北京航空航天大學
BEIHANG UNIVERSITY

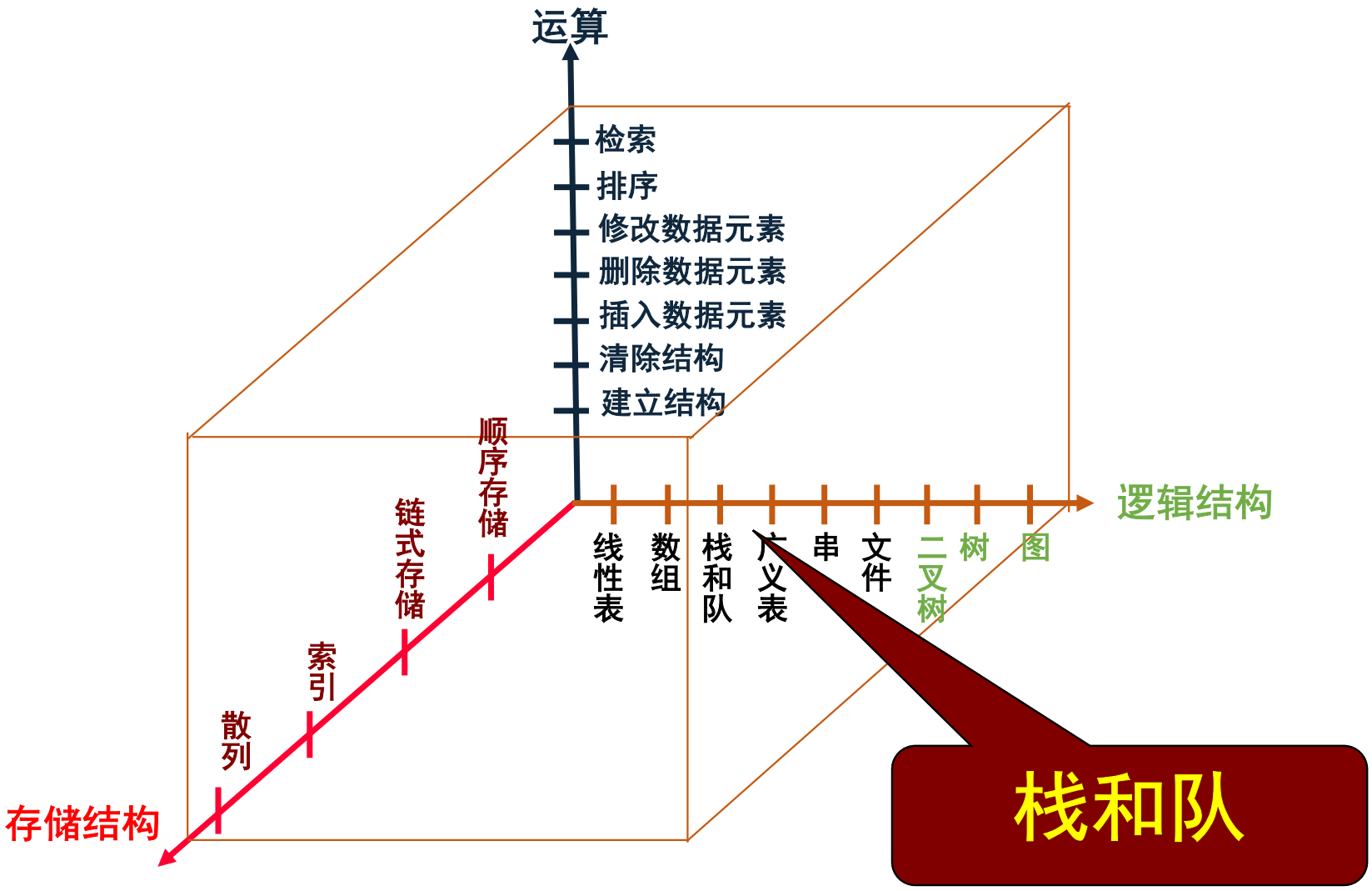
数据结构与程序设计
(Data Structure and Programming)

数据结构

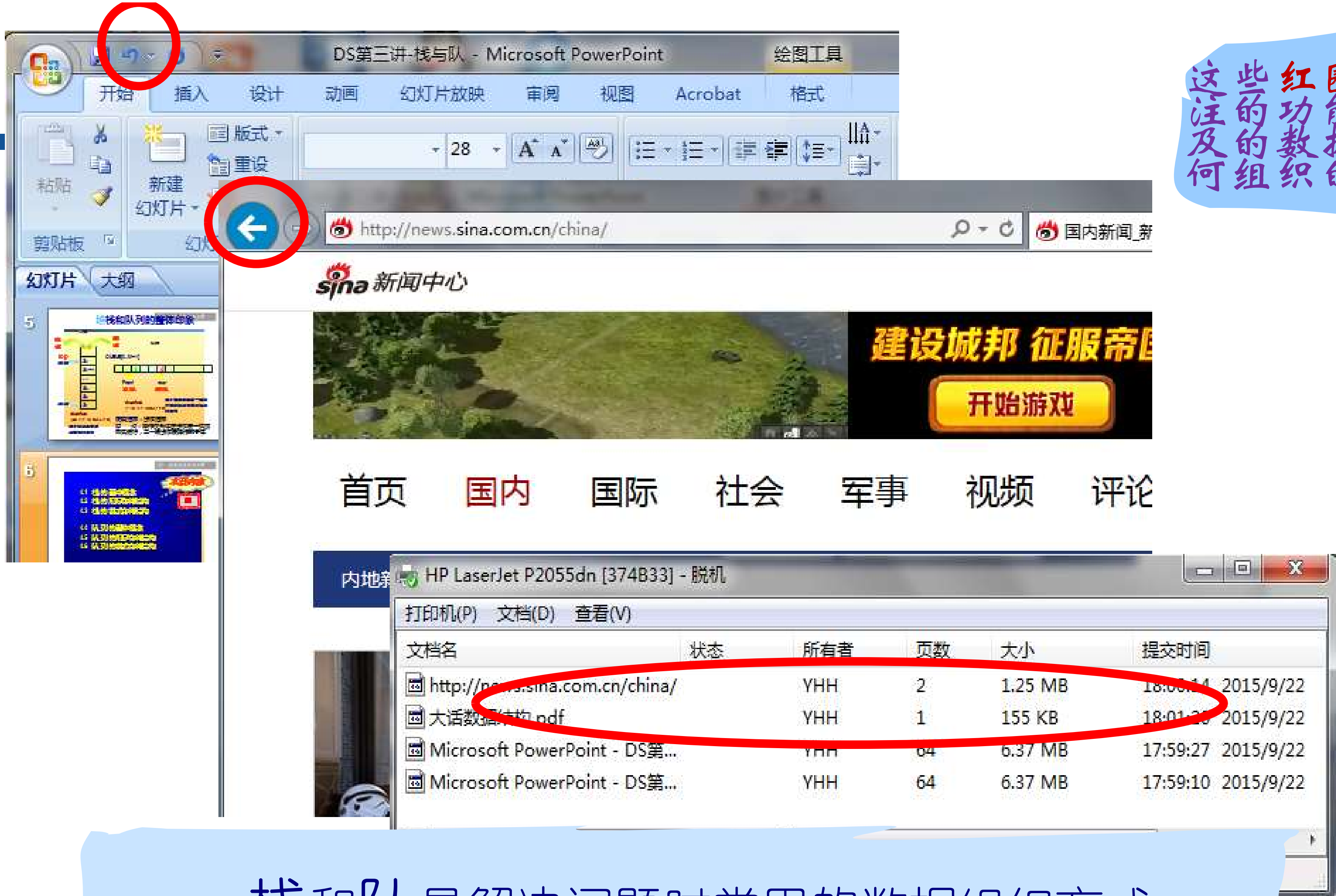
栈与队
(Stack and Queue)

北航计算机学院 晏海华

数据结构的基本问题空间



第三讲
栈和队列



这些红圈所标注的功能所涉及的数据是如何组织的？

栈和队是解决问题时常用的数据组织方式

在计算机领域

程序设计

回溯法

递归程序的执行过程

编译程序

变量的存储空间的分配

表达式的翻译与求值计算

栈

操作系统

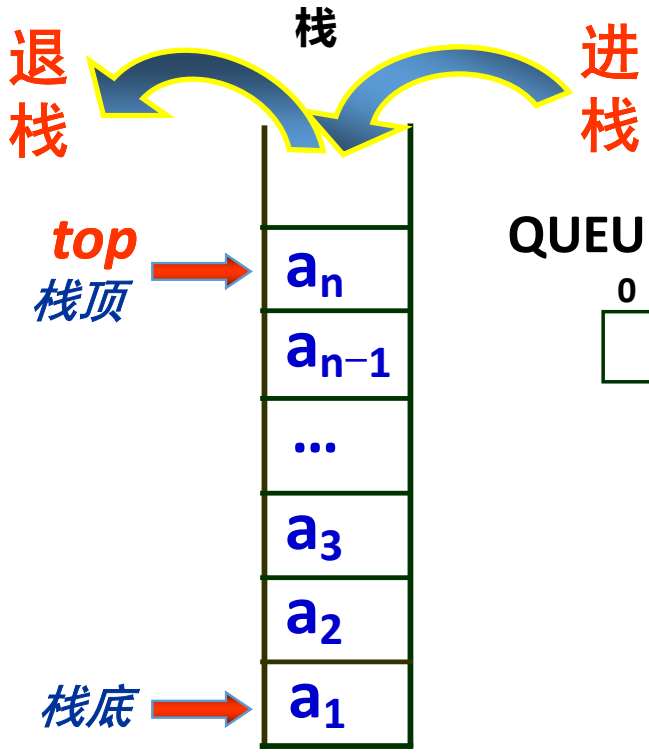
作业调度、进程调度

后续章节

二叉树的层次遍历……

队

栈和队的整体印象

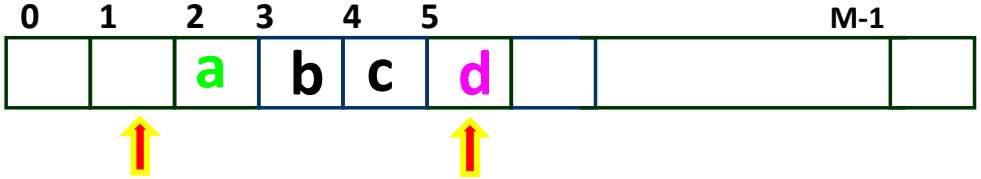


后进先出
(Last In First Out,**LIFO**)

适于处理具有递归结构的数据

队列

QUEUE[0..M–1]



先进先出
(First In First Out,**FIFO**)

适于存放需要按一定次序依次处理但尚未处理的数据

逻辑结构：线性结构
特　　点：操作仅允许在线性表一端或两端进行，是一般线性表操作的子集

本章内容

重点

3.1 栈的基本概念
3.2 栈的顺序存储结构
3.3 栈的链式存储结构

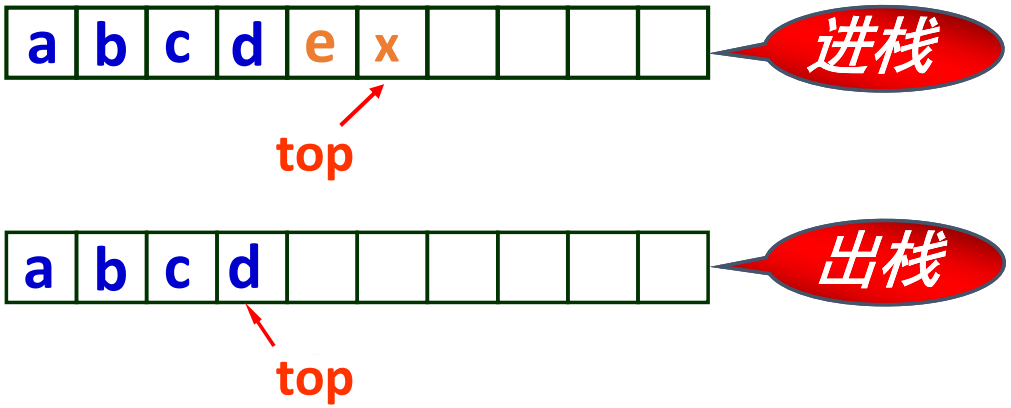
3.4 队的基本概念
3.5 队的顺序存储结构
3.6 队的链式存储结构

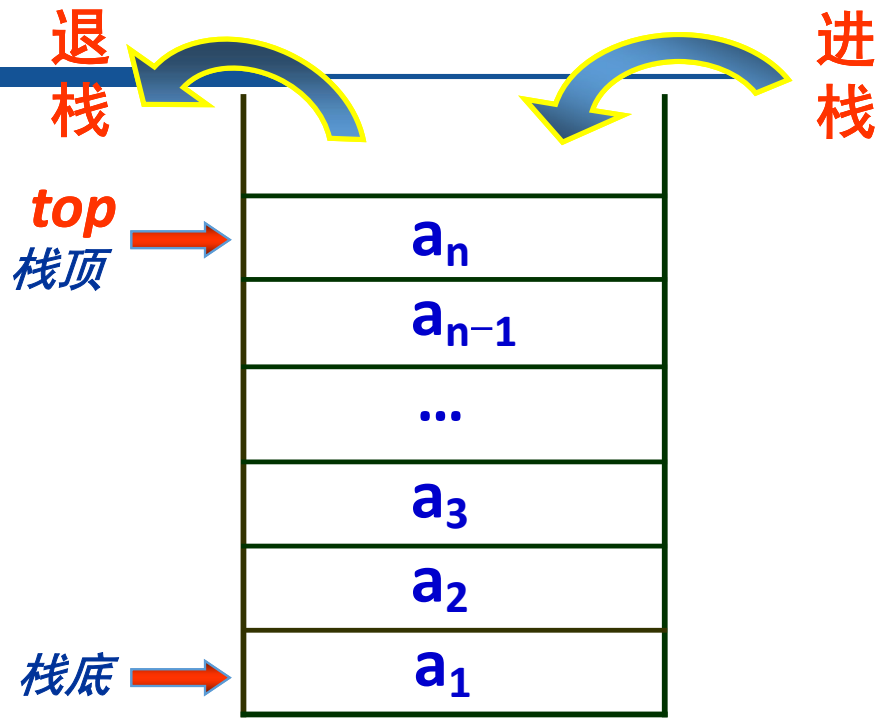
3.1 栈的基本概念

后进先出

(一) 栈的定义

栈(Stack) 是一种只允许在表的一端进行插入操作和删除操作的线性表。允许操作的一端称为栈顶，栈顶元素的位置由一个称为**栈顶位置**的变量给出。当表中没有元素时，称之为**空栈**。





栈的示意图

栈的特点：

1）元素间呈线性关系

2）插入删除在一端进行

3）后进先出，先进后出

LIFO（Last-In-First-Out）

(二) 栈的基本操作

1. 插入（进栈、入栈、压栈） √
2. 删除（出栈、退栈、弹出） √
3. 测试栈是否为空 √
4. 测试栈是否已满
5. 检索当前栈顶元素

特殊性

1. 其操作仅仅是一般线性表的操作的一个子集。
2. 插入和删除操作的位置受到限制。

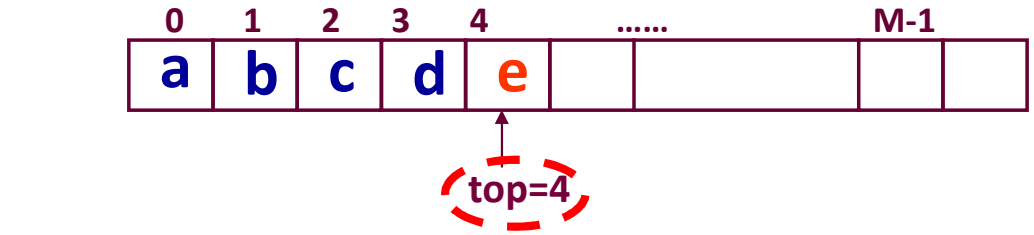
栈的基本操作

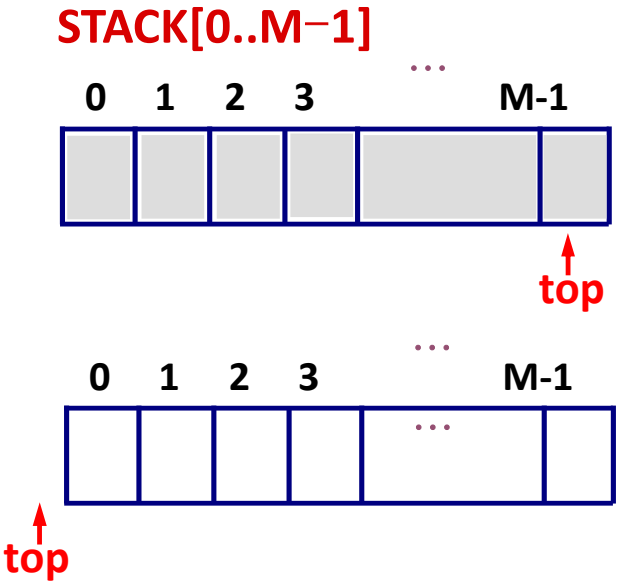
- void push(Stack s, ElemType e) //压一个元素进栈
- ElemType pop(Stack s) //从栈中弹出一个元素
- int isEmpty(Stack s) //判断栈是否为空
- int isFull(Stack s) //判断栈是否已满
- ElemType getTop(Stack s) //获取栈顶元素

3.2 栈的顺序存储结构（顺序栈）

(一) 构造原理

描述栈的顺序存储结构最简单的方法是利用一维数组 STACK[0..M–1] 来表示，同时定义一个整型变量（不妨取名为top）给出栈顶元素的位置。





数组：静态结构
栈：动态结构

溢出

上溢 — 当栈已满时做入栈操作。 (top=M–1)
下溢 — 当栈为空时做出栈操作。 (top=–1)

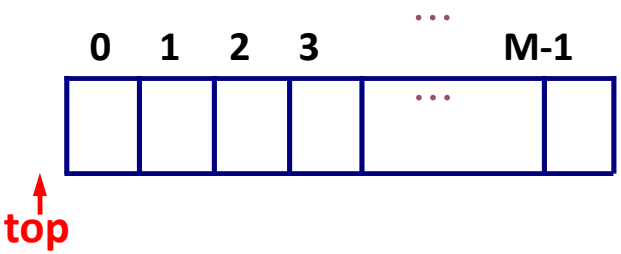
类型定义

```
#define MAXSIZE    1000
ElemType  STACK[MAXSIZE];
int  Top;
```

初始时,Top= –1

由于Top变量需要在多个函数间共享，为了保持函数接口简洁，在此定义为全局变量。

(二) 顺序栈的基本算法



1. 初始化堆栈

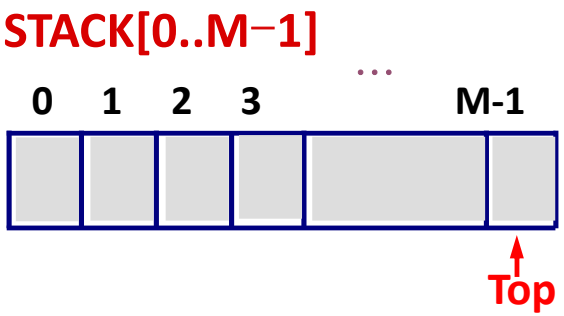
```
void  initStack( )
{
      Top= –1;
}
```

2. 测试堆栈是否为空

栈空,返回1,否则,返回0。

```
int  isEmpty( )
{
      return Top== –1;
}
```

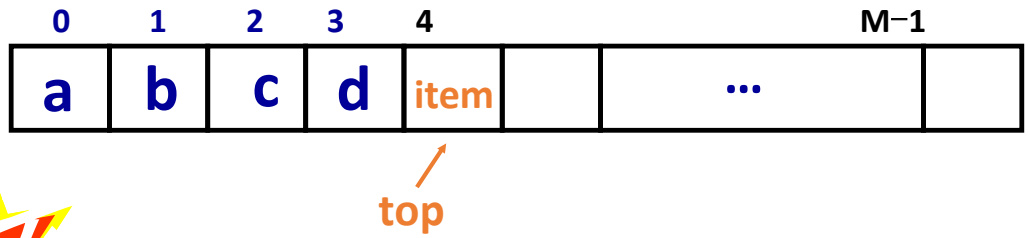
3. 测试堆栈是否已满




```
int isFull( )
{
    return Top==MAXSIZE−1;
}
```

栈满,返回1,否则,返回0。

4. 进栈算法



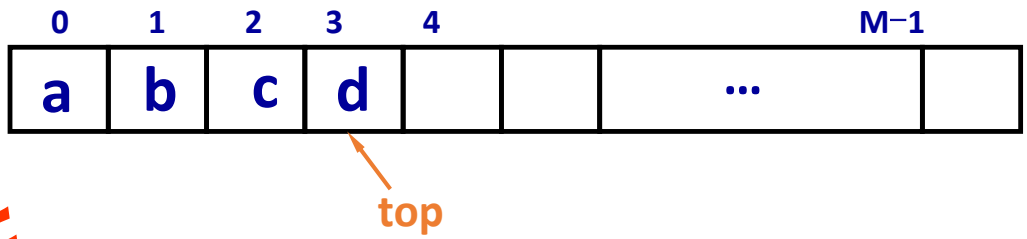

```
void  push( ElemType s[ ],  ElmeType item )
{
      if( isFull() )
            Error(“Full Stack!”);
      else
            s[++Top]=item;

}
```

入栈成功

```
void  Error(char s[])
{
      printf(“%s\n”, s);
      exit( -1);
}
```

5. 出栈算法



算法

```
ElemType pop( ElemType s[ ])
{
    if(isEmpty())
        Error("Empty Stack!");
    else
                                    出栈成功
        return s[Top−−];
}
```

(三) 多栈共享连续空间问题

(以两个栈共享一个数组为例)

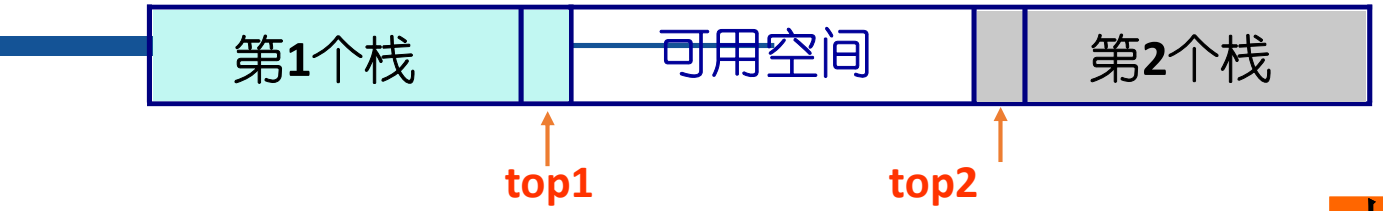
STACK[0..M−1]

Top1、Top2 分别给出第1个与第2个栈的栈顶元素的位置。

进栈

当i=1时，将item 插入第1个栈，

当i=2时，将item 插入第2个栈。

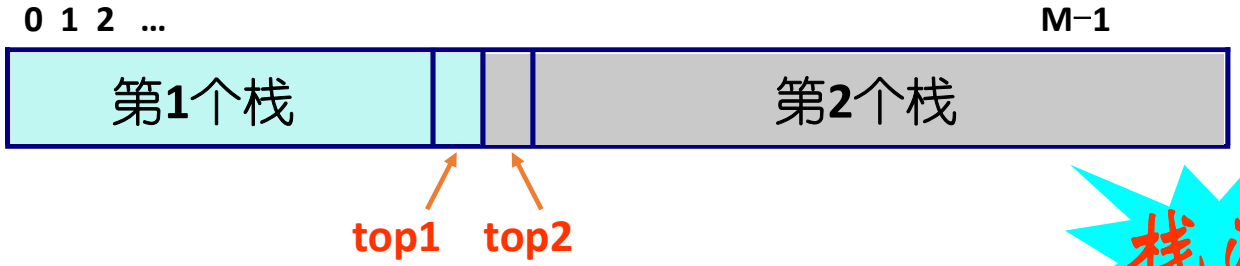


栈1

top1++;
STACK[top1]=item;

栈2

top2−−;
STACK[top2]=item;

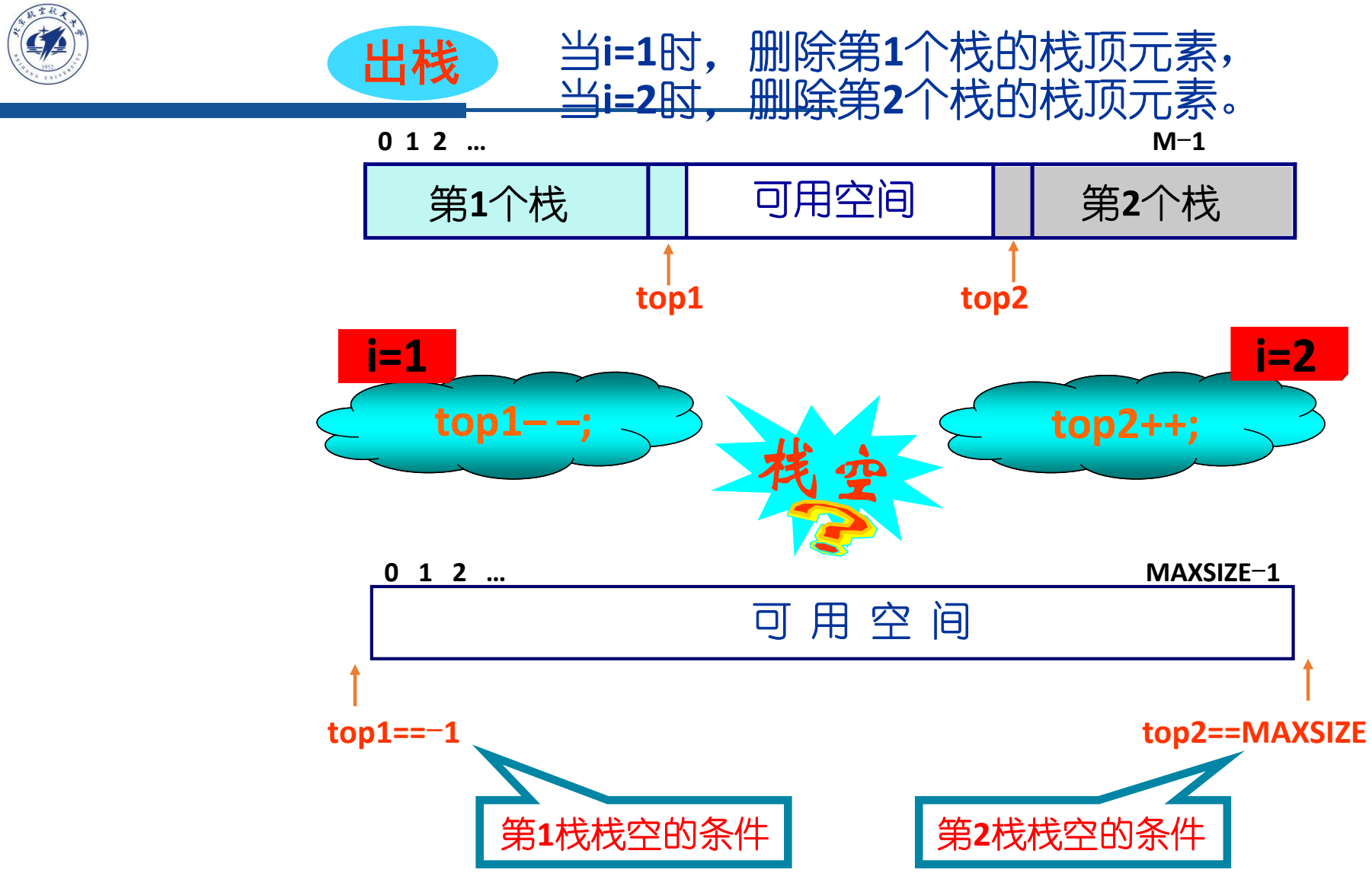


栈满

栈满的条件是

top1==top2−1


```
void  push( ElemType s[ ],  int i, ElemType item )
{
    if(top1==top2–1)          /* 栈满 */
        Error(“Full Stack!”);
    else{
        if(i==1)                    /* 插入第1个栈 */
            STACK[++top1]=item;
        else                        /* 插入第2个栈 */
            STACK[– –top2]=item;
        return;
     }
}
```


出栈
当i=1时，删除第1个栈的栈顶元素，
当i=2时，删除第2个栈的栈顶元素。
0 1 2 ...
M−1
第1个栈
可用空间
第2个栈
top1
top2
i=1
i=2
top1−−;
top2++;
栈空
?
0 1 2 ...
MAXSIZE−1
可 用 空 间
top1==−1
top2==MAXSIZE
第1栈栈空的条件
第2栈栈空的条件

```
EleType  pop( ElemType s[ ],  int i)
{
      if(i==1)
          if(top1==–1)
              Error(“Empty Stack1!”);
          else
              return s[top1– –];

      else
          if(top2==MAXSIZE)
              Error(“Empty Stack2!”);
          else
              return s[top2++];
}
```

对第一个栈进行操作

对第二个栈进行操作

3.3 栈的链式存储结构

(一) 构造原理

链接栈
链栈

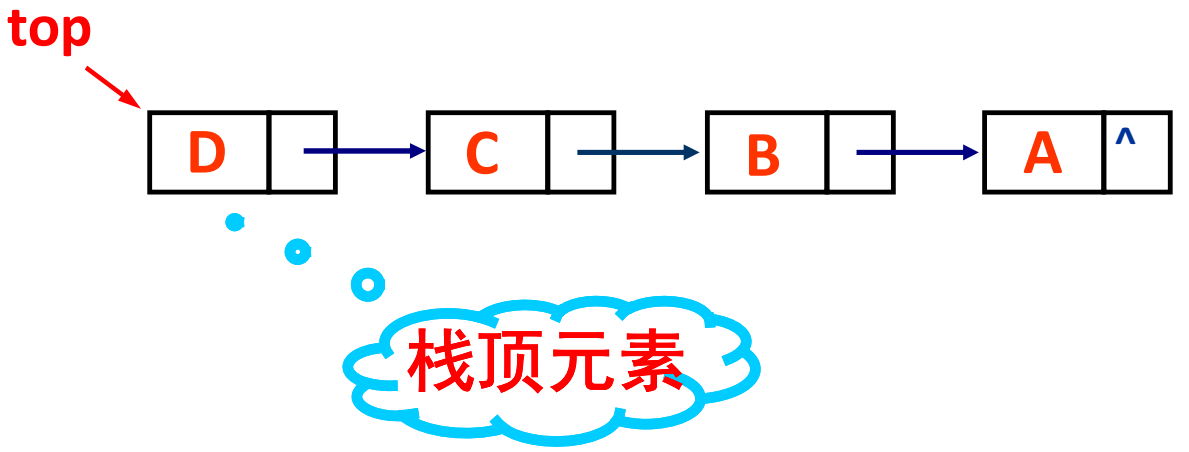
链接栈就是用一个线性链表来实现一个栈结构，同时设置一个指针变量（这里不妨仍用**top**表示）指出当前栈顶元素所在链结点的位置。栈为空时，有**top=NULL**。

例

在一个初始为空的链接栈中依次插入元素

A，B，C，D

以后，栈的状态为



链栈是一种特殊的链表，其结点的插入（进栈）和删除（出栈）操作始终在链表的头。

类型定义

```
struct  node {
        SElmeType  data;
        struct  node  *link;
};
typedef  struct node  *Nodeptr;
typedef  struct node  Node;
Nodeptr Top; //即为链表的头结点指针
```

由于Top变量需要在多个函数间共享，为了简化操作在此定义为**全局变量**。

(二) 链栈的基本算法

1. 栈初始化

```
void initStack( )
{
    Top=NULL;
}
```

思考？

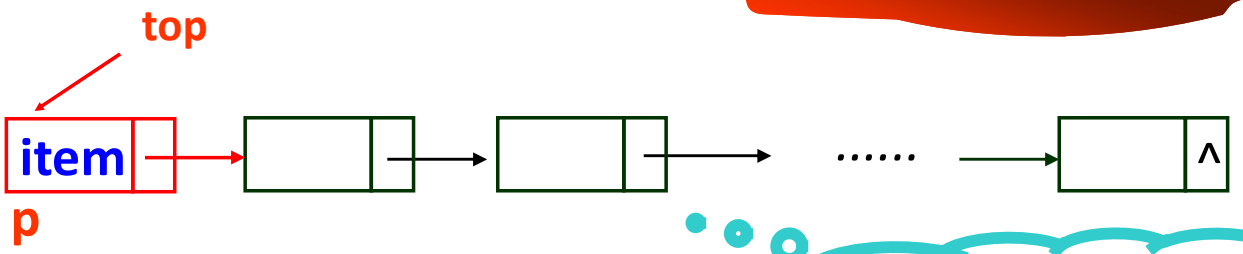
为什么不需要测试栈是否已满

2. 测试堆栈是否为空

```
int  isEmpty( )
{
    return Top==NULL;
}
```

3. 进栈算法

等效于在链表最前面插入一个新结点



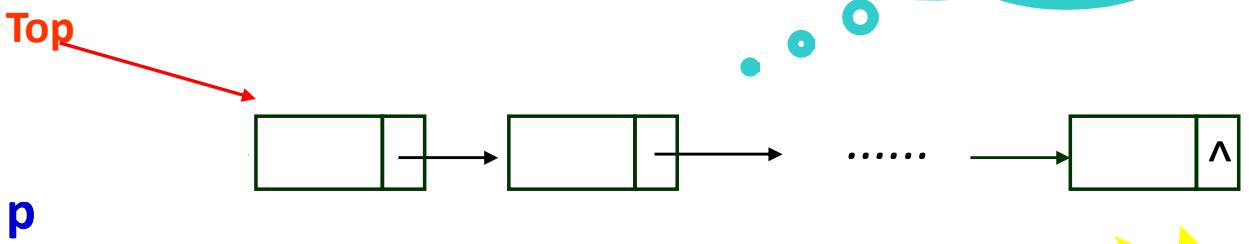
不必判断栈满

算法

```
void  push( ElemType item )
{   Nodeptr p;
    if( (p=(Nodeptr)malloc(sizeof(Node)))==NULL )
        Error("No memory!");
    else{
        p->data=item;          /*将item送新结点数据域*/
        p->link=Top;           /*将新结点插在链表最前面*/
        Top=p;                 /*修改栈顶指针的指向*/
    }
}
```

4. 出栈算法

仍然要判断栈空！



算法

```
ElemType  pop( )
{    Nodeptr p;
     ElemType item;
    if ( isEmpty() )
         Error("Empty Stack!");                    /* 栈中无元素*/
    else{
         p=Top;                          /* 暂时保存栈顶结点的地址*/
         item=Top->data;                 /*保存被删栈顶的数据信息*/
         Top=Top->link;                  /* 删除栈顶结点 */
         free(p);                        /* 释放被删除结点*/
         return item;                        /* 返回出栈元素*/
    }
}
```

问题3.1：计算器（表达式计算）

【问题描述】

从标准输入中读入一个整数算术运算表达式，如24 / (1 + 2 + 36 / 6 / 2 - 2) * (12 / 2 / 2)= ，计算表达式结果，并输出。

要求：

1、表达式运算符只有+、-、*、/，表达式末尾的’=’字符表示表达式输入结束，表达式中可能会出现空格；

2、表达式中会出现圆括号，括号可能嵌套，不会出现错误的表达式；

3、出现除号/时，以整数相除进行运算，结果仍为整数，例如：5/3结果应为1。

【输入形式】

从键盘输入一个以’=’结尾的整数算术运算表达式。

【输出形式】

在屏幕上输出计算结果（为整数）。

【样例1输入】

24 / (1 + 2 + 36 / 6 / 2 - 2) * (12 / 2 / 2)=

【样例1输出】

18

问题3.1：问题分析

对于一般形式的表达式（通常称为中缀表达式(infix)）：

$$a+b*c+(d*e+f)/g$$

后缀表达式的最大好处是没有括号，也不用考虑运算符的优先级！

在（计算机）计算表达式的值时面临的主要问题有：

- 运算符有优先级
- 括号会改变计算的次序

为了方便表达式的（计算机）计算，波兰数学家Lukasiewicz在20世纪50年代发明了一种将运算符写在操作数之后的表达式表示方式（称为后缀表达式(postfix)，或逆波兰表示，Reverse Polish Notation,RPN）

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

中缀到后缀的转换规则

规则：从左至右遍历中缀表达式中每个数字和符号：

- 若是数字直接输出，即成为后缀表达式的一部分；
- 若是符号：
  - 若是)，则将栈中元素弹出并输出，直到遇到“(”，“(”弹出但不输出；
  - 若是(，+，*等符号，则从栈中弹出并输出优先级高于当前的符号，直到遇到一个优先级低的符号；然后将当前符号压入栈中。
    （优先级+，-最低，*，/次之，“(”最高）
- 遍历结束，将栈中所有元素依次弹出，直到栈为空。

后缀表达式计算

规则：从左至右遍历后缀表达式中每个数字和符号：

- 若是数字直接进栈；
- 若是运算符（+，-，*，/），则从栈中弹出两个元素进行计算（注意：后弹出的是左运算数），并将计算结果进栈。
- 遍历结束，将计算结果从栈中弹出（栈中应只有一个元素，否则表达式有错）。

今天许多编译器通常先将数学表达式转换成后缀表达式，然后再将后缀表达式转换成机器执行代码。

问题3.1：算法分析

算法：

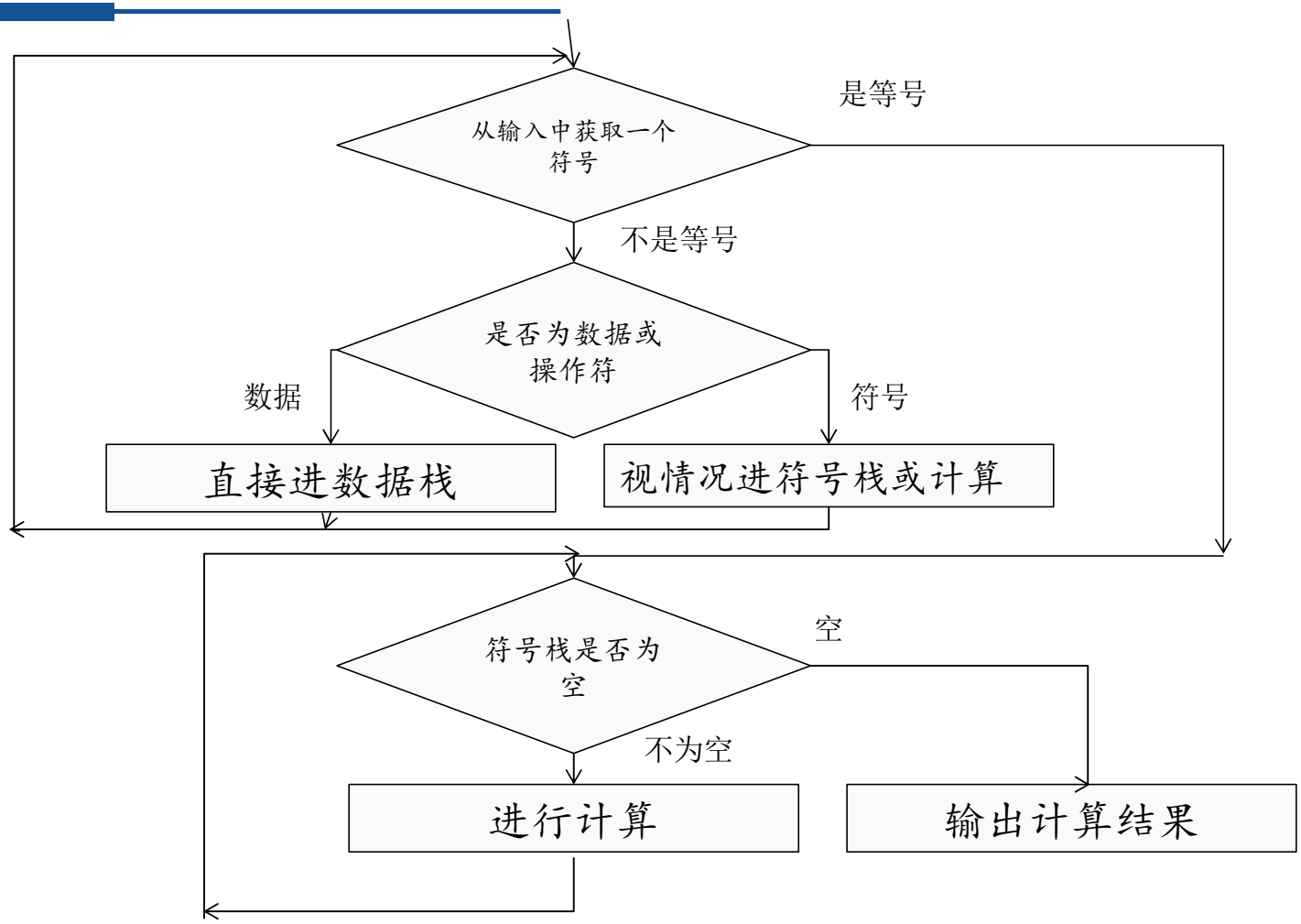
对于问题3.1我们没有必要象编译程序那样先将中缀表达式转换为后缀表达式，然后再进行计算。

为此，可设两个栈，一个为**数据栈**，另一个为**运算符栈**，在转换中缀表达式的同时进行表达式的计算。**主要思路为：当一个运算符出栈时，即与数据栈中的数据进行相应计算，计算结果仍存至数据栈中。算法如下：**

1. 从输入（中缀表达式）中获取一项：

- *若是数字则压入数据栈中；*
- *若是符号：*
  - *若是)，则将符号栈中元素弹出**并与数据栈中元素进行计算、计算结果放回数据栈中**，直到遇到“(”，“(”弹出但不计算；*
  - *若是(，+，*等符号，则从符号栈中弹出优先级高于当前的符号**并与数据栈中元素进行计算、计算结果放回数据栈中**，直到遇到一个优先级低的符号；然后将当前符号压入栈中。*
- *若是=号，将符号栈中所有元素依次弹出**并与数据栈中元素进行计算、计算结果放回数据栈中**，直到符号栈为空，此时数据栈中为计算结果；否则转1。*

问题3.1：算法分析


从输入中获取一个符号
是等号
不是等号
是否为数据或操作符
数据
符号
直接进数据栈
视情况进符号栈或计算
符号栈是否为空
空
不为空
进行计算
输出计算结果

枚举类型（enum）

为了使程序具有更好的扩展性，本问题实现中用到了**枚举**类型

枚举型变量的取值仅限于规定的一组值之一。

- 定义形式：

  enum *枚举名 { 值表 };*

  例：

  enum color { red, green, yellow, white, black }; /* 枚举值是标识符 */

- 枚举变量说明

  enum color chair;

  enum color suite[10];

枚举类型（续）

●在表达式中使用枚举变量

```
chair = red;
suite[5] = yellow;
if( chair = = green ) ...
```

注意：*对枚举变量的赋值并不是将标识符字符串传给它，而是把该标识符所对应的各值表中常数值赋与变量*。C语言编译程序把值表中的标识符视为从0开始的连续整数。另外，枚举类型变量的作用范围与一般变量的定义相同。

如：

```
enum color { red, green, yellow = 5, white, black };
```

则：

red=0, green=1, yellow=5, white=6, black=7

枚举类型（续）

- 枚举类型用途：
  - 枚举类型通常用来说明变量取值为有限的一组值之一，如：enum Boolean { FALSE, TRUE };
  - 用来定义整型常量，如： enum { SIZE=1024 };
  - 符号类型变量可用于数组下标。

问题3.1：代码实现

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
#define  MAXSIZE 100
typedef  int  DataType;
enum  symbol {NUM, OP, EQ,OTHER};//符号类型
enum  oper {EPT,ADD, MIN, MUL, DIV,  LEFT, RIGHT};
//运算类型及优先级
int Pri[]={-1,0,0,1,1,2,2}; //运算符优先级
union sym {
   DataType num;
   enum oper op;
} ; //符号值
enum symbol getSym( union sym *item);
void operate(enum oper op );//操作运算符
void compute(enum oper op ); //进行运算
```

```
int main()
{
   union sym item;
   enum symbol s;
   while( (s = getSym(&item)) ! = EQ) {
      if(s == NUM)
            pushNum(item.num);
      else if(s == OP)
            operate(item.op);
      else {
         printf("Error in the expression!\n");
         return 1;
      }
   }
   while(Otop >=0) //将栈中所有运算符弹出计算
      compute(popOp());
   if(Ntop == 0) //输出计算结果
      printf("%d\n", popNum());
   else
      printf("Error in the expression!\n");
   return 0;
}
```

```
void pushNum(DataType num);
DataType popNum();
void pushOp(enum oper op);
enum oper  popOp();
enum oper  topOp();
```

问题3.1：代码实现（续）*

```
enum symbol getSym( union sym *item)
{
   int  c, n;
   while((c = getchar()) != '=') {
      if(c >= '0' && c <= '9'){
         for(n=0; c >= '0' && c <= '9'; c= getchar())
            n = n*10 + c-'0';
         ungetc(c, stdin);
         item->num = n;
         return NUM;
      }
      else
         switch(c)  {
            case '+': item->op = ADD; return OP;
            case '-': item->op = MIN; return OP;
            case '*': item->op = MUL; return OP;
            case '/': item->op = DIV; return OP;
             case '(': item->op = LEFT; return OP;
             case ')': item->op = RIGHT; return OP;
            case ' ': case '\t': case '\n': break;
            default: return OTHER;
         }
   }
   return EQ;
}
```

```
void compute(enum oper op )
{
   DataType  tmp;
   switch(op) {
      case ADD:
         pushNum(popNum() + popNum()); break;
      case MIN:
         tmp = popNum();
         pushNum(popNum() - tmp); break;
      case  MUL:
         pushNum(popNum() * popNum()); break;
      case DIV:
         tmp = popNum();
         pushNum(popNum() / tmp); break;
   }
}
```

问题3.1：代码实现（续）

```
//数据栈操作
DataType Num_stack[MAXSIZE]; //数据栈
int Ntop=-1; //数据栈顶指示器，初始为空栈
void pushNum(DataType num)
{
   if(Ntop == MAXSIZE -1)
      error(“Data stack is full!”);
   Num_stack[++Ntop] = num;
}
DataType popNum()
{
   if(Ntop == -1)
      error(“Error in the expression!”);
   return Num_stack[Ntop--] ;
}
void error(char s[ ])
{
   fprintf(stderr, “%s\n”,s);
   exit(1);
}
```

```
//运算符栈操作
enum oper Op_stack[MAXSIZE];//符号栈
int  Otop=-1; //运算符栈顶指示器，初始为空栈
void pushOp(enum oper op)
{
   if(Ntop == MAXSIZE -1)
      error(“Operator stack is full!”);
   Op_stack[++Otop] = op;
}
enum operator  popOp()
{
   if(Otop != -1){
      return Op_stack[Otop--] ;
   }
   return EPT;
}
enum operator  topOp()
{
   return Op_stack[Otop];
}
```

问题3.1：

从本例中看出由于使用了**栈**这种数据结构，一方面简化了算法**复杂性**；另一方面程序具有很好的**可扩展性**（如增加新的优先级运算符非常方便）。

思考：修改该表达式计算程序，为其增加：%(**求余**)，>(**大于**)，<(**小于**)等运算符。运算符优先级照C语言中定义。

练习1

若以符号PUSH和POP分别表示对堆栈进行1次进栈操作与1次出栈操作，则对进栈序列 a, b, c, d, e， 经过PUSH, PUSH, PUSH, POP, PUSH, POP, POP, PUSH 以后，栈中状态如何？得到的出栈序列是什么？

c , d , b

练习2

若符号PUSH(k)表示整数k进栈， POP表示栈顶元素出栈，那么，请画出依次执行PUSH(1), PUSH(2), POP, PUSH(5), PUSH(7), POP, PUSH(6)以后堆栈的状态。

2 , 7

练习3

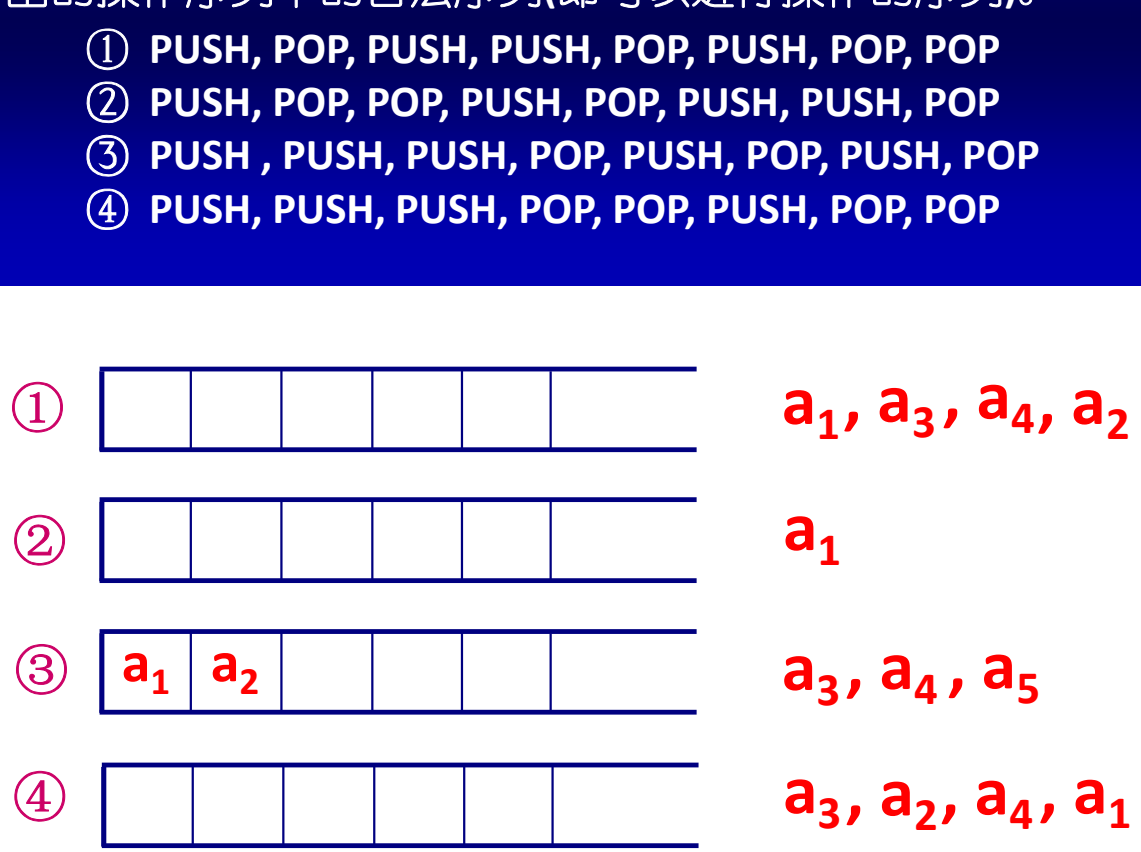
若符号PUSH和POP分别表示1次进栈与1次出栈操作，则进栈和出栈的操作序列可以表示为仅由PUSH和POP组成的序列。对于初态和终态均为空的堆栈，请分别指出下面给出的操作序列中的合法序列(即可以进行操作的序列)。

① PUSH, POP, PUSH, PUSH, POP, PUSH, POP, POP

② PUSH, POP, POP, PUSH, POP, PUSH, PUSH, POP

③ PUSH , PUSH, PUSH, POP, PUSH, POP, PUSH, POP

④ PUSH, PUSH, PUSH, POP, POP, PUSH, POP, POP



① a_1, a_3, a_4, a_2

② a_1

③ a_3, a_4, a_5

④ a_3, a_2, a_4, a_1

练习4

设有一顺序栈S，元素a, b, c, d, e, f依次进栈，如果6个元素出栈的顺序是b, d, c, f, e, a, 则栈的容量至少应该是（　　）

A、2　　B、3　　C、5　　D、6

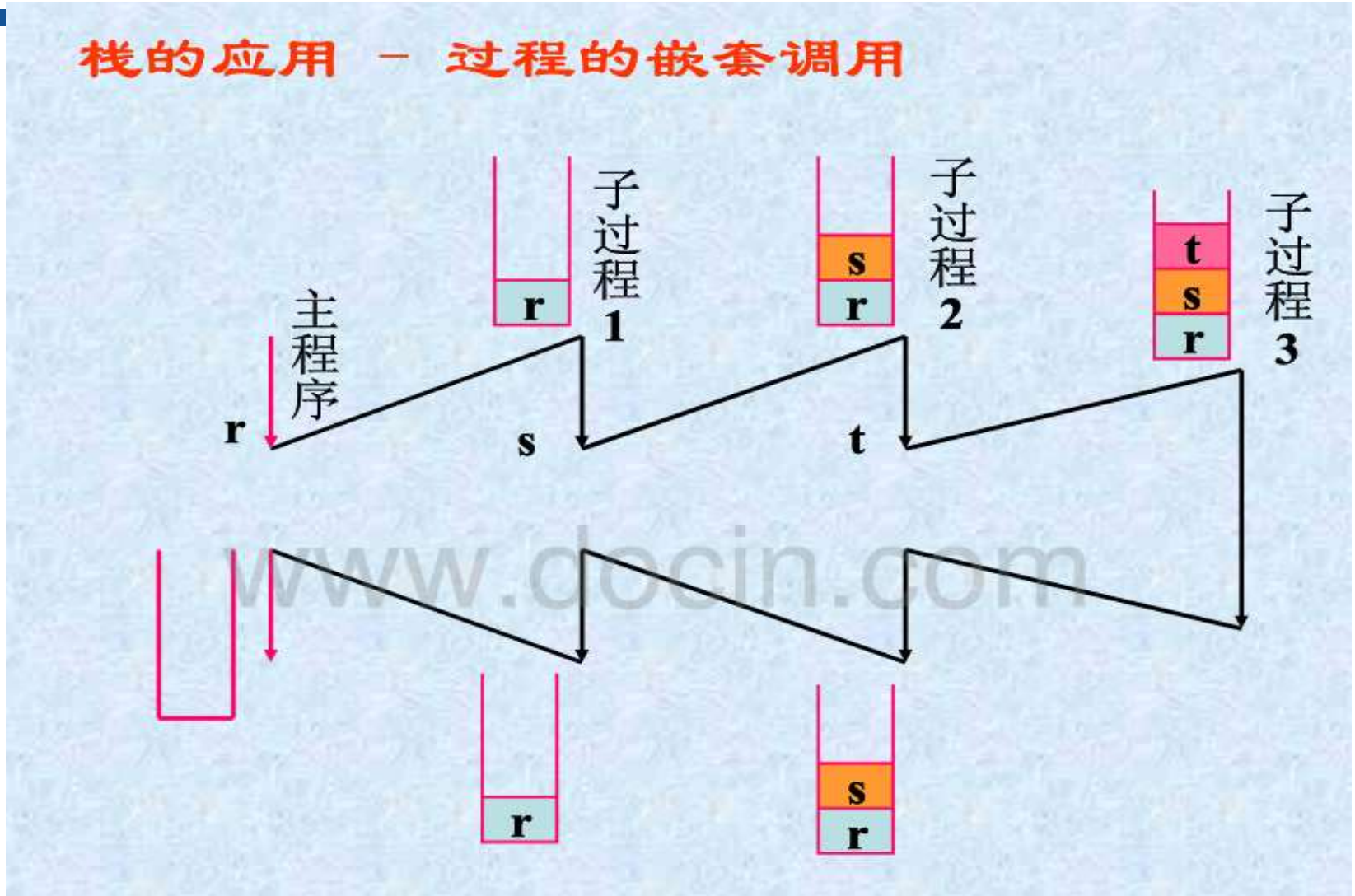
练习5

试将下列递归过程改造为非递归过程

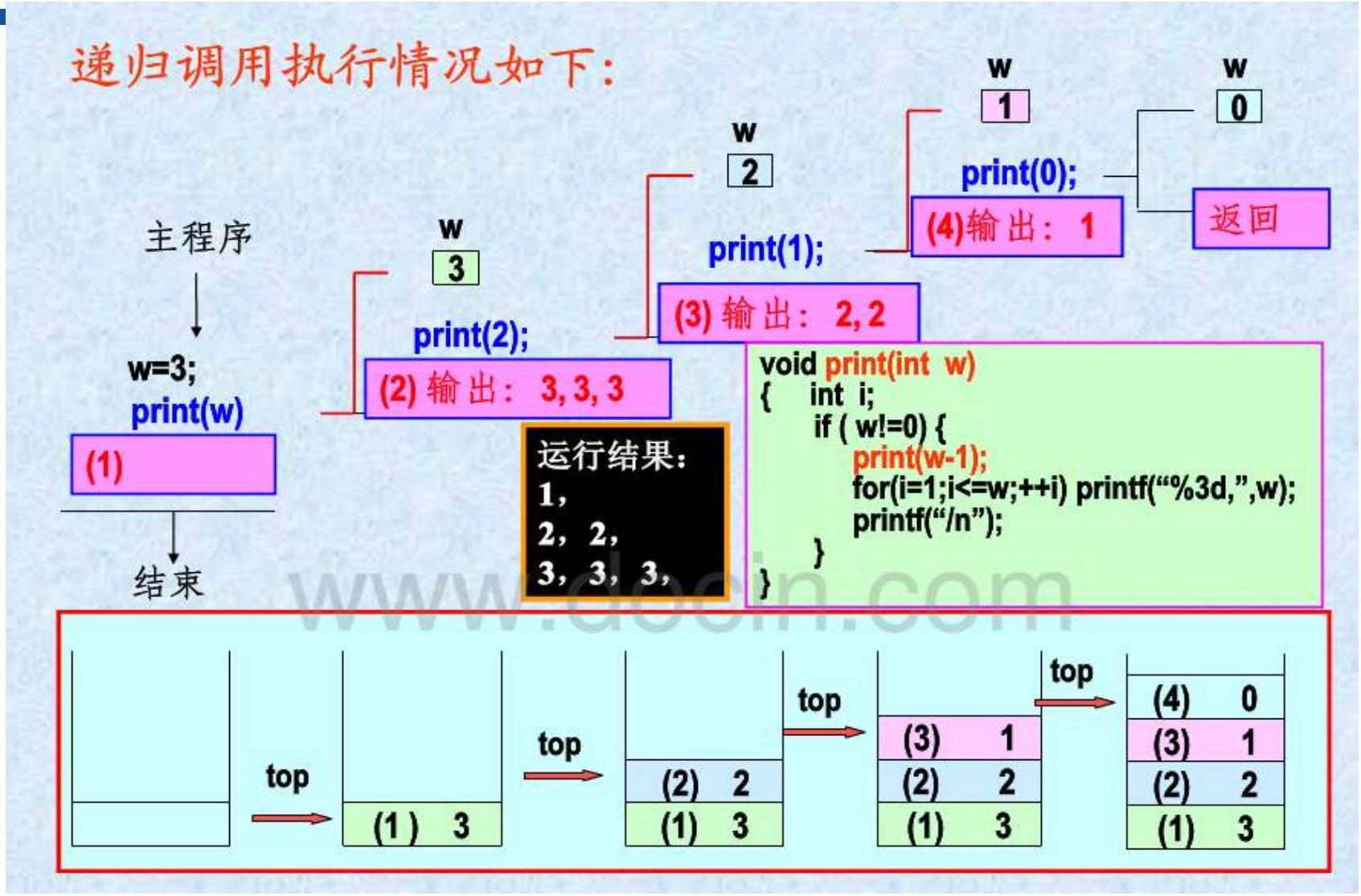
```
void process(int n)
{
    if (n>1)
    {
        printf("%d ", n);
        process(n-1);
    }
}
```

```
void process(int n)
{
    int i;
    i = n;
    while ( i > 1)
        printf("%d ",i--);
}
```

栈与嵌套的过程调用

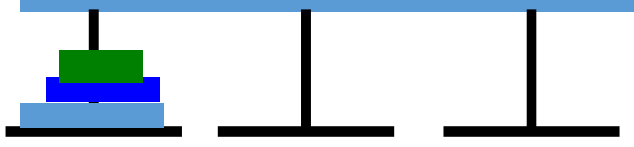


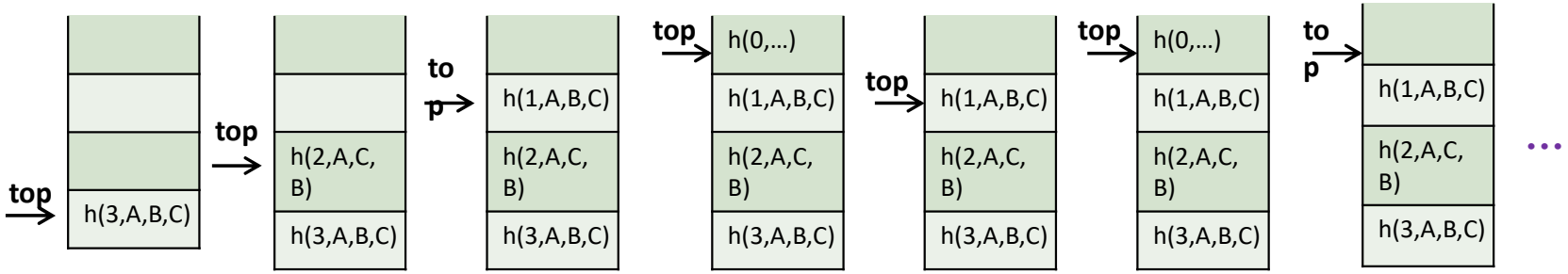
栈与递归


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
运行结果：
1,
2, 2,
3, 3, 3,
void print(int w)
{ int i;
if (w!=0) {
print(w-1);
for(i=1;i<=w;++i) printf("%3d,",w);
printf("/n");
}
}
top
(1) 3
(2) 2
(1) 3
(3) 1
(2) 2
(1) 3
(4) 0
(3) 1
(2) 2
(1) 3

栈与递归

```
汉诺塔(hanoi tower)游戏
void hanoi( int n, char x, char y, char z)
{
   if( n > 0 ) {
      hanoi(n-1, x, z, y);
      printf("MOVE %d: %c → %c\n", n, x, z);
      hanoi(n-1, y, x, z);
   }
}
```

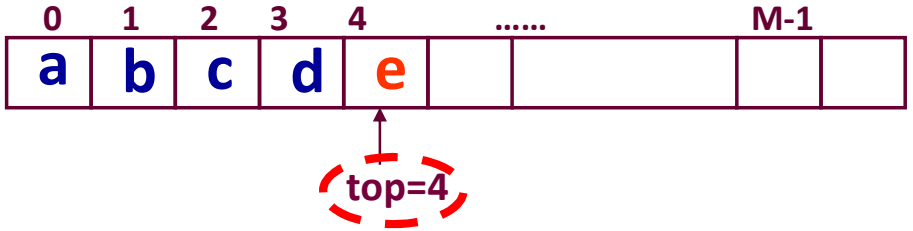


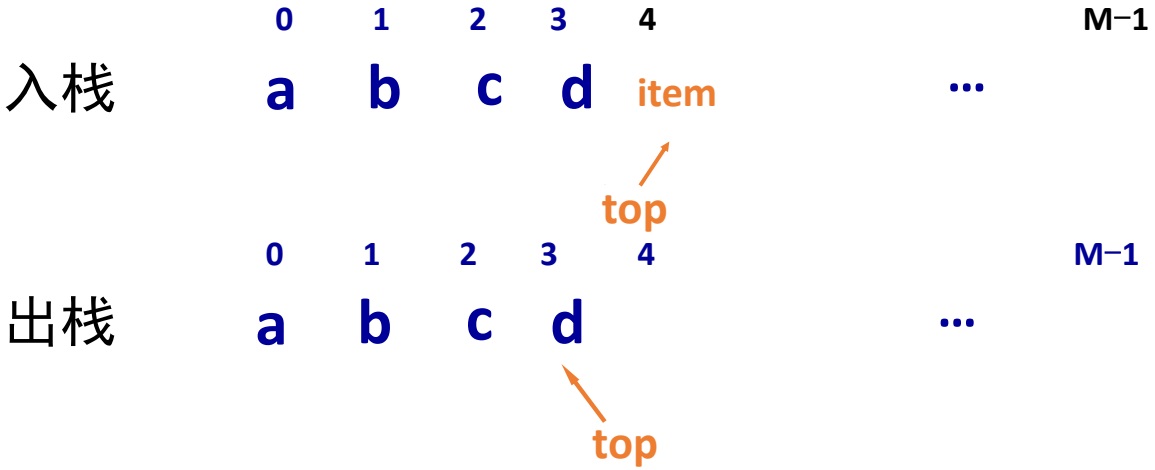


一个稍感困惑的问题

栈的顺序实现方式，可以借助数组来实现

STACK[0..M−1]





如何避免人为恶意地访问top所指以外的元素？

面向对象，C++，类

```
class stack {
public:
   stack();
   DataType pop();                    // 出栈
   void push(const DataType &item);    //入栈
   DataType topData( ) const;          // 读栈顶元素
   bool empty() const;                //  检查栈空
private:
   int top;                                    // 栈顶元素位
   int data[1000];                    //  栈的顺序存储
};
```

外部静态变量，其作用域为所在文件。

在C语言中能实现数据隐藏吗？

```
//main.c
…
int  main()
{…}
getSym()   operate()  compute()
{…}           {…}        {…}
…
```

```
//stack.c
static DataType Num_stack[MAXSIZE
static enum oper Op_stack[MAXSIZE];
static int Nop=-1;
static int  Otop=-1;
void pushNum(DataType num)
{…}
DataType popNum()
{…}
void pushOp(enum oper op)
{…}
enum operator  popOp()
{…}
enum operator  topOp()
{…}
```

静态变量（static）*

① 内部静态变量

- 在局部变量前加上‘static’关键字就成为内部静态变量。
- 内部静态变量仍是局部变量，其作用域仍在定义它的函数内。但该变量采用静态存贮分配（由编译程序在编译时分配，而一般的自动变量和函数形参均采用动态存贮分配，即在运行时分配空间），当函数执行完，返回调用点时，该变量并不撤消，其值将继续保留，若下次再进入该函数时，其值仍存在。

②外部静态变量

- 在函数外部定义的变量前加上“static”关键字便成了外部静态变量。
- 外部静态变量的作用域为定义它的文件，即成为该文件的的“私有”（private）变量，其它文件上的函数一律不得直接进行访问，除非通过它所在文件上的各种函数来对它进行操作，这可实现数据隐藏。(在C++中提供进一步的数据隐藏。)

静态变量（static）（续）*

例：下列程序打印出什么结果。

```
#include <stdio.h>
int f(int i);
main()
 {
        int i;
    for(i=0; i < 5; i++)
      printf("%d ",f(i));
}
int f(int i)
 {
        static int k = 1;

    k += i;
    return (k);
}
```

结果：

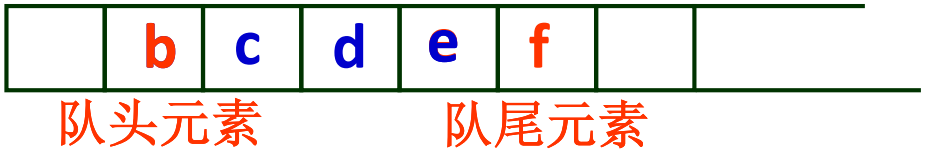
1 2 4 7 11

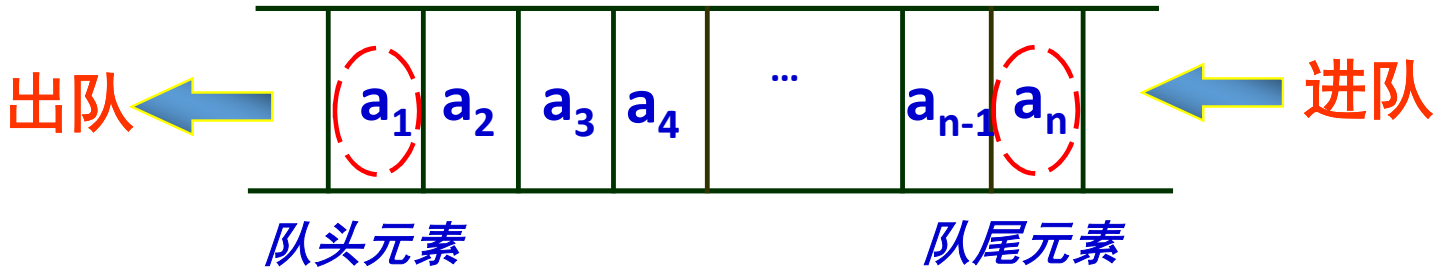
3.4 队(Queue)的基本概念

(一) 队的定义

先进先出

队列 简称**队** 。是一种只允许在表的一端进行插入操作，而在表的另一端进行删除操作的线性表。允许插入的一端称为**队尾**，队尾元素的位置由**rear**指出； 允许删除的一端称为**队头**，队头元素的位置由**front**指出。





FIFO(First-In-First-Out)

队的示意图

(二) 队的基本操作

1. 队的插入 (进队、入队) √
2. 队的删除 (出队、退队) √
3. 测试队是否为空 √
4. 测试队是否为满
5. 检索当前队头元素
6. 创建一个空队

特殊性

1. 其操作仅是一般线性表的操作的一个子集。
2. 插入和删除操作的位置受到限制。

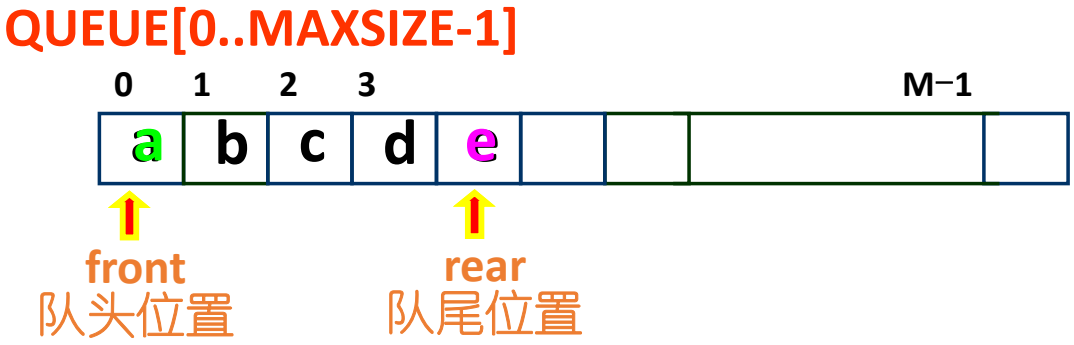
队列的基本操作

- void enQueue(Queue q, ElemType); //元素进队，即在队尾插入一 个元素
- ElemType deQueue(Queue q); //元素出队，即队头删除一个元素
- isFull(Queue q); //测试队列是否已满
- isEmpty(Queue q); //测试队列是否为空

3.5 队的顺序存储结构

(一) 构造原理

在实际程序设计过程中，通常借助一个一维数组**QUEUE[0..M–1]**来描述队的顺序存储结构，同时，设置两个变量 **front**与**rear**分别指出当前队头元素与队尾元素的位置。



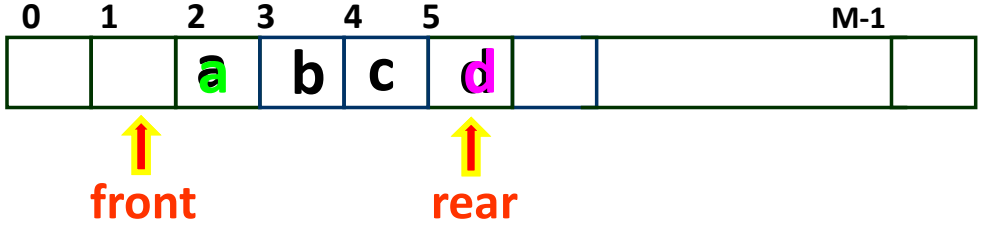
约定

rear 指出实际队尾元素所在的位置，

front 指出实际队头元素所在位置，

count 指出实际队中元素个数。

QUEUE[0..MAXSIZE–1]



初始时，队为空，有

front= 0　rear= –1　count=0

测试队为空的条件是

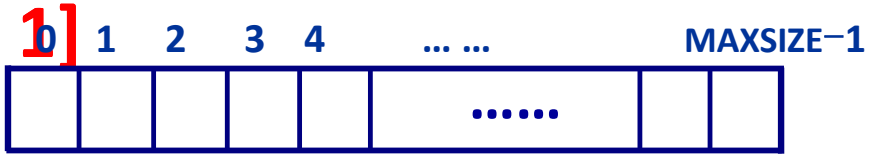
count==0

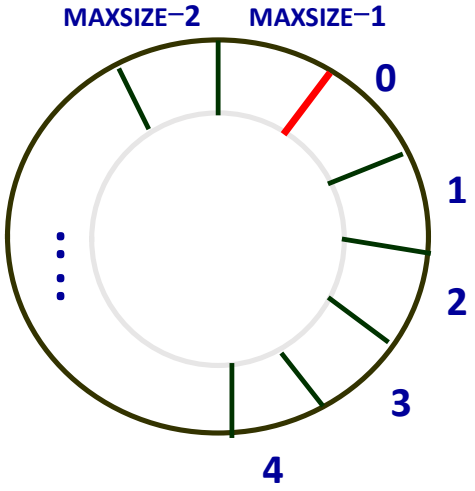
(二) 循环队列

在实际应用中，由于队元素需要频繁的进出，单向结构很容易造成溢出，即rear到达数组尾，而实际队中元素并没有超出数组大小。因此，在实际应用中通常将队设计成一个循环队列，从而提高空间利用率。

把队列(数组)设想成头尾相连的循环表，使得数组前部由于删除操作而导致的无用空间尽可能得到重复利用，这样的队列称为 **循环队列**。

QUEUE[0..MAXSIZE−1]





类型定义

```
#define MAXSIZE    1000
QElemType
QUEUE[MAXSIZE];
int  Front, Rear,Count;
```

由于变量Front和Rear需要在多个操作（函数）间共享，为了方便操作，在此将其设为全局变量。Count为队列中元素个数。

初始时，三个变量为：
Front = 0;
Rear = MAXSIZE – 1;
Count = 0;

（三）（循环队列）基本算法

1. 初始化队列

```
void  initQueue( )
{
      Front = 0;
      Rear = MAXSIZE-1;
      Count = 0;
}
```

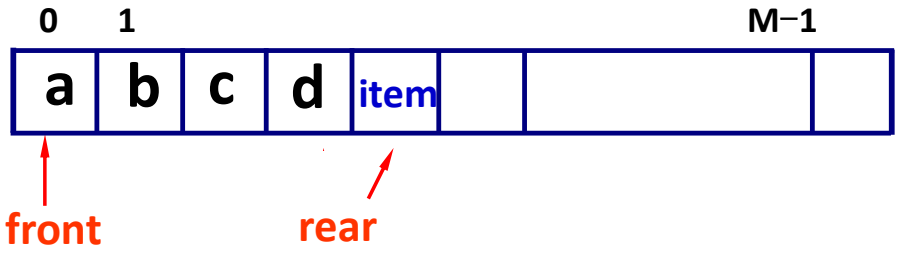
2、测试队列是否为空或满

队空或满，返回1，否则，返回0。

```
int  isEmpty( )
{
     return Count ==
0;
}
```

```
int  isFull( )
{
     return Count ==
MAXSIZE;
}
```

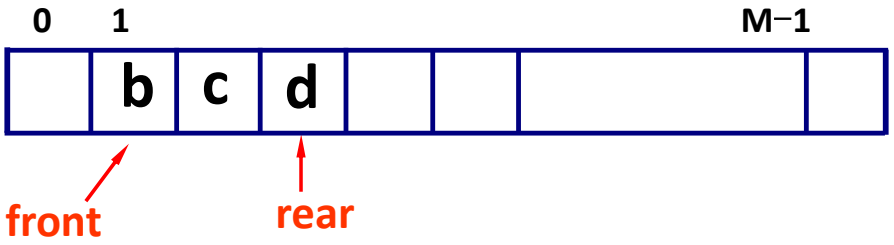
3. 插入(进队)算法



算法

```
void enQueue(ElemType queue[ ], ElemType item)
{
    if(isFull())                    /* 队满，插入失败 */
        Error("Full queue!");
    else{
        Rear = (Rear+1) % MAXSIZE;
        queue[Rear]=item;
        Count++;
                                    /* 队未满，插入成功 */
    }
}
```

4. 删除(出队)算法



算法

```
ElemType  deQueue(ElemType queue[ ])
{
   ElemType e;
     if(isEmpty())
          Error("Empty queue!");    /* 队空，删除失败 */
     else{
          e=queue[Front];
          Count--;                  /* 队非空，删除成功 */
          Front = (Front+1)%MAXSIZE;
          return e;
     }
}
```

3.6 队列的链式存储结构

链接队列
链队

(一) 构造原理

队列的链式存储结构是用一个线性链表表示一个队列，指针**front**与**rear**分别指向实际队头元素与实际队尾元素所在的链结点。

约定

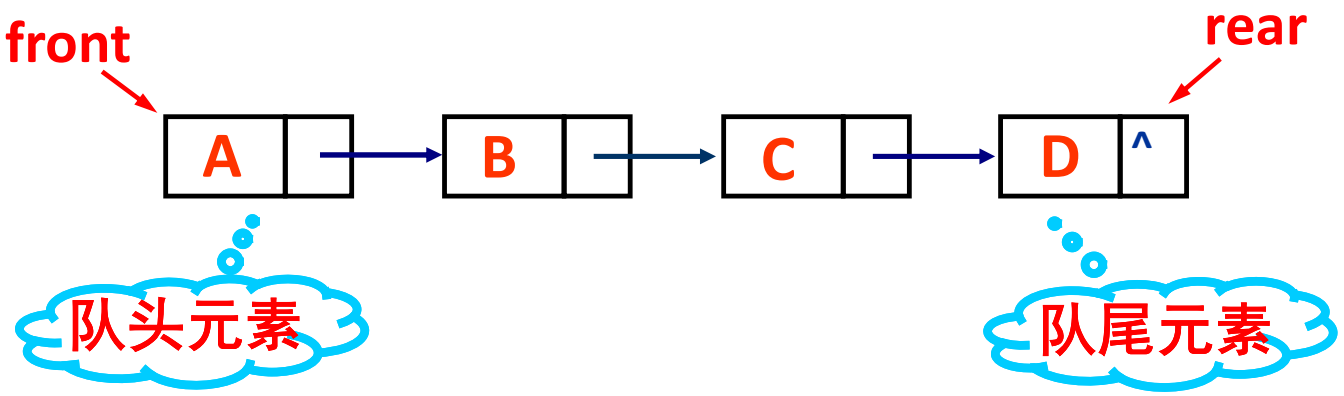
~~rear指出实际队尾元素所在的位置；~~
~~front指出实际队头元素所在位置。~~

front与rear分别指向实际队头和队尾元素

例 在一个初始为空的链接队列中依次插入数据元素

A, B, C, D

以后，队列的状态为



空队对应的链表为空链表，空队的标志是

front = NULL

类型定义

```
struct  node {
        ElmeType   data;
        struct  node   *link;
}
 typedef struct node   QNode;

typedef struct node *QNodeptr;
```

队头及队尾指针**front**和**rear**定义如下：

QNodeptr Front, Rear;

为了操作方便，通常将它们定义为全局变量

(二) 基本算法

1. 初始化队列

```
void  initQueue()
{
      Front=NULL;
      Rear=NULL;
}
```

2. 测试队列是否为空

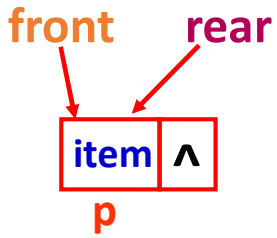
```
int  isEmpty()
{
      return Front==NULL;
}
```

队空,返回1
否则,返回0

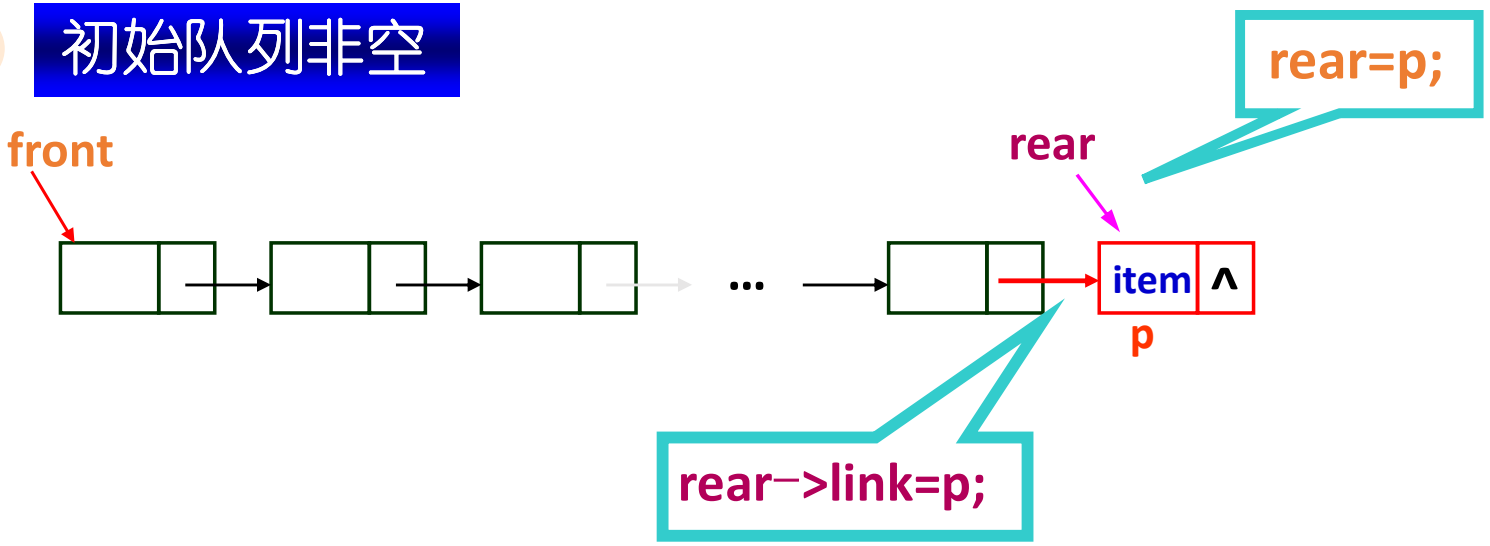
3. 插入(进队)

分两种情况

1 初始队列为空　front=rear=NULL



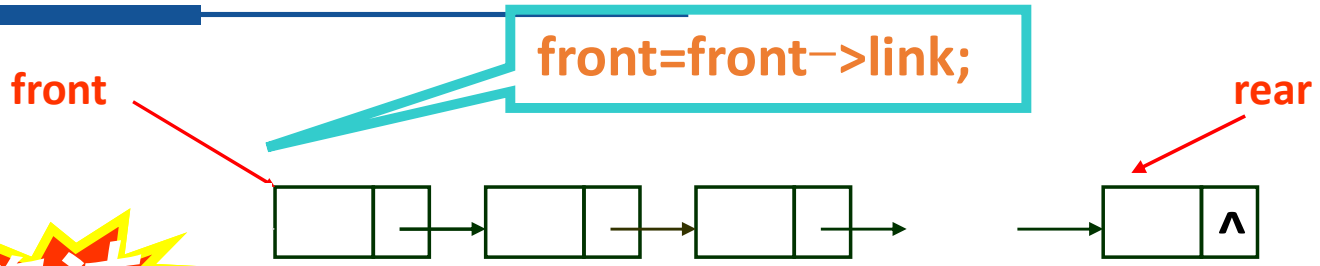
2 初始队列非空



算法

```
void  enLQueue(ElemType  item )
{    QNodeptr  p;
     if((p=(QNodeptr)malloc(sizeof(QNode))) ==NULL)  /* 申请链结点 */
          Error("No memory!");
     p->data=item;
     p->link=NULL;
     if(Front==NULL)
          Front=p;                    /* 插入空队的情况 */
     else
          Rear->link=p;
     Rear=p;                          /* 插入非空队的情况 */
}
```

4. 删除(出队)

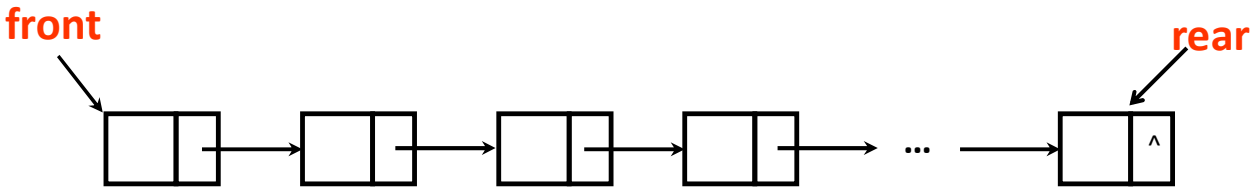


算法

```
ElemType  deLQueue( )
{    QNodeptr  p;
     ElemType item;
     if(isEmpty() )
          Error("Empty queue!");  /* 队为空，删除失败 */
     else{
          p=Front;
          Front=Front->link;
          item=p->data;
          free(p);
          return  item;          /* 队非空，删除成功 */
     }
}
```

5. 销毁一个队

所谓销毁一个队是指将队列所对应的链表中所有结点都删除，并且释放其存储空间，使队成为一个空队(空链表)。



归结为一个线性链表的删除

算法

```
void destroyLQueue()
{
    while(Front != NULL){                    /* 队非空时 */
        Rear=Front->link;
        free(Front);                  /* 释放一个结点空间 */
        Front=Rear;
    }
}
```

问题3.2：银行排队模拟(Simula

一个系统模仿另一个系统行为的技术称为**模拟**，如飞行模拟器。模拟可以用来进行方案认证、人员培训和改进服务。计算机技术常用于模拟系统中。

生产者-消费者（Server-Custom）是常见的应用模式，见于银行、食堂、打印机、医院、超市…提供服务和使用服务的应用中。这类应用的主要问题是消费者如果等待（排队）时间过长，会引发用户抱怨，影响服务质量；如果提供服务者（服务窗口）过多，将提高运管商成本。（排队论-queuing theory）

某银行网点有五个服务窗口，分别为三个对私、一个对公和一个外币窗口。通常对私业务人很多，其它窗口人则较少，可临时改为对私服务。假设当对私窗口客户平均排队人数超过7人时，客户将有抱怨，此时银行可临时将其它窗口中一个或两个改为对私服务，当客户少于7人时，将恢复原有业务。设计一个程序用来模拟银行服务。

输入：首先输入一个整数表示时间周期数，然后再依次输入每个时间周期中因私业务的到达 客户数。注：一个时间周期指的是银行处理一笔业务的平均处理时间，可以是一分钟、三分钟或其它。例如：

6

2 5 13 11 15 9

说明：表明在6个时间周期内，第1个周期来了2个（ID分别为1,2），第2个来了5人（ID分别为3,4,5,6,7），以此类推。

输出：每个客户等待服务的时间周期数。

问题3.2：问题分析及算法设计

在**生产者-消费者**应用中消费者显然是先来先得到服务。在此，显然可用一个**队列**来存放等待服务的**客户**队列。每个客户有二个基本属性：排队序号和等待时间（时间周期数）：

```
struct  cust {
   int id;   //客户排队序号
   int wtime;  //客户等待服务的时间（时间周期数）
};
Struct cust Cqueue[MAXSIZE]; //等待服务的客户队列，一个循环队列
```

为了简化问题，可用一个变量来表示银行当前提供服务的**窗口数**：

```
int  snum;
```

在本问题中，该变量的取值范围为3<= snum <= 5

生产者-消费者模式中的基本策略：**追求各自利益最大化！**

对客户来说：要求尽快得到服务，也就是说，当他到达银行要求得到服务时，排队过长会有抱怨。（要求多开窗口）（当客户到达时，如排队过长，银行应想办法减少排队，避免抱怨产生）

对银行来说：在满足客户基本要求的前提下，尽可能减少成本支出（少开窗口）（当队列中客户减少时，将及时减少窗口）

问题3.2：问题分析及算法设计

主要算法：

```
for(clock=1; ; clock++) //在每个时间周期内
{
    1. If 客户等待队列非空
            将每个客户的等待时间增加一个时间单元;
    2. If(clock <= simulationtime)
            2.1 如果有新客户到来（从输入中读入本周期内新来客户
            数），将其入队;
            2.2 根据等待服务客户数重新计算服务窗口数;
    3. If 客户等待队列非空
            3.1 从客户队列中取（出队）相应数目（按实际服务窗口数）
            客户获得服务;
            3.2 然后根据等待服务客户数重新计算服务窗口数;
       Else 结束模拟
}
```

问题3.2：代码实现

```
//main.c
#include <stdio.h>
#include  <stdlib.h>
#define  MAXSIZE 200  //队列容量
#define THRESHOLD  7  //窗口增加阈值
#define MAXSVR  5   //最大服务窗口数
#define MINSVR  3   //最小服务窗口数
typedef struct {
   int id;
   int wtime;
} CustType;
int Winnum=MINSVR; //提供服务的窗口数
void  updateCustqueue(); //更新等待队列中客户等待时间
void  enCustqueue(CustType c); //客户入等待队列
CustType deCustqueue(); //客户出队
int  getCustnum();  //获取队中等待客户人数
int  isFull();
int  isEmpty();
void arriveCust(); //获取新客户，并加至等待队列中
int  service(); //银行从队列中获取客户进行服务
```

```
//main.c
int main()
{
   int clock, simulationtime;
   scanf("%d",&simulationtime);
   for(clock=1; ; clock++) {
      updateCustqueue();
      if(clock <= simulationtime )
         arriveCust();
      if(service()==0 && clock > simulationtime)
         break;  //等待队列为空且不会有新客户
   }
   return 0;
}
```


77

问题3.2：代码实现

内部静态变量，作用域为当前函数，生存周期同全局变量。它只初始化一次，每次函数调用时，上次的值仍在。

```
//main.c
void arriveCust()
{
   int i,n;
   static int count=1;
   CustType c;
   scanf("%d", &n);
   for(i=0; i<n; i++){
      c.id = count++; c.wtime = 0;
      enCustqueue(c);
   }
   while((getCustnum() / Winnum) >= THRESHOLD &&
       Winnum<MAXSVR)  //增加服务窗口
      Winnum++;
}
int service()
{
   int i;
   CustType c;
   for(i=0; i<Winnum; i++)
      if(isEmpty() ) return 0;
     else {
        c = deCustqueue();
        printf("%d :%d\n", c.id, c.wtime);
     }
    if((getCustnum() / Winnum) < THRESHOLD && Winnum>MINSVR)
       Winnum--;
   return 1;
}
```

问题3.2：思考

在本问题中，当前服务窗口平均排队等待服务的客户人员数小于某个阈值时，临时窗口将不再提供服务，一来该策略不是最优，二来也不符合实际情况。现增加如下规则：

- 外币和对公窗口应优先处理本业务，即当有对应业务（有客户等待时）时应优先处理，只有当本业务没有排队客户时，才能处理对私业务；
- 只有当外币和对公窗口没有等待客户同时对私窗口有等待客户排队时，将处理对私业务（资源利用最大化）。
- 现在个人业务处理时间都一样的（一个时间周期），这不符合实际情况。可将个人业务分为复杂、较复杂和简单。

优先队列（Priority queue）*

在实际应用时，前述简单队列结构是不够的，先入先出机制需要使用某些优先规则来完善。如：

- 在服务行业，通常有残疾人、老人优先
- 在公路上某些特殊车辆（如救护车、消防车）优先
- 在操作系统进程调度中，具有高优先级的进程优先执行

优先队列（Priority Queue）：根据元素的优先级及在队列中的当前位置决定出队的顺序。

优先队列的实现*

- 方法一：使用两种变种链表实现。一种链表是所有元素都按进入顺序排列（队），取元素效率为O(n)；另一种链表是根据元素的优先级决定新增位置（按优先级排序），新增元素效率为O(n)。实现简单。

- 方法二：使用一个链表和一个指针数组，链表用于存放元素，一个指向链表的指针数组用于确定新加入的元素应该在哪个范围中（按优先级），算法的时间复杂度为**O($\sqrt{n}$)**。（J.O.Hendriksen提出）。

- 方法三：**用一个堆（Heap）结构实现。这是常用的一种高效实现优先队列的方法**（原理将在树中讲解），算法的时间复杂度为O($\log_2$ N）。

本章内容小结

栈、队

- 栈、队的基本概念
  - 栈、队的定义
  - 栈、队的基本操作

    栈、队列是特殊线性表(特殊性)
- 栈、队的顺序存储结构
  - 构造原理、特点
  - 对应的插入、删除操作的算法设计

    (循环队列)
- 栈、队的链式存储结构
  - 构造原理、特点
  - 对应的插入、删除操作的算法设计
- 栈、队的应用举例

思考1

设有一顺序栈S，n个元素 依次进栈，问这n个元素的任意排列都可以对应某一个可能的出栈顺序吗?

反例：a,b,c依次入栈，c,a,b是否是一个可能的出栈顺序？

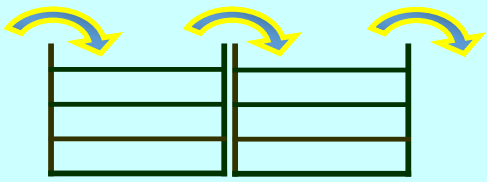
不同的出栈序列个数总共有多少？（与n相关）

卡特兰数

2个堆栈串联的情况呢？

（元素不能反向运动）

多个栈呢？



思考2

练习4扩展：设有一顺序栈S，n个不同的元素$a_1, a_2, a_3, \cdots, a_n$依次进栈，给出一个算法，判断上述元素的一个排列是否是合法的出栈序列，如果是，给出其出栈过程中所需的栈容量最小值。

例如：

- 输入入栈序列abc，出栈序列cba，输出：3
- 输入入栈序列abc，出栈序列cab，输出：不合法
- 输入入栈序列a, b, c, d, e, f，出栈序列 b, d, c, f, e, a，输出：3

结束！