Interview Cake

Implement a queue↴ with 2 stacks.↴ Your queue should have an enqueue and a dequeue method and it should be "first in first out" (FIFO).

Optimize for the time cost of m calls on your queue. These can be any mix of enqueue and dequeue calls.

Assume you already have a stack implementation and it gives $O(1)$ time push and pop.

Gotchas

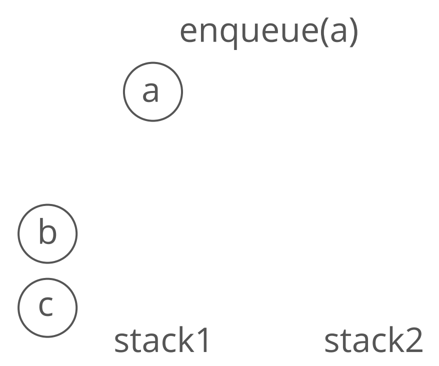
We can get $O(m)$ runtime for m calls. Crazy, right?

Breakdown

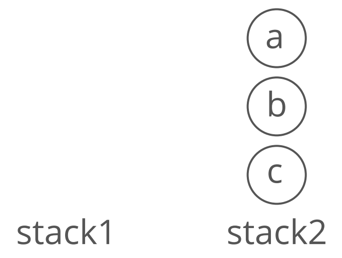
Let's call our stacks `stack1` and `stack2`.

To start, we could just push items onto stack1 as they are enqueued. So if our first 3 calls are enqueues of `a`, `b`, and `c` (in that order) we push them onto `stack1` as they come in.

But recall that stacks are last in, first out. If our next call was a `dequeue()` we would need to return `a`, but it would be on the bottom of the stack.



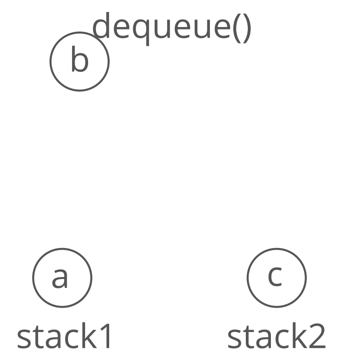
Look at what happens when we pop `c`, `b`, and `a` one-by-one from `stack1` to `stack2`.



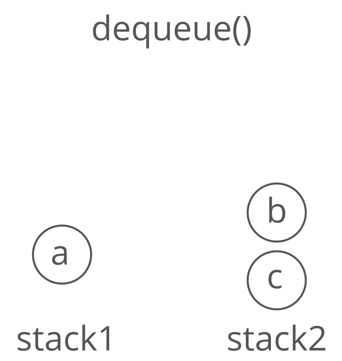
Notice how their order is reversed.

We can pop each item 1-by-1 from `stack1` to `stack2` until we get to `a`.

We could return `a` immediately, but what if our next operation was to enqueue a new item `d`? Where would we put `d`? `d` should get dequeued after `c`, so it makes sense to put them next to each-other . . . but `c` is at the bottom of `stack2`.



Let's try moving the other items back onto `stack1` before returning. This will restore the ordering from before the dequeue, with `a` now gone. So if we enqueue `d` next, it ends up on top of `c`, which seems right.



So we're basically storing everything in `stack1`, using `stack2` only for temporarily "flipping" all of our items during a dequeue to get the bottom (oldest) element.

This is a complete solution. But we can do better.

What's our time complexity for m operations? At any given point we have $O(m)$ items inside our data structure, and if we dequeue we have to move all of them from `stack1` to `stack2` and back again. One dequeue operation thus costs $O(m)$. The number of dequeues is $O(m)$, so our worst-case runtime for these m operations is $O(m^2)$.

Not convinced we can have $O(m)$ dequeues and also have each one deal with $O(m)$ items in the data structure? What if our first $.5m$ operations are enqueues, and the second $.5m$ are alternating enqueues and dequeues. For each of our $.25m$ dequeues, we have $.5m$ items in the data structure.

We can do better than this $O(m^2)$ runtime.

What if we didn't move things back to `stack1` after putting them on `stack2`?

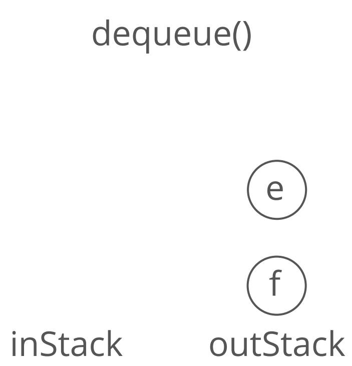
Solution

Let's call our stacks `inStack` and `outStack`.

For enqueue, we simply push the enqueued item onto `inStack`.

For dequeue on an empty `outStack`, the oldest item is at the bottom of `inStack`. So we dig to the bottom of `inStack` by pushing each item one-by-one onto `outStack` until we reach the bottom item, which we return.

After moving everything from `inStack` to `outStack`, the item that was enqueued the *2nd* longest ago (after the item we just returned) is at the top of `outStack`, the item enqueued *3rd* longest ago is just below it, etc. **So to dequeue on a non-empty `outStack`**, we simply return the top item from `outStack`.



With that description in mind, let's write some code!

JavaScript ▾

```
class QueueTwoStacks {
  constructor() {
    this.inStack = [];
    this.outStack = [];
  }

  enqueue(item) {
    this.inStack.push(item);
  }

  dequeue() {
    if (this.outStack.length === 0) {

      // Move items from inStack to outStack, reversing order
      while (this.inStack.length > 0) {
        const newestInStackItem = this.inStack.pop();
        this.outStack.push(newestInStackItem);
      }

      // If outStack is still empty, raise an error
      if (this.outStack.length === 0) {
        throw new Error("Can't dequeue from empty queue!");
      }
    }
    return this.outStack.pop();
  }
}
```

Complexity

Each enqueue is clearly $O(1)$ time, and so is each dequeue when `outStack` has items. Dequeue on an empty `outStack` is order of the number of items in `inStack` at that moment, which can vary significantly.

Notice that the more expensive a dequeue on an empty `outStack` is (that is, the more items we have to move from `inStack` to `outStack`), **the more $O(1)$-time dequeues off of a non-empty `outStack` it wins us in the future**. Once items are moved from `inStack` to `outStack` they just sit there, ready to be dequeued in O(1) time. An item never moves "backwards" in our data structure.

We might guess that this "averages out" so that in a set of m enqueues and dequeues the total cost of all dequeues is actually just $O(m)$. To check this rigorously, we can use the accounting method,↴ **counting the time cost *per item* instead of per enqueue or dequeue**.

So let's look at the worst case for a single item, which is the case where it is enqueued and then later dequeued. In this case, the item enters `inStack` (costing 1 push), then later moves to `outStack` (costing 1 pop and 1 push), then later comes off `outStack` to get returned (costing 1 pop).

Each of these 4 pushes and pops is $O(1)$ time. **So our total cost *per item* is $O(1)$**. Our m enqueue and dequeue operations put m or fewer items into the system, giving a total runtime of $O(m)$.

What We Learned

People often struggle with the runtime analysis for this one. The trick is to think of the cost *per item passing through our queue*, rather than the cost per enqueue() and dequeue().

This trick generally comes in handy when you're looking at the time cost of not just one call, but "m" calls.