Explanation of Test Cases

In my testing file, I separated my testing into three different sections, one for deques, one for stacks, and one for queues. Each of these sections also had subsections, including push, length, pop, peek, and None sections. I also began each section by testing the string function on an empty deque, stack, or queue.

Deque

First was my deque testing. This section was the longest, as deques have the most functionality out of the three data structures.

Within the push section, I had to test both push_front and push_back in all of the different cases. This started with pushing onto an empty deque, then a deque with one item, then a deque with two items. Additionally, I tested both push_front and push_back together in different combinations, such as front, back, front and back, front, back. The purpose of this was to ensure that any combination of pushing front or back would always have the intended outcome. I may not have needed to include the testing for pushing an item onto a deque with more than one item, as each action would be the same, however I wanted to be on the safe side. Additionally, this is why I did not include tests past deques with two items, as the functions can only modify the front and the back, so anything in the middle of a large deque would make no difference. Throughout all of this, the string function was also being tested, as I needed to use the string function to check the contents of the deque after calling push_front or push_back.

Next, when testing the length function, I had tests for an empty deque, deques with one item, and deques with two items. Like with the push section, I had different combinations of push_front and push_back used to create the deque. This way, I would be testing both the length function and the push functions, as the push functions need to be able to increment the size. I may not have needed to use different combinations, but again I wanted to be on the safe side, as using one after the other may have had unforeseen consequences on the length.

I then tested the pop functions in a similar fashion to the example from class. For each combination of pop_front and pop_back, I needed to have three different tests: one for testing the return value, one for testing the remaining deque, and one for testing the resulting length. This would ensure that all three components of the pop functions are performing as intended. I tested popping from deques with one element, deques with two elements, and deques with three elements. With deques with one element and deques with two elements, I only included one pop function per test, however I had two pop functions with the deques with three elements. Originally, I did not include back to back pop operations, but I realized that I had messed up something in my code that caused two consecutive push_back() calls to mess up the contents of the deque. For this reason, I decided it would be best to add this to my testing, as none of the tests had caught this the first time running through them.

My peek section was very similar to the pop section, except I had no cases with peeking twice, as I felt this was unnecessary. Like the pop section, I tested the return value, the remaining deque, and the resulting length. This ensures that peeking returns the desired value, but does not alter the deque in any way.

Last, I had the None section. This section is an alternative to an error section, as there are no errors raised in this project. Instead, None values are returned whenever the function cannot be called properly. This section included popping and peeking (both front and back) with an empty deque, as well as popping twice from a deque with one item. In either case, the last pop will be from an empty deque with a return value of None. These tests are to ensure that None is actually returned and that no errors are raised. The other functions do not have any situations like this, as pushing never returns a value and string and length would simply return '[ ]' and '0', respectively.

Stack

For the explanation of the test cases for my stack and queue sections, I will be much briefer, as the deque section includes all of the same cases as the other two sections. I will focus more on why certain cases were left out or different from before.

First, with the push section, since there is only one push operation, there are many fewer required tests. The push function works the same way that push_front() did for deques, so I only needed to include one test for each stack size. The three stack sizes I used were empty stacks, stacks with one item, and stacks with two items, just like with deques. Once again, the string function and the push function were being tested simultaneously.

Next, the length section also only included three tests and these tests were implemented on stacks of the same sizes as push was. Like before, the incrementing of size for the push operation was also being tested.

With the pop section, once again, there was only one pop operation that worked the same way as pop_front() did for deques. I still had to include three tests for each situation, however, as I would still have to check the return value, the remaining contents, and the length. The three situations I used were popping once from a stack with one item, popping once from a stack with two items, and popping twice from a stack with three items. My reasoning for including the popping twice situation remained the same as before.

My peek section was the exact same as the pop section, except I once again left out a peeking twice situation. I had six tests total, using the same three tests on each situation.

Lastly, my None section only had two tests, one for popping from an empty stack and the other for peeking at an empty stack. I did not decided to pop twice from a stack of one like I did with my deque section, as my main reasoning then was to test popping front and then back or back and then front, but here that was not possible.

Queue

Finally, I had my queue section, which was very similar to the stack section.

First, I had an enqueue section, which works the same as push_back() did for deques. This is where the stacks and queues are different, however they included the same tests, one for each of the three situations (empty, queue with one, and queue with two).

The remainder of the queue section was the exact same as the stack section, as length and peek did not differ and the only difference between pop and dequeue was the naming.

Analysis of Worst-Case Asymptotic Performances

I will go about my performance analysis by explaining each method group. Some methods have the exact same functionality as others for a different data structure, so I will group them together.

(all four)__init__():
For all four implementations, the __init__ function operates in constant time. In Linked_List_Deque, Queue, and Stack, __init__ only has to create an instance of a Linked_List or a Deque. With Arrary_Deque, it needs to set the

capacity, contents, front, back, and size, but all of these are done in constant time.

(all four)__str__():
Since the __str__ method from project 2 was quadratic time, it is for project 3's Linked_List as well. It had a for loop iterating through everything in the list and it also had string concatenation, which is also linear time. The Array_Deque had very similar implementation for its __str__ method and achieves quadratic time as well. It also had a for loop iterating from front to back with string concatenation for each step. Since both implementations achieve quadratic time with the __str__ method, Stack and Queue can also achieve quadratic time, as they use the __str__ method from their self.__dq.

(all four)__len__():
Since I am keeping track of the self.__size in both of my deque implementations, I can achieve constant time with my __len__ method. Like before, Stack and Queue also inherit that constant time.

__grow():
In Array_Deque, my __grow method must pop the contents from self, the deque, and copy them to a temporary array. This action is a linear time action. Next, it reassigns size, contents, capacity, front, and back. All of these can be done in constant time, except for adding empty cells to the contents. However, this action is also done in linear time. Overall, there are three linear time actions being taken here: one to create an empty temporary array, one to add the contents from self to the temporary array, and one to add empty cells to the new self.__contents. This means that three actions happen for every item in the deque, which is still constant time.

(Linked_List_Deque)push_front():
Linked_List_Deque's push_front has constant time performance, as appending an element and inserting an element at position 0 both have constant time performance due to having access to the self.__header.

(Array_Deque)push_front(),push():
Everything in push_front can be done in constant time, as it is just assigning front and back, adding to the contents, and incrementing the size. However, if the capacity is full, Array_Deque's push_front calls __grow, which is linear time. This means that in its worst case, Array_Deque's push_front has linear time performance. Additionally, Stack's push method relies on push_front. It is possible that it inherits push_front from Linked_List_Deque, but in the

worst case, it inherits from Array_Deque, which means that push is also done in linear time.

(both deques)pop_front(),pop(),dequeue():
All four implementations have the same operation when it comes to pop_front, pop, and dequeue. Additionally, they all have the same performance of constant time. With Array_Deque, a value is fetched from the front of the deque and front and size are reassigned. Depending on the circumstances, back may also get reassigned, however all of this can be done in linear time. With Linked_List_Deque, an element is removed from position 0, which is also a constant time operation. Because of this, regardless of the deque they inherit, both Queue and Stack have constant time performance for dequeue and pop.

(both deques)peek_front(),(both queue and stack)peek():
Just like with pop_front, peek_front is also a constant time operation. With Array_Deque, the value at the front of the contents is returned. With Linked_List_Deque, get_element_at(0) is called. Both of these are done in constant time, and therefore, peek is also done in constant time for both Queue and Stack.

(Linked_List_Deque)push_back():
Just like with push_front, push_back is also a constant time operation for Linked_List_Deque. Since we have access to self.__trailer, we can perform a constant time operation at the back of the list, making Linked_List's append_element, and therefore Linked_List_Deque's push_back, constant time.

(Array_Deque)push_back(),enqueue():
Just like with push_front, everything except for the grow method is a constant time operation, but front and back are also switched in this case. This means that in the worst case, push_back is done in linear time. Additionally, if Queue inherits from Array_Deque, enqueue is also a linear time operation in its worst case.

(both deques)pop_back():
Just like with pop_front, pop_back is also a constant time operation for both deque implementations. Additionally, the reasoning for this is the exact same, except front and back are switched and Linked_List removes from self.__tail.previous.

(both deques)peek_back():

peek_back has the same situation as pop_back, as it is just like peek_front, but front and back are switched. Once again, both implementations have constant time performance.

Performance Observations for Towers of Hanoi

In my implementation for the Towers of Hanoi, the Hanoi function is done in linear time, as the while loop gets longer as the number of discs increases. Then, the base case and the recursive case both include a push and a pop operation. The recursive case also includes two Hanoi_rec calls, but with one less disc than before. Because of this, I would expect the function to take about twice as long as the n-1 case before it. After running the function several times for different n values, this was my result with time in seconds:

Test Run	n=1	n=2	n=3	n=4	n=5
#1	.000121	.000254	.000394	.000794	.001698
#2	.000121	.000192	.000481	.000809	.001631
#3	.000120	.000201	.000381	.000820	.001578

As we can see, increasing n by 1 did in fact cause the time to increase by about a factor of 2. This makes sense, as each recursive case is essentially going through the previous case twice.

Discussion About Raising Errors

In this project, I feel that it was not appropriate to not raise exceptions. My reasoning behind this is the fact that our data structures can hold the value 'None', so there is a level of ambiguity between an empty data structure and one that contains the value 'None'. If we were to instead raise an error, then there would no longer be any ambiguity with the contents of the data structure from the perspective of the user. I feel like this is similar to the situation where we decided to point front and back to None for an empty structure. That way, there is no ambiguity between a data structure with no elements and one with one element. I do understand why we might not raise exceptions, however, as we are not giving the program an index that does not exist as an input but are instead calling a function with no parameters. Since there are no parameters, it is debatable whether the user is "doing something wrong" or not. I say this because we raised an error in the last project when the user was "at fault" for providing an invalid index.