第三章 栈和队列

王　　勇

计算机/软件学院　　大数据分析与信息安全团队

21#518　　电　话 13604889411

Email：　wangyongcs@hrbeu.edu.cn

哈尔滨工程大学
Harbin Engineering University

问题提出

- 多窗口排队问题
- 多进程/线程CPU/分配问题
- 铁路火车调度问题
- 回文问题
- 子程序调用现场保护问题

本章主要内容说明

知识点

栈的抽象数据类型
栈的顺序表示
栈的链式表示
栈的应用与实现
队列抽象数据类型
队列的顺序表示
队列的链式表示
循环队列

重 点

栈和队列
结构特点
正确使用

掌握

熟练掌握

栈和队列这两种抽象数据类型的

循环队列和链队列的基本操作实现算法

熟练掌握

理解

递归算法执行过程中栈的状态变化过程

标

栈和队列

结构特点 基本操作的特殊性

- 栈：“后进先出”
- 队列：“先进先出”

不 同 与线性表相比，它们的插入和删除操作受更多的约束和限定，故又称为限定性的线性表结构

- 线性表允许在表内任一位置进行插入和删除
- 栈只允许在表尾一端进行插入和删除
- 队列只允许在表尾一端进行插入，在表头一端进行删除

本章内容

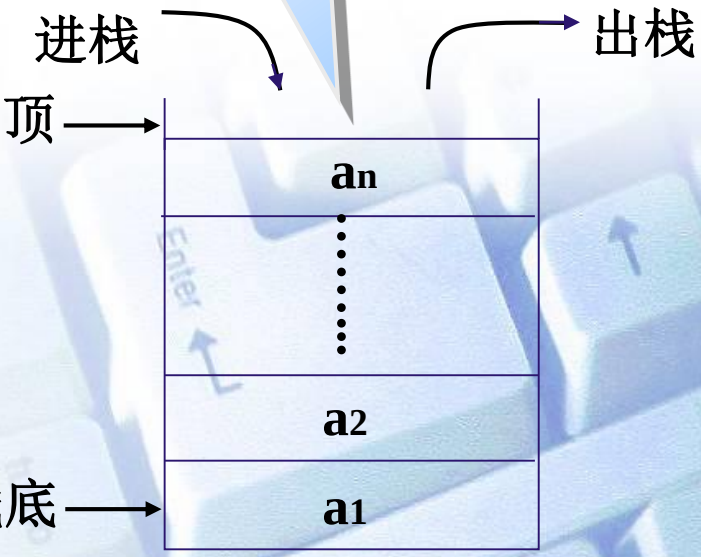
栈

栈

限定仅在表尾进行插入或删除操作的线性表

- 栈顶（top）—表尾
- 栈底（bottom）—表头
- 空栈—不含元素的空表

栈s=(a1,a2,……,an)



特 点

- 先进后出（FILO）
- 后进先出（LIFO）

基本操作

初始化、判空、插入、删除、取栈顶元素

栈

- 栈的抽象数据类型定义
- 栈的表示和实现

栈的抽象数据类型定义

定义

ADT Stack

{

数据对象：$D=\{ a_i | a_i \in ElemSet, i=1, \ldots, n, n>=0\}$

数据关系：$R1=\{<a_{i-1}, a_i> | a_{i-1}, ai \in D, i=2, \ldots, n\}$

基本操作：

InitStack(&S) //建空栈

DestroyStack(&S) //撤消栈

ClearStack(&S) //清空栈

栈的抽象数据类型定义

StackEmpty(&S)　　//判空栈

StackLength(S)　　//返回栈元素个数

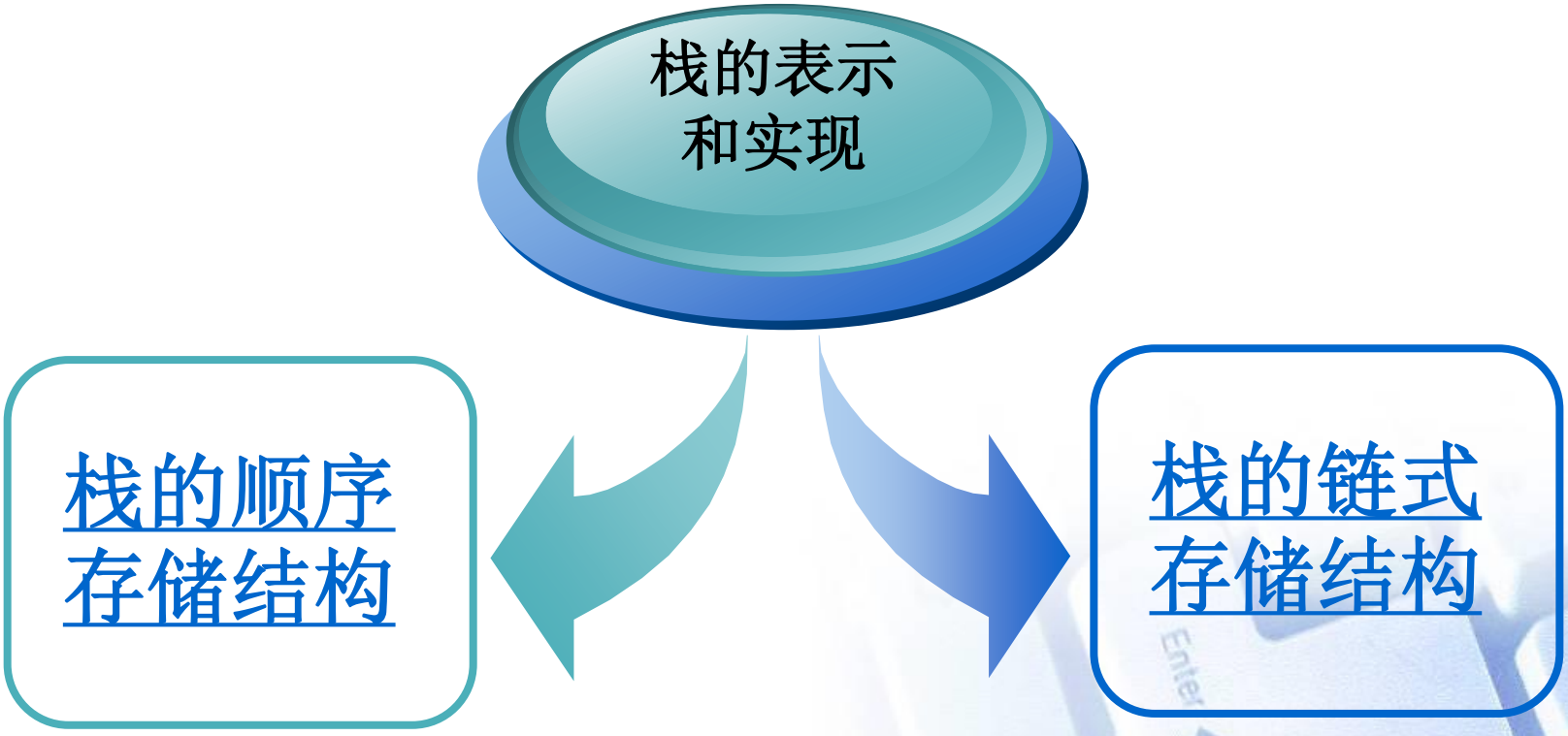
GetTop(S, &e)　　//用e返回栈顶元素

Push(&S, e)　　//在栈顶插入元素e

Pop(&S, &e)　　//在栈顶删除元素，e返回

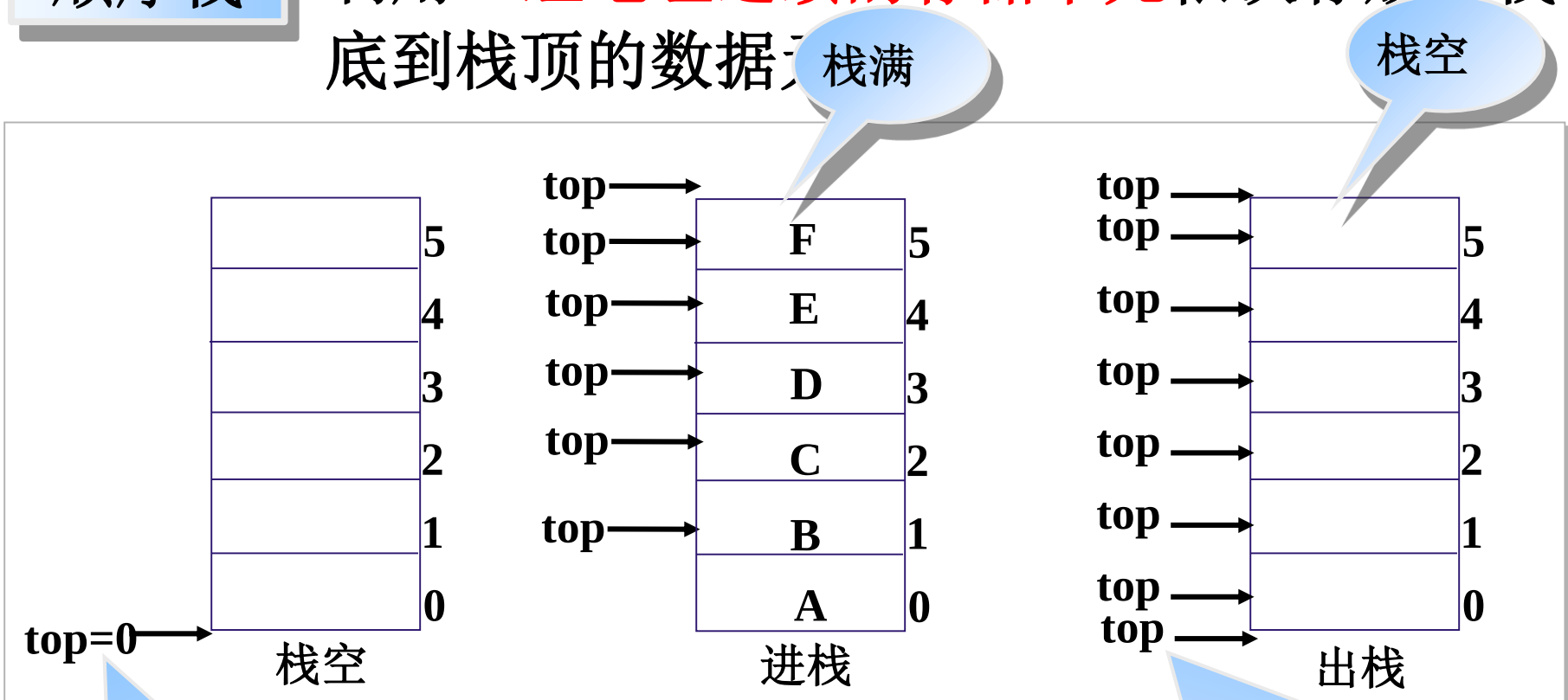
}ADT Stack

栈的表示和实现



栈的表示和实现　　栈的顺序存储结构

顺序栈　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



栈顶指针top，指向实际栈顶后的空位置，栈底base

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

顺序栈表示

```
                                        //----栈的顺序存储表示
#define  STACK_INIT_SIZE  100;   //存储空间初始分配量
#define  STACKINCREMENT  10;   //存储空间分配增量
typedef  struct  {
     SElemType  *base;       //构造之前和销毁之后base为null
     SElemType  *top;
              int  stacksize;  //当前已分配存储空间，单位元素
}SqStack;
//----基本操作的函数原型说明-----
Status InitStack(SqStack &S);          //构造一个空栈S
Status DestroyStack(SqStack &S);     //销毁栈S，S不再存在
Status ClearStack(SqStack &S);         //把S置为空栈
Status StackEmpty(SqStack S);
                //若S为空栈，则返回TRUE，否则返回FALSE
```

int StackLength(SqStack S);

//返回S的元素个数，即栈的长度

Status GetTop(SqStack S, SElemType &e);

//若栈不空，则用 e 返回S的栈顶元素，并返回TRUE；否则返回 FALSE。

Status Push (SqStack &S, SElemType e);

//若栈的存储空间不满，则插入元素 e ，并返回 TRUE；否则返回FALSE。

Status Pop (SqStack &S, SElemType &e);

//若栈不空，则删除S的栈顶元素，用e返回其值，并返回TRUE；否则返回FALSE。

栈的顺序存储结构操作

初始化

若初始化成功，返回OK；否则返回OVERFLOW

Click

入栈

插入元素e为新的栈顶元素

Click

出栈

若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK；否则返回ERROR

Click

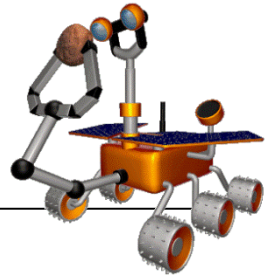
取元素

若栈不空，则用e返回S的栈顶元素，并返回OK；否则返回ERROR

Click

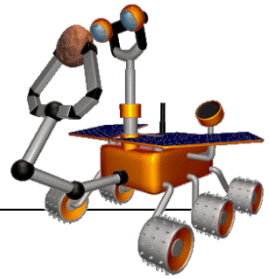
顺序栈初始化算法

```
Status InitStack (SqStack &S){
    // 构造一个空栈 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
```



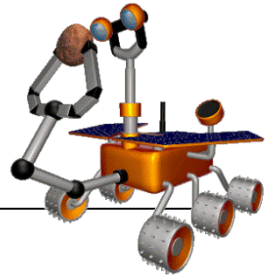
顺序栈入栈算法

```
Status  Push(SqStack  &S, SElemType  e)
{                    //插入元素e为新的栈顶元素
   if(S.top-S.base>=S.stacksize)      //栈满，追加存储空间
   {  S.base=(SElemType*)realloc(S.base,
        (S.stacksize+STACKINCREMENT)*sizeof(SElemType));
      if (!S.base) exit (OVERFLOW);
      S.top=S.base+S.stacksize;
      S.stacksize+=STACKINCREMENT;
   }
   *S.top++=e;
   return  OK;
}//Push
```



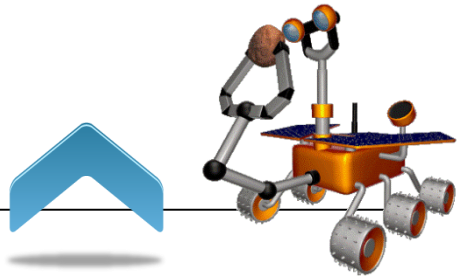
顺序栈出栈算法

```
Status  Pop(SqStack  &S, SElemType &e)
{     //若栈不空，则删除S的栈顶元素，用e返回其值，
      //并返回OK；否则返回ERROR
   if (S.top==S.base)  return  ERROR;
   e=*--S.top;
   return  OK;
}//Pop
```

顺序栈取栈顶算法

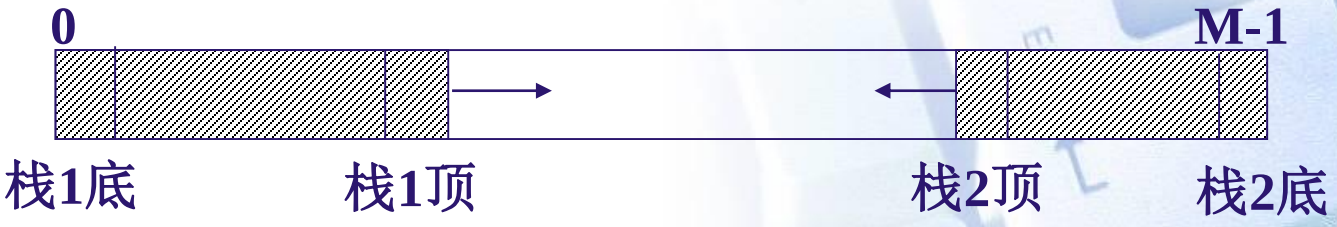
```
Status GetTop(SqStack  S,  SElemType  &e)
{   //若栈不空，则用e返回S的栈顶元素，并返回OK；
    //否则返回ERROR
    if (S.top==S.base)  return  ERROR;
    e=*(S.top-1);       //注意：top-1指针指向栈顶元素，
                                top未移动
    return  OK;
}//GetTop
```

栈溢出 栈满，再入栈，则“上溢”

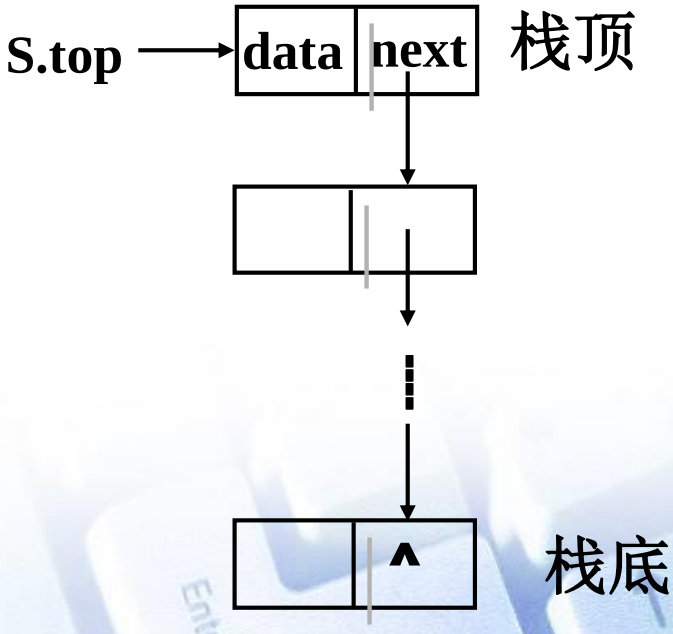
解 决

- 事先分配一个足够大的空间，但事先无法估计
- 双栈，共享一个顺序存储结构。即在一个程序中同时使用两个栈



栈的链式存储结构

链 栈 利用链表依次存放自栈顶到栈底的数据元素



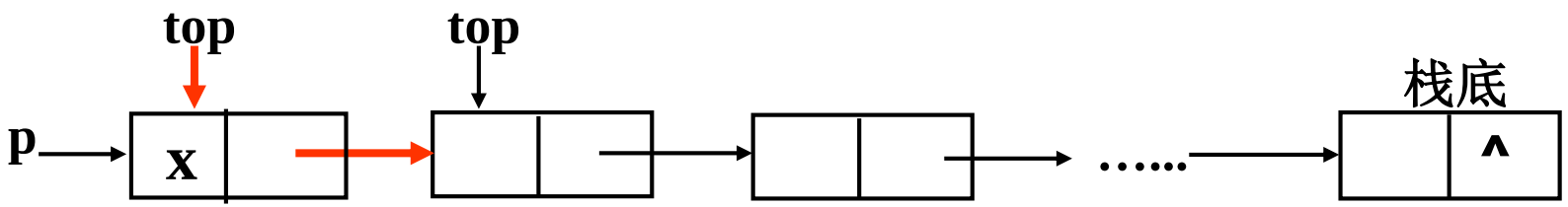
结点定义

```
typedef   struct LNode
   {   int data;
       struct  LNode  *next;
   }LNode;
```

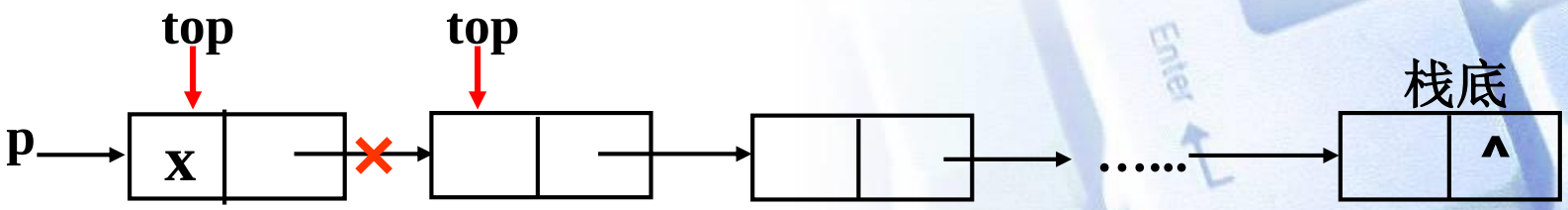
栈的链式存储结构操作

入栈 插入元素e为新的栈顶元素

Click



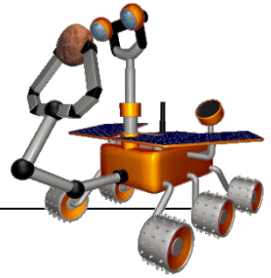
出栈 若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK；否则返回ERROR



Click

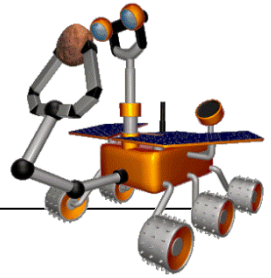
链式栈入栈算法

```
LinkList  *lzjz(Linklist  *top, int x)
{  LinkList *p;
   p=(LinkList*)malloc(sizeof(LNode));
   p->data=x;
   p->next=top;
   top=p;
   return(p);
}
```

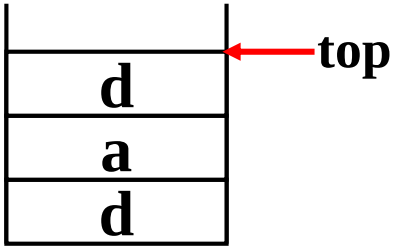
链式栈出栈算法

```
LinkList  *lztz(LinkList &top, int &p)
{   LinkList *q;
    if (top!=NULL)
    {   q=top;
        *p=top->data;
        top=top->next;
        free(q);
    }
    return(top);
}
```



应用一 回文游戏(顺读与逆读字符串一样(不含空格))

如，字符串：“madam im adam”



（1）读入字符串

（2）去掉空格（原串）

（3）压入栈

（4）原串字符（去掉空格）与出栈字符依次比较

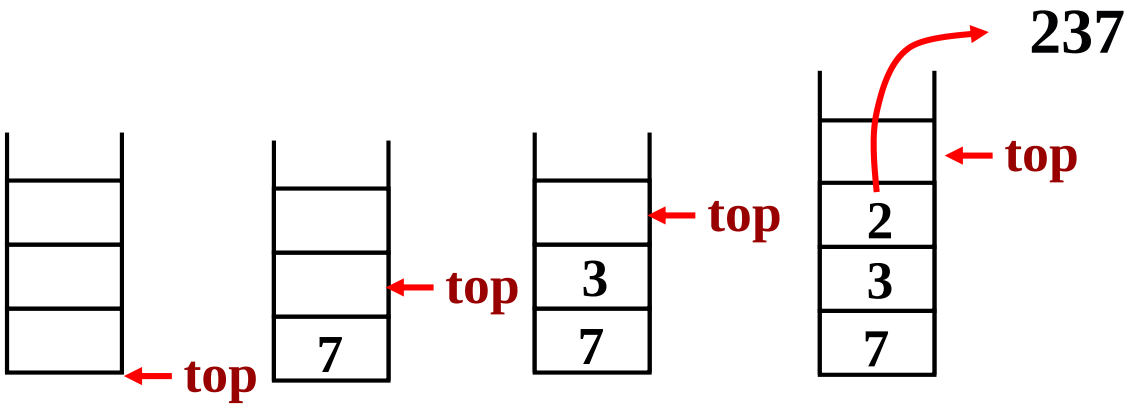
若不等，非回文

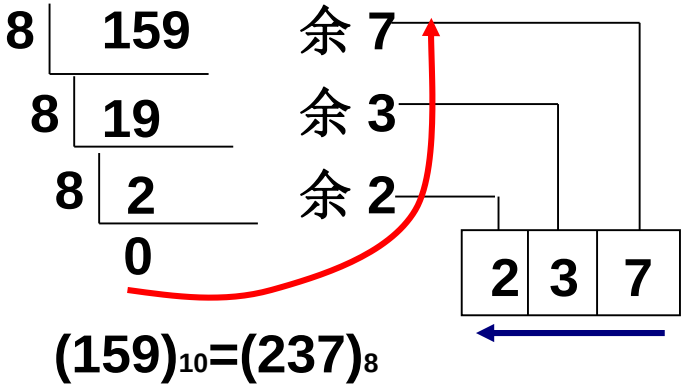
若直到栈空都相等，回文

应用二 数制转换

如，把十进制数159转换成八进制数

如，把十进制数159转换成八进制数





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

数制转换算法

```
Void conversion(){
    //输入任意一非负十进制整数，输出与其等值的八进制数
    InitStack(S);          //初始化空栈
    scanf("%d",N);         //输入非负十进制整数
    while (N)
        {  Push(S,N%8);     //求余，余数入栈
           N=N/8;           // 非零“商”继续运算
        }
    while (!StackEmpty(S))  // 和“求余”所得相逆的
                            顺序输出八进制的各位数
        {  Pop(S,e);
           printf("%d",e);
        }
}//conversion
```

应用三 行编辑程序

- 接受用户从终端输入的程序和数据，并存入数据区
- 允许用户输入出错
- 如发现刚输入的一个字符错时，可补进一个退格符“#”，表示前一个字符无效；
- 如发现当前输入的行内差错较多或难以补救，则可以键入一个退行符“@”，表示当前行中字符均无效

如：**whli##ilr#e(s#*s)**
outcha@putchar(*s=#++);
是：**while(*s)**
putchar(*s++);

思 想

设行输入缓冲区为一个栈结构，每接受一个字符后便进行判断：

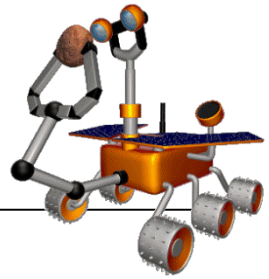
（1）如果它既不是退格符#也不是退行符@，则将其压入栈顶；

（2）如果是一个#，则从栈顶删除一个字符；

（3）如果是一个@，则将栈清为空栈

行编辑程序算法

```
void LineEdit( )   //接收一行入字符栈S并传送至调用过程数据区
 {  InitStack(S);   ch=getchar();
    while (ch!=EOF)
      {  while (ch!=EOF && ch!='\n')  //当前行及整个文本没结束
          {  switch (ch) {                    //判字符ch行内部
                case  '#'  :Pop(S, c);        break;
                case '@' :ClearStack(S); break;
                default :Push(S, ch);       break;
                              }
             ch=getchar();
           }
       ClearStack(S);//输出本行后清空本次操作栈以备下行使用
       if (ch!=EOF)  ch=getchar();  //读下一行的第一个字符（换行）
         }
     DestroyStack(S); //全文输入结束撤销栈
}//LineEdit
```



应用四 表达式求值

定 义

- 前缀表达式：+a b
- 中缀表达式：a+b
- 后缀表达式：a b+

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

算 符

运算符和界符

优先关系：<、=、> （见表3.1）

应用四 表达式求值

组 成

- 操作数(operand) :常数、变量
- 运算符(operator) :算术、关系和逻辑运算符
- 界限符(delimiter) :左右括弧和表达式结束符

规 则

- 先乘除后加减
- 先左后右
- 先括号内后括号外
- 出错
- 如： 4+2*3-10/5
  =4+6-10/5
  =10-10/5
  =10-2
  =8

算符优先关系表3.1

栈顶的

刚读入的

Q1 \ Q2	+	–	*	/	(	)	#
+	>	>	<	<	<	>	>
–	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

即同级运算符栈顶运算符优先级高！！

应用四 中缀表达式求值

思 想

设置两个工作栈

- 操作数栈OPND，置为空栈（放表达式的运算结果）
- 运算符栈OPTR，置栈底为表达式的起始符#；

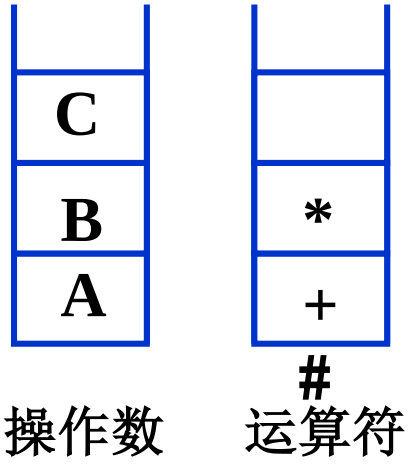
- 自左向右扫描表达式(即依次读每一个字符)
  - 若是操作数，则进栈OPND
  - 若是运算符，则与OPTR栈顶进行优先数比较(同级的栈项为大，刚读入的为小)：
    - 若读的运算符大于OPTR栈顶项，则进栈
    - 若栈顶项大，则栈顶运算符退栈，操作数栈顶两个元素退栈，并作一个运算，结果入栈OPND
    - 若相等且为括号，则脱括号
- 若运算符栈顶项为#，则操作数栈顶为计算结果，结束；否则出错

中缀表达式求值过程

例：扫描 A+B*C-D/(E+F)#

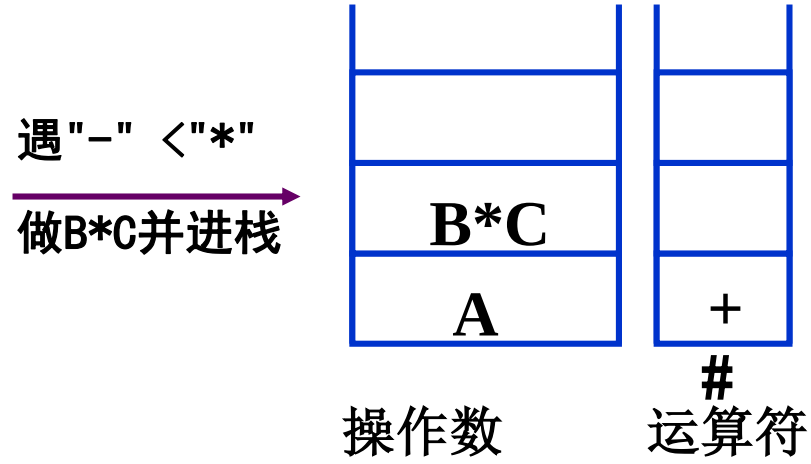
进栈

‘+’>’#’ *’ >’-’



遇"-" <"*"

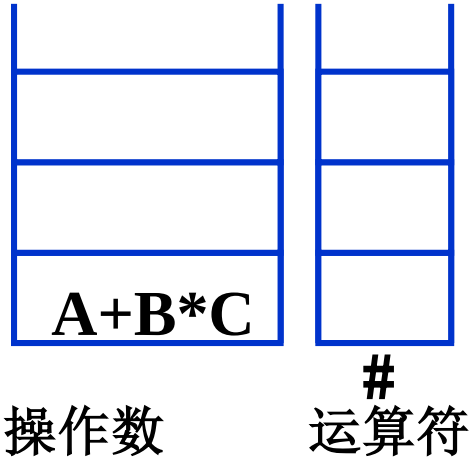
做B*C并进栈



"-"<"+"

做A+B*C

并进栈



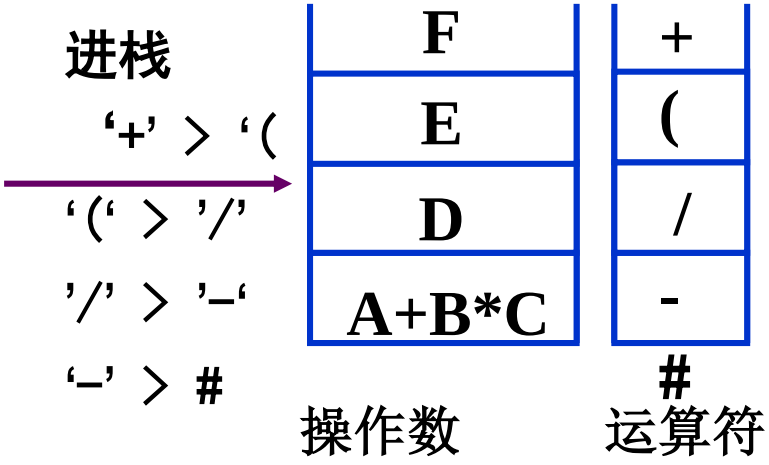
进栈

‘+’ > ‘(

‘(‘ > ’/’

’/’ > ’-‘

‘-’ > #



中缀表达式求值过程

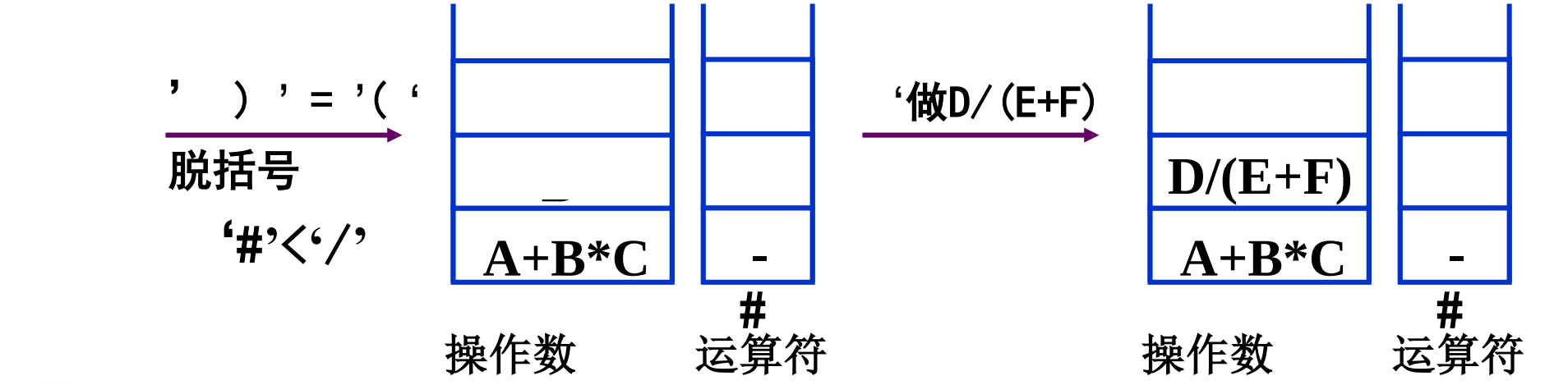
例：扫描 A+B*C-D/(E+F)#

遇’)’ < ’+’

’) ’ = ’(‘

脱括号

‘#’<‘/’

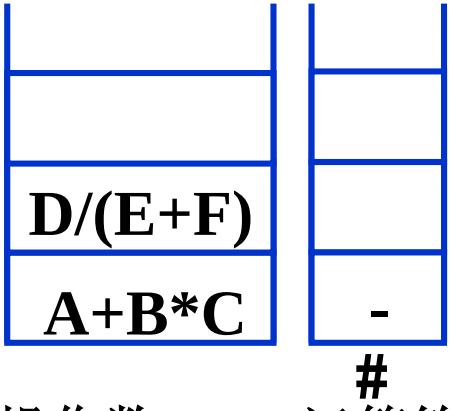


中缀表达式求值过程

例：扫描 A+B*C-D/(E+F)#

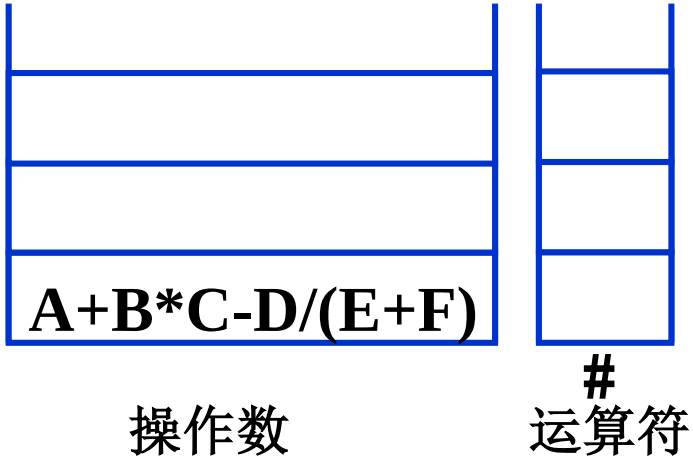
‘#’<’/’

做D/(E+F)



‘#’<’–’

做A+B*C–D/(E+F)



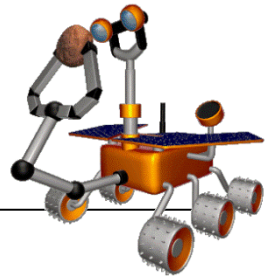
‘#’=‘#’

操作数出栈

即计算结果

中缀表达式求值算法

```
OperandType EvaluateExpression() {
  //OPTR和OPND为运算符栈和操作数栈，OP为运算符集合
  InitStack(OPTR);  Push(OPTR, '#');
  InitStack(OPND);  c=getchar();
  while (c!='#' ||GetTop(OPTR)!='#') {
    if (!In(c,OP)) {Push((OPND,c); c=getchar();}
    else
      switch (Precede(GetTop(OPTR), c)) {
        case '<': Push(OPTR, c);  c=getchar(); break;
        case '=': Pop(OPTR, x); c=getchar(); break;//脱括号
        case '>': Pop(OPTR, theta);
                  Pop(OPND, a); Pop(OPND, b);
                  Push(OPND, Operate(a, theta, b));
                  break;
      }//switch
    }//while
  return GetTop(OPND)
}//EvaluateExpression
```



应用四 后缀表达式求值

思 想

a. 读入表达式一个字符
b. 若是操作数，压入栈，转d
c. 若是运算符，从栈中弹出两个数，运算结果再压入栈
d. 若表达式输入完毕，栈顶即表达式值；否则，表达式未输入完，转a

后缀表达式：435*+

例 计算 4+3*5

仅用一个操作数栈即可！！

栈与递归的实现

函数调用

◆ 运行被调用函数之前

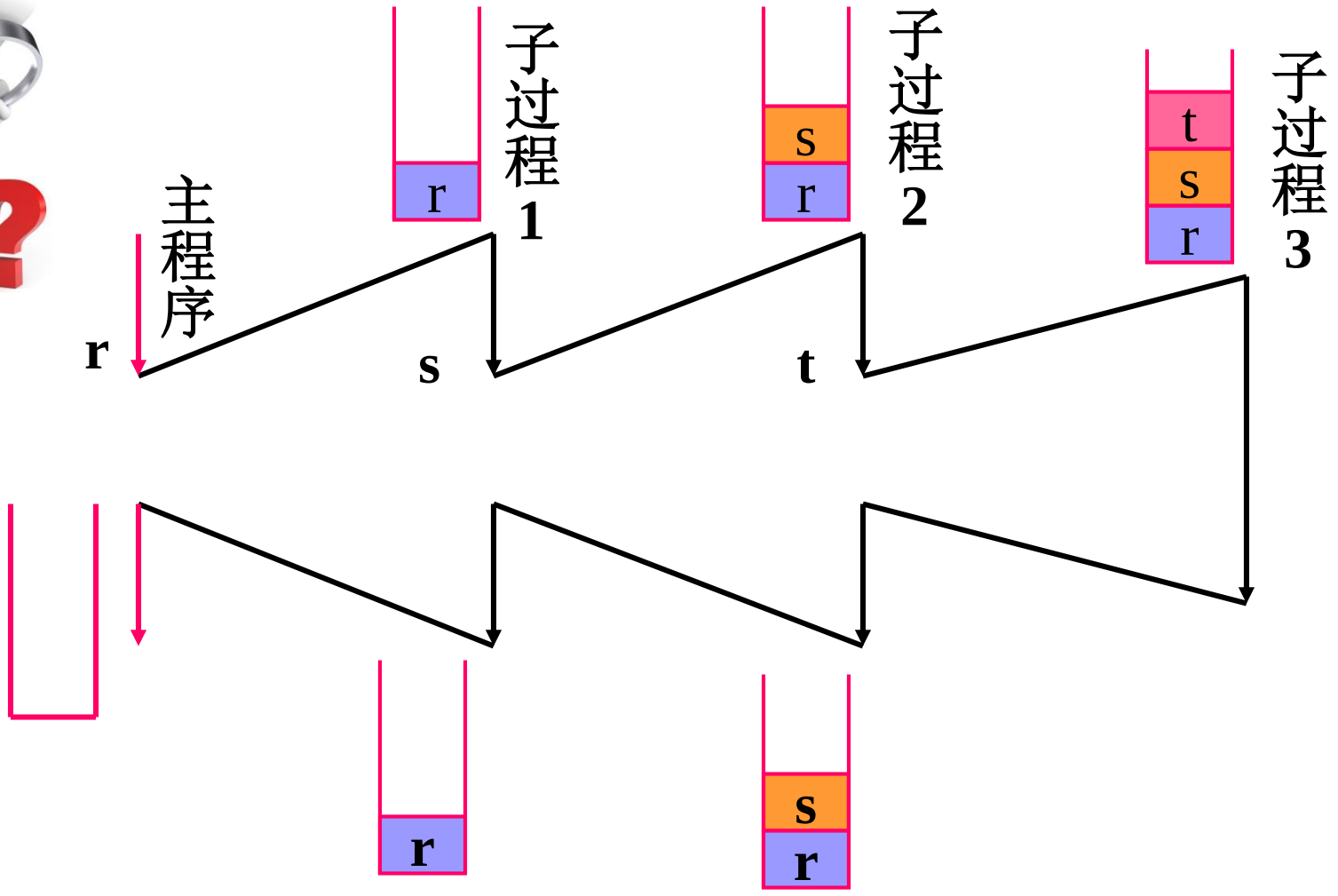
- 将所有的实际参数、返回地址等信息传递给被调用函数保存
- 为被调用函数的局部变量分配存储区
- 将控制转移到被调用函数的入口

◆ 运行被调用函数返回之前

- 保存被调函数的计算结果
- 释放被调函数的数据区
- 依照被调函数保存的返回地址将控制转移到调用函数

结 论 后调用先保存（处理）

1. 过程的嵌套调用



- 递归：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

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

Click

运行结果：

1，

2，2，

3，3，3，

3.递归调用执行情况

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

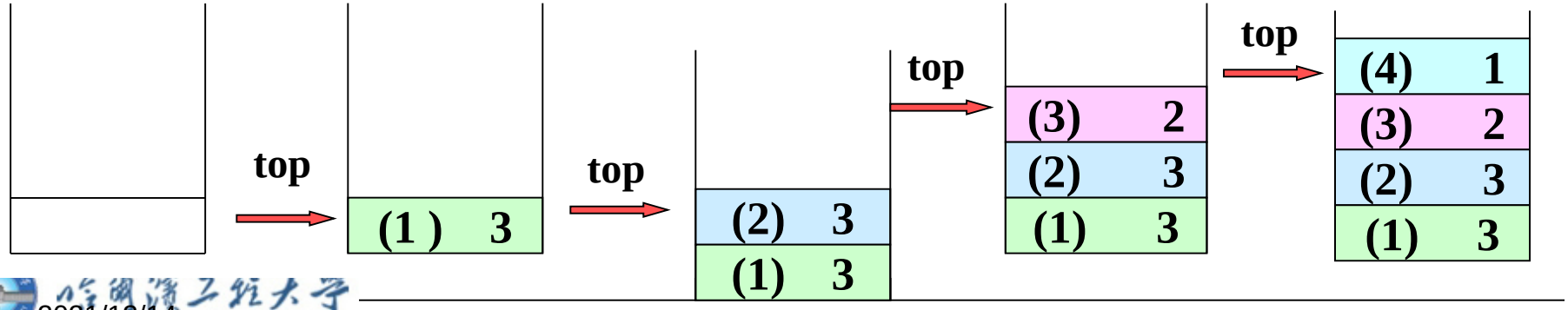
print(0);

(4)输出：1

w
0

返回

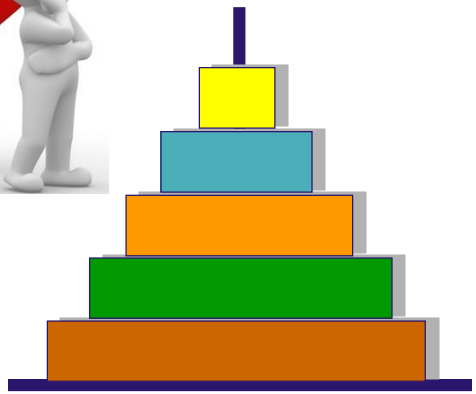
Back



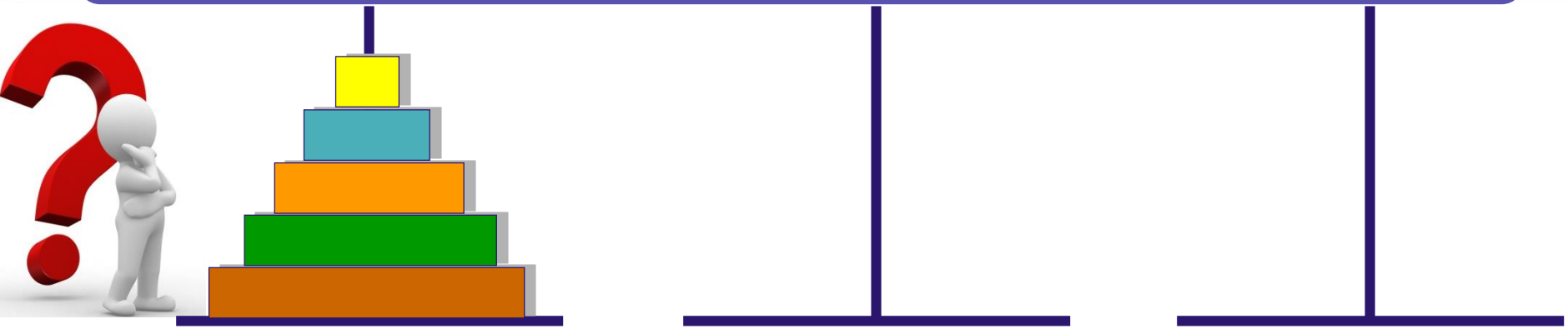
梵塔问题

梵 塔 有X,Y,Z三个塔座，X上套有n个直径不同的圆盘，按直径从小到大叠放，形如宝塔，编号1, 2, 3……n要求将n个圆盘从X移到Z，叠放顺序不变，移动过程中遵循下列原则：

- 每次只能移一个圆盘
- 圆盘可在三个塔座上任意移动
- 任何时刻，每个塔座上不能将大盘压到小盘上



X Y Z

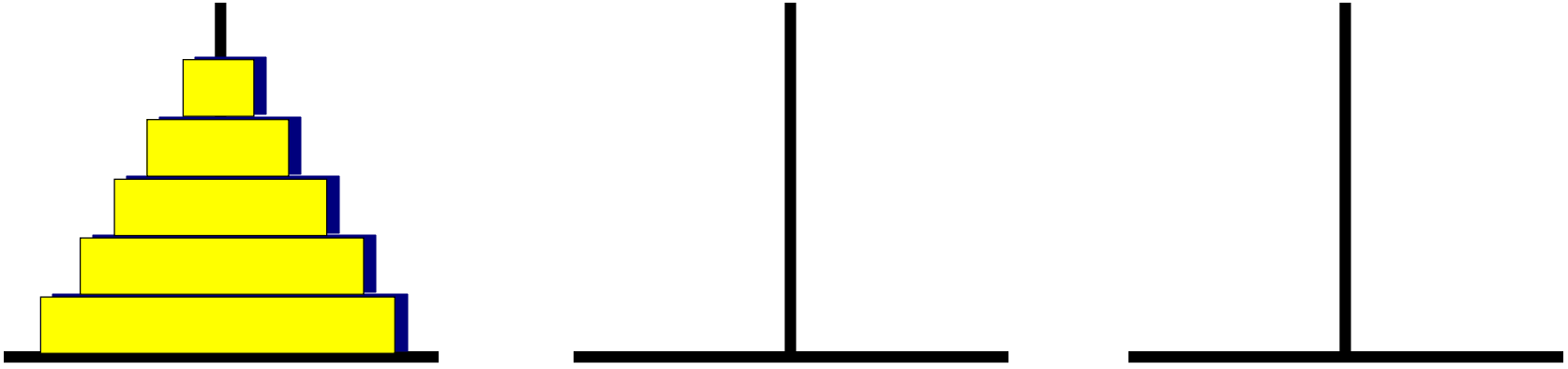


解 决

- n=1时，直接把圆盘从X移到Z
- n>1时
  - 先把上面n-1个圆盘从X移到Y
  - 然后将n号盘从X移到Z
  - 再将n-1个盘从Y移到Z

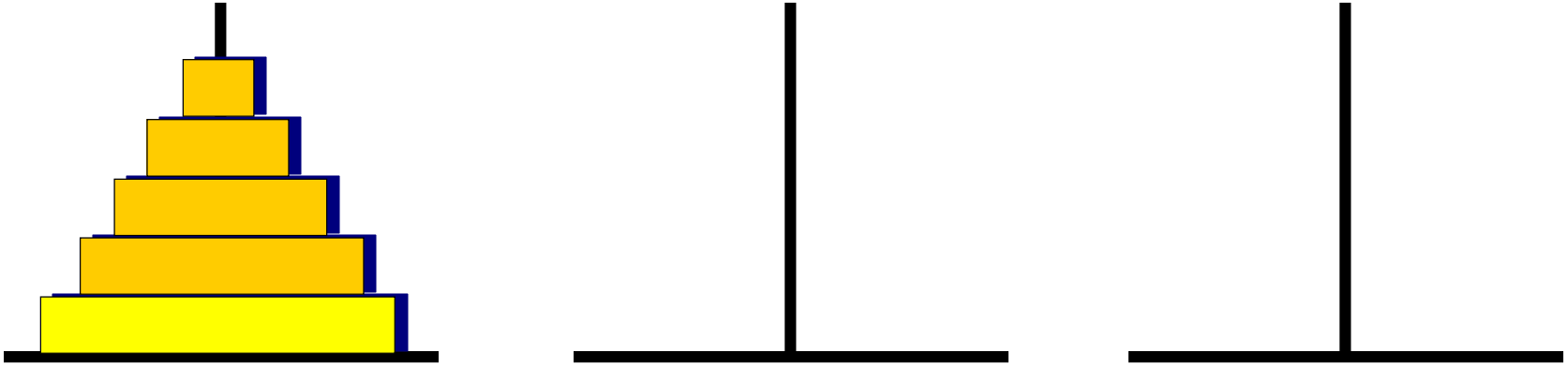
把求解n个圆盘的问题转化为求解n-1个圆盘的问题，依次类推，直至转化成只有一个圆盘的问题

4.梵塔问题执行情况



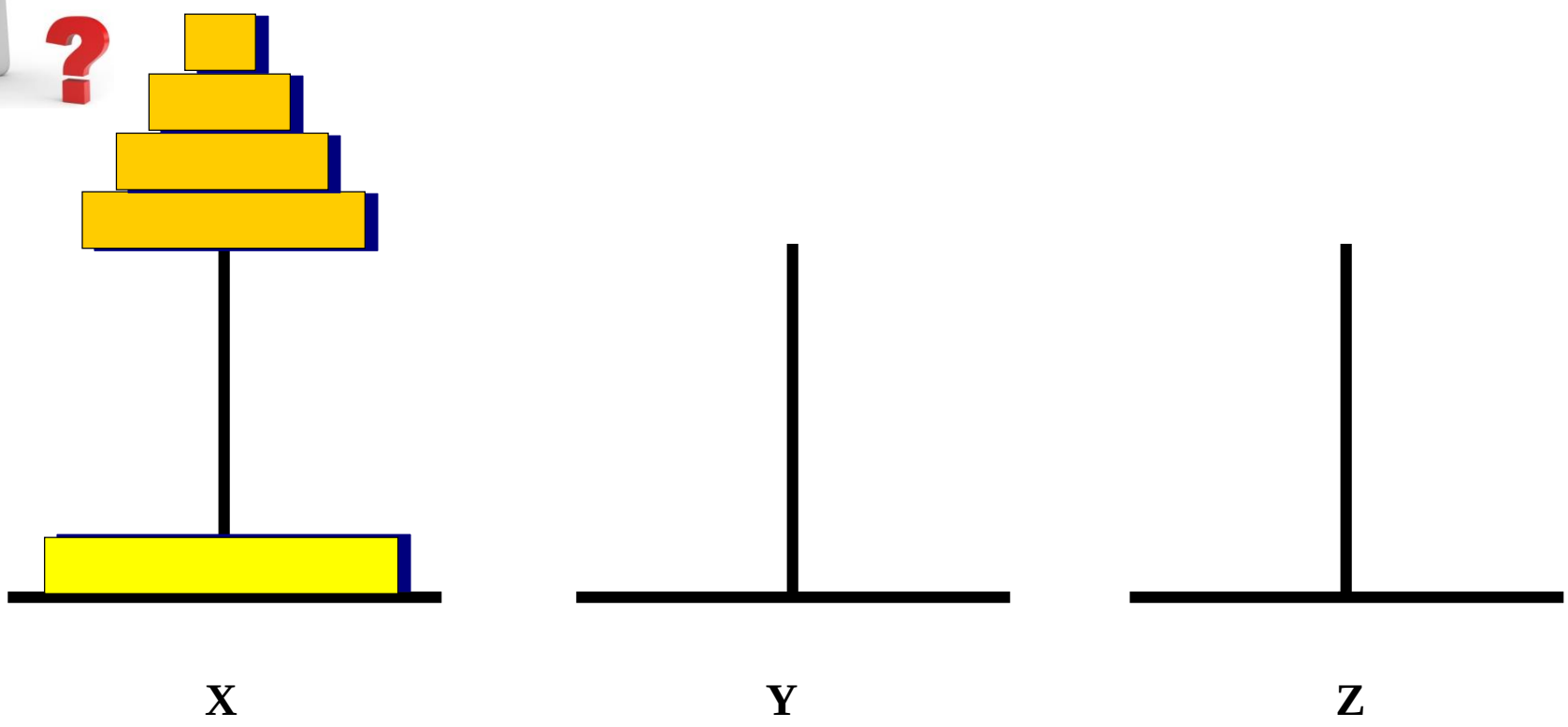
X　　Y　　Z

4.梵塔问题执行情况

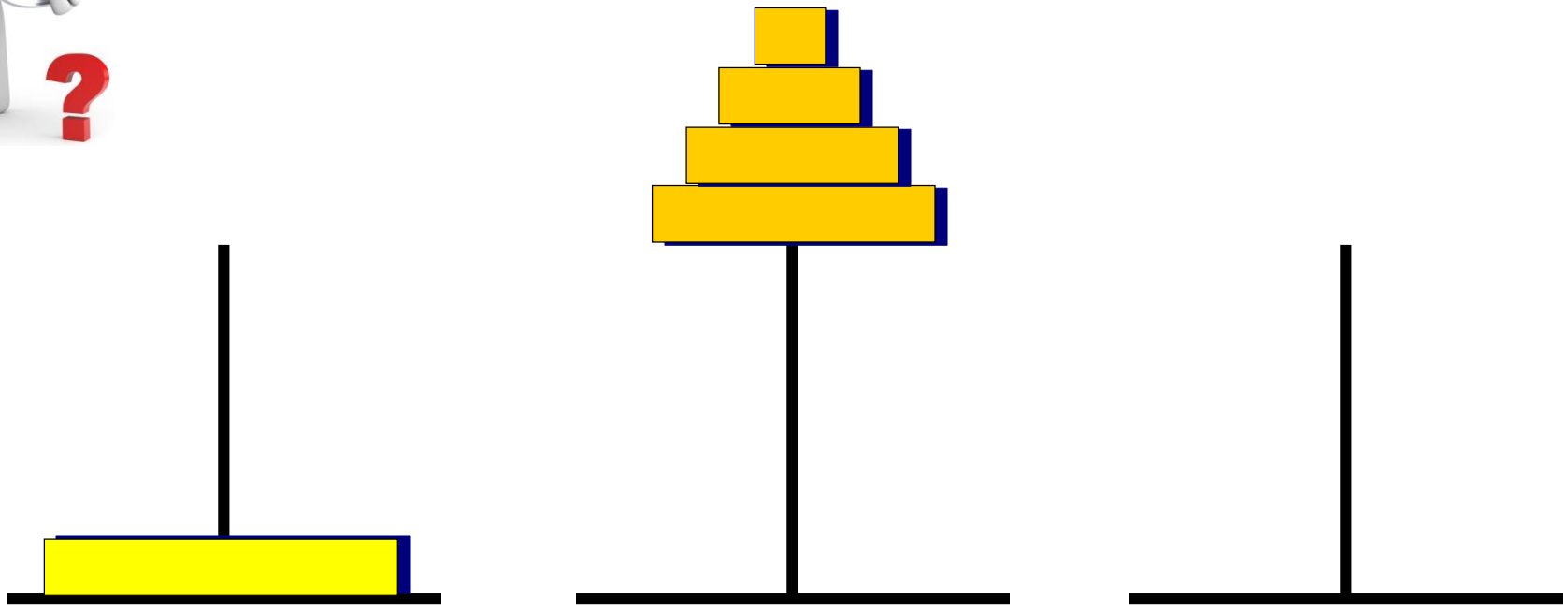


X　　Y　　Z

4.梵塔问题执行情况

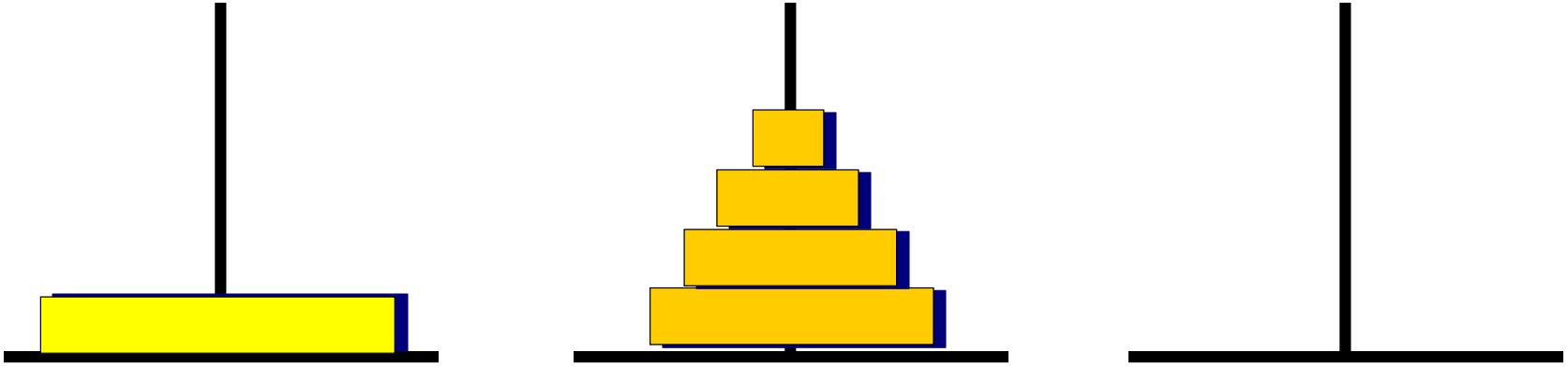


4.梵塔问题执行情况



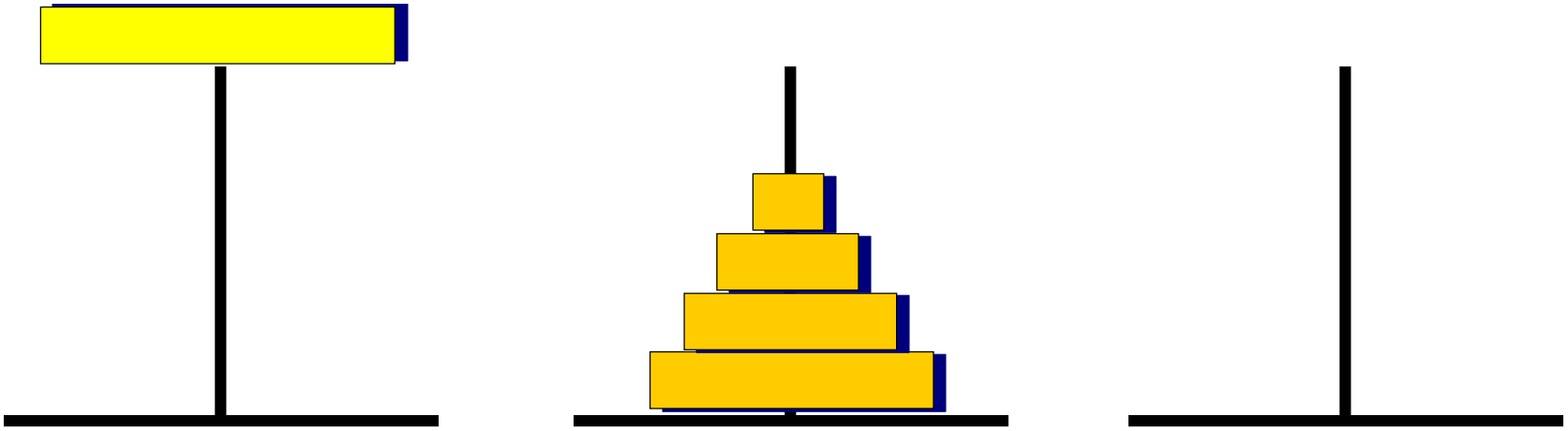
X　　Y　　Z

4.梵塔问题执行情况



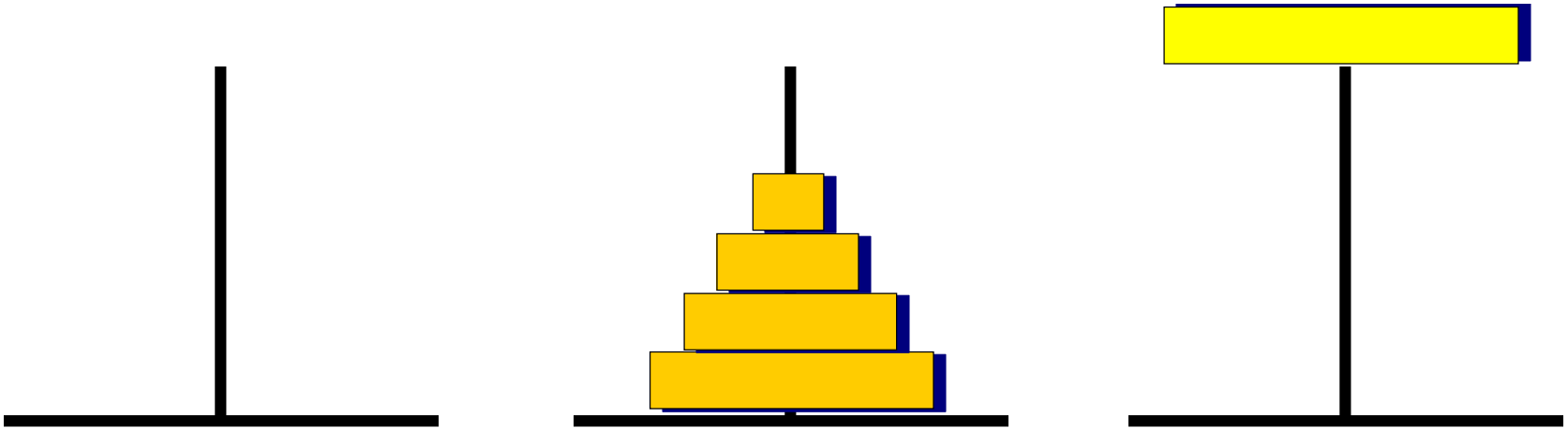
X　　Y　　Z

4.梵塔问题执行情况



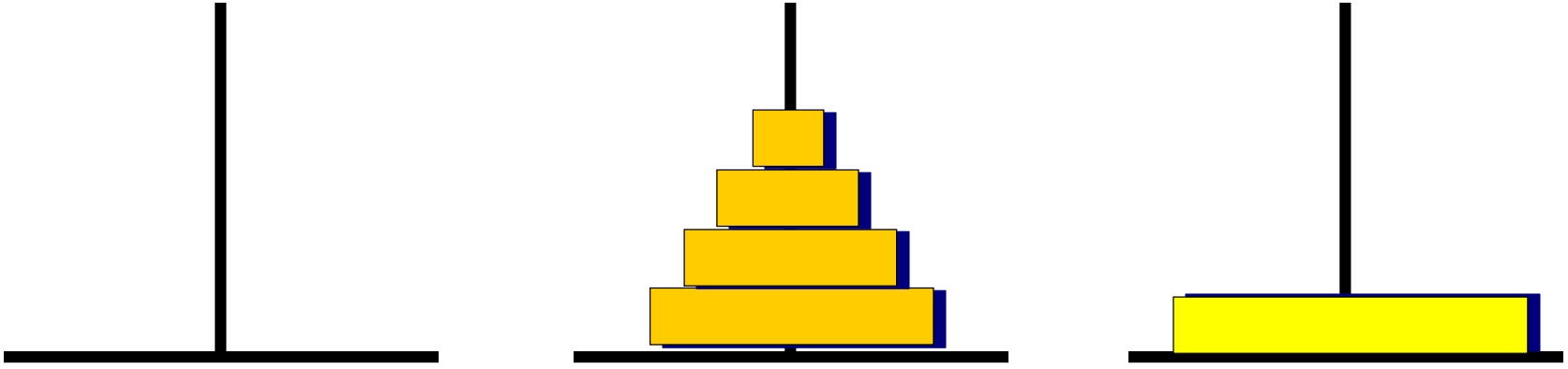
X　　Y　　Z

4.梵塔问题执行情况



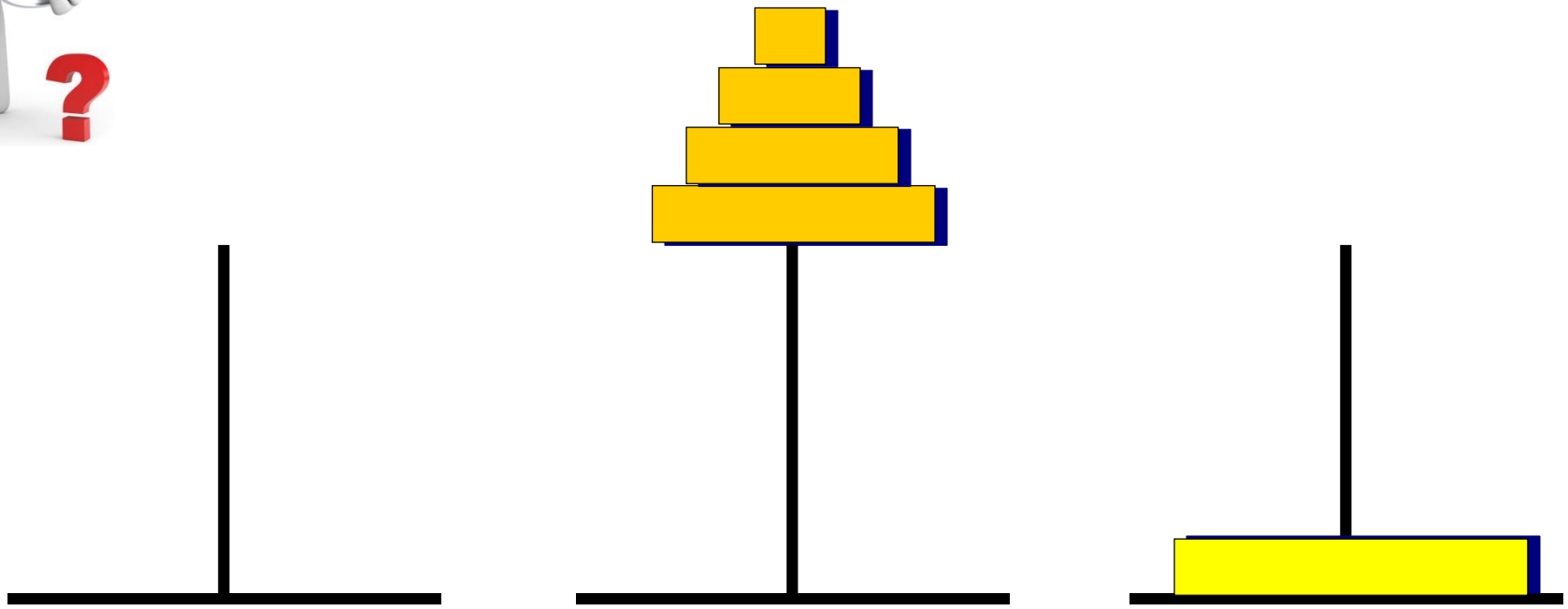
X　　Y　　Z

4.梵塔问题执行情况



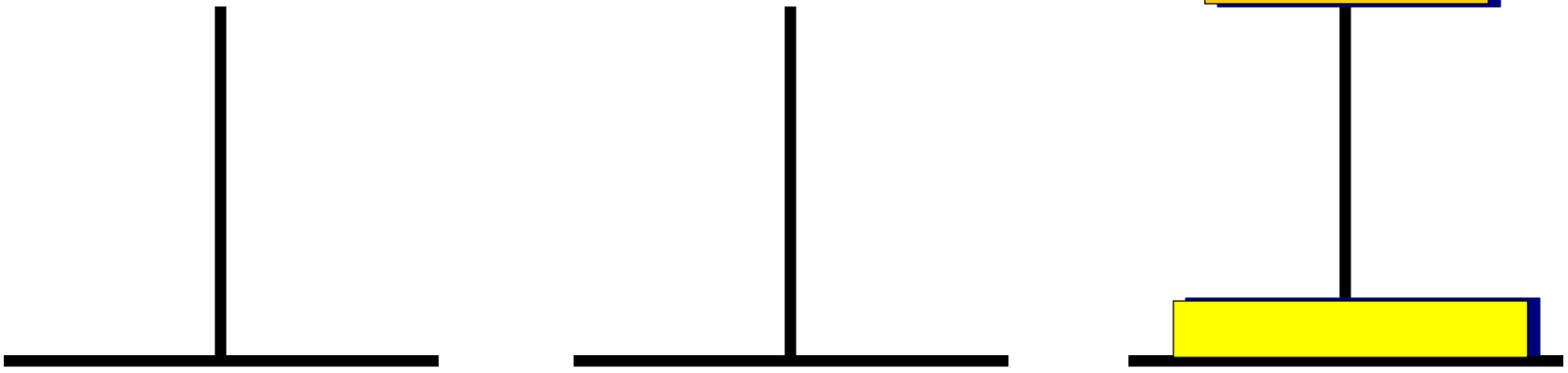
X　　Y　　Z

4.梵塔问题执行情况



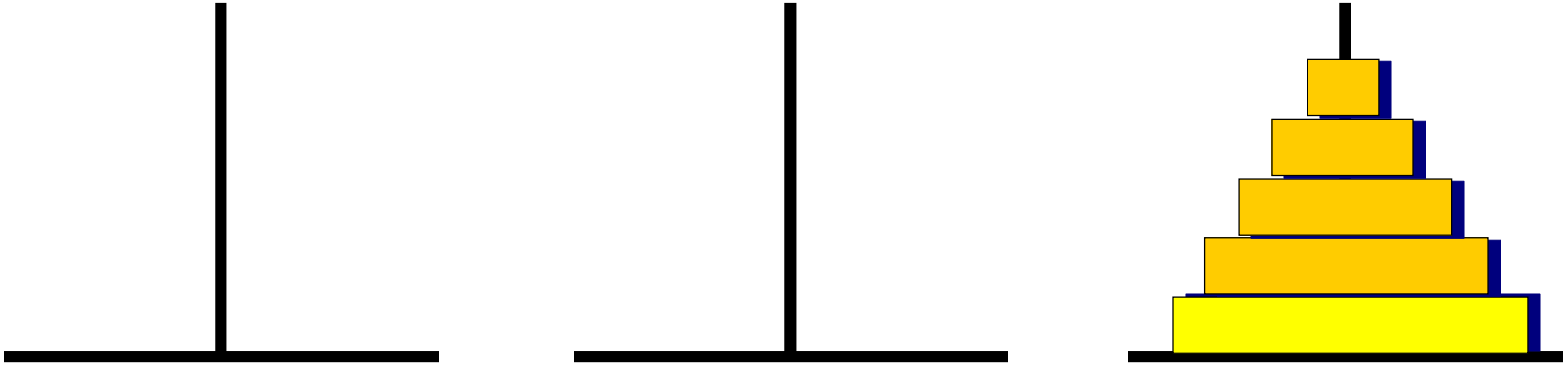
X　　Y　　Z

4.梵塔问题执行情况



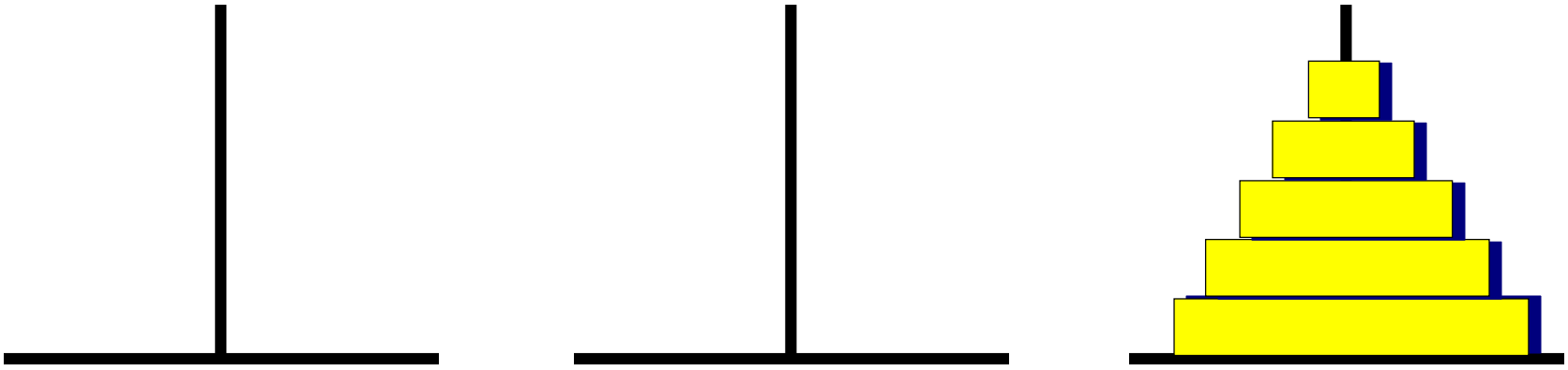
X　　Y　　Z

4.梵塔问题执行情况



X　　Y　　Z

4.梵塔问题执行情况

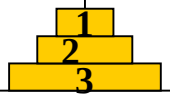


- 递归工作栈保存内容：形参n, x, y, z和返回地址
- 返回地址用行编号表示

n	x	y	z	返回地址

4.梵塔问题执行情况

```
main()
  {   int m;
      printf("Input number of disks ");
      scanf("%d",&m);
      printf(" Steps:%3d disks ",m);
      hanoi(m,'A','B','C');
(0) }
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)        move(1,x,z);
(4)     else{
(5)           hanoi(n-1,x,z,y);
(6)           move(n,x,z);
(7)           hanoi(n-1,y,x,z);
(8)        }
(9)  }
```



A B C

3	A	B	C	0

2	A	C	B	6
3	A	B	C	0

1	A	B	C	6
2	A	C	B	6
3	A	B	C	0

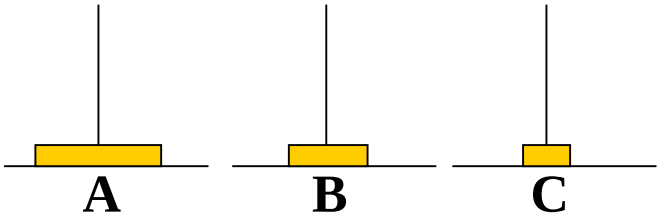
A B C

2	A	C	B	6
3	A	B	C	0

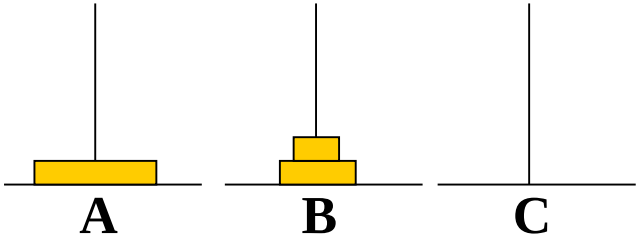
4.梵塔问题执行情况

```
main()
  {   int m;
      printf("Input number of disks" );
      scanf("%d",&m);
      printf(" Steps:%3d disks ",m);
      hanoi(m,'A','B','C');
(0) }
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)        move(1,x,z);
(4)     else{
(5)           hanoi(n-1,x,z,y);
(6)           move(n,x,z);
(7)           hanoi(n-1,y,x,z);
(8)        }
(9)  }
```

2	A	C	B	6
3	A	B	C	0



1	C	A	B	8
2	A	C	B	6
3	A	B	C	0



2	A	C	B	6
3	A	B	C	0

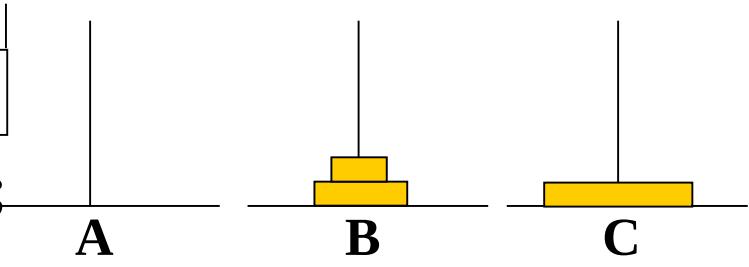
3	A	B	C	0

4.梵塔问题执行情况

```
main()
  {   int m;
      printf("Input number of disks " );
      scanf("%d",&m);
      printf(" Steps:%3d disks ",m);
      hanoi(m,'A','B','C');
(0) }
```

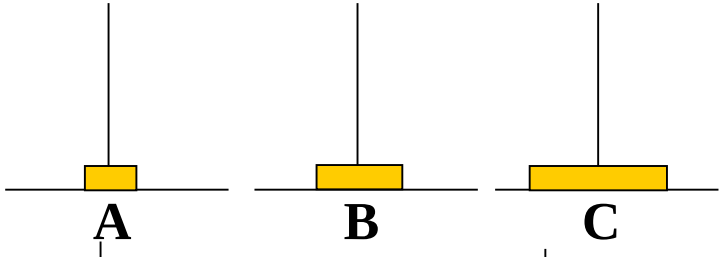
```
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)         move(1,x,z);
(4)     else{
(5)             hanoi(n-1,x,z,y);
(6)             move(n,x,z);
(7)             hanoi(n-1,y,x,z);
(8)         }
(9)  }
```

3	A	B	C	0



2	B	A	C	8
3	A	B	C	0

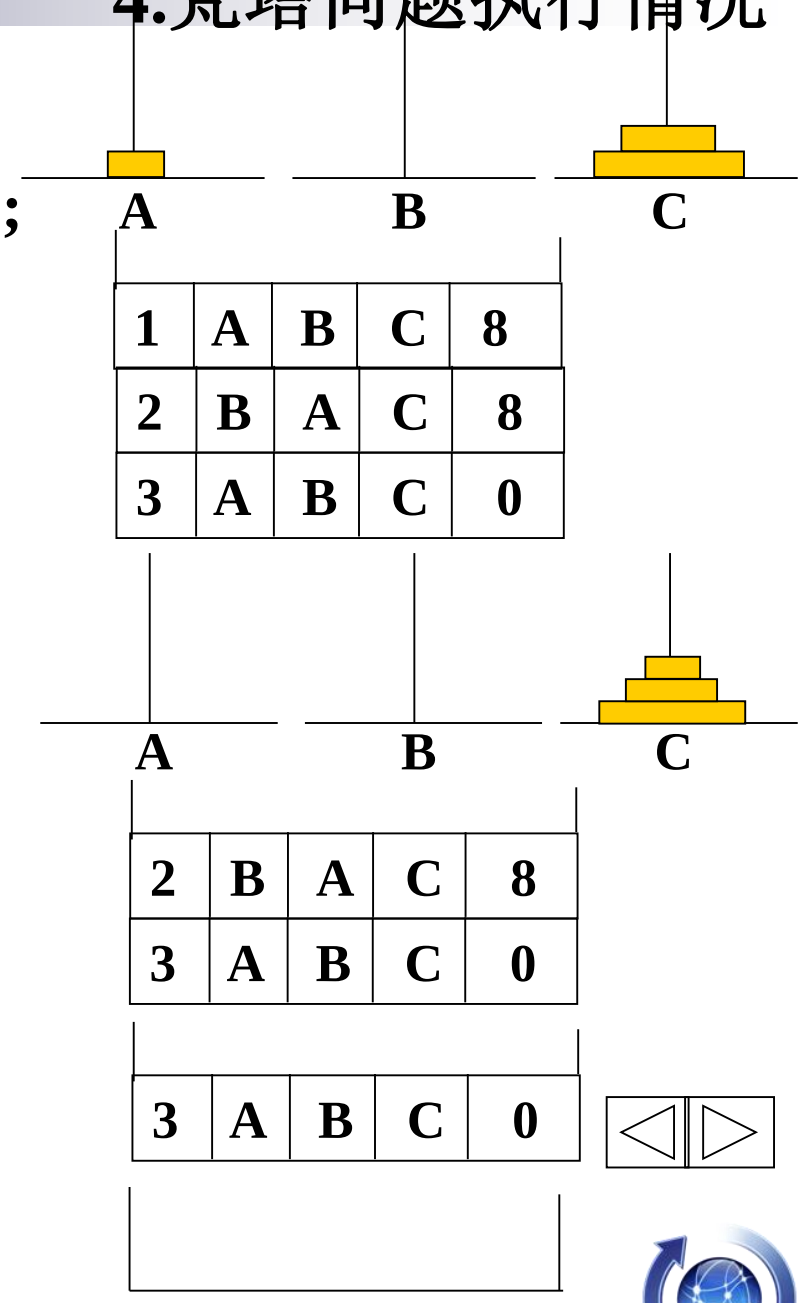
1	B	C	A	6
2	B	A	C	8
3	A	B	C	0



2	B	A	C	8
3	A	B	C	0

4.梵塔问题执行情况

```
main()
  {   int m;
      printf("Input number of disks" );
      scanf("%d",&m);
      printf(" Steps:%3d disks ",m);
hanoi(m,'A','B','C');
(0) }
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)         move(1,x,z);
(4)     else{
(5)             hanoi(n-1,x,z,y);
(6)             move(n,x,z);
(7)             hanoi(n-1,y,x,z);
(8)         }
(9)  }
```



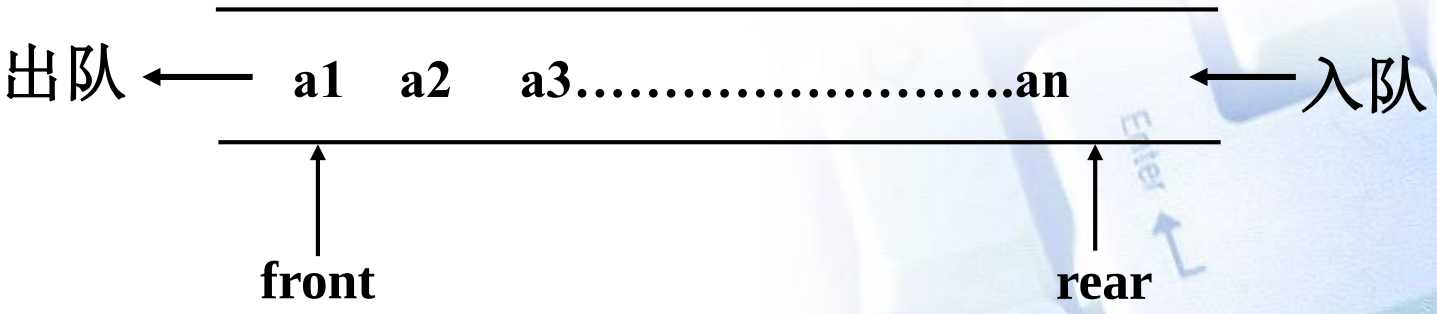
栈空

队 列

- 是限定只能在表的一端进行插入，在表的另一端进行删除的线性表
- 队尾(rear)——允许插入的一端
- 队头(front)——允许删除的一端

特 点

先进先出(FIFO)



- 队列的抽象数据类型定义
- 队列的链式表示和实现
- 队列的顺序表示和实现

定 义

ADT Queue {

数据对象：D＝{ai|ai∈ElemSet, i=1,2,...,n, n≥0}

数据关系：R1＝{ <ai-1,ai> | ai-1,ai∈D,i=2,...,n}

基本操作：

InitQueue(&Q)

操作结果：构造一个空队列 Q。

DestroyQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：队列 Q 被销毁，不再存在。

ClearQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

GetHead(Q,&e)

初始条件：Q 为非空队列。

操作结果：用 e 返回Q的队头元素。

QueueEmpty(Q)

初始条件：队列 Q 已存在。

操作结果：若 Q 为空队列，则返回TRUE，否则返回FALSE。

QueueLength(Q)

初始条件：队列 Q 已存在。

操作结果：返回 Q 的元素个数，即队列的长度

EnQueue(&Q,e)

初始条件：队列 Q 已存在。

操作结果：插入元素 e 为 Q 的新的队尾元素

DeQueue(&Q,&e)

初始条件：Q 为非空队列。

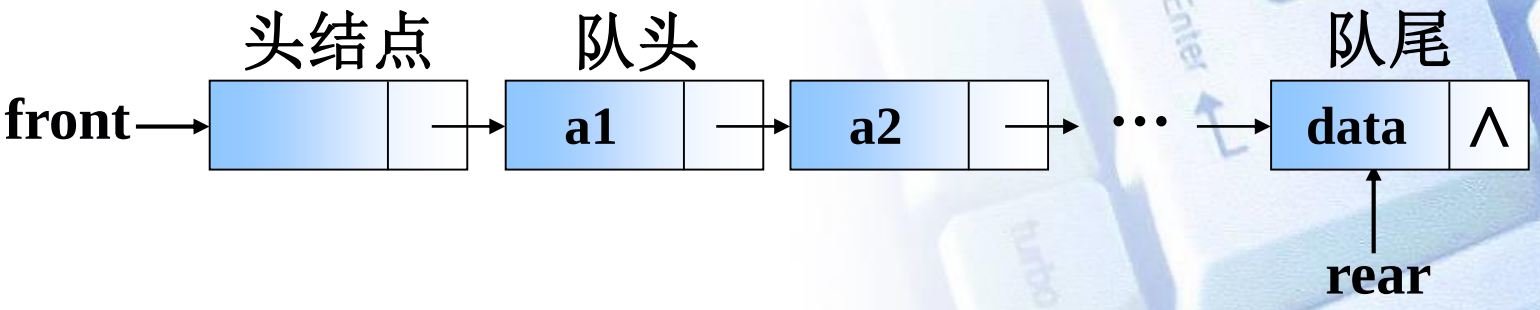
操作结果：删除 Q 的队头元素，并用 e 返回其值

} ADT Queue

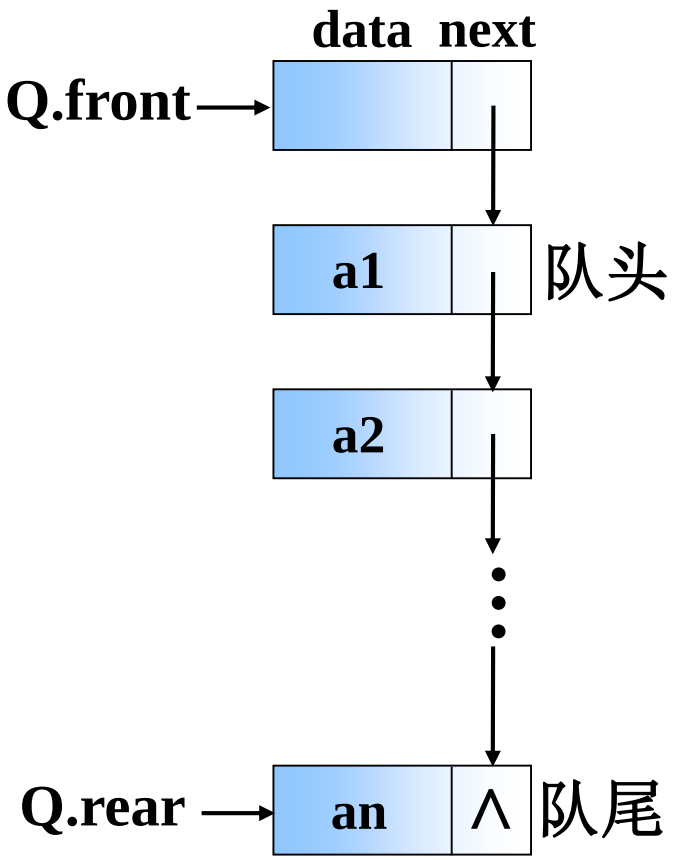
队列的链式表示和实现

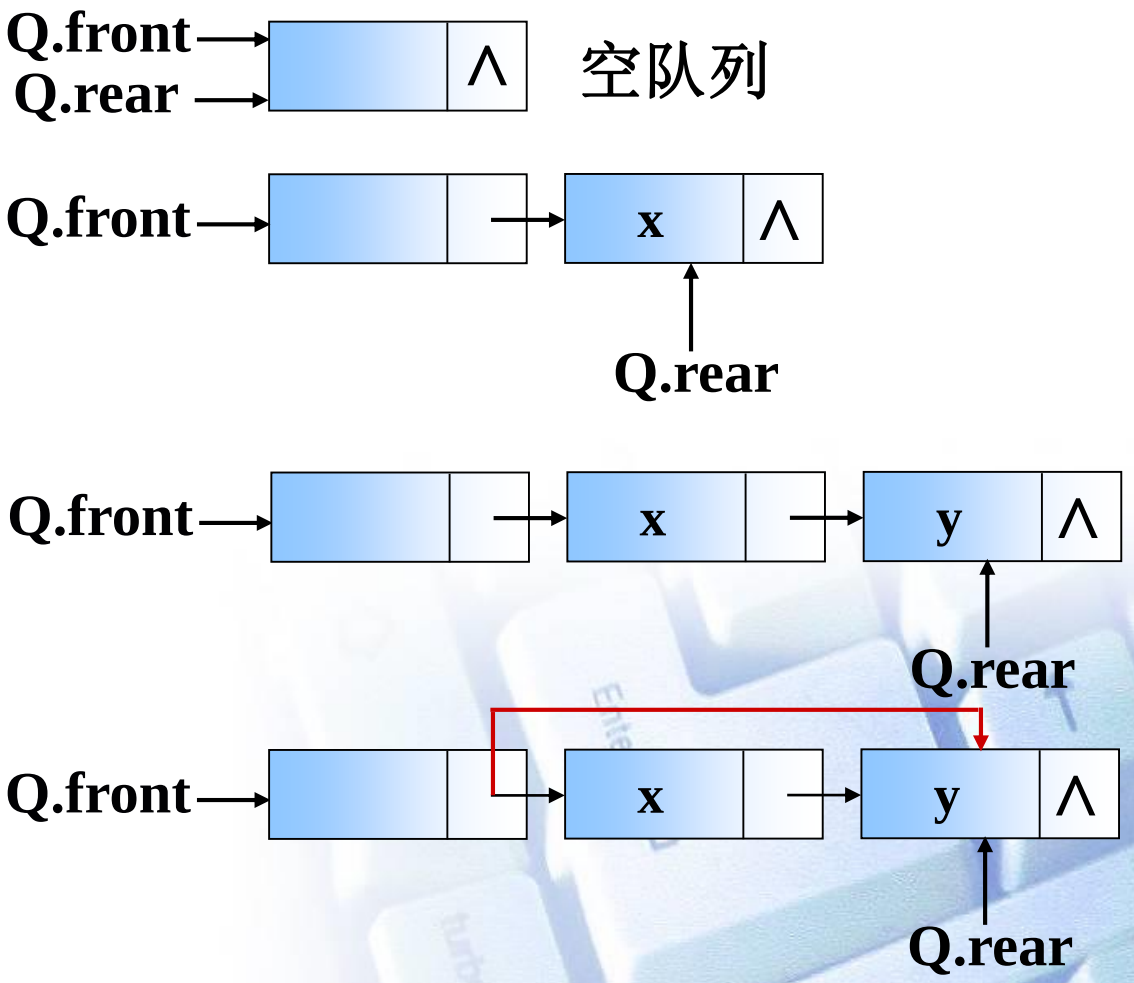
结构定义

```
Typedef   struct Qnode
        {  QElemType       data;
           struct Qnode     *next;
        } Qnode, *QueuePtr;
Typedef  struct
        {   QueuePtr  front;
            QueuePtr  rear;
        } LinkQueue ;
```



队列的链式表示和实现


data next
Q.front
a1
队头
a2
Q.rear
an
队尾


Q.front
Q.rear
空队列
Q.front
x
Q.rear
Q.front
x
y
Q.rear
Q.front
x
y
Q.rear

队列的链式表示和实现

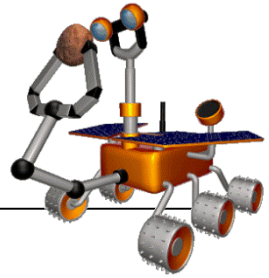
初始化 构造一个空队列 Q

入队列 在当前队列Q的尾元素之后，插入元素 e 为新的队列尾元素

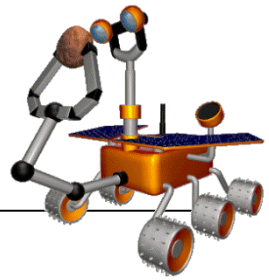
出队列 删除队列 Q 的队头元素，用 e 返回其值

初始化队列算法

```
Status InitQueue (LinkQueue &Q){
    // 构造一个空队列 Q
  Q.front=Q.rear=(QueuePtr * )malloc(sizeof(QNode));
  if (!Q.front) exit(OVERFOLW);   // 存储分配失败
    Q.front->next=NULL;
    return OK;
  }
```



```
Status EnQueue(LinkQueue &Q, QElemType e){
   // 在当前队列的尾元素之后，插入元素 e 为新的队列尾元素
   p=(QueuePtr * )malloc(sizeof(QNode));
   if (!p) exit(OVERFLOW);    // 存储分配失败
   p->data=e;
   p->next = NULL;
   Q.rear->next=p;                    // 修改尾结点的指针
   Q.rear=p;
   return OK;                         // 移动队尾指针
}
```

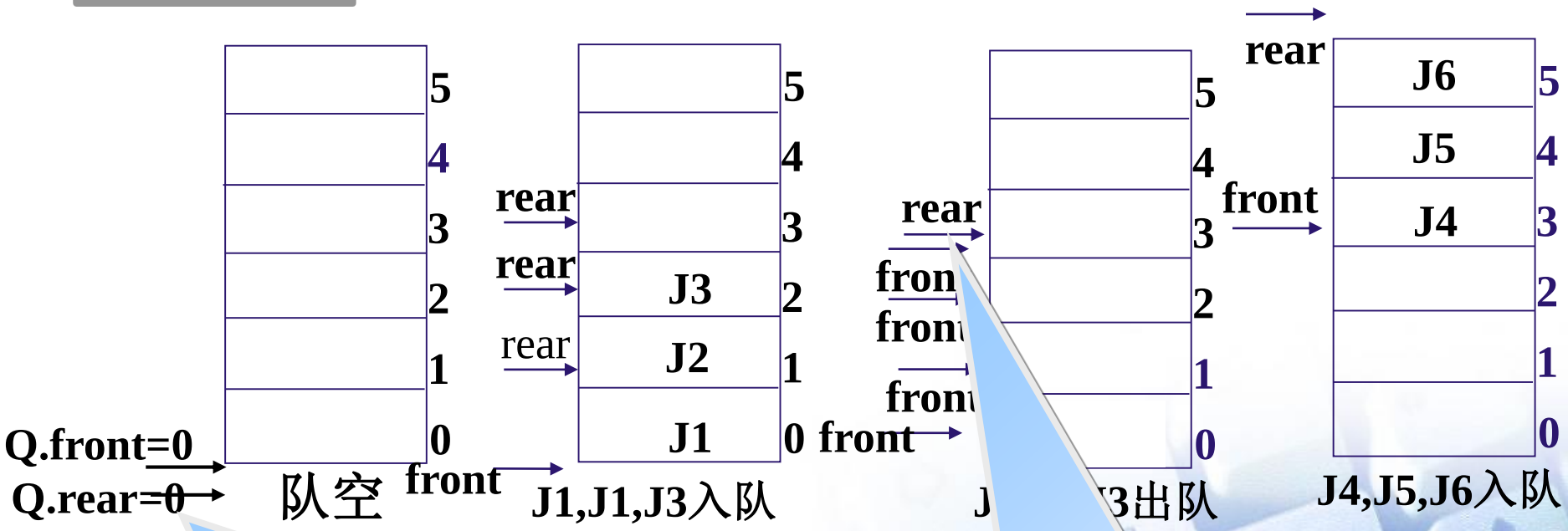


```
Status DeQueue(LinkQueue &Q,QElemType&e)
{// 若队列不空，则删除队列 Q 的队头元素，用 e 返回其值，并返回OK；否则返回ERROR
   if(Q.front==Q.rear) return ERROR;     // 链队列空
   p = Q.front->next;
   e = p->data;                          // 返回被删元素的值
   Q.front->next=p->next;                // 修改队头结点指针
   if(Q.rear==p)  Q.rear=Q.front;
   free(p);                              // 释放被删结点
   return OK;
} // DeQueue
```

队列的顺序表示和实现

循环队列

存储结构 顺序存储结构，用一维数组实现sq[M]



设front, rear, 约定：
Q.rear指示队尾元素的下一个位置
Q.front指示队头元素
初值Q.front=Q.rear=0

空队列条件：front==rear
入队列：sq[rear++]=x;
出队列：x=sq[front++];

队列的顺序表示和实现

循环队列

存在问题 设数组sq[M]大小为M，则：

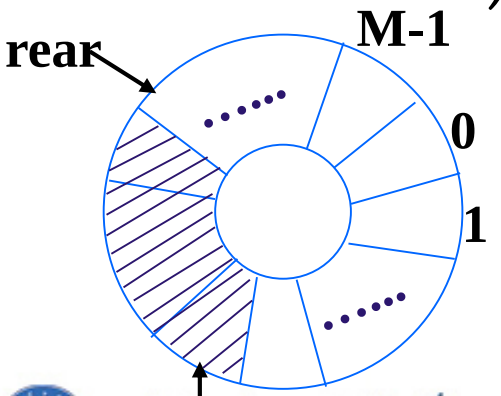
当front=0,rear=M时，再有元素入队发生溢出—真溢出

当front≠0,rear=M时，再有元素入队发生溢出—假溢出

解决方法

- 队首固定，每次出队剩余元素向下移动—浪费时间
- 循环队列，利用假溢出的空闲空间

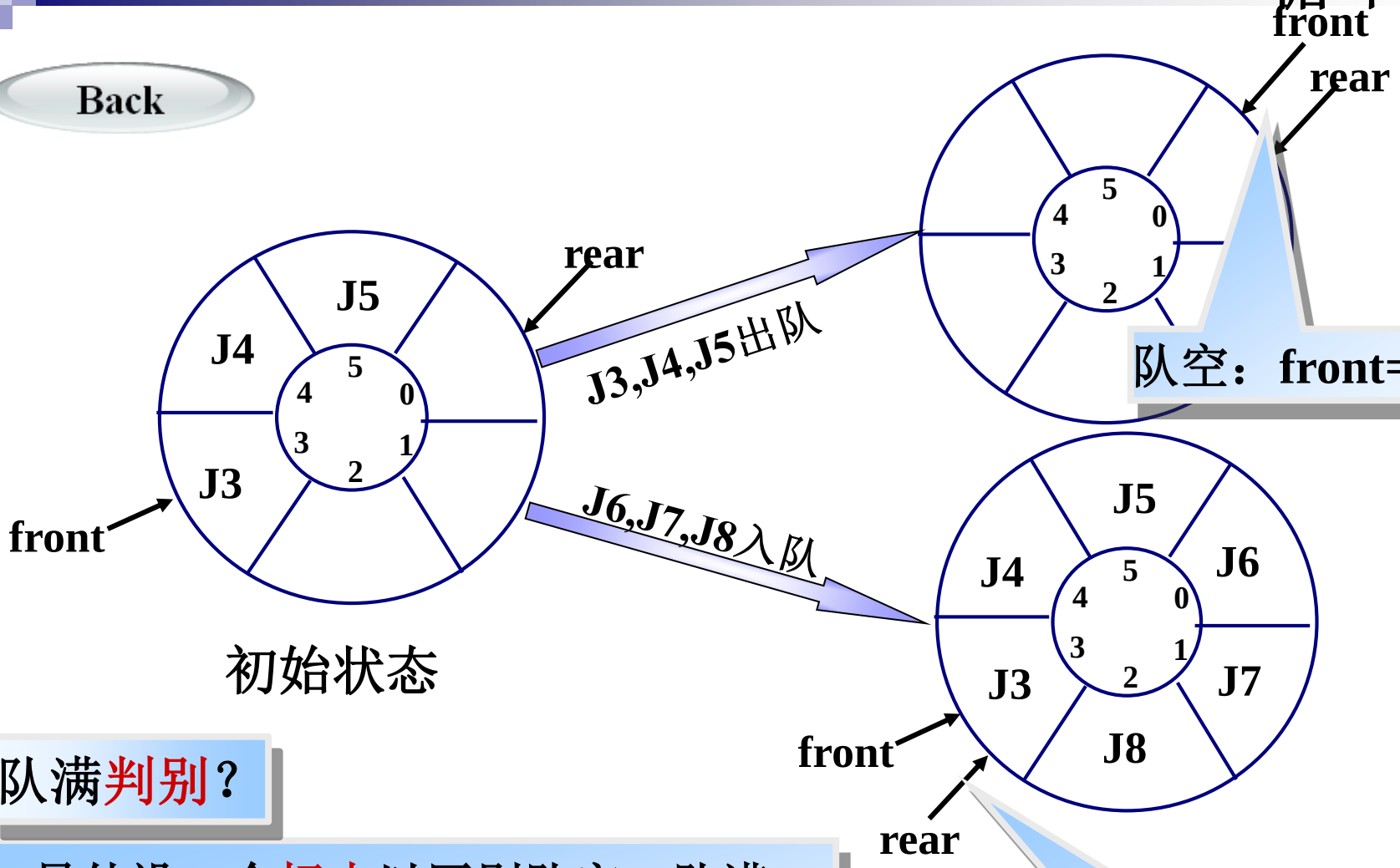
  把队列设想成环形，让sq[0]接在sq[M-1]之后，若rear==M, 则令rear=0; ？



- 实现：利用“模”运算
- 入队： sq[rear]=x; rear=(rear+1)%M;
- 出队： x=sq[front]; front=(front+1)%M;
- 队满、队空判定条件

循环队列

Back



队满判别？

1.另外设一个标志以区别队空、队满

2.少用一个元素空间：

队空：front==rear

队满：(rear+1)%M==front

定 义 循环队列的顺序存储结构

```
#define MAXQSIZE 100 // 最大队列长度
typedef struct {
    QElemType *base;// 初始化的动态分配存储空间
    int rear; // 队尾指针，指向队尾元素的下一个位置
    int front; // 队头指针，指向队头元素
} SqQueue;
```

操 作

初始化 构造一个空队列 Q

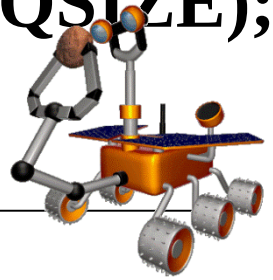
入队列 在当前队列Q的尾元素之后，插入元素 e 为新的队列尾元素

出队列 删除队列 Q 的队头元素，用 e 返回其值

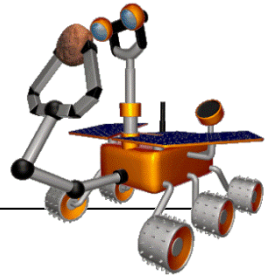
顺序存储队列初始化

```
Status InitQueue (SqQueue &Q){  // 构造一个空队列 Q
    Q.base = (QElemType *)malloc(MAXQSIZE
          *sizeof(QElemType)); // 为循环队列分配存储空间
    if (!Q.base) exit(OVERFLOW); // 存储分配失败
    Q.front = Q.rear = 0;
    return OK;
} // InitQueue


int QueueLength (SqQueue Q){
    // 返回队列Q中元素个数，即队列的长度
   return ((Q.rear-Q.front+MAXQSIZE) % MAXQSIZE);
}
```



```
Status EnQueue (SqQueue &Q, QElemType e)
{// 插入元素 e 为新的队列尾元素
   if((Q.rear+1)%MAXQSIZE==Q.front )
        return ERROR;                    // 队列满
   Q.base[Q.rear] = e;
   Q.rear = (Q.rear+1) % MAXQSIZE;
        return OK;
}
```



顺序存储队列出队列算法

```
Status DeQueue (SqQueue &Q, QElemType &e){
   // 若队列不空，则删除当前队列Q中的头元素，
   //用 e 返回其值,并返回OK
   if (Q.front == Q.rear)    return ERROR;
   e = Q.base[Q.front];
   Q.front = (Q.front+1) % MAXQSIZE;
   return OK;
}
```

本章小结

- 栈和队列都属线性结构，因此它们的存储结构和线性表非常类似，同时由于它们的基本操作要比线性表简单得多，因此它们在相应的存储结构中实现的算法都比较简单，相信对大家来说都不是难点。
- 这一章的重点则在于栈和队列的应用。通过本章所举的例子学习分析应用问题的特点，在算法中适时应用栈和队列。

本章结束啦！