数据结构
Data Structure

2017年秋季学期

刘鹏远

栈与递归

预备：栈与函数调用

函数A在运行期间，函数代码存放在代码区，固定不变

数据的存储：

A的动态分配的变量，保存在堆区域
A的常量、全局变量、静态变量等存放在数据区
A的其他变量需要保存在系统指定的栈区域中

预备：栈与函数调用

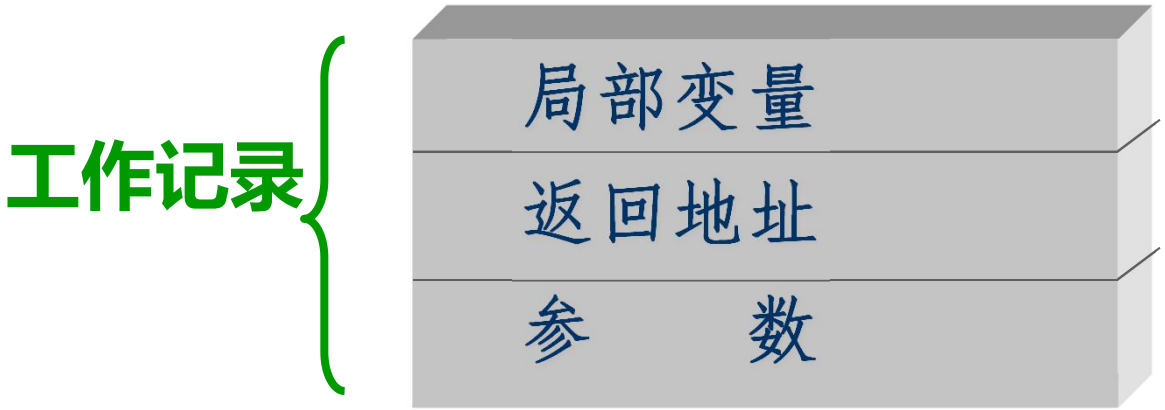
假设main函数调用函数A，在运行A前，栈要保存：

1、所有的实参（为了在程序A中使用）

2、返回地址（为了返回主函数）

3、A的局部变量、参数（A中使用）

以上这些信息构成一个工作记录，放到系统的栈中，形成一个栈帧。



预备：栈与函数调用

执行A，当前执行的函数，对应的工作记录一定在栈顶。
在执行完A以后：

1、将A栈帧弹出
2、按照A栈帧中返回的地址，回到主函数继续执行

函数多层次嵌套调用，也是如此，参考教材P56例子

递归：

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(定义)的；

若一个过程(函数)直接地或间接地调用自己，则称这个过程是递归的过程(函数)；

递归：

以下三种情况常常用到递归方法：

1）定义是递归的：阶乘、Ackman函数，fibonacci

2）数据结构是递归的：链表、二叉树等

3）问题的解法是递归的：汉诺塔、八皇后、迷宫等

递归：

1、定义是递归的　　如：阶乘函数

f(n) = 1　　当 n=0时

= n*f(n-1)　　当n>=1时

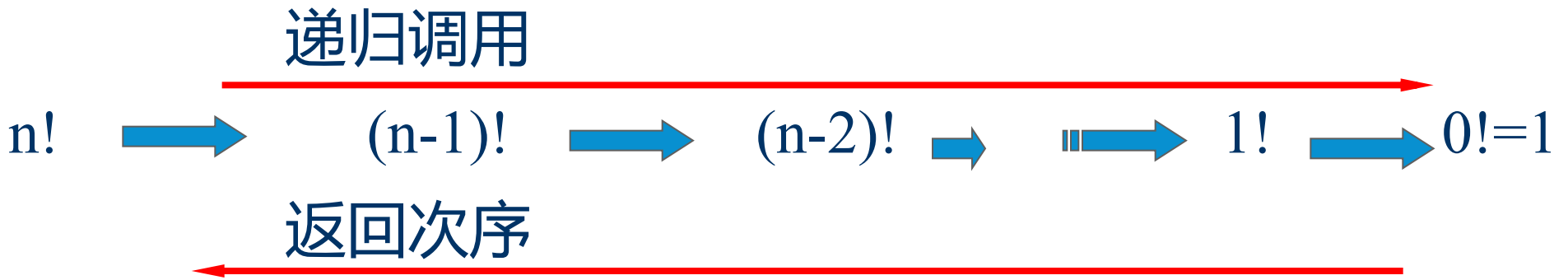
```
long Fact ( long n ) {#递归求解算法，假设n>=0
    if (n == 0) return 1;          函数出口
    else return n*Fact(n-1);
}
```

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

递归过程

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

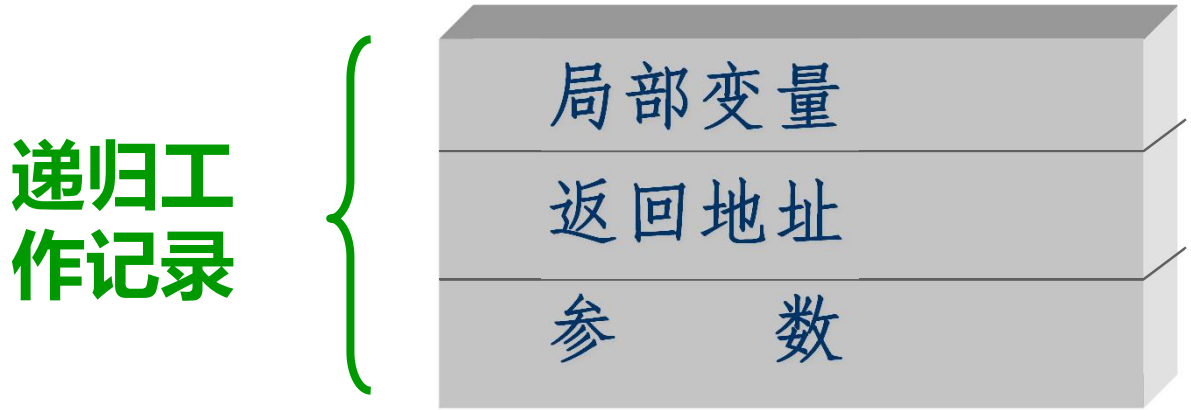


- 主程序第一次调用递归过程为外部调用；递归过程每次递归调用自己为内部调用。

系统对递归函数的调用，与一般函数的调用无本质上的不同

每一层递归包含的信息如：参数、局部变量、上一层的返回地址等，构成一个“工作记录”。

每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录。

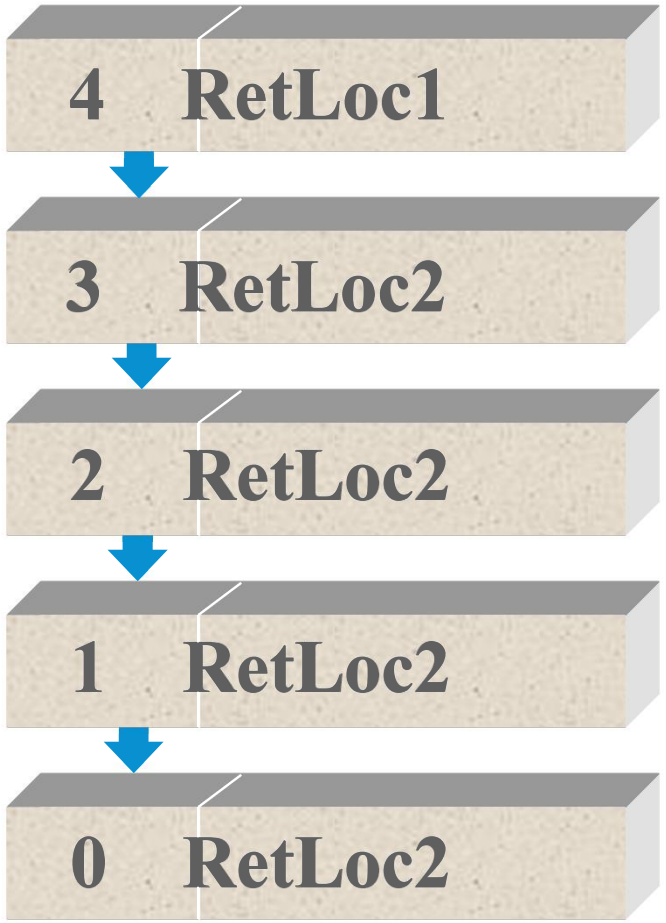


```
long Factorial(long n) {
    int temp;
    if (n == 0) return 1;
    else temp = n * Factorial(n−1);
    return temp;
}

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

ReturnLoc2

ReturnLoc1

递归调用序列



返回时的指令

RetLoc1 return 4*6 //返回24

RetLoc2 return 3*2 //返回6

RetLoc2 return 2*1 //返回2

RetLoc2 return 1*1 //返回1

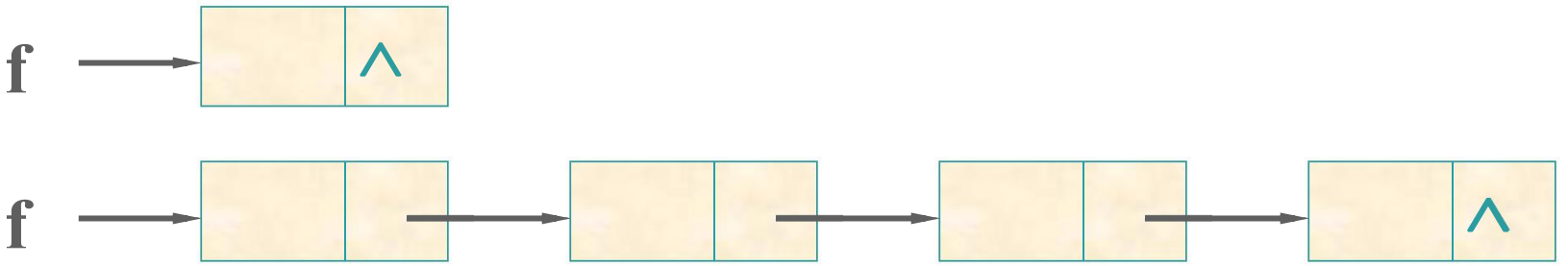
RetLoc2 return 1 //返回1

递归调用是栈的一种重要应用

依赖于栈，程序设计语言中实现函数及递归调用

递归函数调用自身时，返回地址相同，参数一般不同

2、数据结构是递归的：如单链表结构



一个结点，它的指针域为NULL，是一个单链表，其后继是一个空单链表;

一个结点，它的指针域非空，其后继仍是一个单链表。

指针结构表示链表，结构定义为指向自身节点的指针

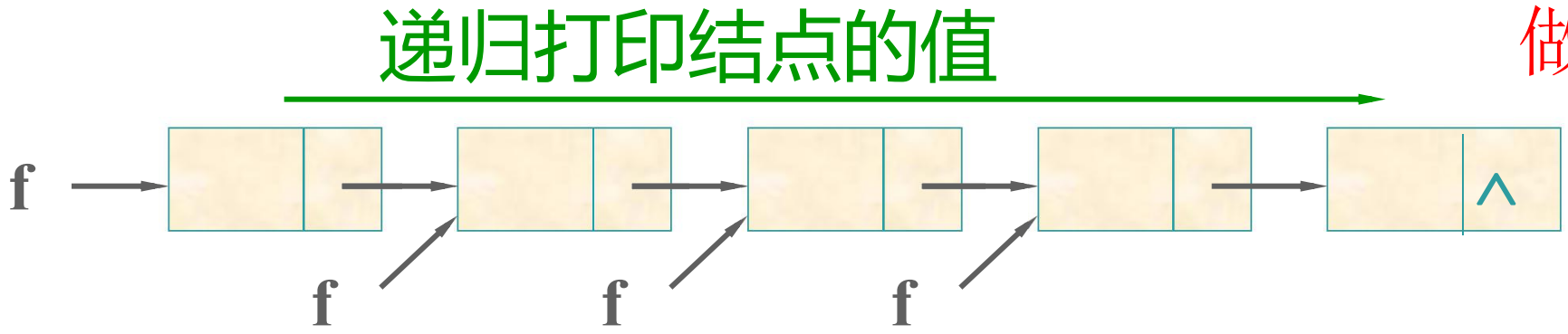
```
typedef struct node {          //定义结点
    elem_type data;
    struct node *next;         //节点包含指向自身（类型）的指针
} ListNode;

typedef  ListNode  *link_list;    //用节点定义指针
```

基于递归定义的数据结构，相应算法的实现均可采用递归方式。

对不带头结点的单链表f，正向打印所有结点所存储的数值。

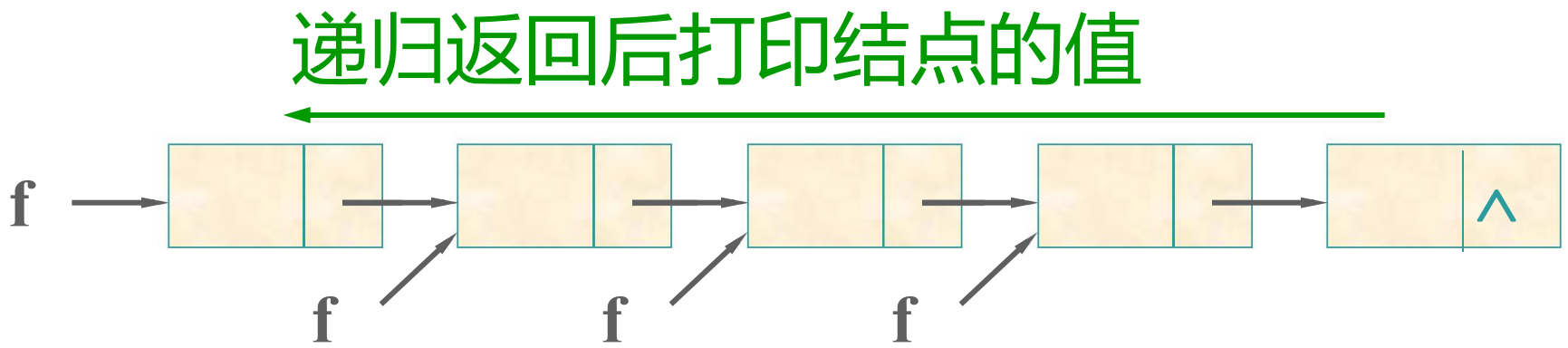
```
void print_value ( ListNode *f ) {
    if ( f ) {          //递归结束条件   出口
        printf ( f ->data );      //打印当前结点的值
        print_value ( f ->next);  //递归打印后续链表
}}
```

反向打印如何做？？



```
void print_value ( ListNode *f ) {
    if ( f ) {          //递归结束条件
        print_value ( f ->next);   //递归
        printf ( f ->data );   //返回后，打印节点 }}
```

先将f->next开始的链表倒序打印; 然后打印当前节点f-data；



3、问题的解法是递归的：如汉诺塔问题解法

假设将写好的这个Hanoi函数可以解决汉诺塔问题。

如果只有一个盘子(n=1)，则直接从A-->C

否则：

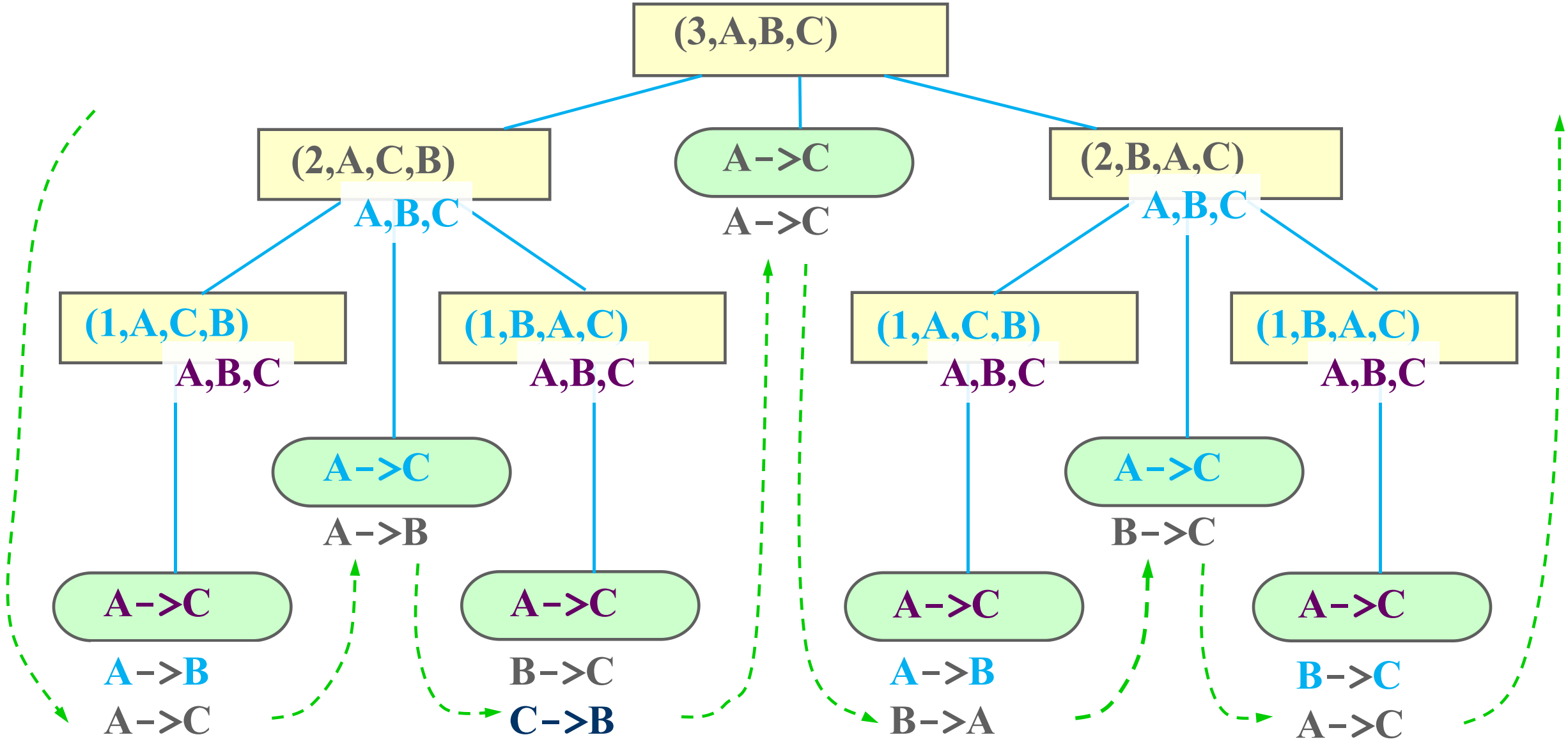
①利用Hanoi将 A 柱上的(n−1) 个盘子移到 B 柱；(利用C)

②将 A 柱上最后一个盘子直接移到 C 柱上；

③利用Hanoi将 B 柱上的(n-1) 个盘子移到 C 柱。(利用A)

分（减）治法

```
//汉诺塔的递归解法
void Hanoi ( int n, char A, char B, char C ) {
 //用A、B、C代表三个柱子，算法模拟汉诺塔问题
    if (n == 1) printf ( " move %c", A, " to  %c ", C );
    else {
        Hanoi ( n-1, A, C, B );
        printf ( " move %c", A, " to  %c ", C );
        Hanoi ( n-1, B, A, C );
    }
}
```


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

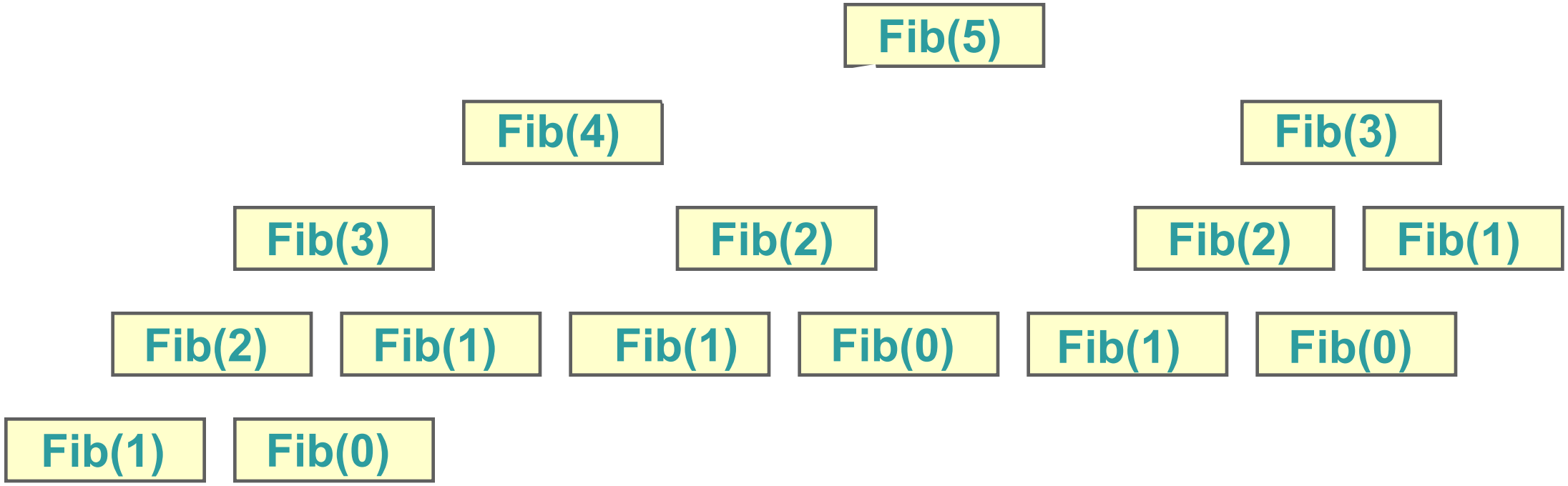
递归过程简洁、易编、易懂。

然而，有时重复计算多，有时受到系统规定的递归工作栈容量限制，或无用栈操作过多，有时造成效率低下。

例：斐波那契数列的递归函数Fib(n)：

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

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



递归调用次数可达

$$NumCall(k) = 2*Fib_{k+1} - 1$$

对Fib(5)，$NumCall(5) = 2*Fib_6 - 1 = 15$ 不必要的重复计算量大

且递归深度达到 5

0 1 1 2 5 8 13 21.....

递归消除

利用循环迭代：尾递归、单向(线性)递归

利用栈：其他递归

尾递归：函数只有一处递归调用，且是在函数返回前的最后一个计算操作。(不需要用递归栈保存函数内容)

尾递归改用迭代法实现

25	36	72	18	99	49	54	63

```
void recfunc( int A[ ],  int n ) {
//反向输出数组 A[] 的值，n 是当前打印元素下标
    if ( n >= 0 ) {
        printf ( "%d", A[n] );
        n--;
        recfunc( A, n );
    }
}//    调用时n为最后一个元素下标
```

程序中只有一个递归语句，且在程序最后。这时递归栈保存的内容无需再利用，所以不需要使用栈。

尾递归改为非递归的迭代算法直接用循环实现（while替换if）。

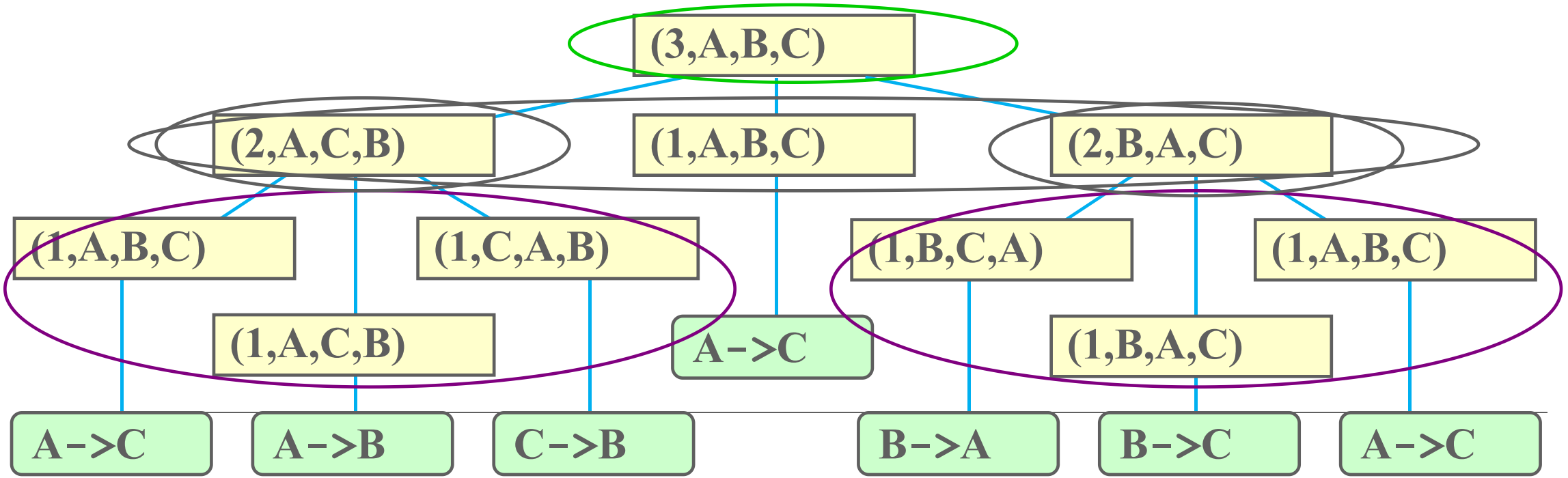
```
void iterfunc( int A[ ], int n ) {
 //消除了尾递归的非递归函数
        while ( n >= 0 ) {
              printf ( "value  %s ”, A[n] );
        n－－;
        }
}
```

线性递归改用迭代法实现

```
long FibIter ( long n ) {
    if ( n <= 1 ) return n;
    long a = 0,  b = 1,  c;
    for ( int i = 2;  i <= n; i++ ) {
        c = a+b;                    //求 Fi = Fi-2 + Fi-1
        a = b;                      //下一个 Fi-2 = 原来的 Fi-1
        b = c;                      //下一个 Fi-1 = 原来的 Fi
    }   return c;
}//单向(线性)递归：递归调用语句都处在递归算法的最后，且递归过程执行时虽然可能有多个分支，但可以通过保存前面计算的结果以供后面的语句使用。
```

利用栈将递归算法改为非递归算法

汉诺塔问题有两个内部递归调用的语句，不属于单向递归和尾递归，必须利用栈记录调用状态。



```
typedef struct {
        int n;
        char a, b, c;
} Quad;  定义栈元素的数据类型

void Hanoi ( int n, char x, char y, char z ) {
        LinkStack  S;   init(&S);
        Quad q;
        q.n = n;   q.a = x;   q.b = y;   q.c = z;
        Push (&S, q);                    //初始布局进栈
```

```
while ( ! is_empty(S) ) {      //当栈非空时
    pop(S, q);                 //取栈顶布局，退栈
    n = q. n;   x = q.a;  y = q.b;  z = q.c;
    if ( n == 1 ) printf ("Move  %c", x, " to %c", z, "\n" );
    else {
        q.n = n-1;   q.a = y;   q.b = x;   q.c = z;  push (S, q);//右
        q.n = 1;   q.a = x;   q.b = y;   q.c = z;     push (S, q);//中
        q.n = n-1;   q.a = x;   q.b = z;   q.c = y;  push (S, q);//左
}}}
```

递归与回溯

常用于搜索过程

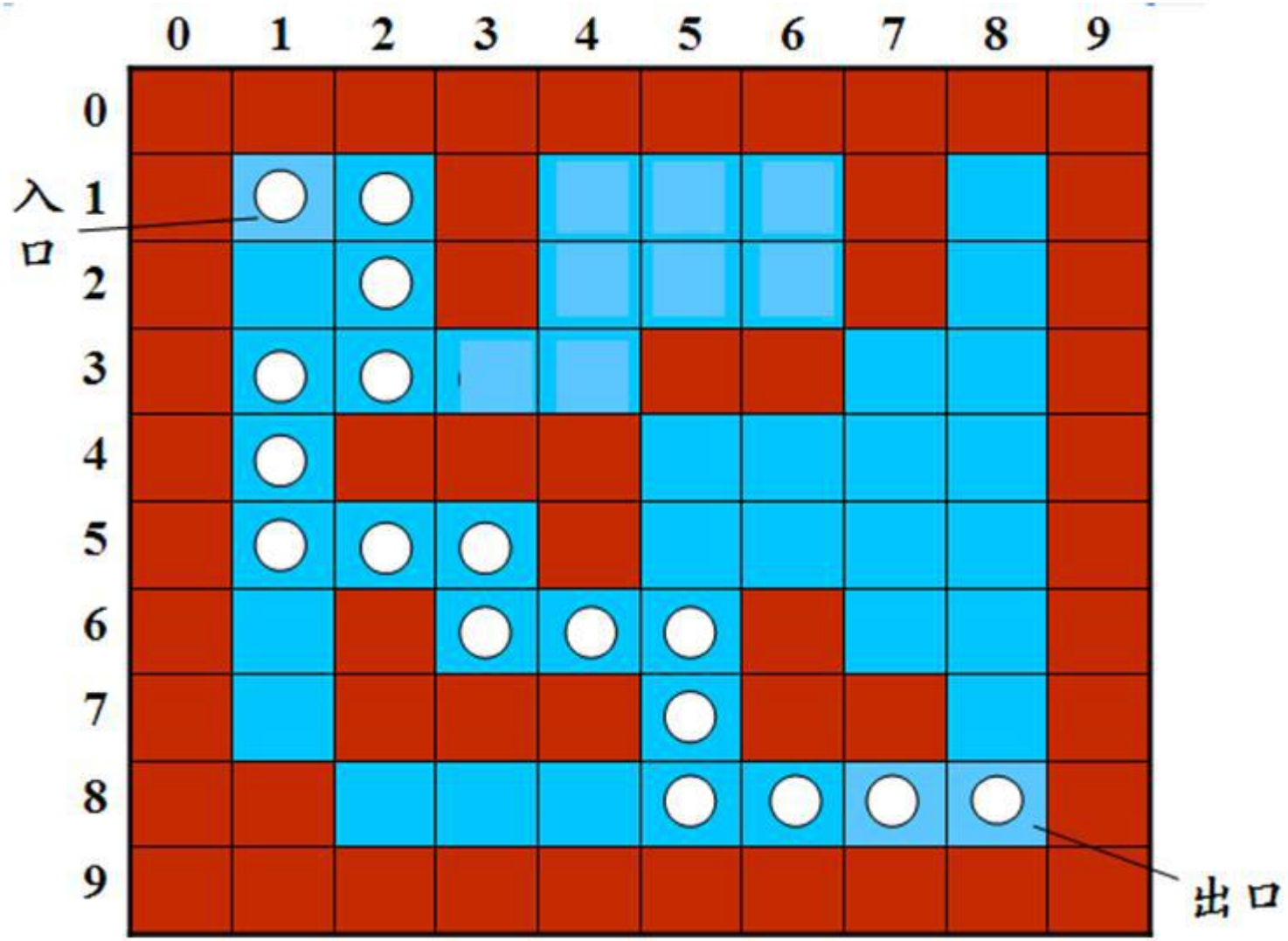
对一个包含有许多结点，且每个结点有多个分支的问题，可以先选择一个分支进行搜索。当搜索到某一结点，发现无法再继续搜索下去时，可以沿搜索路径回退到前一结点，沿另一分支继续搜索。

如果回退之后没有其他选择，再沿搜索路径回退到更前结点，…。依次执行，直到搜索到问题的解，或无解为止。

其实也就是深度优先DFS策略（后续树章节会详细讲到）

回溯法可用迭代+栈（迷宫应用），还可用递归算法求解。

回到迷宫问题



#	#	#	#	#	#	#	#	#	#
#	→	↓	#	$	$	$	#		#
#		↓	#	$	$	$	#		#
#	↓	←	$	$	#	#			#
#	↓	#	#	#				#	#
#	→	→	↓	#				#	#
#		#	→	→	↓	#			#
#	#	#	#	#	↓	#	#		#
#					→	→	→	⊖	#
#	#	#	#	#	#	#	#	#	#

开始就是墙的，没办法走
曾经不通的地方不能再走
除非走到不通，否则不能回头

“入口”到“出口”的简单路径（所经过的不重复通道方块）

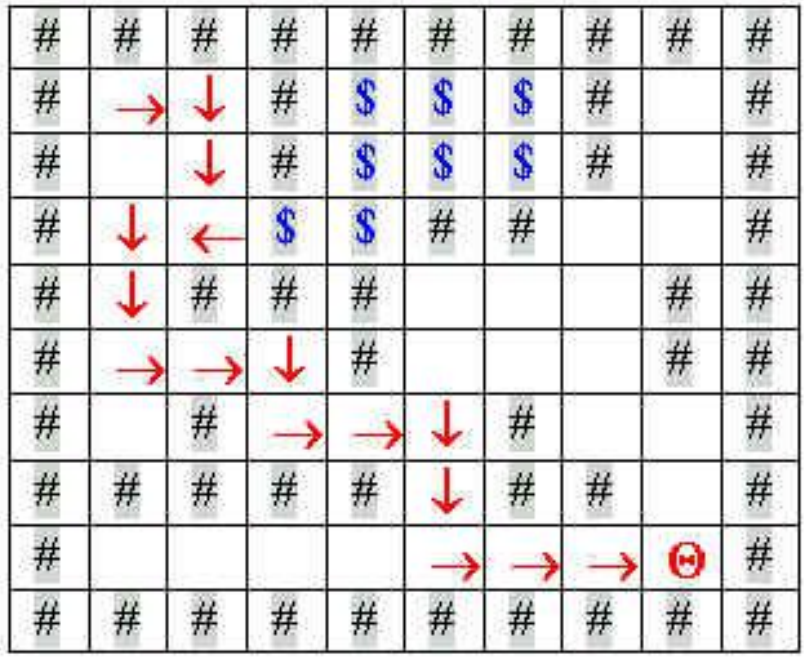
数据表示与定义：

1）迷宫表示：可用二维数组，自行定义

2）方格：坐标，标记(墙标记#、探过不通标记$、当前路径标记-、出口G，空白格o)

3）当前路径（栈）： 路径中各方块的位置;

4）行进下一方格:按四个方向按一定次序试探相邻方格。



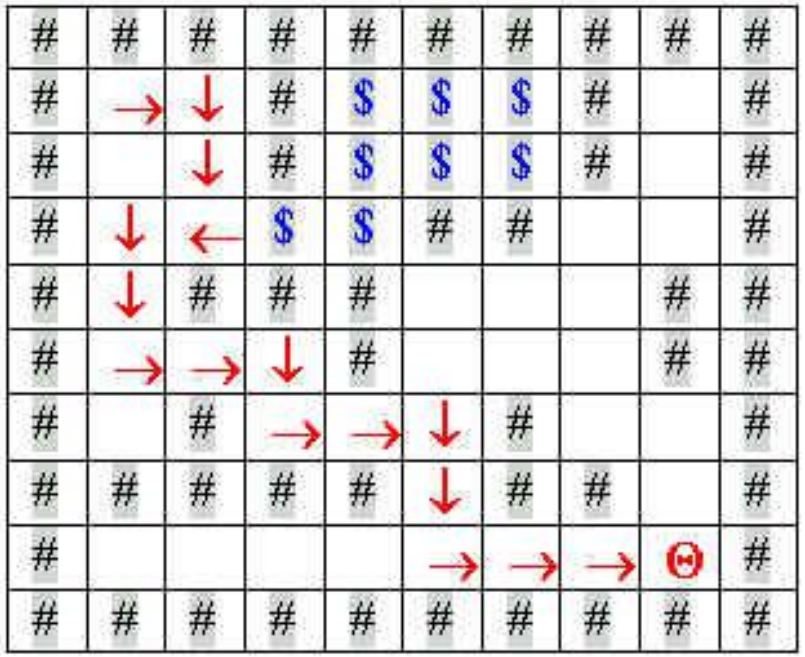
```
typedef struct{
        int x; int y;
}ElemType;
typedef struct{
    ElemType pos;//方格坐标（x,y）
    char data;//方格标记
    int di;//方向标记
}GridType;
GridType maze[m][n];//迷宫数组，
```
可设为全局变量

行进下一方格的探索结果有三种可能：

a. 探索方格为空白格，则：1、将该方格标记更改为-，2、该位置压入路径栈

b.探索方格为其他标记，则继续顺次探索

c.探索方格为出口，则1、该位置入路径栈，栈内即为路径通路。游戏结束。



如行进下一方格探索了所有4方向仍没有遇到空白格，当前方格标记改为$，出栈，路径回退一步（一直走到道路尽头，不撞南墙不回头）

利用栈的迷宫解法关键步骤（有省略）伪码！！

```
do{
    GetTop(S, &CurPos);//栈顶元素作为当前方格位置
    while(maze[CurPos].di<=4){//四个方向探索
        if(GoStep(&CurPos)) {//探索，能走，得到新CurPos
            if(CurPos==出口)    //找到出口，入栈，退出循环
                {push(&S, CurPos);break;}
//没到出口 push(&S, CurPos); maze[CurPos].data='-';}
        else maze[CurPos].di++;//否则该方格方向++
    }//跳出循环
```

```
            if(maze[CurPos].di>4){//确实没有前进的路则
                pop(S, &CurPos);//退栈，回退一步
                 maze[CurPos].data='$';//标记不通}
    }
while(!S.Is_empty()&&CurPos!=end);
```

递归与回溯---迷宫问题的递归解法

可以先选择一个分支进行搜索。当搜索到某一结点，发现无法再继续搜索下去时，可以沿搜索路径回退到前一结点，沿另一分支继续搜索。

0、假设存在某种搜索方法（函数），我们正在编写。

1、如果当前搜索点是终点，则搜索结束

2、如当前搜索点是可能的通路点，则：

做标记表示走到这里

依次向四个方向用这个搜索方法进行搜索（递归）

标记此位置走过，不成，准备回溯

递归与回溯---迷宫问题的递归解法

```
void search(int sx, int sy){
    if(maze[sx][sy] == 'G') exit(0);
    if(maze[sx][sy] == 'o'){
        maze[sx][sy] = '-';
        search(sx,sy+1);
        search(sx+1,sy);
        search(sx,sy-1);
        search(sx-1,sy);
        maze[sx][sy] = '$';
    }}
```

期中考试（预期下周二下午上机随堂考试）
考试范围为线性结构（应不考串）：

1、线性表---顺序表，链表(有头无头)
2、栈---顺序栈、链栈(有头无头)
3、队列---顺序栈（循环）、链栈(有头无头)
4、在这些结构上的应用(预先实现好结构)

```
typedef  struct{
     ElemType  *elem;      /* 存储空间的基址base    */
     int             length;     /* 当前长度                */
     int             size;       /* 当前分配的存储容量    */
  }SqList;
```

```
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}SqStack;
```

```
typedef struct {
     ElemType *elem;
     int front, rear;
} SqQueue;
```

动态分配，Destroy需要free分配空间

```
typedef  struct  Node{
    ElemType  data;          /*数据域，保存结点的值 */
    struct Node *next;       /*指针域*/ ( 双向需两个 )
}Node *LinkList, *LinkStack;      /*结点的类型 */
```

```
typedef struct LinkQueue{
    Node *front;
    Node *rear;
}LinkQueue;
```

```
typedef struct STNode{
    ElemType  data;
    int  cursor;
}STList;
STList mylist[MAXSIZE];
```

带头节点：初始化时候要分配一个头结点。操作实现简单。

无头节点：初始化无需分配节点。后续操作实现相对麻烦。

实际应用：

0、分析问题的逻辑结构
1、选择合适的逻辑结构
2、分析问题，设计算法
3、根据算法的操作，选择合适的存储结构
算法性能（时空复杂度）

下午上机及作业：

用二维数组表示迷宫。迷宫可用字符画出，自定义各类方格：比如墙为'#'，路为'o'，走过的路径可用'-'等。

1、利用栈求解迷宫

需要最后依次打印出走出迷宫的路径坐标

2、递归求解迷宫

要求递归方法得到每一步，并在屏幕显示走的每一步（主要是控制递归中对迷宫的输出，即每步都打印迷宫）。

下周课程进度：

多维数组

期中考试（预期），上机随堂考试。