Linked List in Java

A **Linked List** is a **linear data structure** where elements (called **nodes**) are **connected using pointers** (references).
Each node contains:

1. **Data** — the actual value
2. **Next** — reference (link) to the next node

Unlike arrays, linked lists **don't use contiguous memory** — elements are scattered in memory and connected by links.

Structure of a Node

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

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

Basic Linked List Example

```
public class LinkedListExample {
    Node head; // first node reference

    // Node class (inner class)
    class Node {
        int data;
        Node next;

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

    // Insert at end
    void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null)
            temp = temp.next;
        temp.next = newNode;
    }

    // Display Linked List
    void display() {
        Node temp = head;
        while (temp != null) {
```

```
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");
    }

    // Main function
    public static void main(String[] args) {
        LinkedListExample list = new LinkedListExample();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.display();  // Output: 10 -> 20 -> 30 -> null
    }
}
```

Operations on Linked List

Operation	Description	Time Complexity
Traversal	Visit all nodes	O(n)
Insertion (beginning)	Add at head	O(1)
Insertion (end)	Add at tail	O(n)
Deletion (beginning)	Remove first node	O(1)
Deletion (end)	Remove last node	O(n)
Search	Find element	O(n)

Linked List vs Array

Feature	Array	Linked List
Memory	Contiguous	Non-contiguous
Size	Fixed	Dynamic
Access (index)	O(1)	O(n)
Insertion/Deletion	Costly (shift elements)	Easy (adjust links)
Extra Space	None	Extra for pointer (`next`)

Types of Linked Lists

Type	Description	Structure
Singly Linked List	Each node points to the next node	`A → B → C → null`
Doubly Linked List	Each node has `prev` and `next`	`null ← A ⇄ B ⇄ C → null`
Circular Linked List	Last node points to first node	`A → B → C ↺`

Using Built-in Java LinkedList Class

Java provides a ready-made **LinkedList** class in the `java.util` package.

```
import java.util.LinkedList;

public class BuiltInLinkedList {
    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();

        list.add(10);
        list.add(20);
        list.addFirst(5);
        list.addLast(30);

        System.out.println(list); // [5, 10, 20, 30]

        list.remove(2);           // remove element at index 2
        System.out.println(list); // [5, 10, 30]
    }
}
```

Key Methods:
`add(), addFirst(), addLast(), remove(), removeFirst(), removeLast(), get(), contains(), size()`

Singly Linked List (SLL)

A **Singly Linked List** is a linear data structure where:

- Each node contains **data** and a **reference (next)** to the next node.
- The **last node's next pointer = null**.

Structure:
`Head → [Data|Next] → [Data|Next] → ... → null`

Node Structure

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

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

```
    }
}
```

Complete Example

```
public class SinglyLinkedList {
    Node head;

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

    // INSERT at end
    void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null)
            temp = temp.next;
        temp.next = newNode;
    }

    // INSERT at beginning
    void insertAtBeginning(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }

    // DELETE a node by value
    void delete(int key) {
        Node temp = head, prev = null;

        // if head node holds the key
        if (temp != null && temp.data == key) {
            head = temp.next;
            return;
        }

        // search for key to delete
        while (temp != null && temp.data != key) {
            prev = temp;
            temp = temp.next;
        }

        // if key not found
        if (temp == null) return;

        // unlink the node
        prev.next = temp.next;
    }
```

```
    // SEARCH a value
    boolean search(int key) {
        Node temp = head;
        while (temp != null) {
            if (temp.data == key)
                return true;
            temp = temp.next;
        }
        return false;
    }

    // TRAVERSE (display)
    void traverse() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");
    }

    // MAIN
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        list.insertAtEnd(10);
        list.insertAtEnd(20);
        list.insertAtEnd(30);
        list.insertAtBeginning(5);

        System.out.print("Linked List: ");
        list.traverse(); // 5 -> 10 -> 20 -> 30 -> null

        list.delete(20);
        System.out.print("After Deletion: ");
        list.traverse(); // 5 -> 10 -> 30 -> null

        System.out.println("Search 10: " + list.search(10)); // true
        System.out.println("Search 50: " + list.search(50)); // false
    }
}
```

Operations Summary

Operation	Description	Time Complexity
Insertion at Beginning	Add node at start	O(1)
Insertion at End	Add node at last	O(n)
Deletion by Value	Remove node by key	O(n)
Search	Check if value exists	O(n)
Traverse	Print all nodes	O(n)

Advantages

- Dynamic size (no fixed length like arrays)
- Easy insertion/deletion (no shifting of elements)

Disadvantages

- No random access (must traverse sequentially)
- Extra memory for `next` pointer
- Reverse traversal not possible (use Doubly Linked List instead)

Doubly Linked List in Java

What is a Doubly Linked List?

A **Doubly Linked List (DLL)** is a linear data structure in which each node contains:

- **data** → value of the node
- **prev** → reference to the previous node
- **next** → reference to the next node

It allows **traversal in both directions** — forward and backward.

Node Structure

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

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

DoublyLinkedList Class

```
class DoublyLinkedList {
    Node head;

    // Insert at the Beginning
    void insertAtBeginning(int data) {
        Node newNode = new Node(data);
        if (head != null) {
            newNode.next = head;
            head.prev = newNode;
        }
        head = newNode;
    }

    // Insert at the End
    void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
```

```
        head = newNode;
        return;
    }
    Node temp = head;
    while (temp.next != null)
        temp = temp.next;
    temp.next = newNode;
    newNode.prev = temp;
}

// Delete a Node (by value)
void deleteNode(int key) {
    Node temp = head;

    // Case 1: Empty list
    if (temp == null)
        return;

    // Case 2: Head node has the key
    if (temp.data == key) {
        head = temp.next;
        if (head != null)
            head.prev = null;
        return;
    }

    // Case 3: Search for the node
    while (temp != null && temp.data != key)
        temp = temp.next;

    // Node not found
    if (temp == null) return;

    // Adjust the pointers
    if (temp.next != null)
        temp.next.prev = temp.prev;
    if (temp.prev != null)
        temp.prev.next = temp.next;
}

// Search for an element
boolean search(int key) {
    Node temp = head;
    while (temp != null) {
        if (temp.data == key)
            return true;
        temp = temp.next;
    }
    return false;
}

// Traverse Forward
void traverseForward() {
    Node temp = head;
    System.out.print("Forward: ");
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}
```

```
    // Traverse Backward
    void traverseBackward() {
        if (head == null) return;
        Node temp = head;
        // Go to last node
        while (temp.next != null)
            temp = temp.next;
        System.out.print("Backward: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.prev;
        }
        System.out.println();
    }
}
```

Main Class (Test the DLL)

```
public class Main {
    public static void main(String[] args) {
        DoublyLinkedList dll = new DoublyLinkedList();

        dll.insertAtEnd(10);
        dll.insertAtEnd(20);
        dll.insertAtEnd(30);
        dll.insertAtBeginning(5);

        dll.traverseForward();   // Output: Forward: 5 10 20 30
        dll.traverseBackward();  // Output: Backward: 30 20 10 5

        System.out.println("Search 20: " + dll.search(20)); // true

        dll.deleteNode(10);
        dll.traverseForward();   // Output: Forward: 5 20 30
    }
}
```

Time Complexity

Operation	Time Complexity
Insert at Beginning	O(1)
Insert at End	O(n)
Delete a Node	O(n)
Search	O(n)
Traverse	O(n)

Advantages

- Traversal in both directions (forward & backward)
- Easy deletion of a given node

Disadvantages

- Requires extra space for `prev` pointer

- Slightly more complex implementation

Circular Linked List (CLL)

A **Circular Linked List** is similar to a singly linked list, but the **last node points back to the first node**, forming a **circle**.

It can be of two types:

1. **Singly Circular Linked List (SCLL)** → last node's `next` points to head.
2. **Doubly Circular Linked List (DCLL)** → last node's `next` points to head, and head's `prev` points to last.

Here we'll focus on the **Singly Circular Linked List** (most commonly used in DSA).

Node Structure

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

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

CircularLinkedList Class

```
class CircularLinkedList {
    Node head = null;
    Node tail = null;

    // Insert at the End
    void insertAtEnd(int data) {
        Node newNode = new Node(data);

        if (head == null) {
            head = newNode;
            tail = newNode;
            tail.next = head; // circular connection
        } else {
            tail.next = newNode;
            tail = newNode;
            tail.next = head;
        }
    }

    // Insert at the Beginning
    void insertAtBeginning(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            tail = newNode;
            tail.next = head;
        } else {
            newNode.next = head;
            head = newNode;
            tail.next = head;
```

```
        }
    }

    // Delete a Node
    void deleteNode(int key) {
        if (head == null) return;

        Node current = head, prev = null;

        // If head node holds the key
        if (current.data == key) {
            // Single node case
            if (current.next == head) {
                head = null;
                tail = null;
                return;
            }

            head = head.next;
            tail.next = head;
            return;
        }

        // Traverse to find the node
        do {
            prev = current;
            current = current.next;
        } while (current != head && current.data != key);

        // Node not found
        if (current == head) return;

        prev.next = current.next;
        // If deleting tail
        if (current == tail) {
            tail = prev;
        }
    }

    // Search a Node
    boolean search(int key) {
        if (head == null) return false;
        Node temp = head;
        do {
            if (temp.data == key)
                return true;
            temp = temp.next;
        } while (temp != head);
        return false;
    }

    // Traverse (Display List)
    void traverse() {
        if (head == null) {
            System.out.println("List is empty.");
            return;
        }

        Node temp = head;
        System.out.print("Circular List: ");
        do {
            System.out.print(temp.data + " ");
```

```
            temp = temp.next;
        } while (temp != head);
        System.out.println();
    }
}
```

Main Class (Test Program)

```
public class Main {
    public static void main(String[] args) {
        CircularLinkedList cll = new CircularLinkedList();

        cll.insertAtEnd(10);
        cll.insertAtEnd(20);
        cll.insertAtEnd(30);
        cll.insertAtBeginning(5);

        cll.traverse(); // Output: Circular List: 5 10 20 30

        System.out.println("Search 20: " + cll.search(20)); // true
        System.out.println("Search 40: " + cll.search(40)); // false

        cll.deleteNode(10);
        cll.traverse(); // Output: Circular List: 5 20 30

        cll.deleteNode(5);
        cll.traverse(); // Output: Circular List: 20 30
    }
}
```

Time Complexity

Operation	Time Complexity
Insert at Beginning	O(1)
Insert at End	O(1)
Delete	O(n)
Search	O(n)
Traverse	O(n)

Advantages

- No `null` references (continuous circular flow)
- Useful for **round-robin scheduling** or **queue implementation**
- Easy to move from last node to first

Disadvantages

- Can lead to **infinite loops** if not handled carefully
- Slightly complex insertion/deletion logic

Problems: Reverse Linked List, Detect Loop

Reverse a Singly Linked List

Problem:

Given a linked list, reverse it so that the last node becomes the head.

Example:
`1 → 2 → 3 → 4 → null`
Reversed ⟶ `4 → 3 → 2 → 1 → null`

Approach (Iterative Method)

- Use **three pointers**: `prev`, `current`, `next`.
- Traverse the list and **reverse the `next` pointers**.

Logic:

1. Initialize `prev = null` and `current = head`.
2. For each node:
    - Store `next = current.next`.
    - Reverse pointer: `current.next = prev`.
    - Move `prev` and `current` forward.
3. Finally, set `head = prev`.

Java Code

```
class LinkedList {
    Node head;

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

    void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) { head = newNode; return; }
        Node temp = head;
        while (temp.next != null) temp = temp.next;
        temp.next = newNode;
    }

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

```java
    }

    void reverse() {
        Node prev = null, current = head, next = null;
        while (current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
        }
        head = prev;
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.insertAtEnd(1);
        list.insertAtEnd(2);
        list.insertAtEnd(3);
        System.out.print("Original: ");
        list.traverse();

        list.reverse();
        System.out.print("Reversed: ");
        list.traverse();
    }
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Detect Loop in a Linked List

Problem:

Check if a linked list contains a **cycle/loop**.

Example:
`1 → 2 → 3 → 4 → 2` (loop back to node with value 2)

Approach 1: Floyd's Cycle Detection (Tortoise and Hare)

- Use **two pointers**: `slow` moves 1 step, `fast` moves 2 steps.
- If a loop exists, `slow` and `fast` will **meet at some point**.

Java Code

```java
boolean detectLoop() {
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true; // loop detected
    }
    return false;
}

// Example usage
```

```java
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.insertAtEnd(1);
    list.insertAtEnd(2);
    list.insertAtEnd(3);
    list.insertAtEnd(4);

    // Creating a loop manually
    list.head.next.next.next.next = list.head.next; // 4 -> 2

    System.out.println("Loop Detected? " + list.detectLoop()); // true
}
```

Time Complexity: O(n)
Space Complexity: O(1)

Summary

Problem	Approach	Time	Space
Reverse Linked List	Iterative 3-pointers	O(n)	O(1)
Detect Loop	Floyd's Cycle Detection	O(n)	O(1)