数据结构与算法设计

课程内容

课程内容：数据结构部分

概述	数组与广义表	图	外部排序
线性表	串	查找	
栈与队列	树	内部排序	

课程内容：算法设计部分

概述	贪心	……	计算模型
分治	回溯	……	可计算理论
动态规划	……	……	计算复杂性

Contents 本章内容

数组的概念

数量固定，数据类型相同的（变量）元素组合在一起。

使用一个名称代表它。这个名称就是数组名。

如果要访问其中某个元素(变量)，可以使用元素的索引值(下标)来访问它。

在C语言中，数组元素的索引值从0开始。

```
int A[30][10];        e = A[i][j];
```

4.1 数组的概念

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：

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

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

二维数组是数据元素为线性表的线性表

数组的概念

2.二维数组中的每个元素都受行和列两个线性关系的约束——行关系、列关系

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

在行关系中

a_{ij} 直接前趋是 a_{ij-1}

a_{ij} 直接后继是 a_{ij+1}

在列关系中

a_{ij} 直接前趋是 a_{i-1j}

a_{ij} 直接后继是 a_{i+1j}

N维数组中的每个元素都受N个线性关系的约束

数组的概念

数组的基本操作

初始化操作 InitArray(&A,n,bound1,...,boundn)

销毁操作 DestroyArray(&A)

读元素操作 Value(A,&e,index1,...,indexn)

写元素操作 Assign(&A,e,index1,...,indexn)

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

在C语言中的典型体现：

int A[M][N];

A[i][j] = x; 写

y = A[i][j]; 读

数组的概念

数组的基本操作

1. 读：给定一组下标，读出对应的数组元素；
2. 写：给定一组下标，存储或修改与其相对应的数组元素。

   读/写操作本质上只对应一种操作——寻址。确定指定元素在内存中的物理地址。

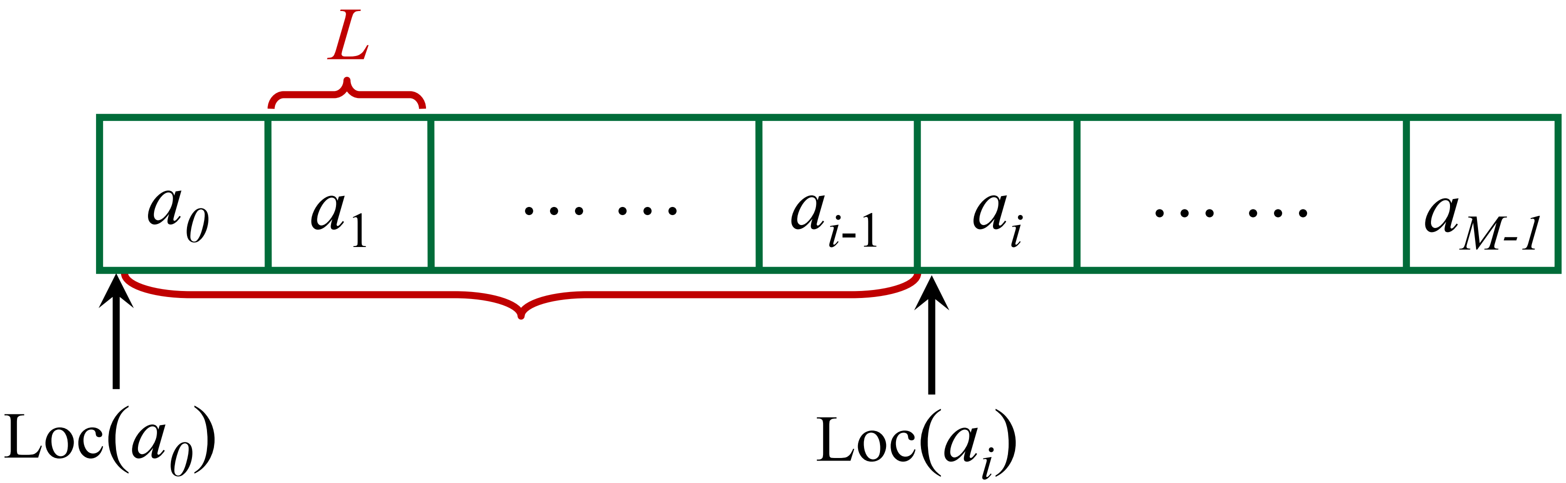
数组的存储

两种形式：既可以是顺序存储，也可以采用链式结构。

数组的存储和实现

数组的存储结构与寻址——一维数组

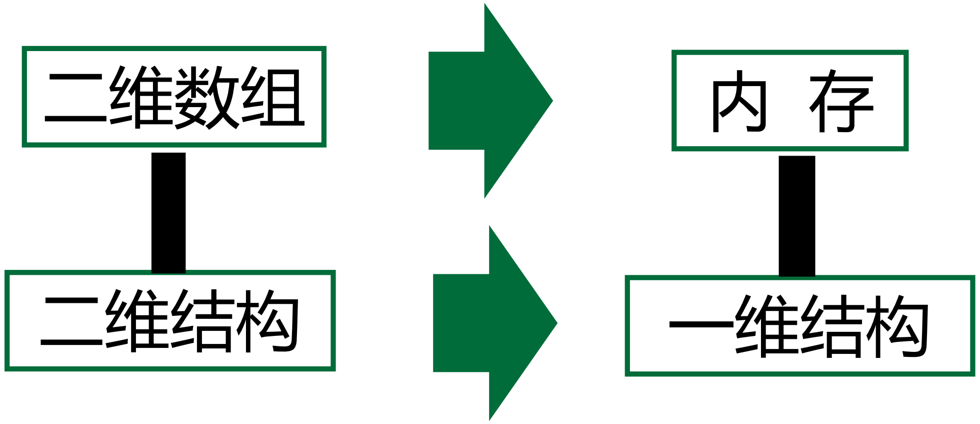
设具有M个元素的一维数组的下标范围为闭区间 [0, M-1]，每个数组元素占用 L 个存储单元。



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

数组的存储和实现

数组的存储结构与寻址——二维数组



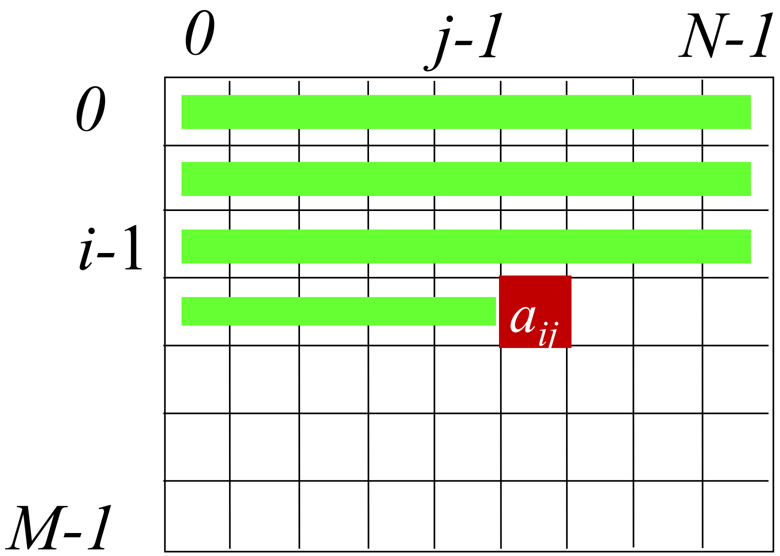
常用的映射方法有两种：

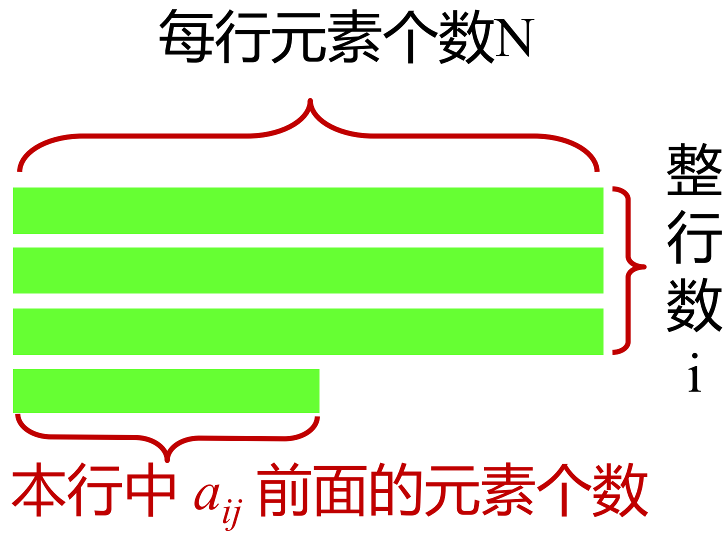
按行优先：先行后列，先存储行号较小的元素，行号相同者先存储列号较小的元素。

按列优先：先列后行，先存储列号较小的元素，列号相同者先存储行号较小的元素。

数组的存储和实现

- 二维数组——按行优先（M×N）



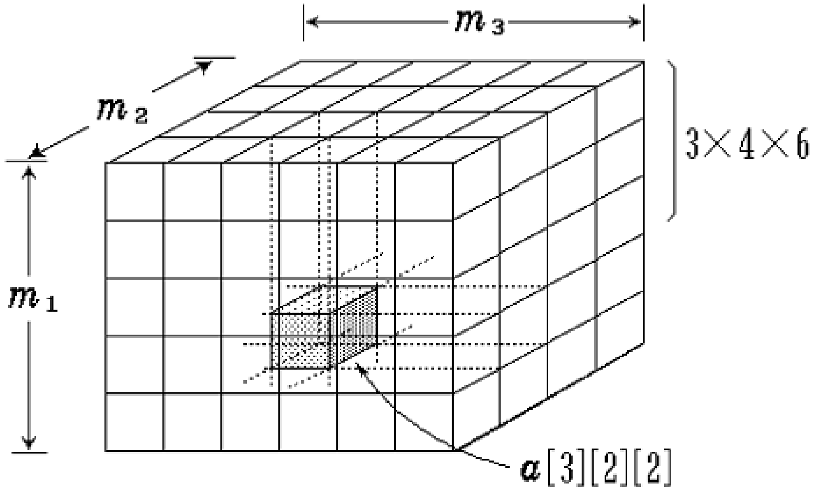


a_{ij}前面的元素个数 = 阴影部分的面积
= 整行数×每行元素个数 + 本行中a_{ij}前面的元素个数
= $i \times \mathrm{N} + j$

$$\mathrm{Loc}(a_{ij}) = \mathrm{Loc}(a_{00}) + (\,\mathrm{N} \times i + j\,) \times L$$

数组的存储和实现

- 三维数组：A[m_1, m_2, m_3]：



$$\mathrm{Loc}(a_{ijk}) = \mathrm{Loc}(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times L$$

数组的存储和实现

N维数组：一般的N维数组：$A[b_1, b_2, \ldots, b_n]$：

$Loc(j_1, j_2, \ldots, j_n)$

$=Loc(0,0,\ldots,0)+(b_2 \times b_3 \times \ldots b_n \times j_1+b_3 \times b_4 \times \ldots b_n \times j_2+\ldots+b_n \times j_{n-1}+j_n)*L$

$$=Loc(0,0,\ldots,0)+\left(\sum_{i=1}^{n-1} j_i \prod_{i+1}^{n} b_k + j_n\right)$$

$$= Loc(0,0,\ldots,0)+\sum_{i=1}^{n} c_i j_i$$

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

数组基址

C_i为常数

数组的存储和实现

- 行优先法

a_{00}	….	a_{0n-1}	a_{10}	….	a_{1n-1}	….	$a_{m-1\ 0}$	….	$a_{m-1\ n-1}$

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

- 二维数组——静态数组表示法

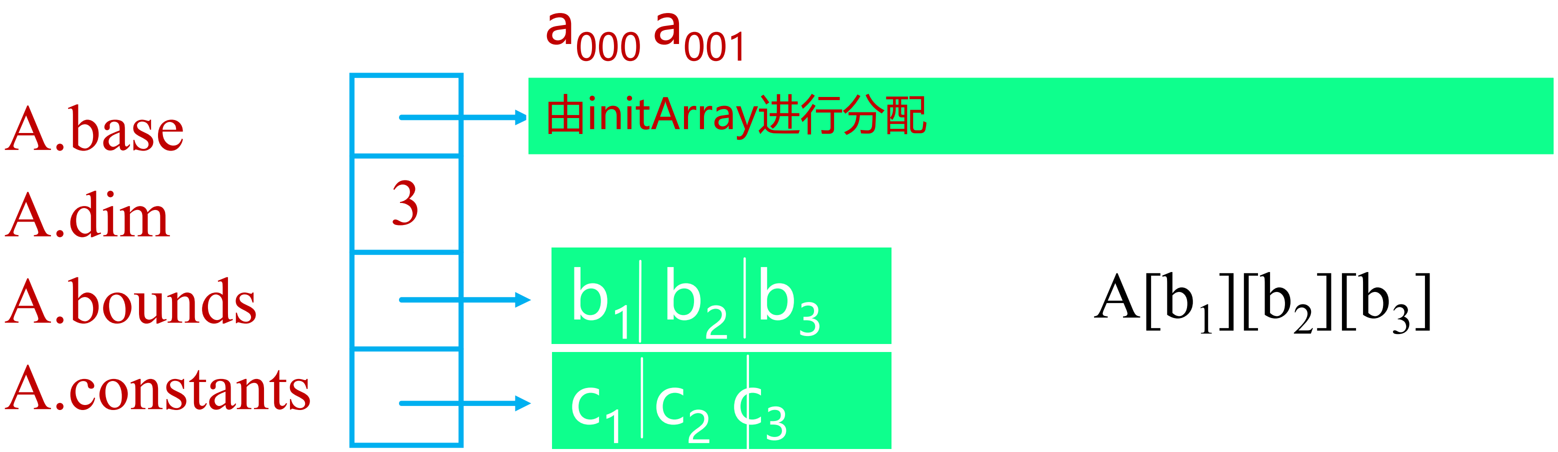
typedef ElemType Array[m*n];

(m和n都是常量)

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
    int  *constants;         // 数组映像函数常量基址
} Array;
```



矩阵的压缩存储

4.3.1 特殊矩阵的压缩存储

4.3.2 稀疏矩阵的压缩存储

矩阵的压缩存储-特殊矩阵

值相同元素或者非零元素的分布有一定规律的矩阵，称为特殊矩阵。

对称矩阵、 上（下）三角矩阵。

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

矩阵的压缩存储-特殊矩阵

- 对称矩阵/上（下）三角矩阵

用一维数组，按行优先存储下三角元素。

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & \ldots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & a_{13} & \ldots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & a_{23} & \ldots & a_{2n-1} \\ \ldots & & & & & \\ a_{n-10} & a_{n-11} & a_{n-12} & a_{n-13} & \ldots & a_{n-1n-1} \end{pmatrix}$$

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

对于下标i, j，线性地址

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

0	1	2	3	4	5	n(n+1)/2				
a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	…	a_{n-10}	a_{n-11}	…	a_{n-1n-1}

矩阵的压缩存储-稀疏矩阵

含有较多值相同元素或较多零元素，且值相同元素或者零元素分布没有一定规律的矩阵称为稀疏矩阵。

讨论含有较多零元素的稀疏矩阵的压缩存储。

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

M有42（6×7）个元素，有8个非零元素

如何进行稀疏矩阵的压缩存储?

矩阵的压缩存储-稀疏矩阵

采用三元组存储：（行, 列, 值)

三元组顺序表

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

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

加上矩阵的行数和列数：6,7

矩阵的压缩存储-稀疏矩阵

```
#define MAXSIZE 12500
 typedef struct {
      int   i,j;              // 非零元的行下标和列下标
      ElemType  e;      // 非零元值
 } Triple;
 typedef struct {
     Triple data [ MAXSIZE];
                                    // 用于存储三元组表
      int   mu,nu,tu;  // 行数、列数和非零元个数
 } TSMatrix;
```

矩阵的压缩存储-稀疏矩阵

按行（行内按列）顺序存储非零元素。

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

M.data

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

M.mu	6
M.nu	7
M.tu	8

矩阵的压缩存储-稀疏矩阵

三元组表的顺序存储——转置算法

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

基本算法：交换对应行、列位置上的元素。

矩阵的压缩存储-稀疏矩阵

一般矩阵的转置算法

```
int  a[m][n],b[m][n];
for ( i =0; i<m; ++i )
    for (j = 0; j<n; ++j )
        b[ j ][ i ] = a[ i ][ j ];
```

算法的时间复杂度为：O(m*n)

矩阵的压缩存储-稀疏矩阵

M.data

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

M.mu	6
M.nu	7
M.tu	8

M.data

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

M.mu	6
M.nu	7
M.tu	8

矩阵的压缩存储-稀疏矩阵

转置运算算法

TransposeSMatrix(TSMatrix M, TSMatrix &T)

- 基本思想

  对 M.data 从头至尾扫描：

  第一次扫描时，将 M.data 中列号为 0 的三元组赋值到 T.data 中；

  第二次扫描时，将 M.data 中列号为 1 的三元组赋值到 T.data中；

  依此类推，直至将 M.data 所有三元组赋值到T.data 中。

矩阵的压缩存储-稀疏矩阵

三元组表的顺序存储——转置算法

第七次扫描查找第7列元素

M.data

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

T.data

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

矩阵的压缩存储-稀疏矩阵

```
Status TranMatrix( TSMatrix M, TSMatrix &T )
{  T.mu=M.nu;  T.nu=M.mu;  T.tu=M.tu;
   if ( T.tu )    // 非零元素个数!=0
   {  q=0;  // q为当前三元组在T.data[ ]存储位置(下标)
      for ( col=0; col<M.nu;  ++col )
         for (p=0;p<M.tu;++p) //p为扫描M.data指示器
              if ( M.data[p].j= =col )
              {  T.data[q].i = M.data[p].j;
                  T.data[q].j = M.data[p].i;
                  T.data[q].e = M.data[p].e;  ++q;
              }
   } // if
   return OK;
} // TranMtrix
```

算法的时间复杂度：O(nu*tu)

时间复杂度分析

- 转置算法 TranMatrix 的时间复杂度为O(nu×tu)
- 当非零元的个数tu和矩阵元素个数 mu×nu 同数量级时，转置运算算法的时间复杂度为 O(nu×mu×nu)，反而比原本的（mu×nu）更坏。
- 因此，该算法一般用于 tu << mu×nu 的情况

矩阵的压缩存储-稀疏矩阵

提高算法效率,增加两个数组num和cpos

num[col]：存储M第 col 列非零元个数

cpos[col]：存储M第 col 列第一个非零元在T.data 中的位置

cpos[col]的计算方法：

$$cpos[0] = 1$$

$$cpos[col] = cpos[col-1] + num[col-1] \quad 2 \leq col \leq n$$

col	0	1	2	3	4	5	6
num[col]	2	2	2	1	0	1	1
cpos[col]	1	3	5	7	8	8	9

矩阵的压缩存储-稀疏矩阵

col	0	1	2	3	4	5	7
num[col]	2	2	2	1	0	1	0
cpot[col]	3	5	7	8	8	9	9

第2列第二个

第1列第二

第0列第一个非零元在b中的位置

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

M.data

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

T.data

第2列第一个非零元在b中的位置

扫描M.data实现M到T 的转置

矩阵的压缩存储-稀疏矩阵

```
Status FastTransMatrix(TSMatrix M, TSMatrix &T)
{//采用三元组顺序表存储稀疏矩阵，求M的转置矩阵T
  T.mu=M.nu; T.nu=M.mu; T.tu=M.tu;
  if ( T.tu )
  {  for ( col=0; col<=M.nu-1; ++col )
         num[col]=0;
     // 求M中每一列非零元个数
     for ( t=1; t<=M.tu; ++t )
         ++ num[ M.data[t].j ];
     //求第 col列中第一个非零元在T.data中的序号
     cpot[0] = 1;
     for ( col=0; col<=M.nu;  ++col )
         cpot[col] = cpot[col] + num[col];
     …
```

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

```
    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 ];
    } // for
  }  // if
  return OK;
}  // FastTranMatrix
```

- 时间复杂度分析

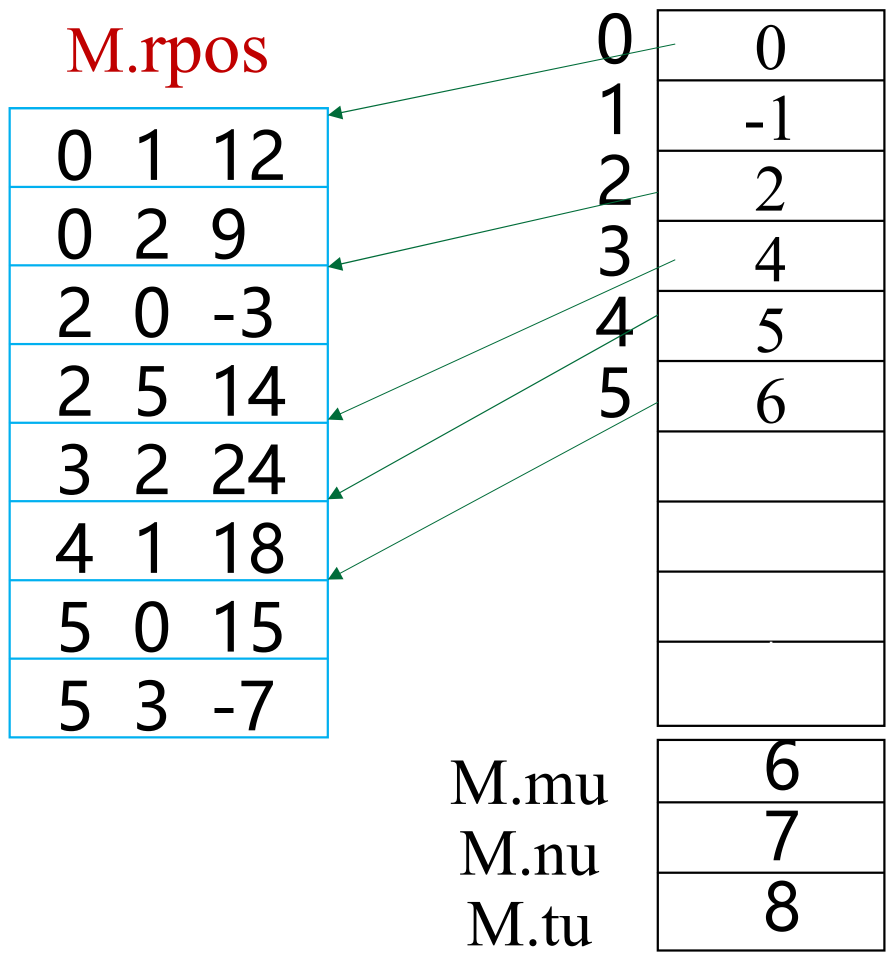
算法中有四个并列的单循环，循环次数分别为 nu、tu、nu 和 tu，时间复杂度为：

$$O(2nu+2tu)$$

矩阵的压缩存储-稀疏矩阵

借助上述思想，可以采用行逻辑链接顺序表来获取任何一个非零元素的值

```
#define  MAXMN  500
typedef struct {
    Triple  data[ MAXSIZE ];
    int  rpos[MAXMN];
    // 每行第一个元素的位置表
    int  mu, nu, tu;
} RLSMatrix;
```



矩阵的压缩存储-稀疏矩阵

例：给定一组下标，求矩阵指定元素的值

```
ElemType value(RLSMatrix M, int r, int c)
{
   p = M.rpos[r];
   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].e;
   else return 0;
}
```

矩阵的压缩存储-稀疏矩阵

稀疏矩阵的链式存储——十字链表

　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：

<table>
<tr><td colspan="2">row</td><td colspan="2">col</td><td colspan="2">item</td></tr>
<tr><td colspan="3">down</td><td colspan="3">right</td></tr>
</table>

row：　存储非零元素的行号

col：　存储非零元素的列号

item：　存储非零元素的值

right：指针域，指向同一行中的下一个三元组

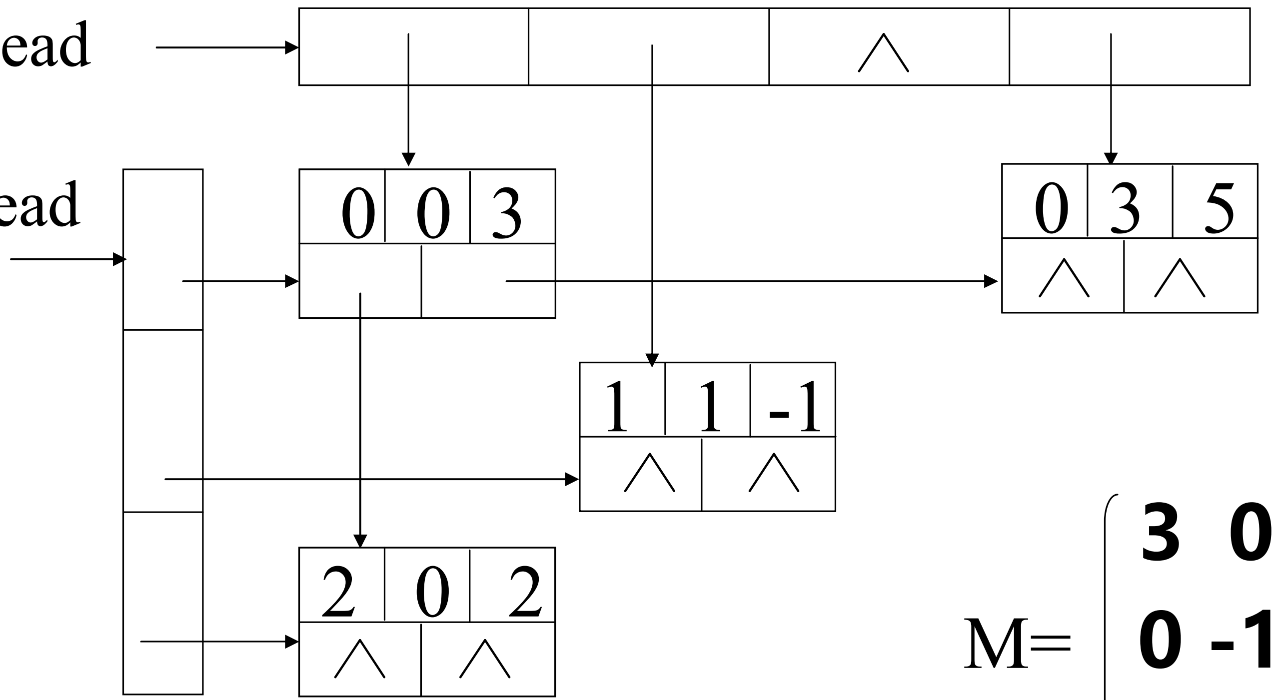
down：指针域，指向同一列中的下一个三元组

矩阵的压缩存储-稀疏矩阵

稀疏矩阵的链式存储——十字链表

列表头，M.chead

行表头，M.rhead



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

矩阵的压缩存储-稀疏矩阵

十字链表的类型定义

```
typedef struct OLNode {
    int  row, col;       // 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   mu, nu, tu;  // 行数、列数和非零元个数
} CrossList;
```

广义表的定义

- **广义表(generalized list)的概念**

  广义表是一种不同构的线性结构，LS = (α_1，α_2，…，α_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。

广义表的定义

广义表的术语

$$LS = (\alpha_1, \underbrace{\alpha_2, \cdots, \alpha_n})$$

表名　表头head　表尾tail　n是表长

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。

广义表的定义

例:

$A = (\ )$ 空表

$B = (b, c, d)$

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

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

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

广义表的定义

例:

A = ( )　空表　　表长：0；深度：1

B = (b, c, d)　　表长：3，深度：1

C = (a, B) = (a, (b,c,d))　　表长：2，深度：2

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

共享表　　表长：3，深度：3

E = (a,E) = (a, (a,E)) = (a, (a, (a, ...)))

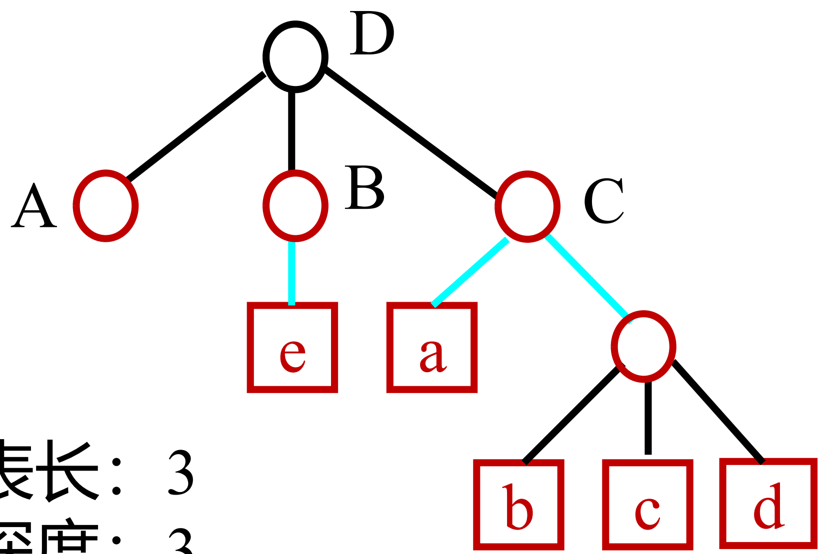
递归表

广义表的定义

广义表的图形化表示

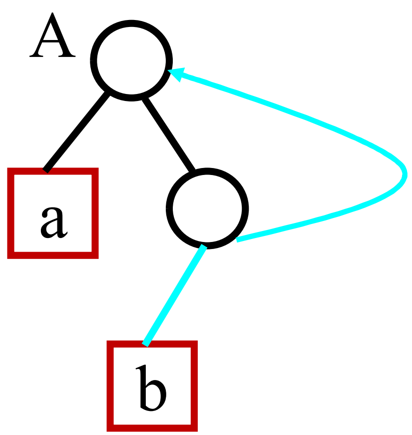
用○表示子表
用□表示原子

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



表长：3
深度：3
深度 = 括号的重数
= ○ 结点的层数

A = (a， (b, A))



表长：2
深度：∞

广义表的定义

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。

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)) = ( )

广义表的定义

广义表的基本操作

1） 创建空的广义表L；
2） 销毁广义表L；
3） 已有广义表L，由L复制得到广义表T；
4） 求广义表L的长度；
5） 求广义表L的深度；
6） 判广义表L是否为空；
7） 取广义表L的表头；
8） 取广义表L的表尾；
9） 在L中插入元素作为L的第一个元素；
10） 删除广义表L的第一个元素，并e用返回其值；
11） 遍历广义表L，用函数visit( )处理每个元素；

广义表的存储结构

链表存储方式

广义表中的数据元素可能为**原子**或**列表**，由此需要两种结点：

表结点：用以表示广义表；

原子结点：用以表示原子。

tag	ptr	
1	hp	tp

表结点

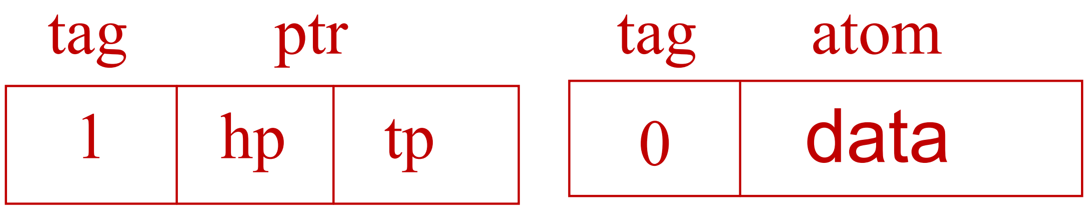
tag	atom
0	data

原子结点

广义表的存储结构

结点的类型定义

```
Typedef  enum { ATOM, LIST } ElemTag;
                              // ATOM==0:原子，LIST==1: 列表
Typedef struct  GLNode {
      ElemTag  tag;              // 标志域
      union {
          AtomType  atom;    // 原子结点的值域
          struct  {struct  GLNode  *hp,*tp;} ptr;
         // 表结点的指针域：ptr.hp指表头，ptr.tp指表尾
      };
} * Glist;
```



表结点　　原子结点

广义表的存储结构

广义表存储方式

空表 $ls=NULL$ → | tag=1 | ∧ | ∧ |

非空表 ls → | tag=1 | | |

指向表尾的指针

指向表头的指针

若表头为原子，则为 | tag=0 | data |

否则，依次类推。

广义表的存储结构

广义表(α_1, α_2 …α_n)的两种分解方法

广义表：表头+表尾

广义表 L= (α_1, α_2 …α_n)

表头: α_1

表尾: (α_2 , …,α_n)

广义表：子表1 +子表2 + ··· + 子表n

广义表 L= (α_1, α_2 …α_n)

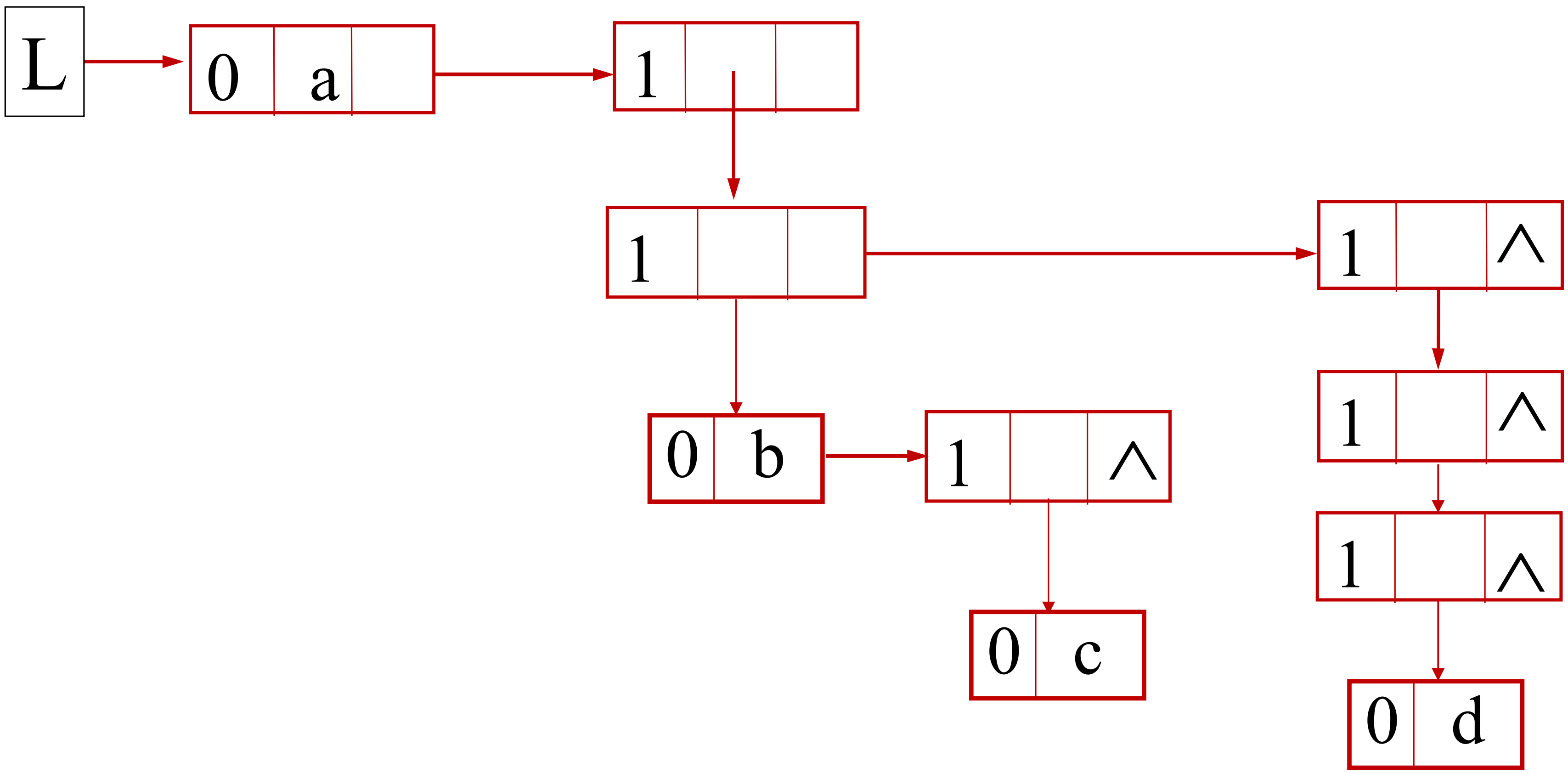
子表: α_1 α_2 α_3 …α_n

广义表的存储结构

头尾表构造法

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

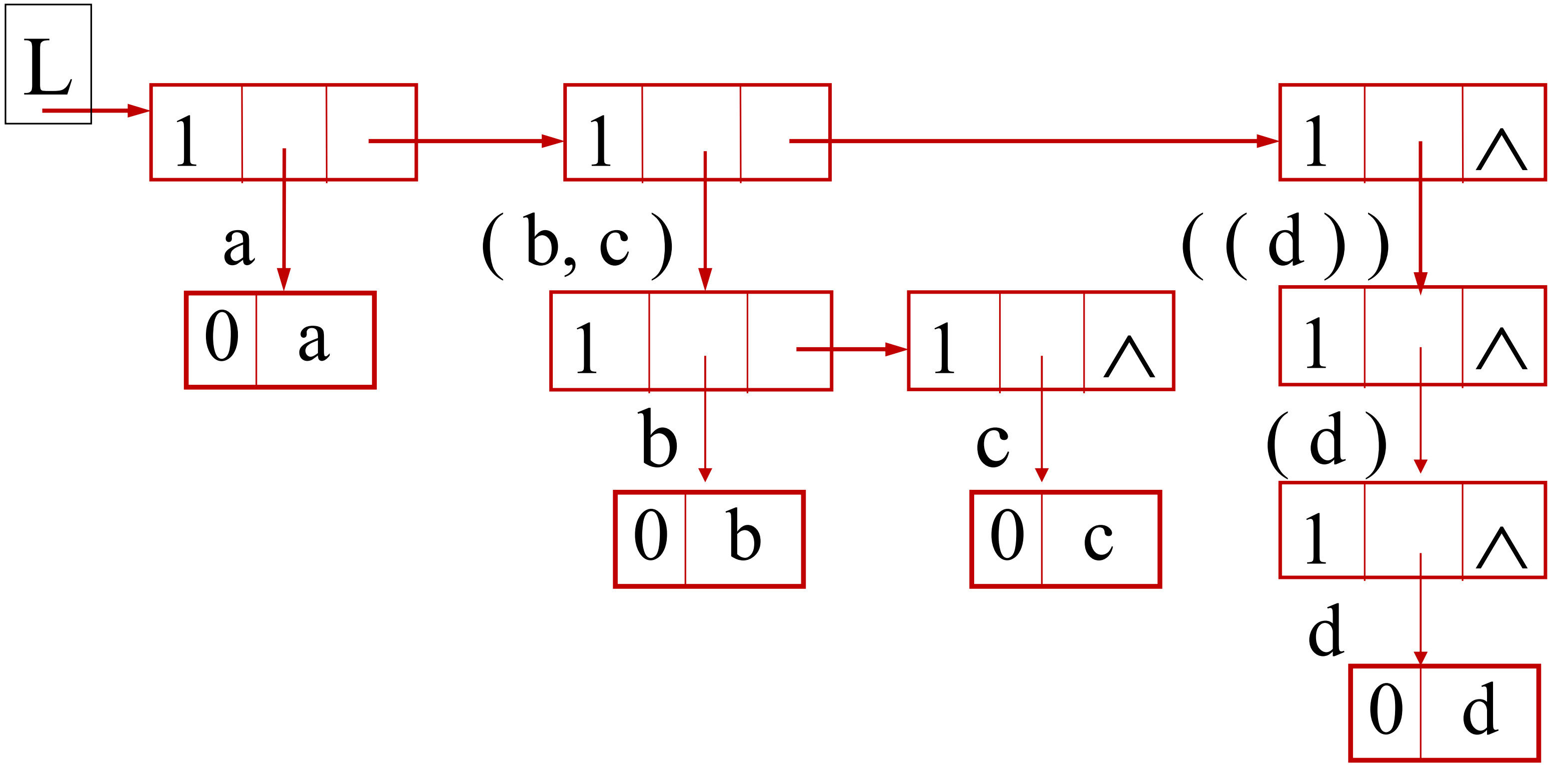
(d)



广义表的存储结构

子表构造法

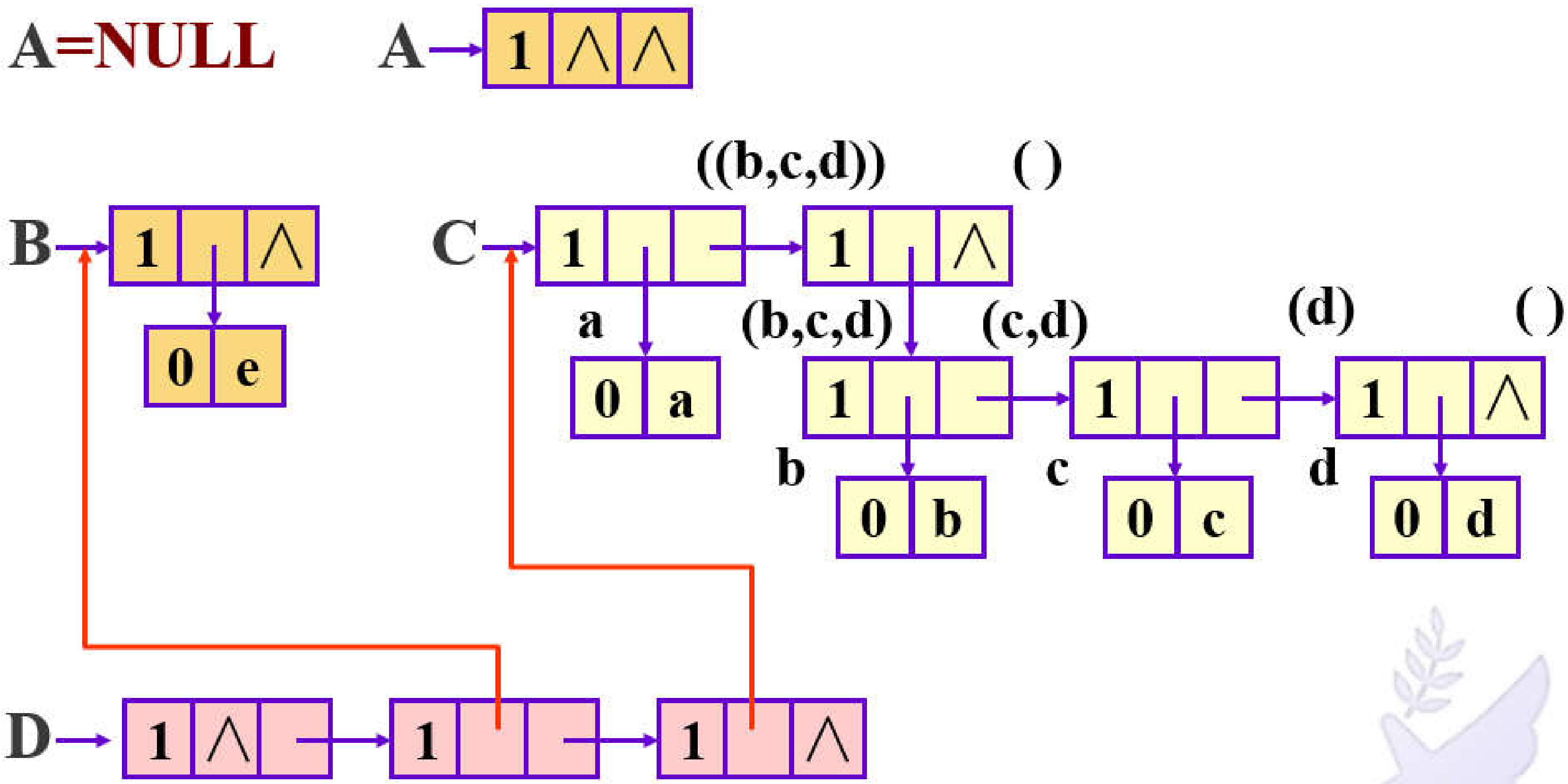
$$L = (a , (b, c) , ((d)))$$



广义表的存储结构

广义表存储方式

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



广义表操作的实现

广义表是递归结构，所以广义表的许多操作可以用递归算法实现。

递归函数

一个含直接或间接调用本函数语句的函数被称之为递归函数，它必须满足以下两个条件：

1. 在每一次调用自己时，必须是（在某种意义上）更接近于解;
2. 必须有一个终止条件。

广义表操作的实现

递归算法的基本思路——分治法

对于一个输入规模为 n 的函数或问题，用某种方法把输入分解成 k($1<k\leq n$)个子集，从而产生 k 个子问题，分别求解这 k 个问题，得出 k 个问题的子解，再用某种方法把它们组合成原来问题的解。

若子问题规模还相当大，则可以反复使用分治法，直至最后所分得的子问题足够小，以至可以直接求解为止。

广义表操作的实现

求广义表的深度 GListDepth(L)

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

广义表操作的实现

求广义表的深度 GListDepth(L)

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

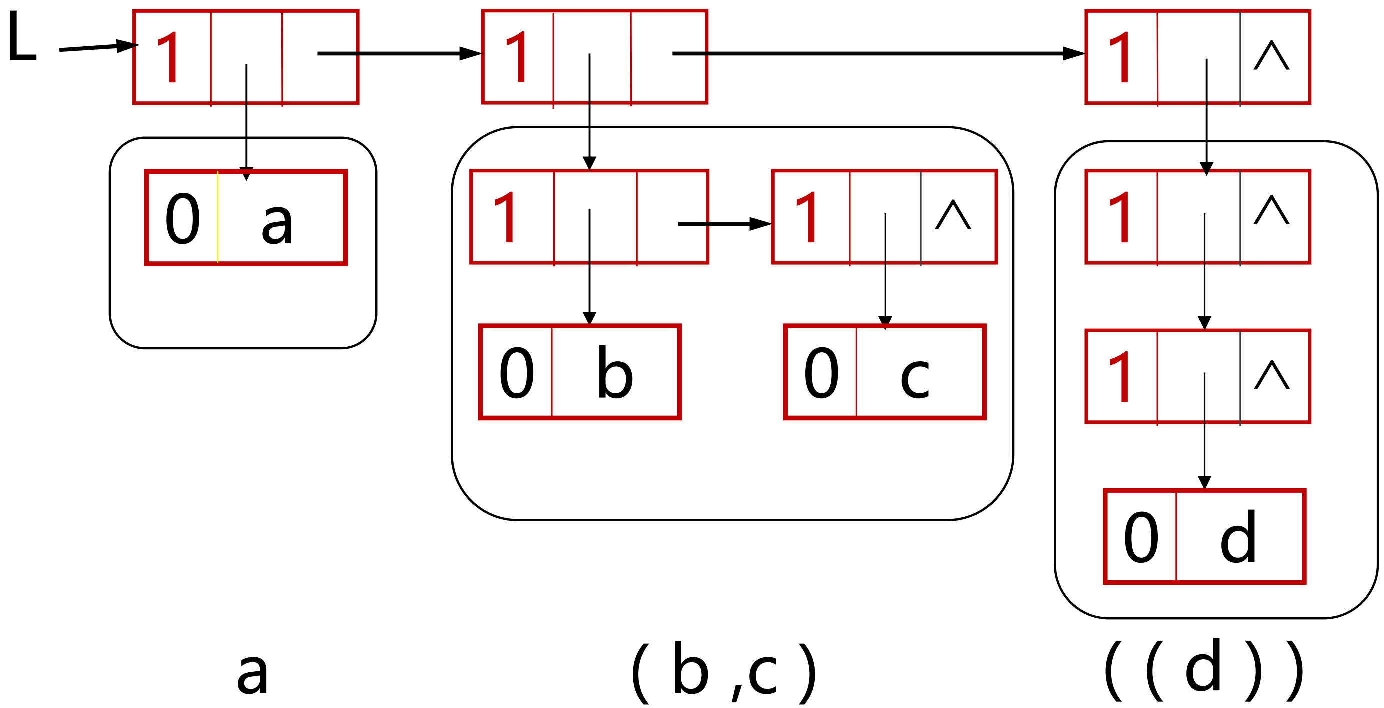
空表：深度 = 1

原子：深度 = 0

广义表操作的实现

求广义表的深度 GListDepth(L)

L = (a , (b ,c) , ((d))) 的深度



广义表操作的实现

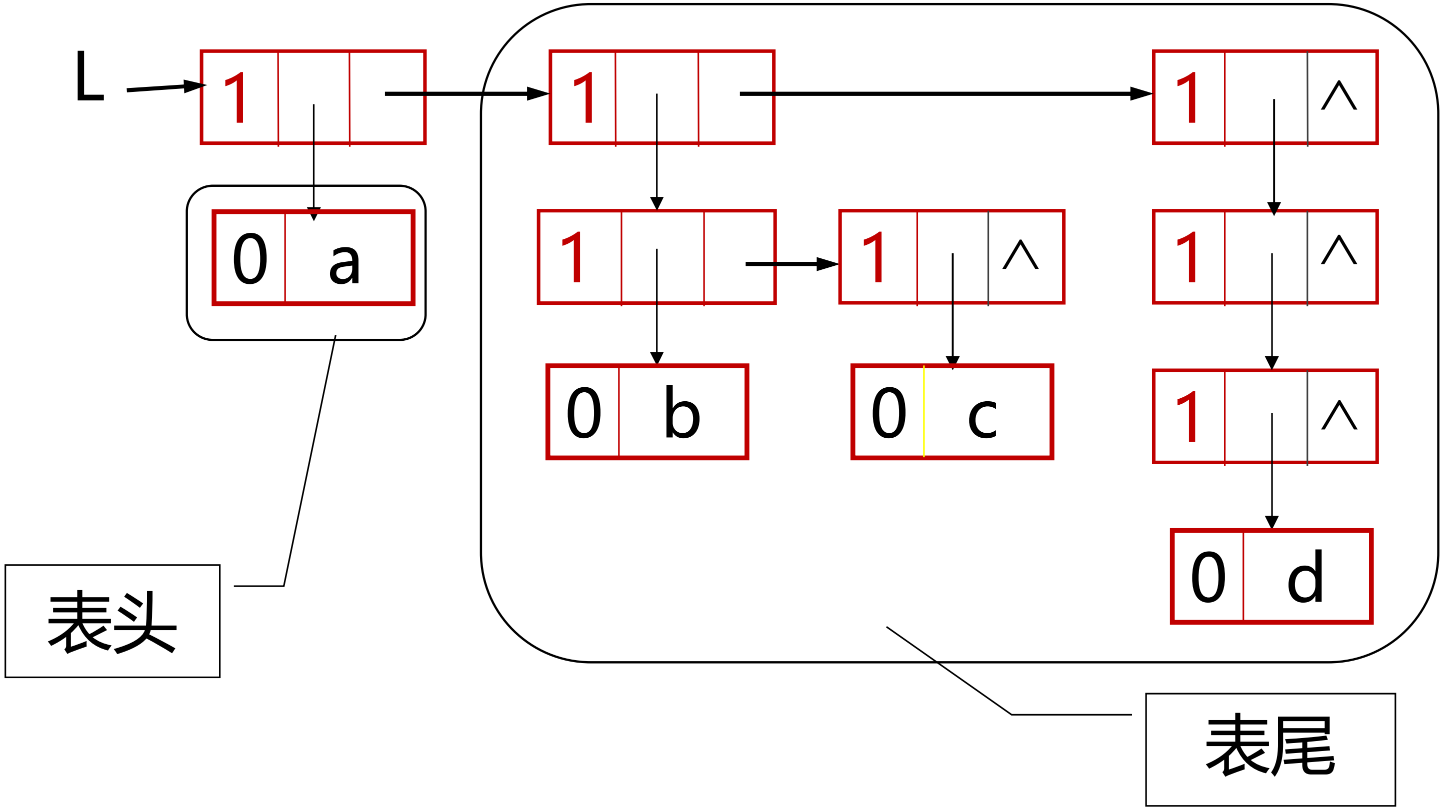
求广义表的深度 GListDepth(L)

```
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;
}
```

广义表操作的实现

复制广义表 CopyGList(T,L)

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



广义表操作的实现

```
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;
     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
     }
   }
}
```

广义表操作的实现

建立广义表

输入：字符串 (α_1, α_2, …, α_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α_1,α_2,…,α_n，分别建立 α_1, α_2, …, α_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

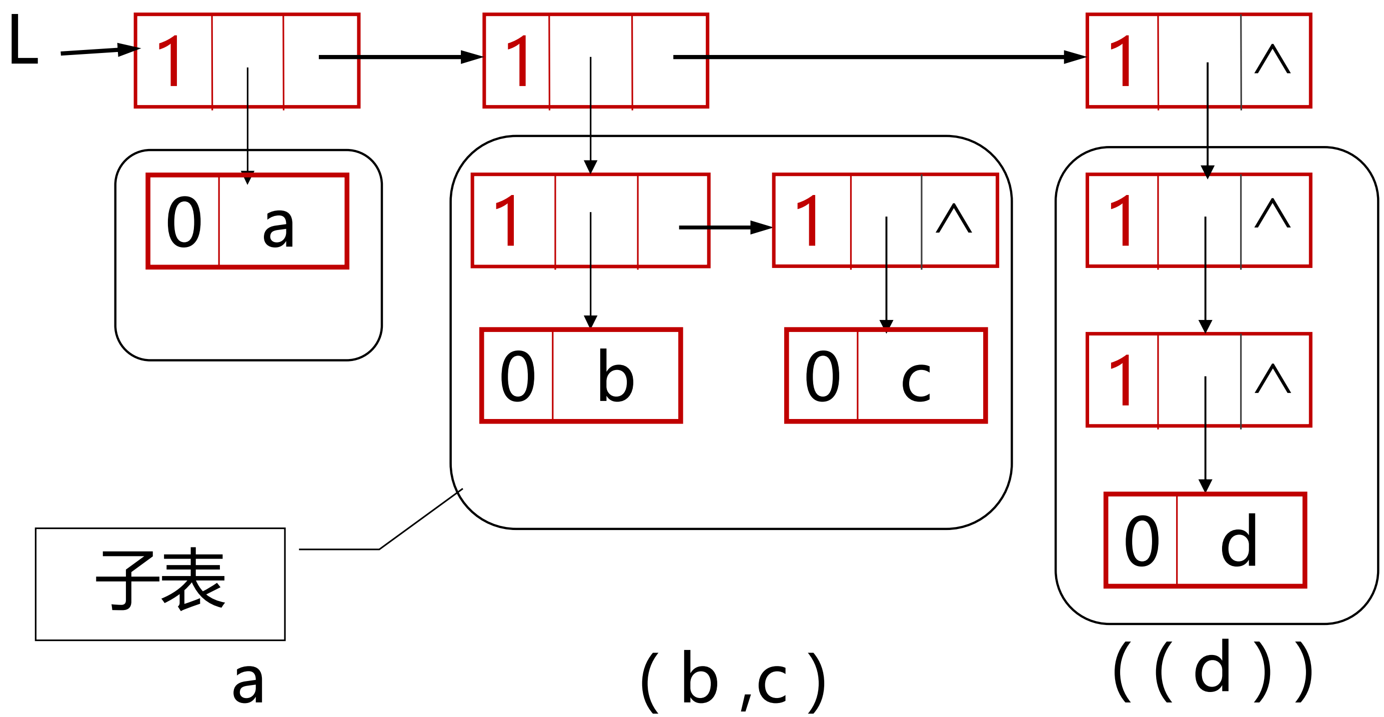
原子: 建立原子结点

广义表操作的实现

建立广义表

子表和广义表的关系

相邻两个子表之间的关系



广义表操作的实现

```
void CreateGList( GList &L, char str[ ] )
{ if ( strcmp( str,"()  " )==0)  L=NULL;   //空表
  else
  {  if(strlen(str)==1) { //建原子结点
          L=(GList)malloc(sizeof(GLNode));
          L->tag=ATOM; L->atom=str[0];
     }
     else {  //非空表，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
```

广义表操作的实现

```
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->tag = LIST;
       p = p->ptr.tp;
    }  //if
} while(!strempty(sub));
p->ptr.tp = NULL;   //最后一个子表的表结点
```

广义表操作的实现

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;

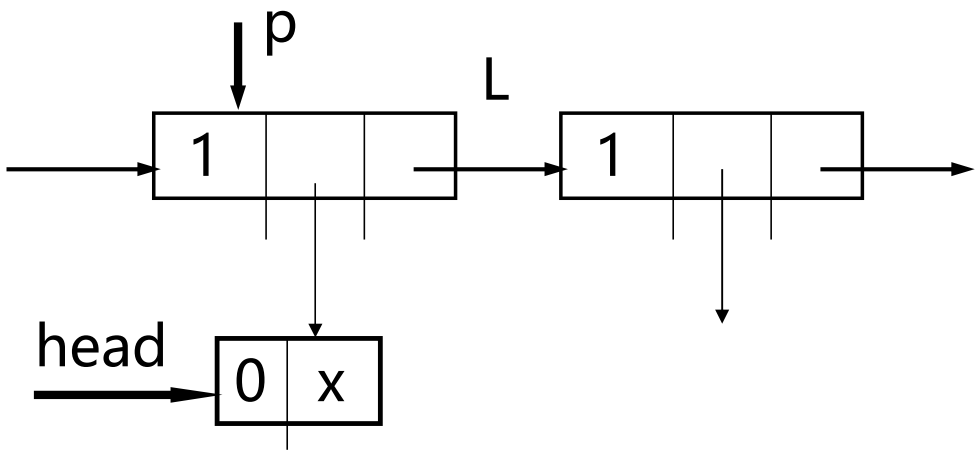
2)删除时，不仅要删除原子结点，还需要删除相应的表结点。

5.4.3 广义表操作的实现

```
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
      else
      {            }    // 第一项没有被删除的情况
   }
}  // Delete_GL
```

5.4.3 广义表操作的实现

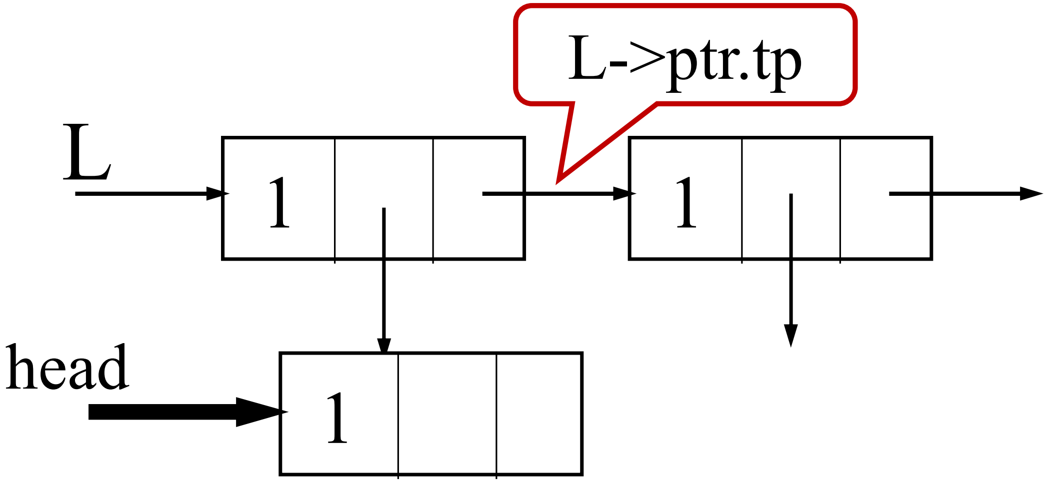
void Delete_GL(Glist&L, AtomType x)




```
p=L;  L = L->ptr.tp;  // 修改指针
free(head);           // 释放原子结点
free(p);              // 释放表结点
Delete_GL(L, x);      // 递归处理剩余表项
```

5.4.3 广义表操作的实现

void Delete_GL(Glist&L, AtomType x)




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

5.4.3 广义表操作的实现

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

5.4.3 广义表操作的实现

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

5.4.3 广义表操作的实现

```
Status InsertFirst_GL(GList &L, GList e)
{
        p = (GList) malloc( sizeof(GLNode) );
        if ( !p )
           exit( OVERFLOW );
        p->tag = LIST;
        p->ptr.hp = e;
        p->ptr.tp = L;
        L = p;
        return OK;
}
```

5.4.3 广义表操作的实现

```
Status DeleteFirst_GL( GList &L, GList &e )
{
        if ( !L )
            return ERROR;
        p = L;
        e = L->ptr.hp;
        L = L->ptr.tp;
        free( e );
        free( p );
        return OK;
}
```

5.4.3 广义表操作的实现

广义表的特殊形态

在广义表中，若任意一个元素(原子、子表)只能在广义表中出现一次，称为纯表。

线性表是只包含原子的纯表。

若存在共享元素（原子或子表在表中出现多次），则称为再入表。

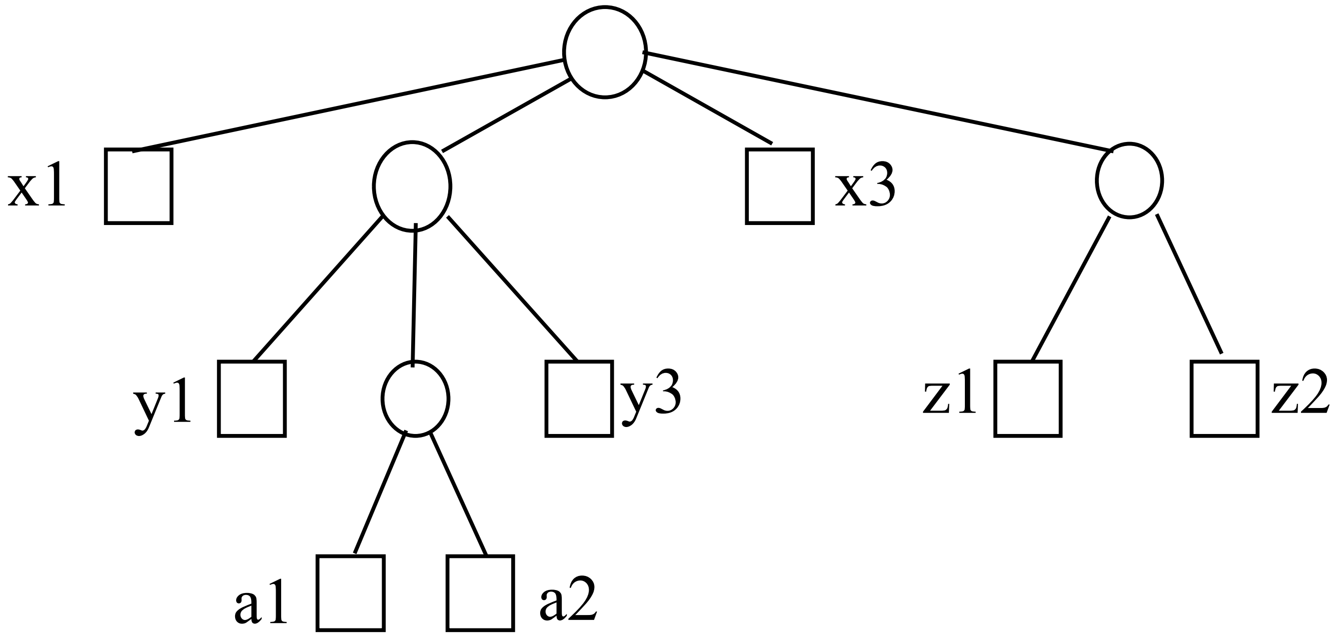
递归表是有回路的再入表。

5.4.3 广义表操作的实现

广义表的特殊形态

纯表：对应一棵树。

（x1，(y1，(a1，a2)，y3)，x3，(z1，z2)）

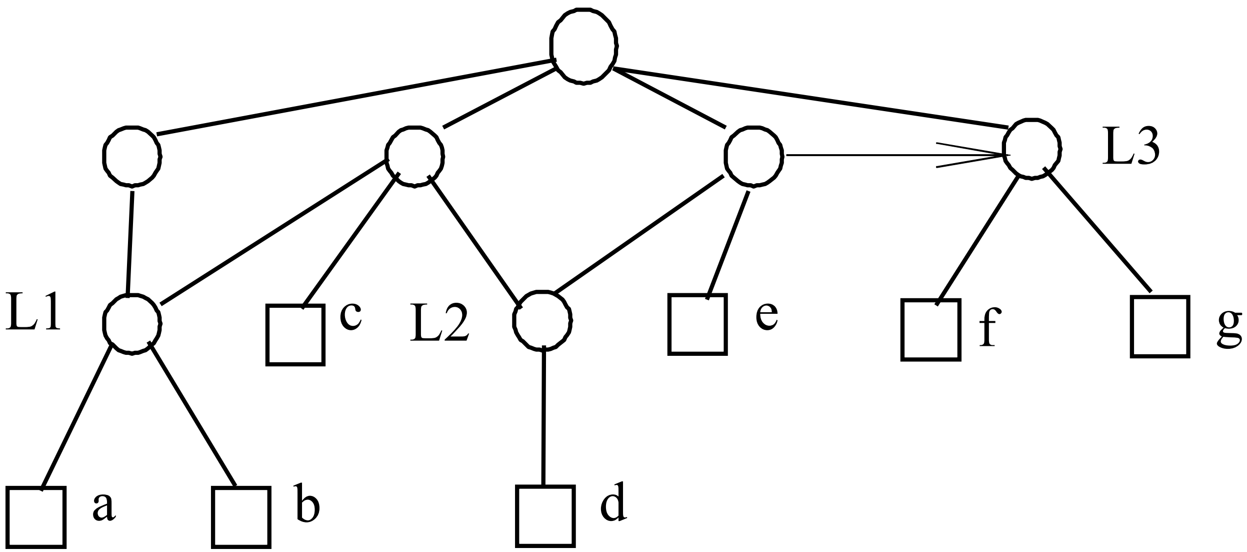


5.4.3 广义表操作的实现

广义表的特殊形态

如果再入表中没有回路：对应一个DAG(有向无环图)。

(((a, b))，((a,b),c,(d))，((d), e, (f, g))，(f ,g))



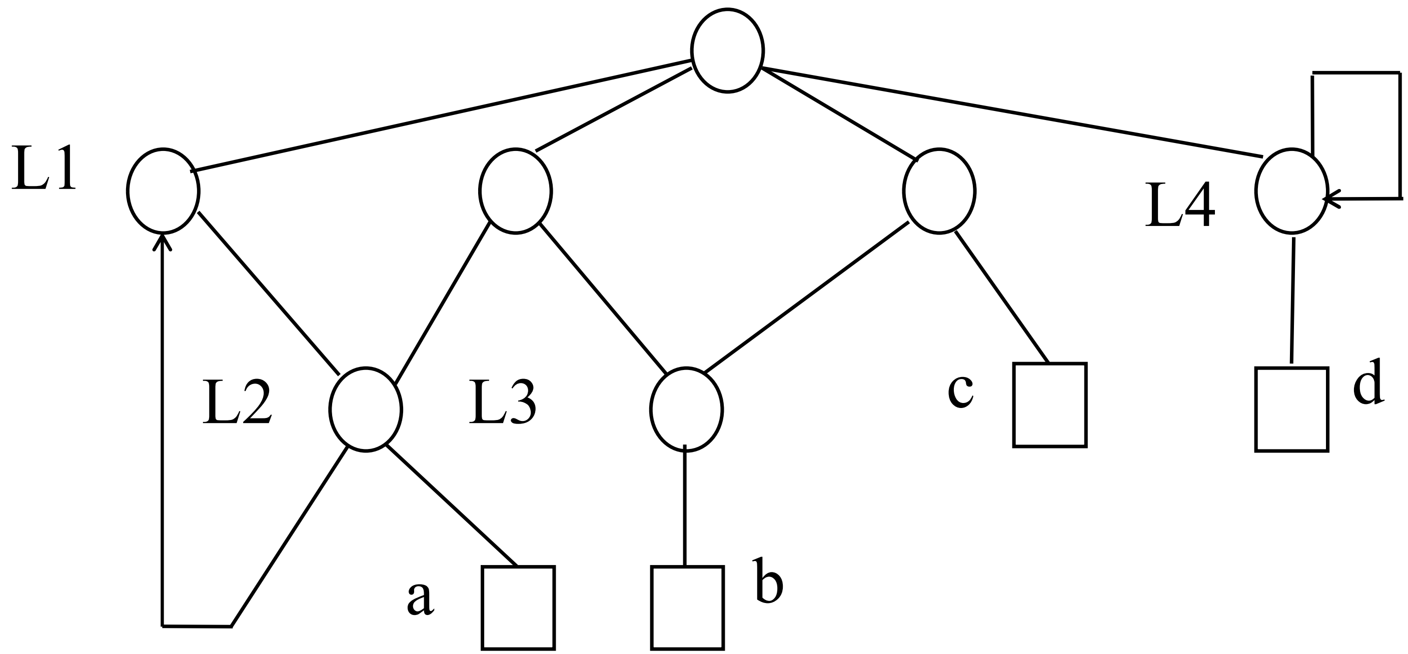
(L1:(a,b), (L1, c , L2:(d)), (L2, e, L3:(f,g)), L3)

5.4.3 广义表操作的实现

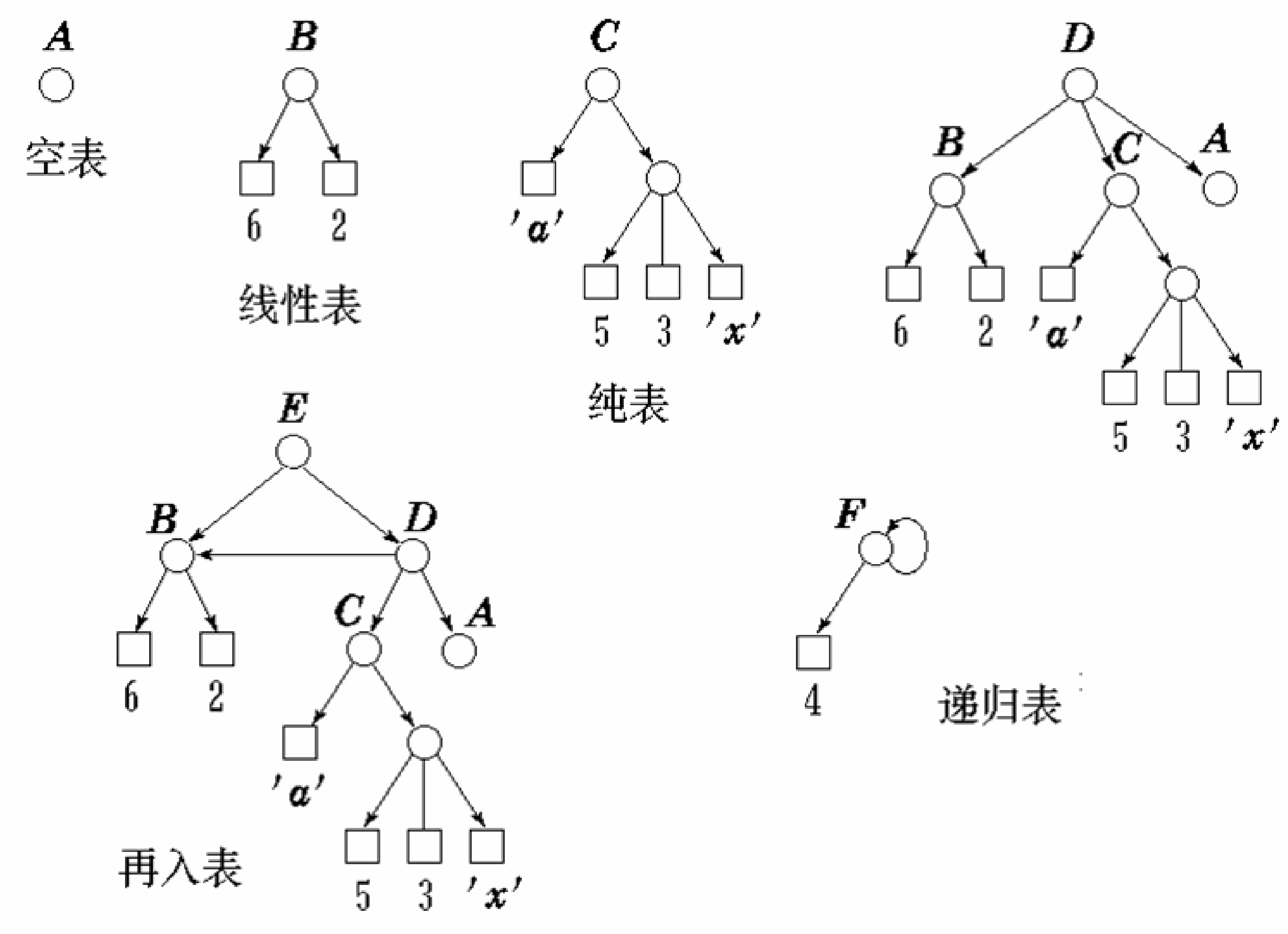
广义表的特殊形态

循环表(cyclic list，递归表)对应有向图。

(L1:(L2:(L1, a)), (L2, L3:(b)), (L3, c), L4:(d,L4))



5.4.3 广义表操作的实现



5.4.3 广义表操作的实现

广义表的特殊形态

图⊇递归表⊇再入表⊇纯表(树)⊇线性表

广义表是线性与树型结构的推广。

广义表应用

函数的调用关系

内存空间的引用关系

LISP语言

5.4.1 广义表的定义

练习:

1. GetTail (b, k, p, h) = ___(k, p, h)___;
2. GetHead ((a, b), (c, d)) = ___(a,b)___ ;
3. GetTail ((a, b), (c,d)) = ___((c, d))___ ;
4. GetTail (GetHead ((a,b), (c,d))) = ___(b)___ ;
5. GetTail (e) = ___( )___;
6. GetHead (( )) = ___( )___;
7. GetTail (( )) = ___( )___。

学习要点

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

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。