**BEAR Bank Software**

**CSC232 Final Project**

Ashton Barnwell - [barnwell247@live.missouristate.edu](mailto:barnwell247@live.missouristate.edu)

Daelon Kingore - [Daelon022@live.missouristate.edu](mailto:Daelon022@live.missouristate.edu)

Riley Grotenhuis - [rg1050@live.missouristate.edu](mailto:rg1050@live.missouristate.edu)

[GitHub Repository](https://github.com/rileygrotenhuis/CSC232-FinalProject)

[Discord Server Link](https://discord.gg/qJQ7YgPCAM)

# 

# 

**Table of Contents**

[**Project Overview**](#_3znysh7) **3**

[**BEAR Bank Features**](#_2et92p0) **4**

[**Data Structures**](#_3dy6vkm) **6**

[**User-Defined Data Types**](#_4d34og8) **10**

[**Extra Credit**](#_17dp8vu) **11**

# 

# **Project Overview**

**How to Run Our Software:**

1. First, you will need to download the source code for our BEAR Bank software. You will be able to get this either from a downloadable .zip file, or you can download the source code from our public GitHub repository.
2. Once you have downloaded the source code for our software, you can run the program in two different ways:
   1. You can simply open the folder that you downloaded the software into, and double click on the executable file titled “ProjectMainFunctions”. Once you double click on this file, the terminal on your machine will open and the software will begin.
   2. To manually run the software, open your terminal and change your directory until you are in the folder that you downloaded the code into. Once you have reached the project folder, you will run the following command: “g++ ProjectMainFunctions”. After that command is run, a new executable file will be opened in the folder and you can run this command “./a.out”. Once you run this command, the BEAR Bank software will be started on your machine.

**Folder Structure:**

accounts/

* Account.h

dataStructures/

* LinkedList.h
* OfficialBST.h
* UserBST.h
* Queue.h
* Stack.h

users/

* BankUser.h
* BankOfficial.h

utilities/

* ceaserCypher.h
* stringToVector.h
* time.h

.gitignore

myUtilities.cpp

ProjectMainFunctions.cpp

README.md

# **BEAR Bank Features**

BEAR Bank software is a terminal-based banking software created using C++ by Ashton Barnwell, Daelon Kingore and Riley Grotenhuis. BEAR Bank software is used to open, close, and manipulate the bank accounts that BEAR Bank offers.

As of today, BEAR Bank offers four different types of accounts:

**Checking Accounts:** A Daily Checking Account will have no monthly fees and

no interest.

**Saving Accounts:** A Daily Savings Account will have no monthly fees but will

include a variable daily interest rate that is set by the Bank Officials and System

Administrator.

**Certificate of Deposit Account:** A Certificate of Deposit Account will have an

early cancelation penalty based on the number of days remaining. There will be a fixed interest rate on the Certificate of Deposit set by the bank admin.

**Custom Bear Bank Accounts:** A Custom Bear Bank Account allows the System Administrator to customize the interest rate and other features of this type of account. All Custom Bear Bank Accounts will be the same as per the Administrator variables and there are monthly and service charges in this type of account.

BEAR Bank also allows for different types of users to login, who will have different types of capabilities depending on what type of user they are. As of today, BEAR Bank has three different types of users:

**System Administrator:** Our BEAR Bank software will have only one System

Administrator (SA). The SA is able to create Bank Official login profiles, as well as

enable/disable those profiles. They can also retrieve the login information and

may change the username and/or password not only for the Bank Officials, but

the Bank Users as well. The SA is also responsible for modifying the different

types of accounts that BEAR Bank offers. An example of this could be the interest

rate of a certain Custom Bear Account.

**Bank Official:** Each Bank Official in our BEAR Bank software will only have a

username and password stored for them. The Bank Official is in charge of

managing each Bank User in the BEAR Bank software, as well as manipulating

each Bank Users’ account(s). The Bank Official can do

any of the following: open/close any type of account for a Bank User,

deposit/withdraw money from a Bank User’s account (with their permission of

giving their username and password), and they are able to search any account in

the BEAR Bank software by the account number or the Bank User’s name or phone number.

**Bank User:** Each Bank User in our BEAR Bank software will have the following

information stored for them: their first/last name, phone number, address, and a

unique username and password. Each Bank User will also have each of the

accounts they are in charge of stored into our BEAR Bank software. Bank Users

are not able to make deposits/withdrawals directly from their accounts without

the help of a Bank Official, but they still have some capabilities. After a Bank User

logs in, they will be able to change their username and/or password at any given

time, see a specific account’s details, view the current balance of a specific

account, and see each of their accounts’ transaction history within the last week.

# 

# **Data Structures**

We chose to implement various data structures into our BEAR Bank software to increase runtime and efficiency while using the software. We deliberately chose these data structures so that BEAR Bank would be a very user-friendly experience. Here is a list of the data structures we chose to use in our software and the operations each one of them has:

**Linked List:** A Linked List is a linear collection of data elements whose order is

not given by their physical placement in memory. Instead, each element points to

the next. It is a data structure consisting of a collection of nodes which together

represent a sequence. Each node in our implementation of a Linked List stores an Account class object (referenced later in the documentation) as the data type, and a pointer to the next node in the Linked List. Our implementation includes the following operations:

* isEmpty() - This operation checks whether or not the Linked List is empty as a boolean value.
* append(Account) - This operation takes an Account class object as a parameter, creates a new node with this new object, and inserts that node to the end of the List.
* insertAfter(string, Account) - This operation is similar to the append operation, in which it creates a new node with an Account class object passed as a parameter, but in this operation you can choose to insert it after a specific item in the List by searching for one of the Account class objects’ ID.
* deleteNode(string) - This operation searches through the Linked List by looking for a specific Account ID. If that ID exists in one of the nodes in the List, it will delete that item in the List, and free the memory slot.
* size() - This operation returns the number of items in the Linked List as an integer value.
* search(string) - This operation searches through the Linked List by looking for a specific Account ID and will return whether or not that Account ID exists as a boolean value.
* get(string) - This operation will search for an Account class object’s Account ID using a string value passed as a parameter and will return a pointer to that specific class object. This will allow that class object to be manipulated in the ways the program needs.
* display() - This operation loops through the entire Linked List and displays the information in each node to the user.

**Stack:** A Stack can be logically thought of as a linear structure represented by a

real physical stack or pile. A structure where insertion and deletion of items takes

place at the top of the stack. The Stack data structure follows the Last In First Out

principle (LIFO). We used class templates in our implementation of Stack, so the

data type that is held in each node will be defined by the programmer when it is

created in the software. Our implementation includes the following operations:

* isEmpty() - This operation checks whether or not the Stack is empty and returns that as a boolean value.
* push(T) - This operation takes the template value (which we call T in our implementation), and then a node will be created and the operation will insert that node to the top of the Stack.
* pop() - This operation returns the value of the top element in the Stack, and then changes the node that is the top of the Stack to the next node down the line. This is essentially removing the top node from the Stack.
* peek() - This operation does nearly the same thing as the pop() operation, but instead of removing that element from the Stack, it just returns the value of the top node in the Stack. This is used so the user and/or programmer can quickly check what is currently at the top of the Stack without removing it.
* size() - This operation returns the number of items that are currently in the Stack as an integer value.

**Queue:** A Queue data structure is somewhat similar to Stacks in which it is thought of as a linear structure. However, the Queue follows the First In First Out principle (FIFO). This can be compared to a queue in the real physical world, like a waitlist for a restaurant. Much like our Stack implementation, our Queue implementation uses class templates so the programmer can decide what values will be stored when they create it. Our implementation includes the following operations:

* isEmpty() - This operation checks whether or not the Queue is empty and returns that as a boolean value.
* enqueue(T) - This operation takes the template value (which we will also call T much like the Stack), and then a node will be created and the operation will insert that node to the end of the queue.
* dequeue() - Much like the pop() operation in our Stack, this operation takes the node that is currently at the front of the Queue and removes it from the Queue and returns that value of the node at the front of the Queue.
* getAll() - This operation is similar to the dequeue() operation, in which it removes a node that is at the front of the Queue, but instead this operation removes each element in the Queue and inserts it to a vector. This vector is then returned as the value for this operation.
* top() - This operation returns the element that is in the node at the front of the Queue currently.
* size() - This operation returns the number of nodes that are currently in the Queue.

**Binary Search Tree:** A Binary Search Tree (BST) is a tree in which all of the

nodes follow the mentioned properties. The value of the key of the left subtree is

less than the value of its parent (root) node’s key. The value of the key on the

right subtree is greater than or equal to the value of its parent (root) node’s key.

We have two different implementations of BST’s in our BEAR Bank software: one that uses a BankUser class object as the data type, and one that uses a

BankOfficial class object as the data type (both mentioned later in the

documentation). Our implementation includes the following operations:

* deleteN(BankUser/BankOfficial) - This operation takes a BankUser/BankOfficial class object as a parameter, and loops through the entire BST looking for that class object in one of the tree’s nodes. If that class object exists in one of the tree’s nodes, the operation will remove that node from the BST and delete that node’s memory slot.
* insertNode(BankUser/BankOfficial) - This operation takes a BankUser/BankOfficial class object as a parameter and the operation will create a new node using this class object. Then the operation will use recursion to loop through the entire tree and insert the node in the correct placement in the BST based on the BST definition stated above.
* height(TreeNode\*) - This operation takes the current root node of the BST as a parameter and uses recursion to loop through the entire tree to find the current height of the tree.
* isBalance(TreeNode\*) - This operation takes the current root node of the BST as a parameter and uses recursion to loop through the entire tree to check whether or not the tree is balanced. The operation will return this as a boolean value.
* getLeafs(TreeNode\*) - This operation takes the current root node of the BST as a parameter and uses recursion to loop through the entire tree to count the total number of Leaf Nodes the current BST has. The operation will return this as an integer value.
* searchNode(string) - This operation takes a string value as a parameter and then the operation will search through the BST’s nodes to find a matching data type. This string can be matched with any of the data types that the BankUser or BankOfficial may have (username, password, firstName, lastName, etc...). If there is a matching value somewhere in the BST, the operation will return so using a boolean value.
* getUserNode(string) - This operation is very similar to the searchNode() operation, however instead of just returning whether the node exists in the BST or not, this operation will return the actual class object that the string parameter matches.
* displayInOrder() - This operation displays each of the nodes in the BST using inorder traversal.
* displayPreOrder() - This operation displays each of the nodes in the BST using preorder traversal.
* displayPostOrder() - This operation displays each of the nodes in the BST using postorder traversal.

As you can see, each of the data structures we have chosen to use in our BEAR Bank software was carefully chosen and carefully crafted with the operations we chose to use in each of them. Each of these data structures are used throughout our software for many different reasons. Here is a table showing the different uses of each of the data structures implemented into our software.

|  |  |  |  |
| --- | --- | --- | --- |
| **Linked List** | **Stack** | **Queue** | **Binary Search Tree (BST)** |
| Our Linked List implementation is used in the BankUser class (BankUser.h file).  We used this implementation to store the different types of Accounts that each Bank User owns. | Our Stack implementation is used in the Bank User class (BankUser.h file). We used this implementation to store the login history for a given user. | Our Queue implementation is used in the Account class (Account.h file). We used this implementation to store the transaction history for a given account. | Our BST implementation is used in our main driver file (myUtilities.cpp). We used two BST’s in our software to store the different number of Bank Users, and the number of Bank Officials that are stored in BEAR Bank. |
| Average Case: O(n)  Worst Case: O(n) | Average Case: O(1)  Worst Case: O(1) | Average Case: O(1)  Worst Case: O(1) | Average Case: O(log n)  Worst Case: O(log n) |

# 

# **User-Defined Data Types**

BEAR Bank software uses a multitude of user-defined data types in order to make the experience for the user fast and efficient. We have created all of our user-defined data types in separate header files, and stored them in classes. Here are the user-defined data types we have created and use in BEAR Bank software:

**Account:** This user-defined data type is used to represent the different types of bank accounts that a user of BEAR Bank can open (these include checking, savings, certificate of deposit, and/or custom account). Each of these accounts can include the following information (depending on the type of account): account ID, password, open date, close date, current balance, interest rate, and a list of the transaction history. Here are the following operations that we chose to include in the Account class:

* deposit(double) - This operation takes a double value as a parameter and adds that amount to the Account’s current balance. Before the operation is able to deposit the money, the program will ask the user to enter the password for their account, at that point they will have three attempts to enter their password correctly.
* withdraw(double) - This operation does the exact same thing as the deposit() operation, except the double amount passed in the parameter is taken from the current balance of the account.

**Bank Official:** This user-defined data type is used to represent a Bank Official user for the BEAR Bank software. Each of these Bank Officials will store only a username and password since that is the only information the Bank Officials will need. Since the Bank Official only stores the username and password, there are no operations for the Bank Official, there are only the appropriate setters and getters for the username and password.

**Bank User:** This user-defined data type is used to represent a Bank User for the BEAR Bank software. Each of these Bank Users will store a first name, last name, phone number, address, username, password, the last login date, and a list of the accounts that each Bank User owns. Here are the following operations that we chose to include in the Bank User class:

* addAccount(Account) - This operation takes an Account class object as a parameter and appends that class object to the end of the Linked List that stores each of the accounts that the Bank User owns.
* getAccount(string) - This operation takes a string value as a parameter (which will represent an account ID), and the operation will loop through the Linked List of the Bank User’s accounts and find the Account class object that has an account ID that matches the string parameter. If there is a match, the operation returns a pointer to that specific Account class object.
* searchAccount(string) - This operation, much like the getAccount() operation, loops through the Linked List of accounts using the string parameter, and returns whether or not there is a matching account ID to the string parameter as a boolean value.

**Extra Credit**

Before we wrote any lines of code, our team got together and planned out how we wanted our software to look when it was completely finished. The way we approached this was by creating a Google Slides presentation that was shared among all the group members that would demonstrate the flow of the program. This played a huge part in the creation of our software because knowing how the actual program would look and feel before writing it helped improve the efficiency that we could code. Here is the link for the [Google Slides presentation.](https://docs.google.com/presentation/d/10it9bhIqWSDfdF9nhf9JmUQR_gMFmN1KXysgJhpsuyU/edit?usp=sharing)

Our team’s collaboration and teamwork was by far the most important aspect in creating the software for this project. As soon as the teams were released in the project instructions, we immediately got together and created a Discord server. This allowed us to streamline our ideas, code, and so much more into one central location instead of a mass email chain. Our Discord server included a “general\_discussion” channel that allowed us to chat about any ideas that we had. Our server also included a “general\_announcements” channel (managed by Ashton Barnwell) which was used to make announcements for the meetings. These announcements detailed what each team member would be working on for the time in-between then and the next team meeting. This was a huge help for us in our time management while working on this project. Knowing who was working on what and dividing it up evenly helped us to better organize our project. We also included a “code\_logs” channel in our server which was a place for each of the team members to post what code they had written and completed. This allowed us to easily see who worked on what aspects of our software, so if anybody had any questions it was easier to track down the source of who wrote that code. We included a live voice channel in our server as well that we used for our team meetings as well as paired programming sessions. There were times (more towards the end of the project) in which two or more group members would get on the voice channel and work on one piece of code together at the same time. This allowed for each of those team members to be able to bounce ideas off of each other, making the whole process smoother. The link for our Discord server is included on the cover page of this document.

The day before we submitted our program to be graded, we all joined our Discord server voice channel to bug test every possible case, and make any changes to our software that we could possibly think of. We believe that this session of bug testing and revision makes our software nearly impossible to break while using it. Every single kind of invalid user input was accounted for.