**Nankai University**

南 开 大 学

计算机学院

数据结构实验三

**实验三**实验报告

冯思程2112213

年级：2021级

专业：计算机科学与技术

指导教师：王玮

目录

**一、实验题目**

[（一）作业要求](#bookmark51" \o "Current Document)

[（二）问题描述](#bookmark60" \o "Current Document)

**[二、实验方案](#bookmark79" \o "Current Document)**

[（一）实验工具](#bookmark84" \o "Current Document)

[（二） 实验结果分析](#bookmark106" \o "Current Document)

[三](#bookmark147" \o "Current Document)**[、算法设计及分析](#bookmark147" \o "Current Document)**

[i代码展示与分析](#bookmark147" \o "Current Document)

[四](#bookmark313" \o "Current Document)**[、复杂度分析](#bookmark313" \o "Current Document)**

1. **[时间复杂度分析](#bookmark313" \o "Current Document)**
2. **[空间复杂度分析](#bookmark313" \o "Current Document)**

[五](#bookmark338" \o "Current Document)**[、实验总结](#bookmark338" \o "Current Document)**

—' 实验题目

（一） 作业要求

使用数组实现栈的出栈、入栈、获取栈顶元素、判断空栈四个方法。并使用该栈进行表达式运算。 表达式包含“+”、“-”、“\*”、“/”、“(”、“)”和数字的字符串，输出该表达式运算结果。

1.不能使用调用外部库实现栈。

（二） 问题描述

1.算法逻辑部分：

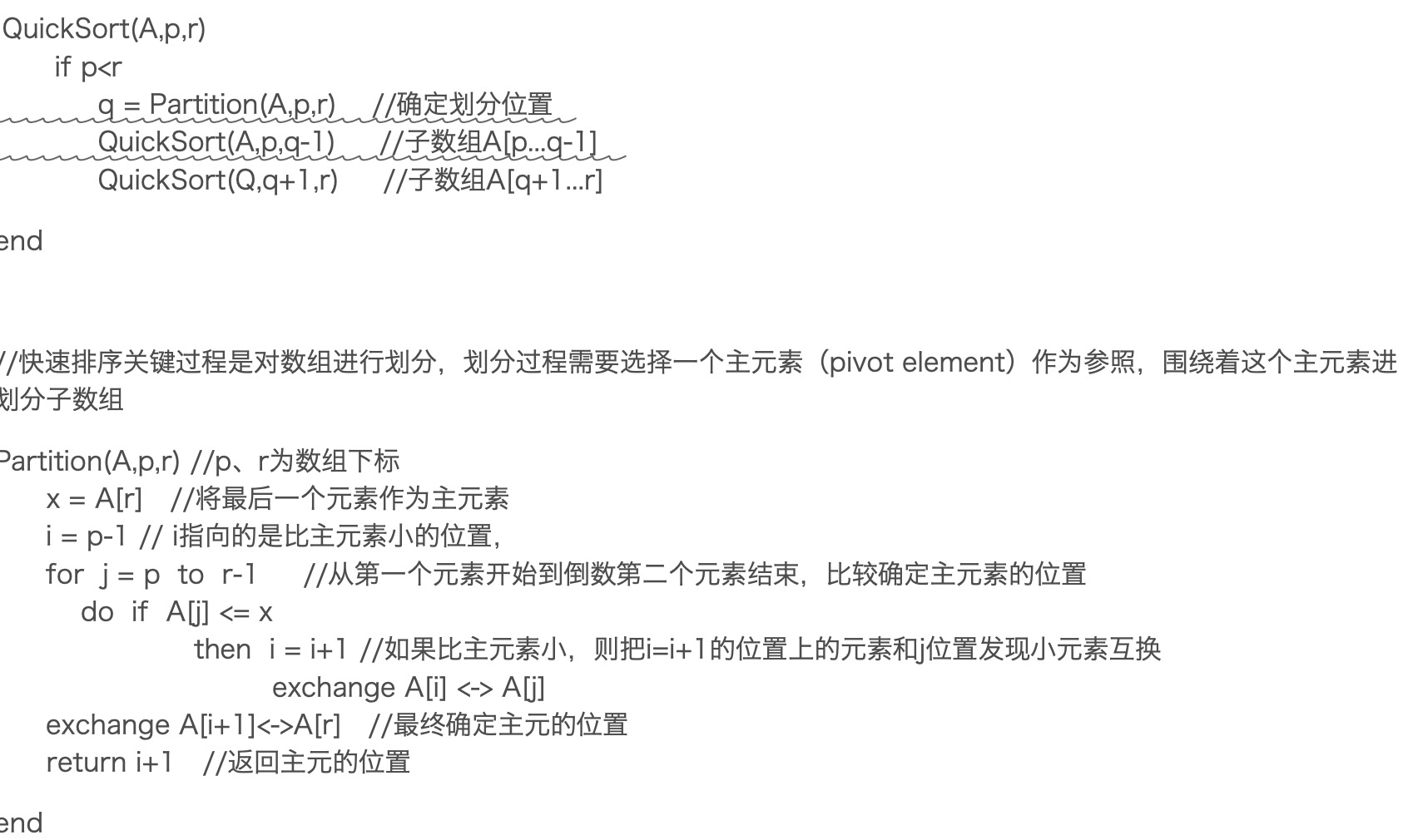
（0）优先级比较，在这里只考虑加减乘除的运算，乘除的优先级优于加减。所以一个返回2一个返回1。

（1）首先定义栈类，然后用instack初始化栈，动态数组进行实现栈。

（2）分别通过base和top指针写出出栈、入栈、判断空栈、获取栈顶元素、删除栈所占用的动态内存空间函数。

（3）然后是eval函数，是主函数算法中用到的计算函数，功能提出数据栈的最上面的两个元素和运算符栈的最上面一个元素，然后进行相应的运算。

（4）然后是主函数的部分，遍历输入表达式的每一位，首先是将数字字符转化成double类型的数字存入数据栈内，这里考虑到了表达式中的数据是多位数，写了一个while循环实现。然后是如果是运算符的话，分情况进行判断，如果是左括号或者此时运算符栈是空栈，则直接存入数据，如果是右括号，则运算eval函数，直到运算符栈的栈顶是左括号，然后将左括号出栈。继续向下遍历，然后是如果是加减乘除运算符，比较优先级，如果比栈顶元素优先级高，则直接进栈，否则就进行一次eval函数运算，然后再将运算符入栈。遍历之后，表达式已经被合理的存入两个栈内，然后循环执行eval运算直到运算符栈为空，输出此时的数据栈栈顶元素，然后删掉栈所占用的内存空间。



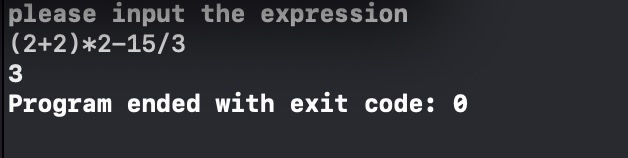
二、 实验方案

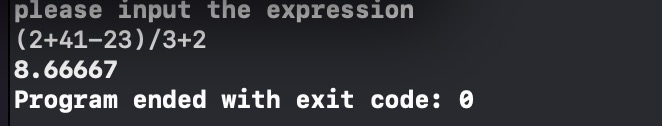
（一）实验工具

1.由于本人的电脑是mac下的macos系统，在本次实验用采用在xcode编译器下使用c++语言完成本次实验。

（二）实验结果分析步骤

1.在编写完程序后，输入多组数据进行输出测试，发现可以成功实现所需的功能。如下图：





1. 算法设计及分析

首先展示整个代码实现：

#include<iostream>

#include<cmath>

#include<cstring>

**using** **namespace** std;

**int** GetLevel(**char** ch)

{

**switch** (ch)

    {

**case** '+':

**case** '-':

**return** 1;

**case** '\*':

**case** '/':

**return** 2;

**default**:

**return** 0;

    };

}

**template**<**class** T>

**class** stack

{

**public**:

    T \*base;

    T \*top;

};

**template**<**class** T>

**void** instack(stack<T> &S){

    S.base=**new** T[100];

    S.top=S.base;

}

**template**<**class** T>

**void** push(stack<T> &S,T num){

    \*(S.top)=num;

    (S.top)++;

}

**template**<**class** T>

T getfirstelem(stack<T> &S){

**return** \*(S.top-1);

}

**template**<**class** T>

**bool** isemptystack(stack<T> &S){

**if**(S.base==S.top){

**return** **true**;

    }

**else**{

**return** **false**;

    }

}

**template**<**class** T>

**int** stacklength(stack<T> &S)

{

**return** **static\_cast**<**int**>(S.top - S.base);

}

**template**<**class** T>

**void** pop(stack<T> &S){

        (S.top)--;

    }

**void** eval(stack<**double**> &numberstack,stack<**char**> &charstack)

{

**double** a=0,b=0;

**char** p='p';

    a=getfirstelem(numberstack);

    pop(numberstack);

    b=getfirstelem(numberstack);

    pop(numberstack);

    p=getfirstelem(charstack);

    pop(charstack);

**double** r = 0;

**if**(p == '+')r=b+a;

**if**(p == '-')r=b-a;

**if**(p == '\*')r=b\*a;

**if**(p == '/')r=b/a;

    push(numberstack,r);

}

**template**<**class** T>

**void** destroystack(stack<T> &S)

{

**delete**[] S.base;

    S.base = S.top = **nullptr**;

}

**int** main(){

    stack<**double**> numberstack;

    stack<**char**> charstack;

    instack(numberstack);

    instack(charstack);

    string s;

    cout<<"please input the expression"<<endl;

    cin>>s;

**for**(**int** i=0;i<s.size();i++){

**if**(isdigit(s[i])){

**double** x=0;

**int** j=i;

**while**(j<s.size()&&isdigit(s[j])){

                x=x\*10+s[j]-'0';

                j++;

            }

            push(numberstack,x);

            i=j-1;

        }

**else**{

**if**(s[i]=='('||stacklength(charstack)==0){

                push(charstack,s[i]);

            }

**else** {

**if**(s[i]==')'){

**while**(getfirstelem(charstack)!='('){

                        eval(numberstack,charstack);

                    }

                    pop(charstack);

                }

**else**{

**if**(GetLevel(getfirstelem(charstack))<GetLevel(s[i])){

                        push(charstack,s[i]);

                    }

**else**{

                        eval(numberstack,charstack);

                        push(charstack,s[i]);

                    }

                }

            }

        }

    }

**while**(stacklength(charstack)!=0){

        eval(numberstack,charstack);

    }

    cout<<getfirstelem(numberstack)<<endl;

    destroystack(numberstack);

    destroystack(charstack);

**return** 0;

}

下面进行分部解释代码：

1. 运算符优先级判断函数：

**int** GetLevel(**char** ch)

{

**switch** (ch)

    {

**case** '+':

**case** '-':

**return** 1;//加减返回数值1

**case** '\*':

**case** '/':

**return** 2;//乘除返回数值2

**default**:

**return** 0;//其他情况返回0

    };

1. 栈和有关栈的操作的定义部分：

**template**<**class** T>

**class** stack

{

**public**:

    T \*base;

    T \*top;

};//定义栈顶和栈底指针

**template**<**class** T>

**void** instack(stack<T> &S){

    S.base=**new** T[100];

    S.top=S.base;

}//定义初始化函数，动态数组容纳的最大值为100个元素

**template**<**class** T>

**void** push(stack<T> &S,T num){

    \*(S.top)=num;

    (S.top)++;

}//入栈函数，每存入一个元素top指针向上移动一位

**template**<**class** T>

T getfirstelem(stack<T> &S){

**return** \*(S.top-1);

}//获取栈顶元素函数，只需要获取top指针的下面一个元素即可

**template**<**class** T>

**bool** isemptystack(stack<T> &S){

**if**(S.base==S.top){

**return** **true**;

    }

**else**{

**return** **false**;

    }

}//判断是否为空栈的函数，如果top和base指针重叠则证明是空栈

**template**<**class** T>

**int** stacklength(stack<T> &S)

{

**return** **static\_cast**<**int**>(S.top - S.base);

}//获取栈长度的函数，即top指针减base指针

**template**<**class** T>

**void** pop(stack<T> &S){

        (S.top)--;

    }//出栈函数，top向下移动一位

**template**<**class** T>

**void** destroystack(stack<T> &S)

{

**delete**[] S.base;

    S.base = S.top = **nullptr**;

}//删除栈的函数，因为之前分配动态内存，要在使用后及时删除

1. eval函数：

**void** eval(stack<**double**> &numberstack,stack<**char**> &charstack)

{

**double** a=0,b=0;

**char** p='p';

    a=getfirstelem(numberstack);

    pop(numberstack);

    b=getfirstelem(numberstack);

    pop(numberstack);//提出数据栈的最上面两个

    p=getfirstelem(charstack);

    pop(charstack);//提出运算符栈的最上面一个

**double** r = 0;

**if**(p == '+')r=b+a;

**if**(p == '-')r=b-a;

**if**(p == '\*')r=b\*a;

**if**(p == '/')r=b/a;//根据运算符进行运算

    push(numberstack,r);//将运算结果压入栈

}

1. 主函数部分：

**int** main(){

    stack<**double**> numberstack;

    stack<**char**> charstack;

    instack(numberstack);

    instack(charstack);//定义和初始化数据栈和运算符栈

    string s;

    cout<<"please input the expression"<<endl;

    cin>>s;//输入表达式

**for**(**int** i=0;i<s.size();i++){

**if**(isdigit(s[i])){

**double** x=0;

**int** j=i;

**while**(j<s.size()&&isdigit(s[j])){

                x=x\*10+s[j]-'0';

                j++;

            }//将表达式是数字字符的转为10进制数字存入数据栈中

            push(numberstack,x);

            i=j-1;

        }

**else**{

**if**(s[i]=='('||stacklength(charstack)==0){

                push(charstack,s[i]);//第一种情况，运算符是左括号或者运算符栈为空时，直接将运算符存入栈。

            }

**else** {

**if**(s[i]==')'){//如果是右括号

**while**(getfirstelem(charstack)!='('){

                        eval(numberstack,charstack);

                    }//运算到出现左括号

                    pop(charstack);//将左括号出栈

                }

**else**{//如果是加减乘除

**if**(GetLevel(getfirstelem(charstack))<GetLevel(s[i])){//判断优先级，如果高

                        push(charstack,s[i]);直接入栈

                    }

**else**{//其他情况

                        eval(numberstack,charstack);

                        push(charstack,s[i]);//先运算一遍栈顶的运算符再将运算符入栈

                    }

                }

            }

        }

    }

**while**(stacklength(charstack)!=0){

        eval(numberstack,charstack);

    }//循环运算直到运算符栈为空

    cout<<getfirstelem(numberstack)<<endl;//输出结果

    destroystack(numberstack);

    destroystack(charstack);//删除两个栈

**return** 0;

}

1. 复杂度分析

首先来说一下时间复杂度和空间复杂度的表示方法：算法的执行效率，粗略地讲，就是算法代码的执行时间。我们假设每行代码执行的时间都一样，都是1个单位时间，从而算出一段代码总的执行时间为多少个单位时间，然后将公式中的低阶、常量、系数这三个不左右增长趋势的部分忽略，只记录最大量级的表示法。例：T(n)=2n+2就可以记为T(n)=O(n)，T(n)=n2+2就可以记为T(n)=O(n2)。这就是大O时间复杂度表示法，大O时间复杂度实际上并不具体表示代码真正的执行时间，而是表示代码执行时间随数据规模增长的变化趋势，也叫做渐进时间复杂度，简称时间复杂度。

1. 时间复杂度分析：

首先是判断优先级的时间复杂度，可以看出是O（1），然后是对栈类的定义和有关栈操作的函数，出栈、入栈、判断空栈、获取栈顶元素、删除栈所占用的动态内存空间函数。函数中都没有循环和递归。时间复杂度都是O（1）。然后是eval计算函数，同理时间复杂度也是O（1）。然后是主函数部分，首先看存栈的整个遍历是一个循环，然后在内部的不同情况下，分别有while循环，一共是二重循环，综上时间复杂度是O（n^2），然后是运算部分，只有一个while循环运算到结束。综上，整个代码的时间复杂度是O（n^2）。

1. 空间复杂度分析：

创建了动态数组实现栈的功能，栈的初始化和入栈和出栈函数，都是线性的空间，空间复杂度是O（n），对于判断空栈和判断优先级函数，没有调用空间占用。获取栈顶元素，只创建了一个单位空间，空间复杂度是O（1），然后是eval函数，同理空间复杂度也是O（1），然后在主函数中，分别创建了数据栈和运算符栈，空间复杂度是O（n），在之后的计算中，只创建过几个单独的空间，这些的空间复杂度都是O（1），综上整个代码的空间复杂度是O（n）。

五、实验总结和程序改进

1.首先是程序流畅度的思考，在上次实验中，发现我在面对一些特殊情况时候，程序无法作出及时的反应，后来在助教学长的提醒下，进行了程序的优化。在这次实验中，我有意的关注了程序应对特殊情况的能力，例如如果表达式中的数字不是一位数的情况，我通过一个循环处理实现的目的。可以体会到大大提升了程序的流畅度和对代码的理解程度。

2.另外，在写代码的初期没有注意到数据栈应该用double类型，因为有除法的存在，还用int型会导致结果的偏差。而且在调试过程出现有括号的表达式不能正确输出结果的问题，最后通过了不断调试，最后成功改正了错误（错误是：在栈内有左括号的情况时，碰到右括号，运算到左括号出现后，没有将左括号进行出栈处理，导致运算错误。）在这个过程中，我也不断提高了自己的代码调试能力。