第五章 递归与广义表

- 递归(ss)的概念
- 迷宫(Maze)问题
- 递归过程与递归工作栈
- 广义表 (General Lists)
- 小结

递归的概念

- 递归的定义 若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；若一个过程直接地或间接地调用自己，则称这个过程是递归的过程。
- 以下三种情况常常用到递归方法。
  - 定义是递归的
  - 数据结构是递归的
  - 问题的解法是递归的

定义是递归的

例如，阶乘函数

$$n!=\begin{cases}1, & \text{当 } n=0 \text{ 时}\\ n*(n-1)!, & \text{当 } n\geq 1 \text{ 时}\end{cases}$$

求解阶乘函数的递归算法

```
long Factorial ( long n ) {
    if ( n == 0 ) return 1;
    else return n*Factorial (n−1);
}
```

求解阶乘 n! 的过程


参数
计算
返回
0
0! = 1
1
参数
计算
返回
1
1*Factorial(0)
1
参数
计算
返回
2
2*Factorial(1)
2
参数
计算
返回
3
3*Factorial(2)
6
参数
计算
返回
4
4*Factorial(3)
24
主程序 main
0
1
2
3
4
参数传递
1
1
2
6
24
返回值

计算斐波那契数列的函数*Fib(n)*的定义

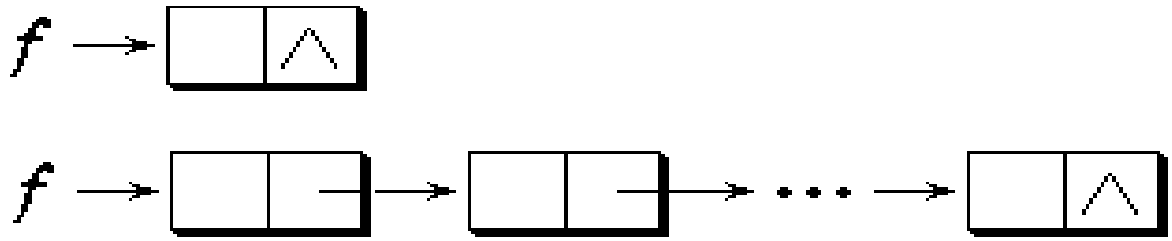
$$Fib(n)=\begin{cases} n, & n=0,1 \\ Fib(n-1)+Fib(n-2), & n>1 \end{cases}$$

求解斐波那契数列的递归算法

```
long Fib ( long n ) {
    if ( n <= 1 ) return n;
    else return Fib (n−1) + Fib (n−2);
}
```

数据结构是递归的

例如，单链表结构



搜索链表最后一个结点并打印其数值

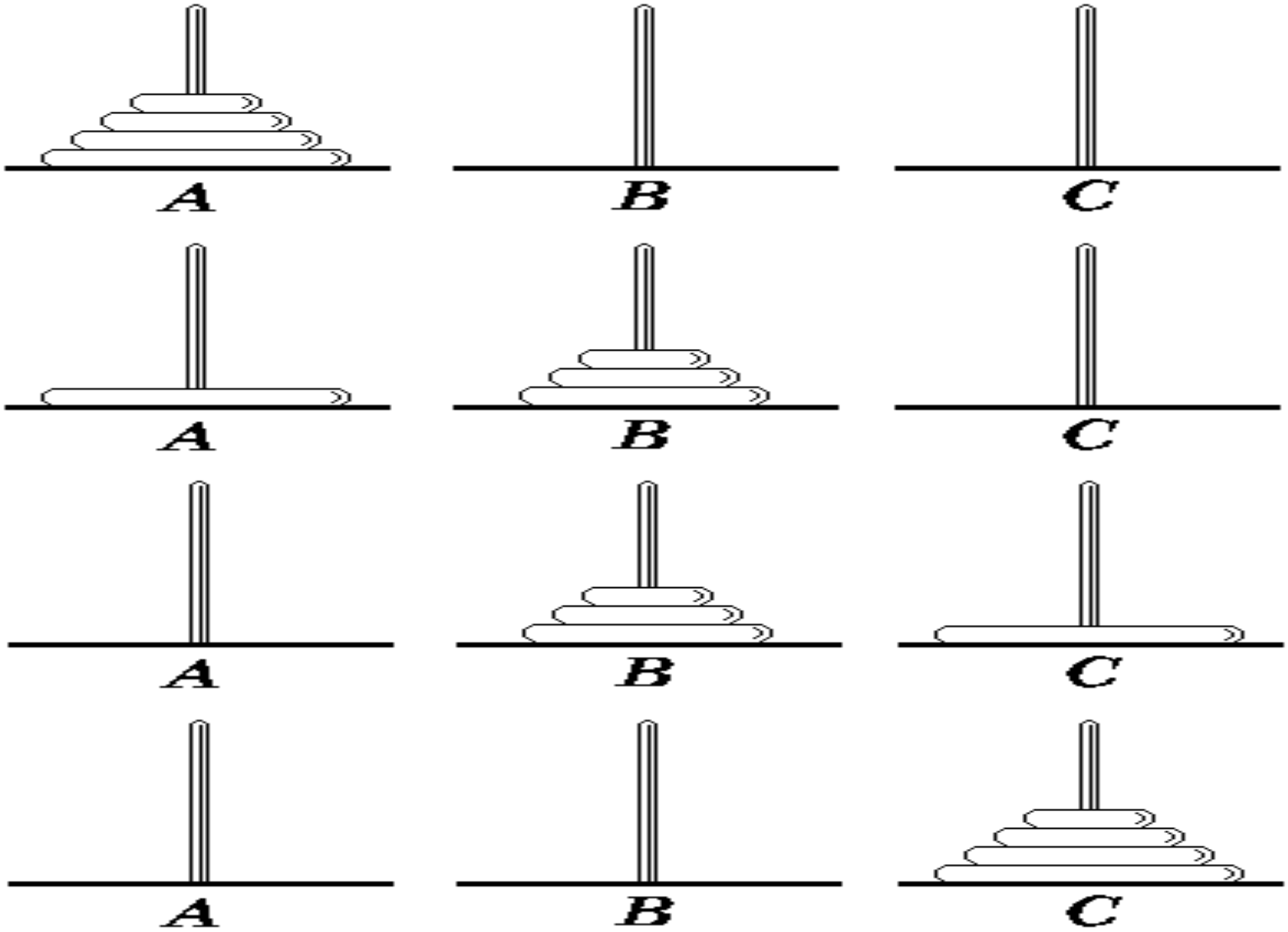
```
template <class Type>
void Find ( ListNode<Type> *f ) {
    if ( f→link == NULL )
        cout << f→data << endl;
    else Find ( f→link );
}
```

在链表中寻找等于给定值的结点并打印其数值

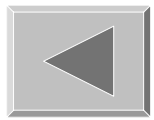
```
template <class Type> void Print
( ListNode<Type> *f, Type& x ) {
    if ( f != NULL)
        if ( f → data == x )
            cout << f→ data << endl;
        else Print ( f→ link, x );
}
```

问题的解法是递归的

例如，汉诺塔(Tower of Hanoi)问题



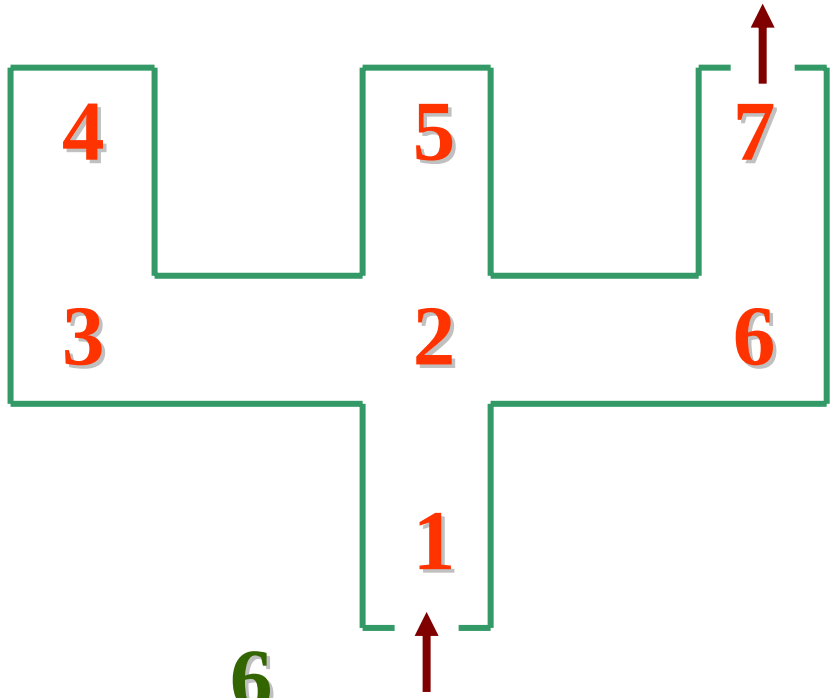
```
#include <iostream.h>
#include "strclass.h"
void Hanoi (int n, String A, String B, String C ) {
 //解决汉诺塔问题的算法
    if ( n == 1 ) cout << " move " << A << " to "
          << C << endl;
    else {  Hanoi ( n-1, A, C, B );
            cout << " move " << A << " to " << C
                 << endl;
            Hanoi ( n−1, B, A, C );
          }
}
```



迷宫问题

小型迷宫

路口	动作	结果
1 (入口)	正向走	进到 2
2	左拐弯	进到 3
3	右拐弯	进到 4
4 (堵死)	回溯	退到 3
3 (堵死)	回溯	退到 2
2	正向走	进到 5
5 (堵死)	回溯	退到 2
2	右拐弯	进到 6
6	左拐弯	进到 7 (出口)



小型迷宫的数据

6		
左行	直行	右行
0	2	0
3	5	6
0	0	4
0	0	0
0	0	0
7	0	0
7		

迷宫的类定义

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
class Maze {
private:
    int MazeSize;
    int EXIT;
    Intersection *intsec;
public:
    Maze ( char *filename );
    int TraverseMaze ( int CurrentPos );
}
```

交通路口结构定义

```
struct Intersection {
    int left;
    int forward;
    int right;
}
```

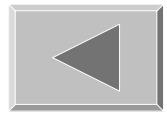
```
Maze :: Maze ( char *filename ) {
//构造函数：从文件 filename 中读取各路口
//和出口的数据
   ifstream fin;
   fin.open ( filename, ios::in | ios::nocreate );
   //为输入打开文件,文件不存在则打开失败
   if ( !fin ) {
       cout << “迷宫数据文件” << filename
            << “打不开” << endl;
       exit (1);
    }
    fin >> MazeSize;              //输入迷宫路口数
```

```
    intsec = new Intersection[MazeSize+1];
    //创建迷宫路口数组
    for ( int i = 1; i <= MazeSize; i++ )
       fin >> intsec[i].left >> intsec[i].forward
           >> intsec[i].right;
    fin >> EXIT;                    //输入迷宫出口
    fin.close ( );
}
```

迷宫漫游与求解算法

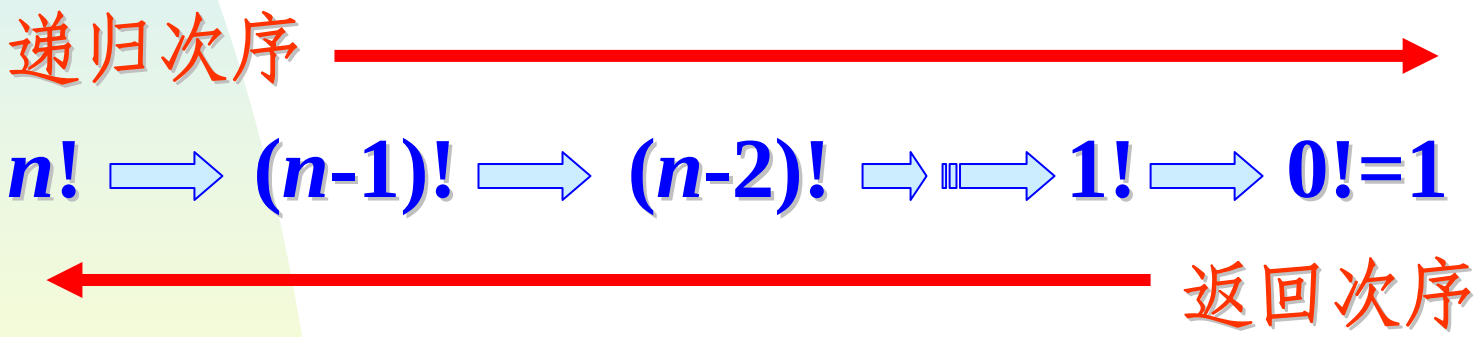
```
int Maze::TraverseMaze ( int CurrentPos ) {
  if ( CurrentPos > 0 ) {      //路口从 1 开始
```

```
    if ( CurrentPos == EXIT ) {     //出口处理
      cout << CurrentPos << "   "; return 1;
    }
    else          //递归向左搜寻可行
    if (TraverseMaze(intsec[CurrentPos].left ))
      { cout << CurrentPos << "   "; return 1; }
    else          //递归向前搜寻可行
    if (TraverseMaze(intsec[CurrentPos].forward))
      { cout << CurrentPos << "   "; return 1; }
    else          //递归向右搜寻可行
    if (TraverseMaze(intsec[CurrentPos].right))
      { cout << CurrentPos << "   "; return 1;  }
  }
  return 0;
}
```



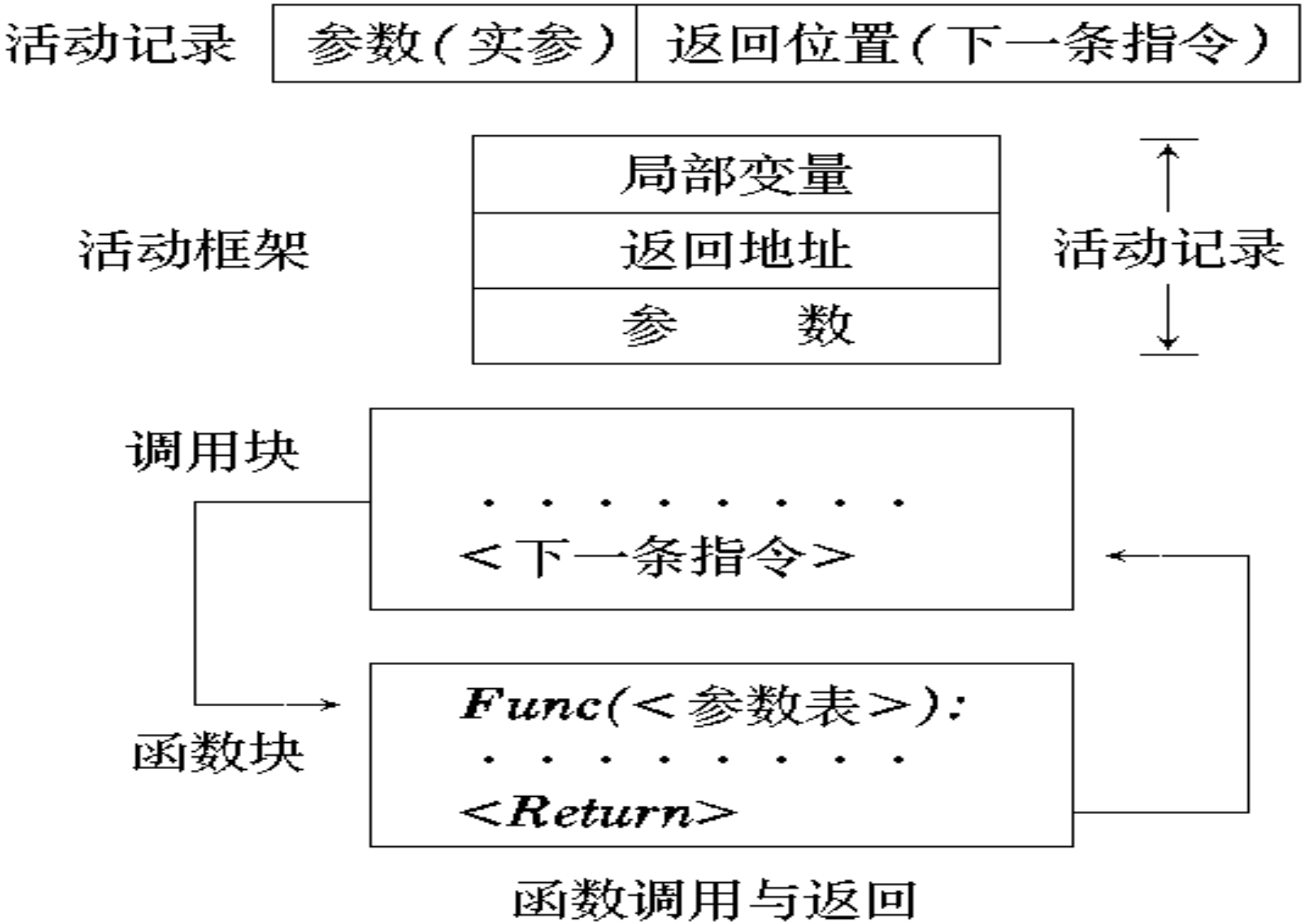
递归过程与递归工作栈

- 递归过程在实现时，需要自己调用自己。
- 每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。
- 层层向下递归，退出时的次序正好相反：



- 因此，每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。

函数递归时的活动记录



函数调用与返回

```
long Factorial ( long n ) {
    int temp;
    if ( n == 0 ) return 1;
    else temp = n * Factorial (n−1);
    return temp;
}

void main ( ) {
    int n;
    n = Factorial (4);
}
```

RetLoc2

RetLoc1

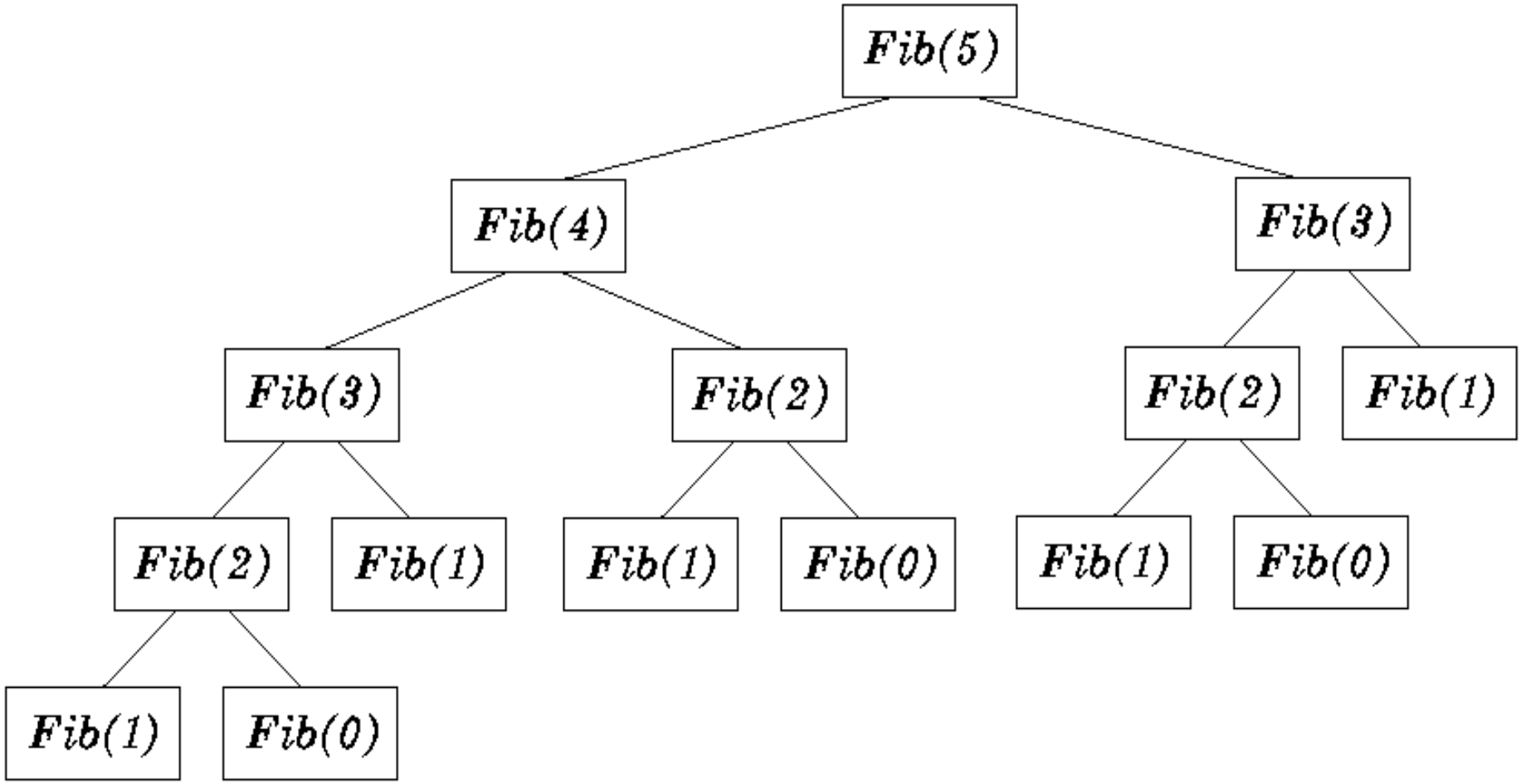
计算*Factorial*时活动记录的内容

活动记录	参数 *long n*	返回地址<下一条指令>

递归调用序列

参数	返回地址	返回时的指令
4	*RetLoc1*	*RetLoc1* `n = Factorial(4);` //返回到*Main*程序
3	*RetLoc2*	*RetLoc2* `temp= 4*6;` //从*Factorial(3)*得到*6* `return temp;` //*temp=24;*
2	*RetLoc2*	*RetLoc2* `temp= 3*2;` //从*Factorial(2)*得到*2* `return temp;` //*temp=6;*
1	*RetLoc2*	*RetLoc2* `temp= 2*1;` //从*Factorial(1)*得到*1* `return temp;` //*temp=2;*
0	*RetLoc2*	*RetLoc2* `temp= 1*1;` //从*Factorial(0)*得到*1* `return temp;` //*temp=1;*

递归返回次序



斐波那契数列的递归调用树

调用次数 $NumCall(k) = 2*Fib(k+1) - 1$。

打印数组A[n]的值

```
void recfunc ( int A[ ],  int n ) {
   if ( n >= 0 ) {
         cout << A[n] << "  ";   n--;
         recfunc ( A, n );
   }
}
```

```
void iterfunc ( int A[ ], int n ) {
//消除了尾递归的非递归函数
   while ( n >= 0 ) {
        cout << "value    " << A[n] << endl;
         n--;
    }
}
```

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
- 有长度
- 有深度
- 可递归
- 可共享

$A = ( )$

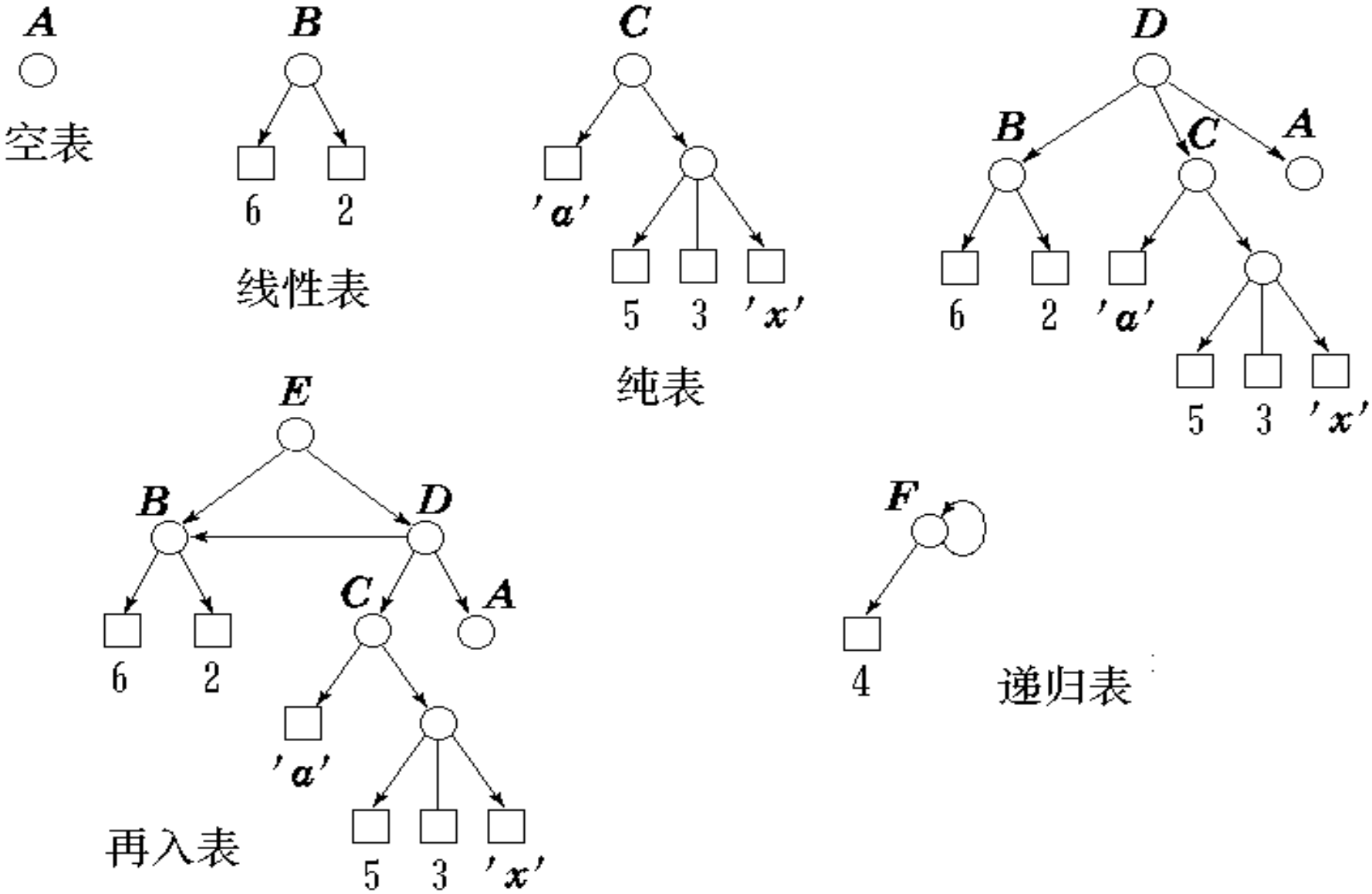
$B = (6, 2)$

$C = (\text{‘}a\text{’}, (5, 3, \text{‘}x\text{’}))$

$D = (B, C, A)$

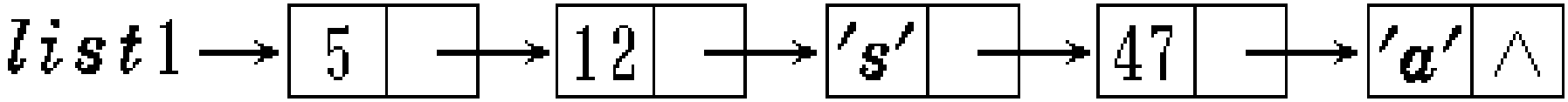
$E = (B, D)$

$F = (4, F)$

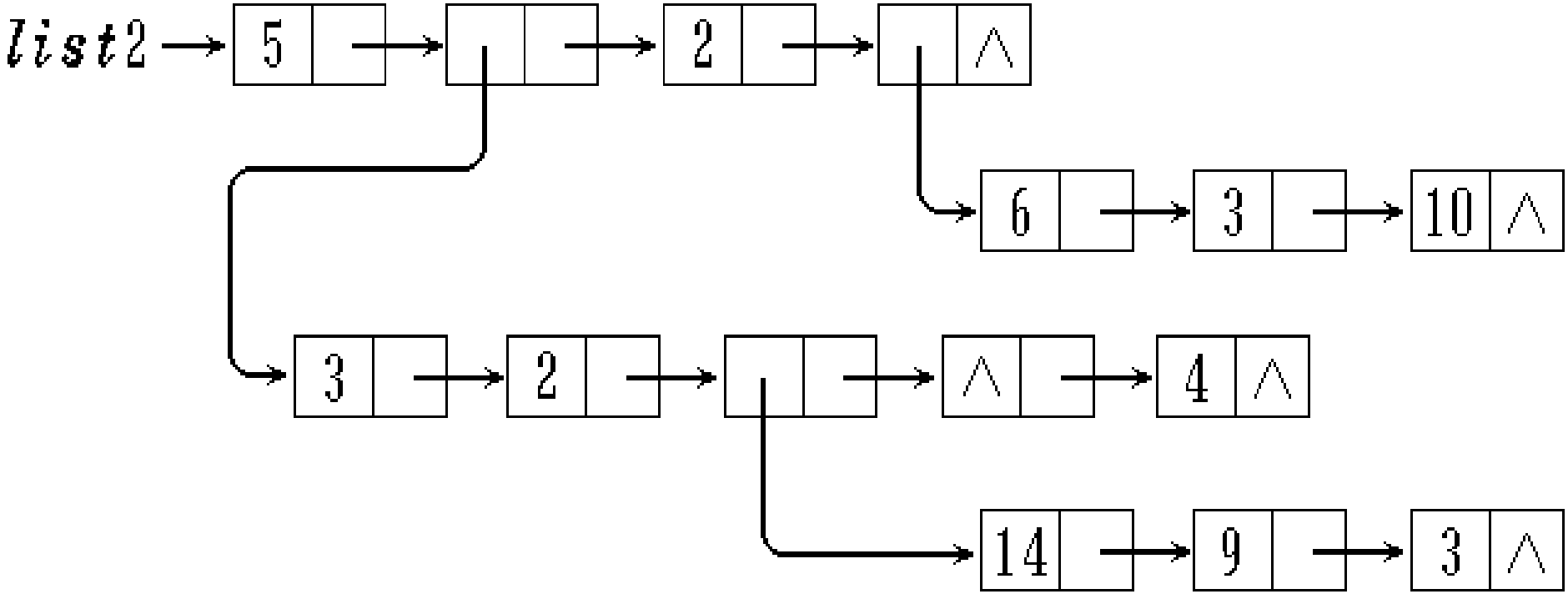


各种广义表的示意图

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



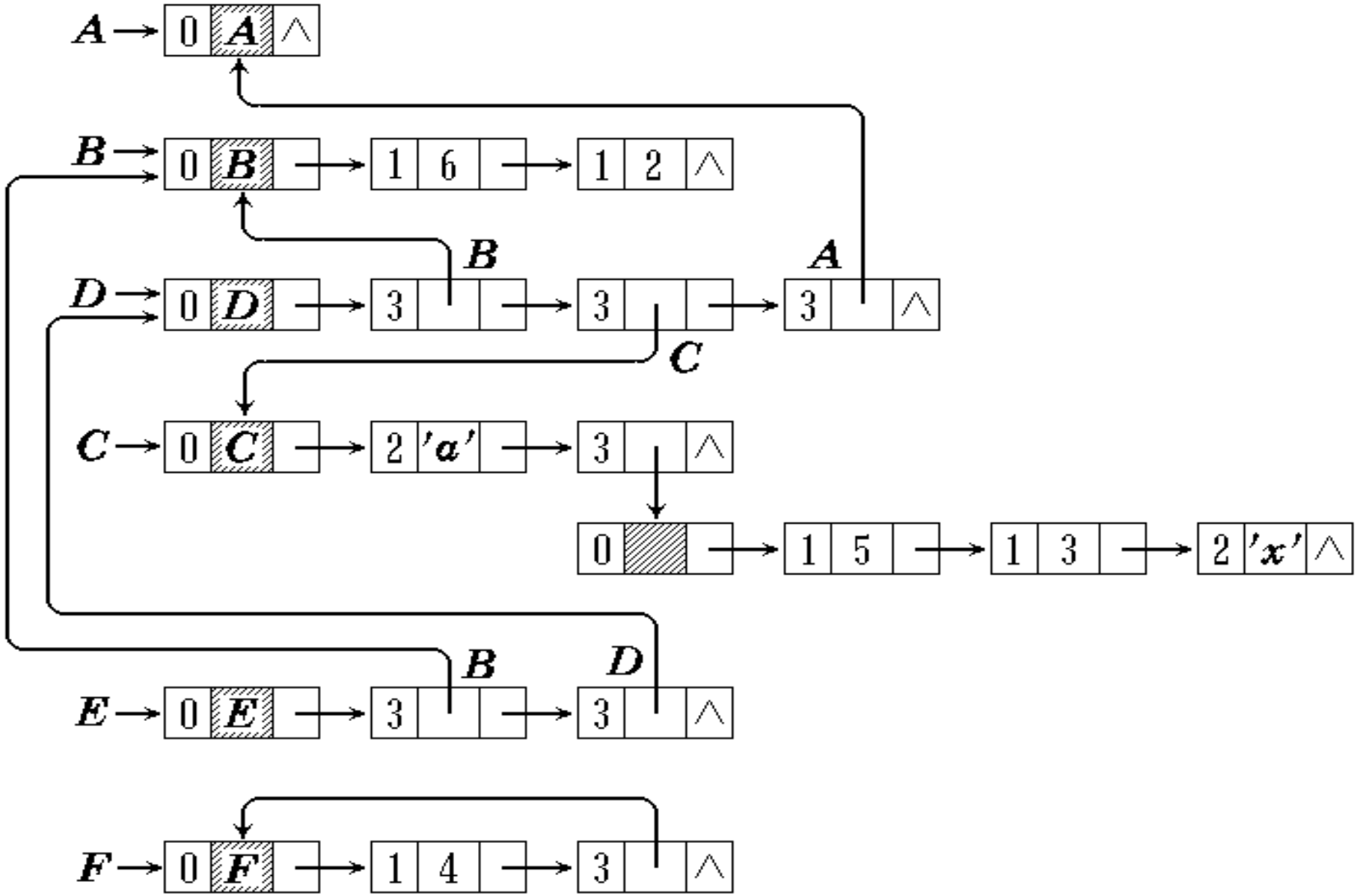
表中套表情形下的广义表链表表示

广义表结点定义

utype = 0/1/2/3	*value* = *ref* /*intgrinfo* /*charinfo* / *hlink*	*tlink*

- 标志域 ***utype***, 表明结点类型。**0**为表头结点，**1**为整型原子结点，**2**为字符型原子结点，**3**为子表结点。
- 值域 ***value***。当 ***utype*** = **0** 时为表引用计数，= **1** 时为整数值，= **2** 时为字符值，= **3** 时为指向子表的表头结点的指针。
- 尾指针域 ***tlink***。当 ***utype*** = **0** 时为指向该表表头元素的指针；当 ***utype*** ≠ **0** 时为指向同一层下一个表结点的指针。

广义表的带表头结点的存储表示



广义表的类定义

```
#define HEAD   0
#define INTGR  1
#define CH   2
#define LST  3
class GenList;

class GenListNode {          //广义表结点类
friend class Genlist;
private:
    int utype;
    GenListNode * tlink;
```

```
    union {
        int ref;                    //utype = 0, 表头结点
        int intgrinfo;              //utype = 1, 整型
         char charinfo;             //utype = 2, 字符型
        GenListNode *hlink;         //utype = 3, 子表结点
    } value;
public:
    GenListNode & Info ( GenListNode *elem );
    int nodetype ( GenListNode *elem )
        { return elem→utype; }
    void setInfo ( GenListNode *elem,
        GenListNode *x );
};
```

```
class GenList {                          //广义表类
private:
    GenListNode* first;                  //广义表表头指针
    GenListNode* Copy ( GenListNode* ls );
    int depth ( GenListNode *ls );
    int equal ( GenListNode *s, GenListNode *t );
    void Remove (GenListNode *ls );
public:
    Genlist ( );
    ~GenList ( );
    GenListNode& Head ( );
    GenListNode& Tail ( );
```

```
    GenListNode *First ( );
    GenListNode *Next ( GenListNode *elem );
    void Push ( GenListNode& x );
    GenList & Addon ( GenList& list,
          GenListNode& x );
    void setHead ( GenListNode& x );
    void setNext ( GenListNode *elem1,
          GenListNode *elem2 );
    void setTail ( GenList& list );
    void Copy ( const GenList& l );
    int depth ( );
    int Createlist ( GenListNode* ls, char* s );
}
```

广义表的访问算法

广义表结点类的存取成员函数

```
GenListNode& GenListNode ::
Info (GenListNode* elem ) {
//提取广义表中指定表元素elem的值
    GenListNode * pitem = new GenListNode;
    pitem→utype = elem→utype;
    pitem→value = elem→value;
    return * pitem;
}
```

```
void GenListNode :: setInfo(GenListNode* elem,
     GenListNode& x ) {
//将表元素elem中的值修改为x
    elem→utype = x→utype;
    elem→value = x→value;
}
```

广义表类的构造和访问成员函数

```
Genlist :: GenList ( ) {
    GenListNode *first = new GenListNode;
    first→utype = 0;  first→ref = 1;
    first→tlink = NULL;     //仅建立表头结点
}
```

```
GenListNode& GenList :: Head ( ) {
//若广义表非空，返回表的表头元素的值
    if ( first→tlink == NULL ) {      //空表
        cout << "无效的 head 操作." << endl;
        exit (1);
    }
    else {
        GenListNode * temp = new GenListNode;
        temp→utype = frist→tlink→utype;
        temp→value = frist→tlink→value;
        return  *temp;
    }
}
```

```
GenListNode& GenList :: Tail ( ) {
//若广义表非空，返回广义表除表头元素外其
//它元素组成的表，否则函数没有定义
    if ( first→tlink == NULL ) {      //空表
        cout << "无效的 Tail 操作." << endl;
        exit (1);
    }
    else {
        GenListNode * temp = new GenListNode;
        temp→frist→tlink = frist→tlink→tlink;
        return  *temp;
    }
}
```

```
void GenList :: Push ( GenListNode& x ) {
//将表结点 x 加到表的最前端
    if ( first→tlink == NULL ) first→tlink = x;
    else {
        x→tlink = first→tlink;  first→tlink = x;
    }
}
```

```
GenList & GenList :: Addon ( GenList & list,
                GenListNode& x ) {
//返回一个以x为头，list为尾的新广义表
    GenList * newlist = new GenList;
    newlist→first = Copy ( list.first );
    x→tlink = newlist→frist→tlink;
    newlist→frist→tlink = x;
    return * newlist;
}
```

广义表的递归算法

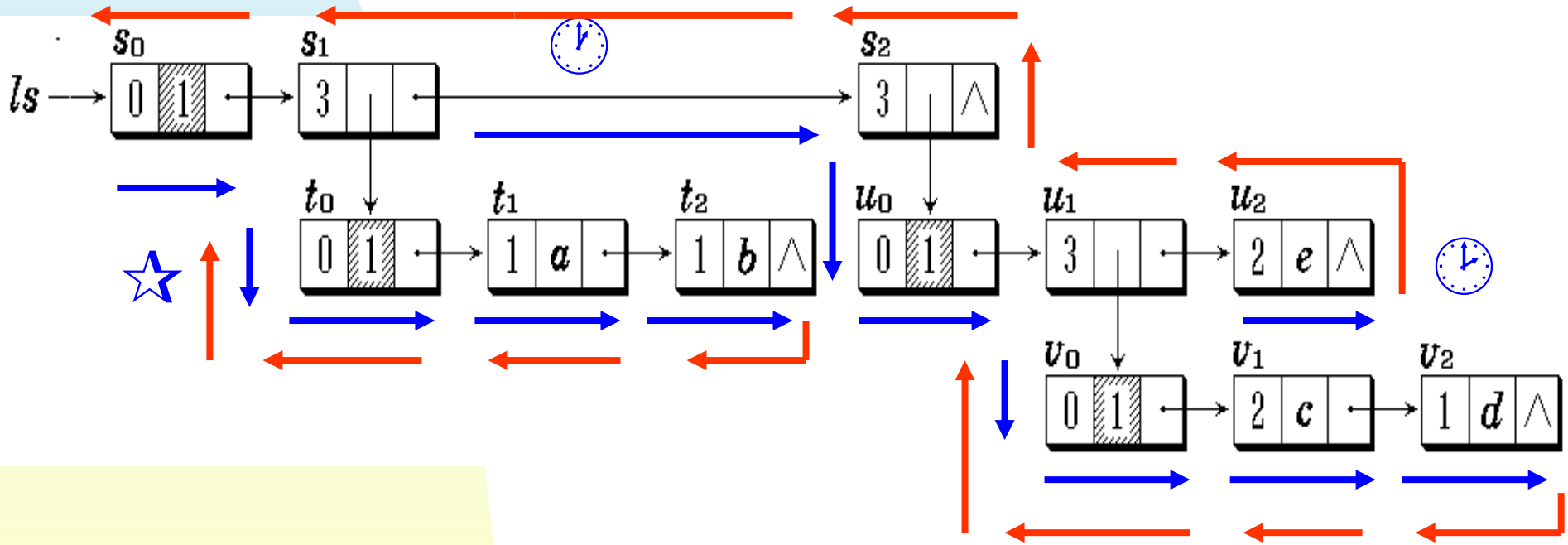
广义表的复制算法

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

GenListNode* GenList :: Copy(GenListNode *ls)
{
    GenListNode *q = NULL;
    if ( ls != NULL ) {
        q = new GenListNode;      //创建表结点
        q→utype = ls→utype;       //复制 utype
```

```
switch ( ls→utype ) {
    case HEAD :
        q→value.ref = ls→value.ref;  break;
    case INTGR :
        q→value.intgrinfo = ls→value.intgrinfo;
        break;
    case CH :
        q→value.charinfo = ls→value.charinfo;
        break;
    case LST :
        q→value.hlink = Copy ( ls→value.hlink );
        break;
}
```

```
        q→tlink = Copy ( ls→tlink );
    }
    return q;
}
```



求广义表的深度

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

例如，对于广义表

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

按递归算法分析:

Depth **(*E*) = 1+*Max* {** ***Depth*** **(*B*),** ***Depth*** **(*D*) }**

Depth **(*B*) = 1+*Max* {** ***Depth*** **(*a*),** ***Depth*** **(*b*) } = 1**

Depth **(*D*) = 1+*Max* {** ***Depth*** **(*B*),** ***Depth*** **(*C*),**

Depth **(*A*)}**

$$Depth\ (C) = 1 + Max\ \{\ Depth\ (u),\ Depth\ ((x, y, z))\ \}$$

$$Depth\ (A) = 1$$

$$Depth\ (u) = 0$$

$$Depth\ ((x, y, z)) = 1 + Max\ \{Depth\ (x),\ Depth\ (y),\ Depth\ (z)\ \} = 1$$

$$Depth\ (C) = 1 + Max\ \{\ Depth\ (u),\ Depth\ ((x, y, z))\ \} = 1 + Max\ \{0, 1\} = 2$$

$$Depth\ (D) = 1 + Max\ \{\ Depth\ (B),\ Depth\ (C),\ Depth\ (A)\} = 1 + Max\ \{1, 2, 1\} = 3$$

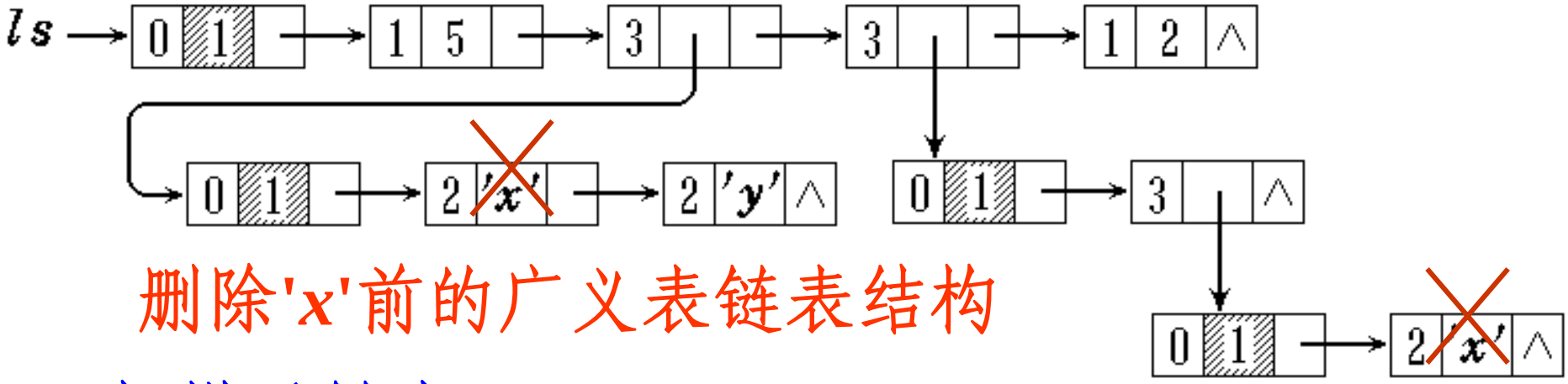
$$\text{Depth (E)} = 1 + Max\ \{\ Depth\ (B),\ Depth\ (D)\ \} = 1 + Max\ \{1, 3\} = 4$$

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

```
int GenList :: depth ( GenListNode *ls ) {
    if ( ls→tlink == NULL ) return 1;   //空表
    GenListNode *temp = ls→tlink;
    int m = 0;
    while ( temp != NULL ) {                    //横扫广义表
        if ( temp→utype == LST ) {      //子表深度
            int n = depth ( temp→value.hlink );
            if ( m < n ) m = n;
        }                   //不是子表不加深度
        temp = temp→tlink;
    }
    return m+1;
}
```

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

广义表的删除算法



删除'x'前的广义表链表结构

- 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 ) {
       GenListNode * p = ls→tlink;   //横扫链表
       while ( p != NULL &&
           ( ( p→utype==INTGR &&
                  p→value.intgrinfo == x ) ||
           ( p→utype==CH &&
                  p→value.charinfo == x ) ) {
          ls→tlink = p→tlink;   delete p;     //删除
          p = ls→tlink;      // p指向同层下一结点
       }
```

```
        //链表检测完或遇到子表或走到子表表
        //头结点或不是含x结点, 转出循环
        if ( p != NULL ) {
          if ( p→utype == LST )      //到子表中删除
             delvalue ( p→value.hlink, x );
          delvalue ( p, x );                //向后递归删除
        }
    }
}

GenList :: ~GenList ( ) {              //析构函数
   Remove ( first );
}
```

```
void GenList :: Remove (GenListNode *ls ) {
    ls→value.ref --;          //引用计数减一
    if ( !ls→value.ref ) {    //引用计数减至0才能删除
      GenListNode * y = ls;
      while ( y→tlink != NULL ) {      //横扫链表
         y = y→tlink;
         if ( y→utype == LST )         //遇到子表
            Remove ( y→value.hlink );  //递归删除
      }
      y→tlink = av;  av = ls;
      //扫描完后,回收到可利用空间表
    }
}
```

从字符串*s*建立广义表的链表表示*ls*

```
int Genlist ::
CreateList ( GenListNode *ls,  char * s ) {
   char *sub, *hsub;  int tag;
   ls = new GenListNode ( );    //建立表头结点
   ls→utype = HEAD;  ls→value.ref = 1;
   if ( strlen (s) == 0 || !strcmp ( s,"( )" ) )
      ls→tlink = NULL;          //空表
   else {
      strncpy ( sub, s+1, strlen (s)−2 );
      //脱去广义表外面的引号
      GenListNode* p = ls;
```

```
while ( strlen (sub) != 0 ) {   //建立表结点
  p = p→tlink = new GenListNode ( );
  //创建新结点,向表尾链接
  if ( sever ( sub, hsub ) ) {
  //以逗号为界,分离第一个表元素hsub
    if ( hsub[0] != '(' && hsub[0] != '\0' ) {
     //非子表,非字符分界符
       p→utype = INTGR;      //转换整型结点
       p→value.intgrinfo = atoi ( hsub );
    }
    else
    if ( hsub[0] != '(' && hsub[0] == '\0' ) {
    //非子表且是字符分界符
```

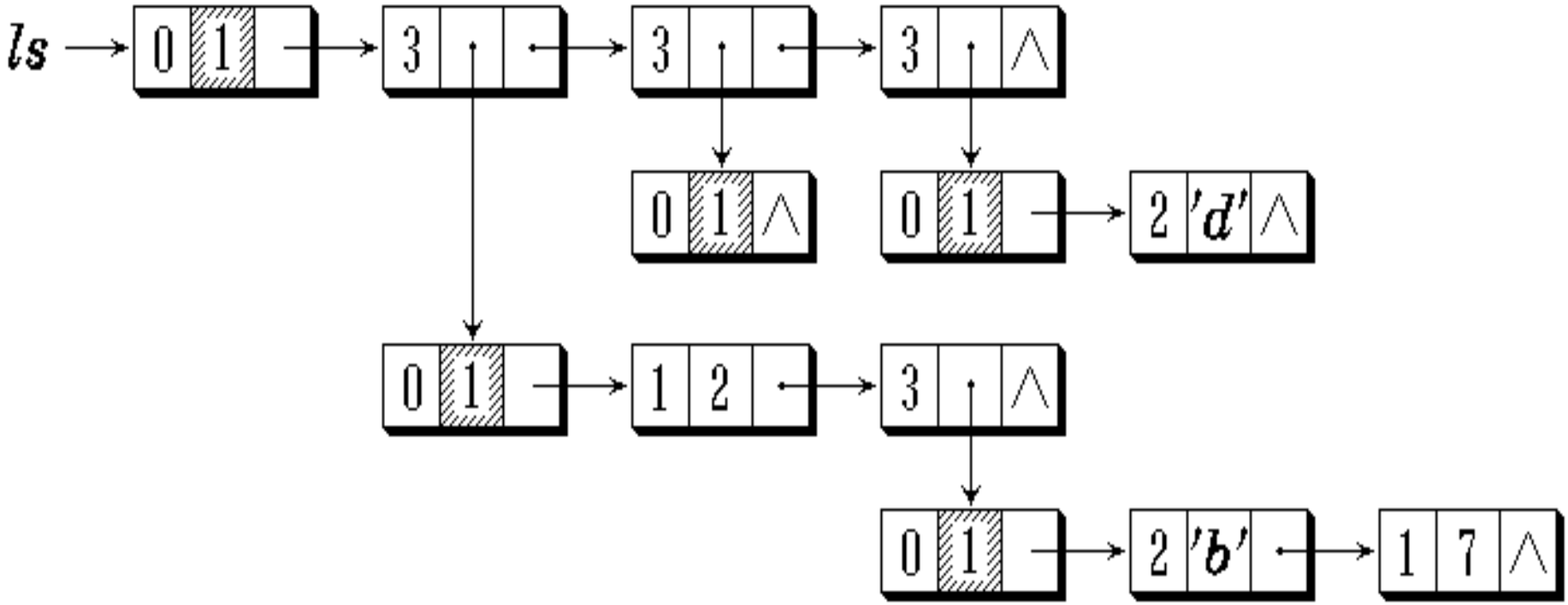
```
            p→utype = CH;
            p→value.charinfo = hsub;
        }
        else {                    //是子表,递归建立子表
          p→utype = LST;
          CreateList ( p→value.hlink, hsub );
        }
      }
      else return 0;     //字符串表达式有错
    }      //循环完成
    p→tlink = NULL;  //封闭最后一个结点
  }
  return 1;
}
```

```
int Genlist::sever ( char *str1, char *hstr1 ) {
//对不含外分界符的字符串分离第一个元素
    char ch = str1[0];
    int n = strlen ( str1 );
    int i = 0,  k = 0;
    //检测str1,扫描完或遇到',' 且括号配对则停
    while ( i < n && ( ch != ',' || k != 0 ) ) {
        if ( ch == '(' ) k++;
        else if ( ch == ')' ) k--;
        i++;  ch = str1[i];          // i 计数, 取一字符
    }
```

```
    if ( i < n ) {
        strncpy ( hstr1, str1, i−1 );
        //str1的前 i−1 个字符传送到hstr1
        strncpy ( str1, str1+i, n−i );
        //后n−i个字符留在str1, 滤去‘,’
        return 1;
    }
    else if ( k != 0 ) return 0;   //括号不配对出错
          else {                   //括号配对
            strcpy ( hstr1, str1 );  str1 = 0;
            //str1全部传送给hstr1
            return 1;
          }
}
```

设 *str* = ‘((2, (‘*b*’, 7)), ( ), (‘*d*’))’

得到的广义表链表结构



小结　需要复习的知识点

- 递归概念:
  - 什么是递归?
  - 递归的函数定义
  - 递归的数据结构
  - 递归问题的解法
- 链表是递归的数据结构
- 可用递归过程求解有关链表的问题

- 递归实现时栈的应用
  - 递归求解思路
  - 递归过程与递归工作栈：递归过程实现的机制及递归工作栈的引用
- 递归的分层(树形)表示 :递归树
- 递归深度(递归树的深度)与递归工作栈的关系
- 单向递归与尾递归的迭代实现

- 广义表
  - 广义表定义
  - 广义表长度、深度、表头、表尾
  - 广义表的表示及操作
- 用图形表示广义表的存储结构
- 广义表的递归算法

递归举例

【例1】求解斐波那契数列的递归算法

```
long Fib ( long n ) {
    if ( n <= 1 ) return n;
    else return Fib (n−1) + Fib (n−2);
}
```

考虑使用递归算法求解的思路

- 递归算法的一般形式

```
void p ( 参数表 )
    if ( 递归结束条件)
        可直接求解步骤;        基本项
    else p ( 较小的参数 );     递归项
```

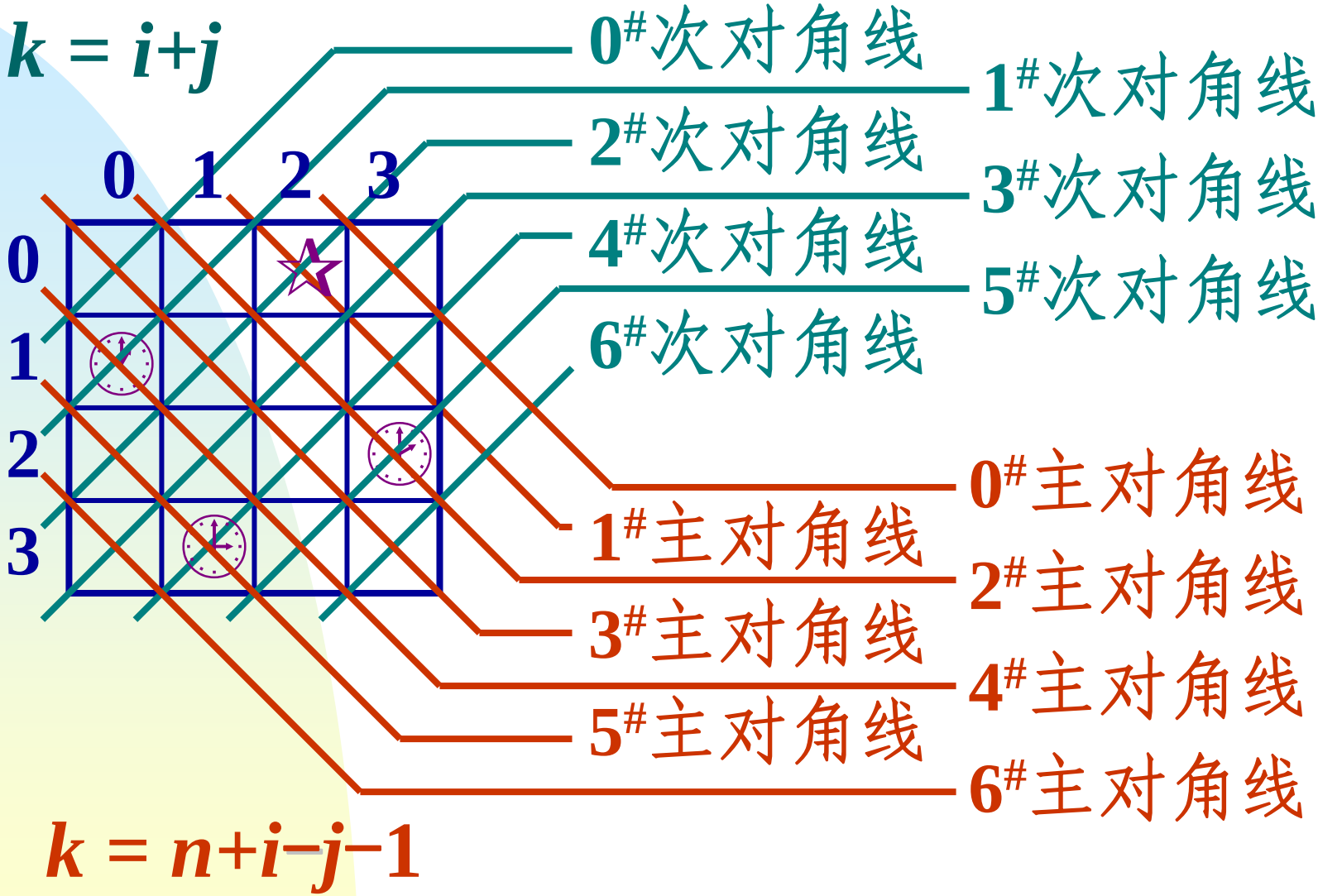
【例2】求数组 *A* 中 *n* 个整数的和

```
int sum ( int n ) {
    int result;
    if ( n == 1 ) result = A[0];
    else
        result = A[n−1] + sum(n−1);
    return result;
}
```

递归与回溯　常用于搜索过程

【例3】*n*皇后问题

在 *n* 行 *n* 列的国际象棋棋盘上，若两个皇后位于同一行、同一列、同一对角线上，则称为它们为互相攻击。*n*皇后问题是指找到这 *n* 个皇后的互不攻击的布局。


k = i+j
0 1 2 3
0
1
2
3
0#次对角线
1#次对角线
2#次对角线
3#次对角线
4#次对角线
5#次对角线
6#次对角线
0#主对角线
1#主对角线
2#主对角线
3#主对角线
4#主对角线
5#主对角线
6#主对角线
k = n+i−j−1

解题思路

- 安放第 *i* 行皇后时，需要在列的方向从 1 到 *n* 试探 (*j* = 1, …, *n*)
- 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。

- 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

```
void Queen( int i ) {
  for ( int j = 1; j <= n; j++ ) {
    if ( 第 i 行第 j 列没有攻击 ) {
      在第 i 行第 j 列安放皇后;
      if ( i == n ) 输出一个布局;
      else Queen ( i+1 );
    }
    撤消第 i 行第 j 列的皇后;
  }
}
```

算法求精

```
void Queen( int i ) {
  for ( int j = 1;  j <= n;  j++ ) {
    if ( !col[j] && !md[n+i−j−1] && !sd[i+j] )
    {
        /*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 );
    }
    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*/
  }
}
```