Name-Harshita Totala Roll No- SEITA14

Div-A

ASSIGNMENT-2
Stack

AIM: Implement stack as an abstract data type using singly linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix and 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:

1) What is an abstract data type?

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.

2) What is stack? Explain stack operations with neat diagrams.

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.

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/initialises the stack
- **Destroy** deletes the contents of the stack (may be implemented by re-initialising
- the stack)

3) Explain how 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. If the 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 is empty then error is returned.

4) With an example explain how an infix expression can be converted to prefix and postfix form with the stack. (Ans to be written by students)

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 3: Stop.
```

Pop Operation:

```
Top & Temp pointer of structure
Stack. Pop()
Step 1:If Top
    !=
    NULL
    Then
      i) Temp = Top;
      ii) element=Te
     mp->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(S tring) 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.

```
InfixToPost fix(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=='('
                      Or Priority(String[I]) < Priority(Temp-
               >Operator)) Then Push(&Top,Temp); Step
      7: Push(&Top,Node);//Push String[I] in
      a stack; Step 8: Increment I;
      Step 9: Temp = Pop(&Top);//pop remaining operators from stack
      Step 10: Do Steps while Temp != NULL //Stack is not
      Empty
                iii)
                         PostExpres8
      8(&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‖
Stack Empty	Display message ―Stack Full‖

Name-Harshita Totala Roll No- SEITA14

Div-A

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

Name-Harshita Totala Roll No- SEITA14

Div-A

Program-

```
#include<iostream>
using namespace std;

typedef struct node
{
char data;
struct node *next;
}n;                             //n array of structure node

class stack
{
n *top;
public:
stack()                        //constructor of class stack
{
top=NULL;
}

int isempty()                  //check whether stack is empty or not.
{
if(top==NULL)
return 1;
else
return 0;
}

void push(char x)
{
n *p;
p=new node();
p->data=x;
p->next=top;
top=p;
}

char pop()
```

```
{
node *p;
char x;
p=top;
x=p->data;
top=top->next;
delete(p);
return x;
}

char topdata()
{
return top->data;
}
};

void infix_postfix(char infix[20],char postfix[20]);
int evaluate(int op1,int op2,char op);
void evaluate_postfix(char postfix[20]);
int precedence(char x);
void reverse(char a[20],char b[20]);
void infix_prefix(char infix[20],char prefix[20]);
void evaluate_prefix(char prefix[20]);
void infix_prefix_1(char infix[20],char prefix[20]);

int main()
{
char infix[20],token,postfix[20],prefix[20];
int ch,result;
do
{
cout<<"\n1. Infix to Postfix Expression";
cout<<"\n2. Evaluate Postfix";
cout<<"\n3. Infix to Prefix Expression";
cout<<"\n4. Evalute Prefix";
cout<<"\n5. Exit";
cout<<"\nEnter your choice: ";
cin>>ch;
switch(ch)
```

Name-Harshita Totala Roll No- SEITA14

Div-A

```
{

case 1:
cout<<"\nEnter Infix expression:  ";
cin>>infix;
infix_postfix(infix,postfix);
cout<<"\n------------------------------";
cout<<"\nPOSTFIX EXPRESSION- "<<postfix<<endl;
cout<<"------------------------------"<<endl;
break;

case 2:
evaluate_postfix(postfix);
break;

case 3:
cout<<"\nEnter Infix expression:  ";
cin>>infix;
infix_prefix(infix,prefix);
cout<<"\n------------------------------";
cout<<"\nPREFIX EXPRESSION- "<<prefix<<endl;
cout<<"------------------------------"<<endl;
break;

case 4:
evaluate_prefix(prefix);
break;

case 5:
cout<<"\nProgram Exited! ";
break;

default:
cout<<"WRONG CHOICE! Enter again...";
}

}while(ch!=5);
}
```

Name-Harshita Totala Roll No- SEITA14

Div-A

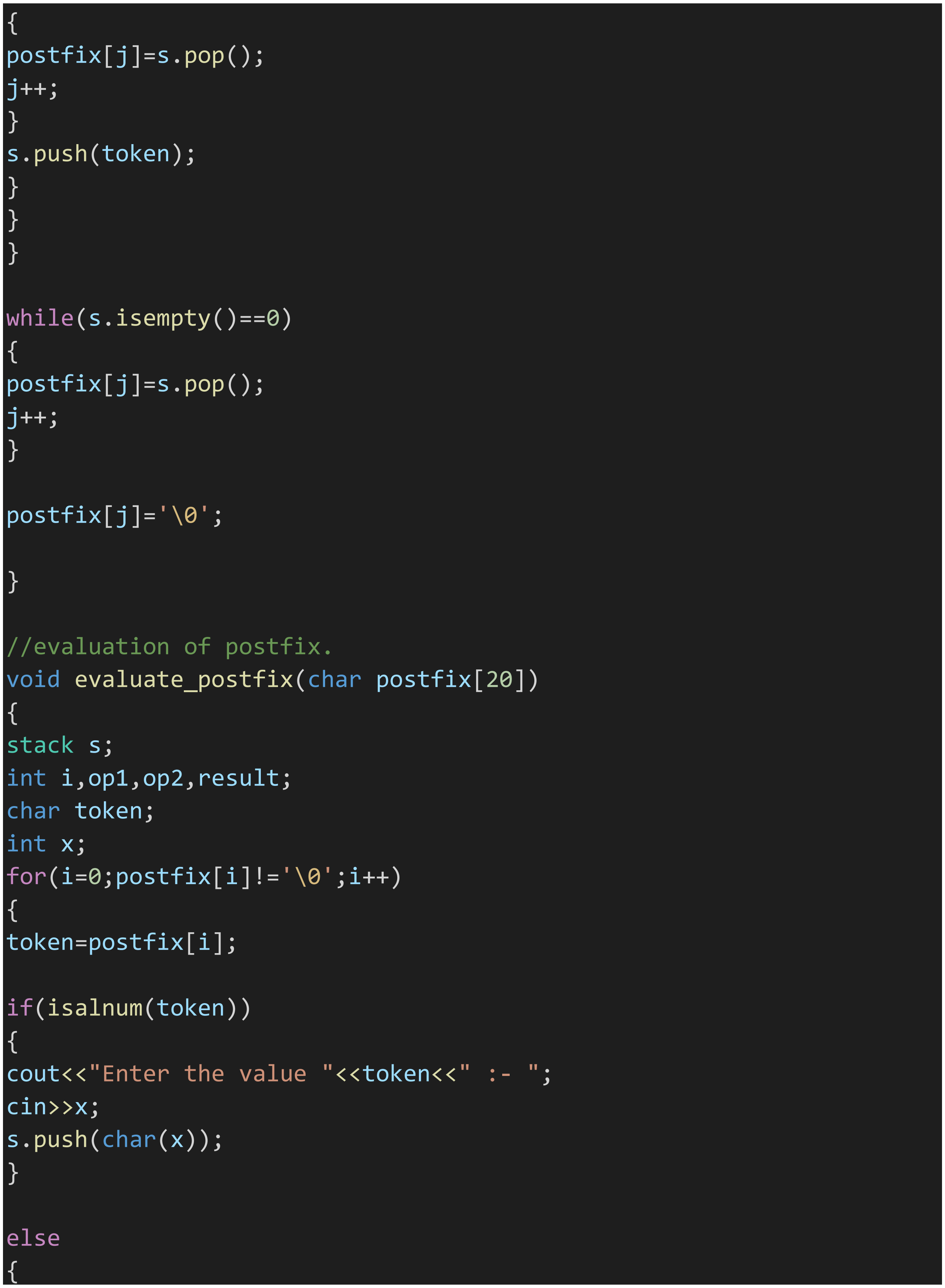
```
//infix----> postfix
void infix_postfix(char infix[20],char postfix[20])
{
stack s;
int i,j=0;
char token,x;
for(i=0;infix[i]!='\0';i++)
{
token=infix[i];

if(isalnum(token))                                       //will check
whether the token is number or an alphabet
{
postfix[j]=token;
j++;
}

else
{
if(token=='(')                                          //check whether
token is opening bracket, then push the token.
s.push(token);

else if(token==')')                                    //check whether
token is closing bracket.
{
while((x=s.pop())!='(')                               //pop all
elements until we get opening bracket.
{
postfix[j]=x;
j++;
}
}

else
{
while(s.isempty()==0 && precedence(token)<=precedence(s.topdata())
)             //first check stack is not empty and then the
precedence.
```



```
{
postfix[j]=s.pop();
j++;
}
s.push(token);
}
}
}

while(s.isempty()==0)
{
postfix[j]=s.pop();
j++;
}

postfix[j]='\0';

}

//evaluation of postfix.
void evaluate_postfix(char postfix[20])
{
stack s;
int i,op1,op2,result;
char token;
int x;
for(i=0;postfix[i]!='\0';i++)
{
token=postfix[i];

if(isalnum(token))
{
cout<<"Enter the value "<<token<<" :- ";
cin>>x;
s.push(char(x));
}

else
{
```

```
op2=s.pop();        //first pop
op1=s.pop();        //second pop
result=evaluate(op1,op2,token);
s.push(char(result));
}
}

result=s.pop();
cout<<"\n------------------------------";
cout<<"\nPOSTFIX EVALUATION- "<<result<<endl;
cout<<"------------------------------"<<endl;
}



//to reverse the given infix for prefix
void reverse(char x[20],char y[20])         //x will store infix and
y will store reverse of infix
{
int i,j=0;
for(i=0;x[i]!='\0';i++);
i--;                                       //since 'for' loop repeat
once again when the condition is false.

for(j=0;i>=0;j++,i--)
{
if(x[i]=='(')
y[j]=')';

else if(x[i]==')')
y[j]='(';

else
y[j]=x[i];
}
y[j]='\0';
}

//infix----> prefix
```

Name-Harshita Totala   Roll No- SEITA14

Div-A

```
void infix_prefix(char infix[20],char prefix[20])
{
char rev[20],pre1[20];
reverse(infix,rev);
infix_prefix_1(rev,pre1);
reverse(pre1,prefix);
}


void infix_prefix_1(char infix[20],char prefix[20])
{
stack s;
int i,j=0;
char token,x;
for(i=0;infix[i]!='\0';i++)
{
token=infix[i];

if(isalnum(token))                                  //will check
whether the token is number or an alphabet
{
prefix[j]=token;
j++;
}
else
{
if(token=='(')                                     //check whether
token is opening bracket, then push the token.
s.push(token);

else if(token==')')                               //check whether
token is closing bracket.
{
while((x=s.pop())!='(')                          //pop all
elements until we get opening bracket.
{
prefix[j]=x;
j++;
}
```

Name-Harshita Totala   Roll No- SEITA14

Div-A

```
}
else
{
while(s.isempty()==0 && precedence(token)<precedence(s.topdata())
)             //first check stack is not empty and then the
precedence.
{
prefix[j]=s.pop();
j++;
}
s.push(token);
}
}
}
while(s.isempty()==0)
{
prefix[j]=s.pop();
j++;
}

prefix[j]='\0';

}



//evaluation of prefix.
void evaluate_prefix(char prefix[20])
{
stack s;
int i,op1,op2,result;
char token,pre[20];
int x;
reverse(prefix,pre);
for(i=0;pre[i]!='\0';i++)
{
token=pre[i];

if(isalnum(token))
```

```
{
cout<<"Enter the value "<<token<<" :- ";
cin>>x;
s.push(char(x));
}

else
{
op1=s.pop();         //first pop
op2=s.pop();         //second pop
result=evaluate(op1,op2,token);
s.push(char(result));
}
}

result=s.pop();
cout<<"\n------------------------------";
cout<<"\nPREFIX EVALUATION- "<<result<<endl;
cout<<"------------------------------"<<endl;
}



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

int evaluate(int op1,int op2,char op)
{
if(op=='+')
return op1+op2;
if(op=='-')
```

```
return op1-op2;
if(op=='*')
return op1*op2;
if(op=='/')
return op1/op2;
if(op='%')
return op1%op2;
}
```

Name-Harshita Totala Roll No- SEITA14

Div-A

Output-

```
1. Infix to Postfix Expression
2. Evaluate Postfix
3. Infix to Prefix Expression
4. Evalute Prefix
5. Exit
Enter your choice: 1

Enter Infix expression:  (5-2)+8/2-1

------------------------------
POSTFIX EXPRESSION- 52-82/+1-
------------------------------

1. Infix to Postfix Expression
2. Evaluate Postfix
3. Infix to Prefix Expression
4. Evalute Prefix
5. Exit
Enter your choice: 2
Enter the value 5 :- 5
Enter the value 2 :- 2
Enter the value 8 :- 8
Enter the value 2 :- 2
Enter the value 1 :- 1

------------------------------
POSTFIX EVALUATION- 6
------------------------------

1. Infix to Postfix Expression
```

```
2. Evaluate Postfix
3. Infix to Prefix Expression
4. Evalute Prefix
5. Exit
Enter your choice: 3

Enter Infix expression:  (5-2)+8/2-1

------------------------------
PREFIX EXPRESSION- -+-52/821
------------------------------

1. Infix to Postfix Expression
2. Evaluate Postfix
3. Infix to Prefix Expression
4. Evalute Prefix
5. Exit
Enter your choice: 4
Enter the value 1 :- 1
Enter the value 2 :- 2
Enter the value 8 :- 8
Enter the value 2 :- 2
Enter the value 5 :- 5

------------------------------
PREFIX EVALUATION- 6
------------------------------

1. Infix to Postfix Expression
2. Evaluate Postfix
3. Infix to Prefix Expression
4. Evalute Prefix
```

```
5. Exit
Enter your choice: 5

Program Exited!
```

Conclusion: In this experiment we understood how 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.