CIS 675 (Fall 2018) Disclosure Sheet

Name: Rui Peng

HW #6

_ Did you consult with anyone on parts of this assignment, including other students, TAs, or the instructor?

_ Did you consult an outside source (such as an Internet forum or a book other than the course textbook) on parts of this assignment?

If you answered **Yes** to one or more questions, please give the details here:

By submitting this sheet through my Blackboard account, I assert that the information on this sheet is true.

Homework 6:

Due November 26 at midnight Eastern time. Submit your solutions typed and in a pdf document. To receive full credit, explain your answers.

If you collaborate with another student or use outside sources, please list those students' names and the URL/title/etc. of the sources that you referred to. Collaboration is permitted, but you must write up your own solutions.

Problem 1: In class, we analyzed the binary counter, in which a binary array was used as a counter. It began at [0, 0, 0, ..., 0], which represented the string 00000...0 = 0. In each step, the binary counter was incremented: bits were flipped so that the value represented by the binary counter went up by one. For example, after the first increment, the binary counter would look like [0, 0, 0, ..., 1], and after the second increment, the binary counter would look like [0, 0, 0, ..., 1, 0]. For this problem, flipping a bit costs \$1.

We modify the in-class problem as follows: Suppose that in addition to paying \$1 to flip a bit, we *also* have to pay to initialize a bit, or create the element in the array the first time it is used. The cost of this initialization is \$$2^i$, where i is the index of the bit (assume the furthest right bit has index 0). After initialization, the new bit will be set to 0 (so you must pay an additional \$1 to change it to a 1, if you want it to be a 1). Assume that the array starts as the empty array [], which we interpret to represent the value 0.

Assume that there are no costs other than those stated above. Over a sequence of n increments, what is the amortized cost per increment? State your answer in terms of \$s, not big-O.

Hint: What is the average cost per increment if we ignore the new costs? What are the new costs? How much do they add up to in total?

Problem 2: Suppose that we have a stack (just a regular stack with Push and Pop, not the modified MultiPop stack) whose size is not allowed to exceed K (for instance, if $K = 10$, then we are not allowed to have more than 10 elements on the stack at any time). If you try to Push an element onto a stack that is already full, then nothing happens. Suppose that after every K operations, we automatically make a copy of the stack for back-up purposes (note that the stack may or may not be full at this point). When doing this backup, the stack itself is not modified.

Suppose that Push and Pop each cost \$1 (including if you try to Push an element onto an already-full stack). Suppose that copying the stack costs \$C, where C is the number of elements currently on the stack. Perform an amortized analysis of the running time of n operations. State your answer in terms of the average cost per operation, in dollars (that is, do not use big-O notation).

Problem 3: In class, we discussed a stack with three operations: Push, Pop, and MultiPop(k). The MultiPop(k) operation is implemented as a series of k pops (or less, if there are fewer than k elements on the stack). Push and Pop each take 1 unit of time, and MultiPop(k) takes up to k units of time. Using amortized analysis, we determined that the average cost of an operation, over n operations, was 2 units of time. Suppose we modify this stack and now, for backup purposes, every time the stack reaches k elements, we write a copy to the hard drive (here k is some fixed value specified by the user) and delete all elements from the stack (they can no longer be popped after they are backed up). Writing k elements to the hard drive takes k units of time, and deleting all k elements also takes k units of time. Consider a sequence of n operations, and perform an amortized running time analysis to find an upper bound on the average length of time that each operation can take. State your answer in terms of the average cost per operation, in dollars (that is, do not use big-O notation).

Problem 4: Consider the extensible array data structure that we learned in class. Consider the version in which there is no extra cost for allocating new memory (the first version we looked at). Suppose that we want to add another operation to this data structure: Remove, which deletes the last element added. In order to make sure that the array doesn't take up too much space, we say that if the array is less than half full, we will reallocate memory that is only half the size of the current array, and copy all the elements over. This is basically the opposite of the insertion operation from before.

Part a: Does amortized analysis make sense here? In other words, does it allow us to get a tighter bound than the standard analysis? Why or why not?

Part b: Instead of reallocating memory if the array becomes half empty, we reallocate/copy if the array becomes at least 3/4 empty (only 1/4 of the array cells are being used). Analyze the running time of a sequence of n operations using amortized analysis. Hint: this is a straightforward modification of the original analysis. If we shrink an array, how many elements get added before we next double it? If we are deleting elements, how many do we delete before shrinking it?

Extra Credit: In class, we performed (or will perform, depending on what day you're reading this assignment) amortized analysis on an algorithm in which operation k had running time k if k is a power of 2, and running time 1 otherwise. We performed the analysis using the aggregate method (that is, we added up the total cost of n operations). Redo the amortized analysis of this algorithm, except this time use the banking method. Assume that one unit of running time is \$1.