Overview:

A deque is a unique data structure, as it has six methods which the user can utilize in interacting with the data. For starters, it has a push_front and a push_back, which allows the user to insert data from the “front” (left side to right) or the “back” (right side to the left). Instead of using a typical array, where you have to insert at the end position, this allows for the quick access to the beginning and end of the array. The next methods which are useful, deal with removing the data using pop_front and pop_back. This allows for quick access in being able to remove from either end, instead of starting at the front position and doing a current walk until you have reached the last index. The final and unique methods which a deque uses, are peek_front and peek_back, which allows the user to see what is at the left-most side, and the right-most side of the data stored. I discuss further in the performance analysis section, regarding this class.

A queue consists of only two major methods, enqueue which is the inserting of a value and dequeue which removes the value. Enqueue inserts a value from the back (right) to the front (left) of a list and dequeue removes the value only at the front. For this reason, it allows for things to move in a more constant stream, as you can only insert and remove items at fixed positions. Additionally, it has similar methods for initialization, the size and stringing values together like the other classes we have looked at. I discuss further in the performance analysis section, regarding this class.

A stack consists of a push method which inserts a value at the top and layers items on top of it. It also has a pop method, which removes the value at the top of the list. It finally has a method called peek, which lets the user interacting with this structure, know the value which is at the top of the stack. Additionally, it has similar methods for initialization, the size and stringing values together like the other classes we have looked at. This stack allows for things to be inserted in constant time, but removed in linear time. I discuss further in the performance analysis section, regarding this class, but next I will be looking at the test cases.

Test Cases:

def test_add_stack_empty(self): This is to check whether or not the push function works in the stack implementation. If so, the string output should be an open bracket, followed by one space, the value inputted, one more single space, and a closing bracket. This also makes sure that the string method prints properly, regarding a single value being inserted.

def test_push_three_pop_two(self): This is to test that the push methods work in conjunction with the pop method, and displays the appropriate result. By doing so, it allows the program to make sure that the push and pop methods are working without any mistakes, and the print statement still can handle the output without having any formatting issues.

def test_pop_three_from_empty_stacks(self): This is to make sure that if the user wishes to pop anything from an empty stack, it will return an index error. This is useful in making sure that the pop implementation works.

def test_add_queue_empty(self): This is to check whether or not the queue implementation works, regarding enqueueing a variable, displaying the correct formatting of an open bracket, one space, variable, one space, and closing bracket. It also is to make sure the enqueue actually works, and displays the variable.

def test_dequeue_three_from_empty_queues(self): This is to make sure that if the user wishes to dequeue from an empty queue, the program raises an error and stops the program from continuing. By doing so, it allows the verification of the dequeueing being implemented properly.

def test_emptyStacks(self): This test case is similar to that of the previous one, except this is to check the stacks, in making sure the proper formatting is displayed when they are empty.

def test_peek_of_stacks(self): This pushes a number of variables onto the stack, and then pops some variables from it, leaving a remaining amount. From this, the peek method is called, to make sure that the appropriate values are returned. By doing so, I am making sure that the push method works, the pop method works, and the peek method also works.

def test_length_of_queues(self): This tests that the queue method works, in conjunction with the counter for the size of the queue. By doing so, it is great for making sure that the queue has proper adding of variables, which also allows the counter to increment per addition. This then, helps make sure that the length is correct and returned for passing the tests.

def test_enqueue_three_dequeue_two(self): This is to test whether or not the queue has proper implementation of being able to enqueue three items in order, and remove two of them, leaving one for the print statement to be seen. By doing so, I am checking whether or not the queue does not run into any issues for adding variables, and can dequeue from the appropriate place and the string output is correctly formatted.

def test_length_of_stacks(self): This tests that the push method works, in conjunction with the counter for the size of the array. This test is great for testing to make sure that there are not any bugs or issues with the push methods of the stack, as well as tests that it increments properly and keeps things in order for the length.

def test_emptyQueues(self): This test case is to check if the queue is empty. If so, the returned string output should be "[ ]" with exactly one space between the two brackets. This makes sure the formatting is proper, regarding how we are to have it spaced for the project.

I will now be taking a look at the performance analysis of each class and their respective methods.

Performance Analysis:

Hanoi.py:

def Hanoi_rec(n, s, a, d) [$O(2^{n-1})$]: Uses recursion style to complete the towers of Hanoi problem. It is passed the number of disks, the three stacks, and then calls itself twice until base-case is met. The best-performance is **$O(2^{n-1})$**, the worst case is **$O(2^n)$**.

def Hanoi(n) [O(n)]: This fills the source stack with one to n-disks, causing it to be **O(l)** to **O(n)** performance.

I solved this Hanoi problem by breaking it down into a recursion style, base-case solution, where the stack pop's when **n** is equal to 0. Because our instructions required us to use recursion, this action takes place until all of the disks on top of the source location are moved onto the auxiliary and destination poles. I then implemented a second case, where the auxiliary pole becomes the source post, and this recursively repeats in similar fashion to the method listed previously. Since there are n disks, and the program recursively uses n-1 twice, the performance is $\mathbf{O(2^{n-1})}$. In closing, when we add additional disks to this problem, using only n poles, the performance will increase (quite literally) exponentially with the worst-case performance being that of $\mathbf{O(2^{n})}$.

Deque:

Linked_List_Deque.py:

def __init__(self): This does not have any performance, as it just initializes the attribute of self__list.

def __str__(self) [O(n)]: Because the string method has to go through the entire list concatenating every value n-times, it is **O(n)** performance. Though has a best case of **O(l)** if there is only one item needed to be strung together.

def __len__(self) [O(l)]: This method only returns the length of the list, which has **O(l)** performance.

def push_front(self, val) [O(l)]: This method only adds a value at the front of the list, which makes it **O(l)** performance.

def pop_front(self) [O(l)]: This method only pops (removes) the value at the front of the list, which makes it **O(l)** performance.

def peek_front(self) [O(l)]: This method only looks at the front of the list, which makes it **O(l)** performance.

def push_back(self, val) [O(l)]: This method only adds a value at the back of the list, which makes it **O(l)** performance.

def pop_back(self) [O(l)]: This method only pops (removes) the value at the back of the list, which makes it **O(l)** performance.

def peek_back(self) [O(l)]: This method only looks at the back of the list, which makes it **O(l)** performance.

Array_Deque.py:

def __init__(self): This only initializes attributes of the class, and therefore has no performance analysis.

def __str__(self) [O(n)]: The string method has a best-case performance of **O(l)** where only one item needs to be strung and printed out. But, if there are two or more values, the performance is amortized to **O(n)** performance.

def __len__(self) [O(l)]: This method returns self.__length, which is incremented and decremented whenever the methods are called, therefore giving it **O(l)** performance.

def __grow(self) [O(n)]: This method is called whenever the array needs additional space, causing the best-case performance to be **O(l)** but amortized to **O(n)** where it needs to be done two or more times.

def push_front(self, val) [O(n)]: This method only adds a value to the front of the list, giving it **O(l)** performance, but if the growth method is invoked, it slows the performance to **O(n)**.

def pop_front(self) [O(l)]: This method pops (removes) the value at the front of the list, giving it **O(l)** performance.

def peek_front(self) [O(l)]: This method only looks at the value stored at the front of the list, giving it **O(l)** performance.

def push_back(self, val) [O(n)]: This method only adds a value to the back of the list, giving it **O(l)** performance, but if the growth method is invoked, it slows the performance to **O(n)**.

def pop_back(self) [O(l)]: This method pops (removes) the value at the back of the list, giving it **O(l)** performance.

def peek_back(self) [O(l)]: This method only looks at the value stored at the back of the list, giving it **O(l)** performance.

Stack.py:

def __init__(self): This method does not have any performance analysis, as it only initializes the attribute self.__dq.

def __str__(self): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(n)**, and if it is the array deque, it will be **O(n)**.

def __len__(self): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(l)**, and if it is the array deque, it will be **O(l)**.

def push(self, val): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(l)**, and if it is the array deque, it will be **O(l)** amortized to **O(n)**.

def pop(self): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(l)**, and if it is the array deque, it will be **O(l)**.

def peek(self): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(l)**, and if it is the array deque, it will be **O(l)**.

Queue.py:

def __init__(self): This method has no performance analysis as it only initializes the deque attribute.

def __str__(self): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(n)**, and if it is the array deque, it will be **O(n)**.

def __len__(self): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(l)**, and if it is the array deque, it will be **O(l)**.

def enqueue(self, val): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(l)**, and if it is the array deque, it will be **O(l)** amortized to **O(n)**.

def dequeue(self): This method has a performance variance based upon which class is called by the **Deque**. If it is the linked list deque, it will be **O(l)**, and if it is the array deque, it will be **O(l)**.

Summary:

When taking a look at each of these classes, it is interesting to see each of their performance pro's and con's. For instance, the string methods stay the same for both the linked list and array deque, as do all of the other methods except when it comes to pushing items. The linked list does pushing of values in constant time, while the array deque goes from constant time to linear time, depending upon the growth method being invoked n-times. The same is true with regards to the queue, as the two classes share similarities in all of the methods, except for the enqueueing of a value. The array deque does this in constant to linear time, but the linked list does this in constant time. Overall, when comparing the two methods, I believe the linked list wins for the best performance of the two.