第3章 栈和队列

3.1 初级实验 1

一、实验目的

掌握栈数据结构的实现，并应用栈实现输入序列的反转，从而验证各个栈的基本操作。

二、实验内容

栈的实现分别用顺序栈和链栈实现，要求至少具有以下基本接口，并在主程序中调用这些接口实现输入序列的反转。

(1) 创建空栈；

(2) 进栈；

(3) 出栈；

(4) 判断栈是否为空；

(5) 取栈顶元素值；

(6) 判断栈是否满(如果用顺序栈)。

三、参考代码

1. 顺序栈

(1) 本程序的文件结构如图 3-1 所示，说明如下。

图 3-1 程序的文件结构图

① SeqStack.h：顺序栈头文件，提供了顺序栈的数据结构类型定义和接口说明。

② SeqStack.c：顺序栈接口的实现文件。

③ main.c：主函数，使用顺序栈实现序列的反转。

(2) SeqStack.h。

```
1   #ifndef SEQSTACK_H
2   #define SEQSTACK_H
3   typedef int DataType;
```

```
struct SeqStack
{
    int MAX;                          //最大容量
    int top;                          //栈顶指针
    DataType * elem;                  //存放元素的起始指针
};
typedef struct SeqStack * SeqStack;
//函数功能:创建空顺序栈
//输入参数 m:顺序栈的最大容量
//返回值: 空的顺序栈
SeqStack SetNullStack_Seq(int m);
//函数功能:判断一个栈是否为空
//输入参数 sstack: 顺序栈
//返回值: 空栈返回 1,否则返回 0
int IsNullStack_seq(SeqStack sstack);
//函数功能:进栈
//输入参数 sstack: 顺序栈
//输入参数 x: 进栈元素
//返回值: 无
void Push_seq(SeqStack sstack, int x);
//函数功能:出栈
//输入参数 sstack: 顺序栈
//返回值: 无
void Pop_seq(SeqStack sstack);
//函数功能:求栈顶元素的值
//输入参数 sstack: 顺序栈
//返回值: 栈顶元素的值
DataType Top_seq(SeqStack sstack);
#endif
```

(3) SeqStack.c。

```
#include <stdlib.h>
#include <stdio.h>
#include "SeqStack.h"
SeqStack SetNullStack_Seq(int m)                    //创建空顺序栈
{
    SeqStack sstack = (SeqStack)malloc(sizeof(struct SeqStack));
    if (sstack!= NULL)
    {
        sstack -> elem = (int * )malloc(sizeof(int) * m);
         if (sstack -> elem!= NULL)
         {
            sstack -> MAX = m;
            sstack -> top =- 1;
            return(sstack);
        }
        else
        {
            free(sstack);
            printf("out of space");
```

```
            return NULL;
        }
    }
    else
    {
        printf("out of space");
        return NULL;
    }
}
int IsNullStack_seq(SeqStack sstack)                    //判断一个栈是否为空
{
    return(sstack->top==-1);
}
void Push_seq(SeqStack sstack, int x)                   //入栈
{
    if (sstack->top>=(sstack->MAX - 1))                 //检查栈是否满
        printf("overflow! \n");
    else
    {
        sstack->top++;                                  //若不满,先修改栈顶变量
        sstack->elem[sstack->top]=x;                    //把元素 x 放到栈顶变量的位置中
    }
}
void Pop_seq(SeqStack sstack)                           //出栈
{
    if (IsNullStack_seq(sstack))                        //判断栈是否为空
        printf("Underflow!\n");
    else
        sstack->top=sstack->top-1;                      //栈顶减 1
}
DataType Top_seq(SeqStack sstack)                       //求栈顶元素的值
{
    if (IsNullStack_seq(sstack))                        //判断 sstack 所指的栈是否为空栈
    {
        printf("it is empty");
        return 0;
    }
    else
        return sstack->elem[sstack->top];
}
```

(4) main.c。

```
//写出主程序,对栈的所有基本算法进行测试
//要求输入一个序列,通过栈实现序列的反转
#include<stdlib.h>
#include<stdio.h>
#include "SeqStack.h"
int main(void)
{
    SeqStack p=SetNullStack_Seq(5);                     //创建一个空栈
```

```
9       int data;
10      printf("请输入进栈的元素,以 0 结束:");
11      scanf_s(" % d", &data);
12      while(data!= 0)
13      {
14          Push_seq(p, data);                          //进栈
15          scanf_s(" % d,", &data);
16      }
17      printf("出栈元素的顺序是:");
18      while (!IsNullStack_seq(p))                     //是否为空栈
19      {
20          printf(" % d ", Top_seq(p));                //取栈顶元素
21          Pop_seq(p);                                 //出栈
22      }
23      printf("\n");
24      return 0;
25  }
```

(5) 测试用例和测试结果。测试用例和测试结果截图如图 3-2 所示。

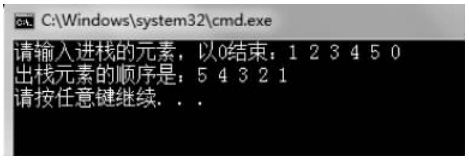

图 3-2　测试截图

2. 链栈

(1) 本程序的文件结构如图 3-3 所示,说明如下。

① LinkStack.h：链栈头文件,提供了链栈的数据结构类型定义和接口说明。

② LinkStack.c：链栈接口的实现文件。

③ main.c：主函数,使用链栈实现序列的反转。

图 3-3　程序的文件结构图

(2) LinkStack.h。

```
1   #ifndef LINKSTACK_H
2   #define LINKSTACK_H
3   typedef int DataType;
4   struct Node
5   {
6       DataType   data;
7       struct Node * next;
8   };
9   typedef struct Node * PNode;
10  typedef struct Node * LinkStack;
11  //函数功能:创建空链栈
12  //输入参数: 无
13  //返回值: 空的链栈
```

```
LinkStack SetNullStack_Link();
//函数功能:判断一个链栈是否为空
//输入参数 top: 链栈栈顶指针
//返回值: 空栈返回 1,否则返回 0
int IsNullStack_link(LinkStack top);
//函数功能:进栈
//输入参数 top: 链栈栈顶指针
//输入参数 x: 进栈元素
//返回值: 无
void Push_link(LinkStack top, DataType x);
//函数功能:出栈
//输入参数 top: 链栈栈顶指针
//返回值: 无
void Pop_link(LinkStack top);
//函数功能:求栈顶元素的值
//输入参数 top: 链栈栈顶指针
//返回值: 栈顶元素的值
DataType Top_link(LinkStack top);
#endif
```

(3) LinkStack.c。

```
#include <stdio.h>
#include <stdlib.h>
#include "LinkStack.h"
LinkStack SetNullStack_Link()                    //创建带有头结点的空链栈
{
   LinkStack top = (LinkStack)malloc(sizeof(struct Node));
   if (top!= NULL) top->next = NULL;
   else printf("Alloc failure");
   return top;                                   //返回栈顶指针
}
int IsNullStack_link(LinkStack top)              //判断一个链栈是否为空
{
    if (top->next == NULL)
        return 1;
    else
        return 0;
}
void Push_link(LinkStack top, DataType x)        //进栈
{
    PNode p;
    p = (PNode)malloc(sizeof(struct Node));
    if (p == NULL)
        printf("Alloc failure");
    else
    {
        p->data = x;
        p->next = top->next;
        top->next = p;
    }
```

```
}
void Pop_link(LinkStack top)                    //删除栈顶元素
{
    PNode p;
    if (top->next == NULL)
        printf("it is empty stack!");
    else
    {
        p = top->next;
        top->next = p->next;
        free(p);
    }
}
DataType Top_link(LinkStack top)                //求栈顶元素的值
{
    if (top->next == NULL)
    {
        printf("It is empty stack!");
        return 0;
    }
    else
        return top->next->data;
}
```

(4) main.c。

```
//写出主程序,对栈的所有基本算法进行测试
//要求输入一个序列,通过栈实现序列的反转
#include <stdlib.h>
#include <stdio.h>
#include "LinkStack.h"
int main(void)
{
    LinkStack p = SetNullStack_Link(5);         //创建一个空栈
    int data;
    printf("请输入进栈的元素,以 0 结束:");
    scanf_s("%d", &data);
    while(data!= 0)
    {
        Push_link(p, data);                     //进栈
        scanf_s("%d", &data);
    }
    printf("出栈元素的顺序是:");
    while (!IsNullStack_link(p))                //是否为空栈
    {
        printf("%d ", Top_link(p));             //取栈顶元素
        Pop_link(p);                            //出栈
    }
    printf("\n");
    return 0;
}
```

（5）测试用例和测试结果。测试用例和测试结果截图如图 3-4 所示。

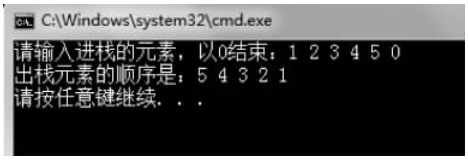

图 3-4　测试截图

四、扩展延伸

（1）运行顺序栈程序，输入为“1,3,5,7,9,11,0”，输出结果是什么？分析为什么。

（2）如果顺序栈的栈顶初始设置为 sstack-> top＝0，则需要修改顺序栈的哪些接口？并说明是否影响序列反转的实现。

（3）链栈的基本接口无清空栈的操作，说明清空栈的主要工作，并编程实现该操作。

3.2　初级实验 2

一、实验目的

使用链栈实现括号匹配算法和进制转换算法，包括十进制转换为八进制和十六进制。

二、实验内容

调用本章初级实验 1 给出的链栈接口，实现以下 3 个算法。

（1）括号匹配算法：要求用户输入带有圆括号的表达式，如果圆括号匹配，输出 1，否则输出 0。

（2）十进制转换为八进制：要求用户在键盘上输入一个十进制数，输出其对应的八进制数。

（3）十进制转换为十六进制：要求用户在键盘上输入一个十进制数，输出其对应的十六进制数。

三、参考代码

1. 本程序的文件结构

本程序的文件结构如图 3-5 所示，说明如下。

（1）LinkStack.h：链栈头文件，提供了链栈的数据结构类型定义和接口说明。

（2）BracketMatch.c：括号匹配文件，实现圆括号的匹配算法。

（3）Conversion.c：进制转换文件，实现十进制到八进制以及十进制到十六进制的转换。其中 BracketMatch.c 和 Conversion.c

图 3-5　程序的文件结构图

都使用了链栈，因此需要包含 LinkStack.h 链栈头文件。

（4）LinkStack.c：链栈接口的实现文件。

2. 括号匹配算法

（1）BracketMatch.c 包含了括号匹配函数和主函数。

```
#include <stdio.h>
#include <stdlib.h>
#include "LinkStack.h"
int BracketMatch( LinkStack mystack)                    //括号匹配算法
{
    int flag = 1;                                       //flag 为 1 则括号匹配
    char ch,temp;
    Push_link(mystack,'#');                             //栈底放 #
    printf("请输入要判断的表达式,用 # 号结束:");
    scanf_s("%c",&ch);
    while (ch!= '#')
    {
        if (ch == '(')                                  //左括号,压栈
            Push_link(mystack, ch);
        else
        {
            if (ch == ')')                              //右括号,出栈
            {
                temp = Top_link(mystack);
                if (temp == '(')
                    Pop_link(mystack);
                else
                {
                    flag = 0; break;
                }
            }//end if(ch == ')')
        }//end if(ch == '(')
        scanf_s("%c",&ch);
    }//end while(ch!= '#')
    if (!flag || Top_link(mystack)!= '#')
    {
        printf("no\n");
        return 0;
    }
    else
    {
        printf("yes\n");
        return 1;
    }
}
int main(void)
{
    LinkStack mystack = NULL;
    mystack = SetNullStack_Link();
```

```
45      BracketMatch(mystack);
46      return 0;
47  }
```

（2）测试用例和测试结果。测试用例和测试结果截图如图 3-6 所示。

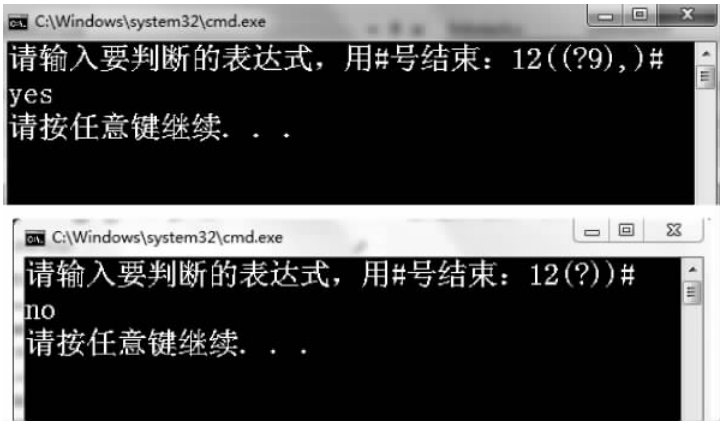

图 3-6　测试截图

3. 进制转换算法

（1）Conversion.c 包含了进制转换函数和主函数。

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "LinkStack.h"
4   void Conversion(LinkStack ps, int n)                    //实现八进制的转换
5   {
6       int temp;
7       while (n)
8       {
9           Push_link(ps, n % 8);
10          n/=8;
11      }
12      printf("转换为八进制后的结果:");
13      while (!IsNullStack_link(ps))
14      {
15          temp=Top_link(ps);
16          printf("%d", temp);
17          Pop_link(ps);
18      }
19  }
20  void Hexconversion(LinkStack ps, int n)                 //实现十六进制的转换
21  {
22      int temp;
23      while (n)
24      {
25          int temp=n%16;
```

```
        switch (temp)
        {
        case 10:temp = 'A'; break;
        case 11:temp = 'B'; break;
        case 12:temp = 'C'; break;
        case 13:temp = 'D'; break;
        case 14:temp = 'E'; break;
        case 15:temp = 'F'; break;
        }
        Push_link(ps, temp);
        n = n / 16;
    }
    printf("转换为十六进制后的结果:");
    while (!IsNullStack_link(ps))
    {
        temp = Top_link(ps);
        if (temp < 10) printf("%d", temp);
        else printf("%c", temp);
        Pop_link(ps);
    }
}
int main(void)
{
    LinkStack mystack = NULL;
    int n, m;
    mystack = SetNullStack_Link();
    printf("请输入需要转换为八进制的十进制数:");
    scanf_s("%d",&n);
    Conversion(mystack, n);
    printf("\n");
    printf("请输入需要转换为十六进制的十进制数:");
    scanf_s("%d", &m);
    Hexconversion(mystack, m);
    printf("\n");
    return 0;
}
```

(2) 测试用例和测试结果。测试用例和测试结果截图如图 3-7 所示。

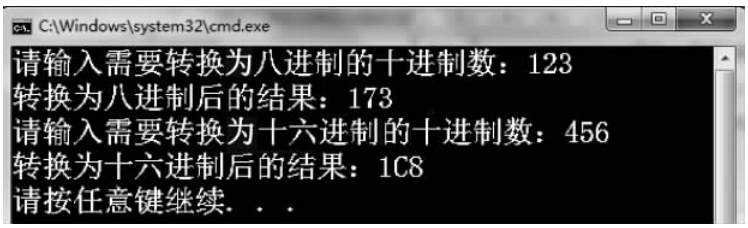

图 3-7 测试截图

四、扩展延伸

模拟 Windows 系统的计算器功能，实现一个简单易用的进制转换计算器，其功能主要是进行二进制、十进制、八进制和十六进制 4 种进制之间的相互转换。

3.3　初级实验 3

一、实验目的

掌握队列的基本操作，并应用栈和队列实现回文的判断。

二、实验内容

队列可以用链队列实现，也可以用顺序队列（循环）实现，要求至少具有以下基本接口，并使用队列实现回文算法，在主程序中进行测试。

（1）创建空队列；

（2）入队；

（3）出队；

（4）判断队列是否为空；

（5）取队头元素值；

（6）判断队列是否满（如果是顺序队列）；

（7）回文判断算法。

三、参考代码

1. 本程序的文件结构

本程序的文件结构如图 3-8 所示，说明如下。

（1）LinkStack.h：链栈头文件，同 3.1 节中的链栈定义。

（2）LinkStack.c：链栈接口的实现文件。

（3）LinkQueue.h：链队列头文件，提供了链队列的数据结构类型定义和接口说明。

（4）LinkQueue.c：链队列接口的实现文件。

（5）SeqQueue.h：循环队列头文件，提供了循环队列的数据结构类型定义和接口说明。

（6）SeqQueue.c：循环队列接口的实现文件。

（7）Polm.c：回文判断文件，实现回文判断算法。Polm.c 使用了链栈和循环队列，因此该文件中需要包含 LinkStack.h 和 SeqQueue.h。

图 3-8　程序的文件结构图

2. 循环队列

(1) SeqQueue.h。

```
#ifndef SEQQUEUE_H
#define SEQQUEUE_H
typedef int DataType;
 struct Queue
 {
    int Max;                                    //循环队列的最大容量
    int f;                                      //队头
    int r;                                      //队尾
    DataType * elem;                            //存放队列元素的一维数组首地址
 };
typedef struct Queue * SeqQueue;
//函数功能:创建空循环队列
//输入参数 m:循环队列的最大容量
//返回值: 空的循环队列
SeqQueue SetNullQueue_seq(int m);
//函数功能:判断一个循环队列是否为空
//输入参数 squeue: 循环队列指针
//返回值: 空队列返回 1,否则返回 0
int IsNullQueue_seq(SeqQueue squeue);
//函数功能:进队
//输入参数 squeue: 循环队列指针
//输入参数 x: 进队元素
//返回值: 无
void EnQueue_seq(SeqQueue squeue, DataType x);
//函数功能:出队
//输入参数 squeue: 循环队列指针
//返回值: 无
void DeQueue_seq(SeqQueue squeue);
//函数功能:求队头元素的值
//输入参数 squeue: 循环队列指针
//返回值: 队头元素的值
DataType FrontQueue_seq(SeqQueue squeue);
#endif
```

(2) SeqQueue.c。

```
#include <stdio.h>
#include <stdlib.h>
#include "SeqQueue.h"
SeqQueue SetNullQueue_seq(int m)                //创建空队列
{
    SeqQueue squeue;
    squeue = (SeqQueue)malloc(sizeof(struct Queue));
    if (squeue == NULL)
    {
        printf("Alloc failure\n");
        return NULL;
```

```
    }
    squeue->elem = (int *)malloc(sizeof(DataType) * m);
    if (squeue->elem != NULL)
    {
        squeue->Max = m;
        squeue->f = 0;
        squeue->r = 0;
        return squeue;
    }
else free(sequeue);
}
int IsNullQueue_seq(SeqQueue squeue)                    //判断队列是否为空
{
    return (squeue->f == squeue->r);
}
void EnQueue_seq(SeqQueue squeue, DataType x)           //入队
{
    if ((squeue->r + 1) % squeue->Max == squeue->f)     //是否满
        printf("It is FULL Queue!");
    else
    {
        squeue->elem[squeue->r] = x;
        squeue->r = (squeue->r + 1) % (squeue->Max);
    }
}
void DeQueue_seq(SeqQueue squeue)                       //出队
{
    if (IsNullQueue_seq(squeue))
        printf("It is empty queue!\n");
    else
        squeue->f = (squeue->f + 1) % (squeue->Max);
}
DataType FrontQueue_seq(SeqQueue squeue)                //求队头元素
{
    if (squeue->f == squeue->r)
        printf("It is empty queue!\n");
    else
        return(squeue->elem[squeue->f]);
}
```

3. 链队列

(1) LinkQueue.h。

```
#ifndef LINKQUEUE_H
#define LINKQUEUE_H
typedef int DataType;
struct Node
{
    DataType data;                                      //数据域
    struct Node *next;
```

```
};
typedef struct Node * PNode;
struct Queue
{  PNode  f;
   PNode  r;
};
typedef struct Queue * LinkQueue;
//函数功能:创建空链队列
//输入参数: 无
//返回值: 空的链队列
LinkQueue SetNullQueue_Link();
//函数功能:判断一个链队列是否为空
//输入参数 lqueue: 链队指针
//返回值: 空队列返回 1,否则返回 0
int IsNullQueue_Link(LinkQueue lqueue);
//函数功能:进队
//输入参数 lqueue: 链队指针
//输入参数 x: 进队元素
//返回值: 无
void EnQueue_link(LinkQueue lqueue, DataType x);
//函数功能:出队
//输入参数 lqueue: 链队指针
//返回值: 无
void DeQueue_link(LinkQueue lqueue);
//函数功能:求队头元素的值
//输入参数 lqueue: 链队指针
//返回值: 队头元素的值
DataType FrontQueue_link(LinkQueue lqueue);
#endif
```

(2) LinkQueue.c。

```
#include <stdio.h>
#include <stdlib.h>
#include "LinkQueue.h"
LinkQueue SetNullQueue_Link()                          //创建空队列
{
    LinkQueue lqueue = NULL;
    lqueue = (LinkQueue)malloc(sizeof(struct Queue));
    if (lqueue!= NULL)
    {
        lqueue->f = NULL;
        lqueue->r = NULL;
    }
    else
        printf("Alloc failure! \n");
    return lqueue;
 }
int IsNullQueue_Link(LinkQueue lqueue)                 //判断队列是否为空
{
    return (lqueue->f == NULL);
```

```
}
void EnQueue_link(LinkQueue lqueue, DataType x)      //入队
{
    PNode p;
    p = (PNode)malloc(sizeof(struct Node));
    if (p == NULL)
        printf("Alloc failure!");
    else
    {
            p -> data = x;
            p -> next = NULL;
            if (lqueue -> f == NULL)                 //空队列的特殊处理
            {
                lqueue -> f = p;
                lqueue -> r = p;
            }
            else
            {
                lqueue -> r -> next = p;
                lqueue -> r = p;
            }
        }
}
void DeQueue_link(LinkQueue lqueue)                  //出队
{
    struct Node * p;
    if (lqueue -> f == NULL)
        printf( "It is empty queue!\n ");
    else
    {
        p = lqueue -> f;
        lqueue -> f = lqueue -> f -> next;
        free(p);
    }
}
DataType FrontQueue_link(LinkQueue lqueue)           //求队头元素
{
    if (lqueue -> f == NULL)
    {
        printf("It is empty queue!\n");
        return 0;
    }
    else
        return (lqueue -> f -> data);
}
```

4. 回文的判断

(1) Polm. c。

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include "LinkStack.h"
#include "SeqQueue.h"
int main(void)
{
    char ch;
    int flag;
    LinkStack stack_pal = SetNullStack_Link();
    SeqQueue queue_pal = SetNullQueue_seq(10);
    printf("输入字符串以#结束\n");
    ch = getchar();
    while (ch != '#')
    {
        Push_link(stack_pal, ch);
        EnQueue_seq(queue_pal, ch);
        ch = getchar();
    }
    flag = 1;
    while ((!IsNullStack_link(stack_pal)) &&(!IsNullQueue_seq(queue_pal)))
    {
        if (Top_link(stack_pal)!= FrontQueue_seq(queue_pal))
        {
            flag = 0;
            break;
        }
        else {
            Pop_link(stack_pal);
            DeQueue_seq(queue_pal);
        }
    }
    if (flag)
        printf("this is palindromic\n");
    else
        printf("this is NOT palindromic\n");
     return 0;
}
```

(2) 测试用例和测试结果。测试用例和测试结果截图如图 3-9 所示。

图 3-9 测试截图

四、扩展延伸

1. 重复定义问题

在回文算法中，如果同时使用书中给出的链式栈和链式队列数据结构来实现，会报错。这是因为书中给出的链式栈和链式队列的头文件中都包含有以下相同的定义。请运行程序，给出错误截图，并思考对于这样的情况需要如何处理。

```
typedef int DataType;
struct Node{
DataType      data;
struct Node * next;
};
typedef struct Node * PNode;
```

2. 循环队列的另一种实现

循环队列可以解决顺序队列假溢出的问题，假设用一维数组 seqqu[m]表示一个循环队列，设置变量 front 和 count 分别表示循环队列的头指针和队列长度计数器，注意不设尾指针，头指针比队列实际第一个元素超前一个位置，要求实现以下接口并用主程序进行测试：创建空队列、判断队列是否为空、入队、出队和取队头元素。

3.4　初级实验 4

一、实验目的

掌握用循环链表来表示队列的方法。

二、实验内容

以循环链表来表示队列，只设置一个尾指针，指向队尾结点，不设置头指针，要求具有下列接口，并写出主程序测试各个接口。

(1) 创建空队列；

(2) 队列是否为空；

(3) 入队；

(4) 出队；

(5) 取队头元素。

三、参考代码

1. 本程序的文件结构

本程序的文件结构如图 3-10 所示，说明如下。

(1) ListQueue.h：链表队列头文件，提供了链表队列的

图 3-10　程序的文件结构图

数据结构类型定义和接口说明。

(2) ListQueue.c：链表队列接口的实现文件。

(3) main.c：主程序，对链表队列接口进行测试，因此需要包含 ListQueue.h。

2. 链表队列的实现

(1) ListQueue.h。

```
#ifndef LISTQUEUE_H
#define LISTQUEUE_H
typedef int DataType;
struct Node
{
    DataType data;
    struct Node   * next;
};
typedef struct Node * PNode;
struct ClinkQueue
{
    PNode r;
};
typedef struct ClinkQueue * LinkQueue;
//函数功能:创建空队列
//输入参数: 无
//返回值: 空的队列
LinkQueue createEmptyQueue_clink();
//函数功能:判断一个队列是否为空
//输入参数 pcqu: 队列
//返回值: 空队列返回 1,否则返回 0
int isEmpty_clink(LinkQueue pcqu);
//函数功能:进队
//输入参数 pcqu: 队列
//输入参数 x: 进队元素
//返回值: 无
void enQueue_clink(LinkQueue pcqu, DataType x);
//函数功能:出队
//输入参数 pcqu: 队列
//返回值: 无
void deQueue_clink(LinkQueue pcqu);
//函数功能:求队头元素的值
//输入参数 pcqu: 队列
//返回值: 队头元素的值
DataType frontQueue_clink(LinkQueue pcqu);
#endif
```

(2) ListQueue.c。

```
#include <stdio.h>
#include <stdlib.h>
#include "ListQueue.h"
LinkQueue createEmptyQueue_clink()                    //创建空队列
```

```
{
    LinkQueue pcqu = (LinkQueue)malloc(sizeof(struct ClinkQueue));
    pcqu->r = NULL;
    return pcqu;
}
int isEmpty_clink(LinkQueue pcqu)                    //判断队列是否为空
{
    if (pcqu->r == NULL)
    {
        printf("队列已经空.\n");
        return 1;
    }
    else return 0;
}
void enQueue_clink(LinkQueue pcqu, DataType x)       //入队
{
    PNode p;
    p = (PNode)malloc(sizeof(struct Node));
    p->data = x;
    if (pcqu->r == NULL)                             //第 1 个元素入队
    {
        pcqu->r = p;
        p->next = p;
        return;
    }
    p->next = pcqu->r->next;
    pcqu->r->next = p;
    pcqu->r = p;
}
void deQueue_clink(LinkQueue pcqu)                   //出队
{
    PNode p;
    if (pcqu->r == NULL)                             //空队列
    {
        printf("队列为空,无法出队\n");
        return;
    }
    if (pcqu->r->next == pcqu->r)                    //队列只有一个结点
    {
        free(pcqu->r);
        pcqu->r = NULL;
        return;
    }
    p = pcqu->r->next;
    pcqu->r->next = p->next;
    free(p);
}
DataType frontQueue_clink(LinkQueue pcqu)            //取队头元素的值
{
    if (pcqu->r == NULL)
    {
```

```
56          printf("Empty queue.\n");
57          return 0;
58      }
59      else
60          return (pcqu->r->next->data);
61  }
```

3. main.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "ListQueue.h"
4   int main(void)
5   {
6       LinkQueue pcqueue = createEmptyQueue_clink();    //创建空队列
7       int data;
8       printf("请输入进队的元素,以 0 结束:");
9       scanf_s("%d", &data);
10      while(data!= 0)
11      {
12          enQueue_clink(pcqueue, data);               //进队
13          scanf_s("%d", &data);
14      }
15      printf("出队元素的顺序是:");
16      while (!isEmpty_clink(pcqueue))
17      {
18          printf("%d ", frontQueue_clink(pcqueue)); //输出队头元素
19          deQueue_clink(pcqueue);                     //出队
20      }
21      printf("\n");
22      retrun 0;
23  }
```

4. 测试用例和测试结果

测试用例和测试结果截图如图 3-11 所示。

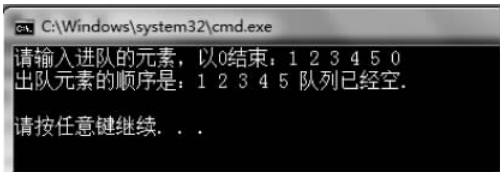

图 3-11 测试截图

四、扩展延伸

(1) 设计算法 reverse( ),实现将队列中元素的次序前后倒置。

算法思路：将队列中的元素依次取出并压入一个辅助栈,然后再依次取出栈中的元素

并入队原有队列。

（2）设计算法 QueuetoStack( )，实现从循环队列创建一个栈，使队头为栈顶、队尾为栈底，算法最后要求队列保持不变。

算法思路：将队列中的元素依次取出并压入一个辅助栈 Stack1，然后再依次取出 Stack1 中的元素入队原有队列并入栈 Stack2。

3.5　中级实验 1

一、实验目的

掌握采用深度优先策略和广度优先策略，并分别使用它们求解迷宫问题。

二、实验内容

本实验解决陷入迷宫的老鼠如何找到出口的问题。老鼠希望尝试所有的路径之后走出迷宫，如果它到达一个死胡同，将原路返回到上一个位置，尝试新的路径。在每个位置上老鼠可以向 8 个方向运动，即东、南、西、北、东南、东北、西南和西北。无论离出口多远，它总是按照这样的顺序尝试，当到达一个死胡同之后，老鼠将进行“回溯”。迷宫只有一个入口、一个出口，设计程序输出迷宫的一条通路，实验内容如下。

（1）设计迷宫的存储结构；

（2）采用回溯法设计求解通路的算法，利用栈实现回溯算法；

（3）采用广度优先策略，利用队列计算迷宫的一条最短通路。

三、参考代码

1. 文件结构和函数调用关系

本程序的文件结构如图 3-12 所示，说明如下。

图 3-12　程序的文件结构图

（1）node.h：结点定义文件，由于在 LinkStack.h 和 LinkQueue.h 中都使用，单独定义以避免重定义。

（2）LinkStack.h：链栈头文件，同 3.1 节中的链栈定义。

（3）LinkStack.c：链栈接口实现文件，同 3.1 节中的链栈定义。

（4）LinkQueue.h：链队列头文件，同 3.3 节中的链队列定义。

（5）LinkQueue.c：链队列接口实现文件，同 3.3 节中的链队列定义。

（6）mazeutil.h：迷宫头文件，定义了迷宫的接口。

（7）mazeutil.c：迷宫问题的辅助接口实现。

（8）MazeBFS.c：用广度优先策略解决迷宫问题的文件。其中使用了栈和队列，因此需要包含链栈头文件 LinkStack.h 和链队列头文件 LinkQueue.h。

(9) MazeDFS.c：用深度优先策略解决迷宫问题的文件。其中使用了栈和队列，因此需要包含链栈头文件 LinkStack.h 和链队列头文件 LinkQueue.h，使用了辅助函数，需要包含 mazeutil.h。

(10) main.c：主程序，测试迷宫算法，需要包含迷宫头文件 mazeutil.h。其中使用了辅助函数，需要包含 mazeutil.h。

2. 迷宫的实现

(1) node.h。

```
#ifndef NODE_H_
#define NODE_H_
typedef int DataType;
struct Node
{
    DataType data;
    struct Node * next;
};
typedef struct Node * PNode;
 #endif
```

(2) mazeutil.h。

```
#ifndef MAZE_H_
#define MAZE_H_
//迷宫的结构
typedef struct  MAZE_STRU
{
    int size;                                   //迷宫大小
    int ** data;                                //二维数组保存迷宫结构
}Maze;
//函数功能:初始化迷宫,分配空间,并将所有点置为 0
//输入参数 size: 迷宫大小
//返回值: 用邻接矩阵表示的图
Maze * InitMaze(int size);
//函数功能:读入迷宫结构,0 代表可以走的路,1 代表墙
//输入参数 maze: 迷宫结构
//返回值: 无
void ReadMaze(Maze * maze);
//函数功能:将迷宫的结构显示出来
//输入参数 maze: 迷宫结构
//返回值: 无
void WriteMaze(Maze * maze);
//函数功能: 广度优先搜索路径
//输入参数 maze: 迷宫结构
//输入参数 entryX: 迷宫入口点的 x 坐标
//输入参数 entryY: 迷宫入口点的 y 坐标
//输入参数 exitX: 迷宫出口点的 x 坐标
//输入参数 exitY 迷宫出口点的 y 坐标
//返回值: 没有路径返回 0, 有路径返回 1
```

```
27  int MazeBFS(int entryX, int entryY, int exitX, int exitY, Maze * maze);
28  //函数功能：深度优先搜索路径
29  //输入参数 maze：迷宫结构
30  //输入参数 entryX：迷宫入口点的 x 坐标
31  //输入参数 entryY：迷宫入口点的 y 坐标
32  //输入参数 exitX：迷宫出口点的 x 坐标
33  //输入参数 exitY：迷宫出口点的 y 坐标
34  //返回值：没有路径返回 0，有路径返回 1
35  int MazeDFS(int entryX, int entryY, int exitX, int exitY, Maze * maze);
36  #endif
```

(3) mazeutil.c。

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include "mazeutil.h"
4   Maze * InitMaze(int size)                    //初始化迷宫,分配空间,并将所有点置为 0
5   {
6       int i;
7       Maze * maze = (Maze * )malloc(sizeof(Maze));
8       maze -> size = size;                     //迷宫大小
9       //给迷宫分配空间
10      maze -> data = (int ** )malloc(sizeof(int * ) * maze -> size);
11      for (i = 0;i < maze -> size;i++)
12      {
13          maze -> data[i] = (int * )malloc(sizeof(int) * maze -> size);
14      }
15      return maze;
16  }
17  void ReadMaze(Maze * maze)                   //读入迷宫结构,0 代表可以走的路,1 代表墙
18  {
19      int i,j;
20      printf("请用矩阵的形式输入迷宫的结构:\n");
21      //读入迷宫的结构
22      for (i = 0;i < maze -> size;i++)
23      {
24          for(j = 0;j < maze -> size;j++)
25              scanf_s(" %d", &maze -> data[i][j]);
26      }
27  }
28  //将迷宫的结构显示出来
29  void WriteMaze(Maze * maze)
30  {
31      int i,j;
32      printf("迷宫结构如下:\n");
33      //输出迷宫的结构
34      for (i = 0;i < maze -> size;i++)
35      {
36          for(j = 0;j < maze -> size;j++)
37              printf(" %5d", maze -> data[i][j]);
38          printf("\n");
```

```
39      }
40  }
```

（4）MazeDFS.c。

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include "mazeutil.h"
4   #include "linkstack.h"                    //包含链栈头文件
5   //迷宫深度遍历算法
6   int MazeDFS(int entryX, int entryY, int exitX, int exitY, Maze * maze)
7   {
8       int direction[8][2] = { { 0, 1 }, { 1, 1 }, { 1, 0 }, { 1, -1 },{ 0, -1 }, { -1, -1 },
                            { -1, 0 }, { -1, 1 } };
9       //用于两个栈,分别保存路径中的点坐标(x,y)
10      LinkStack linkStackX = NULL;
11      LinkStack linkStackY = NULL;
12      int posX, posY;                       //临时变量,存放点坐标(x,y)
13      int preposX, preposY;
14      int **mark;                           //标记二维数组,标记哪些点走过,不再重复走
15      int i, j;                             //循环变量
16      int mov;                              //移动的方向
17      //给做标记的二维数组分配空间,并赋初值
18      mark = (int **)malloc(sizeof(int *) * maze->size);
19      for (i = 0; i < maze->size; i++)
20          mark[i] = (int *)malloc(sizeof(int) * maze->size);
21      //给所有元素设置初值
22      for (i = 0; i < maze->size; i++)
23      {
24          for (j = 0; j < maze->size; j++)
25              mark[i][j] = 0;
26      }
27      linkStackX = SetNullStack_Link();             //初始化栈
28      linkStackY = SetNullStack_Link();             //初始化栈
29      mark[entryX][entryY] = 1;                     //入口点设置标志位
30      Push_link(linkStackX, entryX);                //入口点入栈
31      Push_link(linkStackY, entryY);                //入口点入栈
32      //如果栈不为空且还没有找到迷宫出口点
33      while (!IsNullStack_link(linkStackX))
34      {
35          preposX = Top_link(linkStackX);
36          preposY = Top_link(linkStackY);
37          Pop_link(linkStackX);
38          Pop_link(linkStackY);
39          mov = 0;
40          while (mov < 8)
41          {
42              posX = preposX + direction[mov][0];
43              posY = preposY + direction[mov][1];
44              if (posX == exitX && posY == exitY)       //到达终点
45              {
```

```
46                  Push_link(linkStackX, preposX);     //出口点入栈
47                  Push_link(linkStackY, preposY);     //出口点入栈
48                  printf("\n深度搜索迷宫路径如下:\n");
49                  printf("%d %d\t", exitX, exitY); //打印入口点
50                  while (!IsNullStack_link(linkStackX))  //将路径逆序输出
51                  {
52                      posX = Top_link(linkStackX);    //取栈顶元素
53                      posY = Top_link(linkStackY);    //取栈顶元素
54                      Pop_link(linkStackX);           //出栈
55                      Pop_link(linkStackY);           //出栈
56                      printf("%d %d\t", posX, posY);  //输出栈顶元素
57                  }//end while(!Is NULL Stack_link(linkStackX))
58                  return 1;
59              }//end of(posX == exitX && posY == exitY)
60              //还有路可以走通
61              if (maze->data[posX][posY] == 0 && mark[posX][posY] == 0)
62              {
63                  mark[posX][posY] = 1;
64                  Push_link(linkStackX, preposX);     //入栈
65                  Push_link(linkStackY, preposY);     //入栈
66                  preposX = posX;
67                  preposY = posY;
68                  mov = 0;                        //已经往前走了,因此重新从 0 号方向开始搜索
69              }//end if (maze->data[posX][posY] == 0 && mark[posX][posY] == 0)
70              else
71                  mov++;                              //换个方向试试
72          }//end while (mov < 8)
73      }//end while (!IsNullStack_link(linkStackX))
74      return 0;
75  }
```

(5) MazeBFS.c。

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include "mazeutil.h"
4   #include "LinkQueue.h"                          //包含链队列头文件
5   //迷宫广度遍历算法
6   int MazeBFS(int entryX, int entryY, int exitX, int exitY, Maze* maze)
7   {
8       int direction[8][2] = { { 0, 1 }, { 1, 1 }, { 1, 0 }, { 1, -1 },{ 0, -1 }, { -1, -1 },
                                { -1, 0 }, { -1, 1 } };
9       //用于两个队列,分别保存等待扩展的点的坐标(x,y)
10      LinkQueue linkQueueX = NULL;
11      LinkQueue linkQueueY = NULL;
12      int posX, posY;                                 //临时变量,存放点坐标(x,y)
13      int preposX, preposY;
14      int **preposMarkX;                              //记录迷宫行走过程中的前驱 x 值
15      int **preposMarkY;                              //记录迷宫行走过程中的前驱 y 值
16      int **mark;                                 //标记二维数组,标记哪些点走过,不再重复走
17      int i, j, mov;
```

```
    //给存放前驱 x 值的数组分配空间
    preposMarkX = (int ** )malloc(sizeof(int * ) * maze -> size);
    for (i = 0; i < maze -> size; i++)
    {
        preposMarkX[i] = (int * )malloc(sizeof(int) * maze -> size);
    }
    //给存放前驱 y 值的数组分配空间
    preposMarkY = (int ** )malloc(sizeof(int * ) * maze -> size);
    for (i = 0; i < maze -> size; i++)
    {
        preposMarkY[i] = (int * )malloc(sizeof(int) * maze -> size);
    }
    //给做标记的二维数组分配空间,并赋初值
    mark = (int ** )malloc(sizeof(int * ) * maze -> size);
    for (i = 0; i < maze -> size; i++)
    {
        mark[i] = (int * )malloc(sizeof(int) * maze -> size);
    }
    for (i = 0; i < maze -> size; i++)                    //给所有元素设置初值
    {
        for (j = 0; j < maze -> size; j++)
        {
            preposMarkX[i][j] =- 1;
            preposMarkY[i][j] =- 1;
            mark[i][j] = 0;
        }
    }
    linkQueueX = SetNullQueue_Link();                     //创建空队列
    linkQueueY = SetNullQueue_Link();                     //创建空队列
    EnQueue_link(linkQueueX, entryX);                     //迷宫入口点入队
    EnQueue_link(linkQueueY, entryY);                     //迷宫入口点入队
    mark[entryX][entryY] = 1;                             //入口点设置标志位
    //如果队列不为空且还没有找到迷宫出口点
    while (!IsNullQueue_Link(linkQueueX))
    {
        preposX = FrontQueue_link(linkQueueX);            //取队头
        DeQueue_link(linkQueueX);                         //出队
        preposY = FrontQueue_link(linkQueueY);            //取队头
        DeQueue_link(linkQueueY);                         //出队
        //将与当前点相邻接且满足一定条件的点放入队列
        for (mov = 0; mov < 8; mov++)
        {
            posX = preposX + direction[mov][0];
            posY = preposY + direction[mov][1];
            if (posX == exitX && posY == exitY)           //到达出口点
            {
                preposMarkX[posX][posY] = preposX;
                preposMarkY[posX][posY] = preposY;
                printf("\n 广度搜索迷宫路径如下:\n%d %d\t", posX, posY);
                //将路径逆序输出
                while (!(posX == entryX && posY == entryY))
```

```
69                  {
70                      //继续往前寻找前驱
71                      preposX = preposMarkX[posX][posY];
72                      preposY = preposMarkY[posX][posY];
73                      posX = preposX;
74                      posY = preposY;
75                      printf(" % d  % d\t", posX, posY);
76                  }
77                  return 1;
78              }//end if (posX == exitX && posY == exitY)
79              //如果能走,且没有被扩展过
80              if (maze - > data[posX][posY] == 0 && mark[posX][posY] == 0)
81              {
82                  EnQueue_link(linkQueueX, posX);      //入队扩展
83                  EnQueue_link(linkQueueY, posY);
84                  mark[posX][posY] = 1;                //做标记
85                  preposMarkX[posX][posY] = preposX;   //记录前驱
86                  preposMarkY[posX][posY] = preposY;
87              }//end if (maze - > data[posX][posY] == 0 && mark[posX][posY] == 0)
88          }//end for (mov = 0; mov < 8; mov++)
89      }//end while (!IsNullQueue_Link(linkQueueX))
90      return 0;
91  }
```

3. main.c

```
1   # include < stdio.h >
2   #include "mazeutil.h"
3   int main(void)
4   {
5       Maze * maze;
6       int mazeSize;                                    //迷宫大小
7       int entryX,entryY,exitX,exitY;
8       int find = 0;
9       printf("请输入迷宫大小:");
10      scanf_s(" % d",&mazeSize);
11      entryX = 0; entryY = 0;
12      exitX = mazeSize - 1; exitY = exitX;
13      maze = InitMaze(mazeSize);
14      ReadMaze(maze);
15      printf("输入的迷宫结构如下:\n");
16      printf("请输入迷宫的入口点 x,y,出口点 x,y\n");
17      scanf_s(" % d % d % d % d", &entryX,&entryY,&exitX,&exitY);
18      //广度优先搜索
19      find = MazeBFS(entryX,entryY,exitX,exitY,maze);
20      //深度优先搜索
21      find = MazeDFS(entryX, entryY, exitX, exitY, maze);
22      if (!find)
23      {
24          printf("找不到路径!\n");
```

```
25        }
26        return 0;
27    }
```

4. 测试用例和测试结果

测试用例和测试结果截图如图 3-13 所示。

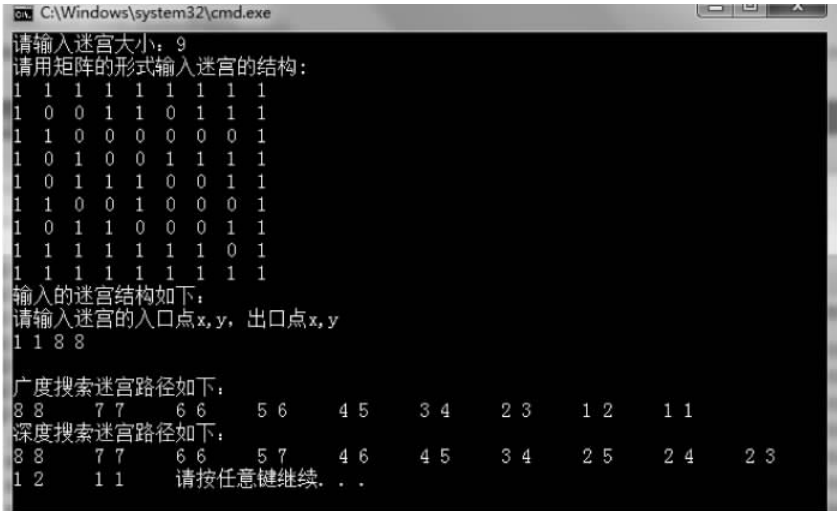

图 3-13 测试截图

四、扩展延伸

(1) 使用 STL 中的队列实现迷宫路径的广度优先搜索，使用 STL 中的栈实现迷宫路径的深度优先搜索。

(2) 模拟用户走迷宫，按一下键盘上的按键 a，则用户向前走一步。

(3) 在参考代码中，在进行迷宫的广度优先遍历时采用了两个二维数组来保存前驱结点的 x，y 值，请思考是否还有其他辅助结构能保存前驱的值。

(4) 对不同形态的迷宫进行测试，比较两种策略的优缺点。

下面给出使用 STL 中的栈的参考：

```
1    #include <stack>
2    #include <iostream>
3    using namespace std;
4    typedef stack<int> STACK_INT;
5    void main()
6    {
7        STACK_INT stack;
8        cout << "栈是否为空: "<<(stack.empty() ? "true" : "false") << endl;
9        cout << "入栈元素 1,3,5" << endl;
10       stack.push(1);                                    //入栈元素 1
11       stack.push(3);                                    //入栈元素 3
12       stack.push(5);                                    //入栈元素 5
```

```
13      if (!stack.empty())                                    //栈不为空,输出栈顶元素
14          cout << "当前栈顶元素是:" <<
15          stack.top() << endl;
16      stack.push(7);                                         //入栈元素 7
17      if (!stack.empty())
18          cout << "当前栈顶元素是: " <<                      //栈不为空,输出栈顶元素
19          stack.top() << endl;
20      while (!stack.empty())                          //栈不为空,输出栈顶元素,出栈,直到栈为空
21      {
22          const int& temp = stack.top();
23          cout << " 当前栈顶元素是:" << temp << endl;
24          stack.pop();
25      }
26  }
```

运行结果如图 3-14 所示。

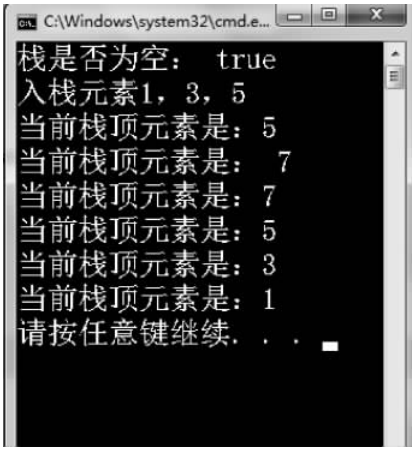

图 3-14　运行结果

3.6　中级实验 2

一、实验目的

掌握广度优先搜索策略，并用队列求解农夫过河问题。

二、实验内容

一个农夫带着一只狼、一只羊和一棵白菜，身处河的南岸，他要把这些东西全部运到北岸，遗憾的是他只有一只小船，小船只能容下他和一件物品。这里只能是农夫来撑船，同时因为狼吃羊、羊吃白菜，所以农夫不能留下羊和狼或者羊和白菜在河的一边，而自己离开，好在狼属于肉食动物，不吃白菜。农夫怎样才能把所有的东西安全运过河呢？实验内容如下。

(1) 设计物品位置的表示方法和安全判断算法；

(2) 可以设计队列的存储结构并实现队列的基本操作(建立空队列、判空、入队、出队、取队头元素),也可以使用 STL 中的队列进行代码的编写;

(3) 采用广度优先策略设计可行的过河算法;

(4) 输出要求:按照顺序输出一种可行的过河方案。

三、参考代码

1. 本程序的文件结构

本程序的文件结构如图 3-15 所示,说明如下。

(1) SeqQueue.h:顺序队列头文件,提供了顺序队列的数据结构类型定义和接口说明,同 3.3 节中顺序队列的定义。

(2) SeqQueue.c:顺序队列接口的实现文件,同 3.3 节中顺序队列的实现。

(3) FarmerRiver.h:农夫过河问题头文件,提供了农夫过河问题的接口说明。

(4) FarmerRiver.c:农夫过河问题的接口实现,由于用到了队列,需要包含 SeqQueue.h。

图 3-15 程序的文件结构图

(5) main.c:主程序,解决农夫过河问题,因此需要包含 FarmerRiver.h。

2. 农夫过河问题的实现

(1) FarmerRiver.h。

```
#ifndef FARMERRIVER_H
#define FARMERRIVER_H
//函数功能:判断当前状态下农夫是否在北岸
//输入参数 status:当前状态
//返回值:在北岸返回 1,否则返回 0
int FarmerOnRight(int status);
//函数功能:判断当前状态下狼是否在北岸
//输入参数 status:当前状态
//返回值:在北岸返回 1,否则返回 0
int WorfOnRight(int status);
//函数功能:判断当前状态下白菜是否在北岸
//输入参数 status:当前状态
//返回值:在北岸返回 1,否则返回 0
int CabbageOnRight(int status);
//函数功能:判断当前状态下羊是否在北岸
//输入参数 status:当前状态
//返回值:在北岸返回 1,否则返回 0
int GoatOnRight(int status);
//函数功能:判断当前状态是否安全
//输入参数 status:当前状态
//返回值:安全返回 1,否则返回 0
int IsSafe(int status);
//函数功能:农夫过河
```

```
//输入参数：无
//返回值：无
void FarmerRiver();
#endif
```

（2）FarmerRiver.c。

```
#include <stdio.h>
#include <stdlib.h>
#include "SeqQueue.h"
#include "FarmerRiver.h"
int FarmerOnRight(int status)                    //判断当前状态下农夫是否在北岸
{
    return (0!= (status & 0x08));
}
int WorfOnRight(int status)                      //判断当前状态下狼是否在北岸
{
    return (0!= (status & 0x04));
}
int CabbageOnRight(int status)                   //判断当前状态下白菜是否在北岸
{
    return (0!= (status & 0x02));
}
int GoatOnRight(int status)                      //判断当前状态下羊是否在北岸
{
    return (0!= (status & 0x01));
}
int IsSafe(int status)                           //判断当前状态是否安全
{
    if((GoatOnRight(status) == CabbageOnRight(status))&&
                (GoatOnRight(status)!= FarmerOnRight(status)))
        return (0);                              //羊吃白菜
    if((GoatOnRight(status) == WorfOnRight(status))&&
                (GoatOnRight(status)!= FarmerOnRight(status)))
        return 0;                                //狼吃羊
    return 1;                                    //其他状态是安全的
}
void FarmerRiver()                               //农夫过河算法
{
    int i, movers, nowstatus, newstatus;
    int status[16];                              //用于记录已考虑的状态路径
    SeqQueue moveTo;                             //用于记录可以安全到达的中间状态
    moveTo = SetNullQueue_seq(20);               //创建空队列
    EnQueue_seq(moveTo, 0x00);                   //初始状态时所有物品在北岸,初始状态入队
    for (i = 0; i < 16; i++)                     //数组 status 初始化为 - 1
        status[i] =- 1;
    status[0] = 0;
      //队列非空且没有到达结束状态
    while (!IsNullQueue_seq(moveTo) && (status[15] ==- 1))
```

```
    {
        nowstatus = FrontQueue_seq(moveTo);    //取队头状态为当前状态
        DeQueue_seq(moveTo);
       for (movers = 1; movers < = 8; movers << = 1)   //遍历 3 个要移动物品
        //考虑各种物品移动
        if ((0!= (nowstatus & 0x08)) == (0!= (nowstatus &movers)))
       //农夫与移动的物品在同一侧
        {
            newstatus = nowstatus ^ (0x08 | movers); //计算新状态
            //如果新状态是安全的且之前没有出现过
           if (IsSafe(newstatus) && (status[newstatus] ==- 1))
           {
                status[newstatus] = nowstatus;      //记录新状态
                EnQueue_seq(moveTo, newstatus);    //新状态入队
            }
        }
    }
    //输出经过的状态路径
    if (status[15]!=- 1)                            //到达最终状态
    {
        printf("The reverse path is : \n");
        for (nowstatus = 15; nowstatus > = 0; nowstatus = status[nowstatus])
        {
            printf("The nowstatus is : % d\n", nowstatus);
            if (nowstatus == 0)
              return;
        }
    }
    else
        printf("No solution.\n");                   //问题无解
}
```

3. main.c

```
#include < stdio.h >
#include < stdlib.h >
#include "FarmerRiver.h"
int main(void)
{
    FarmerRiver();
    return 0;
}
```

4. 测试用例和测试结果

测试用例和测试结果截图如图 3-16 所示。

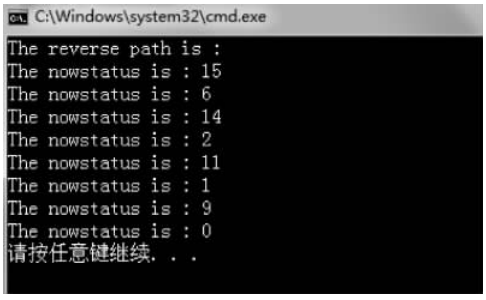

图 3-16　测试截图

四、扩展延伸

(1) 改进算法,要求输出所有的可行方案。

(2) 用深度优先策略解决农夫过河问题,输出一种可行的过河方案。

(3) 使用 STL 中的队列实现农夫过河问题。

下面给出使用 STL 中的队列的参考。

```
#include <queue>
#include <iostream>
using namespace std;
typedef queue<int> QUEUE_INT;
void main()
{
    QUEUE_INT testq;
    cout << "队列是否为空: " <<
       (testq.empty() ? "true" : "false") << endl;
    cout << "入队元素 1,3,5" << endl;
    testq.push(1);                                  //入队元素 1
    testq.push(3);                                  //入队元素 3
    testq.push(5);                                  //入队元素 5
    if (!testq.empty())                             //队列不为空,输出队头元素
        cout << "当前队头元素是:" <<
        testq.front() << endl;
    testq.push(7);                                  //入队元素 7
    testq.pop();                                    //出队
    if (!testq.empty())
        cout << "当前队头元素是:" <<                 //队列不为空,输出队头元素
        testq.front() << endl;
    while (!testq.empty())
                //队列不为空,输出队头元素,出队,直到队列为空
    {
        const int& temp = testq.front();
        cout << "当前队头元素是:" << temp << endl;
        testq.pop();
    }
}
```

3.7 高级实验 1

一、实验目的

掌握递归思想，将“聪明的学生”问题抽象为递归算法并加以实现。

二、实验内容

问题描述：一位逻辑学的教授有 3 个非常聪明、善于推理且精于心算的学生，他们是 A、B 和 C。一天教授给他们出了一道题，教授在每个人的脑门上贴了一个纸条，每个纸条上写了一个正整数，A、B 和 C 分别是 3、5、8，并且教授告诉学生某两个数的和等于第三个数，每个学生只能看见另外两个学生头上的正整数，但是看不见自己的。

教授问的顺序是 A—B—C—A……经过几次提问后，当教授再次问到 C 时，C 露出了得意的笑容，准确地报出了自己头上的数。

假设 A、B、C 脑门上贴的纸条上写的正整数分别是 3、5、8。

实验要求：已知 3 个人头上的正整数，得出教授在第几次提问时轮到回答问题的那个人猜出了自己头上的数(要求用递归程序解决)。

要求：

(1) 将“聪明的学生”问题抽象为递归算法。

(2) 给出测试用例“3,5,8”的输出结果。

三、参考代码

1. 参考提示

提示 1：总是贴着最大数的那个人猜出了自己头上的数。

提示 2：将“聪明的学生”问题抽象为递归算法，特别要注意递归程序的两个方面是递归出口和迭代步骤。

问题分析：依题可知，每个学生都能知道另外两个学生的数字，但不清楚自己的数字。这里以“1,2,3”作为例子来分析。A 只有两种情况，一种是(2－1)，另外一种是(2＋1)，但是 A 自己不能确定是哪一种情况，所以 A 猜不出来。再来看看 B，两种情况是(1＋3)和(3－1)，但是 B 仍然不能确定。最后是 C，两种情况是(1＋2)和(2－1)，但 C 是可以排除(2－1)这种情况的，因为如果 C 是(2－1)，那么 B 在看到 A 是 1、C 是 1 的情况下可以猜出来自己是 2，但是 B 没有猜出来，故 C 可以排除是(2－1)这种情况，因此 C 只能是(2＋1)，也就是 3。可以分析总结出最大的数字总会被最先猜出来，是因为只有最大的数字可以排除相减的情况，因为如果它是相减得到的，前面一定有人可以猜出来。

现在将问题抽象，即 A、B、C 3 个学生，他们头上的数字分别为 x1、x2、x3。从上述结论可知，最大的数总会被最先猜出来。不妨假设，B 是最先猜出来的学生，即 x2＝x1＋x3，而 B 能排除|x1－x3|这种可能性的依据有两个，一是 x1＝x3，那么 B 只能是 x1＋x3，因为 3 个都是正整数；另外一种依据是假设 x2＝|x1－x3|，那么在前面的提问中 A 或者 C 已经先猜

出来了，但他们没有猜出来，可以确定自己是两数之和，而非两数之差（可以结合上面的“1，2，3”看）。至于是 A 先猜出来还是 C 先猜出来，如果 A＞C，那么 A 先猜出来，否则 C 先猜出来。

那么问题可以转化为找出 x1、|x1－x3|、x3 中第一个猜出数字的人所用的次数。这个问题不断地重复成一个子问题，不断地重复这个过程，直到某个学生能够猜出这个数字为止。这样可以归结为如下递归函数：

$$\text{times}(i,j,t1,t2,t3)=\begin{cases} t3 & (i=j) \\ \text{times}(j,i-j,t2,t3,t1)+\text{step}(t1,t3) & (i>j) \\ \text{times}(i,j-i,t1,t3,t2)+\text{step}(t2,t3) & (i<j) \end{cases}$$

其中，t1 编号的人头上的数字为 i，t2 编号的人头上的数字为 j，t3 编号的人头上的数字为 $i+j$（即 t3 将最先猜出来自己的数字），教授提问的次数是 step(t1,t3)，表示按照 1—2—3 的顺序从 t1 问到 t3 所用的最少次数。

2. 代码实现

SmartStu.c：

```
#include <stdio.h>
#include <stdlib.h>
int step(int t1,int t2)                          //找出 t1 到 t2 的最小提问次数
{
    if(t2 > t1)
        return t2 - t1;
    else
        return t2 + 3 - t1;
}
//教授提问多少次时最大数 t3 能够正确回答
int times(int i,int j,int t1,int t2,int t3)
{
    int k; k = i - j;
    if (k == 0)
    {
        return t3;
    }
    if(k > 0)
    {
        return times(j,i - j,t2,t3,t1) + step(t1,t3);
    }
    if (k < 0)
    {
        return times(i,j - i,t1,t3,t2) + step(t2,t3);
    }
}
int main(void)
{
    int result;
    int a = 1,b = 2,c = 3;
```

```
31      int arr[3] = {3,5,8};
32      int index = 0;
33      int max_num =- 1,mid_num =- 1;              //保存最大以及第二大的数
34      int max_index =- 1,mid_index =- 1;          //保存两者的下标
35      for(;index < 3; index++)
36      {
37          if(max_num < arr[index] )
38          {
39              mid_num = max_num;
40              mid_index = max_index;
41              max_num = arr[index];
42              max_index = index;
43          }
44          if(mid_num < arr[index] && arr[index] != max_num)
45          {
46              mid_num = arr[index];
47              mid_index = index;
48          }
49      }
50      c = max_index + 1;
51      b = mid_index + 1;
52      //找出最小数的编号
53      if((c == 1 && b == 2)||(c == 2 && b == ) )
54          a = 3;
55      else if(( c == 2 && b == 3)||(c == 3 && b == 2))
56          a = 1;
57      else
58          a = 2;
59      //调用递归函数,第一个参数为最小数,第二个为中间数
60      //a 为最小值编号,b 为中间值编号,c 为最大值编号
61      result = times(arr[a - 1],arr[b - 1],a,b,c);
62      printf("result = %d\n",result);
63      return 0;
64  }
```

3. 测试用例和测试结果

测试用例和测试结果截图如图 3-17 所示。

图 3-17　测试截图

四、扩展延伸

请思考该问题的其他算法实现。

3.8 高级实验 2

一、实验目的

掌握双端队列 deque，并用双端队列解决具体问题。

双端队列的定义：同时具有队列和栈的性质，可以在头部和尾部插入和删除元素的数据结构。

二、实验内容

问题描述：给定一个长度为 n 的数列 $a_0, a_1, a_2, \cdots, a_{n-1}$ 和一个整数 k，求滑动最小值，即求 $b_i = \min\{a_i, a_{i+1}, \cdots, a_{i+k-1}\}$。

限制条件：$1 \leqslant k \leqslant 10^6$，$1 \leqslant n \leqslant 10^6$，$0 \leqslant a_i \leqslant 10^9$

实验要求：用户从键盘任意输入 n 和 k 的值，并输入 a_i，要求输出滑动最小值。

示例：

输入：$n=5, k=3, a=\{1,3,5,4,2\}$

输出：$b=\{1,3,2\}$

说明：使用 STL 中的双端队列容器，需要添加 #include <deque>。

三、参考代码

1. 算法设计

双端队列开始为空，队列中存放数组 a 的下标，然后通过维护双端队列得到最小值。

(1) 把 0～$k-1$ 依次加入队列。在加入 i 时，若双端队列末尾的值 j 满足 $a_j \geqslant a_i$，则不断从末尾取出，直到双端队列为空或者 $a_j < a_i$，之后在末尾加入 i。

(2) 在 $k-1$ 都加入双端队列后，查看双端队列头部的值 j，$b_0 = a_j$，如果 $j=0$，由于在之后的计算中不再需要，所以从头部删除。

(3) 继续计算 b_i，在双端队列的末尾加入 k，进入(1)重复执行。

2. 代码实现

DoubleQueue.cpp：

```
1   #include <iostream>
2   #include <deque>
3   using namespace std;
4   int main(void)
5   {
6       deque<int> d;
```

```
7       int n, k, a[100], b[100];
8       cout << "请输入 n 和 k: ";
9       cin >> n >> k;
10      cout << "请输入数组 a 的元素: ";
11      for (int i = 0; i < n; i++)
12      {
13          cin >> a[i];
14      }
15      int count = 0;
16      d.push_back(0);
17      for (int i = 1; i < n; i++)
18      {
19          while (!d.empty() && a[d.back()] >= a[i])
20          {
21              d.pop_back();
22          }
23          d.push_back(i);
24          if (i - k + 1 >= 0)
25          {
26              b[i - k + 1] = a[d.front()];
27              count++;
28          }
29          if (d.front() == i - k + 1)
30          {
31              d.pop_front();
32          }
33      }
34      for (int i = 0; i < count; ++i)
35      {
36          printf("%d ", b[i]);
37      }
38      return 0;
39  }
```

3. 测试用例和测试结果

测试用例和测试结果截图如图 3-18 所示。

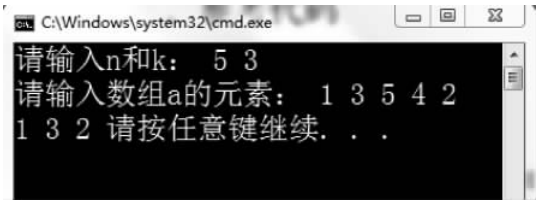

图 3-18 测试截图

四、扩展延伸

实现双端队列，并使用自定义的双端队列实现该实验内容。