第十三章 广义表

General List

广义表 (General Lists)

- 广义表的概念 $n\ (\geq 0)$个表元素组成的有限序列，记作

$$LS = (a_0, a_1, a_2, \ldots, a_{n-1})$$

LS是表名，a_i是表元素，它可以是表 (称为子表)，可以是数据元素(称为原子)。

- n为表的长度。$n = 0$ 的广义表为空表。
- $n > 0$时，表的第一个表元素称为广义表的表头(*head*)，除此之外，其它表元素组成的表称为广义表的表尾(*tail*)。

广义表的特性

- 有次序性
- 有深度
- 可递归
- 有长度
- 可共享

A = ( )

B = (6, 2)

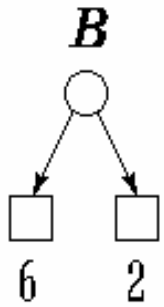
C = (‘*a*’, (5, 3, ‘*x*’))

D = (B, C, A)

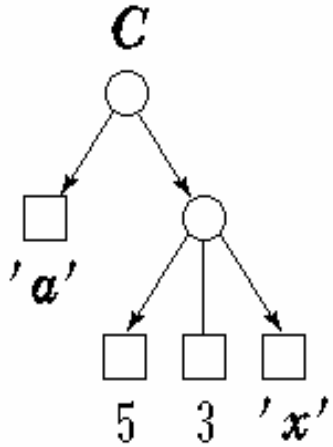
E = (B, D)

F = (4, F)

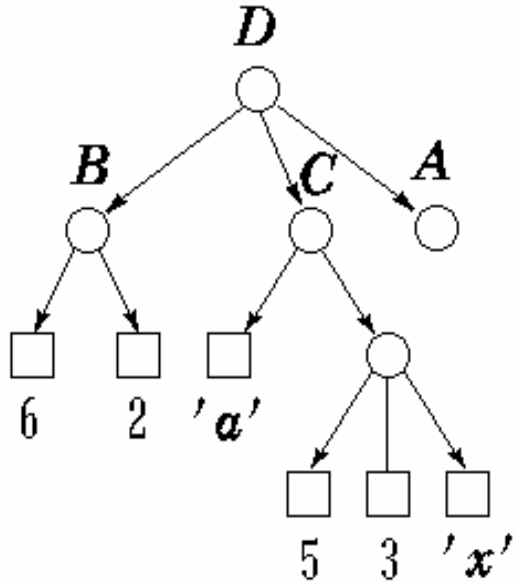
空表

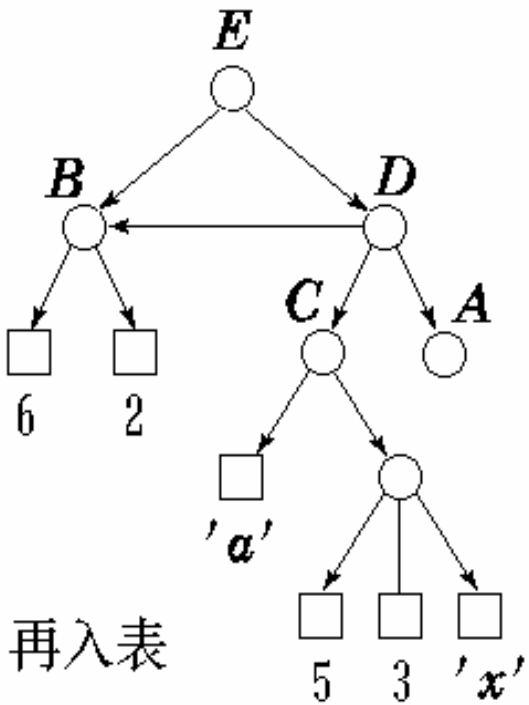


线性表

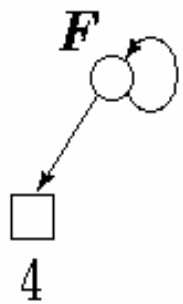


纯表



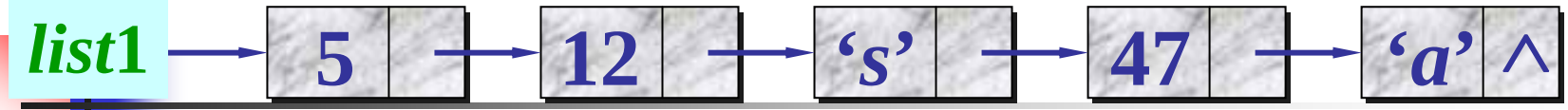


再入表

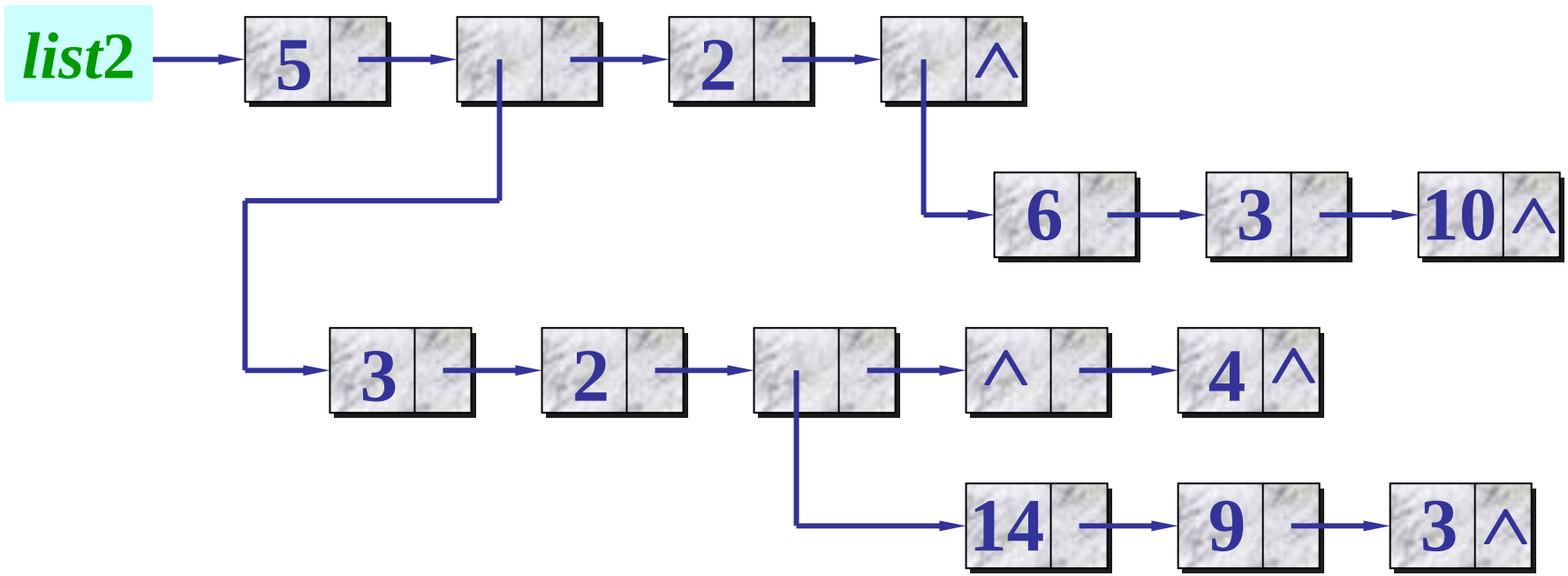


递归表

广义表的表示



只包括整数和字符型数据的广义表链表表示

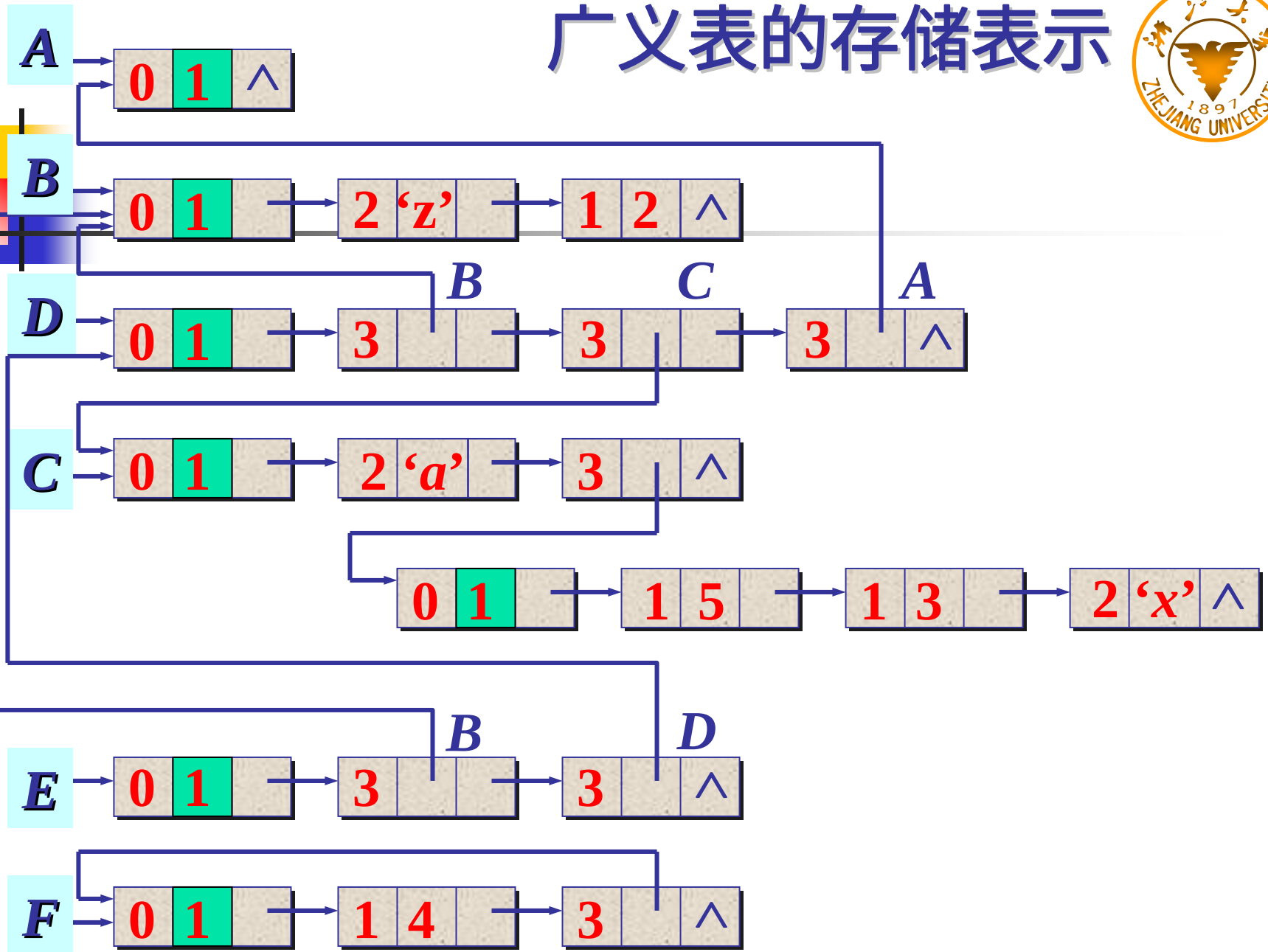


表中套表情形下的广义表链表表示

广义表结点定义

- 结点类型 utype = 0, 表头；= 1, 整型原子；= 2, 字符型原子；= 3, 子表
- 值value utype=0时, 存放引用计数(ref)；utype=1 时，存放整数值(intinfo)；utype=2时，存放字符型数据(charinfo)；utype=3时, 存放指向子表表头的指针(hlink)
- 尾指针tlink utype=0时, 指向该表第一个结点；utype≠0时, 指向同一层下一个结点

广义表的存储表示



节点信息定义

```
struct Items
{    //仅有结点信息的项
   int utype;     //＝0 / 1 / 2 / 3
   union
   {                  //联合
     int ref;                  //utype=0, 存放引用计数
     int intinfo;              //utype=1, 存放整数值
     char charinfo;      //utype =2, 存放字符
     GenListNode *hlink;
                  //utype =3, 存放指向子表的指针
    }value;
}
```

广义表节点类定义

```
class GenListNode
{              //广义表结点类定义
private:
   int utype;                    //=0 / 1 / 2 / 3
   GenListNode * tlink;          //下一结点指针
   union
   {                                        //联合
      int ref;                 //utype=0, 存放引用计数
      int intinfo;             //utype=1, 存放整数值
      char charinfo;           //utype=2, 存放字符
       GenListNode *hlink;     //utype =3, 存放指向子表的指针
   } value;
public:
   GenListNode ( ) : utype (0), tlink (NULL), ref (0) { }   //构造函数
   GenListNode ( int t, GenListNode *next =NULL ) { }   //构造函数：建表结点
   Items& Info ( GenListNode *elem );   //返回表元素elem的值
   int nodetype ( GenListNode *elem )    { return elem->utype; }
                              //返回表元素elem的数据类型
   GenListNode& setInfo ( Items &x );  //将表元素elem中的值修改为x
};
```

广义表类定义(1)

```
class GenList
{                          //广义表类定义
private:
    GenListNode *first;    //广义表头指针

public:
    Genlist ( );                    //构造函数
    ~GenList ( );                   //析构函数
    GenListNode& Head ( );  //返回表头元素
    GenList& Tail ( );              //返回表尾
    GenListNode *First ( );    //返回第一个元素
    GenListNode * Next ( GenListNode *elem );
                                    //返回表元素elem的直接后继元素
    int depth ( );                  //计算一个非递归表的深度
    void setNext ( GenListNode *elem1, GenListNode *elem2 );
                                    //将elem2插到表中元素elem1后
```

广义表类定义(2)

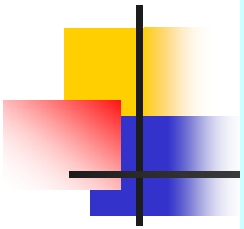
```
public:
    void Copy ( const GenList & l );    //广义表的复制
    int Createlist ( GenListNode *ls, char * s );
            //从广义表的字符串描述 s 出发, 建立一个带表头结点的广义表结构
    GenListNode *Copy ( GenListNode *ls );
            //复制一个 ls 指示的无共享非递归表
    int depth ( GenListNode *ls );       //计算由 ls 指示的非递归表的深度
    int equal (GenListNode *s, GenListNode *t);
            //比较以s和t为表头的两个表是否相等
    void Remove (GenListNode *ls );
            //释放以 ls 为表头结点的广义表
}
```

广义表的访问算法

```
Items& GenListNode ::Info ( GenListNode * elem )
{          //返回表元素elem的值
   Items *pitem = new Items;
   pitem->utype = elem->utype;
   pitem->value = elem->value;
   return * pitem;
}
GenListNode& GenListNode ::setInfo ( Items &x )
{          //修改表元素的值为 x
   utype = x->utype;
   value = x->value;
}
```

广义表类的构造和访问成员函数

```
Genlist :: GenList ( )
{     //构造函数
    GenListNode *first = new GenListNode( );
    first->utype = 0;
    first->value.ref = 1;
    first->tlink = NULL;
}
```



```
Items& GenList :: Head ( )
{   //若广义表非空，则返回其第一个元素的值，否则返回NULL
    if ( first->tlink == NULL ) return NULL;
    else
    {                                   //非空表
       Items * temp = new Items;
       temp->utype = frist->tlink->utype;
       temp->value = frist->tlink->value;
       return * temp;                   //返回类型及值
    }
}
```

```
GenList& GenList :: Tail ( )
{
//若广义表非空，则返回广义表除第一个元
//素外其它元素组成的表, 否则函数没有定义
    if ( frist->tlink == NULL ) return NULL;
    else
   {                                    //非空表
       GenList * temp;
       temp->first = Copy ( first );
       return * temp;
    }
}
```

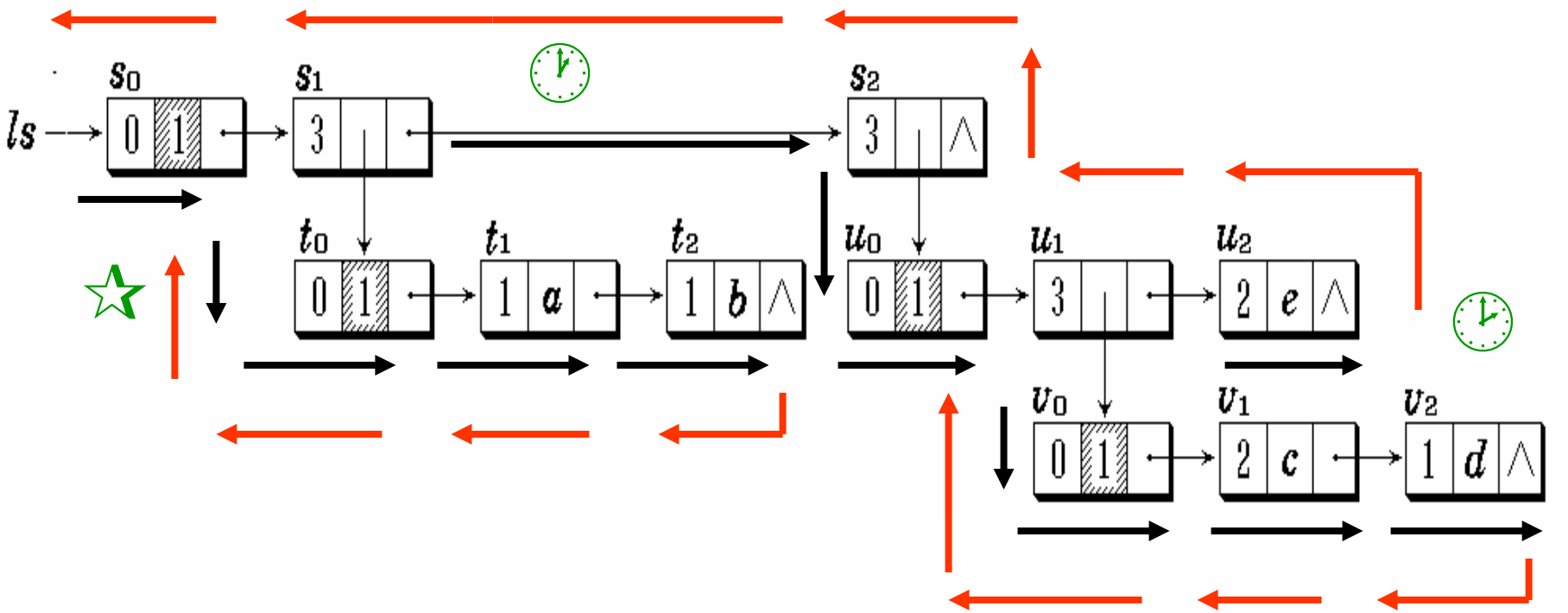
```
GenListNode * GenList :: First ( )
 {
    if ( first->tlink == NULL ) return NULL;
    else return first->tlink;
}

GenListNode * GenList :: Next ( GenListNode *elem )
{
    if ( elem->tlink == NULL ) return NULL;
    else return elem->tlink;
}
```

广义表的递归算法

广义表的复制算法

```
void GenList :: Copy ( const GenList& ls )
{
    first = Copy ( ls.first );
}
```

```
GenListNode* GenList :: Copy ( GenListNode * ls )
{
    GenListNode *q = NULL;
    if ( ls != NULL ) {
    q = new GenListNode ( ls->utype, NULL );
    switch ( ls->utype ) {
        case 0: q->value.ref = ls->value.ref;
            break;
        case 1:
            q->value.intgrinfo = ls->value.intgrinfo;
            break;
        case 2:
            q->value.charinfo = ls->value.charinfo;
            break;
        case 3:
            q->value.hlink = Copy (ls->value.hlink);
      }
      q->tlink = Copy (ls->tlink);
    }
    return q;
}
```


ls
s_0
s_1
s_2
t_0
t_1
t_2
u_0
u_1
u_2
v_0
v_1
v_2
a
b
e
c
d

广义表的深度

$$\mathrm{Depth}(LS) = \begin{cases} 1, & \text{当LS为空表时} \\ 0, & \text{当LS为原子时} \\ 1 + \max\limits_{0 \le i \le n-1} \{\mathrm{Depth}(a_i)\}, & \text{其它，} n \ge 1 \end{cases}$$

例如，对于广义表

E (B (a, b), D (B (a, b), C (u, (x, y, z)), A ( )))

1 1 1 1

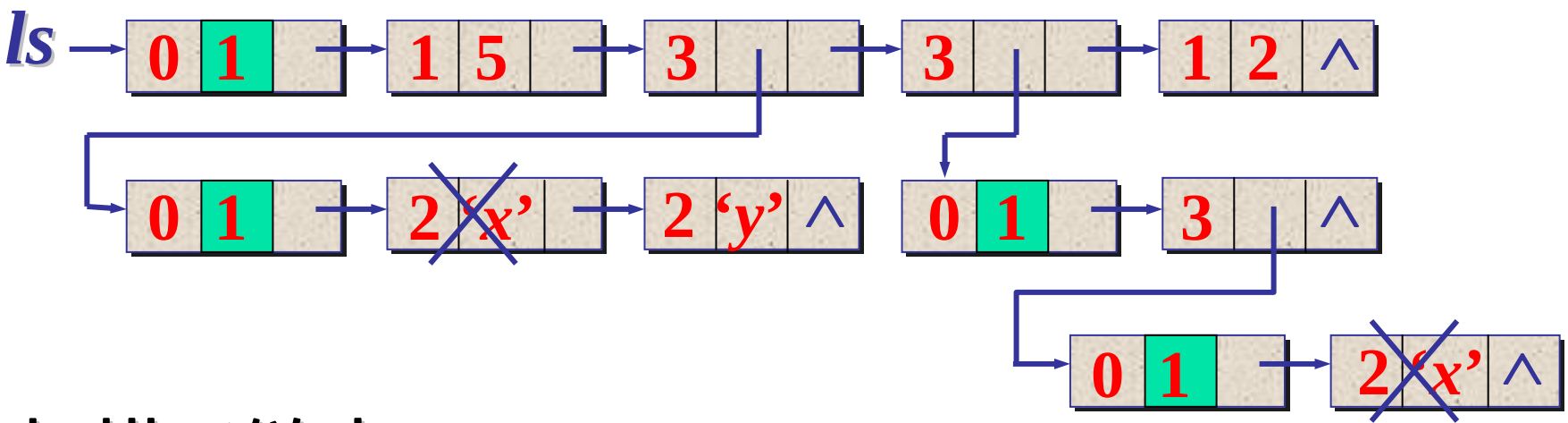
2

3

4

```
int GenList :: depth ( GenListNode *ls )
{
    if ( ls->tlink == NULL ) return 1;   //空表
    GenListNode * temp = ls->tlink;  int m = 0;
    while ( temp != NULL ) {    //在表顶层横扫
        if ( temp->utype == 3 ) {  //结点为表结点
            int n = depth ( temp->value.hlink );
            if ( m < n ) m = n;          //m记最大深度
        }
        temp = temp->tlink;
    }
    return m+1;
}

int GenList :: depth ( )
{
    return depth ( first );
}
```

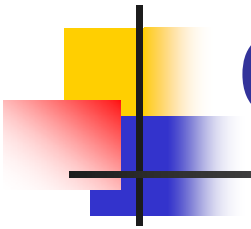
广义表的删除算法



- 扫描子链表
  - 若结点数据为‘*x*’, 删除。 可能做循环连续删。
  - 若结点数据不为‘*x*’， 不执行删除。
  - 若结点为子表，递归在子表执行删除。

```
void delvalue(GenListNode * ls, const value x)
{  //在广义表中删除所有含 x 的结点
   if ( ls->tlink != NULL )
   {                    //非空表
      GenListNode * p = ls->tlink;
      while ( p != NULL &&              //横扫链表
         ( ( p->utype == 1 && p->value.intinfo == x ) ||
           ( p->utype == 2 && p->value.charinfo == x ) )
      {
         ls->tlink = p->tlink;
         delete p;    //删除
         p = ls->tlink;    //指向同一层后继结点
      }
      if ( p != NULL )
      {
         if ( p->utype == 3 )                  //在子表中删除
            delvalue ( p->value.hlink, x );
         delvalue ( p, x );    //在后续链表中删除
      }
   }
}
```

```
GenList :: ~GenList ( )
{              //析构函数
   Remove ( first );
}
void GenList :: Remove ( GenListNode *ls )
{  // 释放以 ls 为表头指针的广义表
   ls->value.ref --;              //引用计数减1
   if ( ls->value.ref == 0 ) {          //如果减到0
      GenListNode *p = ls, *q;   //横扫表顶层
      while ( p->tlink != NULL ) {
         q = p->tlink;                    //到第一个结点
         if ( q->utype == 3 )       //递归删除子表
            Remove ( q->value.hlink );
         p->link = q->link;
         delete q;
      }
   }
}
```



Q&A