Reverse Polish Notation (RPN)

* Infix expressions are what we’re typically used to and they’re written in the from AOB
  + Where A and B are numbers
  + And O is an operator (+, -, \*, /)
* Postfix expressions are a way of writing expressions that eliminate the need for brackets and any other order of execution rules (e.g. BIDMAS). They’re written in the from ABO
  + Where A and B are numbers
  + And O is an operator (+, -, \*, /)

**Converting From Infix to Postfix:**

The conversion makes uses of a stack as well as a queue.

Each operator has precedence:

* () = 0
* + = 2
* - = 2
* \* = 3
* / = 3
* ^ = 4

E.g. the infix expression “(10 + (2\*8))-3” is equivalent to the postfix expression “10 2 8 \* + 3 –“

The best way to go about solving this is by using an algorithm called the shunting-yard:

While there are characters to be read,

Read a token.

If it’s a number add it to queue.

If it’s an operator

While there’s an operator on the top of the stack with greater precedence

Pop operators from the stack onto the output queue

Push the current operator onto the stack

If it’s a left bracket push it onto the stack

If it’s a right bracket

While there’s not a left bracket at the top of the stack

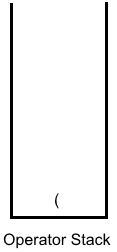
Pop operators from the stack onto the output queue

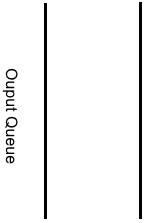
Pop the left bracket from the stack and discard it

While there are operators on the stack, pop them to the queue

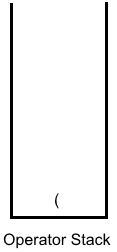
Output = dequeue the queue

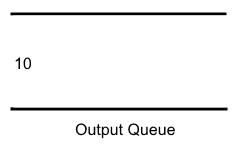
How to get the answer of a postfix expression:

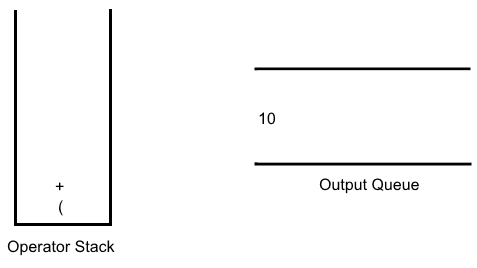
1. Look at the first item in the equation “(“. As it’s an operator it’s pushed onto the operator stack.

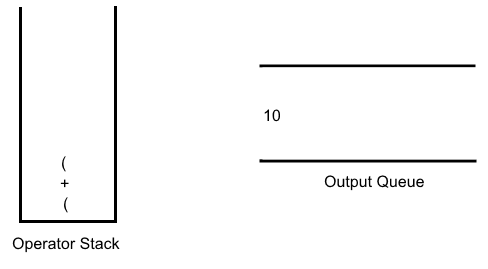


1. The next item is the number 10 so it’s enqueued onto the output queue.

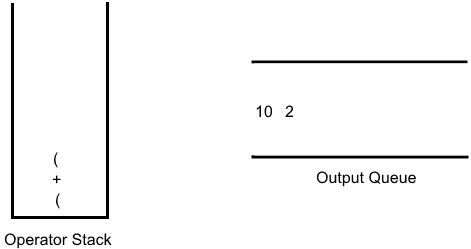




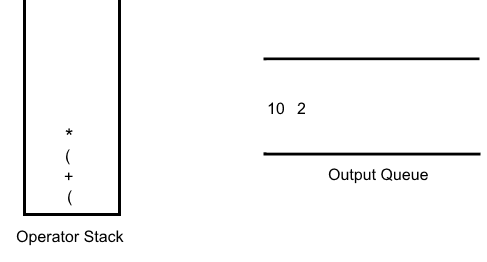
1. The next item is a “+”however, as there’s an operator on the stack, as “+” has a higher precedence than the item on the stack it’s push onto the stack.
2. The next item is “(“so it is pushed onto the operator stack.



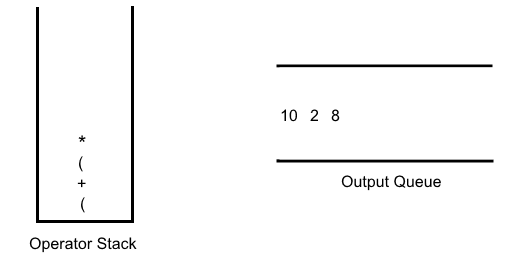
1. The next item is the number 2 so it’s enqueued onto the output queue.



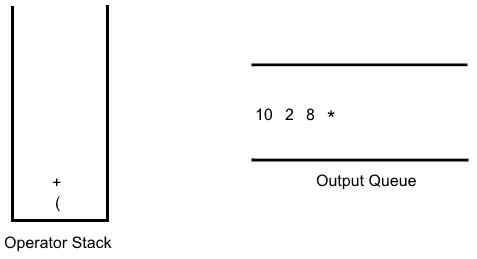
1. The next item is a “\*”however, there’s an operators on the stack, as “\*” has a higher precedence than the item on the stack it’s pushed onto the stack.



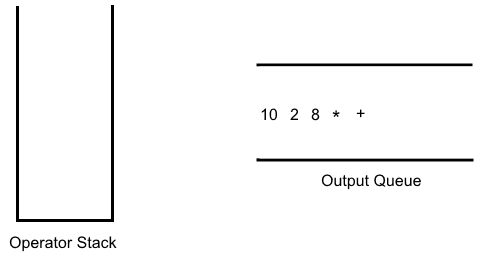
1. The next item is the number 8 so it’s enqueued onto the output queue.

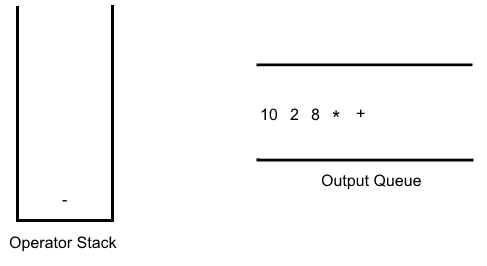
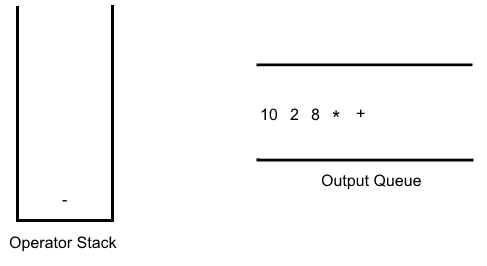


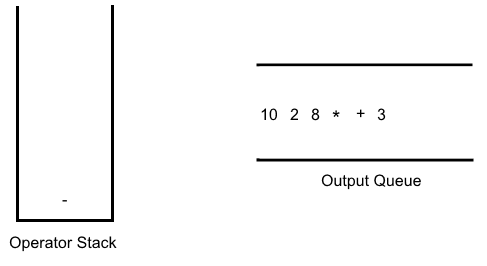
1. The next item is “)”. As it’s a closed bracket everything up to the first open bracket in the stack is popped off and enqueued onto the output queue. The first open bracket is discarded.



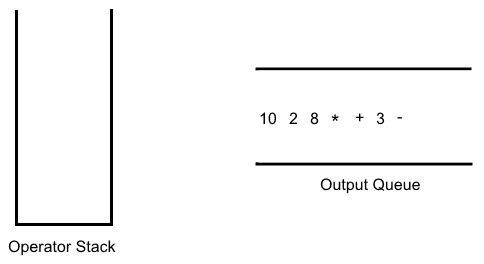
1. The next item is “)”. As it’s a closed bracket everything up to the first open bracket in the stack is popped off and enqueued onto the output queue. The first open bracket is discarded.



1. The next item is “-“. As the stack is empty it is pushed onto the stack.
2. The next item is the number 3 so it is enqueued onto the output queue.



1. As there are no more items in the equation, all the items that are on the operator stack are popped and enqueued onto the output queue.



1. Finally, all the items on the output queue are dequeued to form the result “10 2 8 \* + 3 –“.

**Converting From Postfix to Infix:**

The conversion makes uses of a stack as well as a queue.

E.g. the postfix expression “5 6 9 / - 5 8 / 2 - \*” is equivalent to the infix expression “(5-(6/9))\*((5/8)-2)”

To solve this problem you can use this algorithm:

While there are characters to be read,

Read a token.

If it’s a number push it to the stack

If it’s an **operator**

Pop item off the stack and store in variable **b**

Pop next item off the stack and store in variable **a**

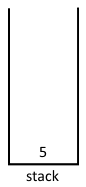
Push onto the stack “**(a operator b)”**

When all characters are read

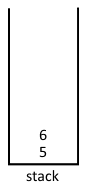
Output = pop all items off the stack

How to get the answer of an infix expression:

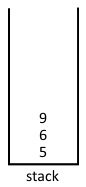
1. Look at the first item in the equation “5“. As it’s a number, push it onto the stack.



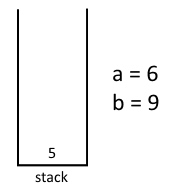
1. The next item is “6”. As it’s a number, push it onto the stack.



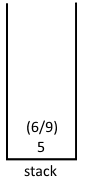
1. The next item is “9”. As it’s a number, push it onto the stack.



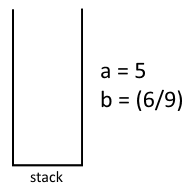
1. The next item is “/”. As it’s an operator, you pop off the first item from the stack and set it to variable b, then you pop of the next item and set it to variable a.



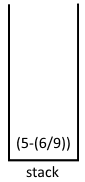
1. You then push onto the stack “(a operator b)”.



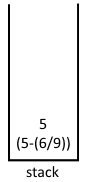
1. The next item is “-”. As it’s an operator, you pop off the first item from the stack and set it to variable b, then you pop of the next item and set it to variable a.



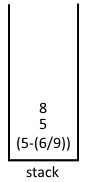
1. You then push onto the stack “(a operator b)”.



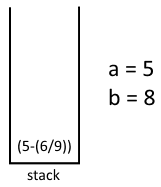
1. The next item is “5”. As it’s a number, you push it onto the stack.



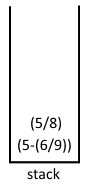
1. Then next item is “8”. As it’s a number, you push it onto the stack.



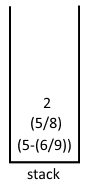
1. The next item is “/”. As it’s an operator, you pop off the first item from the stack and set it to variable b, then you pop of the next item and set it to variable a.



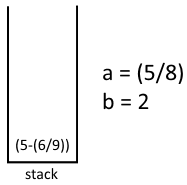
1. You then push onto the stack “(a operator b)”.

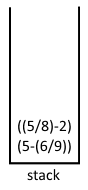


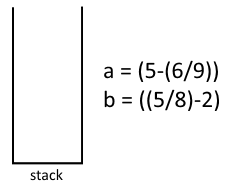
1. The next item is “2”. As it’s a number, you push it onto the stack.

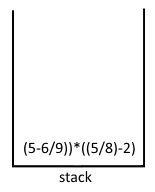


1. The next item is “-”. As it’s an operator, you pop off the first item from the stack and set it to variable b, then you pop of the next item and set it to variable a.



1. You then push onto the stack “(a operator b)”.
2. The next item is “-”. As it’s an operator, you pop off the first item from the stack and set it to variable b, then you pop of the next item and set it to variable a.



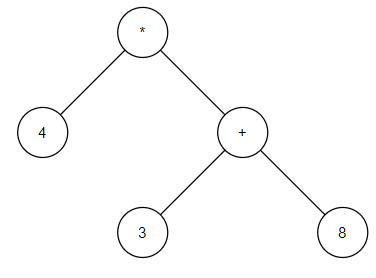
1. You then push onto the stack “(a operator b)”.
2. As there are no more items in the expression, you pop the item off the stack and that gives you your answer for the conversion from postfix into infix.

RPN is useful in computer science as:

* It is unambiguous and doesn’t require the use of brackets
* It can be worked out using a stack.

Computers are not able to work out the value of an infix equation (e.g. (5\*(9-1))). Instead they need calculations to be simpler and to be able to be read from left to right, so the equation is turned into a postfix expression to help do so.

A binary tree can be used to convert an infix expression into a postfix notation by doing a postorder traversal to the binary tree. E.g. Converting 4\*(3+8).



If you were to do a post order traversal on this tree you would get the answer “4 3 8 + \*” which is the postfix expression for “4\*(3+8)”.