NUST College of Electrical & Mechanical Engineering, RAWALPINDI

Group members:

M. Shahryar Ameer (520657)

Seemal Rizwan (505272)

Degree/Dept: EE-46-A

CS – 250: Data Structures & Algorithms

Course Instructor: Dr. Ishfaq Hussain / Lab Instructor: Sir Moiz Shahid

Submission Deadline: 20 Dec 2025

Contents

Semester Project (Complex Engineering Problem)

Intelligent Bank management System

Introduction:

The Intelligent Bank Management and Simulation System (IBMS) is a data-structure–based software project designed to simulate real-world banking operations while demonstrating practical applications of core Data Structures and Algorithms (DSA). Modern banking systems require efficient handling of transactions, customer queues, and financial analysis, all of which rely heavily on optimized data handling techniques.

This project integrates financial transaction management, automated budget classification, and customer flow simulation into a single unified system. IBMS focuses on using fundamental and advanced data structures such as arrays, linked lists, stacks, queues, circular queues, and priority queues to manage banking activities efficiently. The system is implemented using structured programming techniques to ensure modularity, scalability, and clarity.

The IBMS project serves both as a learning tool for understanding DSA concepts and as a simulation model representing real-world banking scenarios.

Objectives of the Project:

The main objectives of the IBMS project are:

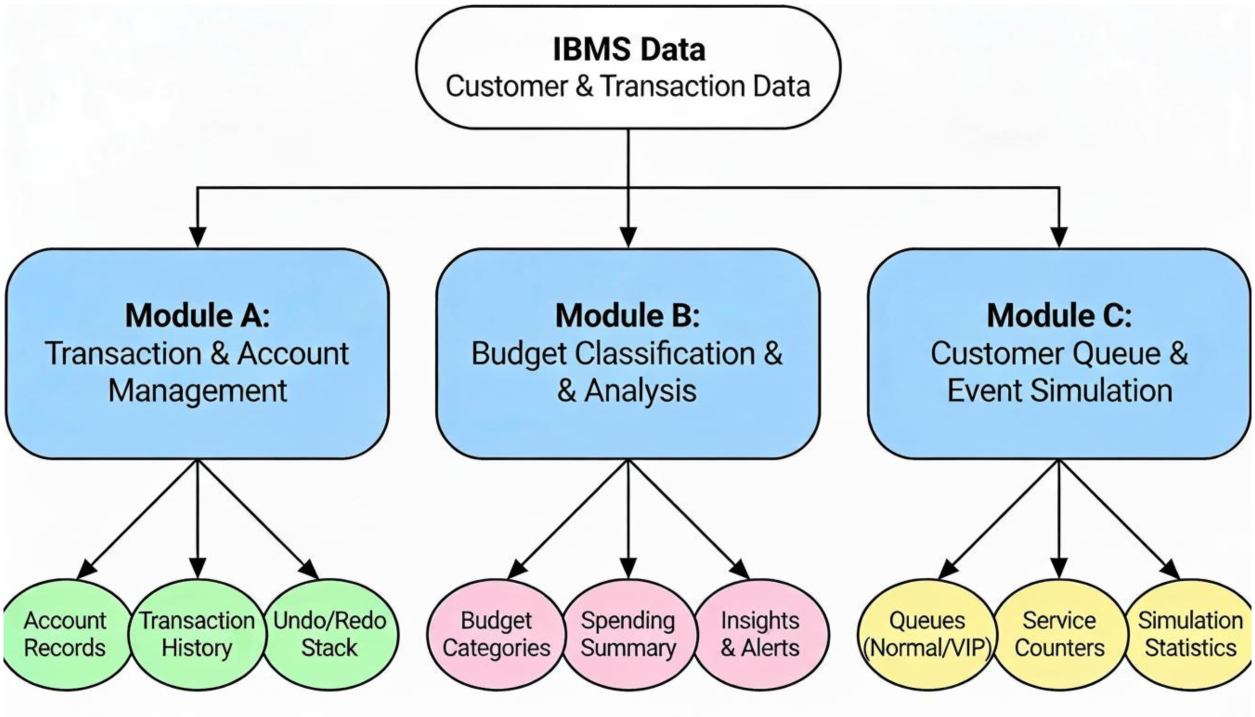
- To apply theoretical Data Structures and Algorithms concepts in a practical banking scenario
- To design a modular system handling transactions, budgets, and customer queues
- To simulate real-time customer flow inside a bank environment
- To demonstrate efficient searching, sorting, and classification techniques
- To strengthen understanding of recursion, stacks, and queues through real use cases
- To generate meaningful financial and operational outputs such as transaction history and queue reports

System Overview:

Account Management	Arrays	Store fixed customer account details

Transaction History	Linked List	Dynamic storage of transactions
Undo/Redo System	Stack	Reverse recent transactions
Transaction Search	Linear/Binary Search	Locate specific transactions
Sorting	Bubble / Selection / STL Sort	Sort transactions by amount/date
Budget Classification	Arrays + Recursion	Categorize transactions
Customer Queue	Queue	Normal customer handling
VIP Handling	Priority Queue	Higher priority service
Event Scheduling	Circular Queue	Time-based event simulation
Account Management	Arrays	Store fixed customer account details

Schematics of data:



Explanation of Each Module:

Module 1: Transaction & Account Management Module

Purpose

This module manages:

- Customer account information
- Deposits and withdrawals
- Transaction history
- Undo/redo functionality
- Operator Overloading

Implementation Details

1. Arrays

- Used to store **customer account data** (Account No, Name, Balance)

- Efficient for indexed access

Account accounts[MAX_CUSTOMERS];

---

2. Linked Lists

- Each account maintains a **linked list of transactions**
- Allows unlimited transaction history

Why Linked List?

- Dynamic memory allocation
- Efficient insertion without shifting data

---

3. Stacks (Undo / Redo)

- Stack stores recent transactions
- Undo pops the last transaction
- Redo restores a reverted transaction

DSA Concept Applied:
LIFO (Last In First Out)

---

4. Searching & Sorting

- Searching accounts using account number
- Sorting transactions by:
  - Date
  - Amount
  - Type (Debit/Credit)

---

5. Operator Overloading

- Used to compare transaction objects
- Enables easy comparison based on amount, date, or type
- Supports sorting and searching operations

- Improves code readability and structure
- Reduces complexity of comparison logic

Module 2: Budget Classification & Analysis Module

Purpose

This module automatically categorizes transactions into predefined budget classes such as:

- Food
- Utilities
- Education
- Government Payments
- Miscellaneous

Implementation Details

1. Arrays

- Store budget categories and keywords

```
string categories[] = {"Food", "Utilities", "Education"};
```

---

2. Searching

- Keyword-based matching in transaction description
- Example:
  - “Electric Bill” → Utilities
  - “University Fee” → Education

---

3. Recursion

- Used for **category prediction**
- Recursively checks keyword matches across categories

Why Recursion?

- Cleaner logic
- Demonstrates advanced DSA application

---

4. Operator Overloading

- Used to compare transactions or budget categories
- Enables readable comparison logic

Output

- Total spending per category
- Monthly expenditure analysis

Module 3: Customer Queue & Event Simulation Module

Purpose

Simulates customer flow in a bank environment including:

- Normal customers
- VIP customers
- Event scheduling
- Service counters

Implementation Details

1. Queue

- Handles **normal customers**
- FIFO behavior

---

2. Priority Queue

- Handles **VIP customers**
- Higher priority customers served first

3. Circular Queue

- Used for **event scheduling**
- Efficient memory reuse

4. Recursion

- Simulates nested events:
    - Customer arrival
    - Service completion
    - Counter availability

5. Activation Records

- Store parameters and local data for each recursive event call.
- Maintain return address to resume simulation correctly.
- Organized in a stack for nested event processing

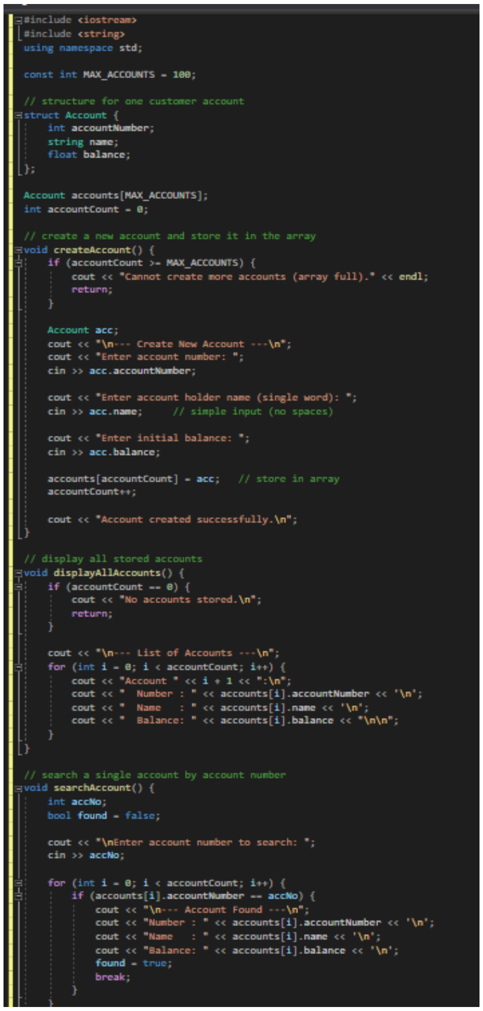
Program Implementation, Code and Results:

Module A: Transaction & Account Management

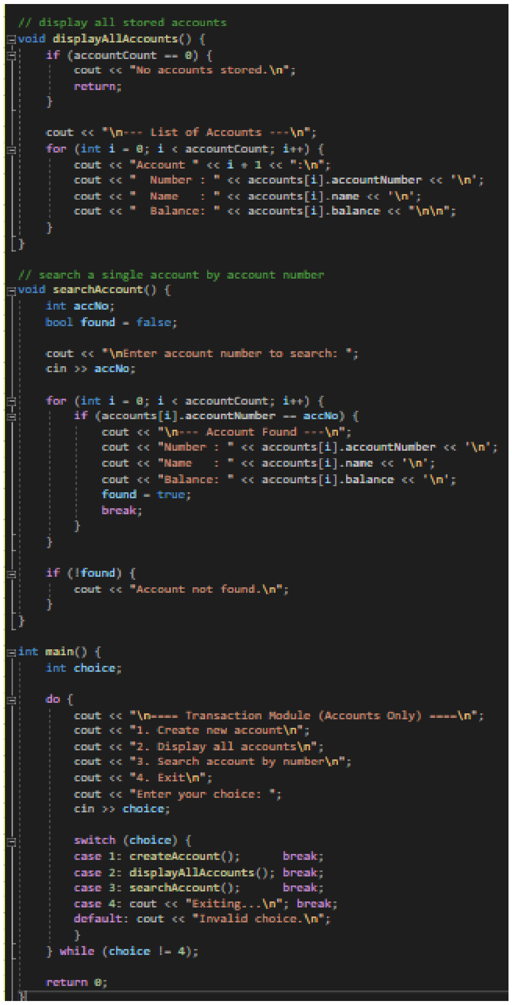
Task 1:

Arrays to store customer accounts

Code:

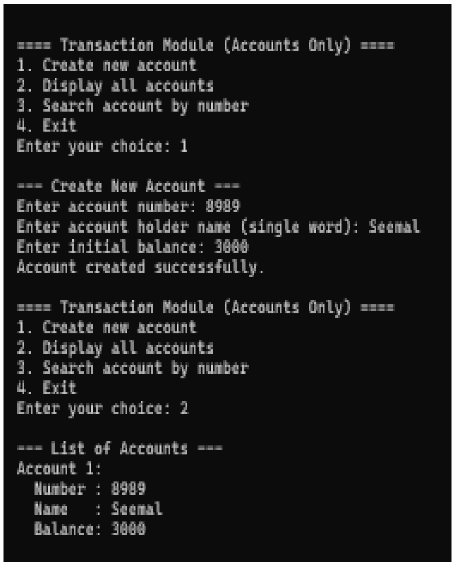


```cpp
#include <iostream>
#include <string>
using namespace std;

const int MAX_ACCOUNTS = 100;

// structure for one customer account
struct Account {
    int accountNumber;
    string name;
    float balance;
};

Account accounts[MAX_ACCOUNTS];
int accountCount = 0;

// create a new account and store it in the array
void createAccount() {
    if (accountCount >= MAX_ACCOUNTS) {
        cout << "Cannot create more accounts (array full)." << endl;
        return;
    }

    Account acc;
    cout << "\n--- Create New Account ---\n";
    cout << "Enter account number: ";
    cin >> acc.accountNumber;

    cout << "Enter account holder name (single word): ";
    cin >> acc.name;     // simple input (no spaces)

    cout << "Enter initial balance: ";
    cin >> acc.balance;

    accounts[accountCount] = acc;   // store in array
    accountCount++;

    cout << "Account created successfully.\n";
}

// display all stored accounts
void displayAllAccounts() {
    if (accountCount == 0) {
        cout << "No accounts stored.\n";
        return;
    }

    cout << "\n--- List of Accounts ---\n";
    for (int i = 0; i < accountCount; i++) {
        cout << "Account " << i + 1 << ":\n";
        cout << "  Number : " << accounts[i].accountNumber << '\n';
        cout << "  Name   : " << accounts[i].name << '\n';
        cout << "  Balance: " << accounts[i].balance << "\n\n";
    }
}

// search a single account by account number
void searchAccount() {
    int accNo;
    bool found = false;

    cout << "\nEnter account number to search: ";
    cin >> accNo;

    for (int i = 0; i < accountCount; i++) {
        if (accounts[i].accountNumber == accNo) {
            cout << "\n--- Account Found ---\n";
            cout << "Number : " << accounts[i].accountNumber << '\n';
            cout << "Name   : " << accounts[i].name << '\n';
            cout << "Balance: " << accounts[i].balance << '\n';
            found = true;
            break;
        }
    }
```

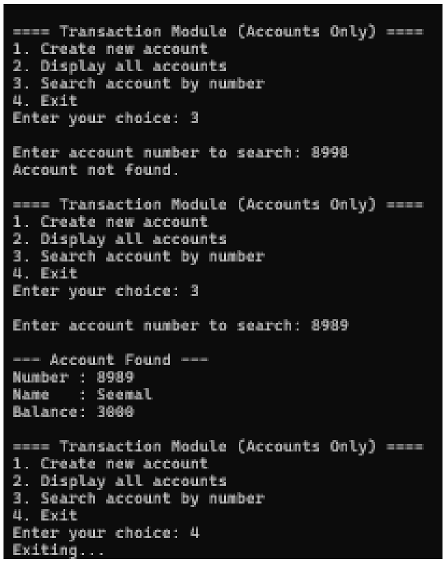


```cpp
// display all stored accounts
void displayAllAccounts() {
    if (accountCount == 0) {
        cout << "No accounts stored.\n";
        return;
    }

    cout << "\n--- List of Accounts ---\n";
    for (int i = 0; i < accountCount; i++) {
        cout << "Account " << i + 1 << ":\n";
        cout << "  Number : " << accounts[i].accountNumber << '\n';
        cout << "  Name   : " << accounts[i].name << '\n';
        cout << "  Balance: " << accounts[i].balance << "\n\n";
    }
}

// search a single account by account number
void searchAccount() {
    int accNo;
    bool found = false;

    cout << "\nEnter account number to search: ";
    cin >> accNo;

    for (int i = 0; i < accountCount; i++) {
        if (accounts[i].accountNumber == accNo) {
            cout << "\n--- Account Found ---\n";
            cout << "Number : " << accounts[i].accountNumber << '\n';
            cout << "Name   : " << accounts[i].name << '\n';
            cout << "Balance: " << accounts[i].balance << '\n';
            found = true;
            break;
        }
    }

    if (!found) {
        cout << "Account not found.\n";
    }
}

int main() {
    int choice;

    do {
        cout << "\n==== Transaction Module (Accounts Only) ====\n";
        cout << "1. Create new account\n";
        cout << "2. Display all accounts\n";
        cout << "3. Search account by number\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1: createAccount();      break;
        case 2: displayAllAccounts(); break;
        case 3: searchAccount();      break;
        case 4: cout << "Exiting...\n"; break;
        default: cout << "Invalid choice.\n";
        }
    } while (choice != 4);

    return 0;
}
```

Output:



```
==== Transaction Module (Accounts Only) ====
1. Create new account
2. Display all accounts
3. Search account by number
4. Exit
Enter your choice: 1

--- Create New Account ---
Enter account number: 8989
Enter account holder name (single word): Seemal
Enter initial balance: 3000
Account created successfully.

==== Transaction Module (Accounts Only) ====
1. Create new account
2. Display all accounts
3. Search account by number
4. Exit
Enter your choice: 2

--- List of Accounts ---
Account 1:
  Number : 8989
  Name   : Seemal
  Balance: 3000
```

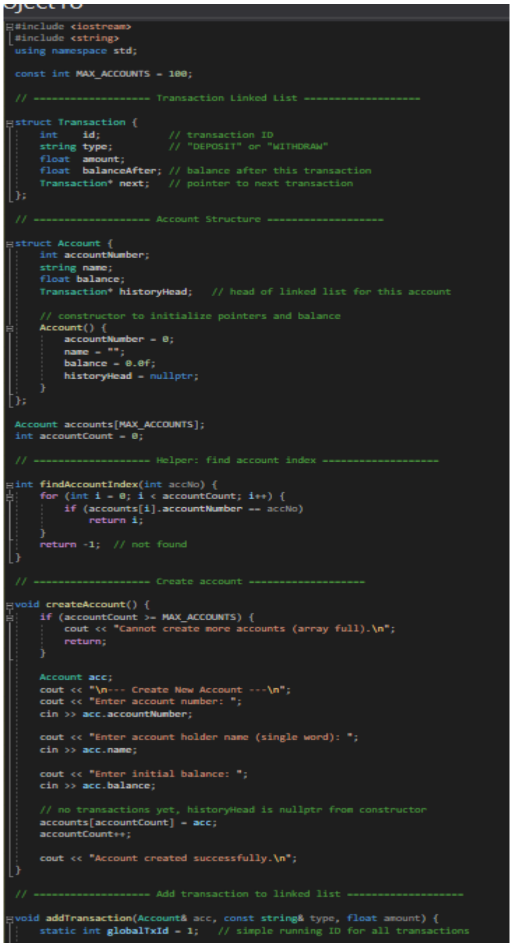


```
==== Transaction Module (Accounts Only) ====
1. Create new account
2. Display all accounts
3. Search account by number
4. Exit
Enter your choice: 3

Enter account number to search: 8998
Account not found.

==== Transaction Module (Accounts Only) ====
1. Create new account
2. Display all accounts
3. Search account by number
4. Exit
Enter your choice: 3

Enter account number to search: 8989

--- Account Found ---
Number : 8989
Name   : Seemal
Balance: 3000

==== Transaction Module (Accounts Only) ====
1. Create new account
2. Display all accounts
3. Search account by number
4. Exit
Enter your choice: 4
Exiting...
```

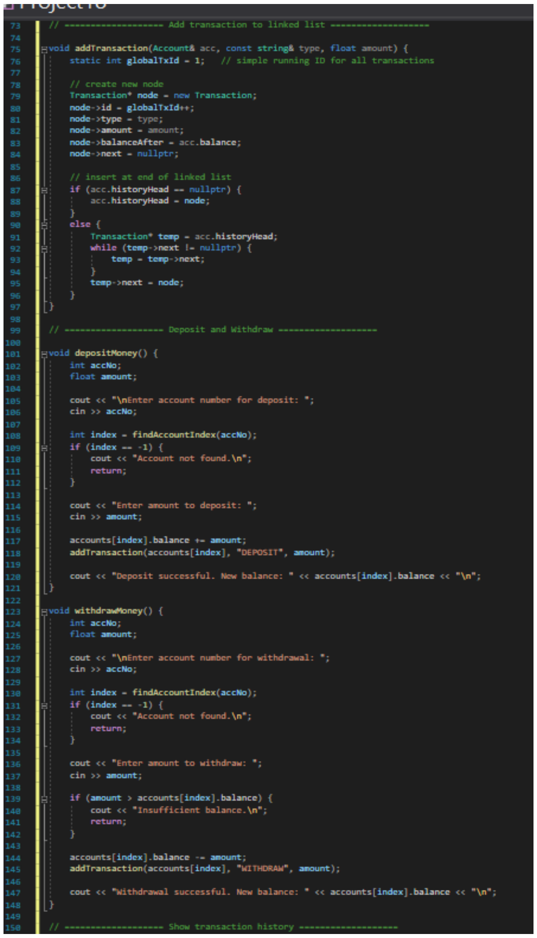
Task 2:

Linked Lists to dynamic transaction histories

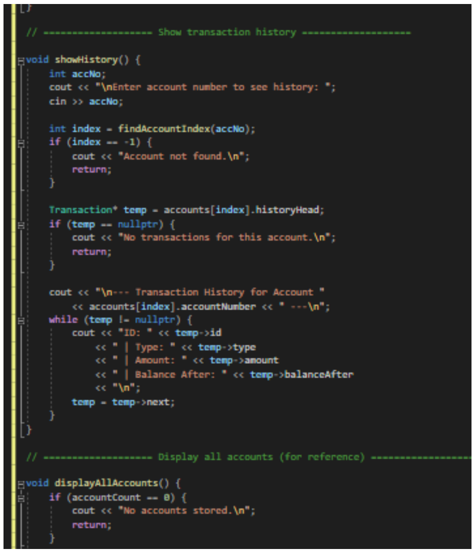
Code:



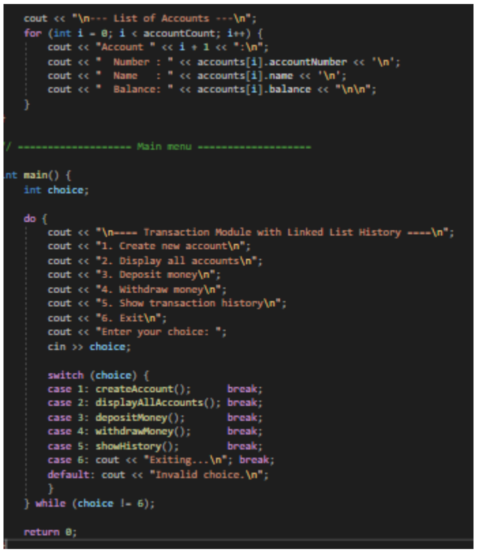
```cpp
#include <iostream>
#include <string>
using namespace std;

const int MAX_ACCOUNTS = 100;

// ------------------ Transaction Linked List ------------------

struct Transaction {
    int    id;             // transaction ID
    string type;           // "DEPOSIT" or "WITHDRAW"
    float  amount;
    float  balanceAfter; // balance after this transaction
    Transaction* next;   // pointer to next transaction
};

// ------------------ Account Structure ------------------

struct Account {
    int accountNumber;
    string name;
    float balance;
    Transaction* historyHead;   // head of linked list for this account

    // constructor to initialize pointers and balance
    Account() {
        accountNumber = 0;
        name = "";
        balance = 0.0f;
        historyHead = nullptr;
    }
};

Account accounts[MAX_ACCOUNTS];
int accountCount = 0;

// ------------------ Helper: find account index ------------------

int findAccountIndex(int accNo) {
    for (int i = 0; i < accountCount; i++) {
        if (accounts[i].accountNumber == accNo)
            return i;
    }
    return -1;  // not found
}

// ------------------ Create account ------------------

void createAccount() {
    if (accountCount >= MAX_ACCOUNTS) {
        cout << "Cannot create more accounts (array full).\n";
        return;
    }

    Account acc;
    cout << "\n--- Create New Account ---\n";
    cout << "Enter account number: ";
    cin >> acc.accountNumber;

    cout << "Enter account holder name (single word): ";
    cin >> acc.name;

    cout << "Enter initial balance: ";
    cin >> acc.balance;

    // no transactions yet, historyHead is nullptr from constructor
    accounts[accountCount] = acc;
    accountCount++;

    cout << "Account created successfully.\n";
}

// ------------------ Add transaction to linked list ------------------

void addTransaction(Account& acc, const string& type, float amount) {
    static int globalTxId = 1;   // simple running ID for all transactions
```



```cpp
// ------------------ Add transaction to linked list ------------------

void addTransaction(Account& acc, const string& type, float amount) {
    static int globalTxId = 1;   // simple running ID for all transactions

    // create new node
    Transaction* node = new Transaction;
    node->id = globalTxId++;
    node->type = type;
    node->amount = amount;
    node->balanceAfter = acc.balance;
    node->next = nullptr;

    // insert at end of linked list
    if (acc.historyHead == nullptr) {
        acc.historyHead = node;
    }
    else {
        Transaction* temp = acc.historyHead;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = node;
    }
}

// ------------------ Deposit and Withdraw ------------------

void depositMoney() {
    int accNo;
    float amount;

    cout << "\nEnter account number for deposit: ";
    cin >> accNo;

    int index = findAccountIndex(accNo);
    if (index == -1) {
        cout << "Account not found.\n";
        return;
    }

    cout << "Enter amount to deposit: ";
    cin >> amount;

    accounts[index].balance += amount;
    addTransaction(accounts[index], "DEPOSIT", amount);

    cout << "Deposit successful. New balance: " << accounts[index].balance << "\n";
}

void withdrawMoney() {
    int accNo;
    float amount;

    cout << "\nEnter account number for withdrawal: ";
    cin >> accNo;

    int index = findAccountIndex(accNo);
    if (index == -1) {
        cout << "Account not found.\n";
        return;
    }

    cout << "Enter amount to withdraw: ";
    cin >> amount;

    if (amount > accounts[index].balance) {
        cout << "Insufficient balance.\n";
        return;
    }

    accounts[index].balance -= amount;
    addTransaction(accounts[index], "WITHDRAW", amount);

    cout << "Withdrawal successful. New balance: " << accounts[index].balance << "\n";
}

// ------------------ Show transaction history ------------------
```

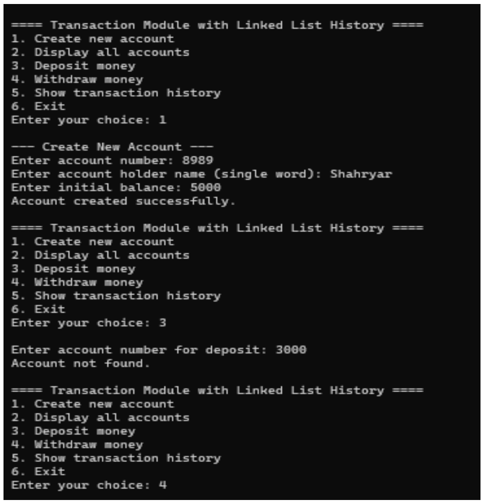


```cpp
}

// ------------------ Show transaction history ------------------

void showHistory() {
    int accNo;
    cout << "\nEnter account number to see history: ";
    cin >> accNo;

    int index = findAccountIndex(accNo);
    if (index == -1) {
        cout << "Account not found.\n";
        return;
    }

    Transaction* temp = accounts[index].historyHead;
    if (temp == nullptr) {
        cout << "No transactions for this account.\n";
        return;
    }

    cout << "\n--- Transaction History for Account "
        << accounts[index].accountNumber << " ---\n";
    while (temp != nullptr) {
        cout << "ID: " << temp->id
            << " | Type: " << temp->type
            << " | Amount: " << temp->amount
            << " | Balance After: " << temp->balanceAfter
            << "\n";
        temp = temp->next;
    }
}

// ------------------ Display all accounts (for reference) ------------------

void displayAllAccounts() {
    if (accountCount == 0) {
        cout << "No accounts stored.\n";
        return;
    }
```

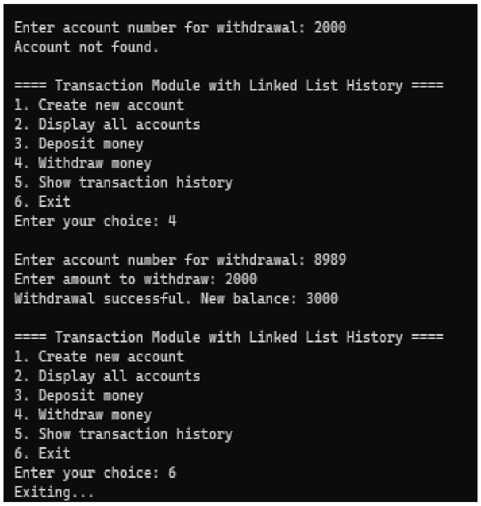


```cpp
    cout << "\n--- List of Accounts ---\n";
    for (int i = 0; i < accountCount; i++) {
        cout << "Account " << i + 1 << ":\n";
        cout << "  Number : " << accounts[i].accountNumber << '\n';
        cout << "  Name   : " << accounts[i].name << '\n';
        cout << "  Balance: " << accounts[i].balance << "\n\n";
    }
}

/ ------------------ Main menu ------------------

nt main() {
    int choice;

    do {
        cout << "\n==== Transaction Module with Linked List History ====\n";
        cout << "1. Create new account\n";
        cout << "2. Display all accounts\n";
        cout << "3. Deposit money\n";
        cout << "4. Withdraw money\n";
        cout << "5. Show transaction history\n";
        cout << "6. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1: createAccount();      break;
        case 2: displayAllAccounts(); break;
        case 3: depositMoney();       break;
        case 4: withdrawMoney();      break;
        case 5: showHistory();        break;
        case 6: cout << "Exiting...\n"; break;
        default: cout << "Invalid choice.\n";
        }
    } while (choice != 6);

    return 0;
```

Output:



```
==== Transaction Module with Linked List History ====
1. Create new account
2. Display all accounts
3. Deposit money
4. Withdraw money
5. Show transaction history
6. Exit
Enter your choice: 1

--- Create New Account ---
Enter account number: 8989
Enter account holder name (single word): Shahryar
Enter initial balance: 5000
Account created successfully.

==== Transaction Module with Linked List History ====
1. Create new account
2. Display all accounts
3. Deposit money
4. Withdraw money
5. Show transaction history
6. Exit
Enter your choice: 3

Enter account number for deposit: 3000
Account not found.

==== Transaction Module with Linked List History ====
1. Create new account
2. Display all accounts
3. Deposit money
4. Withdraw money
5. Show transaction history
6. Exit
Enter your choice: 4
```

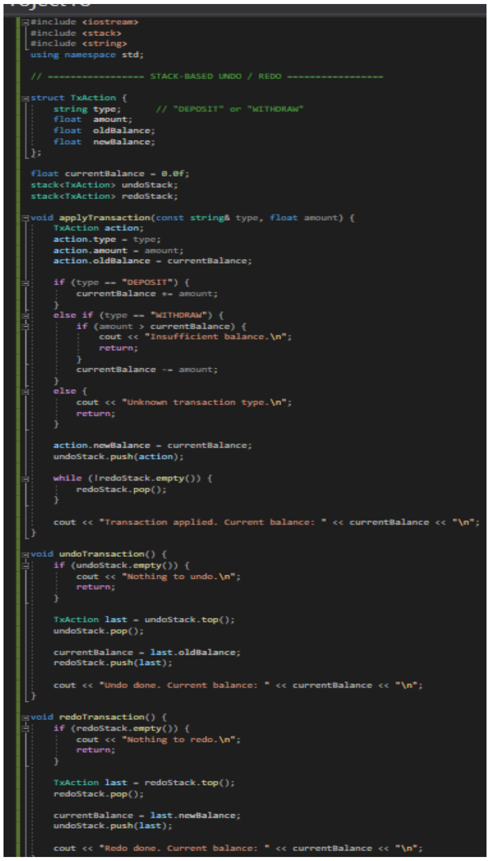


```
Enter account number for withdrawal: 2000
Account not found.

==== Transaction Module with Linked List History ====
1. Create new account
2. Display all accounts
3. Deposit money
4. Withdraw money
5. Show transaction history
6. Exit
Enter your choice: 4

Enter account number for withdrawal: 8989
Enter amount to withdraw: 2000
Withdrawal successful. New balance: 3000

==== Transaction Module with Linked List History ====
1. Create new account
2. Display all accounts
3. Deposit money
4. Withdraw money
5. Show transaction history
6. Exit
Enter your choice: 6
Exiting...
```

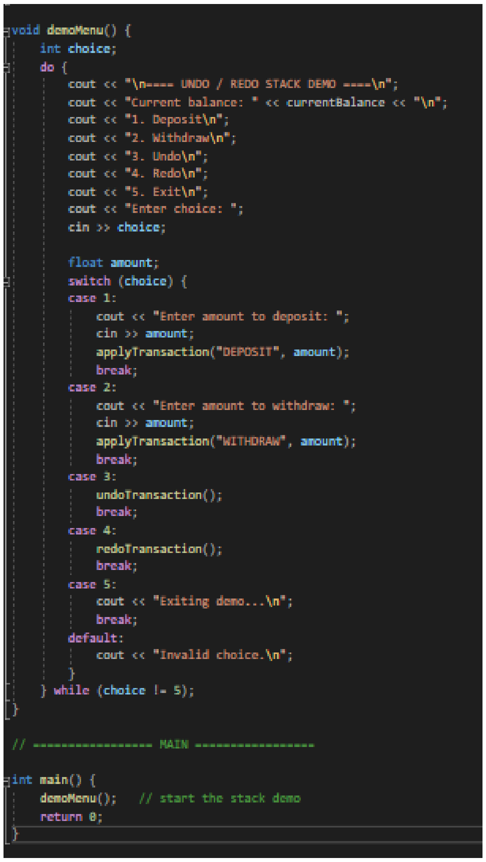
Task 3:

Stacks to implement undo/redo for transactions

Code:

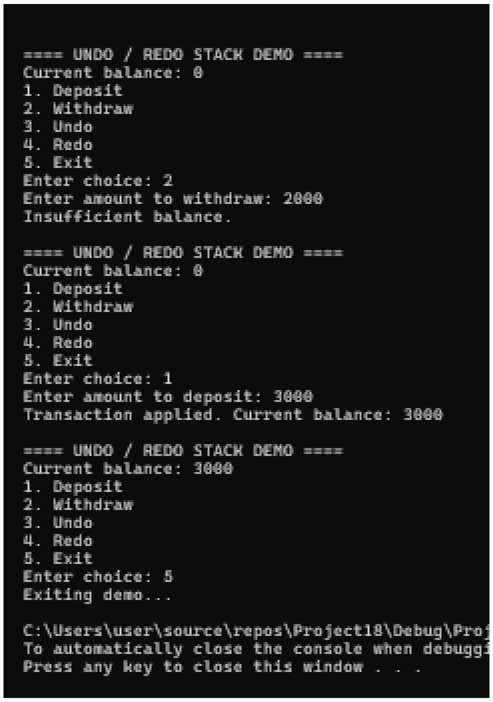


```cpp
#include <iostream>
#include <stack>
#include <string>
using namespace std;

// ---------------- STACK-BASED UNDO / REDO ----------------

struct TxAction {
    string type;      // "DEPOSIT" or "WITHDRAW"
    float  amount;
    float  oldBalance;
    float  newBalance;
};

float currentBalance = 0.0f;
stack<TxAction> undoStack;
stack<TxAction> redoStack;

void applyTransaction(const string& type, float amount) {
    TxAction action;
    action.type = type;
    action.amount = amount;
    action.oldBalance = currentBalance;

    if (type == "DEPOSIT") {
        currentBalance += amount;
    }
    else if (type == "WITHDRAW") {
        if (amount > currentBalance) {
            cout << "Insufficient balance.\n";
            return;
        }
        currentBalance -= amount;
    }
    else {
        cout << "Unknown transaction type.\n";
        return;
    }

    action.newBalance = currentBalance;
    undoStack.push(action);

    while (!redoStack.empty()) {
        redoStack.pop();
    }

    cout << "Transaction applied. Current balance: " << currentBalance << "\n";
}

void undoTransaction() {
    if (undoStack.empty()) {
        cout << "Nothing to undo.\n";
        return;
    }

    TxAction last = undoStack.top();
    undoStack.pop();

    currentBalance = last.oldBalance;
    redoStack.push(last);

    cout << "Undo done. Current balance: " << currentBalance << "\n";
}

void redoTransaction() {
    if (redoStack.empty()) {
        cout << "Nothing to redo.\n";
        return;
    }

    TxAction last = redoStack.top();
    redoStack.pop();

    currentBalance = last.newBalance;
    undoStack.push(last);

    cout << "Redo done. Current balance: " << currentBalance << "\n";
```



```cpp
void demoMenu() {
    int choice;
    do {
        cout << "\n==== UNDO / REDO STACK DEMO ====\n";
        cout << "Current balance: " << currentBalance << "\n";
        cout << "1. Deposit\n";
        cout << "2. Withdraw\n";
        cout << "3. Undo\n";
        cout << "4. Redo\n";
        cout << "5. Exit\n";
        cout << "Enter choice: ";
        cin >> choice;

        float amount;
        switch (choice) {
        case 1:
            cout << "Enter amount to deposit: ";
            cin >> amount;
            applyTransaction("DEPOSIT", amount);
            break;
        case 2:
            cout << "Enter amount to withdraw: ";
            cin >> amount;
            applyTransaction("WITHDRAW", amount);
            break;
        case 3:
            undoTransaction();
            break;
        case 4:
            redoTransaction();
            break;
        case 5:
            cout << "Exiting demo...\n";
            break;
        default:
            cout << "Invalid choice.\n";
        }
    } while (choice != 5);
}

// ================ MAIN ================

int main() {
    demoMenu();   // start the stack demo
    return 0;
}
```

Output:

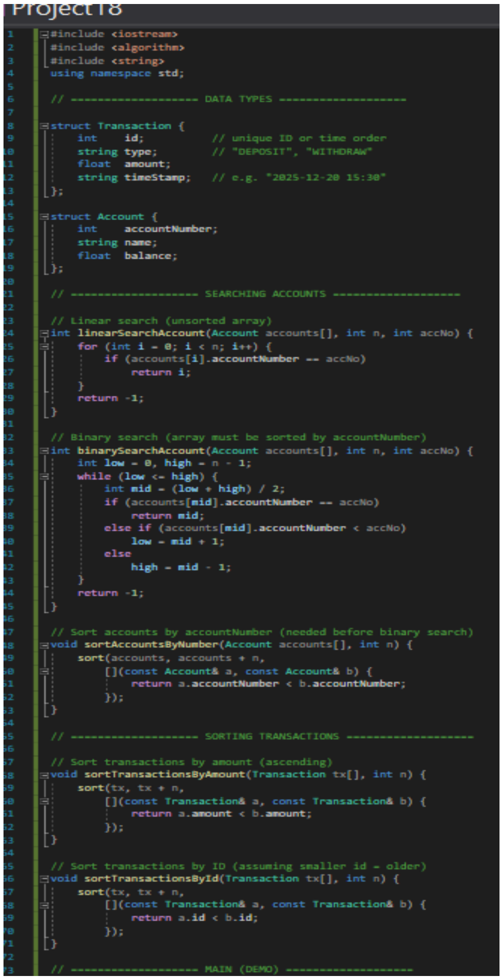


```
==== UNDO / REDO STACK DEMO ====
Current balance: 0
1. Deposit
2. Withdraw
3. Undo
4. Redo
5. Exit
Enter choice: 2
Enter amount to withdraw: 2000
Insufficient balance.

==== UNDO / REDO STACK DEMO ====
Current balance: 0
1. Deposit
2. Withdraw
3. Undo
4. Redo
5. Exit
Enter choice: 1
Enter amount to deposit: 3000
Transaction applied. Current balance: 3000

==== UNDO / REDO STACK DEMO ====
Current balance: 3000
1. Deposit
2. Withdraw
3. Undo
4. Redo
5. Exit
Enter choice: 5
Exiting demo...

C:\Users\user\source\repos\Project18\Debug\Proj
To automatically close the console when debuggi
Press any key to close this window . . .
```

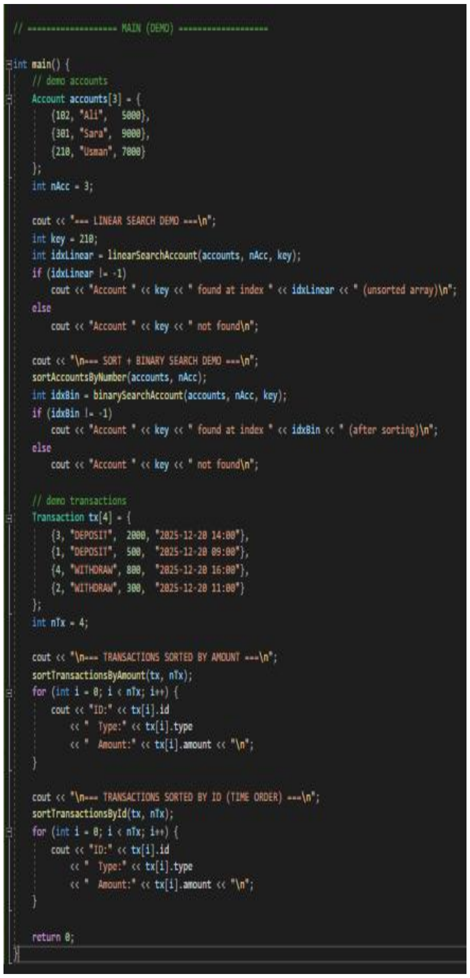
Task 4:

Sorting/Searching to search accounts, sort transactions

Code:

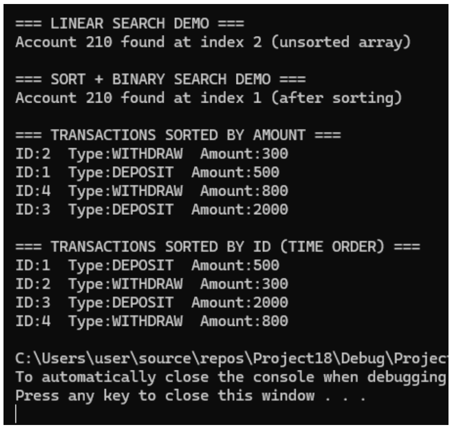


```cpp
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

// ------------------- DATA TYPES -------------------

struct Transaction {
    int    id;           // unique ID or time order
    string type;         // "DEPOSIT", "WITHDRAW"
    float  amount;
    string timeStamp;    // e.g. "2025-12-20 15:30"
};

struct Account {
    int    accountNumber;
    string name;
    float  balance;
};

// ------------------- SEARCHING ACCOUNTS -------------------

// Linear search (unsorted array)
int linearSearchAccount(Account accounts[], int n, int accNo) {
    for (int i = 0; i < n; i++) {
        if (accounts[i].accountNumber == accNo)
            return i;
    }
    return -1;
}

// Binary search (array must be sorted by accountNumber)
int binarySearchAccount(Account accounts[], int n, int accNo) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (accounts[mid].accountNumber == accNo)
            return mid;
        else if (accounts[mid].accountNumber < accNo)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

// Sort accounts by accountNumber (needed before binary search)
void sortAccountsByNumber(Account accounts[], int n) {
    sort(accounts, accounts + n,
        [](const Account& a, const Account& b) {
            return a.accountNumber < b.accountNumber;
        });
}

// ------------------- SORTING TRANSACTIONS -------------------

// Sort transactions by amount (ascending)
void sortTransactionsByAmount(Transaction tx[], int n) {
    sort(tx, tx + n,
        [](const Transaction& a, const Transaction& b) {
            return a.amount < b.amount;
        });
}

// Sort transactions by ID (assuming smaller id = older)
void sortTransactionsById(Transaction tx[], int n) {
    sort(tx, tx + n,
        [](const Transaction& a, const Transaction& b) {
            return a.id < b.id;
        });
}

// ------------------- MAIN (DEMO) -------------------
```



```cpp
// ------------------- MAIN (DEMO) -------------------

int main() {
    // demo accounts
    Account accounts[3] = {
        {102, "Ali",   5000},
        {301, "Sara",  9000},
        {210, "Usman", 7000}
    };
    int nAcc = 3;

    cout << "=== LINEAR SEARCH DEMO ===\n";
    int key = 210;
    int idxLinear = linearSearchAccount(accounts, nAcc, key);
    if (idxLinear != -1)
        cout << "Account " << key << " found at index " << idxLinear << " (unsorted array)\n";
    else
        cout << "Account " << key << " not found\n";

    cout << "\n=== SORT + BINARY SEARCH DEMO ===\n";
    sortAccountsByNumber(accounts, nAcc);
    int idxBin = binarySearchAccount(accounts, nAcc, key);
    if (idxBin != -1)
        cout << "Account " << key << " found at index " << idxBin << " (after sorting)\n";
    else
        cout << "Account " << key << " not found\n";

    // demo transactions
    Transaction tx[4] = {
        {3, "DEPOSIT",  2000, "2025-12-20 14:00"},
        {1, "DEPOSIT",  500,  "2025-12-20 09:00"},
        {4, "WITHDRAW", 800,  "2025-12-20 16:00"},
        {2, "WITHDRAW", 300,  "2025-12-20 11:00"}
    };
    int nTx = 4;

    cout << "\n=== TRANSACTIONS SORTED BY AMOUNT ===\n";
    sortTransactionsByAmount(tx, nTx);
    for (int i = 0; i < nTx; i++) {
        cout << "ID:" << tx[i].id
            << "  Type:" << tx[i].type
            << "  Amount:" << tx[i].amount << "\n";
    }

    cout << "\n=== TRANSACTIONS SORTED BY ID (TIME ORDER) ===\n";
    sortTransactionsById(tx, nTx);
    for (int i = 0; i < nTx; i++) {
        cout << "ID:" << tx[i].id
            << "  Type:" << tx[i].type
            << "  Amount:" << tx[i].amount << "\n";
    }

    return 0;
}
```

Output:

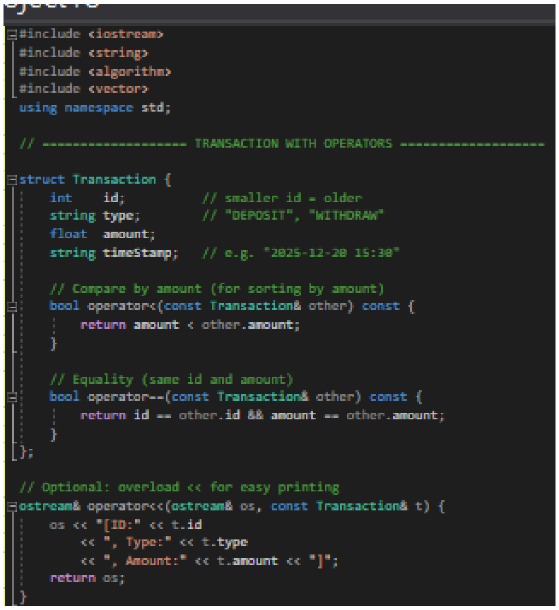


```
=== LINEAR SEARCH DEMO ===
Account 210 found at index 2 (unsorted array)

=== SORT + BINARY SEARCH DEMO ===
Account 210 found at index 1 (after sorting)

=== TRANSACTIONS SORTED BY AMOUNT ===
ID:2  Type:WITHDRAW  Amount:300
ID:1  Type:DEPOSIT  Amount:500
ID:4  Type:WITHDRAW  Amount:800
ID:3  Type:DEPOSIT  Amount:2000

=== TRANSACTIONS SORTED BY ID (TIME ORDER) ===
ID:1  Type:DEPOSIT  Amount:500
ID:2  Type:WITHDRAW  Amount:300
ID:3  Type:DEPOSIT  Amount:2000
ID:4  Type:WITHDRAW  Amount:800

C:\Users\user\source\repos\Project18\Debug\Projec
To automatically close the console when debugging
Press any key to close this window . . .
```

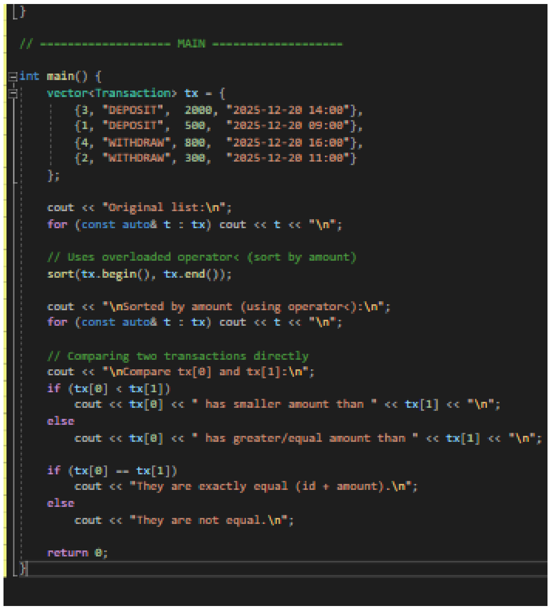
Task 5:

Operator Overloading for comparing transaction objects

Code:

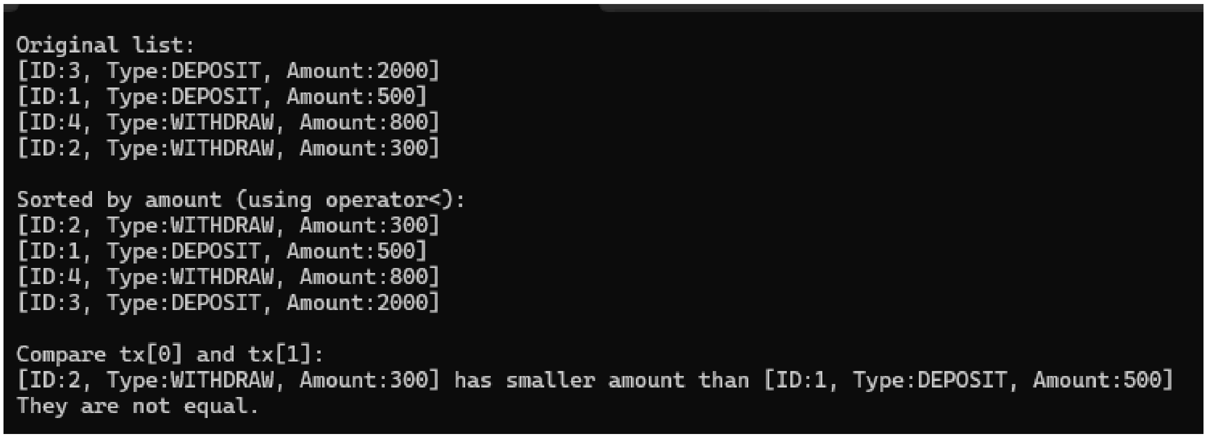


```cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

// ------------------ TRANSACTION WITH OPERATORS ------------------

struct Transaction {
    int    id;            // smaller id = older
    string type;          // "DEPOSIT", "WITHDRAW"
    float  amount;
    string timeStamp;     // e.g. "2025-12-20 15:30"

    // Compare by amount (for sorting by amount)
    bool operator<(const Transaction& other) const {
        return amount < other.amount;
    }

    // Equality (same id and amount)
    bool operator==(const Transaction& other) const {
        return id == other.id && amount == other.amount;
    }
};

// Optional: overload << for easy printing
ostream& operator<<(ostream& os, const Transaction& t) {
    os << "[ID:" << t.id
        << ", Type:" << t.type
        << ", Amount:" << t.amount << "]";
    return os;
}
```



```cpp
}

// ------------------ MAIN ------------------

int main() {
    vector<Transaction> tx = {
        {3, "DEPOSIT",  2000, "2025-12-20 14:00"},
        {1, "DEPOSIT",  500,  "2025-12-20 09:00"},
        {4, "WITHDRAW", 800,  "2025-12-20 16:00"},
        {2, "WITHDRAW", 300,  "2025-12-20 11:00"}
    };

    cout << "Original list:\n";
    for (const auto& t : tx) cout << t << "\n";

    // Uses overloaded operator< (sort by amount)
    sort(tx.begin(), tx.end());

    cout << "\nSorted by amount (using operator<):\n";
    for (const auto& t : tx) cout << t << "\n";

    // Comparing two transactions directly
    cout << "\nCompare tx[0] and tx[1]:\n";
    if (tx[0] < tx[1])
        cout << tx[0] << " has smaller amount than " << tx[1] << "\n";
    else
        cout << tx[0] << " has greater/equal amount than " << tx[1] << "\n";

    if (tx[0] == tx[1])
        cout << "They are exactly equal (id + amount).\n";
    else
        cout << "They are not equal.\n";

    return 0;
}
```

Output:



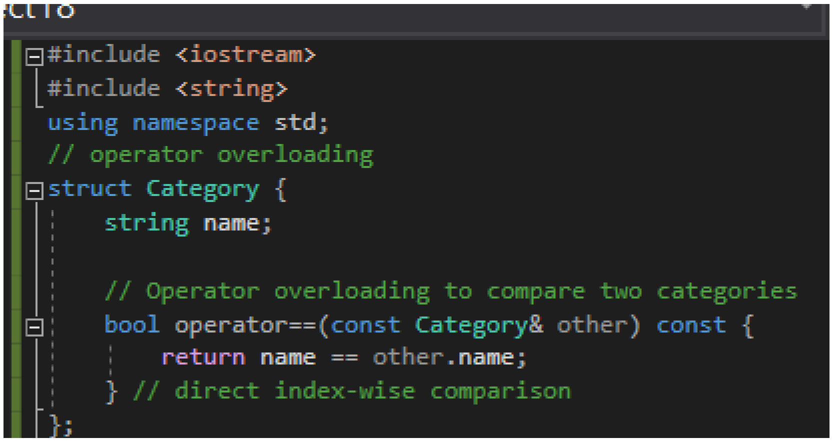
```
Original list:
[ID:3, Type:DEPOSIT, Amount:2000]
[ID:1, Type:DEPOSIT, Amount:500]
[ID:4, Type:WITHDRAW, Amount:800]
[ID:2, Type:WITHDRAW, Amount:300]

Sorted by amount (using operator<):
[ID:2, Type:WITHDRAW, Amount:300]
[ID:1, Type:DEPOSIT, Amount:500]
[ID:4, Type:WITHDRAW, Amount:800]
[ID:3, Type:DEPOSIT, Amount:2000]

Compare tx[0] and tx[1]:
[ID:2, Type:WITHDRAW, Amount:300] has smaller amount than [ID:1, Type:DEPOSIT, Amount:500]
They are not equal.
```

Module B: Budget Classification & Analytics Engine

Task1:

Arrays for categorization.

Code:

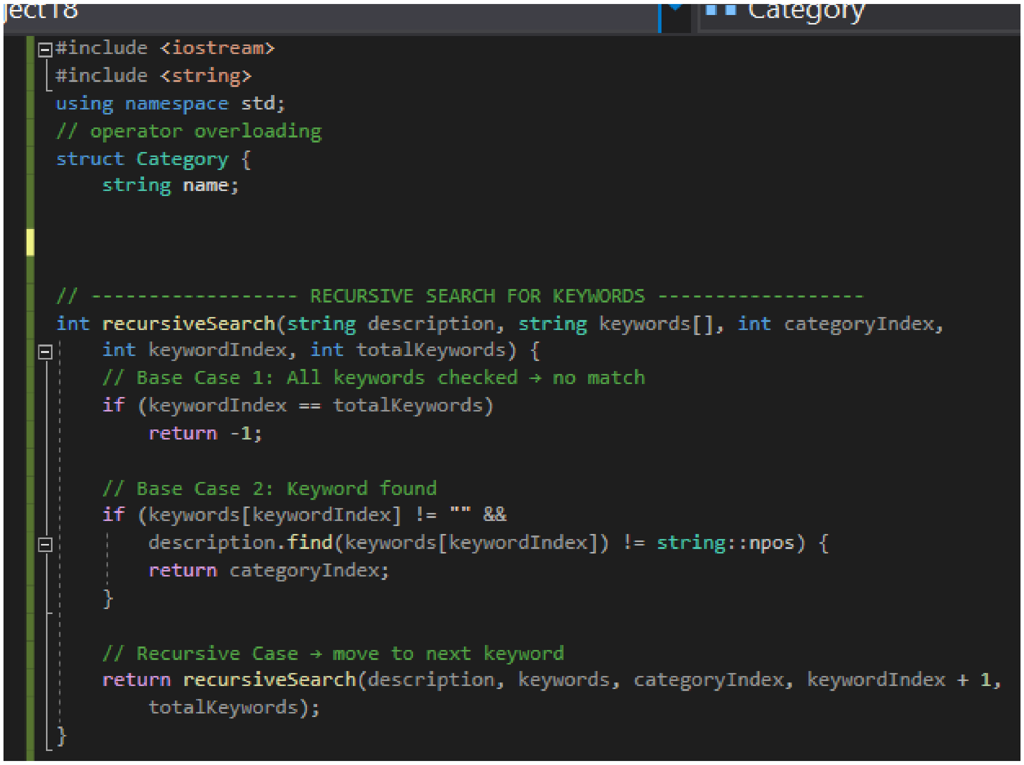


```
#include <iostream>
#include <string>
using namespace std;
// operator overloading
struct Category {
    string name;

    // Operator overloading to compare two categories
    bool operator==(const Category& other) const {
        return name == other.name;
    } // direct index-wise comparison
};
```

Task2:

Searching implemented using keyword-based and rule-based classification.

Code:

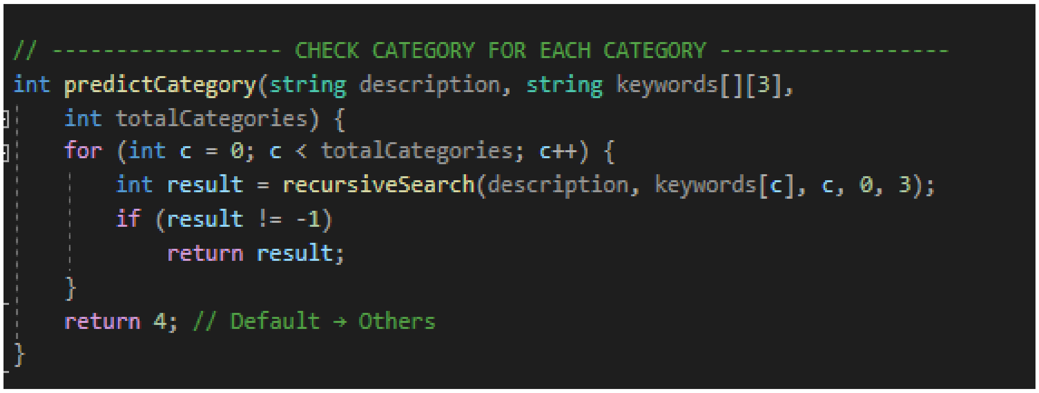


```
#include <iostream>
#include <string>
using namespace std;
// operator overloading
struct Category {
    string name;



// ------------------ RECURSIVE SEARCH FOR KEYWORDS ------------------
int recursiveSearch(string description, string keywords[], int categoryIndex,
    int keywordIndex, int totalKeywords) {
    // Base Case 1: All keywords checked → no match
    if (keywordIndex == totalKeywords)
        return -1;

    // Base Case 2: Keyword found
    if (keywords[keywordIndex] != "" &&
        description.find(keywords[keywordIndex]) != string::npos) {
        return categoryIndex;
    }

    // Recursive Case → move to next keyword
    return recursiveSearch(description, keywords, categoryIndex, keywordIndex + 1,
        totalKeywords);
}
```

Task3:

Recursion for category prediction algorithm.

Code:

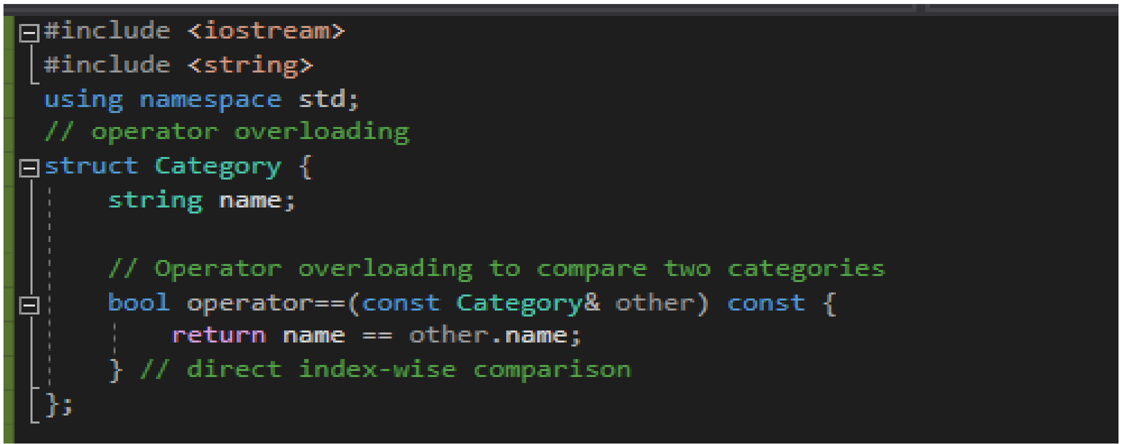


```
// ------------------ CHECK CATEGORY FOR EACH CATEGORY ------------------
int predictCategory(string description, string keywords[][3],
    int totalCategories) {
    for (int c = 0; c < totalCategories; c++) {
        int result = recursiveSearch(description, keywords[c], c, 0, 3);
        if (result != -1)
            return result;
    }
    return 4; // Default → Others
}
```

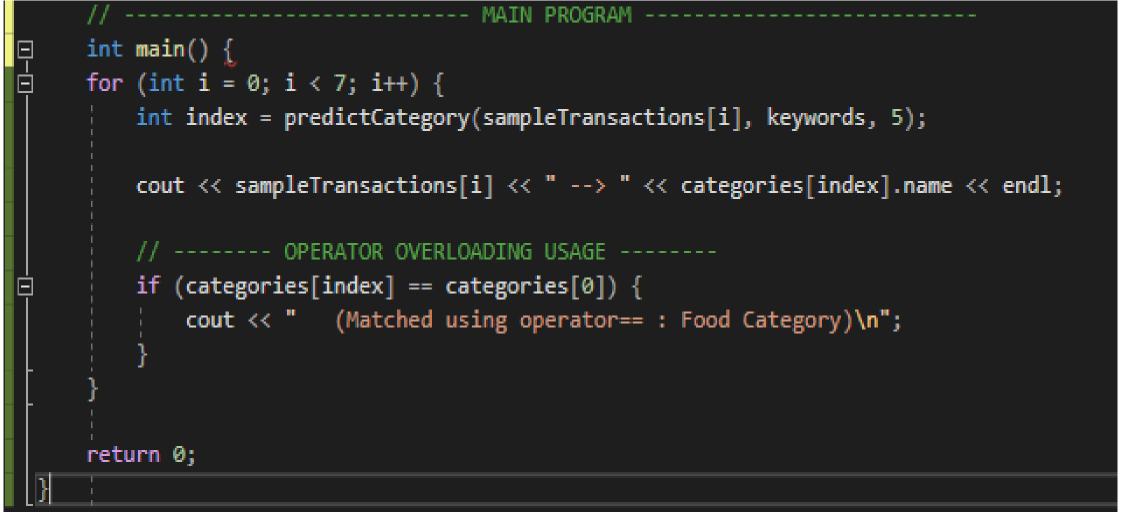
Task4:

Operator overloading for comparing categories.

Code:

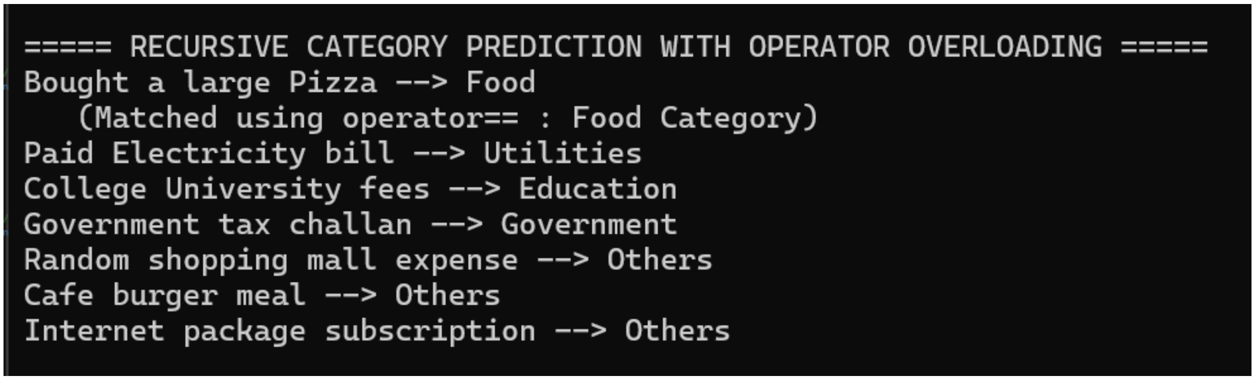


```
#include <iostream>
#include <string>
using namespace std;
// operator overloading
struct Category {
    string name;

    // Operator overloading to compare two categories
    bool operator==(const Category& other) const {
        return name == other.name;
    } // direct index-wise comparison
};
```



```
// --------------------------- MAIN PROGRAM ---------------------------
int main() {
    for (int i = 0; i < 7; i++) {
        int index = predictCategory(sampleTransactions[i], keywords, 5);

        cout << sampleTransactions[i] << " --> " << categories[index].name << endl;

        // -------- OPERATOR OVERLOADING USAGE --------
        if (categories[index] == categories[0]) {
            cout << "   (Matched using operator== : Food Category)\n";
        }
    }

    return 0;
}
```

Combined Output:



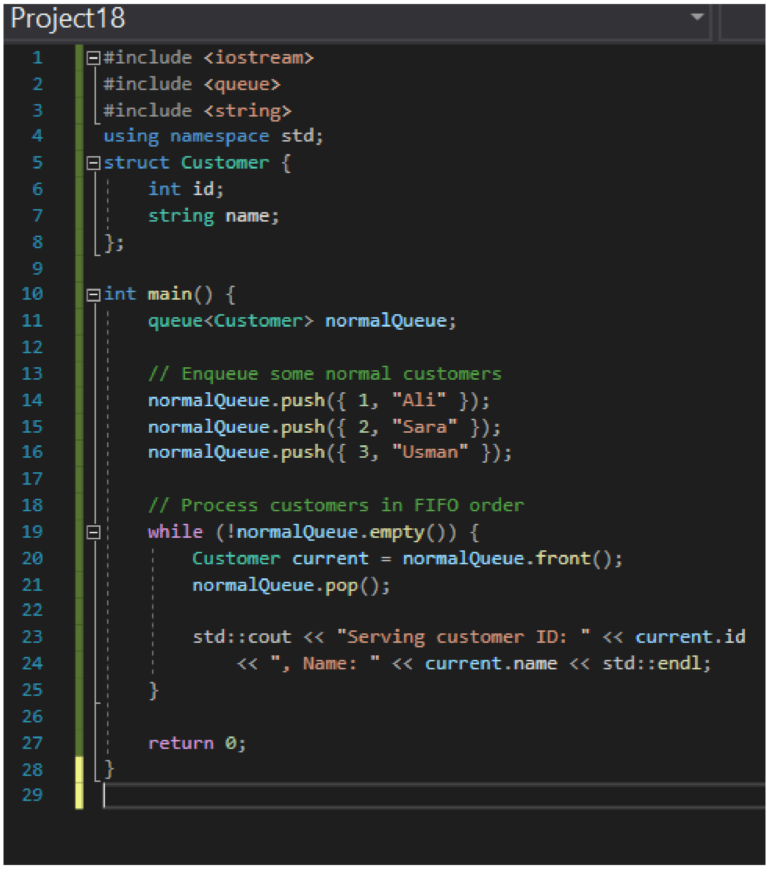
```
===== RECURSIVE CATEGORY PREDICTION WITH OPERATOR OVERLOADING =====
Bought a large Pizza --> Food
   (Matched using operator== : Food Category)
Paid Electricity bill --> Utilities
College University fees --> Education
Government tax challan --> Government
Random shopping mall expense --> Others
Cafe burger meal --> Others
Internet package subscription --> Others
```

Module C: Customer Queue & Event Simulation Module

Task1:

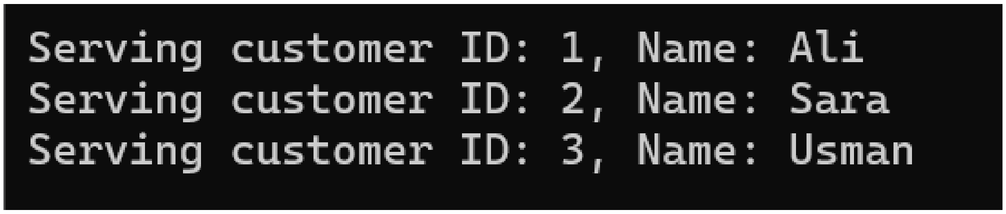
Queue for normal customers

Code:



```cpp
#include <iostream>
#include <queue>
#include <string>
using namespace std;
struct Customer {
    int id;
    string name;
};

int main() {
    queue<Customer> normalQueue;

    // Enqueue some normal customers
    normalQueue.push({ 1, "Ali" });
    normalQueue.push({ 2, "Sara" });
    normalQueue.push({ 3, "Usman" });

    // Process customers in FIFO order
    while (!normalQueue.empty()) {
        Customer current = normalQueue.front();
        normalQueue.pop();

        std::cout << "Serving customer ID: " << current.id
            << ", Name: " << current.name << std::endl;
    }

    return 0;
}

```

Output:

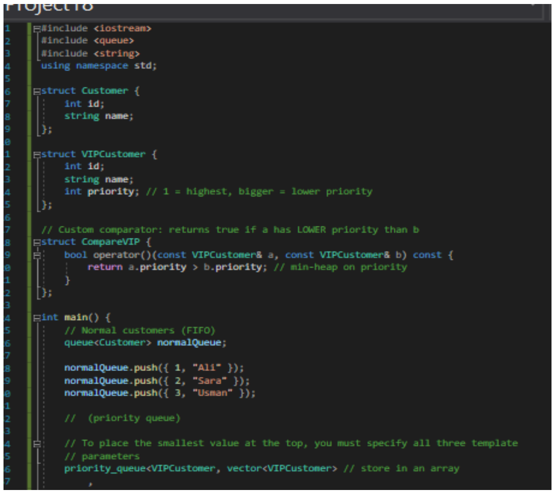


```
Serving customer ID: 1, Name: Ali
Serving customer ID: 2, Name: Sara
Serving customer ID: 3, Name: Usman
```

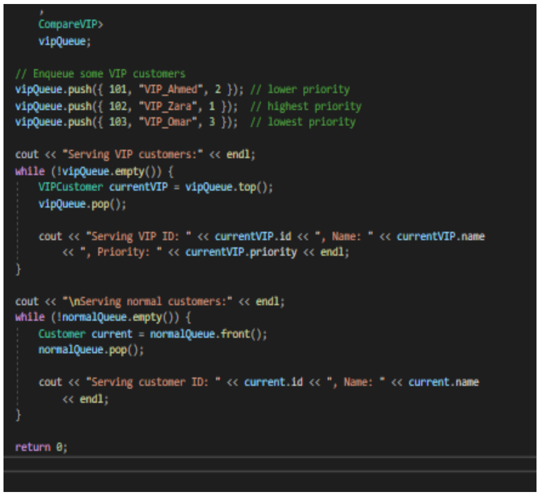
Task2:

Priority Queue for VIP customers

Code:

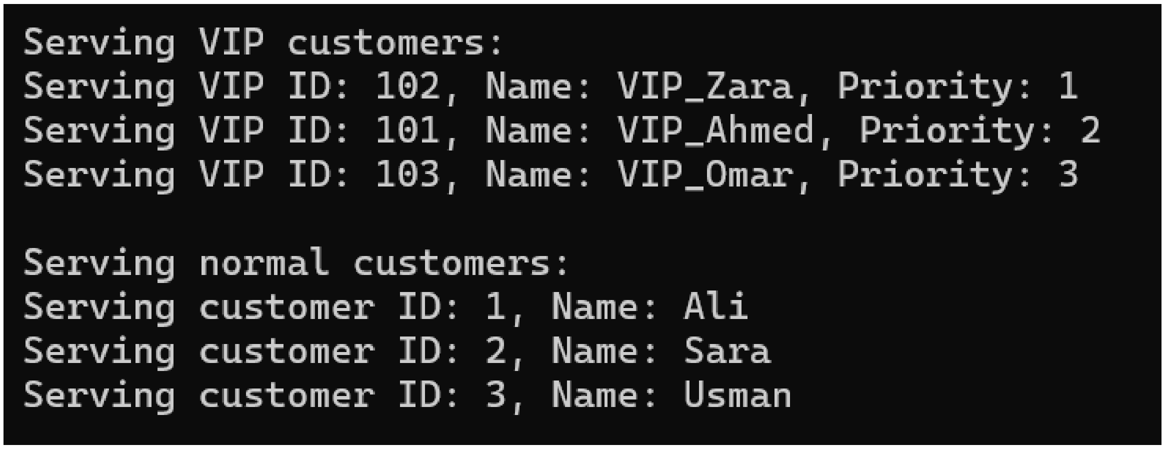


```cpp
#include <iostream>
#include <queue>
#include <string>
using namespace std;

struct Customer {
    int id;
    string name;
};

struct VIPCustomer {
    int id;
    string name;
    int priority; // 1 = highest, bigger = lower priority
};

// Custom comparator: returns true if a has LOWER priority than b
struct CompareVIP {
    bool operator()(const VIPCustomer& a, const VIPCustomer& b) const {
        return a.priority > b.priority; // min-heap on priority
    }
};

int main() {
    // Normal customers (FIFO)
    queue<Customer> normalQueue;

    normalQueue.push({ 1, "Ali" });
    normalQueue.push({ 2, "Sara" });
    normalQueue.push({ 3, "Usman" });

    //  (priority queue)

    // To place the smallest value at the top, you must specify all three template
    // parameters
    priority_queue<VIPCustomer, vector<VIPCustomer> // store in an array
        ,
```



```cpp
        ,
    CompareVIP>
    vipQueue;

// Enqueue some VIP customers
vipQueue.push({ 101, "VIP_Ahmed", 2 }); // lower priority
vipQueue.push({ 102, "VIP_Zara", 1 });  // highest priority
vipQueue.push({ 103, "VIP_Omar", 3 });  // lowest priority

cout << "Serving VIP customers:" << endl;
while (!vipQueue.empty()) {
    VIPCustomer currentVIP = vipQueue.top();
    vipQueue.pop();

    cout << "Serving VIP ID: " << currentVIP.id << ", Name: " << currentVIP.name
        << ", Priority: " << currentVIP.priority << endl;
}

cout << "\nServing normal customers:" << endl;
while (!normalQueue.empty()) {
    Customer current = normalQueue.front();
    normalQueue.pop();

    cout << "Serving customer ID: " << current.id << ", Name: " << current.name
        << endl;
}

return 0;
```

Output:

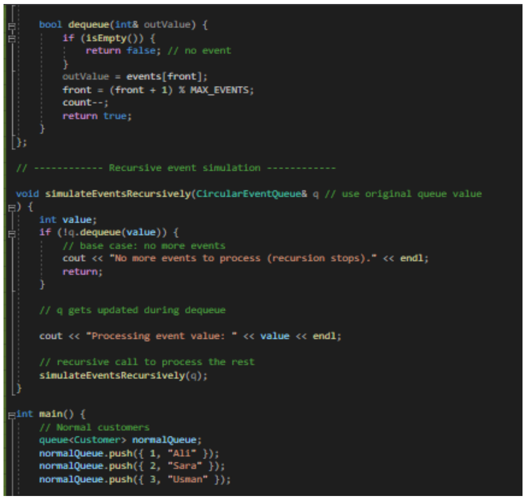


```
Serving VIP customers:
Serving VIP ID: 102, Name: VIP_Zara, Priority: 1
Serving VIP ID: 101, Name: VIP_Ahmed, Priority: 2
Serving VIP ID: 103, Name: VIP_Omar, Priority: 3

Serving normal customers:
Serving customer ID: 1, Name: Ali
Serving customer ID: 2, Name: Sara
Serving customer ID: 3, Name: Usman
```

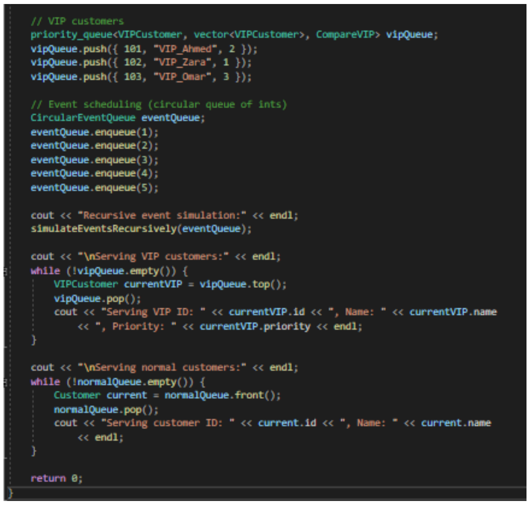
Task3:

Recursion for event simulation

Code:

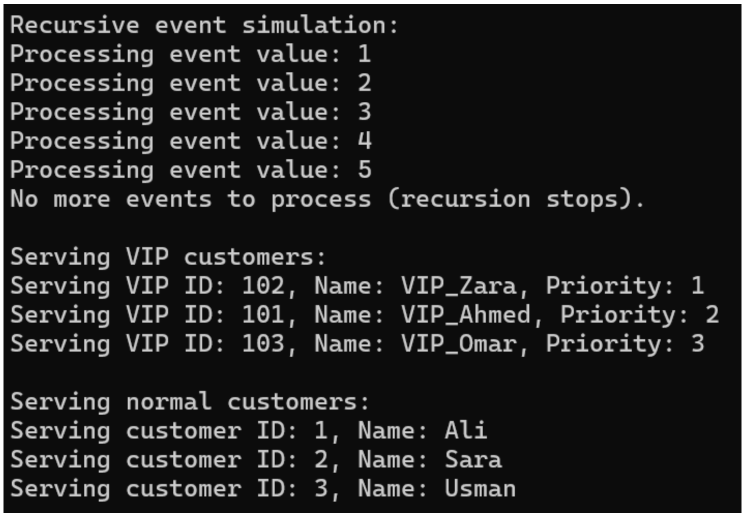


```cpp
    bool dequeue(int& outValue) {
        if (isEmpty()) {
            return false; // no event
        }
        outValue = events[front];
        front = (front + 1) % MAX_EVENTS;
        count--;
        return true;
    }
};

// ----------- Recursive event simulation -----------

void simulateEventsRecursively(CircularEventQueue& q // use original queue value
) {
    int value;
    if (!q.dequeue(value)) {
        // base case: no more events
        cout << "No more events to process (recursion stops)." << endl;
        return;
    }

    // q gets updated during dequeue

    cout << "Processing event value: " << value << endl;

    // recursive call to process the rest
    simulateEventsRecursively(q);
}

int main() {
    // Normal customers
    queue<Customer> normalQueue;
    normalQueue.push({ 1, "Ali" });
    normalQueue.push({ 2, "Sara" });
    normalQueue.push({ 3, "Usman" });
```



```cpp
    // VIP customers
    priority_queue<VIPCustomer, vector<VIPCustomer>, CompareVIP> vipQueue;
    vipQueue.push({ 101, "VIP_Ahmed", 2 });
    vipQueue.push({ 102, "VIP_Zara", 1 });
    vipQueue.push({ 103, "VIP_Omar", 3 });

    // Event scheduling (circular queue of ints)
    CircularEventQueue eventQueue;
    eventQueue.enqueue(1);
    eventQueue.enqueue(2);
    eventQueue.enqueue(3);
    eventQueue.enqueue(4);
    eventQueue.enqueue(5);

    cout << "Recursive event simulation:" << endl;
    simulateEventsRecursively(eventQueue);

    cout << "\nServing VIP customers:" << endl;
    while (!vipQueue.empty()) {
        VIPCustomer currentVIP = vipQueue.top();
        vipQueue.pop();
        cout << "Serving VIP ID: " << currentVIP.id << ", Name: " << currentVIP.name
            << ", Priority: " << currentVIP.priority << endl;
    }

    cout << "\nServing normal customers:" << endl;
    while (!normalQueue.empty()) {
        Customer current = normalQueue.front();
        normalQueue.pop();
        cout << "Serving customer ID: " << current.id << ", Name: " << current.name
            << endl;
    }

    return 0;
}
```

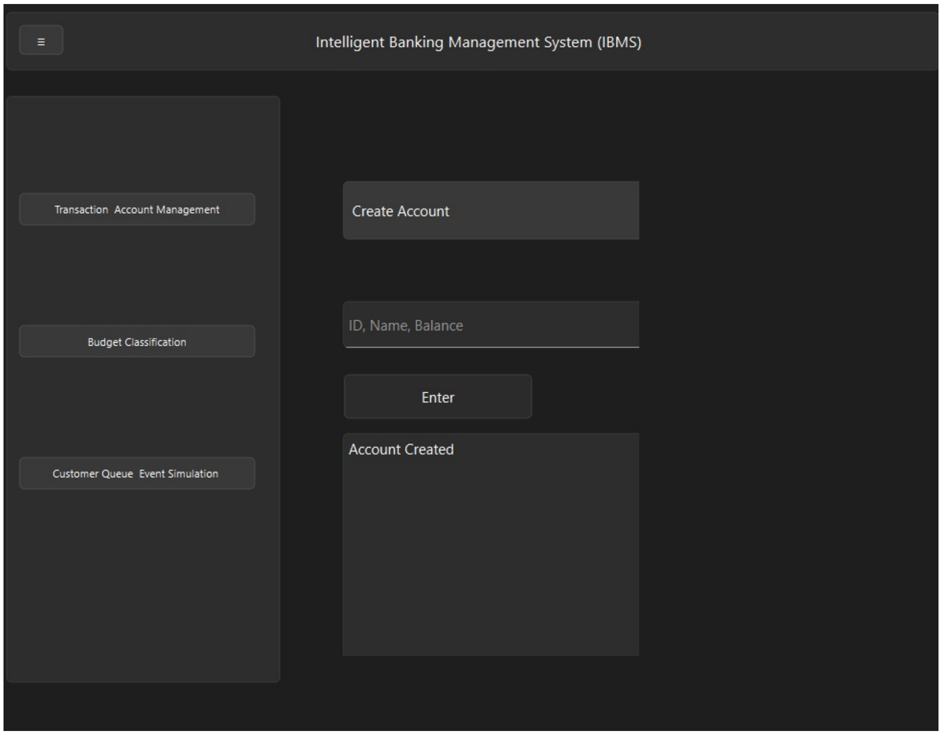
Output:



```
Recursive event simulation:
Processing event value: 1
Processing event value: 2
Processing event value: 3
Processing event value: 4
Processing event value: 5
No more events to process (recursion stops).

Serving VIP customers:
Serving VIP ID: 102, Name: VIP_Zara, Priority: 1
Serving VIP ID: 101, Name: VIP_Ahmed, Priority: 2
Serving VIP ID: 103, Name: VIP_Omar, Priority: 3

Serving normal customers:
Serving customer ID: 1, Name: Ali
Serving customer ID: 2, Name: Sara
Serving customer ID: 3, Name: Usman
```

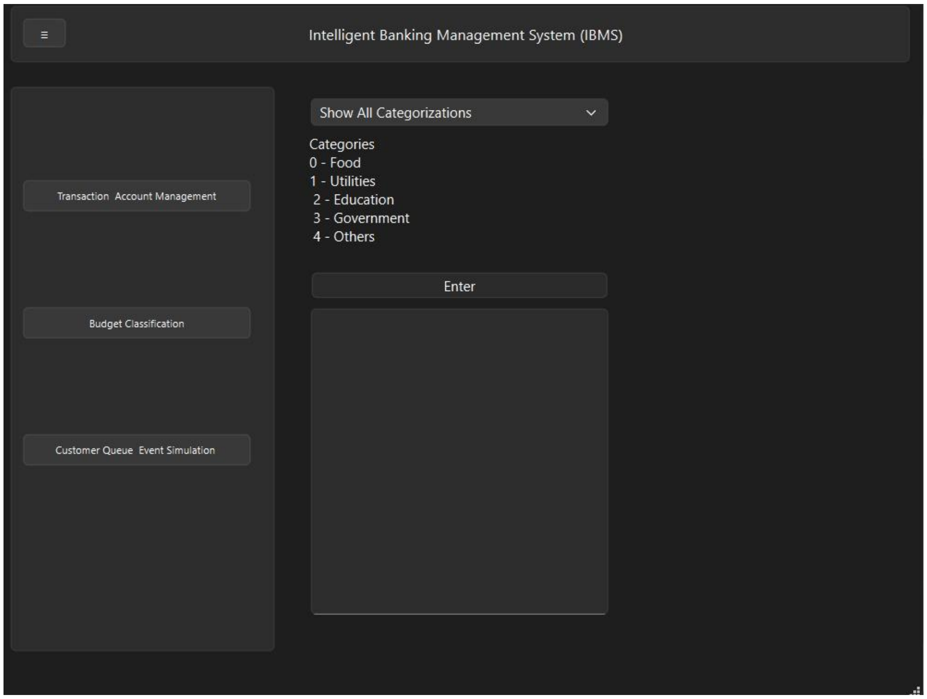
GUI/UX:

Module A:


≡
Intelligent Banking Management System (IBMS)
Transaction Account Management
Budget Classification
Customer Queue Event Simulation
Create Account
ID, Name, Balance
Enter

Account Created

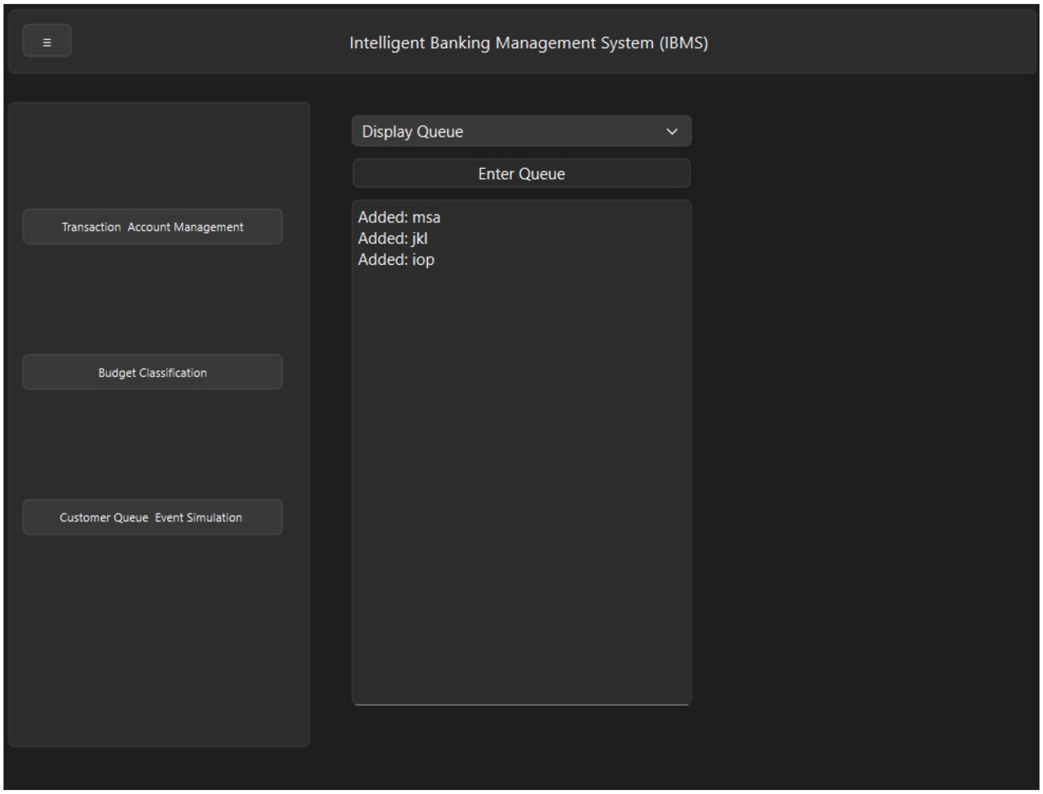
Module B:


≡
Intelligent Banking Management System (IBMS)
Transaction Account Management
Budget Classification
Customer Queue Event Simulation
Enter

Show All Categorizations

```
Categories
0 - Food
1 - Utilities
 2 - Education
 3 - Government
 4 - Others
```

Module C:


≡
Intelligent Banking Management System (IBMS)
Transaction Account Management
Budget Classification
Customer Queue Event Simulation
Display Queue
Enter Queue

```
Added: msa
Added: jkl
Added: iop
```

Problems Faced & Troubleshooting:

1. Infinite Loops

- Occurred in queue implementation
- Resolved by correcting loop conditions

2. Stack Underflow / Overflow

- Occurred during undo/redo
- Fixed by adding boundary checks

3. Incorrect Budget Classification

- Keyword overlap caused misclassification

- Solved by refining keyword rules

---

4. Memory Issues

- Linked list pointers mishandled
- Fixed by proper node deletion

---

5. Priority Queue Errors

- VIP customers not prioritized
- Corrected by adjusting comparison logic

Conclusion:

The **Intelligent Bank Management and Simulation System (IBMS)** successfully demonstrate the practical application of **Data Structures and Algorithms** in a real-world banking context.

Through the implementation of arrays, linked lists, stacks, queues, priority queues, circular queues, recursion, and searching/sorting algorithms, the project achieves both **academic learning objectives** and **practical relevance**.

The system provides:

- Efficient transaction handling
- Automated budget analysis
- Realistic customer flow simulation

This project enhanced our understanding of **DSA concepts**, **problem-solving skills**, and **software design**, making it a valuable learning experience aligned with modern banking systems.
