Linked Lists Lab Session

Data Structures & Algorithms

Class 01 - Hands-On Coding Challenges

Duration: 2 hours (4 challenges × 15 minutes each)

Lab Structure

Each challenge follows this format:

1. **Challenge Presentation** (2 min)
2. **Your Implementation Time** (15 min)
3. **Pseudocode Review** (3 min)
4. **Solution Walkthrough** (8 min)
5. **Code Explanation** (7 min)

Total per challenge: ~35 minutes

Challenge 1

Create Daily Routine Linked List

Challenge 1: Problem Statement

Task: Create a linked list representing your daily routine with the following activities:

```
Wake Up → Breakfast → Study → Lunch → Exercise → Dinner → Sleep
```

Requirements:

- Implement the node structure in C
- Create all 7 nodes
- Connect/link them together correctly
- Make sure the last node points to NULL

Time: 15 minutes ⏰

Challenge 1: Pseudocode

```
ALGORITHM CreateDailyRoutine
  1. Define Node structure with:
     - string data (activity name)
     - pointer to next node

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

  3. In main function:
     - Create head node with "Wake Up"
     - Create 6 additional nodes for remaining activities
     - Link each node to the next:
       * head.next = breakfast
       * breakfast.next = study
       * (continue for all nodes...)
     - Last node (Sleep) points to NULL

  4. Return head pointer
END ALGORITHM
```

Challenge 1: Solution Code

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

// Node structure
struct Node {
    char data[50];
    struct Node* next;
};

// Helper function to create a new node
struct Node* createNode(const char* activity) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    strcpy(newNode->data, activity);
    newNode->next = NULL;
    return newNode;
}
```

Challenge 1: Solution Code (continued)

```
int main() {
    // Create all nodes for daily routine
    struct Node* head = createNode("Wake Up");
    struct Node* breakfast = createNode("Breakfast");
    struct Node* study = createNode("Study");
    struct Node* lunch = createNode("Lunch");
    struct Node* exercise = createNode("Exercise");
    struct Node* dinner = createNode("Dinner");
    struct Node* sleep = createNode("Sleep");

    // Link all nodes together
    head->next = breakfast;
    breakfast->next = study;
    study->next = lunch;
    lunch->next = exercise;
    exercise->next = dinner;
    dinner->next = sleep;
    sleep->next = NULL;  // Last node points to NULL

    return 0;
}
```

Challenge 1: Code Explanation

Part 1: Node Structure

```
struct Node {
    char data[50];      // Activity name (up to 50 characters)
    struct Node* next;  // Pointer to next activity
};
```

- `char data[50]`: Stores activity name as string
- `struct Node* next`: Points to next node in the chain

Challenge 1: Code Explanation

Part 2: Creating Nodes

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

- `malloc(sizeof(struct Node))`: Allocates memory on heap
- `strcpy(newNode->data, activity)`: Copies activity name into node
- `newNode->next = NULL`: Initializes next pointer
- `return newNode`: Returns pointer to the created node

Challenge 1: Code Explanation

Part 3: Linking Nodes

```
head->next = breakfast;
breakfast->next = study;
// ... and so on
```

The `->` operator:

- Used to access members through a pointer
- `head->next` means "access the next member of the node pointed to by head"
- Equivalent to `(*head).next`

Result: Creates a chain where each node points to the next activity

Challenge 1: Code Explanation

Part 4: Memory Allocation

```
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

Breaking it down:

- `sizeof(struct Node)`: Calculates bytes needed for one node
- `malloc(...)`: Allocates that many bytes from heap memory
- `(struct Node*)`: Casts the returned void pointer to Node pointer
- **Why malloc?**: Arrays are fixed size; malloc allows dynamic growth

Best practice: Always check if `malloc()` returns `NULL` (memory allocation failure)

Challenge 2

Print Routine Function

Challenge 2: Problem Statement

Task: Write a function to traverse and print all elements of a linked list.

Requirements:

- Function name: `printList`
- Parameter: pointer to head node
- Return type: `void`
- Print format: `data1 → data2 → data3 → NULL`
- Handle empty lists gracefully
- Use proper traversal technique

Time: 15 minutes ⏰

Challenge 2: Pseudocode

```
ALGORITHM PrintList(head)
  1. Handle edge case:
     - If head is NULL:
       * Print "List is empty"
       * Return

  2. Initialize:
     - current = head (start from first node)

  3. Traverse and print:
     - While current is NOT NULL:
       * Print current.data
       * Print " → "
       * Move to next: current = current.next

  4. Print "NULL" to indicate end of list

  5. Print newline

END ALGORITHM
```

Challenge 2: Solution Code

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

struct Node {
    char data[50];
    struct Node* next;
};

// Function to print the linked list
void printList(struct Node* head) {
    // Handle empty list
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    // Start from head
    struct Node* current = head;

    // Traverse until we reach NULL
    while (current != NULL) {
        printf("%s → ", current->data);
        current = current->next;  // Move to next node
    }

    printf("NULL\n");  // Indicate end of list
}
```

Challenge 2: Solution Code (continued)

```c
// Helper function to create nodes
struct Node* createNode(const char* activity) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    strcpy(newNode->data, activity);
    newNode->next = NULL;
    return newNode;
}

// Example usage
int main() {
    // Create daily routine list
    struct Node* head = createNode("Wake Up");
    head->next = createNode("Breakfast");
    head->next->next = createNode("Study");
    head->next->next->next = createNode("Lunch");

    // Print the list
    printf("My Routine: ");
    printList(head);

    return 0;
}
```

Challenge 2: Code Explanation

Part 1: Edge Case Handling

```
if (head == NULL) {
    printf("List is empty\n");
    return;
}
```

Why check for NULL?

- Prevents dereferencing NULL pointer (crashes!)
- Empty list is a valid state
- Provides user-friendly feedback
- Defensive programming practice

Without this check: `current->data` on NULL would cause **segmentation fault**

Challenge 2: Code Explanation

Part 2: Traversal Logic

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

Step-by-step execution:

Iteration	current points to	Action	Output
1	"Wake Up"	Print, move next	`Wake Up →`
2	"Breakfast"	Print, move next	`Breakfast →`
3	"Study"	Print, move next	`Study →`
4	NULL	Exit loop	—

Challenge 2: Code Explanation

Part 3: Why Use 'current' Instead of 'head'?

```
struct Node* current = head;  // Copy the pointer
while (current != NULL) {
    current = current->next;  // Modify the copy
}
```

Two reasons:

1. **Preserve entry point**: We might need `head` later
2. **Safety**: Never modify the head during traversal

Bad practice:

```
while (head != NULL) {
    head = head->next;  // ❌ Lost access to list!
}
// Now head is NULL - can't access list anymore!
```

Challenge 2: Code Explanation

Part 4: The Arrow Operator Explained

```
current = current->next;
```

Understanding `->`:

- **Short for:** `current = (*current).next`
- **Purpose:** Access members of struct through pointer
- **Why needed:** `current` is a pointer, not the struct itself

Comparison:

```
struct Node myNode;       // Actual struct
myNode.next = ...;        // Use . (dot) operator

struct Node* ptr = ...;   // Pointer to struct
ptr->next = ...;          // Use -> (arrow) operator
```

Challenge 3

Count Nodes Function

Challenge 3: Problem Statement

Task: Write a function that counts the total number of nodes in a linked list.

Requirements:

- Function name: `countNodes`
- Parameter: pointer to head node
- Return type: `int` (number of nodes)
- Handle empty lists (return 0)
- Do not modify the list

Example:

- List: `"Wake Up" → "Breakfast" → "Study" → NULL`
- Expected return: `3`

Time: 15 minutes ⏰

Challenge 3: Pseudocode

```
ALGORITHM CountNodes(head)
  1. Initialize counter:
     - count = 0

  2. Handle edge case:
     - If head is NULL:
       * Return 0

  3. Initialize traversal:
     - current = head

  4. Count nodes:
     - While current is NOT NULL:
       * Increment count by 1
       * Move to next: current = current.next

  5. Return count

END ALGORITHM
```

Challenge 3: Solution Code

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

struct Node {
    char data[50];
    struct Node* next;
};

// Function to count nodes in linked list
int countNodes(struct Node* head) {
    // Initialize counter
    int count = 0;

    // Handle empty list
    if (head == NULL) {
        return 0;
    }

    // Start traversal from head
    struct Node* current = head;

    // Count each node
    while (current != NULL) {
        count++;                    // Increment counter
        current = current->next;    // Move to next node
    }

    return count;  // Return total count
}
```

Challenge 3: Solution Code (continued)

```c
// Helper function
struct Node* createNode(const char* activity) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    strcpy(newNode->data, activity);
    newNode->next = NULL;
    return newNode;
}

// Example usage
int main() {
    // Create routine list
    struct Node* head = createNode("Wake Up");
    head->next = createNode("Breakfast");
    head->next->next = createNode("Study");
    head->next->next->next = createNode("Lunch");
    head->next->next->next->next = createNode("Exercise");

    // Count and display
    int total = countNodes(head);
    printf("Total activities: %d\n", total);  // Output: 5

    // Test with empty list
    printf("Empty list count: %d\n", countNodes(NULL));  // Output: 0

    return 0;
}
```

Challenge 3: Code Explanation

Part 1: Counter Variable

```
int count = 0;
```

Why initialize to 0?

- Start with no nodes counted
- Will increment for each node found
- If list is empty, correctly returns 0
- Simple and intuitive

Common mistake: Starting at 1 (assumes list has at least one node)

Challenge 3: Code Explanation

Part 2: Traversal and Counting

```
while (current != NULL) {
    count++;
    current = current->next;
}
```

Execution trace for list `A → B → C → NULL`:

Iteration	current	count before	Action	count after
Start	A	0	—	0
1	A	0	count++	1
2	B	1	count++	2
3	C	2	count++	3
4	NULL	3	Exit loop	3

Challenge 3: Code Explanation

Part 3: Time Complexity Analysis

```
while (current != NULL) {
    count++;                    // O(1) operation
    current = current->next; // O(1) operation
}
```

Complexity: O(n) where n = number of nodes

Why?

- Must visit **every single node** to count
- No shortcuts or random access
- Each node visited exactly once

Compare with arrays:

- Array length: O(1) - stored in array structure

Challenge 3: Code Explanation

Part 4: Alternative Recursive Solution

```
int countNodesRecursive(struct Node* head) {
    // Base case: empty list
    if (head == NULL) {
        return 0;
    }

    // Recursive case: 1 (current node) + count of rest
    return 1 + countNodesRecursive(head->next);
}
```

How it works:

- `countNodes(A → B → C)` = `1 + countNodes(B → C)`
- `countNodes(B → C)` = `1 + countNodes(C)`
- `countNodes(C)` = `1 + countNodes(NULL)`
- `countNodes(NULL)` = `0`

Challenge 4

Find Maximum Characters Task

Challenge 4: Problem Statement

Task: Write a function to find the task (node) with the maximum character count in a linked list of tasks.

Requirements:

- Function name: `findMaxCharTask`
- Parameter: pointer to head node
- Return type: pointer to node with longest string
- Each node contains a task name (string)
- If multiple tasks have same max length, return first occurrence
- Handle empty lists (return NULL)

Example: `"Wake Up" → "Breakfast" → "Study" → NULL`

Return: Node with "Breakfast" (9 characters)

Time: 15 minutes ⏰

Challenge 4: Pseudocode

```
ALGORITHM FindMaxCharTask(head)
  1. Handle edge case:
     - If head is NULL:
       * Return NULL

  2. Initialize tracking variables:
     - maxNode = head (assume first is longest)
     - maxLength = length of head.data
     - current = head

  3. Traverse list:
     - While current is NOT NULL:
       * currentLength = length of current.data
       * If currentLength > maxLength:
         - Update maxLength = currentLength
         - Update maxNode = current
       * Move to next: current = current.next

  4. Return maxNode

END ALGORITHM
```

Challenge 4: Solution Code

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

struct Node {
    char data[50];
    struct Node* next;
};

// Function to find task with maximum characters
struct Node* findMaxCharTask(struct Node* head) {
    // Handle empty list
    if (head == NULL) {
        return NULL;
    }

    // Initialize tracking variables
    struct Node* maxNode = head;
    int maxLength = strlen(head->data);

    // Start traversal
    struct Node* current = head;
```

Challenge 4: Solution Code (continued)

```
    // Traverse entire list
    while (current != NULL) {
        int currentLength = strlen(current->data);

        // Update if we found a longer task
        if (currentLength > maxLength) {
            maxLength = currentLength;
            maxNode = current;
        }

        current = current->next;
    }

    return maxNode;  // Return node with longest string
}

// Helper function
struct Node* createNode(const char* task) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    strcpy(newNode->data, task);
    newNode->next = NULL;
    return newNode;
}
```

Challenge 4: Solution Code (Test Cases)

```
int main() {
    // Create daily routine list
    struct Node* head = createNode("Wake Up");          // 7 chars
    head->next = createNode("Breakfast");               // 9 chars
    head->next->next = createNode("Study");             // 5 chars
    head->next->next->next = createNode("Lunch");       // 5 chars
    head->next->next->next->next = createNode("Exercise");  // 8 chars
    head->next->next->next->next->next = createNode("Dinner");  // 6 chars

    // Find task with maximum characters
    struct Node* maxTask = findMaxCharTask(head);

    if (maxTask != NULL) {
        printf("Task with maximum characters: %s\n", maxTask->data);
        printf("Character count: %zu\n", strlen(maxTask->data));
    }

    // Test with empty list
    struct Node* emptyMax = findMaxCharTask(NULL);
    printf("Empty list result: %s\n", emptyMax == NULL ? "NULL" : "Not NULL");

    return 0;
}
```

Challenge 4: Code Explanation

Part 1: Initialization Strategy

```
struct Node* maxNode = head;
int maxLength = strlen(head->data);
```

Why start with head?

- **Guarantees valid starting point**: List is not empty (checked earlier)
- **Simplifies logic**: Always have a "current best"
- **Handles single-node list**: Automatically correct

What is strlen()?

- Standard C function from `<string.h>`
- Returns number of characters in string (excluding null terminator)
- Example: `strlen("Hello")` returns `5`

Challenge 4: Code Explanation

Part 2: Comparison Logic

```
int currentLength = strlen(current->data);

if (currentLength > maxLength) {
    maxLength = currentLength;
    maxNode = current;
}
```

Two-step update:

1. **Update maxLength**: Track the longest length seen
2. **Update maxNode**: Remember which node has that length

Why `>` instead of `>=`?

- Returns **first occurrence** if there are ties
- Example: `"Study"` (5) appears before `"Lunch"` (5), return `"Study"`

Challenge 4: Code Explanation

Part 3: Execution Trace

List: `"Wake Up" → "Breakfast" → "Study" → "Lunch" → "Exercise"`

Node	Task	Length	maxLength	maxNode	Action
Init	"Wake Up"	7	7	"Wake Up"	Initialize
1	"Breakfast"	9	9	"Breakfast"	9 > 7 ✓ Update
2	"Study"	5	9	"Breakfast"	5 ≤ 9, no change
3	"Lunch"	5	9	"Breakfast"	5 ≤ 9, no change
4	"Exercise"	8	9	"Breakfast"	8 ≤ 9, no change

Result: Returns node containing `"Breakfast"` (9 characters)

Challenge 4: Code Explanation

Part 4: Return Value Explained

```
return maxNode;  // Returns pointer, not the string!
```

Important distinction:

- **Returns:** `struct Node*` (pointer to node)
- **NOT:** `char*` (pointer to string)

Usage in main:

```
struct Node* result = findMaxCharTask(head);
printf("%s\n", result->data);  // Access the string through pointer
```

Why return the whole node?

- Gives access to entire node (data + next)
- Can chain operations on the node

Challenge 4: Code Explanation

Part 5: Edge Cases

Empty List:

```
if (head == NULL) {
    return NULL;
}
```

- **Input:** `NULL`
- **Output:** `NULL`
- **Prevents:** Crash from `strlen(NULL->data)`

Single Node:

- **Input:** `"Hello" → NULL`
- **Output:** Node with `"Hello"`
- **Works because:** Initialize with head, never updated (no comparisons)

Lab Session Complete!

Summary: What You Learned

✅ **Challenge 1:** Created a linked list with string data (daily routine), implemented node structure, and connected nodes

✅ **Challenge 2:** Traversed and printed all elements in list

✅ **Challenge 3:** Counted total nodes using iteration

✅ **Challenge 4:** Found node with maximum string length

Key Concepts Reinforced

Memory Management

- Dynamic allocation with `malloc()`
- Proper initialization of pointers
- Checking for allocation failures

Pointer Operations

- Using `->` operator for struct access
- Linking nodes together
- Traversing using pointer arithmetic

List Traversal

- Starting from head
- Using `current != NULL` condition
- Never modifying head pointer

Common Mistakes to Avoid

❌ Memory Leaks

```
struct Node* node = createNode("Task");
node = createNode("New Task");  // Lost access to first node!
```

❌ Modifying Head

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

❌ Not Checking NULL

```
head->data;  // ❌ Crashes if head is NULL
```

Best Practices You Learned

✅ Always initialize pointers

```
newNode->next = NULL;  // Don't leave it uninitialized
```

✅ Check for NULL before dereferencing

```
if (head == NULL) return;
```

✅ Use helper functions

```
struct Node* createNode(const char* task) { ... }  // Reusable
```

✅ Use descriptive variable names

```
struct Node* current;  // Not 'ptr' or 'temp'
```