Benny's Bank

Prepared for: Data Structures and Algorithms

Prepared by: Benjamin Sykes, Liam Stickney

March 12, 2019

THE PROBLEM

We will be designing a console-based application that allows the user(ie: the bank) to store bank records including information related to checking account, savings accounts, transaction logs, name associated with accounts and a unique ID for each account. The application will have the ability to load transaction and account data to the data structure from a local file in the same directory as the application. The program will include a simple menu system that allows the user to perform a variety of operations on the data in the structure.

DOMAIN DATA

- Unique 8 digit Account Identifier (**UID**) — Automatically assigned to users accounts and is the BEST way to search for an account in the data structure.
- Account Holder's Name (**firstName** and **lastName**) — The full name of the user who owns the account. This will be **2** separate data fields in order to search for people with same **last name** or same **first name** if you were to, for example, search for people in the same family.
- Checking Account and Savings Account (**cBal** and **sBal**) — Holding the balance of each of the respective accounts.
- Transaction Log (**tLog**) — Holding a list of all transactions **in order of insertion** into the log. Each transaction should include +- amount added or removed from the account, the account being modified (checking or savings), the date of the transaction.

DATA STRUCTURES

To implement Benny's Bank, through some research, we found that the most efficient way to store the actual account collection would be using an ArrayList data structure. The reasoning behind this decision is because we need to be able to dynamically add accounts to the bank without having to redefine a new size and copy elements over, which would be incredibly inefficient. The other reason is because we also need to be able to index accounts at any position in the list without having to iterate over the entire list looking for the account, giving us a very nice constant time complexity of $O(1)$. The ArrayList data structure allows us to do both of these things and also comes with a nice set of operations for adding and sorting the list if needed, even though we will be implementing our own sorting algorithm later anyways. While the ArrayList data structure works exceptionally well for storing the account objects, it falls short when storing the transaction logs. For this, we have decided that we would use a Singly Linked List which will allow us to automatically store transactions in the order that they happen. The Linked List Nodes will also allow us to create more than **one** data field which would help us in differentiating among date, amount, and account type data fields. The linked list data structure works well for our purposes because it also has a worst case indexing time complexity of $O(n)$.

OPERATIONS OF THE DATA STRUCTURE

- Add — Adding an account to the Bank. The user will be prompted for all required input fields.
- Delete — User will be prompted to enter the Unique Identifier of the account they wish to delete. The account will then be removed and all corresponding records and memory will be freed.
- Retrieve — Search by the Unique ID of the account. Then return the account object when found, else return "Not Found!". Most likely will be using a form of a binary search algorithm to do this.
- Load — Loads data into the Bank data structure from a file stored locally in the same directory as the application. This will be done, by having a carefully formatted JSON/XML file depending on which is easier to implement and then reading through the different properties of the JSON/XML object and storing them in their respective places within the Bank data structure.
- Save — Write the data to the JSON/XML file.
- Sort — Sorts the data using the Merge Sort Algorithm. The merge sort algorithm was chosen because it allows for the best time and space complexity in worst case scenarios. The application of a Bank data structure certainly qualifies as one which would have a large number of accounts (millions maybe) and we can't risk a single sort operation taking more than $O(nlogn)$ time.

CONSIDERATIONS

If while implementing the following application, we may switch to a different set of algorithms and applied data structures depending on our experimental analysis of the time complexity when using operations on our Bank data structure.

We will initially be attempting to implement the local file format as a JSON file, however, if we find that an XML or some other file format suits our purposes better, we may switch to that format.

END OF PROPOSAL