CSSE230: Stacks and Queues

# Name(s): Joshua Eckels, Natalie Koenig

# Analysis

**Table 1:** Big-Theta runtimes of enqueue and dequeue for 4 implementations of the Queue ADT:

|  |  |  |
| --- | --- | --- |
| Implementation | Enqueue runtime | Dequeue |
| LinkedList | O(1) | O(1) |
| ArrayList | Best: O(1). Worst: O(n). | O(n) |
| Two stacks | Best: O(1). Worst: O(n). | O(n) |
| Growable circular array | O(n) | O(n) |

# Part 2: Discussion

Justify each of the runtimes in Table 1, as described in the specification:

**LinkedList**

enqueue: Insert a new node right after head. Does not depend on number of items in list. New item will always be put after head. No need to re-index anything.

dequeue: Accessing the last node is O(1) (assuming the LinkedList is doublylinked and tracks the tail node). Then point second-to-last node to the tail, effectively “cutting off” the last item from the list (dequeueing it). Everything is constant runtime; does not depend on number of items.

**ArrayList**

enqueue: Best-case (amortized), the new item can be added at the end in constant time. In the worst-case, the array will have to double in size, which is O(n) runtime.

dequeue: Removing an item from the beginning will require everything in the array to shift left by 1. This re-indexing process is O(n).

**Two** **stacks**

enqueue: This will be O(1) amortized. To enqueue an item, we simply push it onto the stack. However, for similar reasons as the ArrayList, if the stack has to grow in size, worst-case runtime will be O(n).

dequeue: This will be O(n) always. We will always have to iterate through every item in the first stack O(n), push them onto the second stack, and then iterate back through the second stack O(n) after removing the top item.

**Growable circular array**

enqueue: The way we implemented enqueue, the function will always be O(n). This is because we are constantly checking if the array is empty. Checking if the array is empty requires iterating through every item in the array. In the chance that the array is full, we will also have to resize the array, which will also be O(n).

dequeue: Dequeue will be O(n) all the time, for the same reason as enqueue. Since we have to check through all/most items in the array (worst-case and amortized) in order to determine if the array is empty, our function will be O(n) no matter what.