COP3530

Claire Seiler

Programming Assignment #1: Line Editor

9/18/2018

Commentary

1. What is the computational complexity of the methods in the implementation?
   With my current implementation, the functions have the following worst case computational complexity:
   - **insertEnd – O(n)**
     "insertEnd" requires traversing the list until the last position is reached, and then a new node is created and inserted. This will always have an O(n) complexity in my current implementation. If I had kept a pointer to the tail of the list, this could be reduced to O(1).
   - **insert - O(n)**
     "insert" requires traversing the list until the desired position is reached, and then a new node is created and inserted. It has an O(n) worst computational complexity, because in the worst case it will have to traverse the whole list to find the desired position. If it is inserting at the head, it would have a complexity of O(1). If my implementation included a tail, then inserting at the tail would also be O(1).
   - **delete – O(n)**
     "delete" requires traversing the list until the desired position is reached, and then a new node is created and inserted. It thus has an O(n) computational complexity, because in the worst case it will have to traverse the whole list to find the desired position. If it is deleting at the head, the complexity is O(1). If my implementation included a tail, then deleting at the tail would also be O(1).
   - **edit – O(n)**
     "edit" requires traversing the list until the desired position is reached, and then the data of the node is modified. It thus has an O(n) computational complexity, because in the worst case it will have to traverse the whole list to find the desired position. If it is editing at the head, the complexity is O(1). If my implementation included a tail, then editing at the tail would also be O(1).
   - **print – O(n)**
     "print" requires traversing the whole list. Thus, it has O(n) computational complexity.
   - **search – O(n)**
     "search" requires traversing the whole list to check if any nodes have text that matches the desired phrase. It thus has an O(n) computational complexity.
   - **quit – O(n)**

"quit" calls the deconstructor for the linked list, which traverses the whole list to delete every node. Thus, it has O(n) computational complexity. If it didn't call the deconstructor, it would have a complexity of O(1).

If I had adopted a different implementation, I could have potentially reduced the computational complexities of some of these functions. For example, if delete took a pointer to the node to be deleted, its computational complexity would become O(1). However, my current implementation requires traversing the list until the correct position is found and the corresponding node is deleted , an O(n) process.

2. Your thoughts on the use of linked lists for implementing a line editor. What are the advantages and disadvantages?

   Using a linked list has the advantage in that it is easy to implement and straightforward to maintain. If there are a lot of insertions/deletions at the beginning and end of a document, then a linked list is an okay choice for a line editor due to the O(1) complexity of inserting/deleting at the head and tail. However, if a lot of insertions/deletions in the middle or searches are required, a linked list is not the best choice. As most text/line editors would require a high number of these operations, linked lists are probably not the right choice in terms of computational complexity. A better choice would be an AVL tree. For example, AVL trees have O(logn) search, insertion, and deletion times, dependent on the number of nodes in the tree. Rotations are O(1) operations.

3. What did you learn from this assignment and what would you do differently if you had to start over?

   I learned some interesting techniques on input handling and parsing through this assignment, including how to flush cin to avoid unintended behavior or errors. If I had to start over, I would make a variety of changes to improve the computational complexity and elegance of my program. For instance, I would convert my singly linked list into a doubly linked list. A doubly linked list offers the ability to traverse forwards and backwards, which could potentially reduce access times. Furthermore, I would look into better ways to do input validation and parsing. My current implementation using the std::string class does handle buffer overflows and string truncation well, but it still can be abused if specifically crafted input is passed. For example, the following input **insertEnd " " quit** will cause the program to execute both the insertEnd command followed by the quit command, despite being on the same line. Also, if the user types in a command followed by a string with only one quote at the beginning, it will continue accepting input until a second quote is passed (e.g. **insertEnd "**hi). This requires the user to input a quote to effectively discard the malformed command. Given more time, I would have fixed this issue. I tried to consider most edge cases, but I found the more I tried to cover, the more potential edge cases I discovered.