Constant Time

The functions in the implementation that operate in constant time are append_element, __len__, __iter__, and __next__.

Append_element is an O(n) function since I implemented the function by creating a node called "tempnode" which meant I could insert the tempnode between the trailer node and the one before it. I accessed the nodes before the trailer and the trailer by using O(1) ways (using self.__trailer and .prev) and used .next and and.prev to insert the new node, which are all O(1) methods, which meant that worst case scenario is constant time.

The __len__ function is O(1) since the function just returns self.__len, which can be done in O(1) time. The functions __iter__ and __next are also O(1) time complexity, since all __iter__ does is create a pointer to the node right after the header, which is done in O(1) time, and __next__ just sets the pointer to the next node, which is also done in O(1) time.

Linear Time

The functions which operate in linear time are insert_element_at, remove_element_at, get_element_at, rotate_left, __str__, and __reversed__.

Insert_element_at operates in linear time since the way it works is by creating a pointer to the header node, then using a for loop calling .next to take the pointer to the node before where the index where the function inserts the node, and then inserts the node at that location. Since I used a for-loop to get to the node right before where the insert_element_at function inserts the new element, that means that in a worst-case scenario, the method runs in O(n) time.

The function remove_element_at is also linear since it runs similarly to insert_element_at, but insteads removes the node at the specified index, instead of adding a new node the way insert_element_at does.

The rotate_left function works by storing the value of first actual node in the Linked_List, (the one after the trailer). Next the function uses the iterator to go through the Linked_List, setting each nodes value to the value of the next node, stopping at the trailer node. To correct for the problem of the node before the trailer not having the right value, we set the value of the node before the trailer to the value that was stored at the beginning. Iterating through the Linked_Lists with O(1) operations results in an O(n) complexity, therefore the method operates in linear time.

The __str__ function also operates at linear time since the way that it works is that first a string to store the string representation of the Linked_List is created. Next, the function iterates through the Linked_List, adding each value to the string as it goes through. Since the function iterates through the Linked_Lists while just accessing the value (which is O(1)), that is why the function operates in O(n), or linear, time.

Finally the __reversed__ function works by first creating a new Linked_List. Then starting with a pointer to the node before the trailer node, the function appends to the new Linked_list the value of the node currently being pointed at, and then changing the pointer to point at the node before the pointer until the pointer goes to the beginning of the list. The __reversed__ method returns the new Linked_List. Since this function goes through the entire array, and the appened_element method runs at O(1), that means __reversed__ runs in linear time.