第五章 数组和广义表

刘亮亮

2020 年 4 月 26 日

(声明：本讲义为草稿，限内部使用，版权所有，未经同意，不得外传)

- 要点：本章主要介绍（1）数组的定义与存储结构及基本操作的实现；（2）特殊矩阵与稀疏矩阵的压缩存储及矩阵运算的实现；（3）广义表的定义与存储结构，以及广义表的基本操作与应用。
- 重点：1.数组的存储结构与实现；2.矩阵的压缩存储;
- 难点：1.稀疏矩阵的存储结构与实现；2.广义表的递归算法

本章介绍两种线性表的扩展结构——数组与广义表，也是线性结构，也是本课程介绍的最后两种线性结构。相比于线性表存储的元素是“原子”类型（也就是不可以再表示成数据结构），数组和广义表里面的元素既可以是原子类型，也可以是数据结构，也就是说也可以是数组或广义表。

1 数组

在现在的高级程序设计语言中，数组常常看做成一种基本数据类型来使用，这里我们讲的数组，是把数组看成是一种数据结构。这里讨论数组，是为了让大家更深入了解数组的原理，从而更能掌握数组。

1.1 数组定义

数组与线性表、栈和队列一样，都用来存储具有 ”一对一” 逻辑关系数据的线性结构。数组可以直接存储多个顺序表，因为顺序表的底层实现还是

数组，因此等价于数组中存储数组，这和平时高级程序设计语言（如C++）使用的多维数组类似，可以称为“数组中的数组”。数组中的数据元素的类型必须是相同的。

根据数组的维度分，可以将数组分为以下几种：

- 一维数组：指存储不可再分数据元素的数组。示意图如下图1所示：

Figure 1: 一维数组示例

- 二维数组：指的存储一维数组的一维数组，每一维中存放的是一个一维数组，通常称为行和列，对每一维而言，里面的一维数组的长度（元素个数）相同。

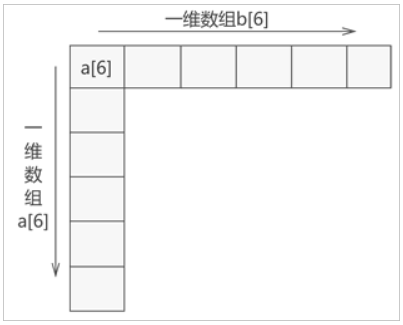

Figure 2: 二维数组示例

- n维数组:指的是存储 n-1 维数组的一维数组。

数组的抽象数据类型定义如下：

ADT Array{

数据对象:

$j_i = 0, ...b_{i-1}, i = 1, 2, ..., n$

D ={$a_{j_1 j_2 ... j_n}$—n称为数组的维数，b_i是数组第i维的长度，j_i是数组元素的第i维下标}

数据关系:

$R = \{R_1, R_2, ..., R_n\}$

$R_i = \{< a_{j_1 ... j_i ... j_n}, a_{j_1 ... j_{i+1} ... j_n} >\}$

基本操作：

$initArray$(&arr,n,$bound$1,...,$boundn$)

操作结果：若维数n合法，构建相应的n维数组。

$value(arr, \&e, index1, ..., indexn)$

初始条件：$index1, ...indexn$不越界。

操作结果： 返回对应$index1, ...indexn$下标的元素 。

$assign(\&arr, e, index1, ..., indexn)$

初始条件：$index1, ...indexn$不越界。

操作结果： 用e给给定下标$index1...indexn$对应的元素赋值。

}End Array.

1.2 数组的存储结构

因为需要通过下标来访问数组中的元素，并且数组一般只做查找和修改数据的操作，因此，数组一般是采用顺序存储结构来存放。由于数组是多维的，而顺序存储结构是一维的，因此数组中数据的存储要定义一个存储的先后次序，才能有序的进行存储。通常，有两种次序：

- 以列序为主的存储结构：依次存放每一列的元素，每一列按行的从小到大进行存放，特点：先列后行。
- 以行序为主的存储结构：依次存放每一行的元素，每一行按列号从小到大的进行存放，特点：先行后列。

以二维数组a[6][6]={{$a_{00}, ..., a_{05}$},{$a_{10}, ..., a_{15}$},...,{$a_{50}, ..., a_{55}$}}为例，列序为主的存储示意图如图3，行序为主的存储示意图如图4所示。

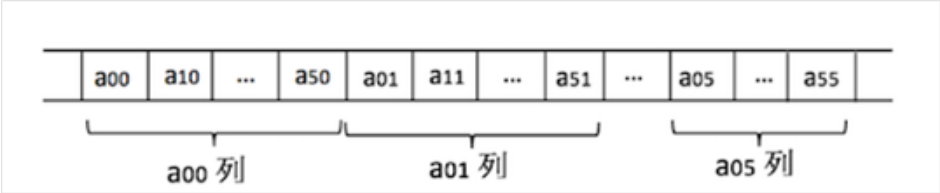

Figure 3: 列序为主的存储示意图

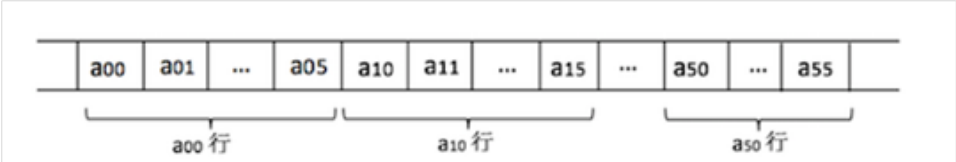

Figure 4: 行序为主的存储示意图

下面以二维数组为例，介绍数组的这两种存储结构。

假设二维数组a[m][n]，第一个元素的地址是Loc(0,0)，即a[0][0]的地址，也是数组的首地址。每个元素占用L个存储单元。

若采用行序为主序的存储结构，则a[i][j]的存储位置可以通过如下公式进行计算得到：

$$Loc(i,j) = Loc(0,0) + (i * n + j) * L$$

若采用以列序存储的方式，则a[i][j]的存储地址可以通过如下公式计算得到：

$$LOC(i,j) = LOC(0,0) + (j * m + i) * L;$$

根据上面的公式，顺序存储的多维数组中查找某个指定元素时，需知道以下信息：

- 多维数组的存储方式（列序还是行序）；
- 多维数组在内存中存放的起始地址；
- 该指定元素在原多维数组的下标；
- 数组中元素的类型，即数据元素所占内存大小，用L表示

上面的存储计算可以推广到n维的情况，以行序为主的顺序存储，每一维的长度分别为b_1,b_1,...,b_n,起始地址为$LOC(0,...,0)$，则：

$$LOC(j_1, j_2, ..., j_n) = LOC(0, 0, ..., 0)+b_2*b_3*...*b_n*j_1+b_3*...*b_n*j_2+...+b_n*j_{n-1}+j_n$$

$$\begin{aligned} LOC(j_1, j_2, ..., j_n) &= LOC(0, 0, ..., 0) + b_2 * b_3 * ... * b_n * j_1 + b_3 * ... * b_n * j_2 + ... + b_n * j_{n-1} + j_n \\ &= LOC(0, 0, ..., 0) + (\sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} b_k + j_n) * L \end{aligned} \tag{1}$$

可以缩写为：

$$LOC(j_1, j_2, ..., j_n) = LOC(0, 0, ..., 0) + \sum_{i=1}^{n} c_i j_i$$

其中:$c_n = L$，$c_{i-1} = b_i * c_i, 1 < i <= n$

存储结构描述如下：

```
#include<stdarg.h>      //C标准头文件，提供宏va_start, va_arg和va_end
                        //用于存取边长参数表
#define MAX_ARRAY_DIM 8    //假设数组维数的最大值为8
typedef struct Array
{
        ElemType *base;   //数组元素基址，由InitArray 分配
        int dim;          //数组维数
        int *bounds;      //数组维界基址，由InitArray 分配
        int *constants;   // 数组映象函数常量基址，由InitArray 分配
}Array;
```

1.3 数组的基本操作实现

数组的基本操作包括初始化数组、销毁数组、取数组元素、对数组元素赋值等。操作声明如下所示：

```
//若维数dim和随后的各维长度合法，则构造相应的数组arr,返回OK
Status initArray(Array &arr,int dim…,);
//销毁数组arr
Status destroyArray(Array &arr);
//各下标不越界，用e返回数组arr对应的元素
Status value(Array arr,ElemType &e…,);
//各下标不越界，对对应下标的arr的元素赋值e
Status assign(Array &arr, ElemType e…,);
```

由于维数是多维，不确定的，因此*initArray*、*value*、*assign*等函数使用了可变参数列表“...”，需要包含*C*语言的头文件（$\#include < stdarg.h >$）或*C++*的（$\#include < cstdarg >$），需要利用到三个宏*va_start,va_arg,va_end*，具体参考*C*或*C++*可变参数知识

1.3.1 初始化数组操作initArray

初始化数组操作，是根据用户指定的维数以及指定的每一维的长度来初始化一个空的数组。由于是多维数组，因此需要用到可变参数来定义初始化数组操作。定义是$initArray(Array\& \ arr, int \ dim, ...)$，其中arr是初始化的空数组，dim是维数参数，“...”是可变参数列表。

算法思想如下：

Step 1：首先判断维数*dim*是否合法（维数不能小于1，也不能大于定义的最大维数），若合法则对arr的维数赋值，否则返回错误。

$if \ (dim < 1||dim > MAX\_ARRAY\_DIM) \quad arr.dim = dim;$

Step 2：根据维数*dim*开辟维数维界基址，赋值给bounds,bounds是存储每一维的长度,如果分配失败，则退出初始化，返回错误。

$arr.bounds = (int *)malloc(dim * sizeof(int));$

Step 3: 根据可变参数，判断每一维是否合法，分别赋值给bounds，并且定义elemtotal变量计算数组arr元素总数。

$va\_start(ap, dim);$ 其中ap是va_list类型，va_start的第二个参数是“...”前面的参数dim。

$for(int \ i = 0; i < dim; ++i)$

{

$arr.bounds[i] = va\_arg(ap, int);$

$elemtotal *= arr.bounds[i];$

}

注：va_arg获取ap中的参数,int获得参数类型

va_end(ap);//关闭ap指针

Step 4: 开辟数组arr的基址，赋值给base,如果分配失败，则退出，返回错误

$arr.base = (ElemType *)malloc(elemtotal * sizeof(ElemType));$

Step 5: 根据前面的推导公式计算数组映象函数常量基址

$arr.constants[dim-1] = 1;$//L=1,表示指针的增减以元素的大小为单位。

$for(i = dim-2; i >= 0; --i)\{arr.constants[i] = arr.bounds[i+1] * arr.constants[i+1];\}$

具体的算法描述如下：

```

Status InitArray(Array *arr,int dim,...)
{
    int elemtotal=1,i; //elemtotal是元素总值
    va_list ap;
    if(dim<1||dim>MAX_ARRAY_DIM)
        return ERROR;
    (*arr).dim=dim;
    (*arr).bounds=(int )malloc(dim*
        sizeof(int));
     if(!(*arr).bounds)
          exit(OVERFLOW);
     va_start(ap,dim);
     for(i=0; i<dim; ++i)
     {
          (*arr).bounds[i]=va_arg(ap,int);
          if((*arr).bounds[i]<0)
              return UNDERFLOW;
          elemtotal*=(*arr).bounds[i];
     }
```

```
    va_end(ap);
    (*arr).base=(ElemType *) malloc
      (elemtotal*sizeof(ElemType));
    if(!(*arr).base)
        exit(OVERFLOW);
    (*arr).constants=(int *)malloc
        (dim*sizeof(int));
     if(!(*arr).constants)
          exit(OVERFLOW);
     (*arr).constants[dim-1]=1;
     for(i=dim-2; i>=0; --i)
     {
       (*arr).constants[i]=(*arr).bounds[i+1]
           * (*arr).constants[i+1];
      }
      return OK;
}
```

1.3.2 销毁数组操作destroyArray

数组的顺序存储结构里面各个组成部分都是采用动态分配的存储空间，因此在使用完数组以后需要对数组进行释放，否则会导致内存泄露。当数组存储结构里面的动态分配的指针不为空的时候，调用free函数进行释放就可以。具体的算法描述如下：

```
Status DestroyArray(Array *arr)
{
    //销毁数组arr
    if((*arr).base)
    {
        free((*arr).base);
        (*arr).base=NULL;
    }
```

```
9       else
10      {
11          return ERROR;
12      }
13      if((*arr).bounds)
14      {
15          free((*arr).bounds);
16          (*arr).bounds=NULL;
17      }
18      else
19      {
20          return ERROR;
21      }
22      if((*arr).constants)
23      {
24          free((*arr).constants);
25          (*arr).constants=NULL;
26      }
27      else
28      {
29          return ERROR;
30      }
31      return OK;
32  }
```

1.3.3 取值操作value

取值是给定下标值，从数组中取出对应的值的操作，函数声明是Status value(ElemType *e,Array arr,...),由于数组是多维数组，只有在用户定义好数组，并且在调用时候才知道数组的维数和给定的下标的个数，因此需要采用可变长参数来实现。

在实现取值操作前，我们首先实现一个公用函数，就是给定数组和下标，找该组下标在数组中对应的偏移位置。函数声明是Status locate(Array& arr,va_list ap,int *off),这个函数在后面的赋值操作也需要用。

算法思想：

Step1:首先判断下标是否合法，在每一维的长度范围内，否则返回错误标识；

Step2:利用前面计算地址的公式进行地址偏移计算，求出相对偏移位置off。需要对可变参数列表进行遍历。

具体的算法描述如下：

```
Status locate(Array& arr,va_list ap,int *off)
{
    //若ap 指示的各下标值合法，则求出该元素在arr 中的相对地址off
    int i,ind;
    *off=0;
    for(i=0; i<arr.dim; i++)
    {
            ind=va_arg(ap,int);
            if(ind<0||ind>=arr.bounds[i])
                return OVERFLOW;
             *off+=arr.constants[i]*ind;
    }
    return OK;
}
```

实现了locate操作，那么value操作功能描述：用e返回指定下标的数据元素

算法思想：

Step1：对给定的下标参数列表，调用locate函数计算指定下标的元素地址

Step2: 对e进行赋值，返回

```
Status Value(ElemType *e,Array arr,...)
{
    //依次为各维的下标值，若各下标合法，则e被赋值为A的相应的元素值
```

```
    va_list ap;
    Status result;
    int off;
    va_start(ap,arr);
    if((result=Locate(arr,ap,&off))==OVERFLOW) //调用locate()
      return result;
    *e=*(arr.base+off);//赋值
    return OK;
}
```

1.3.4 赋值操作assign

给指定下标的元素进行赋值,函数声明是Status assign(Array *arr,ElemType e,...)

算法思想：根据各维的下标值，若各下标合法，调用locate函数根据指定下标计算地址，则将e的值赋给arr的指定的元素。算法描述如下：

```
Status Assign(Array *arr,ElemType e,...)
{
    //依次为各维的下标值，若各下标合法，则将e的值赋给arr指定的元素
    va_list ap;
    Status result;
    int off;
    va_start(ap,e);
    if((result=Locate(*A,ap,&off))==OVERFLOW) //调用Locate()
      return result;
    *((*arr).base+off)=e;
    return OK;
}
```

2 矩阵的压缩存储

矩阵是数学中的概念，是一种非常有用数据对象之一，常用于科学计算中，在机器学习领域里面应用比较广泛，特别是在深度学习中，基本都采用矩阵运算。有些程序设计语言提供了各种矩阵运算的包，比如python，使用很方便。C++等中，一般使用二维数组来存储矩阵。

问题：实际应用中，矩阵中的元素数量多，行数和列数非常大，并且有出现大量相同的元素，或者很多为零的元素，全部存，会占用大量存储空间，无法进行运算？对这种特殊形状的矩阵，在这里我们介绍矩阵的压缩存储。所谓的压缩存储，是指对于相同的元素只存一次，而对于大量为0的元素，不进行存储，从而实现矩阵的压缩存储。

举个例子：在自然语言处理领域，计算词或句子俩俩间的相似度，由于词或句子众多，行数和列数是词或句子的个数，如果用一个全矩阵去存放俩俩间的相似度显然是不合适（比如16万个句子，每个相似度占用8个字节，这样需要18万*18万*8字节这么大的存储空间），而根据计算发现，很多相似度为0或者非常小，对于后面的文本分析意义不大，因此对于对于这些相似度为0或很小的值不进行存放。

下面分别介绍两种矩阵的压缩存储：

一是特殊矩阵：矩阵中的值相同元素或零元素分布具有一定的规律，比如说对称矩阵、上三角矩阵、下三角矩阵等。

一是稀疏矩阵：矩阵中有大量为0的元素。

2.1 特殊矩阵的压缩存储与实现

- 对称矩阵：是指以主对角线为对称轴，各元素对应相等的矩阵。在线性代数中，对称矩阵是一个方形矩阵，其转置矩阵和自身相等，a[i][j]=a[j][i]。例如：

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

- 上三角矩阵：主对角线以下都是零的方阵称为上三角矩。例如：

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

- 下三角矩阵：主对角线以上都是零的方阵称为上三角矩。例如：

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

对称矩阵、上（下）三角矩阵的存储方式一样，可以将将$n*n$个元素压缩存储到长度为$\frac{n(n+1)}{2}$的数组中,对称矩阵只存自对角线向下或向上的一般就可以，上（下）三角矩阵中为0的三角部分不存储。存储在一个长度为$\frac{n(n+1)}{2}$的一维数组中，对任意a[i][j]通过如下公式其在数组中的位置：

$$k = \begin{cases} \frac{i(i+1)}{2} + j - 1 & \text{i>=j} \\ \frac{j(j+1)}{2} + i - 1 & i < j \end{cases} \tag{2}$$

例如，对于如下矩阵5

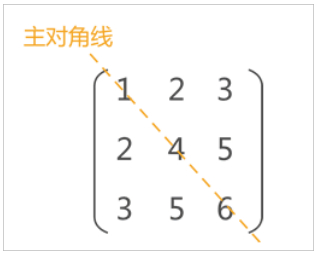

Figure 5: 对称矩阵示例

其存储示意图如下6所示：

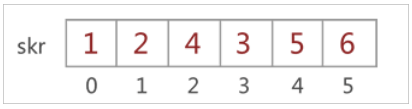

Figure 6: 存储示意图

对于元素a[3][1]，其存放位置通过计算可得：

$$k = \frac{i(i-1)}{2} + j - 1 = \frac{3(3-1)}{2} + 1 - 1 = 3$$

即存放在数组下标为3的存储空间中。

特殊矩阵的操作就是遍历一维数组，一般都是作为取值操作，相对比较简单，此处就不再介绍。

2.2 稀疏矩阵的压缩存储与实现

何为稀疏矩阵？0比较多的矩阵称为稀疏矩阵，“多”这个概念很模糊。一般来说通过稀疏因子来定义稀疏矩阵，一个m行n列的矩阵，其稀疏因子是$\delta = \frac{t}{(m*n)}$（其中t 为非0的元素个数），如果$\delta <= 0.05$该矩阵称为稀疏矩阵。当然在实际应用中，零占有一定的比例的，我们都可以用稀疏矩阵的压缩存储来进行存放。

对于稀疏矩阵的存储，主要有三种存储方式，分别是：

- 三元组顺序表
- 行逻辑链接的顺序表
- 十字链表

下面分别介绍这三种存储结构，以及在各存储结构下的操作的实现。

2.2.1 三元组顺序表

因为为零的元素不存放，只存放非零的元素，要存放一个非零元的元素a[i][j]，需要存储该非零元的以下基本信息：

- 行号
- 列号
- 该非零元素的值

除此之外还需要存储该矩阵的行数和列数，以及为了操作方便还存储该矩阵中非零元的个数。因此要描述一个非零元，需要三个信息，并且采用顺序存储结构进行存储，称为“三元组顺序表”。我们需要用两个结构来进行描述，一个是表示非零元的结构即三元组结构，一个是表示矩阵信息的结构。具体描述如下：

```c
#define MAXSIZE 12500
//三元组结构体
typedef struct {
        int row,col; //行标row，列标col
        Elemtype data;//元素值
}Triple;
```

```c
//矩阵的结构表示
typedef struct {
     Triple data[MAXSIZE+1];//存储所有非0元素的三元组
     int mu,nu,tu;//mu和nu分别记录矩阵的行数和列数，tu所有的非0元素的个数
}TSMatrix;
```

其中data[MAXSIZE+1]数组存放所有非零元素的三元组，存储时候按行依次存放，每行中按列号从小到大依次存放非零元素的三元组信息（行序为主序存储，一般都采用按行序为主序的存储）。

示例：下面图示(图7)稀疏矩阵利用三元组顺序表表示如图8所示：

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 5 \\ 3 & 0 & 0 \end{pmatrix}$$

Figure 7: 稀疏矩阵示例

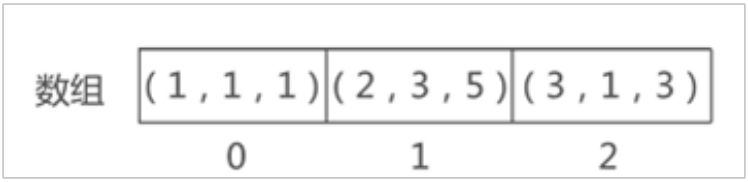

Figure 8: 存储示意图

下面我们介绍在三元组顺序表表示法下的矩阵转置运算，矩阵转置是一种常见的矩阵运算，即转置结果就是行变成列，列变成行，例如原矩阵M(m行n列)转置后的矩阵T是n行m列的矩阵，M[i][j]元素转置后在T中是T[j][i]。

示例：

$$M=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}$$

Figure 9: 转置前矩阵M

转置后矩阵T：

$$T=\begin{bmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 10: 转置前矩阵M

从存储示意图可以看到转置过程：

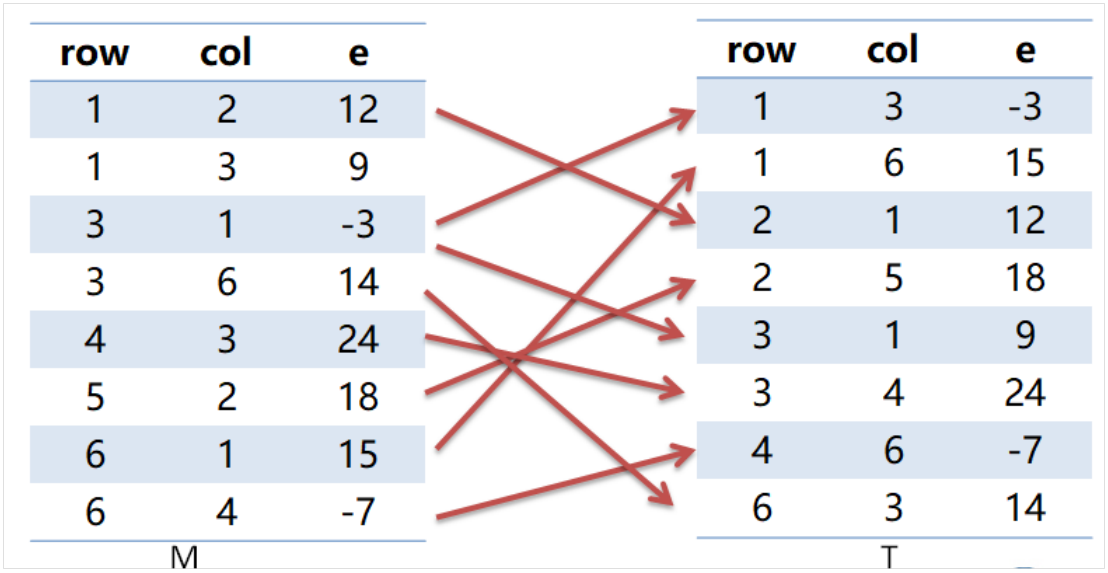

Figure 11: 转置过程示意图

注意的是，转置后的矩阵也必须是按行序为主序进行存放的。

算法思想如下：

Step1: 需要按照M的列序来进行转置,因为列转置后就是转置后矩阵T的;

Step2: 找到M中的每一列中所有非零的元素，因此按M的列进行扫描，需要多轮扫描，把每一列的元素非零元找出来：

每次都是从M的第一行依次扫描，而M是行序为主序存放的，因此从开始扫描找到的每一列非零的元素正好是转置矩阵T的顺序

具体地：

遍历列：for(col=1;*col* <= *M.col*;++col)

对每一列找其非零元素，遍历数组M：

如果当前列号和col相等，则找到，存入T中（转置）:

T.data[q].row=M.data[p].col;

T.data[q].col=M.data[p].row

T.data[q].e=M.data[p].e;

++q;

其中q是T的计数器，顺序记录转置元素的位置

算法描述如下：

```
Status transpose(TSMatrix M, TSMatrix &T)
{
    T.mu = M.nu;
    T.nu = M.mu;
    T.tu= M.tu;
    if(T.tu)
    {
        int q = 0;
        for (int col = 1; col <= M.nu; col++)
        {
            for (int p = 0; p < M.tu; p++)
            {
                if (M.data[p].col == col)
                {
                    T.data[q].row=M.data[p].col;
                    T.data[q].col = M.data[p].row;
                    T.data[q].data = M.data[p].data;
                    q++;
                }//End if
            }//End for
        }//End for
    }//End if
    return OK;
}
```

算法分析：以上算法的时间时间复杂度为:$O(nu*tu)$，其中nu是M的列数，tu为M非零的个数。如果tu和$nu*mu$同数量级，则时间复杂度为$O(mu*nu^2)$ 而一般矩阵的转置算法的时间复杂度为：$O(mu*nu)$（行列交换，只需要变量每行每列就可以），压缩存储的转置算法空间节省了，但转置的时间复杂度提高了。因此该算法使用于$tu << mu*nu$的情况。

上面的转置算法，是按列进行转置，对于每一列，需要遍历整个M的非零元数组，把改列每一行的元素找出来，这个是整个算法最耗时的地方，需要进行多轮的循环把每列的非零元找出来。那有没有更高效的转置算法

呢？我们一般在改进算法的时候，需要分析原算法可改进的地方，这样才有改进的方向。我们做一个这样假设，能不能按行来进行转置，如果需要按行来转置的话我们需要什么辅助的信息？如果能按行来进行转置，那么我们只需要扫描一遍M的非零元数组就可以实现转置，但是转置运算是行变成列，如果按行转置，需要知道每个元素在转置后矩阵T的位置，这样我们就可以按行转置，将每个元素取出放到它对应的位置上。为了达到这样的功能，在转置前需要知道每个元素转置后的位置。定义两个数组，一个数组num是存放M每一列非零元的个数，另外一个数组cpot存放每一列第一个非零元在T的存储数组data中的位置，这样我们很容易就可以得到每一个元素的位置。首先这两个数组很容易求得，只需要遍历一次M.data即可得，num的下标就是列号，这样就可以通过计数来得到。cpot的计算可以通过如下公式来计算，第1列第一个非零元在T中data的位置为1（注意：为了方便表达，0存储单元不用）即cpot[1]=1，那么第2列的第1个非零元的位置是：cpot[2]=cpot[1]+num[1],num[1]为第1非零元的个数，也就是说第i列的第一个非零元的位置是第i-1列第一个非零元的位置和第i-1列非零元的个数之和。即：

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

Figure 12: cpot计算公式

上例M的cpot和num数组的值如下所示：

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

Figure 13: M的cpot和num值

则，快速转置算法思想如下：

Step1：初始化num数组

for(col=1;$col <= M.nu$;++col) num[col]=0;

Step2：遍历M的三元组，统计每一列非0元素个数

for(t=0;$t < M.tu$;++t) ++num[M.data[t].col];

Step3：算每一列第一个非零元素在T的三元组存储的位置

cpot[1]=1;

for(col=2;*col* <= *M.nu*;++col)cpot[col]=cpot[col-1]+num[col-1]

Step4：遍历M的三元组进行转置，把每一个依次放到恰当的位置

col=M.data[p].col;

q=cpot[col];//当前元素存放的位置

T.data[q].row=M.data[p].col;

T.data[q].col=M.data[p].row;

T.data[q].e=M.data[p].e;

++cpot[col];

完整的算法描述如下：

```
Status transpose (TSMatrix M, TSMatrix &T)
{
    T.mu = M.nu;
    T.nu = M.mu;
    T.tu= M.tu;
    if (T.tu)
    {
        for(col=1;col<=M.nu;++col) num[col]=0;
        for(t=0;t<M.tu;++t) ++num[M.data[t].col];
        cpot[1]=1;
        for (int col = 1; col <= M.nu; col++)
            cpot[col]=cpot[col-1]+num[col-1];
        for(p=0;p<M.tu;++p)
        {
            q=cpot[col];
            col=M.data[p].col;
            T.data[q].row=M.data[p].col;
            T.data[q].col = M.data[p].row;
            T.data[q].e= M.data[p].e;
            ++cpot[coll];
        }//end for
```

```
22      }//end if
23      return OK;
24  }
```

算法分析：根据上面的算法描述可知，该算法的时间复杂度为：O(nu+tu)，当tu和mu*nu等数量级的时候，其时间复杂度也为O(nu*mu)，和经典转置算法相同。 但是，多用了两个数组，空间复杂度比算法1要多，但是时间复杂度降低了很多。利用空间换时间——这是常用来改进算法的一种方法，多存储一些信息，避免在算法过程去求得，从而增加时间复杂度。当然这里多的存储空间是可以接受的，因此这个快速的转置算法还是非常有效的。

2.2.2 行逻辑链接的顺序表

利用快速转置算法的改进思路——可以创建一个数组存放每行第一个非零元素的位置，这样就可以随机存取任意一行，可以在三元组顺序表的基础上增加“行”信息数组rpos，这样的存储结构称为带行逻辑连接的顺序表。

存储结构描述如下：

```
1  #define MAXSIZE 12500
2  //三元组结构体
3  typedef struct {
4        int row,col; //行标row , 列标col
5        Elemtype data;//元素值
6  }Triple;
```

```
1  //矩阵的结构表示
2  typedef struct {
3      Triple data[MAXSIZE+1];//存储所有非0元素的三元组
4      int   rpos[MAXRC+1];//各行第一个非0元素的位置
5      int mu,nu,tu;//mu和nu分别记录矩阵的行数和列数，tu所有的
           非0元素的个数
```

```
}RLSMatrix;
```

下面我们介绍在行逻辑链接下的顺序表存储结构下的两个稀疏矩阵乘法的实现方法。

两个矩阵M(m1*n1)和N(m2*n2)相乘，必须满足M的列数和N的行数相等(即：n1=m2)，得到的矩阵Q行数是M的行数，列数是N的列数,即Q是m1行n2列的矩阵。

首先回顾一下不采用压缩存储下的矩阵乘法的算法：

```
for(int i=1;i<=m1;++i)
{
    for(int j=1;j<n2;++j)
    {
        Q[i][j]=0;
        for(int k=1;k<=n1;++k)
        {
            Q[i][j] +=M[i][k]*N[k][j];
        }
    }
}
```

根据算法分析我们可知，该算法的时间复杂度是O(m1*n1*n2)。

下面我们看看稀疏矩阵的乘法的实现是如何进行的，首先看以下例子。

示例：

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

Figure 14: M和N

M和N乘积为：

$$Q = \begin{pmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{pmatrix}$$

Figure 15: 乘积Q

存储结构如下所示：

i	j	e
1	1	3
1	4	5
2	2	−1
3	1	2

M. data

i	j	e
1	2	2
2	1	1
3	1	−2
3	2	4

N. data

i	j	e
1	2	6
2	1	−1
3	2	4

Q. data

Figure 16: 存储结构

问题：如何通过对M和N的存储结构进行操作得到Q呢？

在稀疏矩阵的压缩存储中，只存储非零元，而矩阵乘法，如果有一个元素为0，则成绩肯定就为0，这种压缩存储下的矩阵乘法免去了这种无效的操作。由于相乘只是在非零元进行并且M的非零元的列号和N的非零元的行号相等才能进行相乘。并且对于乘积结果Q的某一个元素Q[i][j]而言，根据矩阵的乘法的运算是M的第i行的元素和N的第j列的元素对应元素的乘积累加而来。即：

$$Q[i][j] = \sum_{k=1}^{n1} M[i][k] * N[k][j]$$

在上例中，M 的(1,1,3)只需要和N的（1,2,2）相乘M的(1,4,5)不需要和N的任何元素相乘（N中无i=4的三元组）,但是M 的(1,1,3)只需要和N的（1,2,2）乘积只是Q[1][2]的一部分，因此可以将结果累加在一个临时变量上。

算法思想如下：

Step1:依次遍历M的每一行，取出M当前行col中的每一个元素(i,k,M(i,k))，用指针p指向;

Step2:对于当前M的元素p,其列号为k，在N.data中找出所有行等于k的元素(k,j,N(k,j))相乘（M.data[p].col==N.data[q].row），由于采用带行逻辑链接的顺序表存储，因此数组N.rpos 存储了每一行第一个非零元的位置，即：N.rpos[row]表示第row行第一个非零元素的位置，N.rpos[row+1]-1则为第row行最后一个非零元的位置;

Step3:遍历N当前行的每一个非零元（q指针指向），对于满足M.data[p].col==N.data[q].row进行相乘M.data[p].e*N.data[q].e乘积只是Q[i][j]的一部分，需要定义变量进行求累计和。定义ctemp 数组，初始化为0，ctemp其下标当前q指针指向元素的列号，将乘积累加到ctemp中。

Step4:N当前行的循环结束，则ctemp中存的值就是Q的第col行的元素，由于乘积求和后可能为0，因此需要对ctemp压缩存储到Q中。依次遍历N的列col，如果当前ctemp[col]的元素不为零，则将将(row,col,ctemp[col])放入到Q中，放入顺序就是Q的存储顺序。

Step5:重复Step1 Step4，直到M的每一行都遍历完。

算法描述如下：

```
Status mulMatrix(RLSMatrix M,RLSMatrix N,RLSMatrix &Q)
{
    //M的列数和N的行数不相等，则返回错误
    if(M.nu!=N.mu) return ERROR;
    //初始化Q
    Q.mu=M.mu;Q.nu=N.nu;Q.tu=0;
    if(M.tu*N.tu!=0){
      //处理M的每一行
      for(arow=1;arow<=M.mu;++arow){
          //当前行各累加器为0
          ctemp[]=0;
          //记录每行第一个非零元位置,Q.tu+1表示当前元素个数
          Q.rpos[arow]=Q.tu+1;
          //tp指向M的下一行第一个元素的位置
          if(arow<M.mu)
              tp=M.rpos[arow+1];
          else //arow到达最后一行，指向最后一个元素的末尾
```

```
18                  tp=M.tu+1;
19              //对当前行中每一个非零元
20              for(p=M.rpos[arow];p<tp;++p)
21              {
22                  //找到对应元素在N中的行号
23                  brow=M.data[p].col;
24                  //t指向N的brow+1行的第一个元素位置
25                  if(brow<N.nu)
26                      t=N.rpos[brow+1];
27                  else //t指向N最后一个元素的下一个位置
28                      t=N.tu+1;
29                  //对N的当前行brow的每一个元素进行相乘
30                  for(q=N.rpos[brow];q<t;++q){
31                      //乘积元素在Q中的列号
32                      ccol=N.data[q].col;
33                      //相乘并累加ccol列
34                      ctemp[ccol]+=M.data[p].e*N.data[q].e;
35                  }//end for(q)
36              }//end for(p)
37              //求得Q中第crow(=arow)行的非零元
38              for(ccol=1;ccol<=Q.nu;++ccol){
39                  //压缩存储该行的非零元
40                  //非零才去存储，为0的元素不存储
41                  if(ctemp[ccol]){
42                      if(++Q.tu>MAXSIZE) return ERROR;
43                      //三元组存放进去Q
44                      Q.data[Q.tu]=(arow,ccol,ctemp[ccol]);
45                  }//end if
46              }//end for(ccol)
47          }//end for(arrow)
48      }//end if
49      return OK;
50  }
```

算法分析：

(1)ctemp数组初始化的时间复杂度：$O(M.mu * N.nu)$

(2)求Q非零元的时间复杂度：$O(M.tu * N.tu/N.mu)$

(3)Q的压缩存储时间复杂度：$O(M.mu * N.nu)$

则，总的时间复杂度：$O(M.mu * N.nu + M.tu * N.tu/N.mu)$

若M为m*n矩阵,N为n*p矩阵,而$M.tu = \delta_M * m * n$,$N.tu = \delta_N * n * p$，则时间复杂度为：$O(m * p * (1 + n * \delta_M \delta_N))$，当$\delta_N < 0.05$, $\delta_M < 0.05$，并且$n < 1000$时候，时间复杂度相当于$O(m * p)$。

2.2.3 十字链表

前面两种存储结构是顺序存储结构，一般的操作不改变非零元的个数和位置，而稀疏矩阵的有些操作会改变非零元的个数或位置，例如矩阵的加法，这个时候利用顺序存储结构来存储，在运算的过程中会进行大量的移动元素的操作，因此此时采用顺序存储结构不是很理想。和线性表一样，矩阵也有链式存储结构，但是矩阵有其特殊性，矩阵中的元素如果从行的角度来看它有前驱和后继（除第一个和最后一个外），从列的维度来看也有前驱和后继（也除第一个和最后一个外），因此对于一个元素既处于行的链表上，也处于列的链表上。这种存储结构称为十字链表。对于矩阵中的每一个非零元素需要定义一个结点结构，存储其行号和列号以及非零元的值，同时要分别定义两个指针，一个指针指向其行链表上的后继结点，一个指针指向其列方向上的后继。另外还要声明链表的结构，分别存储各行的头指针，各列的头指针，以及行数和列数以及非零元的个数。具体的存储描述如下所示：

```
#define MAXSIZE 12500
//三元组结构体
typedef struct  Node{
        int row,col; //行标row，列标col
        Elemtype data;//元素值
        struct Node *right, *down;//right指向行方向上的后继，
            down指向列方向的后继
}Node, *OLink;
```

```
//链表结构
typedef struct {
        OLink *rhead, *chead;//行列头指针向量
        int mu,num,tu;    //行数，列数和非零元个数
}CrossList;
```

示例：下面的矩阵17的存储示意图18

$$M = \begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

Figure 17: 稀疏矩阵示例

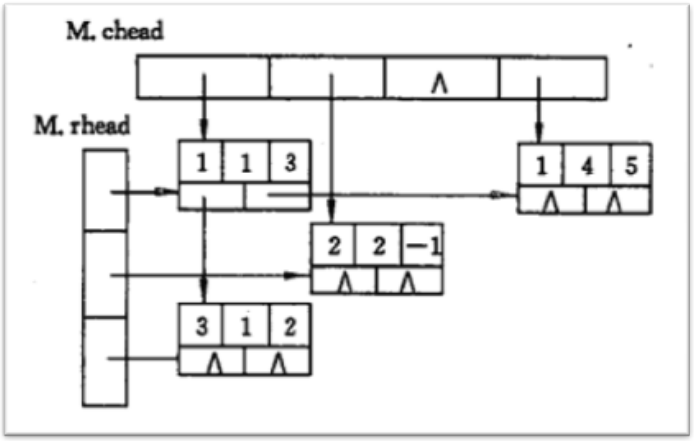

Figure 18: 十字链表示意图

下面介绍创建一个稀疏矩阵的十字链表存储结构的算法，这里通过输入稀疏矩阵的信息来创建，首先输入稀疏矩阵的行数和列数以及非零元的个数，然后输入非零元的信息（行号、列号、非零元的值）。创建十字链表的过程，实际上是将生成的结点根据其行号和列号，分别连接在其行的链

表上及列的链表上，分别在这两个列表中找到插入位置，将新生成的结点插入上去即可。

算法思想如下：

Step1: 根据输入的行数、列数和非零元的个数初始化M的链表结构；

Step2:分别创建行头指针和列头指针

Step3:输入非零元，生成结点并初始化

Step4:在行链表中找到插入位置，插入

Step5:在列链表中找到插入位置，插入

Step6:重复Step3至Step5，直到输入结束。

算法描述如下：

```
Status createMatrix(CrossList &M){
        if(M) free(M);
        cin>>m>>n>>t;
        M.mu=m;m.nu=n;M.tu=t;
        if(!(M.rhead=(OLink*)malloc((m+1)*sizeof(OLink))
            ))exit(OVERFLOW);
        if(!(M.chead=(OLink*)malloc((n+1)*sizeof(OLink))
            ))exit(OVERFLOW);
        M.rhead[]=M.chead[]=NULL;//头指针向量初始化
        for(cin>>i>>j>>e;i!=0;cin>>i>>j>>e){
           if(!(p=(Node*)malloc(sizeof(Node))))exit(
              OVERFLOW);
            p->row=i;p->col=j;p->e=e;//生成结点并初始化
            if(M.rhead[i]==NULL||M.rhead[i]->col>j){  //插
               入行表为第一个结点
                  p->right=M.rhead[i];M.rhead[i]=p;
            }// end if
            else{//查找行表的位置
                for(q=M.rhead[i];(q->right)&&q->right->col
                   <j;q=q->next);
                p->right=q->right;q->right=p;//完成行插入
            }//end else
```

```
18              if(M.rhead[j]==NULL||M.rhead[j]->row>i){//插入
                    列表为第一个结点
19                  p->down=M.rhead[j];M.rhead[j]=p;
20              }
21              else{//查找列表的位置
22                  for(q=M.chead[j];(q->down)&&q->down->row<i
                        ;q=q->down);
23                  p->down=q->down;q->down=p;//完成列插入
24              }//end else
25      }//end for
26      return OK;
27  }//End
```

读者自行去分析和实现两个稀疏矩阵的加法的算法。

3 广义表

数组可以存储原子类型，也可以存储数组（二维数组），但是这两种类型不能同时在一个数组里面。比如说：1,2,3、4,5，7,8是合理的数组，而1,2,3,4是不允许的。在实际应用，有一些结构既可以存储不可再分的元素，也可以存储结构。这里介绍一种称为广义表的结构，它也是线性结构，是线性表的扩展。

3.1 广义表的定义

广义表里面既可以存放原子类型，也可以存放广义表。记为：$LS = (a_1, a_2, ..., a_n)$,$a_i$既可以代表单个元素，也可以代表另一个广义表。

广义表示例：

(1)A = ()：A 表示一个广义表，是空表。

(2)B = (e)：只有一个原子 e

(3)C = (a,(b,c,d)) ：两个元素，原子 a 和子表 (b,c,d)。

(4)D = (A,B,C)：三 个子表，分别是A、B和C。等同于 D = ((),(e),(b,c,d))。

(5)E = (a,E)：两个元素（原子 a 和它本身），递归广义表，等同于：E = (a,(a,(a,...)))

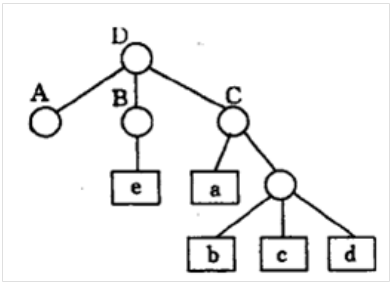

Figure 19: 广义表示例

定义 1. *表头：当广义表不是空表时，称第一个数据（原子或子表）为表头。*

定义 2. *表尾：除第一个外，剩下的数据元素构成的新广义表为广义表的表尾。*

示例1：LS=(1,(1,2,3),5)
表头为原子 1，表尾为子表(1,2,3) 和原子 5 构成的广义表，即 ((1,2,3),5)。

示例2：LS = (1)
表头为原子 1 ，该表的表尾是一个空表，用()表示。
自己动手求一求：上面的广义表示例(1) (5)的表头和表尾。

3.2 广义表的存储结构

由于广义表中既可存储原子，也可以存储子表，很难使用顺序存储结构表示，因为使用顺序表存储，不仅需要操作 n 维数组（例如 1,2,3,4 就需要使用三维数组存储），还会造成存储空间的浪费。一般广义表的存储结构采用链表实现。

由于广义表里面存着两种类型，一种是原子类型，一种是广义表类型，因此需要定义两种结点来存储。一种是原子结点，存放原子类型的元素，另一种是广义表结点，存放广义表。为了区分两种结点，一般设定tag标记，tag=0为原子结点，tag=1为广义表结点。原子结点分为tag域和数据域。广义表分为表头和表尾，因此广义表结点分为三个域，一个是tag域，一个是hp指针指向表头，一个tp指针指向表尾。具体的结点结构描述如下：

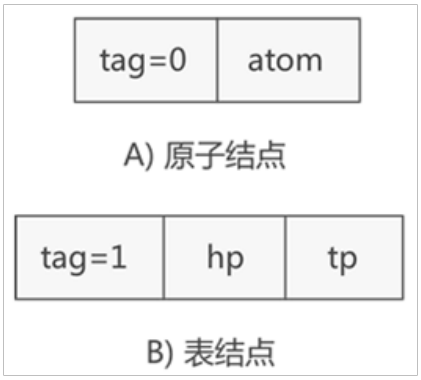

Figure 20: 广义表结点结构

```
typedef struct GLNode{
int tag;//标志域
union{
        ElemType atom;//原子结点的值域
        struct{
                struct GLNode * hp,*tp;
        }ptr;//子表结点的指针域，hp指向表头；tp指向表尾
};
}*Glist;
```

*C/C++*知识点：*union*共用体，也叫联合体，在一个“联合”内可以定义多种不同的数据类型， 一个被说明为该“联合”类型的变量中，允许装入该“联合”所定义的任何一种数据，这些数据共享同一段内存，以达到节省空间的目的。*union*变量所占用的内存长度等于最长的成员的内存长度。

示例： 广义表(a,(b,c,d)) 是由一个原子 a 和子表 (b,c,d) 构成，而子表 (b,c,d) 又是由原子 b、c 和 d 构成。其存储结构示意图如图21所示。

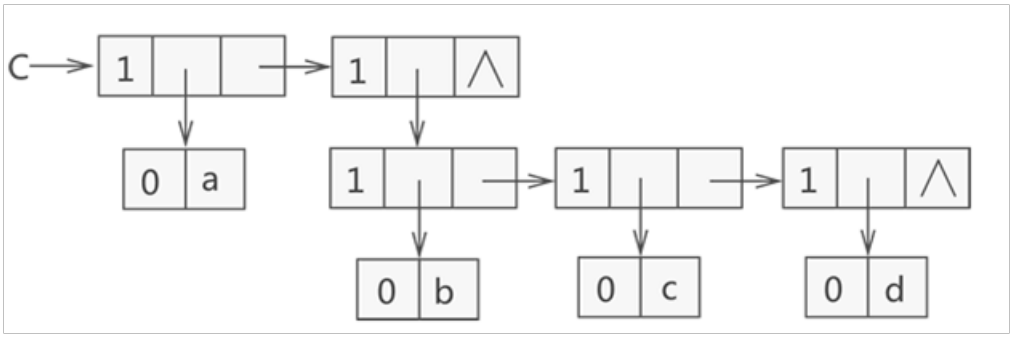

Figure 21: 存储结构示意图

动手试一试：动手画一下上面广义表示例(1) (5)的存储结构示意图

广义表的另一种存储结构：原子的节点构成由 tag 标记位、原子值和 tp 指针构成；子表的节点还是由 tag 标记位、hp 指针和 tp 指针构成。存储结构描述如下：

```
typedef struct GLNode{
int tag;//标志域
union{
        ElemType atom;//原子结点的值域
                struct GLNode *hp;//子表结点的指针域，hp指向表头
};
struct GLNode * tp;//这里的tp相当于链表的next指针，用于指向下一个数据元素
}*Glist;
```

Figure 22: 另一种存储结构

示例：广义表 a,b,c,d的存储示意图如图23

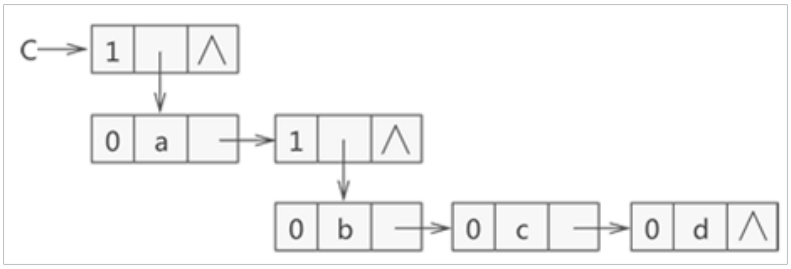

Figure 23: 存储结构示意图

动手试一试：动手画一下上面广义表示例(1) (5)的第二种存储结构示意图

注意：无论哪一种存储结构存储广义表，都不能破坏广义表中各数据元素之间的并列关系。如 (a,(b,c,d)) ，原子 a 和子表 (b,c,d) 是并列的，而在子表 (b,c,d) 中原子 b、c、d 是并列的。

3.3 广义表的操作

广义表的定义是一个递归的定义，每一个广义表中元素可以是原子类型或广义表类型，并且存储结构分为原子结点和子表结点，子表结点又指向表头和表尾，因此广义表的操作一般都是采用递归的方法。下面分别介绍几个广义表的操作。

3.4 求广义表的深度

广义表的深度，可以通过该表中所包含括号的层数间接得到。先看两个例子：

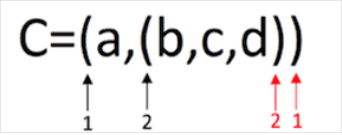

Figure 24: 广义表深度示例

上图24的深度为2。

例2：L=((1,2),(3,(4,5)))——深度为3，因为(1,2)与(3,(4,5))是同一层。

求广义表的深度可以采用递归的思想来实现，依次遍历广义表 C 的每个节点，若当前节点为原子（tag 值为 0），则返回 0；若为空表，则返回 1；反之，则继续遍历该子表中的数据元素。可以设置一个初始值为 0 的整形变量 max，每次递归过程返回时，令 max 与返回值进行比较，并取较大值。这样，当整个广义表递归结束时，max+1 就是广义表的深度。广义表的存储结构采用第一种存储结构。算法实现如下：

```
int GlistDepth(Glist C)
{
    if (!C) {//如果表C为空表时，直接返回长度1;
        return 1;
    }
    if (C->tag==0) {//如果表C为原子时，直接返回0;
        return 0;
    }
    int max=0;//设置表C的初始长度为0;
    for (Glist pp=C; pp; pp=pp->ptr.tp)
    {
        int dep=GlistDepth(pp->ptr.hp);
        if (dep>max) {
            max=dep;//每次找到表中遍历到深度最大的表，并
                用max记录
```

```
15            }
16        }
17        return max+1; //非空表的深度都是在各元素的深度的最大值
              加1;
18    }
```

3.5 求广义表的长度

广义表中存储的每个原子算一个数据，同样每个子表也只算作是一个数据。例如：a,b,c,d 中，它包含一个原子和一个子表，因此该广义表的长度为 2。求长度非常简单，只需要验证tp指针一直遍历就可以得到长度。

```
1  //求广义表的长度
2  int GlistLength(Glist C){
3      int len=0;
4      Glist p=C;
5      while(p){
6          ++len;
7          p=p->ptr.tp; //指向下一个元素
8      }
9      return len;
10 }
```

3.6 复制广义表

任意一个非空广义表来说，都由表头和表尾组成。反之，只要确定的一个广义表的表头和表尾，那么这个广义表就可以唯一确定下来。复制一个广义表，也是不断的复制表头和表尾的过程。如果表头或者表尾同样是一个广义表，依旧复制其表头和表尾。递归的出口有两个：如果当前遍历的数据元素为空表，则直接返回空表;如果当前遍历的数据元素为该表的一个原子，那么直接复制，返回即可。

```
1  void copyGlist(Glist C, Glist *T){
```

```
    if (!C)
    {//如果C为空表，那么复制表为空表
        *T=NULL;
    }
    else
    {
        //C不是空表，给T申请内存空间
        *T=(Glist)malloc(sizeof(GNode));
        if (!*T) exit(0); //申请失败，退出
        (*T)->tag=C->tag;//复制表C的tag值
        if (C->tag==0)
        {//如果是原子，直接复制
            (*T)->atom=C->atom;
        }
        else
        {//复制的是子表，递归调用
            copyGlist(C->ptr.hp, &((*T)->ptr.hp));//复制表头
            copyGlist(C->ptr.tp, &((*T)->ptr.tp));//复制表尾
        }
    }
}
```

无论实现什么操作，都需要首先建立好广义表的存储结构，广义表的存储结构也是采用递归的思想来建立，读者自行去实现广义表存储结构建立操作。

m元多项式的表示与运算是广义表的应用之一，感兴趣的读者可以自行去设计广义表的结构来表示m元多项式，并实现m元多项式的一些基本运算等。

4 小结

本章主要介绍了数组和广义表两种数据线性结构，具体的内容：

- 数组的定义与存储实现：介绍了数组的定义，以及数组的存储表示及

其实现。

- 特殊矩阵的压缩存储与实现：介绍了对称矩阵的压缩存储与实现，将对称矩阵压缩存储在一个一维数组里面。
- 稀疏矩阵的压缩存储与操作的实现：介绍了三种压缩存储方法分别是三元组顺序表、行逻辑链接顺序表、十字链表，以及分别介绍了各存储结构下矩阵的转置、相乘等算法的实现，并对算法进行了算法分析。
- 介绍了广义表的定义、存储及操作：广义表由表头与表尾构成，既可以存原子也可以存广义表。介绍了两种存储结构，并在第一种存储结构下介绍了求广义表的深度、长度、复制广义表等操作及实现。

5 练习题

一、填空题：

1. 已知二维数组A[m][n]采用行序为主序的存储结构，每个元素占用L个存储单元，第一个元素的地址为loc，那么第A[i][j]的地址为_____。

2.一个10阶的对称矩阵A，第一个元素的地址为1（采用行序为主序的存储），那么A[8][5]的存储单元为：_____。

3.稀疏矩阵的压缩存储一般可以分为两种，分别是_____、_____。

4.一般对数组的操作有_____、_____。

5.广义表L=(a,(b,c,(d)),(e,f))的表头是_____，表尾是_____，深度为_____，长度为_____。

二、画图题：

1.稀疏矩阵如图25所示，画出

(1)三元组表示法

(2)行逻辑链接的表示法

(3)十字链表

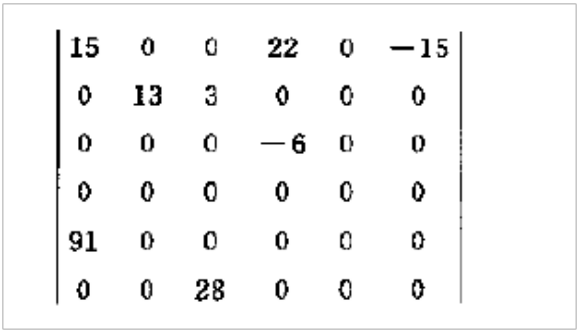

Figure 25: 矩阵1

2. 画出广义表L=(a,(b,c,(d)),(e,f))两种存储的结构示意图

三、编程题

1. 两个稀疏矩阵采用十字链表存储，实现两个稀疏矩阵的加法：C=A+B。

2.实现建立广义表的存储结构（以第一种为例），通过键盘输入广义表。

6 Next

下一章我们介绍第一种非线性结构——树形结构，主要介绍树的定义、二叉树的定义与存储结构及遍历操作、树的存储结构、树与二叉树的转换、哈夫曼编码等应用。树和二叉树是非常重要的数据结构，也是本门课程的重点。

7 附录

7.1 数组代码实现

```
1 //Array.h文件
2 #pragma once
3 #include <stdarg.h>
4 #include "predefine.h"
5 #define MAX_ARRAY_DIM 8
```

```
6  class Array
7  {
8  public:
9          Array();
10         Array(int dim,...);
11         ~Array();
12         Status initArray(int dim, ...);
13         Status destroyArray();
14         Status value(int *e, ...);
15         Status assign(int e, ...);
16 protected:
17         Status locate(va_list ap, int *off);
18 private:
19         int *base; //数组元素基址，由InitArray分配
20         int dim;                    //数组维数
21         int *bounds;          //数组维界基址，由InitArray分配
22         int *constants;       // 数组映象函数常量基址，
               由InitArray分配
23 };
24
25 //Array.cpp
26 #include "Array.h"
27
28
29
30 Array::Array()
31 {
32 }
33 Array::Array(int dim, ...)
34 {
35         int elemtotal = 1; //elemtotal是元素总值
36         va_list vap;
37
38         if (dim<1 || dim>MAX_ARRAY_DIM)
```

```
39                  exit(ERROR);
40          dim = dim;
41          bounds = (int*)malloc(dim *sizeof(int));
42          if (!bounds)
43                  exit(OVERFLOW);
44
45          va_start(vap, dim);
46          for (int i = 0; i < dim; ++i)
47          {
48                  bounds[i] = va_arg(vap, int);
49                  if (bounds[i] < 0)
50                          exit(UNDERFLOW);
51                  elemtotal *= bounds[i];
52          }
53          va_end(vap);
54          base = (int *)malloc(elemtotal * sizeof(int));
55          if (!base)
56                  exit(OVERFLOW);
57          constants = (int *)malloc(dim * sizeof(int));
58          if (!constants)
59                  exit(OVERFLOW);
60          constants[dim - 1] = 1;
61          for (int i = dim - 2; i >= 0; --i)
62          {
63                  constants[i] = bounds[i + 1]*
                        constants[i + 1];
64          }
65  }
66
67  Array::~Array()
68  {
69          //销毁数组arr
70          if (base)
```

```
71          {
72                  free(base);
73                  base = NULL;
74          }
75          if (bounds)
76          {
77                  free(bounds);
78                  bounds = NULL;
79          }
80          if (constants)
81          {
82                  free(constants);
83                  constants = NULL;
84          }
85  }
86  Status Array::locate(va_list ap, int *off)
87  {
88          //若ap指示的各下标值合法，则求出该元素在arr中的相对地
              址off
89          int i, ind;
90          *off = 0;
91          for (i = 0; i < dim; i++)
92          {
93                  ind = va_arg(ap, int);
94                  if (ind < 0 || ind >= bounds[i])
95                          return OVERFLOW;
96                  *off += constants[i] * ind;
97          }
98          return OK;
99  }
100 Status Array::value(int *e,...)
101 {
102         //依次为各维的下标值，若各下标合法，则e被赋值为A的相应
              的元素值
```

```
103         va_list ap;
104         Status result;
105         int off;
106         va_start(ap, e);
107
108
109         if ((result = locate(ap,&off)) == OVERFLOW) //
                调
                用locate()
110                 return result;
111     va_end(ap);
112         *e = *(base + off);//赋值
113         return OK;
114 }
115 Status Array::assign(int e, ...)
116 {
117         //依次为各维的下标值，若各下标合法，则将e的值赋给arr指定
                的元素
118         va_list ap;
119         Status result;
120         int off;
121         va_start(ap, e);
122         if ((result = locate(ap, &off)) == OVERFLOW)
                //调
                用Locate()
123                 return result;
124         va_end(ap);
125         *(base + off) = e;
126         return OK;
127 }
128
129 //测试文件:
130 #include "Array.h"
131
132 using namespace std;
```

```
133
134 int main()
135 {
136         int bound1 = 3;
137         int bound2 = 2;
138         int bound3 = 2;
139         Array arr(3, bound1, bound2, bound3);
140         int e;
141         for (int i = 0; i < bound1; i++)
142         {
143                 for (int j = 0; j < bound2; j++)
144                 {
145                         for (int k = 0; k < bound3; k
                            ++)
146                         {
147                                 arr.assign(i * 100 + j
                                    * 10 + k, i, j, k)
                                    ; // 将i*100+j*10+k赋
                                    值给arr[i][j][k]
148                                 arr.value(&e, i, j, k)
                                    ; //将A[i][j][k]的值赋
                                    给e
149                                 printf("arr[%d][%d][%d
                                    ]=%2d ", i, j, k, e
                                    ); //输
                                    出A[i][j][k]
150                         }
151                         cout << endl;
152                 }
153                 cout << endl;
154         }
155
156         //cout ¡¡ e ¡¡ endl;
157         system("pause");
158         return 0;
```

```
159 }
```

7.2 稀疏矩阵代码实现

```
1
2  //TSMatrix.h
3  #pragma once
4  #include "predefine.h"
5  struct Triple
6  {
7          int row, col;
8          int data;
9  };
10 class TSMatrix
11 {
12 public:
13         TSMatrix();
14         TSMatrix(int, int, int,Triple arr[]);
15         ~TSMatrix();
16
17 public:
18         Status transpose(TSMatrix &T);
19         void print();
20 private:
21     Triple data[MAX+1];//存储所有非0元素的三元组
22     int mu,nu,tu;//mu和nu分别记录矩阵的
23 };
24
25 #include "TSMatrix.h"
26
27
28
```

```
29 TSMatrix::TSMatrix()
30 {
31 }
32 TSMatrix::TSMatrix(int m, int n, int t, Triple arr[])
33 {
34         mu = m;
35         nu = n;
36         tu = t;
37         for (int i = 0; i < t; ++i)
38         {
39                 data[i] = arr[i];
40         }
41 }
42
43 TSMatrix::~TSMatrix()
44 {
45 }
46
47 Status TSMatrix::transpose(TSMatrix &T)
48 {
49         T.mu = nu;
50         T.nu = mu;
51         T.tu = tu;
52         if (T.tu)
53         {
54                 int q = 0;
55                 for (int col = 1; col <= nu; col++)
56                 {
57                         for (int p = 0; p < tu; p++)
58                         {
59                                 if (data[p].col == col
                                    )
60                                 {
```

```
61                                          T.data[q].row
                                                = data[p].
                                                col;
62                                          T.data[q].col
                                                = data[p].
                                                row;
63                                          T.data[q].data
                                                 = data[p].
                                                data;
64                                          q++;
65                                      }//End if
66                                  }//end for
67                              }//end for
68                      }//end if
69                      return OK;
70  }
71
72  void TSMatrix::print()
73  {
74          for (int i = 1; i <= nu; i++) {
75                  for (int j = 1; j <= mu; j++) {
76                          int value = 0;
77                          for (int k = 0; k < tu; k++) {
78                                  if (i == data[k].row
                                     && j == data[k].col
                                     ) {
79                                          cout<<data[k].
                                              data<<" ";
80                                          value = 1;
81                                          break;
82                                  }
83                          }
84                          if (value == 0)
```

```
85                                      cout<<0<<" ";
86                  }
87                  cout << endl;
88          }
89  }
90  //测试文件：MatrixDemo.cpp
91  #include "TSMatrix.h"
92
93  using namespace std;
94
95  int main()
96  {
97          Triple t1;
98          t1.row = 1;
99          t1.col = 1;
100         t1.data = 1;
101         Triple t2;
102         t2.row = 2;
103         t2.col = 3;
104         t2.data = 5;
105         Triple t3;
106         t3.row = 3;
107         t3.col = 1;
108         t3.data = 3;
109         Triple data[] = { t1,t2,t3 };
110
111         TSMatrix ts(3, 3, 3, data);
112
113         ts.print();
114         cout << "转置后:" << endl;
115         TSMatrix t;
116         ts.transpose(t);
117         t.print();
```

```
118         system("pause");
119         return 0;
120 }
```