第五章

数组和广义表

※ 教学内容：

数组的存储结构；稀疏矩阵的表示及操作的实现；广义表的定义和存储结构；广义表的递归算法；

※ 教学重点：

数组在以行为主的存储结构中的地址计算方法；矩阵实现压缩存储时的下标变换；
理解稀疏矩阵的两种存储方式的特点和适用范围，领会以三元组表示稀疏矩阵时进行运算采用的处理方法；广义表的定义及其存储结构，学会广义表的表头，表尾分析方法；学习编制广义表的递归算法。

※ 教学难点：

矩阵实现压缩存储时的下标变换；广义表的存储结构。

5.1 数组的类型定义

5.2 数组的顺序表示和实现

5.3 矩阵的压缩存储

5.4 广义表的类型定义

5.5 广义表的表示方法

5.1 数组的类型定义

一、数组的定义

- 数组是有限个数据元素的集合;
- 数组的所有数组元素具有相同特性;
- 每个数组元素名由数组名和下标组成;
- 每组有定义的下标值都有一个与该下标对应的数组元素值.

一维数组A[n]：简单的线性表$(a_1, a_2, \ldots, a_n)$;

二维数组A[m,n]：看成由m个行向量组成的线性表，
或由n个列向量组成的线性表;

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & \ldots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \ldots & a_{1,n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \ldots & a_{m-1,n-1} \end{pmatrix}$$

N维数组:看成其数据元素为n-1维数组类型的线性表。

二、抽象数据类型数组的定义

ADT Array {

数据对象:

$D = \{a_{j_1,j_2,\ldots,j_i,j_n} \mid j_i = 0,\ldots,b_i - 1,\ i=1,2,..,n \}$

数据关系:

$R = \{R1, R2, \ldots, Rn\}$

$Ri = \{<a_{j_1,\ldots j_i,\ldots j_n}, a_{j_1,\ldots j_i+1,\ldots j_n}> \mid 0 \le j_k \le b_k - 1,$

$1 \le k \le n$ 且$k \ne i,\ 0 \le j_i \le b_i - 2,\ i=2,\ldots,n \}$

基本操作:

} ADT Array

二维数组的定义：

数据对象：

$$D = \{a_{ij} \mid 0 \le i \le b_1-1,\ 0 \le j \le b_2-1\}$$ //b1行数，b2列数

数据关系：

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

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

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

数组一旦被定义，其维数和维界就不再改变，因此，除了初始化和销毁之外，数组只有两种运算：

(1) 给定一组下标，存取相应的数据元素；

(2) 给定一组下标，修改相应数据元素中的某个数据项的值。

数组不能进行元素的插入和删除运算。

基本操作:

InitArray(&A, n, bound1, ..., boundn)

DestroyArray(&A)

Value(A, &e, index1, ..., indexn)

Assign(&A, e, index1, ..., indexn)

InitArray(**&**A, n, bound1, ..., boundn)

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, index1, ..., indexn)

初始条件： A是n维数组，e为元素变量，随后是n 个下标值。

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, index1, ..., indexn)

初始条件： A是n维数组，e为元素变量，随后是n 个下标值。

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。

5.2 数组的顺序表示和实现

类型特点：

1) 只有引用型操作，没有加工型（插入和删除）操作；建立数组后，元素个数与元素之间的关系不会发生变动。可以用顺序存储表示数组。

2) 数组是多维的结构，而存储空间是一个一维的结构。用一组连续存储单元存放数组的元素就有个次序约定问题。

有两种顺序映象的方式：

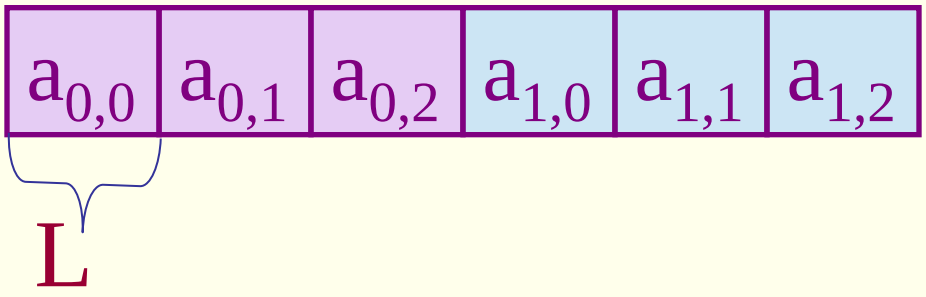
1) 以行序为主序； (C语言采用行序为主序的存储结构)

2) 以列序为主序。（高下标优先）

以“行序为主序”的存储映象

例如：

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$



二维数组$A[b_1,b_2]$中任一元素$a_{i,j}$的存储位置

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

$LOC(0,0)$称为**基地址**或基址。

推广到一般情况，可得到 n 维数组数据元素存储位置的映象关系：

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

可得下列公式：

$$LOC(j_1, j_2, ..., j_n) = LOC(0,0,...,0) + \sum_n c_i j_i$$

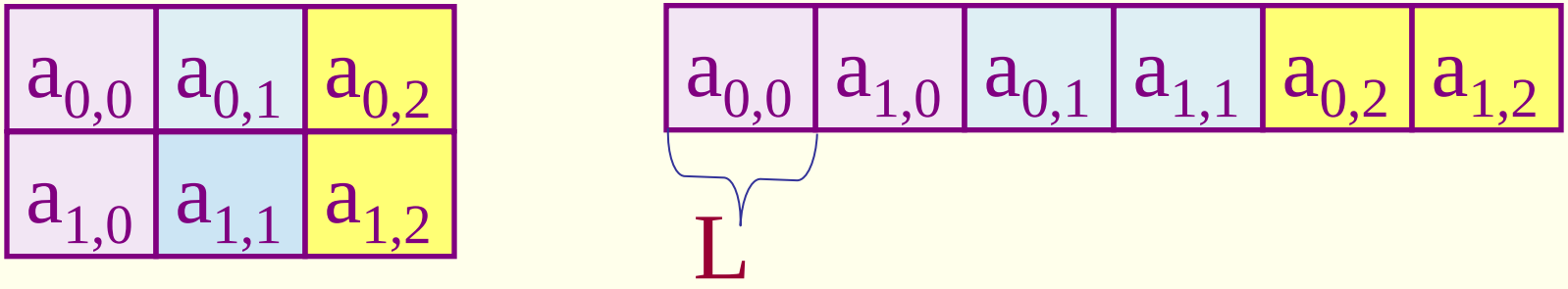
其中 $c_n = L$，$c_{i-1} = b_i \times c_i$，$1 < i \leq n$。$_{i=1}$

称为 n 维数组的映象函数。**数组元素的存储位置是其下标的线性函数。**

next

以“列序为主序”的存储映象

例如：



二维数组$A[b_1,b_2]$中任一元素$a_{i,j}$的存储位置

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

称为基地址或基址。

5.3 矩阵的压缩存储

矩阵是数学对象，如何存储矩阵的元素，使得矩阵的各种运算有效地进行。当阶数很高的矩阵中有许多值相同的元素或零元素，为了节省空间，对矩阵进行压缩存储。实际是将二维数组的数据元素压缩到一维数组上。

- 压缩的含义
  - 为多个值相同的元素只分配一个存贮空间；
  - 零元素不分配存贮空间。
- 特殊矩阵：值相同元素或零元素分布有一定规律的矩阵。
- 稀疏矩阵：值相同元素或零元素分布没有规律的矩阵。

一、特殊矩阵的压缩存储

特殊矩阵：非零元在矩阵中的分布有一定规则

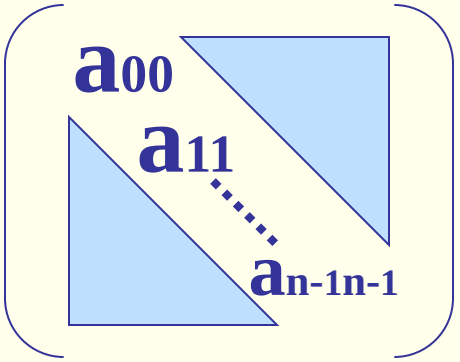
例如：三角矩阵，对角矩阵

1、对称矩阵

若n阶矩阵A满足

$$a_{ij}= a_{ji} \quad 0\le i,j\le n-1$$

则称为n阶对称矩阵。



对于对称矩阵，我们可以为每一对对称元分配一个存储空间，则可将n^2个元压缩存储到$n*(n+1)/2$个元的空间中。不失一般性，我们可以以行序为主序存储其下三角（包括对角线）中的元。

对称矩阵的压缩存储:

a_{00}	a_{10}	a_{11}	a_{20}	…	$a_{n-1,0}$	…	$a_{n-1,n-1}$

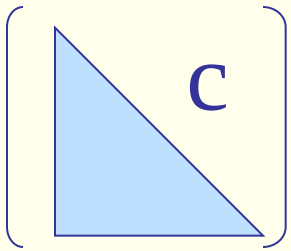
存贮地址计算公式:

$$Loc(a_{ij}) = Loc(a_{00}) + [i*(i+1)/2+j]*L$$

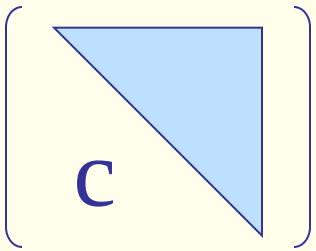
2、三角矩阵

所谓下（上）三角矩阵是指矩阵的上（下）三角（不包括对角线）中的元均为常数c或零的n阶矩阵。

对于三角矩阵，则除了和对称矩阵一样，只存储其下（上）三角中的元之外，再加一个存储常数c的存储空间即可。



下三角矩阵

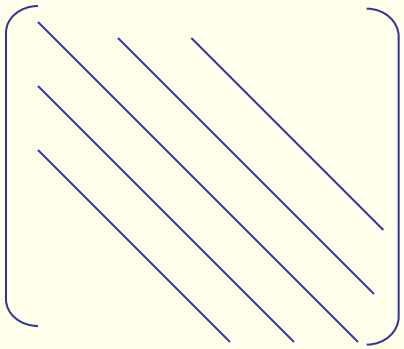


上三角矩阵

3、对角矩阵

所有的非零元都集中在以主对角线为中心的带状区域中。即除了主对角线上和直接在对角线上、下方若干条对角线上的元之外，所有其它的元皆为零。

对于对角矩阵，也可按某个原则（或以行为主，或以对角线的顺序）将其压缩存储到一维数组上。并找出每个非零元在一维数组中的对应关系。



三对角阵

a11 a12

a21 a22 a23

a32 a33 a34

…… aii……

an-1,n-2 an-1,n-1 an-1,n

an,n-1 an,n

二、稀疏矩阵的压缩存储

何谓稀疏矩阵？

假设 m 行 n 列的矩阵含 t 个非零元素，则称

$$\delta = \frac{t}{m \times n}$$

为稀疏因子。

通常认为 $\delta \leq 0.05$ 的矩阵为稀疏矩阵。

- 如果一个矩阵中非零元素的个数远远小于矩阵元素的总数，且非零元素的分布无一定规律，则称之为稀疏矩阵。

例：

$$M=\begin{pmatrix} 0 & 0 & 0 & -3 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 24 & 0 \\ 0 & 0 & -7 & 0 & 0 & 9 \\ 18 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵的基本运算如下：

CreateSMatrix(&M)

操作结果：创建稀疏矩阵M

DestroySMatrix(&M)

初始条件：稀疏矩阵M存在

操作结果：销毁稀疏矩阵M

PrintSMatrix(M)

初始条件：稀疏矩阵M存在

操作结果：输出稀疏矩阵M

CopySMatrix(M,&T)

初始条件：稀疏矩阵M存在

操作结果：由稀疏矩阵M复制得到T

AddSMatrix(M,N,&Q)

初始条件：稀疏矩阵M和N的行数与列数对应相等

操作结果：求稀疏矩阵的和Q=M+N

SubSMatrix(M,N,&Q)

初始条件：稀疏矩阵M和N的行数与列数对应相等

操作结果：求稀疏矩阵的差Q=M-N

MultSMatrix(M,N,&Q)

初始条件：稀疏矩阵M的列数等于N的行数

操作结果：求稀疏矩阵的乘积Q=M*N

TransSMatrix(M,&T)

初始条件：稀疏矩阵M存在

操作结果：求稀疏矩阵M的转置矩阵T

稀疏矩阵的存储

以常规方法，即以二维数组表示高阶的稀疏矩阵时产生的**问题**：

1) 零值元素占了很大空间；

2) 计算中进行了很多和零值的运算，遇除法，还需判别除数是否为零。

解决问题的原则:

1) 尽可能少存或不存零值元素;

2) 尽可能减少没有实际意义的运算;

3) 操作方便。 即:

能尽可能快地找到与下标值(i，j)对应的元素;

能尽可能快地找到同一行或同一列的非零值元。

稀疏矩阵的存储方法

- 顺序存储
  - 三元组表示法
  - 行逻辑联接的顺序表
  - 带辅助行向量的二元组表示法
  - 伪地址表示法
- 链接存储
  - 带行指针向量的单链表示法
  - 行列表示法（十字链表）
  - 多链表示法（正交表）
- 散列存储

顺序存储

1、三元组表示法

用一个线性表来表示稀疏矩阵，线性表的每个结点对应稀疏矩阵的一个非零元素。其中包括三个域，分别为该元素的行下标、列下标和值。结点间的先后顺序按矩阵的行优先顺序排列（跳过零元素），将线性表用顺序的方法存储在连续的存储区里。

三元组表示法

$$M=\begin{pmatrix} 3 & 0 & 0 & 0 & 7 \\ 0 & 0 & -1 & 0 & 0 \\ -1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

i	j	v
1	1	3
1	5	7
2	3	-1
3	1	-1
3	2	-2
5	4	2

若稀疏矩阵共有N个非零元，并且行下标、列下标与值一样都占一个存储单元，则三元组法需要**3N**个存储单元。

稀疏矩阵的三元组表示的 C语言描述：

```
#define MAXSIZE 100  // 假设非零元个数的最大值为100

typedef struct
    { int    i,j;              // 该非零元的行下标和列下标
      elemtype v;               // 该非零元的值
    } Triple;                   // 三元组类型

typedef union
    { Triple data[MAXSIZE+1];   // 非零元三元组表，
                                //          data[0]未用
      int mu,nu,tu;             // 矩阵的行数、列数和非零元个数
    } TSMatrix;                 // 稀疏矩阵类型
```

稀疏矩阵的转置运算的实现

$$M_{m\times n} \xrightarrow{\text{转置}} N_{n\times m}$$

$$M=\begin{pmatrix} 3 & 0 & 0 & 0 & 7 \\ 0 & 0 & -1 & 0 & 0 \\ -1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix} \Rightarrow N=\begin{pmatrix} 3 & 0 & -1 & 0 \\ 0 & 0 & -2 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 7 & 0 & 0 & 0 \end{pmatrix}$$

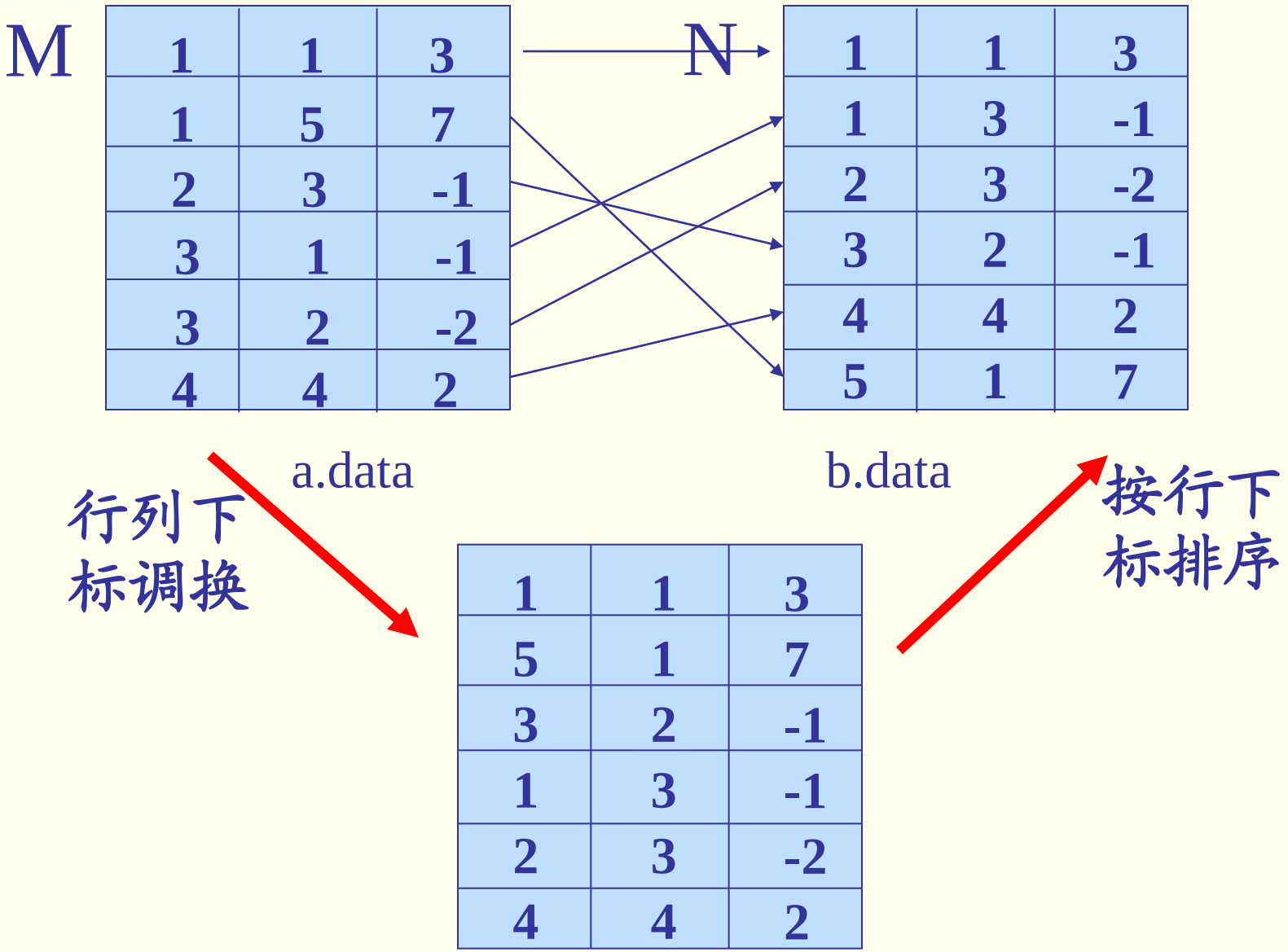
如何求矩阵M的转置矩阵N?

用常规的二维数组表示时的算法

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

其时间复杂度为: **O(mu × nu)**

已知某稀疏矩阵的三元组，求该矩阵转置后的三元组



如何实现转置呢?

（1）将矩阵的行列值相互交换

（2）将三元组中的i和j相互调换

（3）重排三元组之间的次序（按行序为主序）便可实现转置

前两步容易实现，如何实现第三步？有两种方法:

1 按照B.data中三元组的次序依次在A.data中找到相应的三元组进行转置。（即按三元组A的第二个字段值（列下标）由小到大的顺序进行转置----按矩阵M的列序来进行转置）

2 按照A.data中三元组的次序进行转置，并将转置后的三元组置入B中恰当的位置。

方法1:

按照矩阵M的列序进行转置，即按三元组A的第二个字段值（列下标）由小到大的顺序进行转置。为了找到M中每一列中所有的非零元素，需要对其三元组表A.data从第一行起整个扫描一遍，由于A.data是以M的行序为主序来存放每个非零元素的，对于M中具有相同列下标的非零元来讲，先扫描到的非零元的行下标一定小于后扫描到的非零元的行下标，由此得到的恰是B.data应有的顺序。

```
Transmat(TSMatrix M, TSMatrix &N) //方法1
{  N.mu=M.nu; N.nu=M.mu; N.tu=M.tu; //初始化
   if (M.tu!=0)
      { q=1;
        for (col=1;col<=M.nu; col++)    //按M的列转
        for (p=1;p<=M.tu; p++)   // 对M三元组表扫描一遍
          if (M.data[p].j= =col)
            { N.data[q].i=M.data[p].j;
              N.data[q].j=M.data[p].i;
              N.data[q].v=M.data[p].v;
              q++; }                          //q为N三元组表中的行号
      }                                  时间复杂度 O(tu × nu)
}
```

方法2：快速转置

按照A.data中三元组的次序进行转置，并将转置后的三元组置入B中恰当的位置。

如果能预先确定矩阵M中每一列（即N中每一行）的第一个非零元在B.data中应有的位置，那么在对A.data中的三元组依次作转置时，便可直接放到B.data中适当的位置上去。为了确定这些位置，在转置前，应先求得M的每一列中非零元的个数，进而求得每一列的第一个非零元在B.data中的适当位置。

在此，需附设num和cpot两个向量。

num[col]:表示矩阵M中第col列中非零元的个数;

cpot[col]:指示M中第col列的第一个非零元在B.data中的恰当位置。

显然有

cpot[1]= 1

cpot[col]= cpot[col-1]+num[col-1]　　2≤col ≤a.nu

矩阵M的向量num和cpot的值如下表:

A

1	1	3
1	5	7
2	3	-1
3	1	-1
3	2	-2
4	4	2

B

1	1	3
1	3	-1
2	3	-2
3	2	-1
4	4	2
5	1	7

col	1	2	3	4	5
num[col]	2	1	1	1	1
cpot[col]	1	3	4	5	6

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

```
FastTransmat(TSMatrix M, TSMatrix &N) //方法2
{  N.mu=M.nu; N.nu=M.mu; N.tu=M.tu;
   if (M.tu!=0)
      { for (col=1;col<=M.nu; col++)
           num[col]=0;        //初始化num向量
        for (t=1;t<=M.tu; t++)
           num[M.data[t].j]++; //求M中每一列含非零元个数
        cpot[1]=1;
        for (col=2;col<=M.nu;col++)
           cpot[col]=cpot[col-1]+num[col-1]
       //求M中每一列的第一个非零元在B.data中的序号（位置）
      for (p=1;p<=M.tu; p++)
          { col=M.data[p].j; q=cpot[col];
            N.data[q].i=M.data[p].j;   //p为M中的 行号
            N.data[q].j=M.data[p].i;   //q为转置后在N中的 行号
            N.data[q].v=M.data[p].v;
           cpot[col]++; }
      }
}
```

分析算法FastTransSMat的时间复杂度:

```
for (col=1; col<=M.nu; ++col) … …
for (t=1; t<=M.tu; ++t) … …
for (col=2; col<=M.nu; ++col) … …
for (p=1; p<=M.tu; ++p) … …
```

时间复杂度为： O(M.nu+M.tu)

2、行逻辑联接的顺序表

三元组顺序表又称有序的双下标法，它的特点是，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。然而，若需随机存取某一行中的非零元，则需从头开始查找三元组。

修改前述的稀疏矩阵的结构定义，增加一个数据成员**rpos**（每行第一个非零元在三元组中的位置），其值在稀疏矩阵的初始化函数中确定。即把**cpot**向量固定在稀疏矩阵的存储结构中。

```
#define  MAXSIZE  500
typedef struct {
   Triple data[MAXSIZE + 1];          //非零元三元组表
   int rpos[MU + 1]; //各行第一个非零元的位置表
   int  mu, nu, tu;   // 矩阵的行数、列数和非零元的个数
} RLSMatrix;          // 行逻辑链接顺序表类型
```

例1：给定一组下标(r,c)，求矩阵的元素值

```
ElemType value(RLSMatrix M, int r, int c) {
    p = M.rpos[r];
        //直接指向该行第一个非零元素在三元组中的位置
    while (M.data[p].i==r &&M.data[p].j < c)
            p++;
    if (M.data[p].i==r && M.data[p].j==c)
         return M.data[p].v;
       else return 0;
} // value
```

例2: 矩阵M[m1,n1]和N[m2,n2]乘法

矩阵乘法的精典算法:

```
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] += M[i][k] * N[k][j];
  }
```

其时间复杂度为: **O(m1 × n2 × n1)**

两个稀疏矩阵相乘（Q=M×N）的过程可大致描述如下：

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

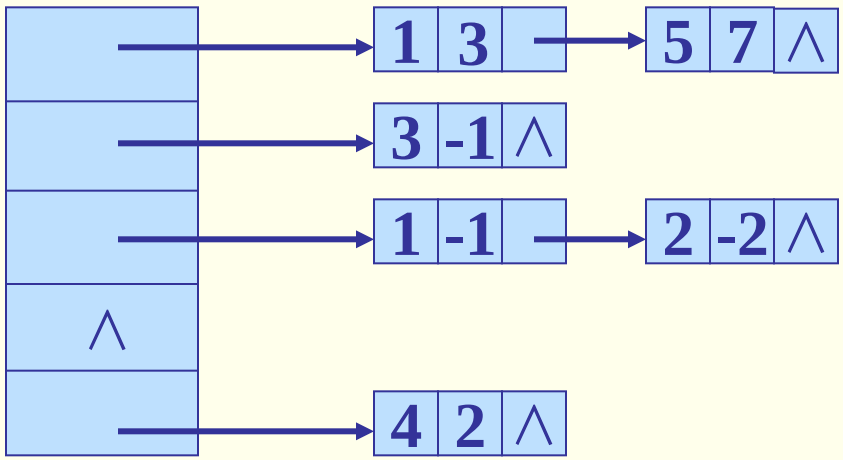
```
Status MultSMatrix
    (RLSMatrix M, RLSMatrix N, RLSMatrix &Q) {
 if (M.nu != N.mu) return ERROR;
 Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0; // 初始化
 if (M.tu*N.tu != 0) {   // Q是非零矩阵
   for (arow=1; arow<=M.mu; ++arow) {
     for (I=1; I<=N.nu;++I) ctemp[I]=0;
                              // 当前行各元素累加器清零
   处理M的每一行
   } // for arow
 } // if
 return OK;
} // MultSMatrix
```

处理M的每一行

```
Q.rpos[arow] = Q.tu+1;
 for (p=M.rpos[arow]; p<M.rpos[arow+1];++p) {
                                        //对当前行中每一个非零元
   brow=M.data[p].j;          //找到对应元在N中的行号
   if (brow < N.mu )  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<=N.nu; ++ccol) if (ctemp[ccol]) {
      if (++Q.tu > MAXSIZE) return ERROR;
      Q.data[Q.tu] = {arow, ccol, ctemp[ccol]};
    } // if
```

1、带行指针向量的单链表表示法

将矩阵的每一行的非零元素链接成一个单链表，每个结点包括三个域：列下标、非零元素值、指针。附设一个行指针向量作为m个单链表的表头指针向量。



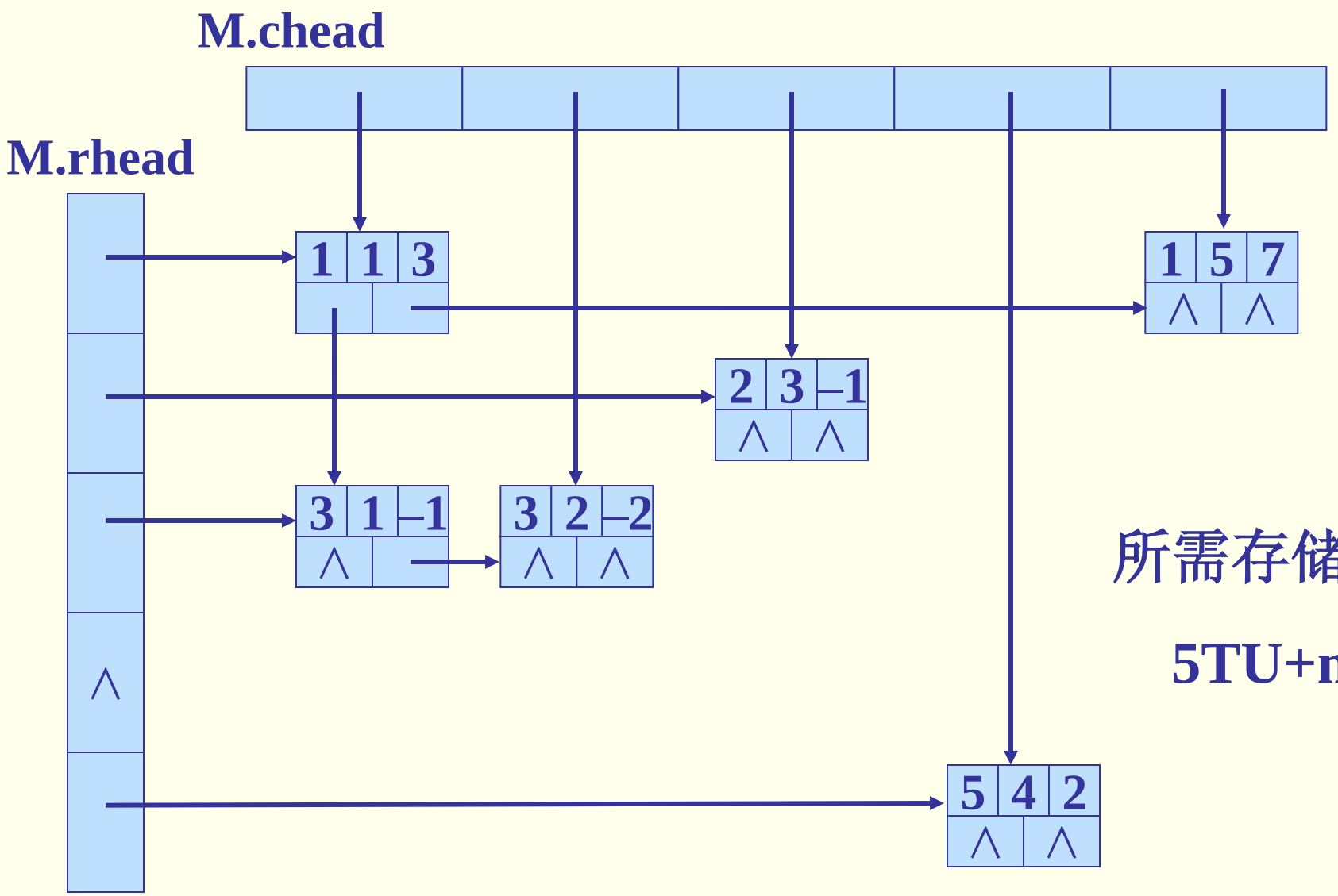
所需存储量

3TU+m

用类似方法也可组织成带列指针向量的单链表示法

2、行-列表示法（十字链表）

将带行指针向量的单链表表示法和带列指针向量的单链表表示法结合在一起来存储稀疏矩阵。每个结点包含五个字段：行下标、列下标、值、行指针、列指针。


M.chead
M.rhead
1 1 3
1 5 7
2 3 –1
3 1 –1
3 2 –2
5 4 2
所需存储量
5TU+m+n

5.4 广义表的类型定义

一、广义表的定义

广义表：是线性表的推广，也称为列表，是n个单元素或子表所组成的有限序列。记作

$LS=(\alpha_1,\alpha_2,\ldots,\alpha_n)$

其中：LS是广义表的名称，n是其长度；

α_i 可以是单个元素，也可以是广义表，分别称为广义表LS的原子和子表。

用大写字母表示广义表的名称，小写字母表示原子；

当广义表LS非空时，称第一个元素α_1为LS的表头，称其余元素组成的表$(\alpha_2, \alpha_3, \ldots,\alpha_n)$是LS的表尾。

广义表是递归定义的线性结构，

$LS = (\alpha_1, \alpha_2, \cdots, \alpha_n)$　　其中：α_i 或为原子 或为广义表

线性表中α_i 只限于单个元素，在广义表中α_i 可以是单个元素，也可以是广义表

例如：

A=( ) ----　　A是一个空表，它的长度为零

B=(e) ----　　列表B只有一个原子e，B的长度为1

C=(a,(b,c,d)) -----　　列表C的长度为2，两个元素分别为原子a和子表（b,c,d）

D=(A,B,C) ----　　列表D的长度为3，三个元素都是列表代入子表值，D=(( ),(e),(a,(b,c,d)))

E=(a,E) ----　　递归表，长度为2。E相当于一个无限的列表E=(a,(a,(a,…)))

广义表是一个多层次的线性结构

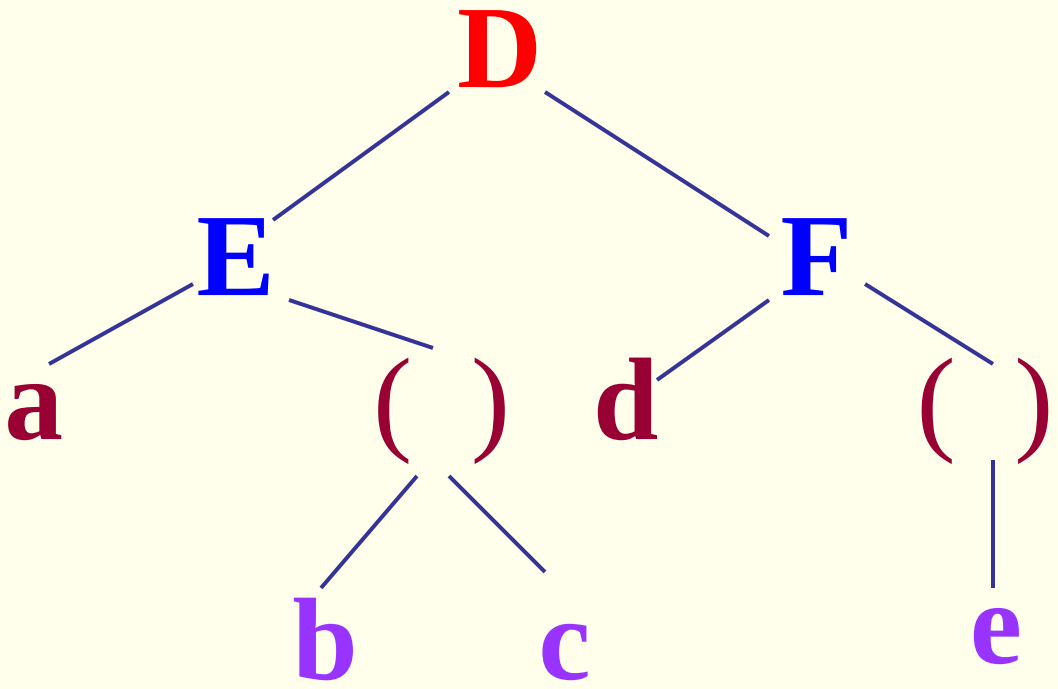
例如:

D=(E, F)

其中:

E=(a, (b, c))

F=(d, (e))



广义表 LS = ($\alpha_1, \alpha_2, \ldots, \alpha_n$)的结构特点：

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包含元素个数；

3) 广义表的深度定义为所含括弧的重数；

注意：“原子”的深度为 0

“空表”的深度为 1

4) 广义表可以共享；用广义表的名称来引用，不必列出广义表的值。

5) 广义表可以是一个递归的表。

递归表的深度是无穷值，长度是有限值。

6) 任何一个非空广义表 LS = (α1, α2, …, αn)
均可分解为

表头 Head(LS) = α1 和

表尾 Tail(LS) = (α2, …, αn) 两部分。

表头可能是原子，也可能是列表，表尾必定是列表

例如: D = (E, F) = ((a, (b, c)), F)

Head(D) = E Tail(D) = (F)

Head(E) = a Tail(E) = ((b, c))

Head(((b, c))) = (b, c) Tail(((b, c))) = ( )

Head((b, c)) = b Tail((b, c)) = (c)

Head((c)) = c Tail((c)) = ( )

注意: 列表（）和（（））不同。

二、广义表的抽象数据类型的定义

ADT Glist {

数据对象: $D = \{e_i \mid i=1,2,..,n;\ n\geq 0;$

$e_i \in$ AtomSet 或 $e_i \in$ GList,

AtomSet为某个数据对象 $\}$

数据关系:

$$LR = \{<e_{i-1}, e_i> \mid e_{i-1}, e_i \in D,\ 2\leq i\leq n\}$$

基本操作:

} ADT Glist

基本操作

- 结构的创建和销毁

InitGList(&L);　DestroyGList(&L);

CreateGList(&L, S);　CopyGList(&T, L);

- 插入和删除操作

InsertFirst_GL(&L, e);

DeleteFirst_GL(&L, &e);

- 状态函数

GListLength(L);　GListDepth(L);

GListEmpty(L);　GetHead(L);　GetTail(L);

- 遍历

Traverse_GL(L, Visit());

5.5 广义表的表示方法

由于广义表中的数据元素可以具有不同的结构，（或是原子，或是列表），因此难以用顺序存储结构表示，通常采用**链式存储结构**，每个数据元素可用一个结点表示。

由于列表中的数据元素可能是原子或列表，因此需要**两类结点**：一类是**表结点**（表示列表），另一类是**原子结点**（表示原子）

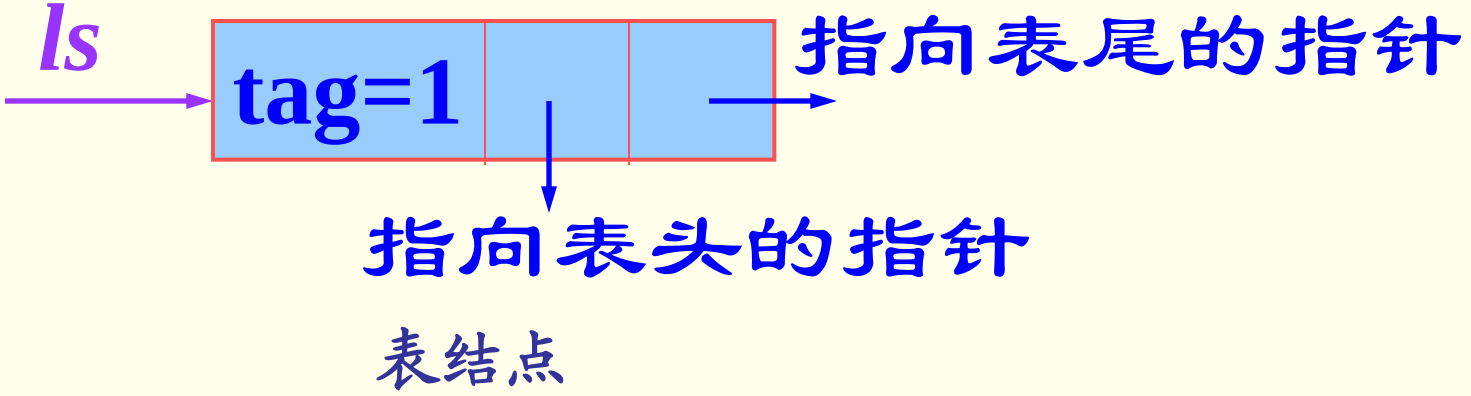
构造存储结构的两种分析方法:

1) 表头、表尾分析法:

若列表不空，将列表分解成表头和表尾。

空表　　*ls=NIL*

非空表 *ls*



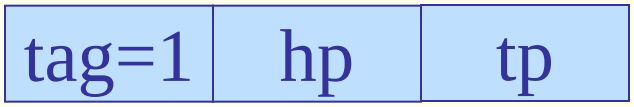
表结点

若表头为原子，则为

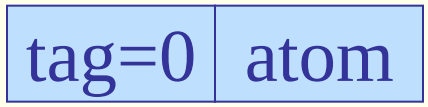
tag=0	atom

原子结点

依次类推。



表结点

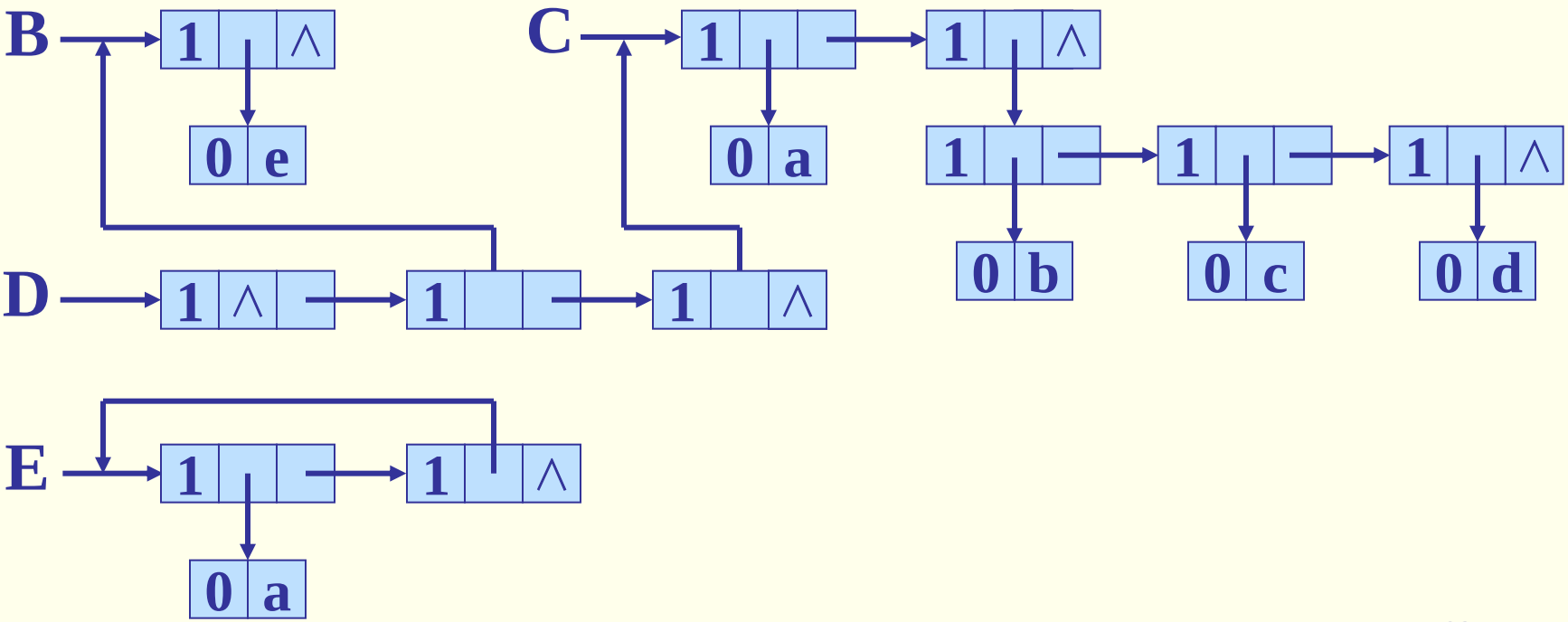


原子结点

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

E=(a,E)　　上述广义表如何用链式结构存储

A=NIL



特点：

(1) 除空表的列表指针为空外，对任何非空列表，其列表指针均指向一个表结点，且该结点中的**hp域**指示列表表头（或为原子结点，或为表结点），**tp域**指向列表表尾（除非表尾为空，则指针为空，否则必为表结点）；

(2) 容易分清列表中原子和子表所在层次；

(3) 最高层的表结点个数即为列表的长度。

优点：

给列表的运算带来方便，如求列表的长度和深度，求表头、表尾等。

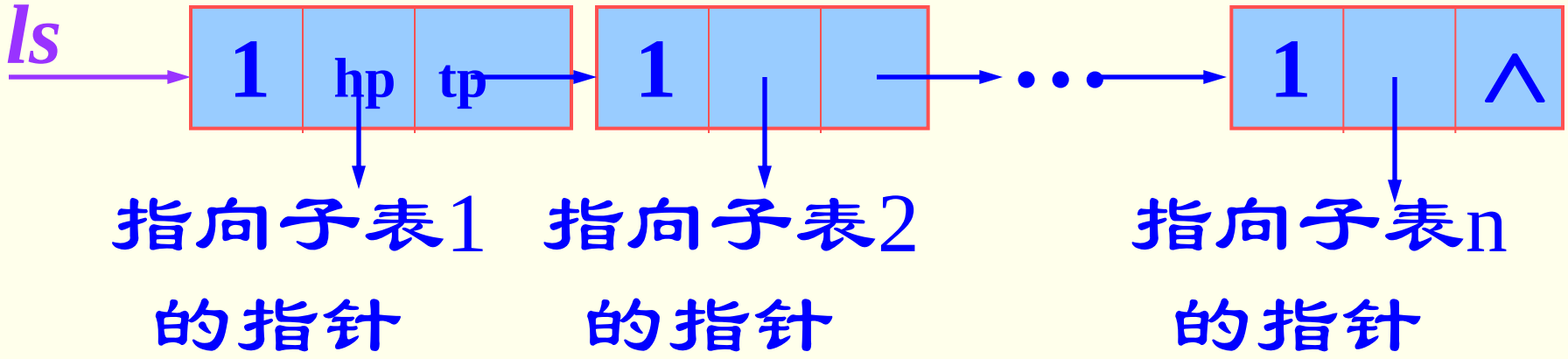
缺点：

表结点个数多，和列表中的括弧对数不匹配，也多占存储空间。

2) 子表分析法：

空表　　*ls=NIL*

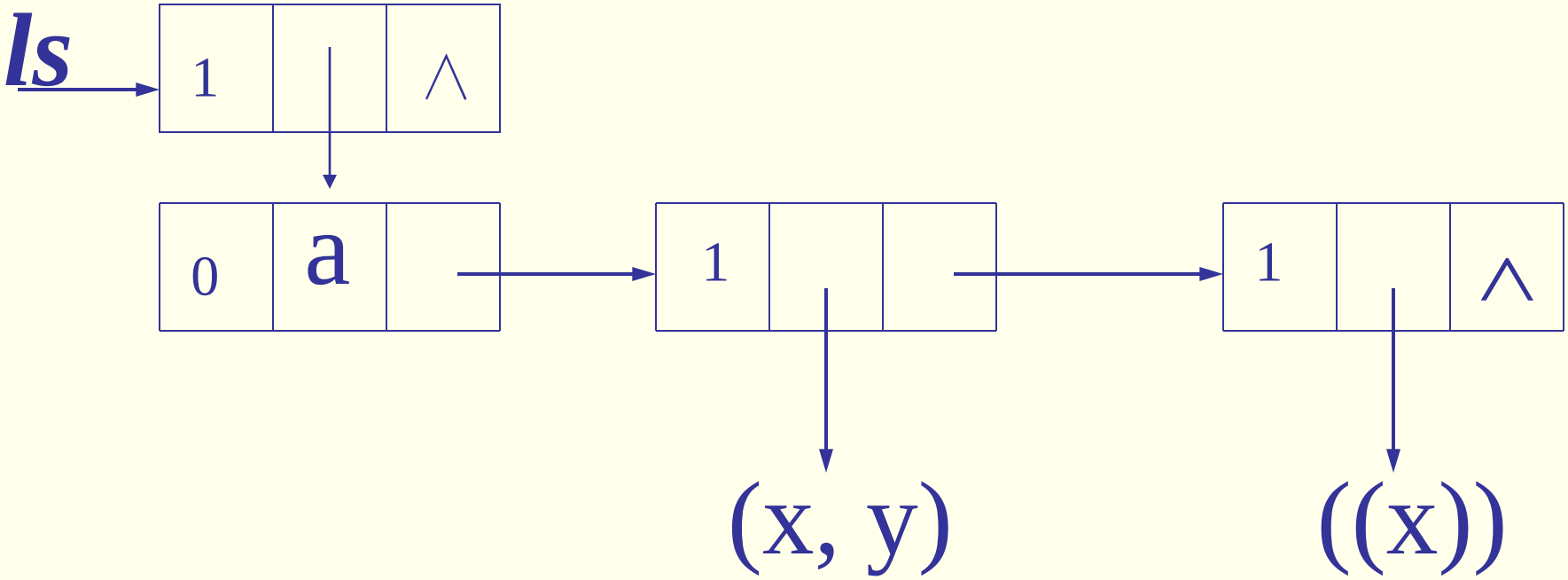
非空表

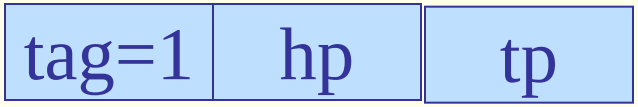


若子表为原子，则为

依次类推。

例如：　　**LS=(a, (x,y), ((x)))**



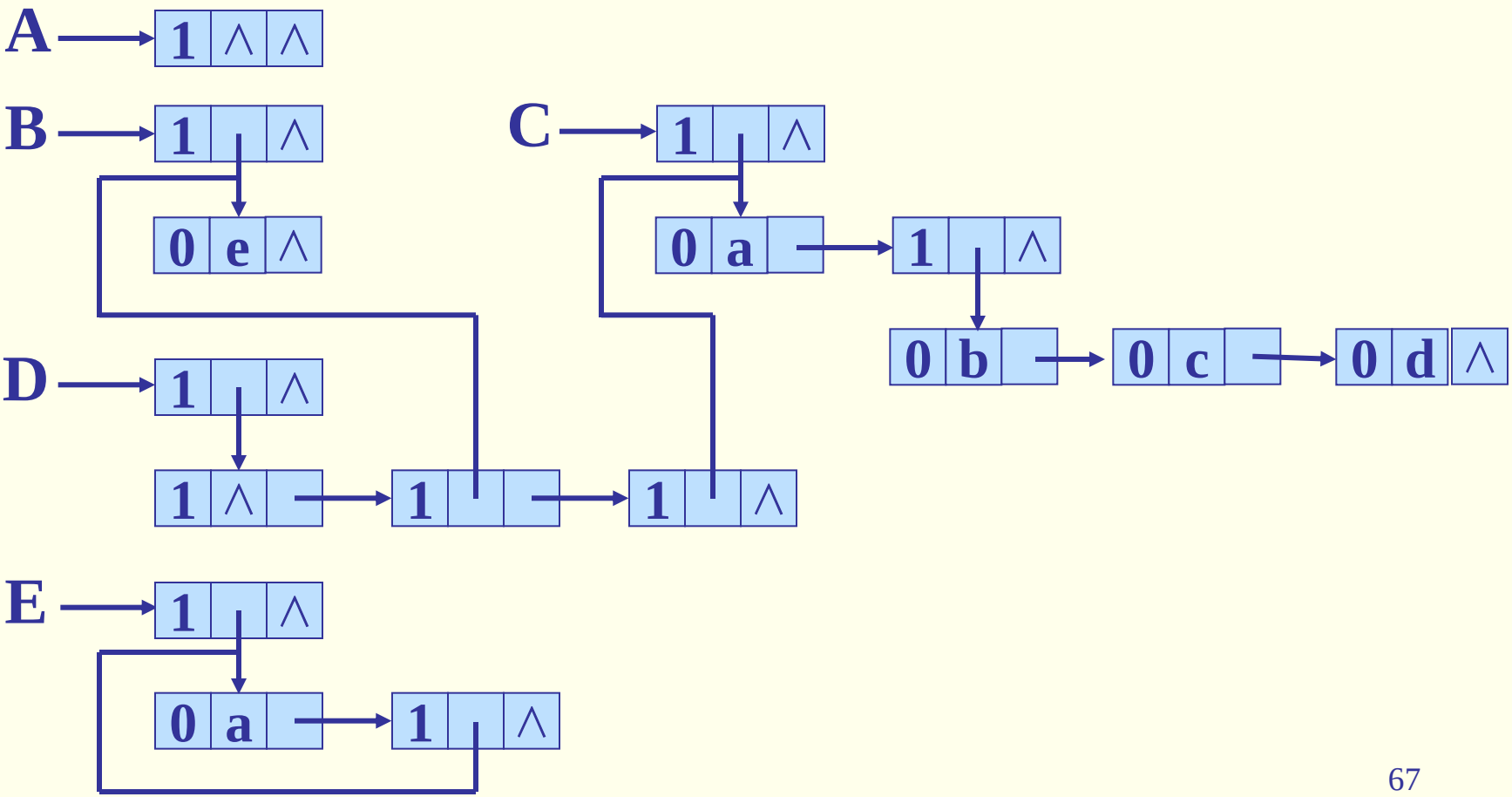


表结点

原子结点

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

E=(a,E)　　上述广义表如何用链式结构存储



特点：

表结点个数少，并且和列表中的括弧对数一致。

缺点：

写递归算法不方便。

例一 求广义表的深度

将广义表分解成 n 个子表，分别(递归)求得每个子表的深度，

广义表的深度=Max {子表的深度} +1

可以直接求解的两种简单情况为：

空表的深度 = 1

原子的深度 = 0

```
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
```

本章学习要点

(1) 了解数组的两种存储表示方法，并掌握数组在以行为主的存储结构中的地址计算方法。

(2) 掌握对特殊矩阵进行压缩存储时的下标变换公式。

(3) 了解稀疏矩阵的两类压缩存储方法的特点和适用范围，领会以三元组表示稀疏矩阵时进行矩阵运算采用的处理方法。

(4) 掌握广义表的结构特点及其存储表示方法，可根据自己的习惯熟练掌握任意一种结构的链表，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。

(5) 学习利用分治法的算法设计思想编制递归算法的方法。

作业：5.5 5.7 5.8 5.10 5.12 5.13

5.5 设有上三角矩阵$(a_{ij})_{nXn}$，将其上三角元素逐行存于数组B[m]中（m充分大），使得B[k]=a_{ij}且k=$f_1(i)+f_2(j)+c$。试推导出函数f_1,f_2和常数c（要求f_1和f_2中不含常数项）

5.7 设有三对角矩阵$(a_{ij})_{nXn}$，将其三条对角线上的元素逐行地存于数组B[3n-2]中，使得B[k]= a_{ij}，求：（1）用i,j表示k的下标变换公式

（2）用k表示i,j的下标变换公式

5.8 假设一个准对角矩阵,按以下方式存于一维数组B[4m]中

$$
\begin{pmatrix}
a_{11} & a_{12} & & & & & & & & & & \\
a_{21} & a_{22} & & & & & & & & & & \\
 & & a_{33} & a_{34} & & & & & & & & \\
 & & a_{43} & a_{44} & & & & & & & & \\
 & & & & \cdots & & & & & & & \\
 & & & & & a_{ij} & & & & & & \\
 & & & & & & \cdots & & & & & \\
 & & & & & & & & & a_{2m-1,2m-1} & a_{2m-1,2m} \\
 & & & & & & & & & a_{2m,2m-1} & a_{2m,2m}
\end{pmatrix}
$$

0	1	2	3	4	5	6		k			4m-2	4m-1
a_{11}	a_{12}	a_{21}	a_{22}	a_{33}	a_{34}	a_{43}	…	a_{ij}	…	$a_{2m-1,2m}$	$a_{2m,2m-1}$	$a_{2m,2m}$

写出由一对下标(i, j)求k的转换公式

5.10 求下列广义表操作的结果

⑴ GetHead【(p,h,w)】

⑵ GetTAil【(b,k,p,h)】

⑶ GetHead【((a,b),(c,d))】

⑷ GetTail【((a,b),(c,d))】

⑸ GetHead【GetTail【((a,b),(c,d))】】

⑹ GetTail【GetHead【((a,b),(c,d))】】

⑺GetHead【GetTail【GetHead【((a,b),(c,d))】】】

⑻ GetTail【GetHead 【GetTail 【((a,b),(c,d))】】】

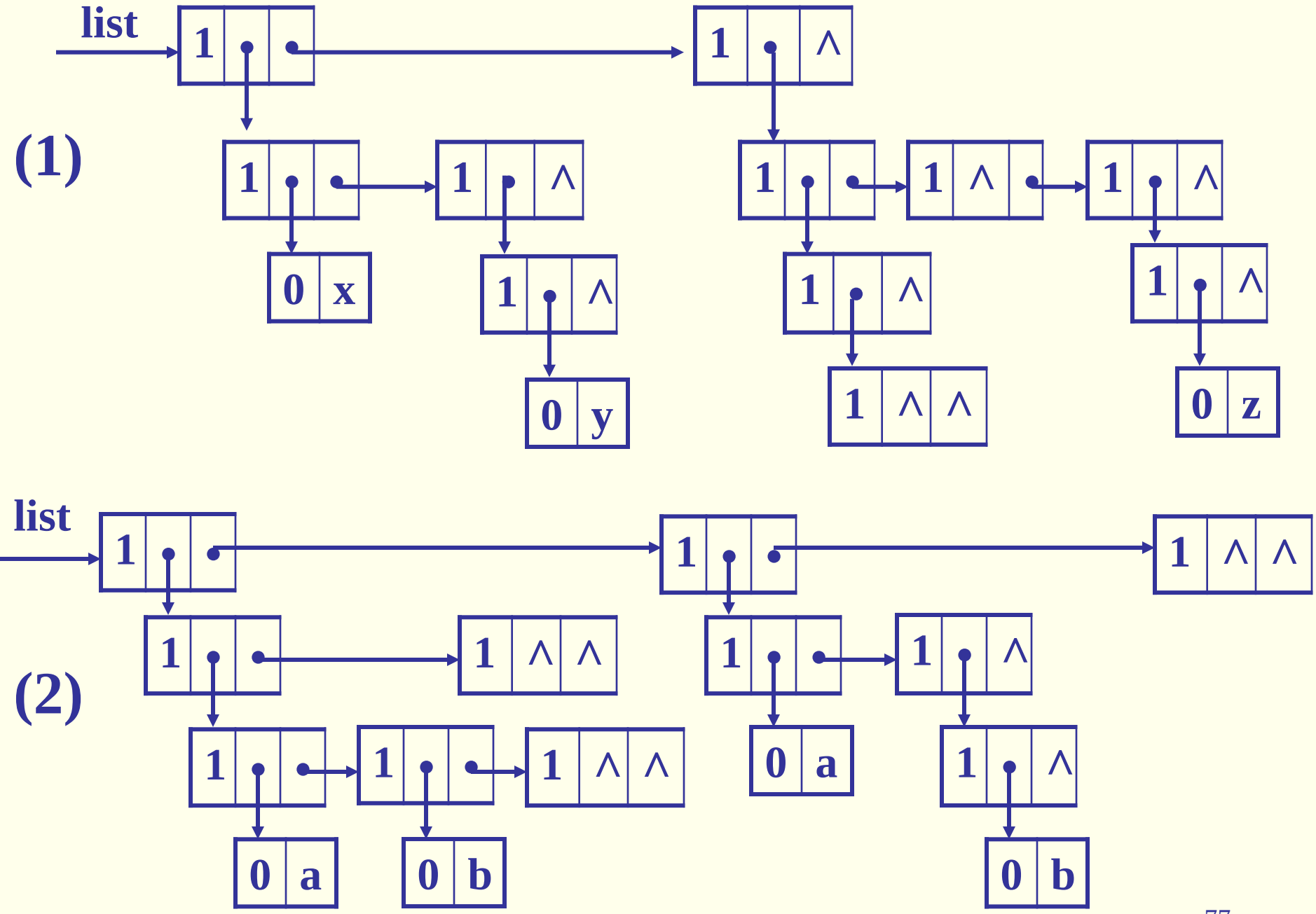
注意: 【】是函数的符号。

5.12 按教科书5.5节中图5.8所示结点结构，画出下列广义表的存储结构图，并求它的深度。

(1) ((( )),a,((b,c),( ),d),(((e))))

(2) ((((a),b)),((( ),d),(e,f)))

5.13 已知以下各图为广义表的存储结构图，其结点结构和5.12题相同，写出各图表示的广义表。


list
1
1
^
(1)
1
1
^
1
1
^
1
^
1
^
0
x
1
^
1
^
1
^
0
y
1
^
^
0
z
list
1
1
1
^
^
(2)
1
1
^
^
1
1
1
^
1
1
1
^
^
0
a
1
^
0
a
0
b
0
b