Assignment 4

Refer to Canvas for assignment due dates for your section.

Objectives:
- Implement and test a mutable ADT.
- Implement and test an immutable ADT.

General Requirements

Create a new Gradle project for this assignment in your course GitHub repo. Make sure to follow the instructions provided in “Using Gradle with IntelliJ” on Canvas.

Create a separate package for each problem in the assignment. Create all your files in the appropriate package.

To submit your work, push it to GitHub and create a release. Refer to the instructions on Canvas.

Your repository should contain:
- One .java file per Java class.
- One .java file per Java test class.
- One pdf or image file for each UML Class Diagram that you create. UML diagrams can be generated using IntelliJ or hand-drawn.
- All non-test classes and non-test methods must have valid Javadoc.

Your repository should **not** contain:
- Any .class files.
- Any .html files.
- Any IntelliJ specific files.

Problem 1

Implement an ADT called `CourseCatalog`—an ordered, **mutable** collection, which will be used as part of a university course registration system. A `Course` class has already been written to store information about each course (download it from the Canvas page for this assignment). The ADT will need to support the following functionality.
- `void append(Course)`: Adds a `Course` to the end of the `CourseCatalog`.
- `void add(Course)`: Adds a `Course` to the beginning of the **`CourseCatalog`**.
- `void remove(Course)`: Removes a specified `Course` from the `CourseCatalog`. Throw a `CourseNotFoundException` if the `Course` doesn’t exist. If the `CourseCatalog` contains multiple instances of the same `Course`, the instance with the lowest index is removed. The `CourseCatalog` should not have any empty slots/nodes

(from the public perspective) after a `Course` is removed. For example, if the `CourseCatalog` contains 5 items and the `Course` at index 0 is removed, then the indices of the remaining `Courses` should be shifted down by 1—the `Course` that was at index 1 should be moved to index 0 and the index of the last `Course` in the catalog should be 3.

- `boolean contains(Course)`: Checks if the specified course exists in the `CourseCatalog`.
- `int indexOf(Course)`: Returns the index of the specified `Course` in the `CourseCatalog`, if it exists. If the `Course` doesn't exist, returns -1.
- `int count()`: Gets the number of `Courses` in the `CourseCatalog`.
- `Course get(int)`: Returns the `Course` at the given index in the `CourseCatalog`. Throws an `InvalidIndexException` if the index doesn't exist.
- `boolean isEmpty()`: Checks if the `CourseCatalog` is empty.

Specify this ADT in an interface and implement it as well as any other classes needed to satisfy the specification. You should also implement `toString`, `equals`, and `hashCode` for this ADT. Your implementation of `equals(Object o)` should return true if and only if the two `CourseCatalogs` contain the same `Courses` in the same order. Ensure that your implementations of `hashCode()` and `equals()` satisfy the contracts for both methods.

<u>You may not use any built-in Java collections, other than arrays, as the underlying data structure</u>. Do not modify the provided `Course` class.

Problem 2

Provide the design and implementation of a `Queue`, a data collection that operates in a **FIFO (first in, first out) manner**. Here is the specification:

- `Queue emptyQueue()`: Creates and returns an empty Queue.
- `Boolean isEmpty()`: Checks if the Queue is empty. Returns true if the Queue contains no items, false otherwise.
- `Queue add(Integer n)`: Adds the given Integer to the end of the Queue (note: queue allows duplicates).
- `Boolean contains(Integer n)`: Returns true if the given Integer is in the Queue, false otherwise.
- `Queue remove()`: Returns a copy of the Queue with the first element removed.
- `Queue removeElement()`: Returns a copy of the Queue with the given Integer removed. If the given Integer is not in the Queue, returns the Queue as is.
- `Integer size()`: Gets the number of items in the Queue.

Your implementation of `equals(Object o)` should return true if and only if the two queues have the same elements in the same position, i.e., for every element in `this`, the same element exists in the same position in `o`, and vice versa. Ensure that your implementations of `hashCode()` and `equals()` satisfy the contracts for both methods.

You may not use any built-in Java collections, other than arrays, as the underlying data structure. As the specification suggests, your implementation should be immutable.

Make sure your implementations for both problems are thoroughly tested. Please also provide UML diagrams.