CSCI 241 Data Structures
Project 2: Literally Loving Linked Lists LOL

The following project has two python files: Linked_List.py and Josephus.py. In Linked_List.py, I implemented a sentineled doubly-linked list with several functions providing manipulation capacities. The Josephus.py file implements these methods in order to solve the Josephus problem.

Explaining the Code

Our data structure includes an outer class, Linked_ List, and a private inner class, __Node. The Linked List class will use objects in the Node class; the Node class must exist inside the Linked List class because a node can only exist inside a linked list. The Node class initiates data, values, elements, and references (next and previous). Although this class name may seem like it is responsible for creating nodes, it is simply representing a node in the linked list and storing referential information. Our outer linked list class is responsible for creating nodes, connecting them, and manipulating them.

def __init__(self)

We begin with the init function of the Linked List; this initiates the list by creating a head, a tail, and a reference between the two. In this particular Linked List our head and tail nodes do not carry variables; thus we establish them simply as a reference point. In order to make sure that we don't count these two nodes in our linked list, we establish the size as zero. An empty linked list would simply be a double link between the head and tail nodes.

def __len__(self)

Our first method that describes our linked list is the length function. This function returns the number of values stored in the Linked List. Why is the length function necessary when python has a built in len() function? The len() function works by returning the __len__ method. Our code shows that this method returns the size of the linked list, which we keep track of in our other methods. This method has a constant runtime [O(1)] because the interpreter only needs to look up a number.

def __iter__(self)

Our second method is a reference to where our loops should begin running. Because our loops should begin on our first node, that would mean that we need to point to the node directly after the head. This method is also constant runtime, as it will always point to one node regardless of how many there are. This function is assigning a variable, thus, the amount of nodes is irrelevant to the runtime [O(1)].

def __next__(self)

The next method works in conjunction with the iter method. While the iter method initiates a loop, next will end a loop. In the first two lines of this method, we see that if the pointer is pointing at the tail, then we can stop the iterations, as there is nothing else to go through, and the tail holds no value. If we are not at the end of the linked list however, the value of the node that the pointer is pointing to needs to be stored in a temporary variable. Because each node has a next value with the exception of the last node, we can "refresh" our temporary variable while the exception doesn't apply. Thus, the pointer variable is updated to the next node. Because next function is simply reassigning its variables, it is constant time [O(1)].

def append_element (self, val)

The append method is the first method that manipulates the nodes itself. This method adds a value to the end of the linked list regardless of its size. The general algorithm for appending an element is such:

1. A new node with an assigned value is created.
2. A "current" label is established on the first node in the linked list
3. Shift the "current" label to the last node
4. Add node to the end of the list by reassigning the links
5. Increase the size of the linked list by one

A very important part of the fourth point in the algorithm is the order in which one reassigns links. If the wrong order is established, we won't be able to reference certain nodes in the linked list since there is no indexing tool. The only parts of the list with references are the head, the tail, and the current node. Thus, every node can only be called on in reference to these three. This general algorithm applies for the majority of the cases in which we need to append to a linked list. However, there is one case in which the references are different, as it unnecessary to establish a current labeled node. If the linked list is empty, we can manipulate the links using references to the head and tail nodes singularly. Thus, we create this as our only exception and reassign the links. The append method carries one loop which relies on the length of the list. Since it must run through every single node, it has a linear runtime [O(n)].

def insert_element_at (self, val, index)

Our next method also manipulates the linked list by adding a node in a particular index. However, there are no indices in linked lists, so we must find a way to travel through each node until we reach the desired spot. Because we must also insert an index for this method, we can run into three special cases that can produce an IndexError. What happens if the user inputs an index which is bigger than the size, smaller than the size, or is at the end of the list? We must also remember that our indexing range begins counting from 0 as the first node; thus, the maximum index will be the length -1. Because our indexing is out of bounds or does not work with the method, we introduce an IndexError. In the append method, we also discussed the difference in establishing a node next to the head or tail nodes. Thus, we must also make an exception for when the user inputs the 0 index, reassign the links with the head references, and increase the size by one.

For the majority of cases, our algorithm will be the same. There are two similar algorithms that could have been used to insert an element. However, not all codes are the same- when coding, one must choose the most efficient and concise algorithm. Because we have two references already established in our linked list before we add in any labels, we can use these to our advantage. If we split the code in two, the labels need to cycle through less nodes in order to find their "index". Thus, I implemented two cases. Because a linked list with an odd number of nodes could be problematic, I chose to use the floor function in the math library. Although this could have been done differently, I believe this is the most concise way to go about splitting an odd list. The first case will have one less node than the second case. The first case addresses the first half of the list; the algorithm assigns a current label to the first node and cycles this to the node before the desired index. Then, the links are reassigned. The range is very important while moving the current label; one more iteration or one less could incorrectly move the current label. Lastly, the size is increased by one. For the second case, our nodes are referenced by the tail. Thus, we establish our current label as the last node. We must move the label backwards through the list; this means that rather than moving it forward a specific number of indices, we must subtract the size by the index to move the inverse amount of nodes. Then, the links are reassigned and the size is increased by one. The runtime for insert_element is linear; however, in its worst case, it only has to run through half of the list. Thus, the runtime is O(n/2).

def remove_element_at (self, index)

Removing an element is very similar to indexing one. Because an index is put into the method by the user, we must again account for out of bounds errors. Next, we must account for cases in which we are removing the first node or the last node. When removing the first node, we set the current label to the first node and reassign the links and subtract one from the list. When removing the last node, we set the current label to the last node and reassign the links and subtract one from the list. For the majority of cases, we once again must divide the list into two as we did in the insert method, move the current label, reassign links, and subtract one from the size. The only difference in the reassigning methods between insert and removal is that the next and previous links of the node we are removing will reference None. Because this method splits the linked list in two, the worst case for this method would be removing an element in the middle of the list. Thus, the runtime is linear O(n/2).

def get_element_at (self, index)

The get element method will serve to find us the value at a particular position in the linked list. Because we ask the user to insert an "index", we must check that the input index is within bounds. If it is not, we once again introduce an IndexError. Because we are once again looking for a position of a particular node, we can split the linked list in half again. The same strategy as the previous methods are applied. No changes are made to the size nor the links; the value of the current node at the particular index is returned. This method employs one loop in its worse case, and only has to go through size/2 nodes. Thus, the runtime is linear O(n/2).

def rotate_left(self)

The rotate left method seems very difficult in theory, yet is a matter of removing the first node by unlinking it and reassigning the head links to the second node, and then appending it to the end manually. One cannot rotate a list with no nodes, so we introduce a first case with an error as a pop up. For all other sizes, the algorithm is applied. Although in theory one could try to call the removal and append methods to have a similar effect, this reduces the time, increases efficiency, and makes the code easier to use. The runtime is constant [O(1)], as the size of the list does not impact the performance.

def __str__(self)

The __str__ method returns the linked list as a string. This somehow affects the print function; when testing my code, my original algorithm would crash whenever I called on the print function. The algorithm to this is quite simple. If the linked list is empty, then we can return an empty string with brackets. If it is not, a current label is applied to the first node, and an empty string is created. Next, we can move the current label down the linked list as long as it does not reach the tail node, and the value of the nodes are added to the list along with a comma. Because this would mean that the last node in the list would have a comma and space after it, the string must be chopped by two at the end, and two brackets need to be added around it instead. This method's time does depend on the length of the list, and thus it is linear time [O(n)]. In it's worse case, it must run until the end of the list.

Testing the Methods

How can one be sure that each method works for every possible linked list? As a math major, I truly don't believe that it is possible for me to be sure that my code works for all of them, as trying a few examples does not assure that the code won't crash somewhere. However, trying as many examples as I could think of did show some errors in the code that needed to be fixed. The test code I submitted did not nearly cover the extensive amount of examples I tried due to some failure with my python interpreter and the try/except statements. The first examples I tried were all very basic; I did not want to test out the exception cases until I could assure that the generic cases worked. Thus, I appended several elements into the linked list including a string and float. I checked that the lengths of the list matched the amount of objects I had appended, and that they were all in the correct order. I inserted objects on both ends of the lists and verified the total length and placement. I did find some errors in my indexing for loops here; the ranges were sometimes off by one. Other errors I found were attribute errors due to reassigning linking in the wrong order. Thus, the appropriate changes were made. I was able to get every element from the linked list with no issue, and could remove from all points of the list.

After I tested my general cases, I moved on to testing all my exceptions. Unfortunately, my try/except functions never worked; however, by using the print function, I was able to verify when Index Errors would pop up. Another thing I implemented but took out for clarity is a print function at the end of every method I had to split into two. The remnants of these tests can be found quoted out in my remove method. By inserting a print statement into my cases, I could see which parts of my code were executing. My method for testing my exceptions was to try anything that might overlap between if statements and testing each conditional statement. I did in fact find overlaps between my if statements; although I added more conditions to them, in retrospect, the overlaps probably occurred because I did not use elif statements nor stopped the algorithm from continuing to check all the other conditionals. My syntactical understanding of python is a little bit limited, and thus my code may have suffered due to that. Although we were to use a similar testing algorithm as Set.py, I would have liked to implement an entire different algorithm in which I could quickly check hundreds of different inputs and different combinations of method orders. Lastly, I did check my iterator, as I wanted to make sure that it was cycling through all the nodes.

The Josephus Problem

The Algorithm provided to solve the Josephus Problem is quite smart; because we need to "kill off" every other person, this strategy both keeps order in mind while avoiding problems one may have with odd vs even starting numbers. By rotating left, one is always skipped. However, this algorithm doesn't account for one problem; if the first element is removed and the second is skipped, the final survivor is different than if the first element is skipped and the second is removed. Thus, for a case with 100 people, the survivor could be 72 or 73 depending on which skipping point is picked. The Josephus algorithm could also be edited to skip more than one person depending on the amount of times the rotate method is iterated. Because the algorithm takes more time depending on how many people there are, it is a linear time [O[n]]. The more people, the more loops.

Positional List vs Project Algorithm

Both Linked List Algorithms have nested classes, but much diverges from there. The Positional List Algorithm has better readability, less repetitiveness, and is faster. Looking through the code, one first notices that there are a lot more methods in the positional list, but they are much shorter than the project algorithm. All positional list methods run in constant time, while the project algorithm has a mixture of constant time and linear time due to the amount of loops throughout the code. Thus, the positional list code is a much more streamlined version of our project. The positional list method focuses on the locations of elements within a list. This lets the user input more positional data, which the algorithm can actually interpret. Although there are more methods, they define more positions in order to avoid the repetitiveness next and prev can have. For example, the positional list code introduces first, last, before, and after accessors that increase the amount of references in the list. Our project code's structure relies mostly on its mutators and constructors (with a few accessors), while the positional list focuses more on accessors and utility methods. Rather than having a Node class, the positional list algorithm has a positional class which represents the locations of elements within a list. Some of these methods return Boolean True or False when tested. While our project relied on repetitive conditional statements, the positional lists code has a validate function which verifies that a position is valid. Another major difference is that the positional list algorithm does not completely disregard a node that is removed from a list as our project code does. Thus, it can still be referenced and added in separately. Perhaps the oddest difference in my perspective is the use of methods inside methods. For example, in the positional list code, the insert_between method is used in a lot of the other mutators. Overall, it seems like the positional list algorithm is better suited for managing doubley linked lists.