|  |
| --- |
| Sheridan College Institute of Advanced Learning and Technology |
| Assignment 4 Report |
| Winter 2019-PROG23672 Data Structures and Algorithms |

|  |
| --- |
| Richard Perocho  4-7-2019 |

# Problem Definition

The problem addressed was online bank account storage. This particular case is an administrative level of access on bank accounts. The administrator is able to manipulate online accounts, updating balance, searching for specific accounts, and setting up new ones.

# Data Structure and Operations

The data structure used was modeled after linked lists, as it allows for easy insertion and removals of any given account. This application used double linked lists as it would make deletion much quicker than singly linked list application.

## Data Operations

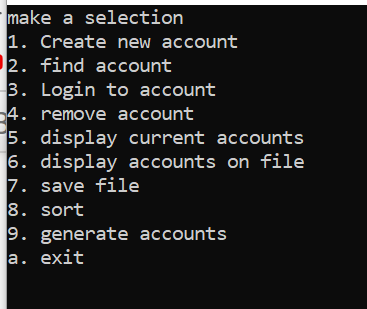
* **Add**: the data structure is able to add new accounts in O(1) time complexity. The adding is similar to that of a stack, where most recent elements are inserted at the front.
* **Retrieve**: accounts are retrievable in the data structure. Originally, I had implemented a balanced search tree algorithm that would re-structure the doubly linked list to be a balanced tree, however, I discovered that I would need to switch between the balanced search tree order and sorted order very frequently, therefore decided to remove it and choose to use a linear search instead. With the balanced search tree implementation, searching would only take O(h) worst case time complexity, where h is the height of the tree. However, converting from sorted to balanced search tree took O(n) time. Currently, worst case search time complexity is O(n) instead of O(h), however, there is no overhead of creating a search tree as it is a linear search.
* **Delete**: accounts are able to be removed from the list using the username as a key. Since all usernames are unique in the list, there will be no duplicate deletions. This operation is the reason I chose doubly linked lists, as it is easy to re-connect the list after removing the target node.
* **Save**: accounts are able to be saved to a text file in JSON format. This was done by opening the input/output streams to read and write from a text file saved in the resources folder of the project.
* **Sort**: Accounts in the list are able to be sorted based on the alphanumeric values of their usernames, in ascending order. One challenge I came across was comparing lowercase alpha values and upper-case alpha values. This is an unfair comparison because for instance: “a” > “Z”. I was able to solve this by creating a function (normalizeAscii) that takes the string and returns a version that has all lower-case letters, making the comparison fair. It does not change the original username, it only uses the modified version for comparisons. The sorting algorithm I chose is merge sort. This is because of its consistent time complexity of O(nlogn). The merge consisted of 3 parts: 1. Find the center of the subtree 2. Recursively create left and right subtrees 3. Recursively merge and sort subtrees. At the end of the sort, the head node is returned.

## Account Operations

In this application, I created functionality that allowed for administrative operations like logging into accounts, updating balance and changing password. These operations are possible because the data each Node in the data structure carries is a UserAccount object, that have getters and setters for its properties. In the application, there is also a function that generates accounts for testing.

# Application

Main menu (available operation)



|  |  |
| --- | --- |
| **Before Sorting** | **After Sorting** |
|  |  |

|  |
| --- |
| **Logging in** |
|  |

|  |
| --- |
| **Saving to file** |
|  |

|  |
| --- |
| **Deleting account** |
|  |

# Reflection

I believe double linked lists and merge sort were the best options for this specific application. Another sorting algorithm I could have used is quicksort, however, its worst-case is O(n^2), unlike merge sort’s worst case O(nlogn). One advantage of using quicksort, however, is that its space complexity worst-case is O(logn) compared to merge sort’s space complexity worst case of O(n). This factor should be considered in the future. I would choose quicksort if storage was limited. One algorithm I considered was Heapsort, because its time complexity worst-case is O(nlogn) similar to merge sort. Not only that, but heapsort’s space complexity is a linear O(1). However, I realized that implementing heapsort on a doubly-linked list is very difficult, since heapsort relies on indexing to be able to traverse the data structure. Heapsort would be better to implement on array-based structures instead. This assignment was much more challenging than others before it. I could have improved it in many aspects, for instance, properly implementing the BST such that I would not need to switch between BST and sorted-list forms would have made searching elements much quicker. I also believe I could have implemented heapsort to an extent. Since I was able to create a balanced tree from the doubly linked list, I may have been able to use that to implement methods such as heapify, since the data structure was transformed into a tree. This would have made deletion and search much faster, since their worst case time complexities would only be O(h) h=tree height. Overall, I am satisfied with the implementation, and I was able to learn a lot from this assignment.