Solution Sketch for Tutorial-1

1. Suppose that in a k-bit binary counter supporting only increment operations, the counter is not initialized to 0 (so the first increment operation can start at any value). Will the O(1) bound of the amortized cost of an increment operation in a sequence of n increment operations still hold in such a counter?

   [Solution Sketch] Yes, as long as $n = \Omega(k)$. See analysis in text (Cormen).

2. Will the O(1) amortized cost of increments in a k-bit binary counter remain valid if you also allow decrement operations (i.e, in any sequence of intermixed increment and decrement operations)? If yes, show the amortized analysis. If not, justify why not.

   [Solution Sketch] No. Just consider a sequence of n operations of the form increment/decrement/increment/decrement/…..starting from the value $2^k - 1$ for a k-bit counter, changes all bits each time, so O(nk) total time. Note that starting from 0 does not change this as n can be arbitrary, so we can have the first $2^k - 1$ to reach upto the value $2^k - 1$, and then follow the sequence considered.

3. A sequence of stack operations is performed on stack whose size never exceeds k. After every k operations, a copy of the entire stack is made for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) using the accounting method.

   [Solution Sketch] There can be two variations. In variation 1, when the copy of the stack is made, the elements are no longer in the stack after the copy. In this case, the copy is like a pop only (with some additional work). In this case, you can include the cost of the pop (if it happens before copy) and copy in Push. So for accounting method, start with amortized cost of Push = 2, Pop = 0, Copy = 0. For a push, of the 2, one is for the push itself, one pays for popping it if done, or copying it to backup if it is not popped. Note that an element can either be popped or copied to backup but not both in this variation.

   In the other variation, when the copy of the stack is made, the elements are left in the stack after the copy. So in this case, simple solution is to start with amortized cost of Push = 2, Pop = 2, and Copy = 0. As the copy is made after every k operations and each copy can copy at most k elements (as stack size never exceeds k), each push/pop operation pays one extra cost for the copy if it needs to be done.
   (Just some additional thoughts: Note that it may seem that pop need not pay the extra cost as the element is removed, but it is needed as there may be earlier elements left in the stack which will be copied again. Consider the first k operations to be push, then a copy (so the pushes pay for the copy of k elements), and then a sequence of k/2 pop-push pairs. If the pop didn't pay the extra cost, you have only a credit of k/2 (after paying the 1 cost out of 2 for each push), which will be used up to pay for the k/2 pops, with nothing left for the copy. Even Push=3 , Pop=0, Copy = 0 do not help, as the k/2 extra credits after paying for the k/2 push and pops are not sufficient to copy the k elements. However, you can show that Push=4, Pop=0 or Push = 3, Pop =1 works.)

4. Consider the implementation of a queue using two stacks A and B (I am sure you all know this implementation). Find the amortized cost of a sequence of n enqueue and dequeue operation using each of aggregate, accounting, and potential method.

   [Solution Sketch] The implementation was discussed in Monday's class, but here it is again. Keep 2 stacks, IN and OUT. A push simply pushes in the IN stack. A pop can have two cases: (i) if OUT stack is empty, pop everything from the IN stack and push it in OUT stack one by one, then pop the top element from the OUT stack, (ii) if OUT stack is not empty, just pop the top element from the OUT stack.

   Aggregate method: Every element can be (i) pushed at most twice, once in the IN stack and once from the IN to OUT stack, and (ii) popped at most twice, either from the OUT stack or for transferring from IN to OUT stack. So total time = O(n), amortized cost O(1) per operation.

   Accounting method: Break the two operations into three operations: (i) Push, (ii) Easy dequeue (when OUT is not empty), (iii) Hard dequeue (when OUT is empty). Give amortized costs Push = 3, Easy dequeue = 1, Hard dequeue = 1. Basically easy dequeue pays for itself (just a pop, does not generate or use a credit), every push stores two credits for the hard dequeue for the move, one for popping it from IN and one for pushing it to OUT. The hard dequeue uses the credits stored with each element pushed to pop it from IN and push it to OUT, plus one final pop. So all O(1).

   Potential method: Choose potential = no. of elements in IN stack. Push increases potential by 1, easy dequeue leaves it unchanged, and hard dequeuer decreases it by –k, where k is the no. of elements in IN stack when the hard dequeue is done. So Amortized cost of push = O(1) + 1 = O(1), of easy dequeuer = O(1) + 0 = O(1), of hard dequeue = O(k) – k = O(1).

5. Consider a stack with a fixed size K with PUSH, POP operations (assume PUSH operations fail in O(1) time if stack is full and POP operations fail in O(1) time if stack is empty). Which of the following are NOT valid potential function for amortized analysis of a sequence of n PUSH and POP operations (choose all that apply)?

   a) The number of elements in the stack

   b) The number of free spaces in the stack

   c) The number of free spaces - the number of elements in the stack

   d) The number of free spaces + the number of elements in the stack

   [Solution Sketch] (b) and (c) cannot be potential functions. Just start with an empty stack, do one Push operation, and see what D_0 and D_1 comes out to be. For (b) and (c), you will see $D_1 – D_0 < 0$ which violates the required conditions for a potential function.