Tests:

First, we tested to make sure that an empty deque would yield the proper result. Then we started with separate front implementations. We tested trying to pop from an empty deque both back and front which should result in None. Then we front and back tested, separately, various pushing, peeking, and popping combinations in the deque to examine the rigor of our code, such as pushing onto a deque at capacity, popping off a deque at capacity, pushing a deque to capacity, and peeking at various locations. Then we combined the calls of back and front and did the same test as before only from both ends, testing the peek, push, and pop functions.

Next, we tested our queue implementations. We first tested to make sure that we could support an empty queue. Then we tested to see the result of dequeueing from an empty queue which should result in None. After that we gave our queue various test of enqueueing and dequeueing functions to test the rigor of it. We dequeued several times and checked both that the string would return correct and that the value we dequeued was correct also.

Finally, for our stack implementations. We first tested to make sure that we could support an empty stack. Then we tested to see the result of popping and peeking from an empty stack which should both yield None. After that we gave our stack various test of pushing, popping, and peeking functions to test its breaking point. We peeked several times and made sure the value we peeked was correct. We also popped several times and then made sure both the value we popped and the string was returned correctly.

Worst Case:

Array Deque:

For __str__, the worst case it is linear time, since we have to go through all the contents in the array.

For __len__, it is always constant time since we store the size in the init__. If we do not store, it can also be constant if we calculate the size with the position of front and back, which only requires simple add and abstract and does not need iteration.

For __grow, it is linear time, because we need to do a sort, which is linear performance.

For push_front, it is divided into two categories. First is when we need to call the __grow: when we call the __grow method it is linear performance; Second is when we do not need to call it: it is constant time since there's no need to walk through.

For push_back, it is the same as push_front, which is categorized in to two: linear for calling __grow and constant for not calling.

For pop_front, pop_back, peek_front, and peek_back, all of these are constant time. We do not and do not need to walk through the deque.

Linked List Deque:

For __str__, it is linear time due to the nature of Linked List.

For __len__, it is constant time. The __init__ creates the list which inherently stores the size, which is equal to the length, of the list.

For push_front, it is constant time. If the list is empty, it calls the append_element method, which is constant time; if the list is not empty, it calls the insert_element_at method, which is constant when index = 0, which is this case.

For push_back, it is constant time because the append_element method that it calls has constant performance.

For pop_front, it is constant time. The remove_element_at method in Linked_List has constant performance when index = 0, which is the position of the "front".

For pop_back, it is linear time. If we want to remove the last element in the Linked_List we have to walk all the way through it, at least according to the Linked_List it has to.

For peek_front, it is constant time. The get_element_at method has constant time when index is 0.

For peek_back, it is linear time. The reason is the same as pop_back, that is, we have to walk from "header" towards the end of the list, in the Linked List we create.

Queue:

For __str__, it is linear time since it is based on Linked list.

For __len__, it is constant time since the length (or size) is already stored when we create the list.

For enqueue, it is constant time. It calls the push_back method in Linked List Deque, which has constant performance.

For dequeue, it is constant time. It calls the pop_front method in linked list deque, which has constant performance.

Stack:

For __str__, it has linear performance since it is also based on Linked list.

For __len__, it has constant time because the length(size) of the list is stored when we create the list in the __init__.

For push, it is constant time. It calls the push_front method in linked_list_deque, which is constant time.

For pop, it is constant time because it is the same as dequeue which calls the the pop_front method in linked list deque, which has constant performance.

For peek, it is constant time. The get_element_at method has constant time when index is 0.

Hanoi:

computed Hanoi(1) in 0.004984000000000016 seconds.

computed Hanoi(2) in 0.014328000000000007 seconds.

computed Hanoi(3) in 0.034316999999999986 seconds.

computed Hanoi(4) in 0.07386200000000001 seconds.

computed Hanoi(5) in 0.15037499999999998 seconds.

computed Hanoi(6) in 0.30404200000000003 seconds.

computed Hanoi(7) in 0.61364 seconds.

computed Hanoi(8) in 1.161279 seconds.

computed Hanoi(9) in 2.348342 seconds.

Clearly the performance of Hanoi is exponential. The time which Hanoi Tower with n disks takes is approximately 2 times of that with n-1 disks. The reason behind this is that, for example, we are going to move 7 disks from source to dest, we first have to move the upper 6 disks to aux, then move the 7th disk to dest, then move the upper 6 ones from aux to dest. Compared to moving 6 disks, moving the 7th is fast which can be ignored (when we only have 2 or 3 disks it cannot be ignored — that's why when we have fewer disks they are not perfectly 2 times of the previous), so basically we move 6 disks from source to aux, then move them from aux to dest, which is exactly two times the time we move 6 disks directly from source to dest.