Performance analysis of the *Linked_List Class* methods:

__init__(self, val) (Node)

The worst-case performance of the *__init__(self, val)* method is constant time *O(1)*. To construct a Node object, what computer does is setting out a box space in memory and binds three attributes together: a *val* as the element of the node, a reference to the *prev* initialized to be None, and a reference to the *next* initialized to be None. All of the work is constant time as they involve constant steps.

__init__(self) (linked_list)

The worst-case performance of the *__init__()* method is constant time *O(1)*. The work done when constructing a linked list is: constructing two nodes with elements None as *trailer* and *header,* linked them together by setting the next of *header* be *trailer* and the prev of *trailer* be *header,* and initialize the size to be 0. All of them can be done in constant time. Constructing a node is constant time as is specified above; setting the node to point to each other is constant time because changing the reference takes constant steps in a computer; and initialize an attribute *size* to be 0 is also constant time as setting out a box in the memory and give it initial value.

__len__(self)

The worst-case performance of the *__len__(self)* method is constant time *O(1)*. Since *self.__size* is an instance variable kept track of in every change of the linked list by adding one more step in the change, it only takes constant time in this specific method to look and return the instance variable that is kept being updated throughout the class.

append_element(self,val)

The worst-case performance of the *append_element(self,val)* method is constant time *O(1)*. There's no loop executed in this method, and every step in it costs constant time, making the total performance of the method a constant function *O(1)*. This method does the same amount of work, only involving constructing a new node and link it to the *trailer* and the current *tail* node, both of which can be done in constant time in a *Linked_List* class, no matter how long the linked list is.

insert_element_at(self,val,index)

The worst-case performance of the *insert_element_at(self,val,index)* method is linear time *O(n)*. In the worst case, the index in front of which to insert is the last position in the linked list, which requires the *for* loop to go through the whole linked list starting from the first node after the header to the tail node, making it a function of *n*; and inside each *for loop* the work is constant time, only involving advancing the *index_node* to its next. Therefore, the cost of this loop is a function of the scale of data, i.e., the length of the linked list. And the total performance is a function of *n* time a constant function plus a constant function that represent the remaining steps after the *current_node* is located. Therefore, the worst-case performance is linear time. However, it only costs constant time to insert values at the head position of the linked list, which is its best case, where the *for*

loop is not executed. Its best-case performance is constant time *O(1)*. This method cannot be used to insert new values as the tail position because that would require the *index* to be *self.__size* which is invalid. This case is handled by the *append* method which is also constant time performance.

remove_element_at(self,index)

The worst-case performance of the *remove_element_at(self,index)* method is linear time *O(n)*. In the worst case, the index to be removed is the second last position, which requires the *for* loop to go through the whole linked list starting from the first node after *header* to the node before *tail* position. The times of loop are linear to the scale of data, i.e., the length of the linked list. And since the work inside each loop is constant, the same as that in the *insert_element_at(self,val,index)* method, only advancing the *index_node,* the total performance is a function of *n* times a constant function plus a constant function that represents the constant steps after the *index_node* is found. So, the total performance is also linear time *O(n)*. Despite the linear worst-case performance, this method also spends constant time to remove the first or the last position in the linked list. When removing the first position, the *for* loop is not executed so that the function of *n* has coefficient 0, making the total performance constant time *O(1)*. When the last position is to be removed, the *if* statement catches this special case and solves it using certain steps that are all constant, making the total performance constant *O(1)*. This can be done because my *Linked_List* class supports constant-time access to the first and last position in the list by calling *self.__header.next* and *self.__trailer.prev,* which eliminates the need to loop to find the node.

get_element_at(self,index)

The worst-case performance of *get_element_at(self,index)* method is linear time *O(n)*. The worst case is where the index to be retrieved is the second last position. The same mechanism of *insert_element_at(self,val,index)* and *remove_element_at(self,index)* applies here: the *for* loop goes through the whole linked list and does constant work in each loop, making the total performance a function of *n* plus a constant function representing the remaining work after the node is found. Similarly, due to the constant-time access the *Linked_List class* supports to the first and last position in the linked list by *self.header.next* and *self.trailer.prev*, this method has constant time performance to get element of the fist or the last node in the linked list.

Summary of the above three methods based on index walking through the linked list:

The essence of the feature of performance of methods *insert_element_at(self,val,index), remove_element_at(self,index)* and *get_element_at(self,index)* are the same: they first (1) locate the target node in the *index* position, and then (2) do certain constant work to insert, remove or return the element. Step (1) has two possible cases: the target node may be at the first (head) or the last (tail) position of the linked list, which the *Linked_List class* we implemented supports a constant-time access; or the target node may be at other positions. In the first case, step (1) is guaranteed constant time performance and so is the whole method; in the second, step (1) is always linear time and so is the whole method. It's worth noting that in the second case it doesn't help anyway whether

we keep track of some interval position in the class so that we have constant access to them. As long as the *index* to be found is not the positions we keep track of, the time cost to locate the *index* will always be a function of *n.* And though interval positions may make the coefficient slightly smaller, it all doesn't matter on the performance analysis where the *n* is pushed to infinity. Additionally, we cannot improve this linear time performance by keeping track of all the *indices* of a linked list, because in that way any change of the linked list element will cause all the *indices* to be invalid and have to be updated, which is itself linear and actually more costly. This reflects the essential feature of a *Linked_List* data structure: the only access to the linked list is through a certain number of nodes and certain ways to traverse the whole linked list. In the *Linked_List class* we implemented, we handle the structure through nodes *trailer* and *header,* and ways to traverse are *next* and *prev.* In this specific implementation, change at the head and tail positions are very convenient (*O(1)*); change in the other positions are linear time *O(n),* but are still more convenient than in an array structure because there's no need to update all the following *indices.*

rotate_left(self)

The worst-case performance of *rotate_left(self)* method is constant time *O(1)*. The worst case (the worse one of the two possible cases) is where the linked list is not empty. Then, the method is constituted of two constant time methods: first to remove the head position of the linked list and reserve the value returned; and then append that value as the new tail position of the linked list. Since it has been established above that the performance of *remove_element_at(0)* and *append_element()* are constant functions, the total performance is constant time *O(1)*.

__str__(self)

The worst-case performance of *__str__(self)* is linear time *O(n)*. When the linked list is not empty, the *while* loop goes through the whole linked list from the head position to the second last position, and does constant work of updating the *result* and advancing *current_node* inside each loop, making the total performance a function of *n.* And the other work before and after the *while* loop is constant time, so the total performance of the *__str__()* method is linear time *O(n)*.

Iterator

The worst-case performance of the *for val in linked_list* loop based on *iterator* is linear time *O(n)*. This loop is composed of *(n+1)* calls of the *__next__()* function inside my *Linked_List* class, starting from the *self.__header.next* to the *self.__trailer* where the *StopIteration* is raised, making up to a total *(n+1)* calls. Inside each call the performance of the *__next__()* function is constant, because there's no loop inside it and it only accesses the *iterator* utilizing the constant-time access built by the *__iter__()* function, keeps and returns its value, and advances it; and the value and advance call are all constant time in my *Linked_List* class. Therefore, the total performance of the iteration through the linked list is a function of *n* times a constant function, which is a function of *n.* So, the performance is linear time *O(n)*.

Approach to testing the *Linked_List class*

Testing __str__(self):

There is no risk of crash in *__str__(self)* because the *if* statement that checks whether the linked list is empty catches all possible cases of the input, so the only thing to test is whether the method produces the right format as specified in the two cases. Therefore, I didn't write a section to test it separately but call a *print()* every time after I change the *test_list* to ensure that the output is rightly formatted. To specify that I take the empty case into consideration, I called to print my initial empty *test_list* at the beginning of the test code. The *print()* works because Python automatically call the *str()* function when called to print a data type that is not string, and the *str()* function will call the *__str__()* function I defined in my *Linked_List class*.

Testing __len__(self):

The *__len__()* function cannot crash, so the only thing to test is that it outputs the right size of the linked list. Therefore, I test this function by this line of code

```
print('test_list has ' + str(len(test_list)) + ' elements')
```

called every time I print the *test_list.* In this way, I can easily check whether the size produced by *__len__()* matches the number of elements in the linked list I have just printed above.

Testing append_element_at(self,val):

Also, *append_element_at()* function does not crash because I assume the *val* is always valid by not specifying an exception in my *Linked_List class.* So the two things I want to test is (1): the function works as specified and add an element at the tail position of a linked list, which should also work well when the linked list is empty; and (2) the type and value of *val* doesn't matter, particularly that I can append "0" and "None" as value to my linked list without causing any problem in all further operations. To make sure (1) and (2) are met, I construct a *test_list* that is initially empty and append different values, including integers, strings and "0" and "None", to my *test_list,* and then print the *test_list* to see everything is appended in the right way. Further, if my further operations on the *test_list* work as expected, it is proved that "0" and "None" can safely be values of the linked list.

Testing insert_element_at(self,val,index):

In this test section I have to consider the risk of crash because it is possible for *insert_element_at()* function to raise an exception on invalid inputs. For this reason, I designed two sets of testing codes, one valid set of indices and one invalid.

(1) For the valid set of indices, I expect to see no message of "Error: Invalid Index" and see that the *test_list* and its size are updated in the right way. This is tested by printing the *test_list* and its size after insertion is done. One particular point I want to make in this section is to ensure that a linked list can have the same value stored in different nodes, so I deliberately call to insert a value identical to one already in the list to show this.

(2) For the invalid set of indices, I expect to see "Successfully Detected: Invalid Index" every time the insertion is called and to see the *test_list* is left unchanged. To ensure the first point I separated the invalid indices so that each one of them can be detected and related to a

"Successfully Detected: Invalid Index" message. For the second I print the *test_list* and its size after each invalid call of insertion and reference it against the previous print to see it is completely unchanged. Additionally, I constructed an *empty_list,* which is empty, to explicitly show that *insertion* cannot work on empty list because there's no valid index in an empty list.

Testing remove_element_at(self,index):

The same as the *insertion,* it is possible to pass invalid index to the *remove_element_at()* function. I followed the same approach to test this function as I did to test the *insertion* function, and I utilized the *empty_list* here again to show that *remove* cannot work on empty lists, either, for the same reason.

Testing get_element_at(self,index):

The approach to test *get_element_at()* function is the same as the ones testing *insert_element_at()* and *remove_element_at(),* which includes testing a valid set of index and an invalid set of index. And an *empty_list* is deliberately set to show that these three methods will not work when the linked list is empty because there's no valid index. The reason why I can test the three methods in the same way is their identical nature as I discussed above. They're all based on a *for* loop to locate the node at the target position to be operated on. This being given, all I need to test is that the *for* loop in each of three methods succeeds in finding the right position to operate, which includes the *head* position and the *(len()-1)* position, and that the invalid *index* is detected before it enters the *for* loop and makes it crash.

Testing iterator

There's no risk of crash of the *iterator* function as I did not specify any in the *Linked_List* class, so what should be tested is whether the iterator accesses every element in the linked list in the right way and right order. To test this, I printed the returned value of every *iteration* and referenced it against the previous *test_list* that I had printed. Since *iterator* does not change the list, I expect to see every element matching perfectly in the string expression of *test_list* and in the result of my *iteration.* One thing I deliberately make sure of in my test code is that the *for* loop should return only the *element* of the node, instead of the actual *node* (which is itself a private type inaccessible to users). The users should not have any way of seeing or handling the actual nodes. Therefore, the *for* loop should work rightly when a direct call of *print(element)* is successful.

Testing rotate_left(self):

Since the *rotate_left()* function is essentially based on *remove_index_at(0)* and *append_element(the removed element),* it should work fine when the two constituents are tested to be good. Therefore, I just called it and print the rotated *test_list* to see it is changed in the specified way. Also, I deliberately called it on the *empty_list* I set as reference to show that this function has no effect on an empty linked list. In this case no *IndexError* is raised because I had an *if* statement to catch the situation where the linked list is empty before the function calls *remove_element_at(0)*. This is more consistent with the real situation because the *rotate_left()* function doesn't involve any index,

and it simply doesn't change an empty linked list.

Solution to the Josephus Problem:

My solution to the Josephus Problem is exactly based on the insight given in the specification: using a linked list structure to store the position (or name or anything that can signal who is the survivor) of the group of persons, I called a *rotate_left()* and a *remove_element_at(0)* in my *Josephus()* function to accomplish one step of killing every second person. Secondly, this step is repeated as long as there are more than 1 person. Therefore, I have a *while* loop that keeps rotating and removing the first element if *len(ll)* is bigger than 1. After each remove in the loop I printed the current linked list to conform to the format given in the specification. This solution works because once you always rotate one turn and then remove the first position in the linked list, you make sure that every second person will be at the first position to be removed; and since a *remove_element_at(0)* call in the *Linked_List* structure actually removes the node at that position from the list, the whole indices are updated (descriptively speaking) and you automatically skip this removed position when you come through the same position again. This simulates the real-life situation where a person is killed and thrown out of the circle and everyone else's position is updated to adjust to the "every second" call.

The worst-case performance of this algorithm for the Josephus Problem is quadratic time $O(n^2)$. The *while* loop will be executed *(n-1)* times, which is a function of *n,* since it starts from *n* people*,* decrement *n* after each loop, and ends when *n* reaches one. Further, inside each loop the work done is linear time since *rotate_left()* costs constant time and *remove_element_at(0)* also costs constant time because of the constant-time access to the first position of the linked list; and the *print()* function is linear times as it links to my *__str__()* functiomn which is linear time *O(n)*. The work done inside each loop is the sum of two constant function and a function of *n,* which is a function of *n.* Therefore, the total performance of this algorithm is a function of *n* times a function of *n*, which is a function of n^2, so its performance is quadratic time $O(n^2)$. The reason why this is so is that I have to format the output in the specified way, which requires me to print the list after every kill, which is linear time. If the task is only to find the final survivor, the performance can be improved to linear time *O(n)* by deleting the *print()* call in my function.

Comparison of my implementation with the textbook approach.

There is no significance difference of performance between the textbook implementation of the linked list class and the way I implemented it (no difference between constant time *O(1)* and linear time *O(n)* when doing the same kind of work). But there is significance difference in terms of the ease of using for users of the class and, more importantly, the way users of the textbook implementation of linked list access and handle it is essentially different than the way my users would do, which has huge implications in risks of crash and violations of the encapsulation principle: the textbook implementation requires users to manipulate the *"Nodes"* of their linked list themselves when they insert or remove elements from the linked list, while my users are offered a descriptive attribute *"index"* through which they can access and change the position they want in the linked list. Given the way textbook implements the *Node* class, that it requires the *prev* and *next*

be passed as parameters, it is extremely difficult (almost impossible) for a user to use the structure only based on its public interface. Ease of using might not be a problem if the textbook's implementation of linked list is just to support its further implementation of stacks and deques based on linked list, but the textbook's linked list is extremely vulnerable against malicious user actions to crash the whole structure, which remains a genuine problem even if the users usually don't use the linked list class in itself. Since the textbook version requires the users of its linked list class to manipulates nodes themselves, it must set all the core attributes of the class, *header* and *trailer,* to be public, otherwise the users have no way of handling any node of the given list. This is extremely risky, because the user can crash the whole structure by setting *header* to be None. The advantage of my implementation over the textbook one lies essentially here: my users can never reach the fundamental attributes of my linked list, namely *header* and *trailer,* as I set them to be private, so my linked list class resists the risk of crash by malicious user actions. The reason I can do this is that I make my users access the positions inside the linked list by a descriptive *index,* the validity of which I can test and the actions of which I can control. This way, my implementation is actually protected from such crash. A detailed discussion of each point of difference of *Node* class, *insert,* and *remove* will follow in terms of performance, ease of use and other implications.

First, for the *Node* class, the textbook implementation has three parameters that have to be passed when a node is constructed, *element, prev and next,* while in my implementation, only *element* must be passed as parameter and the *prev* and *next* are defaulted to be None. The performance is the same in the linked list class because the only thing I have to do more is to point the *prev* and *next* to where they ought to go in an insertion method, which is constant time so would not influence the overall performance of my method. This difference doesn't have much implication in terms of ease of use, either, because users of the linked list class are not expected and should not construct nodes by themselves. Even if in the textbook implementation where they have to pass nodes as parameters of function, users should not construct their own node; rather, they do so by using *header* and *trailer* to locate the node they need. Therefore, this difference is just about the way of writing code and is innocuous.

Second, the important difference is that the insertion and remove functions of the textbook implementation requires nodes as parameters, while I apply a descriptive attribute *index* to do the same thing. For consistency with the *index* nomenclature in arrays, I have an *insert* method and *append* method separately, while the textbook implementation can insert at any position of the list using only one *insert* function. But the thing is that the performance is the same, even though it seems that the textbook *insert* and *remove* always does constant work while my *insert* and *remove* is linear worst-case performance. It is because of the more convenience and encapsulation my implementation provides for the users: to *insert* or *remove* any position in the linked list, my users only need to give that *index* and my functions will correspond that index to the actual node in the list, which is the linear-time work of my functions. As the textbook implementation doesn't have this part, it requires the users to locate the node they want to operate on by themselves, which is essentially linear in the worst case anyway. For example, if a user of linked list class want to remove the seventh element in his list, he would call *remove_element_at(6)* in my implementation of the class and my function will loop 6 times to get that node for him; while if he uses the textbook

implementation, he has to write this loop himself to get a reference to that node, and pass that node as a parameter to the *delete_node(node)* function to run a constant time. The total time cost is exactly the same, but the users have a lot more trouble with the textbook implementation. And, if they don't want the trouble, they have to sacrifice the flexibility: if they don't write a loop themselves, then they can only *insert* and *remove* around the head and tail position of the list which they can access by using several *prev* or *next,* which is still some trouble anyway. Even in these nearly best cases, my program can do equally well. The only case where textbook *insert* and *remove* outperforms mine is probably the second last position, as users of textbook implementation can use *trailer.prev.prev* to access that position but I have to loop from the beginning. But this particular case does not matter much, because there must be a worst-case linear time for users of either implementation. This advantage is even more outweighed when the risk of crash is taken into consideration as I discussed above.

To sum up, the increase of performance of textbook's implementation over mine is very slight (if any), but it is significantly more difficult to use. More importantly, by not encapsulating the core attributes as private, the textbook's implementation has no way of protecting from malicious user actions to crash the structure. In contrast, my implementation is easier to use, and is much more resilient to malicious change of my structure by making the important attributes private. The way I realize it is to provide *index* to users, which is a fully controlled way of interacting with the linked list structure defined inside my functions.