MaxQueue

Jonathan Wenger

1. A naive interpretation of MaxQueue would **not** satisfy the requirements of getting each method to be either O(1) or amortized O(1), because calling the max method would require iterating through the entire array/list (depending on the way it was implemented) to obtain that information, getting max() to have a runtime of O(N), NOT O(1) as we would like.
2. My implementation of the MaxQueue used a linked list. I had 4 instance variables: the head of the list, the tail of the list, an int to store the maximum, and an int to store the size of the queue. I enqueued items at the tail of the list (which was an O(1) operation, because I had the instance variable to the tail and therefore did not have to iterate through the entire linked list). I also would then increase the size instance variable by 1. I then checked to see if the item enqueued was greater than the maximum, and if it was I would change the max instance variable to that value (which is also O(1) time).

   To dequeue, I removed the item at the head of the list, and made the new head the previous head's next. I then checked to see if the max was removed, and if it was, I would iterate through the entire linked list to find the highest value there to make the new max (which can take O(N) time on each occurrence, but amortized is O(K) time, where K is the number of dequeues in a total number of

N enqueues and K dequeues (which is the same as O(N), as k is proportional to N).

3. Going through each of the Big-O constraints
    a. Max is O(1), because I have the max stored as an instance variable, so accessing it takes O(1) time.
    b. Size is O(1), because I have the size stored as an instance variable, so accessing it takes O(1) time.
    c. Enqueue is O(1), because all I am doing is adding it to the tail of the list that I already have a instance variable pointing to (which is O(1), see explanation in #2), updating the size (increasing the size instance variable by 1, which is O(1)), and possibly updating the max if the enqueued variable is higher than the max (which is O(1)). This means that enqueue is O(1).
    d. Dequeue I believe to be amortized O(K), where K is the number of dequeues in a total of N enqueues and dequeues. Here's why. Using only enqueues and dequeues (because enqueues directly affect the performance of dequeues, and size and max are both O(1) all of the time having no effect on the performance of dequeues), let's say we have N operations. If K of them are dequeues, then N-K of them are enqueues. In order to find the amortized runtime, we have to assume the worst case

scenario for every dequeue. For the worst case scenario to occur, every dequeue would iterate through all of the elements in the queue at that time, while the queue is at its highest possible capacity (which is when all of the enqueues are done before any of the dequeues). That means that after all of the enqueues are done, the queue is at a size of N-K elements, and there are about to be K dequeues taking place.

The runtime for those K dequeues is as follows: a dequeue first decreases the size of the queue by 1, then (in a worst case scenario) iterates through the entire queue to find the new max. That means that dequeue would be running K times, each time iterating through the queue at 1 size smaller than it was before. That means that the first time dequeue is run, it runs at a runtime of N-K-1 (the -1 as one element was removed before iterating through the queue), then N-K-2, N-K-3, etc. until N-K-K (as we started with N-K-1 and are going through an K times). N-K-K can be rewritten as N-2K.

A formula to determine the total runtime of all of the dequeues is the sum of integers formula (arithmetic series formula), which says if you're adding a list of consecutive integers from the lowest integer A to the highest integer L, the formula is $\frac{1}{2}(c(a + l))$, where C is the total number of integers being added (see here for details about the formula).

As we said before, our lowest integer being added (A) is N-2K, and our highest integer being added (L) is N-K-1, and the number of integers being added in this case (C) is K. So after applying this formula, the total sum (i.e. the runtime of all of the dequeues in the worst case scenario is $\frac{1}{2}(k((n - 2k) + (n - k - 1))$. This can be rewritten as $k(n - \frac{3}{2}k - \frac{1}{2})$. This formula right here is the runtime for all of the dequeues in a worst case scenario. The runtime of all of the enqueues is N-K, as each enqueue is O(1) and there are N-K enqueues, and N-K multiplied by 1 is still N-K. That means the total runtime of all of the operations is the runtime of the dequeues plus the runtime of the enqueues, which is $k(n - \frac{3}{2}k - \frac{1}{2}) + (n - k)$. As this is algorithm analysis, we only care to focus on the highest growing integer (which in this case is N, as N>K), so let's rewrite the formula to only focus on the N which is $kn + n$. To get the runtime of an individual operation, I'll divide that above formula by N, which will cancel out all of the N's, leaving only $k + 1$. While there are certainly no N's and the amortized runtime is not N exactly, the runtime of each operation is still proportional to K, which is certainly possible (and probable) to be closer to O(N). However, I believe that the average runtime of each operation will be closer to O(1) than to O(N), because for a dequeue to occur and actually have a runtime, there has to be more

enqueues than dequeues, and enqueue is O(1), and dequeue runs quicker the more times you run it.

e. I am only using O(N) space, as my linked list only has the elements that are currently in the queue, which are N elements, along with two instance variables (max and size), which is O(1), and N+2 is O(N).

My Thought Process

I know that my implementation does not fulfill the requirements of getting dequeue to be worst case O(1), rather than being worst case O(N). I drove myself crazy thinking about other solutions that would get the runtimes to be better, but I ultimately decided on submitting this implementation (even though there is only one data structure - the linkedlist, and it was specifically mentioned on Piazza to make sure that this class would be made up of multiple data structures). I will now explain the reason why I chose what I chose and did not go down an alternate route.

I chose the linked list only (without another data structure with the linked list) for a few reasons. Firstly, three out of the four required methods ran in O(1) time, while only one of the four runs in O(N). Additionally, for that method (dequeue) to actually run in O(n) time is very uncommon, as explained before it runs in O(k) (which I know is proportional to N), and I believe that in an average case it would be closer to O(1) time. Other options I considered was using a HashMap in addition to the linked list, which would give me an O(1) time of both inserting and removing ints, but it did not help in

any way finding the max faster, as I would still have to iterate through the entire key set of the HashMap to get the max.

I also very strongly considered (in addition to the linked list) using a TreeMap to hold the keys and values (using the ints being inserted as the keys and the number of times they appear in the queue as the values). That would mean that I could find the max in O(1) time by calling TreeMap.lastKey() (as the keys in a TreeMap are sorted). However, I ultimately did not decide on it, because that would mean inserting and removing (enqueue and dequeue) would be O(Log N), as a tree map under the hood is a balanced binary search tree. While that would make dequeue faster (going from O(N) to O(Log N)), that would make enqueue slower (going from O(1) to O(Log N)), and based on my own intuition and my emails with Dr. Leff, the linked list implementation seems better.

I also thought of using (implementing my own version of) a maxHeap, which would allow me to get the max in O(1) time by just returning the root of the tree, However, that would not be any more efficient than the tree map, as it is still O(Log N) time to insert and remove from a maxHeap.

I also did not see how the ArrayDeque class was supposed to help, as that would still require me to iterate through the entire deque if the max was removed to find the new max.

I was, and am still continuing to, drive myself crazy to think of a better solution, but after a lot of hours and hard work thinking, this is the best I could come up with.