Bank Management System Project Documentation

Introduction

The Bank Management System is a console-based application developed as part of our Data Structures and Algorithms course. This project demonstrates the efficient use of fundamental data structures and algorithms to handle core banking operations, including managing accounts and tracking transaction histories. The system is designed to showcase the application of stacks and trees in solving real-world problems while emphasizing performance, scalability, and maintainability.

Data Structures Used

1. Stack for Transaction History

Why Stack?

We chose a **stack** to store and manage the transaction history for the following reasons:

- **Last-In-First-Out (LIFO)**: The stack structure is ideal for accessing the most recent transactions quickly.
- **Efficient Operations**: Transactions are frequently added or removed, and a stack allows O(1) insertion and deletion at the top.
- **Natural Order**: Using a stack ensures that transaction histories are stored in reverse chronological order, which aligns with how transaction logs are typically reviewed.

Implementation Highlights

- Each transaction is represented as an element in the stack.
- The element contains the following fields:
    - **Transaction ID (Account number in our case)**
    - **Type (Deposit/Withdrawal)**
    - **Amount**
    - **Timestamp (we use system time by using ctime library)**
- Traversal of the stack ensures that transaction histories can be displayed in the order they occurred.

Advantages of Stack in This Context

- Simplifies adding new transactions.
- Provides quick access to the most recent transaction.
- Reduces the complexity of managing transaction history logs.

2. Tree for Account Management

Why Tree?

We implemented a **tree** to store account details because of the following reasons:

- **Efficient Searching**: A tree allows for searching, inserting, and deleting accounts in O(log n) time on average, making it suitable for managing potentially large numbers of accounts.
- **Inorder Traversal for Display**: Trees inherently support inorder traversal, which allows for displaying account details in a sorted order (e.g., by account number).
- **Structured Storage**: Trees provide an organized way to store and retrieve accounts based on unique keys, such as account numbers.

Implementation Highlights

- Each node in the tree represents an account and contains:
    - **Account Number** (Unique Key)
    - **Account Holder's Name**
    - **Balance**
    - **Pointer to Transaction History (Stack)**
- **Data Persistence**: Account information is stored in an external file (accounts.txt) located in the database directory for later retrieval and updating.
- **Balancing**: While our implementation is not strictly self-balancing (e.g., AVL or Red-Black Tree), we ensured the tree remains reasonably balanced through careful insertion logic to avoid performance degradation.

Traversal Algorithms

We used **inorder traversal** to display account information. This algorithm ensures that accounts are displayed in ascending order of account numbers, enhancing the user experience.

Advantages of Tree in This Context

- Allows hierarchical organization of accounts.
- Facilitates efficient account retrieval and management.
- Provides a natural mechanism for sorted traversal.

Why These Data Structures Were Chosen

1. Performance Optimization

The combination of stacks and trees ensures that each operation (insertion, deletion, search, traversal) is optimized for its respective data type:

- Stacks excel at managing dynamic and sequential operations like transaction history.
- Trees are ideal for structured, hierarchical storage and fast retrieval of accounts.

2. Scalability

- **Stack**: Can handle an arbitrary number of transactions without worrying about pre-allocated memory.
- **Tree**: Can efficiently manage hundreds or thousands of accounts while maintaining fast lookup and insertion times.

3. Simplicity and Clarity

The choice of stacks and trees allowed us to implement the system with clarity and focus on fundamental concepts, making the project easy to understand and extend in the future.

Algorithmic Details

1. Inorder Traversal for Tree

We used **inorder traversal** to display accounts in ascending order. The algorithm:

1. Recursively visit the left subtree.
2. Display the current node (account information).
3. Recursively visit the right subtree.

***Time Complexity**: O(n), where n is the number of accounts.*

2. Stack Traversal

To display the transaction history, we traversed the stack from top to bottom, printing each element's data.

***Time Complexity**: O(m), where m is the number of transactions for an account.*

Conclusion

Our Bank Management System efficiently utilizes **stacks** and **trees** to handle core functionalities such as transaction tracking and account management. The thoughtful choice of these data structures ensures that the system is both performant and scalable, meeting the requirements of a modern banking application. Additionally, the use of persistent storage (accounts.txt) ensures that account details are securely maintained for future use. This project not only showcases our understanding of data structures and algorithms but also highlights their practical application in solving real-world problems.