Methods with Constant Time Performance O(1):

1. class __Node, __init__(self, val): this init method just assigns the value of the Node and sets the Node's next and prev to None, so it has O(1) performance.
2. class Linked_List, __init__(self): this init method initializes a linked list and set its size, header, and trailer, so it has O(1) performance.
3. __len__(self): this method only returns the size of the linked list, which is an private attribute of it and can be accessed in constant time, so it has O(1) performance.
4. append_element(self,val): the method initiates a new Node and links it between the trailer and the last node (not trailer), and increases the size of list by 1. Its performance is not affected by the size of the linked list and is always constant as the trailer can be accessed directly, so its performance is O(1).
5. rotate_left(self): this method operates in constant time because it merely makes the header and trailer re-link to two "new" Nodes, which doesn't require to loop through the entire list. Therefore, the performance has a constant performance O(1).
6. __iter__(self): this method merely assigns the self._header to a variable called self.current which is used later in __next__(self), and return self. It has O(1) performance.
7. __next__(self): this method reassigns self.current to its .next node and return its value. When self.current reaches the trailer, the method terminates. This is a one-step method so it has O(1) performance.

Methods with Linear Time Performance O(n):

1. insert_element_at(self, val, index): this method runs in linear time because it needs to visit every node (either from the start or the end) until it reaches the index we want. On each node, it performs a constant number of steps. In the worst case, the function needs to go through n/2 nodes until it reaches that index, so it runs in O(n).
2. remove_element_at(self, index): just like the insert method, this remove method needs to visit every node in the linked-list (either from the start or end) until it reaches the index we want. On each node, it performs a constant number of steps. In the worst case it also needs to go through n/2 nodes, so it has O(n) performance.
3. get_element_at(self, index): similar to the insert & remove methods, this method visits every node in the linked-list (either from the start or end) until it reaches the index we want and return the value of the node on that index, so in the worst case it also needs to go through n/2 nodes, which proves to perform in O(n).
4. __str__(self): to print the value of every node in the linked-list, this function uses an iteration to get every element's value inside the linked list and append the value to a string, which is finally returned. The methods go through n nodes, and each node takes same steps, so the performance is O(n).
5. __reversed__(self): the function creates a new linked list, go through every element in the self linked list from end to start, append the element into the new linked list, and return the new linked list. Because it goes through every element in the self linked list, its performance is O(n) because it is linearly dependent on the scale of the self linked list.

There is no method with quadratic time performance.