5-1 已知 $A[n]$为整数数组，试写出实现下列运算的递归算法：

(1) 求数组 A 中的最大整数。

(2) 求 n 个整数的和。

(3) 求 n 个整数的平均值。

【解答】

```
#include <iostream.h>
class RecurveArray {                          //数组类声明
private:
    int *Elements;                            //数组指针
    int ArraySize;                            //数组尺寸
    int CurrentSize;                          //当前已有数组元素个数
public :
    RecurveArray ( int MaxSize =10 ) :
    ArraySize ( MaxSize ), Elements ( new int[MaxSize] ){ }
    ~RecurveArray ( ) { delete [ ] Elements; }
    void InputArray();                        //输入数组的内容
    int MaxKey ( int n );                     //求最大值
    int Sum ( int n );                        //求数组元素之和
    float Average ( int n );                  //求数组元素的平均值
};

void RecurveArray :: InputArray ( ){     //输入数组的内容
    cout << "Input the number of Array: \n";
    for ( int i = 0; i < ArraySize; i++ ) cin >> Elements[i];
}

int RecurveArray :: MaxKey ( int n ) {          //递归求最大值
    if ( n == 1 ) return Elements[0];

    int temp = MaxKey ( n - 1 );


    if ( Elements[n-1] > temp ) return Elements[n-1];

    else return temp;
}

int RecurveArray :: Sum ( int n ) {             //递归求数组之和
    if ( n == 1) return Elements[0];

    else return Elements[n-1] + Sum (n−1);

}

float RecurveArray :: Average ( int n ) {       //递归求数组的平均值
    if ( n == 1) return (float) Elements[0];

    else return ( (float) Elements[n-1] + ( n - 1) * Average ( n - 1 ) ) / n;
```

```
}

int main ( int argc,   char* argv [ ] ) {

int size = -1;

    cout << "No. of the Elements : ";
    while ( size < 1 ) cin >> size;
    RecurveArray ra ( size );
    ra.InputArray();
    cout<< "\nThe max is:    " << ra.MaxKey ( ra.MaxSize ) << endl;
    cout<< "\nThe sum is:    " << ra.Sum ( ra.MaxSize ) << endl;
    cout<< "\nthe avr is:    " << ra.Average ( ra.MaxSize ) << endl;
    return 0;
}
```

5-2 已知 *Ackerman* 函数定义如下：

$$akm(m,n)=\begin{cases} n+1 & \text{当}m=0\text{时} \\ akm(m-1,\ 1) & \text{当}m\neq 0, n=0\text{时} \\ akm(m-1,\ akm(m,n-1)) & \text{当}m\neq 0, n\neq 0\text{时} \end{cases}$$

(1) 根据定义，写出它的递归求解算法；

(2) 利用栈，写出它的非递归求解算法。

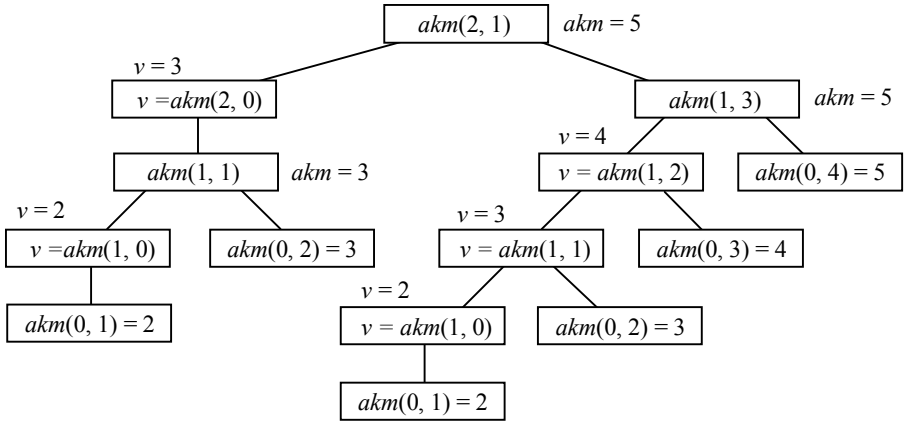
【解答】(1) 已知函数本身是递归定义的，所以可以用递归算法来解决：

```
unsigned akm ( unsigned m, unsigned n ) {
    if ( m == 0 ) return n+1;                          // m == 0
        else if ( n == 0 ) return akm ( m-1, 1 );          // m > 0, n == 0
            else return akm ( m-1, akm ( m, n-1 ) );           // m > 0, n > 0
}
```

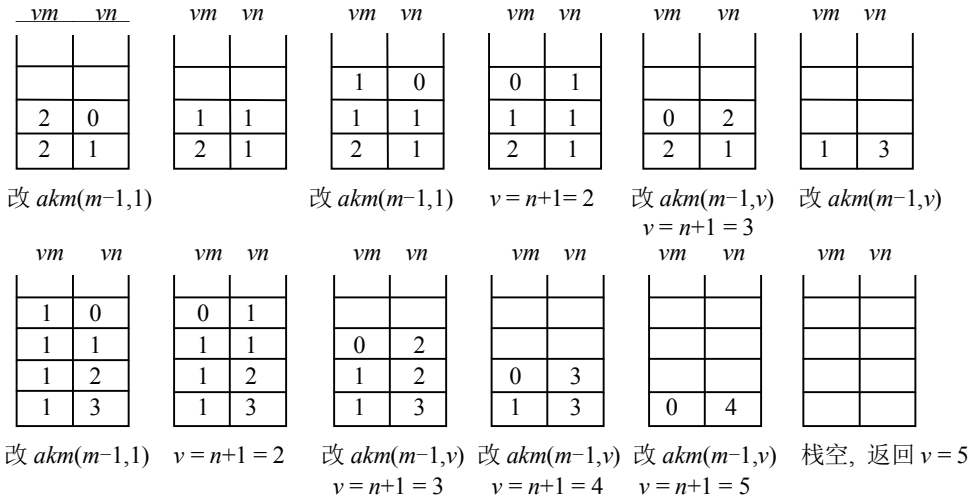
(2) 为了将递归算法改成非递归算法，首先改写原来的递归算法，将递归语句从结构中独立出来：

```
unsigned akm ( unsigned m, unsigned n ) {
    unsigned v;
    if ( m == 0 ) return n+1;                          // m == 0
        if ( n == 0 ) return akm ( m-1, 1 );               // m > 0, n ==0
    v = akm ( m, n-1 ) );                              // m > 0, n > 0
        return akm ( m-1, v );
}
```

计算 *akm*(2, 1)的递归调用树如图所示：



用到一个栈记忆每次递归调用时的实参值，每个结点两个域{*vm*, *vn*}。对以上实例，栈的变化如下：



相应算法如下

```
#include <iostream.h>
#include "stack.h"
#define maxSize 3500;
unsigned akm ( unsigned m, unsigned n ) {
    struct node { unsigned vm, vn; }
    stack<node> st ( maxSize );   node w;   unsigned v;
    w.vm = m;   w.vn = n;   st.Push (w);
    do {
        while ( st.GetTop( ).vm > 0 ) {                  //计算 akm(m−1, akm(m, n−1) )
            while ( st.GetTop( ).vn > 0 )                    //计算 akm(m, n−1), 直到 akm(m, 0)
                { w.vn−−;   st.Push( w ); }
            w = st.GetTop( );   st.Pop( );               //计算 akm(m−1, 1)
            w.vm−−;   w.vn = 1;   st.Push( w );
        }                                                //直到 akm( 0, akm( 1, * ) )
        w = st.GetTop();   st.Pop( );   v = w.vn++;      //计算 v = akm( 1, * )+1
        if ( st.IsEmpty( ) == 0 )                            //如果栈不空, 改栈顶为( m−1, v )
            { w = st.GetTop();   st.Pop( );   w.vm−−;   w.vn = v;   st.Push( w ); }
```

```
    } while ( st.IsEmpty( ) == 0 );
    return v;
}
```

5-3 【背包问题】设有一个背包可以放入的物品的重量为 s，现有 n 件物品，重量分别为 $w[1], w[2], \cdots, w[n]$。问能否从这 n 件物品中选择若干件放入此背包中，使得放入的重量之和正好为 s。如果存在一种符合上述要求的选择，则称此背包问题有解(或称其解为真)；否则称此背包问题无解(或称其解为假)。试用递归方法设计求解背包问题的算法。（提示：此背包问题的递归定义如下：）

$$KNAP(s,n)=\begin{cases} \text{True} & s=0 & \text{此时背包问题一定有解} \\ \text{False} & s<0 & \text{总重量不能为负数} \\ \text{False} & s>0 \text{且} n<1 & \text{物品件数不能为负数} \\ KNAP(s,n-1)\text{或} & s>0 \text{且} n\geq 1 & \text{所选物品中不包括} w[n] \text{时} \\ KNAP(s-w[n],n-1) & & \text{所选物品中包括} w[n] \text{时} \end{cases}$$

【解答】　根据递归定义，可以写出递归的算法。

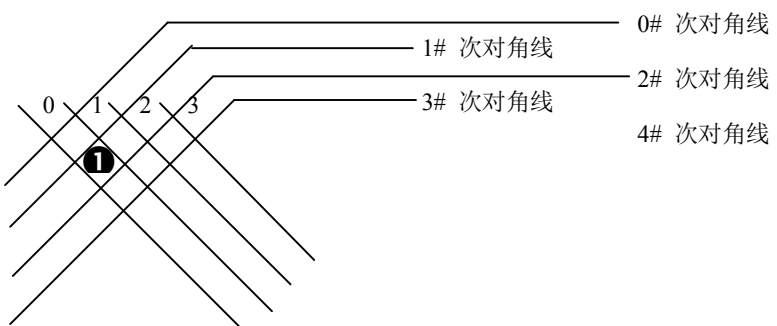
```
enum boolean { False, True }
boolean Knap( int s, int n ) {
    if ( s == 0 ) return True;
    if ( s < 0 || s > 0 && n < 1 ) return False;
    if ( Knap ( s – W[n], n–1 ) == True )
        { cout << W[n] << ' , '; return True; }
    return Knap( s, n–1 );
}
```

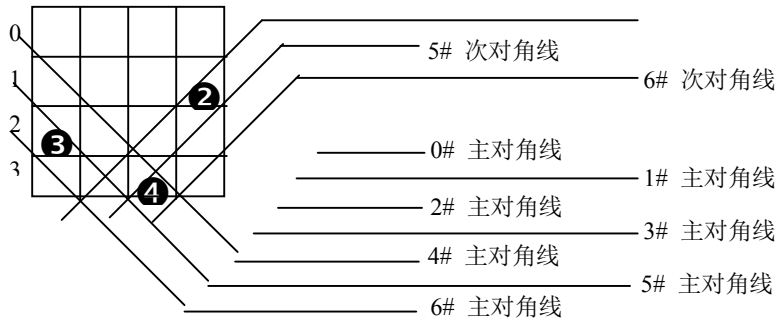
若设 $w = \{ 0, 1, 2, 4, 8, 16, 32 \}$，$s = 51, n = 6$。则递归执行过程如下

递归 ↓	*Knap*(51, 6)					**return** True, 完成
	Knap(51−32, 5)					**return** True, 打印 32
	Knap(19−16, 4)					**return** True, 打印 16
	Knap(3−8, 3)	**return** False	*Knap*(3, 3)			**return** True, 无动作
	s = −5 < 0	**return** False	*Knap*(3−4, 4)	**return** False	*Knap*(3, 2)	**return** True, 无动作
			s = −1 < 0	**return** False	*Knap*(3−2, 1)	**return** True, 打印 2
					Knap(1−1, 0)	**return** True, 打印 1
					s = 0	**return** True

5-4 【八皇后问题】设在初始状态下在国际象棋棋盘上没有任何棋子(皇后)。然后顺序在第 1 行，第 2 行，…。第 8 行上布放棋子。在每一行中有 8 个可选择位置，但在任一时刻，棋盘的合法布局都必须满足 3 个限制条件，即任何两个棋子不得放在棋盘上的同一行、或者同一列、或者同一斜线上。试编写一个递归算法，求解并输出此问题的所有合法布局。(提示：用回溯法。在第 n 行第 j 列安放一个棋子时，需要记录在行方向、列方向、正斜线方向、反斜线方向的安放状态，若当前布局合法，可向下一行递归求解，否则可移走这个棋子，恢复安放该棋子前的状态，试探本行的第 j+1 列。)

【解答】此为典型的回溯法问题。





在解决 8 皇后时，采用回溯法。在安放第 *i* 行皇后时，需要在列的方向从 1 到 *n* 试探(*j* = 1, …, *n*)：首先在第 *j* 列安放一个皇后，如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 *j* 列安放的皇后。如果没有出现攻击，在第 *j* 列安放的皇后不动，递归安放第 *i*+1 行皇后。

解题时设置 4 个数组：

col [*n*] ： *col*[*i*] 标识第 *i* 列是否安放了皇后

md[2*n*-1] ： *md*[*k*] 标识第 *k* 条主对角线是否安放了皇后

sd[2*n*-1] ： *sd*[*k*] 标识第 *k* 条次对角线是否安放了皇后

q[*n*] ： *q*[*i*] 记录第 *i* 行皇后在第几列

利用行号 *i* 和列号 *j* 计算主对角线编号 *k* 的方法是 *k* = *n*+*i*−*j*−1；计算次对角线编号 *k* 的方法是 *k* = *i*+*j*。*n* 皇后问题解法如下：

```
void Queen( int i ) {
    for ( int j = 1;  j <= n;  j++ ) {
        if ( col[j] == 0 && md[n+i−j−1] == 0 && sd[i+j] == 0 ) {      //第 i 行第 j 列没有攻击
            col[j] = md[n+i−j−1] = sd[i+j] = 1;   q[i] = j;           //在第 i 行第 j 列安放皇后
            if ( i == n ) {                                           //输出一个布局
                for ( j = 0; j <= n; j++ ) cout << q[j] << ',';
                cout << endl;
            }
            else   Queen ( i+1 );                                     //在第 i+1 行安放皇后
        }
        col[j] = md[n+i−j−1] = sd[i+j] = 0;   q[i] = 0;               //撤消第 i 行第 j 列的皇后
    }
}
```

5−5 已知 *f* 为单链表的表头指针，链表中存储的都是整型数据，试写出实现下列运算的递归算法：

(1) 求链表中的最大整数。

(2) 求链表的结点个数。

(3) 求所有整数的平均值。

【解答】

```
#include <iostream.h>                          //定义在头文件"RecurveList.h"中
class List;
class ListNode {                               //链表结点类
friend class List;
private:
```

```
    int data;                                   //结点数据
    ListNode *link;                             //结点指针
    ListNode ( const int item ) : data(item), link(NULL) { }    //构造函数
};
class List {                                    //链表类
private:
    ListNode *first, current;
    int Max ( ListNode *f );
    int Num ( ListNode *f );
    float Avg ( ListNode *f,    int& n );
public:
    List ( ) : first(NULL), current (NULL) { }          //构造函数
    ~List ( ){ }                                //析构函数
    ListNode* NewNode ( const int item );       //创建链表结点, 其值为 item
    void NewList ( const int retvalue );        //建立链表, 以输入 retvalue 结束
    void PrintList ( );                         //输出链表所有结点数据
    int GetMax ( ) { return Max ( first ); }    //求链表所有数据的最大值
    int GetNum ( ) { return Num ( first ); }    //求链表中数据个数
    float GetAvg ( ) { return Avg ( first ); }  //求链表所有数据的平均值
};

ListNode* List :: NewNode ( const int item ) {          //创建新链表结点
    ListNode *newnode = new ListNode (item);
    return newnode;
}

void List :: NewList ( const int retvalue ) {           //建立链表, 以输入 retvalue 结束
    first = NULL;    int value;    ListNode *q;
    cout << "Input your data:\n";                       //提示
    cin >> value;                                       //输入
    while ( value != retvalue ) {                       //输入有效
        q = NewNode ( value );                          //建立包含 value 的新结点
        if ( first == NULL ) first = current = q;       //空表时, 新结点成为链表第一个结点
        else { current->link = q;    current = q; }     //非空表时, 新结点链入链尾
        cin >> value;                                   //再输入
    }
    current->link = NULL;                               //链尾封闭
}

void List :: PrintList ( ) {                            //输出链表
    cout << "\nThe List is : \n";
    ListNode *p = first;
```

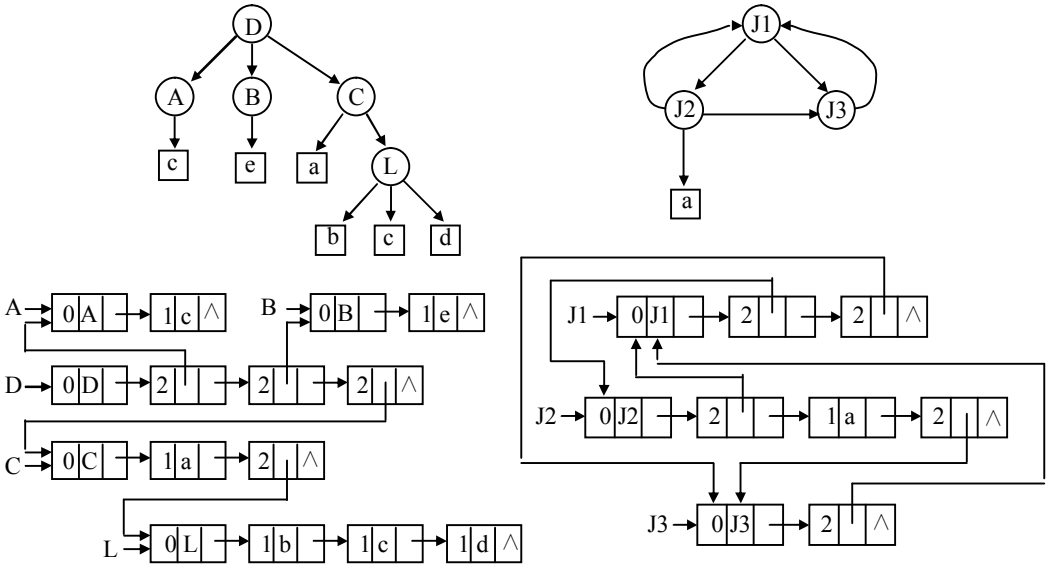
```
    while ( p != NULL ) {  cout << p->data << '    ';    p = p->link;  }
    cout << '\n';
}

int List :: Max ( ListNode *f ) {                    //递归算法 ： 求链表中的最大值
    if ( f->link == NULL ) return f->data;           //递归结束条件
    int temp = Max ( f->link );                      //在当前结点的后继链表中求最大值
    if ( f->data > temp ) return f->data;            //如果当前结点的值还要大, 返回当前检点值
    else return temp;                                //否则返回后继链表中的最大值
}

int List :: Num ( ListNode *f ) {                    //递归算法 ： 求链表中结点个数
    if ( f == NULL ) return 0;                       //空表, 返回 0
    return 1+ Num ( f->link );                       //否则, 返回后继链表结点个数加 1
}

float List :: Avg ( ListNode *f , int& n ) {         //递归算法 ： 求链表中所有元素的平均值
    if ( f->link == NULL )                           //链表中只有一个结点, 递归结束条件
        { n = 1;    return ( float ) (f->data ); }
    else { float Sum = Avg ( f->link, n ) * n;    n++;    return ( f->data + Sum ) / n; }
}

#include "RecurveList.h"                             //定义在主文件中
int main ( int argc, char* argv[ ] ) {
    List test;      int finished;
    cout << "输入建表结束标志数据 ： ";
    cin >> finished;                                 //输入建表结束标志数据
    test.NewList ( finished );                       //建立链表
    test.PrintList ( );                              //打印链表
    cout << "\nThe Max is : " << test.GetMax ( );
    cout << "\nThe Num is : " << test.GetNum ( );
    cout << "\nThe Ave is : " << test.GetAve () << '\n';
    printf ( "Hello World!\n" );
    return 0;
}
```

5-6 画出下列广义表的图形表示和它们的存储表示：

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

(2) J1(J2(J1, a, J3(J1)), J3(J1))

【解答】(1) D(A(c), B(e), C(a, L(b, c, d)))　　(2) J1(J2(J1, a, J3(J1)), J3(J1))



5-7 利用广义表的 *head* 和 *tail* 操作写出函数表达式，把以下各题中的单元素 banana 从广义表中分离出来：

(1) L1(apple, pear, banana, orange)

(2) L2((apple, pear), (banana, orange))

(3) L3(((apple), (pear), (banana), (orange)))

(4) L4((((apple))), ((pear)), (banana), orange)

(5) L5((((apple), pear), banana), orange)

(6) L6(apple, (pear, (banana), orange))

【解答】

(1) Head (Tail (Tail (L1)))

(2) Head (Head (Tail (L2)))

(3) Head (Head (Tail (Tail (Head (L3))))))

(4) Head (Head (Tail (Tail (L4))))

(5) Head (Tail (Head(L5)))

(6) Head (Head (Tail (Head (Tail (L6)))))

5-8 广义表具有可共享性，因此在遍历一个广义表时必需为每一个结点增加一个标志域 *mark*，以记录该结点是否访问过。一旦某一个共享的子表结点被作了访问标志，以后就不再访问它。

(1) 试定义该广义表的类结构；

(2) 采用递归的算法对一个非递归的广义表进行遍历。

(3) 试使用一个栈，实现一个非递归算法，对一个非递归广义表进行遍历。

【解答】(1) 定义广义表的类结构

为了简化广义表的操作，在广义表中只包含字符型原子结点，并用除大写字母外的字符表示数据，表头结点中存放用大写字母表示的表名。这样，广义表中结点类型三种：表头结

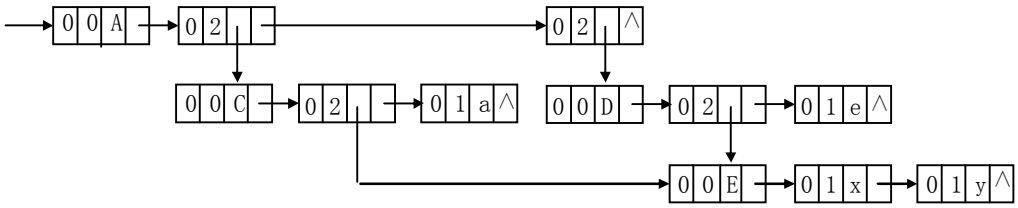
点、原子结点和子表结点。

```
class GenList;                                  //GenList 类的前视声明
class GenListNode {                             //广义表结点类定义
friend class Genlist;
private:
    int mark, utype;                            // utype  =0 / 1 / 2, mark 是访问标记, 未访问为 0
    GenListNode* tlink;                         //指向同一层下一结点的指针
    union {                                     //联合
        char listname;                          // utype = 0, 表头结点情形：存放表名
        char atom;                              // utype = 1, 存放原子结点的数据
        GenListNode* hlink;                     // utype = 2, 存放指向子表的指针
    } value;
public:
    GenListNode ( int tp, char info ) : mark (0), utype (tp), tlink (NULL)          //表头或原子结点构造函
数
        { if ( utype == 0 ) value.listname = info; else value.atom = info; }
    GenListNode (GenListNode* hp )                                  //子表构造函数
        : mark (0), utype (2), value.hlink (hp) { }
    char Info ( GenListNode* elem )                                 //返回表元素 elem 的值

        { return ( utype == 0 ) ? elem->value.listname : elem->value.atom; }

};
class GenList {                                 //广义表类定义
private:
    GenListNode *first;                         //广义表头指针
    void traverse ( GenListNode * ls );             //广义表遍历
    void Remove ( GenListNode* ls );            //将以 ls 为表头结点的广义表结构释放
public:
    Genlist ( char& value );                        //构造函数, value 是指定的停止建表标志数据
    ~GenList ( );                               //析构函数
    void traverse ( );                          //遍历广义表
}
```

(2) 广义表遍历的递归算法

```
void GenList :: traverse ( ) {                  //共有函数
    traverse (first );
}

#include <iostream.h>
void GenList :: traverse ( GenListNode * ls ) {             //私有函数, 广义表的遍历算法
    if ( ls != NULL ) {

        ls->mark = 1;

        if ( ls->utype == 0 ) cout << ls->value.listname << '(';                //表头结点
```

```
        else if ( ls->utype == 1 ) {                                    //原子结点
            cout << ls->value.atom;
            if ( ls->tlink != NULL ) cout << ',';
        }
        else if ( ls->utype == 2 ) {                                    //子表结点
            if ( ls->value.hlink->mark == 0 ) traverse( ls->value.hlink );   //向表头搜索
            else cout << ls->value.hlink->value.listname;
            if ( ls->tlink != NULL ) cout << ',';
        }
        traverse ( ls->tlink );                                         //向表尾搜索
    }
    else cout << ')';
}
```



对上图所示的广义表进行遍历，得到的遍历结果为 A(C(E(x, y), a), D(E, e))。

(3) 利用栈可实现上述算法的非递归解法。栈中存放回退时下一将访问的结点地址。

```
#include <iostream.h>
#include "stack.h"
void GenList :: traverse ( GenListNode *ls ) {
    Stack <GenListNode<Type> *> st;
    while ( ls != NULL ) {
        ls->mark = 1;
        if ( ls->utype == 2 ) {                                 //子表结点
            if ( ls->value.hlink->mark == 0 )                   //该子表未访问过
              { st.Push ( ls->tlink );  ls = ls->value.hlink; }   //暂存下一结点地址, 访问子表
            else {
```

```
            cout << ls->value.hlink->value.listname;                    //该子表已访问过, 仅输出表名

            if ( ls->tlink != NULL ) { cout << ',' ; ls = ls->tlink; }

        }
    }
    else {

        if ( ls->utype == 0 ) cout << ls->value.listname << '(';        //表头结点

        else if ( ls->utype == 1 ) {                                    //原子结点

            cout << ls->value.atom;

            if ( ls->tlink != NULL ) cout << ',';

        }

        if ( ls->tlink == NULL ) {                                      //子表访问完, 子表结束处理

            cout >> ')';
            if ( st.IsEmpty( ) == 0 ) {                                 //栈不空
                ls = st.GetTop ( );   st.Pop ( );                       //退栈
                if ( ls != NULL ) cout << ',';
                else cout << ')';
            }
        }

        else ls = ls->tlink;                                            //向表尾搜索

    }
  }
}
```

(4) 广义表建立操作的实现

```
#include <iostream.h>
#include <ctype.h>
#include "stack.h"
const int maxSubListNum = 20;                                           //最大子表个数
GenList :: GenList ( char& value ) {
    Stack <GenListNode* > st (maxSubListNum);                           //用于建表时记忆回退地址
    SeqList <char> Name (maxSubListNum);                                //记忆建立过的表名
    SeqList <GenListNode * > Pointr (maxSubListNum);                    //记忆对应表头指针
    GenListNode * p, q, r;    Type ch;     int m = 0, ad, br;           //m 为已建表计数, br 用于对消括号
    cout << "广义表停止输入标志数据 value : ";     cin >> value;
    cout << "开始输入广义表数据, 如 A(C(E( x, y ), a ), D(E(x, y), e) )"
    cin >> ch;   first = q = new GenListNode ( 0, ch );                 //建立整个表的表头结点
    if ( ch != value ) { Name.Insert (ch, m);   Pointr.Insert (q, m);   m++; }  //记录刚建立的表头结点
```

```
    else return 1;                                          //否则建立空表, 返回 1
    cin >> ch;    if ( ch == '(' ) st.Push ( q );           //接着应是'(', 进栈
    cin >> ch;
    while ( ch != value ) {                                 //逐个结点加入
        switch ( ch ) {
          case '(' :   p = new GenListNode <Type> ( q );    //建立子表结点, p->hlink = q
                       r = st.GetTop( );   st.Pop( );   r->tlink = p;   //子表结点插在前一结点 r 之后
                       st.Push( p );   st.Push( q );        //子表结点及下一表头结点进栈
                       break;
          case ')' :   q->tlink = NULL;    st.pop( );       //子表建成, 封闭链, 退到上层
                       if ( st.IsEmpty ( ) == 0 ) q = st.GetTop( );   //栈不空, 取上层链子表结点
                       else return 0;                       //栈空, 无上层链, 算法结束
                       break;
          case ',' :   break;
          default :    ad = Name.Find (ch);                 //查找是否已建立, 返回找到位置
                       if ( ad == -1 ) {                    //查不到, 建新结点
                          p = q;
                          if ( isupper (ch) ) {             //大写字母, 建表头结点
                             q = new GenListNode ( 0, ch );
                             Name.Insert (ch, m);   Pointr.Insert (q, m);   m++;
                          }
                          else q = new GenListNode ( 1, ch );   //非大写字母, 建原子结点
                          p->tlink = q;                     //链接
                       }
                       else {                               //查到, 已加入此表
                          q = Pointr.Get (ad);   p = new GenListNode (q);   //建立子表结点, p->hlink = q
                          r = st.GetTop ( );     st.Pop ( );   r->tlink = p;   st.Push (p);   q = p;
                          br = 0;                           //准备对消括号
                          cin >> ch;   if ( ch == '(' ) br++;   //若有左括号, br 加 1
                          while ( br == 0 ) {               //br 为 0 表示括号对消完, 出循环
                             cin >> ch;
                             if ( ch == '(' ) br++;   else if ( ch == ')' ) br--;
                          }
                       }
        }
        cin >> ch;
    }
}
```