第三章 栈和队列

3.1 栈

3.2 栈的应用举例

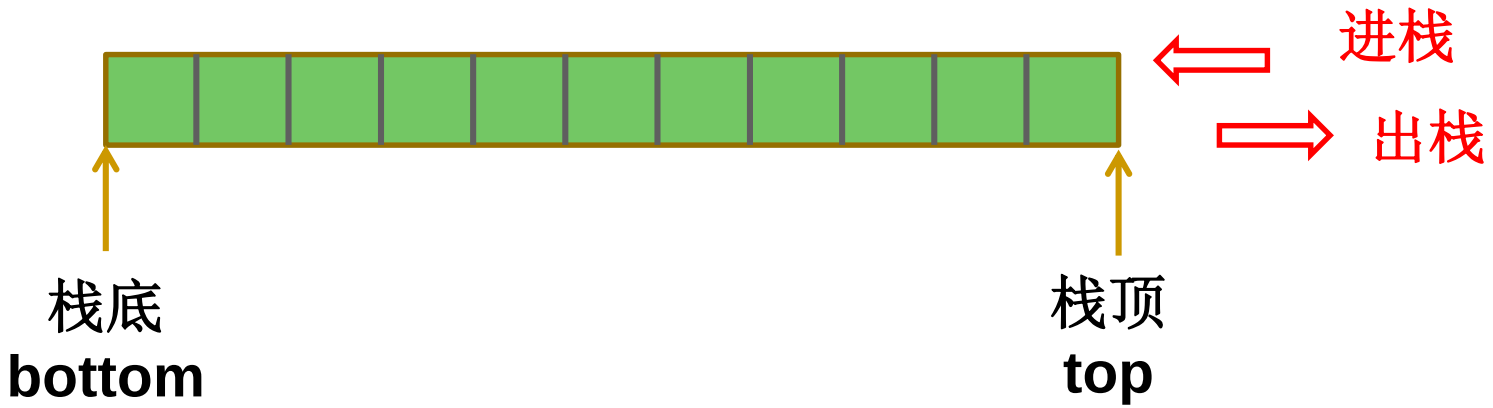
3.3 栈与递归的实现

3.4 队列

3.1 栈

一. 栈的概念

- 栈是一种操作受限的线性表。
- 允许插入和删除操作的一端称为栈顶(top，表尾)，另一端称为栈底(bottom，表头)



- 特点：后进先出 (LastInFirstOut)

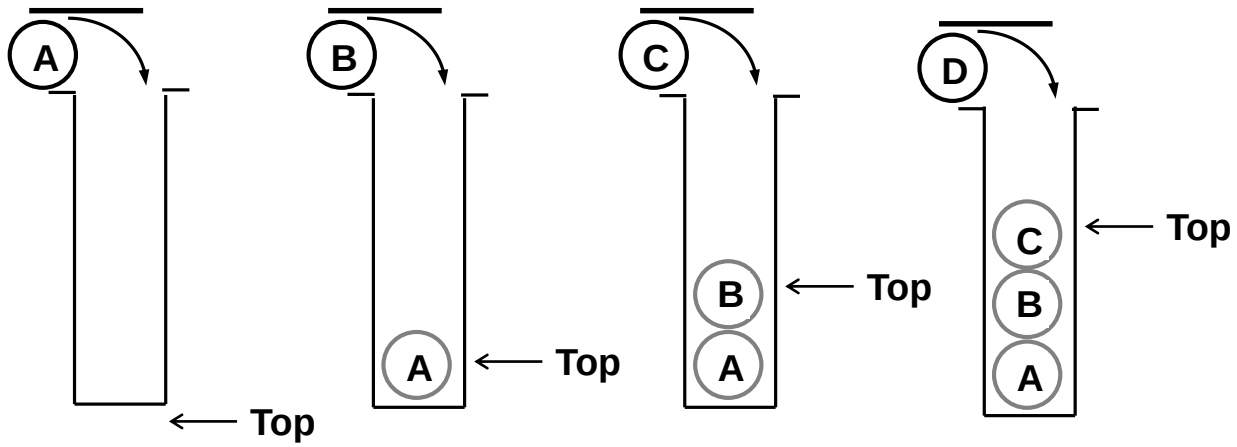
3.1 栈

一. 栈的概念

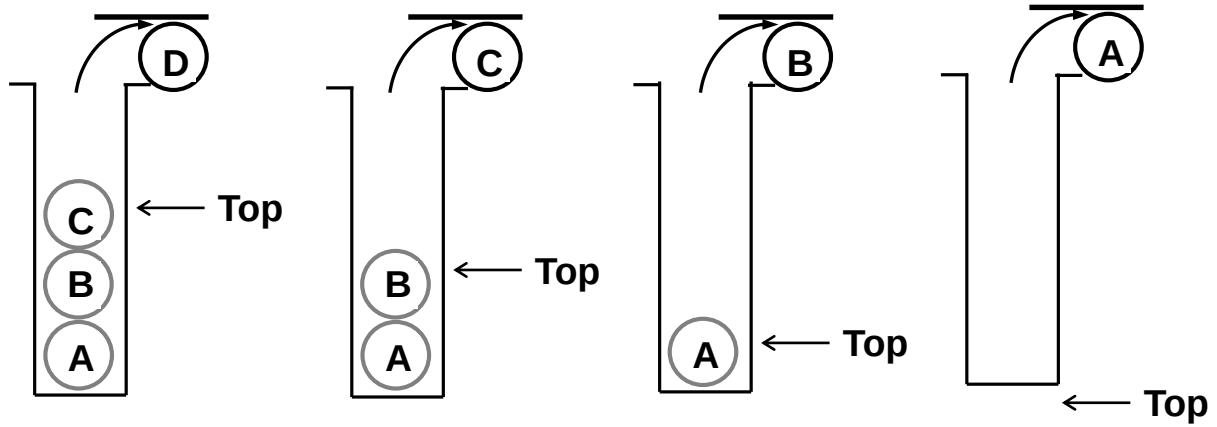
- 栈的ADT

```
ADT Stack {
    数据对象：D = {a_i | a_i∈ElemSet, i=1,2,3,...,n}
    数据关系：R = {<a_i-1, a_i> | a_i-1,a_i∈D}
    基本操作：InitStack(&S)              // 构造空栈
              Push(&S, e)               // 进栈（插入元素）
              Pop(&S, &e)               // 出栈（删除元素）
              GetTop(S, &e)             // 取栈顶元素值
              StackEmpty(S)             // 栈是否为空
} ADT Stack
```

3.1 栈



CreatStack( ); **Push(S,A);** **Push(S,B);** **Push(S,C);**



x=Pop(S); **x=Pop(S);** **x=Pop(S);** **IsEmpty (S)**

3.1 栈

二. 顺序栈

- 顺序栈是栈的一种实现，是顺序存储结构，利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。
- 顺序栈的定义

```
#define STACK_INIT_SIZE    100        // 栈存储空间的初始分配量
#define STACKINCREMENT    10          // 栈存储空间的分配增量
Typedef struct {
      SElemType *base   // 栈底指针，即栈的基址
      SElemType *top;   // 栈顶指针（非空栈，指向栈顶元素的下一个位置）
      int   stacksize;  // 当前分配的存储容量(元素数)
} SqStack;
```

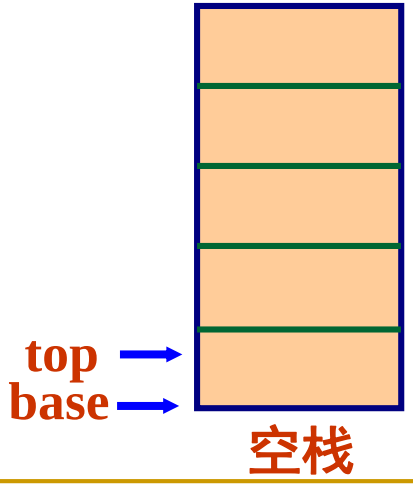
- 顺序栈在存储结构上类似于顺序表，都是一维数组

3.1 栈

二. 顺序栈

- 顺序栈的创建

```
Status InitStack(SqStack &S) {
    S.base = (SElemType *) malloc(STACK_INIT_SIZE *
                                  sizeof(SElemType));
    if ( !S.base )      exit(OVERFLOW);      // 存储分配失败
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    return OK;
} // InitStack
```

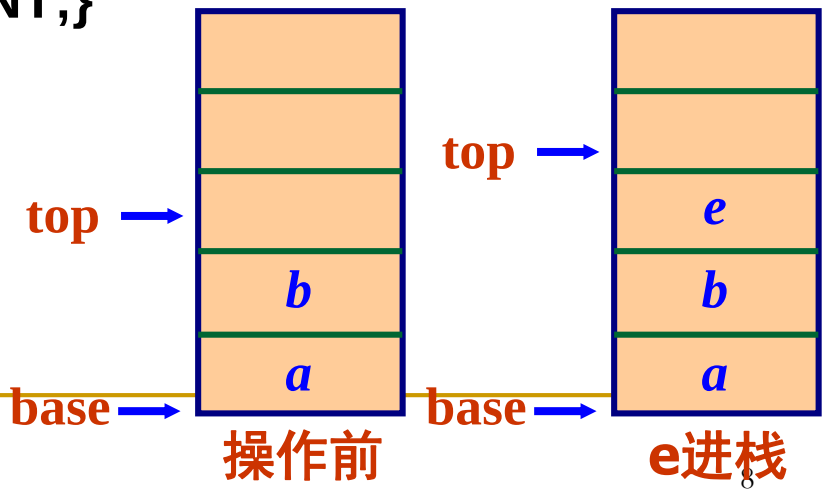


3.1 栈

二. 顺序栈

- 顺序栈的进栈（插入元素）

```
Status Push(SqStack &S, SElemType e) {
    if ( S.top – S.base == S.stacksize ) {      // 栈满，追加存储空间
        S.base = (SElemType *) realloc(S.base, (S.stacksize +
                                   STACKINCRMENT* sizeof(SElemType));
        if (!S.base)       exit(OVERFLOW);
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCRMENT;}
    *S.top++ = e;
    return OK;
} // Push
```

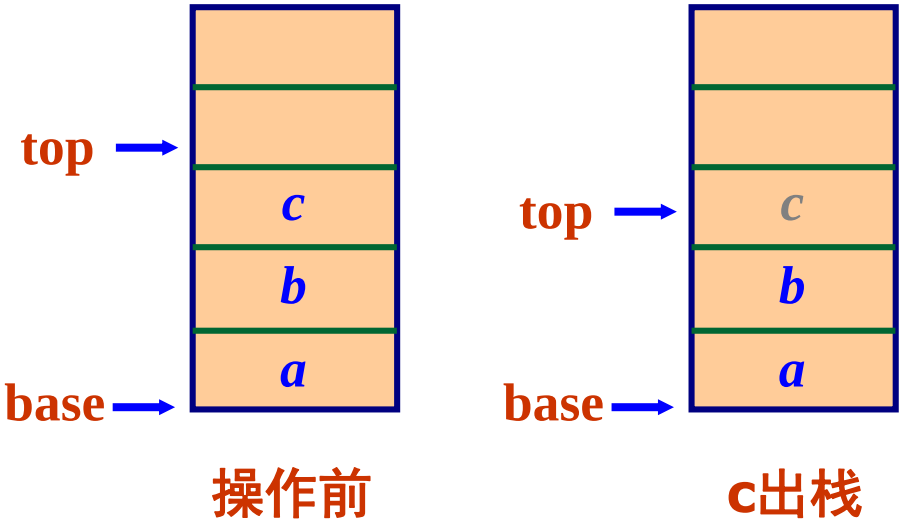


3.1 栈

二. 顺序栈

- 顺序栈的出栈（删除元素）

```
Status Pop(SqStack &S, SElemType &e) {
        if (  S.top == S.base  ) return ERROR;   // 栈空
        e = *--S.top;
        return OK;
} // Pop
```



3.1 栈

二. 顺序栈

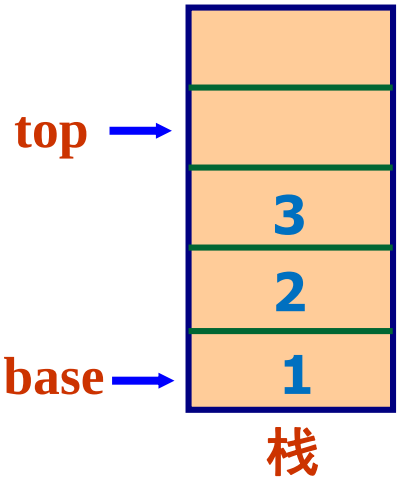
- 顺序栈的取栈顶元素

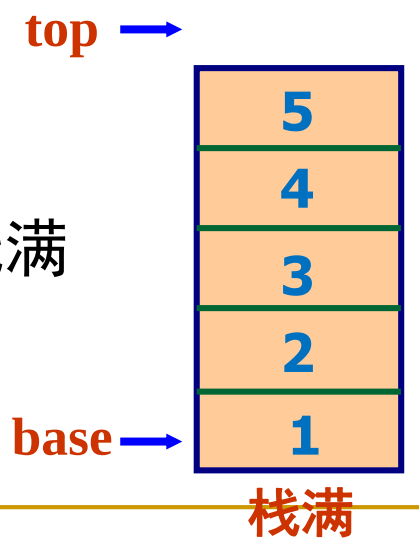
```
Status GetTop(SqStack S, SElemType &e) {
        if ( S.top == S.base )    return ERROR;          // 栈空
        e = *(S.top-1);
        return OK;
} // GetTop
```

3.1 栈

二. 顺序栈

- 顺序栈的特性：
  - top=base 表示空栈
  - base=NULL 表示栈不存在
  - 插入栈顶元素时，指针 top+1
  - 删除栈顶元素时，指针 top-1
  - top - base = stacksize 时，表示栈满

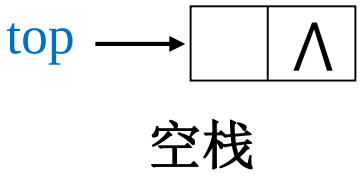




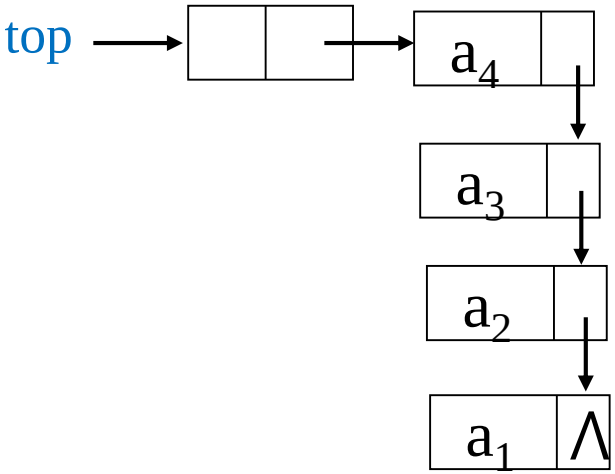
3.1 栈

三. 链栈

- 栈的链式存储结构称为链栈，是运算受限的链表，其插入和删除操作只能在链表的表头进行。



空栈



非空栈

链栈的结点类型说明如下：

```
typedef  struct  Stack_Node
    {   ElemType   data ;
        struct Stack_Node  *next ;
    } Stack_Node ;
```

链栈的优点是不存在栈满上溢的情况。

3.1 栈

三. 链栈

- 初始化

```
void  Init_Link_Stack( Stack_Node *top )
    {
        top = malloc( sizeof(struct Stack_Node ));
        top->next = NULL ;
    }
```

3.1 栈

三. 链栈

- 进栈

```
Status push(Stack_Node *top , ElemType  e)
    {   Stack_Node  *p ;
        p=(Stack_Node  *)malloc(sizeof(Stack_Node)) ;
        if ( !p )  return  ERROR; // 申请新结点失败，返回错误标志
        p->data=e ;
        p->next=top->next  ;
        top->next=p ;
        return OK;
    }
```

3.1 栈

三. 链栈

- 出栈

```
Status pop(Stack_Node  *top , ElemType *e)
    //  将栈顶元素出栈
    {  Stack_Node  *p ;
       ElemType  e ;
       if  ( top->next==NULL )
            return ERROR ;   // 栈空，返回错误标志
       p=top->next ;    e=p->data ;   //  取栈顶元素
       top->next=p->next ;    //  修改栈顶指针
       free(p) ;
       return OK ;
    }
```

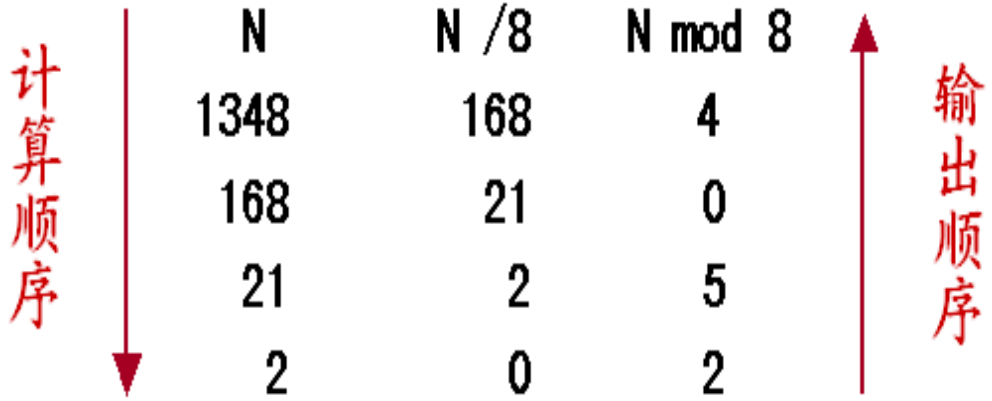
3.2 栈的应用

一. 数制转换

- 将十进制转换为其它进制(d)，其原理为：

$$N = (N/d)*d + N \bmod d$$

例如：$(1348)_{10} = (2504)_8$，其运算过程如下：

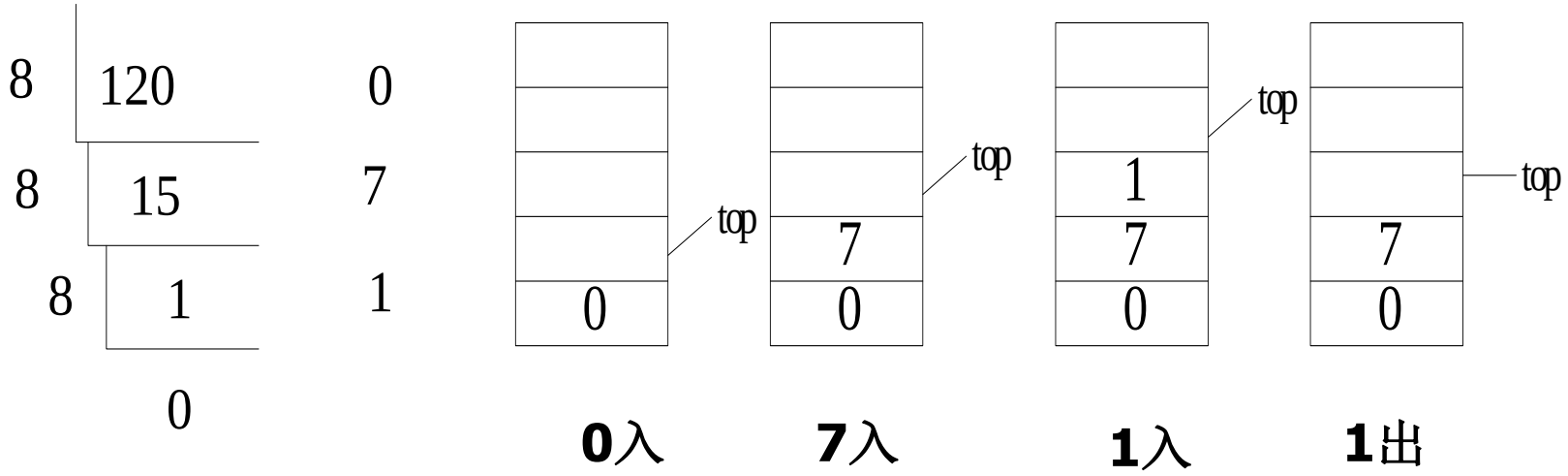


计算时，从低位到高位顺序产生各个数位

输出时，应从高位到低位依次输出各个数位

3.2 栈的应用

一. 数制转换



$$(120)_{10} = (170)_8$$

3.2 栈的应用

一. 数制转换

- 实现函数

```
void conversion () {
      InitStack(S);                    // 创建新栈S
       scanf (“%d”, N );               // 输入一个十进制数N
       while (N) {
         Push(S, N % 8);               // 将余数送入栈中
         N = N/8;                      // 求整除数
       }
       while (!StackEmpty(S)) {        // 如果栈不空
         Pop(S,e);                     // 将栈中数出栈
         printf ( "%d", e );
       }
} // conversion
```

3.2 栈的应用

二. 括号匹配的校验

- 在处理表达式过程中需要对括号匹配进行检验，括号匹配包括三种：“(”和“)”，“[”和“]”，“{”和“}”。例如表达式中包含括号如下：

(	)	[	(	)	(	[	]	)	]	{	}
1	2	3	4	5	6	7	8	9	10	11	12

上例可以看出第1和第2个括号匹配，第3和第10个括号匹配，4和5匹配，6和9匹配，7和8匹配，11和12匹配

3.2 栈的应用

二. 括号匹配的校验

- 求解算法

① 初始化，i=0，建立堆栈，栈为空，输入表达式

② 读取表达式第i个字符

③ 如果第i个字符是左括号，入栈

④ 如果第i个字符是右括号，检查栈顶元素是否匹配？

a) 如果匹配，弹出栈顶元素

b) 如果不匹配，报错退出

⑤ i+1，是否已经到表达式末尾？

a) 未到末尾，重复步骤2

b) 已到达末尾，执行步骤6

⑥ 堆栈为空，返回匹配正确，堆栈不为空，返回错误

3.2 栈的应用

三. 行编辑

- 用户输入一行字符
- 允许用户输入出差错，并在发现有误时，可以用退格符“#”及时更正
  - 假设从终端接受两行字符：

    whli##ilr#e（s#*s)
  - 实际有效行为：

    while（*s)

3.2 栈的应用

三. 行编辑

- 实现函数

```
//对用户输入的一行字符进行处理，直到换行符“\n”
    ch = getchar();              // 从终端输入一个字符
    while ( ch != '\n') {
      switch(ch) {
        case '#': Pop(S, c); break; // 仅当栈非空时退栈
        default:  Push(S, ch);   break;   // 有效字符进栈
      }
      ch = getchar();                  // 从终端输入一个字符
    }
//将从栈底到栈顶的栈内字符传送到调用过程的数据区;
    ……
```

3.2 栈的应用

四. 迷宫求解

- 问题描述

给定一个M * 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。

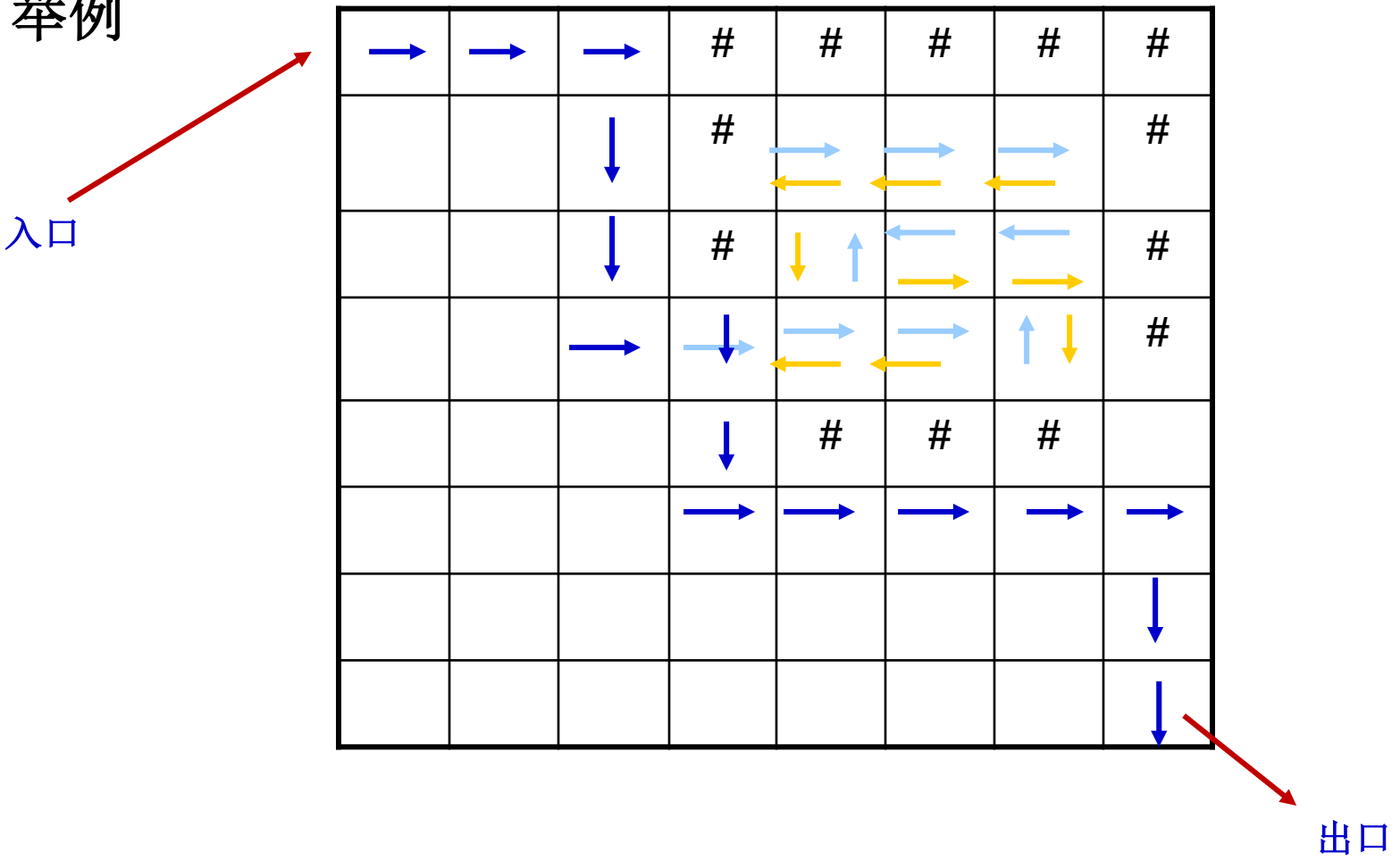
- 求解策略

“穷举法”、“回溯法”

3.2 栈的应用

四. 迷宫求解

- 举例

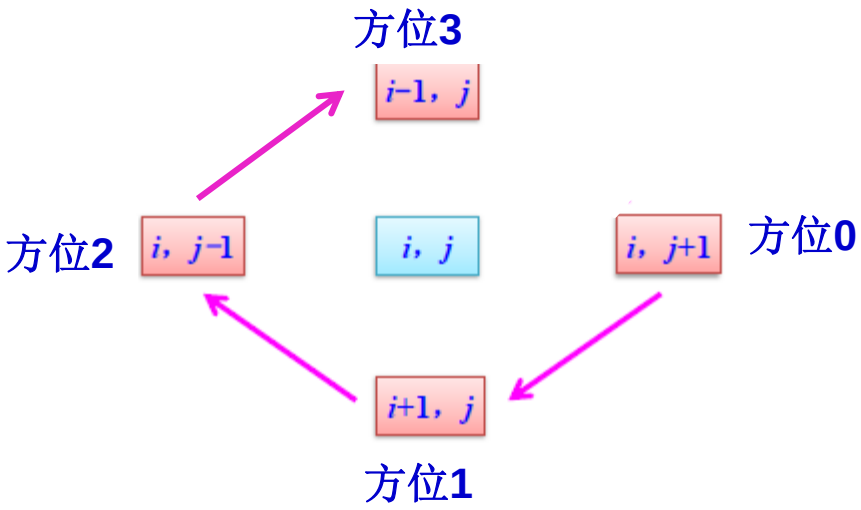


3.2 栈的应用

四. 迷宫求解

- 试探顺序：

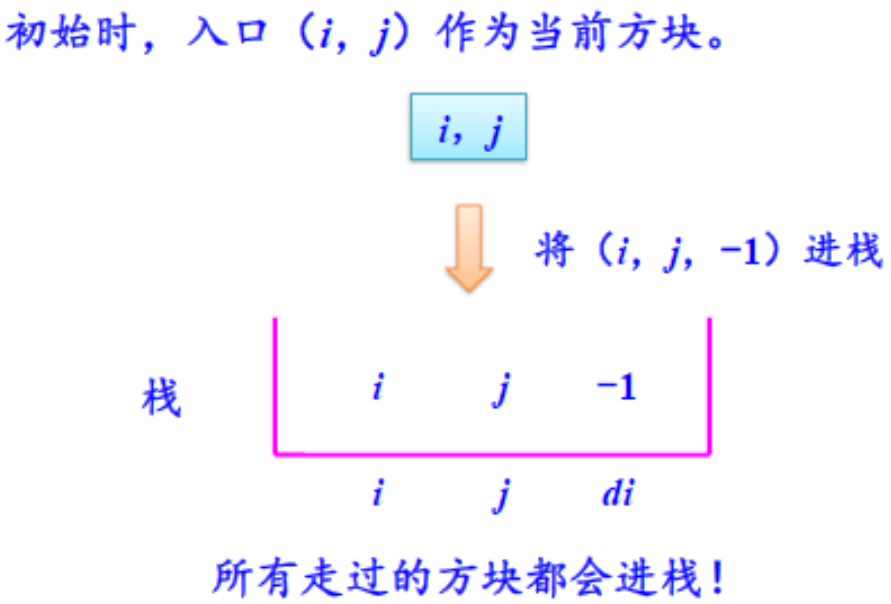
逐一沿顺时针方向查找相邻块（一共四块—东(右)、南(下)，西(左)、北(上)）是否可通，即该相邻块既是通道块，且不在当前路径上。



3.2 栈的应用

四. 迷宫求解

- 用一个栈来记录已走过的路径

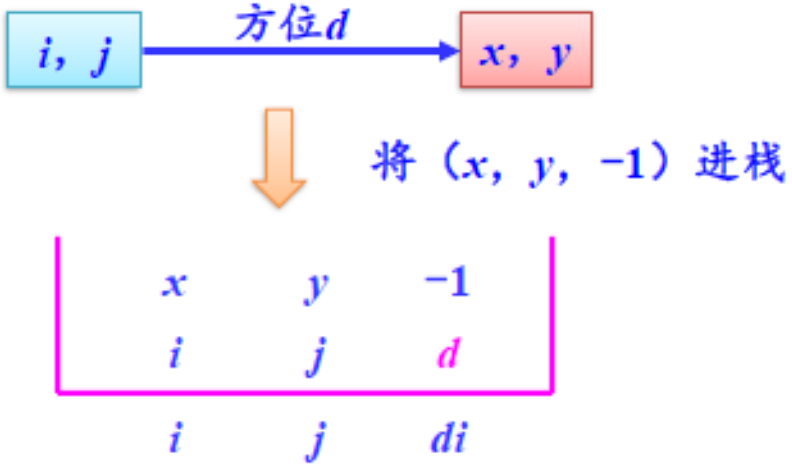


3.2 栈的应用

四. 迷宫求解

- 用一个栈来记录已走过的路径

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。



3.2 栈的应用

四. 迷宫求解

■ 求解算法

```
设定当前位置为入口位置
  do { 若当前位置可通，则
       {
           将该位置插入栈顶(Push)；若该位置是出口，则结束
           否则切换当前位置的东邻方块为当前位置；
         }
      否则
       {
           若栈不空则
          {
             如果栈顶位置的四周均不可通,则删除栈顶位置(Pop)
             并重新测试新的栈顶位置
             如果找到栈顶位置的下一方向未经探索，则将该方向方块设为当前位置
          }
       }
  }while(栈不空)；          找不到路径
```

❖ 栈应用：表达式求值

计算机编译程序是如何自动地理解表达式的？

要实现表达式求值，首先需要正确理解一个表达式，主要是运算的先后顺序。

➢ 算术表达式的基本计算规则：

不同运算符号优先级不一样，先乘除，后加减；先括号内，再括号外； 相同优先级情况下从左到右。

例如，算术表达式 5+6/2-3*4，它的正确计算顺序应该是：

5+6/2-3*4 = 5+3-3*4 = 8-3*4 = 8-12 = -4。

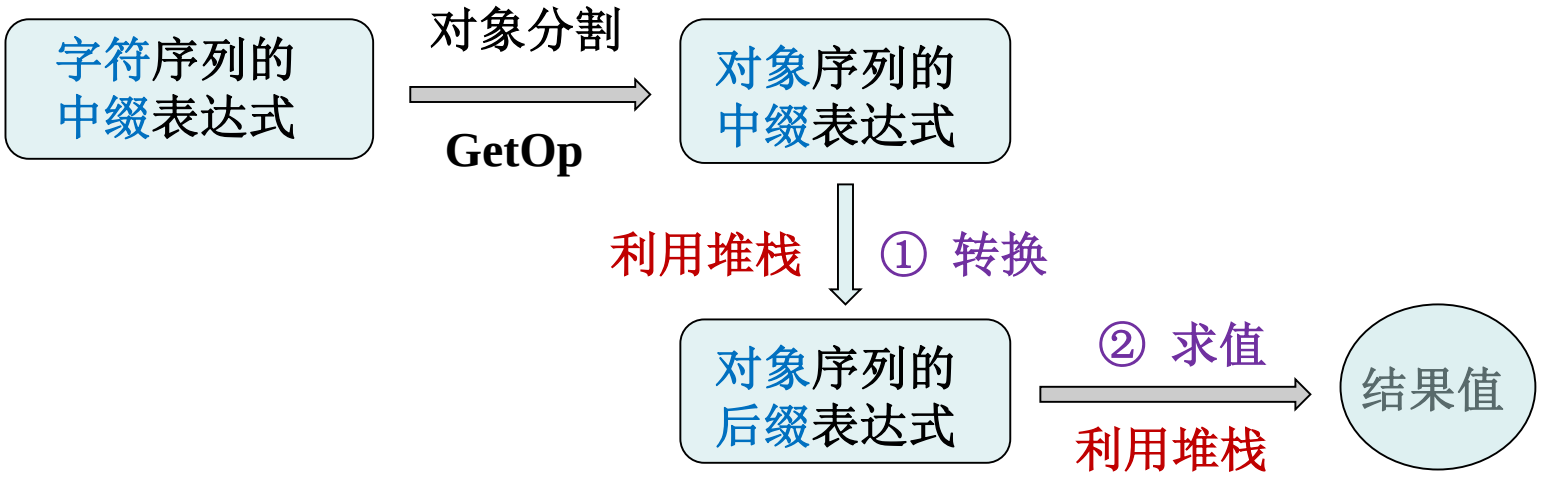
➢算术表达式主要由两类对象构成，即运算数（如2、3、4等）和运算符号（如+、-、*、/等），而且运算符号均位于两个运算数中间。

❖ 中缀表达式求值

➢ 中缀表达式：运算符号位于两个运算数之间。如，$a+b*c$

➢ 后缀表达式：运算符号位于两个运算数之后。如，$a\ b\ c\ *\ +$

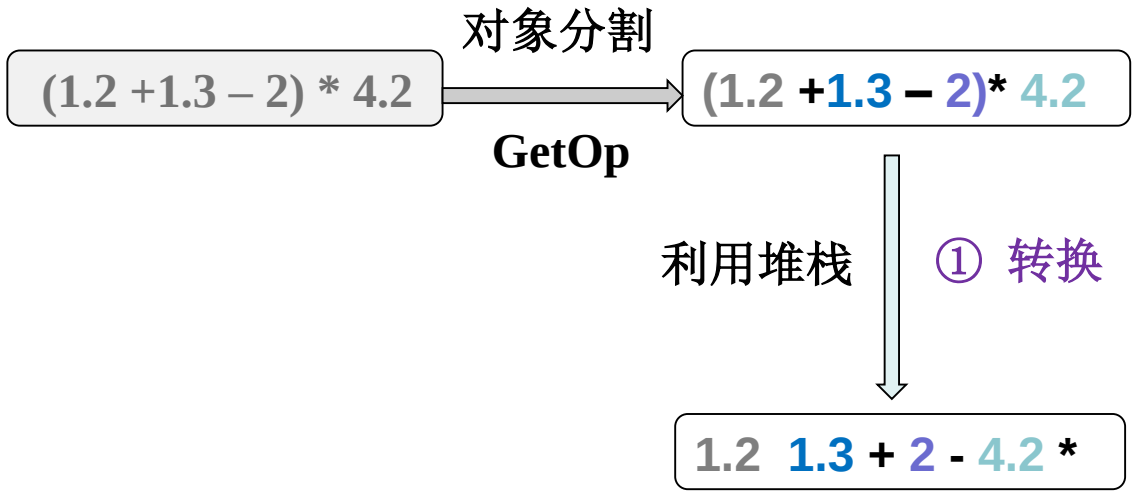
◆ 应用栈将中缀表达式的求值转换成后缀表达式的求值



❖ 中缀表达式求值

- 第一步：应用栈实现中缀表达式到后缀表达式的转换

[例如]



❖（一）中缀表达式如何转换为后缀表达式

➢ 从头到尾读取中缀表达式的每个对象，对不同对象按不同的情况处理。对象分下列6种情况：

① 如果遇到空格则认为是分隔符，不需处理；
② 若遇到运算数，则直接输出；
③ 若是左括号，则将其压入堆栈中；
④ 若遇到的是右括号，表明括号内的中缀表达式已经扫描完毕，将栈顶的运算符弹出并输出，直到遇到左括号（左括号也出栈，但不输出）；
⑤ 若遇到的是运算符，若该运算符的优先级大于栈顶运算符的优先级时，则把它压栈；若该运算符的优先级小于等于栈顶运算符时，将栈顶运算符弹出并输出，再比较新的栈顶运算符，按同样处理方法，直到该运算符大于栈顶运算符优先级为止，然后将该运算符压栈；
⑥ 若中缀表达式处理完毕，则把堆栈中存留的运算符一并输出。

❖ （一）中缀表达式如何转换为后缀表达式

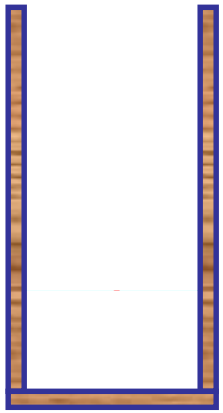
相邻运算符间的优先关系

顺序	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	×
)	>	>	>	>	×	>	>
#	<	<	<	<	<	×	=

❖ （一）中缀表达式如何转换为后缀表达式

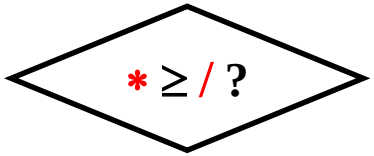
〖例〗 $a * (b + c) / d$ ⇒ $a\ b\ c + * d /$

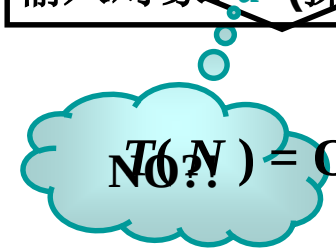
输出： a b c + * d /



← top

输入对象: a (操作数)	输入对象: * (乘法)
输入对象: ((左括号)	输入对象: b (操作数)
输入对象: + (加法)	输入对象: c (操作数)
输入对象:) (右括号)	输入对象: / (除法)
输入对象: d (操作数)	





❖中缀转换为后缀示例： 2*（9+6/3-5）+4

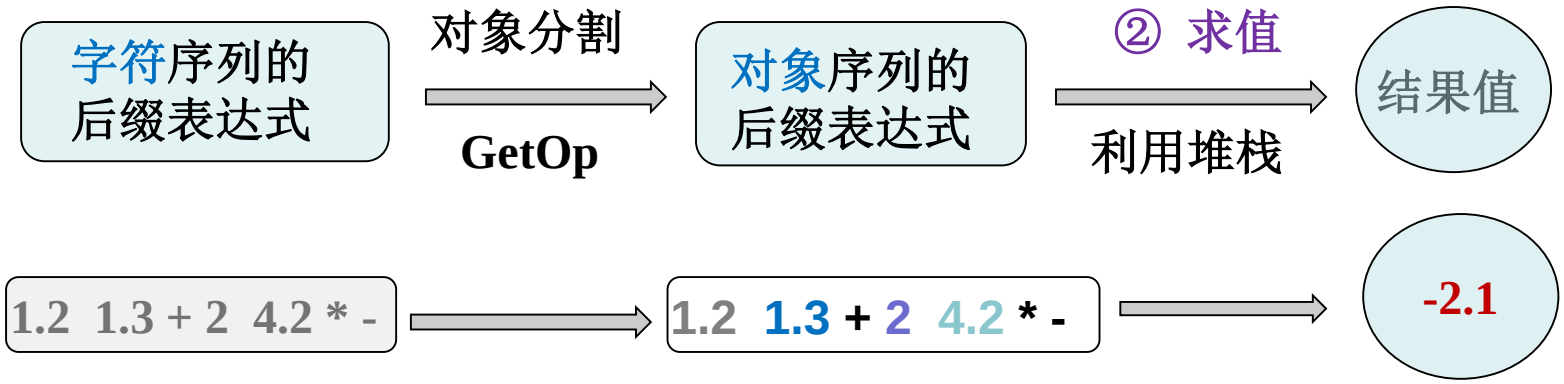
步骤	待处理表达式	堆栈状态 （底←→顶）	输出状态
1	2*（9+6/3-5）+4		
2	*（9+6/3-5）+4		2
3	（9+6/3-5）+4	*	2
4	9+6/3-5）+4	*（	2
5	+6/3-5）+4	*（	2 9
6	6/3-5）+4	*（ +	2 9
7	/3-5）+4	*（ +	2 9 6
8	3-5）+4	*（ + /	2 9 6
9	-5）+4	*（ + /	2 9 6 3
10	5）+4	*（ -	2 9 6 3 / +
11	）+4	*（ -	2 9 6 3 / + 5
12	+4	*	2 9 6 3 / + 5 -
13	4	+	2 9 6 3 / + 5 - *
14		+	2 9 6 3 / + 5 - * 4
15			2 9 6 3 / + 5 - * 4 +

❖ （二）应用栈完成后缀表达式的求值

◆ 基本过程：

从左到右读入后缀表达式的各项（运算符或运算数）； 根据读入的对象（运算符或运算数）判断执行操作， 操作分为下列3种情况：

① 当读入的是一个运算数时，把它压入栈中；
② 当读入的是一个运算符时，就从堆栈中弹出适当数量的运算数，对该运算进行计算，计算结果再压回到栈中；
③ 处理完整个后缀表达式之后，堆栈顶上的元素就是表达式的值。

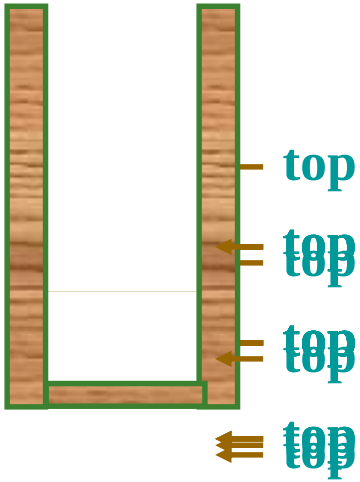


❖ 后缀表达式

中缀表达式：
6/2-3+4*2

后缀表达式的求值计算过程：

例：6 2 / 3 – 4 2 * + 的值是多少？ 8



对象: 6 (运算数)	对象: 2 (运算数)
对象: / (运算符)	对象: 3 (运算数)
对象: – (运算符)	对象: 4 (运算数)
对象: 2 (运算数)	对象: * (运算符)
对象: + (运算符)	Pop: 8

不需要知道运算符的优先规则。

3.3 栈与递归实现

- 栈的另一个重要应用是在程序设计语言中实现递归调用
- 递归调用：一个函数(或过程)直接或间接地调用自己本身。

当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三项任务：

①为调用函数的局部变量分配存储区；
②将所有的实在参数、返回地址等信息传递给被调用函数保存；
③将控制转移到被调用函数的入口。

从被调用函数返回调用函数之前，应该完成下列三项任务：

①保存被调函数的计算结果；

②释放被调函数的数据区；

③依照被调函数保存的返回地址将控制转移到调用函数。

多个函数嵌套调用的规则是：后调用先返回！

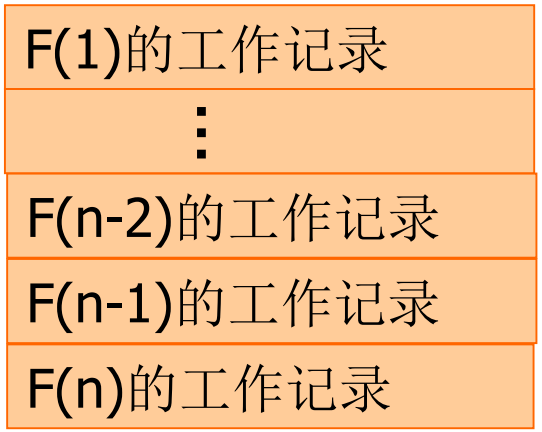
3.3 栈与递归实现

- 为保证递归调用正确执行，系统设立一个“运行栈”，作为整个递归调用过程期间使用的数据存储区。

  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。

- 例如：求n!

```
int f(n)
{ int res=n;
  if ( n==1 )
     return 1;
  else
     res = res * f(n-1);
  return res;
}
```



小结

- 栈是限定操作的线性表，特点是后进先出
- 栈包括top栈顶、base栈底，插入、删除、访问都在top端进行
  - 插入操作就是进栈push
  - 删除操作就是出栈pop
  - 访问就是取栈顶元素GetTop
  - top=base 或 top->next = NULL表示栈空
  - base=NULL 栈不存在
  - top-base = stacksize 时，顺序栈栈满
- 栈的应用：数制转换、行编辑、括号匹配、迷宫求解

练习

- 已知栈SS从栈底到栈顶的元素排列为：A、B、C、D，栈操作包括以下函数，参数CS是栈对象，ct是元素变量。
  - Push(Stack &CS, char ct)，将元素ct压入堆栈CS;
  - Pop(Stack &CS)，将栈CS的栈顶元素弹出
  - GetTop(Stack &CS, char &ct)，将栈顶元素放入到变量ct中

提问：在执行以下函数后，请按栈底到栈顶的顺序写出栈的元素排列，并写出变量CA的值。

```
POP (&SS);
Push(&SS, E’ );
POP(&SS);
POP(&SS);
GetTop(&SS, &CA) ;
Push(&SS, ‘F’ );
```

练习

- 设将整数1、2、3、4依次进栈，但只要出栈时栈非空，则可将出栈操作按任何次序夹入其中，请回答下有问题：

① 若入栈次序为push(1)，pop()，push(2，push(3)，pop()，pop( )，push(4)，pop( )，则出栈的数字序列为什么？

② 能否得到出栈序列423和432？并说明为什么不能得到或如何得到。

③ 请分析1、2、3、4的24种排列中，哪些序列可以通过相应的入、出栈操作得到。

练习

- 当利用大小为N的数组顺序存储一个栈时,假定用top=N表示栈空，则向这个栈插入一个元素时，首先应执行(　)语句修改top指针.

  a) top++

  b) top--

  c) top=0

  d) top=N-1

练习

- 假定利用数组a[N]顺序存储一个栈，top表示栈顶指针，top=-1表示栈空，并已知栈未满，当元素X进栈时所执行的操作为（）。

  a) a[--top]=x

  b) a[top--]=x

  c) a[++top]=x

  d) a[top++]=x

练习

- 指出下列程序段的功能是什么？

```
(1) void demo1( seqstack *s )
{
    int i;  arr[64];   n=0;
    while (  !stackempty(s)   )
            arr[n++] = pop(s);
    for(  i=0;  i<n;  i++  )
            push(  s,  arr[i]  );
  }
```

练习

```
(2) void demo2( seqstack *s, int m)
{
        seqstack t;    int i;
        initstack(t);
       while(  ! Stackempty(s)  )
                    if(    ( i = pop(s))  !=m ) push( t, i );
      While(! Stackempty(t) )
       {
            i = pop(t);
            push(s,i);
      }
   }
```

练习

- 有中缀表达式为(-3)*8/2^2，试描述利用栈结构将其转化为后缀表达式并求值的过程。