Advanced Linked Lists Lab Session

Data Structures & Algorithms

Class 02 - Doubly, Circular & Sentinel Nodes

Duration: 2 hours (4 challenges × 30 minutes each)

Lab Structure

Each challenge follows this format:

1. **Challenge Presentation** (2 min)
2. **Your Implementation Time** (20 min)
3. **Pseudocode Review** (3 min)
4. **Solution Walkthrough** (5 min)
5. **Code Explanation** (5 min)

Total per challenge: ~35 minutes

Today's Topics

Doubly Linked Lists

- Nodes with both `next` and `prev` pointers
- Bidirectional traversal

Circular Linked Lists

- Last node points back to first (no NULL!)
- Endless loop potential

Sentinel Nodes

- Dummy nodes to simplify edge cases
- Eliminate NULL checks

Challenge 1

Build a Music Playlist (Doubly Linked List)

Challenge 1: Problem Statement

Task: Create a doubly linked list representing a music playlist where you can navigate forward and backward through songs.

Playlist:

```
"Bohemian Rhapsody" ⇄ "Stairway to Heaven" ⇄ "Hotel California" ⇄ "Imagine"
```

Requirements:

- Implement doubly linked node structure (both `next` and `prev`)
- Create 4 song nodes
- Link them bidirectionally
- First node's `prev` = NULL
- Last node's `next` = NULL

Time: 20 minutes ⏰

Challenge 1: Pseudocode

```
ALGORITHM CreateMusicPlaylist
  1. Define DoublyNode structure with:
     - string data (song name)
     - pointer to next node
     - pointer to prev (previous) node

  2. Create helper function createDoublyNode(songName):
     - Allocate memory for new node
     - Set node.data = songName
     - Set node.next = NULL
     - Set node.prev = NULL
     - Return pointer to new node

  3. In main function:
     - Create 4 nodes for songs
     - Link forward (next pointers):
       * song1.next = song2
       * song2.next = song3
       * song3.next = song4
     - Link backward (prev pointers):
       * song2.prev = song1
       * song3.prev = song2
       * song4.prev = song3
     - Set boundaries:
       * song1.prev = NULL
       * song4.next = NULL

  4. Return head pointer
END ALGORITHM
```

Challenge 1: Solution Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Doubly linked node structure
struct DoublyNode {
    char data[100];
    struct DoublyNode* next;
    struct DoublyNode* prev;
};

// Helper function to create a doubly linked node
struct DoublyNode* createDoublyNode(const char* song) {
    struct DoublyNode* newNode = (struct DoublyNode*)malloc(sizeof(struct DoublyNode));
    strcpy(newNode->data, song);
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
```

Challenge 1: Solution Code (continued)

```
int main() {
    // Create song nodes
    struct DoublyNode* song1 = createDoublyNode("Bohemian Rhapsody");
    struct DoublyNode* song2 = createDoublyNode("Stairway to Heaven");
    struct DoublyNode* song3 = createDoublyNode("Hotel California");
    struct DoublyNode* song4 = createDoublyNode("Imagine");

    // Link forward (next pointers)
    song1->next = song2;
    song2->next = song3;
    song3->next = song4;
    song4->next = NULL;

    // Link backward (prev pointers)
    song1->prev = NULL;
    song2->prev = song1;
    song3->prev = song2;
    song4->prev = song3;

    // Head is song1
    struct DoublyNode* head = song1; // This is a standard head pointer, not a sentinel node.

    return 0;
}
```

Challenge 1: Code Explanation

Part 1: Doubly Linked Node Structure

```
struct DoublyNode {
    char data[100];           // Song name
    struct DoublyNode* next; // Pointer to next song
    struct DoublyNode* prev; // Pointer to previous song
};
```

Key difference from singly linked list:

- `prev` **pointer**: Enables backward traversal
- **Two connections per node**: Can move both directions
- **More memory**: Each node stores 2 pointers instead of 1

Challenge 1: Code Explanation

Part 2: Forward Linking

```
song1->next = song2;
song2->next = song3;
song3->next = song4;
song4->next = NULL;  // End of playlist
```

Visual representation:

```
song1 → song2 → song3 → song4 → NULL
```

This looks just like a singly linked list! The magic comes with backward links...

Challenge 1: Code Explanation

Part 3: Backward Linking

```
song1->prev = NULL;  // Start of playlist
song2->prev = song1;
song3->prev = song2;
song4->prev = song3;
```

Visual representation:

```
NULL ← song1 ← song2 ← song3 ← song4
```

Combined (bidirectional):

```
NULL ⇄ song1 ⇄ song2 ⇄ song3 ⇄ song4 ⇄ NULL
```

Challenge 1: Code Explanation

Part 4: Why Doubly Linked Lists?

Advantages:

✅ Navigate backward easily (no need to restart from head)

✅ Delete nodes more efficiently (have direct access to previous)

✅ Insert before a given node without traversing from head

Disadvantages:

❌ More memory (extra pointer per node)

❌ More complex operations (maintain 2 pointers)

❌ Slightly slower insertion/deletion (more pointer updates)

Real-world use: Media players, browser history, undo/redo systems

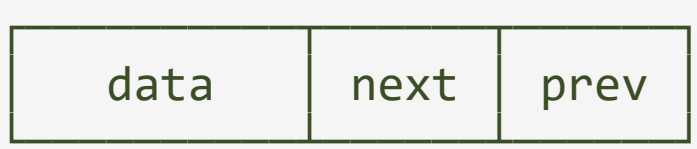
Challenge 1: Code Explanation

Part 5: Memory Layout Comparison

Singly Linked Node:


```
~50 bytes  8 bytes = ~58 bytes
```

Doubly Linked Node:




```
~50 bytes  8 bytes 8 bytes = ~66 bytes
```

Memory overhead: ~14% more memory per node

Challenge 2

Navigate Playlist Forward and Backward

Challenge 2: Problem Statement

Task: Write two functions to traverse a doubly linked list in both directions.

Requirements:

- Function 1: `printForward(head)` - print from first to last
- Function 2: `printBackward(tail)` - print from last to first
- Both return `void`
- Handle empty lists
- Format: `song1 ⇄ song2 ⇄ song3 ⇄ NULL`

Bonus: Write a function `findTail(head)` to locate the last node

Time: 20 minutes ⏰

Challenge 2: Pseudocode

```
ALGORITHM PrintForward(head)
  1. If head is NULL:
     - Print "Playlist is empty"
     - Return

  2. current = head

  3. While current is NOT NULL:
     - Print current.data + " ⇄ "
     - current = current.next

  4. Print "NULL"
END ALGORITHM

ALGORITHM PrintBackward(tail)
  1. If tail is NULL:
     - Print "Playlist is empty"
     - Return

  2. current = tail

  3. While current is NOT NULL:
     - Print current.data + " ⇄ "
     - current = current.prev  // Go backward!

  4. Print "NULL"
END ALGORITHM
```

Challenge 2: Solution Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct DoublyNode {
    char data[100];
    struct DoublyNode* next;
    struct DoublyNode* prev;
};

// Function to print playlist forward
void printForward(struct DoublyNode* head) {
    if (head == NULL) {
        printf("Playlist is empty\n");
        return;
    }

    printf("Forward: ");
    struct DoublyNode* current = head;
    while (current != NULL) {
        printf("%s", current->data);
        if (current->next != NULL) printf(" ⇄ ");
        current = current->next;
    }
    printf(" ⇄ NULL\n");
}
```

Challenge 2: Solution Code (continued)

```c
// Function to print playlist backward
void printBackward(struct DoublyNode* tail) {
    if (tail == NULL) {
        printf("Playlist is empty\n");
        return;
    }

    printf("Backward: ");
    struct DoublyNode* current = tail;
    while (current != NULL) {
        printf("%s", current->data);
        if (current->prev != NULL) printf(" ⇄ ");
        current = current->prev;  // Move backward!
    }
    printf(" ⇄ NULL\n");
}

// Bonus: Find the last node (tail)
struct DoublyNode* findTail(struct DoublyNode* head) {
    if (head == NULL) return NULL;

    struct DoublyNode* current = head;
    while (current->next != NULL) {
        current = current->next;
    }
    return current;
}
```

Challenge 2: Solution Code (Test Cases)

```
int main() {
    // Create playlist
    struct DoublyNode* song1 = createDoublyNode("Bohemian Rhapsody");
    struct DoublyNode* song2 = createDoublyNode("Stairway to Heaven");
    struct DoublyNode* song3 = createDoublyNode("Hotel California");

    // Link them
    song1->next = song2; song2->prev = song1;
    song2->next = song3; song3->prev = song2;
    song1->prev = NULL; song3->next = NULL;

    // Test forward traversal
    printForward(song1);
    // Output: Forward: Bohemian Rhapsody ⇄ Stairway to Heaven ⇄ Hotel California ⇄ NULL

    // Test backward traversal
    struct DoublyNode* tail = findTail(song1);
    printBackward(tail);
    // Output: Backward: Hotel California ⇄ Stairway to Heaven ⇄ Bohemian Rhapsody ⇄ NULL

    return 0;
}
```

Challenge 2: Code Explanation

Part 1: Forward Traversal

```
while (current != NULL) {
    printf("%s", current->data);
    current = current->next;  // Move forward
}
```

Just like singly linked list!

- Same logic, same direction
- `prev` pointer not used here
- Goes from head → tail

Challenge 2: Code Explanation

Part 2: Backward Traversal - The Magic!

```
while (current != NULL) {
    printf("%s", current->data);
    current = current->prev;  // Move backward!
}
```

The key difference:

- `current = current->prev` instead of `current->next`
- Start from tail, go to head
- **Impossible with singly linked lists!**

Challenge 2: Code Explanation

Part 3: Finding the Tail

```
struct DoublyNode* findTail(struct DoublyNode* head) {
    if (head == NULL) return NULL;

    struct DoublyNode* current = head;
    while (current->next != NULL) {
        current = current->next;
    }
    return current;  // Last node (tail)
}
```

Why needed?

- To print backward, we need to start from the end
- No direct access to tail (only have head)
- **Time complexity:** O(n) - must traverse entire list

Challenge 2: Code Explanation

Part 4: Execution Trace for Backward Traversal

Playlist: `song1 ⇄ song2 ⇄ song3 ⇄ NULL`

Iteration	current	Action	Output
Start	song3 (tail)	—	—
1	song3	Print, move prev	`Hotel California ⇄`
2	song2	Print, move prev	`Stairway to Heaven ⇄`
3	song1	Print, move prev	`Bohemian Rhapsody ⇄`
4	NULL	Exit loop	`NULL`

Result: Songs printed in reverse order!

Challenge 3

Build a Running Track (Circular Linked List)

Challenge 3: Problem Statement

Task: Create a circular linked list representing a running track where runners keep looping around checkpoints.

Checkpoints:

```
Start Line → Checkpoint 1 → Checkpoint 2 → Checkpoint 3 → (back to Start)
```

Requirements:

- Use singly linked nodes
- Create 4 checkpoint nodes
- Last node points back to first (circular!)
- No NULL anywhere in the list

Challenge: Write a function to print N laps around the track

Time: 20 minutes ⏰

Challenge 3: Pseudocode

```
ALGORITHM CreateCircularTrack
  1. Create 4 nodes:
     - start = createNode("Start Line")
     - cp1 = createNode("Checkpoint 1")
     - cp2 = createNode("Checkpoint 2")
     - cp3 = createNode("Checkpoint 3")

  2. Link nodes in sequence:
     - start.next = cp1
     - cp1.next = cp2
     - cp2.next = cp3
     - cp3.next = start  // Make it circular!

  3. Return start (head of circular list)
END ALGORITHM

ALGORITHM PrintNLaps(head, laps)
  1. total_checkpoints = laps × 4  // 4 checkpoints per lap

  2. current = head

  3. For i = 1 to total_checkpoints:
     - Print current.data
     - current = current.next

  4. Print "Finished!"
END ALGORITHM
```

Challenge 3: Solution Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Regular singly linked node (no prev)
struct Node {
    char data[50];
    struct Node* next;
};

// Create a node
struct Node* createNode(const char* checkpoint) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    strcpy(newNode->data, checkpoint);
    newNode->next = NULL;
    return newNode;
}
```

Challenge 3: Solution Code (continued)

```
// Create circular running track
struct Node* createCircularTrack() {
    // Create checkpoints
    struct Node* start = createNode("Start Line");
    struct Node* cp1 = createNode("Checkpoint 1");
    struct Node* cp2 = createNode("Checkpoint 2");
    struct Node* cp3 = createNode("Checkpoint 3");

    // Link them in sequence
    start->next = cp1;
    cp1->next = cp2;
    cp2->next = cp3;
    cp3->next = start;  // Make it circular!

    return start;  // Return head
}
```

Challenge 3: Solution Code (Print N Laps)

```c
// Print N laps around the track
void printNLaps(struct Node* head, int laps) {
    if (head == NULL || laps <= 0) {
        printf("Invalid input\n");
        return;
    }

    struct Node* current = head;
    int totalCheckpoints = laps * 4;  // 4 checkpoints per lap

    printf("Running %d lap(s):\n", laps);
    for (int i = 0; i < totalCheckpoints; i++) {
        printf("  %s", current->data);
        current = current->next;

        // Mark lap completion
        if ((i + 1) % 4 == 0) {
            printf(" [Lap %d complete!]\n", (i + 1) / 4);
        } else {
            printf(" → ");
        }
    }
    printf("Finished!\n");
}
```

Challenge 3: Solution Code (Test Cases)

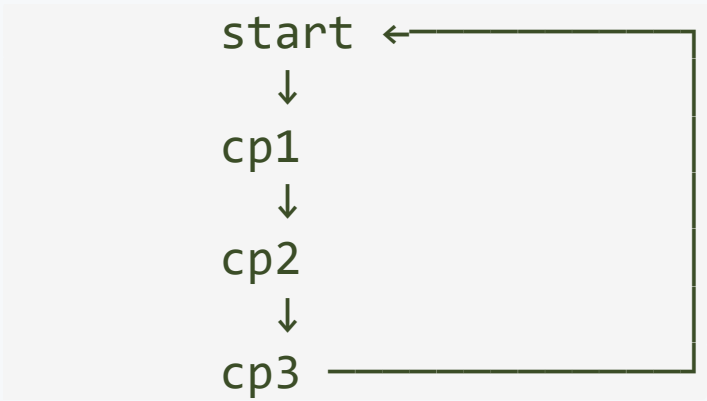
```
int main() {
    // Create circular track
    struct Node* track = createCircularTrack();

    // Run 2 laps
    printNLaps(track, 2);

    /* Expected Output:
    Running 2 lap(s):
      Start Line → Checkpoint 1 → Checkpoint 2 → Checkpoint 3 [Lap 1 complete!]
      Start Line → Checkpoint 1 → Checkpoint 2 → Checkpoint 3 [Lap 2 complete!]
    Finished!
    */

    return 0;
}
```

Challenge 3: Code Explanation

Part 1: The Circular Connection

```
start->next = cp1;
cp1->next = cp2;
cp2->next = cp3;
cp3->next = start;  // KEY: Points back to start!
```

Visual representation:



No NULL anywhere! Every node points to another node.

Challenge 3: Code Explanation

Part 2: Danger Zone - Infinite Loop!

```
// ❌ NEVER DO THIS WITH CIRCULAR LISTS!
while (current != NULL) {
    printf("%s ", current->data);
    current = current->next;
}
// This will NEVER terminate! (current is never NULL)
```

Why?

- In circular lists, `next` is never NULL
- Loop continues forever
- **Infinite loop = program freeze!**

Challenge 3: Code Explanation

Part 3: Safe Traversal with Counter

```
for (int i = 0; i < totalCheckpoints; i++) {
    printf("%s", current->data);
    current = current->next;  // Safe because we count iterations
}
```

Two safe approaches:

1. **Counter-based** (used here): Limit iterations
2. **Sentinel check**: Stop when you reach the starting node again

```
struct Node* start = head;
do {
    printf("%s ", current->data);
    current = current->next;
} while (current != start);
```

Challenge 3: Code Explanation

Part 4: Execution Trace (2 Laps)

Iteration	current	Output	Notes
1	Start Line	`Start Line →`	Lap 1 begins
2	Checkpoint 1	`Checkpoint 1 →`	
3	Checkpoint 2	`Checkpoint 2 →`	
4	Start Line	`Checkpoint 3 [Lap 1 complete!]`	Back to start!
5	Start Line	`Start Line →`	Lap 2 begins
6	Checkpoint 1	`Checkpoint 1 →`	
7	Checkpoint 2	`Checkpoint 2 →`	
8	Checkpoint 3	`Checkpoint 3 [Lap 2 complete!]`	

Notice: We passed through "Start Line" twice!

Challenge 3: Code Explanation

Part 5: When to Use Circular Lists?

Real-world applications:

✅ **Round-robin scheduling**: CPU process scheduling
✅ **Multiplayer games**: Turn-based games (player turns in a circle)
✅ **Music playlists**: Repeat all songs continuously
✅ **Buffering**: Circular buffers in operating systems
✅ **Board games**: Monopoly, Life (circular board)

Key advantage: No end - continuous cycling

Challenge 4

Smart Playlist with Sentinel Nodes

Challenge 4: Problem Statement

Task: Create a doubly linked list with a **sentinel node** (dummy node) that:

- Simplifies insertion/deletion (no NULL checks!)
- Acts as boundary marker
- Never holds actual data

Requirements:

- Use sentinel node for empty playlist
- Sentinel's `next` points to first song (or back to itself if empty)
- Sentinel's `prev` points to last song (or back to itself if empty)
- Implement: `insertAtEnd()` and `printPlaylist()`

Time: 20 minutes ⏰

Challenge 4: Pseudocode

```
ALGORITHM CreatePlaylistWithSentinel
  1. Create sentinel node:
     - sentinel.data = "[SENTINEL]" (or any marker)
     - sentinel.next = sentinel
     - sentinel.prev = sentinel

  2. Return sentinel
END ALGORITHM

ALGORITHM InsertAtEnd(sentinel, songName)
  1. Create new song node

  2. Get current last song:
     - lastSong = sentinel.prev

  3. Insert new song:
     - newSong.next = sentinel
     - newSong.prev = lastSong
     - lastSong.next = newSong
     - sentinel.prev = newSong

  // No NULL checks needed!
END ALGORITHM
```

Challenge 4: Pseudocode (continued)

```
ALGORITHM PrintPlaylist(sentinel)
  1. current = sentinel.next  // Skip sentinel

  2. If current == sentinel:
     - Print "Playlist is empty"
     - Return

  3. While current != sentinel:
     - Print current.data
     - current = current.next

  4. Print "End of playlist"
END ALGORITHM
```

Challenge 4: Solution Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct DoublyNode {
    char data[100];
    struct DoublyNode* next;
    struct DoublyNode* prev;
};

// Create a node
struct DoublyNode* createNode(const char* song) {
    struct DoublyNode* node = (struct DoublyNode*)malloc(sizeof(struct DoublyNode));
    strcpy(node->data, song);
    node->next = NULL;
    node->prev = NULL;
    return node;
}
```

Challenge 4: Solution Code (continued)

```
// Create playlist with sentinel node
struct DoublyNode* createPlaylistWithSentinel() {
    struct DoublyNode* sentinel = createNode("[SENTINEL]");

    // Sentinel points to itself (empty list)
    sentinel->next = sentinel;
    sentinel->prev = sentinel;

    return sentinel;
}

// Insert song at the end
void insertAtEnd(struct DoublyNode* sentinel, const char* songName) {
    struct DoublyNode* newSong = createNode(songName);

    // Get the last song (prev of sentinel)
    struct DoublyNode* lastSong = sentinel->prev;

    // Insert new song
    newSong->next = sentinel;
    newSong->prev = lastSong;
    lastSong->next = newSong;
    sentinel->prev = newSong;

    // No NULL checks needed!
}
```

Challenge 4: Solution Code (Print Function)

```
// Print playlist (skip sentinel)
void printPlaylist(struct DoublyNode* sentinel) {
    struct DoublyNode* current = sentinel->next;  // Skip sentinel

    // Check if empty
    if (current == sentinel) {
        printf("Playlist is empty\n");
        return;
    }

    printf("Playlist:\n");
    int songNum = 1;
    while (current != sentinel) {
        printf("  %d. %s\n", songNum++, current->data);
        current = current->next;
    }
}
```

Challenge 4: Solution Code (Test Cases)

```c
int main() {
    // Create empty playlist with sentinel
    struct DoublyNode* playlist = createPlaylistWithSentinel();

    // Initially empty
    printPlaylist(playlist);
    // Output: Playlist is empty

    // Add songs
    insertAtEnd(playlist, "Bohemian Rhapsody");
    insertAtEnd(playlist, "Stairway to Heaven");
    insertAtEnd(playlist, "Hotel California");

    // Print playlist
    printPlaylist(playlist);
    /* Output:
    Playlist:
      1. Bohemian Rhapsody
      2. Stairway to Heaven
      3. Hotel California
    */

    return 0;
}
```

Challenge 4: Code Explanation

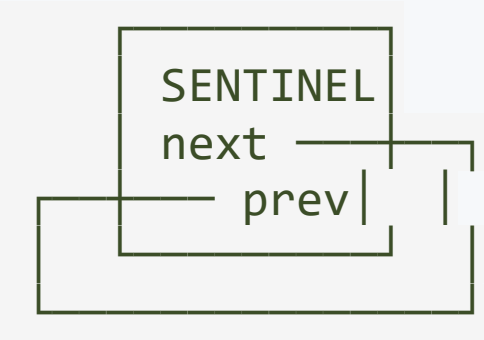
Part 1: What is a Sentinel Node?

Definition: A dummy node that doesn't hold real data, used as a boundary marker.

Initialization:

```
sentinel->next = sentinel;  // Points to itself
sentinel->prev = sentinel;  // Points to itself
```

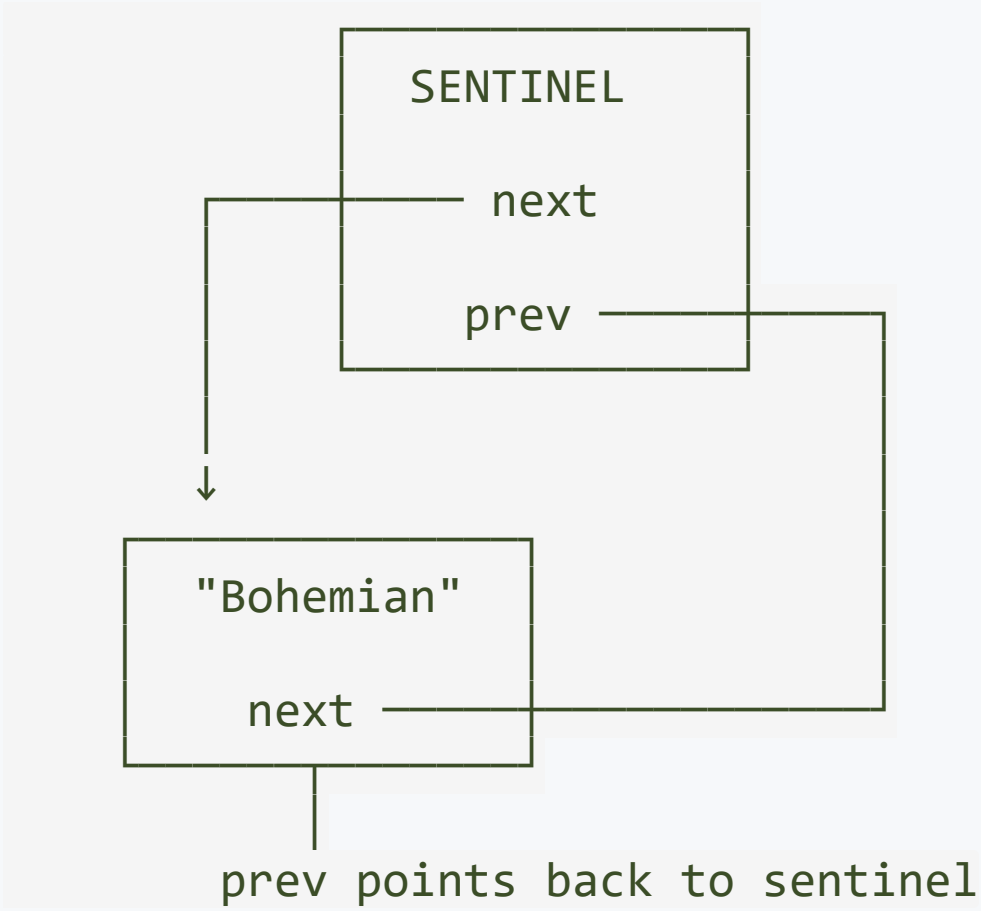
Visual (empty list):



Challenge 4: Code Explanation

Part 2: Sentinel with One Song

After inserting "Bohemian Rhapsody":



Challenge 4: Code Explanation

Part 3: Insertion Logic (No NULL Checks!)

```
void insertAtEnd(struct DoublyNode* sentinel, const char* songName) {
    struct DoublyNode* newSong = createNode(songName);
    struct DoublyNode* lastSong = sentinel->prev;  // Works even if empty!

    newSong->next = sentinel;
    newSong->prev = lastSong;
    lastSong->next = newSong;
    sentinel->prev = newSong;
}
```

Why no NULL checks?

- **Empty list:** `sentinel->prev = sentinel`, so `lastSong = sentinel`
- **Non-empty:** `sentinel->prev = actual last song`
- **Both cases work with same code!**

Challenge 4: Code Explanation

Part 4: Comparison - With vs Without Sentinel

Without Sentinel (traditional approach):

```
void insertAtEnd(struct DoublyNode** head, const char* song) {
    struct DoublyNode* newNode = createNode(song);

    // Edge case: empty list
    if (*head == NULL) {
        *head = newNode;
        return;
    }

    // Normal case: find tail
    struct DoublyNode* tail = *head;
    while (tail->next != NULL) {
        tail = tail->next;
    }
    tail->next = newNode;
    newNode->prev = tail;
}
```

Challenge 4: Code Explanation

Part 5: Why Use Sentinel Nodes?

Advantages:

✅ **Eliminates NULL checks**: No special cases for empty lists

✅ **Simpler code**: Same logic for empty and non-empty

✅ **Fewer bugs**: No forgetting to check for NULL

✅ **Consistent structure**: Always has at least one node

Disadvantages:

❌ **Extra memory**: One node that doesn't store data

❌ **Slightly confusing**: Need to skip sentinel when traversing

Real-world use: Linux kernel lists, Java's LinkedList, many OS data structures

Challenge 4: Code Explanation

Part 6: Execution Trace

Initial state (empty):

- `sentinel->next = sentinel`
- `sentinel->prev = sentinel`

After `insertAtEnd(sentinel, "Song A")`:

- `sentinel->next = Song A`
- `sentinel->prev = Song A`
- `Song A->next = sentinel`
- `Song A->prev = sentinel`

After `insertAtEnd(sentinel, "Song B")`:

- `sentinel->next = Song A`

Lab Session Complete!

Summary: What You Learned

✅ **Challenge 1:** Created doubly linked list with bidirectional pointers (music playlist)
✅ **Challenge 2:** Traversed lists forward and backward
✅ **Challenge 3:** Built circular linked list (running track with no end)
✅ **Challenge 4:** Implemented sentinel nodes to simplify edge cases

Key Concepts Reinforced

Doubly Linked Lists

- Two pointers per node: `next` and `prev`
- Can traverse in both directions
- More memory but more flexibility

Circular Linked Lists

- Last node points back to first
- No NULL pointers
- Must use counters or sentinels to avoid infinite loops

Sentinel Nodes

- Dummy nodes that simplify logic
- Eliminate NULL checks
- Consistent list structure

Common Mistakes to Avoid

❌ Infinite Loop in Circular Lists

```
while (current != NULL) {  // ❌ Never terminates!
    current = current->next;
}
```

❌ Forgetting to Update Both Pointers

```
node1->next = node2;
// ❌ Forgot: node2->prev = node1;
```

❌ Not Closing the Circle

```
last->next = first;
// ❌ Forgot: first->prev = last; (if doubly linked)
```

Common Mistakes to Avoid (continued)

❌ Modifying Head Pointer in Circular List

```
while (head->next != head) {
    head = head->next;  // ❌ Lost original head!
}
```

❌ Treating Sentinel as Real Data

```
while (current != NULL) {  // ❌ Won't work with sentinel!
    printf("%s", current->data);
    current = current->next;
}
// Should be: while (current != sentinel)
```

Best Practices You Learned

✅ Always link both directions in doubly linked lists

```
node1->next = node2;
node2->prev = node1;  // Don't forget!
```

✅ Use counters for circular list traversal

```
for (int i = 0; i < n; i++) {
    // Safe traversal
}
```

✅ Skip sentinel when processing data

```
current = sentinel->next;  // Skip the dummy node
while (current != sentinel) { ... }
```

✅ Initialize sentinel to point to itself

```
sentinel->next = sentinel;
```

Comparison Table

Feature	Singly	Doubly	Circular	Sentinel
Backward traversal	❌	✅	❌ (unless doubly circular)	Depends on base type
Memory per node	1 pointer	2 pointers	1 pointer	+1 dummy node
Has NULL	✅	✅	❌	❌ (sentinel instead)
Edge case handling	Complex	Complex	Medium	Simple
Infinite loop risk	❌	❌	✅	❌
Use case	General	Navigation	Endless loops	Simplify logic

Time Complexity Comparison

Operation	Singly	Doubly	Circular	Sentinel
Insert at head	O(1)	O(1)	O(1)	O(1)
Insert at tail	O(n)	O(1)*	O(n)	O(1)*
Delete from head	O(1)	O(1)	O(1)	O(1)
Delete from tail	O(n)	O(1)*	O(n)	O(1)*
Search	O(n)	O(n)	O(n)	O(n)
Traverse	O(n)	O(n)	∞ (without limit)	O(n)

*If you maintain a tail pointer