# 第3章　限定性线性结构——栈和队列

前面讨论了两种基本的线性表——顺序表和链表的存储结构，下面讨论两种重要的操作受限的线性表——栈和队列。

## 3.1　栈

### 3.1.1　栈的类型定义

栈(Stack)是限定只能在表的一端(表尾)进行插入或删除操作的线性表。允许进行插入或删除的这一端称为栈顶(Top)；另一端则称栈底(Bottom)，不能进行插入或删除。当栈中没有包含数据元素时，称为空栈。栈非空时，处于栈顶位置的元素称为栈顶元素。向一个栈插入新的元素称为入栈或进栈(Push)，此时，插入的元素成为新的栈顶元素；从栈中删除一个元素时，只能删除当前的栈顶元素，称为出栈或退栈(Pop)。

由于栈的插入和删除只能在栈顶进行，最先入栈的元素必定最后出栈，最后入栈的元素最先出栈，因此栈又叫做后进先出(Last In First Out，LIFO)线性表。

现实生活中可以看到许多后进先出的实例。例如，货运列车的编组，为了卸装的方便，总是将最后到达的车厢编在列车的最前端，最先到达的车厢编在最后面，运行过程中每当到达一个站点，就从最后卸下一个车厢，以此类推，直到终点将所有车厢全部卸装；再如弹夹的装弹和卸弹过程，最先装入的被压到最底下，最后装入的最先被击发。这些实例与栈的结构和原理完全一致。

下面引出栈的类型定义。假设线性表

$$S = (a_1, a_2, \cdots, a_i, \cdots, a_n) \tag{3-1}$$

是一个栈，则栈中 $a_1$ 为栈底元素，$a_n$ 为栈顶元素。栈的结构示意图如图3-1所示。

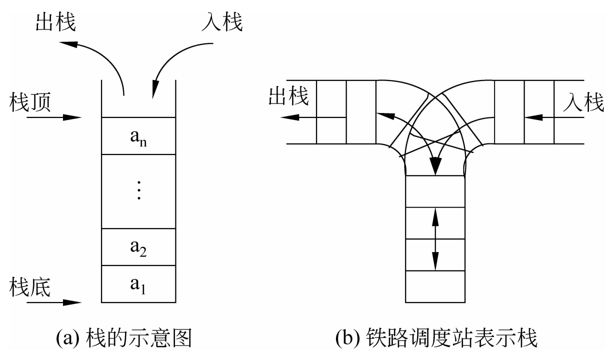


图3-1　栈的结构示意图

栈的基本操作有初始化、入栈、出栈、判空及取栈顶元素等。栈的抽象数据类型定义形式如下：

```
ADT Stack{
    数据对象：D={aᵢ|aᵢ∈ElemSet,i=1,2,…,n,n≥0}
    数据关系：R={⟨aᵢ₋₁,aᵢ⟩|aᵢ₋₁,aᵢ∈D,i=1,2,…,n}
              约定：aₙ 端为栈顶,a₁ 端为栈底
    基本操作：
      InitStack(S)
        操作结果：构造一个空的栈 S。
      DestroyStack(S)
        初始条件：栈 S 已经存在。
        操作结果：将栈 S 销毁。
      ClearStack(S)
        初始条件：栈 S 线性表 L 已经存在。
        操作结果：将栈 S 线性表 L 清空(重置为空表)。
      StackEmpty(S)
        初始条件：栈 S 已经存在。
        操作结果：判断栈 S 是否为空,若为空栈,返回 TRUE,否则返回 FALSE。
      StackLength(S)
        初始条件：栈 S 已经存在。
        操作结果：求栈 S 的长度(返回栈中的数据元素个数)。
      GetTop(S,e)
        初始条件：栈 S 已经存在。
        操作结果：返回栈 S 中栈顶元素的值(赋予变量 e)。
      Push(S,e)
        初始条件：栈 S 已经存在。
        操作结果：在栈 S 插入一个新元素 e,使之成为新的栈顶元素。
      Pop(S,e)
        初始条件：栈 S 已经存在且非空。
        操作结果：删除栈 S 的栈顶元素,其值赋予 e。
    ⋮
}ADT Stack
```

栈既然是一种线性表,因而线性表的顺序存储结构和链式存储结构同样适用于栈的表示和实现,分别称为顺序栈和链栈。

### 3.1.2 顺序栈的表示和实现

与顺序表类似,顺序栈就是用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。同时,为了指示当前的栈顶元素位置,需另设一个指针变量 top,称为栈顶指针,通常 top 指向栈中下一个入栈位置,当栈顶指针指向第一个存储单元时表示空栈。入栈时,首先将新元素的值存入当前的栈顶位置,然后使栈顶指针 top＋1;出栈时则相反,先

使栈顶指针 top－1,然后取出当前位置的元素的值。顺序栈的结构及基本操作如图 3-2 所示。

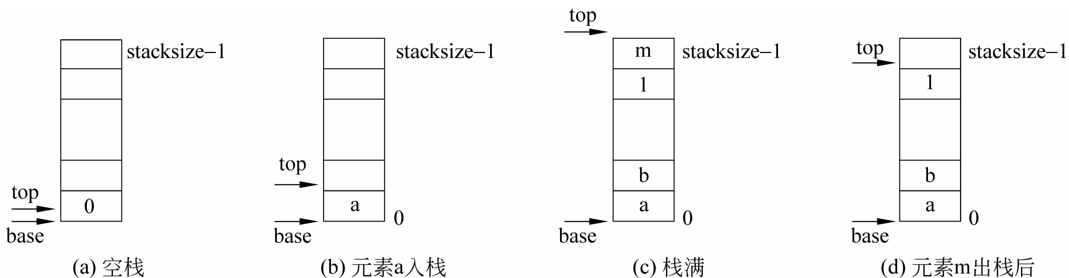


**图 3-2　顺序栈的结构及基本操作**

顺序栈的实现方式可以采用预设的固定长度的一维数组来实现,也可以通过预定义两个常量 INIT_SIZE 和 INCREMENT 分别表示顺序栈的初始长度和增量,以动态分配的顺序存储结构进行描述。下面以动态分配的存储表示说明顺序栈及其基本操作:

```
#define ERROR 0
#define OK 1
#define STACK_INT_SIZE 10                     /* 存储空间初始分配量 */
#define STACKINCREMENT 5                      /* 存储空间分配增量 */
typedef struct
{
    ElemType * base;
    ElemType * top;
    int stacksize;                            /* 当前已分配的存储空间 */
}SqStack;
```

在此基础上,即可实现顺序栈的各种基本操作。

**1. 顺序栈的初始化**

首先申请分配栈空间,然后初始化栈顶指针。

**【算法 3.1】** 顺序栈的初始化。

```
int InitStack(SqStack * S)
{
    S->base=(ElemType *)malloc(STACK_INT_SIZE * sizeof(ElemType));
    if(!S->base) return ERROR;                /* 申请空间失败 */
    S->top=S->base;                           /* 置为空栈 */
    S->stacksize=STACK_INT_SIZE;              /* 设置当前栈空间大小 */
    return OK;
}
```

**2. 判断是否空栈**

通过判断栈顶指针的位置,确定栈是否为空。判断空栈的条件为 S－>top＝＝

S－＞base。

【算法 3.2】 判断顺序栈是否为空。

```
int EmptyStack(SqStack *S)
{
    if(S->top==S->base) return OK;
    else return ERROR;
}
```

**3. 入栈**

在当前栈顶插入一个新的元素，并将栈顶指针加1。元素入栈必须判断当前是否栈满。若栈空间已满，需要重新申请增加存储空间。判断栈满条件为 S－＞top－S－＞base＞＝S－＞stacksize。

【算法 3.3】 入栈。

```
int Push(SqStack *S,ElemType e)
{
    if(S->top-S->base>=S->stacksize)    /*判断当前栈是否满*/
    {
        S->base=(ElemType *)realloc(S->base,(S->stacksize+STACKINCREMENT)
        *sizeof(ElemType));

        if(!S->base) return ERROR;          /*申请失败,结束入栈操作*/
        S->top=S->base+S->stacksize;        /*重新设置当前栈顶指针*/
        S->stacksize+=STACKINCREMENT;
    }
    *S->top++=e;                            /*新元素入栈,栈顶指针加1*/
    return OK;
}
```

**4. 出栈**

将当前栈顶指针减1，所指元素可根据需要确定是否获取。出栈必须判断当前栈是否为空，若为空栈，则无法做出栈操作。

【算法 3.4】 出栈。

```
int Pop(SqStack *S,ElemType *e)
{
    if(S->top==S->base) return ERROR;   /*若栈空,则返回失败*/
    --S->top;                           /*修改栈顶指针减1*/
    *e=*S->top;                         /*e返回出栈元素的值*/
    return OK;
}
```

**5. 取栈顶元素**

只获取当前栈顶元素的值，不做出栈操作，不修改栈顶指针。取栈顶元素同出栈一样，需要判断当前栈是否为空。

**【算法 3.5】** 取栈顶元素的值。

```
int GetTop(SqStack *S,ElemType *e)
{
    if(S->base==S->top) return ERROR;    /*若栈空,则返回失败*/
     *e=*(S->top-1);                     /*只取栈顶元素值,不修改栈顶指针*/
     return OK;
}
```

**说明：**

（1）对于顺序栈，入栈时，必须判定栈是否满。栈满时，若顺序栈采用静态的存储表示，则不能再做入栈操作；若采用上述的动态可变长的存储表示，则需要先申请增加存储空间，再入栈。

（2）出栈和取栈顶元素操作，必须判定栈是否为空，栈空时不能进行出栈或取栈顶元素操作，否则产生错误。

### 3.1.3 链栈的表示和实现

链栈就是采用链式存储结构实现的栈。通常链栈用单链表表示，因此其结点结构与单链表相似，可用 StackNode 表示：

```
typedef struct StackNode
{
    ElemType data;
    struct StackNode *next;
}StackNode,*LinkStack;
```

因为栈中的主要运算是在栈顶插入和删除，显然以链表的头部做栈顶是最方便的，也无须附加头结点。通常将链栈表示成如图 3-3 所示的形式。

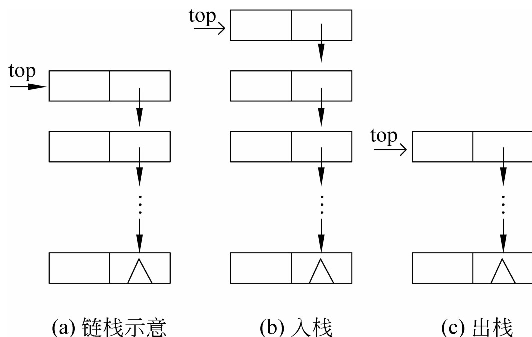


**图 3-3 链栈的结构及基本操作**

链栈基本操作的实现如下：

**1. 链栈的初始化**

链栈的初始化即设置栈顶指针为空。

**【算法 3.6】** 链栈的初始化。

```
LinkStack InitStack_L(LinkStack top)
{
    top=NULL;
    return top;
}
```

**2. 判断是否空栈**

若栈顶指针为空即为空栈，判断条件为 top==NULL。

**【算法 3.7】** 链栈判空。

```
int EmptyStack_L(LinkStack top)
{
    if(!top) return OK;
    else return ERROR;
}
```

**3. 入栈**

申请新结点，链接到栈顶，修改栈顶指针 top 指向新入栈结点。

**【算法 3.8】** 入栈。

```
LinkStack PushStack_L(LinkStack top,ElemType e)
{
    LinkStack p;
    p=(StackNode *)malloc(sizeof(StackNode));   /*申请结点空间*/
    if(!p) return top;                          /*申请空间失败,返回原栈顶指针*/
    p->data=e;
    p->next=top;                                /*结点链接在栈顶*/
    top=p;                                      /*修改栈顶指针指向新结点*/
    return top;                                 /*返回栈顶指针*/
}
```

**4. 出栈**

判断栈是否为空，不为空，取栈顶元素，并修改栈顶指针，指向下一个元素。

**【算法 3.9】** 出栈。

```
LinkStack PopStack_L(LinkStack top,ElemType *e)
{
    LinkStack p;
```

```
    if(!top) return NULL;                          /*栈空,返回空*/
    else
    {
        p=top;
        *e=top->data;                              /*e返回栈顶元素值*/
        top=top->next;                             /*修改栈顶指针*/
        free(p);                                   /*释放原栈顶元素空间*/
    }
    return top;                                    /*返回当前栈顶指针*/
}
```

其他操作算法略。

## 3.2 队列

和栈一样,队列也是一种操作受限的线性表,以下作详细介绍。

### 3.2.1 队列的类型定义

队列(Queue)是限定只能在表的两端分别进行插入或删除操作的线性表。在队列结构中,数据元素只能从一端(表尾)插入,从另一端(表头)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)。当队列中没有包含数据元素时,称为空队。向一个队列插入新的元素称为入队(Enqueue),此时,插入的元素成为新的队尾元素;从队列中删除一个元素时,只能删除当前的队头元素,称为出队(Dequeue)。基于队列的这种“先进先出”的结构特点,因而也称为先进先出(First In First Out,FIFO)线性表。

在实际生活中,有关队列的例子也很多。各种排队现象如排队买票、排队购物、排队上车等,处于队头的首先出队,而新来的总是排到队尾。队列的这种先进先出的特性也反映了现实生活中先来先服务的处理原则。

下面给出队列的类型定义。假设线性表

$$Q = (a_1, a_2, \cdots, a_i, \cdots, a_n) \qquad (3\text{-}2)$$

是一个队列,则队列中 $a_1$ 为队头元素,$a_n$ 为队尾元素。队列的结构如图 3-4 所示。

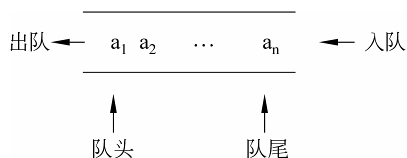


图 3-4 队列的结构示意图

队列的抽象数据类型定义形式如下:

```
ADT Queue{
    数据对象:D={a_i|a_i∈ElemSet,i=1,2,…,n,n≥0}
    数据关系:R={〈a_{i-1},a_i〉|a_{i-1},a_i∈D,i=1,2,…,n}
              约定:a_n 端为队尾,a_1 端为队头。
    基本操作:
      InitQueue(Q)
        初始条件:队列 Q 不存在。
```

```
        操作结果:构造了一个空队列 Q。
      EnQueue(Q,x)
        初始条件:队列 Q 存在。
        操作结果:对已存在的队列 Q,插入一个元素 x 到队尾,队列发生变化。
      DeQueue(Q,x)
        初始条件:队列 Q 存在且非空。
        操作结果:删除队首元素,并返回其值,队列发生变化。
      GetHead(Q,x)
        初始条件:队列 Q 存在且非空。
        操作结果:读队头元素,并返回其值,队列不变。
      QueueEmpty(Q)
        初始条件:队列 Q 存在。
        操作结果:若 Q 为空队列则返回 TRUE,否则返回 FALSE。
      ⋮
}ADT Queue
```

队列的表示和实现也可以采用顺序存储结构和链式存储结构加以描述,以下分别说明。

### 3.2.2 顺序队列的表示和实现

采用顺序存储结构表示队列时,可以用C语言中的一维数组进行描述。由于对于队列的操作只在队头和队尾进行,因此,需要设置两个指针 front 和 rear 分别指示队头和队尾的位置,并可约定队尾指针 rear 指向当前队尾元素后的下一个位置,队头指针 front 指向当前的队头元素。顺序队列的表示形式如下:

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

(1) 初始化队列。队尾指针和队头指针均指向下标为0的单元,令 front=rear=0。

(2) 入队操作。新元素将存放到当前队尾指针所指的单元,并使队尾指针 rear 增1,指示下一次插入的位置,则 rear=rear+1。

(3) 出队操作。将当前队头指针所指单元的值返回,并将队头指针 front 增1即可,则 front=front+1。

入队和出队操作的过程如图3-5所示。可以看出,队列为空,不能做出队操作,因此判断队列为空的条件是 front==rear;经过多次插入和删除操作以后,队列的队尾指针和队头指针将逐步向后移动,最终出现当队尾指针已指向所定义空间中的最后单元时,即 rear=MAXSIZE-1 时,队列满,无法再入队。虽然此时队头指针所指位置之前可能存在若干单元空闲,却无法进行入队操作的问题。这种现象称为“假溢出”。

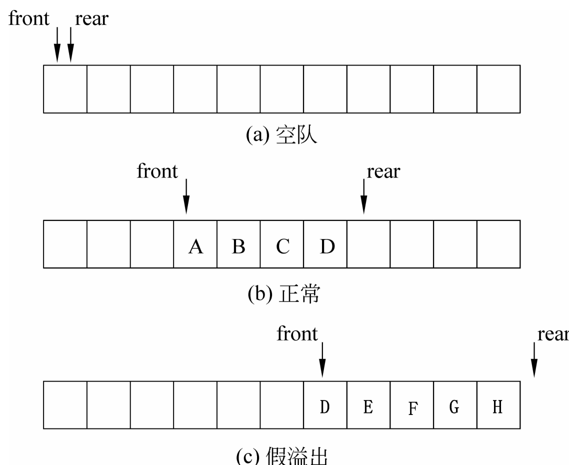


**图 3-5　顺序队列的入队和出队操作的过程**

可以借助两种方法解决顺序队列的“假溢出”问题。一是采用“移动队列”的方法，即每当执行一次出队操作，则依次将队头和队尾指针向数组的起始位置移动，始终保持队头在数组的起始位置。这种方法的代价是产生大量的元素移动，显然不是一种好方法。

更合理有效的解决方法是，将一维数组的最后一个单元和第一个单元连接起来构成循环数组，此时称为“循环队列”。当队尾指针已指向数组的最后时，在进行入队操作过程中，可将队尾指针 rear 移至数组的起始位置，表示下一次入队操作时的队尾。队尾指针移动的方式是 rear＝(rear＋1)％MAXSIZE，队头指针的移动也一样 front＝(front＋1)％MAXSIZE，如图 3-6 所示。

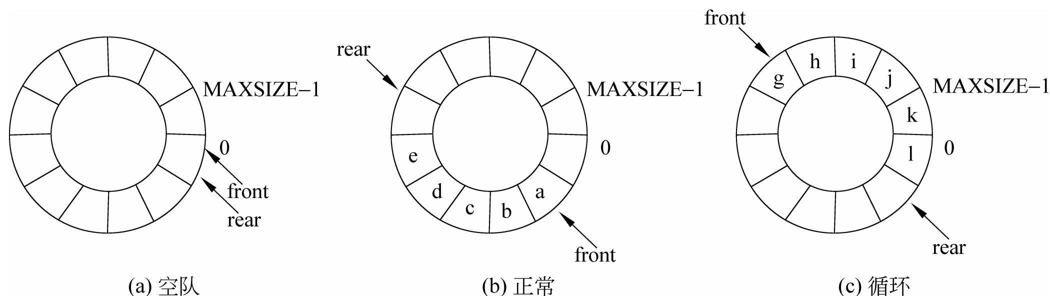


**图 3-6　循环队列的入队和出队操作的过程**

循环队列初始化空队时，front＝rear＝0；当队列经过多次出队入队操作后，会出现队头指针 front 和队尾指针 rear 再次指向同一单元的情况，此时有可能队列空也有可能队列满。为了区分队满和队空的情况，规定当队空时，front＝＝rear；队满时，队尾指针指向队头指针的前一个位置即为队满(rear＋1)％MAXSIZE＝＝front，实际循环队列队满时，所容纳的元素个数为 MAXSIZE－1；通过空留一个存储单元的方式区分循环队列队满和队空的情况。下面详细说明循环队列的存储表示和基本操作的实现方法。

循环队列的顺序存储表示如下：

```
#define MAXQSIZE 100
typedef struct
{
    ElemType * base;                          /* 指向循环队列存储空间 */
    int front;                                /* 队头指针 */
    int rear;                                 /* 队尾指针 */
}SqQueue;
```

以下说明循环队列的各种基本操作。

### 1. 初始化队列

**【算法 3.10】** 循环队列初始化。

```
int InitQueue(SqQueue * Q)
{
    Q->base=(ElemType * )malloc(MAXQSIZE*sizeof(ElemType));
    if(!Q->base)                              /* 申请存储空间失败 */
        return ERROR;
    Q->front=Q->rear=0;                       /* 设置队头队尾指针 */
    return OK;
}
```

### 2. 入队

入队操作需要判断循环队列是否已满。当队列不满,新元素入队,修改队尾指针指向下一个存储单元。

**【算法 3.11】** 入队。

```
int EnQueue(SqQueue * Q,ElemType e)
{
    if((Q->rear+1)%MAXQSIZE==Q->front)        /* 判断队列是否已满 */
        return ERROR;
    Q->base[Q->rear]=e;                       /* 元素 e 入队 */
    Q->rear=(Q->rear+1)%MAXQSIZE;             /* 修改队尾指针 */
    return OK;
}
```

### 3. 出队

出队操作需要判断循环队列是否为空。当队列不空时,队头元素出队,修改队头指针指向下一个元素。

**【算法 3.12】** 出队。

```
int DeQueue(SqQueue * Q,ElemType * e)
{
    if(Q->front==Q->rear)                     /* 判断队列是否为空 */
```

```
        return ERROR;
    *e=Q->base[Q->front];                    /*e返回队头元素值*/
    Q->front=(Q->front+1)%MAXQSIZE;          /*修改队头指针*/
    return OK;
}
```

**4. 判断循环队列是否为空**

**【算法 3.13】** 循环队列判空。

```
int QueueEmpty(SqQueue *Q)
{
    if(Q->front==Q->rear)
        return OK;
    else
        return ERROR;
}
```

## 3.2.3 链队的表示和实现

采用链式存储结构表示队列,称为链队。显然,利用与线性链表类似的方法可以很容易实现链队结构,并分别设置队尾和队头指针指示链队的位置。在链队结构中增加一个附加的头结点,并使队头指针 front 指向它。此时,判断空队的标志是,链队的队尾指针 rear 和队头指针 front 均指向头结点。带头结点的链队结构示意图如图 3-7 所示。

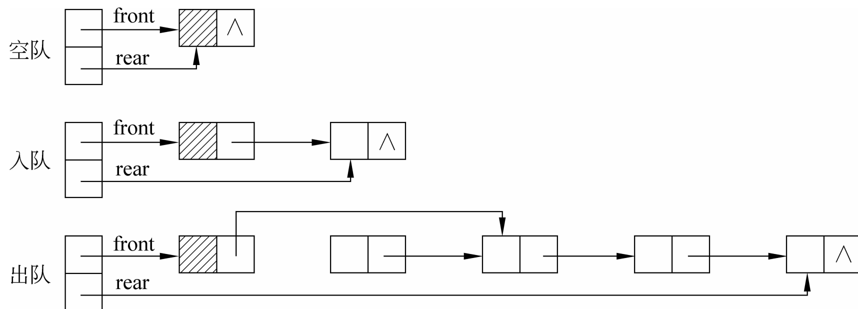


图 3-7 带头结点的链队结构示意

链队的操作十分简单,只需要根据实际操作情况修改相应的队头指针或队尾指针的指向即可。链队的表示和实现形式如下:

```
typedef struct LinkNode          /*链队中结点的类型*/
{
    ElemType data;
    struct LinkNode *next;
}LinkNode;
typedef struct LinkQueue         /*将头尾指针封装在一起的链队*/
```

```
{
    LinkNode * front, * rear;
}LinkQueue;
```

在进行链队的删除(出队)操作时,需要注意的是,如果删除的结点是当前队列的唯一结点时,除了队列的头结点的指针作相应修改外,队尾指针也必须重新赋值为队列的头结点,否则将导致队尾指针的丢失。具体操作算法实现如下:

### 1. 创建一个带头结点的空队

申请头结点存储空间,并设置队头队尾指针指向头结点。

**【算法3.14】** 链队初始化。

```
int InitQueue_L(LinkQueue * Q)
{
    LinkNode * p;
    p=(LinkNode * )malloc(sizeof(LinkNode));    /* 申请附加的头结点空间 */
    if(!p)
        return ERROR;
    p->next=NULL;
    Q->front=Q->rear=p;
    return OK;
}
```

### 2. 判断队列为空

**【算法3.15】** 链队判空。

```
int EmptyQueue_L(LinkQueue * Q)
{
    if(Q->front==Q->rear) return OK;            /* 判断队列为空 */
    else return ERROR;
}
```

### 3. 入队

**【算法3.16】** 链队入队。

```
int EnQueue_L(LinkQueue * Q,ElemType e)
{
    LinkNode * p;
    p=(LinkNode * )malloc(sizeof(LinkNode));    /* 申请新结点 */
    if(!p) return ERROR;
    p->data=e;
    p->next=NULL;
    Q->rear->next=p;                            /* 新结点链接到队尾 */
    Q->rear=p;                                  /* 修改队尾指针 */
```

```
    return OK;
}
```

### 4. 出队

**【算法 3.17】** 链队出队。

```
int DeQueue_L(LinkQueue *Q,ElemType *e)
{
    LinkNode *p;
    if(EmptyQueue_L(Q))
    {
        return ERROR;
    }
    p=Q->front->next;
    *e=p->data;                              /*e返回队头元素值*/
    if(Q->front->next==Q->rear)      /*若出队的是队列的唯一结点,出队后队列空*/
    {
        Q->front->next=NULL;
        Q->rear=Q->front;                    /*队列置空*/
    }
    else
        Q->front->next=p->next;              /*队头元素出队*/
    free(p);
    return OK;
}
```

## 3.3 算法设计举例

**【例 3-1】** 将一个无符号十进制整数转换成八进制数。

**【解】** 将一个十进制整数转换成其他进制数,其基本算法是通过对十进制整数作“除基取余倒排序”运算得到结果。因此,将一个十进制整数转换成八进制数,可以设置一个栈,依次存放每一次“除基取余”运算得到的余数,利用栈的“后进先出”特性,依次输出栈中元素即得到转换的结果。

具体算法如下:

```
void transfer()
{
    int x;
    InitStack(s);               /*初始化栈*/
    scanf("%d",&x);
    while(x)                    /*除基取余,余数依次入栈*/
    {
        Push(s,x%8);
```

```
        x=x/8;
    }
    while(!EmptyStack(s))      /*当栈不为空时,依次出栈*/
    {
        x=Pop(s);
        printf("%d",x);
    }
}
```

**【例 3-2】** 检查表达式中括号是否匹配(只考虑小括号是否匹配)。

**【解】** 利用栈来完成匹配检查。初始化一个字符类型空栈。字符串表达式读入过程中,利用栈对表达式的逐个字符按照以下方式进行处理:

(1)遇到“(”,将其入栈。

(2)遇到“)”,从栈顶出栈一个元素进行判断:

① 若栈顶元素为“(”,则继续读入字符串。

② 若栈顶元素不为“(”,则说明括号不匹配,返回失败标识。

③ 若此时已经为空栈,则匹配失败,返回失败标识。

重复上述处理过程,直到将整个表达式所有字符处理完且栈为空,则匹配成功;否则,匹配失败。

算法设计如下:

```
int Match(char s[],SqStack *sp)
{
    int i;
    char t;
    for(i=0; i<strlen(s); i++)
    {
        if(s[i]=='(')          /*遇到左括号入栈*/
            Push(sp,s[i]);
        if(s[i]==')')          /*遇到右括号出栈*/
        {
            Pop(sp,&t);
            if(t!='(')         /*若当前出栈元素不是左括号,说明不匹配*/
                return ERROR;
        }
    }
    if(EmptyStack(sp))         /*表达式读取完,若此时栈不空,则不匹配*/
        return OK;
    else
        return ERROR;
}
```

**【例 3-3】** 迷宫问题。

**【解】** 迷宫问题的存储,二维数组,用 0 表示通路,1 表示阻断,对于 M×N 的迷宫,可用一个二维数组 Maze[M][N]保存迷宫的初始状态。

```
int Maze[10][10] ={
                  {1,1,1,1,1,1,1,1,1,1},
                  {1,0,0,1,0,0,0,1,0,1},
                  {1,0,0,1,0,0,0,1,0,1},
                  {1,0,0,0,0,1,1,0,0,1},
                  {1,0,1,1,1,0,0,0,0,1},
                  {1,0,0,0,1,0,0,0,0,1},
                  {1,0,1,0,0,0,1,0,0,1},
                  {1,0,1,1,1,0,1,1,0,1},
                  {1,1,0,0,0,0,0,0,0,1},
                  {1,1,1,1,1,1,1,1,1,1}
};
```

算法思想:可借助蛮力法进行穷举求解。分别设置迷宫中每个路径方块的结构(坐标和方向)和查找方向的结构(上下左右 4 种查找方向),并约定从迷宫左上角开始查找。查找策略是,首先找到入口方块的位置;其次从入口方块开始,按照向右、向下、向左、向上 4 个方向的顺序进行路径探查,直到找到出口或无通路结束探查。为了保证在探查过程中的任何位置均可沿原路返回,可设置一个栈保存从入口到当前位置的路径。

整个迷宫求解算法可描述如下:

(1) 若当前方块可以通过,则留下可通过标记并纳入当前路径,继续下一步探查。

(2) 若当前方块不能通过,则沿原路返回前一个方块,在其他方向继续探查;若该方块的所有方向均不通,则从当前路径中删除该方块并标记为不可通过。

重复上述两步,直到走出迷宫或确定迷宫无通路为止。

具体算法可设计如下:

```
typedef struct           /* 定义路径方块查找方向的结构 */
{
    int x;
    int y;
} Direction;
typedef struct           /* 定义路径方块的结构 */
{
    int row;
    int column;
    int direction;
} ElemType;
int Mazepath(SqStack *S,int Maze[][N],Direction Move[])
{
    ElemType temp,tryPath;
```

```
int x,y,d,i,j,tryTime;
for(i=1; i<M; i++)                      /*从左上角开始查找入口*/
{
    for(j=1; j<N; j++)
        if(Maze[i][j]==0)break;
    if(Maze[i][j]==0)
    {
        x=i;
        y=j;
        Maze[x][y]=-1;
        break;                          /*找到入口*/
    }
}
if(Maze[M-2][N-2])return 0;             /*无入口,不探查*/
temp.row=x;
temp.column=y;
temp.direction=0;                       /*初始化路径上的入口路径方块*/
Push(S,temp);                           /*第一个路径方块入栈*/
while(!EmptyStack(S))                   /*循环检查从入口到出口的路径*/
{
    GetTop(S,&temp);                    /*获取栈顶的路径方块*/
    x=temp.row;
    y=temp.column;
    d=temp.direction+1;                 /*设置当前路径方块的查找方向*/
    tryTime=0;                          /*查找次数初始化*/
    while(d<=4)                         /*分4个方向进行探查*/
    {
        i=x+Move[d].x;
        j=y+Move[d].y;                  /*根据探查方向设置相应的方块坐标*/
        if(Maze[i][j]==0)               /*可通过*/
        {
            if(tryTime==0)
            {
                Pop(S,&tryPath);        /*出栈当前路径方块*/
                tryPath.direction=d;    /*更新当前路径方块的探查方向*/
                Push(S,tryPath);        /*此路径方块重新入栈*/
                tryTime++;
            }
            temp.row=i;
            temp.column=j;
            temp.direction=1;           /*设置新的可通路径方块*/
            Push(S,temp);               /*新路径方块入栈*/
            x=i;
```

```
                y=j;
                Maze[x][y]=9;                  /* 在迷宫中留下通过足迹 */
                if((x==M-2)&&(y==N-2)) return 1;     /* 判断到达出口,若是,完成 */
                else d=1;                      /* 设置新路径方块的查找方向,继续探查 */
            }
            else
            {
                d++;                           /* 当前位置不能通过,查找下一个方向 */
                tryTime=0;
            }
        }
        if(d==5&&!EmptyStack(S))               /* 若此路径方块的 4 个方向均探查过 */
        {
            Pop(S,&temp);                      /* 删除此路径方块 */
            Maze[x][y]=2;                      /* 标记此路径方块不通 */
        }
    }
    return 0;                                  /* 迷宫无通路,结束查找,返回失败 */
}
```

**【例 3-4】** 表达式计算。

表达式计算是栈的典型应用。为了更好地描述算法,这里对表达式进行简化。假设表达式的所有操作数均为无符号整数且小于 10,操作符都用一个字符表示,仅包括小括号( )、加+、减－、乘*、除/。例如,(3+4)*5－6/3+9。

**【解】** 应用栈来解决表达式计算的算法分为两个步骤:

(1) 将表达式转换为后缀表达式。

(2) 根据后缀表达式进行求解。

后缀表达式也称为逆波兰式。在后缀表达式中,操作符始终在其两个操作数之后。例如,表达式(3+4)*5－6/3+9 的后缀表达式表示为 34+5*63/－9+。

将表达式转换为后缀表达式借助字符栈实现转换。由于操作符具有优先级,因此首先对设置操作符的优先级,如表 3-1 所示。

**表 3-1 各个操作符的优先级**

| 操作符 | # | ( | + | － | * | / |
| --- | --- | --- | --- | --- | --- | --- |
| 优先级 | －1 | 0 | 1 | 1 | 2 | 2 |

从原表达式求得后缀式的规则如下:

(1) 设立操作符栈。

(2) 设表达式的结束符为"#",预设操作符栈的栈底为"#"。

(3) 若当前字符是操作数,则直接发送给后缀式。

(4) 若当前字符为操作符且优先级大于栈顶操作符,则入栈,否则退出栈顶操作符发

送给后缀式。

（5）若当前字符是结束符，则自栈顶至栈底依次将栈中所有操作符发送给后缀式。

（6）“（”对它之前后的操作符起隔离作用，则若当前操作符为“（”时入栈。

（7）“）”可视为自相应左括号开始表达式的结束符，则从栈顶起，依次退出栈顶操作符发送给后缀式直至栈顶字符为“（”止。“（”不发送到后缀式。

以表达式(3＋4)＊5－6/3＋9为例说明后缀表达式的生成过程，如表3-2所示。

**表3-2 后缀表达式的生成过程**

| 序号 | 读取字符 | 类型 | 操 作 | 栈内变化 | 输出后缀式 |
|---|---|---|---|---|---|
| 0 | | | 预设栈 | # | |
| 1 | ( | 操作符 | (优先级大于#入栈 | # ( | |
| 2 | 3 | 操作数 | | # ( | 3 |
| 3 | ＋ | 操作符 | ＋优先级大于(，入栈 | # ( ＋ | |
| 4 | 4 | 操作数 | | # ( ＋ | 34 |
| 5 | ) | 操作符 | 遇到)，出栈＋<br>直到遇到(出栈 | #(<br># | 34＋ |
| 6 | * | 操作符 | *优先级大于#，入栈 | # * | |
| 7 | 5 | 操作数 | | # * | 34＋5 |
| 8 | － | 操作符 | －优先级小于*，出栈，<br>直到－大于#，入栈 | #<br># － | 34＋5* |
| 9 | 6 | 操作数 | | # － | 34＋5*6 |
| 10 | / | 操作符 | /优先级大于－，入栈 | # － / | 34＋5*6 |
| 11 | 3 | 操作数 | | | 34＋5*63 |
| 12 | ＋ | 操作符 | ＋优先级小于/，出栈<br>＋优先级不大于－，出栈<br>＋优先级大于#，入栈 | # －<br>#<br># ＋ | 34＋5*63/－ |
| 13 | 9 | 操作数 | | # ＋ | 34＋5*63/－9 |
| 14 | # | 结束符 | 出栈 | | 34＋5*63/－9＋ |

利用后缀表达式求解，只需要从左向右依次扫描表达式，遇到操作数入栈，遇到操作符，则做出栈两次，获得两个操作数，后出栈的操作数为第一个操作对象，对它们进行计算，计算结果作为下次运算的操作数入栈。重复上述操作，直到后缀表达式读取结束，既可完成表达式的计算。计算过程如表3-3所示。

表 3-3 后缀表达式计算过程

| 序号 | 读取字符 | 类型 | 操　　作 | 栈内变化 | 计算结果 |
|---|---|---|---|---|---|
| 1 | 3 | 操作数 | 入栈 | 3 | |
| 2 | 4 | 操作数 | 入栈 | 3 4 | |
| 3 | ＋ | 操作符 | 出栈 4，出栈 3<br>7 入栈 | 7 | 3＋4－＞7 |
| 4 | 5 | 操作数 | 入栈 | 7 5 | |
| 5 | * | 操作符 | 出栈 5，出栈 7<br>35 入栈 | 35 | 7*5－＞35 |
| 6 | 6 | 操作数 | 入栈 | 35 6 | |
| 7 | 3 | 操作数 | 入栈 | 35 6 3 | |
| 8 | / | 操作符 | 出栈 3，出栈 6<br>2 入栈 | 35 2 | 6/3－＞2 |
| 9 | － | 操作符 | 出栈 2，出栈 35<br>33 入栈 | 33 | 35－2－＞33 |
| 10 | 9 | 操作数 | 入栈 | 33 9 | |
| 11 | ＋ | 操作符 | 出栈 9，出栈 33<br>42 入栈 | 42 | 33＋9－＞42 |
| 12 | 结束 | | 出栈 | | 42 |

由上述两个计算步骤可知，表达式计算需要用到两个栈：一个用来暂存操作符，实现后缀表达式的转换；另一个用来暂存操作数，实现后缀表达式的计算。算法实现可以先完成后缀表达式，再进行计算，也可以边转换，边计算表达式。

## 3.4 小结

本章介绍了线性结构的两种受限形式——栈和队列。栈属于“后进先出”线性表，它的删除和插入操作都在栈顶进行。队列属于“先进先出”线性表，插入在队尾进行，删除在队头进行。

根据栈和队列各自的特点，本章介绍了顺序栈和链栈的存储表示以及基本操作的实现。栈的应用非常广泛，如进制转换、括弧匹配、表达式计算，递归求解、迷宫问题等。队列的“先进先出”针对于需要遵循“先来先服务”原则的实际问题具有非常好的应用。循环队列既实现了队列的存储和基本操作，又很好地解决了顺序队列溢出的问题。

在本章学习中，掌握栈和队列的特点。

## 习题 3

3.1　简要叙述栈和队列的基本特性。

3.2　解释队列的“假溢出”现象。如何解决“假溢出”？

3.3 若元素的入栈序列为ABCDE，运用栈操作，能否得到出栈序列BCAED和DBACE，请说明原因。

3.4 已知字符串b%-y-3*y^2，请利用栈将其转换为by-%3y2^*-，写出入栈出栈操作序列。

3.5 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点(注意，不设头指针)，试编写相应的队列初始化、入队列和出队列的算法。

3.6 试编写一个表达式转换为后缀表达式的算法。

3.7 试编写一个算法，实现后缀表达式的计算。