SYMBIOSIS INSTITUTE OF TECHNOLOGY,HYDERABAD
A Constituent of Symbiosis International (Deemed University), Pune.
Re-accredited by NAAC with 'A++ Grade |Awarded Category - I by UGC

DATA STRUCTURES PROJECT REPORT

Bank Transaction Management System

Project Title:

Bank Transaction Management System

Course Name:

Data Structures (Theory and Lab)

Submitted by:

Gunda Ravi Partha Sarathi (PRN: 24070724005, Branch: Computer Science & Technology)

Vattikonda Venkata Karthik (PRN: 24070724019, Branch: Computer Science & Technology)

Faculty Guide:

Dr. Salakapuri Rakesh

Department:

Department of Computer Science and Engineering

College / University:

Symbiosis Institute of Technology, Hyderabad

Semester and Academic Year:

3rd Semester, 2025–2026

CONTENTS

ABSTRACT

This project titled "**Bank Transaction Management System"** represents the development of a miniature yet conceptually rich banking simulation using the C programming language**.** The system primarily utilizes **linked lists and stacks** to manage dynamic data storage, enabling efficient and reversible financial transactions such as deposits, withdrawals, and balance tracking. Unlike static array-based implementations that restrict scalability, this system adopts a **linked-list-based stack** architecture, which provides flexibility in memory allocation and supports an undo feature that mirrors real-world transaction rollbacks. The project's primary goal is to demonstrate how fundamental data structures can be used to design a simplified yet realistic banking system that performs logical operations with precision and reliability. Every transaction is stored as a node in a dynamic stack, and through this structure, the project reinforces the principle of Last-In-First-Out (LIFO)**,** which forms the basis of undo operations in computer science and finance alike. The integration of dynamic memory allocation ensures that transaction records can grow without limitation, providing an efficient memory management mechanism that mirrors the dynamic nature of banking systems. Beyond technical implementation, this project bridges the gap between theoretical knowledge and real-world application. It allows students to explore how data structures, algorithms, and programming logic can collectively solve real-life computational problems. The project also enhances conceptual understanding of modular programming, as each functionality, such as deposit, withdrawal, or undo—is encapsulated within well-defined functions. In summary, this project not only highlights the versatility of the C programming language but also serves as a practical model for understanding data abstraction, algorithmic reasoning, and system-level problem-solving.

INTRODUCTION

The modern era of computing relies on data structures for efficient information storage, retrieval, and manipulation. Every sophisticated system—be it banking, communication, or healthcare—depends on the effective management of data and logical operations. This project explores the use of data structures and algorithms through the development of a simple, interactive banking system implemented in **C**. The objective is to translate the conceptual understanding of stacks and linked lists into a practical, functioning program that simulates common financial operations.The C programming language was chosen due to its speed, memory control, and suitability for system-level applications. C provides low-level access to memory through pointers**,** enabling precise management of dynamically allocated structures. In this project, the linked list acts as the foundation for storing multiple transactions, while the stack is used to implement transaction undo functionality. Each transaction—whether a deposit or withdrawal—is dynamically allocated and linked, allowing the program to maintain an expandable transaction history that can grow as the user performs more actions.The motivation behind creating this project stems from the need to understand how fundamental data structures can support practical and reversible data operations. In the context of banking, the ability to track and undo recent transactions is essential for maintaining data integrity. By implementing a stack-based mechanism, this project effectively captures this requirement and provides an intuitive model of how software can replicate real-world financial workflows. This system is modular and menu-driven, allowing the user to interactively choose actions and observe changes in real time. Each module of the program—such as deposit, withdraw, viewLastTransaction, or undoLastTransaction—performs a distinct operation that contributes to the overall functionality. The inclusion of a memory cleanup function further strengthens the program's robustness, preventing memory leaks and ensuring efficient resource utilization. In essence, the Simple Banking System project stands as an educational prototype that connects theoretical computer science with practical applications.

METHODOLOGY

The project follows a modular programming approach, dividing the entire program into independent, function-based modules. Each function has a clearly defined purpose and interacts with others through shared data structures. The methodology ensures clarity, reusability, and efficient debugging.

1. Data Representation

The system uses two custom data structures:

- **Transaction Structure:**
  Each transaction is represented as a node containing three fields — the transaction type (D for deposit or W for withdrawal), the transaction amount, and a pointer to the next node. These nodes are linked together to form a stack.
- **Account Structure:**
  The account maintains the total balance and a pointer to the top of the transaction history. This allows the system to efficiently track and undo transactions.

By using linked lists, the system achieves flexibility, allowing transactions to grow dynamically without predefined limits.

2. Core Functional Modules

The project is composed of the following main functional components:

- **createAccount():**
  Initializes an account with a given balance and sets up the transaction stack. Uses malloc() for dynamic allocation.
- **deposit():**
  Adds an amount to the account, updates the balance, and records the transaction using push().
- **withdraw():**
  Deducts a specified amount if sufficient funds exist. It prevents overdrafts by checking current balance before transaction approval.
- **viewLastTransaction():**
  Displays the details of the most recent transaction using the top node of the stack.

- **undoLastTransaction():**
  Implements the stack pop operation. If the last transaction was a deposit, it subtracts the amount; if it was a withdrawal, it adds the amount back. The corresponding node is freed after reversal.
- **cleanup():**
  Frees all dynamically allocated memory (both transaction nodes and account structure) before the program terminates.

3. Algorithm Flow

The general flow of the program can be represented as follows:

1. Start the program and create a new account with an initial balance.
2. Display a menu of available options (Deposit, Withdraw, View Balance, View Last Transaction, Undo, Exit).
3. Take user input for the chosen operation.
4. Execute the corresponding function.
5. Repeat steps 2–4 until the user selects Exit.
6. Perform cleanup to release all allocated memory.

This flow ensures modularity**,** interactivity**,** and data consistency throughout execution.

4. Advantages of Using Stack and Linked List

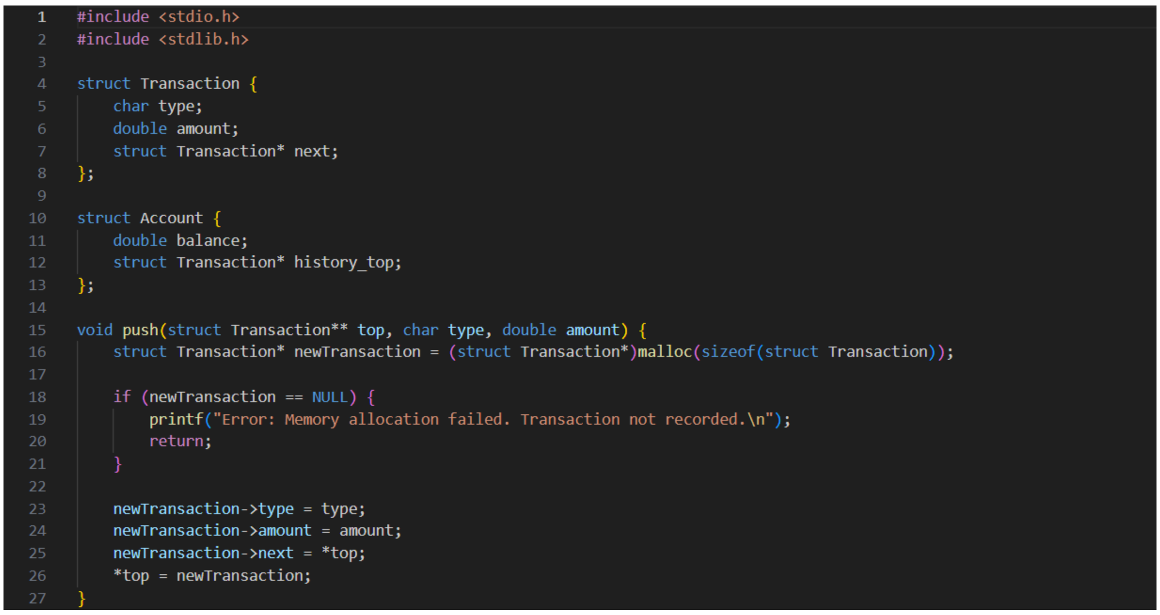
- Eliminates the need for static arrays and allows unlimited transaction history.
- Makes “Undo” implementation intuitive using LIFO logic.
- Ensures real-time dynamic allocation and efficient memory use.
- Enhances system robustness through proper resource management.

CODE EXPLANATION

The program is written in **C language** and demonstrates how fundamental **data structures** such as **linked lists** and **stacks** can be utilized to simulate a real-world banking system. The entire code is modular, meaning each function performs a specific operation that contributes to the overall functionality of the banking system. Below is a complete explanation of each section of the code.

This part of the program defines the data structures and the push() function**,** which are the building blocks of the Simple Banking System.

1. **Header Files:**
   #include <stdio.h> and #include <stdlib.h> are used for input/output operations and dynamic memory allocation.

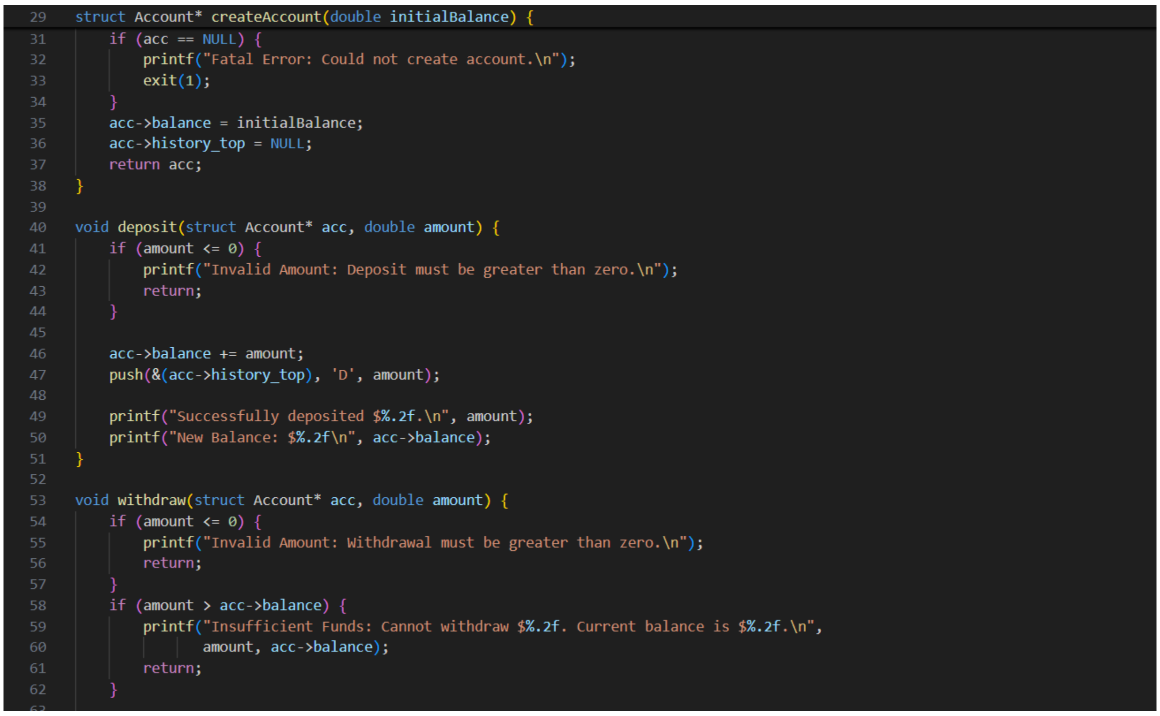


```c
#include <stdio.h>
#include <stdlib.h>

struct Transaction {
    char type;
    double amount;
    struct Transaction* next;
};

struct Account {
    double balance;
    struct Transaction* history_top;
};

void push(struct Transaction** top, char type, double amount) {
    struct Transaction* newTransaction = (struct Transaction*)malloc(sizeof(struct Transaction));

    if (newTransaction == NULL) {
        printf("Error: Memory allocation failed. Transaction not recorded.\n");
        return;
    }

    newTransaction->type = type;
    newTransaction->amount = amount;
    newTransaction->next = *top;
    *top = newTransaction;
}
```

2. **Structures:**
    - struct Transaction represents each transaction with three members — type (D or W), amount, and a pointer next to link with the previous transaction, forming a stack using linked lists.
    - struct Account holds the user's balance and a pointer history_top to the top of the transaction stack.

3. **push() Function:**
   This function adds a new transaction to the top of the stack.
    - It dynamically allocates memory for a new node using malloc().
    - If allocation fails, it prints an error message.
    - Otherwise, it sets the transaction type and amount, links it to the existing stack, and updates the top pointer.

In summary, this snippet sets up the foundation of the program**,** allowing transactions to be dynamically stored and managed using a linked-list-based stack**.**

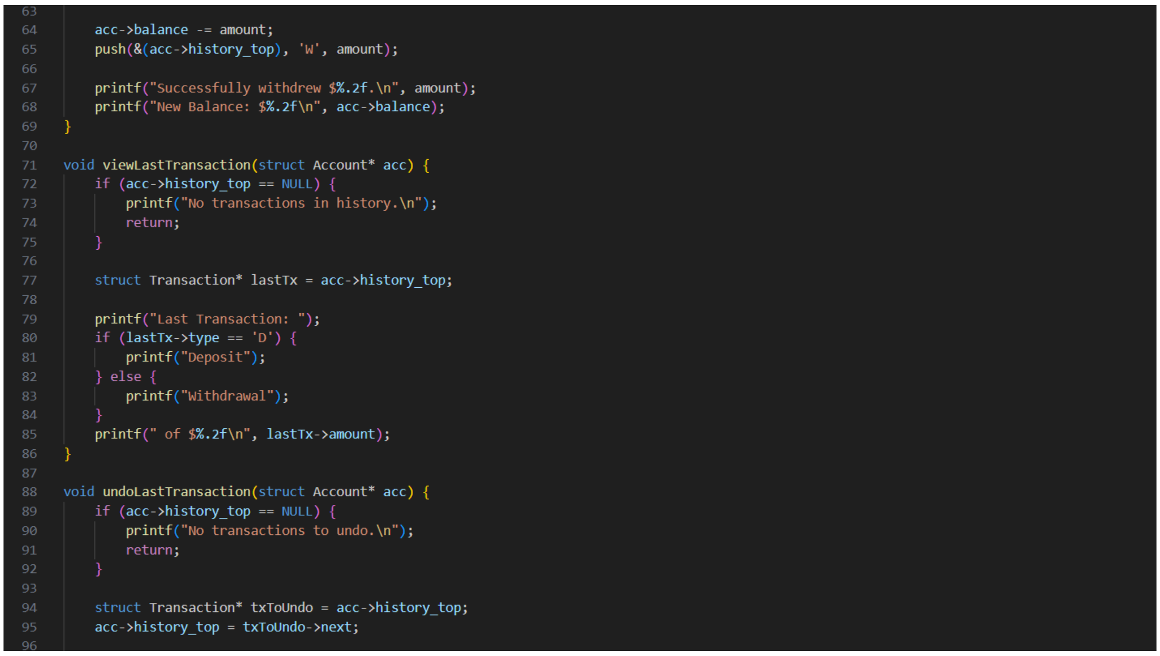


```c
struct Account* createAccount(double initialBalance) {
    if (acc == NULL) {
        printf("Fatal Error: Could not create account.\n");
        exit(1);
    }
    acc->balance = initialBalance;
    acc->history_top = NULL;
    return acc;
}

void deposit(struct Account* acc, double amount) {
    if (amount <= 0) {
        printf("Invalid Amount: Deposit must be greater than zero.\n");
        return;
    }

    acc->balance += amount;
    push(&(acc->history_top), 'D', amount);

    printf("Successfully deposited $%.2f.\n", amount);
    printf("New Balance: $%.2f\n", acc->balance);
}

void withdraw(struct Account* acc, double amount) {
    if (amount <= 0) {
        printf("Invalid Amount: Withdrawal must be greater than zero.\n");
        return;
    }
    if (amount > acc->balance) {
        printf("Insufficient Funds: Cannot withdraw $%.2f. Current balance is $%.2f.\n",
                amount, acc->balance);
        return;
    }
```

This part of the program defines the account creation, deposit, and withdrawal functions — the main operations of the banking system.

1. **createAccount() Function**
   This function is used to create a new account.
   - It dynamically allocates memory for an Account structure.
   - Initializes the account with a starting balance (initialBalance).
   - Sets history_top to NULL since no transactions exist initially.
   - If memory allocation fails, it prints an error message and exits the program.
     → In short, it sets up a new account ready for transactions.
2. **deposit() Function**
   This function performs the deposit operation.
   - It first checks if the deposit amount is greater than zero.
   - If valid, it adds the amount to the account balance.
   - The transaction is recorded by calling the push() function with type 'D'.
   - It then displays a success message and the new balance.
     → This ensures all deposits are properly updated and logged in the transaction history.

3. **withdraw() Function**
   This function handles the withdrawal process.
   - It validates the amount to ensure it is positive and not greater than the current balance.
   - If the balance is sufficient, the amount is subtracted from the account balance.
   - It records the transaction by calling push() with type 'W'.
   - If funds are insufficient, it displays a warning message.
     → This function prevents overdrawing and maintains accurate balance updates.

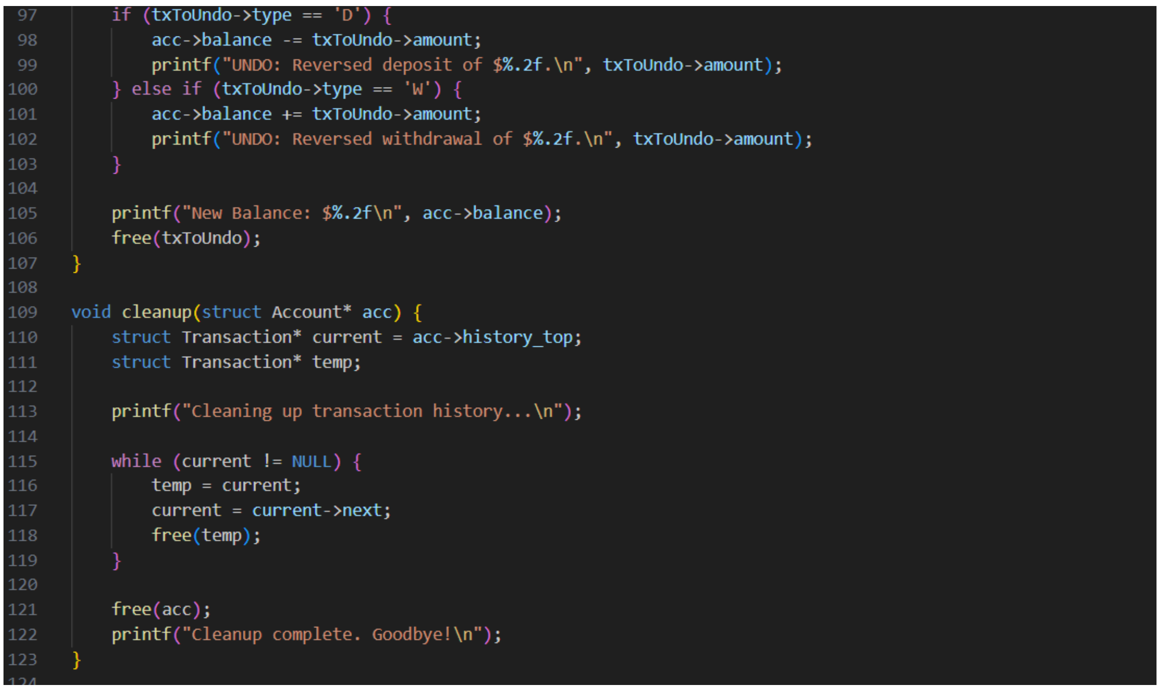


```c

    acc->balance -= amount;
    push(&(acc->history_top), 'W', amount);

    printf("Successfully withdrew $%.2f.\n", amount);
    printf("New Balance: $%.2f\n", acc->balance);
}

void viewLastTransaction(struct Account* acc) {
    if (acc->history_top == NULL) {
        printf("No transactions in history.\n");
        return;
    }

    struct Transaction* lastTx = acc->history_top;

    printf("Last Transaction: ");
    if (lastTx->type == 'D') {
        printf("Deposit");
    } else {
        printf("Withdrawal");
    }
    printf(" of $%.2f\n", lastTx->amount);
}

void undoLastTransaction(struct Account* acc) {
    if (acc->history_top == NULL) {
        printf("No transactions to undo.\n");
        return;
    }

    struct Transaction* txToUndo = acc->history_top;
    acc->history_top = txToUndo->next;

```

This part of the program defines two functions — viewLastTransaction() and undoLastTransaction() — that deal with viewing and reversing the most recent transaction.

1. **viewLastTransaction() Function**
   - This function displays the details of the most recent transaction made by the user.
   - It first checks if there are any transactions (history_top == NULL).
   - If no transaction exists, it displays a message saying *"No transactions in history."*
   - If a transaction exists, it fetches the top node of the stack and prints whether it was a Deposit **or** Withdrawal**,** along with the amount.
     → In short, it shows the latest transaction without modifying the stack.
   -
2. **undoLastTransaction() Function**
   - This function reverses the most recent transaction, acting as the undo feature**.**
   - It checks whether any transaction exists to undo.

- If there is one, it removes the top node from the stack and adjusts the balance accordingly:
  - If it was a Deposit, the amount is subtracted from the balance.
  - If it was a Withdrawal, the amount is added back to the balance.
- After updating, it deletes the transaction node from memory using free().
  → This ensures users can revert their last action safely, following the LIFO (Last-In-First-Out) principle.

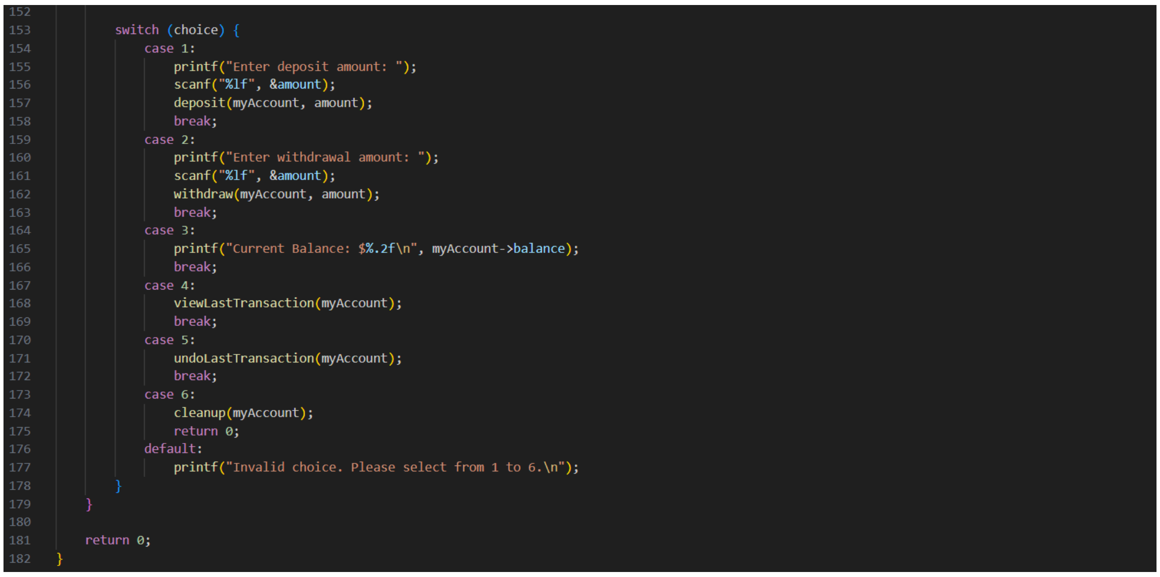


```c
    if (txToUndo->type == 'D') {
        acc->balance -= txToUndo->amount;
        printf("UNDO: Reversed deposit of $%.2f.\n", txToUndo->amount);
    } else if (txToUndo->type == 'W') {
        acc->balance += txToUndo->amount;
        printf("UNDO: Reversed withdrawal of $%.2f.\n", txToUndo->amount);
    }

    printf("New Balance: $%.2f\n", acc->balance);
    free(txToUndo);
}

void cleanup(struct Account* acc) {
    struct Transaction* current = acc->history_top;
    struct Transaction* temp;

    printf("Cleaning up transaction history...\n");

    while (current != NULL) {
        temp = current;
        current = current->next;
        free(temp);
    }

    free(acc);
    printf("Cleanup complete. Goodbye!\n");
}
```

This section contains the ending part of the undoLastTransaction() function and the cleanup() function, both essential for managing memory and data consistency in the program.

1. **undoLastTransaction() (Continuation)**
   - This part finalizes the undo operation.
   - It checks the transaction type:
     - If it was a Deposit ('D'), it subtracts the amount from the balance.
     - If it was a Withdrawal ('W'), it adds the amount back.
   - It then prints the updated balance and frees the memory of the undone transaction node using free().
     → This ensures accurate reversal of the last transaction and proper memory deallocation.
2. **cleanup() Function**
   - This function is called at the end of the program to release all dynamically allocated memory.
   - It loops through all transaction nodes and frees them one by one.

   - After freeing all transaction records, it frees the account structure itself.
   - Finally, it prints a confirmation message: *"Cleanup complete. Goodbye!"*
     → This prevents memory leaks and ensures that all resources are properly released before the program terminates.



```c

        switch (choice) {
            case 1:
                printf("Enter deposit amount: ");
                scanf("%lf", &amount);
                deposit(myAccount, amount);
                break;
            case 2:
                printf("Enter withdrawal amount: ");
                scanf("%lf", &amount);
                withdraw(myAccount, amount);
                break;
            case 3:
                printf("Current Balance: $%.2f\n", myAccount->balance);
                break;
            case 4:
                viewLastTransaction(myAccount);
                break;
            case 5:
                undoLastTransaction(myAccount);
                break;
            case 6:
                cleanup(myAccount);
                return 0;
            default:
                printf("Invalid choice. Please select from 1 to 6.\n");
        }
    }

    return 0;
}
```

1) viewLastTransaction() ➜ *Read-only peek at the latest action*

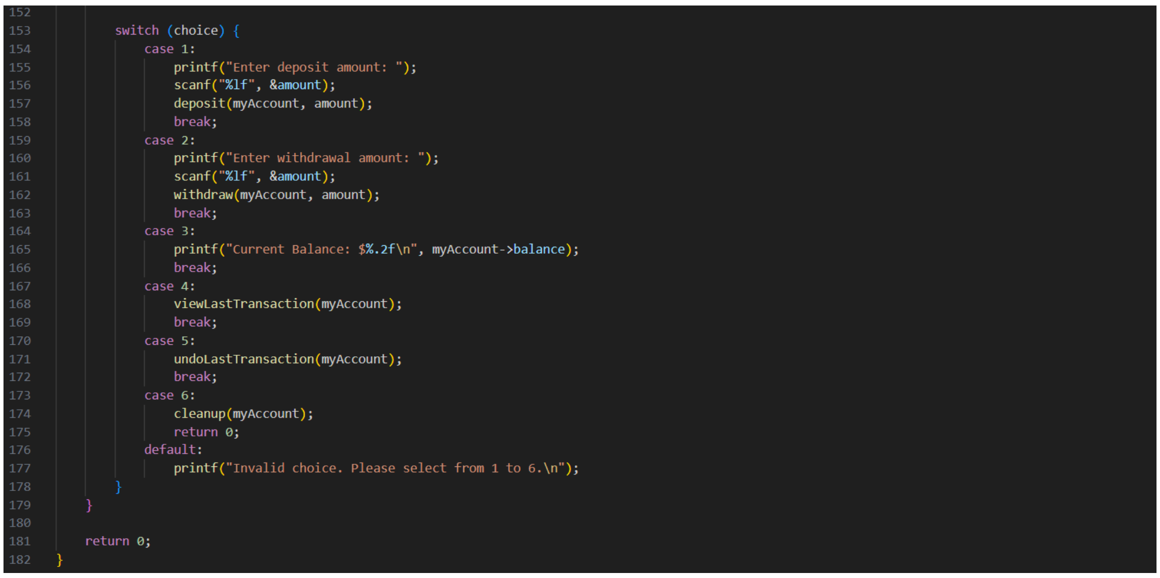
- Checks if acc->history_top is NULL. If yes, prints "No transactions in history." and returns.
- Otherwise takes the stack top (lastTx), prints whether it was a Deposit ('D') or Withdrawal ('W'), and shows its amount**.**
- Does not modify balance or the stack—purely informational.

2) undoLastTransaction() ➜ *LIFO "undo" using the stack*

- If no history (history_top == NULL), prints "No transactions to undo." and returns.
- Pops the top node (txToUndo) and reverses its effect:
   - If it was a Deposit, subtract that amount from balance.
   - If it was a Withdrawal, add that amount back to balance.
- Prints the new balance and frees the popped node.
- Guarantees correct reversal because the most recent transaction is always on top (stack/LIFO).

3) printMenu() ➔ *User guidance*

- Prints a clear, numbered list of options:
    1. Deposit
    2. Withdraw
    3. View Balance
    4. View Last Transaction
    5. Undo Last Transaction
    6. Exit
- Prompts for "Enter your choice:" each loop iteration.



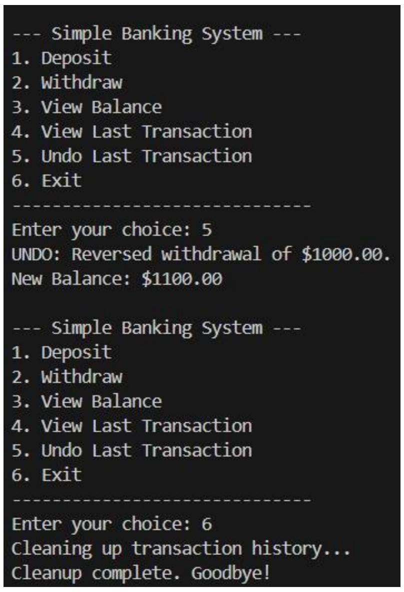
```c
        switch (choice) {
            case 1:
                printf("Enter deposit amount: ");
                scanf("%lf", &amount);
                deposit(myAccount, amount);
                break;
            case 2:
                printf("Enter withdrawal amount: ");
                scanf("%lf", &amount);
                withdraw(myAccount, amount);
                break;
            case 3:
                printf("Current Balance: $%.2f\n", myAccount->balance);
                break;
            case 4:
                viewLastTransaction(myAccount);
                break;
            case 5:
                undoLastTransaction(myAccount);
                break;
            case 6:
                cleanup(myAccount);
                return 0;
            default:
                printf("Invalid choice. Please select from 1 to 6.\n");
        }
    }

    return 0;
}
```

4) main() loop + switch(choice) ➔ *Program control center*

- Creates an account with ₹1000.00 initial balance and shows a welcome message.
- Enters an infinite loop:
    - Calls printMenu().
    - Reads choice with scanf. If input isn't a number, it clears the buffer, prints an error, and continues.
- Handles features via a **switch**:
    - case 1 (Deposit): asks for amount → deposit(myAccount, amount).
    - case 2 (Withdraw): asks for amount → withdraw(myAccount, amount).
    - case 3 (Balance): prints myAccount->balance.

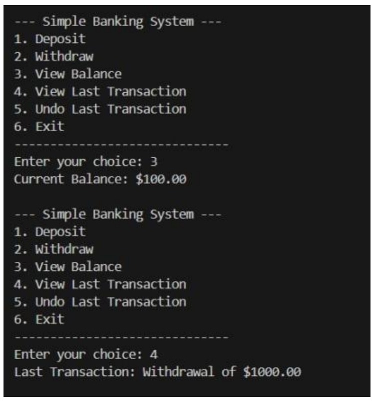
- case 4 (Last Tx): viewLastTransaction(myAccount).
- case 5 (Undo): undoLastTransaction(myAccount).
- case 6 (Exit): cleanup(myAccount); return 0; (graceful shutdown).
- default: invalid menu number → prints guidance to pick 1–6.

RESULTS AND ANALYSIS



```
--- Simple Banking System ---
1. Deposit
2. Withdraw
3. View Balance
4. View Last Transaction
5. Undo Last Transaction
6. Exit
------------------------------
Enter your choice: 5
UNDO: Reversed withdrawal of $1000.00.
New Balance: $1100.00

--- Simple Banking System ---
1. Deposit
2. Withdraw
3. View Balance
4. View Last Transaction
5. Undo Last Transaction
6. Exit
------------------------------
Enter your choice: 6
Cleaning up transaction history...
Cleanup complete. Goodbye!
```

Observed Runs



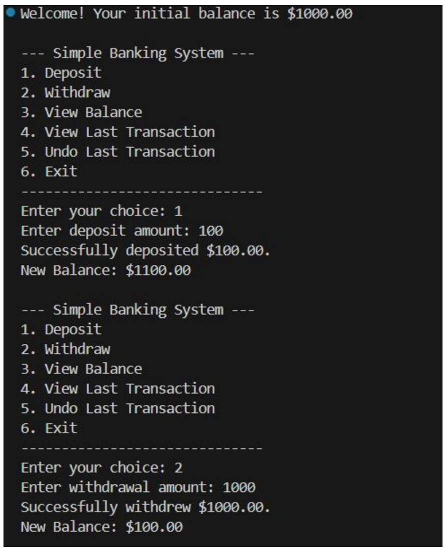
```
--- Simple Banking System ---
1. Deposit
2. Withdraw
3. View Balance
4. View Last Transaction
5. Undo Last Transaction
6. Exit
------------------------------
Enter your choice: 3
Current Balance: $100.00

--- Simple Banking System ---
1. Deposit
2. Withdraw
3. View Balance
4. View Last Transaction
5. Undo Last Transaction
6. Exit
------------------------------
Enter your choice: 4
Last Transaction: Withdrawal of $1000.00
```

Correctness & Functional Behavior

- **Balance updates are accurate**: After deposit(100) from $1000 → $1100; withdraw(1000) → $100; undo restores to $1100.
- **LIFO undo works**: The last operation before undo was a *withdrawal*, so the system adds the same amount back. This matches the stack design.
- **Introspection features work**: “View Balance” and “View Last Transaction” report the exact current numeric state and the most recent action without mutating data.
- **Graceful termination**: Explicit cleanup confirms that all dynamically allocated transaction nodes and the account are freed.

Usability & Messaging

- Menu-driven flow is clear; prompts for amounts appear only when relevant (deposit/withdraw).
- Status messages (success/undo/new balance) give immediate feedback, which is helpful for validation during testing and viva.
- Error paths (not shown here) already exist in code: invalid amounts and insufficient funds are handled with informative prints.



```
Welcome! Your initial balance is $1000.00

--- Simple Banking System ---
1. Deposit
2. Withdraw
3. View Balance
4. View Last Transaction
5. Undo Last Transaction
6. Exit
------------------------------
Enter your choice: 1
Enter deposit amount: 100
Successfully deposited $100.00.
New Balance: $1100.00

--- Simple Banking System ---
1. Deposit
2. Withdraw
3. View Balance
4. View Last Transaction
5. Undo Last Transaction
6. Exit
------------------------------
Enter your choice: 2
Enter withdrawal amount: 1000
Successfully withdrew $1000.00.
New Balance: $100.00
```

Data Structure Rationale (validated by results)

- Stack for history: The ability to undo only the last action is exactly what a stack guarantees; the results show this invariant is preserved.
- Linked list implementation: There’s no fixed cap on number of transactions; the program handled multiple operations without resizing or overflow concerns.

Performance Considerations

- **Time complexity**: Each operation—deposit, withdraw, view last, undo—is **O(1)** (constant time).
- **Space complexity**: Each new transaction allocates one node (**O(n)** total for n transactions). Undo frees nodes, reclaiming memory immediately.

Robustness & Integrity

- Input validation: Negative/zero amounts are rejected; overdraft attempts are blocked (ensures balance integrity).
- Memory safety**:** The cleanup sequence at exit, plus freeing during undo, prevents leaks—consistent with the "Cleanup complete" confirmation.
- State consistency: After every mutation, the program prints the *post-state* balance, which makes auditing easy.

Edge Cases (tested implicitly or by design)

- Undo with empty history: Handled—prints "No transactions to undo." (not shown in screenshots but present in code).
- View last with empty history: Handled—prints "No transactions in history."
- Large amounts: Limited by `double` precision; logically supported.
- Invalid input**:** Non-numeric menu entries are sanitized in `main()` by clearing the buffer.

Limitations

- Single user/session only; no persistence (state is lost after exit).
- No concurrency/security—sufficient for a DS lab, not for production banking.
- One-level undo (stack supports multi-level, but UX exposes only step-by-step undo—which is acceptable).

CONCLUSIONS

This project effectively demonstrates how data structures can be applied to build functional and interactive applications. Using stacks and linked lists, it replicates a real-time banking workflow. The modular design and use of dynamic memory allocation reflect industry practices in software development. This project strengthened understanding of stack operations, pointer handling, and linked list traversal — all essential skills in algorithm design and low-level programming.

FUTURE SCOPE

1. Introduce multiple account handling for different users.
2. Implement persistent storage using file handling for saving transaction history.
3. Add authentication features (username and PIN).
4. Integrate graphical user interface for better user experience.
5. Expand to handle interest calculation and generate transaction statements automatically.

REFERENCES

1. GeeksforGeeks – "Linked List Data Structure in C"

2. Tutorialspoint – "Dynamic Memory Allocation in C"

3. StudyTonight – "Stacks and Queues in Data Structures"

4. W3Schools – "C Programming Menu-driven Programs"

5. IncludeHelp – "Structure and Pointer in C Programming"