Problem 5: Stackland [Email Submission]

By Frederick Qiu

Queralt lives in Stackland, where the only data structure used is... well, the stack. This year, Queralt was accepted to Princeton and is currently taking COS 226, where she's learning about all kinds of different data structures. In particular, Queralt was fascinated by the queue, which she tells her friends about when she goes home for fall break.

News of this mysterious data structure quickly spread throughout Stackland, and people can't wait to try it out. Unfortunately, Queralt had to go back to Princeton before she got a chance to tell anyone how to implement a queue. Can you help the residents of Stackland out?

Part 1 (5 points)

Alice is a software engineer at Facenovel, and would like to implement a queue to automate some large workloads. It doesn't matter to her how long any individual operation takes, but she would like to keep the overall number of stack operations low so the workloads get finished in a reasonable amount of time.

Describe a queue implementation (words or pseudocode) that uses $O(n)$ stack operations to perform n queue operations. Your implementation can only store stack variables. No proof of correctness is required.

How to Submit: Email your complete description to coscon.written.submission@gmail.com with *exact* subject *Problem5aSubmission*. If you must resubmit, *respond to the thread where you sent your original submission; we cannot guarantee that your resubmission will be graded otherwise.*

Part 2 (10 points)

Bob is a quantitative trader at James Street. To trade more efficiently, he wants to build a queue that can handle a sequence of n enqueue operations followed by n dequeue operations. In his case, every nanosecond counts, so he would like to keep the number of stack operations performed per queue operation low.

Implement the code for a queue that uses $O(\log n)$ stack operations per queue operation. You may only use and store primitives and the provided stacks (no arrays!). The items being enqueued cannot be duplicated

and can only be stored on stacks. You can find the starter code on the problems page; you should see `Stack`, a stubbed out `Queue`, and `TestQueue` (for C, you will also see header files for `Stack` and `Queue`). Submit only the `Queue` file.

Hint: There is an implementation that only requires $O(1)$ stack operations for every queue operation except the first dequeue operation.

How to Submit: Email your completed `Queue` implementation (`Queue.java`, `Queue.py`, or `Queue.cpp`) to coscon.written.submission@gmail.com with *exact* subject *Problem5bSubmission*. If you must resubmit, *respond to the thread where you sent your original submission; we cannot guarantee that your resubmission will be graded otherwise.*

Background: Why could this be important?

(Reading this part isn't necessary to solving the problem, but it'll give you some more insight on why the challenges you solved aren't some random silly problems!) Although it seems silly to simulate such a simple data structure as a queue using stacks, being able to do so has important consequences in the world of theoretical CS! For example, it is known that using 6 stacks (and allowing the duplication of items, which we disallow in this problem), you can simulate any queue operation, in any order, using $O(1)$ stack operations. This means that if you want to reason about something that uses stacks and queues, you only need to reason about stacks, since the best-possible queue (up to constant factors) can be simulated using a constant number of stacks.

More generally, doing X using Y is known as a *reduction*. This can be useful when we want to reason about X, but X is very specific or complicated and Y is more general or well-understood.

You might have heard about reductions if you know about NP-complete problems. This class of problems has the property that any NP-complete problem can be reduced to any other NP-complete problem in polynomial time: that is, if you have a polynomial time algorithm to solve one NP-complete problem, you can use that same algorithm to (essentially) solve any NP-complete problem! This works in the other direction as well: if you can prove that one NP-complete problem cannot be solved by any polynomial time algorithm, then no NP-complete problem can be solved by a polynomial-time algorithm.

Another example is in cryptography, as we basically don't know how to make *any* cryptographic scheme that's *provably* secure! However, what allows us to make secure schemes in practice is to reduce these problems to another construct, implement this construct in a way that is secure in practice, then build provably secure schemes assuming practical security of the construct. This is vastly better than trying to create practically secure schemes ad-hoc every time we want to use cryptography. For example, we can reduce the problems of private key encryption, signatures, pseudorandom generators, and more to something called a *one-way function*. If we could construct a one-way function which is secure in practice, we can build all the other schemes with provable security guarantees on top of the one way function (i.e., if the one way function is secure, then the scheme is secure).