

****Note: Animations did not translate (enqueue, dequeue, etc) for stacks/queue diagrams**

Implement a queue ↴

A **queue** stores items in a first-in, first-out (FIFO) order.

Picture a queue like the line outside a busy restaurant. First come, first served.

enqueue(b)



Worst Case

space	$O(n)$
enqueue	$O(1)$
dequeue	$O(1)$
peek	$O(1)$

Strengths:

- **Fast operations.** All queue operations take $O(1)$ time.

Uses

- **Breadth-first search (/concept/bfs)** uses a queue to keep track of the nodes to visit next.
- **Printers** use queues to manage jobs—jobs get printed in the order they're submitted.
- **Web servers** use queues to manage requests—page requests get fulfilled in the order they're received.
- **Processes** wait in the CPU scheduler's queue for their turn to run.

Implementation

Queues are easy to implement with linked lists (/concept/linked-list):

- To enqueue, insert at the tail of the linked list.
- To dequeue, remove at the head of the linked list.

You *could* implement a queue with an array (/concept/array) or dynamic array (/concept/dynamic-array), but it would get kinda messy. Try drawing it out. You'll notice that you'd need to build out a "scoot over" or "re-center" operation that automatically fires when your queue items hit the bottom edge of the array.

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?

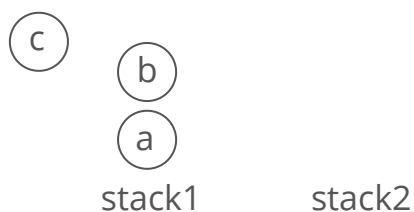
Breakdown

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

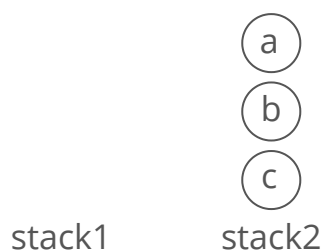
To start, we could just push items onto `stack1` as they are enqueued. So if our first 3 calls are enqueues of `a`, `b`, and `c` (in that order) we push them onto `stack1` as they come in.

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.

enqueue(c)



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

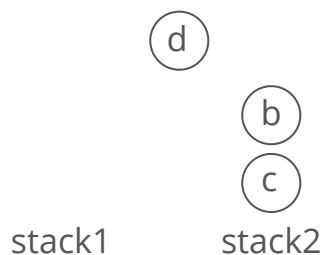


Notice how their order is reversed.

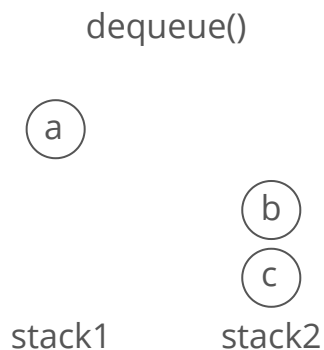
We can pop each item 1-by-1 from stack1 to stack2 until we get to a.

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.

enqueue(d)



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?

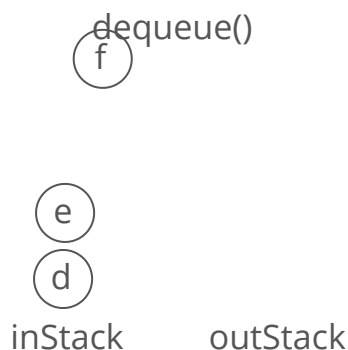
Solution

Let's call our stacks `in_stack` and `out_stack`.

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

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

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



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

```
class QueueTwoStacks(object):
```

Python 3.6 ▼

```
    def __init__(self):
```

```
        self.in_stack = []
```

```
        self.out_stack = []
```

```
    def enqueue(self, item):
```

```
        self.in_stack.append(item)
```

```
    def dequeue(self):
```

```
        if len(self.out_stack) == 0:
```

```
            # Move items from in_stack to out_stack, reversing order
```

```
            while len(self.in_stack) > 0:
```

```
                newest_in_stack_item = self.in_stack.pop()
```

```
                self.out_stack.append(newest_in_stack_item)
```

```
            # If out_stack is still empty, raise an error
```

```
            if len(self.out_stack) == 0:
```

```
                raise IndexError("Can't dequeue from empty queue!")
```

```
        return self.out_stack.pop()
```

Complexity

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

Notice that the more expensive a dequeue on an empty `out_stack` is (that is, the more items we have to move from `in_stack` to `out_stack`), **the more $O(1)$ -time dequeues off of a non-empty `out_stack` it wins us in the future.** Once items are moved from `in_stack` to `out_stack` 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,¹

The accounting method can be used for computing time complexity for things like "the cost of m operations on this data structure."

In the accounting method, you simply look at the time cost incurred by each *item* passing through the system instead of the time cost of each operation.

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 `in_stack` (costing 1 push), then later moves to `out_stack` (costing 1 pop and 1 push), then later comes off `out_stack` 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.

Ready for more?

Check out our full course ➡

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.