MaxQueue

Max Friedman

1. Explain (no more than a single paragraph) why a "naive" implementation (e.g., using ArrayDeque) will not satisfy requirements.

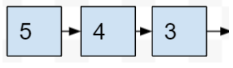
The main reason the naive implementation of ArrayDequeue will not satisfy requirements is due to the max value constantly changing. After we dequeue the max value, my program needs to decide which is the next max value. The naive way to solve this is, when the max value is dequeued, to iterate over every value left in the queue finding the next highest value. This is inefficient and takes O(n) time. Another reason the ArrayDequeue is inefficient is because dequeuing an element takes O(n) time. This is because after you remove the first element, each element replaces each previous index, as each element essentially has to fall back into each empty index. Other data structures (i.e. linked list) can remove the first element in O(1) time.

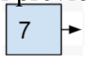
2. Explain the design of your adt implementation (no more than two paragraphs). Note that a good diagram(s) can be very helpful in such explanations: don't hesitate to use such diagrams here.

My adt implementation has two main parts, an array that serves as the queue, and a linked list that tracks max values. The array enqueues values to the "tail" pointer, which shifts over one as an element is added. When an element is dequeued, the element at that index becomes null, and the "head" pointer shifts over one. When the array becomes full, it doubles.

My linked list, designed to solve the problem of needing to iterate over the entire queue to find the next highest value, essentially becomes a sorted, from largest to smallest, list. After it becomes a descending list, when I dequeue the max value, the next value is obviously the next value in the list (i.e. the new head of the list). I achieve this by, while enqueueing, I simply check if that value (x) is greater than the last element of the list. If it is, it removes that value, removing each last element that x is greater than. If x is smaller than the last element of the list however, simply add x to the end of the list. The only situation the code will iterate is when x is greater than the last element in the list.

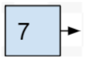
Example 1:
enqueue(5);
enqueue(4);
enqueue(3);

MaxList: 

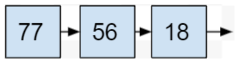
enqueue(7);
(7 erases all previous values leaving 7 as the max)
MaxList: 

Example 2:
enqueue(3);
enqueue(4);
enqueue(5);
enqueue(7);

(each element replaces each other each time we enqueue, leaving 7 as the max)

MaxList: 

Example 3:
enqueue(77);
enqueue(56);
enqueue(16);
enqueue(18);
(18 replaces 16, leaving a sorted, descending max list)

MaxList: 

3. For each of the above "Big-O" constraints, explain how your design satisfies the constraint

- Enqueue - (at worst) amortized O(1)
    - As an element is enqueued, it gets added to the array that houses the queue in constant time. However, when the array becomes full, it doubles, copying the values into a separate array. This is O(1) amortised, because as the array gets bigger and bigger, the chance that the array will double is farther and farther apart, making iteration rare. This leaves us with an average of close to constant time operations.
    - As an element is enqueued, if it's smaller/equal than the last value, it gets added to the end of the max list, which is an O(1) operation.
      If it is greater than the last value, it replaces all previous values it is greater than. This is O(1) amortised, because the only scenario you will iterate over all values is when every element is descending. Almost every other time, it does not replace even close to n elements.

- Dequeue - (at worst) amortized O(1)
    - Dequeue simply makes an element null, then makes the head of the queue shift one over. No iteration occurs.
    - Additionally, if the value removed is in the max list, remove the head from the linked list, leaving the next value as the next head. No iteration occurs.
- max() - O(1)
    - Calling max simply returns a variable, so no iteration occurs
- Size() - O(1)
    - Calling size simply returns a variable, so no iteration occurs