Dalhousie University
CSCI 6057/4117 — Advanced Data Structures
Winter 2022
Assignment 1 Solutions

Questions:

1. [10 marks] A *sorted stack* is a stack in which elements are sorted in increasing order from bottom to top. It supports the following operations:

   `pop(S)`, which removes and returns the top element from the sorted stack S.

   `push(S, x)`, which first pushes item x onto the top of the sorted stack S and then maintains the increasing order by repeatedly removing the item immediately below x until x becomes the largest item in the stack.

   Note that here the `push` operation is different from that of a standard stack. For example, suppose that the items in the sorted stack S, from bottom to top, are currently $-9, -7, 0, 1, 4, 11, 12, 20$. Then, after calling `push(S, 5)`, the content of the stack, from bottom to top, becomes $-9, -7, 0, 1, 4, 5$.

   We implement the stack as a linked list.

   Show that the amortized costs of both operations are $O(1)$.

   It is possible to use any of the three approaches for amortized analysis taught in class. Here I will only show how to use the potential approach; feel free to use any other approach.

   We measure the running time by the number of nodes inserted or removed in the linked list representing the sorted stack.

   Let S_i be the sorted stack after the i-th operation, starting from an empty stack. We define the potential function, $\Phi(S_i)$, to be the number of items in the stack. Then it is clear that $\Phi(S_0) = 0$ and $\Phi(S_i) \geq 0$ for any $i \geq 0$.

   Consider the i-th operation. There are two cases.

   In the first case, the i-th operation is `pop`. The actual cost, c_i, of this operation is 1, and its amortized cost is then $a_i = c_i + \Phi(S_i) - \Phi(S_{i-1}) = 1 - 1 = 0$.

   In the second case, the i-th operation is `push(S,x)`. Let k be the number of items removed from the stack after we push x onto the stack. The actual cost, c_i, is then

$k+1$, since this operation inserts one node and removes k nodes. We also have $\Phi(S_i) - \Phi(S_{i-1}) = 1 - k$, since this operation first increases the number of items in the stack by 1 (by inserting x), and then decreases the number of items in the stack by k. Therefore, the amortized cost is $a_i = c_i + \Phi(S_i) - \Phi(S_{i-1}) = k + 1 + 1 - k = 2$.

In both cases, the amortized cost is $O(1)$.

2. [10 marks] In the vector solution to the problem of maintaining resizable arrays shown in class, we allocate a new array with twice the size as the old one when inserting into a full array.

   In this question, we study the following alternative solution: Initially the array has a block of memory that can store at most c_0 entries, where c_0 is a positive constant. Each time we insert into a full array, we allocate a new memory block whose size (i.e., the maximum number of entries that it could store) is the size of the current memory block plus a fixed constant value c. We then copy all the elements from the current memory block to the newly allocated memory block, deallocate the old memory block and insert the new element into the new block. The new memory block will be used to store the content of the array afterwards, until we attempt to insert into a full array again.

   Prove that, under this scheme, performing a series of n insertions into an initially empty resizable array takes $\Omega(n^2)$ time, no matter what constant value we assign to c.

   Solution: An `INSERT` operation requires us to allocate a new array when the number of elements in the array before insertion is $c_0 + ic$, for $i = 0, 1, \ldots, m-1$, where $m = \lfloor (n - c_0)/c \rfloor$. The cost, measured by the number of element copies, of each of these `INSERT` operations is $c_0 + ic + 1$. Therefore, the total time of handling all these `INSERT` operations which caused us to allocate a new array is

$$\sum_{i=1}^{m-1}(c_0 + ic + 1) = (c_0+1)m + cm(m-1)/2 \geq c((n-c_0)/c - 1)((n-c_0)/c) = \Omega(n^2)$$

   Therefore, it requires $\Omega(n^2)$ time to perform all n insertions, including those that caused us to allocate a new array.

3. [15 marks] In class we saw the example of incrementing an initially 0 binary counter. Now, suppose that the counter is expressed as a base-3 number. That is, it has digits from $\{0, 1, 2\}$.

   (i) [5 marks] Write down the pseudocode for `INCREMENT` for this counter. The running time should be proportional to the number of digits that this algorithm changes.

   Solution: Let array $A[0..k-1]$ store the content of the ternary counter, in which $A[0]$ stores the lowest digit. The following is the pseudocode:

```
INCREMENT($A[0..k-1]$)
1: $i \leftarrow 0$
2: **while** $i < k$ **and** A[i] $= 2$ **do**
3:     $A[i] \leftarrow 0$
4:     $i \leftarrow i + 1$
5: **if** $i < k$ **then**
6:     $A[i] \leftarrow A[i] + 1$
```

(ii) [5 marks] Define the actual cost of `INCREMENT` to be the exact number of digits changed during the execution of this algorithm. Let $C(n)$ denote the total cost of calling `INCREMENT` n times over an initially 0 base-3 counter. Use amortized analysis to show that $C(n) \leq (3/2)n$ for any positive integer n. To simplify your work, assume that the counter will not overflow during this process.

Note: I will give the potential function as a hint after the problem statement, though you will learn more if you try to come up with your own potential function.

Solution: Let D_i be the counter after the i-th `INCREMENT`, and Let $|D_i|_d$ be the number of digit d in D_i for $d \in \{0, 1, 2\}$. Define $\Phi(D_i) = |D_i|_1/2 + |D_i|_2$. Clearly $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i.

Let b_i be the number of 2 digits that our algorithm resets to 0 during the i-th `INCREMENT`. Then the actual cost, c_i, of the i-th `INCREMENT` is $b_i + 1$.

To see the amortized cost, there are two cases.

Case 1: After setting b_i 2 digits to 0, the i-th `INCREMENT` sets a 0 digit to 1. In this case, the number of 1 digits is increased by 1 and the number of 2 digits is decreased by b_i. Thus the amortized cost is

$$\begin{aligned} a_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= b_i + 1 + 1/2 - b_i \\ &= 3/2 \end{aligned}$$

Case 2: After setting b_i 2 digits to 0, the i-th `INCREMENT` sets a 1 digit to 2. In this case, the number of 1 digits is decreased by 1 and the number of 2 digits is decreased by $b_i - 1$. Thus the amortized cost is

$$\begin{aligned} a_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= b_i + 1 - 1/2 + 1 - b_i \\ &= 3/2 \end{aligned}$$

In either case, the amortized cost is 3/2, and thus the total actual cost of n insertions is upper bounded by $(3/2)n$.

(iii) [5 marks] Prove that for any c such that $C(n) \leq cn$ for any positive integer n, the inequality $c \geq 3/2$ holds. Again, assume that the counter will not overflow during this process.

Solution: Assume to the contrary that there exists a number $c' < 3/2$ such that $C(n) \leq c'n$.

Let $\epsilon = 3/2 - c'$. Then $\epsilon > 0$. We need only consider the case in which $\epsilon < 1$ (Otherwise, we have $c' < 1/2$ and we can choose another c'' such that $c' < 1/2 < c'' < 3/2$ and $C(n) \leq c'n < c''n$ holds. We then set $\epsilon = 3/2 - c''$).

Let $\delta = \lceil -\log_3 \epsilon \rceil$. Then $\delta > -\log_3 \epsilon$ and hence $3^{-\delta} < \epsilon$.

We then choose $n = 3^\delta$ and perform `INCREMENT` n times. After this, ignoring the leading 0 digits, the content of the counter will be one 1 digit followed by δ 0 digits. During this process, the first of these digits will be changed exactly once, the second digit will be changed 3^1 times, the third digit will be changed 3^2 times, etc., and the last digit will be changed 3^δ times. Hence the total cost is exactly $C(n) = 1 + 3^1 + 3^2 + \cdots + 3^\delta = (3^{\delta+1} - 1)/2$.

On the other hand, since $3^{-\delta} < \epsilon$, we have $c'n = (3/2 - \epsilon)n < (3/2 - 3^{-\delta})n = (3/2-3^{-\delta})3^\delta = 3^{\delta+1}/2-1 < (3^{\delta+1}-1)/2 = C(n)$. This contradicts the assumption that $C(n) \leq c'n$.

Hint: Let D_i be the counter after the i-th `INCREMENT`, and Let $|D_i|_d$ be the number of digit d in D_i for $d \in \{0, 1, 2\}$. Define $\Phi(D_i) = |D_i|_1/2 + |D_i|_2$.

4. [15 marks] In this problem, we consider two stacks, X and Y. n and m denote the sizes of X and Y (here the size of a stack is the number of elements currently in the stack), respectively. We maintain these two stacks to support the following operations:

   `PushX(a)`: Push element a onto stack X;

   `PushY(a)`: Push element a onto stack Y;

   `MultiPopX(k)`: pop $\min(k, n)$ elements out of stack X;

   `MultiPopY(k)`: pop $\min(k, m)$ elements out of stack Y;

   `Move(k)`: pop $\min(k, n)$ elements out of stack X, and each time an element is popped, it is pushed onto stack Y.

We represent each stack using a doubly-linked list, so that `PushX`, `PushY`, and a single pop operation on either stack can be performed in $O(1)$ worst-case time.

(i) [6 marks] What is the worst-case running time of `MultiPopX(k)`, `MultiPopY(k)` and `Move(k)`?

Solution: $O(\min(k, n))$, $O(\min(k, m))$ and $O(\min(k, n))$. You can also write $O(n)$, $O(m)$, and $O(n)$.

(ii) [9 marks] Perform amortized analysis to show that each of these five operations uses $O(1)$ amortized time.

Hint: Define a potential function that makes use of both n and m. In one possible solution, under the potential function of this method, some operations may have negative amortized cost. This is however fine. Why? Think about what conclusion can be drawn when an operation has negative amortized cost with respect to a particular potential function that starts out at 0 and always remains nonnegative.

Hint: Define a potential function that makes use of both n and m. In one possible solution, under the potential function of this method, some operations may have negative amortized cost. This is however fine. Why? Think about what conclusion can be drawn when an operation has negative amortized cost with respect to a particular potential function that starts out at 0 and always remains nonnegative.

Solution: Let n_i and m_i denote the number of elements in X and Y after the ith operation. We then define $\Phi_i = 3n_i + m_i$. Clearly $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i.

The amortized costs are as follows:

`PushX(a)`:

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + 3n_i + m_i - (3n_{i-1} + m_{i-1}) \\ &= 1 + 3(n_{i-1} + 1) + m_{i-1} - 3n_{i-1} - m_{i-1} \\ &= 4 \end{aligned}$$

`PushY(a)`:

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + 3n_i + m_i - (3n_{i-1} + m_{i-1}) \\ &= 1 + (3n_{i-1} + m_{i-1} + 1) - (3n_{i-1} + m_{i-1}) \\ &= 2 \end{aligned}$$

`MultiPopX(k)`:

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= \min(k, n_{i-1}) + 3n_i + m_i - (3n_{i-1} + m_{i-1}) \\ &= \min(k, n_{i-1}) + 3(n_{i-1} - \min(k, n_{i-1})) + m_{i-1} - (3n_{i-1} + m_{i-1}) \\ &= -2\min(k, n_{i-1}) \end{aligned}$$

`MultiPopY(k)`:

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= \min(k, n_{i-1}) + 3n_i + m_i - (3n_{i-1} + m_{i-1}) \\ &= \min(k, n_{i-1}) + 3n_{i-1} + m_{i-1} - \min(k, n_{i-1}) - (3n_{i-1} + m_{i-1}) \\ &= 0 \end{aligned}$$

`Move(k)`:

$$\begin{aligned}
a_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 2\min(k, n_{i-1}) + 3n_i + m_i - (3n_{i-1} + m_{i-1}) \\
&= 2\min(k, n_{i-1}) + 3(n_{i-1} - \min(k, n_{i-1})) + m_{i-1} + \min(k, n_{i-1}) - (3n_{i-1} + m_{i-1}) \\
&= 0
\end{aligned}$$