Bear Bank

Fall 2020, CSC232

Group Eight

Nathan Obert, nathan829@live.missouristate.edu

Dominic Zucchini, Zucchini8008@live.missouristate.edu

Zijing Zhao, Zhao1626@live.missouristate.edu

Table of Contents

Implemented Features……………………………………………… 2

Client Features 2

Bank Official Features 2

Bank Administrator Features 2

Data Structures……………………………………………………... 3

Introduction 3

Table 3

Users 5

Abstract Data Types 6

Accounts 7

Encryption 8

Extraneous 8

How to Compile and Run…………………………………………... 9

Bonus………………………………………………………………..10

Online Repository….……………………………………………..... 11

Implemented Features

Core features and extra stuff we did!

**Client Features:**

* Client enrollment
* User ID and password recovery
* Recent user activity
* View/Change user information
  + First name, last name, phone number, and address
* Request new accounts
* Deposit, withdraw, external deposit, and view account history

**Bank Official Features:**

* Open/Close accounts for clients
* Change fees, status, and rates for existing accounts
* Deposit/Withdraw from existing accounts (incurs a service fee on the account)
* Search for accounts by,
  + First or last name, user ID, phone number, address, and account number.

**Bank Administrator Features:**

* Modify bank Official accounts
  + Create new and edit existing
* Modify bank Administrator accounts
  + Create new and delete existing
* Modify account types
  + View all, create new, delete existing, and modify existing
* Retrieve a User ID
  + Display all Client, all Official, all Admin, and search by account number
* Change a password
* View table statistics
  + User table, first name table, last name table, phone number table, address table, and account table

Data Structures

Introduction

A table introducing all of the data structures used can be seen below. It offers a preliminary view on the types and contents of each data structure. For a more detailed view on each member of the objects and how the objects interact with each other, please view the UML pdf diagram titled, UML Diagram. Below the table can be found a justification and complexity analysis of each general type of function contained in the data structures. Further, setters and getters were omitted from analysis due to their inherent and negligible constant complexity.

Table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Description | Reference |
| node | User Defined | A node for a BST containing the height and a vector to contain items with the same key. | DataStructures/DataTables/node.h  DataStructures/DataTables/AVL.h  DataStructures/DataTables/AVL.cpp |
| AVLTree | Dictionary, BST | A self balancing binary tree that functions like a dictionary except with many mapped items to one key. Used to keep track of which piece of info maps to which accounts. | DataStructures/DataTables/AVL.h  DataStructures/DataTables/AVL.cpp |
| Node | User Defined | A node for a linked list. | DataStructures/DataTables/LinkedList.h |
| LinkedList | Doubly Linked List | A linked list with a pointer to the head and tail. Used inside other ADTSs. | DataStructures/DataTables/LinkedList.h |
| AccountQueue | Dynamic Queue | This queue is used to store Client requested accounts. Officials will be able to approve or deny them chronologically. It implements a linked list to become dynamic. | DataStructures/DataTables/AccountQueue.h |
| accountEntry | User Defined | An entry with a key of an account number with a value of that accounts info. Created so it could be inserted into a linked list for separate chaining. | DataStructures/DataTables/AccountTable.h  DataStructures/DataTables/AccountTable.cpp |
| AccountTable | Dictionary, Array | A hash-based dictionary that uses a dynamic array with separate chaining to store account information. | DataStructures/DataTables/AccountTable.h  DataStructures/DataTables/AccountTable.cpp |
| EncryptionBox | User Defined | An object used to encrypt, decrypt, and hash information in order to ensure all data is illegible and passwords are secure. The box is purely static. | DataStructures/DataTables/EncryptionBox.h  DataStructures/DataTables/EncryptionBox.cpp |
| tableSet | User Defined | A group of members comprised of AVLTrees and an AccountTable, for easy access. | DataStructures/Users/DataHandler.h  DataStructures/Users/DataHandler.cpp |
| User | User Defined | An abstract class to define the three types of users from. | DataStructures/Users/User.h  DataStructures/Users/User.cpp |
| Admin | User Defined | Inherited class from User. Helps provide admin functionalities and interact with data. | DataStructures/Users/Admin.h  DataStructures/Users/Admin.cpp |
| Official | User Defined | Inherited class from User. Helps provide official functionalities and interact with data. | DataStructures/Users/Official.h  DataStructures/Users/Official.cpp |
| Client | User Defined | Inherited class from User. Helps provide client functionalities and interact with data. | DataStructures/Users/Client.h  DataStructures/Users/Client.cpp |
| DateTools | User Defined | An object used to get and set dates. | DataStructures/Users/Tools.h  DataStructures/Users/Tools.cpp |
| AccountType | User Defined | An object used to help create new account types through various parameters. | DataStructures/Users/Account/Account.h  DataStructures/Users/Account/Account.cpp |
| Account | User Defined | A class derived from AccountType that helps facilitate Client interaction with a bank account based on account type attributes. | DataStructures/Users/Account/Account.h  DataStructures/Users/Account/Account.cpp |
| DataHandler | User Defined | A static class used to help users interact with data efficiently and in a standardized manner. | DataStructures/Users/DataHandler.h  DataStructures/Users/DataHandler/cpp |

Users

This program has three different types of users able to interact with stored data. As such, three distinct user classes were defined and derived from a common, abstract parent user. Beyond the general setter and getters that function in O(1), here are the complexity analyses for a user’s behaviors:

User

* Building from file: O(1)
* Saving to file: O(1)
* Set Last Login Time: O(1)

Each User has a specific set of attributes, so reading from a file will never grow in complexity. Setting the login time involves calling a function to output a string of the date and time. However, this function only performs conversion of readily available data, so this setting will occur in constant time as well.

Admin

* Reset password: O(log(n))
* Create Official or Admin: O(log(n))
* Modifying an Official or Admin: O(log(n))
* Displaying all Users, by type: O(n)
* Display User ID by Account Number: O(1)

The Admin has to interact with stored data. All data is stored in either a hash-based dictionary--which is only for Accounts--or a binary search tree-based dictionary, for which insertion and deletion will either be constant or logarithmic.

Official

* Adding a new Client: O(log(n))
* Searching by Client Information: O(log(n))
* Searching by Account Number: O(1)

Adding new Client data requires inserting into a binary tree, which has a logarithmic complexity. Searching for data is dependent upon which data structure is being searched; the dictionary containing Account information is hash-based, so it has constant complexity. Anything else has to search through a binary search tree, which again has logarithmic complexity.

Client

* Getting Account History: O(n)
* Rewriting Personal Information: O(log(n))

Accessing an Account’s history’s complexity is determined based on how much history there is to read through, which implies a linear relation. Changing information like a phone number, for example, needs to re-write the information stored in binary search trees, which makes the complexity logarithmic.

Abstract Data Types

Data is stored in three main types of containers in this program: binary search trees, a hash-based dictionary, and a queue. Both the dictionary and the queue use a doubly linked list in their implementations, to achieve separate chaining and to become dynamic, respectively. The binary trees were created due to the need to store multiple mapped items to a single key, e.g. many accounts can have the same first name. The dictionary was created to map a single key to a single value, e.g. an account number to its information. The queue was created due to a Client’s need to be able to open another account. If two Clients both wish to create an account, then whoever requested one first should be seen first by an Official. This relationship mimics a queue, and so a queue was created in order to store Client account requests.

Here are the complexity analyses of the general types of functions used inside the program of these abstract data types:

Binary Search Tree

* Insertion: O(log(n))
* Deletion: O(log(n))
* Displaying: O(n)
* Rotation: O(1)
* Searching: O(log(n))
* Statistics: O(1)
* Saving to a File: O(n)
* Reading from a File: O(n)

A self-balancing binary search tree has logarithmic time complexity in all aspects pertaining to locating a specific node. Displaying, however, requires iterating through each node, which is a linear relationship. Rotation is just a re-placing of pointers, which occurs the same way for a single node always, implying a constant time. Statistics for the tree use attributes that are already set inside a node or the tree itself, and accessing those will always be constant. Reading and saving to a file both require linear time since each element must be iterated through.

Doubly Linked List

* Appending: O(1)
* Removing: O(n)
* Removing first Element: O(1)
* Searching: O(n)
* Deep Copying: O(n)

The linked list has a pointer to the head node and the tail node, meaning changing nodes at the front and back of the list will occur in constant time. Removing and searching for an element iterates through each node starting at the head, which means removing has a linear relationship. Deep copying also has to iterate through each node, an act that is also linear.

Hash-Based Dictionary

* Insertion: O(1)
* Deletion: O(1)
* Rehashing: O(n)
* Searching: O(1)
* Statistics: O(1)
* Saving to a File: O(n)
* Reading from a File: O(n)

Inserting in the dictionary uses a hash function to find a specific index inside the array. Inserting into the linked list in that index is constant since the item can be appended in O(1). While deletion can technically be O(n) for n items in the linked list, the hash function has a good distribution, meaning for hundreds of thousands of items, no more than a dozen or so items will ever be placed in the same index. Iterating through 12 items is fast, approaching constant time. Rehashing requires the insertion of each element into a larger array, which implies a linear relationship. Saving and reading from a file require each element to be iterated through, something that is also directly proportional to the amount of elements inside the dictionary.

Dynamic Queue

* Insertion: O(1)
* Deletion: O(1)
* Saving to a File: O(n)
* Reading from a File: O(n)

Since the queue uses a doubly linked list inside, appending and deleting the first element occur in constant time. Saving and reading from a file require each element to be iterated through, something that is also directly proportional to the amount of elements inside the queue.

Accounts

Inside the bank, Clients have accounts, of course.This aroused the need to create a type of account object. However, administrators are able to create new account types. Instead of offering only a few ridgid accounts, a class where parameters were able to be modified was created to facilitate this creation. This led to an account type class with a derived class that is the actual account. Here are the complexity analyses, beyond the setters and getters that function in O(1):

Account Type

* Displaying a Number: O(n)
* Rounding a Number: O(1)

There is not much subsistence to this class beyond setting parameters, but the role it plays is vital to account functionality. Displaying a number returns a string with a double formatted to have only two decimal places, and creating a substring requires iterating through each character in a string. Rounding a number will occur in linear time due to the involvement of an algorithm without any for or while loops. In real time, both of these functions will occur near instantly.

Account

* Increment an Account Number: O(1)
* Deposit: O(1)
* Withdraw: O(1)
* Save attributes to File: O(1)
* Build from File: O(1)
* Display History: O(n)

Incrementing a static account number string uses an algorithm that will never scale up for how large or small the account number is, implying constant time complexity. Depositing and withdrawing use a series of if statements, but nothing is used that would scale up in time either. Since an account has an immutable number of attributes, saving and building from a file will always occur in constant time since no more or less attributes will need to be read in or saved. Displaying history, however, will scale with however much history is inside an account, so it is a linear relationship.

Encryption

In order to preserve privacy, the need to encrypt data into some illegible form was required. To solve this issue, the EncryptionBox was created. A unique type of ascii shift is performed on data, mapping all expected input characters into a specific, larger codomain of illegible characters. Further, this object contains the ability to perform a hash on a string, outputting a hexadecimal value, which was used as the comparison basis for all passwords. This hash function was also used inside the hash-based dictionary to distribute keys through a specific range of indices. Here are the complexity analyses for those functions:

* Encrypt: O(n)
* Decrypt: O(n)
* Hash: O(n)

All three functions have linear complexity for the same reason: the need to iterate through each character in an input string. However, while strings can have varying length, they will never be large. This means that, while technically O(n), all functions will perform like they are O(1).

Extraneous

There are two user defined objects with behaviors left to discuss. They are the Datahandler and DateTools classes. However, these objects are manipulated by the other data structures discussed above. These objects merely provide easy and standardized access to data already available, so their complexity analyses have already been covered.

How To Compile and Run

The BearBank program is designed and tested to run only through cygwin on Windows 64-bit. Beyond having all implementation and header files, the only requirement is have three specific directories: AccountData, Tables, and UserData. These three directories must be visible and on the same level as the compiled executable.

To compile the program, first ensure all files and directories exist. The three aforementioned may be empty. After ensuring this, type the following line into a cygwin terminal:

G++ driver.cpp -o GroupEightProject

./GroupEightProject

Those lines will create an executable file named ‘GroupEightProject’ and run it. At the beginning of the program, there will be messages displayed mentioning specific files could not be found. That is acceptable. Those merely inform the user that no data is being read from, meaning the program will begin from scratch. If a user is wishing to begin from scratch again, all that is needed is to delete all the files in the three aforementioned directions *but not the directories themselves.* It is imperative they exist.

Two default users are provided for the user to interact with, an Official and an Administrator. These two profiles are hard coded, so they will always exist. Further, they are not able to be modified like other BearBank employees. The logins are the following:

User ID: official Password: password1

User ID: admin Password: password1

To login as a Client in the Bear Bank, go through the following enrollment option! All the features of each type of user are outlined in the Core Features section at the beginning.

Happy Bear Banking!

Bonus Criteria:

Group eight is deserving of bonus points for their implementation of multiple advanced data structures, methods for security of user data, and additional interactive features such as usage statistics. Even when using Bear Bank, you will find that all presentations are nicely spaced, and all interactive menus guide the users to easily interact with the bank’s data. The UML class diagram chart shows just how much group eight went beyond the scope of the project requirements to provide a fulfilling banking experience.

Online Repository Link

Over the course of this project, several branches were utilized. However, the only one remaining is ‘main,’ as all other branches were deleted upon culmination of this project. Click the following link to be taken to the repository:

<https://github.com/nathan37226/FinalProject.git>