Programming Assignment 1 - Commentary

Parker Knight

Computational Complexity of Implemented Methods

I will organize the complexity analysis of each method by class, starting with the Node class.

Node

All of the methods in the Node class run in $O(1)$ time. The "setData" method merely saves the input string to the Node's private "data" variable (which is independent of the length of the string), and all the constructor does is call this "setData" method. The "getData" method simply returns the "data" field, so it is clearly $O(1)$.

LinkedList

The private "get" method for the LinkedList class is $O(n)$ where n denotes the length of the list. This is because, in the worst case, the method will need to iterate through the entire list the find the node at the desired index. The LinkedList constructor is $O(1)$, as all it does is initiate the class's fields. On the other hand, the destructor for the LinkedList is $O(n)$, with n being the length of the list. This is because the destructor needs to iterate through the list and delete each node individually.

The "insert" and "delete" methods are also $O(n)$, as they rely on the "get" method described above to insert or delete nodes. However, "insertEnd" actually runs in $O(1)$ time, because it uses the LinkedList's tail pointer to insert a node at the end of the list. This operation is independent of the length of the list, giving it a constant run time. "search" and "edit" also both run in $O(n)$. The "search" methods iterates through the entire list to find each instance of the desired string, so its runtime is linear with respect to length. The "edit" method relies on the "get" helper method, so it too is $O(n)$.

Finally, the "print" method is $O(n)$ as well, because it iterates through each node in the list to print their contents.

Editor

The constructor for the Editor class is $O(1)$, as all it does is initiate the "running" variable, as well as its underlying LinkedList. The Editor's destructor is $O(n)$ where n is the length of the list, as it calls the LinkedList's destructor (described above).

The "getRestOfInput" helper runs in $O(1)$ time, since all it calls is the standard library function "getline", and returns the resulting string. The "cleanString" helper method is $O(n)$, where n is the length of the input string. This is because the method iterates through each character in the string, checking whether or not it is a quotation mark.

Finally, the "interact" method runs in $O(m * n)$ time, where m represents the number of commands that the user passes to the program, and n represents the length of the list. This is because the least efficient runtime of all the LinkedList operations is $O(n)$ with respect to list length, so that is the worst case for the body of the while loop. If we perform this loop m times, we find that the worst case runtime of "interact" is $O(m * n)$.

Analysis of the Use of Linked Lists

Advantages

The main advantage of using a Linked List as the implementation data structure for our line editor is that appending lines to the end of the document can be accomplished in $O(1)$ time. This makes the creation of new a document very efficient, as the user is typically appending several lines in succession.

Disadvantages

The primary disadvantage of the Linked List is that accessing and deleting elements is done in $O(n)$ time. This means that editing lines in the middle of the document can be expensive, especially when compared to an array. This could become a problem for the user if they were making lots of revisions to a particular document.

What I Learned, and What I Would Do Differently

After completing this project, I feel as though I've developed a strong understanding of the Linked List data structure and its numerous applications. I'd used linked lists in previous classes, but the problems that we had to solve were much more trivial. Working on this line editor project (as well as several exercises on Stepik) has forced me to really practice working with and thinking in terms of

linked lists. I feel relatively comfortable handling any exam or interview question regarding the subject thrown my way.

If I could redo this project, one thing that I would do differently is that I would probably use a doubly-linked list, rather than a singly-linked implementation. While doubly-linked lists can more be complicated to deal with, having a pointer to the previous node would have made certain operations more efficient. For example, when deleting a line, it is necessary to keep track of the previous node so that we can "reattach" the linked list. In my current implementation, I accomplish this by using my "get" helper function, which runs in $O(n)$ time. With a doubly-linked list, accessing the previous node could be done in $O(1)$ time. Similar situations come up at other points in the code as well, like in the "insert" method.