第五章 递归和广义表

- 递归的概念
- 递归转化为非递归
- 广义表
- 本章小结

5.1 递归的概念

- 能采用递归解决的问题通常具有这样的特征
  - 规模为N的问题可以被分解为一些规模较小的问题，然后从这些小问题的解可以方便地构造出大问题的解。而这些小问题可以同样的方式继续分解成规模更小的问题，当问题分解到足够小的时候，例如N=1，能直接得到解。
  - 数据结构可以是递归的。
  - 问题的解法可以是递归的。
    - 一个过程直接地或间接地调用其自身。最明显可以用递归解决的问题是问题本身的定义是递归的，或者其中的主要数据结构是递归定义的。

- 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);
}
```

- 递归过程可看作两个过程的组合。
  - 第一个过程是大问题逐渐被分解成小问题，直到递归终止条件，然后开始第二个过程。
  - 第二个过程则是从小问题的解合成出大问题的解。

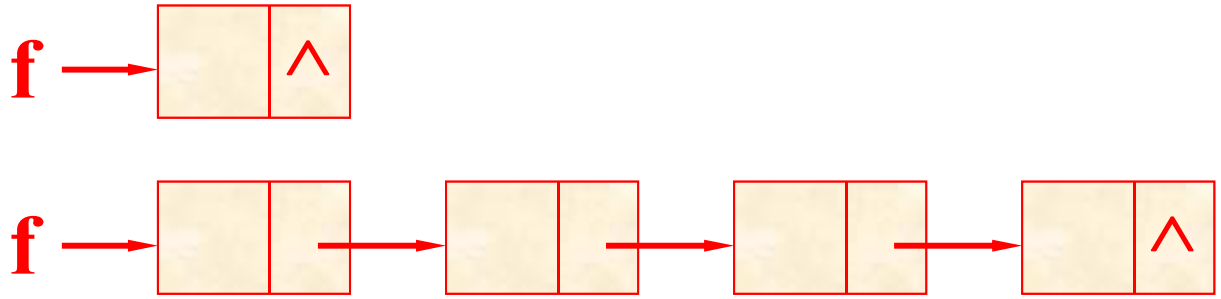
求解阶乘*n*!的过程


主程序 main : fact(4)
参数 4　计算 4*fact(3)　返回 24
参数 3　计算 3*fact(2)　返回 6
参数 2　计算 2*fact(1)　返回 2
参数 1　计算 1*fact(0)　返回 1
参数 0　直接定值 = 1　返回 1
递归调用
参数传递
结果返回
回归求值

递归的数据结构

- 很多数据结构都可以用递归的方式描述。
  - 例如，单链表结构
    - 只有一个结点的单链表，是一个单链表；
    - 一个普通单链表，可以看做由链表的第一个结点，其指针指向由剩余部分形成的一个单链表。
  - 深度为***n***的栈也可以描述为由一个深度为***n*-1**的栈和当前栈顶元素构成，而深度为**0**的栈则是空栈。
  - 链表、树、二叉树都可以类似定义。

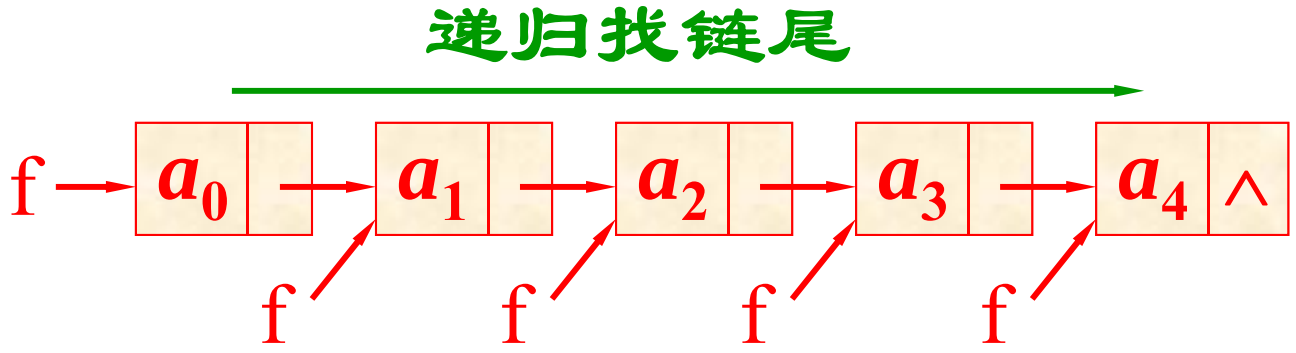
例如，单链表结构



- 一个结点，它的指针域为**NULL**，是一个单链表；
- 一个结点，它的指针域指向单链表，仍是一个单链表。

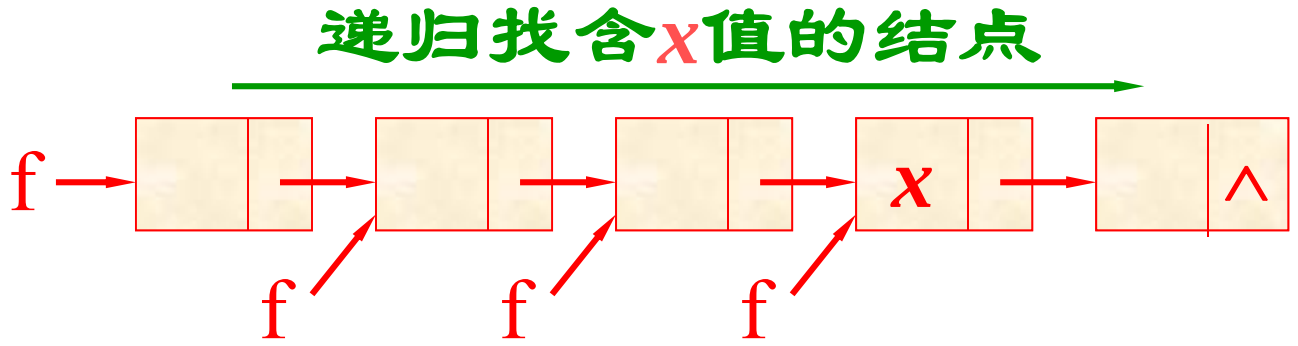
搜索链表最后一个结点并打印其数值

```
template <class Type>
void Print(ListNode <Type> *f) {
    if (f->link==NULL)
        cout<<f->data<<endl;
    else Print(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)问题的解法

如果*n*=1，则将这一个盘子直接从A柱移到C柱上。否则，执行以下三步：

① 用C柱做过渡，将A柱上的(*n*−1)个盘子移到B柱上；

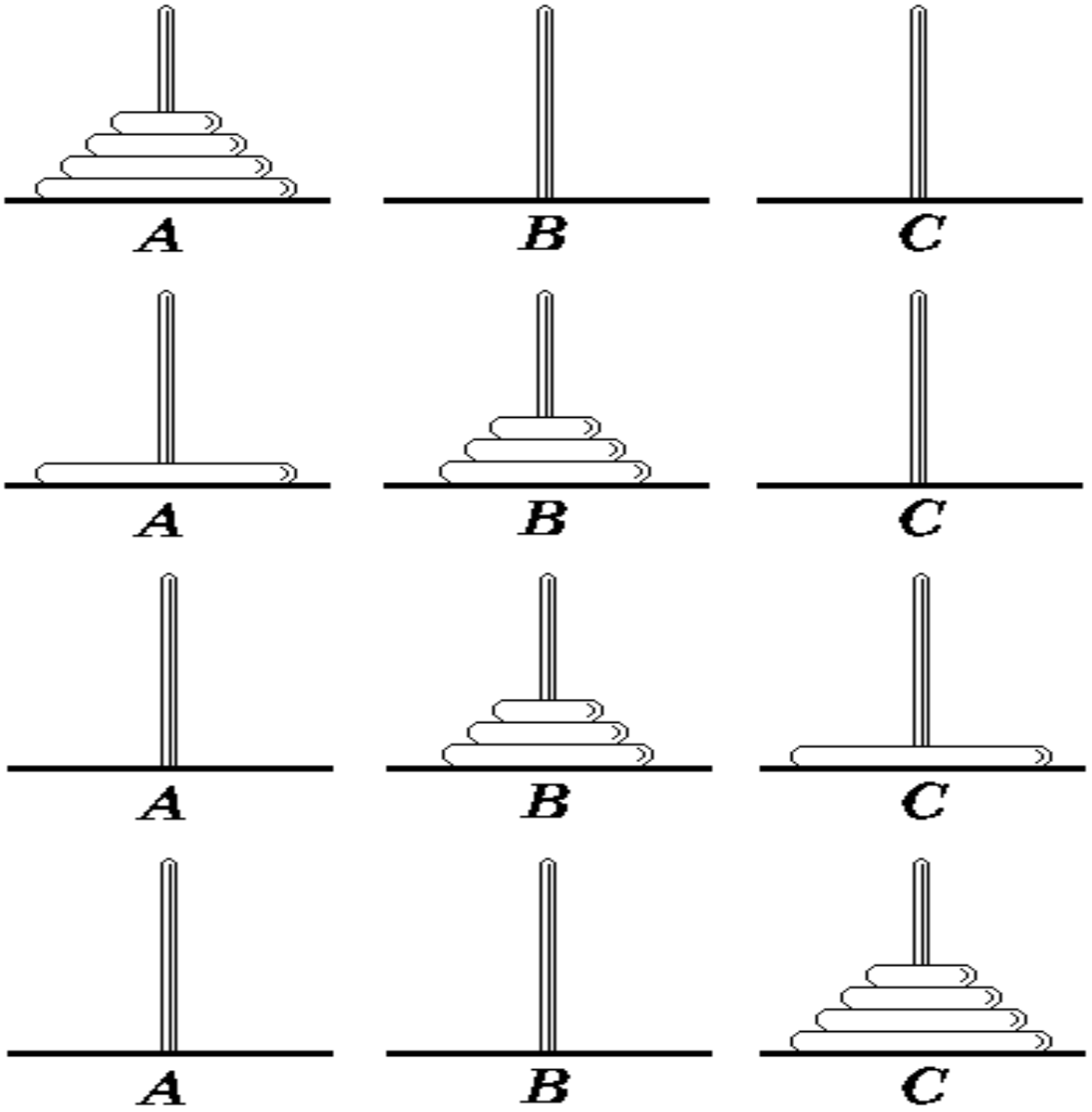
② 将A柱上最后一个盘子直接移到C柱上；

③ 用A柱做过渡，将B柱上的(*n*-1)个盘子移到C柱上。

问题描述——*A*柱上每一个盘子都比下面略小一点，
把*A*柱上的盘子全部移到*C*柱上。

移动条件——①一次只能移动一个盘子，
②移动过程中，大盘不能放在小盘上面。

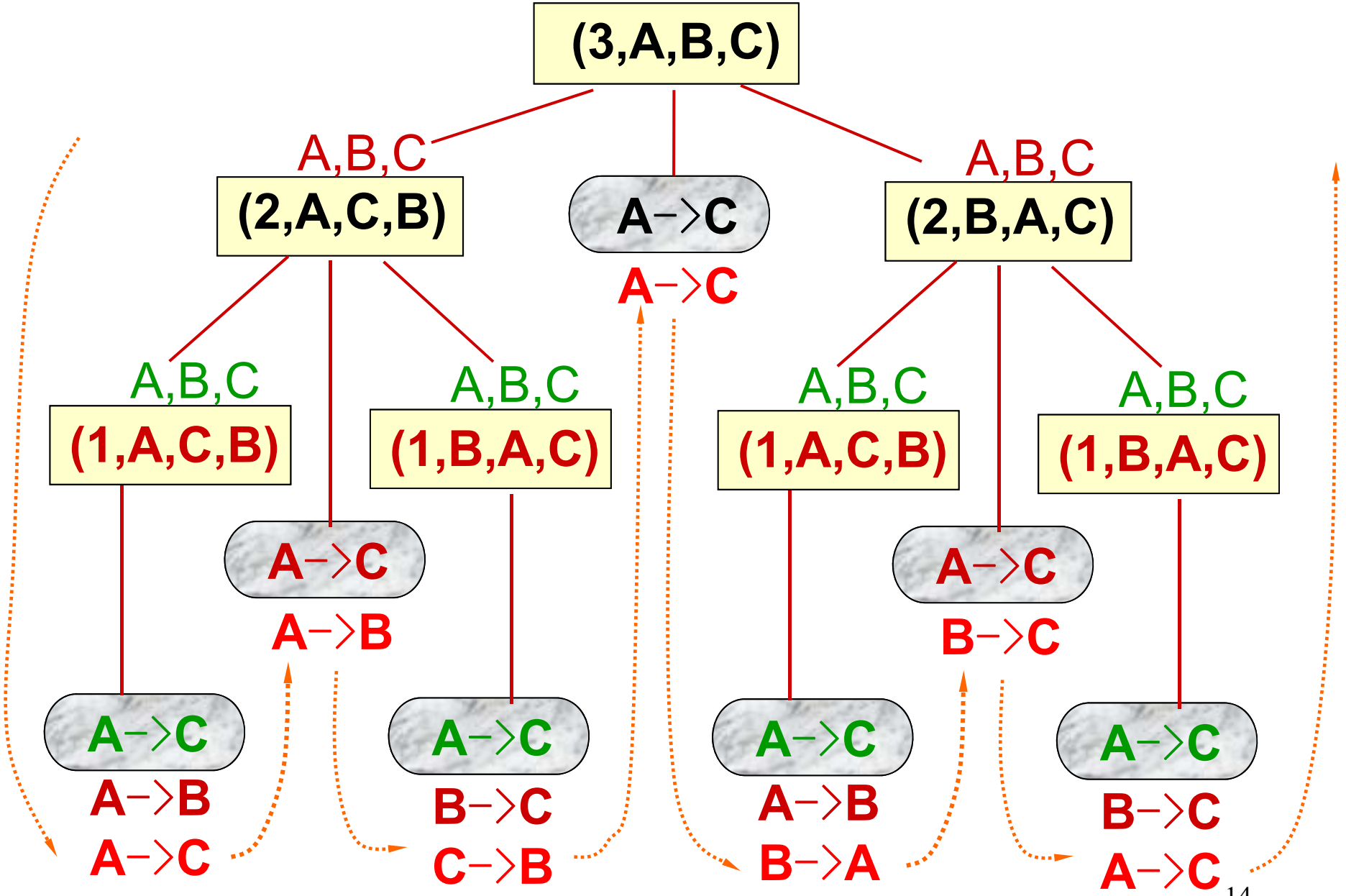
Hanoi塔问题图示



Hanoi塔问题的解法

- 可以把从移动*n*个A针上的金片到B针上的工作分解为
  - 当*n*=1的时候，可把金片直接移到目标针上，不需要利用第三根针的辅助。
  - 否则
    - 先从A利用B移动*n*-1个金片到C上，这是一次递归调用；
    - 然后，把A上的最后一个金片移动到B上；
    - 然后，把C上的*n*-1个金片利用A移动到B上，这也是一次递归调用。

```
#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);
         }
}
```


(3,A,B,C)
A,B,C
(2,A,C,B)
A->C
A->C
A,B,C
(2,B,A,C)
A,B,C
(1,A,C,B)
A,B,C
(1,B,A,C)
A,B,C
(1,A,C,B)
A,B,C
(1,B,A,C)
A->C
A->B
A->C
B->C
A->C
A->B
A->C
B->C
A->C
A->B
A->C
B->C
C->B
B->A
B->C
A->C

递归算法的运算时间复杂度分析

- 当n=0时，T(n)=0；
- 当n>0时，T(n)=2T(n-1)+1；
  - T(n)=2T(n-1)+1=2(2T(n-2)+1)+1=…
    $=2^{n-1}+2^{n-2}+\ldots+1=2^n-1$

证明递归程序的正确性

- 递归程序在逻辑上与常见的数学归纳法类似，因此往往可以用数学归纳法来证明递归程序的正确性。在证明的时候，要证明两个部分。
  - 一个是递归的基础，也就是作为递归终止状态的最基础情况，例如在**Hanoi**塔问题中，***n*=1**的情况。
  - **按照数学归纳法，假设在*n*-1的情况下，Hanoi塔问题的解法是正确的，则由于*n*规模的Hanoi塔问题是通过*n*-1规模的解法解决的，这样就可以很方便地证明Hanoi塔问题的递归算法的正确性。**

自顶向下、逐步分解的策略

- 子问题应与原问题做同样的事情，且更为简单。
- 解决递归问题的策略是把一个规模比较大的问题分解为一个或若干规模比较小的问题，分别对这些比较小的问题求解，再综合它们的结果，从而得到原问题的解。
  — 分而治之策略（分治法）
- 这些比较小的问题的求解方法与原来问题的求解方法一样。

构成递归的条件

- 不能无限制地调用本身，必须有一个出口，化简为非递归状况直接处理。

```
Procedure <name> (<parameter list>)
{
  if (<initial condition>)
    return (initial value);
  else
    return (<name> (parameter exchange));
}
```

递归应用

- 递归与回溯常用于搜索过程
  - 搜索空间通常包含许多结点，且每个结点有多个分支，每次选择一个分支搜索。
  - 当搜索到某一结点无法继续搜索下去时，则沿搜索路径回退到前一结点，选择另一分支继续搜索。
  - 如果回退到的结点没有其它选择，则沿搜索路径继续回退，直到找到新的搜索分支，或者搜索到问题的解，或者搜索完全部可搜索的分支没有找到解。
- 上述问题的特点是将候选解按某种顺序逐一枚举和检验。
  - 当发现当前候选解不是解时，就放弃它而选择下一个，这就是回溯。
  - 求解这类问题时常常以递归方式（或用栈）向前试探和回溯。
    - 之前的迷宫问题是个典型例子。

随堂练习

例1：按以下计算规则求已知两个整数的最大公约数。

$$gcd(a, b) = \begin{cases} a, & a\%b == 0 \\ b, & b\%a == 0 \\ gcd(a\%b, b), & a > b \\ gcd(a, b\%a), & a < b \end{cases}$$

例2：用递归函数实现已知十进制整数返回十进制的反向的整数。如已知十进制数1234，返回反向的十进制数是4321。

例1：按以下计算规则求已知两个整数的最大公约数。

```
//函数以两个已知整数为形参，返回整型结果。
int gcd int a, int b)
{
        if(a % b == 0) return b;
        if(b % a == 0) return a;
    if(a > b) return gca(a%b, b);
    if(a < b) return gcd(a, b % a);
}
```

例2：用递归函数实现已知十进制整数返回十进制的反向的整数。

一个数的反向过程是一个反复将数的个位作为对应的高位的转换过程：设一个数已被译出了一部分，例如，对于1234，已译出部分高位43，还有12还未被转换。问题是如何将12的个位数2转移到43，使新的高位变成432呢？只要将43乘10后加上2即可。若用循环实现有以下函数：

```
int reverInt(int n)
{   int s = 0;
    while(n) {
       s = s*10 + n%10;
       n /= 10;
      }
         return s;
}
```

如果将上述计算改写成递归，需要将循环计算中高位部分的值作为参数。初始调用时，对应s为0值。在被转换的数是个位数情况下，函数的返回值是s*10+n。如果n是多位数，则要继续转换的数变成n/10，新的高位的部分值是s*10+n%10。

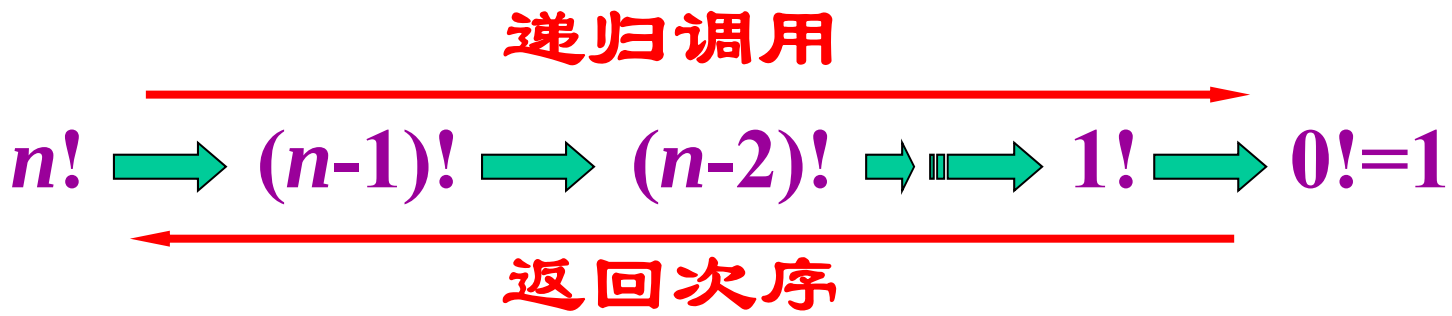
```
int reverIntRe(int n, int s)
{
            if(n<10) return s*10+n;
            return reverIntRe(n/10, s*10 + n % 10);
}
```

5.2 递归转化为非递归

- 递归方法是一种很有效很自然的分析和描述问题的方法。
  - 但递归算法运算效率低，常有重复计算，有时先用递归的思想分析和描述问题，然后转化成非递归的算法。

递归过程与递归工作栈

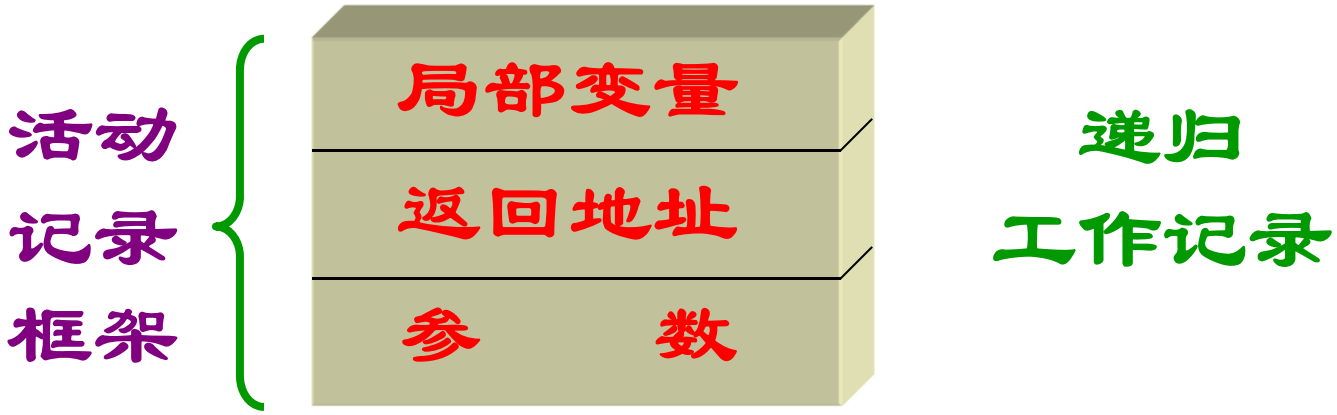
- 递归过程在实现时，需要自己调用自己。
- 层层向下递归，退出时的次序正好相反。



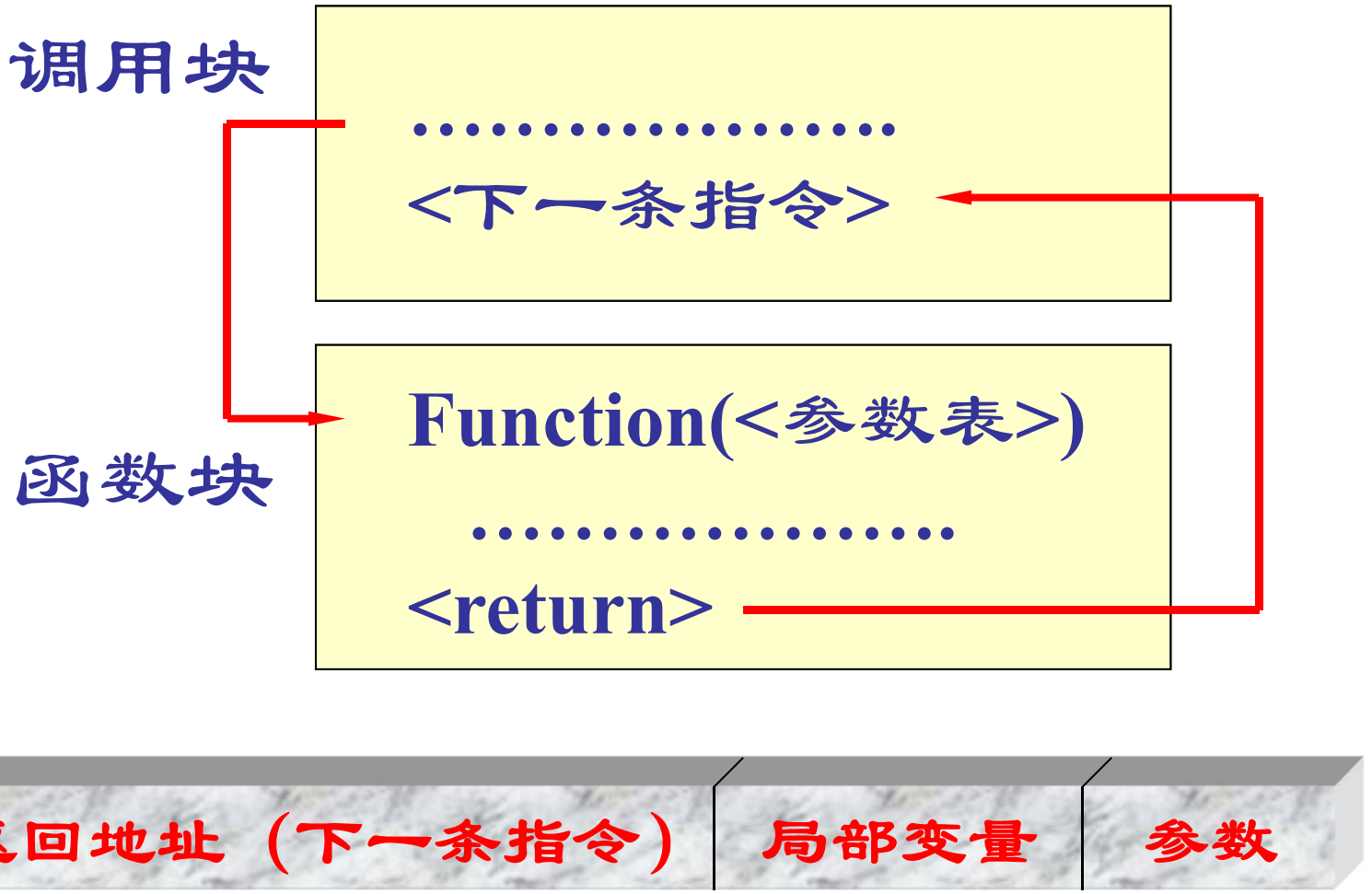
- 主程序第一次调用递归过程为外部调用；
- 递归过程每次递归调用自己为内部调用。
- 它们返回调用它的过程的地址不同。

递归工作栈

- 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。
- 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



```
long Factorial(long n) {
    int temp;
    if (n==0)  return 1;
    else temp=n*Factorial(n-1);
RetLoc2 ──────────────────────┐
     return temp;
  }

void main( ) {
    int n;
    n=Factorial(4);
RetLoc1 ──────┐
}
```

计算Fact时活动记录的内容

递归过程改为非递归过程

- 递归过程简洁、易编、易懂；
- 递归过程效率低，重复计算多；
- 改为非递归过程的目的是提高效率；
- 单向递归和尾递归可直接用迭代实现其非递归过程；
- 其它情形必须借助栈实现非递归过程。

递归转非递归的方法

- 可以分为两个大的方向
  - 一个方向就是寻找与递归算法思路完全不同的非递归算法，这有时候是比较困难的。
  - 另一个方向是仍然利用递归的思路，但进行改造。
    - 以**Hanoi**塔问题为例，通过跟踪递归程序的执行过程，可以发现在递归调用前需要保留程序执行的状态，然后为递归调用进行环境的初始化，进而调用递归程序执行，所调用的过程执行完成时，有时候需要传递返回的值，然后恢复调用前保存的执行状态，然后继续执行。
    - 因此，最基本的递归转换为非递归的方法就是在非递归算法中实现这些程序执行中间状态的保存和恢复。一般情况下，栈是保存这些中间状态的最理想的数据结构。

Hanoi塔问题的递归算法

```
void towers(int n, int A, int B, int C)
{
    if (n==1)  printf("\n%d -> %d\n", A, C);
    else
    {
        towers(n-1, A, C, B);
        printf("\n%d -> %d\n", A, C);
        towers(n-1, B, A, C);
    }
}
```

非递归改造

- 在进行非递归改造时，根据递归算法，按照之前的思路，保存调用的临时状态，就可以构造出对应的非递归算法。
- 在此基础上，进一步消除其中的冗余，并且按结构化编程的要求消除**goto**语句，可以得到结构化良好的非递归的**Hanoi**塔问题解法。
  - 以下算法中
    - **s_tack**过程把它的参数加入栈中。
    - **setvar1(&n, &A, &B, &C)** 把***n***设为***n*-1**，并且交换**C**和**B**的值。
    - **setvar2(&n, &A, &B, &C)** 把***n***设为***n*-1**，并且交换**A**和**B**的值。
    - **restore(&n, &A, &B, &C, &s)**则负责局部变量的恢复。

Hanoi塔问题的非递归算法

```
void towers(int n, int A, int B, int C) {
    stack s;
    int done=FALSE;
    while (!done) {
        while (n>1) {
            s_tack(n, A, B, C, &s);
            setvar1(&n, &A, &B, &C);
        }
        printf("\n %d -> %d\n", A, C);
        if (!empty(&s)) {
            restore(&n, &A, &B, &C, &s);
            printf("\n %d -> %d\n", A, C);
            setvar2(&n, &A, &B, &C);
        }
        else done=TRUE;
    }
}
```

尾递归

- 递归调用处于递归过程的最后一条语句的情况称为尾递归。尾递归通常可以直接转化为循环，循环部分实现递归的从小问题到大问题的集成，而从大问题到小问题的分解过程通常较为简单，已经体现在循环的设计中。

尾递归用迭代法实现

- 尾递归
  - 递归调用语句只有一个，放在过程最后。
  - 递归调用返回时，返回到上一层递归调用语句的下一语句——程序末尾。

25	36	72	18	99	49	54	63

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

- 比尾递归算法复杂一些的是线性递归(**Linear Recursive**)算法，线性递归算法有以下所示的一般形式：

```
proc(int n) {
    Stms1(n);
    if (Eval(n)) {
      proc(n-1); //递归调用
      Stms2(n);
    }
    else
      Stms3(n);
  }
```

- 其中，**Stms1(*n*)**、**Eval(*n*)**、**Stms2(*n*)**和**Stms3(*n*)**是可能与*n*有关的非递归计算的代码段，也可能为空。

- 线性递归能机械地被转换成一个功能等价的非递归描述：

```
proc(int n) {
    int k=n+1;
    do {
        k--;
        Stms1(k);
    } while Eval(k);
    Stms3(k);
    while (k<n) {
        k++;
        Stms2(k);
    }
}
```

- 例如对于求阶乘的问题，如果要考虑得更完善，需要防止在计算过程中发生溢出等异常情况。为此，按照线性递归的一般形式把求阶乘的程序改写为：

```
void factorial(int n, int &temp) {
    if (n<0) { cerr<<n<<"不能为负数"<<endl;  exit(1); }
    if (n>0) {
        factorial(n-1, temp);
        if (Max/temp>n) { temp=temp*n; }
        //Max为允许的最大整数
       else { cerr<<"计算结果溢出"<<endl;  exit(1); }
    }
    else temp=1;
}
```

- 这个程序可以套用线性递归转非递归的模式转换为：

```
void factorial(int n, int &temp) {
    int k=n+1;
    do {
        k--;
        if (k<0) { cerr<<k<<“不能为负数”<<endl;  exit(1); }
    } while (k>0);
    temp=1;
    while (k<n) {
      k++;
      if (Max/temp>k)  temp=temp*k;
      else { cerr<<“计算结果溢出”<<endl;  exit(1); }
   }
}
```

单向递归

- 比线性递归更复杂一些的是单向递归(Single Direction Recursive)。尾递归可以看作一种特殊的单向递归。
  - 普通单向递归可以有多个递归调用，但这些调用都必须处于递归调用过程的最后。
    - 单向递归同时要求递归过程中的这些递归调用的参数由主调用过程的参数决定，而处在同一递归层次的递归调用之间参数不会互相影响。
  - 单向递归同样可以较为方便地转化为循环实现。
    - 由于单向递归中的递归调用处于递归过程的最后，因此同样可以从递归的终止条件开始，通过循环逐渐构造出整个问题的解，每一次循环完成的工作相当于从一个递归层次中返回。
    - 用变量保存中间结果，不需要栈。
  - 以下的例子就是求解斐波那契数列的递归算法和非递归算法。

计算斐波那契数列的函数Fib(n)的定义

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

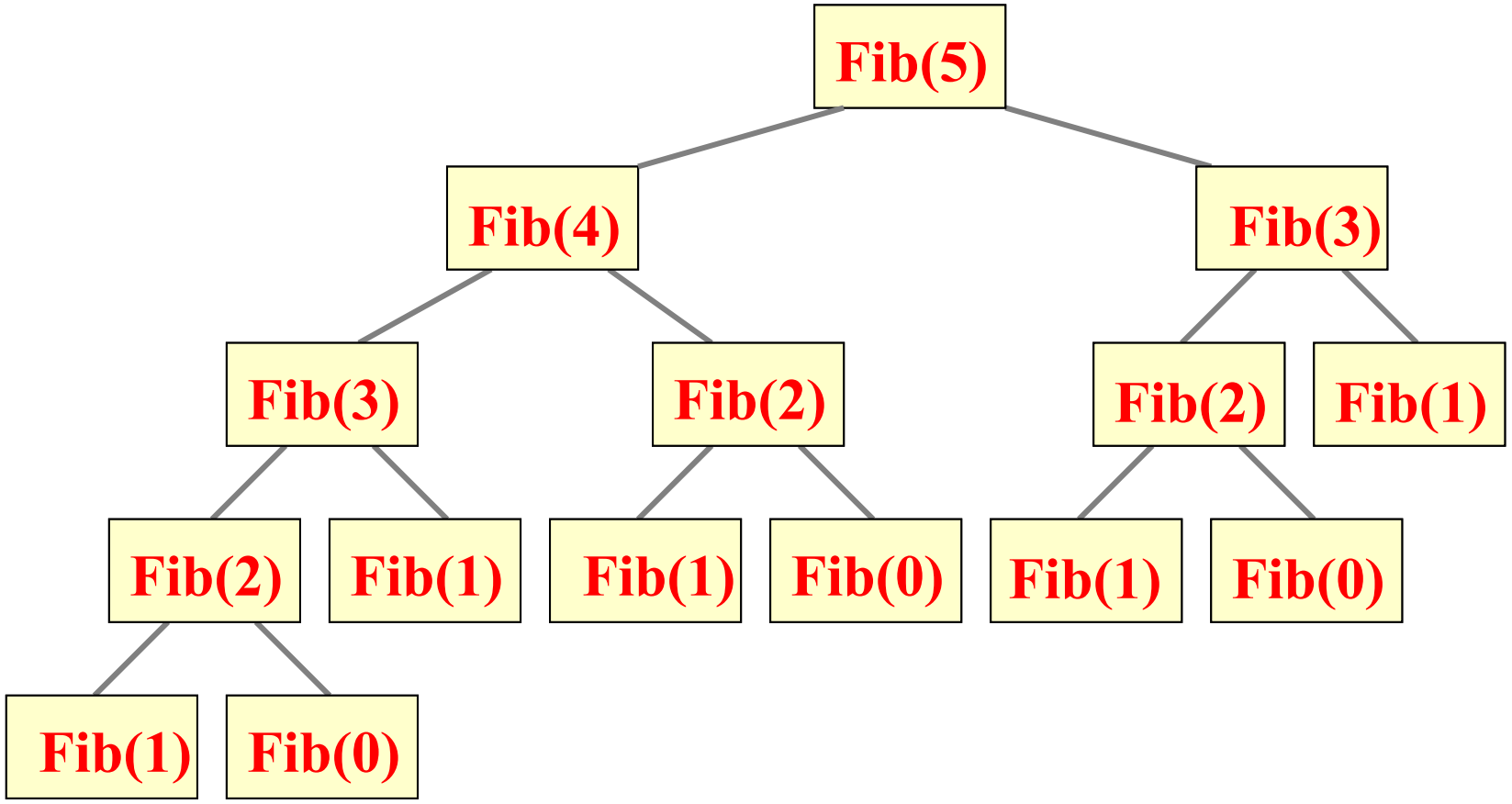
如 $F_0=0$, $F_1=1$, $F_2=1$, $F_3=2$, $F_4=3$, $F_5=5$

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);
}
```

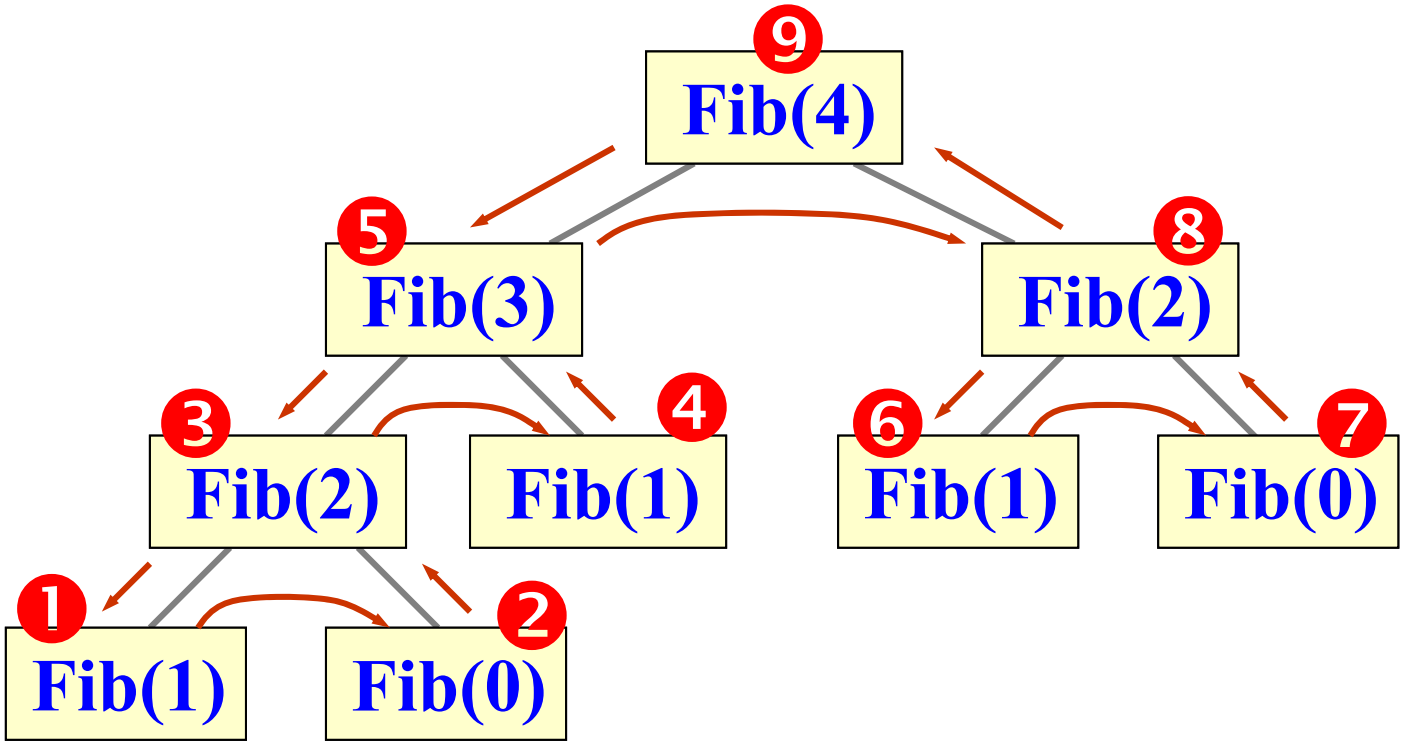
单向递归用迭代法实现

```
long FibIter(long n) {
    if (n<=1)  return n;
    long twoback=0, oneback=1, Current;
    for (int i=2; i<=n; i++) {
      Current=twoback+oneback;
      twoback=oneback;   oneback=Current;
     }
     return Current;
}
```

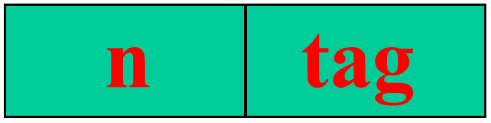


斐波那契数列的递归调用树

调用次数 NumCall(k)=2*Fib(k+1)-1



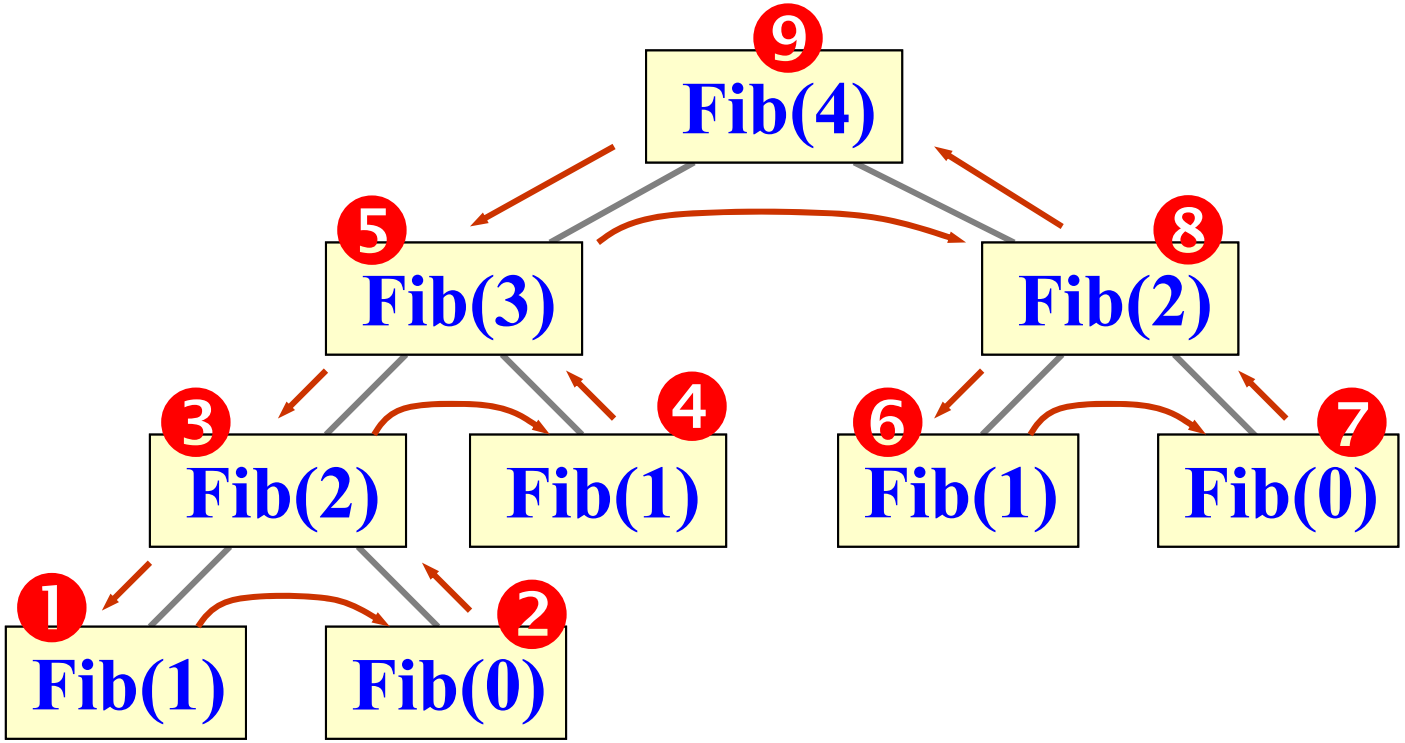
栈结点

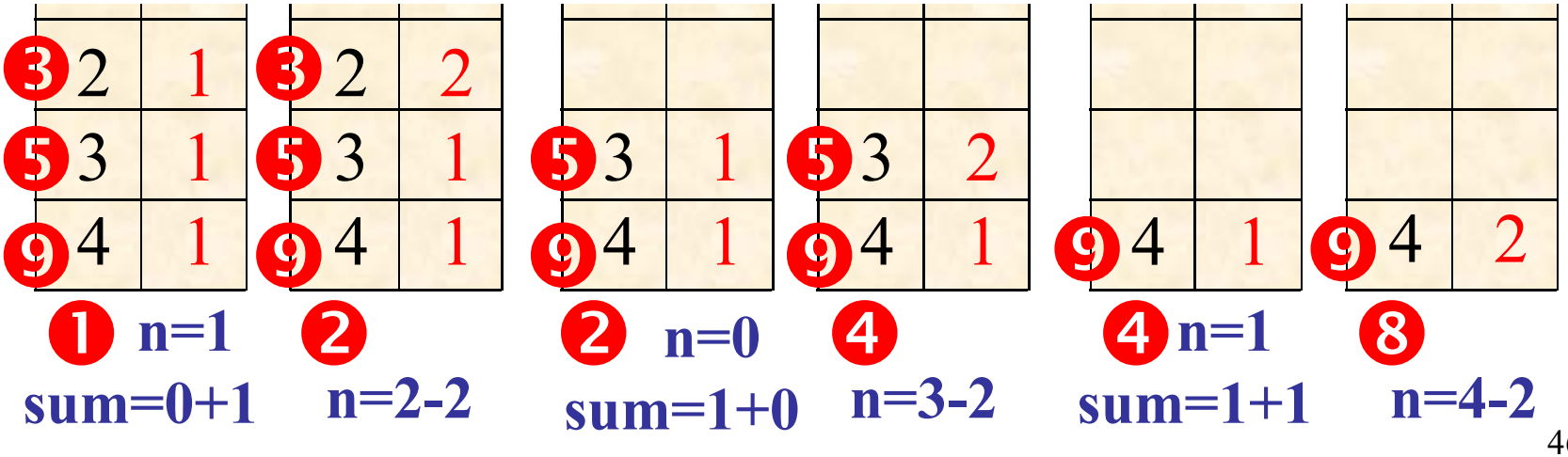


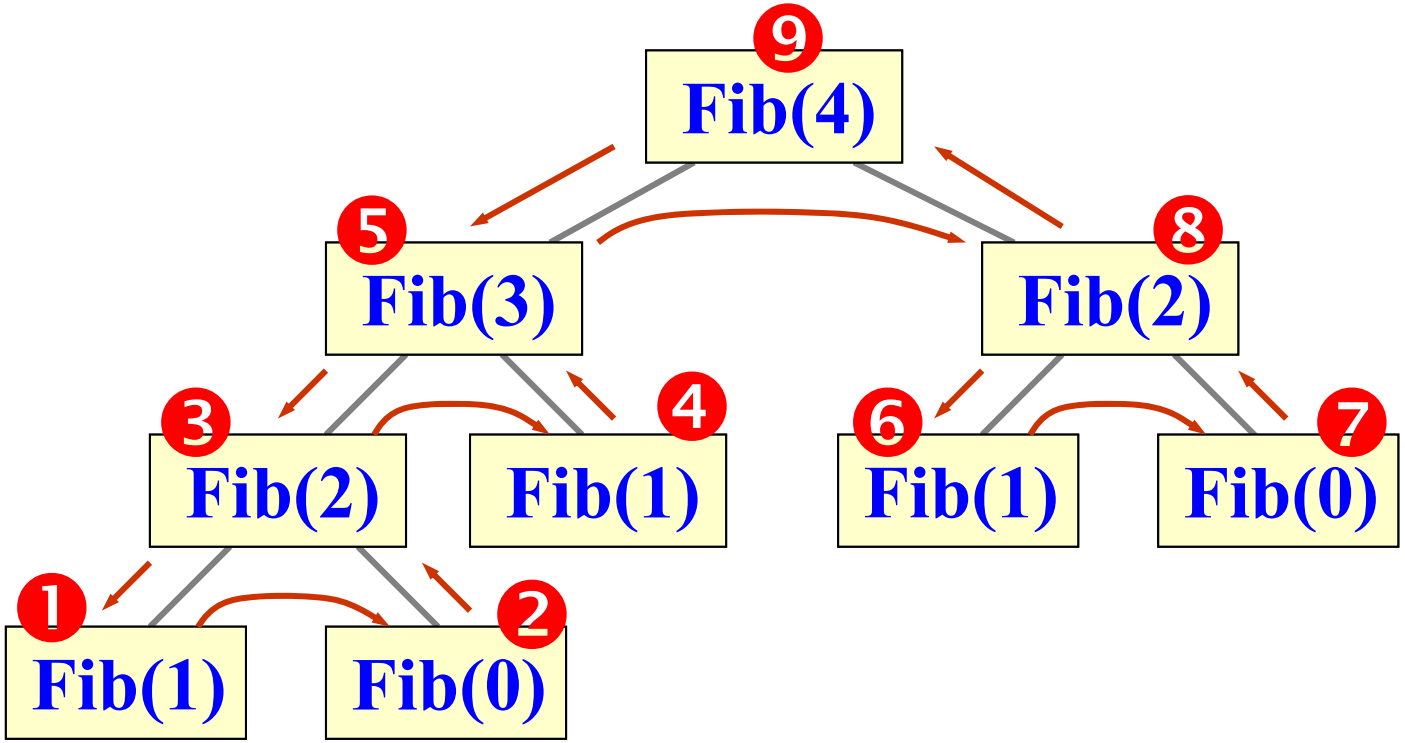

```
struct Node {
    long n;
    int tag;
};
```

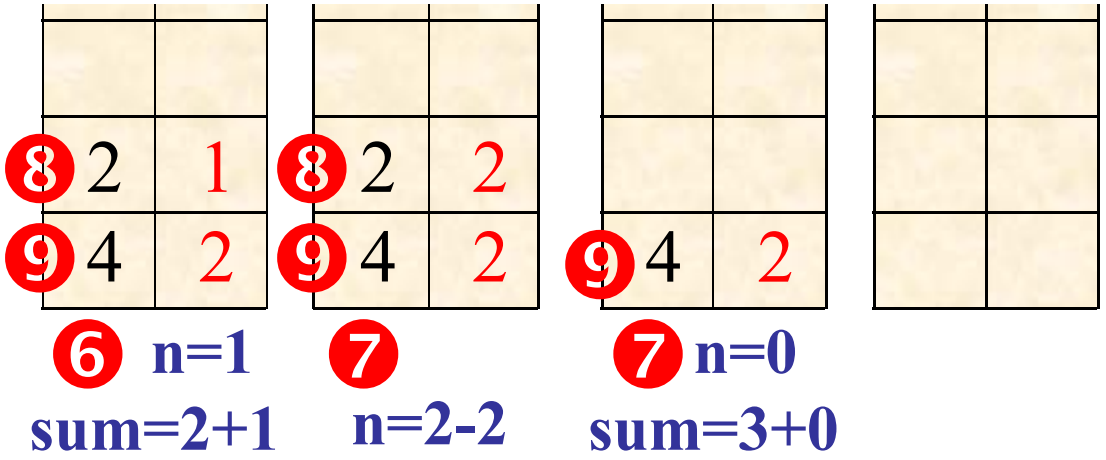
tag=1，向左递归；

tag=2，向右递归；


9
Fib(4)
5
Fib(3)
8
Fib(2)
3
Fib(2)
4
Fib(1)
6
Fib(1)
7
Fib(0)
1
Fib(1)
2
Fib(0)


3 2 1
5 3 1
9 4 1
1 n=1
sum=0+1
3 2 2
5 3 1
9 4 1
2 n=2-2
5 3 1
9 4 1
2 n=0
sum=1+0
5 3 2
9 4 1
4 n=3-2
9 4 1
4 n=1
sum=1+1
9 4 2
8 n=4-2


9
Fib(4)
5
Fib(3)
8
Fib(2)
3
Fib(2)
4
Fib(1)
6
Fib(1)
7
Fib(0)
1
Fib(1)
2
Fib(0)

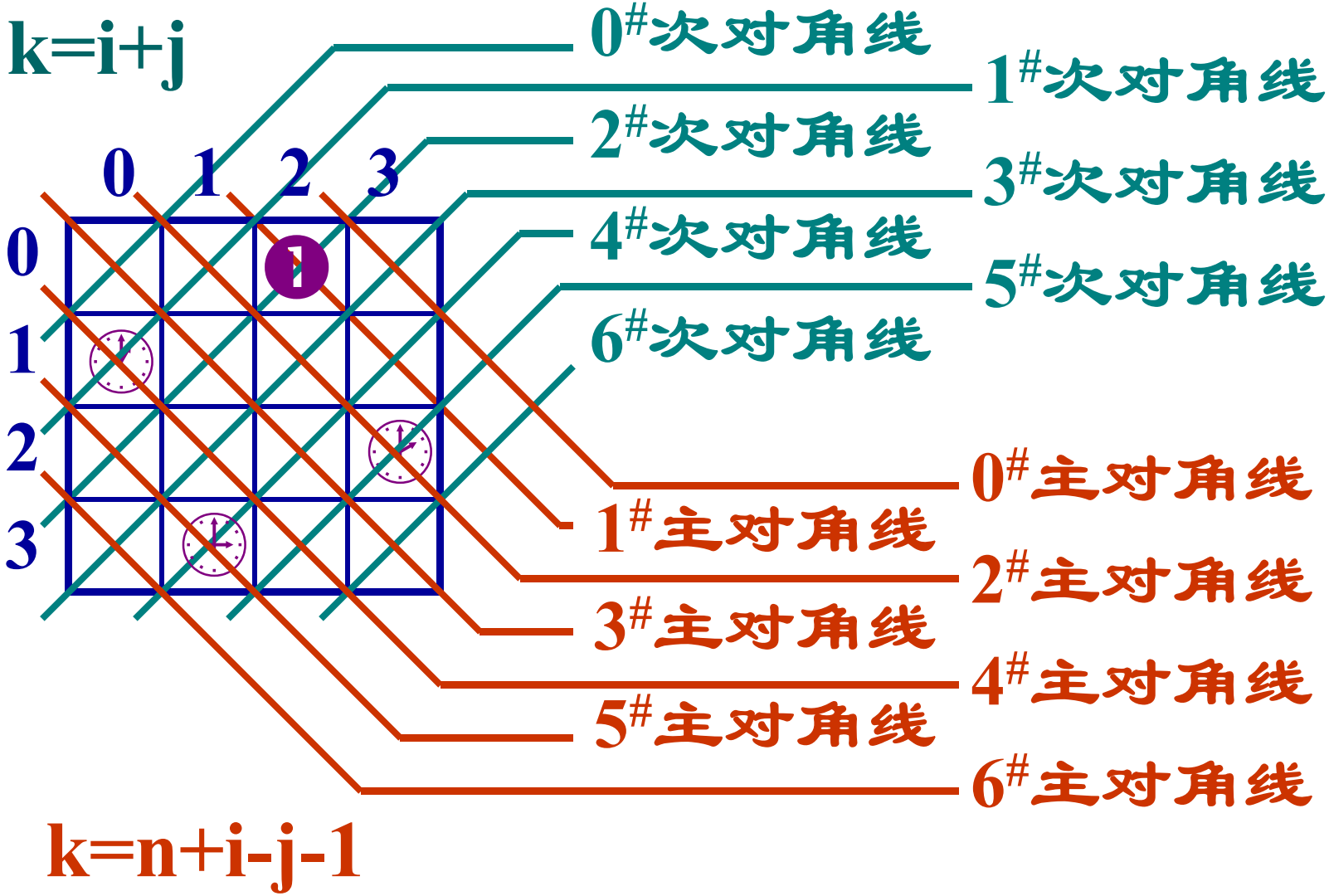

8 2 1
9 4 2
6 n=1
sum=2+1
8 2 2
9 4 2
7
n=2-2
9 4 2
7 n=0
sum=3+0

```
long Fibnacci(long n) {
    Stack <Node> S;  Node w;  long sum=0;
    //反复执行直到所有终端结点数据累加完
    do {
       while (n>1) {
          w.n=n;  w.tag=1;  S.Push(w);
          n--;
       } //向左递归到底，边走边进栈
       sum=sum+n;  //执行求和
```

```
        while (S.IsEmpty( )==false) {
            w=S.GetTop( );  S.Pop( );
            if (w.tag==1) {  //改为向右递归
              w.tag=2;  S.Push(w);
              n=w.n-2;  //F(n)右侧为F(n-2)
              break;
            }
        }
    } while (!S.IsEmpty( ));
    return sum;
}
```

*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0到n-1试探 (j=0, …, n-1) 。
- 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[2n-1]：md[k]标识第k条主对角线是否安放了皇后；
  - sd[2n-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=0; 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-1)  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=0;  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-1) {  //输出一个布局
               for (k=0; k<n; k++)
                  cout<<q[k]<<',';
               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
         }
      }
   }
```

随堂练习

例1：已知Ackerman函数定义如下：

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

(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。

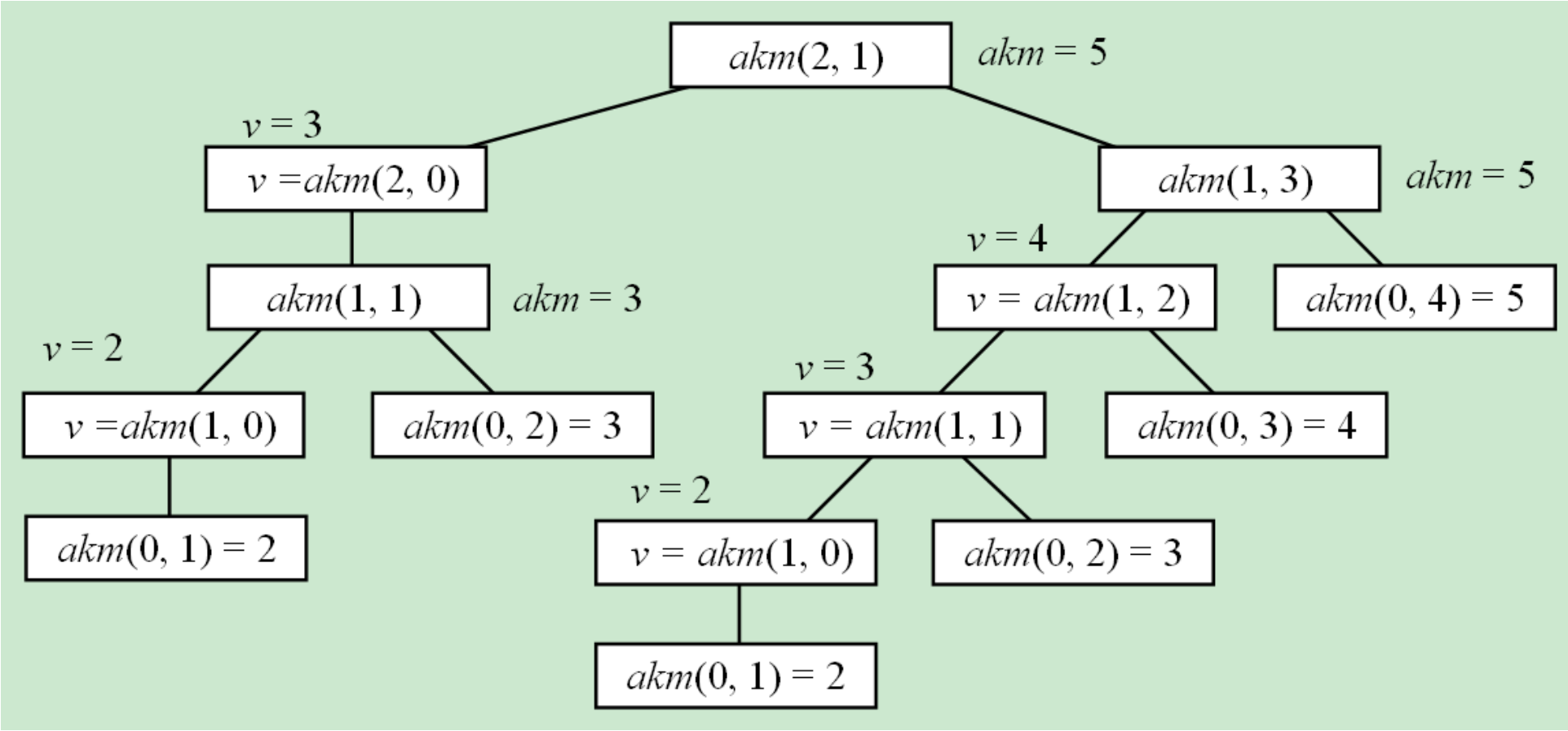
函数本身是递归定义的，所以可以用递归算法来解决：

```
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
}
```

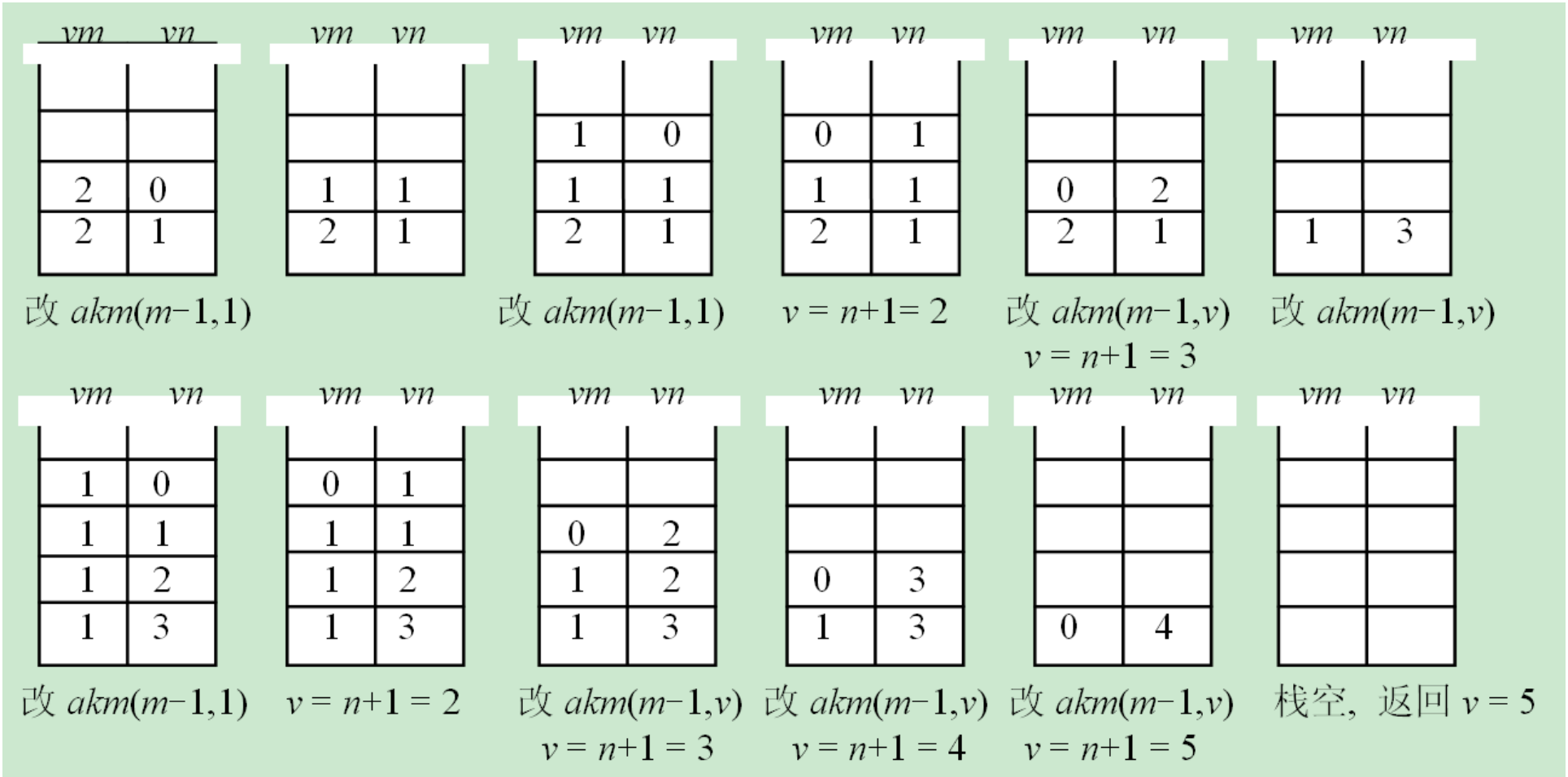
首先改写原来的递归算法，将递归语句从结构中独立出来：

```
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 );
}
```

akm(2, 1)的递归调用树



用栈记忆每次递归调用时的实参值，每个结点两个域{vm, vn}



```
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+1;
            //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;
}
```

关于递归转非递归

- 理论上而言，所有递归程序都可以用非递归程序来实现；这种理论的基础是递归程序的计算总能用一颗树型结构来表示。
- 递归计算从求树根节点的值开始，树根节点的值依赖一个或多个子节点的值，子节点的值又依赖下一级子节点的值，如此直至树的叶子节点。叶子节点的值能直接计算出来，也就是递归程序的出口。

递归转非递归的两种方法

1、可用迭代的算法替代(单向递归、尾递归)。

2、自己用堆栈模拟系统的运行时的栈，通过分析只保存必须保存的信息，从而用非递归算法替代递归算法。

- 画递归调用树
- 分析出栈过程
- 根据出栈过程遍历这颗树的叶子结点

例2：汉诺塔问题的非递归解法。

分析递归解法可知：盘子移动的输出是在两种情况下进行的：其一是当递归到n=1时进行输出；其二是返回到上一层函数调用具有n>=1时进行输出。因此，可以使用一个stack[]数组来实现汉诺塔问题的非递归解法。具体步骤如下：

(1) 将初值n, x, y, z送数组stack[]；

(2) n减1并交换y, z值后将n, x, y, z值送数组stack[]，直到n=1时为止；

(3) 当n=1时输出x值指向z值；

(4) 回退到前一个数组元素，如果此时这个数组元素下标值大于等于1，则输出x值指向z值（相当于递归函数hanoi返回到上一层函数调用时的输出）；当i<=0时程序结束。

(5) 将n减1，如果n>=1则交换x和y值（相当于执行递归函数hanoi中的第二个hanoi调用语句参数替换时的情况）；并且：

A. 如果此时n=1，则输出x值指向z值并继续退回到前一个数组元素处（相当于返回到递归函数hanoi的上一层）；如果i>=1，则输出x值指向z值；

B. 如果此时n>1，则转(2)处继续执行（相当于递归函数hanoi中的第二个hanoi调用语句继续调用的情况）。

5.3 广义表 (General Lists)

- 广义表的概念　n (≥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*)。 $Head(LS)=a_0$，$Tail(LS)=(a_1, a_2, \ldots, a_{n-1})$。

广义表的一些例子

***A*=( )**

空表，长度为0，无表头和表尾。

***B*=(6, 2)**

线性表（只包括原子），长度为2，表头为6，表尾为(2)。

***C*=(‘*a*’, (5, 3, ‘*x*’))**

长度为2，表头为‘*a*’，表尾为((5, 3, ‘*x*’))（子表）。

***D*=(*B*, *C*, *A*)**

长度为3，表头为*B*，表尾为(*C*, *A*)。

***E*=(*B*, *D*)**

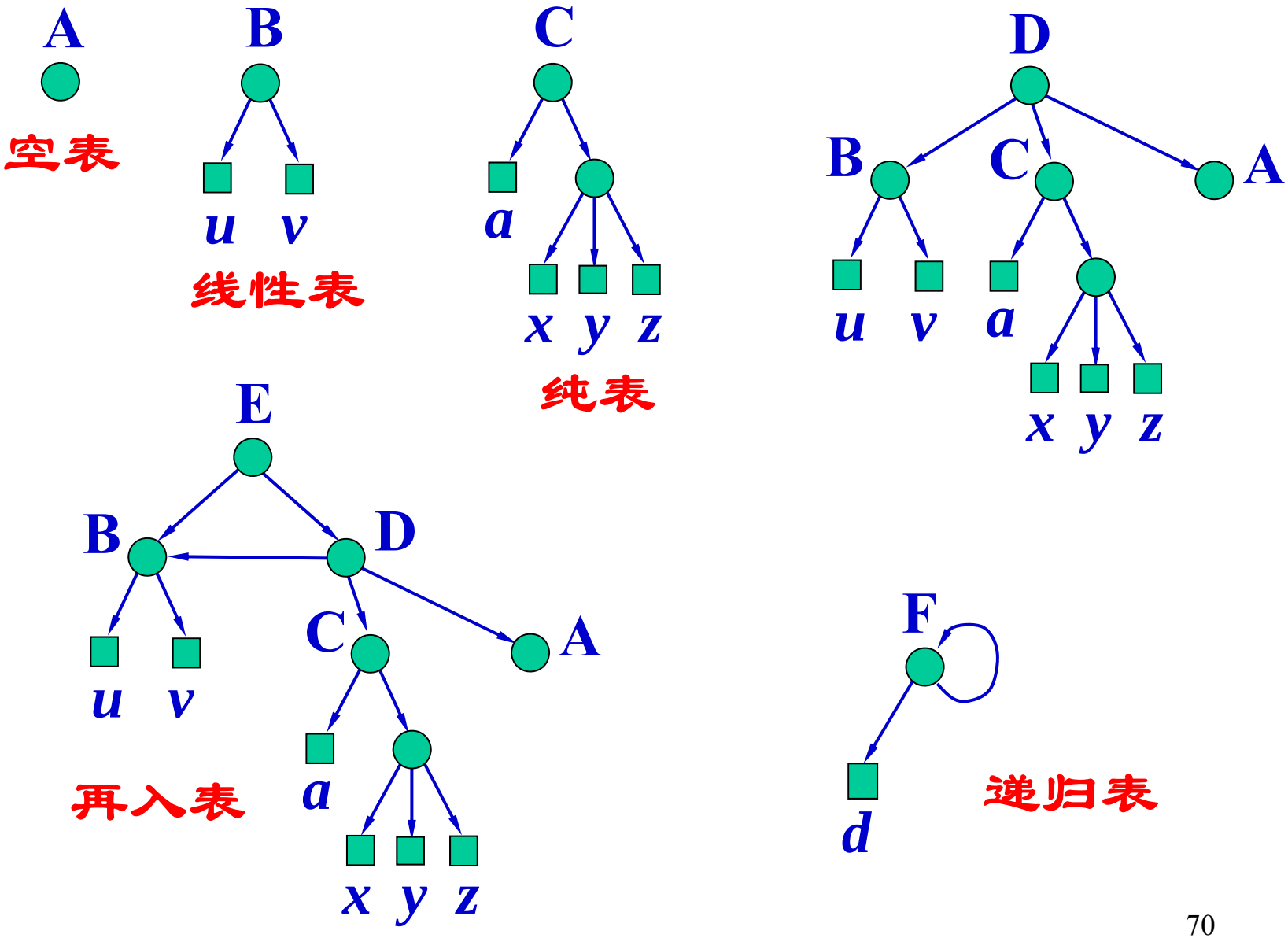
长度为2，表头为*B*，表尾为(*D*)。

***F*=(4, *F*)**

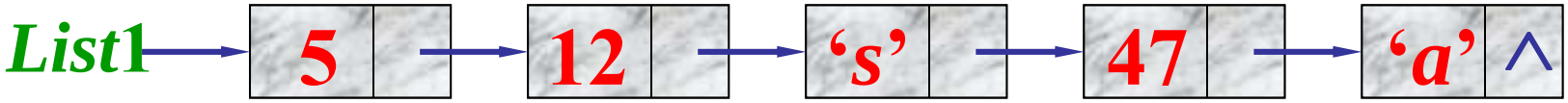
长度为2，表头为4，表尾为(*F*)。

广义表的特性

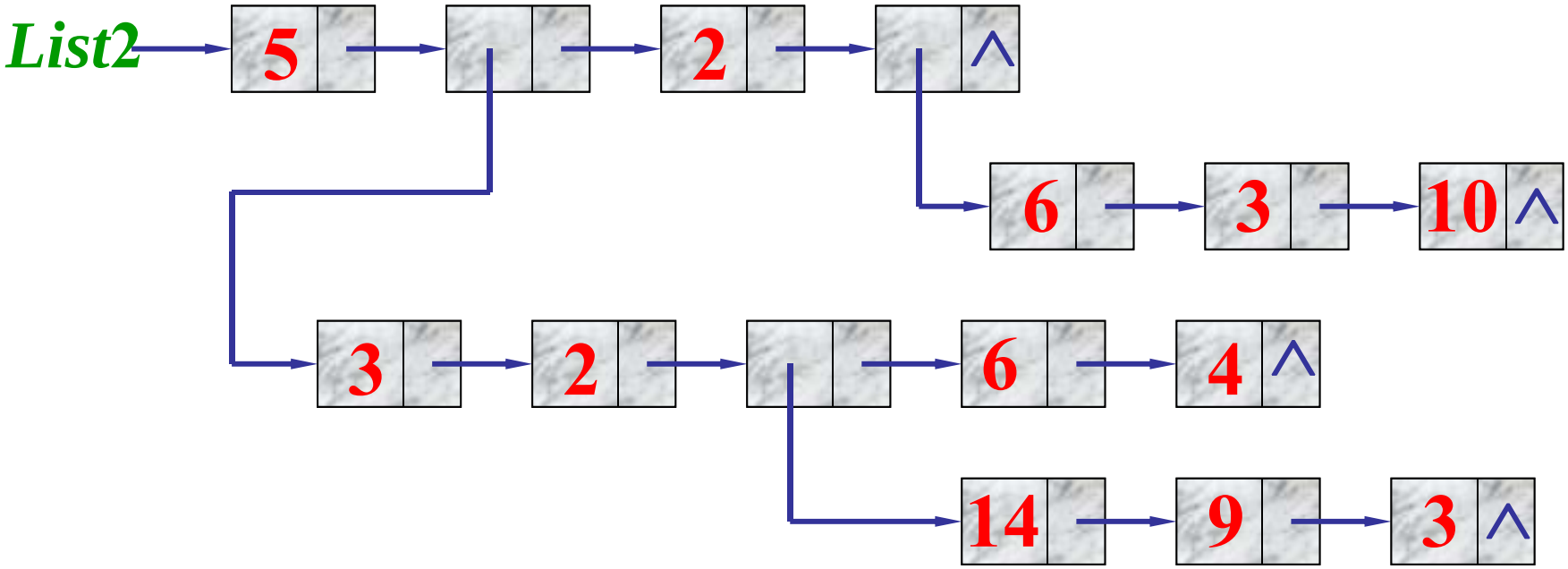
- 有次序性
  - 在广义表中，各表元素在表中以线性序列排列，每个元素至多一个直接前驱，一个直接后继，且次序不能交换。
- 有长度
  - 广义表中元素个数一定，不能是无限的，可以是空表。
- 有深度
  - 广义表是多层次结构，表元素可以是原表的子表，子表的表元素还可以是子表等等。
- 可递归
  - 广义表可以是自己的子表（如***F***）。
- 可共享
  - 广义表可以为其它广义表共享—共享表（如***B***）。


A
空表
B
u v
线性表
C
a
x y z
纯表
D
B C A
u v a
x y z
E
B D
u v
C A
a
x y z
再入表
F
d
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

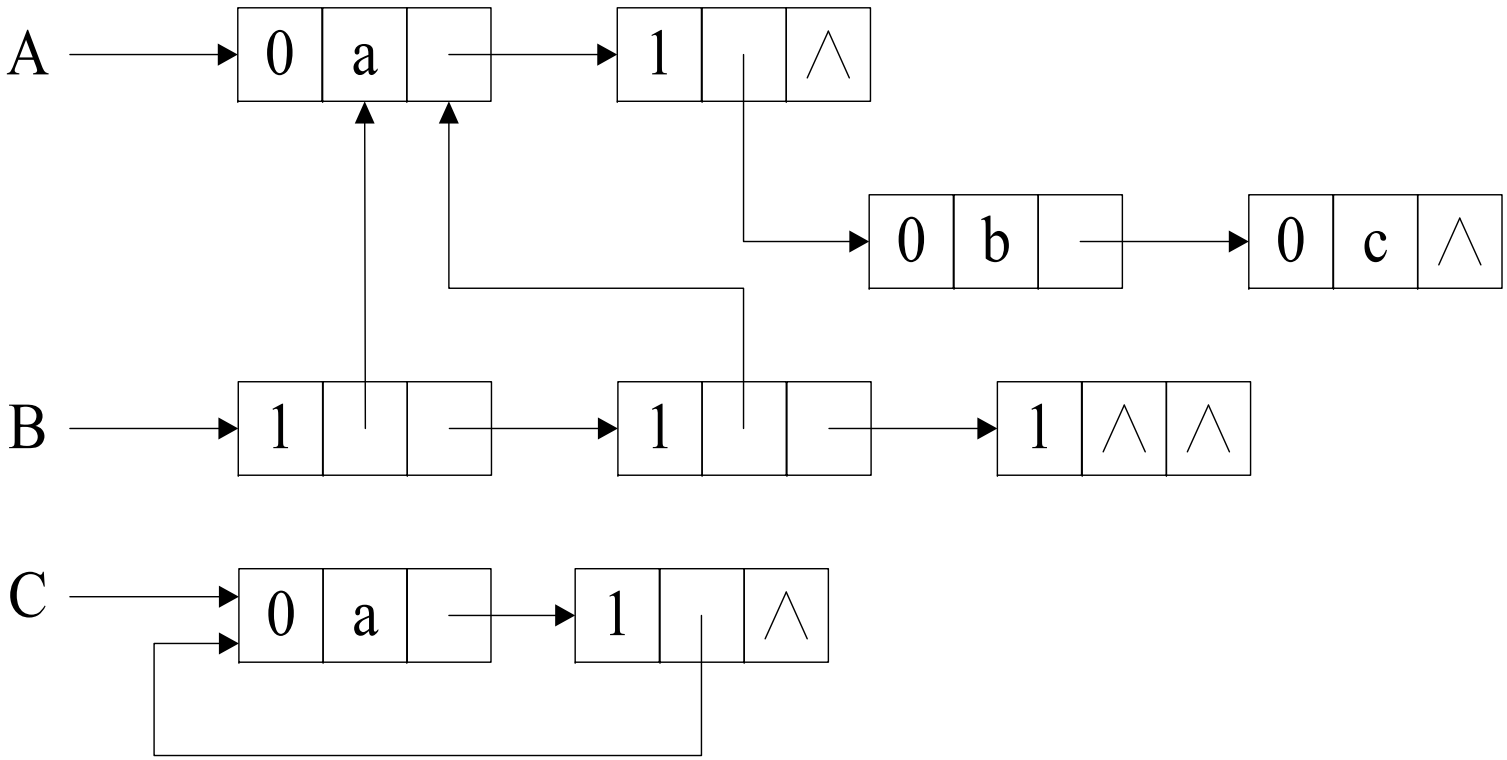


表中套表情形下的广义表链表表示

广义表结点定义(I)

tag	dlink/data	link

- 标志域tag——结点类型，tag=0，原子；tag=1，子表。
- 信息域info——tag=0时，存放数据值(data)；tag=1时，存放指向子表表头的指针(dlink)。
- 尾指针域link——指向同一层下一个结点。



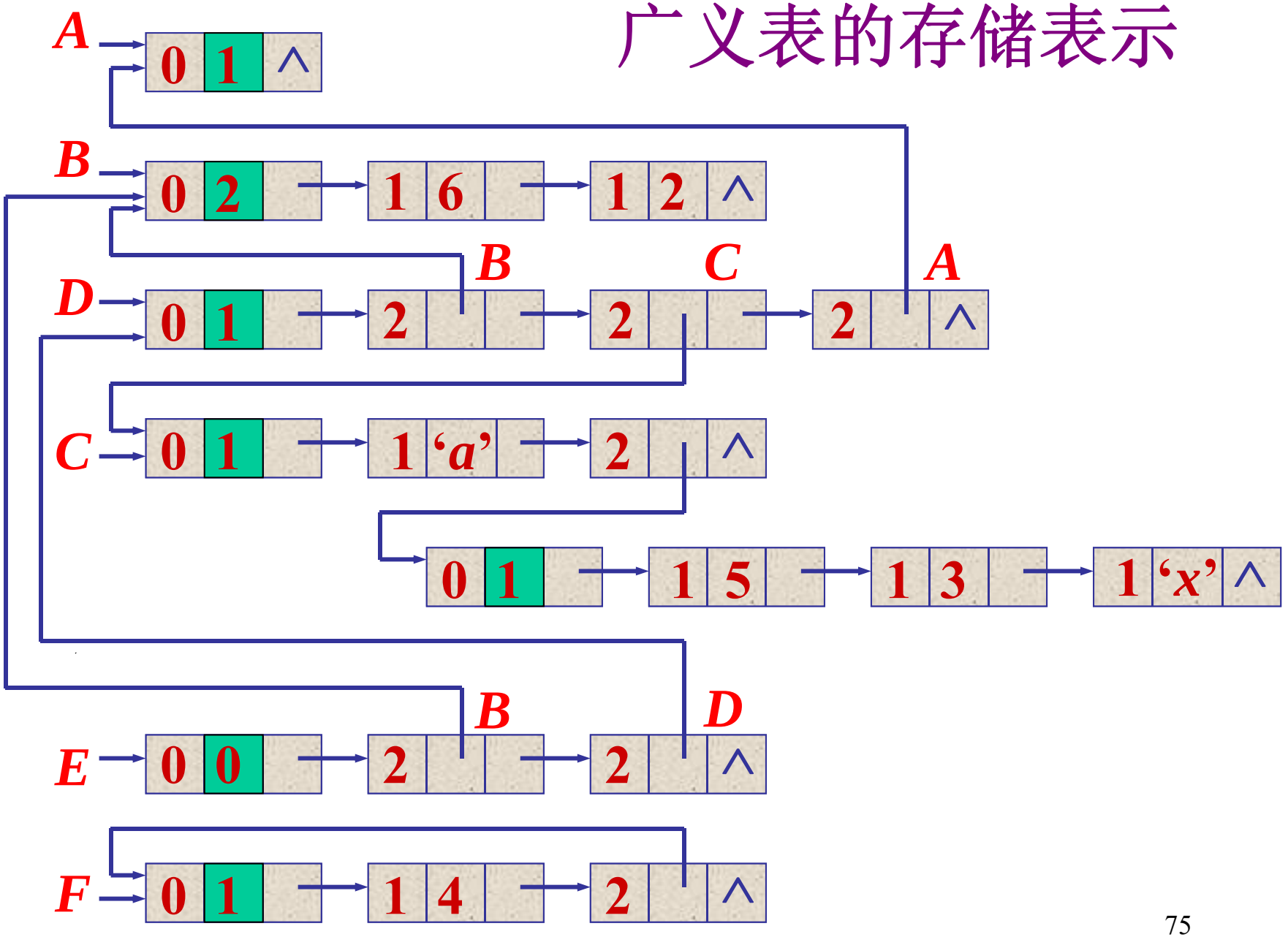
用链接结构表示广义表

广义表结点定义(II)

utype	info	tlink

- 标志域utype——结点类型，utype=0，表头；utype=1，原子；utype=2，子表。
- 信息域info——utype=0时，存放引用计数(ref)；utype=1时，存放数据值(value)；utype=2时，存放指向子表表头的指针(hlink)。
- 尾指针域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



- 存储表示的特点
  - 广义表中的所有表（包括空表），不论哪一层的子表，都带有一个表头结点。
    - 优点——便于操作。
  - 表中结点的层次分明
    - 所有位于同一层的表元素，在其存储表示中也在同一层。
  - 最高一层的表结点个数（除表头结点外）即为表长度。

程序5-1 广义表的类定义(I)

```
#define HEAD 0  //表头结点类型
#define INTEGER 1  //整数类型
#define CHAR 2  //字符类型
#define LIST 3  //子表类型
class GenList;  //广义表的前向引用声明
class GenListNode {  //广义表结点的类定义
    friend class GenList;
    private:
        int tag;  //=0/1/2/3，表示HEAD/INTEGER/CHAR/LIST
        GenListNode *link;  //指向同一层下一结点的指针
        union {  //联合
            int ref;  //tag=HEAD，表头结点类型，用于存放引用计数
            int integer_data;  //tag=HEAD，存放整数值
            char char_data;  //tag=CHAR，存放字符值
            GenListNode *dlink;  //tag=LIST，存放指向子表的指针
        } data;
```

```
    public:
        GenListNode & Data(GenListNode *elem);
        //返回表元素elem的值
        int NodeTypw(GenListNode *elem)
        { return elem->tag; }
        //返回表元素elem的元素值的数据类型
        void SetData(GenListNode *elem, GenListNode &x);
        //将表元素elem中的值修改为x
};
```

```
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( );
        //返回广义表第一个元素的值，即表头元素
        GenListNode & Tail( );
        //返回广义表除第一个元素外其它元素组成的表，即表尾
        GenListNode * First( );  //返回广义表的第一个元素
        GenListNode * Next(GenListNode *elem);
        //返回表元素elem的直接后继元素
        void Push(GenListNode &x);
        //将一个包含值x的元素加入到广义表的最前面
```

```
GenList & Addon(GenList &list, GenListNode &x);
//返回一个以x为头，list为尾的新表
void SetHead(GenListNode &x);
//将广义表的头元素重置为x
void SetNext(GenListNode *elem1, GenListNOde *elem2);
//将elem2插到表中元素elem1后
void SetTail(GenList *list);  //将list定义为广义表的尾
GenListNode * Copy(GenListNode *ls);  //广义表的复制
int Equal(GenListNode *s, GenListNode *t);
//判断两个广义表是否相等
int Depth(GenListNode *s, GenListNode *t);
//计算一个非递归表的深度
void Remove(GenListNode *ls);
//将以ls为表头结点的广义表结构释放
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
};
```

广义表的类定义(II)

```
template <class Type> class GenList;
template <class Type> class GenListNode {  //广义表结点类定义
friend class Genlist;
  private:
    int utype;  //=0/1/2
    GenListNode <Type> *tlink;  //指向同一层下一结点的指针
    union {  //联合
       int ref;  //utype=0，存放引用计数
       Type value;  //utype=1，存放数值
       GenListNode <Type> *hlink;  //utype=2，存放指向子表的指针
     } info;
  public:
     GenListNode( ) : utype(0), tlink(NULL), info.ref(0) { }
     GenListNode(GenListNode <Type> &RL) {
        utype=RL.utype;  tlink=Rl.tlink;  info=RL.info;  }
};
```

```
template <class Type> class GenList {  //广义表类定义
 private:
   GenListNode <Type> *first;  //广义表头指针
   GenListNode <Type> * Copy(GenListNode <Type> *ls);
   //复制一个ls指示的无共享非递归表
   int Depth(GenListNode <Type> *ls);
   //计算由ls指示的非递归表的深度
   int Length(GenListNode <Type> *ls);
   //计算由ls指示的非递归表的长度
   bool Equal(GenListNode <Type> *s, GenListNode <Type> *t);
   //比较以s和t为表头的两个表是否相等
   void Remove(GenListNode <Type> *ls);
   //释放以ls为表头结点的广义表
   void CreateList(istream &in, GenListNode <Type> *&ls,
          SeqList <Type> &L1, SeqList <GenListNode <Type> *> &L2);
   //从输入流对象输入广义表的字符串描述
   //建立一个带附加头结点的广义表结构
```

```
public:
  Genlist( );  //构造函数
  ~GenList( );  //析构函数
  bool Head(info &x);  //返回表头元素
  bool Tail(GenList <Type> &lt);  //返回表尾
  GenListNode <Type> * First( );  //返回第一个元素
  GenListNode <Type> * Next(GenListNode <Type> *elem);
  //返回表元素elem的直接后继元素
  void Copy(const GenList <Type> &R);
  //返回一个以x为头，list为尾的新广义表
  int Length( );  //计算广义表的长度
  int Depth( );  //计算广义表的深度
  friend istream & operator >> (istream &in,
                                GenList <type> &L);
};
```

```
public:
  Genlist( );  //构造函数
  ~GenList( );  //析构函数
  bool Head(info &x);  //返回表头元素
  bool Tail(GenList <Type> &lt);  //返回表尾
  GenListNode <Type> * First( );  //返回第一个元素
  GenListNode <Type> * Next(GenListNode <Type> *elem);
  //返回表元素elem的直接后继元素
  void Copy(const GenList <Type> &R);
  //返回一个以x为头，list为尾的新广义表
  int Length( );  //计算广义表的长度
  int Depth( );  //计算广义表的深度
  friend istream & operator >> (istream &in,
                                GenList <type> &L);
};
```

广义表的访问算法

广义表结点类的存取成员函数

```
template <class Type> GenListNode & GenListNode  <Type> ::
Info( ) {  //返回表元素的值
    GenListNode *pitem=new GenListNode;
    pitem->utype=utype;
    pitem->info.value=info.value;
    return *pitem;
}
template <class Type> void  GenListNode <Type> ::
SetInfo(GenListNode <Type> *elem,
                GenListNode <Type> &x) {  //修改表元素的值为x
    elem->utype=x.utype;
    elem->info.value=x.info.value;
}
```

广义表类的构造和访问成员函数

```
template <class Type> Genlist <Type> ::
GenList( ) {  //构造函数
    GenListNode *first=new GenListNode;
    first->utype=0;
    first->info.ref=1;
    first->tlink=NULL;
    assert(first!=NULL);
}
```

```
template <class Type> bool GenList <Type> ::
Head(GenListNode <Type> &x) {
//若广义表非空，则返回其第一个元素的值
//否则函数没有定义
    if (first->tlink==NULL)  return false;
    else {  //非空表
       x.utype=frist->tlink->utype;
       x.info=frist->tlink->info;
       return true;  //返回类型及值
    }
}
```

```
template <class Type> bool GenList <Type> ::
Tail(GenList <Type> &lt) {
//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 false;
    else {
         lt.first->utype=0;
         lt.first->info.ref=0;
         lt.first->tlink=Copy(first->tlink->tlink);
    }
    return true;
}
```

```
template <class Type> GenListNode * GenList <Type> ::
First( ) {
//返回广义表的第一个元素
//若表空，则返回NULL
   if (first->tlink==NULL)  return NULL;
   else return first->tlink;
}

template <class Type> GenListNode * GenList <Type> ::
Next(GenListNode <Type> *elem) {
//返回表元素elem的直接后继元素
   if (elem->tlink==NULL)  return NULL;
   else return elem->tlink;
}
```

广义表的递归算法

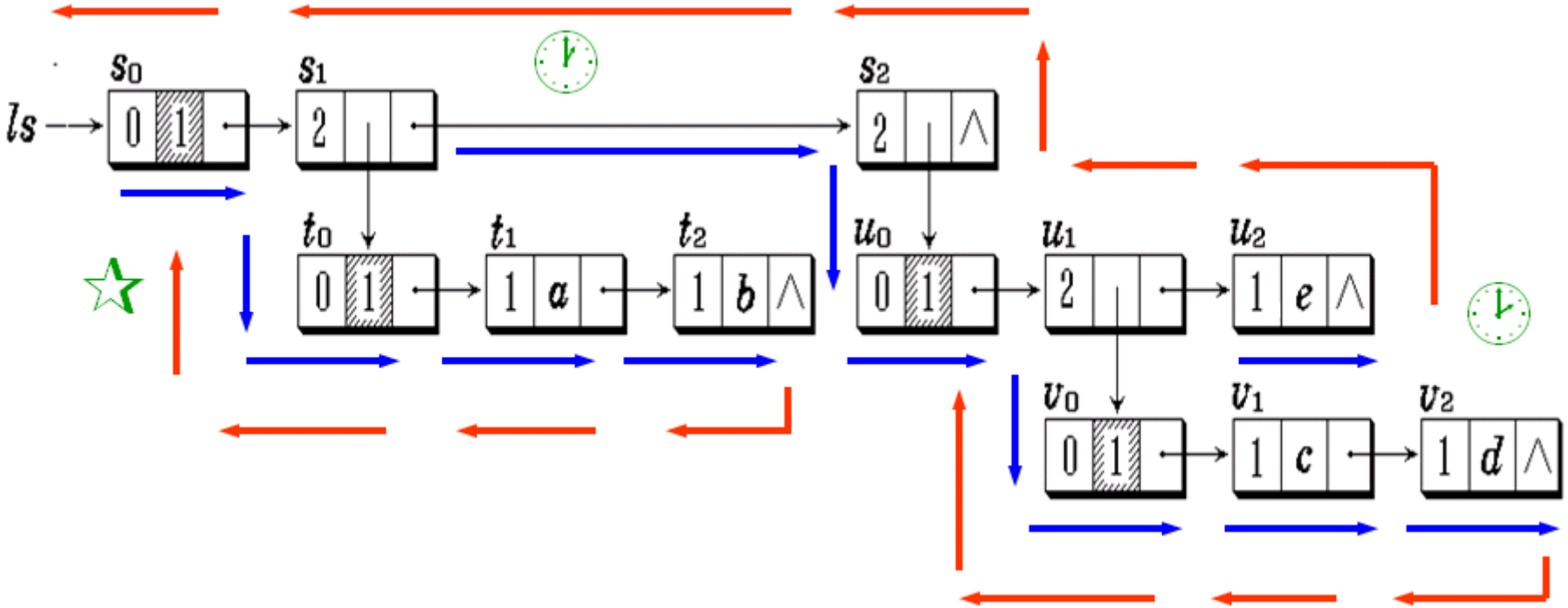
广义表的复制算法

- 任何一个非空的广义表均可分为表头、表尾两个部分，则一对确定的表头和表尾可唯一确定一个广义表。
- 复制一个广义表时，需要分别复制其表头和表尾，然后再合成，其前提是广义表不是共享表或递归表。

程序5-2 广义表的复制

```
GenListNode * GenList :: Copy(GenListNode *ls)
{  //复制一个由ls指示的无共享子表的非递归表
     GenListNode *q=NULL;
     if (ls!=NULL)
     {
          q=new GenListNode;  //创建一个新结点
          q->tag=ls->tag;  //复制类型标记
          switch (ls->tag)
          {  //根据结点类型tag复制值域信息
               case HEAD :  //表头结点
                         q->data.ref=ls->data.ref;  break;
               case INTEGER :  //整型原子结点
                         q->data.integer_data=ls->data.integer_data;
                         break;
               case CHAR :  //字符型原子结点
                         q->data.char_data=ls->data.char_data;  break;
               case LIST :  //子表结点
                         q->data.dlink=Copy(ls->data.dlink);  break;
          }
          q->link=Copy(ls->link);  //复制同一层下一结点为头的表
     }
     return q;
}
```

```
template <class Type> void GenList <Type> ::
Copy(const GenList <Type> &R) {  //公有函数
  first=Copy(R.first);
}
template <class Type> GenListNode * GenList <Type> ::
Copy(GenListNode <Type> *ls) {  //私有函数
   GenListNode <Type> *q=NULL;
   if (ls!=NULL) {
     q=new GenListNode <Type>;
     q->utype=ls->utype;
     switch (ls->utype) {
        case 0: q->info.ref=ls->info.ref;  break;
        case 1: q->info.value=ls->info.value;  break;
        case 2: q->info.hlink=Copy(ls->info.hlink);  break;  }
     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
c
d
e

求广义表的长度

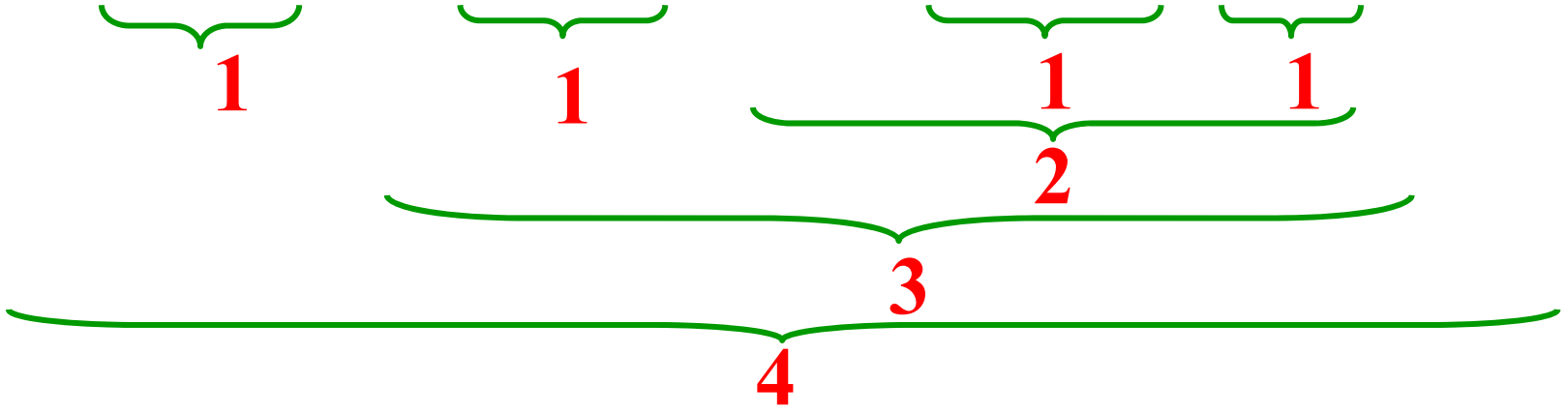
```
template <class Type> int GenList <Type> ::
Length( ) {  //公有函数
    return Length(first->tlink);
}
template <class Type> int GenList <Type> ::
Length(GenListNode <Type> *ls) {  //私有函数
    if (ls!=NULL)  return 1+Length(ls->tlink);
    else return 0;
}
```

求广义表的深度

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

例如，对于广义表

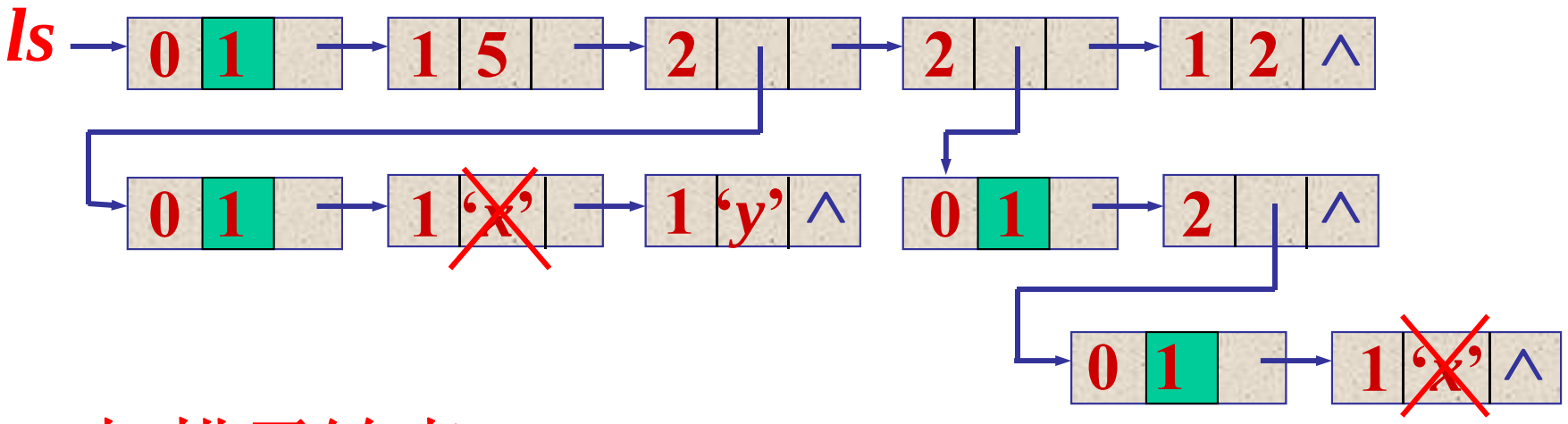
E(B(a, b), D(B(a, b), C(u, (x, y, z)), A ( )))



```
template <class Type> int GenList <Type> :: Depth( ) {
    return Depth(first);
}

template <class Type> int GenList <Type> ::
Depth(GenListNode <Type> *ls) {
    if (ls->tlink==NULL)  return 1;  //空表
    GenListNode <Type> *temp=ls->tlink;  int m=0, n;
    while (temp!=NULL) {  //在表顶层横扫
        if (temp->utype==2) {  //结点为表结点
            n=Depth(temp->info.hlink);
            if (m<n)  m=n;  //m记最大深度  }
        temp=temp->tlink;
    }
    return m+1;
}
```

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。

```
template <class Type> void GenList <Type> ::
DelValue(GenListNode <Type> *ls, Type x)
{  //在广义表中删除所有含x的结点
   if (ls->tlink!=NULL) {  //非空表
      GenListNode <Type> *p=ls->tlink;
      while (p!=NULL &&  //横扫链表
                  ((p->utype==1 && p->info.value==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==2)  //在子表中删除
            DelValue(p->info.hlink, x);
         DelValue(p, x);  //在后续链表中删除  }
   }
}
```

- 对于共享表
  - 如果想删除某一个子表，要看它是否为几个表共享。
  - 如果一个表元素有多个地方使用，删除操作可能会造成其它地方使用出错。
  - 在表头结点中设置一个引用计数，当要做删除时先把该引用计数减**1**，只有引用计数减到**0**时才能执行结点的真正释放。

```
template <class Type> GenList <Type> :: ~GenList( ) {
//析构函数，每个表头结点都有一个引用计数
   Remove(first);
}

template <class Type> void GenList <Type> ::
Remove(GenListNode <type> *ls) {
   ls->info.ref--;  //引用计数减一
   if (ls->info.ref<=0) {  //引用计数减至0才能删除
      GenListNode <Type> *q;
      while (ls->tlink!=NULL) {  //横扫链表
         q=ls->tlink;
         if (q->utype==2) {  //遇到子表
            Remove(q->info.hlink);  //递归删除子表
            if (q->info.hlink->info.ref<=0)
               delete q->info.hlink;  }
          ls->tlink=q->tlink;  delete q;  }
   }
}
```

判断两个广义表是否相等的算法

- 判断两个广义表是否相等
  - 不但两个广义表具有相同的结构，而且对应的数据成员具有相等的值。
    - 如果两个广义表都是空表，则相等。
    - 如果两个广义表的对应结点都是原子结点，对应项的值相等，再递归比较同一层后面的表元素。
    - 如果两个广义表中对应项是子表结点，则递归比较相应的子表。

程序5-3 判别两个广义表是否相同

```
int GenList :: Equal(GenListNode *s, GenListNode *t)
{ //判断由s与t指示的两个广义表是否相等
    int x;
    if (s->link==NULL && t->link==NULL)  return 1;
    //表s与表t都是空表或者所有结点都比较完
    if (s->link!=NULL && t->link!=NULL &&
                                    s->link->tag==t->link->tag)
    { //两个广义表非空且结点类型标志相同
      if (s->link->tag==INTEGER)  //整型原子结点
        if (s->link->data.integer_data==
                                t->link->data.integer_data)
      //比较对应整数值
            x=1;
        else x=0;
```

```
        else if (s->link->tag==CHAR)  //字符型原子结点
                if (s->link->data.char_data==
                                t->link->data.char_data)
                //比较对应字符值
                    x=1;
                else x=0;
            else x=Equal(s->link->data.dlink,
                                        t->link->data.dlink);
                //子表结点，递归比较其子表
        if (x)  return Equal(s->link, t->link);
        //当前结点相等，递归比较同一层的下一结点
        //若当前结点不等，则不再递归比较
    }
    return 0;
}
```

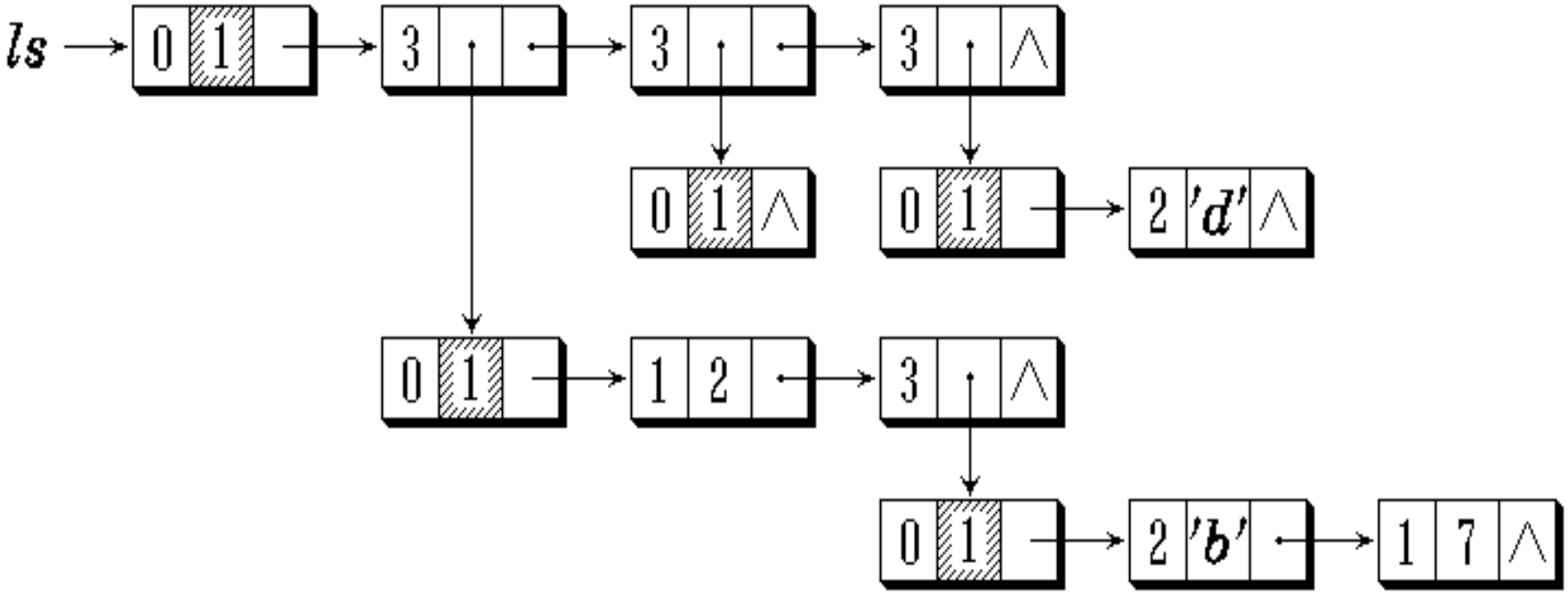
```
template <class Type> bool GenList <Type> ::
Equal(GenListNode <Type> *s, GenListNode <Type> *t) {
    int x;
    if (s->tlink==NULL && t->tlink==NULL)
        return true;  //表s与表t都是空表或者所有结点都比较完
    if (s->tlink!=NULL && t->tlink!=NULL &&
            s->tlink->utype==t->tlink->utype)
    {  //两表都非空且结点标志相同
        if (s->tlink->utype==1)  //原子结点，比较对应数据
            x=(s->tlink->info.value==t->tlink->info.value) ? 1 : 0;
        else if (s->tlink->utype==2)  //原子结点，比较对应数据
            x=Equal(s->tlink->info.hlink, t->tlink->info.hlink);
        if (x==1)  return Equal(s->tlink, t->tlink);
    }
    return false;
}
```

从字符串*s*建立广义表的链表表示*ls*

- 对于从输入流对象输入的字符，检测其内容。
  - 大写字母表示的表名，首先检查其是否存在，如是则说明是共享表，将相应附加头结点的引用计数加一；如果不是，则保存该表名并建立相应广义表。
  - 表名后一定是左括号‘(’，不是则输入错，是则递归建立广义表结构。
  - 如果遇到小写字母表示的原子，则建立原子元素结点；如果遇到右括号‘)’，子表链收尾并退出递归。
  - 注意，空表情形括号里应加入一个非英文字母，如‘#’，不能一个字符也没有。

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

得到的广义表链表结构



“*D*(*B*(*a*, *b*), *C*(*u*, (*x*, *y*, *z*)), *A*(#));”

```
template <class Type> void GenList <Type> ::
CreateList(istream in, GenListNode <Type> *&ls,
        SeqList <Type> &L1, SeqList <GenListNode <Type> *> L2) {
//在表L1存储大写字母的表名
//在表L2中存储表名对应子表结点的地址
    Type chr;
    in>>chr;
    if (isalpha(chr) && isupper(chr)||che==‘(’) {
       ls=new GenListNode <Type>;  //建子表结点
       ls->utype=2;
       if (isalpha(chr) && isupper(chr)) {  //表名处理
          int n=L1.Length( );
          int m=L1.Search(chr);
          if (m!=0) {  //该表已建立
             GenListNode <Type> *p=L2.Locate(m);  //查子表地址
             p->ref++;  //表引用计数加1  }
          else {  L1.Insert(n, chr);  L2.Insert(n, ls);  }  //保存表名及地址
```

```
            in>>chr;
            if (chr!='(')  eixt(1);  //表名后必跟'('
        }
        ls->info.hlink=new GenListNode <Type>;  //建附加头结点
        ls->info.hlink->utype=0;  ls->info.hlink->ref=1;
        CreateList(in, ls->info.hlink->tlink, L1, L2);  //递归建子表
        CreateList(in, ls, L1, L2);  //递归建后继表
    }
    else if (isalpha(chr) && islower(chr)) {  //建原子结点
         ls=new GenListNode <Type>;  ls->utype=1;
         ls->info.value=chr;
         CreateList(in, ls, L1, L2);
     }
     else if (chr==',')  //建后继结点
          CreateList(in, ls->tlink, L1, L2);
     else if (chr==')')  ls->tlink=NULL;  //右括号，链收尾
     else if (chr=='#')  ls==NULL;  //空表，链收尾
}
```

随堂练习

例1：若广义表满足Head(A)=Tail(A)，则A为(())，其长度和深度为_____和_____。

例2：广义表具有可共享性，因此在遍历一个广义表时必须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。一旦作了mark，以后就不再访问该结点。

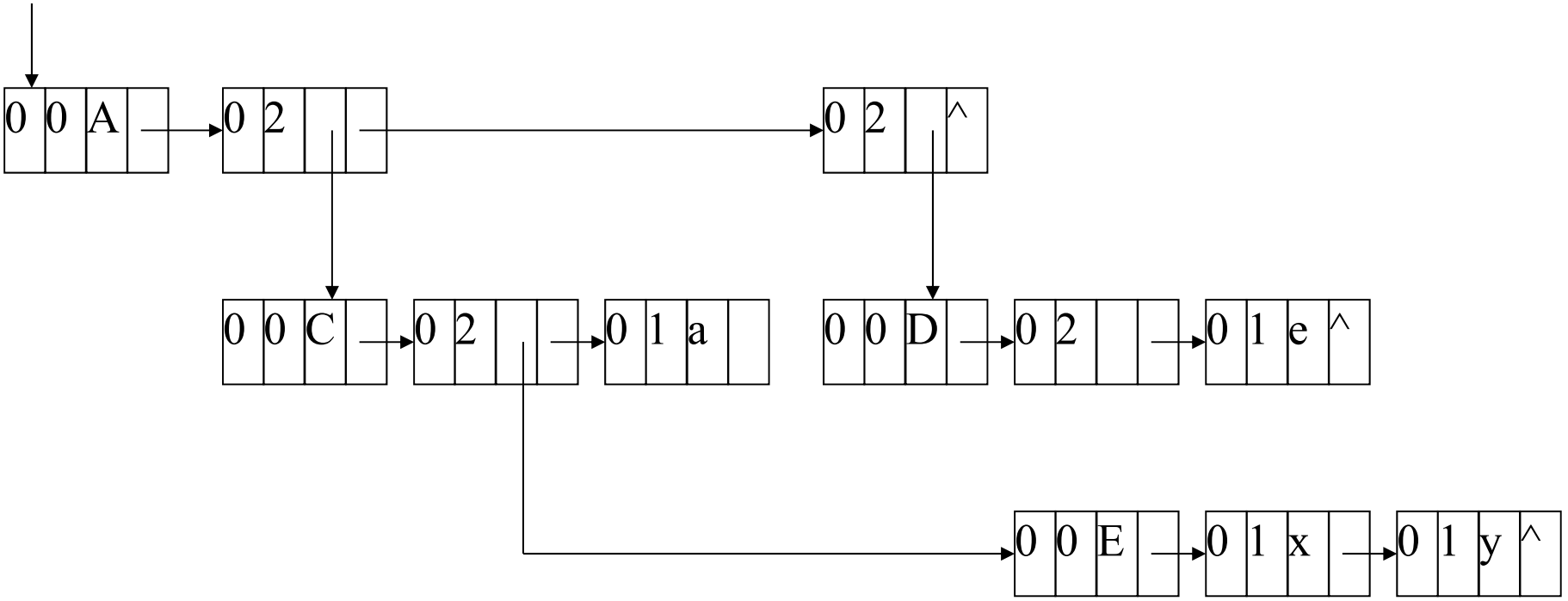
(1) 定义该广义表的类结构；

(2) 采用递归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。

例1：若广义表满足Head(A)=Tail(A)，则A为(())，其长度和深度为 1 和 2 。

例2：



三种结点类型：表头结点，原子结点，子表结点。

简化约定：
a. 原子结点数据为字符型，并且不用大写字母
b. 表名用大写字母表示，存于表头结点

```
class GenList;
class GenListNode
{
 friend class GenList;
 private:
    int mark, utype;
     //utype=0/1/2, mark是访问标记
    GenListNode* tlink;  //指向同层下一结点
    union{
       char listname;  //utype=0,表头结点存表名
       char atom;  //utype=1,存原子结点数据
       GenListNode* hlink; //指向子表
    }value;
```

```
public:
   GenListNode(int tp,char info)
    : mark(0),utype(tp),tlink(NULL) {
        if (utype==0) value.listname=info;
            else value.atom=info;
    } //表头或原子结点构造函数
   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;
    }
};
```

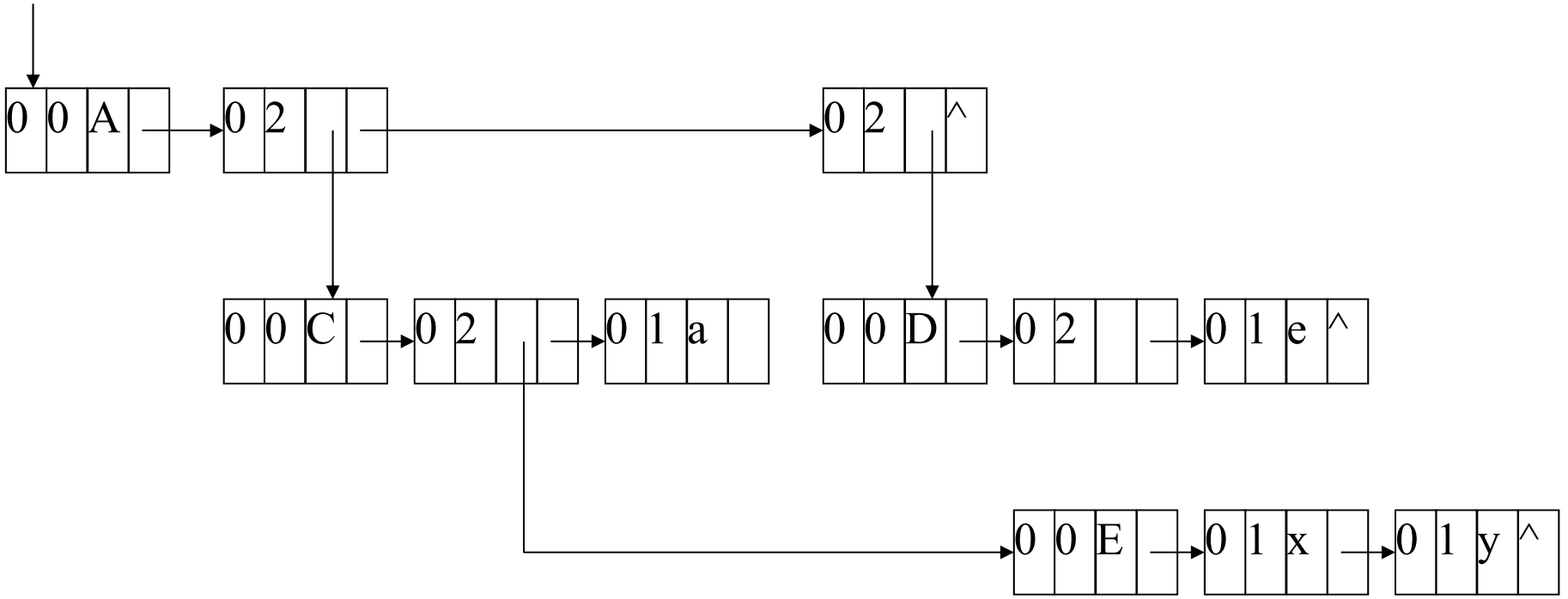
```
class GenList {   //广义表类定义
private:
    GenListNode *first;
    void traverse(GenListNode * ls);
   //遍历广义表
    void Remove(genListNode * ls);
   //释放广义表
public:
    GenList( char& value);
          //value是指定的停止建表标志数据
    ~GenList();
    void traverse();
}
```

递归遍历广义表

```
void GenList::traverse() {
    traverse(first);
}
```

```
void GenList::traverse(GenListNode *ls)
{
    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);
    else cout<<ls->value.hlink
        ->value.listname;
    if(ls->tlink!=NULL) cout<<',';
  }
  traverse(ls->tlink);
} //endif(ls!=NULL)
else cout<<')';
} //end function
```



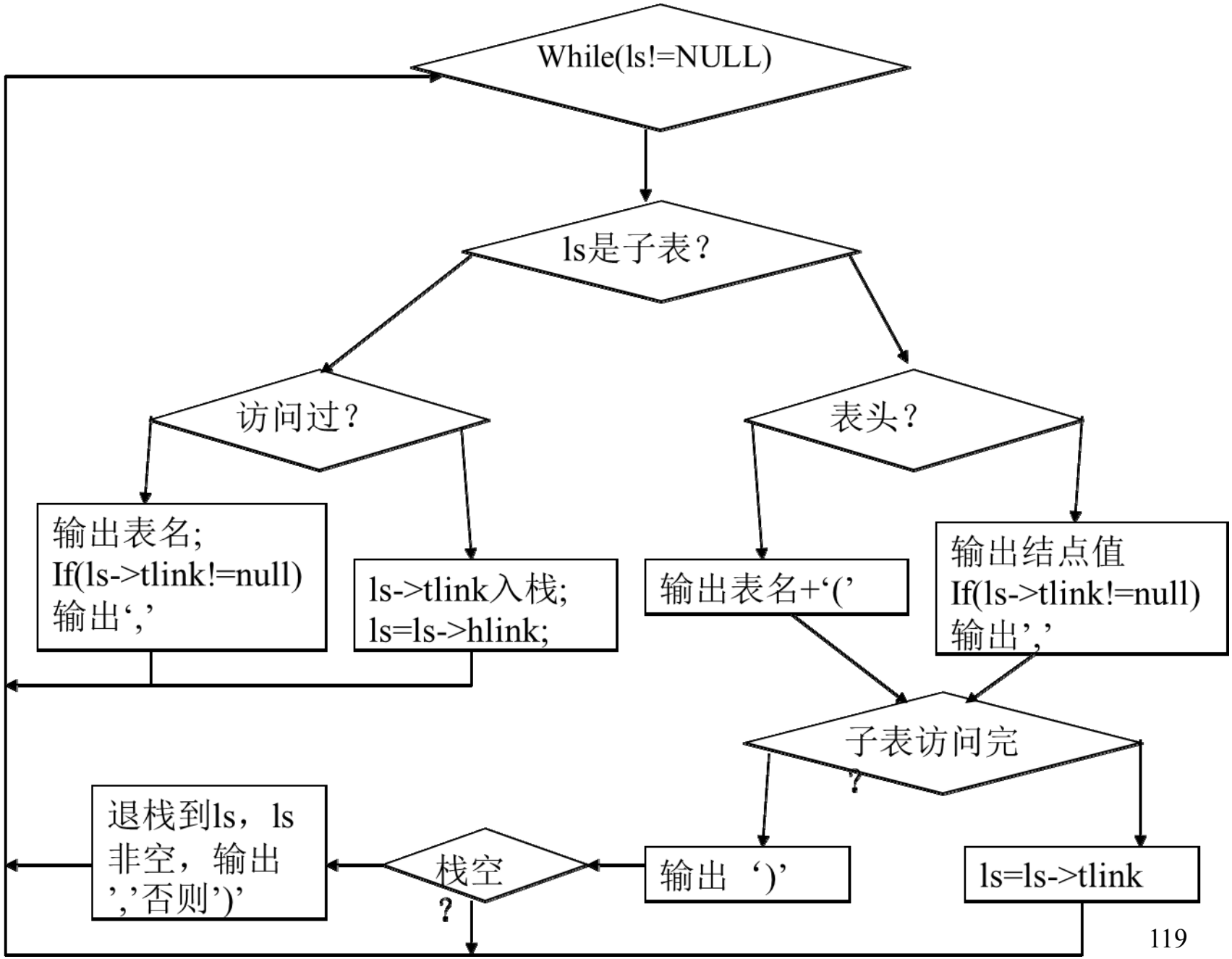
A(C(E(x, y), a), D(E, e))

非递归遍历

利用栈实现

栈中存放回退时下一将访问的结点地址tlink.

Void GenList::traverse(GenListNode *ls)


While(ls!=NULL)
ls是子表？
访问过？
表头？
输出表名;
If(ls->tlink!=null)
输出‘,’
ls->tlink入栈;
ls=ls->hlink;
输出表名+‘(’
输出结点值
If(ls->tlink!=null)
输出’,’
子表访问完
？
退栈到ls，ls
非空，输出
’,’否则’)’
栈空
？
输出 ‘)’
ls=ls->tlink

```
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 {     //非子表结点
    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
  }
}
```

本章小结

- 知识点
  - 递归算法和非递归算法
  - 广义表及其递归算法

常见题型

- 递归算法设计
- 递归转非递归
- 广义表的基本操作

- 课程习题
  - 笔做题——**5.3, 5.4**
    （以作业形式提交）
  - 上机题——**5.1, 5. 7**
  - 思考题——**5.5, 5.6**

Any Suggestion or Question

联系方式：

- 张玥杰

  Email: yjzhang@fudan.edu.cn