

In-class Exercises 10

*University of Wisconsin - La Crosse**Date: April 29*

1. As we saw in class, a queue is represented by a `List` underneath the hood. Of singly linked list, doubly linked list, and array list, which list structure is best from a runtime perspective for representing a queue and why?

Solution: Doubly linked list works well since we need to deal with both ends of the list, and doubly linked list supports $O(1)$ add/remove for both ends. Note, however, that we do not need to remove from the end of a queue. Thus, a singly linked list with a tail pointer (and no sentinel node) provides the constant-time access to the end of the list that we need in order to add. (Think about why that access to the tail is not sufficient to allow removing a node.)

2. In lab, we used an array (*not* an array list) for implementing a stack. Using an array to represent a queue is somewhat more difficult. Why? How might we achieve this in the most efficient way possible? What attribute(s) would you need in the class? What would the methods look like for enqueue, dequeue, and front? Hint: Do not move elements already in the queue.

Solution: We would need two attributes: one to keep track of the front of the queue (i.e., the value we would dequeue or report via front), and a second to keep track of the next available index at which to add. As values are “removed” from the queue, they would be returned, but the elements wouldn’t be moved in the array, allowing $O(1)$ performance. The array might need to be monitored for potential expansion (if that behavior was specified) and copying over of elements to a new, larger array, like with an array list. Note that this approach could not be used with an array list, since array lists require that the elements start at index 0.

3. What is displayed to the console after running the code below? What does the stack looks like over time?

```
1 LinkedList<Integer> stack = new LinkedList<>();
2 stack.push(3);
3 for (int i = 1; i <= 5; ++i) {
4     if (peek() % 2 == 0) {
5         stack.push(i);
6     } else {
7         int r = stack.pop();
8         stack.push(i + r);
9     }
10 }
11 while (!stack.isEmpty()) {
12     System.out.print(stack.pop() + " ");
13 }
```

Solution:

12, 2, 4,

4. What is displayed to the console after running the code below? What does the queue looks like over time?

```
1 LinkedList<Integer> queue = new LinkedList<>();
2 queue.add(3);
3 for (int i = 1; i <= 5; ++i) {
4     if ((i + queue.peek()) % 2 == 0) {
5         queue.add(i);
6     } else {
7         int r = queue.poll();
8         queue.add(i + r);
9     }
10 }
11 while (!queue.isEmpty()) {
12     System.out.print(queue.poll() + " ");
13 }
```

Solution:

5, 3, 5, 5,

5. Explain how you could use a stack and a queue to determine if the characters in a **String** form a palindrome. Note: This solution is not the most efficient way to solve this problem. The problem is an exercise in understanding stacks and queues.

Solution: Put the first half of the characters in a stack and the second half of the characters in a queue (taking care with whether the number of characters is even or odd). Then repeatedly **pop** a character from the stack and **dequeue** a character from the queue and compare them. If none differ, then the **String** is a palindrom.