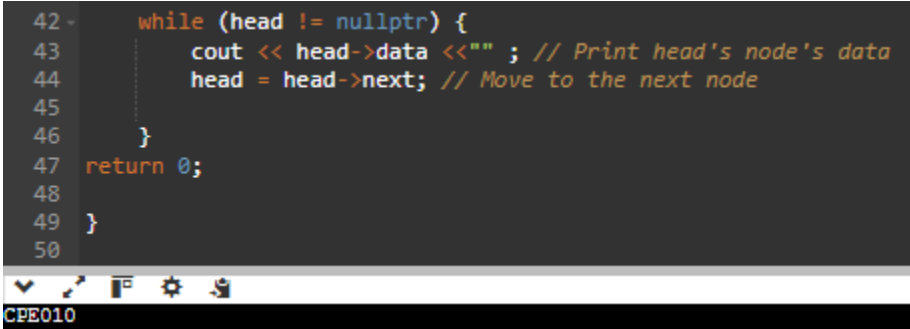


Activity No. <n>	
<Replace with Title>	
Course Code: CPE010	Program: Computer Engineering
Course Title: Data Structures and Algorithms	Date Performed: 09/27/24
Section:CPE21S4	Date Submitted:09/27/24
Name(s): Christan Ray R. Sanchez	Instructor: Professor Ma. Rizzete Sayo

6. Output

TABLE 3.1	SCREENSHOTTED OUTPUT:
	
	<p>How the output came to be and how could it be improved</p> <p>In a singly linked list, each node consists of two parts: the data and the pointer (next). In the main body, we assigned the data for the nodes to `null` as a safety measure to ensure that the data we input is accurate. Next is the program structure, which shows how each node is connected to another, and how we assign data to them so they can be stored in the `Node` class. Finally, we print out the data, starting from the head, and continue traversing the list until it reaches the end.</p> <p>We can also add a function that allows us to delete the stored data so that we can rewrite its contents. Additionally, we can use a specific variable, such as `current`, in the output function, which can be declared and used in the `while` loop to iterate through the nodes.</p>

Operation	Screenshot
Traversal	<pre>// Function to traverse and print the linked list void ListTraversal(Node* n) { while (n != nullptr) { cout << n->data; n = n->next; } cout << endl; }</pre>
Insertion at head	<pre>// Function to insert a node at the head void insertAtHead(Node** head_ref, char new_data) { Node* new_node = new Node(); new_node->data = new_data; new_node->next = *head_ref; *head_ref = new_node; }</pre>
Insertion at any part of the list	<pre>// Function to insert a node after a given node void insertAfter(Node* prev_node, char new_data) { if (prev_node == nullptr) { cout << "Previous node cannot be null." << endl; return; } Node* new_node = new Node(); new_node->data = new_data; new_node->next = prev_node->next; prev_node->next = new_node; }</pre>
Insertion at the end	<pre>// Function to insert a node at the end (tail) void insertAtEnd(Node** head_ref, char new_data) { Node* new_node = new Node(); new_node->data = new_data; new_node->next = nullptr; if (*head_ref == nullptr) { *head_ref = new_node; return; } Node* last = *head_ref; while (last->next != nullptr) { last = last->next; } last->next = new_node; }</pre>

Deletion of a node	<pre>// Function to delete a node by value void deleteNode(Node** head_ref, char key) { Node* temp = *head_ref; Node* prev = nullptr; if (temp != nullptr && temp->data == key) { *head_ref = 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>
--------------------	--

a.	SOURCE CODE	// Task a: Traverse and display the initial list cout << "Initial linked list: "; ListTraversal(head);
	CONSOLE	Initial linked list: GCPE101
b.	SOURCE CODE	// Task b: Insert 'G' at the head insertAtHead(&head, 'G'); cout << "After inserting 'G' at the start: "; ListTraversal(head); // Output: GCPE101
	CONSOLE	After inserting 'G' at the start: GCPE101
c.	SOURCE CODE	// Task c: Insert 'E' after 'P' insertAfter(second, 'E'); cout << "After inserting 'E' after 'P': "; ListTraversal(head); // Output: GCPEE101
	CONSOLE	After inserting 'E' after 'P': GCPEE101
d.	SOURCE CODE	// Task d: Delete node containing 'C' deleteNode(&head, 'C'); cout << "After deleting 'C': "; ListTraversal(head); // Output: GPPE101
	CONSOLE	After deleting 'C': GPPE101
e.	SOURCE CODE	// Task e: Delete node containing 'P'

		deleteNode(&head, 'P'); cout << "After deleting 'P': "; ListTraversal(head); // Output: GEE101
	CONSOLE	After deleting 'P': GEE101
f.	SOURCE CODE	// Task f: Final display of the linked list cout << "Final linked list: "; ListTraversal(head); // Output: GEE101
	CONSOLE	Final linked list: GEE101

SOURCE CODE for the whole output	<pre> #include<iostream> #include<utility> using namespace std; // Define the Node class class Node { public: char data; Node* next; }; // Function to insert a node at the head void insertAtHead(Node** head_ref, char new_data) { Node* new_node = new Node(); new_node->data = new_data; new_node->next = *head_ref; *head_ref = new_node; } // Function to insert a node at the end (tail) void insertAtEnd(Node** head_ref, char new_data) { Node* new_node = new Node(); new_node->data = new_data; new_node->next = nullptr; if (*head_ref == nullptr) { *head_ref = new_node; return; } Node* last = *head_ref; while (last->next != nullptr) { last = last->next; } last->next = new_node; </pre>
----------------------------------	--

```

}

// Function to insert a node after a given node
void insertAfter(Node* prev_node, char new_data) {
    if (prev_node == nullptr) {
        cout << "Previous node cannot be null." << endl;
        return;
    }
    Node* new_node = new Node();
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}

// Function to delete a node by value
void deleteNode(Node** head_ref, char key) {
    Node* temp = *head_ref;
    Node* prev = nullptr;

    if (temp != nullptr && temp->data == key) {
        *head_ref = 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;
}

// Function to traverse and print the linked list
void ListTraversal(Node* n) {
    while (n != nullptr) {
        cout << n->data;
        n = n->next;
    }
    cout << endl;
}

int main() {
    // Step 1: Initialize the nodes
    Node *head = nullptr;
    Node *second = nullptr;
    Node *third = nullptr;
    Node *fourth = nullptr;

```

```

Node *fifth = nullptr;
Node *last = nullptr;

// Step 2: Allocate memory for each node
head = new Node;
second = new Node;
third = new Node;
fourth = new Node;
fifth = new Node;
last = new Node;

// Step 3: Set data and link the nodes
head->data = 'C';
head->next = second;

second->data = 'P';
second->next = third;

third->data = 'E';
third->next = fourth;

fourth->data = '1';
fourth->next = fifth;

fifth->data = '0';
fifth->next = last;

last->data = '1';
last->next = nullptr;

// Task a: Traverse and display the initial list
cout << "Initial linked list: ";
ListTraversal(head);

// Task b: Insert 'G' at the head
insertAtHead(&head, 'G');
cout << "After inserting 'G' at the start: ";
ListTraversal(head); // Output: GCPE101

// Task c: Insert 'E' after 'P'
insertAfter(second, 'E');
cout << "After inserting 'E' after 'P': ";
ListTraversal(head); // Output: GCPEE101

// Task d: Delete node containing 'C'
deleteNode(&head, 'C');
cout << "After deleting 'C': ";
ListTraversal(head); // Output: GPEE101

// Task e: Delete node containing 'P'
deleteNode(&head, 'P');

```

	<pre> cout << "After deleting 'P': "; ListTraversal(head); // Output: GEE101 // Task f: Final display of the linked list cout << "Final linked list: "; ListTraversal(head); // Output: GEE101 return 0; } </pre>
OUTPUT	<pre> Initial linked list: CPE101 After inserting 'G' at the start: GCPE101 After inserting 'E' after 'P': GCPEE101 After deleting 'C': GP EE101 After deleting 'P': GEE101 Final linked list: GEE101 </pre>

7. Supplementary Activity

SOURCE CODE	<pre> #include <iostream> #include <string> using namespace std; // Node structure representing a song in the playlist class SongNode { public: string songName; SongNode* next; // Constructor to create a new node with a song name SongNode(string song) { songName = song; next = nullptr; } }; // Circular Linked List class for managing the song playlist class Playlist { private: SongNode* last; // Points to the last node (song) in the playlist public: Playlist() { last = nullptr; // Initially, playlist is empty } // Function to add a new song to the playlist void addSong(string songName) { </pre>
-------------	---

```

SongNode* newSong = new SongNode(songName);

if (last == nullptr) { // If the list is empty
    last = newSong;
    last->next = last; // Points to itself, forming a
circular link
} else {
    newSong->next = last->next; // Link the new song
to the first song
    last->next = newSong; // The current last song
points to the new song
    last = newSong; // The new song becomes the
last song
}
cout << "Added: " << songName << " to the playlist."
<< endl;
}

// Function to remove a song from the playlist by name
void removeSong(string songName) {
    if (last == nullptr) { // If the playlist is empty
        cout << "Playlist is empty, cannot delete." << endl;
        return;
    }

    SongNode* temp = last->next;
    SongNode* prev = last;
    do {
        if (temp->songName == songName) {
            if (temp == last && temp->next == last) { // Only
one song in the list
                delete temp;
                last = nullptr; // Empty the playlist
            } else {
                prev->next = temp->next;
                if (temp == last) last = prev; // Update last if
last song is removed
                delete temp;
            }
            cout << "Removed: " << songName << " from
the playlist." << endl;
            return;
        }
        prev = temp;
        temp = temp->next;
    } while (temp != last->next); // Stop when we loop
back to the first song

    cout << "Song not found in the playlist." << endl;
}

```



```

// Function to play all songs (traverse playlist in a loop)
void playAllSongs() {
    if (last == nullptr) {
        cout << "Playlist is empty." << endl;
        return;
    }

    SongNode* temp = last->next; // Start from the first
song
    cout << "Playing all songs in the playlist." << endl;
    do {
        cout << temp->songName << " -> ";
        temp = temp->next;
    } while (temp != last->next); // Loop back to the start
    cout << "Repeat..." << endl; // Since it's a circular
playlist
}
};

// Driver function to test the Playlist
int main() {
    Playlist myPlaylist;

    // Add songs to the playlist
    myPlaylist.addSong("Song A");
    myPlaylist.addSong("Song B");
    myPlaylist.addSong("Song C");
    myPlaylist.addSong("Song D");

    // Play all songs
    myPlaylist.playAllSongs();

    // Remove a song and display the playlist again
    myPlaylist.removeSong("Song B");
    myPlaylist.playAllSongs();

    // Remove another song and display the playlist
    myPlaylist.removeSong("Song A");
    myPlaylist.playAllSongs();

    return 0;
}

```

OUTPUT

```
Added: Song A to the playlist.  
Added: Song B to the playlist.  
Added: Song C to the playlist.  
Added: Song D to the playlist.  
Playing all songs in the playlist:  
Song A -> Song B -> Song C -> Song D -> Repeat...  
Removed: Song B from the playlist.  
Playing all songs in the playlist:  
Song A -> Song C -> Song D -> Repeat...  
Removed: Song A from the playlist.  
Playing all songs in the playlist:  
Song C -> Song D -> Repeat...
```

8. Conclusion

In this activity, I really dug into the basics of linked lists, especially focusing on circular linked lists. Here are some key takeaways:

- **Understanding Linked Lists:** I got a solid grasp of how linked lists are structured, learning how nodes connect through pointers and why managing memory is so important for dynamic data structures.
- **What Makes Circular Linked Lists Special:** I discovered that circular linked lists allow you to loop through elements continuously, which is great for applications like music playlists or games where players take turns in a circle.
- **Performing Operations on Data:** I practiced essential operations like adding, removing, and navigating through elements, which are crucial for handling dynamic data.
- **Crafting Functions:** I learned how to design functions that effectively manage and access data within the linked list, which really helped improve my coding skills and understanding of object-oriented programming.

Evaluation of the Procedure

Creating a circular linked list was key in making it easy to add and remove songs from a playlist. Here's a rundown of what I did:

1. **Creating the Node Structure:** I set up a "SongNode" class to represent each song, which includes the song title and a pointer to the next node.
2. **Managing the Playlist:** I built a "Playlist" class that controls the circular linked list, including methods to add, remove, and navigate through songs.
3. **Following Good Coding Practices:** I made sure to keep my code organized, with each function having a clear purpose.
4. **Testing Everything:** I ran through various operations on the playlist, checking for tricky situations like removing the last song and handling an empty playlist, ensuring that everything worked smoothly.

Reflection on the Supplementary Activity

The extra task of implementing a circular linked list for a song playlist was both enlightening and enjoyable. It helped me understand:

- **Real-World Applications:** I saw how circular linked lists can be applied in practical scenarios, like managing music playlists or organizing player turns in games.
- **Critical Thinking:** This activity pushed me to think about how to structure data and consider the impact of different operations.
- **Importance of Efficiency:** I became more aware of the need for writing efficient code, especially in how to navigate and manage memory in linked lists.

Thoughts on the Activity

My experience with this activity was quite challenging. I found it difficult to grasp the new commands and concepts, especially since they were unfamiliar to me. Even though I have a basic understanding of C++, applying these new ideas effectively was tough.

Areas for Improvement

When it comes to coding, I want to focus on a few key areas for improvement:

1. **Cleaner Code:** I aim to write code that is more streamlined and easier to read. This means avoiding unnecessary commands and focusing on clarity in my logic and structure.
2. **Understanding New Concepts:** I need to work on really understanding new commands and concepts as I encounter them. Taking the time to break them down and practice will help solidify my knowledge.
3. **Debugging Skills:** I want to enhance my debugging skills so I can quickly identify and fix issues in my code. Learning to read error messages and trace problems back to their source will be beneficial.
4. **Code Efficiency:** I'd like to explore ways to make my code more efficient, whether that's through optimizing algorithms or reducing resource usage.
5. **Practical Application:** Finally, I want to apply what I learn in real-world projects or exercises. This hands-on experience will help reinforce my knowledge and improve my confidence in coding.

Final Thoughts

Overall, this activity was a valuable learning experience. I'm excited to apply these concepts to more complex topics as I continue my studies in computer science.

9. Assessment Rubric