Question 1

First, __init__ runs in constant time because it only initializes the five private attributes for the LinkedList objects. Next, __len__ operates in constant time because the method only has one step regardless of the length of the list: to access the integer stored in self.__size. Self.__size is incremented and decremented within the other methods that alter the length of the linked list object, so __len__ does not need to walk over the whole length of the list to get its size. The method append_element also runs in constant time because the previous pointers in the list allow it to start at the trailer node and insert the elements without needing to walk over the whole list. Next, the rotate_left method runs in constant time because with the list being sentineled and doubly-linked, you can move the node at the head of the list to the tail position by just moving arrows and not needing to walk over the length of the list. The __iter__ method also runs in constant time because all it does is initialize the iterator at the start of the linked list. Finally, __next__ runs in constant time because it only fetches the next value, moves the iter object, and returns the value. However, inside of a for loop, __next__ is called a linear amount of times.

The first method that runs in linear time is the private __walk_to method because it has to walk over the list to the given index before it can return the node at that index. However, this method can start walking from either the header or trailer node, so it technically operates in n/2 time because the farthest it will ever need to traverse is halfway through the list. The methods insert_element_at, remove_element_at, and get_element_at also run in linear time because they all utilize the __walk_to method in their implementations. Insert_element_at must walk to the node that is in front of the desired insertion point before it can move the proper arrows to insert the new node, so its call __walk_to makes insert_element_at run in linear time. The same situation occurs in remove_element_at, which needs to call __walk_to in order to walk to the specified node before it can remove and return its value. Get_element_at runs in linear time because it uses a call to __walk_to in order to walk to the specified node and return its value. Next, my __str__ method runs in linear time because I first created a list and then appended each node value to the end of the list with a while loop. By using append, the while loop runs in linear time. Then, outside of the loop I used the join method and two string concatenations, so the whole method is able to run in linear time. Finally, __reversed__ runs in linear time because it uses the previous attributes to walk backwards over the list, appending each node's value to a new linked list in reverse order. Since append_element runs in constant time, the loop used to walk backwards over this list runs in linear time.

None of the methods in my implementation run in quadratic time.

Question 2

Overall, the two models are fairly similar as they use the same sentineled, doubly-linked list base, so they both have next and previous attributes, and header and trailer nodes. However, compared with the positional model, our Linked List has a simpler implementation. In addition to all of the parameters of the linked list, the positional model also has to keep track of the positional objects for every node. Along with taking up more space in memory, the positional objects also create more possible error cases that the methods have to test for. For example, methods that take a position as an argument have to verify that the position exists and that it points to a node that is still inside the list. The positional objects can also create some confusion for the user in terms of keeping track of what positions correspond to which nodes or values and which methods return a position versus a value. However, as long as you keep track of the positional instances in the linked list, the positional model has better performance times than our Linked List. The insert_element_at, remove_element_at, and get_element_at mutator methods of our Linked List class all run in linear time, whereas all of the mutator methods in the positional model run in constant time. This is because our Linked List model has to walk over the list in order to reach a specified node while the positional model bypasses this walking by having a reference to each node with its positional objects. That way, the model can just jump to the specified node and complete the proper function and not need to traverse through the rest of the list. So, the positional model has a trade off of having better performance times at the cost of having a more complex and memory-heavy implementation.

Question 3

First, many errors or exceptions occur when a list is empty or has a length of one, so I made sure to test all of my methods on an empty list and then on a list with one node in order to verify that they performed correctly or raised the correct error. For example, insert_element_at should raise an IndexError when used on an empty list. In the same vein, there may be special cases or mistakes that occur when doing things at the head or tail positions of a list, so I performed all of my index-based methods at the head and tail positions to make sure they raised the proper error or performed correctly. Next, I inputted negative indices into the methods that take an index as an argument to check that they would correctly raise an IndexError. I also tested the length of the list after each method call to ensure that the self.__size attribute was being changed in methods that alter the length of the list and not being changed in methods that don't change the size of the list. While testing the length, I checked that only methods that are supposed to change the list were changing the list. For example, __reversed__ should create a new linked list and leave the original list unchanged, and get_element_at should only access a node's value, not mutate the list. Also, I checked to make sure that when an error was raised the list wasn't changed. Next, to check that get_element_at and remove_element_at were correctly returning the value of the node and not the node itself, I used both methods on nodes with integer values and then performed a simple arithmetic operation on the returned value. Then, I tested all the methods on a list with several items in it to ensure that they ran correctly under normal circumstances. In particular, I tested insert_element_at, remove_element_at, and get_element_at using indexes at the front and back of the list to check that my __walk_to method works correctly when it starts from the header node and from the trailer node. Next, I tested the iterator by using a for loop on an empty list, a reversed list, a list with several nodes, and a list with one node inside. Finally, I inserted non-integers as values and tested the methods on them to ensure that this wouldn't cause any unexpected errors.