PRINCETON COS CON

Problem 5 Solution: Stackland

By Frederick Qiu

1 Solution to First Part

The key observation is that Alice only cares about the total time it takes to perform n queue operations, so we only require an implementation that takes $O(1)$ *amortized* stack operations per queue operation. That is, a single operation could be particularly expensive (taking $\Theta(n)$ operations), but overall these expensive steps are few, so our overall average cost per operation is still extremely cheap. One way to do so is as follows:

```
head := new Stack
tail := new Stack

enqueue(item):
    tail.push(item)

dequeue():
    if head.size() = 0:
        while tail.size() > 0:
            head.push(tail.pop())

    return head.pop()
```

Why would this work? Loosely speaking, the problem with using a stack to simulate a queue is that the order items are popped from a stack is the opposite of the order items are dequeued from a queue. The solution is to have a separate stack for handling the dequeue operations, and when that stack is empty, popping the entirety of the other stack into it, which reverses the elements into the correct dequeueing order.

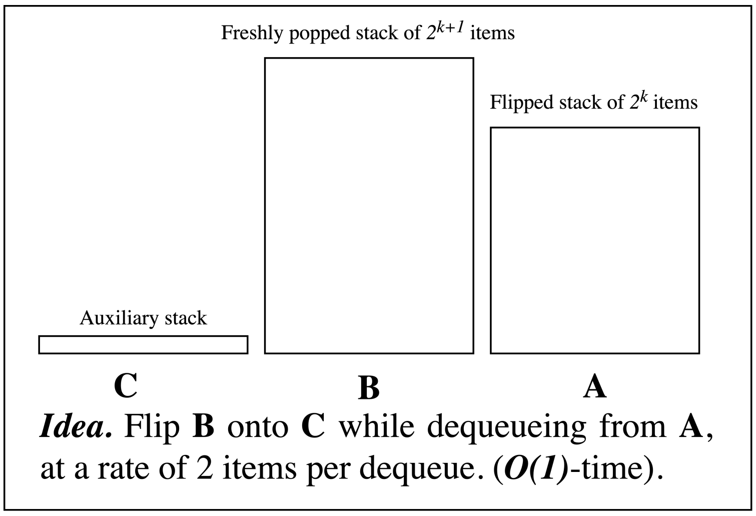
The reason this takes only $O(n)$ time overall is because each item is only the object of a stack operation at most 4 times: when it gets pushed onto `tail` by an enqueue operation, when it gets popped and pushed onto `head`, and when it gets popped by a dequeue operation. So no matter how large n is, or what the type/order of these n operations are, we are doing at most $4n$ pushes/pops from a stack!

2 Solution to Second Part

Simply put, this problem asks for a solution with a better *worst-case* guarantee than the solution to the first part. Our solution from part 1 won't work, because after n enqueues, it will take $2n$ stack operations to flip the entirety of one stack onto another. The trick is to store all the items onto $O(\log n)$ stacks, which then themselves go onto another stack, and flip just $O(\log n)$ stacks instead of n items.

Since we don't know n ahead of time, we can accomplish this by putting 1 item onto the first stack, 2 items onto the second stack, 4 items onto the third stack, and so on, doubling the items on each successive stack. When the first dequeue occurs, we need to retrieve the bottom-most stack, which requires flipping a stack of $O(\log n)$ stacks, which satisfies the desired complexity of a queue operation.

The only other thing to worry about is flipping the items on the stacks to dequeue them in the correct order. Fortunately, we can get around this by maintaining the following invariant: supposing that stack k (which contains 2^k items) is flipped when we begin "dequeueing" from it, by the time we need to "dequeue" from stack $k+1$ (which contains 2^{k+1} items), stack $k+1$ will be flipped. Observe that we need $2 \cdot 2^{k+1} = 4 \cdot 2^k$ stack operations to flip stack $k+1$, since we need to pop and push each item onto an auxiliary stack. Therefore, we can spread this flipping procedure across the 2^k calls to dequeue that happen before stack k becomes empty, which only takes $4 = O(1)$ stack operations per dequeue operation, meeting the desired time bound.


Freshly popped stack of 2^{k+1} items
Flipped stack of 2^k items
Auxiliary stack
C
B
A
Idea. Flip **B** onto **C** while dequeueing from **A**, at a rate of 2 items per dequeue. (***O(1)***-time).

Side Note: I didn't include an explicit "use $O(\log n)$ stacks," as I thought that this would be implicitly required. It turns out that not having this requirement allows for clever (but unintended) solutions that use stacks to construct a singly-linked list or a tree with parent pointers!

3 Grading Criteria and Statistics

Part 1	
5 points	Full Solution
4 points	Somewhat unclear description/implicit assumption that we only need to handle a sequence of n enqueues followed by n dequeues
3 points	Solution which handles n enqueues followed by n dequeues, but not intermixed calls
0 points	Solution which cannot handle n enqueues or n dequeues in a row

Part 2	
10 points	$O(1)$ stack depth, working code (3 graduate student teams achieved this)
8 points	$O(\log n)$ stack depth, working code (no one achieved this)
6 points	$O(\log n)$ stack depth, partially implemented and detailed comments (2 undergraduate student teams achieved this)
5 points	Making significant progress on an $O(1)$ stack depth solution, or a working $O(n)$ stack depth solution
2-4 points	Varying levels of detail in describing an $O(\log n)$ stack depth implementation
1-2 points	Varying levels of detail in describing an $O(n)$ stack depth implementation
0 points	Using the part 1 solution or a solution which requires knowledge of n in advance