Introduction to Linked Lists

Data Structures & Algorithms

Class 01

Topic: What is a Linked List?

Learning Objectives

By the end of this class, you will be able to:

1. Understand what a linked list is and why it exists
2. Explain the fundamental structure of a linked list
3. Differentiate between linked lists and arrays
4. Identify real-world scenarios where linked lists are used
5. Understand basic terminology (node, head, tail, pointer)

A Story to Begin

Treasure Hunt Analogy

Imagine organizing a treasure hunt across your city...

- First clue: **"Go to Rishikesh"**
- At Rishikesh: **"Head to the Old Broken Bridge"**
- Each location points to the next location
- No master list needed! Just follow the chain

This is exactly how a linked list works!

What is a Linked List?

Definition

A **Linked List** is a linear data structure where elements are stored in **nodes**.

Each node contains:

1. **Data** - The actual value/information
2. **Pointer** - The address/link to the next node

Visual Representation

```
[Data|Next] → [Data|Next] → [Data|Next] → NULL
```

Example:

```
[10|•] → [20|•] → [30|•] → NULL
```

- **Head**: First node (where we start)
- **NULL**: End of the list

The Building Block: The Node

Node Analogy

Think of a node as a **train car**:

- It carries **cargo** (data)
- It's **connected** to the next car (pointer)

Node Structure in C

```c
struct Node {
    int data;           // Data part
    struct Node* next;  // Pointer to next node
};

// Function to create a new node
struct Node* createNode(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = NULL;  // Initialize next to NULL
    return newNode;
}
```

Understanding the createNode Function

```
struct Node* createNode(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;          // Initialize data
    newNode->next = NULL;         // Initialize next pointer
    return newNode;               // Return pointer to new node
}
```

- **malloc()** allocates memory dynamically
- **sizeof()** calculates required memory size
- Returns pointer to newly created node

```
struct Node* myNode = createNode(10);  // data=10, next=NULL
```

Real-World Examples

1. Music Playlist

Each song knows which song comes next:

- Skip to next song easily
- Add songs anywhere
- Remove songs without affecting others

```
[Bohemian Rhapsody|•] → [Stairway to Heaven|•]
    → [Hotel California|•] → NULL
```

2. Browser History

Your browser's back button:

- Each page linked to previous page
- Follow links backward
- Chain grows/shrinks as you browse

```
[Google.com|•] ← [Facebook.com|•] ← [YouTube.com|•]
```

3. Train Compartments

Perfect physical analogy:

- Engine (Head node)
- Each compartment connected to next
- Add/remove compartments easily
- Don't need to manufacture together

```
[Engine|•] → [Coach-1|•] → [Coach-2|•] → [Coach-3|•] → NULL
```

4. Blockchain Technology

Cryptocurrencies like Bitcoin:

- Each block contains transaction data
- Each block points to previous block
- Creates unbreakable chain of records

```
[Block-1|•] → [Block-2|•] → [Block-3|•] → [Block-4|•]
```

Let's Build Our First Linked List!

Step-by-Step Example (C)

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

// Step 1: Define the Node structure
struct Node {
    int data;
    struct Node* next;
};

// Helper function to create a node
struct Node* createNode(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = NULL;
    return newNode;
}
```

Creating and Linking Nodes

```c
int main() {
    // Step 2: Create individual nodes
    struct Node* head = createNode(10);      // First node
    struct Node* second = createNode(20);    // Second node
    struct Node* third = createNode(30);     // Third node

    // Step 3: Link the nodes together
    head->next = second;   // First → second
    second->next = third;  // Second → third
    third->next = NULL;    // Third → NULL

    // Now we have: 10 → 20 → 30 → NULL

    return 0;
}
```

Understanding -> and =

```
head->next = second;
```

Part	What it does
`head`	Pointer to a Node
`->`	Access member through pointer
`next`	Member variable being accessed
`=`	Assignment operator
`second`	Value being assigned

Translation: "Access the `next` member of head's Node and assign `second` to it"

Traversing the Linked List

```
void printList(struct Node* head) {
    struct Node* current = head;  // Start from head

    while (current != NULL) {
        printf("%d → ", current->data);
        current = current->next;  // Move to next node
    }

    printf("NULL\n");
}

// Usage: printList(head);
// Output: 10 → 20 → 30 → NULL
```

Key Terminology

Essential Terms

Term	Definition	Example
Node	Single element with data + pointer	One train car
Head	First node in the list	Train engine
Tail	Last node (points to NULL)	Last compartment
Pointer	Link to next node	Coupling between cars
NULL	End of list indicator	End of track
Traversal	Moving through list node by node	Walking through train

Why Do We Need Linked Lists?

Arrays Have Limitations

- ❌ **Fixed Size** - Must know size in advance
- ❌ **Costly Insertions/Deletions** - Requires shifting elements
- ❌ **Wasted Memory** - Allocate space you might not use
- ❌ **One Block Allocation** - Might not get large contiguous memory

Linked Lists Offer

- ✅ **Dynamic Size** - Grows and shrinks as needed
- ✅ **Efficient Insertions/Deletions** - Just change pointers
- ✅ **No Memory Wastage** - Allocate exactly what you need
- ✅ **Scattered Memory** - Don't need contiguous block

Key Properties of Linked Lists

- Successive elements connected by **pointers**
- Last element points to **NULL**
- Can **grow or shrink** during execution
- Can be as long as required (until memory exhausts)
- **No wasted memory** (but uses extra space for pointers)
- Allocates memory **as list grows**

Arrays vs Linked Lists

Quick Comparison

Feature	Array	Linked List
Memory	Contiguous	Scattered
Size	Fixed	Dynamic
Access Time	O(1) - Direct	O(n) - Sequential
Insert at Start	O(n) - Must shift	O(1) - Change pointer
Memory Overhead	None	Pointer storage

When to Use Each?

Use Arrays when:

- Need fast random access to elements
- Size is known and stable
- Memory is contiguous
- Frequent random access operations

Use Linked Lists when:

- Size changes frequently
- Frequent insert/delete at beginning or middle
- Don't need random access
- Memory fragmentation is acceptable

Thought-Provoking Questions

Challenge Questions

1. How would you access the 100th element in a linked list?
2. Can a node point to itself? What would happen?
3. What happens if we lose the reference to the Head node?
4. How is a linked list different from a chain of paper clips?

The 100th Element Question

Answer: You **cannot** directly jump to it!

You must:

1. Start at head (1st node)
2. Follow `next` pointer 99 times
3. Arrive at 100th node

Time complexity: O(n) - Linear time

Array: O(1) - Constant time with `arr[99]`

Code to Access nth Element

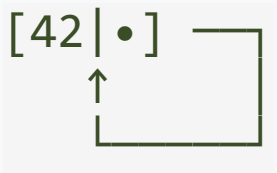
```
struct Node* getNodeAt(struct Node* head, int position) {
    if (head == NULL || position < 1) {
        return NULL;
    }

    struct Node* current = head;
    int count = 1;

    while (current != NULL && count < position) {
        current = current->next;
        count++;
    }

    return current;
}
```

Question 2: Can a Node Point to Itself?

Answer: Yes, technically it can - this creates a **circular reference**!

```
struct Node* selfNode = createNode(42);
selfNode->next = selfNode;  // Points to itself!
```

What happens?



Consequences of Self-Referencing Node

Traversal becomes infinite!

```
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {  // This NEVER becomes NULL!
        printf("%d → ", current->data);
        current = current->next;  // Always points back to itself
    }
    // INFINITE LOOP! Program hangs!
}
```

Result: The loop never terminates because `current` always points to the same node.

Detecting Circular References

Solution: Use two pointers (Floyd's Cycle Detection Algorithm)

```
int hasCycle(struct Node* head) {
    struct Node* slow = head;
    struct Node* fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;          // Move 1 step
        fast = fast->next->next;    // Move 2 steps

        if (slow == fast) {
            return 1;  // Cycle detected! (true)
        }
    }
    return 0;  // No cycle (false)
}
```

Question 3: Losing Head Reference

Answer: The entire list becomes **inaccessible** and causes a **memory leak**!

```
struct Node* head = createNode(10);
head->next = createNode(20);
head->next->next = createNode(30);

head = NULL;  // LOST THE HEAD!
```

Problem: The nodes still exist in memory, but we have no way to reach them!

Memory Leak Visualization

Before losing head:

```
head ──> [10|•] → [20|•] → [30|NULL]
```

After head = NULL:

```
head = NULL

         [10|•] → [20|•] → [30|NULL]
         ↑
         Lost in memory - cannot be accessed or freed!
```

Result: Memory leak - allocated memory cannot be reclaimed.

Proper List Deletion

Solution: Always free all nodes before losing the head reference!

```
void deleteList(struct Node** head) {
    struct Node* current = *head;
    struct Node* nextNode;

    while (current != NULL) {
        nextNode = current->next;  // Save next node
        free(current);             // Free current node
        current = nextNode;        // Move to next
    }

    *head = NULL;  // Now safe to set to NULL
}
```

Question 4: Linked List vs Paper Clips

Key Differences:

Aspect	Linked List	Paper Clip Chain
Connection	Pointer (memory address)	Physical interlocking
Direction	One-way (in singly linked)	Can go both ways
Access	Must start from head	Can start anywhere
Breaking Chain	Lose access to rest	Can still access both parts
Data Storage	Each node has data + pointer	Clips are just connections

Deeper Analogy Comparison

Linked List is MORE like:

A scavenger hunt with clues:

- Each clue (node) tells you where the next clue is
- You MUST start at the first clue (head)
- Each clue contains information (data)
- If you lose the first clue, you're stuck!

Paper Clip Chain is LESS accurate because:

- You can pick it up anywhere (random access)
- It's bidirectional (can traverse both ways)
- No inherent "data" at each clip