ASSIGNMENT 4 - REPORT

Benny's Bank

Liam Stickney, Ben Sykes

April 2nd, 2019 | Sheridan College

Problem Definition

The proposed problem (and solution) we created was a banking application. This application prompted the user for input, and would perform various tasks pertaining to that user's banking accounts, as well as their transaction records. Firstly, the user is presented a menu with 7 options for them to choose from. These options include: adding/deleting an account, retrieving an account, loading and saving accounts from external files, sorting the accounts, and exiting the program. Adding and deleting accounts simply adds or deletes an account from the list respectively, and since the accounts are given a unique ID automatically, the list is automatically sorted by the UID. This makes retrieving an account based on ID easier, as we can just perform a binary search on the list of accounts. Loading and saving accounts was a bit trickier, as we import and export the data in JSON format. This meant that we needed an external library in order to properly build our JSON data. This external library is included in the project, and does not require any installations on the user-end. The sorting method was not implemented, as the list is already sorted, and needs to be already sorted to use the binary search. Within the search account feature is another menu, that allows the user to specifically see a list (linked list) of the corresponding transactions for both checking and savings, as well as being able to add transactions for both accounts as well. Exiting the transaction display menu will return to the account menu, and exiting that menu will return to the main menu.

Data Structures and Operations

As mentioned, we used the array list to hold the accounts, or as it's known in C++, the vector data structure. We chose this particular data structure as it is unknown how many accounts will be in the structure at any given time. Therefore, we need to be able to dynamically add and remove accounts at any point, so it is beneficial to not have to constantly be iterating, searching and adding space to the vector, as that would be very inefficient. The other reason has to do with how we retrieve accounts in the list. Since we utilize the binary search algorithm, it is necessary to be able to quickly locate the middle element in the vector. By using the vector, we can easily find the middle element by index (by using the size of the vector at that time) in a very nice time complexity of ***O(1)***. The array list (vector) allows us to do both of these things, and even comes with its own built-in function to sort itself (which was obviously unused, but can be very helpful). In order to properly add new accounts to the vector, we used the vector method *push_back* which automatically appends an object to the back of the vector. This works well in our case, since as we add the new accounts, the UID will automatically increase. This means that the accounts will always be sorted as we add them. To remove accounts, we use the *erase* method, which removes an object at a specified index. This would mean that both adding and removing accounts work in ***O(1)*** time as well, since we either just append accounts to the back of the vector, or remove an account at a particular index. The area in which the vector falls short in this respect is that upon removing an account, the space in the vector that the account occupied is now permanently empty. We believe this to be a small issue, as in the real world, it would be very tedious and inconvenient to change the UID of thousands or millions of accounts just to save a bit of memory. Although the vector works well for storing the accounts, we believed it would be more efficient to use the singly linked list to

store the transactions within each account. This allowed us to automatically store these transactions in the order that they happen. Each linked list node also contains more than one data field, so we could differentiate between certain pieces of data, like date, amount and account type. The linked list also has a relatively low worst-case search time complexity of ***O(n)***, which works very well in this case. In the proposal, we mentioned that we would sort the list using the merge sort algorithm. The reasoning behind this decision is that, in a real-world application, banks store millions of bank accounts and their corresponding information, so we needed to choose a sorting algorithm that worked well with large amounts of data. Although algorithms like the insertion sort have a much better best-case complexity, insertion sort (as well as selection sort) fall short when working with data sets that become too large. Merge sort had the best time complexity overall for large sets of data, which is always ***O(n log n)***. Quick sort is also a very good algorithm for large data sets, but it's worst case is ***O(n^2)***, and we didn't want to risk having a worst-case complexity worse than n log n. However, we were unable to properly implement merge sort because of problems discussed below.

Rationale Behind the Chosen Data Structures

As mentioned before, we chose the array list because of its flexibility. As data sets get larger and larger (and in the case of banking information, they likely will), we figured that having to constantly move data around, add more space and delete space would become very inefficient when our data becomes too large. This meant that when we were picking our algorithms, we could rule out data structures like the linked lists and the stack, as those require the data to constantly be shuffled around. Not only is the array list very flexible, it is also very simple to use. As mentioned earlier, we only needed to use two of the functions built in to fully add and remove our accounts, and that's not to mention that the array list already includes its own function to sort itself. Since the only insertion method on the array list appends the information to the end of the list, we saw similarities between it and the queue. However, the queue only allows for information to be removed from the front of the list, while the array list can remove any given index in itself, which was obviously much more appropriate for what we were trying to implement. As for our choice for the linked list for the transactions, one might ask why we chose the linked list over another array list. The reasoning behind this decision is that, unlike the accounts, the transaction list will be much smaller. This is because in a real-world application, banks often only keep records of recent transactions available to a user in the past week or month. Therefore, there's less data to work with, and the issues of moving through too much data is much less of a problem. With this in mind, the linked list now poses several advantages over the array list. One of which is that it is faster to access and manipulate the data as, unlike the array list, it doesn't have to keep an internal array of the objects. This means that when editing the transaction list, no bit shifting occurs in the memory. The linked list can also act as both a list and a queue, giving more flexibility when it comes to adding or shifting data around. Overall, the reason we chose the linked list here is that we no longer had to deal with large amounts of data, so it just made more sense to switch to a structure that could handle less data, but could access said data faster.

Analyze the Designed Algorithms

For use on our array list, every function has a linear time complexity. This is because the array list allows us to select/delete/add new elements simply based on index. The array list also dynamically increases in size as we add new objects, so we don't have to worry about constantly freeing more space. Based on this information, both *push_back* and *erase* run in **O(1)** time, since we just need to specify an index value, and we can instantly obtain the value at that index without having to iterate through the entire list. In fact, *push_back* doesn't require an index at all, as it simply just appends the new information to the end of the list. As for sorting the list, since we use the binary search, the time complexity is **O(log n)**. This is because we first select the middle element of the list (**O(1**)), then get the highest and lowest elements as well (**O(1)** for each). Then we check if the UID of the account being searched for is greater or less than the UID of the middle account, and update the new middle account accordingly. In the worst-case, we would have to keep splitting the list in half over and over again until there is only one element left, which at that point, is either the account being searched for, or the account doesn't exist in the list. The time complexity in this case is **O(log n)**. The linked list implementation for our transactions runs its worse-case for searching and retrieving data at **O(n)**, since we'd start at the beginning node, and iterate through until we find the node we're looking for. If the transaction we're looking for is the first transaction in the list, obviously the time complexity will be linear. As for adding new transactions, the complexity is simply **O(1)** since we can just append them to the front of the list. Although the merge sort was never implemented, it's time complexity would have been **O(n log n)**. This is because we just need to get the middle element of the array list (which is linear, as stated before). Then we divide the list into 2, and each of these are divided into 2, etc. This gives a complexity of **O(log n)** and at each stage of the dividing, in the worst case, we'd need to make n comparisons, which gives an overall time complexity of **O(n log n).** This is the best worst-case we could find using a simple implementation that utilizes some sort of comparison sort.

Reflection on the Sorting and Searching Algorithms Used

We believe the binary search was the best way to search for our data. This comes with the basic nature of the application. Using a unique identifier is the best way to differentiate all of the data, and it makes more sense to automatically assign these IDs to avoid any chance of there being duplicate IDs. Since the array list also automatically appends the data to the end of the list, it would make sense to have the unique identifiers also correspond to the data's positon in the list, so the list would automatically be sorted by unique ID. This meant we could avoid extra iterations through the list if, for example, a user could set their own unique ID. If the user could choose an ID, we'd have to iterate through the full array list to make sure that ID isn't already in the list. While this made it easy to retrieve accounts in the list, since we could just retrieve the item in the list with an index of the UID, it made it redundant to include any kind of sorting algorithm. We discovered this issue late into development of the application. Our first solution involved including a merge sort that would sort the accounts by their balance totals, but the issue with that is now our retrieve function would no longer work, since we use

binary search to retrieve the accounts and binary search requires the list to be sorted by UID. The only way to fix this issue would be to add another sort option that sorted the list by UID again, but this would be very redundant as then the list would just be back to its original form once again. We then agreed that excluding the merge sort was the most logical course of action, as in order to properly retrieve data, our list needs to be sorted in the first place, and editing the order of the list would remove a highly important aspect of the application.