**LAB SESSION 2: STACK DATA STRUCTURE**

Aim: To implement expression converter and evaluator using stack data structure

Problem definition:

1. Develop a C program to implement the following

* Convert infix to postfix
* Convert infix to prefix
* Evaluate a postfix expression
* Evaluate a prefix expression

**THEORY**:

A stack is a data structure that follows the LIFO(Last in first out) principle. In this data structure we have a TOP pointer which refers to the element at top of the stack. To retrieve or delete data from the stack we have only one position that is the TOP position.

There are some basic operations that allow us to perform different actions on a stack

* Push:Insert an element to the top of the stack
* Pop: Delete an element from top of a stack
* IsEmpty: Check if the stack is empty
* IsFull : Check if the stack is full
* Peek: Get the value of the top element without removing it

**Working of stack data structure**

The operations work as follows:

* A pointer/location called TOP is used to keep track of the top element in the stack
* When initializing the stack we set TOP value to -1 so that we can check if the stack is empty by comparing TOP==-1( for array implementation or TOP=NULL for LL implementation)
* On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
* On popping an element, we return the element pointed to by TOP and reduce its value.
* Before pushing we check if the stack is already full
* Before popping we check if the stack is already empty

**Algorithms**

1. **Convert infix to postfix**

Initialize variables infix\_symbol, stack\_symbol, i, and j to 0.

Set infix\_symbol as the first character of the infix string.

While infix\_symbol is not a whitespace character:

Switch on infix\_symbol:

If infix\_symbol is one of the operators +, -, \*, /, %, or ^:

While the stack is not empty and the precedence of the top operator in the stack is greater than or equal to the precedence of infix\_symbol, pop the operator from the stack and append it to the postfix string.

Push infix\_symbol onto the stack.

If infix\_symbol is an open parenthesis (, push it onto the stack.

If infix\_symbol is a closing parenthesis ), pop operators from the stack and append them to the postfix string until an open parenthesis is encountered.

If infix\_symbol is neither an operator nor a parenthesis

If infix\_symbol is an alphabet character, append it to the postfix string.

Otherwise, print an error message and exit the program.

Print the current state of the infix and postfix expressions.

Update infix\_symbol to the next character in the infix string.

End of Loop

While the stack is not empty, pop operators from the stack and append them to the postfix string.

Print the final state of the infix and postfix expressions.

Add a null terminator at the end of the postfix string.

1. **Convert infix to prefix**

Initialize variables infix\_symbol, stack\_symbol, i, and j to 0.

Set i to the length of the infix string.

Set infix\_symbol as the character at index i in infix and decrement i.

While infix\_symbol is not a whitespace character:

Switch on infix\_symbol

If infix\_symbol is one of the operators +, -, \*, /, %, or ^:

While the stack is not empty and the precedence of the top operator in the stack is greater than or equal to the precedence of infix\_symbol, pop the operator from the stack and append it to the prefix string.

Push infix\_symbol onto the stack.

If infix\_symbol is a closing parenthesis ), push it onto the stack.

If infix\_symbol is an open parenthesis (:

While the stack is not empty and the top of the stack is not a closing parenthesis, pop operators from the stack into stack\_symbol and append them to the prefix string.

Pop the closing parenthesis.

If infix\_symbol is an alphabet character, append it to the prefix string.

Otherwise, print an error message and exit the program.

Print the current state of the infix and prefix expressions.

Decrement i to get the previous character in the infix string in infix\_symbol.

End of Loop:

Repeat until the entire infix expression is processed.

While the stack is not empty, pop operators from the stack and append them to the prefix string.

Print the final state of the infix and prefix expressions.

Add a null terminator at the end of the prefix string.

1. **Evaluate a postfix expression**

Initialize variables i, pos, and ps to 0.

Set top (stack top) to -1.

Set ps as the first character of the postfix string.

While ps is not a whitespace character:

Check if ps is an Operator:

If ps is an operator (+, -, \*, /, %, ^):

Pop two operands a and b from the stack.

Perform the corresponding operation based on ps and push the result back onto the stack.

If ps is an alphabet character:

Call the search() function to find the position of ps in the variables array.

If pos is -1, it means the variable hasn't been encountered before. Add it to variables with a user-defined value.

If ps is a Variable:

Push the value of the variable onto the stack.

Update ps to the next character in the postfix string.

Repeat until the entire postfix expression is processed.

Pop the final result from the stack and return it.

1. **Evaluate a prefix expression**

Initialize variables i, pos, and ps to 0.

Set top (stack top) to -1.

Set ps as the first character of the prefix string.

While ps is not a whitespace character:

Check if ps is an Operator:

If ps is an operator (+, -, \*, /, %, ^):

Pop two operands a and b from the stack.

Perform the corresponding operation based on ps and push the result back onto the stack.

Check if ps is an Alphabet Character:

If ps is an alphabet character:

Call the search() function to find the position of ps in the variables array.

If pos is -1, it means the variable hasn't been encountered before. Add it to variables with a user-defined value.

If ps is a Variable:

Push the value of the variable onto the stack.

Update ps to the next character in the prefix string.

End of Loop:

Repeat until the entire prefix expression is processed.

Pop the final result from the stack and return it.

**Program for infix to postfix**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<ctype.h>

#include<math.h>

#define BLANK ' '

#define TAB '\t'

#define MAX 50

// structure top store value of variables of postfix equation

struct variable{

long int value;

char c;

}variables[MAX];

long int stack[MAX];

char infix[MAX];

char postfix[MAX];

int top = -1, v\_top =-1;

int is\_operator(char symbol){

if(symbol == '+' || symbol == '-' || symbol == '/' || symbol == '%' || symbol == '\*' || symbol == '^')

return 1;

else return 0;

}

void push(long int symbol){

if(top == MAX-1){

printf("Stack OverFlow\n");

return;

}

stack[++top] = symbol;

}

long int pop(){

if(top == -1){

printf("Stack Underflow\n");

exit(1);

}

return(stack[top--]);

}

int instack\_priority(char symbol){

switch (symbol){

case '(': return 0;

case '+' :case '-' : return 1;

case '/' :case '%' :case '\*' : return 2;

case '^' : return 3;

default : printf("Invalid operator");

}

}

int incoming\_priority(char symbol){

switch (symbol){

case '(': return 0;

case '+' :case '-' : return 1;

case '/' :case '%' :case '\*' : return 2;

case '^' : return 4;

default : printf("Invalid operator");

}

}

int whitespace(char symbol){

if(symbol == BLANK || symbol == TAB || symbol == '\0')

return 1;

else return 0;

}

void print\_solution(int i, int j){

printf("%c",infix[i]); printf("\t");

for(int x=0; x<=top; x++)

printf("%c",stack[x]);printf("\t");

for(int x=0; x<j; x++)

printf("%c",postfix[x]);

printf("\n");

}

void infix\_topostfix()

{

char infix\_symbol, stack\_symbol;

int i = 0, j = 0;

infix\_symbol = infix[i];

while(!whitespace(infix\_symbol))

{

switch (infix\_symbol){

case '+':case'-':case'\*':case'/':case'%':case'^':

while(top != -1 && (instack\_priority(stack[top]) >= incoming\_priority(infix\_symbol))){

postfix[j++] = pop();

}

push(infix\_symbol);

break;

case '(' : push(infix\_symbol); break;

case ')' :

while (top != -1 && ((stack\_symbol = pop()) != '('))

postfix[j++] = stack\_symbol;

break;

default :

if(isalpha(infix\_symbol))

postfix[j++] = infix\_symbol;

else {printf("Invalid Infix Character Encountered! "); exit(1);}

break;

}

print\_solution(i,j);

infix\_symbol = infix[++i];

}

while(top != -1)

postfix[j++] = pop();

print\_solution(i,j);

postfix[j] = '\0';

}

// serches the struct variables array to find if the passed character (which denotes to variable) is present

// and return its position in the array

int search(char c){

int i=0;

while(i<=v\_top){

if(c == variables[i].c)

return i;

i++;

}

return -1;

}

// Scans all possible characters or variables in the postfix and passes as them to struct variables[]

// data member char c, this is basically the identefier of the variable

// it then takes in the value for the variable and stores in struct variables[]

void variable\_value(){

int i=0, pos;

char ps = postfix[i];

while(!whitespace(ps)){

if(isalpha(ps)){

pos = search(ps);

if(pos == -1){

variables[++v\_top].c = ps;

printf("\nEnter value of %c : ",ps);

scanf("%d",&variables[v\_top].value);

}

}

ps = postfix[++i];

}

}

// This function with the help of the above 2 functions gives the solution the postfix equation

long int evaluate\_potfix(){

int i = 0, pos;

top = -1;

char ps = postfix[i];

while(!whitespace(ps)){

if(is\_operator(ps)){

int a, b;

b = pop();

a = pop();

switch (ps){

case '+': push(a + b); break;

case '-': push(a - b); break;

case '\*': push(a \* b); break;

case '/': push(a / b); break;

case '%': push(a % b); break;

case '^': push(pow(a,b)); break;

}

}

if(isalpha(ps)){

pos = search(ps);

stack[++top] = variables[pos].value;

}

ps = postfix[++i];

}

return pop();

}

int main()

{

int n;

char item;

printf("Enter Infix Expression: "); gets(infix);

infix\_topostfix();

printf("\nEnter values of variables\n");

variable\_value();

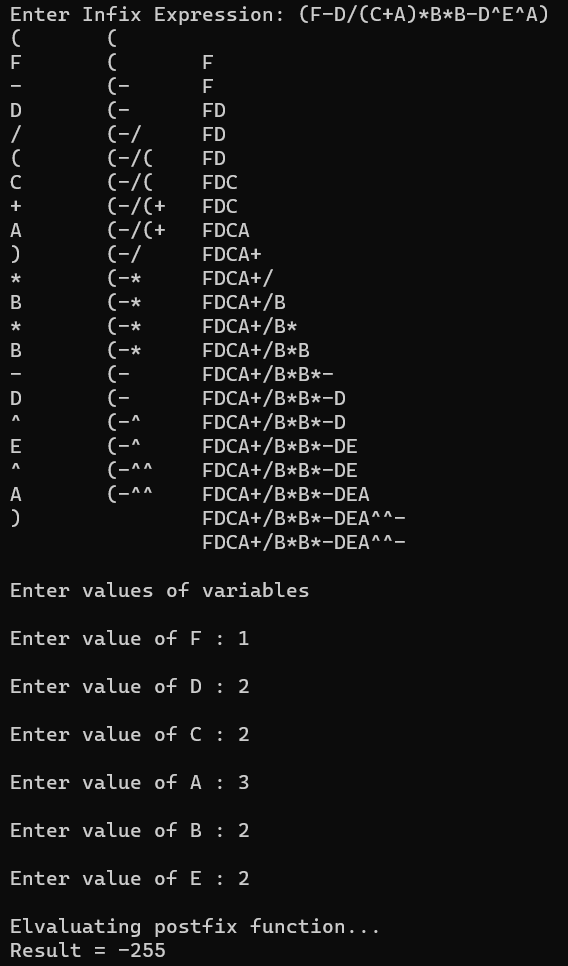
long int result = evaluate\_potfix();

printf("\nElvaluating postfix function...\n");

printf("Result = %d",result);

return 0;

}



**Program for infix to prefix**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<ctype.h>

#include<math.h>

#define BLANK ' '

#define TAB '\t'

#define MAX 50

struct variable{

long int value;

char c;

}variables[MAX];

long int stack[MAX];

char infix[MAX];

char prefix[MAX];

int top = -1, v\_top =-1;

int is\_operator(char symbol){

if(symbol == '+' || symbol == '-' || symbol == '/' || symbol == '%' || symbol == '\*' || symbol == '^')

return 1;

else return 0;

}

void push(long int symbol){

if(top == MAX-1){

printf("Stack OverFlow\n");

return;

}

stack[++top] = symbol;

}

long int pop(){

if(top == -1){

printf("Stack Underflow\n");

exit(1);

}

return(stack[top--]);

}

int instack\_priority(char symbol){

switch (symbol){

case ')': return 0;

case '+' :case '-' : return 1;

case '/' :case '%' :case '\*' : return 2;

case '^' : return 4;

}

}

int incoming\_priority(char symbol){

switch (symbol){

case ')': return 0;

case '+' :case '-' : return 1;

case '/' :case '%' :case '\*' : return 2;

case '^' : return 3;

}

}

int whitespace(char symbol){

if(symbol == BLANK || symbol == TAB || symbol == '\0')

return 1;

else return 0;

}

void print\_solution(int i, int j){

printf("%c",infix[i]); printf("\t");

for(int x=0; x<=top; x++)

printf("%c",stack[x]);printf("\t");

for(int x=0; x<j; x++)

printf("%c",prefix[x]);

printf("\n");

}

void infix\_toprefix(){

char infix\_symbol, stack\_symbol;

int i, j = 0;

i = strlen(infix);

infix\_symbol = infix[--i];

while(!whitespace(infix\_symbol)){

switch (infix\_symbol){

case '+':case'-':case'\*':case'/':case'%':case'^':

while(top != -1 && (instack\_priority(stack[top]) >= incoming\_priority(infix\_symbol))){

prefix[j++] = pop();

}

push(infix\_symbol);

break;

case ')' : push(infix\_symbol); break;

case '(' :

while (top != -1 && ((stack\_symbol = pop()) != ')'))

prefix[j++] = stack\_symbol;

break;

default :

if(isalpha(infix\_symbol))

prefix[j++] = infix\_symbol;

else {printf("Invalid Infix Character Encountered! "); exit(1);}

break;

}

print\_solution(i,j);

infix\_symbol = infix[--i];

}

while(top != -1)

prefix[j++] = pop();

print\_solution(i,j);

prefix[j] = '\0';

}

int search(char c){

int i=0;

while(i<=v\_top){

if(c == variables[i].c)

return i;

i++;

}

return -1;

}

void variable\_value(){

int i=0, pos;

char ps = prefix[i];

while(!whitespace(ps)){

if(isalpha(ps)){

pos = search(ps);

if(pos == -1){

variables[++v\_top].c = ps;

printf("\nEnter value of %c : ",ps);

scanf("%d",&variables[v\_top].value);

}

}

ps = prefix[++i];

}

}

long int evaluate\_prefix(){

int i = 0, pos;

top = -1;

char ps = prefix[i];

while(!whitespace(ps)){

if(is\_operator(ps)){

int a, b;

b = pop();

a = pop();

switch (ps){

case '+': push(a + b); break;

case '-': push(a - b); break;

case '\*': push(a \* b); break;

case '/': push(a / b); break;

case '%': push(a % b); break;

case '^': push(pow(a,b)); break;

}

}

if(isalpha(ps)){

pos = search(ps);

stack[++top] = variables[pos].value;

}

ps = prefix[++i];

}

return pop();

}

int main()

{

int n;

char item;

printf("Enter Infix Expression: "); gets(infix);

infix\_toprefix();

printf("Prefix Equation: ");

for(int i = sizeof(prefix); i>=0; i--)

printf("%c",prefix[i]);

printf("\n\nEnter values of variables\n");

variable\_value();

long int result = evaluate\_prefix();

printf("\nElvaluating prefix function...\n");

printf("Result = %d",result);

return 0;

}

