Our test program does the following:

1) Test append by appending five values to an empty list.

The new list is then printed along with a message stating the current size of the list.

These print calls simultaneously test \_\_str\_\_, \_\_len\_\_, and whether our append method is incrementing the size of the list correctly.

2) Test the insert\_element\_at function in the two places it should work: the beginning of the list, and in the middle of the list.

The "follow-up" print tests are the same, and will remain the same for the remainder of the test program to ensure each function increments and prints as it should

3) Test the insert\_element\_at function with invalid indices; first a too-large index, then a negative index. We use a try/catch block to except these errors correctly.

We achieve our expected result, printing an appropriate error message, and printing the list once again to confirm that nothing has changed.

4) Tests 2 and 3 are ran again in the exact same way, but with the remove\_element\_at function instead.

Previously we had a specific issue with removing the final item in our list, so we run one additional test on that specific index to confirm it is working properly.

5) Tests 2 and 3 are repeated, but with the get\_element\_at function instead.

6) Test the rotate\_left function by printing the current list, and the list that results after the function call.

7) Test the iterator by using a for loop that prints the values stored in each node line by line.

8) After running all of the previous tests, we run each of the functions one more time to make sure there isn't any faulty code causing any sort of unexpected problems.

(We had an issue with a certain function causing a different function to work improperly after running the first time, which prompted us to add this step)

Performance characteristics:

append\_element() has constant-time performance. It only does a few different computations (reassigning .\_\_next's and .\_\_prev's), and doesn't use loops or care about the size of the list.

insert\_element\_at() has linear performance, because it uses a loop to iterate through the list until the provided index is reached.

remove\_element\_at() and get\_element\_at() have the same characteristics as insert\_element\_at(), because the core method used in all 3 functions are the same: we take an index value and use a loop to iterate through our list until we arrive at said index. And when they are at their best case, which is index = 0, they have constant performance because it is a special case which no need to go into the for loop.

The difference between the performance of linear-performance methods is the coefficient. The coefficient of insert\_element\_at() is 1, because in the worst case the loop goes through the whole list; the coefficients of remove\_element\_at() and get\_element\_at() are 1/2, because we divided the list into two parts: the first half we use .\_\_next and the second half with .\_\_prev, so in the worst case the loop only goes through half of the whole list.

The Josephus function has linear performance. Each time we delete the first element (or kill one human) it is constant time: the rotate\_left() is constant time since we simply move the first node to the last position, which does not go through the list at all, and the remove\_element\_at() is constant when index = 0 as we discussed before. In total we have n-1 items to delete (or n-1 human beings to be killed), so the loop should perform n-1 times, which clearly depend on n. Therefore the performance of the function is linear.