Manaswi Alagani

Write-Up: Project 2

Performance Analysis

The *__init__* function for the Node class is constant time (O(1)). This is because the function is only setting the values of self.val, self.prev, and self.next, and does not need to iterate through a list. It is just the constructor for the class and will take the same amount of time to set these three values for any size linked list. The *__init__* function for the Linked List class is also constant time (O(1)) because it is only setting values for self.header, self.trailer, self.header.next, self.trailer.prev, and self.size.

The *append* function is constant time (O(1)). This is because the code will always start from the trailer because the linked list is doubly-linked. If it was singly-linked, the code would have a current value that starts at the head, moving through the nodes to get to the end and add the new node. However, since we can start from the trailer, and we move the node right before the trailer no matter the size of the linked list, the append function is constant.

The *insert_element_at* function is linear time (O(n)). Although the current pointer (which moves through the nodes) can start at either the header or trailer, it will have to move a certain amount of nodes to insert the element at a specific index. The amount of nodes that the current moves through would depend on the size of the linked list. As the linked list grows, the more nodes the current pointer needs to move through and the longer the code takes to insert the value, making the function constant time.

The *remove_at* function is also linear time (O(n)). This would be for the same reason that the *insert_element_at* is linear time. To reach the node that the function want to remove, the current pointer needs to travel a certain amount of indices which would change as the size of the linked list increases.

The *get_element_at* function is also in linear time (O(n)). Travelling through a varying amount of nodes due to the changes in the size of the linked list and the index that the user wants removed will cause the performance of the code to be linear. (The performance of this function is also why we don't use it in the iterator function).

The *rotate_left* function is in constant time (O(1)). This is because the first node will always be moved to right before the trailer and the second node becomes the new first node. It does not matter how many nodes are between this first and last node; the node in the front will always be the only one to move and the code will never have to keep moving a current pointer through the code.

The *__str__* function is in linear time (O(n)). This is because this function is walking through the linked list and storing each value in a separate list (which is ultimately made back into a string). The action of walking through values and storing them will take longer as the linked list increases because there are more values that the current pointer needs to walk through and store values of.

The *__iter__* function is in constant time (O(1)). This is because the function is just storing values (which is used in the *__next__* function). The assignment of values self.__iter_index and self.__current takes the same amount of time no matter how long the linked list is.

The *__next__* function is in linear time (O(n)). This is because the function is used to iterate through a list. As the length of the linked list used in the for loop increases, the longer it takes for the __next__ function to iterate through the function. The self.__current=

self.__current.next will be used more often and it will take longer for self.__iter_index to equal self.__size.

The **Josephus** function was in linear time because although it contained functions rotate_left and remove_element_at, they are both constant. The reason remove_element_at would be constant in this scenario is because it is always removing the value in the 0 index, no matter the length of the list. The entire Josephus function is still linear because as the length of the linked list in the for loop increases, the longer the performance time is.

Test Cases

The test cases I performed implemented every function created in the linked list class. I tested my functions on two linked lists: one that contained elements and one that stayed empty . Then I used my **append** function to populate one of the linked lists, so I can test the other functions in the linked list class. I made sure to include negative and positive values when appending my linked list, so that I know my code is not reading values as indices and raising an IndexError. After appending the list, I printed the original list and length (using the **__str__** and **__len__** function respectively), so that I can track changes. I knew my __str__ and __len__ functions were reliable (at least for the append function) because of the correct output.

I then used the **insert_at_element** with a negative value and valid index, and again, used the __str__ and __len__ function to make sure the linked list and function was being updated. Again, the correct outputs were given. I repeated the same process for valid indices for the functions of **__remove__at__element__** and **get__element__at**, and I was given correct outputs with updated changes in the string version of the linked list and length of the linked list.

I also checked the **_rotate_left** function on both the lists. I knew it gave me a correct output for the populated linked list because when I printed the string version using the __str__ function, it moved the first valued node before the trailer node. It also gave me a correct output for the empty linked list when it raised an index error because the size was 0 which did not all the code to implement the rotate_left function.

I also checked the output of invalid indices by putting invalid indices in the parameters of the functions I called. I made sure to use negative indices and indices that were larger than the size of the linked list, so I know both of these incorrect inputs were taken care of. In each function I called to use incorrect indices, the correct output was given (which was the print statement in the except IndexError block). I also used the __str__ and __len__ function to make sure that the linked lists were not being incorrectly updated (even if an IndexError was raised). The outputs showed that the linked lists were left unchanged.

Finally, I checked the **__iter__** and **__next__** functions using the for loop ("for val in list1). I knew that it worked correctly because each element was collected and printed in a separate line, which meant that the iterator had gone through every node in the linked list.

My Approach v. Textbook Approach

In general, my code is more detailed than the code in the textbook. However, the basic structure is very similar.

Both approaches rely on the .next and .prev functions to move through the list instead of indices. Because of this, the init method for the Linked List class is constructed the same way as the textbook. The important objects such as header, trailer, header.next, trailer.prev and size are initialized. However, my code varies when it comes to the init method for the inner Node class. The textbook's inner Node class takes three arguments (val, prev, next) while my init method class takes one (val). When the textbook makes prev and next arguments, the init method gives

you the option to choose the prev and next before executing the other functions. However, in my init method of my Node class, I do not allow for that option, and have the two prev and next arguments set to None. Because of this current pointer would always either start from the header ot the trailer.

This difference plays a large role in the condensing the code for the textbook. In the textbook, there is not as many lines of code necessary to move throughout the linked list because you do not have to start from the header or trailer; you are given a direct pointer to the successor and predecessor (like in the arguments of insert_node). This increases the efficiency to constant time for both delete_node and insert_node, since they do not have to iterate through multiple nodes of the linked list to get to the desired node. This makes the textbook code more efficient (better worst case performance) than my code.