第 5 章 数组与十字链表

数组中的元素一般同时属于多个线性表，故属于广义线性表。一般的高级程序设计语言都支持数组。但在高级程序设计语言中，重点是数组的使用，而我们这里重点是数组的内部实现，即高级程序设计语言或其他应用如何在计算机内部处理数组。其中主要问题是数组的存储方式与寻址。

当然，也可以将数组视为一种特殊的容器（数据结构），为其建立相应的对象结构（基本操作）。不过，对一维数组，该问题类似于线性表。对高维数组，这里也不做讨论，读者如有兴趣，可自行讨论。

十字链表多作为存储结构使用，其典型的应用是存储稀疏矩阵。由于其比较复杂，而且也代表了一类问题，故在这里单独介绍。

§5.1 数组

数组是一种十分常用的结构，大多数程序设计语言都直接支持数组类型。数组的基本操作主要是元素定位，所以本节的主要内容是讨论数组的存储映射方法。对于一些特殊类型的数组，我们将在下节中专门介绍。

§5.1.1 数组的定义与运算

数组是由一组类型相同的数据元素构成，每个数据元素称为一个数组元素（简称元素），每个元素受 n 个线性关系约束（n≥1），若它在第 1～第 n 个线性关系中的序号分别为 i_1、i_2……i_n， 则称它的**下标**为 i_1、i_2……i_n，若该数组的名为 A，则记下标为 i_1、i_2……i_n，的元素为 $A_{i_1 i_2 \ldots i_n}$，称该数组为 **n 维**数组。

另一方面，可借助线性表的概念递归地定义：

数组定义为一个元素可直接按序号寻址的线性表

$$A=(A_1, A_2, \ldots, A_m)$$

若 A_i 是简单元素（不是数组），则 A 是一维数组；若 A_i 是（k-1)维数组，则 A 是 k 维数组。这里，i=1,2,…,m. 而 k 是大于 0 的整数。

从此定义可直接看出，数组是从线性表的推广而来。

图 5-1 为一个 3 维数组的元素关系示意图。

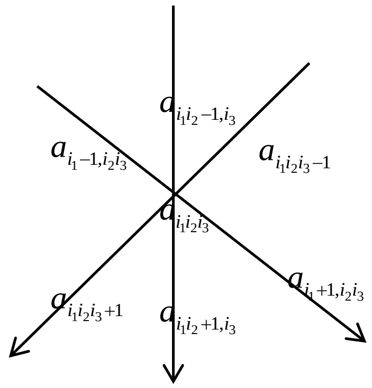

图 5-1 一个 3 维数组的元素关系示意

对一维数组的操作，也可以像线性表那样多种多样。对多维数组，由于是多个线性表的组合，所以情况会复杂些，此时，主要的操作通常是下列两种：

给定一组下标，读出相应的元素。

给定一组下标，修改相应的元素。

它们本质上只对应一种操作：寻址，即根据下标定位相应的元素，所以下面主要针对此问题讨论。

§5.1.2 数组的存储结构与寻址问题

数组是一种特殊的数据结构，一般要求元素的存储地址能根据它的下标（即逻辑关系）计算出来。所以，数组一般也只采用顺序存储结构。

讨论数组元素地址时，将数组的第 1 个元素的起始存储单元作为参考单元，参考单元的绝对地址称为该数组的首地址，数组其他元素的相对地址均相对于首元素的起始地址。设 i_1、i_2、…、i_n 为某 n 维数组中的一个元素的下标，则用 $Loc(i_1, i_2, \cdots, i_n)$表示此元素的相对地址。

对一维数组，与线性表类似，可使用顺序存储方式。但对多维数组，其已不属于线性结构的范畴，所以，不能直接使用顺序存储方式。但多维数组有其特殊性，具有线性结构的痕迹，它可唯一地转换为一维结构，反之亦然。从转换后的一维结构，可根据元素的存储位置推算出元素的逻辑关系（下标）。据此，**可以将多维数组映射为一维结构，然后使用顺序存储方式。**

§5.1.3 一维数组的存储与寻址

一维数组的每个元素只含一个下标，其实质上是线性表，存储方法与普通线性表的顺序存储方法完全相同，即将数组元素按它们的逻辑次序存储在一片连续区域内。设一

维数组为

$$A=(a_0, a_1, \cdots, a_{n-1})$$

则它的元素 a_i 的相对地址为

$$Loc(i)=i \cdot c$$

这里，c 表示每个元素占用的存储单元数目。

一般地，若数组下标范围为闭区间[l_1, h_1]内的整数，即

$$A=(a_{l_1}, a_{l_1+1}, \ldots, a_{h_1})$$

则元素 a_i 的相对地址为

$$Loc(i)=(i-l_1) \cdot c$$

§5.1.4 二维数组的存储

二维数组的每个元素只含 2 个下标，其已不是一般的线性表。

如果将二维数组的第 1 个下标理解为行号，第 2 个下标理解为列号，然后按行列次序排列各元素，则二维数组呈阵列形状。例如

$$A= \begin{matrix} a_{11} & a_{12} \ldots a_{1n} \\ a_{21} & a_{22} \ldots a_{2n} \\ \ldots\ldots & \\ a_{m1} & a_{m2} \ldots a_{mn} \end{matrix}$$

是一个行号为 1 到 m，列号为 1 到 n 的二维数组元素阵列。

对这类非线性结构的存储，需将多维关系映射为一维（线性）关系，即要确定多维到一维的映射。常用的映射方法有两种：**按行**与**按列**。

(一) 按行存储

按行存储的基本思想是：先行后列，即先存储行号较小的元素，行号相同者，先存储列号较小者。

例如，对上列的二维数组 A，按行存储的次序为

$$\underbrace{a_{11}\, a_{12} \cdots a_{1n}}_{\text{第 1 行元素}} \quad \underbrace{a_{21}\, a_{22} \cdots a_{2n}}_{\text{第 2 行元素}} \quad \cdots\cdots \quad \underbrace{a_{m1}\, a_{m2} \cdots a_{mn}}_{\text{第 m 行元素}}$$

设二维数组的行下标与列下标变化范围分别为闭区间[l_1, h_1]与[l_2, h_2]，按行存储，则它的任一元素 a_{ij} 的相对地址计算公式为

$$Loc(i, j) = ((h_2 - l_2+1) (i-l_1) + (j-l_2)) \cdot c$$

这里，c 为每个元素所占单元数目， $i\in[l_1, h_1]$，$j\in[l_2, h_2]$，且 i 与 j 为整数。该公式的正确性是容易证明的。

例 5-1.设某二维数组的两个下标的范围分别为[-1, 2]与[0, 1]，则它的元素按行存储的次序为（用 a_i 表示元素，每个元素占两个单元）：

相对地址	0	2	4	6	8	10	12	14
元素	$a_{-1,0}$	$a_{-1,1}$	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$	$a_{2,0}$	$a_{2,1}$

它的元素 $a_{-1,1}$ 的相对地址计算如下

$$Loc(-1,1) = ((1-0+1)(-1+1)+(1-0))*2 = 2$$

(二) 按列存储

按列存储的基本思想是：先列后行，即先存储列号较小者，列号相同者先存储行号较小者。例如，前述的二维数组 A 按列存储的次序为

$$\underbrace{a_{11}\ a_{21}\ \cdots\ a_{m1}}_{\text{第 1 列元素}}\quad \underbrace{a_{12}\ a_{22}\ \cdots\ a_{m2}}_{\text{第 2 列元素}}\quad \cdots\cdots\quad \underbrace{a_{1n}\ a_{2n}\ \cdots\ a_{mn}}_{\text{第 n 列元素}}$$

设二维数组的行下标与列下标变化范围分别为闭区间[l_1, h_1]与[l_2, h_2]，按列存储，每个元素占 c 个存储单元，则它的任一元素 a_{ij} 的相对地址计算公式为

$$Loc(i, j) = ((j-l_2)(h_1 - l_1+1) + (i-l_1)) \cdot c$$

该公式的正确性是容易证明的。

对 2 维数组，还可以有其他映射（2 维向 1 维的映射）方法，如按正对角线、反对角线等，这里就不去讨论了。

§5.1.5 多维数组的存储

下面将二维数组的结果推广到多维数组。对 n 维数组， 也一般采用两种存储映射方式，它们分别是二维数组的按行与按列存储的推广。

(一) 按行存储

这是一种“左”下标优先的存储方法，即第 1（最左）下标的下标值较小的元素较先存储，第 1 下标值相同者，按第 2 下标优先存储，对任意的 k>1，对第 1~(k-1)维相同者，先存储第 k 维中下标值较小者。

例 5-2 设某三维数组 a 的三个下标范围分别为

[2,4]，[0,1]，[1,3]

则它按行存储的次序为

a201 a202 a203 a211 a212 a213 （接下行）

a301 a302 a303 a311 a312 a313 （接下行）
a401 a402 a403 a411 a412 a413

设 n 维数组第 k 维的下标范围是$[l_k, h_k]$，记

$d_k = h_k - l_k + 1$，$j_k = i_k - l_k$，$k= 1,2, \cdots, n$，

显然，d_k 为第 k 维的体积（元素个数），设每个元素占 c 个单元，则下标为 $i_1, i_2, \ldots, i_n$ 的元素的相对地址的计算公式为

$Loc(i_1, i_2, \cdots, i_n) = c \cdot (j_1d_2d_3\cdots d_n + j_2d_3\cdots d_n + \cdots + j_{n-1}d_n + j_n)$

例如，三维数组的寻址公式为

$Loc(i_1, i_2, i_3) = c \cdot (j_1d_2d_3 + j_2d_3 + j_3)$

n 维数组寻址公式可应用数学归纳法证得。

(二) 按列存储

这是一种“右”下标（最后一维下标为最右）优先的存储映射方式。 先存储第 n 维下标值较小者，第 n 下标相同者，先存储第(n-1)维下标值较小者。即，对任意的 k<n，若第(k+1)~第 n 维下标相同，则先存储第 k 维中下标值较小者。

例如，对前面的三维数组 a（下标范围分别为[2,4]，[0,1]，[1,3]），它的按列存储的次序为

a201 a301 a401 a211 a311 a411 （接下行）
a202 a302 a402 a212 a312 a412 （接下行）
a203 a303 a403 a213 a313 a413

若引用前面的符号，则 n 维数组按列存储的寻址公式为

$Loc(i_1,i_2, \ldots ,i_n) = c \cdot (j_1 + d_1j_2 + d_1d_2j_3 + \ldots + d_1d_2 \ldots d_{n-1}j_n)$

例如，对三维数组，按列存储寻址公式为

$Loc(i_1,i_2,i_3) = c \cdot (j_1+d_1j_2 + d_1d_2j_3)$

§5.1.6 寻址公式的计算

下面考虑如何根据一组给定下标，求出对应的数组元素的地址的问题。这是数组的最重要的基本操作，它一般用在高级语言的实现中。

这里只考虑 n 维数组按行存储的寻址公式的计算。用秦九韶法变换按行存储公式中的主要部份：

$j_1d_2d_3\cdots d_n + j_2d_3\cdots d_n + \cdots + j_{n-1}d_n + j_n$

$= (j_1d_2+ j_2)d_3\cdots d_n + j_3d_4\cdots d_n + \cdots + j_{n-1}d_n + j_n$

$=((j_1d_2+ j_2)d_3+ j_3) d_4\cdots d_n + j_4d_5\cdots d_n + \cdots + j_{n-1}d_n + j_n$

$=\ldots\ldots$

$=(\ldots(j_1d_2+j_2)d_3+j_3)\ d_4+j_4)d_5+\cdots+j_{n-1})d_n+j_n$

$=(\ldots((0*d_1+j_1)d_2+j_2)d_3+j_3)\ d_4+j_4)d_5+\cdots+j_{n-1})d_n+j_n$

此式的值乘以元素占用的单元数 c 即为元素$(i_1i_2\cdots i_n)$的相对地址$(j_k=i_k-l_k)$。此式可按下法计算：

```
s=0;
k=1;
while (k<=n)
{
   s = s*dk+jk ;
   k = k+1;
}
```

下面考虑将该计算过程写成一个 C 函数。我们用三个一维数组 *l*[]、*h*[]、*i*[]分别表示 n 个下标的下界、上界及待求地址的元素的 n 个下标值。

```
long GetArrayElemAddress(long l[], long h[], long i[], long n, int c)
{
  long k, s;

  s=0;
  for (k=0; k<n; k++)    //注意，这里是从 0 号元素起存储内容
     s = s*(h[k] – l[k] + 1) + (i[k] – l[k]);
  return s;
}
```

至于按列存储公式的计算，与按行存储类似，留作练习。

§5.2 特殊数组*

这里介绍一些常用的特殊数组及存储方式。二维数组在形式上是矩阵（但矩阵元素可以是不同类型的！），元素类型一致的矩阵均可按二维数组处理。我们称元素分布具有一定规律的矩阵为特殊矩阵，而称零元素（或相同元素）居多数的矩阵为稀疏矩阵。这是两类特殊数组，本节先讨论特殊矩阵，而将稀疏矩阵放在下节讨论。

常见的特殊矩阵有对称矩阵与上/下三角矩阵，现分述之。

§ 5.2.1 对称矩阵

若 n 阶方阵元素满足

$a_{ij}=a_{ji}$　（i 与 j 为任意的在下标范围内的整数）

则称其为对称矩阵。

显然，对称矩阵的以主对角线为对称轴的对称元素两两相等，因此，它中有近 1/2 元素是相同的，所以在存储时，可只存储不同元素。 如只存储上三角或下三角元素（主对角线及其之下的元素为下三角元素，主对角线及其之上元素为上三角元素）。

例如，下面的数组表示中，虚线框内左下三角内元素为下三角元素。

$$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$$

存储上（或下）三角时，又分为按行和按列存储，故对称矩阵可有 4 种不同的存储方式。这里只介绍下三角的按行与按列存储，将上三角的存储映射方式的讨论（寻址公式的确定）留作练习。

1．按行存储下三角

对上面给出的 4 阶对称矩阵，它的下三角按行存储次序为

$a_{11}\ a_{21}\ a_{22}\ a_{31}\ a_{32}\ a_{33}\ a_{41}\ a_{42}\ a_{43}\ a_{44}$

这种存储方式的寻址公式为

$Loc(i, j) = ((i-l_1)(i-l_1+1)/2+(j-l_2))c$ …………当 $i\geq j$

$Loc(i, j) = Loc(j, i)$ ……………………当 $i<j$

2．按列存储下三角

对上面的 4 阶对称矩阵，它的下三角按列存储映射方法为：

$a_{11}\ a_{21}\ a_{31}\ a_{41}\ a_{22}\ a_{32}\ a_{42}\ a_{33}\ a_{43}\ a_{44}$

这种存储方式的寻址公式为

$Loc(i, j) = ((j-l_2)(2n-j+l_1+1)/2 + i- l_1+j- l_2)c$ ……当 $i\geq j$

$Loc(i, j) =Loc(j, i)$ ………当 $i<j$

这里，n 为矩阵行数，$n=h_1-l_1+1$。

§5.2.2 下/上三角矩阵

若矩阵主对角线上方/下方元素全为 0，则称该矩阵为下/上三角矩阵。

这类矩阵的存储方式完全同对称矩阵，但在寻址时，对下三角，i<j 时元素值为 0；对上三角，i>j 时元素值为 0，对 0 值元素无需寻址。

§5. 3 稀疏矩阵

§5.3.1 稀疏矩阵的逻辑表示

稀疏矩阵是零元素居多的矩阵，它在科学与工程计算中有着十分重要的应用，所以有必要专门讨论它的特殊性。

当稀疏矩阵的阶很高时，它中的零元素就会很多，如果使用二维数组的存储方法，势必存储大量重复元素（即 0 元素），造成存储浪费。 一个显然的解决办法是不存储零元素，这种存储方式称为压缩存储。

由于不存储 0 元素，元素的存储次序不再能代表它们的逻辑关系了，所以必须显式地指出每个元素逻辑次序。一种常用方法是，对每个非 0 元素，用如下形式的一个三元组表示：

（行号，列号，元素值）

各非 0 元素对应的三元组构成的集合就是相应的稀疏矩阵的逻辑表示。

例 5-3 稀疏矩阵

$$M = \begin{matrix} 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 0 & 0 \end{matrix}$$

的三元组集合为

M={(1,3,2), (2,3,3), (2,6,1), (3,1,1), (5,1,4), (5,3,5) }

对于稀疏矩阵的这种逻辑表示，有两种常用存储方式：三元组表法与十字链表法。本节介绍三元组表法，十字链表在下节介绍。

§ 5.3.2 三元组表法

(一) 存储方法

将稀疏矩阵视为由非 0 元素的形如（行号，列号， 元素值）的三元组构成的线性表，表中三元组按行序（或列序）排列，采用连续存储结构。这种非 0 元素三元组线性表的连续存储结构就称为稀疏矩阵的**三元组表存储法**。

例 5-4 上节例中的矩阵 M 的按行排列的顺序存储结构线性表（三元组表）如下表所示。

行号	列号	元素值
5	6	6
1	3	2
2	3	3
2	6	1
3	1	1
5	1	4
5	3	5

这里，为了操作方便，增设一个信息元组，形式为

（行数，列数，非 0 元素个数）

将它作为三元组表的第 1 个元素。

(二) 三元组对象描述

这种三元组表是一种具体的线性表顺序存储结构，所以，它应该是线性表顺序存储结构(TLinearListSqu)的派生类，只是元素类型 TElem 要具体定义：

```
struct TTriTuple    //定义三元组的元素类型
{
  long row, col; //行号、列号
  float val;    //值

  operator ==( TTriTuple &i1)    //重载恒等算符
  {
   return (row == i1.row && col==i1.col && val==i1.val);
     //两个三元组元素的行号、列号、值全对应相等时才算相等
```

```
    };

    TTriTuple& operator =( TTriTuple &i1)    //重载赋值算符
    {
      this->row = i1.row;
      this->col = i1.col;
      this->val = i1.val;
      return i1;
    };

}; // TTriTuple
```

这里，我们对恒等(==)和赋值(=)算符进行了重载，这是因为，我们在TLinearListSqu(在下面将作为三元组类型的父类)中要求元素类型 TElem(以可变类型出现)支持恒等与赋值运算。为了支持父类的输出操作(Print)，还需要对标准输出算符进行重载：

```
ostream& operator << (ostream& oo, TTriTuple &i1)
  {
    oo<<"("<<i1.row<<", "<<i1.col<<", "<<i1.val<<") ";
    return oo;
  };
```

下面是从线性表顺序存储结构派生出的三元组表类：

```
class TTriTupleArray : public TLinearListSqu<TTriTuple>
{
  long rows, cols;    //总行数、列数
  float theZero;    //具体的“0”元素值，可以不是数值 0

  long AddNew(long i, long j, float x);    //在三元组表中加入一个新元素(i,j,x)

  public:

  TTriTupleArray();
  TTriTupleArray(long mSize);
  float& Get(long i, long j);    //读出元素(i,j)的值
  void Set(long i, long j, float x); //将元素(i,j)的值置为 x
  long GetRowsCols(long &r, long &c); //返回最大行号和列号
  float Delete(long i, long j); //删除元素(i,j)
```

```
  int Trans1(TTriTupleArray &tu); //转置
  int Trans2(TTriTupleArray &tu); //转置
};
```

该类的初始化函数的实现如下：

```
TTriTupleArray::TTriTupleArray():TLinearListSqu<TTriTuple>()
{//初始化函数
  rows=0;
  cols=0;
  theZero=0;
};

TTriTupleArray::TTriTupleArray(long mSize)
:TLinearListSqu<TTriTuple>(mSize)    //先调用父类的构造函数，申请了 mSize 个空间
{
  rows=0;
  cols=0;
  theZero=0;
};
```

§5.3.3 三元组表的操作

这里介绍几种稀疏矩阵三元组表所特有的基本操作。

Get(i,j)----返回下标为（i, j）的元素的值的引用。

Set(i, j, x)----将下标为（i, j）的元素的值置为 x. 该操作的内部动作为：若该元素已在三元组表中存在（当前为非零元素），则该操作只是简单地将 x 的值赋予对应元素；若该元素是零元素，则先在三元组表中插入一个元素，然后将其值置为 x；若 x 的值为零，则由于我们规定不存储零元素，则该操作相当于将元素(i,j)从三元组中删除（若存在的话）。

上面两个基本操作是最基础的，有了它们，就可以象访问普通二维数组那样访问三元组数组了（直接使用下标访问元素）。其他操作可根据使用的方便性设置，例如，关于矩阵的转置、加法、乘法等，都可以设计为基本操作。

Trans1()和 Trans2()----将三元组所代表的矩阵转置。

下面给出几个重要的基本操作的实现。关于三元组表矩阵转置算法，将在下节讨论。

下面的算法均假定矩阵下标从 1 开始。

为了方便地动态获取三元组表矩阵的最大行号和列号，特设立 GetRowsCols(long &r, long &c)，它的实现按的源代码为：

```
long TTriTupleArray::GetRowsCols(long &r, long &c)
{//求三元组表代表的矩阵的最大行号和最大列号，结果分别存入 r 和 c;顺便返回非 0 元素个数
  long k;

  r=0; c=0;
  for (k=0; k<len; k++)
  {
    if (r<room[k].row) r =room[k].row;   //
    if (c<room[k].col) c =room[k].col;
  }
  return k; //返回非零元素个数
}
```

由于在三元组表中不保留零元素，所以，当一个元素由零变为非零时，需要在三元组表中增加新元素，该工作由下面的 AddNew 函数完成。但注意，若欲新增加的元素已存在，则该函数相当于单纯的赋值 Set(i, j, x).

```
long TTriTupleArray::AddNew(long i, long j, float x)
{
  long k, kk;

  k=len-1;
  while (k>=0) //从表的尾部起扫描，找到一个 i 行元素（或得知不存在 i 行元素）时跳出
  {
    if (i < room[k].row) k--;
    else break;
  }

  while (k>=0 && room[k].row==i) //在 i 行元素中找 j 列元素，找到(或得知不存在)时跳出
  {
    if (j < room[k].col) k--;
    else break;
  }

  if (k>0 && j==room[k].col )
  {//存在元素(i, j)，不能增加新元素,只是将其值改为 x
    room[k].val = x;
    return -k;
  }
```

```
  for (kk=len-1; kk>k; kk--) //后移元素，准备插入
      room[kk+1] = room[kk];
  room[k+1].row=i;   //插入元素(i,j,x)
  room[k+1].col=j;
  room[k+1].val = x;
  len++;
  if (i>rows) rows=i;
  if (j>cols) cols=j;
  return k+1; //返回表中元素个数
};
```

下面是具体的 Get 和 Set 函数的实现：

```
float& TTriTupleArray::Get(long i, long j)
{
  long k;

  k=0;
  while (k<len)
  {
    if (room[k].row==i)
      if (room[k].col==j) break;
    k++;
  }
  if (k==len) return theZero;
  return room[k].val;
}

void TTriTupleArray::Set(long i, long j, float x)
{
  long k;

  k=0;
  while (k<len)
  {
    if (room[k].row==i)
    {
      if (room[k].col==j) break;
```

```
      }
      k++;
    }

    if( k==len)
    {//表中无(i,j)，插入
      if (x!=theZero)
      AddNew(i, j, x);
    }
    else
    {//表中存在元素(i,j)
      if (x!=theZero) room[k].val = x; //x 非 0 时将 x 的值赋予(i,j)元素
      else Delete(i,j); //x 为 0 时删除元素(i,j)
    }
  }
```

§5.3.4 转置操作

对矩阵的转置，就是使 i 行 j 列元素与 j 行 i 列元素对换位置。若矩阵是用二维数组表示的，则转置操作是很简单的。设 a 是一个用二维数组表示的 m×n 矩阵，其转置的结果存于二维数组 b，它是一个 n×m 矩阵。具体的转置过程可描述如下：

```
int a[m][n], b[n][m];
…………
for (i=0; i<m; i++)
 for (j=0; j<n; j++)
   b[j][i]=a[i][j];
```

如果矩阵是用三元组表表示的，可以利用 Get 和 Set，则转置操作与上面的类似，基本形式为：

```
for (i=0; i<m; i++)
 for (j=0; j<n; j++)
   b.Set(j, i,   a.Get(i, j));
```

这个例子表明，三元组表设置了 Get 和 Set 操作，其所用方式就与二维数组相同了，完全屏蔽了三元组表的存储结构。

若为了提高程序效率，可直接实现转置（不使用 Get 和 Set），则转置的实现没有这样直接。下面就讨论给该问题。

分析转置操作，其主要是将每个元素的行号和列号互换。由于在三元组表中，元素是按行序（或列序）排列，所以，行号和列号互换后，还要调整元素位置，使其仍保持

行序（或列序）排列。实现这一点的一个显然的做法是：

- 将每个元素的行号和列号分别互换；
- 对三元组表排序，使其中元素按行序（或列序）排列。

此算法的时间复杂度为 O(n)+O(nlog(n))，这里，n 为三元组表中元素个数。O(nlog(n)) 是基于比较运算的排序算法的平均最好时间复杂度，在简单排序情况（如冒泡排序等），时间复杂度可上升为 $O(n^2)$。

如果要降低时间复杂度，一个显然的做法是免去排序操作，使在进行元素的行号和列号互换的过程中，顺便实现行序（列序）排列。也就是（假定将 a 转置结果存于 b）：

- 将 a 的每个元素从三元组表中取出，交换它们的行号值与列号值；
- 将所取出的三元组元素存入目标三元组表 b 中适当位置，使三元组表 b 保持行序/列序。

$$A=\begin{pmatrix}0&0&0&5&0&0\\3&0&0&0&0&0\\0&0&0&2&0&0\\0&0&0&0&0&0\\0&1&0&0&2&0\end{pmatrix}\xrightarrow{\text{转置}}B=\begin{pmatrix}0&3&0&0&0\\0&0&0&0&1\\0&0&0&0&0\\5&0&2&0&0\\0&0&0&0&2\\0&0&0&0&0\end{pmatrix}$$

三元组(行序)

行号	列号	元素值
1	4	5
2	1	3
3	4	2
5	2	1
5	5	2

转置

三元组(行序)

行号	列号	元素值
1	2	3
2	5	1
4	1	5
4	3	2
5	5	2

图 5-2 三元组转置

下面假定三元组中元素按行序排序，即原矩阵 a 与转置结果均按行序排列， 至于列序排列情况，处理方法类似。

注意，由于我们这里假定三元组表内元素按行序排列，且转置后元素的行列号互换了，所以，转置后的三元组表中，元素相当于按原三元组表中元素的列序排列。

这里给出一个稀疏矩阵 A 与它的三元组形式 a，以及它们的转置形式 B 与其三元组形式 b(图 5-2).

对三元组转置，一般有两种算法，下面分别介绍。

(一) 转置算法 I

该算法可概括为“直接取，顺序存”，即第 1 次从 a 中选取一个适当元素放置到 b 中的第 1 个位置，第 2 次从 a 中选取一个适当元素放到 b 中的第 2 个位置，……，如此进行，依次生成了 b 中的第 1、第 2，…元素。

由于转置后列号变行号，故转置后元素按原列号次序排列（以保持行序）。因此，该算法的实现，对“直接取”，是在 a 中按列号的大小取元素，即依次取第 1、第 2、…、最后一列元素。当某类列上有多个元素时（列号为 i 的元素有多个时），按它们的行号次序取。“顺序取”是将所取出的元素依次放到 b 中的最近空位处（即最后一个元素的下个位置）。其过程可描述如下：

```
pb=1;   //pb 为 b 中当前空位置中编号最小的位置的指示器
for （col=最小列号; col<=最大列号; col++）
{
  在 a 中按从头到尾的次序，查找有无列号为 col 的三元组;
  若有，则将其行列交换后依次存入 b 中 pb 所指位置;
  pb 加 1;
}
```

注意，由于我们已假定三元组表元素按行序排列，所以，当在上面的程序中，列号等于 col 的元素有多个时，它们中必然是行号较小者较先出现，因此，可以保证列号（在转置后的三元组表中是行号）相同时按行号（在转置后的三元组表中是列号）排列。

据此可写出完整的算法，见下面的程序。

```
int TTriTupleArray::Trans1(TTriTupleArray &tu)
{//将本类所对应的对象转置，结果存入 tu
  long pa, pb, col;
  long mCols, mRows;

  if (len<1) return 0;
  tu.ResizeRoom(len);    //调整 tu 的空间，使其等于本对象中的元素个数
  GetRowsCols(mRows, mCols);    //求最大行号和列号
  pb=0;
  for (col=1; col<=mCols; col++)
  {
```

```
    for (pa=0; pa<len; pa++)
      if (room[pa].col==col)
      {
        tu.room[pb].row=room[pa].col;
        tu.room[pb].col=room[pa].row;
        tu.room[pb].val=room[pa].val;
        pb++;
      }
  }

  tu.cols=mRows;
  tu.rows=mCols;
  tu.len = len;
  return len;
}
```

(二) 转置算法 II

该算法可概括为“顺序取，直接存”，即依次从 a 中第 1、第 2、 ……位置取元素，交换它们的行列位置后放置在 b 中适当位置，该过程可描述为：

```
for（pa=起点；pa<=终点；pa++）
{
 为 a 的 pa 元素确定它在 b 中应放位置 pb;
 将 a 的 pa 元素行列值交换后存入 b 中 pb 位置；
}
```

这里的关键问题是确定当前从 a 中取出的元素在 b 中应放位置 pb。

转置结果 b 中元素实质上是按它们在原矩阵中的列号次序排列的，原矩阵中第 1 列中第 1 个非 0 元素应放置在 b 中第 1 个位置，该列上其它元素应放置在 b 中第 2、第 3、……位置上，处理完原矩阵第 1 列上元素后，应接着按类似的方式依次从原矩阵中获取第 2、第 3，……列上元素，并依次放置在 b 中当前最小空闲位置处。因此， 若能已知原矩阵中每一列上第 1 个非 0 元素在 b 中应放置的位置，则其它元素的放置位置就可通过逐步递增方式获得。

为此，设一个一维数组 cpos[]，令

cpos[i] = 原矩阵第 i 列上第 1 个非 0 元素在 b 中应放置的位置

由于矩阵转置后，按原矩阵的列序存储，所以，若知道了 cpos[i]，则 i 列上的其他元素的位置都是依次相对于 cpos[i]的值，因此，可通过依次给 cpos[i]加一获得 i 列上其他各元素的位置值。在此基础上，上列程序可进一步细化为：

```
for (pa=起点; pa<=终点; pa++)
{
  col=a.room[pa].col;
  pb=cpos[col];
  b.room[pb].row=a.room[pa].col;
  b.room [pb].col=a.room [pa].row;
  b.room [pb].val=a.room [pa].val;
  cpos[col]++;   //使 cpos[col]为 col 列上的下一个非 0 元素在 b 中应放置的位置
}
```

在上面的程序中，语句“cpos[col]++”是关键，也是该算法的巧妙之处。有了针对每列的 cpos[col]，则该列上其他非 0 元素的位置是在对应的 cpos[]上累计得到的。程序中，我们是从头到尾扫描三元组表，所以，行号较小的元素较先遇到，因此，每列上的非 0 元素总是按行序遇到（但不一定连续遇到），否则，这种方法是不奏效的。

至此，问题变为如何求得 cpos 数组。 为了求得 cpos，我们引入另一个一维数组 cnum[]，令

cnum[i] = 原矩阵中第 i 列上非 0 元素个数

这样，cpos 可用下列递推公式求得：

cpos[1]=1

cpos[i]=cpos[i-1]+cnum[i-1]   i>=2

其对应的程序实现为：

```
cpos[1]=1;
for(col=2; col<=最大列号; col++)
  cpos[col]=cpos[col-1]+cnum[col-1];
```

例如，对图 5-2 所示稀疏矩阵 A，对应的 cpos[]和 cnum[]如下：

i	1	2	3	4	5	6
Cnum[i]	1	1	0	2	1	0
cpos[i]	1	2	3	3	5	6

注意，对没有非 0 元素的列，其 cpos[]值不会在转置时被所用，故其值可为任意。但是，为了能递推计算 cpos[]，没有非 0 元素的列的 cpos 也按计算规则置值了。如上面的列 3 和列 6.

而 cnum 可用扫描分档计数的方法求得：

```
for(col=1; col<=最大列号; col++)   cnum[col]=0;   //计数器清 0
for(pa=0; pa<非 0 元素个数; pa++)
  cnum[a.room[pa].col]++;          //计数
```

综合上列结果，就可得到完整程序:

```
int TTriTupleArray::Trans2(TTriTupleArray &tu)
```

```
{//将本类所对应的对象转置，结果存入 tu
  long pa, pb, col;
  long mCols, mRows;

  if (len<1) return 0;
  tu.ResizeRoom(len);   //调整 tu 的大小
  GetRowsCols(mRows, mCols);   //求最大行号与列号

  long *cnum, *cpos;
  cnum=new long[mCols+1];
  cpos=new long[mCols+1];

  pb=0;
  for (col=1; col<=mCols; col++) cnum[col]=0;
  for (pa=0; pa<len; pa++) cnum[room[pa].col]++;

  cpos[1]=1;
  for (col=2; col<=mCols; col++)
     cpos[col]=cpos[col-1]+cnum[col-1];

  for (pa=0; pa<len; pa++)
  {
   col = room[pa].col;
   pb=cpos[col];
   tu.room[pb].row=room[pa].col;
   tu.room[pb].col=room[pa].row;
   tu.room[pb].val=room[pa].val;
   cpos[col]++;
  }

  tu.cols=mRows;
  tu.rows=mCols;
  tu.len = len;

  delete[] cnum;
  delete[] cpos;
  return len;
}
```

该问题解决过程，是个典型的逐步求精例子。

§5.4 十字链表

下面介绍一种特殊的链表——十字链表，它常用于表示稀疏矩阵，可视作稀疏矩阵的一种链式表示，因此，这里以稀疏矩阵为背景介绍十字链表。不过，十字链表的应用远不止稀疏矩阵，一切具有正交关系的结构，都可用十字链表存储。

§5.4.1 存储方式

(a)稀疏矩阵中每个非 0 元素对应一个十字链表结点，每个结点的结构为：

<table>
<tr><td>row</td><td>col</td><td>val</td></tr>
<tr><td colspan="2">down</td><td>right</td></tr>
</table>

其中各字段的含意为：
row——元素在稀疏矩阵中的行号
col——元素在稀疏矩阵中的列号
val——元素值
down——指向同列中下一个非 0 元素结点
right——指向同行中下一个非 0 元素结点

(b)每行/列设一个表头结点（结构同元素结点），以 down/right 为链构成循环链表，即第 i 列头结点的 down 指向该列上第 1 个非 0 元素，第 i 行头结点的 right 指向该行第 1 个非 0 元素。第 i 列/行上最后一个结点的 down/right 指向该列/行的头结点。若某列/行中无非 0 元素，则令它的头结点 down/right 域指向自己。

(c)设一个总头结点（结构同元素结点），令总头结点和各个列/行头结点用 val 字段，按列/行序构成一个循环单链表。

(d)可令总头结点的 row，col 与 val 分别表示矩阵的最大行号、列号与非 0 元素个数，而 down/right 指向第 1 列/行的头结点。该总头结点可作为整个十字链表的代表。

(e)由于行与列的头结点分别使用 right 域与 down 域(不同时使用)，故第 i 列与第 i 行头结点可合用同一个头结点（对所有可能的 i），以节省存储空间。

(f)有时，为了快速访问行/列头结点，设置一个一维数组 headNodes[]，使 headNodes[i] 指向 i 行/列的头结点。但这并不是必须的，因为各行/列的头结点已形成了一个循环单链表，故若已知十字链表总头结点，即可搜索到任一头结点。

例 5-5 设有一个如下形式的矩阵，它所对应的十字链表如图 5-3 所示。

$$A = \begin{pmatrix} 0 & 0 & 5 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 2 & 0 & 4 & 0 & 0 \\ 0 & 1 & 0 & 2 & 0 \end{pmatrix}$$

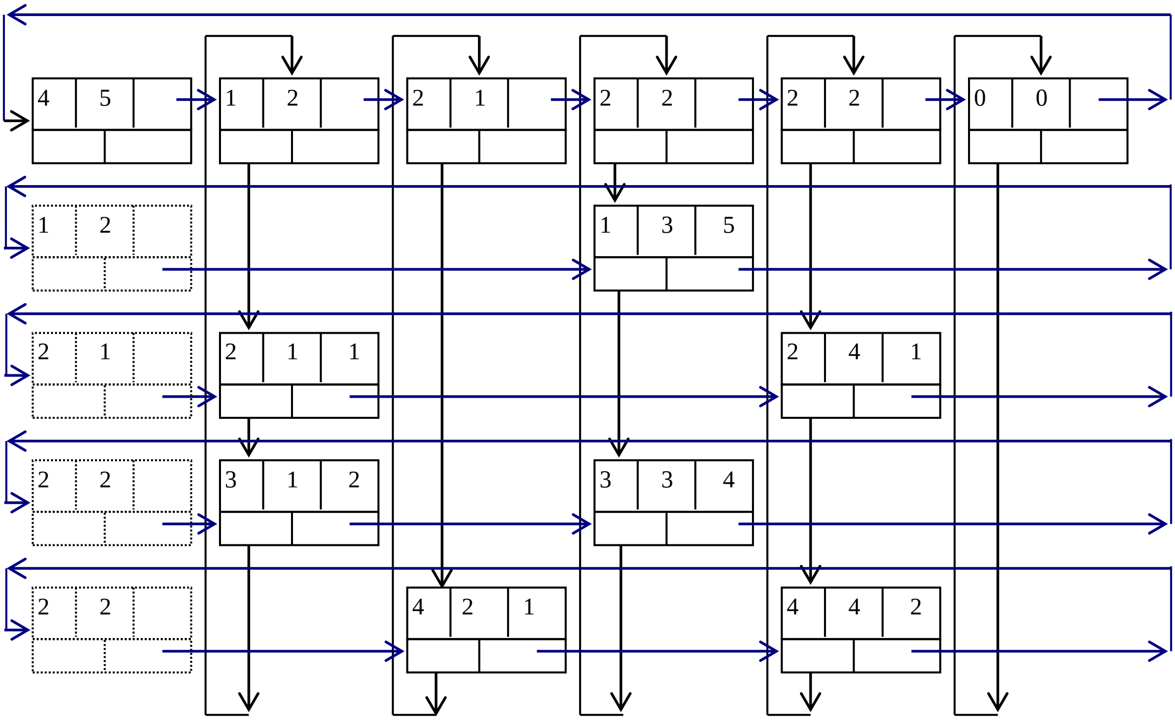

图 5-3 十字链表示意图

§ 5.4.2 十字链表对象

下面将十字链表作为一个对象，给出它的 C++描述。

首先，定义十字链表结点 TCrossNode，为了兼容比较运算、赋值运算和标准输出运算等，还定义了相应的运算符重载。

```
template <class TElem>
struct TCrossNode    //十字链表结点类型
{
```

```
    long row, col; //行号、列号
    union {//让 val 和 next 共享一块存储空间，即一块空间，两个名字
            TElem val;    //元素值
            TCrossNode *next;    //用于将头结点链为链表
          };
    TCrossNode *down, *right; //列/行指针

     operator ==( TCrossNode &oo) //重定义恒等运算
     {
       return (row == oo.row && col==oo.col && val==oo.val);
     };

     TCrossNode& operator =( TCrossNode &oo) //重定义赋值运算
     {
       this->row = oo.row;
       this->col = oo.col;
       this->val = oo.val;
       return oo;
     };
};
```

我们将十字链表定义为包含指向十字链表总头结点的指针的对象。

```
template <class TElem>
class TCrossLink //十字链表对象
{
     TCrossNode<TElem> *head; //指向总头结点的指针
     long rows, cols;    //总行数与总列数
     int theZero;    //"0"元素的值

     void ReleaseAll(void);    //释放所有结点
     TCrossNode<TElem> *Insert(TCrossNode<TElem> *pNode);
     int Insert(TElem& x, long i, long j);
     TCrossNode<TElem> * Delete(long i, long j);

  public:
     long len;

     TCrossLink();
```

```
        TCrossLink(long rows, long cols);
        ~TCrossLink();

        TCrossNode<TElem>* Init(long rows, long cols);

        TElem& Get(long i, long j);
        TCrossNode<TElem>* GetNode(long i, long j);
        TCrossNode<TElem> * Set(long i, long j, TElem& x);

        TCrossNode<TElem>* TCrossLink::GetHead(long k);

        int Print();

    };
```

head----指向十字链表总头结点的指针。根据该指针，可访问到十字链表中各元素。

len----十字链表中元素结点的个数（不含各种头结点），对稀疏矩阵，是非 0 元素的个数。

rows, cols----矩阵的最大行号和列号。

theZero----存储 0 值的变量。主要为了按引用方式返回 0 元素值使用。

Init(long rows, long cols)----初始化操作。用于创建一个具有 rows 行和 cols 列的空的十字链表。构造函数基本上是直接调用 Init()实现的。

Insert(TCrossNode<TElem> *pNode)：将 pNode 作为 i 行 j 列元素结点插入到十字链表。i 与 j 的值在 pNode 中。该函数主要用于 Set 函数。当 Set 函数为一个不存在的结点（即 0 元素）置值时，调用该函数插入一个结点。

Delete(long i, long j)---- 将 i 行 j 列元素结点删除。该函数主要用于 Set 函数。当 Set 函数为一个非 0 元素置 0 值时，调用该函数将对应的结点删除。

GetNode(long i, long j)----返回 i 行 j 列元素结点的指针。若该元素不存在，则返回空。

Get(long i, long j)----返回 i 行 j 列元素的值的引用。若该元素结点不存在（0 元素），则返回 0 元素值。

Set(long i, long j, TElem& x)---- 将 i 行 j 列元素的值置为 x。若 i 行 j 列元素原为 0 元素，则该函数的执行将在十字链表中插入一个新结点；若 x 值为 0，且该结点存在，则该函数的执行将删除该结点。

GetHead(long k)----返回行/列号为 k 的行/列头链表上的结点的指针。

这里只是定义了几个很基本的操作，为了使用方便，还可定义其他更高级的操作，如查找类操作。

§ 5.4.3 基本操作的实现

(一) 初始化操作

该函数生成一个具有 rows1 行和 cols1 列的空的十字链表。空的十字链表只含各行/列的头结点和总头结点。由于序号相同的行与列共享头结点，故总共生成 1+Max(rows1, cols1)个结点，每个结点以 next 链接，形成以总头结点为头的循环链表，其形式如图 5-4 所示。

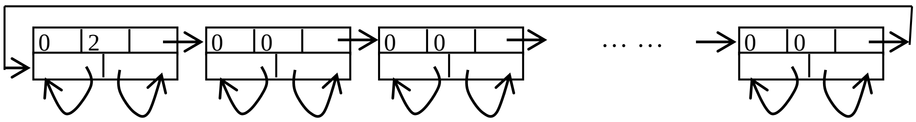

图 5-4 十字链表

算法采用插入方式。先建立总头结点，使其 next 指向自己，形成具有一个结点的循环链表，然后逐步插入，每次都是插入到总头结点的后面（其他结点的最前面）。

```
template <class TElem>
TCrossNode<TElem>* TCrossLink<TElem>::Init(long rows1, long cols1)
{
  TCrossNode<TElem> *p, *h;
  long rc;

  rc=rows1;
  if (rc<cols1) rc=cols1;    //令 rc 为 rows1 与 cols1 中最大者

  if (head!=NULL) ReleaseAll(); //若原表中有结点，则先释放其

  h= new(nothrow) TCrossNode<TElem>; //申请一个新结点，做为总头结点
  if (h==NULL) throw TExcepComm(3);

  h->next = h;
  h->row=rows1;
  h->col=cols1;

  for (long i=0; i<rc; i++)
  {//每次生成一个新结点并插入在总头结点后面
    p = new TCrossNode<TElem>;
    p->row=p->col=0; //给新结点置值
```

```
    p->down = p;
    p->right = p;

    p->next =h->next; //将 p 插入在总头结点后面
    h->next = p;
  }

  rows=rows1;
  cols=cols1;
  head = h;
  return h;
}
```

(二) 插入操作

如果将十字链表仅仅作为稀疏矩阵，则插入操作属于内部操作，它将行号和列号已知的结点插入到十字链表中指定位置。因每个结点同时属于两个链表（行链表与列链表），所以插入一个结点时，应分别插入到这两个链表中。有两个插入版本，一个是将存在的链结点插入，另一个是已知元素值进行插入，它通过调用前者实现。

```
template <class TElem>
TCrossNode<TElem> *TCrossLink<TElem>::
Insert(TCrossNode<TElem> *pNode)
{//将 pNode 插入到十字链表中，pNode 中包含着行号和列号
  long i, j;
  TCrossNode<TElem> *p, *p0;

  i=pNode->row;
  j=pNode->col;
  p0 = GetHead(i);    //得到行 i 的头结点。然后在行 i 上插入 pNode
  if (p0==NULL) throw TExcepComm(1);
  p=p0;
  while (p->right!=p0 && p->right->col<j)    //在行 i 上寻找插入位置 p
     p = p->right;
  pNode->right = p->right; //在 p 后插入 pNode
  p->right = pNode;

  p0 = GetHead(j); //得到列 j 的头结点。然后在列 j 上插入 pNode
```

```
  if (p0==NULL) throw TExcepComm(1);
  p=p0;
  while (p->down!=p0 && p->down->row<i)   //在列 j 上寻找插入位置 p
      p = p->down;
  pNode->down = p->down; //在 p 后插入 pNode
  p->down = pNode;

  len++;
  return head;
}

template <class TElem>
TCrossNode<TElem> *Insert(TElem& x, long i, long j)
{//插入一个以 x 为元素值的结点，行号和列号分别为 i 和 j
  TCrossNode<TElem> *p;

  p = new(nothrow) TCrossNode<TElem>; //申请一个结点
  if (p==NULL) throw TExcepComm(3);

  p->row=i; //给结点置值
  p->col=j;
  p->val=x;
  Insert(p); //调用另一版本的 Insert 将 p 插入
  return p;
}
```

(三) 删除操作

与插入操作类似，如果将十字链表仅仅作为稀疏矩阵，则删除操作也属于内部操作，它将行号和列号已知的结点从十字链表中删除。因每个结点同时属于两个链表（行链表与列链表），所以删除一个结点时，应分别从这两个链表中删除。

删除操作的实现与插入十分类似，具体实现留做练习。

(四) Get 类操作

Get 类操作实现按数组方式访问十字链表，即已知行号和列号求出元素值。有两个版本，第一个是已知行号和列号求对应元素结点的指针，第二个是是已知行号和列号求对应元素的值。后者可在前者基础上实现。

实现方法是搜索给定行（或列）对应的链表，查找列号（或行号）为给定列号（或

行号）的结点。

```
template <class TElem>
TCrossNode<TElem>* TCrossLink<TElem>::GetNode(long i, long j)
{
  TCrossNode<TElem> *p, *p0;

  p0 = GetHead(i);
  if (p0==NULL) return NULL;
  p=p0;
  while (p->right!=p0 && p->right->col<j)
     p = p->right;
  if (p->right->col==j) return p->right;
  else return NULL;
}

template <class TElem>
TElem& TCrossLink<TElem>::Get(long i, long j)
{
  TCrossNode<TElem> *p;
  p= GetNode(i, j);
  if (p==NULL) return (TElem)theZero;
  else return p->val;
}
```

(五) Set 类操作

Set 类操作实现按数组方式给元素置值，即已知行号和列号，给对应的元素置值。

与 Get 类似，需要先按给定的行号和列号查找对应的元素。查找到时，直接赋值即可，但当赋 0 值时应删除对应的结点。查不到时，要调用 Insert 新插入一个结点。

```
template <class TElem>
TCrossNode<TElem> * TCrossLink<TElem>::Set(long i, long j, TElem& x)
{
  TCrossNode<TElem> *p, *p0;

  if (x==0)
  {
    p=Delete(i, j);
    if (p!=NULL)
```

```
    {delete p;
     return head;
    }
    else throw TExcepComm(1);
   }

   p0 = GetHead(i);
   if (p0==NULL) throw TExcepComm(1);
   p=p0;
   while (p->right!=p0 && p->right->col<j)
       p = p->right;
   if (p->right->col==j) p->right->val=x;
   else
   {
    p = new TCrossNode<TElem>;
    p->row = i;
    p->col = j;
    p->val = x;
    Insert(p);
   }
   return p;
  }
```

(六) 十字链表相加法*

设有两个 m×n 矩阵 A 与 B，它们均按十字链表存储。现考虑操作 A←A+B，即将 B 加到 A 上，两矩阵相加，就是对应元素相加：$a_{ij}+b_{ij}$。

在我们上面给出的类 TCrossLink 的基础上，该操作很容易实现：

```
for (i=1; i<=B.rows; i++)
{
 for (j=1; j<=B.cols; j++)
    A.Set(i, j, B.Get(i, j)+ A.Get(i, j));
}
```

下面考虑不借助 Get 和 Set 进行加法。

我们可以采用逐行相加的方法，即逐次将 B 的各行加到 A 上， 实现过程可描述为（伪码）：

```
for (i=1; i<=m; i++)
{
 pHB=B.GetHead(i);   //求 B 中第 i 行头结点
 将 B 的第 i 行加到 A 的第 i 行上;
}
```

所以，问题就转化成了如何“将 B 的第 i 行加到 A 的第 i 行上”。该项操作的实现与多项式加法十分类似，其过程可描述为：

```
pHA = A.GetHead(i); //求得 A 的第 i 行头结点
pA=pHA->right;   //pA 指向 A 中 i 行上当前待处理的结点
pB=pHB->right;   //pB 指向 B 中 i 行上当前待处理的结点
pA0=pHA;
pB0=pHB;   //pA0 与 pB0 分别为 pA 与 pB 的前趋
while (pB->right!=pHB)   //处理 B 的 i 行上各结点
{
 if (pA->col < pB→col)
 {
  pA0=pA;
  pA=pA→right;   //pA 后移一步
 }
 else
 if (pA->col > pB→col)
 {
 从 pB 复制一结点 p；
 将 p 插入到 A 的 i 行链表中 pA 之前，即 pA0 之后；
 将 p 插入到 A 的  pB→col 列链表中适当位置；
 pB0=pB;
 pB=pB->right;
}
else   //pB->col==pA->col
{
 pA->val = pA->val + pB->val;
 if (pA->val !=0 )
 {
  pA0=pA;
  pA=pA→right;
  pB0=pB;
  pB=pB→right;
```

```
 } //pA 与 pB 分别后移一步
 else
 {
  将 pA 从十字链表中摘除;
  释放 pA;
  pA=pA0->right; //令 pA 指向被删结点的下一结点
  pB=pB→right;   //pB 后移一步
 }
}
} //while
```

上列算法描述中，尚有一些操作需细化。下面分别讨论：

从 pB 复制一结点 p：

```
p=new TCrossNode;
p->row = pB->row;
p->col = pB->col;
p->val = pB->val;
```

将 p 插入到 A 的 i 行链表中 pA 之前(pA0 之后)

```
p->right = pA0->ght;
pA0->right = p;
```

将 p 插入到 A 的 pB→col 列链表适当位置：

```
q0 =q = A.GetHead(pB->col); //求得 A 的 pB->col 列的头结点
while (q->down != q0 && q->down→row < p->row)
  q=q->down;   //在 pB→col 列链表中查找插入位置 q
p->down=q->down;
q->down=p; //将 p 插入到 q 之后
```

将 pA 从十字链表中摘除：由于 pA 分属于两个线性链表，故应将它分别从行链表和列链表中摘除。这分别需要持有 pA 在两个链表中的前趋，此时，pA 在行链表中的前趋是已知的（pA0），但在列链表中的前趋需查找才能获得。

```
pA0->right=pA->right;   //将 pA 从行链表中删除
qo = q = A.GotHead(pA->col); //求得 pA 所在列的头结点
while (q->down!=qo && q->down->row<pA->row)
  q=q->down; //查找 pA 的前趋
q->down=pA->down;   //将 pA 从列链表中摘除
free(pA);
```

至于完整的程序，留作练习。

本章小结

本章重点介绍了数组和十字链表。数组是一种广义线性表，即它是从线性表演变而来，但已不属于线性结构了。一个 n 维数组实质上是 n 个线性表的组合，其每一维都是一个线性表。每个元素受 n 个线性关系约束，所以，每个元素由 n 个下标定位，这是数组的基本逻辑特性。数组一般采用顺序存储结构，故存储多维数组时，应先将其确定地转换为一维结构。由于数组的固有特性，这种转换方式总是存在的，例如，按“行”转换（存储）和按“列”转换（存储）。相应的转化公式是关键。

十字链表实质上是一种链式存储结构。每个元素有两个链，用于建立正交关系。所以，十字链表一般用于存储正交关系，即类似于矩阵的结构。稀疏矩阵是零元素居多的矩阵，适合按十字链表存储。针对稀疏矩阵的十字链表存储，设置了十字链表类，其中的 Get(i,j)和 Set(i,j,x)操作使得十字链表下的稀疏矩阵的使用如同二维数组一样方便。

习 题

1 分别证明 n 维数组的按行与按列存储的寻址公式.

2. 设 4 维数组的 4 个下标的范围分别为[-1,0]，[1,2]，[1,3]，[-2,-1]，请分别按行序和按列序列出各元素。

3 对上题，分别计算元素 $a_{-1,2,2-2}$ 的按行存储与按列存储的相对地址（设每个元素占 2 个单元）

4. 写出 n 维数组按列存储的寻址公式的计算函数.

5 设 a 与 b 分别是 m×n 与 n×m 矩阵（二维数组）， 它们均按行序存储在一维结构中，请分别用两种方法写出计算 a×b 的程序。 这两种方法是：①采用二维数组的寻址公式访问元素；②不采用寻址公式访问元素，而直接推算要访问的元素的存储位置。

6. 分别推导出上三角矩阵按行与按列存储的寻址公式

7. 设 a 与 b 是 n×n 的对称矩阵，采用下三角压缩存储（只存储下三角），请分别按下列两种要求写出计算 a×b 的程序：①利用下三角按行存储的寻址公式访问元素；②不利用寻址公式，直接推出要访问的元素的存储位置。

8. 设对称矩阵按行序存储下三角(存储在一维结构中)， 请写一个程序，根据任一元素的存储位置（相对位置）求出它所对应的行号与列号。

9. 设 a 与 b 是两个用三元组表存储（连续结构）的稀疏矩阵，请写出将 b 加到 a 上的程序。

10. 设 a 与 b 是两个用三元组表存储（连续结构）的稀疏矩阵，请写出 a×b 的程序.

11. 写一个将用十字链表存储的稀疏矩阵转置的程序.

12. 编写十字链表的删除操作的程序。