📘 Session 2-3: Stack Applications

🧱 Polish Notation (Postfix & Prefix)
🛠️ Infix to Postfix Conversion
🧮 Postfix Evaluation
📚 Step-by-step Dry Runs
📝 Worksheet for Practice
🎯 Focus: Simple, Beginner-Friendly, Fully Explained

✅ Part 1: Polish Notation (Prefix & Postfix)

🔸 Infix Notation (Normal Way)

- Operator is written **between operands**.
  Example: `A + B`

- Needs **parentheses** and **precedence rules**.

🔸 Postfix Notation (Reverse Polish)

- Operator is written **after operands**.
  Example: `A B +`

- No

🔸 Prefix Notation (Polish)

- Operator is written **before operands**.
  Example: `+ A B`

📌 Why Use Postfix or Prefix?

Reason	Explanation
No brackets needed	Expression is unambiguous

Stack-friendly	Computers can evaluate easily
Fast parsing	Used in compilers & interpreters

✅ Part 2: Infix to Postfix Conversion (Using Stack)

🧠 Goal: Convert `A + B * C` → `A B C * +`

📌 Algorithm: Infix to Postfix (💯 Easy Steps)

To convert an infix expression to postfix, **use a stack**.

📝 Full Step-by-Step Process

1. **Scan the infix expression** from left to right.
2. If the scanned character is an **operand**, **add it** to the postfix expression.
3. If the character is an **operator**:
    - If the **stack is empty**, or top of the stack is `'('`, or the current operator has **higher precedence**, **push** it to the stack.
    - ,
4. If the character is `'('`, **push it** to the stack.
5. If the character is `')'`, **pop and add to postfix** until `'('` is encountered. **Discard both parentheses**.
6. **Repeat** until the full expression is scanned.
7. **Pop remaining operators** in the stack and add them to the postfix expression.

🔶 Operator Precedence Table

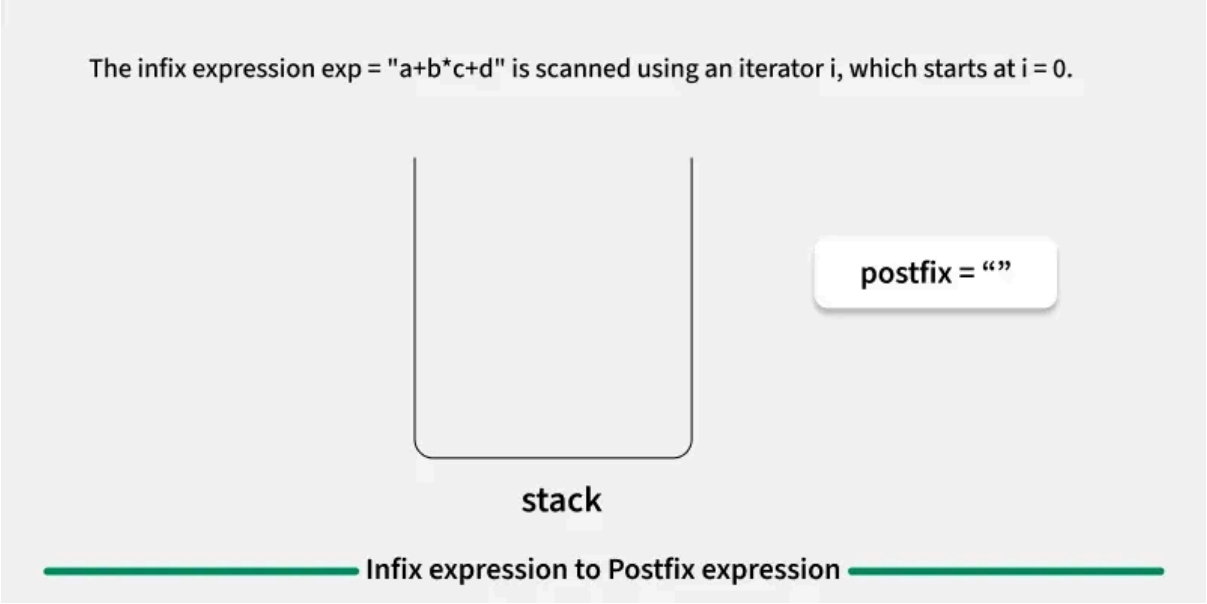
Operator	Precedence
^	Highest

`* /`	Medium
`+ -`	Lowest

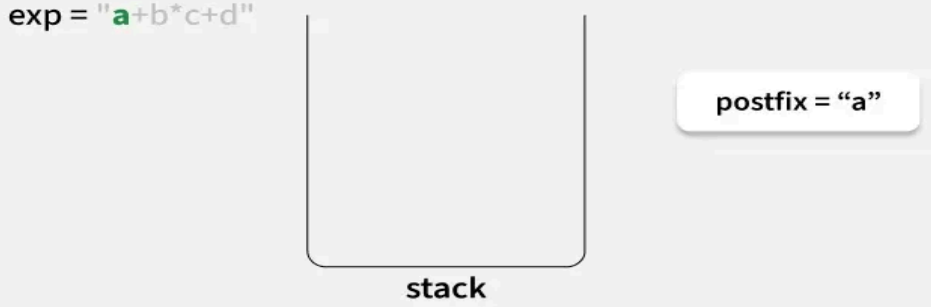
🧾 Example Dry Run: `A + (B * C)`

Step	Symbol	Stack	Postfix Expression
1	A		A
2	+	+	A
3	(	+ (	A
4	B	+ (	A B
5	*	+ (*	A B
6	C	+ (*	A B C
7	)	+	A B C *
8	End		A B C * +

Pictorial view :

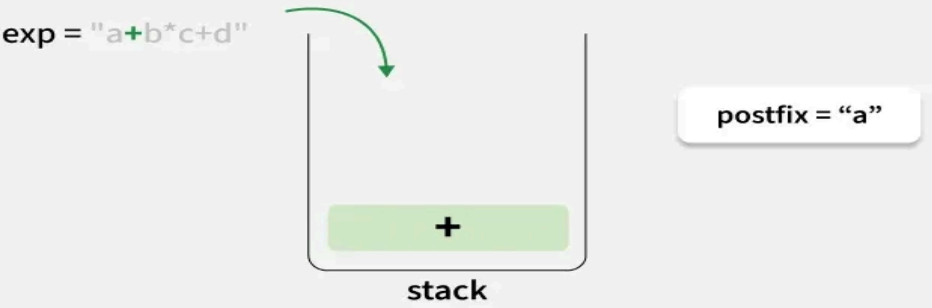


01 Here i = 0 and exp[i] = ‘a’ i.e., an operand. So add this in the postfix expression.



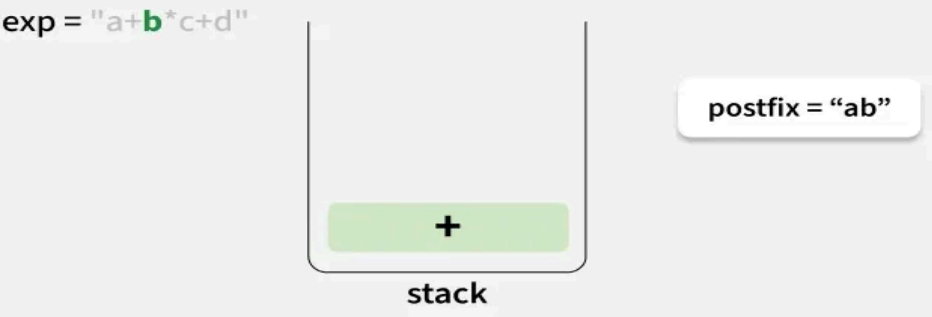
Infix expression to Postfix expression

02 Here i = 1 and exp[i] = ‘+’ i.e., an operator. Push this into the stack.



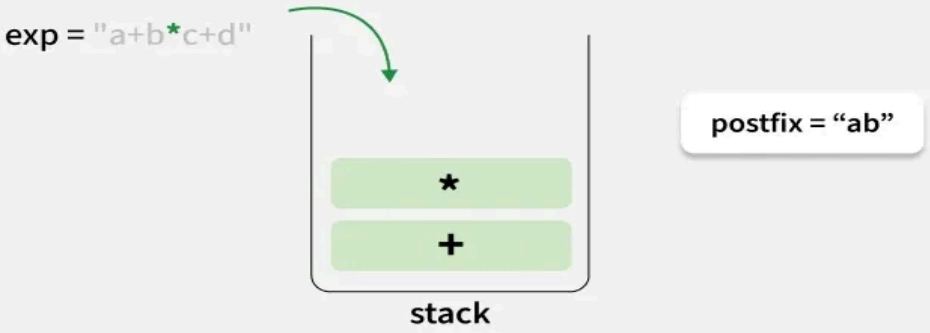
Infix expression to Postfix expression

03 Now i = 2 and exp[i] = ‘b’ i.e., an operand. So add this in the postfix expression.



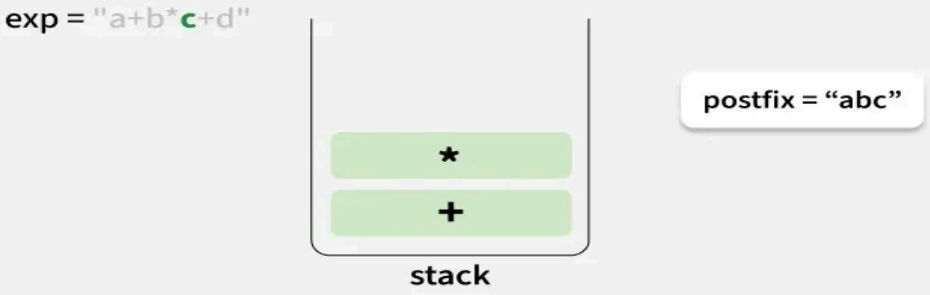
Infix expression to Postfix expression

04 Now i = 3 and exp[i] = '' i.e., an operator and has higher precedence than + so push it into stack.



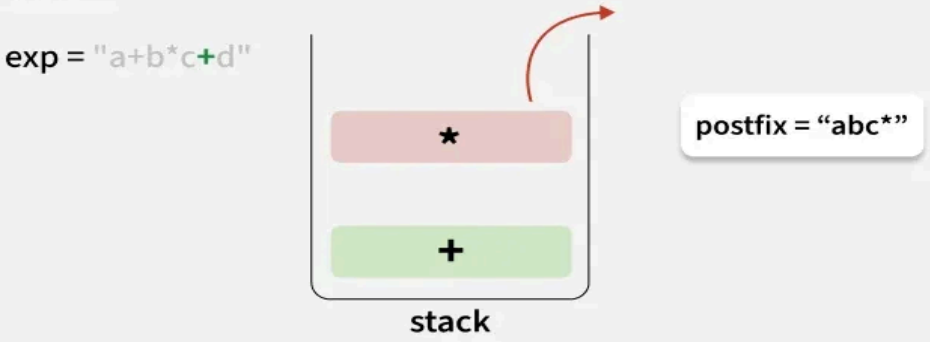
Infix expression to Postfix expression

05 Now i = 4 and exp[i] = 'c' i.e., an operand, Add this in the postfix expression.

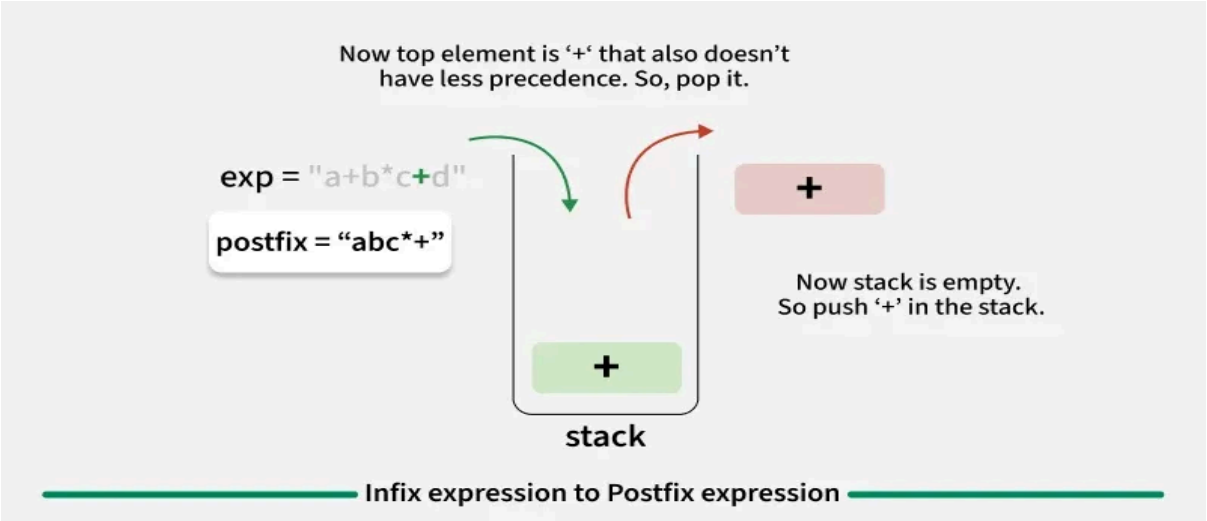


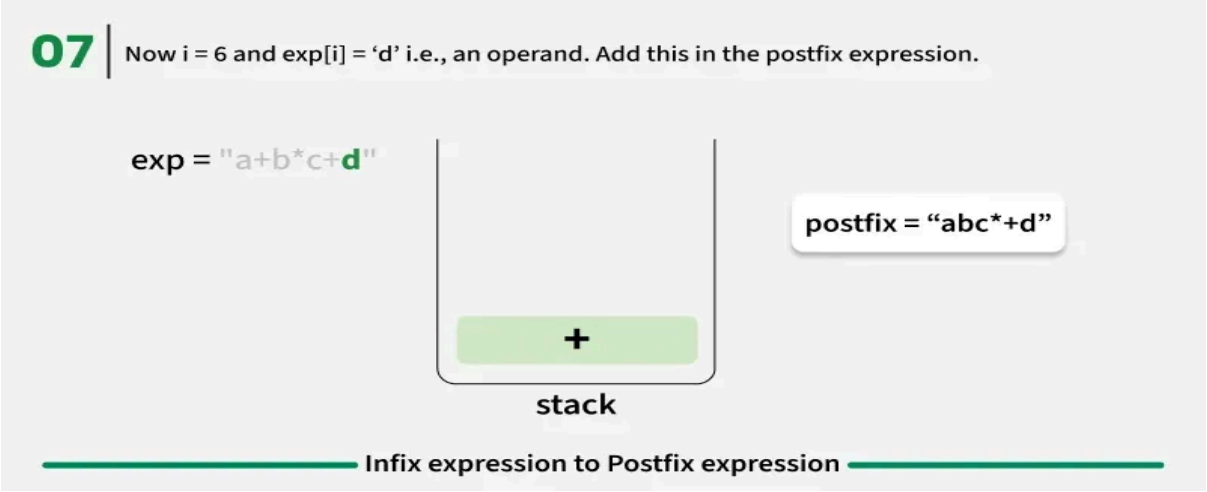
Infix expression to Postfix expression

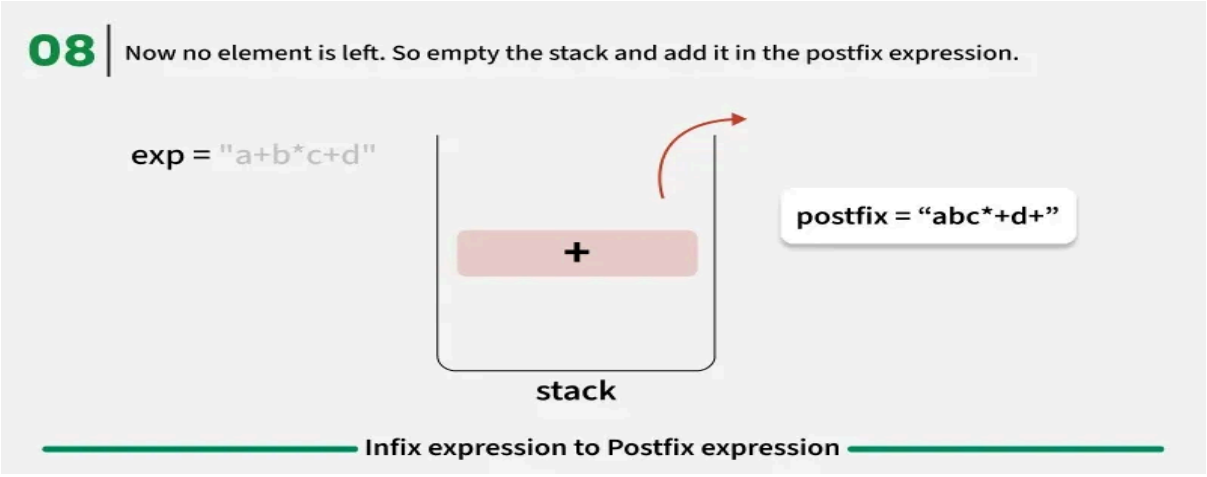
06 Now i = 5 and exp[i] = '+' i.e., an operator. The topmost element of the stack has higher precedence. So, pop the stack until it's empty or the top element has lower precedence, including when the precedence is the same, and add the popped operator to the postfix expression



Infix expression to Postfix expression







🧠 2 Key Examples to Understand Infix to postfix example : -

📘 1. Left-to-Right Associativity Example

Expression:

`A - B + C`

🔍 Step-by-Step:

Step	Operators	Associativity	What Happens
1	`-` and `+` both have same precedence (1)	Left-to-Right	First `A - B` happens
2	Then result is added to `C`: `(A - B) + C`		

📥 Postfix Conversion:

Symbol	Stack	Output
A		A
-	-	A
B	-	A B
+	+	A B -
C	+	A B - C
End		A B - C +

✅ **Postfix:** `A B - C +`

📗 2. Right-to-Left Associativity Example

Expression:

`A ^ B ^ C`

🔍 Step-by-Step:

Step	Operators	Associativity	What Happens
1	`^` and `^` → Same precedence (3)	Right-to-Left	First `B ^ C` happens
2	Then `A ^ (B ^ C)` is evaluated		

📥 Postfix Conversion:

Symbol	Stack	Output
A		A
^	^	A
B	^	A B
^	^ ^	A B
C	^ ^	A B C
End		A B C ^ ^

✅ **Postfix:** `A B C ^ ^`

`K + L - M * N + (O ^ P) * W / U / V * T + Q`

We will go **step-by-step** to:

1. Understand the expression
2. Convert it to **Postfix notation** using **stack**
3. Visualize the **stack + postfix** in a table

🧠 Step 1: Expression Analysis

Expression:

`K + L - M * N + (O ^ P) * W / U / V * T + Q`

Operator Precedence:

Operator	Precedence	Associativity
^	3	Right to Left
* /	2	Left to Right
+ -	1	Left to Right

Parentheses

- `(O ^ P)` means calculate `O ^ P` first.

✅ Step 2: Conversion Rules (Infix → Postfix)

Algorithm Summary:

1. Operand → Add to postfix
2. `(` → Push to stack
3. `)` → Pop till `(`
4. Operator → Pop all higher/equal precedence operators from stack, then push

🔁 Step 3: Step-by-Step Table

We'll scan the expression **from left to right**:

Char	Action	Stack	Postfix
K	Operand → Add to postfix		K
+	Operator → Push to stack	+	K
L	Operand → Add to postfix	+	K L
-	Pop + (same/lower precedence)	-	K L +
M	Operand → Add to postfix	-	K L + M
*	Operator → Higher than - → Push	- *	K L + M
N	Operand → Add to postfix	- *	K L + M N
+	Pop * (higher), then - (same)	+	K L + M N * -
(	Push to stack	+ (	K L + M N * -

O	Operand	+ (	K L + M N * - O
^	Push (^)	+ (^	K L + M N * - O
P	Operand	+ (^	K L + M N * - O P
)	Pop ^ → postfix, pop (	+	K L + M N * - O P ^
*	Push	+ *	K L + M N * - O P ^
W	Operand	+ *	K L + M N * - O P ^ W
/	Push (equal precedence, left-assoc)	+ * /	K L + M N * - O P ^ W
U	Operand	+ * /	K L + M N * - O P ^ W U
/	Pop / → postfix (same), then push	+ * /	K L + M N * - O P ^ W U /
V	Operand	+ * /	K L + M N * - O P ^ W U / V
*	Pop / → postfix, push *	+ * *	K L + M N * - O P ^ W U / V /
T	Operand	+ * *	K L + M N * - O P ^ W U / V / T

+	Pop * and * → postfix, then +	+	K L + M N * - O P ^ W U / V / T * *
Q	Operand	+	K L + M N * - O P ^ W U / V / T * * Q
End	Pop +		K L + M N * - O P ^ W U / V / T * * Q + +

✅ Final Postfix Expression:

```
K L + M N * - O P ^ W * U / V / T * + Q +
```

✅ Code: Infix to Postfix Conversion in C++

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

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

bool isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
}
```

```
string infixToPostfix(string exp) {
    stack<char> st;
    string result;

    for (char ch : exp) {
        if (isalnum(ch)) {
            result += ch;
        } else if (ch == '(') {
            st.push(ch);
        } else if (ch == ')') {
            while (!st.empty() && st.top() != '(') {
                result += st.top();
                st.pop();
            }
            st.pop(); // remove '('
        } else if (isOperator(ch)) {
                  while (!st.empty() && getPrecedence(st.top()) >=
getPrecedence(ch)) {
                result += st.top();
                st.pop();
            }
            st.push(ch);
        }
    }

    while (!st.empty()) {
        result += st.top();
        st.pop();
    }

    return result;
}

int main() {
    string infix;
    cout << "Enter infix expression (e.g., A+B*C): ";
    cin >> infix;

    string postfix = infixToPostfix(infix);
    cout << "Postfix expression: " << postfix << endl;
    return 0;
```

```
}
```

✅ Part 3: Postfix Evaluation (With Integers)

📌 Goal: Evaluate Postfix Like `5 6 2 + *` → Result = `40`

🧮 Algorithm (Using Stack)

1. Scan the postfix expression **from left to right**.
2. If a character is a **number**, push to **stack**.
3. If the character is an **operator**, pop **two values**, apply the operator, and **push the result**.
4. After scanning, top of the stack is the **final answer**.

🔢 Dry Run: `5 6 2 + *`

Step	Symbol	Stack	Action
1	5	5	Push
2	6	5 6	Push
3	2	5 6 2	Push
4	+	5 8	6+2 = 8, Push
5	*	40	5*8 = 40, Push

✅ **Final Answer = 40**

✅ Code: Postfix Evaluation in C++

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

```
int evaluatePostfix(string expr) {
    stack<int> st;
    stringstream ss(expr);
    string token;

    while (ss >> token) {
        if (isdigit(token[0])) {
            st.push(stoi(token));
        } else {
            int b = st.top(); st.pop();
            int a = st.top(); st.pop();
            switch (token[0]) {
                case '+': st.push(a + b); break;
                case '-': st.push(a - b); break;
                case '*': st.push(a * b); break;
                case '/': st.push(a / b); break;
            }
        }
    }
    return st.top();
}

int main() {
    string postfix;
    cout << "Enter postfix expression with spaces (e.g., 5 6 2 + *):
";
    getline(cin, postfix);

    int result = evaluatePostfix(postfix);
    cout << "Result = " << result << endl;
    return 0;
}
```

📝 Practice Worksheet

🔹 A. Convert Infix to Postfix:

1. `A + B` → `_________`

2. `(A + B) * C` → `_____`
3. `A * (B + C)` → `_____`
4. `(A + B) * (C - D)` → `_____`
5. `A + B * C - D / E` → `_____`

🔹 **B. Evaluate Postfix Expressions:**

1. `5 2 +` → `_____`
2. `6 2 + 3 /` → `_____`
3. `10 2 8 * +` → `_____`
4. `100 20 5 + /` → `_____`
5. `5 3 2 * + 6 -` → `_____`

🔹 **C. Dry Run Table (Fill It)**

Infix: `(A + B) * C - D` = `A B + C * D -`

🧪 Quick Quiz for Students

Q1: What is the postfix of `X - Y + Z`?

- A. `X Y - Z +`
- B. `X Y Z - +`
- C. `X Y + Z -`

✅ **Answer:** A. X Y - Z +

Q2: What is the postfix of X ^ Y ^ Z?

- A. X Y ^ Z ^
- B. X Y Z ^ ^
- C. X Y Z ^

✅ **Answer:** B. X Y Z ^ ^

Q3: What is the postfix of A * B / C?

- A. A B * C /
- B. A B C * /
- C. A B C / *

✅ **Answer:** A. A B * C / (because * and / are left-to-right)

INFIX TO PREFIX CONVERSION

📘 Infix to Prefix Conversion –

🔶 1. What is Prefix Notation?

Prefix (Polish) Notation:
In this format, the **operator comes before the operands**.

📌 **Example:**

Infix	Prefix

A + B	+AB
A + B * C	+A*BC
(A + B) * C	*+ABC

🔶 2. Why Convert Infix to Prefix?

- Computers evaluate prefix expressions **faster**.
- **No brackets** are needed in prefix form.
- Removes confusion due to operator **precedence and associativity**.

🔶 3. Rules of Precedence and Associativity

Operator	Precedence	Associativity
^	Highest	Right to Left (R→L)
* / %	Medium	Left to Right (L→R)
+ -	Lowest	Left to Right (L→R)

🔶 4. Algorithm: Infix to Prefix (Using Stack)

🧠 **Step-by-Step:**

1. **Reverse the infix expression**
   - Swap (with) and vice versa
2. **Convert the reversed expression to postfix** using stack (same as infix → postfix rules)
3. **Reverse the postfix expression** → This is your **prefix**

🔁 5. Example Dry Run

📝 **Infix:**

```
(A - B/C) * (A/K - L)
```

✅ **Step 1: Reverse the infix**

```
(L - K/A) * (C/B - A)
```

Also swap (and)

✅ **Step 2: Convert reversed infix → Postfix**

Infix:

```
(L - K / A) * (C / B - A)
```

Postfix:

```
L K A / - C B / A - *
```

✅ **Step 3: Reverse the postfix**

Prefix:

```
* - L / K A - / C B A
```

📋 Session 04 Quiz: Stack Application

✅ Section A: Conceptual Understanding

Q1. What is the main reason we convert infix expressions to postfix before evaluating them in computers?

Q2. Which notation does not require brackets:
a) Infix
b) Postfix
c) Prefix

Q3. What is the stack used for in infix to postfix conversion?

Q4. What is the correct postfix form of the infix expression `A + B * C`?

Q5. In postfix evaluation, how many operands do you pop when you find an operator?

✅ Section B: Operator Precedence

Q6. Arrange the following operators in decreasing order of precedence:
`+ * ^ / -`

Q7. What is the precedence of `+` and `-`?

Q8. What is the precedence of `^`?

✅ Section C: Infix to Postfix Conversion

Convert the following infix expressions to **postfix**:

Q9. `A + B`

Q10. `(A + B) * C`

Q11. `A * (B + C) / D`

Q12. `A + B * C - D`

✅ Section D: Postfix Evaluation

Evaluate the following postfix expressions step-by-step (show stack changes):

Q13. `5 2 +`
Answer:

Q14. `6 2 + 3 /`
Answer:

Q15. `10 2 8 * +`
Answer:

Q16. `100 20 5 + /`
Answer:

✅ Section E: Dry Run Trace

Q17. Dry run (step-by-step table) for infix to postfix conversion of:
`A + (B * C)`

Step	Symbol	Stack	Output
1			
2			
…			

✅ Section F: True or False

Q18. Postfix notation requires parentheses to maintain operator precedence.
→ _____

Q19. In postfix evaluation, the stack always stores operators.
→ _____

Q20. Postfix evaluation is faster for computers than infix.
→ _____

✅ Answer Key (for Students)

Section A

A1. Because it avoids brackets and handles precedence easily using stack.
A2. b) Postfix
A3. To temporarily hold operators and manage precedence.
A4. `A B C * +`
A5. 2 operands

Section B

A6. `^ > * > / > + > -`
A7. 1
A8. 3

Section C

A9. `A B +`
A10. `A B + C *`
A11. `A B C + * D /`
A12. `A B C * + D -`

Section D

A13. 5 2 → + → 7
A14. (6+2)=8 → 8/3 = 2
A15. 2×8=16 → 10+16 = 26
A16. (20+5)=25 → 100/25 = 4

Section E

Dry run for `A + (B * C)`:

Steps	Symbol	Stack	Output
1	A		A
2	+	+	A
3	(	+ (	A

4	B	+ (	A B
5	*	+ (*	A B
6	C	+ (*	A B C
7	)	+	A B C *
8	End		A B C * +

Section F

A18. False
A19. False (stack stores operands)
A20. True