

数据结构

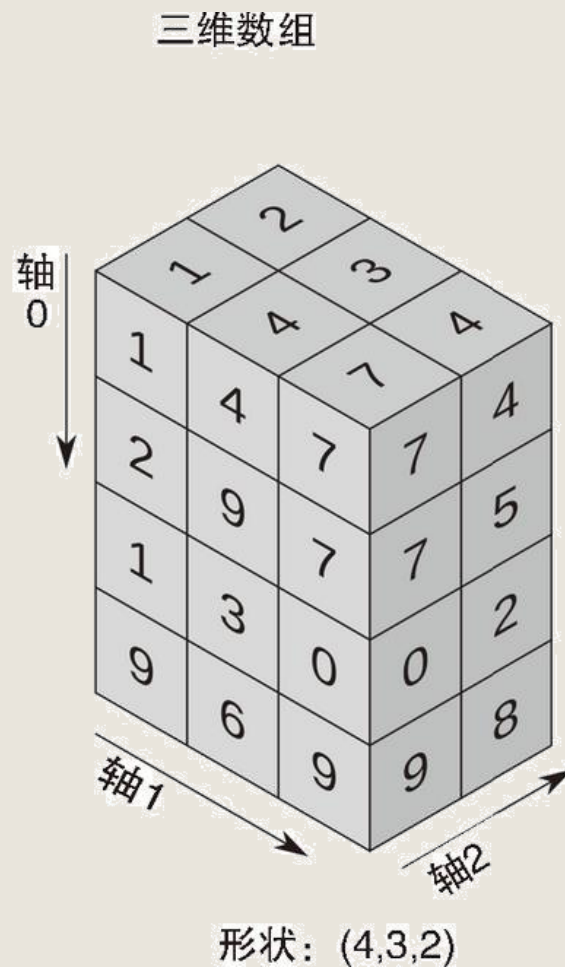
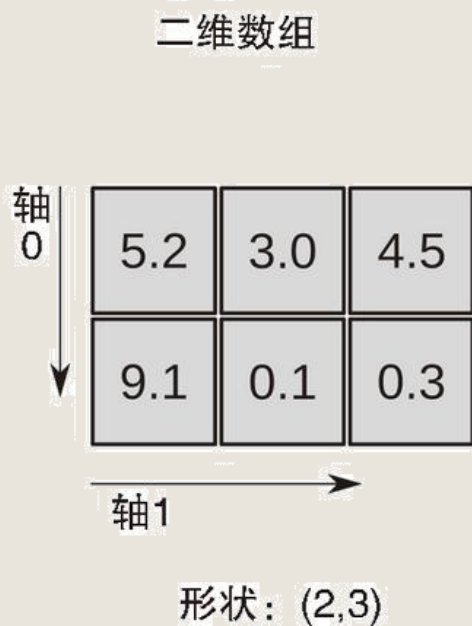
Ch5 数组和广义表

北京邮电大学 计算机学院

第5章 数组和广义表

- 5.1 数组和线性表的关系以及数组的运算
- 5.2 数组的顺序存储结构
- 5.3 矩阵的压缩存储
- 5.4 广义表的定义
- 5.5 广义表的存储结构
- 5.6 广义表的递归算法

5.1 数组和线性表的关系以及数组的运算



5.1 数组和线性表的关系以及数组的运算

任何数组A都可以看作一个线性表

$$A=(a_1, a_2, \dots, a_i, \dots a_n)$$

二维数组 $m \times n$ 时,

a_i 是数组中第 i 列所有元素, 表中每一个元素是一个一维数组;

三维数组时,

表中每一个元素是一个二维数组;

n 维数组时,

表中每一个元素是一个 $(n-1)$ 维数组。

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

数组与线性表之间的关系

线性表的扩展，其数据元素本身也是线性表

数组的特点

- 数组中各元素都具有统一的类型
- 可以认为， d 维数组的非边界元素具有 d 个直接前趋和 d 个直接后继
- 数组维数确定后，数据元素个数和元素之间的关系不再发生改变，适合于顺序存储
- 每组有定义的下标都存在一个与其相对应的值

在数组上的基本操作

- 给定一组下标，取得相应的数据元素值
- 给定一组下标，修改相应的数据元素值

数组的基本操作定义

(1)构造n维数组 **InitArray**(&A, n, bound₁, ..., bound_n)

(2)销毁数组A **DestroyArray**(&A)

(3)取得指定下标的数组元素值

Value(A, &e, index₁, ..., index_n)

(4)为指定下标的数组元素重新赋值

Assign(&A, e, index₁, ..., index_n)



5.2 数组的顺序存储结构

一维数组

ElemType a[n];

基地址

b →



} L个单元

序号从0开始

$$\text{LOC}[i] = \text{LOC}[0] + i \times L$$

$$= b + i \times L$$

$$= c_0 + c_1 \times i$$

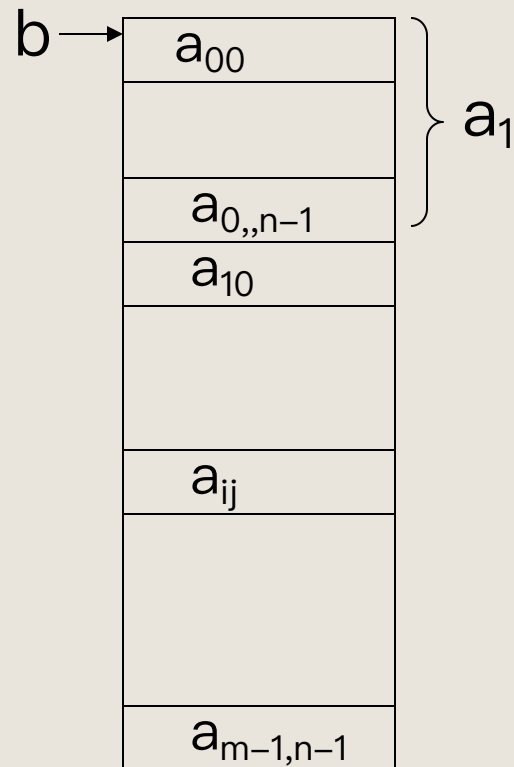
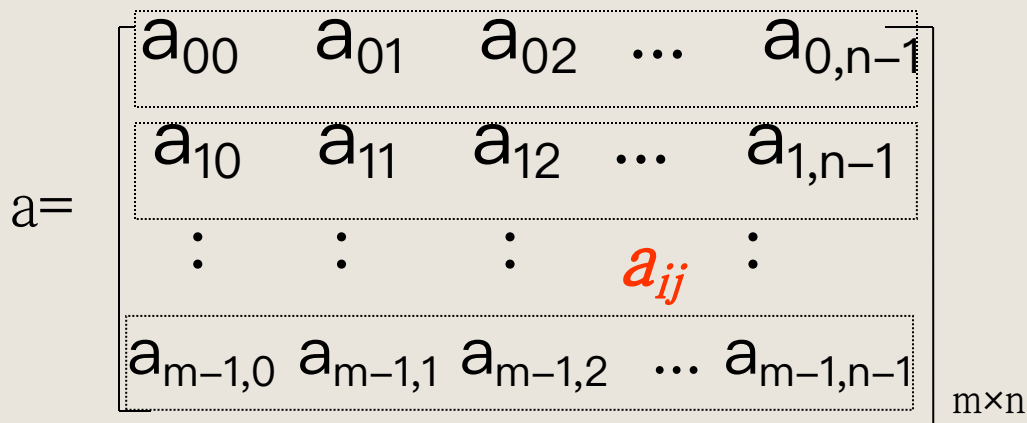
$$\begin{cases} c_1 = L & // \text{步长} \\ c_0 = b = \text{LOC}[0] \end{cases}$$

二维数组

ElemType a[m][n];

b1

b2: 第2维的长度



$$\text{LOC}[i,j] = \text{LOC}[0,0] + (n \times L \times i) + L \times j$$

$$= c_0 + c_1 \times i + c_2 \times j$$

$$\begin{cases} c_2 = L & \text{//第2维的步长} \\ c_1 = n \times L = n \times c_2 = b_2 \times c_2 \\ c_0 = b = \text{LOC}[0,0] \end{cases}$$

注：Pascal、C语言以行序为主序

“行序为主序” 即 “低下标优先”

“列序为主序” 即 “高下标优先”

n维数组

ElemType a[b₁][b₂] ... [b_n];

$$\text{LOC}[j_1, j_2, \dots, j_n] = c_0 + c_1 \times j_1 + c_2 \times j_2 + \dots + c_n \times j_n = c_0 + \sum_{i=1}^n c_i \times j_i$$

$$\begin{cases} c_n = L; & // \text{第} n \text{ 维的步长} \\ c_{i-1} = c_i \times b_i & (1 < i \leq n) \quad // \text{第} n-1 \text{ 维的步长} = \text{第} n \text{ 维的步长} \times \text{第} n \text{ 维的长度} \\ c_0 = b = \text{LOC}[0, 0, \dots, 0] & // \text{序号从} 0 \text{ 开始} \end{cases}$$

数组是一种随机存取结构:对任一元素定位时间相等.

思考: ElemType A[3..7, 0..9, 1..6, 5..12]

设每数组元素占用8个存储单元, 起始地址为1000, 求按低下标优先 (类似行优先) 顺序存储时,

$$\text{LOC}[6, 1, 4, 7] = ?$$

参考解答

ElemType A[3..7, 0..9, 1..6, 5..12]

维数: b1=5, b2=10, b3=6, b4=8

求LOC[6, 1, 4, 7]

相当于A[0..4, 0..9, 0..5, 0..7], 求LOC[3,1,3,2]

$LOC[3,1,3,2]=LOC[0,0,0,0]+c_1j_1+c_2j_2+c_3j_3+c_4j_4$ 每元素占用8字节

$$=1000+c_1j_1+c_2j_2+c_3j_3+8\times 2$$

$$=1000+c_1j_1+c_2j_2+8\times 8\times 3+8\times 2$$

$$=1000+c_1j_1+64\times 6\times 1+8\times 8\times 3+8\times 2$$

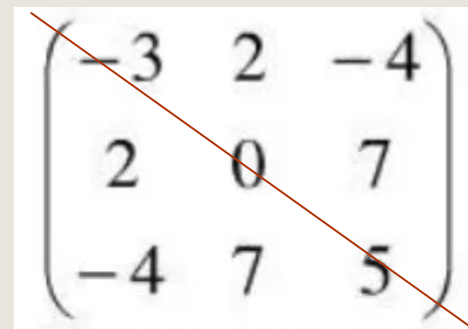
$$=1000+384\times 10\times 3+64\times 6\times 1+8\times 8\times 3+8\times 2$$

$$=13112$$

5.3 矩阵的压缩存储

目的是节省空间。

下标从1开始

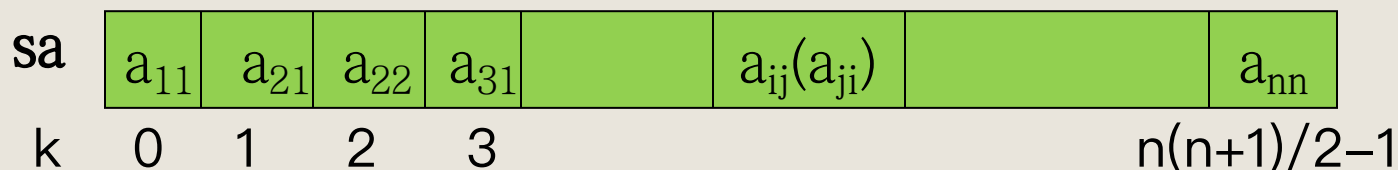

$$\begin{pmatrix} -3 & 2 & -4 \\ 2 & 0 & 7 \\ -4 & 7 & 5 \end{pmatrix}$$

5.3.1 对称矩阵

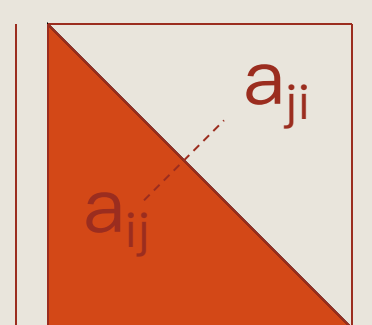
[特点] 在 $n \times n$ 的矩阵 a 中, 满足如下性质:

$$a_{ij} = a_{ji} \quad (1 \leq i, j \leq n)$$

[存储方法] 只存储下(或者上)三角(包括主对角线)的数据元素。共占用 $n(n+1)/2$ 个元素空间: $sa[0 \dots n(n+1)/2-1]$ 。

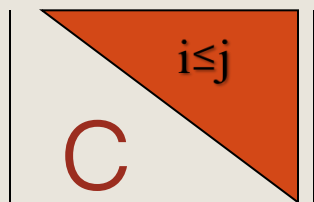


$$k = \begin{cases} i(i-1)/2 + j - 1 & \text{当 } i \geq j \text{ 下三角} \\ j(j-1)/2 + i - 1 & \text{当 } i < j \text{ 上三角} \end{cases}$$

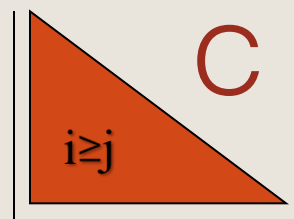


5.3.2 三角矩阵

[特点] 对角线以下(或者以上)的数据元素(不包括对角线)全部为常数 c 。



上三角矩阵



下三角矩阵

[存储方法] 重复元素 c 共享一个元素存储空间，共占用 $n(n+1)/2+1$ 个元素空间： $sa[1.. n(n+1)/2]$ 。 $sa[0]=C$



上三角矩阵

下三角矩阵

$$k = \begin{cases} (i-1) \times (2n-i+2)/2 + j - i + 1 & i \leq j \\ 0 & i > j \end{cases}$$

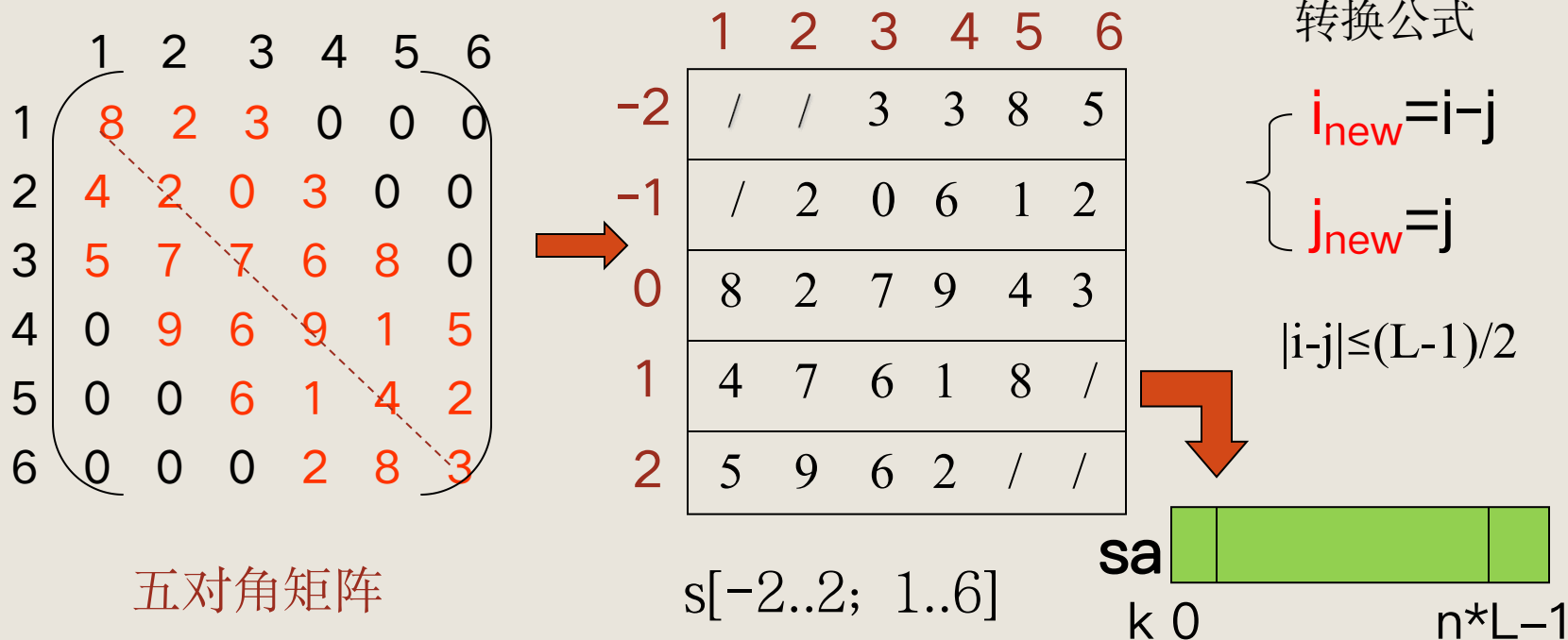
$$k = \begin{cases} i \times (i-1)/2 + j & i \geq j \\ 0 & i < j \end{cases}$$

5.3.3 带状矩阵（对角矩阵）

[特点] 在 $n \times n$ 的方阵中，非零元素集中在主对角线及其两侧共 L (奇数)条对角线的带状区域内 — L 对角矩阵。

[存储方法] 只存储带状区内的元素。

1) 以对角线的顺序存储,共 $n \times L$ 个元素。



2) 只存储带状区内的元素。从上一行的主对角线元素 $a_{i-1,i-1}$ 到本行的主对角线元素 a_{ii} 这一段最多有 L 个元素，共 $(n-1)L+1$ 个元素。sa[0..(n-1)L]

存储地址计算: $k=(i-1)L+(j-i)$ //将主对角线元素 a_{ii} 映射到 $(i-1)L$
 $1 \leq i, j \leq n$ $|i-j| \leq (L-1)/2$

8	2	3	0	0	0
4	2	0	3	0	0
5	7	7	6	8	0
0	9	6	9	1	5
0	0	6	1	4	2
0	0	0	2	8	3

sa	8	2	3	/	4	2	0	3	5	7
k	0	1	2	3	4	5	6	7	8	9
	7	6	8	9	6	9	1	5	6	1
	10	11	12	13	14	15	16	17	18	19
	4	2	/	2	8	3				
	20	21	22	23	24	25				

5.3.4 随机稀疏矩阵

[特点] 大多数元素为零。 稀疏因子: $\delta = t/(m \times n) \leq 0.05$

[常用存储方法] 记录每一非零元素(i, j, a_{ij})

节省空间, 但丧失随机存取功能

顺序存储: 三元组表

链式存储: 十字(正交)链表

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
0	0	28	0	0	0

5.3.4.1 三元组顺序表

[类型定义]

```
#define MAXSIZE 1000

//设定非零元素最大值

typedef struct {
    int i, j;
    ElemType e;
}Triple;

typedef struct {
    Triple data[MAXSIZE+1];
    //三元组表, data[0]未用
    int mu, nu, tu;
    //行数、列数、非零元个数
}TSMatrix;
```

行为主序



	i	j	e
1	1	1	15
2	1	4	22
3	1	6	-15
4	2	2	22
5	2	3	3
6	3	4	-6
7	5	1	91
8	6	3	28
m			

[应用例] 求转置矩阵

a.data				b.data			
	i	j	e		i	j	e
1	1	1	15	转置 →	1	1	15
2	1	4	22		2	5	91
3	1	6	-15		3	2	22
4	2	2	22		4	3	3
5	2	3	3		5	3	28
6	3	4	-6		6	4	22
7	5	1	91		7	4	-6
8	6	3	28		8	6	-15

如何保证转置后仍然行序优先？

先在a中找列号为1的放到b中，再找列号为2的放到b中，依次类推

算法1 { $O(n \times t)$ }

求矩阵a的转置矩阵b

```
void TransMatrix(TSMatrix &b, TSMatrix a)
{
    b.mu=a.nu; b.nu=a.mu; b.tu=a.tu;
    if (b.tu ) {
        q=1; //指示b中存放数据的位置, 初值为1
        for ( col=1; col<=a.nu; col++ ) //尝试找到列为1到nu的元素
            for ( p=1; p<=a.tu; p++ ) //对a的每个三元组检查
                if ( a.data[p].j==col) { //找列号为col的三元组
                    b.data[q].i =a.data[p].j;
                    b.data[q].j =a.data[p].i;
                    b.data[q].e =a.data[p].e;
                    q++;           //修正q值
                }
    }
} //TransMatrix
```

算法2 快速转置法 { $O(n+t)$ }

引入两个辅助向量:

先统计a中每列非零个数, 再算出转置后的位置

num[1 : a.nu]: a中每列的非零元素个数,

cpot[1 : a.nu]: a中每列的第一个非零元素在b中的位置

a中的列号

col num[col] cpot[col]

1	2	1
2	1	3
3	2	4
4	2	6
5	0	8
6	1	8

	i	j	e
1	1	1	15
2	1	4	22
3	1	6	-15
4	2	2	22
5	2	3	3
6	3	4	-6
7	5	1	91
8	6	3	28

转置

	i	j	e
1	1	1	15
2	1	5	91
3	2	2	22
4	3	2	3
5	3	6	28
6	4	1	22
7	4	3	-6
8	6	1	-15

cpot[1]=1

cpot[col]=cpot[col-1] + num[col-1]
($2 \leq \text{col} \leq n$)

Statue **FastTransMatrix**(TSMatrix &b, TSMatrix a)

```
{  b.mu=a.nu; b.nu=a.mu; b.tu=a.tu;
    if ( b.tu ) {
        {    for(col=1; col<= a.nu; ++col)    //num清零
                num[col]=0;
            for (t=1; t<=a.tu; ++t)    //对a.tu个非零元素按列号计数
                ++num[a.data[t].j];
            cpot[1]=1;    //生成cpot, 计算每列第1元素转置后的位置
            for ( col=2; col<=a.nu; ++col )
                cpot[col]=cpot[col-1]+num[col-1];
            for ( p=1; p<=a.tu; ++p ) { //对a.tu个非零元素转置
                col=a.data[p].j; q=cpot[col];
                b.data[q].i =a.data[p].j;
                b.data[q].j =a.data[p].i;
                b.data[q].e =a.data[p].e;
                ++cpot[col] ;
            }
        }
    }
    return OK;
}
} //FastTransMatrix
```

5.3.4.2 行逻辑链接的顺序表

便于随机存取任意一行的非零元素。

```
typedef struct {  
    Triple data[MAXSIZE+1];  
    int rpos[MAXRC+1];  
    int mu, nu, tu;  
}RLSMatrix;
```

```
RLSMatrix M;
```

M.rpos、		M.data		
s		i	j	e
1	1	1	1	15
2	4	1	4	22
3	6	1	6	-15
4	0	2	2	22
5	7	2	3	3
6	8	3	4	-6
		5	1	91
		6	3	28

这种方式可以便于某些运算，如：稀疏矩阵相乘等。

5.3.4.3 十字（正交）链表

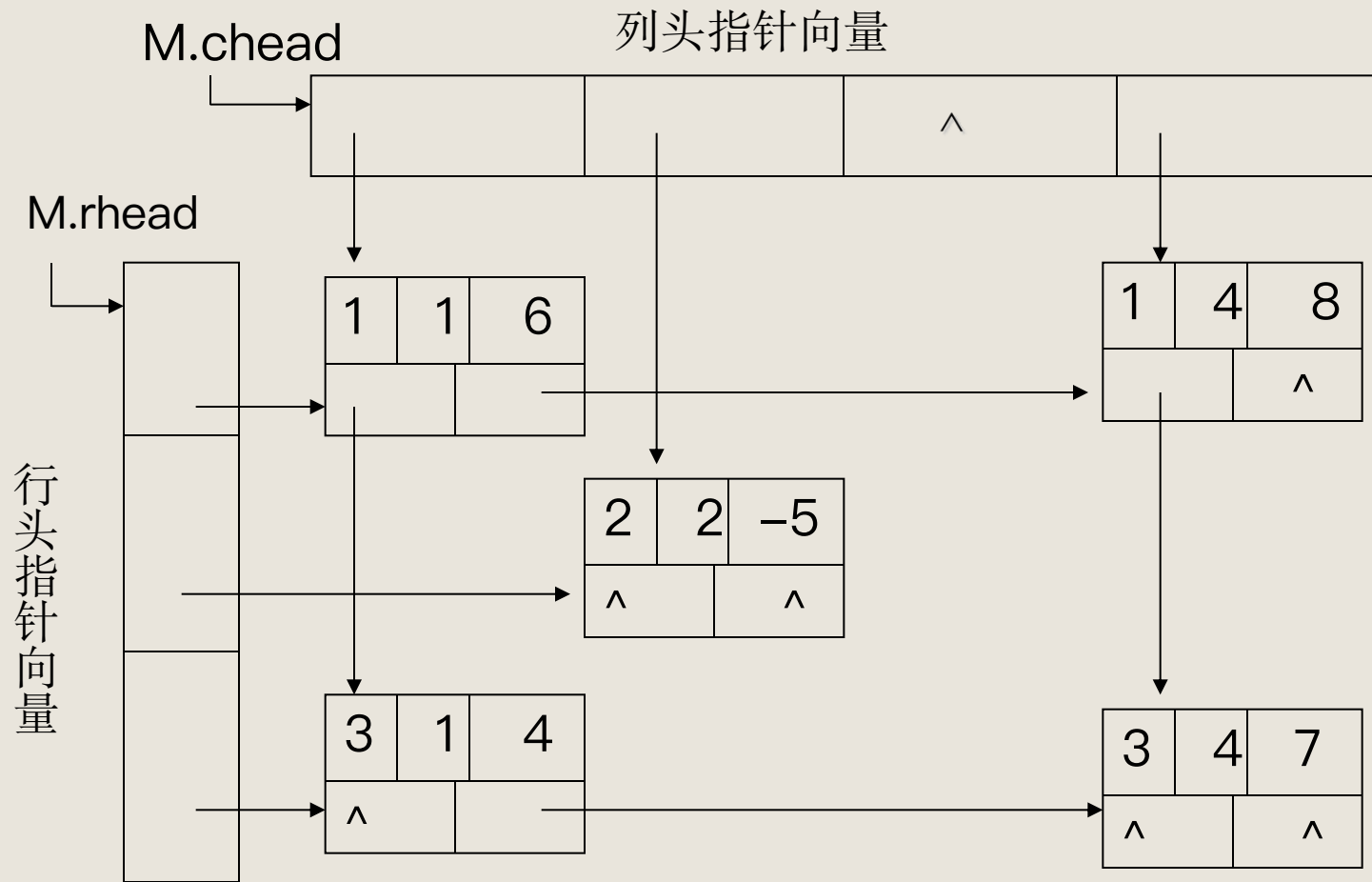
[特点] 在行、列两个方向上，将非零元素链接在一起。克服三元组表在矩阵的非零元素位置或个数经常变动时的使用不便。

[类型定义]

```
typedef struct OLNode
{ int      i, j;
  ElemType e;
  struct OLNode * right, * down;
}OLNode, * OLink;
```

```
typedef struct {
  OLink  * rhead, * chead;
  int     mu, nu, tu;
}CrossList;
```

i	j	e
down		right



CrossList M;

6	0	0	8
0	-5	0	0
4	0	0	7

[算法示例] 从终端接收信息建立稀疏矩阵的十字链表

算法思想:

- 1) 赋值矩阵的行数 μ 、列数 ν 和非零元素个数 t ;
- 2) 申请行、列头指针向量, 将各行、列链表置为空链表;
- 3) 读入一个非零元素的行号 i 、列号 j 、值 e
 - 3.1) 建立该元素的结点, 赋值其三元组
 - 3.2) 寻找该结点在行表中的插入位置并插入
 - 3.3) 寻找该结点在列表中的插入位置并插入
 - 3.4) 存在下一个非零元素, 转3; 否则, 结束。

5.4 广义表（列表）的定义和表示方法

概念 广义表是由零个或多个**原子**或者**子表**组成的有限序列。可以记作： $LS=(d_1, d_2, \dots, d_n)$

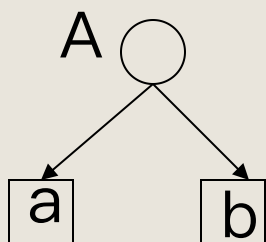
原子：逻辑上不能再分解的元素。

子表：作为广义表中元素的广义表。

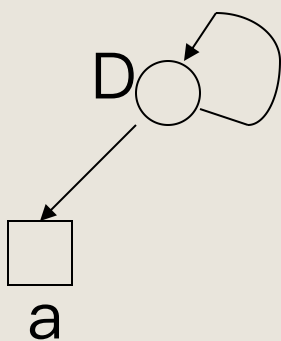
与线性表的关系

广义表中的元素全部为原子时即为线性表。**线性表是广义表的特例，广义表是线性表的推广。**

广义表的图形表达方法



$A=(a,b)$

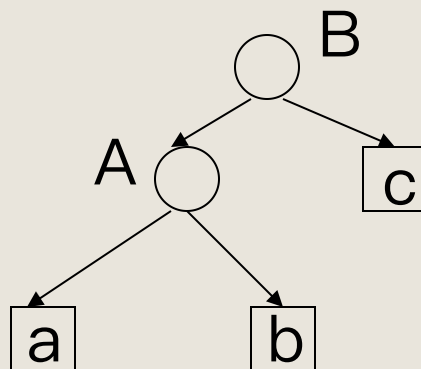


含有递归

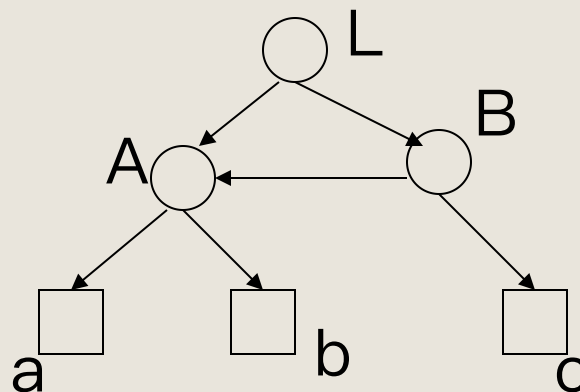
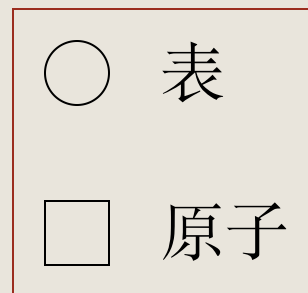
$D=(a,D)$

$= (a, (a,D))$

$= \dots$



$B=(A,c)$



含有共享

$L=(A,B)$

$= (A, (A, c))$

$= ((a,b), ((a,b), c))$

规定所有表都有名字时，广义表的表示方法

习惯上用大写字母表示广义表的名称，用小写字母表示原子。

例 $A(a,b)$

$B(A(a,b), c)$

$L(A(a,b), B(A(a,b),c))$

$D(a, D(a, D(...)))$

广义表的表示方法和相关术语

	表长	表深	表头	表尾
$A = ()$	0	1	—	—
$B = (a, A) = (a, ())$	2	2	a	(())
$C = ((a,b), c, d)$	3	2	(a,b)	(c,d)
$D = (a, D) = (a, (a, (a, \dots)))$	2	∞	a	(D)

表的长度 表中的元素（第一层）个数。

表的深度 表中元素的最深嵌套层数。

表头 表中的第一个元素。

表尾 除第一个元素外，剩余元素构成的广义表。

任何一个非空广义表的表尾必定仍为广义表。

$L(A(a,b), B(A(a,b),c))$

获取表头的操作: $\text{GetHead}(L) = A(a,b)$

获取表尾的操作: $\text{GetTail}(L) = (B(A(a,b),c))$

广义表结构的分类

纯表: 与树型结构对应的广义表。

再入表: 允许结点共享的广义表。

递归表: 允许递归的广义表。

递归表 \supset 再入表 \supset 纯表 \supset 线性表

广义表的应用

如: 程序的语句结构; m元多项式的表示

广义表的应用

如：程序的语句结构； m元多项式的表示

$$\begin{aligned} P(x, y, z) &= x^{10}y^3z^2 + 2x^6y^3z^2 + 3x^5y^2z + 2yz + 15 \\ &= (x^{10}y^3 + 2x^6y^3)z^2 + (3x^5y^2 + 2y)z + 15 \end{aligned}$$

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

$$A = y \left((C, 3) \right)$$

$$C = x \left((1, 10), (2, 6) \right)$$

$$B = y \left((D, 2), (2, 1) \right)$$

$$D = x \left((3, 5) \right)$$

先将 z 设为主元，然后依次在系数多项式中把 y 和 x 作为主元

抽象数据类型定义

ADT GList {

数据对象: $D = \{e_i \mid i=1,2,\dots,n; n \geq 0; e_i \in \text{AtomSet} \text{ 或 } e_i \in \text{GList},$

AtomSet 为某个数据对象}

数据关系: $R = \{ \langle e_{i-1}, e_i \rangle \mid e_{i-1}, e_i \in D, 2 \leq i \leq n \}$

基本操作:

InitGList(&L);

操作结果: 创建空的广义表L。

CreateGList(&L, S);

初始条件: S是广义表的书写形式串。

操作结果: 由S创建广义表L。

.....

} ADT GList

基本操作

- 结构的创建和销毁

InitGLList(&L); DestroyGLList(&L);
CreateGLList(&L, S); CopyGLList(&T, L);

- 状态函数

GListLength(L); GListDepth(L);
GListEmpty(L); GetHead(L); GetTail(L);

- 插入和删除操作

InsertFirst_GL(&L, e);
DeleteFirst_GL(&L, &e);

- 遍历

Traverse_GL(L, Visit());

5.5 广义表的存储结构

5.5.1 方法 1 一头尾链表形式

[类型定义]

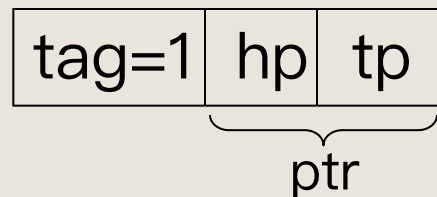
```
typedef enum {ATOM, LIST} ElemTag;
// ATOM==0:原子; LIST==1: 子表
typedef struct GLNode{
    ElemTag tag;
    union {
        AtomType atom;
        struct {
            struct GLNode *hp, *tp;
        } ptr;
    }
} * GList1;
```

$GL = (a_1, a_2, \dots, a_n)$

$head(GL) = a_1$

$tail(GL) = (a_2, \dots, a_n)$

表结点



原子结点



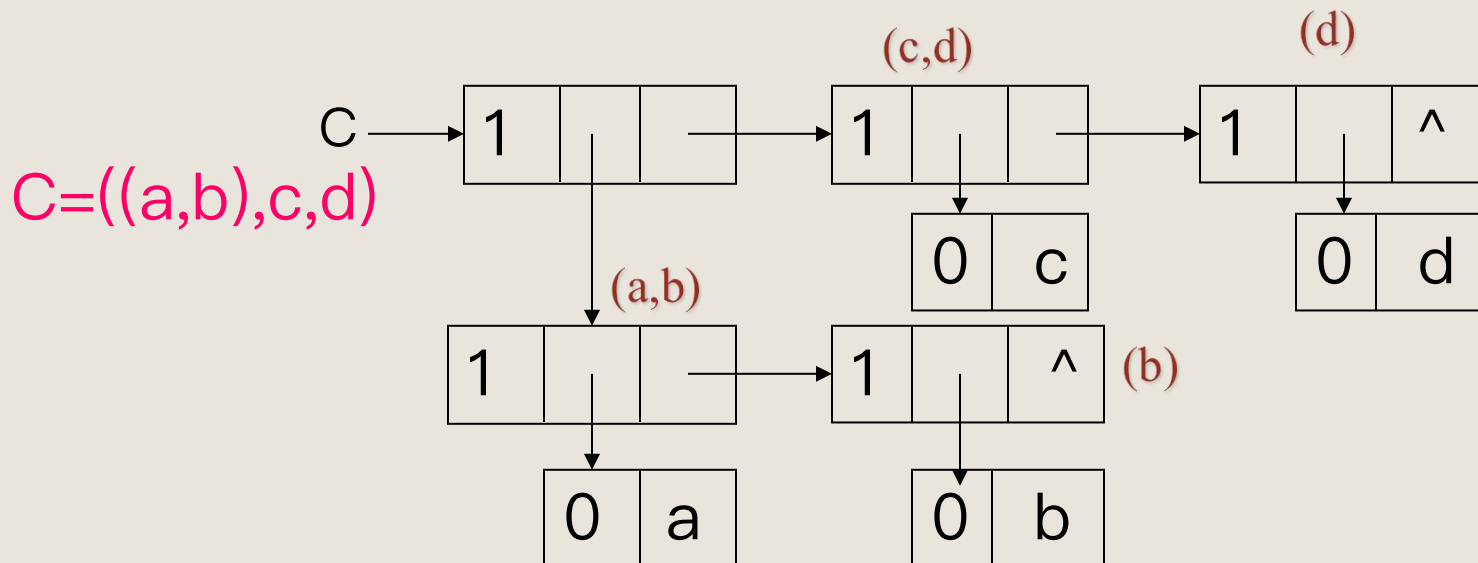
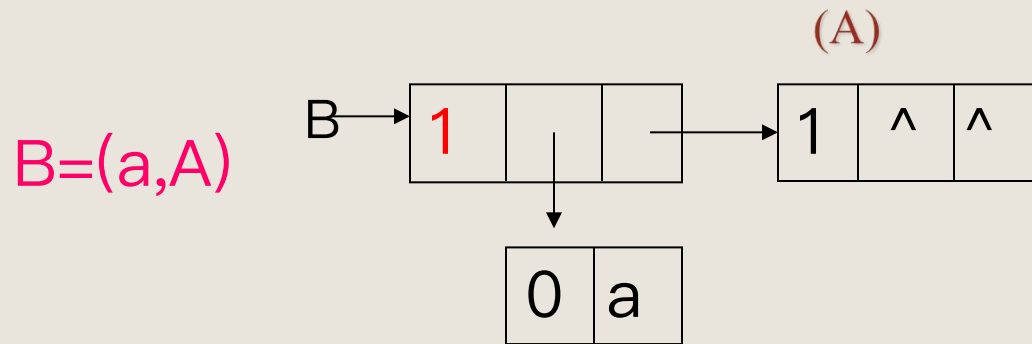
hp: 指示表头的指针

tp: 指示表尾的指针

[示例]

GList1 A, B, C

A=() A = NULL



5.5.2 方法2—扩展的线性链表形式

[类型定义]

(a_1, a_2, \dots, a_n)

```
typedef enum {ATOM, LIST} ElemTag;  
// ATOM==0:原子; LIST==1: 子表  
typedef struct GLNode{  
    ElemTag tag;  
    union {  
        AtomType atom;  
        struct {  
            struct GLNode *hp;  
        }  
    }  
    struct GLNode *tp;  
} * GList2;
```

表结点

tag=1	hp	tp
-------	----	----

原子结点

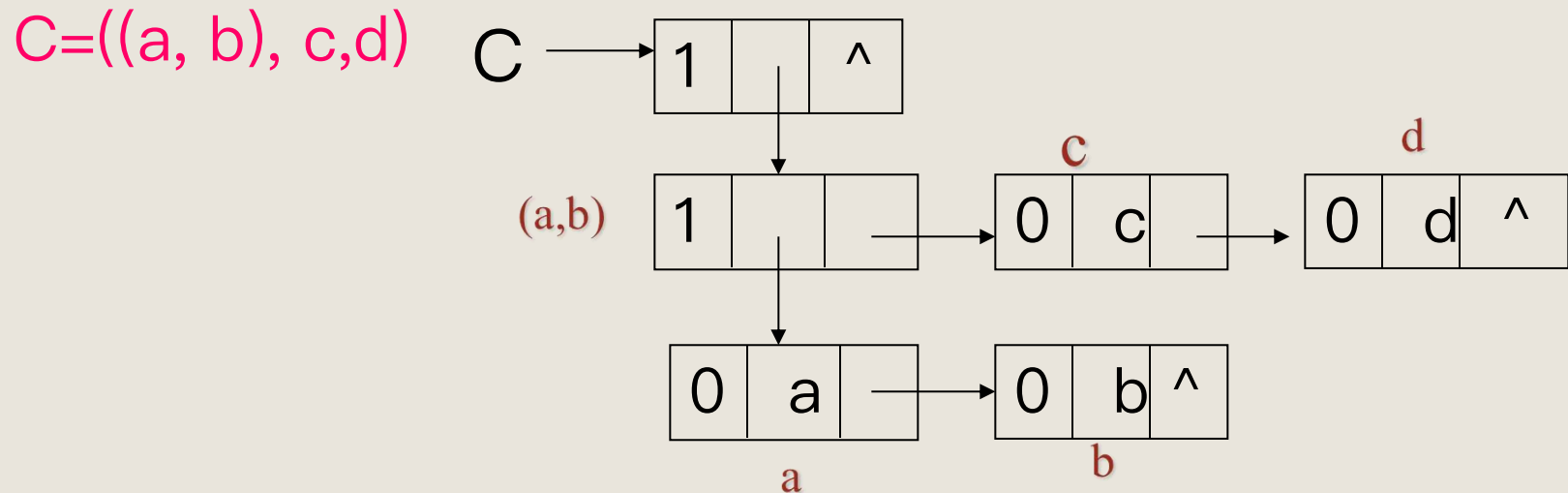
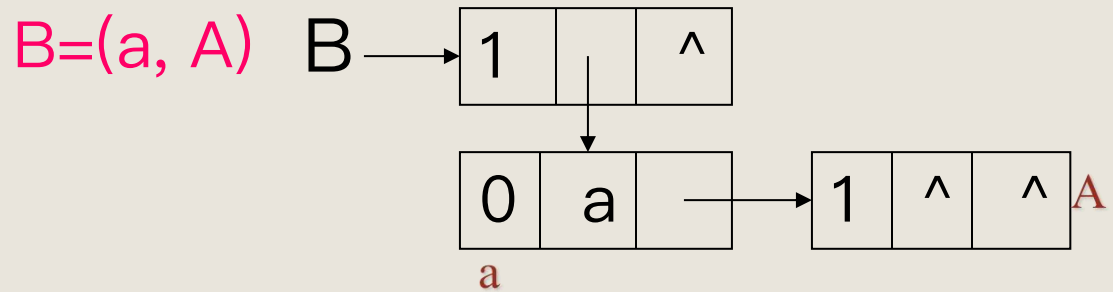
tag=0	atom	tp
-------	------	----

hp: 指向表头的指针

tp: 指向同一层的下一个结点

[示例]

GList2 A, B, C



5.6 广义表的递归算法

广义表的特点：定义是递归的。

示例约定：非递归表且无共享子表。

[示例1] 计算广义表的深度

方法一 分析表中各元素（子表） $LS = (a_1, a_2, \dots, a_n)$

- 求深度的递归函数

基本项： $DEPTH (LS) = 1$ 当LS为空表时

$DEPTH (LS) = 0$ 当LS为原子时

归纳项： $DEPTH (LS) = 1 + \max_{1 \leq i \leq n} \{DEPTH(a_i)\}$ $n \geq 1$

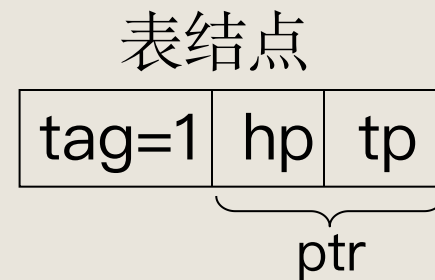
- 算法描述

```

int GListDepth(GList1 L)
{
    //采用头尾链表存储结构，求广义表L的深度
    if (!L) return 1; //空表
    if (L->tag==ATOM) return 0; //单原子
    for (max=0, pp=L; pp; pp=pp->ptr.tp) {
        dep=GListDepth (pp->ptr.hp);
        if (dep>max) max=dep;
    }
    return max+1; //非空表的深度是各元素深度的最大值加1
} //GListDepth

```

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



方法二 分析表头和表尾

一求深度的递归函数

$$\text{depth}(LS) = \begin{cases} 1 & , \text{当} LS \text{为空表} \\ 0 & , \text{当} LS \text{为单原子} \\ \max\{\text{depth}(\text{head}(LS))+1, \text{depth}(\text{tail}(LS))\} & \text{其它情况} \end{cases}$$

一算法描述

其它情况

```
int GListDepth(GList1 L)
{
    if (!L) return 1; //空表
    if (L->tag==ATOM) return 0; //单原子
    dep1=GListDepth (L->ptr.hp)+1;
    dep2=GListDepth (L->ptr.tp);
    if (dep1>dep2) return dep1;
    else return dep2;
} //GListDepth
```

[示例2] 复制广义表

– 复制操作的递归定义

基本项: InitGList(NEWLS)

置空广义 当LS为空表时;

复制单原子结点 当LS为原子时;

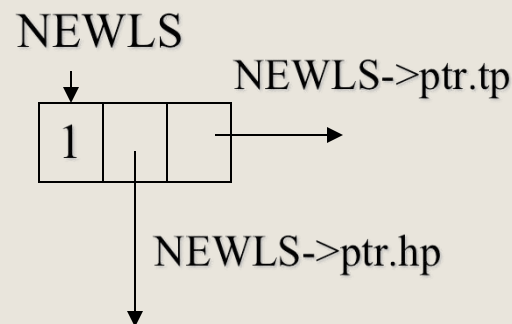
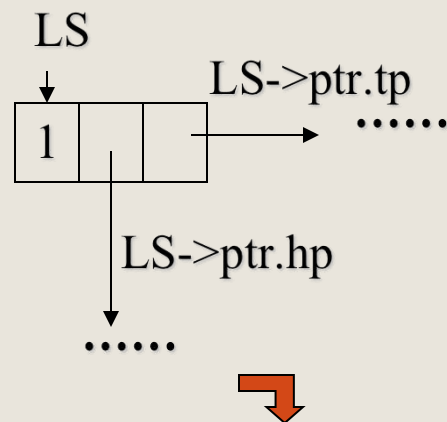
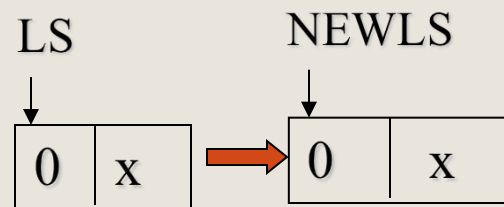
归纳项: 建表结点;

复制表头;

复制表尾;

– 算法描述CopyGLists

LS=NULL \Rightarrow NEWLS=NULL




```
Status CopyGList(GList1 &T, GList1 L)
{
    if (!L) T=NULL; //复制空表
    else {
        T=(GList1)malloc(sizeof(GLNode));
        if (!T) exit(OVERFLOW);
        T->tag=L->tag;
        if (L->tag==ATOM) T->atom=L->atom; //复制单原子
        else {
            CopyGList(T->ptr.hp, L->ptr.hp);
            CopyGList(T->ptr.tp, L->ptr.tp);
        }
    }
    return OK;
} //CopyGList
```



本章学习要点

- 掌握数组类型的特点以及在高级编程语言中的两种存储表示和实现方法，并熟练掌握低下标优先时指定下标的元素在存储结构中的地址计算方法。
- 掌握矩阵压缩存储的常用方法。
- 掌握广义表的结构特点和表长、表深、表头、表尾的定义。
- 了解广义表的存储表示方法，学会对非空广义表进行分解的两种分析方法：即可将一个非空广义表分解为表头和表尾两部分或者分解为 n 个子表。巩固递归算法的设计思想。*