第四章　栈与队列

张史梁
slzhang.jdl@pku.edu.cn

操作受限的线性表

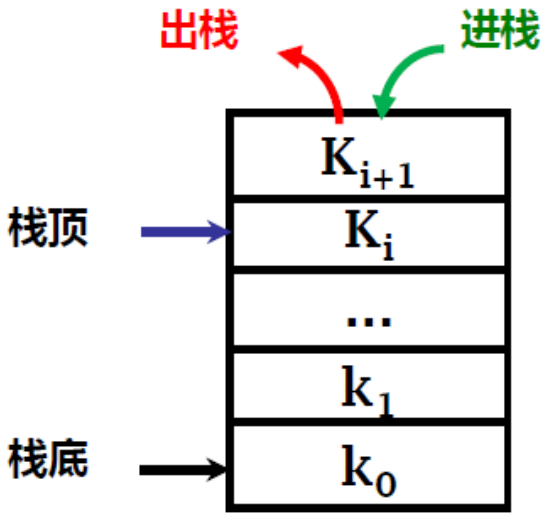
- 栈（Stack）
  - 运算只在表的一端进行
- 队列（Queue）
  - 运算只在表的两端进行

栈 – 内容提要

- 栈及其基本运算
- 栈的实现
- 栈的应用

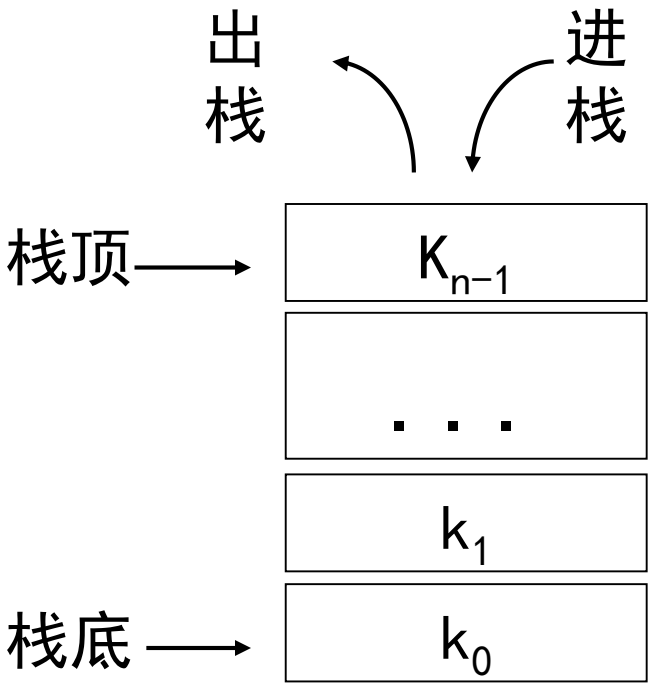
栈

- 后进先出（Last In First Out）
  - 一种限制访问端口的线性表
- 主要操作
  - 进栈（push）、出栈（pop）
- 栈的主要元素
  - 栈顶：栈的唯一可访问元素
    - 元素插入栈称为“入栈”、“压栈”、“进栈”（push）
    - 元素删除称为“出栈”、“退栈”（pop）
  - 栈底：另一端
- 应用
  - 表达式求值、消除递归、回溯算法、深度优先搜索



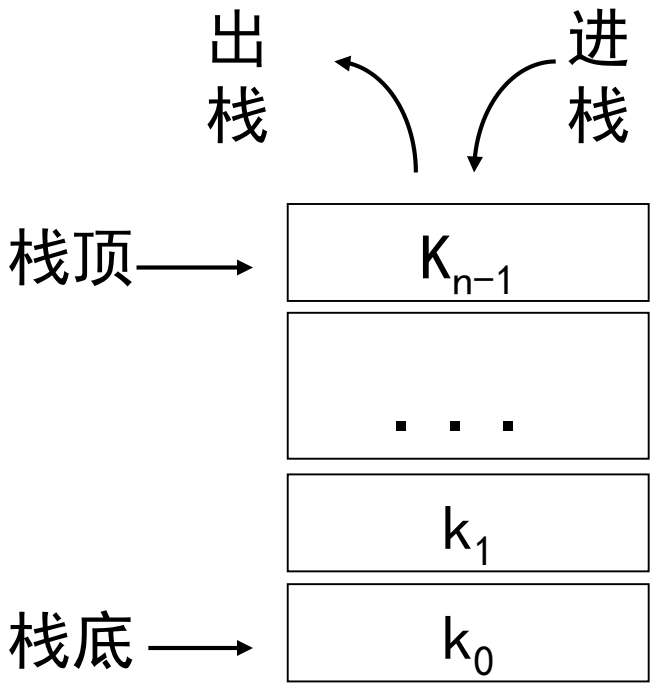
栈的示意图

- 每次取出（并被删除）的总是刚被压进的元素
- 而最先压入的元素则被放在栈的底部
- 栈顶的当前位置是动态变化的，常用一个称为栈顶指针的变量来标识。
- 当栈中没有元素时称为“空栈”

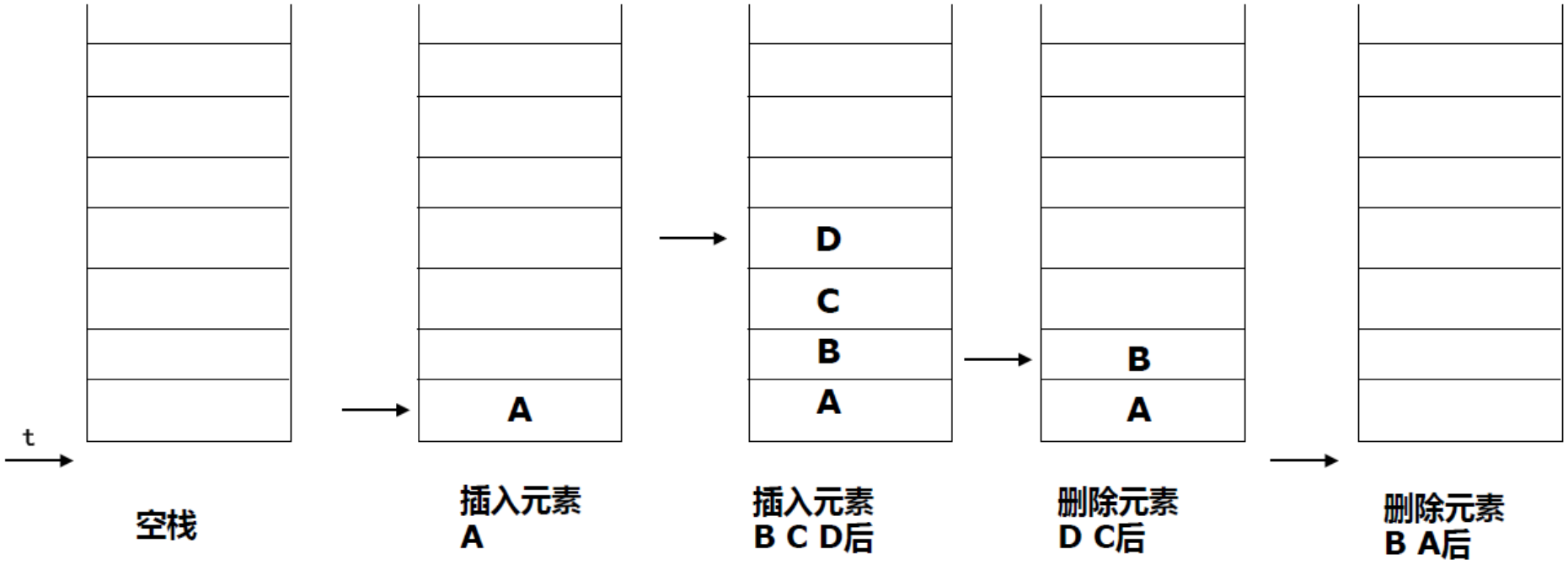


栈的基本运算

- 入栈（push）
- 出栈（pop）
- 取栈顶元素（top）
- 判断栈是否为空（isEmptyStack）
- 创建一个空栈（createEmptyStack）



栈的插入、删除



思考

- 如果入栈的顺序为1，2，3，4，则出栈的顺序可以有那些？
  - 1，2，3，4
  - 1，2，4，3
  - 1，3，2，4
  - 1，3，4，2
  - 1，4，2，3
  - 2，1，3，4
  - 。。。

栈的实现方式

- 顺序栈（array-based stack）
  - 使用向量实现，本质上是顺序表的简化
  - 关键是确定哪一端作为栈顶
  - 上溢、下溢问题

- 链式栈（linked stack）
  - 用单链表方式存储，其中指针的方向是从栈顶向下链接

栈的实现

- 顺序表示
  - 栈的顺序表示
  - 顺序栈基本运算的实现
- 链接表示
  - 栈的链接表示
  - 链栈基本运算的实现

栈的顺序表示

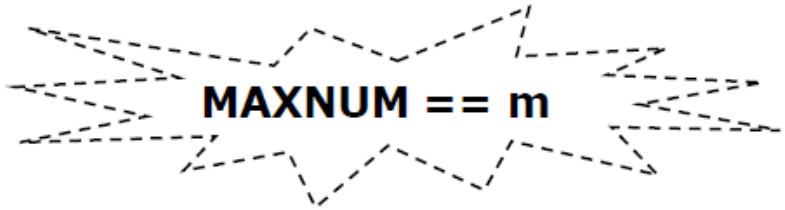
- 顺序栈类型定义:

```
// #define MAXNUM  1000  /* 栈中最大元素个数 */
struct  SeqStack
{
    DataType *s;    // DataType  s[MAXNUM];
    int     t;              /* 指示栈顶位置，空栈= -1 */
}
typedef struct SeqStack, *PSeqStack;
PSeqStack  pastack;          /*指向顺序栈的指针变量*/
```

pastack->t: 指向栈顶的变量
pastack->s: 存放栈中元素的数组
pastack->s[pastack->t]： 当前的栈顶元素

创建空栈

```
PSeqStack createEmptyStack_seq(int  m)  {/* 创建一个空栈 */
    PSeqStack pastack;    //PSeqList palist;
        pastack =(PSeqStack ) malloc (sizeof (struct SeqStack ) );
        if (pastack ! = null)   {
            pastack -> element = (DataType *) malloc (sizeof (DataType ) *m);
           if (pastack -> element ! = NULL)      {
                  palist -> t = -1 ;   return (pastack );
             }
             else free(pastack );
        }
        printf("out of space!\n");
        return (NULL);
}
```



创建空顺序表

```
PSeqList createNullList_seq (int  m)
{
   PSeqList palist;
   palist =(PSeqList ) malloc (sizeof (struct SeqList ) );
   if (palist! = null)
   {
      palist -> element = (DataType *) malloc (sizeof (DataType ) *m);
      if (palist -> element ! = NULL)
      {
           palist -> n = 0 ;   return ( palist );
       }
      else free( palist );
    }
      printf("out of space!\n");
      return (NULL);
}
```

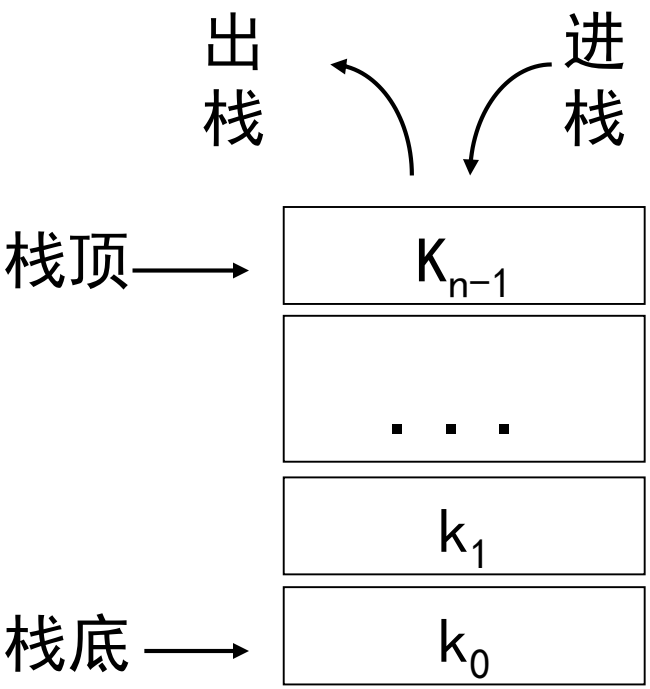
顺序栈的溢出

- 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，如果再进行进栈运算，则会产生溢出，通常称为上溢（overflow).
  - 而对空栈进行出栈运算也会产生溢出，通常称为下溢（underflow).
- 上/下溢出的条件？

栈满条件：pastack->t == MAXNUM-1
栈空条件： pastack-> t == -1
栈上溢条件：pastack-> t>= MAXNUM
栈下溢条件： pastack-> t < -1

顺序栈的基本运算

- 创建一个空栈；
- 判断栈是否为空栈；
- 往栈中插入一个元素
  - 考虑上溢错误
- 从栈中删除一个元素
  - 考虑下溢错误
- 取栈顶元素的值



顺序栈的基本运算

```
① /* 表示往pastack所指的栈中插入（或称推入）一个值为 x 的元素。*/
② int push_seq( PSeqStack pastack, DataType x )
③ {
④     if( pastack->t >= MAXNUM - 1 )  /* 满否？ */
⑤       {
⑥           printf("overflow!\n");
⑦           return ERROR;
⑧       }
⑨       else /* 压入 */
⑩       {
⑪           pastack->t = pastack->t + 1;  /* 先变化指针，再插入 */
⑫           pastack->s[pastack->t] = x;
⑬           return OK;
⑭       }
⑮ }
```

顺序栈的基本运算

```
① /* 表示从pastack所指的栈中删除（或称弹出）一个元素。 */
② Void pop_seq(PSeqStack pastack)
③ {
④       if (pastack->t ==-1)   /* 空否？ */
⑤         {
⑥               printf("Underflow!\n");
⑦         }
⑧         else    /* 弹出 */
⑨         {
⑩               pastack->t = pastack->t-1;
⑪          }
⑫ }
```

顺序栈的基本运算

```
① /* 表示从pastack所指的栈中删除（或称弹出）一个元素。 */
② Datatype pop_seq(PSeqStack pastack)
③ {
④       Datatype temp;
⑤      if (pastack->t ==-1)   /* 空否？ */
⑥       {
⑦             printf("Underflow!\n");
⑧             return ERROR;
⑨       }
⑩       else    /* 弹出 */
⑪       {
⑫             temp=pastack->s[pastack->t];
⑬             pastack->t = pastack->t-1;
⑭             return temp;
⑮       }
⑯ }
```

顺序栈的生成方向

- 若入栈动作使地址向高端生长，称为“向上生成”的栈
- 若入栈动作使地址向低端生长，称为“向下生成”的栈
- 通常我们使用“向上生成”的栈，即：
  - 空栈：top==base （t==-1）
  - 入栈后：top+1 (t++)
  - 出栈后：top-1 (t--)
  - 栈满：top-base==stacksize (t==MAXNUM-1)

栈的实现

- 顺序表示
  - 栈的顺序表示
  - 顺序栈基本运算的实现
- 链接表示
  - 栈的链接表示
  - 链栈基本运算的实现

链栈定义

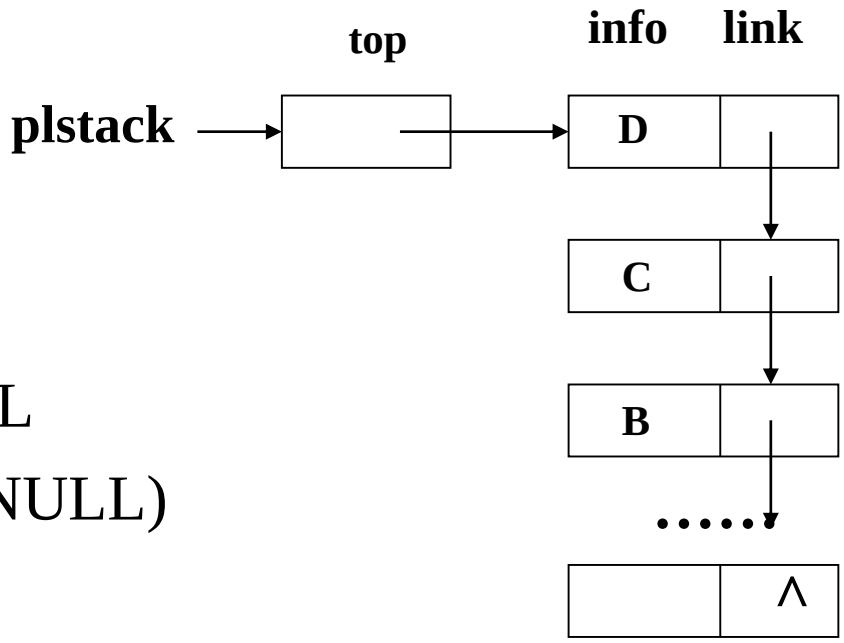
- 把栈组织成一个单链栈，这种栈可称为链栈或链接栈

```
struct  Node;                          /*单链表结点结构*/
typedef struct Node *PNode;            /*指向结点的指针类型 */
struct Node {
     DataType  info;
     PNode   link;                     /* 下一个结点 */
};

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

链栈的表示

- PLinkStack plstack;
- plstack->top: 指向栈顶的指针
- plstack->top->info: 栈顶元素值



- 栈空条件： plstack->top = NULL
- 下溢条件： if (plstack->top == NULL)
  还要继续删除

插入和删除元素

- 压入元素：

在top与栈顶之间插入

（s->link = top

top = s；）

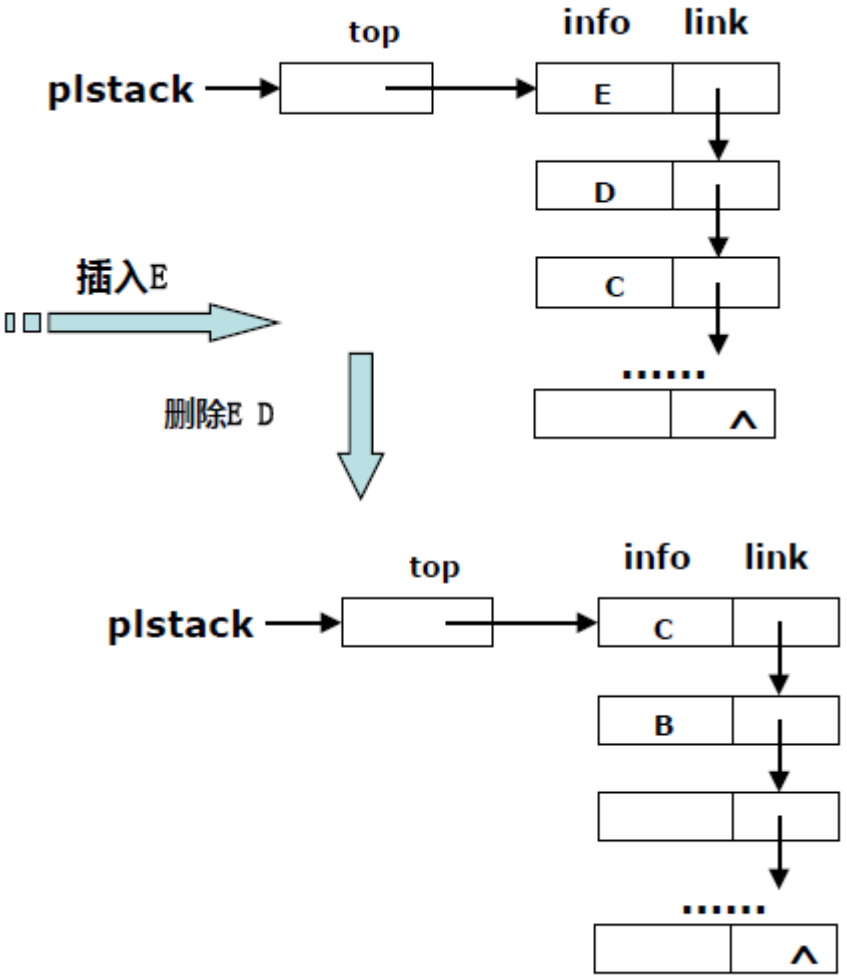
- 弹出元素：

删除栈顶元素

(q = top;

top= q->link;

free(q);)



链栈基本运算的实现

- 创建一个空栈；
- 判断栈是否为空栈；
- 往栈中插入一个元素；
- 从栈中删除一个元素；（考虑下溢错误）
- 取栈顶元素的值。

链栈的基本运算

```
① /* 创建一空链接栈，返回指向空链接栈的指针。 */
② PLinkStack CreateEmptyStack( void )
③ {
④     PLinkStack plstack;
⑤     plstack = (PLinkStack)malloc( sizeof(struct LinkStack));  /* 申请头指针 */
⑥     if (plstack != NULL)
⑦     {
⑧         plstack->top = NULL;     /* 栈顶指针指向空 */
⑨     }
⑩     else
⑪         printf("空间不够! \n");
⑫     return plstack;
⑬ }
```

链栈的基本运算

```
① /* 表示往plstack所指的栈中插入（或称推入）一个值为 x 的元素。 */
② int push_link( PLinkStack plstack, DataType x )
③ {
④     PNode p;
⑤     p = (PNode)malloc( sizeof( struct Node ) ); /* 申请新结点 */
⑥     if(p==NULL)
⑦     {
⑧         printf("Out of space!\n");          return ERROR;
⑨     }
⑩     else /* 在栈顶插入新结点 */
⑪     {
⑫         p->info = x;     p->link = plstack->top;          plstack->top = p;
⑬         return OK;
⑭     }
⑮ }
```

链栈的基本运算

```
① /* 表示从plstack所指的栈中删除（或称弹出）一个元素。 */
② void pop_link( PLinkStack plstack)
③ {
④     PNode p;
⑤    if (isEmptyStack_link( plstack ) ) /* 首先判断空栈否？ */
⑥     {
⑦         printf("Empty stack pop. \n");
⑧     }
⑨     else    /*修改栈顶指针 */
⑩     {
⑪       p = plstack->top;        plstack->top = plstack->top->link;        free(p);
⑫     }
⑬ }
```

栈链的特点

- 运算受限的单链表，插入删除操作限制在表头位置进行
- 由于只能在链表头部进行操作，因此栈链没有必要像单链表那样添加头节点
- 栈顶指针就是链表的头指针，在头指针指向的地方进行插入删除操作
- 不必预先知道栈的最大尺寸，不必担心栈满（除非整个内存空间已满）。

内容提要

- 栈及其基本运算
- 栈的实现
- 栈的应用

栈的应用

- 栈是一种很重要、应用很广泛的数据结构，在编译和运行的过程中，就需要利用栈进行语法检查、表达式求值、递归功能的实现。
- 常见的例子
  - 数制转换
  - 括号匹配的检验
  - 表达式计算
  - 子程序 / 函数调用的管理
  - 消除递归
  - 迷宫问题

数制转换

- 十进制N和其它进制数的转换是计算机实现计算的基本问题，其解决方法很多，其中一个简单算法基于下列原理:

  N = (N div d)*d +N mod d

  (其中:div为整除运算,mod为求余运算)

  例如 $(1348)_{10}=(2504)_8$，其运算过程如下

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

数制转换

```
① /*输入一个十进制数，将其转换为八进制数*/
② void conversion( )
③ {
④       PSeqStack  pstack;    int temp;
        pstack =CreateEmptyStack ( 100 );
①       scanf (“%d”,n);
②       while(n)    {
③           push(pstack,n%8);
④           n=n/8;
⑤        }
⑥        while(!isEmptyStack_seq(pstack))   {
⑦          temp =top(pstack);
⑧          pop(pstack);
⑨          printf(“%d”, temp);
⑩        }
⑪}
```

括号匹配检验

- 表达式中允许括号嵌套，如何检查这些括号都是匹配的？

  ()()()()(((())())())()()()

- 如果有其他符号那？

  ()()()([][][]{}{}{({[}}} ‘’ “”

括号匹配的检验

```
1  void CheckMatch(char* p, SeqStack s){
2     char c;
3     while (*p){
4        switch (*p){
5        case '(':                    /*如果是左括号，将其入栈*/
6           StackPush(&s, *p++);
7           break;
8        case ')':
9           if (StackEmpty(&s)){     /*栈为空，说明没有左括号入栈*/
10             printf("缺少左括号！\n");
11             return;
12          }
13          else{
14             GetTop(&s, &c);      /*将栈顶元素与读入的右括号比较，如匹配，则将栈顶括号出栈*/
15             if (Match(c, *p))
16                StackPop(&s, &c);
17             else{   /*栈顶括号与读入的括号不匹配*/
18                printf("左右括号不匹配！\n");
19                return;
20             }
21          }
22       default:   /*如果读入的不是括号，指针后移一位*/
23          p++;
24       }
25    }
26    if (StackEmpty(&s))  /*栈为空，所有的字符序列读入完毕，说明括号序列匹配*/
27       printf("括号匹配！\n");
28    else
29       printf("缺少右括号！\n");
30 }
```

表达式计算

- 表达式求值是程序设计语言编译中的一个最基本的问题。
- 为讨论方便，对表达式做如下简化：
  1. 假定所有运算分量都是整数；
  2. 所有运算符都是整数的二元操作，且都用一个字符表示。

$$31 * (5 - 22) + 70$$

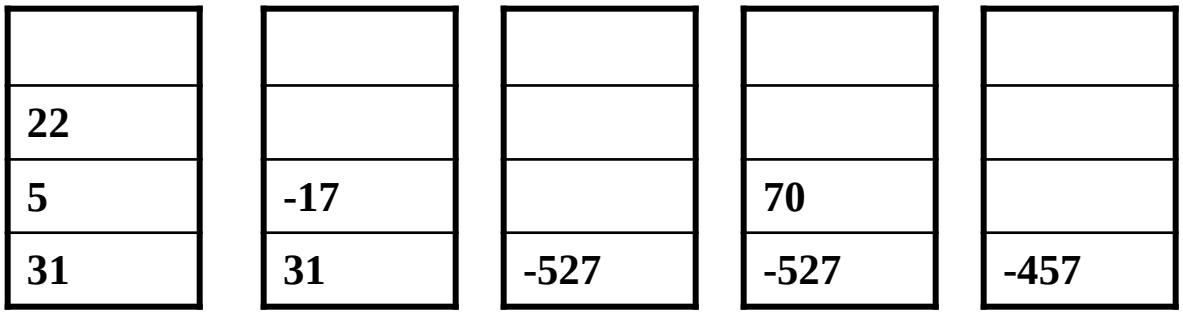
表达式的种类

- 中缀表达式：运算符都出现在它的两个运算分量之间；
  - e.g., 31 * (5 - 22) + 70
- 后缀表达式：运算符都出现在它的两个运算分量之后；
  - e.g., 31 5 22 - * 70 +
- 前缀表达式：运算符都出现在它的两个运算分量之前
  - e.g., +*31-5 22 70

后缀表达式的求值

- 后缀表达式的主要优点是可以写出非常简单的求值过程。
- 需要一个存放操作数的栈：
  - 从左往右扫描表达式，
  - 遇到操作数进栈；
  - 遇到运算符时从栈中弹出两个操作数计算，并将计算的结果再压入栈。
  - 扫描结束时，栈顶元素就是最后的结果。

31 5 22 - * 70 +



后缀表达式的求值

```
① float (PList e)   /* e是一个线性表，节点是运算符或运算数 */
② {
③     DateType current,temp1,temp2;
④     PStack s = CreateEmptyStack ();
⑤     for(int i=1; i<=lengthList(e); i++)
⑥     {
⑦         current=retrieve(e,i);
⑧         if ( current 不是运算符)    push(s,current);
⑨         else
⑩          {
⑪              temp1=top(s);  pop(s);   temp2=top(s);  pop(s);
⑫              push(s,  app(temp1,temp2));    // app实现相应的运算
⑬           }
⑭      }
⑮          return Top(s);
⑯ }
```

中缀表达式到前后缀表达式的转换

- 转化步骤：
  - 按照运算符的优先级对所有的运算单位加括号
  - 将运算符移动到对应括号的前面（前缀表达式）或后面（后缀表达式）
  - 去掉括号，得到前缀或后缀表达式
- 示例：31 * (5 - 22) + 70
  - 加括号 ((31*(5-22))+70)
  - 移动运算符

    对于前缀表达式，变成 +(*(31-(5 22))70)

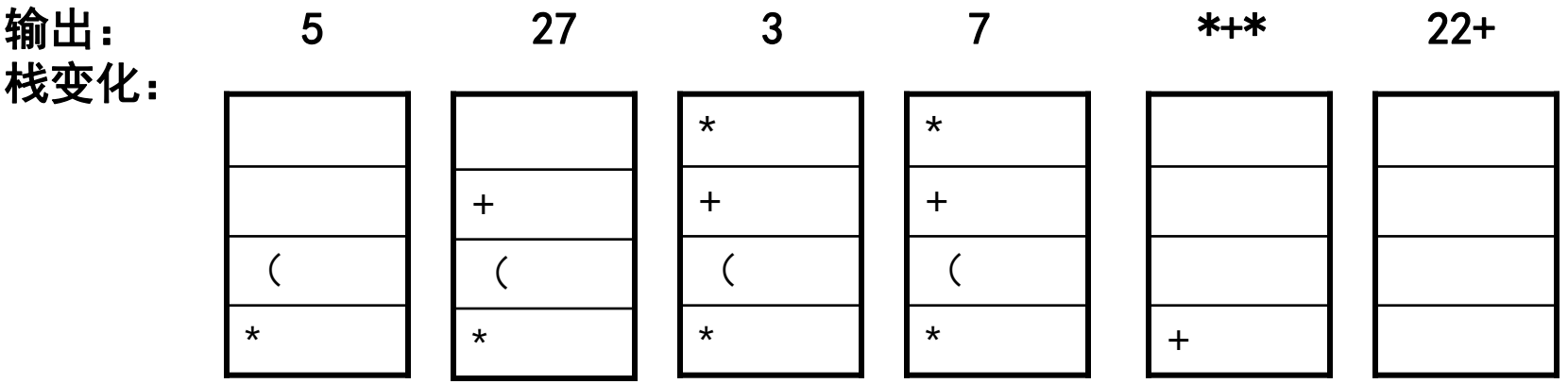
    对于后缀表达式：变成((31(5 22)-)*70)+
  - 去掉括号

    前缀表达式： +*31-5 22 70

    后缀表达式：31 5 22 - * 70 +

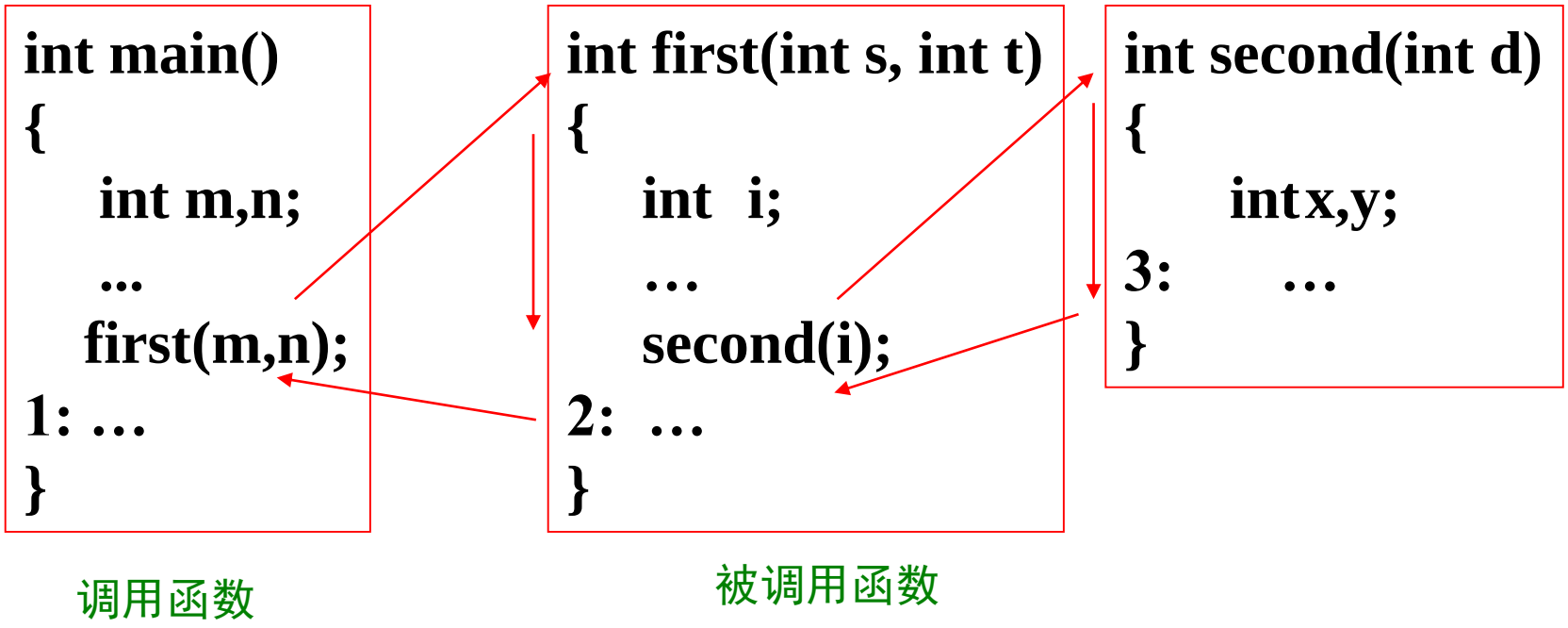
中缀表达式到后缀表达式的转换

- 需要一个存放运算符的栈：
  - 从左往右扫描表达式，为操作数则输出； 遇到运算符，判断：
    - 为 ‘(’立刻入栈；
    - 为运算符，需等到它的两个操作数输出后、并且后面无运算符或后面的运算符优先级低于该运算符，再输出；
    - 为 ‘)’ 将栈中这对括号之间的操作符依次弹出并输出，最后弹出‘(’
- 如：5*(27+3*7)+22 ＝〉5 27 3 7 * + * 22 +



函数调用的过程

多个函数嵌套调用时，按照“后调用先返回”的原则进行，如下所示：



函数调用的过程

- 调用前：
  - ① 调用函数将实参、返回地址传递给被调用函数
  - ② 为被调用函数分配必要的数据区（存放局部变量、实参、返回地址等），接收调用函数传送来的调用信息
  - ③ 将控制转移到被调用函数入口。

- 调用后：
  - ① 传送返回信息，如被调用函数的计算结果。
  - ② 释放被调用函数的数据区
  - ③ 把控制转移到调用函数中

递归的概念-求fact

- 函数fact(n)中又调用了函数fact，这种函数自己调用自己的作法称为递归调用。
- 包含直接或间接递归调用的函数都称为递归函数

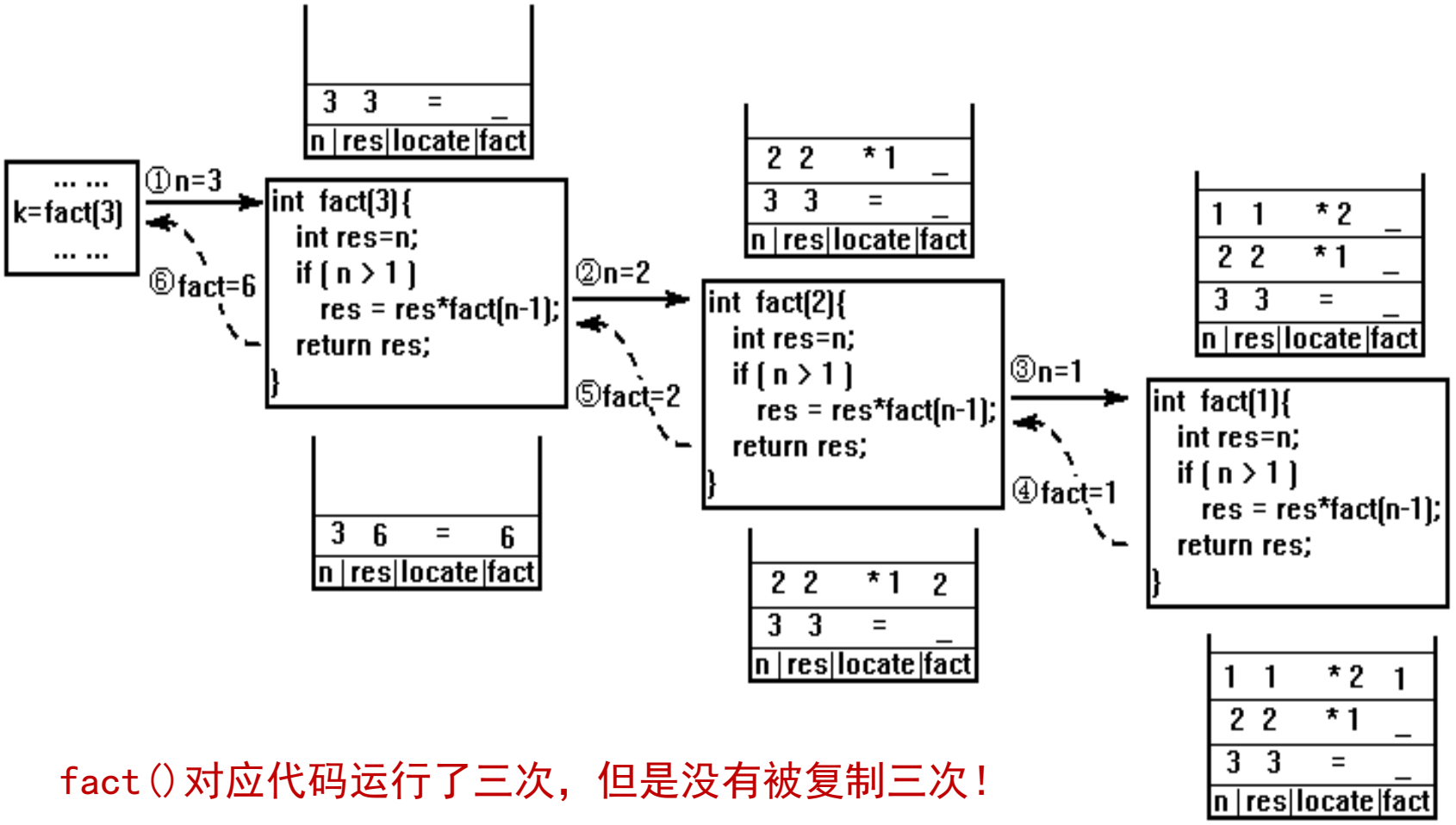
递归实现：

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

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

递归的概念

- 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。



fact()对应代码运行了三次，但是没有被复制三次！

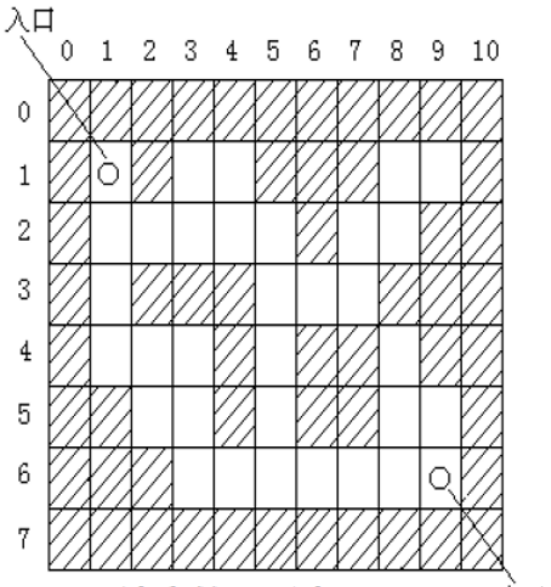
阶乘的非递归计算

```
① int  nfact( int n )
② {
③         int res;
④         PSeqStack  st;                  /* 使用顺序存储结构实现的栈 */
⑤         st = createEmptyStack( );

⑥         while (n>0)                     /* 按照调用次序，压栈 */
⑦         {          push_seq(st,n);  n = n – 1;          }
⑧
⑨         res = 1;                        /* 按照调用的反次序，退栈 */
⑩         while (! isEmptyStack_seq (st))
⑪         {          res = res * top_seq(st);  pop_seq(st);       }
⑫
⑬         free(st);
⑭         return ( res );
⑮ }
```

迷宫问题

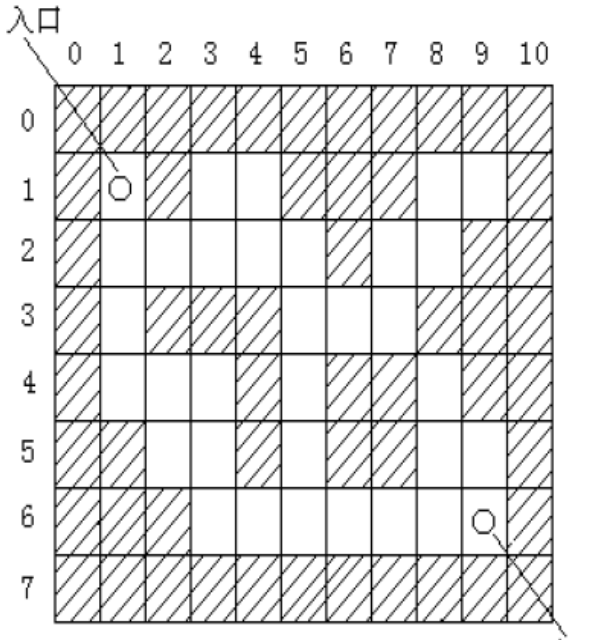
- 在迷宫中求从入口到出口的所有路径是一个经典的程序设计问题。
- 问题分析
  - 迷宫可用下图所示的方块来表示，其中每个元素或为通道（以空白方块表示），或为墙（以带阴影的方块表示）。
  - 只能上下左右运动，要求：从入口到出口的一个以空白方块构成的(无环)路径。



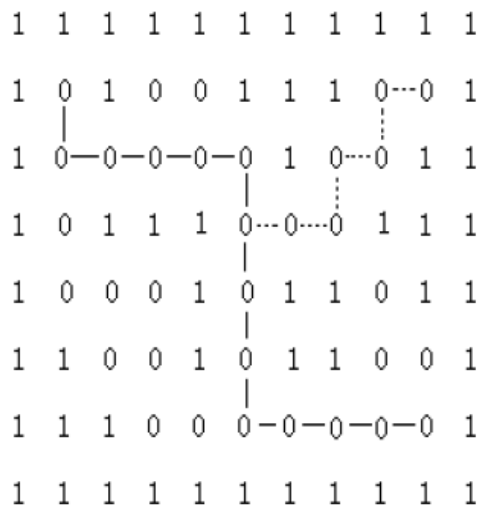
(a) 迷宫的图形表示

程序设计

- 回溯算法：
  - 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。



(a) 迷宫的图形表示



(b) 迷宫的二维数组表示

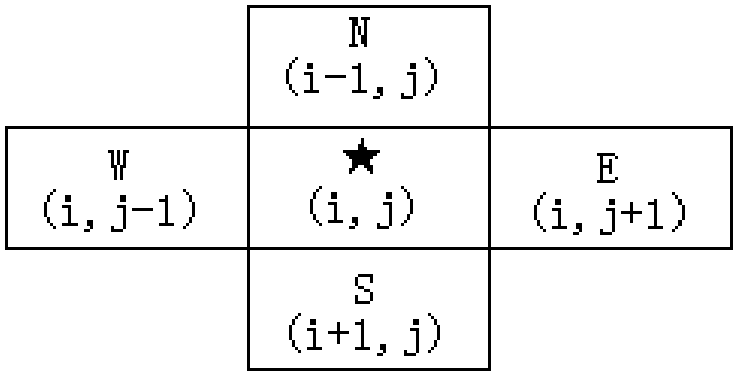
探索-返回对应了入栈-出栈，我们可以利用栈来解决

算法设计

- 数据表示
  - 用二维数组maze[m][n]来表示迷宫，数组中元素为0的表示通道，为1的表示墙。
  - 图中迷宫的入口处为maze[1][1]，出口处为maze[m-2][n-2]；（不同的问题可以自己设定）入口和出口的元素值必为0；
- 回溯算法：
  - 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，沿原路返回上一位置，换一方向再进行探索，直到所有可能的通路都探索到为止。

算法设计

- 任意时刻在迷宫中的位置maze[i][j]时，可能的运动方向有四个：
  - 要进入E方向的位置(g,h)，要根据由该增量值表来修改坐标，即g = i + direction[0][0]; h = j + direction[0][1];
  - 若(i,j)为(3,4)，则其E方向的相邻点为(3,5)。



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

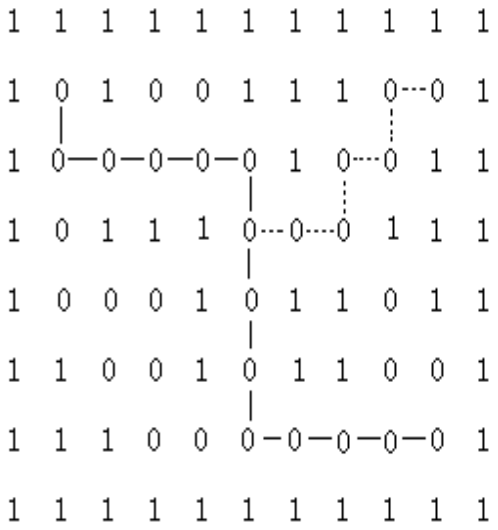
算法设计

- 经过上述设计，求迷宫中一条路径的算法，可以
  - 从入口开始，对每个“当前位置”都从E方向试起，若不能通过，则顺时针试S方向、W方向、N方向。
  - 为避免走回到已进入的点（包括已在当前路径上的点和曾经在当前路径上的点），凡是进入过的点都应做上记号，避免产生死循环。
  - 为记录当前位置及在该位置上所选的方向，设置一个栈，栈中每个元素包括三项，分别记录当前位置的行坐标、列坐标及在该位置上所选的方向。

```
typedef struct NodeMaze { int x, y, d; }  DataType;
```

回溯算法框架

```
mazeFrame( void )  //其算法框架如下＝＝
{
  创建一个（保存探索过程的）空栈;
  把入口位置入栈;
  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的当前位置;
       }
       弹出栈顶元素    // 实现回溯
    }
}
```



小结

- 栈的顺序和链接表示，以及在这两种表示方式下基本运算的实现
- 栈满和栈空的条件及描述

顺序栈和链式栈的比较

- 时间效率
  - 所有操作都只需常数时间
  - 顺序栈和链式栈在时间效率上难分伯仲
- 空间效率
  - 顺序栈须说明一个固定的长度
  - 链式栈的长度可变，但增加结构性开销

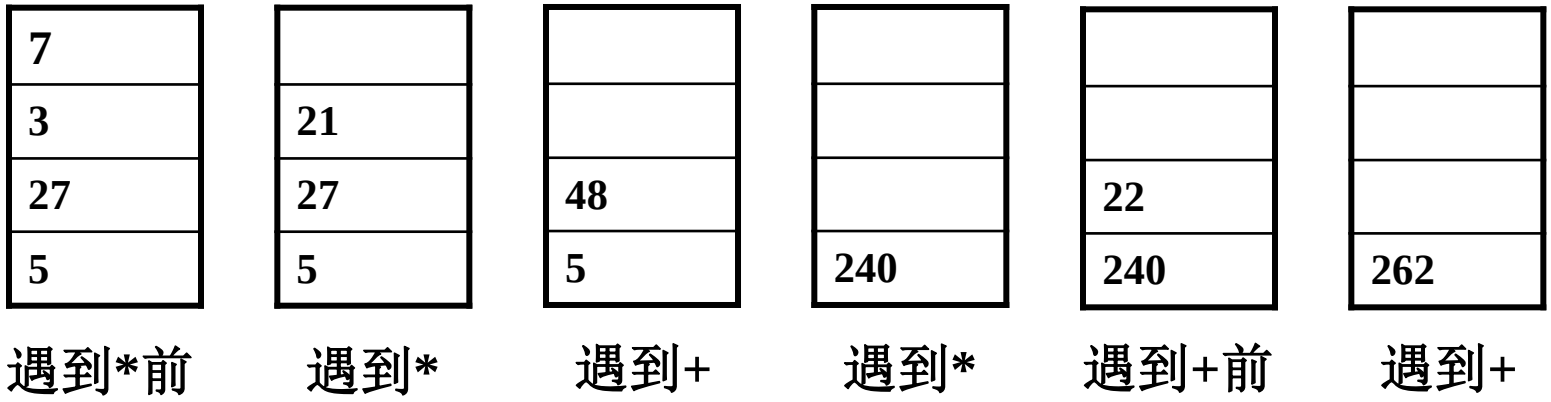
顺序栈和链式栈的比较

- 实际应用中，顺序栈比链式栈用得更广泛
  - 顺序栈容易根据栈顶位置，进行相对位移，快速定位并读取栈的内部元素
  - 顺序栈读取内部元素的时间为O(1)，而链式栈则需要沿着指针链游走，显然慢些，读取第k个元素需要时间为$\mathbf{O}(k)$
- 一般来说，栈不允许“读取内部元素”，只能在栈顶操作。

计算式的求值

- 后缀表达式求值

5 27 3 7 * + * 22 +



如何实现前缀表达式求值？