第五章 数组与广义表

刘杰
人工智能学院

南开大学 Nankai University

目录

数组的定义及特点

数组是由$n(n>1)$个具有相同数据类型的数据元素$a_1, a_2, \dots, a_n$组成的有序序列，且该序列必须存储在一块地址连续的存储单元中。

特点：

- 数组中的数据元素具有相同数据类型。
- 数组是一种随机存取结构，给定一组下标，就可以访问与其对应的数据元素。
- 数组中的数据元素个数是固定的。因此，在数组上主要进行存取和修改元素值的操作。

数组的定义及特点

二维数组示例：

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

每一行可以视为一个数据元素，所以二维数组也属于线性表。

$$A_{m\times n}=<(a_{11},a_{12},\dots,a_{1n}),..,(a_{m1},a_{m2},\dots,a_{mn})>$$

数组的抽象数据类型

ADT Array{

数据对象：$j_i=0,1,\dots,b_{i-1}, i=1,2,\dots,n$；

$D=\{a_{j_1j_2\dots j_i\dots j_n} | n(>0)$称为数组的维数，$b_i$是数组第$i$维的长度，$j_i$是数组元素第$i$维的下标，$a_{j_1j_2\dots j_i\dots j_n}\in ElemSet$为对应下标的数据元素$\}$

数据关系：$R=\{R_1,R_2,\dots,R_n\}$

$R_i=\{<a_{j_1j_2\dots j_i\dots j_n}, a_{j_1j_2\dots j_i+1\dots j_n}>|0\le j_k\le b_k-1, 1\le k\le n$且$k\ne i$，$0\le j_i\le b_i-2, a_{j_1j_2\dots j_i\dots j_n}, a_{j_1j_2\dots j_i+1\dots j_n}\in D\}$

基本操作：……

}

n维数组中有$b_1\times b_2\times\dots\times b_n$个数据元素

高维数组的次序约定问题

以二维数组为例讨论。将二维数组看成是一个定长的线性表，其每个元素又是一个定长的线性表。

设二维数组$A=(a_{ij})_{m\times n}$，则

$A=(\alpha_1, \alpha_2, \dots, \alpha_p)$ $(p=m$或$n)$

其中每个数据元素α_j是一个列向量(线性表)：

$$\alpha_j=(a_{1j}, a_{2j}, \dots, a_{mj})^T \quad 1\le j\le n$$

或是一个行向量：

$$\alpha_i=(a_{i1}, a_{i2}, \dots, a_{in}) \quad 1\le i\le m$$

高维数组的次序约定问题

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad A=\begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} \cdots \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix}$$

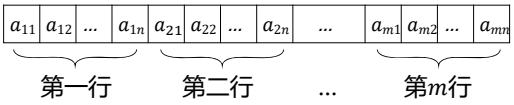
(c) 行向量的一维数组形式　　(b) 列向量的一维数组形式

计算机的内存结构是一维(线性)地址结构，对于多维数组，将其存放(映射)到内存一维结构时，存在次序约定问题。即必须按某种次序将数组元素排成一列序列，然后将这个线性序列存放到内存中。

数组的顺序表示和实现

行优先顺序(Row Major Order)：将数组元素按行排列，第$i+1$个行向量紧接在第i个行向量后面。对二维数组，按行优先顺序存储的线性序列为：

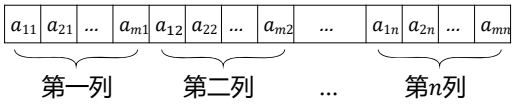
$$a_{11}, a_{12}, \dots, a_{1n}, a_{21}, a_{22}, \dots a_{2n}, \dots, a_{m1}, a_{m2}, \dots, a_{mn}$$



数组的顺序表示和实现

列优先顺序(Column Major Order)：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，对二维数组，按列优先顺序存储的线性序列为：

$$a_{11}, a_{21}, \dots, a_{m1}, a_{12}, a_{22}, \dots, a_{n2}, \dots, a_{1n}, a_{2n}, \dots, a_{mn}$$



数组元素的存储地址计算

设有二维数组$A=(a_{ij})_{m\times n}$，若每个元素占用的存储单元数为l(个)，下标从1开始，LOC$[a_{11}]$表示元素a_{11}的首地址，即数组的首地址。以“行优先顺序”存储为例：

(1)第1行中的每个元素对应的(首)地址：

$$LOC[a_{1j}]=LOC[a_{11}]+(j-1)\times l \quad j=1,2,\dots,n$$

(2)第2行中的每个元素对应的(首)地址：

$$LOC[a_{2j}]=LOC[a_{11}]+nl+(j-1)\times l \quad j=1,2,\dots,n$$

(3) 第m行中的每个元素对应的(首)地址:

$$LOC[a_{mj}]=LOC[a_{11}]+(m-1)\times nl+(j-1)\times l \quad j=1,2,\dots,n$$

数组元素的存储地址计算

由此可知，二维数组中任一元素a_{ij}的(首)地址:

$$LOC[a_{ij}]=LOC[a_{11}]+[(i-1)\times n+(j-1)]\times l \quad ,i=1,2,\dots,m \quad j=1,2,\dots,n$$

对三维数组$A=(a_{ijk})_{m\times n\times p}$，若每个元素占用的存储单元数为$l$(个)，$LOC[a_{111}]$表示元素$a_{111}$的首地址，即数组的首地址，“行优先顺序”存储,三维数组中任一元素a_{ijk}的(首)地址:

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

数组元素的存储地址计算

推而广之，在同样存储方式下，

n维数组中任一元素$a_{j_1j_2\dots j_n}$的(首)地址是:

$$LOC[a_{j_1j_2\dots j_n}]=LOC[a_{11\dots1}]+[(b_2\times\dots\times b_n)\times(j_1-1)+(b_3\times\dots\times b_n)\times(j_2-1)+\dots+b_n\times(j_{n-1}-1)+(j_n-1)]\times l$$

其中$b_2,\dots b_i,\dots,b_n$为第i维的长度。

以列优先的存储计算同理。

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，在高级语言编程时，通常将一个矩阵描述为一个二维数组。这样，可以对其元素进行随机存取，各种矩阵运算也非常简单。

对于高阶矩阵，若其中非零元素呈某种规律分布或者矩阵中有大量的零元素，若仍然用常规方法存储，可能存储重复的非零元素或零元素，将造成存储空间的大量浪费。对这类矩阵进行压缩存储：

- 多个相同的非零元素只分配一个存储空间；
- 零元素不分配空间。

对称矩阵的压缩存储

若一个n阶方阵 $A=(a_{ij})_{n\times n}$ 中的元素满足性质：

$$a_{ij}=a_{ji}\quad 1\le i,j\le n且i\ne j$$

则称A为对称矩阵。

$$A=\begin{pmatrix}1&5&1&3&7\\5&0&8&0&0\\1&8&9&2&6\\3&0&2&5&1\\7&0&6&1&3\end{pmatrix}\qquad A=\begin{pmatrix}a_{11}&&&&\\a_{21}&a_{22}&&&\\a_{31}&a_{32}&a_{33}&&\\\cdots&\cdots&\cdots&\cdots&\\a_{n1}&a_{n2}&\cdots&&a_{nn}\end{pmatrix}$$

对称矩阵示例

对称矩阵的压缩存储

对称矩阵中的元素关于主对角线对称，因此，让每一对对称元素a_{ij}和$a_{ji}(i\ne j)$分配一个存储空间，则n^2个元素压缩存储到$n(n+1)/2$个存储空间，能节约近一半的存储空间。

不失一般性，假设按“行优先顺序”存储下三角形(包括对角线)中的元素。

设用一维数组(向量)$sa[0,1\ldots n(n+1)/2-1]$存储n阶对称矩阵，为了便于访问，必须找出矩阵A中的元素的下标值(i,j)和向量$sa[k]$的下标值k之间的对应关系。

对称矩阵的压缩存储

(1)若$i\ge j$：a_{ij}在下三角形中，直接保存在sa中。a_{ij}之前的$i-1$行共有元素个数：

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

而在第i行上，a_{ij}之前恰有$j-1$个元素，因此，元素a_{ij}保存在向量sa中时的下标值k之间的对应关系是：

$$k=i\times(i-1)/2+j-1\qquad i\ge j$$

对称矩阵的压缩存储

(2)若$i<j$：则a_{ij}是在上三角矩阵中。因为$a_{ij}=a_{ji}$，在向量sa中保存的是a_{ji}。依上述分析可得：

$$k=j\times(j-1)/2+i-1\qquad i<j$$

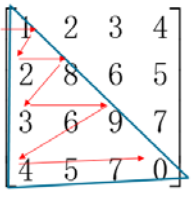
对称矩阵元素a_{ij}保存在向量sa中时的下标值k与(i,j)之间的对应关系是：

$$K=\begin{cases}\boldsymbol{i\times(i-1)/2+j-1} & \text{当}\boldsymbol{i\ge j}\text{时}\\ \boldsymbol{j\times(j-1)/2+i-1} & \text{当}\boldsymbol{i<j}\text{时}\end{cases}\quad \boldsymbol{1\le i,j\le n}$$

对称矩阵的压缩存储示例

以行序为主序存储下三角矩阵（包括对角线元素）：

$$K=\begin{cases}\boldsymbol{i\times(i-1)/2+j-1} & \text{当}\boldsymbol{i\ge j}\text{时}\\ \boldsymbol{j\times(j-1)/2+i-1} & \text{当}\boldsymbol{i<j}\text{时}\end{cases}$$



矩阵索引	11	21	22	31	32	33	41	42	43	44
	1	2	8	3	6	9	4	5	7	0
K	0	1	2	3	4	5	6	7	8	9

三角矩阵的压缩存储

以主对角线划分，三角矩阵有上三角和下三角两种。

(1)上三角矩阵的下三角（不包括主对角线）中的元素均为常数c(一般为0)。

(2)下三角矩阵正好相反，它的主对角线上方均为常数。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ c & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & & \cdots \\ c & c & \cdots & a_{nn} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & c & \cdots & c \\ a_{21} & a_{22} & \cdots & c \\ \cdots & \cdots & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

三角矩阵的压缩存储

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向量$sa[0 \ldots n(n+1)/2]$中，其中c存放在向量的最后一个分量中。

(1)下三角矩阵元素a_{ij}保存在向量sa中时的下标值k与(i,j)之间的对应关系是：

$$K = \begin{cases} \boldsymbol{i \times (i-1)/2 + j - 1} & \text{当} \boldsymbol{i \geq j} \text{时} \\ \boldsymbol{n \times (n+1)/2} & \text{当} \boldsymbol{i < j} \text{时} \end{cases} \quad \boldsymbol{1 \leq i, j \leq n}$$

三角矩阵的压缩存储

(2)上三角矩阵：

当$i \leq j$时，第1行有n个元素，第2行有$(n-1)$个元素，第i行有$(n-i+1)$个元素...

a_{ij}之前的$i-1$行共有元素个数：

$$\sum_{k=1}^{i-1}(n-k+1) = \frac{(i-1)(2n-i+2)}{2}$$

三角矩阵的压缩存储

而在第i行上，a_{ij}之前恰有$j-i$个元素，因此，上三角矩阵的元素a_{ij}保存在向量sa中时的下标值k之间的对应关系是：

$$K = \begin{cases} \boldsymbol{(2n+2-i) \times (i-1)/2 + j - i} & \text{当} \boldsymbol{i \leq j} \text{时} \\ \boldsymbol{n \times (n+1)/2} & \text{当} \boldsymbol{i > j} \text{时} \end{cases} \quad \boldsymbol{1 \leq i, j \leq n}$$

对角矩阵的压缩存储

除了主对角线和主对角线上或下方若干条对角线上的元素之外，其余元素皆为零的矩阵称为对角矩阵。

$$A = \begin{bmatrix} 1 & 4 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ 0 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 \end{bmatrix}$$

如上图三对角矩阵，非零元素仅出现在主对角$(a_{i,i}, 1 \leq i \leq 4)$上、主对角线上的那条对角线$(a_{i,i+1}, 1 \leq i \leq 3)$、主对角线下的那条对角线上$(a_{i+1,i}, 1 \leq i \leq 3)$。显然，当$|i-j|>1$时，元素$aij=0$。

由此可知，一个k对角矩阵(k为奇数)A是满足下述条件：当$|i-j| > (k-1)/2$时，$a_{ij}=0$

对角矩阵的压缩存储

以三对角矩阵为例讨论。

非零元素所在的位置：

(1) $i=1$，$j=1$、2,首行元素

(2) $i=n$，$j=n-1$、n，最后一行

(3) $1<i<n-1, j=i-1$、i、$i+1$

对这种矩阵，当以按“行优先顺序”存储时，第1行和第n行是2个非零元素，其余每行的非零元素都要是3个，则需存储的元素个数为3n-2。

对角矩阵的压缩存储

数组sa中的元素$sa[k]$与三对角矩阵中的元素aij存在一一对应关系，在a_{ij}之前有$i-1$行,共有$3\times(i-1)-1$个非零元素，在第i行，有$j-i+1$个非零元素，这样，非零元素a_{ij}的地址为：

$$LOC[a_{ij}] = LOC[a_{11}] + [3\times(i-1)-1+(j-i+1)]\times l$$

$$= LOC[a_{11}] + (2\times i + j - 3)\times l$$

j与i的差小于等于1

对角矩阵的压缩存储示例

在三对角上的元素位置：

$$LOC[a_{ij}] = LOC[a_{11}] + (2\times i + j - 3)\times l$$

其他位置元素为0。

$$A = \begin{bmatrix} 1 & 4 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ 0 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 \end{bmatrix}$$

矩阵索引	11	12	21	22	23	32	33	34	43	44
	1	4	3	4	1	2	3	4	1	3
K	0	1	2	3	4	5	6	7	8	9

稀疏矩阵的压缩存储

设矩阵A是一个n×m的矩阵中有s个非零元素，设 $\delta = s/(n\times m)$，称δ为稀疏因子，如果某一矩阵的稀疏因子δ满足 $\delta \le 0.05$ 时称为稀疏矩阵。

$$A = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵的压缩存储

对于稀疏矩阵，采用压缩存储方法时，只存储非0元素。必须存储非0元素的行下标值、列下标值、元素值。因此，一个三元组(i,j,a_{ij})唯一确定稀疏矩阵的一个非零元素。

$$A = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(1,2,12)
(1,3,9)
(3,1,-3)
(3,8,4)
(4,6,2)
(5,2,18)
(6,7,-7)
(7,4,-6)

三元组顺序表

若以行序为主序，稀疏矩阵中所有非0元素的三元组，就可以得构成该稀疏矩阵的一个三元组顺序表。相应的数据结构定义如下：

```
//三元组结点定义
typedef int ELEM;
typedef struct
{
int   row;      //行下标
int   col;      //列下标
ELEM value;    //元素值
}Triple;
```

```
//三元组顺序表定义
typedef struct
{
int  rn;          //行数
int  cn;          //列数
int  tn;          //非0元素个数
Triple data[MAX_SIZE];
}TMatrix;
```

稀疏矩阵三元组表示的矩阵运算

矩阵的运算包括矩阵的转置、矩阵求逆、矩阵的加减、矩阵的乘除等。在此，先讨论在这种压缩存储结构下的求矩阵的转置的运算。

一个$m\times n$的矩阵A，它的转置B是一个$n\times m$的矩阵，且$b[i][j] = a[j][i]$，$1 \le i \le n$，$1 \le j \le m$，即B的行是A的列，B的列是A的行。

矩阵转置算法

设稀疏矩阵A是按行优先顺序压缩存储在三元组表$a.data$中，若仅仅是简单地交换$a.data$中i和j的内容，得到三元组表$b.data$，$b.data$将是一个按列优先顺序存储的稀疏矩阵B，要得到按行优先顺序存储的$b.data$，就必须重新排列三元组表$b.data$中元素的顺序。基本算法思想是：

① 将矩阵的行、列下标值交换。即将三元组表中的行、列位置值i、j相互交换；

② 重排三元组表中元素的顺序。即交换后仍然是按行优先顺序排序的。

矩阵转置算法

方法一：

算法思想：

按稀疏矩阵A的三元组表$a.data$中的列次序依次找到相应的三元组存入$b.data$中。

每找转置后矩阵的一个三元组，需从头至尾扫描整个三元组表$a.data$。找到之后自然就成为按行优先的转置矩阵的压缩存储表示。

```
//方法一求转置矩阵的算法,将矩阵a转置存入矩阵b:
void TransMatrix(TMatrix a, TMatrix &b)
{
    int p, q, col;
    b.rn = a.cn;  b.cn = a.rn;  b.tn = a.tn;
    /*置三元组表b.data的行、列数和非0元素个数 */
    if (b.tn == 0)    cout << "The Matrix A = 0" << endl;
    else {
        q = 0;
        for (col = 1; col <= a.cn; col++)
            for (p = 1; p<=a.tn; p++)/*   循环次数是非0元素个数   */
                if (a.data[p].col == col){
                    b.data[q].row = a.data[p].col;
                    b.data[q].col = a.data[p].row;
                    b.data[q].value = a.data[p].value;
                    q++;
                }
    }
}
```

算法分析：本算法主要的工作是在p和col的两个循环中完成的，时间复杂度为$O(cn\times tn)$，即矩阵的列数和非0元素的个数的乘积成正比。

算法复杂度分析

而一般传统矩阵的转置算法为：

```
for (col = 1; col <= n; ++col)
    for (row = 1; row <= m; ++row)
        b[col][row] = a[row][col];
```

时间复杂度为$O(n\times m)$。当非零元素的个数tn和$m\times n$同数量级时，之前算法的时间复杂度为$O(m\times n^2)$。使用三元组虽然节省了存储空间，但时间复杂度却大大增加。所以上述算法只适合于稀疏矩阵中非0元素的个数tn远远小于$m\times n$的情况。

快速转置算法

算法思想：直接按照稀疏矩阵A的三元组表$a.data$的次序依次顺序转换，并将转换后的三元组放置于三元组表$b.data$的恰当位置。

前提：若能预先确定原矩阵A中每一列的(即B中每一行)第一个非0元素在$b.data$中应有的位置，则在作转置时就可直接放在$b.data$中恰当的位置。因此，应先求得A中每一列的非0元素个数。

快速转置算法

附设两个辅助向量$num[\,]$和$cpot[\,]$。

- $num[col]$：统计A中第col列中非0元素的个数；
- $cpot[col]$：指示A中第col列第一个非0元素在$b.data$中的恰当位置。

显然有位置对应关系：

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

求每一列的非零元素个数，转置之后就变成了每一行的非零元素个数

快速转置示例

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

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

num[col]和cpot[col]的值表

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

示例

num[col]和cpot[col]的值表

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

p=1

A=

1	1,2,12
2	1,3,9
3	3,1,-3
4	3,8,4
5	4,6,2
6	5,2,18
7	6,7,-7
8	7,4,-6

1	
2	2,1,12
3	
4	
5	
6	
7	
8	

q=2

p=1,col=A[p].col=2,q=cpot[col]=2, cpot[col]++

示例

num[col]和cpot[col]的值表

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

p=2

A=

1	1,2,12
2	1,3,9
3	3,1,-3
4	3,8,4
5	4,6,2
6	5,2,18
7	6,7,-7
8	7,4,-6

1	
2	2,1,12
3	
4	3,1,9
5	
6	
7	
8	

q=4

p=2,col=A[p].col=3,q=cpot[col]=4,cpot[col]++

示例

num[col]和cpot[col]的值表

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

p=3

A=

1	1,2,12
2	1,3,9
3	3,1,-3
4	3,8,4
5	4,6,2
6	5,2,18
7	6,7,-7
8	7,4,-6

1	1,3,-3
2	2,1,12
3	
4	3,1,9
5	
6	
7	
8	

q=1

p=3,col=A[p].col=1,q=cpot[col]=1,cpot[col]++

示例

num[col]和cpot[col]的值表

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

p=4

A=

1	1,2,12
2	1,3,9
3	3,1,-3
4	3,8,4
5	4,6,2
6	5,2,18
7	6,7,-7
8	7,4,-6

1	1,3,-3
2	2,1,12
3	
4	3,1,9
5	
6	
7	
8	8,3,4

q=8

p=4,col=A[p].col=8,q=cpot[col]=8,cpot[col]++

示例

num[col]和cpot[col]的值表

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

p=5

A=

1	1,2,12
2	1,3,9
3	3,1,-3
4	3,8,4
5	4,6,2
6	5,2,18
7	6,7,-7
8	7,4,-6

1	1,3,-3
2	2,1,12
3	
4	3,1,9
5	
6	6,4,2
7	
8	8,3,4

q=6

p=5,col=A[p].col=6,q=cpot[col]=6,cpot[col]++

示例

num[col]和cpot[col]的值表

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

p=6

A=

1	1,2,12
2	1,3,9
3	3,1,-3
4	3,8,4
5	4,6,2
6	5,2,18
7	6,7,-7
8	7,4,-6

1	1,3,-3
2	2,1,12
3	2,5,18
4	3,1,9
5	
6	6,4,2
7	
8	8,3,4

q=3

p=6,col=A[p].col=2,q=cpot[col]=3,cpot[col]++

示例

num[col]和cpot[col]的值表

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

p=7

A=

1	1,2,12
2	1,3,9
3	3,1,-3
4	3,8,4
5	4,6,2
6	5,2,18
7	6,7,-7
8	7,4,-6

1	1,3,-3
2	2,1,12
3	2,5,18
4	3,1,9
5	
6	6,4,2
7	7,6,-7
8	8,3,4

q=7

p=7,col=A[p].col=7,q=cpot[col]=7,cpot[col]++

示例

num[col]和cpot[col]的值表

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

p=8

A=

1	1,2,12
2	1,3,9
3	3,1,-3
4	3,8,4
5	4,6,2
6	5,2,18
7	6,7,-7
8	7,4,-6

1	1,3,-3
2	2,1,12
3	2,5,18
4	3,1,9
5	4,7,-6
6	6,4,2
7	7,6,-7
8	8,3,4

q=5

p=8,col=A[p].col=4,q=cpot[col]=5,cpot[col]++

```
//快速转置算法
void  FastTransMatrix(TMatrix a, TMatrix &b) {
    int p, q, col, k;
    int num[MAX_SIZE], cpot[MAX_SIZE];
    b.rn = a.cn; b.cn = a.rn; b.tn = a.tn;
    /*   置三元组表b.data的行、列数和非0元素个数   */
    if (b.tn == 0)    cout<<"The Matrix A = 0"<<endl;
    else {
        for (col = 1; col <= a.cn; ++col)    num[col] = 0;
        /*  向量num[]初始化为0    */
        for (k = 1; k <= a.tn; ++k)
            ++num[a.data[k].col];//求原矩阵中每一列非0元素个数
        for (cpot[1] = 1, col = 2; col <= a.cn; ++col)
            cpot[col] = cpot[col - 1] + num[col - 1];
        /*  求第col列中第一个非0元在b.data中的序号 */
        for (p = 1; p <= a.tn; ++p) {
            col = a.data[p].col;  q = cpot[col];
            b.data[q].row = a.data[p].col;
            b.data[q].col = a.data[p].row;
            b.data[q].value = a.data[p].value;
            ++cpot[col];
        }
    }
}
```

行逻辑链接的三元组顺序表

将上述方法二中的辅助向量$cpot[\,]$固定在稀疏矩阵的三元组表中，用来指示“行”的信息。得到另一种顺序存储结构：行逻辑链接的三元组顺序表。其类型描述如下：

```
typedef struct
{
    Triple data[MAX_SIZE];     /*  非0元素的三元组表  */
    int    rpos[MAX_ROW];      /* 各行第一个非0位置表 */
    int    rn, cn, tn;        /*  矩阵的行、列数和非0元个数  */
}RLSMatrix;
```

稀疏矩阵的乘法

设有两个矩阵：$A=(a_{ij})_{m\times n}$，$B=(b_{ij})_{n\times p}$

则：$C=(c_{ij})_{m\times p}$，其中 $c_{ij}=\sum a_{ik}\times b_{kj}$

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

经典算法是三重循环：

```
for (i = 1; i <= m; ++i) {
    for (j = 1; j <= p; ++j) {
        c[i][j] = 0;
        for (k = 1; k <= n; ++k)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
    }
}
```

此算法的复杂度为O(m×n×p)。

稀疏矩阵的乘法

设两个稀疏矩阵$A=(a_{ij})_{m\times n}$,$B=(b_{ij})_{n\times p}$，存储结构采用行逻辑链接的三元组顺序表。

算法思想：对于A中的每个元素$a.data[p](p=1,2,\dots,a.tn)$，找到B中所有满足条件：

$$a.data[p].col=b.data[q].row$$

的元素$b.data[q]$，并计算p、q结点对应值的乘积，该乘积是c_{ij}中的一部分。求得所有这样的乘积并累加求和就能得到c_{ij}。

稀疏矩阵的乘法

为得到非0的乘积，只要对$a.data[1\dots a.tn]$中每个元素

$$(i,\ k,\ a_{ik})\ (1\le i\le a.rn, 1\le k\le a.cn)$$

找到$b.data$中所有相应的元素

$$(k,\ j,\ b_{kj})(1\le k\le b.rn,\ 1\le j\le b.cn)$$

相乘即可。则必须知道矩阵B中第k行的所有非0元素，而$b.rpos[\ ]$向量中提供了相应的信息。

算法思路($Q=M\times N$)

```
Q初始化;
if  Q是非零矩阵 {  // 逐行求积
   for (arow=1; arow<=M.rn; ++arow) {
                  //  处理M的每一行
      ctemp[] = 0;        // 累加器清零
      计算Q中第arow行的积并存入ctemp[] 中;
      将ctemp[] 中非零元压缩存储到Q.data;
   } // for arow
}
```

乘法示例

$$M=\begin{bmatrix}3&0&0&5\\0&-1&0&0\\2&0&0&0\end{bmatrix},\ N=\begin{bmatrix}0&2\\-1&0\\-2&4\\0&0\end{bmatrix}$$

M三元组表示

Row	Col	Value
1	1	3
1	4	5
2	2	-1
3	1	2

N三元组表示

Row	Col	Value
1	2	2
2	1	1
3	1	-2
3	2	4

指针$arow$遍历M中的每一行，p为本行中的一个元素，以p的col值去找N表row值相同、col值记为$ccol$的元素相乘并累加，作为Q表在$(arow,ccol)$位置的乘积结果。

乘法示例

M三元组表示

Row	Col	Value
1	1	3
1	4	5
2	2	-1
3	1	2

N三元组表示

Row	Col	Value
1	2	2
2	1	1
3	1	-2
3	2	4

Q三元组表示

Row	Col	Value
1	2	6

$arow=1, col=1$，$ccol=2$,Q添加元素(1,2,6)

乘法示例

M三元组表示

Row	Col	Value
1	1	3
1	4	5
2	2	-1
3	1	2

N三元组表示

Row	Col	Value
1	2	2
2	1	1
3	1	-2
3	2	4

Q三元组表示

Row	Col	Value
1	2	6

$arow=1, col=4$未找到对应位置的元素。

乘法示例

M三元组表示

Row	Col	Value
1	1	3
1	4	5
2	2	-1
3	1	2

N三元组表示

Row	Col	Value
1	2	2
2	1	1
3	1	-2
3	2	4

Q三元组表示

Row	Col	Value
1	2	6
2	1	-1

$arow = 2, col = 2，\ ccol = 1$, Q添加元素$(2,1,-1)$

乘法示例

M三元组表示

Row	Col	Value
1	1	3
1	4	5
2	2	-1
3	1	2

N三元组表示

Row	Col	Value
1	2	2
2	1	1
3	1	-2
3	2	4

Q三元组表示

Row	Col	Value
1	2	6
2	1	-1
3	2	4

$arow = 3, col = 1，\ ccol = 2$, Q添加元素$(3,2,4)$

代码实现

```
//求矩阵乘积Q=M*N,采用行逻辑链接存储表示
int MultSMatrix(RLSMatrix M, RLSMatrix N, RLSMatrix &Q)
{ /* 求稀疏矩阵乘积Q=M×N。算法5.3 */
    int arow, brow, p, q, ccol, ctemp[MAXRC + 1];
    if (M.cn != N.rn) /* 矩阵M的列数应和矩阵N的行数相等 */
        return - 1;
    Q.rn = M.rn; /* Q初始化 */
    Q.cn = N.cn;
    Q.tn = 0;
    M.rpos[M.rn + 1] = M.tn + 1; /* 为方便后面的while循环临时设置 */
    N.rpos[N.rn + 1] = N.tn + 1;
    if (M.tn*N.tn != 0) /* M和N都是非零矩阵 */
    {
        for (arow = 1; arow <= M.rn; ++arow)
        { /* 从M的第一行开始，到最后一行，arow是M的当前行 */
            for (ccol = 1; ccol <= Q.cn; ++ccol)
                ctemp[ccol] = 0; /* Q的当前行的各列元素累加器清零 */
```

```
            Q.rpos[arow] = Q.tn + 1; /* Q当前行的第1个元素位于上1行最后1个元素之后 */
            for (p = M.rpos[arow]; p<M.rpos[arow + 1]; ++p){//对M当前行中每一个非零元
                brow = M.data[p].col; /* 找到对应元在N中的行号(M当前元的列号) */
                for (q = N.rpos[brow]; q<N.rpos[brow + 1]; ++q) {
                    ccol = N.data[q].col; /* 乘积元素在Q中列号 */
                    ctemp[ccol] += M.data[p].value*N.data[q].value;
                }
            } /* 求得Q中第arow行的非零元 */
            for (ccol = 1; ccol <= Q.cn; ++ccol) //压缩存储该行非零元
                if (ctemp[ccol]) {
                    if (++Q.tn>MAX_SIZE)
                        return - 1;
                    Q.data[Q.tn].row = arow;
                    Q.data[Q.tn].col = ccol;
                    Q.data[Q.tn].value = ctemp[ccol];
                }
        }
    }
    return 1;
}
```

十字链表

对于稀疏矩阵，当非0元素的个数和位置在操作过程中变化较大时，采用链式存储结构表示比三元组的线性表更方便。

矩阵中非0元素的结点所含的域有：行、列、值、行指针(指向同一行的下一个非0元)、列指针(指向同一列的下一个非0元)。其次，十字交叉链表还有一个头结点，结点的结构如图所示。

row	col	value
down	right	

(a) 结点结构

rn	cn	tn
down	right	

(b) 头结点结构

十字链表

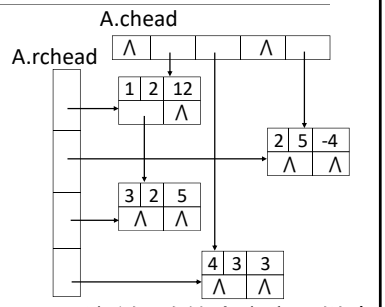
由定义知，稀疏矩阵中同一行的非0元素的由right指针域链接成一个行链表， 由down指针域链接成一个列链表。则每个非0元素既是某个行链表中的一个结点，同时又是某个列链表中的一个结点，所有的非0元素构成一个十字交叉的链表。称为十字链表。

此外，还可用两个一维数组分别存储行链表的头指针和列链表的头指针。

示例

$$A=\begin{pmatrix}0 & 12 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & -4\\ 0 & 5 & 0 & 0 & 0\\ 0 & 0 & 3 & 0 & 0\end{pmatrix}$$

(a) 稀疏矩阵



(b) 稀疏矩阵的十字交叉链表

十字链表的存储表示

```
typedef struct OLNode
{
    int row, col; /* 该非零元的行和列下标 */
    ELEM value; /* 非零元素值 */
    struct OLNode *right, *down; /* 该非零元所在行表和列表的后继链域 */
}OLNode, *OLink;
typedef struct
{
    OLink *rhead, *chead; /* 行和列链表头指针向量基址,由CreatSMatrix_OL()分配 */
    int rn, cn, tn; /* 稀疏矩阵的行数、列数和非零元个数 */
}CrossList;
```

插入新结点时需要同时对行和列链表添加指针链接。

```
void CreateSMatrix(CrossList &M)
{ /* 创建稀疏矩阵M,采用十字链表存储表示。算法5.4 */
    int i, j, k, m, n, t;
    ELEM e;
    OLNode *p, *q;
    cout << "请输入稀疏矩阵的行数 列数 非零元个数: " << endl;
    cin >> m >> n >> t;
    M.rn = m; M.cn = n; M.tn = t;
    M.rhead = (OLink*)malloc((m + 1) * sizeof(OLink));
    if (!M.rhead)
        exit(-1);
    M.chead = (OLink*)malloc((n + 1) * sizeof(OLink));
    if (!M.chead)
        exit(-1);
    for (k = 1; k <= m; k++) /* 初始化行头指针向量;各行链表为空链表 */
        M.rhead[k] = NULL;
    for (k = 1; k <= n; k++) /* 初始化列头指针向量;各列链表为空链表 */
        M.chead[k] = NULL;
    cout << "请按任意次序输入%d个非零元的行 列 元素值:" << M.tn;
    for (k = 0; k<t; k++)
    {
        scanf("%d%d%d", &i, &j, &e);
        cin >> i >> j >> e;
```

```
    p = (OLNode*)malloc(sizeof(OLNode));
    p->row = i; /* 生成结点 */
    p->col = j;
    p->value = e;
    if (M.rhead[i] == NULL || M.rhead[i]->col>j) //p插在该行第一个结点处
    {
        p->right = M.rhead[i];
        M.rhead[i] = p;
    }
    else /* 寻查在行表中的插入位置 */ {
        for (q = M.rhead[i]; q->right&&q->right->col<j; q = q->right);
        p->right = q->right; /* 完成行插入 */
        q->right = p; }
    if (M.chead[j] == NULL || M.chead[j]->row>i) {
        /* p插在该列的第一个结点处 */
        p->down = M.chead[j];
        M.chead[j] = p; }
    else /* 寻查在列表中的插入位置 */{
        for (q = M.chead[j]; q->down&&q->down->row<i; q = q->down);
        p->down = q->down; /* 完成列插入 */
        q->down = p; }
}
}
```

广义表

广义表是线性表的推广和扩充。

之前，我们把线性表定义为$n(n \geq 0)$个元素$a_1, a_2, \ldots, a_n$的有穷序列，该序列中的所有元素具有相同的数据类型且只能是原子项(Atom)。所谓原子项可以是一个数或一个结构，是指结构上不可再分的。若放松对元素的这种限制，容许它们具有其自身结构，就产生了广义表的概念。

广义表

广义表(Lists,又称为列表)：是由$n(n \geq 0)$个元素组成的有穷序列：

$$LS = (a_1, a_2, \ldots, a_n)$$

其中 a_i 或者是原子项，或者是一个广义表。LS是广义表的名字，n为它的长度。若a_i是广义表，则称为LS的子表。

习惯上：原子用小写字母，子表用大写字母。

相关术语

若广义表LS非空时：

- a_1(表中第一个元素)称为表头。
- 其余元素组成的子表(a_2，a_3，…，a_n)称为表尾。
- 广义表中所包含的元素(包括原子和子表)的个数称为表的长度。
- 广义表中括号的最大层数称为表深 (度)。

广义表示例

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

广义表及其示例

广 义 表	表长n	表深h
A=()	0	0
B=(e)	1	1
C=(a,(b,c,d))	2	2
D=(A,B,C)	3	3
E=(a,E)	2	∞
F=(())	1	2

广义表的重要结论

(1) 广义表的元素可以是原子，也可以是子表，子表的元素又可以是子表…即广义表是一个多层次的结构。

(2) 广义表可以被其它广义表所共享，也可以共享其它广义表。广义表共享其它广义表时通过表名引用。

(3) 广义表本身可以是一个递归表,即广义表也可以是其本身的一个子表。

广义表的存储结构

由于广义表中的数据元素具有不同的结构，通常用链式存储结构表示，每个数据元素用一个结点表示。因此，广义表中只有两类结点：

(1)一类是表结点，用来表示广义表项，由标志域，表头指针域，表尾指针域组成;

(2)另一类是原子结点，用来表示原子项，由标志域，原子的值域组成。

只要广义表非空，都是由表头和表尾组成。即一个确定的表头和表尾就唯一确定一个广义表。

广义表的存储结构

标志tag=0	原子的值

(a) 原子结点

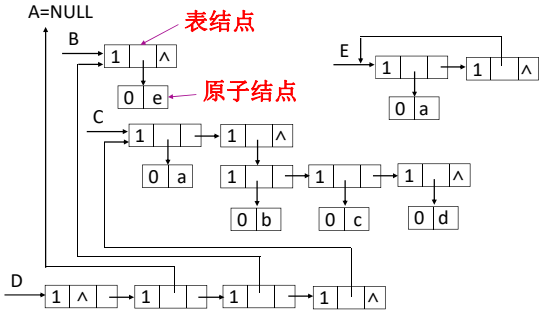
标志tag=1	表头指针hp	表尾指针tp

(b) 表结点

```
typedef struct GLNode
{
    int   tag;     /*  标志域，为1：表结点;为0 ：原子结点  */
    union
    {
        ELEM value;     /* 原子结点的值域  */
        struct
        {
            struct GLNode  *hp, *tp;
        }ptr;   //  ptr为表结点的指针域，hp、tp分别指向表头和表尾
    }Gdata;
} *GList,GLNode;      /* 广义表结点类型  */
```

示例

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



分析

对于上述存储结构，有如下几个特点：

(1) 若广义表为空，表头指针为空；否则，表头指针总是指向一个表结点，其中hp指向广义表的表头结点(或为原子结点，或为表结点)，tp指向广义表的表尾(表尾为空时，指针为空，否则必为表结点)。

(2) 这种结构求广义表的长度、深度、表头、表尾的操作十分方便。

(3) 表结点太多，造成空间浪费。

广义表的扩展线性链表存储

标志tag=0	值域value	next结点指针tp

(a) 原子结点

标志tag=1	表头指针hp	next结点指针tp

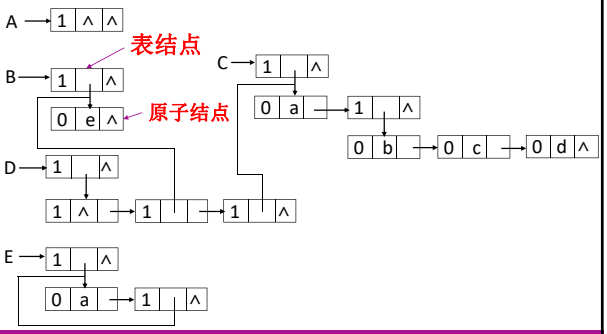
(b) 表结点

```
typedef struct GLNode
{
    int   tag;     /*  标志域，为1：表结点;为0 ：原子结点  */
    union
    {
        ELEM value;     /* 原子结点的值域  */
        struct GLNode *hp;/* 表结点的表头指针  */
    }Gdata;
    struct GLNode *tp;//相当于线性链表的next指向下一个元素结点
} *GList, GLNode;       /* 广义表结点类型  */
```

示例

A=()，B=(e)，C=(a, (b, c, d))

D=(A, B, C)，E=(a, E)



m元多项式的表示

对一个三元多项式：

$$P(x,y,z) = x^{10}y^3z^2 + 2x^6y^3z^2 + 3x^5y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz + 15$$

可以改写为：

$$P(x,y,z) = \left((x^{10} + 2x^6)y^3 + 3x^5y^2\right)z^2 + ((x^4 + 6x^3)y^4 + 2y)z + 15$$

多项式P为变元z的多项式，即$Az^2 + Bz + 15$,其中A和B又是(x,y)的多项式，符合广义表的结构。

m元多项式的表示

$$P(x,y,z) = \left((x^{10} + 2x^6)y^3 + 3x^5y^2\right)z^2 + ((x^4 + 6x^3)y^4 + 2y)z + 15$$

可以分解为：

$$P = z((A,2),(B,1),(15,0))$$

其中$A = y((C,3),(D,2))$

$$C = x((1,10),(2,6))$$

$$D = x((3,5))$$

$$B = y((E,4),(F,1))$$

$$E = x((1,4),(6,3))$$

$$F = x((2,0))$$

m元多项式的结点存储结构

结构与扩展的广义表存储方式类似，添加了exp来存储多项式的指数。

tag=0	exp	hp	tp

(a) 原子结点

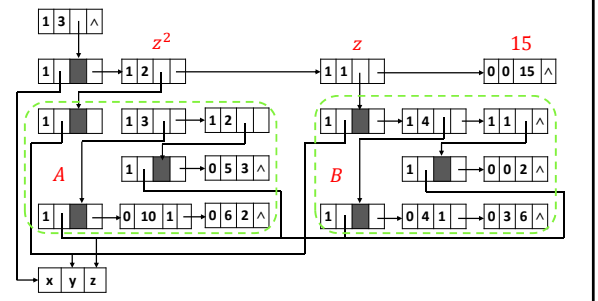
tag=1	exp	coef	tp

(b) 表结点

```
typedef struct MPNode
{
    int   tag;     /*  标志域，为1：表结点;为0 ：原子结点  */
    int exp;  //指数域
    union
    {
        float coef;     //系数域
        struct MPnode * hp; //表结点的表头指针
    }Gdata;
    struct MPNode *tp;//相当于线性链表的next指向下一个元素结点
} MPNode;       /* m元多项式广义表类型  */
```

$P(x,y,z) = ((x^{10}+2x^6)y^3 + 3x^5y^2)z^2 + ((x^4+6x^3)y^4 + 2y)z + 15$

A B



广义表的递归算法

递归函数

一个含直接或间接调用本函数语句的函数被称之为递归函数，它必须满足以下两个条件：

1)在每一次调用自己时，必须是(在某种意义上)更接近于解；

2)必须有一个终止处理或计算的准则。

广义表的递归算法

广义表从结构上可以分解成

广义表 =

子表1 + 子表2 + … + 子表n

因此常利用分治法求解之。

算法设计中的关键问题是，如何将 l 个子问题的解组合成原问题的解。

求广义表的深度

将广义表分解成 n 个子表，分别(递归)求得每个子表的深度，

广义表的深度=Max {子表的深度} +1

可以直接求解的两种简单情况为：

空表的深度 = 1

原子的深度 = 0

算法实现

```
int GListDepth(GList L)
{ /* 采用头尾链表存储结构,求广义表L的深度。 */
    int max, dep;
    GList pp;
    if (!L)
        return 1; /* 空表深度为1 */
    if (L->tag == 0)
        return 0; /* 原子深度为0 */
    for (max = 0, pp = L; pp; pp = pp->Gdata.ptr.tp)
    {
        dep = GListDepth(pp->Gdata.ptr.hp); /* 求以pp->Gdata.ptr.hp
为头指针的子表深度 */
        if (dep>max)
            max = dep;
    }
    return max + 1; /* 非空表的深度是各元素的深度的最大值加1 */
}
```

复制广义表

将广义表分解成表头和表尾两部分，分别(递归)复制求得新的表头和表尾，

新的广义表由新的表头和表尾构成。

可以直接求解的两种简单情况为：

空表复制求得的新表自然也是空表；

原子结点可以直接复制求得。

算法描述

若 ls= NIL 则 newls = NIL

否则

构造结点 newls,

由表头ls->Gdata.ptr.hp 复制得 newhp

由表尾 ls->Gdata.ptr.tp 复制得 newtp

并使newls->Gdata.ptr.hp = newhp,

newls->Gdata.ptr.tp = newtp

广义表的复制算法实现

```
void CopyGList(GList &T, GList L)
{ /* 采用头尾链表存储结构,由广义表L复制得到广义表T。算法5.6 */
    if (!L) T = NULL;
    else  {
        T = (GList)malloc(sizeof(GLNode)); /* 建表结点 */
        if (!T) exit(-1);
        T->tag = L->tag;
        if (L->tag == 0)
            T->Gdata.value = L->Gdata.value; /* 复制单原子 */
        else
        {
            CopyGList(T->Gdata.ptr.hp, L->Gdata.ptr.hp);
          /* 复制广义表L->Gdata.ptr.hp的一个副本T->Gdata.ptr.hp */
            CopyGList(T->Gdata.ptr.tp, L->Gdata.ptr.tp);
         /* 复制广义表L->Gdata.ptr.tp的一个副本T->Gdata.ptr.tp */
        }
    }
}
```