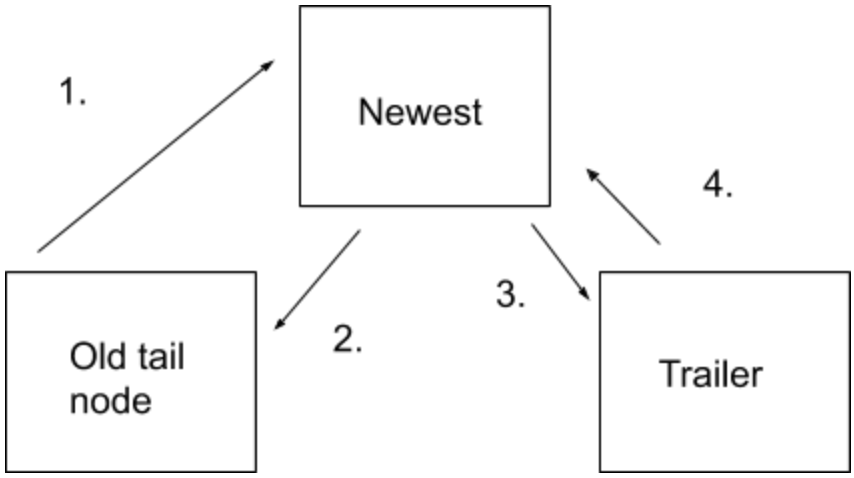
Anatoly Osgood

The first method in my LinkedList file was the initialization for the Node class. In this method I take the value passed to the node and set variable called “value” equivalent to the value passed by the user. I also created next and previous variables for the node which I initialized to None. The performance of this method is O(1) or constant time. This is because, this method works in the same way every time.

Moving onto the LinkedList class, the first method is again the initialization method. We are creating a sentiniled, doubly-linked list. Therefore, I create a header and trailer node. I initialize the header’s next to point at the trailer and the trailer’s previous to point at the header. I also set both the header’s previous and trailer’s next to None as they will never point at anything because the list is not circular. I also create the variable size to store the length of the list. I set this value to 0 in the initialization method as at the time of LinkedList creation, the size will be 0. The performance of this method is also O(1) or constant time. This is because, this method works in the same way every time.

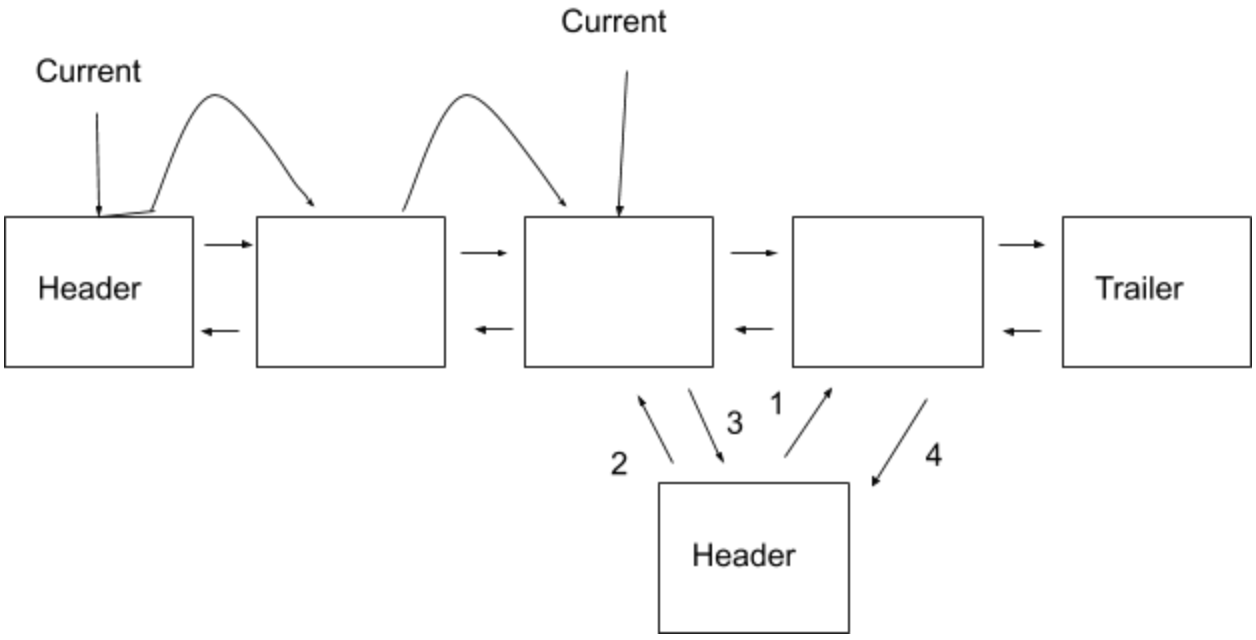
The next method in the LinkedList class is the length method. This method is very short, all it does is return the length of the LinkedList. This is accomplished by returning the variable size. The performance of this method is also O(1) or constant time. This is because, this method simply returns a value.

The next method in the LinkedList class is the append_element method. This method adds a node to the tail position of the list and increases the size of the list by one accordingly. Importantly, this method is the only way of adding a node the tail position. First, the method creates a new node with a value equivalent to the value passed to it by the user, we’ll call this node “newest”. Then, it looks at the node in the current tail position, takes its next variable and sets it to point at newest. Then, the method takes newest’s previous variable and sets it equal to the node that is in the tail position. After that, it sets newest’s next variable and points it at trailer, while setting trailer’s previous variable to point at newest. It then increments the size of list by 1 as 1 new node as has been appended in the tail position. The performance of this method is also O(1) or constant time. This is because, no matter what the input is, this method works in the same way every time. This method is shown in a diagram below with numbers indicating the order that it was completed:



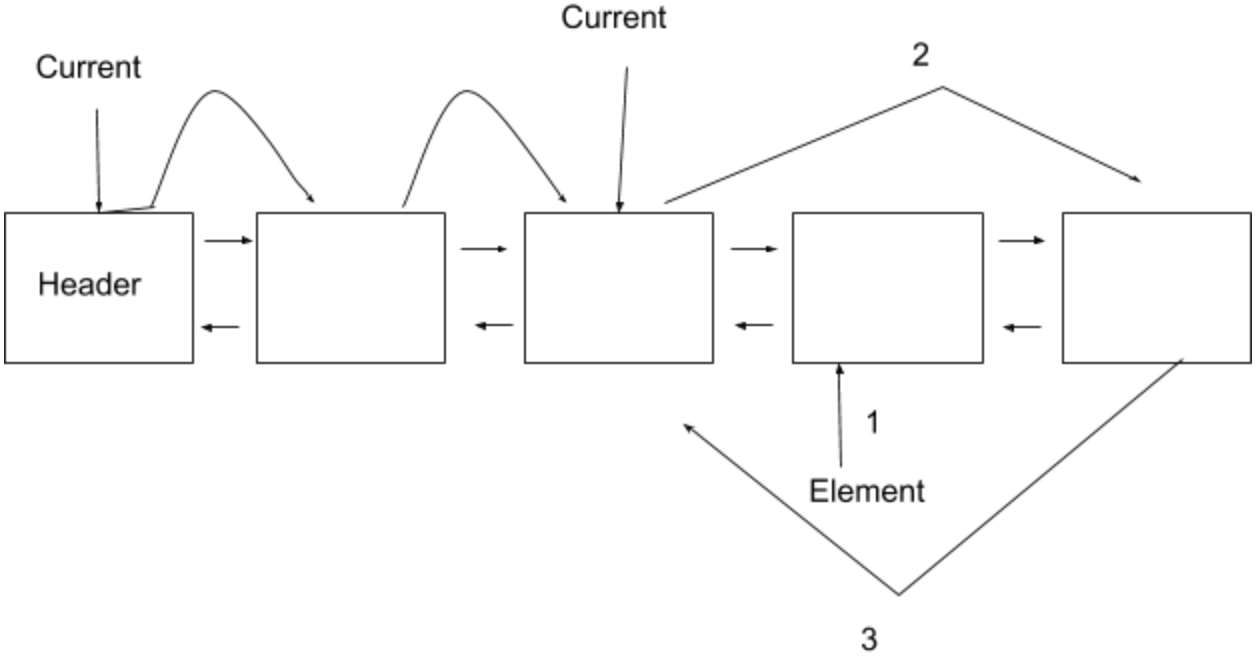
The next method in the LinkedList class is the insert_element_at method. This method adds a node at any index except for the tail index. This is because the append_element method is used for adding any nodes to the tail. If the user tries to add a node to the tail, an IndexError is raised. First, this method checks the index that is passed. If the index passed is greater than or equal to the size of the list or less than 0, an IndexError is raised. This is to avoid any incorrect indices or an attempt to add a node at the tail. If the index passed is in the second half of the list (indices greater than the size divided by 2), the method creates a new node with a value equivalent to the value passed to it by the user, we'll call this node "newest". Then, I created a variable called current which starts at the trailer position and walks backward to the node right after the index we need to insert the node. Then I set newest.previous equal to current.previous so that newest is between current and the node that used to before it. Then I set newest.next to point at current and current.previous to point at newest. Finally, I pointed the node before newest (accessed by newest.previous) to point its next at newest. I then increment the size by one.

Otherwise, if the index is in the first half of the list, (less than or equal to the size divided by 2) the method creates a new node with a value equivalent to the value passed to it by the user, we'll call this node "newest". Then, I created a variable called current which starts at the header position and walks through the list the node right before the index where newest is to be inserted. Then it sets newest's next variable to point at the next node in the list (via current's next) and sets newest's previous variable to point at current. Then, the method sets current's next and points it at newest. Finally, I set the node after newest to use its previous to point at newest. I used newest.next to access the node ahead of it because, current.next now points at newest. The performance of this method is O(n) or linear time. This is because of the for loop, which at worst case would be iterated n-2 times. The diagram below illustrates how this part of the method works:



The next method in my LinkedList class is the remove_element_at method. This method removes and returns the value of the node at the index. The first thing I do is check the index value passed by the user. If the index passed is greater than or equal to the size of the list or less than 0, an IndexError is raised. This is to avoid any incorrect indices. If the index passed is in the second half of the list (indices greater than the size divided by 2), I create a variable called current at the trailer. It starts at the trailer position and walks backward to the node right after the index we need to remove the node. I then create a new variable called element which is the node I'd like to remove It is accessed by calling current.previous. I then set current.previous to element.previous and current.previous.next to current so that element is no longer pointed to by either node. Finally, I decrement the size by 1 and return the value of element as the spec requires.

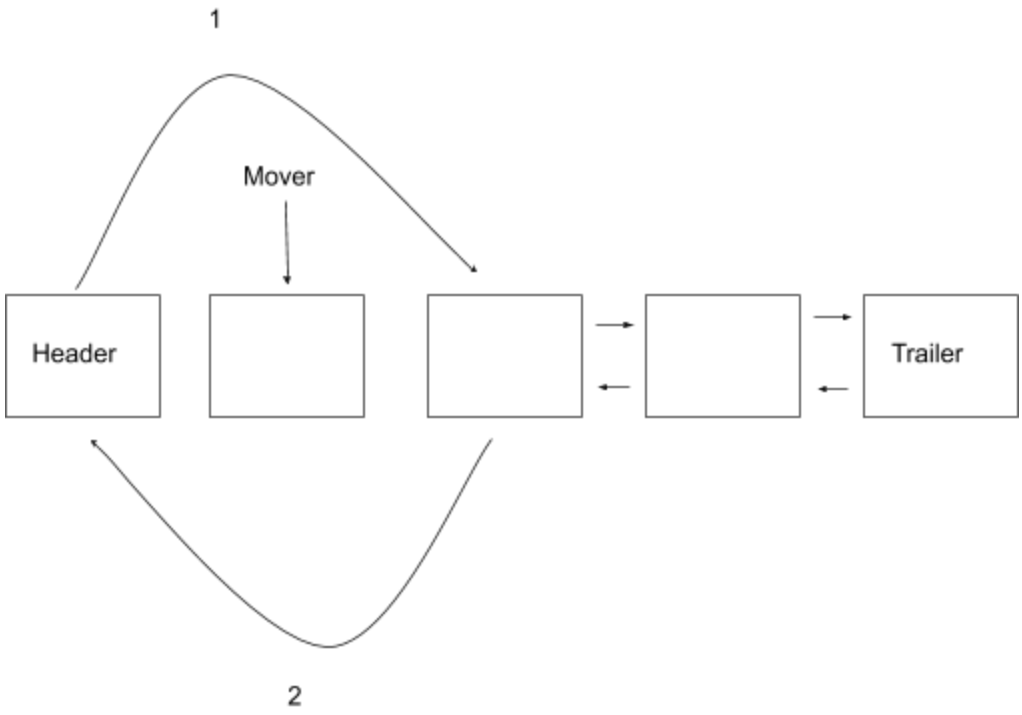
Otherwise, if the index is in the first half of the list, (less than or equal to the size divided by 2), I again create a variable called current at the header. Just like in insert_element_at, I loop forward through the list so that current is equal to the node right before the element that the user would like to remove. Then I create a variable called element which is the node that the user would like to remove. Then, I set current.next equal to the node after the node that is removed. This accessed by using element.next. Then I set that node's previous value to point at current by using current.next.previous. Finally, I decremented the list's size by 1 and returned the value of the node that was removed. After this, the node (element) is garbage collected and removed. The worst case performance of this method is O(n) or linear time. This is because of the for loop which would go through almost the entirety of the list (n-2) in the worst case. The diagram below illustrates how this part of the method works:

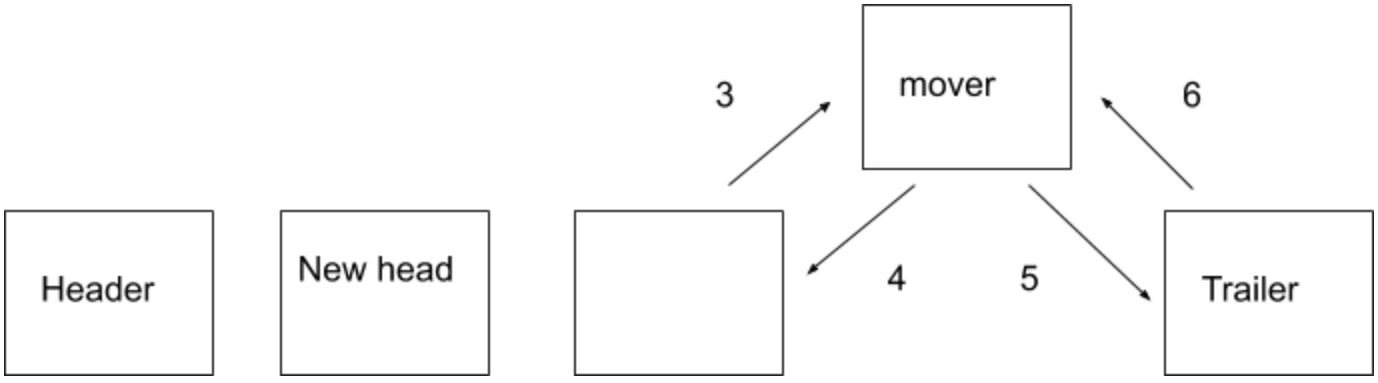


The next method is get_element_at. This method returns the value of the node at the index passed by the user. The first thing I do is check the index value passed by the user. If the

index passed is greater than or equal to the size of the list or less than 0, an IndexError is raised. This is to avoid any incorrect indices. If the index passed is correct and in the first half of the list (less than or equal to the size divided by 2), I create a variable called current at the header. Just like in insert_element_at, I loop forward through the list so that current is equal to the node right before the element that the user would like to see the value of. Then, I return the value of the next node. If the index passed is correct and in the second half of the list (greater than the size divided by 2), I create a variable called current, but at the trailer. I then walk backwards through the list so that current is equal to the node right after the element that the user would like to see the value of. Then, I move to the node before current and return its value. The performance of this method is O(n) because the for loop in the worst case would go through n-2 elements.

The next method is rotate_left, which is essential for the Josephus question. This method rotates the list left by one position. This means that the index of every element decreases by 1, except for the head. The head moves to the back of the list and becomes the new tail. If the size of the list is 0, this method should do nothing. The first thing I do is check if the size of the list is 0 or 1. This is because if the list has a length of 0 or 1, nothing happens to the appearance of the list. Otherwise, I create a variable called mover which is equivalent to the head variable (the variable at the front of the list that header.next points at). Then I set header.next to point at mover.next (the second node in the list). I then set the second node's previous (accessed by mover.next.previous) to point at header. This has successfully pulled the head node from the front of the list, but we must now place it in the tail position. I access the current tail node's next to point at mover by using trailer.previous.next. I then point mover's previous at the old tail node using trailer.previous. Finally, I set mover.next to point at trailer and pointed trailer's previous at mover. The performance of this method is O(1) or constant time. The 2 part diagram below illustrates how this method works:





The next method is __str__ which returns a string representation of the list's contents. The first thing I do is check if the size of the list is 0, if it is I return '[ ]' as required by the project spec. If the length of the list is 1, then I return '[x]', where x is the only value in the list. Again this is done in accordance with the demands of the spec. Otherwise, I used a while loop to walk through the list. I used the condition of "while current.next.next.value is not None" as I wanted the loop to stop at the second to last value in the list. This is because of the way I build the string representation of the list. For every node I attach its value and the required comma. However, for the last node, the representation does not require a comma. So, I chose to break out of the loop on the second to last node and then add solely the value of the last node in the list, along with a closing bracket. The performance of this method is O(n) or linear because of the while loop. It loops through the entire length of list minus 1.

The next method is __iter__ which initializes an attribute for walking through the list. I created a variable called __iter_walker, which is set to the header. I set it to header because the list may be empty so my variable must start at the header. This is variable is the variable used in the next method to walk through the list and return the value of each node in the list. The performance of this method is O(1) as it's simply creation of a variable equivalent to the header.

The final method is __next__ which uses the variable created in __iter__ (__iter_walker) to iterate through the list. The method first checks if the value of the next node is None, if it is it raises a StopIteration exception. Otherwise, __iter_walker goes to the next node and returns its value. This method, in conjunction with __next__, is essential for looping through the list using the for loop as this method actually drives __iter_walker to go through the list. The performance of this method is O(1) or constant as the method always raises a StopIteration or iterates __iter_walker onto the next node and returns the value of the node.

For the Josephus problem I use the methods created in my LinkedList file. I use a while loop to make sure the last value is kept and not removed. In the while loop I use two functions. First, I rotate the list left (using rotate_left), then I remove the node at the front of the list (using remove_element_at) as specified by the spec. Then, I print the list. Once there is only one value left, the program breaks out of the while loop and prints the final survivors number like in the spec using the get_element_at. In the main method, we take input from the user to see how many people are in their Josephus problem. Then we initialize the linked list that will be used to store each person's number. The performance of this file is $O(n^2)$ because of the while loop and print function calling the __str__ method on each part of the while loop. The __str__ method is time O(n), so this combines to give us $O(n^2)$. On each run on the file we have to go through n elements because of the while loop, and iterate through the list of n objects every single run.

I will now discuss the testing component of my program. The first thing I tested was if appending to the list added an element at the new tail position and increased the size by one. I created a Linked List and then used append_element to add elements, print the list and it's size out to verify the method was working. Then, I tested if inserting an item at a valid index increased the size by one and correctly modified the list's structure. I tried a variety of numbers in positions in the first and second half of the lists to verify both my conditionals in the insert_element_at method worked as desired. Each time I would print the size and the list to verify the method was working as desired. Then I tried inserting elements at incorrect indices using try and except to see if I was correctly receiving IndexErrors. I correctly received IndexErrors every time, telling me that my method correctly blocked users from inserting elements at the tail or other incorrect indices (ex. Negative index). Then I tested whether removing an item at a valid index decreased the size by one and correctly modified the list's structure. I created a list with 10 nodes and then deleted nodes from both halves of the list, testing my conditionals. I also printed the size and list every time to make sure the method worked as expected. I also deleted two nodes and then checked the list to make sure using the remove_element_at method performed as expected. I also used the print statement once to make sure it was correctly returning the value of the node that was removed. Then, I tested the remove_element_at method by giving it invalid indices, which the method correctly caught. I tested whether calling the length of the list always returns the number of values stored in the list. I did this by append elements and printing the size to make sure the length was correctly

calculated every time a node was added. I already tested that size worked when nodes were removed. Another test I did was to make sure my string representation of my list was always correct. To do so, I printed my list when it was empty, when it had 1 node, and when it had a lot of nodes. I also looped through a list to make sure my __iter__ method was working and accessed nodes using get_element_at to make sure that method worked.

Now I will discuss differences between our and the textbook's implementation of a doubly-linked list. The first thing I noticed in the textbook's linked list implementation was the _insert_between class. I think our insert element is much superior as its simpler and simply requires an index to insert the node. I also think a lot of methods like first, last, before, and after are unnecessary and much prefer our get_element_at method. I think these elements are redundant and we much better address the needs of the user in our implementation. Add_first, add_last, add_before, and add_after all continue this trend of unnecessary methods that simply add a lot more code in my opinion. Why not just let the user define where they want to put a node based off of an index-system? I noticed the textbook's __iter__ method was pretty much identical and contained the same logic of starting a node at the head and then moving it node by node through the list. That said, I do like their replace method as that could be useful. Also, each one of the Positional Linked List methods runs at O(1) which is impressive and definitely better in terms of general performance than our implementation.This is because of their Poistion class which stores exactly where an element is - sort of like an array. This makes access much quicker than ours, where we had to use .next and .previous and an index to go through our list and access the desired nodes.