Test Cases: Each method groups both Linked_List_Deque() and Array_Deque() together for faster testing. The purpose for all the test methods (except for the last one) is to evaluate the functionality of Stack and Queue and ensure that normal functions behave as expected. The last test method goes into the technical methods of push__front/back and pop_front/back of the get_deque (LL or Array deque) and see if the desired output is satisfied, including the str output.

Analysis

Deque: Performance time for Linked_Listed_Deque is O(1) for all methods within the deque, except for the string method (O(n)) due to the Linked_List class concatenation. This is because all variables within the Linked_List class is designed to be very efficient: The pop, push, and peek methods are organized so that only the sentinel and the neighboring nodes are involved in any invocation, therefore giving the entire deque constant time performance, regardless that the Linked_List methods in the class itself is O(n) for insert_element_at and get_element_at. For the pop_back and peek_back methods, the Linked_List algorithm always starts the pointer from the trailer sentinel node, therefore, the pointer does not have to traverse the entire Linked_List to get to the last index.

Array_Deque is more complicated due to the concatenation of it's string and grow methods. Because of the grow method called upon in the Array_Deque, this gives the push methods amortized time from O(1) to O(n) in worst case. This is resulted from when the capacity of the array needs to grow, the size of the array doubles, and then follows by the reordering of existing values with the front being assigned to index 0. Given the worst case where the capacity needs to double, at n^{th} iteration of push that needs to grow, the capacity is given by (2^{n-1}), not including the shifting of the current values. However, in realistic performance, the grow method is only invoked when the capacity reaches its full limit, which is not often, therefore, giving the overall time complexity of O(n). For the string method, the concatenation gives this method a time complexity of O(n), where n is the existing values ignoring the None values as the capacity grows. The peek methods, len(), and the pop methods all have O(1) since self.__size, front pointer combined with modular arithmetic allows instantaneous access to the front and back value location within the array.

Stack: The performance of Stack is resulted directly from the implication of the Deque class:

Stack	**Array_Deque**	**Linked_List_Deque**
str	O(n)	O(n)
len	O(1)	O(1)
push	O(1) amortized to O(n)	O(1)
pop	O(1)	O(1)
peek	O(1)	O(1)

Queue:

Queue	Array_Deque	Linked_List_Deque
str	O(n)	O(n)
len	O(1)	O(1)
enqueue	O(1) amortized to O(n)	O(1)
dequeue	O(1)	O(1)

Tower of Hanoi: The recursion of the Tower of Hanoi can be broken into the base case where the stack would pop when n is 0. Then the premise is to have the recursion move all the top disks to the auxiliary post until there remains one disk left in the source post. Then the disk would move from the source post to the destination post. It follows that the recursion repeats, but this time the auxiliary post becomes the source post and vice versa. The time performance of the recursion is based on the number of iterations(steps) the algorithm goes through to move all the disks from the source post to the destination post. A mathematical formula is given for the number of steps needed to complete this process: 2^{n-1}, where n is the number of disks in the source post. Even with O(1) stack methods, the time complexity relies solely on the recursion itself, which gives the time performance of $O(2^n)$ for worst case scenario. This means that there exists a tremendous time difference when n grows from 10 to 20. Although the algorithm is inefficient, the printout gives step-by-step positions of the disks and how to solve the tower.