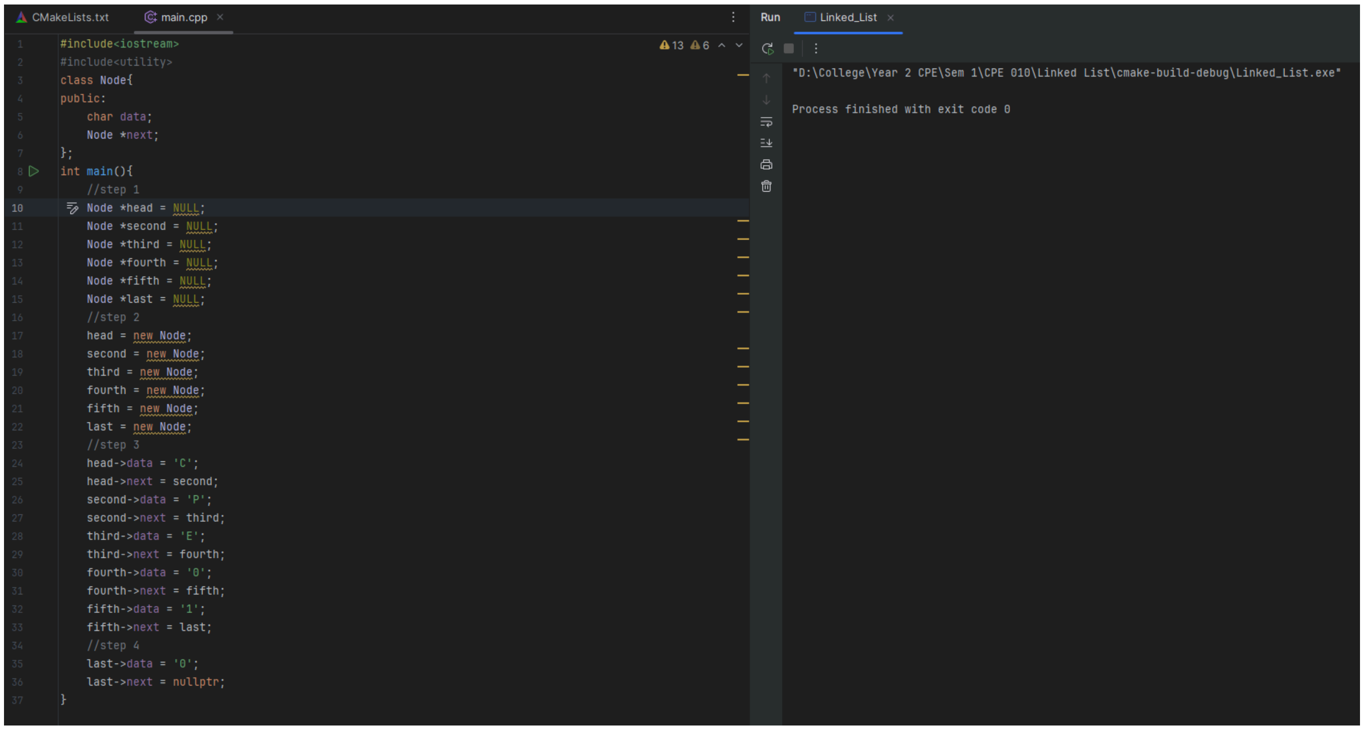
Hands-on Activity 3.1	
Linked Lists	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 08 - 14 - 25
Section: CPE21S4	**Date Submitted:** 08 - 14 - 25
Name(s): Santiago, David Owen A.	**Instructor:** Engr. Jimlord Quejado

6. Output

Table 3-1



```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;
}
```

```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Linked List\cmake-build-debug\Linked_List.exe"

Process finished with exit code 0
```

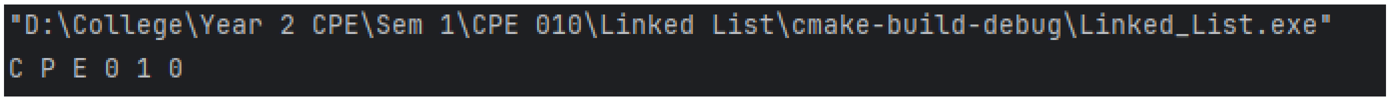
This is a simple implementation of linked lists. It begins by initializing a linked list by creating all of the nodes pointing toward null initially. After declaring the linked list, the second step involves allocating memory toward the individual nodes using the 'new' keyword. The final step is actually storing values in each node. After each node, it then points toward the next node in the linked list. Store value, point toward the head of the next, and continue.

Table 3.2

1. Traversing through list

```cpp
void printList(Node* head) {
   Node* curr = head;
   while (curr != NULL) {
       std::cout << curr->data << " ";
       curr = curr->next;
   }
   std::cout << std::endl;
}
```

Function usage:



```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Linked List\cmake-build-debug\Linked_List.exe"
C P E 0 1 0
```

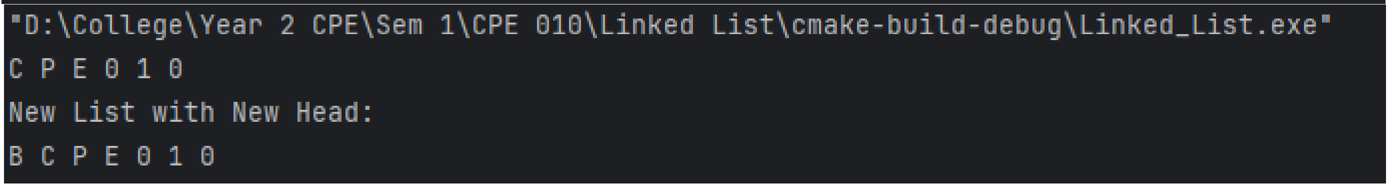
While the current node doesn't point toward null, the program will continuously print the value of the node. After printing, the current node is reassigned to the next node of the current node. This loop will continue until it reaches the end of the list or the tail.

2. Insert a node at the head:

```
void insertAtFront(Node **head, std::string new_data) { // Location of head, data to be inserted
   Node* new_node = new Node(); // create new node
   new_node->data = new_data;
   new_node->next = *head; // new node points to old head
   *head = new_node;       // head of list points to new node
}
```

Function usage:

```
head->printList(head);

head->insertAtFront(&head, "B");
std::cout << "New List with New Head:\n";
head->printList(head);
```



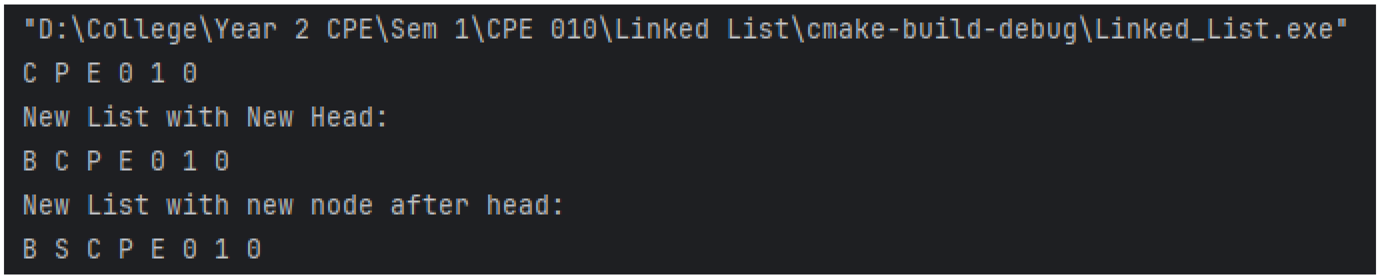
```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Linked List\cmake-build-debug\Linked_List.exe"
C P E 0 1 0
New List with New Head:
B C P E 0 1 0
```

In this section, I created a function that creates a new node and assigns a value toward it. It uses two parameters: a double pointer that points to the head to locate the node head as well as the data to be inserted. The pointer points toward the head pointer which points toward the value of the head in order to update the node directly.

3. Insert a node at any location between the head and tail:

```
void insertAfter(Node *prev, std::string new_data) {
   if(prev == NULL) {
       // check if prev is NULL
       std::cout << "Previous cannot be NULL\n";
       return;
   }
   Node* new_node = new Node(); // create new node
   new_node->data = new_data;
   new_node->next = prev->next; // connect a to b
   prev->next = new_node; // connect b to c
}
```

Usage:



```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Linked List\cmake-build-debug\Linked_List.exe"
C P E 0 1 0
New List with New Head:
B C P E 0 1 0
New List with new node after head:
B S C P E 0 1 0
```

This code first checks if the previous node is null, if it is, it will print an error message. Otherwise it will create a new node. The previous node will now point toward the new node, while the new node will point toward what the previous node was pointing towards. If A connects to C and new node B comes after A, then A will connect to B then B will connect to C.

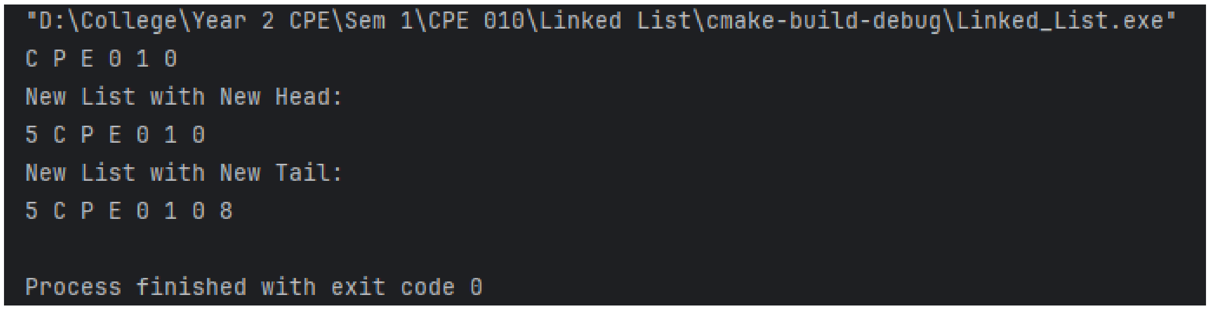
4. Insert a node at the end:

```
void insertAtBack(Node *&head, std::string new_data) {
   Node* new_node = new Node;
   new_node->data = new_data;
   new_node->next = NULL;
   if(head == NULL) {
       head = new_node;
       return;
   }
   Node *temp = head;
   while (temp->next != NULL) {
       temp = temp->next;
   }
   temp->next = new_node;
}
```

Usage:

```
head->insertAtBack(head, "8");
std::cout << "New List with New Tail:\n";
head->printList(head);
```

This program uses two parameters: a dereferencer for the head node and the new data to be inserted. It first creates a new node then inserts its data then points this new node toward null, signifying that it’s the last node. Next, it first checks if the list is empty or not, if it is, then the new node is considered the head. If the list isn’t empty, then it traverses toward the end by using a temporary node that points toward the head. While the head does not point toward null, it will continuously update the temp node to point toward the next node until it reaches the last node. Once at the last node, the next node will be updated to the previously created new node.



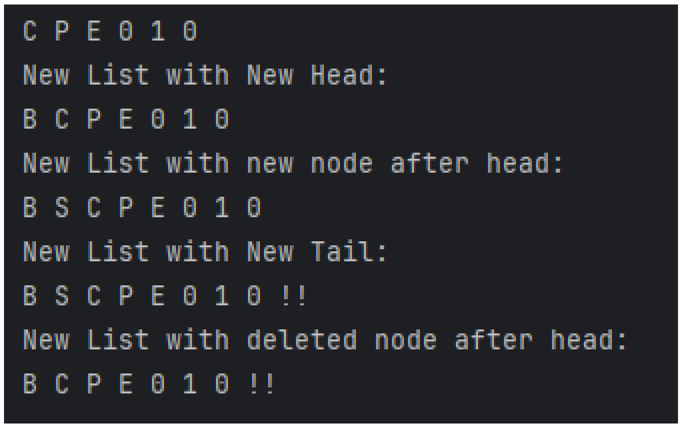
```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Linked List\cmake-build-debug\Linked_List.exe"
C P E 0 1 0
New List with New Head:
5 C P E 0 1 0
New List with New Tail:
5 C P E 0 1 0 8

Process finished with exit code 0
```

5. To delete a node from linked list:

```
void deleteAfter(Node *prev) {
   if (prev == NULL|| prev->next == NULL) {return;}
   Node *temp = prev->next; // stores node to be deleted
   prev->next = temp->next; // updates previous next node
   delete temp; //delete node
```

Usage:

```
head->deleteAfter(head);
std::cout << "New List with deleted node after head:\n";
head->printList(head);
```



```
C P E 0 1 0
New List with New Head:
B C P E 0 1 0
New List with new node after head:
B S C P E 0 1 0
New List with New Tail:
B S C P E 0 1 0 !!
New List with deleted node after head:
B C P E 0 1 0 !!
```

This function deletes the node next to the previous node. The argument it takes is the previous node, so that we can delete the node after it. It first checks if it's the last node by checking if it points to null, if it does, then there’s no node to delete because it’s at the last node. However, if it’s not the last node, then it will create a temporary node that stores the node of the node next to previous. It then updates the new next of the previous node to whatever the node after it points to. Then, the temp node is deleted.

Table 3-3

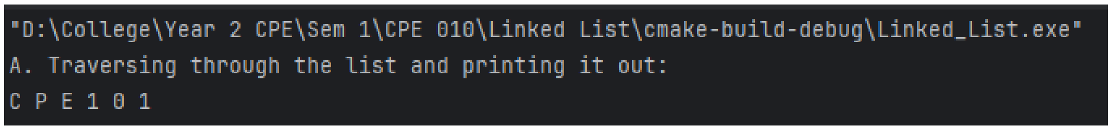
A

Source Code:

```
void printList(Node* head) {
   Node* curr = head;
   while (curr != NULL) {
       std::cout << curr->data << " ";
       curr = curr->next;
   }
   std::cout << std::endl;
}
```

```
std::cout<<"A. Traversing through the list and printing it out: \n";
head->printList(head);
```

Console:



```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Linked List\cmake-build-debug\Linked_List.exe"
A. Traversing through the list and printing it out:
C P E 1 0 1
```

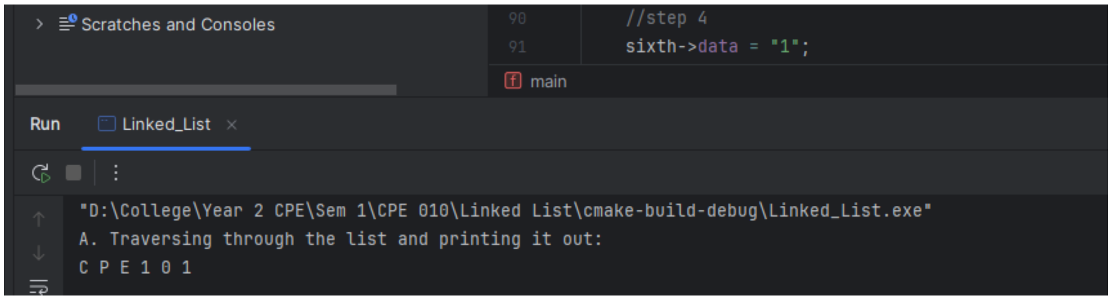
B

Source Code:

```
void insertAtFront(Node **head, std::string new_data) { // Location of
head, data to be inserted
   Node* new_node = new Node(); // create new node
   new_node->data = new_data;
   new_node->next = *head; // new node points to old head
   *head = new_node;       // head of list points to new node
}
```

```
head->insertAtFront(&head, "G");
std::cout << "\nB. Adding G to Insert the element \'G\' at the start
of the list to replace the current node. Output should now show
\"GCPE101\": "<< std::endl;
head->printList(head);
```

Console:



```
//step 4
sixth->data = "1";
```

```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Linked List\cmake-build-debug\Linked_List.exe"
A. Traversing through the list and printing it out:
C P E 1 0 1
```

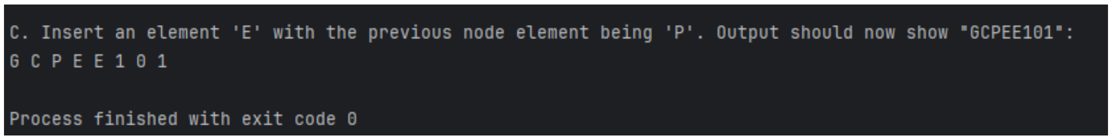
C

Source Code:

```
void insertAfter(Node *prev, std::string new_data) {
   if(prev == NULL) {
       // check if prev is NULL
       std::cout << "Previous cannot be NULL\n";
       return;
   }
   Node* new_node = new Node(); // create new node
   new_node->data = new_data;
   new_node->next = prev->next; // connect a to b
   prev->next = new_node; // connect b to c
}
```

```
head->insertAfter(second, "E");
std::cout << "\nC. Insert an element \'E\' with the previous node
element being \'P\'. Output should now show \"GCPEE101\": "<<
std::endl;
head->printList(head);
```

Console:



```
C. Insert an element 'E' with the previous node element being 'P'. Output should now show "GCPEE101":
G C P E E 1 0 1

Process finished with exit code 0
```

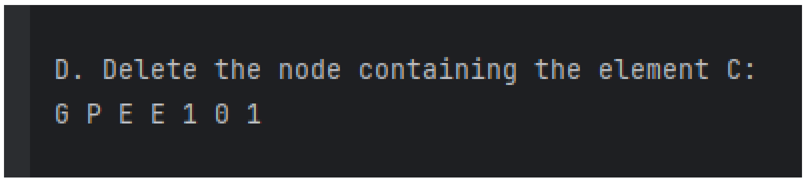
D

Source Code:

```
void deleteAfter(Node *prev) {
   if (prev == NULL|| prev->next == NULL) {return;}
   Node *temp = prev->next; // stores node to be deleted
   prev->next = temp->next; // updates previous next node
   delete temp; //delete ndoe
}
```

```
head->deleteAfter(head);
std::cout << "\nD. Delete the node containing the element C: "<<
std::endl;
head->printList(head);
```

Console:



```
D. Delete the node containing the element C:
G P E E 1 0 1
```

E

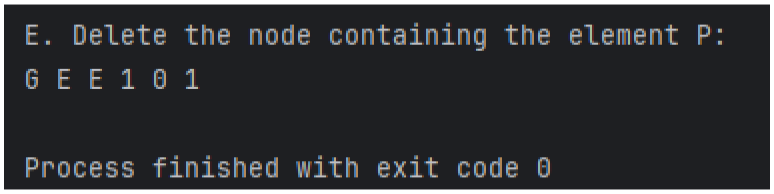
Source Code:

```
void deleteAfter(Node *prev) {
   if (prev == NULL|| prev->next == NULL) {return;}
   Node *temp = prev->next; // stores node to be deleted
```

```
    prev->next = temp->next; // updates previous next node
    delete temp; //delete ndoe
}
```

```
head->deleteAfter(head);
std::cout << "\nE. Delete the node containing the element P: "<<
std::endl;
head->printList(head);
```

Console:



```
E. Delete the node containing the element P:
G E E 1 0 1

Process finished with exit code 0
```

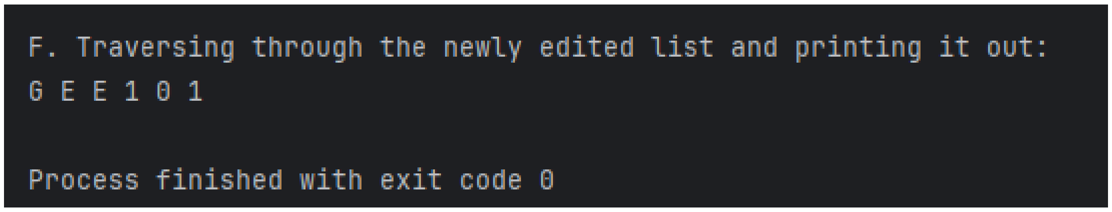
F

Source Code:

```
void printList(Node* head) {
    Node* curr = head;
    while (curr != NULL) {
        std::cout << curr->data << " ";
        curr = curr->next;
    }
    std::cout << std::endl;
}
```

```
std::cout<<"\nF. Traversing through the newly edited list and printing
it out: \n";
head->printList(head);
```

Console:



```
F. Traversing through the newly edited list and printing it out:
G E E 1 0 1

Process finished with exit code 0
```

Table 3-4. Modified Operations for Doubly Linked Lists

Screenshots	Analysis
See code below	Traversing the list can now be done in two ways: forwards or backwards. By entering two arguments (head, and tail), the program will be able to traverse through the linked list both from the head and to the tail. I simply declared two temporary nodes, one to serve as the head, then the other as the tail.

```
void printList(Node* head, Node* tail) {
    Node* curr = head;

    std::cout << "Head to Tail \n";
    while (curr != NULL) {
        std::cout << curr->data << " ";
        curr = curr->next;
    }
    std::cout << std::endl;

    Node* curr2 = tail;
    std::cout << "Tail to Head \n";
    while (curr2 != NULL) {
        std::cout << curr2->data << " ";
        curr2 = curr2->prev;
    }
    std::cout << std::endl;
}
```

```
void insertAtFront(Node **head, std::string new_data) { // Location of head, data to be inserted
    Node* new_node = new Node(); // create new node
    new_node->data = new_data;
    new_node->prev = NULL;
    new_node->next = *head; // new node points to old head
    *head = new_node;       // head of list points to new node
}
```

I added a redeclaration of the previous node for the new head so that it now points to null. Everything else stays the same.

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

    // link new node's next to prev's next
    new_node->next = prev->next;
    new_node->prev = prev; // new node's prev points to prev

    // if prev's next exists, update its prev to the new node
    if (prev->next != nullptr) {
        prev->next->prev = new_node;
    }
    // link prev's next to new node
    prev->next = new_node;
}
```

I modified the insertAfter function, for a doubly linked list, it adds a new node immediately after a given node (prev) while keeping both forward and backward links intact. It first checks if prev is valid, since insertion cannot happen after a nullptr. A new node is then created, storing the provided data, and its prev pointer is set to prev, while its next pointer is set to prev->next (the node that originally came after prev). If a node exists after prev, its prev pointer is updated to point back to the new node, maintaining the backward link. Then, prev->next is updated to point to the new node.

```
void insertAtBack(Node **head, std::string new_data) {
    Node* new_node = new Node(); // create new node
    new_node->data = new_data;
    new_node->next = NULL;

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

In this new insertAtBack function, it first creates a new node containing the argument data, setting its next pointer to NULL since it will be the last node.
If the list is empty, the new node becomes the head and its prev pointer is set to NULL.
Otherwise, the function traverses the list until it reaches the current last node, then updates that node’s next pointer to point to the new node and sets the new node’s prev pointer to link back to the former last node.

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

}
```

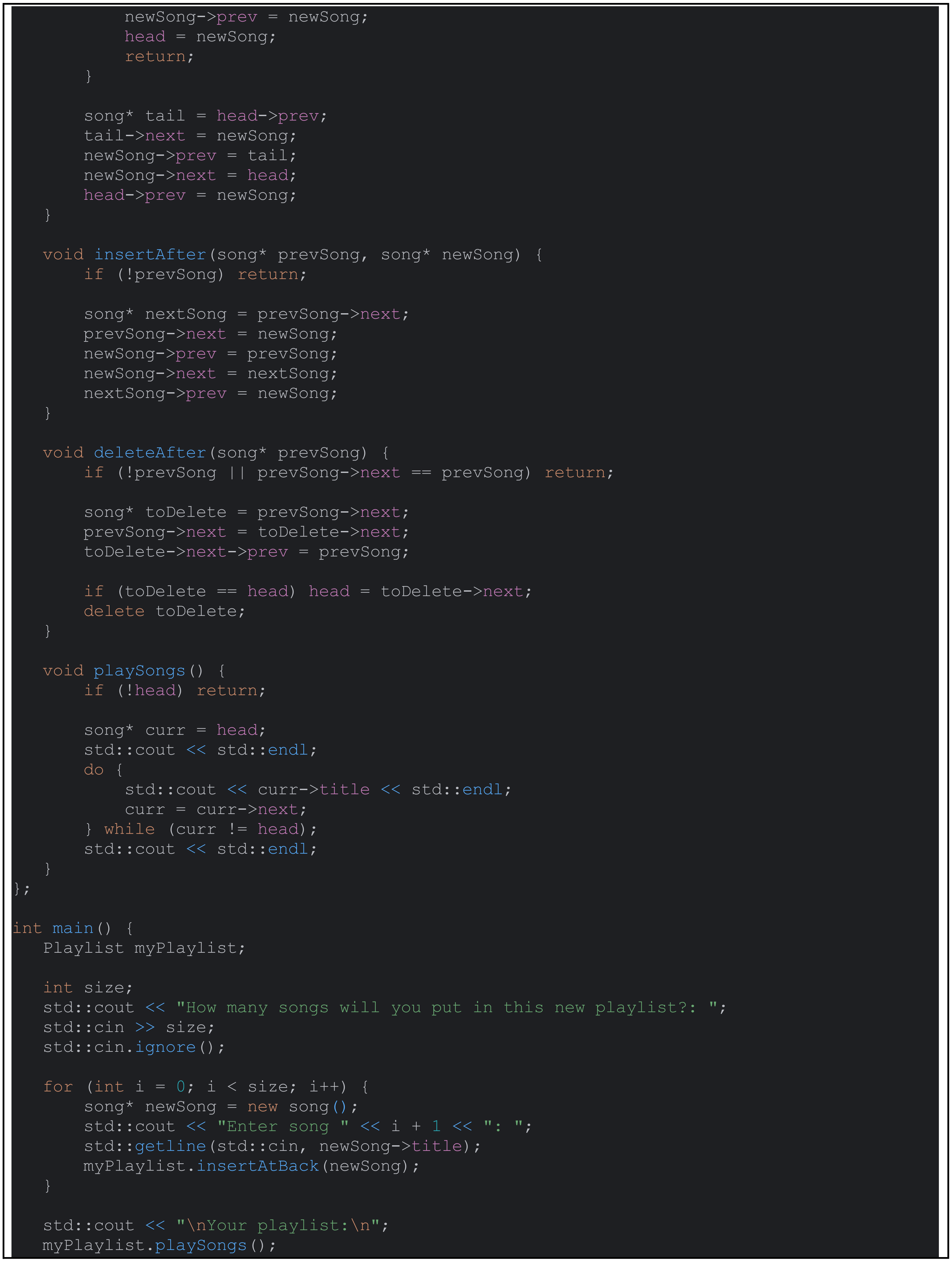
```
void deleteAfter(Node *prev) {
    if (prev == NULL|| prev->next == NULL)
{return;}
    Node *temp = prev->next; // stores node
to be deleted
    prev->next = temp->next; // updates
previous next node

    if (temp->next != NULL) {
        temp->next->prev = prev;
    }
    delete temp; //delete ndoe
}
```

This new function first checks if there is a node to delete. Then it saves the node to be deleted, updates the next pointer of the previous node to skip the deleted node, and if there's a node after the deleted one, it updates that node's prev pointer to point back to the previous node. Afterwhich, it deletes the unwanted node.

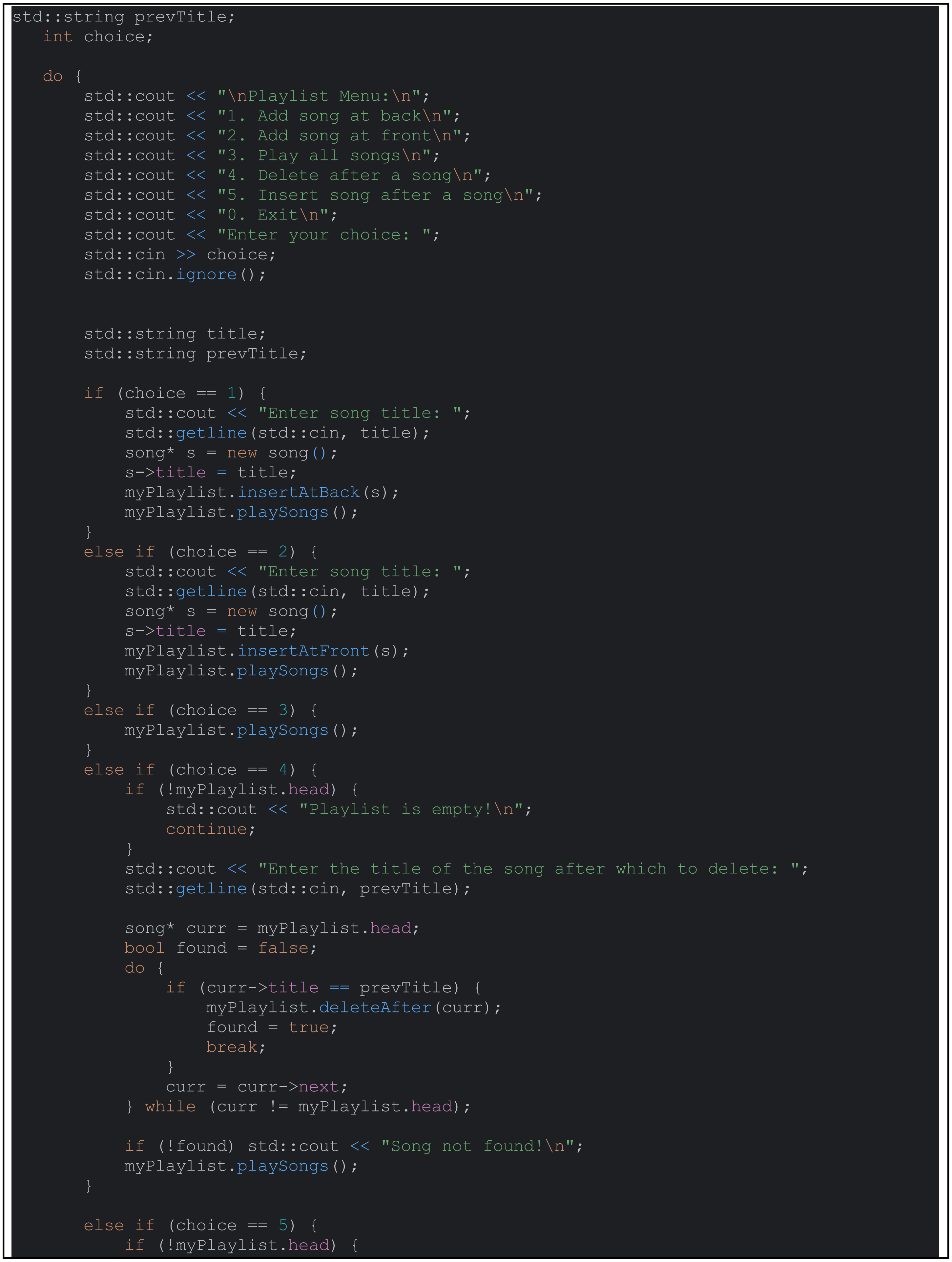
NOTE: I used several references such as Geeks for Geeks, CodeBeauty (YouTube), and W3Schools

7. Supplementary Activity

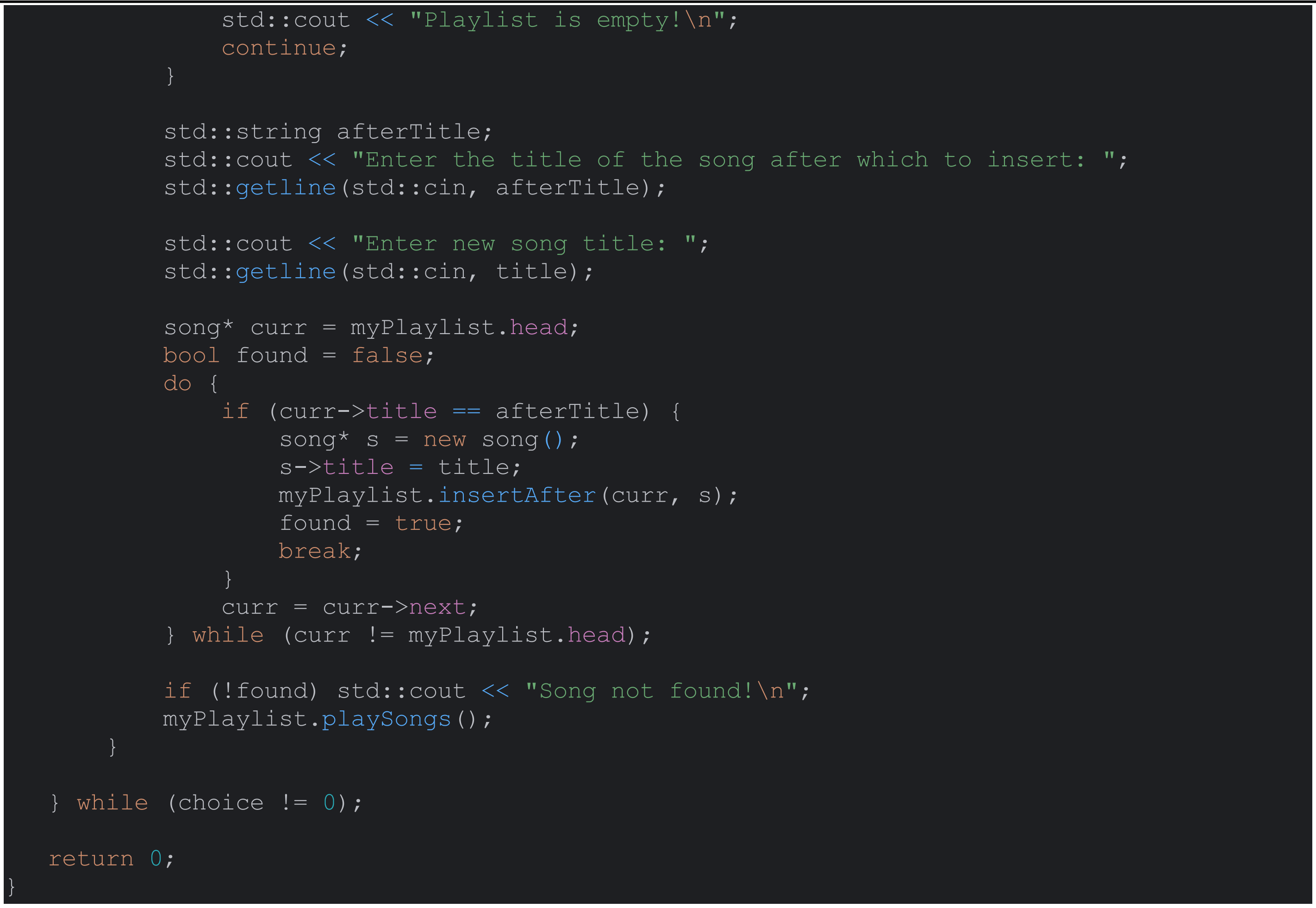
```
#include <iostream>
#include <string>

class song {
public:
   std::string title;
   song* prev;
   song* next;
};

class Playlist {
public:
   song* head = nullptr;

   void insertAtFront(song* newSong) {
       if (!head) {
           newSong->next = newSong;
           newSong->prev = newSong;
           head = newSong;
           return;
       }

       song* tail = head->prev;
       newSong->next = head;
       newSong->prev = tail;
       tail->next = newSong;
       head->prev = newSong;
       head = newSong;
   }

   void insertAtBack(song* newSong) {
       if (!head) {
           newSong->next = newSong;
```



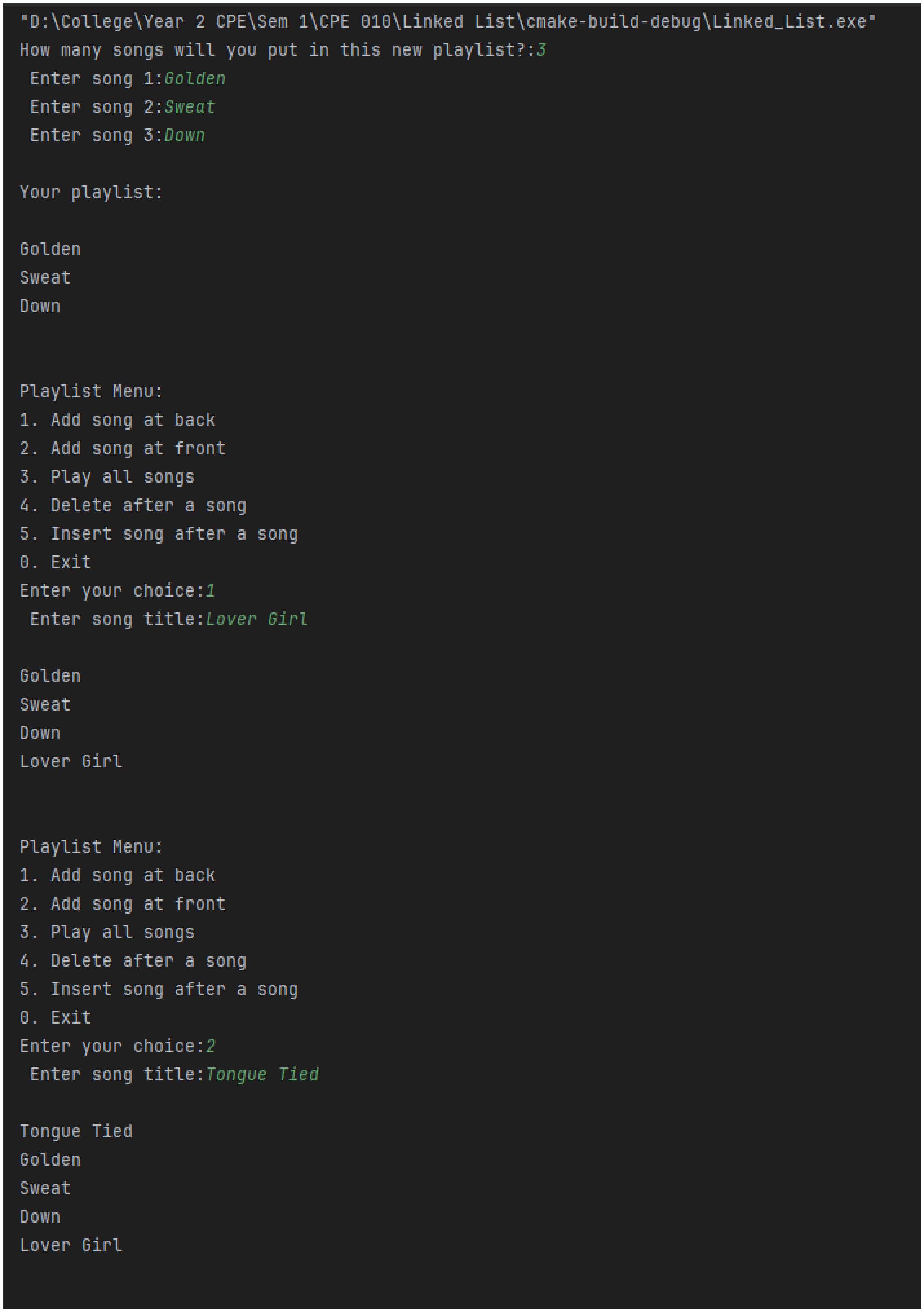
```
            newSong->prev = newSong;
            head = newSong;
            return;
        }

        song* tail = head->prev;
        tail->next = newSong;
        newSong->prev = tail;
        newSong->next = head;
        head->prev = newSong;
    }

    void insertAfter(song* prevSong, song* newSong) {
        if (!prevSong) return;

        song* nextSong = prevSong->next;
        prevSong->next = newSong;
        newSong->prev = prevSong;
        newSong->next = nextSong;
        nextSong->prev = newSong;
    }

    void deleteAfter(song* prevSong) {
        if (!prevSong || prevSong->next == prevSong) return;

        song* toDelete = prevSong->next;
        prevSong->next = toDelete->next;
        toDelete->next->prev = prevSong;

        if (toDelete == head) head = toDelete->next;
        delete toDelete;
    }

    void playSongs() {
        if (!head) return;

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

int main() {
    Playlist myPlaylist;

    int size;
    std::cout << "How many songs will you put in this new playlist?: ";
    std::cin >> size;
    std::cin.ignore();

    for (int i = 0; i < size; i++) {
        song* newSong = new song();
        std::cout << "Enter song " << i + 1 << ": ";
        std::getline(std::cin, newSong->title);
        myPlaylist.insertAtBack(newSong);
    }

    std::cout << "\nYour playlist:\n";
    myPlaylist.playSongs();
```



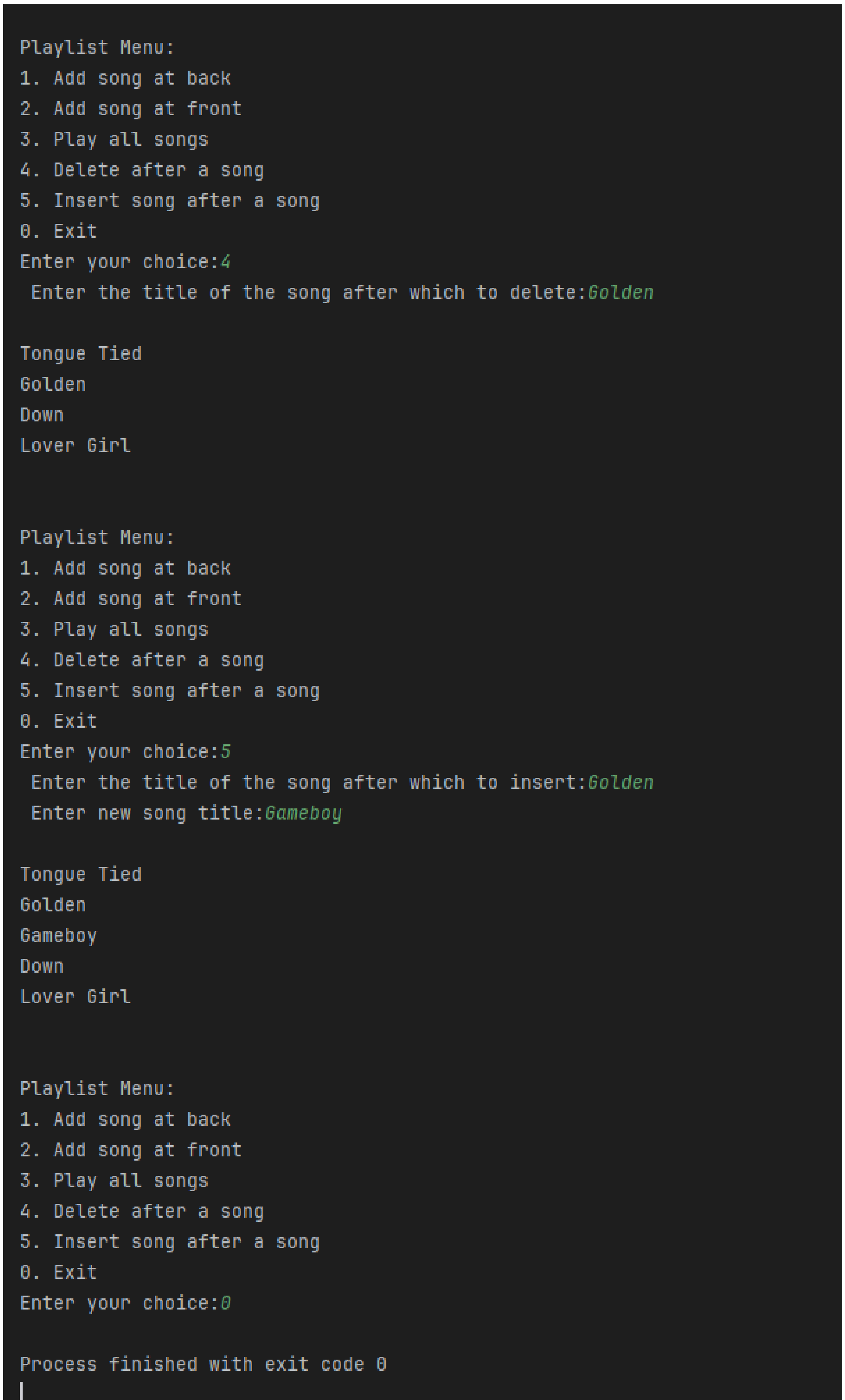
```cpp
std::string prevTitle;
   int choice;

   do {
       std::cout << "\nPlaylist Menu:\n";
       std::cout << "1. Add song at back\n";
       std::cout << "2. Add song at front\n";
       std::cout << "3. Play all songs\n";
       std::cout << "4. Delete after a song\n";
       std::cout << "5. Insert song after a song\n";
       std::cout << "0. Exit\n";
       std::cout << "Enter your choice: ";
       std::cin >> choice;
       std::cin.ignore();


       std::string title;
       std::string prevTitle;

       if (choice == 1) {
           std::cout << "Enter song title: ";
           std::getline(std::cin, title);
           song* s = new song();
           s->title = title;
           myPlaylist.insertAtBack(s);
           myPlaylist.playSongs();
       }
       else if (choice == 2) {
           std::cout << "Enter song title: ";
           std::getline(std::cin, title);
           song* s = new song();
           s->title = title;
           myPlaylist.insertAtFront(s);
           myPlaylist.playSongs();
       }
       else if (choice == 3) {
           myPlaylist.playSongs();
       }
       else if (choice == 4) {
           if (!myPlaylist.head) {
               std::cout << "Playlist is empty!\n";
               continue;
           }
           std::cout << "Enter the title of the song after which to delete: ";
           std::getline(std::cin, prevTitle);

           song* curr = myPlaylist.head;
           bool found = false;
           do {
               if (curr->title == prevTitle) {
                   myPlaylist.deleteAfter(curr);
                   found = true;
                   break;
               }
               curr = curr->next;
           } while (curr != myPlaylist.head);

           if (!found) std::cout << "Song not found!\n";
           myPlaylist.playSongs();
       }

       else if (choice == 5) {
           if (!myPlaylist.head) {
```



```cpp
                std::cout << "Playlist is empty!\n";
                continue;
            }

            std::string afterTitle;
            std::cout << "Enter the title of the song after which to insert: ";
            std::getline(std::cin, afterTitle);

            std::cout << "Enter new song title: ";
            std::getline(std::cin, title);

            song* curr = myPlaylist.head;
            bool found = false;
            do {
                if (curr->title == afterTitle) {
                    song* s = new song();
                    s->title = title;
                    myPlaylist.insertAfter(curr, s);
                    found = true;
                    break;
                }
                curr = curr->next;
            } while (curr != myPlaylist.head);

            if (!found) std::cout << "Song not found!\n";
            myPlaylist.playSongs();
        }

    } while (choice != 0);

    return 0;
}
```



```
"D:\College\Year 2 CPE\Sem 1\CPE 010\Linked List\cmake-build-debug\Linked_List.exe"
How many songs will you put in this new playlist?:3
 Enter song 1:Golden
 Enter song 2:Sweat
 Enter song 3:Down

Your playlist:

Golden
Sweat
Down


Playlist Menu:
1. Add song at back
2. Add song at front
3. Play all songs
4. Delete after a song
5. Insert song after a song
0. Exit
Enter your choice:1
 Enter song title:Lover Girl

Golden
Sweat
Down
Lover Girl


Playlist Menu:
1. Add song at back
2. Add song at front
3. Play all songs
4. Delete after a song
5. Insert song after a song
0. Exit
Enter your choice:2
 Enter song title:Tongue Tied

Tongue Tied
Golden
Sweat
Down
Lover Girl
```



```
Playlist Menu:
1. Add song at back
2. Add song at front
3. Play all songs
4. Delete after a song
5. Insert song after a song
0. Exit
Enter your choice:4
 Enter the title of the song after which to delete:Golden

Tongue Tied
Golden
Down
Lover Girl


Playlist Menu:
1. Add song at back
2. Add song at front
3. Play all songs
4. Delete after a song
5. Insert song after a song
0. Exit
Enter your choice:5
 Enter the title of the song after which to insert:Golden
 Enter new song title:Gameboy

Tongue Tied
Golden
Gameboy
Down
Lover Girl


Playlist Menu:
1. Add song at back
2. Add song at front
3. Play all songs
4. Delete after a song
5. Insert song after a song
0. Exit
Enter your choice:0

Process finished with exit code 0
```

Short Analysis:

This program creates and manages a music playlist using a circular doubly linked list with two classes and designated functions. Each song is stored as a node with a title and links to the previous and next songs, allowing the list to loop endlessly. The Playlist class adds songs to the front or back, inserts one song after another, deletes a song, and plays all the songs in order. In the main program, you first enter your initial set of songs, and then I implemented a menu that lets you keep adding, inserting, deleting, or playing songs continuously using a simple do-while. Because the list is circular, we can traverse through the entire list endlessly unless the program is terminated.

8. Conclusion

NOTE: I used several references such as Geeks for Geeks, CodeBeauty (YouTube), and W3Schools

In conclusion, this is the most challenging activity in my C++ journey by far. The use of multiple pointers, classes, and functions have proven to be confusing and difficult to implement with the amount of parameters and factors to consider. I had to utilize countless resources like YouTube, GeeksForGeeks, and Reddit to be able to structure my code properly. It was challenging and time-consuming (it took me five hours to complete both the procedure and supplementary activities). Regardless of the difficulty, I still managed to learn and improve my skills in programming with C++. I now know how to apply the logic of linked lists in proper code syntax as well as the implementation of class methods and such.

9. Assessment Rubric