Overview:

A doubly linked list is comprised of one or more nodes (each storing data), which are linked together, by each node having a previous, and next attribute. In the example of this project, we were tasked with making a doubly-linked list, which adds two sentinel nodes known as “header” which is located at the “head” or front of the list and “trailer” which is located at the “tail” or end of the list respectively. These two nodes which do not have any data stored within them, are utilized to help with reducing any special cases which a singly linked-list struggles with, in addition to helping keep the overall runtime reduced in certain instances which is discussed in the performance portion of this writeup.

Test Cases:

With regards to the testing of my linked list, I first had to create an instance named “dllist” and assign this to the Linked_List class. Following this, I began testing the functionality by invoking all of the class methods.

To start off, I wanted to make sure the append_element, __str__, __next__ and __iter__ methods worked appropriately, in addition to the formatting being appropriate (“[]”). So I created dllist.append_element with nine different elements, and had them print out in a list. Everything so far looked as though it worked without any issues, so I then made moved on to finding elements stored in the link list.

In order to accomplish this, I created a section with both positive and negative values being passed in, to make sure that the class method raised index errors when appropriate. Additionally, I wanted to make sure that the indexing worked and was going to the right location and returning the appropriate results. So I created seven dllist.get_element_at instances, two with out of scope provocations, and five that are within the list. The two out of scope raised appropriate errors and stopped the program from continuing (as it is implemented in the Python language).

The next thing I tested was to make sure the inserting of elements worked appropriately, and stored the passed value at the passed index. In order to make sure everything was in order, I made eight instances of dllist.insert_element_at where two were out of scope, and the remaining six were within the list. The out of order raised the appropriate errors, while the remaining six inserted without any issues.

The next thing I wanted to try, was making sure the rotate_left worked appropriately. In order to accomplish this, I made three instances of dllist.rotate_left, and printed out the results. As with the previous instances, the rotate worked and moved the element without any issues. Next I wanted to check the remove_element_at function, so I made five test instances, where two of them were out of scope, while three of them were within the range, and then printed the results.

The next thing was for me to check and make sure the classes were incrementing and decrementing appropriately. So after the adding and removing, I called the dllist.__len__ function, in making sure it returned the appropriate amount of elements. It did so without any issues, and returned the appropriate amount of nodes stored in the list.

Analyzing Time Performance:

Linked_List.py:

def append_element(self, val) [O(l)]: Because the append_element alters the bi-directional arrows (next, previous), so that the node being created points to the previous node and the trailer, the performance for this method is constant. The order as to which the arrows link is explicitly important, where the newly created node's previous arrow is pointing towards the previous node, and the next arrow is pointing toward the trailer. Additionally, you must also make the previous node's next arrow, and the trailer's previous arrows point towards the newly created node in order for the links to work and have a non-broken list.

def insert_element_at(self, val, index) [O(n)]: Due to insert_element_at needing to find an index in the list to add the value to this list, the performance for this method is linear. To accomplish this, the function utilizes a current walk, starting from the next node of the header, and goes until it finds the index. It raises an error if the index pass in is out of scope.

def remove_element_at(self, index) [O(n)]: Just as the insert element, the remove_element_at requires finding an index. For this reason, it is linear performance, as it uses a current-walk to find the index, remove the node and link the nodes before and after the removed node together. It raises an index error if the index passed is out of scope.

def get_element_at(self, index) [O(n)]: Just like the remove_element_at and insert_element_at, the get_element_at, requires finding an index in the list. For this reason, it is linear performance, as it uses a current-walk to find the index and return the value stored at that index. It raises an index error if the index passed in is out of scope.

def rotate_left(self) [O(l)]: The rotate_left method is constant, as the indexes of three nodes are shifted.

def __str__(self) [O(n)]: The __str__ method's performance is affected by the size of the linked list. Every time the method is called, the python interpreter goes through all of the stored data in the link-list, and appends it into a python array("list") as a string attribute, it then returns the list in a proper format for the interpreter to display on the screen.

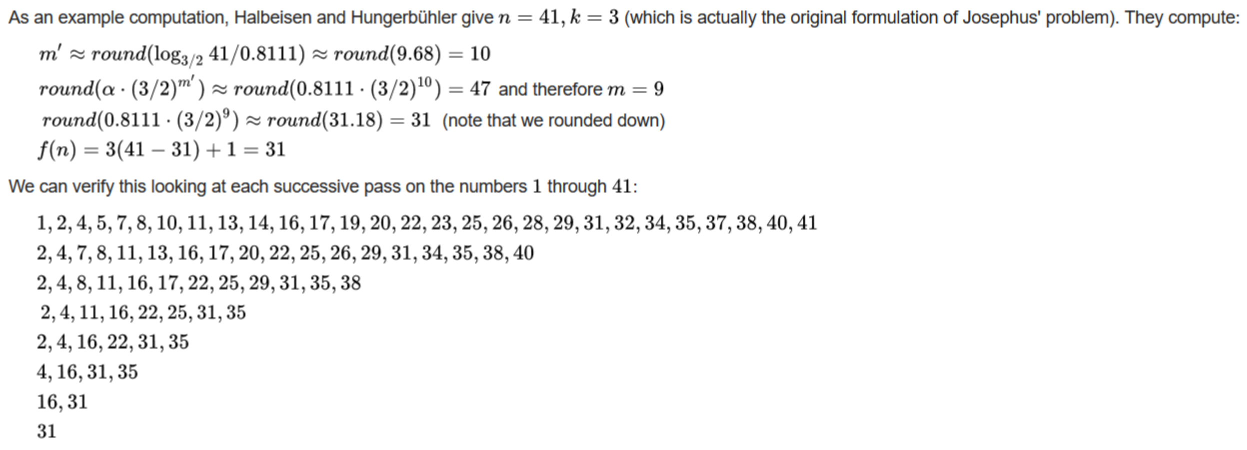
def __iter__(self) and **def __next__(self) [O(l)]:** In order to go through the entire list would technically be considered a linear performance. But, due to these methods only shifting one element of the linked list, it is constant time.

def __len__ [O(l)]: This method is constant, as the data length are stored in the constructor __init__. Because __len__ only returns the self.__count of this constructor, it is of constant time-performance.

Josephus.py:

def Josephus(ll) [O(n)]: This method initially asks the user how many “people” (elements) they wish to store in the linked list. From here, the method fills the list beginning with one (1) and going until the linked list is filled with the appropriate amount of people (O(n) performance). From here, the Josephus method then begins killing off (removing) one node at a time, per cycle, until the last person (node) is left alive (not erased). For this reason, the overall performance of the method is linear (O(n)), as the cycle kills off one person at a time.

To prove this, and that the project works, you can take the equation which is provided by the Wikipedia page regarding this problem, using n=input provided by the user, you will find the solution to be the same.



As an example computation, Halbeisen and Hungerbühler give $n = 41, k = 3$ (which is actually the original formulation of Josephus' problem). They compute:

$m' \approx round(\log_{3/2} 41/0.8111) \approx round(9.68) = 10$

$round(\alpha \cdot (3/2)^{m'}) \approx round(0.8111 \cdot (3/2)^{10}) = 47$ and therefore $m = 9$

$round(0.8111 \cdot (3/2)^{9}) \approx round(31.18) = 31$ (note that we rounded down)

$f(n) = 3(41 - 31) + 1 = 31$

We can verify this looking at each successive pass on the numbers 1 through 41:

$1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 28, 29, 31, 32, 34, 35, 37, 38, 40, 41$

$2, 4, 7, 8, 11, 13, 16, 17, 20, 22, 25, 26, 29, 31, 34, 35, 38, 40$

$2, 4, 8, 11, 16, 17, 22, 25, 29, 31, 35, 38$

$2, 4, 11, 16, 22, 25, 31, 35$

$2, 4, 16, 22, 31, 35$

$4, 16, 31, 35$

$16, 31$

31

Source: https://en.wikipedia.org/wiki/Josephus_problem

Us vs Book:

If I understand correctly, the book utilizes similar functionalities to the class methods which we use, such as __len__, __init__, _Node, and that is about it. However, instead of to constantly walk throughout the entire list to insert a value, they utilize an “insert_between” method, which inserts a node between the predecessor and successor. Additionally, in the constructor, it keeps a function is_empty to check and see if the list is empty. They also use a delete_node method which is similar to our remove_element_at method. Their biggest advantages to the methods we use, have to deal with enqueueing and dequeueing. Because those are at the beginning and end of the linked list, it can be done in constant time, whereas we have to traverse to the beginning or end of the list, causing it to be linear. Furthermore, they require that the user knows which indices are needed to put a new node, where we simply insert it into our linked list using the insert_element_at method.

Summary:

In closing, this project has taught me to better understand the linked list structure, how it works, the various time-performance which each method utilizes in accomplishing their programmed task. Additionally, it has also helped gain better insight into how real world implications can be used, by storing data into various structures (as seen with Josephus).