DS—第五章

数组和广义表

Arrays & GList

内容回顾

- 串的相关概念
- 串的ADT定义
- 串的表示与实现

  I.定长顺序表示

  II.堆分配存储表示

  III.块链存储表示
- 模式匹配算法

  I.基本匹配算法

  II.改进匹配算法(KMP)

  III.模式串的next[]

  IV.改进的模式串的nextval[]

- 数组的定义
- 数组的顺序表示和实现
- 矩阵的压缩存储
- 广义表的概念、性质
- 广义表的存储结构、相关递归算法

河北师范大学软件学院
Software College of Hebei Normal University

- 数组的定义
- 数组的顺序表示和实现
- 矩阵的压缩存储
- 广义表的概念、性质
- 广义表的存储结构、相关递归算法

5.1 数组的定义

数组：按一定格式排列起来的

具有**相同类型**的数据元素的集合。

一维数组：若线性表中的数据元素为非结构的简单

元素，则称为一维数组。

一维数组的逻辑结构：线性结构。定长的线性表。

声明格式： 数据类型 变量名称[长度]；

例：int num[5] = {0，1，2，3，4};

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

$A=(\alpha_0,\alpha_1,\ldots,\alpha_p)$ ($p = m$-1 or n-1)

$\alpha_j=(a_{0j},a_{1j},\ldots,a_{m-1,j})$ $0\le j\le n-1$

$\alpha_i=(a_{i0},a_{i1},\ldots,a_{i,n-1})$ $0\le i\le m-1$

二维数组：若一维数组中的数据元素又是一维数组结构，则称为二维数组。

二维数组逻辑结构

- 非线性结构 每一个数据元素既在一个行表中，又在一个列表中。
- 线性结构 定长的线性表 该线性表的每个数据元素也是一个定长的线性表。

声明格式： 数据类型 变量名称[行数] [列数] ；

例：int num[5] [8] ;

在 C 语言中，一个二维数组类型也可以定义为一维数组类型（其分量类型为一维数组类型），即：

```
typedef  elemtype  array2[m][n];
```

等价于:

```
typedef elemtype array1[n];
typedef  array1 array2[m];
```

三维数组：若二维数组中的元素又是一个一维数组结构，则称作三维数组。

⋮

n 维数组：若 *n* -1 维数组中的元素又是一个一维数组结构，则称作 *n* 维数组。

结论

数组结构又是线性表结构的扩展。

数组特点：结构固定：定义后维数和维界不再改变。

数组基本操作：除了结构的初始化和销毁之外，只有取元素和修改元素值的操作。

数组的抽象数据类型的定义

ADT Array {

数据对象：D={$a_{j1\,j2\,\ldots\,jn}$ | j_i = 0, ..., b_i -1, i = 1, 2, .., n,

n (>0) 称为数组的维数，

b_i 是数组第 i 维的长度，

j_i 是数组元素的第 i 维下标，

$a_{j1\,j2\,\ldots\,jn} \in$ ElemSet }

数据关系：R = {R1, R2, ..., Rn}

Ri = {<$a_{j1\ldots ji\ldots jn}$, $a_{j1\ldots ji+1\ldots jn}$ >

| $0 \le j_k \le b_k$ -1, $1 \le k \le n$ 且 $k \ne i$, $0 \le j_i \le b_i$ - 2,

$a_{j1\ldots ji\ldots jn}$, $a_{j1\ldots ji+1\ldots jn} \in$ D, i = 2, ..., n }

二维数组的抽象数据类型的数据对象和数据关系的定义

数据对象:

$$D = \{a_{ij} \mid 0 \le i \le b_1 - 1,\ 0 \le j \le b_2 - 1\}$$

数据关系:

$$R = \{ ROW, COL \}$$

$$ROW = \{< a_{i,j}, a_{i+1,j} > \mid 0 \le i \le b_1 - 2,\ 0 \le j \le b_2 - 1\}$$

$$COL = \{< a_{i,j}, a_{i,j+1} > \mid 0 \le i \le b_1 - 1,\ 0 \le j \le b_2 - 2\}$$

基本操作：

InitArray(&A, n, bound$_1$, ..., bound$_n$)

操作结果：若维数 n 和各维长度合法，则构造相应的数组 A，并返回 OK。

DestroyArray(&A)

操作结果：销毁数组 A。

Value(A, &e, index$_1$, ..., index$_n$)

初始条件：A 是 n 维数组，e 为元素变量。

操作结果：若各下标不超界，则 e 赋值为所指定的 A 的元素值，并返回 OK。

Assign(&A, e, index$_1$, ..., index$_n$)

初始条件：A 是 n 维数组，e 为元素变量。

操作结果：若下标不超界，则将 e 的值赋给所指定的 A 的元素，并返回 OK。

} ADT Array

- 数组的定义
- 数组的顺序表示和实现
- 矩阵的压缩存储
- 广义表的概念、性质
- 广义表的存储结构、相关递归算法

5.2 数组的顺序表示和实现

因为
- 数组特点：结构固定——维数和维界不变。
- 数组基本操作：初始化、销毁、取元素、改元素值。一般不做插入和删除操作。

所以： 一般都是采用顺序存储结构来表示数组。

注意：数组可以是多维的，但存储数据元素的内存单元地址是一维的，因此，在存储数组结构之前，需要解决将多维关系映射到一维关系的问题。

两种顺序存储方式
- 以行序为主序 (低下标优先)
- 以列序为主序 (高下标优先)

以行序为主序存放:

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

0	a_{00}
1	a_{01}
⋮	…….
n -1	$a_{0,n-1}$
n	a_{10}
	a_{11}
	……..
	$a_{1,n-1}$
	……….
	$a_{m-1,0}$
	$a_{m-1,1}$
	……..
m*n -1	$a_{m-1,n-1}$

二维数组中任一元素 a_{ij} 的存储位置

$$LOC(i, j) = \boxed{LOC(0, 0) + (b_2 \times i + j) \times L}$$

基地址或基址　　二维数组的映象函数

某个元素的地址就是它前面所有行所占的单元加上它所在行前面所有列元素所占的单元数之和。

按列序为主序存放

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

0	a_{00}
1	a_{10}
⋮	…….
m -1	$a_{m-1,0}$
m	a_{01}
	a_{11}
	……..
	$a_{m-1,1}$
	……….
	$a_{0,n-1}$
	$a_{1,n-1}$
	……..
m*n -1	$a_{m-1,n-1}$

二维数组中任一元素 a_{ij} 的存储位置

$$\mathrm{LOC}(i, j) = \mathrm{LOC}(0, 0) + (b_1 \times j + i) \times L$$

某个元素的地址就是它前面所有列所占的单元加上它所在列前面所有行元素所占的单元数之和。

例 1：一个二维数组 A，行下标的范围是 1 到 6，列下标的范围是 0 到 7，每个数组元素用相邻的 6 个字节存储，存储器按字节编址。那么，这个数组的体积是<u>288</u>个字节。

答： Volume = m×n×L

= (6 – 1 + 1) ×(7 – 0 + 1) ×6

= 48×6 = 288

例 2： 〖某校计算机系考研题〗

设数组 A[0…59, 0…69] 的基地址为 2048，每个元素占 2 个存储单元，若以列序为主序顺序存储，则元素 A[31, 57] 的存储地址为___8950___。

解：LOC(*i*, *j*) = LOC(31, 57)

$= LOC(0, 0)+(b_1 \times j + i) \times L$

$= 2048 + (60 \times 57 + 31) \times 2$

$= 8950$

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,69} \\ a_{10} & a_{11} & \cdots & a_{1,69} \\ \cdots\cdots & \cdots\cdots & \cdots\cdots & \\ \cdots\cdots & a_{31,57} & \cdots\cdots & \\ \cdots\cdots & \cdots\cdots & \cdots\cdots & \\ a_{59,0} & a_{59,1} & \cdots & a_{59,69} \end{bmatrix}$$

同理，对三维数组$A[b_1][b_2][b_3]$,可以看成b_1个$b_2 \times b_3$的二维数组，若首元素的存储地址为LOC[0,0,0],则元素a_{i00}的存储地址为

$$LOC(i,0,0)=LOC(0,0,0)+(i\times b_2\times b_3)*L$$

这是因为该元素之前有i个$b_2\times b_3$的二维数组，所以

$$LOC(i,j,k)=LOC(0,0,0)+(i\times b_2 \times b_3 +j\times b_3 +k)*L$$

推广到一般情况，可得到 n 维数组数据元素存储位置的映像关系

$$LOC(j_1,j_2,\ldots,j_n)=LOC(0,0,0)+(b_2\times\ldots\times b_n\times j_1+ b_3\times\ldots\times b_n\times j_2+\ldots+b_n\times j_{n-1}+ j_n) \times L$$

即

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

$$\text{其中} c_n = L, \quad c_{i-1} = b_i \times c_i, \quad 1 < i \le n.$$

称为 n 维数组的映像函数。数组元素的存储位置是其下标的线性函数

数组的顺序表示及相关说明

```
typedef   struct{
  ElemType  *base;  //存放元素的基址
  int dim;                //维数
  int *bounds;           //等价整形数组，存各维长度
  int *constants;       //每变化一维的跨度
}Array;
```

va_list ap;

va_start(ap,dim):使ap 指向dim后面的实参表;

va_arg(ap,int):取出ap所指向的int型数据后，
ap++。

va_end(ap):空语句

```
Status InitArray(Array  &A , int dim , ...){
 if (dim<1 || dim>8 ) return  ERROR ;
 A.dim=dim;
A.bounds=(int*)malloc(dim*sizeof(int));
if( !A.bounds) exit(OVERFLOW);
elemtotal=1;
va_start(ap,dim);
for(i=0 ; i<dim ; i++){
   A.bounds[i]=va_arg(ap , int );
   if(A.bounds[i]<1) return UNDERFLOW;
  elemtotal*=A.bounds[i];  }
va_end(ap) ;
A.base=(ElemType*)malloc(elemtotal*sizeof(ET));
if( ! A.base) exit(OVERFLOW);
A.constants=(int *)malloc(dim *sizeof(int));
if( !A.constants) exit(OVERFLOW);
A.constants[dim-1]=1;
for(i=dim-2 ; i >=0 ; i--)
     A.constants[i]=A.constants[i+1]*A.bounds[i+1];
return OK;          }
```

InitArray(AA,3,7,4,5)

bounds

7	4	5

elemtotal 140

base

				… …		

constants

20	5	1

```
Status  Locate(Array A,va_list ap,int &off){
    off=0;
     for(i=0;i<A.dim;++i){
          ind=va_arg(ap,int);
          if(ind<1||ind>=A.bounds[i])return  false;
          off+=A.constants[i]*ind;
     }
     return ok;
}
```

Locate (AA,ap,off);

ap:3,2,4

off= **60 70 74**

ind= **3 2 4**

constants			
constants	**20**	**5**	**1**

- 数组的定义
- 数组的顺序表示和实现
- 矩阵的压缩存储
- 广义表的概念、性质
- 广义表的存储结构、相关递归算法

5.3 矩阵的压缩存储

矩阵定义： 一个由 $m \times n$ 个元素排成的 m 行（横向）n 列（纵向）的表。

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

矩阵的常规存储：

将矩阵描述为一个二维数组。

矩阵的常规存储的特点：

可以对其元素进行随机存取；

矩阵运算非常简单；存储的密度为 1。

不适宜常规存储的矩阵： 值相同的元素很多且呈某种规律分布；零元素多。

矩阵的压缩存储： 为多个相同的非零元素只分配一个存储空间；对零元素不分配空间。

5.3.1 特殊矩阵

特殊矩阵：元素值的排列具有一定规律的矩阵。

对称矩阵、下、上三角矩阵、对角线矩阵等

1、对称矩阵

在一个 n 阶方阵 A 中，若元素满足下述性质：

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

则称 A 为对称矩阵。

$$\begin{bmatrix} 1 & 5 & 1 & 3 & 7 \\ 5 & 0 & 8 & 0 & 0 \\ 1 & 8 & 9 & 2 & 6 \\ 3 & 0 & 2 & 5 & 1 \\ 7 & 0 & 6 & 1 & 3 \end{bmatrix} \qquad \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

对称矩阵的存储结构

对称矩阵上下三角中的元素数均为：　$n(n+1)/2$

可以行序为主序将元素存放在一个一维数组 $sa[n(n+1)/2]$ 中。

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

以行序为主序存储下三角：

	a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	…	a_{n1}	…	a_{nn}
$k=$	0	1	2	3	4		$n(n-1)/2$		$n(n+1)/2-1$

以行序为主序存储下三角：

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

a_{11}	a_{21}	a_{22}	a_{31}	…	a_{n1}	…	a_{nn}

k= 0　1　2　3　　$n(n$-1)/2

则 a_{ij} 和 $sa[k]$ 存在着一一对应关系：

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

a_{ij} 前的 i -1 行有 1+2+…+(i -1)= i(i -1)/2 个元素，在第 i 行上有 j 个元素。

因为$a_{ij} = a_{ji}$，所以只要交换关系式中的 i 和 j 即可。

2、三角矩阵

以主对角线划分，三角矩阵有上（下）三角两种。上（下）三角矩阵的下（上）三角（不含主对角线）中的元素均为常数。在大多数情况下，三角矩阵常数为零。

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

上三角矩阵　　　　**下三角矩阵**

三角矩阵的存储：**除了存储主对角线及上（下）三角中的元素外，再加一个存储常数 c 的空间。**

3、 对角矩阵

$$
\begin{bmatrix}
a_{11} & a_{12} & & & 0 \\
a_{21} & a_{22} & a_{23} & & \\
 & & \ddots & & \\
 & & & & a_{n-1,n} \\
0 & & & a_{n,n-1} & a_{nn}
\end{bmatrix}
$$

对角矩阵可按行优先顺序或对角线的顺序，将其压缩存储到一维数组中，且也能找到每个非零元素和向量下标的对应关系。

5.3.2 稀疏矩阵

稀疏矩阵： 设在 $m \times n$ 的矩阵中有 t 个非零元素。

令 $\delta = t/(m \times n)$

当 $\delta \le 0.05$ 时称为**稀疏矩阵**。

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

M 由

{(1,2,12), (1,3,9), (3,1,-3), (3,6,14), (4,3,24), (5,2,18), (6,1,15), (6,4,-7) }

和矩阵维数 (6, 7) 唯一确定。

三元组 (i, j, a_{ij}) 惟一确定矩阵的一个非零元。

压缩存储原则： 存各非零元的值、行列位置和矩阵的行列数。

三元组的不同表示方法可决定稀疏矩阵不同的压缩存储方法。

稀疏矩阵的压缩存储方法——顺序存储结构

1、三元组顺序表

```
#define MAXSIZE 12500
          //假设非零元个数的最大值
typedef   struct {
    int  i, j;   //该非零元的行列下标
    Elemtype  e;
}Triple;
typedef  struct {
    Triple  data[MAXSIZE + 1];
     int       mu, nu, tu;
    //矩阵的行、列数和非零元个数
}TSMatrix;
```

	i	*j*	*tu*
0	6	7	8
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

- 求转置矩阵

转置矩阵：一个 $m \times n$ 的矩阵 M，它的转置 T 是一个 $n \times m$ 的矩阵，且 $T(i, j) = M[j, i]$，$1 \le i \le n$，$1 \le j \le m$，即 M 的行是 T 的列，M 的列是 T 的行。

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

问题描述： 已知一个稀疏矩阵的三元组表，求该矩阵转置矩阵的三元组表。

M.data

	i	j	tu
0	6	7	8
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

解决思路：

① 将矩阵行、列维数互换

② 将每个三元组中的 *i* 和 *j* 相互调换

③ 重排三元组次序，使 *T.data* 中元素以 *T* 的行 (*M*的列) 为主序。

T.data

	i	j	tu
0	7	6	8
1	2	1	12
2	3	1	9
3	1	3	-3
4	6	3	14
5	3	4	24
6	2	5	18
7	1	6	15
8	4	6	-7

T.data

	i	j	tu
0	7	6	8
1	1	3	-3
2	1	6	15
3	2	1	12
4	2	5	18
5	3	1	9
6	3	4	24
7	4	6	-7
8	6	3	14

方法一：按 *M* 的列序转置

6	7	8
1	2	12
1	3	9
3	1	-3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	-7

M.data

```
for (col = 1;  col <= M.nu;  ++ col)
    for (p = 1; p <= M.tu;  ++ p)
      if ( M.data[p].j == col )
      { T.data[q].i = M.data[p].j ;
        T.data[q].j = M.data[p].i ;
        T.data[q].e = M.data[p].e;
        ++ q;
      }
```

7	6	8
1	3	-3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	-7
6	3	14

T.data

```
Status TransposeSMatrix(TSMatrix M, TSMatrix &T) {
  T.mu=M.nu; T.nu=M.mu; T.tu=M.tu;
  if (T.tu)
  { q = 1;
    for (col = 1;  col <= M.nu;  ++ col)
      for (p = 1; p <= M.tu;  ++ p)
        if ( M.data[p].j == col )
        { T.data[q].i = M.data[p].j ;
          T.data[q].j = M.data[p].i ;
          T.data[q].e = M.data[p].e;   ++ q;
        }
  }
  return OK;
} // TransposeSMatrix
```

6	7	8
1	2	12
1	3	9
3	1	-3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	-7

p →

q →

7	6	8
1	3	-3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	-7
6	3	14

时间复杂度：$O(nu \times tu)$

若 tu 与mu×nu 同数量级，

则为：$O(mu \times nu^2)$

一般矩阵转置算法:

```
for (col = 1;  col <= nu;  ++ col)
    for (row = 1;  row <= mu;   ++ row)
        T[col][row] = M[row][col];
```

一般矩阵转置算法时间复杂度：$O(mu \times nu)$

用三元组顺序表存储的矩阵转置算法时间复杂度：$O(nu \times tu)$

若 tu 与mu×nu 同数量级，则为：$O(mu \times nu^2)$

结论
- 用三元组顺序表存储稀疏矩阵节约存储空间（优点）；
- tu与mu×nu同数量级时，算法时间复杂度高（缺点）；
- 算法仅适用于 tu << mu×nu 的情况。

方法 2：按 *M* 的行序转置 —— 快速转置

实施步骤:

1、确定 *M* 的第 1 列的第 1 个非零元在 *T.data* 中的位置。　1

2、确定 *M* 的第 col -1 列的非零元个数。存入数组 num[*M.nu*]

3、确定 *M* 的第 col 列的第一个非零元在 *T.data* 中的位置。

存入数组 cpot[*M.nu*]

cpot[1] = 1;

cpot[col]=cpot[col–1]+num[col–1]

$2 \le col \le a.nu$

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

M.data

6	7	8
1	2	12
1	3	9
3	1	-3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	-7

T.data

7	6	8
1	3	-3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	-7
6	3	14

```
Status FastTransposeSMatrix( TSMatrix M, TSMatrix &T ) {
  // 采用三元组顺序表存储表示，求稀疏矩阵 M 的转置矩阵 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=1; t<=M.tu; ++t)   ++ num[M.data[t].j];
         // 求 M 中各列非零元的个数
      cpot[1] = 1;
      for (col=2; col<=M.nu; ++col)
         cpot[col] = cpot[col -1] + num[col -1];
           // 求 M 中各列的第一个非零元在 T.data 中的序号
```

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

6	7	8
1	2	12
1	3	9
3	1	-3
3	6	14
4	3	24
5	2	18
6	1	15
6	4	-7

7	6	8

```
for (p=1; p<=M.tu; ++p) {  // 转置矩阵元素
    col = M.data[p].j;    q = cpot[col];
    T.data[q].i
    T.data[q].j
    T.data[q].e
    ++ cpot[cc
  } // for
} // if
return OK;
} // FastTransposeSM
```

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

col	1	2	3	4	5	6	7
num(col)	2	2	2	1	0	1	0
cpot(col)	3	5	7	8	8	9	9

6	7	8	0
1	2	12	1
1	3	9	2
3	1	-3	3
3	6	14	4
4	3	24	5
5	2	18	6
6	1	15	7
6	4	-7	8

7	6	8	0
1	3	-3	1
1	6	15	2
2	1	12	3
2	5	18	4
3	1	9	5
3	4	24	6
4	6	-7	7
6	3	14	8

q

```
Status FastTransposeSMatrix( TSMatrix M, TSMatrix &T ) {
 // 采用三元组顺序表存储表示，求稀疏矩阵 M 的转置矩阵 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=1; t<=M.tu; ++t)   ++ num[M.data[t].j];
            // 求 M 中各列非零元的个数
        cpot[1] = 1;
        for (col=2; col<=M.nu; ++col)  cpot[col] = cpot[col -1] + num[col -1];
          // 求 M 中各列的第一个非零元在 T.data 中的序号
        for (p=1; p<=M.tu; ++p) {  // 转置矩阵元素
            col = M.data[p].j;    q = cpot[col];
            T.data[q].i = M.data[p].j;    T.data[q].j = M.data[p].i;
            T.data[q].e = M.data[p].e;    ++ cpot[col];
        } // for
    } // if
    return OK;
} // FastTransposeSMatrix
```

时间复杂度为：O(nu + tu)

与经典算法相同。

若 tu 与mu×nu 同数量级，则为：O(mu×nu)

三元组顺序表又称**有序的双下标法**。

三元组顺序表的**优点**：非零元在表中按行序有序存储，因此**便于进行依行顺序处理的矩阵运算**。

三元组顺序表的**缺点**：不能随机存取。若按行号存取某一行中的非零元，则需从头开始进行查找。

2、行逻辑联接的顺序表（带行表的三元组）

在稀疏矩阵中若**随机存取**任意一行的非零元

⇓

需知道每行首个非零元在三元组表中的位置

⇓

在存储三元组表的同时存储一个行表 rpos

（快速转置算法中“带行链接信息”的 cpot ）

⇓

行逻辑联接的顺序表

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

7	6	8
1	3	-3
1	6	15
2	1	12
2	5	18
3	1	9
3	4	24
4	6	-7
6	3	14

两个稀疏矩阵相乘时，可以看出这种表示方法的优越性。

```
#define MAXSIZE 12500
          //假设非零元个数的最大值
typedef   struct {
    int  i, j;    //该非零元的行列下标
    Elemtype  e;
}Triple;

typedef  struct {
    Triple  data[MAXSIZE + 1];
    int  rpos[MAXRC + 1]; // 指示各行第一个非零元的位置
    int  mu, nu, tu;
    //矩阵的行、列数和非零元个数
} RLSMatrix;
```

两个稀疏矩阵相乘时，可以看出这种表示方法的优越性。

矩阵乘法

设矩阵M是$m_1 \times n_1$矩阵，N是 $m_2 \times n_2$矩阵；只有$n_1 = m_2$时，才能相乘得到结果矩阵$Q=M \times N$ (一个$m_1 \times n_2$的矩阵)。

矩阵相乘的经典算法:

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

$$\underbrace{M[i][k] * N[k][j]}_{\text{不论是否为零，都要进行乘法运算。}}$$

没必要!

$$M = \begin{array}{c|c} i \rightarrow & \begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \\ & \uparrow k \end{array}$$

$$N = \begin{array}{cc} k \rightarrow & \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix} \\ & \uparrow j \end{array}$$

$$Q = M \times N = \begin{array}{cc} i \rightarrow & \begin{bmatrix} 0 & 6 \\ -1 & 0 \\ 0 & 4 \end{bmatrix} \\ & \uparrow j \end{array}$$

$$M=\begin{bmatrix}3 & 0 & 0 & 5\\ 0 & -1 & 0 & 0\\ 2 & 0 & 0 & 0\end{bmatrix} \quad N=\begin{bmatrix}0 & 2\\ 1 & 0\\ -2 & 4\\ 0 & 0\end{bmatrix} \quad Q=M\times N=\begin{bmatrix}0 & 6\\ -1 & 0\\ 0 & 4\end{bmatrix}$$

注意：两个稀疏矩阵相乘的结果不一定是稀疏矩阵。

$$\begin{bmatrix}1 & 0 & 0\\ 1 & 0 & 0\\ 1 & 0 & 0\end{bmatrix}\times\begin{bmatrix}1 & 1 & 1\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}=\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

i	*j*	*e*
1	2	6
2	1	-1
3	2	4

row	1	2	3	4
rpos[row]	1	2	3	5

0	4

```
int MulSMatrix (RLSMatrix M, RLSMatrix N, RLSMatrix  *Q)
{ if (M.nu != N.mu) return ERROR;
    Q.mu = M.mu;  Q.nu = N.nu;  Q.tu = 0;  // Q 初始化
    if (M.tu * N.tu != 0)   // Q 是非零矩阵
      for (arow=1;  arow<=M.mu;  ++arow)    //逐行处理 M
     {  ctemp[ ]=0 ;    // 当前行各元素的累加器清零
        Q.rpos[arow] = Q.tu + 1;
        if (arow<M.mu)  tp=M.rpos[arow+1];
        else  tp=M.tu+1;
        for (p=M.rpos[arow];  p<tp;  ++p)
       { //对当前行中每一个非零元找到对应元在 N 中的行号
           brow=M.data[p].j;
           if (brow < N.nu )   t = N.rpos[brow+1];
           else  t = N.tu+1;
```

```
            for (q=N.rpos[brow];  q< t;  ++q) {
              ccol = N.data[q].j;    // 乘积元素在Q中列号
              ctemp[ccol] += M.data[p].e * N.data[q].e;
            } // for q
          } // 求得Q中第crow( =arow)行的非零元
          for (ccol=1;  ccol<=Q.nu;  ++ccol) // 压缩存储该行非零元
            if (ctemp[ccol]) {
              if (++Q.tu > MAXSIZE) return ERROR;
              Q.data[Q.tu] = {arow, ccol, ctemp[ccol]};
            } // if
        } // for arow
      } // if
      return OK;
    } // MultSMatrix
```

上述算法的时间复杂度分析：

- 累加器ctemp初始化的时间复杂度为 $O(M.mu \times N.mu)$
- 求Q的所有非零元的时间复杂度为 $O(M.tu \times N.tu/N.mu)$
- 进行压缩存储的时间复杂度为 $O(M.mu \times N.nu)$

总的时间复杂度就是 $O(M.mu \times N.nu + M.tu \times N.tu/N.mu)$。

若M是m行n列的稀疏矩阵，N是n行p列的稀疏矩阵，则M中非零元的个数 $M.tu = \delta_M \times m \times n$，N中非零元的个数 $N.tu = \delta_N \times n \times p$，相乘算法的时间复杂度就是 $O(m \times p \times (1+n\delta_M\delta_N))$ ，当$\delta_M<0.05$ 和$\delta_N<0.05$ 及 $n <1000$时，相乘算法的时间复杂度就相当于 $O(m \times p)$。显然，这是一个相当理想的结果。如果事先能估算出所求乘积矩阵Q不再是稀疏矩阵，则以二维数组表示Q，相乘的算法也就更简单了。

3、稀疏矩阵的链式存储结构：十字链表

优点：它能够灵活地插入因运算而产生的新的非零元素，删除因运算而产生的新的零元素，实现矩阵的运算。

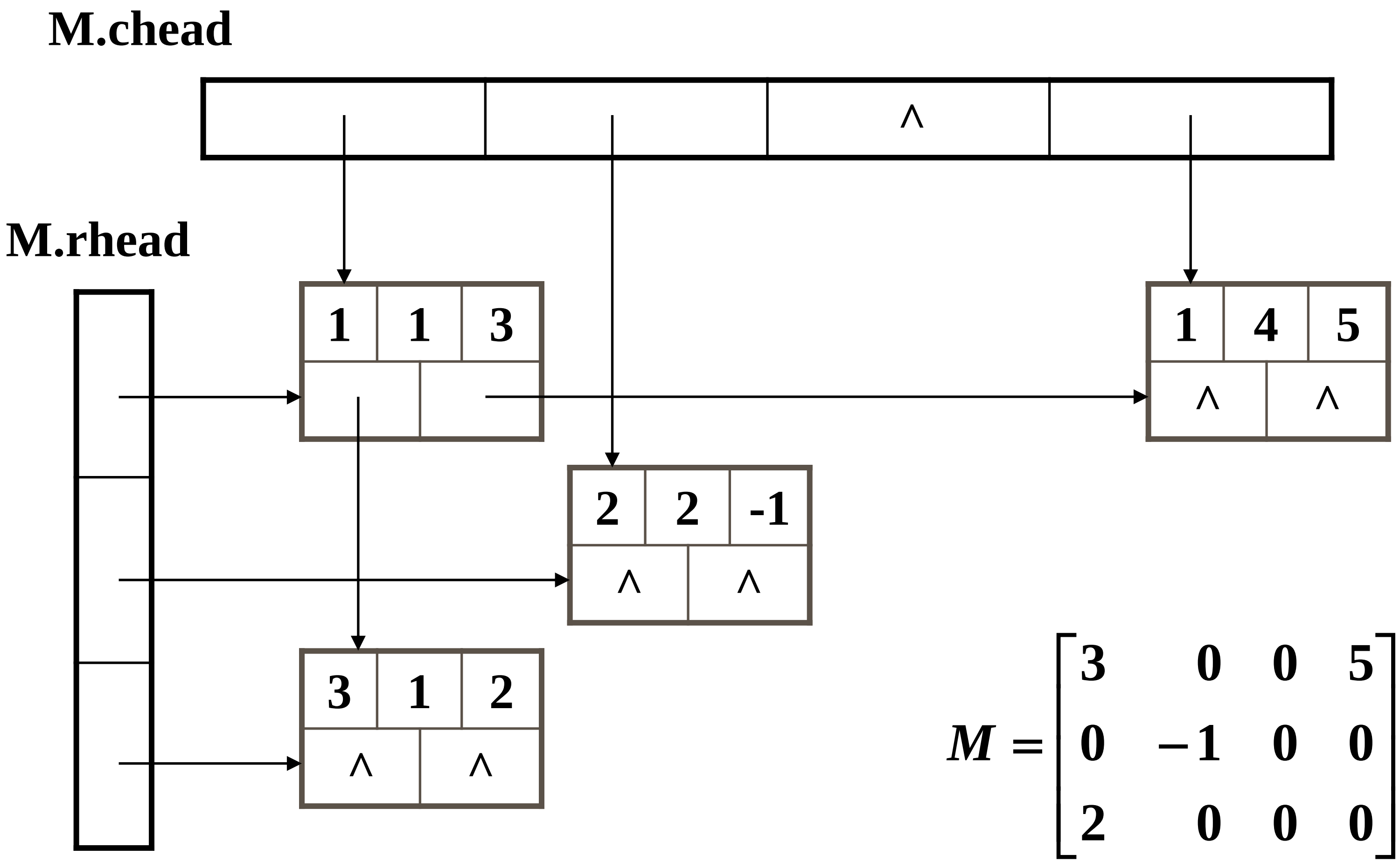
在十字链表中，矩阵的每一个非零元素用一个结点表示，该结点除了（row，col，value）外，还有两个域：

right： 用于链接同一行中的下一个非零元素；

down：用以链接同一列中的下一个非零元素。

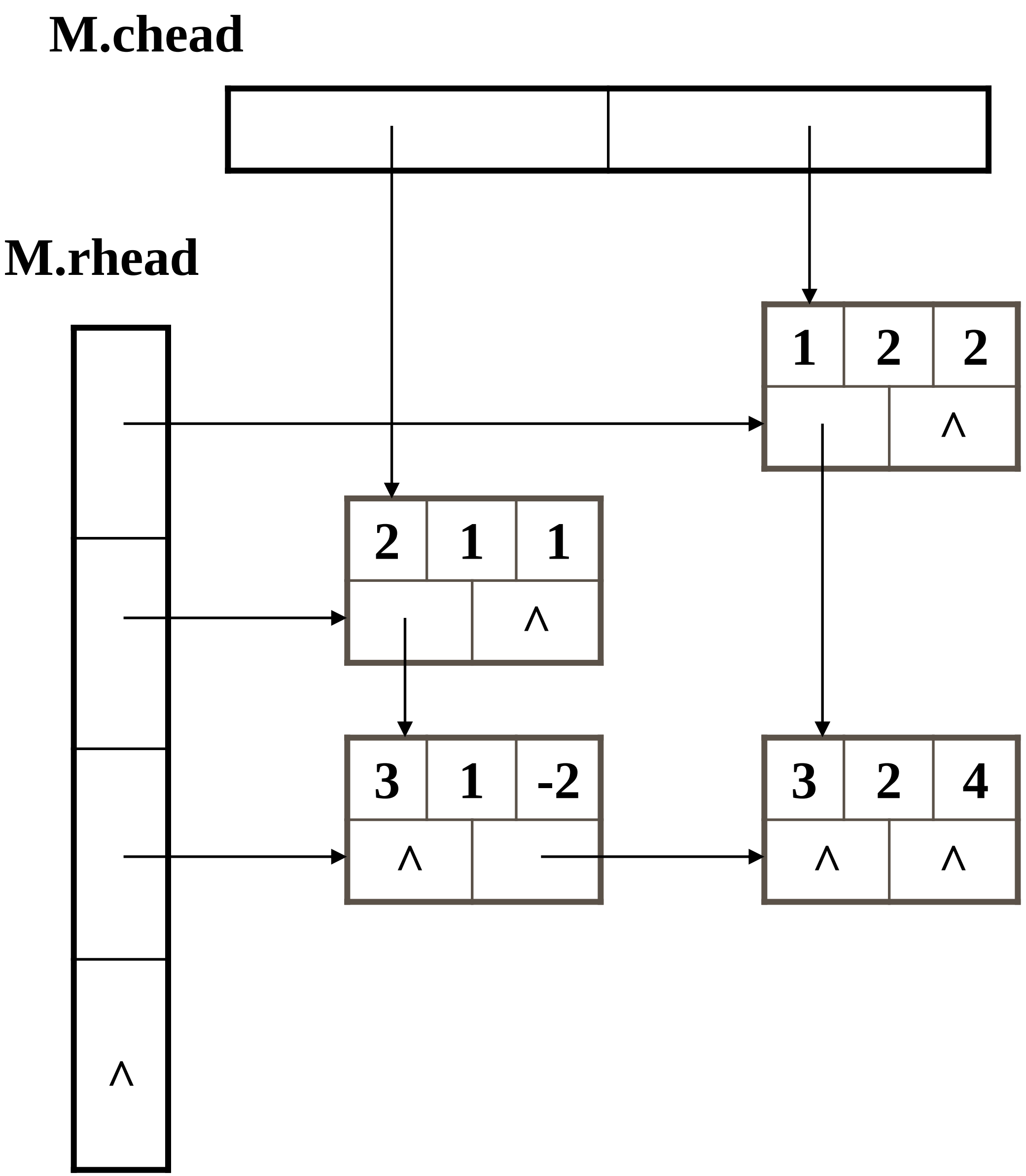
十字链表中结点的结构示意图：

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



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

$$N = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix}$$



十字链表的结构类型说明如下：

```
typedef struct OLNode
{ int                i,  j;      // 非零元素的行和列下标
   ElemType   e;
   struct OLNode  * right,  *down;
          // 非零元素所在行表列表的后继链域
}OLNode; *OLink;

typedef struct
{  OLink  * rhead,  *chead;  //行、列链表的头指针向量基址
    int  mu, nu, tu;   //稀疏矩阵的行数、列数、非零元个数
}CrossList;
```

建立稀疏矩阵的十字链表算法：

```
CreateCrossList (CrossList * M)
{//采用十字链表存储结构，创建稀疏矩阵M
if(M!=NULL) free(M);
scanf(&m,&n,&t);  //输入M的行数,列数和非零元素的个数
M->m=m;M->n=n;M->len=t;
If(!(M->row_head=(Olink*)malloc((m+1)sizeof(OLink)))) exit(OVERFLOW);
If(!(M->col_head=(OLink * )malloc((n+1)sizeof(OLink)))) exit(OVERFLOW);
M->row_head[ ]=M->col_head[ ]=NULL;
//初始化行、列头指针向量，各行、列链表为空的链表
for(scanf(&i,&j,&e);i!=0; scanf(&i,&j,&e))
{if(!(p=(OLNode *) malloc(sizeof(OLNode)))) exit(OVERFLOW);
p->row=i;p->col=j;p->value=e;  //生成结点
```

```
if(M->row_head[i]==NULL)   M->row_head[i]=p;
  else{  /*寻找行表中的插入位置*/
        for(q=M->row_head[i];  q->right&&q->right->col<j;  q=q->right)
            p->right=q->right; q->right=p;  /*完成插入*/
      }
if(M->col_head[j]==NULL)   M->col_head[j]=p;
  else{  /*寻找列表中的插入位置*/
        for(q=M->col_head[j];  q->down&&q->down->row<i;  q=q->down)
            p->down=q->down; q->down=p;   /*完成插入*/
      }
    }
  }
```

两个矩阵相加的算法描述:

(1) 初始令pa和pb分别指向A和B的第一行的第一个非零元素的结点，即

pa = A.rhead[1]; pb = B.rhead[1]; pre = NULL;

且令hl初始化 for (j=1; j<=A.nu; ++j) hl[j] = A.chead[j];

(2) 重复本步骤，依次处理本行结点，直到B的本行中无非零元素的结点，即pb==NULL为止：

① 若pa==NULL或pa->j〉pb->j（即A的这一行中非零元素已处理完），则需在A中插入一个pb所指结点的复制结点。假设新结点的地址为p，则A的行表中的指针作如下变化：

```
if pre == NULL rhead[p->i]=p;
else { pre->right = p; }
p->right = pa; pre = p;
```

A的列链表中的指针也要作相应的改变。首先需从hl[p->j]开始找到新结点在同一列中的前驱结点，并让hl[p->j]指向它，然后在列链表中插入新结点：

```
if chead[p->j] == NULL
{ chead[p->j] = p; p->down = NULL; }
else {
p->down = hl[p->j]->down; hl[p->j]->down = p; }
hl[p->j] = p;
```

② 若pa->j〈pb->j且pa->j!=0，则令pa指向本行下一个非零元结点，即 pre = pa; pa = pa->right;

③ 若pa->j == pb->j，则将B中当前结点的值加到A中当前结点上，即

pa->e + = pb->e;

此时若pa->e!=0，则指针不变，否则删除A中该结点，即行表中指针变为：

```
if pre == NULL rhead[pa->i] = pa->right;
else { pre->right = pa->right; }
p=pa; pa=pa->right;
```

同时，为了改变列表中的指针，需要先找到同一列中的前驱结点，且让hl[pa->j]指向该结点，然后如下修改相应指针：

```
if chead[p->j] == p
chead[p->j] = hl[p->j] = p->down;
else { hl[p->j]->down = p->down; }
free (p);
```

(3) 若本行不是最后一行，则令pa和pb指向下一行的第一个非零元结点，转(2)；否则结束。

此算法时间复杂度：O(ta+tb)

- 数组的定义
- 数组的顺序表示和实现
- 矩阵的压缩存储
- 广义表的概念、性质
- 广义表的存储结构、相关递归算法

5.4 广义表的定义

广义表（又称列表 Lists）是$n \geq 0$个元素 $a_1, a_2, \ldots, a_n$ 的有限序列，其中每一个a_i 或者是**原子**，或者是一个**子表**。

单个元素

例：中国举办的国际足球邀请赛，参赛 可表示如下：

（阿根廷，巴西，德国，法国，（ 班牙，

意大利，英国，（国家队，建业，实德））

在这个表中，韩国队应排在法国队后面，但由于其水平低未敢参加，成为空表。国家队、建业队、实德队均作为东道主的参赛队参加，构成一个小的线性表，成为原线性表的一个数据元素。这种**拓宽了的线性表就是广义表**。

广义表通常记作： $LS = (a_1, a_2, \ldots, a_n)$

其中： LS 为表名， n 为表的长度， 每一个 a_i 为表的元素。

习惯上，一般用大写字母表示广义表，小写字母表示原子。

表头：若 LS 非空 ($n \geq 1$)，则其第一个元素 a_1 就是表头。

记作 head(LS) = a_1。注：表头可是原子，也可是子表。

表尾：除表头之外的其它元素组成的表。

记作 tail(LS) = (a_2, ..., a_n)。

注：表尾不是最后一个元素，而是一个子表。

例：(1) $A=(\ )$ 空表，长度为 0。

(2) $B=((\ ))$ 长度为 1，表头、表尾均为 $(\ )$。

(3) $C=(a, (b, c))$ 长度为 2，由原子 a 和子表 (b, c) 构成。

表头为 a；表尾为 $((b, c))$。

(4) $D=(x, y, z)$ 长度为 3，每一项都是原子。

表头为 x；表尾为 (y, z)。

(5) $E=(C, D)$ 长度为 2，每一项都是子表。

表头为 C；表尾为 (D)。

(6) $F=(a, F)$ 长度为 2，第一项为原子第二项为它本身。

表头为 a；表尾为 (F)。

$F=(a, (a, (a, \ldots)))$

广义表的性质

(1) 广义表中的数据元素有相对**次序**；

(2) 广义表的**长度**定义为最外层所包含元素的个数；

如： $C=(a, (b, c))$ 是长度为 2 的广义表。

(3) 广义表的**深度**定义为该广义表**展开后**所含**括号的重数**；

$A = (b, c)$ 的深度为 1，$B = (A, d)$ 的深度为 2，

$C = (f, B, h)$ 的深度为 3。

注意： “原子”的深度为 **0**；“空表”的深度为 **1** 。

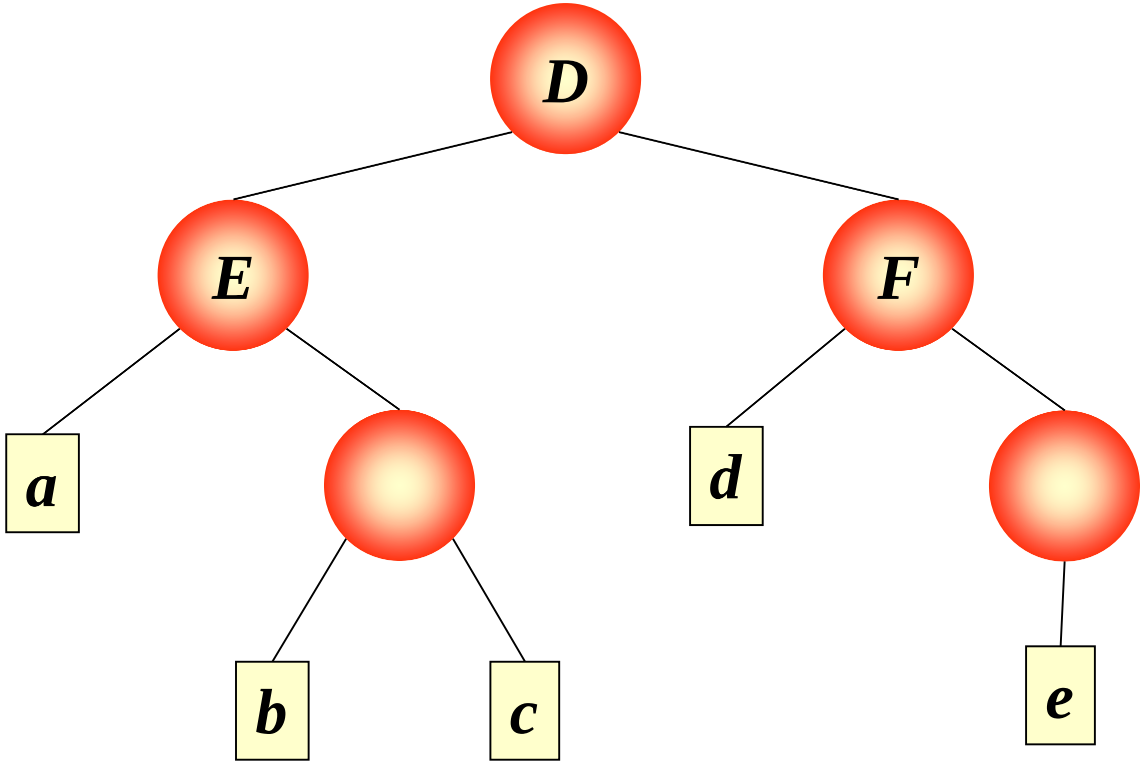
(4) 广义表可以为其他广义表**共享**；如：广义表 B 就共享表 A。在 B 中不必列出 A 的值，而是通过名称来引用。

(5) 广义表可以是**递归**的表。如：$F=(a, F)=(a, (a, (a, \ldots)))$

注意： **递归表的深度是无穷值，长度是有限值。**

(6) 广义表是**多层次**结构，广义表的元素可以是单元素，也可以是子表，而子表的元素还可以是子表，…。可以用图形象地表示。

例： $D=(E, F)$ 其中： $E=(a, (b, c))$ $F=(d, (e))$



广义表可看成是线性表的推广，线性表是广义表的特例。

广义表的结构相当灵活，在某种前提下，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。

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。

另外，树和有向图也可以用广义表来表示。

由于广义表不仅集中了线性表、数组、树和有向图等常见数据结构的特点，而且可有效地利用存储空间，因此在计算机的许多应用领域都有成功使用广义表的实例。

广义表基本运算

取表头运算 GetHead 和取表尾运算 GetTail

若广义表 $LS=(a_1, a_2, \ldots, a_n)$，

则 $\text{GetHead}(LS) = a_1$ $\text{GetTail}(LS) = (a_2, \ldots, a_n)$。

注意：取表头得到的结果可以是原子，也可以是一个子表。

取表尾得到的结果一定是一个子表。

例: $D = (E, F) = ((a, (b, c))，F)$

$\text{GetHead}(D) = E$ $\text{GetTail}(D) = (F)$

$\text{GetHead}(E) = a$ $\text{GetTail}(E) = ((b, c))$

$\text{GetHead}(((b, c))) = (b, c)$ $\text{GetTail}(((b, c))) = ( )$

$\text{GetHead}((b, c)) = b$ $\text{GetTail}((b, c)) = (c)$

$\text{GetHead}((c)) = c$ $\text{GetTail}((c)) = ( )$

- 数组的定义
- 数组的顺序表示和实现
- 矩阵的压缩存储
- 广义表的概念、性质
- 广义表的存储结构、相关递归算法

5.5 广义表的存储结构

由于广义表是递归定义的，其元素可具有不同的结构（原子或列表）

~~⇒~~ 用顺序存储结构表示

⇒ 采用链式存储结构（**广义链表**）

1、首尾链表

广义表（不空） $\xrightarrow{\text{分解}}$ 表头 + 表尾

表头 + 表尾 $\xrightarrow{\text{确定}}$ 广义表（不空）

首尾表示法就是根据这一性质设计的一种存储方法。

- 结点的结构形式

表结点由三个域组成：
- 标志域 *tag* = 1
- 指示表头的指针域 *hp*
- 指示表尾的指针域 *tp*

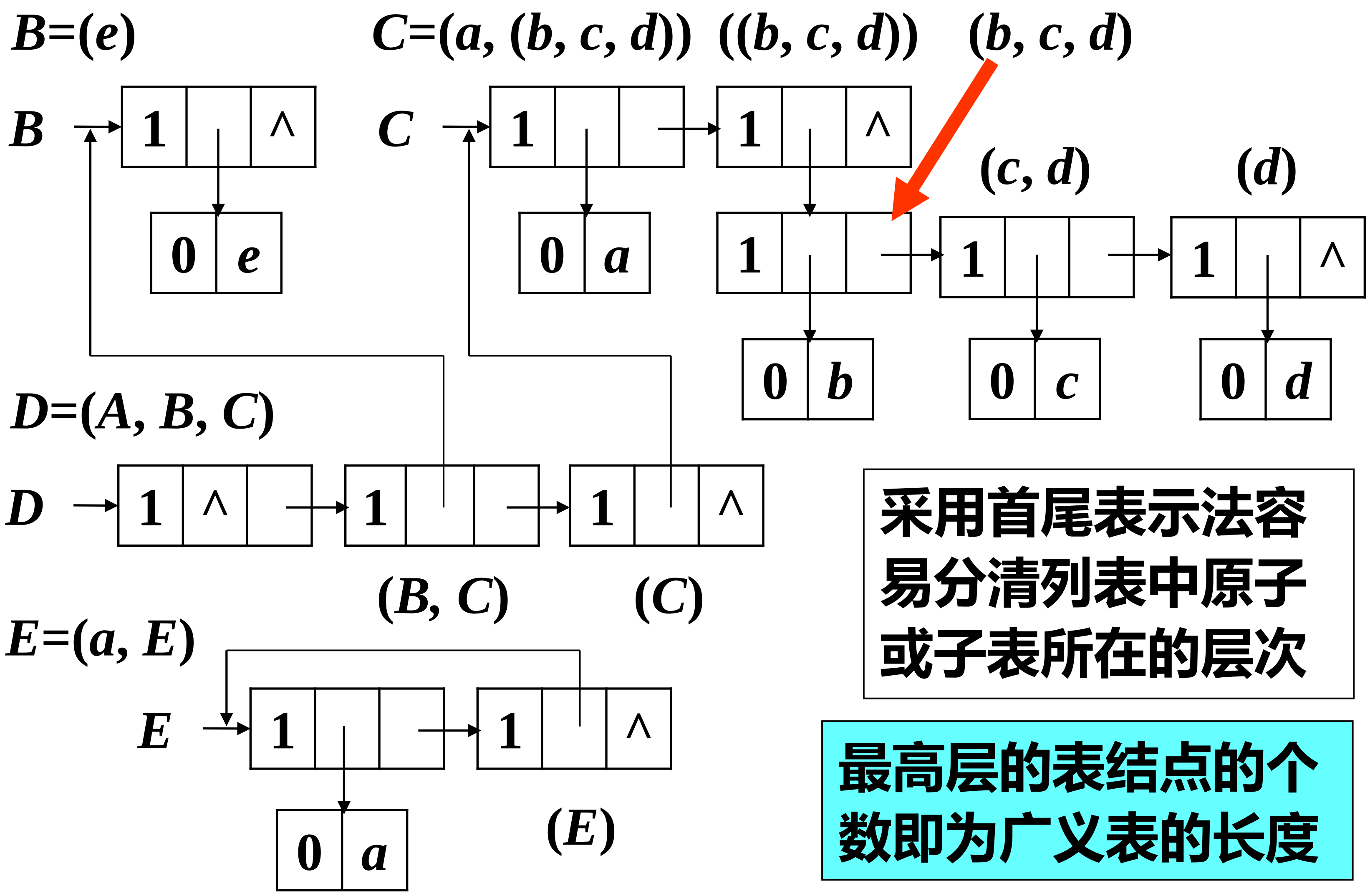
tag = 1	*hp*	*tp*

原子结点由两个域组成：
- 标志域 *tag* = 0
- 值域 *atom*

tag = 0	*atom*

```
typedef  enum {ATOM, LIST} ElemTag;
                    // ATOM=0：单元素；LIST=1：子表
typedef  struct  GLNode {
   Elemtag  tag;   // 标志域，用于区分元素结点和表结点
   union {           // 元素结点和表结点的联合部分
     Atomtype  atom;     // atom 是原子结点的值域
     struct
         {struct GLNode  *hp,  *tp;
         }ptr;  // ptr是表结点的指针域，ptr.hp 和 ptr.tp分别
                 // 指向表头和表尾
   };
 }*GList;                     // 广义表类型
```

- **结点的链接** $A=(\ )$ $A=\text{NULL}$



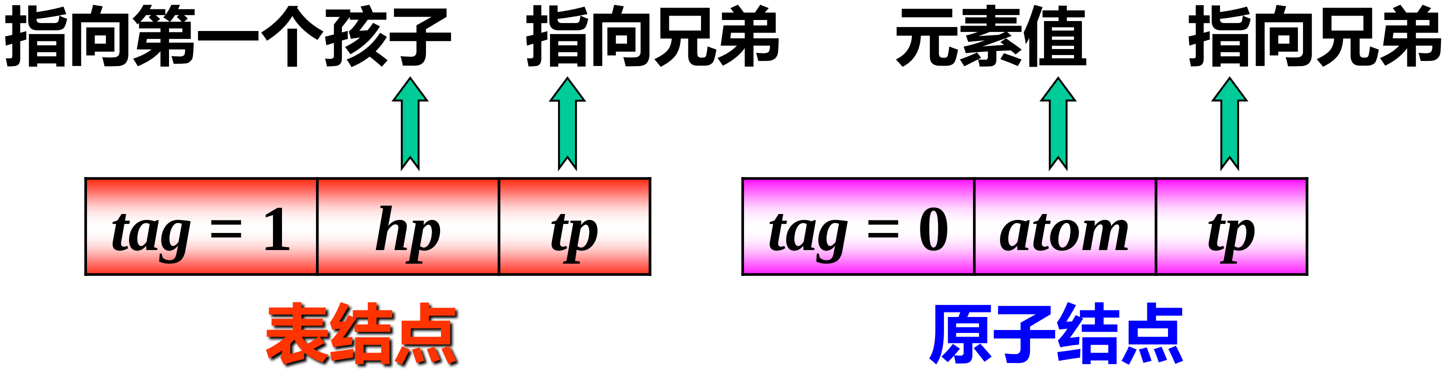
采用首尾表示法容易分清列表中原子或子表所在的层次

最高层的表结点的个数即为广义表的长度

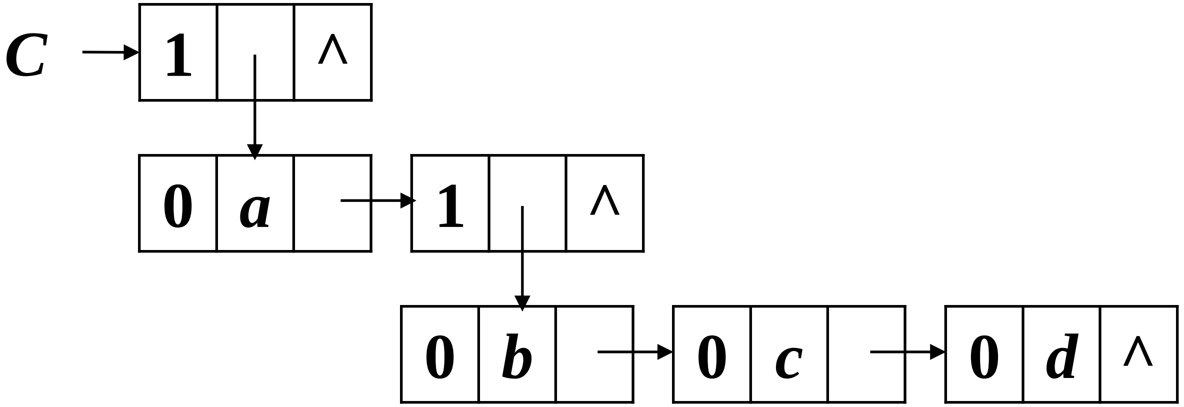
2、扩展线性链表（孩子兄弟链表）

- 结点的结构形式

两种结点形式：有孩子结点，用以表示列表；
无孩子结点，用以表示单元素。

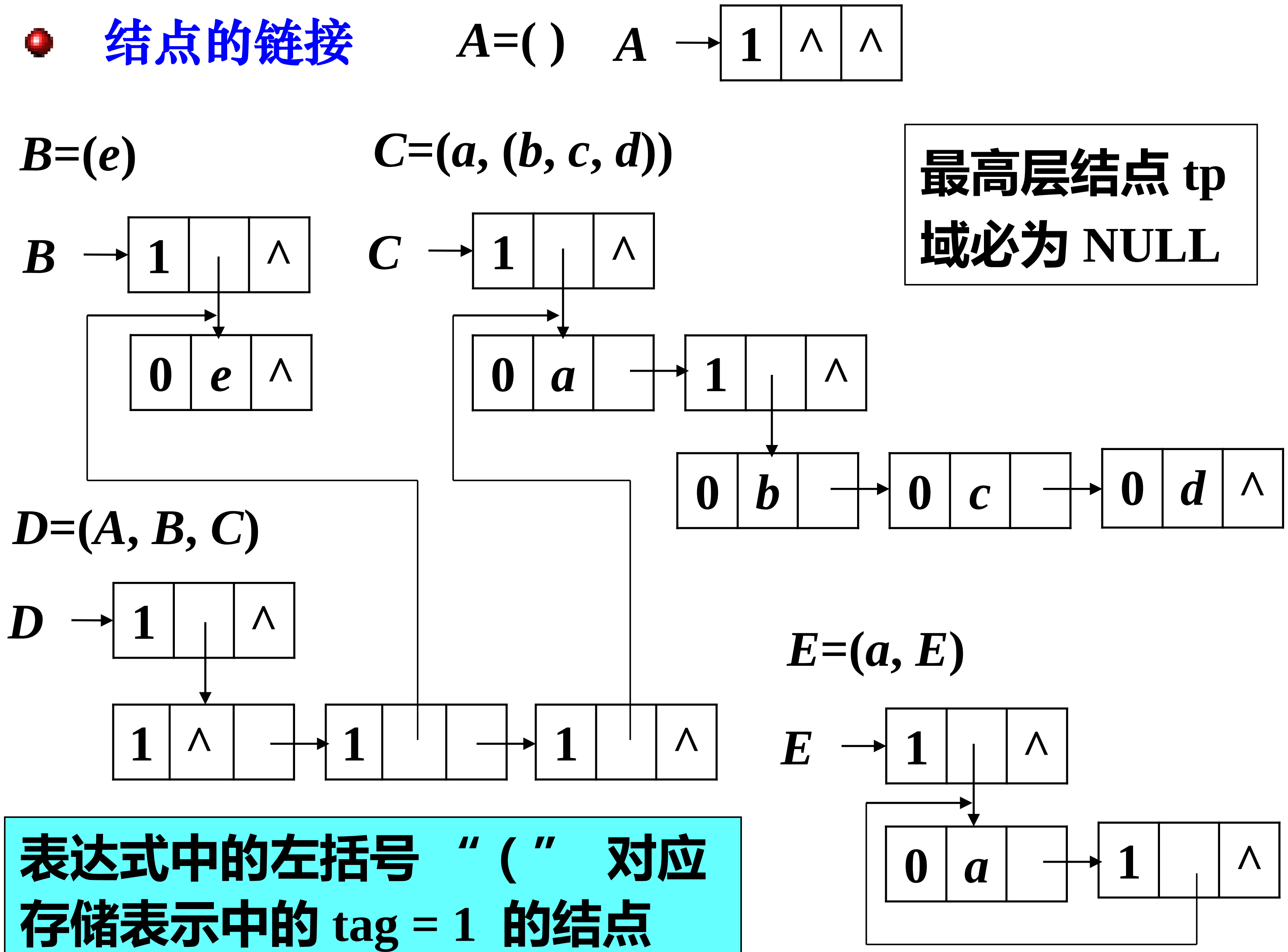


$C=(a, (b, c, d))$



```
typedef  enum {ATOM, LIST} Elemtag;
                    // ATOM=0：单元素；LIST=1：子表
typedef  struct  GLNode {
   Elemtag  tag;     // 标志域，用于区分元素结点和表结点
   union {                   // 元素结点和表结点的联合部分
        Atomtype  atom;     // atom 是原子结点的值域
        struct GLNode  *hp;        // 表结点的表头指针
   };
   struct GLNode  *tp;            // 指向下一个结点
}*GLists;                   // 广义表类型
```

- 结点的链接



最高层结点 tp 域必为 NULL

表达式中的左括号“(”对应存储表示中的 tag = 1 的结点

广义表的递归算法

- 求广义表的深度
- 复制广义表
- 创建广义表存储结构

- **求深度算法**

1、广义表的深度=Max {子表的深度} +1

2、可以直接求解的两种简单情况为:

空表的深度 = 1

原子的深度 = 0

```
int GlistDepth(Glist L) {
  // 返回指针L所指的广义表的深度
  if (!L) return 1;
  if (L->tag == ATOM) return 0;
  for (max=0, pp=L; pp; pp=pp->ptr.tp){
    dep = GlistDepth(pp->ptr.hp);
    if (dep > max) max = dep;
  }
  return max + 1;
} // GlistDepth
```

将广义表分解成表头和表尾两部分，分别(递归)复制求得新的表头和表尾，

可以直接求解的两种简单情况为:

空表复制求得的新表自然也是空表; 原子结点可以直接复制求得。

复制求广义表的算法描述如下:

若 ls= NIL 则 newls = NIL

否则

构造结点 newls,

由 表头ls->ptr.hp 复制得 newhp

由 表尾 ls->ptr.tp 复制得 newtp

并使 newls->ptr.hp = newhp,

newls->ptr.tp = newtp

```
Status CopyGList(Glist &T, Glist L) {
  if (!L) T = NULL;  // 复制空表
  else {
    if ( !(T = new GLNode) )
      exit(OVERFLOW);  // 建表结点
    T->tag = L->tag;
    if (L->tag == ATOM)
      T->atom = L->atom;  // 复制单原子结点
    else { 分别复制表头和表尾 }
  } // else
  return OK;
} // CopyGList
```

创建广义表存储结构

假设以字符串 $S = '(\alpha_1, \alpha_2, \cdots, \alpha_n)'$ 的形式定义广义表 L，建立相应的存储结构。

由于S中的每个子串α_i定义 L 的一个子表，从而产生 n 个子问题，即分别由这 n个子串 (递归)建立 n 个子表，再组合成一个广义表。

可以直接求解的两种简单情况为:

由串'( )'建立的广义表是空表;

由单字符建立的子表只是一个原子结点。

教学要求

I. 数组与线性表的关系及其ADT定义
II. 数组的顺序表示和实现
III. 两种特殊矩阵及其压缩存储(操作)
IV. 广义表的两个重要概念
V. 广义表ADT定义、表示和实现
VI. 广义表的递归算法