Restaurant Order Bill System USING C AND DATA STRUCTURES

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

P Karthik (PRN: 24070721030) &
A Tej Kiran Reddy (PRN: 24070721005

For the fulfillment

of

Data Structure and Algorithm Course

Symbiosis Institute of Technology, Hyderabad
Survey No. 292, Off Bangalore Highway, Modallaguda Village,
Nandigama Mandal, Ranga Reddy District, Telangana – 509217

CONTENTS

CONTENTS……...2

ABSTARCT……3

INTRODUCTION……3

METHODOLOGY……5

CODE EXPLINATION……7

RESULTS AND DISCUSSIONS……17

CONCLUSIONS……..23

FUTURE SCOPE…….....23

REFERENCES…….23

ABSTRACT

This project, the “Restaurant Order Bill System,” is a C program built to demonstrate the practical use of data structures in a real-world scenario. It functions as a simple billing machine for a restaurant. The core of the application is a **stack** data structure, which we implemented from the ground up using a **singly linked list** to store and manage the items ordered by a customer.

The system's main features include adding new items to a bill, calculating the running total cost, and a critical "undo" function that removes the last item added. We chose a stack for this project specifically to implement this undo feature. A stack operates on the **Last-In-First-Out (LIFO)** principle, which is the perfect logic for correcting mistakes—the last item entered is the first one that can be removed.

Using a linked list as the foundation for our stack makes the system dynamic and efficient, as it is not limited by a fixed size like an array. This allows a bill to grow to any length. This report explains our design process, provides the complete C code, and discusses the results from testing the system's functionality.

INTRODUCTION

Project Domain

In any restaurant, a billing system is very important. It keeps track of what customers order and calculates their total bill. When a waiter takes an order, they add items to the bill one by one.

However, people make mistakes. A waiter might enter the wrong item or add "Coke" twice by accident. The billing system must have an easy way to fix these mistakes. The most common and easiest fix is to remove the *last item* that was entered. This project builds a simple version of this system to solve that exact problem.

Problem Statement

A restaurant needs a basic program to manage a customer's bill. This program needs to be able to:

1. Add new items and their quantity to a bill.
2. Keep a running total of the bill's cost.
3. Let the user remove the *most recent* item they added (an "undo" feature).
4. Show a list of all items currently on the bill.
5. Work for any number of items, not just a fixed amount.

We had to choose the right data structure to make adding items and "undoing" them very fast and simple.

Choice of Technology: The C Language

We chose to use the C programming language for a few main reasons:

- **Memory Control:** C lets you control memory directly using malloc() and free(). This is needed to build our own linked list from scratch. We didn't want to use a pre-built tool; we wanted to build the data structure ourselves for this course.
- **Speed:** C is very fast, which is good for any system that users interact with.
- **Good for Learning:** C makes you manage everything, like pointers and memory. This helps in deeply understanding how data structures actually work, which is the point of this course.

The Role of Data Structures

A data structure is just a way to organize data. The data structure we choose decides how good our program will be.

- **Why not an array?** We could have used an array (like OrderItem[100]), but that has a fixed size. If a customer orders 101 items, the program would fail. It's also wasteful if they only order 5 items.
- **Why a Linked List?** A linked list is dynamic. It can grow as we add more items. We only use memory when we add a new item, which is very efficient.
- **Why a Stack?** The "undo" feature is the key. A stack is a **Last-In-First-Out (LIFO)** data structure. This means the last item you *push* (add) is the first item you *pop* (remove). This is exactly what an "undo" button does.

This project uses a linked list to build a stack. This gives us the LIFO logic of a stack and the dynamic size of a linked list.

Project Objectives

Our main goals for this project were:

1. To learn how to build a stack using a singly linked list in C.
2. To use this stack to solve a real-world problem (a billing system).
3. To practice C programming, especially using structs, pointers, and malloc/free.
4. To write a clean, working program with a simple menu for the user.
5. To see how fast our stack operations were.

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 structs (data structures):

- OrderItem Structure (The Node):

Each item added to the bill is represented as a node. It holds the information for one item (like item name, quantity, and price). It also has a next pointer, which links it to the node that was added before it. These nodes are linked together to form a stack.

- Bill Structure (The Stack Manager):

This struct maintains the overall state of the bill. It holds the totalAmount and a pointer called order_top. This pointer always points to the last item added (the top of the stack). This allows the system to efficiently track and manage the order.

By using a linked list to implement the stack, the system achieves flexibility, allowing the list of ordered items to grow dynamically without a predefined limit.

2. Core Functional Modules

The project is composed of the following main functional components:

- createBill():

Initializes a new, empty bill. It sets the totalAmount to 0 and the order_top pointer to NULL. Uses malloc() for dynamic allocation.

- addItem():

Adds a new item to the bill. It calculates the item's cost, adds it to the totalAmount, and records the transaction by pushing a new OrderItem node onto the top of the stack.

- viewFullBill():

Traverses the entire stack from top to bottom, printing the details of every item currently on the bill. It finishes by printing the final totalAmount.

- viewLastItem():

Displays the details of the most recent item added (the order_top node) without removing it. This is a "peek" operation on the stack.

- undoLastItem():

Implements the stack pop operation. It removes the top OrderItem node, subtracts that item's price from the totalAmount, and then frees the node's memory. This reverses the last addItem action.

- cleanup():

Frees all dynamically allocated memory (all OrderItem nodes and the Bill structure itself) before the program terminates to prevent memory leaks.

3. Algorithm Flow

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

1. Start the program and create a new, empty bill using createBill().
2. Display a menu of available options (Add Item, View Bill, View Last Item, Remove Last Item, Exit).
3. Take user input for the chosen operation.
4. Execute the corresponding function based on the user's choice.
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 the program's execution.

4. Advantages of Using Stack and Linked List

- Eliminates the need for static arrays and allows the bill to hold an unlimited number of items.
- Makes the “Remove Last Item” (Undo) feature intuitive and simple to build using **LIFO (Last-In-First-Out)** logic.
- Ensures that the most common operations (addItem and undoLastItem) are extremely fast, taking constant time ($O(1)$).
- Ensures real-time dynamic allocation and efficient use of memory.
- Enhances system robustness through proper memory 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 restaurant billing system.

The entire code is modular, meaning each function performs a specific operation that contributes to the overall functionality of the billing system.

Below is a complete explanation of each section of the code.

1. Header Files, Structures, and addItem() (Push)

This part of the program defines the data structures and the addItem() function, which are the building blocks of the Restaurant Billing System.

1. **Header Files:** #include <stdio.h>, #include <stdlib.h>, and #include <string.h> are used for input/output operations, dynamic memory allocation, and copying strings (like item names).
2. **Structures:**
    - struct OrderItem represents each item on the bill as a node. It has members for itemName, quantity, price, and a pointer next to link with the previous item, forming a stack using a linked list.
    - struct Bill holds the bill's running totalAmount and a pointer order_top to the top of the item stack.
3. **addItem() (Push) Function:** This function adds a new item to the top of the stack.
    - It gets the user's choice and quantity.
    - It dynamically allocates memory for a new OrderItem node using malloc().
    - If allocation fails, it prints an error message.
    - Otherwise, it sets the item's details (name, quantity, price), links it to the existing stack (newNode->next = myBill->order_top), and updates the order_top pointer to this new node.
    - Finally, it adds the item's price to the totalAmount.

4. In summary, this snippet sets up the foundation, allowing ordered items to be dynamically stored and managed using a linked-list-based stack.

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

typedef struct OrderItem {
    char itemName[50];
    int quantity;
    double price;
    struct OrderItem* next;
} OrderItem;

typedef struct Bill {
    double totalAmount;
    OrderItem* order_top;
} Bill;

Bill* createBill();
void displayMenu();
void addItem(Bill* myBill);
void undoLastItem(Bill* myBill);
void viewLastItem(Bill* myBill);
void viewFullBill(Bill* myBill);
void cleanup(Bill* myBill);
void printMainMenu();
int getValidIntInput();

int main() {
    Bill* myBill = createBill();
    int choice = 0;

    printf("Welcome to the Restaurant Billing System!\n");
    printf("Created by P Karthik & A Tej Kiran Reddy\n");

    while (1) {
        printMainMenu();
        choice = getValidIntInput();

        switch (choice) {
            case 1:
                addItem(myBill);
                break;
            case 2:
                viewFullBill(myBill);
                break;
            case 3:
```

```
                viewLastItem(myBill);
                break;
            case 4:
                undoLastItem(myBill);
                break;
            case 5:
                printf("Thank you for using the system.\n");
                cleanup(myBill);
                return 0;
            default:
                printf("Invalid choice. Please enter a number between 1 and 5.\n");
        }
        printf("\n");
    }

    return 0;
}

void printMainMenu() {
    printf("----------------------------------------\n");
    printf("Main Menu:\n");
    printf("1. Add Item to Bill\n");
    printf("2. View Full Bill\n");
    printf("3. View Last Item Added\n");
    printf("4. Remove Last Item (Undo)\n");
    printf("5. Exit\n");
    printf("----------------------------------------\n");
    printf("Enter your choice: ");
}

int getValidIntInput() {
    int value;
    char buffer[100];

    if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
        if (sscanf(buffer, "%d", &value) == 1) {
            return value;
        }
    }
    return -1;
}

Bill* createBill() {
    Bill* myBill = (Bill*)malloc(sizeof(Bill));
    if (myBill == NULL) {
        printf("FATAL ERROR: Could not allocate memory for the bill.\n");
```

```
92          exit(1);
93      }
94      myBill->totalAmount = 0.0;
95      myBill->order_top = NULL;
96      return myBill;
97  }
98
99  void displayMenu() {
100     printf("--- Restaurant Menu ---\n");
101     printf("1. Pizza    - ₹350.00\n");
102     printf("2. Burger   - ₹150.00\n");
103     printf("3. Pasta    - ₹250.00\n");
104     printf("4. Coke     - ₹60.00\n");
105     printf("5. Sandwich - ₹120.00\n");
106     printf("-----------------------\n");
107 }
108
109 void addItem(Bill* myBill) {
110     displayMenu();
111     int choice = 0;
112     int quantity = 0;
113     char itemName[50];
114     double unitPrice = 0.0;
115     double itemTotalPrice = 0.0;
116
117     printf("Enter item number: ");
118     choice = getValidIntInput();
119
120     printf("Enter quantity: ");
121     quantity = getValidIntInput();
122
123     if (quantity <= 0) {
124         printf("Invalid quantity. Must be 1 or more. Returning to menu.\n");
125         return;
126     }
127
128     switch (choice) {
129         case 1:
130             strcpy(itemName, "Pizza");
131             unitPrice = 350.00;
132             break;
133         case 2:
134             strcpy(itemName, "Burger");
135             unitPrice = 150.00;
136             break;
137         case 3:
```

```
            strcpy(itemName, "Pasta");
            unitPrice = 250.00;
            break;
        case 4:
            strcpy(itemName, "Coke");
            unitPrice = 60.00;
            break;
        case 5:
            strcpy(itemName, "Sandwich");
            unitPrice = 120.00;
            break;
        default:
            printf("Invalid item choice. Returning to menu.\n");
            return;
    }

    itemTotalPrice = unitPrice * quantity;

    OrderItem* newNode = (OrderItem*)malloc(sizeof(OrderItem));
    if (newNode == NULL) {
        printf("ERROR: Could not allocate memory for new item.\n");
        return;
    }

    strcpy(newNode->itemName, itemName);
    newNode->quantity = quantity;
    newNode->price = itemTotalPrice;

    newNode->next = myBill->order_top;
    myBill->order_top = newNode;

    myBill->totalAmount += itemTotalPrice;

    printf("\nSUCCESS: Added %d x %s.\n", quantity, itemName);
    printf("New Total Amount: ₹%.2f\n", myBill->totalAmount);
}

void undoLastItem(Bill* myBill) {
    if (myBill->order_top == NULL) {
        printf("No items in the bill to undo.\n");
        return;
    }

    OrderItem* nodeToFree = myBill->order_top;

    double removedPrice = nodeToFree->price;
```

```c
    char removedName[50];
    strcpy(removedName, nodeToFree->itemName);
    int removedQty = nodeToFree->quantity;

    myBill->order_top = nodeToFree->next;

    myBill->totalAmount -= removedPrice;

    free(nodeToFree);

    printf("\nUNDO: Removed %d x %s (worth ₹%.2f).\n", removedQty, removedName,
        removedPrice);
    printf("New Total Amount: ₹%.2f\n", myBill->totalAmount);
}

void viewLastItem(Bill* myBill) {
    if (myBill->order_top == NULL) {
        printf("No items in the bill yet.\n");
        return;
    }

    OrderItem* topItem = myBill->order_top;
    printf("--- Last Item Added ---\n");
    printf("Item:     %s\n", topItem->itemName);
    printf("Quantity: %d\n", topItem->quantity);
    printf("Price:    ₹%.2f\n", topItem->price);
}

void viewFullBill(Bill* myBill) {
    OrderItem* current = myBill->order_top;

    printf("\n========== FULL BILL ==========\n");

    if (current == NULL) {
        printf("The bill is currently empty.\n");
    } else {
        while (current != NULL) {
            printf("Item: %-10s | Qty: %-3d | Price: ₹%.2f\n",
                   current->itemName, current->quantity, current->price);
            current = current->next;
        }
    }

    printf("================================\n");
    printf("FINAL TOTAL: ₹%.2f\n", myBill->totalAmount);
    printf("================================\n");
```

```
229 }
230
231 void cleanup(Bill* myBill) {
232     printf("Cleaning up order history...\n");
233     OrderItem* current = myBill->order_top;
234     OrderItem* tempNode;
235
236     while (current != NULL) {
237         tempNode = current;
238         current = current->next;
239         free(tempNode);
240     }
241
242     free(myBill);
243     printf("Cleanup complete. Goodbye!\n");
244 }
```

2. createBill(), viewFullBill(), and viewLastItem()

This part of the program defines the bill creation and the two "read-only" functions.

1. **createBill() Function:** This function is used to create a new, empty bill.
    - It dynamically allocates memory for a Bill structure.
    - It initializes the totalAmount to 0.0.
    - It sets order_top to NULL since no items exist initially.
    - If memory allocation fails, it prints an error and exits.

→ In short, it sets up a new bill, ready to receive orders.

2. **viewFullBill() Function:** This function traverses the entire stack to print all items.
    - It creates a temporary pointer current and sets it to order_top.
    - It loops as long as current is not NULL.
    - Inside the loop, it prints the current item's details and then moves to the next item (current = current->next).
    - It finishes by printing the final totalAmount.

→ This allows the user to see everything on the bill at any time.

3. **viewLastItem() (Peek) Function:** This function displays the details of the most recent item added.
    - It first checks if the stack is empty (order_top == NULL).
    - If no item exists, it displays "No items in the bill yet."
    - If an item exists, it fetches the order_top node and prints its details (name, quantity, price).

3. undoLastItem() (Pop) Function

This part of the program defines the undoLastItem() function, which reverses the most recent addItem operation.

1. **undoLastItem() Function:**
    - This function reverses the most recent transaction, acting as the "undo" feature.
    - It first checks if any item exists to undo (order_top == NULL).
    - If there is one, it prepares to remove the top node:
        - It saves a temporary pointer to the order_top node (e.g., nodeToFree).
        - It gets the price of that node.
        - It subtracts this price from the bill->totalAmount.
        - It updates the order_top pointer to point to the *next* node in the list (myBill->order_top = nodeToFree->next).
    - After updating the stack, it deletes the old top node from memory using free(nodeToFree).

4. cleanup() and main() Functions

This section contains the cleanup() function for memory management and the main() function that controls the program.

1. **cleanup() Function:**

- This function is called at the end of the program to release all dynamically allocated memory.
- It loops through all OrderItem nodes (starting from order_top) and frees them one by one.
- After freeing all the item records, it frees the Bill structure itself.
- Finally, it prints a confirmation message: “Cleanup complete. Goodbye!”

→ This prevents memory leaks and ensures all resources are properly released.

2. **main() Function and printMainMenu():**
   - This is the control center of the program.
   - It calls createBill() to set up the bill.
   - It enters an infinite while(1) loop:
     - Calls printMainMenu() to show the user their options.
     - Reads the user's choice.
     - Uses a switch statement to call the correct function (addItem, viewFullill, undoLastItem, etc.).
     - case 5 (Exit) calls cleanup() and then uses return 0; to shut down the program gracefully.

Functionality Summary

1. **viewLastItem() ➜ Read-only peek at the latest action**
   - Checks if myBill->order_top is NULL. If yes, prints “No items in the bill yet.”
   - Otherwise, takes the stack top (topItem) and prints its name, quantity, and price.
   - Does not modify the totalAmount or the stack—purely informational.
2. **undoLastItem() ➜ LIFO “undo” using the stack**
   - If no history (order_top == NULL), prints “No items in the bill to undo.”
   - Pops the top node (nodeToFree) and reverses its effect:

        - Subtracts nodeToFree->price from totalAmount.
        - Updates order_top to point to nodeToFree->next.
    - Prints the new totalAmount and frees the popped node.
    - Guarantees correct reversal because the most recent item is always on top (stack/LIFO).

3. **printMainMenu() ➜ User guidance**
    - Prints a clear, numbered list of options:
        1. Add Item to Bill
        2. View Full Bill
        3. View Last Item Added
        4. Remove Last Item (Undo)
        5. Exit
    - Prompts for "Enter your choice:" each loop.

4. **Main () loop + switch(choice) ➜ Program control center**
    - Creates an empty bill with createBill() (Total: ₹0.00).
    - Enters an infinite loop:
        - Calls printMainMenu().
        - Reads the choice.
    - Handles features via a switch:
        - case 1 (Add Item): Asks for item/quantity → addItem(myBill).
        - case 2 (View Bill): → viewFullBill(myBill).
        - case 3 (Last Item): → viewLastItem(myBill).
        - case 4 (Undo): → undoLastItem(myBill).
        - case 5 (Exit): → cleanup(myBill); return 0; (graceful shutdown).
        - default: Invalid menu number → prints an error message.

RESULTS AND DISCUSSION

```
Welcome to the Restaurant Billing System!
Created by P Karthik & A Tej Kiran Reddy
----------------------------------------
Main Menu:
1. Add Item to Bill
2. View Full Bill
3. View Last Item Added
4. Remove Last Item (Undo)
5. Exit
----------------------------------------
Enter your choice: 2

========== FULL BILL ==========
The bill is currently empty.
==============================
FINAL TOTAL: ₹0.00
==============================


----------------------------------------
Main Menu:
1. Add Item to Bill
2. View Full Bill
3. View Last Item Added
4. Remove Last Item (Undo)
5. Exit
----------------------------------------
Enter your choice: 1
--- Restaurant Menu ---
1. Pizza     - ₹350.00
2. Burger    - ₹150.00
3. Pasta     - ₹250.00
4. Coke      - ₹60.00
5. Sandwich  - ₹120.00
```

```
----------------------
Enter item number: 1
Enter quantity: 2

SUCCESS: Added 2 x Pizza.
New Total Amount: ₹700.00

----------------------------------------
Main Menu:
1. Add Item to Bill
2. View Full Bill
3. View Last Item Added
4. Remove Last Item (Undo)
5. Exit
----------------------------------------
Enter your choice: 1
--- Restaurant Menu ---
1. Pizza     - ₹350.00
2. Burger    - ₹150.00
3. Pasta     - ₹250.00
4. Coke      - ₹60.00
5. Sandwich - ₹120.00
----------------------
Enter item number: 4
Enter quantity: 3

SUCCESS: Added 3 x Coke.
New Total Amount: ₹880.00

----------------------------------------
Main Menu:
1. Add Item to Bill
2. View Full Bill
3. View Last Item Added
4. Remove Last Item (Undo)
5. Exit
```

```
----------------------------------------
Enter your choice: 3
--- Last Item Added ---
Item:     Coke
Quantity: 3
Price:    ₹180.00

----------------------------------------
Main Menu:
1. Add Item to Bill
2. View Full Bill
3. View Last Item Added
4. Remove Last Item (Undo)
5. Exit
----------------------------------------
Enter your choice: 2

========== FULL BILL ==========
Item: Coke       | Qty: 3   | Price: ₹180.00
Item: Pizza      | Qty: 2   | Price: ₹700.00
==============================
FINAL TOTAL: ₹880.00
==============================


----------------------------------------
Main Menu:
1. Add Item to Bill
2. View Full Bill
3. View Last Item Added
4. Remove Last Item (Undo)
5. Exit
----------------------------------------
Enter your choice: 4

UNDO: Removed 3 x Coke (worth ₹180.00).
New Total Amount: ₹700.00
```

```
New Total Amount: ₹700.00

----------------------------------------
Main Menu:
1. Add Item to Bill
2. View Full Bill
3. View Last Item Added
4. Remove Last Item (Undo)
5. Exit
----------------------------------------
Enter your choice: 2

========== FULL BILL ==========
Item: Pizza      | Qty: 2   | Price: ₹700.00
==============================
FINAL TOTAL: ₹700.00
==============================

----------------------------------------
Main Menu:
1. Add Item to Bill
2. View Full Bill
3. View Last Item Added
4. Remove Last Item (Undo)
5. Exit
----------------------------------------
Enter your choice: 5
Thank you for using the system.
Cleaning up order history...
Cleanup complete. Goodbye!
```

Observed Runs

1. **Add Items**
    - Initial Total: ₹0.00
    - Add (2x Pizza): "SUCCESS: Added 2 x Pizza. New Total Amount: ₹700.00."
    - Add (3x Coke): "SUCCESS: Added 3 x Coke. New Total Amount: ₹880.00."
2. **Querying State**
    - View Full Bill (option 2): Prints both items and "FINAL TOTAL: ₹880.00."

- View Last Item (option 3): “Last Item Added: Coke, Quantity: 3, Price: ₹180.00.”

3. **Undo + Exit**
    - Undo Last (option 4): “UNDO: Removed 3 x Coke (worth ₹180.00). New Total Amount: ₹700.00.”
    - Exit (option 5): “Cleaning up order history… Cleanup complete. Goodbye!”

Correctness & Functional Behavior

- **Total updates are accurate:** After addItem(2x Pizza) from ₹0 → ₹700; addItem(3x Coke) → ₹880; undoLastItem() restores to ₹700. The math is correct.
- **LIFO undo works:** The last operation before undo was adding "Coke". The system correctly removes "Coke" and subtracts its price. This matches the stack design.
- **Introspection features work:** “View Full Bill” and “View Last Item” report the exact current state of the bill without changing any data.
- **Graceful termination:** The explicit cleanup() message confirms that all dynamically allocated OrderItem nodes and the Bill structure are freed.

Usability & Messaging

- Menu-driven flow is clear; prompts for item number and quantity appear only when addItem is selected.
- Status messages (e.g., "SUCCESS: Added...", "UNDO: Removed...") give immediate feedback, which is helpful for confirming actions.
- Error paths (shown in sample run) such as "No items in the bill to undo" are handled with informative prints. The code also handles invalid quantity and invalid menu choices.

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 LIFO invariant is preserved.
- **Linked list implementation:** There is no fixed limit on the number of items on the bill. The program handled multiple addItem operations without any overflow concerns.

Performance Considerations

- **Time complexity:** Each core operation—addItem (push), undoLastItem (pop), and viewLastItem (peek)—is **$O(1)$** (constant time), as they only interact with the order_top of the stack.
- viewFullBill is an **$O(n)$** (linear time) operation, as it must visit every node (item) in the stack.
- **Space complexity:** Each new item added allocates one new node, leading to a total space complexity of **$O(n)$** for n items on the bill. undoLastItem frees nodes, reclaiming memory immediately.

Robustness & Integrity

- **Input validation:** Invalid quantities (<= 0) and invalid menu choices are rejected.
- **Memory safety:** The cleanup() function at exit, plus the free() call during undoLastItem, prevents memory leaks. This is confirmed by the "Cleanup complete" message.
- **State consistency:** After every mutation (addItem, undoLastItem), the program prints the new totalAmount, which makes auditing the program's state easy.

Edge Cases (tested implicitly or by design)

- **Undo with empty history:** Handled—prints "No items in the bill to undo." (shown in sample run).
- **View last with empty history:** Handled—prints "No items in the bill yet."
- **Invalid input:** Non-numeric menu entries and invalid menu numbers are sanitized by getValidIntInput() and the default switch case.

Limitations

- Single user/bill only; no persistence (the bill is lost after exiting).
- The menu is hard-coded; it cannot be changed without recompiling the program.
- The system only allows undoing the *last* item. It is not possible to remove an item from the middle of the bill.

CONCLUSION

This project effectively demonstrates how data structures can be applied to build functional and interactive applications. Using a **stack implemented with a linked list**, the program successfully replicates a real-time restaurant billing workflow. The modular design and use of dynamic memory allocation reflect strong programming practices. This project strengthened our understanding of stack operations (push, pop, peek), pointer handling, and linked list traversal — all essential skills in algorithm design and C programming.

FUTURE SCOPE

1. **Implement Multiple Tables:** Use an array of Bill* pointers or a hash table to manage multiple active bills at the same time (e.g., Bill* tables[10]).
2. **Implement Persistent Storage:** Use file handling (fopen, fwrite) to save the bill to a text file, so it is not lost when the program closes.
3. **Dynamic Menu:** Load the restaurant menu (items and prices) from an external menu.txt file at startup.
4. **Add Kitchen Order Queue:** Implement a **Queue** data structure. When an item is added to the bill (pushed to the stack), it is also *enqueued* to a "Kitchen" queue to be "cooked" in FIFO (First-In-First-Out) order.
5. **Integrate Graphical User Interface (GUI):** Rebuild the frontend using a library like GTK+ or Qt to create a clickable, user-friendly interface.

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"