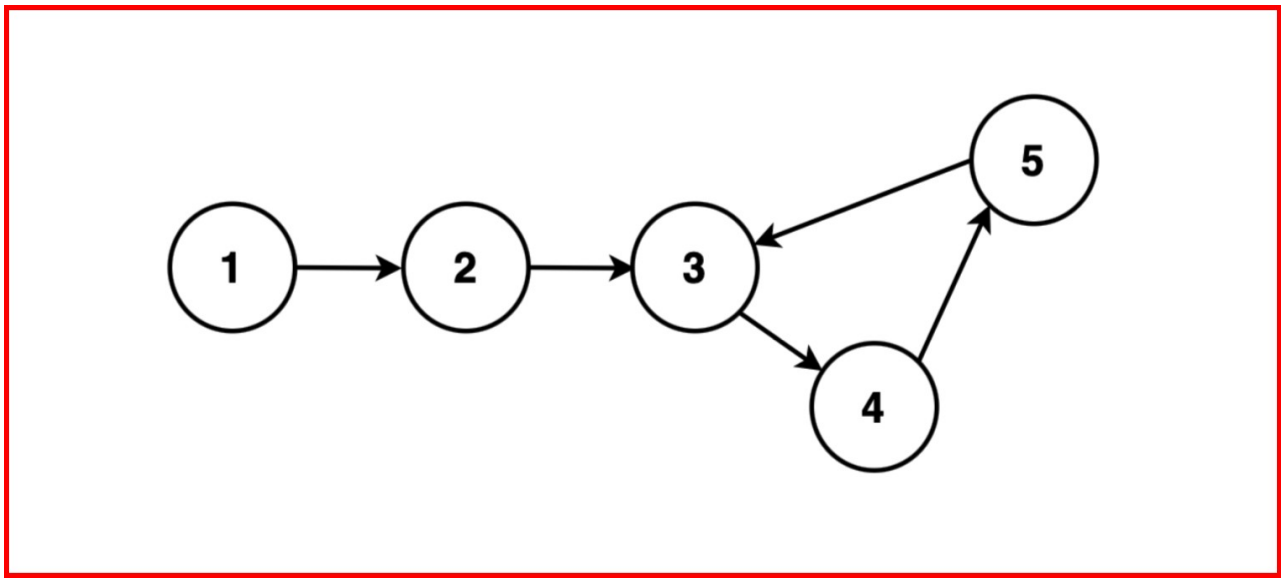
Starting point of loop in a Linked List

Problem Statement: Given the head of a linked list that may contain a cycle, return the starting point of that cycle. If there is no cycle in the linked list return null.

Examples

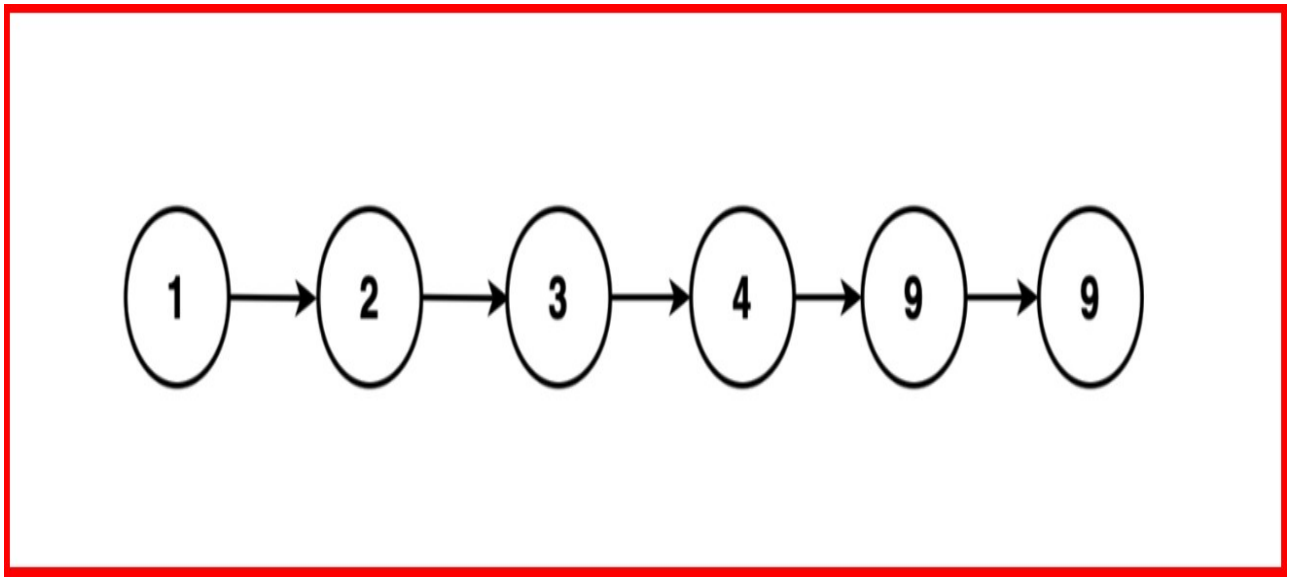
Example 1:
Input: LL: 1 2 3 4 5



Output: 3

Explanation: This linked list contains a loop of size 3 starting at node with value 3.

Example 2:
Input: LL: LL: 1 -> 2 -> 3 -> 4 -> 9 -> 9

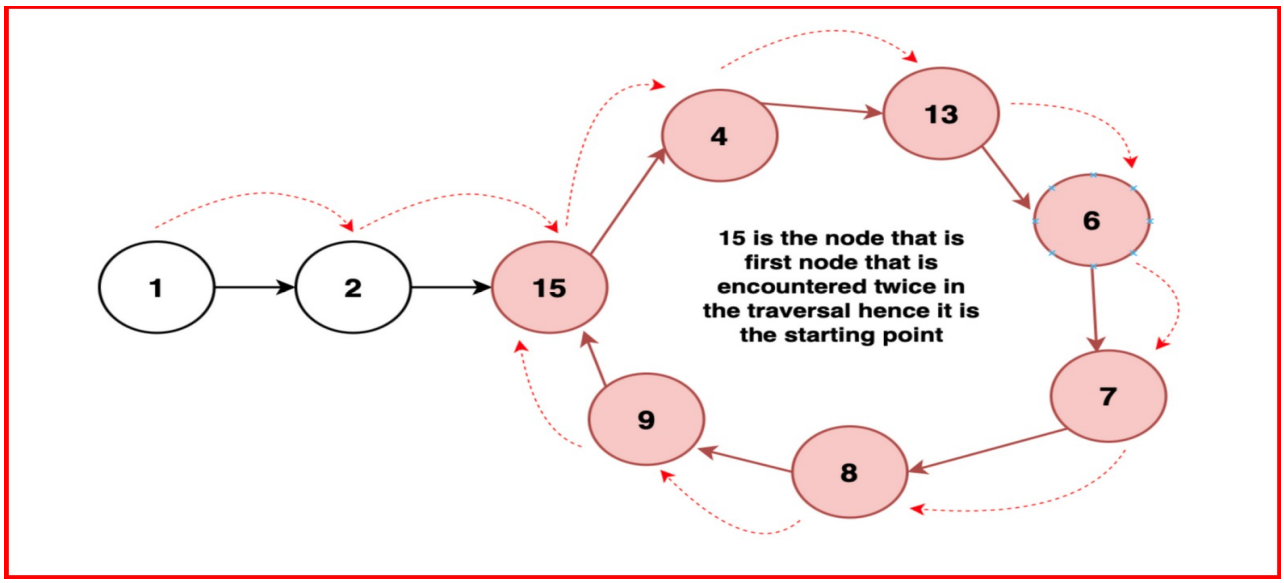


Output: NULL

Explanation: This linked list does not contain a loop hence has no starting point.

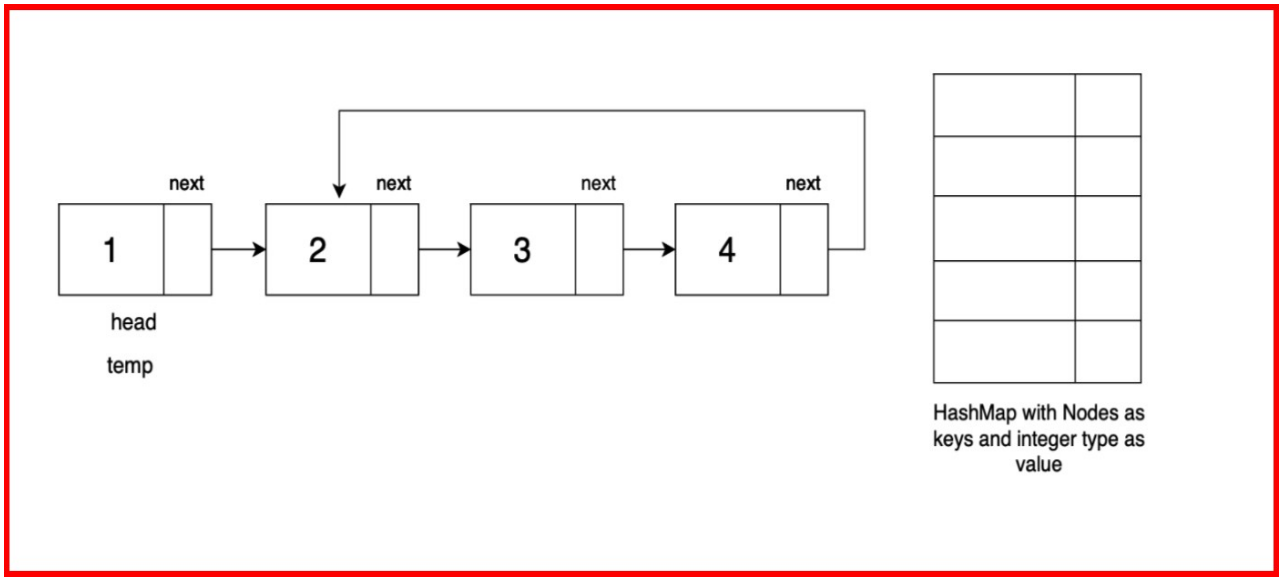
Brute Force Approach
Algorithm / Intuition

The starting point of a loop of the linked list is the first node we visit twice during its traversal. It's the point where we realise that we are no longer moving forward in the list but rather entering a cycle.

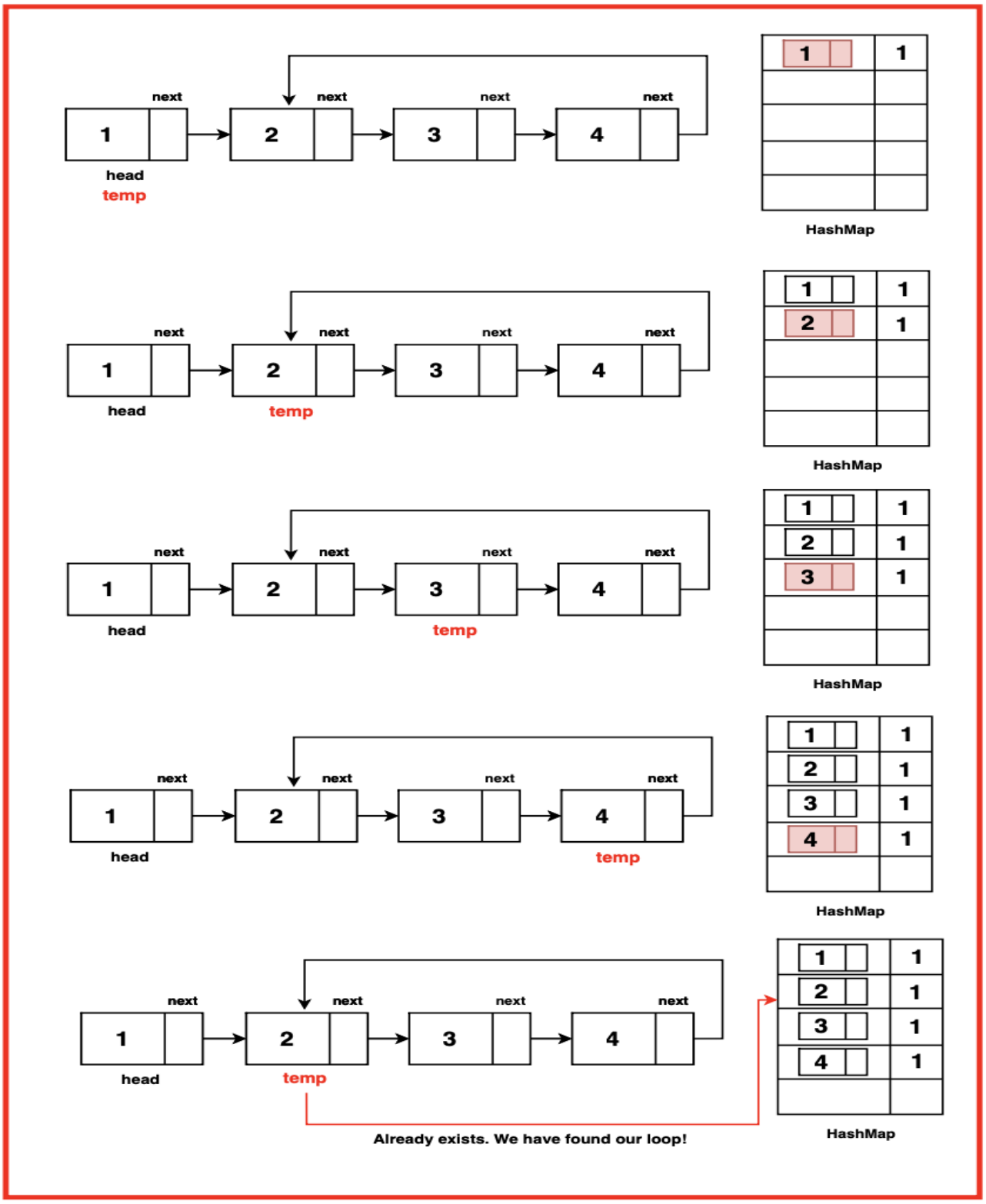


Algorithm:

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 that node.

Step 4: If the traversal is completed, and we reach the last point of the LL which is null, it means there was no loop, hence we return null.

Code

```
import java.util.HashMap;

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

    // Constructor with both data and
    // next node as parameters
    Node(int data1, Node next1) {
        data = data1;
        next = next1;
    }

    // Constructor with only data as a
    // parameter, sets next to null
    Node(int data1) {
        data = data1;
        next = null;
    }
}

public class LinkedListLoopDetection {
    // Function to detect a loop in a linked list
    // and return the starting node of the loop
    public static Node detectLoop(Node head) {
        // Use temp to traverse the linked list
        Node temp = head;

        // HashMap to store all visited nodes
        HashMap<Node, int> nodeMap = new HashMap<>();

        // Traverse the list using temp
        while (temp != null) {
            // Check if temp has been encountered again
            if (nodeMap.containsKey(temp)) {
                // A loop is detected, hence return temp
                return temp;
            }

            // Store temp as visited
            nodeMap.put(temp, 1);

            // Iterate through the list
            temp = temp.next;
        }

        // If no loop is detected, return null
        return null;
    }

    public static void main(String[] args) {
        // Create a sample linked list with a loop
        Node node1 = new Node(1);
        Node node2 = new Node(2);
        node1.next = node2;
        Node node3 = new Node(3);
        node2.next = node3;
```

```
        Node node4 = new Node(4);
        node3.next = node4;
        Node node5 = new Node(5);
        node4.next = node5;

        // Make a loop from node5 to node2
        node5.next = node2;

        // Set the head of the linked list
        Node head = node1;

        // Detect the loop in the linked list
        Node loopStartNode = detectLoop(head);

        if (loopStartNode != null) {
            System.out.println("Loop detected. Starting node of the loop is: " +
loopStartNode.data);
        } else {
            System.out.println("No loop detected in the linked list.");
        }
    }
}
```

Output: Loop detected. Starting node of the loop is: 2

Complexity Analysis

Time Complexity: O(N) The code traverses the entire linked list once, where 'N' is the number of nodes in the list. Therefore, the time complexity is linear, O(N).

Space Complexity : O(N) The code uses a hash map/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 space complexity is O(N) due to the use of the map to track nodes.

Optimal Approach
Algorithm / Intuition

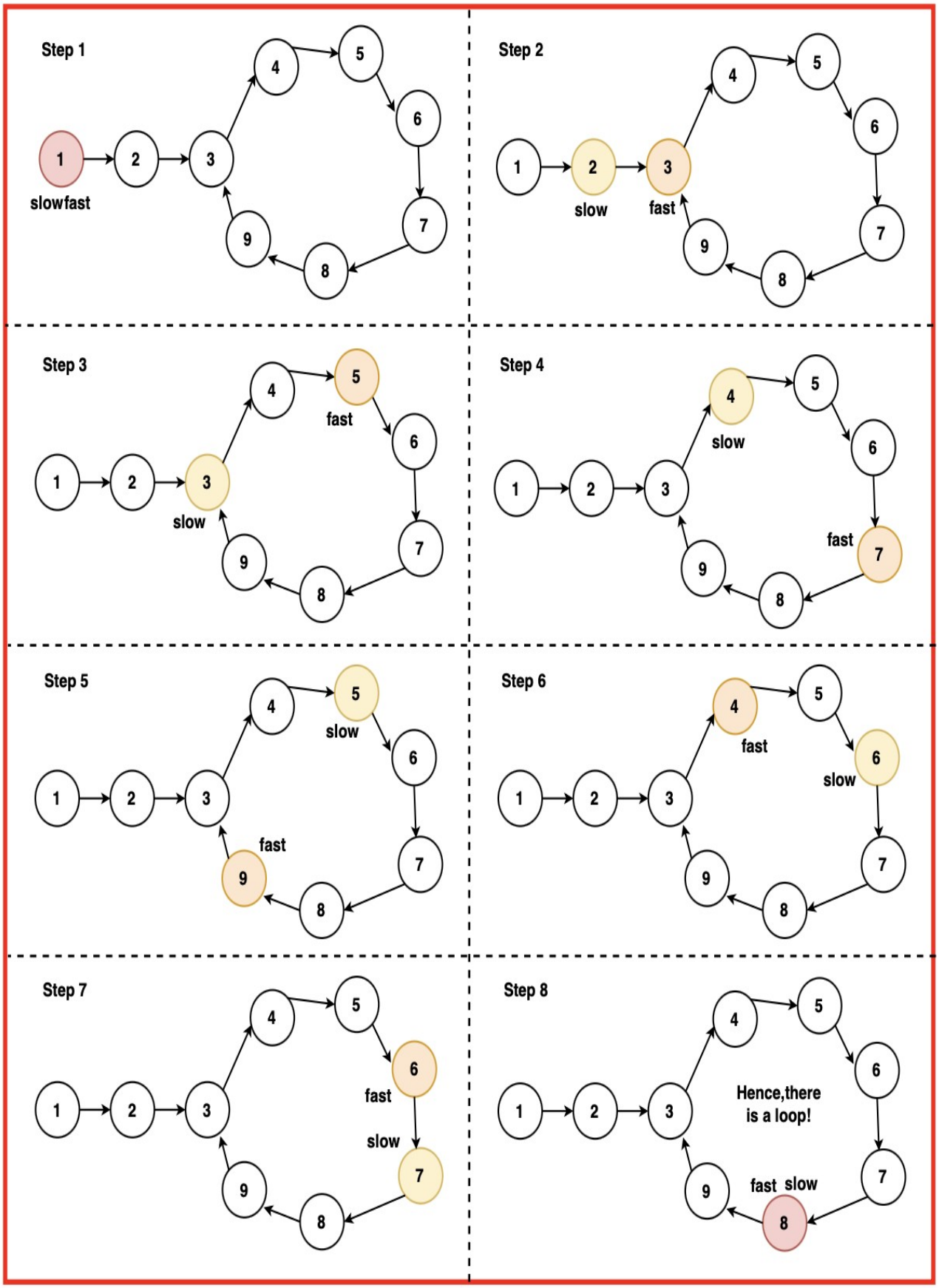
The previous method uses O(N) additional memory, which can become quite large as the linked list length grows. To enhance efficiency, the Tortoise and Hare Algorithm is introduced as an optimization.

The key insight is that when the slow and fast pointers meet inside the loop, the distance travelled by each pointer can be used to calculate the starting point of the loop.

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.

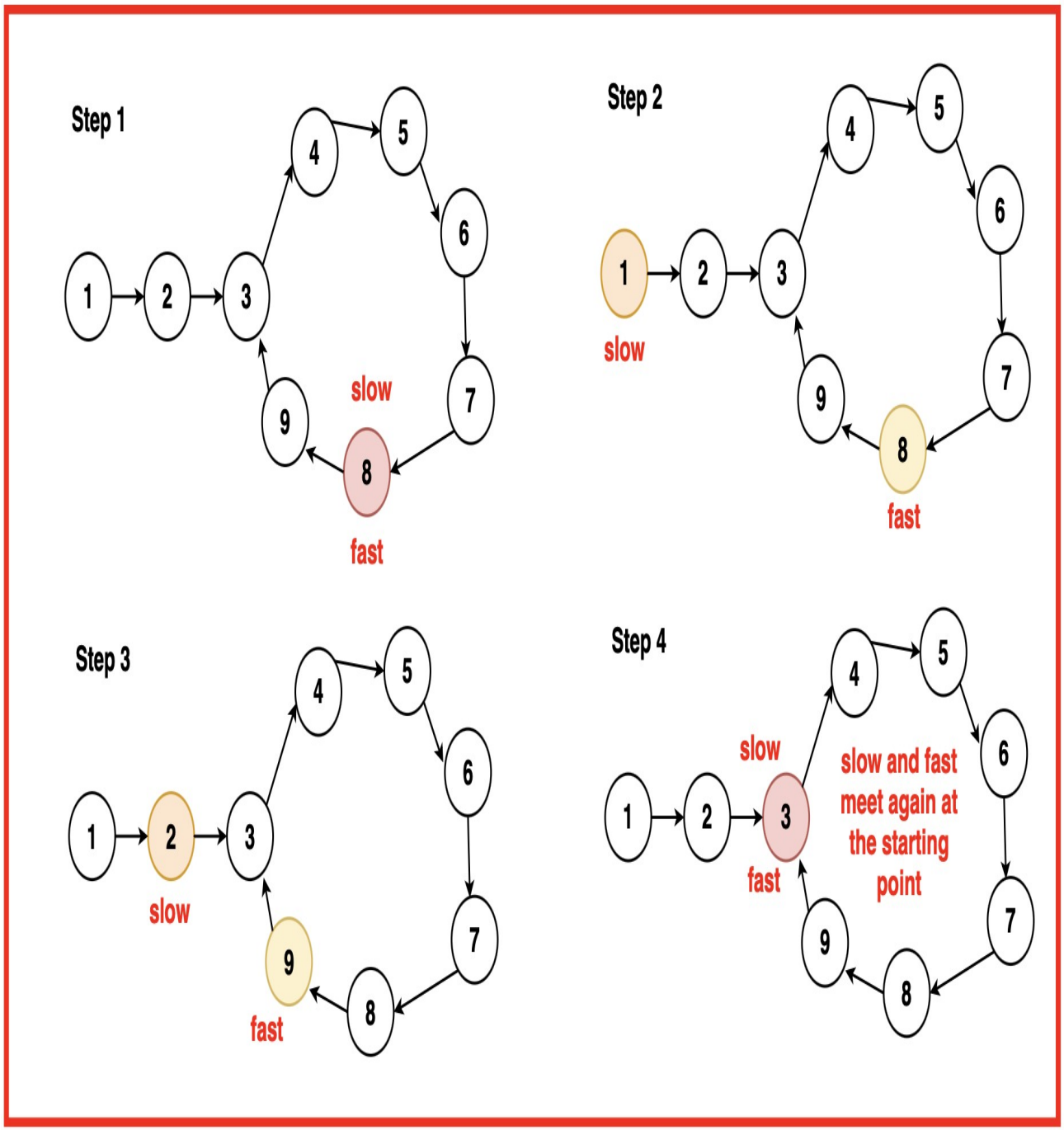


Step 3: Continue this traversal until one of the following conditions is met:

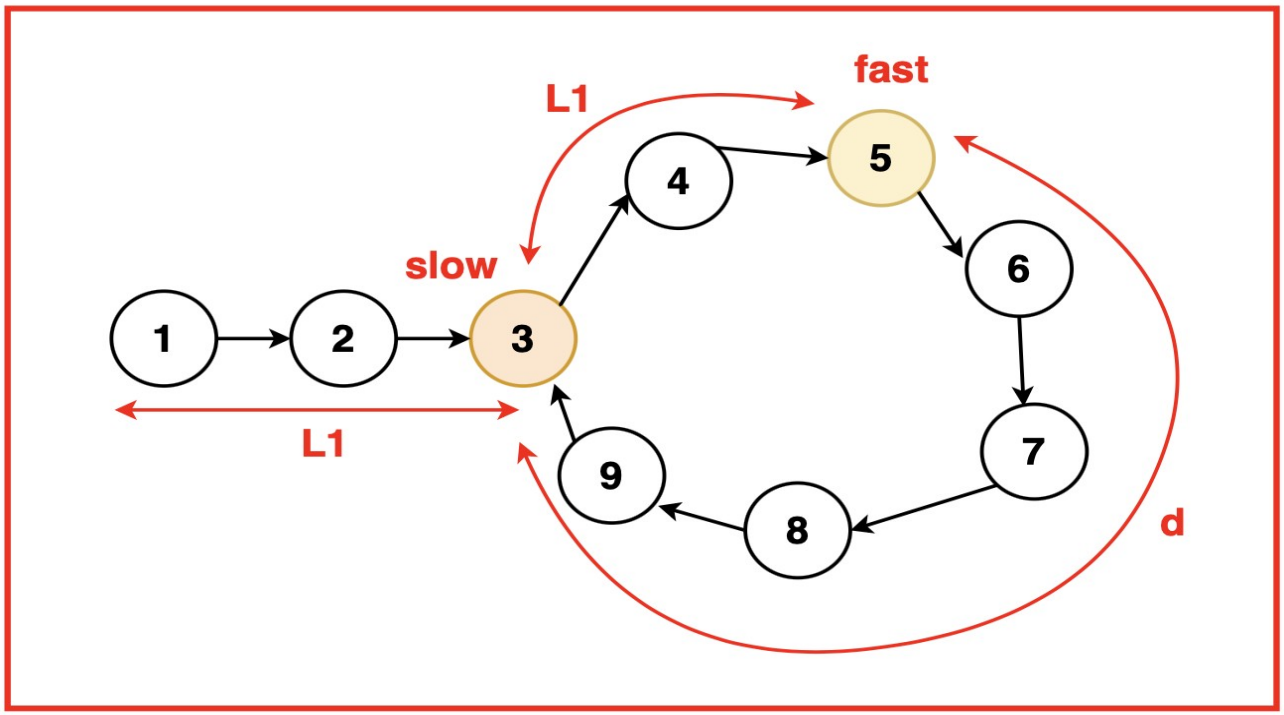
1. `fast` or `fast.next` reaches the end of the linked list (i.e., becomes null). In this case, there is no loop in the linked list, and the algorithm terminates by returning null.

2. `fast` and `slow` pointers meet at the same node. This indicates the presence of a loop in the linked list.

Step 4: Reset the `slow` pointer to the head of the linked list. Move `fast` and `slow` one step at a time until they meet again. The point where they meet again is the starting point.



You may be curious about the proof for this algorithm, and it hinges on the idea that the point where the slow and fast pointers converge can be leveraged to determine the starting point of the loop.



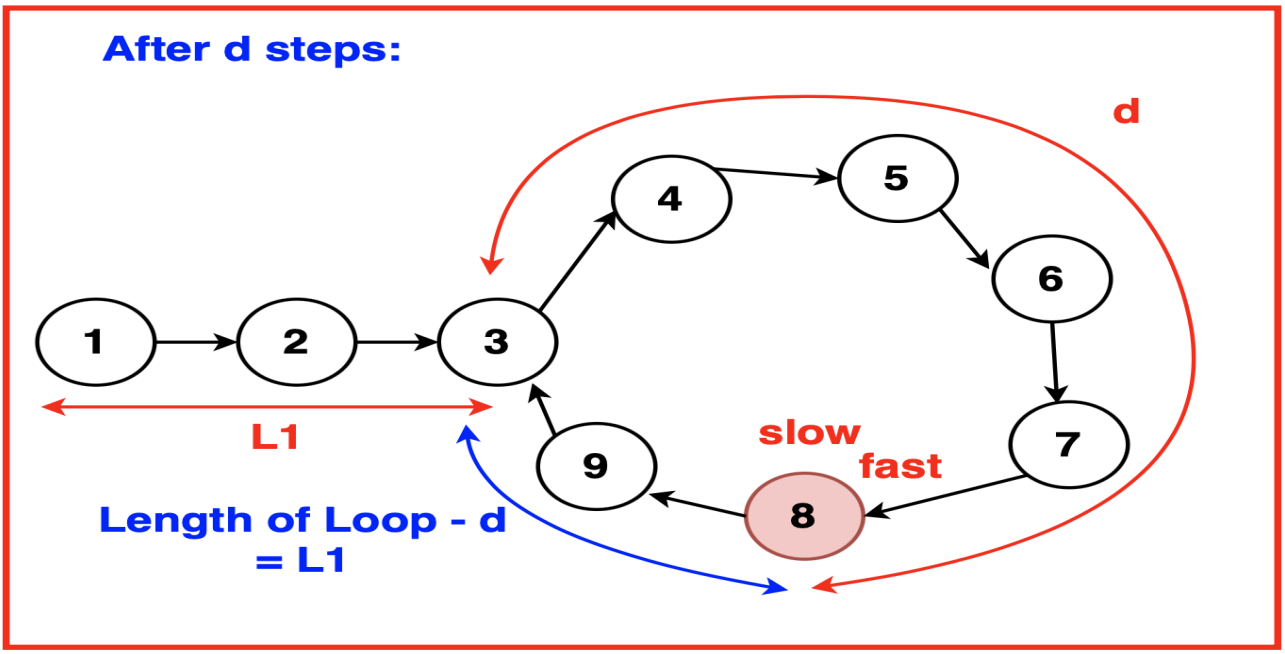
In the "tortoise and hare" algorithm for detecting loops in a linked list, when the slow pointer (tortoise) reaches the starting point of the loop, the fast pointer (hare) is positioned at a point that is twice the distance travelled by the slow pointer. This is because the hare moves at double the speed of the tortoise.

If slow has travelled distance L1 then fast has travelled 2 x L1. Now that slow and fast have entered the loop, the distance fast will have to cover to catch up to slow is the total length of loop minus L1. Let this distance be d.

```
Distance travelled by slow = L1
Distance travelled by fast = 2 * L1
Total length of loop = L1 + d
```

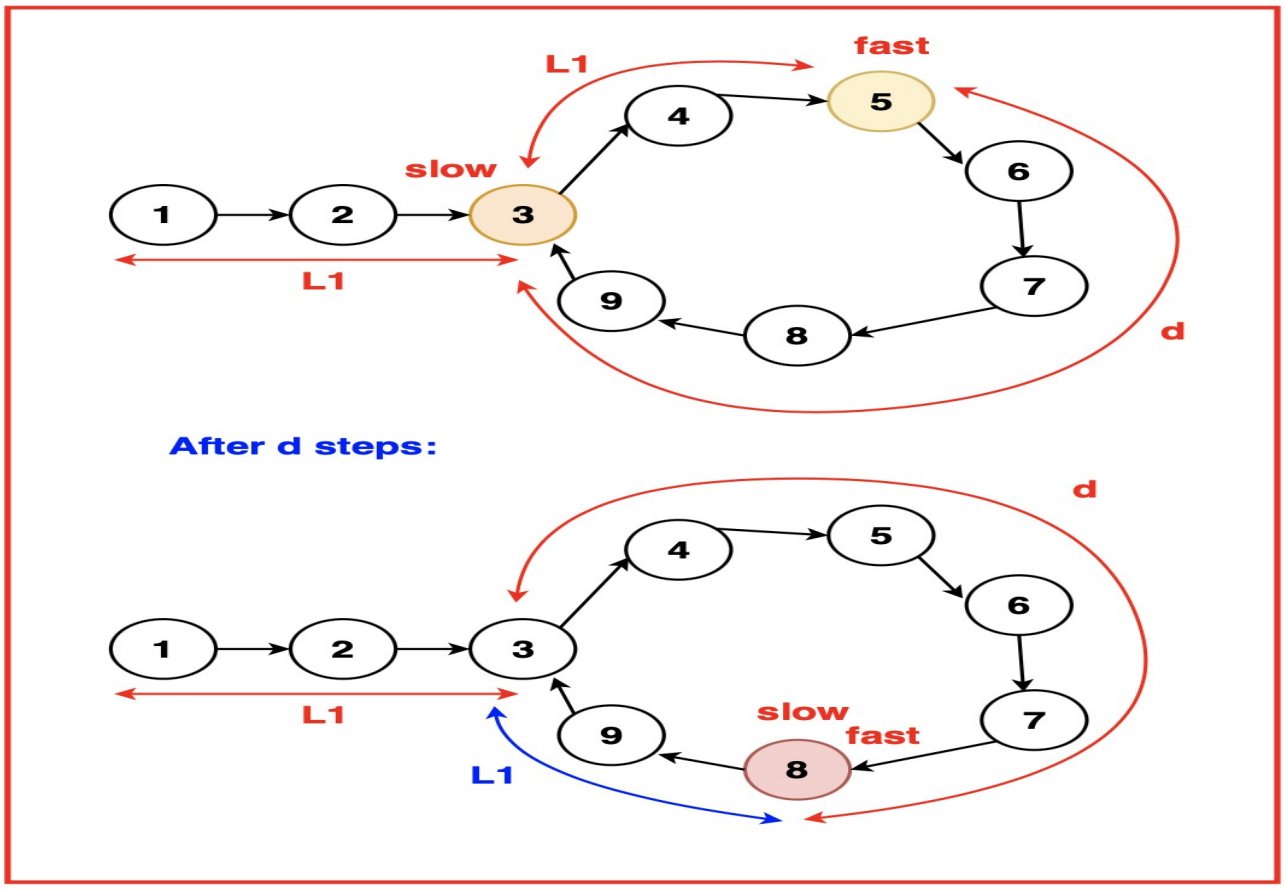
In this configuration, the fast pointer advances toward the slow pointer with two jumps per step, while the slow pointer moves away with one jump per step. As a result, the gap between them decreases by 1 with each step. Given that the initial gap is d, it takes exactly d steps for them to meet.

```
Total length of loop = L1 + d
Distance between slow and fast= d
```



During these d steps, the slow pointer effectively travels d steps from the starting point within the loop and fast travels 2 x d and they meet a specific point. Based on our previous calculations, the total length of the loop is L1 + d. And since the distance covered by the slow pointer within the loop is d, the remaining distance within the loop is equal to L1.

Therefore, it is proven that the distance between the starting point of the loop and the point where the two pointers meet is indeed equal to the distance between the starting point and head of the linked list.



Code

```
import java.util.HashMap;

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

    // Constructor with both data
    // and next node as parameters
    Node(int data1, Node next1) {
        data = data1;
        next = next1;
    }

    // Constructor with only data as
    // a parameter, sets next to null
    Node(int data1) {
        data = data1;
        next = null;
    }
}

public class LinkedListLoopDetection {
    // Function to find the first node
    // of the loop in a linked list

    public static Node firstNode(Node head) {
        // Initialize a slow and fast
        // pointers to the head of the list
        Node slow = head;
        Node fast = head;

        // Phase 1: Detect the loop
        while (fast != null && fast.next != null) {
            // Move slow one step
            slow = slow.next;

            // Move fast two steps
            fast = fast.next.next;

            // If slow and fast meet,
            // a loop is detected
            if (slow == fast) {
                // Reset the slow pointer
                // to the head of the list
                slow = head;

                // Phase 2: Find the first node of the loop
                while (slow != fast) {
                    // Move slow and fast one step
                    // at a time
                    slow = slow.next;
                    fast = fast.next;

                    // When slow and fast meet again,
                    // it's the first node of the loop
                }
```

```
                // Return the first node of the loop
                return slow;
            }
        }

        // If no loop is found, return null
        return null;
    }

    public static void main(String[] args) {
        // Create a sample linked list with a loop
        Node node1 = new Node(1);
        Node node2 = new Node(2);
        node1.next = node2;
        Node node3 = new Node(3);
        node2.next = node3;
        Node node4 = new Node(4);
        node3.next = node4;
        Node node5 = new Node(5);
        node4.next = node5;

        // Make a loop from node5 to node2
        node5.next = node2;

        // Set the head of the linked list
        Node head = node1;

        // Detect the loop in the linked list
        Node loopStartNode = firstNode(head);

        if (loopStartNode != null) {
            System.out.println("Loop detected. Starting node of the loop is: " +
loopStartNode.data);
        } else {
            System.out.println("No loop detected in the linked list.");
        }
    }
}
```

Output: Loop detected. Starting node of the loop is: 2

Complexity Analysis

Time Complexity: O(N) The code traverses the entire linked list once, where 'n' is the number of nodes in the list. This traversal has a linear time complexity, O(n).

Space Complexity : O(1) The code uses only a constant amount of additional space, 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 constant space complexity, O(1).