inToPos:

( 2 + 3 ) \* 4 : operator stack:( output:2 operator stack:( + output:2 3 operator stack:\* output:2 3 + 4 operator stack: output:2 3 + 4 Finally output:2 3 + 4 \*

( 2 + 3 ) \* sin 90 :

sin 90 + 5 : 90 sin 5 +

( ( 2 + 3 ) \* 4 ) + sin 90 : 2 3 + 4 \* 90 sin +

InToPre:

( ( 2 + 3 ) \* 4 ) + sin 90: + \* + 2 3 4 sin 90

2 + 3 \* 4 : Finally output:2 3 4 \* +

( ( 2 + 3 ) \* 4 ) / 6 : Finally output:2 3 + 4 \* 6 /

( ( 2 + 3 ) \* 4 ) ^ 6 : Finally output:2 3 + 4 \* 6 ^

PosToPre:

2 3 4 \* + : Finally output: + 2 \* 3 4

2 3 + 4 \* 6 ^ :Finally output: ^ \* + 2 3 4 6

2 3 + 4 \* 90 sin + : + \* + 2 3 4 sin 90

90 sin 5 +: + sin 90 5

Invalid infix:

+ …… +

.……...sin

… + + ….

…. sin + …

…2 3 ….

….( ( ….. - > for each '(' we must have ' ) ' after it .

…( + … + ) …

….2 ( ….) 2….

….sin (90)

….) ( …..

Test case:

+ 2 + 3 +

2 + sin

2 + + 3 \* 4

2 + sin + 3

2 3 4

2 + ( ( 3 \* 4 ) 🡪//

2 ( 3 + 4 ) 5

2 + ( + 3 \* 4 - )

2 \* sin ( 90 ) 🡪 //

Invalid Postfix

USE THIS ALGURITM:

18

I'm assuming here that what you mean by valid is that executing the code will never underflow the stack and will leave a single value on the stack. If you have a more stringent notion of validity, you'll need a more sophisticated checker.

If you want to check for this kind of validity, it is not necessary to evaluate the string, and you can **use a counter**, not a stack. The counter tracks the number of values that would be on the stack if you evaluated. To simplify, let's suppose you have only literals, binary operators, and unary operators. This algorithm uses a special decrement operation: if when you decrement, the counter goes below zero, the string is invalid:

1. Initialize the counter to 0.
2. When you see a literal, increment the counter.
3. When you see a binary operator, decrement the counter twice, then increment it.
4. When you see a unary operator, decrement the counter, then increment it.
5. At the end of the string, if the counter is 1, and if it never went below 0, the string is valid.

..( …)..

import java.util.\*;

class GFG {

    // function to check if character

    // is operator or not

    static boolean isOperator(char x)

    {

        switch (x) {

        case '+':

        case '-':

        case '/':

        case '\*':

            return true;

        }

        return false;

    }

    // Convert prefix to Postfix expression

    static String preToPost(String pre\_exp)

    {

        Stack<String> s = new Stack<String>();

        // length of expression

        int length = pre\_exp.length();

        // reading from right to left

        for (int i = length - 1; i >= 0; i--)

        {

            // check if symbol is operator

            if (isOperator(pre\_exp.charAt(i)))

            {

                // pop two operands from stack

                String op1 = s.peek();

                s.pop();

                String op2 = s.peek();

                s.pop();

                // concat the operands and operator

                String temp = op1 + op2 + pre\_exp.charAt(i);

                // Push String temp back to stack

                s.push(temp);

            }

            // if symbol is an operand

            else {

                // push the operand to the stack

                s.push(pre\_exp.charAt(i) + "");

            }

        }

        // stack contains only the Postfix expression

        return s.peek();

    }

    // Driver Code

    public static void main(String args[])

    {

        String pre\_exp = "\*-A/BC-/AKL";

        System.out.println("Postfix : "

                           + preToPost(pre\_exp));

    }

}

// This code is contributed by Arnab Kundu

**Algorithm for Prefix to Postfix**:

* Read the Prefix expression in reverse order (from right to left)
* If the symbol is an operand, then push it onto the Stack
* If the symbol is an operator, then pop two operands from the Stack   
  Create a string by concatenating the two operands and the operator after them.   
  **string = operand1 + operand2 + operator**   
  And push the resultant string back to Stack
* Repeat the above steps until end of Prefix expression.
* C++
* Java
* Python 3
* C#
* Javascript

|  |
| --- |
| // JavaProgram to convert prefix to pos |

**Algorithm for Prefix to Infix Conversion:**

Step 1: Start

Step 2: Read the Prefix expression from right to left.

Step 3: If the scanned character is an operand, then push it onto the Stack.

Step 4: If the scanned character is an operator, pop two operands from the

stack and concatenate them in order of 'operand1,operator, operand2'.

Push the result into the stack.

Step 5: Repeat steps 2-4 until the prefix expression ends.

Step 6: Pop all the rest elements of the stack if any.

Step 7: Stop

**Explanation:**

Let's take an example to understand \*a+bc,

1. Reading from right to left, we scan the operands 'c' 'b' respectively and push it into the the stack.
2. '+' is scanned and operands 'c', 'b' are popped and concatenated in form of b+c
3. 'a' is then pushed into the stack.
4. '\*', is encountered and 'a' and string 'b+c' is popped and concatenated to give a string a\*(b+c)
5. Postfix form is obtained: a\*(b+c)