Worst-case performance analysis for every method in Linked List class:

__init__():

First, we have two __init__ methods, one for the __Node class and one for our Linked_List class. Both of these consist only of initializing attributes, so they have no conditionals or loops that would affect performance. Since the performance is always the same, there is no worst case. Additionally, this means that both methods operate in O(1), or constant time.

__len__():

The __len__ method returns the size attribute stored in the Linked_List class. Since this value is stored, no counting is required, so performance is the same regardless of the size of the linked list. This means that once again, there is no worst case and the method operates in constant time.

append_element():

The append_element method also operates in O(1) time. Since we have a doubly linked list and a trailer node, instead of iterating through the list to reach the end, we only need to use self.__trailer.previous. This means that regardless of the size, appending an element at the end will always have the same performance, as assigning and reassigning .next and .previous node attributes is always done in constant time. Similarly, the size must be incremented by one, but since our Linked_List class has a stored size attribute, we can add to it in constant time as well. Once again, we have no worst case.

__get_node():

Next we have the private __get_node method that I added to use for inserting, removing, and getting an element in order to avoid repeating code. The performance of this method is linear time. It takes an index and current walks through the list to reach the desired index, meaning that time increases with the length of the list. We do have a conditional that checks for an IndexError, but this is not the worst case. This relies on checking the size of the list, which happens in constant time, and then raising an error exits the method. Because of this, the conditional cannot be a part of the worst case.

Since we have a doubly linked list with both a header and trailer, we can current walk from either the head or the tail, depending on the location of the given index. If the index is in the first half of the list, the current walk will be from the head, and likewise, if it is in the second half of the list, the current walk will be from the tail. This means that the performance is O(n), but better than the current walk starting at the head every time. The worst case is when the index is the direct middle of the list for an odd sized list, or either side of the direct middle if the list has an even size.

insert_element_at():

Since the insert_element_at method relies on __get_node, that means we know it has at best O(n) performance. The rest of the method assigns and reassigns the necessary .next and .previous attributes of the nodes in the list, all which can be done in constant time. Additionally, size is incremented by one, which can also be done in constant time. Since the rest of the method can be done in constant time, that means that the performance of this method is O(n). The worst case of this method is the same as the worst case for __get_node.

remove_element_at():

The remove_element_at method is very similar to the inser_element_at method. remove_element_at also relies on the __get_node method and the rest of its operations, reassigning .next and .previous attributes and decrementing size, can be done in constant time. It also returns the value of the removed element, but since we have assigned the node to cur and the node has a stored value attribute, this return can also be done in constant time. Once again, since the rest of the method can be done in constant time, the performance of this method is O(n) and its worst case is the case as for __get_node.

get_element_at():

The get_element_at method is the exact same as the remove_element_at method, except it does not need to reassign any .next or .previous attributes or decrement the size. This means that it has the same performance, O(n), and the same worst case.

rotate_left():

The performance for rotate_left is O(1). This is because we do not need to shift every node over by 1 and instead, only need to move the current head node to the tail position. This consists of reassigning .next and .previous attributes at the head and then at the tail. Once again, this can be done in constant time. The conditional can also be done in constant time, as it is checking the size of the list and returns, or exits the function, if the condition is met. Since the method can be done in constant time whether the condition is met or not, that means that the overall performance is also constant time. Technically, if the condition is not met and the attributes need to be reassigned, it would take slightly longer than if the condition was met and the function was exited, so the worst case is if the size is two or greater.

__iter__():

Our __iter__ method has a performance of O(1). It initializes the iter index and assigns cur to be the head, both which can be done in constant time. It then returns the Linked_List class, something that can also be done in constant time. All of these actions will always be the same, so there is no worst case.

__next__():

Similar to the __iter__ method, the next method is also O(1). Assigning val, incrementing the iter index, setting cur to cur.next, and returing val are all done in constant time regardless of the size of the list. The condition can also be done in constant time, as it is once again checking the size attribute of the list. Here it is comparing it to the iter_index and if they are the same size, it raises a StopIteration. All of this can be done in constant time, so the overall performance is also constant time. Similar to the rotate_left method, the conditional would technically take slightly less time if the condition is met, so the worst case would be if the conditional is not met.

__str__():

Lastly, we have the __str__ method. This method relies on __iter__ and __next__, as it has a for loop within it. Despite both methods being constant time, the __str__ method is in linear time, as the loop must iterate through the entire linked list. If the conditional at the beginning is met, then the method can be done in constant time, but if not, then the for loop causes it to become linear. This means the worst case is if the size is not 0, meaning the conditional is not met. Assigning string and cur can both be done in constant time and reassigning cur to cur.next, assigning value to cur.value, and adding the next value to the string can all also be done in constant time. Similarly, the return statement can be done in constant time, as a string does not have to be iterated through to get to a specific index, unlike with our linked list.

Formal worst-case performance analysis of Josephus problem:

In my solution to the Josephus problem, I start out with a while loop that rotates the list to the left, removes the element at index zero, and then prints the remaining list. In this case, rotating left can be done in constant time, as that was also the case above when examining rotate_left's performance. Although remove_element_at's performance was not constant time, in this case it is, as the element is always being removed from position zero, regardless of the length of the list. On the other hand, printing the rest of the list calls the __str__ method, which has a performance of O(n). This means that the interior of the while loop has a performance of O(n) as well. Since the while loop has to repeat for each element in the list, that means that it increases the performance to O(n^2), as the longer the list, the more times the while loop runs, and within the while loop, the more iterations the __next__ method needs to be run within the __str__ method's for loop.

At the end, similar to the remove_element_at situation, get_element_at can be done in constant time, as it only returns the first (and only) element in the list every time.

For the main section of Josephus.py, I create an instance of the Linked_List class called 'll', which can be done in constant time, as the __init__ methods within

the Linked_List class and nested __Node class both has O(1) performance. Then, the for loop iterates through the natural numbers and stops at the number that the user input. Within the for loop, I append each element, each containing one of the values that the for loop iterates through, to the end of the list. Since append_element can be done in constant time, that means that the for loop has a performance of O(n), as this must be done once for each number in the specified range. Lastly, printing the initial order also has O(n) performance, as it calls the __str__ method again just like printing the remaining list after each iteration within the Josephus function.

Detailed explanation of approach to testing of linked list:

In my testing, I wanted to focus first on empty lists, then lists with one element, and lastly on list with many elements. This is because some of the methods have special cases or raise IndexErrors, so there would be different outcomes depending on the size of the list. For example, insert_element_at cannot be used on an empty list, as there are no valid indices, but it can be used on a list with one element, as long as the specified index is zero.

With an empty list, I first tested the __str__ and __len__ methods to make sure they said '[ ]' and '0', respectively, by printing them. This ensures that the size of the list starts at zero when the list is initialized and that the __str__ method does not have any errors when there are no elements in the list. I then verified that insert_element_at, remove_element_at, and get_element_at all raised errors using exceptions, as none of them should be able to alter an empty list. I did this by trying to insert, remove, and get elements at indices 0 and 1, and I had the list printed in between to ensure that there were no alterations. Since none of these methods are able to work when there are no valid indices (as the list is empty), this is an important step, as it verifies that this is the case. Next, I called rotate_left() to make sure that it also does not alter the list, as moving the header and trailer or any of their attributes would be detrimental to the list. Since their values point to None, I made sure that rotate_left did nothing by printing the contents of the list again. Since None was not one of the contents, that means that rotate_left correctly did nothing. Lastly for the empty list, I checked __iter__ and __next__ using a for loop where I printed each value in the list. Since the list is empty, nothing should be printed, but there should also be no errors. Since all of this worked as expected, the only thing left to do was check append_element, which sets us up for a list with one element. This is an important method, as it is the only one that can alter an empty list, so without it, we would not be able to create a list with elements. Since the first thing I check with a list with one element are the __str__ and __len__ methods, that will verify that append_element worked correctly as well, as the list will have one element and the size will have been incremented by one.

Next, with a list with one element, as with an empty list, I checked the __str__ and __len__ methods first. This step was the same as before, but this time they should print '[1]' and '1', respectively. Since this was the case, that means that both methods work and it also verifies that append_element was also correctly

implemented for the empty list. I then removed the element at index zero to ensure that remove_element_at works for a list with one element. After appending an element again to return the size to one, I implement two tries for each of insert_element_at, remove_element_at, and get_element_at. There should be one that works and one that raises an error for each. This ensures that when the index is valid (0 in this case), the operation is correctly carried out, and when the index is invalid, an IndexError is raised. I also print the current elements and size of the linked list as well as the values returned from remove_element_at and get_element_at to verify that the methods did actually do what was expected when the index was valid. Additionally, those that raised an error should not have altered the list. Since everything went according to plan here, we know that for empty list and lists with one element, the methods that have been tested so far function correctly. Next, I checked rotate_left to make sure that it does nothing again. By printing the contents of the list, I can verify that there is only one element, and there is no occurrence of None. Since '[1]' was correctly printed, I know that rotate_left also functions properly. Lastly, I checked __iter__ and __next__ using a for loop again. This time, it should call next twice, once to return the lone value of '1' in the linked list and a second time to terminate the iteration. This would result in '1' being printed, which ended up working correctly.

At this point, we know that all of the methods can be reliably used for an empty list or a list with one element, as they will either catch an out of index input with an IndexError or perform their intended function. That leaves us with a list with multiple elements to check.

First, I appended two more elements to the end of the linked list, bringing the size to three. I then checked __str__ and __len__ once again using print and they performed as intended. Additionally, they verified that append_element worked for both lists with one element and lists with multiple elements. I then repeated the same steps with insert_element_at, remove_element_at, and get_element_at as I did previously with the single element list, except I did not have the extra step of removing and then appending an element at the beginning. This means I had two instances of each method, where one performs their function on a valid index and the other is given an invalid index to raise an IndexError on. I also printed the return values as well as the current contents and size of the list again. Like before, all three performed correctly so I could now move on to rotate_left. Before with rotate_left, both sizes zero and one were special cases, so I could now check if the list can actually rotate as intended. I printed the result afterwards and, as expected, the head value was moved to the tail. Lastly, I implemented one more for loop to check __iter__ and __next__. This time, since the size is three, __next__ should be called four times: three times to return a value from the list and one time to stop the iteration. Since this also had the correct outcome, that means that all of the methods can be reliably used for lists of any size.

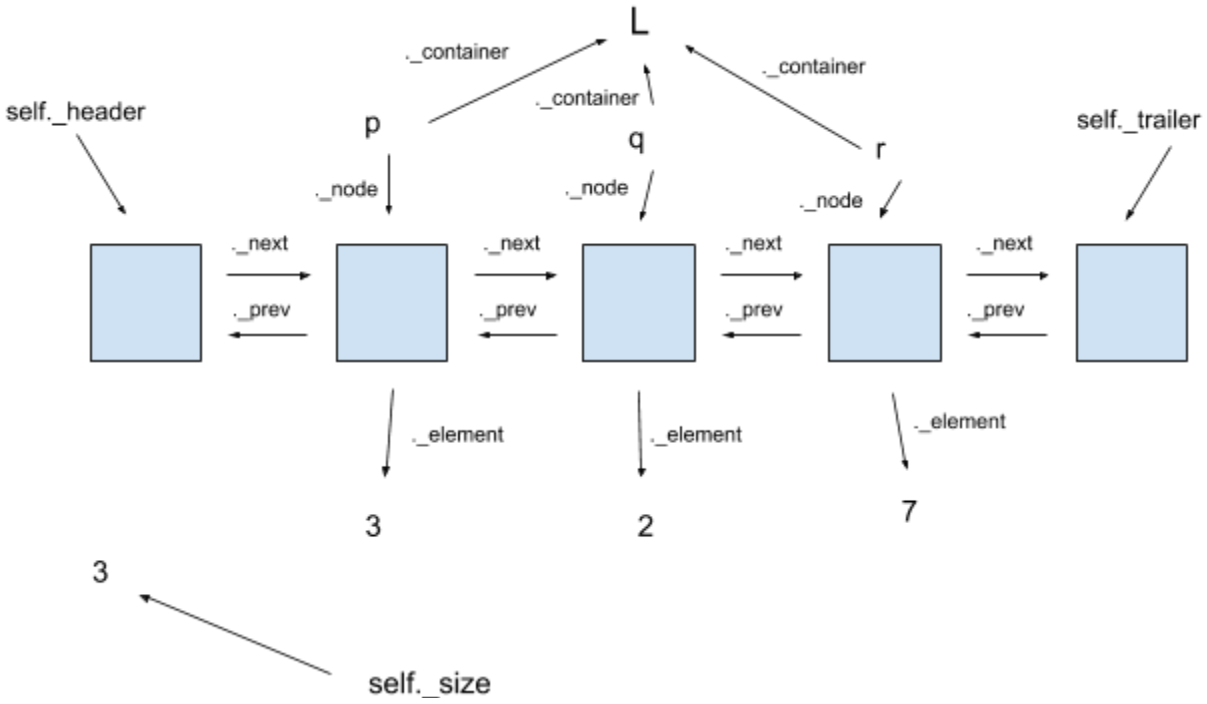
Thorough comparison of our approach and 7.4's approach:

First, in books approach, they rely on the _DoublyLinkedBase class they they created previously, which has a similar _Node class to us with a ._element (in my case .value) and a ._prev (mine is .previous) and ._next (.next). Their attributes and class are both private, however our attributes are public, but can only be used inside the Linked_List class, since the nested _Node class is private. The _DoublyLinkedBase's __init__ function is also the same, initializing a header and trailer, setting the header's next attribute to be the trailer and the trailer's previous attribute to be the header, and setting the size to 0. It also has _insert_between which works the same as our insert_element_at, except it cannot be used to place an element at position 0. _DoublyLinkedBase also has the _delete_node method, which works similarly to our remove_element_at. Where it does differ is their removal of the node whose value is being returned. For our implementation, since nothing is referencing our returned node, we let Python's garbage collection take care of the node for us. Their implementation does have an additional method, is_empty, which checks whether the positional list is empty. So far, the performance and ease of implementation for the two classes are very similar. Our insert_element_at and remove_element_at methods do slightly outshine their _insert_between and _delete_node, however, as insert_element_at has slightly more functionality and remove_element_at requires fewer steps and has a slightly faster performance.

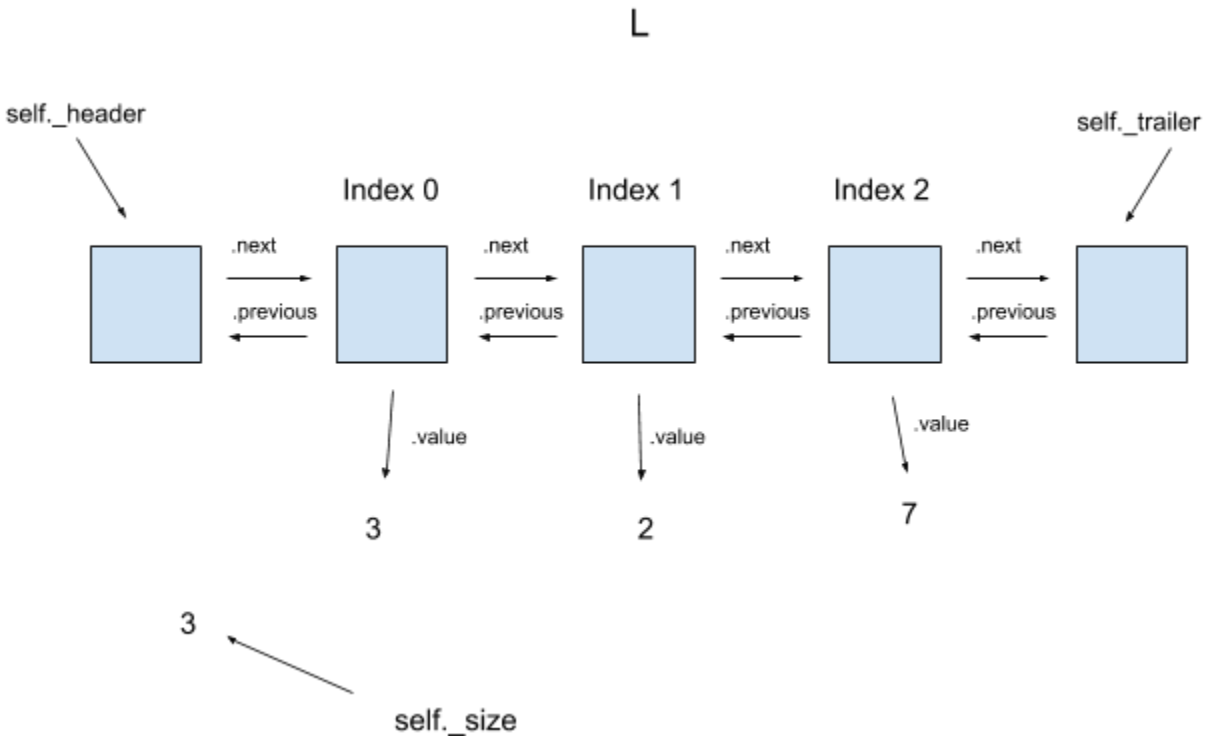
Moving on, the book uses a Position class within the PositionalList that references each of the nodes, where altering the position reference of one node does not affect the other references. On the other hand, we use indexing to reference our nodes. If we add or remove an element, it affects the indices of the following nodes. This means that it may be easier to track the location of an element within the book's implementation than in our implementation. The book's implementation also has more convenient features, such a O(1) performance of adding, removing, or checking values within the list. This is because they can use the position references to the nodes to avoid needing to current walk to get to the desired index, which has O(n) performance like in our implementation. This does come at a cost, however, as there are extra steps that must be made, such as creating the position class for each node and validating positions to ensure that they are actually part of the positional list. Additionally, there is more data that Python needs to keep track of, so their implementation takes up more storage. The book is able to have more freedom and flexibility, as they can not only return, add, or remove elements of positions in constant time, but also the elements before or after a certain position or at the beginning or end of the list. Lastly, there are some methods that are not mutually shared between the two implementations. The book has more accessors, such as first, last, before, and after, whereas we have get_element_at, which works for any index. In terms of mutators, we have a rotate_left method and the book has add_first, add_before, add_after, and replace. Besides the book having easy access to elements before or after a certain position, only preference dictates whether or not these methods are included in either implementation. Overall, our approach is easier to implement, as there are fewer steps and variables to keep track of, but at the cost of having a worse performance

and fewer accessors. Below are diagrams that may help visualize the differences between the two approaches.

PositionalList (Book's Approach)



Linked_List (Our Approach)



Note: although I have listed indices 0, 1, and 2 for our approach, we cannot use L[0] to access the value 3, but instead need to pass the index 0 into a method, such as get_element_at(0).

Citation:

Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. 2013. *Data Structures and Algorithms in Python(1st. ed.).* Wiley Publishing.