# 📚 DSA Topic: Linked List (Complete Guide)

## 1. What is a Linked List?

**The Analogy:** Imagine a treasure hunt.

* **Array:** A row of lockers numbered 1, 2, 3. You can go to locker #50 instantly ($O(1)$) because you know exactly where it is using math.
* **Linked List:** You have a box (**Node**). Inside, there is a piece of paper (**Pointer**) telling you where the *next* box is hidden. You **must** go to the first box to find the second, and so on.

### Key Characteristics

* **Non-Contiguous Memory:** Unlike arrays (which sit together in memory blocks), Linked List nodes are scattered everywhere in the Heap memory. The "pointers" are the only things connecting them.
* **Dynamic Size:** No need to declare size upfront (like int[10]). You can create new nodes endlessly until memory runs out.

## 2. The Three Types

1. **Singly Linked List:** One-way street. You can only move forward (Next).
   * *End:* The last node points to null.
2. **Doubly Linked List:** Two-way street. Every node has a Next and a Previous pointer.
   * *Use case:* Music players (Next song, Previous song), Browser History.
3. **Circular Linked List:** An infinite loop. The Tail (last node) points back to the Head (first node) instead of null.

## 3. 🧠 Complexity Analysis (The "Why")

| **Operation** | **Complexity** | **Reason** |
| --- | --- | --- |
| **Search / Access** | $O(N)$ | **No Random Access.** You cannot calculate the address of Node 5. You must ask Node 1 for Node 2, Node 2 for Node 3, etc. |
| **Insert at Start** | $O(1)$ | You just change the head pointer. No shifting required. |
| **Insert at End** | $O(N)$ | You have to walk (traverse) the entire list to find the last node first. |
| **Delete at Start** | $O(1)$ | Just move head to head.next. |
| **Delete at End** | $O(N)$ | You have to walk to the *second to last* node to disconnect the last one. |
| **Get Size** | $O(1)$ | *Optimized:* We maintain a size variable that updates on every add/delete. |

## 4. The Golden Rules (Crucial for Interviews)

### Rule #1: The "Head is Sacred" 👑

* **The Problem:** head is your **only** anchor to the list. If you move it (head = head.next) during a traversal, you lose the start of the list forever. The Garbage Collector will delete the unreachable nodes.
* **The Solution:** Always create a temporary "Scout" pointer (Node currNode = head) to walk through the list. The head pointer should never move unless you are explicitly adding/removing the first node.

### Rule #2: Always Check for Null

* Before doing head.next or curr.next, ask yourself: *"What if the list is empty?"*
* Accessing next on a null object causes the dreaded NullPointerException.

### Rule #3: Tracking Size

* **Rookie Mistake:** Calculating size by looping through the list every time ($O(N)$).
* **Pro Approach:** Declare a private int size; variable. Increment on Add, Decrement on Delete. This makes getSize() instant ($O(1)$).

## 5. 💻 Part A: Manual Implementation (For Interviews)

*This is the code you write when an interviewer asks: "Implement a Linked List from scratch."*

public class LL {  
  
 // Encapsulation: Node is private so only LL class can touch it  
 private class Node {  
 String data;  
 Node next;  
  
 Node(String data) {  
 this.data = data;  
 this.next = null;  
 }  
 }  
  
 private Node head;  
 private int size; // Optimization: Track size  
  
 // Constructor  
 public LL() {  
 this.size = 0;  
 }  
  
 // --- ADD OPERATIONS ---  
 public void addFirst(String data) {  
 Node newNode = new Node(data);  
 size++;   
 if (head == null) {  
 head = newNode;  
 return;  
 }  
 newNode.next = head; // 1. Link new node to old head  
 head = newNode; // 2. Move head pointer  
 }  
  
 public void addLast(String data) {  
 Node newNode = new Node(data);  
 size++;  
 if (head == null) {  
 head = newNode;  
 return;  
 }  
 Node currNode = head; // Use a 'Scout' pointer  
 while (currNode.next != null) {   
 currNode = currNode.next;  
 }  
 currNode.next = newNode;  
 }  
  
 // --- DELETE OPERATIONS ---  
 public void deleteFirst() {  
 if (head == null) {  
 System.out.println("List is empty");  
 return;  
 }  
 size--;  
 head = head.next; // Move head forward  
 }  
  
 public void deleteLast() {  
 if (head == null) {  
 System.out.println("List is empty");  
 return;  
 }  
 size--;  
 if (head.next == null) { // Only one node  
 head = null;  
 return;  
 }  
 Node secondLast = head;  
 while (secondLast.next.next != null) { // Stop at 2nd to last  
 secondLast = secondLast.next;  
 }  
 secondLast.next = null;  
 }  
  
 // --- UTILITY ---  
 public void printList() {  
 if (head == null) {  
 System.out.println("List is empty");  
 return;  
 }  
 Node currNode = head;  
 while (currNode != null) {  
 System.out.print(currNode.data + " -> ");  
 currNode = currNode.next;  
 }  
 System.out.println("NULL");  
 }  
  
 public static void main(String args[]) {  
 LL list = new LL();  
 list.addFirst("a");  
 list.addFirst("is");  
 list.addLast("list");  
 list.printList(); // Output: is -> a -> list -> NULL  
 }  
}

## 6. 💻 Part B: Java Collections Framework (For Development)

*This is the built-in Java class you use in real projects or online assessments where you don't need to implement the Node class yourself.*

**Class:** java.util.LinkedList

import java.util.LinkedList;   
  
public class LLUsingCollection {  
 public static void main(String args[]) {  
 // 1. Creation  
 // Syntax: LinkedList<ObjectType> name = new LinkedList<>();  
 // Note: Use 'Integer' not 'int', 'Character' not 'char' (Generics rules)  
 LinkedList<String> list = new LinkedList<String>();  
  
 // 2. Add Operations  
 list.addFirst("a");  
 list.addFirst("is");  
 list.addLast("list");  
 list.add("of"); // .add() defaults to addLast  
   
 System.out.println(list); // Output: [is, a, list, of]  
  
 // 3. Size Operation  
 System.out.println("Size: " + list.size()); // Optimized O(1) internally  
  
 // 4. Delete Operations  
 list.removeFirst();   
 list.removeLast();  
 list.remove(1); // Remove object at index 1  
   
 System.out.println(list); // Output: [a]  
 }  
}

### ⚡ Comparison: Scratch vs. Collections

| **Feature** | **Scratch Implementation (LL.java)** | **Java Collections (java.util.LinkedList)** |
| --- | --- | --- |
| **Setup** | Manually defined Node class. | Just import java.util.LinkedList; |
| **Printing** | Had to write a while loop to traverse. | Just System.out.println(list); |
| **Accessing** | We traversed manually. | list.get(i) (Note: This is still $O(N)$ internally). |
| **Deleting** | Handled edge cases (null checks) manually. | removeFirst() throws an Exception if list is empty! |
| **Data Types** | We hardcoded String. | Uses **Generics** (<String>, <Integer>), so it works for any type. |

## 7. Mistakes I Made (Review Checklist)

1. **Duplicate Declarations:** I accidentally declared head twice.
2. **Forgetting Size Updates:** If you maintain a size variable, you **must** update it in *every* Add and Delete function.
3. **Delete Last Logic:** I learned that curr.next != null stops at the last node, but to delete it, I need to stop at the *second-to-last* node using curr.next.next != null.