第四章 数组和广义表

史树敏
计算机学院

本章内容

- 数组的类型定义
- 数组的顺序表示和实现
- 特殊矩阵的压缩存储
- 广义表的类型定义
- 广义表的表示方法
- 广义表操作的递归函数

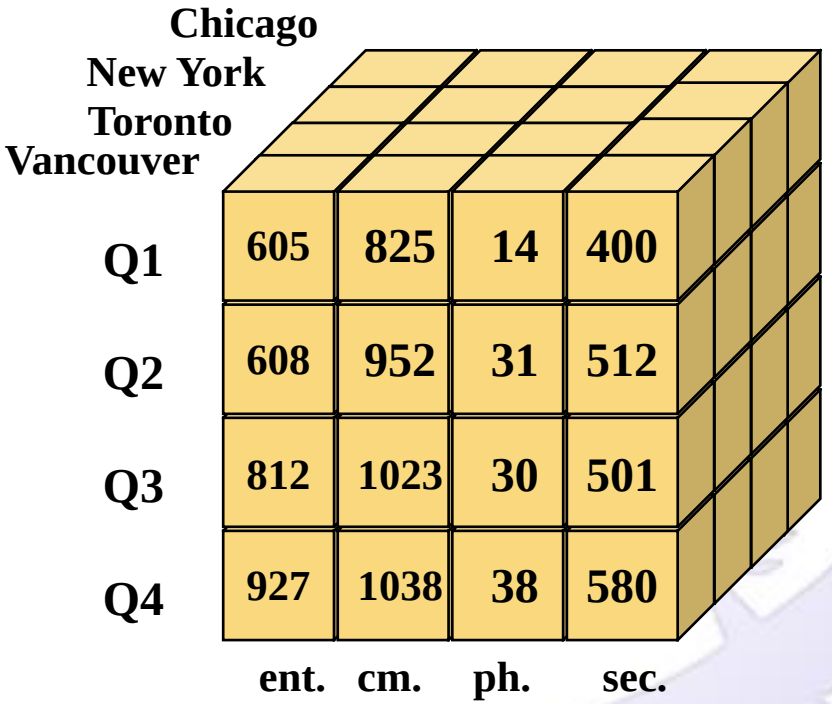
数组

a_0	a_1	a_2								a_n

一维数组：顺序表

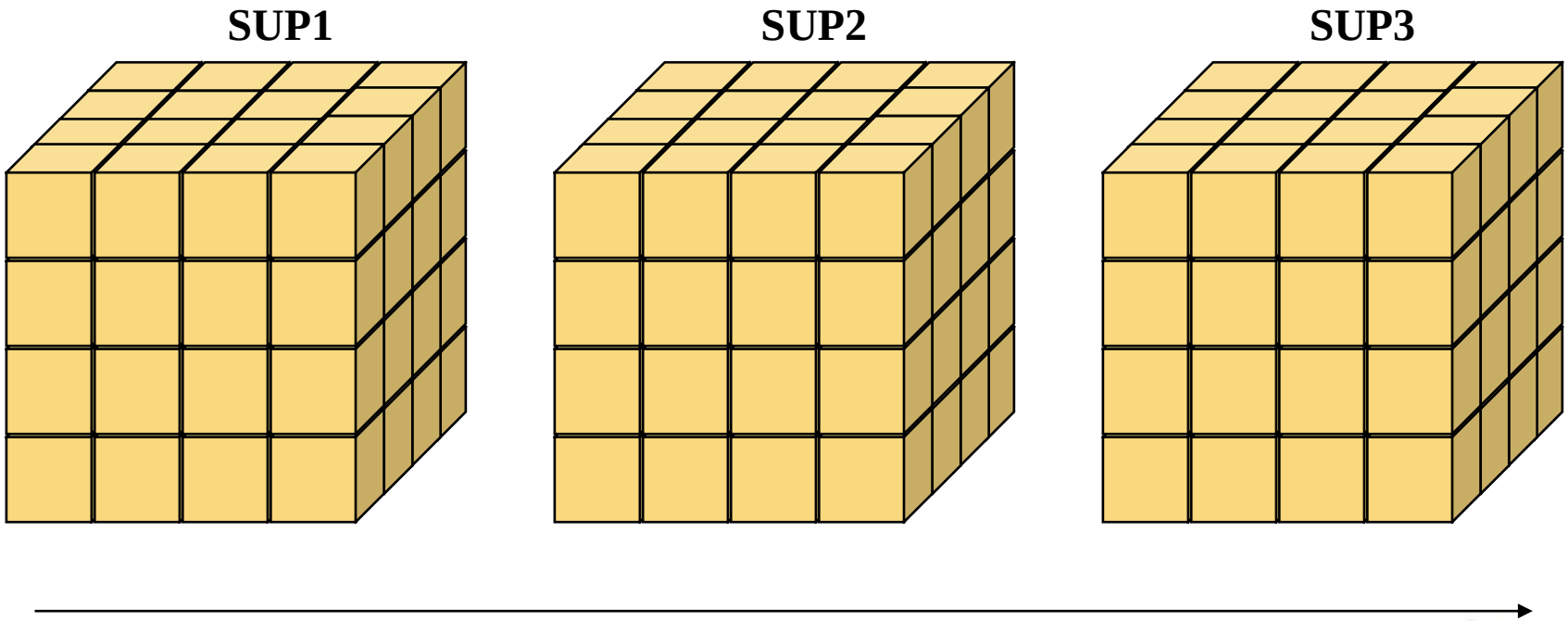
a_{00}	a_{01}	…	a_{0n-1}
a_{20}	a_{21}		a_{2n}
…			…
a_{n0}	a_{n1}	…	a_{n-1n-1}

二维数组



三维数组

数组



四维数组

数组的定义

- 一维数组常被称为向量（Vector）
- 二维数组 A[m][n] 可以看成由 m 个行向量组成的向量，可看成由 n 个列向量组成的向量。
- 一个二维数组类型可以定义为其分量类型为一维数组类型的一维数组类型:

```
typedef  T array2[m][n];        //T为元素类型
```

等价于:

```
typedef  T array1[n];           //行向量类型
typedef array1 array2[m];       //二维数组类型
```

数组的定义

- **数组的逻辑结构**

  1. 线性结构扩展

$$A_{M\times N}=\begin{pmatrix} a_{0\,0} & a_{0\,1} & \cdots & a_{0\,N-1} \\ a_{1\,0} & a_{1\,1} & \cdots & a_{1\,N-1} \\ & & & \\ a_{M-1\,0} & a_{M-1\,1} & \cdots & a_{M-1\,N-1} \end{pmatrix}$$

$$A=(A_0,\ A_1,\ \ldots,\ A_{N-1})$$

其中：

$$A_i=(a_{0i},\ a_{1i},\ \ldots,\ A_{m-1i})$$

$$(0\le i\le N-1)$$

二维数组是以线性表作为数据元素的线性表

数组的定义

- **数组的逻辑结构**

2.二维数组中每个元素都受两个线性关系的约束:行、列

$$A_{M\times N}=\begin{pmatrix} a_{0\,0} & a_{0\,1} & \cdots & a_{0\,N-1} \\ a_{1\,0} & a_{1\,1} & \cdots & a_{1\,N-1} \\ & aij & & \\ & \cdots\cdots & & \\ a_{M-1\,0} & a_{M-1\,1} & \cdots & a_{M-1\,N-1} \end{pmatrix}$$

在行关系中:

$a_{i,j}$ **直接前趋** $a_{i,j-1}$

$a_{i,j}$ **直接后继** $a_{i,j+1}$

在列关系中:

$a_{i,j}$ **直接前趋** $a_{i-1,j}$

$a_{i,j}$ **直接后继** $a_{i+1,j}$

N维数组中的每个元素受N个线性关系的约束

数组的类型定义

ADT Array {//二维数组的定义

数据对象:

$0 \le i \le b_1-1,\ 0 \le j \le b_2-1$

$D = \{a_{ij} \mid a_{ij} \in ElemSet\}$

b_1:第1维的长度

b_2:第2维的长度

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: ……

} ADT Array

数组的类型定义

b_i:第i维的长度

n:数组的维数

ADT Array {//n维数组

数据对象:

$j_i = 0,\ldots,b_i - 1,\ i = 1,2,\ldots,n$

$D = \{a_{j_1,j_2,\ldots,j_i,j_n} \mid a_{j1,j2,\ldots,ji,jn} \in ElemSet\}$

数据关系:

$R = \{R1, R2, \ldots, Rn\}$//定义每一维的序列关系

$Ri = \{<a_{j_1,\ldots\ j_i,\ldots\ j_n}, a_{j_1,\ldots j_i+1,\ldots j_n}> \mid$

$0 \le j_i \le b_i - 2,\ i = 2,\ldots,n,$

$0 \le j_k \le b_k - 1,\ 1 \le k \le n$ 且 $k \ne i$

$a_{j1,\ldots\ j_i,\ldots\ jn}, a_{j1,\ldots j_i+1,\ldots jn} \in ElemSet\}$

基本操作:

} ADT Array

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。**

基本操作

- Value(A, &e, $index_1$, ..., $index_n$)
  - 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, $index_1$, ..., $index_n$)
  - 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。

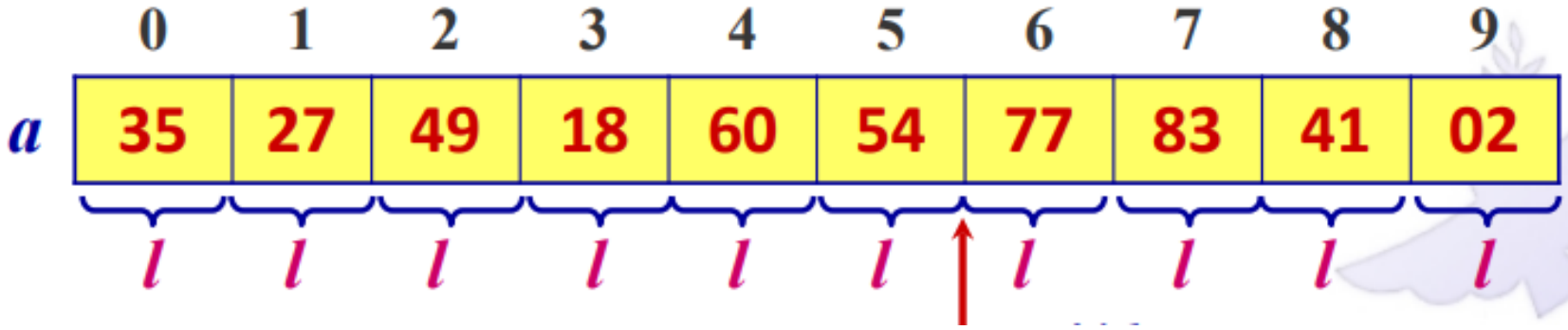
数组的类型定义

- 数组是一种很特殊的数据结构
- 数组是存储结构，是多种编程语言内置的数据类型，其操作只有按下标“读 / 写”
- 数组是一种逻辑结构
  - 一维数组属于线性结构，但不是线性表。因为一维数组中的数据可以不连续。
- 一维数组的数组元素为不可再分割的单元素时，是线性结构；但它的数组元素也是数组时，即为多维数组，则是非线性结构。

数组的顺序表示和实现

- 一维数组的连续存储表示
- 设每个数组元素占据相等的 l 个存储单元，第 0 号元素的存储地址为 a，则第 i 号数组元素的存储地址 LOC(i) 为：

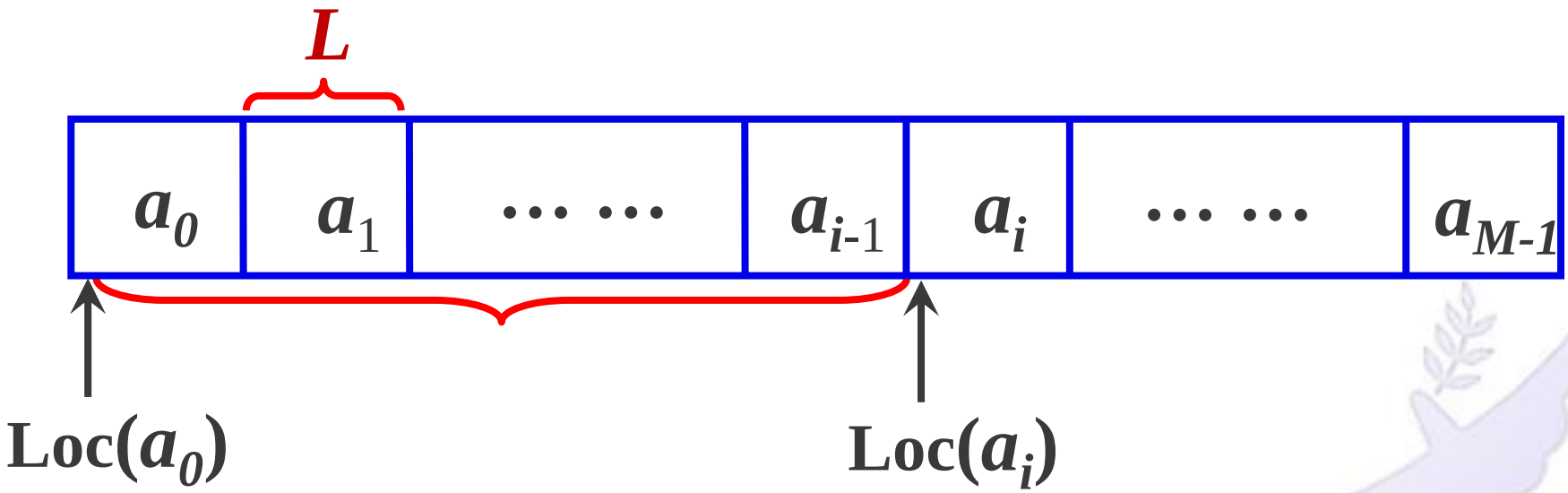
$$\mathrm{LOC}(A[i]) = \begin{cases} a, & i = 0\text{时} \\ \mathrm{LOC}(A[i-1]) + l = a + i * l, & i > 0\text{时} \end{cases}$$



数组的存储和实现

◆一维数组的存储结构与寻址:

设一维数组有M个元素, 下标范围区间为 [0, M-1], 每个元素占 L 个存储单元。



a_i 的存储地址：$Loc(a_i) = Loc(a_0) + i \times L$

数组的顺序表示和实现

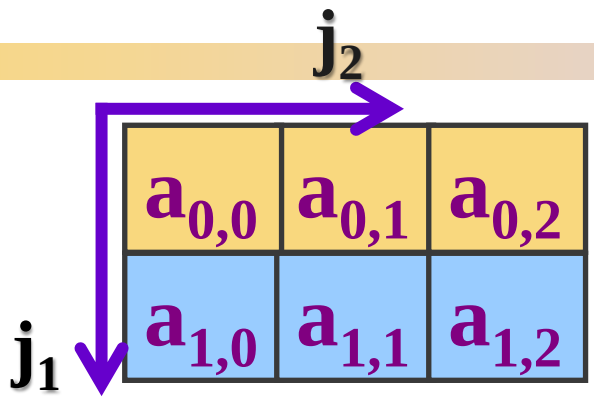
- 多维数组的连续存储表示
- 类型特点:
  - 1) 按下标“读/写”；
  - 2) 数组是多维的结构，而存储空间是一个一维的结构。
- 两种顺序映象方式:
  - 1)以行序为主序 (低下标优先);
  - 2)以列序为主序 (高下标优先)。

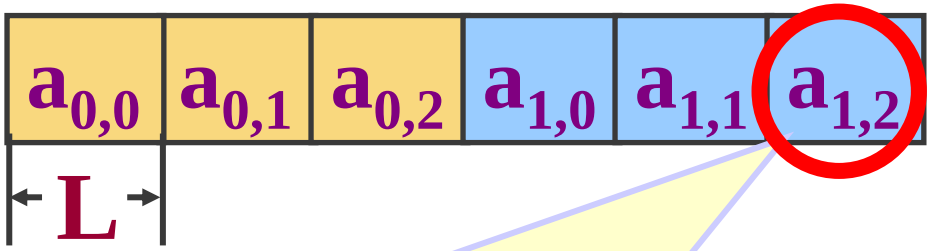
以“行序为主序”的存储映象

A(2,3)

$b_1 = 2$

$b_2 = 3$





$$LOC(1,2) = LOC(0,0) + (3\times1 + 2)\times L$$

二维数组A中任一元素a_{j_1,j_2} 的存储位置：

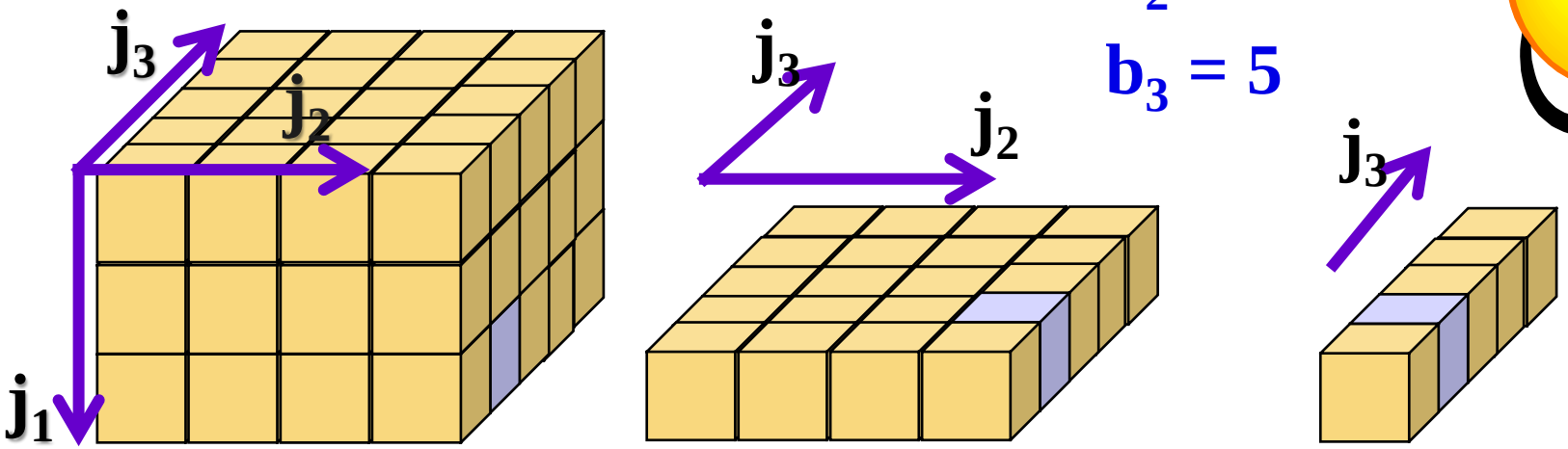
$$LOC(j_1,j_2) = LOC(0,0) + (b_2\times j_1 + j_2)\times L$$

基地址

数组A(3,4,5)，求LOC(2,3,1)? $b_1 = 3$

$b_2 = 4$

$b_3 = 5$



$LOC(2,3,1) = LOC(0,0,0) + (4\times5\times2 + 5\times3 + 1)\times L$

三维数组A中任一元素a_{j_1,j_2,j_3}的存储位置：

$$LOC(j_1,j_2,j_3) = LOC(0,0,0) +$$

$$+(b_2\times b_3\times j_1 + b_3\times j_2 + j_3)\times L$$

以“行序为主序”的存储映象

- **n 维数组数据元素存储位置的映象关系：**

$$\mathrm{LOC}(j_1, j_2, \ldots, j_n) = \mathrm{LOC}(0,0,\ldots,0) +$$

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

$$\mathrm{LOC}(0,0,\ldots,0) + L\sum_{i=1}^{n} c_i j_i \quad (c_n = 1, c_{i-1} = b_i \times c_i, 1 < i \le n)$$

称为 n 维数组的映象函数。

即数组元素的存储位置是其下标的线性函数。

（数组动态存储的理论基础）

数组的存储和实现

- **二维数组——静态数组表示法**

  typedef ElemType Array[m*n];

  Array A;

$$A_{m\times n}=\begin{pmatrix} a_{0\,0} & a_{0\,1} & \cdots & a_{0\,n-1} \\ a_{1\,0} & a_{1\,1} & \cdots & a_{1\,n-1} \\ & & & \cdots\cdots \\ a_{m-1\,0} & a_{m-1\,1} & \cdots & a_{m-1\,n-1} \end{pmatrix}$$

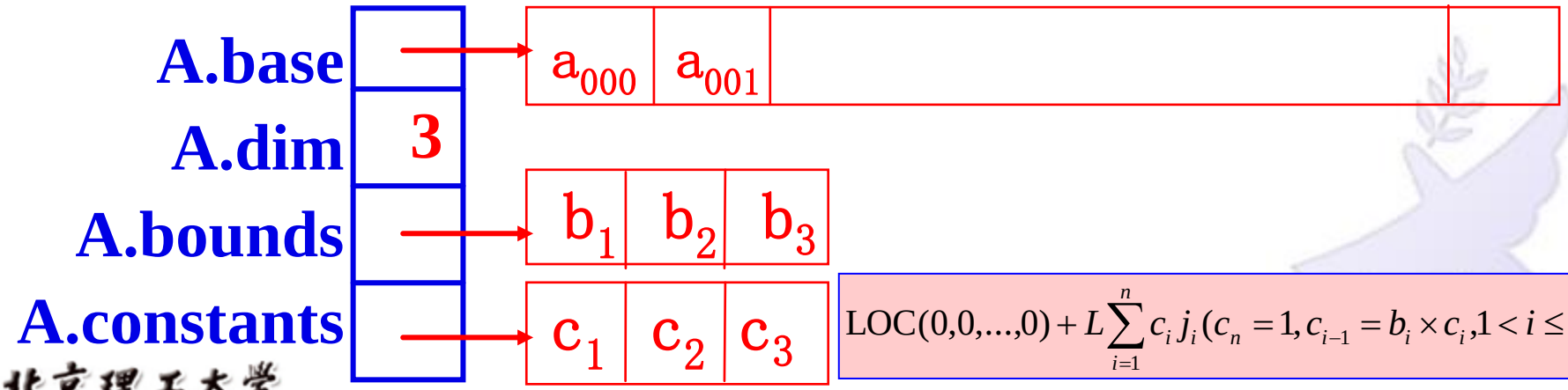
$a_{0\,0}$	……	$a_{0\,n-1}$	$a_{1\,0}$	……	$a_{1\,n-1}$	……	$a_{m-1\,0}$	……	$a_{m-1\,n-1}$

数组的存储和实现

- **数组的动态表示法**

```
typedef struct {
   ElemType * base;     // 动态空间基址
   int  dim;            // 数组维数
   int  * bound;        // 维界基址（各维大小）
   int  * constants;    // 数组映像函数常量基址
} Array;
```

A[b1][b2][b3]



$$LOC(0,0,...,0) + L\sum_{i=1}^{n} c_i j_i \quad (c_n = 1, c_{i-1} = b_i \times c_i, 1 < i \le n)$$

数组的存储和实现

```
Status InitArray( Array2 &A,  int b1,  int b2 )
 { //数组的初始化
   if ( b1<=0 || b2<=0 )
        return ERROR;
   else
   {  A.bound1=b1;
      A.bound2=b2;
      if ( ! ( A.base = (ElemType*)
                          malloc( b1*b2*sizeof( ElemType ) ) ) )
          exit( OVERFLOW );
      return OK;
   }
}// InitArray()
```

数组的存储和实现

```
Status DestroyArray( Array2 &A )
{ /* 销毁数组A */
    if ( A.base )
    {   free( A.base );
        A.base = NULL;
        A.bound1 = 0;
        A.bound2 = 0;
        return OK;
    }
    else   return ERROR;
}// DestroyArray()
```

数组的存储和实现

```
Status Value ( Array2 A,  ElemType  &e, int  j1,  int j2 )
{ /* 若各下标 不超界，则将所指定的A的元素值赋值给
     e，并返回OK */
    if ( ( j1<0 ) || ( j1>=A.bound1 )
                  || ( j2<0 ) || ( j2>=A.bound2 ) )
        return  ERROR;
    e = *( A.base + A.bound2*j1+ j2 );
    return OK;
}// Value ()
```

数组的存储和实现

```
Status Assign( Array2  &A,  ElemType  e, int  j1,  int  j2 )
{ /* 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。 */
    if ( ( j1<0 ) || ( j1>=A.bound1 ) || ( j2<0 ) ||
                                          ( j2 >= A.bound2 ) )
        return  ERROR;
    *( A.base + A.bound2*j1+ j2 ) = e;
    return OK;
}// Assign()
```

特殊矩阵的压缩存储

- **特殊矩阵的种类**
- **1) 特种矩阵**
  - ¶ 非零元在矩阵An×n中的分布有一定规律
  - ¶ 例如: 对称矩阵、三角矩阵、对角矩阵
- **2) 随机稀疏矩阵**
  - ¶ 非零元很少，且在矩阵Am×n中随机出现

$$\begin{bmatrix} 1 & 2 & 3 & 30 \\ 2 & 4 & 6 & -1 \\ 3 & 6 & 7 & -8 \\ 30 & -1 & -8 & 10 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 \\ 3 & 6 & 7 & 0 \\ 0 & -1 & -8 & 10 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

对称矩阵的压缩存储

核心问题：确定a_{ij}在一维数组中的存储位置 k

- **存储下三角矩阵(行序优先)：**用一维数组按行优先存储下三角元素。任意位置的元素a_{ij}与它的存储位置k是一一对应的;

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

- **关系如下：**

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

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

B= 0 1 2 3 4 5 ... $n(n+1)/2 -1$

a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	.....	a_{n1}	a_{n2}	.....	a_{nn}

三对角矩阵的压缩

- 所有非零元素都集中在主对角线及其相邻两侧的对角线上
- 在三条对角线上的元素 aij 满足：

  $0 \le i \le n-1,\ i-1 \le j \le i+1$
- 共有3n-2个非零元素
- 行序优先

$$A=\begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & a_{n-2n-3} & a_{n-2n-2} & a_{n-2n-1} \\ 0 & 0 & 0 & 0 & a_{n-1n-2} & a_{n-1n-1} \end{bmatrix}$$

	0	1	2	3	4	5	6	7			3n−3
B	a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	…	a_{n-1n-2}	a_{n-1n-1}

稀疏矩阵

- 假设 m 行 n 列的矩阵，其中非零元素（t 个）占总数的比例小于5%的矩阵为稀疏矩阵。

  即，稀疏因子 $\delta \leq 0.05$ 的矩阵为稀疏矩阵。

- 稀疏因子定义为：

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

42（6×7）个元素
只有8个非零元素

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

稀疏矩阵的存储

- **以二维数组表示高阶的稀疏矩阵，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: 能尽可能快地 找到与下标值(i，j)对应的元素;
  4. 操作方便: 能尽可能快地 找到同一行或同一列的非零值元素。

随机稀疏矩阵的压缩存储方法

- 稀疏矩阵的**顺序存储**
  - 三元组顺序表
  - 行逻辑链接顺序表
- 稀疏矩阵的**链式存储**
  - 简单链式存储
  - 行链表组
  - 十字链表

三元组顺序表

行下标	列下标	数据

按行序存储

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

M.data

	i	*j*	*e*
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

行数M.Rows	6
列数M.Cols	7
元素个数M.Terms	8

三元组顺序表的定义

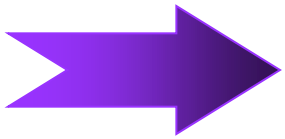
```
#define  MAXSIZE  12500
typedef struct {
    int  i, j;      //该非零元的行row下标和列col下标
    ElemType  e;    // 该非零元的值
} Triple;  // 三元组类型
```

```
typedef struct {
    Triple  data[MAXSIZE]; //三元数组
     int Rows, Cols, Terms;//行数、列数、元素个数
} SparseMatrix;  // 稀疏矩阵类型
```

如何求转置矩阵?

$$\begin{bmatrix} 0 & 14 & 0 & 0 & -5 \\ 0 & -7 & 0 & 0 & 0 \\ 36 & 0 & 0 & 28 & 0 \end{bmatrix}$$

M



$$\begin{bmatrix} 0 & 0 & 36 \\ 14 & -7 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 28 \\ -5 & 0 & 0 \end{bmatrix}$$

T

$$T_{ij} = M_{ji}$$

用常规的二维数组表示时的算法

一般矩阵的转置算法

```
…

int  a[m][n], b[n][m];

for ( i =0; i<m; ++i )
      for ( j = 0; j<n; ++j )

            b[ j ][ i ] = a[ i ][ j ];
```

用常规的二维数组表示时的算法

$$T_{ij} = M_{ji}$$

```
for (col=0; col<=Cols-1; ++col)
    for (row=0; row<=Rows-1; ++row)
        T[col][row] = M[row][col];
```

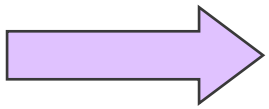
时间复杂度为: O(Rows×Cols)

用常规的二维数组表示时的算法

M.data

	i	j	e
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

M. Rows	6
M. Cols	7
M. Terms	8



T.data

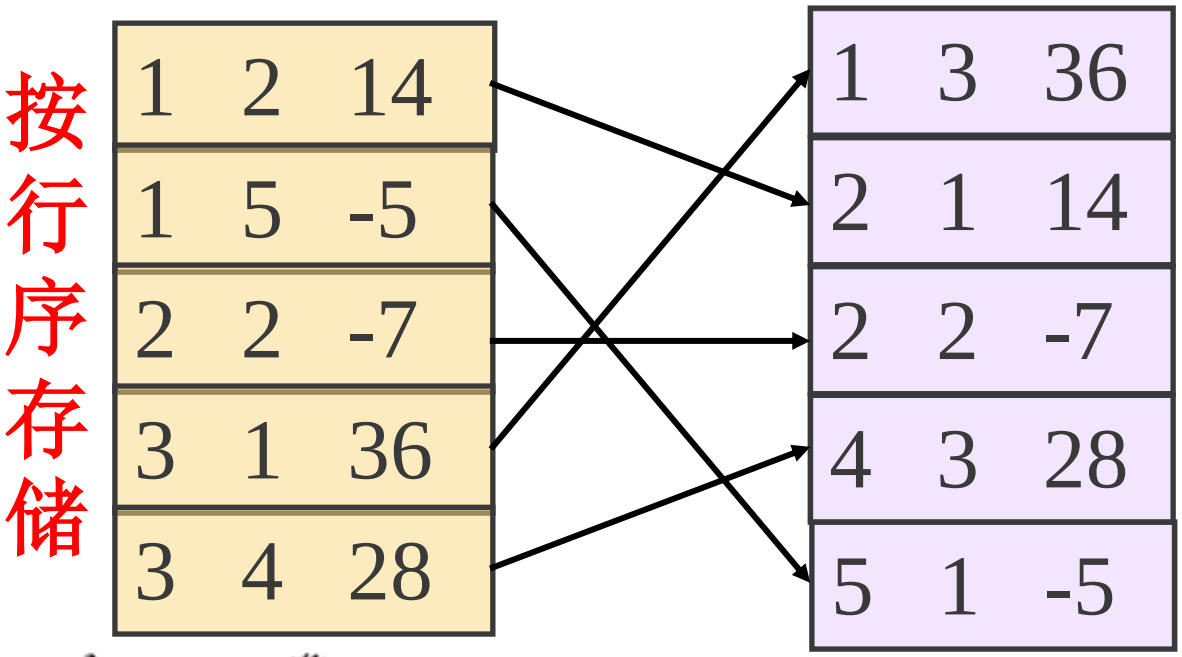
	i	j	e
1	1	3	−3
2	1	6	15
3	2	1	12
4	2	5	18
5	3	1	9
6	3	4	24
7	4	6	−7
8	6	3	14

T. Rows	7
T. Cols	6
T. Terms	8

用“三元组”表示时如何实现？

$$M=\begin{bmatrix}0 & 14 & 0 & 0 & -5\\ 0 & -7 & 0 & 0 & 0\\ 36 & 0 & 0 & 28 & 0\end{bmatrix}$$

$$T=\begin{bmatrix}0 & 0 & 36\\ 14 & -7 & 0\\ 0 & 0 & 0\\ 0 & 0 & 28\\ -5 & 0 & 0\end{bmatrix}$$



基本操作：

按照M的列序，依次从M中找出属于当前列的元素放在T中

用“三元组”表示时的操作步骤

- void TranMatrix (SparseMatrix M, SparseMatrix &T)
- 设置T的各个参数：
  - T.Rows=M.Cols; T.Cols=M.Rows; T.tu=M.tu;
- 设指向T中当前元素的指针为q；
- 每次得到T中的一行元素，设当前行号为row
  - for (row=0; row<=T.Rows-1; ++row)
  - //一行中的每一个元素怎么得到?
  - //从M中所有的元素中找其列号j==row的元素
    - for (p=0;p<=M.tu-1;++p)
    - 条件：M.data[p].j == row

```
void TranMatrix(TSMatrix M, TSMatrix &T)
{ //采用三元组表存储稀疏矩阵，求M的转置矩阵T
T.Rows=M.Cols; T.Cols=M.Rows;   T.tu=M.tu;
  if (T.tu <=0 ) return;
  q=0;  // q为T.data[ ]当前三元组的位置(下标)
  for (row=0; row<=T.Rows-1;  ++row)
    for (p=0;p<=M.tu-1;++p)//p:扫描M.data指示器
          if (M.data[p].j == row)
          {     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;
          }
}//  TransposeSMtrix
```

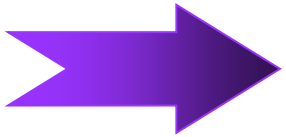
复杂度：	O(Cols*tu)

用“三元组”表示时的时间复杂度

- 时间复杂度分析
  - 转置算法 TranMatrix（）的时间复杂度为 O(Cols×tu)
  - 当非零元的个数tu 和矩阵元素个数 Cols×Rows 同数量级时，转置运算算法的时间复杂度就“上升”为 O(Cols×Rows×Cols)
  - 所以此算法一般用于 tu << Cols×Rows 的情况

能否直接放到正确位置?

$$\begin{bmatrix} 0 & 14 & 0 & 0 & -5 \\ 0 & -7 & 0 & 0 & 0 \\ 36 & 0 & 0 & 28 & 0 \end{bmatrix}$$



$$\begin{bmatrix} 0 & 0 & 36 \\ 14 & -7 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 28 \\ -5 & 0 & 0 \end{bmatrix}$$

矩阵快速转置算法

$$\begin{bmatrix} 0 & 14 & 0 & 0 & -5 \\ 0 & -7 & 0 & 0 & 0 \\ 36 & 0 & 0 & 28 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 36 \\ 14 & -7 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 28 \\ -5 & 0 & 0 \end{bmatrix}$$

- 1）确定M中每一列的第一个非零元素在T中的位置
- 2）将M中的元素放在T中恰当位置

按行序存储

1	2	14
1	5	-5
2	2	-7
3	1	36
3	4	28

1	3	36
2	1	14
2	2	-7
4	3	28
5	1	-5

M的列	T中的位置
1	1
2	2
3	4
4	4
5	5

确定M中每一列的第一个非零元素在T中的起始位置

矩阵快速转置算法

$$\begin{bmatrix} 0 & 14 & 0 & 0 & -5 \\ 0 & -7 & 0 & 0 & 0 \\ 36 & 0 & 0 & 28 & 0 \end{bmatrix}$$

- 1）M中每一列的第一个非零元素在T中开始的位置
- num[col] : M中第col列非零元素个数
- cpot[col]: M中第col列第一个非零元素的位置

1	2	14
1	5	-5
2	2	-7
3	1	36
3	4	28

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

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

矩阵快速转置算法

$$\begin{bmatrix} 0 & 14 & 0 & 0 & -5 \\ 0 & -7 & 0 & 0 & 0 \\ 36 & 0 & 0 & 28 & 0 \end{bmatrix}$$

- 2）将M中的元素放在T中恰当位置

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

1 2 14

1 3 36

按行序存储

```
for (p=1; p<=M.tu; ++p){
    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]++;
}
```

矩阵快速转置算法的操作步骤

1. 设置T的各个参数：
   - T.Rows=M.Cols; T.Cols=M.Rows; T.tu=M.tu;
2. 依次累加出M中每一列的元素个数
3. 依次求M中每一列的元素在T中的起始位置
4. 依次将M中的元素放到正确的位置

- num[col] : M中第col列非零元素个数
- cpot[col]: M中第col列第一个非零元素的位置

```
Status FastTransposeSM
(SparseMatrix &T){
  T.Rows = M.Cols; T.Cols = M.Rows; T.tu = M.tu; //步骤1
   if (T.tu <=0) return NOELEM;

  for (col=1; col<= M.Cols; ++col)  num[col] = 0; //步骤2
  for (t=1; t<=M.tu; ++t)  ++num[M.data[t].j];

   cpot[1] = 1;                                    //步骤3
   for (col=2; col<=M.Cols; ++col)
         cpot[col] = cpot[col-1] + num[col-1];

    for (p=1; p<=M.tu; ++p) {…转置矩阵元素} //步骤4

   return OK;
} // FastTransposeSMatrix
```

O(Cols+tu)

算法分析

空间换时间

- 用常规的二维数组表示时的算法
  - 时间复杂度：O(Cols×Rows)
  - 空间复杂度：O(1)
- 矩阵转置算法
  - 时间复杂度：O(Cols×tu)
  - 空间复杂度：O(1)
- 矩阵快速转置算法
  - 时间复杂度：O(Cols+tu)
  - 空间复杂度：O(Cols)

行逻辑链接 顺序表

行下标	列下标	数据

- 带行表的三元组

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

M. Rpos[ ]

[1]	1
[2]	3
[3]	3
[4]	5
[5]	6
[6]	7

	i	j	e
M.Data[0]			
[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

按行序存储

行数M.Rows 6
列数M.Cols 7
元素个数M.tu 8

行逻辑连接，每行第1个元素在表中位置

带行表的三元组

```
#define  MAXROW  100
   typedef struct {
      Triple  data[ MAXSIZE ]; // 数据
      int  rpos[ MAXROW];       // 行表
      int  Rows, Cols, tu; ;          //行数，列数，元素个数
   } rtripletable;
```

带行表的三元组

给定一组下标，求矩阵指定元素的值

```
ElemType value(rtripletable M, int r, int c )
{
   p = M.rpos[ r ];            // 取r 行第一个非零元素位置
   while ( M.data[p].i==r && M.data[p].j < c )
      p++;                     // 从p开始，查找指定的列 c 位置
   if ( M.data[p].i == r && M.data[p].j == c )
      return M.data[p].e;          // 若找到，则取值
   else
      return 0;                    // 没找到，返回 0
} //value
```

矩阵的乘法

$$M=\begin{pmatrix}3 & 0 & 0 & 5\\ 0 & -1 & 0 & 0\\ 2 & 0 & 0 & 0\end{pmatrix}\quad (3*4)$$

$$N=\begin{pmatrix}0 & 2\\ 1 & 0\\ -2 & 4\\ 0 & 0\end{pmatrix}\quad (4*2)$$

$$Q=\begin{pmatrix}0 & 6\\ -1 & 0\\ 0 & 4\end{pmatrix}\quad (3*2)$$

$$Q(i,j)=\sum_{k=1}^{n1} M(i,k)\times N(k,j)$$

若用二维数组存储矩阵，乘法的算法包含三重循环。

用三元组表示

$$Q(i,j)=\sum_{k=1}^{n1} M(i,k)\times N(k,j)$$

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

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

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

1	1	1	2	6
2	2	2	1	-1
3	3	3	2	4

可以每次计算Q中的一行元素

考察N中第j行的元素：

- M每个元素M(i, j)分别与N中第j行的元素(j, ccol)相乘；
- 乘积累加到Q(i, ccol)中。

稀疏矩阵: 用三元组表示矩阵时的算法

矩阵乘法的基本思想：

- **每次计算Q中的一行元素，当前行号为arrow。**
- **设临时变量ctemp[ ]累加器保存该行元素，将其清零。**
- **计算每行元素：**
  - M中第arrow行的每一个元素(arrow, j)分别与N中第arrow行的元素(j, ccol)相乘，其结果累加到ctemp[ccol]中。
- **计算完后将该行元素压缩存储到Q的三元组中。**

稀疏矩阵：求矩阵Q=M*N，采用行逻辑链接顺序表

```
Status MultSMatrix(rtripletable M, rtripletable N, rtripletable &Q)
{  Q.Rows = M.Rows; Q.Cols = N.Cols;  Q.tu = 0; //初始化Q
   if (M.tu* N.tu != 0)                    //如果Q是非零矩阵
   {  for (arrow=1; arrow<=M.Rows; arrow++) {
        //逐行求积，处理M的每一行
        ctemp[ ] = 0;
        //计算Q中第arrow行的积并存入ctemp[ ]中;
        //将ctemp[ ]中非零元素压缩存储到Q.data;
      }// for arrow
   }//if
  return OK;
} // MultSMatrix
```

稀疏矩阵：求矩阵Q=M*N, 采用行逻辑链接顺序表

```
//计算Q中第arrow行的积并存入ctemp[ ]中
  Q.rpos[arrow] = Q.tu+1;
  if (arrow < M.Rows)    //设tp指向M第arrow+1行的第一个元素
        tp = M.rpos[arrow+1];
  else tp = M.tu + 1;
  p = M.rpos[arrow]; //设p指向M第arrow行的第一个元素
  for (;p<tp; p++) {     // 对M中当前行的每一非零元素
   brow = M.data[p].j; //brow是M当前元素对应的在N中行号
   if (brow < N.Rows) t = N.rpos[brow+1];
   else t = N.tu  + 1;
   for(q = N.rpos[arrow]; q<t; q++){
      ccol = N.data[q].j;       // 乘积元素在Q中的列号
      ctemp[ccol] += M.data[p].e * N.data[p].e;
      //for q
   }// for p
```

稀疏矩阵：求矩阵Q=M*N，采用行逻辑链接顺序表

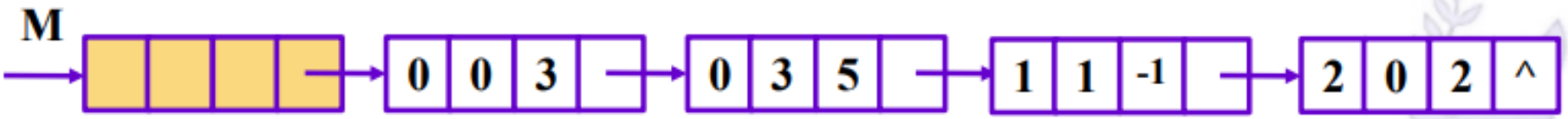
```
//将ctemp[ ]中非零元素压缩存储到Q.data
for (ccol = 1; ccol<=Q.tu; ccol++) {
    // 压缩存储当前行非零元素
    if (temp[ccol] != 0){
        if(++Q.tu > MAXSIZE) return ERROR;
        Q.data[Q.tu] = ( arrow, ccol, ctemp[ccol];
}// if temp
```

若M和N都是稀疏矩阵，M(m1*n1)、N(m2*n2)，
算法复杂度可以降至O(m1*n2)

稀疏矩阵的链式存储

- (1)简单链式存储
  - 不能直接存取元素
  - 但增加了灵活性，易于增删非零元素

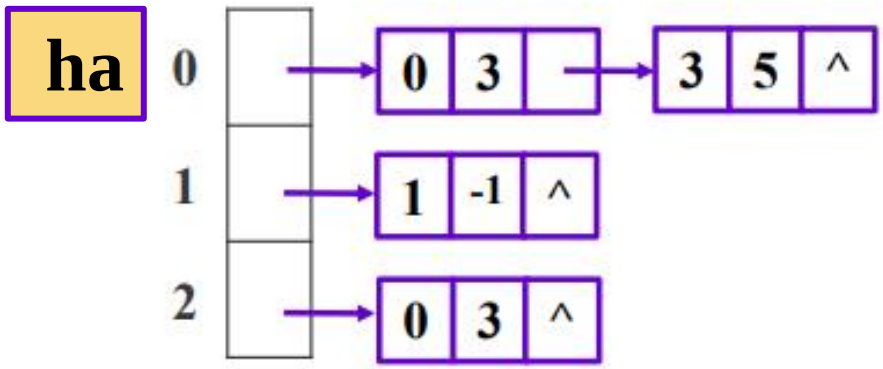
$$\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$



稀疏矩阵的链式存储

- (2)行链表组
  - 提高了存取元素的效率
  - 易于增删非零元素

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



十字链表

- **(3)十字链表（正交链表）**

  采用链接存储结构存储三元组表，每个非零元素对应的三元组存储为一个链表结点：

row	col	item
down		right

row： 非零元素的行号

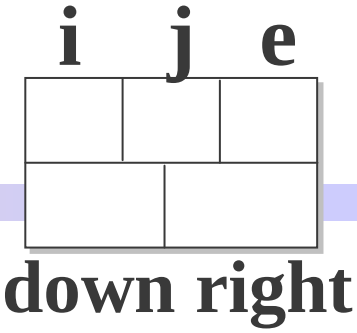
col： 非零元素的列号

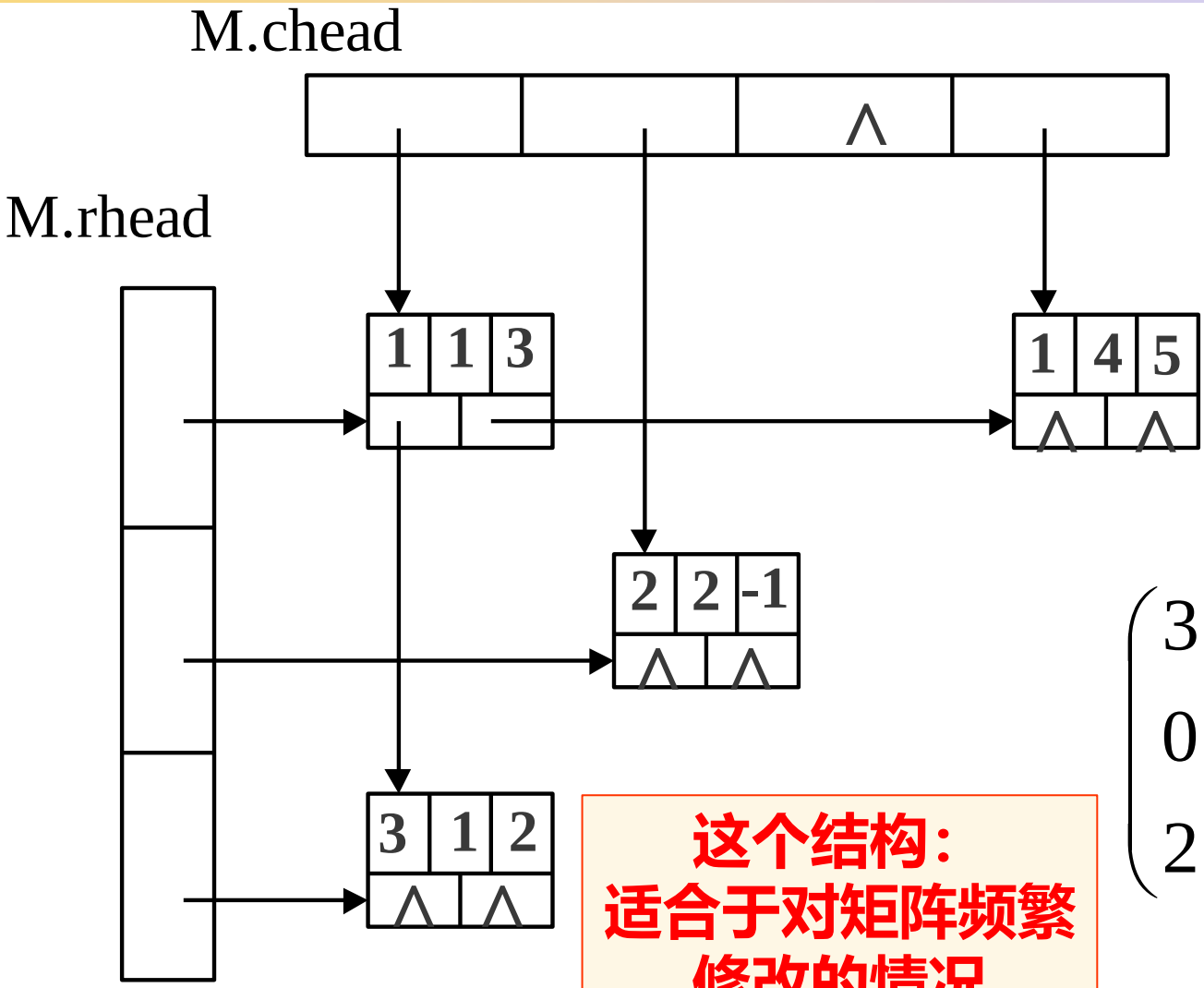
item： 非零元素的值

right： 指向同一行中的下一个三元组

down： 指向同一列中的下一个三元组

十字链表



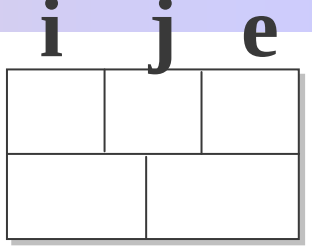


$$\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

这个结构：
适合于对矩阵频繁修改的情况

十字链表的类型定义

Orthogonal

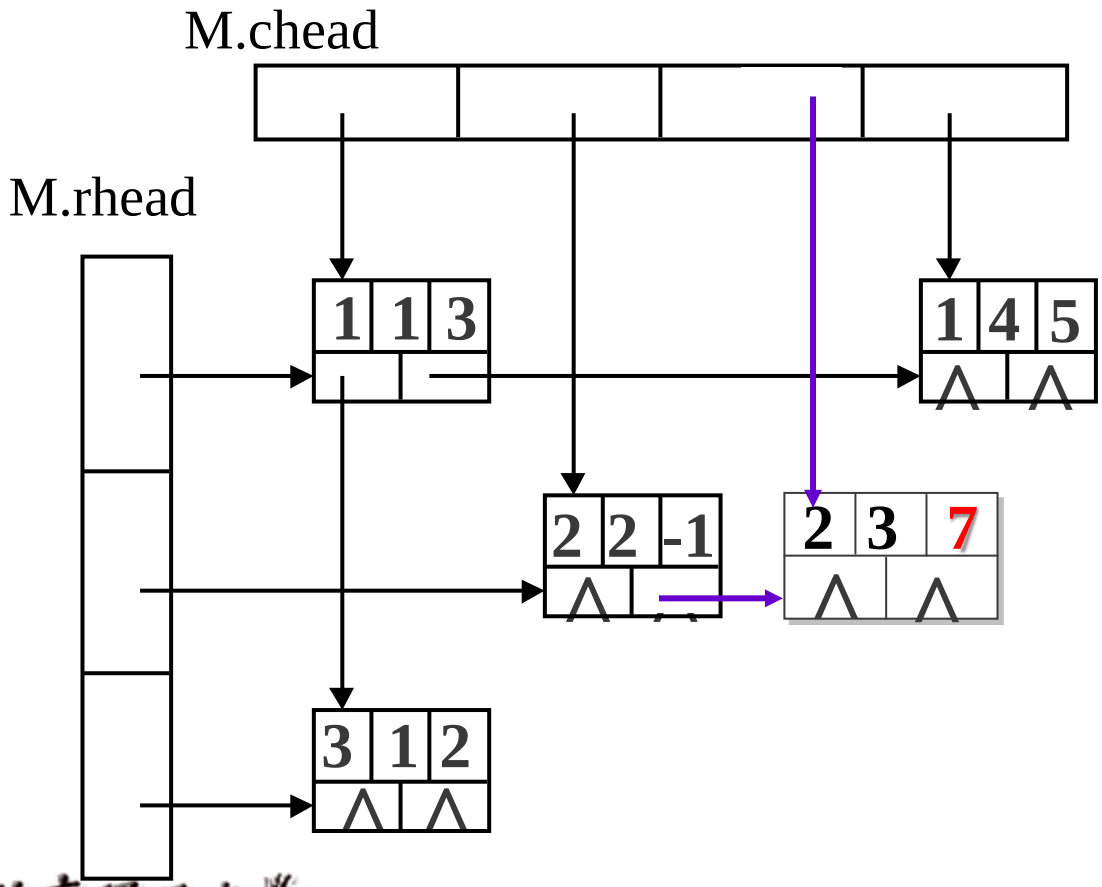



```
typedef struct OLNode {
   int   i, j;      // 非零元的行下标和列下标
   ElemType  e;                 //非零元值
   STRUCT OLNode *right,*down
}OLNode, OLink;
```

```
typedef struct {
    OLink *rhead,*chead; //行、列表头指针数组
    int    Rows,Cols, tu;  //行数、列数和非零元个数
 }CrossList;
```

十字链表的操作

- 类似线性表
- 区别：对横纵两个方向的链表同时操作



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

$$\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 7 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

广义表

- **广义表(generalized list)：是一种不同构的数据结构**

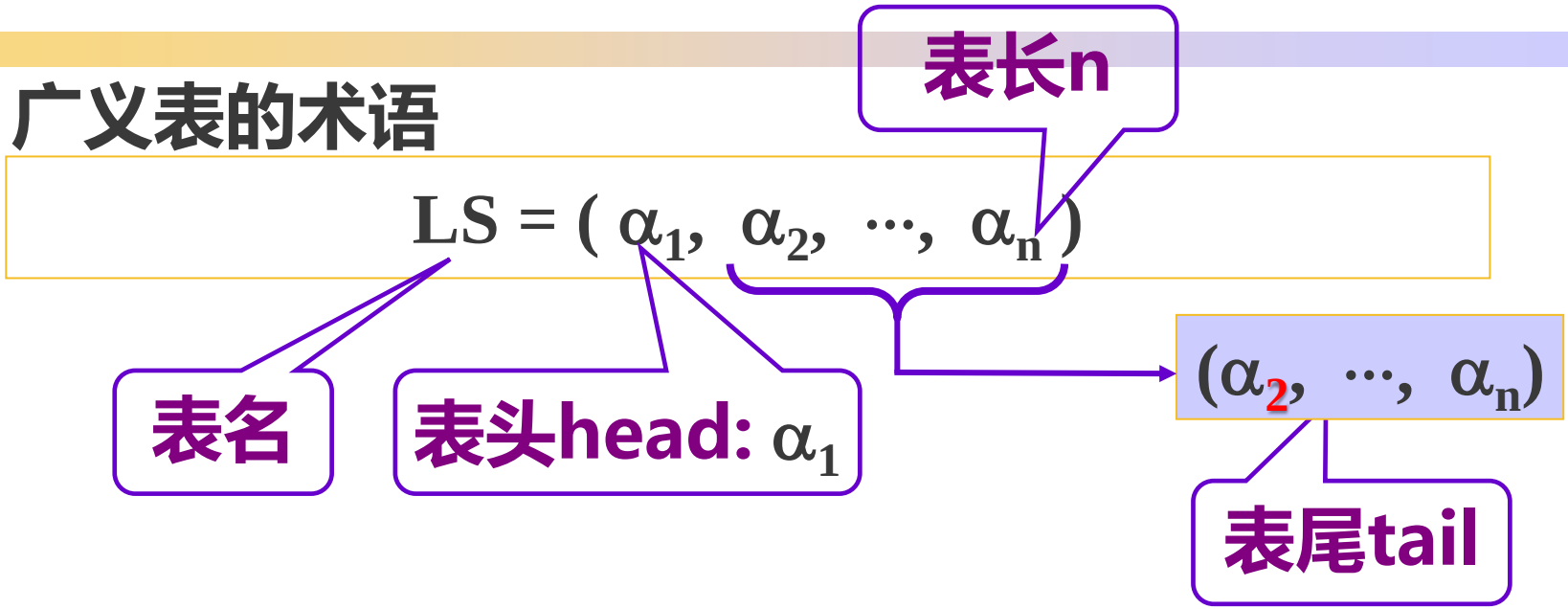
$$LS = (\alpha_1, \alpha_2, \cdots, \alpha_n)$$

其中：α_i 或为原子(atom)或为广义表。

- **广义表的基本性质**
  - 广义表的定义是一个递归定义，在描述广义表时又用到了广义表；
  - 在线性表中数据元素是单个元素，而在广义表中，元素可以是单个元素称为原子(atom)，也可以是广义表，称为广义表的子表(sublist)；
  - 当每个元素均为原子且类型相同时，就是线性表。

广义表的定义

- 广义表的术语



表头：LS的第一个元素称为表头

表尾：其余元素组成的表称为LS的表尾

表长：为最外层包含元素个数（不含括号和分隔符）

深度：所含括号的重数（原子的深度为0，“空表”的深度为1）。

广义表的定义

- 例如:
  - ¶ A = ( ) 空表　　长度:0；深度:1
  - ¶ F = (d, (e))　　长度:2；深度:2
  - ¶ D = ((a,(b,c)), F)　　长度:2；深度:3
  - ¶ C = (A, D, F)　　长度:3；深度:4

共享表

- ¶ B = (a, B) = (a, (a, (a, … ,))) 递归定义

⇔B = (a, (a, (a, … ,)))

递归表

长度: 2；深度:无限 ∞

广义表的结构特点

- **广义表的结构特点(五个)**
  - ¶ 1) 广义表中的数据元素有相对次序;
  - ¶ 2) 广义表可以共享;
  - ¶ 3) 广义表可以是一个递归的表;
  - ¶ 4)广义表是一个多层次的线性结构(图形化表示);

如:

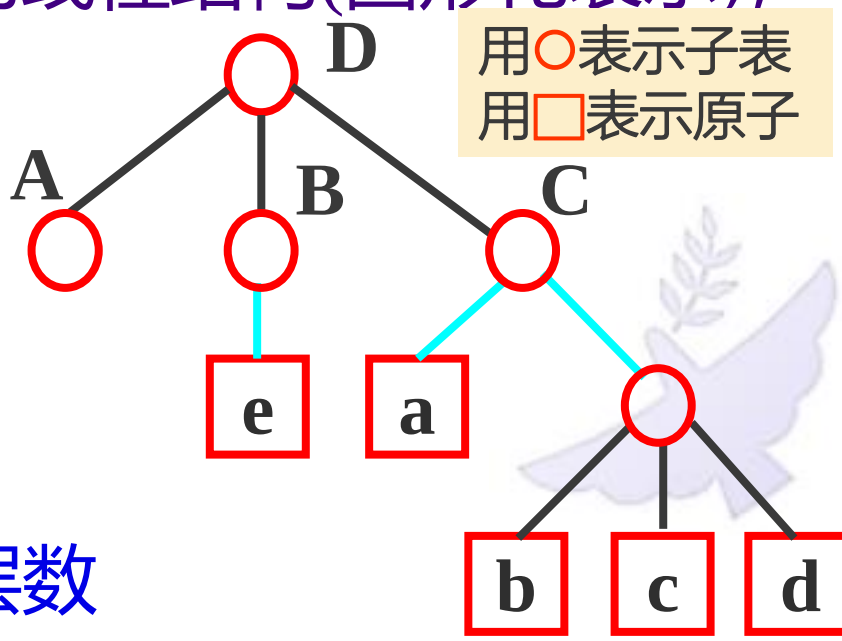
D = (A, B, C)

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

表长: 3

深度: 3 = 括号的重数

= ○ 结点的层数



广义表的结构特点

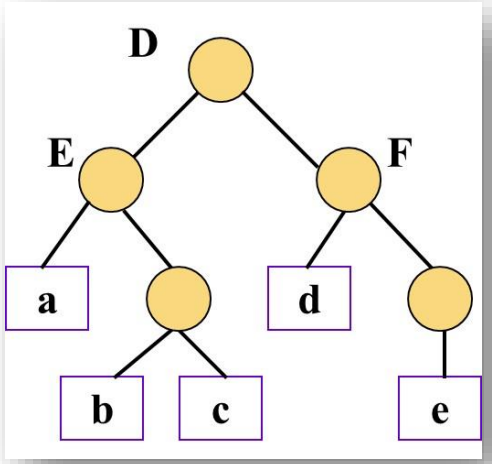
第一个元素

其余元素构成的广义表

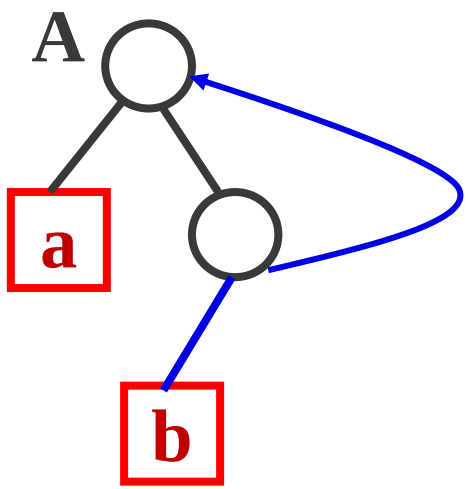
- 5）任何一个非空广义表LS = (α1, α2, …, αn)均可分解为
  - **表头：Head(LS) = α1**
  - **表尾：Tail(LS) = (α2, …, αn)**
- **例如：D = (E, F) = ((a, (b, c)), F)**
- **Head(D) = E　　Tail(D) = (F)**
- **Head(E) = a　　Tail(E) = ((b, c))**
- **Head(((b, c))) = (b, c)　Tail(((b, c))) = ( )**
- **Head((b, c)) = b　Tail((b, c)) = (c)**
- **Head((c)) = c　　Tail((c)) = ( )**

练习

- 1. 广义表形如：D = (E, F) = ((a, (b, c))，F)，则
  - a= ?
    - ✓ a=Head(Head(D))
  - b= ?
    - ✓ b=Head (Head(Tail(Head(D))))



2. A = (a , (b, A))



表长：？ 2

深度：？ ∞

广义表结构特点小结

特点

- 有次序　一个直接前驱和一个直接后继
- 有长度　＝表中元素个数
- 有深度　＝表中括号的重数
- 可递归　自己可以作为自己的子表
- 可共享　可以为其他广义表所共享

( )

A

空表

(*u*, *v*)

B

u *v*

线性表

(*a*, (*x*, *y*, *z*))

C

a

x *y* *z*

纯表

(*B*, *C*, *A*)

D

B C A

u *v* *a*

x *y* *z*

E

B D

u *v*

C A

a

x *y* *z*

再入表

(*B*, *D*)

F

d

递归表

(*d*, *F*)

广义表的表示方法

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为某个数据对象集合}

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);

基本操作

- 状态函数

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

- 遍历

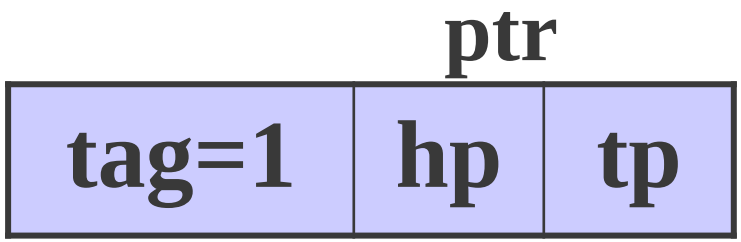
Traverse_GL(L, Visit());

广义表的存储结构

1. 广义表的头尾表示法

- 头尾链表结构

表结点:

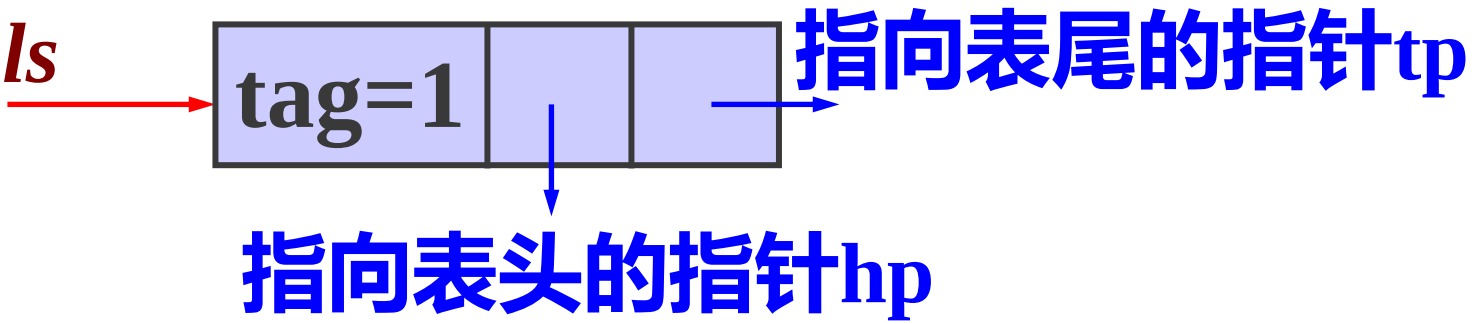


原子结点:

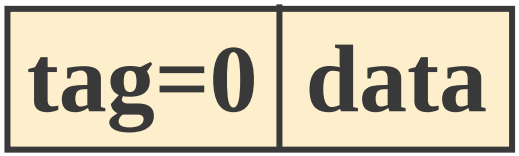
广义表的头尾链表存储表示_头尾表示法

空表　　*ls=NULL*

非空表　*ls*



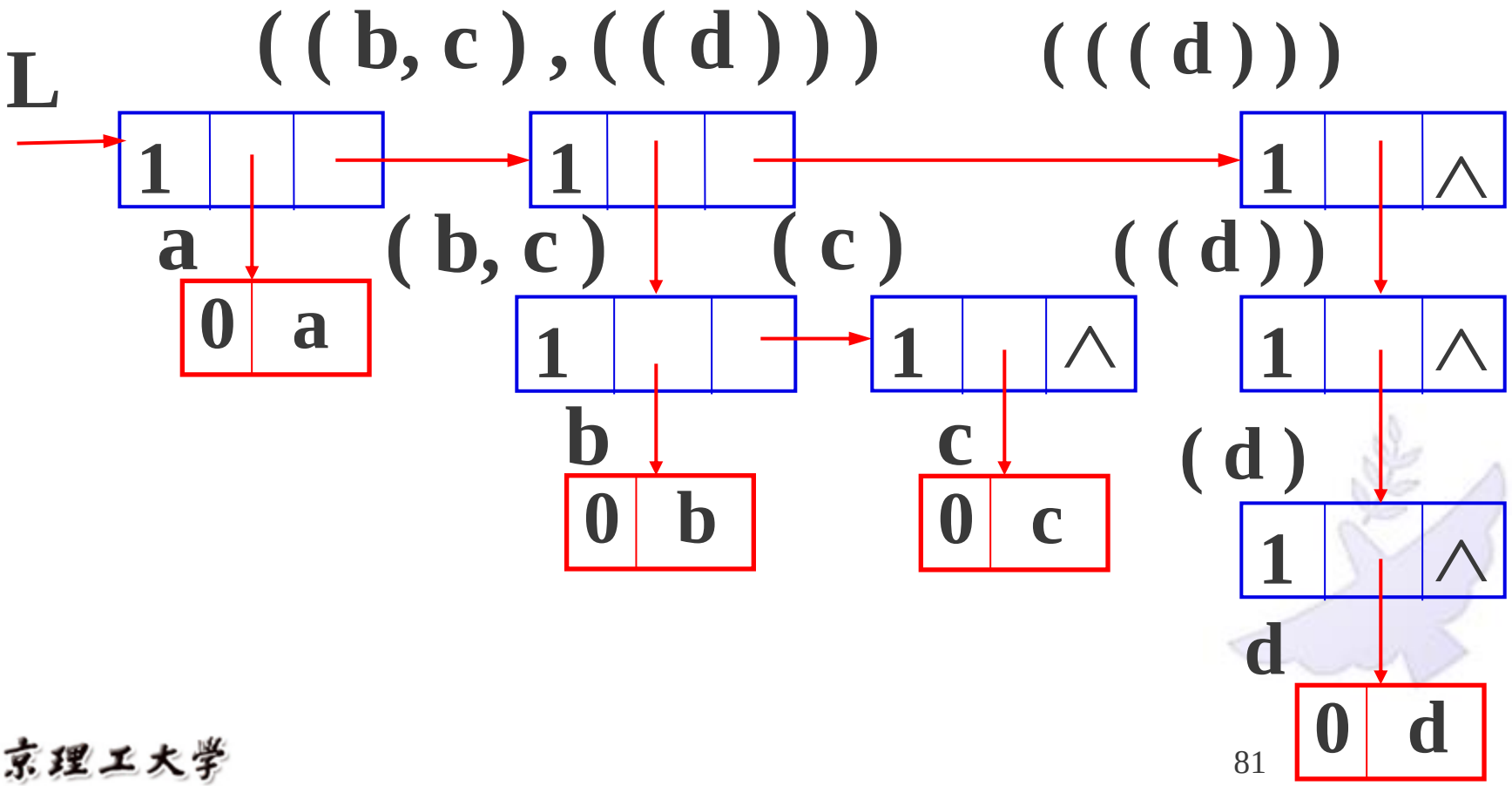
若表头为原子，则为



否则，依次类推。

广义表的头尾链表存储表示_头尾表示法

L = (a , (b, c) , ((d)))



广义表的头尾链表存储表示

表结点

tag=1	hp	tp

原子结点

Tag=0	data

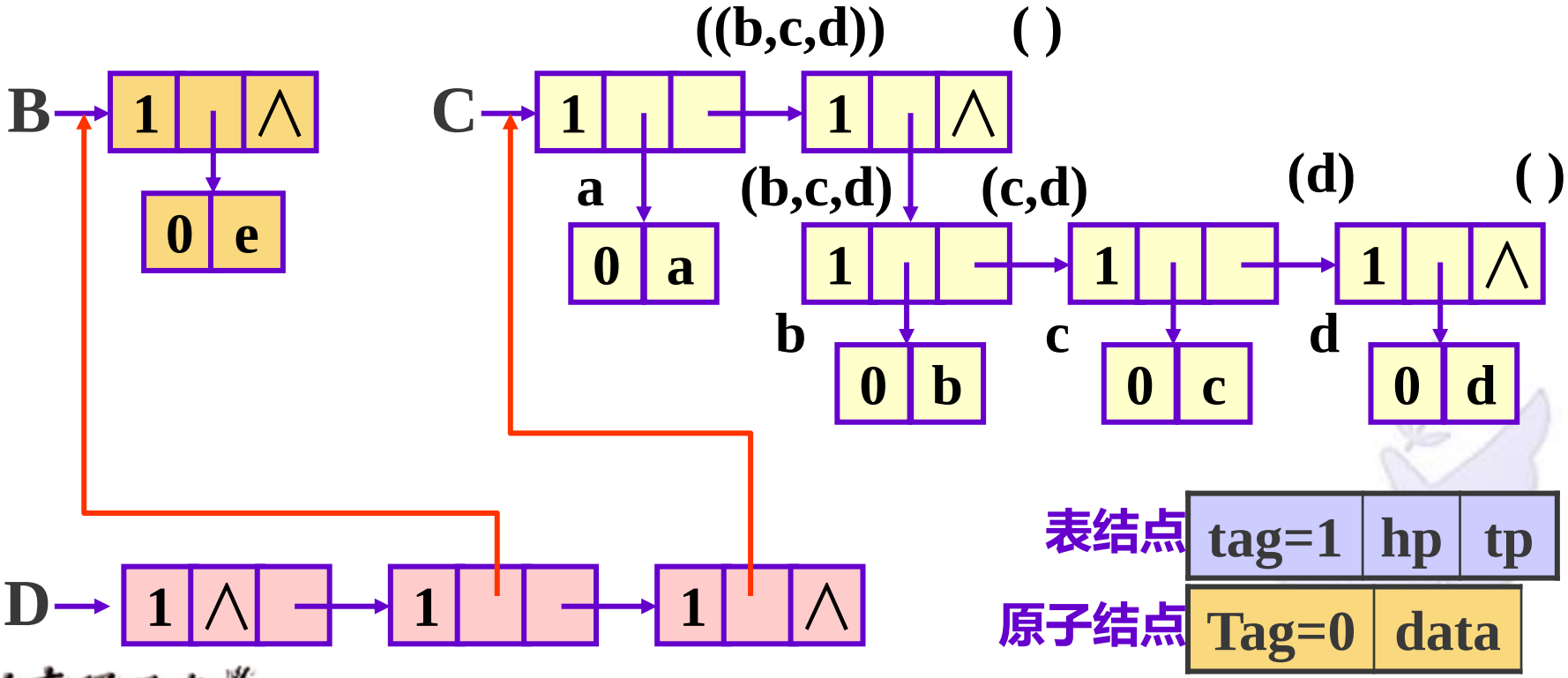
```
typedef enum {ATOM, LIST} ElemTag;
 // ATOM==0:原子, LIST==1:子表
```

```
typedef struct GLNode {
   ElemTag  tag;                  // 标志域
   union{
     AtomType  atom;       // 原子结点的数据域
     struct {
       struct GLNode *hp, *tp;
     } ptr;                                  //子表
              // 表结点的指针域: ptr.hp指表头, ptr.tp指表尾
   };
} *GList
```

广义表的头尾链表存储表示_头尾表示法

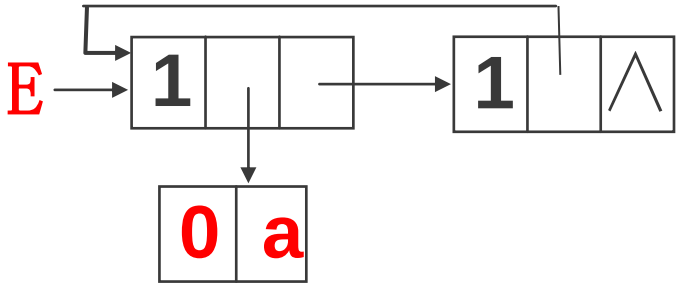
A=( ) B = (e) C = (a, (b, c, d)) D = (A, B, C)

A=NULL



广义表的头尾链表存储表示_头尾表示法

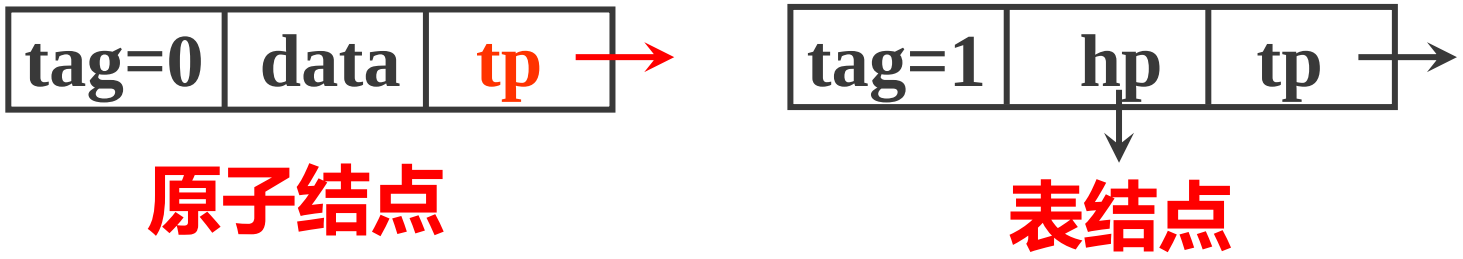
$E = (a, E)$



广义表的存储结构

2. 广义表的扩展线性链表表示

- 两种结点：原子结点和表结点。

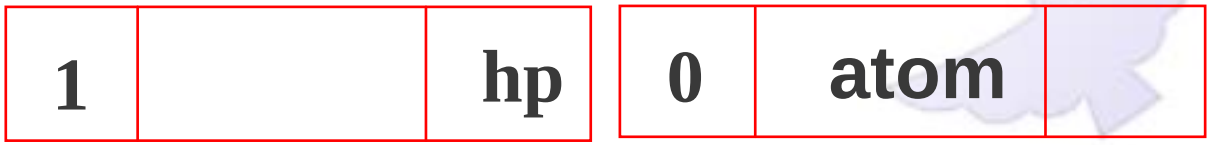


- **原子结点**标志 tag=0，data存放元素的值，tp 指向同一层下一个表元素结点的指针；
- **表结点**标志 tag=1，hp 存放指向子表的指针 hp，tp 与表元素结点 tp 的含义相同。

广义表的存储结构_扩展线性链表表示

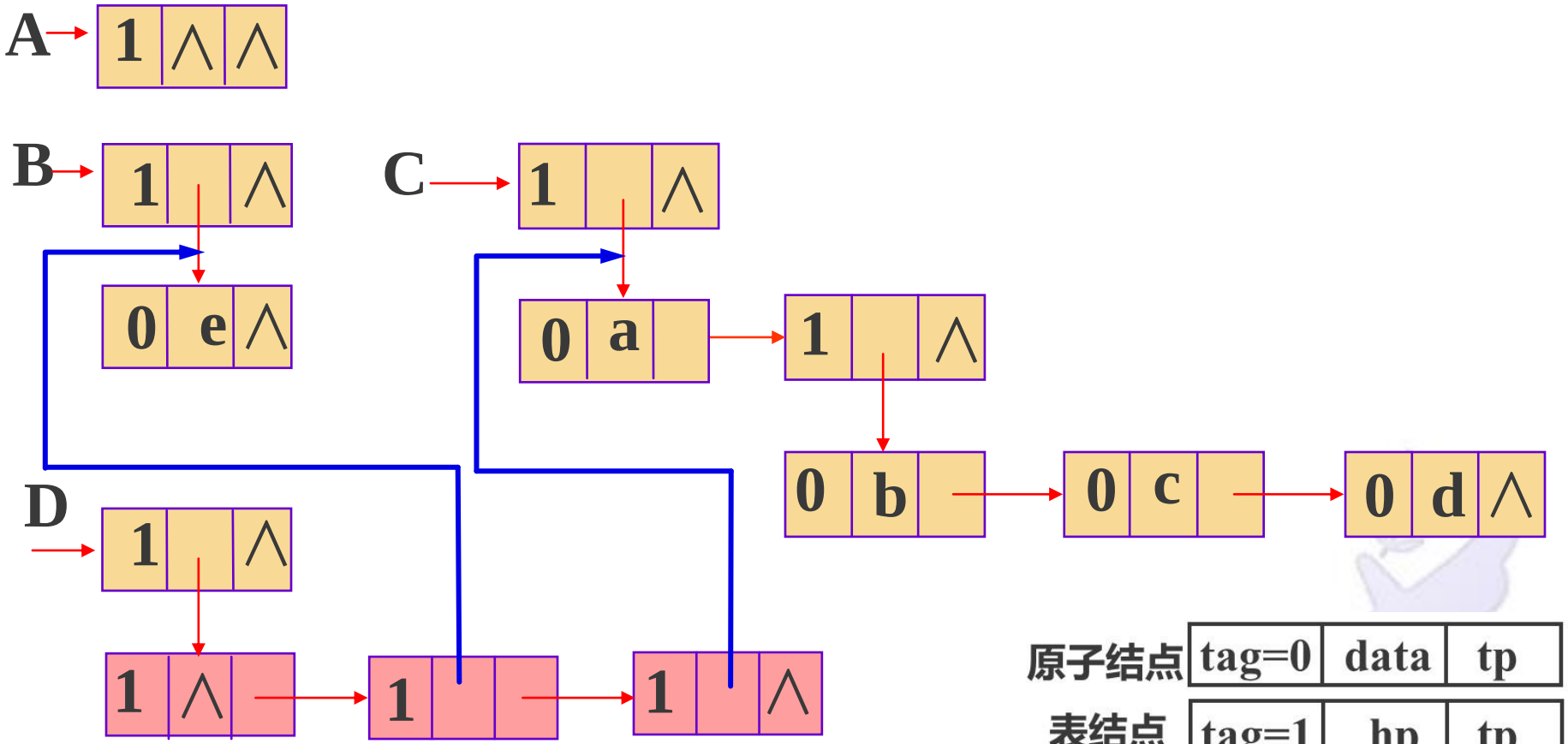
```
typedef enum {ATOM, LIST} ElemTag;
 // ATOM==0:原子, LIST==1:子表
```

```
typedef struct GLNode {
   ElemTag  tag;                 // 标志域
   union{
     AtomType  atom;       // 原子结点的数据域
     struct GLNode *hp; // 表结点的表头指针
     };
    struct GLNode *tp     //指向下一个元素结点
} *GList
```



广义表的存储结构_扩展线性链表表示

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



广义表的存储结构_扩展线性链表表示

◆**优点:** 每个广义表都有一个起到“头结点”作用的表结点，即使空表，也有一个表结点。

◆**缺点:** 每个对子表引用的指针没有指向子表的“头结点”，而是指向了广义表的表头元素结点（是第一个表元素结点），所以造成了对表头元素结点插入或删除时的困难。

- 如果某表头元素为多个表结点共享，插入或删除它后如何找到所有共享它的结点？必须确认指向这个结点的所有指针，也就是检查有过的广义表。

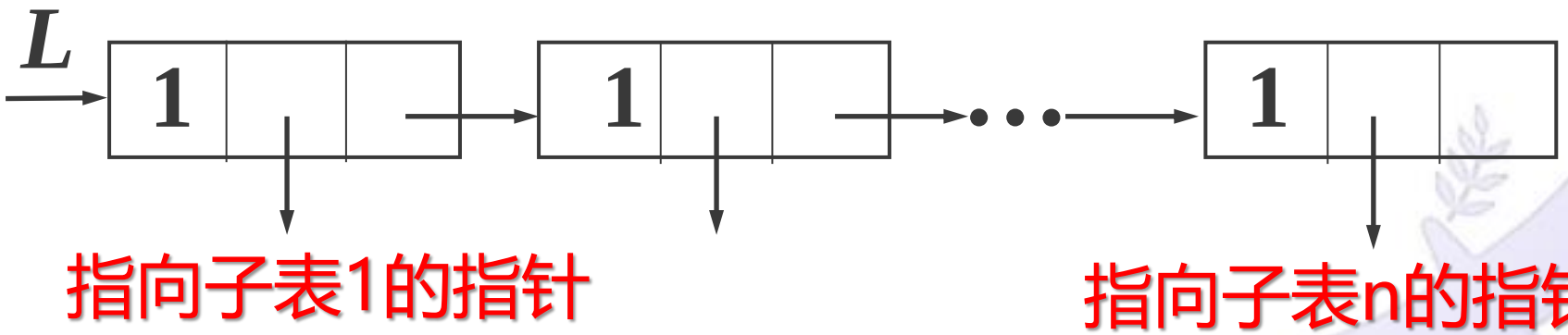
广义表的存储结构

3. 子表链表存储（层次链）

广义表：子表1 + 子表2 + … + 子表n

广义表： $L = (\alpha_1, \alpha_2, \ldots, \alpha_n)$

子表： $\alpha_1\ \alpha_2\ \alpha_3 \ldots \alpha_n$



广义表的存储结构_层次表示

1. 结点类型 tag

0: 表头结点；1: 原子结点；2: 子表结点

2. 信息info：

tag=0时，存放引用计数ref；

tag=1时，存放数据值value；

tag=2 时， 存放指向子表头结点的指针hlink。

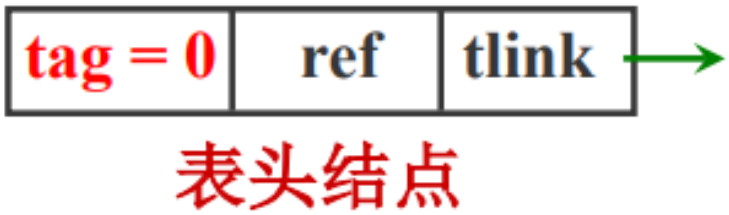
3. 尾指针tlink：

tag=0时，指向该表第一个结点；

tag≠0 时，指向同一层下一个结点。

广义表的存储结构_层次表示

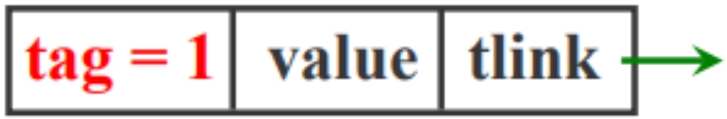
三种结点：表头结点；原子结点；子表结点



表头结点

ref：该表的引用计数

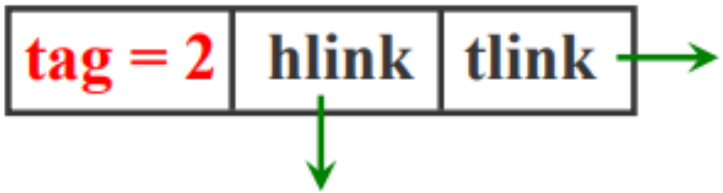
tlink：指向表的第一个结点



原子结点

value：数据

tlink：指向同层下一个结点



子表结点

hlink：指向子表的指针

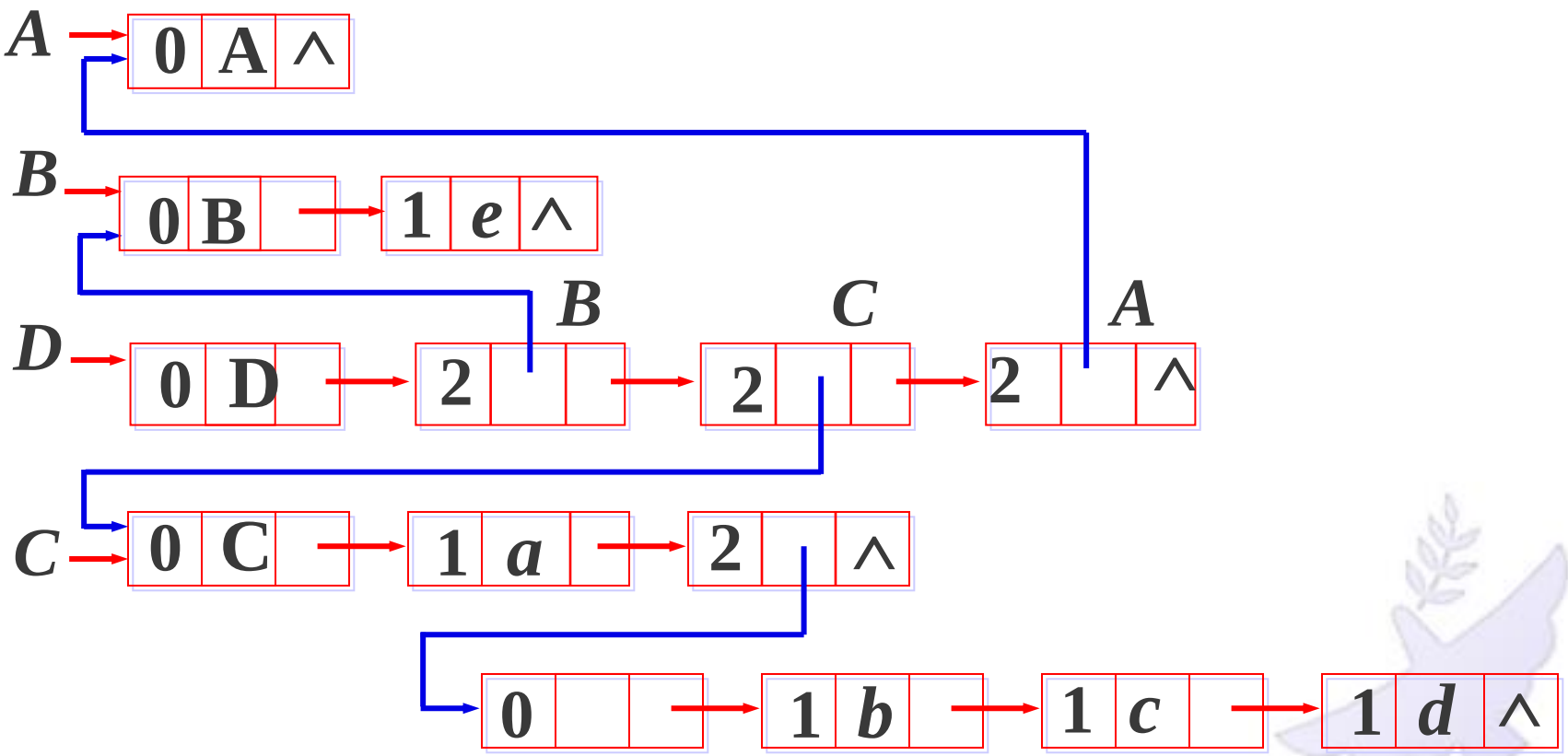
tlink：指向同层下一个结点

广义表的存储结构_层次表示

```
typedef struct node {          //广义表结点定义
    int tag;
          // 0：头结点，1：原子结点，2：子表结点
    struct node * tlink;   // 指向同层下一结点的指针
    union {                    // 3个域叠压使用同一空间
        char name;          // tag=0，存放表名
        char value;         // tag=1，存放数据
        struct node * hlink;
                            // tag=2，存放指向子表的指针
    } info;
} GenListNode, *GenList;
```

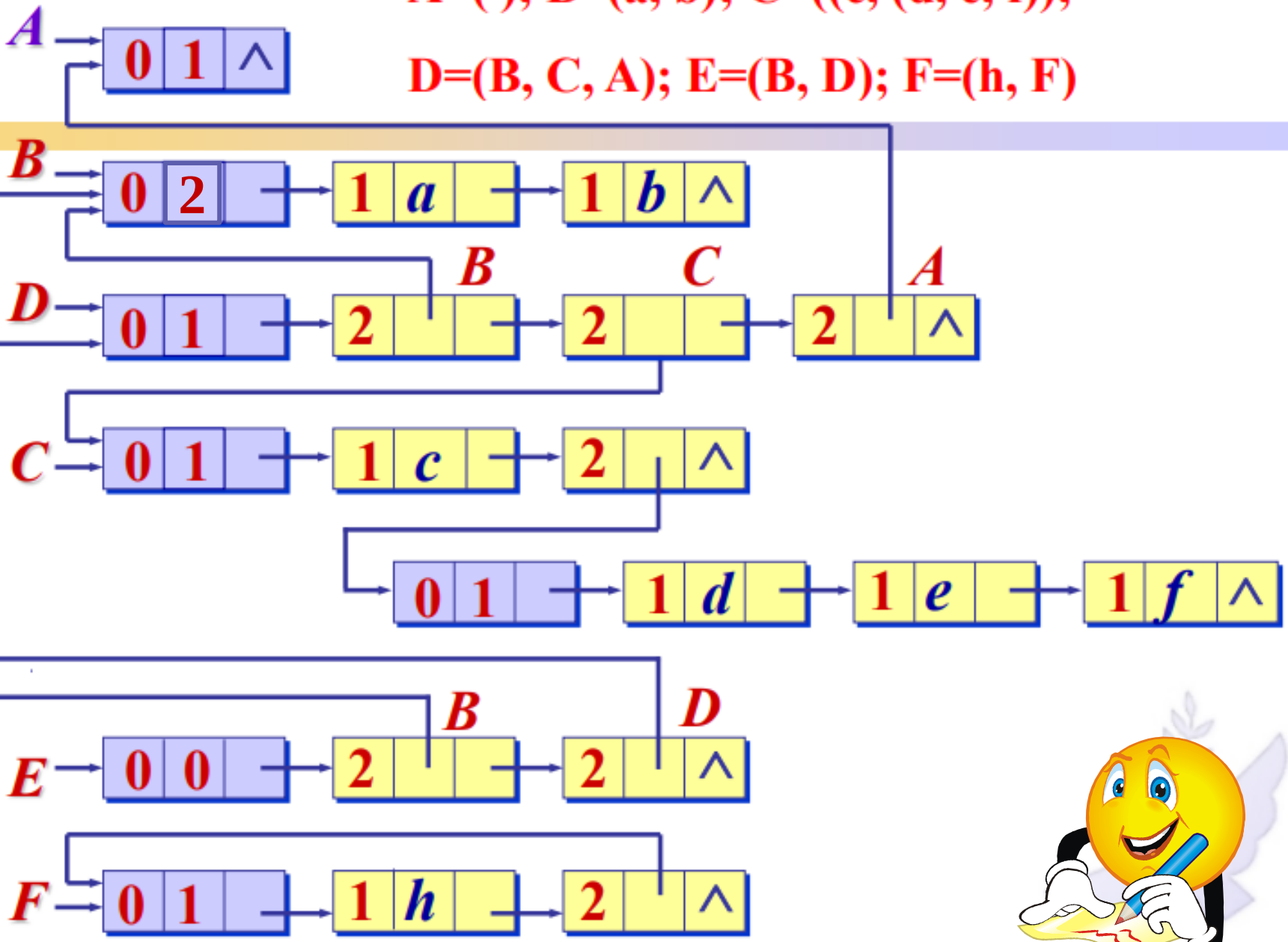
广义表的存储结构_层次表示

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



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

D=(B, C, A); E=(B, D); F=(h, F)



广义表的存储结构_层次表示

- 特点：
  - 表示更简洁
  - 有表头结点，所以插入删除操作简单

广义表操作的实现

回顾

- 广义表从结构上可以分解成
  - 广义表 = 表头 + 表尾
  - 广义表 = 子表1 + 子表2 + … + 子表n

由此出发，可以实现广义表的存储和操作。

回顾-基本操作

- 结构的创建和销毁

  InitGList(&L);　　DestroyGList(&L);

  CreateGList(&L, S);　CopyGList(&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);

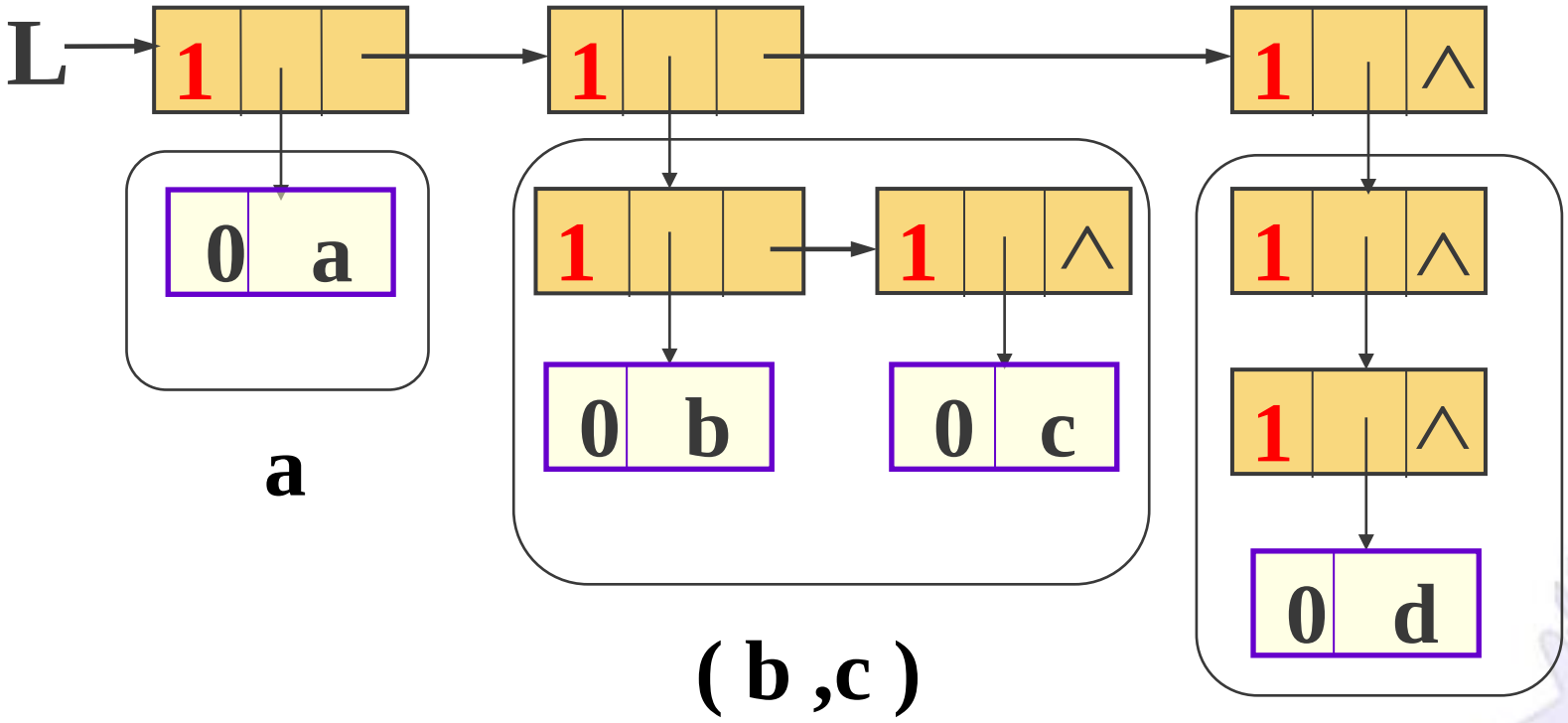
  DeleteFirst_GL(&L, &e);

- 遍历

  Traverse_GL(L, Visit());

广义表操作的实现

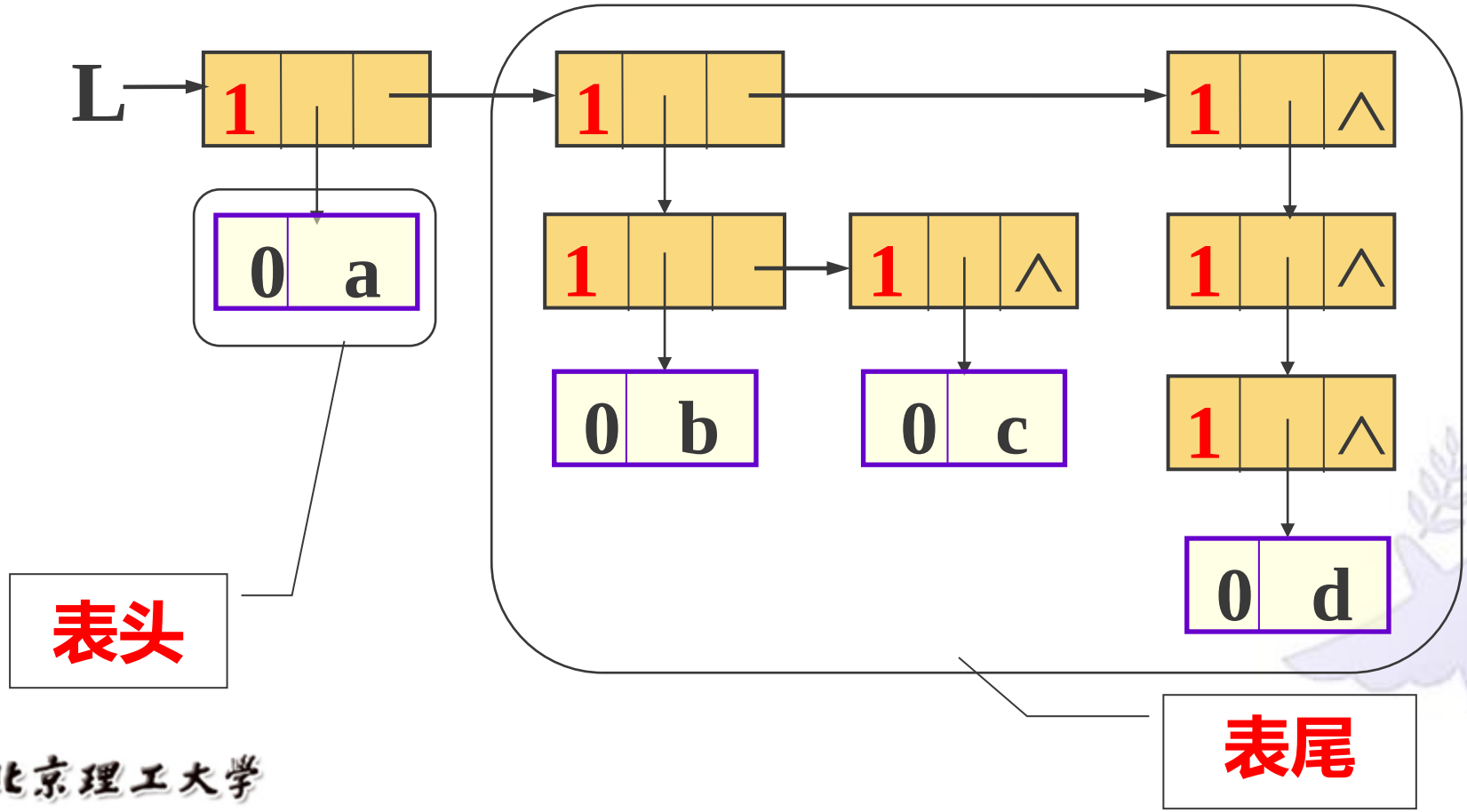
L = (a , (b ,c) , ((d)))



回顾：头尾链表结构

广义表操作的实现

L = (a , (b ,c) , ((d)))



广义表操作的实现

- 1 求广义表的深度 GListDepth(L)
- 2 复制广义表 CopyGList(T, L)
- 3 建立广义表CreateGList(&L, str[ ])
- 4 删除广义表中所有元素为 x 的原子结点
- ……

1. 求广义表的深度

广义表L的深度 = 广义表L中括号重数

GListDepth(L) = 1 + MAX(GListDepth(L的元素))

例 L = (a , (b ,c) , ((d)))

GListDepth(a) = 0 原子

GListDepth((b,c)) = 1 线性表

GListDepth(((d))) = 2

GListDepth(L) = 3

1. 求广义表的深度

GListDepth(L)的递归描述

分解： 将广义表分解成 n 个子表，分别求得每个子表的深度。

组合： 广义表的深度 = max{子表的深度}+1

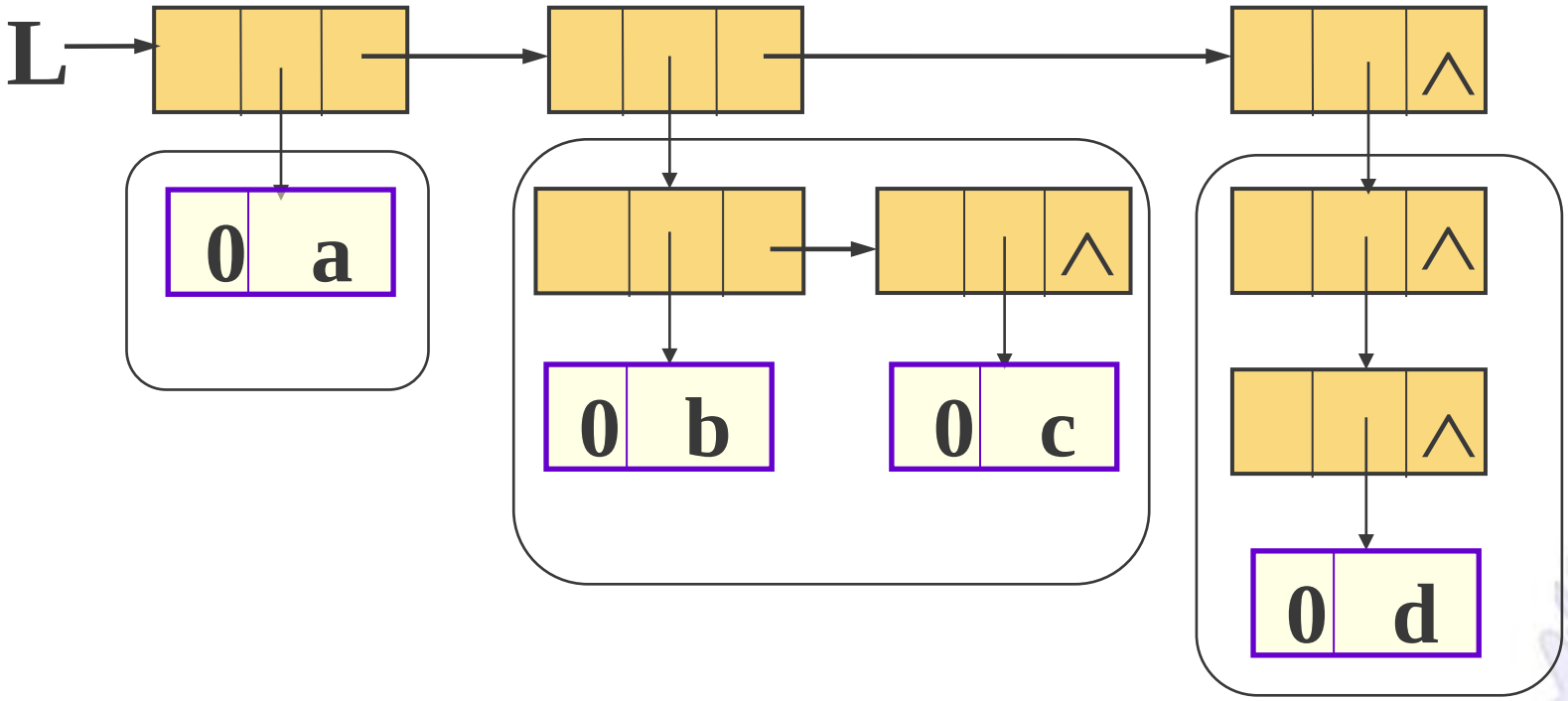
直接求解

空表：深度 = 1

原子：深度 = 0

1. 求广义表的深度

L = (a , (b ,c) , ((d))) 的深度



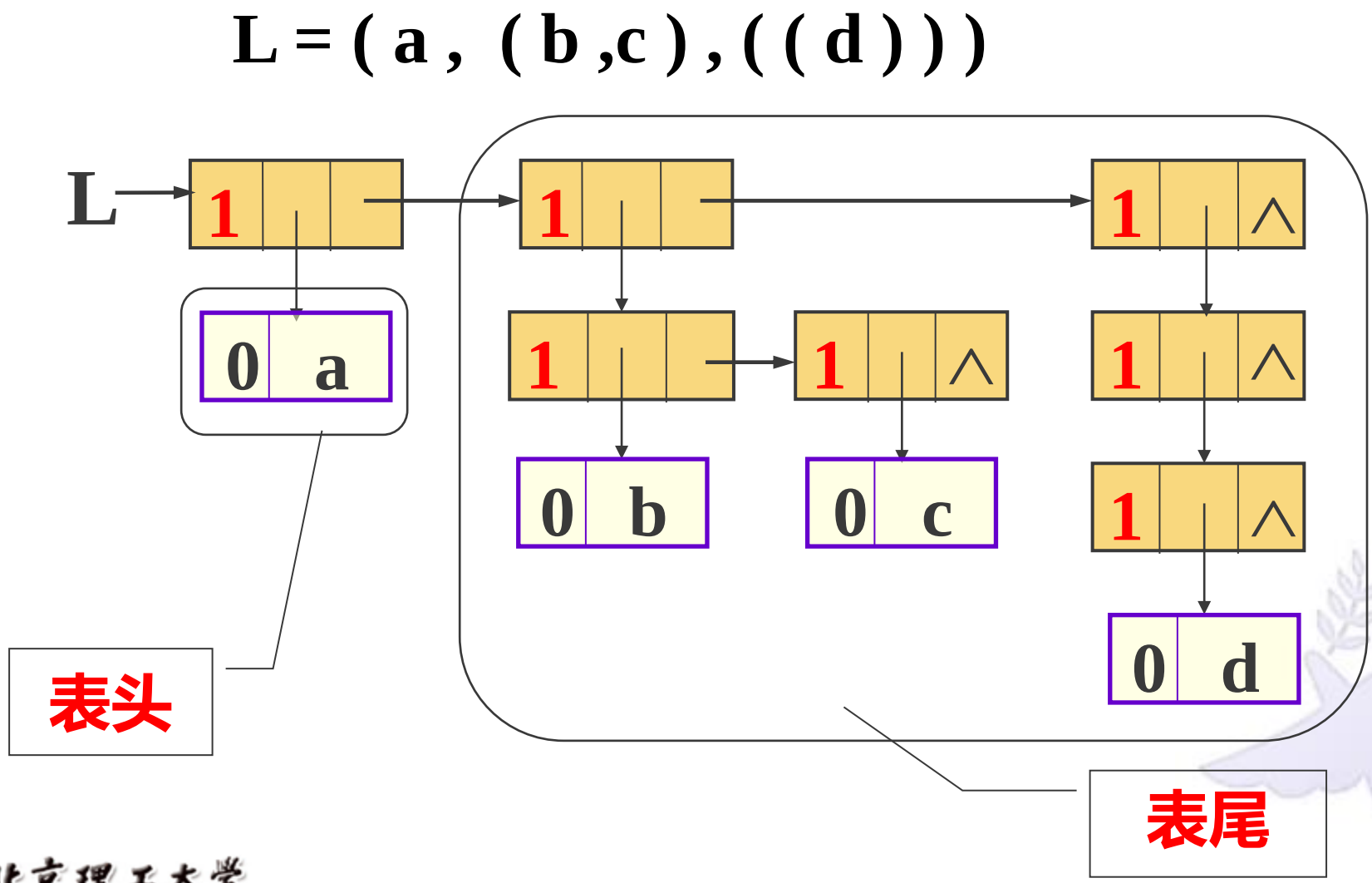
1. 求广义表的深度

表结点: | tag=1 | hp | tp |

原子结点: | Tag=0 | data |

```
int GListDepth( GList L )
 { //采用头尾链表存储结构，求广义表L的深度
      if ( !L )  return 1;                          // 空表深度1
      if ( L->tag==ATOM ) return 0;   // 原子深度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
```

2. 复制广义表 CopyGList(T,L)



```
void GListCopy( GList &T, GList L )
{  /*由广义表L复制得到广义表T */
    if ( !L ) T=NULL;          // 复制空表
    else {
        T=(GList) malloc( sizeof(GLNode) );          // 建表结点
        if ( !T ) exit(OVERFLOW);
        T->tag = L->tag;                              //复制标志项
        if ( L->tag==ATOM )  T->data = L->data; // 原子
        else{
            GListCopy( T->ptr.hp, L->ptr.hp );  // 复制hp
            GListCopy( T->ptr.tp, L->ptr.tp );    // 复制tp
        }//if ( L->tag==ATOM ) else
     } //if ( !L ) else
}// GListCopy
```

p=T->ptr.hp；GListCopy(p, L->ptr.hp)？？

3. 建立广义表　　L = (a , (b ,c) , ((d)))

输入：字符串 $(\alpha_1, \alpha_2, \cdots, \alpha_n)$

结果: 建立广义表的头尾链表

分解： 将广义表分解成 n个子表$\alpha_1,\alpha_2,\cdots,\alpha_n$，分别建立 $\alpha_1, \alpha_2, \cdots, \alpha_n$ 对应的子表。

组合： 将 *n* 个子表组合成一个广义表

直接求解：

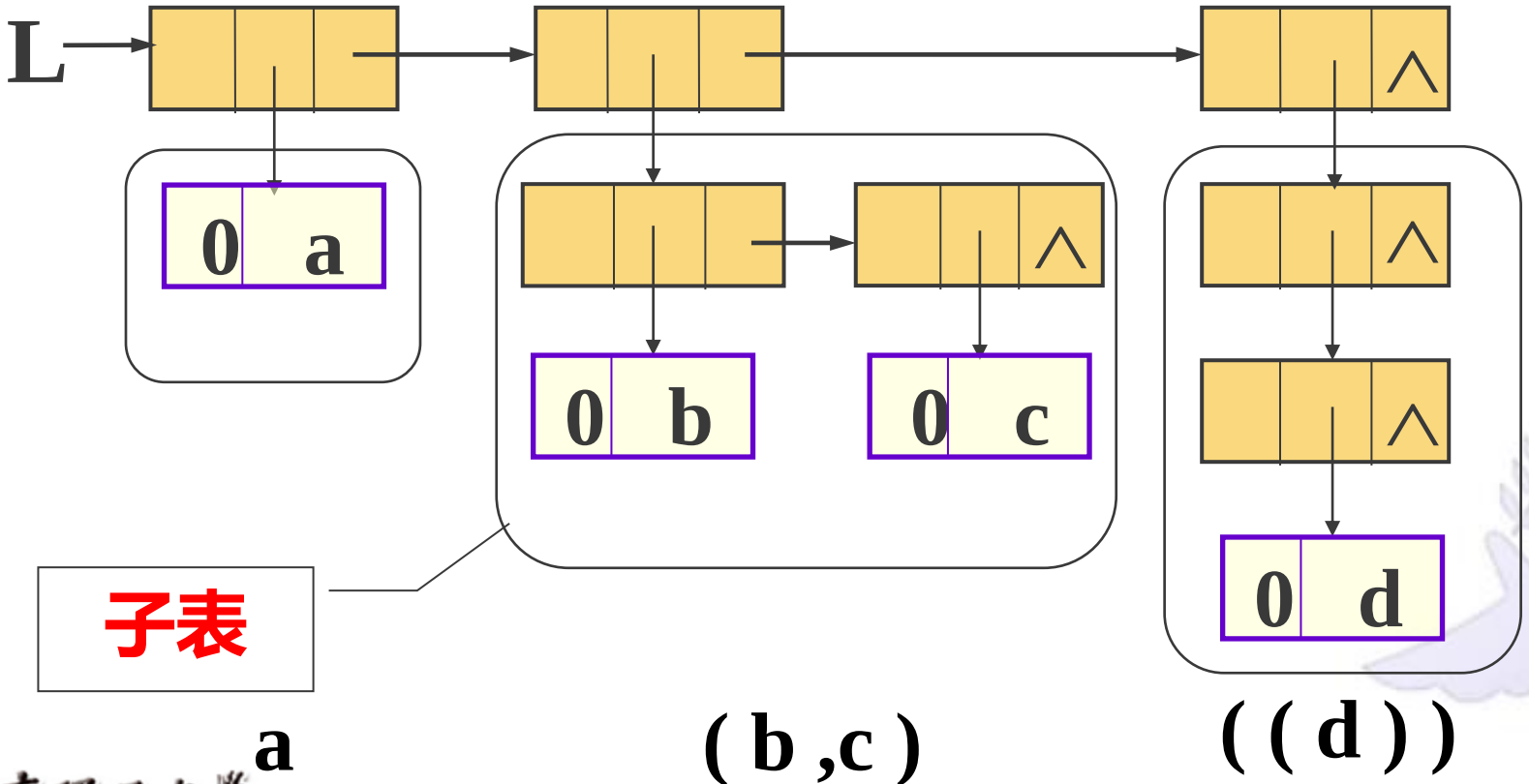
空表: NULL

原子: 建立原子结点

3. 建立广义表

子表和广义表的关系　　**L = (a , (b ,c) , ((d)))**

相邻两个子表之间的关系



```
void CreateGList( GList &L, char str[ ] ){
  if ( strcmp( str,"()" )==0)  L=NULL;   //空表
  else
  { if (strlen(str)==1) {                          //原子结点
        L=(GList)malloc(sizeof(GLNode));
        L->tag=ATOM; L->atom=str[0];
    }
    Else //非空表，非原子节点，建表结点
    {     L=(GList)malloc(sizeof(GLNode));
          L->tag=LIST;      p=L;
          SubString(sub,str,2,strlen(str)-2); //脱外层括号
          由sub中所含n个子串建立n个子表;
    }
  }  // else
} // CreateGList
```

L = (a , (b ,c) , ((d)))

3. 建立广义表

a , (b ,c) , ((d))

```
do {  //由sub中所含n个子串建立n个子表
    sever( sub, hsub );  //分离出子表串hsub=αi
    CreateGList( p->ptr.hp, hsub );
                          // 建hsub对应的子表
    if ( !strempty(sub) )
    {     //建下一个子表的表结点
       p->ptr.tp = (GList)malloc(sizeof(GLNode));
       p = p->ptr.tp;
       p->tag = LIST;
     }  //if
} while(!strempty(sub));
p->ptr.tp = NULL;   //最后一个子表的表结点
```

3. 建立广义表-层次表示法

- 假设采用广义表层次链表表示法
- 输入：广义表字符串s，如s =“A(c, H(d, e, f)) ”;
- 输出：广义表层次链表表示法
- 假设：
  - 建立无共享、无递归的广义表
  - 每一个子表都有名字，无语法错误
- 基本思想：从s中取得一个字符，检测其内容
  - 大写字母
  - 左括号、右括号
  - 小写字母

3. 建立广义表-层次表示法

- 基本思想：从s中取得一个字符，检测其内容

1. 大写字母：建表头结点，保存表名。
2. 左括号：因为输入无语法错误，所以表名后一定是左括号，递归建广义表
3. 小写字母：建立原子结点
4. 右括号：表示子表结束，退出递归

- 自己阅读算法4-20(P146)

4. 删除广义表中所有元素为 x 的原子结点

分析：

比较广义表和线性表的结构特点。

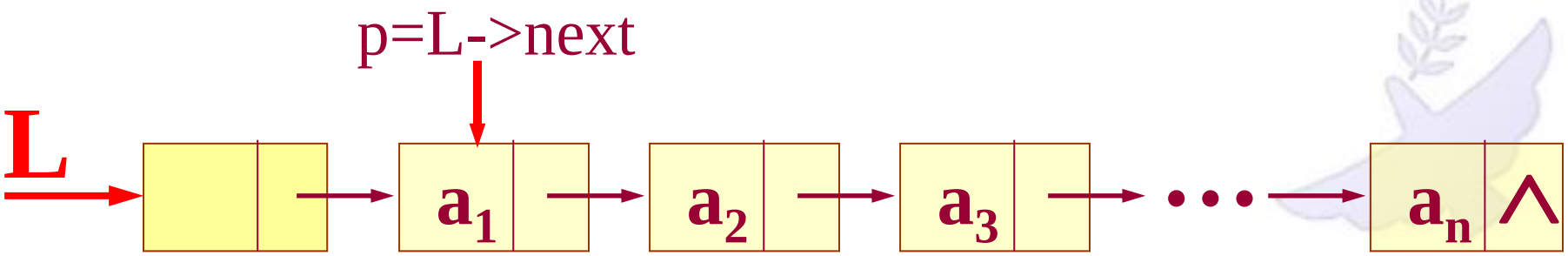
- **相似处：都是链表结构。**
- **不同处：**

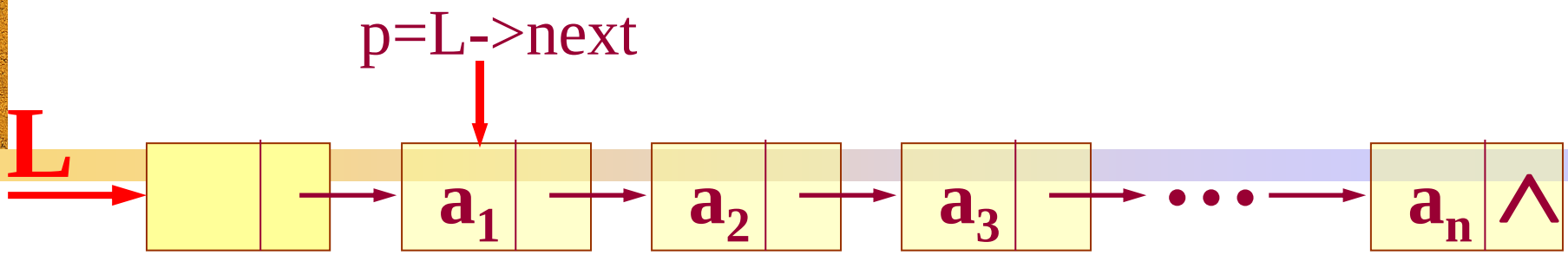
1)广义表的数据元素可能还是个广义表；

2)删除时，不仅要删除原子结点，还需要删除相应的表结点。

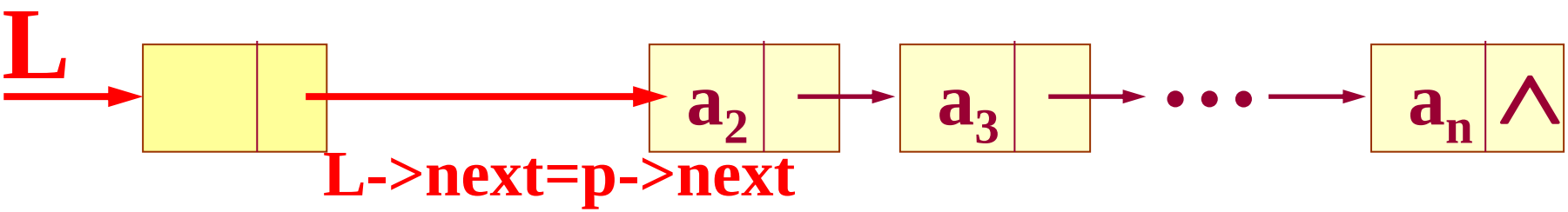
4 删除广义表中所有元素为 x 的原子结点

- 删除**单链表**中所有值为x 的数据元素
- 基本思想：
  - 1) 单链表是一种顺序结构，必须从第一个结点起，逐个检查每个结点的数据元素；
  - 2) 从另一角度看，链表又是一个递归结构
  - 若 L 是线性链表 ($a_1, a_2, \ldots, a_n$) 的头指针
  - 则 L->next是线性链表 ($a_2, \ldots, a_n$)的头指针。

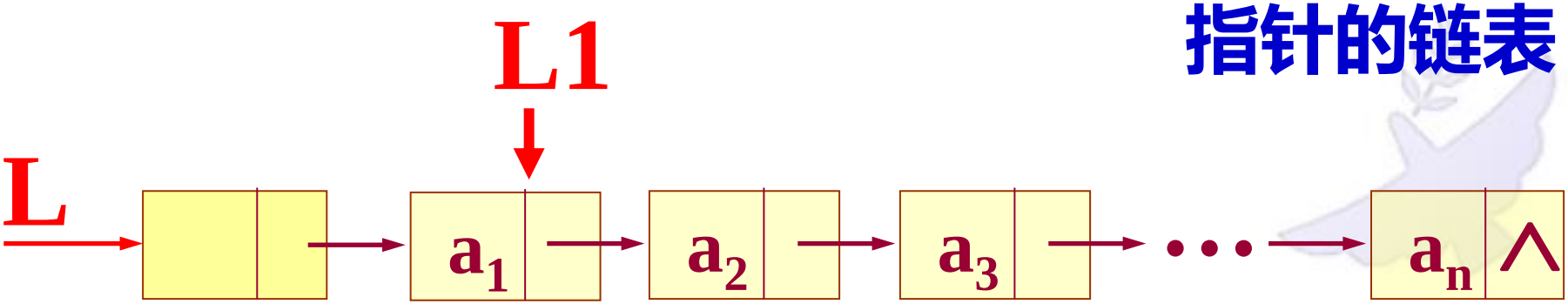




1) “a1=x”，则 L 仍为删除 x 后的链表头指针



2) “a1≠x”，则余下问题是考虑以 L->next 为头指针的链表



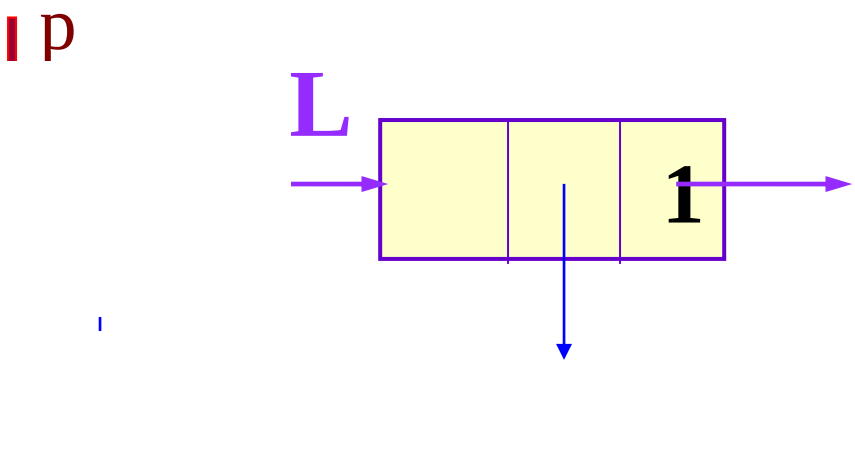
删除单链表中所有值为x 的数据元素

```
void delete(LinkList L, ElemType x) {
 // 删除以L为头指针的带头结点的单链表中
 // 所有值为x的数据元素
      if (L->next) {
          if (L->next->data==x) {
              p=L->next;  L->next=p->next;
              free(p);  delete(L, x);
          }
          else  delete(L->next, x);
       }
} // delete
```

删除广义表中所有元素为x的原子结点

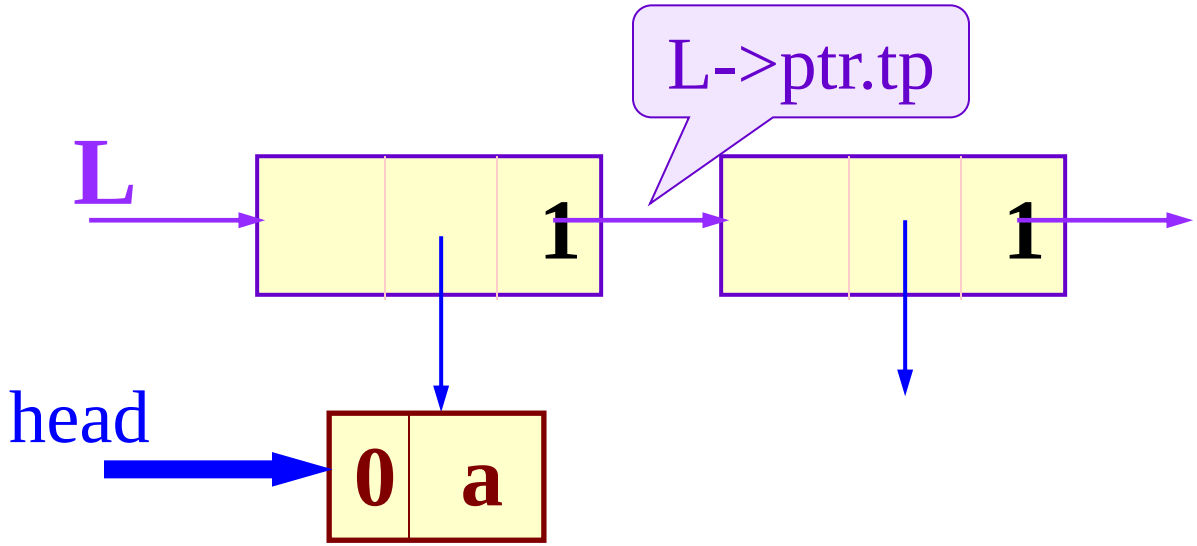
```
void Delete_GL(Glist&L, AtomType x) {
  //删除广义表L中所有值为x的原子结点
  if (L) {
     head = L->ptr.hp;  // 考察第一个子表
     if ((head->tag == Atom) && (head->atom == x))
       {    ……          }    // 删除原子项 x的情况
     elseif((head->tag == Atom) &&  (head->atom != x))
       {  ……          }// 不为x
     elseif (head->tag == LIST)
       {  ……          }// 子表
  }
} // Delete_GL
```

第一项是原子项，且等于x




```
if ((head->tag == Atom) &&  (head->atom == x))
{   p=L;    L = L->ptr.tp;     // 修改指针
    free(head);  free(p);       // 释放原子结点、表结点
    Delete_GL(L, x);            // 递归处理剩余表项
   }
```

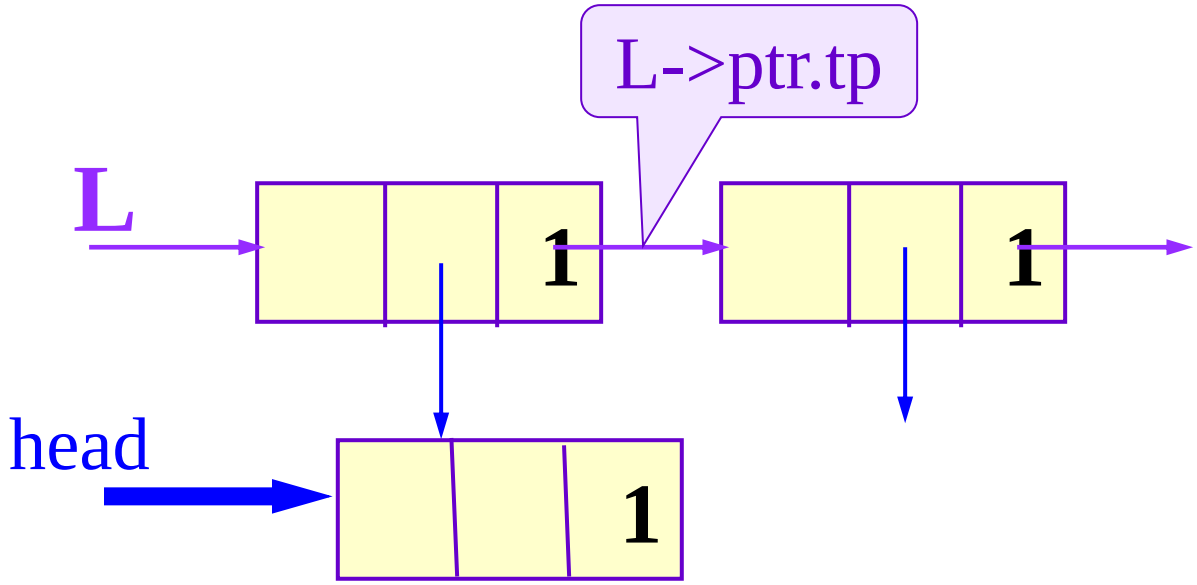
第一项是原子项，但不等于x




```
else if ((head->tag == Atom) &&  (head->atom != x))
```

```
//该项为原子项，不等于x
Delete_GL(L->ptr.tp, x);
        // 递归处理剩余表项
```

第一项不是原子项，是广义表




```
else if (head->tag == LIST) //该项为广义表
    Delete_GL(head, x); // 处理第一个广义表节点
    Delete_GL(L->ptr.tp, x); // 递归处理剩余表项
```

广义表操作的实现—删除广义表

```
void DestroyGList(GList &L) {
    if (!L) return;
    if (L->tag == LIST) {
        DestroyGList(L->ptr.hp);
        DestroyGList(L->ptr.tp);
    }
    free(L);
    L = NULL;
}//DestroyGList
```

广义表操作的实现—计算广义表长度

```
int GListLength(GList L) {
    if (L!=NULL)
        return (1 + GListLength(L->ptr.tp));
    else
        return 0;
}//GListLength
```

广义表操作的实现—计算广义表深度

```
int GListDepth(GList L) {
        if (!L) return 1;
        if (L->tag == ATOM)
                return 0;
        dh = GListDepth(L->ptr.hp) + 1;
        dt =  GListDepth(L->ptr.tp);
        return ((dh>dt)?dh:dt);
}//GListDepth
```

广义表操作的实现—插入节点

```
Status InsertFirst_GL(GList &L, GList e) {
//在广义表第一个节点前插入一个子表节点
    p =(GList)malloc(sizeof(GLNode));
    if (!p) exit(OVERFLOW);
    p->tag = LIST;          p->ptr.hp = e;
    p->ptr.tp = L;
    L = p;
    return OK;
}//InsertFirst_GL
```

广义表操作的实现—删除节点

```
Status DeleteFirst_GL( GList &L, GList &e )
{//删除广义表第一个节点
        if ( !L )   return ERROR;
        p = L;
        e = L->ptr.hp;
        L = L->ptr.tp;
        free( e );
        free( p );
        return OK;
}// DeleteFirst_GL
```

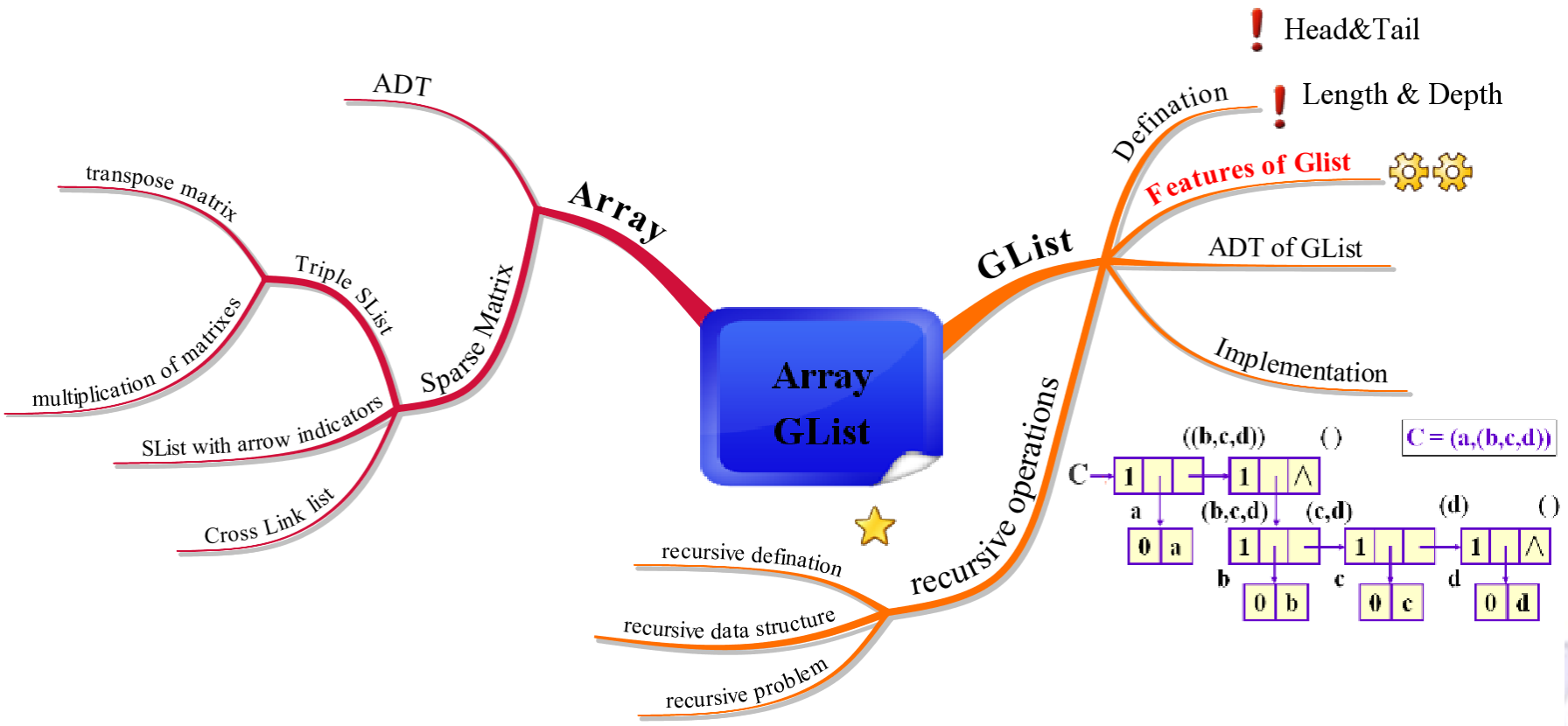
?

~~free(e);~~ →→DestroyGList(e)

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，学会对非空广义表进行分解的方法：即可将一个非空广义表分解为表头和表尾两部分。
5. 掌握广义表的递归算法设计。

Review


Array
ADT
Sparse Matrix
Triple SList
transpose matrix
multiplication of matrixes
SList with arrow indicators
Cross Link list
Array
GList
GList
Defination
Head&Tail
Length & Depth
Features of Glist
ADT of GList
Implementation
recursive operations
recursive defination
recursive data structure
recursive problem
C = (a,(b,c,d))
((b,c,d))
( )
C
a
(b,c,d)
(c,d)
(d)
( )
b
c
d

END OF CHAPTER IV

史树敏
计算机学院