Exercises on Amortized Analysis*

April 19, 2015

1. An extendable array is a data structure that stores a sequence of items and supports the following operations.

- AddToFront(A, x) adds x to the beginning of the extendable array A.
- AddToEnd(A, x) adds x to the end of the extendable array A.
- Lookup(A, k) returns the k-th item in the extendable array A, or Null if the current length of the array is less than k.

Describe a simple data structure that implements an extendable array. Give pseudo-code descriptions of the three operations above and provide an amortized analysis of the operations. Your AddToFront and AddToBack algorithms should take $O(1)$ *amortized time*, and your Lookup algorithm should take $O(1)$ *worst-case time*. The data structure should use $O(n)$ space, where n is the number of elements stored in the data structure.

2. An ordered stack is a data structure that stores a sequence of integers and supports the following operations.

- OrderedPush(S, x) removes all integers smaller than x from the ordered stack S then adds x to the top of the stack.
- Pop(S) deletes and returns the top item in the ordered stack S (or Null if the stack is empty).

Suppose we implement an ordered stack with a simple linked list (the number of nodes in the list being equal to the number of elements stored in the stack). Give pseudo-code descriptions of the two operations above and provide an amortized analysis of the operations. Both operations should take $O(1)$ amortized time.

3. Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. Give pseudo-code descriptions of your enqueue and dequeue operations and provide an amortized analysis of the operations. You

*Adapted from the exercises in Jeff Erickson's handout on amortized analysis.

may assume that the stacks provide the standard Push and Pop functions both of which run in $O(1)$ worst-case time. The only access you have to the stacks is through these Push and Pop functions.

4. A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

- QuackPush(Q, x): add a new item x to the left end of the quack Q;
- QuackPop(Q): remove and return the item on the left end of quack Q;
- QuackPull(Q): remove the item on the right end of quack Q.

Describe how to implement a quack using three stacks and $O(1)$ additional memory, so that the amortized time for any QuackPush, QuackPop, or QuackPull operation is $O(1)$. There is a restriction that each element in the quack must be stored in exactly one of the three stacks. Give pseudo-code descriptions of these three operations above and provide an amortized analysis of the operations. You may assume that the stacks provide the standard Push and Pop functions both of which run in $O(1)$ worst-case time. The only access you have to the stacks is through these Push and Pop functions.