**Experiment No 2: Conversion of Infix to postfix expression using stack ADT**

**Aim:** To convert infix expression to postfix expression using stack ADT

**Objective:**1) Understand the use of stack

2) Understand how to import an ADT in an application program

3) Understand the instantiation of stack ADT in an application program

4) Understand how the member function of an ADT are accessed in an application program

**Theory:** Infix to postfix conversion is a fundamental operation in computer science and is commonly used in expression evaluation and parsing. It involves converting an infix expression (where operators are placed between operands) into a postfix expression (also known as Reverse Polish Notation or RPN, where operators are placed after their operands).

**Infix:** Infix notation is a common way of writing mathematical expressions or operations where the operators are placed between the operands. For example, in the expression "2 + 3," the operator "+" is placed between the operands "2" and "3." This is the typical way we write mathematical expressions and equations.

**Postfix:** Postfix notation, also known as Reverse Polish Notation (RPN), is a mathematical notation in which operators are placed after their operands. In other words, in postfix notation, you write the expression by first listing the operands and then specifying the operation to be performed. For example, the infix expression "2 + 3" would be written as "2 3 +" in postfix notation. This notation eliminates the need for parentheses to indicate the order of operations and is often used in calculators and computer programs for efficient expression evaluation.

**Algorithm:**

1) Initialize an empty stack for operators and an empty list for the output postfix expression.

2)Scan the infix expression from left to right: a. If the token is an operand (number), add it to the output list. b. If the token is an operator (+, -, \*, /, etc.): i. Pop operators from the stack and add them to the output list while they have higher or equal precedence than the current operator, or have equal precedence and are left-associative. ii. Push the current operator onto the stack. c. If the token is an open parenthesis '(', push it onto the stack. d. If the token is a closing parenthesis ')': i. Pop operators from the stack and add them to the output list until an open parenthesis '(' is encountered (discard the '('). ii. Pop and discard the open parenthesis '(' from the stack.

3)After scanning the entire infix expression, pop any remaining operators from the stack and add them to the output list.

4)The output list now represents the postfix expression.

**Code:** #include<stdio.h>

#include<conio.h>

#include<ctype.h>

char stack[100];

int top = -1;

void push(char x)

{

stack[++top] = x;

}

char pop()

{

if(top == -1)

return -1;

else

return stack[top--];

}

int priority(char x)

{

if(x=='(')

return 0;

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

return 1;

if(x == '\*' || x == '/')

return 2;

return 0;

}

int main()

{

char exp[100];

char \*e, x;

printf("Enter the expression:");

scanf("%s",exp);

printf("\n");

e = exp;

while(\*e!='\0')

{

if(isalnum(\*e))

printf("%c",\*e);

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

push(\*e);

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

{

while((x = pop()) != '(')

printf("%c", x);

}

else

{

while(priority(stack[top]) >= priority(\*e))

printf("%c", pop());

push (\*e);

}

e++;

}

while(top != -1)

{

printf("%c", pop());

}

return 0;

}

**Output:**

**Conclusion:** In conclusion, the infix to postfix conversion algorithm showcases the versatility and power of stack data structures in solving problems related to expression manipulation. By following the algorithm's steps, we can convert complex infix expressions into a format that is easier to evaluate and process, contributing to various applications in compilers, calculators, and more.