Array Implementation of a Stack (stack.c)

In a simple array implementation of a stack (where we impose a strict limit on the maximum size of the stack), we need two components in the struct to store the stack:

1) An array that stores the items
2) An integer that stores the index of the top of the stack.

For example, a stack that has 3 on the bottom, with 8 on top of it and 4 on top of that at the top of the stack would be stored in a such a struct as follows:

index	0	1	2	3	4	5	6	7	8	9
items	3	8	4							

top 2

With such a storage system, we must discuss implementing the following functions:

0) initialization
1) push
2) pop
3) top
4) isEmpty
5) isFull

Here are the key issues that come up with each function:

0) Set top to -1 to indicate that no items are currently in the stack.

1) Increment top, store item to push in new top index.

2) Decrement top and return the value that was stored in the old top index.

3) Just return what is stored in index top, as long as the stack isn’t empty.

4) top would be -1 if the stack is empty.

5) top would be the size of the array minus 1 if it’s full.

Linked List Implementation of a Stack

We can essentially use a standard linked list to simulate a stack, where a push is simply designated as inserting into the front of the linked list, and a pop would be deleting the front node in a linked list.

There are multiple ways to achieve this:

1) Create just one struct for the stack which essentially acts similar to the struct defined for use with linked lists.

2) Use an existing linked list data structure, and simply borrow an instance of that, creating a new struct that has a pointer to a linked list as a component.

In my example, I chose the former method. A good exercise for you would be to see if you can adapt my code to work the other way, using the linked list code we have already gone over.

Some issues to think about when writing this code:

1) Making sure changes get reflected in the original stack struct. (This is done using **.) The other way would be to return a pointer.

2) Checking to see if memory was allocated properly for a new node.

Most of the rest of the code (even though it looks long), is reasonably straight-forward.

Exercise to use the Stack Data Structure Presented

Plan the outline to a program that will read in a postfix expression and evaluate it. For this exercise, assume that the postfix expression read in is valid.

If you come up with a good outline, start filling in some of the implementation details.

For your outline, assume you have a function that reads in each "token" as a string, and automatically converts it to either an integer or an operator (which is a character), and you can simply proceed based upon this assumption.