Jeffrey Wan/Module 2/Prompt Questions/Spring 2020

Jot down short answers for each prompt question as you go along. Review and update your answers after finishing and submit before Tuesday midnight. Click on "Module 2 - Discussion Prompt Questions" above to submit.

The prompt questions for this module are:

1. **Identify some key characteristics of stacks or stack operations.**

- items are inserted on or removed from the top of the stack. It's a Last In First Out ADT.
- The Stack could be implemented with a linked list, array, or a vector (unsure what a vector is).

2. What methods are *required* to implement a stack?

The stack push operation inserts an item on the top of the stack. The stack pop operation removes and returns the item at the top of the stack. I think these two moethods are the mandatory ones.

3. What methods might be helpful in managing a stack, but are not required?You might consider methods that could be implemented as a combination of methods identified in the last question.

Optional methods include Peek, IsEmpty, and getLength.

ReplaceTop would be an interesting method to have. It pops first and pushes an item onto the stack, thus replacing the top of the stack.

Empty would be interesting and just uses a while loop popping out items until the getLength returns 0. That empties out the stack.

4. Name an example (or two) of an application for which stacks would be helpful, and explain why. Consider real-life applications, not just classic algorithms.

-Order processing stacks would be useful to use in any ecommerce platform and could be implemented using a stack. A queue might be better, but I don't see why orders couldn't be put on the stack with the last in order being processed first. True, a first in and first out queue might be better… but a stack still processing orders correctly when time doesn't really matter.

Maybe a better example is the memory system of a computer. Last in first out is how operations need to handled by a computer. Imagine a function that calls other functions. The last function called needs to be resolved first before the other previous functions can resolve.

5. Name an example (or two) of an application for which stacks would NOT be helpful, and explain why. Consider real-life applications, not just classic algorithms.

When order matters, a stack might not be useful. For example a stock trading algorithm that processes stack trades. Since multiple trades might target the same stock share… the first one in needs to be processed first. The last order in cannot bypass the other previous stock orders otherwise people will be angry that first orders weren't getting priority. So, a queue might be needed and not a stack.

When order matters, beware of using a stack.

6. Have you used prefix or postfix notation in the past? In what context? Did you find it useful?

I have but I haven't thought about it much. I think I've seen it when writing assembly or something? From wiki:

> Prefix notation is a mathematical notation. It is a way to write down equations and other mathematical formulae. Prefix notation is also known as Polish notation. The notation was invented by Jan Łukasiewicz in 1920. He wanted to simplify writing logic equations. When prefix notation is used, no grouping elements (like parenthesis) are needed. With prefix notation, the function is noted before the arguments it operates on.

I guess prefix notation is something like this: + 3 4 which equals 7? Infix, our standard notation, is 3 + 4 = 7

Apparently postfix is easy for a computer to evaluate since postfix can be directly translated to code if you use a stack-based processor; you simply take the operator on the right of a postfix expression and that immediately tells a computer how to evaluate the next two operands. Postfix expression doesn't has the operator precedence also. In prefix notation, * + 5 6 3" is (5+6)*3, and cannot be interpreted as 5+(6*3), whereas parenthesis is required to achieve with infix.

I guess the trick is how do you convert some infix like a + b * c + d to postfix or prefix so that operator precedence does not matter and so you don't need brackets.

7. What are some pros and cons of choosing to implement a stack using an array versus a list implementation?

List implementation: In a linked list with just a head pointer, the time cost to prepend a value to the list is O(1) — we create the new element, set the pointer to point to the old head of the linked list, then update the head pointer to point to the new element. The memory overhead of the linked list is usually O(n) total extra memory due to the storage of an extra pointer in each element. The pointer takes up memory so there is always O(n) memory overhead for the list structure.

Array implementation: The first element (usually at the zero offset) is the bottom, resulting in array[0] being the first element pushed onto the stack and the last element popped off. We also need to keep track of how big the array is so we know where to put the next item.
In an array-based implementation we maintain the following fields: an array A of a default size (≥ 1), the variable top that refers to the top element in the stack and the capacity that refers to the array size.

I think the two implementations are very similar?