哈尔滨工业大学计算机科学与技术学院

线性表及应用

数据结构与算法 Lab1

班　　级　1150310314

学　　号　1603005

姓　　名　黄炼

指导教师　李秀坤

实验地点　理学楼 208

实验时间

目录

一、 实验目的

通过将中缀表达式变为后缀表达式来加深对栈的理解，并且会用栈解决一些简单的实际问题。

二、 实验内容

1) 算术表达式求值

要求：

键盘可以重复输入中缀算术表达式，编程实现转换成后缀表达式输出，再对该后缀表达式求值计算输出结果。

2) 利用两个栈模拟队列的五个操作。

3) 利用两个队列模拟栈的五个操作。

三、 实验环境

- 格物楼 208 机房。
- Ubuntu 下 vim gcc gdb make
- C++

四、 实验过程

4.1 算术表达式求值

键盘可以重复输入中缀算术表达式，编程实现转换成后缀表达式输出，再对该后缀表达式求值计算输出结果。

算术表达式运算符号包括

+ - * \ %

包含括号

()

4.1.2 问题分析

说明：以下涉及到的运算符号和操作符是同义词。数值和操作数不区分视为同义词。

需要实现三个功能

A. 表达式的输入以及合法性检测。

B. 将中缀表达式转换为后缀表达式，并输出。

C. 计算后缀表达式结果，并输出。

1) 表达式的输入以及合法性检测。

输入的一行是一个表达式。

合法的表达式中包括基本的运算符号，括号和数值。合法的表达式是后续的计算的基础。非法表达式中可能包含任何的键盘符号。

其中，数值有整数（一位或者多位），浮点小数。数值有正负的区分，即有-可能是负数的符号，也可能是运算符号。

2) 将中缀表达式转换为后缀表达式，并输出。

中缀表达式运算顺序依据运算符号出现的顺序以及符号的优先度和括号指定的次序计算。

后缀表达式不包含括号，运算符放在两个运算对象的后面，所有的计算按运算符出现的顺序，严格从左向右进行（不再考虑运算符的优先规则）。

所以需要依据中缀表达式中各运算的优先顺序以及括号来调整符号在后缀表达式中出现的顺序。

3) 计算后缀表达式结果，并输出。

遇到一个运算符号，取出两个操作数做符号对应的计算，将计算结果保留到操作数序列中。重复这个过程直到求解出表达式。

4.1.3 设计思想

4.1.3.1 表达式的输入以及合法性检测

图 1 是对此部分内容概括

#插入图片# 表达式合法性检查

将一行输入作为一个表达式，即表达式是一个字符串。对输入的字符串的处理几位对表达式的处理。则一个合法的表达式字符串被限定为了以下符号集合的组成。

+ - * / % 1 2 3 4 5 6 7 8 9 0

为了允许输入的表达式可读性更好，允许录入空白符号。

凡是包含了不在上述集合中的字符的表达式，都是非法的（无效的）表达式。空白符号在处理时跳过。

现在依次的处理字符串的每个字符在表达式中的有效性，藉此判断表达式的有效性。依据问题分析将字符划分为以下 4 种情况讨论。

- 数值。是可以转化为一个操作数的序列，所以是一个或者多个字符的组成。

包括「0~9」，负号「-」，小数点「.」。为了便于分析。将0~9 又叫做数。

- 括号。一个字符

即左括号「（」和右括号「）」。

- 运算符。号一个字符

即 「+-*/%」

- -符号。一个字符。

「-」即使是数值也是「减」运算符号。所以再次的列出来。

每个字符的合法与否都依赖于前一个字符或者前一个数值。

1) 数值。一个完整操作数是在遇上一个非数值类型的字符时形成的。

一个数有效的情况：

- 没有前一个字符：即出现在表达式的开始。
- 前一个字符是一个数值类型的字符，运算符号或者左括号。

一个负号有效的情况

为了避免出现两个「-」出现的情况，负数跟在运算符号后面时要求用括号包括。所以一个负号有效的情况是：

- 没有前一个字符
- 前一个字符是左括号。

一个小数点有效的情况

- 前面应当是一个数且在已经形成的数值里没有小数点

2) 括号

左括号有效的情况

- 作为第一个符号
- 跟在一个左括号后面
- 跟在一个运算符号后面

右括号有效的情况

- 跟在一个数的后面
- 跟在一个有效的右括号的后面。

对于整个表达式还需要验证括号的匹配。

3) 运算符号

由于使用的都是双操作数的运算符号所以一个运算符号的有效情况应当是

- 前面是一个数
- 前面是一个右括号。

4) -符号

需要判断是一个减号还是一个负数的符号

- 前面是一个运算符号、空白、或者左括号，是负数的符号

- 前面是一个数，是减号，

注意：对于整个表达式还需要保证括号的匹配。表达式的结束应当是一个完整有效的数值（不仅是「-」或者「.」）或者右括号。

4.1.3.2 将中缀表达式转换为后缀表达式，并输出

对中缀表达式从左至右依次扫描

由于操作数的顺序保持不变，当遇到操作数时直接输出到后缀表达式中；

为调整运算顺序，设立一个栈用以保存操作符，扫描到操作符时，将操作符压入栈中。

进栈的原则是保持栈顶操作符的优先级要高于栈中其他操作符的优先级。优先级的表示是对于不同优先级的符号给予不同的整数值。具体值见实现部分。

否则，将栈顶操作符依次退栈并输出，直到满足要求为止。

遇到“（”进栈，当遇到“）”时，退栈输出直到“）”为止。

4.1.3.3 计算后缀表达式结果，并输出。

建立一个栈 S 。从左到右读表达式，如果读到操作数就将它压入栈 S 中，如果读到 n 元运算符(即需要参数个数为 n 的运算符)则取出由栈顶向下的 n 项按操作数运算，再将运算的结果代替原栈顶的 n 项，压入栈 S 中 。如果后缀表达式未读完，则重复上面过程，最后输出栈顶的数值则为结束。

4.1.4 具体实现过程

4.1.4.1 功能完成情况

- 表达式合法性检查
- 对括号，负数的处理
- 中缀表达式转换为后缀表达式
- 对后缀表达式求值

4.1.4.2 具体实现陈诉

数据结构

#插入图片# Expr 头文件

更多细节见附录文件：./Expr/include/expr.h

- 表达式的存储

使用单链表来存储表达式，每个节点是一个操作数或者一个操作符。

```C++ code
//   每一个节点
struct node{
   int tag; //  标记存储的数据是操作数或者操作符号。
            // 0  操作符。其他，浮点数,
            // 1 整数,
            // 2 浮点数
   union {
```

```
        char op;
        int n;
        double d;
    }data;
    struct Node * next;
};
Node * infix_ ;  // 中缀表达式
Node * postfix_ ;  // 后缀表达式
double value_ // 保存表达式计算结果
```

➢ 基本操作

```
// 显示中缀表达式
void Expr::ShowInfix(void) const ;

// 显示后缀表达式
void Expr::ShowPostfix(void) const ;

// 显示表达式计算结果
void Expr::ShowResult(void) const ;

// 表达式的合法性
bool Expr::is_valid(void) const;
```

关键实现

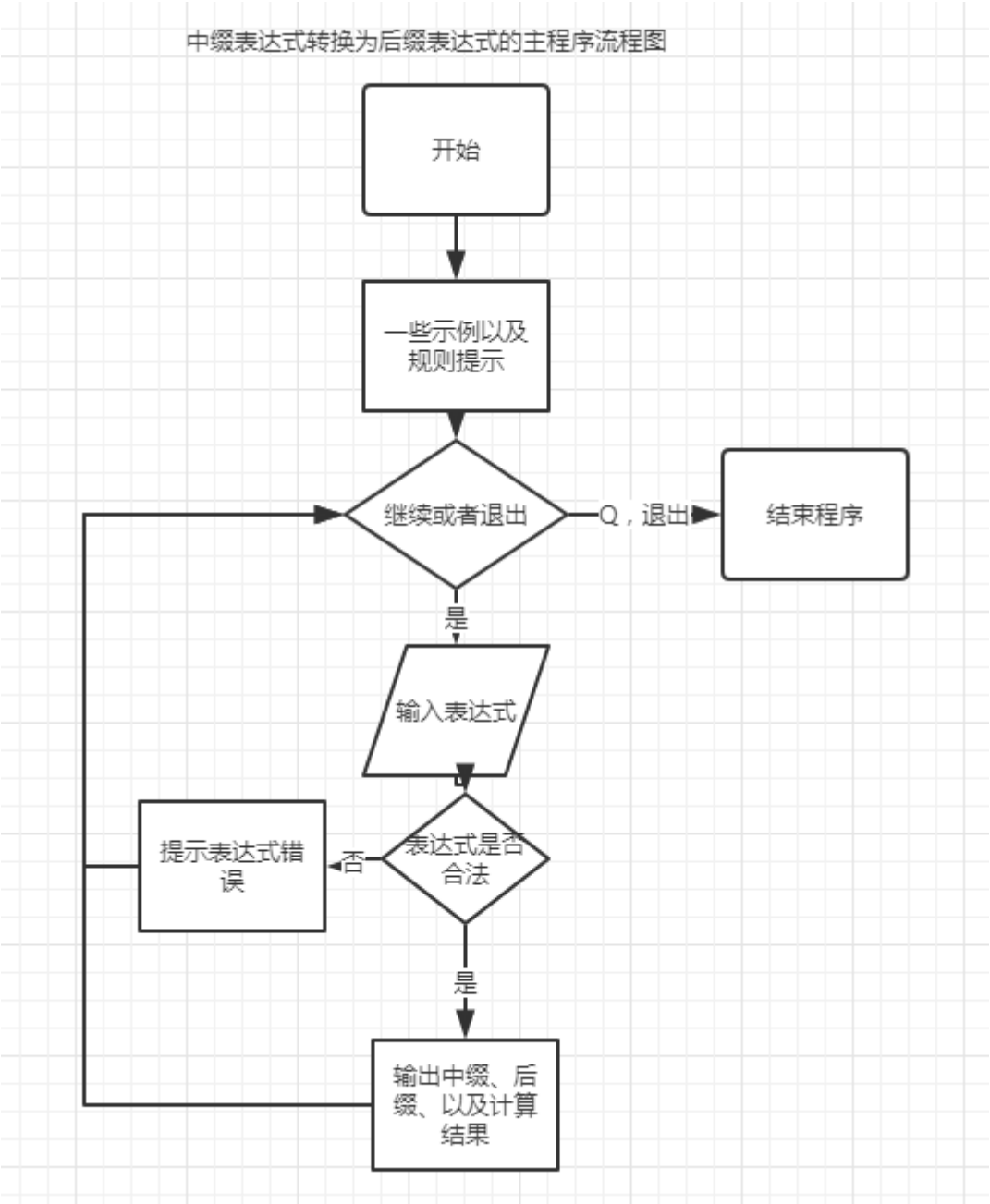
更多细节见附录文件：./Expr/src/expr.cpp

```
// 依据输入的字符串设置中缀表达式以及检测合法性。
// 不合法的表达式设置非法，并且中缀表达式值为 0
void Expr::set_infix_expr(const char * c_str_expr);

// 依据中缀表达式来设置后缀表达式
void Expr::set_postfix_expr(void);

// 依据后缀表达式来求表达式的值。
void Expr::set_value(void);
```

4.1.5 主程序流程及图示



4.1.6 测试用例及测试结果

测试方法见附录：如何验证测试

4.1.6.1 合法用例

包含全部运算符号，以及负数，小数，括号的测试

```
Input is :
-1.254-12*(342/34)+7%4
----------------------
Infix Expression:
-1.254 - 12 * ( 342 / 34 ) + 7 % 4
Postfix Expression:
-1.254 12 342 34 / * - 7 4 % +
Value = -118.96
----------------------
```

负数在表达式中间，减去一个负数

```
Input is :
-1.254 - (-1)/12*(342-99/34)+7%4
----------------------
Infix Expression:
-1.254 - ( -1 ) / 12 * ( 342 - 99 / 34 ) + 7 % 4
Postfix Expression:
-1.254 -1 12 / 342 99 34 / - * - 7 4 % +
Value = 30.0034
```

4.1.6.2 特殊情况：除零错误

a) 除以 输入的 0

```
Input is :
89 / 12/0
输入的表达式不合法!
```

b) 除以 运算过程中得到的 0

```
Input is :
34/(15-15)
输入的表达式不合法!
```

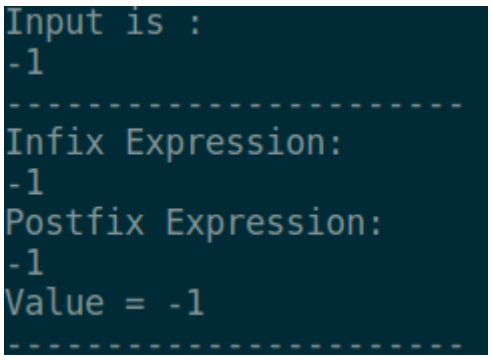
#插入图片# 除以 运算过程中得到的 0

4.1.6.3 边界条件

a) 过大的长度

```
Input is :
46557824233294735+1
输入的表达式不合法!
```

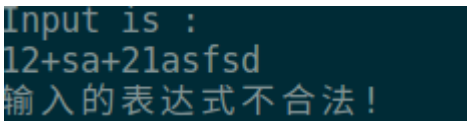
b) 1 长度



c) 0 长度

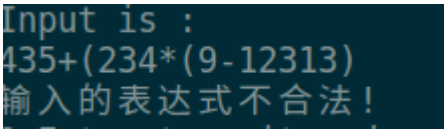
4.1.6.4 非法用例

a) 非法表达式符号



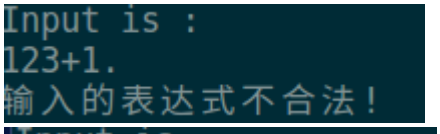
b) 括号

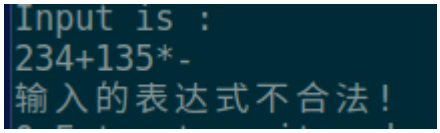
括号不匹配



包含空括号

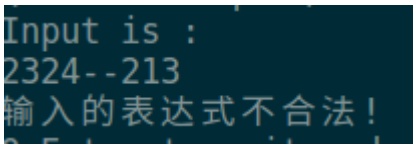
c) 以. – 结尾



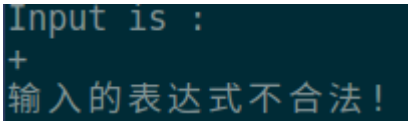


d) 运算符号重复

连续两个—



e) 只有运算符号



4.1.7 系统缺陷以及优点

无法解决数字长度 10 位以上的运算

仅支持五种基本运算

充分的考虑了鲁棒性

对负数和小数支持良好

4.2 利用两个栈模拟队列的五个操作。

利用两个栈模拟队列的五个操作。

已知栈的操作：

- Push()
- Pop()
- Top()
- MakeNull()
- IsEmpty()
- IsFull()

据此实现队列操作

- EnQueue()
- DeQueue()
- Front()
- MakeNull()
- IsEmpty()
- IsFull()

4.2.2 问题分析

栈的特性是后进先出，队列特性是先进先出。

核心问题是

- 如何由入栈转换为入队（栈顶-队尾）。
- 如何由出栈转换为出队（栈顶-队首）。

栈始终是对栈顶操作。一个栈无法完成对队列的模拟，再添加一个栈来缓存数据。假定栈不会溢出。

4.2.3 设计思想

4.2.3.1 思考 1 入栈等价于入队。、

即考虑入队就是直接将新元素压入的栈中，不进行栈的其他操作。

- 入队：假定栈低到栈顶的顺序即为队首到队尾。则
  - 数据压入栈 S1，
- 出队：
  - 两个栈都为空，出队失败。

- 否则，S1 的数据逐一取出并压入到 S2 空栈中。此时 S2 栈顶即为队首。栈 S2 进行一次出栈操作即为出队。再次将栈 S2 元素逐一取出压入原来的栈 S1 中，保持 S1 栈顶可入队。

4.2.3.2 思考 2 出栈等价于出队。

即考虑出队就是直接从栈中弹出一个元素，不进行栈的其他操作。

- 出队：假定栈顶到栈底的顺序即为队首到队尾的顺序。
  - 如果两个栈都空，出队失败
  - 否则，S1 栈顶即为队首，栈 S1 进行一次出栈操作即为出队。
- 入队：
  - 如果两个栈都空，将数据压入到其中一个栈 S2。
  - 否则，S1 栈底即为队尾。将栈 S1 数据逐一取出压入空栈 S2，然后将新数据压入 S1，即入队，然后将原来的数据压回 S1。即保持 S1 栈顶为可出队状态。

4.2.3.3 思考 3 更加优化的方案。入栈即入队，出栈即出队。

可以看出思考 1 和思考 2 都始终的维护 S1，S2 仅仅是作为缓存的作用。都存在需要将数据「倒」两次的情况：即将原来的数据倒入空栈 S2 中，操作好了，再将数据倒回原来的栈 S1 中。现在考虑能否不重复倒。其中思考 2 要比思考 1 稍微好一点，因为访问队首是比较容易的（取队首）。能否减少转移数据呢？

考虑在栈 S1 中，栈顶即为队尾，在栈 S2 中，S2 非空，栈顶即队首。则:

- 入队：数据直接压入栈 S1

- 出队：
  - 如果 S2 非空，出队即可
  - 否则，将 S1 的数据逐一取出压入 S2 中，S2 出栈即可。

4.2.4 具体实现过程

4.2.4.1 功能完成情况

全部 5 个功能

4.2.4.2 具体实现陈诉

数据结构

（更多细节见附录文件：(./stack_queue/stack_queue.h ）

存储结构

```
LinkStack<Type> stin_;
LinkStack<Type> stout_;
```

关键实现

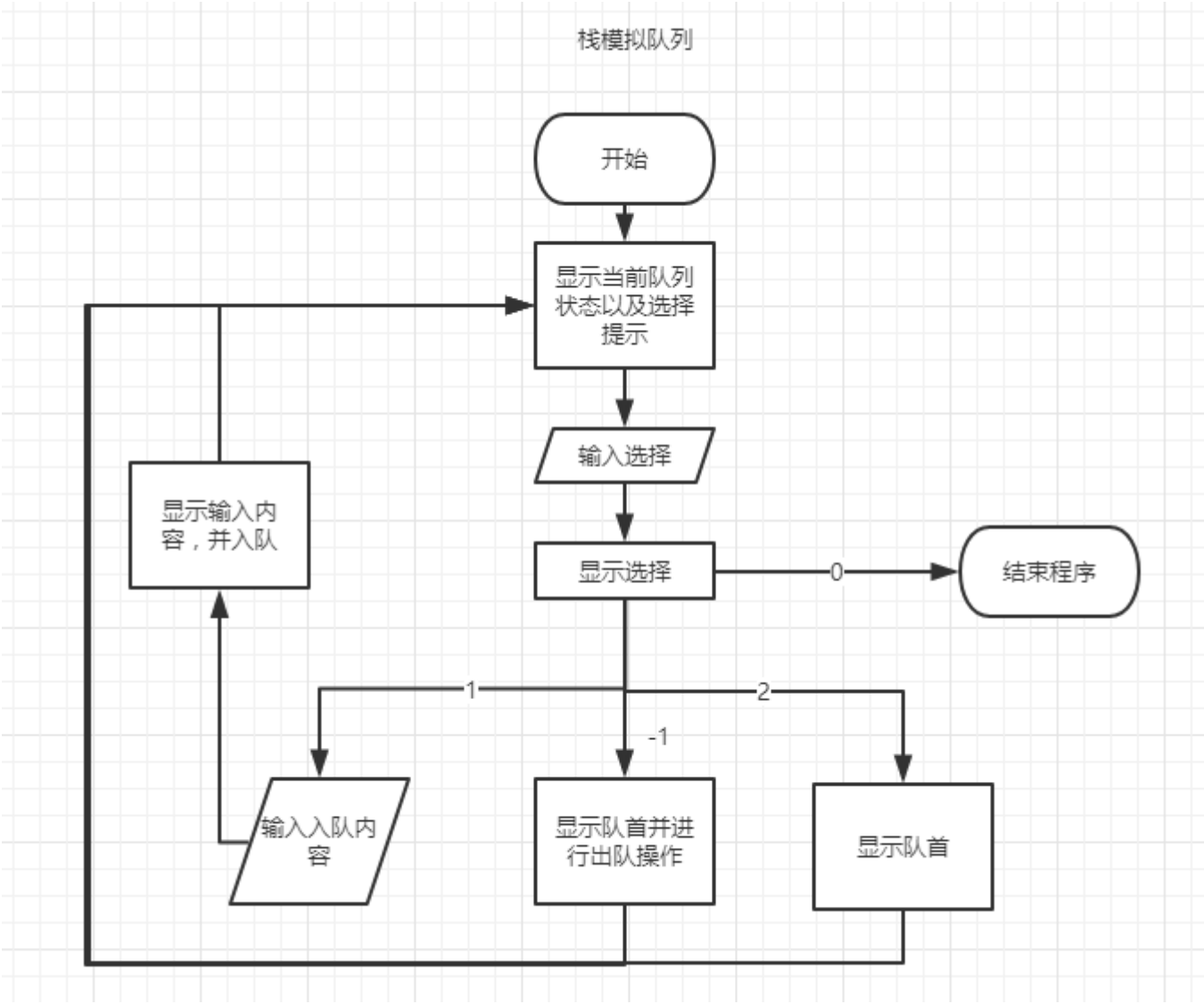
（更多细节见附录文件：(./stack_queue/stack_queue.h ）

EnQueue()

DeQueue()

Front()

4.2.5 主程序流程及图示



4.2.6 测试用例及测试结果

测试方法见附录：如何验证测试

入队

空队列

```
队首<----->队尾
NULL
        0.exit  1.EnQueue       -1.DeQueue      2.Front
choice #1
input n to enqueue:input is 23

23 EnQueue!
队首<----->队尾
23
```

非空队列

```
队首<----->队尾
23
        0.exit  1.EnQueue       -1.DeQueue      2.Front
choice #1
input n to enqueue:input is 323

323 EnQueue!
队首<----->队尾
23      323
```

出队

空队列

```
队首<----->队尾
NULL
        0.exit  1.EnQueue       -1.DeQueue      2.Front
choice #2
queue is empty.
```

非空队列

```
队首<----->队尾
23      323
        0.exit  1.EnQueue       -1.DeQueue      2.Front
choice #-1
23 DeQueue!
队首<----->队尾
323
```

访问队首

空队列

```
队首<----->队尾
NULL
        0.exit  1.EnQueue       -1.DeQueue      2.Front
choice #2
queue is empty.
```

非空队列

```
队首<----->队尾
5       6
        0.exit  1.EnQueue       -1.DeQueue      2.Front
choice #2
 front is 5
队首<----->队尾
```

4.2.7 系统缺陷以及优点

使用了自定义的队列结构，基础了其优缺点：在队列空时，返回值未定义。

使用了模板类，且封装良好，易复用

4.3 利用两个队列模拟栈的五个操作。

利用两个队列模拟栈的五个操作。

已知队列的

- EnQueue()
- DeQueue()
- Front()
- MakeNull()
- IsEmpty()
- IsFull()

据此实现栈操作

- Push()
- Pop()
- Top()
- MakeNull()
- IsEmpty()
- IsFull()

4.3.2 问题分析

栈的特性是后进先出，队列特性是先进先出。

核心问题是

- 如何由入队转换为入栈（队尾-栈顶）
- 如何由出队转换为出栈（队首-栈顶）

4.3.3 设计思想

4.3.3.1 思考 1 入队等价于入栈

入队即实现入栈。不进行其他队列操作。

Q1 的队尾即为栈顶。

入栈：直接入队到 q1

出栈：将非空队列中的元素逐一取出并入队到另外空队列中。非空队列的最后一个元素不入队到另外的队列中。

4.3.3.2思考 2 出队等价于出栈

出队操作即实现出队栈操作，不进行其他队列操作。

出栈：如果存在非空队列，非空队列中的队首到队尾序列即为栈顶到栈底部。

直接出队即可。

入栈：如果队列都为空，入队到 Q1。否则将元素入队到空队列中，然后将另外队列中的元素全部取出加入到该队列。

4.3.4 具体实现过程

在上述的两个方法，第二种在取栈顶时更加的有效，本实现是采用了第二种。

4.3.4.1 功能完成情况

全部 5 个功能

4.3.4.2具体实现陈诉

数据结构

（更多细节见附录文件：(./queue_stack / queue_stack.h)

关键实现

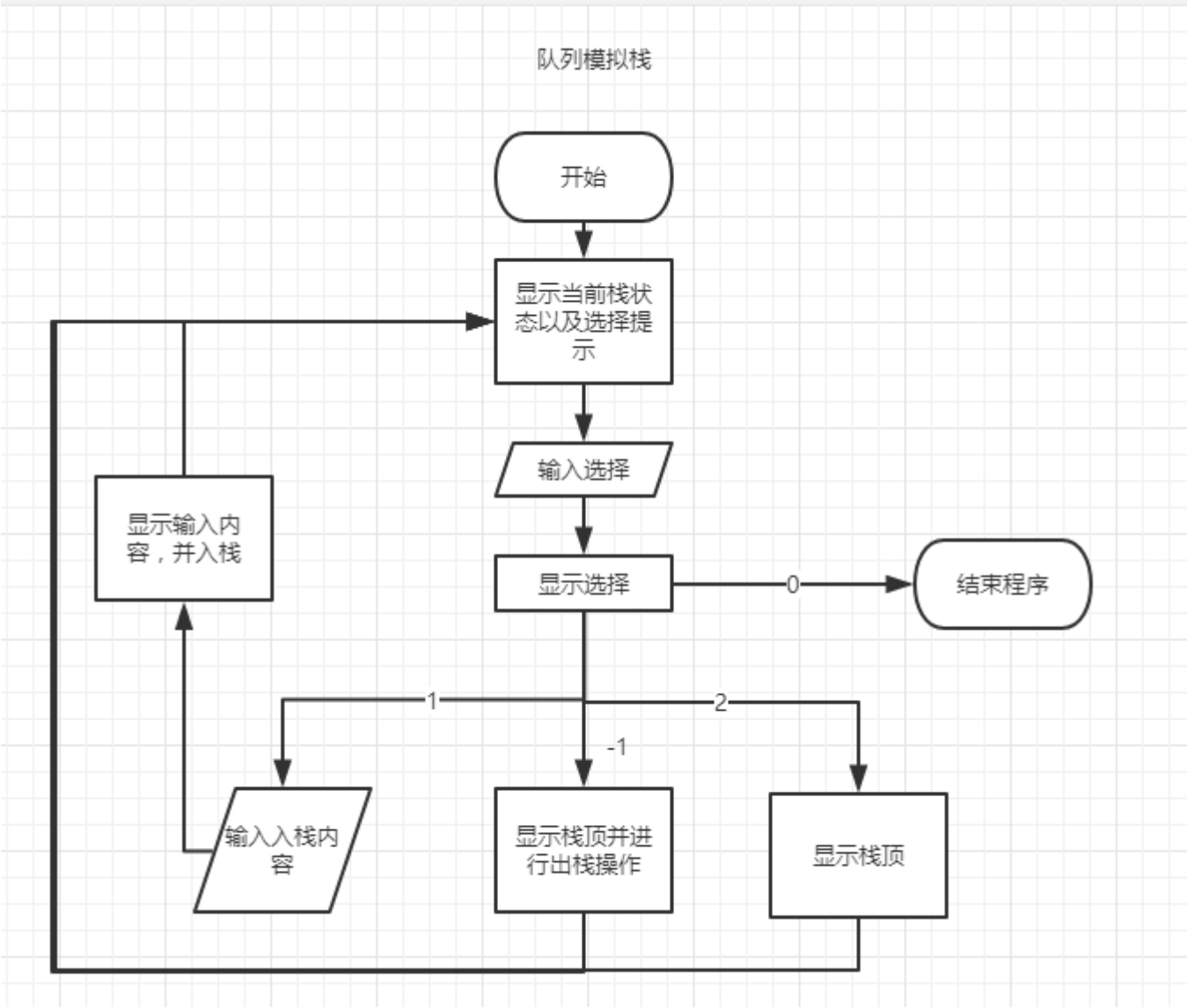
（更多细节见附录文件：(./queue_stack / queue_stack.h)

Push()

Pop()

Top()

4.3.5 主程序流程及图示



4.3.6 测试用例及测试结果

测试方法见附录：如何验证测试

Push()

空栈

```
栈顶<----->栈底
NULL
        0.exit  1.Push  -1.Pop  2.Top
choice is #1
input n to push:input is  23
栈顶<----->栈底
23
```

非空栈

```
栈顶<----->栈底
323     23
        0.exit  1.Push  -1.Pop  2.Top
choice is #1
input n to push:input is  12
栈顶<----->栈底
12      323     23
```

#插入图片#

Pop()

空栈

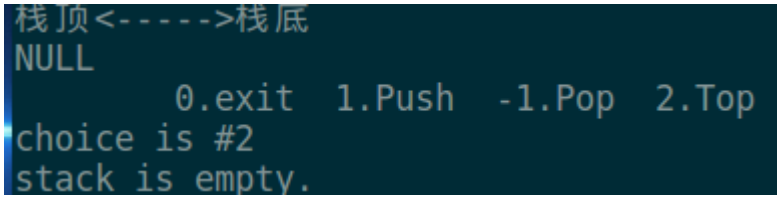
```
栈顶<----->栈底
NULL
        0.exit  1.Push  -1.Pop  2.Top
choice is #-1
empty!
```

非空栈

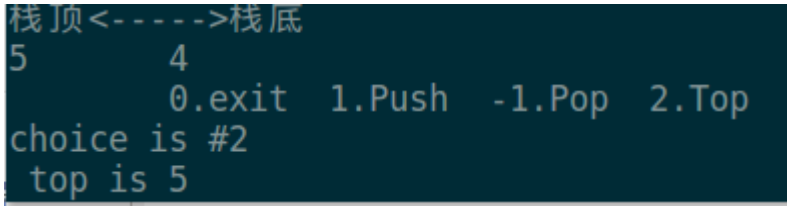
```
栈顶<----->栈底
12      323     23
        0.exit  1.Push  -1.Pop  2.Top
choice is #-1
Pop!
栈顶<----->栈底
323     23
        0.exit  1.Push  -1.Pop  2.Top
choice is #-1
Pop!
栈顶<----->栈底
23
```

Top()

空栈



非空栈



4.3.7 系统缺陷以及优点

使用了自定义的队列结构，基础了其优缺点：在队列空时，返回值未定义。

使用了模板类，且封装良好，易复用

五、 实验总结

1． 更加深刻的理解了栈和队列的特性。

a) 栈的特性是后进后出。也叫我们也可以叫它穿脱特性。具体的应用可以用于解决入库问题。比较容易用到的地方，比如单链表的逆序。

b) 队列的特性是先进先出。在需要次序的场景下可以使用：比如用于排队。

2． 由算术表达式的实验意识程序的鲁棒性问题，并且藉此学会了使用导图来分解问题，用二分法模拟实际过程，枚举得到各种问题分支和边界（表达式合法性检测）。

六、 附录

源代码

各个部分的实验源代码在本文档同目录下对应文件夹中。文件中分别对应为

算术表达式 Expr/expr.*

栈模拟队列 stack_queue/ stack_queue.h

队列模拟栈 queue_stack/queue_stack.h

文件结构说明

文件主要有两级，第一级是按照类别的分类，第二级是具体的代码或者说明文件。

除了源代码部分对应的文件夹，include/下存放是我自己实现的链式结构的栈和队列。

每个文件夹中都有以下四个文件。文件名和功能对应如下：

readme.md 当前目录的说明文件，或者是实现的描述

main.cpp 测试的主要源代码

makfile 用于编译和测试的 makefile 文件

test.txt 保存着上文中使用到的测试数据。测试结果已经在本文中出现，故另外不再保存。

和文件夹同名的文件是实验问题的源代码。

如何执行实验代码

直接执行（推荐）

切换到各个实验部分的目录下的/bin/目录下，直接点击对应可执行文件。类 Linux 系统下点击 *.out 文件。Windows 下点击执行*.exe。

可使用输入重定向输入测试数据。

Linux 下编译执行：

使用终端切换到源代码目录在 shell 中执行 make run 直接编译运行。执行 make test 执行测试（推荐）。make 生成可执行文件。

如果需要在 windows 下使用上述命令，需要系统安装 make 以及配置 g++路径