The University of Azad Jammu and Kashmir, Muzaffarabad

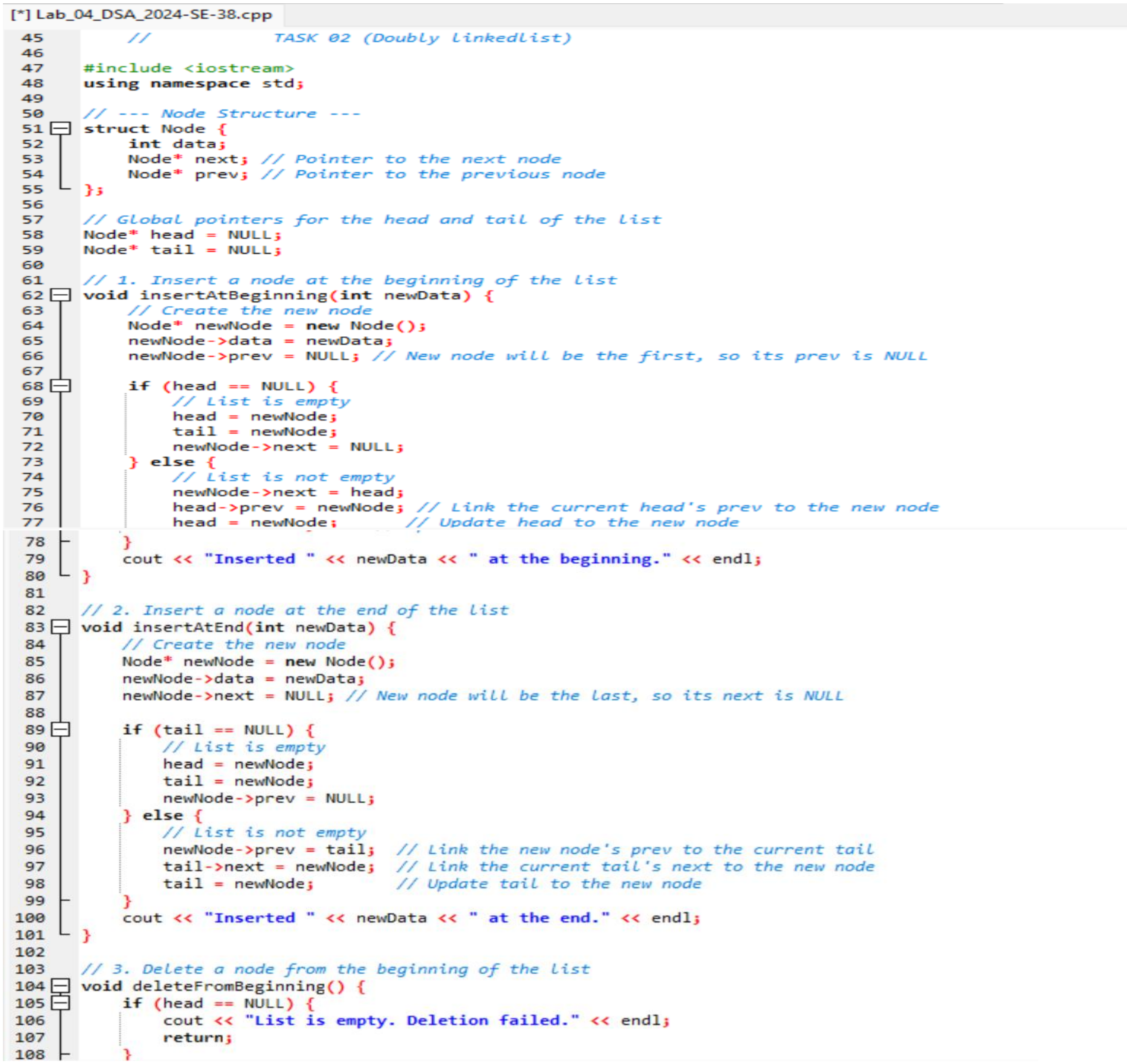
Name	Kamal Ali Akmal
Course Name	Data Structure & Algorithm
Submitted to	Engr. Sidra Rafique
Semester	3rd
Session	2024-2028
Roll No	2024-SE-38
Lab No	04
Submission date	04 November 2025

Department of Software Engineering

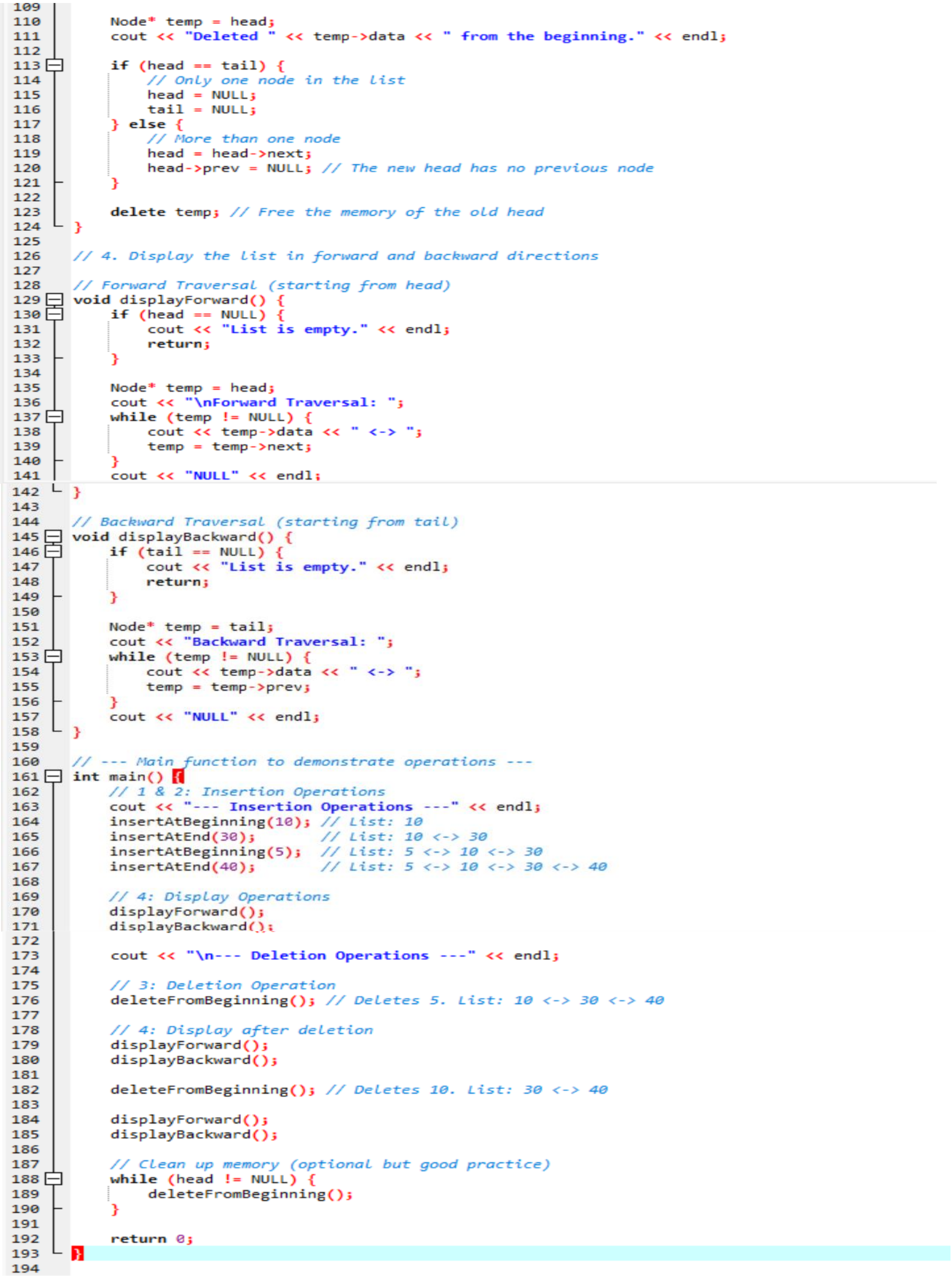
Doubly Linked List

- Each node points to both the next and previous nodes.
- A Doubly Linked List is like a singly linked list, but each node has an additional pointer, typically called “prev”, which points to the **previous** node in the sequence. This allows for efficient traversal in both forward and backward directions.

Code

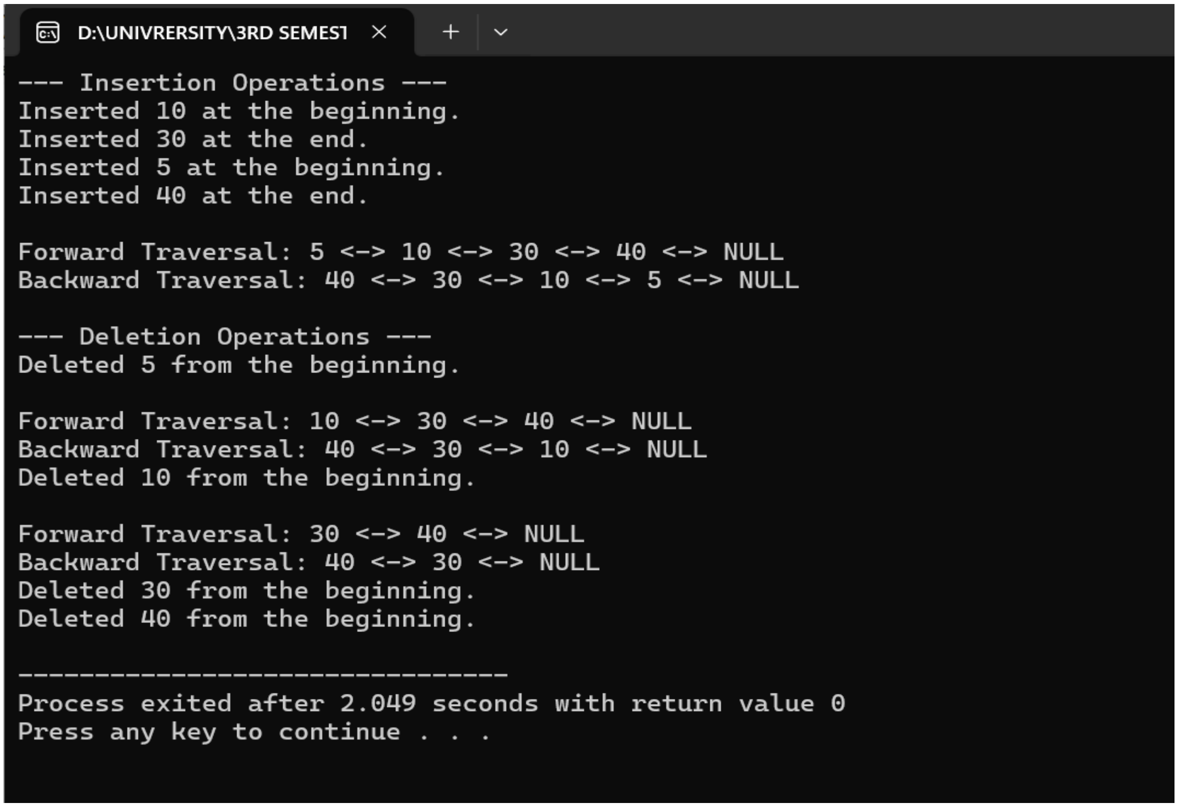


```cpp
    //            TASK 02 (Doubly Linkedlist)

#include <iostream>
using namespace std;

// --- Node Structure ---
struct Node {
    int data;
    Node* next; // Pointer to the next node
    Node* prev; // Pointer to the previous node
};

// Global pointers for the head and tail of the list
Node* head = NULL;
Node* tail = NULL;

// 1. Insert a node at the beginning of the list
void insertAtBeginning(int newData) {
    // Create the new node
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->prev = NULL; // New node will be the first, so its prev is NULL

    if (head == NULL) {
        // List is empty
        head = newNode;
        tail = newNode;
        newNode->next = NULL;
    } else {
        // List is not empty
        newNode->next = head;
        head->prev = newNode; // Link the current head's prev to the new node
        head = newNode;       // Update head to the new node
    }
    cout << "Inserted " << newData << " at the beginning." << endl;
}

// 2. Insert a node at the end of the list
void insertAtEnd(int newData) {
    // Create the new node
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = NULL; // New node will be the last, so its next is NULL

    if (tail == NULL) {
        // List is empty
        head = newNode;
        tail = newNode;
        newNode->prev = NULL;
    } else {
        // List is not empty
        newNode->prev = tail;  // Link the new node's prev to the current tail
        tail->next = newNode;  // Link the current tail's next to the new node
        tail = newNode;        // Update tail to the new node
    }
    cout << "Inserted " << newData << " at the end." << endl;
}

// 3. Delete a node from the beginning of the list
void deleteFromBeginning() {
    if (head == NULL) {
        cout << "List is empty. Deletion failed." << endl;
        return;
    }
```



```
    Node* temp = head;
    cout << "Deleted " << temp->data << " from the beginning." << endl;

    if (head == tail) {
        // Only one node in the list
        head = NULL;
        tail = NULL;
    } else {
        // More than one node
        head = head->next;
        head->prev = NULL; // The new head has no previous node
    }

    delete temp; // Free the memory of the old head
}

// 4. Display the list in forward and backward directions

// Forward Traversal (starting from head)
void displayForward() {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "\nForward Traversal: ";
    while (temp != NULL) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}

// Backward Traversal (starting from tail)
void displayBackward() {
    if (tail == NULL) {
        cout << "List is empty." << endl;
        return;
    }

    Node* temp = tail;
    cout << "Backward Traversal: ";
    while (temp != NULL) {
        cout << temp->data << " <-> ";
        temp = temp->prev;
    }
    cout << "NULL" << endl;
}

// --- Main function to demonstrate operations ---
int main() {
    // 1 & 2: Insertion Operations
    cout << "--- Insertion Operations ---" << endl;
    insertAtBeginning(10); // List: 10
    insertAtEnd(30);       // List: 10 <-> 30
    insertAtBeginning(5);  // List: 5 <-> 10 <-> 30
    insertAtEnd(40);       // List: 5 <-> 10 <-> 30 <-> 40

    // 4: Display Operations
    displayForward();
    displayBackward();

    cout << "\n--- Deletion Operations ---" << endl;

    // 3: Deletion Operation
    deleteFromBeginning(); // Deletes 5. List: 10 <-> 30 <-> 40

    // 4: Display after deletion
    displayForward();
    displayBackward();

    deleteFromBeginning(); // Deletes 10. List: 30 <-> 40

    displayForward();
    displayBackward();

    // Clean up memory (optional but good practice)
    while (head != NULL) {
        deleteFromBeginning();
    }

    return 0;
}
```

Output



```
--- Insertion Operations ---
Inserted 10 at the beginning.
Inserted 30 at the end.
Inserted 5 at the beginning.
Inserted 40 at the end.

Forward Traversal: 5 <-> 10 <-> 30 <-> 40 <-> NULL
Backward Traversal: 40 <-> 30 <-> 10 <-> 5 <-> NULL

--- Deletion Operations ---
Deleted 5 from the beginning.

Forward Traversal: 10 <-> 30 <-> 40 <-> NULL
Backward Traversal: 40 <-> 30 <-> 10 <-> NULL
Deleted 10 from the beginning.

Forward Traversal: 30 <-> 40 <-> NULL
Backward Traversal: 40 <-> 30 <-> NULL
Deleted 30 from the beginning.
Deleted 40 from the beginning.

--------------------------------
Process exited after 2.049 seconds with return value 0
Press any key to continue . . .
```

Key Concepts

- **Node Structure:** Each Node contains data, a pointer to the **next** node, and a pointer to the **previous** node (prev).
- **Head and Tail:** We use two global pointers, **head** and **tail**, to keep track of the first and last nodes, respectively.
- **Insertion Logic:** Insertion requires updating **two** pointers (the next and prev) for the new node and potentially for the adjacent existing nodes. For example, when inserting at the beginning, you link the new node's next to the old head and the old head's prev back to the new node.
- **Deletion Logic:** Deletion also involves updating adjacent nodes' pointers to bypass the deleted node and freeing the memory.