📚 Mastering Linked Lists – From Basics to Advanced

🔗 What is a Linked List?

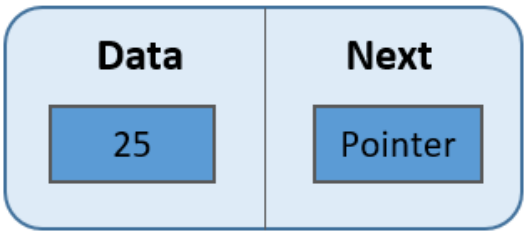
A Linked List is a way of organizing data where each item, called a node, holds two things — the actual data and a reference to the next item in the list.

Think of it like a chain of people, where each person is holding something (the data) and also pointing to the next person in line. The important part is: unlike arrays, these people (or nodes) aren't standing next to each other — they could be scattered all around, but as long as each one knows who's next, the chain stays connected.

So, each node has:

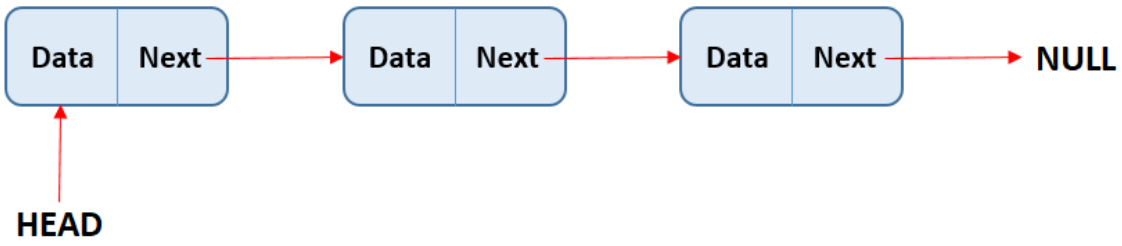
- A part that stores the data.
- A part that stores a pointer (or reference) to the next node.

This setup allows the linked list to grow or shrink easily, without needing a large block of continuous memory.



This image shows how a Linked List works in a very simple way.

- It all starts with the HEAD, which is just a reference to the first node in the list. Think of it like a starting point — without the HEAD, you can't access the list.
- Each node in the list has two parts:
    1. Data – which stores the actual information (like a number, name, etc.).
    2. Next – which holds the address (or pointer) of the next node in the chain.
- The red arrows in the image represent the connections between the nodes — each node points to the next one.
- This continues until the last node, which doesn't point to anything further. Instead, its "Next" part contains NULL, meaning "end of the list."



🔠 Node Structure

Each node in a singly linked list contains:

- **Data** – The value stored.
- **Next** – A pointer/reference to the next node.

```
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

🔗 Types of Linked Lists

1. Singly Linked List – Each node points to the next. (→)

- Insert at Beginning - Time Complexity: O(1)

```
newNode.next = head;
head = newNode;
```

- Insert at End - Time Complexity: O(n)*

```
tail.next = newNode;
tail = newNode;
```

- Delete from Beginning - Time Complexity: O(1)

```
head = head.next;
```

- Delete from End - Time Complexity: O(n)

```
while (temp.next.next != null) temp = temp.next;
temp.next = null;
```

- Search - Time Complexity: O(n)

```
while (temp != null) {
    if (temp.data == key) return true;
    temp = temp.next;
}
```

- Traverse - Time Complexity: O(n)

```
Node temp = head;
while (temp != null) {
    System.out.print(temp.data + " ");
    temp = temp.next;
}
```

*** O(1) if using tail pointer**

2. Doubly Linked List – Each node has both next and prev pointers. (↔)

- Insert at Beginning - Time Complexity: O(1)

```
newNode.next = head;
head.prev = newNode;
head = newNode;
```

- Insert at End - Time Complexity: O(1)*

```
tail.next = newNode;
newNode.prev = tail;
tail = newNode;
```

- Delete from Beginning - Time Complexity: O(1)

```
head = head.next;
head.prev = null;
```

- Delete from End - Time Complexity: O(1)

```
tail = tail.prev;
tail.next = null;
```

- Search - Time Complexity: O(n)

```
while (temp != null) {
    if (temp.data == key) return true;
    temp = temp.next;
}
```

- Traverse Forward - Time Complexity: O(n)

```
for (DNode temp = head; temp != null; temp = temp.next)
    System.out.print(temp.data + " ");
```

- Traverse Backward - Time Complexity: O(n)

```
for (DNode temp = tail; temp != null; temp = temp.prev)
    System.out.print(temp.data + " ");
```

*** O(1) if tail maintained**

3. Circular Linked List

   - Singly Circular: Last node points to the head. (🔁)
     - Insert at End - Time Complexity: O(1)

```
tail.next = newNode;
newNode.next = head;
tail = newNode;
```

     - Delete from Beginning - Time Complexity: O(1)

```
head = head.next;
tail.next = head;
```

     - Traverse - Time Complexity: O(n)

```
Node temp = head;
do {
    System.out.print(temp.data + " ");
    temp = temp.next;
} while (temp != head);
```

- Doubly Circular: Both ends connected forming a circle.
  - Insert at End - Time Complexity: O(1)

```
tail.next = newNode;
newNode.prev = tail;
newNode.next = head;
head.prev = newNode;
tail = newNode;
```

  - Delete from Beginning - Time Complexity: O(1)

```
head = head.next;
head.prev = tail;
tail.next = head;
```

  - Delete from End - Time Complexity: O(1)

```
tail = tail.prev;
tail.next = head;
head.prev = tail;
```

  - Traverse Forward - Time Complexity: O(n)

```
DCNode temp = head;
do {
    System.out.print(temp.data + " ");
    temp = temp.next;
} while (temp != head);
```

🧠 Terminology

- **Head** – First node in the list.
- **Tail** – Last node, typically points to null (or head in circular list).
- **Traversal** – Visiting each node.
- **Insertion** – Adding nodes.
- **Deletion** – Removing nodes.
- **Search** – Finding nodes with a specific value.

🔍 Advanced Understanding of Linked Lists

This section helps you understand how linked lists work in real life — when to use them, where they perform better than arrays, and what common mistakes to avoid. It's about seeing the why behind the code.

1️⃣ Linked List vs Array – A Deep Dive

Arrays and linked lists are both linear data structures, but they function very differently under the hood.

Feature	Array	Linked List
Memory Allocation	Contiguous	Non-contiguous
Access Time	O(1) (Random)	O(n) (Sequential)
Insertion/Deletion	Costly (O(n))	Efficient (O(1) at head)
Cache Performance	High	Low
Resizing	Fixed/Resize cost	Dynamic (flexible)

📌 **Use Case Tip:** Use arrays when you need frequent indexed access; use linked lists when insert/delete operations dominate.

2️⃣ Real-World Applications of Linked Lists

Linked lists are everywhere — here's how they're used:

Use Case	Type of List	Description
Music/Video Playlists	Doubly Linked List	Navigate forward/backward
Undo/Redo in Editors	Doubly Linked List	Reversible actions
Operating Systems (Memory)	Singly Linked List	Manage free blocks
LRU Cache Implementation	Doubly Linked List + HashMap	Fast eviction
Round-Robin Task Scheduling	Circular Linked List	Tasks run in cycles

3️⃣ Why Linked Lists Are a Favorite in Interviews

- **Pointer management**: Interviewers test how well you handle references and memory.
- **Edge cases**: Think nulls, empty lists, one-node lists.
- **Recursion vs iteration**: Tests logical flow handling.
- **Follow-ups**: Easy to build on—merge, reverse, detect cycles.

💡 *Pro Tip:* Always clarify whether the list is singly or doubly linked, and whether it's circular, before solving.

4️⃣ Types of Linked Lists: Comparison and Trade-offs

Type	Forward	Backward	Circular	Memory Cost	Use Case Example
Singly Linked List	✅	❌	❌	Low	Stack
Doubly Linked List	✅	✅	❌	Medium	Undo/Redo, LRU
Circular SLL	✅	❌	✅	Low	Round Robin
Circular DLL	✅	✅	✅	High	Cyclic Navigation

5️⃣ Pointer Manipulation Concepts

Pointers (or references) are the heart of a linked list.

- next → points to the next node
- prev → in DLL, points to previous node
- null → denotes end (or start in some circular types)

⚠️ Common Mistakes:

- Skipping a node by misplacing next = next.next
- Losing head reference → causes memory leaks
- Infinite loops from improper circular handling

6️⃣ Linked Lists in System Design

Linked lists are often embedded inside more complex data structures:

Data Structure	Role of Linked List
Stack (LL-based)	Push/Pop at head (O(1))
Queue (LL-based)	Enqueue at tail, Dequeue at head
HashMap (Chaining)	Handle collisions via separate chaining
Graph (Adjacency List)	Maintain neighbor nodes

📌 **Bonus:** LRU Cache uses DLL + HashMap for O(1) access & eviction.

7️⃣ Time and Space Complexity Table

Operation	Singly LL	Doubly LL	Array
Insert at Head	O(1)	O(1)	O(n)
Insert at Tail	O(n)*	O(1)*	O(n)
Delete at Head	O(1)	O(1)	O(n)
Delete at Tail	O(n)	O(1)	O(n)
Search	O(n)	O(n)	O(1)
Access by Index	O(n)	O(n)	O(1)

***With maintained tail pointer**

8️⃣ Static vs Dynamic Memory

- **Arrays**: Stored in stack (static memory)
- **Linked Lists**: Stored in heap (dynamic memory)
- In C/C++, manual memory management with malloc()/free()
- In Java/Python, handled by Garbage Collector

💡 **Tip:** Linked lists give more control but can lead to memory leaks if not handled carefully (especially in C/C++).

9️⃣ When Not to Use a Linked List

Avoid linked lists if:

- You need random indexed access (arrays are O(1))
- Memory is tight (linked lists use extra space for pointers)
- Cache performance is critical (linked lists are scattered in memory)

🔟 Common Pitfalls and How to Avoid Them

Mistake	Solution
Infinite loop in circular list	Always set termination condition
Losing head reference	Use a dummy node or temp variable
Skipping node on delete	Ensure correct next linkage
Modifying while traversing	Use two-pointer or dummy technique
Forgetting to handle null pointers	Always null-check before access

🔧 Advanced Linked List Operations (with Code Snippets)

These operations are frequently asked in interviews and competitive programming. Below are concise Java code snippets with time and space complexities.

1️⃣ Reverse a Linked List

```
Node reverse(Node head) {
    Node prev = null, curr = head;
    while (curr != null) {
        Node next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
// Time: O(n), Space: O(1)
```

2️⃣ Detect Cycle in Linked List (Floyd's Algorithm)

```
boolean hasCycle(Node head) {
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true;
    }
    return false;
}
// Time: O(n), Space: O(1)
```

3️⃣ Find the Middle Node

```
Node findMiddle(Node head) {
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
// Time: O(n), Space: O(1)
```

4️⃣ Check if a Linked List is a Palindrome

```
boolean isPalindrome(Node head) {
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    Node secondHalf = reverse(slow);
    while (secondHalf != null) {
        if (head.data != secondHalf.data) return false;
        head = head.next;
        secondHalf = secondHalf.next;
    }
    return true;
}
// Time: O(n), Space: O(1)
```

5 Merge Two Sorted Linked Lists

```
Node merge(Node l1, Node l2) {
    if (l1 == null) return l2;
    if (l2 == null) return l1;
    if (l1.data < l2.data) {
        l1.next = merge(l1.next, l2);
        return l1;
    } else {
        l2.next = merge(l1, l2.next);
        return l2;
    }
}
// Time: O(n + m), Space: O(n + m) – recursive stack
```

6 Remove N-th Node from End

```
Node removeNthFromEnd(Node head, int n) {
    Node dummy = new Node(0);
    dummy.next = head;
    Node fast = dummy, slow = dummy;
    for (int i = 0; i <= n; i++) fast = fast.next;
    while (fast != null) {
        fast = fast.next;
        slow = slow.next;
    }
    slow.next = slow.next.next;
    return dummy.next;
}
// Time: O(n), Space: O(1)
```

7 Find Intersection Point of Two Linked Lists

```
Node getIntersection(Node a, Node b) {
    Node p1 = a, p2 = b;
    while (p1 != p2) {
        p1 = (p1 == null) ? b : p1.next;
        p2 = (p2 == null) ? a : p2.next;
    }
    return p1;
}
// Time: O(n + m), Space: O(1)
```

8 Clone Linked List with Random Pointer

```
Node copyRandomList(Node head) {
    if (head == null) return null;
    Node curr = head;
    while (curr != null) {
        Node copy = new Node(curr.data);
        copy.next = curr.next;
        curr.next = copy;
        curr = copy.next;
    }
    curr = head;
    while (curr != null) {
        if (curr.random != null)
            curr.next.random = curr.random.next;
        curr = curr.next.next;
    }
    Node dummy = new Node(0), copyCurr = dummy;
    curr = head;
    while (curr != null) {
        copyCurr.next = curr.next;
        curr.next = curr.next.next;
        curr = curr.next;
        copyCurr = copyCurr.next;
    }
    return dummy.next;
}
// Time: O(n), Space: O(1)
```

9️⃣ Rotate Linked List by K

```
Node rotateRight(Node head, int k) {
    if (head == null) return null;
    int len = 1;
    Node tail = head;
    while (tail.next != null) {
        tail = tail.next;
        len++;
    }
    k %= len;
    if (k == 0) return head;
    tail.next = head; // make circular
    for (int i = 0; i < len - k; i++) tail = tail.next;
    Node newHead = tail.next;
    tail.next = null;
    return newHead;
}
// Time: O(n), Space: O(1)
```

🔟 Flatten Multilevel Linked List

```
Node flatten(Node head) {
    Node curr = head;
    while (curr != null) {
        if (curr.child != null) {
            Node temp = curr.next;
            curr.next = curr.child;
            Node tail = curr.child;
            while (tail.next != null) tail = tail.next;
            tail.next = temp;
            curr.child = null;
        }
        curr = curr.next;
    }
    return head;
}
// Time: O(n), Space: O(1)
```

📘 Basic Linked List Problems

#	Problem Title	Platform	Link	Key Concepts
1	Reverse Linked List	LeetCode – Easy	🔗 Link	Iteration, Recursion
2	Delete Node in a Linked List	LeetCode – Medium	🔗 Link	Deletion
3	Middle of the Linked List	LeetCode – Easy	🔗 Link	Fast/Slow Pointer
4	Merge Two Sorted Lists	LeetCode – Easy	🔗 Link	Recursion
5	Remove Linked List Elements	LeetCode – Easy	🔗 Link	Deletion
6	Palindrome Linked List	LeetCode – Easy	🔗 Link	Reverse, Compare
7	Convert Binary Number in LL to Integer	LeetCode – Easy	🔗 Link	Traversal
8	Linked List Cycle	LeetCode – Easy	🔗 Link	Floyd's Algorithm
9	Remove Duplicates from Sorted List	LeetCode – Easy	🔗 Link	Basic Traversal
10	Search in Linked List	GFG – Easy	🔗 Link	Traversal

🚀 Advanced Linked List Problems

#	Problem Title	Platform	Link	Key Concepts
1	Linked List Cycle II	LeetCode – Medium	🔗 Link	Detect Cycle Start
2	Intersection of Two Linked Lists	LeetCode – Easy	🔗 Link	Length Diff, Pointers
3	Rotate List	LeetCode – Medium	🔗 Link	Modulo, Tail Handling
4	Remove N-th Node from End	LeetCode – Medium	🔗 Link	Two Pointers
5	Add Two Numbers	LeetCode – Medium	🔗 Link	Math in Linked List
6	Copy List with Random Pointer	LeetCode – Medium	🔗 Link	Cloning, Hashing
7	Flatten a Multilevel Doubly Linked List	LeetCode – Medium	🔗 Link	Flattening, Recursion
8	Reverse Nodes in k-Group	LeetCode – Hard	🔗 Link	Recursion, k-Group
9	Merge k Sorted Lists	LeetCode – Hard	🔗 Link	Min Heap
10	LRU Cache	LeetCode – Medium	🔗 Link	DLL + HashMap

📁 GitHub DSA Repository (by Harsh Raj)

🔗 BasicLinkedListOperations.java

🔗 LinkedListProblems.java

👉 https://github.com/eh-harsh02/dsa