DSA Assignment-1:

Submitted by: Sibghatullah
Course: DSA_theory
Roll no: 230896

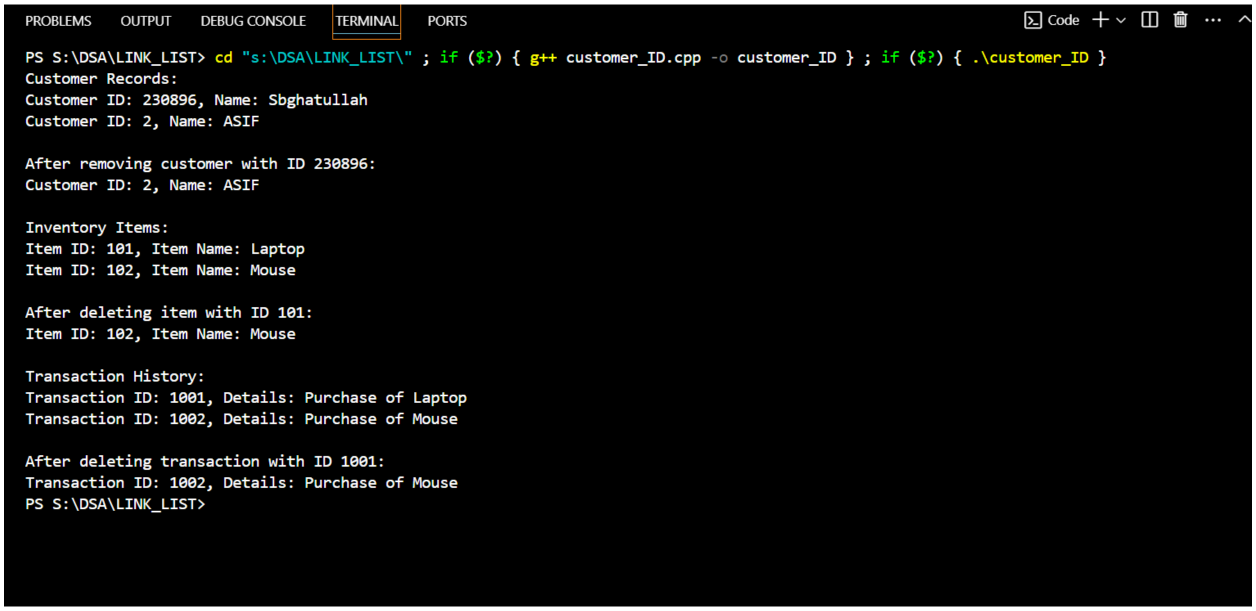
TASK: 1-3-5:

```
#include <iostream>

#include <string>


class CustomerNode {

public:

    std::string name;

    int id;

    CustomerNode* next;


    CustomerNode(std::string customerName, int customerId)

        : name(customerName), id(customerId), next(nullptr) {}

};


class CustomerLinkedList {

private:
    CustomerNode* head;

public:
    CustomerLinkedList() : head(nullptr) {}

    void addCustomer(const std::string& name, int id) {
        CustomerNode* newNode = new CustomerNode(name, id);
        if (!head) {
            head = newNode;
        } else {
            CustomerNode* current = head;
            while (current->next) {
                current = current->next;
            }
            current->next = newNode;
        }
    }

    void removeCustomer(int id) {
        if (!head) return;

        if (head->id == id) {
            CustomerNode* temp = head;
            head = head->next;
            delete temp;
            return;
        }

        CustomerNode* current = head;
        while (current->next && current->next->id != id) {
            current = current->next;
        }

        if (current->next) {
            CustomerNode* temp = current->next;
            current->next = current->next->next;
            delete temp;
        }
    }

    void displayCustomers() const {
        CustomerNode* current = head;
        while (current) {
            std::cout << "Customer ID: " << current->id << ", Name: " << current->name << '\n';
            current = current->next;
        }
    }
};



class InventoryNode {
public:
    std::string itemName;
    int id;
    InventoryNode* next;
    InventoryNode* prev;
    InventoryNode(std::string name, int itemId)
        : itemName(name), id(itemId), next(nullptr), prev(nullptr) {}
};

class InventoryDoublyLinkedList {
private:
    InventoryNode* head;

public:
    InventoryDoublyLinkedList() : head(nullptr) {}

    void addItem(const std::string& name, int id) {
        InventoryNode* newNode = new InventoryNode(name, id);
        if (!head) {
            head = newNode;
        } else {
            InventoryNode* current = head;
            while (current->next) {
                current = current->next;
            }
            current->next = newNode;
            newNode->prev = current;
        }
    }

    void removeItem(int id) {
        InventoryNode* current = head;
        while (current) {
            if (current->id == id) {
                if (current->prev) {
                    current->prev->next = current->next;
                } else {
                    head = current->next; // Update head if needed
                }
                if (current->next) {
                    current->next->prev = current->prev;
                }
                delete current;
                return;
            }
            current = current->next;
        }
    }

    void displayItems() const {
        InventoryNode* current = head;
        while (current) {
            std::cout << "Item ID: " << current->id << ", Item Name: " << current->itemName << '\n';
            current = current->next;
        }
    }
};




class TransactionNode {
public:
    std::string transactionDetails;
    int id;

    TransactionNode* next;


    TransactionNode(std::string details, int transactionId)

        : transactionDetails(details), id(transactionId), next(nullptr) {}

};


class TransactionCircularLinkedList {

private:

    TransactionNode* head;


public:

    TransactionCircularLinkedList() : head(nullptr) {}


    void addTransaction(const std::string& details, int id) {

        TransactionNode* newNode = new TransactionNode(details, id);

        if (!head) {

            head = newNode;

            newNode->next = head; // Point to itself

        } else {

            TransactionNode* current = head;

            while (current->next != head) {

                current = current->next;

            }

            current->next = newNode;

            newNode->next = head; // Make it circular

        }

    }
```

```
void removeTransaction(int id) {
    if (!head) return;

    TransactionNode* current = head;
    TransactionNode* previous = nullptr;

    do {
        if (current->id == id) {
            if (previous) {
                previous->next = current->next;
            } else {
                if (current->next == head) {
                    head = nullptr; // Only one node
                } else {
                    TransactionNode* last = head;
                    while (last->next != head) {
                        last = last->next;
                    }
                    head = current->next;
                    last->next = head;
                }
            }
            delete current;
            return;
        }
        previous = current;
        current = current->next;
    } while (current != head);
}
```

```
    void displayTransactions() const {
        if (!head) return;

        TransactionNode* current = head;
        do {
            std::cout << "Transaction ID: " << current->id << ", Details: " << current->transactionDetails << '\n';
            current = current->next;
        } while (current != head);
    }
};


int main() {
    // Customer Records Management
    CustomerLinkedList customerList;
    customerList.addCustomer("Sbghatullah", 230896);
    customerList.addCustomer("ASIF", 2);
    std::cout << "Customer Records:\n";
    customerList.displayCustomers();

    customerList.removeCustomer(230896);
    std::cout << "\nAfter removing customer with ID 230896:\n";
    customerList.displayCustomers();

    // Inventory Tracking
    InventoryDoublyLinkedList inventoryList;
    inventoryList.addItem("Laptop", 101);
    inventoryList.addItem("Mouse", 102);
    std::cout << "\nInventory Items:\n";
    inventoryList.displayItems();

    inventoryList.removeItem(101);
    std::cout << "\nAfter deleting item with ID 101:\n";
    inventoryList.displayItems();

    // Transaction History Management
    TransactionCircularLinkedList transactionList;
    transactionList.addTransaction("Purchase of Laptop", 1001);
    transactionList.addTransaction("Purchase of Mouse", 1002);
    std::cout << "\nTransaction History:\n";
    transactionList.displayTransactions();

    transactionList.removeTransaction(1001);
    std::cout << "\nAfter deleting transaction with ID 1001:\n";
    transactionList.displayTransactions();

    return 0;
}
```

OUTPUT:



```
PS S:\DSA\LINK_LIST> cd "s:\DSA\LINK_LIST\" ; if ($?) { g++ customer_ID.cpp -o customer_ID } ; if ($?) { .\customer_ID }
Customer Records:
Customer ID: 230896, Name: Sbghatullah
Customer ID: 2, Name: ASIF

After removing customer with ID 230896:
Customer ID: 2, Name: ASIF

Inventory Items:
Item ID: 101, Item Name: Laptop
Item ID: 102, Item Name: Mouse

After deleting item with ID 101:
Item ID: 102, Item Name: Mouse

Transaction History:
Transaction ID: 1001, Details: Purchase of Laptop
Transaction ID: 1002, Details: Purchase of Mouse

After deleting transaction with ID 1001:
Transaction ID: 1002, Details: Purchase of Mouse
PS S:\DSA\LINK_LIST>
```

Task 2: Application of Singly Linked List

Suitability of Singly Linked List for Customer Records Management

A singly linked list is an ideal data structure for managing customer records due to its straightforward implementation and efficient memory usage.

Advantages:

1. **Dynamic Size**: Singly linked lists can grow and shrink dynamically, which is beneficial for applications where the number of customer records can change frequently.
2. **Efficient Insertions/Deletions**: Adding a new customer at the end of the list or removing a customer by ID can be done with minimal overhead, as it only requires updating a few pointers.
3. **Memory Efficiency**: Each node contains only one pointer to the next node, which reduces memory overhead compared to other structures like doubly linked lists.

Limitations:

1. **Sequential Access**: Searching for a customer by ID requires traversing the list from the head to the desired node, which can be time-consuming for large datasets.
2. **No Backward Traversal**: Singly linked lists do not allow backward traversal, which can limit certain operations, such as finding the previous customer record.

In summary, the singly linked list is suitable for customer records management due to its dynamic nature and efficient memory usage, despite its limitations in search and traversal capabilities.

Task 4: Application of Doubly Linked List

Benefits of Doubly Linked List for Inventory Management

A doubly linked list is particularly beneficial for inventory management due to its ability to traverse in both directions, which enhances the efficiency of various operations.

Advantages:

1. **Bidirectional Traversal**: The ability to traverse the list forwards and backwards allows for more flexible operations, such as quickly accessing the previous or next inventory item.
2. **Easier Deletion**: Removing an item from the list is more efficient since each node has pointers to both its previous and next nodes, eliminating the need to traverse the list to find the predecessor.
3. **Enhanced Updates**: Updating inventory items can be done more efficiently, as the structure allows easy access to neighboring nodes.

Scenarios for Traversing:

- **Inventory Audits**: When conducting audits, being able to traverse both forwards and backwards can help quickly verify item counts and details.
- **Sorting Operations**: If the inventory needs to be sorted or rearranged, the ability to move in both directions simplifies the process.

In conclusion, the doubly linked list is advantageous for inventory management due to its bidirectional traversal capabilities, making it easier to manage and update inventory items.

Task 6: Application of Circular Linked List

Suitability of Circular Linked List for Transaction History Management

A circular linked list is an excellent choice for managing transaction history due to its unique structure that allows for efficient access to the most recent transactions.

Advantages:

1. **Continuous Traversal**: The circular nature allows for continuous traversal of transaction records without needing to reset to the head, making it easy to cycle through recent transactions.
2. **Quick Access to Recent Transactions**: The last transaction can be accessed directly from the last node, which is beneficial for applications that frequently need to display the most recent transaction.
3. **Memory Efficiency**: Like singly linked lists, circular linked lists do not require additional memory for pointers to the end of the list, as the last node points back to the first node.

Benefits Over Other Structures:

- **Singly Linked List**: Unlike singly linked lists, circular linked lists do not require a traversal from the head to access the last transaction, enhancing performance for recent transaction retrieval.

- **Doubly Linked List**: While doubly linked lists allow for bidirectional traversal, the circular linked list's structure is more suited for applications where the most recent data is frequently accessed and cycled through.

In summary, the circular linked list is well-suited for managing transaction history due to its efficient traversal capabilities and quick access to recent transactions, making it a superior choice for this specific application.