Stacks

A stack is a data structure that stores information, arranged like a stack. We have seen stacks before when we used the stack model to trace through recursive programs. The essential idea is that the last item placed into the stack is the first item removed from the stack, or LIFO(Last In, First Out) for short.

Here are the two operations that are used to modify stack contents:

push – This pushes an item onto the top of the stack
pop – This pops off the top item in the stack and returns it.

Here are other operations that access information from a stack, but do not modify it:

empty – typically implemented as a boolean function that returns true if no items are in the stack.
full – returns true if no more items can be added to the stack. In theory a stack should never become full, but any actual implementation of a stack has a limit on the number of elements in can store.
top – Simply returns the value stored at the top of the stack without popping it off the stack.

Note: Each of these would take in the data representation of a stack as a parameter, if these three were implemented as functions.

Keep in mind that a push can only be done if the stack isn't full, and a pop can only be done on a non-empty stack.

A stack is essentially an abstraction. It is easy to follow a stack model with these basic instructions. Let's take a look at an example. Consider tracing the contents of a stack through these operations:

push(7)
push(3)
push(2)
pop
pop
push(5)
pop

So, you might ask what a stack is useful for. There are many examples outside of the scope of this class, but for now, the main problem we will show you to solve by use of stacks is evaluating post-fix expressions, or as they are often known (and I have NO IDEA WHY!), expressions in reverse polish notation.

As the term post-fix indicates, the operator in a post-fix expression comes last instead of in the middle, thus, the expression we know as 3 + 5 is really 3 5 + in post-fix. Initially, this notation seems odd to people, but consider that our typical notation is in-fix, which is analogous to in some ways to the inorder tree traversal I spoke of last time.

In a similar manner, post-fix notation is analogous to a postorder tree traversal. Thus, the difference between infix and postfix notation is similar to the difference between an inorder and postorder tree traversal. Let me illustrate this with a simple example.

Now, let's look at how a stack can be used to evaluate a post-fix notation:

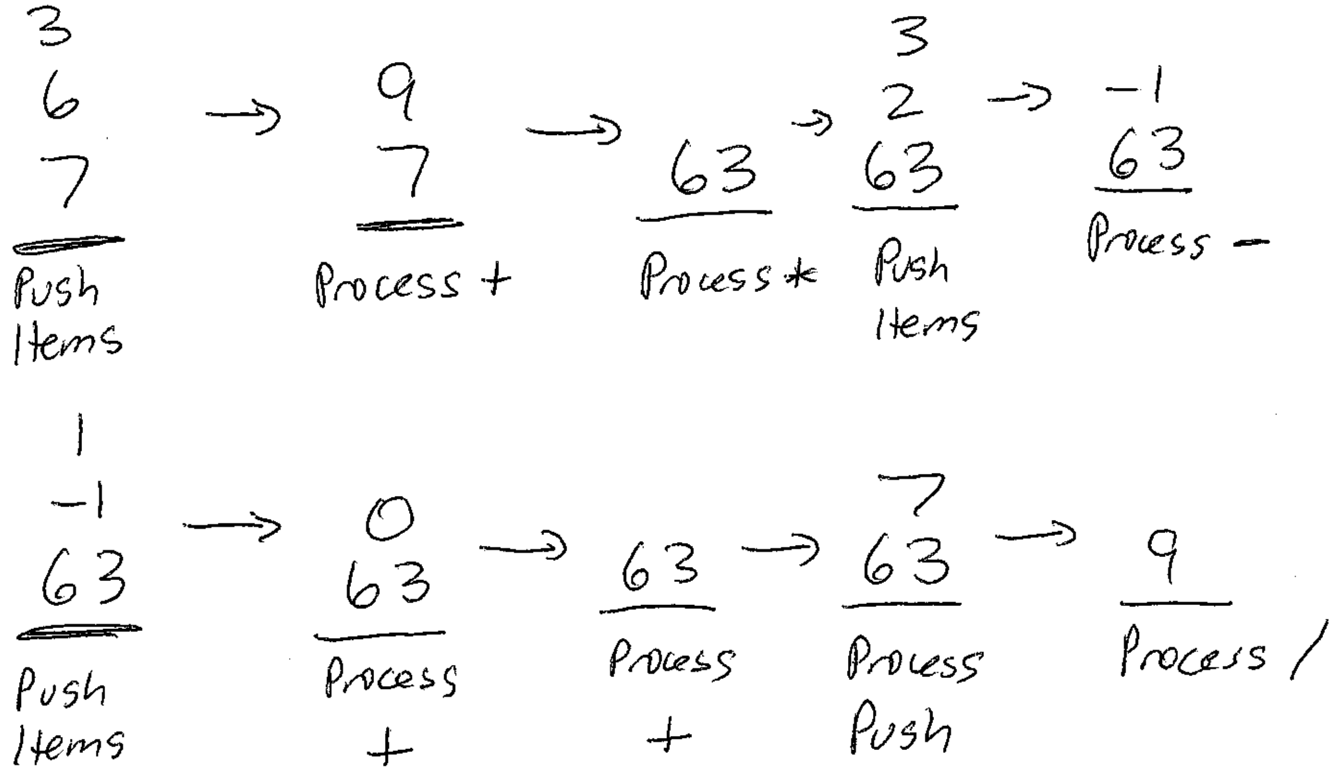
Consider a post-fix expression such as:

7 6 3 + * 2 3 – 1 + + 7 /

Here are the rules:

1) **Each number gets pushed onto the stack.**
2) **Whenever you get to an operator OP, you pop off the last two values off the stack, s1 and s2 respectively. Then you push the value s2 OP s1 back onto the stack. If there are not two values to pop off, the expression being evaluated is not in valid post-fix notation.**
3) **When you are done, you should have a single value left on the stack that the expression evaluates to.**

Let's go through this example, step by step:



Practice Problems

Use a stack to evaluate these post-fix expressions. Are each of these valid? If not, what is left on the stack at the end of evaluating the expression, or is an illegal operation attempted?

1) 1 2 3 + - 2 1 * 7 + +

2) 8 7 + * 2 3 6 1 / * +

3) 8 2 – 7 + 5 4 *

Converting an Infix Expression to Postfix

Consider the following infix (the notation you are used to) expression:

(7*(6+3) + (2-3) + 1) / 7

We can use a stack to convert this expression into its postfix equivalent. Here are the steps:

Process each token one by one doing the following with the token as you process it. (Note that you will have an output expression you are forming as you go along.)

1) For all operands, automatically place them in the output expression.

2) For an operator (+, -, *, /, or a parenthesis)

> **If the operator is an open parenthesis, push it onto the stack.**
>
> **Else if the operator is an arithmetic one, then do this:**
>
> > **Continue popping off items off the stack and placing them in the output expression until you hit an operator with lower precedence than the current operator or until you hit an open parenthesis. At this point, push the current operator onto the stack.**
>
> **Else**
>
> > **Pop off all operators off the stack one by one, placing them in the output expression until you hit the first(matching) open parenthesis. When this occurs, pop off the open parenthesis and discard both ()s.**

Practice Problems

Use a stack to convert the following expressions in infix notation into postfix notation:

1) ((4 + 6) / (2 * 3 - 1* 4) + 3 / (2 - 1 % 2)) / 4

2) 9 + 3*4 - 36 / (((1 + 2) + 3*4) / 7 + 7)