Prefix to Postfix Conversion (Updated)

Problem Statement

Prefix to Postfix Conversion:

Given a prefix expression, convert it into its equivalent postfix expression using a stack-based approach.

Example:

Input Prefix: -+1/*+26483

Output Postfix: 126+4/8*3-

Logic Behind the Code

Logic for Prefix to Postfix Conversion:

1. Use a stack to store postfix expressions.
2. Traverse the prefix expression from **right to left**.
3. If an **operand** is found, push it onto the stack.
4. If an **operator** is found:
   - Pop the top two elements from the stack.
   - Concatenate them in postfix format: `operand1 operand2 operator`.
   - Push the result back onto the stack.
5. After the loop, the stack's top contains the final postfix expression.

C++ Code

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

string solve(string v1, string v2, char ch) {
    return v1 + v2 + ch; // Postfix format: operand1 operand2 operator
}

int main() {
    string s = "-+1/*+26483"; // Prefix expression
    stack<string> st;

    for(int i = s.length() - 1; i >= 0; i--) {
        if(s[i] >= '0' && s[i] <= '9') {
            st.push(string(1, s[i])); // Convert char to string and push
        } else {
            string v1 = st.top();
            st.pop();
            string v2 = st.top();
            st.pop();
            string ans = solve(v1, v2, s[i]);
            st.push(ans);
        }
    }

    cout << "Postfix Expression: " << st.top();
    return 0;
}
```

Dry Run Example (Updated)

Expression: -+1/*+26483

Step-by-step Execution:

1. Read '3' -> Push "3"
2. Read '8' -> Push "8"
3. Read '4' -> Push "4"
4. Read '6' -> Push "6"
5. Read '2' -> Push "2"
6. Read '+' -> Pop "2", "6", merge -> Push "26+"
7. Read '*' -> Pop "26+", "4", merge -> Push "26+4*"
8. Read '/' -> Pop "26+4*", "8", merge -> Push "26+4*8/"
9. Read '1' -> Push "1"
10. Read '+' -> Pop "1", "26+4*8/", merge -> Push "126+4*8/+"

11. Read '-' -> Pop "126+4*8/+", "3", merge -> Push "126+4*8/+3-"

Final Output (Postfix): 126+4*8/+3-

Conclusion

Conclusion:

The given prefix expression is successfully converted to postfix notation using a stack-based approach.

This method ensures that operands and operators are placed correctly without needing extra precedence handling.

Short Notes

Short Notes:

- **Prefix Expression**: Operator appears before operands (e.g., `+AB`).
- **Postfix Expression**: Operator appears after operands (e.g., `AB+`).
- **Traversal Order**: Prefix expressions are processed **right to left**.
- **Stack Usage**: Helps manage operand-operator relationships automatically.
- **Time Complexity**: O(N), where N is the length of the expression.