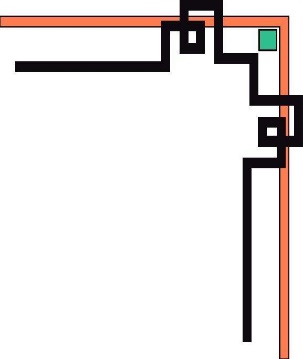
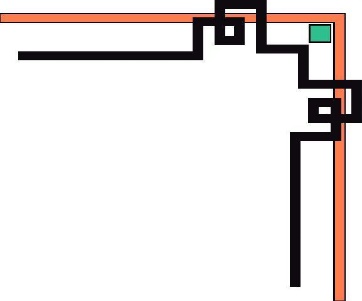
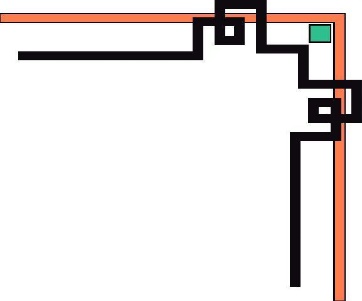
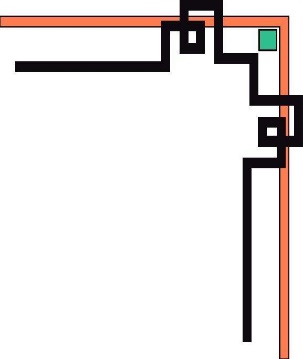
**MINISTRY OF EDUCATION AND TRAINING**



**HO CHI MINH UNIVERSITY OF SCIENCE**

**FACULTY OF INFORMANTION TECHNOLOGY \_ K20**

**Data Structures and Algorithms**

**PROJECT**



**ARITHMETIC EXPRESSION CALCULATION**

**Theory lecturer: Ngô Minh Nhựt**

**Practical lecturer: Bùi Huy Thông, Trần Thị Thảo Nhi**

**Student:**

**Trần Tiến Hoàng – 20127424**

**Lâm Kim Nhân – 20127579**

**Lê Võ Huỳnh Thanh - 20127072**

**Class: 20CLC09**

**Table of contents**

[WORK ASSIGNMENT 2](#_Toc75333008)

[CHAPTER 1: OVERVIEW 3](#_Toc75333009)

[I. Complete 3](#_Toc75333010)

[II. What we learn 3](#_Toc75333011)

[CHAPTER 2: DATA STRUCTURE 4](#_Toc75333012)

[1.1 Research 4](#_Toc75333013)

[ Converting an infix expression to a prefix expression. 4](#_Toc75333014)

[ Converting an infix expression to a postfix expression. 5](#_Toc75333015)

[ Converting a prefix expression to a postfix expression. 7](#_Toc75333016)

[1.2 Programming. 7](#_Toc75333017)

[REFERENCES 15](#_Toc75333018)

# WORK ASSIGNMENT

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Name | Job Description | Work progress |
| 20127424 | Trần Tiến Hoàng | Set up Stack programming.  Functional programming:   * string postfix(string s). * int Rank(char c). * float calculation(string s). | Accomplished |
| 20127579 | Lâm Kim Nhân | * Programming main function * Presenting research in 1.1 how to convert infix expressions to postfix. * Write project report. | Accomplished |
| 20127072 | Lê Võ Huỳnh Thanh | * Programming fuction bool CheckExpression(string s) * Presenting research 1.1 how to convert infix expressions to prefixes and prefix expressions to postfix * Write project report. | Accomplished |

# CHAPTER 1: OVERVIEW

## Complete

1.1 Research

* Converting an infix expression to a prefix expression.
* Converting an infix expression to a postfix expression.
* Converting a prefix expression to a postfix expression.

1.2 Coding

* Complete program to convert infix expression to prefix expression
* The result that appears is the execution number from the expression suffix

## What we learn

* Teamwork skill.
* Understand how to convert expressions from infix to prefixes, suffixes, or prefixes to suffixes.
* Know more new libraries (stack, iomanip, ..).

# CHAPTER 2: DATA STRUCTURE

## 1.1 Research

### Converting an infix expression to a prefix expression.

Algorithm:

Initialize an empty stack and i start from 0.

If an input expression is correct, reverse the infix expression, start from I, get the value of character at i location

* If it’s an operand: output.
* If it’s a close bracket, we put it to stack.
* If it’s an open bracket “(“, we will get the operator from stack to output as long as we meet the close bracket “)”. The open brackets will be got rid of stack but not appear in output.
* If it’s an operator: while on top of the stack, if it’s an operator and the priority of operator equal or higher than current one, we get it out of stack and put it in output. Then the current will be pushed in stack. While checking all the expressions in infix, if stack still has elements, we get the value of elements and then put it in output
* Lastly, we reverse expression again and that’s the result we want.
* E.g: convert infix expression ((A+B)\*C)/(D-E) to postfix expression:

Firstly, we reverse infix expression )E-D(/)C\*)B+A(( then convert to postfix expression.

|  |  |  |
| --- | --- | --- |
| **Infix** | **Stack** | **Prefix** |
| **)E-D(/)C\*)B+A((** | {Empty} | **{Empty}** |
| **E-D(/)C\*)B+A((** | ) | **{Empty}** |
| **-D(/)C\*)B+A((** | ) | **E** |
| **D(/)C\*)B+A((** | )- | **E** |
| **(/)C\*)B+A((** |  | **ED-** |
| **/)C\*)B+A((** |  | **ED-** |
| **)C\*)B+A((** | / | **ED-** |
| **C\*)B+A((** | /) | **ED-** |
| **\*)B+A((** | /) | **ED-C** |
| **)B+A((** | /)\* | **ED-C** |
| **B+A((** | /)\*) | **ED-C** |
| **+A((** | /)\*) | **ED-CB** |
| **A((** | /)\*)+ | **ED-CB** |
| **((** | /)\*)+ | **ED-CBA** |
| **(** | /)\* | **ED-CBA+** |
| **{Empty}** | / | **ED-CBA+\*** |
| **{Empty}** | {Empty} | **ED-CBA+\*/** |
| **{Empty}** | {Empty} | **/\*+ABC-DE** |

Reverse again and we will get the prefix expression.

### Converting an infix expression to a postfix expression.

Algorithm:

Initialize an empty stack and i start from 0.

If an input expression is correct, reverse the infix expression, start from I, get the value of character at i location:

* If it’s an operand: output.
* If it’s an open bracket “(“, put it to stack.
* If it’s a close bracket “)”, we get all the operator in stack and put it in output till we meet the open bracket “)”. The bracket will we push out of stack but not appear in output.
* If it’s operator: as long as the operator is still on top of the stack and the priority is equal or higher the current operator, we push that operator to output. Then put the current operator to stack. While checking all the infix expression, if stack still have element, we get all the value of character and put it in output.

E.g:

Convert infix expression A+(B-C)\*D+E/(F\*G) to postfix expression.

|  |  |  |
| --- | --- | --- |
| **Infix** | **Stack** | **Postfix** |
| **A+(B-C)\*D+E/(F\*G)** | {Empty} | **{Empty}** |
| **+(B-C)\*D+E/(F\*G)** | A | **{Empty}** |
| **(B-C)\*D+E/(F\*G)** | + | **A** |
| **B-C)\*D+E/(F\*G)** | +( | **A** |
| **-C)\*D+E/(F\*G)** | +( | **AB** |
| **C)\*D+E/(F\*G)** | +(- | **AB** |
| **)\*D+E/(F\*G)** | +(- | **ABC** |
| **\*D+E/(F\*G)** | + | **ABC-** |
| **D+E/(F\*G)** | +\* | **ABC-** |
| **+E/(F\*G)** | +\* | **ABC-D** |
| **E/(F\*G)** | + | **ABC-D\*+** |
| **/(F\*G)** | + | **ABC-D\*+E** |
| **(F\*G)** | +/ | **ABC-D\*+E** |
| **F\*G)** | +/( | **ABC-D\*+E** |
| **\*G)** | +/( | **ABC-D\*+EF** |
| **G)** | +/(\* | **ABC-D\*+EF** |
| **)** | +/(\* | **ABC-D\*+EFG** |
| **{Empty}** | +/ | **ABC-D\*+EFG\*** |
| **{Empty}** | {Empty} | **ABC-D\*+EFG\*/+** |

### Converting a prefix expression to a postfix expression.

Read each character of the string from right to left. If it’s an operator, we attach the operator to integer 2 (using pair in c++). And push the pair to stack. If it’s an operand, we output it and get the highest pair of stack mines 1. If the highest pair of stack equal to 0 then we get the pair out of stack and output operand. While reading all the character in prefix expression, we output each character left in stack sequently.

E.g: convert prefix expression: + x – y z to postfix expression.

|  |  |  |
| --- | --- | --- |
| Prefix | Stack | Postfix |
| + x – y z | {Empty} | **{Empty}** |
| x – y z | {+,2} | **{Empty}** |
| – y z | {+,1} | **x** |
| Y z | {+,1} , {-,2} | **x** |
| Z | {+,1},{-,1} | **x y** |
| {Empty} | {+,1} | **x y z -** |
| {Empty} | {Empty} | **x y z - +** |

## 1.2 Programming.

Struct node (declare struct node type): each node includes 2 information: value and pointer to the next node.

Algorithm:

* Class stack: initiate 1 node\*p is p\_high to save the highest node in stack (value of p\_high is NULL at first) and initiate a count variable start from 0.
* Push: create a new node in order to save input’s value and a pointer to current p\_high, change p\_high with the new node and update the count of nodes in stack.
* Pop: if stack is not empty, delete current p\_high and update new p\_high is the next node of the old node, then update the count of node in stack.
* Top: if stack is not empty then return the value of the highest node(p\_high).
* Size: return the value of count of node in stack.
* Class Destructor: delete all elements in stack.

template <class T>

struct node

{

T value;

node\* next;

};

template <class T>

class stack

{

private:

node<T>\* p\_high = NULL;

int count = 0;

public:

~stack()

{

while (count > 0)

this->pop();

}

void push(T);

void pop();

T top();

int size();

};

template <class T>

void stack<T>::push(T value)

{

node<T>\* temp;

temp = new node<T>;

temp->value = value;

temp->next = p\_high;

p\_high = temp;

count++;

}

template <class T>

void stack<T>::pop()

{

if (p\_high == NULL)

return;

node<T>\* temp;

temp = p\_high;

p\_high = p\_high->next;

delete temp;

count--;

}

template <class T>

T stack<T>::top()

{

if (p\_high != NULL)

return p\_high->value;

}

template <class T>

int stack<T>::size()

{

return count;

}

int Rank (c): return the priority of operator, operand c.

Algorithm:

* Check character c, if it’s ‘ ‘, the priority is 0, integer from 0 to 9 or “.”,the priority is 1, plus “+” or mine “-“ is 2, multiply “\*” or divine “/” is 3, open bracket “(“ is 4 and close bracket “)” is 5.

int Rank(char c)

{

if (c == ' ')

return 0;

if (c >= '0' && c <= '9')

return 1;

if (c == '+' || c == '-')

return 2;

if (c == '\*' || c == '/' || c == '^')

return 3;

if (c == '(')

return 4;

if (c == ')')

return 5;

if (c == '.')

return 6;

}

bool CheckExpression (string s) return true if s is a right expression, false if not and Rank (c) to support CheckExpression.

Algorithm:

- We have some rules to check the validity of an expression:

* The number of “open bracket” always equal or higher than “close bracket” from the beginning of string to any character or always equal from start to end.
* 2 of Operator and operand must not be next to each other.
* After “open bracket” must be operand.
* After “close bracket” must be operator or end of expression.
* First character is not an operator.

- Algorithm:

* A loop running from 0 to the end of length of string s.
* Check variable to save the rank of previous operand/operator.
* Compare the rank of previous operand/operator to current operand/operator, if violate the rules, return false.
* End of loop, update the check variable by the rank of current operand/operator.
* During the loop, if it’s “open bracket” then push to stack, if it’s “close bracket” then compare it to the highest operator of stack, if it’s a right bracket, delete “open bracket” in highest of stack, else return false.
* . Return true.

bool CheckExpression(string s)

{

if (Rank(s[0]) == 2 || Rank(s[0]) == 3 || Rank(s[0]) == 6)

return false;

int check = 0;

bool dot = true;

stack<char> temp;

for (int i = 0; i < s.length(); i++)

{

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

{

temp.push(s[i]);

s[i] = '(';

dot = true;

}

if (s[i] == ')' || s[i] == ']' || s[i] == '}')

{

if (temp.size() < 1)

return false;

if (s[i] == ')' && temp.top() != '(')

return false;

if (s[i] == ']' && temp.top() != '[')

return false;

if (s[i] == '}' && temp.top() != '{')

return false;

temp.pop();

s[i] = ')';

dot = true;

}

if (check == 1)

{

if (Rank(s[i]) == 1 && Rank(s[i - 1]) == 0)

return false;

if (Rank(s[i]) == 4)

return false;

}

if (check == 2 || check == 3)

{

if (Rank(s[i]) == 2 || Rank(s[i]) == 3)

return false;

dot = true;

}

if (check == 4)

{

if (Rank(s[i]) == 2 || Rank(s[i]) == 3 || Rank(s[i]) == 5)

return false;

dot = true;

}

if (check == 5)

{

if (Rank(s[i]) == 1 || Rank(s[i]) == 4)

return false;

dot = true;

}

if (check == 6)

{

if (Rank(s[i]) != 1)

return false;

if (!dot)

return false;

dot = false;

}

if (Rank(s[i]) > 0)

check = Rank(s[i]);

}

if (temp.size() != 0)

return false;

return true;

}

string postfix(string s) return postfix expression from infix expression of s. The algorithm has been displayed at 1.1.

string postfix(string s)

{

string s\_new = "";

stack<char> temp;

for (int i = 0; i < s.length(); i++)

{

if (Rank(s[i]) == 0 && s\_new[s\_new.length() - 1] != ' ')

{

s\_new += s[i];

continue;

}

if (Rank(s[i]) == 1 || Rank(s[i]) == 6)

{

s\_new += s[i];

continue;

}

if (Rank(s[i]) == 2)

{

if (temp.size() > 0 && Rank(temp.top()) <= 3 && Rank(temp.top()) >= 2)

{

if (s\_new[s\_new.length() - 1] != ' ')

s\_new += ' ';

s\_new += temp.top();

temp.pop();

}

temp.push(s[i]);

continue;

}

if (Rank(s[i]) == 3)

{

if (temp.size() > 0 && Rank(temp.top()) == 3)

{

if (s\_new[s\_new.length() - 1] != ' ')

s\_new += ' ';

s\_new += temp.top();

temp.pop();

}

temp.push(s[i]);

continue;

}

if (Rank(s[i]) == 4)

temp.push(s[i]);

if (Rank(s[i]) == 5)

{

while (temp.top() != '(')

{

if (s\_new[s\_new.length() - 1] != ' ')

s\_new += ' ';

s\_new += temp.top();

temp.pop();

}

temp.pop();

}

}

while (temp.size() > 0)

{

if (s\_new[s\_new.length() - 1] != ' ')

s\_new += ' ';

s\_new += temp.top();

temp.pop();

}

return s\_new;

}

bool calculation (string s, float output) return the value of expressions string s (s is a postfix expression).

Idea:

* Create an empty stack.
* Create a loop with the stop condition is read all the character of string s.
* If it’s operand, push it to stack.
* If it’s operator, we calculate 2 operands which are got from stack then push the result to stack.
* Do to the end of string, if the expression return false, output “E” else output the result.

bool calculation(string s, float& output)

{

stack<float> temp;

stringstream ss(s);

float f, float\_temp1, float\_temp2;

char c;

while (!ss.eof())

{

ss >> c;

if (c >= '0' && c <= '9')

{

ss.seekg(-1, ios::cur);

ss >> f;

temp.push(f);

continue;

}

if (temp.size() >= 2)

{

float\_temp2 = temp.top();

temp.pop();

float\_temp1 = temp.top();

temp.pop();

if (c == '+')

temp.push(float\_temp1 + float\_temp2);

if (c == '-')

temp.push(float\_temp1 - float\_temp2);

if (c == '\*')

temp.push(float\_temp1 \* float\_temp2);

if (c == '/')

if (float\_temp2 != 0)

temp.push(float\_temp1 / float\_temp2);

else

return false;

if (c == '^')

temp.push(pow(float\_temp1, float\_temp2));

}

}

output = temp.top();

return true;

}

Main: user input file’s path and choose the action (-c) to calculate, (-t) to convert to postfix expression. If the expression is wrong, output “E”.

int main()

{

ifstream input;

ofstream output;

string s, s1, PostFix;

string choice;

int n;

cout << "nhap file txt input: " << endl;

cin >> s;

input.open(s);

if (!input.is\_open())

{

cout << "file nhap vao khong hop le " << endl;

}

else {

cout << "nhap so luong phep tinh: " << endl;

cin >> n;

cout << "chon hanh dong: " << endl;

cout << "-c: tinh toan " << endl;

cout << "-t: chuyen doi" << endl;

cin >> choice;

string a;

float cal;

if (choice == "-c")

{

cout << "nhap file txt output: " << endl;

cin >> s1;

output.open(s1);

for (int i = 0; i < n; i++)

{

getline(input, a);

if (CheckExpression(a) == true)

{

PostFix = postfix(a);

if (calculation(PostFix, cal) == true)

{

output << setprecision(3) << cal << endl;

}

else

output << "E" << endl;

}

else

{

output << "E" << endl;

}

}

}

else if (choice == "-t")

{

cout << "nhap file txt output: " << endl;

cin >> s1;

output.open(s1);

for (int i = 0; i < n; i++)

{

getline(input, a);

if (CheckExpression(a) == true)

{

PostFix = postfix(a);

output << PostFix << endl;

}

else

{

output << "E" << endl;

}

}

}

else

cout << "nhap khong hop le " << endl;

}

input.close();

output.close();

}

# REFERENCES

<https://yinyangit.wordpress.com/2011/01/26/algorithm-chuy%E1%BB%83n-bi%E1%BB%83u-th%E1%BB%A9c-trung-t%E1%BB%91-sang-ti%E1%BB%81n-t%E1%BB%91-va-h%E1%BA%ADu-t%E1%BB%91-b%E1%BA%B1ng-stack/>

<https://www.youtube.com/watch?v=MyCBpybscNo>

<https://www.youtube.com/watch?v=OPmN_YYQmro>

<https://www.stdio.vn/giai-thuat-lap-trinh/ung-dung-stack-bieu-thuc-tien-to-prefix-hspu1>