EXPERIMENT 1

Vineet Parmar

2021300092

Batch B2

Aim : Implement Infix to Postfix conversion using Stack.

Theory:

- **Stack:**
  Stack is an abstract Data type, commonly used in programming languages. It is named stack as it behaves in the real world, for example – a deck of cards, or a pile of plates, etc.
  Stack is a Last In First Out (LIFO) Data Structure. In stack terminologies, insertion is called as push operation and deletion is called as pop operation.
- **Polish and Reverse Polish Notation:**
  Polish notation (PN), also known as normal Polish notation (NPN), Łukasiewicz notation, Warsaw notation, Polish prefix notation or simply prefix notation, is a mathematical notation in which operators precede their operands, in contrast to the more common infix notation, in which operators are placed between operands, as well as reverse Polish notation (RPN), in which operators follow their operands. It does not need any parentheses as long as each operator has a fixed number of operands.
  Polish Notation – Prefix
  Reverse Polish Notation – Postfix Notation.
- **Conversion of Infix to Polish Notation:**
  1. Read expression from Left-to-Right and:
  2. If an operand is read, copy it to the ouput.
  3. If the operator is '(', then push it into the stack.
  4. If the operator is ')', then print the top element of the stack and pop it and repeat it until '(' is found. When that occurs, both parantheses are discarded.
  5. If an operator is read and has a higher precedence than the topmost element of the stack, the operator is pushed into the stack.

6. If an operator is read and has a lower precedence or equal precedence than the topmost element of the stack, the topmost element of the stack is printed and the popped out and then the read operator is pushed into the stack.

7. After reading the input, Print all the elements of stack by popping it.

Algorithm:

Class InfixtoPostfix

Main method

1. Create array to store operators in order of precedence
2. Create object of STACK
3. Input an equation from the user
4. Convert the equation into a character array
5. Create new character array to store answer
6. For Loop from I = 0 to I = length of equation array:
   a. check if eq[i] is an operand. If yes, then store it in answer array
   b. if not, then check if it is '('. If yes, then push it into stack
   c. if not, then check if it is ')'. If yes, then keep printing and popping the elements of the stack until '(' is found. Discard the '('.
   d. if not, check if the operator has higher precedence than the topmost element of the stack. If yes, push the element into the stack. If no, then print the top most element and pop it and repeat the process till operator has a higher precedence.
7. Print and pop the topmost elements of the stack till it is empty.

IndexOf method (for precedence)

1. If 0^{th} index or 1^{st} index of the array matches the character, return 1.
2. If 2^{nd} index or 3^{rd} index of the array matches the character, return 2.
3. If any other index, matches the character, return the index.
4. Else, return 0.

Class STACK

Data Members:

- Character array stack
- Integer variable t (denoting top)
- Integer variable size

STACK() Constructor

1. Set size to 100.
2. Set t to -1
3. Initialize char array.

Boolean IsEmpty() method

1. Return true if t equals to -1

Boolean IsFull() method

1. Return true if t equals to size – 1

Void push(char c) method

1. Check if stack is full or not. If not then add the element into the stack after incrementing t.

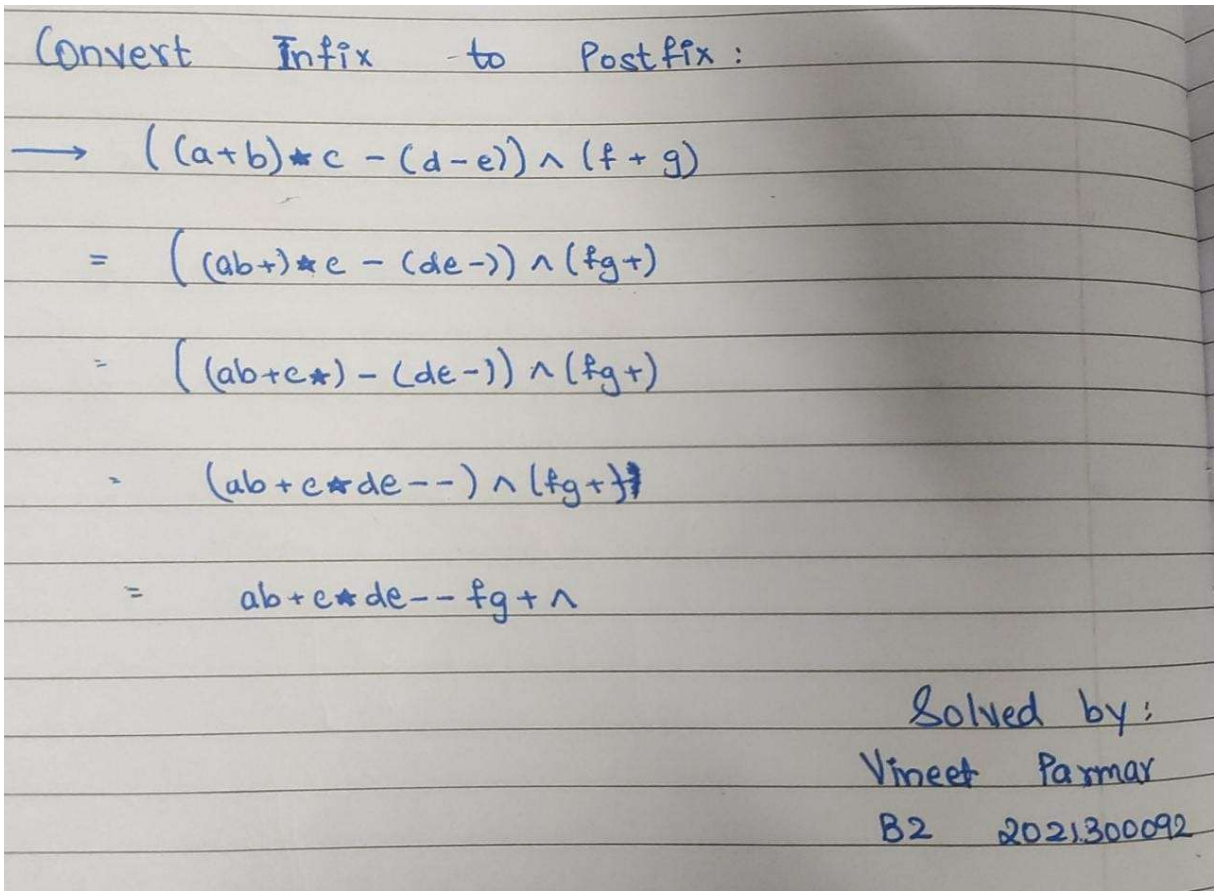
Void pop() method

1. Check if stack is empty or not. If not then decrement t by 1

Char top() method

1. Check if stack is empty or not. If not then return stack[t]

Problem Solving:



CODE:

```
import dsa.STACK;
import java.util.*;
class InfixtoPostfix
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        char[] pr = {'+', '-', '*', '/', '%', '^'};
        STACK obj = new STACK();
        System.out.println("Enter an equation: ");
        String e = sc.nextLine();
        char[] eq = e.toCharArray();
        char[] ans = new char[eq.length];
        int k = 0, flag = 0;
        for(int i = 0; i < eq.length; i++)
        {
```

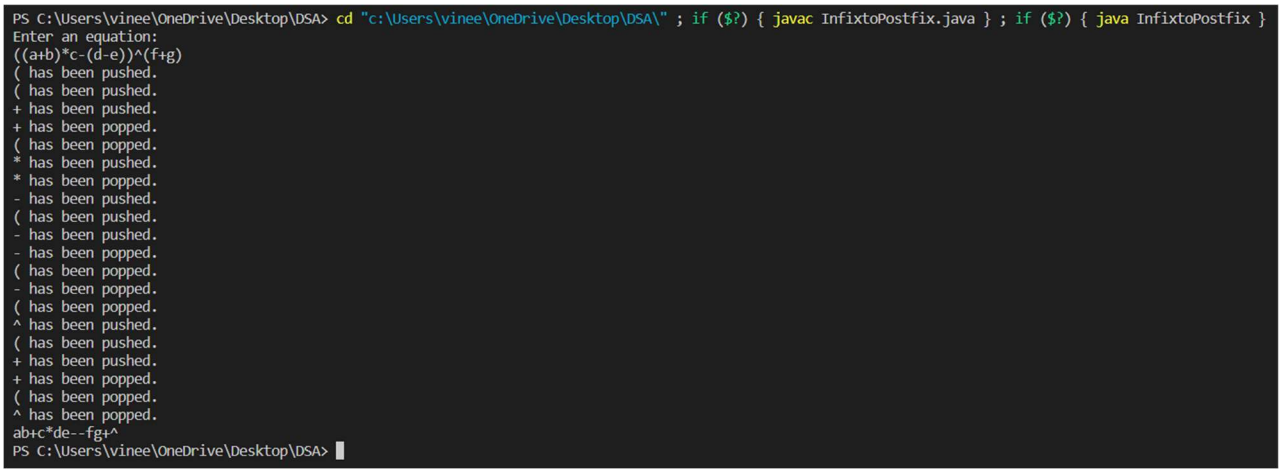
```
            if(eq[i] != '+' && eq[i] != '-' && eq[i] != '*' && eq[i] != '*' &&
eq[i] != '/' && eq[i] != '%' && eq[i] != '^' && eq[i] != '(' && eq[i] != ')')
            {
                ans[k++] = eq[i];
            }
            else if(eq[i] == '(')
            {
                obj.push(eq[i]);
            }
            else if(eq[i] == ')')
            {

                while(obj.top() != '(')
                {
                    ans[k++] = obj.top();
                    obj.pop();
                }
                obj.pop();
            }
            else
            {
                if(obj.isEmpty() || indexOf(pr, (eq[i])) > indexOf(pr,
obj.top()))
                {
                    obj.push(eq[i]);
                }
                else
                {
                    ans[k++] = obj.top();
                    obj.pop();
                    obj.push(eq[i]);
                }
            }
        }
        while(!obj.isEmpty())
        {
            ans[k++] = obj.top();
            if(obj.top() == '(')
            {
                flag = 1;
            }
            obj.pop();
        }
        if(flag == 1)
        {
```

```
            System.out.println("Invalid Equation");
        }
        else
        {
            for(int i = 0; i < eq.length; i++)
            {
                System.out.print(ans[i]);
            }
        }
    }
    static int indexOf(char[] arr, char c)
    {
        for(int i = 0; i < arr.length; i++)
        {
            if(arr[0] == c || arr[1] == c)
            {
                return 1;
            }
            else if(arr[2] == c || arr[3] == c)
            {
                return 2;
            }
            else if(arr[i] == c)
            {
                return i;
            }
        }
        return 0;
    }
}
package dsa;
public class STACK
{
    char stack[];
    int size;
    int t;
    public STACK()
    {
        size = 100;
        t = -1;
        stack = new char[size];
    }
    public void push(char n)
    {
        if(!isFull())
```

```
        {
            stack[++t] = n;
            System.out.println(n + " has been pushed.");
        }
        else
        {
            System.out.println("STACK OVERFLOW.");
        }
    }
    public void pop()
    {
        if(!isEmpty())
        {
            System.out.println(top() + " has been popped.");
            t--;
        }
        else
        {
            System.out.println("STACK UNDERFLOW.");
        }
    }
    public char top()
    {
        return stack[t];
    }
    public boolean isEmpty()
    {
        return t == -1;
    }
    public boolean isFull()
    {
        return t == size - 1;
    }
}
```

OUTPUT:



CONCLUSION:

With the help of this experiment, we learned about the STACK Data Structure. We learned about the basic Stack operations such as push(), pop() and the conditions isEmpty() and isFull(). Using this knowledge, we then went on to write a program that converted an infix expression to a postfix expression (Reverse Polish Notation). We used Stacks to store the operators and the parantheses, and then used the push() and pop() operations according to the given conditions and successfully got the desired output.