Project Report

INFIX TO POSTFIX CALCULATOR

DATA STRUCTURE AND ALGORITHMS

DEPT OF COMPUTER SCIENCES

4/07/2022 Bahria University Karachi Campus

ACKNOWLEDGMENT

We have taken efforts in this project. However, it would not have been possible without the kind support and help of some of our friends. I would like to extend my sincere thanks to all of them. I am highly indebted to “Miss Saba” for their guidance and constant supervision as well as providing necessary information regarding the project and their support through videos in the completion of our project. We would like to express my special thanks of gratitude to our institute who gave us the golden opportunity to do this wonderful project on the topic “INFIX TO POSTFIX CALCULATOR” and we learn a lot of things when doing research for this project. In the end our thanks and appreciations also go to our colleagues in developing the project and people who have willingly helped me out with their abilities.

GROUP MEMBERS

AQSA MALIK (02-134211-106)

MUHAMMAD HUZAIFA (02-134211-059)

ASHIR AZEEM (02-134211-039)

SUBMITTED TO

LAB INSTRUCTOR:

MISS SABA

COURSE INSTRUCTOR

MISS LUBNA

DATE OF SUBMISSION

04-JULY-2022

TABLE OF CONTENT

CHAPTER ONE

1.1 Overview
1.2 Significance of Study
1.3 Related Work
1.4 Background Study
1.5 Information about programming languages for the project
1.6 Objective

CHAPTER TWO

2.1 Hardware Requirements

2.2 Software Requirements

CHAPTER THREE

3.1 Algorithms

3.2 Flowchart

CHAPTER FOUR

4.1 Code

4.2 Snippet

CHAPTER FIVE

5.1 Future Work

5.2 Examples

CHAPTER SIX

6.1 Conclusion

CHAPTER #01

1.1 Overview:

In this project we take input of infix notation from user and covert it into postfix and evaluate it. Just suppose we give a infix notation it will converts into postfix then evaluate it i.e.: we are evaluating postfix expression and then there are operands like 1 till 9 then value will display otherwise if operand are in the form of alphabets a,b,c etc, then it will show a message.

Infix notation is an arithmetic and logical notation which represents the operator placed between two operands. It is not easy to parse by the computer. In infix notation, the order of operations is mandatory to indicate, and operands and operators must be surrounded by parentheses. Post-fix notation is a mathematical notation which is used to parse a machine. Post-fix notation is also called Reverse Polish.

This problem requires you to write a program to convert an infix expression to a postfix expression. The evaluation of an infix expression such as A + B * C requires knowledge of which of the two operations, + and *, should be performed first. In general, A + B * C is to be interpreted as A + (B * C) unless otherwise specified. We say that multiplication takes precedence over addition. Suppose that we would now like to convert A + B * C to postfix. Applying the rules of precedence, we first convert the portion of the expression that is evaluated first, namely the multiplication. Doing this conversion in stages, we obtain

A + B * C Given infix form

A + B C * Convert the multiplication

A B C * + Convert the addition

The major rules to remember during the conversion process are that the operations with highest precedence are converted first and that after a portion of an expression has been converted to postfix, it is to be treated as a single operand. Let us now consider the same example with the precedence of operators reversed by the deliberate insertion of parentheses.

(A + B) * C Given infix form

A B + * C Convert the addition

A B + C * Convert the multiplication

Note that in the conversion from AB + * C to AB + C *, AB+ was treated as a single operand. The rules for converting from infix to postfix are simple, if you know the order of precedence.

We consider five binary operations: addition, subtraction, multiplication, division, and exponentiation. These operations are denoted by the usual operators, +, –, *, /, and ^, respectively. There are three levels of operator precedence. Both * and / have higher precedence than + and –. ^ has higher precedence than * and /. Furthermore, when operators of the same precedence are scanned, +, –, * and / are left associative, but ^ is right associative. Parentheses may be used in infix expressions to override the default precedence.

1.2 Significance Of Study:

Postfix has several advantages over infix for expressing algebraic formulas. First any formula can be expressed without parenthesis. Second, it is very convenient for evaluating formulas on computers with stack. Third, infix operators have precedence.

1.3 Related Work:

In converting infix expressions to postfix notation, the following fact should be taken into consideration: In infix form, the order of applying operators is governed by the possible appearance of parentheses and the operator precedence relations; however, in postfix form, the order is simply the "natural" order – i.e., the order of appearance from left to right.

Accordingly, subexpressions within innermost parentheses must first be converted to postfix, so that they can then be treated as single operands. In this fashion, parentheses can be successively eliminated until the entire expression has been converted. The last pair of parentheses to be opened within a group of nested parentheses encloses the first subexpression within the group to be transformed. This last-in, first-out behavior should immediately suggest the use of a stack.

Your program should utilize the basic stack methods. You will need to PUSH certain symbols on the stack, POP symbols, test to see if the stack is EMPTY, look at the TOP element of the stack, etc. In addition, you must devise a Boolean method that takes two operators and tells you which has higher precedence

1.4 Background study:

The main data structure which we implement in this project is Stack and the principle by which stack is ordered is called LIFO, the distinguishing characteristic of a stack is that the addition or removal of items takes place at the same end The advantage of postfix notation in data structure is you don't need rules of precedence. You don't need rules for right and left associativity. You don't need parentheses to override the above rules.

1.5 Information about programming languages for the project:

Our project is based on C++programming language. So, for this project you must know about appropriate Data Structures and Algorithms and their proper use in C++ programming language. Some of the data structures are graph, stack, Binary Search tree, queue, linked list, hash table, arrays but for this project the main data structure which we must know about is stack and in our project, we implement stack by using the C++ programming language.

1.6 Objective:

The main objection is a collection of error-free simple arithmetic expressions. Expressions are presented one per line. The input has an arbitrary number of blanks between any two symbols. A symbol may be a letter (A – Z), an operator (+, –, *, or /), a left parenthesis, or a right parenthesis. Each operand is composed of a single letter. The input expressions are in infix notation.

CHAPTER #02

2.1 Software Requirements:

- Visual Studio 2022
- Visual Studio 2012

2.2 Hardware Requirements:

- Laptop/PC

CHAPTER #03

Infix to Postfix conversion algorithm:

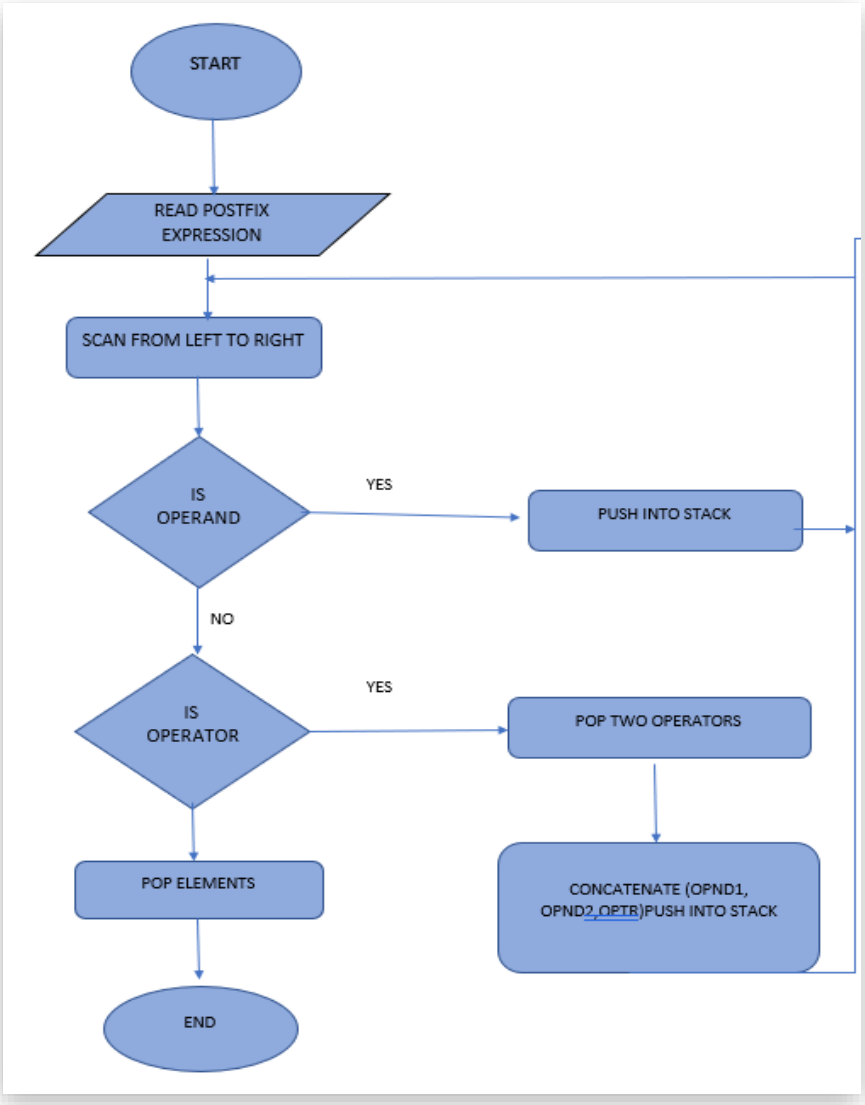
- There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers.
- Visual Studio 2022
- The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.
- In this algorithm, all operands are printed (or sent to output) when they are read. There are more complicated rules to handle operators and parentheses.

Algorithm:

- POLISH(Q,P)
- Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.
- Push "(" onto STACK , add ")" to the end of Q.
- Scan Q from left to right and repeat Step 3 to 6 for each element Q until the STACK is empty:
- If an operand is encountered, add it to P.
- If a left parenthesis is countered push it onto the STACK.
- If an operator ϕ is encountered then
- a. Repeatedly pop from the STACK and add to P each operator (on the top of the STACK) which has the same precedence as or higher precedence than ϕ.
- b. Add ϕ to STACK
- [End of If Structure]
- If a right parenthesis is encountered, then:
- a. Repeatedly pop from the STACK and add to P each operator (on the top of the STACK) until a left parenthesis is encountered.

- b. Remove the left parenthesis. [Do not add the left parenthesis to P].
- [End of if structure]
- [End of If structure]
- Exit

3.1 FlowChart:



CHAPTER #04

4.1 Code:

```
#include<iostream>
#include<stack>
using namespace std;

bool IsOperator(char);
bool IsOperand(char);
bool eqlOrhigher(char, char);
string convert(string);

bool IsOperator(char c)
{
	if (c == '+' || c == '-' || c == '*' || c == '/' || c == '^')
		return true;
	return false;
}


bool IsOperand(char c)
{
	if (c >= 'A' && c <= 'Z')
		return true;
	if (c >= 'a' && c <= 'z')
```

```
                return true;
        if (c >= '0' && c <= '9')
                return true;
        return false;
}

int precedence(char op)
{
        if (op == '+' || op == '-')
                return 1;
        if (op == '*' || op == '/')
                return 2;
        if (op == '^')
                return 3;
        return 0;
}


bool eqlOrhigher(char op1, char op2)
{
        int p1 = precedence(op1);
        int p2 = precedence(op2);
        if (p1 == p2)
        {
                if (op1 == '^')
                        return false;
                return true;
```

```
        }
        return  (p1 > p2 ? true : false);
}


string convert(string infix)
{
        stack <char> S;
        string postfix = "";
        char ch;

        S.push('(');
        infix += ')';

        for (int i = 0; i < infix.length(); i++)
        {
                ch = infix[i];

                if (ch == ' ')
                        continue;
                else if (ch == '(')
                        S.push(ch);
                else if (IsOperand(ch))
                        postfix += ch;
                else if (IsOperator(ch))
                {
                        while (!S.empty() && eqlOrhigher(S.top(), ch))
```

```
                {
                        postfix += S.top();
                        S.pop();
                }
                S.push(ch);
        }
        else if (ch == ')')
        {
                while (!S.empty() && S.top() != '(')
                {
                        postfix += S.top();
                        S.pop();
                }
                S.pop();
        }
    }
    return postfix;
}
char expr[50];
int SIZE = sizeof(expr) / sizeof(expr[0]);

int top = -1;
void PUSH(char element) {
    if (top == SIZE - 1) {
        cout << "Stack is full" << endl;

    }
```

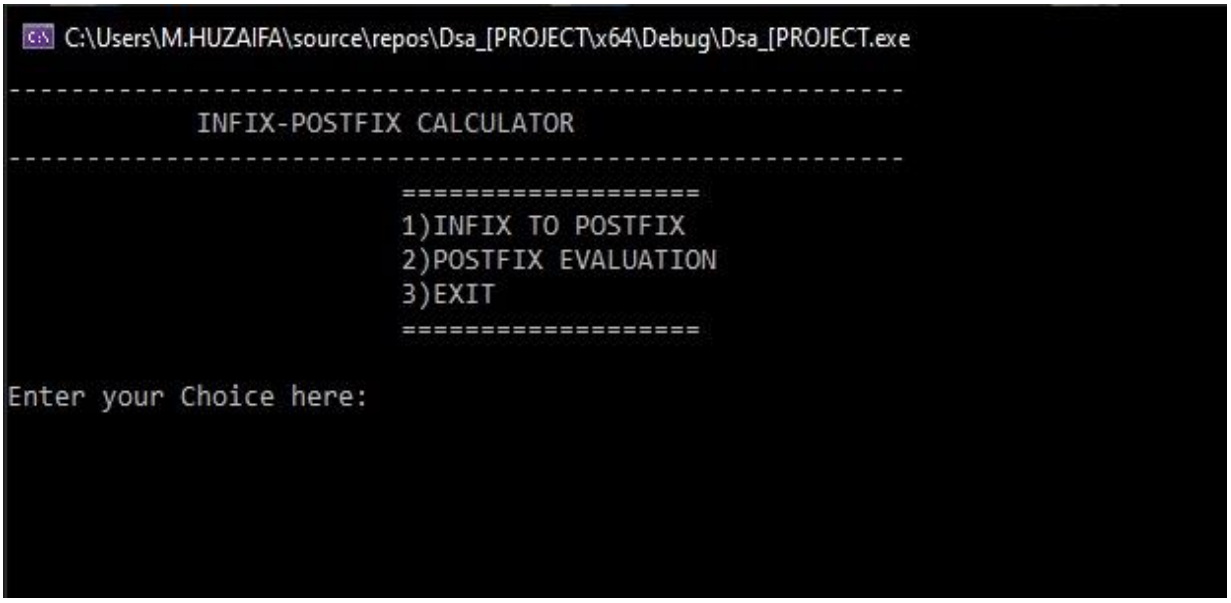
```
	top++;
	expr[top] = element;
}
int POP() {
	if (top == -1) {
		//cout << "Stack is empty" << endl;
		return 0;

	}
	int returnValue = expr[top];
	top--;
	return returnValue;
}
int postFix(char* exp) {
	for (int i = 0; i < SIZE; i++)
	{
		if (exp[i] == ')')
		{
			return POP();
		}
		else if (isdigit(exp[i]))
		{
			PUSH(exp[i] - '0');
		}
		else
		{
			int num1 = POP();
```

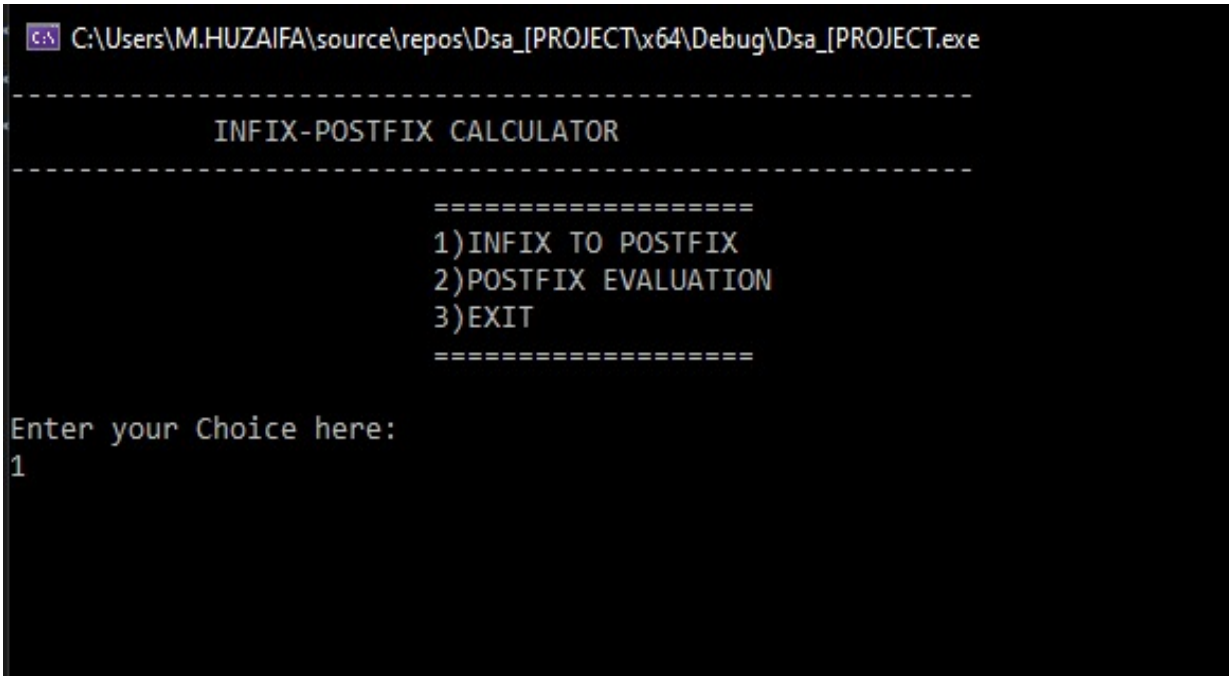
```
                int num2 = POP();
                switch (exp[i])
                {
                case'+':
                        PUSH(num2 + num1);
                        break;
                case'-':
                        PUSH(num2 - num1);
                        break;
                case'*':
                        PUSH(num2 * num1);
                        break;
                case'/':
                        PUSH(num2 / num1);
                        break;
                }
            }
        }
        return POP();
}
void display() {
        for (int i = 0; i < SIZE; i++) {
            cout << expr[i] << " ";
        }
        cout << endl;
}
```

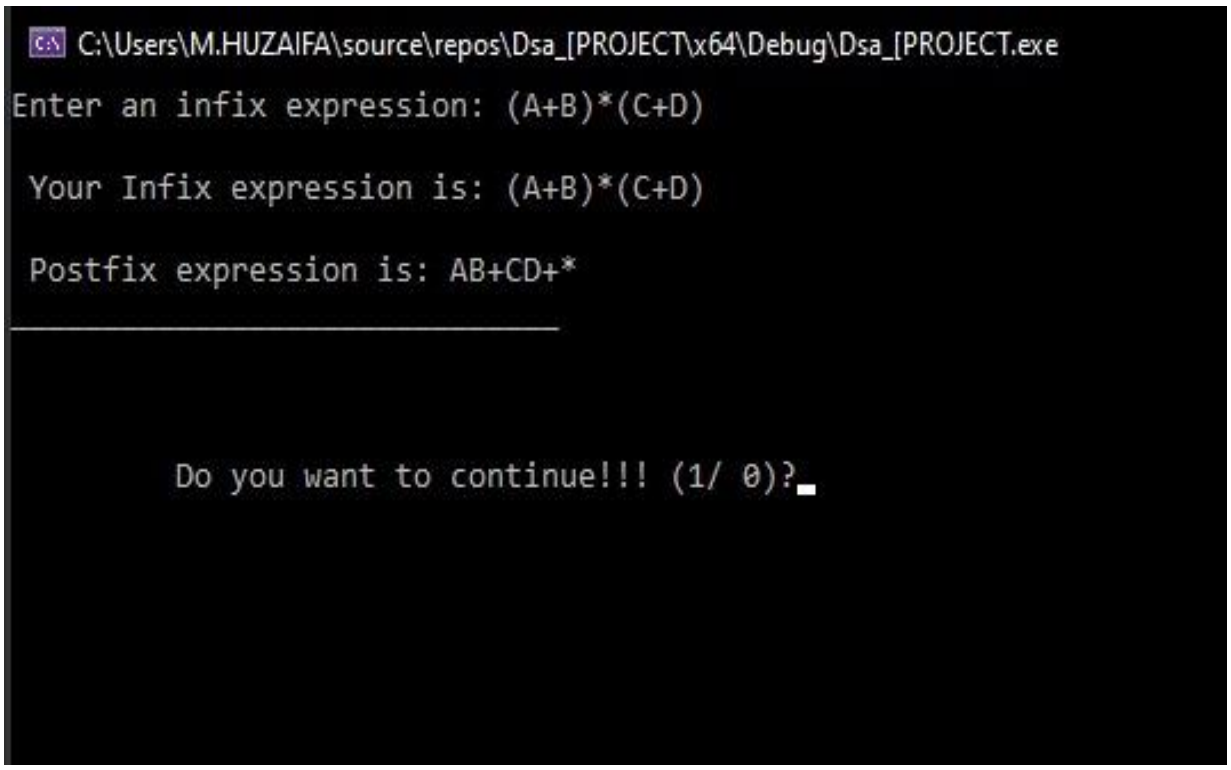
```
int main()
{
	string infix_expression, postfix_expression;
	char ex[50];
	int ch;
	int input;
	do
	{
		system("CLS");
	cout << "-----------------------------------------------------" << endl;
	cout << "           INFIX-POSTFIX CALCULATOR              " << endl;
	cout << "-----------------------------------------------------" << endl;
	cout << "\t\t\t 1)INFIX TO POSTFIX" << endl;
	cout << "\t\t\t 2)POSTFIX EVALUATION" << endl;
	cout << "\t\t\t 3)EXIT" << endl;
	cout << endl;
	cout << "Enter your Choice here: ";
	cin >> input;
	switch (input)
	{
	case 1:
		system("CLS");
		cout << "Enter an infix expression: ";
		cin >> infix_expression;
		postfix_expression = convert(infix_expression);
		cout << "\n Your Infix expression is: " << infix_expression << endl;
		cout << "\n Postfix expression is: " << postfix_expression;
```

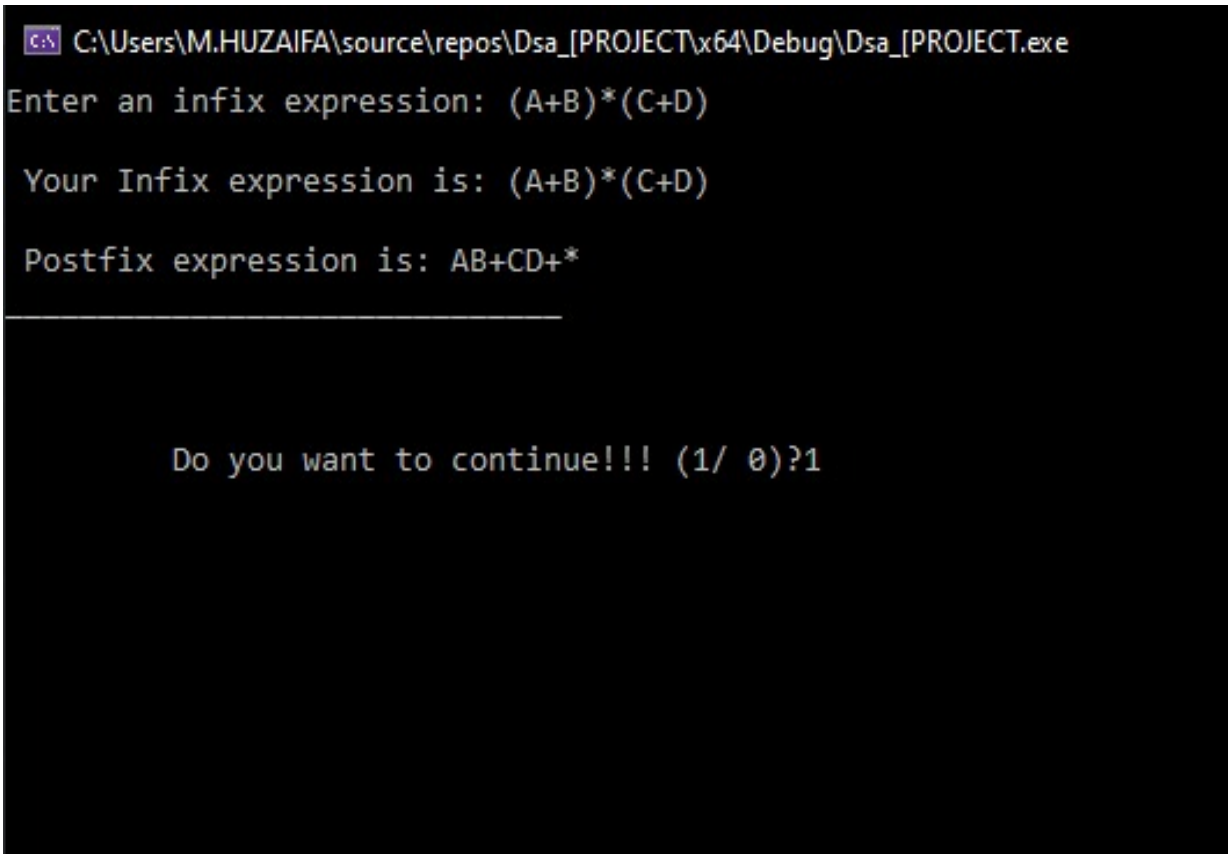
```
            cout << endl;
            cout << endl;
            break;

        case 2:
            system("CLS");
            cout << "Enter PostFix Expression Again: ";
            cin >> ex;
            cout << "Values is =  " << postFix(ex);
            break;

        case 3:
            system("CLS");
            exit(0);
            break;
        }
        cout << "\n \t Do you want to continue!!! (1/ 0)?";
        cin >> ch;
        } while (ch == 1);
        return 0;
}
.
```

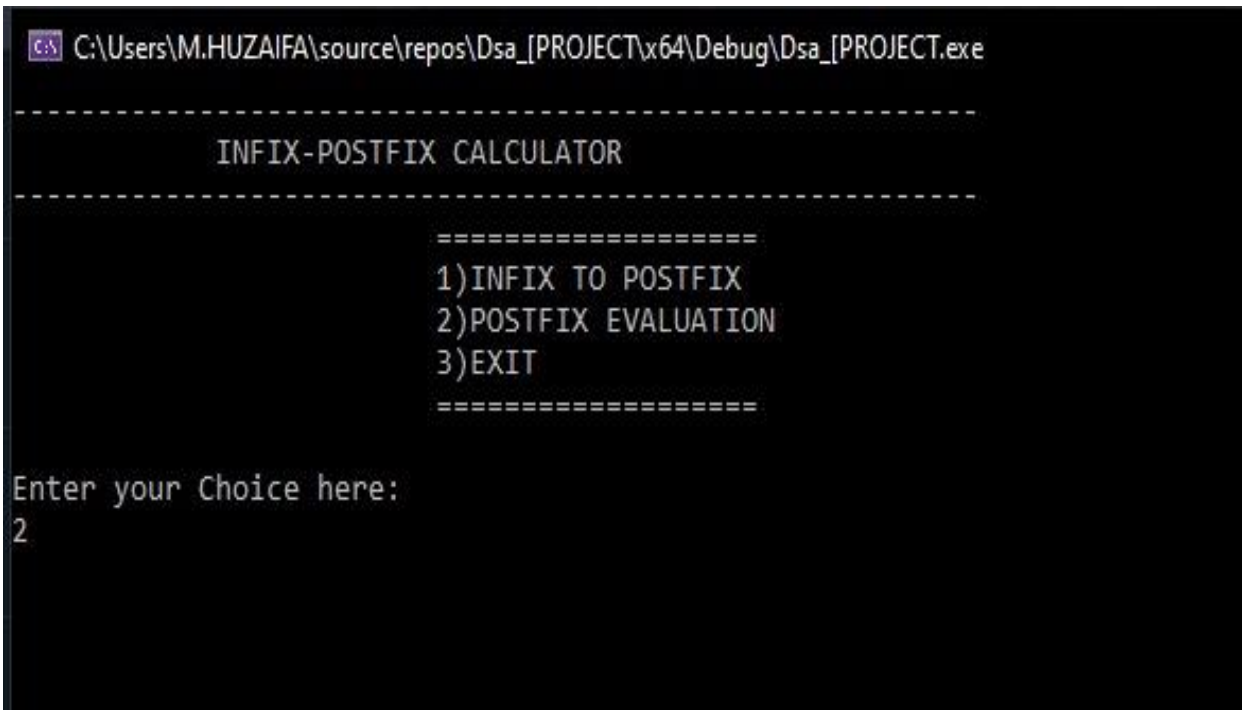
4.2 Snippets:

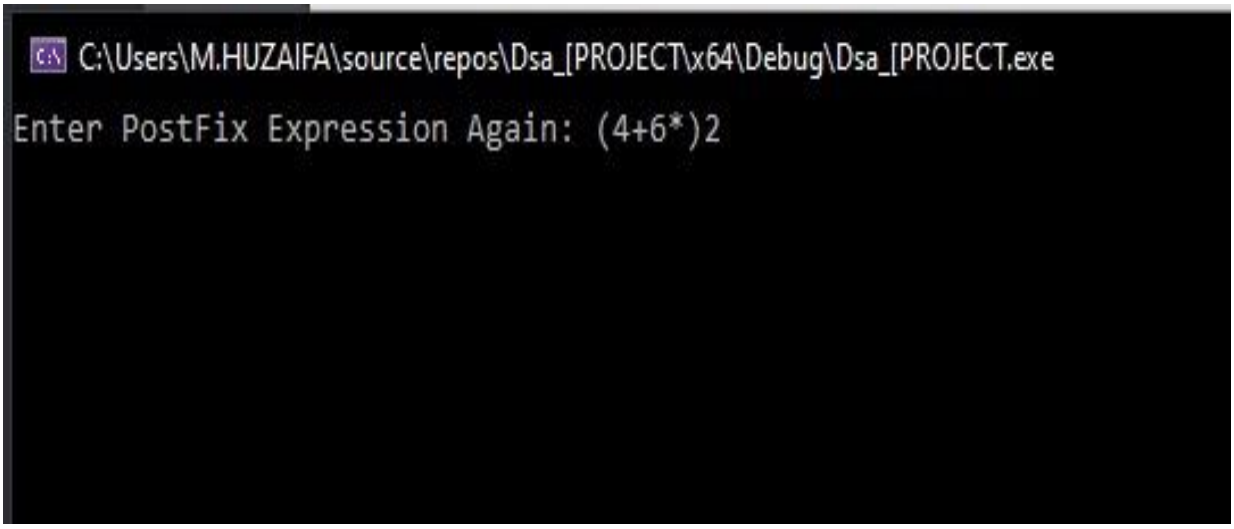

C:\Users\M.HUZAIFA\source\repos\Dsa_[PROJECT\x64\Debug\Dsa_[PROJECT.exe
INFIX-POSTFIX CALCULATOR
==========
1)INFIX TO POSTFIX
2)POSTFIX EVALUATION
3)EXIT
==========
Enter your Choice here:

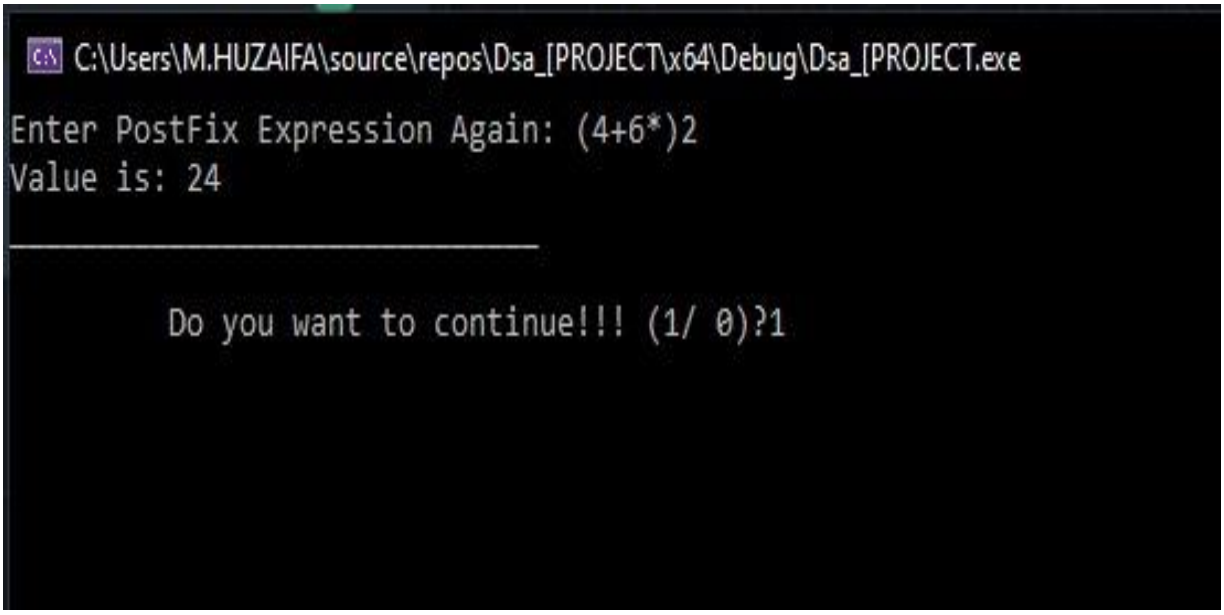

C:\Users\M.HUZAIFA\source\repos\Dsa_[PROJECT\x64\Debug\Dsa_[PROJECT.exe
INFIX-POSTFIX CALCULATOR
==========
1)INFIX TO POSTFIX
2)POSTFIX EVALUATION
3)EXIT
==========
Enter your Choice here:
1

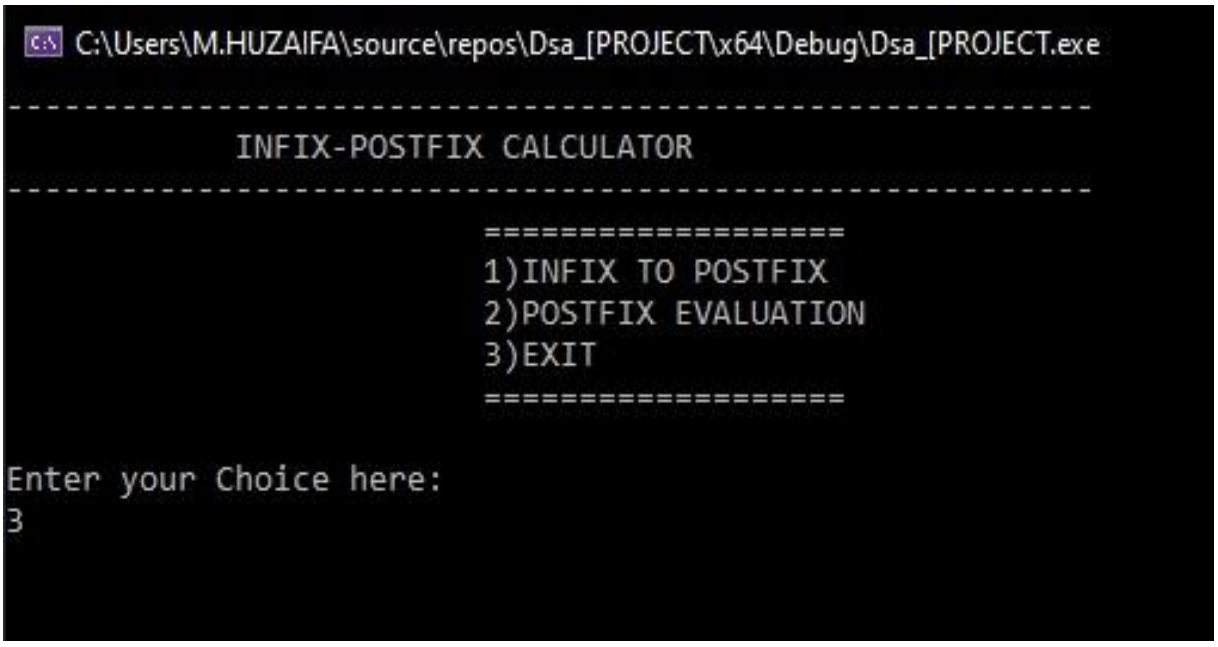

C:\Users\M.HUZAIFA\source\repos\Dsa_[PROJECT\x64\Debug\Dsa_[PROJECT.exe
Enter an infix expression: (A+B)*(C+D)
Your Infix expression is: (A+B)*(C+D)
Postfix expression is: AB+CD+*
Do you want to continue!!! (1/ 0)?


C:\Users\M.HUZAIFA\source\repos\Dsa_[PROJECT\x64\Debug\Dsa_[PROJECT.exe
Enter an infix expression: (A+B)*(C+D)
Your Infix expression is: (A+B)*(C+D)
Postfix expression is: AB+CD+*
Do you want to continue!!! (1/ 0)?1


C:\Users\M.HUZAIFA\source\repos\Dsa_[PROJECT\x64\Debug\Dsa_[PROJECT.exe
INFIX-POSTFIX CALCULATOR
1)INFIX TO POSTFIX
2)POSTFIX EVALUATION
3)EXIT
Enter your Choice here:
2


C:\Users\M.HUZAIFA\source\repos\Dsa_[PROJECT\x64\Debug\Dsa_[PROJECT.exe
Enter PostFix Expression Again: (4+6*)2


C:\Users\M.HUZAIFA\source\repos\Dsa_[PROJECT\x64\Debug\Dsa_[PROJECT.exe
Enter PostFix Expression Again: (4+6*)2
Value is: 24
Do you want to continue!!! (1/ 0)?1


C:\Users\M.HUZAIFA\source\repos\Dsa_[PROJECT\x64\Debug\Dsa_[PROJECT.exe
INFIX-POSTFIX CALCULATOR
1)INFIX TO POSTFIX
2)POSTFIX EVALUATION
3)EXIT
Enter your Choice here:
3

CHAPTER #05

5.1 Future Work:

Infix Postfix Notation helps student mostly and it will helps the user to write the infix notation and this will convert it into postfix notation and evaluate the value . It is beneficent because computer reads the postfix notation easily so in future it will helps us to convert the notations and evaluating the values.

5.2 Examples:

Here are two examples to help you understand how the algorithm works. Each line below demonstrates the state of the postfix string and the stack when the corresponding next infix symbol is scanned. The rightmost symbol of the stack is the top symbol. The rule number corresponding to each line demonstrates which of the six rules was used to reach the current state from that of the previous line.

Hence it is concluded that our project follows the stack data structure and by using stack we make infix postfix calculator which converts the infix notation into postfix notation and evaluates the value which is helpful in a way that computer understands the postfix notations easily. So, by this project we can apply the appropriate data structure and understands the behavior of stack and the proper use of infix postfix notations. This project might be helpful from future perspective.

Example 1
Input expression: A + B * C / D - E

Next Symbol	*Postfix String*	*Stack*	*Rule*
A	A		2
+	A	+	3
B	A B	+	2
*	A B	+ *	3
C	A B C	+ *	2
/	A B C *	+ /	3
D	A B C * D	+ /	2

-	A B C * D / +	-	3
E	A B C * D / + E	-	2
	A B C * D / + E -		6

Example 2
Input expression: (A + B * (C - D)) / E.

Next Symbol	*Postfix String*	*Stack*	*Rule*
(		(	4
A	A	(	2
+	A	(+	3
B	A B	(+	2
*	A B	(+ *	3
(	A B	(+ * (	4
C	A B C	(+ * (	2
-	A B C	(+ * (-	3
D	A B C D	(+ * (-	2
)	A B C D -	(+ *	5
)	A B C D - * +		5
/	A B C D - * +	/	3

CHAPTER #06

6.1 Conclusion:

Hence it is concluded that our project follows the stack data structure and by using stack we make infix postfix calculator which converts the infix notation into postfix notation and evaluates the value which is helpful in a way that computer understands the postfix notations easily. So, by this project we can apply the appropriate data structure and understands the behavior of stack and the proper use of infix postfix notations. This project might be helpful from future perspective.