第3章 栈和队列

从组成元素的逻辑关系看，栈和队列都属于线性结构。栈和队列与线性表的不同之处在于它们的相关运算具有一些特殊性。更准确地说，一般线性表上的插入、删除运算不受限制，而栈和队列上的插入、删除运算均受某种特殊限制，因此栈和队列也称为操作受限的线性表。

本章介绍栈和队列的基本概念、存储结构、基本运算算法设计和应用实例。

3.1 栈

栈是一种常用而且重要的数据结构之一，如用于保存函数调用时所需要的信息，通常在将递归算法转换成非递归算法时需要使用到栈。本节主要讨论栈及其应用。

3.1.1 栈的定义

栈(stack)是一种只能在一端进行插入或删除操作的线性表。表中允许进行插入、删除操作的一端称为**栈顶**(top)，表的另一端称为**栈底**(bottom)，如图 3.1 所示。栈顶的当前位置是动态的，栈顶的当前位置由一个被称为栈顶指针的位置指示器来指示。当栈中没有数据元素时称为**空栈**。栈的插入操作通常称为**进栈**或**入栈**(push)，栈的删除操作通常称为**出栈**或**退栈**(pop)。

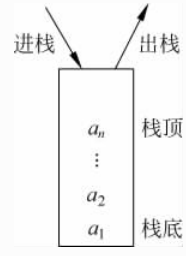

图 3.1 栈示意图

栈的主要特点是“后进先出”(Last In First Out，LIFO)，即后进栈的元素先出栈。每次进栈的数据元素都放在原来栈顶元素之前成为新的栈顶元素，每次出栈的数据元素都是当前栈顶元素。栈也称为**后进先出表**。

例如，若干个人走进一个死胡同，假设该死胡同的宽度恰好只够一个人进出，那么走出死胡同的顺序和走进的顺序正好相反。这个死胡同就是一个栈。

栈抽象数据类型的定义如下：

```
ADT Stack
{   数据对象：
        D={ a_i | 1≤i≤n,n≥0,a_i 为 ElemType 类型}      //ElemType 是自定义类型标识符
    数据关系：
        R={<a_i,a_i+1> | a_i、a_i+1∈ D,i=1,…,n-1}
    基本运算：
        InitStack(&s)：初始化栈，构造一个空栈 s。
        DestroyStack(&s)：销毁栈，释放栈 s 占用的存储空间。
        StackEmpty(s)：判断栈是否为空，若栈 s 为空，则返回真；否则返回假。
        Push(&s,e)：进栈，将元素 e 插入到栈 s 中作为栈顶元素。
        Pop(&s,&e)：出栈，从栈 s 中删除栈顶元素，并将其值赋给 e。
        GetTop(s,&e)：取栈顶元素，返回当前的栈顶元素，并将其值赋给 e。
}
```

【例 3.1】 若元素的进栈序列为 1234，能否得到 3142 的出栈序列？

解 为了让 3 作为第一个出栈元素，1、2 先进栈，此时要么 2 出栈，要么 4 进栈后出栈，出栈的第 2 个元素不可能是 1，所以得不到 3142 的出栈序列。

【例 3.2】 用 S 表示进栈操作、X 表示出栈操作，若元素的进栈顺序为 1234，为了得到 1342 的出栈序列，给出相应的 S 和 X 操作串。

解 为了得到 1342 的出栈序列，其操作过程是 1 进栈，1 出栈，2 进栈，3 进栈，3 出栈，

4 进栈，4 出栈，2 出栈。因此相应的 S 和 X 操作串为 SXSSXSXX。

说明：n 个不同的元素通过一个栈产生的出栈序列的个数为 $\frac{1}{n+1}C_{2n}^{n}$。例如 $n=4$ 时，出栈序列的个数等于 14。

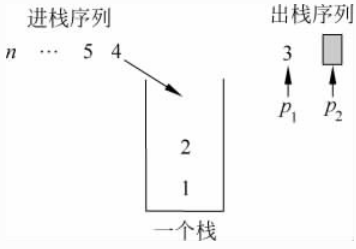

图 3.2　栈操作的一个时刻

【例 3.3】 一个栈的进栈序列为 1,2,…,n，通过一个栈得到出栈序列 $p_1,p_2,\cdots,p_n$（$p_1,p_2,\cdots,p_n$ 是 1,2,…,n 的一种排列）。若 $p_1=3$，则 p_2 可能取值的个数是多少？

解 为了让 3 作为第一个出栈元素，将 1、2、3 依次进栈，3 出栈，此时如图 3.2 所示。之后可以让 2 出栈，$p_2=2$，也可以让 4 进栈再出栈，$p_2=4$，也可以让 4、5 进栈再出栈，$p_2=5$，…，所以 p_2 可以是 2,4,5,…,n，不可能是 1 和 3，即 p_2 可能取值的个数是 $n-2$。

3.1.2　栈的顺序存储结构及其基本运算的实现

栈中数据元素的逻辑关系呈线性关系，所以栈可以像线性表一样采用顺序存储结构进行存储，即分配一块连续的存储空间来存放栈中元素，并用一个变量（如 top）指向当前的栈顶元素以反映栈中元素的变化。采用顺序存储结构的栈称为**顺序栈**（sequential stack）。

假设栈的元素个数最大不超过正整数 MaxSize，所有的元素都具有同一数据类型，即 ElemType，可用下列方式来声明顺序栈的类型 SqStack：

```
typedef struct
{    ElemType data[MaxSize];        //存放栈中的数据元素
     int top;                       //栈顶指针，即存放栈顶元素在 data 数组中的下标
} SqStack;                          //顺序栈类型
```

栈到顺序栈的映射过程如图 3.3 所示。本节采用栈指针 s（不同于栈顶指针 top）的方式创建和使用顺序栈，如图 3.4 所示。

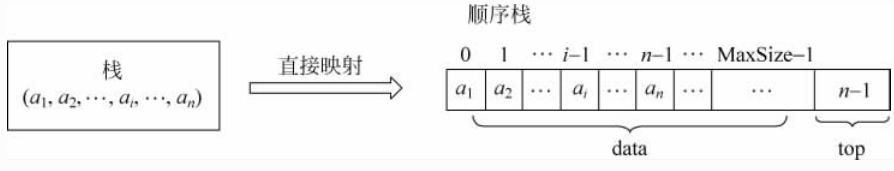

图 3.3　栈到顺序栈的映射

图 3.5 是一个顺序栈操作示意图。图 3.5(a)是初始情况，它是一个空栈；图 3.5(b)表示元素 a 进栈以后的状态；图 3.5(c)表示元素 b、c、d 进栈以后的状态；图 3.5(d)表示元素 d 出栈以后的状态。

图 3.4　顺序栈指针 s

综上所述，对于 s 所指的顺序栈（即顺序栈 s），初始时设置 $s->top=-1$，可以归纳出对后面算法设计来说非常重要的 4 个要素。

- 栈空的条件：$s->top==-1$。

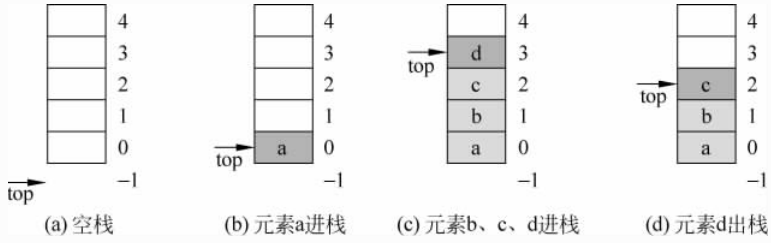

图 3.5　栈操作示意图

- 栈满的条件：s->top==MaxSize-1(data 数组的最大下标)。
- 元素 e 的进栈操作：先将栈顶指针 top 增 1，然后将元素 e 放在栈顶指针处。
- 出栈操作：先将栈顶指针 top 处的元素取出放在 e 中，然后将栈顶指针减 1。

在顺序栈上对应栈的基本运算算法设计如下。

1）初始化栈 initStack(&s)

该运算创建一个空栈，由 s 指向它。实际上就是分配一个顺序栈空间，并将栈顶指针设置为-1。算法如下：

```
void InitStack(SqStack *&s)
{   s=(SqStack *)malloc(sizeof(SqStack));   //分配一个顺序栈空间,首地址存放在 s 中
    s->top=-1;                              //栈顶指针置为-1
}
```

2）销毁栈 DestroyStack(&s)

该运算释放顺序栈 s 占用的存储空间。算法如下：

```
void DestroyStack(SqStack *&s)
{
    free(s);
}
```

3）判断栈是否为空 StackEmpty(s)

该运算实际上用于判断条件 s->top==-1 是否成立。算法如下：

```
bool StackEmpty(SqStack *s)
{
    return(s->top==-1);
}
```

4）进栈 Push(&s,e)

该运算的执行过程是，在栈不满的条件下先将栈顶指针增 1，然后在该位置上插入元素 e，并返回真；否则返回假。算法如下：

```
bool Push(SqStack *&s,ElemType e)
{   if (s->top==MaxSize-1)          //栈满的情况,即栈上溢出
        return false;
```

```
    s->top++;                          //栈顶指针增 1
    s->data[s->top]=e;                 //元素 e 放在栈顶指针处
    return true;
}
```

5）出栈 Pop(&s,&e)

该运算的执行过程是，在栈不为空的条件下先将栈顶元素赋给 *e*，然后将栈顶指针减 1，并返回真；否则返回假。算法如下：

```
bool Pop(SqStack *&s,ElemType &e)
{   if (s->top==-1)                    //栈为空的情况，即栈下溢出
        return false;
    e=s->data[s->top];                 //取栈顶元素
    s->top--;                          //栈顶指针减 1
    return true;
}
```

6）取栈顶元素 GetTop(*s*,&*e*)

该运算在栈不为空的条件下将栈顶元素赋给 *e* 并返回真；否则返回假。算法如下：

```
bool GetTop(SqStack *s,ElemType &e)
{   if (s->top==-1)                    //栈为空的情况，即栈下溢出
        return false;
    e=s->data[s->top];                 //取栈顶元素
    return true;
}
```

和出栈运算相比，本算法只是没有移动栈顶指针。上述 6 个基本运算算法的时间复杂度均为 $O(1)$，说明这是一种非常高效的设计。

【例 3.4】 设计一个算法利用顺序栈判断一个字符串是否为对称串。所谓对称串指从左向右读和从右向左读的序列相同。

解 *n* 个元素连续进栈，产生的连续出栈序列和输入序列正好相反，本算法就是利用这个特点设计的。对于字符串 str，从头到尾将其所有元素连续进栈，如果所有元素连续出栈产生的序列和 str 从头到尾的字符依次相同，表示 str 是一个对称串，返回真；否则表示 str 不是对称串，返回假。算法如下：

```
bool symmetry(ElemType str[])          //判断 str 是否为对称串
{   int i; ElemType e;
    SqStack *st;                       //定义顺序栈指针
    InitStack(st);                     //初始化栈
    for (i=0;str[i]!='\0';i++)         //将 str 的所有元素进栈
        Push(st,str[i]);
    for (i=0;str[i]!='\0';i++)         //处理 str 的所有字符
    {   Pop(st,e);                     //退栈元素 e
        if (str[i]!=e)                 //若 e 与当前串字符不同表示不是对称串
```

```
        {   DestroyStack(st);          //销毁栈
            return false;              //返回假
        }
    }
    DestroyStack(st);                  //销毁栈
    return true;                       //返回真
}
```

顺序栈采用一个数组存放栈中的元素。如果需要用到两个相同类型的栈，这时若为它们各自开辟一个数组空间，极有可能出现这样的情况：第一个栈已满，再进栈就溢出了，而另一个栈还有很多空闲存储空间。解决这个问题的方法是将两个栈合起来，如图 3.6 所示，用一个数组来实现这两个栈，这称为**共享栈**(share stack)。

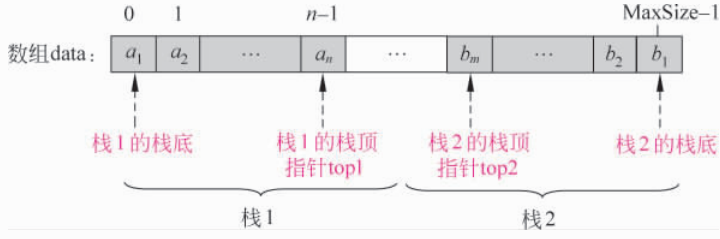

图 3.6　共享栈

在设计共享栈时，由于一个数组(大小为 MaxSize)有两个端点，两个栈有两个栈底，让一个栈的栈底为数组的始端，即下标为 0 处，另一个栈的栈底为数组的末端，即下标为 MaxSize－1，这样在两个栈中进栈元素时栈顶向中间伸展。

共享栈的 4 个要素如下。

- 栈空条件：栈 1 空为 top1==－1；栈 2 空为 top2==MaxSize。
- 栈满条件：top1==top2－1。
- 元素 x 进栈操作：进栈 1 操作为 top1＋＋；data[top1]＝x；进栈 2 操作为 top2－－；data[top2]＝x。
- 出栈 x 操作：出栈 1 操作为 x＝data[top1]；top1－－；出栈 2 操作为 x＝data[top2]；top2＋＋。

在上述设置中，data 数组表示共享栈的存储空间，top1 和 top2 分别为两个栈的栈顶指针，这样该共享栈通过 data、top1 和 top2 来标识，也可以将它们设计为一个结构体类型：

```
typedef struct
{   ElemType data[MaxSize];        //存放共享栈中的元素
    int top1,top2;                 //两个栈的栈顶指针
} DStack;                          //共享栈的类型
```

在实现共享栈的基本运算算法时需要增加一个形参 i，指出是对哪个栈进行操作，如 i＝1 表示对栈 1 进行操作，i＝2 表示对栈 2 进行操作。

3.1.3　栈的链式存储结构及其基本运算的实现

栈中数据元素的逻辑关系呈线性关系，所以栈可以像线性表一样采用链式存储结构。

采用链式存储结构的栈称为**链栈**(linked stack)。链表有多种,这里采用带头结点的单链表来实现链栈。

链栈的优点是不存在栈满上溢出的情况。规定栈的所有操作都是在单链表的表头进行的(因为给定链栈后,已知头结点地址,在其后面插入一个新结点和删除首结点都十分方便,对应算法的时间复杂度均为 $O(1)$)。

图 3.7 所示为头结点指针为 s 的链栈,首结点是栈顶结点,尾结点是栈底结点。栈中元素自栈底到栈顶依次是 $a_1,a_2,\cdots,a_n$。

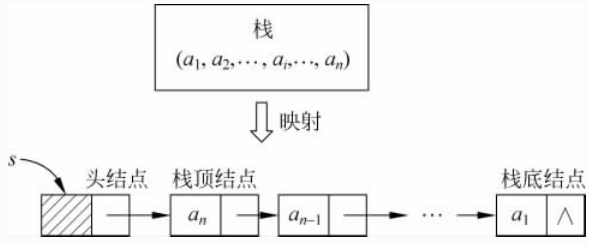

图 3.7 栈到链栈的映射

链栈中结点类型 LinkStNode 的声明如下:

```
typedef struct linknode
{    ElemType data;                  //数据域
     struct linknode * next;         //指针域
} LinkStNode;                        //链栈结点类型
```

在以 s 为头结点指针的链栈(简称链栈 s)中,可以归纳出对后面算法设计来说非常重要的 4 个要素。

- 栈空的条件:s->next==NULL。
- 栈满的条件:由于只有内存溢出时才出现栈满,通常不考虑这样的情况,所以在链栈中可以看成不存在栈满。
- 元素 e 的进栈操作:新建一个结点存放元素 e(由 p 指向它),将结点 p 插入到头结点之后。
- 出栈操作:取出首结点的 data 值并将其删除。

在链栈上对应栈的基本运算算法设计如下。

1) 初始化栈 initStack(&s)

该运算创建一个空链栈 s,如图 3.8 所示。实际上是创建链栈的头结点,并将其 next 域置为 NULL。算法如下:

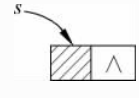

图 3.8 创建一个空栈

```
void InitStack(LinkStNode * &s)
{    s=(LinkStNode * )malloc(sizeof(LinkStNode));
     s->next=NULL;
}
```

本算法的时间复杂度为 $O(1)$。

2）销毁栈 DestroyStack(&s)

该运算释放链栈 s 占用的全部结点空间，和单链表的销毁算法完全相同。算法如下：

```
void DestroyStack(LinkStNode *&s)
{   LinkStNode *pre=s, *p=s->next;          //pre 指向头结点，p 指向首结点
    while (p!=NULL)                         //循环到 p 为空
    {   free(pre);                          //释放 pre 结点
        pre=p;                              //pre、p 同步后移
        p=pre->next;
    }
    free(pre);                              //此时 pre 指向尾结点，释放其空间
}
```

本算法的时间复杂度为 $O(n)$，其中 n 为链栈中的数据结点个数。

3）判断栈是否为空 StackEmpty(s)

该运算判断 s->next=NULL 的条件是否成立。算法如下：

```
bool StackEmpty(LinkStNode *s)
{
    return(s->next==NULL);
}
```

本算法的时间复杂度为 $O(1)$。

4）进栈 Push(&s,e)

该运算新建一个结点，用于存放元素 e（由 p 指向它），然后将其插入到头结点之后作为新的首结点。算法如下：

```
void Push(LinkStNode *&s,ElemType e)
{   LinkStNode *p;
    p=(LinkStNode *)malloc(sizeof(LinkStNode));     //新建结点 p
    p->data=e;                                      //存放元素 e
    p->next=s->next;                                //将 p 结点插入作为首结点
    s->next=p;
}
```

本算法的时间复杂度为 $O(1)$。

5）出栈 Pop(&s,&e)

该运算在栈不为空的条件下提取首结点的数据域赋给引用型参数 e，然后将其删除。算法如下：

```
bool Pop(LinkStNode *&s,ElemType &e)
{   LinkStNode *p;
    if (s->next==NULL)          //栈空的情况
        return false;           //返回假
    p=s->next;                  //p 指向首结点
    e=p->data;                  //提取首结点值
    s->next=p->next;            //删除首结点
```

```
    free(p);                    //释放被删结点的存储空间
    return true;                //返回真
}
```

本算法的时间复杂度为 $O(1)$。

6）取栈顶元素 GetTop(*s*,&*e*)

该运算在栈不为空的条件下提取首结点的数据域赋给引用型参数 *e*。算法如下：

```
bool GetTop(LinkStNode *s,ElemType &e)
{   if (s->next==NULL)          //栈空的情况
        return false;           //返回假
    e=s->next->data;            //提取首结点值
    return true;                //返回真
}
```

和出栈运算相比，本算法只是没有改变栈顶结点，其时间复杂度为 $O(1)$。

【例 3.5】 设计一个算法判断输入的表达式中括号是否配对（假设只含有左、右圆括号）。

解 该算法在表达式括号配对时返回真，否则返回假。设置一个链栈 st，扫描表达式 exp，遇到左括号时进栈；遇到右括号时，若栈顶为左括号，则出栈，否则返回假。当表达式扫描完毕而且栈为空时返回真；否则返回假。算法如下：

```
bool Match(char exp[],int n)
{   int i=0; char e;
    bool match=true;
    LinkStNode *st;
    InitStack(st);                      //初始化链栈
    while (i<n && match)                //扫描 exp 中的所有字符
    {   if (exp[i]=='(')                //当前字符为左括号,将其进栈
            Push(st,exp[i]);
        else if (exp[i]==')')           //当前字符为右括号
        {   if (GetTop(st,e)==true)     //成功取栈顶元素 e
            {   if (e!='(')             //栈顶元素不为'('时
                    match=false;        //表示不匹配
                else                    //栈顶元素为'('时
                    Pop(st,e);          //将栈顶元素出栈
            }
            else match=false;           //无法取栈顶元素时表示不匹配
        }
        i++;                            //继续处理其他字符
    }
    if (!StackEmpty(st))                //栈不空时表示不匹配
        match=false;
    DestroyStack(st);                   //销毁栈
    return match;
}
```

3.1.4 栈的应用

在实际应用中，栈通常作为一种存放临时数据的容器。如果后存入的元素先处理，则采用栈。本小节通过简单表达式求值和迷宫问题的求解过程来说明栈的应用。

1. 简单表达式求值

1）问题描述

这里限定的简单表达式求值问题是用户输入一个包含＋、－、＊、/、正整数和圆括号的合法算术表达式，计算该表达式的运算结果。

2）数据组织

简单表达式采用字符数组 exp 表示，其中只含有＋、－、＊、/、正整数和圆括号。为了方便，假设该表达式都是合法的算术表达式，例如 exp＝"1＋2＊(4＋12)"，在设计相关算法中用到栈，这里采用顺序栈存储结构。

3）设计运算算法

在算术表达式中，运算符位于两个操作数中间的表达式称为**中缀表达式**(infix expression)，例如 1＋2＊3 就是一个中缀表达式。中缀表达式是一种最常用的表达式形式，日常生活中的表达式一般都是中缀表达式。

对中缀表达式的运算一般遵循“先乘除，后加减，从左到右计算，先括号内，后括号外”的规则，因此中缀表达式不仅要依赖运算符优先级，还要处理括号。

算术表达式的另一种形式是**后缀表达式**(postfix expression)或逆波兰表达式，就是在算术表达式中运算符在操作数的后面，如 1＋2＊3 的后缀表达式为 1 2 3 ＊ ＋。在后缀表达式中已经考虑了运算符的优先级，没有括号，只有操作数和运算符，而且越放在前面的运算符越优先执行。

同样，在算术表达式中，如果运算符在操作数的前面，称为**前缀表达式**(prefix expression)，如 1＋2＊3 的前缀表达式为＋ 1 ＊ 2 3。

后缀表达式是一种十分有用的表达式，它将复杂表达式转换为可以依靠简单的操作得到计算结果的表达式。所以对中缀表达式的求值过程是先将中缀算术表达式转换成后缀表达式，然后对该后缀表达式求值。

(1) 将算术表达式转换成后缀表达式。

在将一个中缀表达式转换成后缀表达式时，操作数之间的相对次序是不变的，但运算符的相对次序可能不同，同时还要除去括号。所以在转换时需要从左到右扫描算术表达式，将遇到的操作数直接存放到后缀表达式中，将遇到的每一个运算符或者左括号都暂时保存到运算符栈，而且先执行的运算符先出栈。

假设用 exp 字符数组存储满足前面条件的简单中缀表达式，其对应的后缀表达式存放在字符数组 postexp 中。下面讨论几种情况。

例如，若 exp＝"1＋2＋3"，转换过程是首先将操作数 1 存入 postexp；遇到第 1 个'＋'，尚未确定它是否最先执行，将其进栈；再将操作数 2 存入 postexp；

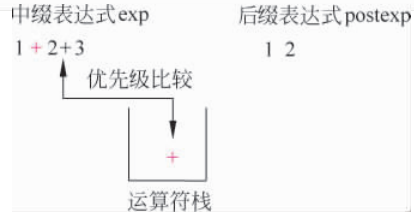

图 3.9　两个'＋'进行优先级比较

又遇到第 2 个'+',需要两个'+'进行优先级比较,如图 3.9 所示,如果直接将第 2 个'+'进栈,它以后一定先出栈,表示第 2 个'+'比第 1 个'+'先执行,显然是错误的。正确的做法是先将栈中的第 1 个'+'出栈并存入 postexp,然后再将第 2 个'+'进栈(表示第 1 个'+'先执行);最后将操作数 3 存入 postexp;此时 exp 扫描完毕,出栈第 2 个'+'并存入 postexp。得到的最后结果是 postexp="1 2+3 +"。

归纳 1:在扫描 exp 遇到一个运算符 op 时,如果栈为空,直接将其进栈;如果栈不空,只有当 op 的优先级高于栈顶运算符的优先级时才直接将 op 进栈(以后 op 先出栈表示先执行它);否则依次出栈运算符并存入 postexp(出栈的运算符都比 op 先执行),直到栈顶运算符的优先级小于 op 的优先级为止,然后再将 op 进栈。

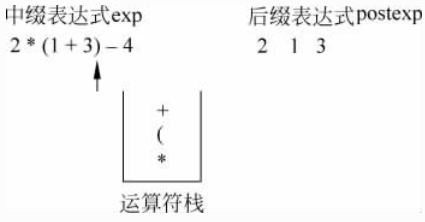

图 3.10 遇到')'的情况

再看看带有括号的例子,若 exp="2*(1+3)-4",转换过程是将操作数 2 存入 postexp;遇到'*',将其进栈;遇到'(',将其进栈;将操作数 1 存入 postexp;遇到'+',将其进栈;将操作数 3 存入 postexp;遇到')',如图 3.10 所示,出栈'+'并存入 postexp,出栈'(';遇到'-',出栈'*'并存入 postexp,将'-'进栈;将操作数 4 存入 postexp;此时 exp 扫描完毕,出栈'-'并存入 postexp。得到的最后结果是 postexp="2 1 3 + * 4 -"。

归纳 2:在扫描 exp 遇到一个运算符 op 时,如果 op 为'(',表示一个子表达式的开始,直接将其进栈;如果 op 为')',表示一个子表达式的结束,需要出栈运算符并存入 postexp,直到栈顶为'(',再将'('出栈;如果 op 是其他运算符,而栈顶为'(',直接将其进栈。

设置一个运算符栈 Optr,初始时为空。为了方便后面将数值串转换为对应的数值,在后缀表达式中的每个数字串末尾添加一个'#'。将算术表达式 exp 转换成后缀表达式 postexp 的过程如下:

```
while (从 exp 读取字符 ch,ch!='\0')
{   ch 为数字: 将后续的所有数字均依次存放到 postexp 中,并以字符'#'标识数字串结束;
    ch 为左括号'(': 将此括号进栈到 Optr 中;
    ch 为右括号')': 将 Optr 中出栈时遇到的第一个左括号'('以前的运算符依次出栈并
            存放到 postexp 中,然后将左括号'('出栈;
    ch 为其他运算符:
            if (栈空或者栈顶运算符为'(') 直接将 ch 进栈;
            else if (ch 的优先级高于栈顶运算符的优先级)
                直接将 ch 进栈;
            else
                依次出栈并存入到 postexp 中,直到 ch 的优先级高于栈顶运算符,然后将 ch 进栈;
}
若 exp 扫描完毕,则将 Optr 中的所有运算符依次出栈并存放到 postexp 中。
```

对于简单的算术表达式,'+'和'-'运算符的优先级相同,'*'和'/'运算符的优先级相同,只有'*'和'/'运算符的优先级高于'+'和'-'运算符的优先级。所以上述过程进一步改为如下:

```
while (从 exp 读取字符 ch,ch!='\0')
{   ch 为数字: 将后续的所有数字均依次存放到 postexp 中,并以字符'#'标识数字串结束;
    ch 为左括号'(': 将此括号进栈到 Optr 中;
    ch 为右括号')': 将 Optr 中出栈时遇到的第一个左括号'('以前的运算符依次出栈并
            存放到 postexp 中,然后将左括号'('出栈;
    ch 为'+'或'-': 出栈运算符并存放到 postexp 中,直到栈空或者栈顶为'(',然后将 ch
            进栈;
    ch 为'*'或'/': 出栈运算符并存放到 postexp 中,直到栈空或者栈顶为'('、'+'或'-',
            然后将 ch 进栈;
}
若 exp 扫描完毕,则将 Optr 中的所有运算符依次出栈并存放到 postexp 中。
```

例如对于表达式"(56－20)/(4＋2)",其转换为后缀表达式的过程如表 3.1 所示,最后得到的后缀表达式为"56＃20＃－4＃2＃＋/"。

表 3.1　表达式"(56－20)/(4＋2)"转换成后缀表达式的过程

操　　作	postexp	Optr 栈 (栈底→栈顶)
遇到 ch 为'(',将此括号进栈		(
遇到 ch 为数字,将 56＃存入 postexp 中	56＃	(
遇到 ch 为'－',直接将 ch 进栈	56＃	(－
遇到 ch 为数字,将 20＃存入 postexp 中	56＃20＃	(－
遇到 ch 为')',将栈中'('之前的运算符'－'出栈并存入 postexp 中,然后将'('出栈	56＃20＃－	
遇到 ch 为'/',将 ch 进栈	56＃20＃－	/
遇到 ch 为'(',将此括号进栈	56＃20＃－	/(
遇到 ch 为数字,将 4＃存入 postexp 中	56＃20＃－4＃	/(
遇到 ch 为'＋',由于栈顶运算符为'(',则直接将 ch 进栈	56＃20＃－4＃	/(＋
遇到 ch 为数字,将 2＃存入 postexp 中	56＃20＃－4＃2＃	/(＋
遇到 ch 为')',将栈中'('之前的运算符'＋'出栈并存入 postexp 中,然后将'('出栈	56＃20＃－4＃2＃＋	/
str 扫描完毕,则将 Optr 栈中的所有运算符依次出栈并存入 postexp 中,得到最终的后缀表达式	56＃20＃－4＃2＃＋/	

设置运算符栈类型 SqStack 中的 ElemType 为 char 类型。根据上述原理得到的 trans()算法如下:

```
void trans(char *exp,char postexp[])      //将算术表达式 exp 转换成后缀表达式 postexp
{   char e;
    SqStack *Optr;                        //定义运算符栈指针
    InitStack(Optr);                      //初始化运算符栈
    int i=0;                              //i 作为 postexp 的下标
    while (*exp!='\0')                    //exp 表达式未扫描完时循环
    {   switch(*exp)
        {
        case '(':                         //判定为左括号
```

```
            Push(Optr,'(');                    //左括号进栈
            exp++;                             //继续扫描其他字符
            break;
        case ')':                              //判定为右括号
            Pop(Optr,e);                       //出栈元素 e
            while (e!='(')                     //不为'('时循环
            {   postexp[i++]=e;                //将 e 存放到 postexp 中
                Pop(Optr,e);                   //继续出栈元素 e
            }
            exp++;                             //继续扫描其他字符
            break;
        case '+':                              //判定为加或减号
        case '-':
            while (!StackEmpty(Optr))          //栈不空循环
            {   GetTop(Optr,e);                //取栈顶元素 e
                if (e!='(')                    //e 不是'('
                {   postexp[i++]=e;            //将 e 存放到 postexp 中
                    Pop(Optr,e);               //出栈元素 e
                }
                else                           //e 是'('时退出循环
                    break;
            }
            Push(Optr,*exp);                   //将'+'或'-'进栈
            exp++;                             //继续扫描其他字符
            break;
        case '*':                              //判定为'*'或'/'号
        case '/':
            while (!StackEmpty(Optr))          //栈不空循环
            {   GetTop(Optr,e);                //取栈顶元素 e
                if (e=='*' || e=='/')          //将栈顶'*'或'/'运算符出栈并存放到 postexp 中
                {   postexp[i++]=e;            //将 e 存放到 postexp 中
                    Pop(Optr,e);               //出栈元素 e
                }
                else                           //e 为非'*'或'/'运算符时退出循环
                    break;
            }
            Push(Optr,*exp);                   //将'*'或'/'进栈
            exp++;                             //继续扫描其他字符
            break;
        default:                               //处理数字字符
            while (*exp>='0' && *exp<='9')     //判定为数字字符
            {   postexp[i++]=*exp;
                exp++;
            }
            postexp[i++]='#';                  //用#标识一个数字串结束
    }
}
while (!StackEmpty(Optr))                      //此时 exp 扫描完毕,栈不空时循环
{   Pop(Optr,e);                               //出栈元素 e
```

```
            postexp[i++]=e;                    //将 e 存放到 postexp 中
        }
        postexp[i]='\0';                       //给 postexp 表达式添加结束标识
        DestroyStack(Optr);                    //销毁栈
    }
```

(2) 后缀表达式求值。

后缀表达式的求值过程是从左到右扫描后缀表达式 postexp,若读取的是一个操作数,将它进操作数栈,若读取的是一个运算符 op,从操作数栈中连续出栈两个操作数,假设为 a(第 1 个出栈元素)和 b(第 2 个出栈元素),计算 b op a 的值,并将计算结果进操作数栈。当整个后缀表达式扫描结束时,操作数栈中的栈顶元素就是表达式的计算结果。

在后缀表达式求值算法设计中操作数栈为 Opnd,用于临时存放要进行某种算术运算的操作数。下面给出后缀表达式求值的过程,假设 postexp 存放的后缀表达式是正确的,在 while 循环结束后,Opnd 栈中恰好有一个操作数,它就是该后缀表达式的求值结果。

```
while (从 postexp 读取字符 ch,ch!='\0')
{   ch 为'+': 从 Opnd 栈中出栈两个数值 a 和 b,计算 c=b+a;将 c 进栈;
    ch 为'-': 从 Opnd 栈中出栈两个数值 a 和 b,计算 c=b-a;将 c 进栈;
    ch 为'*': 从 Opnd 栈中出栈两个数值 a 和 b,计算 c=b*a;将 c 进栈;
    ch 为'/': 从 Opnd 栈中出栈两个数值 a 和 b,若 a 不零,计算 c=b/a;将 c 进栈;
    ch 为数字字符: 将连续的数字串转换成数值 d,将 d 进栈;
}
返回 Opnd 栈的栈顶操作数(即后缀表达式的值);
```

后缀表达式"56#20#-4#2#+/"的求值过程如表 3.2 所示,最后的求值结果为 6,与原表达式"(56-20)/(4+2)"的计算结果一致。

表 3.2 后缀表达式"56#20#-4#2#+/"的求值过程

操 作	Opnd 栈(栈底→栈顶)
遇到 56#,将 56 进栈	56
遇到 20#,将 20 进栈	56,20
遇到'-',出栈两次,将 56-20=36 进栈	36
遇到 4#,将 4 进栈	36,4
遇到 2#,将 2 进栈	36,4,2
遇到'+',出栈两次,将 4+2=6 进栈	36,6
遇到'/',出栈两次,将 36/6=6 进栈	6
postexp 扫描完毕,算法结束,栈顶数值 6 即为所求	

设置操作数栈类型 SqStack1 中的 ElemType 为 double 类型,将栈基本运算名称后面加上"1"以区别前面字符栈的基本运算。根据上述计算原理得到求后缀表达式值的算法如下:

```
double compvalue(char *postexp)            //计算后缀表达式的值
{   double d,a,b,c,e;
    SqStack1 *Opnd;                        //定义操作数栈
    InitStack1(Opnd);                      //初始化操作数栈
```

```
    while (*postexp!='\0')                    //postexp 字符串未扫描完时循环
    {   switch (*postexp)
        {
        case '+':                             //判定为'+'号
            Pop1(Opnd,a);                     //出栈元素 a
            Pop1(Opnd,b);                     //出栈元素 b
            c=b+a;                            //计算 c
            Push1(Opnd,c);                    //将计算结果 c 进栈
            break;
        case '-':                             //判定为'-'号
            Pop1(Opnd,a);                     //出栈元素 a
            Pop1(Opnd,b);                     //出栈元素 b
            c=b-a;                            //计算 c
            Push1(Opnd,c);                    //将计算结果 c 进栈
            break;
        case '*':                             //判定为'*'号
            Pop1(Opnd,a);                     //出栈元素 a
            Pop1(Opnd,b);                     //出栈元素 b
            c=b*a;                            //计算 c
            Push1(Opnd,c);                    //将计算结果 c 进栈
            break;
        case '/':                             //判定为'/'号
            Pop1(Opnd,a);                     //出栈元素 a
            Pop1(Opnd,b);                     //出栈元素 b
            if (a!=0)
            {   c=b/a;                        //计算 c
                Push1(Opnd,c);                //将计算结果 c 进栈
                break;
            }
            else
            {   printf("\n\t 除零错误!\n");
                exit(0);                      //异常退出
            }
            break;
        default:                              //处理数字字符
            d=0;                              //将连续的数字字符转换成对应的数值存放到 d 中
            while (*postexp>='0' && *postexp<='9')      //判定为数字字符
            {   d=10*d+*postexp-'0';
                postexp++;
            }
            Push1(Opnd,d);                    //将数值 d 进栈
            break;
        }
        postexp++;                            //继续处理其他字符
    }
    GetTop1(Opnd,e);                          //取栈顶元素 e
    DestroyStack1(Opnd);                      //销毁栈
    return e;                                 //返回 e
}
```

4）设计求解程序

设计以下主函数调用上述算法：

```
int main()
{   char exp[]="(56-20)/(4+2)";                    //可将 exp 改为键盘输入
    char postexp[MaxSize];
    trans(exp,postexp);                            //将 exp 转换为 postexp
    printf("中缀表达式:%s\n",exp);                  //输出 exp
    printf("后缀表达式:%s\n",postexp);              //输出 postexp
    printf("表达式的值:%g\n",compvalue(postexp));   //求 postexp 的值并输出
    return 1;
}
```

5）运行结果

运行本程序，得到对应的结果如下：

```
中缀表达式:(56-20)/(4+2)
后缀表达式:56#20#-4#2#+/
表达式的值:6
```

2. 求解迷宫问题

扫一扫

视频讲解

1）问题描述

给定一个 $M\times N$ 的迷宫图，求一条从指定入口到出口的迷宫路径。假设一个迷宫图如图 3.11 所示(这里 $M=8,N=8$)，其中的每个方块用空白表示通道，用阴影表示障碍物。

一般情况下，所求迷宫路径是简单路径，即在求得的迷宫路径上不会重复出现同一方块。一个迷宫图的迷宫路径可能有多条，这些迷宫路径有长有短，这里仅仅考虑用栈求一条从指定入口到出口的迷宫路径。

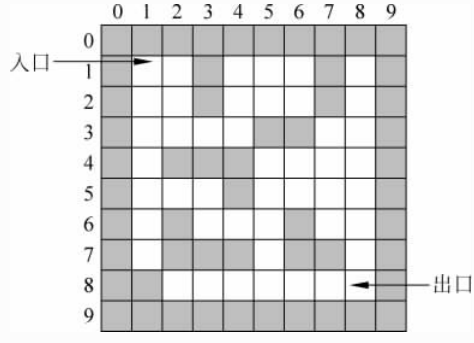

图 3.11　一个迷宫的示意图

2）数据组织

为了表示迷宫，设置一个数组 mg，其中每个元素表示一个方块的状态，为 0 时表示对应方块是通道，为 1 时表示对应方块是障碍物(不可走)。为了算法方便，一般在迷宫的外围加一条围墙。图 3.11 所示的迷宫对应的迷宫数组 mg(由于迷宫四周加了一条围墙，故 mg 数组的行数和列数均加上 2)如下：

```
int mg[M+2][N+2]=
{   {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} };
```

另外，在算法中用到的栈采用顺序栈存储结构，即将迷宫栈声明如下：

```
typedef struct
{    int i;                    //当前方块的行号
     int j;                    //当前方块的列号
     int di;                   //di 是下一相邻可走方位的方位号
} Box;                         //方块类型
typedef struct
{    Box data[MaxSize];
     int top;                  //栈顶指针
} StType;                      //顺序栈类型
```

3）设计运算算法

对于迷宫中的每个方块，有上、下、左、右 4 个方块相邻，如图 3.12 所示，第 i 行第 j 列的当前方块的位置记为(i,j)，规定上方方块为方位 0，并按顺时针方向递增编号。在试探过程中，假设按从方位 0 到方位 3 的方向查找下一个可走的相邻方块。

求迷宫问题就是在一个指定的迷宫中求出从入口到出口的一条路径。在求解时采用“穷举法”，即从入口出发，按方位 0 到方位 3 的次序试探相邻的方块，一旦找到一个可走的相邻方块就继续走下去，并记下所走的方位；若某个方块没有相邻的可走方块，则沿原路退回到前一个方块，换下一个方位再继续试探，直到所有可能的通路都试探完为止。

为了保证在任何位置上都能沿原路退回（称为回溯），需要保存从入口到当前位置的路径上走过的方块，由于回溯的过程是从当前位置退回到前一个方块，体现出后进先出的特点，所以采用栈来保存走过的方块。

若一个非出口方块(i,j)是可走的，将它进栈，每个刚刚进栈的方块，其方位 d_i 置为 -1（表示尚未试探它的周围），然后开始从方位 0 到方位 3 试探这个栈顶方块的四周，如果找到某个方位 d 的相邻方块(i_1,j_1)是可走的，则将栈顶方块(i,j)的方位 d_i 置为 d，同时将方块(i_1,j_1)进栈，再继续从方块(i_1,j_1)做相同的操作。若方块(i,j)的四周没有一个方位是可走的，将它退栈，如图 3.13 所示，前一个方块(x,y)变成栈顶方块，再从方块(x,y)的下一个方位继续试探。

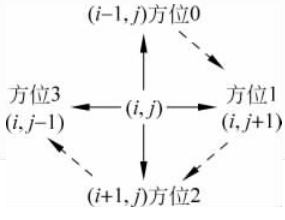

图 3.12　迷宫方位图

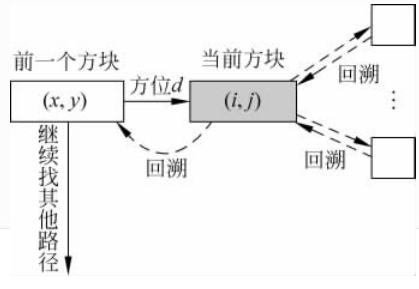

图 3.13　方块(i,j)的四周没有一个方位可走的情况

在算法中应保证试探的相邻可走方块不是已走路径上的方块。如方块(i,j)已进栈，在试探方块$(i+1,j)$的相邻可走方块时又会试探到方块(i,j)。也就是说，从方块(i,j)出发会试探方块$(i+1,j)$，而从方块$(i+1,j)$出发又会试探方块(i,j)，这样可能会引起死循环，为

此在一个方块进栈后将对应的 mg 数组元素值改为－1(变为不可走的相邻方块)，当退栈时(表示该栈顶方块没有可走相邻方块)将其恢复为 0。

求解迷宫中从入口(xi,yi)到出口(xe,ye)的一条迷宫路径的过程如下：

```
将入口(xi,yi)进栈(其初始方位设置为－1);
mg[xi][yi]＝－1;
while (栈不空)
{   取栈顶方块(i,j,di);
    if ((i,j)是出口(xe,ye))
    {   输出栈中的全部方块构成一条迷宫路径;
        return true;
    }
    查找(i,j,di)的下一个相邻可走方块;
    if (找到一个相邻可走方块)
    {   该方块为(i1,j1),对应方位 d;
        将栈顶方块的 di 设置为 d;
        (i1,j1,－1)进栈;
        mg[i1][j1]＝－1;
    }
    if (没有找到(i,j,di)的任何相邻可走方块)
    {   将(i,j,di)出栈;
        mg[i][j]＝0;
    }
}
return false;                    //没有找到迷宫路径
```

根据上述过程得到求迷宫问题的算法如下：

```
bool mgpath(int xi,int yi,int xe,int ye)        //求解路径为(xi,yi)－>(xe,ye)
{   Box path[MaxSize], e;
    int i,j,di,i1,j1,k;
    bool find;
    StType *st;                                 //定义栈 st
    InitStack(st);                              //初始化栈顶指针
    e.i＝xi; e.j＝yi; e.di＝－1;                 //设置 e 为入口
    Push(st,e);                                 //方块 e 进栈
    mg[xi][yi]＝－1;                            //将入口的迷宫值置为－1,避免重复走到该方块
    while (!StackEmpty(st))                     //栈不空时循环
    {   GetTop(st,e);                           //取栈顶方块 e
        i＝e.i; j＝e.j; di＝e.di;
        if (i＝＝xe && j＝＝ye)                  //找到了出口,输出该路径
        {   printf("一条迷宫路径如下:\n");
            k＝0;
            while (!StackEmpty(st))
            {   Pop(st,e);                      //出栈方块 e
                path[k++]＝e;                   //将 e 添加到 path 数组中
            }
            while (k>＝1)
            {   k－－;
```

```
                printf("\t(%d,%d)",path[k].i,path[k].j);
                if ((k+2)%5==0)                    //每输出5个方块后换一行
                    printf("\n");
            }
            printf("\n");
            DestroyStack(st);                      //销毁栈
            return true;                           //输出一条迷宫路径后返回true
        }
        find=false;
        while (di<4 && !find)                      //找方块(i,j)的下一个相邻可走方块(i1,j1)
        {   di++;
            switch(di)
            {
                case 0:i1=i-1; j1=j; break;
                case 1:i1=i; j1=j+1; break;
                case 2:i1=i+1; j1=j; break;
                case 3:i1=i; j1=j-1; break;
            }
            if (mg[i1][j1]==0) find=true;          //找到一个相邻可走方块,设置find为真
        }
        if (find)                                  //找到了一个相邻可走方块(i1,j1)
        {   st->data[st->top].di=di;               //修改原栈顶元素的di值
            e.i=i1; e.j=j1; e.di=-1;
            Push(st,e);                            //相邻可走方块e进栈
            mg[i1][j1]=-1;                         //将(i1,j1)迷宫值置为-1,避免重复走到该方块
        }
        else                                       //没有路径可走,则退栈
        {   Pop(st,e);                             //将栈顶方块退栈
            mg[e.i][e.j]=0;                        //让退栈方块的位置变为其他路径可走方块
        }
    }
    DestroyStack(st);                              //销毁栈
    return false;                                  //表示没有可走路径,返回false
}
```

4）设计求解程序

建立以下主函数调用上述算法：

```
int main()
{   if (!mgpath(1,1,M,N))
        printf("该迷宫问题没有解!");
    return 1;
}
```

5）运行结果

对于图3.11所示的迷宫，从入口(1,1)到出口(8,8)的求解结果如下：

```
一条迷宫路径如下:
    (1,1)  (1,2)  (2,2)  (3,2)  (3,1)
```

```
(4,1)  (5,1)  (5,2)  (5,3)  (6,3)
(6,4)  (6,5)  (5,5)  (4,5)  (4,6)
(4,7)  (3,7)  (3,8)  (4,8)  (5,8)
(6,8)  (7,8)  (8,8)
```

上述迷宫路径的显示结果如图 3.14 所示，图中路径上方块(i,j)中的箭头表示从该方块行走到下一个相邻方块的方位，例如方块(1,1)中的箭头是“→”，该箭头表示方位 1，即方块(1,1)走方位 1 到相邻方块(1,2)。显然这个解不是最优解，即不是最短路径，在使用队列求解时可以找出最短路径，这将在后面介绍。

实际上，在使用栈求解迷宫问题时，当找到出口后输出一个迷宫路径，然后可以继续回溯搜索下一条迷宫路径。采用这种回溯方法可以找出所有的迷宫路径。

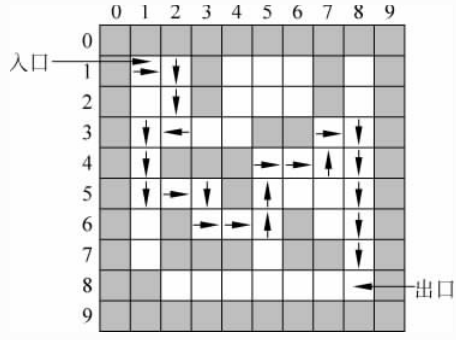

图 3.14　用栈求解的迷宫路径

3.2　队　列

队列也有广泛的应用，特别是在操作系统的资源分配和排队论中大量地使用了队列。本节主要讨论队列及其应用。

3.2.1　队列的定义

队列(queue)简称队，它也是一种操作受限的线性表，其限制为仅允许在表的一端进行插入操作，而在表的另一端进行删除操作。把进行插入的一端称为**队尾**(rear)，把进行删除的一端称为**队头**或**队首**(front)，如图 3.15 所示。向队列中插入新元素称为**进队**或**入队**(enqueue)，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**(dequeue)，元素出队后，其直接后继元素就成为队首元素。

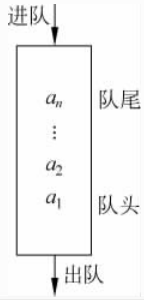

图 3.15　一个队列

由于队列的插入和删除操作分别是在各自的一端进行的，每个元素必然按照进入的次序出队，所以又把队列称为**先进先出表**(First In First Out，FIFO)。

例如，若干个人走过一个独木桥，下桥的顺序和上桥的顺序相同，在这里该独木桥就是一个队列。

队列抽象数据类型的定义如下：

```
ADT Queue
{    数据对象：
        D={ a_i | 1≤i≤n, n≥0, a_i 为 ElemType 类型}      //ElemType 是自定义类型标识符
```

数据关系：

$R=\{<a_i,a_{i+1}> \mid a_i,a_{i+1}\in D,i=1,\cdots,n-1\}$

基本运算：

InitQueue(&q)：初始化队列，构造一个空队列 q。

DestroyQueue(&q)：销毁队列，释放队列 q 占用的存储空间。

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。

enQueue(&q,e)：进队列，将元素 e 进队作为队尾元素。

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

}

【例 3.6】 若元素的进队顺序为 1234，能否得到 3142 的出队顺序？

解 若进队顺序为 1234，不同于栈，出队的顺序只有一种，即 1234(先进先出)，所以不能得到 3142 的出队顺序。

3.2.2 队列的顺序存储结构及其基本运算的实现

队列中数据元素的逻辑关系呈线性关系，所以队列可以像线性表一样采用顺序存储结构进行存储，即分配一块连续的存储空间来存放队列中的元素，并用两个整型变量来反映队列中元素的变化，它们分别存储队首元素和队尾元素的下标位置，分别称为队首指针(队头指针)和队尾指针。采用顺序存储结构的队列称为**顺序队**(sequential queue)。

假设队列中元素个数最多不超过整数 MaxSize，所有的元素都具有 ElemType 数据类型，则顺序队类型 SqQueue 声明如下：

```
typedef struct
{    ElemType data[MaxSize];        //存放队中元素
     int front,rear;                //队头和队尾指针
} SqQueue;                          //顺序队类型
```

队列到顺序队的映射过程如图 3.16 所示，并且约定在顺序队中队头指针 front 指向当前队列中队头元素的前一个位置，队尾指针 rear 指向当前队列中队尾元素的位置。本节采用队列指针 q 的方式建立和使用顺序队。

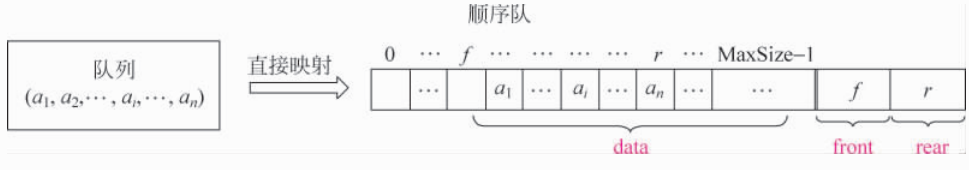

图 3.16 队列到顺序队的映射

1. 顺序队中实现队列的基本运算

图 3.17 所示为一个顺序队操作过程的示意图，其中 MaxSize=5。初始时 front=rear=-1。图 3.17(a)表示一个空队；图 3.17(b)表示进队 5 个元素后的状态；图 3.17(c)表示出队两个元素后的状态；图 3.17(d)表示再出队 3 个元素后的状态。

从图中可以看到，队空的条件为 front=rear(图 3.17(a)和图 3.17(d)都是这种情况)；

元素进队时队尾指针 rear 总是增 1，所以队满条件是 rear 指向最大下标，即 rear==MaxSize−1(图 3.17(b)和图 3.17(c)都是这种情况)。

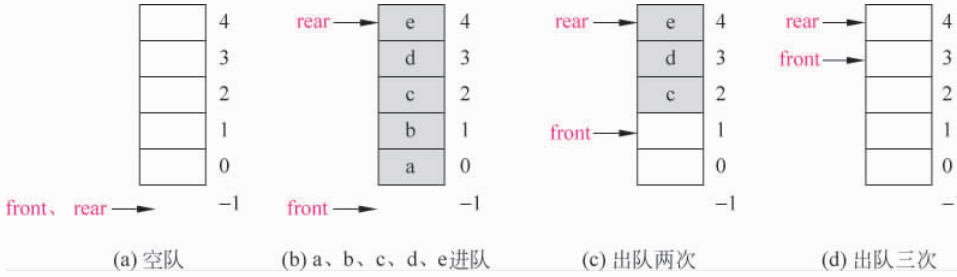

图 3.17　队列操作过程的示意图

综上所述，对于 q 所指的顺序队(即顺序队 q)，初始时设置 $q->rear=q->front=-1$，可以归纳出对后面算法设计来说非常重要的 4 个要素。

- 队空的条件：$q->front==q->rear$。
- 队满的条件：$q->rear==MaxSize-1$(data 数组的最大下标)。
- 元素 e 的进队操作：先将 rear 增 1，然后将元素 e 放在 data 数组的 rear 位置。
- 出队操作：先将 front 增 1，然后取出 data 数组中 front 位置的元素。

在顺序队上对应队列的基本运算算法设计如下。

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。算法如下：

```
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。算法如下：

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

3) 判断队列是否为空 QueueEmpty(q)

若队列 q 为空，返回真；否则返回假。算法如下：

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

4) 进队列 enQueue(&q,e)

在队列 q 不满的条件下先将队尾指针 rear 增 1，然后将元素 e 插入到该位置。算法如下：

```
bool enQueue(SqQueue * &q, ElemType e)
{   if (q -> rear==MaxSize-1)          //队满上溢出
        return false;                  //返回假
    q -> rear++;                       //队尾增 1
    q -> data[q -> rear]=e;            //rear 位置插入元素 e
    return true;                       //返回真
}
```

5）出队列 deQueue(&*q*,&*e*)

在队列 q 不空的条件下先将队头指针 front 增 1，并将该位置的元素值赋给 e。算法如下：

```
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;
}
```

上述 5 个基本运算算法的时间复杂度均为 $O(1)$。

2. 环形队中实现队列的基本运算

在前面的顺序队操作中，元素进队时队尾指针 rear 增 1，元素出队时队头指针 front 增 1，当队满的条件（即 rear==MaxSize-1）成立时，表示此时队满（上溢出）了，不能再进队元素。实际上，当 rear==MaxSize-1 成立时，队列中可能还有空位置，这种因为队满条件设置不合理导致队满条件成立而队列中仍然有空位置的情况称为**假溢出**（false overflow），图 3.17(c)所示就是假溢出的情况。

可以看出，在出现假溢出时队尾指针 rear 指向 data 数组的最大下标，而另外一端还有若干个空位置。解决的方法是把 data 数组的前端和后端连接起来，形成一个环形数组，即把存储队列元素的数组从逻辑上看成一个环，称为**环形队列**或者**循环队列**（circular queue）。

环形队列首尾相连后，当队尾指针 rear=MaxSize-1 后，再前进一个位置就到达 0，于是就可以使用另一端的空位置存放队列元素了。实际上存储器中的地址总是连续编号的，为此采用数学上的求余运算（%）来实现：

```
队头指针 front 循环增 1: front=(front+1)%MaxSize
队尾指针 rear 循环增 1: rear=(rear+1)%MaxSize
```

环形队列的队头指针 front 和队尾指针 rear 初始化时都置为 0，即 front=rear=0。在进队元素和出队元素时，队尾指针和队头指针分别循环增 1。

那么，环形队列 q 的队满和队空条件如何设置呢？显然队空条件是 q->rear==q->front。当进队元素的速度快于出队元素的速度时，队尾指针会回过来很快赶上队首指针，此时可以看出环形队列的队满条件也是 q->rear==q->front，也就是说无法仅通过这两个指针的当前位置区分开队空和队满。

那么怎样区分队空和队满呢？改为以“队尾指针循环增 1 时等于队头指针”作为队满条件，也就是说尝试进队一次，若达到队头，就认为队满了，不能再进队。这样环形队列少用一个元素空间，即该队列中在任何时刻最多只能有 MaxSize－1 个元素。

因此，在环形队列 q 中设置队空条件是 q－> rear＝＝q－> front；队满条件是（q－> rear＋1）％ MaxSize＝＝q－> front。而进队操作和出队操作与非环形队列的对应操作相同。

图 3.18 说明了环形队列操作的几种状态，这里假设 MaxSize 等于 5。

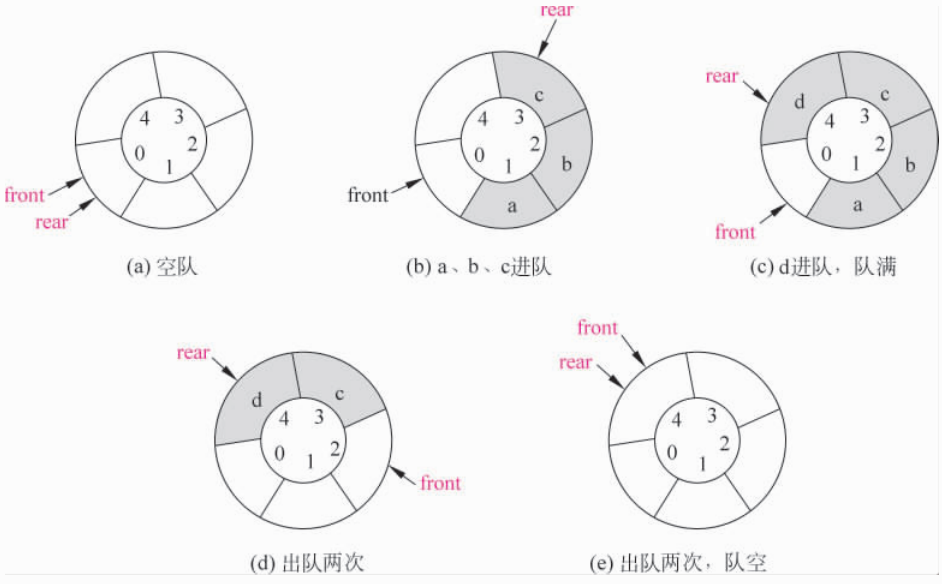

图 3.18　环形队列操作示意图

在这样设计的环形队列中，实现队列的基本运算算法如下。

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 0 值。算法如下：

```
void InitQueue(SqQueue * &q)
{   q=(SqQueue *)malloc (sizeof(SqQueue));
    q->front=q->rear=0;
}
```

2）销毁队列 DestroyQueue(&q)

释放队列 q 占用的存储空间。算法如下：

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

3）判断队列是否为空 QueueEmpty(q)

若队列为空返回真；否则返回假。算法如下：

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

4）进队列 enQueue(&q,e)

在队列不满的条件下先将队尾指针 rear 循环增 1，然后将元素插入到该位置。算法如下：

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

5）出队列 deQueue(&q,&e)

在队列 q 不空的条件下将队首指针 front 循环增 1，取出该位置的元素并赋给 e。算法如下：

```
bool deQueue(SqQueue * &q, ElemType &e)
{   if (q -> front==q -> rear)          //队空下溢出
        return false;
    q -> front=(q -> front+1)%MaxSize;
    e=q -> data[q -> front];
    return true;
}
```

同样，上述 5 个基本运算算法的时间复杂度均为 $O(1)$。

说明：在环形队列中，队头指针 front 指向队中队头元素的前一个位置，队尾指针 rear 指向队中的队尾元素，队列中的元素个数=(rear−front+MaxSize)%MaxSize。

需要说明的是，环形队列解决了假溢出现象，更充分地利用了队列空间。那么是不是在任何情况下都采用环形队列呢？答案是否定的。在环形队列中，随着多次元素的进队和出队，出队元素的空间可能被新进队的元素覆盖。在有些情况下，需要利用出队的元素来求解（从 data 数组角度看，只要未被覆盖，出队的元素仍在其中），例如用队列求解迷宫问题就属于这种情况，所以采用环形队列还是非环形队列根据实际求解问题来确定。

【例 3.7】 对于环形队列来说，如果知道队头指针和队列中的元素个数，则可以计算出队尾指针。也就是说，可以用队列中的元素个数代替队尾指针。设计出这种环形队列的初始化、进队、出队和判队空算法。

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;              //队列中的元素个数
} QuType;                   //本例的环形队列类型
```

当已知队列的队头指针 front 和队列中的元素个数 count 后，队尾指针 rear 的计算公式是 rear＝(front＋count)％MaxSize。因此，这种队列的队空条件为 count＝＝0；队满条件为 count＝＝MaxSize；元素 e 的进队操作是先根据队头指针和元素个数求出队尾指针 rear，将 rear 循环增 1，然后将元素 e 放置在 rear 处；出队操作是先将队头指针循环增 1，然后取出该位置的元素。

对应的算法如下：

```
void InitQueue(QuType * &qu)                          //初始化算法
{   qu=(QuType * )malloc(sizeof(QuType));
    qu -> front=0;                                    //队头指针设置为 0
    qu -> count=0;                                    //队列中的元素个数设置为 0
}
bool EnQueue(QuType * &qu,ElemType x)                 //进队算法
{   int rear;                                         //临时存放队尾指针
    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;                        //队尾指针循环增 1
        qu -> data[rear]=x;
        qu -> count++;                                //元素个数增 1
        return true;
    }
}
bool DeQueue(QuType * &qu,ElemType &x)                //出队算法
{   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)                          //判队空算法
{
    return(qu -> count==0);
}
```

注意：采用本例设计的环形队列中最多可以放置 MaxSize 个元素。

3.2.3 队列的链式存储结构及其基本运算的实现

队列中数据元素的逻辑关系呈线性关系，所以队列可以像线性表一样采用链式存储结

构。采用链式存储结构的队列称为**链队**(linked queue)。链表有多种,这里是采用单链表来实现链队的。

在这样的链队中只允许在单链表的表头进行删除操作(出队)和在单链表的表尾进行插入操作(进队),因此需要使用队头指针 front 和队尾指针 rear 两个指针,用 front 指向队首结点,用 rear 指向队尾结点。和链栈一样,链队中也不存在队满上溢出的情况。

链队存储结构如图 3.19 所示。链队中数据结点的类型 DataNode 声明如下:

```
typedef struct qnode
{    ElemType data;              //存放元素
     struct qnode * next;        //下一个结点指针
} DataNode;                      //链队数据结点的类型
```

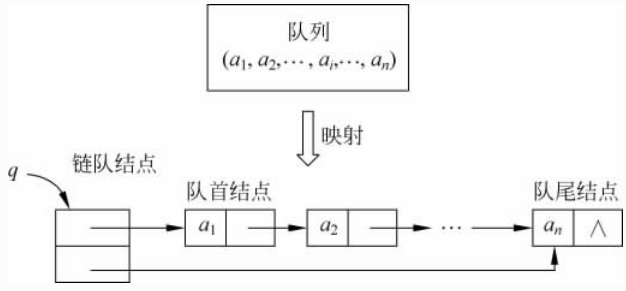

图 3.19 链队存储结构

链队头结点(或链队结点)的类型 LinkQuNode 声明如下:

```
typedef struct
{    DataNode * front;           //指向队首结点
     DataNode * rear;            //指向队尾结点
} LinkQuNode;                    //链队结点的类型
```

图 3.20 说明了一个链队 q 的动态变化过程。图 3.20(a)是链队的初始状态,图 3.20(b)是在链队中 3 个元素进队后的状态,图 3.20(c)是链队中一个元素出队后的状态。

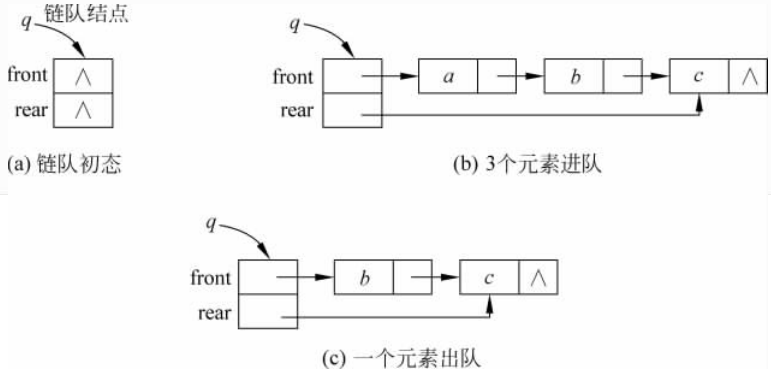

图 3.20 一个链队的动态变化过程

在以 q 为链队结点指针的链队(简称链队 q)中,可以归纳出对后面算法设计来说非常重要的 4 个要素。

- 队空的条件:q->rear==NULL(也可以为 q->front==NULL)。
- 队满的条件:不考虑。
- 元素 e 的进队操作:新建一个结点存放元素 e(由 p 指向它),将结点 p 插入作为尾结点。
- 出队操作:取出队首结点的 data 值并将其删除。

在链队上对应队列的基本运算算法设计如下。

1) 初始化队列 InitQueue(&q)

构造一个空队,即创建一个链队结点,其 front 和 rear 域均置为 NULL。算法如下:

```
void InitQueue(LinkQuNode *&q)
{   q=(LinkQuNode *)malloc(sizeof(LinkQuNode));
    q->front=q->rear=NULL;
}
```

本算法的时间复杂度为 $O(1)$。

2) 销毁队列 DestroyQueue(&q)

释放链队占用的全部存储空间,包括链队结点和所有数据结点的存储空间。算法如下:

```
void DestroyQueue(LinkQuNode *&q)
{   DataNode *pre=q->front, *p;          //pre 指向队首结点
    if (pre!=NULL)
    {   p=pre->next;                     //p 指向结点 pre 的后继结点
        while (p!=NULL)                  //p 不空循环
        {   free(pre);                   //释放 pre 结点
            pre=p;p=p->next;             //pre、p 同步后移
        }
        free(pre);                       //释放最后一个数据结点
    }
    free(q);                             //释放链队结点
}
```

本算法的时间复杂度为 $O(n)$,其中 n 为链队中数据结点的个数。

3) 判断队列是否为空 QueueEmpty(q)

若链队为空,返回真;否则返回假。算法如下:

```
bool QueueEmpty(LinkQuNode *q)
{
    return(q->rear==NULL);
}
```

本算法的时间复杂度为 $O(1)$。

4) 进队列 enQueue(&q,e)

创建一个新结点用于存放元素 e(由 p 指向它)。若原队列为空,则将链队结点的两个域均指向结点 p,否则将结点 p 链接到单链表的末尾,并让链队结点的 rear 域指向它。算法如下:

```
void enQueue(LinkQuNode * &q, ElemType e)
{   DataNode * p;
    p=(DataNode * )malloc(sizeof(DataNode));   //创建新结点
    p -> data=e;
    p -> next=NULL;
    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=p;
    else                                       //若链队不空
    {   q -> rear -> next=p;                   //将结点 p 链到队尾,并将 rear 指向它
        q -> rear=p;
    }
}
```

本算法的时间复杂度为 $O(1)$。

5）出队列 deQueue(&q,&e)

若原队列为空,则下溢出返回假；若原队列不空,则将首结点的 data 域值赋给 e,并删除之,若原队列只有一个结点,则需将链队结点的两个域均置为 NULL,表示队列已为空。算法如下：

```
bool deQueue(LinkQuNode * &q, ElemType &e)
{   DataNode * t;
    if (q -> rear==NULL)                       //原来队列为空
        return false;
    t=q -> front;                              //t 指向首结点
    if (q -> front==q -> rear)                 //原来队列中只有一个数据结点时
        q -> front=q -> rear=NULL;
    else                                       //原来队列中有两个或两个以上结点时
        q -> front=q -> front -> next;
    e=t -> data;
    free(t);
    return true;
}
```

本算法的时间复杂度为 $O(1)$。

【例 3.8】 采用一个不带头结点只有一个尾结点指针 rear 的循环单链表存储队列,设计队列的初始化、进队和出队算法。

解 本例的链队如图 3.21 所示,用只有尾结点指针 rear 的循环单链表作为队列存储结构,其中每个结点的类型为 LinkNode(LinkNode 为单链表结点类型,在第 2 章中已声明),rear 指针用于唯一标识链队,对应链队的 4 个要素如下。

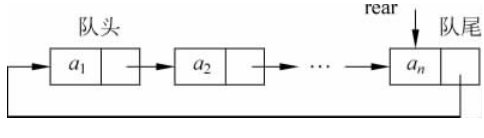

图 3.21 用只有尾结点指针的循环单链表作为队列存储结构

- 队空的条件：rear==NULL。
- 队满的条件：不考虑。
- 元素 e 的进队操作：新建一个结点存放元素 e(由 p 指向它)，将结点 p 插入作为尾结点，让 rear 指向这个新的尾结点。
- 出队操作：取出队头结点(rear 所指结点的后继结点)的 data 值并将其删除。

需要注意的是，在该链队进队和出队操作后链队或者为空，或者为一个不带头结点的由尾结点指针 rear 唯一标识的循环单链表，不能改变其结构特性。

对应的队列基本运算算法如下：

```
void initQueue(LinkNode * &rear)                    //初始化算法
{
    rear=NULL;
}
void enQueue(LinkNode * &rear, ElemType e)          //进队算法
{   LinkNode * p;
    p=(LinkNode * )malloc(sizeof(LinkNode));        //创建新结点
    p->data=e;
    if (rear==NULL)                                 //原链队为空
    {   p->next=p;                                  //改为循环链表
        rear=p;                                     //rear 指向新结点
    }
    else                                            //原链队不空
    {   p->next=rear->next;                         //将 p 结点插入到 rear 结点之后
        rear->next=p;                               //改为循环链表
        rear=p;                                     //rear 指向新结点
    }
}
bool deQueue(LinkNode * &rear, ElemType &e)         //出队算法
{   LinkNode * t;
    if (rear==NULL)                                 //队空
        return false;
    else if (rear->next==rear)                      //原队中只有一个结点
    {   e=rear->data;
        free(rear);
        rear=NULL;                                  //让 rear 为空链表
    }
    else                                            //原队中有两个或两个以上的结点
    {   t=rear->next;                               //t 指向队头结点
        e=t->data;
        rear->next=t->next;                         //删除 t 结点
        free(t);                                    //释放结点空间
    }
    return true;
}
bool queueEmpty(LinkNode * rear)                    //判队空算法
```

```
{
    return(rear==NULL);
}
```

3.2.4 队列的应用举例

在实际应用中，队列通常作为一种存放临时数据的容器。如果先存入的元素先处理，则采用队列。本小节通过报数问题和迷宫问题的求解过程介绍队列的应用。

1. 求解报数问题

1）问题描述

设有 n 个人站成一排，从左向右的编号分别为 1～n，现在从左往右报数“1，2，1，2，…”，数到“1”的人出列，数到“2”的立即站到队伍的最右端。报数过程反复进行，直到 n 个人都出列为止。要求给出他们的出列顺序。

例如，当 n=8 时初始序列为：

```
1  2  3  4  5  6  7  8
```

则出列顺序为：

```
1  3  5  7  2  6  4  8
```

2）数据组织

用一个队列解决出列问题，由于这里不需要使用已经出队后的元素，所以采用环形队列。

3）设计运算算法

采用的算法思想是先将 n 个人的编号进队，然后反复执行以下操作，直到队列为空。

（1）出队一个元素，输出其编号（报数为 1 的人出列）。

（2）若队列不空，再出队一个元素，并将刚出列的元素进队（报数为 2 的人站到队伍的最右端，即队尾）。

对应的算法如下：

```
void number(int n)
{   int i; ElemType e;
    SqQueue *q;                    //环形队列指针 q
    InitQueue(q);                  //初始化队列 q
    for (i=1;i<=n;i++)             //构建初始序列
        enQueue(q,i);
    printf("报数出列顺序:");
    while (!QueueEmpty(q))         //队列不空循环
    {   deQueue(q,e);              //出队一个元素 e
        printf("%d ",e);           //输出元素编号
        if (!QueueEmpty(q))        //队列不空
```

```
            {   deQueue(q,e);                    //出队一个元素 e
                enQueue(q,e);                    //将刚出队的元素进队
            }
        }
        printf("\n");
        DestroyQueue(q);                         //销毁队列 q
}
```

4）设计求解程序

设计一个主函数调用上述算法：

```
int main()
{   int i,n=8;
    printf("初始序列:");
    for (i=1;i<=n;i++)
        printf("%d ",i);
    printf("\n");
    number(n);
    return 1;
}
```

5）运行结果

上述程序的运行结果如下：

```
初始序列: 1 2 3 4 5 6 7 8
报数出列顺序: 1 3 5 7 2 6 4 8
```

2. 求解迷宫问题

1）问题描述

参见 3.1.4 节的问题描述。

2）数据组织

用队列解决求迷宫路径问题。使用一个顺序队 qu 保存走过的方块，该队列的类型声明如下：

```
typedef struct
{   int i,j;                    //方块的位置
    int pre;                    //本路径中上一个方块在队列中的下标
} Box;                          //方块类型
typedef struct
{   Box data[MaxSize];
    int front,rear;             //队头指针和队尾指针
} QuType;                       //顺序队类型
```

这里使用的顺序队列 qu 不是环形队列，因为在找到出口时需要利用队列中的所有方块查找一条迷宫路径。如果采用环形队列，出队的方块可能被新进队的方块覆盖，从而无法求

出迷宫路径。这里要求非环形队列 qu 有足够大的空间。

3）设计运算算法

搜索从入口(xi,yi)到出口(xe,ye)路径的过程是,首先将入口(xi,yi)进队,在队列 qu 不为空时循环,出队一个方块 e(由于不是环形队列,该出队方块不会被覆盖,其下标为 front)。然后查找方块 e 的所有相邻可走方块,假设为 e_1 和 e_2 两个方块,将它们进队,它们在队列中的位置分别为 rear1 和 rear2,并且将它们的 pre 均设置为 front(因为在迷宫路径上 e_1 和 e_2 两个方块的前一个方块都是方块 e),如图 3.22 所示。

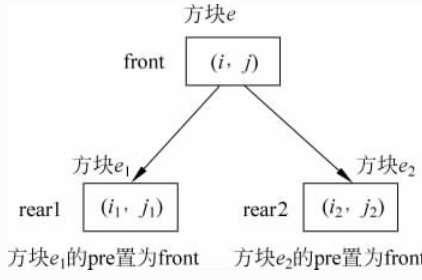

图 3.22　设置相邻方块的 pre

当找到出口时,通过出口方块的 pre 值前推找到出口,所有经过的中间方块构成一条迷宫路径。对应的完整过程如下:

```
将入口(xi,yi)的 pre 置为-1 并进队;
mg[xi][yi]=-1;
while (队列 qu 不空)
{   出队一个方块 e,其在队列中的位置是 front;
    if (方块 e 是出口)
    {   输出一条迷宫路径;
        return true;
    }
    for (对于方块 e 的所有相邻可走方块 e1)
    {   设置 e1 的 pre 为 front;
        将方块 e1 进队;
        将方块 e1 的迷宫数组值设置为-1;
    }
}
return false;                    //没有迷宫路径,返回假
```

实际上,上述过程是从入口(xi,yi)开始,利用队列的特点,一层一层向外扩展查找可走的方块,直到找到出口为止,这个方法就是将在第 8 章介绍的广度优先搜索方法。

在找到出口后,输出路径的过程是根据当前方块(即出口,其在队列 qu 中的下标为 front)的 pre 值回推找到迷宫路径。对于图 3.11 所示的迷宫,在找到出口后,队列 qu 中 data 的全部数据如表 3.3 所示。当前的 front=40,qu->data[40].pre 为 35,表示路径上的前一个方块为 qu->data[35];而 qu->data[35].pre 为 30,表示路径的上一个方块为 qu->data[30];qu->data[30].pre 为 27,表示路径上的前一个方块为 qu->data[27],…,以此类推,找到入口为 qu->data[0]。在对应的 print 函数中,为了正向输出路径,在前面的

回推过程中修改路径上每个方块的 pre 值，使该迷宫路径上的所有方块的 pre 值置为－1，然后从开头输出所有 pre 为－1 的方块，从而正向输出了一条迷宫路径。

表 3.3　队列 qu 中 data 的全部数据

下标	*i*	*j*	pre	下标	*i*	*j*	pre
0	1	1	－1	21	1	6	18
1	1	2	0	22	6	5	20
2	2	1	0	23	5	5	22
3	2	2	1	24	7	5	22
4	3	1	2	25	4	5	23
5	3	2	3	26	5	6	23
6	4	1	4	27	8	5	24
7	3	3	5	28	4	6	25
8	5	1	6	29	5	7	26
9	3	4	7	30	8	6	27
10	5	2	8	31	8	4	27
11	6	1	8	32	4	7	28
12	2	4	9	33	5	8	29
13	5	3	10	34	6	7	29
14	7	1	11	35	8	7	30
15	1	4	12	36	8	3	31
16	2	5	12	37	3	7	32
17	6	3	13	38	4	8	32
18	1	5	15	39	6	8	33
19	2	6	16	40	8	8	35
20	6	4	17				

根据上述搜索过程得到以下用队列求解迷宫的算法：

```
bool mgpath1(int xi,int yi,int xe,int ye)   //搜索路径为(xi,yi)->(xe,ye)
{   Box e;
    int i,j,di,i1,j1;
    QuType * qu;                    //定义顺序队指针 qu
    InitQueue(qu);                  //初始化队列 qu
    e.i=xi; e.j=yi; e.pre=-1;
    enQueue(qu,e);                  //(xi,yi)进队
    mg[xi][yi]=-1;                  //将其赋值-1,以避免回过来重复搜索
    while (!QueueEmpty(qu))         //队不空循环
    {   deQueue(qu,e);              //出队方块 e,由于不是环形队列,该出队元素仍在队列中
        i=e.i; j=e.j;
        if (i==xe && j==ye)         //找到了出口,输出路径
        {   print(qu,qu->front);    //调用 print 函数输出路径
            DestroyQueue(qu);       //销毁队列
            return true;            //找到一条路径时返回真
        }
        for (di=0;di<4;di++)        //循环扫描每个方位,把每个可走的方块插入到队列中
```

```
            {   switch(di)
                {
                case 0:i1=i-1; j1=j; break;
                case 1:i1=i; j1=j+1; break;
                case 2:i1=i+1; j1=j; break;
                case 3:i1=i; j1=j-1; break;
                }
                if (mg[i1][j1]==0)
                {   e.i=i1; e.j=j1;
                    e.pre=qu->front;          //指向路径中上一个方块的下标
                    enQueue(qu,e);            //(i1,j1)方块进队
                    mg[i1][j1]=-1;            //将其赋值-1,以避免回过来重复搜索
                }
            }
        }
        DestroyQueue(qu);                     //销毁队列
        return false;                         //未找到任何路径时返回假
}
void print(QuType *qu,int front)              //从队列 qu 中输出一条迷宫路径
{   int k=front,j,ns=0;
    printf("\n");
    do                                  //反向找到最短路径,将该路径上的方块的 pre 成员设置成-1
    {   j=k;
        k=qu->data[k].pre;
        qu->data[j].pre=-1;
    } while (k!=0);
    printf("一条迷宫路径如下:\n");
    k=0;
    while (k<MaxSize)                         //正向搜索到 pre 为-1 的方块,即构成正向的路径
    {   if (qu->data[k].pre==-1)
        {   ns++;
            printf("\t(%d,%d)",qu->data[k].i,qu->data[k].j);
            if (ns%5==0) printf("\n");  //每输出 5 个方块后换一行
        }
        k++;
    }
    printf("\n");
}
```

4）设计求解程序

建立以下主函数调用上述算法：

```
int main()
{   if (!mgpath1(1,1,M,N))
        printf("该迷宫问题没有解!");
    return 1;
}
```

5）运行结果

对于图 3.10 所示的迷宫，求解结果如下：

```
一条迷宫路径如下：
    (1,1)  (2,1)  (3,1)  (4,1)  (5,1)
    (5,2)  (5,3)  (6,3)  (6,4)  (6,5)
    (7,5)  (8,5)  (8,6)  (8,7)  (8,8)
```

上述迷宫路径的显示结果如图 3.23 所示，图中路径上方块(i,j)中的箭头指向路径的前一个相邻方块，例如方块(2,1)的箭头是“↑”，表示路径上前一个方块是方块(1,1)，即出口。显然这个解是最优解，也就是最短路径。

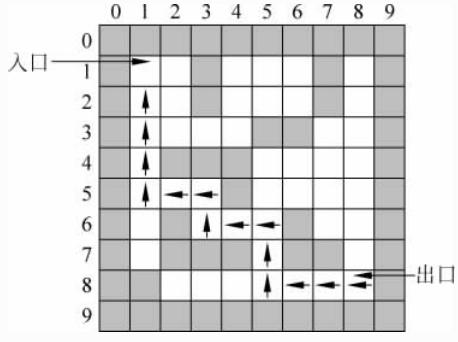

图 3.23　用队列求解的迷宫路径

3.2.5　双端队列

所谓**双端队列**(deque，double-ended queue)是指两端都可以进行进队和出队操作的队列，如图 3.24 所示。将队列的两端分别称为前端和后端，两端都可以进队和出队。其元素的逻辑关系仍是线性关系。

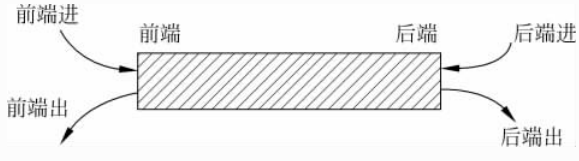

图 3.24　一个双端队列

在双端队列中进队时，前端进的元素排列在队列中后端进的元素的前面，后端进的元素排列在队列中前端进的元素的后面。在双端队列中出队时，无论前端出还是后端出，先出的元素排列在后出的元素的前面。

例如有 a、b、c、d 元素进队，能否产生 dcab 的出队序列呢？答案是肯定的。其操作方式为 a 后端进，b 后端进，c 前端进，d 前端进，再全部从前端出队，便可以得到这样的出队序列。那么是不是可以通过双端队列得到任意次序的出队序列呢？答案是否定的，例如 a、b、c、d 元素进队就不能产生 dacb 的出队序列，因为 a 进队，b 从后端进，c 无论从前端进还是从后端进都不可能在 a、b 的中间。

实际上，从前面的双端队列可以看出，从后端进前端出或者从前端进后端出体现先进先出的特点，从前端进前端出或从后端进后端出体现出后进先出的特点。

在实际使用中，还可以有输出受限的双端队列（即允许两端进队，但只允许一端出队的双端队列）和输入受限的双端队列（即允许两端出队，但只允许一端进队的双端队列），前者如图 3.25 所示，后者如图 3.26 所示。如果限定双端队列中从某端进队的元素只能从该端出队，则该双端队列就蜕变为两个栈底相邻的栈了。

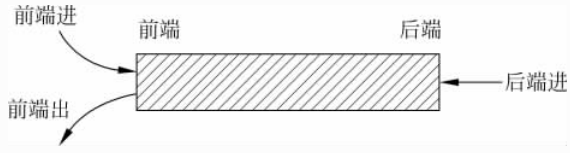

图 3.25　一个输出受限的双端队列

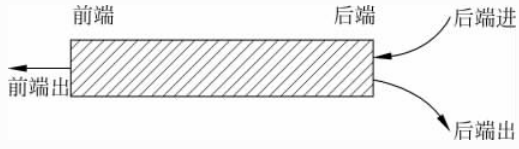

图 3.26　一个输入受限的双端队列

【例 3.9】 某队列允许在两端进行入队操作，但仅允许在一端进行出队操作，若 a、b、c、d、e 元素进队，则以下不可能得到的顺序有哪些？

（1）bacde　（2）dbace　（3）dbcae　（4）ecbad

解 本题的队列实际上是一个输出受限的双端队列，这样的双端队列如图 3.25 所示。

（1）a 后端进，b 前端进，c 后端进，d 后端进，e 后端进，全出队。

（2）a 后端进，b 前端进，c 后端进，d 前端进，e 后端进，全出队。

（3）a 后端进，b 前端进，因 d 未出，此时只能进队，c 怎么进都不可能在 b、a 之间。

（4）a 后端进，b 前端进，c 前端进，d 后端进，e 前端进，全出队。

所以不可能得到的顺序为（3）。

【例 3.10】 如果允许在环形队列的两端进行插入和删除操作（这样的队列即为双端队列），若仍采用前面定义的 SqQueue 队列类型，设计“从队尾删除”和“从队头插入”的算法。

解 从前面介绍的环形队列结构可以看到，队头指针 front 指向队列中队首元素的前一个位置，而队尾指针 rear 指向队列中的队尾元素。所以“从队尾删除”运算应先提取队尾元素，再循环后退一个位置，而“从队头插入”运算应先在队头插入元素，再循环后退一个位置。

实现“从队尾删除”运算的算法如下：

```
bool deQueue1(SqQueue *&q,ElemType &e)                    //从队尾删除算法
{   if (q->front==q->rear)                                //队空返回假
        return false;
    e=q->data[q->rear];                                   //提取队尾元素
    q->rear=(q->rear-1+MaxSize)%MaxSize;                  //修改除尾指针
```

```
    return true;
}
```

实现“从队头插入”运算的算法如下：

```
bool enQueue1(SqQueue *&q,ElemType e)                    //从队头插入算法
{   if ((q->rear+1)%MaxSize==q->front)                   //队满返回假
        return false;
    q->data[q->front]=e;                                 //元素 e 进队
    q->front=(q->front-1+MaxSize)%MaxSize;               //修改队头指针
    return true;
}
```

本章小结

本章的基本学习要点如下：

(1) 理解栈和队列的特性以及它们之间的差异。

(2) 掌握栈的两类存储结构(即顺序栈和链栈)的设计特点，注意顺序栈和链栈中栈满和栈空的条件判断。

(3) 掌握在顺序栈和链栈中实现栈的基本运算的算法设计方法。

(4) 掌握队列的两类存储结构(即顺序队和链队)的设计特点，环形队列和非环形队列的差异，注意各种存储结构中队满和队空的条件判断。

(5) 掌握在顺序队和链队中实现队列的基本运算的算法设计方法。

(6) 理解栈和队列的作用，知道在何时使用哪一种数据结构，用栈和队列求解迷宫问题的差异。

(7) 灵活地运用栈和队列两种数据结构解决一些综合应用问题。

练习题 3

1. 有 5 个元素，其进栈次序为 A、B、C、D、E，在各种可能的出栈次序中以元素 C、D 最先出栈(即 C 第一个且 D 第二个出栈)的次序有哪几个？

2. 在一个算法中需要建立多个栈(假设 3 个栈或以上)时可以选用以下 3 种方案之一，试问这些方案相比各有什么优缺点？

(1) 分别用多个顺序存储空间建立多个独立的顺序栈。

(2) 多个栈共享一个顺序存储空间。

(3) 分别建立多个独立的链栈。

3. 在以下几种存储结构中哪个最适合用作链栈？

(1) 带头结点的单链表。

(2) 不带头结点的循环单链表。

(3) 带头结点的双链表。

4. 简述以下算法的功能(假设 ElemType 为 int 类型)。

```
void fun(ElemType a[],int n)
{   int i; ElemType e;
    SqStack *st1,*st2;
    InitStack(st1);
    InitStack(st2);
    for (i=0;i<n;i++)
        if (a[i]%2==1)
            Push(st1,a[i]);
        else
            Push(st2,a[i]);
    i=0;
    while (!StackEmpty(st1))
    {   Pop(st1,e);
        a[i++]=e;
    }
    while (!StackEmpty(st2))
    {   Pop(st2,e);
        a[i++]=e;
    }
    DestroyStack(st1);
    DestroyStack(st2);
}
```

5. 简述以下算法的功能(顺序栈的元素类型为 ElemType)。

```
void fun(SqStack *&st,ElemType x)
{   SqStack *tmps;
    ElemType e;
    InitStack(tmps);
    while(!StackEmpty(st))
    {   Pop(st,e);
        if(e!=x) Push(tmps,d);
    }
    while (!StackEmpty(tmps))
    {   Pop(tmps,e);
        Push(st,e);
    }
    DestroyStack(tmps);
}
```

6. 简述以下算法的功能(栈 st 和队列 qu 的元素类型均为 ElemType)。

```
bool fun(SqQueue *&qu,int i)
{   ElemType e; int j=1;
    int n=(qu->rear-qu->front+MaxSize)%MaxSize;
    if (j<1 || j>n) return false;
    for (j=1;j<=n;j++)
    {   deQueue(qu,e);
        if (j!=i)
```

```
            enQueue(qu,e);
        }
        return true;
    }
```

7. 什么是环形队列？采用什么方法实现环形队列？

8. 环形队列一定优于非环形队列吗？在什么情况下使用非环形队列？

9. 假设以 I 和 O 分别表示进栈和出栈操作，栈的初态和终栈均为空，进栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列。

(1) 在下面所示的序列中哪些是合法的？

A. IOIIOIOO　　B. IOOIOIIO　　C. IIIOIOIO　　D. IIIOOIOO

(2) 通过对(1)的分析，设计一个算法判定所给的操作序列是否合法，若合法返回真，否则返回假(假设被判定的操作序列已存入一维数组中)。

10. 假设表达式中允许包含圆括号、方括号和大括号 3 种括号，编写一个算法判断表达式中的括号是否正确配对。

11. 设从键盘输入一序列的字符 $a_1,a_2,\cdots,a_n$。设计一个算法实现这样的功能：若 a_i 为数字字符，a_i 进队；若 a_i 为小写字母，将队首元素出队；若 a_i 为其他字符，表示输入结束。要求使用环形队列。

12. 设计一个算法，将一个环形队列(容量为 n，元素下标从 0 到 $n-1$)的元素倒置。例如，图 3.27(a)中为倒置前的队列($n=10$)，图 3.27(b)中为倒置后的队列。

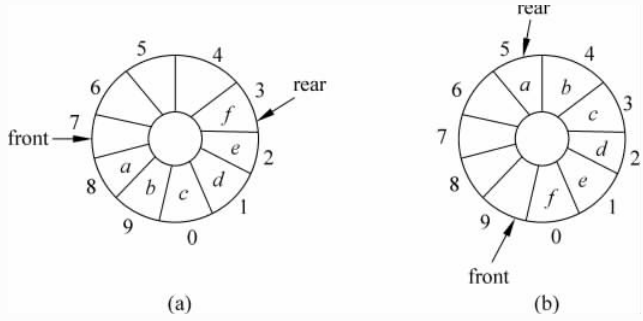

图 3.27　一个环形队列倒置前后的状态

13. 编写一个程序，输入 n(由用户输入)个 10 以内的数，每输入 $i(0\leqslant i\leqslant 9)$就把它插入到第 i 号队列中，最后把 10 个队中的非空队列按队列号从小到大的顺序串接成一条链，并输出该链的所有元素。

上机实验题 3

验证性实验

实验题 1：实现顺序栈的各种基本运算的算法

目的：领会顺序栈存储结构和掌握顺序栈中的各种基本运算算法设计。

内容：编写一个程序 sqstack.cpp，实现顺序栈（假设栈中元素类型 ElemType 为 char）的各种基本运算，并在此基础上设计一个程序 exp3-1.cpp 完成以下功能。

（1）初始化栈 s。

（2）判断栈 s 是否非空。

（3）依次进栈元素 a、b、c、d、e。

（4）判断栈 s 是否非空。

（5）输出出栈序列。

（6）判断栈 s 是否非空。

（7）释放栈。

实验题 2：实现链栈的各种基本运算的算法

目的：领会链栈存储结构和掌握链栈中的各种基本运算算法设计。

内容：编写一个程序 listack.cpp，实现链栈（假设栈中元素类型 ElemType 为 char）的各种基本运算，并在此基础上设计一个程序 exp3-2.cpp 完成以下功能。

（1）初始化栈 s。

（2）判断栈 s 是否非空。

（3）依次进栈元素 a、b、c、d、e。

（4）判断栈 s 是否非空。

（5）输出出栈序列。

（6）判断栈 s 是否非空。

（7）释放栈。

实验题 3：实现环形队列的各种基本运算的算法

目的：领会环形队列存储结构和掌握环形队列中的各种基本运算算法设计。

内容：编写一个程序 sqqueue.cpp，实现环形队列（假设栈中元素类型 ElemType 为 char）的各种基本运算，并在此基础上设计一个程序 exp3-3.cpp 完成以下功能。

（1）初始化队列 q。

（2）判断队列 q 是否非空。

（3）依次进队元素 a、b、c。

（4）出队一个元素，输出该元素。

（5）依次进队元素 d、e、f。

（6）输出出队序列。

（7）释放队列。

实验题 4：实现链队的各种基本运算的算法

目的：领会链队存储结构和掌握链队中的各种基本运算算法设计。

内容：编写一个程序 liqueue.cpp，实现链队（假设栈中元素类型 ElemType 为 char）的各种基本运算，并在此基础上设计一个程序 exp3-4.cpp 完成以下功能。

（1）初始化链队 q。

（2）判断链队 q 是否非空。

（3）依次进链队元素 a、b、c。

(4) 出队一个元素,输出该元素。

(5) 依次进链队元素 d、e、f。

(6) 输出出队序列。

(7) 释放链队。

设计性实验

实验题 5:用栈求解迷宫问题的所有路径及最短路径程序

目的:掌握栈在求解迷宫问题中的应用。

内容:编写一个程序 exp3-5.cpp,改进 3.1.4 节的求解迷宫问题程序,要求输出图 3.28 所示的迷宫的所有路径,并求第一条最短路径长度及最短路径。

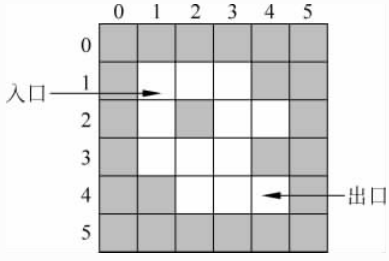

图 3.28 迷宫示意图

实验题 6:编写病人看病模拟程序

目的:掌握队列应用的算法设计。

内容:编写一个程序 exp3-6.cpp,反映病人到医院排队看医生的情况。在病人排队过程中主要重复下面两件事。

(1) 病人到达诊室,将病历本交给护士,排到等待队列中候诊。

(2) 护士从等待队列中取出下一位病人的病历,该病人进入诊室就诊。

要求模拟病人等待就诊这一过程。程序采用菜单方式,其选项及功能说明如下:

1:排队——输入排队病人的病历号,加入到病人排队队列中;

2:就诊——病人排队队列中最前面的病人就诊,并将其从队列中删除;

3:查看排队——从队首到队尾列出所有的排队病人的病历号;

4:不再排队,余下依次就诊——从队首到队尾列出所有的排队病人的病历号,并退出运行。

5:下班——退出运行。

实验题 7:求解栈元素排序问题

目的:掌握栈应用的算法设计。

内容:编写一个程序 exp3-7.cpp,按升序对一个字符栈进行排序,即最小元素位于栈顶,最多只能使用一个额外的栈存放临时数据,并输出栈排序过程。

综合性实验

实验题 8:用栈求解 n 皇后问题

目的:深入掌握栈应用的算法设计。

内容:编写一个程序 exp3-8.cpp 求解 n 皇后问题,即在 $n\times n$ 的方格棋盘上放置 n 个皇后,要求每个皇后不同行、不同列、不同左右对角线,图 3.29 是八皇后问题的一个解。(1)皇后个数 n 由用户输入,其值不能超过 20,输出所有的解(2)采用类似于用栈求解迷宫问题的方法。

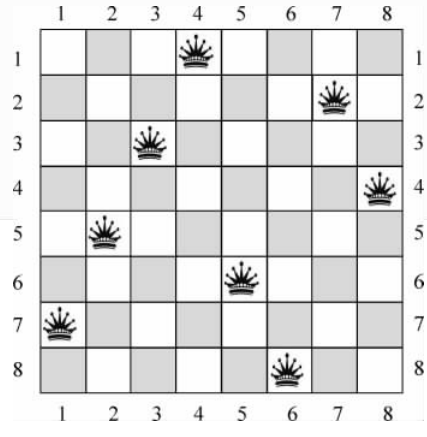

图 3.29 八皇后问题

实验题 9：编写停车场管理程序

目的：深入掌握栈和队列应用的算法设计。

内容：编写满足以下要求的停车场管理程序 exp3-9.cpp。设停车场内只有一个可停放 n 辆汽车的狭长通道，且只有一个大门可供汽车进出。

汽车在停车场内按车辆到达时间的先后顺序依次由南向北排列(大门在最北端，最先到达的第一辆车停放在停车场的最南端)，若停车场内已停满 n 辆车，则后来的汽车只能在门外的便道(即候车场上)等候，一旦有车开走，则排在便道上的第一辆车即可开入；当停车场内某辆车要离开时，在它之后进入的车辆必须先退出停车场为它让路，待该辆车开出大门外，其他车辆再按原次序进入停车场，每辆停放在停车场的车在它离开停车场时必须按停留的时间长短交纳费用。整个停车场的示意图如图 3.30 所示。

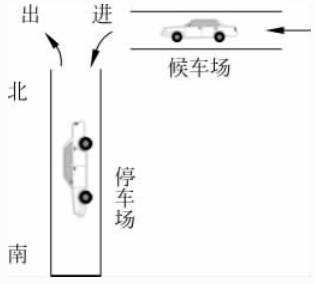

图 3.30　停车场示意图