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 functions 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 an operation is not defined, your code should raise an error.

We also provided you with a struct definition for `cons` pairs, which (*i.e.,* 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 an operation is not defined, 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

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 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 use these five songs from my playlist[1] as I compose this assignment:

- Stop Them Jah — Augustus Pablo — King Tubbys Meets Rockers Uptown
- Horology — King Gizzard & the Lizard Wizard — Polygondwanaland
- Fuzz Gong Fight — Xiu Xiu — OH NO
- Mo is On — Elmo Hope — Trio and Quintet
- Storm — Godspeed You! Black Emperor — Lift Your Skinny Fists Like Antennas to Heaven!

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. You can find our implementation of ring buffers on Canvas; please copy-paste the `RingBuffer` class to your `stack-queue.rkt` for your tests (and ours too, so leave the ring buffer implementation in when you submit).

[1] I generally prefer to listen to entire albums at a time, but that doesn't make for as interesting an example.

Honor code

Every homework assignment you hand in must begin with the following definition (taken from the Provost's website[2]; 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.

Deliverables

The provided file `stack-queue.rkt`, containing

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

Your code will be evaluated for correctness, resource efficiency, thoroughness, and style.

Submission

Your homework must be submitted via Canvas.

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