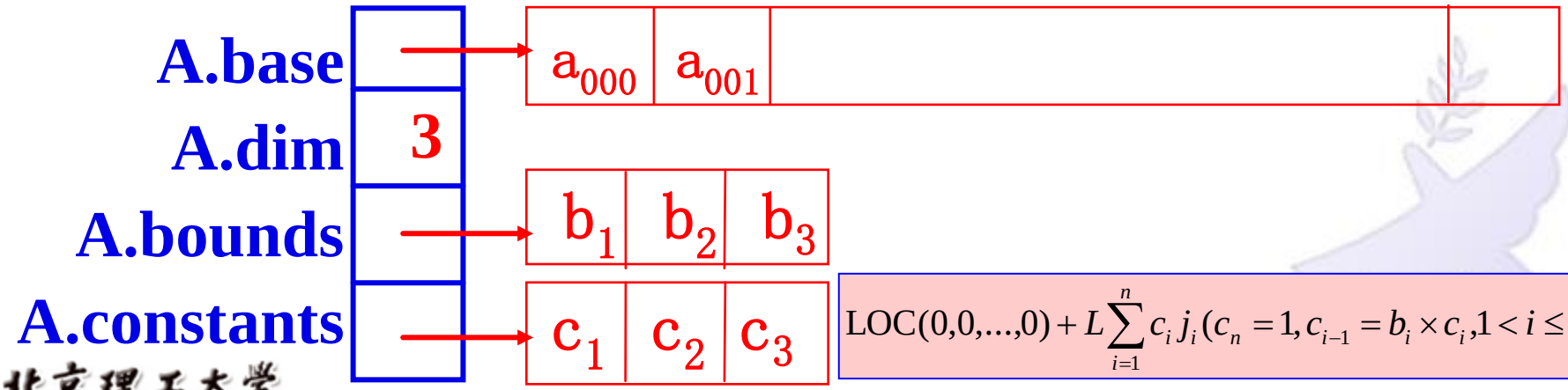
数组的存储和实现

- **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()
```

三元组顺序表的定义

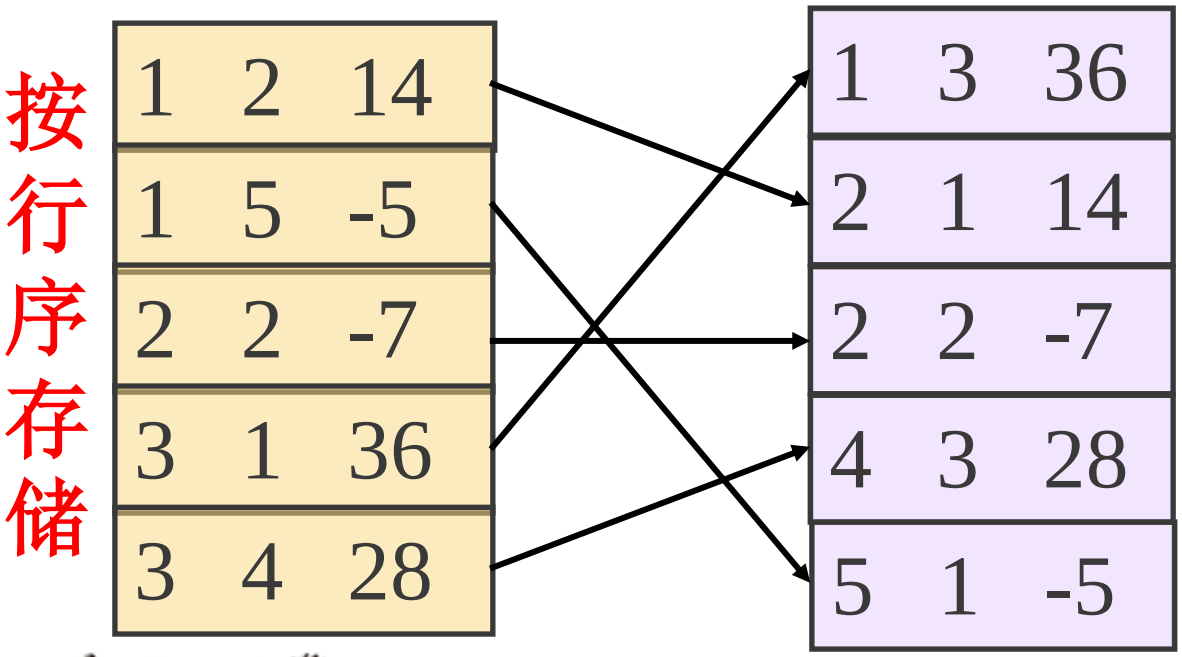
```
#define  MAXSIZE  12500
typedef struct {
    int  i, j;      //该非零元的行row下标和列col下标
    ElemType  e;    // 该非零元的值
} Triple;  // 三元组类型
```

```
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
```

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

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)

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)

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

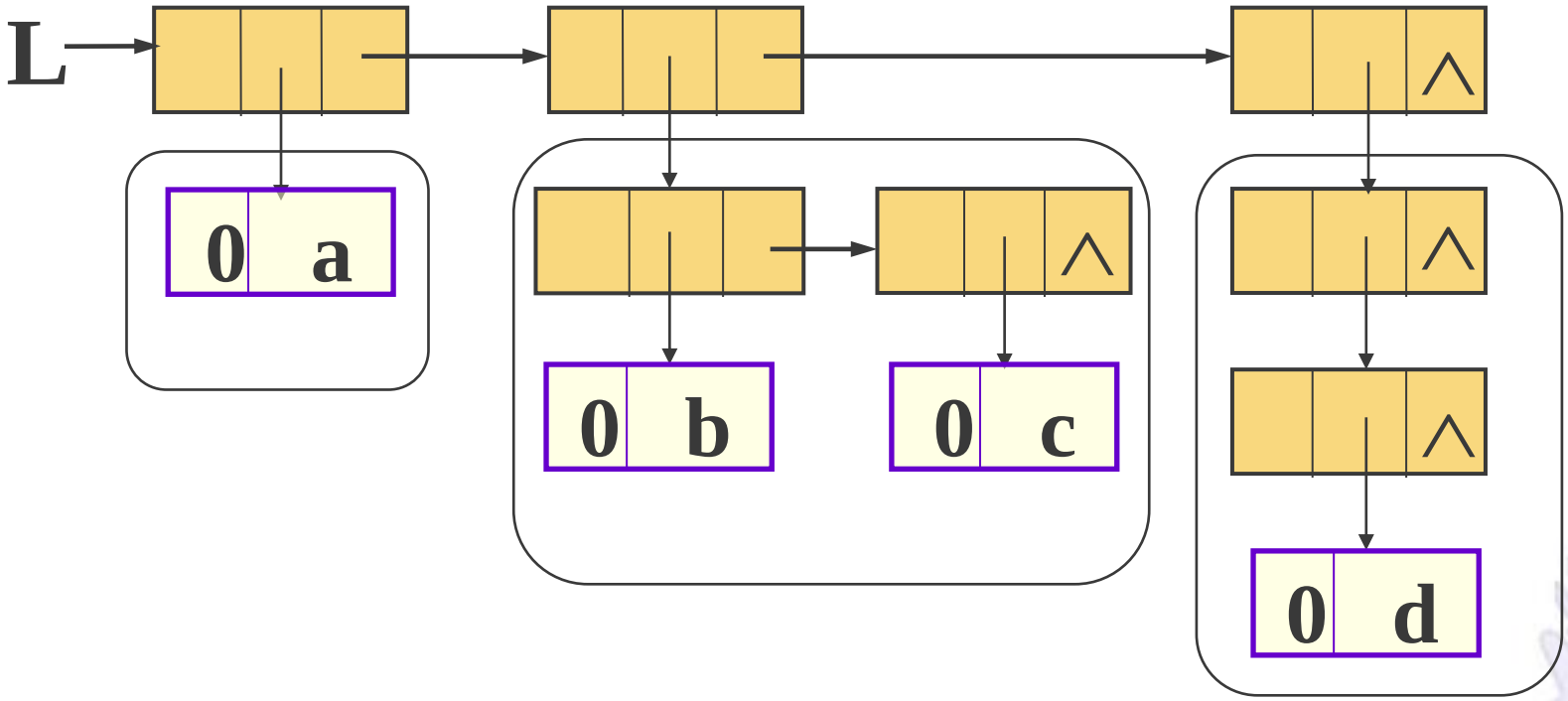
直接求解

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



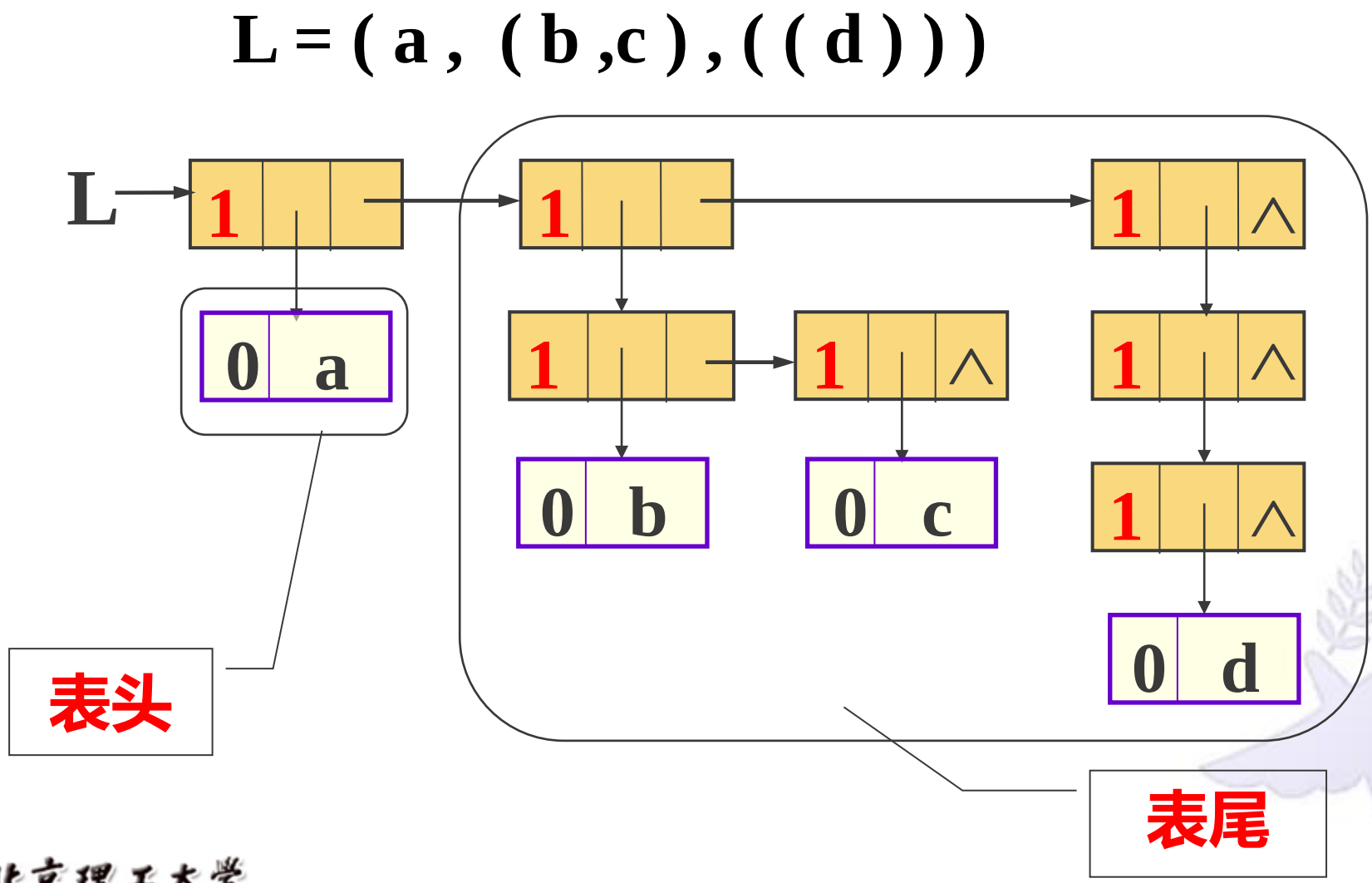
1. 求广义表的深度

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：

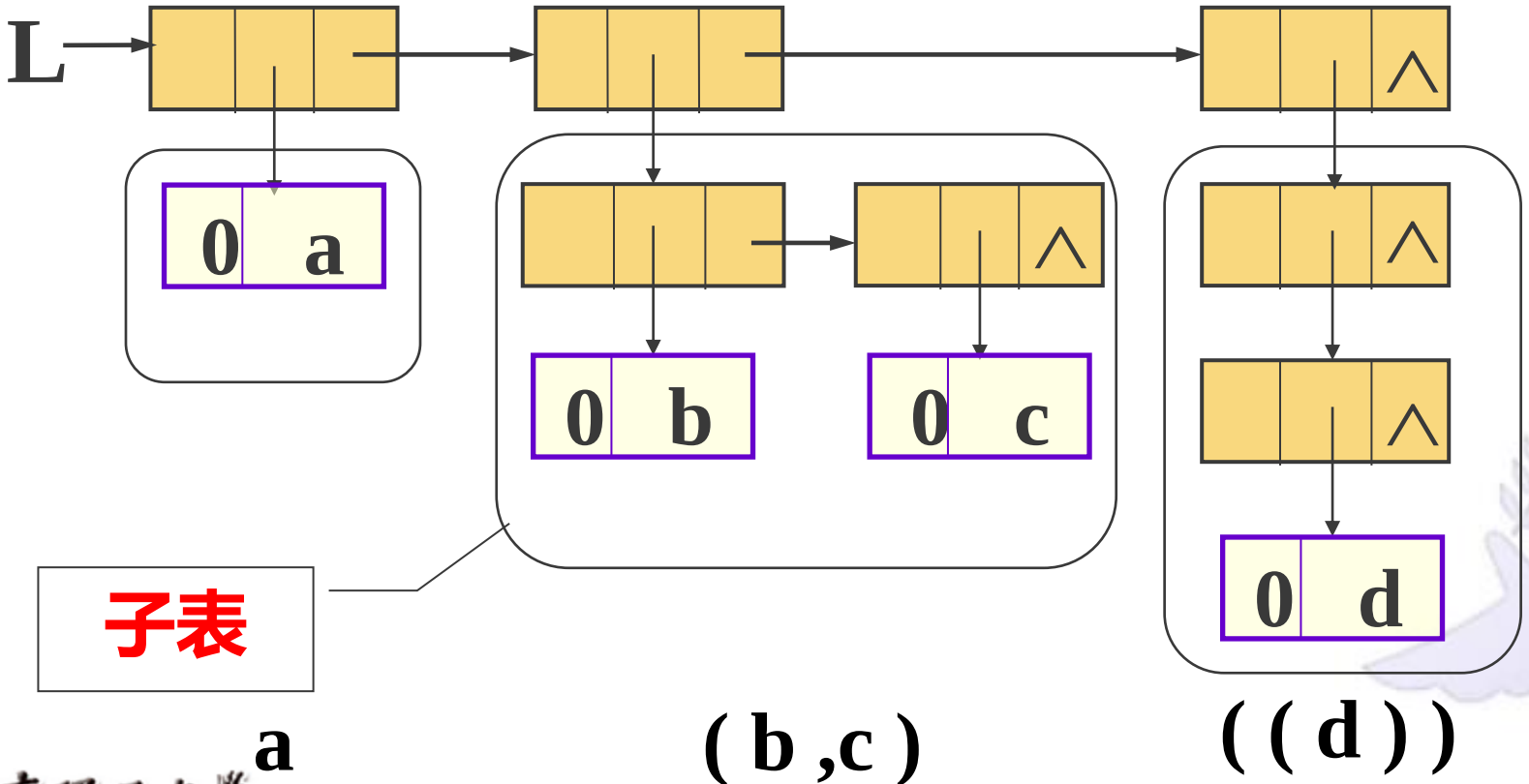
空表: NULL

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
```

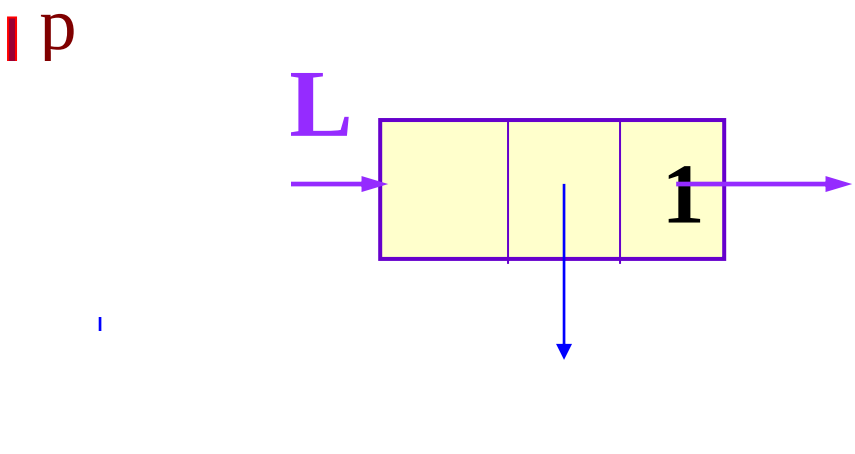
删除单链表中所有值为x 的数据元素

```
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

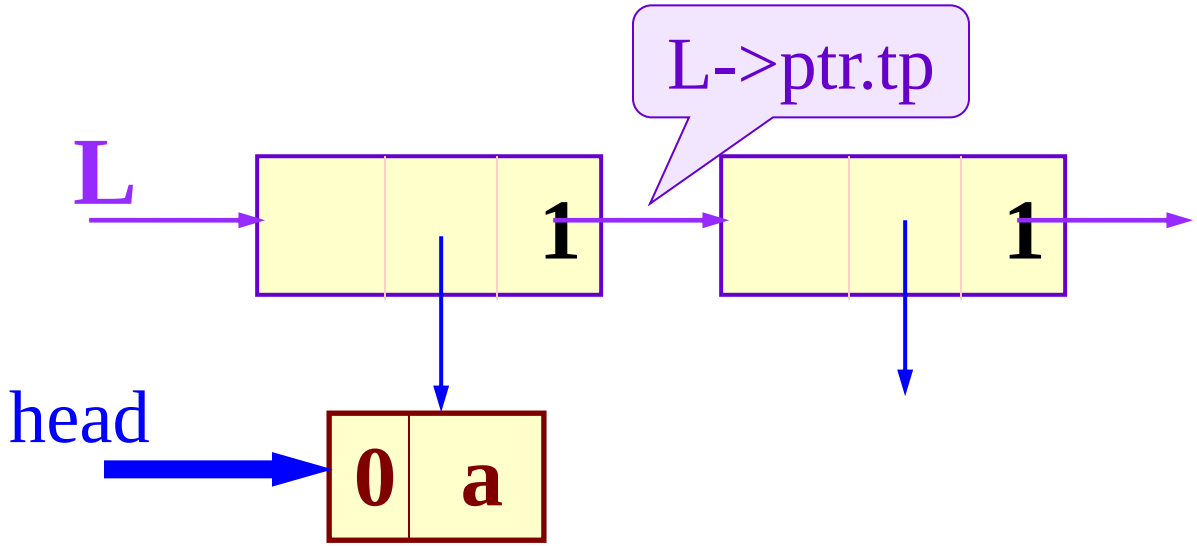
```
void Delete_GL(Glist&L, AtomType x) {
  //删除广义表L中所有值为x的原子结点
  if (L) {
     head = L->ptr.hp;  // 考察第一个子表
     if ((head->tag == Atom) && (head->atom == x))
       {    ……          }    // 删除原子项 x的情况
     elseif((head->tag == Atom) &&  (head->atom != x))
       {  ……          }// 不为x
     elseif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

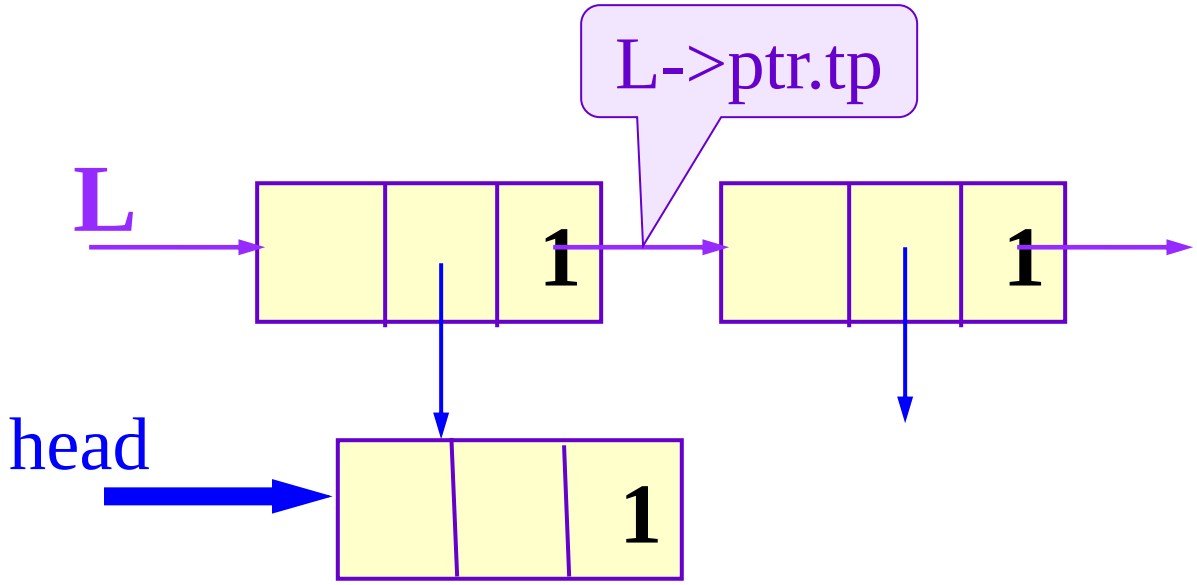
第一项是原子项，但不等于x




```
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)