# 第3章　栈和队列

线性表

栈

队列

# 3.1 栈

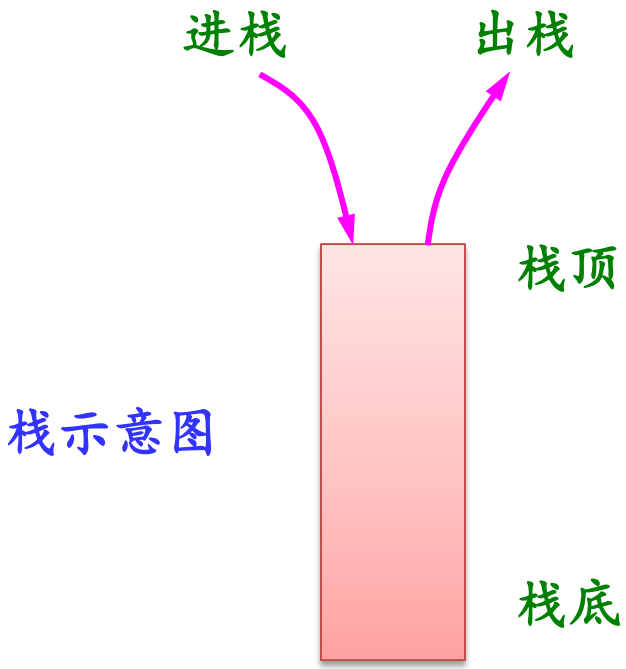
## 3.1.1 栈的定义

栈是一种只能在一端进行插入或删除操作的线性表。


**栈只能选取同一个端点进行插入和删除操作**

# 栈的几个概念

- 允许进行插入、删除操作的一端称为栈顶。
- 表的另一端称为栈底。
- 当栈中没有数据元素时，称为空栈。
- 栈的插入操作通常称为进栈或入栈。
- 栈的删除操作通常称为退栈或出栈。




栈示意图

**栈**: **后进先出**(LIFO, Last In First Out)或**先进后出**(FILO, First In Last Out)结构，最先(晚)到达栈的结点将最晚(先)被删除。

例如：


假设死胡同的宽度
恰好只够正一个人

走进死胡同的5人
要按相反次序退出


死胡同就是一个栈！

【例】设一个栈的输入序列为*a*，*b*，*c*，*d*，则借助一个栈所得到的输出序列不可能是（　）。

A．*c*，*d*，*b*，*a*　　B．*d*，*c*，*b*，*a*

C．*a*，*c*，*d*，*b*　　D．*d*，*a*，*b*，*c*

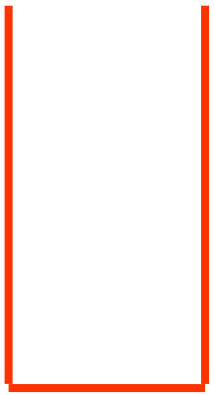
【例】设一个栈的输入序列为*a*，*b*，*c*，*d*，则借助一个栈所得到的输出序列不可能是（　）。

A．*c*，*d*，*b*，*a*　　B．*d*，*c*，*b*，*a*

C．*a*，*c*，*d*，*b*　　D．*d*，*a*，*b*，*c*

选项D是不可能的？

*d*　*c*　*b*　*a*



下一步不可能出栈*a*

栈

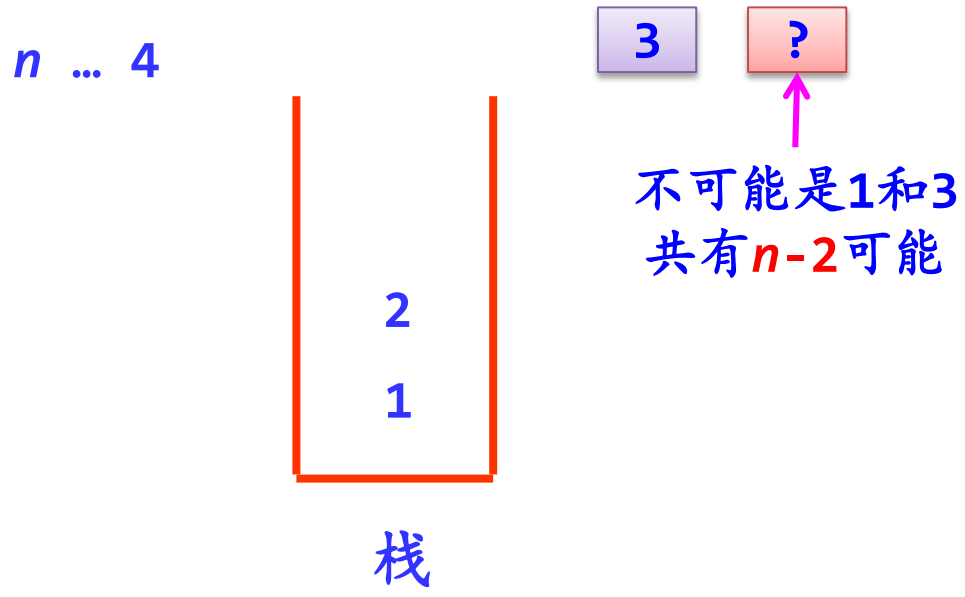
【例】一个栈的入栈序列为1，2，3，…，$n$ ，其出栈序列是$p_1$，$p_2$，$p_3$，…，$p_n$。若$p_1$=3，则$p_2$可能取值的个数是_多少？

A.$n-3$　　B.$n-2$　　C.$n-1$　　D．无法确定

【例】一个栈的入栈序列为1，2，3，…，$n$ ，其出栈序列是$p_1$，$p_2$，$p_3$，…，$p_n$。若$p_1$=3，则$p_2$可能取值的个数是_多少？

A.$n$-3　　B.$n$-2　　C.$n$-1　　D．无法确定

1、2、3进栈，　3出栈的结果：



【例】一个栈的入栈序列为1，2，3，…，$n$ ，其出栈序列是$p_1$，$p_2$，$p_3$，…，$p_n$。若$p_2$=3，则$p_3$可能取值的个数是（　）多少？

A.$n$-3　　B.$n$-2　　C.$n$-1　　D．无法确定

**解：** $p_3$可以取1：1进，2进，2出，3进，3出（$p_2$），1出($p_3$)，…。

$p_3$可以取2：1进，1出，2进，3进，3出($p_2$)，2出($p_3$)，…。

$p_3$可以取4：1进，1出，2进，3进，3出($p_2$)，4进，4出($p_3$)，…。

$p_3$可以取5：1进，1出，2进，3进，3出($p_2$)，4进，5进，5出($p_3$)，…。

…

$p_3$可以取除了3外的任何值。答案为C。

# 栈抽象数据类型＝逻辑结构＋基本运算（运算描述）

栈的几种基本运算如下：

❶ InitStack(&s)：初始化栈。构造一个空栈s。

❷ DestroyStack(&s)：销毁栈。释放栈s占用的存储空间。

❸ StackEmpty(s)：判断栈是否为空：若栈s为空，则返回真；否则返回假。

❹ Push(&S, e)：进栈。将元素e插入到栈s中作为栈顶元素。

❺ Pop(&s, &e)：出栈。从栈s中退出栈顶元素，并将其值赋给e。

❻ GetTop(s, &e)：取栈顶元素。返回当前的栈顶元素，并将其值赋给e。

顺序栈

链栈

# 3.1.2 栈的顺序存储结构

假设栈的元素个数最大不超过正整数MaxSize，所有的元素都具有同一数据类型ElemType，则可用下列方式来定义顺序栈类型SqStack：

```
typedef struct
{   ElemType data[MaxSize];
    int top;                //栈顶指针
}   SqStack;
```

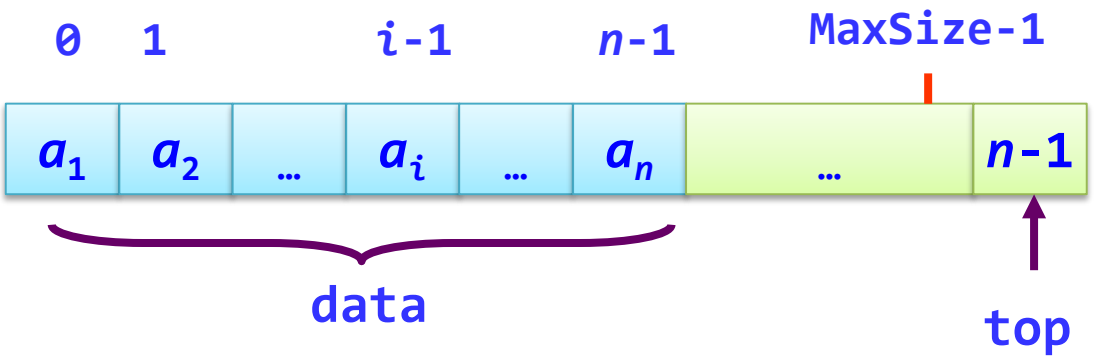
逻辑结构

栈

$(a_1, a_2, \ldots, a_i, \ldots a_n)$

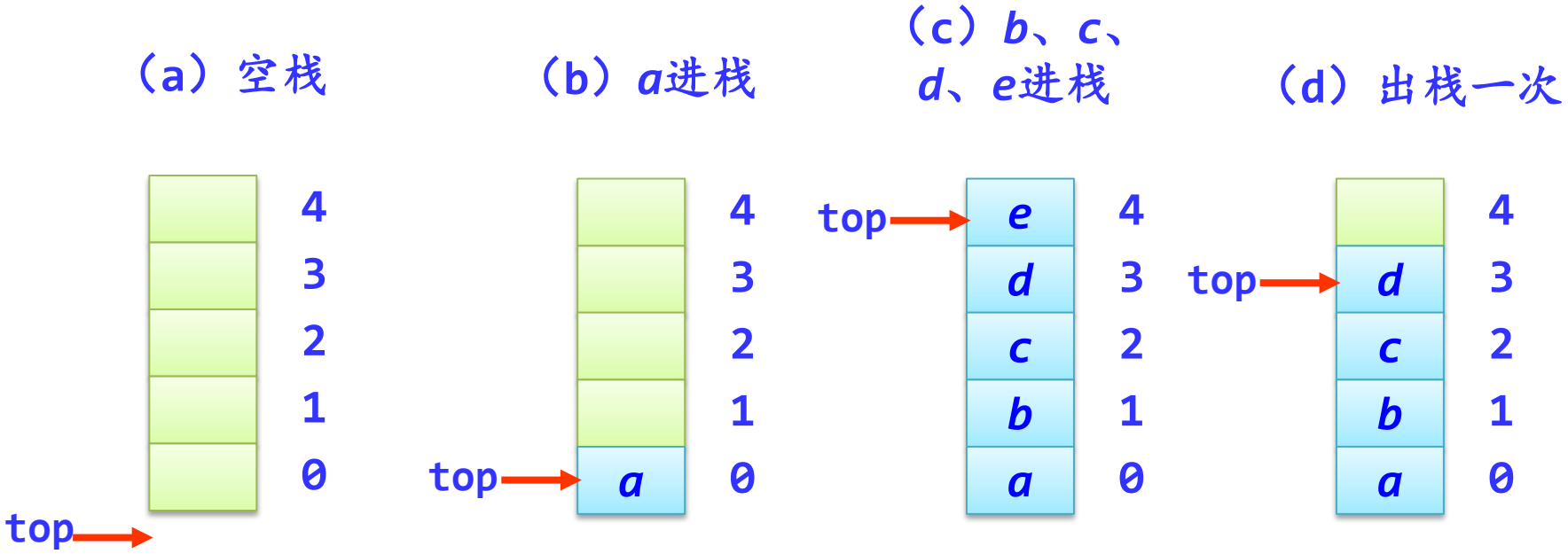
直接映射

存储结构




顺序栈的示意图

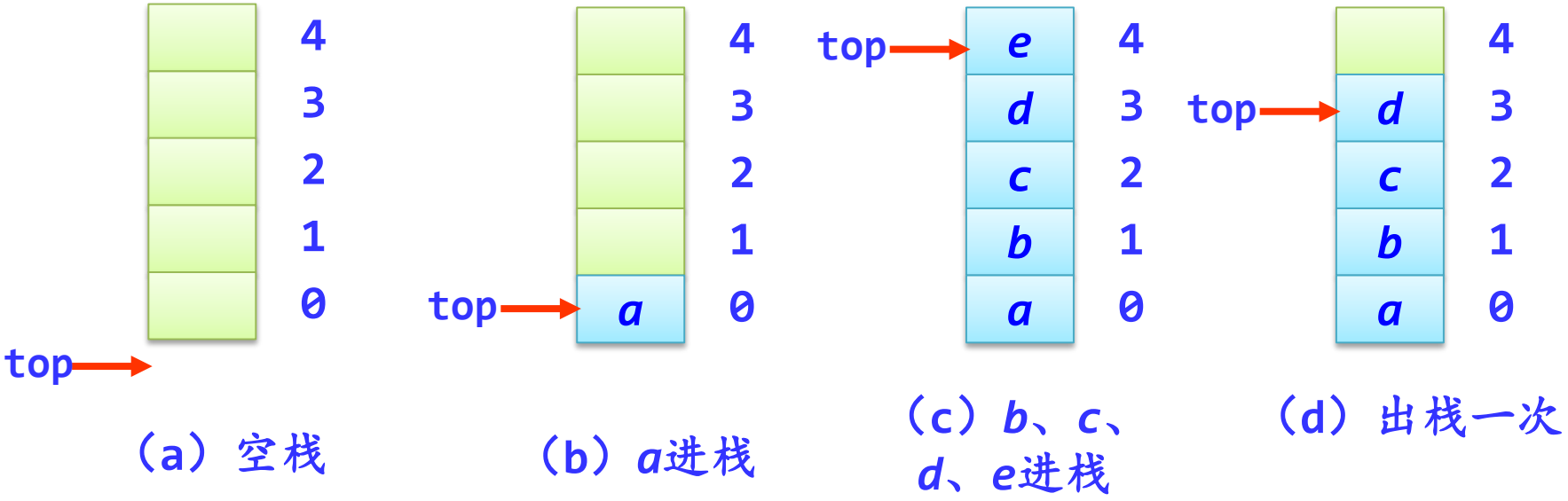
例如：MaxSize=5




总结：

- 约定top总是指向栈顶元素，初始值为-1
- 当top=MaxSize-1时不能再进栈—栈满
- 进栈时top增1，出栈时top减1

# 顺序栈的各种状态




（a）空栈　（b）a进栈　（c）b、c、d、e进栈　（d）出栈一次

顺序栈4要素：

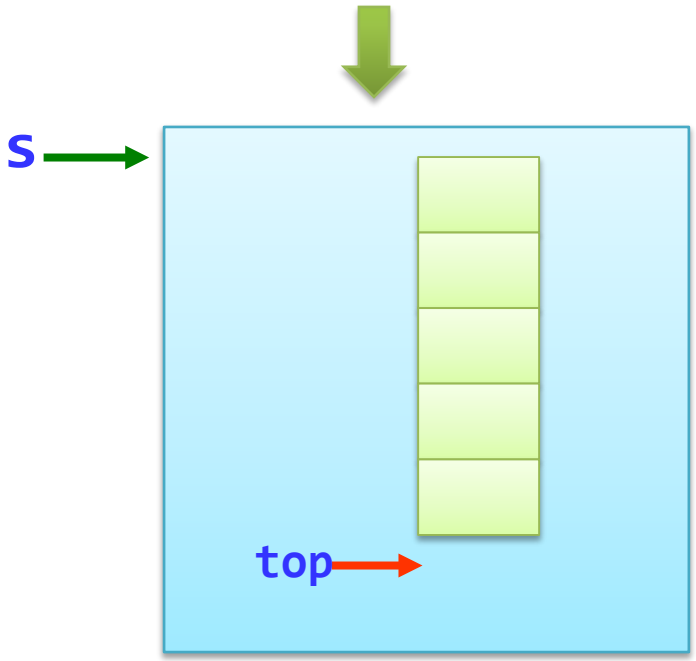
- 栈空条件：top=-1
- 栈满条件：top=MaxSize-1
- 进栈e操作：top++；将e放在top处
- 退栈操作：从top处取出元素e；top--；

在顺序栈中实现栈的基本运算算法。

## (1) 初始化栈InitStack(&s)

建立一个新的空栈s，实际上是将栈顶指针指向-1即可。

```
void InitStack(SqStack *&s)
{   s=(SqStack *)malloc(sizeof(SqStack));
    s->top=-1;
}
```




**注意：** s为栈指针，top为s所指栈的栈顶指针

## (2) 销毁栈DestroyStack(&s)

释放栈s占用的存储空间。

```
void  DestroyStack(SqStack *&s)
{
    free(s);
}
```

## (3) 判断栈是否为空StackEmpty(s)

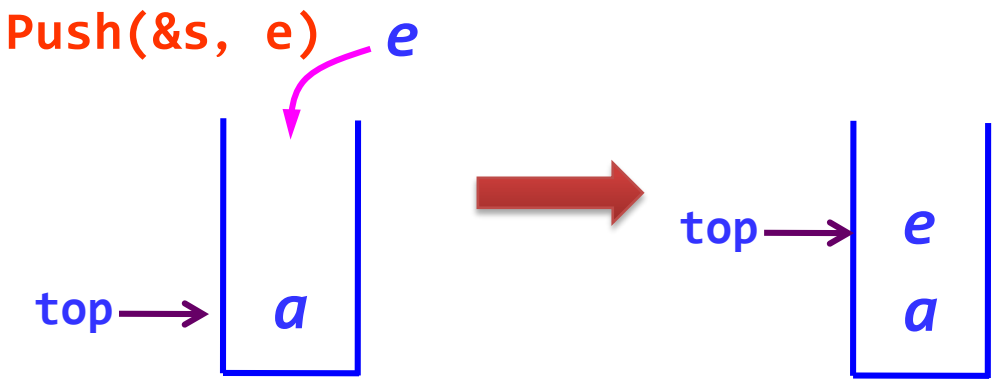
栈S为空的条件是s->top==-1。

```
bool StackEmpty(SqStack *s)
{
    return(s->top==-1);
}
```

## (4) 进栈Push(&s, e)

在栈不满的条件下，先将栈指针增1，然后在该位置上插入元素e。

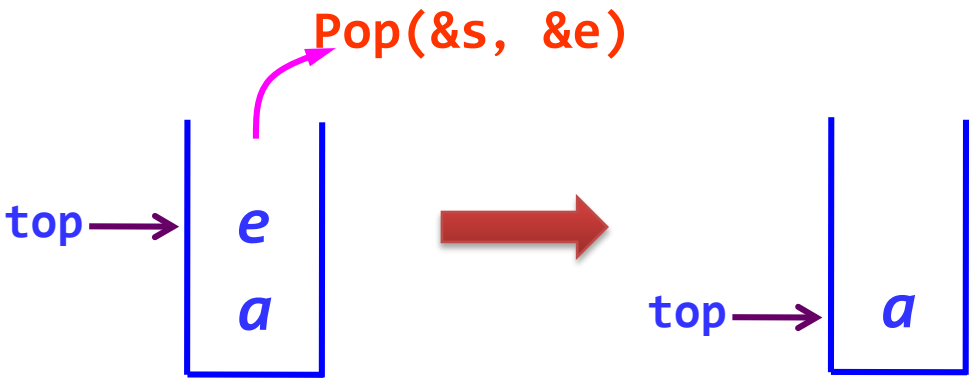
```
bool Push(SqStack *&s, ElemType e)
{   if (s->top==MaxSize-1)      //栈满的情况，即栈上溢出
        return false;
    s->top++;                   //栈顶指针增1
    s->data[s->top]=e;          //元素e放在栈顶指针处
    return true;
}
```



## (5) 出栈Pop(&s, &e)

在栈不为空的条件下，先将栈顶元素赋给e，然后将栈指针减1。

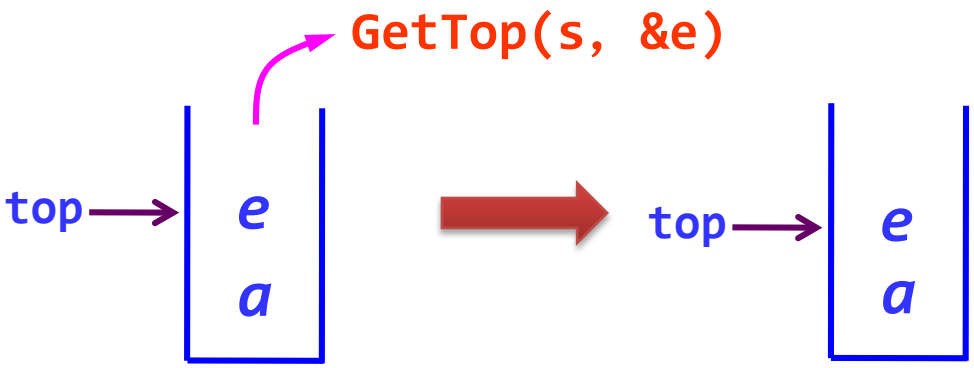
```
bool Pop(SqStack *&s, ElemType &e)
{   if (s->top==-1)     //栈为空的情况，即栈下溢出
        return false;
    e=s->data[s->top]; //取栈顶指针元素的元素
    s->top--;          //栈顶指针减1
    return true;
}
```



## (6) 取栈顶元素GetTop(s, &e)

在栈不为空的条件下，将栈顶元素赋给e。

```
bool GetTop(SqStack *s, ElemType &e)
{
    if (s->top==-1)        //栈为空的情况，即栈下溢出
      return false;
    e=s->data[s->top];     //取栈顶指针元素的元素
    return true;
}
```



【例】 设计一个算法利用顺序栈判断一个字符串是否是对称串。所谓对称串是指从左向右读和从右向左读的序列相同。

## 算法设计思路

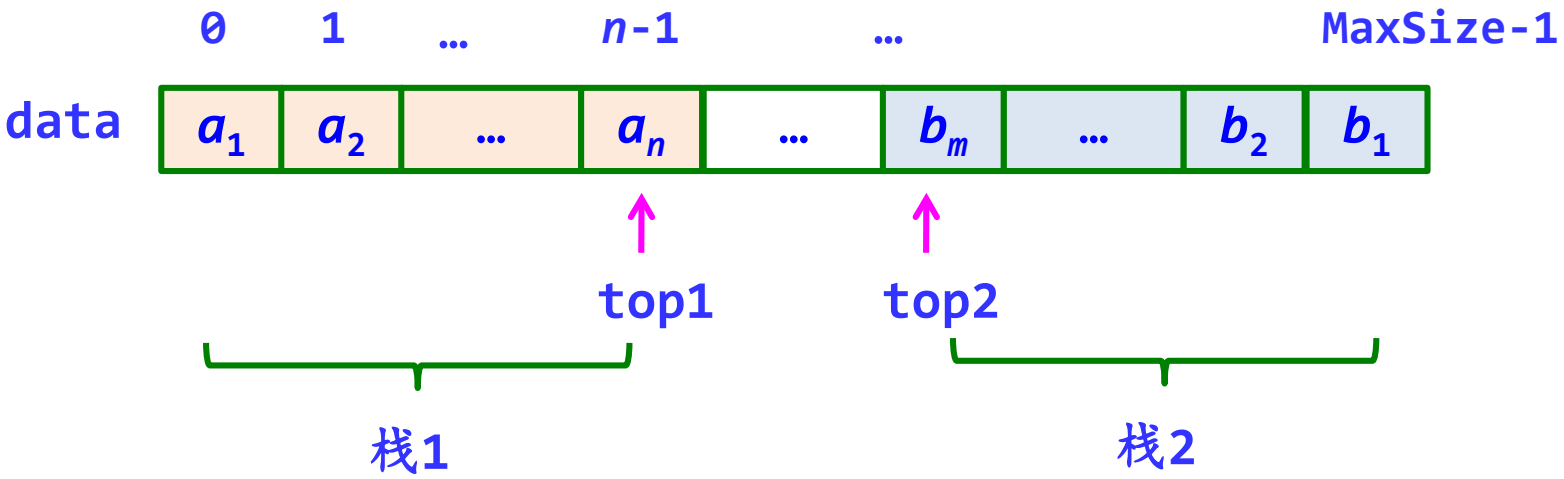
字符串str的所有元素依次进栈，产生的出栈序列正好与str的顺序相反。

```
bool symmetry(ElemType str[])
{  int i;  ElemType e; SqStack *st;
   InitStack(st);                          //初始化栈

   for (i=0;str[i]!='\0';i++)              //将串所有元素进栈
      Push(st, str[i]);                    //元素进栈

   for (i=0;str[i]!='\0';i++)
   {  Pop(st, e);                          //退栈元素e
      if (str[i]!=e)                       //若e与当前串元素不同则不是对称串
      {  DestroyStack(st);                 //销毁栈
         return false;
      }
   }

   DestroyStack(st);                       //销毁栈
   return true;
}
```

str的所有元素依次进栈

判断正反序是否相同

若需要用到两个相同类型的栈，可用一个数组data[0..MaxSize-1]来实现这两个栈，这称为**共享栈**。

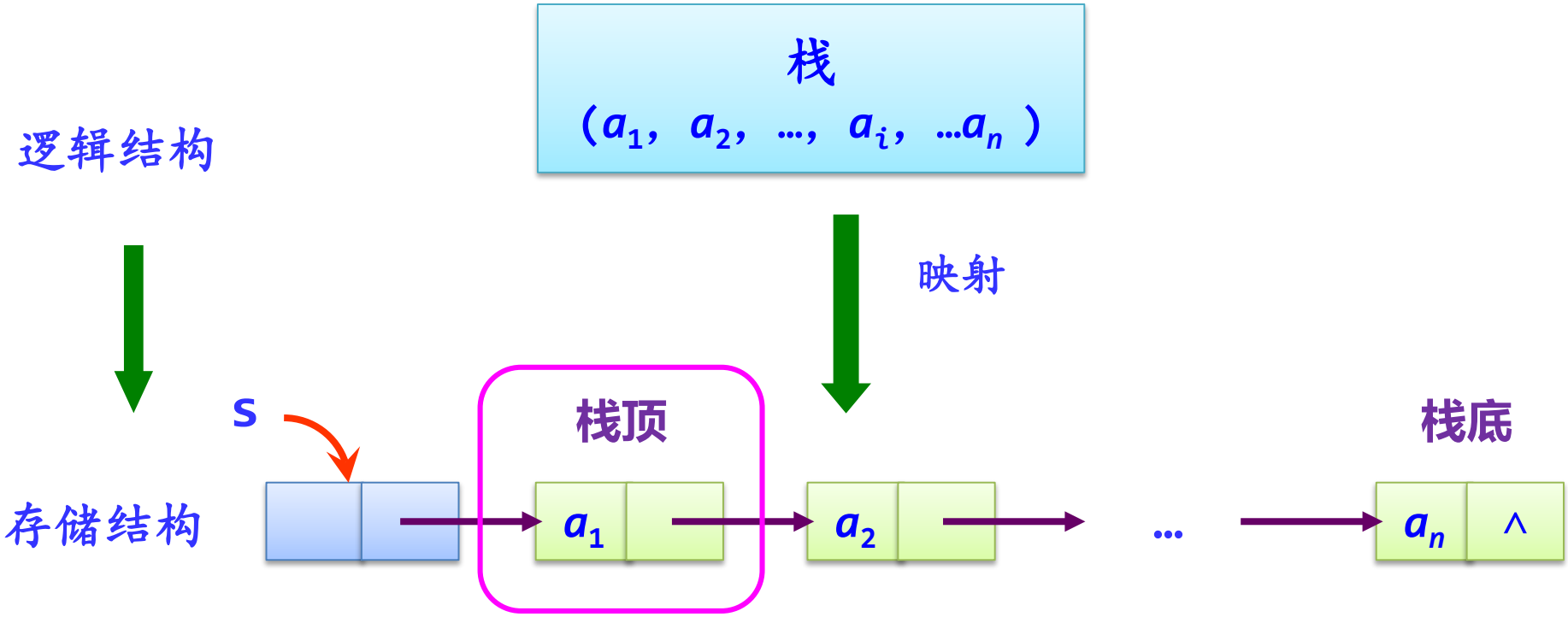



**共享栈类型：**

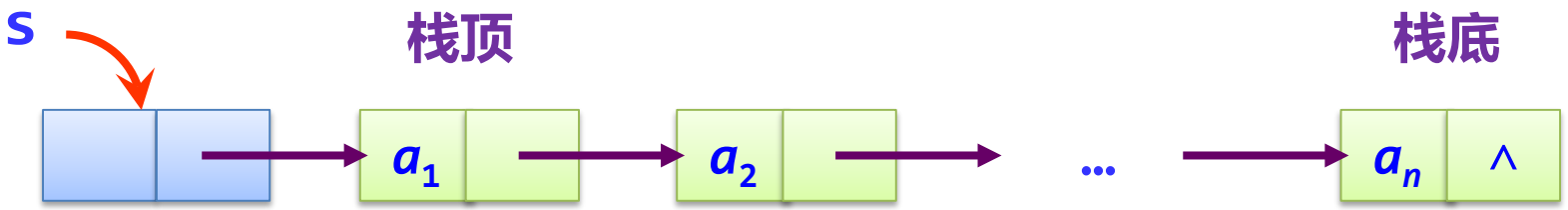
```
typedef struct
{   ElemType data[MaxSize];    //存放共享栈中元素
    int top1, top2;            //两个栈的栈顶指针
} DStack;
```

# 3.1.3 栈的链式存储结构

采用链表存储的栈称为**链栈**，这里采用带头结点的单链表实现。




一个链栈的示意图




链栈的4要素：

- 栈空条件：s->next=NULL
- 栈满条件：不考虑
- 进栈e操作：将存放e的结点插入到头结点之后
- 退栈操作：取出头结点之后结点的元素并删除之

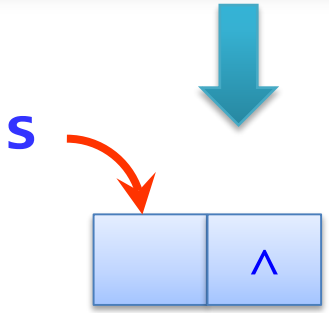
链栈中数据结点的类型LinkStNode声明如下：

```
typedef struct linknode
{    ElemType data;              //数据域
     struct linknode *next;      //指针域
}  LinkStNode;
```

## (1) 初始化栈initStack(&s)

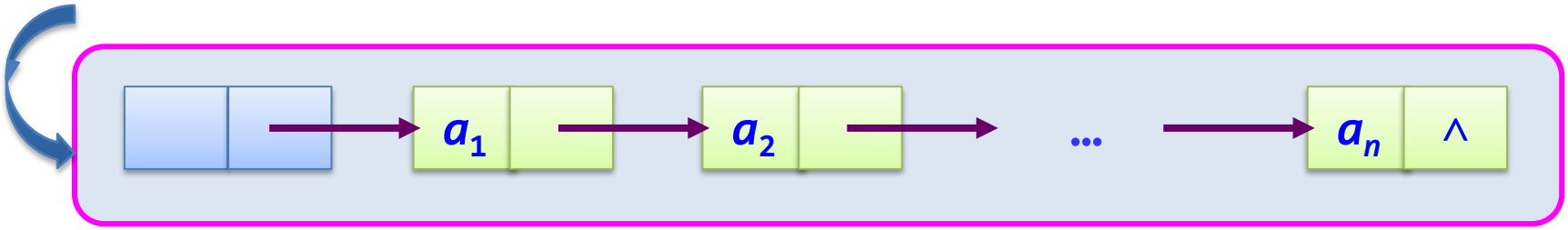
建立一个空栈s。实际上是创建链栈的头结点，并将其next域置为NULL。

```
void InitStack(LinkStNode *&s)
{  s=(LinkStNode *)malloc(sizeof(LinkStNode));
   s->next=NULL;
}
```



## (2) 销毁栈DestroyStack(&s)

释放栈s占用的全部存储空间。

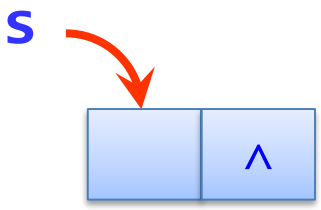



```
void DestroyStack(LinkStNode *&s)
{   LinkStNode *p=s, *q=s->next;
    while (q!=NULL)
    {   free(p);
        p=q;
        q=p->next;
    }
    free(p);   //此时p指向尾结点，释放其空间
}
```

## (3) 判断栈是否为空StackEmpty(s)

栈S为空的条件是s->next==NULL，即单链表中没有数据结点。

```
bool StackEmpty(LinkStNode *s)
{
    return(s->next==NULL);
}
```

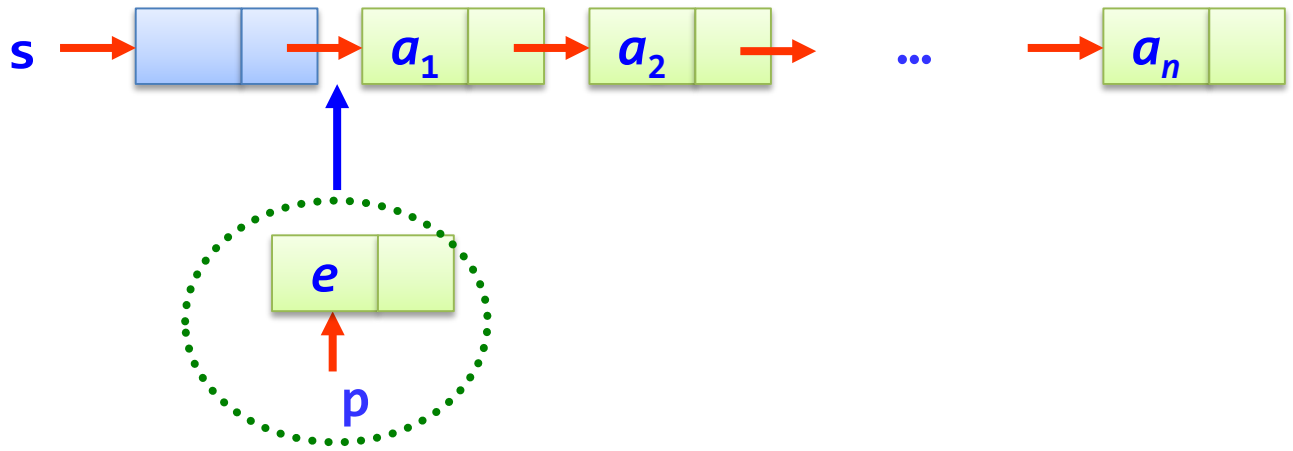



空栈的情况

# (4) 进栈Push(&s, e)

将新数据结点插入到头结点之后。

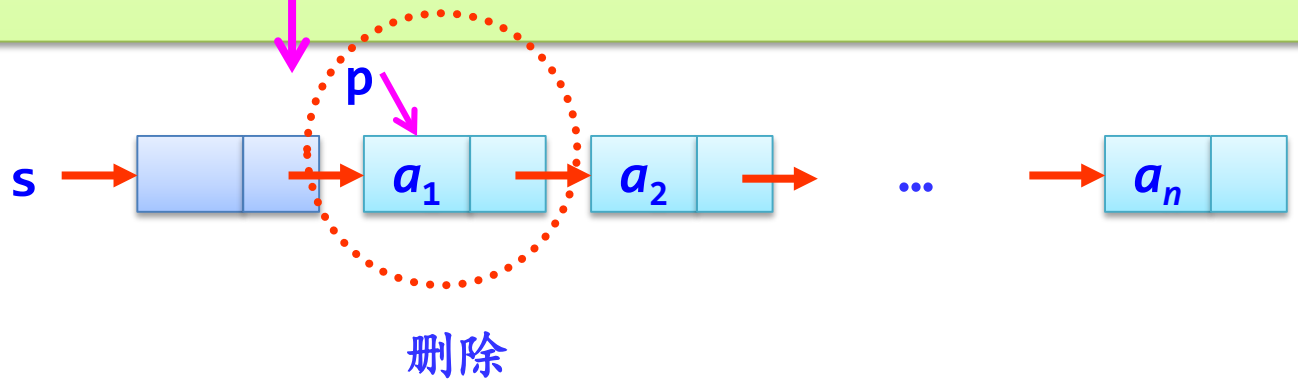
```
void Push(LinkStNode *&s, ElemType e)
{  LinkStNode *p;
   p=(LinkStNode *)malloc(sizeof(LinkStNode));
   p->data=e;              //新建元素e对应的结点p
   p->next=s->next;        //插入p结点作为开始结点
   s->next=p;
}
```



## (5) 出栈Pop(&s, &e)

在栈不为空的条件下，将头结点后继数据结点的数据域赋给e，然后将其删除。

```
bool Pop(LinkStNode *&s, ElemType &e)
{  LinkStNode *p;
   if (s->next==NULL)          //栈空的情况
     return false;
   p=s->next;                  //p指向开始结点
   e=p->data;
   s->next=p->next;            //删除p结点
   free(p);                    //释放p结点
   return true;
}
```

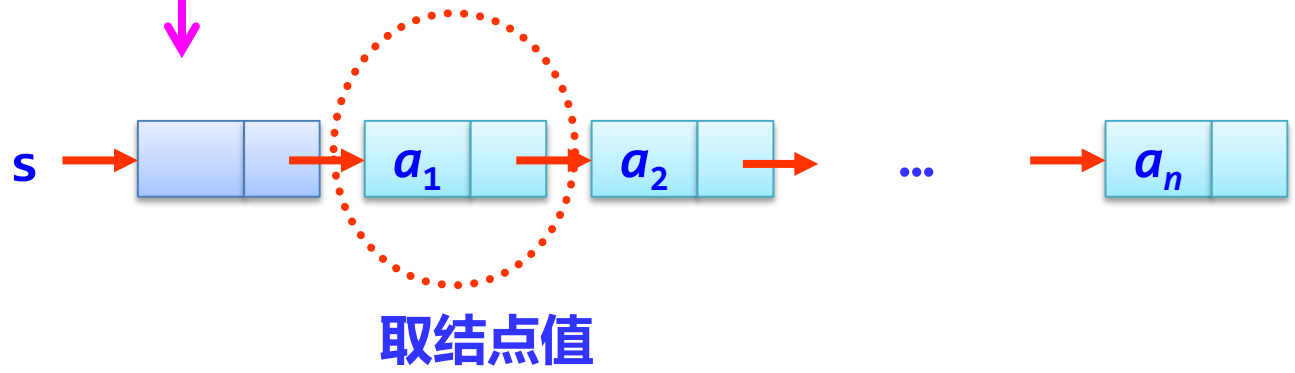



删除

## (6) 取栈顶元素GetTop(s, e)

在栈不为空的条件下，将头结点后继数据结点的数据域赋给e。

```
bool GetTop(LinkStNode *s, ElemType &e)
{  if (s->next==NULL)          //栈空的情况
     return false;
   e=s->next->data;
   return true;
}
```



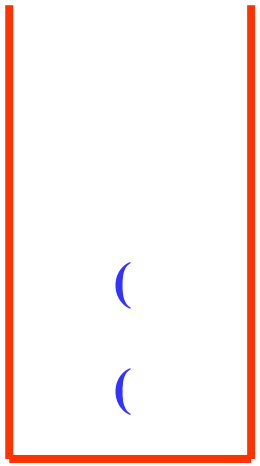
【例】编写一个算法判断输入的表达式中括号是否配对（假设只含有左、右圆括号）。

## 算法设计思路

一个表达式中的左右括号是按最近位置配对的。所以利用一个栈来进行求解。这里采用链栈。

# 表达式括号不配对情况的演示

例如：exp=“( ( ) ) )”




① ‘( ‘进栈

② ‘( ‘进栈

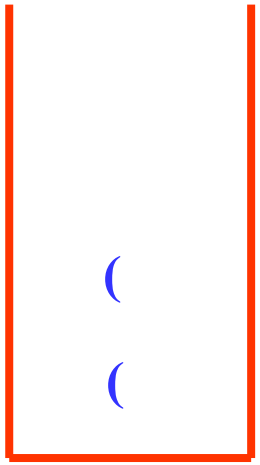
③ 遇到’ )’，栈顶为’ ( ‘，退栈

④ 遇到’ )’，栈顶为’ ( ‘，退栈

⑤ 遇到’ )’，栈为空，返回false

# 表达式括号配对情况的演示

例如：exp=“( ( ) )”




① ‘(‘进栈

② ‘(‘进栈

③ 遇到’)’，栈顶为’(‘，退栈

④ 遇到’)’，栈顶为’(‘，退栈

栈空且exp扫描完，返回true

```
bool Match(char exp[], int n)
{  int i=0; char e;
   bool match=true;
   LinkStNode *st;
   InitStack(st);                    //初始化栈
   while (i<n && match)              //扫描exp中所有字符
   {
      if (exp[i]=='(')
          Push(st, exp[i]);
```

配对时为true;
否则为false

链栈指针

遇到任何左括号都进栈

```
    else if (exp[i]==')')           //当前字符为右括号
    {  if (GetTop(st, e)==true)
       {  if (e!='(')               //栈顶元素不为'('时不匹配
             match=false;
          else
             Pop(st, e);            //将栈顶元素出栈
        }
        else  match=false;          //无法取栈顶元素时不匹配
    }

    i++;                            //继续处理其他字符
}
```

遇到右括号判断是否匹配

```
    if (!StackEmpty(st))
        match=false;

    DestroyStack(st);           //销毁栈
    return match;
}
```

栈不空时表示不匹配

**注意：只有在表达式扫描完毕且栈空时返回true。**

# 3.1.4 栈的应用

- 如果后放入的数据先处理，一般使用栈。

## 1．简单表达式求值

### 问题描述

这里限定的简单表达式求值问题是：用户输入一个包含“+”、“-”、“*”、“/”、正整数和圆括号的合法算术表达式，计算该表达式的运算结果。

## 数据组织

简单表达式采用字符数组*exp*表示，其中只含有“+”、“-”、“*”、“/”、正整数和圆括号。

为了方便，假设该表达式都是合法的算术表达式，例如，*exp*=“1+2*(4+12)”；

在设计相关算法中用到栈，这里采用顺序栈存储结构。

中缀表达式的运算规则：“先乘除，后加减，从左到右计算，先括号内，后括号外”。

中缀表达式不仅要依赖运算符优先级，而且还要处理括号。

算术表达式的另一种形式是后缀表达式或逆波兰表达式，就是在算术表达式中，运算符在操作数的后面，如1+2*3的后缀表达式为1 2 3 * +。

**后缀表达式：**

- 已考虑了运算符的优先级。
- 没有括号。
- 只有操作数和运算符，而且越放在前面的运算符来越优先执行。

在算术表达式中，如果运算符在操作数的前面，称为前缀表达式，如1+2*3的前缀表达式为+ 1 * 2 3。

- 中缀表达式：1 + 2 * 3
- 后缀表达式：1 2 3 * +
- 前缀表达式：+ 1 * 2 3

运算数的相对次序相同

中缀表达式的求值过程：

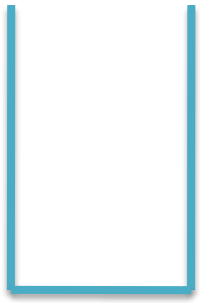
- 将中缀算术表达式转换成后缀表达式。
- 对该后缀表达式求值。

## （1）将算术表达式转换成后缀表达式

*exp* ⇒ *postexp*

扫描*exp*的所有字符：

- 数字字符直接放在*postexp*中
- 运算符通过一个栈来处理优先级

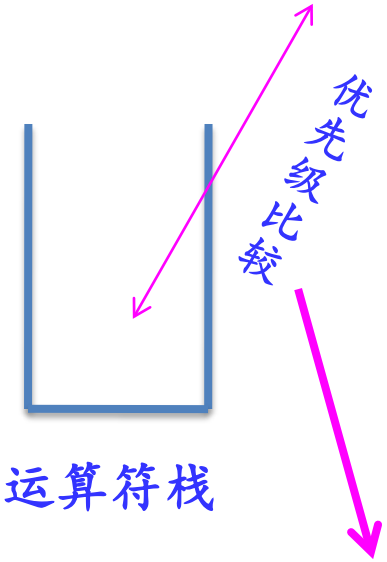


运算符栈

*exp* ⇒ *postexp*

## 情况1（没有括号）

*exp*= “1 + 2 + 3”




*postexp*:

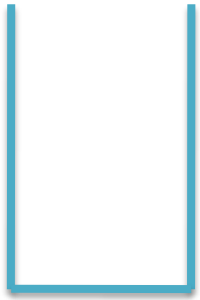
“1+2+3” ⇒ “1 2 + 3 +”

- 先进栈的先退栈即先执行：
只有大于栈顶优先级才能直接进栈
- *exp*扫描完毕，所有运算符退栈

*exp* ⇒ *postexp*

## 情况2（带有括号）

*exp*= “2 * ( 1 + 3 ) - 4 ”



运算符栈

*postexp*:

*postexp*= “2 1 3 + * 4 -”

- 开始时，任何运算符都进栈
- (：一个子表达式开始，进栈
- 栈顶为(：任何运算符进栈
- )：退栈到（
- 只有大于栈顶的优先级，才进栈；否则退栈

```
while (从exp读取字符ch, ch!='\0')
{  ch为数字: 将后续的所有数字均依次存放到postexp中,
           并以字符'#'标志数值串结束;
   ch为左括号'(': 将此括号进栈到Optr中;
   ch为右括号')': 将Optr中出栈时遇到的第一个左括号'('以前的运算符依次出
           栈并存放到postexp中, 然后将左括号'('出栈;
   ch为其他运算符:
       if  (栈空或者栈顶运算符为'(')  直接将ch进栈;
       else if (ch的优先级高于栈顶运算符的优先级)
            直接将ch进栈;
       else
            依次出栈并存入到postexp中, 直到栈顶运算符优先级小于ch的
           优先级, 然后将ch进栈;
}
```

若exp扫描完毕, 则将Optr中所有运算符依次出栈并存放到postexp中。

中缀表达式“(56-20)/(4+2)” ⇨ 后缀表达式“56#20#-4#2#+/”

| 操　作 | postexp | Optr栈（栈底→栈顶） |
|---|---|---|
| '(': 将此括号进栈 | | ( |
| ch为数字: 将56#存入postexp中 | 56# | ( |
| '-': 直接将ch进栈 | 56# | (- |
| ch为数字: 将20#存入postexp中 | 56#20# | (- |
| ')': 将栈中'('之前的运算符'-'出栈并存入postexp中，然后将'('出栈 | 56#20#- | |
| '/': 将ch进栈 | 56#20#- | / |

| 操　作 | postexp | Optr栈（栈底→栈顶） |
| --- | --- | --- |
| '('：将此括号进栈 | 56#20#- | /( |
| ch为数字：将4#存入postexp中 | 56#20#-4# | /( |
| '+'：栈顶运算符为'('，直接将ch进栈 | 56#20#-4# | /(+ |
| ch为数字：将2#存入postexp中 | 56#20#-4#2# | /(+ |
| ')'：将栈中'('之前的运算符'+'出栈并存入postexp中，然后将'('出栈 | 56#20#-4#2#+ | / |
| str扫描完毕，将Optr栈中的所有运算符依次出栈并存入postexp中 | 56#20#-4#2#+/ | |

**算法：**将算术表达式*exp*转换成后缀表达式*postexp*。

```
void trans(char *exp, char postexp[])
{  char e;
   SqStack *Optr;                          //定义运算符栈指针
   InitStack(Optr);                        //初始化运算符栈
   int i=0;                                //i作为postexp的下标
   while (*exp!='\0')                      //exp表达式未扫描完时循环
   {   switch(*exp)
       {
       case '(':                           //判定为左括号
                Push(Optr, '(');           //左括号进栈
                exp++;                     //继续扫描其他字符
                break;
```

```
case ')':                          //判定为右括号
    Pop(Optr, e);                  //出栈元素e
    while (e!='(')                 //不为'('时循环
    {  postexp[i++]=e;             //将e存放到postexp中
       Pop(Optr, e);               //继续出栈元素e
    }
    exp++;                         //继续扫描其他字符
    break;
```

```
case '+':                                   //判定为加或减号
case '-':
     while (!StackEmpty(Optr))              //栈不空循环
     {  GetTop(Optr, e);                    //取栈顶元素e
        if (e!='(')                         //e不是'('
        {  postexp[i++]=e;                  //将e存放到postexp中
           Pop(Optr, e);                    //出栈元素e
        }
        else                                //e是'(时退出循环
            break;
     }
     Push(Optr, *exp);                      //将'+'或'-'进栈
     exp++;                                 //继续扫描其他字符
     break;
```

```
case '*':                                   //判定为'*'或'/'号
case '/':
      while (!StackEmpty(Optr))             //栈不空循环
      {  GetTop(Optr, e);                   //取栈顶元素e
         if (e=='*' || e=='/')
         {  postexp[i++]=e;  //将e存放到postexp中
            Pop(Optr, e);    //出栈元素e
         }
         else                               //e为非'*'或'/'运算符时退出循环
             break;
      }
      Push(Optr, *exp);                     //将'*'或'/'进栈
      exp++;                                //继续扫描其他字符
      break;
```

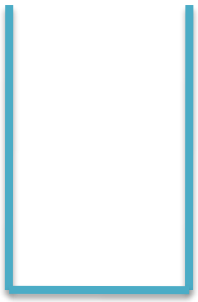
```
        default:                                    //处理数字字符
            while (*exp>='0' && *exp<='9')          //判定为数字字符
            {  postexp[i++]=*exp;
               exp++;
            }
            postexp[i++]='#';          //用#标识一个数值串结束
        }
    }
    while (!StackEmpty(Optr))          //此时exp扫描完毕，栈不空时循环
    {  Pop(Optr, e);                   //出栈元素e
       postexp[i++]=e;                 //将e存放到postexp中
    }
    postexp[i]='\0';                   //给postexp表达式添加结束标识
    DestroyStack(Optr);                //销毁栈
}
```

## （2）后缀表达式求值

*postexp* ⇒ 值

扫描*postexp*的所有字符：

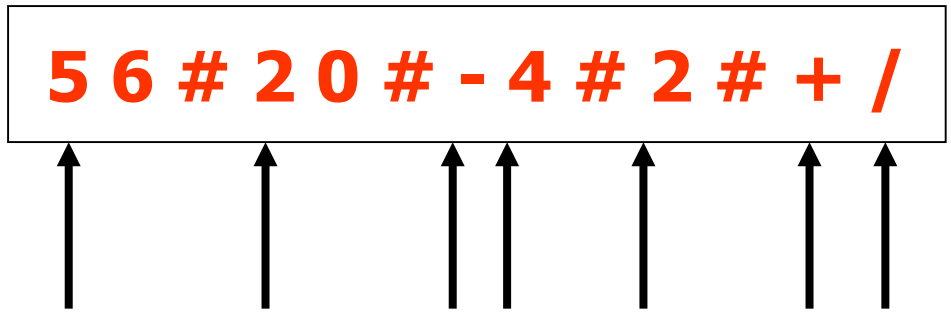
- **数字字符：转换为数值并进栈**
- **运算符：退栈两个操作数，计算，将结果进栈**



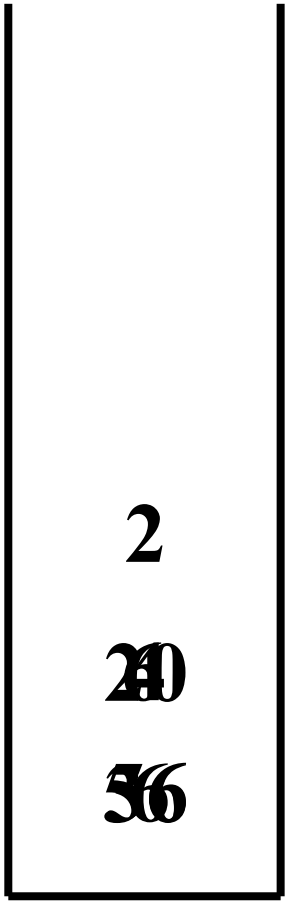
操作数栈

运算数栈

postexp




≠



```
while (从postexp读取字符ch, ch!='\0')
{   ch为'+': 从Opnd栈中出栈两个数值a和b, 计算c=b+a;将c进栈;
    ch为'-': 从Opnd栈中出栈两个数值a和b, 计算c=b-a;将c进栈;
    ch为'*': 从Opnd栈中出栈两个数值a和b, 计算c=b*a;将c进栈;
    ch为'/': 从Opnd栈中出栈两个数值a和b, 若a不零, 计算c=b/a;将c进栈;
    ch为数字字符: 将连续的数字串转换成数值d, 将d进栈;
}
返回Opnd栈的栈顶操作数即后缀表达式的值;
```

后缀表达式“56#20#-4#2#+/”的求值过程。

| 操 作 | Opnd栈(栈底→栈顶) |
| --- | --- |
| 56#：将56进栈 | 56 |
| 20#：将20进栈 | 56，20 |
| '-'：出栈两次，将56-20=36进栈 | 36 |
| 4#：将4进栈 | 36，4 |
| 2#：将2进栈 | 36，4，2 |
| '+'：出栈两次，将4+2=6进栈 | 36，6 |
| '/'：出栈两次，将36/6=6进栈 | 6 |
| postexp扫描完毕，算法结束，栈顶数值6即为所求 | |

**算法：**计算后缀表达式*postexp*的值。

```
double compvalue(char *postexp)
{  double d,  a,  b,  c,  e;
   SqStack1 *Opnd;                    //定义操作数栈
   InitStack1(Opnd);                  //初始化操作数栈
   while (*postexp!='\0')             //postexp字符串未扫描完时循环
   {  switch (*postexp)
      {
      case '+':                       //判定为'+'号
      Pop1(Opnd, a);                  //出栈元素a
      Pop1(Opnd, b);                  //出栈元素b
      c=b+a;                          //计算c
      Push1(Opnd, c);                 //将计算结果c进栈
      break;
```

```
case '-':                                   //判定为'-'号
        Pop1(Opnd, a);          //出栈元素a
        Pop1(Opnd, b);          //出栈元素b
        c=b-a;                  //计算c
        Push1(Opnd, c);                 //将计算结果c进栈
        break;
case '*':                                   //判定为'*'号
       Pop1(Opnd, a);           //出栈元素a
       Pop1(Opnd, b);           //出栈元素b
       c=b*a;                   //计算c
       Push1(Opnd, c);          //将计算结果c进栈
       break;
```

```
case '/':                                   //判定为'/'号
        Pop1(Opnd, a);                      //出栈元素a
        Pop1(Opnd, b);                      //出栈元素b
        if (a!=0)
        {  c=b/a;                           //计算c
           Push1(Opnd, c);                  //将计算结果c进栈
           break;
        }
        else
        {   printf("\n\t除零错误!\n");
            exit(0);                        //异常退出
        }
        break;
```

```
        default:                    //处理数字字符
            d=0;                    //转换成对应的数值存放到d中
            while (*postexp>='0' && *postexp<='9')
            {   d=10*d+*postexp-'0';
                postexp++;
            }
            Push1(Opnd, d);         //将数值d进栈
            break;
        }
        postexp++;                  //继续处理其他字符
    }
    GetTop1(Opnd, e);               //取栈顶元素e
    DestroyStack1(Opnd);            //销毁栈
    return e;                       //返回e
}
```

## 设计求解程序

建立如下主函数调用上述算法：

```
int main()
{   char exp[]="(56-20)/(4+2)";          //可将exp改为键盘输入
    char postexp[MaxSize];
    trans(exp, postexp);
    printf("中缀表达式:%s\n", exp);
    printf("后缀表达式:%s\n", postexp);
    printf("表达式的值:%g\n", compvalue(postexp));
    return 0;
}
```

## 运行结果

```
中缀表达式:(56-20)/(4+2)
后缀表达式:56#20#-4#2#+/
表达式的值:6
```

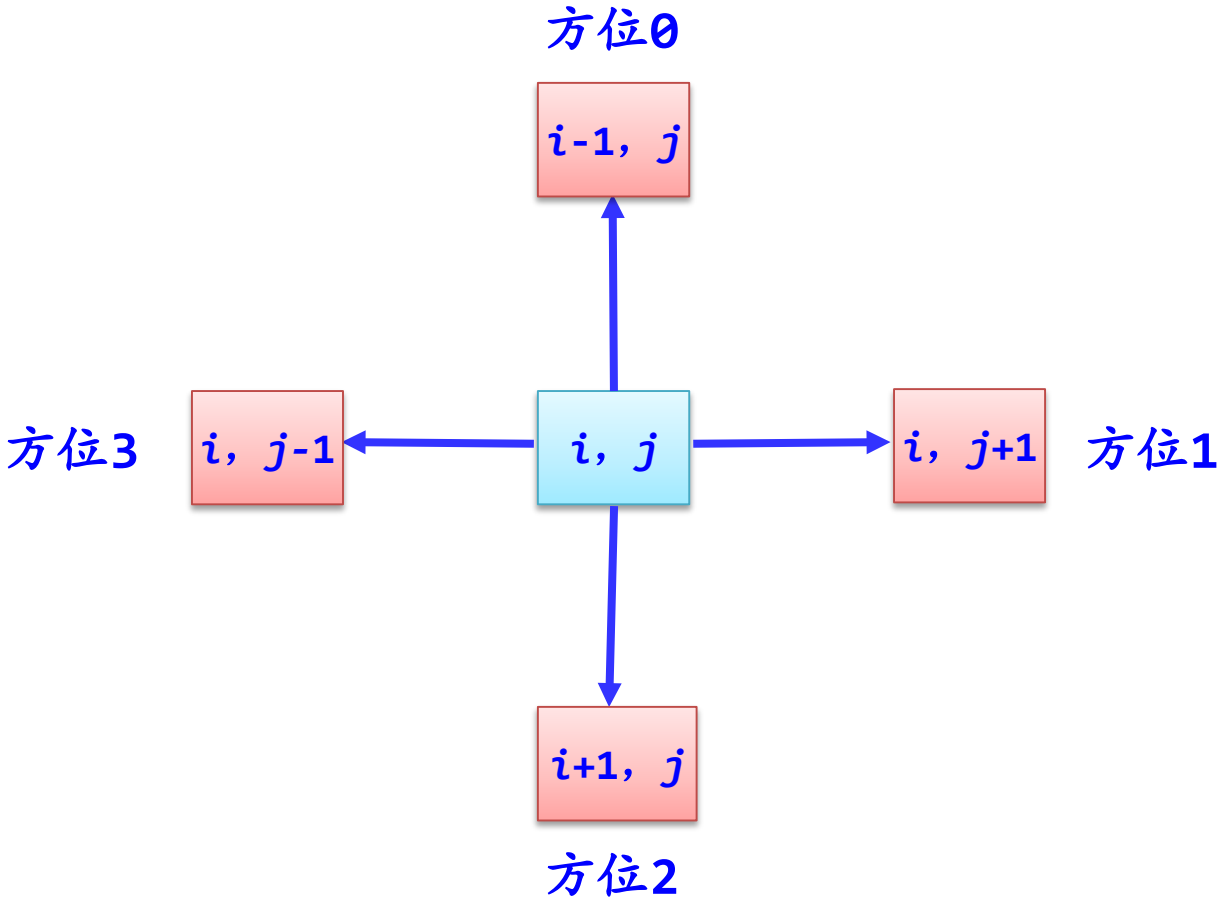
# 2、用栈求解迷宫问题

## 问题描述

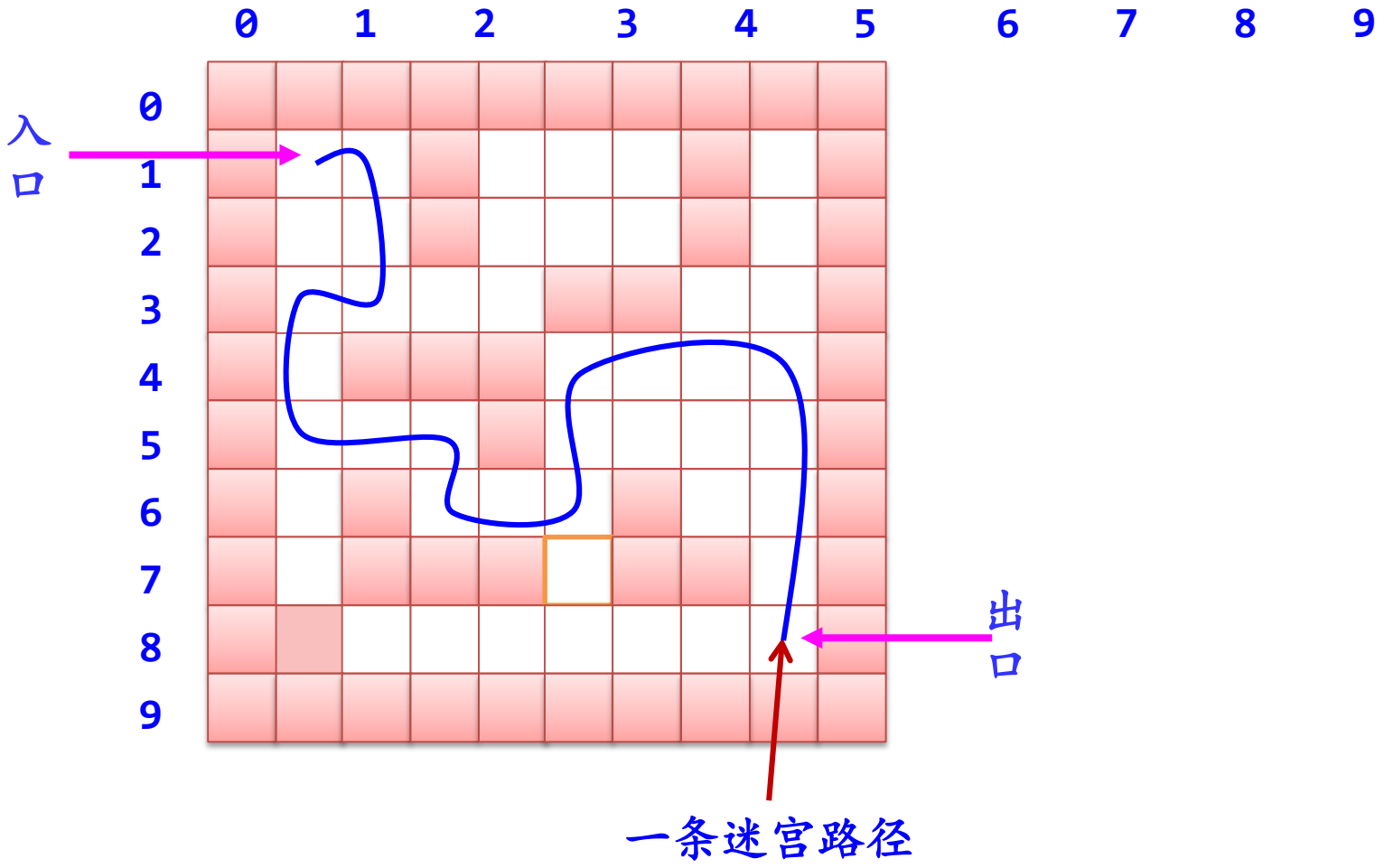
给定一个$M \times N$的迷宫图、入口与出口、行走规则。求一条从指定入口到出口的路径。

所求路径必须是简单路径，即路径不重复。

**行走规则**：上、下、左、右相邻方块行走。其中（i，j）表示一个方块

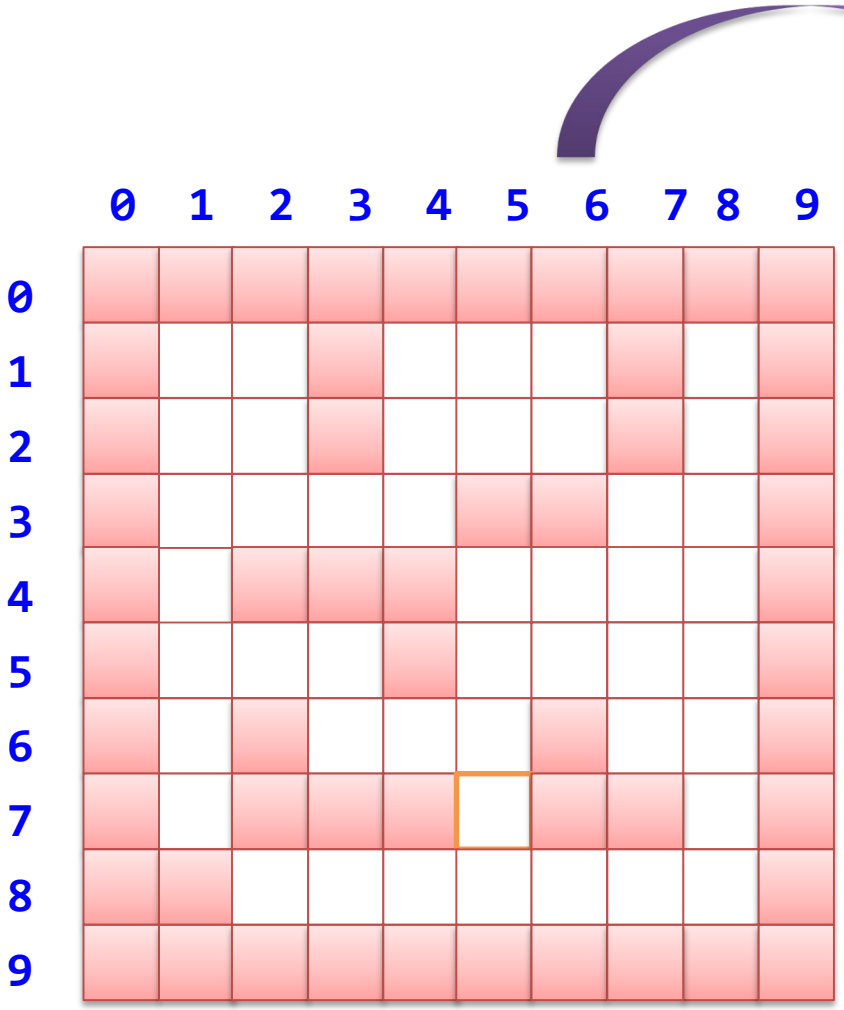


例如，M=8，N=8，图中的每个方块，用空白表示通道，用阴影表示障碍物。为了算法方便，一般在迷宫外围加上了一条围墙。



## 数据组织

设置一个迷宫数组mg，其中每个元素表示一个方块的状态，为0时表示对应方块是通道，为1时表示对应方块不可走。

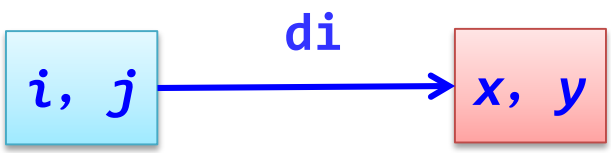



```
int mg[M+2][N+2]=
{
   {1, 1,1,1,1,1,1,1,1,  1},
   {1, 0,0,1,0,0,0,1,0,  1},
   {1, 0,0,1,0,0,0,1,0,  1},
   {1, 0,0,0,0,1,1,0,0,  1},
   {1, 0,1,1,1,0,0,0,0,  1},
   {1, 0,0,0,1,0,0,0,0,  1},
   {1, 0,1,0,0,0,1,0,0,  1},
   {1, 0,1,1,1,0,1,1,0,  1},
   {1, 1,0,0,0,0,0,0,0,  1},
   {1, 1,1,1,1,1,1,1,1,  1}
};
```

M×N

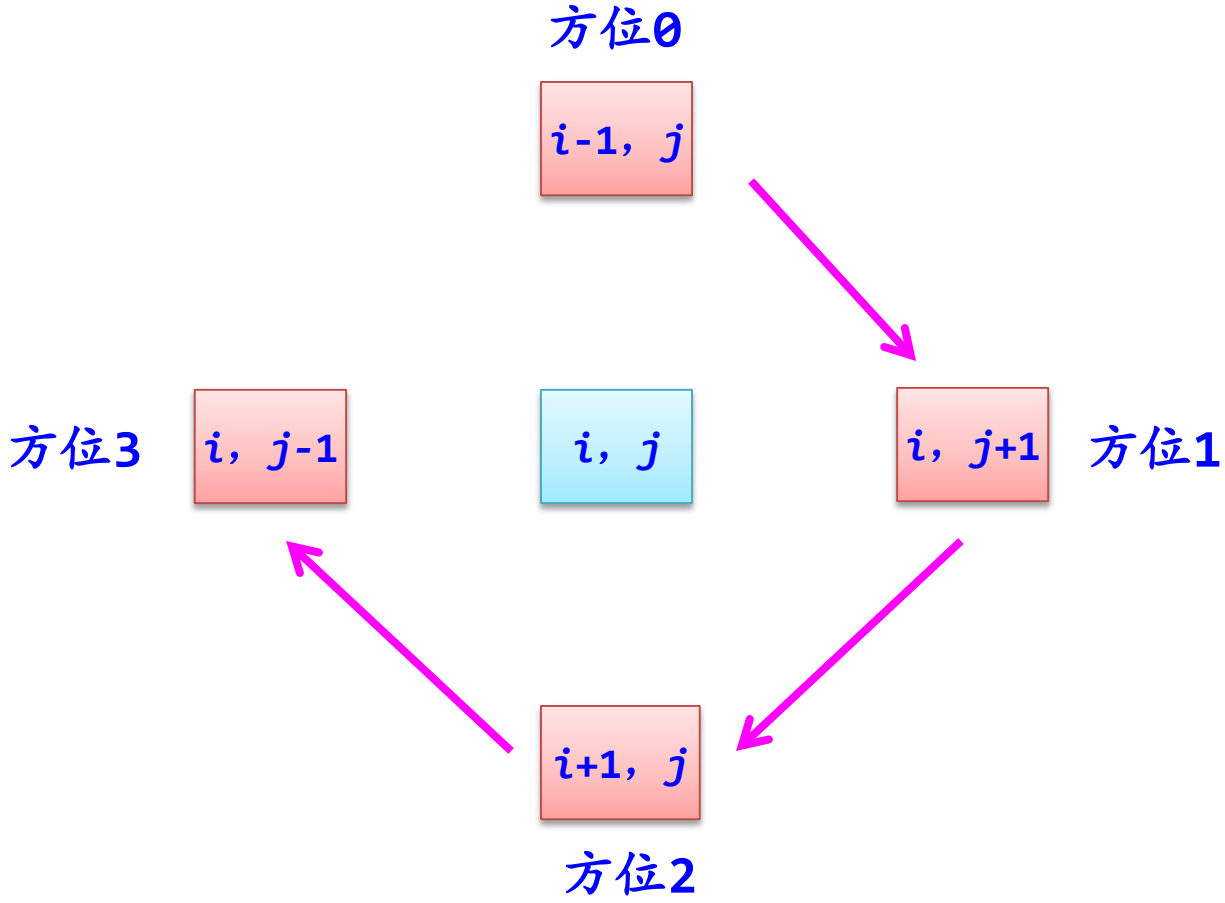
在算法中用到的栈采用顺序栈存储结构，即将栈定义为：

```
typedef struct
{   int i;                          //当前方块的行号
    int j;                          //当前方块的列号
    int di;                         //di是下一可走相邻方位的方位号
} Box;                              //定义方块类型
typedef struct
{   Box data[MaxSize];
    int top;                        //栈顶指针
}   StType;                         //定义顺序栈类型
```

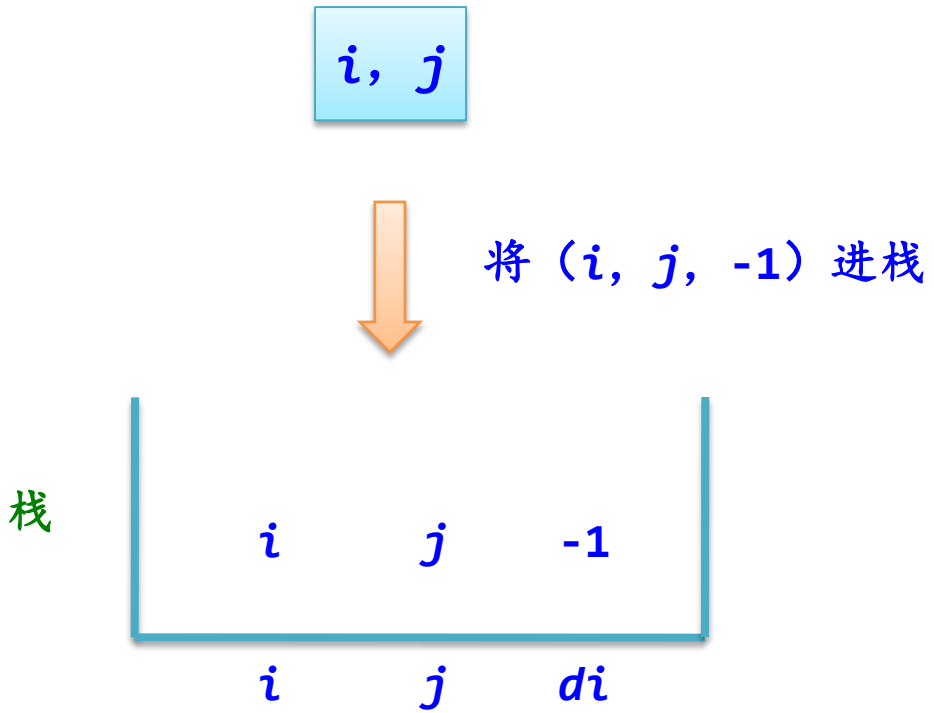


# 算法设计

**试探顺序：**从方位0开始，顺时针方向

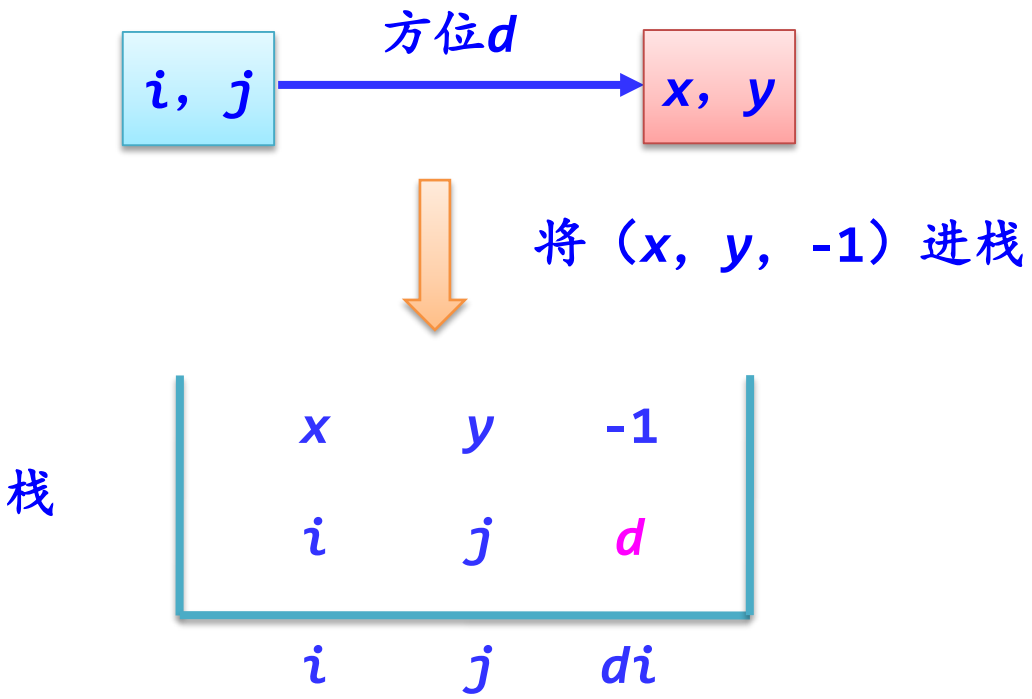


初始时，入口（i，j）作为当前方块。

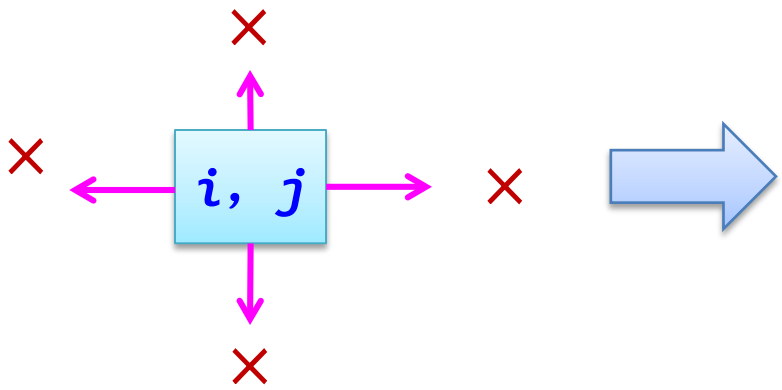


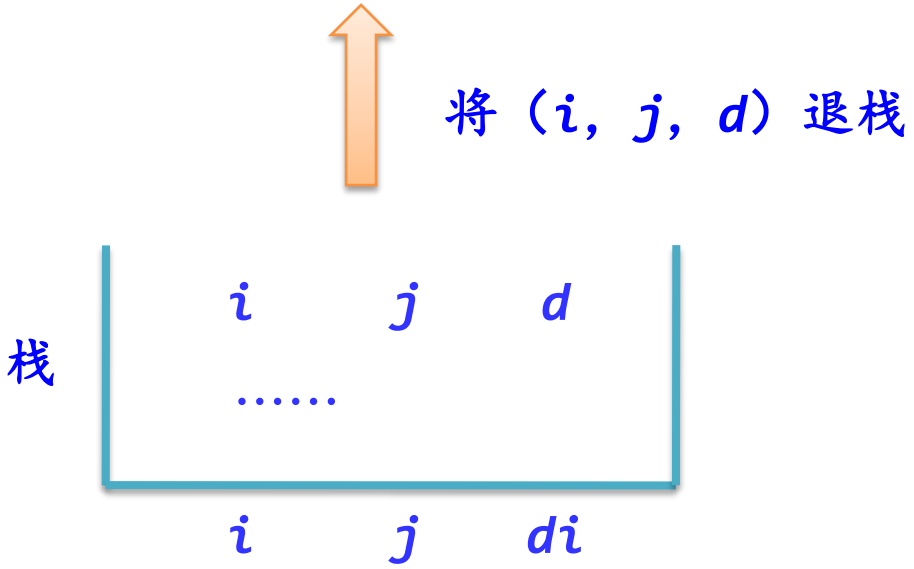

**所有走过的方块都会进栈！**

如果一个当前方块（i，j）找到一个相邻可走方块（x，y），就继续从（x，y）走下去。

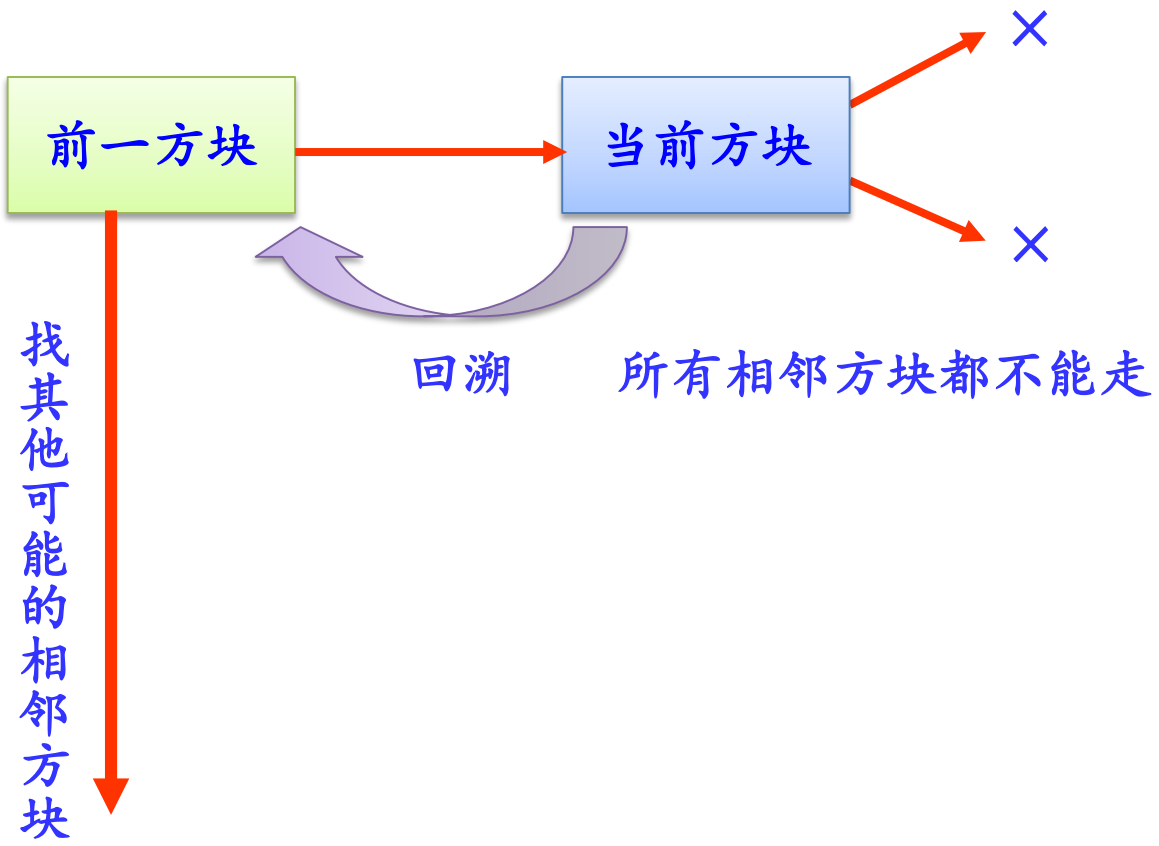


如果一个当前方块（$i$，$j$）没有找到任何相邻可走方块，表示此时无路可走，将其退栈。



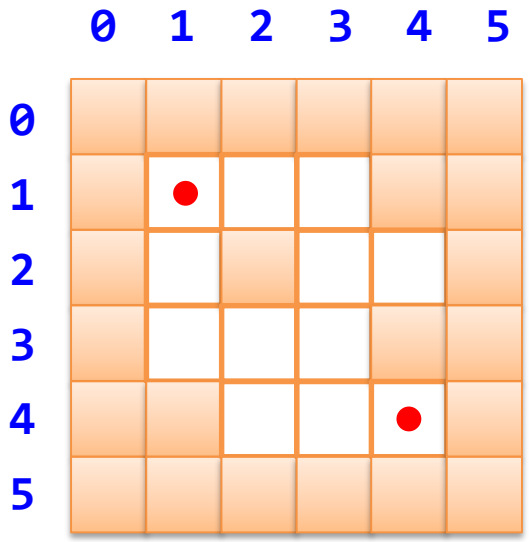



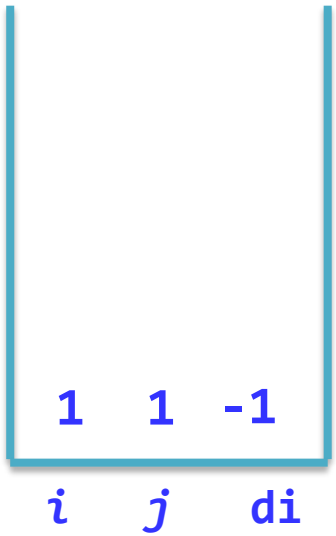
求解迷宫路径的过程：



# 用栈求一条迷宫路径的算法：（xi，yi）⇨（xe，ye）

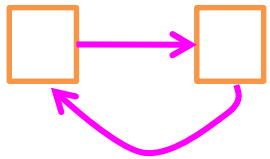
```
bool mgpath(int xi, int yi, int xe, int ye)
{  Box path[MaxSize], e; int i, j, di, i1, j1, k;  bool find;
   StType *st;                              //定义栈st
   InitStack(st);                           //初始化栈顶指针
   e.i=xi; e.j=yi; e.di=-1;                 //设置e为入口
   Push(st,e);                              //方块e进栈
   mg[xi][yi]=-1;     //入口的迷宫值置为-1避免重复走到该方块
```




一个栈

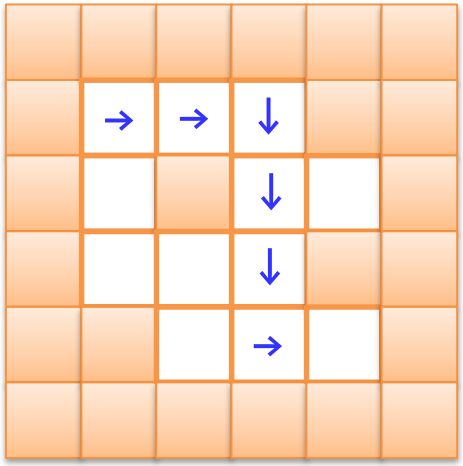
为了避免重复，当一个方块进栈时，将迷宫值改为-1



```
while (!StackEmpty(st))                    //栈不空时循环
{  GetTop(st,e);                           //取栈顶方块e
   i=e.i; j=e.j; di=e.di;
   if (i==xe && j==ye)                     //找到了出口,输出该路径
   {  printf("一条迷宫路径如下:\n");
      k=0;
      while (!StackEmpty(st))
      {  Pop(st,e);              //出栈方块e
         path[k++]=e; //将e添加到path数组中
      }
```

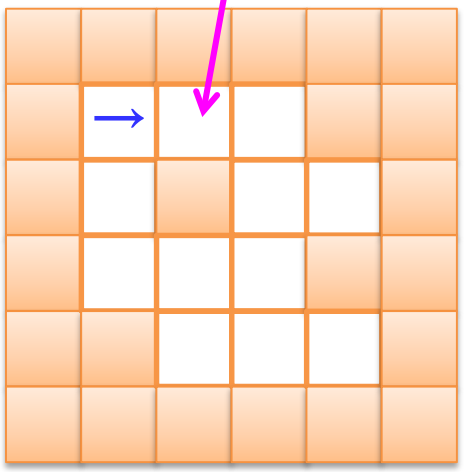
```
    while (k>=1)
    {  k--;
       printf("\t(%d,%d)",path[k].i,path[k].j);
          if ((k+2)%5==0)     //每输出每5个方块后换一行
       printf("\n");
    }
    printf("\n");
    DestroyStack(st);          //销毁栈
    return true;               //输出一条迷宫路径后返回true
}
```

| i | j | di |
|---|---|---|
| 4 | 4 | -1 |
| 4 | 3 | 1 |
| 3 | 3 | 1 |
| 2 | 3 | 2 |
| 1 | 3 | 2 |
| 1 | 2 | 1 |
| 1 | 1 | 1 |



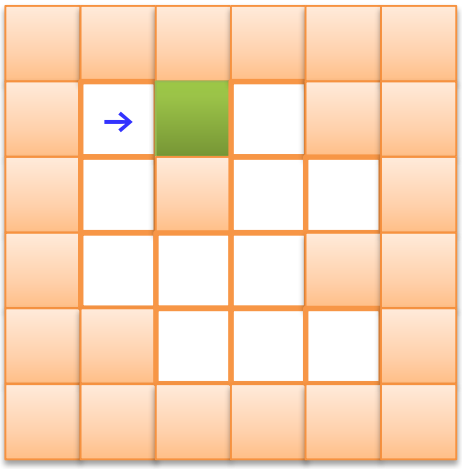
```
find=false;
while (di<4 && !find)   //找相邻可走方块(i1,j1)
{   di++;
    switch(di)
    {
    case 0:i1=i-1; j1=j;    break;
    case 1:i1=i;   j1=j+1;  break;
    case 2:i1=i+1; j1=j;    break;
    case 3:i1=i;   j1=j-1;  break;
    }
    if (mg[i1][j1]==0)  find=true;
    //找到一个相邻可走方块，设置find为真
}
```

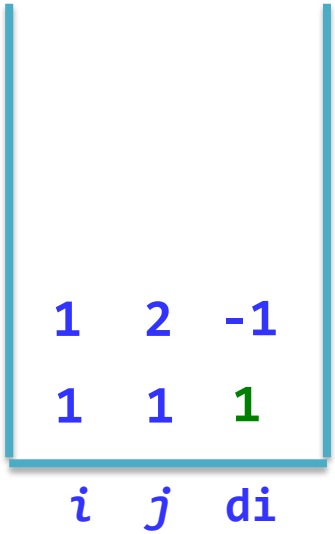
从入口（1,1）出发找到一个可走方块（1，2）



```
if (find)                              //找到了一个相邻可走方块(i1,j1)
{   st->data[st->top].di=di;           //修改原栈顶元素的di值
    e.i=i1; e.j=j1; e.di=-1;
    Push(st,e);                        //相邻可走方块e进栈
    mg[i1][j1]=-1;
                    //(i1,j1)迷宫值置为-1避免重复走到该方块
}
```

从入口（1,1）出发找到一个可走方块（1，2）:将（1，2，-1）进栈



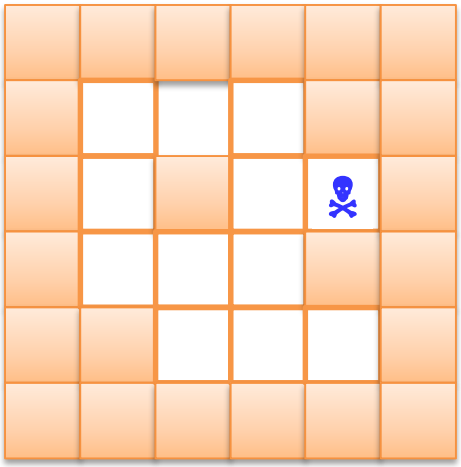

一个栈

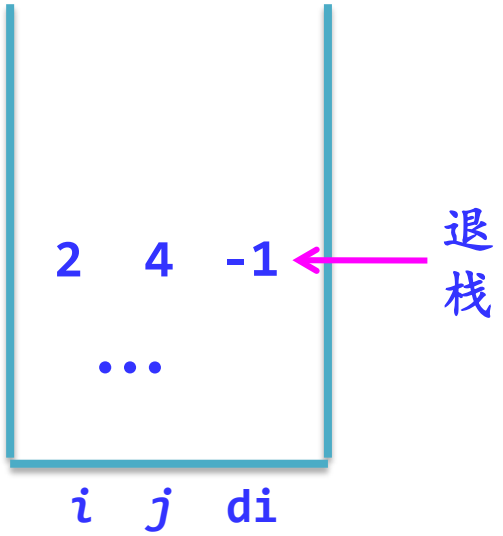
```
    else                    //没有路径可走,则退栈
    {   Pop(st,e);          //将栈顶方块退栈
        mg[e.i][e.j]=0;
         //让退栈方块的位置变为其他路径可走方块
    }
  }
  DestroyStack(st);         //销毁栈
  return false;             //表示没有可走路径
}
```

**疑难解答：**

**这里不将mg[栈顶方块]设置为0，程序执行也是正确的，但从原理上应该这样做，回退后需要恢复环境！**

（2,4）方块没有通路



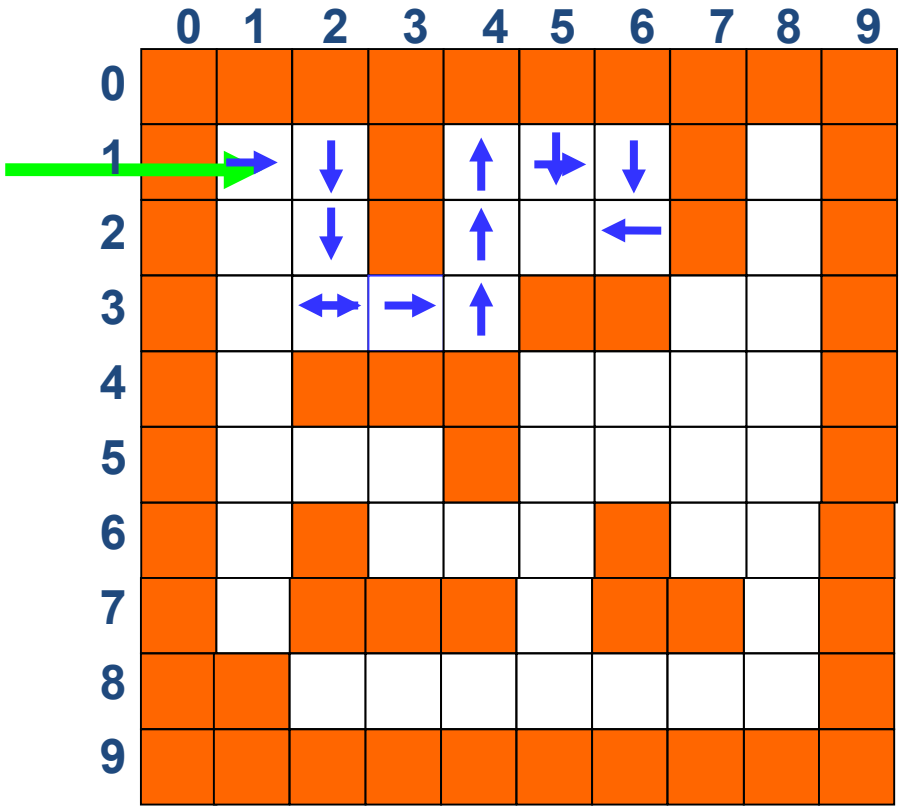



一个栈

# 设计求解程序

建立如下主函数调用上述算法：

```
int main()
{  if  (!mgpath(1,1,M,N))
       printf("该迷宫问题没有解!");
   return 1;
}
```



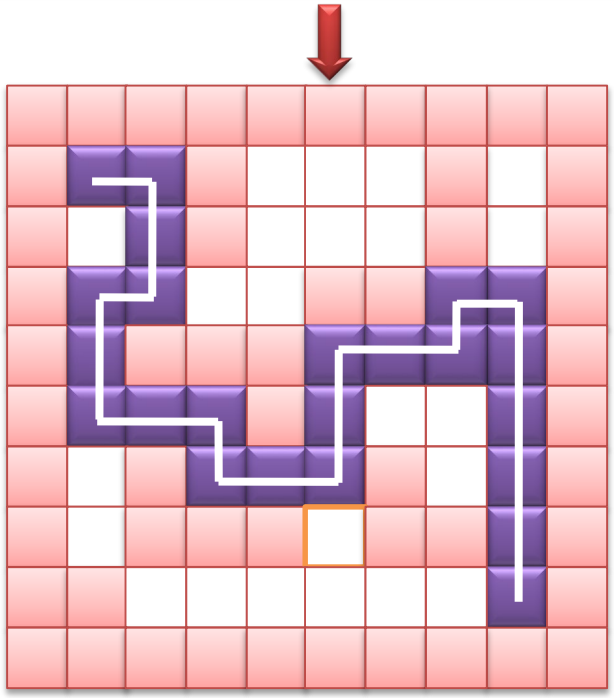

| | | |
|---|---|---|
| 2 | 5 | -1 |
| 2 | 6 | 3 |
| 1 | 6 | -1 |
| 1 | 5 | 2 |
| 1 | 4 | 1 |
| 2 | 4 | 0 |
| 3 | 4 | 0 |
| 3 | 3 | 1 |
| 3 | 2 | 3 |
| 2 | 2 | 2 |
| 1 | 2 | 2 |
| 1 | 1 | 1 |

# 运行结果

求解结果如下：

迷宫路径如下：

(1, 1) (1, 2) (2, 2) (3, 2) (3, 1)
(4, 1) (5, 1) (5, 2) (5, 3) (6, 3)
(6, 4) (6, 5) (5, 5) (4, 5) (4, 6)
(4, 7) (3, 7) (3, 8) (4, 8) (5, 8)
(6, 8) (7, 8) (8, 8)



显然，这个解不是最优解，即不是最短路径。为什么？

# 栈的经典例题

- 最小栈
  - 设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。
    - push(x) – 将元素 x 推入栈中。
    - pop() – 删除栈顶的元素。
    - top() – 获取栈顶元素。
    - getMin() – 检索栈中的最小元素。

# 栈的经典例题

- **例：Trapping Rain Water**
  - **Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.**
  - **The height of each pillar is at most K units.**
  - **时间复杂度O(N)**

# 栈的经典例题

- **例：Daily Temperatures**
  - **Given a list of daily temperatures T, return a list such that, for each day in the input, tells you how many days you would have to wait until a warmer temperature. If there is no future day for which this is possible, put 0 instead.**

**For example, given the list of temperatures T = [73, 74, 75, 71, 69, 72, 76, 73], your output should be [1, 1, 4, 2, 1, 1, 0, 0].**

# ——Thanks——