The __str__ method runs in O(n) for both the Array_Deque and the Linked_List_Deque since the way they both work is by accessing n elements using an O(1) function to retrieve the value at the location.
Both the Array_Deque and the Linked_List_Deque have their __len__ functions running at O(1) time since they just return the .__len attribute of the class.
The grow function of Array_Deque has a complexity of O(n) since the grow function works by setting the .__contents attribute to a new array that is the original .__contents array plus an array of None attributes, which is the size of the original .__contents. Since the time complexity of adding two lists is O(n), this means the .grow function has a time complexity of O(n).
For the Linked_List_Deque, the push_front method runs at a time complexity of O(1), since it is the same as inserting a node at 0, which has a constant run-time. However, for the Array_Deque, the push_front method runs at a complexity of O(n) since at worst case, the grow function is called, and then each element is moved over one to the next index in the array so that the 0^{th} index can be set to the new value, which all runs at a complexity of O(n).
For the Linked_List_Deque, the pop_front method runs at a time complexity of O(1), since it is the same as removing the node at 0, which just a constant run-time for Linked_List. However, for the Array_Deque, the pop_front method runs at a complexity of O(n) since at worst case, each element after the 0^{th} index is moved over one to the previous index, which is done in O(n) time.
Both the Linked_List_Deque and Array_Deque have their peek_front method run in constant time, since accessing the first value of either an array or linked list can be done in constant time.
While Linked_List_Deque has push_back run in constant time since it has the same complexity as the Linked_List append method. Array_Deque has its push_back method run in linear time since in the worst-case scenario, the grow function gets called.
Both data structures have their pop_back method run in O(1) time since removing the last element in either the array or linked_list can be done in constant time.
TheArray_Deque has it's peek_back run in O(1) complexity since the index is just accessed from the array, and the peek_back method in Linked_List_Deque also runs in O(1) since that is the complexity of accessing the last element in my implementation of Linked_List.
In array_deque, I distinguished between an empty deque and a deque with one entry by using the size variable. When size was 0, the first element would be None, but when adding an element, size would change to one, and an if statement that would check for the array starting at None, would change the first element to whatever value the user wanted to set it to.

The reason why the grow method doubles the size of the array instead of just increasing it by one is to make the program more efficient. If we made grow just increase the size of the array by 1, that would mean we'd have to call grow every time we wanted to add an element, which is just inefficient.

To check the Deque's, I used my test cases start as empty Deques. What I did was just add few elements to both sides each structure, and then remove all of the elements using both remove methods, while printing at each step, and checking the value of peek for each data structure. This test covers a check of all the functions in Deque, and if the result of the tests doesn't match how the Deque structure would react, that's when we know that Deque itself does not work. This covers all cases since we start with an empty structure and examine what happens when we add elements to the structure, and then fully remove all the elements.