<u>Tests</u>:

When I say that something is an *n* entry structure, I mean that that structure should end up with *n* entries, not start with them. I based this off of the idea from Project 2 to test our methods on Linked Lists with no, one, and many entries.

I tested the creation of an empty Deque, Stack, and Queue. I did this so that I could prove the structures were being created correctly, and that the generation of them wasn't what was causing problems in later tests. I also did this to avoid being redundant in including a no entry test for a method whose functionality would be identical to other methods (ie. not pushing anything forward and not pushing anything back to make sure push front and back worked in an empty list). The reason this redundancy was so possible is outlined below.

I tested many of the following methods with structures containing no entries, one entry, and three entries to ensure that the method would work no matter the size of the structure it was performing on. This proved to be an important aspect to the test, because oftentimes my method would work on a structure with no or one entries, but it would not work on structures with multiple entries (for example, the push method of a Queue could possibly work for one entry, but maybe that entry would be erased by pushing another entry into the queue due to a faulty implementation).

I tested the length of empty, one entry, and three entry Deques, Stacks, and Queues to ensure that the length was being properly calculated for each structure. This was an important test because I had to ensure that the method worked in case the user called for the length and to make sure incorrect length wasn't going to cause later methods to fail.

I tested push functions (push_front and push_back for Deques and enqueue for Queues) in one and three entry Deques, Stacks, and Queues. I did this solely to ensure that the methods worked for their respective structures. I didn't need to test it with a no entry list because of the general no entry test (which would be functionally the same).

I tested pop functions (pop_front and pop_back for Deques and dequeue for Queues) by testing pop on empty, one entry, and three entry lists. I included empty list because I wanted to make sure it returned None instead of returning an IndexError. I also included a test to ensure that said empty lists contents did not change from popping a value that wasn't there. To take full advantage of testing structures with three entries, I popped two values to ensure popping once would mess up the structure and not allow for another pop. I also ensured that popping returned the correct value on the first (in one and three entry lists) and second pops. This decision was made to make sure pop returns the correct value no matter how much you use it.

Next, I tested peek (peek_front and peek_back for Deques) on empty, one entry, and three entry lists in Deques and Stacks. I included empty list because I wanted to make sure it returned None instead of returning an IndexError. I also tested that peek was returning the correct values for one and three entry structures just to make sure the method was correctly serving its purpose. Finally, I ensured that the structure wasn't modified by the peek method by using peek on a structure with one entry in it and making sure the structure matched what it was before using peek.

<u>Worst-Case Performance</u>:

The performance for __str__ in Deque is O(n) because both the array and linked list implementations use loops to iterate through every entry in themselves to be able to print. Stack and Queue just call for the __str__ method of the Deque they're built from, so they also use O(n) as well.

The __len__ for Deque, Stack, and Queue all lead back to Deque keeping track of how many things have been entered (in both the Array and Linked List versions), so that value is just called and the method works in O(1).

The __grow method in the Array Deque has O(n) performance because the method must iterate through every entry in the array in order to transfer those values to the new, larger array.

push_front calls __grow in the Array Deque, but otherwise, it works at O(1) efficiency, so that gives it O(n) performance in its worst-case scenario. The performance of the Linked List Deque, however, is O(1) because it always inserts at the front. This is also the performance of the push method in Stack, which just calls this method.

push_back works similarly: the Array Deque has O(n) performance when it must call __grow, while Linked List Deque knows to insert directly at the end without the need for iteration. This is also the performance of enqueue in Queue, which just calls push_back.

pop_front has O(1) in both implementations: array just erases a value at a known index while linked list dodges the need for iteration by consistently only being tasked to remove the first value. This is the same performance as the pop method in Stack, which just calls this method. In addition, this has the same performance as dequeue from Queue, which also just calls pop_front.

pop_back has O(1) in the array implementation because it, again, simply removes a value at a known index. However, the linked list implementation requires through the entire list to get to the end, where it will then remove the last value. This could be fixed by adding a special condition to the linked list class for how to handle an index at the tail position, but it wasn't included in our original model for a linked list.

peek_front and peek_back have the same performance as pop_front and pop_back respectively for similar reasons (the only difference being that instead of removing the value at the index, they return it). peek_front also has the same performance as the peek method in Stack, as peek simply calls peek_front from the Deque it's built on.

The Towers of Hanoi have potentially catastrophic performance troubles. The problem takes exponentially longer the more discs you have the program solve (the program doubles in length to complete every time *n* increases by one, which would give the problem about 2^n plus an unknown constant that probably changes based on your computer's power if you're trying to use the equation to calculate how long it would take in seconds to complete execution. In big-oh, it would be $O(2^n)$). This is why computer scientists try to only use recursion when necessary, exponential performance is the worst kind of performance.

<u>On Not Raising Errors</u>:

I think it was mostly appropriate to not raise errors. On the one hand, a user might be confused why they're getting None when trying to pop or peek an empty structure, especially if they don't know anything about the software they're using. Only a person with internal knowledge of how the program works could be certain that None is returning because the list is empty, and that's uncertainty that isn't good for a final product. However, many users would probably be able to guess why nothing was being returned, so it's still usable. I think it makes sense that an IndexError wouldn't be appropriate for a program in which the user is not inputting indices, however, I think some kind of notification would be appropriate. Instead of returning None, a message could return explaining what happened to the user and why. This has its own flaws (potentially breaking a user who is trying to do calculations using our structure where None is a perfectly valid value to have returned but a message is not), but returning None is not a perfect solution either. I'd recommend that there be an explanation to the user as to what None being returned means, and then none of the user's activities would have to be interrupted/ruined by a built-in explanation.