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

队列的几个概念

- 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。

队列示意图

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

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：

❶ InitQueue(&q)：初始化队列。构造一个空队列q。

❷ DestroyQueue(&q)：销毁队列。释放队列q占用的存储空间。

❸ QueueEmpty(q)：判断队列是否为空。若队列q为空，则返回真；否则返回假。

❹ enQueue(&q,e)：进队列。将元素e进队作为队尾元素。

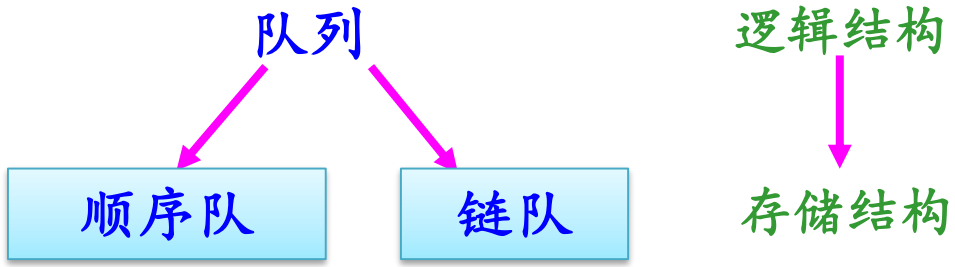
❺ deQueue(&q,&e)：出队列。从队列q中出队一个元素，并将其值赋给e。

思考题：

队列和线性表有什么不同？

队列和栈有什么不同？

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 队列的顺序存储结构及其基本运算的实现

顺序队类型SqQueue定义如下：

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

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。

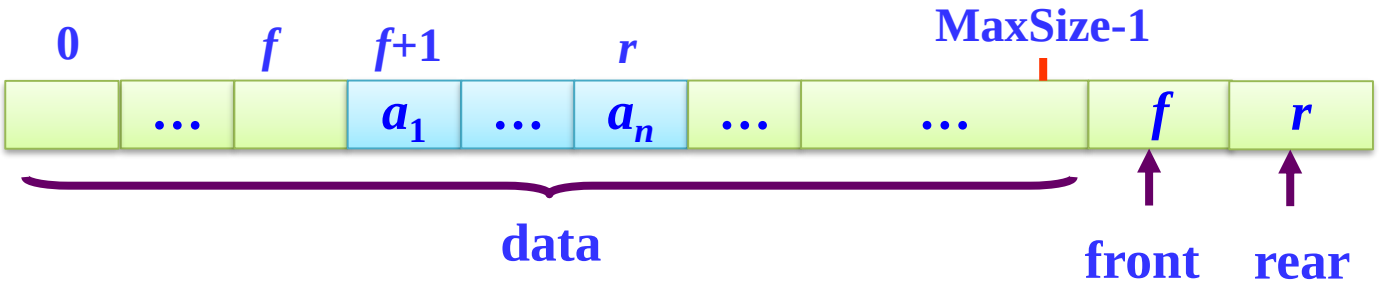
逻辑结构

队列

$(a_1, a_2, \ldots, a_i, \ldots, a_n)$

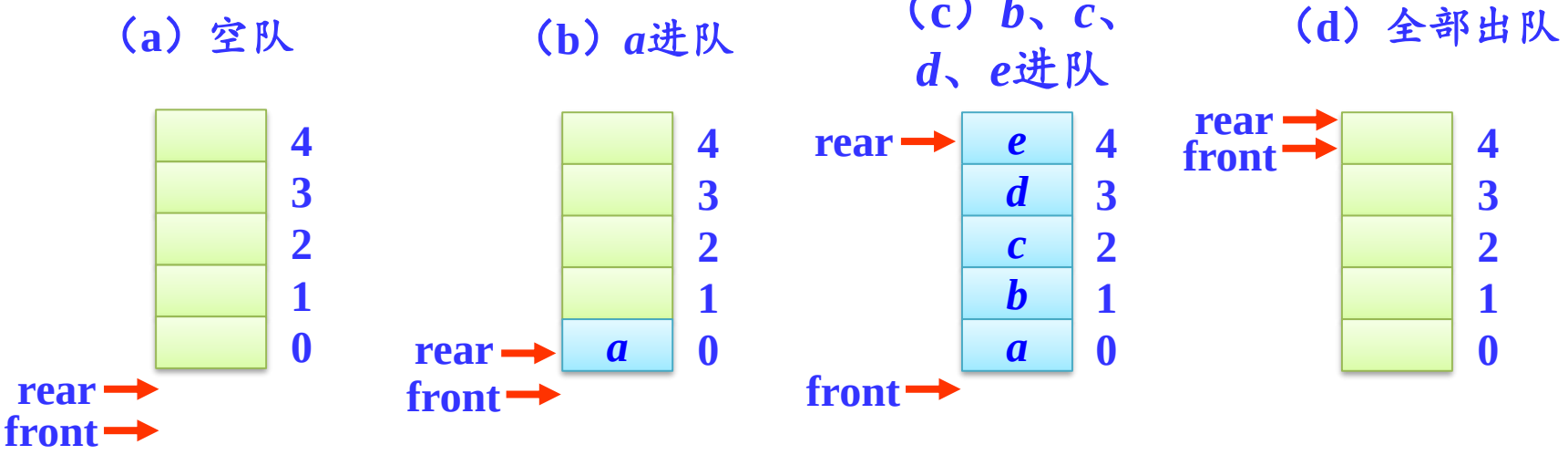
映射

存储结构



顺序队的示意图

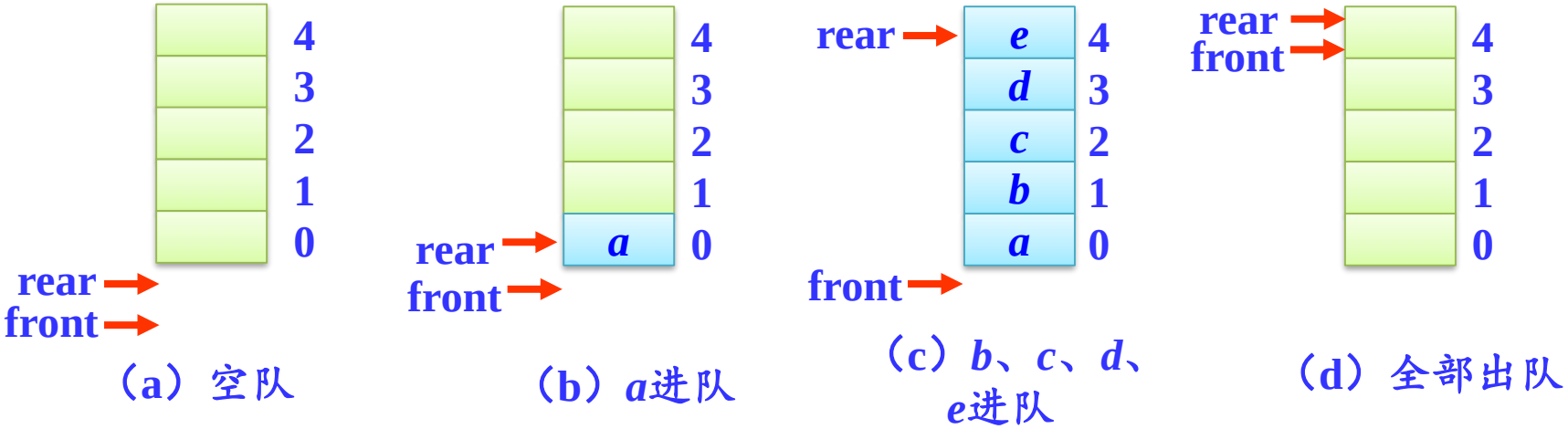
例如：MaxSize=5



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rear=MaxSize−1时不能再进队

队列的各种状态



（a）空队　（b）*a*进队　（c）*b*、*c*、*d*、*e*进队　（d）全部出队

顺序队的4要素（初始时front=rear=−1）：

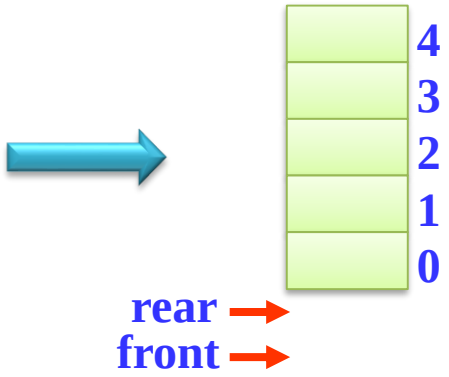
- 队空条件：front = rear
- 队满条件：rear = MaxSize－1
- 元素*e*进队：rear++; data[rear]=*e*;
- 元素*e*出队：front++; *e*=data[front];

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(SqQueue *&q)
{     q=(SqQueue *)malloc (sizeof(SqQueue));
      q->front=q->rear=-1;
}
```



（2）销毁队列DestroyQueue(q)

释放队列q占用的存储空间。

```
void DestroyQueue(SqQueue *&q)
{
    free(q);
}
```

（3）判断队列是否为空QueueEmpty(q)

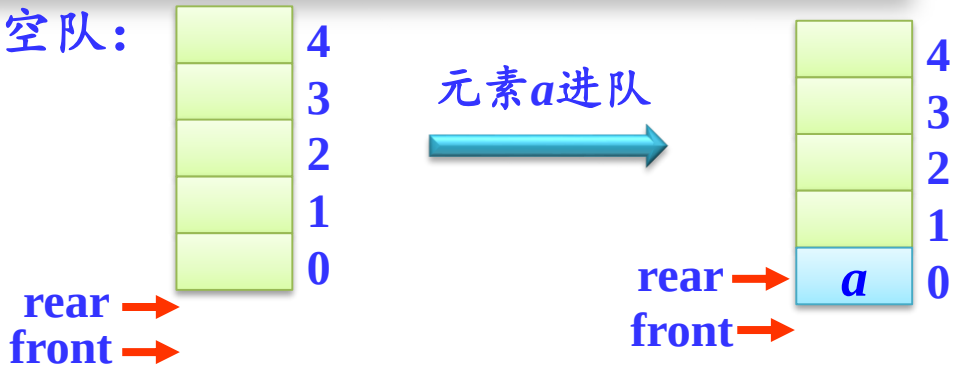
若队列q满足q->front==q->rear条件，则返回true；否则返回false。

```
bool QueueEmpty(SqQueue *q)
{
    return(q->front==q->rear);
}
```

（4）进队列enQueue(q,e)

若队列不满，将队尾指针rear循环增1，然后将元素添加到该位置。

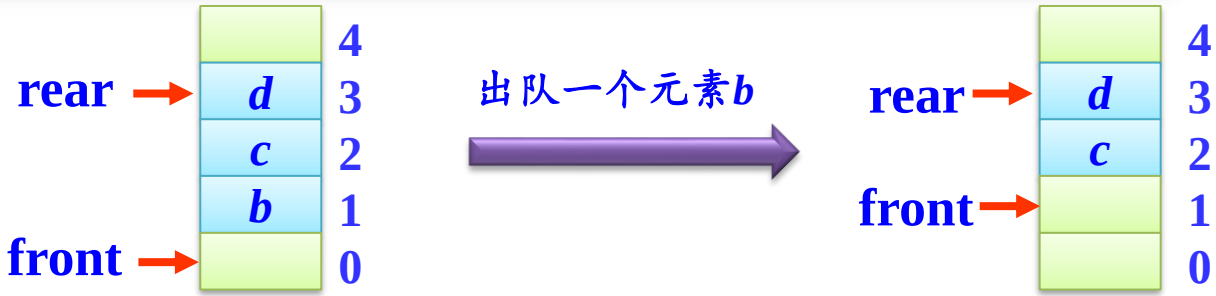
```
bool enQueue(SqQueue *&q,ElemType e)
{       if (q->rear==MaxSize-1)          //队满上溢出
            return false;
        q->rear++;
        q->data[q->rear]=e;
        return true;
}
```



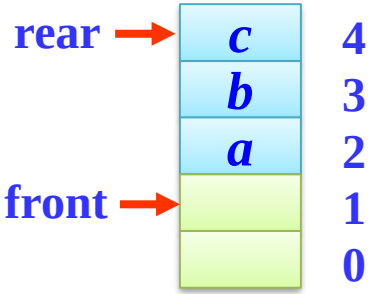
（5）出队列deQueue(q,e)

若队列q不空，将队首指针front循环增1，并将该位置的元素值赋给e。

```
bool deQueue(SqQueue *&q,ElemType &e)
{     if (q->front==q->rear)        //队空下溢出
            return false;
      q->front++;
      e=q->data[q->front];
      return true;
}
```



2、环形队列（或循环队列）中实现队列的基本运算



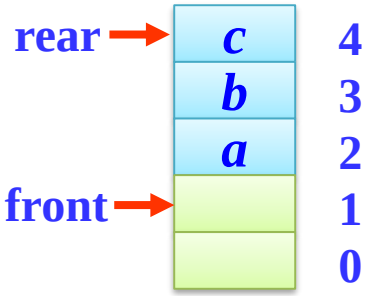
这是因为采用rear==MaxSize−1作为队满条件的缺陷。当队满条件为真时，队中可能还有若干空位置。

这种溢出并不是真正的溢出，称为**假溢出**。

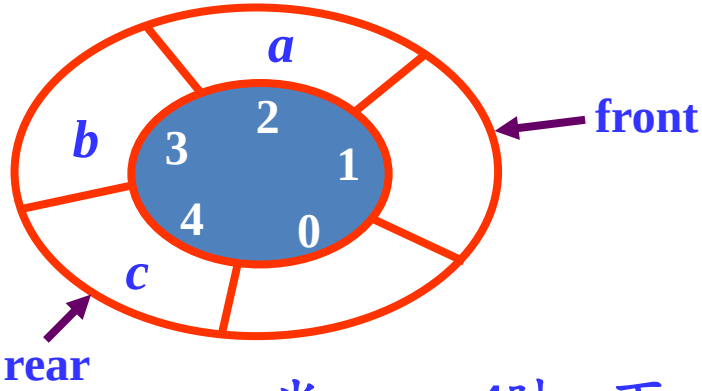
解决方案

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**。

例如

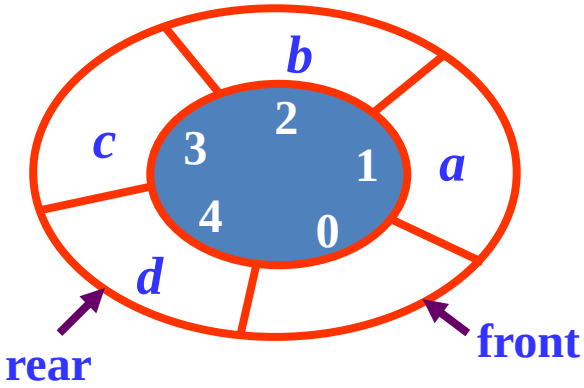


当rear = 4时，不能再进队



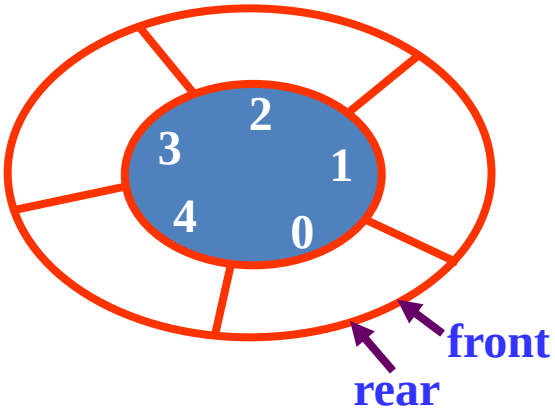
当rear = 4时，下一步到位置0，可以进队

环形队列：

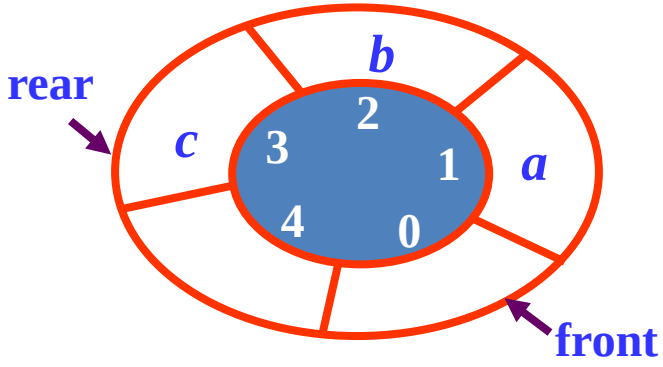


实际上内存地址一定是连续的，不可能是环形的，这里是通过逻辑方式实现环形队列，也就是将rear++和front++改为：

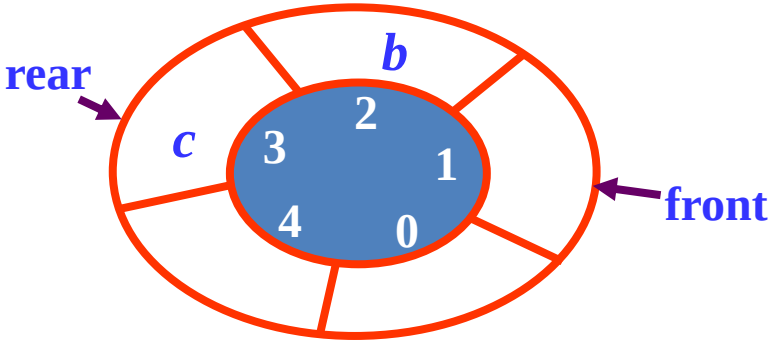
- rear=(rear+1)%MaxSize
- front=(front+1)%MaxSize



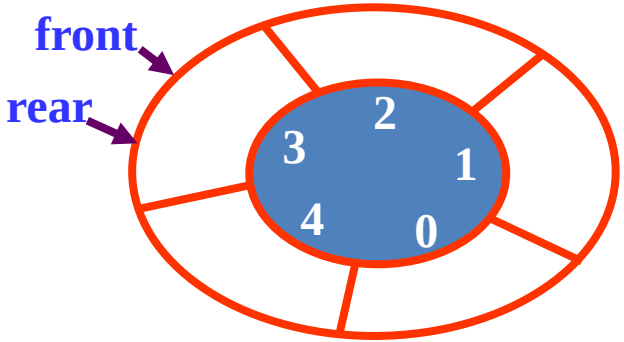
（a）空队



（b）*a*、*b*、*c*进队

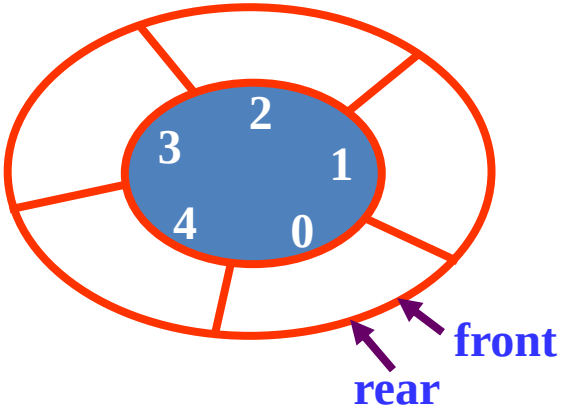


（c）出队一次

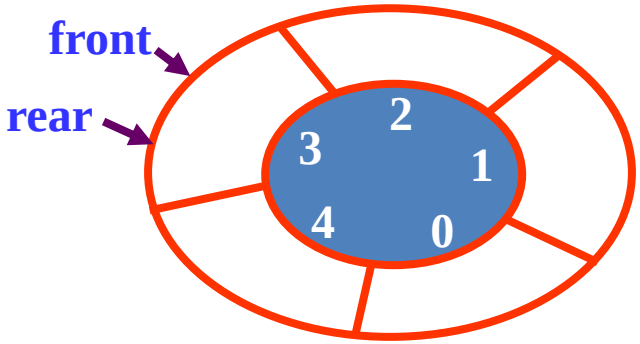


（d）出队2次

现在约定rear=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

那么如何设置队满的条件呢？

让rear=front为队空条件，并约定

(rear+1)%MaxSize=front

为队满条件。

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：front = rear
- 队满条件：(rear+1)%MaxSize = front
- 进队*e*操作：rear=(rear+1)%MaxSize; 将*e*放在rear处
- 出队操作：front=(front+1)%MaxSize; 取出front处元素*e*;

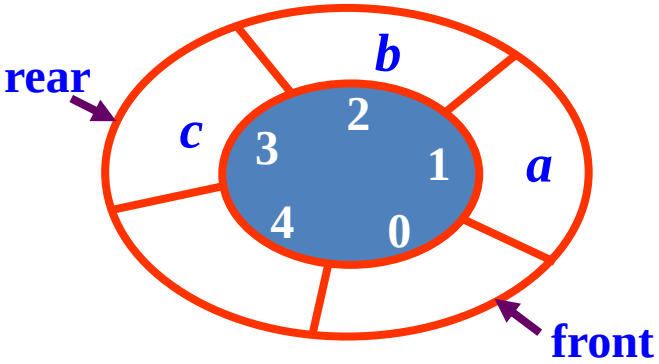
在环形队列中，实现队列的基本运算算法与非环形队列类似，只是改为上述4要素即可。

【例3-5】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 = ？

MaxSize＝5

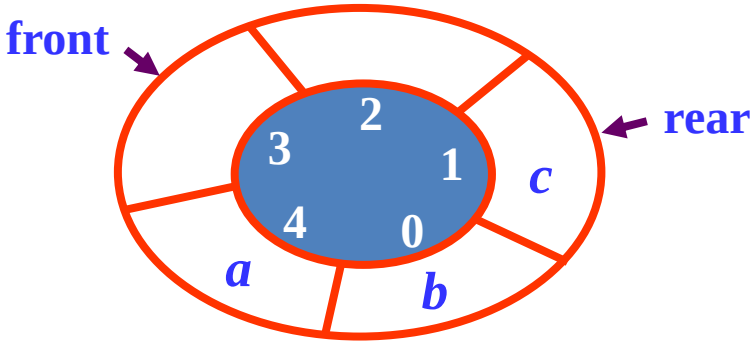


count=(3−0)=3

count=rear−front ？

count=(3−0+MaxSize)=8 ×

count=(3−0+MaxSize)%MaxSize=3 √



count=(1−3)=−2 ×

count=(1−3+MaxSize)=3

count=(1−3+MaxSize)%MaxSize=3 √

已知front、rear，求队中元素个数count：

count=(rear−front+MaxSize)%MaxSize

已知front、count，求rear：

rear=(front+count)%MaxSize

已知rear、count，求front：

front=(rear−count+MaxSize)%MaxSize

解： 依题意设计的环形队列类型如下：

```
typedef struct
{     ElemType data[MaxSize];
      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：count＝0
- 队满条件：count＝MaxSize
- 进队*e*操作：rear=(rear+1)%MaxSize；将*e*放在rear处
- 出队操作：front=(front+1)%MaxSize；取出front处元素*e*；

注意： 这样的环形队列中最多可放置MaxSize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;
}
```

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

它是一个局部变量，队列qu中不保存该值

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

```
bool 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。

——本讲完——