2. Stack

TITLE:

Write a program to implement stack as an abstract data type using linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix/prefix expression.

OBJECTIVE:

1) To understand the concept of abstract data type.
2) How different data structures such as arrays and a stacks are represented as an ADT.

THEORY:

- **Abstract Data Type (ADT):**

An Abstract Data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles. In general terms, an abstract data type is a *specification* of the values and the operations that has two properties:

- It specifies everything you need to know in order to use the data type
- It makes absolutely no reference to the manner in which the data type will be implemented.

When we use abstract data types, our programs divide into two pieces:

- The Application: The part that uses the abstract data type.
- The Implementation: The part that implements the abstract data type.

- **Stack and Stack operations:**

In computer science, a **stack** is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds an item to the top of the stack, hiding any items already on the stack, or initializing the stack if it is empty. A pop either reveals previously concealed items, or results in an empty stack. A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also mean that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest. A collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop. Often top and isEmpty are available, too. Also known as "last-in, first-out" or LIFO.

Stack Operations:

An abstract data type (ADT) consists of a data structure and a set of **primitive**

- **Push** adds a new element
- **Pop** removes a element

Additional primitives can be defined:

- **IsEmpty**reports whether the stack is empty
- **IsFull**reports whether the stack is full
- **Initialise**creates/initializes the stack
- **Destroy** deletes the contents of the stack (may be implemented by re-initializingthe stack)

❖ **Stack can be implemented as an ADT**

User can Add, Delete, Search, and Replace the elements from the stack. It also checks for Overflow/Underflow and returns user friendly errors. You can use this stack implementation to perform many useful functions. In graphical mode, this C program displays a startup message and a nice graphic to welcome the user.

The program performs the following functions and operations:

- **Push**: Pushes an element to the stack. It takes an integer element as argument. Ifthe stack is full then error is returned.
- **Pop**: Pop an element from the stack. If the stack is empty then error is returned.The element is deleted from the top of the stack.
- **DisplayTop**: Returns the top element on the stack without deleting. If the stack isempty then error is returned.

❖ **ALGORITHM:**

Abstract Data Type Stack:

Define Structure for stack(Data, Next Pointer)

Stack Empty:

Return True if Stack Empty else False.
Top is a pointer of type structure stack.
Empty(Top)
Step 1: If Top = = NULL
Step 2: Return 1;
Step 3: Return 0;

Push Opertion:

Top & Node pointer of structure Stack.
Push(element)
Step 1: Node->data=element;
Step 2; Node->Next = Top;
Step 3: Top = Node

Step 4: Stop.

Pop Operation:

Top & Temp pointer of structure Stack.
Pop()
Step 1:If Top != NULL
Then
i) Temp = Top;
ii) element=Temp->data;
iiI) Top = (Top)->Next;
iv) delete temp;
Step 2: Else Stack is Empty.
Step 3: return element;

Infix to Prefix Conversion:

String is an array of type character which store infix expression.
This function return Prefix expression.
InfixToPrefix(String)
Step 1: I = strlen(String);
Step 2: Decrement I;
Step 3: Do Steps 4 to 9 while I>= 0
Step 4: If isalnum(String[I] Then PreExpression[J++] = String[I];
Step 5: Else If String[I]=='('
Then a) Temp=Pop(&Top);//Pop Operator from stack
b) Do Steps while Temp->Operator!=')' and Temp!=NULL
i) PreExpression[J++] = Temp->Operator;
ii) Temp = Pop(&Top);}

Step 6: Else If String[I]==')' Then Push(&Top,Node);//push _)‘ in a stack
Step 7: Else

a) Temp = Pop(&Top);//Pop operator from stack
b) DO Steps while Temp->Operator != ')' And Temp != NULL
And Priority(String[I])<Priority(Temp->Operator)
i) PreExpression[J++] = Temp->Operator;
ii) Temp = Pop(&Top);}
c) If Temp!=NULL And Temp->Operator==')'
Or Priority(String[I])>=Priority(Temp->Operator))
Then Push(&Top,Temp);
Step 8: Push(&Top,Node);//Push String[I] in a stack;

Step 9: Decrement I;

Step 10: Temp = Pop(&Top);//pop remaining operators from stack
Step 11: Do Steps while Temp != NULL //Stack is not Empty
i) PreExpression[J++] = Temp->Operator;
ii) Temp = Pop(&Top);}
Step 12: PreExpression[J] = NULL;

Step 13: Reverse PreExpression;
Step 14: Return PreExpression.

Infix to Postfix Conversion:

String is an array of type character which store infix expression.
This function return Postfix expression.
InfixToPostfix(String)
Step 1: I = 0;
Step 2: Do Steps 3 to 8 while String[I] != NULL
Step 3: If isalnum(String[I] Then PostExpression[J++] = String[I];
Step 4: Else If String[I]==')'
Then a) Temp=Pop(&Top);//Pop Operator from stack
b) Do Steps while Temp->Operator!='(' and Temp!=NULL
i) PostExpression[J++] = Temp->Operator;
ii) Temp = Pop(&Top);}

Step 5: Else If String[I]=='(' Then Push(&Top,Node);//push _)‘ in a stack
Step 6: Else
a) Temp = Pop(&Top);//Pop operator from stack
b) DO Steps while Temp->Operator != '(' And Temp != NULL And Priority(String[I]) >= Priority(Temp->Operator)
iii) PreExpression[J++] = Temp->Operator;
iv) Temp = Pop(&Top);}
c) If Temp!=NULL And Temp->Operator=='('

iii) PostExpres88(&Top);}
Step 11: PostExpression[J] = NULL;
Step 12: Return PostExpression.

PostFix Expression Evaluation:

String is an array of type character which store infix expression.
Postfix_Evaluation(String)
Step 1: Do Steps 2 & 3 while String[I] != NULL
Step 2: If String[I] is operand
Then Push it into Stack
Step 3: If it is an operator Then
Pop two operands from Stack;
Stack Top is 1st Operand & Top-1 is 2nd Operand;
Perform Operation;
Step 4: Return Result.

INPUT:

Test Case	O/P
If Stack Empty	Display message —"Stack Empty"
If Stack Full	Display message —"Stack Full"

INPUT:	POSTFIX OUTPUT:
(A+B) * (C-D)	AB+CD-*
A$B*C-D+E/F/(G+H)	AB$C*D-EF/GH+/+
((A+B)*C-(D-E))$(F+G)	AB+C*DE—FG+$
A-B/(C*D$E)	ABCDE$*/-
A^B^C	ABC^^

INPUT:	PREFIX OUTPUT:
(A+B) * (C-D)	*+AB-CD
A$B*C-D+E/F/(G+H)	+-*$ABCD//EF+GH
((A+B)*C-(D-E))$(F+G)	$-*+ABC-DE+FG
A-B/(C*D$E)	-A/B*C$DE
A^B^C	^A^BC

NOTE: Here, $ AND ^ are used as raised to operator.

CONCLUSION: Thus, we have studied stack implementation as an ADT and use same ADT for conversion of infix expression to postfix and prefix expression and evaluation of prefix, postfix expression.

FAQ:

1. What is data structure?
2. Types of data structure?
3. Examples of linear data structure & Non-linear data structure?
4. What are the operations can implement on stack?
5. What are the applications of stack?
6. Convert the expression ((A + B) * C - (D - E) ^ (F + G)) to equivalent Prefix and Postfix notations.