Ignat Miagkov

CSCI 241

March 6 2020

Project 2 Write-Up: Literally Loving Linked Lists

For project 2, we were asked to create our own implementation of a sentineled doubly linked list. In this write-up, I will discuss each method present in my implementation and compare it to the methods presented in the textbook. I will compare performance for both the methods in my implementation, as well as the textbooks. Finally, I will describe my solution for the Josephus problem using my implementation of my Linked List class, also detailing the worst-case run time and performance for this problem.

def __init__(self, val) [Node]

This method initializes a node object. The node will have three different attributes: value, previous, and next. Value stores the current value in the node, while next and pervious reference the next and previous nodes in the list, respectively. My implementation is a bit different from the textbook, as I only include one parameter for creating the node (value), while automatically setting next and previous to none, and then changing the references to next and previous within the methods of the implementation. The text decides to initialize nodes with the next and previous references within the parameters, which does not make sense in my opinion, since we will never initialize a node with its references as parameters, we will adjust the references on the fly. The run time for my implementation and the textbook's is O(1), as it will always take a constant amount of time to create the attributes for the node objects and will not change depending on what is being initialized.

def __init__(self) [Linked List]

We initialize two new private nodes: header and trailer. Due to the nature of the node constructor, they point to none when first created, but then we set them to point at each other: header's next attribute points to trailer, while trailer's previous attribute points to header. We also initialize a private size attribute to track the length of our linked list. Textbook implementation is similar; however, its node class is slightly different from ours, thus uses extra parameters in initializing the header and trailer nodes. Performance for both methods is O(1), as creating the first linked list object will always be constant time.

def __len__(self)

Return the size of the list. Constant time, returns the same attribute with every call.

def append_element(self, val)

Inserts new element at the end of the list. Textbook has no append function, as it uses insert_element_between for all insertions. Additionally, insert element in the textbook has to include the two nodes you want to place in between of in the parameters, which does not make sense, as the point is that we do not have the specific objects stored at a defined location, but rather of chain of references between them. Anyways, append takes constant time to complete, as it will always first refer to the trailer node, a private attribute to the linked list, and then add a node right before it.

def insert_element_at(self, val, index)

Inserts element at given index. Does so by traversing to the position in the list in which the new element should be inserted and changes the references between nodes such that no node is lost in the process. Textbook implementation is very different: the nodes between one should place the new node have to be given in the parameters. However, in both cases, inserting an element at a given position is a constant run time method: we do not have to shift the position of each subsequent element. Using a linked list, we must change the references around to allow for the insert, thus taking constant time. This is one major advantage of our linked list implementation: since we are only storing references to other nodes, nodes that are not involved in the insertion do not have to be shifted down the list, thus having performance equivalent to O(1).

def remove_element_at(self, index)

Removes element at given index. Textbook implementation requires that the node to be deleted should be in the parameters. This does not make sense because we do not have the specific node stored, just the references to that node through other nodes in our list. In my implementation, user passes in an index. Like insertion in a linked list, removal only requires changing the references of the nodes around the element that is to be removed, thus taking constant time. Although we must traverse to a certain index, nothing in the while loop raises our run time to linear. The only part that impacts the run time of this method is the actual removal of the node we wish to remove, thus having O(1).

def get_element_at(self, index)

Retrieves element at given index. This is where the linked list structure falls short of an array. Textbook does not have an implementation of the get element method. The problem with the linked list approach to the retrieval without removal is the fact that the nodes are linked together by references, not by a physical location. The run time of the get element method is O(n), as we must hit each node in the list before we get to the node we desire to obtain. In an array, an index represents a physical location in memory, and the user can access it directly, thus having a constant run time. In a linked list, we must traverse through each reference to find the index we are looking for. Even if starting at either header or trailer (whichever is closest), it will still take at least O(n/2), which rounds up to O(n)

def rotate_left()

If this was a singly linked list, then the run time of this method would be linear. However, since we have a doubly linked list, we can take advantage of the fact that we can reference nodes in the previous direction. The size of our list does not impact the run time of rotating the list one element to the left. The only thing we have to worry is about changing our references of the first element in the list: instead of pointing to header in the previous direction, it must now point to trailer in the next direction, as well as update other points to make the list complete. This will always take the same amount of time, since we have private header and trailer attributes. The run time of this method is constant, or O(1).

def __str__(self)

This method prints out the contents of our linked list as a string. The run time of this method is linear, as it is dependent on the number of nodes in our list. We must visit every single node and get the element at every node and add it to the string. Since it is directly dependent on the size of the list, the run time of this method is linear, or O(n).

def __iter__(self)

Creates and returns an iteration index. Just creates an integer, constant run time.

def __next__(self)

The calling of next invokes the get element method mentioned earlier. Due to this, the run time of this method will be linear, as it uses a different method that is already linear time. Calling next accesses the element at the index of the iterator, and returns it. Next has performance of O(n) as it will traverse through all the values using the get function.

Test Cases

Firstly, I create a new Linked list and try to append several elements to the list. If that works, then I print out the list and length of the list to determine that the length function is working properly, as well as the __str__ method. Next, I try to use insert element on an index in the middle of the list and then I print the list and length again, to ensure that the element was added to the appropriate spot and that the size was updated. Then, I try to use insert element on an index that is not in the list, and it should spit out an error message for the user. I follow this up by attempting to add to an index right after the end of the list: this should also spit out an error message, as only append should be used to add to the end of the list. I then try to use my remove methods, once somewhere in the middle of the list, once at an index out of bounds, once at the beginning, and once at the end, tracking and printing the list and its size with every run through. The only call that should spit out an error message is the remove with an index outside of the current list. I then add some more elements with both append and insert to ensure that both still work correctly. I then try to use get element at some index in the list and will print out the element at the index if worked properly. I then print the list and length again to ensure that the get element method did not tinker with the actual contents of the list. I try to use get element again, but this time with an index that is out of bounds, and an error message should be pushed to the user. I then use rotate left and check it with my previous list to make sure it works properly. Finally, I test my iterator by creating a loop that traverses through all the values in the list, printing each element on a new line with each run of the loop. The elements should be printed in the same order as the list that I checked right after using rotate left on it. The most important part of the test cases was to print out the list and its size each time I tried to use a new method, as it allows me to check that none of the content were lost inappropriately. This pattern of testing allows for silent errors to not be ignored: they are returned every time the user does something that the method or the developer did not expect him to do. By testing our implementation as such, we make sure that we have accounted for all possible scenarios, and as developers, determine what the user can do with our implementation.

Josephus Problem

In the Josephus problem, every second element of the list should be removed as if the list was circular. So, to solve for our list not being circular, we rotate left one position and then remove the element that is at the beginning of the list. This process is repeated until only one element is left in the list. The surviving element is then printed to the terminal. To initially create our Josephus problem, we prompt the user to determine how many people will be in our problem. We then append elements 1 through n+1, where n is the number of initial survivors prompted by the user, into a newly created linked list. The list is then passed into our Josephus method, and with each iteration of the while loop, the resulting survivors are printed. The performance of the Josephus problem is O(n). Our removal of the appropriate node takes constant time, as well as our rotate left method. The only thing that is hindering our performance is the while loop, as the termination of our while loop depends on the initial length of our list and the amount of times we had to remove an element. Since our implementation of the Josephus problem only depends on the length of the initial list, it hsas a linear run time.