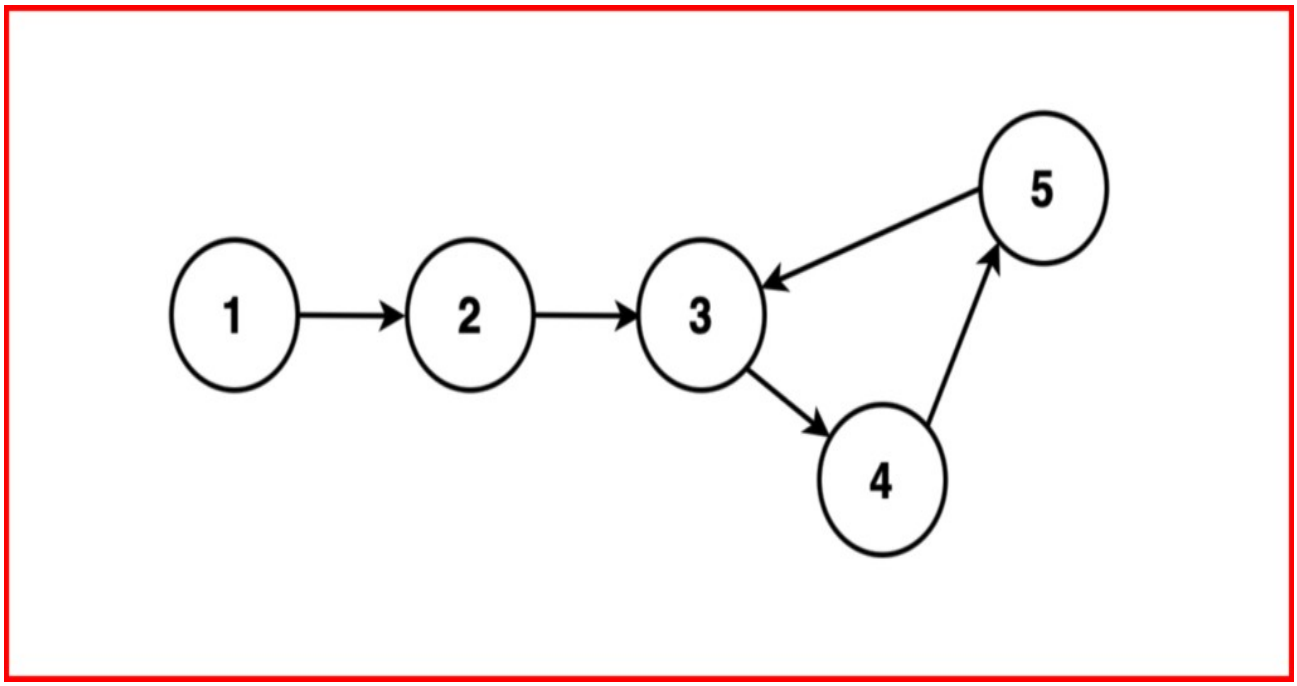
Detect a Cycle in a Linked List

Examples

Example 1:

Input Format:

LL: 1 2 3 4 5



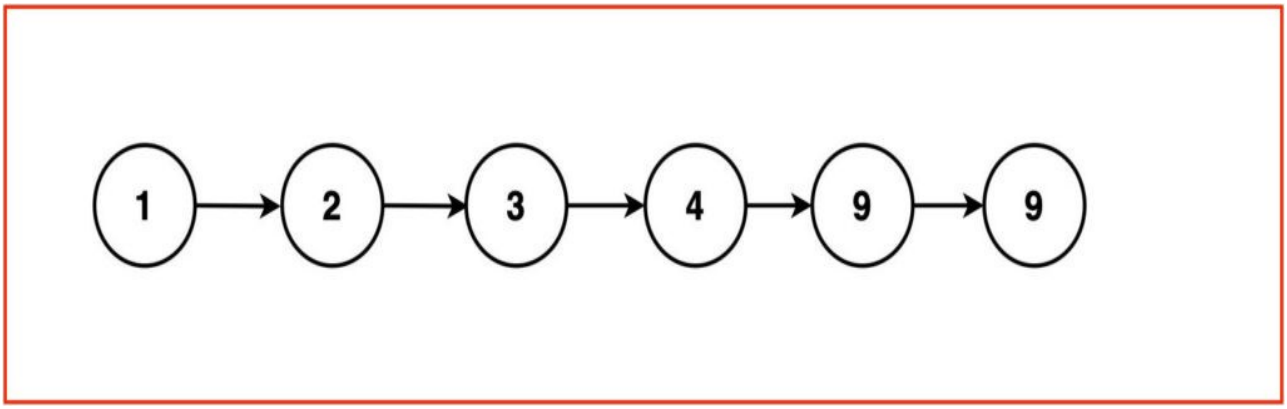
Result: True

Explanation: The last node with the value of 5 has its 'next' pointer pointing back to a previous node with the value of 3. This has resulted in a loop, hence we return true.

Example 2:

Input Format:

LL: 1 2 3 4 9 9

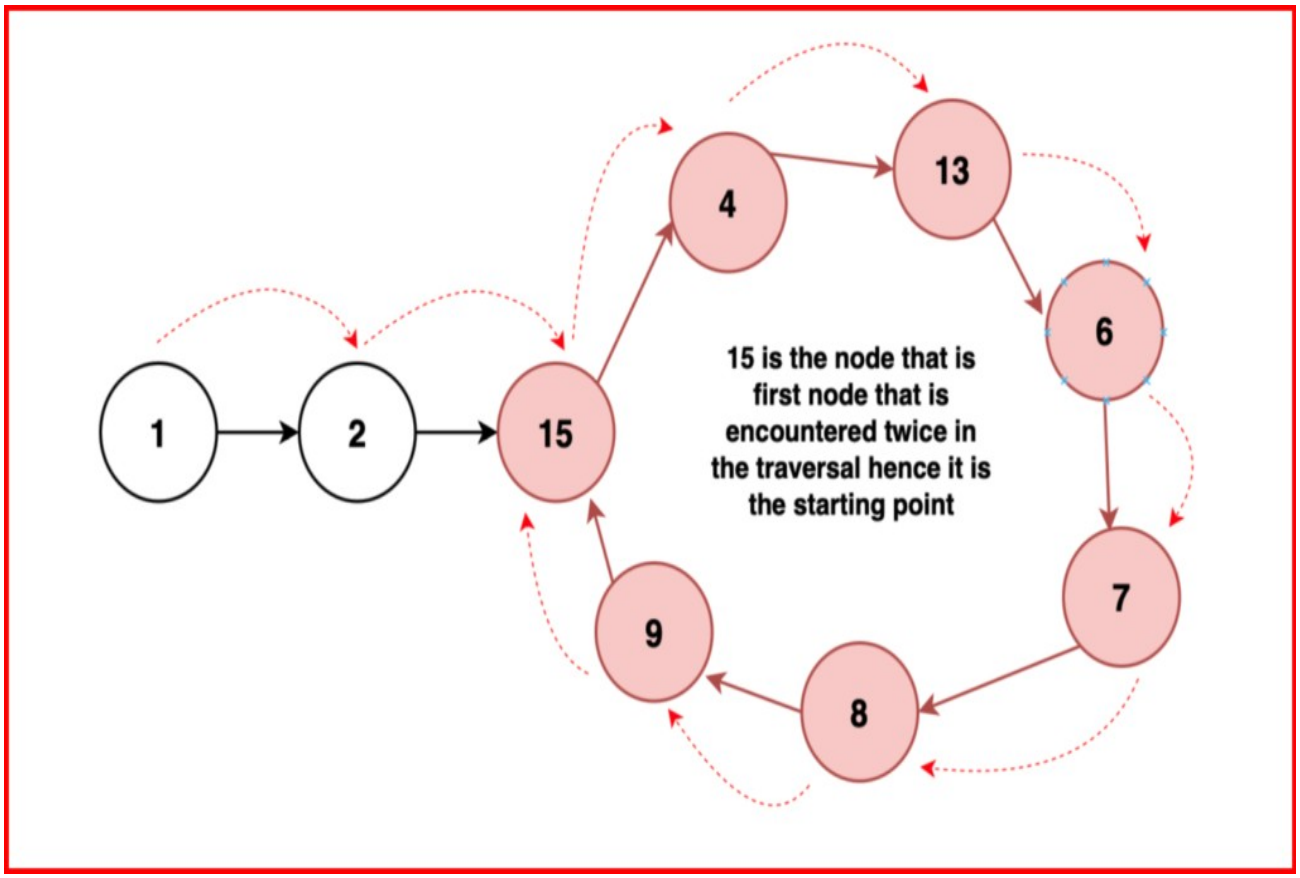


Result: False

Explanation: : In this example, the linked list does not have a loop hence returns false.

Brute Force Approach
Algorithm / Intuition

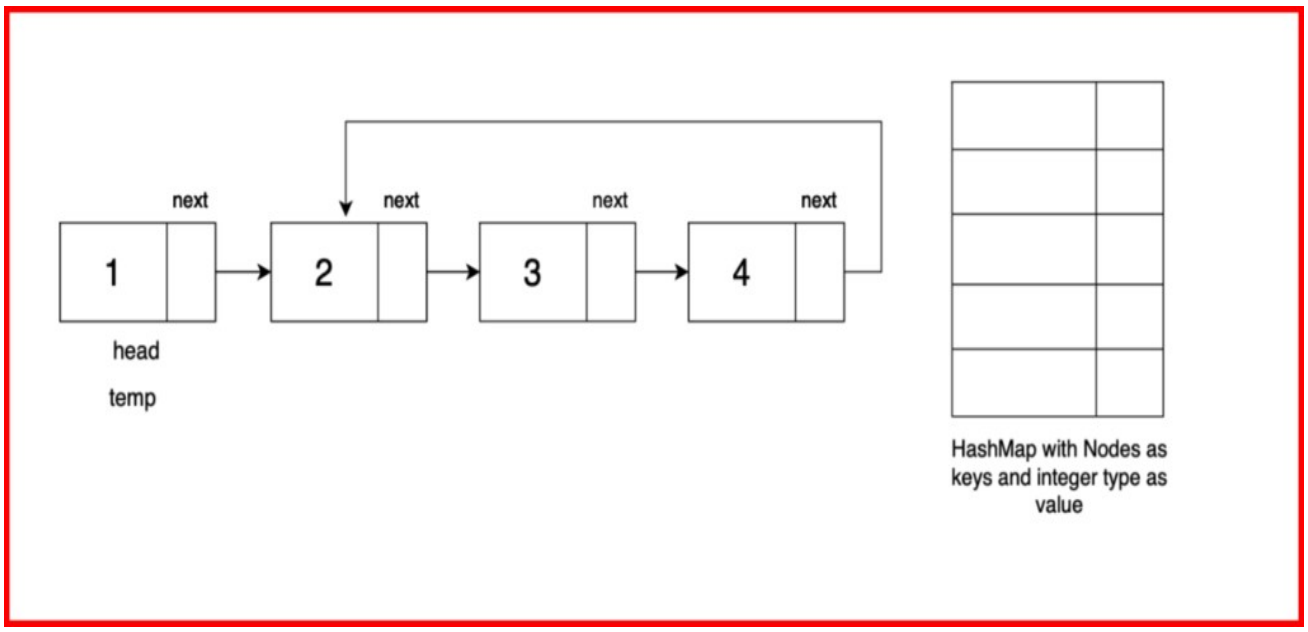
A **loop** in a linked list occurs when there's a node that, **when followed**, **brings you back** to it, indicating a **closed loop** in the list.



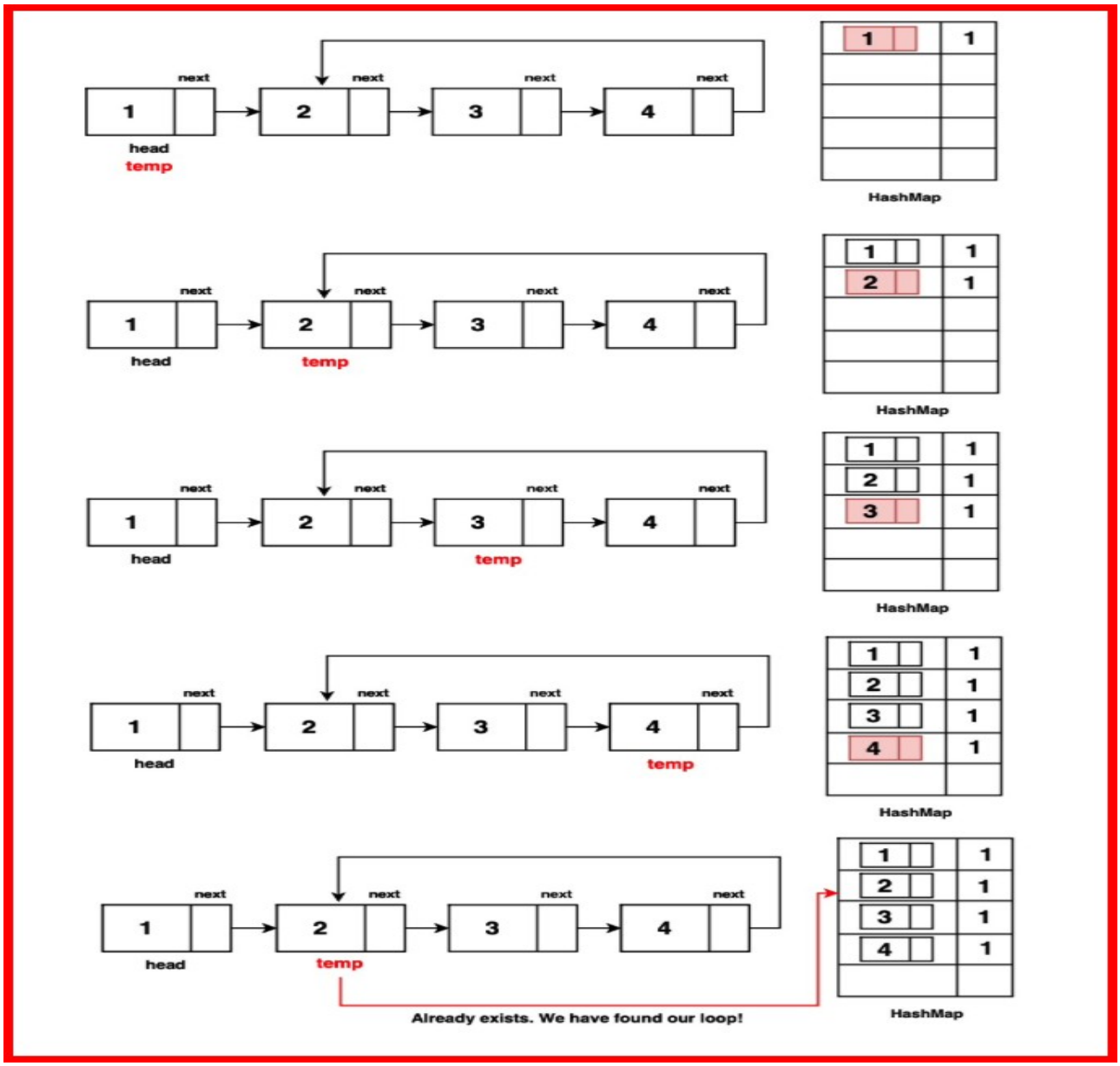
Hence it's important to keep track of nodes that have already been visited so that loops can be detected. One common way to do this is by using hashing.

Algorithm:

Step 1: Traverse through the LL using the traversal technique of assigning a **temp** node to the head and **iterating** by moving to the next element till we reach **null**.



Step 2: While traversing, keep a track of the visited nodes in the map data structure.



Note: Storing the entire node in the map is essential to distinguish between nodes with **identical values** but **different positions** in the list. This ensures **accurate loop detection** and not just duplicate value checks.

Step 3: If a **previously visited node** is encountered again, that proves that there is a **loop** in the linked list hence return **true**.

Step 4: If the traversal is completed, and we reach the last point of the LL which is **null**, it means there was **noloop**, hence we return **false**.

Code

```
import java.util.HashMap;
import java.util.Map;

// Node class represents a
// node in a linked list
class Node {
    // Data stored in the node
    public int data;
    // Pointer to the next node in the list
    public Node next;

    // Constructor with both data
    // and next node as parameters
    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }

    // Constructor with only data as
    // a parameter, sets next to null
    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

public class Main {
    // Function to detect a
    // loop in a linked list
    public static boolean detectLoop(Node head) {
        // Initialize a pointer 'temp'
        // at the head of the linked list
        Node temp = head;

        // Create a map to keep track
        // of encountered nodes
        Map<Node, int> nodeMap = new HashMap<>();

        // Step 2: Traverse the linked list
        while (temp != null) {
            // If the node is already in
            // the map, there is a loop
            if (nodeMap.containsKey(temp)) {
                return true;
            }
            // Store the current node in the map
            nodeMap.put(temp, 1);
            // Move to the next node
            temp = temp.next;
        }
```

```
        // Step 3: If the list is successfully
        // traversed without a loop, return false
        return false;
    }

    public static void main(String[] args) {
        // Create a sample linked list
        // with a loop for testing
        Node head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);
        Node fourth = new Node(4);
        Node fifth = new Node(5);

        head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;
        // Create a loop
        fifth.next = third;

        // Check if there is a loop
        // in the linked list
        if (detectLoop(head)) {
            System.out.println("Loop detected in the linked list.");
        } else {
            System.out.println("No loop detected in the linked list.");
        }

        // No need to explicitly free memory
        // in Java; the garbage collector handles it
    }
}
```

Output: Loop detected in the linked list.

Complexity Analysis

Time Complexity: O(N * 2 * log(N))The algorithm traverses the linked list **once**, performing hashmap **insertions** and **searches** in the while loop for each node. The insertion and search operations in the unordered_map have a worst-case time complexity of **O(log(N))**. As the loop iterates through N nodes, the total time complexity is determined by the product of the traversal **(O(N))** and the average-case complexity of the **hashmap operations (insert and search)**, resulting in **O(N * 2 * log(N)).**

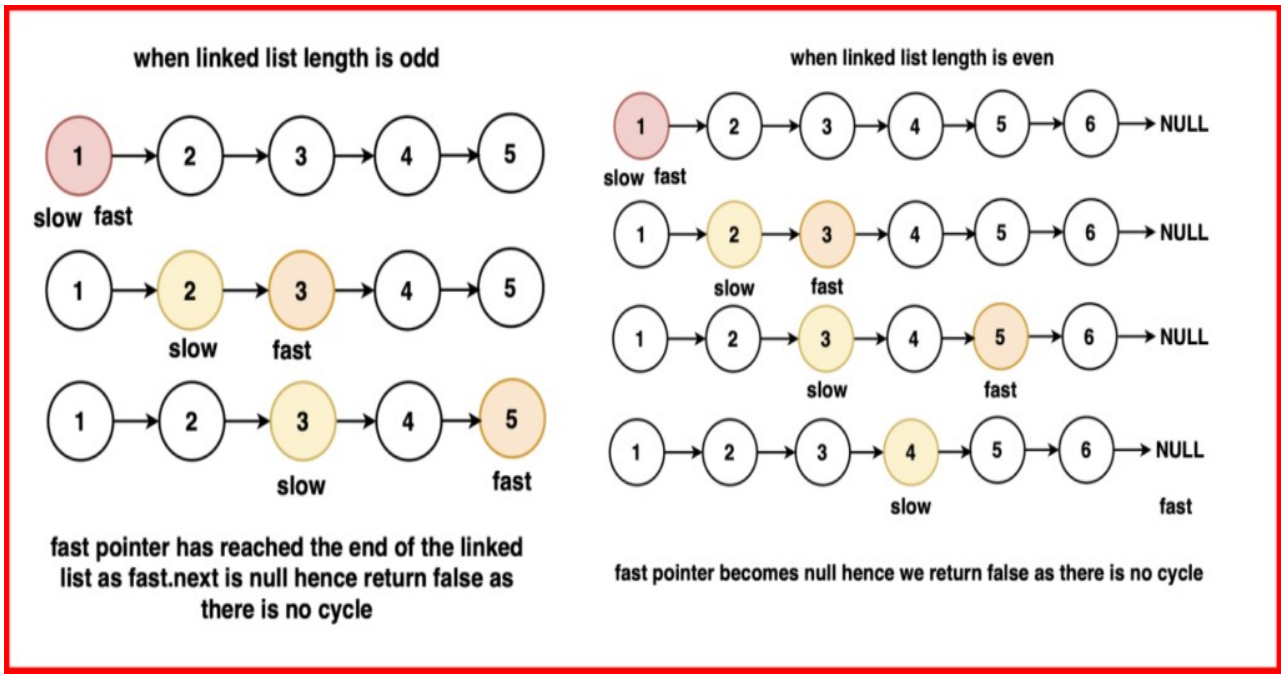
Space Complexity: O(N) The code uses a **hashmap**/**dictionary** to store encountered nodes, which can take up to O(N) additional space, where 'n' is the number of nodes in the list. Hence, the **spacecomplexity** is O(N) due to the use of the map to track nodes.

Optimal Approach
Algorithm / Intuition

The previous method uses O(N) **additional memory**, which can become quite large as the **linked list length grows**. To enhance efficiency, the **Tortoise** and **Hare Algorithm** is introduced as an **optimization**.

When the **tortoise** and **hare** enter the loop, they may be at different positions within the loop due to the difference in their **speeds**. The hare is moving **faster**, so it will traverse a **greater distance** in the same amount of time.

If there is **no loop** in the linked list, the hare will eventually reach the **end**, and the algorithm will terminate without a meeting occurring.



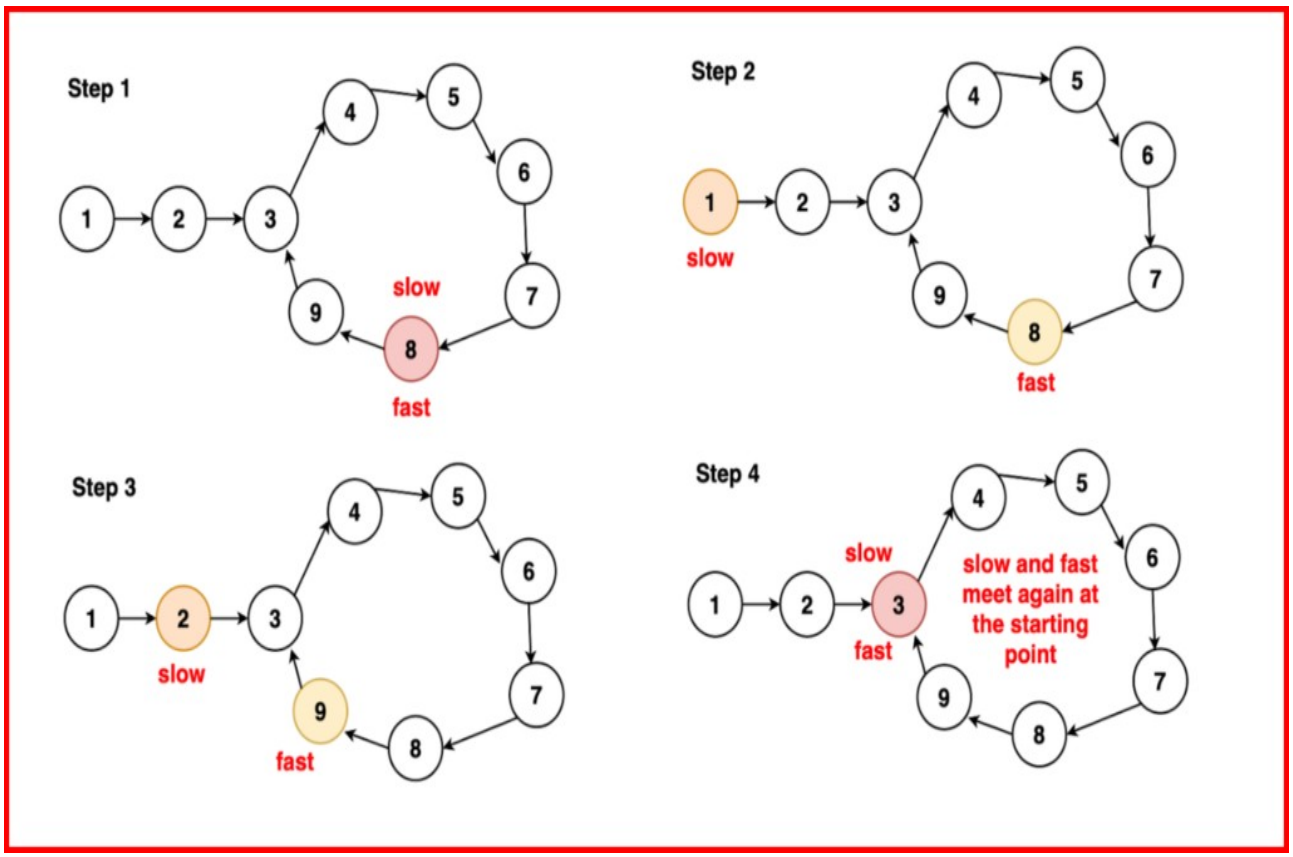
Algorithm

Step 1: Initialise two pointers, **`slow`** and **`fast`**, to the head of the linked list. **`slow`** will advance **one step** at a time, while **`fast`** will advance **two steps** at a time. These pointers will move **simultaneously**.

Step 2: Traverse the linked list with the **`slow`** and **`fast`** pointers. While traversing, repeatedly move **`slow`** **one step** and **`fast`** **two steps** at a time.

Step 3: Continue this traversal until one of the following conditions is met:

1. **`fast`** or **`fast.next`** reaches the end of the linked list (i.e., becomes **null**). In this case, there is **no loop** in the linked list ie. the linked list is **linear**, and the algorithm terminates by returning **false**.
2. **`fast`** and **`slow`** pointers meet at the same node. This indicates the presence of a **loop** in the linked list, and the algorithm terminates by returning **`true`**.



Intuition:

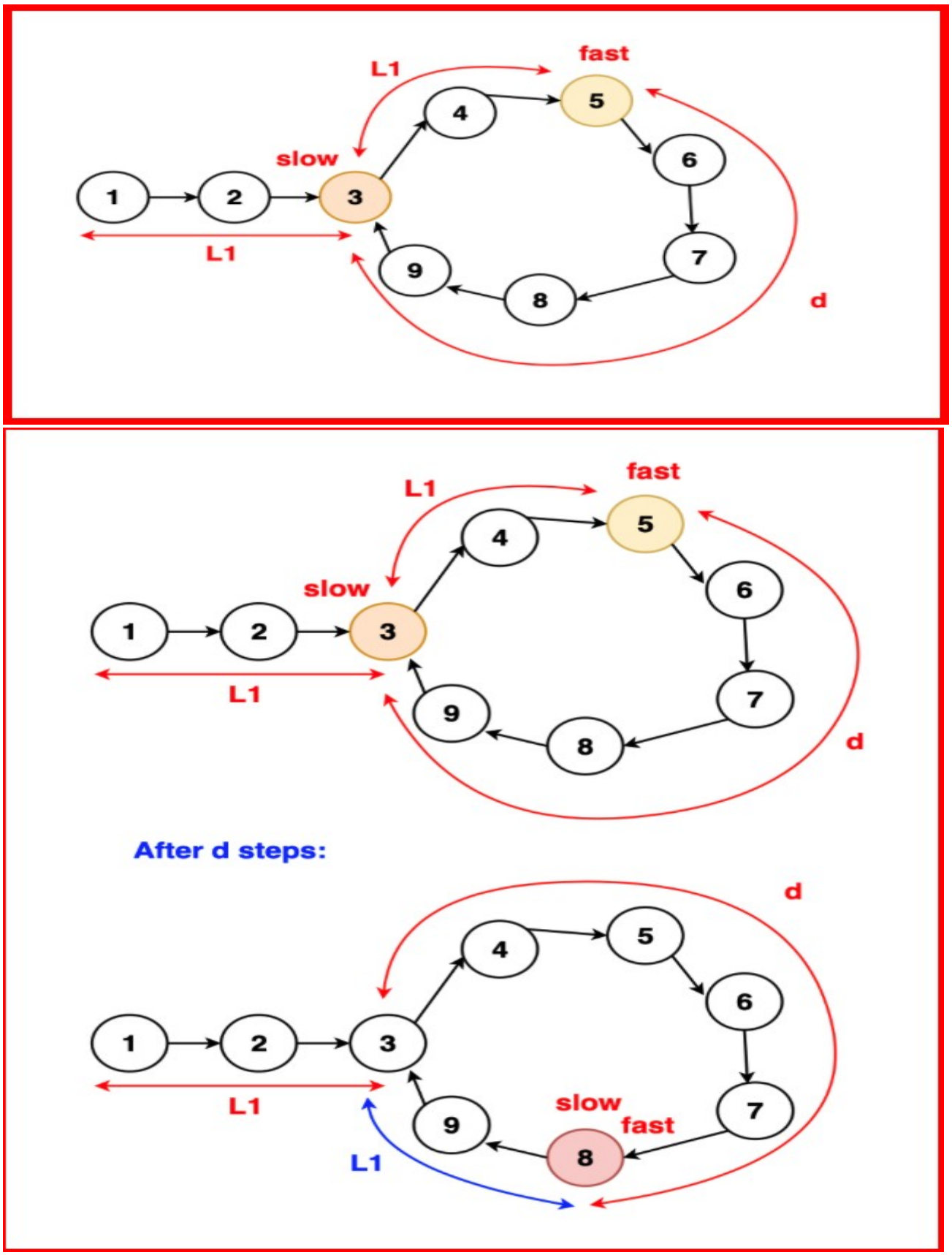
In a linked list with a loop, consider two pointers: one that moves one node at a time (**slow**) and another that moves two nodes at a time (**fast**). If we start moving these pointers with their defined speed they will surely enter the loop and might be at some distance **'d'** from each other within the loop.

The key insight here is the **relative speed** between these pointers. The fast pointer, moving at double the speed of the slow one, **closes the gap** between them by **one node in every iteration**. This means that with each step, the distance decreases by one node.

Imagine a race where one runner moves at **twice** the speed of another. The faster runner covers the ground faster and closes the gap, resulting in a reduction in the distance between them. Similarly, the **fast** pointer catches up to the **slow** pointer in the looped linked list, closing in the gap between them until it reaches zero.

Proof:

Let **'d'** denote the initial distance between the **slow** and **fast** pointers inside the loop. At each step, the fast pointer moves ahead by two nodes while the slow pointer advances by one node.



The **relative speed** between them causes the gap to decrease by one node in each iteration (fast gains two nodes while slow gains one node). This continuous reduction ensures that the difference between their positions **decreases steadily**. Mathematically, if the fast pointer **gains ground** twice as fast as the slow pointer, the difference in their positions reduces by one node after each step.

Consequently, this reduction in the distance between them continues **until** the **difference becomes zero**.

Hence, the proof lies in this **iterative process** where the faster rate of the fast pointer leads to a continual decrease in the gap distance, ultimately resulting in their collision within the looped linked list.

Code

```
import java.util.HashMap;
import java.util.Map;

// Node class represents a
// node in a linked list
class Node {
    // Data stored in the node
    public int data;
    // Pointer to the next node in the list
    public Node next;

    // Constructor with both data
    // and next node as parameters
    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }

    // Constructor with only data as
    // a parameter, sets next to null
    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

public class Main {

    // Function to detect a loop in a linked list
    // using the Tortoise and Hare Algorithm
    public static boolean detectCycle(Node head) {
        // Initialize two pointers, slow and fast,
        // to the head of the linked list
        Node slow = head;
        Node fast = head;

        // Step 2: Traverse the linked list
        // with the slow and fast pointers
        while (fast != null && fast.next != null) {
            // Move slow one step
            slow = slow.next;
            // Move fast two steps
            fast = fast.next.next;

            // Check if slow and fast pointers meet
            if (slow == fast) {
                return true;  // Loop detected
            }
        }

        // If fast reaches the end of the
        // list, there is no loop
        return false;
```

```
    }


    public static void main(String[] args) {
        // Create a sample linked list
        // with a loop for testing
        Node head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);
        Node fourth = new Node(4);
        Node fifth = new Node(5);

        head.next = second;
        second.next = third;
        third.next = fourth;
        fourth.next = fifth;
        // Create a loop
        fifth.next = third;

        // Check if there is a loop
        // in the linked list
        if (detectCycle(head)) {
            System.out.println("Loop detected in the linked list.");
        } else {
            System.out.println("No loop detected in the linked list.");
        }

        // No need to explicitly free memory
        // in Java; the garbage collector handles it
    }
}
```

Output: Loop detected in the linked list.

Complexity Analysis

Time Complexity: O(N), where N is the number of nodes in the linked list. This is because in the **worst-case scenario**, the **fast** pointer, which **moves quicker**, will either reach the end of the list (in case of no loop) or meet the **slow** pointer (in case of a loop) in a **linear time** relative to the length of the list.

The key insight into why this is **O(N)** and **not something slower** is that each step of the algorithm reduces the distance between the fast and slow pointers (when they are in the loop) by one. Therefore, the **maximum number** of **steps** needed for them to meet is **proportional** to the number of nodes in the list.

Space Complexity : O(1) The code uses only a **constantamount** of **additionalspace**, regardless of the linked list's length. This is achieved by using two pointers (**slow** and **fast**) to detect the loop without any significant extra memory usage, resulting in **constantspace** complexity, O(1).