

Activity No. 3

Course Code: CPE010

Program: Computer Engineering

Course Title: Data Structures and Algorithms

Date Performed: 8/14/2025

Section: CPE21S4

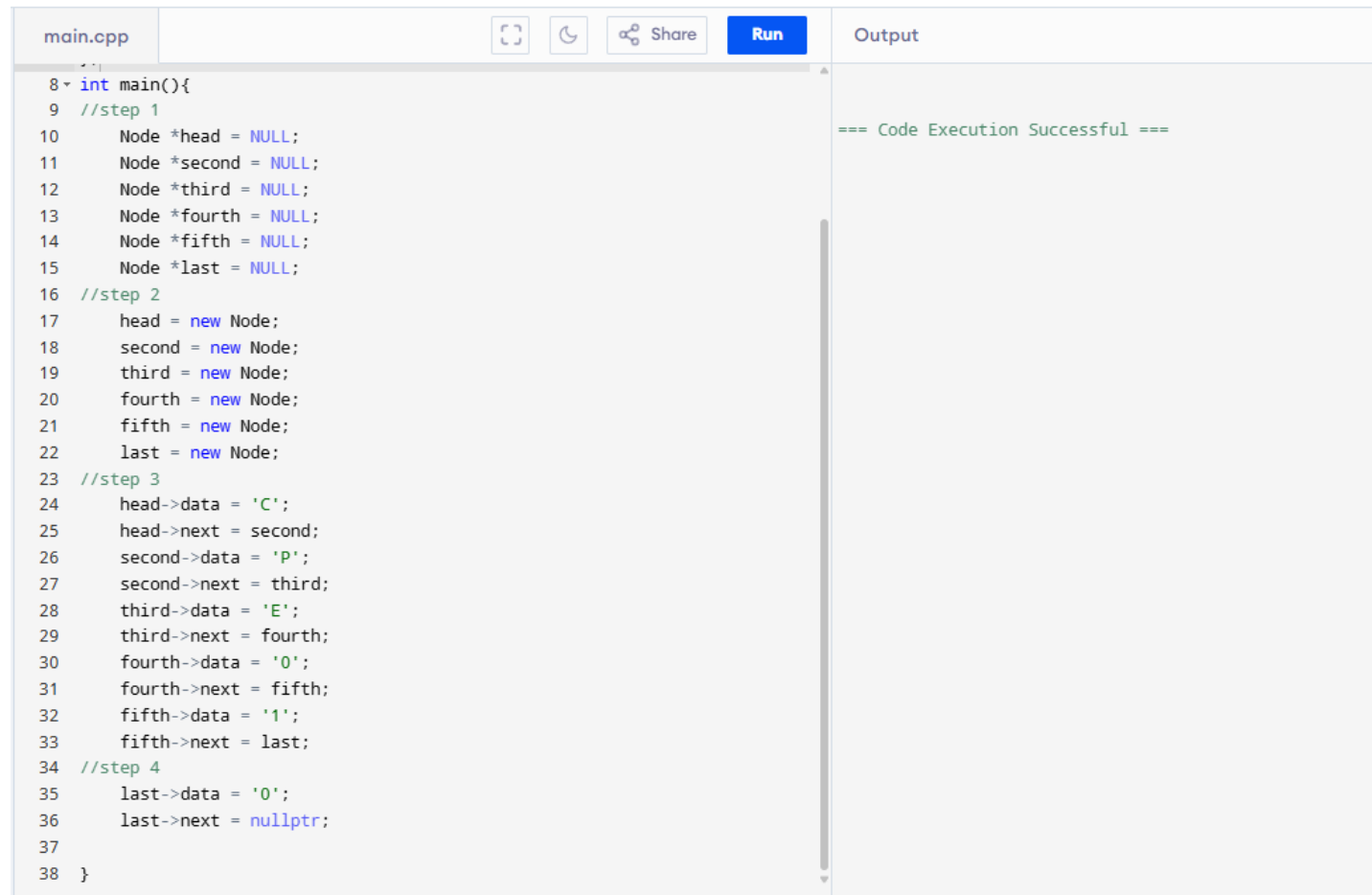
Date Submitted: 8/14/2025

Name(s): Anastacio, Lester Arvid P.

Instructor: Engr. Jimlord M. Quejado

6. Output

TABLE 3-1
CODE:



The screenshot displays a C++ code editor with a file named 'main.cpp'. The code implements a linked list with five nodes. It starts by initializing pointers for head, second, third, fourth, fifth, and last to NULL. Then, it creates five new Node objects and assigns them to these pointers. Next, it sets the 'data' and 'next' fields for each node: head points to second, second to third, third to fourth, fourth to fifth, and fifth to last. Finally, it sets last's data to '0' and its next pointer to nullptr. The code is commented with //step 1 through //step 4. To the right of the code editor, the 'Output' pane shows the message '=== Code Execution Successful ==='.

```
8~ int main(){
9  //step 1
10     Node *head = NULL;
11     Node *second = NULL;
12     Node *third = NULL;
13     Node *fourth = NULL;
14     Node *fifth = NULL;
15     Node *last = NULL;
16  //step 2
17     head = new Node;
18     second = new Node;
19     third = new Node;
20     fourth = new Node;
21     fifth = new Node;
22     last = new Node;
23  //step 3
24     head->data = 'C';
25     head->next = second;
26     second->data = 'P';
27     second->next = third;
28     third->data = 'E';
29     third->next = fourth;
30     fourth->data = '0';
31     fourth->next = fifth;
32     fifth->data = '1';
33     fifth->next = last;
34  //step 4
35     last->data = '0';
36     last->next = nullptr;
37
38 }
```

Output: === Code Execution Successful ===

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

```

63
64 // Traversal function
65 void ListTraversal(struct Node* n) {
66     while (n != NULL) {
67         printf("%d ", n->data);
68         n = n->next;
69     }
70     printf("\n");
71 }
72

```

INSERTION AT AHEAD

```

,
9 // Insert node at the head
9 void insertAtHead(struct Node** head_ref, int new_data) {
1     struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
2     new_node->data = new_data;
3     new_node->next = *head_ref;
4     *head_ref = new_node;
5 }
5

```

INSERTION AT ANY PART OF LIST

```

16
17 // Insert after a given node
18 void insertAfter(struct Node* prev_node, int new_data) {
19     if (prev_node == NULL) {
20         printf("Previous node cannot be NULL.\n");
21         return;
22     }
23     struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
24     new_node->data = new_data;
25     new_node->next = prev_node->next;
26     prev_node->next = new_node;
27 }
28

```

INSERTION AT THE END

```

28
29 // Insert node at the end
30 void insertAtEnd(struct Node** head_ref, int new_data) {
31     struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
32     struct Node* last = *head_ref;
33     new_node->data = new_data;
34     new_node->next = NULL;
35
36     if (*head_ref == NULL) {
37         *head_ref = new_node;
38         return;
39     }
40
41     while (last->next != NULL)
42         last = last->next;
43     last->next = new_node;
44 }

```

DELETION OF NODE

```

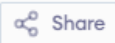
45
46 // Delete a node by key
47 void deleteNode(struct Node** head_ref, int key) {
48     struct Node* temp = *head_ref;
49     struct Node* prev = NULL;
50
51     if (temp != NULL && temp->data == key) {
52         *head_ref = temp->next;
53         free(temp);
54         return;
55     }
56
57     while (temp != NULL && temp->data != key) {
58         prev = temp;
59         temp = temp->next;
60     }
61
62     if (temp == NULL) return;
63
64     prev->next = temp->next;
65     free(temp);
66 }
67

```

TABLE 3-3

A.

main.cpp



Run

Output

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct Node {
5     char data;
6     struct Node* next;
7 };
8
9 void insertAtHead(struct Node** head_ref, char new_data) {
10     struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
11     new_node->data = new_data;
12     new_node->next = *head_ref;
13     *head_ref = new_node;
14 }
15
16 void ListTraversal(struct Node* n) {
17     while (n != NULL) {
18         printf("%c", n->data);
19         n = n->next;
20     }
21     printf("\n");
22 }
23
```

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.

```
5
6 int main() {
7     struct Node* head = NULL;
8
9     // a. Create initial list: C P E 1 0 1
10    insertAtHead(&head, '1');
11    insertAtHead(&head, '0');
12    insertAtHead(&head, '1');
13    insertAtHead(&head, 'E');
14    insertAtHead(&head, 'P');
15    insertAtHead(&head, 'C');
16
17    printf("a. Traverse the list:\n");
18    ListTraversal(head); // Output: CPE101
19
20    // b. Insert 'G' at the start of the list
21    insertAtHead(&head, 'G');
22    printf("b. Insert 'G' at the start of the list:\n");
23    ListTraversal(head); // Output: GCPE101
24
25    // c. Insert 'E' after node 'P'
26    struct Node* pNode = findNode(head, 'P');
27    if (pNode != NULL) {
28        insertAfter(pNode, 'E');
29    }
30    printf("c. Insert 'E' after node 'P':\n");
31    ListTraversal(head); // Output: GCPEE101
32
33    // d. Delete node containing 'C'
34    deleteNode(&head, 'C');
35    printf("d. Delete node containing 'C':\n");
36    ListTraversal(head); // Output: GPEE101
37
38    return 0;
39 }
```

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':
GP EE101
```

```
=== 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: GP EE101
97
98     // e. Delete node containing 'P'
99     deleteNode(&head, 'P');
100    printf("e. Delete node containing 'P':\n");
101    ListTraversal(head); // Output: GEE101
102
103    return 0;
104 }
```

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':
GP EE101
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: GP EE101

    // 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':
GP EE101
e. Delete node containing 'P':
GE E101
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:

```

9
0 void insertAfter(Node* prev_node, char new_data) {
1     if (prev_node == nullptr) {
2         std::cout << "Previous node cannot be NULL.\n";
3         return;
4     }
5     Node* new_node = new Node();
6     new_node->data = new_data;
7
8     new_node->next = prev_node->next; // new node's next is prev_node's
        next
9     new_node->prev = prev_node;      // new node's prev is prev_node
10
11     if (prev_node->next != nullptr)
12         prev_node->next->prev = new_node; // update next node's prev
13
14     prev_node->next = new_node;      // prev_node's next is new node
15 }

```

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:

```

0 void deleteNode(Node** head_ref, char key) {
1     Node* temp = *head_ref;
2
3     // Find the node to delete
4     while (temp != nullptr && temp->data != key) {
5         temp = temp->next;
6     }
7
8     if (temp == nullptr) // Not found
9         return;
10
11     // If node to be deleted is head
12     if (*head_ref == temp)
13         *head_ref = temp->next;
14
15     // Change next only if node to be deleted is NOT the last node
16     if (temp->next != nullptr)
17         temp->next->prev = temp->prev;
18
19     // Change prev only if node to be deleted is NOT the first node
20     if (temp->prev != nullptr)
21         temp->prev->next = temp->next;
22
23     delete temp;
24 }

```

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:

```

55 void ListTraversal(Node* node) {
56     while (node != nullptr) {
57         std::cout << node->data;
58         node = node->next;
59     }
60     std::cout << std::endl;
61 }
62

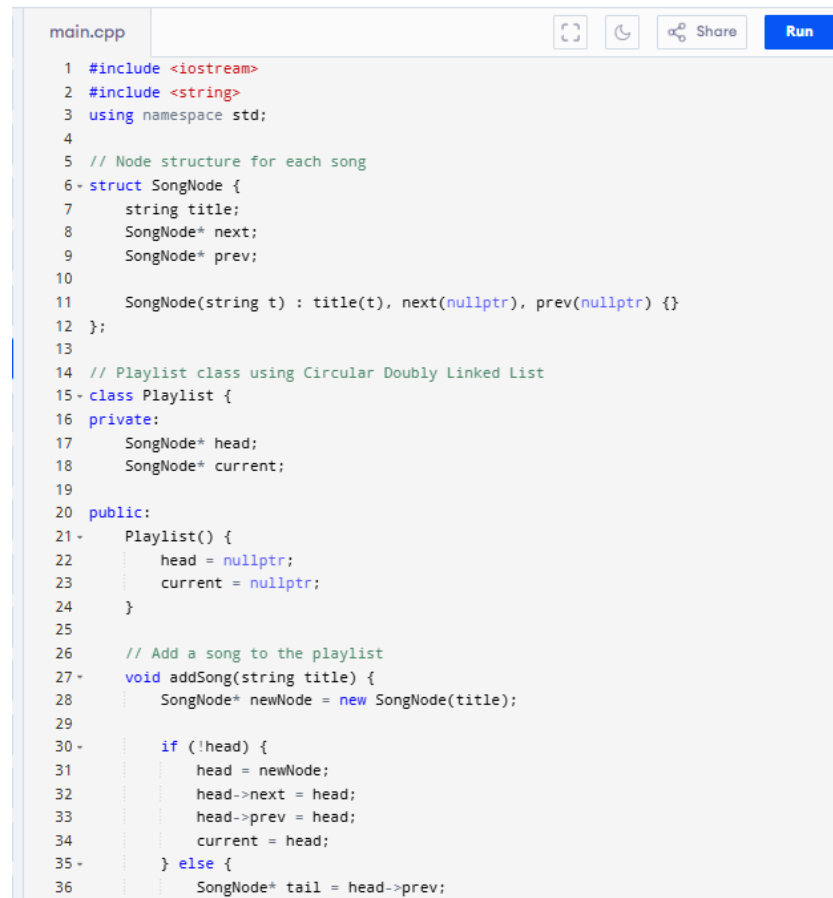
```

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
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 // Node structure for each song
6 struct SongNode {
7     string title;
8     SongNode* next;
9     SongNode* prev;
10
11     SongNode(string t) : title(t), next(nullptr), prev(nullptr) {}
12 };
13
14 // Playlist class using Circular Doubly Linked List
15 class Playlist {
16 private:
17     SongNode* head;
18     SongNode* current;
19
20 public:
21     Playlist() {
22         head = nullptr;
23         current = nullptr;
24     }
25
26     // Add a song to the playlist
27 void addSong(string title) {
28     SongNode* newNode = new SongNode(title);
29
30     if (!head) {
31         head = newNode;
32         head->next = head;
33         head->prev = head;
34         current = head;
35     } else {
36         SongNode* tail = head->prev;

```

main.cpp



Share

Run

```
35-     } else {
36-         SongNode* tail = head->prev;
37-         tail->next = newNode;
38-         newNode->prev = tail;
39-         newNode->next = head;
40-         head->prev = newNode;
41-     }
42-
43-     cout << "Added: " << title << endl;
44- }
45-
46- // Remove a song from the playlist
47- void removeSong(string title) {
48-     if (!head) {
49-         cout << "Playlist is empty." << endl;
50-         return;
51-     }
52-
53-     SongNode* temp = head;
54-     bool found = false;
55-
56-     do {
57-         if (temp->title == title) {
58-             found = true;
59-             break;
60-         }
61-         temp = temp->next;
62-     } while (temp != head);
63-
64-     if (!found) {
65-         cout << "Song '" << title << "' not found in the playlist." << endl;
66-         return;
67-     }
68-
69-     if (temp->next == temp) { // only one node
70-         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;
105     } while (temp->next != temp);

```

```

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:

Output

```
Added: Song A
Added: Song B
Added: Song C
Now playing: Song A
Now playing: Song B
Now playing: Song C
Now playing: Song B
Playing all songs in playlist:
Now playing: Song A
Now playing: Song B
Now playing: Song C
Removed: Song B
Playing all songs in playlist:
Now playing: Song A
Now playing: Song C
Added: Song D
Playing all songs in playlist:
Now playing: Song A
Now playing: Song C
Now playing: Song D

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

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