第 3 章 特殊线性表

栈、队列和串是 3 种特殊的线性表，其中，栈和队列都是操作特殊的线性表，串是一种数据对象和操作都特殊的线性表。本章将分别讨论这 3 种特殊线性表的顺序存储结构、链式存储结构及基本操作的实现。

3.1 栈

3.1.1 栈的定义和基本操作

1. 栈的定义

栈(stack)是一种特殊的线性表，它只允许在一端进行插入和删除操作，允许插入和删除的一端称**栈顶**，另一端称为**栈底**。处于栈顶位置的元素称为**栈顶元素**。栈中含有元素的个数称为**栈长**，含有 0 个数据元素的栈称为**空栈**。通过栈的定义可以看出，栈的特点是后进先出(LIFO)或先进后出(FILO)，因此，栈又称后进先出的线性表。

下面通过例子说明栈结构的特点。假设有一个很窄的死胡同，胡同里能容纳若干辆汽车，胡同的宽度一次只允许一辆车进出。现有五辆车，编号分别为 A、B、C、D、E。按编号的顺序依次进入死胡同，如图 3.1 所示。此时若编号为 D 的车要退出胡同，则必须等编号为 E 的车退出后才可以；若编号为 A 的车要退出胡同，则必须等编号为 E、D、C、B 的车都依次退出后才可以。这个死胡同就可以比作一个栈，编号为 A 的车的位置称为栈底，编号为 E 的车的位置称为栈顶，车进出胡同可以看作栈的插入和删除操作，插入和删除操作都在栈顶进行。

图 3.1 死胡同示意图

习惯上，把栈的插入操作称为**入栈**(或进栈、压栈)，把栈的删除操作称为**出栈**(或退栈、弹栈)。

2. 栈的基本操作

栈的基本操作主要有以下 7 种。

(1) 初始化操作 InitStack(S),用于初始化一个空栈 S。

(2) 求栈长操作 GetLen(S),用于返回栈 S 的元素个数,即栈长。

(3) 取栈顶元素操作 GetTop(S,e),通过 e 带回栈 S 的栈顶元素的值。

(4) 入栈操作 Push(S,x),用于将值为 x 元素压入栈 S,使 x 成为新的栈顶元素。

(5) 出栈操作 Pop(S,e),用于将非空栈的栈顶元素删除,同时通过 e 带回栈顶元素的值,新的栈顶元素为栈 S 中原栈顶元素的前一个元素。

(6) 判栈空操作 EmptyStack(S),用于判断栈 S 是否为空,若栈 S 为空,则返回 1,否则返回 0。

(7) 输出栈操作 List(S),用于依次输出栈 S 中的所有元素的值。

栈是一个线性表,它也有顺序存储和链式存储这两种存储结构,分别称为顺序栈和链栈。

3.1.2 顺序栈表示及实现

1. 顺序栈

栈的顺序存储结构称为**顺序栈**,它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,同时利用一个变量记录当前栈顶的位置(下标或指针),称为**栈顶指针**,栈顶指针并不一定是指针变量,也可以是下标变量。为了用 C 语言方便描述,在此约定:用下标变量记录栈顶的位置,栈顶指针始终指向栈顶元素的下一个单元;在初始化栈时,栈顶指针值为 0,表示空栈;在栈中插入新的元素后,栈顶指针增 1;在栈中删除栈顶元素后,栈顶指针减 1。

假设用一维数组 S[5]表示一个顺序栈,变量 top 为栈顶指针,图 3.2 所示为这个顺序栈的几种状态。

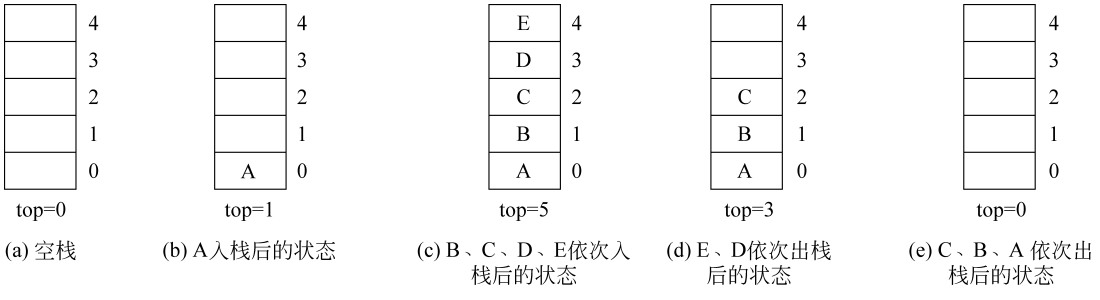


图 3.2 顺序栈 S 的几种状态

图 3.2(a)表示顺序栈为空栈,这也是初始化操作的结果,此时栈顶指针 top=0。

图 3.2(b)表示在图 3.2(a)的状态下元素 A 入栈后的状态,此时栈顶指针 top=1。

图 3.2(c)表示在图 3.2(b)的状态下元素 B、C、D、E 依次入栈后的状态,此时栈顶指针 top=5,表示栈满。

图 3.2(d)表示在图 3.2(c)的状态下元素 E、D 依次出栈后的状态,此时栈顶指针 top=3。

图 3.2(e)表示在图 3.2(d)的状态下元素 C、B、A 依次出栈后的状态，此时栈顶指针 top＝0，表示栈空。

由上面的例子可知，在用数组表示栈时，无论数组定义得多么大，栈在使用过程中都有可能会出现栈满的情况。当栈满时，C 语言中定义的数组是无法扩充空间的，因此在 C 语言中使用动态分配的一维数组，即在初始化时先用函数 malloc()为栈分配一个初始容量，在操作过程中，若栈的空间不足，则用函数 realloc()重新申请一个足够大的空间。顺序栈的类型定义如下：

```
#define INITSIZE 100            /* 存储空间的初始分配量 */
typedef int ElemType;           /* 数据元素类型 */
typedef struct
{ ElemType *base;               /* 存放元素的动态数组起始地址 */
  int top;                      /* 栈顶指针 */
  int stacksize;                /* 当前栈空间的大小 */
}SeqStack;
```

其中，INITSIZE 是为栈分配的存储空间的初始量；base 是栈空间的起始地址，也称栈底指针，它始终指向栈底的位置；top 是栈顶指针，指向栈顶元素的下一个单元，其初值为 0，作为栈空的标记；stacksize 是当前栈可以使用的最大容量。top＝＝stacksize 时表示栈满，此时若有元素入栈，则需要增加存储空间，否则将产生(上)溢出错误；top＝＝0 时表示栈空，此时若进行出栈操作，则将产生(下)溢出错误。

2. 顺序栈基本操的实现

(1) 初始化操作

创建一个空栈 S。

算法思路：分配存储空间，将栈顶指针初始化为 0，栈空间的大小为初始分配量。

```
void InitStack(SeqStack *S)
{ S->base=(ElemType *)malloc(INITSIZE*sizeof(ElemType));/* 申请存储空间 */
  S->top=0;                     /* 栈顶指针初始值为 0 */
  S->stacksize=INITSIZE;        /* 栈容量为初始值 */
}
```

(2) 求栈长操作

返回栈 S 的元素个数，即栈的长度。

算法思路：栈 S 的长度为 S－＞top。

```
int GetLen(SeqStack *S)
{ return (S->top);
}
```

(3) 取栈顶元素操作

取出栈 S 的栈顶元素的值。

算法思路：若栈不为空，则将栈顶元素值送到指定的内存单元，S－＞top 的值不变。

```
int GetTop(SeqStack * S,ElemType * e)
{ if(S->top==0) return 0;          /* 栈空,返回 0 */
  * e=S->base[S->top-1];           /* 栈顶元素值存入指针 e 所指向的内存单元 */
  return 1;                        /* 取栈顶元素成功,返回 1 */
}
```

(4) 压栈操作

将值为 x 的数据元素插入栈 S,使之成为新的栈顶元素。

算法思路:若栈满,则增加栈容量。先将 x 存入 S->top 所指的内存单元,然后将栈顶指针 S->top 增 1。

```
int Push(SeqStack * S,ElemType x)
{ if(S->top>=S->stacksize)         /* 若栈满,则增加一个存储单元 */
  { S->base=(ElemType *)realloc(S->base,(S->stacksize+1) * sizeof(ElemType));
    if(!S->base) return 0;         /* 空间分配不成功,返回 0 */
    S->stacksize++;
  }
  S->base[S->top++]=x;             /* 插入元素后,栈顶指针后移 */
  return 1;
}
```

(5) 弹栈操作

取出栈 S 的栈顶元素值,同时栈顶指针减 1。

算法思路:若栈不为空,则先将栈顶指针 S->top 减 1,然后将 S->top 单元的元素值存入指定的内存单元。

```
int Pop(SeqStack * S,ElemType * e)
{ if(S->top==0) return 0;          /* 栈空,返回 0 */
  * e=S->base[--S->top];           /* 先将栈顶指针减 1,再取顶元素值 */
  return 1;                        /* 弹栈成功,返回 1 */
}
```

(6) 判栈空操作

判断栈 S 是否为空。

算法思路:栈 S 为空的条件是 S->top 的值为 0。

```
int EmptyStack(SeqStack * S)
{ if(S->top==0) return 1;          /* 栈为空则返回 1,否则返回 0 */
  else return 0;
}
```

(7) 输出栈操作

输出栈 S 自栈顶到栈底的元素值。

算法思路:依次输出下标从 S->top-1 到 0 的元素值。

```
void List(SeqStack * S)
```

```
{ int i;
  for(i=S->top-1;i>=0;i--)
    printf("%4d ",S->base[i]);
  printf("\n");
}
```

由于栈结构具有后进先出的固有特性，因此栈成为程序设计中的有用工具。栈在计算机语言处理和将递归算法改为非递归算法等方面起着非常重要的作用。

【例 3.1】 编写算法，将十进制正整数 m 转换成 n(2≤n≤9) 进制数。

算法思路：利用“辗转相除法”，即不断地用 n 除以 m，直到 m＝0 为止，将各次除得的余数倒排。在此，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 Conversion(int m,int n)
{ SeqStack S;
  InitStack(&S);
  while(m!=0)
  { Push(&S,m%n); m=m/n;}        /*余数入栈,商作为下一个被除数*/
  List(&S);                      /*输出*/
}
```

【例 3.2】 编写算法，用非递归方法计算 n!。

算法思路：设置一个栈，将 n～1 依次入栈，然后依次出栈并乘到初值为 1 的变量 f 上即可。

```
long Fac(int n)
{ long f=1; ElemType x; SeqStack S;
  InitStack(&S);                 /*初始化空栈*/
  while(n>0)
  { Push(&S,n);n--;}             /*n~1 依次入栈*/
  while(!EmptyStack(&S))
  { Pop(&S,&x);f*=x;}            /*1~n 依次出栈并乘到初值为 1 的变量 f 上*/
  return f;
}
```

【例 3.3】 利用一个栈实现下列函数的非递归计算。

$$P_n(x)=\begin{cases}1 & n=0\\ 2x & n=1\\ 2xP_{n-1}(x)-2(n-1)P_{n-2}(x) & n>1\end{cases}$$

算法思路：设置一个栈，用于保存 n 和对应的 $P_n(x)$，栈中相邻元素的 $P_n(x)$有上述关系。然后一边出栈一边计算 $P_n(x)$，直到栈空时，计算的 $P_n(x)$即为所求。

```
typedef struct
{ int no;                        /*n 值*/
  double val;                    /*P 值*/
```

```
}ElemType;                          /*栈中的数据元素为结构体类型 */
double P(int n,double x)
{ double fv1,fv2;
  int i;
  SeqStack S;
  if(n==0) return 1;
  if(n==1) return 2*x;
  InitStack(&S);
  fv1=1;fv2=2*x;
  for(i=n;i>=2;i--) S.base[S.top++].no=i;
  while(!EmptyStack(&S))           /*若栈不为空,则计算各个 P 值*/
  { S.top--;
    S.base[S.top].val=2*x*fv2-2*(S.base[S.top].no-1)*fv1;
    fv1=fv2;
    fv2=S.base[S.top].val;
  }
  return fv2;                      /*返回最终结果*/
}
```

【例 3.4】 设计一个算法,判断一个表达式中括号“(”与“)”、“[”与“]”、“{”与“}”是否匹配。若匹配,则返回 1,否则返回 0。

算法思路:设置一个栈 S,用 i(初值为 0) 扫描表达式 exps,当遇到“(”“[”“{”时,将其入栈;若遇到“}”“]”“)”时,判断栈顶元素是否为相匹配的括号,若不匹配,则表明表达式有误;若表达式扫描完成后栈空,则表明表达式括号匹配。

```
typedef char ElemType;                  /*数据元素类型为字符型*/
int Match(ElemType *exps)
{ int i=0,nomatch=0;
  SeqStack S;
  ElemType x;
  InitStack(&S);                        /*初始化空栈*/
  while(!nomatch&&exps[i]!='\0')
  { switch(exps[i])
    { case '(':                         /*当前字符为"("时,将其入栈*/
      case '[':                         /*当前字符为"["时,将其入栈*/
      case '{':Push(&S,exps[i]);break;  /*当前字符为"{"时,将其入栈*/
      case ')':GetTop(&S,&x);
               if(x=='(') Pop(&S,&x);
               else nomatch=1;
               break;                   /*判断栈顶元素是否为相匹配的括号"("*/
      case ']':GetTop(&S,&x);
               if(x=='[') Pop(&S,&x);
               else nomatch=1;
               break;                   /*判断栈顶元素是否为相匹配的括号"["*/
```

```
        case '}':GetTop(&S,&x);
                 if(x=='{') Pop(&S,&x);
                 else nomatch=1;          /* 判断栈顶元素是否为相匹配的括号"{" */
      }
      i++;
    }
    if(EmptyStack(&S)&&!nomatch) return 1;    /* 栈空且符号匹配,返回 1 */
    else return 0;                            /* 否则返回 0 */
}
```

* 3.1.3　链栈表示及实现

1. 链栈

栈的链式存储结构称为**链栈**。链栈实际上是一个仅在表头进行操作的单链表，这种单链表的第一个结点称为栈顶结点，最后一个结点称为栈底结点，头指针指向栈顶结点（不带头结点）或头结点（带头结点）。图 3.3 是一个带头结点的链栈的示意图。本节讨论的链栈均指带头结点的链栈。

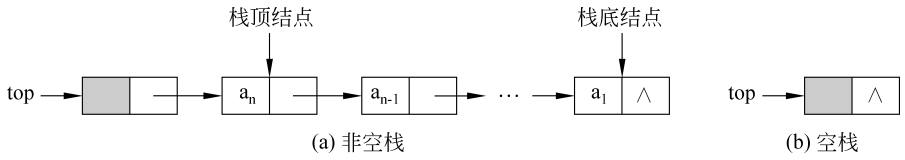


图 3.3　链栈示意图

链栈的结点类型定义如下：

```
typedef int ElemType;          /* 数据元素类型 */
typedef struct node
{ ElemType data;               /* 数据域 */
  struct node *next;           /* 指针域 */
}LinkStack;
```

这实质上就是单链表的结点类型定义。

2. 链栈基本操作的实现

下面介绍链栈基本操作的实现。

（1）初始化操作

创建一个带头结点的空栈 S。

算法思路：创建头结点并将头结点指针域置为空。

```
LinkStack *InitStack(void)
{ LinkStack *S;
  S=(LinkStack *)malloc(sizeof(LinkStack));
  S->next=NULL;
```

```
  return S;
}
```

(2) 取栈顶元素操作

取出栈 S 的栈顶元素值。

算法思路：若栈不为空，则将栈顶结点的值送到指定的内存单元，栈顶结点不变。

```
int GetTop(LinkStack *S,ElemType *e)
{ if(S->next==NULL) return 0;       /*若栈空,返回 0*/
  *e=S->next->data;                 /*否则,取栈顶元素值*/
  return 1;
}
```

(3) 求栈长操作

返回栈 S 的元素个数，即栈的长度。

其设计思路与单链表的求表长操作相同。

```
int GetLen(LinkStack *S)
{ LinkStack *p; int i;
  p=S->next; i=0;
  while(p!=NULL)
  { i++; p=p->next;}
  return i;
}
```

(4) 入栈操作

将值为 x 的数据元素插入栈 S，使 x 成为新的栈顶元素。

算法思路：先创建一个新结点，其数据域的值为 x，然后将该结点插入头结点之后作为栈顶结点。

```
int Push(LinkStack *S,ElemType x)
{ LinkStack *p;
  p=(LinkStack *)malloc(sizeof(LinkStack));     /*申请存储空间*/
  if(!p) return 0;                              /*若空间申请失败,则返回 0*/
  p->data=x;
  p->next=S->next;                              /*插入头结点之后*/
  S->next=p;
  return 1;
}
```

(5) 出栈操作

删除栈 S 的栈顶元素。

算法思路：若栈不为空，则先将栈顶结点的值送到指定的内存单元，然后删除栈顶结点。

```
int Pop(LinkStack *S,ElemType *e)
```

```
{ LinkStack *p;
  if(S->next==NULL) return 0;      /*栈空,删除失败,返回 0*/
  p=S->next;
  *e=p->data;                      /*栈顶结点值送到指针 e 所指向的单元*/
  S->next=p->next;
  free(p);
  return 1;                        /*出栈成功,返回 1*/
}
```

(6) 判栈空操作

判断栈 S 是否为空。

算法思路：栈空的条件是头结点的指针域为空。

```
int EmptyStack(LinkStack *S)
{ if(S->next==NULL) return 1;      /*栈空则返回 1,否则返回 0*/
  else return 0;
}
```

(7) 输出栈

输出栈 S 自栈顶到栈底的元素值。

其设计思路与单链表的输出操作相同。

```
void List(LinkStack *S)
{ LinkStack *p;
  p=S->next;
  while(p!=NULL)
  { printf("%4d ",p->data);
    p=p->next;
  }
  printf("\n");
}
```

【例 3.5】 编写算法，利用栈将带头结点的单链表逆置。

方法一：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 TurnLink1(LinkStack *S1,LinkStack *S2)
{ ElemType x;
  while(S1->next!=NULL)
  { Pop(S1,&x);
    Push(S2,x);
  }
}
```

方法二：将链表中的元素利用顺序栈交换顺序。

```
void TurnLink2(slink *head)
```

```
{ SeqStack S;
  slink *p;
  InitStack(&S);                    /*初始化空栈*/
  p=head->next;
  while(p)
  {Push(&S,p->data);p=p->next;}     /*链表中的元素依次入栈*/
  p=head->next;
  while(!EmptyStack(&S))
  {Pop(&S,&p->data);p=p->next;}     /*栈中的元素依次出栈到链表中的对应结点上*/
}
```

当然,该算法也可以使用链栈实现。

【例 3.6】 设计一个算法,判断一个字符串是否对称。若是,则返回 1,否则返回 0。

算法思路:先将长度为 len 的字符串的前半部分(str[0]～str[len/2－1])入栈,然后用后半部分(str[(len＋1)/2]～str[len－1])的字符依次与出栈的元素相比较;不相等时返回 0;若比较完毕且栈为空,则字符串对称。

```
typedef char ElemType;                  /*这里的 ElemType 类型设定为 char*/
int Symmetric(ElemType *str)
{ ElemType x,same=1;
  int len,i;
  LinkStack *S;
  S=InitStack();                        /*初始化空栈*/
  for(len=0;str[len]!='\0';len++);      /*计算字符串长度*/
  for(i=0;i<len/2;i++)
    Push(S,str[i]);                     /*字符串前半部分入栈*/
  for(i=(len+1)/2;i<len;i++)
  { Pop(S,&x);
    if(x!=str[i])                       /*对应字符比较*/
    { same=0;break;}
  }
  if(EmptyStack(S)&&same) return 1;     /*对称则返回 1,否则返回 0*/
  else return 0;
}
```

3.2 队　　列

3.2.1 队列的定义和基本操作

1. 队列的定义

队列(queue)也是一种特殊的线性表,它只允许在一端进行插入操作,在另一端进行删除操作。允许插入的一端称为队尾,允许删除的一端称为队头,新插入的结点只能添加到队尾,要删除的结点只能是排在队头的结点。在队列中插入结点的操作称为入队列,在

队列中删除结点的操作称为**出队列**。队列中含有的元素的个数称为**队列长度**，含有 0 个元素的队列称为**空队列**。图 3.4 是一个队列的示意图。

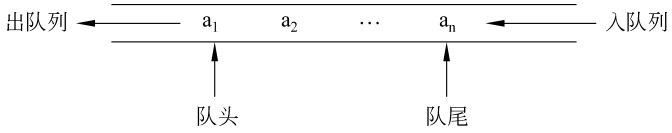


图 3.4　队列示意图

图 3.4 中的数据元素是按照 a_1，a_2，…，a_n 的顺序进入队列的，出队列的顺序也必须按照这个顺序，也就是说，只有在 $a_1 \sim a_{i-1}$（$2 \leqslant i \leqslant n$）都出队列后，$a_i$ 才能出队列。由此可知，队列的特点是先进先出（FIFO）或后进后出（LILO），因此队列又称先进先出的线性表。

2. 队列的基本操作

队列的基本操作主要有以下 7 种。

（1）初始化操作 InitQueue(Q)，用于初始化一个空队列 Q。

（2）求队列长度操作 GetLen(Q)，用于返回队列 Q 的元素个数，即队列的长度。

（3）取队头元素操作 GetFront(Q,e)，通过 e 带回队列 Q 的队头元素值。

（4）入队操作 EnQueue(Q,x)，用于将值为 x 的元素插入队列 Q，使 x 成为新的队尾元素。

（5）出队操作 OutQueue(Q,e)，用于删除队列 Q 中的队头元素，同时将队头元素值通过 e 带回，原队列中的第 2 个元素成为新的队头元素。

（6）判队空操作 EmptyQueue(Q)，用于判断队列 Q 是否为空，若队列为空，则返回 1，否则返回 0。

（7）输出队列操作 List(Q)，用于从队头到队尾依次输出队列 Q 中的所有元素的值。

队列也是一个线性表，其存储结构也分为顺序存储和链式存储两种，分别称为顺序队列和链队列。

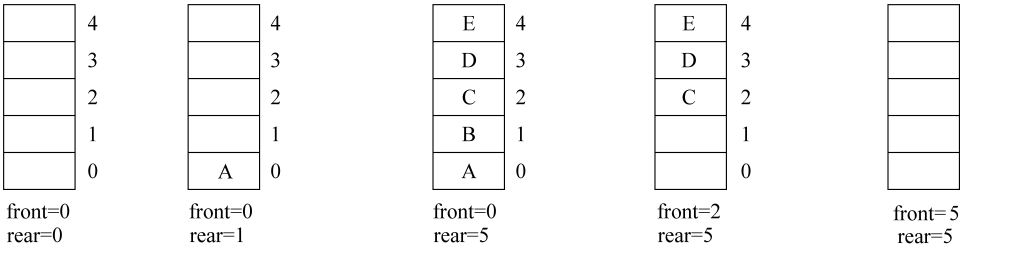
3.2.2　顺序队列表示及实现

1. 顺序队列

队列的顺序存储结构称为**顺序队列**，它利用一组地址连续的存储单元依次存放从队头到队尾的数据元素，同时利用两个变量分别记录当前队列中队头元素和队尾元素的位置，这两个变量分别称为**队头指针**和**队尾指针**。队头指针和队尾指针并不一定是指针变量，也可以是下标变量。在此，用下标变量描述队列。在初始化空队列时，队头指针和队尾指针的值都为 0；元素入队列后，队尾指针增 1；元素出队列后，队头指针增 1。因此在非空队列中，队头指针始终指向队头元素，队尾指针始终指向队尾元素的下一个单元（队尾元素下标＋1 处）。

假设用一维数组 Q[5]表示一个顺序队列，变量 front 和 rear 分别为队头指针和队尾指针。图 3.5 所示是这个顺序队列的几种状态。

图 3.5(a)表示空队列，front＝0，rear＝0，这也是初始化操作的结果。



(a) 空队列　(b) A入队列　(c) B、C、D、E入队列　(d) A、B出队列　(e) C、D、E出队列

图 3.5　顺序队列的几种状态

图 3.5(b)表示在图 3.5(a)的状态下元素 A 入队列后的状态，front＝0，rear＝1。

图 3.5(c)表示在图 3.5(b)的状态下元素 B、C、D、E 依次入队列后的状态，front＝0，rear＝5，此时队列满。

图 3.5(d)表示在图 3.5(c)的状态下元素 A、B 依次出队列后的状态，front＝2，rear＝5。

图 3.5(e)表示在图 3.5(d)的状态下元素 C、D、E 依次出队列后的状态，front＝5，rear＝5，此时队列为空。

从图 3.5 可以看到，图 3.5(c)为满队列状态，图 3.5(d)和图 3.5(e)也是满队列状态，但还有可利用的空间，只是不可以进行入队列操作，这种状态称为“假满”，此时进行入队列操作会产生“假溢出”现象。为了充分利用空间，解决假溢出现象，通常使用循环队列。

2. 循环队列

把顺序队列从逻辑上看成一个环，即当队尾指针或队头指针达到队列容量(MAXSIZE)时，再从下标为 0 的位置开始，这种队列称为**循环队列**。

假设用一维数组 Q[5]表示一个循环队列，变量 front 和 rear 分别为队头指针和队尾指针。图 3.6 所示是这个循环队列的几种状态。

图 3.6(a)表示空队列，front＝0，rear＝0，这也是初始化操作的结果。

图 3.6(b)表示在图 3.6(a)的状态下元素 A 入队列后的状态，front＝0，rear＝1。

图 3.6(c)表示在图 3.6(b)的状态下元素 B、C、D、E 依次入队列后的状态，front＝0，rear＝0，此时队列为满。

图 3.6(d)表示在图 3.6(c)的状态下元素 A、B、C 依次出队列后的状态，front＝3，rear＝0。

图 3.6(e)表示在图 3.6(d)的状态下元素 F、G 依次入队列后的状态，front＝3，rear＝2。

图 3.6(f)表示在图 3.6(e)的状态下元素 D、E、F、G 依次出队列后的状态，front＝2，rear＝2，此时队列为空。

在循环队列中，队头指针和队尾指针的后移可以利用取余运算实现，即

队头指针后移操作：

```
front=(front+1)%MAXSIZE;
```

队尾指针后移操作：

```
rear=(rear+1)%MAXSIZE;
```

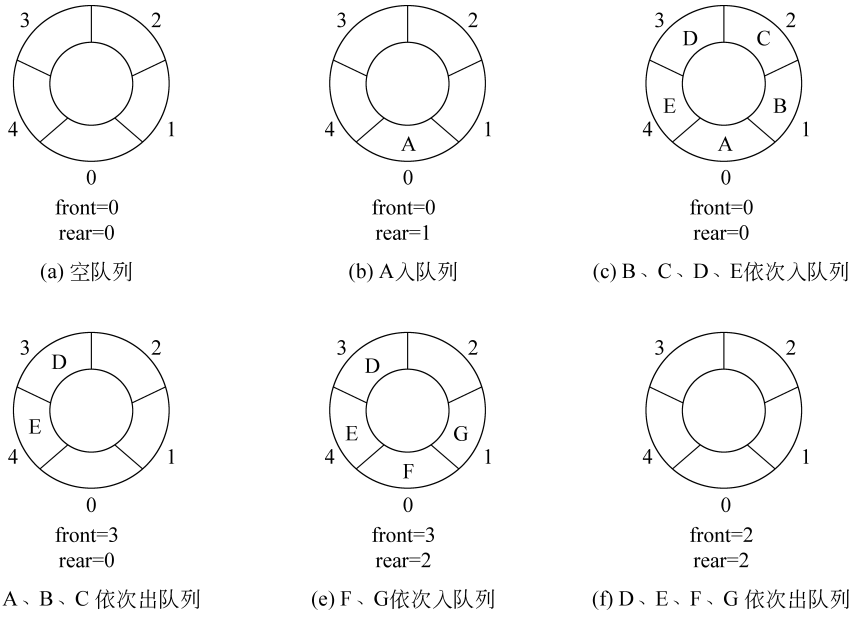


图 3.6　循环队列 Q 的几种状态

从图 3.6 可以看出，当循环队列为空时，有 front＝＝rear(见图 3.6(a)和图 3.6(f))，当循环队列为满时，也有 front＝＝rear(见图 3.6(c))，这就产生了二义性，显然是不合理的，因为无法判断循环队列究竟是空还是满。因此，为了区分循环队列的空或满状态，解决的方法之一是少用一个元素空间，规定当(rear＋1)％MAXSIZE＝＝front 时为循环队列满，即当队尾指针指向队头元素的上一个位置时就认为队列已满，如图 3.6(e)所示。当然，也可另设标志以区分循环队列的空和满。

循环队列的类型定义如下：

```
#define MAXSIZE 100              /*队列空间大小*/
typedef int ElemType;            /*数据元素类型*/
typedef struct
{ ElemType *base;                /*基地址*/
  int front;                     /*队头指针*/
  int rear;                      /*队尾指针*/
}CirQueue;
```

其中，MAXSIZE 是循环队列容量；base 是队列空间的起始地址(基地址)；front 是队头指针，指向队头元素，其初值为 0；rear 是队尾指针，指向队尾元素的下一个单元，其初值也为 0。

3. 循环队列基本操作的实现

(1) 初始化操作

创建一个空循环队列 Q。

算法思路：申请存储空间，并将队头指针和队尾指针都初始化为 0。

```
void InitQueue(CirQueue *Q)
{ Q->base=(ElemType *)malloc(MAXSIZE*sizeof(ElemType));     /*分配内存*/
  Q->front=Q->rear=0;                /*队头指针和队尾指针初始值都为 0*/
}
```

(2) 求队列长操作

计算循环队列中数据元素的个数。

算法思路：根据循环队列的特点，有以下几种情况：

$$\text{队列长度}=\begin{cases}0 & (\text{front}==\text{rear})\\ \text{rear}-\text{front} & (\text{rear}>\text{front})\\ \text{rear}+\text{MAXSIZE}-\text{front} & (\text{rear}<\text{front})\end{cases}$$

归纳起来，循环队列长度的计算公式为

$$\text{队列长度}=(\text{rear}+\text{MAXSIZE}-\text{front})\%\text{MAXSIZE}$$

```
int GetLen(CirQueue *Q)
{ return((Q->rear-Q->front+MAXSIZE)%MAXSIZE);
}
```

(3) 取队头元素操作

取出循环队列 Q 的队头元素值。

算法思路：先判断队列是否为空，若不为空，则将队头元素值送到指定的内存单元。

```
int GetFront(CirQueue *Q,ElemType *e)
{ if(Q->front==Q->rear) return 0;     /*队空,返回 0*/
  *e=Q->base[Q->front];               /*取队头元素*/
  return 1;
}
```

(4) 入队列操作

在循环队列 Q 的队尾插入值为 x 的元素。

算法思路：先判断队列是否为满，若不为满，则先在队尾指针处存放 x，然后将队尾指针后移一个位置。

```
int EnQueue(CirQueue *Q,ElemType x)
{ if((Q->rear+1)%MAXSIZE==Q->front) return 0;    /*队满,返回 0*/
  Q->base[Q->rear]=x;                            /*入队列*/
  Q->rear=(Q->rear+1)%MAXSIZE;                   /*修改队尾指针*/
  return 1;
}
```

(5) 出队列操作

将循环队列 Q 的队头元素删除。

算法思路：先判断队列是否为空，若不为空，则先将队头元素值送到指定的内存单元，然后将队头指针后移一个位置。

```
int OutQueue(CirQueue *Q,ElemType *e)
{ if(Q->front==Q->rear) return 0;          /*队空,返回 0*/
  *e=Q->base[Q->front];                    /*取队头元素值*/
  Q->front=(Q->front+1)%MAXSIZE;           /*修改队头指针*/
  return 1;
}
```

(6) 判队空操作

判断循环队列 Q 是否为空。

算法思路：循环队列 Q 为空的条件是 Q->front==Q->rear。

```
int EmptyQueue(CirQueue *Q)
{ if(Q->front==Q->rear) return 1;
  else return 0;
}
```

(7) 输出操作

输出循环队列 Q 从队头到队尾的所有元素值。

算法思路：依次输出下标自 front 至 rear-1 的元素值，修改循环变量 i 的方法是 i=(i+1) %MAXSIZE。

```
void List(CirQueue *Q)
{ int i;
  i=Q->front;
  while(i!=Q->rear)
  { printf("%4d ",Q->base[i]);
    i=(i+1)%MAXSIZE;
  }
}
```

【例 3.7】 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；当 $a_i=0$ 时算法结束。要求利用循环队列完成，并在异常情况下(如队空)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；若小于 0，则出队列一个元素并输出；若等于 0，则退出循环。

```
void Fun(void)
{ CirQueue Q;ElemType x;
  InitQueue(&Q);
```

```
  printf("输入数据(0:结束):");
  scanf("%d",&x);
  while(x!=0)        /* 当输入 0 时退出循环,同时算法结束 */
  { if(x>0)
    { if(!EnQueue(&Q,x))
      { printf("队列满!\n");break;}
    }
    else if(!OutQueue(&Q,&x))
    { printf("队列空!\n");break;}
    scanf("%d",&x);
  }
}
```

3.2.3 链队列表示及实现

1. 链队列

队列的链式存储结构称为**链队列**。链队列实际上是一个带头指针和尾指针的单链表,这种单链表的第一个结点称为队头结点,最后一个结点称为队尾结点,尾指针始终指向队尾结点。在带头结点的链队列中,头指针指向头结点;在不带头结点的链队列中,头指针指向队头结点。图 3.7 是一个带头结点的链队列的示意图。

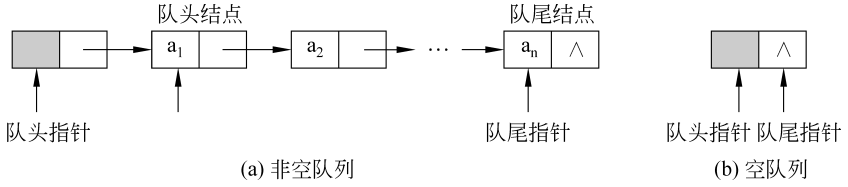


图 3.7 带头结点的链队列示意图

链队列的类型定义如下:

```
typedef int ElemType;              /* 数据元素类型 */
typedef struct node
{ ElemType data;                   /* 数据域 */
  struct node *next;               /* 指针域 */
}qlink;                            /* 结点类型定义 */
typedef struct
{ qlink *front;                    /* 队头指针 */
  qlink *rear;                     /* 队尾指针 */
}LinkQueue;                        /* 链队列类型定义 */
```

2. 链队列基本操作的实现

下面介绍带头结点的链队列的基本操作的实现。

(1) 初始化操作

建立空的链队列 Q。

算法思路：创建头结点，队头指针和队尾指针都指向它，并将其指针域置为空（NULL）。

```
void InitQueue(LinkQueue *Q)
{ Q->front=Q->rear=(qlink *)malloc(sizeof(qlink)); /*创建头结点*/
  Q->front->next=NULL;                              /*头结点指针域置为空*/
}
```

（2）求队列长度操作

计算链队列 Q 中数据元素的个数。

其设计思路与单链表的求表长操作相同。

```
int GetLen(LinkQueue *Q)
{ int i;qlink *p;
  p=Q->front->next;i=0;
  while(p!=NULL)
  { i++;p=p->next;}
  return i;
}
```

（3）判队空操作

判断链队列 Q 是否为空。

算法思路：链队列 Q 为空的条件是 Q－＞front＝＝Q－＞rear 或 Q－＞front－＞next＝＝NULL。

```
int EmptyQueue(LinkQueue *Q)
{ if(Q->front==Q->rear) return 1;      /*队空的条件为 front==rear*/
  else return 0;
}
```

（4）取队头元素操作

取出链队列 Q 的队头元素值。

算法思路：若链队列不为空，则将队头结点数据域值送到指定的内存单元。

```
int GetFront(LinkQueue *Q,ElemType *e)
{ if(Q->front==Q->rear) return 0;      /*队空,返回 0*/
  *e=Q->front->next->data;
  return 1;
}
```

（5）入队列操作

在队列 Q 中插入一个值为 x 的结点，使之成为新的队尾结点。

算法思路：先创建新结点，数据域值为 x，指针域值为空（NULL），然后将新结点插入尾结点之后，最后修改队尾指针，使其指向新插入的结点。

```
void EnQueue(LinkQueue *Q,ElemType x)
```

```
{ qlink *p;
  p=(qlink *)malloc(sizeof(qlink));     /*创建新结点*/
  p->data=x;
  p->next=NULL;
  Q->rear->next=p;                      /*插入链队列的尾部*/
  Q->rear=p;
}
```

(6) 出队操作

删除链队列中的队头结点。

算法思路:若链队列不为空,则先将队头结点数据域值送到指定的内存单元,然后删除队头结点。若删除结点为队尾结点,则需要修改队尾指针。

```
int OutQueue(LinkQueue *Q,ElemType *e)
{ qlink *p;
  if(Q->front==Q->rear) return 0;   /*队空,返回 0*/
  p=Q->front->next;
  *e=p->data;
  Q->front->next=p->next;
  if(Q->rear==p)                    /*当队中只有一个元素时,出队后应重新修改队尾
                                      指针 rear*/
    Q->rear=Q->front;
  free(p);
  return 1;
}
```

(7) 输出队列操作

输出链队列从队头结点至队尾结点的数据域值。

其设计思路与单链表的输出操作相同。

```
void List(LinkQueue *Q)
{ qlink *p;
  p=Q->front->next;
  while(p!=NULL)
  { printf("%4d",p->data);
    p=p->next;
  }
}
```

3.3 串

3.3.1 串的定义和基本操作

1. 串的定义

串(string)又称字符串,是一种特殊的线性表,其特殊性体现在表中的每个数据元素

都是一个字符，即**串**是由 0 个或多个字符组成的有限序列。一般记为

$$S=\text{"}a_1a_2a_3\cdots a_n\text{"}\quad(n\geqslant 0)$$

其中，S 是**串名**，用双引号括起来的字符序列是**串值**，双引号本身不是串的内容，是串的定界符。$a_i(1\leqslant i\leqslant n)$代表一个字符，可以是字母、数字或其他字符。串中的字符个数 n 称为**串长**，含有 0 个字符的串称为**空串**。

串中任意连续的字符所组成的子序列称为该串的**子串**，包含子串的串称为**主串**。字符在串中的位序称为该字符在串中的位置。子串在主串中的位置是以子串的第一个字符在主串中的位置表示的。若两个串的长度相等，并且各个对应位置上的字符都相同，则称这两个串**相等**。假设 S1、S2、S3 为如下三个串：

```
S1="data",S2="structure",S3="data structure"
```

则它们的长度分别为 4、9、14，并且 S1、S2 是 S3 的子串，S1 在 S3 中的位置是 1，S2 在 S3 中的位置是 6。

2. 串的基本操作

串的基本操作主要有以下 12 种。

(1) 初始化操作 InitString(s)，用于初始化一个空串 s。

(2) 串赋值操作 StrAssign(s1,s2)，用于将串常量 s2 赋给串变量 s1。

(3) 串复制操作 Assign(s1,s2)，用于将串变量 s2 的值赋给串变量 s1。

(4) 求串长操作 Length(s)，用于返回串 s 的长度。

(5) 判串等操作 Equal(s,t)，用于判断串 s 和 t 的值是否相等，若相等则返回 1，否则返回 0。

(6) 串连接操作 Concat(s,s1,s2)，用于将串 s2 连接到串 s1 的后面，结果存到串 s 中。

(7) 取子串操作 SubStr(s,i,j,t)，用于将串 s 的从第 i 个字符开始的 j 个连续字符存到 t 中。在此，1≤i≤Length(s)，0≤j≤Length(s)－i＋1。

(8) 插入操作 Insert(s,i,t)，用于将串 t 插入串 s 的第 i 个字符之前。在此，1≤i≤Length(s)＋1。

(9) 删除操作 Delete(s,i,j)，用于在串 s 中删除从第 i 个字符开始的长度为 j 的子串。在此，1≤i≤Length(s)，1≤j≤Length(s)－i＋1。

(10) 串查找操作 Index(s,t,pos)，用于从串 s 的第 pos 个字符开始查找串 t 首次出现的位置。在此，1≤pos≤Length(s)。该操作也称为串的模式匹配，将在 3.3.4 节具体介绍。

(11) 替换操作 Replace(s,i,j,t)，用于将串 s 的从第 i 个字符开始的连续 j 个字符替换成串 t。在此，1≤i≤Length(s)，1≤j≤Length(s)－i＋1。

(12) 输出操作 List(s)，用于输出串 s 的值。

串也是线性表，也有顺序存储和链式存储这两种存储结构，分别称为顺序串和链串。

3.3.2 顺序串表示及实现

1. 顺序串

串的顺序存储结构称为**顺序串**,即串中的字符被依次存放在一组连续的存储单元中。一般来说,一个字符占用 1 字节的存储空间,因此一个存储单元可以存储多个字符。例如,一个 32 位的内存单元可以存储 4 个字符。串的顺序存储有两种格式:一种是每个存储单元只存放一个字符,称为**非紧缩格式**;另一种是每个存储单元存放多个字符,称为**紧缩格式**。假设一个存储单元占 4 字节,存放的起始地址为 i,则串"DATA STRUCTURE"的非紧缩格式存储和紧缩格式存储如图 3.8 和图 3.9 所示。

i	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8	i+9	i+a	i+b	i+c	i+d
D	A	T	A		S	T	R	U	C	T	U	R	E

图 3.8 非紧缩格式示例

i	i+1	i+2	i+3
D		U	R
A	S	C	E
T	T	T	
A	R	U	

图 3.9 紧缩格式示例

串的顺序存储还有两种方式:串的定长表示和串的顺序表表示。所谓串的定长表示就是指预先定义一个固定的字符数组空间,例如:

```
#define MAXSIZE 256
char string[MAXSIZE];
```

其中,0 号单元用于存放串长,因此这种表示法能存放的字符个数至多为 255 个,并且空间不能扩充,目前基本不采用此法。由于串长无法预测,因此采用动态分配的一维数组存储串值,即串的顺序表表示。下面仅介绍顺序表表示方法,读者可以自行用定长表示法完成串的一些基本操作。顺序串的类型定义如下:

```
#define INITSIZE 100          /* 为串分配的存储空间的初始量 */
typedef struct
{ char *ch;                   /* 串存放的起始地址 */
  int length;                 /* 串长 */
  int strsize;                /* 当前为串分配的存储空间容量 */
}SeqStr;
```

其中,INITSIZE 是为串分配的存储空间的初始量;ch 是串存放的起始地址,串中的第 i 个字符存储在 ch[i-1]中;length 是串长,最后一个字符的下标为 length-1;strsize 是当前分配的存储空间容量,如果在操作过程中存储空间不足,则可以用函数 realloc()进行再分配,为顺序串增加存储空间。

2. 顺序串基本操作的实现

(1) 初始化操作

创建一个空串 s。