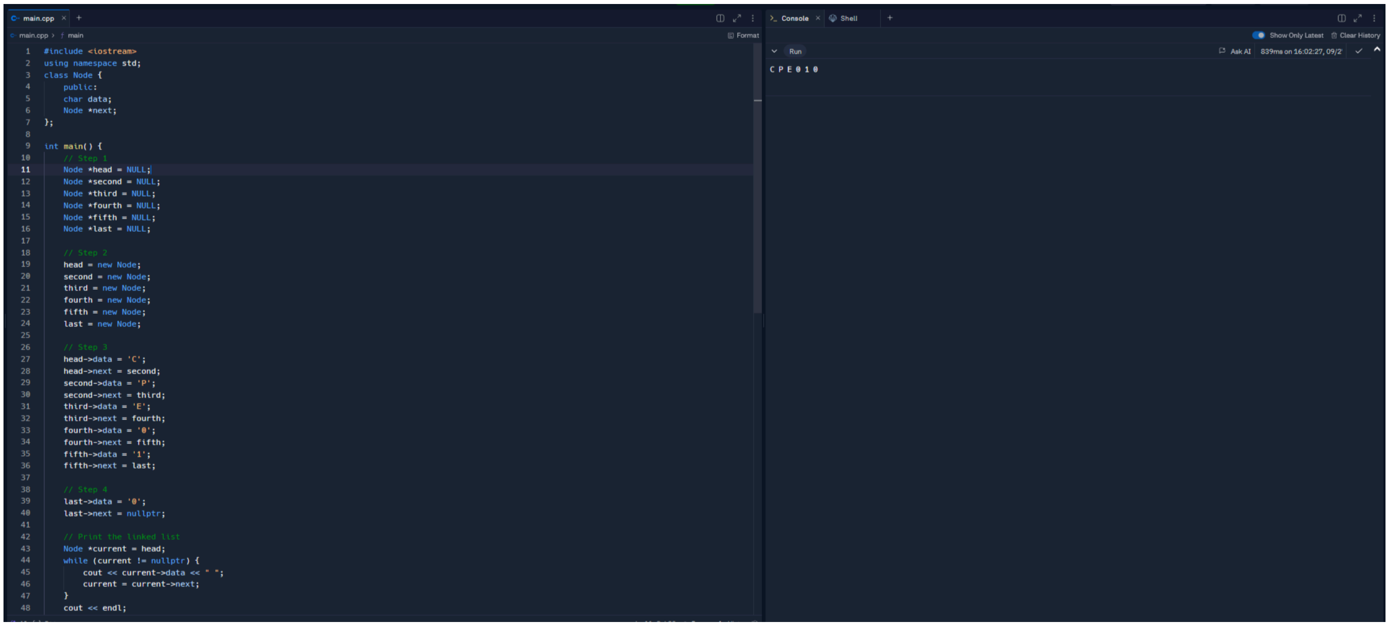
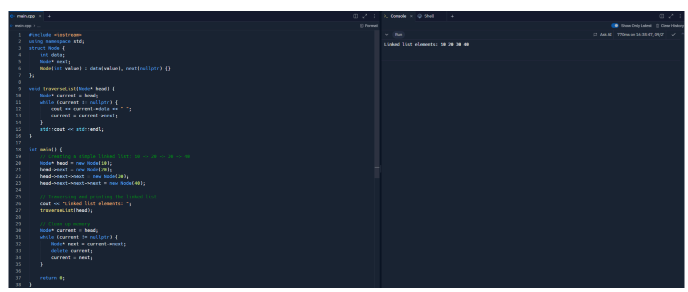
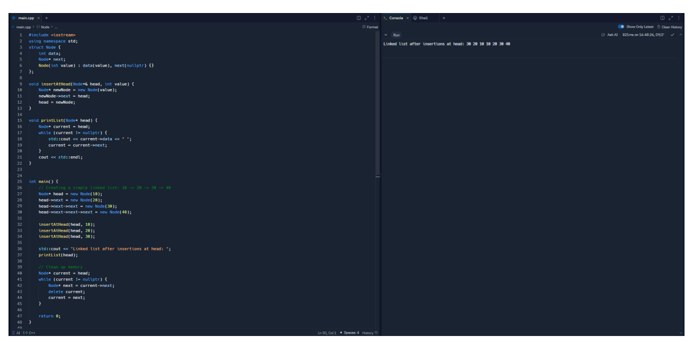
Hands-on Activity 3.1 Linked Lists

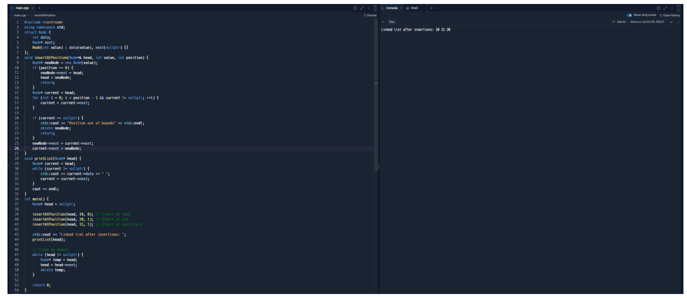
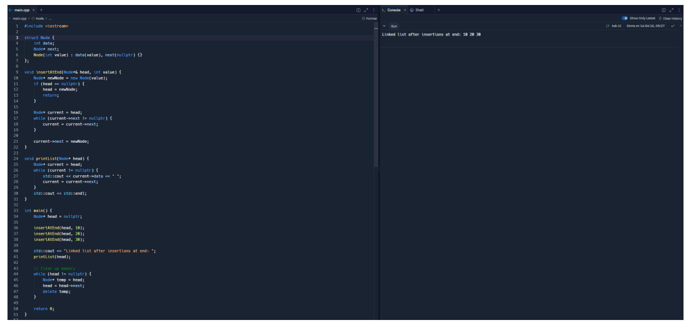
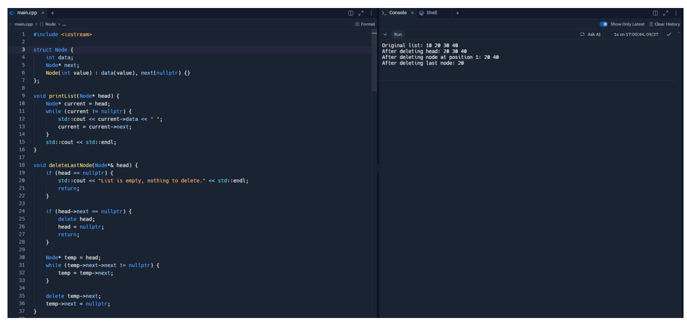
LINKED LISTS	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: SEPTEMBER 27, 2024**
Section: CPE21S1	**Date Submitted: SEPTEMBER 30, 2024**
Name(s): GASPAR, AARON ROWEN O.	**Instructor: MA'AM MARIA RIZETTE SAYO**

6. Output

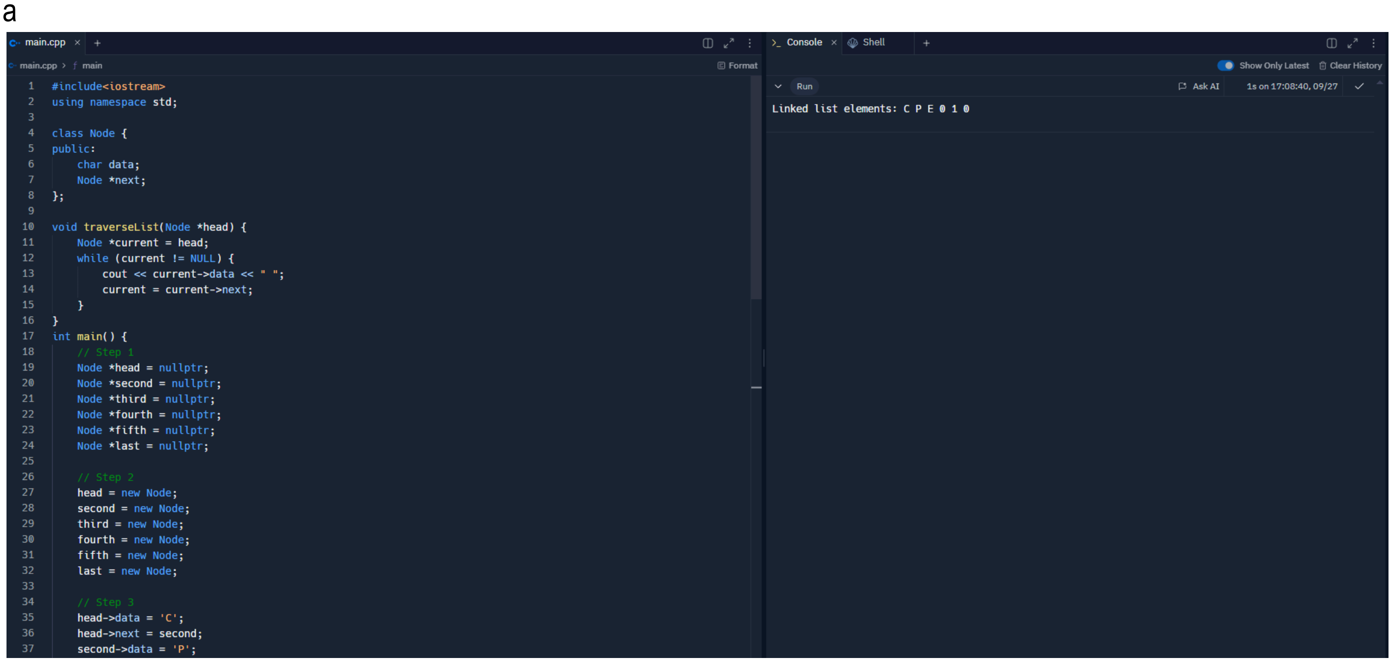


Discussion: The following code is a simple implementation of linked list. The output gives nothing, so I just added some code to print the output of the program.

Operation	Screenshot
Transversal	
Insertion at head	

Insertion at any part of the list	
Insertion at the end	
Deletion of node	

a



b

```cpp
#include<iostream>

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

// Function to insert a new node at the head of the list
Node* insertAtHead(Node* head, char new_data) {
    // Step 1: Create a new node
    Node* new_node = new Node();
    new_node->data = new_data;

    // Step 2: Make the next of the new node point to the current head
    new_node->next = head;

    // Step 3: Update the head to be the new node
    head = new_node;

    return head;
}

// Function to traverse and print the linked list
void traverseList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}

int main() {
    // Step 1
    Node *head = nullptr;
    Node *second = nullptr;
```

```
Linked List after insertion at head: G C P E 0 1 0
```

C

```cpp
#include<iostream>

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

// Function to insert a new node after a given node
void insertAfter(Node* prev_node, char new_data) {
    if (prev_node == nullptr) {
        std::cout << "The given previous node cannot be NULL" << std::endl;
        return;
    }
    // Step 1: Create a new node
    Node* new_node = new Node();
    new_node->data = new_data;

    // Step 2: Make the next of the new node point to the next of prev_node
    new_node->next = prev_node->next;

    // Step 3: Make the next of prev_node point to the new node
    prev_node->next = new_node;
}

// Function to insert a new node at the head of the list
Node* insertAtHead(Node* head, char new_data) {
    // Step 1: Create a new node
    Node* new_node = new Node();
    new_node->data = new_data;

    // Step 2: Make the next of the new node point to the current head
    new_node->next = head;

    // Step 3: Update the head to be the new node
    head = new_node;

    return head;
}
```

```
Linked List after insertion at head: G C P E E 0 1 0
```

d

```cpp
#include<iostream>

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

// Function to delete a node with a given key
Node* deleteNode(Node* head, char key) {
    // Store head node
    Node* temp = head;
    Node* prev = nullptr;

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

    // Search for the key to be deleted, keep track of the previous node
    while (temp != nullptr && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == nullptr) return head;
```

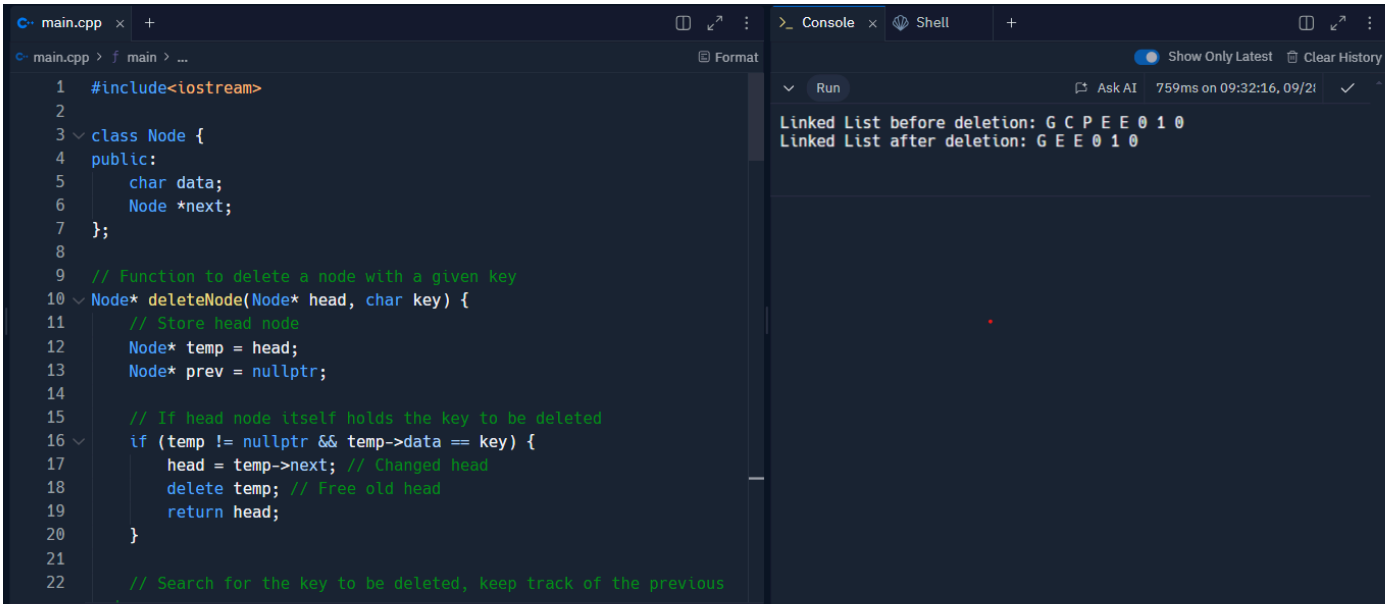
```
Linked List before deletion: G C P E E 0 1 0
Linked List after deletion: G P E E 0 1 0
```

e

```cpp
#include<iostream>

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

// Function to delete a node with a given key
Node* deleteNode(Node* head, char key) {
    // Store head node
    Node* temp = head;
    Node* prev = nullptr;

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

    // Search for the key to be deleted, keep track of the previous node
    while (temp != nullptr && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == nullptr) return head;
```

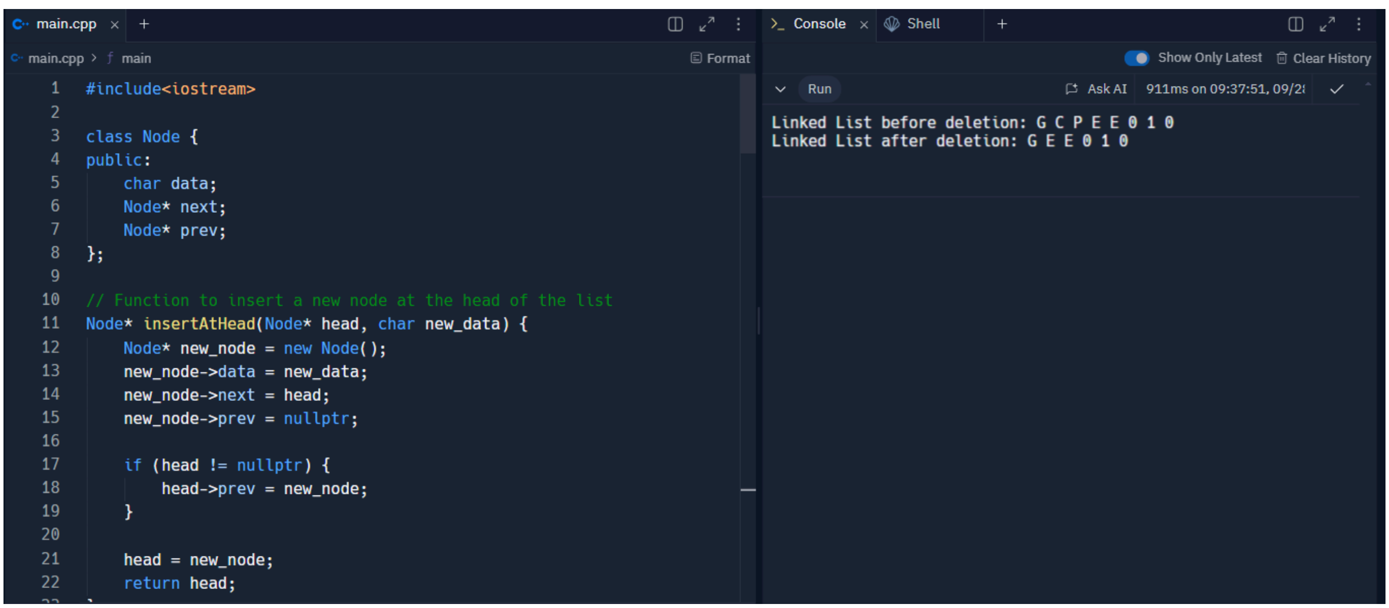
```
Linked List before deletion: G C P E E 0 1 0
Linked List after deletion: G E E 0 1 0
```

f



```cpp
#include<iostream>

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

// Function to delete a node with a given key
Node* deleteNode(Node* head, char key) {
    // Store head node
    Node* temp = head;
    Node* prev = nullptr;

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

    // Search for the key to be deleted, keep track of the previous
```

```
Linked List before deletion: G C P E E 0 1 0
Linked List after deletion: G E E 0 1 0
```



```cpp
#include<iostream>

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

// Function to insert a new node at the head of the list
Node* insertAtHead(Node* head, char new_data) {
    Node* new_node = new Node();
    new_node->data = new_data;
    new_node->next = head;
    new_node->prev = nullptr;

    if (head != nullptr) {
        head->prev = new_node;
    }

    head = new_node;
    return head;
```

```
Linked List before deletion: G C P E E 0 1 0
Linked List after deletion: G E E 0 1 0
```

Analysis:

The Node class is defined with three members: data, next, and prev. This allows each node to point to both the next and previous nodes in the list, enabling bidirectional traversal. The insertAtHead function inserts a new node at the beginning of the list. The insertAfter function inserts a new node after a specified node. The deleteNode function deletes a node with a specified key. The traverseList function prints the data of each node in the list. Overall, the code efficiently handles the basic operations of a doubly linked list.

7. Supplementary Activity

Program:

```cpp
int main() {
    Playlist myPlaylist;

    myPlaylist.addSong("Song 1");
    myPlaylist.addSong("Song 2");
    myPlaylist.addSong("Song 3");

    std::cout << "Playing all songs in the playlist:" << endl;
    myPlaylist.playAllSongs();

    std::cout << "\nPlaying next song:" << endl;
    myPlaylist.nextSong();

    std::cout << "\nPlaying previous song:" << endl;
    myPlaylist.previousSong();

    std::cout << "\nRemoving 'Song 2' from the playlist." << endl;
    myPlaylist.removeSong("Song 2");

    std::cout << "\nPlaying all songs in the playlist:" << endl;
    myPlaylist.playAllSongs();

    return 0;
}
```

```cpp
    // Function to play the next song
    void nextSong() {
        if (current) {
            current = current->next;
            std::cout << "Playing: " << current->song << endl;
        }
    }

    // Function to play the previous song
    void previousSong() {
        if (current) {
            current = current->prev;
            std::cout << "Playing: " << current->song << endl;
        }
    }

    // Function to play all songs in the playlist
    void playAllSongs() const {
        if (!head) return;

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

    ~Playlist() {
        if (!head) return;

        Node* temp = head;
        do {
            Node* next = temp->next;
            delete temp;
            temp = next;
        } while (temp != head);
    }
};
```

```
// Function to remove a song from the playlist
void removeSong(const string& songName) {
    if (!head) return;

    Node* temp = head;

    // If the song to be deleted is the head
    if (head->song == songName) {
        if (head == head->next) {
            delete head;
            head = nullptr;
            current = nullptr;
        } else {
            Node* tail = head->prev;
            head = head->next;
            tail->next = head;
            head->prev = tail;
            delete temp;
            current = head;
        }
        return;
    }

    // Search for the song to be deleted
    do {
        if (temp->song == songName) {
            temp->prev->next = temp->next;
            temp->next->prev = temp->prev;
            if (current == temp) {
                current = temp->next;
            }
            delete temp;
            return;
        }
        temp = temp->next;
    } while (temp != head);
}
```

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

struct Node {
    string song;
    Node* next;
    Node* prev; // Adding a previous pointer for doubly linked list
};

class Playlist {
private:
    Node* head;
    Node* current; // Pointer to keep track of the current song

public:
    Playlist() : head(nullptr), current(nullptr) {}

    // Function to add a song to the playlist
    void addSong(const string& songName) {
        Node* newNode = new Node();
        newNode->song = songName;
        if (!head) {
            head = newNode;
            head->next = head;
            head->prev = head;
            current = head;
        } else {
            Node* tail = head->prev;
            tail->next = newNode;
            newNode->prev = tail;
            newNode->next = head;
            head->prev = newNode;
        }
    }

```

Output:

```
Playing all songs in the playlist:
Song 1
Song 2
Song 3

Playing next song:
Playing: Song 2

Playing previous song:
Playing: Song 1

Removing 'Song 2' from the playlist.

Playing all songs in the playlist:
Song 1
Song 3
```

8. Conclusion

In conclusion, learning linked lists in C++ is indeed a crucial milestone for any aspiring programmer. It offers a hands-on approach to understanding dynamic data structures, which are essential for managing data efficiently. By working with linked lists, you gain valuable insights into memory management and pointer manipulation—skills that are fundamental in C++ programming. Mastering linked lists not only sets the stage for tackling more advanced topics in computer science, such as trees, graphs, and complex algorithms, but it also equips you with the practical know-how to solve real-world problems. Whether you're optimizing a music playlist, managing a dynamic set of data, or implementing a game loop, the principles you learn from linked lists are directly applicable.