第3章 栈和队列

【本章学习目标】

栈和队列是在程序设计中被广泛使用的两种线性数据结构。栈只允许在表的一端进行插入或删除操作，而队列只允许在表的一端进行插入操作在另一端进行删除操作。因而，栈和队列也可以被称作操作受限的线性表。通过本章的学习，要求：

- 理解栈和队列的特点；
- 熟练掌握顺序栈和链栈基本操作的算法实现；
- 熟练掌握循环队列和链队列基本操作的算法实现；
- 掌握栈和队列的应用。

3.1 栈

3.1.1 栈的引例

为了说明栈的概念，举一个简单的例子。把餐馆中洗净的一叠盘子看作一个栈。通常情况下，最先洗净的盘子总是放在最下面，后洗净的盘子放在先洗净的盘子上面，最后洗净的盘子总是放在最上面；使用时，总是先从顶上取走，也就是说，后洗净的盘子先取走，先洗净的盘子后取走，即所谓的“先进后出”。栈的应用非常广泛，对高级语言中表达式的处理就是通过栈来实现的。在程序设计中，如果需要以与保存数据时相反的顺序来使用数据，就可以用栈来实现。

3.1.2 栈的类型定义

栈(Stack)是限定在表的一端进行插入和删除操作的线性表。允许插入和删除的一端称作栈顶(Top)，固定的另一端称作栈底(Bottom)。当栈中没有元素时称为空栈。

如图 3.1 所示，栈中有 n 个元素，进栈的顺序是 $a_0, a_1, \cdots, a_{n-1}$，当需要出栈时，其顺序为 $a_{n-1}, \cdots, a_1, a_0$，所以栈又称为后进先出(Last In First Out)的线性表，简称 LIFO 表。

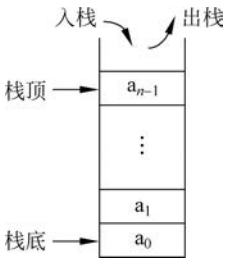

图 3.1 栈的示意图

对于栈，有以下几种基本运算。

(1) 栈的初始化 Init_Stack(s)：构造一个空栈。

(2) 判栈空 Empty_Stack(s)：若栈 s 为空栈，则函数返回值为 1，否则返回值为 0。

(3) 入栈 Push_Stack(s,x)：在栈 s 的顶部插入一个新元素 x，x 成为新的栈顶元素，栈

发生变化。

(4) 出栈 Pop_Stack(s)：将栈 s 的顶部元素从栈中删除，栈中少了一个元素，栈发生变化。

(5) 读栈顶元素 Top_Stack(s)：返回栈顶元素，栈不变化。

(6) 求栈的长度 StackLength(s)：返回栈 s 中的元素个数。

3.1.3 栈的顺序存储表示和操作的实现

与第 2 章讨论的一般顺序存储结构的线性表一样，利用一组地址连续的存储单元依次存放从栈底到栈顶的数据元素，这种形式的栈称为顺序栈。因此，可以使用预设的足够长度的一维数组 datatype data[MAXSIZE]来实现，将栈底设在数组小下标的一端，由于栈顶位置是随着元素的进栈和出栈操作而变化的，因此用一个指针 int top 指向栈顶元素的当前位置。用 C 语言描述顺序栈的数据类型如下：

```
#define datatype  char
#define MAXSIZE  100
typedef  struct
{datatype  data[MAXSIZE];
 int  top;
}SEQSTACK;
```

定义一个指向顺序栈的指针：

```
SEQSTACK  *s;
```

MAXSIZE 是栈 s 的最大容量。鉴于 C 语言中数组的下标约定是从 0 开始的，因而使用一维数组作为栈的存储空间时，应设栈顶指针 s->top=-1 时为空栈。元素入栈时，栈顶指针加 1，即 s->top++；元素出栈时，栈顶指针减 1，即 s->top--。当 top 等于数组的最大下标值时则栈满，即 s->top=MAXSIZE-1。图 3.2 说明了顺序栈中数据元素与栈顶指针的变化。这里设 MAXSIZE=5，图 3.2(a)是空栈状态，图 3.2(b)是元素 A 入栈后的状态，图 3.2(c)是栈满状态，图 3.2(d)是在图 3.2(c)之后 E、D 相继出栈，此时栈中还有 3 个元素，或许最近出栈的元素 D、E 仍然在原先的单元存储着，但 top 指针已经指向了新的栈顶，则元素 E、D 已不在栈中了。

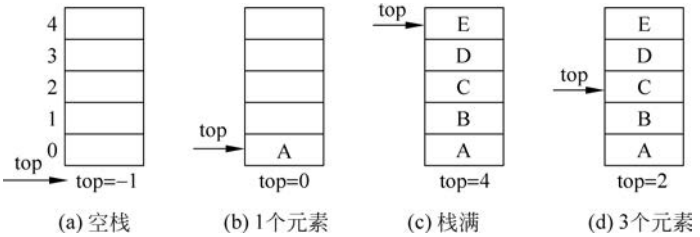

图 3.2 栈顶指针 top 与栈中数据元素的关系

在上述存储结构上基本操作的算法实现如下：

1. 初始化空栈操作

```
void  Init_Stack(SEQSTACK *s)          /*创建一个空栈由指针 s 指出*/
{ s->top=-1;
}
```

2. 判栈空操作

```
int Empty_Stack (SEQSTACK * s)          /* 栈 s 空时,返回 1; 非空时,返回 0 */
{
 if(s->top == -1)
   return 1;
 else
   return 0;
}
```

3. 入栈操作

```
void  Push_Stack(SEQSTACK * s,datatype x)       /* 将元素 x 插入栈 s 中,作为 s 的新栈顶 */
{
  if(s->top == MAXSIZE-1)              /* 栈满 */
     printf("Stack full\n");
  else
     { s->top++;
     s->data[s->top] = x;
     }
}
```

4. 出栈操作

```
datatype  Pop_Stack(SEQSTACK * s )      /* 若栈 s 不为空,则删除栈顶元素 */
{
 datatype  x;
 if(Stack_Empty(s))                     /* 栈空 */
 { printf(" Stack empty\n");
   return NULL;}
else
  {x = s->data[s->top];
  s->top-- ;
  return x;}
}
```

5. 读栈顶元素操作

```
datatype Top_Stack(SEQSTACK * s)        /* 若栈 s 不为空,则返回栈顶元素 */
{
  datatype x;
  if(Stack_Empty(s))                    /* 栈空 */
   { printf("Stack empty.\n");
   return NULL;}
  else
    {x = s->data[s->top];
    return x;}
}
```

6. 求栈的长度操作

```
int StackLength(SEQSTACK * s)
{
 return s->top + 1;
}
```

对于其他类型栈的基本操作的实现,只需改变结构体 SEQSTACK 中的数组

data[MAXSIZE]的基本类型,即对 datatype 重新进行宏定义。

对于栈的顺序存储结构的两点说明如下。

(1) 对于顺序栈,入栈时,首先判断栈是否满,栈满的条件为 s->top==MAXSIZE-1,栈满时不能入栈,否则会出现空间溢出,引起错误,这种现象称为上溢。

(2) 出栈和读栈顶元素时,要先判断栈是否为空,为空时不能操作,否则产生错误(或称为下溢)。通常栈空常作为一种控制转移的条件。

3.1.4 栈的链式存储表示和操作的实现

栈也可以采用链式存储结构表示,这种链式存储结构的栈称为链栈。用 C 语言描述链栈的数据类型如下:

```
typedef struct Stacknode
{
datatype data;
struct Stacknode * next;
}LINKSTACK;
```

定义一个栈顶指针:

```
LINKSTACK * top;
```

在一个链栈中,栈底就是链表的最后一个结点,而栈顶总是链表的第一个结点。栈顶指针 top 唯一确定一个链栈,当 top=NULL 时,该链栈是一个空栈。新入栈的元素成为链表的第一个结点,只要系统还有存储空间,就不会有栈满的情况发生。图 3.3 给出了链栈中数据元素与栈顶指针 top 的关系。

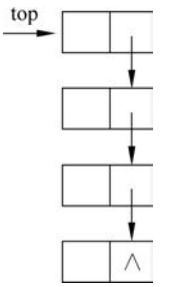

图 3.3 链栈示意图

链栈基本操作的算法实现如下。

1. 初始化空栈

```
void Init_Stack(LINKSTACK * top)
{
top = NULL;
}
```

2. 判栈空操作

```
int Empty_Stack(LINKSTACK  * top)              /* 栈 s 为空时,返回 1; 非空时,返回 0 */
{
 if(top == NULL)
    return 1;
 else
    return 0;
}
```

3. 入栈操作

```
void  Push_Stack(LINKSTACK  * top, datatype x)     /* 将元素 x 压入链栈 top 中 */
{
LINKSTACK * p;
p = (LINKSTACK * )malloc(sizeof(LINKSTACK));     /* 申请一个结点 */
p -> data = x;
```

```
p->next = top;
top = p;
}
```

4. 出栈操作

```
datatype  Pop_Stack(LINKSTACK  *top)          /*从链栈top中删除栈顶元素*/
{
 datatype x;
 LINKSTACK *p;
 if (top == NULL)                             /*空栈*/
 {printf(" Stack empty\n");
  return NULL;}
 else
 { p = top;
  top = top->next;
  x = p->data;
  free(p);
  return x;
 }
}
```

5. 读栈顶元素操作

```
datatype Top_Stack(LINKSTACK *top)            /*若栈s不为空,则返回栈顶元素*/
{
 datatype x;
if (top == NULL)                              /*空栈*/
{printf(" Stack empty\n");
 return NULL;}
else
{ x = top->data;
 return x;}
}
```

3.2 栈的应用

【例 3.1】 编写算法，将十进制正整数 m 转换为 n 进制数。

算法分析：利用“辗转相除法”，即不断地用 m 除以 n，直到 $m=0$ 为止，将各项除得的余数倒排。这里将十进制数 m 除以 n 所得的各位余数入栈，然后依次出栈并输出即可。

```
void zhuan(int m,int n)
{int i;
 SEQSTACK *S;
 S->top = -1;
 While(m!= 0)
 {Push(s,m % n);
  m = m/n;}
For(i = S->top;i>=0;i--)
   Printf(" %d",s->data[i]);
}
```

【例3.2】 编写算法,利用栈将带头结点的单链表逆置。

算法分析:把单链表S1看成一个链栈,将从S1出栈的元素依次入栈S2,直到S1为空为止,此时S2就是S1的逆置。

```
Void nizhi(LinkStack * S1,LinkStack * S2)
{int x;
 While(S1 -> next!= NULL)
 {x = Pop(S1);
  Push(S2,x);}}
```

3.3 队　　列

3.3.1 队列的引例

和栈一样,队列也是一种特殊的线性表。对于队列我们并不陌生,商场、银行的柜台前需要排队,餐厅的收款机旁需要排队。这里我们来看一个简单的事件排队问题,用户可以输入和保存一系列事件,每个新事件只能在队尾插入;每次先处理队头事件,即先输入和保存的事件,当队头事件处理完毕后,它就会从事件队列中被删除;还可以查询事件队列中剩余的事件。

3.3.2 队列的定义及其基本操作

队列(Queue)也是一种运算受限的线性表。它只允许在表的一端进行插入,在另一端进行删除。允许插入的一端称为队尾(Rear),允许删除的一端称为队头(Front)。在队列中,先进入队列的成员总是先离开队列。因此,队列也称作先进先出(First In First Out)的线性表,简称FIFO表。图3.4显示了队列的这种特点。

当队列中没有元素时,称为空队列。在空队列中依次加入元素$a_1,a_2,\cdots,a_n$之后,a_1称为队头元素,a_n称为队尾元素。图3.4所示是一个有n个元素的队列,入队的顺序依次为素$a_1,a_2,\cdots,a_n$,出队时的顺序将依然是$a_1,a_2,\cdots,a_n$。

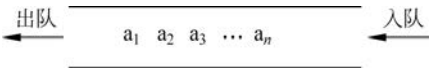

图3.4　队列示意图

对于队列,有以下几种基本运算。

(1) 队列初始化Init_Queue(q):构造了一个空队列。

(2) 判队列空Empty_Queue(q):若队列q为空队列,则函数返回值为1,否则返回值为0。

(3) 入队列Add_Queue(q,x):在队列q的尾部插入一个新元素x,x成为新的队尾。

(4) 出队列Del_Queue(q):删除队头元素,并返回该队头元素,队列发生变化;若队列空,则函数返回值为0。

(5) 读队头元素Front_Queue(q):返回队头元素,队列不变;若队列空,则函数返回值为0。

(6) 求队列的长度Length_Queue(q):返回队列q中的元素个数。

3.3.3 队列的顺序存储表示和操作的实现

队列是一种特殊的线性表，因此队列可采用顺序存储结构存储，也可以使用链式存储结构存储。利用一组地址连续的存储单元依次存放队列中的数据元素，这种形式的队列称为顺序队列。一般情况下，使用一维数组作为队列的顺序存储空间，另外再设立两个指针：一个为指向队头元素位置的指针 front；另一个为指向队尾元素位置的指针 rear。随着元素的入队和出队操作，front 和 rear 是不断变化的。用 C 语言描述顺序队列的数据类型如下：

```
#define  datatype  char
#define  MAXSIZE  100                          /* 队列的最大容量 */
typedef  struct
{ datatype   data[MAXSIZE];                    /* 队列的存储空间 */
  int  front,rear;                             /* 队头队尾指针 */
}SEQUEUE;
```

定义一个指向顺序队列的指针变量：

```
SEQUEUE  *q;
```

下面分析队列的基本操作的实现。设 MAXSIZE＝5，C 语言中，数组的下标是从 0 开始的，因此为了算法设计的方便，我们约定：初始化队列时，q－＞front＝q－＞rear＝－1。图 3.5(a)所示为初始化空队列的情况。在队列中插入一个元素 x 时，即在队尾插入，则有操作 q－＞rear＋＋；q－＞data[q－＞rear]＝x。图 3.5(b)所示为在队列中依次插入 2 个元素 A、B 后的情况，此时尾指针 rear 始终指向队尾元素所在位置，头指针 front 指向队列中第一个元素的前面一个位置。出队操作，即从队头删除一个数据元素，则有 q－＞front＋＋，如图 3.5(c)所示，在图 3.5(b)之后元素 A、B 依次出队列，队列空，此时队头指针发生变化，队尾指针不变。由此可见，当 front 和 rear 等于数组的同一个下标值时队列空，即 q－＞front＝q－＞rear。图 3.5(d)是在图 3.5(c)之后，C、D、E 相继插入队列，队列满，此时队尾指针 q－＞rear＝MAXSIZE－1，队头指针不变。由此可见，当 rear 等于数组的最大下标值时，则队列满，即 q－＞rear＝MAXSIZE－1。

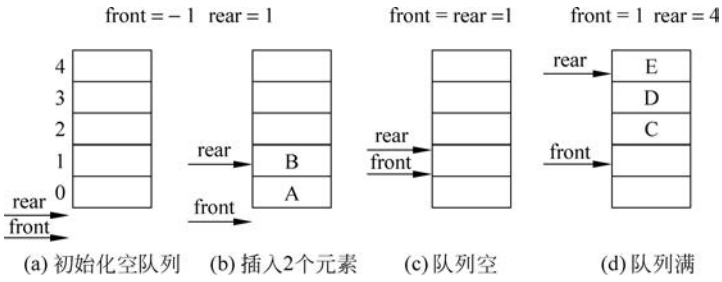

图 3.5 顺序队列基本操作分析图示

随着入队出队操作的进行，整个队列整体向后移动，这样就出现了图 3.5(d)所示的现象：q－＞rear＝4，给出了队列满信号，再有元素入队就会出现溢出，而事实上此时队列中并未真的“满员”，这种现象为“假溢出”，这是由于“队尾入，队头出”这种受限操作造成的。解决这一问题的常用方法是采用循环队列，将队列存储空间的最后一个位置绕到第一个位置，

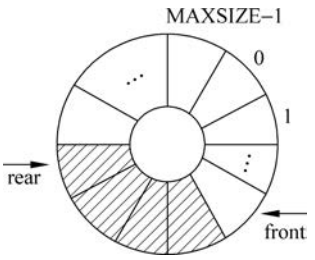

图 3.6　循环队列示意图

即将 q->data[0]接在 q->data[MAXSIZE-1]之后,形成逻辑上的环状空间,如图 3.6 所示。

在循环队列中,每插入一个新元素时,就把队尾指针 rear 沿顺时针方向移动一个位置。即

```
q->rear = q->rear + 1;
if(q->rear == MAXSIZE) q->rear = 0;
```

或改成一种更简洁的描述方式:

```
q->rear = (q->rear + 1) % MAXSIZE;
```

在循环队列中,每删除一个元素时,就把队头指针 front 沿顺时针方向移动一个位置。即

```
q->front = q->front + 1;
if(q->front == MAXSIZE) q->front = 0;
```

或改成一种更简洁的描述方式:

```
q->front = (q->front + 1) % MAXSIZE;
```

从图 3.7 所示的循环队列中可以看出,图 3.7(a)中有 A、B、C 三个元素,此时 front=2、rear=0;随着元素 D、E 相继入队,队中有 5 个元素,队列满情况出现,此时 front=2、rear=2,有 q->front=q->rear,如图 3.7(b)所示。若在图 3.7(a)情况下,元素 A、B、C 相继出队,队列空情况出现,此时 front=0、rear=0,也有 q->front=q->rear,如图 3.7(c)所示。上述队列满和队列空情况出现的过程不同,但队列满和队列空情况的结果相同。这显然是必须要解决的一个问题。

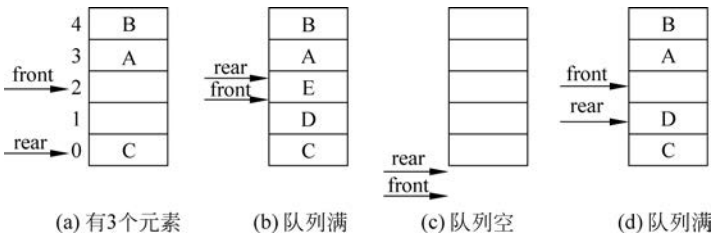

图 3.7　循环队列队列空、队列满条件分析

解决这个问题的方法有很多种。一种简单的方法是:损失一个元素空间不用,即当循环队列中元素的个数是 MAXSIZE-1 时就认为队列满了,图 3.7(d)所示的情况就是队列满,这样队列满的条件变为(q->rear+1) % MAXSIZE=q->front,也能和队列空区别开。

现将循环队列的基本操作的准则总结如下。

循环队列的初始化条件:q->front=q->rear=0。

循环队列的队列满条件:(q->rear+1) % MAXSIZE=q->front。

循环队列的队列空条件:q->front=q->rear。

下面给出循环队列基本操作的算法实现。

1. 初始化队列

```
void  Init_Queue(SEQUEUE *q)                    /*创建一个空队列由指针 q 指出*/
{
```

```
q->front = 0;
q->rear = 0;
}
```

2. 判队列空操作

```
int  Empty_Queue(SEQUEUE  *q)                /*队列q为空时,返回1; 非空时,返回0*/
{
if (q->front == q->rear)
 return 1;
else
 return 0;
}
```

3. 入队列操作

```
void  Add_Queue(SEQUEUE *q,datatype  x)      /*将元素x插入队列q中,作为q的新队尾*/
{
if ((q->rear + 1) % MAXSIZE == q->front)     /*队列满*/
  printf(" Queue full\n");
else
 {q->rear = (q->rear + 1) % MAXSIZE;
   q->data[q->rear] = x;
 }
}
```

4. 出队列操作

```
datatype  Del_Queue(SEQUEUE *q)      /*若队列q不为空,则删除队头元素,并返回队头元素*/
{
datatype  x;
if (Empty_Queue(q))                  /*队列为空*/
{ printf(" Queue empty\n");
  return NULL;}
else
{q->front = (q->front + 1) % MAXSIZE;
 x = q->data[q->front];
 return x; }
}
```

5. 读队头元素操作

```
datatype  Front_Queue(SEQUEUE *q)    /*若队列q不为空,则返回队头元素*/
{
datatype  x;
if (Empty_Queue(q))                  /*队列为空*/
  { printf(" Queue empty\n");
    return NULL;}
else
  {x = q->data[(q->front + 1) % MAXSIZE];
   return x; }
}
```

6. 求队列长度操作

```
int Length_Queue(SEQUEUE *q)         /*返回队列q的元素个数*/
{
  int len;
```

```
    len = (MAXSIZE + q->rear - q->front) % MAXSIZE;
return len;
}
```


3.3.4 队列的链式存储表示和操作的实现

队列也可以采用链式存储结构表示,这种链式存储结构的队列称为链队列。用C语言描述链队列的数据类型如下:

```
typedef struct Queuenode
 { datatype  data;
   struct  Queuenode * next;
 } Linknode;                          /* 链队结点的类型 */
typedef struct
 { Linknode   * front, * rear;
 }LINKQUEUE;                          /* 将头尾指针封装在一起的链队列 */
```

定义一个指向链队列的指针:

```
LINKQUEUE  * q;
```

为了操作方便,和线性链表一样,也给链队列添加一个头结点,并设定头指针指向头结点。因此,空队列的判定条件就变为头指针和尾指针都指向头结点。图3.8给出了链队列头尾指针与数据元素的关系。图3.8(a)是具有一个头结点的空队列;元素a、b相继入队,将链队列中最后一个结点的指针域指向新元素,还要将队列中的尾指针指向新元素,如图3.8(b)和图3.8(c)所示;在图3.8(c)的情况下,删除队首元素a,只需修改头结点的指针域,将其指向队首元素的下一个结点,如图3.8(d)所示;若链队列中只有一个元素时,除了修改头结点的指针域,还要修改链队列的尾指针,将其指向头结点,如图3.8(e)所示。

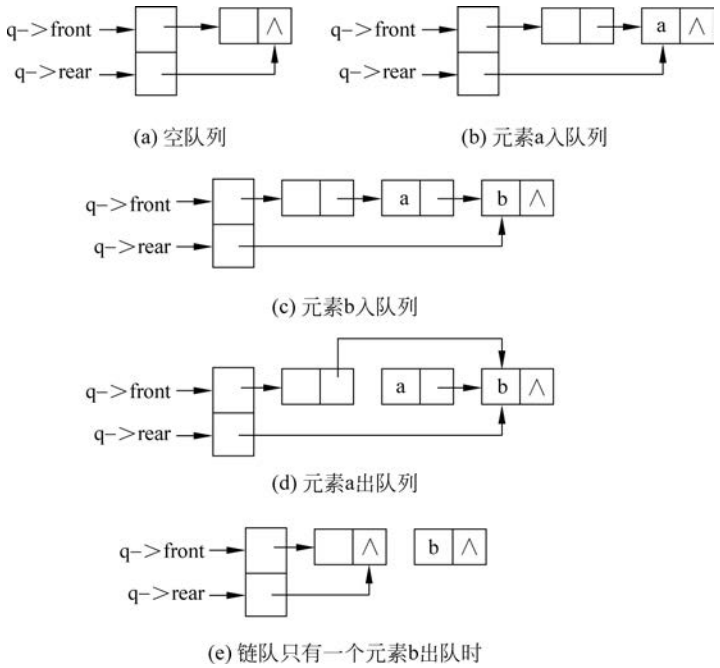

图3.8 链队列指针front、rear与数据元素的关系

链队列的基本运算如下。

1. 初始化队列

```
void Init_Queue(LINKQUEUE * q)
{/ * 创建一个带头结点的空链队列 q * /
 Linknode * p;
 p = (Linknode * )malloc(sizeof(Linknode));  / * 申请链队列头结点 * /
 p - > next = NULL;
 q - > front = q - > rear = p;
}
```

2. 判队列空操作

```
int  Empty_Queue(LINKQUEUE * q)
{ / * 队列 q 为空时,返回 1; 非空时,返回 0 * /
if (q - > front == q - > rear)   return 1;
else  return 0;
}
```

3. 入队列操作

```
void Add_Queue(LINKQUEUE * q,datatype x)
{/ * 将元素 x 插入到链队列 q 中,作为 q 的新队尾 * /
Linknode * p;
p = (Linknode * )malloc(sizeof(Linknode));
p - > data = x;
p - > next = NULL;                          / * 置新结点的指针为空 * /
q - > rear - > next = p;                     / * 将链队列中最后一个结点的指针指向新结点 * /
q - > rear = p;                             / * 将队尾指向新结点 * /
}
```

4. 出队列操作

```
datatype Del_Queue(LINKQUEUE * q)
{ / * 若链队列 q 不为空,则删除队头元素,并由 x 返回其元素值 * /
Linknode * p;
datatype x;
if (Empty_Queue(q))                         / * 队列为空 * /
{ printf(" Queue empty\n");
return NULL;}
else
{p = q - > front - > next;                   / * 取队头 * /
x = p - > data;
q - > front - > next = p - > next;           / * 删除队头结点 * /
if(p - > next == NULL)
    q - > rear = q - > front;
free(p);
return x;
}
}
```

5. 读队头元素操作

```
datatype Front_Queue(LINKQUEUE * q)
{ / * 若队列 q 不为空,则返回队头元素 * /
Linknode * p;
datatype x;
```

```
if (Empty_Queue(q))                          /* 队列为空 */
{ printf(" Queue empty\n");
return NULL;}
else { p = q -> front -> next;               /* 取队头 */
x = p -> data;
return x;}
}
```

3.4 队列的应用

【例 3.3】 编写程序,从键盘输入一个整数序列 $a_1, a_2, \cdots, a_n$,若 $a_i > 0$,则 a_i 入队列;若 $a_i < 0$,则队头元素出队列;若 $a_i = 0$,则算法结束。要求利用循环队列完成,并在出现异常(如队空)时打印错误信息。

算法思路:先建立一个空的循环队列 Q,然后通过循环接收用户输入的整数。若输入的值大于 0,则将该数入队列;若输入的值小于 0,则将一个元素出队列并输出;若输入的值等于 0,则退出循环。

```
void Fun(SEQUEUE *q)
{
int x;
Init_Queue(q);
scanf("%d",&x);
while(x!= 0)
  if(x > 0)
  {
   if(!Add_Queue(q,x))
    {printf("队列满!\n");break;}
   elseif (!Del_Queue(q))
    {printf("队列空!\n");break;}
   scanf("%d:,&x);
  }
}
```

本章小结

(1) 栈是一种限定其插入、删除操作只能在线性表的一端进行的特殊结构。由于在栈中的数据元素具有后进先出的特点,因此,人们又将它称为 LIFO 线性表。栈可以用顺序存储结构和链式存储结构表示。

(2) 队列是一种限定其插入在线性表的一端进行,删除则在线性表的另一端进行的特殊结构。由于在队列中的数据元素具有先进先出的特点,因此,人们又将它称为 FIFO 线性表。同栈一样,队列也可以利用顺序存储结构或链式存储结构表示。在利用顺序存储结构表示时,为了避免"假溢出"现象的发生,需要将队列构成循环状,这就形成了循环队列。

(3) 分别介绍了栈和队列的各种运算的算法实现及简单应用。

知识拓展

栈这种处理事情的先后机制在生活中并不多见，因为它“不合情理”。人们通常认为凡事先来后到、先来先处理是公平的，先来反而等待很长时间是不公平的，因此很多人刚进入计算机专业时，会奇怪为什么实行这样一种机制。但是在计算机处理问题时，常常要把一个大问题自上而下分解成很多小问题，一个个拆解开、分别解决之后，再回过头来得到整个问题的答案。栈能够记录这中间一步步分解的复杂过程，而且由于最后合并的过程与开始时拆解的过程正好相反，因此它后进先出的特点可以很自然地将已分解的问题合并。为了体会这一点，不妨思考一下通过堆栈实现二叉树深度优先遍历的过程。从事计算机软件开发的人，要想“站得高，看得远”，就要对自己提出更高的要求，对于自己所经手的任何程序，必须了解它内部各种函数和过程调用的关系，而这些调用都是在栈的帮助下实现的。因此，对栈的理解可以帮助开发者自上而下深入理解一个程序。

计算机科学中所说的队列和日常生活中相应的概念是一致的。例如，要对一段视频进行处理，需要把它变成一帧帧图像，然后按照时间顺序送入程序中，按照先来后到的顺序逐一处理，处理后再合并、拼接成完整的视频。由于在视频文件中后一帧的内容和前一帧有相关性，因此先后顺序必须保持原样，不能随意变动，这就需要有一种先来先服务、先进先出的机制，也就是队列机制。