**1. Problem Discussion :-**

Prerequisite for this question is the previous problem of the same module, i.e. Infix Evaluation. It is very important that before moving to the solution of this problem, you give it a fair try. Let's jump to the problem! In this problem you are provided with an infix expression and you are required to convert it to postfix and prefix and print both these expressions. You don't need to worry about input; it is already managed for you. What are postfix and prefix expressions? Postfix Expression: When operator is post fixed to operands, i.e. operator is placed after operands. For example: Infix expression: x\*y Postfix expression: xy\* Prefix Expression When operator is prefixed to operands, i.e. operator is placed before operands. For example: Infix expression: x\*y Prefix expression: \*xy Sample Input: 2 \* ( 6 - 4 + 8 ) / 3 Sample Output: 2 6 4 - 8 + \* 3 / / \* 2 + - 6 4 8 3 Let's look at the steps:

**2. Approach :-**

First of all, let's take a look at what needs to be done by defining the rules. We will take three stacks one for operator and other 2 for evaluating postfix and prefix expression. Now while scanning the expression, as soon as we get an operand we push that in postfix and prefix stacks. If we get an opening bracket while scanning the expression, we push that in the operator stack. If we get a closing bracket while scanning the expression, we pop out the items of the operator stack until we get an opening bracket. And as soon as we get an opening bracket we pop that out too. If an operator comes then all the operators in the operator stack with greater or equal precedence get popped out until we get an opening bracket or the operator stack empties out. And then we push our current operator in our stack. And whenever an operator is popped out then at the same time two elements are also popped out from prefix and postfix stacks and some work is performed using two strings and one operator. While performing this operation, element that is popped first out of postfix and prefix stacks will be considered value 2 and element which is popped out second will be considered as value 1. Using the three elements i.e. value1, value2 and operator in above step, we push a newer string in both postfix and prefix stack. In the postfix stack, newer strings will be arranged as: value1 + value2 + operator. In the prefix stack, newer strings will be arranged as: operator+ value1 + value2.

Now let's take an example, say we are given an expression: 2+ (5-3\*6/2)

We push "2" into both postfix and prefix stacks and "+" into the operation stack because the stack is empty and none of the conditions is violated.

"(", is pushed into the operation stack.

We push "5" into both postfix and prefix stacks and "-" into the operation stack because the topmost element of operator stack is "(" and therefore none of the condition is violated.

We push "3" into both postfix and prefix stacks and "\*" into operation stack because the top most element of the operator stack is "-", which is of lower precedence.

We push "6" into both postfix and prefix stacks and before pushing "/" into operation stack, we check the top of the operation stack, there we find "\*" at the top which is equal in precedence to "/". So we pop "\*" and two topmost elements of both postfix and prefix stacks i.e. 6 and 3 in this case.

Arrange the new string using value1= "3", value2="6" and operator= "\*" as expected in both postfix and prefix stacks. In postfix stack, newer string will be arranged as: value1 + value2 + operator: "3 6 \*" and push this into the postfix stack. In prefix stack, newer string will be arranged as: operator+ value1 + value2: "\* 3 6" and push this into the prefix stack.

Now we move to the next elements of expression, i.e. "2" which we push into both postfix and prefix stacks.

Last element is ")", on getting this we simply pop out the elements from both the stacks and perform the required operation as we performed before for multiplication. So in this case topmost operator is "/" and 2 elements at the top are "2" and "3 6 \*" in the postfix stack. We get "3 6 \* 2 / " on performing required steps and therefore we push this into the postfix stack. Whereas elements at the top are in postfix stack "2" and "\* 3 6". We get "/ \*3 6 2 " on performing required steps and therefore we push this into the prefix stack. Now the topmost operator is "-" and 2 elements at the top of postfix stack are "3 6 \* 2 /" and "5". We get "5 3 6 \* 2 / -" on performing this operation and therefore we push this into the postfix stack. Whereas elements at the top are in postfix stack "5" and "/ \* 3 6 2". We get "- 5 / \*3 6 2 " on performing required steps and therefore we push this into the prefix stack.

After this, we find "(" at the top of the operand stack, we pop this out and. Now we empty out our operator stack by popping out elements out of these and performing required operations on both the postfix and prefix stack. Here we are left with only one operator which means only one operation more is left to perform.

Now the topmost operator is "+" and 2 elements at the top of postfix stack are "2" and "5 3 6 \* 2 / -". We get "2 5 3 6 \* 2 / - +" on performing this operation and therefore we push this into the postfix stack. Whereas elements at the top are in postfix stack "2" and "- 5 / \* 3 6 2". We get "+ 2 - 5 / \*3 6 2 " on performing required steps and therefore we push this into the prefix stack.

After this, we find "(" at the top of the operand stack, we pop this out and. Now we empty out our operator stack by popping out elements out of these and performing required operations on both the postfix and prefix stack. Here we are left with only one operator which means only one operation more is left to perform.

Now the topmost operator is "+" and 2 elements at the top of postfix stack are "2" and "5 3 6 \* 2 / -". We get "2 5 3 6 \* 2 / - +" on performing this operation and therefore we push this into the postfix stack. Whereas elements at the top are in postfix stack "2" and "- 5 / \* 3 6 2". We get "+ 2 - 5 / \*3 6 2 " on performing required steps and therefore we push this into the prefix stack.

**3. CODE -:**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

String exp = br.readLine();

// code

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

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

Stack< Character> operators = new Stack<>();

for (int i = 0; i < exp.length(); i++) {

char ch = exp.charAt(i);

if (ch == '(') {

operators.push(ch);

} else if ((ch >= '0' && ch <= '9') || (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) {

postfix.push(ch + "");

prefix.push(ch + "");

} else if (ch == '+' || ch == '-' || ch == '\*' || ch == '/') {

while (operators.size() > 0 && operators.peek() != '(' && precedence(ch) <= precedence(operators.peek())) {

char op = operators.pop();

String postval2 = postfix.pop();

String postval1 = postfix.pop();

postfix.push(postval1 + postval2 + op);

String preval2 = prefix.pop();

String preval1 = prefix.pop();

prefix.push(op + preval1 + preval2);

}

operators.push(ch);

} else if (ch == ')') {

while (operators.size() > 0 && operators.peek() != '(') {

char op = operators.pop();

String postval2 = postfix.pop();

String postval1 = postfix.pop();

postfix.push(postval1 + postval2 + op);

String preval2 = prefix.pop();

String preval1 = prefix.pop();

prefix.push(op + preval1 + preval2);

}

if (operators.size() > 0) {

operators.pop();

}

}

}

while (operators.size() > 0) {

char op = operators.pop();

String postval2 = postfix.pop();

String postval1 = postfix.pop();

postfix.push(postval1 + postval2 + op);

String preval2 = prefix.pop();

String preval1 = prefix.pop();

prefix.push(op + preval1 + preval2);

}

System.out.println(postfix.peek());

System.out.println(prefix.peek());

}

public static int precedence(char op) {

if (op == '+') {

return 1;

} else if (op == '-') {

return 1;

} else if (op == '\*') {

return 2;

} else {

return 2;

}

}

}

**4. CODE DISCUSSION -:**

We need a function precedence which we will be using quite often. So first look at that. It will return us the precedence of the particular operator. Since the precedence of the "+" and "-" operator is the same, we return 1. Also the precedence of "\*" and "/" is the same but greater than "+" and "-" so we return 2.

Now we start to write our code in main. Here we first of all define our three stacks, one for postfix, second for prefix and third for operators.

Now we start to write our code in main. Here we first of all define our three stacks, one for postfix, second for prefix and third for operators.

After that we run a for loop on our expression and process its each character one by one by capturing it in a character variable ch.

If the character is "(" then without thinking any further we push this character into the operator stack.

Else if the character is a digit or any variable varying from a-z or A-Z then we push the character into prefix and postfix stacks.

Else if the character is any operator (+, -, \*, /) then we run a while loop and pop out the elements of operator stack until we get "("or an operator with smaller precedence at the top of operator stack, making sure that at any time the stack is not empty. While popping the elements out of the operator stack, we simultaneously pop out two elements of both the other stacks and store them in a variable of type string. In both these stacks, the first element that pops out will be value2 and the second element that pops out will be value1. Since, in the postfix expression the order of arrangement is: value1 + value2 + operator. So, we do the same and concatenate all three popped out elements in the same order and push it in the postfix stack. Similarly, in the prefix expression the order of arrangement is: operator + value1 + value2. So, we concatenate all three popped out elements in the same order and push it in the prefix stack. And when we move out of this while loop we push the ch into the operator stack.

Else if the character is ")" then we run a while loop and pop out the elements of operator stack until we get "(" at the top of operator stack. While popping the elements out of the operator stack, we simultaneously pop out two elements of both the other stacks and store them in a variable of type string. In both these stacks, the first element that pops out will be value2 and the second element that pops out will be value1. Since, in the postfix expression the order of arrangement is: value1 + value2 + operator. So, we do the same and concatenate all three popped out elements in the same order and push it in the postfix stack. Similarly, in the prefix expression the order of arrangement is: operator + value1 + value2. So, we concatenate all three popped out elements in the same order and push it in the prefix stack. And when we move out of this while loop this means that the top element is "(" so we perform pop operation once more on operator stack.

When we come out of the for loop, it's important to empty out the operator stack in case it's not. First of all we pop out the operator out of the operator stack, making sure that at any time the stack is not empty. After popping out the operator we store it in a character op and simultaneously pop out 2 elements of both prefix and postfix stacks and store these in val2 and val1 respectively from both the stacks. As mentioned previously, in the postfix expression, the order of arrangement is: value1 + value2 + operator. So, we do the same and concatenate all three popped out elements in the same order and push it in the postfix stack. Similarly, in the prefix expression, the order of arrangement is: operator + value1 + value2. So, we concatenate all three popped out elements in the same order and push it in the prefix stack. The loop runs until the operator stack is not empty. At last we print postfix.peek and prefix.peek which store our final postfix and prefix expressions.

**5. Analysis -:**

Time Complexity : O(n)

This time complexity is O(n).

Space Complexity : O(n)

The space complexity is O(n).

That was easy. Wasn't it? Our desire to make you learn will remain unsatisfactory if you still have doubts. We strongly recommend you to watch our video lecture on Infix Conversions for clearing any type of doubts. Suggestions and feedback are always welcomed. You can contact us via our website. All the best for a bright future! Happy Coding!