CHAPTER 3

第3章 栈和队列

栈和队列是两种常用的数据结构，它们的数据元素的逻辑关系也是线性关系，但在运算上不同于线性表。本章介绍栈和队列的基本概念、存储结构以及相关运算的实现过程。

3.1 栈

本节先介绍栈的定义，接着介绍栈的存储结构和基本运算算法设计，最后通过实例讨论栈的应用。

3.1.1 栈的定义

先看一个例子，假设有一个老鼠洞，口径只能容纳一只老鼠，有若干只老鼠依次进洞，如图 3.1 所示，当到达洞底时，这些老鼠只能一只一只按原来进洞时相反的次序出洞，如图 3.2 所示。在这个例子中，老鼠洞就是一个栈，由于其口径只能容纳一只老鼠，所以不论洞中有多少只老鼠，它们只能一只一只地排列，从而构成一种线性关系。再看老鼠洞的主要操作，显然有进洞和出洞两种操作，且进洞只能从洞口进，出洞只能从洞口出。

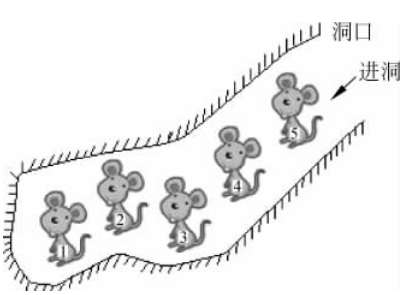

图 3.1 老鼠进洞的情况

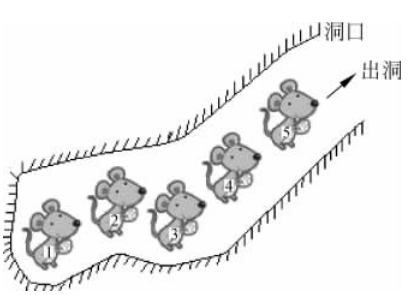

图 3.2 老鼠出洞的情况

抽象起来，栈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**。栈顶的当前位置是动态的，由一个

被称为栈顶指针的位置指示器来指示。表的另一端称为**栈底**，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**，栈的删除操作通常称为**退栈**或**出栈**。

说明：对于线性表，可以在中间和两端任何地方插入和删除元素，而栈只能在同一端插入和删除元素。

栈的主要特点是“后进先出”，即后进栈的元素先出。每次进栈的数据元素都放在原当前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原当前栈顶元素。栈也称为**后进先出表**。

图 3.3 是一个栈的动态示意图，图中箭头表示当前栈顶元素位置。图 3.3(a)表示一个空栈；图 3.3(b)表示数据元素 a 进栈以后的状态；图 3.3(c)表示数据元素 b、c、d 进栈以后的状态；图 3.3(d)表示一个数据元素出栈以后的状态。

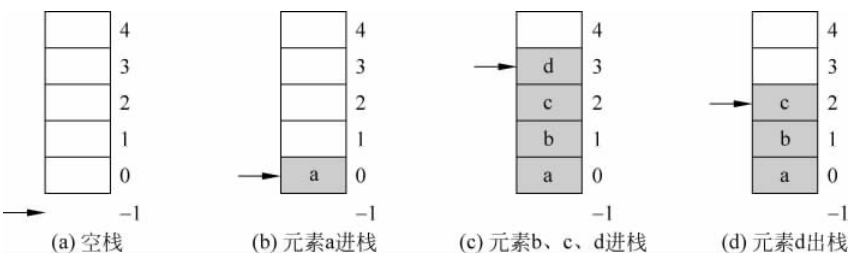

图 3.3　栈的动态示意图

抽象数据类型栈的定义如下：

```
ADT Stack
{
数据对象:
    D = {ai | 1≤i≤n,n≥0,ai 为 T 类型}
数据关系:
    R = {r}
    r = {<ai,ai+1> | ai,ai+1∈D, i = 1,…,n − 1}
基本运算:
    bool StackEmpty(string e)          //判断栈是否为空,若为空返回真,否则返回假
    bool Push(string e)                //进栈,将元素 e 插入到栈中作为栈顶元素
    bool Pop(ref string e)             //出栈,从栈中退出栈顶元素,并将其值赋给 e
    GetTop(ref string 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】 若元素的进栈顺序为 1234，能否得到 3142 的出栈顺序？

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 的出栈顺序。

【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 的出栈顺序，给出相应的 S 和 X 操作串。

解：为了得到 1342 的出栈顺序，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。

3.1.2 栈的顺序存储结构及其基本运算的实现

栈可以采用顺序存储结构,分配一块连续的存储空间 data(大小为常量 MaxSize)来存放栈中元素,并用一个变量 top(栈顶指针)指向当前的栈顶以反映栈中元素的变化。采用顺序存储的栈称为顺序栈。

和顺序表一样,顺序栈类模板 SqStackClass<T>的设计如下:

```
template <typename T>
class SqStackClass                    //顺序栈类模板
{   T *data;                          //存放栈中元素
    int top;                          //栈顶指针
public:
    SqStackClass();                   //构造函数
    ~SqStackClass();                  //析构函数
    bool StackEmpty();                //判断栈是否为空
    bool Push(T e);                   //进栈算法
    bool Pop(T &e);                   //出栈算法
    bool GetTop(T &e);                //取栈顶元素算法
};
```

顺序栈存储结构示意图如图 3.4 所示,带底纹的元素为栈中元素。初始时置栈顶指针 top=-1,栈空的条件为 top==-1;栈满的条件为 top==MaxSize-1。元素 e 的进栈操作是先将栈顶指针增 1,然后将元素 e 放在栈顶指针处;出栈操作是先将栈顶指针处的元素取出,然后将栈顶指针减 1。

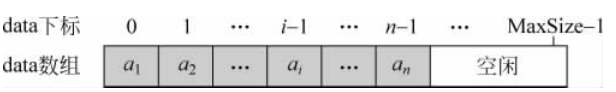

图 3.4 顺序栈存储结构示意图

顺序栈的基本运算算法如下。

(1) 顺序栈的初始化和销毁

顺序栈的初始化和销毁分别通过构造函数和析构函数来实现,对应的算法如下:

```
template <typename T>
SqStackClass<T>::SqStackClass()           //构造函数
{   data = new T[MaxSize];                //为 data 分配栈空间
    top = -1;                             //栈顶指针初始化
}
template <typename T>
SqStackClass<T>::~SqStackClass()          //析构函数
{   delete [] data; }                     //释放 data 占用的空间
```

(2) 判断栈是否为空:StackEmpty()

若栈顶指针 top 为-1,表示空栈。其对应的算法如下:

```
template <typename T>
bool SqStackClass<T>::StackEmpty()        //判断栈是否为空
```

```
{    return(top == -1); }
```

(3) 进栈：Push(*e*)

元素进栈只能从栈顶进，不能从栈底或中间位置进，如图 3.5 所示。

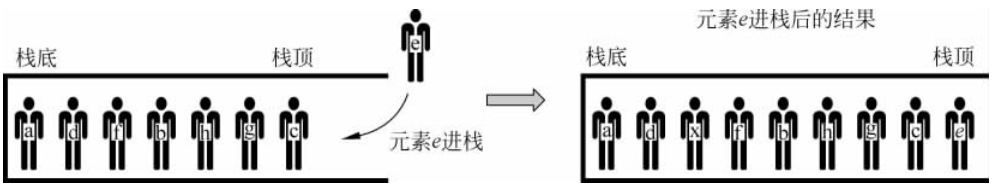

图 3.5　元素进栈示意图

在进栈运算中，如果栈不满，先将栈顶指针增 1，然后在该位置上插入元素 *e*。其对应的算法如下：

```
template <typename T>
bool SqStackClass<T>::Push(T e)                    //进栈算法
{    if (top == MaxSize - 1) return false;         //栈满时返回 false
     top++;                                        //栈顶指针增 1
     data[top] = e;                                //将元素 e 进栈
     return true;
}
```

(4) 出栈：Pop(*e*)

元素出栈只能从栈顶出，不能从栈底或中间位置出，如图 3.6 所示。

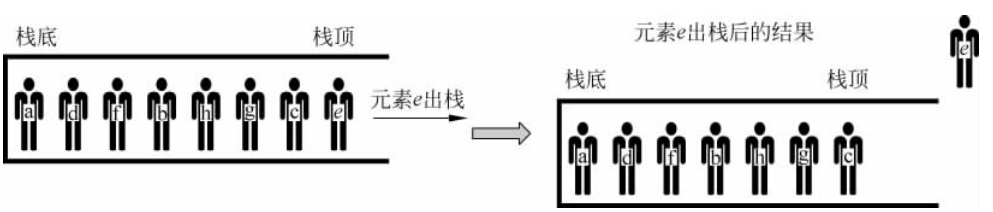

图 3.6　元素出栈示意图

在出栈运算中，如果栈不为空，先将栈顶元素赋给 *e*，然后将栈顶指针减 1。其对应的算法如下：

```
template <typename T>
bool SqStackClass<T>::Pop(T&e)                     //出栈算法
{    if (StackEmpty())return false;                //栈为空的情况，即栈下溢出
     e = data[top];                                //取栈顶指针元素的元素
     top--;                                        //栈顶指针减 1
     return true;
}
```

(5) 取栈顶元素：GetTop(*e*)

在栈不为空的条件下，将栈顶元素赋给 *e*，不移动栈顶指针。其对应的算法如下：

```
template <typename T>
bool SqStackClass<T>::GetTop(T&e)                  //取栈顶元素算法
```

```
{   if (StackEmpty())return false;                    //栈为空的情况,即栈下溢出
    e = data[top];                                    //取栈顶指针位置的元素
    return true;
}
```

【例 3.3】 有 1～n 的 n 个元素,通过一个栈可以产生多种出栈序列,设计一个算法判断序列 str 是否为一个合适的出栈序列,并给出操作过程,要求用相关数据进行测试。

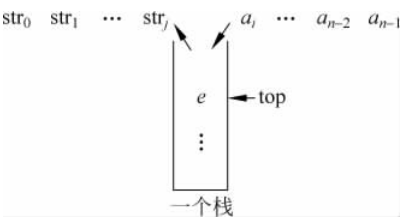

图 3.7 用一个栈判断 str 是否为合适的出栈序列

解:先建立一个整型顺序栈,将进栈序列 1～n 存放到数组 a 中。令 i、j 的初始值均为 0,即分别指向 a、str 数组的第一个元素。比较栈顶元素 e 和 str[j]的大小,若两者不相等,则将 $a[i]$进栈,i 加 1;若栈顶元素 e 和 str[j]相等,则出栈栈顶元素 e,且 j 加 1,如此反复操作,如图 3.7 所示。当 i 大于等于 n 或 j 大于等于 n 时,上述循环过程结束。如果序列 str 是出栈序列,则此时必有 $j==n$,即 a 中的所有元素都通过一个栈产生了一个合适的出栈序列。

对应的程序如下:

```
#include "SqStack.cpp"                                //包含顺序栈类模板的定义
bool isSerial(int str[],int n)
{   int i,j,e; int a[MaxSize];
    SqStackClass<int> st;                             //建立一个顺序栈
    for (i = 0;i<n;i++) a[i] = i + 1;                 //将 1～n 放入数组 a 中
    i = 0;j = 0;
    while (i<n &&j<n)
    {   if (st.StackEmpty() || (st.GetTop(e) && e! = str[j]))
        {   st.Push(a[i]);
            cout << "  元素" << a[i] << "进栈\n";
            i++;
        }
        else
        {   st.Pop(e);
            cout << "  元素" << e << "出栈\n";
            j++;
        }
    }
    while (!st.StackEmpty() && st.GetTop(e) && e == str[j])
    {   st.Pop(e);
        cout << "  元素" << e << "出栈\n";
        j++;
    }
    if (j == n) return true;                          //str 是出栈序列时返回 true
    else return false;                                //str 不是出栈序列时返回 false
}
void Disp(int str[],int n)                            //输出 str
{   int i;
    for (i = 0;i<n;i++)
        cout << str[i];
```

```
}
void main()
{   int n = 4;
    int str[ ] = {3,4,2,1};
    cout << "由 1～" << n << "产生"; Disp(str,n);
    cout << "的操作序列:\n";
    if (isSerial(str,n))
    {   Disp(str,n);
        cout << "是合适的出栈序列\n";
    }
    else
    {   Disp(str,n);
        cout << "不是合适的出栈序列\n";
    }
}
```

该程序判断 3421 是否为合适的出栈序列的结果如下：

```
由 1～4 产生 3421 的操作序列:
  元素 1 进栈
  元素 2 进栈
  元素 3 进栈
  元素 3 出栈
  元素 4 进栈
  元素 4 出栈
  元素 2 出栈
  元素 1 出栈
3421 是合适的出栈序列
```

【例 3.4】 设计一个算法，利用顺序栈检查用户输入的表达式中的括号是否匹配(假设表达式中可能含有圆括号、中括号和大括号)，并用相关数据进行测试。

解：用 str 存放含有各种括号的表达式，建立一个字符型顺序栈 st，用 i 遍历 str，当遇到各种类型的左括号时进栈，当遇到右括号时退栈元素 e，若退栈操作失败返回 false，若 e 与 str[i]不匹配返回 false，如图 3.8 所示，这是因为括号匹配过程遵循后进栈的左括号先判断是否匹配的原则。当 str 遍历完毕后，如果栈 st 为空返回 true，否则返回 false。

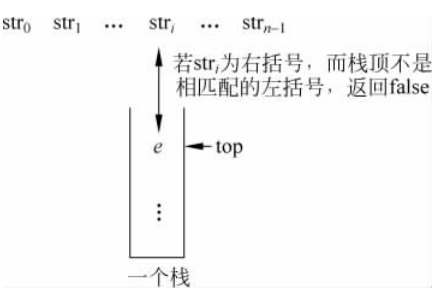

图 3.8　用一个栈判断 str 中的括号是否匹配

对应的程序如下：

```
#include "SqStack.cpp"                          //包含顺序栈类模板的定义
bool isMatch(char str[],int n)                  //判断 str 中的括号是否匹配
{   int i = 0; char e;
    SqStackClass<char> st;                      //建立一个顺序栈
    while (i<n)
    {    if (str[i] == '(' || str[i] == '[' || str[i] == '{')
```

```
            st.Push(str[i]);                    //将左括号进栈
        else
        {   if (str[i] == ')')
            {   if (!st.Pop(e)) return false;   //栈空返回 false
                if (e! = '(') return false;     //栈顶不是相匹配的左括号返回 false
            }
            if (str[i] == ']')
            {   if (!st.Pop(e)) return false;   //栈空返回 false
                if (e! = '[') return false;     //栈顶不是相匹配的左括号返回 false
            }
            if (str[i] == '}')
            {   if (!st.Pop(e)) return false;   //栈空返回 false
                if (e! = '{') return false;     //栈顶不是相匹配的左括号返回 false
            }
        }
        i++;                                    //继续遍历 str
    }
    if (st.StackEmpty()) return true;           //栈空返回 true
    else return false;                          //栈不空返回 false
}
void main()
{   int n = 4; char str[] = "([)]";
    if (isMatch(str,n)) cout << str << "中括号是匹配的\n";
    else cout << str << "中括号不匹配\n";
}
```

该程序判断“([)]”表达式中的括号是否匹配的结果如下：

```
([)]中括号不匹配
```

【例 3.5】 设计一个算法，利用顺序栈判断用户输入的字符串表达式是否为回文，并用相关数据进行测试。

解：用 str 存放表达式，建立一个字符型顺序栈 st，用 i 遍历 str，将所有字符进栈，从栈顶到栈底分别为 str_{n-1}、str_{n-2}、…、str_1、str_0。再次遍历 str，每次退栈元素 e，若当前遍历的字符 str[i]与 e 不相等返回 false，否则继续比较，如图 3.9 所示。当 str 遍历完毕后返回 true。

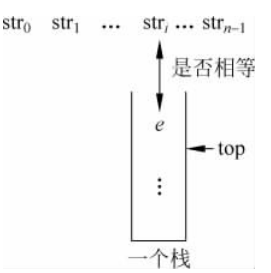

图 3.9 用一个栈判断 str 是否为回文

对应的程序如下：

```
#include "SqStack.cpp"                          //包含顺序栈类模板的定义
bool isPalindrome(char str[],int n)             //判断 str 是否为回文
{   int i = 0; char e;
    SqStackClass<char> st;                      //建立一个顺序栈
    while (i<n)                                 //将 str 的所有字符进栈
    {   st.Push(str[i]);
        i++;                                    //继续遍历 str
    }
    i = 0;
    while (i<n)                                 //再次遍历 str
```

```
    {   st.Pop(e);                              //退栈元素 e
        if (str[i]!=e) return false;            //若 str[i]不等于 e 返回 false
        i++;
    }
    return true;                                //是回文返回 true
}
void main()
{   int n=5;
    char str[]="abcba";
    if (isPalindrome(str,n)) cout << str << "是回文\n";
    else cout << str << "不是回文\n";
}
```

该程序判断“abcba”是否为回文的结果如下：

```
abcba 是回文
```

【例 3.6】 设有两个栈 S1 和 S2，它们都采用顺序栈方式，并且共享一个存储区 $s[0..M-1]$，为了尽量利用空间，减少溢出的可能，请设计这两个栈的存储方式。

解：为了尽量利用空间，减少溢出的可能，可以采用栈顶相向、迎面增长的存储方式，如图 3.10 所示。

两个栈的栈顶指针分别为 top1 和 top2。

栈 S1 空的条件是 top1＝－1；栈 S1 满的条件是 top1＝top2－1；元素 e 进栈 S1(栈不满时)的操作是 top1＋＋；s[top1]＝e；元素 e 出栈 S1(栈不空时)的操作是 e＝s[top1]；top1－－。

栈 S2 空的条件是 top2＝M；栈 S2 满的条件是 top2＝top1＋1；元素 e 进栈 S2(栈不满时)的操作是 top2－－；s[top2]＝e；元素 e 出栈 S2(栈不空时)的操作是 e＝s[top2]；top2＋＋。

图 3.10　两个顺序栈的存储结构

3.1.3　栈的链式存储结构及其基本运算的实现

采用链式存储的栈称为链栈，这里采用单链表实现。链栈的优点是不需要考虑栈满上溢出的情况。规定栈的所有操作都是在单链表的表头进行的，图 3.11 所示为用带头结点的单链表 head 表示的链栈，第一个数据结点是栈顶结点，最后一个结点是栈底结点，栈中元素自栈顶到栈底依次是 a_1、a_2、…、a_n。

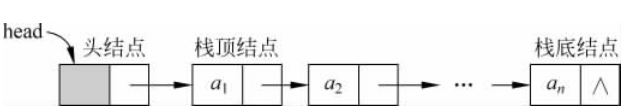

图 3.11　链栈的存储结构

和单链表一样,链栈中每个结点的类型 LinkStack<T>的定义如下:

```
template <typename T>
struct LinkStack                                    //链栈结点类型
{   T data;                                         //数据域
    LinkStack *next;                                //指针域
};
```

链栈类模板 LinkStackClass<T>的设计如下:

```
template <typename T>
class LinkStackClass                                //链栈类模板
{   LinkStack<T> *head;                             //链栈头结点指针
public:
    LinkStackClass();                               //构造函数
    ~LinkStackClass();                              //析构函数
    bool StackEmpty();                              //判栈空算法
    void Push(T e);                                 //进栈算法
    bool Pop(T &e);                                 //出栈算法
    bool GetTop(T &e);                              //取栈顶元素
};
```

从链栈的存储结构可以看到,初始时只含有一个头结点 head 并置 head->next 为 NULL。栈空的条件为 head->next==NULL,由于只有在内存溢出时才会出现栈满,通常不考虑这种情况。元素 e 的进栈操作是将包含该元素的结点插入作为第一个数据结点,出栈操作是删除第一个数据结点。

在链栈中实现栈的基本运算的算法如下。

(1) 链栈的初始化和销毁

链栈的初始化和销毁分别通过构造函数和析构函数来实现,对应的算法如下:

```
template <typename T>
LinkStackClass<T>::LinkStackClass()                 //构造函数
{   head = new LinkStack<T>();
    head->next = NULL;
}
template <typename T>
LinkStackClass<T>::~LinkStackClass()                //析构函数
{   LinkStack<T> *pre = head, *p = pre->next;
    while (p! = NULL)
    {   delete pre;
        pre = p; p = p->next;                       //pre、p同步后移
    }
    delete pre;
}
```

(2) 判断栈是否为空:StackEmpty()

链栈为空的条件是 head->next==NULL,即单链表中没有任何数据结点。其对应的算法如下:

```
template <typename T>
```

```
bool LinkStackClass<T>::StackEmpty()          //判栈空算法
{   return (head->next==NULL); }
```

(3) 进栈:Push(e)

新建包含数据元素 e 的结点 p,将 p 结点插入到头结点之后。其对应的算法如下:

```
template <typename T>
void LinkStackClass<T>::Push(T e)             //进栈算法
{   LinkStack<T> *p=new LinkStack<T>();
    p->data=e;                                //新建元素 e 对应的结点 *p
    p->next=head->next;                       //插入 *p 结点作为开始结点
    head->next=p;
}
```

(4) 出栈:Pop(e)

在链栈不为空的条件下,将第一个数据结点的数据域赋给 e,然后将其删除。其对应的算法如下:

```
template <typename T>
bool LinkStackClass<T>::Pop(T&e)              //出栈算法
{   LinkStack<T> *p;
    if (head->next==NULL) return false;       //栈空的情况
    p=head->next;                             //p 指向开始结点
    e=p->data;
    head->next=p->next;                       //删除 *p 结点
    delete p;                                 //释放 *p 结点
    return true;
}
```

(5) 取栈顶元素:GetTop(s,e)

在栈不为空的条件下,将第一个数据结点的数据域赋给 e,但不删除该结点。其对应的算法如下:

```
template <typename T>
bool LinkStackClass<T>::GetTop(T&e)           //取栈顶元素
{   LinkStack<T> *p;
    if (head->next==NULL) return false;       //栈空的情况
    p=head->next;                             //p 指向开始结点
    e=p->data;
    return true;
}
```

【例 3.7】 设计一个算法,利用栈的基本运算将链栈中的所有元素逆置,并用相关数据进行测试。

解:在将链栈中的所有元素逆置时,先出栈 st 中的所有元素并保存在一个数组 a 中,再将数组 a 中的所有元素依次进栈。

完整的程序如下:

```
#include "LinkStack.cpp"                      //包含链栈类模板的定义
template <typename T>
```

```
void Reverse(LinkStackClass<T>&st)              //逆置链栈中所有元素的算法
{   T *a=new T[MaxSize]; T e;
    int i, n=0;
    while (!st.StackEmpty())                    //将出栈的元素放到数组 a 中
    {   st.Pop(e);
        a[n]=e; n++;
    }
    for (i=0;i<n;i++)                           //将数组 a 的所有元素进入栈
        st.Push(a[i]);
    delete [] a;                                //释放 a 动态数组的空间
}
void main()
{   LinkStackClass<char> st; char e;
    cout << "建立一个链栈 st" << endl;
    cout << "元素 a～e 进栈" << endl;
    st.Push('a'); st.Push('b'); st.Push('c'); st.Push('d'); st.Push('e');
    cout << "逆置链栈中的所有元素" << endl;
    Reverse<char>(st);
    cout << "出栈序列:";
    while (!st.StackEmpty())
    {   st.Pop(e);
        cout << e << " ";
    }
    cout << endl << "销毁链栈 st" << endl;
}
```

该程序的执行结果如下：

```
建立一个链栈 st
元素 a～e 进栈
逆置链栈中的所有元素
出栈序列:a b c d e
销毁链栈 st
```

从中可以看到，元素进栈次序为 a～e，逆置后元素的出栈次序也为 a～e。

3.1.4 栈的应用示例

下面通过利用栈求解简单算术表达式值和求解迷宫问题两个示例来说明栈的应用。

1. 用栈求解简单的算术表达式求值问题

(1) 问题描述

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

(2) 数据组织

简单算术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数学表达式，例如 exp＝"1＋2*(4＋12)"。在设计相关算法中用到了两个栈，一个运算符栈 op 和一个运算数栈 st，均采用顺序栈存储结构，直接利用前面介绍的顺序栈类模板建立，这两个栈对象的定义如下：

```
SqStackClass<char> op;                          //运算符栈
```

```
SqStackClass<double> st;                    //运算数栈
```

(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。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方式,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。

所谓后缀表达式,就是运算符在操作数的后面,例如,1+2*3 的后缀表达式为 123*+。在后缀表达式中已考虑了运算符的优先级,没有括号,只有操作数和运算符。

对后缀表达式求值的过程是,从左到右读入后缀表达式,若读入的是一个操作数,将它入数值栈,若读入的是一个运算符 op,从数值栈中连续出栈两个元素(两个操作数),假设为 x 和 y,计算 x op y 的值,并将计算结果入数值栈。对整个后缀表达式读入结束后,栈顶元素就是计算结果。

表达式的求值过程是,先将算术表达式转换成后缀表达式,然后对该后缀表达式求值。

假设用 exp 存放简单中缀表达式,用字符数组 postexp 存放后缀表达式,设计求表达式值的类 ExpressClass 如下(后面求表达式值的方法均包含在该类中):

```
class ExpressClass                          //求表达式值类
{   char *exp;                              //存放中缀表达式
    char postexp[MaxSize];                  //存放后缀表达式
    int pnum;                               //postexp 中字符的个数
public:
    void Setexp(char *str);                 //获取一个中缀表达式
    void Disppostexp();                     //输出后缀表达式
    void Trans();               //将算术表达式 exp 转换成后缀表达式 postexp
    bool GetValue(double &v);   //计算后缀表达式 postexp 的值
    void Trans1();              //输出将算术表达式 exp 转换成后缀表达式 postexp 的过程
    bool GetValue1(double &v);  //输出计算后缀表达式 postexp 的值的过程
};
```

将算术表达式转换成后缀表达式 postexp 的过程是,对于数字符,将其直接放到 postexp 中;对于“(”,将其直接进栈;对于“)”,退栈并将其放到 postexp 中,直到遇到“(”(不将“(”放到 postexp 中);对于运算符 op_2,将其和栈顶运算符 op_1 的优先级进行比较,只有当 op_2 的优先级高于 op_1 的优先级时,才将 op_2 直接进栈,否则将栈中“(”(如果有)之前的优先级等于或大于 op_2 的运算符均退栈并放到 postexp 中,如图 3.12 所示,再将 op_2 进栈。对于本例的简单表达式,其转换过程如下:

```
while (若 exp 未读完)
{   从 exp 读取字符 ch;
    ch 为数字: 将后续的所有数字依次存放到 postexp 中,并以字符"#"标识数值串结束;
    ch 为左括号"(": 将"("进栈;
    ch 为右括号")": 将 op 栈中"("之前的运算符依次出栈并存放到 postexp 中,再将"("退栈;
    若 ch 的优先级高于栈顶运算符的优先级,则将 ch 进栈,否则退栈并存入 postexp 中,再将 ch 进栈;
}
若字符串 exp 扫描完毕,则退栈 op 中的所有运算符并存放到 postexp 中。
```

在简单算术表达式中,只有“*”和“/”运算符的优先级高于“+”和“-”运算符的优先级。所以,上述过程可进一步修改为:

```
while(若 exp 未读完)
{   从 exp 读取字符 ch;
    ch 为数字: 将后续的所有数字依次存放到 postexp 中,并以字符"#"标志数值串结束;
    ch 为左括号"(": 将"("进栈;
    ch 为右括号")": 将 op 栈中"("之前的运算符依次出栈并存放到 postexp 中,再将"("退栈;
    ch 为"+"或"-":将 op 栈中"("之前的运算符出栈并存放到 postexp 中,再将"+"或"-"进栈;
    ch 为"*"或"/":将 op 栈中"("之前的"*"或"/"运算符出栈并放入 postexp 中,再将"*"或"/"进栈;
}
```

若字符串 exp 扫描完毕,则退栈 op 中的所有运算符并存放到 postexp 中。

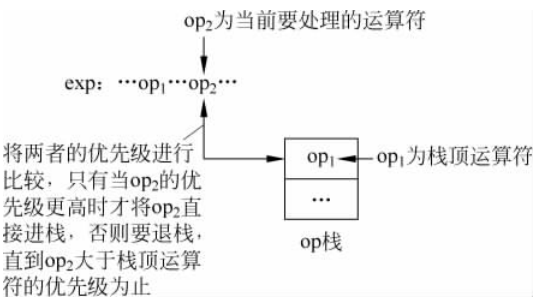

图 3.12 当前运算符的操作

对于算术表达式“(56-20)/(4+2)”,其转换成后缀表达式的过程如表 3.1 所示。

表 3.1 表达式“(56-20)/(4+2)”转换成后缀表达式的过程

op 栈	postexp	说 明
(		遇到 ch 为“(”,将此括号进栈到 op 中
(	56#	遇到 ch 为数字,将 56 存入数组 exp 中,并插入一个字符“#”
(-	56#	遇到 ch 为“-”,由于 op 中“(”之前没有字符,则直接将 ch 进栈到 op 中
(-	56#20#	遇到 ch 为数字,将 20# 存入数组 exp 中
	56#20#-	遇到 ch 为“)”,将栈 op 中“(”之前的字符依次删除并存入数组 exp 中,然后将“(”删除
/	56#20#-	遇到 ch 为“/”,将 ch 进栈到 op 中
/(	56#20#-	遇到 ch 为“(”,将此括号进栈到 op 中
/(	56#20#-4#	遇到 ch 为数字,将 4# 存入数组 exp 中
/(+	56#20#-4#	遇到 ch 为“+”,由于 op 中“(”之前没有字符,直接将 ch 进栈到 op 中
/(+	56#20#-4#2#	遇到 ch 为数字,将 2# 存入数组 exp 中
/	56#20#-4#2#+	遇到 ch 为“)”,将栈 op 中“(”之前的字符依次删除存入数组 exp 中,然后将“(”删除
	56#20#-4#2#+/	str 扫描完毕,将栈 op 中的所有运算符依次弹出并存入数组 exp 中,然后再将 ch 存入数组 exp 中,得到后缀表达式

根据上述原理得到的 Trans()算法如下:

```
void ExpressClass::Trans()          //将算术表达式 exp 转换成后缀表达式 postexp
{   SqStackClass<char> op;          //运算符栈
    int i=0,j=0;                    //i、j 作为 exp 和 postexp 的下标
    char ch,e;
    while (exp[i])                  //exp 表达式未扫描完时循环
    {   ch=exp[i];
        if (ch=='(') op.Push(ch);//判定为左括号,将左括号进栈
        else if (ch==')')           //判定为右括号
        {   while (!op.StackEmpty() && op.GetTop(e) && e!='(')
            {                       //将 op 栈中"("之前的运算符退栈并存放到 postexp 中
                op.Pop(e);
                postexp[j++]=e;
            }
            op.Pop(e);              //将"("退栈
        }
        else if (ch=='+'||ch=='-')  //判定为加号或减号
        {   while (!op.StackEmpty() && op.GetTop(e) && e!='(')
            {                         //将 op 栈中"("之前的所有运算符退栈并存放到 postexp 中
                op.Pop(e);
                postexp[j++]=e;
            }
            op.Push(ch);              //再将"+"或"-"进栈
        }
        else if (ch=='*'||ch=='/')     //判定为"*"或"/"号
        {   while (!op.StackEmpty() && op.GetTop(e) && e!='('&&(e=='*'||e=='/'))
            {              //将 op 栈中"("之前的"*"或"/"运算符依次出栈并存放到 postexp 中
                op.Pop(e);
                postexp[j++]=e;
            }
            op.Push(ch);  //再将"*"或"/"进栈
        }
        else               //处理数字字符
        {   while (ch>='0'&&ch<='9')  //判定为数字
            {   postexp[j++]=ch; i++;   //将连续的数字放入 postexp
                if (exp[i]) ch=exp[i];
                else break;
            }
            i--;                        //退一个字符
            postexp[j++]='#';           //用"#"标识一个数值串结束
        }
        i++;                            //继续处理其他字符
    }
    while (!op.StackEmpty())            //此时 exp 扫描完毕,栈不空时循环
    {   op.Pop(e);                      //将栈中的所有运算符退栈并放入 postexp
        postexp[j++]=e;
    }
    pnum=j;                             //保存 postexp 中字符的个数
}
```

输出算术表达式 exp 转换成后缀表达式 postexp 的 Tranl 算法的设计方法与此类似，在此不再介绍。

在后缀表达式求值算法中要用到一个数值栈 st,后缀表达式的求值过程如下:

```
while (若 postexp 未读完)
{   从 postexp 读取字符 ch;
    ch 为"+": 从栈 st 中出栈两个数值 a 和 b,计算 c = a + b,将 c 进栈;
    ch 为"-": 从栈 st 中出栈两个数值 a 和 b,计算 c = b - a,将 c 进栈;
    ch 为"*": 从栈 st 中出栈两个数值 a 和 b,计算 c = b * a,将 c 进栈;
    ch 为"/": 从栈 st 中出栈两个数值 a 和 b,若 a 不为零,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;
}
```

对于后缀表达式"56＃20＃－4＃2＃＋/",其求值过程如表 3.2 所示。

表 3.2　后缀表达式"56＃20＃－4＃2＃＋/"的求值过程

st 栈	说　明
56	遇到 56＃,将 56 进栈
56,20	遇到 20＃,将 20 进栈
36	遇到"－",出栈两次,将 56－20＝36 进栈
36,4	遇到 4＃,将 4 进栈
36,4,2	遇到 2＃,将 2 进栈
36,6	遇到"＋",出栈两次,将 4＋2＝6 进栈
6	遇到"/",出栈两次,将 36/6＝6 进栈
	postexp 扫描完毕,算法结束,栈顶数值 6 即为所求

根据上述计算原理得到的算法如下:

```
bool ExpressClass::GetValue(double &v)      //计算后缀表达式 postexp 的值 v
{   SqStackClass<double> st;                //运算数栈 st
    double a,b,c,d;     int i = 0; char ch;
    while (i<pnum)                          //postexp 字符串未扫描完时循环
    {   ch = postexp[i];                    //从后缀表达式中取一个字符 ch
        switch (ch)
        {
        case '+':                           //判定为"+"号
            st.Pop(a);                      //退栈取数值 a
            st.Pop(b);                      //退栈取数值 b
            c = b + a;                      //计算 c
            st.Push(c);                     //将计算结果进栈
            break;
        case '-':                           //判定为"-"号
            st.Pop(a);                      //退栈取数值 a
            st.Pop(b);                      //退栈取数值 b
            c = b - a;                      //计算 c
            st.Push(c);                     //将计算结果进栈
            break;
        case '*':                           //判定为"*"号
            st.Pop(a);                      //退栈取数值 a
            st.Pop(b);                      //退栈取数值 b
            c = b * a;                      //计算 c
            st.Push(c);                     //将计算结果进栈
```

```
            break;
        case '/':                          //判定为"/"号
            st.Pop(a);                     //退栈取数值 a
            st.Pop(b);                     //退栈取数值 b
            if (a! = 0)
            {   c = b/a;                   //计算 c
                st.Push(c);                //将计算结果进栈
            }
            else return false;             //除零错误返回 false
            break;
        default:                           //处理数字字符
            d = 0;                         //将连续的数字符转换成数值存放到 d 中
            while (ch>= '0' && ch<= '9')   //判定为数字字符
            {   d = 10 * d + (ch - '0');
                i++;
                ch = postexp[i];
            }
            st.Push(d);                    //将数值 d 进栈
            break;
        }
        i++;                               //继续处理其他字符
    }
    st.GetTop(v);                          //栈顶元素即为求值结果
    return true;
}
```

输出计算后缀表达式 postexp 值的 GetValue1 算法的设计方法与此类似，在此不再介绍。

(4) 设计主函数

设计以下主函数求简单算术表达式“(56－20)/(4＋2)”的值：

```
void main()
{   double v;
    ExpressClass obj;
    char * str = "(56 - 20)/(4 + 2)";
    obj.Setexp(str);
    cout << "中缀表达式" << str <<"转换为后缀表达式的过程:\n";
    obj.Trans1();
    cout << "求得的后缀表达式:"; obj.Disppostexp();
    cout << "求后缀表达式的过程:\n";
    obj.GetValue1(v);
    cout << "求得的表达式值:" << v << endl;
}
```

(5) 程序执行结果

该程序的执行结果如下：

```
中缀表达式(56 - 20)/(4 + 2)转换为后缀表达式的过程:
  运算符'('进栈
  5→postexp       6→postexp       postexp 中加 #
```

```
  运算符'-'进栈
  2→postexp       0→postexp       postexp 中加 #
  运算符'-'退栈→postexp
  运算符')'退栈
  运算符'/'进栈
  运算符'('进栈
  4→postexp       postexp 中加 #
  运算符'+'进栈
  2→postexp       postexp 中加 #
  运算符'+'退栈→postexp
  运算符')'退栈
  运算符'/'退栈→postexp
求得的后缀表达式:56#20#-4#2#+/
求后缀表达式的过程:
  运算数 56 进栈
  运算数 20 进栈
  运算数 20 退栈        运算数 56 退栈
  计算 56-20=36
  运算数 36 进栈
  运算数 4 进栈
  运算数 2 进栈
  运算数 2 退栈        运算数 4 退栈
  计算 4+2=6
  运算数 6 进栈
  运算数 6 退栈        运算数 36 退栈
  计算 36/6=6
  运算数 6 进栈
  取栈顶数值 6
求得的表达式值:6
```

2. 用栈求解迷宫问题

(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路径。假设迷宫图如图 3.13 所示(其中 $M=10$、$N=10$,含外围加上一圈不可走的方块,这样做的目的是避免在查找时出界),迷宫由方块构成,空白方块表示可以走的通道,带阴影方块表示不可走的障碍物。要求所求路径必须是简单路径,即在求得的路径上不能重复出现同一个空白方块,而且从每个方块出发只能走向上、下、左、右 4 个相邻的空白方块。

(2) 数据组织

为了表示迷宫,设置一个数组 a,其中每个元素表示一个方块的状态,当为 0 时表示对应方块是通道,当为 1 时表示对应方块不可走。为了算法方便,在一般的迷宫外围加了一条围墙。图 3.13 所示的迷宫对应的迷宫数组 a(由于迷宫四周加了一条围墙,故数组 a 的外围元素均为 1)如下:

```
int a[M][N]= { {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} };
```

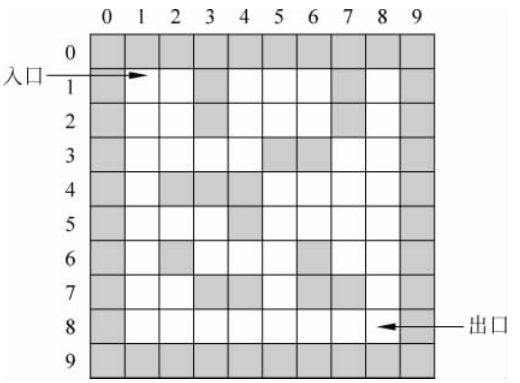

图 3.13　迷宫示意图

另外,在算法中用到的栈采用顺序栈存储结构,即将栈定义如下:

```
struct Box                                  //方块结构体类型
{   int i;                                  //方块的行号
    int j;                                  //方块的列号
    int di;                                 //di 是下一个可走相邻方位的方位号
};
class Stack                                 //顺序栈类
{   Box * data;                             //存放栈中的方块
    int top;                                //栈顶指针
public:
    Stack()                                 //构造函数:栈初始化
    {   data = new Box[StackSize];
        top = -1;
    }
    ~Stack()                                //析构函数:释放栈空间
    {   delete [] data; }
    bool StackEmpty()                       //判断栈是否为空
    {   return(top == -1); }
    void Push(int x,int y,int d)            //进栈一个方块
    {   top++;
        data[top].i = x; data[top].j = y; data[top].di = d;
    }
    void GetTop(int &x,int &y,int &d)       //取栈顶方块
    {   x = data[top].i; y = data[top].j; d = data[top].di; }
    void Pop()                              //退栈一个方块
    {   top--; }
    void Setdi(int d)                       //修改栈顶元素的 di 值
    {   data[top].di = d; }
    void DispBox()                          //输出栈中所有方块构成一条迷宫路径
    {   int k;
        cout << "一条迷宫路径如下:\n";
        for (k = 0;k <= top;k++)
        {   cout << "(" << data[k].i << "," << data[k].j << ")";
```

```
            if ((k+1)%5==0) cout << endl;     //每行输出5个方块
        }
        cout << endl;
    }
};
```

(3) 设计运算算法

求迷宫问题就是在一个指定的迷宫中求从入口到出口的路径。在求解时,通常用"穷举求解"的方法,即从入口出发,顺某一方向向前试探,若能走通,则继续往前走;否则沿原路退回,换一个方向再继续试探,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),需要用一个后进先出的栈来保存从入口到当前位置的路径。

对于迷宫中的每个方块,有上、下、左、右4个方块相邻,如图3.14所示。第i行第j列的方块的位置记为(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,假设从方位0到方位3的方向查找下一个可走的方块。为了便于回溯,对于可走的方块都要进栈,并试探它的下一个可走的方位,将这个可走的方位保存到栈中。

求解迷宫(xi,yi)到(xe,ye)的路径的过程是,先将入口进栈(将其初始方位设置为-1),在栈不为空时循环,即取栈顶方块(不退栈),若该方块是出口,输出栈中的所有方块(即为路径),否则找下一个可走的相邻方块,若不存在这样的方块,说明当前路径不可能走通,回溯,也就是恢复当前方块为0后退栈。若存在这样的方块,则将其方位保存到栈顶元素中,并将这个可走的相邻方块进栈(将其初始方位设置为-1)。

求迷宫路径的回溯过程如图3.15所示,从前一方块找到一个可走相邻方块(即当前方块)后,再从当前方块找相邻可走方块,若没有这样的方块,说明当前方块不可能是从入口到出口路径上的一个方块,则从当前方块回溯到前一方块,继续从前一方块找另一个可走的相邻方块。

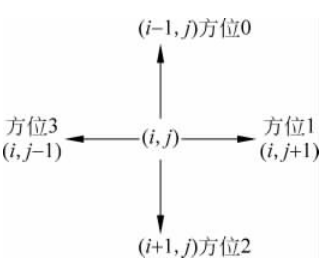

图3.14 方位图

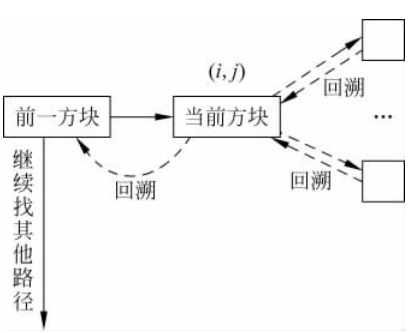

图3.15 求迷宫路径的回溯过程

为了保证试探的可走相邻方块不是已走路径上的方块,如(i,j)已进栈,在试探$(i+1,j)$的下一个可走方块时又试探到(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a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。

求解迷宫问题的Maze类的定义如下:

```
class Maze1                                    //用栈求解一条迷宫路径类
{   int a[MaxSize][MaxSize];                   //迷宫数组
    int m,n;                                   //迷宫行列数
public:
    void Seta(int mg[][MaxSize],int m1,int n1)   //设置迷宫数组
    {   int i,j;
        m = m1; n = n1;
        for (i = 0;i < m;i++)
            for (j = 0;j < n;j++)
                a[i][j] = mg[i][j];
    }
    bool mgpath(int xi,int yi,int xe,int ye)   //求一条从(xi,yi)到(xe,ye)的迷宫路径
    {   int i,j,di,i1,j1;
        bool find;
        Stack st;                              //建立一个空栈
        st.Push(xi,yi, - 1);                   //入口方块进栈
        a[xi][yi] = - 1;                       //为避免来回找相邻方块,将进栈的方块置为 - 1
        while (!st.StackEmpty())               //栈不空时循环
        {   st.GetTop(i,j,di);                 //取栈顶方块,称为当前方块
            if (i == xe && j == ye)            //找到了出口,输出栈中的所有方块构成一条路径
            {   st.DispBox();
                return true;                   //找到一条路径后返回 true
            }
            find = false;                      //否则继续找路径
            while (di < 4 && !find)            //找下一个相邻可走方块
            {   di++;                          //找下一个方位的相邻方块
                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 (a[i1][j1] == 0) find = true;     //找到下一个可走相邻方块(i1,j1)
            }
            if (find)                          //找到了下一个可走方块
            {   st.Setdi(di);                  //修改原栈顶元素的 di 值
                st.Push(i1,j1, - 1);           //下一个可走方块进栈
                a[i1][j1] = - 1;               //为避免来回找相邻方块,将进栈的方块置为 - 1
            }
            else                               //没有路径可走,则退栈
            {   a[i][j] = 0;                   //恢复当前方块的迷宫值
                st.Pop();                      //将栈顶方块退栈
            }
        }
        return false;                          //没有找到迷宫路径,返回 false
    }
};
```

其中,mgpath(int x*i*,int y*i*,int x*e*,int y*e*)成员函数用于求一条从入口(x*i*,y*i*)到出口

(x*e*,y*e*)的迷宫路径,当成功找到出口后,栈 st 中从栈底到栈顶恰好是一条从入口到出口的迷宫路径,输出该迷宫路径并返回 true,否则返回 false。

(4) 设计主函数

以下主函数用于求图 3.13 所示的迷宫图中从(1,1)到(8,8)的一条迷宫路径:

```
void main()
{    int mg[M][N] = { {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} };
     Maze1 mz;                                  //创建一个 Maze1 对象 mz
     mz.Seta(mg,10,10);                         //设置迷宫数组
     cout << "求(1,1)到(8,8)的迷宫路径\n";
     if (!mz.mgpath(1,1,8,8))                   //求入口为(1,1)、出口为(8,8)的迷宫路径
         cout << "不能存在迷宫路径\n";
}
```

(5) 程序执行结果

该程序的执行结果如下:

```
求(1,1)到(8,8)的迷宫路径
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)
```

该路径如图 3.16 所示(用浅阴影表示迷宫路径,方块上的箭头指示路径上下一个方块的方位),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,将在后面介绍)。

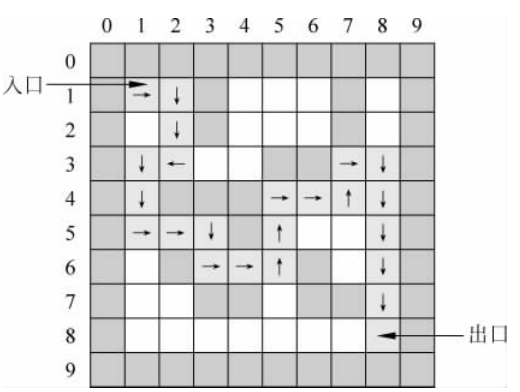

图 3.16　用栈找到的一条路径

3.2　队列

本节先介绍队列的定义，然后介绍队列的存储结构和基本运算算法设计，最后通过实例讨论队列的应用。

3.2.1　队列的定义

同样先看一个例子，假设有一个独木桥，桥右侧有一群小兔子要到桥左侧去，桥宽只能容纳一只兔子，那么这群小兔子怎么过桥呢？结论是只能一个接一个地过桥，如图 3.17 所示。在这个例子中，独木桥就是一个队列，由于其宽度只能容纳一只兔子，所以不论有多少只兔子，它们只能是一只一只地过桥，从而构成一种线性关系。再看独木桥的主要操作，显然有上桥和下桥两种操作，上桥表示从桥右侧走到桥上，下桥表示离开桥。

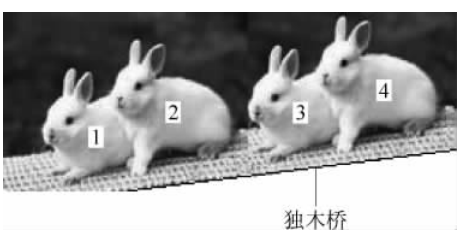

图 3.17　一群小兔子过独木桥

归纳起来，队列（简称为队）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，而在表的另一端进行删除。一般情况下，将进行插入的一端称为**队尾**（rear），将进行删除的一端称为**队头**或**队首**（front）。向队列中插入新元素称为**进队**或**入队**，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**，元素出队后，其直接后继元素就成为队首元素。

由于队列的插入和删除操作分别是在表的一端进行的，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**。

图 3.18 所示为一个队列的动态示意图，图中 front 指针指向队首位置（实际上是队首元素的前一个位置），rear 指针指向队尾位置（正好是队尾元素的位置）。图 3.18(a)表示一个空队；图 3.18(b)表示插入 5 个数据元素后的状态；图 3.18(c)表示出队一次后的状态；图 3.18(d)表示出队 4 次后的状态。

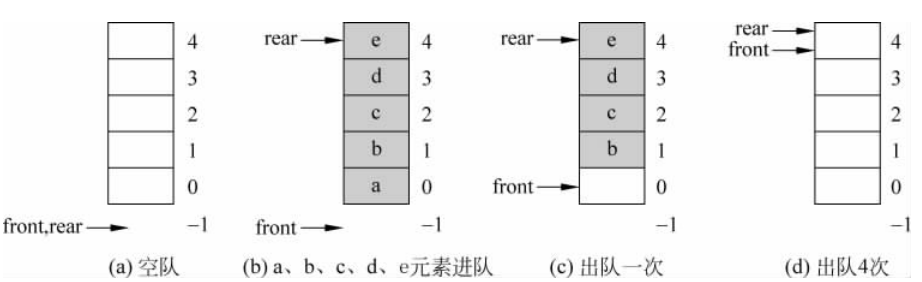

图 3.18　队列的动态示意图

抽象数据类型队列的定义如下：

```
ADT Queue
{
数据对象:
    D={ai | 1≤i≤n,n≥0,ai 为 T 类型}
数据关系:
    R={r}
    r={<ai,ai+1> | ai,ai+1∈D, i=1,…,n-1}
基本运算:
    bool QueueEmpty()          //判断队列是否为空,若队列为空,返回真,否则返回假
    bool deQueue(ref string e)//进队运算,将元素 e 进队作为队尾元素
    bool deQueue(ref string e)//出队运算,从队列中出队一个元素,并将其值赋给 e
}
```

【例 3.8】 若元素进队顺序为 1234,能否得到 3142 的出队顺序?

解：进队顺序为 1234,则出队顺序也为 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

用户可以采用顺序存储结构存储一个队列,其中使用一个数组 data 和两个整型变量,数组 data(大小为常量 MaxSize)顺序存储队列中的所有元素,两个整型变量 front 和 rear 分别作为队首指针(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**。

顺序队存储结构如图 3.19 所示,带阴影的元素为队中元素。顺序队分为非循环队列和循环队列两种形式。

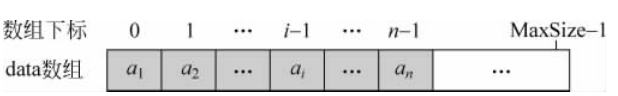

图 3.19 顺序队示意图

非循环队列类模板 SqQueueClass<T>的设计如下：

```
template <typename T>
class SqQueueClass              //非循环队列类模板
{   T *data;                    //存放队中元素
    int front, rear;            //队头和队尾指针
public:
    SqQueueClass();             //构造函数
    ~SqQueueClass();            //析构函数
    bool QueueEmpty();          //判断队列是否为空
    bool enQueue(T e);          //进队列算法
    bool deQueue(T &e);         //出队列算法
};
```

循环队列类模板 SqQueueClass1<T>的设计如下：

```
template <typename T>
class SqQueueClass1             //循环队列类模板
```

```
{    T * data;                  //存放队中元素
     int front, rear;           //队头和队尾指针
public:
     SqQueueClass1();           //构造函数
     ~SqQueueClass1();          //析构函数
     bool QueueEmpty1();        //判断队列是否为空
     bool enQueue1(T e);        //进队列算法
     bool deQueue1(T &e);       //出队列算法
};
```

1. 在非循环队列中实现队列的基本运算

图 3.18 可以看成是非循环队列，其中，初始时置 front＝rear＝－1，这样队空的条件为 front＝＝rear，队满的条件为 rear＝＝MaxSize－1(rear 指向数组最大下标时为队满)。元素 e 进队的操作是先将队尾指针 rear 增 1，然后将 e 放在队尾处；出队操作是先将队头指针 front 增 1，然后取出队头处的元素。

说明：在顺序队中，队尾指针总是指向当前队列中队尾的元素，而队头指针总是指向当前队列中队头元素的前一个位置。

在非循环队列中实现队列的基本运算算法如下。

(1) 非循环队列的初始化和销毁

非循环队列的初始化和销毁分别通过构造函数和析构函数来实现，对应的算法如下：

```
template <typename T>
SqQueueClass<T>::SqQueueClass()        //构造函数
{    data = new T[MaxSize];            //为 data 分配空间
     front = rear = -1;                //将队头和队尾指针置初值
}
template <typename T>
SqQueueClass<T>::~SqQueueClass()       //析构函数
{    delete [] data; }
```

(2) 判断队列是否为空：QueueEmpty()

若队列满足 front＝＝rear 条件，则返回 true，否则返回 false。其对应的算法如下：

```
template <typename T>
bool SqQueueClass<T>::QueueEmpty()     //判断队列是否为空
{    return (front == rear); }
```

(3) 进队运算：enQueue(e)

元素进队只能从队尾进，不能从队头或中间位置进队，如图 3.20 所示。

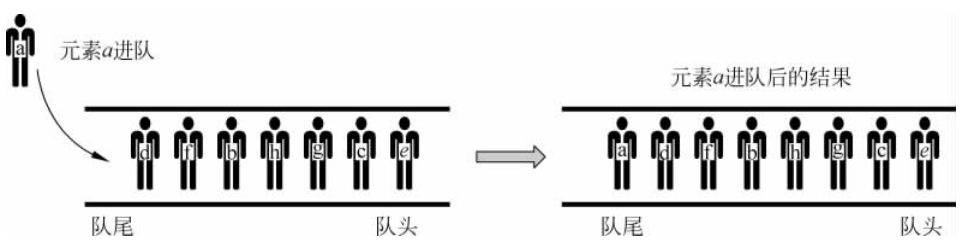

图 3.20　元素进队的示意图

在进队运算中,如果队列不满,先将队尾指针 rear 增 1,然后将元素 e 放到该位置处。其对应的算法如下:

```
template <typename T>
bool SqQueueClass<T>::enQueue(T e)     //进队列算法
{    if (rear == MaxSize - 1)          //队满上溢出
         return false;
     rear++; data[rear] = e;
     return true;
}
```

(4) 出队列:deQueue(e)

元素出队只能从队头出,不能从队头或中间位置出,如图 3.21 所示。

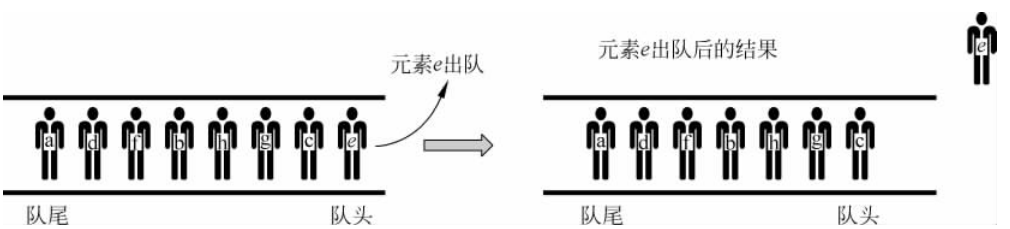

图 3.21　元素出队的示意图

在出队运算中,当队列不为空时,将队首指针 front 增 1,并将该位置的元素值赋给 e。其对应的算法如下:

```
template <typename T>
bool SqQueueClass<T>::deQueue(T&e)     //出队列算法
{    if (front == rear)                //队空下溢出
         return false;
     front++;
     e = data[front];
     return true;
}
```

2. 在循环队列中实现队列的基本运算

在非循环队列中,元素进队时队尾指针 rear 增 1,元素出队时队头指针 front 增 1,当进队 MaxSize 个元素后,满足队满的条件(即 rear==MaxSize-1 成立),此时即使出队若干元素,队满条件仍成立(实际上队列中有空位置),这是一种假溢出。为了能够充分地使用数组中的存储空间,把数组的前端和后端连接起来,形成一个循环的顺序表,即把存储队列元素的表从逻辑上看成一个环,称为循环队列(也称为环形队列)。

循环队列首尾相连,当队首指针 front=MaxSize-1 后,再前进一个位置就自动到 0,这可以利用求余运算(%)来实现。

队首指针进 1:front=(front+1)%MaxSize

队尾指针进 1:rear=(rear+1)%MaxSize

循环队列的队头指针和队尾指针初始化时都置 0,即 front=rear=0。在进队元素和出队元素时,队头和队尾指针都循环前进一个位置。

那么，循环队列的队满和队空的判断条件是什么呢？显然，循环队列为空的条件是 rear==front。如果进队元素的速度快于出队元素的速度，队尾指针很快就赶上了队首指针，此时可以看出循环队列的队满条件也为 rear==front。

怎样区分这两者之间的差别呢？通常约定在进队时少用一个数据元素空间，以队尾指针加 1 等于队首指针作为队满的条件，即队满条件为(rear+1)%MaxSize==front。队空条件仍为 rear==front。

图 3.22 所示说明了循环队列的几种状态，这里假设 MaxSize 等于 5。图 3.22(a)为空队，此时 front=rear=0；图 3.22(b)中有 3 个元素，当进队元素 d 后，队中有 4 个元素，此时满足队满的条件。

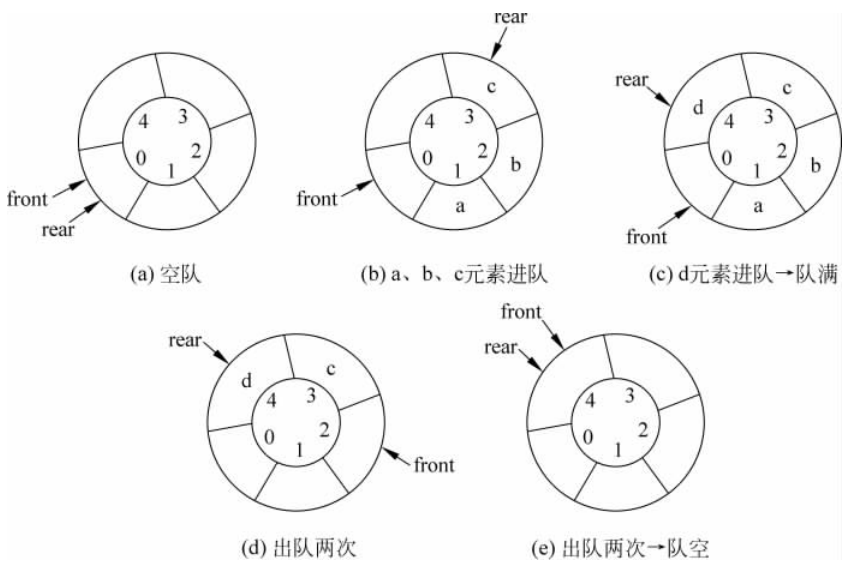

图 3.22　循环队列进队和出队操作示意图

说明：在上述循环队列中，队首指针 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，队中的元素个数等于(rear−front+MaxSize)%MaxSize。这样的循环队列中最多只能保存 MaxSize−1 个元素。

注意：循环队列和非循环队列相比，前者解决了“假溢出”现象，当多次进队和出队时，在队列中可以存放更多的元素。但是，当每个进队的元素对求解结果有用时(例如后面介绍的用队列求解迷宫)，不应该使用循环队列，而应该使用非循环队列(在非循环队列中，所有进队的元素都没有被覆盖，所以可用于求解最终结果)。

在这样的循环队列中，实现队列的基本运算算法如下。

(1) 循环队列的初始化和销毁

循环队列的初始化和销毁分别通过构造函数和析构函数来实现，对应的算法如下：

```
template <typename T>
SqQueueClass1<T>::SqQueueClass1()          //构造函数
{   data = new T[MaxSize];                 //为 data 分配空间
    front = rear = 0;                      //将队头、队尾指针置初值
}
```

```
template <typename T>
SqQueueClass1<T>::~SqQueueClass1()          //析构函数
{   delete [] data; }
```

(2) 判断队列是否为空：QueueEmpty(q)

若队列满足 front==rear 条件，返回 true，否则返回 false。其对应的算法如下：

```
template <typename T>
bool SqQueueClass1<T>::QueueEmpty1()        //判断队列是否为空
{   return (front == rear); }
```

(3) 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 e 放到该位置处。其对应的算法如下：

```
template <typename T>
bool SqQueueClass1<T>::enQueue1(T e)        //进队列算法
{   if ((rear + 1) % MaxSize == front)      //队满上溢出
        return false;
    rear = (rear + 1) % MaxSize;
    data[rear] = e;
    return true;
}
```

(4) 出队列：deQueue(q,e)

在队列不为空的条件下，将队首指针 front 循环增 1，并将该位置的元素值赋给 e。其对应的算法如下：

```
template <typename T>
bool SqQueueClass1<T>::deQueue1(T&e)        //出队列算法
{   if (front == rear)                      //队空下溢出
        return false;
    front = (front + 1) % MaxSize;
    e = data[front];
    return true;
}
```

【例 3.9】 设计求循环队列中元素个数的算法，并利用循环队列基本运算设计进队和出队第 k($k\geqslant1$)个元素的算法，要求用相关数据进行测试。

解：队列中没有直接取 k($k\geqslant1$)个元素的基本运算，进队第 k($k\geqslant1$)个元素 e 的算法思路是，出队前 $k-1$ 个元素，边出边进，再将元素 e 进队，将剩下的元素边出边进。出队第 k($k\geqslant1$)个元素 e 的算法思路是，出队前 $k-1$ 个元素，边出边进，出队第 k 个元素 e，e 不进队，将剩下的元素边出边进。由于是循环队列，在 SqQueueClass1 类模板中增加以下友元函数：

```
friend void Display(SqQueueClass1<T> &qu);              //从队头到队尾输出队中的所有元素
friend int GetCount(SqQueueClass1<T> &qu);              //返回队中元素的个数
friend bool enQueuek(SqQueueClass1<T> &qu,int k,T e);   //进队第 k 个元素 e
friend bool deQueuek(SqQueueClass1<T> &qu,int k, T &e); //出队第 k 个元素 e
```

实现本例功能的完整程序如下：

```
#include "SqQueue1.cpp"                          //包含循环队列的定义
template <typename T>
void Display(SqQueueClass1<T> &qu)               //从队头到队尾输出队中的所有元素
{   int i=(qu.front+1)%MaxSize;
    while (i!=qu.rear)
    {   cout<<qu.data[i]<<" ";
        i=(i+1)%MaxSize;
    }
    cout<<endl;
}
template <typename T>
int GetCount(SqQueueClass1<T> &qu)               //返回队中元素的个数
{   return ((qu.rear-qu.front+MaxSize)%MaxSize); }
template <typename T>
bool enQueuek(SqQueueClass1<T> &qu,int k,T e)//进队第 k 个元素 e
{   T x;
    int i=1,n=GetCount(qu);
    if (k<1 || k>n+1) return false;              //参数 k 错误返回 false
    if (k<=n)
        for (i=1;i<=n;i++)                       //循环处理队中的所有元素
        {   if (i==k) qu.enQueue1(e);            //将 e 元素进队到第 k 个位置
            qu.deQueue1(x);                      //出队元素 x
            qu.enQueue1(x);                      //进队元素 x
        }
    else qu.enQueue1(e);                         //k=n+1 时直接进队 e
    return true;
}
template <typename T>
bool deQueuek(SqQueueClass1<T> &qu,int k, T&e)   //出队第 k 个元素 e
{   T x;
    int i=1, n=GetCount(qu);
    if (k<1 || k>n) return false;                //参数 k 错误返回 false
    for (i=1; i<=n; i++)                         //循环处理队中的所有元素
    {   qu.deQueue1(x);                          //出队元素 x
        if (i!=k) qu.enQueue1(x);                //将非第 k 个元素进队
        else e=x;                                //取第 k 个出队的元素
    }
    return true;
}
void main()
{   SqQueueClass1<char> sq;                      //定义一个字符顺序队 sq
    char e;
    cout<<"建立一个空队 sq\n";
    cout<<"元素 a 进队\n"; sq.enQueue1('a');
    cout<<"元素 b 进队\n"; sq.enQueue1('b');
    cout<<"元素 c 进队\n"; sq.enQueue1('c');
    cout<<"元素 d 进队\n"; sq.enQueue1('d');
    cout<<"元素 e 进队\n"; sq.enQueue1('e');
    cout<<"sq 中元素的个数:"<<GetCount(sq)<<endl;
```

```
    cout << "进队第 2 个元素 x" << endl;
    enQueuek(sq,2,'x');
    cout << "队头到队尾元素:"; Display(sq);
    cout << "出队第 4 个元素" << endl;
    deQueuek(sq,4,e);
    cout << "第 4 个元素为" << e << endl;
    cout << "队头到队尾元素:"; Display(sq);
    cout << "销毁队 sq" << endl;
}
```

该程序的执行结果如下:

```
建立一个空队 sq
元素 a 进队
元素 b 进队
元素 c 进队
元素 d 进队
元素 e 进队
sq 中元素的个数:5
进队第 2 个元素 x
队头到队尾元素:a x b c d
出队第 4 个元素
第 4 个元素为 c
队头到队尾元素:a x b d
销毁队 sq
```

【例 3.10】 对于循环队列来说,如果知道队头指针和队列中元素的个数,则可以计算出队尾指针,也就是说,可以用队列中元素的个数代替队尾指针。设计出这种循环队列的进队、出队、判队空和求队中元素个数的算法。

解:本例的循环队列包含 data 数组、队头指针 rear 和队中元素个数 count。初始时 front 和 count 均置为 0,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 rear1,将 rear1 循环增 1,然后将元素 e 放置在 rear1 处;出队操作是,先将队首指针循环增 1,然后取出该位置的元素。设计对应的循环队列类 SqQueueClass2 如下:

```
#include <iostream.h>
const int MaxSize = 20;                    //队大小
template <typename T>
class SqQueueClass2                        //循环队列类模板
{   T *data;
    int front,count;                       //队头和队中元素个数
public:
    SqQueueClass2();                       //构造函数,用于队列的初始化
    ~SqQueueClass2();                      //析构函数,用于释放空间
    bool QueueEmpty();                     //判断队列是否为空
    bool enQueue(T e);                     //进队列算法
    bool deQueue(T &e);                    //出队列算法
    int GetCount();                        //求队列中元素的个数
};
template <typename T>
```

```
SqQueueClass2<T>::SqQueueClass2()          //构造函数,用于队列的初始化
{   data = new T[MaxSize];
    front = 0; count = 0;
}
template <typename T>
SqQueueClass2<T>::~SqQueueClass2()         //析构函数,用于释放空间
{   delete [] data; }
template <typename T>
bool SqQueueClass2<T>::QueueEmpty()        //判断队列是否为空
{   return (count == 0); }
template <typename T>
bool SqQueueClass2<T>::enQueue(T e)        //进队列算法
{   int rear1;
    rear1 = (front + count) % MaxSize;
    if (count == MaxSize) return false;    //队满上溢出
    rear1 = (rear1 + 1) % MaxSize;
    data[rear1] = e; count++;
    return true;
}
template <typename T>
bool SqQueueClass2<T>::deQueue(T&e)        //出队列算法
{   if (count == 0) return false;          //队空下溢出
    front = (front + 1) % MaxSize;
    e = data[front]; count--;
    return true;
}
template <typename T>
int SqQueueClass2<T>::GetCount()           //求队列中元素的个数
{   return count; }
```

说明：本例设计的循环队列中最多可保存 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

队列的链式存储结构也是通过由结点构成的单链表实现的，此时只允许在单链表的表首进行删除操作和在单链表表尾进行插入操作，因此需要使用两个指针，即队首指针 front 和队尾指针 rear，用 front 指向队首结点，用 rear 指向队尾结点。用于存储队列的单链表简称为**链队**。

链队存储结构如图 3.23 所示，链队中数据结点的类型 LinkNode<T>的定义如下：

```
template <typename T>
struct LinkNode                            //链队数据结点类型
{    T data;                               //结点数据域
     LinkNode * next;                      //指向下一个结点
};
```

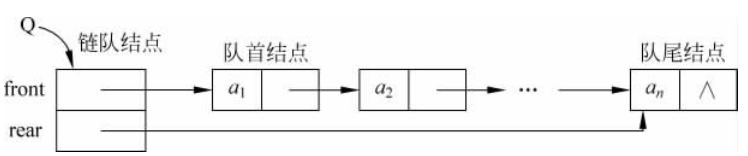

图 3.23　链栈存储结构的示意图

链队结点的类型 LinkQueue<T>的定义如下：

```
template <typename T>
struct LinkQueue                          //链队结点类型
{    LinkNode<T> *front;                  //指向队首结点
     LinkNode<T> *rear;                   //指向队尾结点
};
```

设计链队类模板 LinkQueueClass<T>如下：

```
template <typename T>
class LinkQueueClass                      //链队类模板
{    LinkQueue<T> *Q;                     //链队结点指针 Q
public:
    LinkQueueClass();                     //构造函数
    ~LinkQueueClass();                    //析构函数
    bool QueueEmpty();                    //判断队列是否为空
    void enQueue(T e);                    //进队算法
    bool deQueue(T &e);                   //出队算法
};
```

图 3.24 说明了一个链队 Q 的动态变化过程。图 3.24(a)是链队的初始状态，图 3.24(b)是在链队中进队 3 个元素后的状态，图 3.24(c)是在链队中出队一个元素后的状态。

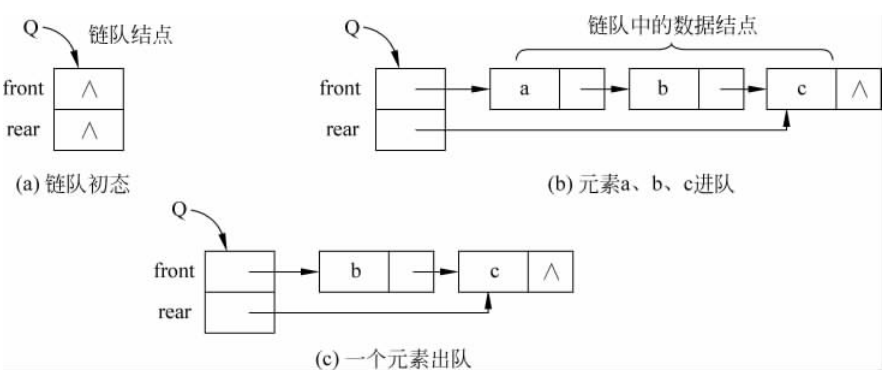

图 3.24　一个链队的动态变化过程

从图 3.24 中可以看到，初始时置 Q->rear=Q->front=NULL，队空的条件为 Q->rear==NULL、Q->front==NULL 或 Q->front==Q->rear==NULL，这里不妨设队空的条件为 Q->rear==NULL。由于只有内存溢出时才出现队满，通常不考虑这样的情况，所以可以看成在链队中不存在队满。结点 *p 进队的操作是在单链表尾部插入结点 *p，并让队尾指针指向它；出队的操作是取出队头所指结点的 data 值并将其从链队中删除。对应队列的基本运算算法如下。

(1) 链队的初始化和销毁

链队的初始化和销毁分别通过构造函数和析构函数来实现，对应的算法如下：

```
template <typename T>
LinkQueueClass<T>::LinkQueueClass()       //构造函数
{    Q = new LinkQueue<T>();              //创建链队结点
```

```
    Q->front = NULL;                          //置队头为空
    Q->rear = NULL;                           //置队尾为空
}
template <typename T>
LinkQueueClass<T>::~LinkQueueClass()          //析构函数
{   LinkNode<T> *pre = Q->front, *p;
    if (pre! = NULL)                          //非空队的情况
    {   if (pre == Q->rear)                   //只有一个数据结点的情况
            delete pre;                       //释放 *pre 结点
        else                                  //有两个或多个数据结点的情况
        {   p = pre->next;
            while (p! = NULL)
            {   delete pre;                   //释放 *pre 结点
                pre = p; p = p->next;         //pre、p 同步后移
            }
            delete pre;                       //释放尾结点
        }
        delete Q;                             //释放链队结点
    }
}
```

(2) 判断队列是否为空：QueueEmpty()

若链队结点的 rear 域值为 NULL，表示队列为空，返回 true，否则返回 false。其对应的算法如下：

```
template <typename T>
bool LinkQueueClass<T>::QueueEmpty()          //判断队列是否为空
{   return(Q->rear == NULL); }
```

(3) 进队列：enQueue(e)

创建 data 域为 e 的数据结点 $*p$，若原队列为空，则将链队结点的两个域均指向 $*p$ 结点，否则将 $*p$ 结点链接到单链表的末尾，并让链队结点的 rear 域指向它。其对应的算法如下：

```
template <typename T>
void LinkQueueClass<T>::enQueue(T e)          //进队算法
{   LinkNode<T> *p = new LinkNode<T>();
    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
    {   Q->rear->next = p;                    //将 *p 结点链接到队尾，并将 rear 指向它
        Q->rear = p;
    }
}
```

(4) 出队列：deQueue(e)

若原队列不为空，则将第一个数据结点的 data 域值赋给 e，并将其删除。若出队之前队列中只有一个结点，则需将链队结点的两个域均置为 NULL，表示队列已为空。其对应的算法如下：

```
template <typename T>
bool LinkQueueClass<T>::deQueue(T &e)          //出队算法
{   LinkNode<T> *p;
    if (Q->rear == NULL) return false;         //队列为空
    p = Q->front;                              //p指向第一个数据结点
    if (Q->front == Q->rear)                   //队列中只有一个结点时
        Q->front = Q->rear = NULL;
    else                                       //队列中有多个结点时
        Q->front = Q->front->next;
    e = p->data;
    delete p;                                  //释放出队结点
    return true;
}
```

【例 3.11】 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，设计出这种链队的进队、出队、判队空和求队中元素个数的算法。

解：如图 3.25 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<T>(为本节前面链队的数据结点类型)。

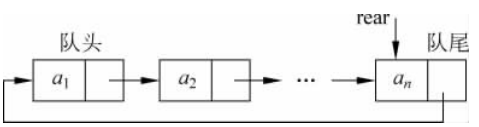

图 3.25　用只有尾结点指针的循环单链表作为队列存储结构

在这样的链队中，没有结点时队列为空，即 rear==NULL，进队在链表的表尾进行，出队在链表的表头进行。本例中包含各种运算的链队类模板 LinkQueueClass2<T>的定义如下：

```
#include <iostream.h>
template <typename T>
class LinkQueueClass2                          //链队类模板
{   LinkNode<T> *rear;                         //链队队尾指针
public:
    LinkQueueClass2();                         //构造函数,用于初始化队列
    ~LinkQueueClass2();                        //析构函数,用于释放队列空间
    bool QueueEmpty();                         //判队空运算算法
    void enQueue(T e);                         //进队运算算法
    bool deQueue(T &e);                        //出队运算算法
    int GetCount();                            //求链队中元素的个数
};
template <typename T>
LinkQueueClass2<T>::LinkQueueClass2()          //构造函数,用于初始化队列
{   rear = NULL; }
template <typename T>
LinkQueueClass2<T>::~LinkQueueClass2()         //析构函数,用于释放队列空间
{   LinkNode<T> *pre = rear, *p;
    if (rear == NULL) return;                  //空队直接返回
    p = pre->next;
```

```
    while (p! = rear)
    {   delete pre;
        pre = p; p = p -> next;
    }
    delete pre;
}
template < typename T >
bool LinkQueueClass2 < T >:: QueueEmpty()        //判队空运算算法
{   return (rear == NULL);  }
template < typename T >
void LinkQueueClass2 < T >:: enQueue(T e)        //进队运算算法
{   LinkNode < T >  * p;
    p = new LinkNode < T >();                    //创建新结点
    p -> data = e;
    if (rear == NULL)                            //原链队为空
    {   p -> next = p;                           //构成循环单链表
        rear = p;
    }
    else
    {   p -> next = rear -> next;                //将 * p 结点插入到 * rear 结点之后
        rear -> next = p;
        rear = p;                                //让 rear 指向这个新插入的结点
    }
}
template < typename T >
bool LinkQueueClass2 < T >:: deQueue(T & e)      //出队运算算法
{   LinkNode < T >  * q;
    if (rear == NULL)     return false;          //队空返回 false
    else if (rear -> next == rear)               //原队中只有一个结点
    {   e = rear -> data;
        delete rear;                             //释放 * rear 结点
        rear = NULL;                             //将 rear 置为 NULL 表示队空
    }
    else                                         //原队中有两个或两个以上的结点
    {   q = rear -> next; e = q -> data;
        rear -> next = q -> next;
        delete q;                                //释放出队结点
    }
    return true;
}
template < typename T >
int LinkQueueClass2 < T >:: GetCount()           //求链队中元素的个数
{   LinkNode < T >  * p = rear;
    if (rear == NULL) return 0;                  //空链队返回 0
    p = rear -> next;
    int i = 1;
    while (p! = rear)
    {   i++;
        p = p -> next;
    }
    return i;
}
```

3.2.4 队列的应用示例

本节用队列求解迷宫问题来讨论队列的应用。

1. 问题描述

参见 3.1.4 小节的用栈求解迷宫问题。

2. 数据组织

用队列解决求迷宫路径问题,使用一个非循环顺序队 qu,需将所有试探的方块均进队。如图 3.26 所示,假设当前方块为(i,j),它进队后在 qu 中的下标为 k,然后找它所有的相邻方块,假设有 4 个相邻方块可走,则这 4 个相邻可走的方块均进队,它们的下标分别为 k_1～k_4,将方块(i,j)称为父方块,它的可走的相邻方块称为子方块。当找到出口时,队列 qu 中的方块数可能很多,不像求解同一问题的栈恰好保存迷宫路径上的方块,所以需要通过回推导出从出口到入口的逆路径,为此将进队的每个方块设置一个 pre,它保存父方块在队列中的下标(位置),入口方块的 pre 值置为－1。

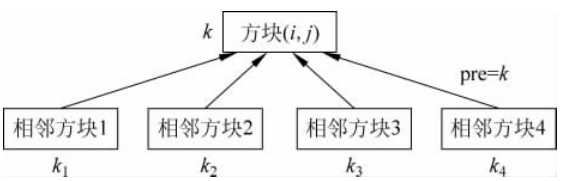

图 3.26　当前方块和相邻方块

每个方块的类型定义如下:

```
struct Box                              //方块结构体类型
{    int i;                             //方块的行号
     int j;                             //方块的列号
     int pre;                           //本路径中父方块在队列中的下标
};
```

经过分析,求解迷宫问题中的队列类的设计如下:

```
class Queue                             //非循环顺序队列类
{    Box *data;                         //存放队中方块
     int front,rear;                    //队头、队尾指针
public:
     Queue();                           //构造函数,用于初始化队列
     ~Queue();                          //析构函数,用于释放队列空间
     bool QueueEmpty();                 //判断队列是否为空
     void edQueue(int x,int y,int pre1);        //进队一个方块
     void deQueue(int &x,int &y,int &cfront);   //出队一个方块
     void DispBox(int cfront);          //从 cfront 出发找一条迷宫路径
};
```

说明:这里使用的顺序队列 qu 不是循环队列,因此在出队时不会将出队元素真正地从队列中删除,因为要利用它们输出迷宫路径。

3. 设计运算算法

查找从(xi,yi)到(xe,ye)路径的过程是,首先将(xi,yi)进队,在队列 qu 不为空时循环。如果出队一次(由于不是循环队列,该出队元素仍在队列中),称该出队的方块为当前方块,qu.front 为该方块在队列中的下标位置。如果当前方块是出口,则按入口到出口的次序输出该路径并结束。

否则,按顺时针方向找出当前方块的 4 个方位中可走的相邻方块(对应的迷宫 a 数组值为 0),将这些可走的相邻方块均插入到队列 qu 中,其 pre 设置为本搜索路径中上一方块在 qu 中的下标值,也就是当前方块的 qu.front 值,并将相邻方块对应的 a 数组元素值置为 −1,以免回过来重复搜索。如果此队列为空,表示未找到出口,即不存在路径。

实际上,本算法的思想是从(xi,yi)开始,利用队列的特点,一层一层向外扩展可走的点,直到找到出口为止,这个方法就是将在第 7 章介绍的广度优先搜索方法。

在找到路径后,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 cfront)的 pre 值回推找到迷宫路径。对于图 3.13 所示的迷宫,在找到路径后,队列 qu 中 data 的所有数据如表 3.3 所示。当前的 front=40(8,8),qu.data[40].pre 为 35(8,7),表示路径的上一方块为 qu.data[35];qu.data[35].pre 为 30(8,6),表示路径的上一方块为 qu.data[30];qu.data[30].pre 为 27(8,5),表示路径的上一方块为 qu.data[27],…,如此找到入口为 qu.data[0]。在 print 函数中,为了正向输出路径,在前面的回推过程中修改路

表 3.3　队列 qu 中 data 的数据

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				

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,从而输出了正向的迷宫路径。在 Queue 类中实现输出路径功能的 DispBox 成员函数如下:

```
void Queue::DispBox(int cfront)                    //从 cfront 出发找一条迷宫路径
{   Box path[QueueSize];
    int k,d = -1;
    k = cfront;
    while (k! = -1)                                //找到入口为止
    {   d++;                                       //将一个方块保存到 path 中
        path[d].i = data[k].i; path[d].j = data[k].j;
        k = data[k].pre;                           //回推找上一个方块
    }
    cout << "一条迷宫路径如下:\n";
    for (k = d;k >= 0;k -- )                       //反向输出构成一条正向的迷宫路径
    {   cout << "  (" << path[k].i << "," << path[k].j << ")";
        if ((d - k + 1) % 5 == 0) cout << endl;    //每行输出 5 个方块
    }
    cout << endl;
}
```

Queue 类中方块进队和出队的成员函数设计如下:

```
void Queue::edQueue(int x,int y,int pre1)          //进队一个方块
{   rear++;
    data[rear].i = x; data[rear].j = y; data[rear].pre = pre1;
}
void Queue::deQueue(int &x,int &y,int &cfront)     //出队一个方块
{   front++;
    x = data[front].i; y = data[front].j; cfront = front;
}
```

最后设计用队列求解从入口(x*i*,y*i*)到出口(x*e*,y*e*)的一条迷宫路径的类 Maze2 如下:

```
class Maze2                                        //用队列求解一条迷宫路径类
{   int a[MaxSize][MaxSize];                       //迷宫数组
    int m,n;                                       //迷宫行、列数
public:
    void Seta(int mg[][MaxSize],int m1,int n1);    //设置迷宫数组
    bool mgpath(int xi,int yi,int xe,int ye);      //求(xi,yi)到(xe,ye)的一条迷宫路径
};
void Maze2::Seta(int mg[][MaxSize],int m1,int n1)  //设置迷宫数组
{   int i,j;
    m = m1; n = n1;
    for (i = 0;i < m;i++)
        for (j = 0;j < n;j++)
            a[i][j] = mg[i][j];
}
bool Maze2::mgpath(int xi,int yi,int xe,int ye)    //求(xi,yi)到(xe,ye)的一条迷宫路径
{   int i,j,di,cfront,i1,j1;
```

```
    Queue qu;                           //创建一个空队 qu
    qu.edQueue(xi,yi,-1);               //入口进队,其 pre 置为 -1
    a[xi][yi] = -1;                     //为避免来回找相邻方块,将进队的方块置为 -1
    while (!qu.QueueEmpty())            //队不空时循环
    {   qu.deQueue(i,j,cfront);         //出队一个方块,该方块仍在队列中
        if (i == xe && j == ye)         //找到了出口,输出路径
        {   qu.DispBox(cfront);         //从 cfront 出发回推导出迷宫路径并输出
            return true;                //找到一条路径时返回 true
        }
        for (di = 0;di < 4;di++)        //循环扫描每个方位,把每个可走的方块进队
        {   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 (a[i1][j1] == 0)         //找到一个相邻的可走方块
            {   qu.edQueue(i1,j1,cfront);
                                        //将该相邻方块进队,并置其 pre 为父方块下标 cfront
                a[i1][j1] = -1;         //为避免来回找相邻方块,将进队的方块置为 -1
            }
        }
    }
    return false;                       //未找到任何路径时返回 false
}
```

4. 设计主函数

以下主函数用于求图 3.13 所示的迷宫图中从(1,1)到(8,8)的一条迷宫路径：

```
void main()
{   int mg[M][N] = {{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} };
    Maze2 mz;                           //创建一个 Maze2 对象 mz
    mz.Seta(mg,10,10);                  //设置迷宫数组
    cout << "求(1,1)到(8,8)的迷宫路径\n";
    if (!mz.mgpath(1,1,8,8))            //求入口为(1,1)、出口为(8,8)的迷宫路径
        cout << "不存在迷宫路径\n";
}
```

5. 程序执行结果

该程序的执行结果如下：

```
求(1,1)到(8,8)的迷宫路径
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)
```

该路径如图 3.27 所示(用浅阴影表示迷宫路径,方块上的数字表示该方块在队列 qu 中的下标),显然这个解是最优解,也就是最短路径。

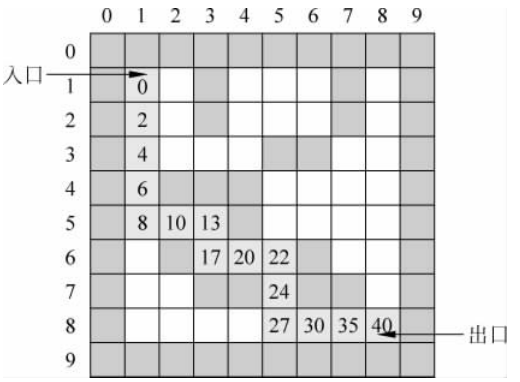

图 3.27　用队列求出的一条迷宫路径

本章小结

本章的学习要点如下:

(1) 理解栈和队列的特性以及它们之间的差异,知道在何时使用哪种数据结构。

(2) 重点掌握在顺序栈上和链栈上实现栈的基本运算算法,注意栈满和栈空的条件。

(3) 重点掌握在顺序队上和链队上实现队列的基本运算算法,注意循环队队满和队空的条件。

(4) 灵活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. 单项选择题

(1) 若元素 a、b、c、d、e、f 依次进栈,允许进栈、出栈操作交替进行,但不允许连续 3 次出栈,则不可能得到的出栈序列是_____。

A. dcebfa　　B. cbdaef　　C. bcaefd　　D. afedcb

(2) 一个栈的进栈序列是 a、b、c、d、e,则不可能的栈的输出序列是_____。

A. edcba　　B. decba　　C. dceab　　D. abcde

(3) 已知一个栈的进栈序列是 1,2,3,…,n,其输出序列的第一个元素是 i($1 \leqslant i \leqslant n$),则第 j($1 \leqslant j \leqslant n$)个出栈元素是_____。

A. i　　B. $n-i$　　C. $j-i+1$　　D. 不确定

(4) 已知一个栈的进栈序列是 1,2,3,…,n,其输出序列是 $p_1, p_2, \cdots, p_n$,若 $p_1 = n$,则 p_i 的值_____。

A. i　　B. $n-i$　　C. $n-i+1$　　D. 不确定

(5) 设有 5 个元素,其进栈序列是 a、b、c、d、e,其输出序列是 c、e、d、b、a,则该栈的容量至少是_____。

A. 1　　B. 2　　C. 3　　D. 4

(6) 表达式(a+a * b) * a+c * b/a 的后缀表达式是_____。

A. a a b * + a * c b * a / +　　B. a a * b + a * c b * a / +

C. a a b * a * c b * + a / +　　D. a a b * + a c b * a / + *

(7) 若一个栈用数组 data[1..n]存储,初始栈顶指针 top 为 n+1,则以下元素 x 进栈的正确操作是_____。

A. top++;data[top]=x;　　B. data[top]=x;top++;

C. top--;data[top]=x;　　D. data[top]=x;top--;

(8) 若一个栈用数组 data[1..n]存储,初始栈顶指针 top 为 n,则以下元素 x 进栈的正确操作是_____。

A. top++;data[top]=x;　　B. data[top]=x;top++;

C. top--;data[top]=x;　　D. data[top]=x; top--;

(9) 若一个栈用数组 data[1..n]存储,初始栈顶指针 top 为 0,则以下元素 x 进栈的正确操作是_____。

A. top++;data[top]=x;　　B. data[top]=x;top++;

C. top--;data[top]=x;　　D. data[top]=x; top--;

(10) 若一个栈用数组 data[1..n]存储,初始栈顶指针 top 为 1,则以下元素 x 进栈的正确操作是_____。

A. top++;data[top]=x;　　B. data[top]=x;top++;

C. top--;data[top]=x;　　D. data[top]=x; top--;

(11) 栈和队列的共同点是_____。

A. 都是先进后出　　B. 都是后进先出

C. 只允许在端点处插入和删除元素　　D. 没有共同点

(12) 栈和队列的不同点是_____。

A. 都是线性表

B. 都不是线性表

C. 栈只能在一端进行插入、删除操作,而队列在不同端进行插入、删除操作

D. 没有不同点

(13) 设循环队列中数组的下标是 0～N−1,其队头、队尾指针分别为 f 和 r(f 指向队首元素的前一位置,r 指向队尾元素),则其元素个数为_____。

A. $r-f$　　B. $r-f-1$

C. $(r-f)\%N+1$　　D. $(r-f+N)\%N$

(14) 设循环队列的存储空间为 a[0..20],且当前队头指针和队尾指针的值分别为 8 和 3,则该队列中元素的个数为_____。

A. 5　　B. 6　　C. 16　　D. 17

(15) 若用一个大小为 6 的数组来实现循环队列,且当前 rear 和 front 的值分别为 0 和 3,当从队列中删除一个元素再加入两个元素后,rear 和 front 的值分别为_____。

A. 1 和 5　　B. 2 和 4　　C. 4 和 2　　D. 5 和 1

2. 问答题

(1) 简述线性表、栈和队列的异同。

(2) 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,D 第二个出栈)的次序有哪几个?

(3) 在一个算法中需要建立 $n(n\geqslant 3)$个栈时可以选择下列 3 种方案之一,试问这 3 种方案各有什么优缺点?

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。

(4) 设栈 S 和队列 Q 的初始状态为空,元素 a、b、c、d、e、f、g、h 依次通过栈 S,每个元素出栈后立即进入队列 Q,若 8 个元素出队列的顺序是 c、f、g、e、h、d、b、a,则栈 S 的容量至少应该是多少?

3. 算法设计题

(1) 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,写出一个算法判断所给的操作序列是否合法,若合法返回 1,否则返回 0(假设被判断的操作序列已存入一维数组中)。

(2) 假设表达式中允许包含圆括号、方括号和大括号,编写一个算法判断表达式中的括号是否正确匹配。

(3) 用一个一维数组 S(设大小为 MaxSize)作为两个栈的共享空间,说明共享方法,栈满、栈空的判断条件,并用 C/C++语言设计公用的初始化栈运算 InitStack1(st)、判栈空运算 StackEmpty1(st,i)、进栈运算 Push1(st,i,x)和出栈运算 Pop1(st,i,x),其中,i 为 1 或 2,用于表示栈号,x 为入栈或出栈元素。

(4) 设计一个循环队列,用 front 和 rear 分别作为队头指针和队尾指针,另外用 tag 标识队列可能空(0)或可能满(1),这样加上 front==rear 可以作为队空或队满的条件,要求设计队列的相关基本运算算法。

(5) 已知一循环队列的存储空间是 data[m..n],其中 $n>m$,队头和队尾指针分别为 front 和 rear,完成以下各小题:

① 设计该队列的类型。

② 设计相应的初始化队列、判队空否、进队和出队算法。

③ 求队中元素的个数并设计相应的算法。