Homework # 2
due Monday, September 16, 10:00 PM

In Homework # 1, you implemented two ADTs: `Pixel` and `Raster`. For this assignment, you will build on them to implement a RasterSeq ADT, a sequence of rasters and use this in a program to display a sequence of images: a video.

Note: for convenience, we have added getter methods for `x` and `y` to `Raster`. The methods are `x()` and `y()`. If we hadn't done this, you would have needed to use the length of the arrays if you needed to deduce their values.

1 Concerning the RasterSeq ADT

The RasterSeq ADT is a variant of the `Sequence` ADT from the textbook (Section 3.3 on pages 145–158 (142–155, 3rd ed.)). Please note that the ADT in this assignment differs substantially from the one in the textbook. You should refer to the textbook for the definition of the ADT, and then modify it a number of ways. The data structure is used in the same way (with one addition: a boolean indicating whether there is a current element), but not having a current element is treated separately from being at the end of the sequence.

The sequence stores a number of `Raster` objects in an order, and it also maintains a current location within the sequence. The location may be either at a current element (one of the `Raster`s in the sequence) or a non-element location either in between two elements, before the first element, or after the last element. The sequence can only be added to either before or after its current location. Only the current element can be retrieved or removed. The current location is changed either by the `start` operation, which moves to the first element, or the `advance` operation, which moves to the next element.

There are three ways for the sequence to reach a state with no current element. First, a new sequence is empty and has no current element until either `start` or `advance` is called. Second, if the current element is removed, we don't have a current element any more (unlike the textbook), but the sequence remembers where it was, so if a new element is added (using either `addBefore` or `addAfter`), the new element is placed where the removed element was. Third, if the sequence is `advance`d past the final element, there will be no current element. In that case, any addition (again with either `addBefore` or `addAfter`) is then placed at the end of the sequence.

Another important difference is that adding elements *does not* change the current element. If there was a current element before adding, that element is still the current element after adding, although one or more new elements has been added before or after it. If there was not a current element, there is still no current element; the current location is the same, and the new element(s) has been inserted either before or after that location.

The following methods thus have different semantics from the textbook: `addBefore`, `addAfter`, `removeCurrent`, `advance`, `addAll`. We also add a new public method:

`atEnd` Return whether the current location is at the end of the sequence. If at the end, there is never any current element.

We have provided a `toString` method which may be useful when debugging the ADT. Be aware that the returned string will include information about array contents that are not part of the sequence, i.e., from indices outside the range that holds valid data.

Unlike the lab exercise, `null` values *are* permitted. Nulls should count toward the size, and a null raster can be the current element.

Unlike the textbook, we recommend that you do *not* use `System.arrayCopy`. While it does indeed run faster than doing a loop, it's not very clear what it is doing. Use a "for" loop for clarity.

2 Concerning Invariants and Assertions

The sequence-specific data structure makes use of two integer fields, a boolean field, and an array field. There are certain configurations that make no sense. In situations like this, programmers are recommended to define and check object state invariants. For CompSci 351, this recommendation is converted to a requirement. See page 126 (3rd ed. p. 123) in the textbook, but remember that we also have added the new `hasCurrent` field.

There are conventions and Java language features to help you codify and test the invariant. For this homework, you will implement the class invariant as a private boolean method named `wellFormed()`. Then the beginning of every public method must have code as follows:

```
assert wellFormed() : "Invariant failed at start";
```

and at the end of any public method that changes any field, there must be the following line:

```
assert wellFormed() : "Invariant failed at end";
```

We have placed these lines in the code in the skeleton file for your convenience. In future assignments, you will have to place them yourself.

Public methods assert the invariant in this way, so they should *not* be called by other methods that are in the process of breaking and (one assumes) eventually restoring the invariant. Neither should they be called by code that is *checking* the invariant!

An invariant may be expensive to check. Therefore in Java, assertions are turned off by default. For that reason, an assertion should not include some side-effect that needs to happen. Don't rely on an assertion to check (say) that a parameter is good.

You should always turn assertions on while running the main test suite. With the command line, this is done by passing the flag `-ea` to the `java` executor. In Eclipse, assertions are enabled by adding `-ea` (don't forget the hyphen!) as a "VM argument" on the "Arguments" tab of the run configuration.

3 Concerning `ensureCapacity`

You must implement `ensureCapacity` in the same way that you did in lab 2. In particular, compared to the old capacity, if the needed capacity is:

- less then or equal, do not change the capacity
- greater, but not more than double, double the capacity
- more than double, set the new capacity to the needed capacity

4 VideoDemo.java

We provide most of a program to play a short video. It uses a `RasterSeq` to store rasters and play them back as frames of a video.

Your first task is to write `buildVideo`, which will add frames to the sequence to construct a video. After you do this, you will be able to see a single pixel character moving from left to right.

Your second task is to write `sloMo`, which will slow the video down by duplicating frames. After you do this, you will see the video play a second time, but more slowly.

5 Files

In the git repository for this assignment, we provide the solution to Homework # 1 (in a JAR file) and the following files:

- `src/VideoDemo.java` Main program for video rendering. Modify this one!
- `src/TestRasterSeq.java` JUnit test cases. Do not modify.
- `src/TestEfficiency.java` Test the efficiency of your code. It should take less than a second to run. Turn off assertions to run this. Do not modify.
- `src/TestInternals.java` Test private methods that are not part of the ADT, and are implementation-specific, namely `RasterSeq.wellFormed` and `RasterSeq.ensureCapacity`. Do not modify.
- `src/edu/uwm/cs351/RasterSeq.java` A skeleton implementation of the ADT. Modify this one!
- `src/UnlockTest.java` Run this to unlock the locked tests. Do not modify.