בס״ד

MaxQueue

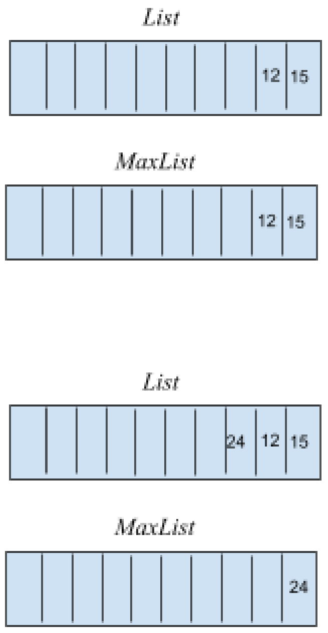
Samuel Sicklick, Oct. 12 2021

Failures of Naive Implementation

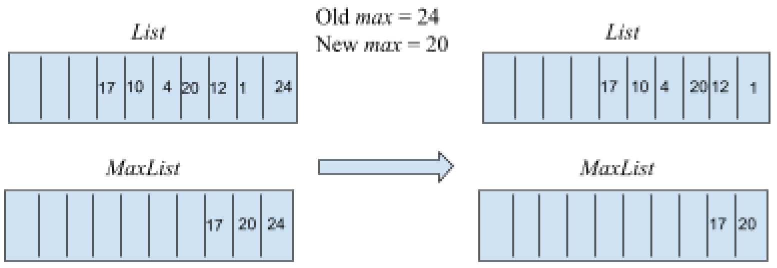
Using a naive implementation, such as a single ArrayDeque or LinkedList, would not have sufficed to meet the assignments O(1) and amortized O(1) requirements. Even if the *max* was set based on comparing values immediately when they were added via *enqueue*, the entire data structure would have to be traversed in order to determine what the new *max* value is after *dequeue* is called on a (prior) *max*.

ADT Implementation Explanation

In order to avoid the aforementioned issue, my implementation uses a second LinkedList (*MaxList*) to keep track of the current and next possible *max* values. This is aside from the primary LinkedList (*List*) which stores all the elements in the MaxQueue. When the first element is added by *enqueue* to *List*, it is also placed in *MaxList* and set as the current *max*. Each subsequent element added, after being placed in *List*, is then evaluated to determine if it should be added to the *MaxList* or not. If the subsequent element is greater than the current *max* of the MaxQueue ,which is always the front element of *MaxList*, then *MaxList* is replaced by a new LinkedList to which the element is added and its value is set to be the new *max*. However, if the new element is less than all values on the *MaxList* then it is simply added to the *MaxList*. Yet, if the element falls between the *max* and any element on the *MaxList*, then all elements which it is greater than are popped off of the *MaxList* and it is added in their place. Thus the *MaxList* will always store the *max* in the front node, followed by every element in the MaxQueue which would replace it should the *max* be removed. Accordingly, each element in the *MaxList* represents an element on the *List* which is greater than all of the values between it and the next element of *MaxList*.



When *dequeue* is called on anything other than the *max*, the element is simply popped off of the *List*. However, if the *max* is removed by *dequeue*, then the front elements of both the *List* and *MaxList* are removed. The next element of the *MaxList* is set to be the new *max*. This process allows for new *max* values to be set without having to traverse all of the data, as the *MaxList* will build up storing all the potential follow up elements.



Big - O Constraint Adjustments

Enqueue

Given *n* elements, the *MaxList* will contain *m* elements of all current of potential *max* values. If a new element is added which is greater than the *max* then the entire *MaxList* is cleared by being set to a new empty LinkedList where it is added as the first element, thus it is O(1). If a new element is added which is less than the back element of the *MaxList* then it is simply added behind it, also in O(1) time. However, if an element is added that is greater than the last value on the *MaxList* yet less than some other item on the *Maxlist*, then all the items which it is greater than are popped off of the list and it is subsequently added. However, this can only happen a limited number of times, as the more items popped off of the *MaxList* from a single call to *enqueue* reflects a higher value of the new element being added. Accordingly, if the new element is greater than *c* elements on the *MaxList* then on the next call which would require elements to be popped off the highest possible number of items which would have to be popped is the difference between the current *max* and the new added value. Thus, after a large number of items are popped off, the next call to enque will be O(1) and the number of possible inputs which can render it O(c) is diminished, rendering it amortized O(1).

Dequeue

If the element *dequeue* is being called on is not in the *MaxList* then it is simply O(1) as it is just removed from the *List*. Moreover, if the element being removed is the *max* then it will be removed from both the *List* and the *MaxList*, and the entry behind it is set to be the new *max*, also in O(1) time.

Size

Calling *size* simply returns the number of elements stored in the *List*, which is incremented and decremented upon each call to *enqueue* and *dequeue*, respectively. Thus returning this value does not require any traversal of data and is performed at O(1).

Max

Calling *max* simply returns the *max* value stored in the MaxQueue, which is updated upon necessity when a higher value is added via *enqueue* or when a previous *max* was removed via *dequeue*. Thus returning this value does not require any traversal of data and is performed at O(1).