长度为 2,E 表相当于一个无穷表。

从以上例子可以看出,广义表可以共享子表,且允许递归。另外,广义表的元素之间除了存在次序关系外,还存在层次关系。广义表中元素的最大层次为表的深度。元素的层次就是包含该元素的括号对的数目。例如:

$$F=(a,b,(c,(d)))$$

其中,数据元素 a、b 在第一层,数据元素 c 在第二层,数据元素 d 在第三层。广义表 F 的深度为 3。

根据对表头和表尾的定义可知,任何一个非空列表其表头可能是原子,也可能是列表,而其表尾必定为列表。例如:

$$Head(D)=A \quad Tail(D)=(B,C)$$
$$Head(C)=a \quad Tail(C)=((b,c,d))$$

为了简单起见，把每个表的名称(若有的话)写在该表的前面，这也是一种表示广义表的方法。对于上面例子中的表，可以相应地表示为：

A()；B(e)；C(a,(b,c,d))；D(A,B,C)；E(a,E)；F(a,b,(c,(d)))

这里规定：用圆圈和方框分别表示单元素和子表元素，并用线段把表和其元素(放在表结点的下方)连接起来，则可以得到广义表的图形表示，如图 5.12 所示。

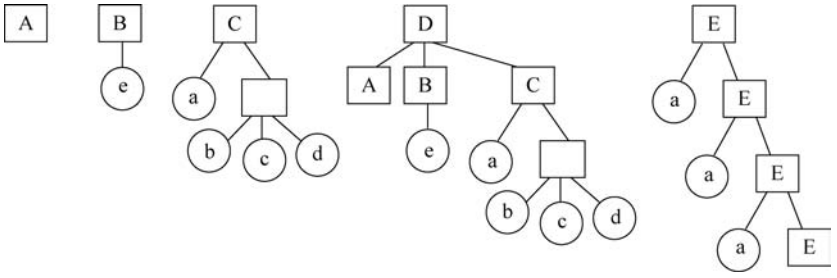


图 5.12　广义表图形表示

从广义表的图形可以看出，广义表的图形表示像倒着画的一棵树，树根结点代表整个广义表，各层树枝结点代表相应的子表，树叶结点代表单元素。

一个广义表中括号嵌套的最大层数称为广义表的深度。在图形表示中，是指从树根结点开始到每个树枝结点(表结点，而非树叶结点(单元素))所经过的结点个数的最大值。

## 5.4.2　广义表的存储结构

由于广义表($a_1$,$a_2$,…,$a_n$)中的数据元素可以具有不同的结构(或是原子，或是列表)，因此难以用顺序存储结构表示，通常采用链式存储结构，每个数据元素可用一个结点表示。

如何设定结点的结构？由于列表中的数据元素可能为原子或子表，由此需要两种结构的结点；一种是表结点，用以表示子表；另一种是原子结点，用以表示原子(单元素)。

对于单元素原子结点来说，应包括值域和指向其后继结点的指针域。用 tag＝0 表示单元素结点，如图 5.13 所示。

对于子表结点来说，应包括指向子表中第一个结点的表头指针域和其后继结点的指针域。用 tag＝1 表示子表结点，用 sublist 表示子表的第一结点的表头指针，如图 5.14 所示。

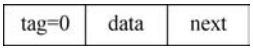


图 5.13　单元素结点结构

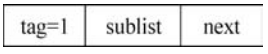


图 5.14　子表结点结构

为了方便这样的存储，须将广义表进行转换，转换的目标是一条链。转换时保留父结点与左孩子的连线，抹去所有父结点与右孩子的连线；加上左孩子与其同父右孩子的连线，若有多个孩子，则依次连接，如图 5.15 所示。

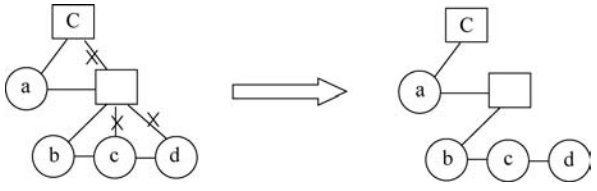


图 5.15　广义表的转换

根据上面的分析可知,单元素结点和子表结点共用一个域。如果是单元素结点,则是数据域;如果是子表结点,则是子表指针域,如图 5.16 所示。


图 5.16　表结点与元素结点的结构

中间部分采用 Python 的联合体。于是,得出广义表的结构类型定义如下。

**定义 5.2**　广义表的结构类型

```
import copy
class GLNode(object):
    def __init__(self):
        self.tag = None                    #标志域
        self.union = None                  #联合域
        self.next = None
```

在这种存储结构中有如下几种情况:①除空表的表头指针为空外,对任何非空广义表,其表头指针均指向一个表结点。②容易分清列表中原子和子表所在层次。如在列表 D 中,原子 a 和 e 在同一层次上,而 b、c 和 d 在同一层次且比 a 和 e 低一层,A、B 和 C 是同一层的子表。③最高层的表结点个数即为广义表的长度。图 5.17 和图 5.18 给出了例 5.1 中广义表的存储结构示意。

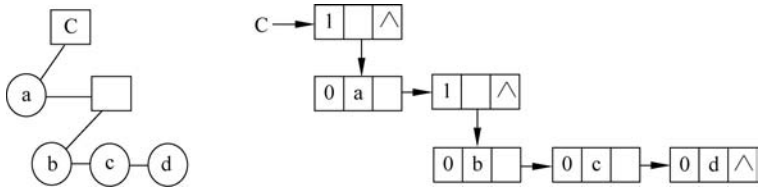


图 5.17　例 5.1 中广义表存储结构示例图

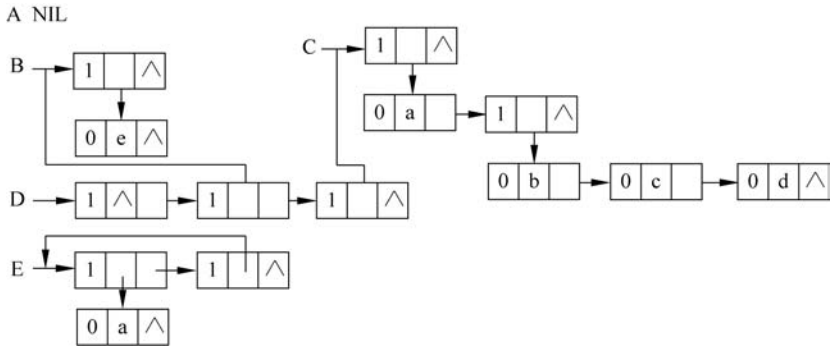


图 5.18　例 5.1 中广义表存储结构

在图 5.18 广义表的表示中,D 表使用了 B、C 表。

## 5.4.3　广义表的运算

广义表的运算主要有求广义表的长度、深度、向广义表插入元素、从广义表中查找元素、

删除元素和输出广义表等。

**1. 求广义表的长度**

在广义表中，同一层次的每一个结点是通过 next 域连接起来的，所以可把它看作是由 next 域链接起来的单链表。这样，求广义表的长度就是求单链表的长度。用以前的方法计算即可。

在进行广义表操作时，首先找到表头，然后利用表头进行操作。

**算法 5.5** 获取广义表表头的算法

```
def GetGListHead(self,GList):
    if GList!= None and GList.union!= None:
        head = copy.deepcopy(GList.union)
        head.next = None
        return head
    else:
        print("无法获取表头!")
```

**算法 5.6** 求广义表长度的递归算法

```
#求广义表长度的递归算法
def Glist_Length(self):
    cNode = self.head
    if cNode!= None:
        return Glist_Length(cNode.next) + 1
    else:
        return 0
```

**算法 5.7** 求广义表长度的非递归算法

```
def Get_Length(self):
    cNode = self.head
    cNode = cNode.next
    count = 0
    while cNode.data!= None:                    #广义表的结点值不为空,就记录一个值
        count = count + 1
        cNode = cNode.next
    return count
```

**2. 求广义表的深度**

广义表的深度定义为广义表中括弧的重数，是广义表的一种量度。

设非空广义表为

$$GL=(a_1,a_2,\cdots,a_n)$$

其中，$a_i(i=1,2,\cdots,n)$为原子或为 GL 的子表，则求 GL 的深度可分解为 n 个子问题，每个子问题为 $a_i$ 的深度，若 $a_i$ 是原子，则由定义知其深度为 0；若 $a_i$ 是广义表，则和上述一样处理，而 GL 的深度为各 $a_i(i=1,2,\cdots,n)$的深度中最大值加 1。空表也是广义表，并由定义可知空表的深度为 1。

由此可见，求广义表的深度是一个递归算法，它有两个终结状态：空表和原子，且只要

求得 $a_i$(i=1,2,…,n)的深度,广义表的深度就容易求得了。显然,它应比子表的最大值多 1。

深度 DEPTH(GL)的递归定义如下。

基本项:DEPTH(GL)=0(当 GL 为原子时)

DEPTH(GL)=1(当 GL 为空表时)

归纳项:DEPTH(GL)=1+Max{DEPTH($a_i$)} (n≥1)

由此定义容易写出求深度的递归函数,其深度的算法如下。

**算法 5.8** 求广义表深度的递归算法

```
def GL_Depth(GL):
    Max = 0                                  #Max 为同一层所求过的子表中深度的最大值
    if GL.tag == 0:                          #原子,深度为 0
            return 0
    if GL.data == None:                      #空表,深度为 1
            return 1
    while GL.next!= None:                    #遍历表中的每一个元素
            if GL.tag == 1:                  #表中元素为子表
                dep = GL_Depth(GL)           #递归求表长
                if dep > Max:
                    Max = dep
            GL = GL.next                     #移向下一个元素
    return Max + 1                           #返回表长
```

**例 5.2** 求广义表 D=(A,B,C)=((),(e),(a,(b,c,d)))的深度。

**解**:DEPTH(D)=1+Max{DEPTH(A),DEPTH(B),DEPTH(C)}

DEPTH(A)=1

DEPTH(B)=1+Max{DEPTH(e)}=1+0=1

DEPTH(C)=1+Max{DEPTH(a),DEPTH((b,c,d))}=2

DEPTH(a)=0

DEPTH((b,c,d))=1+Max{DEPTH(b),DEPTH(c),DEPTH(d)}

=1+0=1

由此,DEPTH(D)=1+Max{1,1,2}=3。

**3. 建立广义表的存储结构**

假设把广义表的书写形式看成是一个字符串 S,每个原子的值被限定为英文字母。并假定广义表是一个表达式,其格式为:元素之间用一个逗号分隔,子表元素的起止符号分别为左、右括号,空表在其括号内不包含任何字符。例如,(a,(b,c,d))就是一个符合上述规定的广义表达式(如果广义表用图形表示,则不难将它转换为广义表的表达式格式)。

建立广义表存储结构的算法同样是一个递归算法。该算法使用一个具有广义表格式的字符串参数 s,返回由它生成的广义表存储结构的头结点指针 h。在算法的执行过程中,需要从头到尾扫描 s 的每个字符。当碰到左括号时,表明它是一个表元素的开始,则应建立一个由 h 指向的表结点,并用它的 sublist 域作为子表的表头指针进行递归调用来建立子表的存储结构;当碰到一个英文字母时,表明它是一个空表,则应置 h 为空;当建立了一个由 h 指向的结点后,接着碰到逗号字符时,表明存在后继结点,需要建立当前结点(即由 h 指向的

结点)的后继表；当碰到右括号时，表明当前所处理的表已结束，应该置当前结点的 next 域为空。

**4. 广义表的输出**

以 h 作为带表头附加结点的广义表的表头指针，打印输出该广义表时，需要对子表进行递归调用。当 h 结点为表元素结点时，首先输出作为一个表的起始符号的左括号，然后再输出以 h.sublist 为表头指针的表；当 h 结点为单元素结点时，则应输出该元素的值。当以 h.sublist 为表头指针的表输出完毕后，应在其最后输出一个作为表终止符的右括号。当 h 结点输出结束后，若存在后继结点，则应首先输出一个逗号作为分隔符，然后再递归输出由 h.next 指针所指向的后继表。即广义表的输出也是一个递归过程。设需输出的广义表为 LS，广义表输出操作的递归定义如下。

基本项：当 GL 指向原子元素时，输出原子元素。

　　当 GL 为空表时，输出一对空的括号。

归纳项：当 GL 指向列表时，先输出表头，后输出表尾。

综上所述，广义表输出算法如下。

**算法 5.9**　广义表的输出算法

```
#遍历广义表的函数
def PrintGList(self,GList):
    if GList!= None:
        if GList.tag == 0:
            print(GList.union,end = '')
        else:
            print('(',end = '')
            if GList.union == None:
                print('#',end = '')
            else:
                self.PrintGList(GList.union)
            print(')',end = '')
        if GList.next!= None:
            print(',',end = '')
            self.PrintGList(GList.next)
```

算法 5.9 的时间复杂度与建立广义表存储结构的情况相同，均为 O(n)，n 为广义表中所有结点的个数。

## 本 章 小 结

本章讨论了二维和三维数组顺序存储结构的地址计算方法，不同的语言分别采用行优先和列优先顺序。对于各种特殊矩阵，为了节约存储空间，采用了压缩存储给出它们的地址计算。而对于稀疏矩阵，则采用了三元组形式来存储数据，并用三元组结构实现了矩阵的转置运算。

在广义表中主要讨论基本概念、存储结构及其运算。

# 习 题 5

## 一、单项选择题

1. 设有一个10阶的对称矩阵**A**,采用压缩存储方式,以行序为主存储,$a_{11}$为第一个元素,其存储地址为1,每个元素占一个地址空间,则$a_{85}$的地址为_____。

A. 13　　B. 33　　C. 18　　D. 40

2. 设有数组A[1:8,1:10],数组的每个元素长度为3字节,数组从内存首地址BA开始顺序存放,当用以列为主存放时,元素A[5,8]的存储首地址为_____。

A. BA+141　　B. BA+180

C. BA+222　　D. BA+225

3. 假设以行序为主序存储二维数组A[1..100,1..100],设每个数据元素占2个存储单元,基地址为10,则LOC[5,5]=_____。

A. 808　　B. 818　　C. 1010　　D. 1020

4. 将一个A[1..100,1..100]的三对角矩阵按行优先存入一维数组B[1..298]中,A中元素$A_{66,65}$(即该元素下标i=66,j=65),在B数组中的位置k为_____。

A. 198　　B. 195　　C. 197　　D. 196

5. 二维数组A的每个元素是由6个字符组成的串,其行下标i=0,1,2,…,8,列下标j=1,2,…,10。若A按行先存储,元素A[8,5]的起始地址与当A按列先存储时的元素_____的起始地址相同。设每个字符占一个字节。

A. A[8,5]　　B. A[3,10]　　C. A[5,8]　　D. A[0,9]

6. 若对n阶对称矩阵**A**以行序为主序方式将其下三角形的元素(包括主对角线上所有元素)依次存放于一维数组B[1..(n(n+1))/2]中,则在B中确定$a_{ij}$(i≥j)的位置k的关系为_____。

A. i*(i-1)/2+j　　B. j*(j-1)/2+i

C. i*(i+1)/2+j　　D. j*(j+1)/2+i

7. 设**A**是n×n的对称矩阵,将**A**的对角线及对角线上方的元素以列为主的次序存放在一维数组B[1..n(n+1)/2]中,对上述任一元素$a_{ij}$(1≤i,j≤n,且i≤j)在B中的位置为_____。

A. i(i-l)/2+j　　B. j(j-l)/2+i

C. j(j-l)/2+i-1　　D. i(i-l)/2+j-1

8. **A**[N,N]是对称矩阵,将下面三角(包括对角线)以行序存储到一维数组T[N(N+1)/2]中,则对任一上三角元素a[i][j]对应T[k]的下标k是_____。

A. i(i-1)/2+j　　B. j(j-1)/2+i

C. i(j-i)/2+1　　D. j(i-1)/2+1

9. 设二维数组A[1.. m,1.. n](即m行n列)按行存储在数组B[1..m×n]中,则二维数组元素A[i,j]在一维数组B中的下标为_____。

A. (i-1)n+j　　B. (i-1)n+j-1

C. i(j-1)　　D. jm+i-1

10. 有一个 100×90 的稀疏矩阵，非 0 元素有 10 个，设每个整型数占 2 字节，则用三元组表示该矩阵时，所需的字节数是_____。

A. 60　　B. 66　　C. 18 000　　D. 33

11. 数组 A[0..4，－1..－3，5..7]中含有元素的个数为_____。

A. 55　　B. 45　　C. 36　　D. 16

12. 广义表 L＝(a，(b，c))，进行 Tail(L)操作后的结果为_____。

A. c　　B. b，c　　C. (b，c)　　D. ((b，c))

13. 广义表((a，b，c，d))的表头是_____，表尾是_____。

A. a　　B. ()　　C. (a，b，c，d)　　D. (b，c，d)

**二、判断题(判断正确与错误，正确的打√，错误的打×)**

1. 数组不适合作为任何二叉树的存储结构。　(　　)

2. 从逻辑结构上看，n 维数组的每个元素均属于 n 个向量。　(　　)

3. 稀疏矩阵压缩存储后，必会失去随机存取功能。　(　　)

4. 数组是同类型值的集合。　(　　)

5. 数组可看成线性结构的一种推广，因此与线性表一样，可以对它进行插入、删除等操作。　(　　)

6. 一个稀疏矩阵 $\mathbf{A}_{m\times n}$ 采用三元组形式表示，若把三元组中有关行下标与列下标的值互换，并把 m 和 n 的值互换，则就完成了 $\mathbf{A}_{m\times n}$ 的转置运算。　(　　)

7. 二维以上的数组其实是一种特殊的广义表。　(　　)

8. 广义表的取表尾运算，其结果通常是个表，但有时也可是个单元素值。　(　　)

9. 若一个广义表的表头为空表，则此广义表也为空表。　(　　)

10. 广义表中的元素或者是一个不可分割的原子，或者是一个非空的广义表。　(　　)

**三、填空题(将正确答案填写到相应空格中)**

1. 数组的存储结构采用_____存储方式。

2. 设二维数组 A[－20..30，－30..20]，每个元素占有 4 个存储单元，存储起始地址为 200。如果按行优先顺序存储，则元素 A[25，18]的存储地址为_____；如果按列优先顺序存储，则元素 A[－18，－25]的存储地址为_____。

3. 设数组 a[1..50，1..80]的基地址为 2000，每个元素占 2 个存储单元，若以行序为主序顺序存储，则元素 a[45，68]的存储地址为_____；若以列序为主序顺序存储，则元素 a[45，68]的存储地址为_____。

4. 广义表的表尾是指除第一个元素之外，_____。

5. 广义表简称表，是由零个或多个原子或子表组成的有限序列。原子与表的差别仅在于_____。为了区分原子和表，一般用_____表示表，用_____表示原子。一个表的长度是指_____，而表的深度是指_____。

6. 广义表的_____定义为广义表中括号的重数。

7. 设广义表 L＝((),())，则 head(L)是_____；tail(L)是_____；L 的长度是_____；深度是_____。

8. 设某广义表 H＝(A，(a，b，c))，运用 head 函数和 tail 函数求出广义表 H 中某元素 b 的运算式_____。

9. 广义表 A(( ),(a,(b),c)),head(tail(head(tail(head(A))))等于_____。

10. 广义表运算式 HEAD(TAIL(((a,b,c),(x,y,z))))的结果是_____。

**四、简答题**

1. 数组 A[1..8,-2..6,0..6]以行为主序存储,设第一个元素的首地址是 78,每个元素的长度为 4,试求元素 A[4,2,3]的存储首地址。

2. 已知 n×n 阶 b 对角矩阵($a_{ij}$),以行主序将 b 条对角线上的非零元存储在一维数组中,每个数据元素占 L 个存储单元,存储基地址为 S,请用 i,j 表示出 $a_{ij}$ 的存储位置。

3. 假设按行优先存储整型数组 A(-3:8,3:5,-4:0,0:7)时,第一个元素的字节存储地址是 100,每个整数占 4 字节,问 A(0,4,-2,5)的存储地址是什么?

4. 设有三维数组 A[-2:4,0:3,-5:1]按列序存放,数组的起始地址为 1210,试求 A(1,3,-2)所在的地址。

5. 三维数组 A[1..10,-2..6,2..8]的每个元素的长度为 4 字节,试问该数组要占多少字节的存储空间? 如果数组元素以行优先的顺序存储,设第一个元素的首地址是 100,试求元素 A[5,0,7] 的存储首地址。

6. 画出下列广义表的两种存储结构图:((),A,(B,(C,D)),(E,F))。

**五、算法分析**

1. 设整数 $x_1,x_2,\cdots,x_N$ 已存放在数组 A 中,编写 Python 递归过程,输出从这 n 个数中取出所有 k(k≤n)个数的所有组合。例如,若 A 中存放的数是 1,2,3,4,5,k 为 3,则输出结果应为 543,542,541,532,531,521,432,431,421,321。

2. 设计一个算法,根据随机数建立一个稀疏矩阵 **A**,并利用三元组存储形式存储其中的非零元素,然后将 **A** 转置为 **B**,再将 **B** 采用三元组形式存储。

3. 广义表具有共享性,因此在访问时考虑是否可以在表中添加一个域用来标记结点是否被访问过。根据此思路,设计一个算法,要求将广义表中所有值为 X 的原子替换为值为 Y。并思考,这样的广义表是不是可以运用到其他数据结构中。

# 实　　训

**一、实训目标**

1. 掌握计算顺序数组的元素地址。
2. 掌握稀疏矩阵的运算。
3. 掌握广义表的运算。

**二、实训要求**

1. 掌握二维及其以上矩阵的存储地址计算技术。
2. 编程实现稀疏矩阵的算法。
3. 计算迷宫问题。
4. 广义表计算与算法。

**三、实训内容**

1. 三元组表算法。
2. 稀疏矩阵十字链存储及其算法。

3. 迷宫算法。

4. 广义表运算。

**四、实训案例：用三元组转置矩阵**

(1) 定义类。

```
import numpy
from numpy import *
#定义并初始化三元组表
class Three_tuple(object):
    #初始化三元组
    def __init__(self):
        self.listrow = []
        self.listcol = []
        self.liste = []
```

(2) 创建三元组。

```
def CreatThreeTuple(self):
    rows = input("请输入行号: ")
    if rows == "#":
        return
    cols = input("请输入列号: ")
    if cols == "#":
        return
    element = input("请输入结点值: ")
    if element == "#":
        return
    while 1:
        self.listrow.append(int(rows))
        self.listcol.append(int(cols))
        self.liste.append(int(element))
        rows = input("请输入行号: ")
        if rows == "#":
            break
        cols = input("请输入列号: ")
        if cols == "#":
            break
        element = input("请输入结点值: ")
        if element == "#":
            break
```

(3) 输出三元组。

```
def PrintThreeTuple(self):
    length = len(self.Liste)
    i = 0
    if length == 0:
        print("当前三元组表为空表!")
        return
    else:
        print("{")
```

```
        while i < length:
            r = self.listrow[i]
            c = self.listcol[i]
            print("( %d, %d, %d)" % (r,c,self.liste[i]))
            i = i + 1
        print("}")
```

(4) 获取三元组值。

```
def Get_TriTupleTable(self,x):
    length = len(self.liste)
    i = 0
    if length == 0:
        print("当前三元组表为空表!")
        return
    else:
        Rows = self.listcol[x]
        Cols = self.listrow[x]
        Element = self.liste[x]
        return Rows,Cols,Element
```

(5) 根据三元组创建稀疏矩阵。

```
    def Creat_matrix(self):
        Row = max(self.listrow)
        Col = max(self.listcol)
        print("请输入矩阵的行数(建议不低于",Row + 1,"行: )")
        rows = int(input())
        print("请输入矩阵的列数(建议不低于",Col + 1,"列: )")
        cols = int(input())
        print("初始化的矩阵为: ")
        matrix = [[0 for  i in range(rows)] for i in range(cols)]
        print("matrix = \n",mat(matrix))
        i = 0
        while i < len(self.Liste):
            (k,l,e) = self.Get_TriTupleTable(i)
            matrix[int(k)][int(l)] = int(e)
            i = i + 1
        return matrix
```

(6) 三元组矩阵转置。

```
    def TransTriTupleTable(self):
        TriTupleTable_A = Three_tuple()
        Rows = len(self.listcol)
        Cols = len(self.listrow)
        Rows_A = Cols
        Cols_A = Rows
        if len(self.liste) == 0:
            print("当前三元组表为空表!")
            return
```

```
        else:
            print("转置后的三元组为: \ni   j   e\n{")
            for i in  range(Rows):
                TriTupleTable_A.listrow.append(self.listcol[i])
                TriTupleTable_A.listcol.append(self.listrow[i])
                TriTupleTable_A.liste.append(self.liste[i])
            for i in range(Rows):
                self.listcol[i] = TriTupleTable_A.listcol[i]
                self.listrow[i] = TriTupleTable_A.listrow[i]
                self.liste[i] = TriTupleTable_A.liste[i]
            print("}")
```

(7) 转置后的三元组矩阵。

```
def Trans_Matrix(self,matrix):
    Num = numpy.array(matrix)
    rows = Num.shape[0]
    cols = Num.shape[1]
    Matrix = [[0 for  i in range(cols)] for i in range(rows)]
    i = 0
    self.TransTriTupleTable()
    while i < len(self.liste):
        (k,l,e) = self.Get_TriTupleTable(i)
        Matrix[int(k)][int(l)] = int(e)
        i = i + 1
    return Matrix
```

(8) 测试主程序。

```
#实例化三元组和调用三元组方法
ThT = Three_tuple()
ThT.CreatThreeTuple()
print("你创建的三元组为: \ni   j   e\n{")
ThT.PrintThreeTuple()
print("根据三元组创建的稀疏矩阵为: \n")
M = mat(ThT.Creat_matrix())
print(M)
print("转置后的稀疏矩阵为: \nMatrix = ")
print(mat(ThT.Trans_Matrix(M)))
```

(9) 运行结果。

① 初始化矩阵。

初始化的矩阵为：

```
matrix =
  [[0 0 0 0 0 0 0]
   [0 0 0 0 0 0 0]
   [0 0 0 0 0 0 0]
   [0 0 0 0 0 0 0]
   [0 0 0 0 0 0 0]
```

```
   [0 0 0 0 0 0 0 ]
   [0 0 0 0 0 0 0 ]]
```

② 三元组数据输入。

```
i  j  e
{
(0,0,1)
(6,6,36)
(2,5,10)
(3,4,12)
(5,2,36)
(3,0,12)
(2,3,6)
(5,0,14)
(2,0,33)
}
```

③ 根据三元组创建的矩阵。

```
[[ 1  0  14  3  0  0  35]
 [ 0  0   0 68  0  0   0]
 [ 0  0   0  0 38  0   0]
 [ 0  0   0  0  0  0   0]
 [ 0  0   0  0  0  0   0]
 [66  0   0  12 0 19   0]
 [ 0  0   0  0  0  0  36]]
```

④ 转置后的矩阵。

```
[[ 1  0  0  0  0 66  0]
 [ 0  0  0  0  0  0  0]
 [14  0  0  0  0  0  0]
 [ 3 68  0  0  0 12  0]
 [ 0  0 38  0  0  0  0]
 [ 0  0  0  0  0 19  0]
 [35  0  0  0  0  0 36]]
>>>
```