第5章 多维数组和广义表

本章要点

- ◇ 数组的类型定义和存储结构
- ◇ 特殊矩阵和稀疏矩阵压缩存储方法及运算的实现
- ◇ 广义表的概念和基本运算

本章学习目标

- ◇ 了解数组的存储表示方法，并掌握数组在以行为主的存储结构中的地址计算方法
- ◇ 掌握对特殊矩阵进行压缩存储时的下标变换公式
- ◇ 了解稀疏矩阵的两种压缩存储方法的特点和适用范围，领会以三元组表示稀疏矩阵时进行矩阵运算采用的处理方法
- ◇ 掌握广义表的概念和基本运算，掌握对非空广义表进行分解的分析方法

前面各章中介绍的线性表、栈、队列和串等都是线性结构，它们共同的逻辑特征是每个数据元素至多有一个直接前趋和直接后继。本章将要介绍的多维数组和广义表都是非线性结构，它们的逻辑特征是每个数据元素可能有多个直接前趋和多个直接后继。

5.1 多维数组

5.1.1 多维数组的定义

数组(array)是数据结构中常用的数据类型，程序设计语言一般都直接支持数组类型。数组一旦建立，其元素个数和元素之间的关系就确立了。

在讨论多维数组之前，先看一下一维数组的定义：一维数组是向量，在逻辑结构上是有序元素的有限集合，它的每个元素用一个整数标号(亦称下标)来标志。数组(向量)是存储于计算机的连续存储空间中的多个具有统一类型的数据元素。同一数组的不同元素通过不同的下标标识，如$(a_0, a_2, \cdots, a_{n-1})$。一维数组可以看作一个线性表。

二维数组$\boldsymbol{A}_{mn}$可看成由m个行向量组成的向量，或由n个列向量组成的向量。二维数组中的每个元素a_{ij}既属于第i行的行向量，又属于第j列的列向量。由此可见，对于二维数组，可以看成每个元素为一维数组的线性表，这个元素可以是行向量，也可以是列向量，如图5.1所示。

类似地，我们可以把三维数组看成每个元素都为二维数组的线性表。依此类推，可以把一个n维数组看成每个元素是$n-1$维数组的线性表。

$$\boldsymbol{A}_{mn}=\overbrace{\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}}^{n\text{个列向量}}\left.\vphantom{\begin{bmatrix} a \\ a \\ \vdots \\ a \end{bmatrix}}\right\}m\text{ 个行向量}$$

图 5.1　二维数组的向量表示

数组是由高级语言直接给出的。通常情况下,数组只有两种基本运算:读和写。

(1) 读:给定一组下标,读出相应的元素。

(2) 写:给定一组下标,修改相应的元素。

5.1.2　数组的存储结构

在高级语言中已经实现了数组这种数据结构,数组的类型定义是由高级语言中的数组类型直接给出的。一般取数组的开始地址作为基准(基址),然后考察其他元素相对此基址的偏移量(偏移地址)。因此,基址加上该数组元素的偏移量就是该元素的绝对地址。

数组是数据元素的线性组合,对于一维数组而言,其实质就是线性表。它的数据元素可以通过下标 (index) 直接访问。设一维数组为 $A=(a_0,a_2,\cdots,a_{n-1})$,第 1 个元素下标值为 0,设其存储地址为 Loc(0),且每个元素占用的存储单元数为 L,则元素 a_i 的地址为:

$$\mathrm{Loc}(i)=\mathrm{Loc}(0)+i\times L \tag{5.1}$$

由于内存是一段连续的一维存储空间,并不是一个多维的存储空间,因此多维数组中的数据要存储于内存中,需要按照一定的顺序存储在一维空间中,这也是 C 语言处理多维数组的方法。下面介绍如何将多维数组转化成一维数组来表示。

对二维数组通常有两种映像方法,即"以行(序)为主(序)"的映像方法和"以列(序)为主(序)"的映像方法。"以行为主"的存储结构是对二维数组进行"按行切分",即将数组中的数据元素按行依次排放在存储器中;"以列为主"的存储结构是对二维数组进行"按列切分",即将数组中的数据元素按列依次排放在存储器中。

假设二维数组 $\boldsymbol{A}_{mn}$ 中每个数据元素占 L 个存储单位,Loc(i,j)表示下标为(i,j)的数据元素的存储地址,则该数组中下标为(i,j)对应的数据元素在"以行为主"的顺序映像中的存储地址为:

$$\mathrm{Loc}(i,j)=\mathrm{Loc}(0,0)+(i\times n+j)\ L \tag{5.2}$$

在"以列为主"的顺序映像中的存储地址为:

$$\mathrm{Loc}(i,j)=\mathrm{Loc}(0,0)+(j\times m+i)\ L \tag{5.3}$$

其中 Loc(0,0) 是二维数组的基址,即第 1 个数据元素下标为(0,0)的存储地址。

类似地,假设三维数组 $\boldsymbol{R}_{pmn}$ 中每个数据元素占 L 个存储地址,并以 Loc(i,j,k)表示下标为(i,j,k)的数据元素的存储地址,则该数组中任何一对下标为(i,j,k)的数据元素在"以行为主"的顺序映像中的存储地址为:

$$\mathrm{Loc}(i,j,k)=\mathrm{Loc}(0,0,0)+(i\times m\times n+j\times n+k)\times L \tag{5.4}$$

由此，可以推广到 N 维数组，公式的推导留待读者思考。

5.2 矩阵的压缩存储

矩阵是数值程序设计中经常用到的数学模型，它是由 m 行和 n 列的数值构成的（$m=n$ 时称为**方阵**）。在用高级语言编制的程序中，通常用二维数组表示矩阵，它使矩阵中的每个元素都可在二维数组中找到相对应的存储位置。然而在数值分析的计算中经常出现一些有下列特性的矩阵，即矩阵中有很多值相同的元素或零值元素，为了节省存储空间，需要对它们进行压缩存储，即不存储或少存储这些值相同的元素或零值元素，这称为**矩阵的压缩存储**。

5.2.1 特殊矩阵

如果矩阵中值相同的元素或零值元素在矩阵中的分布有一定的规律，则称为**特殊矩阵**。大致有以下三类特殊矩阵。

1. 对称矩阵

若 n 阶方阵 $\boldsymbol{A}_{nn}$ 中的元素满足特性：

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

则称为 n 阶**对称矩阵**。图 5.2 所示即为一个 5 阶对称矩阵。

对称矩阵中的元素关于主对角线对称，因此只要存储矩阵的上三角或下三角中的元素，让每两个对称的元素共享一个存储空间，就能节省近一半的存储空间。按行优先顺序存储主对角线（包括对角线）以下的元素，如图 5.3 所示。

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

图 5.2 对称矩阵示例

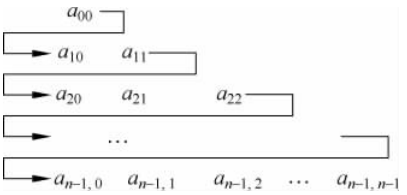

图 5.3 对称矩阵的行优先存放

按 $a_{00},a_{10},a_{11},\cdots,a_{n-1,0},a_{n-1,1},\cdots,a_{n-1,n-1}$ 次序存放在一个大小为 $n(n+1)/2$ 的向量 $\boldsymbol{S}_a$ 中（该下三角矩阵的元素总数为 $n(n+1)/2$）。其中：

$$\boldsymbol{S}_a[0]=a_{00}$$
$$\boldsymbol{S}_a[1]=a_{10}$$
$$\vdots$$
$$\boldsymbol{S}_a[n(n+1)/2-1]=a_{n-1,n-1}$$

再看下三角矩阵中元素 $a_{ij}(i\geqslant j)$ 在 $\boldsymbol{S}_a$ 中的存放位置，a_{ij} 元素前有 i 行（从第 0 行到第 $i-1$ 行），一共有 $1+2+\cdots+i=i(i+1)/2$ 个元素；在第 i 行上，a_{ij} 之前恰有 j 个元素（a_{i0}，$a_{i1},\cdots,a_{i,j-1}$），因此有 $\boldsymbol{S}_a[i(i+1)/2+j]=a_{ij}$。

由矩阵的对称性，可以得到在上三角矩阵中元素 $a_{ij}(i<j)$ 在 $\boldsymbol{S}_a$ 中的存放位置为 $\boldsymbol{S}_a[j(j+1)/2+i]=a_{ij}$。

现在考虑矩阵中一般元素 a_{ij} 和 $\boldsymbol{S}_a[k]$ 之间的对应关系：

若 $i \geqslant j$，则有 $\quad k = i(i+1)/2 + j \qquad 0 \leqslant k < n(n+1)/2$ (5.5a)

若 $i < j$，则有 $\quad k = j(j+1)/2 + i \qquad 0 \leqslant k < n(n+1)/2$ (5.5b)

令 $I = \max(i,j)$，$J = \min(i,j)$，则 k 和 i、j 的对应关系可统一为：

$$k = I(I+1)/2 + J \qquad 0 \leqslant k < n(n+1)/2 \tag{5.6}$$

通过下标变换公式，能立即找到矩阵元素 a_{ij} 在其压缩存储表示的向量 $\boldsymbol{S}_a$ 中的对应位置 k，因此这种压缩方法是随机存取结构。

例 5.1 a_{21} 和 a_{12} 均存储在 $\boldsymbol{S}_a[4]$ 中，这是因为由式(5.6)有：

$$k = I(I+1)/2 + J = 2 \times (2+1)/2 + 1 = 4$$

2. 三角矩阵

以主对角线划分，三角矩阵有上三角矩阵和下三角矩阵两种。上三角矩阵如图 5.4(a)所示，它的下三角(不包括主角线)中的元素均为常数 c。下三角矩阵与上三角矩阵相反，它的主对角线上方均为常数 c，如图 5.4(b)所示。

注意：在多数情况下，三角矩阵的常数 c 为零。

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

(a) 上三角矩阵　　(b) 下三角矩阵

图 5.4　三角矩阵

三角矩阵中的重复元素 c 可共享一个存储空间，其余的元素正好有 $n(n+1)/2$ 个，因此，三角矩阵可压缩存储到大小为 $n(n+1)/2+1$ 的向量 $\boldsymbol{S}_a$ 中，其中 c 存放在向量的最后一个分量中。

在上三角矩阵中，主对角线之上的第 p 行($0 \leqslant p < n$)恰有 $n-p$ 个元素，按行优先顺序存放上三角矩阵中的元素 a_{ij} 时，a_{ij} 元素前有 i 行(从第 0 行到第 $i-1$ 行)，元素个数为：

$$(n-0) + (n-1) + (n-2) + \cdots + (n-i+1) = i(2n-i+1)/2$$

在第 i 行上，a_{ij} 之前恰有 $j-i$ 个元素($a_{ii}, a_{i,i+1}, \cdots, a_{i,j-1}$)，因此有：

$$\boldsymbol{S}_a[i(2n-i+1)/2 + j - i] = a_{ij}$$

所以在上三角矩阵中 a_{ij} 和 $\boldsymbol{S}_a[k]$ 之间的对应关系为：

若 $i \leqslant j$，则有 $\quad k = i(2n-i+1)/2 + j - i \qquad 0 \leqslant k < n(n+1)/2 + 1$ (5.7a)

若 $i > j$，则有 $\quad k = n(n+1)/2$ (5.7b)

在下三角矩阵中元素 a_{ij} ($i \geqslant j$)前面有 i 行(从第 0 行到第 $i-1$ 行)，元素个数为：

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

在第 i 行上，a_{ij} 之前恰有 j 个元素($a_{i0}, a_{i1}, \cdots, a_{i,j-1}$)，因此有：

$$\boldsymbol{S}_a[i(i+1)/2 + j] = a_{ij}$$

所以下三角矩阵中 a_{ij} 和 $\boldsymbol{S}_a[k]$ 之间的对应关系为：

若 $i \geqslant j$，则有 $\quad k = i(i+1)/2 + j \qquad 0 \leqslant k < n(n+1)/2$ (5.8a)

若 $i<j$，则有 $$k=n(n+1)/2 \tag{5.8b}$$

3. 对角矩阵

所有的非零元素集中在以主对角线为中心的带状区域中，即除了主对角线和主对角线相邻两侧的若干条对角线上的元素之外，其余元素皆为零的矩阵为对角矩阵。图 5.5 所示即为对角矩阵，这里给出的是一个三对角矩阵。

$$\begin{bmatrix} a_{00} & a_{01} & & & & \\ a_{10} & a_{11} & a_{12} & & & \\ & a_{21} & a_{22} & a_{23} & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-2,n-3} & a_{n-2,n-2} & a_{n-2,n-1} \\ & & & & a_{n-1,n-2} & a_{n-1,n-1} \end{bmatrix}$$

图 5.5　三对角矩阵

非零元素集中在主对角线（a_{ii}，$0\leqslant i\leqslant n-1$）、紧邻主对角线上面的那条对角线（$a_{ij}$，$j=i+1$，$0\leqslant i\leqslant n-2$）和紧邻主对角线下面的那条对角线（$a_{ij}$，$j=i-1$，$1\leqslant i\leqslant n-1$）上。当 $|i-j|>1$ 时，元素 $a_{ij}=0$。

由此可知，一个 ***k* 对角矩阵**（k 为奇数）$\boldsymbol{A}$ 是满足下述条件的矩阵：若 $|i-j|>(k-1)/2$，则元素 $a_{ij}=0$。

对角矩阵可按行优先顺序或对角线的顺序，压缩存储到一个向量中，并且也能找到每个非零元素和向量下标的对应关系。

需要指出的是，上述的几种特殊矩阵，其非零元素的分布有规律可循，总能找到一种方法将它们压缩存储到一个向量中，并且能找到矩阵中的元素和该向量下标的对应关系，从而仍然能对矩阵元素进行随机存取。

5.2.2　稀疏矩阵

如果矩阵中只有少量的非零元素，并且这些非零元素在矩阵中的分布没有一定规律，则称为**随机稀疏矩阵**，简称为**稀疏矩阵**。至于矩阵中究竟含多少个零值元素才被称为稀疏矩阵，目前还没有一个确切的定义，它只是一个凭人的直觉来理解的概念。

如何存储稀疏矩阵中的非零元素呢？如果仍然采用二维数组表示稀疏矩阵，那么二维数组中存放了大量没有用的零值元素，并且在矩阵运算的时候进行了很多与零值元素相关的运算，这样既浪费了空间，又浪费了时间。

由此可知，稀疏矩阵压缩存储的目标是：

（1）尽可能减少或不存储零值元素。

（2）尽可能不做和零值元素相关的运算。

（3）便于进行矩阵运算，即易于根据一对行列号（i,j）找到矩阵中相应的元素，易于找到同一行或同一列的非零元素。

最简单的方法是将非零元素的值和它所在的行号、列号作为一个结点存放在一起，这样矩阵中的每一个非零元素就由一个三元组（行号，列号，元素值）唯一确定。很明显，稀疏矩

阵的压缩存储将失去随机存取的功能。所有非零元素对应的三元组构成的集合就是稀疏矩阵的逻辑表示,它有两种常用的存储方式:三元组表和十字链表。

将稀疏矩阵非零元素的三元组按行(或者列)的顺序排列,则得到一个结点均是三元组的线性表。该线性表的顺序存储结构称为稀疏矩阵的**三元组表**。因此,三元组表是稀疏矩阵的一种顺序存储结构。注意,在以下讨论中,均假定三元组是按行优先顺序排列的。

为了运算的方便,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:

```
#define MaxSize 10 000                /*非零元素个数的上限,三元组表的容量*/
typedef struct {
    int i,j;                          /*非零元素的行号、列号*/
    DataType v;                       /*非零元素的值*/
}TriTupleNode;                        /*三元组结点的类型*/
typedef struct{
    TriTupleNode data[MaxSize];       /*三元组表*/
    int m,n;                          /*矩阵的行数、列数*/
    int t;                            /*当前表长,即非零元素的个数*/
}TriTupleTable;                       /*稀疏矩阵类型*/
```

图5.6(a)所示的稀疏矩阵 $\boldsymbol{A}$ 的三元组表如图5.6(b)所示。

$$\boldsymbol{A}_{4\times5}=\begin{bmatrix}0 & 5 & 0 & 0 & 8\\ 1 & 0 & 3 & 0 & 0\\ 0 & -2 & 0 & 0 & 0\\ 6 & 0 & 0 & 0 & 0\end{bmatrix}$$

(a) 稀疏矩阵 $\boldsymbol{A}$

	i	j	v
0	0	1	5
1	0	4	8
2	1	0	1
⋮	1	2	3
	2	1	−2
a->t−1	3	0	6
⋮			
MaxSize−1			

(b) $\boldsymbol{A}$ 的三元组表 a -> data

图5.6 稀疏矩阵 $\boldsymbol{A}$ 和它的三元组表 $*a$

如果考虑到C语言中数组下标是从0开始的,而矩阵行号、列号一般是从1开始的,也可以从数组的1号单元开始存放非零元素,而0号单元正好用来存放数组的行数、列数和非零元素个数。这时,上述稀疏矩阵的类型定义以及后面的有关运算也需要略作修改。

稀疏矩阵三元组表的基本运算也是读 GET(i,j)和写 SET(i,j,x),其实现比较简单。只是在写时,三元组表中有可能没有对应元素(该元素原来的值为零),这时需要在三元组表中插入一个元素;若 $x=0$,则要在三元组表中删除对应的元素(如果存在的话),这是因为三元组表不存储零元素。有了读写运算,就可以像访问普通数组那样访问三元组表了。其他运算可以根据使用的需要进行设置,如矩阵的转置、加法、乘法等。下面主要讨论矩阵的转置运算在三元组表上的实现。

矩阵的转置是指它的行列互换。例如,有一个 $m\times n$ 的矩阵 $\boldsymbol{A}$,则它的转置矩阵 $\boldsymbol{B}$ 是一个 $n\times m$ 的矩阵,且 $\boldsymbol{A}[i][j]=\boldsymbol{B}[j][i]$($0\leqslant i<m$,$0\leqslant j<n$),即 $\boldsymbol{A}$ 的行是 $\boldsymbol{B}$ 的列,$\boldsymbol{A}$ 的列是 $\boldsymbol{B}$ 的行。图5.7(a)所示的矩阵 $\boldsymbol{B}$ 和图5.6(a)所示的矩阵 $\boldsymbol{A}$ 互为转置矩阵。

$$\boldsymbol{B}_{5\times4}=\begin{bmatrix}0&1&0&6\\5&0&-2&0\\0&3&0&0\\0&0&0&0\\8&0&0&0\end{bmatrix}$$

(a) 稀疏矩阵 $\boldsymbol{B}$

	i	j	v
0	0	1	1
1	0	3	6
2	1	0	5
⋮	1	2	−2
	2	1	3
$b->t-1$	4	0	8
⋮			
MaxSize−1			

(b) $\boldsymbol{B}$ 的三元组表 b -> data

图 5.7　稀疏矩阵 $\boldsymbol{B}$ 和它的三元组表 $*b$

利用读运算 GET 和写运算 SET，可以实现稀疏矩阵的转置：

```
for(i = 1;i <= m;i++)
for(j = 1;j <= n;j++)
     B.SET(j,i,A.GET(i,j));
```

这个算法虽然简单，但是效率不高。为了提高效率，可以不通过 GET 和 SET 而直接实现转置运算。

用三元组表表示的稀疏矩阵转置的步骤如下：

第一步：根据 $\boldsymbol{A}$ 矩阵的行数、列数和非零元素总数确定 $\boldsymbol{B}$ 矩阵的列数、行数和非零元素总数。

第二步：当三元组表非空（$\boldsymbol{A}$ 矩阵的非零元素不为 0）时，根据 $\boldsymbol{A}$ 矩阵三元组表的结点空间 data（以下简称为三元组表），将 $\boldsymbol{A}$ 的三元组表的 a -> data 置换为 $\boldsymbol{B}$ 的三元组表的 b -> data。

如果在转置中简单地将每个三元组的行号和列号互换，则转置后的三元组将不是按行序也不是按列序排列的，这时需要对三元组重新排序。为了降低时间复杂度，显然应该设法避免进行单独的排序运算，这就需要在进行元素行号和列号交换的过程中，顺便按行序排列。具体地说，一般有以下两种方法。

1）按列序转置，顺序存放

由于 $\boldsymbol{A}$ 的列是 $\boldsymbol{B}$ 的行，因此，按 a -> data 的列序转置，所得到的转置矩阵 $\boldsymbol{B}$ 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 $\boldsymbol{A}$ 中的每一列 col（$0\leqslant col\leqslant$ 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 $\boldsymbol{B}$ 的按行优先的压缩存储表示。

例 5.2　一个用三元组表表示的稀疏矩阵的转置（按列序转置，顺序存放）的实例。

```
#include  "stdio.h"
#define MaxSize 10 000                    /* 非零元素个数的上限，三元组表的容量 */
typedef int DataType;                     /* 非零元素的类型定义 */
typedef struct {
    int i,j;                              /* 非零元素的行号、列号 */
    DataType v;                           /* 非零元素的值 */
}TriTupleNode;                            /* 三元组结点的类型 */
```

```
typedef struct{
    TriTupleNode data[MaxSize];                 /*三元组表*/
    int m,n;                                    /*矩阵的行数、列数*/
    int t;                                      /*当前表长,即非零元素的个数*/
}TriTupleTable;                                 /*稀疏矩阵类型*/

void TransMatrix(TriTupleTable *b,TriTupleTable *a); /*函数说明*/
void main()
{
    TriTupleTable *a,*b;
    TriTupleNode  node[MaxSize];                /*存放三元组表的数组*/
    int num;
    node[0].i=0;node[0].j=1;node[0].v=5;
    node[1].i=0;node[1].j=4;node[1].v=8;
    node[2].i=1;node[2].j=0;node[2].v=1;        转置前矩阵所对应的三元组表,
    node[3].i=1;node[3].j=2;node[3].v=3;        对前面的B矩阵按列优先存
    node[4].i=2;node[4].j=1;node[4].v=-2;
    node[5].i=3;  node[5].j=0;  node[5].v=6;

    for(num=0;num<=5;num++)
    {a->data[num].i=node[num].i;                将三元组表node[]复制
    a->data[num].j=node[num].j;                 到三元组表a->data[]中
    a->data[num].v=node[num].v;
    }
    a->m=4;a->n=5;a->t=6;  /*记下三元组表中表示阵a的行、列和非零元素个数*/
    TransMatrix(b,a);                           /*函数调用进行转置*/
    for(num=0;num<b->t;num++)                   /*输出转置后的三元组表*/
        printf("%d,%d,%d\n",b->data[num].i,b->data[num].j,b->data[num].v);
}

void TransMatrix(TriTupleTable *b,TriTupleTable *a)
{/**a,*b是矩阵A、B的三元组表表示,求A转置为B  */
    int pa,pb,col;
    b->m=a->n; b->n=a->m;                       /*A和B的行列总数互换*/
    b->t=a->t;                  /*非零元素总数,转置后阵A和阵B的非零元素相等*/
    if(b->t<=0)                                 /*判断稀疏矩阵有无非零元素*/
        {printf("A=0");  exit(0);}               /*A中无非零元素,退出*/
    pb=0;                                       /*pb为B中三元组表当前空位置,*/
    /*当在a->data[]中找到非零元素则就放到pb所指的b->data[pb]中*/
    for(col=0;col<a->n;col++)                   /*对A的每一列*/
        for(pa=0;pa<a->t;pa++)                  /*扫描A的三元组表*/
            if(a->data[pa].j==col)
            { /*找列号为col的三元组结点,行列号互换*/
                b->data[pb].i=a->data[pa].j;
                b->data[pb].j=a->data[pa].i;
                b->data[pb].v=a->data[pa].v;
                pb++;
            }
}
```

运行结果如下：

```
0,1,1
0,3,6
1,0,5
1,2,-2
2,1,3
4,0,8
```

由于三元组表的元素按行序排列，上述算法在扫描 ***A*** 的三元组表时，同一列上的非零元素必然是按行号大小的顺序出现的，因此转置后的三元组表中行号相同的元素正好按列号排列，列号相同的元素正好按行号排序。

该算法的时间主要耗费在 col 和 pa 的二重循环上：若 ***A*** 的列数为 n，非零元素个数为 t，则执行时间为 $O(n\times t)$，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是 $O(m\times n)$，它与矩阵行数和列数的乘积成正比。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耗费大于通常的转置算法的时间耗费。

2）按行序转置，按列索引存放

对于矩阵 ***A*** 和 ***B***，***B*** 的行数、列数和非零元素的个数等于 ***A*** 的列数、行数和非零元素的个数，且 ***B*** 中的每个非零元素和 ***A*** 中的非零元素相比，它们的值相同，但行、列号互换。由于三元组表中元素的顺序约定为以行序为主序，即在三元组表 b -> data 中非零元素的排列次序是以它们在三元组表 a -> data 中的列号为主序的。因此转置的主要操作就是要确定 ***A*** 中的每个非零元素在 ***B*** 的三元组顺序表中的位序，即分析两个矩阵中值相同的非零元素分别在 a -> data 和 b -> data 中的位序之间的关系。

由此可知，该转置算法的操作步骤为：

（1）求 ***A*** 矩阵的每一列中非零元素的个数。

（2）确定 ***B*** 矩阵的每一行中第 1 个非零元素在 b -> data 中的序号。

（3）将 a -> data 中的每个元素依次复制到 b -> data 中相应的位置。

例 5.3 一个用三元组表表示的稀疏矩阵的转置（按行序转置，按列索引存放）的实例。具体算法如下：

```
void FastTransMatrix(TriTupleTable *a,TriTupleTable *b)
{/** a、*b 是矩阵 A、B 的三元组表表示，将 A 转置为 B*/
    int pa,pb,col;
    int *cpos;  /*位置向量，记 A 数组中每列的第 1 个非零元素在三元组表 b 中的位置*/
    int *cnum;    /*记数向量，记录矩阵 A 中各列上非零元素的个数*/
    b->m=a->n; b->n=a->m;            /*A 和 B 的行列总数互换*/
    b->t=a->t;                       /*非零元素总数*/
    if(a->t<=0)                      /*A 中无非零元素，退出*/
    {
        printf("A 中无非零元素，退出");
        exit(0);
    }
    /*申请空间，以存放 a 每列的第 1 个非零元素在三元组 b 中的存放位置*/
```

```
    cpos = malloc(sizeof(int) * ( a -> n));
    cnum = malloc(sizeof(int) * ( a -> n)); /* 申请空间,存放 a 的每列非零元素的个数 */
    /* 动态申请辅助空间(不考虑空间不足申请失败的情况) */
    for(col = 0;col < a -> n;col++)
    cnum[col] = 0;                          /* 开始时数组 a 的每列的非零元素个数为 0 */
    for(pa = 0;pa < a -> t;pa++)            /* 累计每列的非零元素个数 */
    {
        col = a -> data[pa].j;              /* 将三元组表中元素的列号送入 col 中 */
        cnum[col]++;   /* 用指针变量的下标表示法,记下数组 a 的每列的非零元素个数 */
    }
    for(col = 0;col < a -> n;col++)
        cpos[col] = 0;
    for (col = 1; col < a -> n; col++)     /* 求 A 中每列第 1 个非零元素在 b→data 中的位置 */
       cpos[col] = cpos[col - 1] + cnum[col - 1];
    for(pa = 0;pa < a -> t;pa++) /* 将 a -> data 中每个元素依次复制到 b -> data 中相应位置 */
    {
        col = a -> data[pa].j;
        pb = cpos[col];                     /* A 中该列在三元组表 b 中的起始位置 */
        b -> data[pb].i = a -> data[pa].j;
        b -> data[pb].j = a -> data[pa].i;
        b -> data[pb].v = a -> data[pa].v;
        cpos[col]++;                        /* 矩阵 A 中该列下一个元素在 b 中的位置 */
    }
    free (cpos);   /* 释放辅助数组空间 */
    free (cnum);
}
```

运行结果如下:

```
0,1,1
0,3,6
1,0,5
1,2,-2
2,1,3
4,0,8
```

上述算法的时间复杂度为 $O(n+t)$,比前一个效率高得多,故这种转置又称为快速转置。即使对非稀疏矩阵,该算法也有意义,因为当 t 接近 $m\times n$ 时,时间复杂度为 $O(m\times n)$,与不压缩直接转置的算法一样。

实际上,为了方便某些矩阵运算,在按行优先存储的三元组表中,加入一个行表来记录稀疏矩阵中每行的非零元素在三元组表中的起始位置。这就是带行(索引)表的三元组表。这样可以方便地找到某行的第1个非零元素以及该行非零元素的个数,其原理与前面建立列索引的方法类似。

三元组表和带行表的三元组表相应的算法描述较为简单,但这类顺序存储方式对于非零元素的位置或个数经常发生变化的矩阵运算就不太适合。例如,执行将矩阵 $\boldsymbol{B}$ 加到矩阵 $\boldsymbol{A}$ 上的运算时,某位置上的结果可能会由非零值变为零值,但也可能由零值变为非零值,这就会引起在三元组表中进行删除和插入操作,从而导致大量结点的移动。对此类运算采用

链式存储结构为宜。稀疏矩阵的链式结构有十字链表等方法，适用于非零元素变化大的场合，比较复杂，限于版面，在此就不讨论了。

5.3 广 义 表

广义表(lists)也称为**列表**，它是线性表的推广。线性表是 $n(n\geqslant 0)$ 个元素 $a_1,a_2,\cdots,a_i,\cdots,a_n$ 的有限序列。线性表的元素仅限于原子项，所谓**原子**，指的是结构上不可再分割的一种成分，它可以是一个数，也可以是一个结构。如果放宽对线性表元素的这种限制，允许它们具有其自身独立的类型结构，那么就产生了广义表的概念。

广义表是 $n(n\geqslant 0)$ 个元素 $a_1,a_2,\cdots,a_i,\cdots,a_n$ 的有限序列，其中 a_i 可以是原子，也可以是一个广义表。通常，广义表可记做 $LS=(a_1,a_2,\cdots,a_i,\cdots,a_n)$。LS 是广义表的名字，$n$ 为广义表 LS 的**长度**。若 a_i 本身也是广义表，则称它为 LS 的**子表**。不包含任何元素($n=0$)的广义表称为空表。

需要指出的是：

(1) 广义表通常用圆括号括起来，用逗号分隔其中的元素。

(2) 为区分原子和广义表，用大写字母表示广义表，用小写字母表示原子。

(3) 若广义表 LS 非空($n\geqslant 1$)，则 a_1 称为 LS 的**表头**，其余元素组成的表($a_2,\cdots,a_i,\cdots,a_n$)称为 LS 的**表尾**。显然，表尾一定是子表，但表头可以是原子，也可以是子表。

(4) 广义表是递归定义的，因为在定义广义表时又用到了广义表的概念。

可见，若不考虑广义表元素内部的结构，广义表 LS 就是一个线性表；反之，线性表就是一个不含子表的广义表。

广义表的**深度**是指该表展开后所含括号的层数。

例 5.4 广义表示例。

(1) $E=()$：E 是一个空表，它既无表头，又无表尾，其长度为 0，深度为 1。

(2) $L=(a,b)$：表头为 a，表尾为(b)，长度为 2，深度为 1，它是一个线性表。

(3) $A=(x,L)=(x,(a,b))$：表头为 x，表尾为$((a,b))$，长度为 2，深度为 2。

(4) $B=(A,y)=((x,(a,b)),y)$：表头为 A，表尾为(y)，长度为 2，深度为 3。

(5) $C=(A,B)=((x,(a,b)),((x,(a,b)),y))$：表头为 A，表尾为(B)，长度为 2，深度为 4。

(6) $D=(a,D)=(a,(a,(a,(\cdots))))$：表头为 a，表尾为(D)，长度为 2，深度为 ∞。这是一个递归表，展开后，它是一个无限的广义表。

(7) $F=(())$：表头为()，表尾为()，长度为 1，深度为 2。这里，F 的表头和表尾都为空表。

> 需要注意的是，广义表()和(())不同。前者是长度为 0 的空表，对其不能做求表头和表尾的运算；而后者是长度为 1 的非空表(只不过该表中唯一的一个元素是空表)，对其可以进行分解，得到的表头和表尾均是空表()。

如果规定任何表都是有名字的，为了既表明每个表的名字，又说明它的组成，则可以在

每个表的前面冠以该表的名字,于是上例中的各表又可以写成:

(1) $E()$。

(2) $L(a,b)$。

(3) $A(x,L(a,b))$。

(4) $B(A(x,L(a,b)),y)$。

(5) $C(A(x,L(a,b)),B(A(x,L(a,b)),y))$。

(6) $D(a,D(a,D(\cdots)))$。

(7) $F(())$。

广义表还可以用图形来形象地表示,图5.8给出了上面几个广义表的图形表示,其中的分支结点对应广义表,非分支结点(叶子)对应原子或者空表。如果与后面将要介绍的树、图等内容联系起来,可以把与树对应的广义表称为**纯表**(pure list),这种表中没有共享和递归的成分,即没有任何成分出现多次,它限制了表中成分的共享和递归,例如图5.8(a)、图5.8(b)、图5.8(c)都是纯表;把与有向无环图对应的表称为**再入表**,这种表存在元素共享,在图中表现为存在结点共享,例如,图5.8(d)中,子表A是共享结点,它既是C的一个元素,又是子表B的元素;把与有回路的有向图对应的表称为**递归表**,这种表的某个成员内含有广义表自己,例如,图5.8(e)中,表D是其自身的子表。各种表之间的关系满足:

递归表⊃再入表⊃纯表⊃线性表

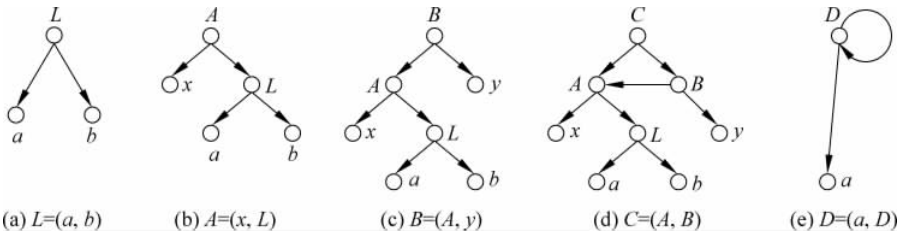

图5.8 广义表的图形表示

由此可见,广义表不仅是线性表的推广,也是树和图的推广。由于广义表的元素可以递归,所以广义表具有很强的表达能力,这是广义表最重要的特性。

广义表的基本运算,除包括线性表的基本运算外,还有求深度、取表头、取表尾、遍历等。这些运算中大部分与对应的线形表、树或者图的运算类似,只是取表头和取表尾是广义表特有的运算。

在此,只讨论广义表的两个特殊的基本运算:取表头head(LS)和取表尾tail(LS)。根据表头、表尾的定义可知:任何一个非空广义表的表头是表中第1个元素,它可以是原子,也可以是子表,而其表尾必定是子表。

例5.5 求例5.4中广义表L和B的表头与表尾。

$$\text{head}(L)=a,\quad \text{tail}(L)=(b)$$

$$\text{head}(B)=A,\quad \text{tail}(B)=(y)$$

例5.6 通过取表头head(LS)和取表尾tail(LS)运算,从广义表$A=(x,(a,b),y)$中取出原子b。

分析：在广义表中取某个元素，需要将该元素所在的子表逐步分离出来，直到所求的元素成为某个子表的表头，再用取表头运算取出。值得注意的是，最终取出某个元素时，不能再取表尾，因为它得到的是该元素组成的子表，而不是元素本身。本例题求解的过程为：

(1) 取表尾 tail(A)：得到 $B=((a,b),y)$

(2) 取表头 head(B)：得到 $C=(a,b)$

(3) 取表尾 tail(C)：得到 $D=(b)$

(4) 取表头 head(D)：得到 b

于是，可以得到：head(tail(head(tail(A))))$=b$。

本章小结

- 多维数组是一种最简单的非线性结构，其存储结构也是比较简单的，大多数程序设计语言采用顺序存储方式表示数组，存放顺序有的采用以行(序)为主(序)，有的则采用以列(序)为主(序)。在 C 语言中采用以行(序)为主(序)的存放顺序。
- 由于二维数组与矩阵相对应，因此平时二维数组的使用最为频繁。对于一些特殊的矩阵，采用二维数组表示会浪费存储空间。本章介绍了数组的定义和表示方式，以及特殊矩阵和稀疏矩阵的压缩存储方法及其运算的实现。
- 广义表是线性表的推广，它是一种复杂的非线性结构。本章简要地介绍了广义表的概念和基本运算。

习　题　5

一、基础知识题

1. 给出 C 语言的三维数组地址计算公式。

2. n 阶对称矩阵 $\boldsymbol{A}$ 的下三角元素存储在一维数组 B 中，则 B 包含多少个元素？

3. 设有三对角矩阵 n 阶方阵 $\boldsymbol{A}[1..n][1..n]$，将其三条对角线上的元素逐行地存储到向量 $\boldsymbol{B}[0..3n-3]$中，使得 $\boldsymbol{B}[k]=a_{ij}$，求：

(1) 用 i、j 表示 k 的下标变换公式。

(2) 用 k 表示 i、j 的下标变换公式。

4. 设二维数组 $\boldsymbol{A}[5][6]$的每个元素占 4 字节，已知 Loc(a_{00})$=1000$，则 $\boldsymbol{A}$ 共占多少字节？$\boldsymbol{A}$ 的终端结点的起始地址是什么？按行和按列优先存储时，$a[2][5]$的起始地址分别是什么？

5. 特殊矩阵和稀疏矩阵哪一种压缩存储后会失去随机存取的功能？为什么？

6. 数组、广义表与线性表之间有什么样的关系？

7. 画出下列广义表的图形表示：

(1) $A(a,B(b,d),C(e,B(b,d),L(f,g)))$

(2) $A(a,B(b,A))$

8. 设广义表 $L=(()，())$，试问 head(L)、tail(L)、L 的长度、深度各为多少？

9. 利用广义表的 head 和 tail 运算,把原子 d 分别从下列广义表:

$$L_1=(((((a),b),d),e));\ L_2=(a,(b,((d)),e))$$

中分离出来。

10. 求下列广义表运算的结果:

(1) head(tail(((a,b),(c,d),(e,f))))

(2) tail(head(((a,b),(c,d),(e,f))))

(3) head(tail(head(((a,b),(e,f)))))

(4) tail(head(tail(((a,b),(e,f)))))

(5) tail(tail(head(((a,b),(e,f)))))

二、算法设计题

1. 编写一个过程,对一个 $n\times n$ 矩阵,通过行变换,使其每行元素的平均值按递增顺序排列。

2. 当稀疏矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 均以三元组表作为存储结构时,试写出矩阵相加的算法,其结果存放在三元组表 C 中。