Adam Frenkel

Why a basic implementation wouldn't work.

A basic implementation would mean that every time we would enqueue a new item into the ArrayDeque we would check if it was greater than the current max, and if it is the new max then we would set the max to the new value. This would take O(1) Time. The problem though comes on dequeue. Every time you dequeue you have to check if it's the max. If it is the max, then you have to go through the entire ArrayDeque to find the new max. This means that dequeue() has to go through the entire ArrayDeque every time the max is removed, which would take O(n) time.

My Implementation and why it takes amortized constant time.

My Implantation works by having one ArrayDeque that functions as the main queue. Here items are enqueued and dequeued as normal. In order to keep track of the max I create a list where I keep track of the max and all the future maxes that will come as the old maxes are dequeued (this will be elaborated on). The key insight is that if a number in enqueued that is greater than any given number in the array then all those numbers can be ignored while keeping track of the max. To illustrate:

Say this is the current ArrayDeque:

			1	2	4	8

That means the current max list looks like this:
1 – 2 – 4 – 8 – head
This means that the current max is 8, and next max after 8 is dequeued will be 4, then 2, then 1.

Now the number 7 is enqueued:
This is how the Queue will look:

		7	1	2	4	8

But **this is the important part**, look what happens to the max list:
7 – 8 – head
The numbers 1, 2 and 4 can all be forgotten about, because they will never be the max!

- What this means is that **max**() is trivial as all it takes is returning the max value which enqueue and dequeue have been keeping track of.
- **size()** is also trivial, as that will always be kept track of as one enqueues and dequeues, thus one only has to return the size value, which takes O(1).
- **dequeue()** is also very simply O(1), as one simply removes the first element from the Queue. If that element is equal to the current max, then one simply removes the first element from the max list and checks the value of the next element and sets that to be the new max.
- The amazing thing is that **enqueue()** will be amortized O(1). Inserting into the queue is trivial. But even adding the number to the max list isn't a problem. Say there are 15 numbers currently stored in the list. There is a 50 % chance at every number that the new number will be bigger than it. That means the odds that the new number will have to travel a mere ten spots down the list is 1/1024 (less than .1% chance). The odds of such a large list even existing in the first place is so low, because, as explained, once any larger number is added the rest of the list can be ignored. This means that adding to the list doesn't take a significantly longer time as n increases. Therefore, enqueue() will take amortized constant time as both adding to the queue and to the list will take O(1) time.
- The implementation is **O(n) space**, as the queue will at most contain n elements and the list will never grow to more than a couple dozen elements and even if does grow to n in a worst case, still 2n space is O(n).