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.

We also provided you with a struct definition for `cons` pairs, which (*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`.

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.

Technically we saw two approaches that used singly linked lists to represent queues in class, but only the second 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 let you 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.

In case you're feeling uninspired, or would prefer not to share your musical tastes, feel free to five songs that have my earworms lately:

-
- Storm Coming — Gnarles Barkley — St. Elsewhere
- The High Road — Broken Bells — Broken Bells
- Bebop — Dizzy Gillespie — For Musicians Only
- I Can't Wait — Nu Shooz — Tha's Right
- A Taste of Honey — Herb Alpert's Tijuana Brass — Whipped Cream & Other Delights

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 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 which uses the other queue implementation we have seen: a ring buffer. The starter code `imports` the ring buffer implementation we saw in class, so you can use it directly from your tests.

Grading

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

- definitions for the two classes and one function described above,
- sufficient tests to be confident of 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.

The outcome your submission will earn will be determined as follows:

- **Got it**: passes all four test suites.
- **Almost there**: passes both basic test suites, and fails a single advanced test suite.
- **On the way**: either passes both basic test suites, or passes both the basic and advanced test suite for one of the two topics.
- **Not yet**: does not achieve “on the way” requirements.
- **Cannot assess**: we could not successfully grade your submission.
- **Missing honor code**: your submission did not include the honor code.

Non-Functional Correctness

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

- Thorough testing, including edge cases
- Rigorous checking of error cases
- Efficiency from the use of the correct representation and operations