Data Structures and Algorithm Project Report

Bank Management System

Name: Afnan Bin Abbas

Reg No: 2022048

Faculty: Cyber Security

Code: CS221-D

Submission Date: 29/12/23

Submitted to: Dr Ali Imran Sandhu

Table of contents

Introduction

In this Project, we will cover how to create a bank management system in C++ programming language. A **bank management system** can be used to manage a bank account, withdraw and deposit funds, and check the account balance.
Specifically, our system will support user login with a username and password as well as the ability to save data of an account and its banking information.

The provided code implements a simple Banking Management System using C++. The system is structured into three main components: Account Management System, Transaction History, and Account Verification. Each component is implemented using different data structures such as Linked List, Hash Table, Queue, and Binary Search Tree.

What are the requirements for this program?

System Requirements

1. C++ compiler
2. Command Prompt or Terminal to run the program

Technical Requirements

1. Knowledge of C++ OOP and Data Structures and Algorithm principles and classes.

Project Requirements

1. Banking functionality: Account creation, transactions, account retrieval, and other banking operations to be performed by the user.
2. User input/output: input will be gathered for account creation and navigation of the banking program. Output in terminal will be used to display instructions to the user or account data.

What classes will be utilized?

This banking program utilizes a multitude of classes that each have their own responsibilities:

1. **CustomerList**: responsible for saving comprehensive customer information, enabling account creation, customer display.
2. **HashTable**: A hash table class swiftly manages customer accounts by mapping account numbers for efficient retrieval and accelerated account management operations.
3. **TransactionQueue**: Organizes account transaction history with deposit, withdrawal, and fund transfer functionalities, enabling users to enqueue new transactions, dequeue older ones, and display a well-organized limited history
4. **Credentials**: manages banking manager accounts by providing login functionality and the ability to view user data.
5. **AccountVerification:**

Program Design

Each class represents and organizes the different responsibilities within the banking system. Through keeping an organized structure, the code can be easily updated and maintained.
The **main** function takes care of user interactions and ensures all users can easily access and use their bank account to perform their banking needs.

Step 1: Create 'CustomerDetails' Class

For our users and their banking data to be stored, we will define a CustomerDetails class that initiates the members that contains the details of a customer when an instance is made such as **Customer_Name, Customer_Address, Customer_AccountNo., Customer_Contact.**

```
#include <iostream>
#include <list>
#include <string>
#include <unordered_map>
#include <queue>

using namespace std;

// Linked List Node for Customer Information
Class CustomerDetails
{
Public:
    long long accountNumber;
    string name;
```

```
    string address;
    long long contact;
    CustomerDetails *next;

    CustomerDetails() : accountNumber(0), name(" "), address(" "), contact(0),
next(nullptr)
    {
    }
};
```

Step 2: Define 'CustomerList' Class.

2.1 Linked List

A linked list is employed to store customer information, where each node represents a customer. The details stored include account number, name, address, and contact information. Users can create new accounts, display all customers, and retrieve customer details by account number.

2.2 Hash Table

A hash table is used to efficiently manage customer accounts. The hash table maps account numbers to corresponding customer information, ensuring quick and efficient retrieval. This enhances the speed of operations related to account management.

```
// Linked List for Account Management
class CustomerList
{
private:
    CustomerDetails *head;
    HashTable hashtable;

public:
    CustomerList() : head(nullptr)
    {
    }
    // Function to create a new customer account
    void createAccount()
    {
        CustomerDetails *newNode = new CustomerDetails;

        cout << "\n<============== Create Account here ==============>\n\n";
        cout << "Enter Account No: ";
        cin >> newNode->accountNumber;
        cout << "Enter Name: ";
        cin.ignore();
```

```
        getline(cin, newNode->name);
        cout << "Enter Address: ";
        getline(cin, newNode->address);
        cout << "Enter Contact No: ";
        cin >> newNode->contact;

        // Add the new customer to the linked list
        if (head == nullptr)
        {
            head = newNode;
        }
        else
        {
            newNode->next = head;
            head = newNode;
        }

        // Insert the customer into the hash table for efficient retrieval
        hashtable.insert(newNode->accountNumber, newNode);
    }
// Function to display details of all customers
    void displayAllCustomers()
    {
        int count = 0;
        CustomerDetails *temp = head;

        cout << "\n<============ Customer Details ============>\n";
        cout << "\t[Display All Customers]\n";

        while (temp != nullptr)
        {
            count++;
            cout << "\n\n";
            cout << "Customer #[" << count << "]\n";
            cout << "\nAccount No.: " << temp->accountNumber;
            cout << "\nName: " << temp->name;
            cout << "\nContact No.: " << temp->contact;
            cout << "\nAddress: " << temp->address;

            temp = temp->next;
        }
        cout << endl;
        cout << "-----------------------------------------------";
    }
    // Function to retrieve customer details by account number
    CustomerDetails *getCustomer(long long accNo)
    {
        return hashtable.retrieve(accNo);
    }
};
```

getCustomer(): saves the input by the user inside the hashtable using the Account ID.

displayAllCustomers(): Displays all the customers who have their accounts in the Bank.

createAccount(): Takes input from a new customer while opening the account. Eg. **Customer_Name, Customer_Address, Customer_AccountNo., Customer_Contact**.

Step 3: Define 'TransactionQueue' Class

3.1 Queue

A queue is implemented to maintain transaction history for each account. The system supports deposit, withdrawal, and fund transfer functionalities. Users can enqueue new transactions, dequeue older transactions, and display the transaction history. This ensures a well-organized and limited transaction history.

```
// Queue for Transaction History
class TransactionQueue
{
private:
    queue<string> transactionHistory;
public:
    // Function to enqueue a new transaction
    void enqueueTransaction(const string &transaction)
    {
        // Limiting to 10 transactions for efficiency
        if (transactionHistory.size() >= 10)
        {
            transactionHistory.pop();
        }
        transactionHistory.push(transaction);
    }
    // Function to display transaction history
    void displayTransactionHistory()
    {
        cout << "\n<============ Transaction History ============>\n";
        cout << "\t[Display Transaction History]\n";

        queue<string> tempQueue = transactionHistory;

        while (!tempQueue.empty())
        {
            cout << tempQueue.front() << endl;
            tempQueue.pop();
```

```
        }
        cout << "-----------------------------------------";
    }
};
```

Step 4: Define ‘AccountVerification’ and ‘Credentials’ Class

4.1 Binary Search Tree

A binary search tree (BST) is utilized for efficient verification of account details during login or transactions. The BST structure enables quick determination of whether an account exists and facilitates the retrieval of associated information. This enhances the security and efficiency of the login process.

```
class Credentials
{
Public:
    long long accountNumber;
    string password;
    string name;
    long long contact;
    string address;

    Credentials *left;
    Credentials *right;

    Credentials() : accountNumber(0), password(" "), left(nullptr), right(nullptr)
    {
    }
};

class AccountVerification
{
public:
    Credentials *root;

    AccountVerification() : root(nullptr)
    {
    }

    // Function to insert new account credentials into the binary search tree
    void insert(Credentials *&node, long long accNo, const string &password, const
string &name, long long contact, const string &address)
    {
        if (node == nullptr)
        {   node = new Credentials;
```

```
            node->accountNumber = accNo;
            node->password = password;
            node->name = name;
            node->contact = contact;
            node->address = address;
        }
        else if (accNo < node->accountNumber)
        {
            insert(node->left, accNo, password, name, contact, address);
        }
        else if (accNo > node->accountNumber)
        {
            insert(node->right, accNo, password, name, contact, address);
        }
        // If accNo already exists, you might want to handle it accordingly (perhaps
update the existing node).
    }

    // Function to search for account credentials in the binary search tree
    Credentials *search(Credentials *node, long long accNo)
    {
        if (node == nullptr || node->accountNumber == accNo)
        {
            return node;
        }

        if (accNo < node->accountNumber)
        {
            return search(node->left, accNo);
        }

        return search(node->right, accNo);
    }
    // Other functions for BST operations can be added as needed
};
```

insert(): Function to insert new account credentials into the binary search tree.

search(): Searches for the accno that is being passed by the user and returns the correct accno. Function to search for account credentials in the binary search tree (BST).

Step 5: Finalize The Banking Simulation

To complete our banking simulation, we must ensure that the user can choose between different banking and account options. To do so, we will display these choices as a list and prompt the user to enter the number corresponding with the choice they'd like to make:

```
int main()
{
    CustomerList customerList;
    TransactionQueue transactionQueue;
    AccountVerification accountVerification;

    int choice;

    // Main menu loop
    do
    {
        cout << "\n\n<============= Banking Management System =============>\n";
        cout << "\t[Main Menu]\n\n";
        cout << "1. Account Management System (Linked List, Hash Table)\n";
        cout << "2. Transaction History (Queue)\n";
        cout << "3. Account Verification (Binary Search Tree)\n";
        cout << "4. Exit\n";
        cout << "\nEnter your choice: ";
        cin >> choice;
        cout << "\n";

        switch (choice)
        {
        case 1:
        {
            int accountManagementChoice;

            // Account Management System submenu
            do
            {
                cout << "\t[Account Management System]\n";
                cout << "1. Create Account\n";
                cout << "2. Display All Customers\n";
                cout << "3. Get Customer Details by Account No.\n";
                cout << "4. Back to Main Menu\n";
                cout << "\nEnter your choice: ";
                cin >> accountManagementChoice;
                cout << "\n";

                switch (accountManagementChoice)
                {
                case 1:
                {
                    customerList.createAccount();
                    break;
                }
                case 2:
                {
                    customerList.displayAllCustomers();
                    break;
```

```
            }
            case 3:
            {
                long long accNo;
                cout << "Enter Account No. to retrieve details: ";
                cin >> accNo;
                CustomerDetails *customer = customerList.getCustomer(accNo);
                if (customer != nullptr)
                {
                    cout << "\nCustomer Details for Account No. " << accNo << ":\n";
                    cout << "Name: " << customer->name << "\n";
                    cout << "Contact No.: " << customer->contact << "\n";
                    cout << "Address: " << customer->address << "\n";
                }
                else
                {
                    cout << "\nCustomer not found for Account No. '" << accNo <<
"'!!\n";
                }
                break;
            }
            case 4:
            {
                break; // Go back to the main menu
            }
            default:
                cout << "Invalid choice. Please enter a valid option.\n";
            }

        } while (accountManagementChoice != 4);
        break;
    }
    case 2:
    {
        int transactionHistoryChoice;

        // Transaction History submenu
        do
        {
            cout << "\t[Transaction History]\n";
            cout << "1. Enqueue Transaction\n";
            cout << "2. Display Transaction History\n";
            cout << "3. Back to Main Menu\n";
            cout << "\nEnter your choice: ";
            cin >> transactionHistoryChoice;
            cout << "\n";

            switch (transactionHistoryChoice)
            {
            case 1:
```

```
                {
                    string transaction;
                    cout << "Enter Transaction Details: ";
                    cin.ignore();
                    getline(cin, transaction);
                    transactionQueue.enqueueTransaction(transaction);
                    break;
                }
                case 2:
                {
                    transactionQueue.displayTransactionHistory();
                    break;
                }
                case 3:
                {
                    break; // Go back to the main menu
                }
                default:
                    cout << "Invalid choice. Please enter a valid option.\n";
                }

            } while (transactionHistoryChoice != 3);
            break;
        }
        case 3:
        {
            int accountVerificationChoice;

            // Account Verification submenu
            do
            {
                cout << "\t[Account Verification]\n";
                cout << "1. Create Account Credentials\n";
                cout << "2. Login\n";
                cout << "3. Back to Main Menu\n";
                cout << "\nEnter your choice: ";
                cin >> accountVerificationChoice;
                cout << "\n";

                switch (accountVerificationChoice)
                {
                case 1:
                {
                    long long accNo;
                    string password, name, address;
                    long long contact;

                    cout << "\n<============= Create Account Credentials
=============>\n\n";
                    cout << "Enter Account No: ";
```

```
                    cin >> accNo;
                    cout << "Enter Name: ";
                    cin.ignore();
                    getline(cin, name);
                    cout << "Enter Address: ";
                    getline(cin, address);
                    cout << "Enter Contact No: ";
                    cin >> contact;
                    cout << "Set Password: ";
                    cin.ignore();
                    getline(cin, password);

                    accountVerification.insert(accountVerification.root, accNo,
password, name, contact, address);
                    break;
                }
                case 2:
                {
                    long long accNo;
                    string password;
                    cout << "\t\t[Login]\n\n";
                    cout << "Enter Account No: ";
                    cin >> accNo;
                    cout << "Enter Password: ";
                    cin.ignore();
                    getline(cin, password);

                    Credentials *foundCustomer =
accountVerification.search(accountVerification.root, accNo);
                    if (foundCustomer != nullptr && foundCustomer->password == password)
                    {
                        cout << "\nLogin Successful!\n";
                        cout << "Customer Details:\n";
                        cout << "Name: " << foundCustomer->name << "\n";
                        cout << "Contact No.: " << foundCustomer->contact <<"\n";
                        cout << "Address: " << foundCustomer->address << "\n";
                    }
                    else
                    {
                     cout << "\nLogin Failed. Invalid Account No. or Password.\n";
                    }
                    break;
                }
                case 3:
                {
                    break; // Go back to the main menu
                }
                default:
                    cout << "Invalid choice. Please enter a valid option.\n";
                }
```

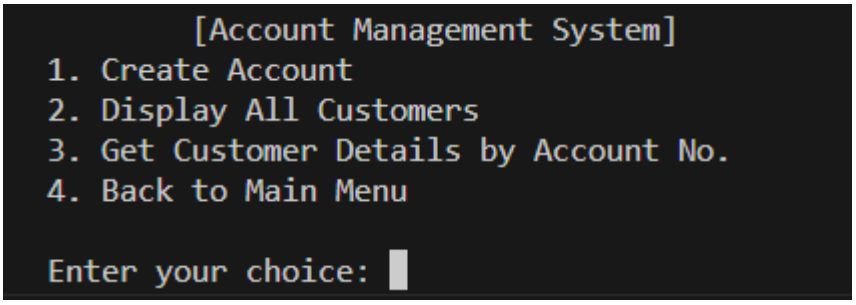
```
            } while (accountVerificationChoice != 3);
            break;
        }
        case 4:
        {
            break; // Exit the program
        }
        default:
            cout << "Invalid choice. Please enter a valid option.\n";
        }
    } while (choice != 4);
    return 0;
}
```

Step 6: Testing our Implementation.

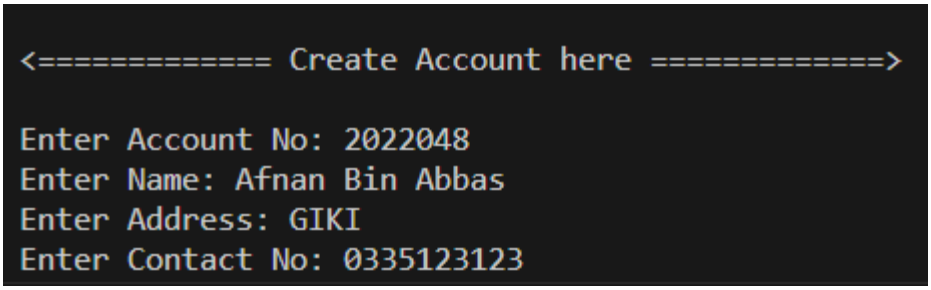
Main Menu

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    SEARCH ERROR

PS C:\Users\SS\Desktop\Bank Management System> cd "c:\Users\SS\Deskt
A_project.cpp -o DSA_project } ; if ($?) { .\DSA_project }


<============= Banking Management System =============>
        [Main Menu]

1. Account Management System (Linked List, Hash Table)
2. Transaction History (Queue)
3. Account Verification (Binary Search Tree)
4. Exit

Enter your choice:
```

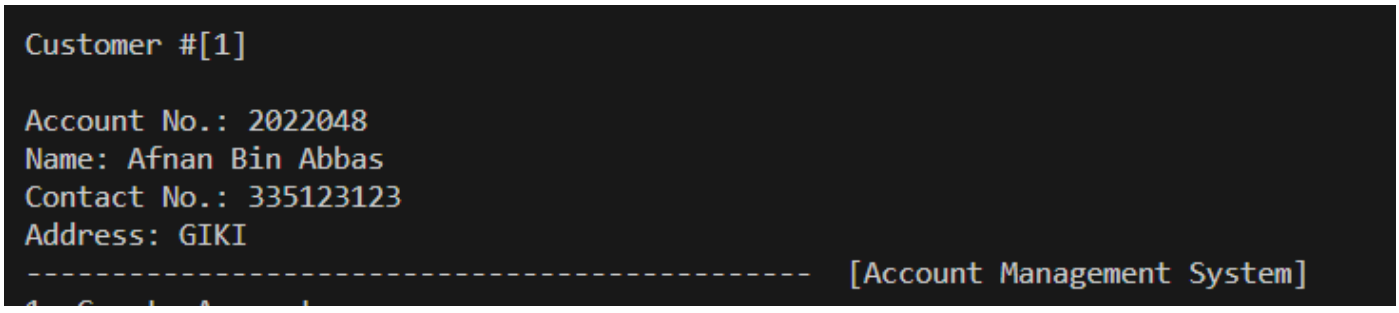
Account Creation Menu:



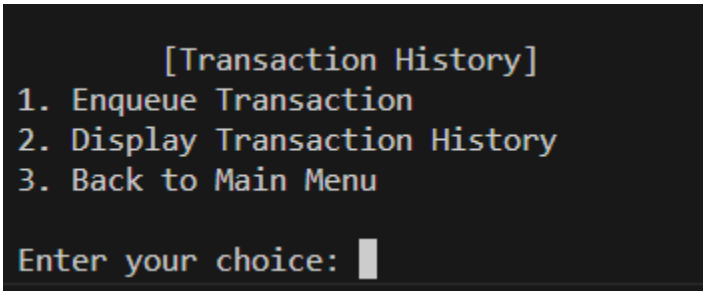
Taking Input:



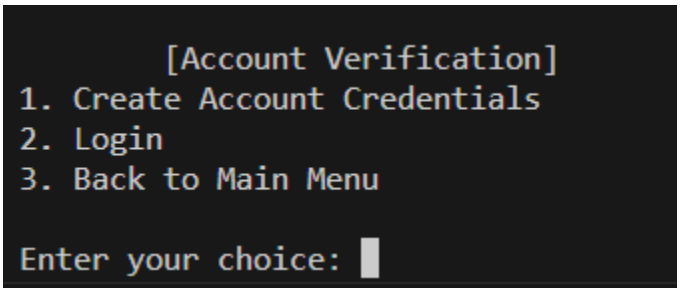
Display Account Details:



Transaction Menu:



Account Verification:



Conclusion

The Banking Management System demonstrates a comprehensive approach to handling various aspects of a banking system. The modular design with linked list, hash table, queue, and binary search tree ensures efficiency, organization, and security. The implementation adheres to good coding practices and provides a foundation for further enhancements and customization.