Assignment 5 CSC263
Due: Dec 4, 2008

1. [15 marks] Let $G = (V, E)$ be a directed acyclic graph in which each vertex $u \in V$ is labelled with a unique integer $L(u)$ from the set $\{1, 2, \ldots, |V|\}$. For each vertex $u \in V$, let $R(u)$ be the set of vertices that are reachable from u. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, i.e., $\min(u)$ is the vertex v such that $L(v) = \min\{L(w) : w \in R(u)\}$. Give an $O(|V| + |E|)$ time algorithm that computes $\min(u)$ for all vertices $u \in V$. Briefly justify your algorithm's correctness and why it runs in $O(|V| + |E|)$ time.

2. Consider an "extendable array" data structure for representing a dictionary ADT in which only the following two operations are supported:

   - ADD(x): adds the element x to the next empty space in the array.
   - RETRIEVE(i): returns the ith item in the array.

   Suppose that in our implementation, instead of copying the elements of the array into an array of double the size (i.e., from n to $2n$) whenever capacity is reached, we copy the elements into an array with $\lceil\sqrt{n}\rceil$ additional cells. That is, whenever the array is full, we go from capacity n to $n + \lceil\sqrt{n}\rceil$.

   (a) [15 marks] Use the aggregate method to give a tight bound on the amortized complexity of ADD operations in this implementation.

   (b) *Bonus* [10 marks] Now use the accounting method to give a tight bound on the amortized complexity of ADD operations.

3. In this question we will investigate implementing an open addressing hash table using a dynamic array. The hash table will be a dynamic array that doubles whenever it becomes 3/4 (or more) full and halves whenever it becomes 1/4 (or less) full. More precisely, when the array grows, we have to create a new array of twice the size, go through every slot in the old array and, whenever we find a nonempty slot, rehash the item into the new array. We handle the shrinking case similarly. Assume that the array starts out empty. The hashing will be done by some arbitrary hash function with some arbitrary type of probing. Throughout the question, we will measure the cost of each INSERT and DELETE by the number of array slots that we need to access (read or write).

   (a) [2 marks] Recall from lecture that the expected number of probes needed to INSERT or DELETE an item from a hash table with n elements and m slots is $\frac{1}{1-a}$ where $a = \frac{n}{m}$ is the load factor. In the scheme described above, what is a_{max}, the biggest that the load factor ever becomes? For simplicity, assume from now on that every INSERT or DELETE requires exactly $\frac{1}{1-a_{max}}$ probes.

   (b) [5 marks] Let m be the current size of the array. What is the cost of doing an INSERT in the case where the array needs to double? What is the cost of doing a delete in the case where the array needs to halve? Briefly explain your answers.

   (c) [8 marks] Use the accounting method to show that the amortized cost for an INSERT or a DELETE operation is $O(1)$. In particular, detail a "credit scheme" so that we can always cover all the costs of an insertion or a deletion. Make sure to specify how much to charge for INSERT and DELETE, what the credit invariant will be and, briefly, how to maintain the credit invariant.

   (d) [5 marks] Now consider an extensible hash-table that only halves the hash-table size whenever the load factor falls *strictly below* 3/8 (i.e., re-sizing is done if the load factor falls to any number less than 3/8 but is not done if the load factor is *exactly* 3/8.) In such an implementation, is the amortized cost of a INSERT or a DELETE operation still $O(1)$? Justify your answer: if yes, the accounting method to prove that the amortized cost is $O(1)$; if not, give a sequence of n INSERT and DELETE operations that take cannot be completed in time $O(n)$.