**Deliverable 2:**

**Initial Implementation & Data Structure Integration Report**

**1. Basic Working Code – Implementation of Core Data Structures**

**The implemented banking system uses the following data structures:**

a. Linked List (std::list<Customer>)

Used to store customer records.

Allows dynamic insertion and deletion of customers.

Sample Input/Output:

// Adding a customer

bank.add\_customer();

// Output: Customer added successfully. Account number: 1

b. Hash Table (std::unordered\_map<int, list<Customer>::iterator>)

Maps account numbers to customer records for O(1) access.

Sample Input/Output:

// Searching for a customer

bank.display\_user(1, 1234);

// Output: Displays customer details if account number and PIN are correct.

C. Binary Search Tree (Custom Implementation)

Used for efficient searching of customers by account number.

Sample Input/Output:

//Depositing money (uses BST search)

bank.deposit(1, 1234, 500.0);

Output: Updates balance if account number and PIN are correct.

d. Queue (std::queue<Transaction>)

Stores transaction history for each customer (FIFO order).

Sample Input/Output:

Displaying transactions

customer.displayTransactions();

Output: Lists all transactions in chronological order.

2. Functional Demonstration

Key Functionalities Demonstrated:

1. Adding a New Customer

Input: Name, PIN, address, contact, email.

Output: Success message with generated account number.

2. Displaying Customer Details

Input: Account number and PIN.

Output: Customer details (name, balance, transactions, etc.) if authenticated.

3. Depositing Money

Input: Account number, PIN, amount.

Output: Updated balance and transaction record.

4. Withdrawing Money

Input: Account number, PIN, amount.

Output: Success message if sufficient balance, else error.

5.Transaction History

Input: Account number and PIN.

Output: List of all past transactions (deposits/withdrawals).

3. Efficiency Analysis

| \*\*Data Structure\*\* | \*\*Time Complexity\*\* | \*\*Space Complexity\*\* | \*\*Use Case\*\* |

| \*\*Linked List (`std::list`)\*\* | - Insertion: O(1) <br> - Deletion: O(1) <br> - Search: O(n) | O(n) | Storing customer records |

| \*\*Hash Table (`std::unordered\_map`)\*\* | - Insertion: O(1) <br> - Search: O(1) <br> - Deletion: O(1) | O(n) | Fast account number lookup |

| \*\*Binary Search Tree (Custom BST)\*\* | - Insertion: O(log n) avg, O(n) worst <br> - Search: O(log n) avg, O(n) worst <br> - Deletion: O(log n) avg, O(n) worst | O(n) | Efficient searching of customers |

| \*\*Queue (`std::queue`)\*\* | - Enqueue: O(1) <br> - Dequeue: O(1) <br> - Peek: O(1) | O(n) | Transaction history management |

4. Application of Class Concepts

Encapsulation:

`Customer` struct encapsulates all customer-related data and operations (deposit, withdraw, transaction history).

`Bank` class manages all banking operations while hiding internal data structures.

Modularity:

Each data structure has a distinct role (list for storage, hash table for fast access, BST for search, queue for transactions).

Efficiency:

* Hash table provides O(1) access for account verification.
* BST ensures O(log n) search time for customer operations.

Real-world Applicability:

Mimics actual banking operations (secure access, transaction logging, account management).

Conclusion

The implemented system successfully integrates multiple data structures to efficiently manage a banking application. Each structure serves a specific purpose, ensuring optimal performance for different operations. Future improvements could include balancing the BST (AVL/Red-Black Tree) and implementing persistent storage (file/database integration).