第 5 章　数组和广义表

数组和广义表是线性表的推广，表中的数据元素本身也可以是一个数据结构。本章介绍数组和广义表的定义、存储结构与算法实现。

5.1　数组的基本概念

5.1.1　数组的定义与特点

1. 数组的定义

数据结构中讨论的数组(array)不同于高级语言中的数组。高级语言中数组是一种数据类型，而且只有顺序结构。本章所讨论的数组是一种数据结构，既可以是顺序结构，也可以是链式结构，用户可根据需要选择。

数组 $A=(a_0,a_1,a_2,\cdots,a_{n-1})$ 由一组名字相同、下标不同的变量构成。若 $a_i(0\leqslant i\leqslant n-1)$ 是同类型的元素，则 A 是一维数组。若 $a_i(0\leqslant i<n)$ 是同类型的定长线性表，则 A 是多维数组。

在 C++ 语言中，一个二维数组可以用其分量一维数组来定义。同样，一个三维数组可以用其数据元素为二维数组的线性表来定义，以此类推，可得到 n 维数组的递归定义。

2. 数组的特点

在数组中，对于一组有意义的下标都存在一个与其相对应的值。这种下标与值一一对应的关系是数组的特点。

一维数组的每个数组元素只有一个下标，每个元素 $a_i(1\leqslant i\leqslant n-2)$ 有唯一一个直接前趋结点 a_{i-1} 和唯一一个直接后继结点 a_{i+1}。其中，a_0 没有直接前趋结点，a_{n-1} 没有直接后继结点。

二维数组的每个数组元素有两个下标，每个元素 a_{ij} 受到两个关系(行关系和列关系)的约束。$a_{ij}(1\leqslant i\leqslant m-2,1\leqslant j\leqslant n-2)$ 有两个直接前趋结点 $a_{i,j-1}$ 和 $a_{i-1,j}$，两个直接后继结点 $a_{i,j+1}$ 和 $a_{i+1,j}$。其中，a_{00} 没有前趋结点，称为开始结点；$a_{m-1,n-1}$ 没有后继结点，称为终端结点。

可以把二维数组中的元素 a_{ij} 看成是属于两个线性表的，即第 i 行的线性表 A_i 和第 j 列的线性表 B_j。例如，一个 $m\times n$ 的二维数组可以看成是 m 行的一维数组，如图 5.1 所示；也可看成是 n 列的一维数组，如图 5.2 所示。

因此，二维数组是一个数据元素值为一维数组的线性表。

以此类推，一个 $m\times n\times l$ 的三维数组中，每个元素属于 3 个线性表，每个元素最多有 3 个直接前趋和 3 个直接后继。

例如，一个数组元素 $a_{j1,j2,j3}$，其直接前趋为：$a_{j1-1,j2,j3}$，$a_{j1,j2-1,j3}$，$a_{j1,j2,j3-1}$；其直接后继为：$a_{j1+1,j2,j3}$，$a_{j1,j2+1,j3}$，$a_{j1,j2,j3+1}$。

$$\boldsymbol{A}_{mn}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \qquad \boldsymbol{A}_{mn}=\begin{bmatrix} (a_{00} & a_{01} & \cdots & a_{0,n-1}) \\ (a_{10} & a_{11} & \cdots & a_{1,n-1}) \\ \vdots & \vdots & \cdots & \vdots \\ (a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}) \end{bmatrix} \begin{matrix} \boldsymbol{A}_0 \\ \boldsymbol{A}_1 \\ \vdots \\ \boldsymbol{A}_{m-1} \end{matrix} \qquad \boldsymbol{A}=\begin{pmatrix} \boldsymbol{A}_0 \\ \boldsymbol{A}_1 \\ \vdots \\ \boldsymbol{A}_{m-1} \end{pmatrix}$$

(a) $m\times n$ 的二维数组 $\boldsymbol{A}_{mn}$　　(b) $\boldsymbol{A}_{mn}$ 可看成 m 行一维数组　　(c) 用 m 行一维数组表示二维数组

图 5.1　用 m 行的一维数组表示 $m\times n$ 的二维数组

$$\boldsymbol{A}_{mn}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \qquad \boldsymbol{A}_{mn}=\begin{bmatrix} \begin{pmatrix} a_{00} \\ a_{10} \\ \vdots \\ a_{m-1,0} \end{pmatrix} & \begin{pmatrix} a_{01} \\ a_{11} \\ \vdots \\ a_{m-1,1} \end{pmatrix} & \cdots & \begin{pmatrix} a_{0,n-1} \\ a_{1,n-1} \\ \vdots \\ a_{m-1,n-1} \end{pmatrix} \\ \uparrow \boldsymbol{B}_0 & \uparrow \boldsymbol{B}_1 & \cdots & \uparrow \boldsymbol{B}_{n-1} \end{bmatrix} \qquad \boldsymbol{A}_{mn}=(\boldsymbol{B}_0\ \boldsymbol{B}_1\cdots\boldsymbol{B}_{n-1})$$

(a) $m\times n$ 的二维数组 $\boldsymbol{A}_{mn}$　　(b) $\boldsymbol{A}_{mn}$ 可视为 n 列一维数组　　(c) 用 n 列一维数组表示二维数组

图 5.2　用 n 列的一维数组表示 $m\times n$ 的二维数组

进一步，一个 n 维数组可以看成是由若干个 $n-1$ 维数组组成的线性表，在 n 维数组 $\boldsymbol{A}$ 中每个元素 $a_{j_1,j_2,\cdots,j_n}$（$0\leqslant j_i\leqslant n-1$）属于 n 个线性表，每个元素最多有 n 个直接前趋和 n 个直接后继。

例如，数组元素 $a_{j_1,j_2,\cdots,j_n}$，其直接前趋为：$a_{j_1-1,j_2,\cdots,j_n}$，$a_{j_1,j_2-1,\cdots,j_n}$，…，$a_{j_1,j_2,\cdots,j_n-1}$；其直接后继为：$a_{j_1+1,j_2,\cdots,j_n}$，$a_{j_1,j_2+1,\cdots,j_n}$，…，$a_{j_1,j_2,\cdots,j_n+1}$。

5.1.2　数组的存储结构

1. 一维数组的顺序存储

在计算机中，如果用一批连续的存储单元来表示数组，则称为数组的顺序存储结构。C++ 语言中数组的顺序分配允许有两种方法：静态存储分配和动态存储分配，简称静态数组和动态数组。

（1）静态数组：是指数组大小在程序中预先给出、在程序编译时建立的数组。程序运行期间，数组大小是静态的，不能改变的。

（2）动态数组：是指在程序执行过程中动态建立和撤销的数组。

数组是采用计算地址的方法来实现的。一维数组的地址计算公式和顺序表的元素分配一样。

如图 5.3 所示，对一个有 n 个数据元素的一维数组，设 a_0 是下标为 0 的数组元素，Loc(a_0) 是 a_0 的存储地址，L 是每个数据元素所需的存储单元，则数组中任一数据元素 a_i（$0\leqslant i\leqslant n-1$）的存储地址 Loc($a_i$) 的计算函数为

$$\mathrm{Loc}(a_i)=\mathrm{Loc}(a_0)+i\times L,\qquad 0\leqslant i\leqslant n-1$$

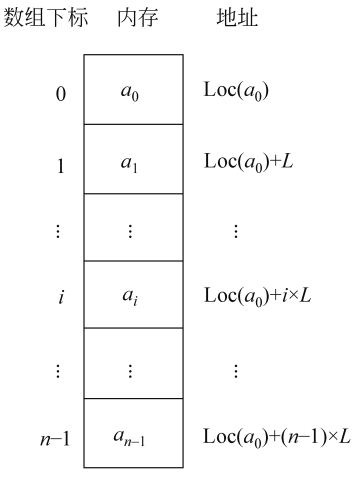


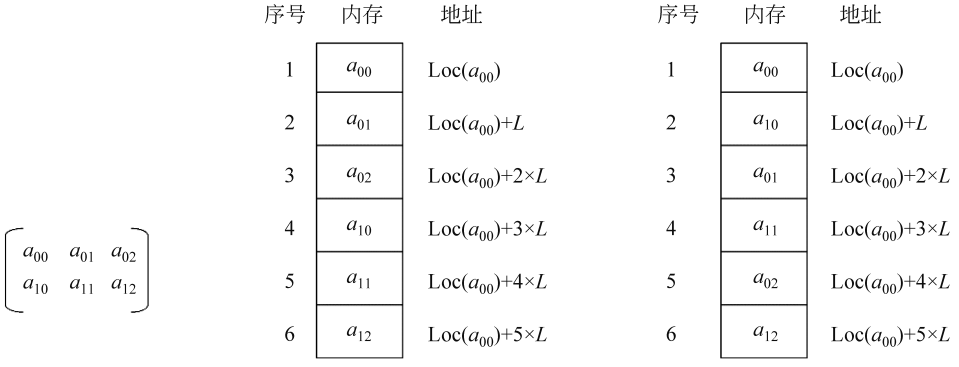
图 5.3　一维数组顺序存储示意图

C++ 语言中内置的数组是符合上述定义的数据结构，因此在这里使用它来存储一维数组。这种情况下，不需要显性地对数据元素的存储地址进行管理。

2. 二维数组的顺序存储

对一个 m 行 n 列的二维数组，由于计算机的存储单元都是一维的，就有一个二维向一维映射的问题。对二维数组可有两种存储方式：一种是行主序(row major order)存储，即一行存完后再存放下一行；另一种是列主序(column major order)存储，即一列存完后再存放下一列。

例如，图 5.4(a)的二维数组 $\boldsymbol{A}[0\cdots1,0\cdots2]$，其行主序存储如图 5.4(b)所示，其列主序存储如图 5.4(c)所示。



$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$$

(a) 二维数组 $\boldsymbol{A}$　　(b) 行主序存储　　(c) 列主序存储

图 5.4　二维数组顺序存储示意

C++ 语言中的数组元素是行主序的存放方法。设 a_{00} 是行列下标均为 0 的数组元素，Loc(a_{00})是 a_{00} 的存储地址，L 是每个数据元素所需的存储单元。数组元素 a_{ij} 前已存放了 i 行，即已存放了 $i\times n$ 个数据元素，占用了 $i\times n\times L$ 个存储单元；在第 i 行上，数组元素 a_{ij} 前已存放了 j 列，即已存放了 j 个数据元素，占用了 $j\times L$ 个存储单元，所以数组元素 a_{ij} ($0\leqslant i\leqslant m-1, 0\leqslant j\leqslant n-1$)的存储地址 Loc($a_{ij}$)为上述 3 个部分之和。因此，数组中任一数据元素 a_{ij} 的存储地址 Loc(a_{ij})的计算函数为

$$\text{Loc}(a_{ij})=\text{Loc}(a_{00})+(i\times n+j)\times L,\qquad 0\leqslant i\leqslant m-1, 0\leqslant j\leqslant n-1$$

在列主序存储中，数组元素地址的计算函数可仿照行主序的计算函数推导出来。C++ 语言中数组的各维下界均固定为 0，而其他语言不一定为 0，因此计算公式略有变化。

由于在二维数组中涉及行和列的定义，因此不难由此联想到矩阵的概念。

5.2　特殊矩阵的压缩存储

矩阵(matrix)运算是许多科学和工程计算中经常遇到的问题。通常用二维数组来存储矩阵数据元素。利用矩阵元素在一维数组中的地址计算函数就可以快速地访问矩阵中的每个元素。

实际应用中常常会遇到一些阶数很高的矩阵，它们有许多值相同的元素或零元素，且值相同的元素或零元素的分布有一定的规律，这种分布有一定规律的矩阵称为特殊矩阵。常

见的特殊矩阵有对称矩阵、三角矩阵和对角矩阵。

特殊矩阵可以进行压缩存储。所谓压缩存储，就是为多个值相同的元素只分配一个存储空间，且对零元素不分配空间。

1. 对称矩阵的压缩存储

若一个 n 阶矩阵 $\boldsymbol{A}$ 中的数据元素满足 $a_{ij}=a_{ji}(0\leqslant i\leqslant n-1,0\leqslant j\leqslant n-1)$，则称其为 n 阶对称矩阵。

由于对称矩阵中的数据元素以主对角线为中线对称，因此在存储时可只存储对称矩阵中上三角或下三角的数据元素，即让对称的数据元素共享一个存储空间，这样就可将 n^2 个数据元素压缩存储到 $n(n+1)/2$ 个数据元素空间中。

如图 5.5 所示，不失一般性，以存储的是对称矩阵（见图 5.5(a)）的下三角（包括对角线）的数据元素（见图 5.5(b)）为例进行讨论，图 5.5(c)为对称矩阵的一维数组压缩存储。

$$\begin{pmatrix} 1 & 6 & 7 & 3 & 9 \\ 6 & 0 & 8 & 2 & 8 \\ 7 & 8 & 9 & 2 & 9 \\ 3 & 2 & 2 & 0 & 1 \\ 9 & 8 & 9 & 1 & 6 \end{pmatrix} \qquad \begin{pmatrix} a_{00} & & & & \\ a_{10} & a_{11} & & & \\ a_{20} & a_{21} & a_{23} & & \\ \vdots & \vdots & \vdots & & \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & \cdots & a_{n-1,n-1} \end{pmatrix}$$

(a) 对称矩阵　　(b) 存储下三角部分的数据元素

a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	...	$a_{n-1,n-2}$	$a_{n-1,n-1}$

(c) 对称矩阵的一维数组压缩存储

图 5.5　对称矩阵及其下三角存储

可用一维数组 sa[0…$n(n+1)/2-1$]作为 n 阶对称矩阵 $\boldsymbol{A}$ 的存储结构，使得 n 阶对称矩阵 $\boldsymbol{A}$ 中的所有下三角（包括对角线）的数据元素对应不同的存储单元，所有上三角（不包括对角线）的数据元素对应下三角的相应存储单元。设 a_{ij} 为 n 阶对称矩阵中 i 行 j 列的数据元素，k 为一维数组 sa 的下标序号。

(1) 若 $i\geqslant j$，则 a_{ij} 在下三角中。a_{ij} 之前的 i 行共有 $i(i+1)/2$ 个元素，在第 i 行上，a_{ij} 之前有 j 个元素。a_{ij} 和 $sa[k]$ 之间的对应关系是

$$k=i(i+1)/2+j,\qquad 0\leqslant k<n(n+1)/2$$

(2) 若 $i<j$，则 a_{ij} 在上三角中。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 a_{ij} 和 sa[k]之间的对应关系，即

$$k=j(j+1)/2+i,\qquad 0\leqslant k<n(n+1)/2$$

2. 三角矩阵的压缩存储

当一个方阵的主对角线上方或下方的所有元素皆为常数时，该矩阵称为三角矩阵。以主对角线划分，三角矩阵有上三角和下三角两种。如图 5.6(a)所示，上三角矩阵的下三角（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，如图 5.6(b)所示。在大多数情况下，三角矩阵常数为零。

三角矩阵可以用一维数组 sa[0…$n(n+1)/2$]存储，将常量存入第一个或最后一个存储

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

(a) 上三角矩阵　　(b) 下三角矩阵

图 5.6　三角矩阵

单元。

3. 对角矩阵的压缩存储

设矩阵 $\boldsymbol{A}$ 为 n 阶方阵。序列(a_{ii},$0\leqslant i\leqslant n-1$)称为 $\boldsymbol{A}$ 的主对角线。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 5.7(a)是一个带宽为 3 的对角矩阵及其顺序结构存储,其非零元素仅出现在主对角线上(a_{ii},$0\leqslant i\leqslant n-1$),紧邻主对角线上面的那条对角线上($a_{i,i+1}$,$0\leqslant i\leqslant n-2$)和紧邻主对角线下面的那条对角线上($a_{i+1,i}$,$0\leqslant i\leqslant n-2$)。当$|i-j|>1$时,元素 $a_{ij}=0$。

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

不失一般性,对角矩阵可按行优先顺序或对角线的顺序,将其压缩存储到一个一维数组中,并且也能找到每个非零元素和数组下标的对应关系。图 5.7(b)为行主序的一维数组压缩存储。若采用行优先顺序,则存储带宽为 d 的对角矩阵中非零元素 a_{ij} 的地址公式为

$$\mathrm{Loc}(a_{ij})=\mathrm{Loc}(a_{00})+d\times i-1+(j-i+(d-1)/2)$$

$$\begin{pmatrix} a_{00} & a_{01} & & & & & \\ a_{10} & a_{11} & a_{12} & & & \text{全 } 0 & \\ & a_{21} & a_{22} & a_{23} & & & \\ & & \cdots & \cdots & \cdots & & \\ & & & a_{n-2,n-3} & a_{n-2,n-2} & a_{n-2,n-1} & \\ & \text{全 } 0 & & & a_{n-1,n-2} & a_{n-1,n-1} & \end{pmatrix}$$

(a) 带宽为 3 的对角矩阵

a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	...	$a_{n-1,n-2}$	$a_{n-1,n-1}$

(b) 行主序的一维数组压缩存储

图 5.7　对角矩阵及其压缩存储

4. 稀疏矩阵

设 $m\times n$ 的矩阵 $\boldsymbol{A}$ 中有 s 个非零元素,若 s 远远小于矩阵元素的总数,则称 $\boldsymbol{A}$ 为稀疏矩阵。令 $e=s/(m\times n)$,称 e 为矩阵的稀疏因子。通常认为 $e\leqslant 0.05$ 时为稀疏矩阵。

稀疏矩阵中非零元素的分布一般是没有规律的,因此在存储非零元素的同时,还必须同时记下它所在的行和列的位置(i,j)。

稀疏矩阵中每个非零元素和它对应的行下标和列下标构成一个三元组。三元组的结构

如图 5.8 所示，其中 i，j 分别表示非零元素的行号和列号，a_{ij} 表示非零元素的值。

i	j	a_{ij}

图 5.8　三元组的结点结构

一个三元组(i,j,a_{ij})可以唯一确定矩阵 $\boldsymbol{A}$ 的一个非零元素。稀疏矩阵可由表示非零元素的三元组及其行列数唯一确定。

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

(a) 稀疏矩阵 M

6	7	6
1	3	9
1	5	−7
3	1	8
4	3	−2
5	2	5
6	6	3

(b) 稀疏矩阵 M 的三元组表 a.data

图 5.9　稀疏矩阵 $\boldsymbol{M}$ 及其三元组表 a.data

例如，图 5.9(a)所示的稀疏矩阵 $\boldsymbol{M}$ 有 42 个元素，其中只有 6 个非零元素。6 个三元组(1,3,9)，(1,5,−7)，(3,1,8)，(4,3,−2)，(5,2,5)，(6,6,3)表示该矩阵的 6 个非零元素。若以行主序将这 6 个三元组排列起来，再加上一个表示矩阵 $\boldsymbol{M}$ 的行数、列数及非零元素的个数的特殊三元组(6,7,6)，则所形成的三元组表就能唯一确定稀疏矩阵 $\boldsymbol{M}$，如图 5.9(b)所示。

三元组表可用顺序表存储，也可用链表存储。用顺序表存储的三元组表称为三元组顺序表，用链表存储的三元组表称为三元组链表。

对于稀疏矩阵，三元组表存储结构对存储空间的需求量比通常的方法少得多。例如，$m\times n$ 的矩阵 $\boldsymbol{M}$，非零元的个数为 num，若用三元组表来表示，在每个元素占一个存储单元的情况下，只需要 $3\times(\text{num}+1)$个存储单元（包括特殊三元组所占用的 3 个单元）；若用传统的二维数组表示，则需要 $m\times n$ 个存储单元。当矩阵越大、越稀疏时，三元组存储方式的优越性就越明显。

5.3　矩阵的算法实现

矩阵最常用的操作有矩阵加、矩阵减、矩阵乘、矩阵转置、矩阵求逆、矩阵行列式等。在掌握线性表的基本操作的基础上，这些操作都不难实现。但是，稀疏矩阵的快速转置算法思想巧妙，性能优良。因此，下面仅以稀疏矩阵的快速转置算法为例进行算法实现。

一个 $m\times n$ 的矩阵 $\boldsymbol{M}$，它的转置矩阵 $\boldsymbol{T}$ 是一个 $n\times m$ 的矩阵，且 $T_{ij}=M_{ji}$，其中，$0\leqslant i\leqslant n-1$，$0\leqslant j\leqslant m-1$。

在这里沿用上面介绍的三元组的方式存储矩阵。

明确了存储方式后，就不难进行相应的存储结构的定义。

三元组的定义如下。

```
1.  template<typename T>
2.  struct Triple {
3.      int i, j;                                    //非零元素的行下标和列下标
4.      T elem;
5.  };
```

矩阵类的定义如下。

```
1.  template<typename T>
2.  struct Matrix {
3.      int row, col;
4.      int num;
5.      Triple<T> * data;
6.
7.      Matrix(int r, int c, int n) : row(r), col(c), num(n) {
8.          data =new Triple<T>[num +1];             //非零三元组表,data[0]未用
9.      }
10.
11.     ~Matrix() { delete[]data; }
12. };
```

上述的两段代码共同构成了矩阵的头文件。

1. 算法功能

采用三元组表存储矩阵,由矩阵 ***M*** 的三元组表 a.data 求得其转置矩阵 ***T*** 的三元组表 b.data,实现矩阵 ***M*** 的转置。

2. 算法思路

(1) 预先确定矩阵 ***M*** 中每一列,即转置矩阵 ***T*** 中每一行的第一个非零元素在 b.data 中的位置。为此,需事先求得矩阵 ***M*** 的每一列中非零元素的个数。可设置两个数组 num[col]和 cpot[col],分别存放矩阵 ***M*** 中每一列的非零元素个数和每一列第一个非零元素在 b.data 中的位置,即

- M 中的列变量用 col 表示。
- num[col]存放 ***M*** 中第 col 列中非零元素个数。
- cpot[col]存放 ***M*** 中第 col 列的第一个非零元素在 b.data 中的位置。

于是有

$$\begin{cases} \text{cpot}[1]=1 \\ \text{cpot}[\text{col}]=\text{cpot}[\text{col}-1]+\text{num}[\text{col}-1] \end{cases}, \qquad 2\leqslant \text{col} \leqslant \text{a.col}$$

(2) 按照 a.data 中三元组的次序进行转置。对于扫描到的 a.data 中的当前元素,利用其列信息,即可在 cpot[col]中直接查出它在 b.data 中的最终存放位置。

转置后的元素不连续存放,直接按照 cpot[col]的结果将其放到 b.data 中最终的位置上。同时,修改该列的 cpot[col]为 cpot[col]+1,为该列的下一个元素的存放做好准备。这样既可避免元素移动,又只需对 a.data 扫描一次即可实现矩阵的转置。

3. 实例描述

稀疏矩阵 ***M*** 的三元组表 a.data 如图 5.10(a)所示，其中，row 为行号，col 为列号，value 为值。求该矩阵的稀疏矩阵 ***T***。

(1) 首先将 num[col]和 cpot[col]初始化，数组元素清零。

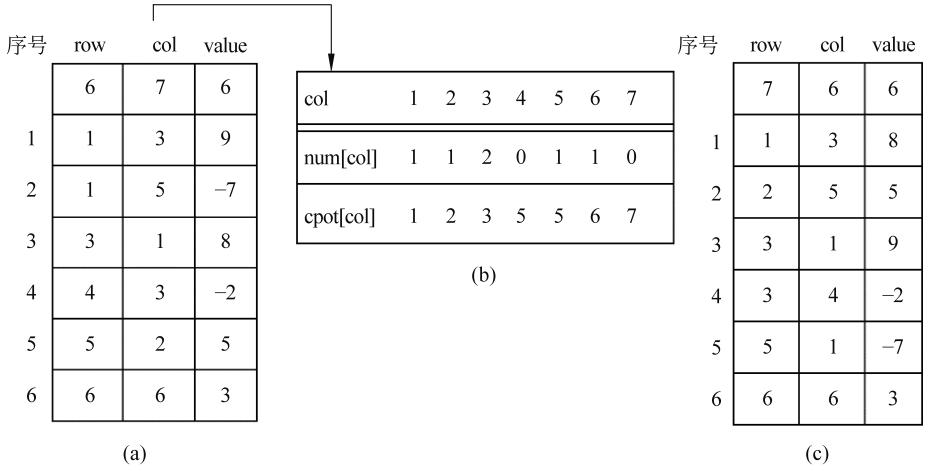


图 5.10　矩阵 ***M*** 的快速转置过程

(2) 扫描矩阵 ***M*** 的三元组表 a.data，计算 num[col]的值，如图 5.10(b)所示。

第一个元素(1,3,9)，列号为 3，num[3]＝1；

第二个元素(1,5,－7)，列号为 5，num[5]＝1；

第三个元素(3,1,8)，列号为 1，num[1]＝1；

第四个元素(4,3,－2)，列号为 3，num[3]＝2；

第五个元素(5,2,5)，列号为 2，num[2]＝1；

第六个元素(6,6,3)，列号为 6，num[6]＝1。

(3) 计算 cpot[col]的值，如图 5.10(b)所示。

cpot[1]＝1；

cpot[2]＝cpot[1]＋num[1]＝1＋1＝2；

cpot[3]＝cpot[2]＋num[2]＝2＋1＝3；

cpot[4]＝cpot[3]＋num[3]＝3＋2＝5；

cpot[5]＝cpot[4]＋num[4]＝5＋0＝5；

cpot[6]＝cpot[5]＋num[5]＝5＋1＝6；

cpot[7]＝cpot[6]＋num[6]＝6＋1＝7。

(4) 扫描矩阵 ***M*** 的三元组表 a.data，生成转置矩阵 ***T*** 的三元组表 b.data，如图 5.10(c)所示。

第一个元素(1,3,9)列号为 3，cpot[3]＝3，转置后的元素(3,1,9)是 b.data 中的第 3 个元素，并修改 cpot[3]＝4；

第二个元素(1,5,－7)，列号为 5，cpot[5]＝5，转置后的元素(5,1,－7)是 b.data 中的第 5 个元素，并修改 cpot[5]＝6；

第三个元素，(3,1,8)，列号为 1，cpot [1]＝1，转置后的元素(1,3,8)是 b.data 中的第 1

个元素,并修改 cpot[1]=2;

第四个元素(4,3,-2),列号为 3,cpot [3]=4,转置后的元素(3,4,-2)是 b.data 中的第 4 个元素,并修改 cpot[3]=5;

第五个元素(5,2,5),列号为 2,cpot [2]=2,转置后的元素(2,5,5)是 b.data 中的第 2 个元素,并修改 cpot[2]=3;

第六个元素(6,6,3),列号为 6,cpot [6]=6,转置后的元素(6,6,3)是 b.data 中的第 6 个元素,并修改 cpot[6]=7。

只需要扫描一遍矩阵 ***M*** 的三元组表 a.data,而且不需要元素的移动,即可生成转置矩阵 ***T*** 的三元组表 b.data。

4. 参考程序

定义文件中的主程序如下。

```
#include "Matrix.h"
#include <iostream>
using namespace std;

void create_Matrix(Matrix<int> &M)
{
    cout <<"请按照行优先顺序输入稀疏矩阵"
         <<M.num <<"个非零元素信息:\n";
    for (int k =1; k <=M.num; k++)
    {
        cout <<"第" <<k <<"个元素的行标、列标以及元素的值:";
        cin >>M.data[k].i >>M.data[k].j >>M.data[k].elem;
    }
    cout <<endl;
}

//采用三元组表存储表示,求稀疏矩阵 M 的转置矩阵 T
Matrix<int>FastTranspose_Matrix(Matrix<int> &M)
{
    Matrix<int> T(M.col, M.row, M.num);
    if (M.num ==0)
        return T;
    int * num =new int[M.col +1]();
    int * cpot =new int[M.col +1]();

    //计算 num[col]的值
    for (int t =1; t <=M.num; t++)
        num[M.data[t].j]++;
    //计算 cpot[col]的值
    cpot[1] =1;
    for (int col =2; col <=M.col; col++)
        cpot[col] =cpot[col -1] +num[col -1];
```

```
33.     //扫描矩阵 M 的三元组表 a.data,生成转置矩阵 T 的三元组表 b.data
34.     for (int p =1; p <=M.num; p++)
35.     {
36.         int q =cpot[M.data[p].j];
37.         T.data[q].i =M.data[p].j;
38.         T.data[q].j =M.data[p].i;
39.         T.data[q].elem =M.data[p].elem;
40.         cpot[q]++;
41.     }
42.
43.     delete[] num;
44.     delete[] cpot;
45.     return T;
46. }
47.
48. void print_Matrix(Matrix<int> &M)
49. {
50.     cout <<"共有" <<M.num <<"个非零元素:\n";
51.
52.     int t =1;
53.     cout <<"稀疏矩阵为:\n";
54.     for (int p =1; p <=M.row; p++)
55.     {
56.         for (int q =1; q <=M.col; q++)
57.             //打印非零元素
58.             if (M.data[t].i ==p && M.data[t].j ==q)
59.                 printf("%5d", M.data[t++].elem);
60.             else //打印零元素
61.                 printf("%5d", 0);
62.         cout <<endl;
63.     }
64.
65.     cout <<"\n 稀疏矩阵三元组顺序表为:\n";
66.     printf("%5c%5c%5c\n", 'i', 'j', 'v');
67.     for (int k =1; k <=M.num; k++)
68.         printf("%5d%5d%5d\n",
69.                M.data[k].i, M.data[k].j, M.data[k].elem);
70. }
71.
72. int main()
73. {
74.     int r, c, n;
75.     cout <<"输入稀疏矩阵行数与列数:";
76.     cin >>r >>c;
77.     cout <<"输入稀疏矩阵非零元素个数:";
```