```
int main ( int argc, char *argv[] )
{
        int i;
        for(i=0; i<6; i++)
        { ... }
}
```

([ ]) { ( ) { } }   ---   VALID

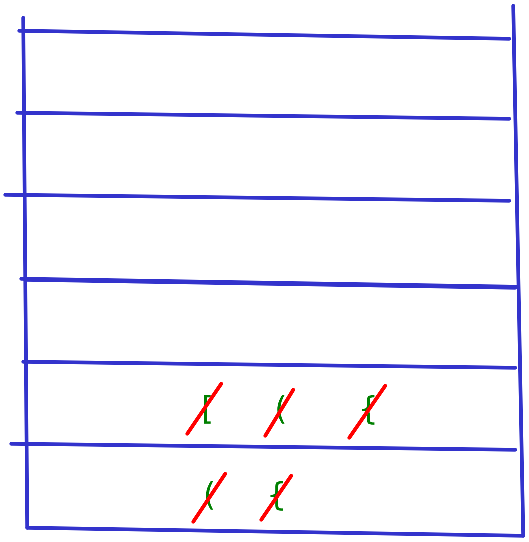
([ )]   ---   INVALID

([ ]   ---   INVALID (Stack not empty)

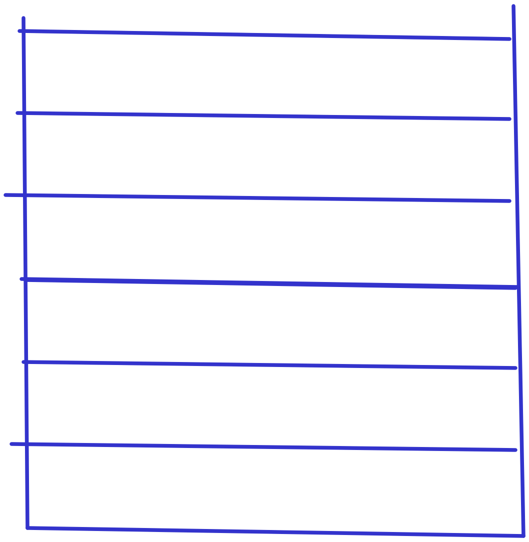
([ )   ---   INVALID

[ ])   ---   INVALID

{ { } } }   ---   INVALID



```
bool bracketCheck(const std::string& s){
        STACK st;
        int i, n;
        char ch, tmp;
        n = s.length();
        for(i=0; i<n; i++){
                ch = s[i];
                if( (ch=='(') || (ch=='{') || (ch=='[') )
                        st.push(ch);
                else if( ch == ')' ){
                        if( st.isEmpty() )
                                return false;
                        tmp = st.pop();
                        if( tmp != '(' )
                                return false;
                }
                else if( ch == '}' ){
                        if( st.isEmpty() )
                                return false;
                        tmp = st.pop();
                        if( tmp != '{' )
                                return false;
                }
                else if( ch == ']' ){
                        if( st.isEmpty() )
                                return false;
                        tmp = st.pop();
                        if( tmp != '[' )
                                return false;
                }
        }
        if( !st.isEmpty() )
                return false;
        return true;
}
```



(( ) ( ))

ct = 1, 2, 1, 2, 1, 0

(( ) ( ) ( )

ct = 1, 2, 1, 2, 1, 2, 1

(( ))) (( )

ct = 1, 2, 1, 0, -1, 0, 1, 0

([ ]) [( ])

ct1 = 1, 0
ct2 = 1, 0

1 + (2 * 3) - 4   ->   Infix expression   (<operand1> <operator> <operand2>)

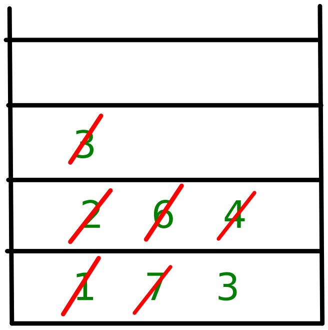
1 2 3 * + 4 -   ->   Postfix expression   (<operand1> <operand2> <operator>)

1 2 3 * + 4 -

1 6 + 4 -

7 4 -

3



op2 = st.pop() = 3
op1 = st.pop() = 2
result = op1 * op2 = 2 * 3 = 6
st.push(result)

op2 = st.pop() = 6
op1 = st.pop() = 1
result = op1 + op2 = 1 + 6 = 7
st.push(result)

op2 = st.pop() = 4
op1 = st.pop() = 7
result = op1 - op2 = 7 - 4 = 3
st.push(result)

1 2 3 * + 4 - $

-2 4 + -5 -

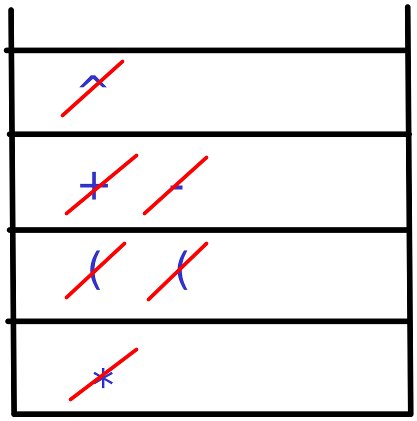
=> (-2 + 4) - (-5)

2 * (1 + 3) * (5 - 4) = 2 * 4 * 1 = 8

2 1 3 + 5 4 - * * = 2 4 5 4 - * * = 2 4 1 * * = 2 4 * = 8

I/p: 2 * (1 + 3 ^ 7) * (5 - 4) $

O/p: 2 1 3 7 ^ + * 5 4 - *



1 + 2 * 3 - 4   ->   Infix expression   (<operand1> <operator> <operand2>)

- + * 2 3 1 4   ->   Prefix expression   (<operator> <operand1> <operand2>)

- + * 2 3 1 4

- + 6 1 4

- 7 4

3

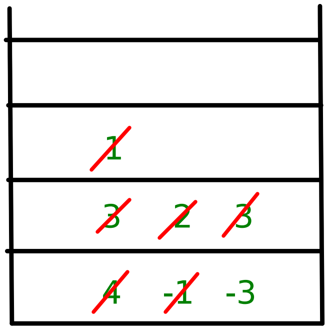
(1 + 2) * (3 - 4)

* + 1 2 - 3 4

(1 + 2) * (3 - 4) = 3 * -1 = -3

* + 1 2 - 3 4

4 3 - 2 1 + * $



op1 = st.pop() = 3
op2 = st.pop() = 4
result = op1 - op2 = 3 - 4 = -1
st.push(result)

op1 = st.pop() = 1
op2 = st.pop() = 2
result = op1 + op2 = 1 + 2 = 3
st.push(result)

op1 = st.pop() = 3
op2 = st.pop() = -1
result = op1 * op2 = 3 * (-1) = -3

Infix Expression: (1 + 2) * (3 - 4)

Reverse of Infix: (4 - 3) * (2 + 1)

Postfix of Reverse: 4 3 - 2 1 + *

Reverse of above: * + 1 2 - 3 4

Infix Expression: (1 + 2 * 3) / (3 - 2 * 4)

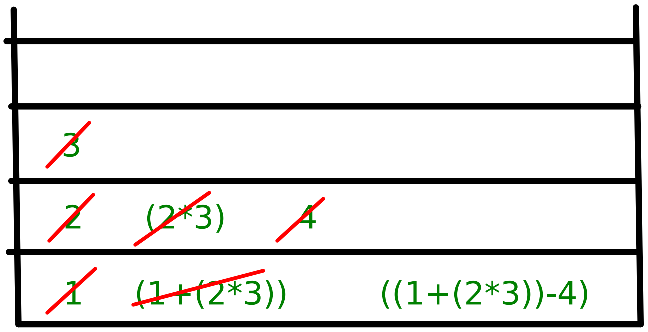
Reverse of Infix: (4 * 2 - 3) / (3 * 2 + 1)

Postfix of Reverse: 4 2 * 3 - 3 2 * 1 + /

Reverse of above: / + 1 * 2 3 - 3 * 2 4

1 2 3 * + 4 - -> Postfix expression (<operand1> <operand2> <operator>)

1 + (2 * 3) - 4 -> Infix expression (<operand1> <operator> <operand2>)



exp2 = st.pop() = 3
exp1 = st.pop() = 2
result = "(<exp1> * <exp2>)" = "(2 * 3)"
st.push(result)

exp2 = st.pop() = "(2 * 3)"
exp1 = st.pop() = 1
result = "(<exp1> + <exp2>)" = "(1 + (2 * 3))"
st.push(result)

exp2 = st.pop() = 4
exp1 = st.pop() = "(1 + (2 * 3))"
result = "(<exp1> - <exp2>)" = ((1 + (2 * 3)) - 4)
st.push(result)

1 2 3 * + 4 - $

((1+(2*3))-4)

Input: (4 * 2 - 3) / (3 * 2 + 1) $

Output: 4 2 * 3 - 3 2 * 1 + /