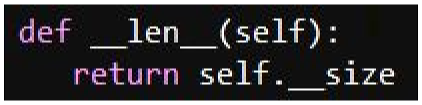
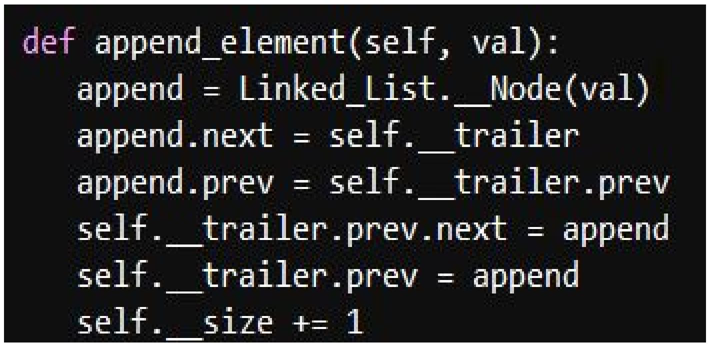
Linked lists are a unique data structure which contain nodes that store a value and have references to the next node's value (.next). In addition to that, the list has two pointers commonly named 'head' and 'tail' which respectively point to the first and last element in the list. Furthermore, there is a type of linked list which offers many advantages in comparison to normal linked lists which are doubly linked lists with sentinel nodes. These structures are actually the same as linked lists but with the addition of having references to the previous element in the list (.prev). Moreover, the first and last nodes in the list are now empty nodes respectively named 'header' and 'trailer' with header only having a reference to the next node and trailer only having a reference to its previous node. These structures in particular offer many advantages in the performance of many of its methods with the exception of a few cases where it lacks in performance.

The worst case performance of its __len__ method is O(1) (constant time). This is simply due to the fact that it is just returning the size of the list without any additional steps.

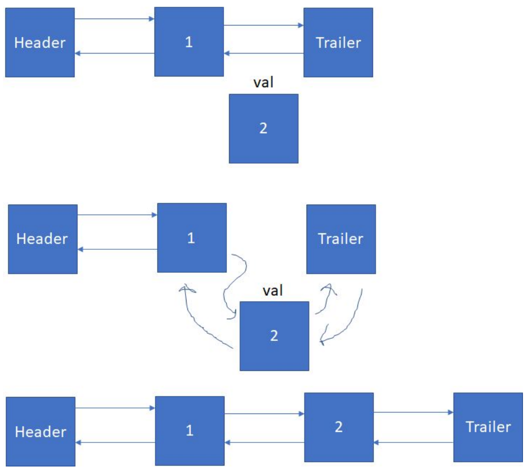


```
def __len__(self):
    return self.__size
```



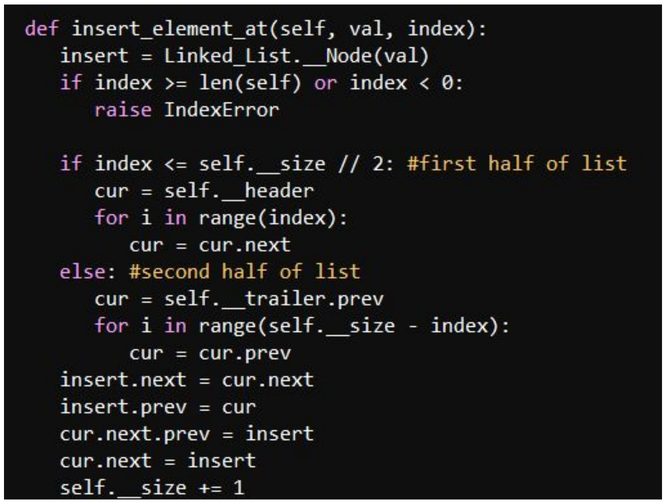
```
def append_element(self, val):
    append = Linked_List.__Node(val)
    append.next = self.__trailer
    append.prev = self.__trailer.prev
    self.__trailer.prev.next = append
    self.__trailer.prev = append
    self.__size += 1
```

Moreover, the worst case performance of its append_element method is O(1) (constant time). This occurs because all that needs to be done is to connect the trailer's old previous node with the appended value and connect the trailer to the appended value which will always take a set number of method calls everytime time append_element is called.



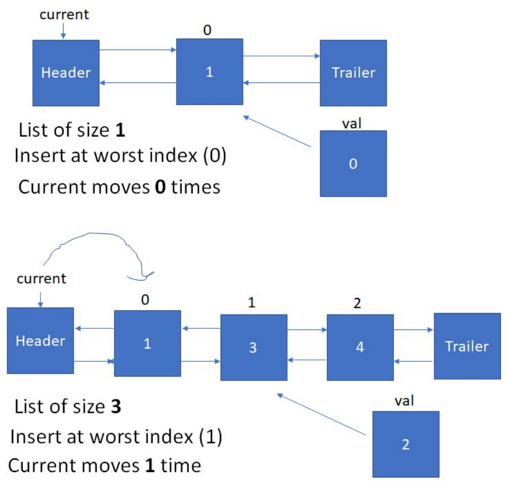
Furthermore, this means that a set number of arrows (references to other nodes) are changed so that they are pointing to other nodes as shown in the image on the left of the value 2 being appended. In this case, 4 arrows are going to be reassigned every time append_element is called.

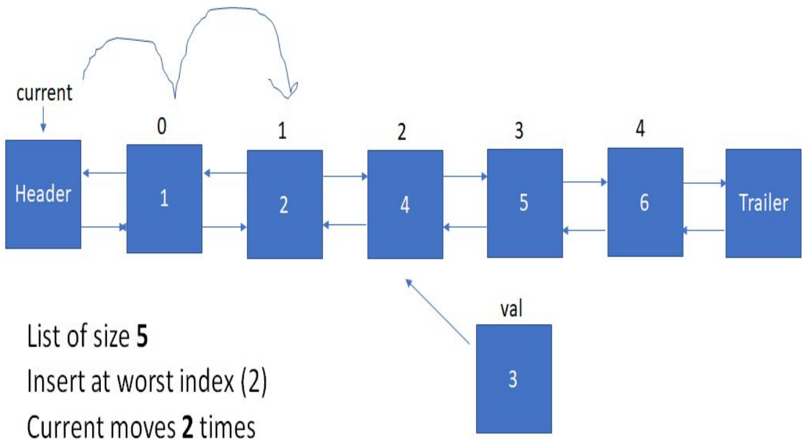
In addition, the worst case performance of its insert_element_at method is O(n) (linear time). Although there are conditionals to check for an index error, this runs in constant time with no effect on the overall performance of the method. Moreover, the conditionals for checking which side to start from (header or trailer) have a small effect on the performance making it O(n/2), but this would just simplify to O(n). This linear time performance occurs because something (current) needs to be pointing to the value before the index where the new value is going to be inserted. In order to get this current pointer to be before the index where the new value is inserted, the list needs to be traversed. Moreover, the worst case index (the middle index which is equidistant from both the header and trailer) makes it so that an increase in size leads to an increase in time taken to traverse the list. The image below shows an example of how an increase

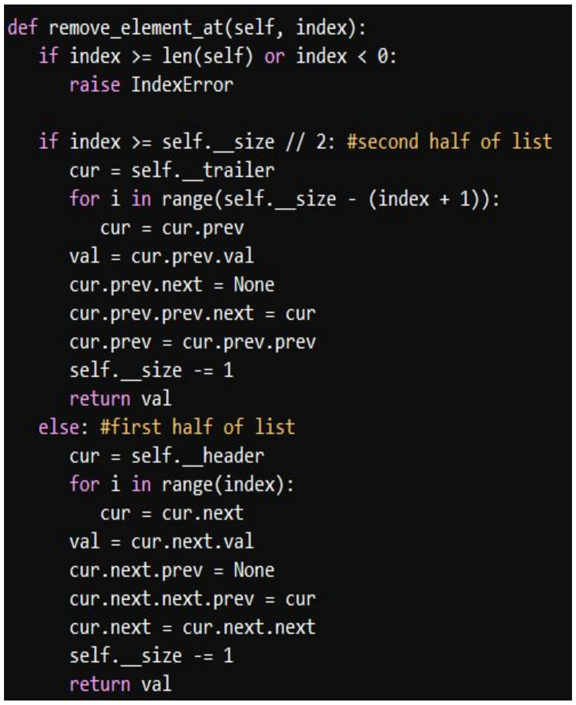


```
def insert_element_at(self, val, index):
   insert = Linked_List.__Node(val)
   if index >= len(self) or index < 0:
      raise IndexError

   if index <= self.__size // 2: #first half of list
      cur = self.__header
      for i in range(index):
         cur = cur.next
   else: #second half of list
      cur = self.__trailer.prev
      for i in range(self.__size - index):
         cur = cur.prev
   insert.next = cur.next
   insert.prev = cur
   cur.next.prev = insert
   cur.next = insert
   self.__size += 1
```

in size of the list leads to an increase in the numbers of steps taken through the movement of its current pointer.

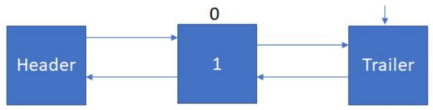




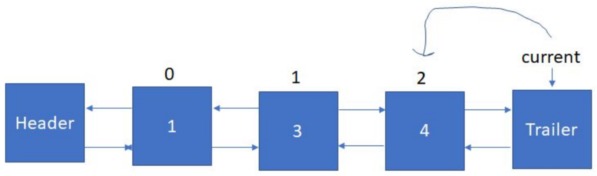


```
def remove_element_at(self, index):
   if index >= len(self) or index < 0:
      raise IndexError

   if index >= self.__size // 2: #second half of list
      cur = self.__trailer
      for i in range(self.__size - (index + 1)):
         cur = cur.prev
      val = cur.prev.val
      cur.prev.next = None
      cur.prev.prev.next = cur
      cur.prev = cur.prev.prev
      self.__size -= 1
      return val
   else: #first half of list
      cur = self.__header
      for i in range(index):
         cur = cur.next
      val = cur.next.val
      cur.next.prev = None
      cur.next.next.prev = cur
      cur.next = cur.next.next
      self.__size -= 1
      return val
```

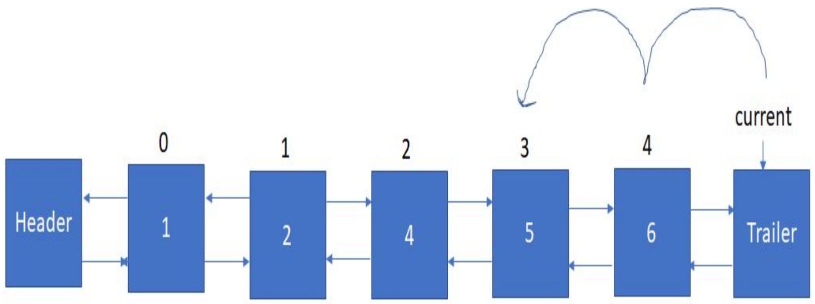
Additionally, the worst case performance of remove_element_at is O(n) (linear time). This is the case because similar to insert_element, as the size of the list increases, the time it takes to complete the remove_method increases. Moreover, this method also involves pointing something (current) to a node that is next to or previous to the node that is going to be removed. The image below shows that current needs to be moved more as the size of the list increases.



List of size **1**
Remove at worst index (0)
Current moves **0** times

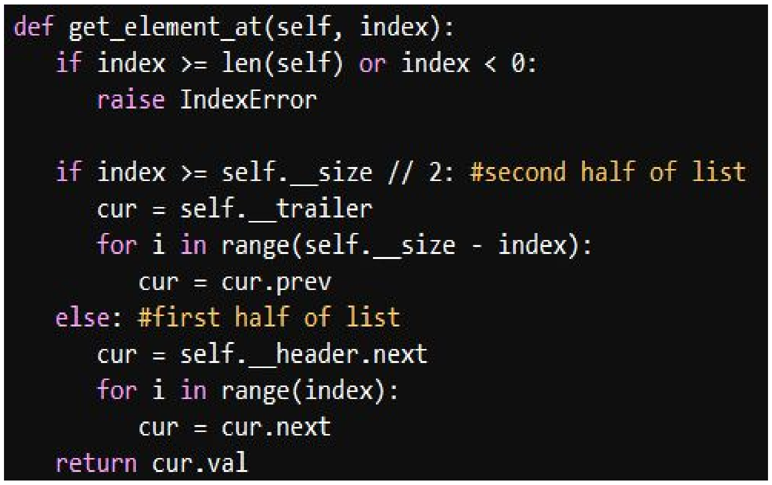


List of size **3**
Remove at worst index (1)
Current moves **1** time

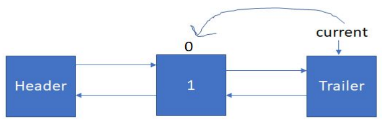


List of size **5**
Remove at worst index (2)
Current moves **2** times

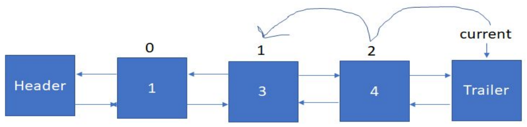
Similarly to remove_element_at and insert_element,_at, the get_element_at method also has a worst case performance of O(n) (linear time). However, instead of having something (current) placed besides the node of interest like in remove or insert, get_element_at places current at the node of interest. Then it just returns the value of where current is. This method also increases in time as the size of the list increases. The image below demonstrates how the numbers of steps to move current increases as the size of the list increases.



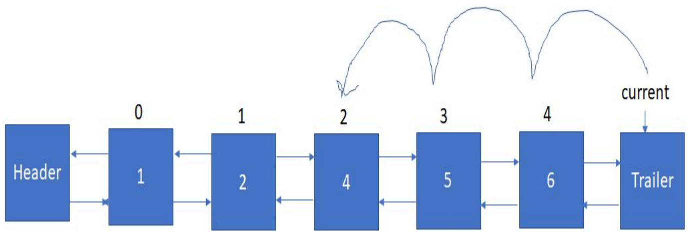
```
def get_element_at(self, index):
   if index >= len(self) or index < 0:
      raise IndexError

   if index >= self.__size // 2: #second half of list
      cur = self.__trailer
      for i in range(self.__size - index):
         cur = cur.prev
   else: #first half of list
      cur = self.__header.next
      for i in range(index):
         cur = cur.next
   return cur.val
```



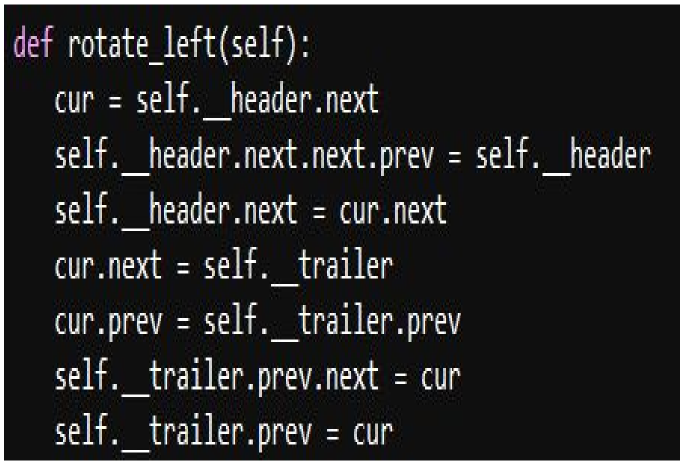
List of size **1**
Get at worst index (0)
Current moves **1** time



List of size **3**
Get at worst index (1)
Current moves **2** times

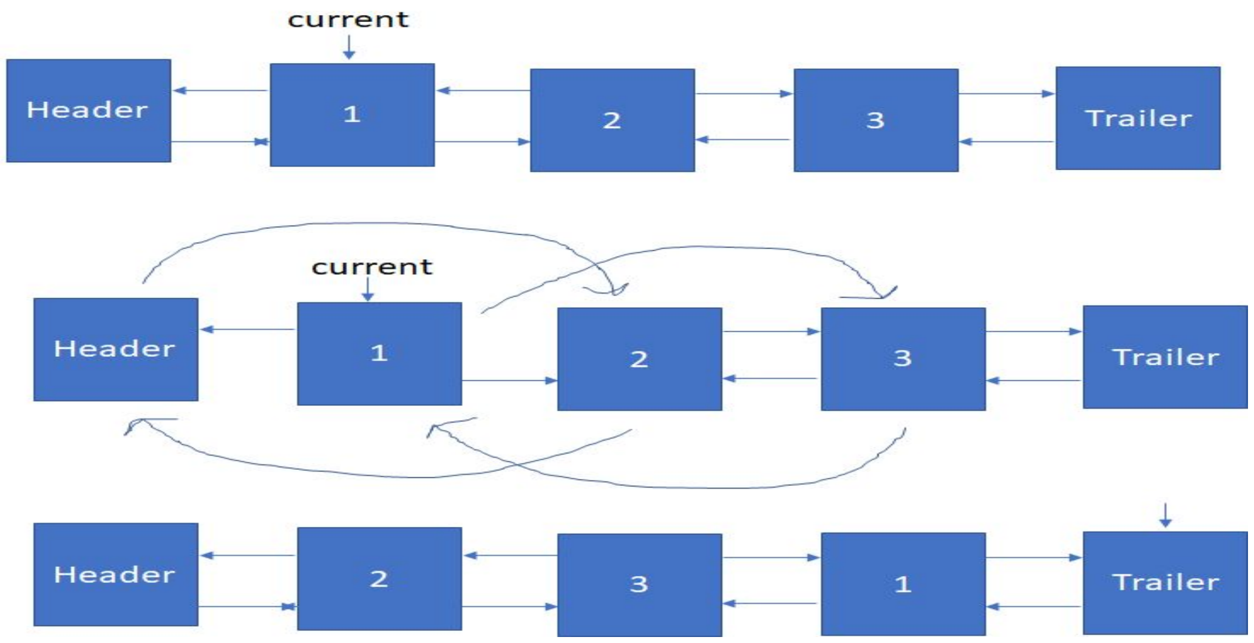


List of size **5**
Get at worst index (2)
Current moves **3** times

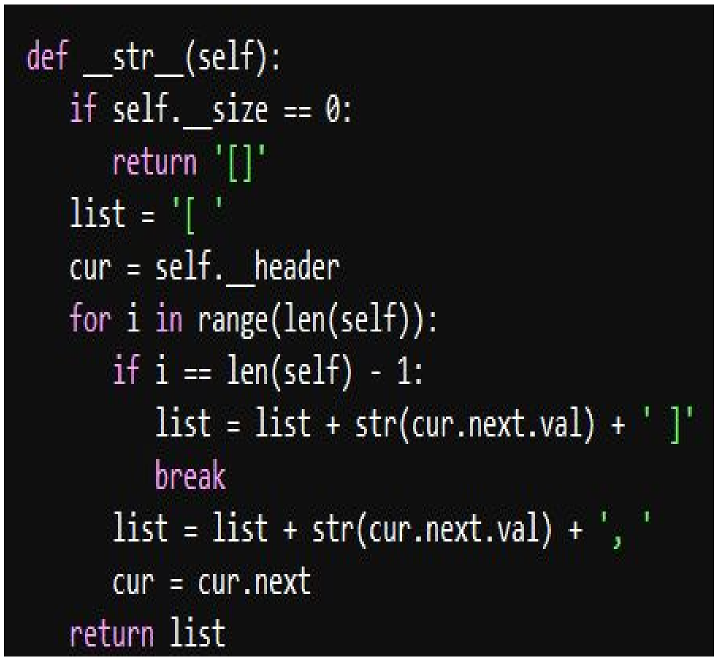


```
def rotate_left(self):
    cur = self._header.next
    self._header.next.next.prev = self._header
    self._header.next = cur.next
    cur.next = self._trailer
    cur.prev = self._trailer.prev
    self._trailer.prev.next = cur
    self._trailer.prev = cur
```

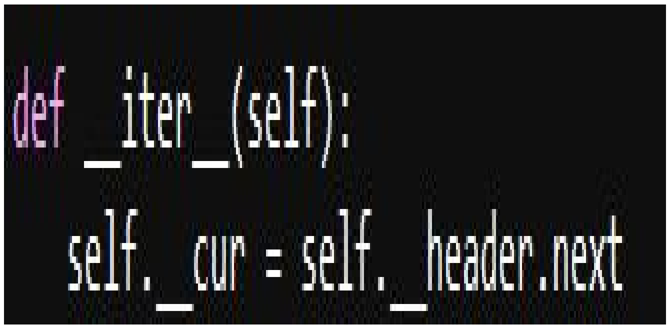
Similar to the append_element method, rotate_left's worst case performance is O(1) (constant time). This is because it also involves a set number of method calls regardless of the size. This method connects the header to the node that is two spots away from it and connects the trailer to the node that is in between the header and the node two spots away from it. The image below demonstrates how the arrows are reassigned in this method. Again just like in append_element, 4 arrows are going to be changed every time rotate_left is called.



The worst case performance for the __str__ method is O(n) (linear time). This is because as the size of the list increases, more commas need to be added to the list. For example for a list of size two ([1, 2]), two steps would need to be taken one of which is to add the end brace and the other to add the comma. Moreover, if the list increased to a size of three ([1, 2, 3]), three steps would need to be taken, one of which is to add the end brace and the other two to add two commas.



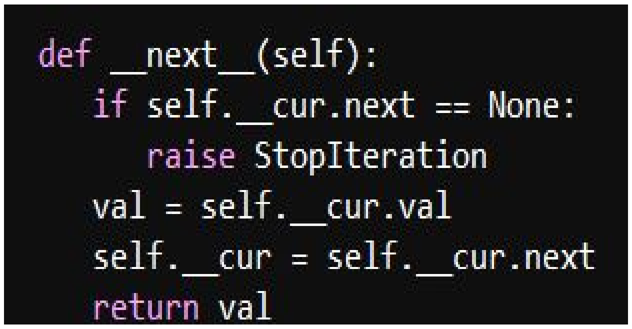
```
def __str__(self):
    if self.__size == 0:
        return '[]'
    list = '[ '
    cur = self.__header
    for i in range(len(self)):
        if i == len(self) - 1:
            list = list + str(cur.next.val) + ' ]'
            break
        list = list + str(cur.next.val) + ', '
        cur = cur.next
    return list
```



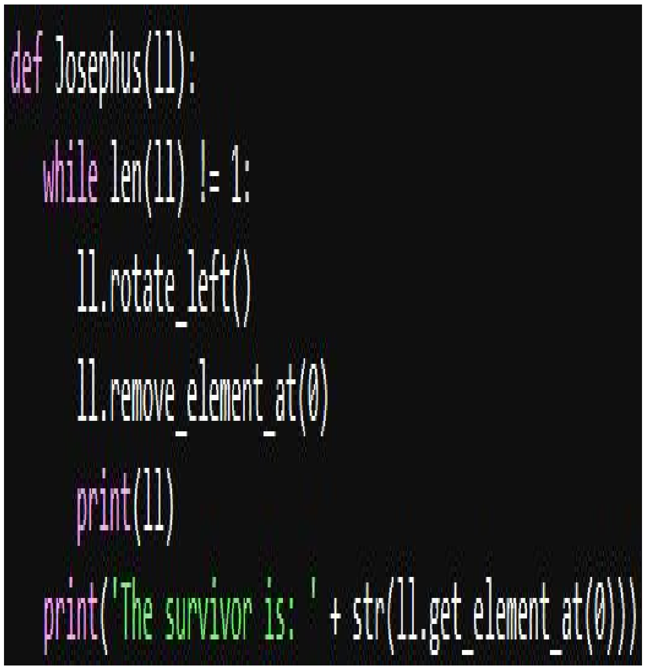
```
def __iter__(self):
    self.__cur = self.__header.next
```

Moreover, the __iter__ method’s worst case performance is O(1) (constant time) because it just initializes a new attribute (cur) for walking through the list at the header.next.

Furthermore, the __next__ method’s worst case performance is also O(1) (constant time). This method just retrieves the next value and returns it using self.__cur from the __iter__ method to do so. Although there is a conditional to check if cur’s next value is None, it does not affect the overall performance of the method.



```
def __next__(self):
    if self.__cur.next == None:
        raise StopIteration
    val = self.__cur.val
    self.__cur = self.__cur.next
    return val
```

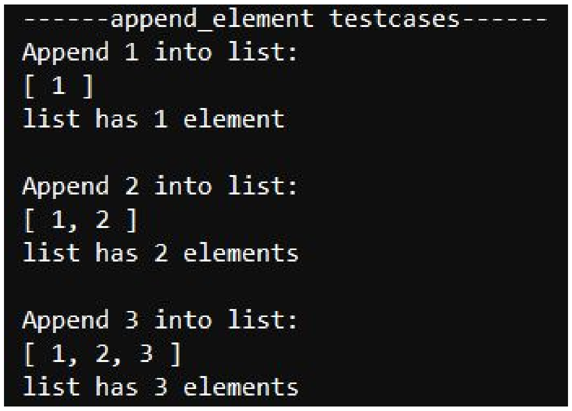


```
def Josephus(ll):
  while len(ll) != 1:
    ll.rotate_left()
    ll.remove_element_at(0)
    print(ll)
  print('The survivor is: ' + str(ll.get_element_at(0)))
```

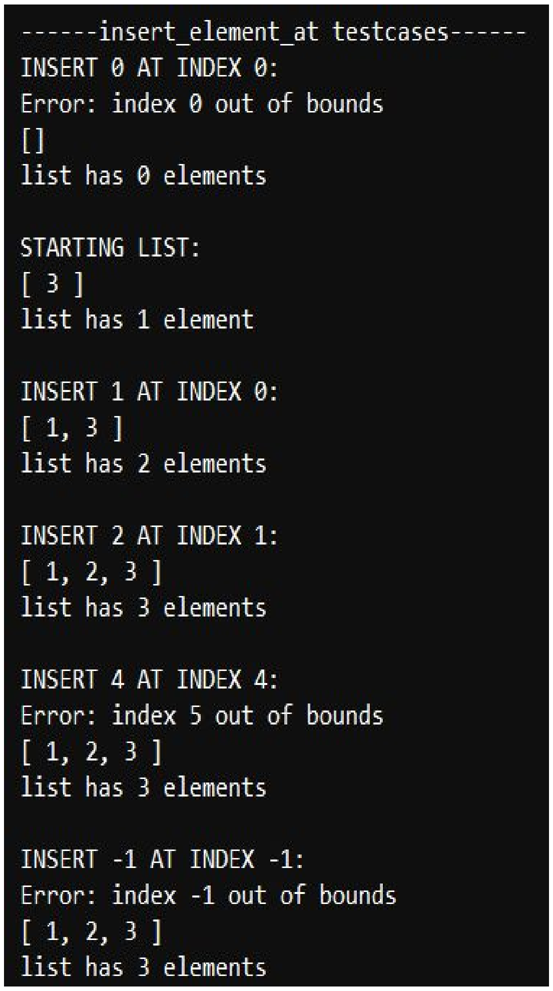
The solution to the Josephus problem's worst case performance is O(n) (linear time). This is the case because of the while loop. Moreover, as the size of the list increases, the longer it will take for the Josephus solution to run. This is because this method keeps removing elements until there is only one element in the list left. This means that if there are more elements in the list, it will require more removals to get to one element remaining in the list, thus taking more time. Although remove_element_at is normally O(n), in this case it is O(1) (constant time) due to the fact that the first value in the list is always being removed. This means that the remove method never has to traverse through the list.

My general approach to testing my methods was to test if the method worked correctly in all cases and raised errors when necessary. Moreover, my test cases checked when the list was empty, when there was only one element, and when there were many elements.

Firstly, for the append_element method, my test cases checked if the size updated correctly (increased by one after each call) and if the value got appended correctly.



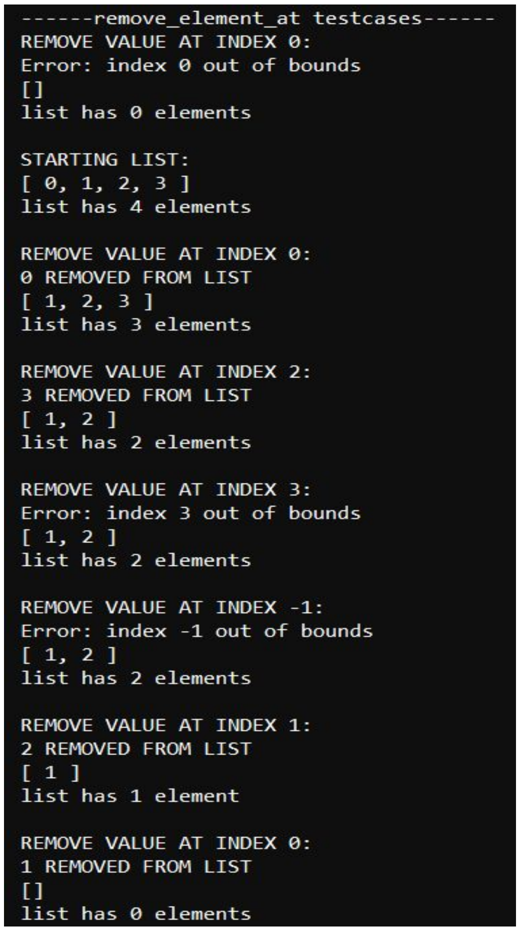
```
------append_element testcases------
Append 1 into list:
[ 1 ]
list has 1 element

Append 2 into list:
[ 1, 2 ]
list has 2 elements

Append 3 into list:
[ 1, 2, 3 ]
list has 3 elements
```



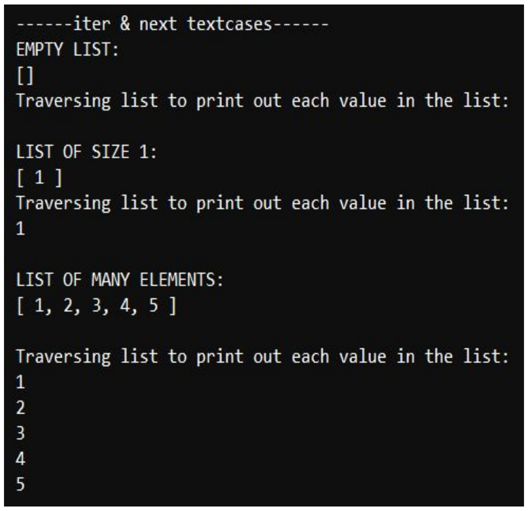
```
------insert_element_at testcases------
INSERT 0 AT INDEX 0:
Error: index 0 out of bounds
[]
list has 0 elements

STARTING LIST:
[ 3 ]
list has 1 element

INSERT 1 AT INDEX 0:
[ 1, 3 ]
list has 2 elements

INSERT 2 AT INDEX 1:
[ 1, 2, 3 ]
list has 3 elements

INSERT 4 AT INDEX 4:
Error: index 5 out of bounds
[ 1, 2, 3 ]
list has 3 elements

INSERT -1 AT INDEX -1:
Error: index -1 out of bounds
[ 1, 2, 3 ]
list has 3 elements
```

Moreover, for the insert_element_at method, my test cases checked if the size updated correctly (increased by one after each call), if the value got inserted at the correct index, and if errors were raised when appropriate. My test cases tested the different errors that could arise from inserting into an empty list and inserting at an index that is out of bounds (negative or greater than or equal to the size of the list). In particular, I checked if insert worked properly for the ends of the list. For the index = 0, the value should have been placed in the beginning of the list and for index = size of list - 1, the value should have been placed in the second to last spot of list (not the end of the list).

Furthermore, for the remove_element_at method, my test cases checked if the size updated correctly (decreased by one after each call), removed the correct element from the list, and if errors were raised when appropriate. My test cases raised errors when the index was out of bounds (negative or greater than or equal to the size of the list) and when an element was being removed from an empty list. In addition to removing the proper element from the list, it also checked to see if the removed value was returned properly.



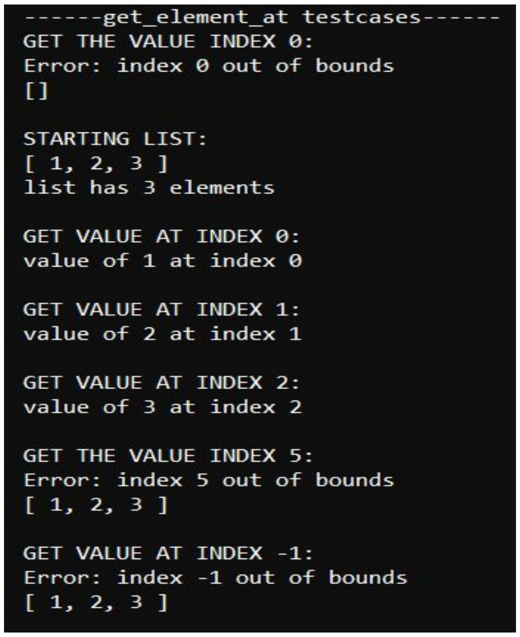
```
------remove_element_at testcases------
REMOVE VALUE AT INDEX 0:
Error: index 0 out of bounds
[]
list has 0 elements

STARTING LIST:
[ 0, 1, 2, 3 ]
list has 4 elements

REMOVE VALUE AT INDEX 0:
0 REMOVED FROM LIST
[ 1, 2, 3 ]
list has 3 elements

REMOVE VALUE AT INDEX 2:
3 REMOVED FROM LIST
[ 1, 2 ]
list has 2 elements

REMOVE VALUE AT INDEX 3:
Error: index 3 out of bounds
[ 1, 2 ]
list has 2 elements

REMOVE VALUE AT INDEX -1:
Error: index -1 out of bounds
[ 1, 2 ]
list has 2 elements

REMOVE VALUE AT INDEX 1:
2 REMOVED FROM LIST
[ 1 ]
list has 1 element

REMOVE VALUE AT INDEX 0:
1 REMOVED FROM LIST
[]
list has 0 elements
```



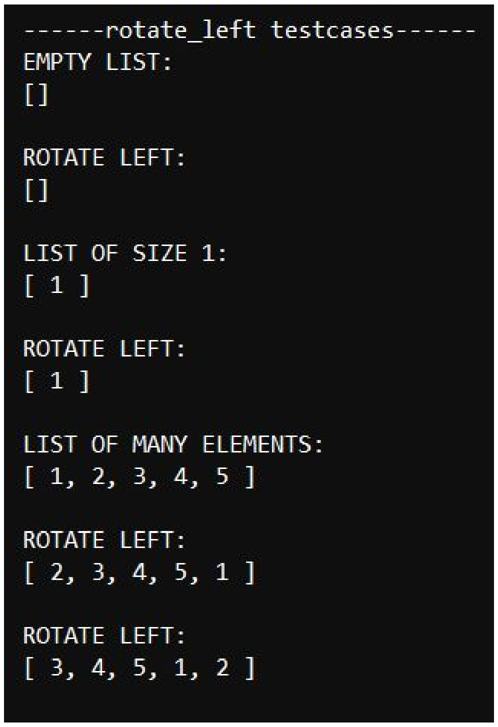
```
------iter & next textcases------
EMPTY LIST:
[]
Traversing list to print out each value in the list:

LIST OF SIZE 1:
[ 1 ]
Traversing list to print out each value in the list:
1

LIST OF MANY ELEMENTS:
[ 1, 2, 3, 4, 5 ]

Traversing list to print out each value in the list:
1
2
3
4
5
```

Also, for the __iter__ and __next__ methods, my test cases checked if a linked list could be traversed. Moreover, my test cases did so by printing out each element in a linked list of varying sizes (empty, size of 1, many elements) using a for loop which traversed through the linked list.

Additionally, for the get_element_at method, my test cases checked if the correct element was got from the list and if errors were raised when necessary. My test cases raised errors when the index was out of bounds (negative or greater than or equal to the size of the list) and when the get method was used when the list was empty.



```
------get_element_at testcases------
GET THE VALUE INDEX 0:
Error: index 0 out of bounds
[]

STARTING LIST:
[ 1, 2, 3 ]
list has 3 elements

GET VALUE AT INDEX 0:
value of 1 at index 0

GET VALUE AT INDEX 1:
value of 2 at index 1

GET VALUE AT INDEX 2:
value of 3 at index 2

GET THE VALUE INDEX 5:
Error: index 5 out of bounds
[ 1, 2, 3 ]

GET VALUE AT INDEX -1:
Error: index -1 out of bounds
[ 1, 2, 3 ]
```



```
------rotate_left testcases------
EMPTY LIST:
[]

ROTATE LEFT:
[]

LIST OF SIZE 1:
[ 1 ]

ROTATE LEFT:
[ 1 ]

LIST OF MANY ELEMENTS:
[ 1, 2, 3, 4, 5 ]

ROTATE LEFT:
[ 2, 3, 4, 5, 1 ]

ROTATE LEFT:
[ 3, 4, 5, 1, 2 ]
```

Lastly, For the rotate_left method, my test cases checked if the list was rotated to the left properly using an empty list, list of size one, and list of many elements.

The textbooks' use of a double linked list is similar in many ways to my implementation of a double linked list but there are also many differences between the two. First off, in both the textbook's implementation and my implementation there is a nested class inside some sort of linked list class. However, the nested class in my implementation represents the node (node class) while the textbook's implementation represents the position of a node (position class). The position class also has a couple of methods that need to be defined such as eq, ne, and element in addition to its init function. On the other hand, the node class only requires the init function to be defined. The biggest difference is in its accessor and mutator methods. The textbook has 5 different mutator methods just to add a new node into the list which are add_before, add_after, _insert_between, add_last, and add_first. However, my implementation condenses these methods into just 2 methods which are append_element and insert_element_at. Furthermore, the textbook has 4 different accessor methods which are called first, last, before, and after. However, in my implementation these methods are condensed into one method called get_element_at. A key difference between the two implementations is the fact that the textbook's implementation has a replace method to replace the node at a certain position. However, the same effect can be reproduced using insert / append and remove in my implementation. On the other hand, methods like the element / delete / iter method in the textbook's implementation function are the same as the get_element_at / remove_element_at / iter method in my implementation. In short, both my implementation and the textbook's implementation offer many different and similar ways to create doubly linked lists.