



信息与软件工程学院

# 程序设计与算法基础II

主讲教师：陈安龙

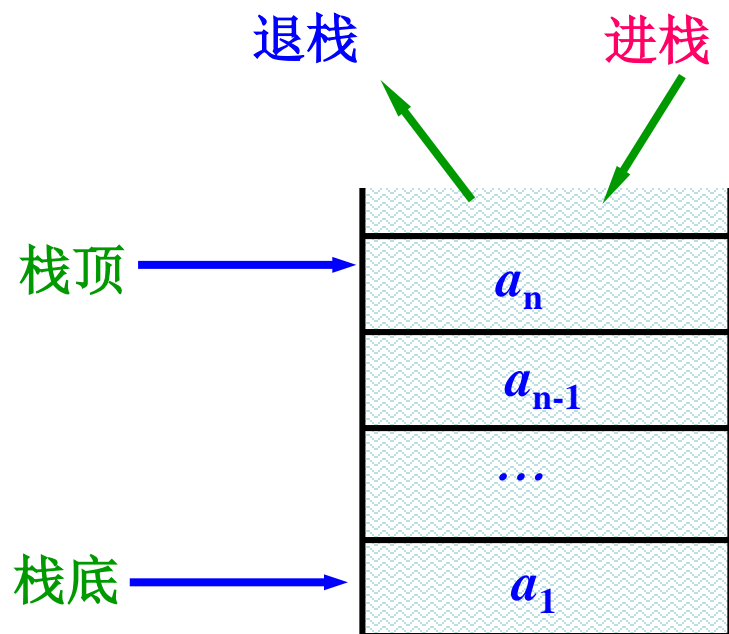
## 第3章 限定性线性表---栈和队列

- 栈的定义
- 栈的顺序存储
- 栈的链式存储
- 栈的应用
- 队列的定义
- 队列的顺序存储
- 队列的链式存储
- 队列的应用

## 3.1 栈

### ■ 栈 (stack)

- ✓ 栈和队列是特殊的线性表，是操作受限的线性表，称限定性数据结构
- ✓ 特点：先进后出（FILO）或后进先出（LIFO）



- ① 允许进行插入、删除操作的一端称为栈顶。
- ② 表的另一端称为栈底。
- ③ 当栈中没有数据元素时，称为空栈。
- ④ 栈的插入操作通常称为进栈或入栈。
- ⑤ 栈的删除操作通常称为退栈或出栈。

**思考题：** 栈和线性表有什么不同？

【例3-1】 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.  $c, d, b, a$

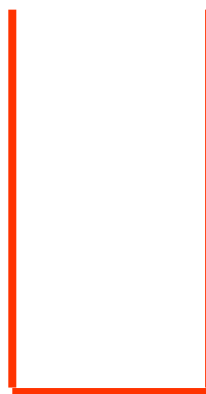
B.  $d, c, b, a$

C.  $a, c, d, b$

D.  $d, a, b, c$

选项D是不可能的？

$d \quad c \quad b \quad a$



栈

下一步不可能出栈 $a$

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

A.  $n-3$

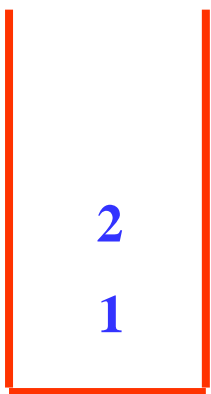
B.  $n-2$

C.  $n-1$

D. 无法确定

1、2、3进栈，3出栈的结果：

$n \dots 4$



栈



不可能是 3，共有  $n-1$  可能

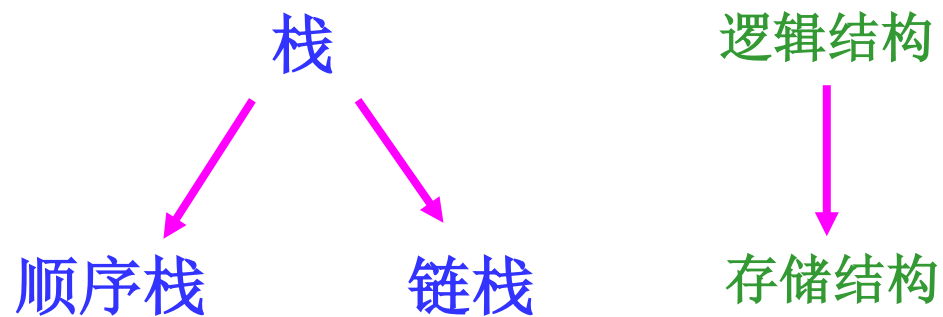
可以一进栈就出栈  
也可以进栈多次后再出一次  
栈

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：

- ① **InitStack(s)**: 初始化栈。构造一个空栈s。
- ② **ClearStack(s)**: 栈已经存在，将栈置为空栈。
- ③ **IsEmpty(s)**: 判断栈是否为空:若栈s为空，则返回真；否则返回假。
- ④ **IsFull(s)**: 判断栈是否为满:若栈s为满，则返回真；否则返回假。
- ⑤ **Push(s,x)**: 进栈。将元素 $e$ 插入到栈s中作为栈顶元素。
- ⑥ **Pop(s,x)**: 出栈。从栈s中退出栈顶元素，并将其值赋给 $x$ 。
- ⑦ **GetTop(s,x)**: 取栈顶元素。返回当前的栈顶元素，并将其值赋给 $x$ 。

栈中元素逻辑关系与线性表的相同，**栈可以采用与线性表相同的存储结构。**



# 栈的顺序存储结构及其基本运算实现

假设栈的元素个数最大不超过正整数`StackSize`，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`SeqStack`:

```
#define TRUE 1
#define FALSE 0
# define StackSize 50 //定义栈的大小
typedef struct
{
    ElemType data[StackSize];
    int top;           //栈顶指针
} SeqStack;
```

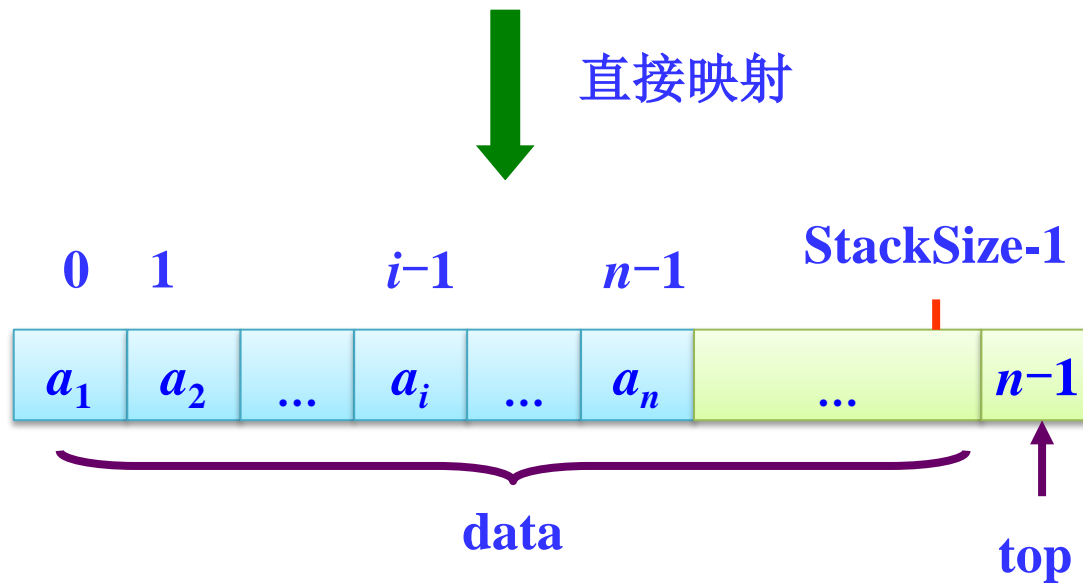


逻辑结构



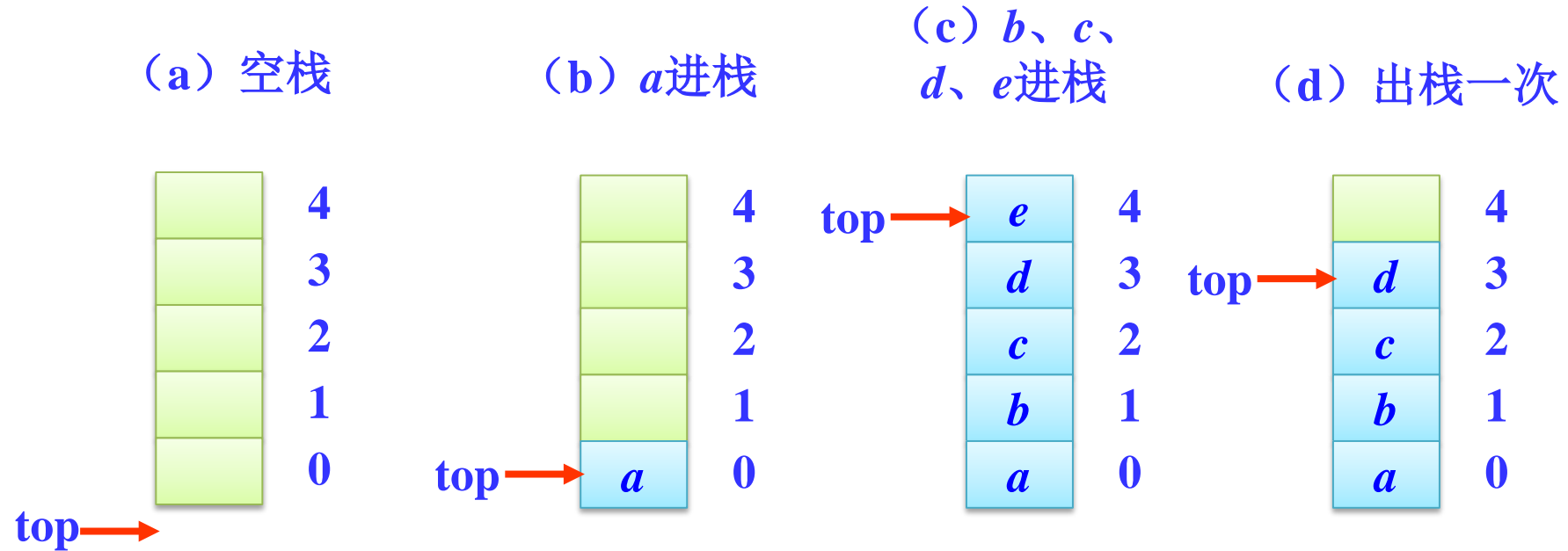
直接映射

存储结构



顺序栈的示意图

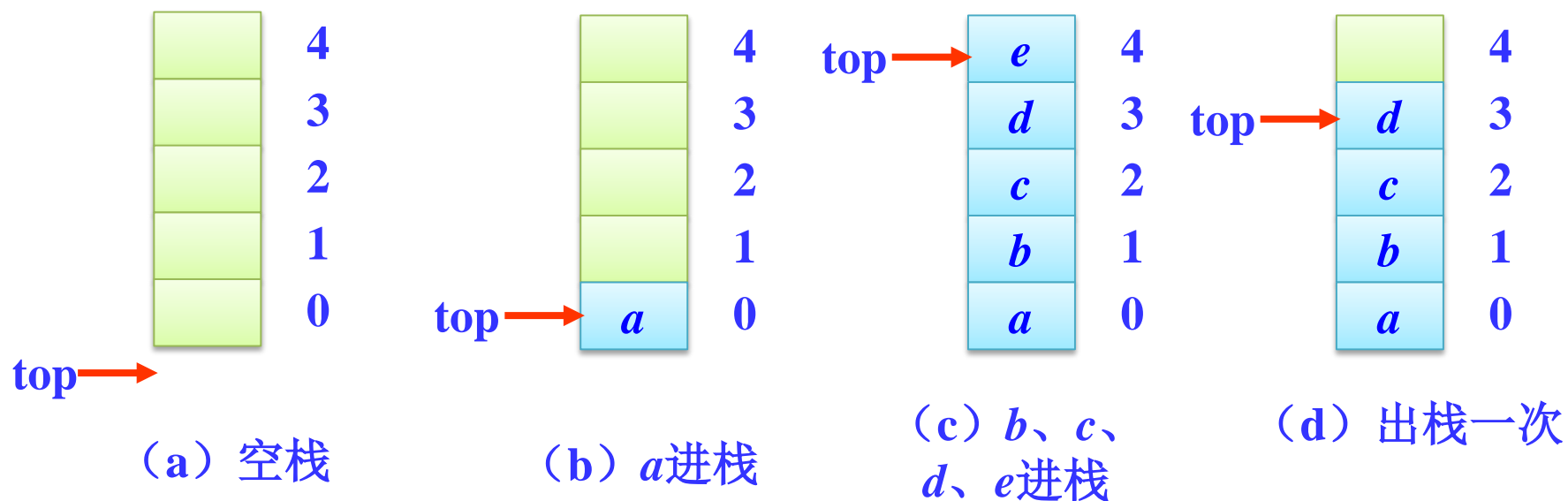
例如: StackSize=5



总结:

- ① 约定 $top$ 总是指向栈顶元素, 初始值为-1
- ② 当 $top=StackSize-1$ 时不能再进栈—栈满
- ③ 进栈时 $top$ 增1, 出栈时 $top$ 减1

## 顺序栈的各种状态



### 顺序栈4要素:

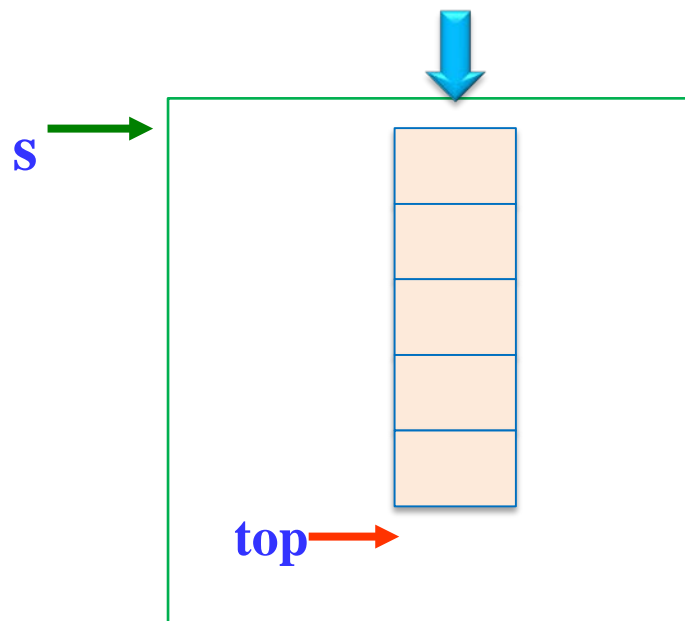
- ① 栈空条件:  $\text{top} = -1$
- ② 栈满条件:  $\text{top} = \text{StackSize} - 1$
- ③ 进栈 $e$ 操作:  $\text{top}++$ ; 将 $e$ 放在 $\text{top}$ 处
- ④ 退栈操作: 从 $\text{top}$ 处取出元素 $e$ ;  $\text{top}--$ ;

在顺序栈中实现栈的基本运算算法。

### (1) 初始化栈InitStack(SeqStack \*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(SeqStack *s)
{
    //构造一个空栈
    s->top=-1;
}
```



**注意：** s为栈指针，  
top为s所指栈的栈顶指针

## (2) 判断栈是否为空 `IsEmpty(SeqStack *s)`

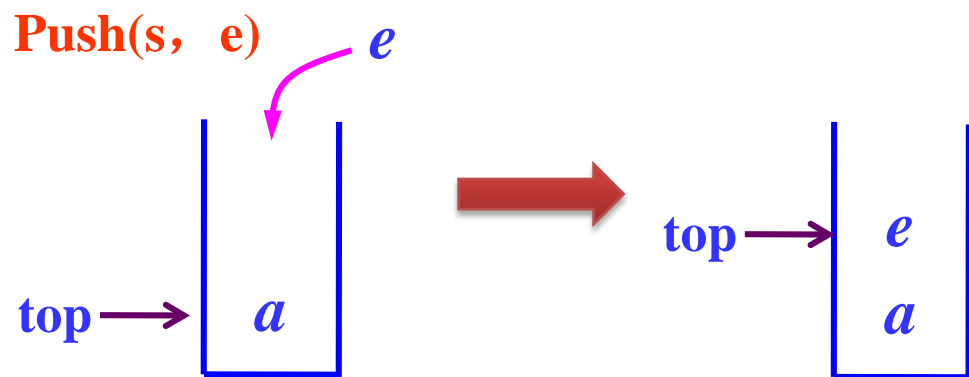
栈S为空的条件是 `s->top == -1`。

```
int IsEmpty(SeqStack *s)
{
    return(s->top == -1);
}
```

### (3) 进栈Push(SeqStack \*s, ElemType 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 $e$ 。

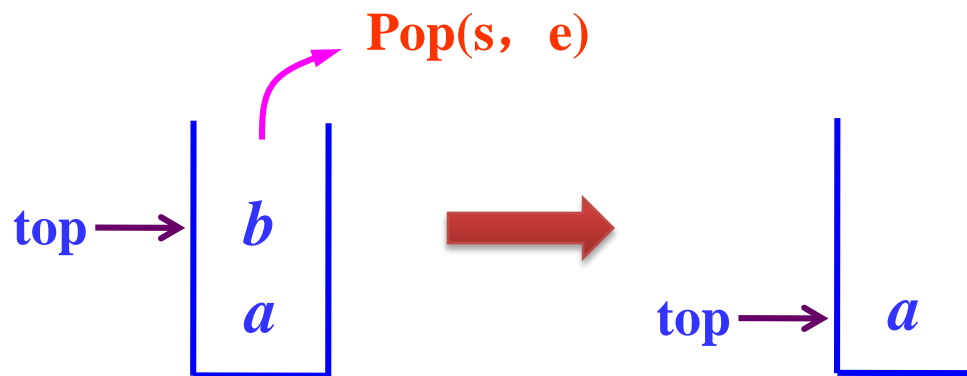
```
int Push(SeqStack *s, ElemType e)
{   if (s->top==StackSize-1)    //栈满的情况，即栈上溢出
    return 0;
    s->top++;                    //栈顶指针增1
    s->data[s->top]=e;           //元素 $e$ 放在栈顶指针处
    return 1;
}
```



#### (4) 出栈Pop(SeqStack \*s, ElemType \*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。

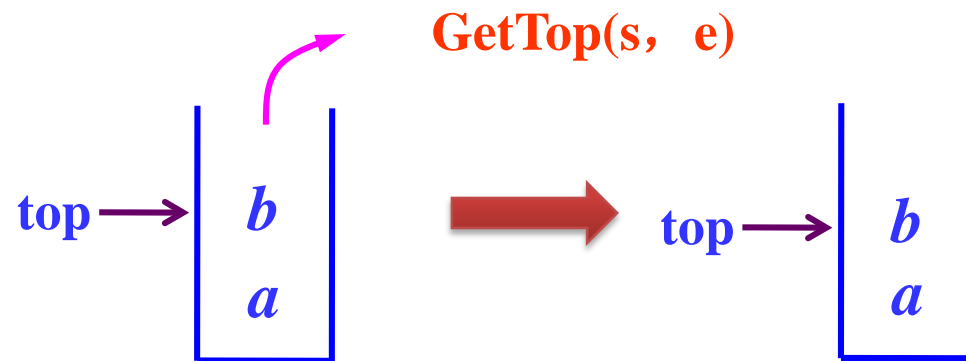
```
int Pop(SeqStack *s, ElemType *e)
{  if (s->top==-1)           //栈为空的情况，即栈下溢出
    return 0;
    *e=s->data[s->top]; //取栈顶指针元素的元素
    s->top--;           //栈顶指针减1
    return 1;
}
```



## (5) 取栈顶元素 `GetTop(SeqStack *s, ElemType *e)`

在栈不为空的条件下，将栈顶元素赋给 $e$ 。

```
int GetTop(SeqStack *s, ElemType *e)
{
    if (s->top == -1)           //栈为空的情况，即栈下溢出
        return 0;
    *e = s->data[s->top];      //取栈顶指针元素的元素
    return 1;
}
```





**【例1】** 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。

将从栈中弹出的元素与字符串按顺序  
比对

```
int symmetry(ElemType str[])
{   int i; ElemType e; SeqStack st;
    InitStack(st);
```

//初始化栈

```
    for (i=0;str[i]!='\0';i++)
        Push(&st, str[i]);
```

//将串所有元素进栈  
//元素进栈

→ str的所有元素依次进栈

```
    for (i=0;str[i]!='\0';i++)
    {   Pop(&st, e);
        if (str[i]!=e)
        {
            return 0;
        }
    }
```

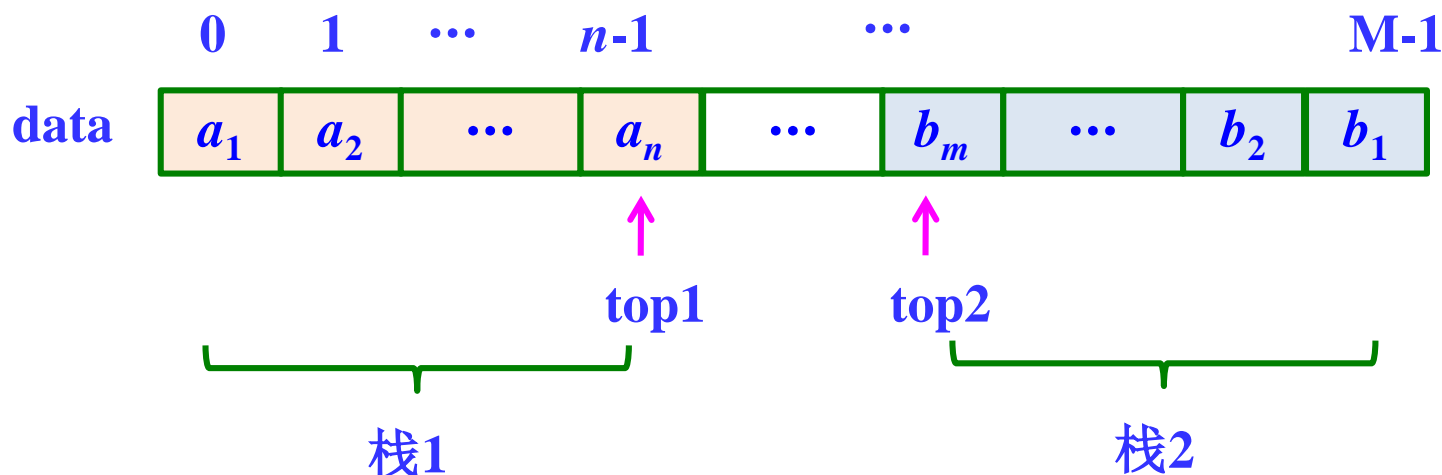
//退栈元素e  
//若e与当前串元素不同则不是对称串

```
    return 1;
}
```

判断正反序是否相同

## 多栈共享技术

如果需要用到两个相同类型的栈，可以用一个数组`ata[0...M-1]`来实现这两个栈，这称为**共享栈**。



共享栈类型:

```
#define TRUE  1
#define FALSE 0
#define M 100
typedef struct
{   ElemType Stack[M];           //存放共享栈中元素
    int top[2];                  //两个栈的栈顶指针
} DqStack;
```

## 1) 两栈共享的初始化操作算法

```
void InitStack(DqStack *S)  
{  
    S->top[0]=-1;  
    S->top[1]=M;  
}
```

## 2) 两栈共享的进栈操作算法

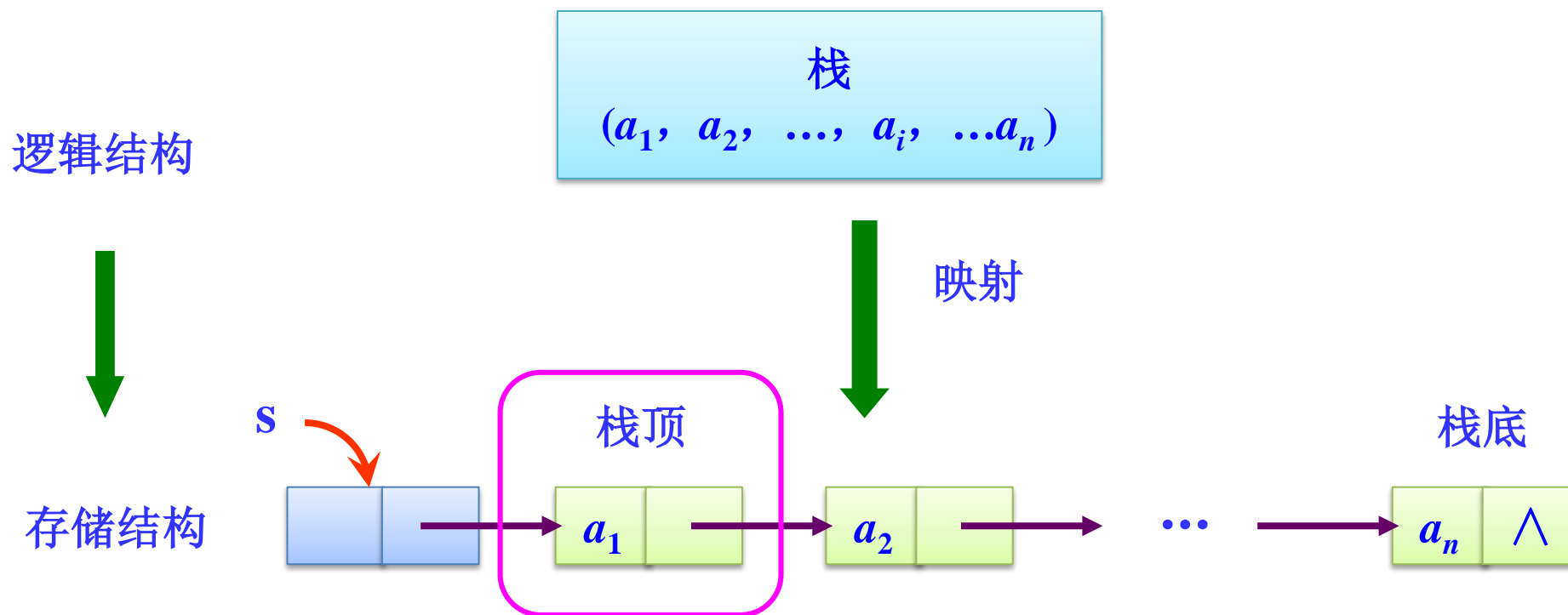
```
int Push(DqStack *S, StackElementType x, int i)
{
    if(S->top[0]+1==S->top[1]) /*栈已满*/
        return(FALSE);
    switch(i)
    {
        case 0: S->top[0]++; S->Stack[S->top[0]]=x; break;
        case 1: S->top[1]--; S->Stack[S->top[1]]=x; break;
        default: return(FALSE)
    }
    return(TRUE);
}
```

### 3) 两栈共享的出栈操作算法

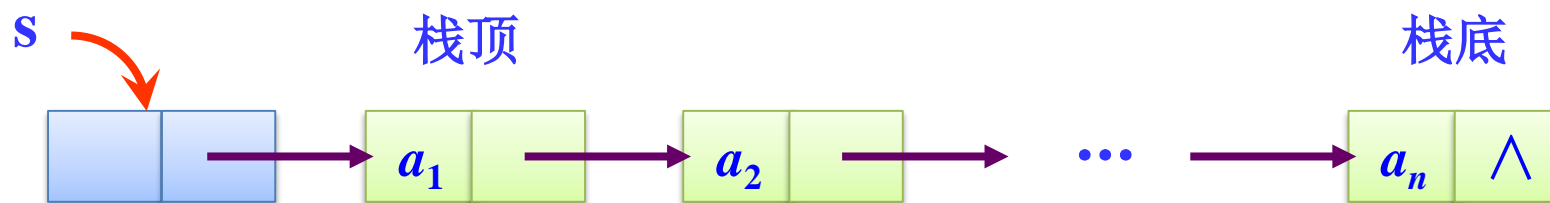
```
int Pop(DqStack *S, StackElementType *x, int i)
{switch(i)
{  case 0: if(S->top[0]==-1) return(FALSE);
    *x=S->Stack[S->top[0]]; S->top[0]--; break;
case 1: if(S->top[1]==M) return(FALSE);
    *x=S->Stack[S->top[1]]; S->top[1]++;break;
default: return(FALSE);
}
return(TRUE);
}
```

## 2 栈的链式存储结构及其基本运算的实现

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:  $s \rightarrow \text{next} = \text{NULL}$
- 栈满条件: 不考虑
- 进栈 $e$ 操作: 将包含 $e$ 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LinkStackNode定义如下:

```
typedef struct node
```

```
{
```

```
    ElementType data; //数据域
```

```
    struct node *next; //指针域
```

```
} LinkStackNode;
```

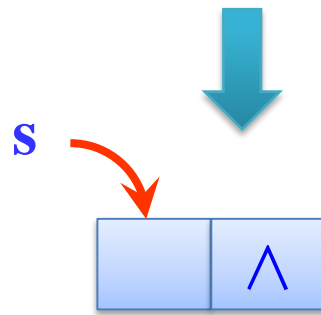
```
typedef LinkStackNode *LinkStack;
```

在链栈中，栈的基本运算算法如下。

### (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 (LinkStack *s)
{
    *s=(LinkStack)malloc(sizeof(LinkStackNode));
    (*s)->next=NULL;
}
```



## (2) 销毁栈DestroyStack(s)

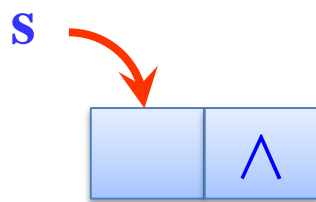
释放栈s占用的全部存储空间。

```
void DestroyStack(LinkStackNode *s)
{   LinkStackNode *p=s, *q=s->next; ➔ 栈顶
    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 $s \rightarrow \text{next} == \text{NULL}$ ，即单链表中没有数据结点。

```
int StackEmpty(LinkStackNode *s)
{
    return(s->next==NULL);
}
```

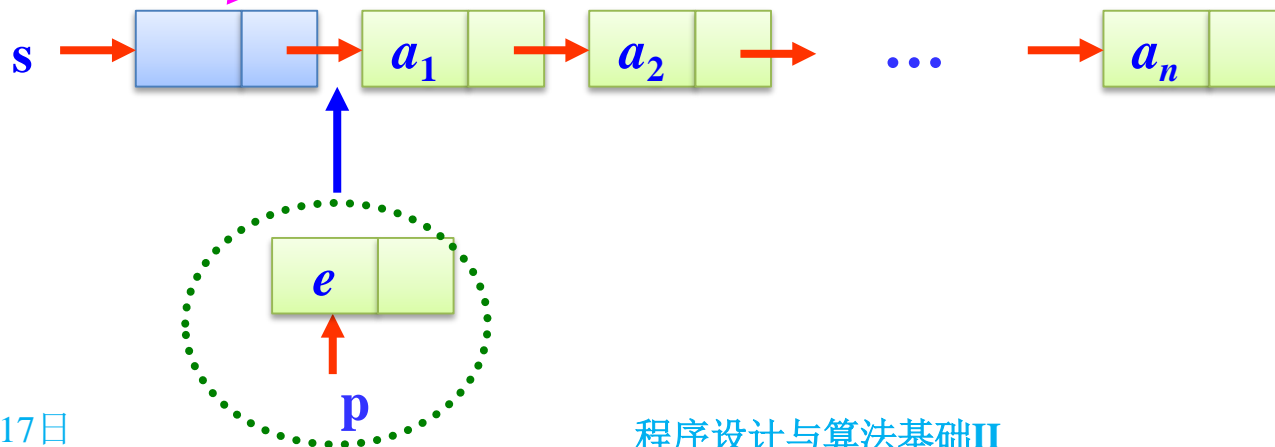


空栈的情况

#### (4) 进栈Push(s, e)

将新数据结点插入到头结点之后。

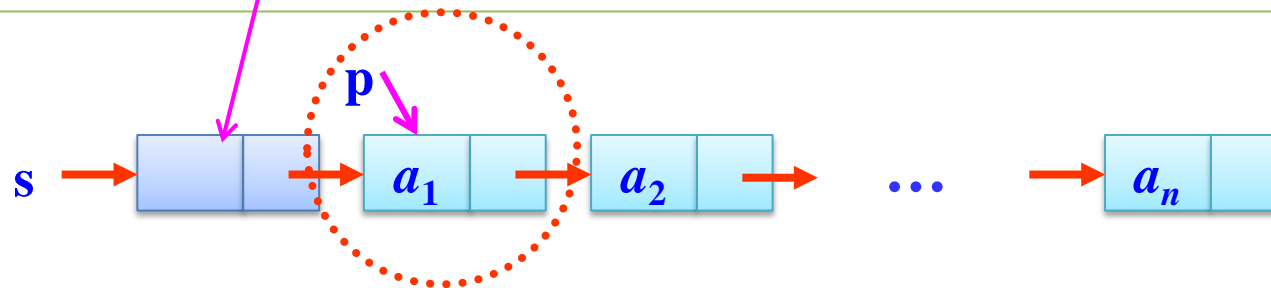
```
int Push(LinkStack s, ElemType e)
{
    LinkStackNode *p;
    p=(LinkStackNode *)malloc(sizeof(LinkStackNode));
    if (p==NULL) return (FALSE);
    p->data=e;           //新建元素e对应的结点*p
    p->next=s->next;     //插入*p结点作为开始结点
    s->next=p;
    return (TRUE);
}
```



## (5) 出栈Pop(s, x)

假设栈不为空，将头结点后继数据结点的数据域赋给x，然后将其删除的算法如下：

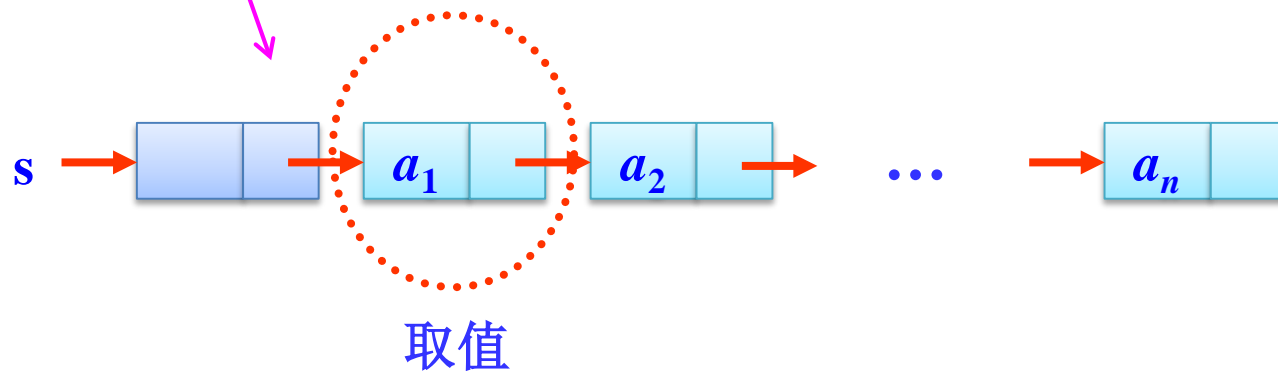
```
int Pop(LinkStack s, ElemType *x)
{
    LinkStackNode *p;
    p=s->next;                //p指向开始结点
    if (p == NULL)             //栈空的情况
        return false;
    x=p->data;
    s->next=p->next;           //删除*p结点
    free(p);                   //释放*p结点
    return true;
}
```



## (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 $e$ 。

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



## 思考题

链栈和顺序栈两种存储结构有什么不同？



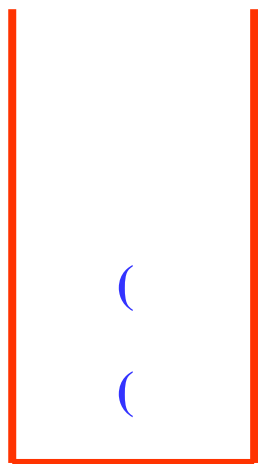
**【例2】** 编写一个算法判断输入的表达式中括号是否配对（假设只含有左、右圆括号）。

### 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

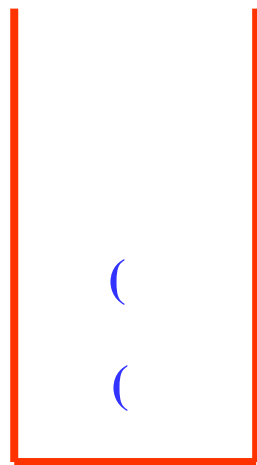
③ 遇到’ )’, 栈顶为’ (’, 退栈

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

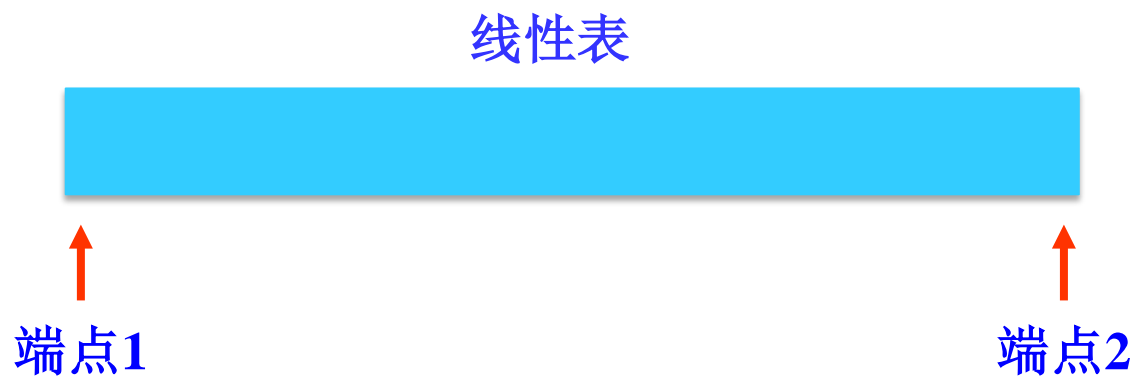
栈空且`exp`扫描完, 返回`true`

<b>int Match(char exp[], int n)</b>	<b>//配对时返回1； 否则返回为0</b>
<b>{ int i=0; char e;</b>	
<b>int match=1;</b>	
<b>LinkStackNode st;</b>	
<b>InitStack(&amp;st);</b>	<b>//初始化栈</b>
<b>while (i&lt;n &amp;&amp; match)</b>	<b>//扫描exp中所有字符</b>
<b>{ if (exp[i]=='(') Push(&amp;st, exp[i]);</b>	<b>//遇到任何左括号都进栈</b>
<b>else if (exp[i]==')')</b>	<b>//当前字符为右括号</b>
<b>{ if (GetTop(&amp;st, &amp;e)==true)</b>	
<b>{ if (e!='(')</b>	<b>//栈顶元素不为'('时表示不匹配</b>
<b>match=0;</b>	
<b>else</b>	
<b>Pop(&amp;st, &amp;e);</b>	<b>//将栈顶元素出栈</b>
<b>}</b>	
<b>else match=0;</b>	<b>//无法取栈顶元素时表示不匹配</b>
<b>}</b>	
<b>i++;</b>	<b>//继续处理其他字符</b>
<b>}</b>	
<b>if (!StackEmpty(&amp;st)) match=0;</b>	
<b>return match;</b>	
<b>}</b>	

## 3.2 队列

### 3.2.1 队列的定义

队列简称队，它也是一种运算受限的线性表。



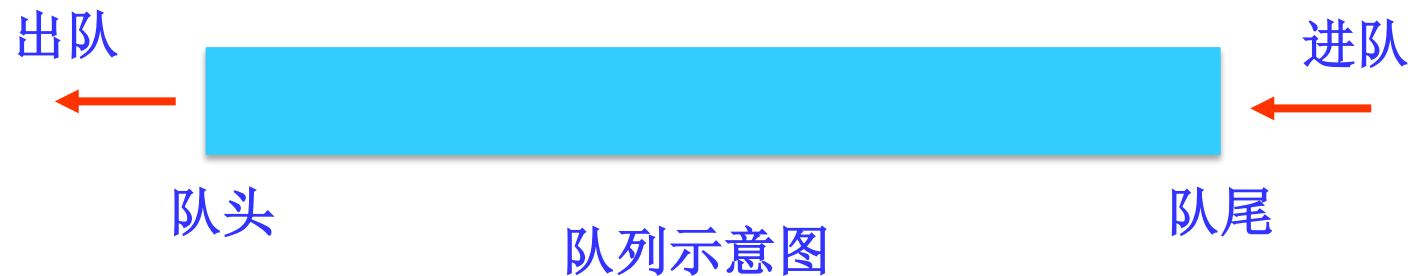
队列只能选取一个端点进行插入操作，另一个端点进行删除操作

把进行插入的一端称做**队尾（rear）**。

进行删除的一端称做**队首或队头（front）**。

向队列中插入新元素称为**进队或入队**，新元素进队后就成为**新的队尾元素**。

从队列中删除元素称为**出队或离队**，元素出队后，其后继元素就成为**队首元素**。



队列的主要特点是先进先出，所以又把队列称为**先进先出表**。

例如：



假如5个人  
过独木桥



只能按上桥的  
次序过桥



这里独木桥就是一个队列

**思考题：**

队列和线性表有什么不同？

队列和栈有什么不同？

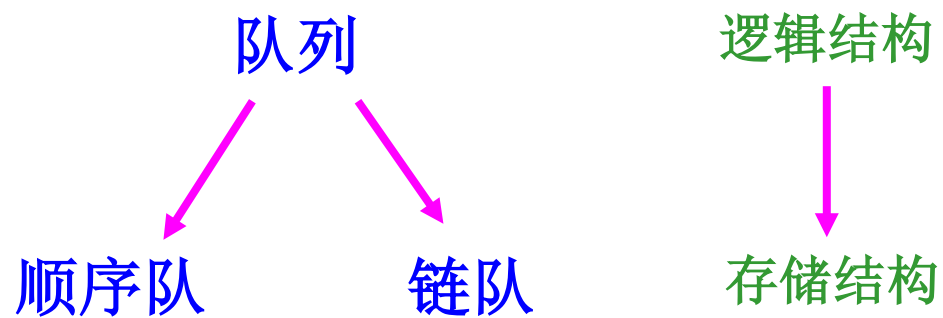
## 队列抽象数据类型=逻辑结构+基本运算（运算描述）

队列的基本运算如下：

- ① **InitQueue(Q)**: 初始化队列。构造一个空队列Q。
- ② **IsEmpty(Q)**: 判断队列是否为空。若队列q为空，则返回真；否则返回假。
- ③ **IsFull(Q)**: 判断队列是否为满。若队列Q为满，则返回真；否则返回假。
- ④ **EnterQueue(Q,x)**: 进队列。将元素x进队作为队尾元素。
- ⑤ **DeleteQueue(Q,x)**: 出队列。从队列Q中出队一个元素，并将其值赋给x。
- ⑥ **GetHead(Q,x)**: 取队头元素（但不出队），有x返回。
- ⑦ **ClearQueue(Q)**: 将队列Q置空。

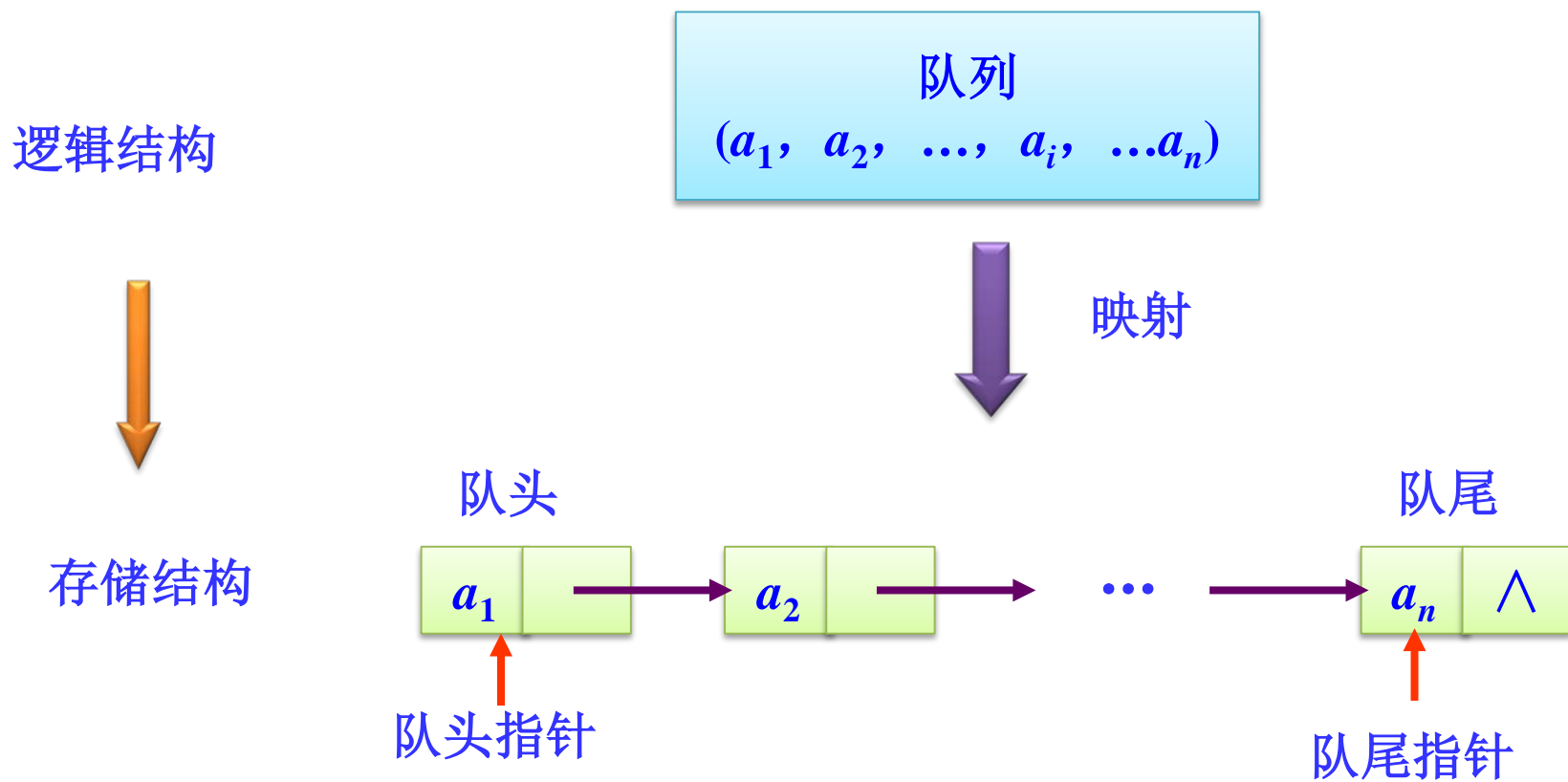


既然队列中元素逻辑关系与线性表的相同，队列可以**采用与线性表相同的存储结构**。



### 3.2.2 队列的链式存储结构及其基本运算的实现

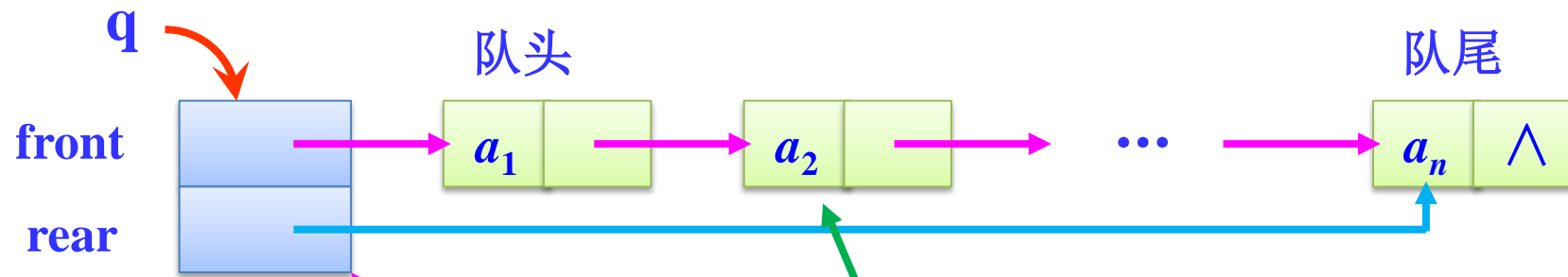
采用链表存储的队列称为链队，这里采用不带头结点的单链表实现。



链队示意图

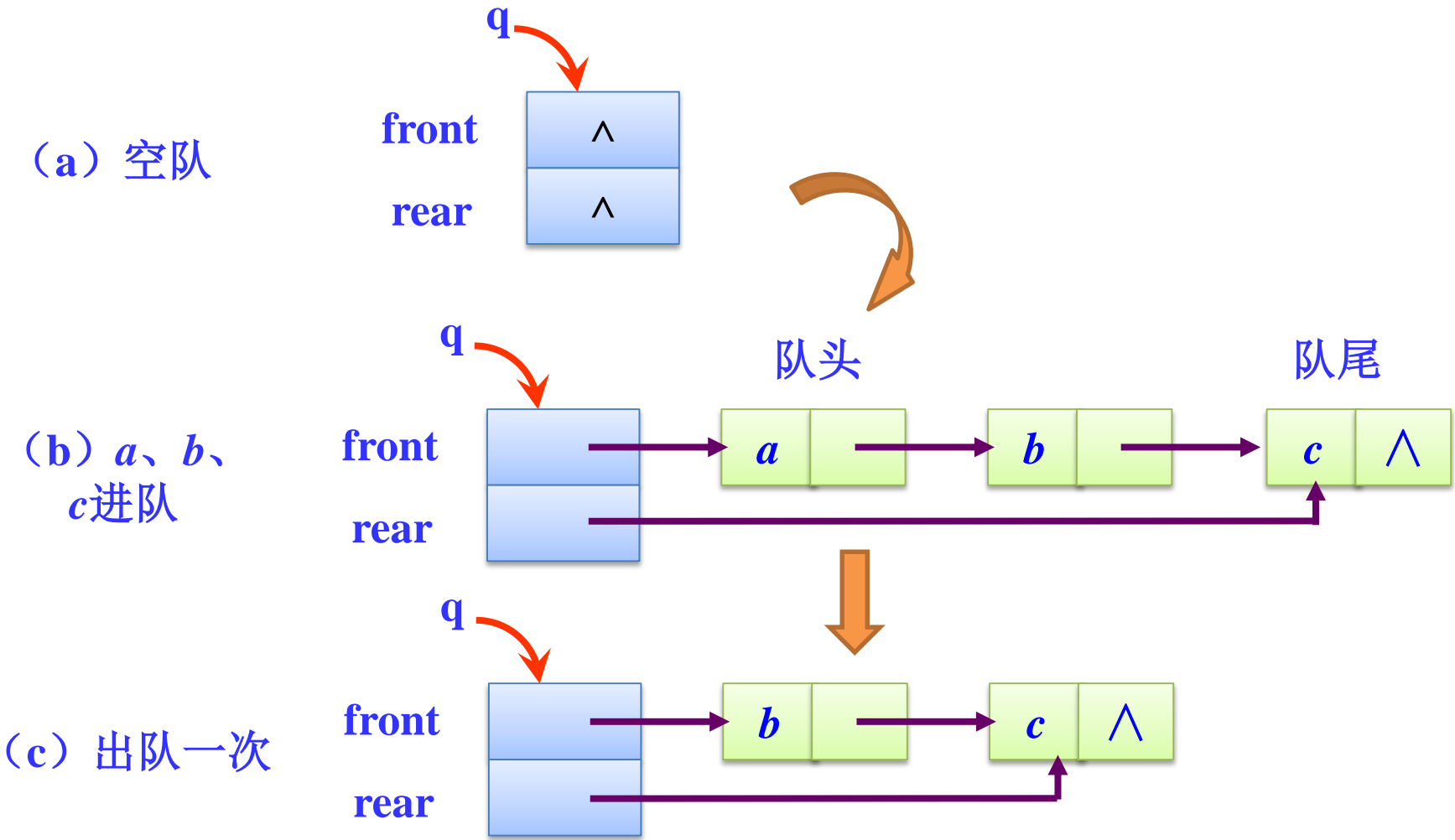
通常将队头和队尾两个指针合起来：

## 其他教材的链表存储结构



链队组成：

- (1) 存储队列元素的单链表结点
- (2) 指向队头和队尾指针的链队头结点



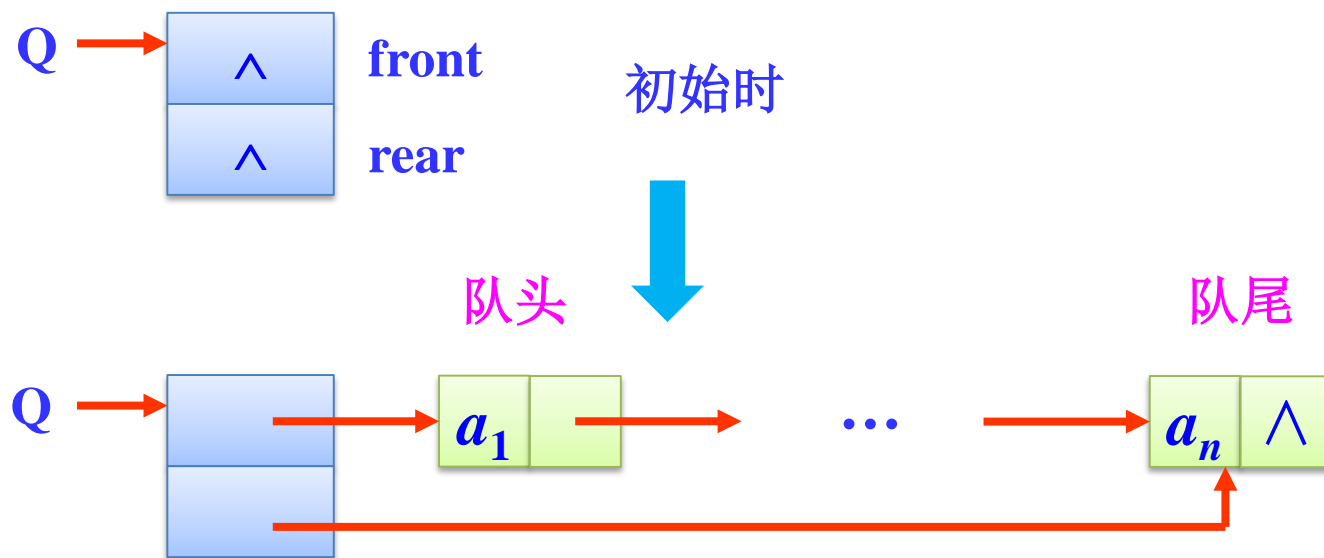
单链表中数据结点类型LinkQueueNode定义如下:

```
typedef struct Node
{
    ElemType data;    //数据元素
    struct Node *next;
} LinkQueueNode;
```

链队中头结点类型LinkQueue定义如下:

```
typedef struct
{
    LinkQueueNode *front;    //指向单链表队头结点
    LinkQueueNode *rear;    //指向单链表队尾结点
} LinkQueue;
```

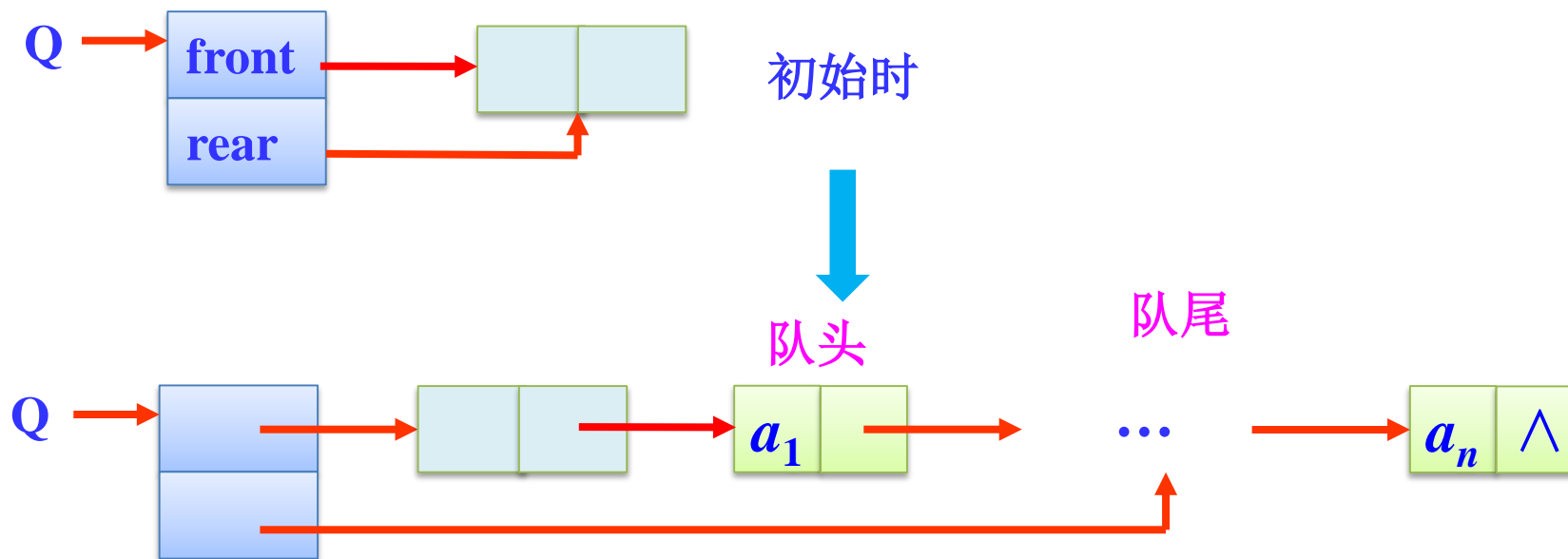
## 其他教材的链表存储结构



链队的4要素:

- ✓ 队空条件: **front=rear=NULL**
- ✓ 队满条件: 不考虑
- ✓ 进队 $e$ 操作: 将包含 $e$ 的结点插入到单链表表尾
- ✓ 出队操作: 删除单链表首数据结点

## 本教材的链表存储结构——增加了空的结点



链队的4要素:

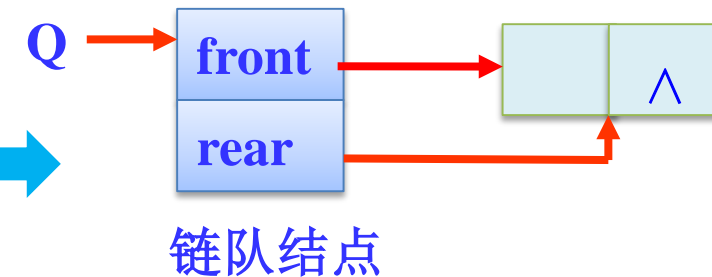
- ✓ 队空条件:  $\text{front}=\text{rear}$
- ✓ 队满条件: 不考虑
- ✓ 进队 $e$ 操作: 将包含 $e$ 的结点插入到单链表表尾
- ✓ 出队操作: 删除单链表首数据结点

在链队存储中，队列的基本运算算法如下。

### (1) 初始化队列InitQueue(q)

构造一个空队列，即只创建一个链队头结点，其front和rear域均置为NULL。

```
int InitQueue(LinkQueue *Q)
{
    Q->front=(LinkQueueNode *)malloc(sizeof(LinkQueueNode));
    if(Q->front != NULL) {
        Q->rear = Q->front;
        Q->front->rearnext=NULL;
        return 1;
    }
    else return 0;
}
```



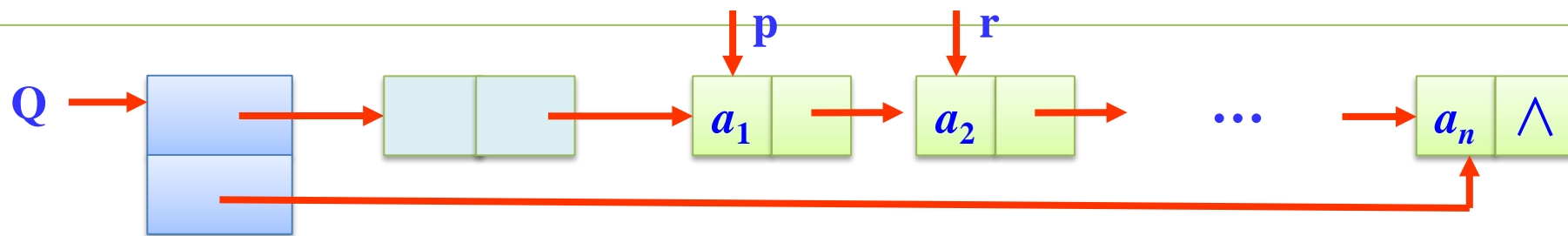


## (2) 清空队列ClearQueue(Q)

释放队列占用的存储空间，包括链队所有数据结点的存储空间。

```
void ClearQueue(LinkQueue *Q)
{
    LinkQueueNode *p=Q->front->next, *r;    //p指向队头数据结点
    if (p!=NULL)                                //释放数据结点占用空间
    {
        r=p->next;
        while (r!=NULL)
        {
            free(p);
            p=r;
            r=p->next;
        }
    }
    free(p);                                    //释放链队最后数据结点占用空间
    Q->rear = Q->front;
}
```

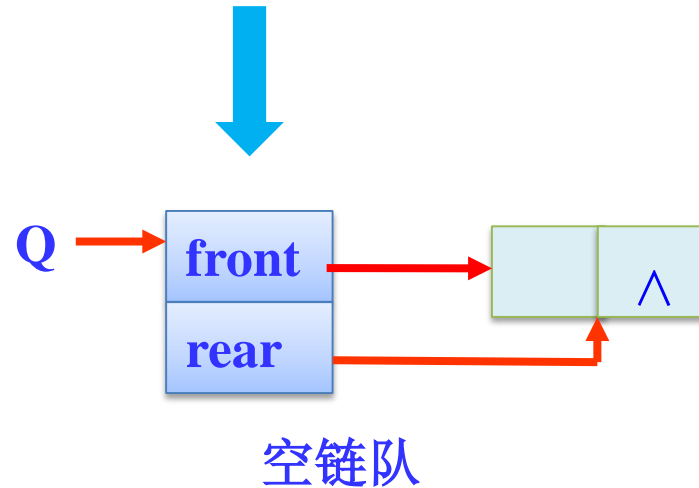
头删法



### (3) 判断队列是否为空IsEmpty(Q)

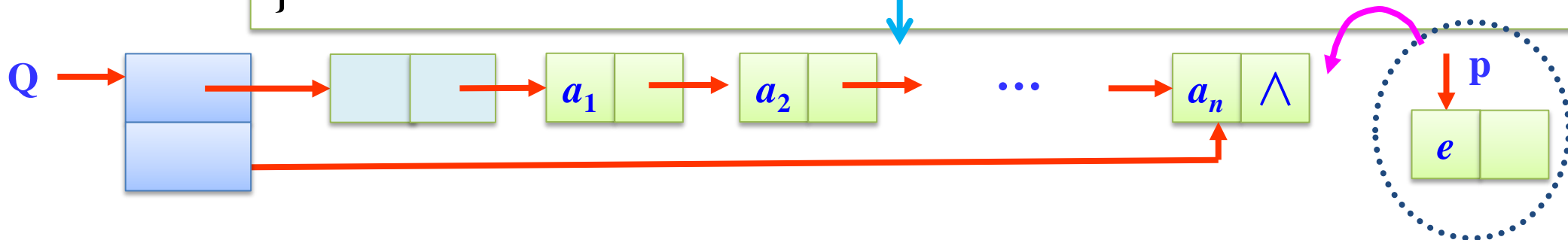
若链队结点的 $Q \rightarrow \text{rear} = Q \rightarrow \text{front}$ ，表示队列为空，返回1；否则返回0。

```
int IsEmpty(LinkQueue *Q)
{
    return(Q->rear==Q->front);
}
```



## (4) 进队enQueue(q, e)

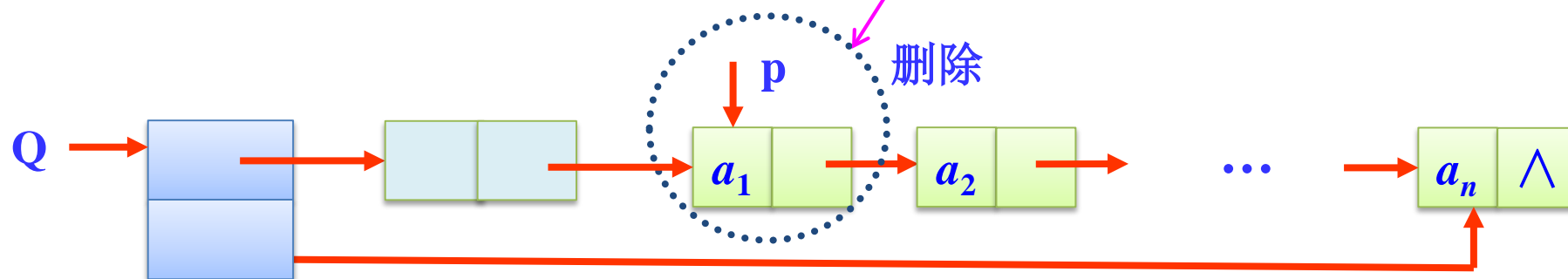
```
int EnterQueue(LinkQueue *Q, QueueElementType x)
{ /* 将数据元素x插入到队列Q中 */
    LinkQueueNode * NewNode;
    NewNode=(LinkQueueNode * )malloc(sizeof(LinkQueueNode));
    if(NewNode!=NULL)
    {
        NewNode->data=x;
        NewNode->next=NULL;
        Q->rear->next=NewNode;
        Q->rear=NewNode;
        return 1;
    }
    else return 0; /* 溢出! */
}
```



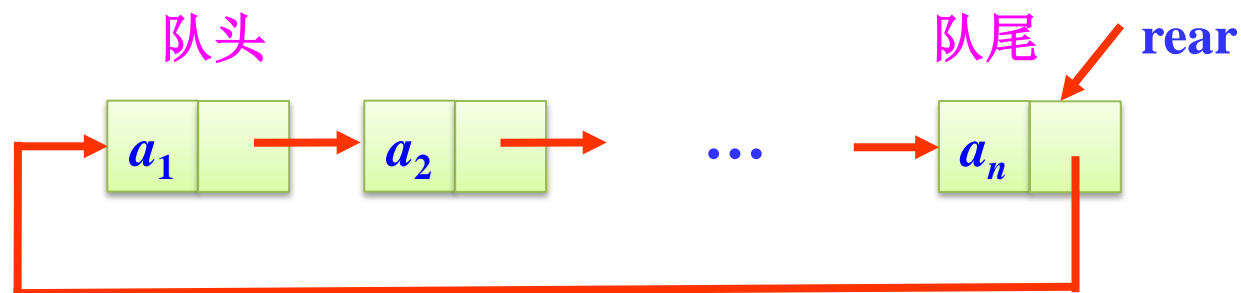
## (5) 出队DeleteQueue(Q, x)

```
int DeleteQueue(LinkQueue * Q, QueueElementType *x)
{ /* 将队列Q的队头元素出队，并存放至x所指的存储空间中 */
    LinkQueueNode * p;
    if(Q->front==Q->rear) return 0;

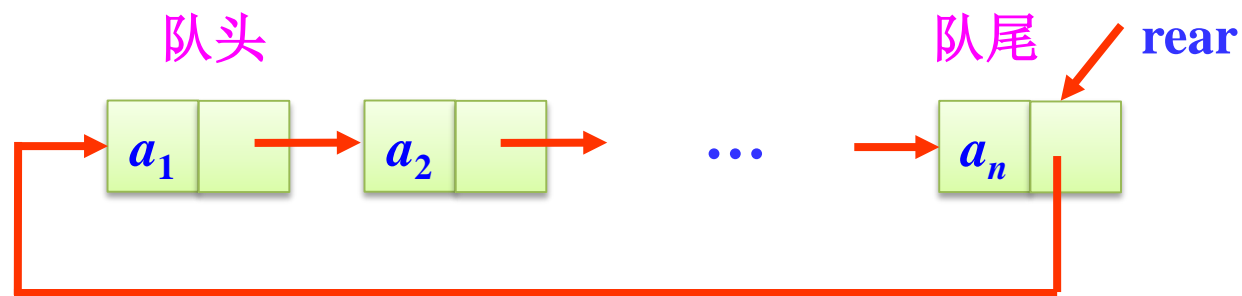
    p=Q->front->next;
    Q->front->next=p->next; /* 队头元素p出队 */
    if(Q->rear==p) /* 如果队中只有一个元素p，则p出队后成为空队 */
        Q->rear=Q->front;
    *x=p->data;
    free(p); /* 释放存储空间 */
    return 1;
}
```



**【例3】** 采用一个不带头结点只有一个尾结点指针`rear`的循环单链表存储队列，设计队列的初始化、进队和出队等算法。



这样的链队通过尾结点指针`rear`唯一标识。



这样的链队通过尾结点指针rear唯一标识。

链队的4要素：

- ✓ 队空条件：rear=NULL
- ✓ 队满条件：不考虑
- ✓ 进队 $e$ 操作：将包含 $e$ 的结点插入到单链表表尾
- ✓ 出队操作：删除单链表首结点

```
void initQueue(LinkList rear) //初始化队运算算法
{
    rear=NULL;
}

bool IsEmpty(LinkList rear) //判队空运算算法
{
    return(rear==NULL);
}
```

```
int EnterQueue(LinkList rear, ElemType x) //进队运算算法
```

```
{ LinkList p;
```

```
  p=(LinkList)malloc(sizeof(Node)); //创建新结点
```

```
  if (!p) return 0;
```

```
  p->data=x;
```

```
  if (rear==NULL)
```

//原链队为空

```
  { p->next=p;
```

//构成循环链表

```
    rear=p;
```

```
  }
```

```
  else
```

```
  { p->next=rear->next;
```

//将\*p结点插入到\*rear结点之后

```
    rear->next=p;
```

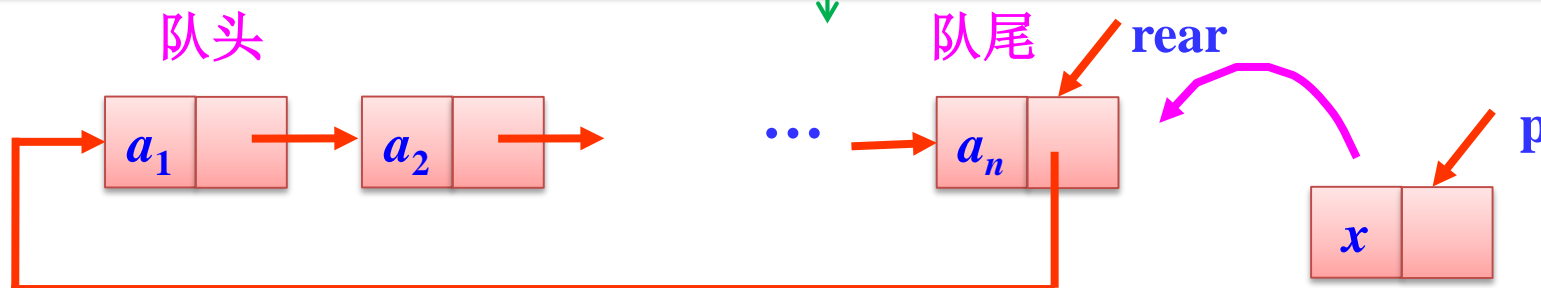
//让rear指向这个新插入的结点

```
    rear=p;
```

```
  }
```

```
  return 1;
```

```
}
```





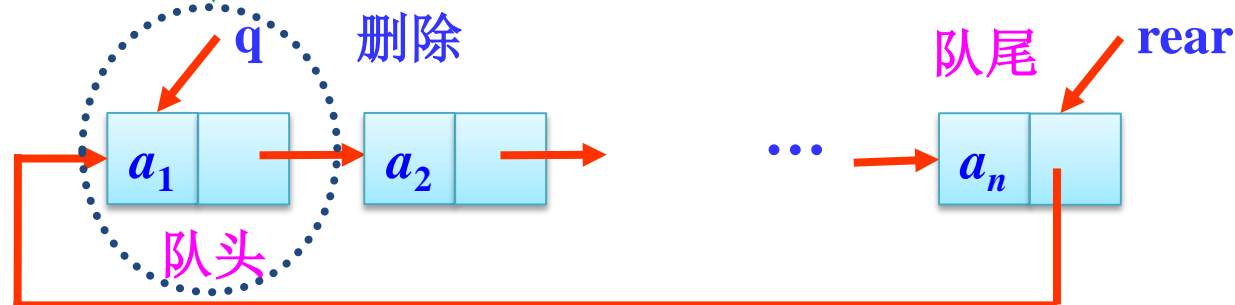
int DeleteQueue(LinkList \*rear, ElemType \*x) //出队运算算法

```
{ LinkList *q;  
  if (rear==NULL) return 0;  
  else if (rear->next==rear)  
  { x=rear->data;  
    free(rear);  
    rear=NULL;  
  }  
  else  
  {  
    q=rear->next;  
    x=q->data;  
    rear->next=q->next;  
    free(q);  
  }  
  return 1;  
}
```

//队空

//原队中只有一个结点

//原队中有两个或以上的结点



### 3.2.3 队列的顺序存储结构及其基本运算的实现

顺序队类型SeqQueue定义如下：

```
typedef struct
{
    ElemType data[StackSize];
    int front,rear;    //队首和队尾指针
} SeqQueue;
```

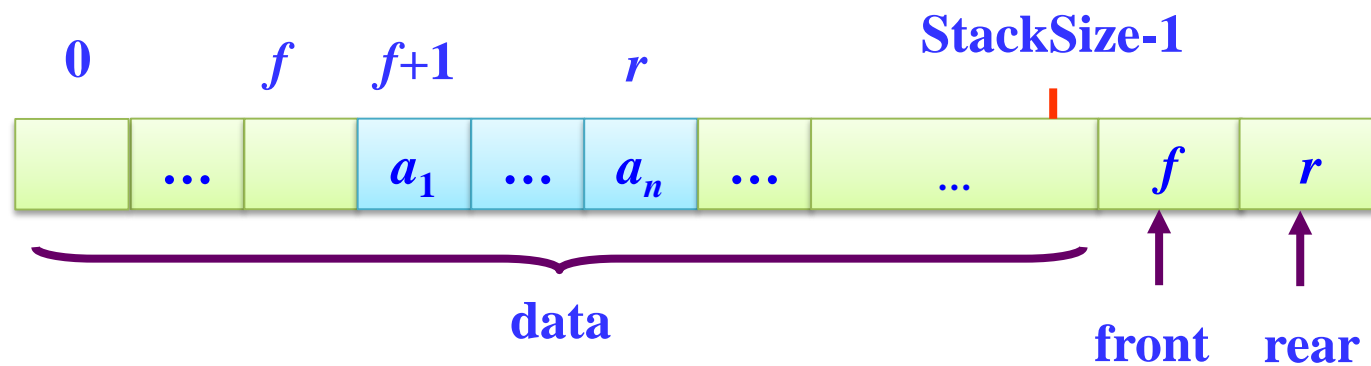
因为队列两端都在变化，所以需要两个指针来标识队列的状态。

逻辑结构



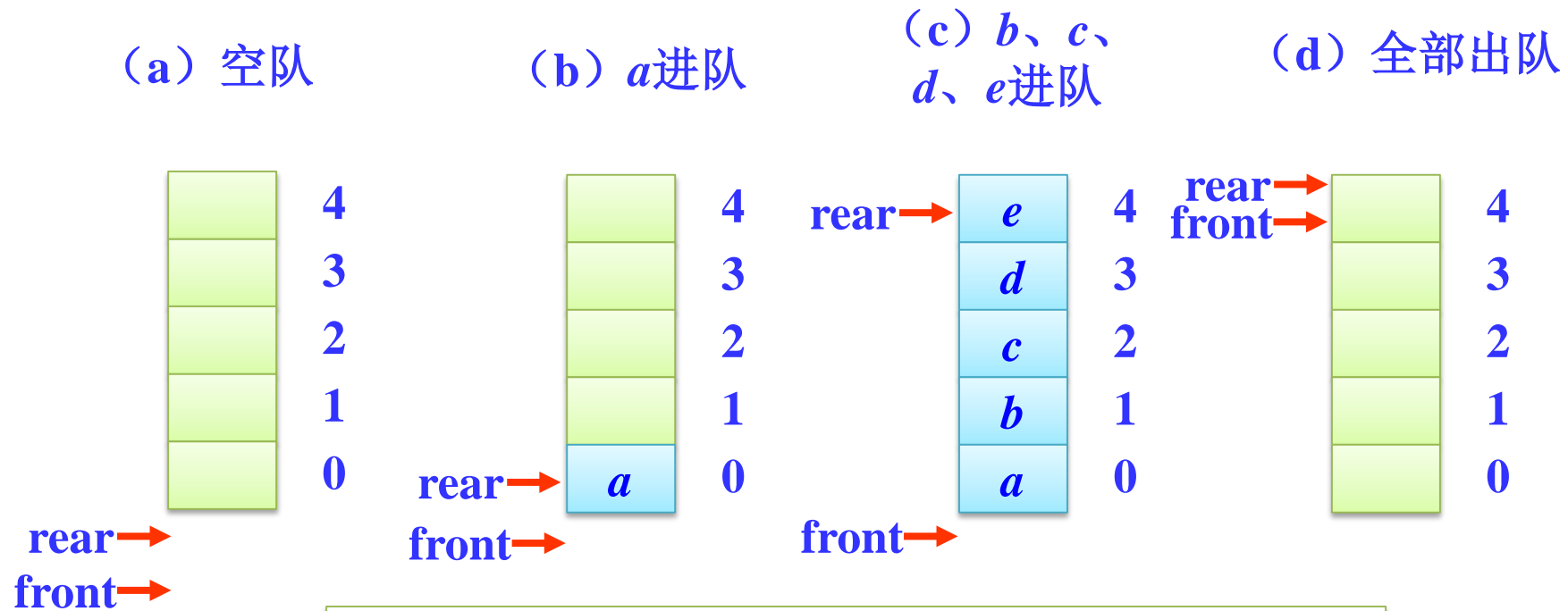
直接映射

存储结构



顺序队的示意图

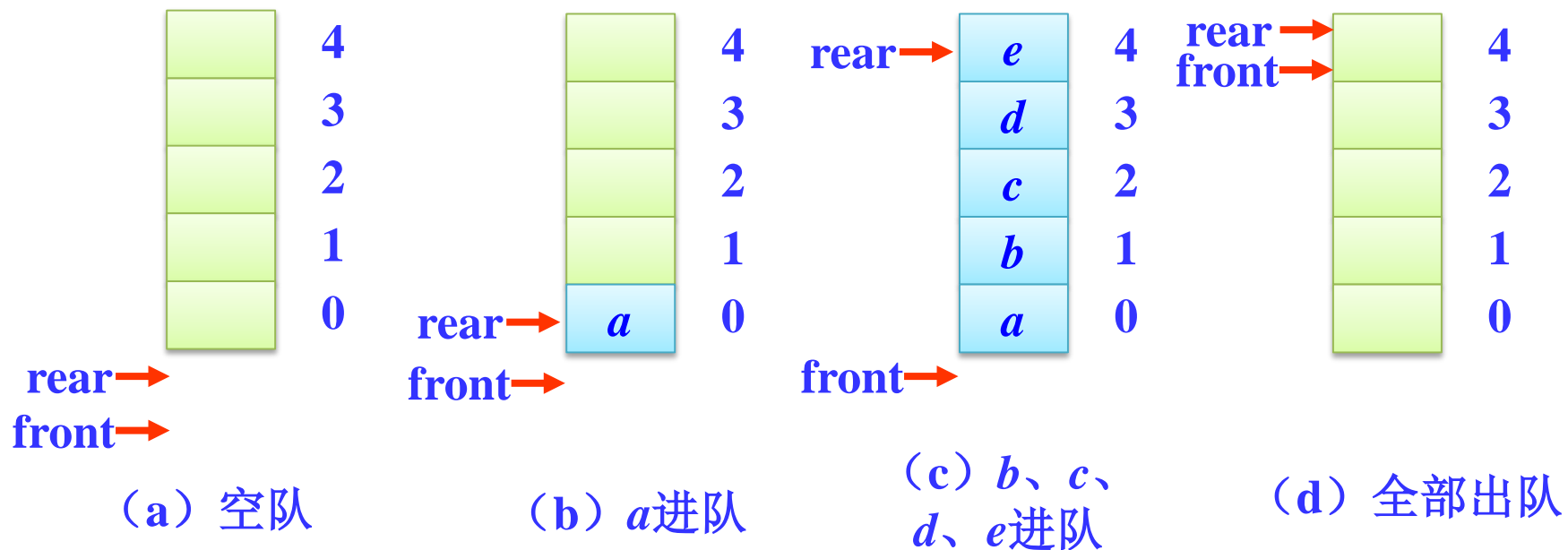
例如: StackSize=5



总结

- ✓ 约定rear总是指向队尾元素
- ✓ 元素进队, rear增1
- ✓ 约定front指向当前队中队头元素的前一位置
- ✓ 元素出队, front增1
- ✓ 当 $\text{rear} = \text{StackSize} - 1$ 时不能再进队

## 队列的各种状态



顺序队的4要素（初始时 $\text{front}=\text{rear}=-1$ ）：

- ✓ 队空条件： $\text{front} = \text{rear}$
- ✓ 队满条件： $\text{rear} = \text{StackSize} - 1$
- ✓ 元素 $e$ 进队： $\text{rear}++$ ;  $\text{data}[\text{rear}]=e$ ;
- ✓ 元素 $e$ 出队： $\text{front}++$ ;  $e=\text{data}[\text{front}]$ ;

→ 头删法

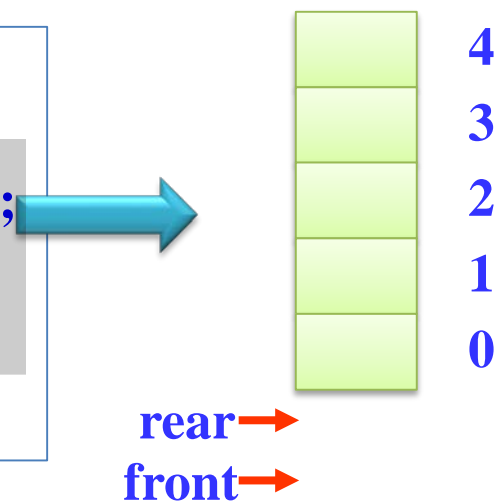
**注意：**  $\text{rear}$ 指向队尾元素； $\text{front}$ 指向队头元素的前一个位置。

## 1、顺序队中实现队列的基本运算

### (1) 初始化队列InitQueue(q)

构造一个空队列q。将front和rear指针均设置成初始状态即-1值。

```
void InitQueue(SeqQueue *q)
{
    q=(SeqQueue *)malloc (sizeof(SeqQueue));
    q->front=q->rear=-1;
}
```



## (2) 销毁队列ClearQueue(q)

释放队列q占用的存储空间。

```
void ClearQueue(SeqQueue *q)
{
    q->front=-1;
    q->rear=-1;
}
```

### (3) 判断队列是否为空 `QueueEmpty(q)`

若队列`q`满足`q->front==q->rear`条件，则返回1；否则返回0。

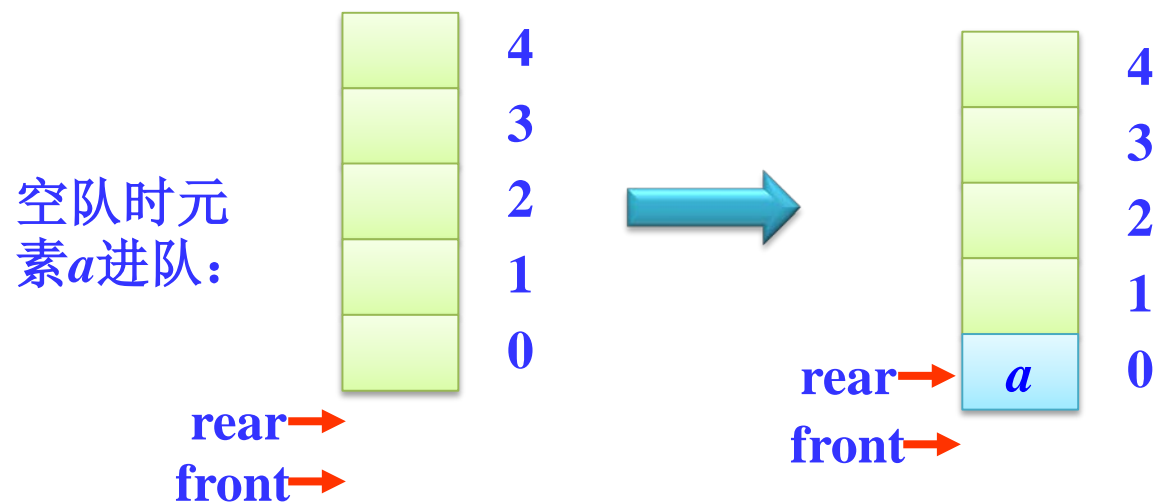
```
int QueueEmpty(SeqQueue *q)
{
    return(q->front==q->rear);
}
```



#### (4) 进队列EnterQueue(q,e)

在队列不满的条件下，先将队尾指针rear循环增1，然后将元素添加到该位置。

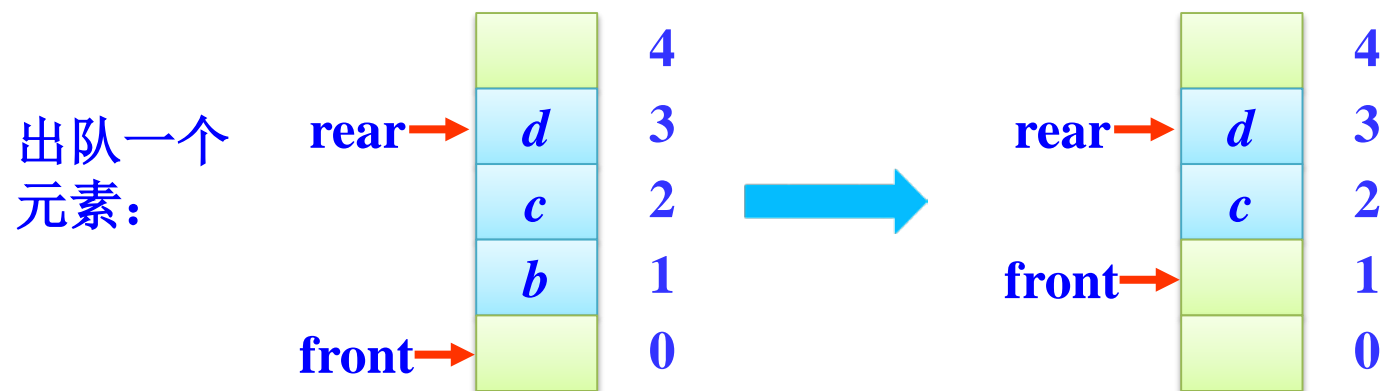
```
int EnterQueue(SeqQueue *q, ElemType e)
{
    if (q->rear == StackSize - 1) //队满上溢出
        return 0;
    q->rear++;
    q->data[q->rear] = e;
    return 1;
}
```



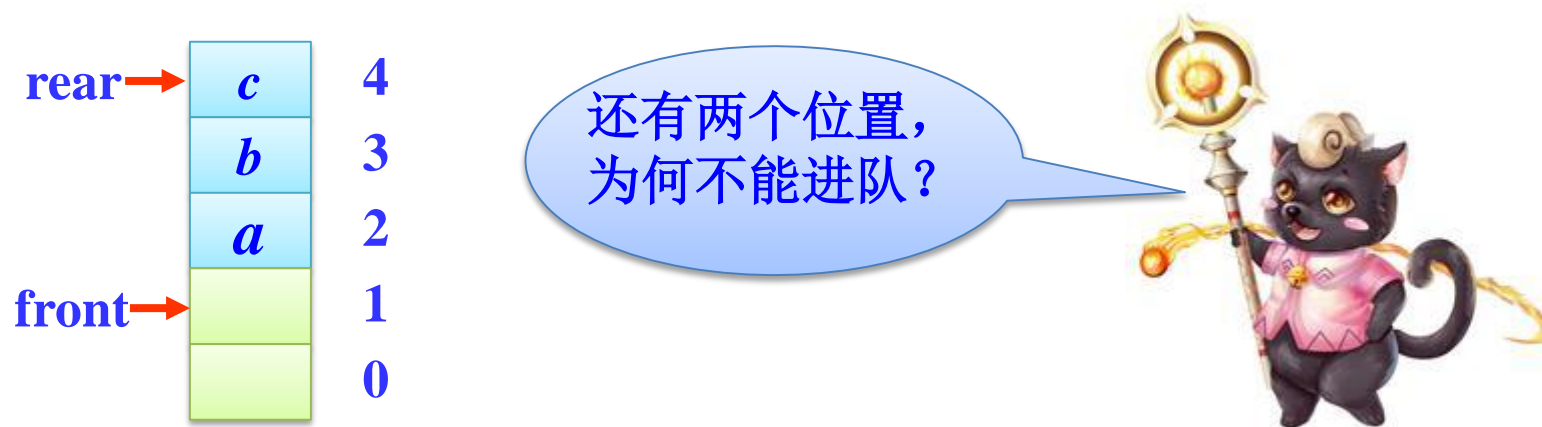
## (5) 出队列DeleteQueue(q,e)

在队列q不为空的条件下，将队首指针front循环增1，并将该位置的元素值赋给e。

```
bool DeleteQueue(SeqQueue *q, ElemType *e)
{   if (q->front == q->rear)    //队空下溢出
    return 0;
    q->front++;
    e = q->data[q->front];
    return 1;
}
```



## 2、环形队列（或循环队列）中实现队列的基本运算



这是因为采用 `rear == StackSize - 1` 作为队满条件的缺陷。当队满条件为真时，队中可能还有若干空位置。

这种溢出并不是真正的溢出，称为假溢出。

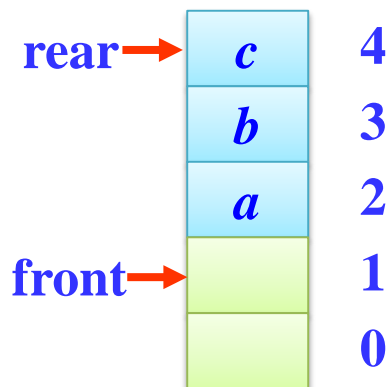
## 解决方案

把数组的前端和后端连接起来，形成一个环形的顺序表，即把存储队列元素的表从逻辑上看成一个环，称为**环形队列或循环队列**。

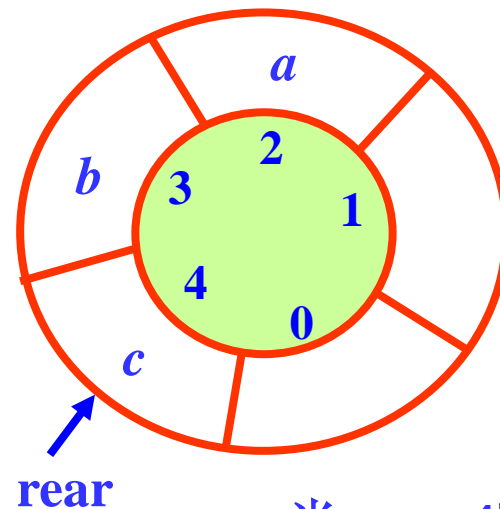
$rear \% stacksize$

$n \% m$

例如



当 $rear=4$ 时，不能再进队



当 $rear=4$ 时，下一步到位置0，可以进队

$0 \% 5 = 0$   
 $1 \% 5 = 1$   
 $2 \% 5 = 2$   
 $\vdots$   
 $4 \% 5 = 4$   
 $5 \% 5 = 0$

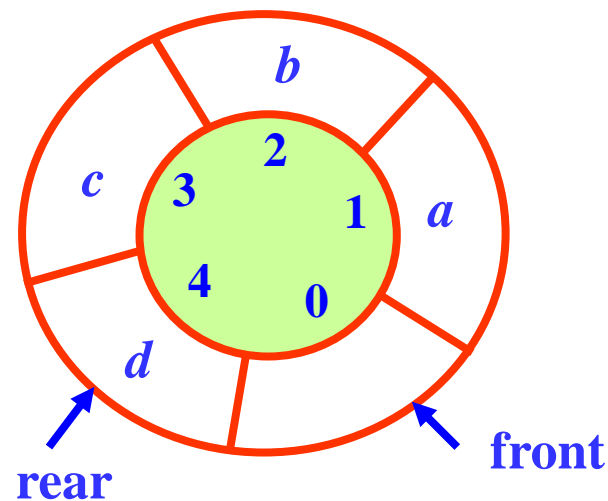
通过取余运算来实现循环

逻辑上为环  
物理上为普通数组

原来如此，简单！



## 环形队列:

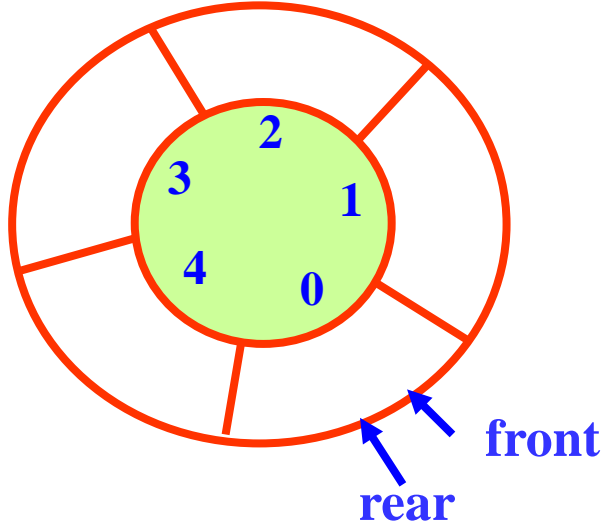


实际上内存地址一定是连续的，不可能是环形的，这里是通过逻辑方式实现环形队列，也就是将 $\text{rear}++$ 和 $\text{front}++$ 改为：

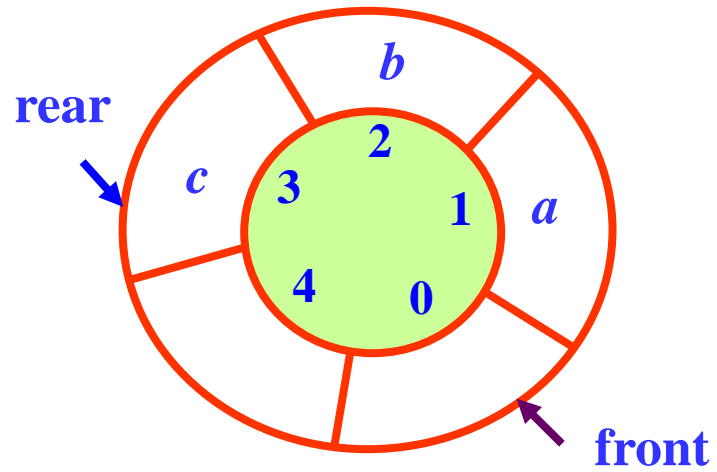
①  $\text{rear} = (\text{rear} + 1) \% \text{StackSize}$

②  $\text{front} = (\text{front} + 1) \% \text{StackSize}$

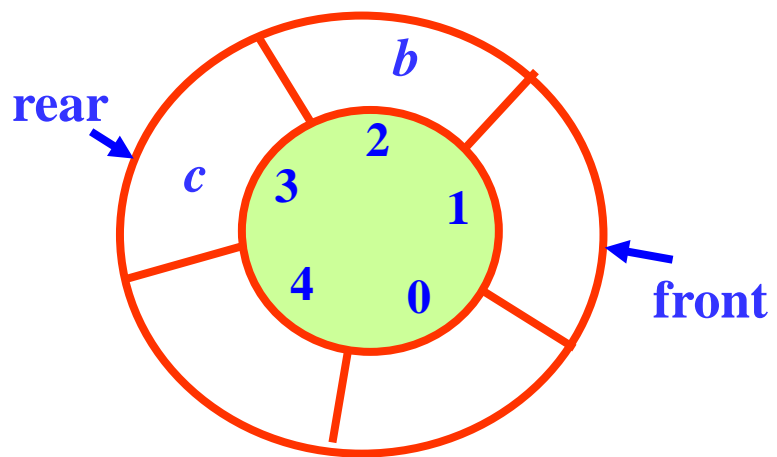
到达边界又回来



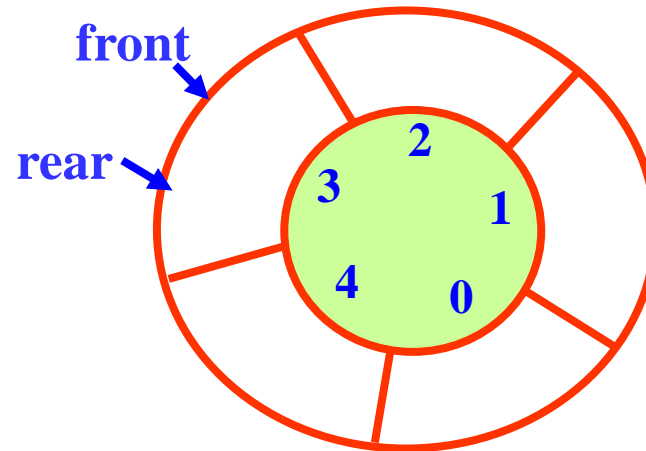
(a) 空队



(b)  $a$ 、 $b$ 、 $c$ 进队

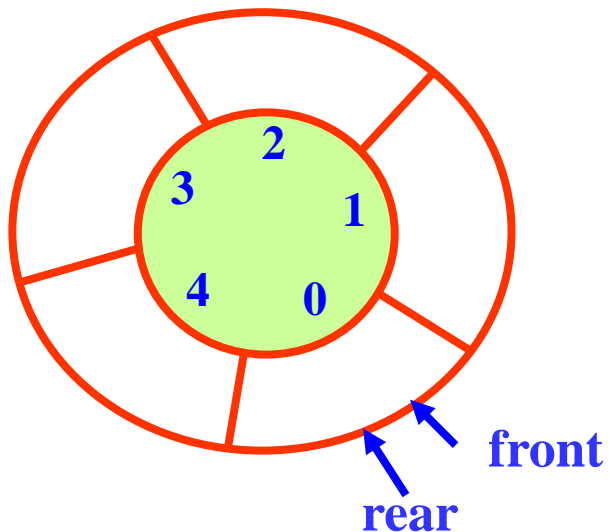


(c) 出队一次

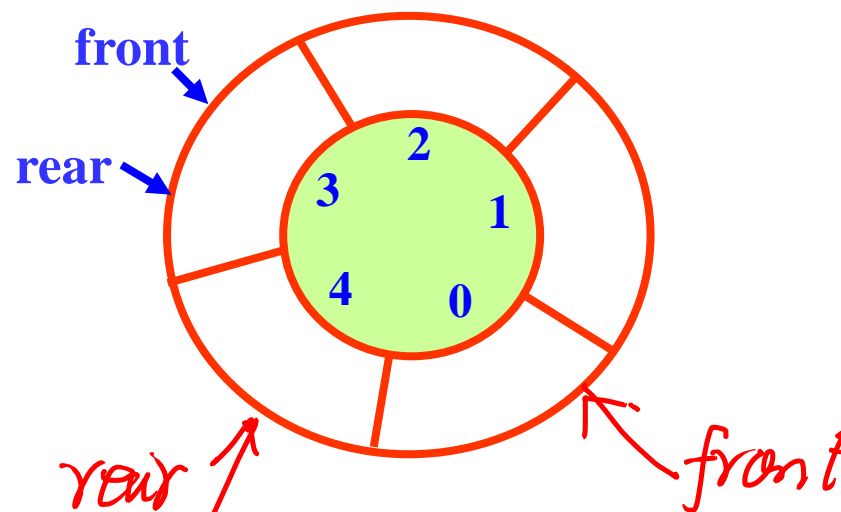


(d) 出队2次

现在约定 $\text{rear}=\text{front}$ 为队空，以下两种情况都满足该条件：



初始状态



进队的所有元素均出队

队满

那么如何设置队满的条件呢？



让 $\text{rear}=\text{front}$ 为队空条件，并约定

$(\text{rear}+1)\% \text{StackSize}=\text{front}$

为队满条件。

理解：

$\text{rear}$ 追上 $\text{front}$ 为满  
 $\text{front}$ 追上 $\text{rear}$ 为空



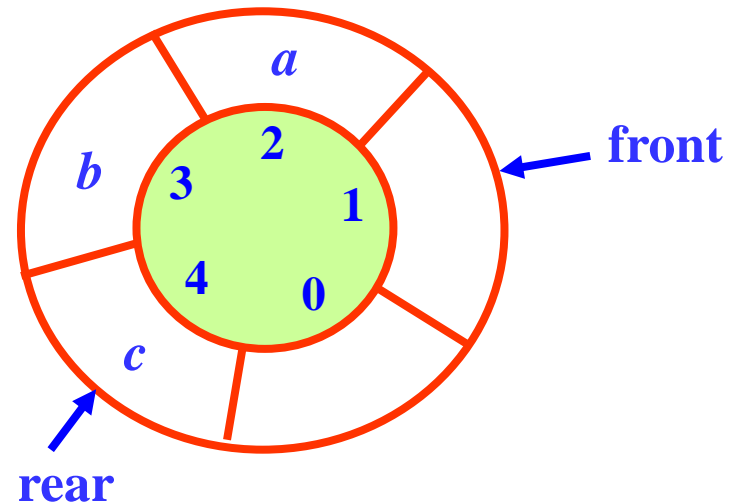
进队一个元素时  
到达队头，就认  
为队满了。这样  
做会少放一个元  
素，牺牲一个元  
素没关系的



## 环形队列的4要素:

- ✓ 队空条件:  $\text{front} = \text{rear}$
- ✓ 队满条件:  $(\text{rear} + 1) \% \text{StackSize} = \text{front}$
- ✓ 进队 $e$ 操作:  $\text{rear} = (\text{rear} + 1) \% \text{StackSize}$ ; 将 $e$ 放在 $\text{rear}$ 处
- ✓ 出队操作:  $\text{front} = (\text{front} + 1) \% \text{StackSize}$ ; 取出 $\text{front}$ 处元素 $e$ ;

在环形队列中，实现队列的基本运算算法与非环形队列类似，只是改为上述4要素即可。



# 初始化操作

```
void InitQueue (SeqQueue * Q)
{   /* 将*Q初始化为一个空的循环队列 */
    Q->front=Q->rear=0;
}
```

## 入队操作

```
int EnterQueue(SeqQueue *Q, QueueElementType x)
{   /*将元素x入队*/

    if((Q->rear+1)%MAXSIZE==Q->front) return 0; /*队列已经满了*/

    Q->data[Q->rear]=x;

    Q->rear=(Q->rear+1)%MAXSIZE; /* 重新设置队尾指针 */

    return 1; /*操作成功*/

}
```

## 出队操作

```
int DeleteQueue(SeqQueue *Q, QueueElementType * x)
{ /*删除队列的队头元素，用x返回其值*/

    if (Q->front==Q->rear) return 0; /*队列为空*/

    *x=Q->element[Q->front];

    Q->front=(Q->front+1)%MAXSIZE; /*重新设置队头指针*/

    return 1; /*操作成功*/

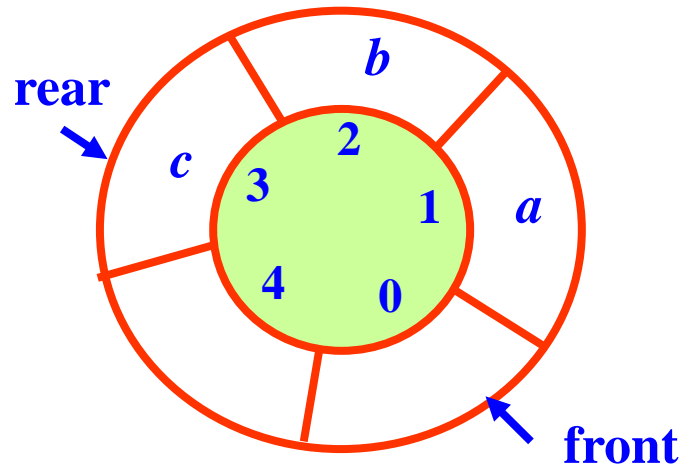
}
```

**【例4】**对于环形队列来说，如果知道队头指针和队列中元素个数，则可以计算出队尾指针。也就是说，可以用队列中元素个数代替队尾指针。

设计出这种环形队列的初始化、入队、出队和判空算法。

已知front、rear，求队中元素个数count = ?

StackSize=5



$$\text{count} = (3 - 0) = 3$$

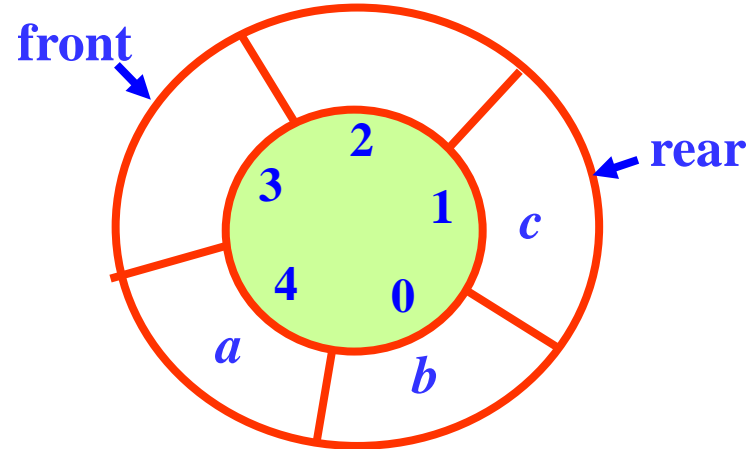


$$\text{count} = \text{rear} - \text{front} \quad ?$$

$$\text{count} = (3 - 0 + \text{StackSize}) = 8 \quad \times$$



$$\text{count} = (3 - 0 + \text{StackSize}) \% \text{StackSize} = 3 \quad \checkmark$$



$$\text{count} = (1 - 3) = -2 \quad \times$$



$$\text{count} = (1 - 3 + \text{StackSize}) = 3$$



$$\text{count} = (1 - 3 + \text{StackSize}) \% \text{StackSize} = 3 \quad \checkmark$$



已知front、rear，求队中元素个数count:

$$\text{count} = (\text{rear} - \text{front} + \text{StackSize}) \% \text{StackSize}$$

已知front、count，求rear:

$$\text{rear} = (\text{front} + \text{count}) \% \text{StackSize}$$

已知rear、count，求front:

$$\text{front} = (\text{rear} - \text{count} + \text{StackSize}) \% \text{StackSize}$$

**解：**依题意设计的环形队列类型如下：

```
typedef struct
{
    ElemType data[StackSize];
    int front;           //队头指针
    int count;           //队列中元素个数
} QuType;
```

该环形队列的4要素：

- ✓ 队空条件：  $\text{count} = 0$
- ✓ 队满条件：  $\text{count} = \text{StackSize}$  由front和count求出
- ✓ 进队 $e$ 操作：  $\text{rear} = (\text{rear} + 1) \% \text{StackSize}$ ; 将 $e$ 放在rear处
- ✓ 出队操作：  $\text{front} = (\text{front} + 1) \% \text{StackSize}$ ; 取出front处元素 $e$ ;


**注意：**这样的环形队列中最多可放置StackSize个元素。



对应的算法如下：

```
void InitQueue(QuType *qu)    //初始化队运算算法
{
    qu=(QuType *)malloc(sizeof(QuType));
    qu->front=0;
    qu->count=0;
}
```

它是一个局部变量，队列qu中不保存该值



```
int EnQueue(QuType *qu, ElemType x) //进队运算算法
{
    int rear; //临时队尾指针
    if (qu->count == StackSize) //队满上溢出
        return 0;
    else
    {
        rear = (qu->front + qu->count) % StackSize; //求队尾位置
        rear = (rear + 1) % StackSize; //队尾循环增1
        qu->data[rear] = x;
        qu->count++; //元素个数增1
        return 1;
    }
}
```

```

int DeleteQueue(QuType *qu, ElemType *x)    //出队运算算法
{
    if (qu->count==0)                        //队空下溢出
        return 0;
    else
    {
        qu->front=(qu->front+1)%StackSize;    //队头循环增1
        *x=qu->data[qu->front];
        qu->count--;                          //元素个数减1
        return 1;
    }
}

```

```

int QueueEmpty(QuType *qu)    //判队空运算算法
{
    return(qu->count==0);
}

```

## 注意：

- ✓ 显然环形队列比非环形队列更有效利用内存空间，即环形队列会重复使用已经出队元素的空间。不会出现假溢出。
- ✓ 但如果算法中需要**使用所有进队的元素来进一步求解**，此时可以使用非环形队列。

## 思考题

链队和顺序队两种存储结构有什么不同？

本章结束