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 functions of an ADT are accessed in an application program.

Theory:

Postfix notation is a way of representing algebraic expressions without parentheses or operator precedence rules. In this notation, expressions are evaluated by scanning them from left to right and using a stack to perform the calculations. When an operand is encountered, it is pushed onto the stack, and when an operator is encountered, the last two operands from the stack are popped and used in the operation, with the result then pushed back onto the stack. This process continues until the entire postfix expression is parsed, and the result remains in the stack.

Conversion of infix to postfix expression

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

Algorithm:

Conversion of infix to postfix

Step 1: Add ")" to the end of the infix expression

Step 2: Push "(" on to the stack

Step 3: Repeat until each character in the infix notation is scanned

IF a "(" is encountered,push it on the stack
IF an operand (whether a digit or a character) is encountered, add it to the postfix expression.
IF a ")" is encountered, then
a. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
b.Discard the "(". That is, remove the "(" from stack and do not add it to the postfix expression
IF an operator 0 is encountered, then
a. Repeatedly pop from stack and add each operator (popped from the stack) to t postfix expression which has the same precedence or a higher precedence than o
b. Push the operator o to the stack [END OF IF]

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5: EXIT

Code:

```
#include<stdio.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:

Enter the expression : A+B-(c*D)/E

A B + c D * E / -

Conclusion:

Convert the following infix expression to postfix (A+(C/D))*B

Current Character	Stack	Result
(	(	
A	(	A
+	(+	A
(	(+(	A
C	(+(	AC
/	(+(/	AC
D	(+(/	ACD
)	(+	ACD/
*	(*)	ACD/+
B	(*)	ACD/+B

End Expression :
ACD/+

How many push and pop operations were required for the above conversion?

- To convert the infix expression to postfix it will require a total of **8 push** and **7 pop** operations.

Where is the infix to postfix conversion used or applied?

The infix to postfix conversion is used or applied in various situations, such as:

- Evaluating expressions: The postfix expressions can be evaluated easily using a stack, unlike the infix expressions that require scanning and parsing. The postfix expressions also eliminate the need for parentheses and precedence rules, which can make the infix expressions more complex and ambiguous.
- Compiling programs: The compilers often convert the infix expressions in the source code to postfix expressions in the intermediate code, which can then be translated to machine code. The postfix expressions are more suitable for generating code for stack-based machines.
- Implementing algorithms: Some algorithms, such as the Shunting-yard algorithm, use the infix to postfix conversion to process mathematical expressions and produce output in a desired notation. The postfix notation is also useful for implementing recursive functions, reverse Polish notation, and some data structures.