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:** 9/27/24
Section: CPE21S4	**Date Submitted:** 9/27/24
Name(s): Santos, Andrei R.	**Instructor:** Professor Maria Rizette Sayo
6. Output	

Screenshot

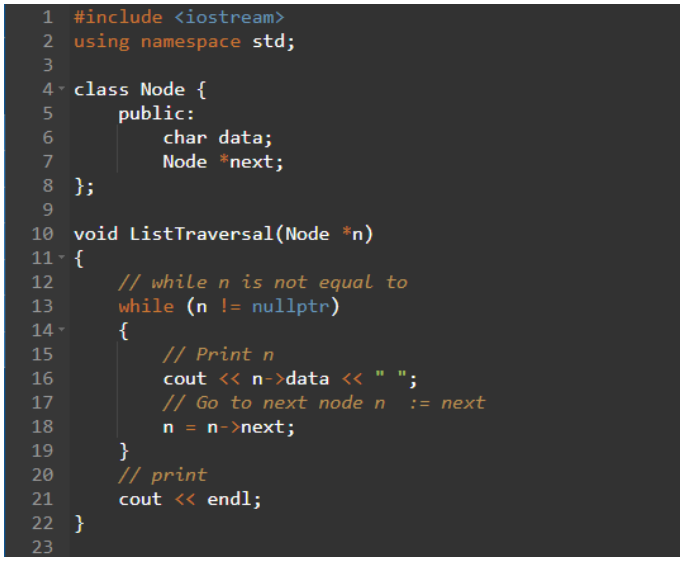
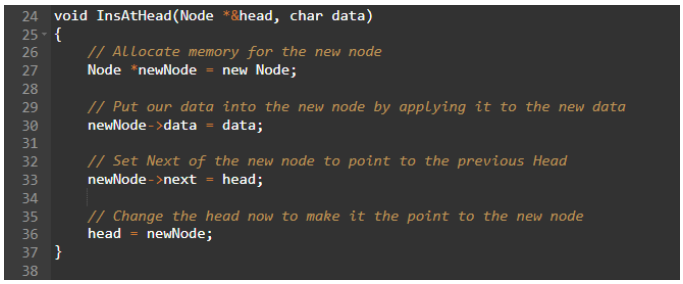
main.cpp

```
#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;

}
```

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

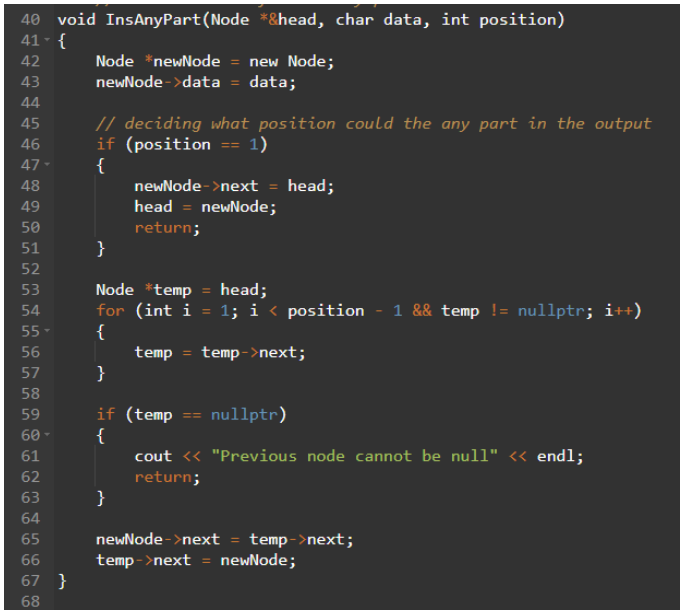
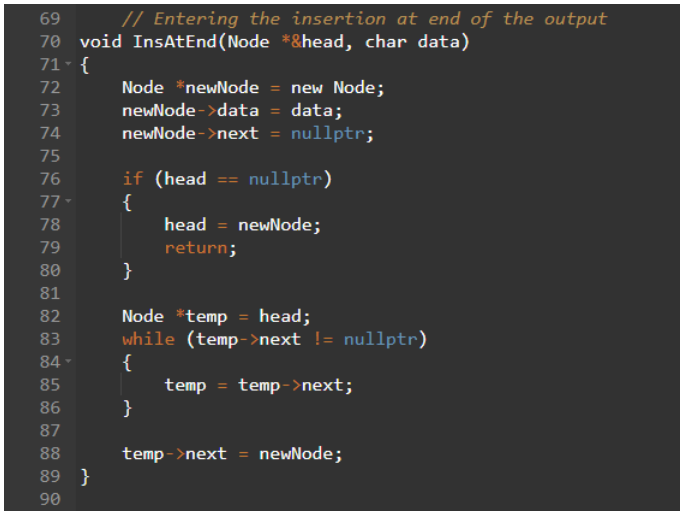
Discussion	After I completed coding the program, it just needs a simple output since we can see there is no cout wherein the cout helps to show the output of the program after compiling.

Table 3-1. Output of Initial/Simple Implementation

Operation	**Screenshot**
Traversal	
Insertion at head	

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

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

void ListTraversal(Node *n)
{
    // while n is not equal to
    while (n != nullptr)
    {
        // Print n
        cout << n->data << " ";
        // Go to next node n  := next
        n = n->next;
    }
    // print
    cout << endl;
}
```

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

    // Put our data into the new node by applying it to the new data
    newNode->data = data;

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

    // Change the head now to make it the point to the new node
    head = newNode;
}
```

Insertion at any part of the list	
Insertion at the end	

```cpp
void InsAnyPart(Node *&head, char data, int position)
{
    Node *newNode = new Node;
    newNode->data = data;

    // deciding what position could the any part in the output
    if (position == 1)
    {
        newNode->next = head;
        head = newNode;
        return;
    }

    Node *temp = head;
    for (int i = 1; i < position - 1 && temp != nullptr; i++)
    {
        temp = temp->next;
    }

    if (temp == nullptr)
    {
        cout << "Previous node cannot be null" << endl;
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}
```

```cpp
    // Entering the insertion at end of the output
void InsAtEnd(Node *&head, char data)
{
    Node *newNode = new Node;
    newNode->data = data;
    newNode->next = nullptr;

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

    Node *temp = head;
    while (temp->next != nullptr)
    {
        temp = temp->next;
    }

    temp->next = newNode;
}
```

Deletion of a node	

```
91      // now, the deletion of the chosen character
92  void DelNo(Node *&head, char data)
93  {
94      if (head == nullptr) {
95          cout << "This is an Empty list" << endl;
96          return;
97      }
98
99      if (head->data == data)
100     {
101         Node *temp = head;
102         head = head->next;
103         delete temp;
104         return;
105     }
106
107     Node *temp = head;
108     while (temp->next != nullptr)
109     {
110         if (temp->next->data == data)
111         {
112             Node *nodeToDelete = temp->next;
113             temp->next = temp->next->next;
114             delete nodeToDelete;
115             return;
116         }
117         temp = temp->next;
118     }
119
120     cout << "The node was not found." << endl;
121 }
```

Table 3-2. Code for the List Operations

a	Source Code	`// Task a final ouput is CPE101`
		`cout << "Task a (Initial traversal of the list): ";`
		`ListTraversal(head);`

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

b	Source Code	`// Task b final ouput is GCPE101`
		`InsAtHead(&head, 'G');`
		`cout << "Task b (Insert 'G' at the start): ";`
		`ListTraversal(head);`

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

c	Source Code	`// Task c final ouput is GCPEE101`
		`Node* temp = head;`
		`while (temp != nullptr && temp->data != 'P')  // Find node with 'P'`
		`{`
		`  temp = temp->next;`
		`}`

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

d	Source Code	`// Task d final ouput is GPEE101`
		`deleno(&head, 'C');`
		`cout << "Task d (Delete node with 'C'): ";`
		`ListTraversal(head);`

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

e	Source Code	`// Task e final ouput is GEE101`
		`deleno(&head, 'P');`
		`cout << "Task e (Delete node with 'P'): ";`
		`ListTraversal(head);`

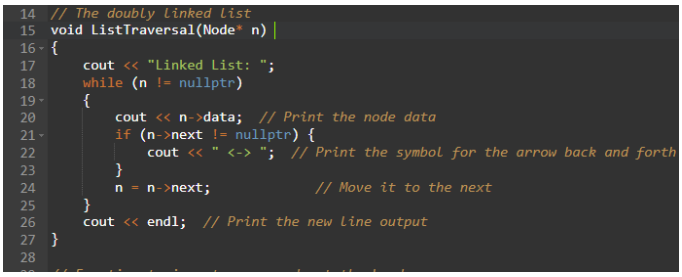
	Console	`Task e (Delete node with 'P'): G E E 1 0 1`

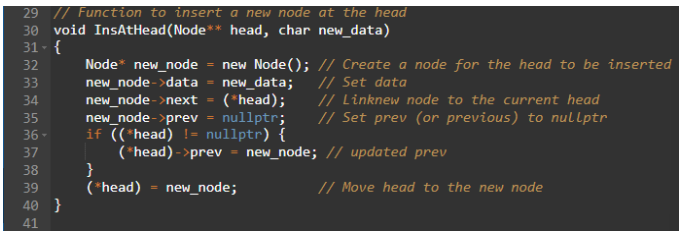
f	Source Code	`// Task f final ouput is GEE101`
		`cout << "Task f (Final list traversal): ";`
		`ListTraversal(head);`

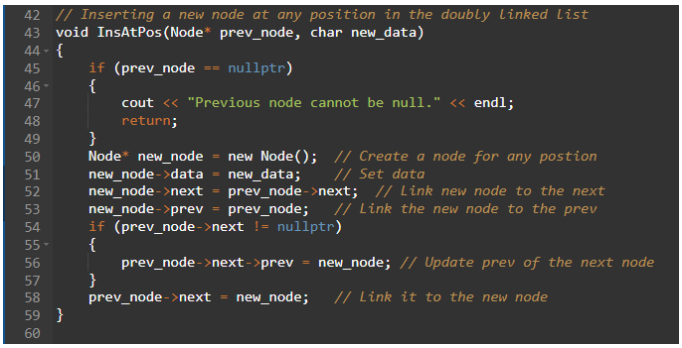
	Console	`Task f (Final list traversal): G E E 1 0 1`

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

Screenshots(s)	Analysis


`// The doubly linked list`
`void ListTraversal(Node* n)`
`{`
`  cout << "Linked List: ";`
`  while (n != nullptr)`
`  {`
`  cout << n->data;  // Print the node data`
`  if (n->next != nullptr) {`
`  cout << " <-> ";  // Print the symbol for the arrow back and forth`
`  }`
`  n = n->next;  // Move it to the next`
`  }`
`  cout << endl;  // Print the new line output`
`}`
	In this function traversal, showing the data we can see the symbol that I used, the <-> which represents the bidirectional links of the codes.


`// Function to insert a new node at the head`
`void InsAtHead(Node** head, char new_data)`
`{`
`  Node* new_node = new Node(); // Create a node for the head to be inserted`
`  new_node->data = new_data;  // Set data`
`  new_node->next = (*head);  // Linknew node to the current head`
`  new_node->prev = nullptr;  // Set prev (or previous) to nullptr`
`  if ((*head) != nullptr) {`
`  (*head)->prev = new_node; // updated prev`
`  }`
`  (*head) = new_node;  // Move head to the new node`
`}`
	Here, I updated it with the prev and next, which are the previous of the node and the next of the node letting and making the list updated.


`// Inserting a new node at any position in the doubly linked list`
`void InsAtPos(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 node for any postion`
`  new_node->data = new_data;  // Set data`
`  new_node->next = prev_node->next;  // Link new node to the next`
`  new_node->prev = prev_node;  // Link the new node to the prev`
`  if (prev_node->next != nullptr)`
`  {`
`  prev_node->next->prev = new_node; // Update prev of the next node`
`  }`
`  prev_node->next = new_node;  // Link it to the new node`
`}`
	In this part, I can manipulate the desired data that I can put inside the doubly linked lists at the specified position, still it links to the previous up to the next one.

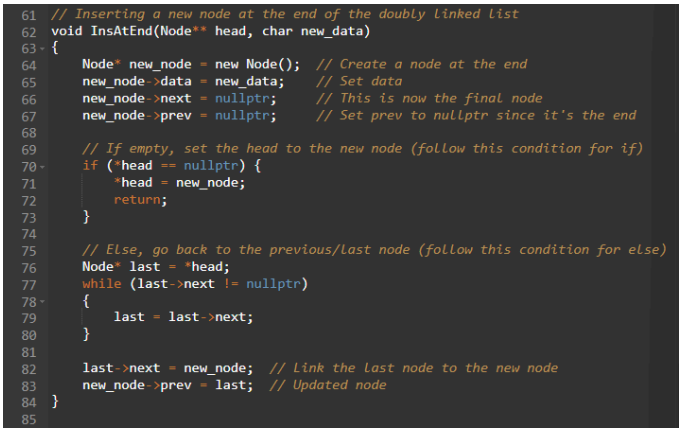
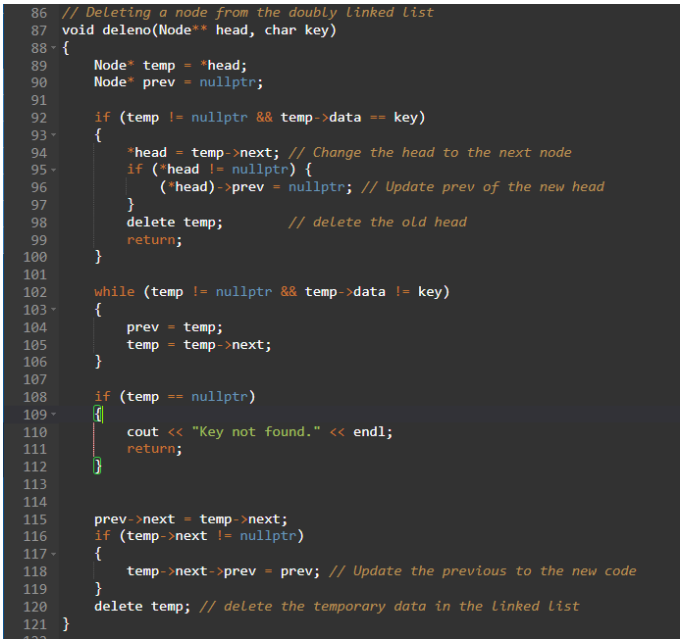
	With the use of the prev and next pointers to the existing and new node, it creates and inserts from the beginning to the end of the node until it reaches to the null
	Lastly, I can now delete a specific node, which I can choose from the given with the data. With the help of the prev and next, I can delete anode maintaining the lists links from its previous and next nodes.

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(string s) : song(s), next(nullptr) {}
};

class Music_opm_list {
   Node* head = nullptr;
public:
   void AddSong(string s) {
      Node* new_node = new Node(s);
```

```
        if (!head) {
            head = new_node;
            new_node->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) temp = temp->next;
            temp->next = new_node;
            new_node->next = head;
        }
    }

    void RemoveSong(string s) {
        if (!head) return;
        Node* temp = head, *prev = nullptr;
        if (head->song == s) {
            if (head->next == head) { delete head; head = nullptr; }
            else {
                while (temp->next != head) temp = temp->next;
                temp->next = head->next;
                delete head; head = temp->next;
            }
            return;
        }
        do {
            prev = temp; temp = temp->next;
        } while (temp != head && temp->song != s);
        if (temp != head) { prev->next = temp->next; delete temp; }
    }

    void dispsong() {
        if (!head) return;
        Node* temp = head;
        do {
            cout << "• " << temp->song << endl;
            temp = temp->next;
        } while (temp != head);
    }
};

int main() {
    Music_opm_list playlist;

    // Create OPM song playlist
    cout << "Playlist: my Favorite OPM Songs\n";
    playlist.AddSong("Dahan Dahan");
    playlist.AddSong("Bawat Piyesa");
    playlist.AddSong("Sino");
    playlist.dispsong();

    // Add a song to the playlist
    cout << "\nAdding song: Pwede Ba\n";
```

```
    playlist.AddSong("Pwede Ba");
    playlist.dispsong();

    // Playing the song in loop
    cout << "\nNow playing in loop: \n";
    playlist.dispsong();

    // Remove the song "Pwede Ba"
    cout << "\nRemoving song: Pwede Ba\n";
    playlist.RemoveSong("Pwede Ba");
    playlist.dispsong();

    return 0;
}
```

The output:

8. Conclusion

I have learned how to insert and delete data in nodes using linked lists, understanding the role of head and null. I also realized that singly linked lists and doubly linked lists share similarities, but differ in traversal part. Singly linked lists follow a linear path, while doubly linked lists allow backward and forward movement. In terms of efficiency, singly linked lists are better suited for simpler tasks, while doubly linked lists are ideal for more complex operations, it shows greater flexibility and requires more memory and run time. The supplementary activity demonstrated how to implement adding or deleting data in a playlist or list.

This activity was a bit challenging for me at first since it introduced new concepts and tasks I wasn't familiar with. However, through my own understanding and a little bit of research, I was able to grasp the use of singly and doubly linked lists, particularly how the prev and next pointers function. By working through the steps of adding and deleting songs in the code.

9. Assessment Rubric