

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: 9/27/2024
Section: CPE21s4	Date Submitted: 9/27/2024
Name(s): Alexzander J. Reyes	Instructor: Ma'am Sayo

6. Output

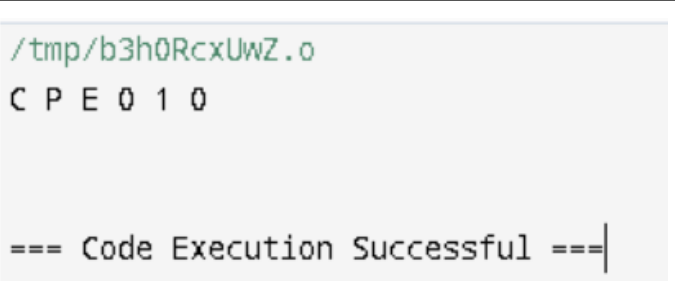
SCREENSHOT	 <pre> /tmp/b3h0RcxUwZ.o C P E 0 1 0 === Code Execution Successful === </pre>
DISCUSSION	<p>Although the code generates a linked list successfully, it might be made better by adding functions for insertion, deletion, and modification, handling errors, and dynamic allocation. For flexibility, iterators and templates might also be taken into account. The code provided produced no output, however after some adjustments, the output is displayed.</p>

Table 3-1. Output of Initial/Simple Implementation

Operation	Screenshot
Traversal	 <pre> // Function to traverse the linked list void traverse(Node* head) { Node* current = head; while (current != nullptr) { cout << current->data << " -> "; current = current->next; } cout << "NULL" << endl; } </pre>
Insertion at head	 <pre> 1 // Function to insert a node at the head of the list 2 void insertAtHead(Node*& head, char data) { 3 Node* newNode = new Node(); 4 newNode->data = data; 5 newNode->next = head; 6 head = newNode; 7 } </pre>

Insertion at any part of the list	<pre> 1 // Function to insert a node after a specific node 2 void insertAfter(Node* prevNode, char data) { 3 if (prevNode == nullptr) { 4 cout << "Previous node cannot be null." << endl; 5 return; 6 } 7 Node* newNode = new Node(); 8 newNode->data = data; 9 newNode->next = prevNode->next; 10 prevNode->next = newNode; 11 } </pre>
Insertion at the end	<pre> 1 // Function to insert a node at the end of the list 2 void insertAtEnd(Node*& head, char data) { 3 Node* newNode = new Node(); 4 newNode->data = data; 5 newNode->next = nullptr; 6 7 if (head == nullptr) { 8 head = newNode; 9 return; 10 } 11 12 Node* last = head; 13 while (last->next != nullptr) { 14 last = last->next; 15 } 16 last->next = newNode; 17 } </pre>
Deletion of a node	<pre> 1 // Function to delete a node with a specific key 2 void deleteNode(Node*& head, char key) { 3 Node* temp = head; 4 Node* prev = nullptr; 5 6 if (temp != nullptr && temp->data == key) { 7 head = temp->next; 8 delete temp; 9 return; 10 } 11 12 while (temp != nullptr && temp->data != key) { 13 prev = temp; 14 temp = temp->next; 15 } 16 17 if (temp == nullptr) return; 18 19 prev->next = temp->next; 20 delete temp; 21 } </pre>

Table 3-2. Code for the List Operations

A.	Source Code	<pre> 1 // Function to traverse the linked list 2 void traverse(Node* head) { 3 Node* current = head; 4 while (current != nullptr) { 5 cout << current->data << " -> "; 6 current = current->next; 7 } 8 cout << "NULL" << endl; 9 }</pre>
	Console	Initial list: C -> P -> E -> 0 -> 1 -> 0 -> NULL
B.	Source Code	<pre> 1 // Function to insert a node at the head of the list 2 void insertAtHead(Node*& head, char data) { 3 Node* newNode = new Node(); 4 newNode->data = data; 5 newNode->next = head; 6 head = newNode; 7 } 8</pre>
	Console	After inserting 'G' at head: G -> C -> P -> E -> 0 -> 1 -> 0 -> NULL
C.	Source Code	<pre> 1 // Function to insert a node after a specific node 2 void insertAfter(Node* prevNode, char data) { 3 if (prevNode == nullptr) { 4 cout << "Previous node cannot be null." << endl; 5 return; 6 } 7 Node* newNode = new Node(); 8 newNode->data = data; 9 newNode->next = prevNode->next; 10 prevNode->next = newNode; 11 }</pre>
	Console	After inserting 'E' after 'P': G -> C -> P -> E -> E -> 0 -> 1 -> 0 -> NULL
D.	Source Code	<pre> 1 // Function to delete a node with a specific key 2 void deleteNode(Node*& head, char key) { 3 Node* temp = head; 4 Node* prev = nullptr; 5 6 if (temp != nullptr && temp->data == key) { 7 head = temp->next; 8 delete temp; 9 return; 10 } 11 12 while (temp != nullptr && temp->data != key) { 13 prev = temp; 14 temp = temp->next; 15 } 16 17 if (temp == nullptr) return; 18 19 prev->next = temp->next; 20 delete temp; 21 }</pre>
	Console	After deleting 'C': G -> P -> E -> E -> 0 -> 1 -> 0 -> NULL

E.	Source Code	<pre> 1 // Call to delete node containing 'P' 2 deleteNode(head, 'P'); 3 </pre>
	Console	After deleting 'P': G -> E -> E -> 0 -> 1 -> 0 -> NULL
F.	Source Code	<pre> 1 cout << "Final list: ";
traverse(head); </pre>
	Console	Final list: G -> E -> E -> 0 -> 1 -> 0 -> NULL

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

Screenshot(s)	Analysis
<pre> // A. Function to traverse the doubly linked list void traverse(Node* head) { Node* current = head; while (current != nullptr) { cout << current->data << " <-> "; current = current->next; } cout << "NULL" << endl; } </pre>	This function correctly traverses the list and prints each node's data, confirming the links in both directions.
<pre> // B. Function to insert a node at the head of the list void insertAtHead(Node*& head, char data) { Node* newNode = new Node(); newNode->data = data; newNode->next = head; newNode->prev = nullptr; if (head != nullptr) { head->prev = newNode; } head = newNode; } </pre>	The head is updated correctly, and the previous head node's prev pointer is set to the new node.

```
// C. Function to insert a node after a specific node
void insertAfter(Node* prevNode, char data) {
    if (prevNode == nullptr) {
        cout << "Previous node cannot be null." << endl;
        return;
    }
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = prevNode->next;
    newNode->prev = prevNode;

    if (prevNode->next != nullptr) {
        prevNode->next->prev = newNode;
    }
    prevNode->next = newNode;
}
```

The new node is inserted correctly, maintaining the integrity of both **next** and **prev** pointers.

```
// D. Function to delete a node from the doubly linked list
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;
}
```

```
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;
}'''
```

```

// Step 1: Insert nodes into the list
insertAtHead(head, 'C');
insertAtHead(head, 'P');
insertAtHead(head, 'E');
insertAtHead(head, '0');
insertAtHead(head, '1');
insertAtHead(head, '0');

// Step 2: Traverse the list
cout << "Initial list: ";
traverse(head);

// Step 3: Insert 'G' at the head
insertAtHead(head, 'G');
cout << "After inserting 'G' at head: ";
traverse(head);

// Step 4: Insert 'E' after 'P'
Node* temp = head->next->next; // Node 'P'
insertAfter(temp, 'E');
cout << "After inserting 'E' after 'P': ";
traverse(head);

// Step 5: Delete the node containing 'C'
deleteNode(head, head->next->next->next); // Deletes 'C'
cout << "After deleting 'C': ";
traverse(head);

// Step 6: Delete the node containing 'P'
deleteNode(head, head->next->next); // Deletes 'P'
cout << "After deleting 'P': ";
traverse(head);

// Step 7: Final list
cout << "Final list: ";
traverse(head);

return 0;
}

```

The final list confirms that all operations have been executed correctly, reflecting the updated structure.

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 Playlist {
private:
    Node* last;
public:
    Playlist() {
        last = nullptr;
    }
    void addSong(string song) {
        Node* newNode = new Node(song);

        if (last == nullptr) {
            last = newNode;
            last->next = last;
        } else {
            newNode->next = last->next;
            last->next = newNode;
            last = newNode;
        }
        cout << song << " has been added to the playlist.\n";
    }
    void removeSong(string song) {
        if (last == nullptr) {
            cout << "The playlist is empty.\n";
            return;
        }
        Node* current = last->next;
        Node* prev = last;

        do {
            if (current->song == song) {
                if (current == last && current->next == last) {
                    last = nullptr;
                } else if (current == last) {
                    prev->next = current->next;
                    last = prev;
                } else if (current == last->next) {
                    last->next = current->next;
                } else {
                    prev->next = current->next;
                }
                delete current;
                cout << song << " has been removed from the playlist.\n";
                return;
            }
            prev = current;
            current = current->next;
        } while (current != last->next);
        cout << song << " was not found in the playlist.\n";
    }
};

```

```

}
void playAll() {
    if (last == nullptr) {
        cout << "The playlist is empty.\n";
        return;
    }
    Node* current = last->next;
    cout << "Playlist: ";
    do {
        cout << current->song << " -> ";
        current = current->next;
    } while (current != last->next);
    cout << "back to " << last->next->song << endl;
}
void nextSong() {
    if (last == nullptr) {
        cout << "The playlist is empty.\n";
        return;
    }
    last = last->next;
    cout << "Now playing: " << last->next->song << endl;
}
void previousSong() {
    if (last == nullptr) {
        cout << "The playlist is empty.\n";
        return;
    }
    Node* current = last->next;
    while (current->next != last) {
        current = current->next;
    }
    last = current;
    cout << "Now playing: " << last->next->song << endl;
}
};

int main() {
    Playlist playlist;

    playlist.addSong("Apple Dance");
    playlist.addSong("Walwal");
    playlist.addSong("Freaky Friday");

    playlist.playAll();

    playlist.nextSong();

    playlist.removeSong("Walwal");

    playlist.playAll();
}

```



```
playlist.previousSong();

return 0;
}
```

Output

Clear

```
/tmp/RycgxmloaY.o
Apple Dance has been added to the playlist.
Walwal has been added to the playlist.
Freaky Friday has been added to the playlist.
Playlist: Apple Dance -> Walwal -> Freaky Friday -> back to Apple Dance
Now playing: Walwal
Walwal has been removed from the playlist.
Playlist: Freaky Friday -> Apple Dance -> back to Freaky Friday
Now playing: Apple Dance

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

8. Conclusion

In this activity, I gained a deeper understanding of linked lists as a dynamic data structure and how they differ from arrays, particularly in terms of flexibility for frequent insertions and deletions. Implementing both singly and doubly linked lists reinforced the concept of pointer-based structures, while modifying operations for the doubly linked list highlighted the advantages of bidirectional traversal. The supplementary activity, which involved creating a circular linked list for a song playlist, showcased how variations of linked lists can be applied to real-world scenarios, enhancing functionality like looping through songs. Overall, I believe I performed well, successfully implementing the required functions, though I recognize the need for further practice on more complex operations and edge cases, such as handling empty lists or optimizing insertion and deletion processes.

9. Assessment Rubric