

Activity No. 3	
LINKED LISTS	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: September 27, 2024
Section: CPE21S4	Date Submitted: September 27, 2024
Name(s): Bona, Andrei Nycole So	Instructor: Prof. Maria Rizette Sayo

6. Output

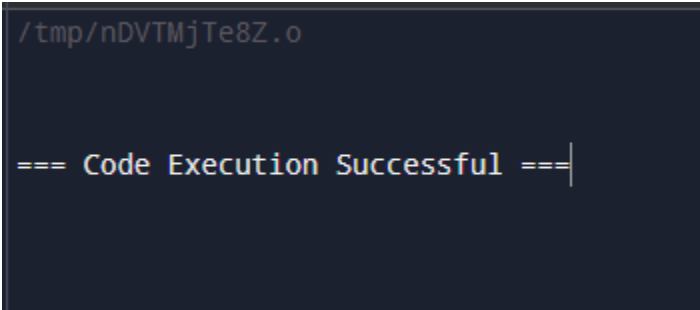
Screenshot	
Discussion	<p>The code constructs a linked structure within the memory by making six allocates and connecting all of these in a certain order. Each of the nodes is given a character and the final one is assigned nullptr to signify end of the list. In addition, it should be observed the output is not seen since there is no code that traverses or prints the list so the structure remains hidden during the run. To improve the program include adding a functionality that will traverse the list and display the data for each node so that the linked list will be able to be seen.</p>

Table 3-1. Output of Initial/Simple Implementation

Operation	Screenshot
Traversal	 <pre>void traversal(Node *n) { while (n != nullptr) { cout << n->data << " -> "; n = n->next; } cout << "nullptr" << endl; }</pre> <p>output: Initial list: C -> P -> E -> 0 -> 1 -> 0 -> nullptr</p>
Insertion at head	 <pre>void insertAtHead(Node *&head, char newData) { Node *newNode = new Node(); newNode->data = newData; newNode->next = head; head = newNode; }</pre>

	<p>output:</p> <pre>After inserting 'X' at head: X -> C -> P -> E -> 0 -> 1 -> 0 -> nullptr</pre>
Insertion at any part of the list	<pre>void insertAfter(Node *previousNode, char newData) { if (previousNode == nullptr) { cout << "Error." << endl; return; } Node *newNode = new Node(); newNode->data = newData; newNode->next = previousNode->next; previousNode->next = newNode; }</pre> <p>output:</p> <pre>After inserting 'X' after the second node: X -> C -> X -> P -> E -> 0 -> 1 -> 0 -> X -> nullptr</pre>
Insertion at the end	<pre>void insertAtEnd(Node *&head, char newData) { Node *newNode = new Node(); newNode->data = newData; newNode->next = nullptr; if (head == nullptr) { head = newNode; return; } Node *last = head; while (last->next != nullptr) { last = last->next; } last->next = newNode; }</pre> <p>output:</p> <pre>After inserting 'X' at end: X -> C -> P -> E -> 0 -> 1 -> 0 -> X -> nullptr</pre>

Deletion of a node	<pre> void deleteNode(Node *&head, char key) { Node *temp = head; Node *prev = nullptr; if (temp != nullptr && temp->data == key) { head = temp->next; delete temp; return; } while (temp != nullptr && temp->data != key) { prev = temp; temp = temp->next; } if (temp == nullptr) return; prev->next = temp->next; delete temp; } </pre> <p>output:</p> <p>After deleting '1': X -> C -> X -> P -> E -> 0 -> 0 -> X -> nullptr</p>
--------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 3-2. Code for the List Operations

a.	Source Code	<pre> cout << "Initial list: "; traversal(head);_ </pre>
	Console	Initial list: C -> P -> E -> 1 -> 0 -> 1 -> nullptr
b.	Source Code	<pre> insertAtHead(head, 'G'); cout << "After inserting 'G' at head: "; traversal(head); </pre>
	Console	After inserting 'G' at head: G -> C -> P -> E -> 1 -> 0 -> 1 -> nullptr

c.	Source Code	<pre>Node *current = head; while (current != nullptr && current->data != 'P') { current = current->next; } if (current != nullptr) { insertAfter(current, 'E'); } cout << "After inserting 'E' after 'P': "; traversal(head);</pre>
	Console	After inserting 'E' after 'P': G -> C -> P -> E -> E -> 1 -> 0 -> 1 -> nullptr
d.	Source Code	<pre>deleteNode(head, 'C'); cout << "After deleting 'C': "; traversal(head);</pre>
	Console	After deleting 'C': G -> P -> E -> E -> 1 -> 0 -> 1 -> nullptr
e.	Source Code	<pre>deleteNode(head, 'P'); cout << "After deleting 'P': "; traversal(head);</pre>
	Console	After deleting 'P': G -> E -> E -> 1 -> 0 -> 1 -> nullptr
f.	Source Code	<pre>cout << "Final list: "; traversal(head);</pre>
	Console	Final list: G -> E -> E -> 1 -> 0 -> 1 -> nullptr

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

Screenshot(s)	Analysis
---------------	----------

```

void traverseForward(Node *head) {
    Node *current = head;
    while (current != nullptr) {
        cout << current->data << " <-> ";
        current = current->next;
    }
    cout << "nullptr" << endl;
}

void traverseBackward(Node *tail) {
    Node *current = tail;
    while (current != nullptr) {
        cout << current->data << " <-> ";
        current = current->prev;
    }
    cout << "nullptr" << endl;
}

```

To traverse a doubly linked list, you can move both forward and backward.

```

void insertAtHead(Node *&head, char newData)
{
    Node *newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    newNode->prev = nullptr;

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

```

To insert a node at the head of a doubly linked list, you must set the previous pointer of the original head node.

```
void insertAtEnd(Node *&head, char newData)
```

```
{  
    Node *newNode = new Node();  
    newNode->data = newData;  
    newNode->next = nullptr;  
  
    if (head == nullptr) {  
        newNode->prev = nullptr;  
        head = newNode;  
        return;  
    }  
  
    Node *last = head;  
    while (last->next != nullptr) {  
        last = last->next;  
    }  
  
    last->next = newNode;  
    newNode->prev = last;  
}
```

Insert a new node at the end of the doubly linked list and update the pointers accordingly.

```
void insertAfter(Node *prevNode, char newData) {  
    if (prevNode == nullptr) {  
        cout << "Previous node cannot be null." << endl;  
        return;  
    }  
  
    Node *newNode = new Node();  
    newNode->data = newData;  
    newNode->next = prevNode->next;  
    newNode->prev = prevNode;  
  
    if (prevNode->next != nullptr) {  
        prevNode->next->prev = newNode;  
    }  
    prevNode->next = newNode;  
}
```

To insert a node after a given node, adjust the next and previous pointers accordingly.

```

void deleteNode(Node *&head, Node *delNode) {
    if (head == nullptr || delNode == nullptr) return;

    if (head == delNode) {
        head = delNode->next;
    }

    if (delNode->next != nullptr) {
        delNode->next->prev = delNode->prev;
    }

    if (delNode->prev != nullptr) {
        delNode->prev->next = delNode->next;
    }

    delete delNode;
}

```

To delete a node, update the pointers of the adjacent nodes.

Table 3-4. Modified Operations for Doubly Linked Lists

7. Supplementary Activity

ILO B: Solve given problems utilizing linked lists in C++

Problem Title: Implementing a Song Playlist using Linked List

Source: Packt Publishing

```
#include <iostream>
```

```
using namespace std;
```

```

class Node
{
public:
    string songName;
    Node *next;

    Node(string name) : songName(name), next(nullptr) {}
};

```

```

class CircularLinkedList
{
private:
    Node *tail;

public:
    CircularLinkedList() : tail(nullptr) {}

    void addSong(const string &songName)
    {
        Node *newNode = new Node(songName);
        if (!tail)

```

```

{
    tail = newNode;
    tail->next = tail;
} else
{
    newNode->next = tail->next;
    tail->next = newNode;
    tail = newNode;
}
}

```

```

void removeSong(const string &songName)

```

```

{
    if (!tail) return;

    Node *current = tail->next;
    Node *prev = tail;

    do {
        if (current->songName == songName)
        {
            if (current == tail)
            {
                if (current->next == current)
                {
                    delete current;
                    tail = nullptr;
                } else
                {
                    prev->next = current->next;
                    tail = prev;
                    delete current;
                }
            } else
            {
                prev->next = current->next;
                delete current;
            }
            return;
        }
        prev = current;
        current = current->next;
    } while (current != tail->next);
}

```

```

void playAllSongs()

```



```

{
    if (!tail)
    {
        cout << "Playlist is empty." << endl;
        return;
    }

    Node *current = tail->next;
    do {
        cout << "Playing: " << current->songName << endl;
        current = current->next;
    } while (current != tail->next);
}

void displayPlaylist()
{
    if (!tail)
    {
        cout << "Playlist is empty." << endl;
        return;
    }

    Node *current = tail->next;
    do {
        cout << current->songName << " -> ";
        current = current->next;
    } while (current != tail->next);
    cout << "(back to " << tail->next->songName << ")" << endl;
}
};

int main()
{
    CircularLinkedList playlist;

    playlist.addSong("Song 1");
    playlist.addSong("Song 2");
    playlist.addSong("Song 3");
    playlist.addSong("Song 4");

    cout << "Current Playlist: ";
    playlist.displayPlaylist();

    cout << "\nPlaying all songs:" << endl;
    playlist.playAllSongs();

    playlist.removeSong("Song 3");
    cout << "\nPlaylist after removing 'Song 3': ";

```

```

playlist.displayPlaylist();

cout << "\nPlaying all songs after removal:" << endl;
playlist.playAllSongs();

playlist.removeSong("Song 1");
playlist.removeSong("Song 2");
playlist.removeSong("Song 4");

cout << "\nPlaylist after removing all songs: ";
playlist.displayPlaylist();

return 0;
}

```

```

/tmp/PZUC0sML2C.o
Current Playlist: Song 1 -> Song 2 -> Song 3 -> Song 4 -> (back to Song 1)

Playing all songs:
Playing: Song 1
Playing: Song 2
Playing: Song 3
Playing: Song 4

Playlist after removing 'Song 3': Song 1 -> Song 2 -> Song 4 -> (back to Song 1)

Playing all songs after removal:
Playing: Song 1
Playing: Song 2
Playing: Song 4

Playlist after removing all songs: Playlist is empty.

=== Code Execution Successful ===

```

8. Conclusion

This laboratory aimed to learn about linked lists and the advantages of using linked lists for data organization. After we had all been learned on a simple single linked list, we did insertion, deletion, and traversal, and then came up with a doubly linked list for particular uses.

The supplementary activity focused on creating a music playlist with a heavy emphasis placed on the user's perspective and light on the back end with the ability to add, delete, and play music. To sum up, this exercise deepened our understanding of linked lists and enhanced our programming abilities, showing how it is valuable in data struct and algorithms

I acknowledge that my knowledge about linked lists is still very basic because there are still methods and techniques of working with linked lists that are quite advanced which I have not mastered yet. I use different resources online to answer this laboratory and to improve my understanding and programming capability.

9. Assessment Rubric