DS-LAB EXPERIMENT 2

NAME: YASH SARANG D6AD/47

Aim: To implement an application of stack using Array.
i. Parenthesis Matching.
ii. Infix to Postfix using Stack.
iii. Infix to Prefix using Stack.

Theory: Stack is a linear data structure that follows a particular order in which the operations are performed.
Few of its usages are given below:

1. Expression Conversion- a. Infix to Postfix
   b. Infix to Prefix
   c. Postfix to infix
   d. Prefix to infix
2. Expression Evaluation
3. Parsing
4. Simulation of Recursion
5. Function Call

There are 3 popular methods for the representation of expressions.

a. Infix x+y (operator between operands)
b. Prefix +xy (operator before operands)
c. Prefix xy+ (operator after operands)

Infix expressions are not used inside a computer, due to the additional complexity of handling precedence.
Prefix and Postfix expressions are free from any precedence.
They are more suited for mechanization.
The computer uses the postfix form for the representation of an expression.

---

1. PARENTHESIS MATCHING

Algorithms: s-stack of characters
x-character type

Step 1: START

Step 2: x reads the next token

Step 3: if (x==’(‘) Push(s,x);

Step 4: if (x==’)‘) if(top element of stack is ‘(‘)
pop(s);

else print “Mismatch”

Step 5: if (more tokens) - go to - step 1

Step 6: if (stack is not empty) Print "Mismatch"
else Print " Proper Paranthesis”

Step 7: Stop.

OUTPUT:

```
Enter an equation:
    a-b)+c*(d
    The Parenthesis are 'Miss Matched'.
```

```
Enter an equation:
    ((a+b)-c)*d*(b-a)
    The Parenthesis are correctly Matched.
```

PROGRAM CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 30

char stack[MAX_SIZE];
char str[MAX_SIZE];
int top = -1;

void push()
{
    if(top==MAX_SIZE-1)
    {
        printf("\n** Stack OVERFLOW !!! ***");
    }
    else
        stack[++top]='(';
}
void pop()
{
    if(top>-1)
    {
        top--;
    }
    else
        return;
}
int main()
{
    int i;
    printf("\nEnter an equation:\n\t");
    gets(str);
    for(i=0; i<strlen(str);i++)
    {
        if(str[i]== '(')
            push();
        else if(str[i]==')')
            pop();
    }
    if(top==-1)
        printf("\nThe Parenthesis are correctly Matched.\n");
    else
        printf("\nThe Parenthesis are 'Miss Matched'.\n");

    return 0;
}
```

2. Infix to Postfix using Stack

Algorithm:

Step 1: START

Step 2: Push "("onto Stack, and add ")" to the end of X.

Step 3: Scan X from left to right. Repeat 3-6 for each element of X until the stack is empty.

Step 4: If an operand is encountered, add it to Y.

Step 5: If (opening parenthesis is encountered) push it into Stack.

Step 6: If (operator is encountered),
repeatedly pop from Stack and add each operator to Y (on the top of stack)
which has the same precedence as or higher precedence than the operator.
Add the operator to thestack.

Step 7: If (closing parenthesis is encountered),
repeatedly pop from Stack and add each operator to Y (on the Stack)
until a left parenthesis is encountered.
Remove the left parenthesis.

Step 8: Stop

OUTPUT:

```
Enter Infix expression : a*b/c*d-e/f
Postfix Expression: ab*c/d*ef/-
```

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>

#define SIZE 100

char stack[SIZE];
int top = -1;

void push(char item)
{
        if(top >= SIZE-1)
        {
                printf("\nStack Overflow.");
        }
        else
        {
                top = top+1;
                stack[top] = item;
        }
}

char pop()
{
        char item ;

        if(top <0)
        {
                printf("stack under flow: invalid infix expression");
                getchar();
                exit(1);
        }
        else
        {
                item = stack[top];
                top = top-1;
                return(item);
        }
}

int is_operator(char symbol)
{
```

```
        if(symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' ||
symbol =='-')
        {
                return 1;
        }
        else
        {
        return 0;
        }
}

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

void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
        int i, j;
        char item;
        char x;

        push('(');
        strcat(infix_exp,")");

        i=0;
        j=0;
        item=infix_exp[i];

        while(item != '\0')
```

```
    {
        if(item == '(')
        {
            push(item);
        }
        else if( isdigit(item) || isalpha(item))
        {
            postfix_exp[j] = item;
            j++;
        }
        else if(is_operator(item) == 1)
        {
            x=pop();
            while(is_operator(x) == 1 && precedence(x)>=
precedence(item))
            {
                postfix_exp[j] = x;
                j++;
                x = pop();
            }
            push(x);
            push(item);
        }
        else if(item == ')')
        {
            x = pop();
            while(x != '(')
            {
                postfix_exp[j] = x;
                j++;
                x = pop();
            }
        }
        else
        {
            printf("\nInvalid infix Expression.\n");
            getchar();
            exit(1);
        }
        i++;
        item = infix_exp[i];
    }
    if(top>0)
    {
        printf("\nInvalid infix Expression.\n");
```

```
                getchar();
                exit(1);
        }
        if(top>0)
        {
                printf("\nInvalid infix Expression.\n");
                getchar();
                exit(1);
        }
        postfix_exp[j] = '\0';
}

int main()
{
        char infix[SIZE], postfix[SIZE];
        printf("\nEnter Infix expression : ");
        gets(infix);

        InfixToPostfix(infix,postfix);

        printf("Postfix Expression: ");
        puts(postfix);

        return 0;
}
```

---

3. Infix to Prefix using Stack

Algorithm

Step 1: START

Step 2: Push ")" onto the stack, and add "(" to the end of A.

Step 3: Scan A from right to left and repeat steps 3 to 6 for each element of A until the Stack is empty.

Step 4: If an operand is encountered add it to B.

Step 5. If a right parenthesis is encountered, push it onto the Stack.

Step 6: If an operand is encountered then:
Repeatedly pop from Stack and add to B each operator
(on the top of STACK) which has the same on higher precedence than the operator.
Add operator to STACK.

Step 7: If left parenthesis is encountered then:
Repeatedly pop from Stack and add to B (each operator on top of Stack until a left parenthesis is encountered)
Remove the left parenthesis

Step 8: Stop

PROGRAM CODE:

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>

# define MAX 20

void infixtoprefix(char infix[20],char prefix[20]);
void reverse(char array[30]);
char pop();
void push(char symbol);
int isOperator(char symbol);
int prcd(symbol);
int top=-1;
char stack[MAX];
void main()
{
        char infix[20],prefix[20],temp;
        printf("Enter infix operation: ");
        gets(infix);
        infixtoprefix(infix,prefix);
        reverse(prefix);
```

```
        printf("Prefix operation is: ");
        puts((prefix));
}

void infixtoprefix(char infix[20],char prefix[20]) {
        int i,j=0;
        char symbol;
        stack[++top]='#';
        reverse(infix);
        for (i=0;i<strlen(infix);i++) {
                symbol=infix[i];
                if (isOperator(symbol)==0) {
                        prefix[j]=symbol;
                        j++;
                } else {
                        if (symbol==')') {
                                push(symbol);
                        } else if(symbol == '(') {
                                while (stack[top]!=')') {
                                        prefix[j]=pop();
                                        j++;
                                }
                                pop();
                        } else {
                                if (prcd(stack[top])<=prcd(symbol)) {
                                        push(symbol);
                                } else {
                                        while(prcd(stack[top])>=prcd(symbol)) {
                                                prefix[j]=pop();
                                                j++;
                                        }
                                        push(symbol);
                                }
                        }
                }
        }
        while (stack[top]!='#') {
                prefix[j]=pop();
                j++;
        }
        prefix[j]='\0';
}

void reverse(char array[30])
{
```

```
        int i,j;
        char temp[100];
        for (i=strlen(array)-1,j=0;i+1!=0;--i,++j) {
                temp[j]=array[i];
        }
        temp[j]='\0';
        strcpy(array,temp);
        return array;
}

char pop() {
        char a;
        a=stack[top];
        top--;
        return a;
}

void push(char symbol) {
        top++;
        stack[top]=symbol;
}

int prcd(symbol)
{
        switch(symbol) {
                case '+':
                        case '-':
                        return 2;
                break;
                case '*':
                        case '/':
                        return 4;
                break;
                case '$':
                        case '^':
                        return 6;
                break;
                case '#':
                        case '(':
                        case ')':
                        return 1;
                break;
        }
}
```

```
int isOperator(char symbol) {
        switch(symbol) {
                case '+':
                        case '-':
                        case '*':
                        case '/':
                        case '^':
                        case '$':
                        case '&':
                        case '(':
                        case ')':
                        return 1;
                break;
                default:
                        return 0;
        }
}
```

OUTPUT:

```
Enter infix operation: a*b/c*d-e/f
Prefix operation is: -*/*abcd/ef
```

• Conclusion:

We have learned important elements and applications of Stack using Arrays. We have studied and implemented Parenthesis Matching and Expression Conversion mainly Infix to Postfix and Infix to Prefix.
