Test Cases:

Under the test environment, we first created an instance called "ll" and assigned it to the Linked_List class. Then, we tested the functionality by invoking all the class methods. The first invocation was printing out the instance and see if the __str__ method and the formatting were correct ("[ ]"). The append_element method was tested by adding the first three elements inside the linked-list, and the print statement was invoked to ensure consistent formatting. We also invoked the insert_element_at(val, len(ll) – 1) to make sure that the value was not added to the end of the linked-list. Afterward, insert_element_at method was used multiple times followed with the corresponding print statements to see if the values were inserted in the correct position. The rotate_left method was tested with its print statement to see if the first element in the list moved to the last index. The get_element_at method was next in the test. To test the __iter__ and __next__ methods, a for loop was used to invoke the output of the values in each node of the linked_list. With this, we could see that the get_element_at method did not unlink the nodes since all the values were printed in order. The remove_element_at method was invoked with its return values. Lastly, the __len__ method was tested.

Testing Exceptions:

The remove_element_at, get_element_at, and insert_element_at methods were tested using the console. The goal was for the console to raise IndexError when invalid inputs were entered. Index inputs used for these methods were either larger than the size of the linked-list and negative.

Time Performance Analysis:

append_element, O(1): Since this method alters the bi-direction arrows so that the new node points towards the previous last node and the trailer, the time performance is constant. Here, the order in which the arrows follow is crucial: First, the left arrow(.prev) must point to the left arrow of the trailer. Then, have the left arrow of trailer points to the newest node. Finally, have the newest node right(next) arrow point to the trailer.

insert_element_at, O(n): In this method, a check of index was placed to decide which sentinel node should the current alias be pointing to (header or trailer). This was to improve the efficiency in case one index was closer to either sentinel nodes. In the case where the linked-list has even size, current alias will point to the header. The insertion time is linear in the worst case when the item is in the middle of the list; it takes n/2 steps to get to the element.

get_element_at, O(n): Requested index is also put into check of whether the index is closer to the header or the trailer. This is two improve on access time. However, the performance is still O(n) since the current alias traverses based on the position of the index.

remove_element_at, O(n): Similar to the insert_element_at method, the removal of the node requires a specific order of reconnection of bi-directional arrows, but still has to travel to the element. Performance is still O(n).

rotate_left, O(1): In this method, the index of exactly three nodes were shifted: 0, 1, and the last index.

__len__, O(1): Size of the list is already stored in the __init__ constructor. __len__ only returns the self._size, therefore, its time performance is constant.

__str__, O(n): __str__ performance is affected by the size of the linked-list. Every time this method is invoked, the interpreter goes through each element of the linked-list, appends each value into the python array(list) as a string attribute, removes unnecessary characters, and then returns the list in proper format.

__iter__ and **__next__, O(1):** Traversing the entire list in a linear time operation. However, one call of the iter and next methods are only constant time as we are only moving one element.

Josephus, O(n): Since every element but one is removed from the list, the run time of the method is dependent on the length, thus linear time. To make sure that the algorithm works, the result computed is compared to the mathematical proof of the problem. Since the method eliminate every other person, a pattern can be derived to find out the winning position. Ultimately, this position can be calculated via the formula:

$$P(n) = 2^m + l => 2(l) + 1$$

Where l is the remainder that satisfies the sum of the maximum value can represented by the powers of 2's:

$$\text{Ex. } P(21) = 2^4 + 5 => 2(5) + 1 = 11$$

With this formula, the results from the test can be compared and checks.