?

Infix to Postfix Conversion

① Print Operands as they arrive

② If stack is empty or contains a left paranthesis on top, push the incoming operator onto the stack

③ If incoming symbol is '(', push it onto stack

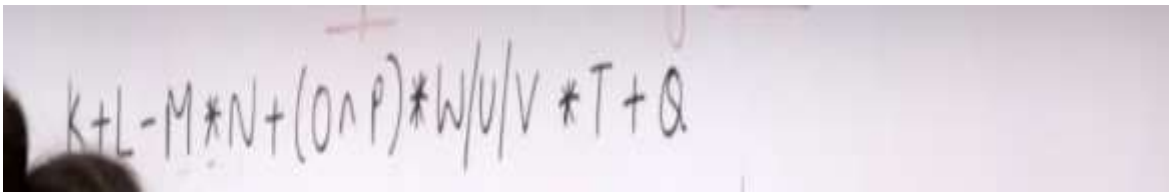
④ If incoming symbol is ')', pop the stack & print the operators until left paranthesis is found.

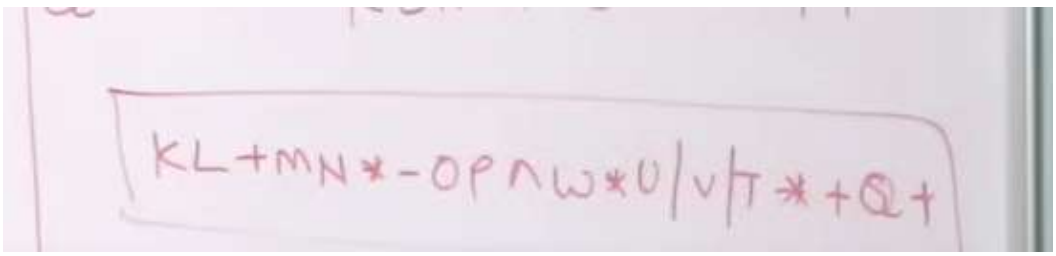
⑤ If incoming symbol has higher precedence than the top of the stack, push it on the stack.

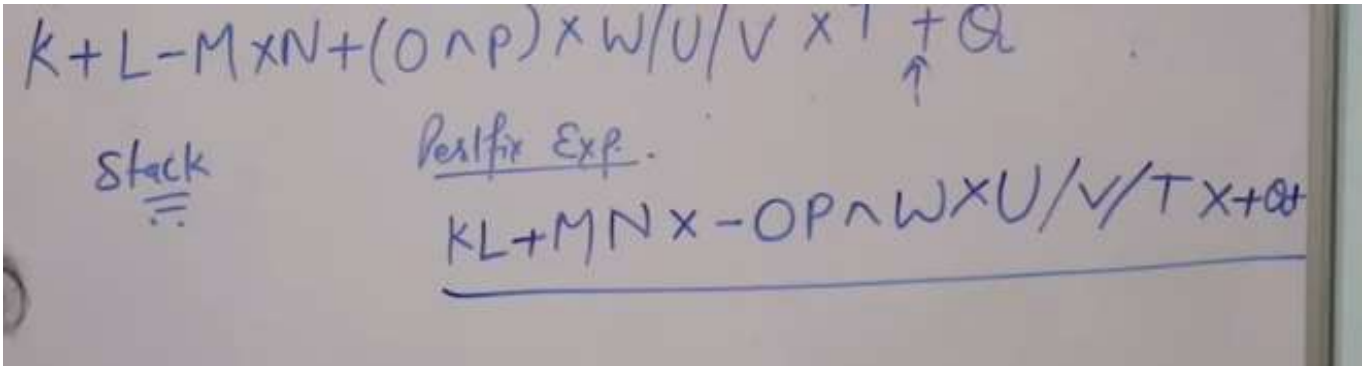
⑥ If incoming symbol has lower precedence than the top of the stack, pop & print the top. Then test the incoming operator against the new top of the stack.

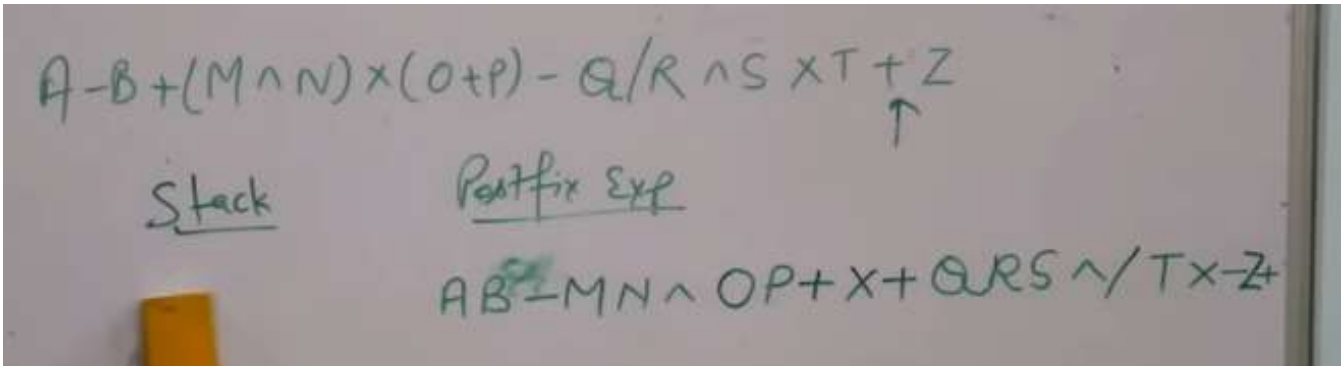
⑦ If incoming operator has equal precedence with the top of the stack, use associativity rule.
⇒ associativity L to R then pop & print the top of the stack & then push the incoming operator
⇒ R to L then push the incoming operator

⑧ At the end of the expression, pop & print all operators of stack.









The following table briefly tries to show the difference in all three notations –

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation –

```
Step 1. Scan the expression from left to right
Step 2. If it is an operand push it to stack
Step 3. If it is an operator pull operand from stack and
        perform operation
Step 4. Store the output of step 3, back to stack
Step 5. Scan the expression until all operands are consumed
Step 6. Pop the stack and perform operation
```

For Prefix evaluation: Scan the prefix expression from right and follow the same steps as above.