DATA STRUCTURES AND ALGORITHMS

CHALLENGE 1

ARITHMETIC EXPRESSION CALCULATION

LECTURERS

- PhD. Nguyễn Hải Minh
- M.Sc. Bùi Huy Thông
- M.Sc. Trần Thị Thảo Nhi

GROUP 039298 – 19CLC3

- Nguyễn Hữu Đạt 19127003
- Nguyễn Thanh Tình 19127292
- Lê Yến Nhi 19127498

Faculty of Information Technology
HCMC University of Science

November 2020

1

WORKING PROGRESS

Group Information

Student ID	Full Name	Email	Role
19127003	Nguyễn Hữu Đạt	19127003@student.hcmus.edu.vn	Leader, Dev
19127292	Nguyễn Thanh Tình	19127292@student.hcmus.edu.vn	Tester
19127498	Lê Yến Nhi	19127498@student.hcmus.edu.vn	Secretary

Project Information

Project name	ARITHMETIC EXPRESSION CALCULATION	
Tools and functions	Messenger	Communicate each other
	CodeBlocks	Coding
	Google Documents, Office 365	Write report and functions
	ZOOM Cloud Meetings	Meeting

Meeting

1. Overview

Group name: Group 039298

Members: Nguyễn Hữu Đạt, Nguyễn Thanh Tình, Lê Yến Nhi

Purpose of the meeting:

1. Discuss about the project.
2. Get the division of works.

Place: ZOOM Cloud Meetings, Messenger.

Date: Sunday 8/11/2020

Time: 9:30' – 11AM

Choose option: Floating – point (100% done)

2. Table of works

ID	NAME	DESCRIPTION	STATUS
19127003	Nguyễn Hữu Đạt	- Project Manager - Developer	DONE
19127292	Nguyễn Thanh Tình	- Write report - Summarize the project	DONE
19127498	Lê Yến Nhi	- Write report - Summarize the project	DONE

2

RESEARCH

2.1 Converting an infix expression to a prefix expression

While we use infix expressions in our daily lives, computers have trouble understanding this format because they need to keep in mind rules of operator precedence and also brackets. Prefix and Postfix expressions are easier for a computer to understand and evaluate (Geeksforgeeks).

Given two operands **a**, **b** and an operator **?**, the infix notation implies that **?** will be placed in between **a** and **b** (ex: **a ? b**), when the operator is placed before both operands **a** and **b** instead (ex: **? a b**) in prefix notation.

But, with an arithmetic infix expression (could be represented by a string), how could we convert it into prefix expression? To convert an infix to prefix, we could get the answer from the reversed of "reversed infix" in postfix:

- **Step 1:** Reverse the infix expression (ex: A+B*C becomes C*B+A). Note while reversing each '(' will become ')' and each ')' becomes '('.
- **Step 2:** Convert that expression into postfix. (ex: C B * A +)
- **Step 3:** Reverse the postfix expression. Hence in our example the prefix expression is + A * B C.

The detailed description about "Infix to Postfix" will be described in the next part of this document.

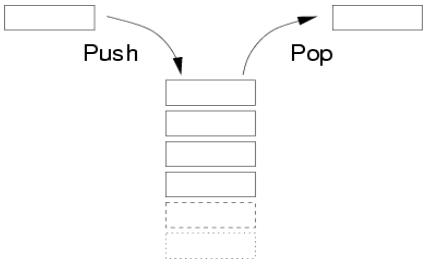
2.2 Converting an infix expression to a postfix expression

Given two operands **a**, **b** and an operator **?**, the infix notation implies that **?** will be placed in between a and b (ex: **a ? b**), when the operator is placed after both operands **a** and **b** instead (ex: **a b ?**) instead in postfix notation.

But, with an arithmetic infix expression (could be represented by a string), how could we convert it into postfix expression? The answer must be *stack programming technique*!

In computer science, **stack** is an **Abstracted Data Type (ADT)** which is based on **Last-in-First-out** (LIFO) principle. Some methods usually used:

- push(data): add a node to the top of the list.
- pop(): remove a node from the top of the list.
- top(): get the data of the top node of the list.
- empty(): check for emptiness of the list.



Stack (Wikipedia)

In this algorithm, we use a stack to save all operators ('+', '-', '*', '/', '^', '(', ')', '{', '}', '[', ']'). Also, the priority level of each operator is regulated by an function named getPriority(): The most priority is '^', next ones are '*' and '/', after those they're '+' and '-', the least priority are brackets.

- **Step 1:** Get an element T in the infix expression.
- **Step 2:** If T is a number, add it into the postfix answer string and go to **step 4**, else go to the next step.
- **Step 3:** If T is a operator
    - **Step 3.1**: If T is an open-bracket, push it to the stack.
    - **Step 3.2**: If T is an close-bracket, add to postfix answer string all operators in the stack until we met an open-bracket by pop().
    - **Step 3.3**: If T is '+', '-', '*', '/', '^', add to postfix answer string all operators in the stack which have no-greater priority-level than T by pop(). After that, push T into the stack.
- **Step 4:** If the expression still has element(s), back to **step 1**.
- **Step 5:** Take all operators left in the stack out, and add it to the postfix answer string.
- **Output:** Postfix answer string.

Example: Given an infix expression **(1 + 2) ^ 3**

- Read: **(**

Push it into the stack.

Stack	Postfix answer string
	

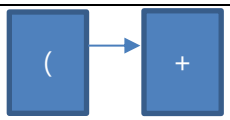
- Read: **1**

Add to postfix answer string.

Stack	Postfix answer string
	1

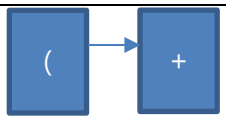
- Read: **+**

Push it into the stack.

Stack	Postfix answer string
	1

- Read: **2**

Add to postfix answer string.

Stack	Postfix answer string
	1 2

- Read: **)**

Add to postfix answer string all operators in the stack until we met an open-bracket by pop() method.

Stack	Postfix answer string
	1 2 +

- Read: **^**

Push it into the stack.

Stack	Postfix answer string
	1 2 +

- Read: **3**

Add to postfix answer string.

Stack	Postfix answer string
	1 2 + 3

- All elements in the expression are checked, take all operators left in the stack out, and add it to the postfix answer string.

Stack	Postfix answer string
	1 2 + 3 ^

- The output is postfix answer string: **1 2 + 3 ^**.

2.3 Converting a prefix expression to a postfix expression

Step 1: Read the Prefix expression in reverse order (from right to left)

Step 2: If the symbol is an operand, then push it into the Stack

Step 3: - If the symbol is an operator, then pop two operands from the Stack .

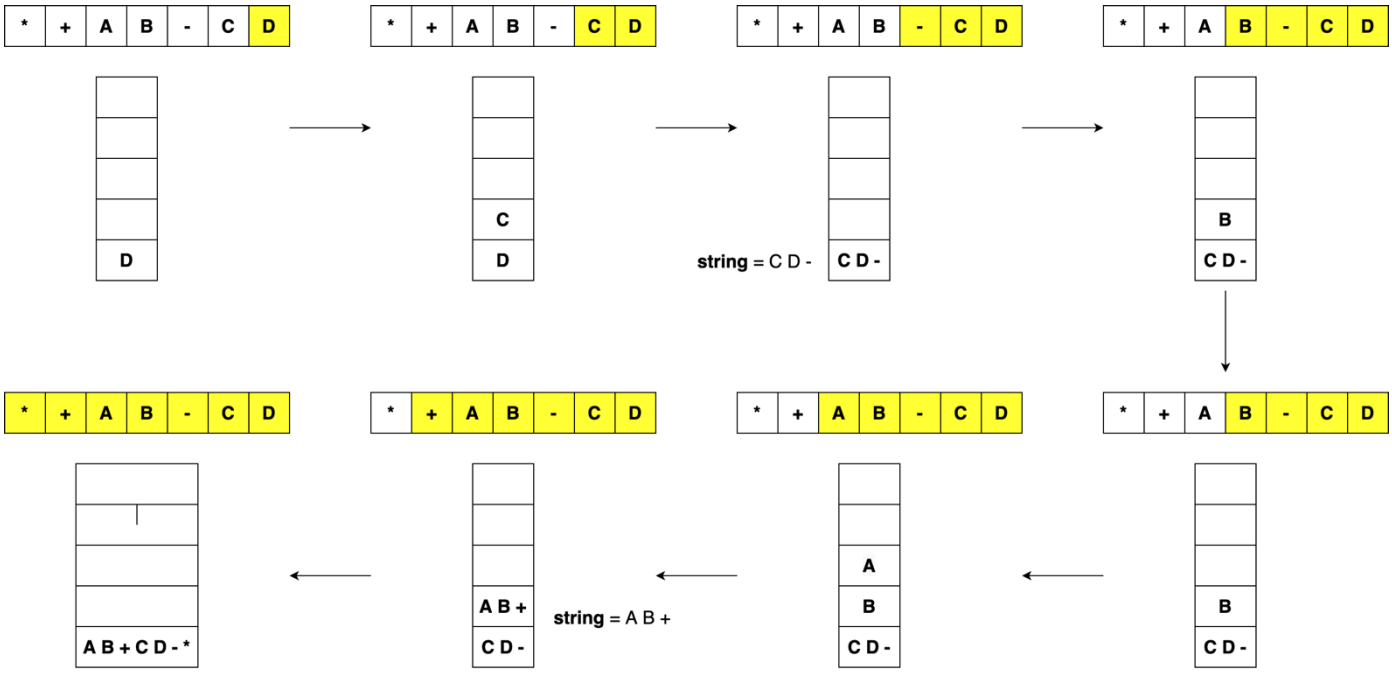
- Create a string by concatenating the two operands and the operator after them.

string = operand 1 + operand 2 + operator

- And push the resultant string back to Stack

Step 4: Repeat the above steps until end of Prefix expression.

Example: ***Prefix:*** * + A B – C D



→ ***Postfix:*** A B + C D - *

3

INFORMATION OF CODE FRAGMENTS

3.1 Data structures

Class template

std::**stack** (C++ 11)

```
template <class T, class Container = deque<T>> class stack;
```

3.2 Algorithms

```
// Main
for (i = 0 → Number Of Line To Read - 1 (given in agrv))
{
          string tmp = Get a string in the input file;
          if tmp is valid
          {
               if required action is '-t', print the postfix;
               if required action is '-c', print the value of
               calculating infix;
          }
          else print "E"
}


// Check for valid
```

- Check for errors of brackets:

+ Open-brackets is +1

+ Close-brackets is -1

+ Sum all of them must be always greater than 0 while browsing, and must be equal 0 at the end of iterator.

+ Wrong priority brackets // valid order: **{}** → **[]** → **()**

- Check for sequential elements in expression

+ Two adjacent numbers is wrong

+ Two adjacent operators is wrong

+ Before open-brackets are a number or a close-bracket is wrong

// Infix To Postfix Convertion Algorithm is described above (**2.2.** Converting an infix expression to a postfix expression)

// Calculate Infix – It really similars to Postfix Convertor Algorithm:

- **Step 1:** Get an element T in the infix expression.
- **Step 2:** If T is a number, push it into operand stack and go to **step 4**, else go to the next step.
- **Step 3:** If T is a operator
    - **Step 3.1**: If T is an open-bracket, push it to the operator stack.
    - **Step 3.2**: If T is an close-bracket, process all operators in the operator stack until we met an open-bracket by pop().
    - **Step 3.3**: If T is '+', '-', '*', '/', '^', process all operators in the operator stack which have no-greater priority-level than T by pop(). After that, push T into the operator stack.
- **Step 4:** If the expression still has element(s), back to **step 1**.
- **Step 5:** Process all operators left in the operator stack.
- **Output:** The answer is the top element of operand stack.

3.3 Functions

- Get input data into a vector from a file
```
void getInput(char* fileInp, int n, vector<string>& a);
```

- Check for validity of a expressionz
```
bool checkError(string e);
```

- Get priority of a operator
```
int getPriority(char c);
```

- Modity a string, make it easier to process
```
void modifyString(string& s);
```

- Convert infix expression to postfix expression
```
void infix_to_postfix(string& e);
```

- Calculate an infix expression
```
void calExpression(string e);
```

4

REFERENCES

- Geeksforgeeks – Prefix Postfix Conversion
  https://www.geeksforgeeks.org/prefix-postfix-conversion/
- Wikipedia: Stack
  https://vi.wikipedia.org/wiki/Ng%C4%83n_x%E1%BA%BFp
- Cplusplus.com – std::string::npos
  http://www.cplusplus.com/reference/string/string/npos/