第四章 数组和广义表

高春晓

本章内容

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

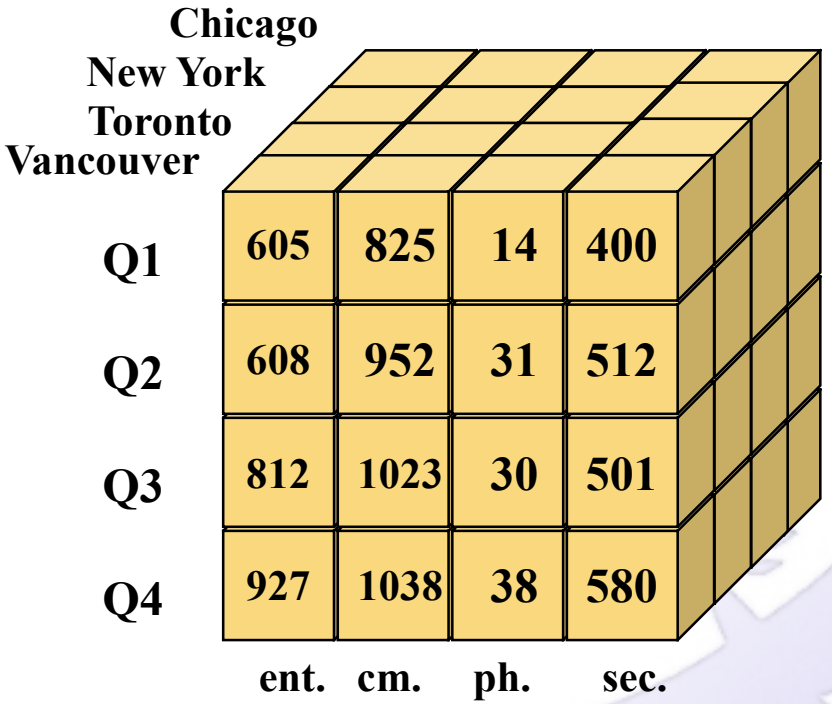
4.1 数组的类型定义

a_0	a_1	a_2								a_n

一维数组

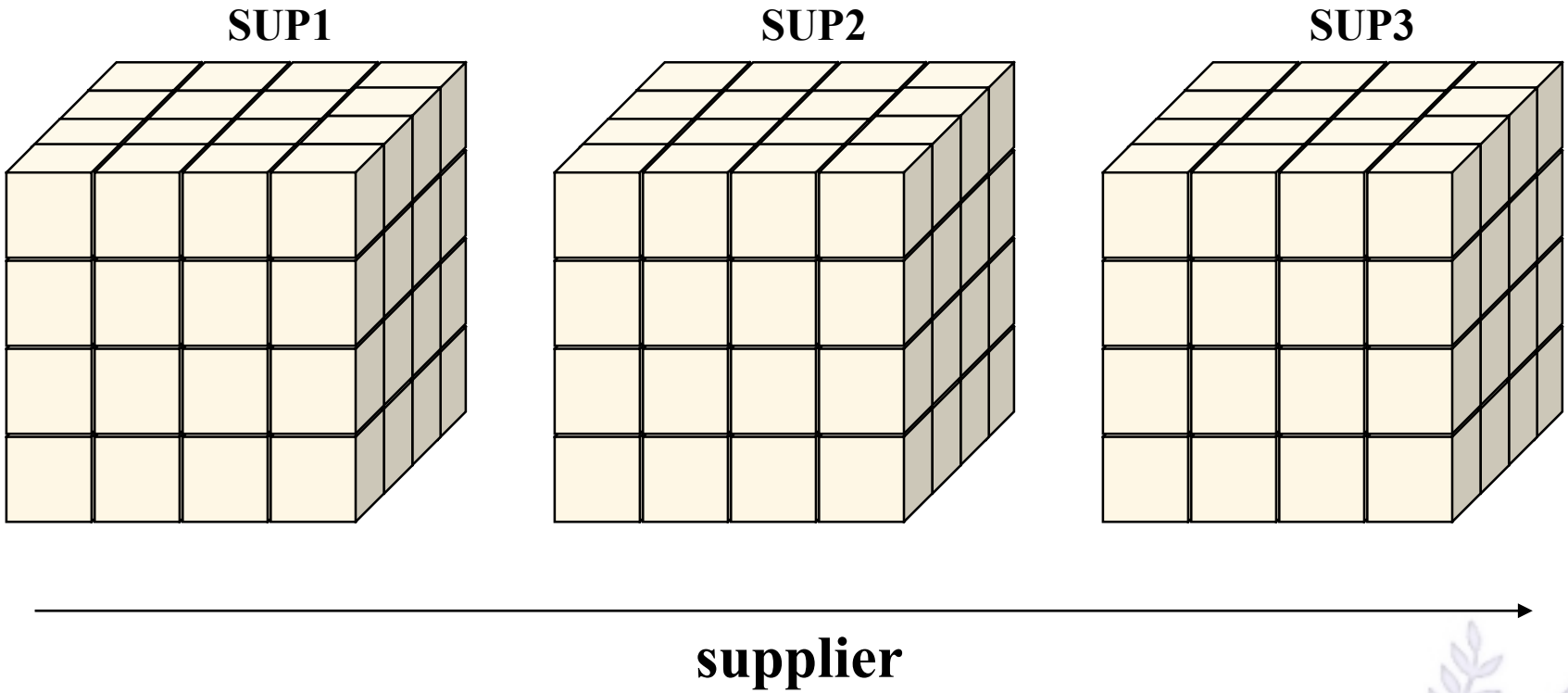
a_{00}	a_{01}	…	a_{0n-1}
a_{20}	a_{21}		a_{2n}
…			…
a_{n0}	a_{n1}	…	a_{n-1n-1}

二维数组



三维数组

4.1 数组的类型定义



四维数组

4.1 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

4.1 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：

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

$D = \{a_{j1,j2,...,ji,jn} \mid a_{j1,j2,...,ji,jn} \in ElemSet\}$

数据关系：

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

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

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

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

$a_{j1,...\ ji,...\ jn}, a_{j1,...ji+1,...jn} \in ElemSet\}$

基本操作：

} ADT Array

基本操作

- **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。**

基本操作

- **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。

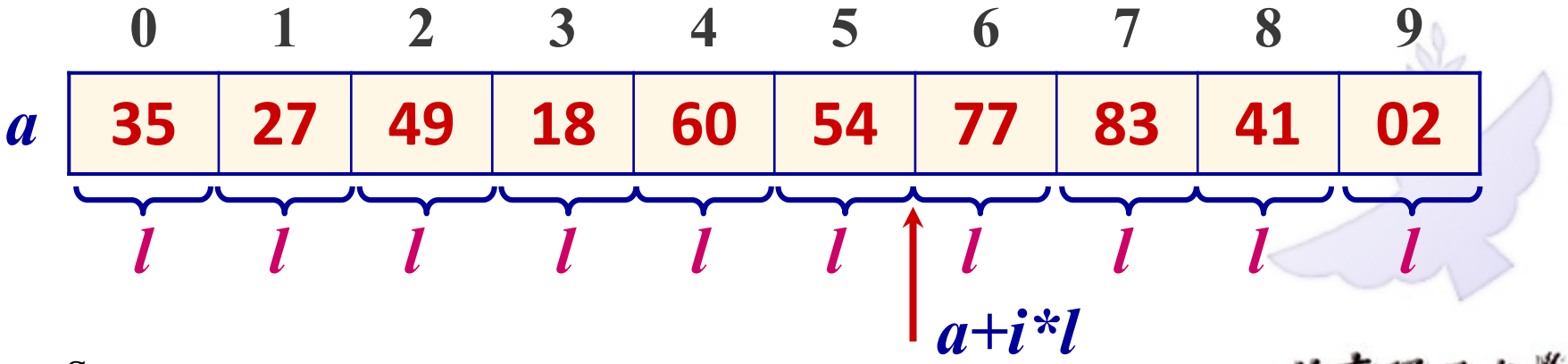
4.1 数组的类型定义

- 数组是一种很特殊的数据结构：
- 数组是一种存储结构，是多种编程语言内建的数据类型。它的操作只有按下标“读 / 写”。
- 数组又是一种逻辑结构，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；但它的数组元素是数组时，是多维数组，是非线性结构。

4.2 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：

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



4.2 数组的顺序表示和实现

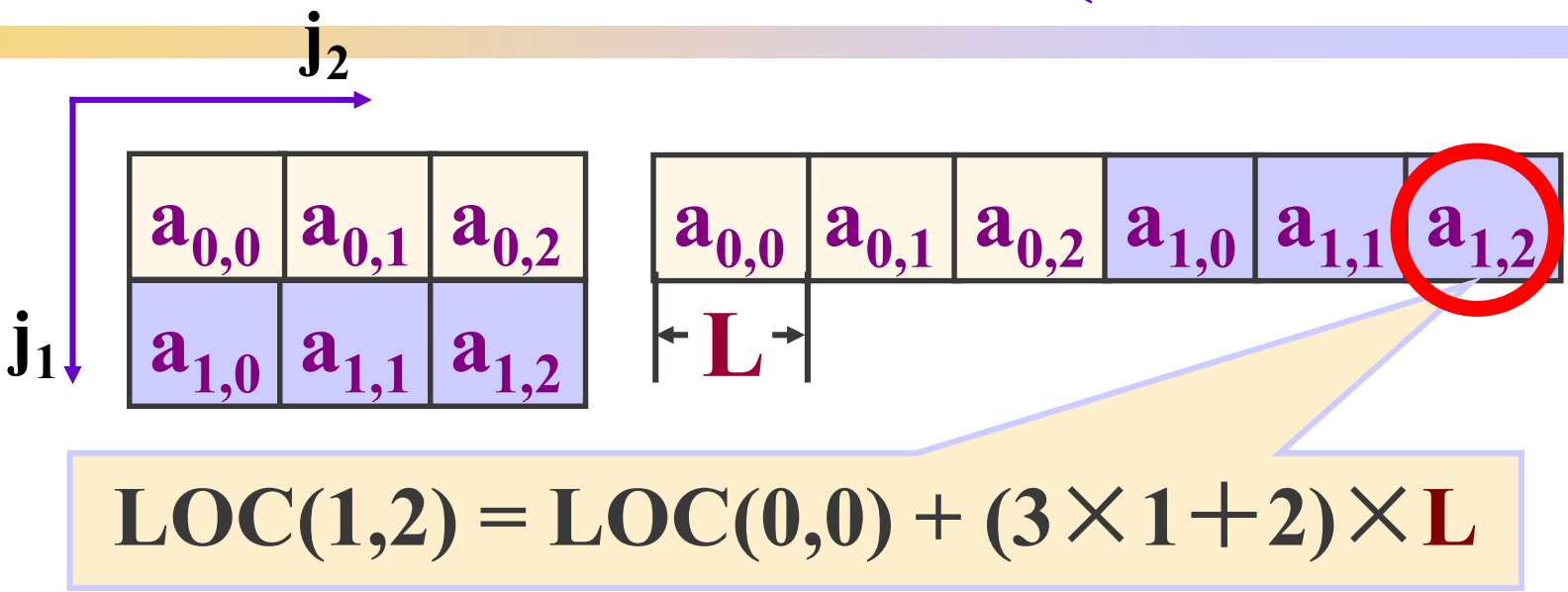
- 多维数组的连续存储表示
- 类型特点:
  - 1) 按下标“读 / 写”;
  - 2) 数组是多维的结构，而存储空间是一个一维的结构。
- 两种顺序映象方式:
  - 1)以行序为主序(低下标优先);
  - 2)以列序为主序(高下标优先)。

以“行序为主序”的存储映象(低下标优先)

A[2][3]

$b_1 = 2$

$b_2 = 3$

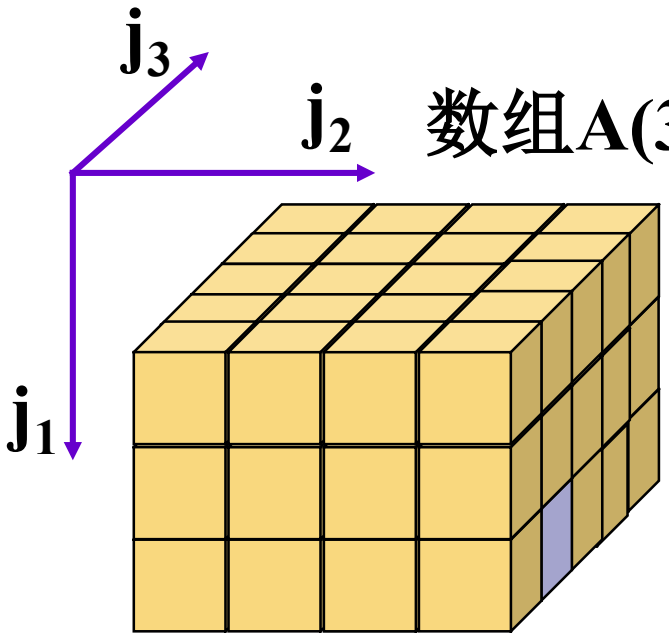


$$LOC(1,2) = LOC(0,0) + (3\times1+2)\times L$$

二维数组A中任一元素a[j1][j2] 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

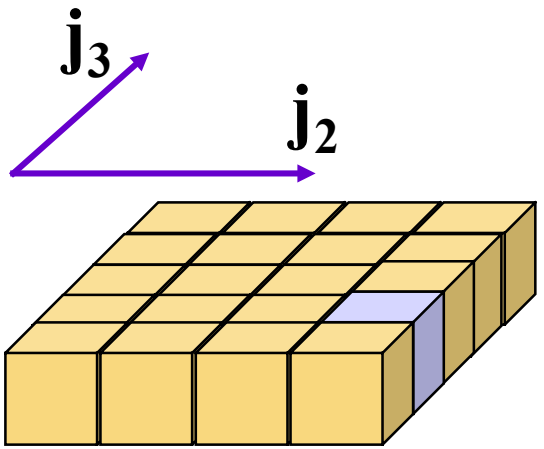


数组A(3,4,5)，求LOC(2,3,1)?

$b_1 = 3$

$b_2 = 4$

$b_3 = 5$



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

三维数组A中任一元素$a_{j1,j2,j3}$ 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(低下标优先)

- 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。

数组元素的存储位置是其下标的线性函数。

4.3 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、带状阵
- 2) 随机稀疏矩阵
  - 非零元很少，且在矩阵Am×n中随机出现

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

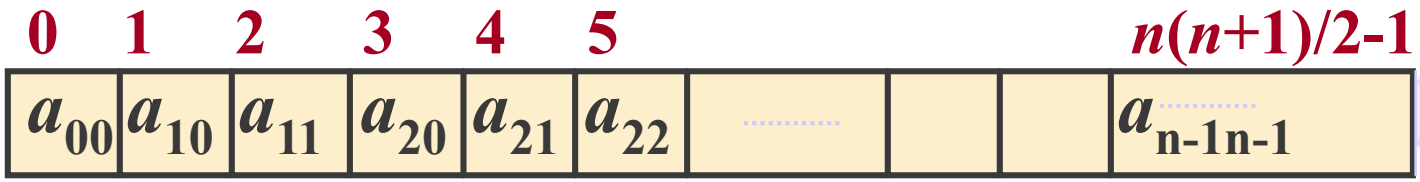
4.3.1 对称矩阵的压缩存储

- 存储下三角数据(行序优先)：矩阵中任意位置的元素a_{ij}与它的存储位置k是一一对应的，其关系是：

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

$a_{ij}=a_{ji}$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$



三对角矩阵的压缩存储

- 总共有$3n-2$个非零元素
- 在三条对角线上的元素a_{ij}满足：

  $0 \le i \le n-1, 0 \le j \le n-1,$

  $i-1 \le j \le i+1$
- 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}

什么是稀疏矩阵

- 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。
- 稀疏因子 $\delta \leq 0.05$ 的矩阵为稀疏矩阵。
- 稀疏因子定义为：

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

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

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。

4.3.2 随机稀疏矩阵的压缩存储方法

- 稀疏矩阵的顺序存储
  - (1)三元组顺序表
  - (2)行逻辑链接顺序表
- 稀疏矩阵的链式存储
  - (1)简单链式存储
  - (2)行链表组
  - (3)十字链表

(1)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.elem

	row	col	*val*
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	7
列数M.Cols	7
元素个数M.Terms	8

三元组顺序表的定义

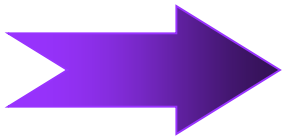
```
#define  MAXSIZE  12500
typedef struct {
    int  row, col;      //该非零元的行下标和列下标
    DataType  value;    // 该非零元的值
} Triple;  // 三元组类型
```

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

如何求转置矩阵？

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

A



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

B

$$B_{ij} = A_{ij}$$

用常规的二维数组表示时的算法

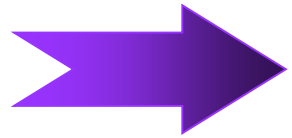
$$B_{ij} = A_{ij}$$

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

其时间复杂度为: O(Rows×Cols)

用“三元组”表示时如何实现？

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



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

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

基本操作：

按照A的列序，
依次从A中找出
属于当前列的元素
放在B中

用“三元组”表示时的操作步骤

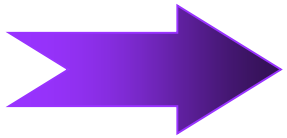
- void TranMatrix(SparseMatrix A, SparseMatrix &B)
- 设置B的各个参数：
  - B.Rows=A.Cols；B.Cols=A.Rows；B.Terms=A.Terms;
- 设指向B中当前元素的指针为q；
- 每次得到B中的一行元素，设当前行号为row
  - for (row=0; row<=B.Rows-1；++row)
  - 一行中的每一个元素怎么得到？
  - 从M中所有的元素中找其列号j==row的元素
    - for (p=0;p<=A.Terms-1;++p)
    - 条件：A.elem[p].col == row

```
void TranMatrix(SparseMatrix A, SparseMatrix &B)
{ //采用三元组表存储稀疏矩阵，求A的转置矩阵B
  B.Rows=A.Cols;  B.Cols=A.Rows;  B.Terms=A.Terms;
  if (B.Terms <=0 ) return;
  q=0;  // q为B.elem[ ]当前三元组的位置(下标)
  for (row=0; row<B.Rows;  ++row)
     for (p=0;p<A.Terms;++p)//p:扫描A.elem指示器
          if (A.elem[p].col == row)
          {      B.elem[q].row =A.elem[p].col;
                 B.elem[q].col=A.elem[p].row;
                 B.elem[q].value=A.elem[p].value; ++q;
          }
}//  TransposeSMtrix
```

复杂度：	O(Cols*Terms)

能否直接放到正确位置？

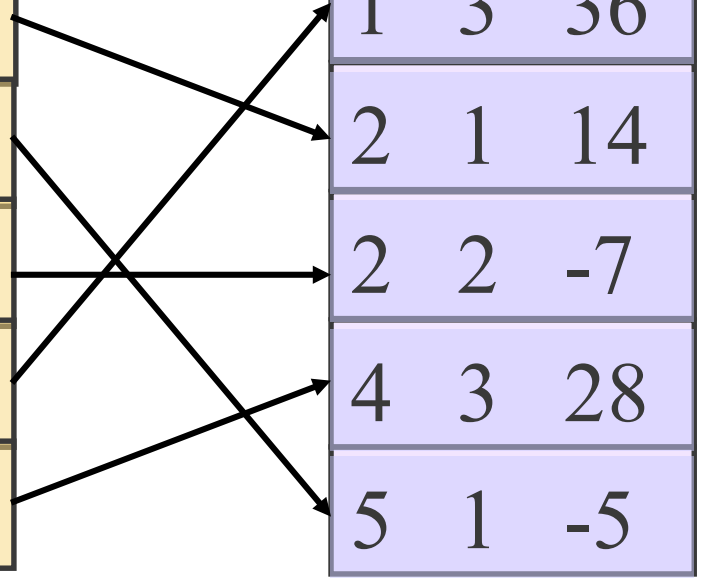
$$\begin{bmatrix} 0 & 14 & 0 & 0 & -5 \\ 0 & -7 & 0 & 0 & 0 \\ 36 & 0 & 0 & 28 & 0 \end{bmatrix}$$



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

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

A的列	B中的位置
1	1
2	2
3	4
4	4
5	5

确定A中每一列的第一个非零元素在B中开始的位置

矩阵快速转置算法

- 1）确定A中每一列的第一个非零元素在B三元组中的位置
- 2）将A中的元素放在B中恰当位置

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

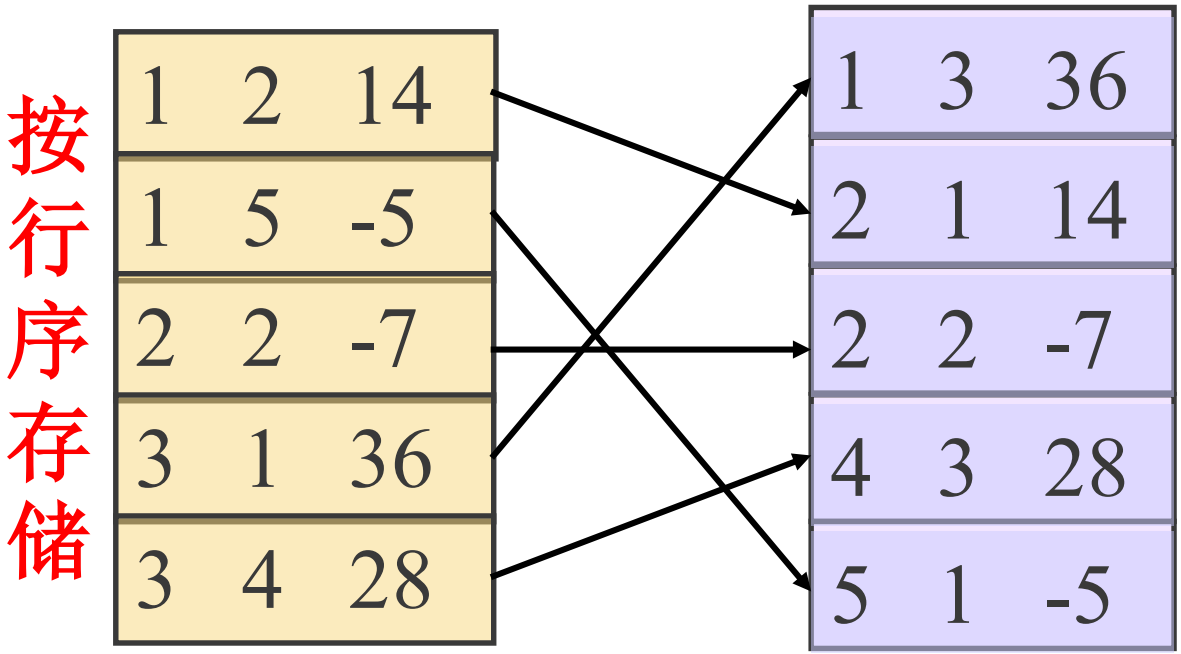
```
cpot[1] = 1;
for (col=2; col<=A.Cols; col ++;)
        cpot[col] = cpot[col-1] + num[col-1];
```

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A中的元素放在B中恰当位置

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



- 2）将A中的元素放在B中恰当位置

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

```
for (p=0; p<=A.Terms-1; ++p){//把非零元素放置到正确位置
    col = A.elem[p].col; q=cpot[col];
    B.elem[q].row = A.elem[p].col;
    B.elem[q].col = A.elem[p].row;
    B.elem[q].value = A.elem[p].value;
    cpot[col]++;
}
```

矩阵快速转置算法的操作步骤

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

```
Status FastTransposeSMatrix(SparseMatrix M, SparseMatrix &T){
  B.Rows = A.Cols;  B.Cols = A.Rows;
  B.Terms = A.Terms; //步骤1
  if (B.Terms <=0) return NOELEM;
  for (col=0; col<= A.Cols-1; ++col)  num[col] = 0; //步骤2
  for (t=0; t<=A.Terms-1; ++t)  ++num[A.elem[t].j];
  cpot[1] = 0; //步骤3
  for (col=1; col<=A.Cols; ++col)
        cpot[col] = cpot[col-1] + num[col-1];
  for (p=0; p<=A.Terms-1; ++p) {…转置矩阵元素} //步骤4
  return OK;
} // FastTransposeSMatrix
```

复杂度：O(Cols+Terms)

算法分析

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

空间换时间!

(2)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.elem

	row	col	val
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
	6	1	15
	6	4	-7

M. rpos

1	1
2	3
3	3
4	5
5	6
6	7

行数M.Rows	7
列数M.Cols	7
元素个数M.Terms	8

按行序存储

带行表的三元组

```
#define maxrow 100
typedef struct{
    triple elem[maxsize];//数据
    int    rpos[maxrow];//行表
    int    Rows, Cols, Terms;//行数，列数，元素个数
}rtripletable
```

矩阵的乘法

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

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

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

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

如果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

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

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

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第i行的每个元素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。

用三元组表示矩阵时的算法

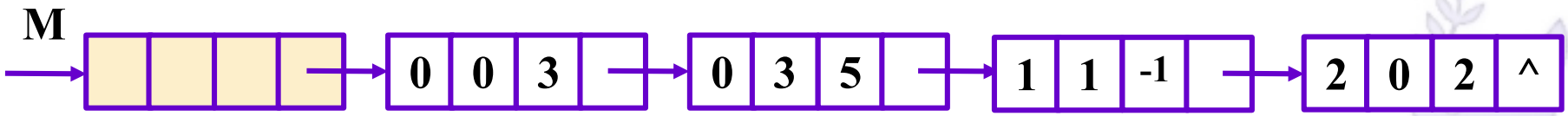
- 算法的基本思想：
- 每次计算Q中的一行元素，设当前行号为arrow。
- 设临时变量ctemp[ ]保存该行元素，先将其清零。
- 计算每行元素：
  - M中第arrow行的每一个元素(arrow, j)分别与N中第j行的元素(j, ccol)相乘，其结果累加到ctemp[ccol]中。
- 计算完后将该行元素压缩存储到Q的三元组中。

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

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

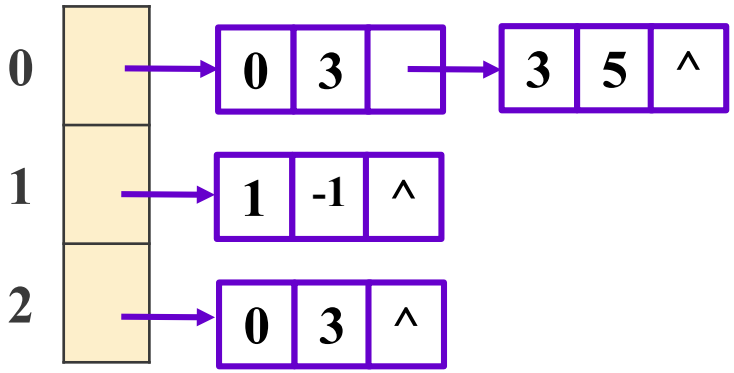


稀疏矩阵的链式存储

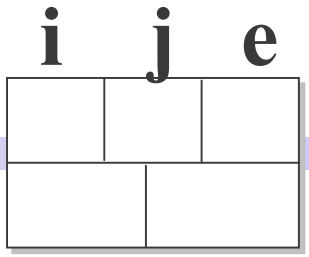
- (2)行链表组

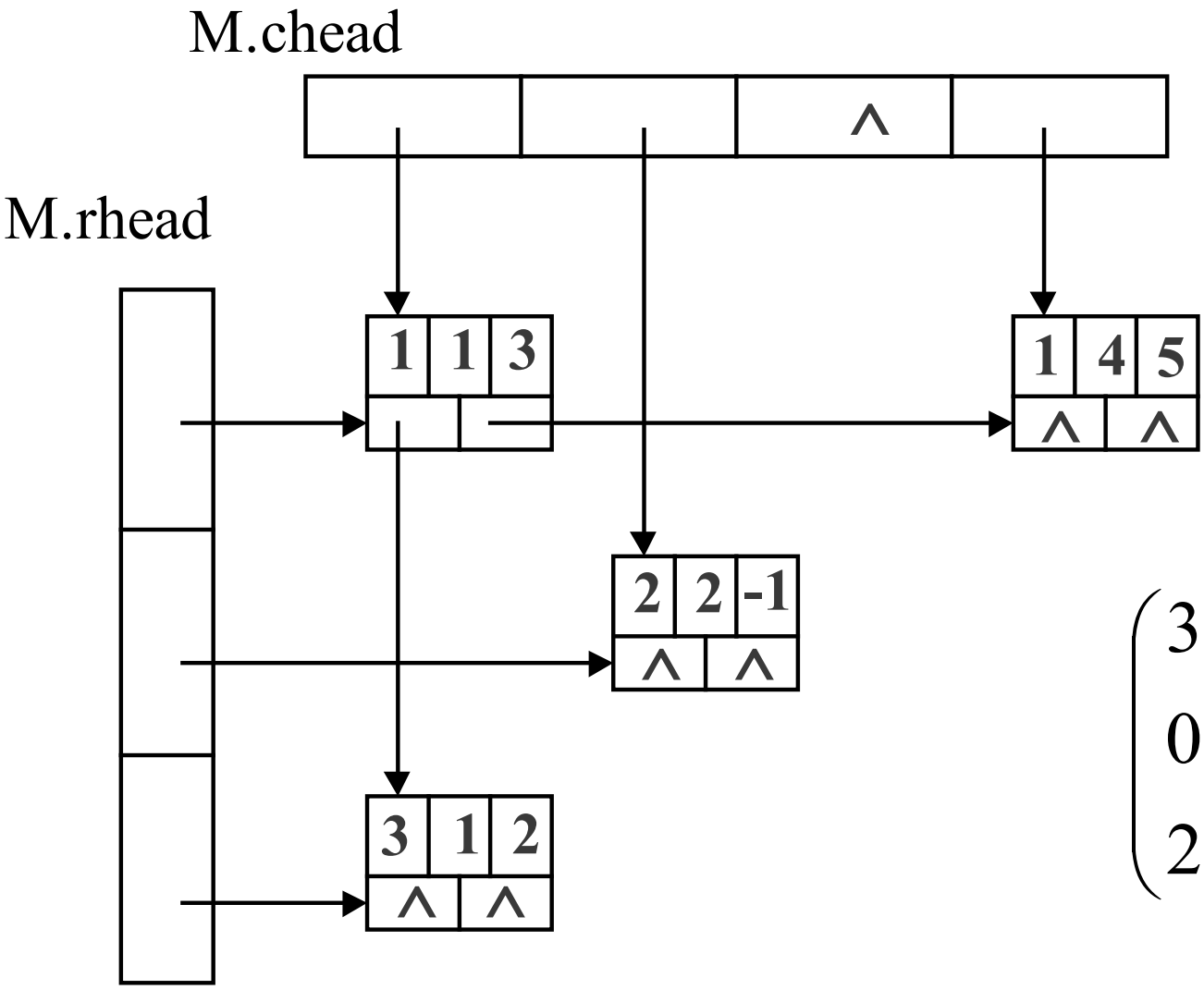
  - 提高了存取元素的效率
  - 易于增删非零元素

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



（3）十字链表

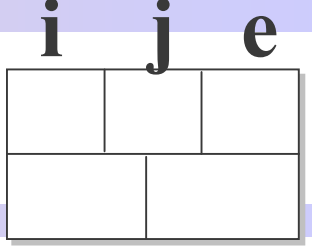




适合于对矩阵频繁修改的情况

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

十字链表的类型定义 Orthogonal

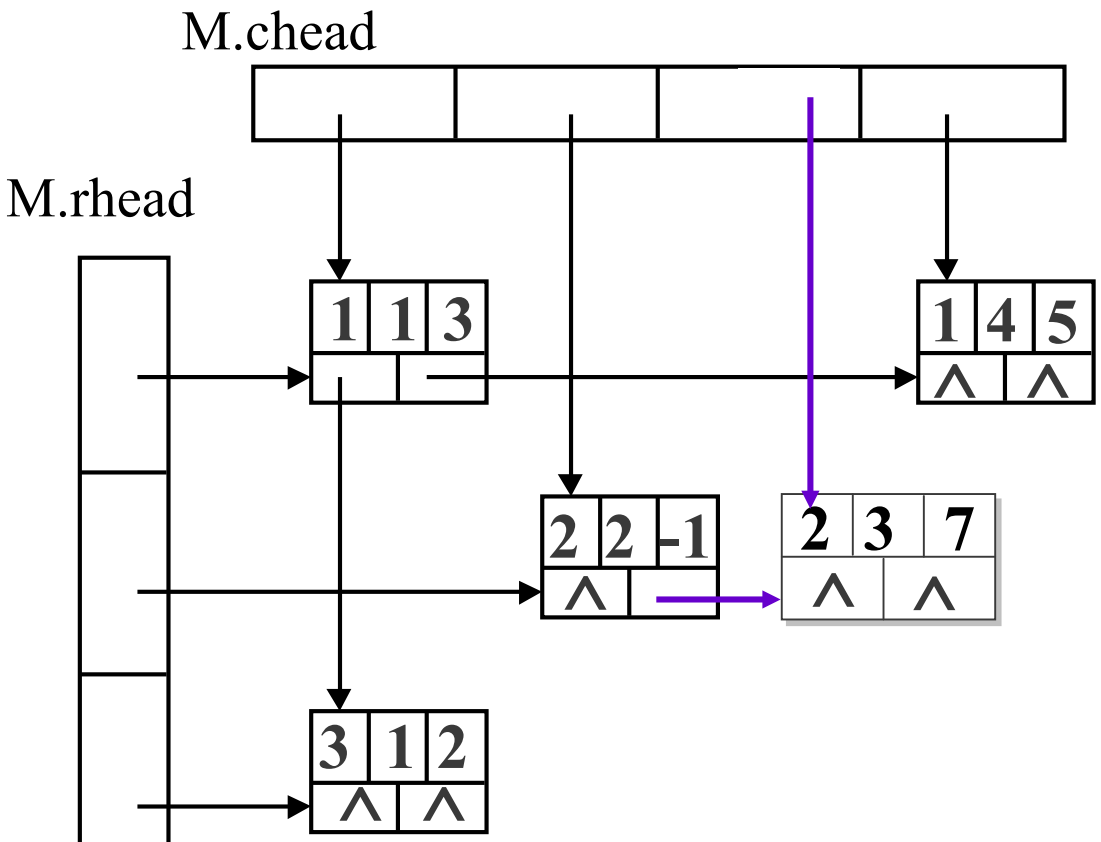



```
typedef struct OLNode {
   int   i, j;      // 非零元的行下标和列下标
   ElemType  e;                 //非零元值
   STRUCT OLNode *right,*down
}OLNode, OLink;
```

```
typedef struct {
    OLink *rhead,*chead; //行、列表头指针数组
    int    Rows, Cols, Terms;  //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



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

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

4.4 广义表

- 广义表(generalized list)：一种不同构的线性结构。

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

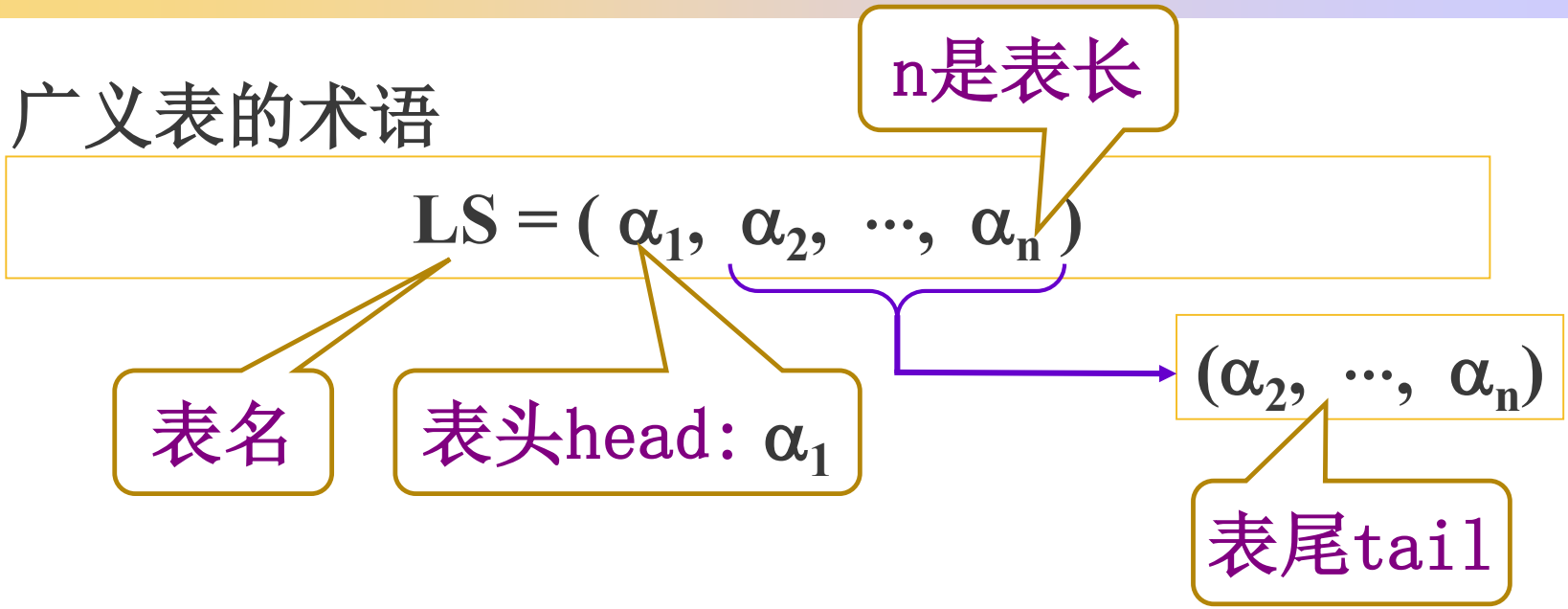
其中：α_i 或为原子(atom) 或为广义表。

- 广义表的基本性质

1. 广义表的定义是一个递归定义，因为在描述广义表时又用到了广义表；
2. 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)；
3. 当每个元素均为原子且类型相同时，就是线性表。

4.4.1 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

深度：所含括弧的重数。原子的深度为0，“空表”的深度为1。

4.4.1 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
  - B = (a, B) = (a, (a, (a, … ,))) 递归定义

    长度：2；深度:无限

  - B = (a, (a, (a, … ,)))

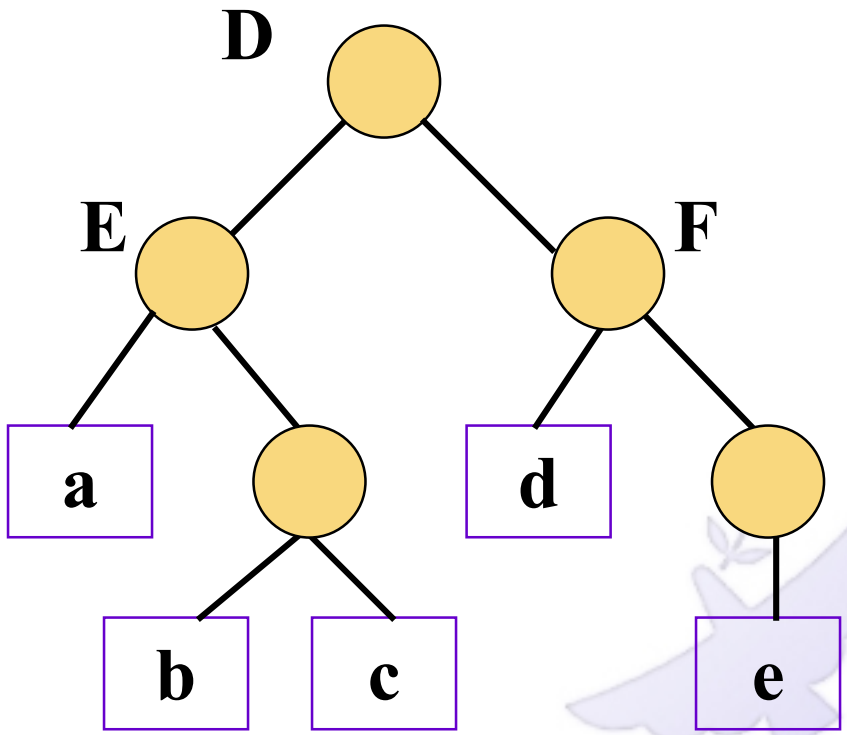
广义表的结构特点

- 广义表的应用
  - LISP语言
  - 专家系统
  - 数据挖掘……
- 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。

广义表的结构特点

- 4）广义表是一个多层次的线性结构
- 例如：D=(E, F)
  - E=(a, (b, c))
  - F=(d, (e))

用○表示子表
用□表示原子



D：长度：2；深度：3

深度＝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)) = ( )

广义表的结构特点

- 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)
- a= ?
- a=Head(Head(D))
- b= ?
- b=Head (Head(Tail(Head(D))))

( )

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

C A

u v

a

x y z

再入表

(B, D)

F

d

递归表

(d, F)

4.4 广义表的表示方法

ADT GList {

数据对象：D＝{e_i | i=1,2,..,n; n≥0;

e_i∈AtomSet 或 e_i∈GList,

AtomSet为某个数据对象集合}

数据关系：

LR＝{<e_{i-1}, e_i >| e_{i-1} ,e_i∈D, 2≤i≤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);

- **状态函数**

  GListLength(L);　GListDepth(L);

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

基本操作

- 插入和删除操作

  InsertFirst_GL(&L, e);

  DeleteFirst_GL(&L, &e);

- 遍历

  Traverse_GL(L, Visit());

4.5 广义表的存储结构

- 4.5.1 广义表的头尾表示法
- 头尾链表结构

表结点：

tag=1	hp	tp

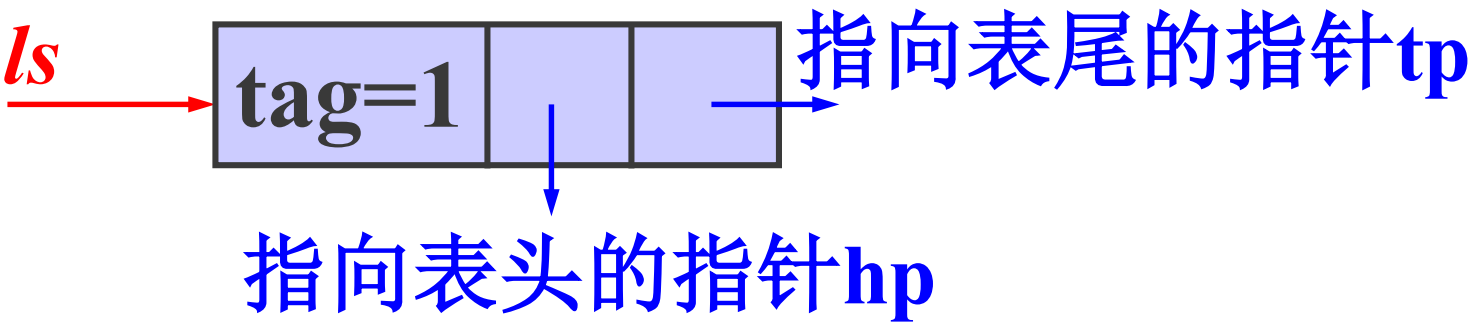
原子结点：

tag=0	data

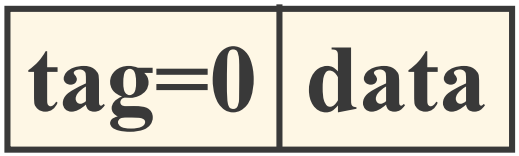
4.5.1 广义表的头尾链表存储表示

空表　　*ls=NULL*

非空表 *ls*



若表头为原子，则为



否则，依次类推。

4.5.1 头尾链表存储表示

表结点：

tag=1	hp	tp

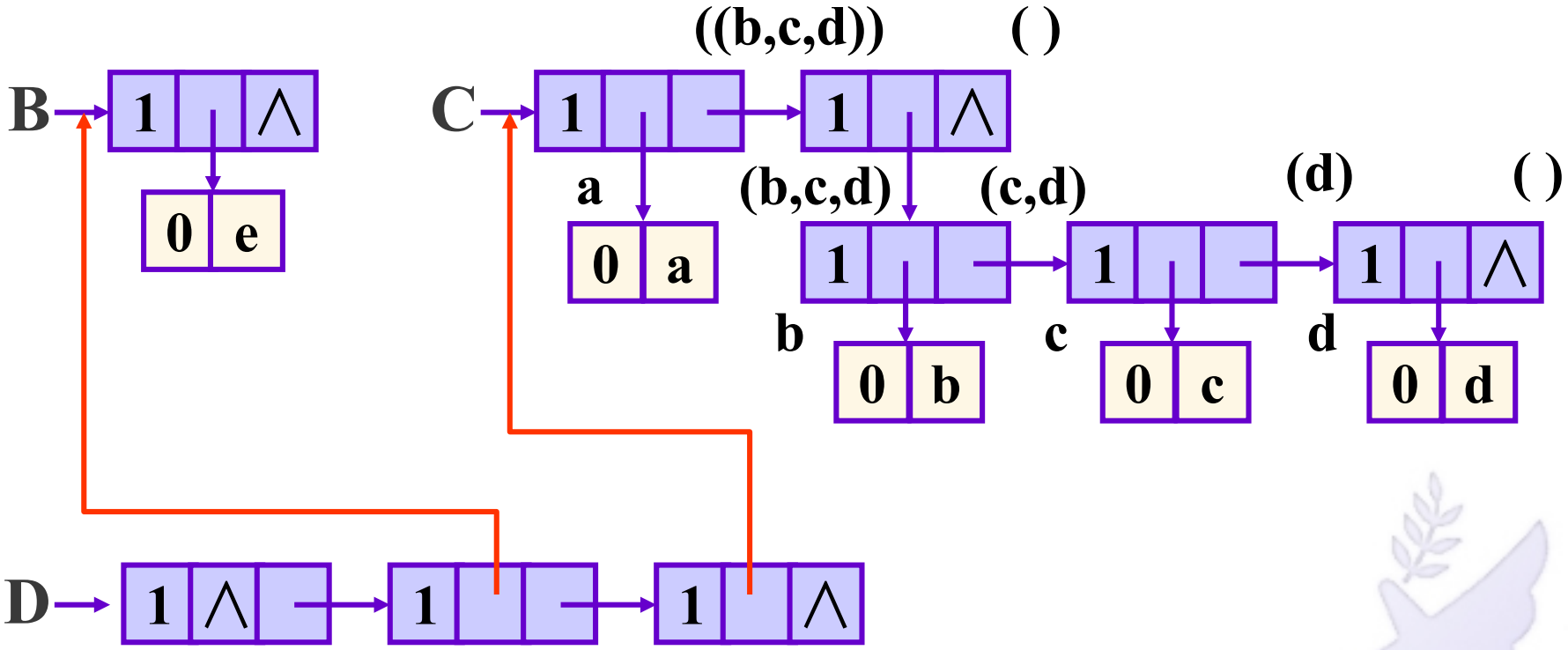
原子结点：

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
```

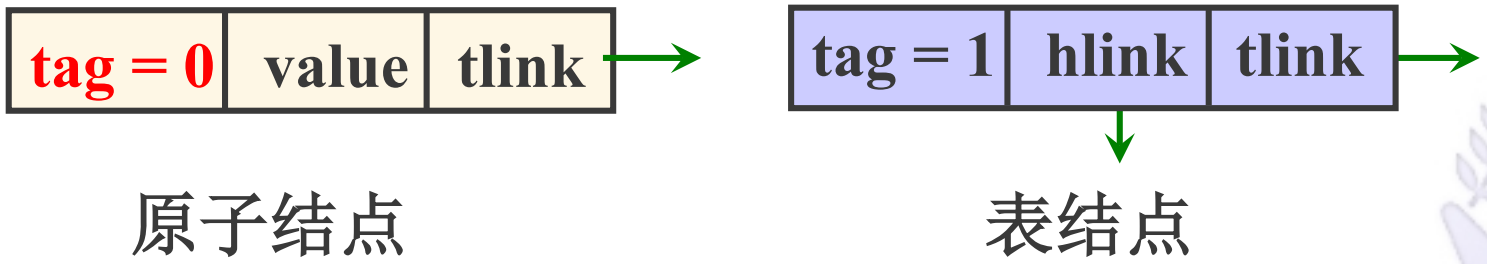
A= （） B = (e)　C = (a,(b,c,d))　D = (A,B,C)

A = NULL

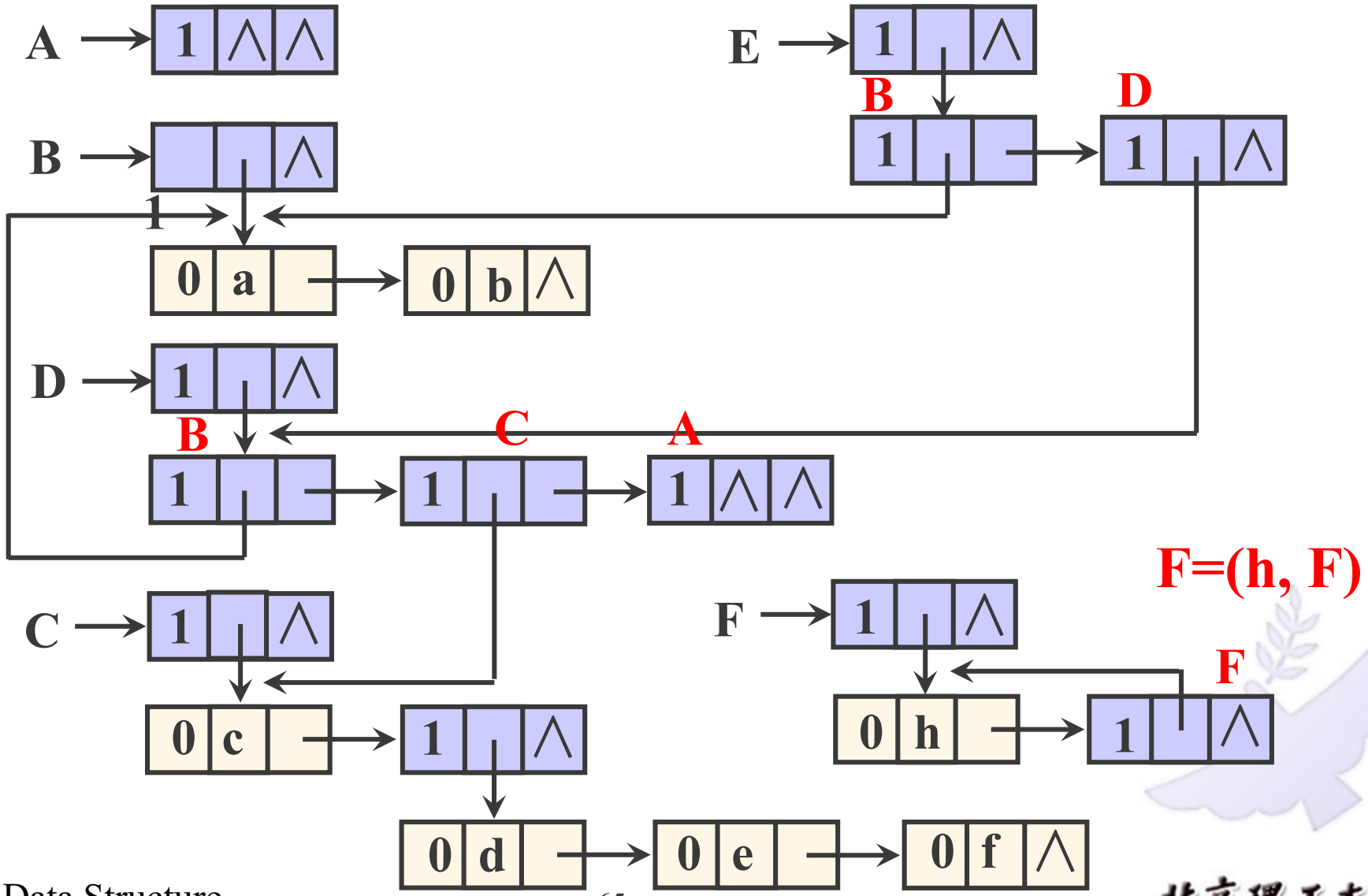


4.5.2 广义表的扩展线性链表表示

- 有两种结点：原子结点和表结点。
  - ¶ 原子结点：标志 tag = 0，value 存放元素的值，tlink 存放指向同一层下一个表元素结点的指针；
  - ¶ 表结点的标志 tag = 1，hlink 存放指向子表的指针，tlink 与单元素结点tlink的含义相同



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

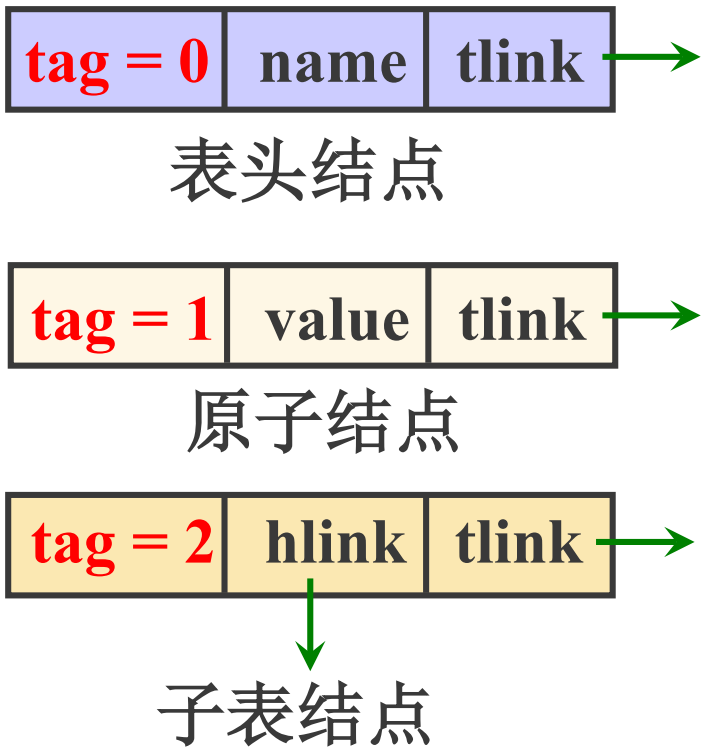


4.5.2 广义表的扩展线性链表表示

- 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”，而是直接指向了广义表的表头元素结点（是第一个表元素结点），所以对表头元素结点插入或删除时的困难。

4.4.3 广义表的层次表示

- 含有 3 种结点：原子结点、子表结点和头结点（非表头元素结点）



name：表名

tlink：指向表的第一个结点

value：数据

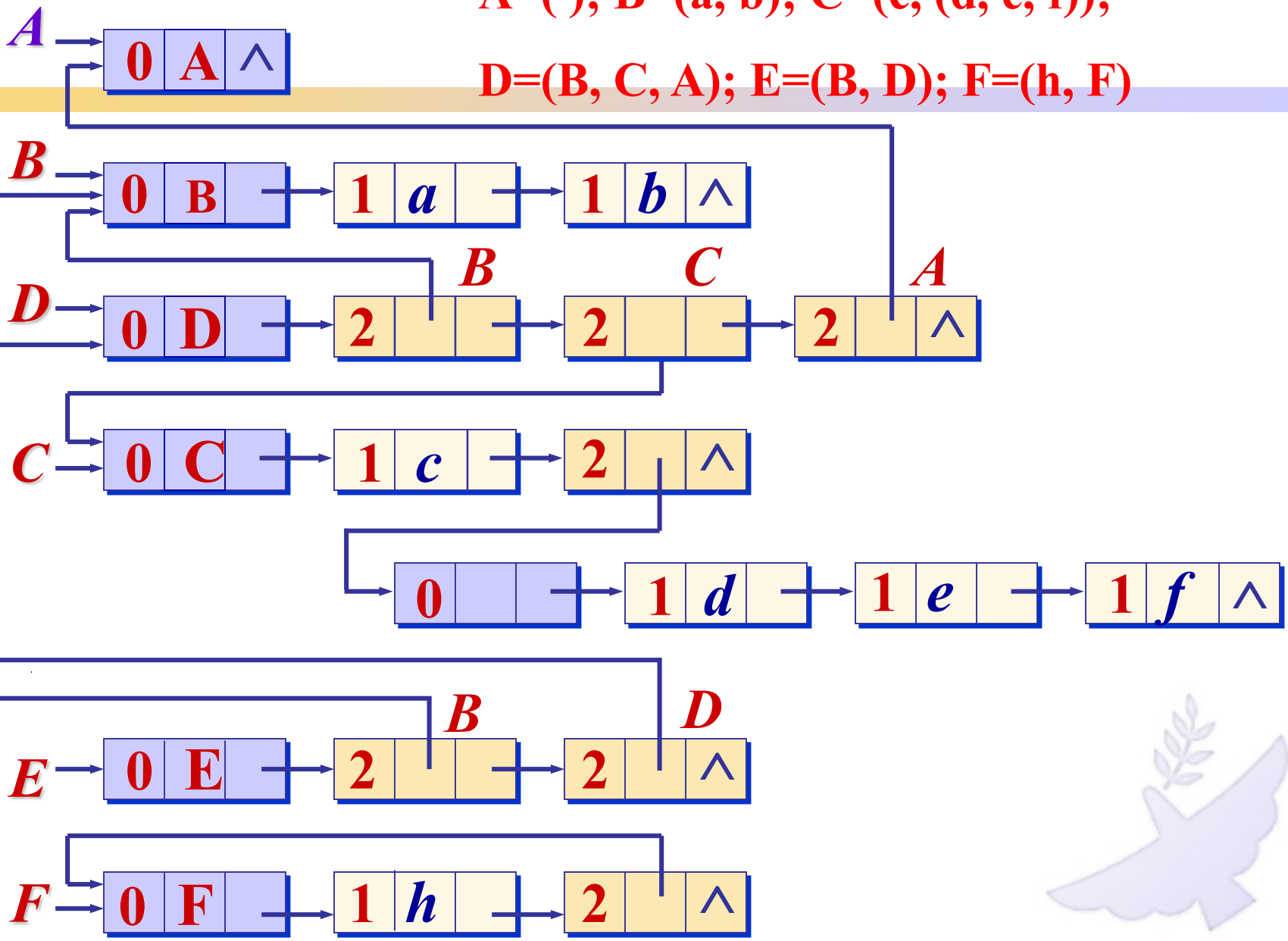
tlink：指向同层下一个结点

hlink：指向子表的指针

tlink：指向同层下一个结点

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

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



4.4.3 广义表的层次表示

```
typedef struct node {      //广义表结点定义
    int tag;
     //＝0为头结点，=1为原子结点，=2是子表结点
    struct node *tlink;      //指向同层下一结点的指针
    union {          //共用体，此3个域叠压在同一空间
        char name;   //tag=0，存放表名，设为单字符
        char value;  //tag=1，存放数据，设为单字符
        struct node *hlink;
                     //tag=2，存放指向子表的指针
    } info;
} GenListNode, *GenList;
```

4.4.3 广义表的层次表示

- 特点：
  - 表示更简洁
  - 有表头结点，所以插入删除操作简单

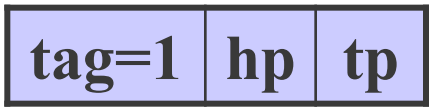
4.6 广义表的操作

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

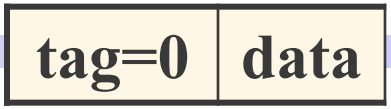
- 以下操作以广义表头尾链表结构为例

回顾：头尾链表存储表示

表结点：



原子结点：

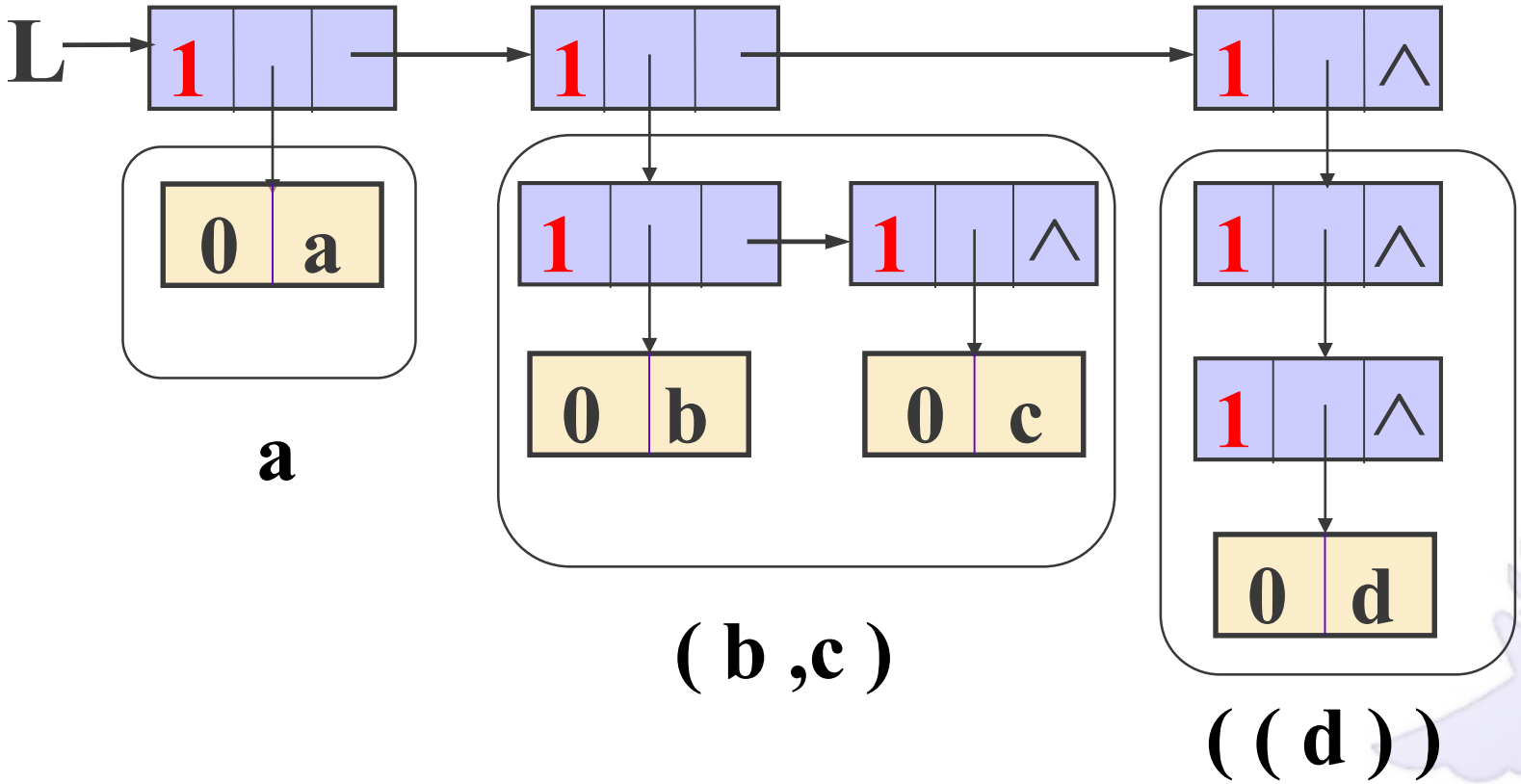



```
typedef enum {ATOM, LIST} ElemTag;
        // ATOM==0:原子, LIST==1:子表
```

```
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
```

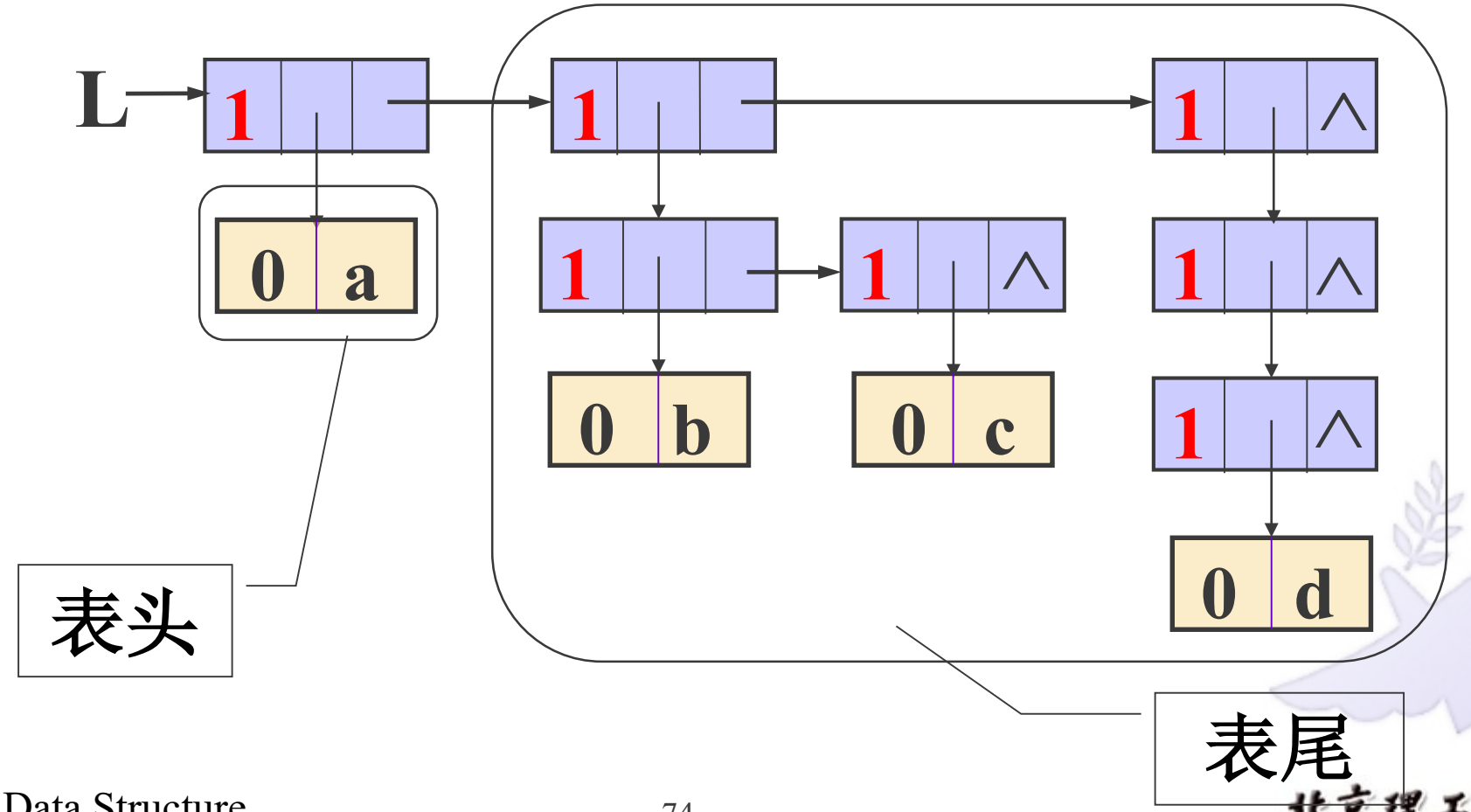
回顾：头尾链表结构

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



回顾：头尾链表结构

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



4.6 广义表操作的实现

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

4.6.1 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

4.6.1 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

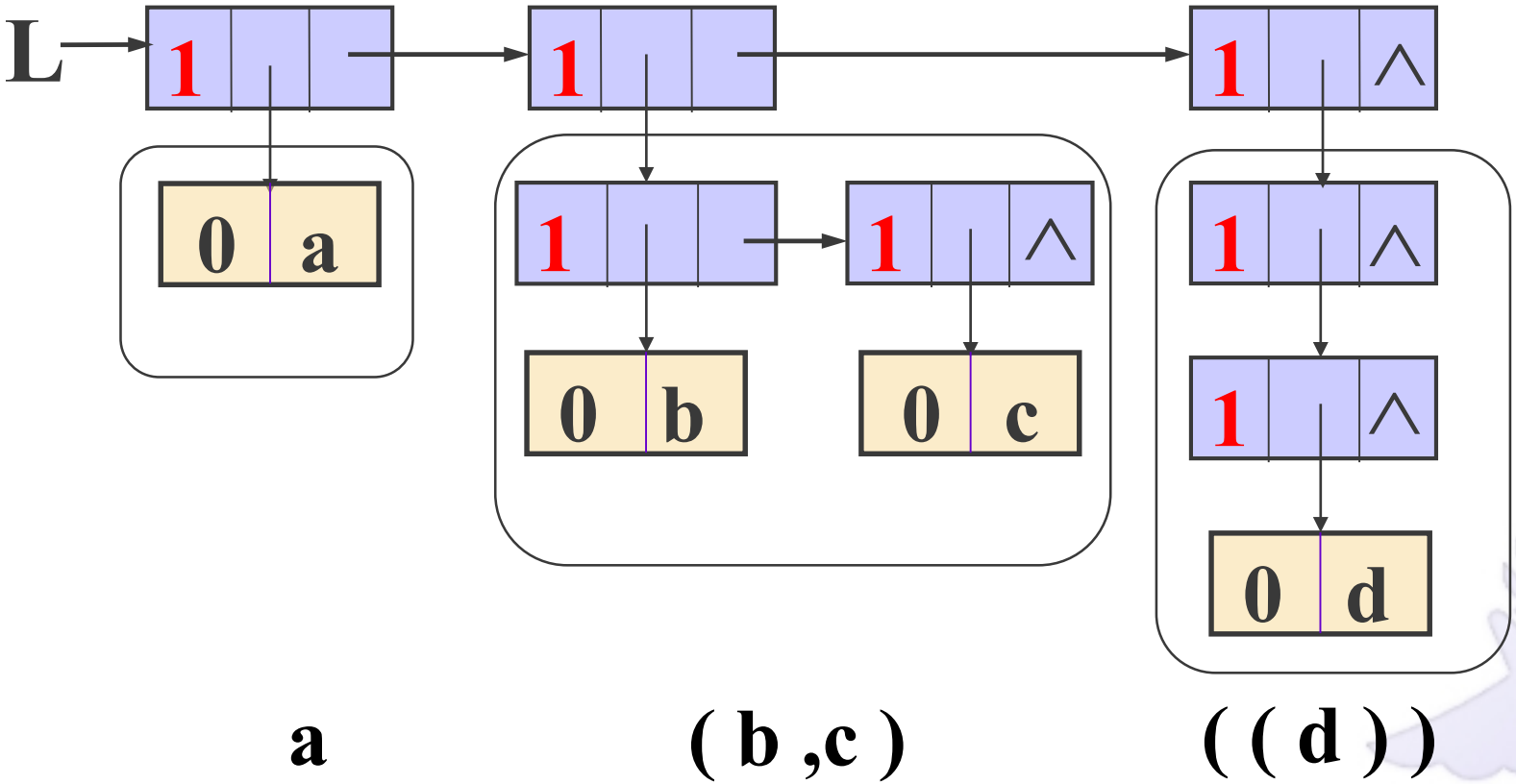
直接求解

空表：深度 = 1

原子：深度 = 0

4.6.1 求广义表的深度

L = (a , (b ,c) , ((d))) 的深度



4.6.1 求广义表的深度

表结点：

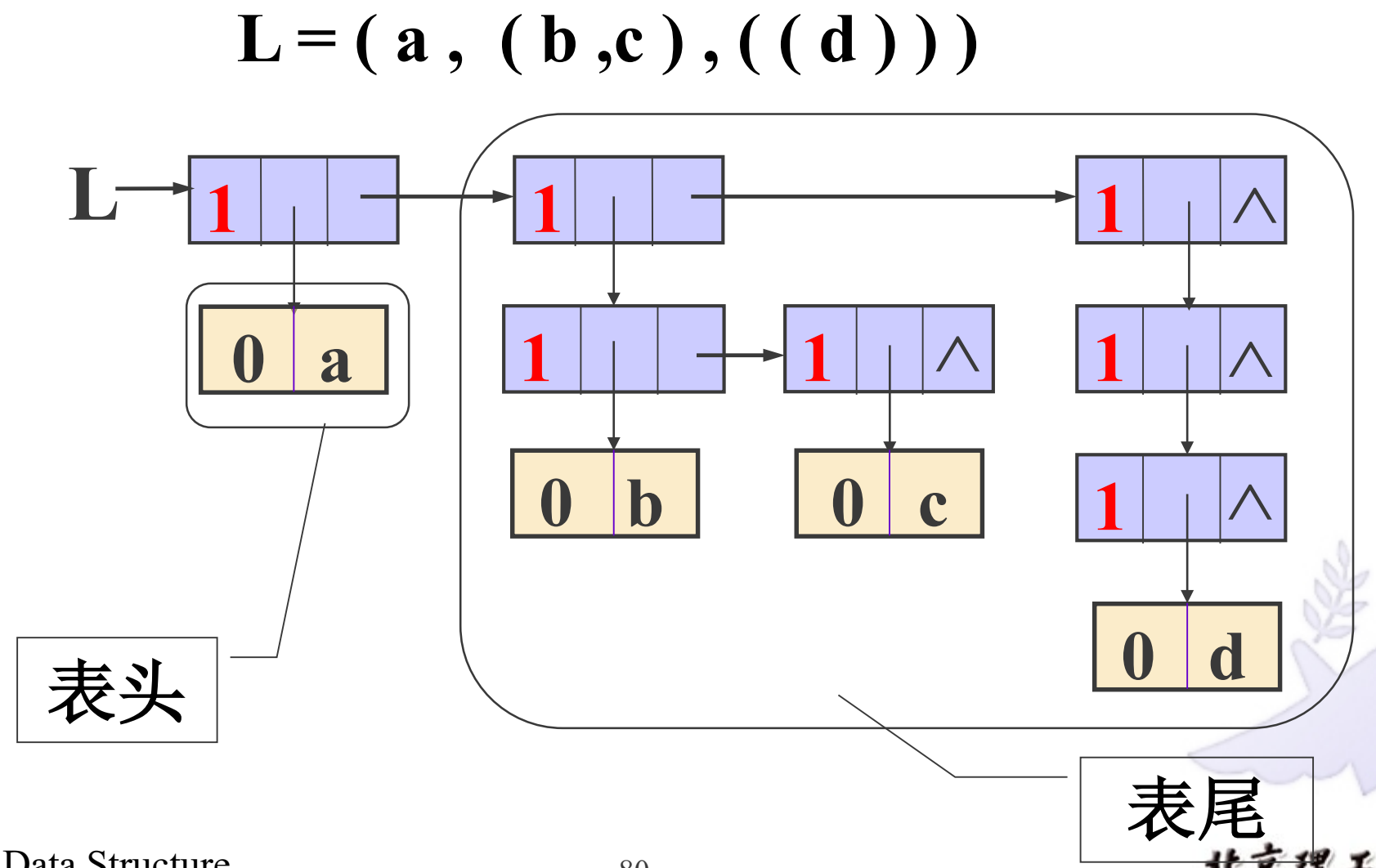
tag=1	hp	tp

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==NULL )  return 1;      // 空表深度1
      if ( L->tag==ATOM ) return 0;   // 原子深度0
      for ( max=0, pp=L; pp!=NULL; pp=pp->ptr.tp )
      {
          dep = GListDepth( pp->ptr.hp );
          if ( dep>max )    max = dep;
      }
      return max+1;
}// GListDepth
```

4.6.2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 == NULL ) T=NULL;          // 复制空表，终止条件
    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,终止条件
        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
```

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

4.6.3 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。

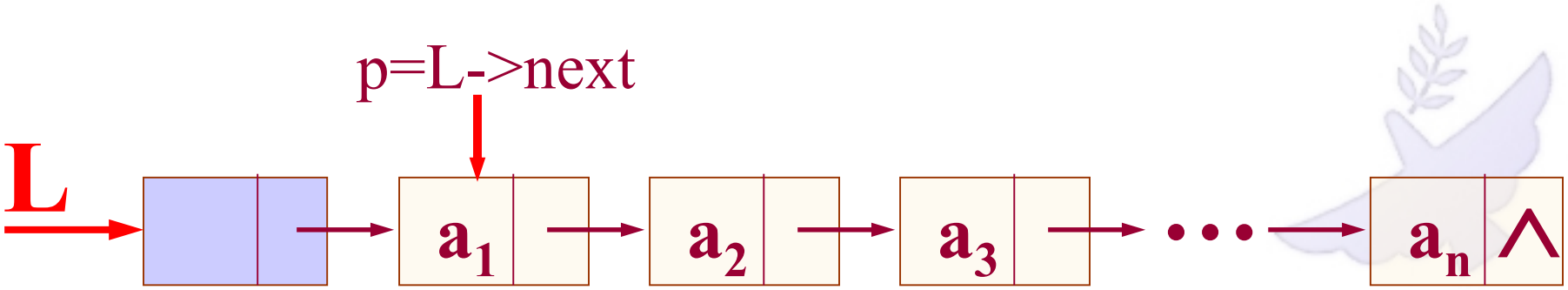
不同处：

1)广义表的数据元素可能还是个广义表；

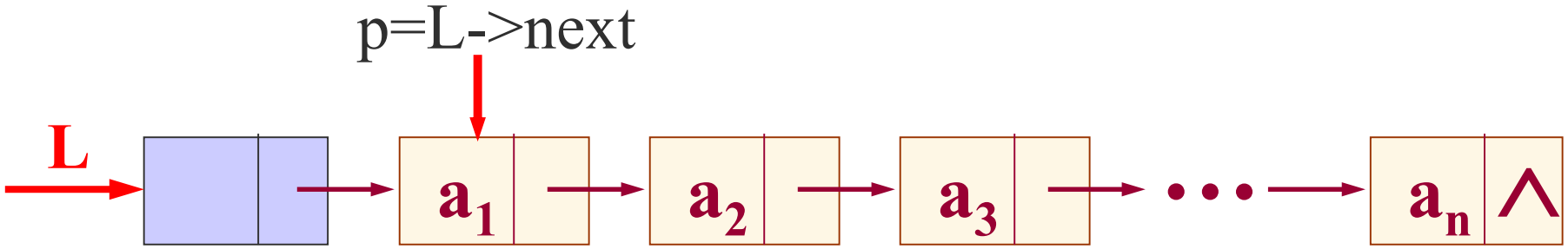
2)删除时，不仅要删除原子结点，还需要删除相应的表结点。

4.6.3 删除广义表中所有元素为 x 的原子结点

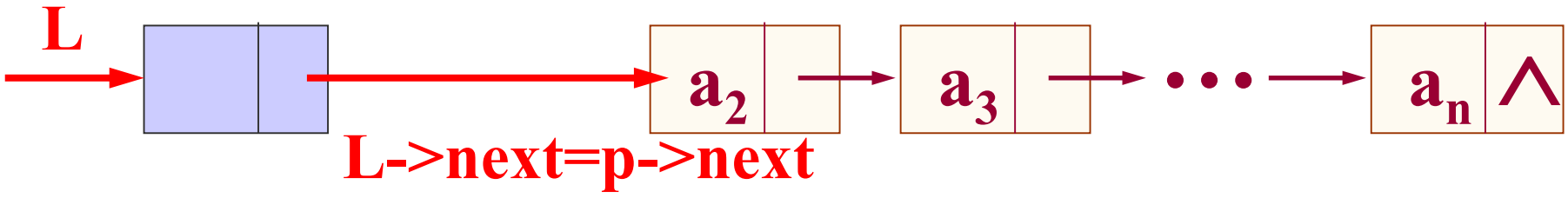
- 删除单链表中所有值为x 的数据元素
- 基本思想：
  - 1) 单链表是一种顺序结构，必须从第一个结点起，逐个检查每个结点的数据元素；
  - 2) 从另一角度看，链表又是一个递归结构
  - 若 L 是线性链表 ($a_1, a_2, \ldots, a_n$) 的头指针
  - 则 L->next是线性链表 ($a_2, \ldots, a_n$)的头指针。



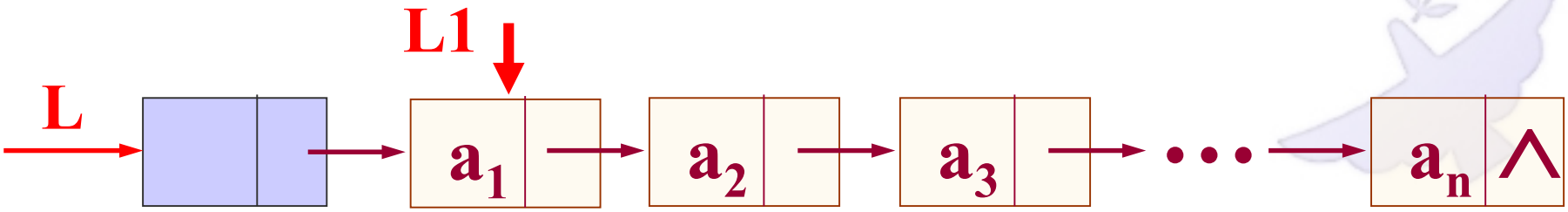
删除单链表中所有值为x 的数据元素



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



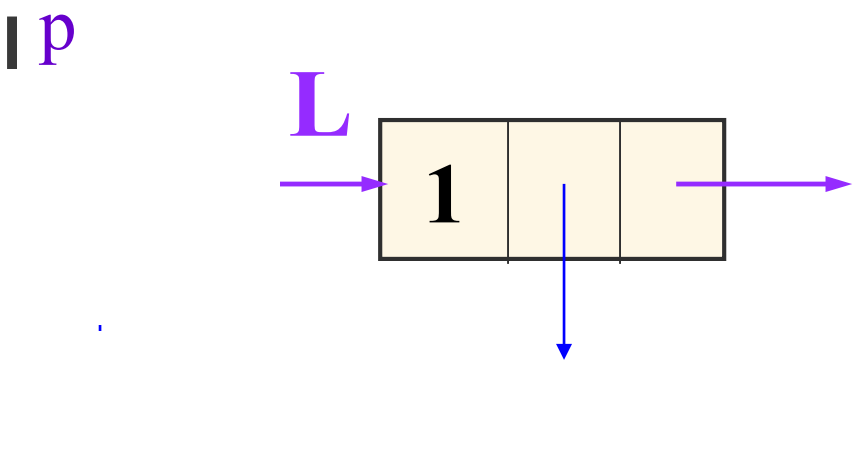
删除单链表中所有值为x 的数据元素

```
void delete(LinkList L, ElemType x) {
    if (L != NULL) {
        if (L->next != NULL) {
            if (L->next->data==x) {//a1=x
                p=L->next;  L->next=p->next;
                free(p);  delete(L, x);
            }
            else  delete(L->next, x);//a1!=x
        }//if (L->next != NULL)
    }//if (L != NULL)
} // delete
```

删除广义表中所有元素为x的原子结点

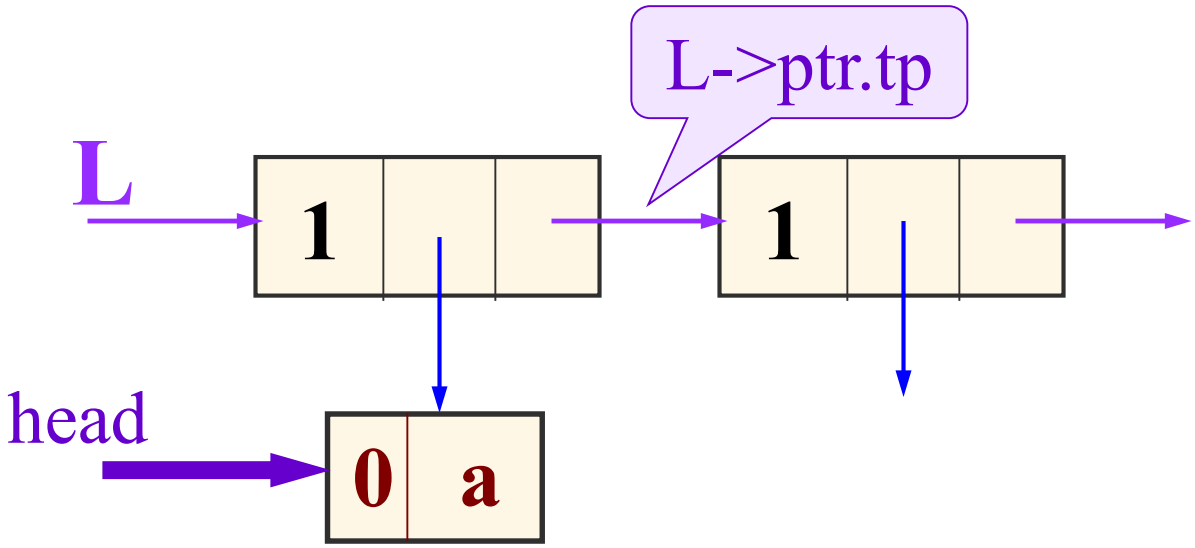
```
void Delete_GL(Glist&L, AtomType x) {
  //删除广义表L中所有值为x的原子结点
  if (L != NULL) {
     head = L->ptr.hp;  // 考察第一个子表
     if ( (head->tag == Atom) &&
                          (head->atom == x))
       {   ……          }   // 原子项 等于x的情况
     elseif((head->tag == Atom) &&  (head->atom != x))
       {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

第一项是原子项，且等于x



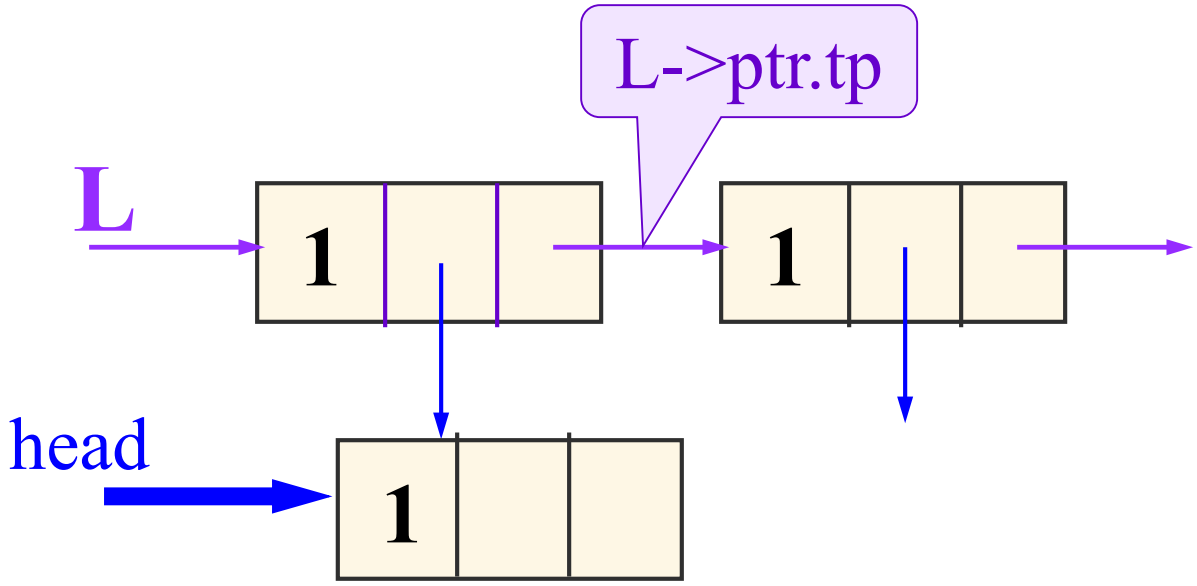

```
if ((head->tag == Atom) &&  (head->atom == x))
{   p=L;    L = L->ptr.tp;    // 修改指针
    free(head); free(p); // 释放原子结点、表结点
    Delete_GL(L, x);  // 递归处理剩余表项
   }
```

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—计算广义表长度

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

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

广义表操作的实现—计算广义表深度

```
int GListDepth(GList L) {
        if (L == NULL) 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
```

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

广义表操作的实现—删除节点

```
Status DeleteFirst_GL( GList &L)
{//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 );   →→DestroyGList(e)
        free( p );
        return OK;
}// DeleteFirst_GL
```

?

4.6.3 建立广义表

- 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
  - 逗号

4.6.3 建立广义表

s =“A(c, H(d, e, f)) ”

- 基本思想：从s中取得一个字符，检测其内容

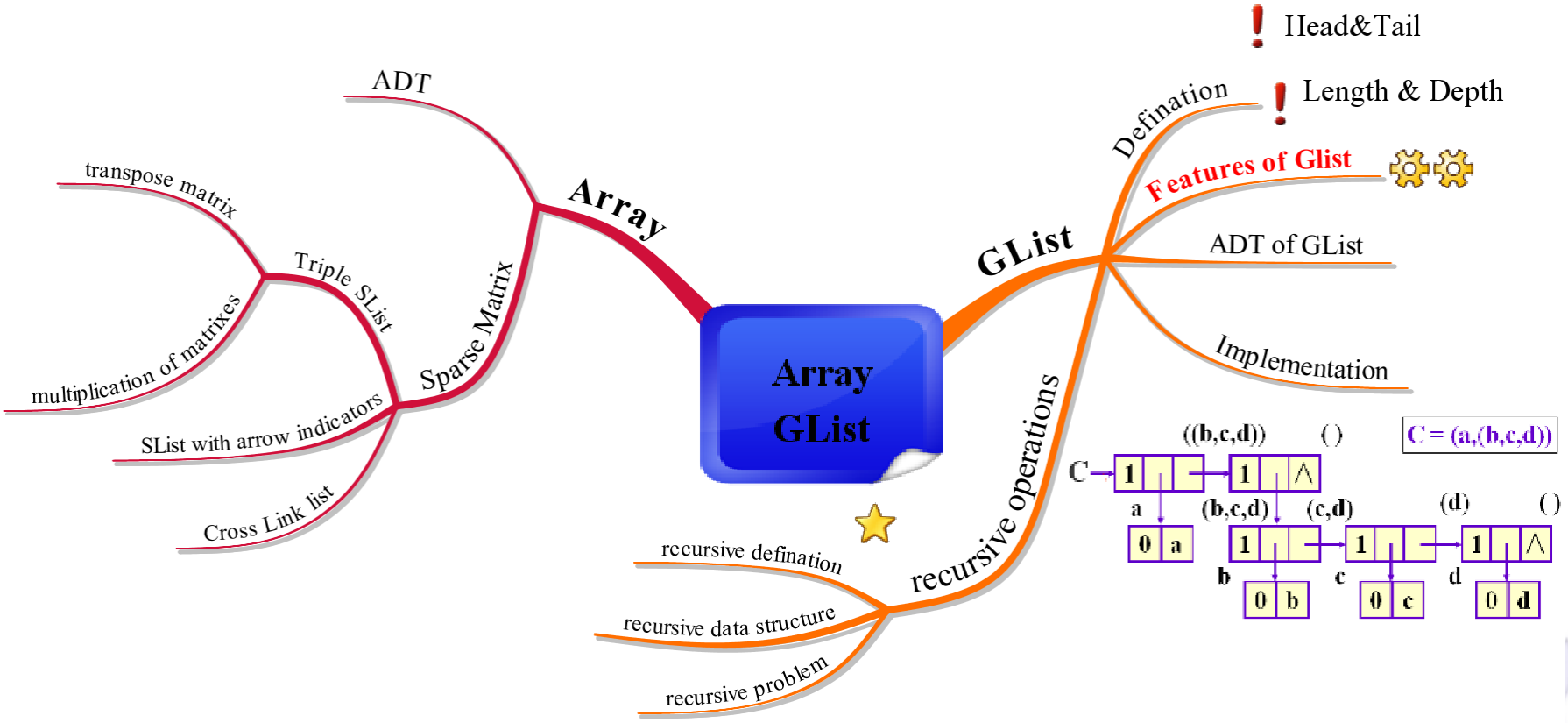
1. 大写字母：建表头结点，保存表名。
2. 左括号：因为输入无语法错误，所以表名后一定是左括号，递归建广义表
3. 逗号: 建立同层的下一个节点
4. 小写字母：建立原子结点
5. 右括号：表示子表结束，退出递归

- 请自己阅读算法4-20(P146)

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
GList
ADT
Sparse Matrix
Triple SList
transpose matrix
multiplication of matrixes
SList with arrow indicators
Cross Link list
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 4