Project 3: Queue and the Stacking of the Deque - Writeup

Test Cases

For Deque.py, Stack.py, and for Queue.py I created 4 test case scenarios that of 0, 1, 2, and 3 elements. The test case for zero elements were used to make sure that when pop and peek is called, it returns None. The test cases also makes sure that the string, even though it is empty, is formatted correctly. The test case for one element was pushed to see if the strings were formatted correctly and to also check if the size was incrementing properly. When the pop method was called, the deque emptied correctly. When the peek method was called at the deque, it worked properly by returning the only element in the deque. The test case for two elements made sure that the size was incrementing properly. The test codes checked to see if the strings printed the contents of the deque in the correct order. More importantly, it made sure that when it is popped from the front or back, the correct elements were removed and the size was decremented. With three elements, the test code made sure that the size of the deque was correct after popping and pushing. It also checked that the peeking and popping returned the correct element. It also made sure that the strings printed the elements in the proper order as well.

Worst Case Performance Analysis

Deque:

Array Deque: Similarly, the __str__ method operates in linear time as it iterates through all the values in the array to create a string. The __len__ method operates in constant time as self.__size is simply updated and stored in memory when it is incremented or decremented. Pop and peek run in constant time as there are pointers which stores the location of the front and back. In the worse case scenario, push front and push back operate in linear time because the __grow method is called to increase the capacity.

Linked_List Deque: The __str__method operates in linear time as every value is iterated to create a string. The __len__ method runs in constant time as self.__size is simply updated and stored in memory when it is incremented or decremented. As the header and trailer does not require iteration to access, push front and back, pop front and back, and peek front and back all operate in constant time as they are referencing the nodes next to the header or trailer.

Queue:

In sum, the worse case scenario occurs when the method __grow is called. For instance, when pushing a value to the front or back to a large array deque when the capacity is full, the __grow method is called which runs in linear time. However, the __grow method will never be called if the data structure is a Linked_List since a Linked_List can simply add a value to the front or back and does not have to worry about its capacity. As a Linked_List possesses a header and trailer, adding a value works in constant time. The __str__ method for both the Array deque and the Linked_List deque operates in linear time as it has to iterate through all of the elements in the deque. In both the Array and Linked_List deques, the __len__ method runs in constant time as it simply updates self.__size when its length is added or subtracted. adds or subtracts a value from self.__size.

Stack:

Similar to the Queue, the __str__ method operates in linear time and the __len__ method runs in constant time. The worse case scenario is the same to that of a Queue, when pushing a value onto a full array as it calls the __grow method which runs in linear time as it uses a for loop to get the values from the old array and put it in a new array with a larger capacity. Pop and peek operate in constant time because there are pointers that track their position.

Towers of Hanoi Performance

Hanoi:

The while loop used to construct the deque operates in linear time as every item has to be pushed to the deque. The recursion process for the towers of Hanoi operates in exponential time as the amount of operations needed to solve the towers in respect to n increases exponentially at a rate of $O(2^n)$