CHAPTER 3

栈和队列 第3章

3.1 问答题及其参考答案

3.1.1 问答题

1. 简述线性表、栈和队列的异同。

2. 设输入元素为1、2、3、P和A，进栈次序为123PA，元素经过栈后到达输出序列，当所有元素均到达输出序列后，有哪些序列可以作为高级语言的变量名？

3. 假设以I和O分别表示进栈和出栈操作，则初态和终态为栈空的进栈和出栈操作序列可以表示为仅由I和O组成的序列，称可以实现的栈操作序列为合法序列（例如IIOO为合法序列，IOOI为非法序列）。试给出区分给定序列为合法序列或非法序列的一般准则。

4. 有n个不同元素的序列经过一个栈产生的出栈序列个数是多少？

5. 若一个栈的存储空间是data[0..$n-1$]，则对该栈的进栈和出栈操作最多只能执行n次。这句话正确吗？为什么？

6. 若采用数组data[0..$m-1$]存放栈元素，回答以下问题：

（1）只能以data[0]端作为栈底吗？

（2）为什么不能以data数组的中间位置作为栈底？

7. 链栈只能顺序存取，而顺序栈不仅可以顺序存取，还能够随机存取。这句话正确吗？为什么？

8. 什么叫队列的“假溢出”？如何解决假溢出？

9. 假设循环队列的元素存储空间为data[0..$m-1$]，队头指针f指向队头元素，队尾指针r指向队尾元素的下一个位置（例如data[0..5]，若队头元素为data[2]，则front=2，若队尾元素为data[3]，则rear=4），则在少用一个元素空间的前提下，表示队空和队满的条件各是什么？

10. 在算法设计中有时需要保存一系列临时数据元素，如果先保存的后

处理,应该采用什么数据结构存放这些元素?如果先保存的先处理,应该采用什么数据结构存放这些元素?

3.1.2 问答题参考答案

1. 答:线性表、栈和队列的相同点是它们的元素的逻辑关系都是线性关系;不同点是运算不同,线性表可以在两端和中间任何位置插入和删除元素,而栈只能在一端插入和删除元素,队列只能在一端插入元素,在另外一端删除元素。

2. 答:高级语言变量名的定义规则是以字母开头的字母数字串。进栈次序为123PA,以A最先出栈的序列为AP321,以P最先出栈的序列为P321A、P32A1、P3A21、PA321。可以作为高级语言的变量名的序列为AP321、P321A、P32A1、P3A21和PA321。

3. 答:合法的栈操作序列必须满足以下两个条件。

① 在操作序列的任何前缀(从开始到任何一个操作时刻)中,I的个数不得少于O的个数。

② 整个操作序列中I和O的个数相等。

4. 答:设 n 个不同元素的序列经过一个栈产生的出栈序列(顺序)的个数是 $f(n)$,设该输入序列为 $a,b,c,d,\cdots$,出栈序列有 n 个位置,元素 a 的各种可能性如下。

① 若元素 a 在出栈序列的第1个位置,则其操作是 a 进栈,a 出栈,还剩下 $n-1$ 个元素,出栈序列个数是 $f(n-1)$,这种情况下的出栈序列个数等于 $f(n-1)$。

② 若元素 a 在出栈序列的第2个位置,则一定有一个元素比 a 先出栈,即有 $f(1)$ 种可能的顺序(只能是 b),还剩 c、d、$\cdots$,其顺序个数是 $f(n-2)$。根据乘法原理,顺序个数为 $f(1)\times f(n-2)$。

③ 如果元素 a 在出栈序列的第3个位置,那么一定有两个元素比 a 先出栈,即有 $f(2)$ 种可能顺序(只能是 b、c),还剩 d、$\cdots$,其顺序个数是 $f(n-3)$。根据乘法原理,顺序个数为 $f(2)\times f(n-3)$。

以此类推,按照加法原理,假设 $f(0)=1$,有

$$f(n)=\sum_{i=0}^{n-1} f(i)\times f(n-1-i)$$

可以求出

$$f(n)=\frac{1}{n+1}C_{2n}^{n}=\frac{(2n)!}{(n+1)\times(n!)^2}$$

例如,$n=3$ 时,

$$f(3)=\frac{1}{3+1}C_{6}^{3}=\frac{6\times5\times4}{4\times3\times2\times1}=5。$$

$n=4$ 时,

$$f(4)=\frac{1}{4+1}C_{8}^{4}=\frac{8\times7\times6\times5}{5\times4\times3\times2\times1}=14。$$

5. 答:错误。从理论上讲,对该栈的进栈和出栈操作次数没有限制,但连续的进栈操作最多只能执行 n 次。

6. 答：(1) 也可以将 data[$m-1$]端作为栈底。

(2) 栈中元素是从栈底向栈顶方向生长的，如果以 data 数组的中间位置作为栈底，那么栈顶方向的另外一端空间就不能使用，造成空间浪费，所以不能以 data 数组的中间位置作为栈底。

7. 答：栈具有顺序存取特性，假设从栈底到栈顶的元素是 $a_0,a_1,\cdots,a_{n-2},a_{n-1}$，出栈栈顶元素 a_{n-1} 后，下一次可以出栈新栈顶元素 a_{n-2}，以此类推，这称为顺序存取特性。链栈和顺序栈都是栈的存储结构，体现栈的特性，都只能顺序存取，而不能随机存取。

8. 答：在非循环顺序队中，当队尾指针已经到了数组的上界，不能再做进队操作，但其实数组中还有空位置，这就叫“假溢出”。解决假溢出的方式之一是采用循环队列。

9. 答：一般教科书中设计循环队列时，让队头指针 f 指向队头元素的前一个位置，队尾指针 r 指向队尾元素。这里是队头指针 f 指向队头元素，队尾指针 r 指向队尾元素的下一个位置。这两种方法本质上没有差别，实际上最重要的是能够方便设置队空、队满的条件。

对于题目中指定的循环队列，f、r 的初始值为 0，仍然以 $f==r$ 作为队空的条件，$(r+1)\%m==f$ 作为队满的条件。

元素 x 进队操作：data[r]$=x$；$r=(r+1)\%m$。队尾指针 r 指向队尾元素的下一个位置。

元素 x 出队操作：$x=$data[f]；$f=(f+1)\%m$。队头元素出队后，下一个元素成为队头元素。

10. 答：如果先保存的后处理，则应该采用栈数据结构存放这些元素。如果先保存的先处理，则应该采用队列数据结构存放这些元素。

3.2　算法设计题及其参考答案

3.2.1　算法设计题

1. 给定一个字符串 str，设计一个算法，采用顺序栈判断 str 是否为形如“序列 1@序列 2”的合法字符串，其中序列 2 是序列 1 的逆序，在 str 中恰好只有一个@字符。

2. 假设有一个链栈 st，设计一个算法，出栈从栈顶开始的第 k 个结点。

3. 设计一个算法，利用顺序栈将一个十进制正整数 d 转换为 $r(2\leqslant r\leqslant 16)$ 进制的数，要求 r 进制数采用字符串 string 表示。

4. 用于列车编组的铁路转轨网络是一种栈结构，如图 3.1 所示。其中，右边轨道是输入端，左边轨道是输出端。当右边轨道上的车皮编号顺序为 1、2、3、4 时，如果执行操作进栈、进栈、出栈、进栈、进栈、出栈、出栈、出栈，则在左边轨道上的车皮编号顺序为 2、4、3、1。设计一个算法，给定 n 个整数序列 a 表示右边轨道上的车皮编号顺序，用上述转轨栈对这些车皮重新编号，使得编号为奇数的车皮都排在编号为偶数的车皮的前面，要求产生所有操作的字符串 op 和最终结果字符串 ans。

5. 设计一个算法，利用一个顺序栈将一个循环队列中的所有元素倒过来，队头变队尾，队尾变队头。

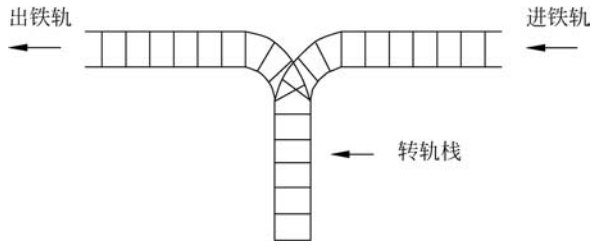

图 3.1　铁路转轨网络

6. 对于给定的正整数 $n(n>2)$,利用一个队列输出 n 阶杨辉三角形。5 阶杨辉三角形如图 3.2(a)所示,其输出结果如图 3.2(b)所示。

图 3.2　5 阶杨辉三角形及其生成过程

7. 有一个整数数组 a,设计一个算法,将所有偶数位的元素移动到所有奇数位的元素的前面,要求它们的相对次序不变。例如,$a=\{1,2,3,4,5,6,7,8\}$,移动后 $a=\{2,4,6,8,1,3,5,7\}$。

8. 设计一个循环队列 QUEUE<T>,用 data[0..MaxSize−1]存放队列元素,用 front 和 rear 分别作为队头和队尾指针,另外用一个标志 tag 标识队列可能空(false)或可能满(true),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。

3.2.2　算法设计题参考答案

1. 解:设计一个栈 st。遍历 str,将其中'@'字符前面的所有字符进栈,再扫描 str 中'@'字符后面的所有字符,对于每个字符 ch,退栈一个字符,如果两者不相同则返回 false。当循环结束时,若 str 扫描完毕并且栈空则返回 true,否则返回 false。对应的算法如下:

```
bool match(string str)
{  SqStack<char> st;                          //定义一个顺序栈
   char e;
   int i=0;
   while (i<str.length() && str[i]!='@')
   {   st.push(str[i]);
       i++;
   }
   if (i==str.length())                       //没有找到@,返回 false
       return false;
   i++;                                       //跳过@
   while (i<str.length() && !st.empty())      //str 没有扫描完毕并且栈不空时循环
```

```
    {   st.pop(e);
        if (str[i]!=e)                          //两者不等返回 false
            return false;
      i++;
    }
    if (i==str.length() && st.empty())          //str 扫描完毕并且栈空时返回 true
        return true;
    else                                        //其他返回 false
        return false;
}
```

2. 解：从链栈头结点 head 开始查找第 $k-1$ 个结点 pre，p 指向其后继结点。本算法是通过结点 pre 删除结点 p 并且取该结点的值，若删除成功则返回 true，若参数错误则返回 false。对应的算法如下：

```
bool popk(LinkStack<int> &st,int k,int &e)
{ if (k<=0) return false;
  LinkNode<int> *pre=st.head, *p;
  int j=0;
  while (pre!=NULL && j<k-1)                    //查找第 k-1 个结点 pre
  {   pre=pre->next;
      j++;
  }
  if (pre==NULL) return false;                  //参数 k 错误
  p=pre->next;                                  //p 指向第 k 个结点
  if (p==NULL) return false;                    //参数 k 错误
  e=p->data;                                    //取结点 p 的值
  pre->next=p->next;                            //删除结点 p
  delete p;
  return true;
}
```

3. 解：设置一个顺序栈 st，采用辗转相除法将十进制数 d 转换成 r 进制数，从低到高产生各个位并进栈，再通过栈从高到低将各个位连接起来生成字符串 s，最后返回 s。对应的算法如下：

```
string trans(int d,int r)
{ int x;
  SqStack<int> st;                              //定义一个顺序栈
  while (d>0)                                   //产生转换后的各个位并进栈
  {   st.push(d%r);
      d/=r;
  }
  string chars="0123456789ABCDEF";
  string s="";
  while (!st.empty())                           //将各个位从高到低连接起来
  {   st.pop(x);
      s+=chars[x];
  }
  return s;
}
```

4. 解：将铁路转轨网络看成一个栈，a 数组表示进栈序列，要求编号为奇数的车皮都排在编号为偶数的车皮的前面，所以遇到奇数的车皮将其进栈保存，遇到偶数的车皮将其进栈后立即出栈，最后将栈中的所有车皮出栈。op 表示操作字符串，ans 表示重编后的车皮序列(初始时均为空串)。对应的算法如下：

```
void solve(int a[],int n,string &op,string &ans)
{ SqStack<int> st;                          //定义一个顺序栈
  for (int i=0;i<n;i++)
  {  if (a[i]%2==1)                         //若车皮编号为奇数,则进栈
     {  st.push(a[i]);
        op+="\t"+to_string(a[i])+"进队\n";
     }
     else                                   //若车皮编号为偶数,则进栈后立即出栈
     {  op+="\t"+to_string(a[i])+"进栈\n";
        op+="\t"+to_string(a[i])+"出栈\n";
        ans+=to_string(a[i])+" ";
     }
  }
  int x;
  while (!st.empty())                       //出栈所有的车皮
  {  st.pop(x);
     op+="\t"+to_string(x)+"出栈\n";
     ans+=to_string(x)+" ";
  }
}
```

5. 解：设置一个顺序栈 st，先将循环队列 qu 中的所有元素出队并进到 st 栈中，再将栈 st 中的所有元素出栈并进到 qu 队列中。对应的算法如下：

```
void Reverse(CSqQueue<int> &qu)
{ int x;
  SqStack<int> st;                          //定义一个顺序栈
  while (!qu.empty())                       //出队所有元素并进栈
  {  qu.pop(x);
     st.push(x);
  }
  while (!st.empty())                       //出栈所有元素并进队
  {  st.pop(x);
     qu.push(x);
  }
}
```

6. 解：由 n 阶杨辉三角形的特点可知，其高度为 n，第 $r(1\leqslant r\leqslant n)$ 行恰好包含 r 个数字。在每行前后添加一个 0(第 r 行包含 $r+2$ 个数字)，采用迭代方式，定义一个队列 qu，由第 r 行生成第 $r+1$ 行：

① 先输出第 1 行，仅输出 1，将 0、1、0 进队。

② 当队列 qu 中包含第 r 行的全部数字时(队列中共 $r+2$ 个元素)，生成并输出第 $r+1$ 行的过程是进队 0，出队元素 s(第一个元素为 0)，再依次出队元素 t(共执行 $r+1$ 次)，$e=s+t$，输出 e 并进队，重置 $t=s$，最后进队 0，这样输出了第 $r+1$ 行的 $r+1$ 个元素，队列中含

$r+3$ 个元素。图 3.3 所示为生成前 3 行的过程。

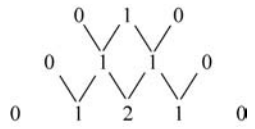

图 3.3　生成前 3 行的过程

对应的算法如下：

```
void YHTriangle(int n)
{ int s,t,e;
  CSqQueue<int> qu;                    //定义一个循环队列
  printf("%4d\n",1);                   //输出第 1 行
  qu.push(0);                          //第 1 行进队
  qu.push(1);
  qu.push(0);
  for (int r=2;r<=n;r++)               //输出第 2 行到第 n 行
  {  qu.push(0);
     qu.pop(s);
     for (int c=0;c<r;c++)             //输出第 r 行的 r 个数字
     {  qu.pop(t);
        e=s+t;
        printf("%4d",e);
        qu.push(e);
        s=t;
     }
     qu.push(0);
     printf("\n");
  }
}
```

7. 解：采用两个队列来实现，先将 a 中的所有奇数位元素进队 qu1 中，所有偶数位元素进队 qu2 中，再将 qu2 中的元素依次出队并放到 a 中，qu1 中的元素依次出队并放到 a 中。对应的算法如下：

```
void Move(int a[],int n)
{ CSqQueue<int> qu1;                   //存放奇数位元素
  CSqQueue<int> qu2;                   //存放偶数位元素
  int i=0,x;
  while (i<n)
  {  qu1.push(a[i]);                   //奇数位元素进 qu1 队
     i++;
     if (i<n)
     {  qu2.push(a[i]);                //偶数位元素进 qu2 队
        i++;
     }
  }
  i=0;
  while (!qu2.empty())                 //先取 qu2 队列的元素
  {  qu2.pop(x);
```

```
        a[i]=x;
        i++;
    }
    while (!qu1.empty())                    //再取 qu1 队列的元素
    {   qu1.pop(x);
        a[i]=x;
        i++;
    }
}
```

8. 解：初始时 tag=false,front=rear=0,成功的进队操作后 tag=true(任何进队操作后队列都不可能空,但可能满),成功的出队操作后 tag=false(任何出队操作后队列都不可能满,但可能空),因此这样的队列的四要素如下。

① 队空条件：front==rear and tag=false

② 队满条件：front==rear and tag=true

③ 元素 x 进队：rear=(rear+1)%MaxSize；data[rear]=x；tag=true；

④ 元素 x 出队：front=(front+1)%MaxSize；x=data[front]；tag=false；

设计对应的循环队列类 QUEUE<T>如下：

```
#define MaxSize 100                         //队列的容量
template <typename T>
class QUEUE                                 //循环队列类模板
{
public:
    T *data;                                //存放队中元素
    int front,rear;                         //队头和队尾指针
    bool tag;                               //为 false 表示可能队空,为 true 表示可能队满
    QUEUE()                                 //构造函数
    {   data=new T[MaxSize];                //为 data 分配容量为 MaxSize 的空间
        front=rear=0;                       //队头、队尾指针置初值
        tag=false;                          //初始时队空,tag 置为 false
    }
    ~QUEUE()                                //析构函数
    {
        delete [] data;
    }
    //-------------循环队列基本运算算法--------------------
    bool empty()                            //判队空运算
    {
        return front==rear && tag==false;
    }
    bool full()                             //判队满运算
    {
        return front==rear && tag==true;
    }
    bool push(T e)                          //进队列运算
    {   if (full())                         //队满上溢出
            return false;
        rear=(rear+1)%MaxSize;
```

```
        data[rear]=e;
        tag=true;                          //进队操作,队可能满
        return true;
    }
    bool pop(T &e)                         //出队列运算
    {   if (empty())                       //队空下溢出
            return false;
        front=(front+1)%MaxSize;
        e=data[front];
        tag=true;                          //出队操作,队可能空
        return true;
    }
    bool gethead(T &e)                     //取队头运算
    {   if (empty())                       //队空下溢出
            return false;
        int head=(front+1)%MaxSize;
        e=data[head];
        return true;
    }
};
```

3.3　基础实验题及其参考答案

3.3.1　基础实验题

1. 设计整数顺序栈的基本运算程序,并用相关数据进行测试。
2. 设计整数链栈的基本运算程序,并用相关数据进行测试。
3. 设计整数循环队列的基本运算程序,并用相关数据进行测试。
4. 设计整数链队的基本运算程序,并用相关数据进行测试。

3.3.2　基础实验题参考答案

1. 解：顺序栈的基本运算算法的设计原理参见《教程》中的第 3.1.2 节。包含顺序栈基本运算算法类 SqStack 以及测试主程序的 Exp1-1.cpp 文件如下：

```
#include <iostream>
using namespace std;
const int MaxSize=100;                     //栈的容量
template <typename T>
class SqStack                              //顺序栈类
{  T *data;                                //存放栈中元素
   int top;                                //栈顶指针
public:
   SqStack()                               //构造函数
   {   data=new T[MaxSize];                //为 data 分配容量为 MaxSize 的空间
       top=-1;                             //栈顶指针初始化
   }
   ~SqStack()                              //析构函数
```

```
    {
        delete [] data;
    }
    //————————————栈基本运算算法————————————————————
    bool empty()                                    //判断栈是否为空
    {
        return(top==-1);
    }
    bool push(T e)                                  //进栈算法
    {   if (top==MaxSize-1)                         //栈满时返回 false
            return false;
        top++;                                      //栈顶指针增 1
        data[top]=e;                                //将 e 进栈
        return true;
    }
    bool pop(T &e)                                  //出栈算法
    {   if (empty()) 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
        top--;                                      //栈顶指针减 1
        return true;
    }
    bool gettop(T &e)                               //取栈顶元素算法
    {   if (empty()) 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;
    }
};
int main()
{ SqStack<char> st;                                 //定义一个字符顺序栈 st
  char e;
  cout << "\n 建立空顺序栈 st\n";
  cout << " 栈 st" << (st.empty()?"空":"不空") << endl;
  cout << " 字符 a 进栈\n"; st.push('a');
  cout << " 字符 b 进栈\n"; st.push('b');
  cout << " 字符 c 进栈\n"; st.push('c');
  cout << " 字符 d 进栈\n"; st.push('d');
  cout << " 字符 e 进栈\n"; st.push('e');
  cout << " 栈 st" << (st.empty()?"空":"不空") << endl;
  st.gettop(e);
  cout << " 栈顶元素:" << e << endl;
  cout << " 所有元素出栈次序: ";
  while (!st.empty())                               //栈不空时循环
  {   st.pop(e);                                    //出栈元素 e 并输出
      cout << e << " ";
  }
  cout << endl;
  cout << " 销毁栈 st" << endl;
  return 0;
}
```

上述程序的一次执行结果如图 3.4 所示。

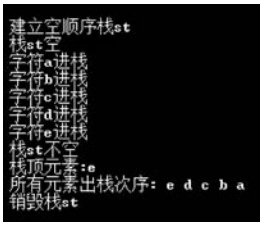

图 3.4　第 3 章基础实验题 1 的执行结果

2. 解：链栈的基本运算算法的设计原理参见《教程》中的第 3.1.4 节。包含链栈基本运算算法类 LinkStack 以及测试主程序的 Exp1-2.cpp 文件如下：

```
#include <iostream>
using namespace std;
template <typename T>
struct LinkNode                                    //链栈结点类型
{  T data;                                         //数据域
   LinkNode *next;                                 //指针域
   LinkNode():next(NULL) {}                        //构造函数
   LinkNode(T d):data(d),next(NULL) {}             //重载构造函数
};
template <typename T>
class LinkStack                                    //链栈类模板
{
public:
   LinkNode<T> *head;                              //链栈的头结点
   LinkStack()                                     //构造函数
   {
      head=new LinkNode<T>();
   }
   ~LinkStack()                                    //析构函数
   {  LinkNode<T> *pre=head, *p=pre->next;
      while (p!=NULL)
      {  delete pre;
         pre=p; p=p->next;                         //pre、p 同步后移
      }
      delete pre;
   }
   bool empty()                                    //判栈空算法
   {
      return head->next==NULL;
   }
   bool push(T e)                                  //进栈算法
   {  LinkNode<T> *p=new LinkNode<T>(e);           //新建结点 p
      p->next=head->next;                          //插入结点 p 作为首结点
      head->next=p;
      return true;
   }
   bool pop(T &e)                                  //出栈算法
   {  LinkNode<T> *p;
```

```
        if (head->next==NULL)                    //栈空的情况
            return false;
        p=head->next;                            //p指向开始结点
        e=p->data;
        head->next=p->next;                      //删除结点p
        delete p;                                //释放结点p
        return true;
    }
    bool gettop(T &e)                            //取栈顶元素
    {   LinkNode<T> *p;
        if (head->next==NULL)                    //栈空的情况
            return false;
        p=head->next;                            //p指向开始结点
        e=p->data;
        return true;
    }
};
int main()
{   LinkStack<char> st;                          //定义一个字符链栈st
    char e;
    cout << "\n 建立空链栈 st\n";
    cout << " 栈 st" << (st.empty()?"空":"不空") << endl;
    cout << " 字符 a 进栈\n"; st.push('a');
    cout << " 字符 b 进栈\n"; st.push('b');
    cout << " 字符 c 进栈\n"; st.push('c');
    cout << " 字符 d 进栈\n"; st.push('d');
    cout << " 字符 e 进栈\n"; st.push('e');
    cout << " 栈 st" << (st.empty()?"空":"不空") << endl;
    st.gettop(e);
    cout << " 栈顶元素:" << e << endl;
    cout << " 所有元素出栈次序: ";
    while (!st.empty())                          //栈不空时循环
    {   st.pop(e);                               //出栈元素e并输出
        cout << e << " ";
    }
    cout << endl;
    cout << " 销毁栈 st" << endl;
    return 0;
}
```

上述程序的一次执行结果如图 3.5 所示。

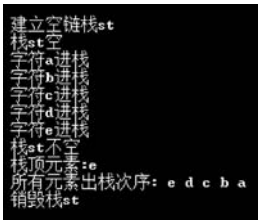

图 3.5　第 3 章基础实验题 2 的执行结果

3. 解：循环队列的基本运算算法的设计原理参见《教程》中的第 3.2.2 节。包含循环队列基本运算算法类 CSqQueue 以及测试主程序的 Exp1-3.cpp 文件如下：

```
#include <iostream>
using namespace std;
#define MaxSize 100                          //队列的容量
template <typename T>
class CSqQueue                               //循环队列类模板
{
public:
  T *data;                                   //存放队中元素
  int front, rear;                           //队头和队尾指针
  CSqQueue()                                 //构造函数
  {  data=new T[MaxSize];                    //为 data 分配容量为 MaxSize 的空间
     front=rear=0;                           //队头、队尾指针置初值
  }
  ~CSqQueue()                                //析构函数
  {
     delete [] data;
  }
  //————————————循环队列基本运算算法————————————————————
  bool empty()                               //判队空算法
  {
     return (front==rear);
  }
  bool push(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;
  }
  bool pop(T &e)                             //出队列运算
  {  if (front==rear)                        //队空下溢出
        return false;
     front=(front+1)%MaxSize;
     e=data[front];
     return true;
  }
  bool gethead(T &e)                         //取队头运算
  {  if (front==rear)                        //队空下溢出
        return false;
     int head=(front+1)%MaxSize;
     e=data[head];
     return true;
  }
};
int main()
{ CSqQueue<char> qu;                         //定义一个字符顺序队 sq
  char e;
```

```
    cout << "\n   建立空顺序队 sq\n";
    cout << "   队列 sq" << (qu.empty()?"空":"不空") << endl;
    cout << "   元素 a 进队\n"; qu.push('a');
    cout << "   元素 b 进队\n"; qu.push('b');
    cout << "   元素 c 进队\n"; qu.push('c');
    cout << "   元素 d 进队\n"; qu.push('d');
    cout << "   元素 e 进队\n"; qu.push('e');
    cout << "   队列 sq" << (qu.empty()?"空":"不空") << endl;
    cout << "   所有元素出队次序: ";
    while (!qu.empty())                                    //队不空时循环
    {   qu.pop(e);                                         //出队元素 e
        cout << e << " ";                                  //输出元素 e
    }
    cout << endl;
    cout << "   销毁队 sq" << endl;
    return 0;
}
```

上述程序的一次执行结果如图 3.6 所示。

图 3.6　第 3 章基础实验题 3 的执行结果

4. 解：链队的基本运算算法的设计原理参见《教程》中的第 3.2.4 节。包含链队基本运算算法类 LinkQueue 以及测试主程序的 Exp1-4.cpp 文件如下：

```
#include <iostream>
using namespace std;
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
  LinkNode():next(NULL) {}                                 //构造函数
  LinkNode(T d):data(d),next(NULL) {}                      //重载构造函数
};
template <typename T>
class LinkQueue                                            //链队类模板
{
public:
  LinkNode<T> *front;                                      //队头指针
  LinkNode<T> *rear;                                       //队尾指针
  LinkQueue():front(NULL),rear(NULL) {}                    //构造函数
  ~LinkQueue()                                             //析构函数
  {   LinkNode<T> *pre=front, *p;
      if (pre!=NULL)                                       //非空队的情况
```

```
    {   if (pre==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
        }
    }
    bool empty()                                //判队空运算
    {
        return rear==NULL;
    }
    bool push(T e)                              //进队运算
    {   LinkNode<T> *p=new LinkNode<T>(e);
        if (rear==NULL)                         //链队为空的情况
            front=rear=p;                       //新结点既是队首结点又是队尾结点
        else                                    //链队不空的情况
        {   rear->next=p;                       //将 p 结点链到队尾,并将 rear 指向它
            rear=p;
        }
        return true;
    }
    bool pop(T &e)                              //出队运算
    {   if (rear==NULL)                         //队列为空
            return false;
        LinkNode<T> *p=front;                   //p 指向首结点
        if (front==rear)                        //队列中只有一个结点时
            front=rear=NULL;
        else                                    //队列中有多个结点时
            front=front->next;
        e=p->data;
        delete p;                               //释放出队结点
        return true;
    }
    bool gethead(T &e)                          //取队头运算
    {   if (rear==NULL)                         //队列为空
            return false;
        e=front->data;                          //取首结点的值
        return true;
    }
};
int main()
{ LinkQueue<char> qu;                           //定义一个字符链队 qu
  char e;
  cout << "\n   建立空链队 qu\n";
  cout << "   队列 qu" << (qu.empty()?"空":"不空") << endl;
  cout << "   元素 a 进队\n"; qu.push('a');
```

```
    cout << "   元素 b 进队\n"; qu.push('b');
    cout << "   元素 c 进队\n"; qu.push('c');
    cout << "   元素 d 进队\n"; qu.push('d');
    cout << "   元素 e 进队\n"; qu.push('e');
    cout << "   队列 qu" << (qu.empty()?"空":"不空") << endl;
    cout << "   所有元素出队次序: ";
    while (!qu.empty())                                    //队不空时循环
    {   qu.pop(e);                                         //出队元素 e
        cout << e << " ";                                  //输出元素 e
    }
    cout << endl;
    cout << "   销毁队 qu" << endl;
    return 0;
}
```

上述程序的一次执行结果如图 3.7 所示。

图 3.7　第 3 章基础实验题 4 的执行结果

3.4　应用实验题及其参考答案

3.4.1　应用实验题

1. 改进用栈求解迷宫问题的算法,累计如图 3.8 所示的迷宫的路径条数,并输出所有的迷宫路径。

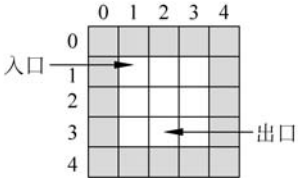

图 3.8　迷宫示意图

2. 括号匹配问题。在某个字符串(长度不超过 100)中有左括号、右括号和大小写字母,规定(与常见的算术表达式一样)任何一个左括号都从内到外与在它右边且距离最近的右括号匹配。编写一个实验程序,找到无法匹配的左括号和右括号,输出原来的字符串,并在下一行标出不能匹配的括号。不能匹配的左括号用“$”标出,不能匹配的右括号用“?”标出。例如,输出样例如下:

```
( (ABCD(x)
 $$
) (rttyy())sss)(
?              ?$
```

3. 修改《教程》第 3 章 3.2 节中的循环队列算法,增加数据成员 length 表示长度,并且其容量可以动态扩展,在进队元素时若容量满则按两倍扩大容量,在出队元素时若当前容量

大于初始容量并且元素的个数只有当前容量的 1/4，则栈当前容量缩小为一半。通过测试数据说明队列容量变化的情况。

4. 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。

5. 设计一个队列类 QUEUE，其包含判断队列是否为空、进队和出队运算。要求用两个栈 st1、st2 模拟队列，其中栈用 stack＜T＞容器表示。

6. 设计一个栈类 STACK，其包含判断栈是否为空、进栈和出栈运算。要求用两个队列 qu1、qu2 模拟栈，其中队列用 queue＜T＞容器表示。

3.4.2　应用实验题参考答案

1. 解：修改《教程》中第 3.1.7 节用栈求解迷宫问题的 mgpath()算法，用 cnt 累计找到的迷宫路径条数(初始为 0)。当找到一条路径后并不返回，而是将 cnt 增加 1，输出该迷宫路径，然后出栈栈顶方块 *b* 并将该方块的 mg 值恢复为 0，继续前面的过程，直到栈空为止，最后返回 cnt。对应的实验程序 Exp2-1.cpp 如下：

```
#include<iostream>
#include<stack>
using namespace std;
const int MAX=10;
int cnt=0;                                  //累计迷宫路径条数
int mg[MAX][MAX]={{1,1,1,1,1},{1,0,0,0,1},{1,0,0,0,1},{1,0,0,0,1},{1,1,1,1,1}};
int m=5,n=5;                                //一个 5 行 5 列的迷宫图
int dx[]={-1,0,1,0};                        //x 方向的偏移量
int dy[]={0,1,0,-1};                        //y 方向的偏移量
struct Box                                  //方块结构体
{  int i;                                   //方块的行号
   int j;                                   //方块的列号
   int di;                                  //di 是下一可走相邻方块的方位号
   Box() {}                                 //构造函数
   Box(int i1,int j1,int di1)               //重载构造函数
   {   i=i1;
       j=j1;
       di=di1;
   }
};
int 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;
   Box b,b1;
   stack<Box> st,st1;                       //定义一个顺序栈
   b=Box(xi,yi,-1);                         //建立入口方块对象
   st.push(b);                              //入口方块进栈
   mg[xi][yi]=-1;                           //为避免来回找相邻方块，置 mg 值为-1
   while (!st.empty())                      //栈不空时循环
   {   b=st.top();                          //取栈顶方块，称为当前方块
       if (b.i==xe && b.j==ye)              //找到了出口，输出栈中的所有方块构成一条路径
       {   cnt++;
           printf("    迷宫路径%d: ",cnt);
           while(!st.empty())               //将 st 的所有方块出栈并进栈 st1
```

```
            {   st1.push(st.top());
                st.pop();
            }
            while (!st1.empty())                    //输出一条迷宫路径
            {   b1=st1.top(); st1.pop();
                st.push(b1);                        //恢复 st 栈
                printf("[%d,%d] ",b1.i,b1.j);
            }
            printf("\n");
            mg[b.i][b.j]=0;                         //让该位置变为其他路径可走方块
            st.pop();                               //退栈
        }
        else
        {   find=false;                             //继续找路径
            di=b.di;
            while (di<3 && find==false)             //找 b 的一个相邻可走方块
            {   di++;                               //找下一个方位的相邻方块
                i=b.i+dx[di]; j=b.j+dy[di];         //找 b 的 di 方位的相邻方块(i,j)
                if (i>=0 && i<m && j>=0 && j<n && mg[i][j]==0)
                    find=true;                      //(i,j)方块有效且可走
            }
            if (find)                               //找到了一个相邻可走方块(i,j)
            {   st.top().di=di;                     //修改栈顶方块的 di 为新值
                b1=Box(i,j,-1);                     //建立相邻可走方块(i,j)的对象 b1
                st.push(b1);                        //b1 进栈
                mg[i][j]=-1;                        //为避免来回找相邻方块,置 mg 值为-1
            }
            else                                    //没有路径可走,则退栈
            {   mg[b.i][b.j]=0;                     //恢复当前方块的迷宫值
                st.pop();                           //将栈顶方块退栈
            }
        }
    }
    return cnt;                                     //返回找到的迷宫路径数
}
int main()
{   int xi=1,yi=1,xe=3,ye=2;
    printf("\n  求(%d,%d)到(%d,%d)的迷宫路径\n",xi,yi,xe,ye);
    int cnt=mgpath(xi,yi,xe,ye);
    printf("  共有%d 条迷宫路径\n",cnt);
    return 0;
}
```

上述程序的执行结果如图 3.9 所示。

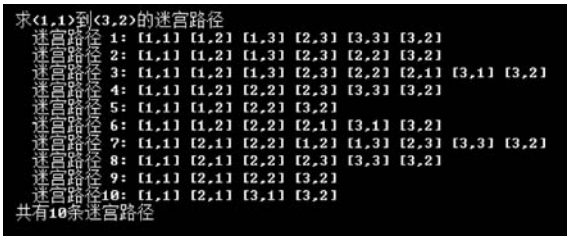

图 3.9　第 3 章应用实验题 1 的执行结果

2. 解：对于字符串 s，设对应的输出字符串为 mark，采用栈 st 来产生 mark。遍历字符 $s[i]$，遇到'('时将其下标 i 进栈，遇到')'时，若栈中存在匹配的'('，置 mark[i]=' '，否则置 mark[i]='?'。当 s 遍历完毕时，若 st 栈不空，则 st 栈中的所有左括号都是没有右括号匹配的，将相应位置 j 的 mark 值置为'$'。对应的实验程序 Exp2-2.cpp 如下：

```
#include<iostream>
#include<stack>
using namespace std;
string solve(string s)                    //求解算法
{ stack<int> st;                          //定义一个栈
  string mark(s.length(),'*');            //定义输出字符串
  for (int i=0;i<s.length();i++)
  {  if (s[i]=='(')                       //遇到'('则入栈
     {  st.push(i);                       //将'('的下标暂存在栈中
        mark[i]=' ';                      //对应输出字符串暂且为' '
     }
     else if (s[i]==')')                  //遇到')'
     {  if (st.empty())                   //栈空,即没有'('相匹配
           mark[i]='?';                   //对应输出字符串改为'?'
        else                              //有'('相匹配
        {  mark[i]=' ';                   //对应输出字符串改为' '
           st.pop();                      //栈顶位置的左括号与其匹配,弹出已经匹配的左括号
        }
     }
     else                                 //其他字符与括号无关
        mark[i]=' ';                      //对应输出字符串改为' '
  }
  while (!st.empty())                     //若栈不空,则都没有匹配的左括号
  {  mark[st.top()]='$';                  //对应输出字符串改为'$'
     st.pop();
  }
  return mark;
}
int main()
{ printf("\n");
  printf("   测试 1\n");
  string s="((ABCD(x)";
  cout << "    表达式: " << s << endl;
  string ans=solve(s);
  cout << "    结　果: " << ans << endl;
  printf("   测试 2\n");
  s=")(rttyy())sss)(";
  cout << "    表达式: " << s << endl;
  ans=solve(s);
  cout << "    结　果: " << ans << endl;
  return 0;
}
```

上述程序的执行结果如图 3.10 所示。

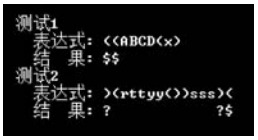

图 3.10 第 3 章应用实验题 2 的执行结果

3. 解：用全局变量 Initcap 存放初始容量，队列中增加 capacity 属性表示队列的当前容量，增加 recap(newcap)方法用于将当前容量改变为 newcap。其过程如下：

① 当参数 newcap 正确时(newcap>n)，建立长度为 newcap 的列表 tmp。

② 出队 data 中的所有元素并依次存放到 tmp 中(从 tmp[1]开始)。

③ 置 data 为 tmp，队头指针 front 为 0，队尾指针 rear 为 n，新容量为 newcap。

在进队中队满和出队中满足指定的条件时调用 recap(newcap)方法。对应的实验程序 Exp2-3.cpp 如下：

```
#include<iostream>
using namespace std;
const int Initcap=3;                    //全局变量,初始容量为 3
template<typename T>
class CSqQueue                          //非循环队列类
{
public:                                 //为了方便测试,将所有成员设置为公有的
  T *data;                              //存放队中元素
  int capacity;                         //data 容量
  int length;                           //队中实际元素个数,即长度
  int front;                            //队头指针
  int rear;                             //队尾指针
  CSqQueue()                            //构造函数
  {  data=new T[Initcap];               //为 data 分配容量为 Initcap 的空间
     capacity=Initcap;                  //设置容量
     front=rear=0;                      //初始化队头和队尾指针
     length=0;                          //初始化长度
  }
  ~CSqQueue()                           //析构函数
  {  delete [] data;  }
  void recap(int newcap)                //改变队列容量为 newcap
  {  if (newcap<length)
        throw("新容量大小错误");        //检测 newcap 参数的错误
     printf("  原容量=%d,原长度=%d,修改容量=%d",capacity,length,newcap);
     T *tmp=new T[newcap];              //新建存放队列元素的空间
     int head=(front+1)%capacity;
     for (int i=0;i<length;i++)         //出队所有元素存放到 tmp[1..length]中
     {  tmp[i+1]=data[head];            //从 tmp[1]开始,tmp[0]暂时不用
        head=(head+1)%capacity;
     }
     delete [] data;                    //释放原 data 空间
     data=tmp;                          //data 指向新空间
     front=0;                           //重置 front
     rear=length;                       //重置 rear
```

```
        capacity=newcap;                    //重置 capacity
    }
    bool empty()                            //判队空运算
    {   return length==0;   }
    bool full()                             //判队满运算
    {   return length==capacity;   }
    bool push(T e)                          //进队列运算
    {   cout << "    进队" << e;
        if (full())                         //队满上溢出
            recap(2 * capacity);            //队满时倍增容量
        printf("\n");
        rear=(rear+1)%capacity;
        data[rear]=e;
        length++;                           //增加一个元素
        return true;
    }
    bool pop(T &e)                          //出队列运算
    {   if (empty())                        //队空下溢出
            return false;
        front=(front+1)%capacity;
        e=data[front];
        length--;                           //减少一个元素
        cout << "    出队" << e;
        if (capacity > Initcap && length==capacity/4)
            recap(capacity/2);              //满足要求则容量减半
        printf("\n");
        return true;
    }
    bool gethead(T &e)                      //取队头运算
    {   if (front==rear)                    //队空下溢出
            return false;
        int head=(front+1)%capacity;
        e=data[head];
        return true;
    }
};
int main()
{  int x;
   printf("\n");
   CSqQueue<int> qu;
   printf("   (1)进队 1,2\n");
   qu.push(1);
   qu.push(2);
   printf("   元素个数=%d,容量=%d\n",qu.length,qu.capacity);
   printf("   (2)进队 3～13\n");
   for (int i=3;i<=13;i++) qu.push(i);
   printf("   元素个数=%d,容量=%d\n",qu.length,qu.capacity);
   printf("   (3)出队所有元素\n");
   while (!qu.empty()) qu.pop(x);
   printf("   元素个数=%d,容量=%d\n",qu.length,qu.capacity);
```

```
    return 0;
}
```

上述程序的执行结果如图 3.11 所示。

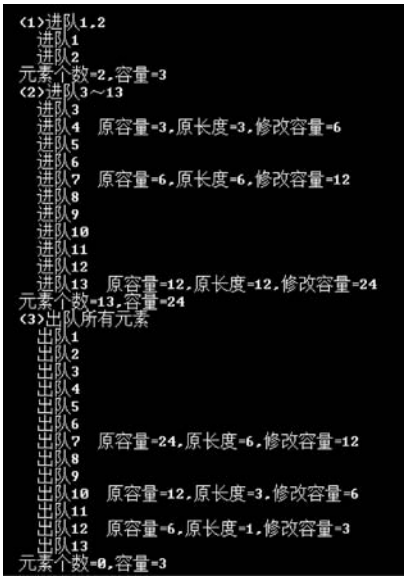

图 3.11　第 3 章应用实验题 3 的执行结果

4. 解：用只有尾结点指针 rear 的循环单链表作为队列存储结构，如图 3.12 所示，其中每个结点的类型为 LinkNode(同第 3 章基础实验题 4 中链队的结点类)。

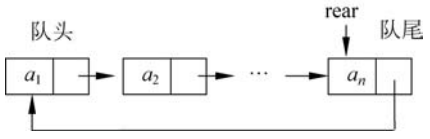

图 3.12　用只有尾结点指针的循环单链表作为队列存储结构

在这样的链队中，队列为空时 rear＝NULL，进队在链表的表尾进行，出队在链表的表头进行。例如，在空链队中进队 a、b、c 元素的结果如图 3.13(a)所示，出队两个元素后的结果如图 3.13(b)所示。

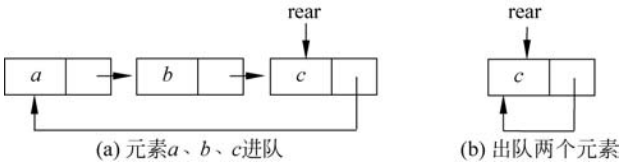

图 3.13　链队的进队和出队操作

对应的实验程序 Exp2-4.cpp 如下：

```
#include <iostream>
using namespace std;
template <typename T>
```

```
struct LinkNode                          //链栈结点类型
{ T data;                                //数据域
  LinkNode *next;                        //指针域
  LinkNode():next(NULL) {}               //构造函数
  LinkNode(T d):data(d),next(NULL) {}    //重载构造函数
};
template < typename T >
class LinkQueue                          //链队类模板
{
public:
  LinkNode < T > *rear;                  //链队的尾结点指针
  LinkQueue():rear(NULL) {}              //构造函数
  ~LinkQueue()                           //析构函数
  {  if (rear==NULL) return;
     LinkNode < T > *pre, *p;
     pre=rear; p=pre->next;
     while (p!=rear)                     //用 p 遍历结点并释放其前驱结点
     {  delete pre;                      //释放 pre 结点
        pre=p; p=p->next;                //pre、p 同步后移一个结点
     }
     delete pre;                         //p 等于 rear 时 pre 指向尾结点,此时释放尾结点
  }
  bool empty()                           //判队空运算
  {
     return rear==NULL;
  }
  bool push(T e)                         //进队运算
  {  LinkNode < T > *p=new LinkNode < T >(e);
     if (rear==NULL)                     //链队为空的情况
     {  rear=p;                          //新结点既是队首结点又是队尾结点
        rear->next=rear;                 //构成循环单链表
     }
     else                                //链队不空的情况
     {  p->next=rear->next;              //将 p 结点插入 rear 结点之后
        rear->next=p;
        rear=p;                          //让 rear 指向 p 结点
     }
     return true;
  }
  bool pop(T &e)                         //出队运算
  {  if (empty()) return false;
     if (rear->next==rear)               //原链队只有一个结点
     {  e=rear->data;                    //取该结点值
        rear=NULL;                       //置为空队
     }
     else                                //原链队有多个结点
     {  e=rear->next->data;              //取队头结点值
        rear->next=rear->next->next;     //删除队头结点
     }
     return true;
  }
  bool gethead(T &e)                     //取队头运算
  {  if (empty()) return false;
     e=rear->next->data;
```

```
        return true;
    }
};
int main()
{  LinkQueue<char> qu;                          //定义一个字符队 qu
   char e;
   cout << "\n   建立一个空队 qu\n";
   cout << "   队 qu" << (qu.empty()?"空":"不空") << endl;
   cout << "   元素 a 进队\n"; qu.push('a');
   cout << "   元素 b 进队\n"; qu.push('b');
   qu.gethead(e); cout << "   队头元素: " << e << endl;
   cout << "   元素 c 进队\n"; qu.push('c');
   cout << "   元素 d 进队\n"; qu.push('d');
   cout << "   元素 e 进队\n"; qu.push('e');
   cout << "   队 qu" << (qu.empty()?"空":"不空") << endl;
   qu.gethead(e); cout << "   队头元素: " << e << endl;
   cout << "   所有元素出队次序: ";
   while (!qu.empty())                          //队不空时循环
   {   qu.pop(e);                               //出队元素 e
       cout << e << " ";                        //输出元素 e
   }
   cout << endl;
   cout << "   销毁队 qu" << endl;
   return 0;
}
```

上述程序的执行结果如图 3.14 所示。

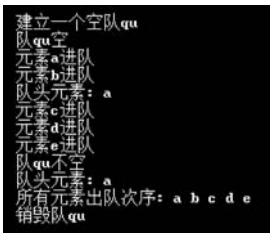

图 3.14　第 3 章应用实验题 4 的执行结果

5. 解：由于栈的特点是先进后出，而队列的特点是先进先出，在用两个栈 st1、st2 模拟队列时，st1 栈负责“进队”，st2 栈负责“出队”(反向)，在 st1 和 st2 都非空时保证 st2 中的元素都是先于 st1 中的元素进队。

队空的条件：栈 st1 和 st2 均为空。

元素 e 进队的操作：此时只需要直接将 e 进到 st1 栈，如图 3.15(a)所示(这里没有考虑栈满，若考虑栈满的情况，当 st1 栈满时先将 st1 的所有元素出栈并进栈 st2，再将 e 进到 st1 栈)。

出队元素 e 的操作：若 st1 和 st2 均为空，则返回 false；若 st2 不空，则 st2 出栈元素 e；若 st2 空但栈 st1 不空，则将栈 st1 中的所有元素出栈并进到 st2 栈中，再从 st2 出栈元素 e，如图 3.15(b)所示。

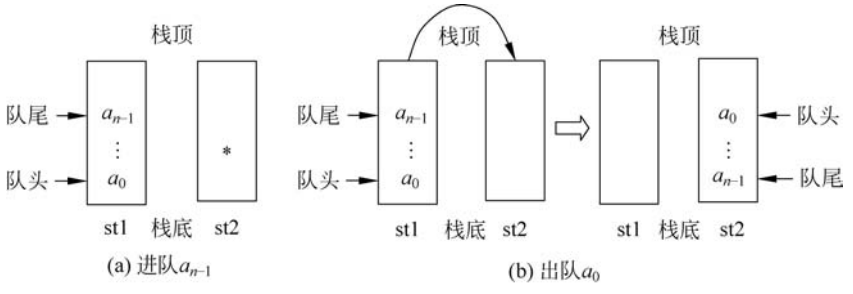

图 3.15　两个栈模拟队列

对应的实验程序 Exp2-5.cpp 如下：

```
#include<iostream>
#include<stack>
using namespace std;
template<typename T>
class QUEUE                                    //两个栈模拟的队列
{ stack<T> st1,st2;
public:
  bool empty()                                 //判队空运算
  {
    return st1.empty() && st2.empty();
  }
  void push(T e)                               //进队运算
  {
    st1.push(e);
  }
  bool pop(T &e)                               //出队运算
  { if (empty()) return false;
    if (!st2.empty())                          //st2 不空时从 st2 出栈元素 e
    { e=st2.top();                             //st2 出栈元素 e
      st2.pop();
    }
    else                                       //st2 空时
    { while (!st1.empty())                     //将栈 st1 中的所有元素出栈并进到 st2 栈中
      { st2.push(st1.top());
        st1.pop();
      }
      e=st2.top();                             //st2 出栈元素 e
      st2.pop();
    }
    return true;
  }
};
int main()
{ QUEUE<int> qu;                               //定义一个整数队 qu
  int e;
  cout<<"\n  建立一个空队 qu\n";
  cout<<"  队 qu"<<(qu.empty()?"空":"不空")<<endl;
  cout<<"  元素 1 进队\n"; qu.push(1);
  cout<<"  元素 2 进队\n"; qu.push(2);
  qu.pop(e); cout<<"  出队元素："<<e<<endl;
  cout<<"  元素 3 进队\n"; qu.push(3);
  cout<<"  元素 4 进队\n"; qu.push(4);
  qu.pop(e); cout<<"  出队元素："<<e<<endl;
  cout<<"  元素 5 进队\n"; qu.push(5);
  cout<<"  队 qu"<<(qu.empty()?"空":"不空")<<endl;
  cout<<"  其他元素出队次序：";
  while (!qu.empty())                          //队不空时循环
  { qu.pop(e);                                 //出队元素 e
    cout<<e<<" ";                              //输出元素 e
```

```
    }
    cout << endl;
    cout << "   销毁队 qu" << endl;
    return 0;
}
```

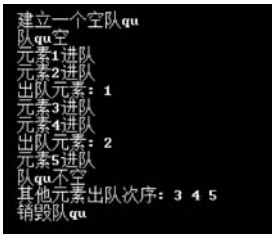

图 3.16　第 3 章应用实验题 5 的执行结果

上述程序的执行结果如图 3.16 所示。

6. 解：由于队列不会改变顺序，在用两个队列 qu1 和 qu2 模拟栈时采用来回倒的方法，保证一个队列是空的，用空队列临时存储队尾外的所有元素。

栈空的条件：两个队列均为空。

元素 e 进栈的操作：总有一个队列是空的，将 e 进到非空队中。假设 qu1 非空，进栈 a_{n-1} 的操作如图 3.17(a)所示。

出队元素 e 的操作：总有一个队列是空的，假设 qu1 非空，先将 qu1 中的前 $n-1$ 个元素 $a_0 \sim a_{n-2}$ 出队并进到 qu2，如图 3.17(b)所示，再从 qu1 出队元素 a_{n-1}。qu2 非空的操作与之类似。

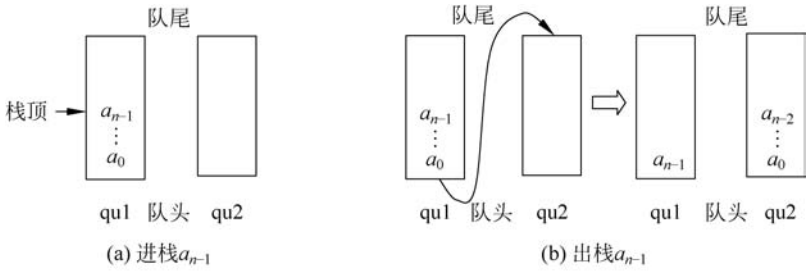

图 3.17　两个队列模拟栈

对应的实验程序 Exp2-6.cpp 如下：

```
#include <iostream>
#include <queue>
using namespace std;
template <typename T>
class STACK                                    //两个队列模拟的栈
{ queue<T> qu1,qu2;
public:
  bool empty()                                 //判栈空运算
  {
     return qu1.empty() && qu2.empty();
  }
  void push(T e)                               //进栈运算
  {  if (!qu1.empty())
        qu1.push(e);
     else
        qu2.push(e);
  }
  bool pop(T &e)                               //出栈运算
  {  if (empty()) return false;
```

```
        if (!qu1.empty())                    //qu1 不空时
        {   while (qu1.size()>1)             //qu1 中的前 n-1 个元素出队并进到 qu2
            {   qu2.push(qu1.front());
                qu1.pop();
            }
            e=qu1.front(); qu1.pop();        //qu1 出队最后一个元素 e
        }
        else                                 //qu1 空时
        {   while (qu2.size()>1)             //qu2 中的前 n-1 个元素出队并进到 qu1
            {   qu1.push(qu2.front());
                qu2.pop();
            }
            e=qu2.front(); qu2.pop();
        }
        return true;
    }
};
int main()
{  STACK<int> st;                            //定义一个整数栈 st
   int e;
   cout << "\n   建立一个空栈 st\n";
   cout << "   栈 st" << (st.empty()?"空":"不空") << endl;
   cout << "   元素 1 进栈\n"; st.push(1);
   cout << "   元素 2 进栈\n"; st.push(2);
   st.pop(e); cout << "   出栈元素: " << e << endl;
   cout << "   元素 3 进栈\n"; st.push(3);
   cout << "   元素 4 进栈\n"; st.push(4);
   st.pop(e); cout << "   出栈元素: " << e << endl;
   cout << "   元素 5 进栈\n"; st.push(5);
   cout << "   栈 st" << (st.empty()?"空":"不空") << endl;
   cout << "   其他元素出栈次序: ";
   while (!st.empty())                       //队不空时循环
   {   st.pop(e);                            //出队元素 e
       cout << e << " ";                     //输出元素 e
   }
   cout << endl;
   cout << "   销毁栈 st" << endl;
   return 0;
}
```

上述程序的执行结果如图 3.18 所示。

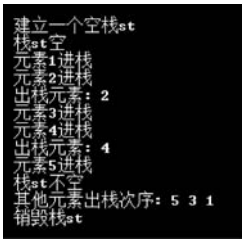

图 3.18　第 3 章应用实验题 6 的执行结果