Isaac Gutt

October 18, 2021

Intro to Algorithms

Max Queue Non-Programming

1. Using just the ArrayDequeue, it would be impossible to get the max in O(1) time, while also having enqueue() and dequeue() also be amortized O(1) time. You can keep a variable holding the max every time enqueue is called, but once dequeue comes in and removes the max, you'll have to go through the entire queue to find the newest max, an O(n) operation. Therefore, different data structures will be needed to satisfy these requirements.

2. My implementation was to use the opposite of a queue, a stack. Stacks are last in first out, instead of first in first out. I have two main stacks, one for enqueue, and one for dequeue, and meanwhile have one more for each, to hold the max. The first time enqueue is called, the number is pushed into the enqueue stack and also the enqueue max stack, since it is the current max. The next time enqueue is called, the number is pushed onto the enqueue stack, and then enqueue() checks to see if the new number is bigger than the top number in the enqueue max stack. If it is, then the new number is pushed, if it's not, then the previous number is pushed again. In this way, the top number in the enqueue max stack is always the max.

   The first time dequeue is called, the entire enqueue stack is poured into the dequeue stack. Last in first out, now becomes first in first out, the way a queue should be. While the dequeue stack is being filled, dequeue() also fills the dequeue max stack, once again adding the first value of the dequeue stack (since it's the current max in the dequeue stack), and always adding either the previous max, or the new max if the dequeue stack just received it's biggest value. Once again, the max is at the top of the stack, only to be different when the max is removed from the actual dequeue stack. Then, the top item from both the dequeue stack and dequeue max stacks are removed (the actual dequeue part of all this). Although the first dequeue operation is

an O(n) operation, it's worth it because the next n times dequeue is called, it will simply be popped off the dequeue stack (dequeue max stack will pop it's top value as well) in a constant time operation, so it's amortized O(1). After the first dequeue, the enqueue stack and enqueue max stacks continue to be filled with enqueue calls, only to fill the dequeue stacks when the dequeue stacks are empty. Enqueue and dequeue increment and decrement an integer instance variable called size, respectively. Size returns the size instance variable. Max returns the greater of the two values at the top of the two max stacks.

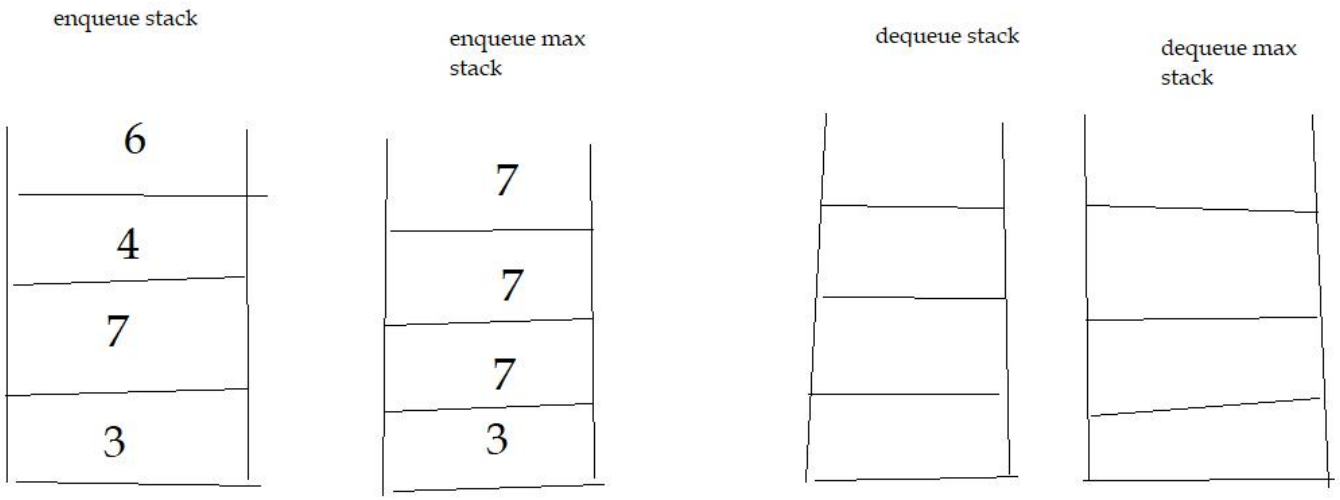
(visual example shown below)

3. dequeue() does execute an O(n) operation with a single while loop that only runs whenever the dequeue stack is empty, but that execution is worth it, because the next n times dequeue is called, all it will do is pop the dequeue stacks, a constant time operation. Therefore, it is an amortized O(1) operation.

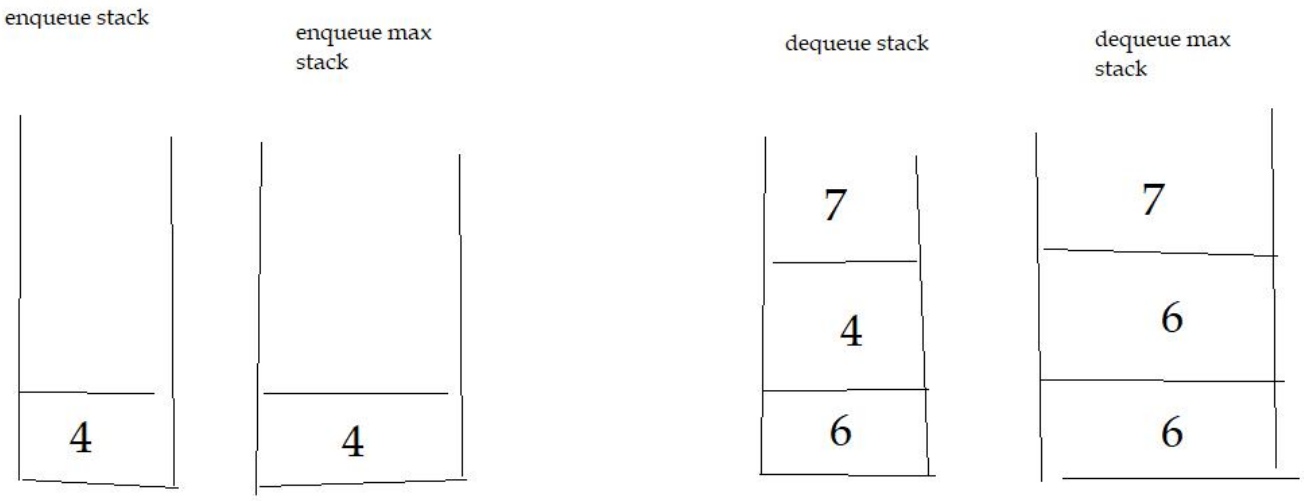
   max() peeks the two max stacks and returns the greater of the two values, an O(1) operation.

   size() returns the integer variable called size, an O(1) operation.

   enqueue() has no loops in it, it simply pushes the new number into the enqueue stack, the current max (of the enqueue stack) in the enqueue stack, and increments size, so it's an O(1) operation.



enqueue(3)
enqueue(7)
enqueue(4)
enqueue(6)



dequeue()
enqueue(4)