CS 120: Intro to Computer Programming II

In-Class Activity - 13 Stacks and Queues - Day 1

Activity 1 - Turn in this one

The simplest version of a stack is one that's based on an array. Today, you're going to write a class that encapsulates that logic.

Write a class named `ArrayStack`, which represents the Stack ADT ("Abstract Data Type"), and implements it using a stack. Implement the following methods:

- The constructor takes no parameters. A newly created Stack should be empty.
- `is_empty()` returns a boolean value (`True` or `False`).
- `push(val)` pushes a new value onto the stack. `assert` that the value isn't `None`; otherwise, you should accept any input value.
- `pop()` pops one value off of the stack, and returns it. `assert` that the stack is not empty.

Write some code to test out your class: that is, create a couple of stack objects, push and pop a few values, and use `assert` statements to verify that the code works as expected.

Then, discuss the performance of your code. How long does each operation take to run?

Activity 2 - Turn in this one

We're about to implement a linked-list version of a stack. But the obvious way to do this - the way that most programmers try it, at first - its pretty poor. Let's find out why.

Normally, when people try to implement a linked list stack, they use the `head` pointer as the root of the stack, and all of the updates - pushes and pops - happen at the tail.

Discuss with your group (and turn in your notes to your TA) about why this is a bad idea. What performance problems does it cause? Are there any ways to get around this, to improve performance?

Activity 3 - Turn in this one

Ok, let's implement this **the right way.** Write a new class, `ListStack`, which supports the exact same Stack ADT. This means that it must support all of the same methods as `ArrayStack`, with exactly the same arguments, the same return values, and the same `assert` statements. But this time, the internal implementation must be a linked list.

But here's the key feature, which makes it efficient: all of the pushing and popping must be done **at the head.**

Once your code is written, test it out. If you've written things correctly, you can simply re-use your testing code from Activity 1: just replace the line(s) that create the new stack(s). Create `ListStack` instead of `ArrayStack`, and it should work perfectly.

Discuss with your group the big-Oh performance of all of your methods in this class, and turn that in to your TA. But don't bother with confirming your analysis - we need to save some time.

(activity continues on the next page)

Activity 4 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Now, write an `ArrayQueue` class. This models the Queue ADT, and must provide the following methods:

- Constructor takes no parameters, and creates an empty queue
- `is_empty()` returns a boolean
- `enqueue(val)` adds a new element (`assert` that it is not `None`)
- `dequeue()` removes an element from the queue and returns it.

NOTE: It's very hard (maybe impossible???) to implement `ArrayQueue` in such a way that it is $O(1)$ for **all** operations; it's easy to make **some** of them fast, but others will be slow.

Discuss with your group why this is the case, and turn that (along with your `ArrayQueue` class to your TA.

Activity 5 - Optional

OPTIONAL. Complete this if you have time, and turn it in. If you don't have time, you may report to your TA that you ran out of time.

Finally, write a `ListQueue` class. Interestingly, it **is** possible write a version of `ListQueue` which has $O(1)$ performance for all of the methods.

Do your best to find the very-efficient implementation. But if you can't find it, that's OK - it's alright to write a version which has $O(n)$ performance for one of the methods.

Challenge Activity - Do not turn in this one

It can be difficult to implement a full, general queue in an array. Yet arrays are extremely fast and memory efficient. Can we find a compromise?

A **ringbuffer** is a queue with a simple limitation: it has a firm upper limit on how many elements it can store. The reason that this is attractive is because we **never need to shift elements;** instead, we will allow the "first" element in the queue to be **anywhere.** Basically, when you queue up a new data item, it lands somewhere, and then stays there forever, until you dequeue it. Thus, the code must keep track of the **position** of the first and last elements in the queue. And, you must handle "wrapping" - since it's legal for the queue to wrap around, from the end of the array to the front.

Start by drawing a picture. Draw an array, add a few elements to it (leaving some empty, unused spaces) and then "dequeue" a few. Show how it's possible to dequeue an element without shifting anything - if we keep track of the current position of the "first" element in the queue.

Now, implement a class `RingBuffer`, which implements the following methods:

- The constructor takes a `size` parameter, which indictes the length of the array to pre-allocate. **Never resize this array, so long as the ringbuffer exists.**
- `is_empty()`
- `enqueue(val)` works the same as in an ordinary Queue, except that it **returns a boolean:** `True` if the element was queued, or `False` if there wasn't space.
- `dequeue()` works the same as in an ordinary Queue.