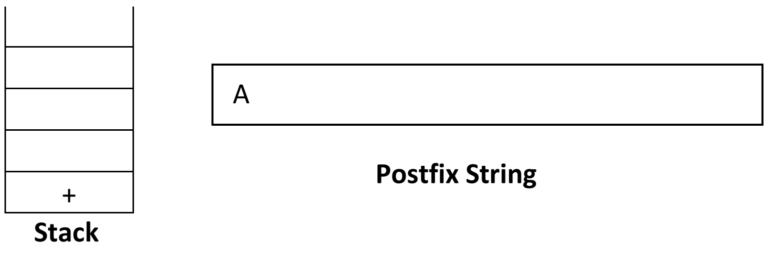
Infix to Post-fix Conversion

Let us see how the algorithm will be implemented using the example we showed in class today:

Infix expression: A + ((B – C) * (E – F) – G) / (H – I)

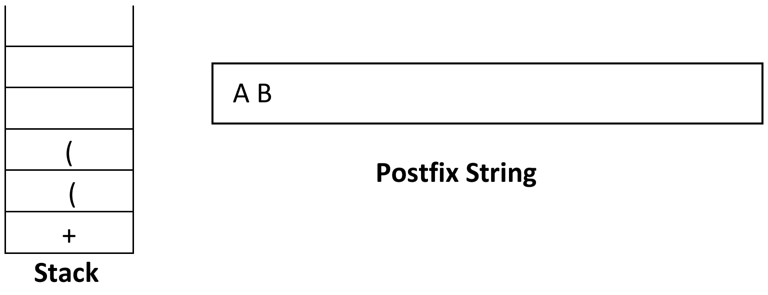
1. Initially the stack is empty and the postfix output string is also empty. Now, we scan the infix expression from left to right. The first character scanned is ‘A’, the next character scanned is ‘+’. Since ‘+’ is an operator and the stack is empty, we just pushed it on top of the stack

Original infix expression: **A +** ((B – C) * (E – F) – G) / (H – I)



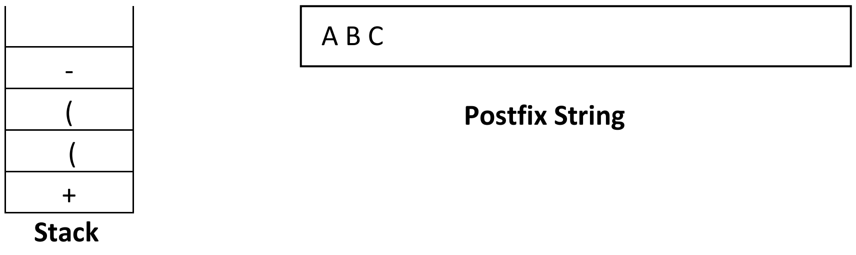
2. Next character is left parenthesis, just push them on top of the stack, followed by character ‘B’ which we will append it at the end of the Postfix String

Original infix expression: **A + ((B** – C) * (E – F) – G) / (H – I)



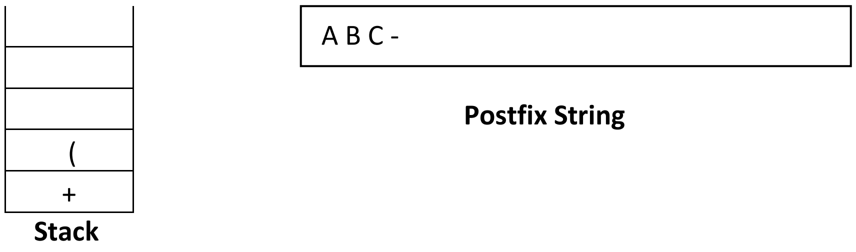
3. Next character is ‘-‘ which is an operator, first we need to check the top of the stack, it is a left parenthesis, so we just push it on top of the stack, we then have character ‘C’, just append it to the postfix string.

Original infix expression: **A + ((B – C**) * (E – F) – G) / (H – I)



4. Next character is a right parenthesis ‘)‘ , whenever we encounter a right parenthesis, we will pop everything out from the stack until we encounter the first left parenthesis on stack, we then stop. (Note: the first encountered left parenthesis on stack will be pop out and discarded)

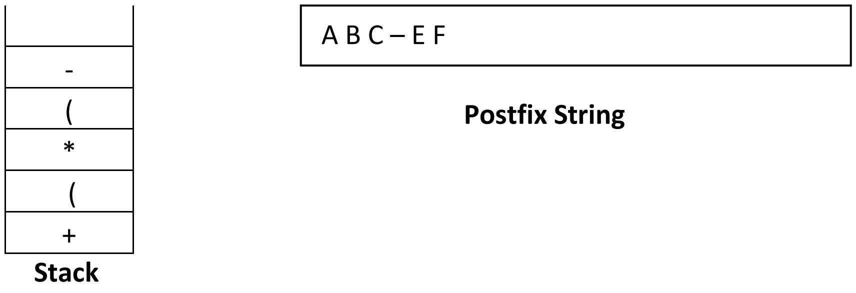
Original infix expression: **A + ((B – C)** * (E – F) – G) / (H – I)



5. Next character is a ‘*’ which is an operator, whenever we push an operator on top of the stack, we will compare it with the element currently is at the top of the stack, for the current case, since it is a ‘(‘, we just push ‘*’ on top of stack.

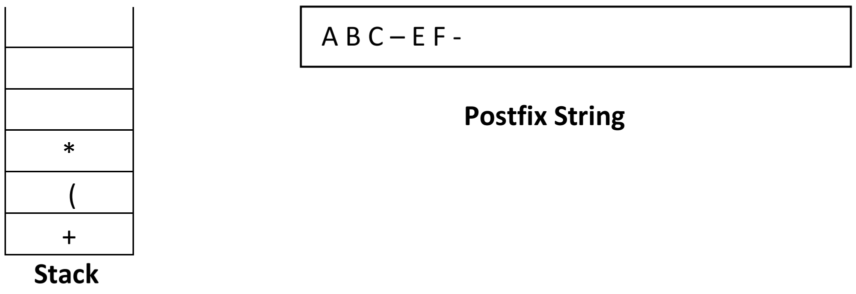
For the next four characters ‘(‘, ‘E’, ‘-‘ and ‘F’, we follow same rule as described above.

Original infix expression: **A + ((B – C) * (E – F**) – G) / (H – I)



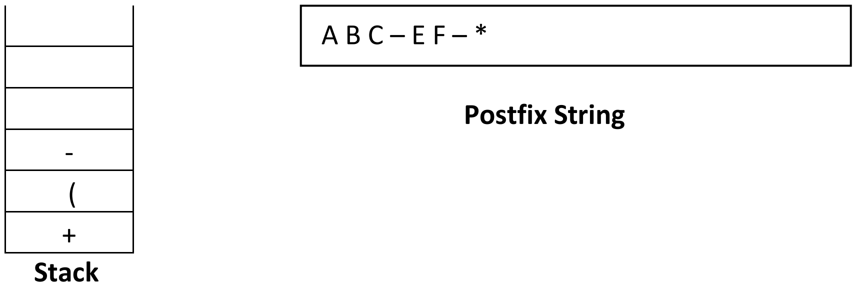
6. Next character is a ‘)’ , we will follow the ruled as described in 4..

Original infix expression: **A + ((B – C) * (E – F)** – G) / (H – I)



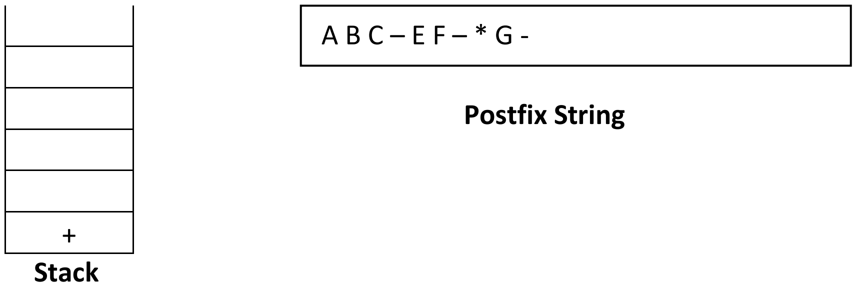
7 [****Important**]: Next character is a '-' which is an operator, we will store it inside the stack, but first we will need to compare it with the element currently at the top of the stack, for our case which is '*', since '*' has higher order of precedence than '-', we will need to pop it out from the stack, append it at the end of the postfix string, we will continue this procedure until we see an operator which has lower precedence than '-' (of course this is impossible for this assignment) or we see the left parenthesis. For more details of this procedure, please check the algorithm's description inside assignment #11.

Original infix expression: **A + ((B – C) * (E – F) –** G) / (H – I)



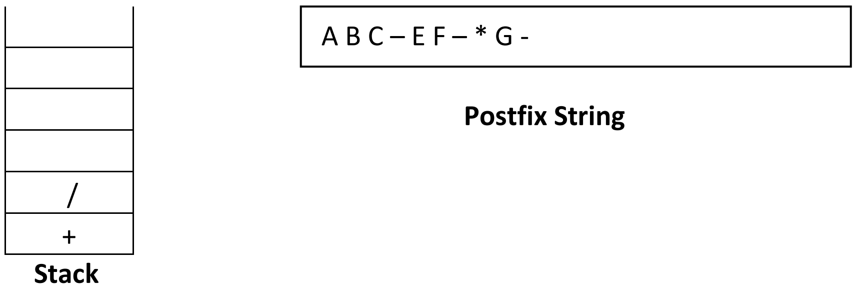
8. For the next two character 'G' and ')', just follow the rule described above, we have

Original infix expression: **A + ((B – C) * (E – F) – G)** / (H – I)



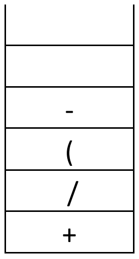
9. For the next character '/' which is an operator, since it has higher order of precedence than the top element of the stack ('+'), we just push it on top of the stack.

Original infix expression: **A + ((B – C) * (E – F) – G) /** (H – I)



10. For the next 4 characters '(', 'H', '-'and 'I', just follow similar rule as described above.

Original infix expression: **A + ((B – C) * (E – F) – G) / (H – I**)



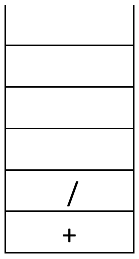
Stack

A B C – E F – * G – H I

Postfix String

11. When we scan the last character which is a ')', follow the rule as described in 4, we will have:

Original infix expression: **A + ((B – C) * (E – F) – G) / (H – I)**



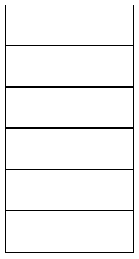
Stack

A B C – E F – * G – H I -

Postfix String

12. We then pop the remaining operators out from the stack one-by-one and append it at the end of the postfix string, we will have:

Original infix expression: **A + ((B – C) * (E – F) – G) / (H – I)**



Stack

A B C – E F – * G – H I - / +

Postfix String