Infix to Postfix Conversion Using Stack

The infix expression is the one where the operator is placed between operands (e.g., `A + B`). On the other hand, in the postfix expression, the operator follows the operands (e.g., `A B +`). When converting an infix expression to a postfix expression, a stack data structure is used to help reorder operators based on their precedence.

Steps for Conversion:

1. Initialize an empty stack for operators and an empty output list for the result.

2. Traverse the infix expression from left to right:

   - If the current character is an operand, add it to the output list.

   - If the current character is an operator, pop operators from the stack to the output list until the operator at the top of the stack has less precedence. Then push the current operator onto the stack.

   - If the current character is an opening parenthesis (`(`), push it onto the stack.

   - If the current character is a closing parenthesis (`)`), pop operators from the stack to the output list until an opening parenthesis is encountered. Remove the opening parenthesis from the stack.

3. Pop all remaining operators from the stack to the output list.

Operator Precedence:

`*`, `/`, `%` have higher precedence.

`+`, `-` have lower precedence.

- Parentheses `()` change the normal precedence.

Example: Infix Expression `A + B * C`

Let's use this example to explain the process with a stack diagram.

Infix: `A + B * C`

Step-by-Step Conversion:

1. **Read `A`:**

   - It's an operand, so add it to the output.

   - Output: `A`

   - Stack: `[]`

2. Read `+`:

   - It's an operator, and the stack is empty, so push it onto the stack.

   - Output: `A`

   - Stack: `[+]`

3. **Read `B`:**

   - It's an operand, so add it to the output.

   - Output: `A B`

   - Stack: `[+]`

4. **Read `*`:**

   - It's an operator. The operator on the stack (`+`) has lower precedence than `*`, so push `*` onto the stack.

   - Output: `A B`

   - Stack: `[+, *]`

5. **Read `C`:**

   - It's an operand, so add it to the output.

- Output: `A B C`

- Stack: `[+, *]`

6. **End of the expression:**

- Pop all operators from the stack to the output list.

- Pop `*` and add to output.

- Pop `+` and add to output.

- Output: `A B C * +`

- Stack: `[]`

Final Postfix Expression:

`A B C * +`

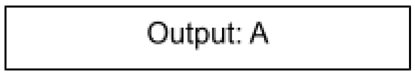
Stack Diagram Explanation:

Steps	**Read Character**	**Stack**	**Output**
1	A	Empty	A
2	+	+	A
3	B	+	AB
4	*	+,*	AB
5	C	+,*	ABC
6	Pop *, then +	Empty	**ABC*+**

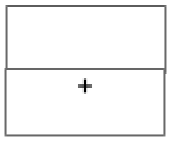
Visualization:

1. Initial: Empty stack, begin processing.

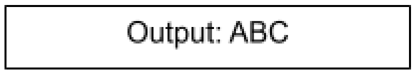
2. After reading `A`: Operand added to the output directly.



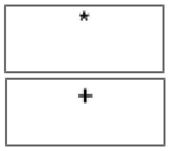
3. After reading `+`: Operator pushed to the stack.



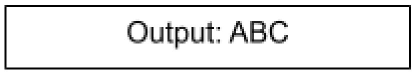
4. After reading `B`: Operand added to output.



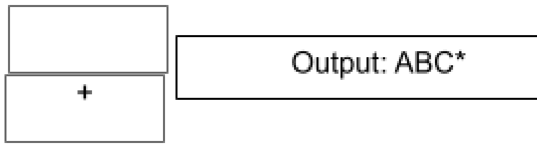
5. After reading `*`: Stack now has two operators, but precedence keeps `*` above `+`.

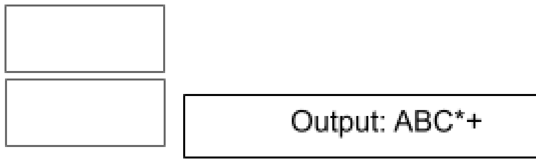


6. After reading `C`: Operand added to output.



7. Final: Pop operators `*` and `+` from the stack in the correct order.





This is how the infix expression `A + B * C` becomes `A B C * +` in postfix using the stack mechanism.