第3章 栈

- ADT栈
- 栈的存储结构
  - 用数组实现栈
  - 用指针实现栈
- 栈的应用

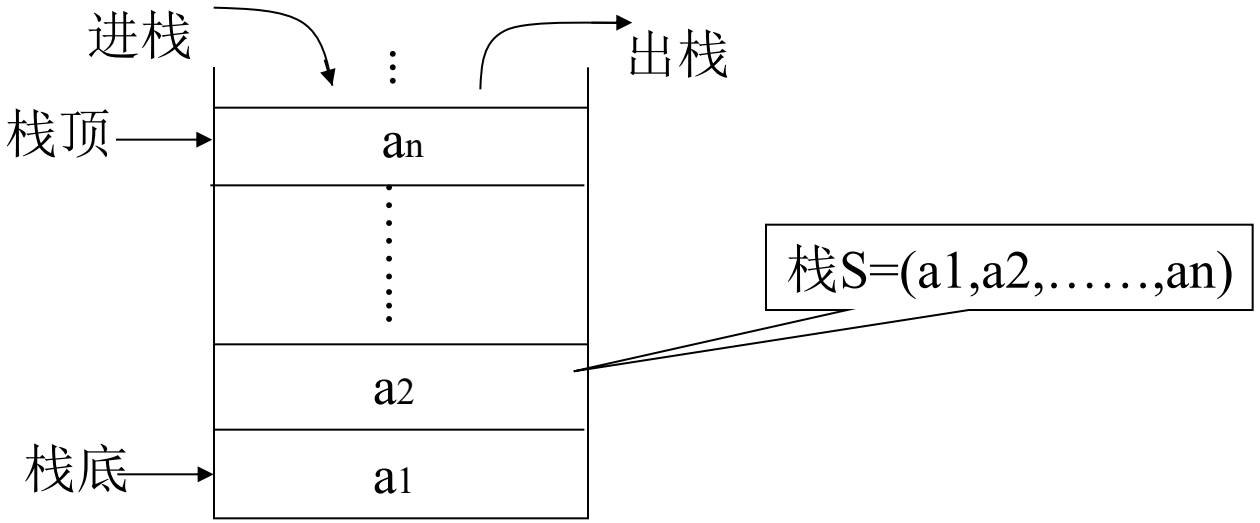
3.1 ADT栈（Stack）

- 栈是一种特殊的线性表，是操作受限的线性表，称其为限定性数据结构。
- 定义：限定仅在表首进行插入或删除操作的线性表，表首称为栈顶，表尾称为栈底，不含任何元素的空表称空栈。
- 特点：先进后出（FILO）或后进先出（LIFO）

例

- 栈的操作示例

假设栈S=(a1，a2，a3，…an)，则a1称为栈底元素，an为栈顶元素。栈中元素按a1，a2，a3，…an的次序进栈，退栈的第一个元素应为栈顶元素。



3.1 ADT栈（Stack）

- 栈上定义的常用的基本运算
  - **StackEmpty(S): 测试栈S是否为空**
  - **StackFull(S):测试栈S是否已满**
  - **StackTop(S): 返回栈S的栈顶元素**
  - **Push(x，S):在栈S的栈顶插入元素x，即将元素x入栈**
  - **Pop(S):删除并返回栈S的栈顶元素，简称为抛栈**

3.1.1 栈应用的简单例子

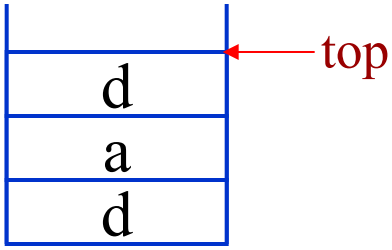
例1、程序编译时的表达式或字符串的括号匹配问题。

- 例如，算术表达式(x*(x+y)-z)，其中位置1和4处有左括号，而位置8和11处有右括号，满足配对要求。
- 但算术表达式 (x+y)*z)(，其中位置8处的右括号没有可与之配对的左括号，而位置9处的左括号没有可与之配对的右括号。
- 这里要在栈的基本运算的基础上定义算术表达式（字符串）expr中的圆括号配对检查运算

```
void Parenthesis(char *expr)
```

例2、回文游戏：顺读与逆读字符串一样（不含空格）

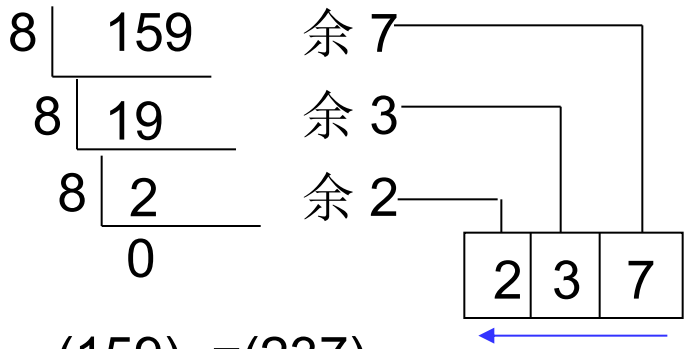
1. 读入字符串
2. 去掉空格（原串）
3. 压入栈
4. 原串字符与出栈字符依次比较
   若不等，非回文
   若直到栈空都相等，回文



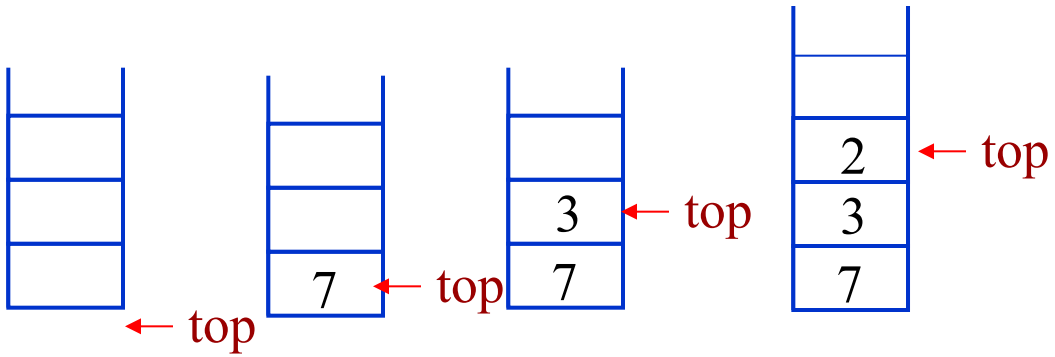
字符串：“madam im adam”

例3、多进制输出：

例 把十进制数159转换成八进制数



$(159)_{10}=(237)_8$



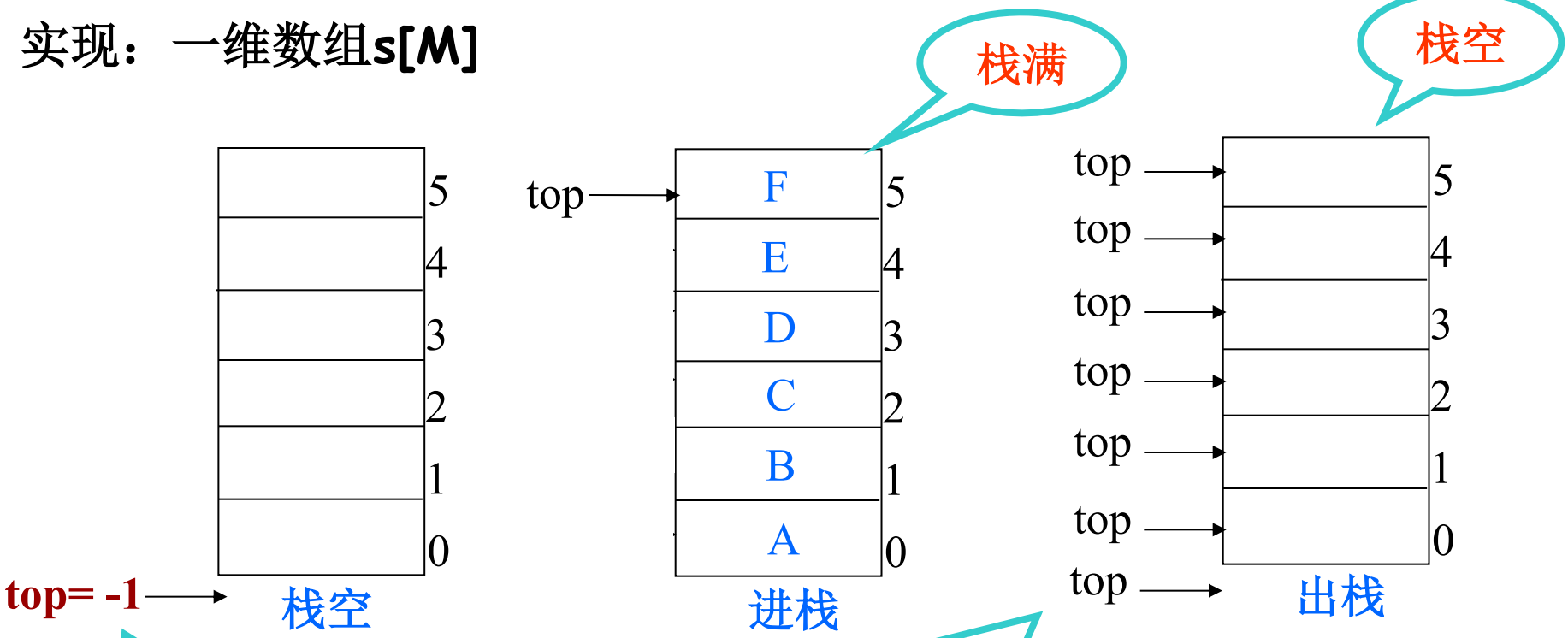
3.2 栈的存储结构

3.2.1 用数组实现栈——顺序栈

- 由于栈是操作受限的线性表，因此线性表的存储结构对栈也适应。
- 栈的顺序存储结构简称为顺序栈，可用数组来实现顺序栈。
- 栈顶位置是随着进栈和退栈操作而变化，故需设置一个变量top来指示当前栈顶的位置。

3.2.1 用数组实现栈——顺序栈

实现：一维数组s[M]



栈顶指针top,指向实际栈顶后的空位置，初值为-1

设数组维数为M
top= -1,栈空，此时出栈，则下溢（underflow)
top=M-1,栈满，此时入栈，则上溢（overflow)

3.2.1 用数组实现栈——顺序栈

- 用数组实现的栈结构**Stack**定义如下：

```
typedef  struct astack   *Stack;
typedef  struct astack {
     int  top;                    /* 栈顶位置*/
     int  maxtop;                /*栈顶位置的最大值*/
     StackItem *data;            /*栈元素数组*/
} Astack;
```

顺序栈上的基本运算的实现

1、创建空栈

创建一个容量为**size**的栈，实现方法如下：

```
Stack StackInit (int size)
{
    Stack S=malloc(sizeof *S);
    S->data=malloc (size*sizeof(StackItem));
    S->maxtop=size;
    S->top=-1;
    return S;
}
```

2、判断栈是否为空

```
int StackEmpty (Stack S)
{
   return S->top<0;
}
```

3、判断栈是否已满

```
int StackFull (Stack S)
{
   return S->top==S->maxtop;
}
```

4、取栈顶元素

栈顶元素存储在data[top]中。

```
StackItem StackTop (Stack S)
{
   if(StackEmpty(S))    Error("Stack is empty");
   else    return S->data[S->top];
}
```

5、进栈

新栈顶元素**x**应存储在data[top+1]中。

```
void Push(StackItem x,Stack S)
{
   if(StackFull(S))    Error("Stack is full");
   else    S->data[++S->top]=x;
}
```

6、出栈

删除栈顶元素后，新栈顶元素在data[top-1]中。

```
StackItem Pop(Stack S)
{
  if(StackEmpty(S))    Error("Stack is empty");
  else    return S->data[S->top--];
}
```

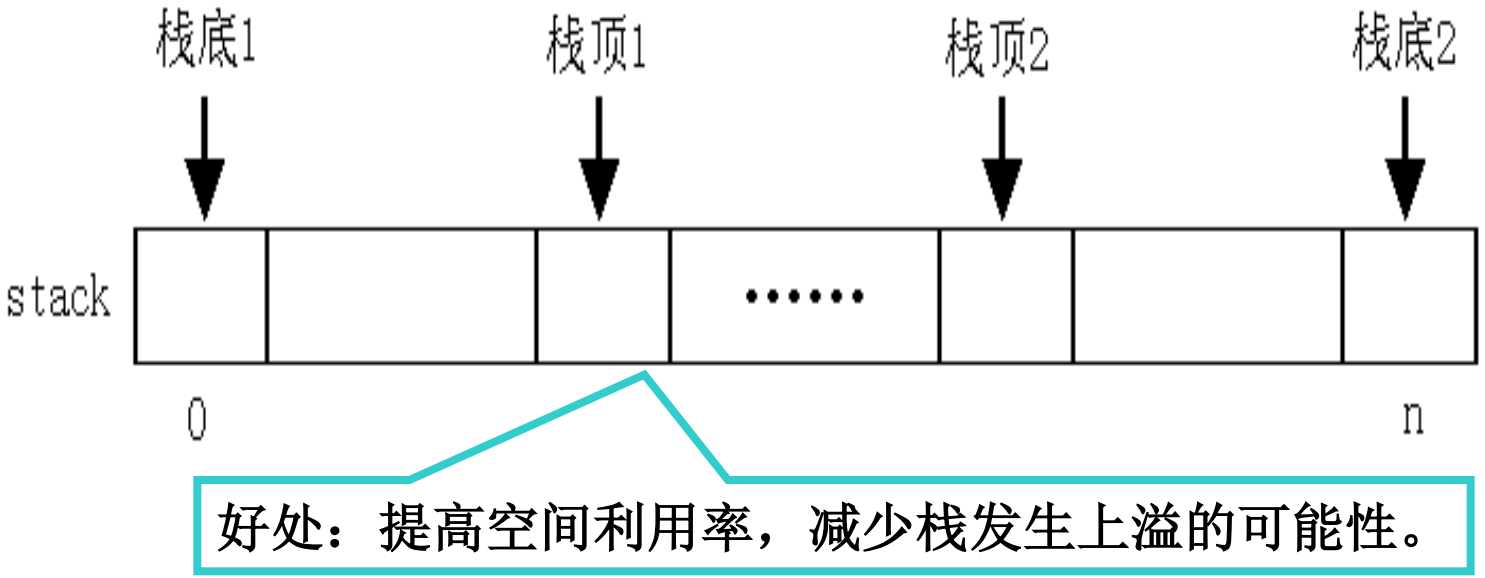
用数组实现栈的优缺点

- 优点
  - 所列的6个基本运算都可在O(1)的时间里完成，效率高。
- 缺点
  - 为了使每个栈在算法运行过程中不会溢出，通常要为每个栈预置一个较大的栈空间。另一方面，由于各个栈的实际大小在算法运行过程中不断变化。经常会发生其中一个栈满，而另一个栈空的情形，空间利用率低。

3.2.2 两个栈共用一个数组的实现

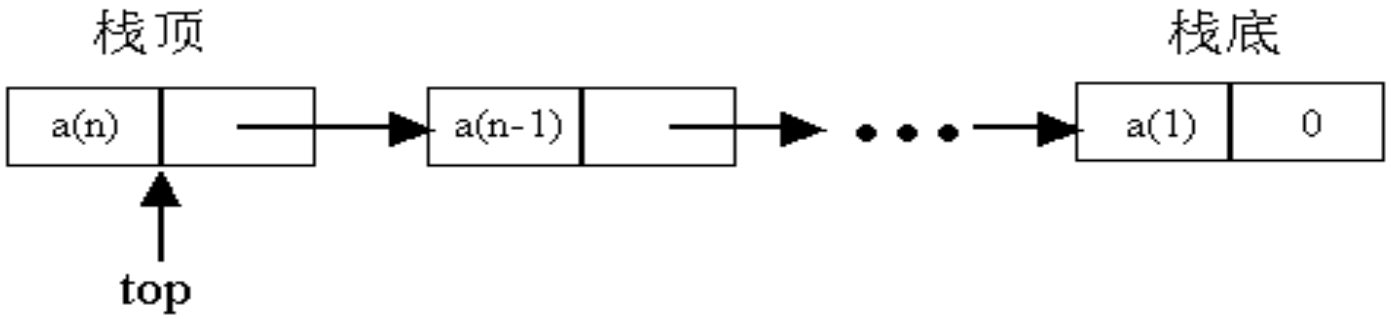
2个栈共享一个数组stack[0..n]：

- 利用栈底位置不变的特性，可以将2个栈的栈底分别设在数组stack的两端。然后各自向数组stack的中间伸展，如图所示。



3.2.3 用指针实现栈—链栈

- 在算法中要用到多个栈时，最好用链式存储结构存储栈，即用指针实现栈。用这种方式实现的栈也成为链栈。
- 链栈是操作受限的单链表，其插入和删除操作仅限制在表头位置上进行。链栈通常用不带头结点的单链表来实现。栈顶指针就是链表的头指针。



- 链栈的结点类型定义：

```
typedef struct  snode *slink;
typedef struct  snode
{  StackItem  element;
   slink  next;
}StackNode;
```

- 产生一个新结点的函数**NewStackNode( )**

```
slink NewStackNode( )
{
    slink p;
    if ((p=malloc(sizeof(StackNode)))==0)
       Error("Exharsted memory.");
    else return p;
}
```

- 用指针实现的链栈**Stack**定义如下：

```
typedef struct lstack *Stack;
typedef struct lstack
{
    slink  top;     //栈定点指针
}Lstack;
```

链栈上的基本运算的实现

1、创建空栈

将**top**置为空指针，创建一个空栈，实现方法如下：

```
Stack StackInit ( )
{
    Stack S=malloc(sizeof *S);
    S->top=0;
    return S;
}
```

2、判断栈是否为空

检测指向栈顶的指针**top**是否为空指针，实现方法如下：

```
int StackEmpty (Stack S)
{
   return S->top==0;
}
```

3、判断栈是否已满

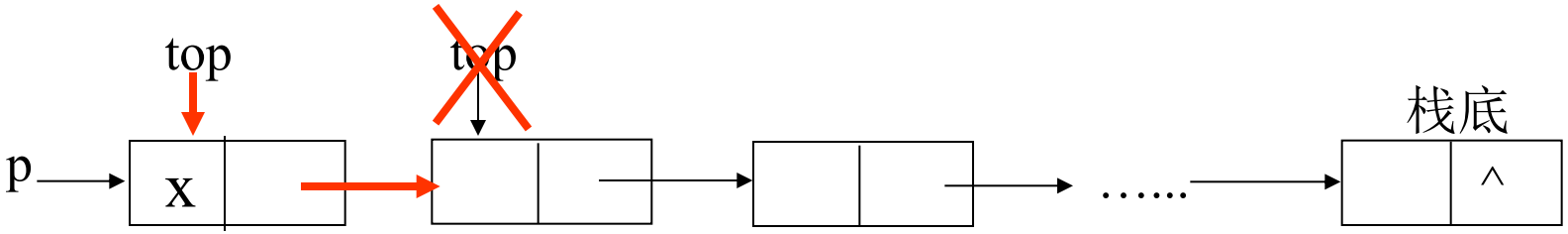
为栈S试分配一个新结点，检测栈空间是否已满。

```
int StackFull (Stack S)
{
   return StackMemFull( );
}
int StackMemFull( )
{
   slink p;
   if((p=malloc(sizeof(StackNode)))==0    return 1;
   else  {  free(p);   return 0;}
}
```

4、返回栈顶元素

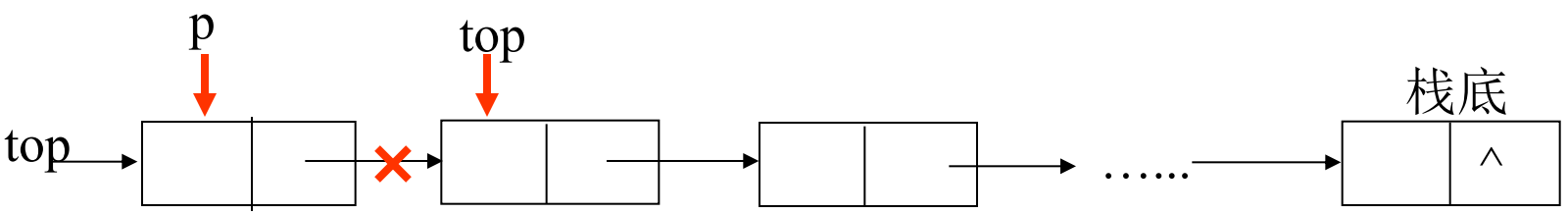
```
StackItem StackTop (Stack S)
{
   if(StackEmpty(S))     Error("Stack is empty");
   else     return S->top->element;
}
```

5、入栈



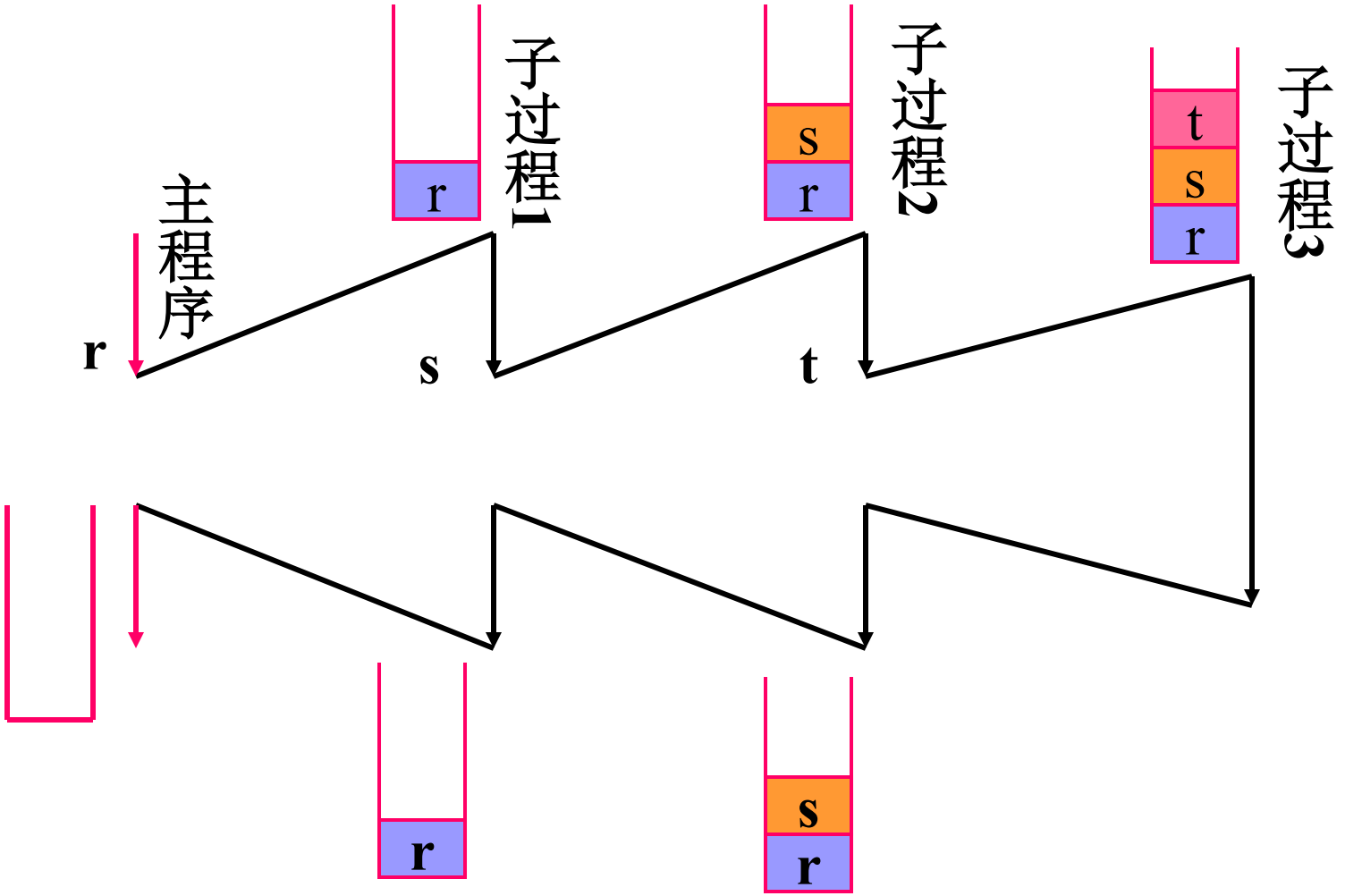

```
void Push (StackItem x, Stack S)
{
   slink p;
   if(StackFull(S))   Error("Stack is full");
   p=NewStackNode( );
   p->element=x;
   p->next=S->top;
   S->top=p;
}
```

6、出栈




```
StackItem Pop (Stack S)
{
   slink p;  StackItem x;
   if(StackEmpty(S))   Error("Stack is empty");
   x=S->top->element;
   p=S->top;
   S->top=p->next;
   free(p);
   return x;
}
```

- 3.3 栈的应用
  - 例1 过程的嵌套调用



- 例2 递归过程及其实现
  - 定义：函数直接或间接的调用自身叫递归
  - 实现：建立递归工作栈

例 递归的执行情况分析（假设初始w=3）


```
void    print(int  w)
{     int  i;
      if ( w!=0)
      {    print(w-1);
           for (i=1; i<=w; ++i)
                printf("%3d,",w);
           printf("/n");
      }
}
```

运行结果：

1，
2，2，
3，3，3，

递归调用执行情况如下：

主程序

w=3;
print(w)

(1)

结束

w
3

print(2);

(2) 输出：3, 3, 3

w
2

print(1)

(3) 输出：2, 2

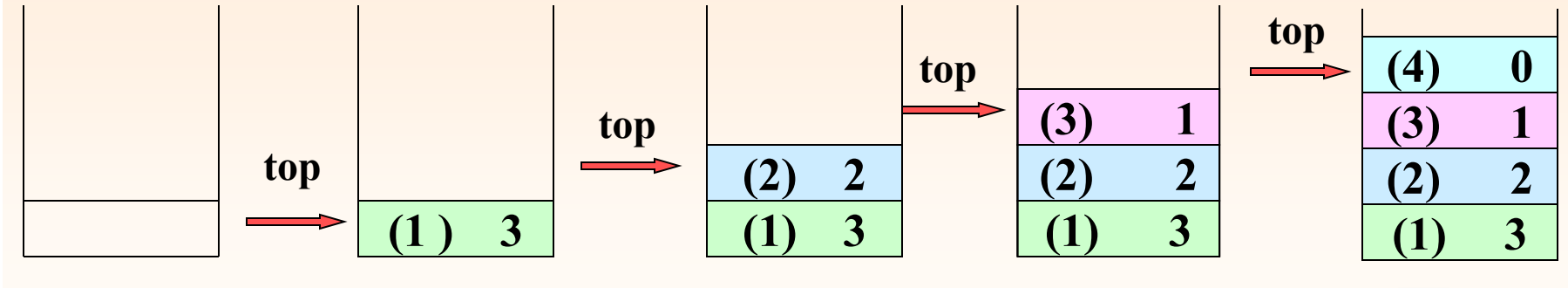
w
1

print(0);

(4)输出：1

w
0

返回



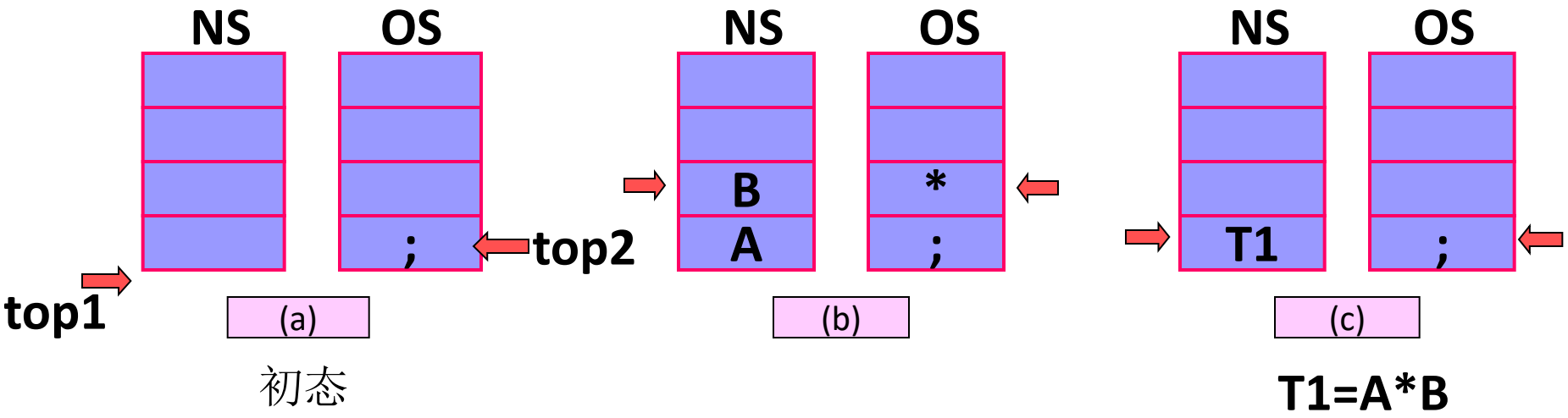
Back

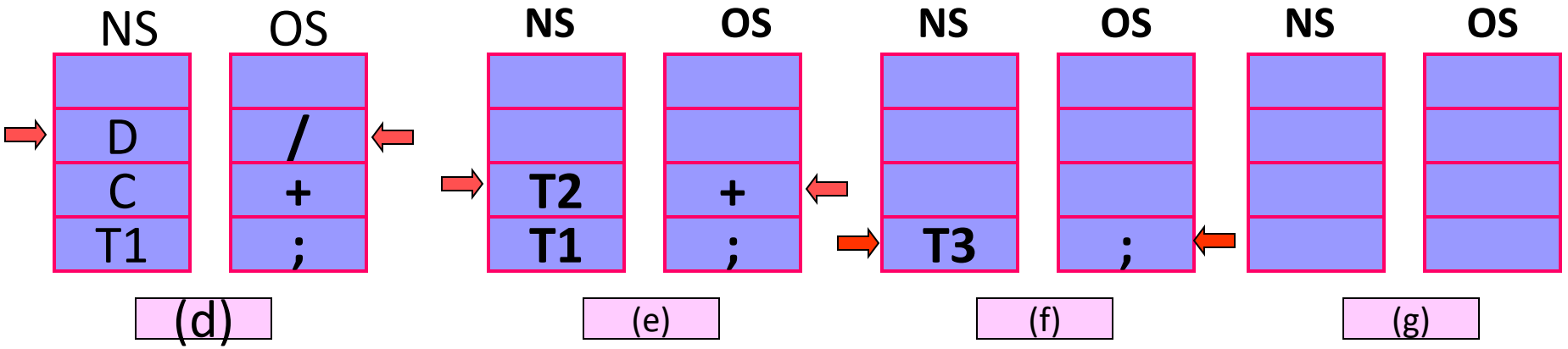
A*B + C/D;

- 思想：从左到右扫描表达式，若当前读入的是操作数，则进操作数（NS）栈，若读入的符号是运算符，应进行判断：

- 若是“（”，进运算符栈；若是“）”，当运算符栈顶是“（”，则弹出栈顶元素，继续扫描下一符号。否则当前读入符号暂不处理，从操作数栈弹出两个操作数，从运算符栈弹出一个运算符，生成运算指令，结果送入操作数栈，继续处理当前读入符号。
- 若读入的运算符的优先级大于运算符栈顶的优先级，则进运算符栈，继续扫描下一符号；否则从操作数栈顶弹出两个操作数，从运算符栈弹出一个运算符，生成运算指令，把结果送入操作数栈。继续处理刚才读入的符号。
- 若读入的是“；”，且运算符栈顶的符号也是“；”时，则表达式处理结束。从操作数栈弹出表达式结果。

A*B + C/D;


NS
OS
top1
top2
;
B
A
*
;
T1
;
(a)
(b)
(c)
初态
T1=A*B

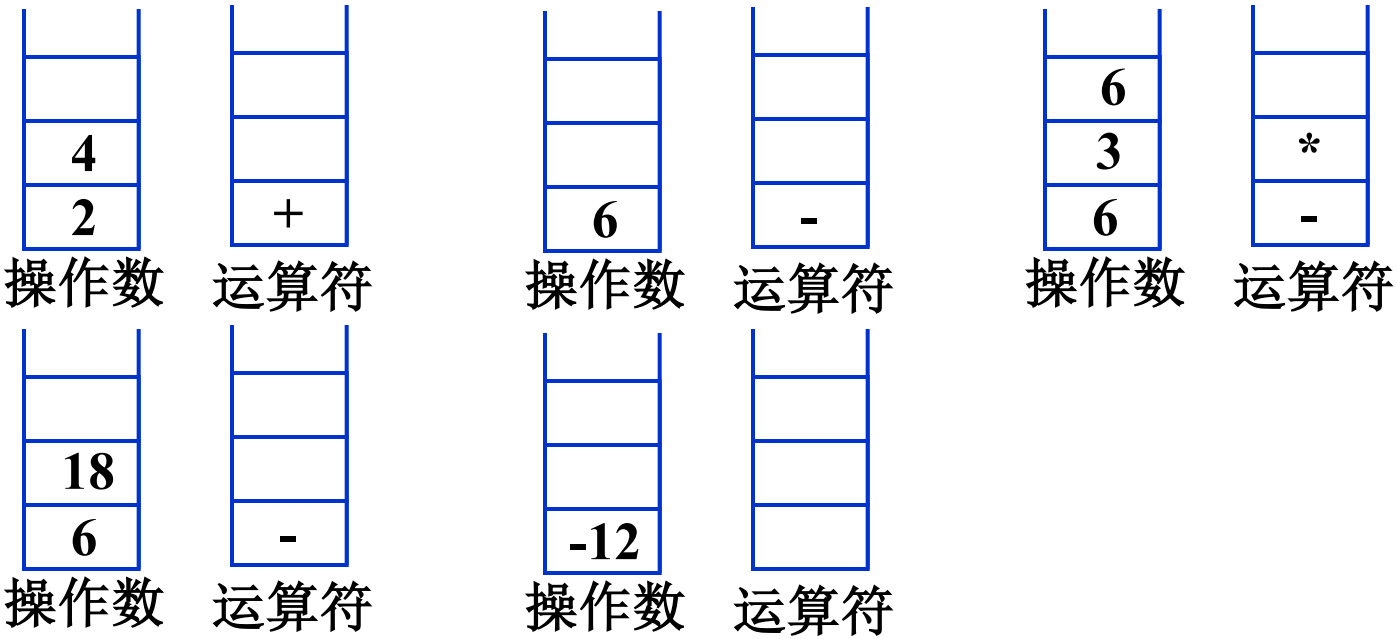

NS
OS
D
C
T1
/
+
;
T2
T1
+
;
T3
;
(d)
(e)
(f)
(g)
T2=C/D
T3=T2+T1

- 例3 表达式求值

中缀表达式	后缀表达式（RPN)
a*b+c	ab*c+
a+b*c	abc*+
a+(b*c+d)/e	abc*d+e/+

中缀表达式：操作数栈和运算符栈

☞ 计算 2+4-3*6

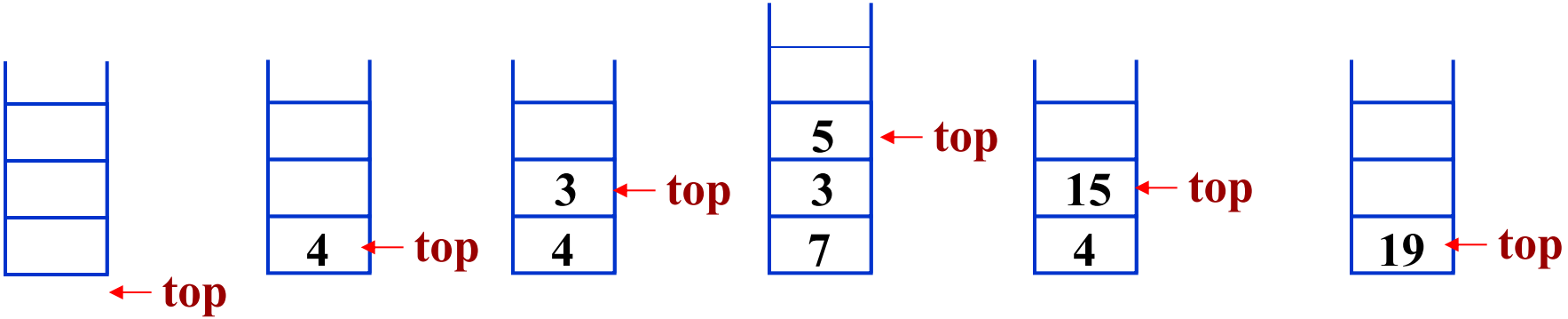


后缀表达式求值步骤：

1、读入表达式一个字符
2、若是操作数，压入栈，转4
3、若是运算符，从栈中弹出2个数，将运算结果再压入栈
4、若表达式输入完毕，栈顶即表达式值；
　　若表达式未输入完，转1

☞ 计算 4+3*5

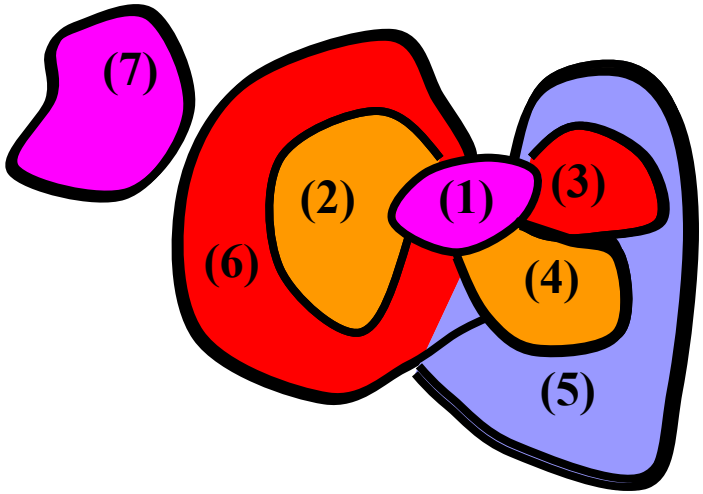
后缀表达式：435*+



例4 地图四染色问题

R [7][7]

	1	2	3	4	5	6	7
1	0	1	1	1	1	1	0
2	1	0	0	0	0	1	0
3	1	0	0	1	1	0	0
4	1	0	1	0	1	1	0
5	1	0	1	1	0	1	0
6	1	1	0	1	1	0	0
7	0	0	0	0	0	0	0



1# 紫色

2# 黄色

3# 红色

4# 绿色

1	2	3	4	5	6	7
1	2	3	2	4	3	1

- 例：已知字符串的内容为b%-y-3*y↑2，利用栈的进栈和退栈操作将其转换成by-%3y2 ↑ *-试给出进栈和退栈的操作序列。

设进栈用 I 表示，出栈用 O 表示

操作序列：

I O I I I O O O I I O I I O I I O O O O

本章小结

- 理解栈是满足LIFO存取原则的表。
- 熟悉定义在抽象数据类型栈上的基本运算。
- 掌握用数组实现栈的步骤和方法。
- 掌握用指针实现栈的步骤和方法。
- 理解用栈解决实际问题的方法。