Activity No. 3 .1	
Hands-on Activity 3.1 Linked Lists	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 9/29/2024**
Section: CPE21S4	**Date Submitted: 9/29/2024**
Name(s): CALVIN EARL PLANTA	**Instructor: Dr. Sayo**

6. Output

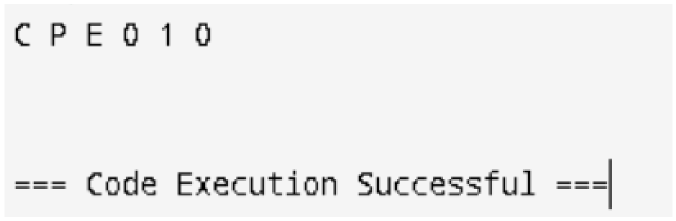
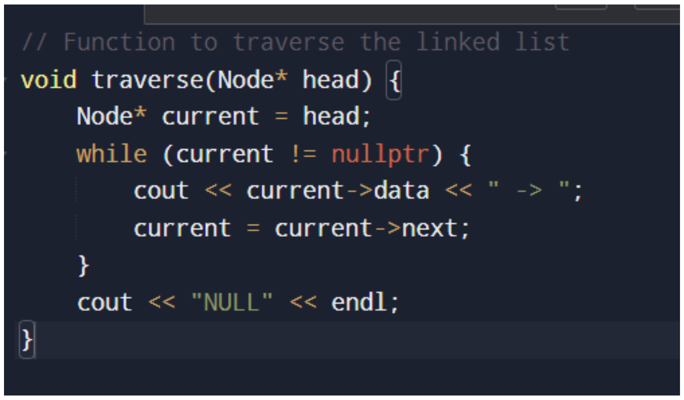
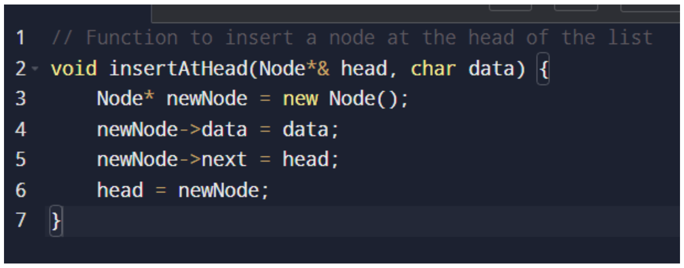
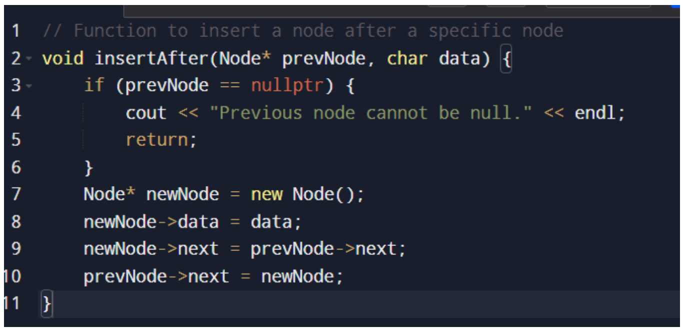
SCREENSHOT	 `C P E 0 1 0` `=== Code Execution Successful ===`
DISCUSSION	The code creates a linked list successfully, but it may be improved by including error-handling, dynamic allocation, and functions for insertion, deletion, and modification. Iterators and templates may also be considered for flexibility. The code supplied did not generate any output; however, with some modifications, the output is shown.

Table 3-1. Output of Initial/Simple Implementation

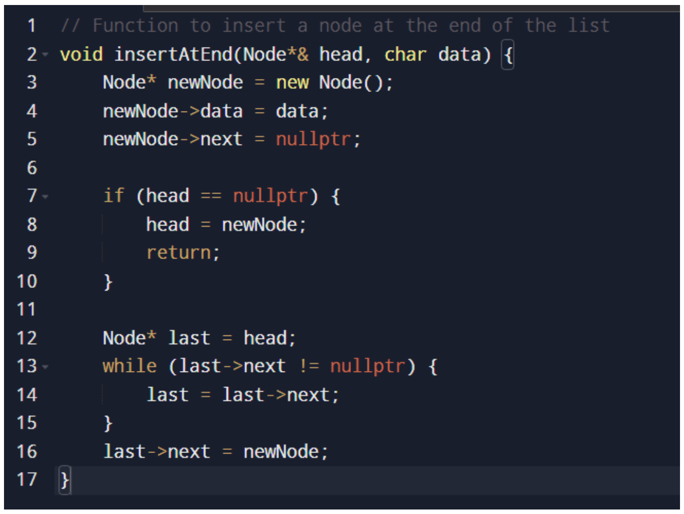
Operation	**Screenshot**
Traversal	 `// Function to traverse the linked list` `void traverse(Node* head) {` `Node* current = head;` `while (current != nullptr) {` `cout << current->data << " -> ";` `current = current->next;` `}` `cout << "NULL" << endl;` `}`
Insertion at head	 `// Function to insert a node at the head of the list` `void insertAtHead(Node*& head, char data) {` `Node* newNode = new Node();` `newNode->data = data;` `newNode->next = head;` `head = newNode;` `}`

Insertion at any part of the list



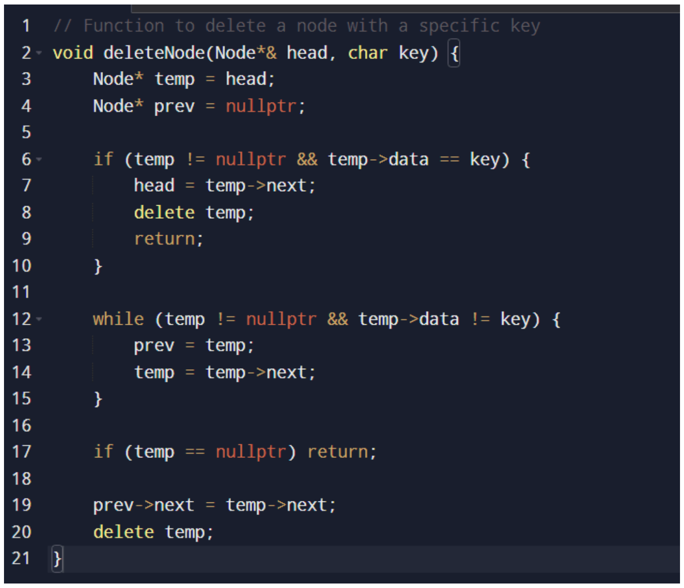
```cpp
// Function to insert a node after a specific node
void insertAfter(Node* prevNode, char data) {
    if (prevNode == nullptr) {
        cout << "Previous node cannot be null." << endl;
        return;
    }
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = prevNode->next;
    prevNode->next = newNode;
}
```

Insertion at the end



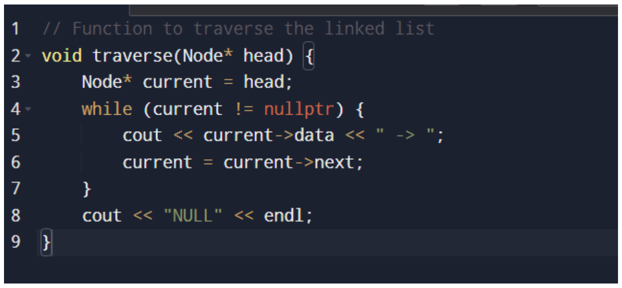
```cpp
// Function to insert a node at the end of the list
void insertAtEnd(Node*& head, char data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
        return;
    }

    Node* last = head;
    while (last->next != nullptr) {
        last = last->next;
    }
    last->next = newNode;
}
```

Deletion of a node

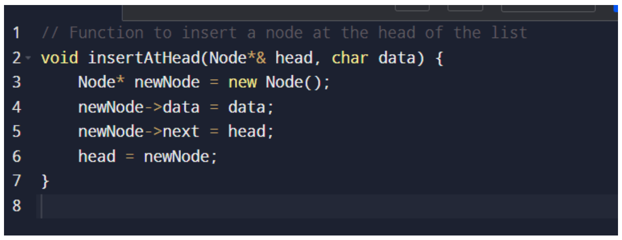


```cpp
// Function to delete a node with a specific key
void deleteNode(Node*& head, char key) {
    Node* temp = head;
    Node* prev = nullptr;

    if (temp != nullptr && temp->data == key) {
        head = temp->next;
        delete temp;
        return;
    }

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

    if (temp == nullptr) return;

    prev->next = temp->next;
    delete temp;
}
```

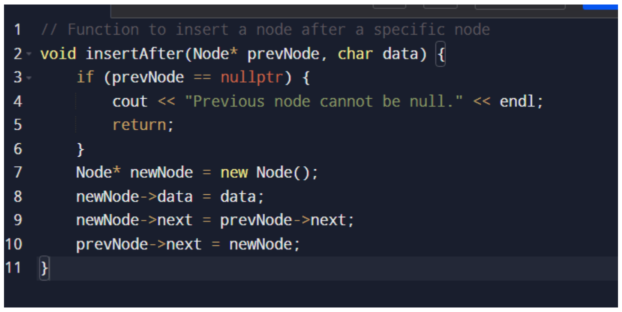
Table 3-2. Code for the List Operations

A.	**Source Code**	
	Console	Initial list: C -> P -> E -> 0 -> 1 -> 0 -> NULL

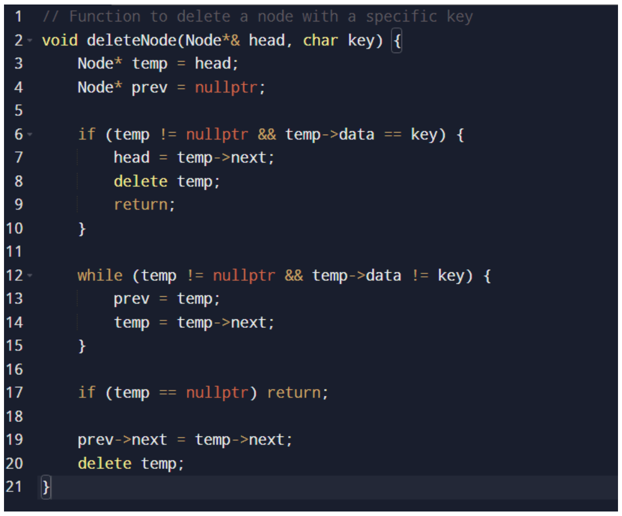
```cpp
// Function to traverse the linked list
void traverse(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " -> ";
        current = current->next;
    }
    cout << "NULL" << endl;
}
```

B.	**Source Code**	
	Console	After inserting 'G' at head: G -> C -> P -> E -> 0 -> 1 -> 0 -> NULL

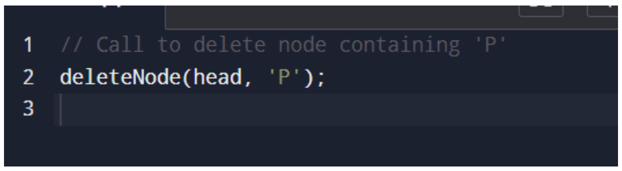
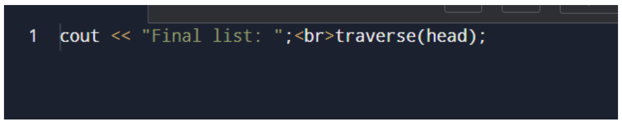
```cpp
// Function to insert a node at the head of the list
void insertAtHead(Node*& head, char data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

```

C.	**Source Code**	
	Console	After inserting 'E' after 'P': G -> C -> P -> E -> E -> 0 -> 1 -> 0 -> NULL

```cpp
// Function to insert a node after a specific node
void insertAfter(Node* prevNode, char data) {
    if (prevNode == nullptr) {
        cout << "Previous node cannot be null." << endl;
        return;
    }
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = prevNode->next;
    prevNode->next = newNode;
}
```

D.	**Source Code**	
	Console	After deleting 'C': G -> P -> E -> E -> 0 -> 1 -> 0 -> NULL

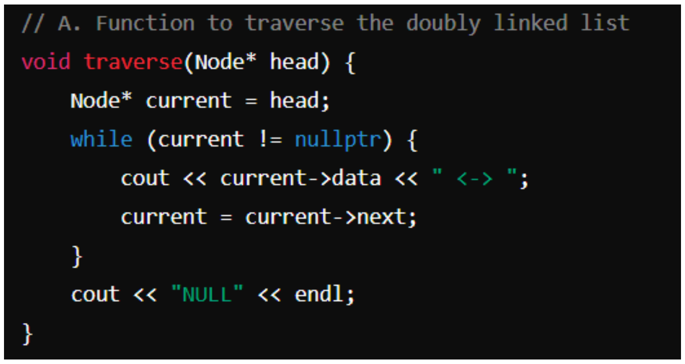
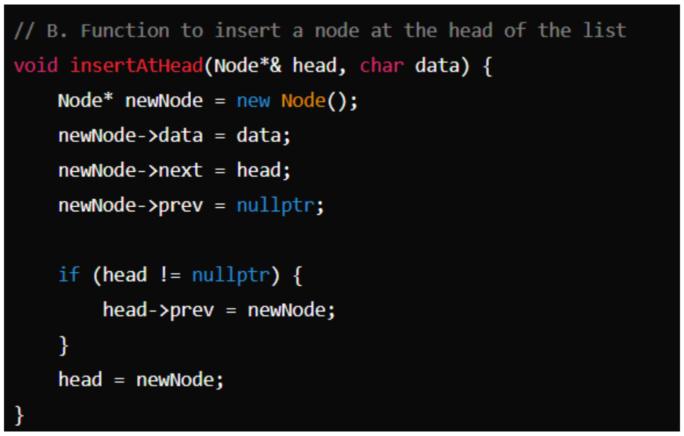
```cpp
// Function to delete a node with a specific key
void deleteNode(Node*& head, char key) {
    Node* temp = head;
    Node* prev = nullptr;

    if (temp != nullptr && temp->data == key) {
        head = temp->next;
        delete temp;
        return;
    }

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

    if (temp == nullptr) return;

    prev->next = temp->next;
    delete temp;
}
```

E.	**Source Code**	```cpp
// Call to delete node containing 'P'
deleteNode(head, 'P');
``
	Console	After deleting 'P': G -> E -> E -> 0 -> 1 -> 0 -> NULL
F.	**Source Code**	```cpp

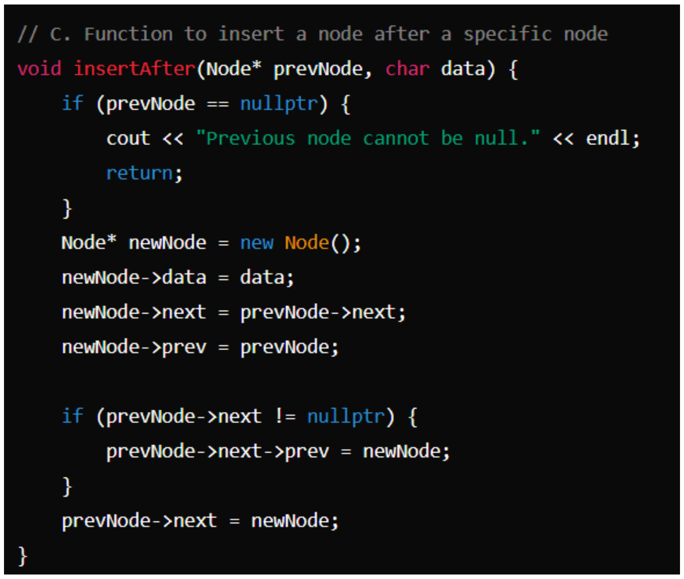
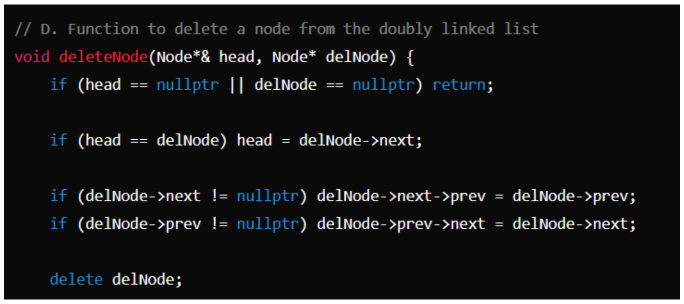
cout << "Final list: ";
traverse(head);

``
	Console	Final list: G -> E -> E -> 0 -> 1 -> 0 -> NULL

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

Screenshot(s)	**Analysis**
```cpp
// A. Function to traverse the doubly linked list
void traverse(Node* head) {
Node* current = head;
while (current != nullptr) {
cout << current->data << " <-> ";
current = current->next;
}
cout << "NULL" << endl;
}
``	This function correctly traverses the list and prints each node's data, confirming the links in both directions.
```cpp
// B. Function to insert a node at the head of the list
void insertAtHead(Node*& head, char data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = head;
    newNode->prev = nullptr;

    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
}
``` | The head is updated correctly, and the previous head node's prev pointer is set to the new node. |

	The new node is inserted correctly, maintaining the integrity of both next and prev pointers.
	delNode == nullptr) return; if (head == delNode) head = delNode->next; if (delNode->next != nullptr) delNode->next->prev = delNode->prev; if (delNode->prev != nullptr) delNode->prev->next = delNode->next; delete delNode; }```

```
// C. Function to insert a node after a specific node
void insertAfter(Node* prevNode, char data) {
    if (prevNode == nullptr) {
        cout << "Previous node cannot be null." << endl;
        return;
    }
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = prevNode->next;
    newNode->prev = prevNode;

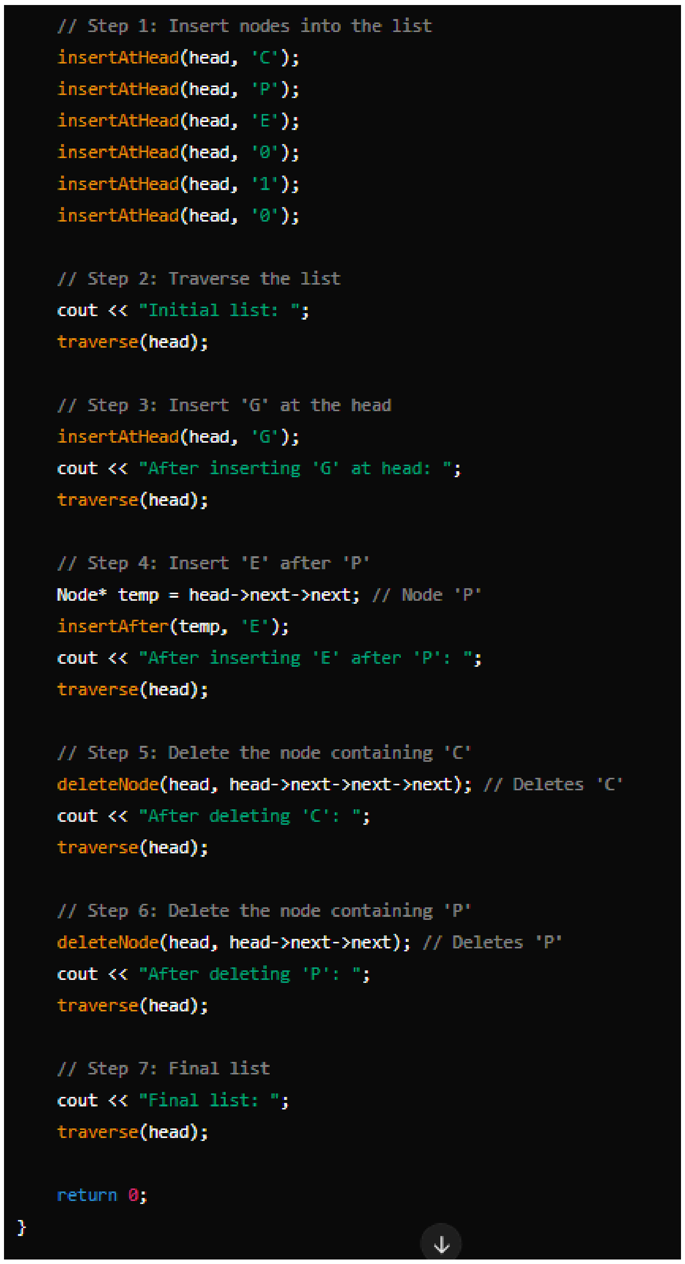
    if (prevNode->next != nullptr) {
        prevNode->next->prev = newNode;
    }
    prevNode->next = newNode;
}
```

```
// D. Function to delete a node from the doubly linked list
void deleteNode(Node*& head, Node* delNode) {
    if (head == nullptr || delNode == nullptr) return;

    if (head == delNode) head = delNode->next;

    if (delNode->next != nullptr) delNode->next->prev = delNode->prev;
    if (delNode->prev != nullptr) delNode->prev->next = delNode->next;

    delete delNode;
```



```
    // Step 1: Insert nodes into the list
    insertAtHead(head, 'C');
    insertAtHead(head, 'P');
    insertAtHead(head, 'E');
    insertAtHead(head, '0');
    insertAtHead(head, '1');
    insertAtHead(head, '0');

    // Step 2: Traverse the list
    cout << "Initial list: ";
    traverse(head);

    // Step 3: Insert 'G' at the head
    insertAtHead(head, 'G');
    cout << "After inserting 'G' at head: ";
    traverse(head);

    // Step 4: Insert 'E' after 'P'
    Node* temp = head->next->next; // Node 'P'
    insertAfter(temp, 'E');
    cout << "After inserting 'E' after 'P': ";
    traverse(head);

    // Step 5: Delete the node containing 'C'
    deleteNode(head, head->next->next->next); // Deletes 'C'
    cout << "After deleting 'C': ";
    traverse(head);

    // Step 6: Delete the node containing 'P'
    deleteNode(head, head->next->next); // Deletes 'P'
    cout << "After deleting 'P': ";
    traverse(head);

    // Step 7: Final list
    cout << "Final list: ";
    traverse(head);

    return 0;
}
```

The final list confirms that all operations have been executed correctly, reflecting the updated structure.

Table 3-4. Modified Operations for Doubly Linked Lists

7. Supplementary Activity

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

class Node {
public:
    string song;
    Node* next;

    Node(const string& song) : song(song), next(nullptr) {}
};
```

```
class CircularLinkedList {
private:
    Node* head;

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

    void addSong(const string& song) {
        Node* newNode = new Node(song);
        if (!head) {
            head = newNode;
            head->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
        }
    }

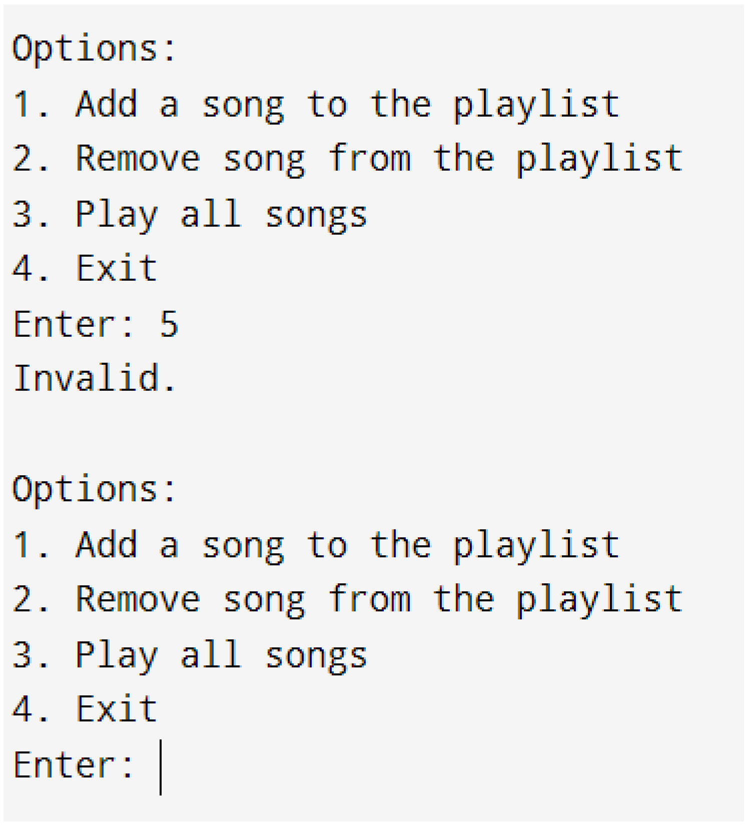
    void removeSong(const string& song) {
        if (head) {
            if (head->song == song) {
                if (head->next == head) {
                    delete head;
                    head = nullptr;
                } else {
                    Node* temp = head;
                    while (temp->next != head) {
                        temp = temp->next;
                    }
                    Node* toDelete = head;
                    temp->next = head->next;
                    head = head->next;
                    delete toDelete;
                }
            } else {
                Node* prev = nullptr;
                Node* temp = head;
                while (temp->next != head && temp->song != song) {
                    prev = temp;
                    temp = temp->next;
                }
                if (temp->song == song) {
                    prev->next = temp->next;
                    delete temp;
                }
            }
        }
    }

    void playAllSongs() const {
        if (head) {
            Node* temp = head;
            do {
                std::cout << temp->song << endl;
                temp = temp->next;
            } while (temp != head);
        }
    }
};

int main() {
    CircularLinkedList playlist;
    int choice;
    string song;

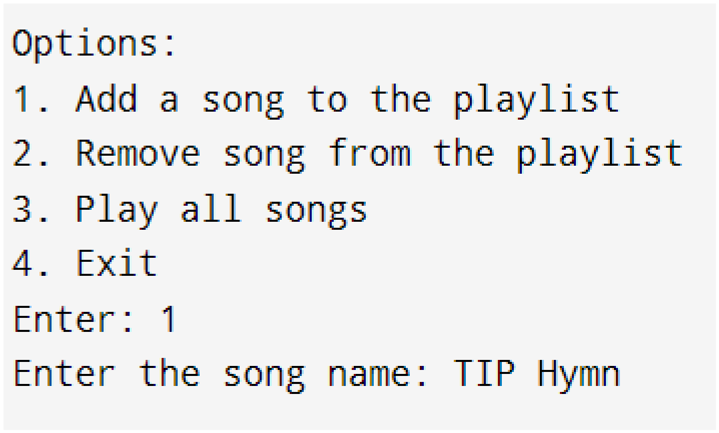
    while (true) {
        cout << endl << "Options:" << endl;
        cout << "1. Add a song to the playlist" << endl;
        cout << "2. Remove song from the playlist" << endl;
        cout << "3. Play all songs" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter the song name: ";
                cin.ignore();
                getline(cin, song);
                playlist.addSong(song);
                break;
            case 2:
                cout << "Enter the song name to remove: ";
                cin.ignore();
                getline(cin, song);
                playlist.removeSong(song);
                break;
            case 3:
                cout << "Playing all songs in the playlist:" << endl;
                playlist.playAllSongs();
                break;
            case 4:
                return 0;
            default:
                cout << "Invalid." << endl;
        }
    }
}
```

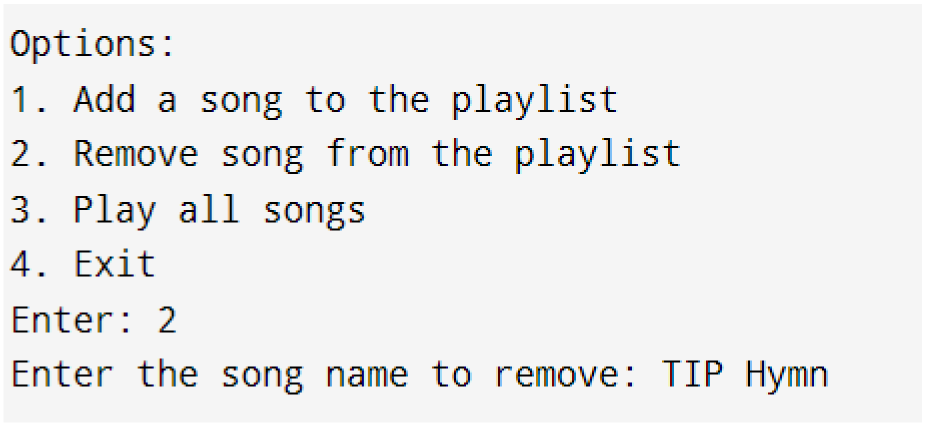


```
Options:
1. Add a song to the playlist
2. Remove song from the playlist
3. Play all songs
4. Exit
Enter: 5
Invalid.

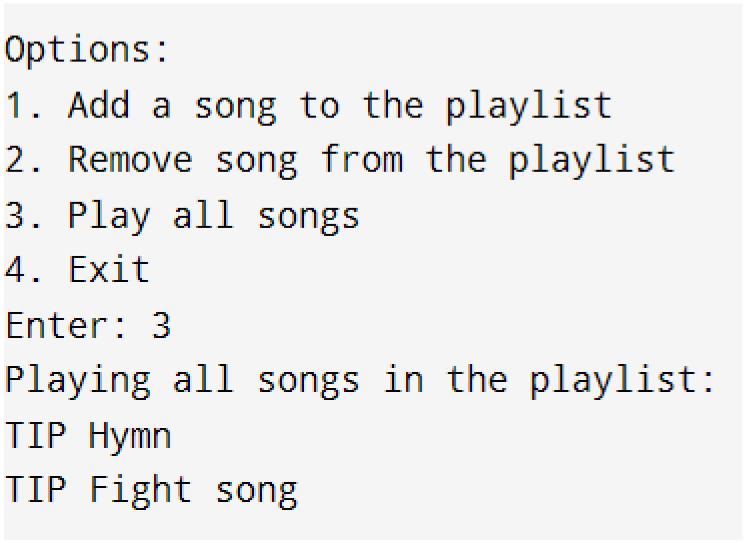
Options:
1. Add a song to the playlist
2. Remove song from the playlist
3. Play all songs
4. Exit
Enter: 
```



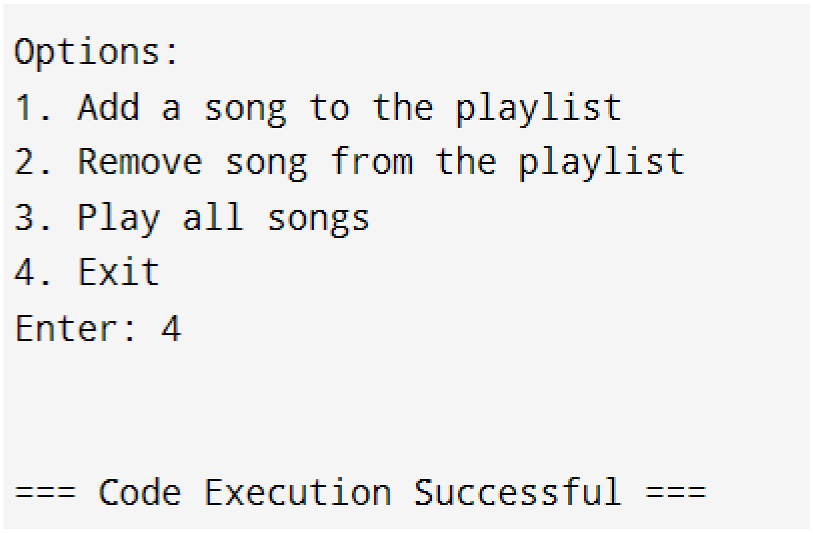
```
Options:
1. Add a song to the playlist
2. Remove song from the playlist
3. Play all songs
4. Exit
Enter: 1
Enter the song name: TIP Hymn
```



```
Options:
1. Add a song to the playlist
2. Remove song from the playlist
3. Play all songs
4. Exit
Enter: 2
Enter the song name to remove: TIP Hymn
```



```
Options:
1. Add a song to the playlist
2. Remove song from the playlist
3. Play all songs
4. Exit
Enter: 3
Playing all songs in the playlist:
TIP Hymn
TIP Fight song
```



```
Options:
1. Add a song to the playlist
2. Remove song from the playlist
3. Play all songs
4. Exit
Enter: 4


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

8. Conclusion

I now have a better grasp of linked lists as dynamic data structures and how they differ from arrays, especially with regard to their flexibility for frequent additions and deletions, thanks to this exercise. The concept of pointer-based structures was reinforced by implementing both singly and doubly linked lists, and the benefits of bidirectional traversal were emphasized by altering procedures for the doubly linked list. The additional task, which comprised building a circular linked list for a playlist of songs, demonstrated how linked list modifications may be used in practical contexts to improve features like music looping. Overall, I think I did well, putting the necessary functions into reality, but I am aware that I still need to practice more with more complicated operations and edge circumstances, such handling empty lists or maximizing insertion

9. Assessment Rubric