CMPS 1600 Introduction to Computer Science II – Spring 14

11/24/14

2. Homework

Programming portion (problems 2 and 3(a)) due **Tuesday 2/4/14** at 11:55pm on Blackboard.
Written portion (problems 1 and 3(b)) due **Wednesday 2/5/14** at the beginning of class.

Please zip the (Eclipse) project directory for this homework, and use the following naming convention for the name of the project (and directory): `lastName_firstName_hw2`. **In order to receive any credit for the programming portions, you are required to thoroughly comment and test your code.**

1. **Stack specification (5 points)**
   Consider the `push` method in the array-based `Stack` class that we have covered in class.
   (a) (1 point) In words, what is the desired functionality of this method? Make sure to take the limited capacity of the stack into account.
   (b) (1 point) On paper, provide modified Java code for the `push` method that provides the desired functionality from part (a). Your method should throw an exception to handle capacity limitations.
   (c) (2 point) Please give the input specification and output specification for the `push` method from part (b). Use logical formulas as well as pictures/diagrams to explain your specifications.
   (d) (1 point) Now consider the `DynamicStack` class that implements a stack using a linked list. How do the input and output specifications for the `push` and `pop` methods change, if at all? Justify your answer.

2. **Dynamic Array (7 points)**
   We would like to implement an array-like data structure that automatically grows in size as necessary. For simplicity, it will store `String` objects for now. Such a *dynamic array* should have the following functionality:
   - There should be a method `set(i, s)` that stores the String `s` at index `i`. Just as in an array, indices should start at `0`.
   - There should be a method `get(i)` that returns the String stored at index `i`, or `null` if nothing has been stored at index `i`.
   - The `toString()` method should be implemented, to return a String representation of the whole data structure for test purposes.

   Implement a class `DynamicArray` that internally uses an array to implement the desired functionality.
   - Use a `DEFAULT_CAPACITY` to allocate an initial array of `String` objects.
   - The method `set(i,s)` should store the String `s` physically at position `i` in your array, and if the capacity of the array is too small, then the array should be resized to double its size.

FLIP OVER TO BACK PAGE ⟹

- In order to resize the array, you need to allocate a new array of double the size, and copy the previous array values over. You could write a private method to do that.
- Note that Java initializes the array values with `null`. So, if no String has been stored at a particular index `i` in the array yet, then the value will be `null` by default.
- Annotate your methods with their worst-case runtimes.

3. **Queue from Two Stacks (8 points)**
A first-in first-out (FIFO) queue supports the following functionality: `enqueue(x)` adds element `x` to the end of the queue; `dequeue()` returns the element from the front of the queue. You will implement a `QueueFromStacks` class that implements the queue functionality using two stacks.

   (a) (6 points) A queue can be implemented using two stacks `stackA` and `stackB` as follows: In order to enqueue an item, push it onto `stackA`. In order to dequeue an item, pop the top item from `stackB`; but if `stackB` is empty, first pop all elements from `stackA` and push them onto `stackB`, and *afterwards* pop the top item from `stackB`.

   Implement a `QueueFromStacks` class that uses two stacks to implement the `enqueue` and `dequeue` methods of a queue of integers this way. For this, use either the array-based stack class or the linked list-based stack class that we implemented in class. You will need the `push` and `pop` methods, and an additional `isEmpty()` method that returns `true` if the stack is empty; please add the `isEmpty()` method to the stack class that you use. *Do not use any other container data structure than the two stacks.*

   Annotate your methods with their runtimes.

   (b) (2 points) Can you explain why the implementation of the queue functionality as described in (a) is correct? Why do the two stacks together correctly implement the first-in first-out functionality of a queue?

   It might help to consider the following sequence of operations, and draw pictures on how the queue and the corresponding stacks change: `enqueue(1)`, `enqueue(2)`, `enqueue(3)`, `enqueue(4)`, `dequeue()`, `enqueue(5)`, `enqueue(6)`, `dequeue()`.