Computational Complexity of Methods in the Implementation

```
insertEnd() : O(1) (constant)
insertAt() : O(n) (linear, uses findLine)
deleteLine() : O(n) (linear, uses findLine)
editLine() : O(n) (linear, uses findLine)
search() : O(n) (linear)
printDoc() : O(n) (linear)
findLine() : O(n) (linear)
getFront() : O(1) (constant)
getSize() : O(1) (constant)
isEmpty() : O(1) (constant)
```

Advantages and Disadvantages of Using a Linked List

The main advantage of using a line editor for this assignment is that most of the operations performed are insertions and deletions, so insertion and deletion is O(n). Also, because accessing most nodes is done sequentially since the user can't print a specific line, most access must go through each element anyways, which is an advantage of using a linked list. Using an array-based list would require moving each element after the node inserted or deleted, whereas with a linked list, the program just has to navigate to the node and then can delete it in constant time. The obvious disadvantage of using a linked list is that the program can't access a node in constant time, and because most user commands, with the exception of `insertEnd`, need to either utilize an index or go through each node, making the only user command that executes in constant time `insertEnd`. With an array-based list, `edit` could also be executed in constant time because it does not require insertions or deletions.

Lessons Learned

Although I already had a good amount of practice with linked lists prior to this project, because every implementation has different requirements, not all lists need to be implemented the same. I did have to comment out a significant amount of error handling, because I had already implemented it when we were told not to output any error statements. If I started the project over, I might implement a sort of iterator within my linked list that points to the current node and stores its index based on the current state of the list, and the index would be changed when the list is modified, etc., and doubly-link the list to allow faster average-case access of elements. This would allow me to modify the `findLine` method to not have it iterate through the entire list, and could also see if the iterator was closer to the requested index then the front or back, and dynamically choose which node to start from.

I would also add back in the output statements for error handling, because having a program that doesn't tell you when you input a command wrong can not only be confusing for the user, but also in testing it. If a command statement is not exactly right, for instance, by leaving off a quote or misspelling a command, the program currently just ignores it without telling the user that the command wasn't executed, which is not a very healthy program in my opinion. Originally, I had output statements to show the correct syntax of each command, if the keyword was valid but the arguments weren't, and gave the user the option to list syntax for valid commands if the keyword wasn't recognized. Although for the sake of testing the program, these aren't necessary, there should be some indication to the user that their command was not executed successfully.