Hands-on Activity 3.1	
Linked List	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** Aug.14,2025
Section: CPE21S4	**Date Submitted:** Aug.14,2025
Name(s): Bautista.Mariela S.	**Instructor: ENGR.**Quejado

6. Output

ILO A: Construct C++ code for a singly and doubly linked list in C++

A.1. Singly Linked List

main.cpp

```cpp
#include <iostream>
using namespace std;

class Node {
public:
    char data;
    Node* next;
    Node(char val) : data(val), next(nullptr) {}
};

// Traversal
void traverseList(Node* n) {
    while (n != nullptr) {
        cout << n->data;
        n = n->next;
    }
    cout << "\n";
}

// Insert at head
void insertAtHead(Node*& head, char new_data) {
    Node* new_node = new Node(new_data);
    new_node->next = head;
    head = new_node;
}

// Insert after given prev_node
void insertAfter(Node* prev_node, char new_data) {
    if (prev_node == nullptr) {
        cout << "Previous node cannot be null.\n";
        return;
    }
    Node* new_node = new Node(new_data);
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}

// Insert at end
void insertAtEnd(Node*& head, char new_data) {
    Node* new_node = new Node(new_data);
    if (head == nullptr) {
        head = new_node;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) temp = temp->next;
    temp->next = new_node;
}

// Delete by key
void deleteNode(Node*& head, char key) {
    if (head == nullptr) return;

    if (head->data == key) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* prev = nullptr;
    Node* curr = head;
    while (curr != nullptr && curr->data != key) {
        prev = curr;
        curr = curr->next;
    }

    if (curr == nullptr) return; // key not found
    prev->next = curr->next;
    delete curr;
}

int main() {
    // Step 1: Create initial list C -> P -> E -> 0 -> 1 -> 0
    Node* head = new Node('C');
```

```cpp
int main() {
    // Step 1: Create initial list C -> P -> E -> 0 -> 1 -> 0
    Node* head = new Node('C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    cout << "Initial list: ";
    traverseList(head); // a.

    // b. Insert G at start
    insertAtHead(head, 'G');
    cout << "After inserting G at start: ";
    traverseList(head);

    // c. Insert E after P
    Node* temp = head;
    while (temp != nullptr && temp->data != 'P') temp = temp->next;
    insertAfter(temp, 'E');
    cout << "After inserting E after P: ";
    traverseList(head);

    // d. Delete node containing C
    deleteNode(head, 'C');
    cout << "After deleting C: ";
    traverseList(head);

    // e. Delete node containing P
    deleteNode(head, 'P');
    cout << "After deleting P: ";
    traverseList(head);

    // f. Final output
    cout << "Final list: ";
    traverseList(head);

    return 0;
}
```

Output :

```
Initial list: CPE010
After inserting G at start: GCPE010
After inserting E after P: GCPEE010
After deleting C: GPEE010
After deleting P: GEE010
Final list: GEE010


...Program finished with exit code 0
Press ENTER to exit console.
```

A.2. Doubly Linked List

main.cpp

```cpp
#include <iostream>
using namespace std;

class DNode {
public:
    char data;
    DNode* next;
    DNode* prev;
    DNode(char val) : data(val), next(nullptr), prev(nullptr) {}
};

void traverseList(DNode* n) {
    while (n != nullptr) {
        cout << n->data;
        n = n->next;
    }
    cout << "\n";
}


void insertAtHead(DNode*& head, char new_data) {
    DNode* new_node = new DNode(new_data);
    new_node->next = head;
    if (head != nullptr) head->prev = new_node;
    head = new_node;
}


void insertAfter(DNode* prev_node, char new_data) {
    if (prev_node == nullptr) {
        cout << "Previous node cannot be null.\n";
        return;
    }
    DNode* new_node = new DNode(new_data);
    new_node->next = prev_node->next;
    new_node->prev = prev_node;
    if (prev_node->next != nullptr)
        prev_node->next->prev = new_node;
    prev_node->next = new_node;
}

void insertAtEnd(DNode*& head, char new_data) {
    DNode* new_node = new DNode(new_data);
    if (head == nullptr) {
        head = new_node;
        return;
    }
    DNode* temp = head;
    while (temp->next != nullptr) temp = temp->next;
    temp->next = new_node;
    new_node->prev = temp;
}


void deleteNode(DNode*& head, char key) {
    if (head == nullptr) return;

    DNode* curr = head;
    while (curr != nullptr && curr->data != key) curr = curr->next;

    if (curr == nullptr) return;

    if (curr->prev != nullptr) curr->prev->next = curr->next;
    else head = curr->next;
    if (curr->next != nullptr) curr->next->prev = curr->prev;

    delete curr;
}

int main() {
    DNode* head = new DNode('C');
    insertAtEnd(head, 'P');
    insertAtEnd(head, 'E');
    insertAtEnd(head, '0');
    insertAtEnd(head, '1');
    insertAtEnd(head, '0');

    cout << "Initial Doubly List: ";
    traverseList(head);

    insertAtHead(head, 'G');
    cout << "After inserting G at start: ";
    traverseList(head);

    DNode* temp = head;
    while (temp != nullptr && temp->data != 'P') temp = temp->next;
    insertAfter(temp, 'E');
```

```
    insertAfter(temp, 'E');
    cout << "After inserting E after P: ";
    traverseList(head);

    deleteNode(head, 'C');
    cout << "After deleting C: ";
    traverseList(head);

    deleteNode(head, 'P');
    cout << "After deleting P: ";
    traverseList(head);

    cout << "Final Doubly List: ";
    traverseList(head);

    return 0;
}
```

Output:

```
Input
Initial Doubly List: CPE010
After inserting G at start: GCPE010
After inserting E after P: GCPEE010
After deleting C: GPEE010
After deleting P: GEE010
Final Doubly List: GEE010


...Program finished with exit code 0
Press ENTER to exit console.
```

Table 3-2. Code for the List Operations

Operation	Screenshot

Traversal	1. Traversal
	Procedure TRAVERSE(head)
	IF head == NULL THEN
	PRINT "List is empty"
	RETURN
	ENDIF
	
	SET current = head
	WHILE current != NULL DO
	PRINT current.data
	current = current.next
	ENDWHILE
	END Procedure

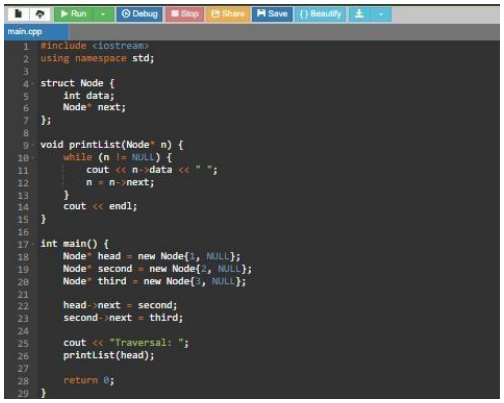
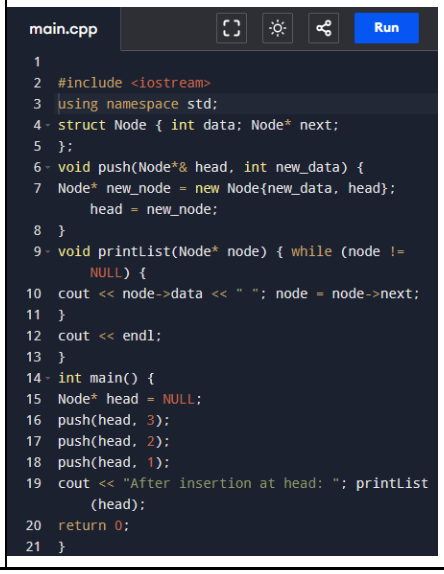
Insertion at head	2. Insertion at Head Procedure INSERT_HEAD(head_ref, newData) CREATE newNode newNode.data = newData newNode.next = head_ref head_ref = newNode END Procedure

Insertion at any part of the list	3. Insertion at Any Position
	Procedure INSERT_AT_POSITION(head_ref, position, newData)
	CREATE newNode
	newNode.data = newData
	
	IF position == 0 THEN
	CALL INSERT_HEAD(head_ref, newData)
	RETURN
	ENDIF
	
	SET current = head_ref
	FOR i = 0 TO position - 1 DO
	IF current == NULL THEN
	PRINT "Position out of range"
	RETURN
	ENDIF
	current = current.next
	ENDFOR
	
	newNode.next = current.next
	current.next = newNode
	END Procedure

Insertion at the end	4. Insertion at End
	Procedure INSERT_END(head_ref, newData)
	CREATE newNode
	newNode.data = newData
	newNode.next = NULL
	
	IF head_ref == NULL THEN
	head_ref = newNode
	RETURN
	ENDIF
	
	SET current = head_ref
	WHILE current.next != NULL DO
	current = current.next
	ENDWHILE
	
	current.next = newNode
	END Procedure

Deletion of a node	5. Deletion of a Node
	Procedure DELETE_NODE(head_ref, key)
	SET temp = head_ref
	SET prev = NULL
	
	IF temp != NULL AND temp.data == key THEN
	head_ref = temp.next
	DELETE temp
	RETURN
	ENDIF
	
	WHILE temp != NULL AND temp.data != key DO
	prev = temp
	temp = temp.next
	ENDWHILE
	
	IF temp == NULL THEN
	PRINT "Key not found"
	RETURN
	ENDIF
	
	prev.next = temp.next
	DELETE temp
	END Procedure

Table 3-3. Code and Analysis for Singly Linked Lists

a. Source Code **Console**	**GeeksforGeeks**  
b. Source Code **Console**	**GeeksforGeeks** 

```
After insertion at head: 1 2 3


=== Code Execution Successful ===
```

C. Source Code

Console

GeeksforGeeks

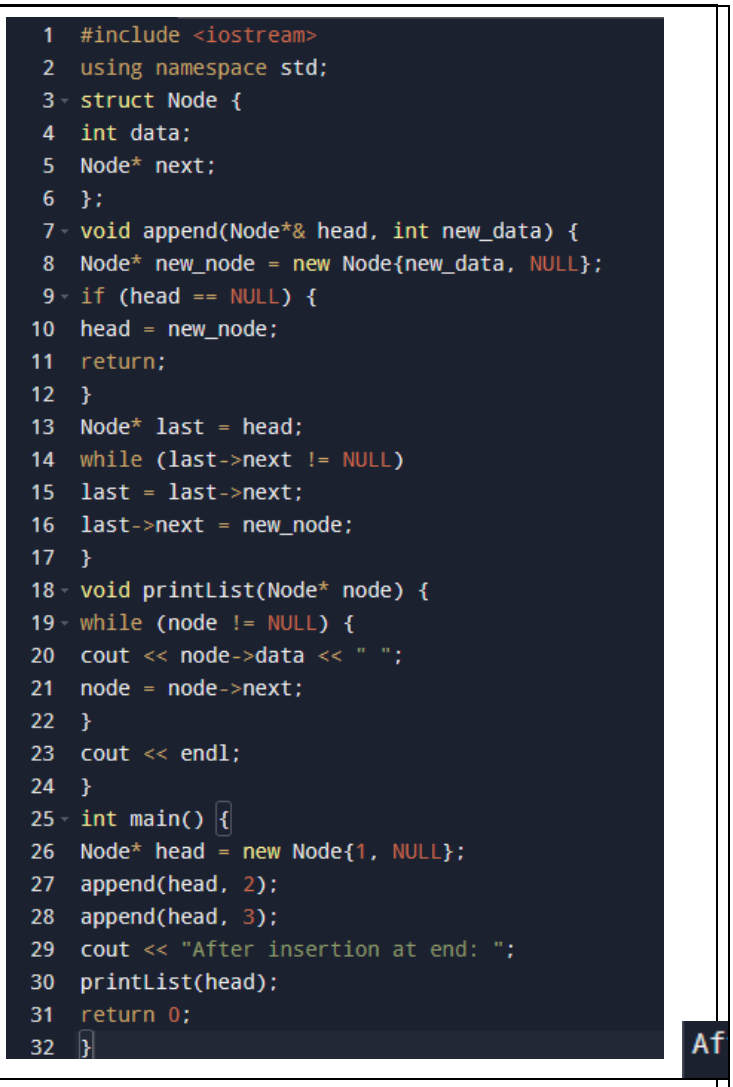
```cpp
#include <iostream>
using namespace std;
struct Node {
int data;
Node* next;
};
void insertAfter(Node* prev_node, int new_data
    ) {
if (prev_node == NULL) {
cout << "Previous node cannot be NULL\n";
return;
}
Node* new_node = new Node{new_data, prev_node
    ->next};
prev_node->next = new_node;
}
void printList(Node* node) {
while (node != NULL) {
cout << node->data << " ";
node = node->next;
}
cout << endl;
}
int main() {
Node* head = new Node{1, NULL};
Node* second = new Node{2, NULL};
Node* third = new Node{4, NULL};
head->next = second;
second->next = third;
insertAfter(second, 3);
cout << "After insertion at position: ";
printList(head);
return 0;
}
```

```
After insertion at position: 1 2 3 4
```

D. Source Code

Console

GeeksforGeeks

	
E. Source Code **Console**	**GeeksforGeeks**

```cpp
#include <iostream>
using namespace std;
struct Node {
int data;
Node* next;
};
void deleteNode(Node*& head, int key) {
Node* temp = head;
Node* prev = NULL;
if (temp != NULL && temp->data == key) {
head = temp->next;
delete temp;
return;
}
while (temp != NULL && temp->data != key) {
prev = temp;
temp = temp->next;
}
if (temp == NULL) return;
prev->next = temp->next;
delete temp;
}
void printList(Node* node) {
while (node != NULL) {
cout << node->data << " ";
node = node->next;
}
cout << endl;
}
int main() {
Node* head = new Node{1, NULL};
Node* second = new Node{2, NULL};
Node* third = new Node{3, NULL};
head->next = second;
second->next = third;
cout << "Before deletion: ";
printList(head);
deleteNode(head, 2);
cout << "After deletion: ";
printList(head);
```

```
Before deletion: 1 2 3
After deletion: 1 3
```

F. Source Code
 	
Console
	GeeksforGeeks
	
	`#include <iostream>`
	`using namespace std;`
	`struct Node {`
	`int data;`
	`Node* next;`
	`};`
	`bool search(Node* head, int key) {`
	`Node* current = head;`
	`while (current != NULL) {`
	`if (current->data == key)`
	`return true;`
	`current = current->next;`
	`}`
	`return false;`
	`}`
	`int main() {`
	`Node* head = new Node{1, NULL};`
	`Node* second = new Node{2, NULL};`
	`Node* third = new Node{3, NULL};`
	`head->next = second;`
	`second->next = third;`
	`int key = 2;`
	`if (search(head, key))`
	`cout << key << " is found in the list." << endl;`
	`else`
	`cout << key << " is not found in the list." << endl;`
	`return 0;`
	`}`
	
	`2 is found in the`

Table 3-4. Modified Operations for Doubly Linked Lists

Screenshots(s)	Analysis
Traversal	**Description:** Visits each node starting from the head, moving forward through the next pointer until the end, optionally also traversing backward using prev. **Time Complexity:** O(n) because we visit every node exactly once. **Space Complexity:** O(1) since we don't use extra memory apart from a temporary pointer.
Insertion at the Beginning	**Description:** Creates a new node and sets its next pointer to the current head, then updates head's prev pointer to the new node. The head pointer is then moved to the new node. **Time Complexity:** O(1) — constant time because no traversal is needed. **Space Complexity:** O(1) — only an extra node is created.
Insertion at the End	**Description:** Traverses to the last node, sets its next pointer to the new node, and sets the new node's prev pointer to the last node. **Time Complexity:** O(n) — traversal needed to reach the tail. **Space Complexity:** O(1) — only an extra node is created.

Insertion after a given node	**Description:** Finds the target node, adjusts pointers so the new node sits right after it, linking both next and prev correctly. **Time Complexity:** O(n) — in worst case, we may traverse the entire list to find the target. **Space Complexity**: O(1).
Deletion of a Node	**Description**: Adjusts the next pointer of the previous node and prev pointer of the next node to bypass the node to be deleted, then frees the memory. **Time Complexity:** O(n) — in worst case, we search the entire list for the node to delete. **Space Complexity:** O(1) — no extra memory is used apart from a pointer
Reverse Traversal	**Description:** Starts from the tail and moves backward using prev pointers, printing or processing each node. **Time Complexity:** O(n) — every node is visited exactly once. **Space Complexity:** O(1).

ILO B: Solve given problems utilizing linked lists in C++ CODE:

main.cpp

```cpp
#include <iostream>
#include <string>
using namespace std;


struct SongNode {
    string title;
    SongNode* next;
    SongNode(const string& t) : title(t), next(nullptr) {}
};

class Playlist {
    SongNode* head;
public:
    Playlist() : head(nullptr) {}


    void insertSong(const string& title) {
        SongNode* node = new SongNode(title);
        if (!head) {
            node->next = node;
            head = node;
            return;
        }
        SongNode* temp = head;
        while (temp->next != head) temp = temp->next;
        temp->next = node;
        node->next = head;
    }


    void removeSong(const string& title) {
        if (!head) return;


        if (head->title == title && head->next == head) {
            delete head;
            head = nullptr;
            return;
        }

        SongNode* curr = head;
        SongNode* prev = nullptr;
        do {
            if (curr->title == title) {
                if (prev) prev->next = curr->next;
                if (curr == head) {

                    SongNode* tail = head;
                    while (tail->next != head) tail = tail->next;
                    head = head->next;
                    tail->next = head;
                }
                delete curr;
                return;
            }
            prev = curr;
            curr = curr->next;
        } while (curr != head);
    }


    void playAll() const {
        if (!head) {
            cout << "Playlist is empty.\n";
            return;
        }
        const SongNode* temp = head;
        do {
            cout << "Playing: " << temp->title << "\n";
            temp = temp->next;
        } while (temp != head);
    }

    void next() {
        if (head) head = head->next;
    }

    void previous() {
        if (!head) return;
        SongNode* temp = head;
        while (temp->next != head) temp = temp->next;
        head = temp;
    }

    void showCurrent() const {
        if (head) cout << "Current: " << head->title << "\n";
```

```
        if (head) cout << "Current: " << head->title << "\n";
        else cout << "Playlist is empty.\n";
    }
};

int main() {
    Playlist pl;


    pl.insertSong("Big Girls Don't Cry - Fergie");
    pl.insertSong("Lady of My Life - Michael Jackson");
    pl.insertSong("Sweet - Lana Del Rey");

    cout << "============== My Playlist ==============\n";
    pl.playAll();

    cout << "\n============== Next Song ==============\n";
    pl.next();
    pl.showCurrent();

    cout << "\n============== Previous Song ==============\n";
    pl.previous();
    pl.showCurrent();

    return 0;
}
```

Output:

8. Conclusion

Summary of Lessons Learned:

In this activity, I learned how Doubly Linked Lists (DLLs) work and how they're different from Singly Linked Lists. DLLs let you move through the list in both directions because each node has two pointers: one for the next node and one for the previous. I also learned how to add and remove items from the list by carefully updating these pointers. Lastly, I practiced checking how fast and memory-efficient these operations are to make sure everything works well.

Analysis of the Procedure:

The procedure involved coding various DLL operations:

Traversal – Visiting all elements forward and backward. Insertion –

At the head, at the end, or after a specific node.

Deletion – Removing a node while maintaining proper links between neighbors.

Every step needed careful pointer handling to keep the list working properly and avoid memory issues. The code showed that adding at the beginning is fast, but searching or moving through the list takes more time. I used console outputs to check that everything worked as expected.

Analysis of the Supplementary Activity:
This extra activity helped me understand things better by making me both code and think about how it works. I learned why some actions are fast (O(1)) and others take longer (O(n), like searching), and how working with pointers is different from using arrays. It also showed why handling special cases—like adding to an empty list or removing the first or last item—is important. Overall, it connected the theory with real coding practice.

Concluding Statement / Feedback:
I think I did well in this activity because I was able to write and test all the Doubly Linked List operations correctly. My code worked as expected, and my analysis matched the theory. Still, I know I can get better at fixing bugs faster and writing stronger code for tricky situations, like deleting several items in a row. With more practice, I can make my code cleaner, better organized, and more reliable.