In this project I divided up the test functions into three main sections. Each one of these sections checked a different implementation of a deque. Each type of deque was further broken down into groups based on what each test was trying to look for in each method. These groups were broken down with three #'s and a new line for easier visual identification.

The first implementation I tested was the deque structure. This structure should have been able to utilize every method that needs to be present when a deque is created with no restrictions. This includes the **peek_front, peek_back, push_front, push_back, pop_front, pop_back**, the length, grow, and string function. Both the length and the string functions weren't explicitly tested for because they are essential for every test, and any problems within these functions would cause errors on every test case. Moving forward, for tests cases involving deques, I will abbreviate by only using **peek, pop, or push**, while the actual methods include both **_front** and **_back** at the end of them.

The first subset of tests for the deque structure were **peek, pop, and push** for both the front and back of the deque while it was completely empty. All of these tests checked to see if the methods could correctly handle returning either a blank list or a list with one value inside. If there is no value inside of the deque already, the **peek and pop** methods should return a blank list. If **push** is called, the list should contain just a single value inside of it.

The second subset of tests for a deque was to check **pop** and **peek** when there was a single value already inside of the deque. The methods should behave normally, and they are isolated from any problems that may occur with the grow function, because only adding one value keeps the capacity the same. This is the simplest test and a failure could mean a problem with the initialization of the deque. This group of tests also checks the **push** methods with multiple calls to also check the grow method at the same time. If the deque uses an array, the grow method will be called once and the items in the deque should remain in order.

The third group of tests checks the **pop** and **peek** methods with deques with a size of four. This means the **grow** function would have been called twice for an array deque. These tests check if the methods are pulling data off of the correct ends of the deque and are properly maintaining the positions of front and back through multiple calls to **grow**. The tests also check that **peek** is not modifying the data while still returning the value at the correct position. **Pop** is tested to ensure that it pulls off the correct value from the deque and the size of the deque is correctly modified.

The fourth group tests deques that have been filled with 4 values and then emptied. This mainly serves to check the grow function to ensure that the positions of front and back are being maintained after two calls. The emptied deques should have a size of zero and a capacity of four. The methods **pop** and **peek** should return None because there isn't anything in the deque after four calls to **push** then four calls to **pop**. These tests also ensure that there are no residual values that are being left over inside of the structure after everything should have been removed.

The last two implementations I tested were the stack and queue structure. I have grouped these two implementations together because the tests for them are almost identical and to avoid repetition. The only significant difference being instead of **pop and** push methods, there is **dequeue** and **enqueue**. Also, the string representation of queue displays the most recent enqueued item at the beginning of the string instead of the back. The stack structure should have been only able to push items onto the front of the structure and could only pop items off of the front as well. **Peek** should return the most recent value pushed onto the stack and will return none if the stack is empty. For queue structures, the methods should have only been able to push values on to the front and pop them off of the back. Peek should return the value that has been inside of the queue for the longest amount of time. The **len** and **str** methods again were not tested for because they are already included in every test case.

The first subset of tests for this structure checked the **push, pop, and peek** methods while the stack was empty from the start. The **peek** and **pop** methods should return nothing and the stack should remain the same and display *[ ]* as the output.

The second set of tests verified that **peek and pop** would function when there was only one value in the stack and when the size and the capacity were the same. The **push** method was tested to ensure that multiple calls in a row would place the values in the correct order when they were output as a string. I arranged the string, so the top of the stack is the first value to appear, with the bottom of the stack being the last value.

The third set of testing checked that both **peek** and **pop** worked while there were a larger number of values in the stack. After two possible calls to **grow**, the methods should return the correct value which is in the top position of the stack. The fourth set of testing was very similar to the testing for the deque structure and checks the **peek** and **pop** methods when the structure is filled then emptied completely. All of the methods should only return none because there are no values that can be pulled out of the stack.

Many of the methods present in the deque, queue, and stack implementations have either linear time or constant time. This is because the user's access to all of the methods is relatively restricted and only a few of the methods have to iterate through the structure. For all of the **init** methods, only basic variable assignments are made and the structures are initially created, so it is in constant time with no worst or best case. All of the **string** methods operate in linear time in the worst case because they use a **for loop** to iterate through the length of the structures to pull out each individual value and add it to a string. The **len** methods are in constant time because their only job is to return the value of the **__size** variable which requires no iteration.

The methods **push_front, push_back, push,** and **enqueue** all require exponential time in the worst case because this could require infinite calls to the **grow** method. The **grow** method operates in linear time because it needs to iterate through the entire structure to create a new structure in order from front to back with twice the original capacity. The **push and enqueue** methods need to call the **grow** method as the size of **n** increases and this is compounded by the iteration in the **grow** method which causes exponential time. However, this problem disappears if a linked list instead of an array is used to construct the structure.

The methods **pop_front, peek_front, pop_back, peek_back, pop, peek,** and **dequeue** all require constant time even in the worst case. All of the implementations are either using a doubly linked list or an array, so accessing both ends of the structure can occur in constant time. By nature, arrays can access any value in constant time, and a doubly linked list only have to iterate one time regardless of the size of the structure. The **pop, peek, and dequeue** methods in **Stack and** Queue are operating off of calls to the **pop_front, peek_front, pop_back, peek_back** methods in Deque which are also in constant time. In summary, the methods that need to add values to the structure operate in exponential time in the worst case, and the methods that remove or look at values in the structure operate in constant time.

My performance observations for the Tower of Hanoi were that the function is exponential because of the recursion that takes place. There are two lines of code where the function calls itself, and this means that the entire function has a performance of O(2^n – 1) which can also be written as O(2^n). The function has to iterate through every value once for each recursion. This means that for n disks, the recursion has to be called two times for every n value, which results in 2^n. Because the base case is for one disk where the disk is moved to the correct destination in one move, the entire performance of n disks is O(2^n – 1).

Overall, I agree with the decision to not raise exceptions. I think that for most of the structure and methods, there is no way for the user to input an invalid value. Since the methods the user can access is controlled, there is a limited amount of ways to insert and retrieve values from the structure. The only possible way to create an error would be in the syntax of the code the user has written. For example, if there is an invalid value being inserted for a call to **push or enqueue**. However, I think this type of error can easily be seen and corrected without the need for any type of exception handing. However, some exceptions could have been to allow the structure to ignore invalid inputs for **push and enqueue**. I do not think that raising exceptions is completely necessary in this case, but they could have been helpful in making the structures more robust.