数据结构

第四讲 栈

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年9月28日


新浪首页 - Windows Internet Explorer
www.sina.com.cn
文件(F) 编辑(E) 查看(V) 收藏夹(A) 工具(T) 帮助(H)
SOSO
搜索
收藏夹
建议网站
免费 Hotmail
获取更多附加模块
百度
新浪首页
登录名
sina新浪网
sina.com.cn
北京 晴 29~16℃

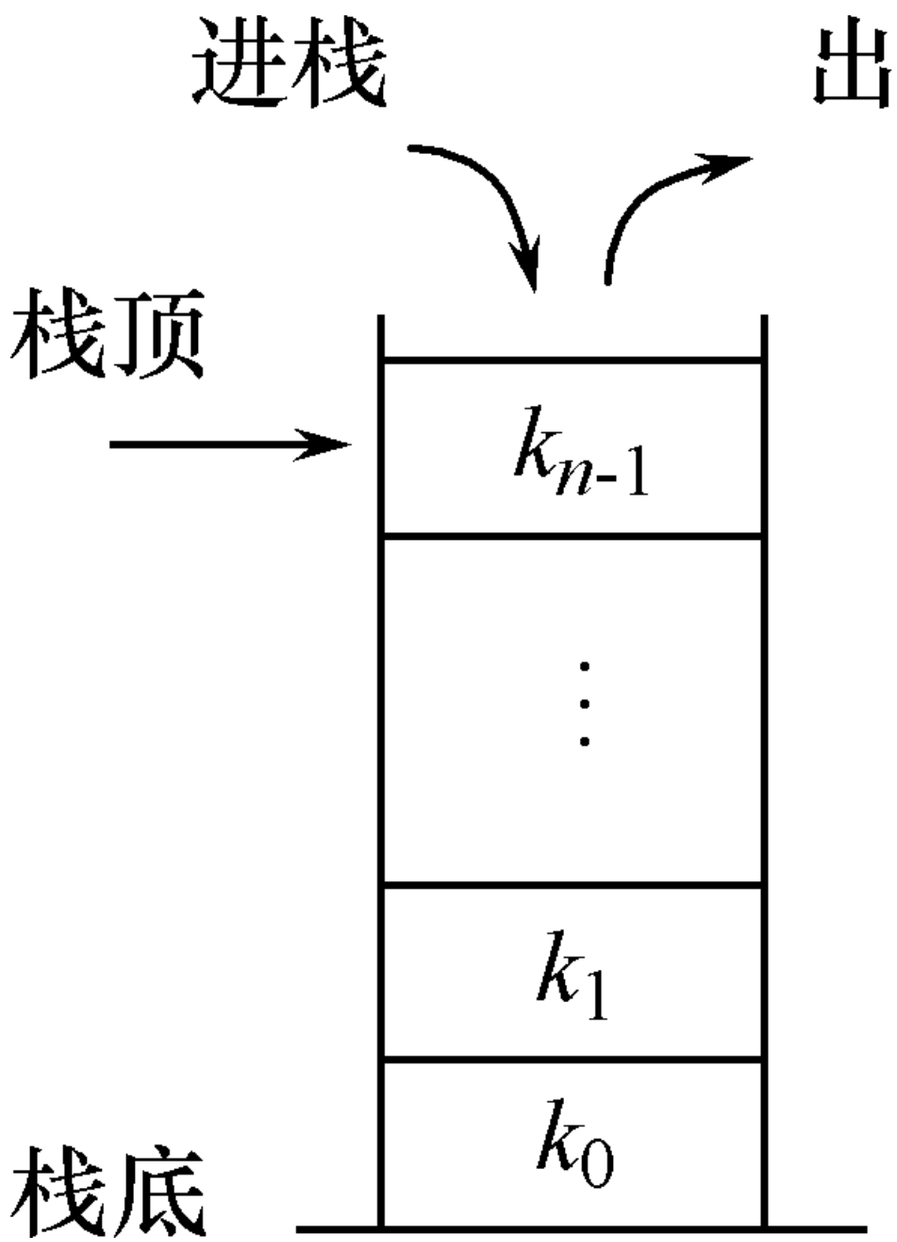
课程内容

- 栈及其抽象数据类型
- 栈的实现
- 栈的应用

栈及其抽象数据类型

- 基本概念
  - 栈是一种特殊的线性表，它所有的插入和删除都限制在表的同一端进行。
  - 表中允许进行插入、删除操作的一端叫做栈的顶。
  - 表的另一端则叫做栈的底。
  - 当栈中没有元素时，称之为空栈。
  - 栈的插入运算通常称为进栈或入栈，
  - 栈的删除运算通常称为退栈或出栈。

后进先出（LIFO）表



抽象数据类型

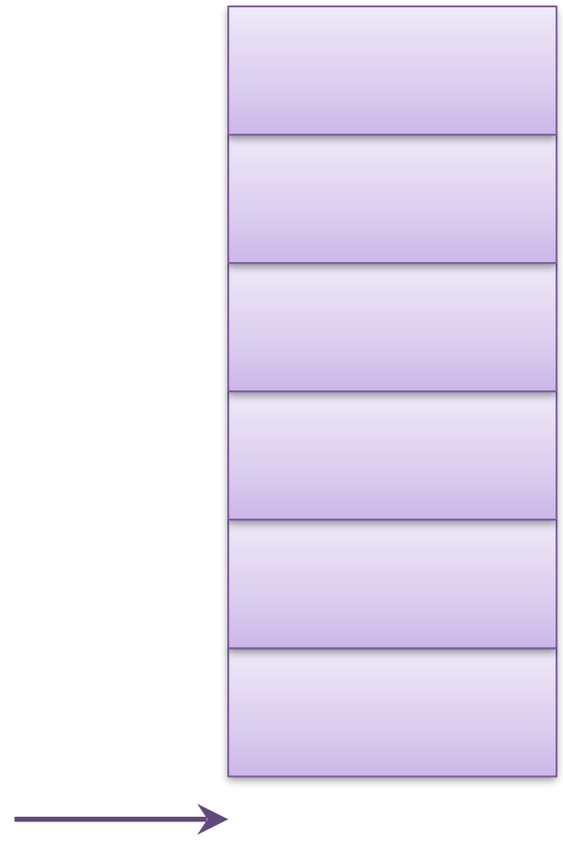
ADT Stack **is**
operations
  Stack createEmptyStack (void)
    创建一个空栈。
  int isEmptyStack (Stack st)
    判断栈st是否为空栈。
  void push (Stack st, DataType x)
    往栈st的栈顶插入一个值为x的元素。
  void pop (Stack st)
    从栈st的栈顶删除一个元素。
  DataType top (Stack st)
    求栈顶元素的值。
end ADT Stack

栈的实现

- 用顺序的方式实现栈时，可定义如下：

```
struct  SeqStack  {     /* 顺序栈类型定义 */
   int  MAXNUM;      /* 栈中最大元素个数 */
   int  t;   /* t < MAXNUM，指示栈顶位置而非元素个数 */
   DataType  *s;
};
typedef  struct SeqStack  *PSeqStack;
/* 顺序栈的指针类型 */
```

顺序表结构表示的栈



栈的溢出问题

- 由于栈是一个动态结构，而数组是静态结构，因此会出现所谓的溢出问题。
- 当栈中已经有 MAXNUM个元素时，如果再作进栈运算，则会产生溢出，通常称为上溢（Overflow）。
- 而对空栈进行出栈运算时也会产生溢出，通常称为下溢（Underflow）。

基本运算的实现

- 创建一个空栈

  PSeqStack createEmptyStack_seq(int m)

- 与创建空表PSeqList createNullList_seq(int m)类似，需为栈结构申请空间，不同之处是将栈顶变量赋值为-1。
- 判断栈是否为空栈

  int isEmptyStack_seq(PSeqStack pastack)

- 当pastack所指栈为空栈时，则返回1，否则返回0。

进栈运算

```
void  push_seq( PSeqStack pastack, DataType x ) {
/* 在栈中压入一元素x */
  if( pastack->t >= MAXNUM - 1  )
      printf( "Overflow! \n" );
  else{
    pastack->t = pastack->t + 1;
    pastack->s[pastack->t] = x;
   }
}
```

出栈运算

```
void  pop_seq( PSeqStack pastack ) {
      /* 删除栈顶元素 */
   if (pastack->t == -1) printf( "Underflow!\n" );
      else pastack->t = pastack->t - 1;
}
```

取栈顶元素运算

```
DataType  top_seq( PSeqStack pastack ) {
  /* 当pastack所指的栈不为空时，求栈顶元素的值 */
   if (pastack->t == -1 ) printf( "It is empty!\n" );
     else return (pastack->s[pastack->t]);
}
```

栈的实现

- 用链接方式实现栈时，每个结点的结构可定义如下：

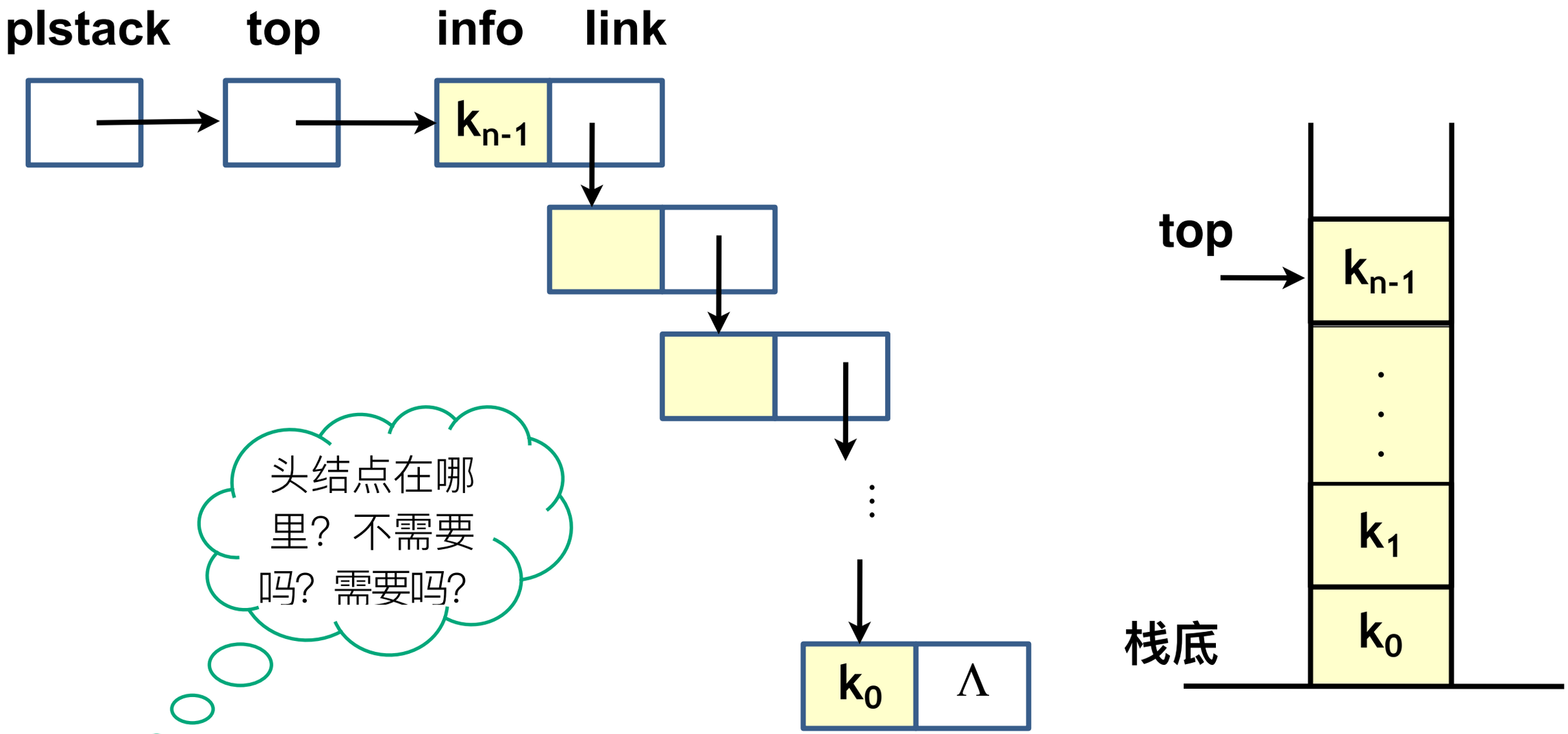
```
struct  Node;        /* 单链表结点 */
typedef  struct Node  *PNode;
                     /* 指向结点的指针类型 */
struct  Node {       /* 单链表结点定义 */
  DataType  info;
  Pnode  link;
};
```

LinkStack结构

- 为了强调栈顶是栈的一个属性，这里对栈增加了一层封装，引入LinkStack结构的定义。

```
struct LinkStack        /* 链接栈类型定义 */
  {
    PNode  top;         /* 指向栈顶结点 */
  };
typedef  struct LinkStack  *PLinkStack;
                        /* 链接栈类型的指针类型 */
```

plstack，plstack->top和plstack->top->info



创建空链接栈

```
PLinkStack  createEmptyStack_link(void) {
   PLinkStack plstack;
   plstack = (PLinkStack)malloc(sizeof(struct LinkStack));
   if (plstack != NULL)
      plstack->top = NULL;
   else
      printf("Out of space! \n");              /*创建失败*/
   return  plstack ;
}
```

判断栈是否为空

```
int  isEmptyStack_link( PLinkStack plstack ) {
   return (plstack->top == NULL);
}
```

进栈运算

```
void push_link( PLinkStack plstack, DataType x ) {
   PNode  p;
   p = (PNode)malloc( sizeof( struct Node ) );
   if ( p == NULL ) printf("Out of space!\n");
   else {
      p->info = x;
      p->link = plstack->top;
      plstack->top = p;
   }
}
```

出栈运算

```
void  pop_link( PLinkStack plstack ) {
   PNode  p;
   if(isEmptyStack_link(plstack))printf("Empty stack pop.\n");
   else{
      p = plstack->top;
      plstack->top = plstack->top->link;
      free(p);
   }
}
```

取栈顶元素运算

```
DataType  top_link( PLinkStack plstack ) {
    if (pastack->top == NULL )  printf( "Stack is empty!\n" );
    else  return(plstack->top->info);
}
```

栈的应用

- 栈与递归
- 迷宫问题

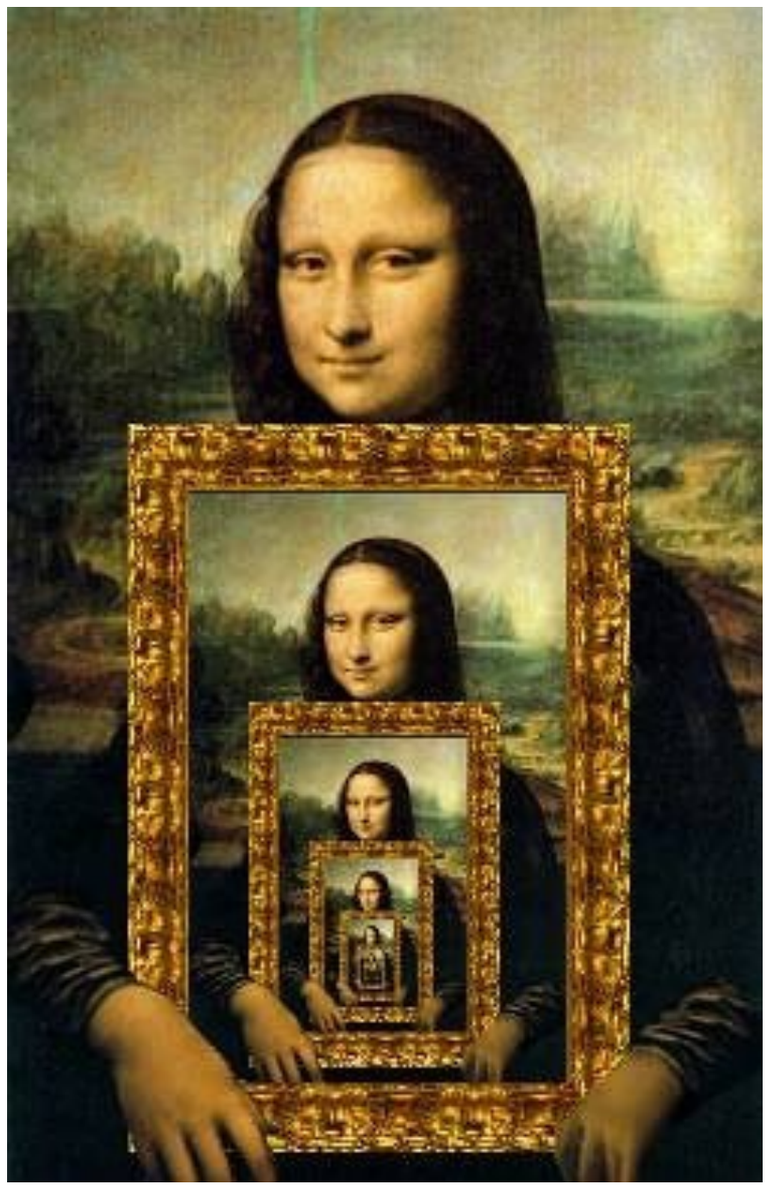
栈与递归

- 我们在引入递归概念的基础上，介绍栈是怎样用来实现递归，以及怎样把一个递归的函数转换成一个等价的非递归的函数。
- 设有一个程序sub要调用函数rout(x)，sub本身也是一个函数，称之为调用函数，而称rout为被调函数。调用函数中使用调用语句rout(a)来引起rout函数的执行，这里a称为实参，x称为形参。

递归

- 通常用来说明递归的最简单的例子是阶乘的定义，它可以表示成：

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$



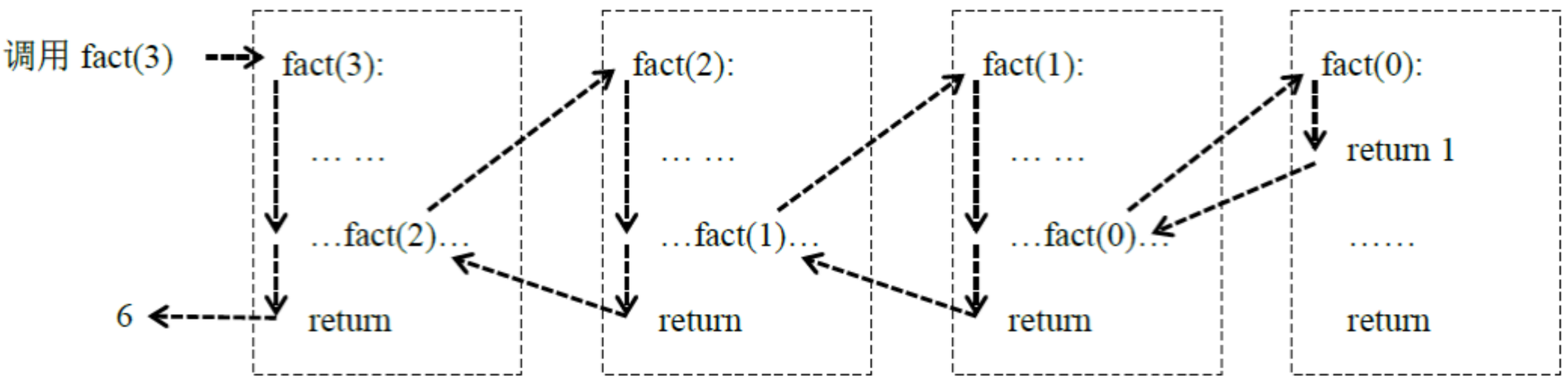
- 这种用自身的简单情况来定义自己的方式，称为递归定义。在n阶乘的定义中，当n为0时定义为1，它不再用递归来定义，称为递归定义的出口，简称为递归出口。

递归函数

```
int  fact( int n ) {
    int res=n;
    if ( n >1 )
       res=res* fact( n - 1 ) ;
    return res;
}
```

递归函数的执行过程

- 假设（主）程序中包含一个k=fact(3)语句，这个语句的执行过程如下图所示：



递归函数的执行过程

- **fact函数计算过程中程序运行栈的变化：**

3	–	–
n	**fact**	**res**

递归函数到非递归函数的转换

- 一般来说，函数调用的实现可以分解成下列三步来进行：
  (1)　传送调用信息。
  (2)　分配被调函数需要的数据区，并接收传送来的调用信息。
  (3)　把控制转移到被调函数的入口。

- 当被调函数运行结束，需要返回到调用函数时，一般的返回处理也可以分解成下列三步：
  (1)　传送返回信息。
  (2)　释放被调函数的数据区。
  (3)　把控制按返回地址转移到调用函数中去。

递归函数到非递归函数的转换

- 在非递归调用的情况下，数据区的分配可以在程序运行前进行，一直到整个程序运行结束才释放，这种分配称为静态分配。
- 在递归调用的情况下，被调函数的局部量不能分配给固定的某些单元，而必须每调用一次就分配一份，当前程序使用的所有的量（包括参数、局部变量和中间工作单元等），都必须是最近一次递归调用时所分配的数据区中的量。即所谓的动态分配。

递归函数到非递归函数的转换

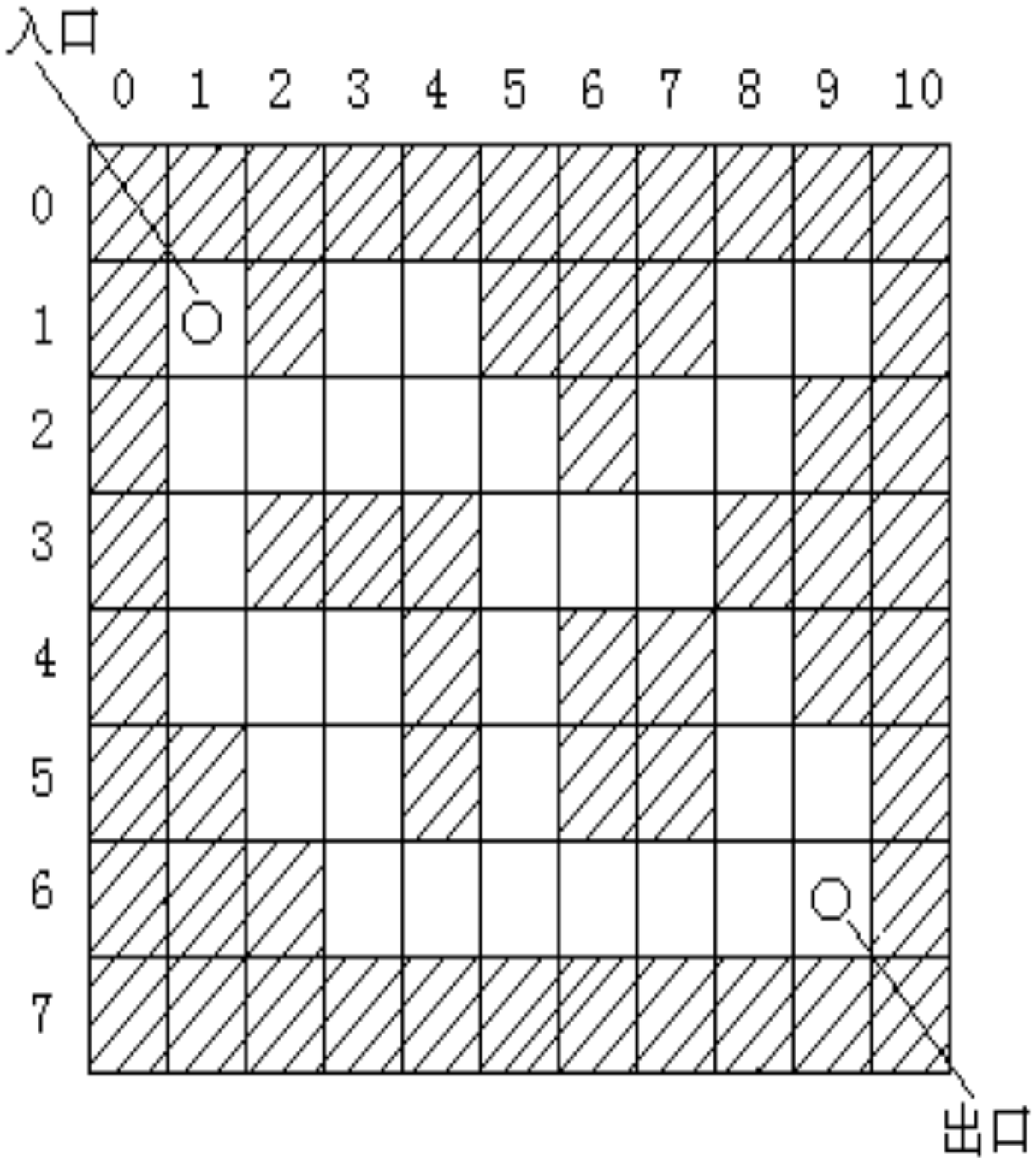
- 动态分配通常的处理方法是：在内存中开辟一个存储区域称为运行栈（或简称栈）。
- 每次调用时，在栈上为递归定义函数开辟一块区域，称为一个函数帧（或简称帧），保存这个调用的相关信息；
- 函数执行总以栈顶的帧为当前帧；
- 每次返回时，函数的上一层执行取得下层函数调用得到的结果，执行系统弹出已经结束的调用对应的帧，然后回到调用前上一层执行时的状态。

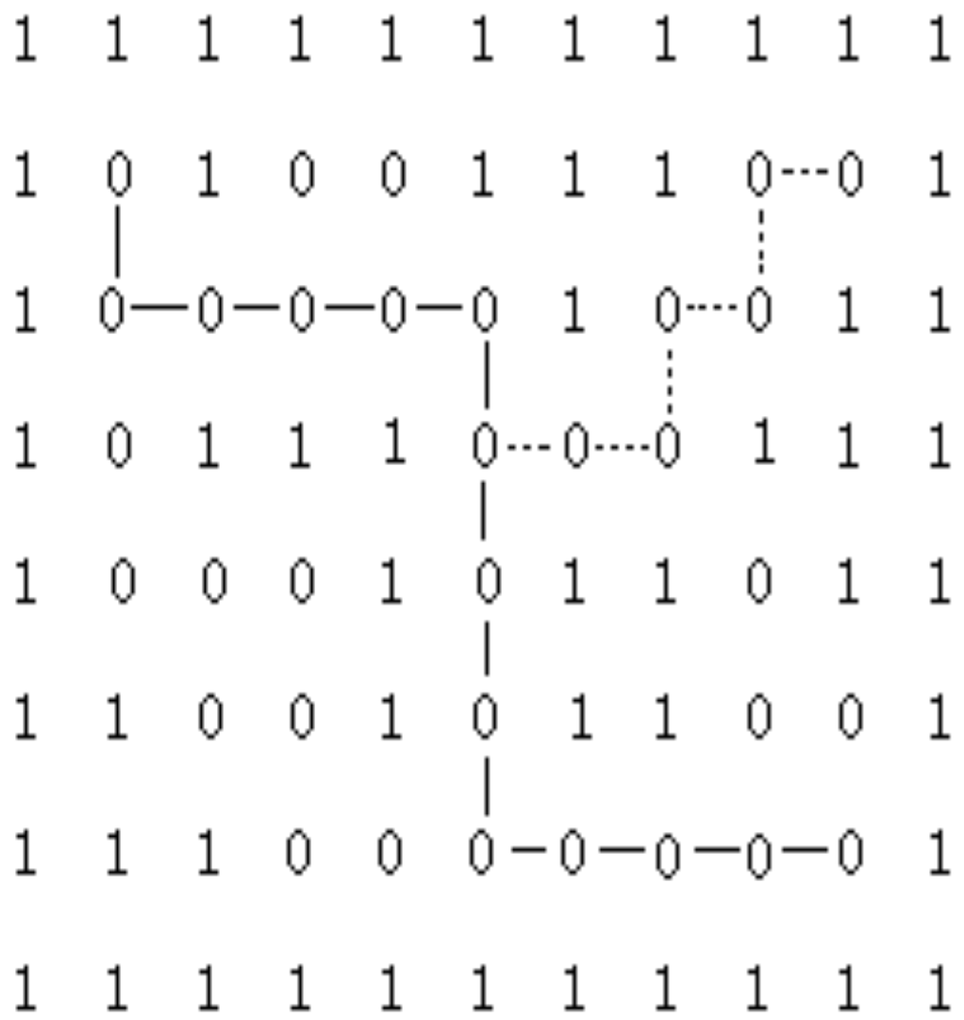
fact(int n)非递归计算

```
int  nfact( int n ) {
    int  res;
    PSeqStack  st;                          /* 使用顺序存储结构实现的栈 */
    st = createEmptyStack_seq( );
    while (n>0)
        {push_seq(st,n);
         n = n – 1;
        }
    res = 1;
    while (! isEmptyStack_seq(st))
        {res = res * top_seq(st);
         pop_seq(st);
        }
    free(st);
    return ( res );
}
```

迷宫问题

- 迷宫可用下页的左图所示的方块来表示，其中每个元素或为通道（以空白方块表示），或为墙（以带阴影的方块表示）。
- 迷宫问题要求的就是：从入口到出口的一个以空白方块构成的(无环)路径。

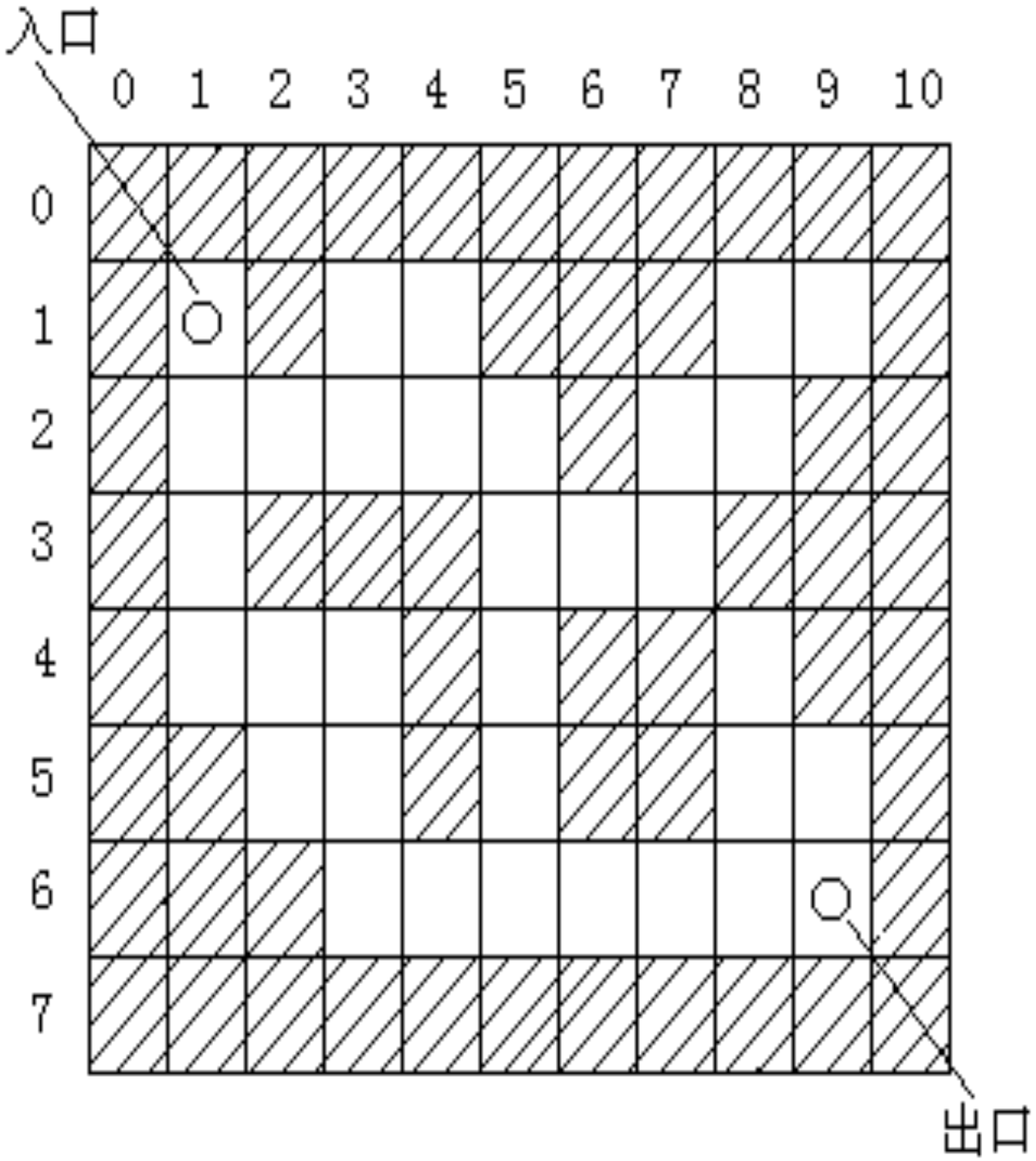

入口
0 1 2 3 4 5 6 7 8 9 10
0
1
2
3
4
5
6
7
出口


1 1 1 1 1 1 1 1 1 1 1
1 0 1 0 0 1 1 1 0 0 1
1 0 0 0 0 0 1 0 0 1 1
1 0 1 1 1 0 0 0 1 1 1
1 0 0 0 1 0 1 1 0 1 1
1 1 0 0 1 0 1 1 0 0 1
1 1 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1

求解迷宫问题的思路

- 从入口出发，沿某一方向进行探索，若能走通，则继续向前走；否则沿原路返回，换一方向再进行探索，直到所有可能的通路都探索到为止。

- 这类方法统称回溯法




```
1 1 1 1 1 1 1 1 1 1 1
1 0 1 0 0 1 1 1 0---0 1
  |               :
1 0—0—0—0—0 1 0---0 1 1
          |     :
1 0 1 1 1 0---0---0 1 1 1
          |
1 0 0 0 1 0 1 1 0 1 1
          |
1 1 0 0 1 0 1 1 0 0 1
          |
1 1 1 0 0 0—0—0—0—0 1
1 1 1 1 1 1 1 1 1 1 1
```

回溯法

- 从入口出发，采用试探方法，搜索到目标点(出口)的路径
  - 遇到出口则成功结束
  - 遇到分支点时选一个方向向前探索。这时需记录当时的分支点和在这里已试探过的分支(和尚未试探过的分支)
  - 若遇到死路(所有方向都不能走或已试探过)，就退回前一分支点，换一方向再探索。直到找到目标，或者所有可能通路都探索到为止

回溯法求迷宫问题一个解的框架

```
def mazeFrame:
    创建一个（保存探索过程的）空栈
    把入口位置压入栈中
    while   栈不空时:
        取栈顶位置并设置为当前位置
        while 当前位置存在试探可能:
            取下一个试探位置
            if 下一个位置是出口:
                打印栈中保存的探索过程然后返回
            if 下一个位置是通道:
                把下一个位置进栈并且设置为当前位置
```

数据结构

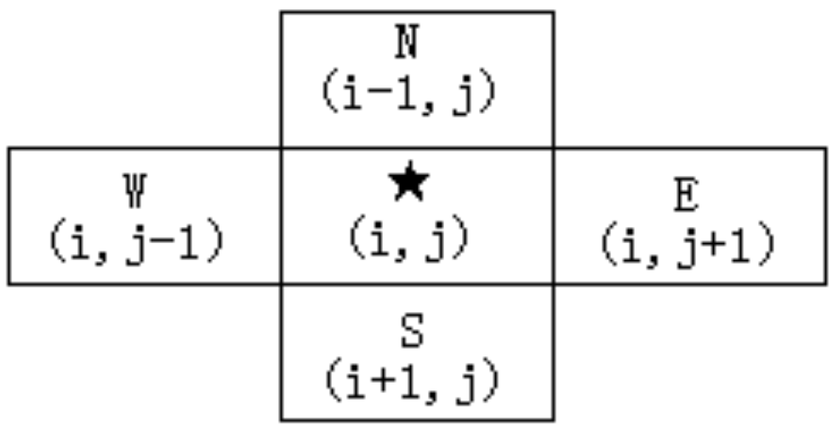
- 迷宫可用二维数组maze[m][n]来表示.
- 数组中元素为0的表示通道，为1的表示墙。
- 迷宫的入口处为maze[1][1]，出口处为maze[m-2][n-2]，它们的元素值必为0。
- 任意时刻在迷宫中的位置可用元素的行下标和列下标(i,j)来表示。

数据结构

在某一点maze[i][j]时，可能的运动方向有四个。可以建立一个数组direction[4][2]，给出相对于位置(i,j)的四个方向上，i与j的增量值。

若在位置(i,j)，要进入E方向的位置(g,h)，则可根据由该增量值表来修改(i,j)的坐标，

```
g = i + direction[0][0];
h = j + direction[0][1];
```



direction[4][2]

0	0	1	方向 E
1	1	0	方向 S
2	0	-1	方向 W
3	-1	0	方向 N

数据结构

栈中元素需要记录走过的位置和已经选择过的方向。

- 包括位置的行、列坐标，及在该位置已试探过的方向的最大下标(4个方向编码为directon数组的下标值0、1、2、3)
- 下面的算法使用顺序栈，栈元素类型：

```
typedef struct {
    int  x, y, d;  /* 当前位置(x,y)和已试探方向的最大下标 d */
} DataType;
```

回溯法实现

- 对迷宫问题，需要从入口出发搜索
  - 遇到出口时成功结束
  - 遇分支结点时记录信息，继续探查并可能回溯
- 搜索中把哪些位置入栈？存在两种合理的选择：
  - 从入口到当前探查位置，途径的所有位置都入栈
  - 只在栈里保存上述路径中存在未探查方向的那些位置。这一方式要求在入栈操作前检查所考虑位置的情况，有可能节省空间
- 仔细考虑，可以看到两个情况：
  - 把一个存在未探查方向的位置入栈，后来回溯到这里时也可能不再存在未探查方向了（原有的未探查方向在此期间已经检查过了）
  - 为在算法最后输出找到的路径，也需要知道路径上所有的位置

下面算法采用记录经过所有位置的方式，主要是为了输出结果路径

算法实现

```
void mazePath
(int *maze[],int *direction[],int x1,
    int y1,int x2,int y2,int M,int N) {
int i,j,k,g,h;
PSeqStack  st;
DataType element;
st = createEmptyStack_seq(M*N );
maze[x1][y1] = 2;
element.x = x1;
element.y = y1;
element.d = -1;
push_seq(st,element);
while (! isEmptyStack_seq(st)) {
element = top_seq(st);
pop_seq(st);
i = element.x;
j = element.y;
k = element.d + 1;
while (k<=3) {
 g = i + direction[k][0];
 h = j + direction[k][1];
 if (g==x2 && h==y2 && maze[g][h]==0) {
     printf("The revers path is:\n");
     while(!isEmptyStack-seq(st)){
        element=top_seq(st);
        pop_seq(st);
        printf("the node is: %d %d \n",
   element.x,element.y);    }
  return;
 }
 if (maze[g][h]==0) {
    maze[g][h] = 2;
    element.x = i;
    element.y = j;
    element.d = k;
    push_seq(st,element);
    i = g;    j = h;  k = -1;
  }
  k = k + 1;
}
}
printf("The path has not been found.\n");
}
```

讨论

- 迷宫问题是具有下列特征的一大类问题的代表
  - 存在一组可能的状态（位置、情况等）
  - 存在一个初始状态 s_0
  - 有一个或者多个结束状态（或存在一种判断结束的方法)
  - 对每个状态 s，neighbor(s) 表示与 s 相邻的状态（一步可达）
  - 有一个判断函数 valid(s) 判断 s 是否为合法的可行状态
  - 共同问题：找出从 s_0 到某个（或全部）结束状态的路径；或者是从 s_0 出发，设法找到一个/全部解（一个/全部结束状态）
- 这类路径搜索问题，都可以用递归的方法求解；也可以借助于一个栈，通过回溯法求解。这类问题也被称为搜索问题。
- 其他例子如：八皇后问题，骑士周游问题等。实际中的例子包括许多调度问题（例如背包问题），定理证明等
- 许多实际应用问题需要通过空间搜索的方式解决，如
  - 许多调度、规划、优化问题（如背包问题）
  - 数学定理证明（有一些事实和推理规则）

本讲重点

- **栈的ADT，存储表示和算法的实现**
- **栈与递归的内在联系**
- **栈在回溯法求解中的作用**

本讲内容非常重要！！


热烈
国庆节
快
乐