TEST CASES:

Part 1: Deque Test Cases

I first checked to see if a regular empty deque was printed properly with string. I then pushed (both front and back equally) one parameter and checked to see if the string matched properly with the specified method used to push. I then did push front then push back and vice versa to see if the str was printed properly and the deque was created properly. I did the same but then did it in the case of push front twice and then push back to see if the deque was working as expected, vice versa.

My following test cases tested to see if pop worked properly. I pushed three values via the front (and back) and then popped from the front (and back) to see if what was left in the deque was what I expected to see. I did the same but popped two from the front (and back) to see if one was left. Then I pushed the three values (in both directions) and then popped the front and then the popped the back (and vice versa). Each time I popped I had a test case to see if the right parameter was popped. I then tested to see how popped reacted with an empty deque. I saw that None was returned when popped in an empty deque.

I then checked length. I checked with an empty deque to see that the correct value was returned for length. Then I checked to see when I pushed values from either side that the length would acknowledge that a object was added to the deque. I then checked to see if popped also only decreased by one each time it was called on an already constructed deque. Or did not impact the deque length if the deque was empty.

I then checked to see that peek method was working properly with deques. First with an empty deque. I checked to see that the peek from either side returned None, then saw that length was not impacted nor was the string representation of the deque. Then I created deques of length 1 and 2. I checked to see that both sides of peek only returned the value of the front or back and that length was not impacted when called upon nor was the string representation of the deque.

Part 2: Stack Test Cases

I first checked to see that the empty stack has the correct string representation along with a length of 0. I then popped from the empty stack, testing to see that the string representation was not affected by the method and the length as well.

Then I checked to see that when I pushed one parameter to the stack, its string representation was correct and also that the length was incremented correctly. I did the same procedure when I pushed two parameters.

With a stack with one object already in, I checked to see that pop method properly returned the value and took out the object from the list and length reduced accordingly. I did the same with two objects in the stack to see that everything turned out as expected on paper.

I tested to see if peeked method worked properly on an empty stack first. A value of None would have to be returned and no changes to its string representation and length.

I then created a stack with one object in it. I tested to see that when calling peek that front object was returned without any impact on the string representation and the length of the stack. I again ran the test with a stack containing two objects to see that the peek method only returned what the object at the front had and did not impact the length nor the original string representation.

Part 3: Queue Test Cases

I checked to see that enqueue worked properly with an empty queue and checked to see if the length of queue was incremented correctly. I checked to see if dequeuer worked as expected on a empty queue, but returning a value None and not impacting the length of the queue.

With a queue of length one, I checked to see that enqueuer a parameter pushed it to the front and when calling dequeuer that object at the front of the queue was taken out and returned properly with the length of both queues incrementing and decrementing properly. I did the same with the queue of length two and tested to see if the length was returned properly and the string representation of the queue.

Performance of Every Method:

Linked List Deque:

The __str__ method is in linear performance since it must go through a while loop to copy down the linked list into a string. O(n)

The __len__ method is in constant time since I increment the size every time a new node is added in the original file. O(1)

The push_front(and back) method is in constant time since the value is inserted at the same position each time. O(1)

The pop_front method is in constant time since the value that is being removed is always at the same location in the list. O(1)

The peek_front method is in constant time since what is being looked at is always at index 0. O(1)

The pop_back method is in linear time since depending on how large the deque is how long it will take the method to find and remove the value at the end. This is the same as peek_back method but the method does not remove the value. O(n)

Array Deque:

The str, __grow, and push (front and back) are linear performance time since they go through a loop in order to carry out their function calls in their worse case. O(n)

Len, pop (front and back), peek (front and back) are all constant by using the circular array since modulus gives the correct index every time it is called upon. O(1)

Stack:

Str and push is the only one that is linear since it must go into a for loop or while loop in order to be formatted correctly. And push in the worse case must call on grow that will convert it to linear. O(n)

The rest of the methods are done in constant time since objects are added and removed from the stack from only the front. And size is incremented as things are added. O(1)

Queue:

Str and enqueue in the worse case are linear. Str has to go into a loop in order to get formatting correct. Enqueue in its worse case for an array deque must call on grow which takes linear time thus converting it into linear. O(n)

Len and dequeue are done in constant time since they must not go into any loop and are either incremented along the way or removed at the same position every call.

Tower of Hanoi:

In tower of Hanoi I noticed that there was a recursive formula for each incremented number. For the next number of moves it was M(n) = 2 M(n-1) + 1. Using math, I got an explicit formula for the M(n)= 2^n – 1. This gives me a O(2^n) since that is how the increment is worked through on explicitly. This can be seen in the code by using two recursion loops that will ultimately make the performance of Hanoi_rec exponentially .I'm not sure if I had implemented the recursion properly but it worked for 3 and 4 and that was enough to convince me by principal of mathematical induction. O(2^n)