Laboratory Activity # 3	
Introduction to Object-Oriented Programming	
Course Code: CPE009B	**Program:** BSCPE
Course Title: Object-Oriented Programming	**Date Performed:** 09-27-2024
Section: CPE21S4	**Date Submitted:** 09-27-2024
Name(s): Masangkay, Frederick D.	**Instructor:** Ma'am Maria Rizette Sayo

6. Output

Screenshot

```
// Online C++ compiler to run C++ program online
#include <iostream>

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 = 'O';
    fourth -> next = fifth;

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

    // step 4
    last -> data = 'O';
    last -> next = nullptr;

    return 0;
}
```

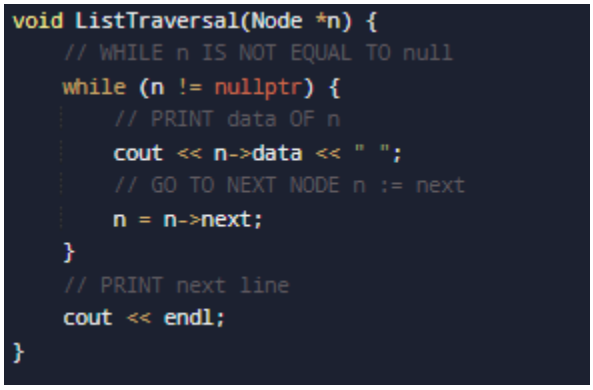
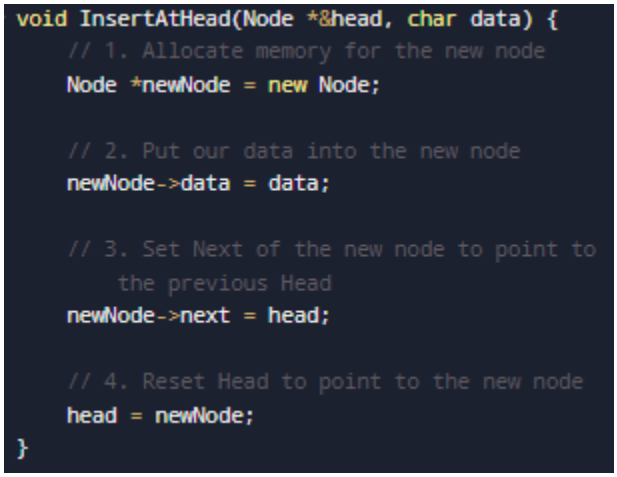
```
/tmp/g1qGjG76N7.o

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

Discussion

The provided code snippet demonstrates the creation of a linked list in C++ using a Node class. The code begins by declaring seven node pointers, followed by dynamically allocating memory for each node using the new keyword. The node data and next pointers are then initialized, with the data field assigned a character value and the next pointer set to point to the next node in the list. Finally, the next pointer of the last node is set to nullptr, indicating the end of the linked list. This code snippet provides a basic example of creating a linked list in C++, but could be improved through the use of more efficient memory allocation and error checking, as well as additional comments and descriptive variable names to enhance readability.

Table 3-1. Output of Initial/Simple Implementation

Operation	Screenshot
Traversal	
Insertion at head	

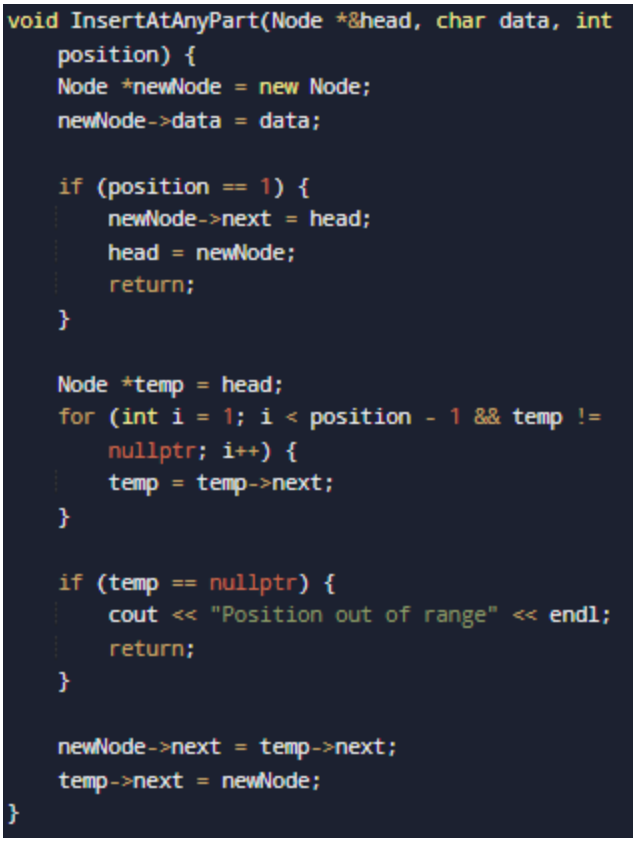
```
void ListTraversal(Node *n) {
    // WHILE n IS NOT EQUAL TO null
    while (n != nullptr) {
        // PRINT data OF n
        cout << n->data << " ";
        // GO TO NEXT NODE n := next
        n = n->next;
    }
    // PRINT next line
    cout << endl;
}
```

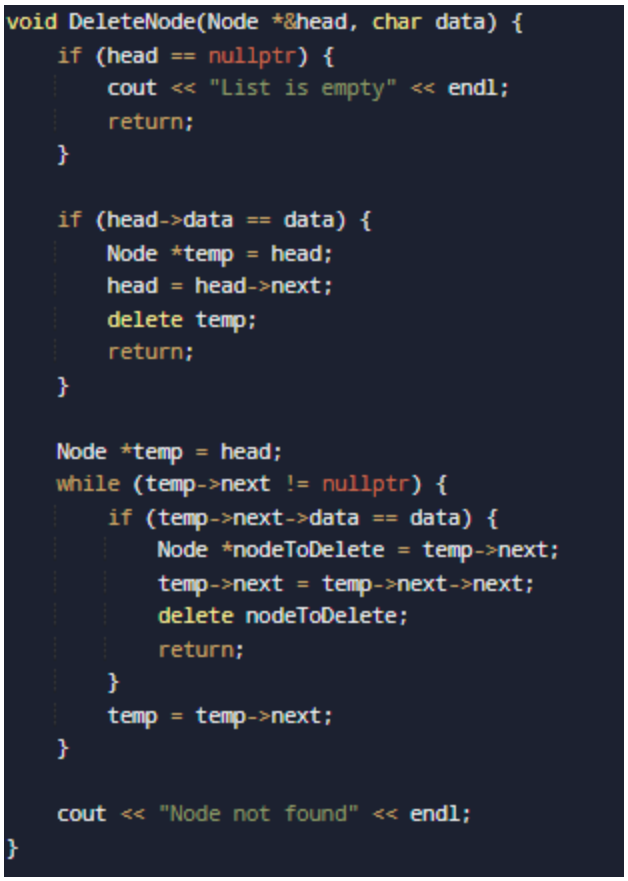
```
void InsertAtHead(Node *&head, char data) {
    // 1. Allocate memory for the new node
    Node *newNode = new Node;

    // 2. Put our data into the new node
    newNode->data = data;

    // 3. Set Next of the new node to point to
        the previous Head
    newNode->next = head;

    // 4. Reset Head to point to the new node
    head = newNode;
}
```

Insertion at any part of the list	
Insertion at the end	

Deletion of a node	

```
void DeleteNode(Node *&head, char data) {
    if (head == nullptr) {
        cout << "List is empty" << endl;
        return;
    }

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

    Node *temp = head;
    while (temp->next != nullptr) {
        if (temp->next->data == data) {
            Node *nodeToDelete = temp->next;
            temp->next = temp->next->next;
            delete nodeToDelete;
            return;
        }
        temp = temp->next;
    }

    cout << "Node not found" << endl;
}
```

Table 3-2. Code for the List Operation

a.	**Source Code**	#include <iostream>
		using namespace std;
		
		// Node class definition
		class Node {
		public:
		  char data;
		  Node *next;
		};
		// Traversal function
		void ListTraversal(Node* n) {
		  while (n != nullptr) {
		  cout << n->data << " "; // Print node data
		  n = n->next; // Move to the next node
		  }
		  cout << endl; // Print new line after traversal
		}

		int main() {
		  // Initialize an empty linked list
		  Node* head = nullptr;
		
		  // Step 1: Insert data into the list (C -> P -> E -> 0 -> 1 -> 0)
		  InsertAtEnd(&head, 'C');
		  InsertAtEnd(&head, 'P');
		  InsertAtEnd(&head, 'E');
		  InsertAtEnd(&head, '0');
		  InsertAtEnd(&head, '1');
		  InsertAtEnd(&head, '0');
		
		  // Task a: Traverse the list
		  cout << "Task a: Initial traversal of the list: ";
		  ListTraversal(head);
		
		  return 0;
		}

	Console	Task a: Initial traversal of the list: C P E 0 1 0

b.	**Source Code**	#include <iostream>
		using namespace std;
		
		// Node class definition
		class Node {
		public:
		  char data;
		  Node *next;
		};
		// Insertion at the head of the list
		void InsertAtHead(Node** head, char new_data) {
		  Node* new_node = new Node(); // Create a new node
		  new_node->data = new_data; // Set data
		  new_node->next = (*head); // Link the new node to the current head
		  (*head) = new_node; // Move head to point to the new node
		}
		
		int main() {
		  // Initialize an empty linked list
		  Node* head = nullptr;
		
		  // Step 1: Insert data into the list (C -> P -> E -> 0 -> 1 -> 0)

		InsertAtEnd(&head, 'C');
		InsertAtEnd(&head, 'P');
		InsertAtEnd(&head, 'E');
		InsertAtEnd(&head, '0');
		InsertAtEnd(&head, '1');
		InsertAtEnd(&head, '0');
		
		// Task b: Insert 'G' at the start of the list
		InsertAtHead(&head, 'G');
		cout << "Task b: Insert 'G' at the start: ";
		ListTraversal(head);
		
		return 0;
		}

	Console	Task b: Insert 'G' at the start: G C P E 0 1 0

c.	**Source Code**	#include <iostream>
		using namespace std;
		
		// Node class definition
		class Node {
		public:
		char data;
		Node *next;
		};
		// Insertion at any position in the list
		void InsertAtPosition(Node* prev_node, char new_data) {
		if (prev_node == nullptr) {
		cout << "Previous node cannot be null." << endl;
		return;
		}
		Node* new_node = new Node(); // Create a new node
		new_node->data = new_data; // Set data
		new_node->next = prev_node->next; // Link new node to the next of the previous node
		prev_node->next = new_node; // Link the previous node to the new node
		}
		int main() {
		// Initialize an empty linked list
		Node* head = nullptr;
		
		// Step 1: Insert data into the list (C -> P -> E -> 0 -> 1 -> 0)
		InsertAtEnd(&head, 'C');

		InsertAtEnd(&head, 'P');
		InsertAtEnd(&head, 'E');
		InsertAtEnd(&head, '0');
		InsertAtEnd(&head, '1');
		InsertAtEnd(&head, '0');
		
		// Task c: Insert 'E' after 'P'
		Node* temp = head;
		while (temp != nullptr && temp->data != 'P') // Find node with 'P'
		temp = temp->next;
		InsertAtPosition(temp, 'E'); // Insert 'E' after 'P'
		cout << "Task c: Insert 'E' after 'P': ";
		ListTraversal(head);
		
		return 0;
		}

	Console	Task c: Insert 'E' after 'P': G C P E E 0 1 0

d.	**Source Code**	#include <iostream>
		using namespace std;
		
		// Node class definition
		class Node {
		public:
		char data;
		Node *next;
		};
		// Deletion of a node by value
		void DeleteNode(Node** head, char key) {
		Node* temp = *head;
		Node* prev = nullptr;
		
		// If head node itself holds the key
		if (temp != nullptr && temp->data == key) {
		*head = temp->next; // Change head to the next node
		delete temp; // Free the old head
		return;
		}
		
		// Search for the key in the list
		while (temp != nullptr && temp->data != key) {
		prev = temp;

		temp = temp->next;
		}
		
		// If key is not present in the list
		if (temp == nullptr) {
		cout << "Key not found." << endl;
		return;
		}
		
		// Unlink the node from the list and free memory
		prev->next = temp->next;
		delete temp;
		}
		
		int main() {
		// Initialize an empty linked list
		Node* head = nullptr;
		
		// Step 1: Insert data into the list (C -> P -> E -> 0 -> 1 -> 0)
		InsertAtEnd(&head, 'C');
		InsertAtEnd(&head, 'P');
		InsertAtEnd(&head, 'E');
		InsertAtEnd(&head, '0');
		InsertAtEnd(&head, '1');
		InsertAtEnd(&head, '0');
		
		// Task d: Delete the node containing 'C'
		DeleteNode(&head, 'C');
		cout << "Task d: Delete node with 'C': ";
		ListTraversal(head);
		
		return 0;
		}

	Console	Task d: Delete node with 'C': G P E E 0 1 0

e.	**Source Code**	#include <iostream>
		using namespace std;
		
		// Node class definition
		class Node {
		public:
		char data;

```
    Node *next;
};
// Deletion of a node by value
void DeleteNode(Node** head, char key) {
    Node* temp = *head;
    Node* prev = nullptr;

    // If head node itself holds the key
    if (temp != nullptr && temp->data == key) {
        *head = temp->next; // Change head to the next node
        delete temp;        // Free the old head
        return;
    }

    // Search for the key in the list
    while (temp != nullptr && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If key is not present in the list
    if (temp == nullptr) {
        cout << "Key not found." << endl;
        return;
    }

    // Unlink the node from the list and free memory
    prev->next = temp->next;
    delete temp;
}

int main() {
    // Initialize an empty linked list
    Node* head = nullptr;

    // Step 1: Insert data into the list (C -> P -> E -> 0 -> 1 -> 0)
    InsertAtEnd(&head, 'C');
    InsertAtEnd(&head, 'P');
    InsertAtEnd(&head, 'E');
    InsertAtEnd(&head, '0');
    InsertAtEnd(&head, '1');
    InsertAtEnd(&head, '0');

    // Task e: Delete the node containing 'P'
    DeleteNode(&head, 'P');
```

		cout << "Task e: Delete node with 'P': "; ListTraversal(head); return 0; }
	Console	Task e: Delete node with 'P': G E E 0 1 0

f.	**Source Code**	#include <iostream>
		using namespace std;
		
		// Node class definition
		class Node {
		public:
		char data;
		Node *next;
		};
		// Traversal function
		void ListTraversal(Node* n) {
		while (n != nullptr) {
		cout << n->data << " "; // Print node data
		n = n->next; // Move to the next node
		}
		cout << endl; // Print new line after traversal
		}
		
		int main() {
		// Initialize an empty linked list
		Node* head = nullptr;
		
		// Step 1: Insert data into the list (C -> P -> E -> 0 -> 1 -> 0)
		InsertAtEnd(&head, 'C');
		InsertAtEnd(&head, 'P');
		InsertAtEnd(&head, 'E');
		InsertAtEnd(&head, '0');
		InsertAtEnd(&head, '1');
		InsertAtEnd(&head, '0');
		
		// Task f: Final traversal cout << "Task f: Final list traversal: ";
		ListTraversal(head);
		return 0;
		}

	Console	Task f: Final list traversal: G E E 0 1 0

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

Screenshot(s)	Analysis


// Traversal function (Forward Traversal)
void ListTraversalForward(Node* head)
{
Node* temp = head;
while (temp != nullptr)
{
cout << temp->data << " "; // Print node data
temp = temp->next; // Move to the next node
}
cout << endl; // New line after traversal
}
 	
// Traversal function (Backward Traversal)
void ListTraversalBackward(Node* tail)
{
Node* temp = tail;
while (temp != nullptr)
{
cout << temp->data << " "; // Print node data
temp = temp->prev; // Move to the previous node
}
cout << endl; // New line after traversal
}
	Traversal functions are expanded to support both forward and backward navigation. Forward traversal uses next pointers from head to tail, while backward traversal uses prev pointers from tail to head. This allows more flexible navigation compared to the singly linked list.


// Insertion at the head of the list
void InsertAtHead(Node** head, char new_data)
{
Node* new_node = new Node(); // Create a new node
new_node->data = new_data; // Set data
new_node->next = (*head); // Link the new node to the current head
(*head) = new_node; // Move head to point to the new node
}
	Insertion operations are modified to maintain both next and prev pointers. The InsertAtHead and InsertAtEnd functions now update the prev and next pointers correctly, ensuring bidirectional linking. The InsertAtPosition function adjusts links between nodes, preserving the doubly linked structure.


void InsertAtPosition(Node* prev_node, char new_data)
{
if (prev_node == nullptr)
{
cout << "Previous node cannot be null." << endl;
return;
}
 	
Node* new_node = new Node(new_data); // Create a new node with data
new_node->next = prev_node->next; // Link new node to next of prev_node
new_node->prev = prev_node; // Link new node back to prev_node
 	
if (prev_node->next != nullptr)
{
prev_node->next->prev = new_node; // Link next node back to new node
}
 	
prev_node->next = new_node; // Link prev_node to new node
}
	Insertion operations are modified to maintain both next and prev pointers. The InsertAtHead and InsertAtEnd functions now update the prev and next pointers correctly, ensuring bidirectional linking. The InsertAtPosition function adjusts links between nodes, preserving the doubly linked structure.

	Deletion operations are simplified with the prev pointer, allowing direct access to neighboring nodes. The DeleteNode function updates adjacent nodes' pointers when a node is removed, whether it's at the head, tail, or middle, ensuring the list remains intact.

Table 3-4. Modified Operation for Doubly Linked Lists

7. Supplementary Activity

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

// Node class definition for circular linked list
class Node {
public:
    string song;
    Node *next;

    Node(string new_song) {
        song = new_song;
        next = nullptr;
    }
};
class CircularLinkedList {
private:
    Node* head;

public:
    CircularLinkedList() : head(nullptr) {}
```

```
// Function to create a playlist with multiple songs
void CreatePlaylist(string songs[], int n) {
    for (int i = 0; i < n; i++) {
        AddSong(songs[i]);
    }
}

// Function to add a song to the playlist
void AddSong(string new_song) {
    Node* new_node = new Node(new_song);
    if (head == nullptr) {
        head = new_node;
        new_node->next = head; // Pointing to itself
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next; // Traverse to the last node
        }
        temp->next = new_node;
        new_node->next = head; // Make it circular
    }
    cout << "Added: " << new_song << endl;
}

// Function to remove a song from the playlist
void RemoveSong(string song) {
    if (head == nullptr) {
        cout << "Playlist is empty." << endl;
        return;
    }

    Node* temp = head;
    Node* prev = nullptr;

    // If the song to be deleted is the head
    if (head->song == song) {
        if (head->next == head) { // Only one node
            delete head;
            head = nullptr;
        } else {
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = head->next;
            Node* to_delete = head;
```

```
            head = head->next;
            delete to_delete;
        }
        cout << "Removed: " << song << endl;
        return;
    }

    // Search for the song
    do {
        prev = temp;
        temp = temp->next;
    } while (temp != head && temp->song != song);

    // If the song was found
    if (temp == head) {
        cout << "Song not found." << endl;
    } else {
        prev->next = temp->next; // Remove the node
        delete temp;
        cout << "Removed: " << song << endl;
    }
}

// Function to play all songs
void PlayAllSongs() {
    if (head == nullptr) {
        cout << "Playlist is empty." << endl;
        return;
    }

    Node* temp = head;
    cout << "Playing all songs: ";
    do {
        cout << temp->song << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

// Function to get the next song
string NextSong() {
    if (head == nullptr) {
        return "No songs in the playlist.";
    }
    head = head->next; // Move to the next song
    return head->song;
```

```
    }

    // Function to get the previous song
    string PreviousSong() {
        if (head == nullptr) {
            return "No songs in the playlist.";
        }

        Node* temp = head;
        // Traverse to find the previous node
        while (temp->next != head) {
            temp = temp->next; // Find the last node
        }
        head = temp; // Move to the last node
        return head->song;
    }
};
int main() {
    CircularLinkedList playlist;

    // Creating a playlist with multiple songs
    string songs[] = {"Song1", "Song2", "Song3", "Song4"};
    playlist.CreatePlaylist(songs, 4);

    // Play all songs
    playlist.PlayAllSongs();

    // Add songs to the playlist
    playlist.AddSong("Song5");
    playlist.AddSong("Song6");

    // Play all songs again
    playlist.PlayAllSongs();

    // Remove a song
    playlist.RemoveSong("Song2");
    playlist.PlayAllSongs();

    // Try removing a non-existing song
    playlist.RemoveSong("Song10");

    // Final playlist
    playlist.PlayAllSongs();

    // Test next and previous functionality
    cout << "Next song: " << playlist.NextSong() << endl;
```

```
    cout << "Previous song: " << playlist.PreviousSong() << endl;

    return 0;
}
```

8. Conclusion

In this activity, I learned about circular linked lists and their advantages in applications like music players, which require circular traversal. I gained hands-on experience in designing data structures tailored to specific use cases, implementing functionalities such as adding, removing, and playing songs, and managing dynamic memory effectively. The procedure involved defining a node structure, developing essential list operations, and testing them systematically, which reinforced the importance of structured coding practices. Additionally, adding navigation features for next and previous songs enhanced user interaction, demonstrating the practical benefits of circular data structures. Overall, I believe I did well, but I recognize areas for improvement, such as enhancing error handling, testing edge cases more thoroughly, and creating a more user-friendly interface for better user experience in future projects.

9. Assessment Rubric