**Assignment-2**

**Name:**

**Roll No:**

**Branch:** S.E. I.T.

**Date:**

**Title: Stack**

/\* Implement stack as an abstract data type using singly linked list and use this ADT for conversion

of infix expression to postfix, prefix and evaluation of postfix and prefix expression.\*/

* **Program**

/\*Ass2\_new.cpp\*/

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

// Stack Class Implementation

class ClassStack {

vector<int> stack;

public:

bool IsEmpty() {

return stack.empty();

}

void Push(int value) {

stack.push\_back(value);

}

int Pop() {

if (IsEmpty()) {

cout << "Stack Underflow!" << endl;

exit(1); // Handle error gracefully

}

int top = stack.back();

stack.pop\_back();

return top;

}

int StackTop() {

if (IsEmpty()) {

cout << "Stack is empty!" << endl;

exit(1);

}

return stack.back();

}

};

// Conversion Class

class Conversion {

string expr, output;

const char\* token;

public:

Conversion() : token(nullptr) {}

void getExpr();

bool validExpr();

void display();

bool isOperator(char);

int priority(char);

bool IsAssoRL(char);

void PostfixConversion();

void PrefixConversion();

void EvalPostfix();

void EvalPrefix();

};

// Accept Infix Expression

void Conversion::getExpr() {

output = "";

cout << "\nEnter an infix expression: ";

cin >> expr;

}

// Display the Output Expression

void Conversion::display() {

cout << "OUTPUT: " << output << endl;

}

// Checking for operator

bool Conversion::isOperator(char ch) {

return (ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '%' || ch == '^');

}

// Check if the entered infix expression is valid

bool Conversion::validExpr() {

int pcount = 0, oper = 0, operand = 0;

for (token = &expr[0]; \*token != '\0'; token++) {

if (\*token == '(')

pcount++;

else if (\*token == ')')

pcount--;

else if (isOperator(\*token))

oper++;

else if (isalnum(\*token))

operand++;

}

if (pcount != 0 || operand != oper + 1) {

cout << "Invalid Expression." << endl;

return false;

}

return true;

}

// Assign Priority to Operators

int Conversion::priority(char oper) {

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

return 1;

else if (oper == '\*' || oper == '/' || oper == '%')

return 2;

else if (oper == '^')

return 3;

return 0;

}

// Check for Right-to-Left Associativity

bool Conversion::IsAssoRL(char oper) {

return (oper == '^');

}

// Convert Infix to Postfix

void Conversion::PostfixConversion() {

ClassStack s;

for (token = &expr[0]; \*token != '\0'; token++) {

if (isalnum(\*token))

output += \*token;

else if (\*token == '(')

s.Push(\*token);

else if (\*token == ')') {

while (s.StackTop() != '(')

output += s.Pop();

s.Pop();

} else if (isOperator(\*token)) {

while (!s.IsEmpty() && priority(\*token) <= priority(s.StackTop())) {

if (priority(\*token) == priority(s.StackTop()) && IsAssoRL(\*token))

break;

output += s.Pop();

}

s.Push(\*token);

}

}

while (!s.IsEmpty())

output += s.Pop();

}

// Convert Infix to Prefix

void Conversion::PrefixConversion() {

ClassStack s;

string tempres;

int cnt = expr.length();

for (int i = cnt - 1; i >= 0; i--) {

char ch = expr[i];

if (isalnum(ch))

tempres += ch;

else if (ch == ')')

s.Push(ch);

else if (ch == '(') {

while (s.StackTop() != ')')

tempres += s.Pop();

s.Pop();

} else if (isOperator(ch)) {

while (!s.IsEmpty() && priority(ch) < priority(s.StackTop())) {

tempres += s.Pop();

}

s.Push(ch);

}

}

while (!s.IsEmpty())

tempres += s.Pop();

// Reverse the result to get the prefix expression

output = string(tempres.rbegin(), tempres.rend());

}

// Calculator Function

int calculate(int op1, char oper, int op2) {

switch (oper) {

case '+': return op1 + op2;

case '-': return op1 - op2;

case '\*': return op1 \* op2;

case '/': return op1 / op2;

case '^': return pow(op1, op2);

}

return -1;

}

// Evaluate Postfix Expression

void Conversion::EvalPostfix() {

ClassStack s;

int op1, op2, res;

for (token = &output[0]; \*token != '\0'; token++) {

if (isdigit(\*token))

s.Push(\*token - '0');

else if (isOperator(\*token)) {

op2 = s.Pop();

op1 = s.Pop();

res = calculate(op1, \*token, op2);

s.Push(res);

}

}

cout << "Evaluated value is: " << s.Pop() << endl;

}

// Evaluate Prefix Expression

void Conversion::EvalPrefix() {

ClassStack s;

int op1, op2, res;

for (int i = output.length() - 1; i >= 0; i--) {

char ch = output[i];

if (isdigit(ch))

s.Push(ch - '0');

else if (isOperator(ch)) {

op1 = s.Pop();

op2 = s.Pop();

res = calculate(op1, ch, op2);

s.Push(res);

}

}

cout << "Evaluated value is: " << s.Pop() << endl;

}

// Main Function

int main() {

Conversion e, p;

bool res;

int ch;

do {

cout << "\nConversions\n";

cout << "1. Infix to Postfix Conversion\n";

cout << "2. Infix to Prefix Conversion\n";

cout << "3. Exit\n";

cout << "Enter your choice: ";

cin >> ch;

switch (ch) {

case 1:

do {

e.getExpr();

res = e.validExpr();

} while (!res);

e.PostfixConversion();

e.display();

e.EvalPostfix();

break;

case 2:

do {

p.getExpr();

res = p.validExpr();

} while (!res);

p.PrefixConversion();

p.display();

p.EvalPrefix();

break;

case 3:

return 0;

}

} while (ch != 3);

}

/\*stack1.h\*/

#include<iostream>

using namespace std;

struct StructStack

{

char Data;

StructStack \*Next;

};

class ClassStack

{

StructStack \*Top;

public:

ClassStack()

{

Top=NULL;

}

bool IsFull()

{

StructStack \*Temp;

Temp = new StructStack;

if(Temp == NULL)

return true;

else

{

delete Temp;

return false;

}

}

bool IsEmpty()

{

if(Top == NULL)

return true;

else

return false;

}

bool Push(char Data)

{

if(IsFull())

return false;

else

{

StructStack \*Temp;

Temp = new StructStack;

Temp->Data = Data;

Temp->Next = Top;

Top = Temp;

return true;

}

}

char Pop()

{

if (IsEmpty())

//return -1;

cout<<"stack empty";

else

{

char Data;

StructStack \*Temp;

Temp = Top;

Data = Top->Data;

Top = Top->Next;

delete Temp;

return Data;

}

}

char StackTop()

{

if(IsEmpty())

// return -1;

cout<<"stack empty";

else

return Top->Data;

}

~ClassStack()

{

while(Top!=NULL)

{

StructStack \*Temp;

Temp = Top;

Top = Top->Next;

delete Temp;

}

}

};

***OUTPUT***

Conversions

1. Infix to Postfix Conversion

2. Infix to Prefix Conversion

3. Exit

Enter your choice: 1

Enter an infix expression: a+b-c\*d^e

OUTPUT: ab+cde^\*-Enter the value of a: 3

Enter the value of b: 4

Enter the value of c: 5

Enter the value of d: 8

Enter the value of e: 6

Evaluated value is : 7

Conversions

1. Infix to Postfix Conversion

2. Infix to Prefix Conversion

3. Exit

Enter your choice: 2

Enter an infix expression: a+b-c\*d^e

OUTPUT: -+ab\*c^deEnter the value of e: 4

Enter the value of d: 8

Enter the value of c: 6

Enter the value of b: 5

Enter the value of a: 2

Evaluated value is : 7

Conversions

1. Infix to Postfix Conversion

2. Infix to Prefix Conversion

3. Exit

Enter your choice: 3

-------------------------------------------------