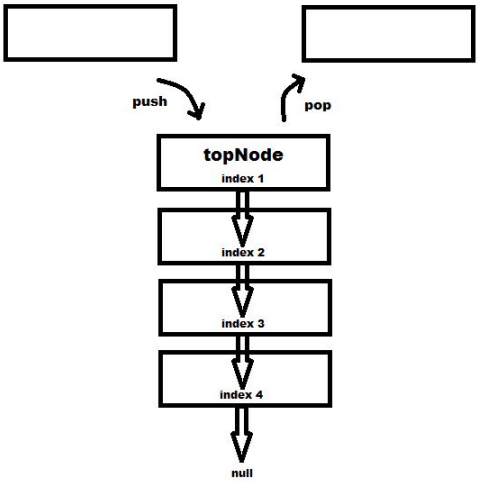
CSCI 1933 • Fall 2015

Introduction to Algorithms and Data Structures

Lab 8

Introduction

In class you have been learning how to implement data structures called linked lists. For this lab, we will use a similar strategy to implement abstract data types called stacks.



You might think of a stack as a linked list in which one member is figuratively stacked on top of the others. Removing elements from a stack is done on a last-in, first-out basis: the most recently added elements are removed first.

Imagine a simple stack made up of two objects, a and b. Say that we add a to our stack and then b. If we go to remove a member of the stack, we must remove b before we can reach a. It is as if b is stacked on top of a, blocking access to it.

Stacks are used for many fundamental computer applications. In computer architecture for example, stacks are commonly used to allocate and access memory.

When adding an element to the top of a stack, we say that we are "pushing" the element. When removing an element, we say that we are "popping" it.

Your Tasks

Your task is to implement push, pop and a number of other methods in the `MyStack.java` class provided to you. `MyStack.java` is very similar to Stack.java, which is part of the Java Class Library (just like ArrayList.java and LinkedList.java).

You will instantiate each stack object as a member of the `Node` class defined in `MyStack.java`. Each `Node` has a `next` attribute which we will use for pointing to the `Node` directly beneath it in the stack.

`MyStack` includes a `topNode` variable which will keep track of the top `Node` in the stack at all times. Calling `topNode.next` will return the second node in the stack, and so forth.

To complete this lab, do the following:

1. In IntelliJ, make new project named Lab 8.
2. Import the following files into your Lab 8 project:
    a. `MyStack.java`
    b. `MyStackTest.java`

3. Add JUnit to your project (e.g. by putting the cursor over `TestCase` in MyStackTest.java and selecting "Add JUnit to project" (or something close to that, depending on the IntelliJ version).
4. Implement the push method

   The `push` method takes one parameter: the item to be placed on the stack. This item should be used to create a new `Node` which will be the new `topNode`. Make sure that the new `Node` points to the `Node` beneath it in the stack.

   - Create a new `Node` object with the provided `item` parameter.
   - Make sure that `newNode.next` points the appropriate member of the current stack.
   - Update the `topNode` variable to keep track of the stack top.
   - Increment the stack size by one. Notice that `MyStack` has an instance variable named `size`.
5. Implement the `pop` method.

   The `pop` method takes zero parameters and returns the current top item in the stack.

   - Check whether the stack is empty, and return `null` if it is.
   - Otherwise, extract the top item by calling `topNode.data`.
   - Remove the `topNode` from the stack and the save the data from the node.
   - Update the `topNode` variable to point to the new stack top.
   - Decrement the stack size by one only if there was an item to "pop".
   - Return the data you saved when you removed the old `topNode`.

6. Implement the `contains` method. This method works identically to the contains method for ArrayLists and LinkedLists that we've discussed in class.
7. Run the unit tests in `MyStackTest.java` to ensure that you implemented these methods correctly. The `testContains` test will not pass unless you choose to do the optional extra work for this lab.
8. Implement the `isEmpty` and `numContains(Object o)` (short for `numContains` hereafter) methods. `isEmpty` method checks if the stack object contains any element. `numContains` methods returns the count of a specific element `o` contained in the stack.
9. Write unit tests for `isEmpty` and `numContains` that establish that they're working. Use `System.out` to print out reasonable status indicators of what is being tested.
10. What is the efficiency of your `numContains` method (i.e. is it O(n), O(n^2))? Once you've made your assessment, do the following:
    a. Write a new class with a `main` method that generates 1,000,000 random integers and places these numbers in a new stack (hint: use the `Random` class).

b. Add a counter to your `numContains` method that counts the number of `equals` operations that occur (Hint: you could use a field in `MyStack.java`, although there are other approaches too).

c. For 10 random numbers, run `numContains` on your 1,000,000-item stack and print out the number of `equals` operations performed. Was your efficiency assessment correct?