队列

（1）掌握队列的相关概念、特点和基本操作（入队、出队、判队空等）。

（2）掌握队列的顺序存储和链式存储的实现。

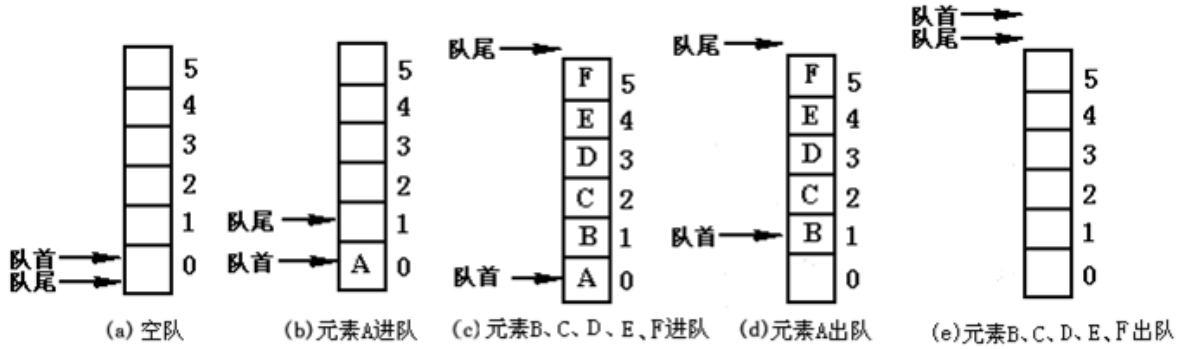
（3）了解队列的一些典型应用。

队列

定义

- 与堆栈一样，**队列**也是一种特殊的线性表，这种表在一端进行插入操作，而在另一端进行删除操作
- 允许插入的一端叫**队尾(rear)**，在下图的上方
- 允许删除的一端叫**队首(front)**，在下图的下方
- 位于队首和队尾的元素分别称为**队首元素**和**队尾元素**
- 当表中无元素时，称为**空队**

图示



注意事项

- 队列的插入称为**进队**，队列的删除称为**出队**
- 数据元素从**rear（队尾，上方）**进队，从**front（队首，下方）**出队
- **先进入队列的元素比后进入队列的元素先出队列（后进入队列的元素比先进入的元素后出队列）**，即队列是一个先进先出(FIFO)表
- 队列判空条件为：**front == rear**, 而front==rear==0是队列初始状态（本书约定）
- 进队时，先把元素插入到rear所指示的位置，然后rear++(保证rear始终**指向队尾元素的后一位**)
- 出队时，先取出front所指示的队首元素，然后front++(保证front始终**指向真正的队首元素**)
- 如果存储队列长度为maxsize，则rear >= maxsize时，队满；当rear == front时，队空

抽象数据类型ADT

```
ADT Queue{
   Data：数据元素相同数据类型，相邻元素具有前驱和后继的关系。
   Operation：
        InitQueue(&Q，maxsize，incresize)
            操作结果：构造一个容量为maxsize的空队列Q。
        ClearQueue(&Q)
            初始条件：队列Q已存在。
            操作结果：将Q清为空队列。
```

```
        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的队首元素，并用e返回其值。
        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。
        QueueTraverse(Q)
             初始条件：队列Q已存在且非空。
             操作结果：从队头到队尾依次输出Q中各个数据元素。
        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。
        DestroyQueue(&Q)
            初始条件：队列Q已存在。
             操作结果：队列 Q 被撤销，不再存在。
}ADT Queue
```

顺序队列

定义

- **顺序队列**就是队列以顺序存储结构存储
- **顺序队列**包括**顺序非循环队列**和**顺序循环队列**
- **顺序队列**与顺序栈一样，都是特殊的顺序表，只是顺序栈的插入操作在队尾进行，删除操作在队首进行

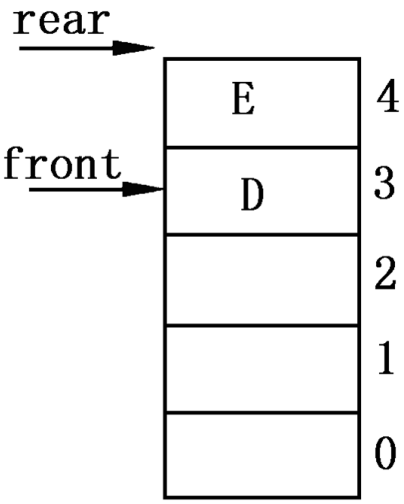
注意事项

- 顺序队列中，front和rear的值其实都是数组下标（假指针）
- 顺序队列的判空条件是： rear == front，当其都为0时，队列为初始状态

假溢出

定义

- 当元素被插入到数组中下标最大的位置上时，数组的底端还有空闲空间，此时如果还有元素入队，就会发生“溢出”现象，显然这种溢出并不是真正的溢出，而是**“假溢出”**。
- 相当于：**“上面满了，下面还有空的“。**
- **假溢出**的判断条件是： rear >= maxsize && front > 0。
- **假溢出**是队满时候的一种情况，另一种是真溢出。



- **假溢出**只存在于**非循环顺序队列**中。

解决方法

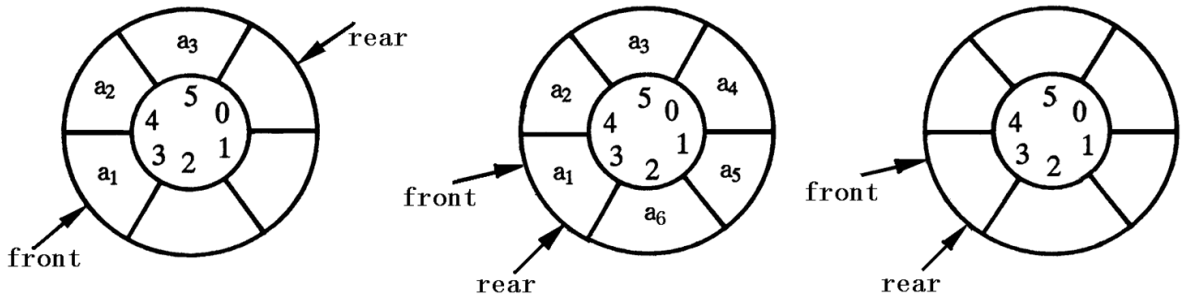
1. 修改**出队**算法（事先避免假溢出），使每次出队列后都把队列中剩余的元素向front方向移动一个位置。O(n)
2. 修改**进队**算法（假溢出发生时的解决方法），使得当真溢出时返回false; 当假溢出时，把队列中的所有元素向front方向移动一个位置。O(n)
3. 顺序循环队列。

顺序循环队列

定义

- **顺序循环队列**就是把顺序队列改造成一个头尾相连的循环表
- **顺序循环队列**初始状态为front = rear = 0
- 入队时，把元素插到rear指示位置，然后rear++
- 出队时，把front指示位置元素删除，然后front++
- **顺序循环队列**可以解决**假溢出**问题，因为如果maxsiza-1个位置被占用后，只要前方还有空间，就可以将新的元素加入下标为0的位置。（如图3.13）
- 判断下次进入的位置：[例]若maxsize = 6，rear = 5, 则下次进队的位置为0，因为(rear+1)%6 = 0

状态图示



(a)队列中有3个元素　(b)队满状态　(c)队空状态

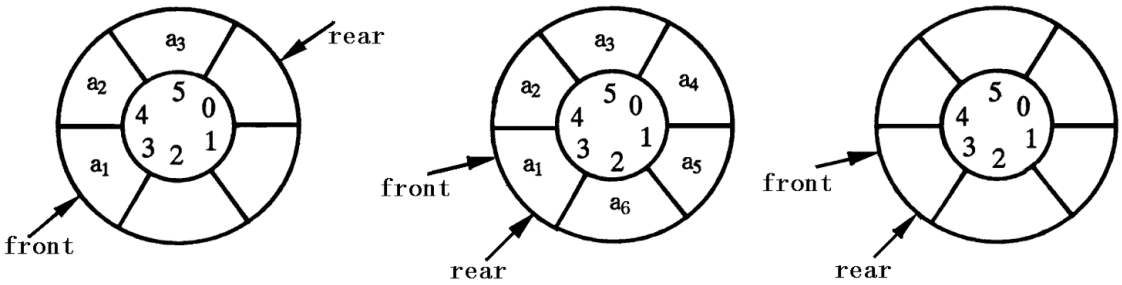
- 第一种状态：队列中有元素（既非空，也非满）
  rear != front
- 第二种状态：队满
  (rear == front) && **front与rear所指示的位置有元素**
- 第三种状态：队空
  (rear == front) && **front与rear所指示的位置无元素**

注意事项

- **顺序循环队列**可以解决"假溢出"问题，但是会带来"无法判空"的问题。
- **顺序队列**判空条件是rear == front，而满足这个条件的**顺序循环队列**有可能为**空**，有可能为**满。**

如何解决"无法判空"问题？

- **解决方案1：少用一个存储单元**(源代码所用方案)
  即修改判满操作，使得进行插入操作时，首先判断rear所指示的下一个位置是否是front，如果是，则停止插入。（队尾留一个单元）
  判满条件：**(rear+1) %maxsize == front**
  判空条件：**rear == front**



(a)队列中有3个元素　(b)队满状态　(c)队空状态

- **解决方案2：设置一个标志位**
  即设置一个标志位tag=0,当进队成功时tag=1；当出队成功时tag=0
  判满条件：**(rear==front) && (tag==1)**
  判空条件：**(rear==front) && (tag==0)**
- **解决方案3：设置计数器 即设置一个计数器count=0，当进队成功时count++；当出队成功时count--**
  判满条件：(rear==front) && count>0** 判空条件：**count == 0**

如何求顺序循环队列里面有多少元素

元素个数N = (rear-front+maxsize) % maxsize

顺序循环队列的数据结构

```
[in SqQueue.h]
typedef struct {
ElemType  *queue;                    // 存储数据元素的一维数组
  int front;                         // 队首指针，指向队首元素
  int rear;                          // 队尾指针，指向队尾元素的下一个位置
  int queuesize;                     // 循环队列当前的最大容量
  int incrementsize;                 // 增补空间量
} SqQueue;
```

顺序循环队列的初始化

```
[in SqQueue.h]
void InitQueue_Sq(SqQueue &Q, int maxsize=QUEUE_INIT_SIZE,
                                  int incresize=QUEUEINCREMENT  )
{
    Q.queue=(ElemType *)malloc(maxsize*sizeof(ElemType));
    if(!Q.queue)  exit(1);
     Q.front=Q.rear=0;
     Q.queuesize=maxsize;
     Q.incrementsize=incresize;
}// InitQueue_Sq
```

求顺序循环队列长度

```
[in SqQueue.h]
int QueueLength_Sq(SqQueue Q)
{
    return (Q.rear-Q.front+Q.queuesize) % Q.queuesize;
}// QueueLength_Sq
```

顺序循环队列进队

插入元素e到队尾，成功插入返回true，否则返回false

```
[in SqQueue.h]
bool EnQueue_Sq(SqQueue &Q,ElemType e)
{
  if((Q.rear+1)%Q.queuesize==Q.front)   // 队满，给循环队列增补空间
  { Q.queue=(ElemType *)realloc(Q.queue,
(Q.queuesize+Q.incrementsize)*sizeof(ElemType));
    if(!Q.queue)     return false;
if(Q.front>Q.rear)    // 队尾指针在队首指针前面，重新确定队首指针的位置
     {  for(int i=Q.queuesize-1;i>=Q.front;i--)
// 将Q.front到queuesize-1之间的元素后移Q.incrementsize个位置
          Q.queue[i+Q.incrementsize]=Q.queue[i];
         Q.front+=Q.incrementsize;       // 队首指针后移Q.incrementsize个位置
      }
        Q.queuesize+=Q.incrementsize;  // 队列容量增加Q.incrementsize
    }
    Q.queue[Q.rear]=e;                   // 元素e插入到队尾
    Q.rear=(Q.rear+1) %Q.queuesize;      // 队尾指针顺时针移动一个位置
    return true;
}// EnQueue_Sq
```

顺序循环队列出队

删除队尾元素，并用e返回其值，成功删除返回true;否则返回false

```
[in SqQueue.h]
bool DeQueue_Sq(SqQueue &Q,ElemType &e)
{
  if(Q.front==Q.rear)    return false;     // 队空
    e=Q.queue[Q.front];                    // 取出队首元素
  Q.front=(Q.front+1) %Q.queuesize;        // 队首指针顺时针移动一个位置
return true;
}// DeQueue_Sq
```

顺序循环队列取队首元素

```
[in SqQueue.h]
bool GetHead_Sq(SqQueue Q, ElemType &e)
{
  if(Q.front==Q.rear)    return false;    // 队空
  e=Q.queue[Q.front];
  return true;
}// GetHead_Sq
```

顺序循环队列判断队空

```
[in SqQueue.h]
bool QueueEmpty_Sq(SqQueue Q)
{      // 队空返回true，队满返回false
   return Q.rear==Q.front;
 }// QueueEmpty_Sq
```

顺序循环队列撤销队列

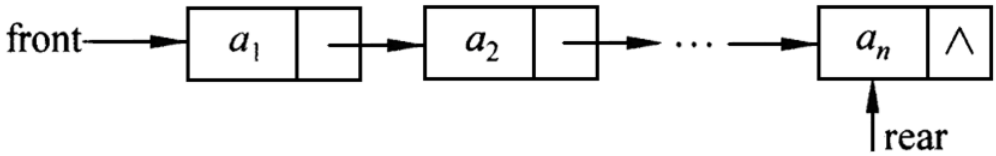
```
[in SqQueue.h]
void DestroyQueue_Sq(SqQueue &Q )
{
    free(Q.queue);
    Q.queue=NULL;
    Q.queuesize=0;
    Q.front=Q.rear=0;
}// DestroyQueue_Sq
```

链队

定义

- 队列的链式存储结构简称**链队**，链队中每一个数据元素用一个节点表示，实际上就是一个单链表(但是操作上有约束)。

逻辑图示



- 此图为不带头结点的链队

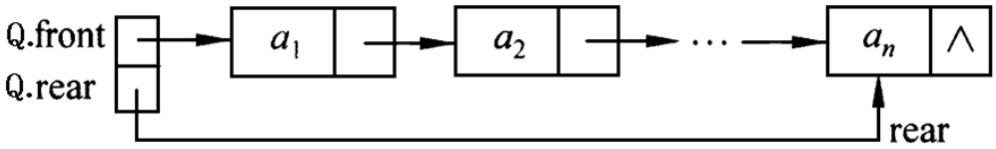
注意事项

- 实际上就是一个(带或不带头结点)单链表(但是操作上有约束)
- **链队有两个指针**，队首指针front和队尾指针rear
- front指向队列的当前队首节点位置，rear指向队列的当前队尾节点位置
- 链队也有带头结点的链队和不带头结点的链队

链对的数据结构

与单链表一样

```
[in LinkQueue.h]
typedef struct {
   QueuePtr front;                  // 队首指针
   QueuePtr rear;                   // 队尾指针
}LinkQueue;                         // 链队
```



链队的初始化

```
[in LinkQueue.h]
void InitQueue_L(LinkQueue &Q)
  {
     Q.front=Q.rear=NULL;              // 队首指针和队尾指针置空
  }// InitQueue_L
```

求链队的长度

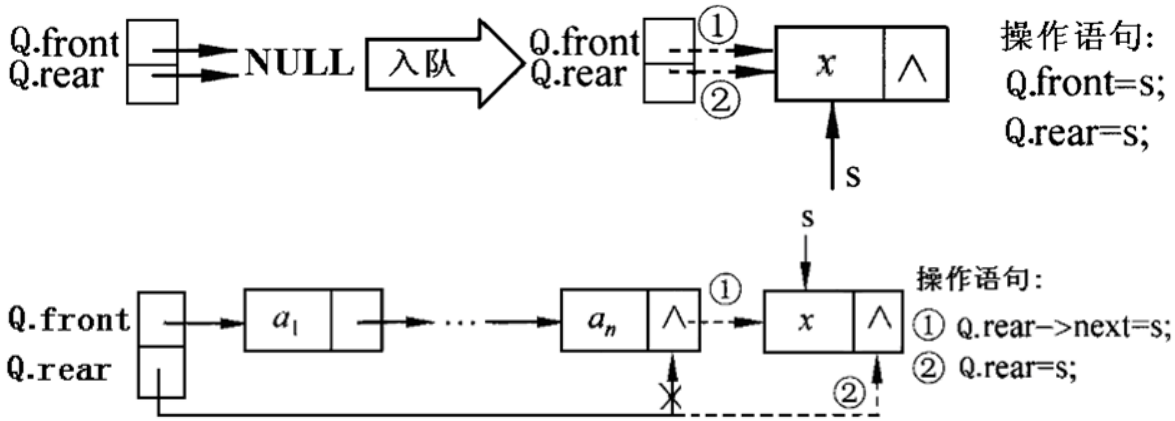
统计链队Q中数据元素的个数，并返回统计结果

```
[in LinkQueue.h]
int QueueLength_L(LinkQueue Q)
{
  int k=0;
  QueuePtr p=Q.front;
    while(p)
    { k++;
      p=p->next;                          // 访问下一个结点
    }
   return k;
}// QueueLength_L
```

进链队操作

插入元素e为链队Q中新的队尾元素

```
[in LinkQueue.h]
bool EnQueue_L (LinkQueue &Q, ElemType e)
{
   QueuePtr s;
  if((s=(LNode *)malloc(sizeof(LNode)))==NULL) return false;  // 存储分配失败
   s->data = e;                                  // 把e的值作为新结点的值域
   s->next = NULL;                               // 新结点的指针域置空
   if(Q.rear==NULL)                  // 若链队为空,则新结点既是队首结点又是队尾结点
       Q.front=Q.rear=s;
   else                             // 若链队非空，则新结点被链接到队尾并修改队尾指针
     Q.rear=Q.rear->next=s;
 return true;
}// EnQueue_L
```

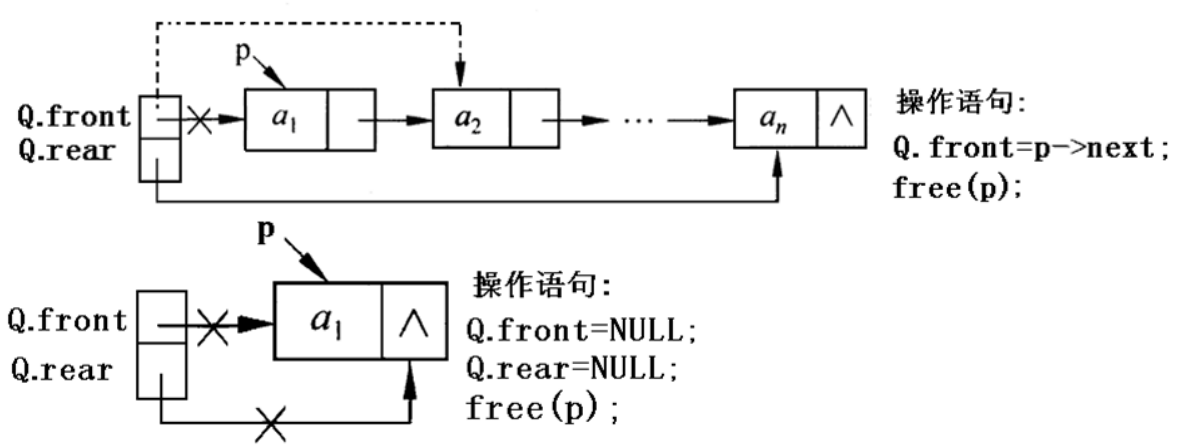


出链队操作

删除Q的队头元素，并用e返回其值。成功删除返回true；否则返回false

```
[in LinkQueue.h]
bool DeQueue_L(LinkQueue &Q, ElemType &e)
{
   QueuePtr p;
   if (Q.front==NULL)                // 若链队为空，则返回"假"
```

```
      return false;
   p=Q.front;                    // 暂存队首指针以便回收队首结点
   e=p->data;                    // e返回队首元素的值
   Q.front = p->next;              // 队首指针指向下一个结点
   if (Q.front==NULL)              // 若删除后队列为空，则使队尾指针为空
      Q.rear = NULL;
   free(p);                        // 回收原队首结点
   return true;
}// DeQueue_L
```



取队首元素操作

```
[in LinkQueue.h]
bool GetHead_L(LinkQueue Q,ElemType &e)
{     // 取队首元素，并让e返回其值
   if(Q.front)                       // 队非空
  {  e=Q.front->data;          // 元素e返回其值
        return true;
}
   else return false;       // 队空，取栈顶元素失败
 }// GetHead_L

bool QueueEmpty_L(LinkQueue Q)
{
  if(!Q.front) return true;
  else return false;
}// QueueEmpty_L
```

链队判空

判断链栈S是否为空，若为空则返回true,否则返回false

```
[in LinkQueue.h]
bool QueueEmpty_L(LinkQueue Q)
{
 if(!Q.front) return true;
 else return false;
}// QueueEmpty_L
```

撤销链栈操作

释放链栈所占空间

```
[in LinkQueue.h]
void DestroyQueue_L(LinkQueue &Q )
{
  QueuePtr p,p1;
  p=Q.front;
  while(p)
   {  p1=p;
      p=p->next;
      free(p1);
   }
   Q.front=Q.rear=NULL;
}// DestroyQueue_L
```
