第 5 章 数组和广义表

CHAPTER 5

主要知识点

- 数组和广义表的类型定义。
- 数组的顺序表示以及基于顺序表的基本操作实现。
- 矩阵的压缩存储以及基于存储结构的基本操作实现。
- 广义表的链式表示以及基于链表的基本操作实现。

数组和广义表是两种广义的线性表。在线性表、栈、队列及串的讨论中，不难看出元素都是原子类型，元素的值是不再分解的。而数组和广义表可以看成是线性表在下述含义上的扩展：表中的元素本身也是一个数据结构。大多数程序设计语言都可使用数组来描述数据，其他数据结构的顺序存储结构也多是以数组形式来描述的。广义表在文本处理、人工智能及计算机图形学等领域得到广泛应用，且使用价值和应用效果逐渐受到人们重视。在 LISP 和 PROLOG 程序中，所有的概念和对象都是用广义表表示的。

5.1 数组

从逻辑结构上看，多维数组是一维数组的扩充；但从存储结构上看，一维数组是多维数组的特例。在程序设计语言中，重点是数组使用，而本节重点是数组的内部实现，即如何在计算机内部处理数组，其中主要问题是数组的存储结构与寻址方法。

5.1.1 数组的类型定义

1. 数组的定义

数组(Array)是由下标与值组成的数偶的有序集合，即它的每个元素是由一个值与一组下标所确定的。对于每组有定义的下标总有一个相应的数值与之对应，且这些值都是同一类型的。下标决定了元素的位置，数组中各元素之间的逻辑关系由下标体现出来，下标的个数决定了数组的维数。因为由下标所决定的位置之间的关系可以看成是一种有序的线性关系，因此可以说数组是有限个相同类型数据元素组成的有序序列。从这个角度看，数组是线性表的推广，其逻辑结构实际上是一种线性结构。

1) 一维数组

一维数组($a_0, a_1, \cdots, a_{n-1}$)由 n 个元素组成，每个元素除具有相同类型的值外，还有一个下标以确定元素的位置，显然一维数组就是一个线性表。一维数组在计算机内是存放在一

块连续的存储单元中，适合于随机查找。

2）二维数组

二维数组由 m×n 个元素组成，元素之间是有规则的排列。每个元素由相同类型的值及一对能够确定元素位置的下标组成。二维数组可以看成是一维数组的推广。

例如，设 A 是一个有 m 行 n 列的二维数组，如图 5-1(a)所示。其中，每个元素是一个列向量形式的线性表，如图 5-1(b)所示；或者是一个行向量形式的线性表，如图 5-1(c)所示。因此，二维数组可以看成是有 m 个(或 n 个)元素的特殊线性表，其元素为一维数组。

$$A_{m\times n}=\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

(a) 二维数组 A

$$A_{m\times n}=\begin{bmatrix} \begin{bmatrix} a_{00} \\ a_{10} \\ \vdots \\ a_{m-1,0} \end{bmatrix} & \begin{bmatrix} a_{01} \\ a_{11} \\ \vdots \\ a_{m-1,1} \end{bmatrix} & \cdots & \begin{bmatrix} a_{0,n-1} \\ a_{1,n-1} \\ \vdots \\ a_{m-1,n-1} \end{bmatrix} \end{bmatrix}$$

(b) 二维数组 A 的列向量是一个线性表

$$A_{m\times n}=((a_{00}\,a_{01}\cdots a_{0,n-1}),(a_{10}\,a_{11}\cdots a_{1,n-1}),\cdots,(a_{m-1,0}\,a_{m-1,1}\cdots a_{m-1,n-1}))$$

(c) 二维数组 A 的行向量是一个线性表

图 5-1　二维数组是线性表的推广

3）多维数组

n 维数组的每个元素由相同类型的值及 n 个能确定元素位置的下标组成，按数组的 n 个下标变化次序关系的描述，可以确定数组元素的前驱和后继关系并写出对应的线性表。

n 维数组也可以由元素为(n－1)维数组的特殊线性表来定义，这样维数大于一的多维数组是由线性表结构辗转合成得到的，是线性表的推广。

2. 数组的抽象数据类型

数组一旦被定义，它的维数和维界就不再改变。因此，除了结构初始化和销毁外，数组通常只有两种基本运算：

(1) 存取：给定一组下标，存取相应的数组元素。

(2) 修改：给定一组下标，修改相应数据元素中的某一个或某几个数据项的值。

```
ADT Array {
    Data:
        j = 0, … , bi - 1, i = 1, 2, … , n
        D = {a_{j1j2…jn} | a_{j1j2…jn} ∈ ElemSet,
            其中：n(n > 0)是数组维数，b_i 是数组第 i 维长度，j_i 是数组元素第 i 维下标}
    Relation:
        R = {R1, R2, … , Rn}
        Ri = {< a_{j1j2…jn}, a_{j1j2…jn} > |
            0≤j_k≤b_k - 1, 1≤k≤n 且 k≠i, 0≤j_i≤b_i - 2, a_{j1j2…jn}, a_{j1j2…jn} ∈ D, i = 2, … , n}
    Operation:
        InitArray(&A, dim, bound1, … , boundn)
```

初始条件：无。
操作结果：若维数 dim 和各维长度 bound1，…，boundn 合法则构造相应的 n 维数组 A。
DestroyArray(&A)
初始条件：n 维数组 A 已经存在。
操作结果：销毁数组 A。
ValueArray(A,&e,index1，…，indexn);
初始条件：n 维数组 A 已经存在。
操作结果：若给定的各下标值 index1，…，indexn 不超界，则用 e 返回给定下标值所指定的 A 的元素值。
AssignArray(&A,e,index1，…，indexn);
初始条件：n 维数组 A 已经存在。
操作结果：若给定的各下标值 index1，…，indexn 不超界，则将 e 的值赋予给定下标值所指定的 A 的元素。
} ADT Array

5.1.2 数组的顺序表示及操作实现

数组一般不做插入或删除操作。也就是说，一旦建立了数组，结构中的元素个数和元素之间的关系就不再发生变动。因此，数组一般采用顺序存储的方式。

1. 数组顺序表的定义

把数组中的元素按照逻辑次序依次存放在一组地址连续的存储单元里的方式称为数组的顺序存储结构，采用这种存储结构的数组称为数组顺序表(Array Sequential List)。

1) 一维数组的顺序存储

一维数组(a_0，a_1，…，a_{n-1})由 n 个元素组成，如果数组的每个元素占 L 个存储单元且从地址 A 开始依次分配数组各元素，则数组 A 中任一元素 a_i 的存储地址可以由式(5-1)表示，其分配情况如图 5-2 所示。

$$LOC(a_i)=LOC(a_0)+i\times L \tag{5-1}$$

其中：$LOC(a_0)$是 a_0 的存储地址，即一维数组 A 的基地址。

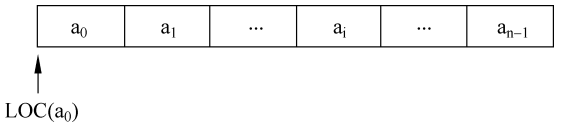


图 5-2 一维数组存储分配

2) 二维数组的顺序存储

二维数组的每个元素含有两个下标，需要将二维关系映射为存储器的一维关系。常用的映射方法有两种：按行优先和按列优先。例如，C/C++中的数组采用的是按行优先存储方式，FORTRAN、PASCAL 中的数组采用的是按列优先存储方式。

(1) 按行优先存储方式

按行优先存储方式是指先行后列，先存储行号较小的元素，行号相同者先存储列号较小的元素。如果每个元素占 L 个存储单元且从地址 A 开始依次分配数组中各元素，则数组 A 中任一元素 a_{ij} 的存储地址可以由式(5-2)表示，其分配情况如图 5-3 所示。

$$LOC(a_{ij})=LOC(a_{00})+(b_2\times i+j)\times L \tag{5-2}$$

其中：$i\in[0,\ b_1-1]$，$j\in[0,\ b_2-1]$；b_1 是数组 A 第一维的长度，b_2 是数组 A 第二维的长

度；LOC(a_{ij})是 a_{ij} 的存储地址；LOC(a_{00})是 a_{00} 的存储地址，即二维数组 A 的基地址。

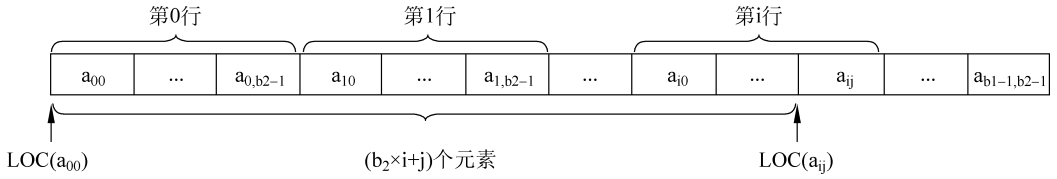


图 5-3　二维数组按行优先存储分配

(2) 按列优先存储方式

按列优先存储方式是指先列后行，先存储列号较小的元素，列号相同者先存储行号较小的元素。如果每个元素占 L 个存储单元且从地址 A 开始依次分配数组各元素，则数组 A 中任一元素 a_{ij} 的存储地址可以由式(5-3)确定，其分配情况如图 5-4 所示。

$$LOC(a_{ij}) = LOC(a_{00}) + (b_1 \times j + i) \times L \tag{5-3}$$

其中：$i \in [0, b_1-1]$，$j \in [0, b_2-1]$；b_1 是数组 A 第一维的长度，b_2 是数组 A 第二维的长度；LOC(a_{ij})是 a_{ij} 的存储地址；LOC(a_{00})是 a_{00} 的存储地址，即二维数组 A 的基地址。

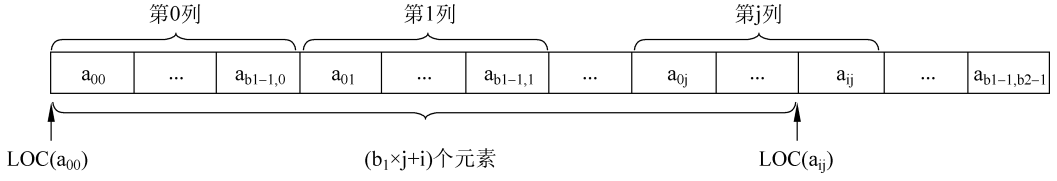


图 5-4　二维数组按列优先存储分配

3) 多维数组的顺序存储

对于 n(n>2)维数组，一般也采用按行优先和按列优先两种存储方法。按行优先存储的基本思想是：最右边的下标先变化，即最右下标从小到大，循环一遍后，右边第二个下标再变，……，最后是最左下标。按列优先存储的基本思想恰好相反：最左边的下标先变化，即最左下标从小到大，循环一遍后，左边第二个下标再变，……，最后是最右下标。

设 n 维数组 A 第 k 维($1 \leqslant k \leqslant n$)的下标范围是$[0, b_k-1]$，如果每个元素占 L 个存储单元且从地址 A 开始依次分配数组各元素，则数组 A 中下标为 j_1、j_2、…、j_n 的元素的按行优先存储地址可以由式(5-4)确定。

$$\begin{aligned} LOC(a_{j_1 j_2 \cdots j_n}) &= LOC(a_{00\cdots0}) + (j_1 \times b_2 \times \cdots \times b_n + j_2 \times b_3 \times \cdots \times b_n + \cdots + j_{n-1} \times b_n + j_n) \times L \\ &= LOC(a_{00\cdots0}) + \left(\sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} b_k + j_n \right) L \end{aligned} \tag{5-4}$$

可以将式(5-4)缩写成式(5-5)的形式：

$$LOC(a_{j_1 j_2 \cdots j_n}) = LOC(a_{00\cdots0}) + \sum_{i=1}^{n} c_i j_i \tag{5-5}$$

其中：$c_n = L$，$c_{i-1} = b_i \times c_i$，$1 < i \leqslant n$。

式(5-5)称为 n 维数组的映像函数。容易看出，一旦确定了数组各维的长度，c_i 就是常数，因此，数组元素的存储地址是其下标的线性函数。数组中的任一元素可以在相同的时间内存取，即数组顺序表是一个随机存取结构。

2. 数组顺序表的类型定义

```
// 数组顺序表的存储表示
#include<stdarg.h>                          // 标准头文件
                                            // 提供宏 va_start、va_arg 和 va_end
                                            // 用于存取变长参数表
#define MAX_ARRAY_DIM  8                    // 假设数组维数的最大值为 8
typedef struct  {
    ElemType    *base;                      // 数组元素基址
    int         dim;                        // 数组维数
    int         *bounds;                    // 数组维界基址
    int         *constants;                 // 数组映像函数常量基址
} Array;

// 二维数组顺序表的存储表示
typedef struct  {
    ElemType   *base;                       // 存储数组元素的基地址
    int         b1, b2;                     // 二维数组第一维和第二维的维界
} Array;
```

为了方便,下面的操作实现将基于二维数组顺序表的类型来讨论。

3. 数组顺序表的操作实现

1）初始化操作

算法 5-1

```
void InitArray(Array &A,int m,int n) {
// 构造一个维界分别是 m 和 n 的空二维数组 A
    A.base=new ElemType[n*m];
    if(!A.base)  Error(" Overflow!");
    A.b1=m;
    A.b2=n;
} // InitArray
```

2）销毁操作

算法 5-2

```
void DestroyArray(Array &A)  {
// 释放数组顺序表 A 所占用的存储空间
    delete []A.base;
    A.b1=0;
    A.b2=0;
} // DestroyArray
```

3）取值操作

算法 5-3

```
void ValueArray(Array A,ElemType &e,int i,int j) {
// 若下标 i 和 j 合理,则用 e 返回其按行优先存储数组 A 的元素值; 否则给出相应信息并退出运行
// 下标 i 和 j 的合理值为 i∈[0,b1-1]和 j∈[0,b2-1]
    if((0<=i)&&(i<A.b1)&&(0<=j)&&(j<A.b2)) {
        off=A.b2*i+j;
```

```
        e = A.base[off];
    }
    else  Error("Suffix Error!");
} // ValueArray
```

4）赋值操作

算法 5-4

```
void AssignArray(Array &A,ElemType e,int i,int j) {
// 若下标 i 和 j 合理,则将 e 赋给其按行优先存储的数组 A; 否则给出相应信息并退出运行
// 下标 i 和 j 的合理值为 i∈[0,b1 - 1]和 j∈[0,b2 - 1]
    if((0 <= i)&&(i < A.b1)&&(0 <= j)&&(j < A.b2)) {
        off = A.b2 * i + j;
        A.base[off] = e;
    }
    else  Error("Suffix Error!");
} // AssignArray
```

例 5-1　二维数组 A 的每个元素是由 6 个字符组成的串，行下标的范围是[0，8]，列下标的范围是[0，9]，试问：

（1）存放二维数组 A 至少需要多少个字节？

（2）如果 A 按行优先方式存储，则元素 a_{85} 的起始地址与当 A 按列优先方式存储时的哪一个元素的起始地址一致。

答：

（1）因为二维数组 A 为 9 行 10 列，共有 90 个元素，所以存放 A 至少需要 90×6＝540 个存储单元。

（2）因为二维数组 A 的元素 a_{85} 按行优先存储的起始地址为 $LOC(a_{85})=LOC(a_{00})+(8\times10+5)\times c=LOC(a_{00})+85\times c$，二维数组 A 的元素 a_{ij} 按列优先存储的起始地址为 $LOC(a_{ij})=LOC(a_{00})+(9\times j+i)\times c$；解此方程，得到 i＝4，j＝9；所以当 A 按行优先方式存储时元素 a_{85} 的起始地址与当 A 按列优先方式存储时元素 a_{49} 的起始地址一致。

5.2　矩阵的压缩存储

矩阵即二维数组，是很多科学与工程计算问题中研究的数学对象。在数据结构中，研究者感兴趣的不是矩阵本身，而是如何存储矩阵中的元素，使矩阵的各种运算能够有效地进行。通常，在使用高级语言编制程序时都是用二维数组来存储矩阵元素。然而，在数值分析中经常会出现有些阶数很高的矩阵，且矩阵中非零元素呈现某种规律分布，或矩阵中有许多值相同的元素或零元素。为了节省存储空间，可以对这类矩阵进行压缩存储，即为多个值相同的非零元素只分配一个存储空间，且对零元素不分配存储空间。

5.2.1　特殊矩阵的压缩存储

非零元素或零元素分布具有一定规律的矩阵称为特殊矩阵（Especial Matrix）。常见特殊矩阵有三种：对称矩阵、三角矩阵和对角矩阵。

1. 对称矩阵

在一个 n 阶矩阵 A 中，如果元素满足 $a_{ij}=a_{ji}(0\leqslant i,j\leqslant n-1)$，则称 A 为 n 阶对称矩阵。图 5-5 给出了一个 5 阶对称矩阵。

因为对称矩阵中的元素关于主对角线是对称的，所以只要存储矩阵中上三角和下三角中每两个对称的元素共享一个存储空间，这样就能节约近一半的存储空间。

$$A=\begin{pmatrix} 3 & 6 & 4 & 7 & 8 \\ 6 & 2 & 8 & 4 & 2 \\ 4 & 8 & 1 & 6 & 9 \\ 7 & 4 & 6 & 0 & 5 \\ 8 & 2 & 9 & 5 & 7 \end{pmatrix}$$

图 5-5　5 阶对称矩阵

1）存储方式

采用按行优先方式存储主对角线(包括对角线)以下的元素，即按照 $a_{00},a_{10},a_{11},\cdots,a_{n-1,0},a_{n-1,1},\cdots,a_{n-1,n-1}$ 的次序依次存放在一维数组 SA[0..n(n+1)/2−1] 中。其中：$SA[0]=a_{00}$，$SA[1]=a_{10}$，…，$SA[n(n+1)/2-1]=a_{n-1,n-1}$。图 5-6 给出了对称矩阵压缩存储的示意图。

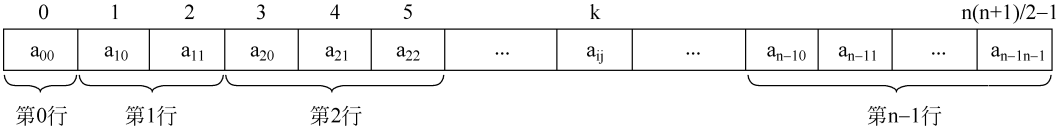


图 5-6　对称矩阵按行优先压缩存储

2）a_{ij} 和 SA[k]之间的对应关系

非零元素 a_{ij} 的前面有 i 行(从第 0 行到第 i−1 行)，一共有 $1+2+\cdots+i=i\times(i+1)/2$ 个元素；在第 i 行上，a_{ij} 之前恰有 j 个元素(即 $a_{i0},a_{i1},\cdots,a_{i,j-1}$)，因此有式(5-6)：

$$SA[i\times(i+1)/2+j]=a_{ij} \tag{5-6}$$

如果 $i\geqslant j$，则 $k=i\times(i+1)/2+j$，且 $0\leqslant k<n(n+1)/2$；如果 $i<j$，则 $k=j\times(j+1)/2+i$，且 $0\leqslant k<n(n+1)/2$。令 $I=\max(i,j)$，$J=\min(i,j)$，k 和 i,j 的对应关系可以统一为式(5-7)：

$$k=I\times(I+1)/2+J,\quad 0\leqslant k<n(n+1)/2 \tag{5-7}$$

3）对称矩阵的地址计算

如果矩阵中的每个元素占 L 个存储单元，那么根据式(5-7)，对称矩阵的下标变换公式由式(5-8)表示：

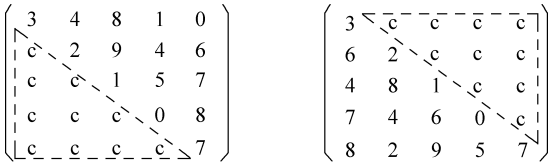
$$\begin{aligned} LOC(a_{ij}) &= LOC(SA[k])=LOC(SA[0])+k\times L \\ &= LOC(SA[0])+[I\times(I+1)/2+J]\times L \end{aligned} \tag{5-8}$$

例如，在图 5-5 所示的 5 阶对称矩阵中，a_{21} 和 a_{12} 均存储在数组 SA[4]中，这是因为 $k=I\times(I+1)/2+J=2\times(2+1)/2+1=4$。

2. 三角矩阵

按主对角线划分，三角矩阵分为上三角矩阵和下三角矩阵两种。下三角(不包括对角线)中的元素均为常数 c 或零的 n 阶矩阵称为上三角矩阵。上三角(不包括对角线)中的元素均为常数 c 或零的 n 阶矩阵称为下三角矩阵。图 5-7(a)给出了一个 5 阶上三角矩阵，图 5-7(b)给出了一个 5 阶下三角矩阵。

三角矩阵按主对角线进行划分，其压缩存储与对称矩阵类似，不同之处仅在于除了存储主对角线一边三角中的元素以外，还要存储对角线另一边三角的常数 c。这样原来需要 n×n 个存储单元，现在只需要 n×(n+1)/2+1 个存储单元，节约了近一半的存储单元。



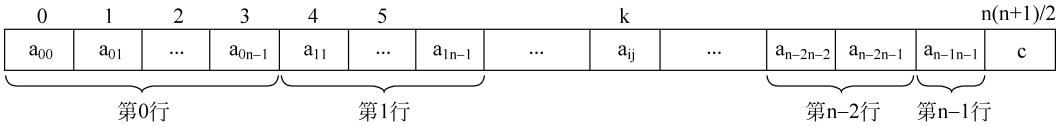
(a) 上三角矩阵　　(b) 下三角矩阵

图 5-7　5 阶三角矩阵

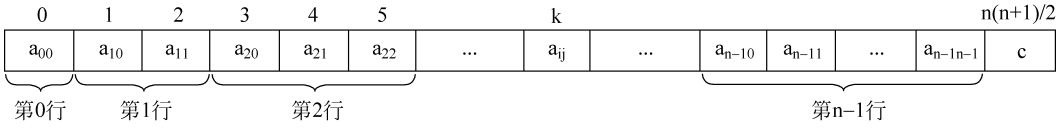
1) 存储方式

采用按行优先方式存储上三角矩阵中的元素，即按照 a_{00}，a_{01}，…，$a_{0,n-1}$，a_{11}，…，$a_{1,n-1}$，…，$a_{n-2,n-2}$，$a_{n-2,n-1}$，$a_{n-1,n-1}$，c 的次序依次存放在一维数组 SA[0..n(n+1)/2]中。其中：SA[0]=a_{00}，SA[1]=a_{01}，…，SA[n(n+1)/2−1]=$a_{n-1,n-1}$，SA[n(n+1)/2]=c。图 5-8(a)给出了上三角矩阵压缩存储的示意图。

采用按行优先方式存储下三角矩阵中的元素，即按照 a_{00}，a_{10}，a_{11}，…，$a_{n-1,0}$，$a_{n-1,1}$，…，$a_{n-1,n-1}$，c 的次序依次存放在一维数组 SA[0..n(n+1)/2]中。其中：SA[0]=a_{00}，SA[1]=a_{10}，…，SA[n(n+1)/2−1]=$a_{n-1,n-1}$，SA[n(n+1)/2]=c。图 5-8(b)给出了下三角矩阵压缩存储的示意图。



(a) 上三角矩阵按行优先压缩存储



(b) 下三角矩阵按行优先压缩存储

图 5-8　三角矩阵按行优先压缩存储

2) a_{ij} 和 SA[k]之间的对应关系

在上三角矩阵中，非零元素 a_{ij} 的前面有 i 行(从第 0 行到第 i−1 行)，一共有(n−0)+(n−1)+(n−2)+…+(n−i+1)=i×(2n−i+1)/2 个元素；在第 i 行上，a_{ij} 之前恰有 j−i 个元素(即 a_{ii}，$a_{i,i+1}$，…，$a_{i,j-1}$)，因此有式(5-9)：

$$SA[i\times(2n-i+1)/2+j-i]=a_{ij} \tag{5-9}$$

得到的上三角矩阵中 k 和 i、j 的对应关系由式(5-10)表示：

$$k=\begin{cases} i\times(2n-i+1)/2+j-i & (i\leqslant j)\\ n(n+1)/2 & (i>j)\end{cases} \tag{5-10}$$

在下三角矩阵中，非零元素 a_{ij} 前面有 i 行(从第 0 行到第 i−1 行)，一共有 1+2+…+i=i×(i+1)/2 个元素；在第 i 行上，a_{ij} 之前恰有 j 个元素(即 a_{i0}，a_{i1}，…，$a_{i,j-1}$)，因此有式(5-11)：

$$SA[i\times(i+1)/2+j]=a_{ij} \tag{5-11}$$

得到的下三角矩阵中 k 和 i、j 的对应关系由式(5-12)表示：

$$k=\begin{cases} i\times(i+1)/2+j & (\text{当 } i\geqslant j \text{ 时}) \\ n(n+1)/2 & (\text{当 } i<j \text{ 时}) \end{cases} \tag{5-12}$$

3) 三角矩阵的地址计算

如果矩阵中的每个元素占 L 个存储单元，那么根据式(5-10)，上三角矩阵的下标变换公式由式(5-13)表示：

$$\begin{aligned} LOC(a_{ij}) &= LOC(SA[k]) = LOC(SA[0]) + k\times L \\ &= LOC(SA[0]) + [i\times(2n-i+1)/2+j-i]\times L \end{aligned} \tag{5-13}$$

如果矩阵中的每个元素占 L 个存储单元，那么根据式(5-11)，下三角矩阵的下标变换公式由式(5-14)表示：

$$\begin{aligned} LOC(a_{ij}) &= LOC(SA[k]) = LOC(SA[0]) + k\times L \\ &= LOC(SA[0]) + [i\times(i+1)/2+j]\times L \end{aligned} \tag{5-14}$$

上(下)三角矩阵重复元素的下标变换公式由式(5-15)表示：

$$\begin{aligned} LOC(a_{ij}) &= LOC(SA[k]) = LOC(SA[0]) + k\times c \\ &= LOC(SA[0]) + [n\times(n+1)/2]\times L \end{aligned} \tag{5-15}$$

例如，在图 5-7(a)所示的上三角矩阵中，a_{12} 存储在数组 SA[6]中，这是因为 $k=i\times(2n-i+1)/2+j-i=1\times(2\times5-1+1)/2+1=6$。在图 5-7(b)所示的下三角矩阵中，$a_{21}$ 存储在数组 SA[4]中，这是因为 $k=i\times(i+1)/2+j=2\times(2+1)/2+1=4$。

3. 对角矩阵

在一个 n 阶矩阵中，所有的非零元素集中在以主对角线为中心的带状区域中，除了主对角线(a_{ii}，$0\leqslant i\leqslant n-1$)和主对角线相邻两侧的若干条对角线($a_{i,i+1}$，$0\leqslant i\leqslant n-2$；$a_{i+1,i}$，$0\leqslant i\leqslant n-2$)上的元素之外，其余元素皆为零的矩阵称为 n 阶对角矩阵。由此可知，一个 w 对角线矩阵(w 为奇数)A 满足：如果 $|i-j|>(w-1)/2$，则元素 $a_{ij}=0$。图 5-9 给出了一个 3 对角 5 阶矩阵。

$$A=\begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

图 5-9　一个 3 对角 5 阶矩阵

1) 存储方式

首先将一个 n 阶的 w 对角矩阵转换为一个 n 行 w 列的 n×w 矩阵(如图 5-10(a))，在该矩阵中共有 $(w\times n-(w^2-1)/4)$ 个非零元素及 $(w^2-1)/4$ 个零元素；然后将这些元素按行优先依次存储到一维数组 SA[n×w]中(如图 5-10(b))。图 5-10 给出了一个 5 阶 3 对角矩阵压缩存储的示意图。

$$\begin{pmatrix} 0 & a_{00} & a_{01} \\ a_{10} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} \\ a_{32} & a_{33} & a_{34} \\ a_{43} & a_{44} & 0 \end{pmatrix}$$

(a) 5 阶 3 对角矩阵转换为一个 5×3 矩阵

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	a_{43}	a_{44}	0

(b) 5×3 矩阵按行优先压缩存储

图 5-10　对角矩阵按行优先压缩存储

2) a_{ij} 和 SA[k]之间的对应关系

非零元素 a_{ij}($|i-j|\leqslant(w-1)/2$)转换为一个 n 行 w 列 n×w 矩阵中的元素 a_{ts}($t\in[0, n-1]$,$s\in[0, w-1]$)的映射关系如式(5-16)所示：

$$\begin{cases} t=i \\ s=j-i+(w-1)/2 \end{cases} \tag{5-16}$$

该 n×w 矩阵中的元素 a_{ts} 在一维数组 SA 中的下标 k 与 t、s 的关系如式(5-17)所示：

$$k=w\times t+s \tag{5-17}$$

3) 对角矩阵的地址计算

如果矩阵中的每个元素占 L 个存储单元，那么根据式(5-16)和式(5-17)，w 对角矩阵的下标变换公式由式(5-18)表示：

$$\begin{aligned} LOC(a_{ij}) &= LOC(SA[k]) = LOC(SA[0]) + k\times L \\ &= LOC(SA[0]) + [w\times t+s]\times L \\ &= LOC(SA[0]) + [w\times i+(j-i+(w-1)/2)]\times L \end{aligned} \tag{5-18}$$

例如，在图 5-9 所示的 5 阶 3 对角矩阵中，a_{12} 存储在数组 SA[5]中，这是因为 k=w×i+(j−i+(w−1)/2)=3×1+(2−1+1)=5；a_{43} 存储在数组 SA[12]中，这是因为 k=w×i+(j−i+(w−1)/2)=3×4 +(3−4+1)=12。

此外，n 阶 w 对角矩阵也可以采用按行优先方式将非零元素直接存储到一个一维数组 SA[w×n−(w^2−1)/4]中。

例如，对于 5 阶 3 对角矩阵中的非零元素 a_{ij}($|i-j|>(w-1)/2$)，前面所有行的非零元素个数为(3×i−1)，第 i 行所在列前面的非零元素个数为(j−i+1)，二者相加得到总的非零元素个数为(2×i+j)；存储到 SA[k]中如图 5-11 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12
a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	a_{43}	a_{44}

图 5-11　5 阶 3 对角矩阵按行优先压缩存储

在所有这些统称为特殊矩阵的矩阵中，非零元的分布都有一个明显的规律，从而都可以将其压缩存储到一维数组中，并找到每个非零元素在一维数组中的对应关系。

5.2.2　稀疏矩阵的压缩存储

在实际应用中，经常会遇到另一类型的矩阵，其阶数高、非零元素较零元素少，且分布没有一定规律。假设在 m×n 矩阵中，如果有 t 个非零元素，令 δ=t/(m×n)，则称 δ 为矩阵的稀疏因子。通常认为 δ≤0.05 的矩阵为稀疏矩阵(Sparse Matrix)。这类矩阵的压缩存储要比特殊矩阵复杂。

1. 稀疏矩阵的抽象数据类型

```
ADT SparseMatrix {
    Data:
        D = {aij | aij ∈ ElemSet, i = 0,1, …,m - 1, m≥0; j = 0,1, …,n - 1, n≥0
                                                m 和 n 分别称为矩阵的行数和列数 }
    Relation:
        R = {Row,Col}
```

```
        Row = {< a_ij, a_i,j+1 >| 0≤i≤m-1, 0≤j≤n-2}
        Col = {< a_ij, a_i+1,j >| 0≤i≤m-2, 0≤j≤n-1}
    Operation:
        InitSMatrix(&M);
            初始条件: 无。
            操作结果: 构造一个空稀疏矩阵 M。
        DestroySMatrix(&M);
            初始条件: 稀疏矩阵 M 已经存在。
            操作结果: 销毁 M。
        CopySMatrix(M,&T);
            初始条件: 稀疏矩阵 M 已经存在。
            操作结果: 将 M 复制给 T。
        AddSMatrix(M,N,&Q);
            初始条件: 稀疏矩阵 M 与 N 已经存在,且行数与列数对应相等。
            操作结果: 用 Q 返回 M 和 N 的和。
        MulSMatrix(M,N,&Q);
            初始条件: 稀疏矩阵 M 与 N 已经存在,且 M 的列数等于 N 的行数。
            操作结果: 用 Q 返回 M 和 N 的积。
        TransposeSMatrix(M,&T);
            初始条件: 稀疏矩阵 M 已经存在。
            操作结果: 用 T 返回 M 的转置矩阵。
} ADT SparseMatrix
```

按照压缩存储的概念,只存储稀疏矩阵的非零元素。因此,除了存储非零元素值 a_{ij} 之外还必须同时记下其所在行和列的位置(i, j),这样组成了一个三元组(i, j, a_{ij})。反之,一个三元组(i, j, a_{ij})唯一确定了矩阵的一个非零元素。将稀疏矩阵非零元素对应的三元组所构成的集合,按行优先顺序排列成一个线性表,称为三元组表(List of 3-Tuples)。由此,稀疏矩阵可以由表示非零元素的三元组及其行列数唯一确定。

例如,图 5-12 给出了一个稀疏矩阵。其三元组表为((0, 0, 15), (0, 3, 22), (0, 5, −15), (1, 1, 11), (1, 2, 3), (2, 3, 6), (4, 0, 91)),加上(5, 6)这一对行、列值便可以作为该稀疏矩阵的另一种描述。而由上述三元组表的不同表示方法可以引出稀疏矩阵不同的压缩存储方法。

$$M=\begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

图 5-12 稀疏矩阵 M

2. 三元组顺序表及操作实现

1) 三元组顺序表的定义

将表示稀疏矩阵非零元素的三元组按行优先顺序排列,并依次存放在一组地址连续的存储单元里的方式称为稀疏矩阵的顺序存储结构,采用这种存储结构的稀疏矩阵称为三元组顺序表(Triple Sequential List)。

2) 三元组顺序表的类型定义

为了运算方便,将稀疏矩阵行列数及非零元素总数均作为三元组顺序表的属性进行描述。其类型描述为:

```
#define TriList_Size 1000                    // 三元组表大小,可根据实际需要而定
typedef struct  {
    int         row,col;                     // 非零元的行下标和列下标
    ElemType    e;                           // 非零元的元素值
```

```
} Triple;
typedef struct  {
    Triple      data[TriList_Size + 1];          // 非零元三元组表,data[0]未用
    int         mu, nu, tu;                      // 矩阵的总行数、总列数和总非零个数
} TSMatrix;
```

图 5-13 给出了图 5-12 表示的稀疏矩阵 M 对应的三元组顺序表。

3) 转置操作

转置运算是一种最简单的矩阵运算，对于一个 m×n 的矩阵 M，其转置矩阵 T 是一个 n×m 的矩阵，且 T(j,i)＝M(i,j)，(i＝0,2,…,n－1,j＝0,2,…,m－1)。如图 5-14 所示的矩阵 T 即为图 5-12 给出的稀疏矩阵 M 的转置矩阵。显然，一个稀疏矩阵的转置矩阵仍然是稀疏矩阵。

	row	col	e
1	0	0	15
2	0	3	22
3	0	5	−15
4	1	1	11
5	1	2	3
6	2	3	6
7	4	0	91
	空闲	空闲	空闲
TriList_Size	5		矩阵的行数
	6		矩阵的列数
	7		非零元个数

图 5-13　图 5-12 表示的稀疏矩阵 M 对应的三元组顺序表

$$T=\begin{pmatrix} 15 & 0 & 0 & 0 & 91 \\ 0 & 11 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 22 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -15 & 0 & 0 & 0 & 0 \end{pmatrix}$$

图 5-14　稀疏矩阵 M 的转置矩阵 T

当用三元组顺序表表示稀疏矩阵时，转置运算就演变为“由 M 的三元组表求得 T 的三元组表”的操作。图 5-15(a)和(b)分别列出了 M 和 T 的三元组顺序表。

	row	col	e
1	0	0	15
2	0	3	22
3	0	5	−15
4	1	1	11
5	1	2	3
6	2	3	6
7	4	0	91

M. data

(a) 矩阵 M 的三元组顺序表

	row	col	e
1	0	0	15
2	0	4	91
3	1	1	11
4	2	1	3
5	3	0	22
6	3	2	6
7	5	0	−15

T. data

(b) 矩阵 T 的三元组顺序表

图 5-15　稀疏矩阵 M 和 T 的三元组顺序表

分析图 5-15(a)和(b)的差异发现只要做到下面两点，就可以由 M. data[]得到 T. data[]，实现矩阵转置：

(1) 将稀疏矩阵 M 中非零元素的行、列值相互调换，即将 M. data[]中每个元素的 row 和 col 相互调换；

(2) 因为三元组顺序表中元素是以按行优先顺序排列的，T.data[]中元素的顺序和M.data[]中元素的顺序不同，所以需要重排三元组顺序表中元素的次序。

在这两点中，最关键的是第二点，即如何实现 T.data[]中所要求的按行优先顺序。常用的处理方法有两种：一种是直接取—顺序存；另一种是顺序取—直接存。

方法一：直接取-顺序存。

(1) 算法设计

基于 T.data[]按行优先的顺序，在 M 中依次扫描第 0 列、第 1 列、…、第 M.nm－1 列的三元组，从中“找出”元素进行“行列互换”后顺序插入到 T.data[]中。通常，也称该方法为“按需点菜”法。

(2) 算法描述

算法 5-5

```
void TransSMatrix_TSM(TSMatrix M,TSMatrix &T) {
// 用 T 返回三元组顺序表 M 的转置矩阵
    InitSMatrix_TSM(T);                          // 初始化三元组顺序表 T
    T.mu = M.nu;  T.nu = M.mu;  T.tu = M.tu;     // 设置 T 的行数、列数和非零元素个数
    if(T.tu) {
        pt = 1;                                  // pt 为 T.data 的下标，初始为 1
        for(i = 0;i < M.nu;++i)
            for(pm = 1;pm <= M.tu;++pm)          // pm 为 M.data 的下标，初始为 1
                if(M.data[pm].col == i) {        // 进行转置
                    T.data[pt].row = M.data[pm].col;
                    T.data[pt].col = M.data[pm].row;
                    T.data[pt].e = M.data[pm].e;
                    ++pt;
                }
    }
}                                                // TransSMatrix_TSM
```

(3) 算法分析

① 问题规模：矩阵 M 的列数 M.nu(简称 nu)和非零元素个数 M.tu(简称 tu)；

② 基本操作：行列互换；

③ 时间分析：算法中基本操作的执行次数主要依赖于嵌套的两个 for 循环，因此算法 5-5 的时间复杂度为 O(nu×tu)。

一般矩阵的转置算法为：

```
void Transpose(ElemType M[][],ElemType T[][],int mu,int nu) {
    for(i = 0;i < nu;++i)
        for(j = 0;j < mu;++j)  T[i][j] = M[j][i];
} // Transpose
```

其时间复杂度为 O(nu×mu)。当 M 中的非零元素个数 tu 和其元素总数 mu×nu 同数量级时，算法 5-5 的时间复杂度为 $O(mu\times nu^2)$，其时间性能劣于一般矩阵转置算法，因此算法 5-5 仅适于 tu≪mu×nu 的情况。

方法二：顺序取-直接存。

(1) 算法设计

如果能预先确定矩阵 M 中每一列(即转置矩阵 T 中每一行)的第一个非零元素在

T.data[]中的位置，则在对 M.data[]中的三元组依次作转置时，就能直接将其放到 T.data[]中恰当的位置。这样，对 M.data[]进行一次扫描就可以使所有非零元素的三元组在 T.data[]中“一次到位”。通常，也称该方法为“按位就座”法。

提出问题：如何确定当前从 M.data 中取出的三元组在 T.data 中应有的位置？

注意到 M.data 中第 0 列的第一个非零元素一定存储在 T.data 中下标为 1 的位置上，该列中其他非零元素则应该存放在 T.data 中后面连续的位置上；M.data 中第 1 列的第一个非零元素在 T.data 中的位置等于其第 0 列的第一个非零元素在 T.data 中的位置加上其第 0 列的非零元素个数；……；以此类推。

解决方法：引入两个数组作为辅助数据结构解决上述问题。

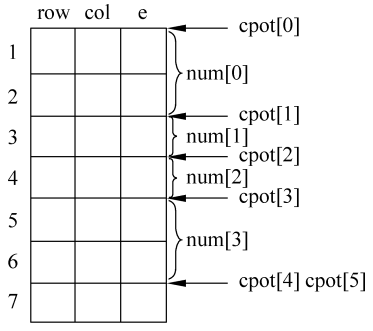
数组 num[col]：存储 M 中第 col 列的非零元素个数。数组 cpot[col]：指示 M 中第 col 列第一个非零元素在 T.data 中的恰当位置。由此可以得到如式(5-19)所示的递推关系：

$$\begin{cases} \text{cpot}[0]=1; \\ \text{cpot}[\text{col}]=\text{cpot}[\text{col}-1]+\text{num}[\text{col}-1]; \quad 1\leqslant \text{col}\leqslant \text{M.nu}-1 \end{cases} \tag{5-19}$$

图 5-12 所示的稀疏矩阵 M 的 num 和 cpot 的数组值如图 5-16(a)所示，num 和 cpot 之间的关系如图 5-16(b)所示。

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

(a) 稀疏矩阵M的num与cpot值



(b) num与cpot之间关系示意图

图 5-16　辅助数组 num 和 cpol

在求出 cpot[col]后只需要扫描一遍 M.data，当扫描到一个 col 列的元素时，直接将其存放在 T.data 中下标为 cpot[col]的位置上，然后将 cpot[col]加 1，即 cpot[col]中存放的始终是下一个 col 列元素(如果有的话)在 T.data 中的位置。

(2) 算法描述

算法 5-6

```
void FastTransSMatrix_TSM(TSMatrix M,TSMatrix &T) {
// 用 T 返回三元组顺序表 M 的转置矩阵
    InitSMatrix_TSM(T);                          // 初始化三元组顺序表 T
    T.mu = M.nu;   T.nu = M.mu;   T.tu = M.tu;   // 设置 T 的行数、列数和非零元素个数
    if(T.tu) {
        for(i = 0;i < M.nu;++i)                  // num 数组初始化为 0
            num[i] = 0;
        for(i = 1;i <= M.tu;++i)                 // 计算 M 中每一列非零元素的个数
            ++num[M.data[i].col];
```

```
        cpot[0] = 1;                              // 置 M 第 0 列第一个非零元素下标为 1
        for(i = 1;i < M.nu;++i)                   // 计算 M 每一列第一个非零元素的下标
            cpot[i] = cpot[i - 1] + num[i - 1];
        for(i = 1;i <= M.tu;++i)  {               // 扫描 M.data,依次进行转置
            j = M.data[i].col;                    // 当前三元组列号
            k = cpot[j];                          // 当前三元组在 T.data 中的下标
            T.data[k].row = M.data[i].col;
            T.data[k].col = M.data[i].row;
            T.data[k].e = M.data[i].e;
            ++cpot[j];                            // 预置同一列下一个三元组的下标
        }
    }
}                                                 // FastTransSMatrix_TSM
```

(3) 算法分析

① 问题规模：矩阵 M 的列数 M.nu(简称 nu)和非零元素个数 M.tu(简称 tu)；

② 基本操作：行列互换；

③ 时间分析：算法中的执行次数依赖于四个并列的 for 循环，因此算法 5-6 的时间复杂度为 O(nu+tu)；

④ 空间分析：算法 5-6 中所需要的存储空间比算法 5-5 多了两个辅助数组。

当 M 中的非零元素个数 tu 和其元素总数 mu×nu 同数量级时，算法 5-6 的时间复杂度为 O(mu×nu)，与一般矩阵转置算法一致。由此可见，算法 5-6 的平均时间性能优于算法 5-5，又称为快速转置法。

当矩阵的非零元素个数和位置在操作过程中变化较大时，就不宜采用顺序存储结构来表示三元组的线性表。例如，在进行“将矩阵 B 加到矩阵 A 上”的操作时，由于非零元素的插入和删除将会引起 A.data 中元素的大量移动，为此，对于这种类型的矩阵，采用链式存储结构表示三元组的线性表更为合适。

3. 十字链表及操作实现

1) 十字链表的定义

将稀疏矩阵的非零元素用一个包含五个域的结点表示：行号域 row、列号域 col、值域 e、列链接指针域 down 和行链接指针域 right，如图 5-17 所示。

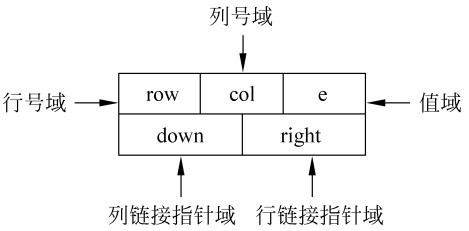


图 5-17 稀疏矩阵十字链表结点结构

稀疏矩阵中同一行的非零元素通过 right 连接成一个行链表，同一列的非零元素通过 down 连接成一个列链表；每个非零元素既是某个行链表中的一个结点，又是某个列链表中的一个结点，整个矩阵构成了一个十字交叉的链表；依靠行列指针连接建立相邻的逻辑关系的方式称为稀疏矩阵的链式存储结构，采用这种存储结构的稀疏矩阵称为十字链表(Cross Linked List)。

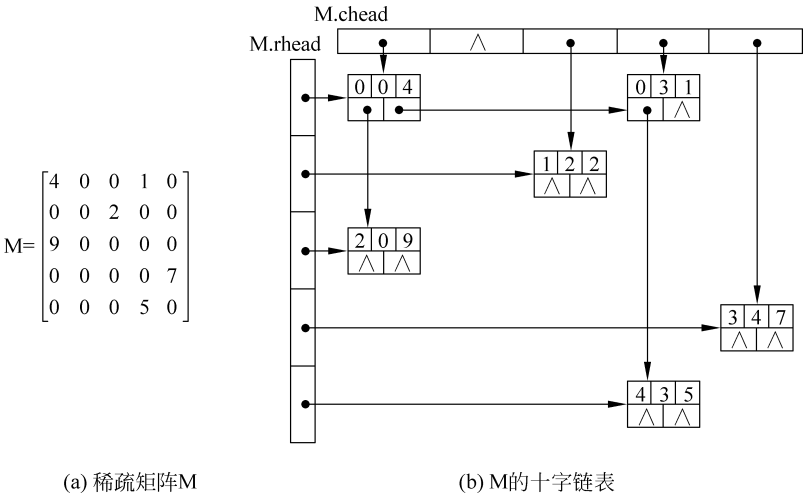
2) 十字链表的类型定义

通常，可以使用两个一维数组分别存储行链表头指针和列链表头指针。其类型描述为：

```
typedef struct OLNode  {
```

```
    int             row,col;                    // 非零元的行下标和列下标
    ElemType        e;                          // 非零元的值
    struct  OLNode  * right, * down;            // 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
    int              mu,  nu,  tu;              // 稀疏矩阵行列数和总非零元个数
} CrossList;
```

图 5-18(b)给出了图 5-18(a)表示的稀疏矩阵 M 对应的十字链表。



(a) 稀疏矩阵M　　(b) M的十字链表

图 5-18　稀疏矩阵 M 及其对应的十字链表

3) 相加操作

(1) 算法设计

对十字链表 A 和十字链表 B 求和,并将其结果存放在 A 中,可以从矩阵第一行开始逐行进行。相加之后的和矩阵中非零元素可能有三种情况。其一,结果为 $a_{ij}+b_{ij}$;其二,结果为 a_{ij}($b_{ij}=0$);其三,结果为 b_{ij}($a_{ij}=0$)。每一行都从行链表头指针出发分别找到 A 和 B 在该行中的第一个非零元素结点后开始比较,按照不同的情况分别处理。

设非空指针 pa 和 pb 分别指向十字链表 A 和 B 中行值相同的两个结点,pa 为空时表明 A 在该行中没有非零元素。为了便于操作,设置一些辅助指针:在 A 的行链表上设 pre 指针,指示 pa 所指结点的前驱;在 A 的列链表上设一个指针 cp[j],其初值和列链表头指针相同。

初始时,令 pa 和 pb 分别指向 A 和 B 第 0 行的第一个非零元素的结点。重复下面操作,依次处理本行结点,直到 B 的本行中没有非零元素结点为止。

- pa==NULL 或(pa->col)>(pb->col)

在 A 中插入一个值为 b_{ij} 的结点;此时需要改变同一行中前一结点 right 域的指针及同一列中前一结点 down 域的指针。

- pa!=NULL 且(pa->col)<(pb->col)

将 pa 指向本行的下一个非零元素结点。

• (pa->col)==(pb->col)且((pa->e)+(pb->e))!=0

将 $a_{ij}+b_{ij}$ 的值赋给 pa 所指结点的 e 域即可，其他所有域的值均不变。

• (pa->col)==(pb->col)且((pa->e)+(pb->e))==0

删除 A 中 pa 所指结点；此时需要改变同一行中前一结点 right 域的指针及同一列中前一结点 down 域的指针。

(2) 算法描述

算法 5-7

```
void AddSMatrix_OL(CrossList &A,CrossList B) {
// 用 A 返回十字链表 A 与 B 之和
    for(j = 0;j < M.nu;j++)  cp[j] = A.chead[j];
    for(i = 0;i < A.mu;i++)  {
        pa = A.rhead[i];
        pb = B.rhead[i];
        pre = NULL;
        while(pb) {
            if((pa == NULL)||(pa->col > pb->col))   {
                p = New(OLNode);
                p->row = pb->row;
                p->col = pb->col;
                p->e = pb->e;
                if(!pre)  A.rhead[i] = p;                    // 插入到 A 的行链表中
                else  pre->right = p;
                p->right = pa;
                pre = p;
                if(!A.chead[p->col])  {                      // 插入到 A 的列链表中
                    A.chead[p->col] = p;
                    p->down = NULL;
                }
                else  {
                    while((cp[p->col]->down!= NULL)
                                &&(cp[p->col]->down->row < p->row))
                        cp[p->col] = cp[p->col]->down;       // 寻找同一列中的前驱
                    p->down = cp[p->col]->down;
                    cp[p->col]->down = p;
                }
                cp[p->col] = p;
            }
            else if(pa->col < pb->col)  {                    // pa 右移一步
                pre = pa;
                pa = pa->right;
            }
            else if(pa->e + pb->e)  {                        // 直接相加
                pa->e += pb->e;
                pre = pa;
                pa = pa->right;
                pb = pb->right;
            }
            else  {                                          // 从行、列链表中删除
```

```
                if(!pre)   A.rhead[i] = pa->right;
                else   pre->right = pa->right;
                p = pa;
                pa = pa->right;
                if(A.chead[p->col] == p) A.chead[p->col] = cp[p->col] = p->down;
                else   cp[p->col]->down = p->down;
                delete p;
            }
        }
    }
} // AddSMatrix_OL
```

(3) 算法分析

① 问题规模：稀疏矩阵 A 和 B 的非零元素的个数 A.tu 和 B.tu；

② 基本操作：逐行扫描，元素相加；

③ 时间分析：从一个结点来看，进行比较、修改指针所需要的时间是一个常数；整个运算过程在于对 A 和 B 的十字链表逐行扫描，其循环次数取决于 A.tu 和 B.tu。因此算法 5-7 的时间复杂度为 O(A.tu+B.tu)。

例 5-2　创建十字链表。

(1) 算法设计

① 初始化行链表的头指针数组和列链表的头指针数组；

② 将非零元素结点插入行链表：如果行链表为空，则直接插入；如果行链表非空，则先寻找插入位置，然后插入；

③ 将非零元素结点插入列链表：如果列链表为空，则直接插入；如果列链表非空，则先寻找插入位置，然后插入。

(2) 算法描述

算法 5-8

```
void CreateSMatrix_OL(CrossList &M)  {
// 按任意次序输入非零元，创建稀疏矩阵的十字链表 M
    cin>>m>>n>>t;                              // 输入稀疏矩阵总行数、列数和非零元个数
    M.mu = m;   M.nu = n;   M.tu = t;
    M.rhead = new OLNode[m+1];
    M.rhead[ ] = NULL;                         // 初始化行头指针向量；各行链表为空链表
    M.chead = new OLNode[n+1];
    M.chead[ ] = NULL;                         // 初始化列头指针向量；各列链表为空链表
    for(cin>>i>>j>>e; e!=0; cin>>i>>j>>e)  {
        p = new OLNode;                        // 生成新结点
        p->row = i;
        p->col = j;
        p->e = e;
        if(M.rhead[i] == NULL)  {              // 完成行插入
            M.rhead[i] = p;
            p->right = NULL;
        }
        else if(M.rhead[i]->col > j)  {        // 寻找在行链表中的插入位置
            p->right = M.rhead[i];
```

```
            M.rhead[i] = p
        }
        else  {
            for(q = M.rhead[i];(q -> right)&&(q -> right -> col < j);q = q -> right);
            p -> right = q -> right;
            q -> right = p;
        }
        if(M.chead[j] == NULL)  {                    // 完成列插入
            M.chead[j] = p;
            p -> down = NULL;
        }
        else if(M.chead[j] -> row > i)  {            // 寻找在列链表中的插入位置
            p -> down = M.chead[j];
            M.chead[j] = p
        }
        else  {
            for(q = M.chead[j];(q -> down)&&(q -> down -> row < i);q = q -> down);
            p -> down = q -> down;
            q -> down = p;
        }
    }
}                                                    // CreateSMatrix_OL
```

(3) 算法分析

① 问题规模：待创建稀疏矩阵的总行数、总列数及非零元个数(分别设置为 m、n 和 t)；

② 基本操作：在十字链表中建立非零元素结点；

③ 时间分析：算法的时间主要花费在建立非零元素结点上。因为每建立一个非零元素结点都需要先寻找它在行列链表中的插入位置，再完成插入；所以对于 m 行 n 列，且有 t 个非零元素的稀疏矩阵来说，算法 5-8 的执行复杂度为 $O(t\times s)$，$s=\mathrm{Max}\{m, n\}$。

算法 5-8 对非零元素输入的先后次序没有要求。反之，如果按照行优先的顺序依次输入三元组，则可以修改算法 5-8，使得其时间复杂度为 O(t)。

5.3 广义表

广义表，又称为列表(Lists，采用复数形式是为了与统称的表 List 的区别)，是线性表的一种推广和扩充，即在广义表中取消了对线性表元素的原子限制，允许它们具有其自身的结构；广义表又区别于数组，即不要求每个元素具有相同类型。

5.3.1 广义表的类型定义

1. 广义表的定义

广义表(Generalized List)是 $n(n\geqslant 0)$个元素的有限序列，一般记作：

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

其中：LS 是广义表的名称；$a_i(1\leqslant i\leqslant n)$是 LS 的成员(也称为直接元素)，它可以是单个元素，也可以是一个广义表，分别称为 LS 的原子和子表。一般用大写字母表示广义表，用小写字母表示原子。

当广义表 LS 非空时，第一个直接元素称为 LS 的表头(Head)；广义表 LS 中除去表头后其余的直接元素组成的广义表称为 LS 的表尾(Tail)。广义表 LS 中的直接元素个数称为 LS 的长度；广义表 LS 中括号的最大嵌套层数称为 LS 的深度。

下面是一些广义表的例子：

(1) A=()：长度为 0，深度为 1，没有元素，是一个空表；

(2) B=(e)：长度为 1，深度为 1，只有一个原子；

(3) C=(a，(b，c，d))：长度为 2，深度为 2，有一个原子和一个子表；

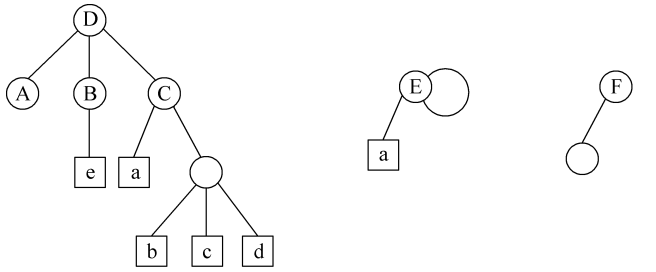
(4) D=(A，B，C)：长度为 3，深度为 3，有三个子表；

(5) E=(a，E)：长度为 2，深度为无穷大，是一个递归表；

(6) F=(())：长度为 1，深度为 2，只有一个空表。

2. 广义表的图形表示

可以采用图形方法表示广义表的逻辑结构：对每个广义表元素 a_i 用一个结点来表示，如果 a_i 为原子，则用矩形结点表示；如果 a_i 为广义表，则用圆形结点表示；结点之间的边表示元素之间的“包含/属于”关系。对于上面列举的 6 个广义表，其图形表示如图 5-19 所示。



(a) 广义表A、B、C、D　　(b) 广义表E　　(c) 广义表F

图 5-19　广义表图形表示的示意图

3. 广义表的主要特性

从上述广义表的定义和例子可以看出，广义表具有以下四个特性：

(1) 广义线性性：对任意广义表来说，如果不考虑其元素的内部结构，则它是一个线性表，它的直接元素之间是线性关系。

(2) 元素复合性：广义表中的元素有原子和子表两种，因此，广义表中元素的类型不统一。一个子表，在某一层上被当作元素，但就它本身的结构而言，也是广义表。在其他数据结构中，并不把子表这样的复合元素看作元素。

(3) 元素递归性：广义表可以是递归的。广义表的定义并没有限制元素的递归，即广义表也可以是其自身的子表，这种递归性使得广义表具有较强的表达能力。

(4) 元素共享性：广义表以及广义表的元素可以为其他广义表所共享。例如，对于上面列举的 6 个广义表，广义表 A、广义表 B、广义表 C 是广义表 D 的共享子表。

广义表的上述四个特性对于它的使用价值和应用效果起到了很大的作用。广义表的结构相当灵活，它可以兼容线性表、数组、树和有向图等各种常用的数据结构。例如，当二维数组的每行(或每列)作为子表处理时，二维数组即为一个广义表；如果限制广义表中元素的共享和递归，广义表和树对应；如果限制广义表的递归并允许元素共享，则广义表和图

对应。

4. 广义表的基本运算

由于广义表是线性表的一种推广和扩充，因此和线性表类似，也可以对广义表进行查找、插入、删除和取表元素值等操作。但广义表在结构上较线性表复杂得多，操作的实现不如线性表简单。在这些操作中，最重要的两个基本运算是取广义表表头 GetHead 和取广义表表尾 GetTail。

任何一个非空广义表表头是表中第一个元素，它可以是原子，也可以是广义表；而其表尾必定是广义表。下面是对给出的广义表取表头和取表尾的例子：

(1) A=()：因为是空表，所以不能进行取表头和取表尾的操作；

(2) B=(e)：GetHead(B)=e，GetTail(B)=()；

(3) C=(a，(b，c，d))：GetHead(C)=a，GetTail(C)=((b，c，d))；

(4) D=(A，B，C)：GetHead(D)=A，GetTail(D)=(B，C)；

(5) E=(a，E)：GetHead(E)=a，GetTail(E)=(E)；

(6) F=(())：GetHead(F)=()，GetTail(F)=()。

值得注意的是：广义表()和广义表(())是不同的。()为空表，其长度 n=0，不能分解成表头和表尾；(())不是空表，其长度 n=1，可以分解得到其表头是空表()、表尾是空表()。

5. 广义表的抽象数据类型

```
ADT GList {
    Data:
        D = {e_i | e_i ∈ AtomSet 或 e_i ∈ GList, i = 1,2,…,n, n≥0, AtomSet 为某个数据对象}
    Relation:
        R = {< e_{i-1}, e_i > | e_{i-1}, e_i ∈ D, i = 2,…,n}
    Operation:
        InitGList(&GL);
            初始条件：无。
            操作结果：构造一个空广义表 GL。
        CreateGList(&GL,S);
            初始条件：已知广义表书写形式串 S。
            操作结果：用 GL 返回一个由 S 创建的广义表。
        DestroyGList(&GL);
            初始条件：广义表 GL 已经存在。
            操作结果：销毁 L。
        CopyGList(&T,GL);
            初始条件：广义表 GL 已经存在。
            操作结果：用 T 返回由 GL 复制得到的广义表。
        GListLength(GL);
            初始条件：广义表 GL 已经存在。
            操作结果：求 L 的长度。
        GListDepth(GL);
            初始条件：广义表 GL 已经存在。
            操作结果：求 GL 的深度。
        GetHead(GL);
            初始条件：广义表 GL 已经存在。
            操作结果：取 GL 的表头。
        GetTail(GL);
```

```
            初始条件：广义表 GL 已经存在。
            操作结果：取 GL 的表尾。
        TraverseGList(GL);
            初始条件：广义表 GL 已经存在。
            操作结果：依次访问 GL 中的每个元素。
} ADT GList
```

5.3.2　广义表的链式表示及操作实现

由于广义表中元素可以具有不同结构(原子或子表)，很难采用顺序存储结构表示，通常采用链式存储结构，每个数据元素可以使用一个结点表示。广义表的链式表示有两种形式：头尾链表和扩展线性链表。

1. 头尾链表

如果广义表非空，则可以分解为表头和表尾；反之，一对确定的表头和表尾可以确定唯一一个广义表。根据这一性质，广义表可以采用头尾链表(Head-Tail Linked)存储结构。由于广义表中的数据元素既可以是广义表也可以是原子，因此在头尾链表中的结点结构也有两种：一种是表结点，用以存储广义表；另一种是原子结点，用于存储原子。为了区分这两类结点，在结点中还要设置一个标志域。

(1) 表结点：由三个域组成，分别为标志域(tag＝1)、指示表头结点的指针域 hp 和指示表尾结点的指针域 tp，如图 5-20(a)所示。

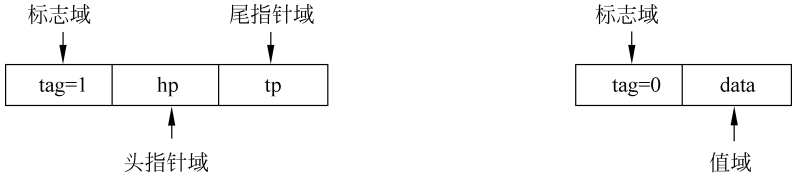


图 5-20　头尾链表中的结点结构

(2) 原子结点：由两个域组成，分别为标志域(tag＝0)和存储原子数据的值域 data，如图 5-20(b)所示。

```
// 广义表的头尾链表存储表示
typedef enum{ATOM, LIST} ElemTag;                  // ATOM == 0: 原子,LIST == 1: 子表
typedef struct GLNode {
    ElemTag        tag;                            // 标志域,用于区分原子结点和表结点
    union {                                        // 原子结点和表结点的共用体
        ElemType   data;                           // 原子结点的数据域
        struct {
            struct GLNode * hp, * tp;              // hp 指向表头,tp 指向表尾
        } ptr;
    };
} GLNode;
typedef GLNode * GList;
```

对于前面列举的 6 个广义表，采用头尾链表存储结构的示意图如图 5-21 所示。可以看出头尾链表具有三个特点，在某种程度上给广义表的操作带来了方便。

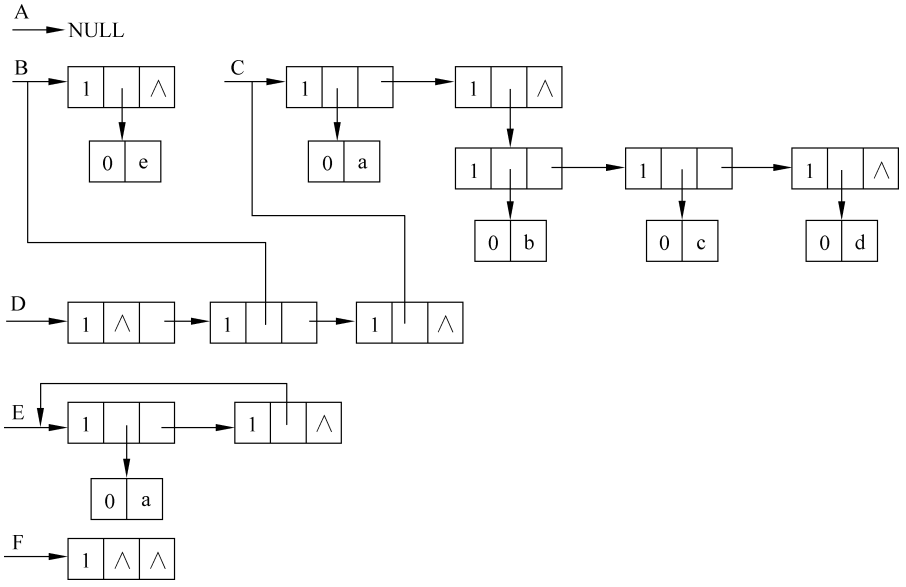


图 5-21　广义表头尾链表存储表示示例

(1) 除空表的表头指针为空外，对任何非空广义表，其表头指针均指向一个表结点，且该结点中的 hp 域指示广义表表头(原子结点或表结点)，tp 域指示广义表表尾(除非表尾为空，则指针为空，否则必为表结点)。

(2) 容易分清广义表中原子或子表所在的层次。例如，在广义表 D 中，原子 a 和 e 在同一层上，而原子 b、c、d 在同一层且比 a 和 e 低一层，子表 B 和 C 在同一层。

(3) 容易求得广义表的长度，即最上层的结点个数为广义表的长度。例如，在广义表 D 的最高层有三个表结点，其广义表的长度为 3。

2. 扩展线性链表

广义表是线性表的一种推广和扩充，因此也可以采用另一种结点结构的链表表示广义表，称为扩展线性链表(Extended Linklist)。在扩展线性尾链表中的结点结构也有两种：一种是表结点，用于存储广义表；另一种是原子结点，用于存储原子。为了区分这两类结点，在结点中也要设置一个标志域。

(1) 表结点：由三个域组成，分别为标志域(tag＝1)、指示表头的指针域 hp 和指示下一个结点的指针域 tp，如图 5-22(a)所示。

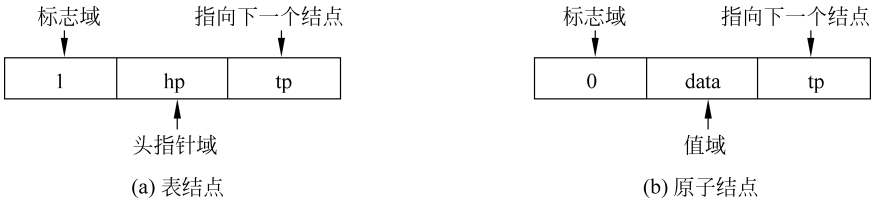


图 5-22　扩展线性链表中的表结点和原子结点结构

(2) 原子结点：由三个域组成，分别为标志域(tag＝0)、存储原子数据的值域 data 和指示下一个结点的指针域 tp，如图 5-22(b)所示。

```
// 广义表的扩展线性链表存储表示
typedef enum{ATOM, LIST}  ElemTag;              // ATOM == 0: 原子,LIST == 1: 子表
typedef struct GLNode  {
    ElemTag             tag;                    // 公共部分,用于区分原子结点和表结点
    union  {                                    // 原子结点和表结点的联合部分
        AtomType        data;                   // 原子结点的值域
        struct GLNode   * hp;                   // 表结点的表头指针
    };
    struct GLNode       * tp;                   // 指向下一个元素结点,相当于链表的 next
} GLNode;
typedef GLNode * GList;
```

对于前面列举的 6 个广义表,采用扩展线性链表存储结构的示意图如图 5-23 所示。

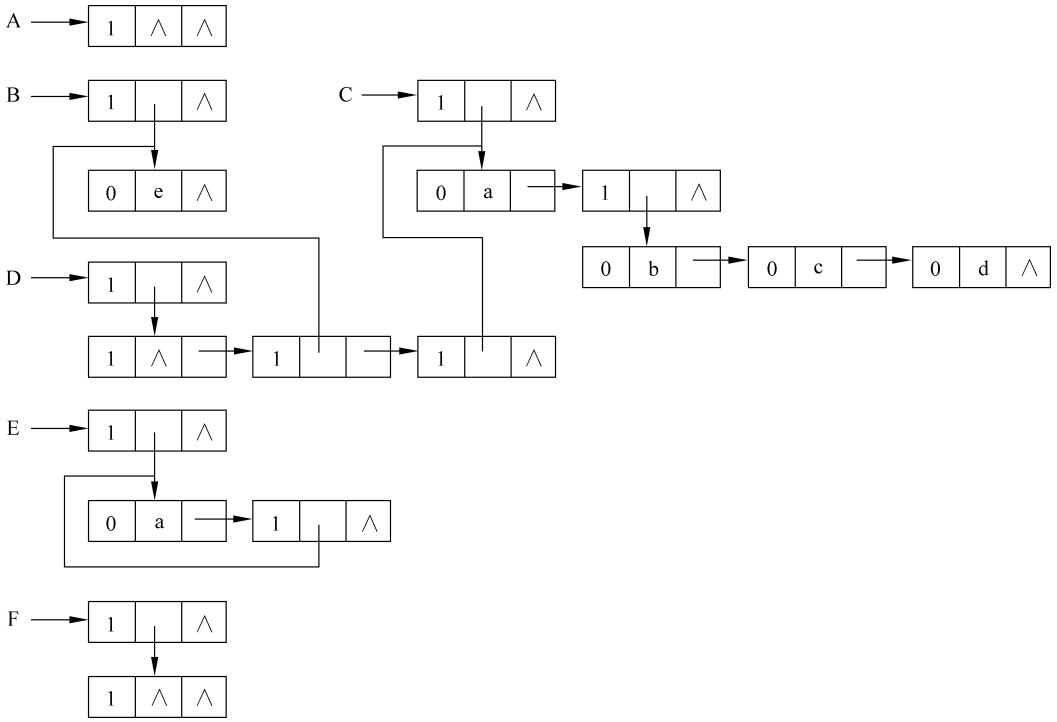


图 5-23　广义表扩展线性链表表示示例

对于广义表的这两种存储结构,读者只需要根据自己的习惯掌握其中一种结构即可。

3. 头尾链表的操作实现

1) 初始化操作

(1) 算法设计

因为空广义表头尾链表的指针为空,所以直接返回空指针即可。

(2) 算法描述

算法 5-9

```
void InitGList(GList &GL)  {
// 构造一个空的广义表头尾链表 GL
    GL = NULL;
```

```
} // InitGList
```

2）复制操作

（1）算法设计

任何一个非空广义表均可以分解成表头和表尾；反之，一对确定的表头和表尾也可唯一地确定一个广义表。采用递归方式先分别复制非空广义表的表头和表尾，再合成即可完成广义表的复制。

假设 GL 是原始的广义表头尾链表，GT 是复制的广义表头尾链表，递归模型如下：

基本项：当 GL 为空表时，置空表 GT；

归纳项：当 GL 为非空表时，分别复制其表头和表尾。

（2）算法描述

算法 5-10

```
void CopyGList(GList &GT,GList GL)  {
// 采用递归方法,实现复制广义表头尾链表 GL 到 GT 的操作
    if(!GL)  GT = NULL;                          // 复制空表
    else  {
        GT = new GLNode;                         // 创建一个表结点
        if(!GT)  Error(" Overflow!");
        GT -> tag = GL -> tag;
        if(GL -> tag == ATOM)                    // 复制原子
            GT -> data = GL -> data;
        else  {                                  // 复制表头和表尾
            CopyGList(GT -> ptr.hp,GL -> ptr.hp);
            CopyGList(GT -> ptr.tp,GL -> ptr.tp);
        }
    }
}                                                // CopyGList
```

3）求广义表长度操作

（1）算法设计

采用递归方式先求解广义表头尾链表 GL 表尾的长度，然后再令其加 1 即为广义表的长度。递归模型如下：

基本项：当 GL 为空表时，表长为 0；

归纳项：当 GL 为非空表时，表长为表尾长度加 1。

（2）算法描述

算法 5-11

```
int GListLength(GList GL) {
// 采用递归方式求解并返回头尾链表 GL 的长度
    if(GL == NULL)  return 0;
    else  {
        taillen = GListsLength(GL -> ptr.tp);    // 递归求解 GL 表尾的长度
        return (taillen + 1);
    }
}                                                // GListLength
```

4）求广义表深度操作

（1）算法设计

采用递归方式从表头开始依次求解广义表 GL 中所有直接元素的深度，然后令其中的最大深度加 1 即为广义表的深度。递归模型如下：

基本项：当 GL 是空表时，深度为 1；

　　当 GL 是原子时，深度为 0；

归纳项：当 GL 是非空表时，深度为其直接元素的最大深度加 1。

（2）算法描述

算法 5-12

```
int GListDepth(LinkLists GL) {
// 采用递归方法求解并返回头尾链表 GL 的深度
    if(!GL)  return 1;
    if(GL->tag == ATOM)  return 0;
    for(max = 0,p = GL;p;p->ptr.tp) {
        depth = GListsDepth(p->ptr.hp);        // 递归求解 GL 表头的深度
        if(depth > max)  max = depth;
    }
    return(max + 1);
}                                              // GlistDepth
```

5）输出广义表中元素操作

（1）算法设计

采用递归方式求解。递归模型如下：

基本项：当 GL 为空表时，输出“()”；

　　当 GL 是原子时，输出原子值；

归纳项：当 GL 为非空表时，依次输出该子表中的元素。

（2）算法描述

算法 5-13

```
void TraverseGList(GList GL) {
// 采用递归方法依次输出头尾链表 GL 中的元素值
    if(!GL)  cout <<"()";
    else  {
        if(GL->tag == ATOM)  cout << GL->data;
        else  {
            cout <<"(";
            p = GL;
            while(p) {
                TraverseGList(p->ptr.hp);      // 递归调用，输出第 i 项元素
                p = p->ptr.tp;
                if(p)  cout <<',';             // 若表尾非空，则输出逗号
            }
            cout <<')';                        // 输出 GL 的右括号
        }
    }
}                                              // TraverseGList
```

6）求广义表的表头操作

（1）算法设计

① 如果广义表 GT 是空表，则不能进行求表头操作，给出相应信息；

② 如果广义表 GT 中第一个直接元素是原子，则直接输出其原子值；

③ 如果广义表 GT 中第一个直接元素是子表，则调用算法 5-13（TraverseGList）依次输出子表中的元素。

（2）算法描述

算法 5-14

```
void GetHead(GList GL)  {
// 求头尾链表 GL 的表头,并输出
    if(GL == NULL)  Error("空表不能取表头!");
    head = GT -> ptr.hp;
    if(head -> tag == ATOM)  cout <<"表头: "<< head -> data << endl;
    else  {
        cout <<"表头: ";
        TraverseGList(head);                    // 参见算法 5-13,输出表头
        cout << endl;
    }
}                                               // GetHead
```

7）求广义表的表尾操作

（1）算法设计

① 如果广义表 GT 是空表，则不能进行求表尾操作，给出相应信息；

② 如果广义表 GT 是非空表，则调用算法 5-13（TraverseGList）依次输出表尾中的元素。

（2）算法描述

算法 5-15

```
void GetTail(GList GL) {
// 求广义表头尾链表 GL 的表尾,并输出
    if(GL == NULL)  Error("空表不能取表尾!");
    tail = p -> ptr.tp;
    cout <<"表尾: ";
    TraverseGList(tail);                        // 参见算法 5-13,输出表尾
    cout << endl;
}                                               // GetTail
```

例 5-3 创建广义表的头尾链表。

（1）算法设计

① 假设把广义表的书写形式看成是一个字符序列 S，那么 S 可能有下面两种情况：

- S＝()（带括号的空串）；
- S＝(a_1，a_2，…，a_n)，其中 a_i(i＝1，2，…，n)是 S 的子串。

对应于第一种情况，S 的广义表为空表；对应于第二种情况，S 的广义表中含有 n 个子表，每个子表的书写形式即为子串 a_i(i＝1，2，…，n)，此时可以类似于求广义表深度的操作，分析由 S 建立的广义表和由 a_i(i＝1，2，…，n)建立的子表之间的关系。

如果创建广义表的头尾链表，则含有 n 个子表的广义表中有 n 个表结点序列。第 i(i=1，2，…，n−1)个表结点中的表尾指针 tp 指向第 i+1 个表结点；第 n 个表结点的表尾指针 tp 为 NULL。并且，如果把原子也看成是子表的话，则第 i 个表结点的表头指针 hp 指向由 a_i(i=1，2，…，n)建立的子表。因此，由 S 建立广义表头尾链表的问题就可以转化为由 a_i(i=1，2，…，n)建立子表的问题。

② a_i 可能有三种情况：

- 带括号的空串；
- 长度=1 的单字符；
- 长度>1 的字符串。

显然，前两种情况为递归的终结状态，子表为空表或只含一个原子结点；后一种情况为递归调用。

③ 在不考虑输入字符串可能出错的前提下，可以得到下面创建广义表头尾链表的递归模型：

基本项：当 S 为空串时，置空广义表；当 S 为单字符时，建立原子结点广义表。

归纳项：假设串 sub 为脱去 S 中最外层括号的子串，记为"s_1，s_2，…，s_n"，其中 s_i(i=1，2，…，n)为非空字符串。对每一个 s_i 建立一个表结点，并令其 hp 域的指针为由 s_i 建立的子表的头指针，除了最后建立的表结点的尾指针为 NULL 外，其余表结点的尾指针均指向在它之后建立的表结点。

④ 设计一个函数 Decompose(Str，Hstr)，其功能是从串 Str 中取出第一个“,”之前的子串赋给 HStr，并使 Str 成为删除子串 HStr 和','之后的剩余串。如果串 Str 中没有字符',',则操作后的 HStr 即为操作前的 Str，而操作后的 Str 为空串 NULL。

(2) 算法描述

算法 5-16

```
void CreateGList(GList &GL,SString S)  {
// 从广义表书写形式串 S 创建广义表头尾链表 GL
    es = "()";
    if(StrCompare_SS(S,es))  GL = NULL;          // 创建空广义表
    else  {
        GL = new GLNode;                         // 建立结点空间
        if(!GT)  Error(" Overflow!");
        if(StrLength_SS(S) == 1)  {              // 创建单原子广义表
            GL -> tag = ATOM;
            GL -> data = S[1];
        }
        else  {                                  // 创建非空且非单原子广义表
            GL -> tag = LIST;
            p = GL;
            SubString_SS(Sub,S,2,StrLength_SS(S) - 2);
                                                 // 脱外层括号
            do  {                                // 重复建立 n 个子表
                Deocompose(Sub,HSub);            // 从 Sub 中分离表头串 HSub
                CreateGList(p -> ptr.hp,HSub);
                q = p;
                if(StrLength_SS(Sub)> 0)  {      // 表尾非空
```

```
                    if(!(p = new GLNode))  Error(" Overflow!");
                    p -> tag = LIST;
                    q -> ptr.tp = p;
                }
            } while(StrLength_SS(Sub)> 0);
            q -> ptr.tp = NULL;
        }
    }
}                                                   // CreateGList

void Decompose(SString &Str,SString &HStr)  {
// 将非空串 str 分割成两部分: HStr 为第一个字符 ','之前的子串,Str 为之后的子串
    n = StrLength_SS(Str);
    i = 1;
    k = 0;                                          // 设 k 为尚未配对的左括号个数
    ch = "";
    for(i = 1,k = 0;(i <= n)&&(ch!= ",")||(k!= 0);++i)  {
                                                    // 搜索最外层的第一个逗号
        SubString_SS(ch,Str,i,1);
        if(ch == "(")  ++k;
        if(ch == ")")  --k;
    }
    if(i <= n) {
        HStr = SubString_SS(Str,1,i - 2);
        Str = SubString_SS(Str,i,n - i + 1);
    }
    else  {
        StrCopy_SS(HStr,Str);
        ClearString_SS(Str);
    }
}                                                   // Decompose
```

在一般情况下使用的广义表大多数既不是递归表,也不为其他表所共享。对广义表可以这样来理解,广义表中的一个数据元素可以是另一个广义表,一个 m 元多项式的表示就是广义表的这种应用的典型实例。在第 2 章中讨论了一元多项式,一个一元多项式可以用一个长度为 m 且数据元素有两个数据项(系数项和指数项)的线性表来表示。下面例 5-4 将讨论如何表示 m 元多项式。

例 5-4 根据式 5-20 给出的三元多项式,设计其广义表的链式存储结构。

$$P(x,y,z)=2x^7y^3z^2+x^5y^4z^2+6x^3y^5z+3xyz+10 \tag{5-20}$$

解答:

(1) 链式存储结构分析

① 可以将 P(x, y, z)以 z 为主变量,合成为式(5-21):

$$P(x,y,z)=A(x,y)z^2+B(x,y)z+10 \tag{5-21}$$

其中:$A(x,y)=2x^7y^3+x^5y^4$,$B(x,y)=6x^3y^5+3xy$。

② 对 A 和 B 分别以 y 为主变量,继续合成为式(5-22):

$$A(x,y)=C(x)y^4+D(x)y^3,\quad B(x,y)=E(x)y^5+F(x)y \tag{5-22}$$

其中:$C(x)=x^5$,$D(x)=2x$,$E(x)=6x^3$,$F(x)=3x$。

③ 将 P、A、B、C、D、E、F 以广义表的形式表示。表示形式为式(5-23)：

主变量(主变量的系数,主变量的幂数)　　(5-23)

则 P、A、B、C、D、E、F 分别表示为：P＝z((A,2),(B,1),(10,0))，A＝y((C,4),(D,3))，B＝y((E,5),(F,1))，C＝x((1,5))，D＝x((2,7))，E＝x((6,3))，F＝x((3,1))。

(2) 链式存储结构设计

① 结点结构设计如图 5-24 所示。

标志域	指数域	指针域 1/系数域	指针域 2

图 5-24　广义表链式存储表示的结点结构

其中：指数域指示该元素结点所表示的项的指数；标志域为 0 时，“指针域 1/系数域”指示该项的系数，为 1 时，“指针域 1/系数域”指向系数子表；指针域 2 指示广义表中该元素结点的下一个元素结点。

② 每一个子表都有一个表头结点。表头结点结构设计如图 5-25 所示。

标志域	主变量域		指针域

图 5-25　广义表链式存储表示的表头结点结构

其中：标志域为 1 时，主变量域指示该层的主变量；指针域指示该子表的第一个结点。

③ 广义表链式存储结构有一个头结点。头结点结构设计如图 5-26 所示。

标志域	变量数域	指针域	∧

图 5-26　广义表链式存储表示的头结点结构

其中：标志域为 1 时，“变量数域”指示该多项式中的变量个数；指针域指示第一个表头结点。

(3) 链式存储结构实现

广义表链式存储结构实现如图 5-27 所示。

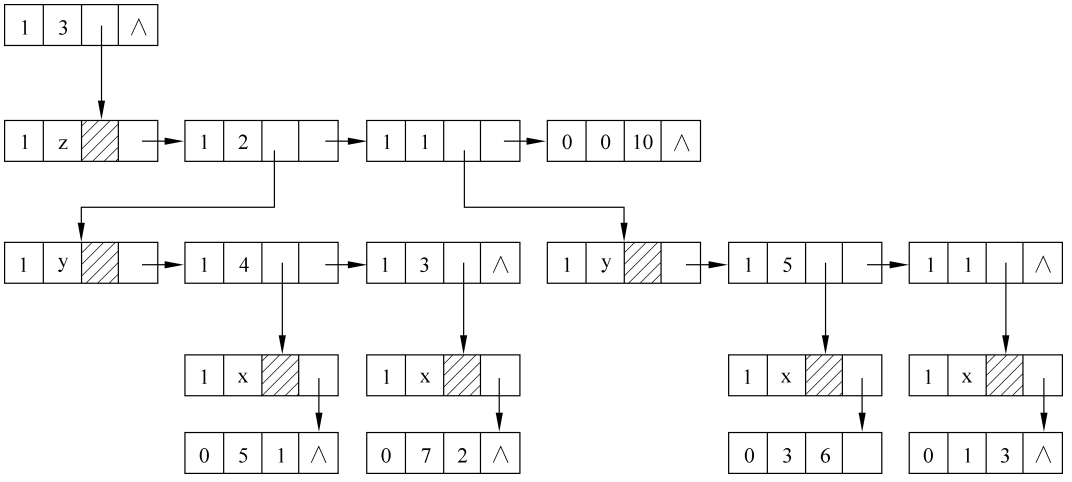


图 5-27　广义表链式存储结构实现

习题

一、填空题

1. 数组通常只有两种基本运算：(　　)和(　　)，这决定了数组通常采用(　　)结构来实现存储。

2. 二维数组 A 中行下标从 10 到 20，列下标从 5 到 10，如果按行优先存储，每个元素占 4 个存储单元，A[10][5]的存储地址是 1000，则元素 A[15][10]的存储地址是(　　)。

3. 设有一个 10 阶的对称矩阵 A，如果采用压缩存储，A[0][0]为第一个元素，其存储地址为 LOC(a_{00})，每个元素的长度是 1 个字节，则元素 A[8][5]的存储地址为(　　)。

4. 广义表 LS=((a),(((b),c)),(d))，其长度是(　　)，深度是(　　)，表头是(　　)，表尾是(　　)。

5. 广义表 LS=(a, (b, c, d), e)，如果采用函数 GetHead 和 GetTail 取出 LS 中的原子 b，则运算是(　　)。

二、选择题

1. 将数组称为随机存取结构是因为(　　)。

(A) 数组元素是随机的　　(B) 对数组任一元素存取时间相等

(C) 随时可以对数组进行访问　　(D) 数组的存储结构是不定的

2. 二维数组 A 的每个元素是 8 个字节组成的双精度实数，行下标的范围是[0,7]，列下标的范围是[0,9]，则存放 A 至少需要(　　)个字节。

(A) 80　　(B) 144　　(C) 504　　(D) 640

3. 对特殊矩阵采用压缩存储的目的主要是为了(　　)。

(A) 表达变得简单　　(B) 对矩阵元素的存取变得简单

(C) 去掉矩阵中的多余元素　　(D) 减少不必要的存储空间

4. 如果广义表 LS 满足 GetHead(LS)=GetTail(LS)，则 LS 为(　　)。

(A) ()　　(B) (())　　(C) ((), ())　　(D) ((), (), ())

5. 下面的说法中，不正确的是(　　)。

(A) 广义表是一种多层次的结构　　(B) 广义表是一种非线性结构

(C) 广义表是一种共享结构　　(D) 广义表是一种递归

三、问答题

1. 特殊矩阵和稀疏矩阵哪一种压缩存储后会失去随机存取的功能？为什么？

2. 数组、广义表与线性表之间有什么样的关系？

3. 设一个二维数组 A[5][6]的每个元素占 4 个字节，已知 LOC(a_{00})=1000，那么 A 共占多少个字节？A 的终端结点 a_{45} 的起始地址是多少？当按行优先存储时，a_{25} 的起始地址是多少？当按列优先存储时，a_{25} 的起始地址是多少？

4. 分别画出下列广义表的图形表示：

(1) A(a, B(b, d), C(e, B(b, d), L(f, g)))　　(2) A(a, B(b, A))

5. 下列广义表运算的结果是什么？

(1) GetHead(GetTail(((a, b), (c, d), (e, f))))

(2) GetTail(GetHead(((a, b), (c, d), (e, f))))

(3) GetHead(GetTail(GetHead(((a, b), (e, f)))))

(4) GetTail(GetHead(GetTail(((a, b), (e, f)))))

(5) GetTail(GetTail(GetHead(((a, b), (e, f)))))

四、算法设计题

1. 如果在矩阵 A 中存在一个元素 $a_{ij}(0\leqslant i\leqslant m-1,0\leqslant j\leqslant n-1)$，该元素是第 i 行元素中最小值且又是第 j 列元素中最大值，则称此元素为该矩阵的一个马鞍点。假设以二维数组常规存储矩阵 A，试设计一个算法：求解该矩阵的所有马鞍点。

2. 已知两个 $n\times n$ 的对称矩阵按压缩存储方法存储在一维数组 A 和 B 中，试设计一个算法：求解对称矩阵的乘积。

3. 试设计一个算法：创建稀疏矩阵的十字链表。

4. 试设计一个算法：创建广义表的头尾链表。

5. 试设计一个算法：求解一个广义表所拥有的原子个数。