Problem Set 3 Solutions

This problem set is due **at 11:59pm** on **Wednesday, March 1, 2017.**

EXERCISES (NOT TO BE TURNED IN)

Amortized analysis

- Do Exercise 17.1-3 on CLRS at page 456
- Do Exercise 17.2-3 on CLRS at page 459
- Do Exercise 17.3-7 on CLRS at page 463

Problem 3-1. Extreme FIFO Queues [50 points]

Design a data structure that maintains a FIFO (First In, First Out) queue of integers, supporting operations ENQUEUE, DEQUEUE, and FIND-MIN, each in $O(1)$ amortized time. (i.e. any sequence of m operations should take time $O(m)$.) You may assume that in any execution, all the items that get enqueued are distinct.

(a) [15 points] Describe your data structures and provide your ENQUEUE, DEQUEUE and FIND-MIN procedures. (You may describe the procedures either in words or as pseudo-code) Be sure to include clear invariants describing the key properties of your data structures.

Hint: Use an actual queue plus auxiliary data structure(s) for bookkeeping.

Solution:

Pre-Solution: The problem is centered around augmenting a FIFO queue to perform some operations faster. So, we can start by setting up a basic FIFO queue and seeing which operations can already be done in O(1) amortized time - we can see that Enqueue and Dequeue can. However, we'll have to search the entire queue for Find-Min - so we'll set up an auxiliary data structure to keep track of possible minimum values.

Initially, we can try to simply keep track of the absolute minimum value and update during every enqueue and dequeue operation. This looks promising - but if we dequeue the minimum value, then it takes more than O(1) time for us to find the new minimum. This is not only bad in the worst case - an adversary could cause us to dequeue the min in every single step, making the amortized cost of dequeing high. This leads to the idea of keeping track of the minimum in the tail-segments of our queue, as in our solution. As in many problems that deal with amortization, the inspiration for our solution is the following: inserting any element also accounts for some constant time manipulation of the element itself (in this case deletion from the minimum tracker data structure), which makes that element not cause any additional costs to other operations.

Solution: For our choice of data structures, we might use a FIFO queue $Main$ and an auxiliary linked list, Min, satisfying the following two invariants:

1 Item x appears in Min if and only if x is the minimum element of some tail-segment of $Main$.

2 Min is sorted in increasing order, front to back.

A tail-segment of $Main$ is any subset of consecutively inserted elements of $Main$ which includes the tail (the first element inserted and still present). Our procedures are specified as follows:

```
ENQUEUE($x$)
1  Add $x$ to the end of $Main$.
2  Starting at the end of the list, examine elements of $Min$ and remove those that are larger than $x$;
   stop examining elements if you encounter one that is smaller than $x$.
3  Add $x$ to the end of $Min$.
```

```
DEQUEUE()
1  Remove and return the first element $x$ of $Main$.
2  If $x$ is the first element in $Min$, remove it.
```

```
FIND-MIN()
1  Return the first element of $Min$.
```

(b) [20 points] Prove that your operations give the right answers by showing that their correctness follows from the invariants on your data structures. Be sure to sketch arguments for why the invariants hold.

Solution:

Pre-Solution: The only operations that give answers are Dequeue and Find-Min.

Correctness for Dequeue means that the FIFO property of the queue is maintained. Since our Dequeue operation just removes the first element from our queue, arguing correctness just involves arguing that the order of elements in the queue doesn't change, in any possible sequence of Enqueue, Dequeue and Find-Min operations.

Correctness for Find-Min means that it returns the smallest element in the queue, which is harder to show. There are several ways that this can be proven - however, the question specifically asks for invariants on the data structures.

Since Find-Min returns the first element in our linked list, we need to show that our linked list starts off with the correct minimum value and our enqueue and dequeue operations update it correctly. The corresponding invariants on our linked list becomes that our list will contain only proper minimum values and that the first minimum value is the appropriate one to return.

Solution: This solution is for the choices of data structure and procedures given above; your own may be different.

The only two operations that return answers are DEQUEUE and FIND-MIN. DEQUEUE returns the first element of $Main$, which is correct because $Main$ maintains the actual queue. FIND-MIN returns the first element of Min. This is the smallest element of Min because Min is sorted in increasing order (by Invariant 2 above). The smallest element of $Main$ is the minimum of the tail-segment consisting of all of $Main$, which is the smallest of all the tail-mins of $Main$. This is the smallest element in Min (by Invariant 1). Therefore, FIND-MIN returns the smallest element of $Main$, as needed.

Proofs for the invariants: The invariants are vacuously true in the initial state. We argue that ENQUEUE and DEQUEUE preserve them; FIND-MIN does not affect them.

It is easy to see that both operations preserve Invariant 2: Since a DEQUEUE operation can only remove an element from Min, the order of the remaining elements is preserved. For ENQUEUE(x), we remove elements from the end of Min until we find one that less than x, and then add x to the end of Min. Because Min was in sorted order prior to the ENQUEUE(x), when we stop removing elements, we know that all the remaining elements in Min are less than x. Since we do not change the order of any elements previously in Min, all the elements are still in sorted order.

So it remains to prove Invariant 1. There are two directions:

- The new Min list contains all the tail-mins.
  ENQUEUE(x): x is the minimum element of the singleton tail-segment of $Main$ and it is added to Min. Additionally, since every tail-segment now contains the value x, all elements with value greater than x can no longer be tail-mins. So, after their removal, Min still contains all the tail-mins.
  DEQUEUE of element x: The only element that could be removed from Min is x. It is OK to remove x, because it can no longer be a tail-min since it is no longer in $Main$. All other tail-mins are remain in Min.
- All elements of the new Min are tail-mins.
  ENQUEUE(x): x is the only value that is added to Min. It is the min of the singleton tail-segment. Every other element y remaining in the Min list was a tail-min before the ENQUEUE and is less than x. So y is still a tail-min after the ENQUEUE.

  DEQUEUE of element x: Then we claim that, if x is in Min before the operation, it is the first element of Min and therefore is removed from Min as well. Now, if x is in Min, it must be the minimum element of some tail of $Main$. This tail must include the entire queue, since x is the first element of $Main$. So x must be the smallest element in Min, which means it is the first element of Min. Every other element y in Min was a tail-min before the DEQUEUE, and is still a tail-min after the DEQUEUE.

(c) [15 points] Analyze the time complexity: the worst-case cost for each operation, and the amortized cost of any sequence of m operations.

Solution:

Pre-Solution We start by noting that Find-Min and Dequeue are always O(1) time - since they only remove a single element. Thus, the only possibly expensive operation is Enqueue. To explore how to calculate amortized costs or define potential functions, we start by coming up with a sequence of operations where Enqueue is expensive. When we come up with these sequences, we can see that Enqueue is expensive when many elements need to be removed from Min.

This gives the idea for the potential function, which depends on the size of Min, since it represents the state of the data structure.

On the other hand, if we were using the accounting method, we can use the fact that only the Enqueue function adds new elements to Min - by adding a constant amortized cost, we can cover the ”cost” of removing the added element later.

Solution

DEQUEUE and FIND-MIN are $O(1)$ operations, in the worst case.

Solution 1: Accounting Method

ENQUEUE is $O(m)$ in the worst case. To see that the cost can be this large, suppose that ENQUEUE operations are performed for the elements $2, 3, 4, \ldots, m-1, m$, in order. After these, Min contains $\{2, 3, 4, \ldots, m-1, m\}$. Then perform ENQUEUE$(1)$. This takes $\Omega(m)$ time because all the other entries from Min are removed one by one. However, the amortized cost of any sequence of m operations is $O(m)$. To see this, we use the accounting method.

First, define the actual costs of the operations as follows: The cost of any FIND-MIN operation is 1. The cost of any DEQUEUE operation is 2, for removal from $Main$ and possible removal from Min. The cost of an ENQUEUE operation is $2 + s$, where s is the number of elements removed from Min.

Next, consider a sequence of operations, $o_1, o_2, \ldots, o_m$. Let us assign each ENQUEUE an amortized cost of 3, each DEQUEUE an amortized cost of 2, and each FIND-MIN an amortized cost of 1. Hence the net amortized cost of m operations is at most $3m$. We claim that $3m \geq \sum_{i=1}^{m} \hat{c}_i \geq \sum_{i=1}^{m} c_i$ where c_i denotes the actual cost of operation o_i and $\hat{c}_i$ denotes the amortized cost of operation instance o_i. To see that this is the case, note that ENQUEUE(x) contributes an amortized cost of 3, which covers its own actual cost of 2 plus the possible cost of removing x from Min later. This yields the needed $O(m)$ amortized bound.

Alternative Solution: Potential Method

ENQUEUE is $O(m)$ in the worst case. To see that the cost can be this large, suppose that ENQUEUE operations are performed for the elements $2, 3, 4, \ldots, m-1, m$, in order. After these, Min contains $\{2, 3, 4, \ldots, m-1, m\}$. Then perform ENQUEUE$(1)$. This takes $\Omega(m)$ time because all the other entries from Min are removed one by one.

However, the amortized cost of any sequence of m operations is $O(m)$. To see this, we use a potential argument. First, define the actual costs of the operations as follows: The cost of any FIND-MIN operation is 1. The cost of any DEQUEUE operation is 2, for removal from $Main$ and possible removal from Min. The cost of an ENQUEUE operation is $2 + s$, where s is the number of elements removed from Min. Define the potential function $\Phi = |Min|$.

Now consider a sequence $o_1, o_2, \ldots, o_m$ of operations and let c_i denote the actual cost of operation o_i. Let Φ_i denote the value of the potential function after exactly

i operations; let Φ_0 denote the initial value of Φ, which here is 0. Define the amortized cost $\hat{c}_i$ of operation instance o_i to be $c_i + \Phi_i - \Phi_{i-1}$.

We claim that $\hat{c}_i \le 2$ for every i. If we show this, then we know that the actual cost of the entire sequence of operations satisfies:

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} \hat{c}_i + \Phi_0 - \Phi_m \le \sum_{i=1}^{m} \hat{c}_i \le 2m.$$

This yields the needed $O(m)$ amortized bound.

To show that $\hat{c}_i \le 2$ for every i, we consider the three types of operations. If o_i is a FIND-MIN operation, then

$$\hat{c}_i = 1 + \Phi_i - \Phi_{i-1} = 1 < 2.$$

If o_i is a DEQUEUE, then since the lengths of the lists cannot increase, we have:

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \le 2 + 0 \le 2.$$

If o_i is an ENQUEUE, then

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \le 2 + s - s = 2,$$

where s is the number of elements removed from Min. Thus, in every case, $\hat{c}_i \le 2$, as claimed.

Problem 3-2. [50 points] Ben Bitdiddle takes up a new job as a plumber in Gatestown. The town has a unique structure in that there is a single horizontal pipe that cuts through the center of the town that takes care of all the plumbing needs - however, the pipe often leaks.

Ben Bitdiddle has an office in the center of Gatestown (which you can think of as $x = 0$) and two employees. Whenever there is a leak in the pipe (at some point $x = k$), one of his employees needs to travel to $x = k$ to fix the leak. You can assume that the employees travel at the exact same rate and once they reach a point where there is a leak they can fix it instantaneously. Employees do not need to return to the office between fixing leaks and leaks occur one at a time (a new leak will not happen until the previous one is fixed).

Ben would like to design an algorithm that minimizes the total traveled distance between his two employees.

(a) [10 points] Let's assume that the two employees are at points $x = x_1$ and $x = x_2$. His employee suggests the following algorithm - if there is a leak at point $x = k$, send the closer employee to fix it. More precisely, if $|x_1 - k| < |x_2 - k|$, send the first employee - else, send the second.

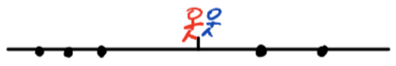
Construct a sequence of leaks S where this algorithm is inefficient - assuming x_1 = x_2 = 0, initially.

To be precise, define D_{OPT} as the optimal minimum distance that the employees have to travel to repair all the leaks in S, assuming that the entire sequence S is already known. Define D_{ALG} as the total traveled distance when this algorithm is used on the same sequence S. Construct a sequence S S where: $D_{ALG} > 100 * D_{OPT}$.

Advice: Spatial intuition will be very helpful when trying to understand this problem, especially the later parts. Thus, we suggest drawing in order to frame the problem more effectively.

Solution:

Pre-Solution: As from advice, we think that the easier way to understand the problem is to draw it out, in the initial situation:



Now, let's try to create a bad sequence for the employee's algorithm: intuitively, that algorithm will always cause the smallest possible movement per turn, but it can be bad in that it can make the employees move more often than necessary. Imagine that we have a lot of requests far on the positive side and a lot far on the negative side. Here, this algorithm works well because everyone stays in their domain. However, if we have requests only on the positive side, we would be better off with both employees there. This should give us an idea of what we can look for in our solution.

Solution: Consider the sequence $S = [2, 3, 2, 3...]$. The optimal solution would be to send an employee to $x = 2$ and an employee to $x = 3$, assuming they both start at 0. This has a total travel distance of 5. Our algorithm would send one employee back and forth between the two leaks, and if S has length over 1000, then D_{OPT} would be much greater than $100 * D_{OPT}$

(b) [15 points] Instead of listening to his employee, Ben chooses to approach you for advice. After hearing his problem, you suggest the following online algorithm: Let's assume that the employees are at points x_1, x_2, respectively and the leak occurs at $x = k$. Without loss of generality, let's also assume that $x_1 < x_2$

- If $k < x_1$, we send the employee at $x = x_1$ to repair the leak.
- If $k > x_2$, we send the employee at $x = x_2$ to repair the leak.
- If the leak occurs between x_1 and x_2, we send both employees to fix the leak. They each travel some distance d until one of the employees reaches the leak. At this point, the other employee stops traveling as well.

To prove to Ben that this algorithm is good, you need to analyze its performance relative to the optimal, offline algorithm. We define the following quantities for our analysis:

- $OPT_{i,1}$ and $OPT_{i,2}$: represent the locations of employees 1 and 2 after fixing leak i in the offline algorithm, where $OPT_{i,1} < OPT_{i,2}$.
- $ALG_{i,1}$ and $ALG_{i,2}$ represent the locations of employees 1 and 2 after fixing leak i in our suggested online algorithm, where $ALG_{i,1} < ALG_{i,2}$.
- We can then define a distance metric for the employees for the online and offline algorithms. That is, define $DIST_{i,j} = |OPT_{i,1} - ALG_{j,1}| + |OPT_{i,2} - ALG_{j,2}|$ Geometrically, $DIST_{i,j}$ represents the distance required to move the online employees after repairing leak j to where their corresponding offline workers are after repairing leak i.
  Note that if $i = j$, this represents the distances between the corresponding employees in the online and offline algorithm after they repair the same leak i. If $i \neq j$, $DIST_{i,j}$ is the sum of the distance between the employees with larger x-coordinates in the offline and online algorithms plus the distance between the employees with the smaller x-coordinates, after the offline algorithm has dealt with leak i and the online algorithm has dealt with leak j.
- Finally, we define f_i as the distance between the employees in the online algorithm after repairing leak i. That is, $f_i = ALG_{i,2} - ALG_{i,1}$

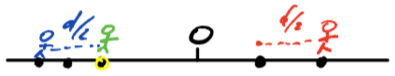
To compare online and offline performances, we define a potential function $Q_{i,j} = 2 * DIST_{i,j} + f_j$. We can use $Q_{i,i}$ to find a relationship between the optimal distance traveled and the distance that the online algorithm suggests.

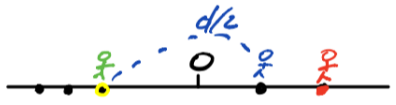
Assume the online algorithm requires the employees to travel $D_{ALG,k}$ to repair leak k after repairing leak $k - 1$. Prove that $Q_{k,k} - Q_{k,k-1} \leq -D_{ALG,k}$

Hint: Consider the two cases where only one employee has to travel and where both employees must travel to repair leak k in the online algorithm.

Solution:

Pre-solution: As hinted, we want to consider the situation in the two possible cases. The key insight here is to notice that the offline algorithm has just dealt with the leak, so there must be an offline employee (shown in green) on the leak:





Notice in the first picture that one of the employee moves closer to the offline employee by $d/2$. Regardless of the position of the other offline employee, this will allow us to show the desired result.

The second picture is even easier to analyze, as only one online employee moves, and necessarily moves towards the corresponding offline employee.

Solution:

Case 1: both employees travel a distance of $\frac{D_{ALG,k}}{2}$ until one employee reaches the leak.

Note that in this case, the offline algorithm has also sent a employee to resolve leak k. Since this implies that one of the employees in your online algorithm has gotten $\frac{D_{alg,k}}{2}$ closer to the employee in the online algorithm (the online employee who travels $\frac{D_{alg,k}}{2}$ and now exactly coincides with the offline employee at the location of the leak) - even if the other online employee has traveled $\frac{D_{alg,k}}{2}$ farther from the corresponding offline employee, the quantity $DIST_{i,i}$ will not be larger than $DIST_{i,i-1}$.

Since $Q_{k,k} - Q_{k,k-1} = 2(DIST_{k,k} - DIST_{k,k-1}) + f_k - f_{k-1}$ and we know that $DIST_{k,k} - DIST_{k,k-1} \leq 0$, we have the following bound.

$Q_{k,k} - Q_{k,k-1} \leq 0 + f_k - f_{k-1}$

In this case, the employees are moving closer to each other, so $f_k - f_{k-1} = -D_{ALG,k}$

Hence, $Q_{k,k} - Q_{k,k-1} \leq -D_{ALG,k}$.

Case 2: one employee travels a distance of $D_{ALG,k}$ to fix the leak

In this case, the distance between one of the online and the offline employees doesn't change (neither moves). However, the distance between the other online employee and offline employees decreases by $D_{ALG,k}$, since that's how far apart they originally were and now they are both at leak k.

However, the distance between the two employees in the online algorithm increases by $D_{ALG,k}$ since they are moving farther away from each other.

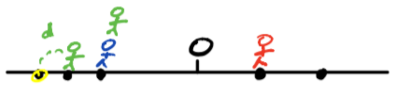
Thus, $Q_{k,k} - Q_{k,k-1} = 2(DIST_{k,k} - DIST_{k,k-1}) + f_k - f_{k-1} = 2 * (-D_{ALG,k}) + D_{ALG,k} = -D_{ALG,k}$

(c) [15 points] Suppose that the offline algorithm requires the employees to travel $D_{OPT,k}$ to repair leak k after repairing leak $k-1$. Prove that $Q_{k,k-1} - Q_{k-1,k-1} \leq 2 * D_{OPT,k}$

Hint: Consider cases where the two offline employees meet between leaks steps k and $k-1$ and when they don't

Solution:

Pre-solution: The two cases here are actually quite similar, as the only metric that changes is $DIST$. Remember the meaning of $DIST$: as intuitively we can see in the picture, moves of total size d will not increase $DIST$ by more than d:



In order to prove the result rigorousl, however, it is better to follow the hint and describe the two cases separately.

Solution:

$Q_{k,k-1} - Q_{k-1,k-1} = 2(DIST_{k,k-1} - DIST_{k-1,k-1}) + f_{k-1} - f_{k-1} = 2(DIST_{k,k-1} - DIST_{k-1,k-1})$

Thus, we just have to show that $DIST_{k,k-1} - DIST_{k-1,k-1} < D_{OPT,k}$

Case 1: Offline employee with larger x-coordinate still has larger x-coordinate

Let's assume that the offline employee with the larger x-coordinate travels a units. This implies that $|OPT_{i,1} - ALG_{i-1,1}| = |OPT_{i-1,1} - ALG_{i-1,1}| \leq a$, so $DIST_{k,k-1} - DIST_{k-1,k-1}$ increases by at most a due to this employee.

By similar logic, the other employee increases $DIST_{k,k-1} - DIST_{k-1,k-1}$ by at most $D_{OPT,k} - a$. Thus, $DIST_{k,k-1} - DIST_{k-1,k-1}$ increases by at most $D_{OPT,k}$.

Case 2: Offline employee with larger x-coordinate no longer has larger x-coordinate

This can be analyzed very similarly to Case 1. Let's assume the two employees travel a distance of u before they meet - by the logic of case 1, this increases $DIST_{k,k-1} - DIST_{k-1,k-1}$ by at most u. After they meet, we have reduced this to case 1 since the employee with the larger x-coordinate will end up with the larger x-coordinate. Thus, again, like in the first case, $DIST_{k,k-1} - DIST_{k-1,k-1}$ increases by at most $D_{OPT,k} - u$.

Hence, in total for either case, $DIST_{k,k-1} - DIST_{k-1,k-1} \leq D_{OPT_k}$

(d) [10 points] Define D_{ALG} as the total distance traveled by the employees in the online algorithm and D_{OPT} as the total distance traveled by the employees in the optimal algorithm. Using the previous parts, show that the online algorithm is 2-competitive - i.e. for any sequence S, show that $D_{ALG} \leq 2 * D_{OPT}$.

Solution: We note that $Q_{0,0} = 0$ and that $Q_{a,b}$ is always positive.

From combining parts (b) and (c), we know that

$Q_{k,k} - Q_{k-1,k-1} \leq 2 * D_{OPT,k} - D_{ALG,k}$

$Q_{n,n} = Q_{n,n} - Q_{0,0}$

$= \sum_{j=1}^{n} Q_{j,j} - Q_{j-1,j-1}$

$\leq \sum_{j=1}^{n} 2 * D_{OPT,j} - D_{ALG,j} = 2 * D_{OPT} - D_{ALG}$

Since we know $Q_{n,n} \geq 0$

$2 * D_{OPT} - D_{ALG} \geq 0$

$D_{ALG} \leq 2 * D_{OPT}$, which shows our algorithm is 2-competitive.