The performance of the length function is constant time because it is returning a value that is always stored in memory and is constantly updated in every other method.

The performance of the append element function is constant time. It is constant because the linked list is doubly linked, because the trailer node is always accessible in constant time. When append element is called, the new node is attached right next to the sentineled trailer node which will always take the same amount of time no matter how long the list is.

The performance of the insert element function is linear time. This function starts by creating a new node from the __init__ function and then locating the node at the index the user wants to insert at. While iterating through the list n number of times the performance is 0(n/2). The worst-case scenario is if the index the user wants to insert at is at middle of the list. In this case the function has to iterate through half the length of the list. In my function I used floor division to check if the desired index was near the beginning or the end of the list. Since the list is doubly linked the function can iterate either backwards or forwards depending on which half the index is on. The insert element function can't insert at the end of the list but is otherwise able to access every other positive index that is less than the length of the list.

The performance of the remove element function is similar to the insert element function and operates in linear time. For every instance of the function it has to iterate over at most half the size of the list. This makes the performance O(n/2) because the list is doubly linked. Remove element is able to access any index in the list and will de link the node at that index unless the index is out of range. The index will be out of range if it is longer than the length of the list or a negative value.

The performance of get_element_at is linear time which is O(n/2). The function will first check if the index is within the range of the list and then iterate through the list for n number of

times. The linked list is doubly linked so it can iterate either forwards or backwards depending on which half of the list it is on. At the worst case the function will have to iterate over half the length of the list.

The rotate left function is able to run in constant time. It can run in constant time which is 0(1) because the function has the same instructions no matter how long the function is. The function starts by pulling out the node at the first index and storing it with the variable holder. Then the function uses code similar to the append method to reattach the node to the end of the list. This effectively shifts the entire list to the left. Since the list is doubly linked, both the beginning and end of the list can be accessed in constant time.

The string method is called every time the variable for the list is printed or converted to a string. When this happens, the method begins by printing out every value in the list. Because of the current walk that has to occur, the function is linear and is O(n). When the length of the list is 0, the function is constant time because there is an if statement that accounts for this. In every other case, the function needs to walk through and append every value in the list to the string that will be returned in the end.

Both the __iter__ and __next__ functions are in constant time and is O(1). Iter is called every time a for loop is created in the main program. Iter keeps track of the current node in the for loop so it does not have to search for the node at the next index every time it is called. __Next__ works by shifting the pointer which is storing the value of the current node in the for loop. When the pointer reaches the end of the linked list, it will point to the value None and cause the iteration of the for loop to end. This means that while the for loop works in linear time, the actual methods of __iter__ and __next__ work in constant time.

To solve the Josephus problem, I made us of the rotate left and remove element methods to calculate the survivor. This function works by using a while loop that runs until there is one number left in the list. For every pass of the loop, the function rotates the list one time to the left. It then removes the element at index 0 which simulates the first person in the group killing the second person. At the end of the loop, it prints out the remaining survivor using the __str__ method and slicing to format the result.

For my testing I focused on the user giving valid and invalid indexes. I was able to test the exception handling as well as the __str__ method while I was testing the methods. To start, I created a linked list from 1-10 to test the append element method. This also created the list that I would use to test the rest of the methods.

The second method I tested was the insert element method. For the first test case I tried to insert a 2 at the end of the list. This should fail because only the append function is allowed to insert and the end of lists. When the IndexError was raised in the method the except block caught this and printed that the index was out of range and that the append method has to be used for adding to the end of a list. The second and third test cases I used were with valid indexes. The try block tried to insert the element and printed out the list and length afterwards. If this was to fail, the except block would catch the error and print that the index was out of range. The fourth test case was meant to test if the function was able to insert at the head or beginning of the list. This should work without any errors and print out the list and length when it is done with the try block. The fifth test case I used was inserting at a negative index. If the exception was caught the except block should print out the list and length but not modify anything in the list.

The third method I tested was the remove element function. The first test case was meant to try a removal at a valid index of 6. The second test case is meant to test for an index out of

range and larger than the length of the list. This should raise an error that will be caught by the except block. The last test case is meant to try to remove an element at an invalid negative index. In both of the invalid index cases, the list shouldn't be modified, and the length of the list should remain the same.

The next method I tested was get element. The first two test cases are meant to test for invalid indexes both larger than the length and negative cases. They should print out "correctly caught error" if the method works correctly and raises an error. The third test cases checks if the function works for valid indexes and in this case it prints out the element at index 4.

The last method I tested was rotate left by placing it inside of a for loop with a range of 6. This try block should work with no problem and tests the case where the list doesn't fully wrap around. The next case I used tested for rotating left 20 times which would make the list wrap around itself. The last methods I tested were the __iter__ and __next__ functions. I did this by iterating over the list and printing out every value for every pass of the loop to check if the iteration was working correctly.

The main difference between the textbooks implementation and my implementation is that the textbook code trades storage space for speed. My implementation works by iterating through the list every time a certain node at an index needs to be accessed. When the get element at, remove element at, or insert element at, my function runs in linear time because it has to iterate through the list with a time of O(n/2). At the worst case, it will have to go through half the length of the list. However, this implementation is space efficient because it only stores the first and last nodes and relies on .next and .prev to go through the list. The textbooks implementation is much more efficient and perform every method in constant time. This is because it keeps track of every single node at every index of the list. While this takes up more space, the functions can

immediately jump to the correct node that needs to be accessed. After that, the nodes around the current node can be accessed very quickly and in constant time. The functions can perform the same number of actions no matter where the node is in the list because no iteration is needed to run through the list and find the correct index. The textbook implementation is better because with very large data sets the methods can hop around to the nodes they need and won't waste time running through the same parts of the list every time. This means the performance benefits outweigh the storage costs to store the individual nodes and their attributes.