2043. Simple Bank System

Medium Design Array Hash Table Simulation Leetcode Link

Problem Description

This problem simulates the operations of a bank with `n` accounts through a class called `Bank`. Each account in the bank is numbered from 1 to `n`, and the initial balance for each account is given in a 0-indexed array; this means that the balance of the i-th account in the array corresponds to account number i + 1 in our bank. The `Bank` class needs to support three types of transactions: transfers between two accounts, deposits into an account, and withdrawals from an account. A transaction is considered valid if the account numbers are within the valid range (1 to `n`) and for transfers and withdrawals, there is enough money in the account to cover the transaction. Implementing the required functionality in the `Bank` class involves creating methods that successfully handle these transactions while maintaining accurate account balances and ensuring all business rules are respected.

Intuition

When designing the `Bank` class and its methods, we should first consider the constraints around account numbers and balances. To handle transactions, we need to check that the requested operation is valid: the account number(s) must be within the valid range and there must be sufficient funds to complete a withdrawal or transfer.

We start with the constructor `__init__`, which initializes the `Bank` with a list of account balances and the total number of accounts. This sets up our initial state.

For the `transfer` method, we check the validity of the account numbers and the availability of funds in the source account. If these conditions are met, we subtract the transferred money from `account1` and add it to `account2`. If the conditions are not met, we return `False`.

The `deposit` method is slightly simpler, as it only requires a check to ensure the account exists (the account number is within range). If it does, we add the money to the account's balance.

The `withdraw` method involves checking both the account number validity and the availability of funds, much like the `transfer` method. If the checks pass, we subtract the money from the account.

Each of these methods returns a boolean indicating whether the operation was successful (`True`) or not (`False`).

By carefully following the rules stated in the problem description and using basic conditional statements to enforce the rules, we arrive at a straightforward solution that satisfies all the requirements of the `Bank` class.

Solution Approach

The solution is structured around the `Bank` class with three key methods: `transfer`, `deposit`, and `withdraw`.

- The `__init__` constructor simply assigns the provided balance list and calculates the number of accounts using the `len` function which is stored in `self.n`.
- For the `transfer` method:
  1. It first checks if both the source (`account1`) and destination (`account2`) account numbers are within the valid range by comparing them with `self.n`. It also checks if the source account has enough balance to transfer by comparing `money` with the balance of `account1`.
  2. If any condition fails, it returns `False`.
  3. If the conditions are met, the method subtracts the amount of `money` from the balance of `account1` and adds that amount to the balance of `account2`.
  4. It finally returns `True` indicating a successful transaction.
- For the `deposit` method:
  1. It verifies if the account number is valid.
  2. If the account is invalid, it returns `False`.
  3. If valid, it adds the `money` to the balance of the specified account.
  4. It then returns `True` to signify success.
- The `withdraw` method:
  1. Checks if the account number is within the valid range and if the account has a balance greater than or equal to the withdrawal `money`.
  2. Should the account not exist or have insufficient funds, the method returns `False`.
  3. Otherwise, it subtracts the withdrawal `money` from the account's balance and returns `True`.

The design employs basic array indexing to represent individual account balances. This simple data structure is effective in providing efficient direct access to specific elements, which is essential for operations like `transfer`, `deposit`, and `withdraw` that require looking up and modifying values based on account numbers. Conditionals (if-else statements) are used throughout to enforce business rules and validate transactions.

Overall, the solution follows a pragmatic approach, applying core programming constructs like lists, condition checking, arithmetic operations, and boolean return values to fulfill the transaction requirements of a banking system.

Example Walkthrough

Consider a scenario where we have a `Bank` with 3 accounts. The initial balances for these accounts are `[10, 100, 20]`, which correspond to accounts 1, 2, and 3 respectively. Let's walk through a series of transactions to demonstrate the solution approach.

1. **Initialization**:
   - We create a `Bank` object using the initial balances.
   - The `__init__` method sets up our internal state with these balances and establishes that we have 3 accounts.
2. **Deposit Attempt**:
   - We call the `deposit` method for account number 2 with an amount of `50`.
   - The method checks if account number 2 is valid (which is true, as it's within the range of 1 to 3).
   - It adds 50 to account number 2's balance, resulting in the new balances being `[10, 150, 20]`.
   - The method returns `True` since the transaction was successful.
3. **Withdrawal Attempt**:
   - Next, we attempt to withdraw `30` from account number 3.
   - The `withdraw` method checks if account 3 has at least `30`. It does, so the money is subtracted.
   - The balance is updated to `[10, 150, -10]`, and the method returns `True`.
   - Note: This outcome reveals an oversight in the provided approach, as it allows the balance to go negative. A real implementation should check and ensure the withdrawal does not result in a negative balance, so the transaction should instead return `False`.
4. **Transfer Attempt**:
   - We now want to transfer `100` from account 1 to account 3. We call the `transfer` method with `account1 = 1`, `account2 = 3`, and `money = 100`.
   - The `transfer` method first checks if account 1 exists and has at least `100` to transfer. It doesn't, as the balance is currently `10`.
   - Since there are insufficient funds, the method returns `False`, and no transfer occurs.

Through this example, the essential process of depositing, withdrawing, and transferring money between accounts is illustrated using array indexing, condition statements, and boolean values to handle the business logic of these transactions systematically. However, modifications might be necessary to prevent negative account balances as noted in step 3.

Python Solution

```python
# Define the Bank class with appropriate methods for banking operations
class Bank:
    def __init__(self, balance: List[int]):
        self.balance = balance  # Initialize an account balance list
        self.num_accounts = len(balance)  # Store the number of accounts based on the length of the balance list

    # Transfer money from one account to another if valid and possible
    def transfer(self, account1: int, account2: int, money: int) -> bool:
        # Check for valid account numbers and sufficient funds before transfer
        if account1 > self.num_accounts or account2 > self.num_accounts or self.balance[account1 - 1] < money:
            return False  # conditions not met, return False

        # Perform the transfer
        self.balance[account1 - 1] -= money  # Deduct money from the source account
        self.balance[account2 - 1] += money  # Add money to the destination account
        return True  # return True on successful transfer

    # Deposit money into a given account if the account is valid
    def deposit(self, account: int, money: int) -> bool:
        # Check if the account number is valid
        if account > self.num_accounts:
            return False  # Invalid account, return False

        # Perform the deposit
        self.balance[account - 1] += money  # Add the money to the account balance
        return True  # return True on successful deposit

    # Withdraw money from a given account if the account is valid and has sufficient funds
    def withdraw(self, account: int, money: int) -> bool:
        # Check for valid account number and if the account has sufficient funds
        if account > self.num_accounts or self.balance[account - 1] < money:
            return False  # conditions not met, return False

        # Perform the withdrawal
        self.balance[account - 1] -= money  # Deduct the money from the account balance
        return True  # return True on successful withdrawal


# Example of instantiation and method calls:
# obj = Bank(balance)
# success_transfer = obj.transfer(account1, account2, money)
# success_deposit = obj.deposit(account, money)
# success_withdraw = obj.withdraw(account, money)

```

Java Solution

```java
class Bank {
    private long[] balances; // An array to store the balance of each account.
    private int accountCount; // The total number of accounts in the bank.

    // Constructor to initialize the bank with a given array of balances.
    public Bank(long[] balances) {
        this.balances = balances;
        this.accountCount = balances.length;
    }

    // Method to transfer money from one account to another.
    public boolean transfer(int fromAccount, int toAccount, long amount) {
        // Check if either account number is invalid or if the fromAccount has insufficient funds.
        if (fromAccount > accountCount || toAccount > accountCount || balances[fromAccount - 1] < amount) {
            return false; // Return false to indicate the transfer failed.
        }
        // Deduct the amount from the sender's account.
        balances[fromAccount - 1] -= amount;
        // Add the amount to the receiver's account.
        balances[toAccount - 1] += amount;
        return true; // Return true to indicate the transfer was successful.
    }

    // Method to deposit money into an account.
    public boolean deposit(int account, long amount) {
        // Check if the account number is invalid.
        if (account > accountCount) {
            return false; // Return false to indicate the deposit failed.
        }
        // Add the amount to the account's balance.
        balances[account - 1] += amount;
        return true; // Return true to indicate the deposit was successful.
    }

    // Method to withdraw money from an account.
    public boolean withdraw(int account, long amount) {
        // Check if the account number is invalid or if the account has insufficient funds.
        if (account > accountCount || balances[account - 1] < amount) {
            return false; // Return false to indicate the withdrawal failed.
        }
        // Deduct the amount from the account's balance.
        balances[account - 1] -= amount;
        return true; // Return true to indicate the withdrawal was successful.
    }
}

```

C++ Solution

```cpp
#include <vector>

class Bank {
public:
    // Use underscore naming for private member variables to distinguish from method parameters
    std::vector<long long> _balances;
    int _numAccounts; // Variable to store the number of accounts

    // Constructor to initialize the Bank object with a list of balances
    Bank(std::vector<long long>& balances) {
        _balances = balances;
        _numAccounts = balances.size();
    }

    // Transfer method - moves 'money' from 'account1' to 'account2'
    bool transfer(int account1, int account2, long long money) {
        // Check if either of the account numbers are invalid or if the balance is insufficient
        if (account1 > _numAccounts || account2 > _numAccounts || _balances[account1 - 1] < money) {
            return false;
        }
        // Perform the transfer by adjusting the balances of both accounts
        _balances[account1 - 1] -= money;
        _balances[account2 - 1] += money;
        return true;
    }

    // Deposit method - adds 'money' to 'account'
    bool deposit(int account, long long money) {
        // Check if the account number is invalid
        if (account > _numAccounts) {
            return false;
        }
        // Increase the balance of the account by 'money'
        _balances[account - 1] += money;
        return true;
    }

    // Withdraw method - deducts 'money' from 'account'
    bool withdraw(int account, long long money) {
        // Check if the account number is invalid or if the balance is insufficient
        if (account > _numAccounts || _balances[account - 1] < money) {
            return false;
        }
        // Decrease the balance of the account by 'money'
        _balances[account - 1] -= money;
        return true;
    }
};

```

Typescript Solution

```typescript
// The balance for each account.
let bankBalance: number[];

/**
 * Initializes the bank balance.
 *
 * @param {number[]} balance - The initial amount of money in each account.
 */
function initializeBankBalance(balance: number[]): void {
    bankBalance = balance;
}

/**
 * Transfers money from one account to another.
 *
 * @param {number} accountFrom - The account number to transfer money from.
 * @param {number} accountTo - The account number to transfer money to.
 * @param {number} amount - The amount of money to transfer.
 * @returns {boolean} True if the transfer was successful, false otherwise.
 */
function transfer(accountFrom: number, accountTo: number, amount: number): boolean {
    // Check for valid account numbers and sufficient balance in the source account.
    if (
        accountFrom > bankBalance.length ||
        accountTo > bankBalance.length ||
        amount > bankBalance[accountFrom - 1]
    )
        return false;

    // Perform the transfer.
    bankBalance[accountFrom - 1] -= amount;
    bankBalance[accountTo - 1] += amount;
    return true;
}

/**
 * Deposits money into an account.
 *
 * @param {number} account - The account number to deposit money into.
 * @param {number} amount - The amount of money to deposit.
 * @returns {boolean} True if the deposit was successful, false otherwise.
 */
function deposit(account: number, amount: number): boolean {
    // Check for a valid account number.
    if (account > bankBalance.length) return false;

    // Perform the deposit.
    bankBalance[account - 1] += amount;
    return true;
}

/**
 * Withdraws money from an account.
 *
 * @param {number} account - The account number to withdraw money from.
 * @param {number} amount - The amount of money to withdraw.
 * @returns {boolean} True if the withdrawal was successful, false otherwise.
 */
function withdraw(account: number, amount: number): boolean {
    // Check for valid account numbers and sufficient balance.
    if (account > bankBalance.length || amount > bankBalance[account - 1]) {
        return false;
    }

    // Perform the withdrawal.
    bankBalance[account - 1] -= amount;
    return true;
}

// Example usage:
// initializeBankBalance([100, 200, 300]);
// const transferSuccess = transfer(1, 2, 50);
// const depositSuccess = deposit(3, 75);
// const withdrawalSuccess = withdraw(2, 25);

```

Time and Space Complexity

Time Complexity

- **`__init__()`**: The time complexity is `O(1)` because only a reference to the list is created (assuming that the list is passed by reference and not copied).
- **`transfer(account1, account2, money)`**: The time complexity is `O(1)` because it performs a constant number of operations: checking whether the accounts are valid, whether the balance is sufficient, and updating the balances.
- **`deposit(account, money)`**: The time complexity is `O(1)` as it involves a simple validation of the account existence and an addition operation on the balance.
- **`withdraw(account, money)`**: The time complexity is `O(1)` since it includes the checking of whether the account exists and whether the balance is enough, followed by the subtraction from the balance.

Space Complexity

- **`__init__()`**: The space complexity is `O(n)` because it stores a list of balances for `n` accounts, where `n` is the length of the initial list. Here, `n` represents the number of accounts.
- **`transfer(account1, account2, money)`**: The space complexity is `O(1)` as it does not use any additional space that scales with the input size.
- **`deposit(account, money)`**: The space complexity is `O(1)` given that no extra space proportional to the size of input data is used besides what is already stored in the Bank class.
- **`withdraw(account, money)`**: The space complexity is also `O(1)` for the same reason as deposit and transfer; it does not require additional space depending on the input.