Activity No. 3	
Hands-on Activity 3.1 Linked Lists	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** Sept. 27, 2024
Section: CPE21S4	**Date Submitted:** Sept. 29, 2024
Name(s): Edson Ray San Juan	**Instructor:** Prof. Maria Rizzette Sayo

6. Output

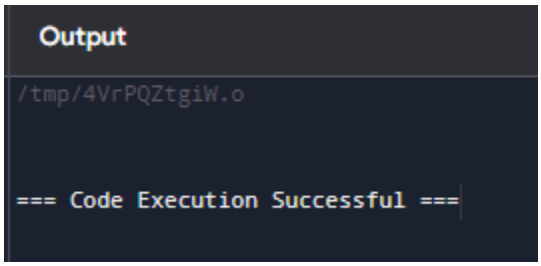
Screenshot

Code:

main.cpp

```
#include<iostream>
#include<utility>

class Node{
public:
char data;
Node *next;
};

int main(){
//step 1
Node *head = NULL;
Node *second = NULL;
Node *third = NULL;
Node *fourth = NULL;
Node *fifth = NULL;
Node *last = NULL;

//step 2
head = new Node;
second = new Node;
third = new Node;
fourth = new Node;
fifth = new Node;
last = new Node;

//step 3
head->data = 'C';
head->next = second;

second->data = 'P';
second->next = third;

third->data = 'E';
third->next = fourth;

fourth->data = '0';
fourth->next = fifth;

fifth->data = '1';
fifth->next = last;

//step 4
last->data = '0';
last->next = nullptr;
}
```

Output:

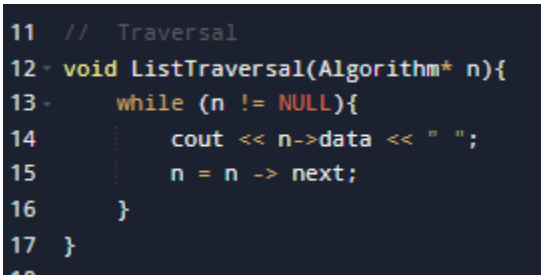
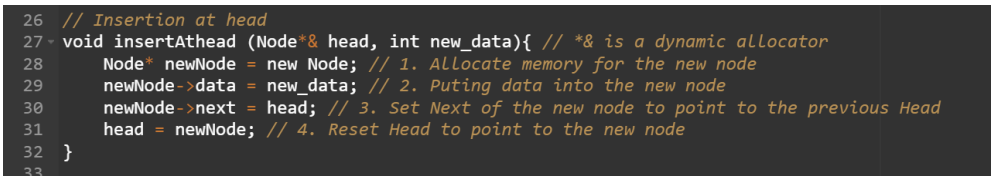
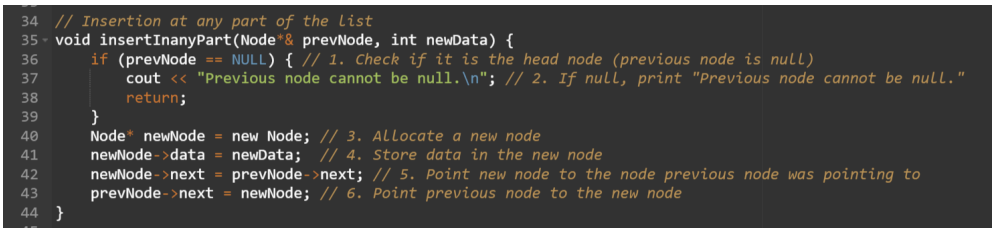
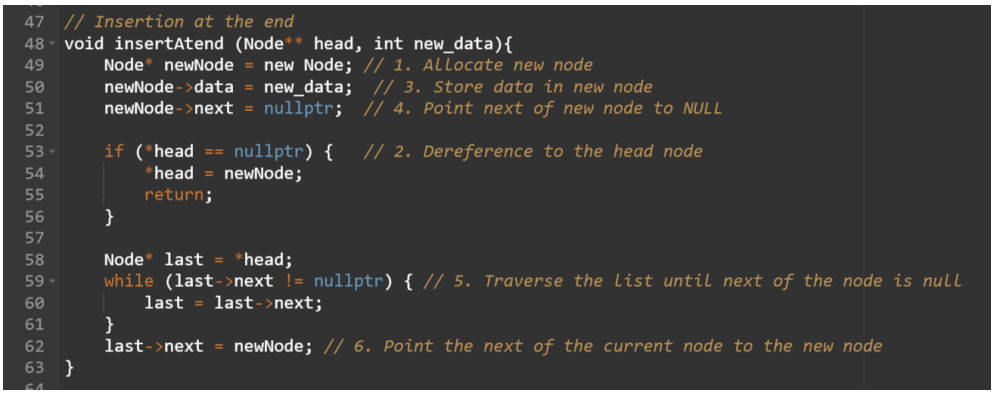


Discussion

The code defines a simple singly linked list using a Node class, where each node contains a character and a pointer to the next node. In the main() function, six nodes are created and linked together, with the characters 'C', 'P', 'E', '0', '1', and '0' assigned to them sequentially. The last node's next pointer is set to nullptr, indicating the end of the list. Although the code executes successfully, it lacks functionality to display or manipulate the list, highlighting the need for

	additional methods to traverse and print the nodes.

Table 3-1. Output of Initial/Simple Implementation

Operation	Screenshot
Traversal	
Insertion at head	
Insertion at any part of the list	
Insertion at the end	

```
11  //  Traversal
12  void ListTraversal(Algorithm* n){
13      while (n != NULL){
14          cout << n->data << " ";
15          n = n -> next;
16      }
17  }
```

```
26  // Insertion at head
27  void insertAthead (Node*& head, int new_data){ // *& is a dynamic allocator
28      Node* newNode = new Node; // 1. Allocate memory for the new node
29      newNode->data = new_data; // 2. Puting data into the new node
30      newNode->next = head; // 3. Set Next of the new node to point to the previous Head
31      head = newNode; // 4. Reset Head to point to the new node
32  }
```

```
34  // Insertion at any part of the list
35  void insertInanyPart(Node*& prevNode, int newData) {
36      if (prevNode == NULL) { // 1. Check if it is the head node (previous node is null)
37          cout << "Previous node cannot be null.\n"; // 2. If null, print "Previous node cannot be null."
38          return;
39      }
40      Node* newNode = new Node; // 3. Allocate a new node
41      newNode->data = newData;  // 4. Store data in the new node
42      newNode->next = prevNode->next; // 5. Point new node to the node previous node was pointing to
43      prevNode->next = newNode; // 6. Point previous node to the new node
44  }
```

```
47  // Insertion at the end
48  void insertAtend (Node** head, int new_data){
49      Node* newNode = new Node; // 1. Allocate new node
50      newNode->data = new_data;  // 3. Store data in new node
51      newNode->next = nullptr;  // 4. Point next of new node to NULL
52
53      if (*head == nullptr) {   // 2. Dereference to the head node
54          *head = newNode;
55          return;
56      }
57
58      Node* last = *head;
59      while (last->next != nullptr) { // 5. Traverse the list until next of the node is null
60          last = last->next;
61      }
62      last->next = newNode; // 6. Point the next of the current node to the new node
63  }
```

Deletion of a node	

```
// Deletion of a node
void deleteNode(Node** head, int key) {
    Node* temp = *head;
    Node* prev = nullptr;

    if (temp != nullptr && temp->data == key) { // 1. Find previous node of the node to be deleted.
        *head = temp->next; // 2. Change the next of previous node.
        delete temp; // 3. Free memory for the node to be deleted.
        return;
    }

    while (temp != nullptr && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == nullptr)
    return;

    prev->next = temp->next; // 2. Change the next of previous node.
    delete temp; // 3. Free memory for the node to be deleted.
}
```

Table 3-2. Code for the List Operations

a.	Source Code	

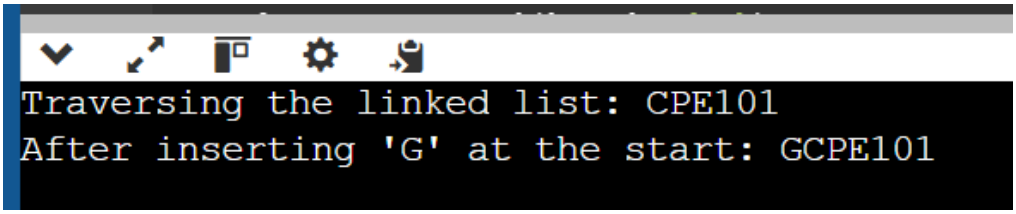
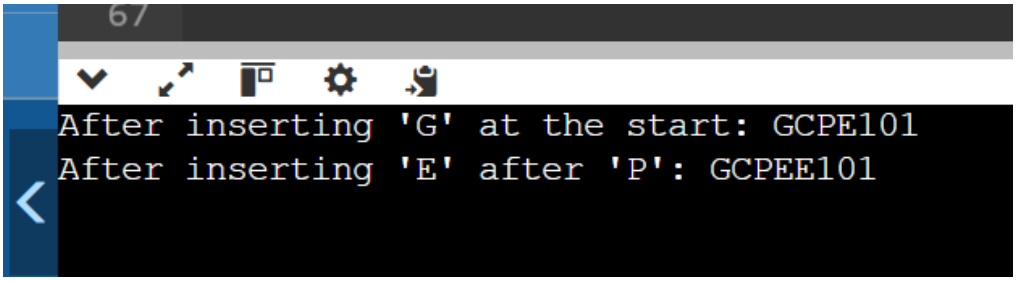
```
void traverseList(Node* head) {
    Node* current = head;
    while (current != NULL) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}
```

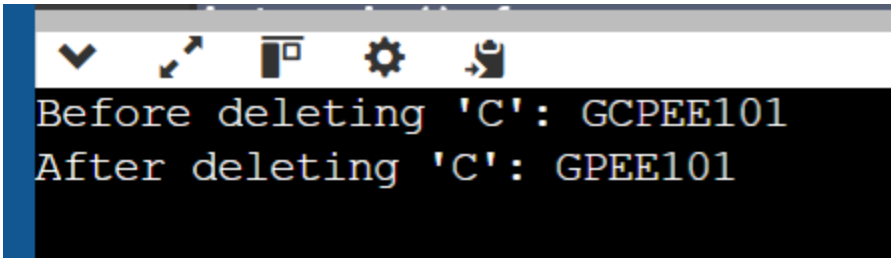
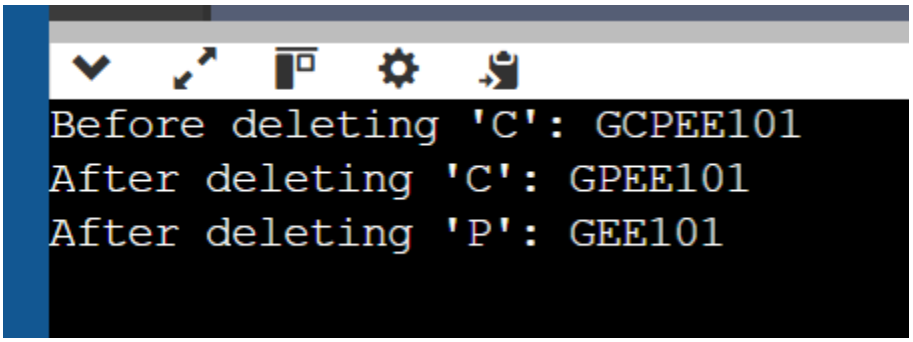
	Console	

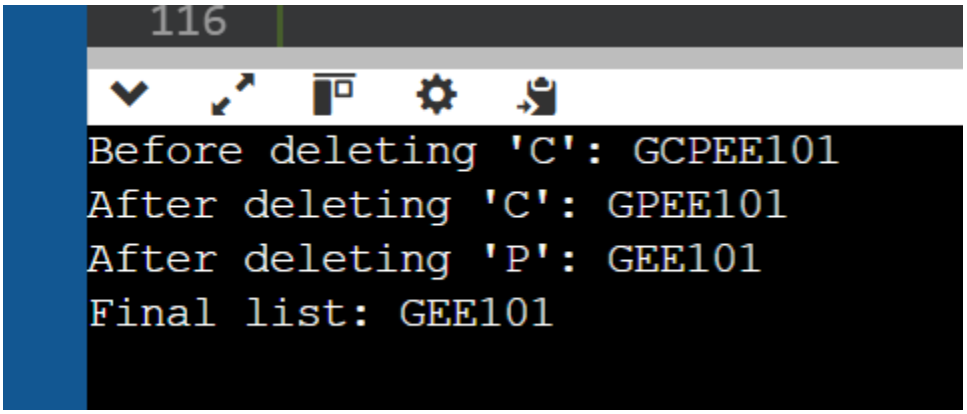
```
Traversing the linked list: C P E 0 1 0
```

b.	Source Code	

```
void insertAtHead(Node*& head, char new_data) {
    Node* newNode = new Node;
    newNode->data = new_data; // Assign the new data ('G')
    newNode->next = head;
    head = newNode;
}
```

	Console	
c.	Source Code	``` Node* current = head; while (current != NULL) { if (current->data == 'P') { insertAfter(current, 'E'); // Insert 'E' after 'P' break; } current = current->next; } ```
	Console	
d.	Source Code	``` // Delete the node containing the element C. void deleteNode(Node*& head, char value) { Node* current = head; Node* prev = NULL; while (current != NULL && current->data != value) { prev = current; current = current->next; } if (current != NULL) { if (prev == NULL) { head = current->next; } else { prev->next = current->next; } delete current; // Free the memory of the deleted node } else { cout << "Node with value '" << value << "' not found." << endl ; } } deleteNode(head, 'C'); // Deleting the node containing 'C' ```

	Console	
e.	Source Code	``` 56 // Delete the node containing the element specified by value 57 void deleteNode(Node*& head, char value) { 58 Node* current = head; 59 Node* prev = NULL; 60 61 while (current != NULL && current->data != value) { 62 prev = current; 63 current = current->next; 64 } 65 66 if (current != NULL) { 67 if (prev == NULL) { 68 head = current->next; 69 } else { 70 prev->next = current->next; 71 } 72 delete current; 73 } else { 74 cout << "Node with value '" << value << "' not found." << endl ; 75 } 76 } 106 // Deleting the node containing 'P' 107 deleteNode(head, 'P'); // Deleting the node containing 'P' ```
	Console	

f.	Source Code	98 99 cout << "Before deleting 'C': "; 100 traverseList(head); 101 102 deleteNode(head, 'C'); // Deleting the node containing 'C' 103 cout << "After deleting 'C': "; 104 traverseList(head); 105 106 deleteNode(head, 'P'); // Deleting the node containing 'P' 107 cout << "After deleting 'P': "; 108 traverseList(head); 109 110 // Show the elements in the list 111 cout << "Final list: "; 112 traverseList(head); // Displaying the final list 113
	Console	

Screenshot(s)	Analysis
4 class Node { 5 public: 6 char data; 7 Node* next; 8 Node* prev; 9 10 // Constructor to initialize the node 11 Node(char value) { 12 data = value; 13 next = nullptr; 14 prev = nullptr; 15 } 16 };	The node stores a char type data, a next pointer, and a prev pointer to enable both forward and backward traversal. This is important for implementing a doubly linked list, allowing efficient navigation through the list in both directions.

`` class DoublyLinkedList { private: Node* head; Node* tail; public: // Constructor to initialize the list DoublyLinkedList() { head = nullptr; tail = nullptr; } ```	The constructor initializes an empty list with head and tail pointers set to nullptr. meaning the list has no nodes when it's created. As new nodes are added, the head and tail pointers will be updated to point to the correct first and last nodes of the list.
`` // Function to insert a node at the end void insertAtEnd(char value) { Node* newNode = new Node(value); if (head == nullptr) { head = newNode; tail = newNode; } else { tail->next = newNode; newNode->prev = tail; tail = newNode; } } ```	The insertAtEnd method adds a new node to the end of the list. If the list is empty, the new node becomes both the head and tail. If the list already has nodes, the new node is added after the current tail, and then the tail pointer is updated to point to the new node. This method uses the next and prev pointers to link the nodes together properly.
`` // Function to display the list from the head to the tail void displayForward() { Node* temp = head; while (temp != nullptr) { cout << temp->data << " "; temp = temp->next; } cout << endl; } // Function to display the list from the tail to the head void displayBackward() { Node* temp = tail; while (temp != nullptr) { cout << temp->data << " "; temp = temp->prev; } cout << endl; } }; ```	The displayForward method goes through the list from the head and prints each node's data. The displayBackward method starts at the tail and moves backward using the prev pointer. Both methods show how you can move through the list in both directions.

```cpp
int main() {
    DoublyLinkedList list;

    // Insert nodes into the list
    list.insertAtEnd('4');
    list.insertAtEnd('5');
    list.insertAtEnd('6');
    list.insertAtEnd('7');

    // Display the list in forward and backward directions
    cout << "List in forward direction: ";
    list.displayForward();

    cout << "List in backward direction: ";
    list.displayBackward();

    return 0;
}
```

In the main function, nodes are added to the list, and the displayForward and displayBackward methods are used to print the list in both directions. The output confirms that the list works correctly by showing the expected order of elements going forward and backward.

Table 3-4. Modified Operations for Doubly Linked Lists

7. Supplementary Activity

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

Problem Title: Implementing a Song Playlist using Linked List
Source: Packt Publishing

Problem Description:
In this activity, we'll look at some applications for which a singly linked list is not enough or not convenient. We will build a tweaked version that fits the application. We often encounter cases where we have to customize default implementations, such as when looping songs in a music player or in games where multiple players take a turn one by one in a circle.

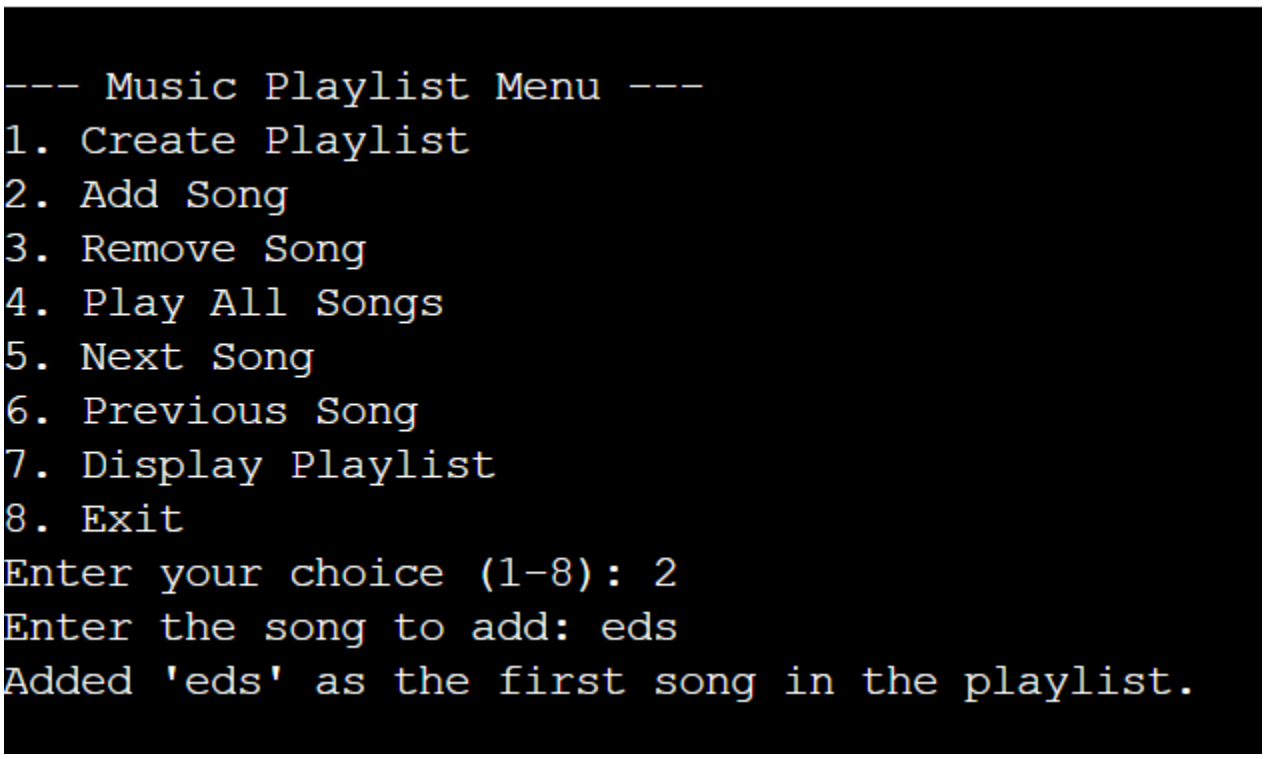
These applications have one common property – we traverse the elements of the sequence in a circular fashion. Thus, the node after the last node will be the first node while traversing the list. This is called a circular linked list.

We'll take the use case of a music player. It should have following functions supported:
• Create a playlist using multiple songs.
• Add songs to the playlist.
• Remove a song from the playlist.
• Play songs in a loop (for this activity, we will print all the songs once).

Here are the steps to solve the problem:
• Design the basic structure that supports circular data representation.
• After that, implement the insert and delete functions in the structure.
• Implement a function for traversing the playlist.

The driver function should allow for common operations on a playlist such as: next, previous, play all songs, insert and remove.

Output:



8. Conclusion

In our lab activity, I implemented key linked list operations like traversal, insertion, and deletion. I created a function to traverse the list and display its current state. I inserted 'G' at the beginning of the list and added 'E' after 'P', showing how to modify the list. I also deleted the nodes containing 'C' and 'P', which changed the list dynamically. After all the changes, the final result was "GEE101".

I also created a music playlist using a circular linked list in C++. It allows songs to play continuously, and supports adding, removing, and playing songs. The playlist has a simple console interface and can be expanded with features like shuffle play or saving songs.

This lab activity proved that our linked list could handle key operations. The modular design made the code easier to manage, and careful memory management avoided leaks. Overall, it strengthened our understanding of data structures and how to apply them in C++.

9. Assessment Rubric