جامـــــعة المـــــستقبل
AL MUSTAQBAL UNIVERSITY

قســـم الامـــــن الـــــــــسيبرانــــــــــي
Department of Cyber Security

Subject: Data Structure

Class: Second

Lecturer: Asst. Prof. Dr. Ali Kadhum Al-Quraby

Lecture: 8

Convert Infix expression to Postfix expression

Study Year: 2024-2025

• Convert Infix expression to Postfix expression

• Write a program to convert an Infix expression to Postfix form.

- ***Infix expression:*** *The expression of the form "a operator b" (a + b) i.e., when an operator is in-between every pair of operands.*
- ***Postfix expression:*** *The expression of the form "a b operator" (ab+) i.e., When every pair of operands is followed by an operator.*

Examples:

- ***Input:*** *A + B * C + D*
  Output: *ABC*+D+*

- ***Input:*** *((A + B) – C * (D / E)) + F*
  Output: *AB+CDE/*-F+*

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.
Consider the expression: **a + b * c + d**

- The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it.
- The result is then added to d after another scan.

The repeated scanning makes it very inefficient. Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix (or prefix) form before evaluation.
The corresponding expression in postfix form is **abc*+d+**. The postfix expressions can be evaluated easily using a stack.

How to convert an Infix expression to a Postfix expression?

*To convert infix expression to postfix expression, use the **stack data structure**. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.*
Below are the steps to implement the above idea:

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
   - If the precedence of the current scanned operator is higher than the precedence of the operator on top of the stack, or if the stack is empty, or if the stack contains a '(', then push the current operator onto the stack.

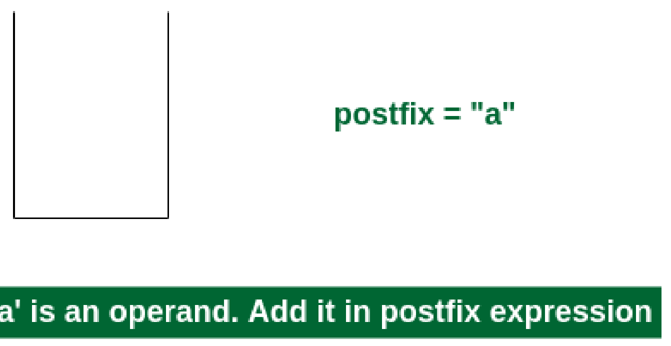
   - Else, pop all operators from the stack that have precedence higher than or equal to that of the current operator. After that push the current operator onto the stack.
4. If the scanned character is a ‘**(**‘, push it to the stack.
5. If the scanned character is a ‘**)**’, pop the stack and output it until a ‘**(**‘ is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Illustration:

Follow the below illustration for a better understanding

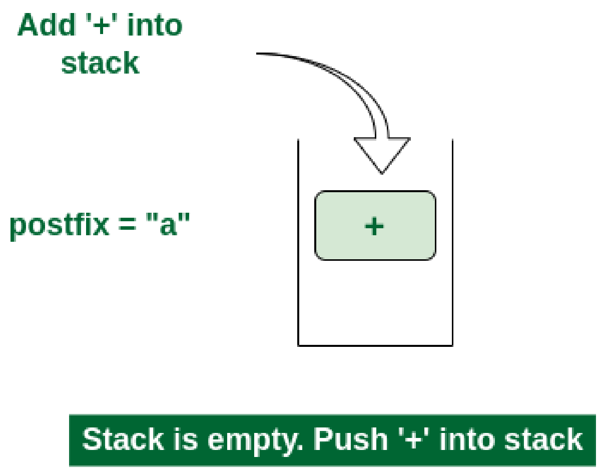
*Consider the infix expression **exp** = **“a+b*c+d”** and the infix expression is scanned using the iterator **i**, which is initialized as **i = 0**.*

***1st Step:** Here i = 0 and exp[i] = ‘a’ i.e., an operand. So add this in the postfix expression. Therefore, **postfix = “a”**.*



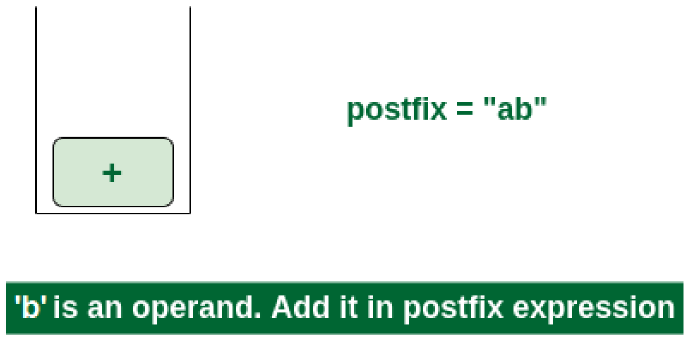
Add ‘a’ in the postfix

***2nd Step:** Here i = 1 and exp[i] = ‘+’ i.e., an operator. Push this into the stack. **postfix = “a”** and **stack = {+}**.*



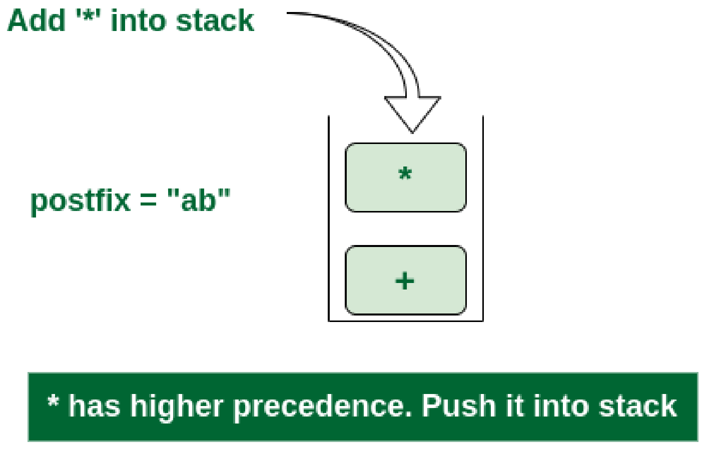
Push '+' in the stack

***3rd Step:** Now i = 2 and exp[i] = 'b' i.e., an operand. So add this in the postfix expression. **postfix = "ab"** and **stack = {+}**.*



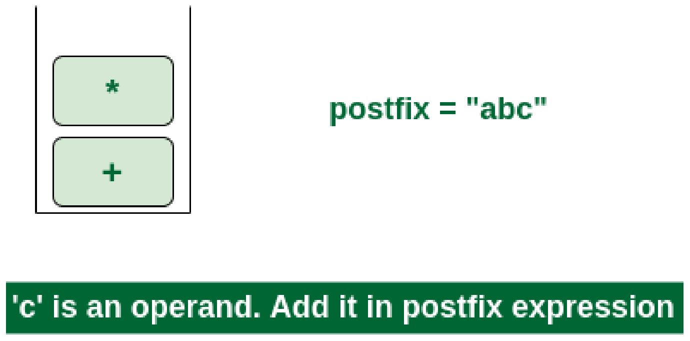
Add 'b' in the postfix

***4th Step:** Now i = 3 and exp[i] = '*' i.e., an operator. Push this into the stack. **postfix = "ab"** and **stack = {+, *}**.*



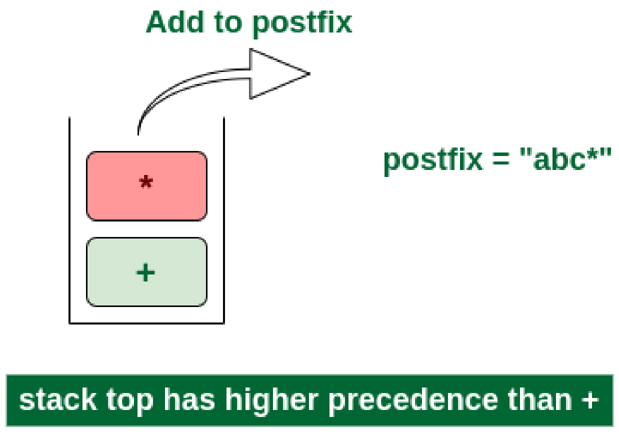
Push '' in the stack*

***5th Step:** Now i = 4 and exp[i] = 'c' i.e., an operand. Add this in the postfix expression. **postfix = "abc"** and **stack = {+, *}**.*



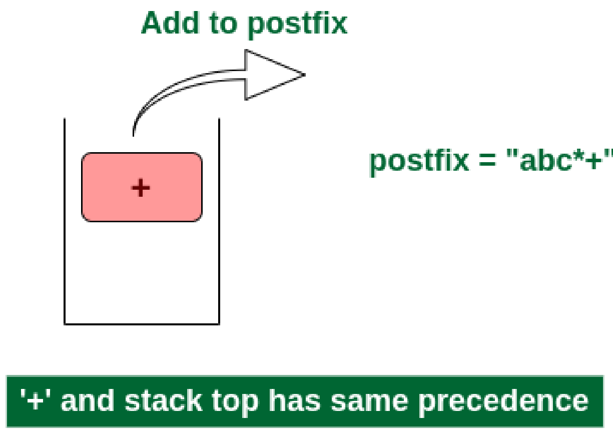
Add 'c' in the postfix

6th Step: *Now i = 5 and exp[i] = '+' i.e., an operator. The topmost element of the stack has higher precedence. So pop until the stack becomes empty or the top element has less precedence. '*' is popped and added in postfix. So **postfix = "abc*"** and **stack = {+}**.*



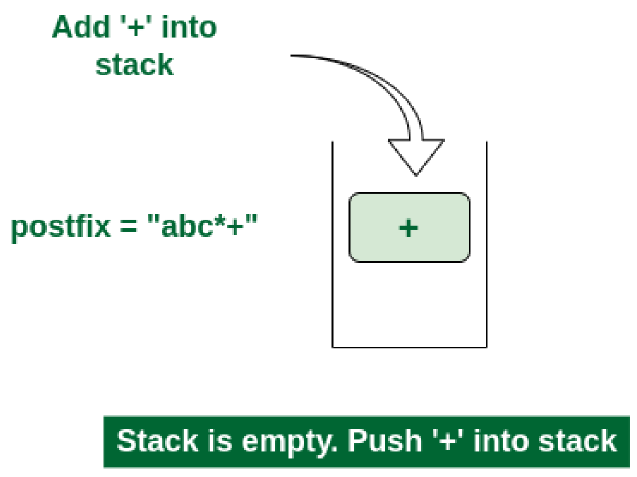
Pop '' and add in postfix*

*Now top element is '**+**' that also doesn't have less precedence. Pop it. **postfix = "abc*+"**.*



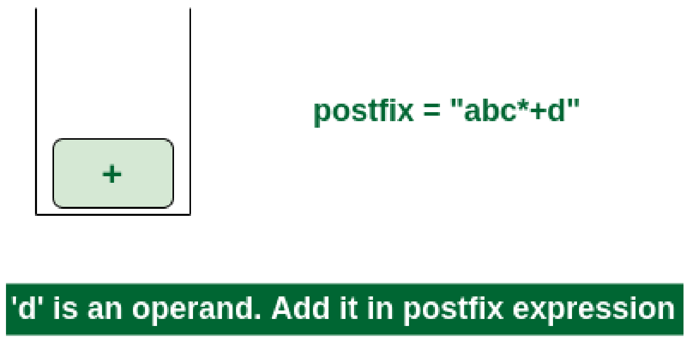
Pop '+' and add it in postfix

*Now stack is empty. So push **'+'** in the stack. **stack = {+}**.*



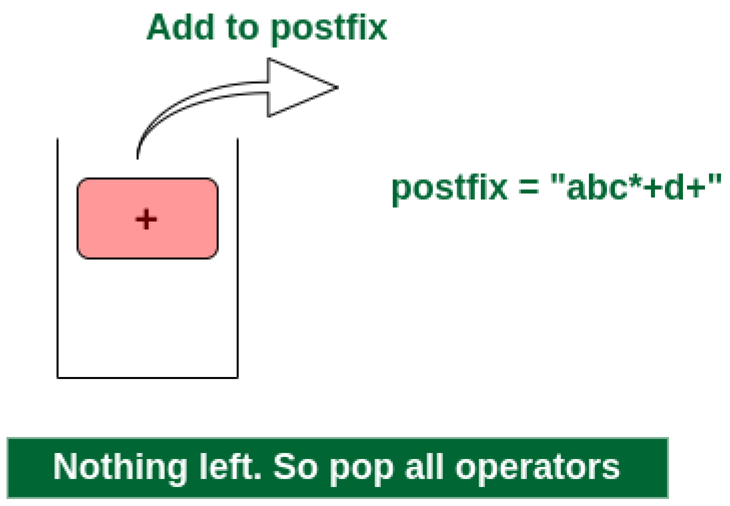
Push '+' in the stack

7th Step: *Now i = 6 and exp[i] = 'd' i.e., an operand. Add this in the postfix expression.* ***postfix = "abc*+d"***.



Add 'd' in the postfix

Final Step: *Now no element is left. So empty the stack and add it in the postfix expression.* ***postfix = "abc*+d+"***.



Pop '+' and add it in postfix

Below is the implementation of the above algorithm:

```cpp
#include <bits/stdc++.h>
using namespace std;

// Function to return precedence of operators
int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

// The main function to convert infix expression
// to postfix expression
void infixToPostfix(string s) {
    stack<char> st;
    string result;

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        // If the scanned character is
        // an operand, add it to the output string.
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c
<= '9'))
            result += c;

        // If the scanned character is an
        // ‘(‘, push it to the stack.
        else if (c == '(')
            st.push('(');

        // If the scanned character is an ‘)’,
        // pop and add to the output string from the stack
        // until an ‘(‘ is encountered.
        else if (c == ')') {
            while (st.top() != '(') {
                result += st.top();
                st.pop();
            }
            st.pop();
        }

        // If an operator is scanned
        else {
            while (!st.empty() && prec(c) < prec(st.top()) ||
                   !st.empty() && prec(c) == prec(st.top())) {
```

```
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }

    // Pop all the remaining elements from the stack
    while (!st.empty()) {
        result += st.top();
        st.pop();
    }

    cout << result << endl;
}

int main() {
    string exp = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}
```

Output

```
abcd^e-fgh*+^*+i-
```

Time Complexity: O(N), where N is the size of the infix expression
Auxiliary Space: O(N), where N is the size of the infix expression