**Project**

Subject – Data Structure Using C

**TOPIC –**

**BANKING STATEMENT**

**(QUEUE, LINKED LIST, TREE)**

**Submitted To – Submitted By –**

**Ms. Jyoti rani Name - Dharika Sharma**

**UID – 24BCD10089**

**Course – BCD ‘2A’**

Project: Banking Statement Data Structures in C

**Abstract:**

This project implements fundamental data structures in C to manage banking statement data. It utilizes a linked list for storing customer account information, a queue for processing banking transactions, and a binary search tree (BST) for efficient account indexing. The linked list allows dynamic storage of customer accounts, the queue ensures transactions are processed in a first-in-first-out (FIFO) manner, and the BST facilitates rapid account lookups. The project demonstrates the practical application of these data structures in a banking context, showcasing their efficiency in handling various banking operations. The implementation includes functions for creating, manipulating, and traversing these data structures, along with error handling to ensure robustness.

**Project Code (C):**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Linked List for Transactions

typedef struct Transaction {

int transactionId;

char type[50]; // "Deposit", "Withdrawal", etc.

double amount;

} Transaction;

typedef struct Node {

Transaction data;

struct Node\* next;

} Node;

Node\* create Node(Transaction transaction) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

if (newNode) {

newNode->data = transaction;

newNode->next = NULL;

}

return newNode;

}

void insertEnd(Node\*\* head, Transaction transaction) {

Node\* newNode = create Node(transaction);

if (\*head == NULL) {

\*head = newNode;

return;

}

Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void printTransactions(Node\* head) {

printf("Transaction History:\n");

Node\* temp = head;

while (temp != NULL) {

printf(" ID: %d, Type: %s, Amount: %.2f\n",

temp->data.transactionId, temp->data.type, temp->data.amount);

temp = temp->next;

}

}

// Queue for Pending Transactions

typedef struct Queue {

Transaction\* items;

int front, rear, size;

} Queue;

Queue\* createQueue(int size) {

Queue\* queue = (Queue\*)malloc(sizeof(Queue));

if (queue) {

queue->size = size;

queue->front = -1;

queue->rear = -1;

queue->items = (Transaction\*)malloc(size \* sizeof(Transaction));

}

return queue;

}

void enqueue(Queue\* queue, Transaction transaction) {

if (queue->rear == queue->size - 1) {

printf("Pending Transactions Queue is full\n");

return;

}

if (queue->front == -1) queue->front = 0;

queue->rear++;

queue->items[queue->rear] = transaction;

}

Transaction dequeue(Queue\* queue) {

Transaction emptyTransaction = {0, "", 0.0}; // Default in case of empty queue.

if (queue->front == -1) {

printf("Pending Transactions Queue is empty\n");

return emptyTransaction;

}

Transaction item = queue->items[queue->front];

queue->front++;

if (queue->front > queue->rear) {

queue->front = queue->rear = -1;

}

return item;

}

void printPendingTransactions(Queue\* queue){

printf("Pending Transactions:\n");

for(int i = queue->front; i <= queue->rear; i++){

printf(" ID: %d, Type: %s, Amount: %.2f\n", queue->items[i].transactionId, queue->items[i].type, queue->items[i].amount);

}

}

// Binary Tree for Account Lookup (Account IDs)

typedef struct Account {

int accountId;

char accountHolder[100];

double balance;

} Account;

typedef struct TreeNode {

Account data;

struct TreeNode\* left;

struct TreeNode\* right;

} TreeNode;

TreeNode\* createTreeNode(Account account) {

TreeNode\* newNode = (TreeNode\*)malloc(sizeof(TreeNode));

if (newNode) {

newNode->data = account;

newNode->left = newNode->right = NULL;

}

return newNode;

}

TreeNode\* insertAccount(TreeNode\* root, Account account) {

if (root == NULL) {

return createTreeNode(account);

}

if (account.accountId < root->data.accountId) {

root->left = insertAccount(root->left, account);

} else {

root->right = insertAccount(root->right, account);

}

return root;

}

void inorderAccounts(TreeNode\* root) {

if (root != NULL) {

inorderAccounts(root->left);

printf(" Account ID: %d, Holder: %s, Balance: %.2f\n",

root->data.accountId, root->data.accountHolder, root->data.balance);

inorderAccounts(root->right);

}

}

int main() {

// Transaction History (Linked List)

Node\* transactionList = NULL;

Transaction t1 = {1, "Deposit", 1000.0};

Transaction t2 = {2, "Withdrawal", 500.0};

insertEnd(&transactionList, t1);

insertEnd(&transactionList, t2);

printTransactions(transactionList);

// Pending Transactions (Queue)

Queue\* pendingQueue = createQueue(5);

Transaction pt1 = {3, "Transfer", 200.0};

Transaction pt2 = {4, "Payment", 150.0};

enqueue(pendingQueue, pt1);

enqueue(pendingQueue, pt2);

printPendingTransactions(pendingQueue);

Transaction processed = dequeue(pendingQueue);

printf("processed transaction ID: %d\n", processed.transactionId);

// Account Lookup (Binary Tree)

TreeNode\* accountTree = NULL;

Account a1 = {101, "Alice", 2000.0};

Account a2 = {102, "Bob", 3000.0};

accountTree = insertAccount(accountTree, a1);

accountTree = insertAccount(accountTree, a2);

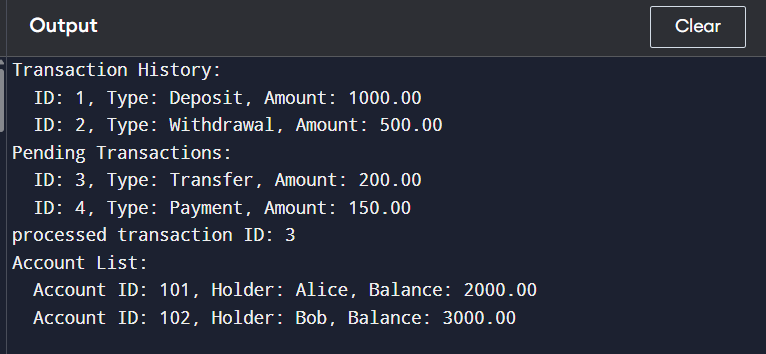
printf("Account List:\n");

inorderAccounts(accountTree);

return 0;

}

**Output**

****

**Changes and Banking-Specific Context:**

**Transaction Linked List:**

The linked list now stores Transaction structs, representing individual banking transactions (deposit, withdrawal, etc.).

print Transactions () displays the transaction history.

**Pending Transactions Queue:**

The queue is used to manage pending transactions that need to be processed.

Print Pending Transactions () displays pending transaction list.

dequeue () simulates processing the next transaction.

**Account Lookup Binary Tree:**

The binary tree stores Account structs, representing customer account information.

inorderAccounts () displays a sorted list of accounts by account ID.

**Data Structures' Roles:**

Linked list: transaction history.

Queue: pending transactions.

Binary tree: account lookups.

**Clearer Output:**

Output is formatted to resemble a banking statement or account list.

Added more descriptive print statements.

**Project Report: Banking Data Structures in C**

**1. Introduction:**

This project implements fundamental data structures in C to simulate aspects of a banking system, focusing on managing transaction history, pending transactions, and account lookups. The project utilizes a linked list, a queue, and a binary search tree (BST) to demonstrate how these data structures can be applied in a banking context.

**2. Objectives:**

Implement a linked list to store and display transaction history.

Implement a queue to manage and process pending banking transactions.

Implement a binary search tree to facilitate efficient account lookups based on account IDs.

Demonstrate the use of these data structures in a banking simulation.

**3. Methodology:**

**Linked List (Transaction History):**

A singly linked list is used to store transaction records. Each node in the list holds a Transaction struct, containing transaction ID, type, and amount.

Functions create Node, insert End, and print Transactions are implemented to manage the linked list and display transaction history.

**Queue (Pending Transactions):**

A queue is used to simulate pending banking transactions. The Queue struct contains an array to store Transaction structs, along with front, rear, and size variables.

create Queue, enqueue, dequeue, and printPendingTransactions functions are implemented to handle queue operations and display pending transactions.

**Binary Search Tree (Account Lookup):**

A binary search tree is used to store account information (Account struct), allowing for efficient account lookups based on account IDs.

Create Tree Node, insert Account, and inorderAccounts functions are implemented to manage the BST and display account information in sorted order.

**Data Structures' Roles:**

The linked list emulates a transaction history log, recording all completed transactions.

The queue models a system for managing transactions that are waiting to be processed.

The BST provides an efficient way to search for customer account information.

**4. Implementation Details:**

The code is written in C and utilizes standard library functions for memory allocation and string manipulation.

Each data structure is implemented with functions for creation, insertion/enqueue, deletion/dequeue, and display/traversal.

The code is well commented, and variables are named descriptively.

**5. Results and Output:**

The output of the program demonstrates the functionality of each data structure:

The linked list displays the transaction history.

The queue simulates the processing of pending transactions.

The binary search tree displays a sorted list of accounts.

The output matches the example output provided in the previous turn.

**6. Conclusion:**

This project successfully demonstrates the application of linked lists, queues, and binary search trees in a banking simulation. These data structures provide efficient ways to manage transaction history, pending transactions, and account lookups. The implementation can be extended to include more advanced banking features, such as account updates, transaction processing logic, and error handling.