Homework 2: Stacks and Queues

Stacks and queues are commonly used abstract data types for keeping track of data and retrieving it in a particular order: *last-in, first-out* (LIFO) for stacks, and *first-in, first-out* (FIFO) for queues.

In lecture, we have seen two efficient strategies for implementing each of the two, in both cases one relying on singly-linked lists, and another relying on arrays.

In this assignment, you will implement both a stack and a queue, using the linked-list-based strategy in both cases, and write a very small bit of client code which uses queues.

In `stack-queue.rkt` we've supplied headers for the classes and function that you'll need to write along with an insufficient number of tests.

Your task

The `ListStack` Class

Your first task is to implement a singly-linked-list-based stack class, which must satisfy the `STACK` interface which we saw in class (without modifications!), and which is provided for you in `stack-queue.rkt`.

Your implementation must satisfy the laws we saw for stacks; in cases where the laws don't mandate specific behavior, your code should raise an error with the message `"empty stack"`.

We also provided you with a struct definition for `_cons` pairs (*i.e.,* linked list nodes), which you should use in your representation. Finally, we also provided one sorry excuse for a test; you'll definitely want to add your own.

The `ListQueue` Class

Your second task is to implement a singly-linked-list-based queue class, which must satisfy the `QUEUE` interface which we saw in class (without modifications!), and which is provided for you in `stack-queue.rkt`.

Note: the definition of the `QUEUE` interface is *commented out*. This is intentional: `QUEUE` is defined in the imported `ring_buffer` library, and having a duplicate definition here would cause an error. Do not uncomment this definion unless you want to see a a confusing error message!

Your implementation must satisfy the laws we saw for queues; in cases where the laws don't mandate specific behavior, your code should raise an error with the message `"empty queue"`.

Technically we saw a few approaches that used singly linked lists to represent queues in class, but only the third allows efficient ($\mathcal{O}(1)$) `enqueue` and `dequeue` operations. As such, this is the strategy we want you to implement.

You should again use our definition for `_cons` pairs in your representation.

We also gave you a pathetic attempt at a test; we highly recommend your own attempts be more thorough.

Managing Playlists

Context: Starting with this assignment, programming assignments now include an open-ended section to give you some practice *using* the data structures you build, and to allow you to get a bit creative. Those are not meant to be especially time consuming or challenging; if you're having a hard time, you may be overthinking something.

One application of queues is to manage playlists in a music player application. When users think of new music they want to listen to, they *enqueue* it, and the music player will get to playing it after it's done with what's already in the queue. Songs are played in a *first-in, first-out* order: the first song to enter the queue is the first one to be played, and the last song to get enqueued is the last one to play.

To get you programming with queues, your last task will be to write a function which simulates an interaction with a music player's queue.

You must write a `fill_playlist` function, which takes an empty queue as an argument, and uses queue operations to add (at least) five of your favorite songs (as `song` structs) to that queue, then uses another queue operation to return the first song the music player should play. If passed a non-empty queue as argument, your function must raise an error with the message `"non-empty queue"`.

In case you're feeling uninspired, or would prefer not to share your musical tastes, feel free to use these five songs from my playlist as I compose this assignment:

- How to Save a Life — The Fray — The Fray
- Hikaru Nara — Goose house — Milk
- Sirivennela — Anurag Kulakarni — Shyam Singha Roy
- APT. — ROSÉ, Bruno Mars — APT.
- He Lei Pāpahi No Lilo a me Stitch — Mark Keali'i Ho'omalu — Lilo & Stitch

Of course, if your function uses only queue operations which are part of the interface, then it can work with any conforming queue implementation. To enforce that restriction, the function's argument is protected with the `QUEUE!` contract, which altogether forbids access outside of what the interface provides.

To test your function, you will need to create an empty queue (your choice of implementation) and pass it to your function.

In addition to testing your `fill_playlist` function with your own `ListQueue` implementation, please also write (at least) one test case in which `fill_playlist` uses the other queue implementation we have seen: a ring buffer. The starter code `import`s the ring buffer implementation we saw in class, so you can use it directly from your tests.

Honor Code

Every programming assignment you hand in must begin with the following definition (taken from the Provost's website;[1] see that for a more detailed explanation of these points):

```
let eight_principles = ["Know your rights.",
     "Acknowledge your sources.",
     "Protect your work.",
     "Avoid suspicion.",
     "Do your own work.",
     "Never falsify a record or permit another person to do so.",
     "Never fabricate data, citations, or experimental results.",
     "Always tell the truth when discussing your work with your instructor."]
```

If the definition is not present, you receive no credit for the assignment.

Note: Be careful about formatting the above in your source code! Depending on your pdf reader, directly copy-pasting may not yield valid DSSL2 formatting. To avoid surprises, be sure to test your code *after* copying the above definition.

[1] `http://www.northwestern.edu/provost/students/integrity/rules.html`

Grading

Please submit your completed version of `stack-queue.rkt`, containing:

- definitions for the two classes and the one function described above,
- sufficient tests to be confident in your code's correctness,
- and the honor code.

Be sure to remove any leftover debugging printing code, comment out any code that would cause infinite loops, and make sure that your submission can run successfully. If your submission produces excessive output, loops infinitely, or crashes, we will not be able to give either you feedback on it or credit for it. **Please run your code one last time before submitting!**

Functional Correctness

We will use four separate test suites to test your submission:

- **Basic stack**: correct LIFO behavior in non-error cases.
- **Advanced stack**: correct behavior in error cases.
- **Basic queue**: correct FIFO behavior in non-error cases.
- **Advanced queue**: correct behavior in error cases, `fill_playlist` can work with multiple queue implementations.

To get credit for a test suite, your submission must pass *all* its tests.

Non-Functional Correctness

For this assignment, the self-evaluation will be specifically looking for:

- Efficiency from the use of the correct representation and operations
- Rigorous checking of error cases
- Representation choices that respect consistency and reduce the risk of bugs

The self-evaluation will "spot check" your implementation's non-functional correctness. For example, to evaluate how extensively you may have tested, it may spot check if you have written a specific test case with a question possibly worded like "Did you write a unit test for X function to test that Y input produces the correct value?". For such a question, you would receive credit if you did write such a test and we could verify its existence in your submission. Specific instructions will be given when the self-evaluation is released.

Mutation Testing

Starting with this assignment, we will be using a cutting edge software engineering technique called *mutation testing* to assess the thoroughness of your test suites.

Mutation testing consists of running a test suite on a series of intentionally-buggy implementations—referred to as *mutants*—and seeing if the tests can catch the bug in each one—referred to as *killing* a mutant. A good test suite kills all the mutants (i.e., catches all the bugs) whereas a poor one leaves some of them to survive (i.e., some of the bugs went undetected!)

We have created a number of intentionally-broken stacks and queues, and will be running the tests in your submissions on them. The more of these mutants your test suite kills, the more thorough it is! To help you refine your approach to testing, you will receive a report with the mutation testing results.

See the syllabus for the specifics on how mutation testing contributes to the *design* track.

Important: The tests in your test suites *must* accept *any* correct implementation of the homework's specification; not just yours. In engineering practice, test suites that "overfit" to a given implementation are brittle and thus best avoided: even minor changes that preserve correct behavior are prone to breaking them.

In particular, your tests must rely solely on methods provided by the interface. If a test uses additional methods you added to your class (for debugging resons, say) beyond those in the interface, that entire test block will be rejected and will not be used when trying to kill mutants. So be sure to stick to the interface and get your tests counted!

Advice: To help you write tests, refer to the **"Testing" video** and the **"A Beginner's Guide to Testing.pdf" document** under "Supplemental Materials" on Canvas for detailed guidance. Keep your test blocks small, and split your testing across multiple test blocks, one for each aspect of the assignment. That way if you made a small mistake in your tests, only the small test block with the mistake will be rejected; as opposed to an entire larger test block that happens to have a mistake in a small part of it.

Hints: Here are a couple non-comprehensive hints to get you started thinking about behaviors to test: (1) does your stack work in LIFO order?; and (2) does your queue work in FIFO order? It is an exercise for you to determine how to confidently test these behaviors.

Stretch Goal

If you'd like a bit of additional practice with stacks and queues, or just want to have a bit of fun with what you built, here's a small problem you can try to solve.

Build a class that represents a text field—the kind you may find in a text editor, or on a web page—which supports the following operations: inserting a block of text at a certain position, deleting a certain number of characters at a certain position, or undoing the last operation (insertion or deletion). Undoing multiple times should undo operations progressively further back in history.

Think about the representation for each of the pieces of your system. Do different choices make for more or less efficient operations? Does combining multiple representations make more efficient operations possible?

Note: This stretch goal is completely optional, and will not be graded. If you do include it in your submission, be careful not to break any of the standard functionality of the assignment (e.g., do not modify the interface, and do not alter operation behavior).