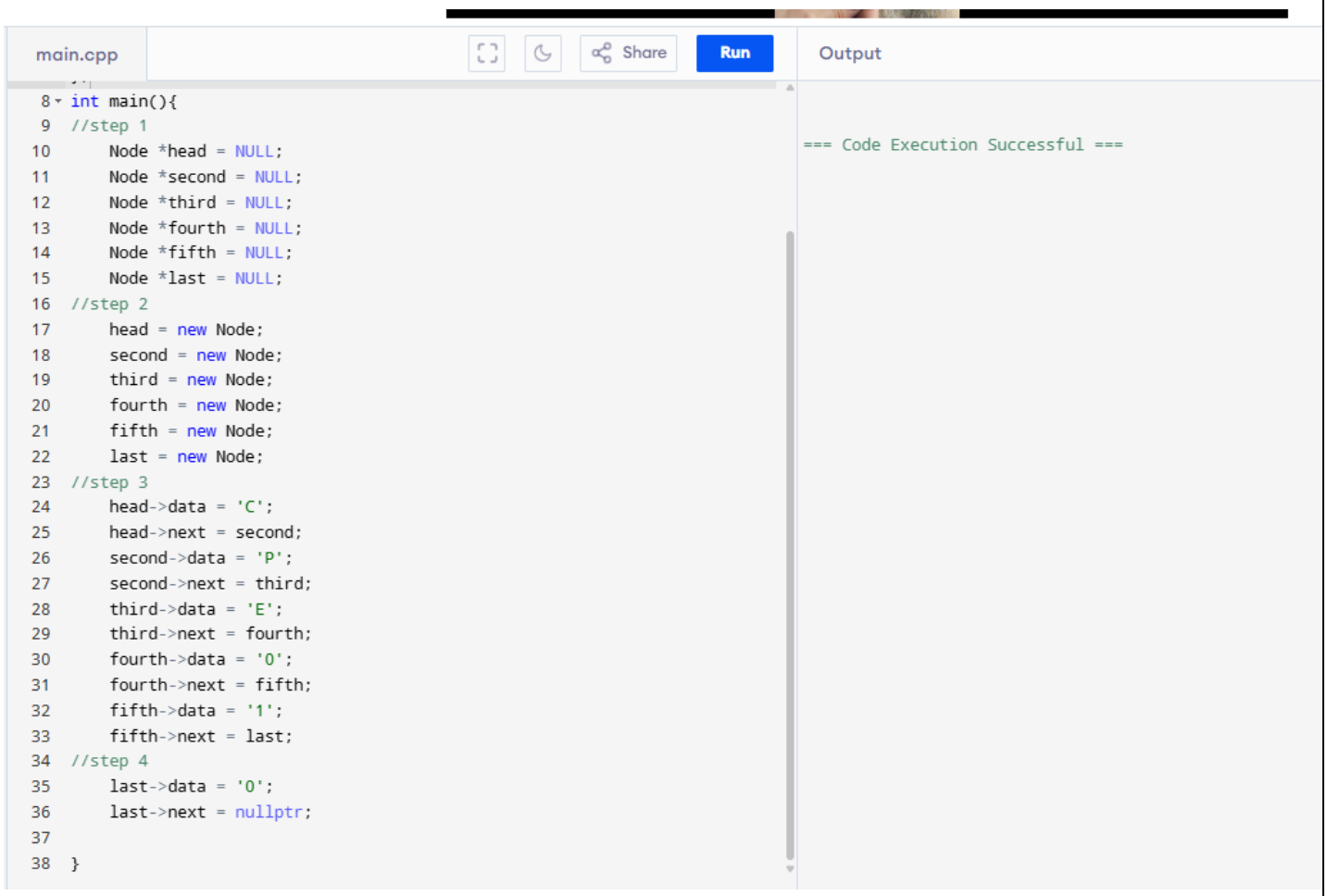
Activity No. 3	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 8/14/2025**
Section: CPE21S4	**Date Submitted: 8/14/2025**
Name(s): Anastacio, Lester Arvid P.	**Instructor: Engr. Jimlord M. Quejado**

6. Output

TABLE 3-1
CODE:



DISCUSSION:
I've noticed as I observe the code closely, that it lacks a way to actually show the output to find out whether it works perfectly, another thing is the excessive use of the null on step 1, it doesn't really harm the code but it is an necessary amount and lastly is the lack of way to actually delete nodes, we only have a way to create new nodes but no way to delete those nodes, in which could result into a overflow.

TABLE 3-2
TRAVERSAL

```c
// Traversal function
void ListTraversal(struct Node* n) {
    while (n != NULL) {
        printf("%d ", n->data);
        n = n->next;
    }
    printf("\n");
}
```

INSERTION AT AHEAD

```c
// Insert node at the head
void insertAtHead(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}
```

INSERTION AT ANY PART OF LIST

```c
// Insert after a given node
void insertAfter(struct Node* prev_node, int new_data) {
    if (prev_node == NULL) {
        printf("Previous node cannot be NULL.\n");
        return;
    }
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}
```

INSERTION AT THE END

```c
// Insert node at the end
void insertAtEnd(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    struct Node* last = *head_ref;
    new_node->data = new_data;
    new_node->next = NULL;

    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }

    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

DELETION OF NODE

```c
// Delete a node by key
void deleteNode(struct Node** head_ref, int key) {
    struct Node* temp = *head_ref;
    struct Node* prev = NULL;

    if (temp != NULL && temp->data == key) {
        *head_ref = temp->next;
        free(temp);
        return;
    }

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

    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}
```

TABLE 3-3
A.

main.cpp

Share Run

Output

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    char data;
    struct Node* next;
};

void insertAtHead(struct Node** head_ref, char new_data) {
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

void ListTraversal(struct Node* n) {
    while (n != NULL) {
        printf("%c", n->data);
        n = n->next;
    }
    printf("\n");
}
```

```
a. Traverse the list:
CPE101


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

B.

```
int main() {
    struct Node* head = NULL;

    // Insert in reverse order to get C P E 1 0 1
    insertAtHead(&head, '1');
    insertAtHead(&head, '0');
    insertAtHead(&head, '1');
    insertAtHead(&head, 'E');
    insertAtHead(&head, 'P');
    insertAtHead(&head, 'C');

    printf("a. Traverse the list:\n");
    ListTraversal(head);  // Output: CPE101


    insertAtHead(&head, 'G');
    printf("b. Insert 'G' at the start of the list:\n");
    ListTraversal(head);
```

Output

```
a. Traverse the list:
CPE101
b. Insert 'G' at the start of the list:
GCPE101
```

C.

```
int main() {
    struct Node* head = NULL;

    // a. Create initial list: C P E 1 0 1
    insertAtHead(&head, '1');
    insertAtHead(&head, '0');
    insertAtHead(&head, '1');
    insertAtHead(&head, 'E');
    insertAtHead(&head, 'P');
    insertAtHead(&head, 'C');

    printf("a. Traverse the list:\n");
    ListTraversal(head);  // Output: CPE101

    // b. Insert 'G' at the start of the list
    insertAtHead(&head, 'G');
    printf("b. Insert 'G' at the start of the list:\n");
    ListTraversal(head);  // Output: GCPE101

    // c. Insert 'E' after node 'P'
    struct Node* pNode = findNode(head, 'P');
    if (pNode != NULL) {
        insertAfter(pNode, 'E');
    }
    printf("c. Insert 'E' after node 'P':\n");
    ListTraversal(head);  // Output: GCPEE101

    return 0;
}
```

Output

```
a. Traverse the list:
CPE101
b. Insert 'G' at the start of the list:
GCPE101
c. Insert 'E' after node 'P':
GCPEE101


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

D.

```
int main() {
    struct Node* head = NULL;

    // a. Create initial list: C P E 1 0 1
    insertAtHead(&head, '1');
    insertAtHead(&head, '0');
    insertAtHead(&head, '1');
    insertAtHead(&head, 'E');
    insertAtHead(&head, 'P');
    insertAtHead(&head, 'C');

    printf("a. Traverse the list:\n");
    ListTraversal(head);  // Output: CPE101

    // b. Insert 'G' at the start of the list
    insertAtHead(&head, 'G');
    printf("b. Insert 'G' at the start of the list:\n");
    ListTraversal(head);  // Output: GCPE101

    // c. Insert 'E' after node 'P'
    struct Node* pNode = findNode(head, 'P');
    if (pNode != NULL) {
        insertAfter(pNode, 'E');
    }
    printf("c. Insert 'E' after node 'P':\n");
    ListTraversal(head);  // Output: GCPEE101

    // d. Delete node containing 'C'
    deleteNode(&head, 'C');
    printf("d. Delete node containing 'C':\n");
    ListTraversal(head);  // Output: GPEE101

    return 0;
}
```

Output

```
a. Traverse the list:
CPE101
b. Insert 'G' at the start of the list:
GCPE101
c. Insert 'E' after node 'P':
GCPEE101
d. Delete node containing 'C':
GPEE101


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

E.

```
65
66  int main() {
67      struct Node* head = NULL;
68
69      // a. Create initial list: C P E 1 0 1
70      insertAtHead(&head, '1');
71      insertAtHead(&head, '0');
72      insertAtHead(&head, '1');
73      insertAtHead(&head, 'E');
74      insertAtHead(&head, 'P');
75      insertAtHead(&head, 'C');
76
77      printf("a. Traverse the list:\n");
78      ListTraversal(head);  // Output: CPE101
79
80      // b. Insert 'G' at the start of the list
81      insertAtHead(&head, 'G');
82      printf("b. Insert 'G' at the start of the list:\n");
83      ListTraversal(head);  // Output: GCPE101
84
85      // c. Insert 'E' after node 'P'
86      struct Node* pNode = findNode(head, 'P');
87      if (pNode != NULL) {
88          insertAfter(pNode, 'E');
89      }
90      printf("c. Insert 'E' after node 'P':\n");
91      ListTraversal(head);  // Output: GCPEE101
92
93      // d. Delete node containing 'C'
94      deleteNode(&head, 'C');
95      printf("d. Delete node containing 'C':\n");
96      ListTraversal(head);  // Output: GPEE101
97
98      // e. Delete node containing 'P'
99      deleteNode(&head, 'P');
00      printf("e. Delete node containing 'P':\n");
01      ListTraversal(head);  // Output: GEE101
02
03      return 0;
04  }
```

Output

```
a. Traverse the list:
CPE101
b. Insert 'G' at the start of the list:
GCPE101
c. Insert 'E' after node 'P':
GCPEE101
d. Delete node containing 'C':
GPEE101
e. Delete node containing 'P':
GEE101
```

F.

```
int main() {
    struct Node* head = NULL;

    // a. Create initial list: C P E 1 0 1
    insertAtHead(&head, '1');
    insertAtHead(&head, '0');
    insertAtHead(&head, '1');
    insertAtHead(&head, 'E');
    insertAtHead(&head, 'P');
    insertAtHead(&head, 'C');

    printf("a. Traverse the list:\n");
    ListTraversal(head);  // Output: CPE101

    // b. Insert 'G' at the start of the list
    insertAtHead(&head, 'G');
    printf("b. Insert 'G' at the start of the list:\n");
    ListTraversal(head);  // Output: GCPE101

    // c. Insert 'E' after node 'P'
    struct Node* pNode = findNode(head, 'P');
    if (pNode != NULL) {
        insertAfter(pNode, 'E');
    }
    printf("c. Insert 'E' after node 'P':\n");
    ListTraversal(head);  // Output: GCPEE101

    // d. Delete node containing 'C'
    deleteNode(&head, 'C');
    printf("d. Delete node containing 'C':\n");
    ListTraversal(head);  // Output: GPEE101

    // e. Delete node containing 'P'
    deleteNode(&head, 'P');
    printf("e. Delete node containing 'P':\n");
    ListTraversal(head);  // Output: GEE101

    // f. Final traversal
    printf("f. Final traversal of the list:\n");
    ListTraversal(head);  // Output: GEE101

    return 0;
}
```

Output

```
a. Traverse the list:
CPE101
b. Insert 'G' at the start of the list:
GCPE101
c. Insert 'E' after node 'P':
GCPEE101
d. Delete node containing 'C':
GPEE101
e. Delete node containing 'P':
GEE101
f. Final traversal of the list:
GEE101
```

TABLE 3-4

SCREENSHOTS:

```
7
8  void insertAtHead(Node** head_ref, char new_data) {
9      Node* new_node = new Node();
10     new_node->data = new_data;
11     new_node->prev = nullptr;        // new node has no previous node
12     new_node->next = *head_ref;      // next points to old head
13
14     if (*head_ref != nullptr)
15         (*head_ref)->prev = new_node;  // old head prev points back to new
               node
16
17     *head_ref = new_node;            // head now points to new node
18 }
19
```

ANALYSIS:

It adds a new node to the beginning of a **doubly linked list**. It takes a pointer to a pointer to the head of the list and a character new_data as input, dynamically allocates a new node, sets its data to new_data and its prev pointer to nullptr, and then updates the `next` pointer of the new node and the prev pointer of the old head (if it exists) to correctly link the new node to the list before finally updating the head of the list to point to the new node.

SCREENSHOT:

```cpp
void insertAfter(Node* prev_node, char new_data) {
    if (prev_node == nullptr) {
        std::cout << "Previous node cannot be NULL.\n";
        return;
    }
    Node* new_node = new Node();
    new_node->data = new_data;

    new_node->next = prev_node->next;  // new node's next is prev_node's
        next
    new_node->prev = prev_node;         // new node's prev is prev_node

    if (prev_node->next != nullptr)
        prev_node->next->prev = new_node;  // update next node's prev

    prev_node->next = new_node;          // prev_node's next is new node
}
```

ANALYSIS:
it creates a new node, sets its data, and then correctly links the `next` and prev pointers of the new node, the prev_node, and the node that comes after prev_node to properly insert the new node into the list.

SCREENSHOT:

```cpp
void deleteNode(Node** head_ref, char key) {
    Node* temp = *head_ref;

    // Find the node to delete
    while (temp != nullptr && temp->data != key) {
        temp = temp->next;
    }

    if (temp == nullptr)  // Not found
        return;

    // If node to be deleted is head
    if (*head_ref == temp)
        *head_ref = temp->next;

    // Change next only if node to be deleted is NOT the last node
    if (temp->next != nullptr)
        temp->next->prev = temp->prev;

    // Change prev only if node to be deleted is NOT the first node
    if (temp->prev != nullptr)
        temp->prev->next = temp->next;

    delete temp;
}
```

ANALYSIS:
It is designed to remove a node with a specific char key from a doubly linked list. The function takes a pointer to the head of the list and a character key as input. It first iterates through the list to find the node with the matching key. If the node isn't found, the function returns. If the node to be deleted is the head, it updates the head to the next node.

SCREENSHOT:

```
void ListTraversal(Node* node) {
    while (node != nullptr) {
        std::cout << node->data;
        node = node->next;
    }
    std::cout << std::endl;
}
```

ANALYSIS:

This traverses a singly or doubly linked list and prints the data of each node. It takes a pointer to a node as input. The function uses a while loop that continues as long as the current node pointer is not nullptr. Inside the loop, it prints the `data` of the current node and then updates the node pointer to point to the `next` node in the list. After the loop finishes, it prints a newline character to format the output.

7. Supplementary Activity

CODE SCREENSHOTS:

main.cpp Share Run

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

// Node structure for each song
struct SongNode {
    string title;
    SongNode* next;
    SongNode* prev;

    SongNode(string t) : title(t), next(nullptr), prev(nullptr) {}
};

// Playlist class using Circular Doubly Linked List
class Playlist {
private:
    SongNode* head;
    SongNode* current;

public:
    Playlist() {
        head = nullptr;
        current = nullptr;
    }

    // Add a song to the playlist
    void addSong(string title) {
        SongNode* newNode = new SongNode(title);

        if (!head) {
            head = newNode;
            head->next = head;
            head->prev = head;
            current = head;
        } else {
            SongNode* tail = head->prev;
```

main.cpp

Share Run

```
    } else {
        SongNode* tail = head->prev;
        tail->next = newNode;
        newNode->prev = tail;
        newNode->next = head;
        head->prev = newNode;
    }

    cout << "Added: " << title << endl;
}

// Remove a song from the playlist
void removeSong(string title) {
    if (!head) {
        cout << "Playlist is empty." << endl;
        return;
    }

    SongNode* temp = head;
    bool found = false;

    do {
        if (temp->title == title) {
            found = true;
            break;
        }
        temp = temp->next;
    } while (temp != head);

    if (!found) {
        cout << "Song '" << title << "' not found in the playlist." << endl;
        return;
    }

    if (temp->next == temp) { // only one node
        delete temp;
```

```
69      if (temp->next == temp) { // only one node
70          delete temp;
71          head = nullptr;
72          current = nullptr;
73      } else {
74          temp->prev->next = temp->next;
75          temp->next->prev = temp->prev;
76
77          if (temp == head) head = temp->next;
78          if (temp == current) current = temp->next;
79
80          delete temp;
81      }
82
83      cout << "Removed: " << title << endl;
84  }
85
86  // Play current song
87  void playCurrent() {
88      if (current)
89          cout << "Now playing: " << current->title << endl;
90      else
91          cout << "No song is selected." << endl;
92  }
93
94  // Play all songs in a loop (once through the list)
95  void playAllSongs() {
96      if (!head) {
97          cout << "Playlist is empty." << endl;
98          return;
99      }
100
101     cout << "Playing all songs in playlist:" << endl;
102     SongNode* temp = head;
103     do {
104         cout << "Now playing: " << temp->title << endl;
```

```
100
101         cout << "Playing all songs in playlist:" << endl;
102         SongNode* temp = head;
103         do {
104             cout << "Now playing: " << temp->title << endl;
105             temp = temp->next;
106         } while (temp != head);
107     }
108
109     // Go to next song
110     void nextSong() {
111         if (current)
112             current = current->next;
113         playCurrent();
114     }
115
116     // Go to previous song
117     void previousSong() {
118         if (current)
119             current = current->prev;
120         playCurrent();
121     }
122 };
123
124 // Main function to test the playlist
125 int main() {
126     Playlist myPlaylist;
127
128     myPlaylist.addSong("Song A");
129     myPlaylist.addSong("Song B");
130     myPlaylist.addSong("Song C");
131
132     myPlaylist.playCurrent();    // Should play "Song A"
133     myPlaylist.nextSong();       // Should play "Song B"
134     myPlaylist.nextSong();       // Should play "Song C"
135     myPlaylist.previousSong();   // Should play "Song B"
```

```
23
24 // Main function to test the playlist
25 int main() {
26     Playlist myPlaylist;
27
28     myPlaylist.addSong("Song A");
29     myPlaylist.addSong("Song B");
30     myPlaylist.addSong("Song C");
31
32     myPlaylist.playCurrent();    // Should play "Song A"
33     myPlaylist.nextSong();       // Should play "Song B"
34     myPlaylist.nextSong();       // Should play "Song C"
35     myPlaylist.previousSong();   // Should play "Song B"
36
37     myPlaylist.playAllSongs();   // Should play A -> B -> C
38
39     myPlaylist.removeSong("Song B");
40     myPlaylist.playAllSongs();   // Should play A -> C
41
42     myPlaylist.addSong("Song D");
43     myPlaylist.playAllSongs();   // Should play A -> C -> D
44
45     return 0;
46 }
47
```

OUTPUT:

8. Conclusion

During this activity, I've learned how to implement the single and double linked list and it greatly improved my skills in C++, it was also quite enjoyable trying out new functions and such to create the music player and also ways to how the linking actually works, from head to traversers and etc. and during this activity, I've also realized how much I can improved myself more, since I did struggled quite a lot just to make some functions works and all, and also learned my lesson to always save the documents so that I wont have to redo everything again from scratch.

9. Assessment Rubric