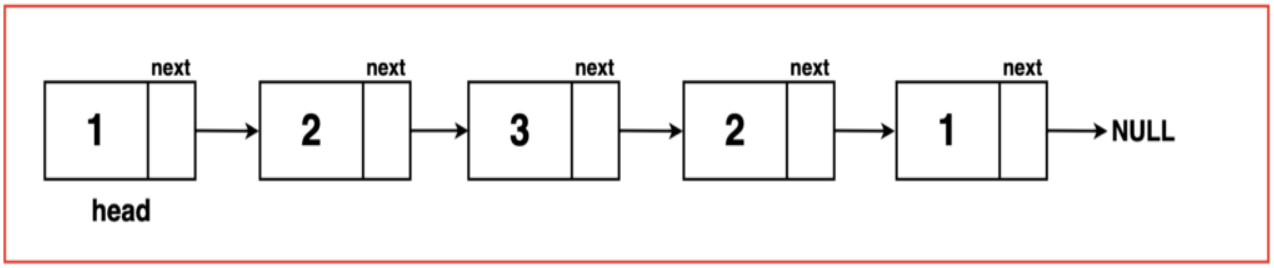
Check if the given Linked List is Palindrome

Examples

Example 1:

Input Format:

LL: 1 2 3 2 1



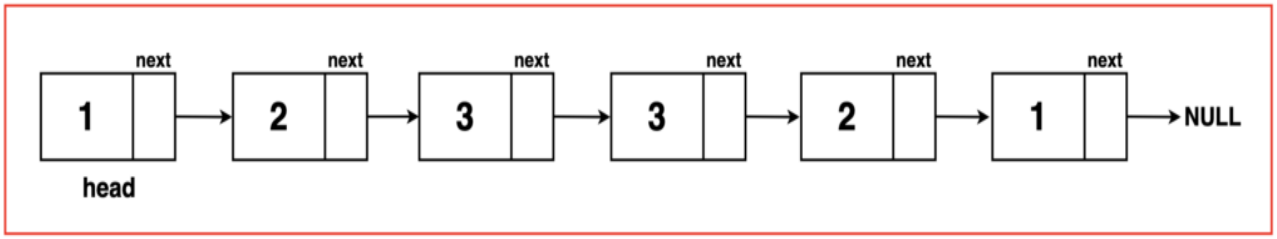
Output: True

Explanation: A linked list with values "1 2 3 2 1" is a palindrome because its elements read the same from left to right and from right to left, making it symmetrical and mirroring itself.

Example 2:

Input Format:

LL: 1 2 3 3 2 1



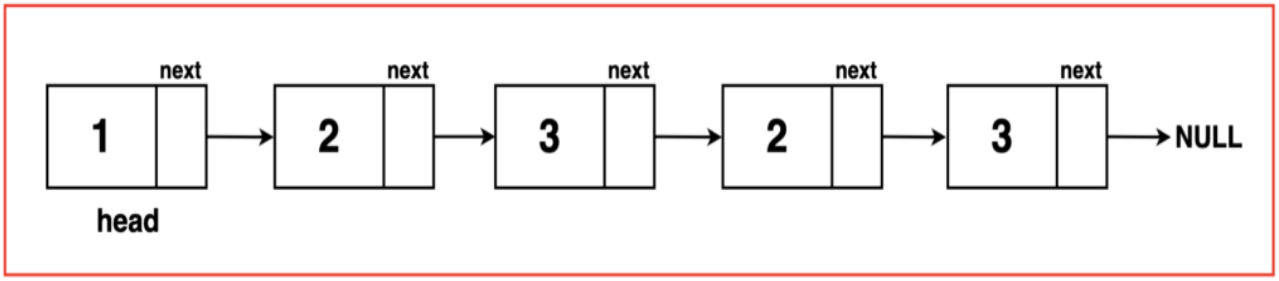
Output: True

Explanation: A linked list with values "1 2 3 3 2 1" is a palindrome because it reads the same forwards and backwards.

Example 3:

Input Format:

LL: 1 2 3 2 3



Output: False

Explanation: The linked list "1 2 3 2 3" is not a palindrome because it reads differently in reverse order, where "3 2 3 2 1" is not the same as the original sequence "1 2 3 2 3."

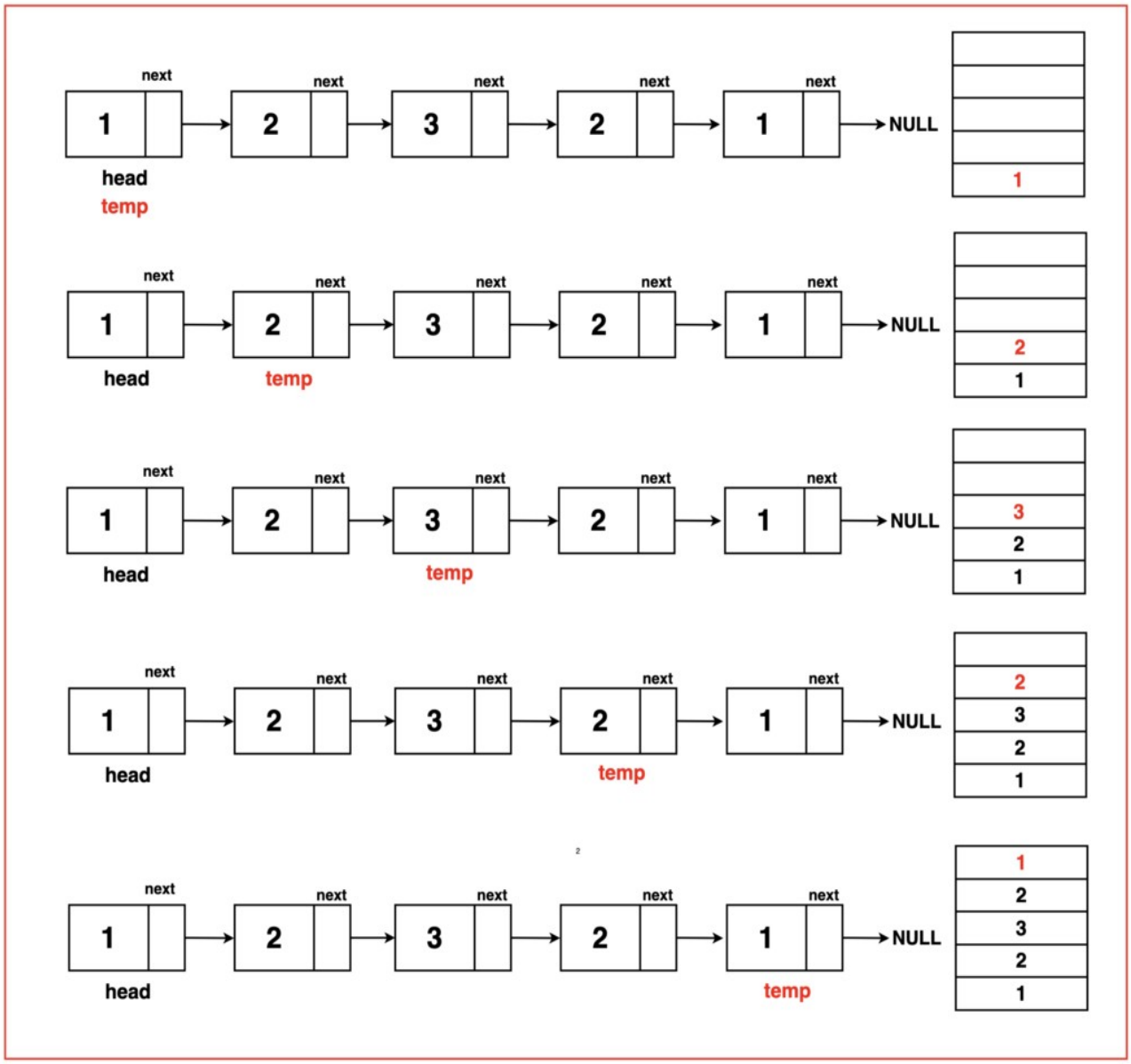
Brute Force Approach
Algorithm / Intuition

A straightforward approach to checking if the given linked list is a palindrome or not is to **temporarily store** the values in an **additional data structure**. We can use a **stack** for this. By pushing each node onto the stack as we traverse the list, we effectively **store** the **data values** in the **reverse order**. Once all the nodes are stored in the **stack,** we **traverse** the linked list again comparing each node's value with the values popped from the **top** of the **stack.**

Algorithm:

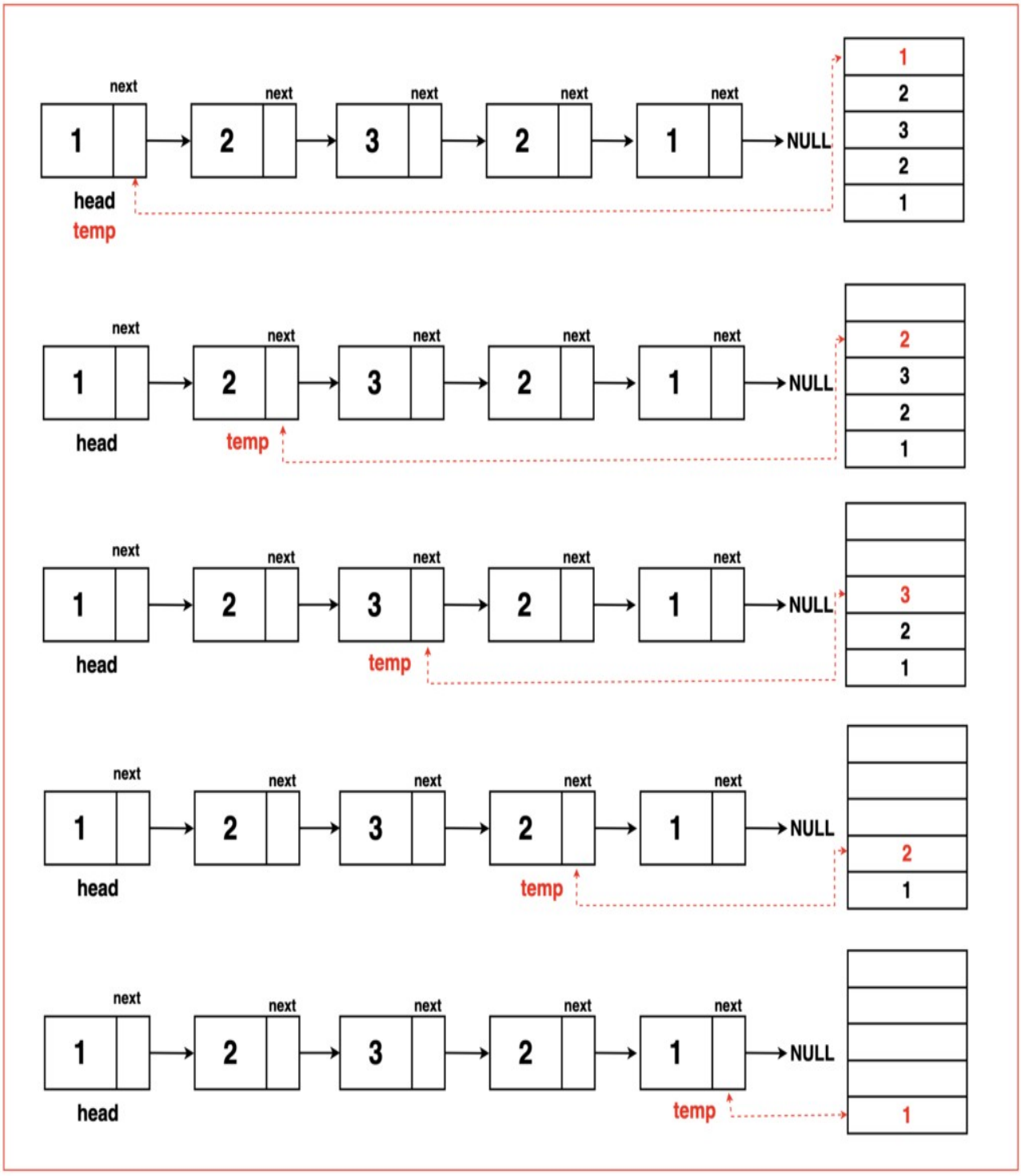
Step 1: Create an empty **stack**. This stack will be used to temporarily store the nodes from the original linked list as we traverse it.

Step 2: Traverse the linked list using a temporary variable **`temp`** till it reaches null. At each node, push the value at the current node onto the stack.



Step 3: Set variable **`temp`** back to the head of the linked list. While the stack is not empty, **compare** the **value** at the **temp** node to the value at the **top** of the stack. **Pop** the stack and move th**e temp** to the **next node** till it reaches the end.

During the **comparison**, if at any point the values do not match, the linked list is **not** a palindrome and hence returns **false.**



Step 4: If all **values match** till emp reaches the end, it means that the linked list is a **palindrome**, as the values read the same way both forward and backward hence we return **true.**

Code

```
import java.util.Stack;

// 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 data, Node next) {
        this.data = data;
        this.next = next;
    }

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


    public boolean isPalindrome(Node head) {
        // Create an empty stack
        // to store values
        Stack<Integer> st = new Stack<>();

        // Initialize a temporary pointer
        // to the head of the linked list
        Node temp = head;

        // Traverse the linked list and
        // push values onto the stack
        while (temp != null) {
            // Push the data from the
            // current node onto the stack
            st.push(temp.data);

            // Move to the next node
            temp = temp.next;
        }

        // Reset the temporary pointer back
        // to the head of the linked list
        temp = head;

        // Compare values by popping from the stack
        // and checking against linked list nodes
        while (temp != null) {
            if (temp.data != st.peek()) {
                // If values don't match,
                // it's not a palindrome
                return false;
            }
```

```
            // Pop the value from the stack
            st.pop();

            // Move to the next node
            // in the linked list
            temp = temp.next;
        }

        // If all values match,
        // it's a palindrome
        return true;
    }



    // Function to print the linked list
    public static void printLinkedList(Node head) {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Create a linked list with
        // values 1, 5, 2, 5, and 1 (15251, a palindrome)
        Node head = new Node(1);
        head.next = new Node(5);
        head.next.next = new Node(2);
        head.next.next.next = new Node(5);
        head.next.next.next.next = new Node(1);

        // Print the original linked list
        System.out.print("Original Linked List: ");
        printLinkedList(head);

        // Check if the linked list is a palindrome
        if (isPalindrome(head)) {
            System.out.println("The linked list is a palindrome.");
        } else {
            System.out.println("The linked list is not a palindrome.");
        }
    }

}
```

Output: Original Linked List: 1 5 2 5 1 The linked list is a palindrome.

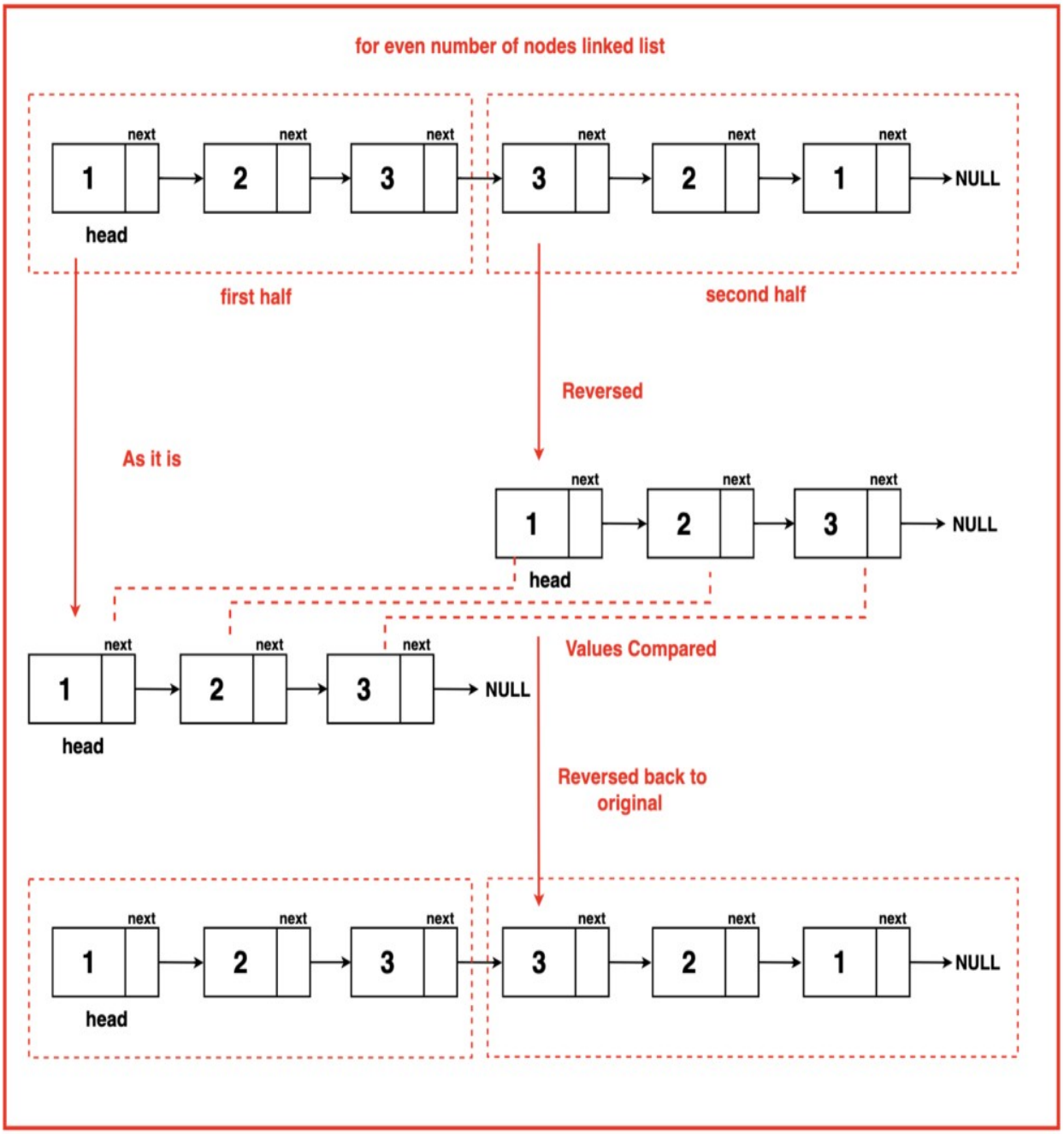
Complexity Analysis

Time Complexity: O(2 * N) This is because we **traverse** the linked list **twice**: once to push the values onto the stack, and once to pop the values and compare with the linked list. Both traversals take **O(2*N) ~ O(N) time.**

Space Complexity: O(N) We use a **stack** to store the values of the linked list, and in the worst case, the stack will have all **N values,** ie. storing the complete linked list.

Optimal Approach
Algorithm / Intuition

The previous approach uses **O(N) additional space**, which can be avoided by **reversing** only **half** of the linked list and comparing the **first** and **second halves**. If they match, reverse the portion that was originally reversed, and then return **true** else return **false.**

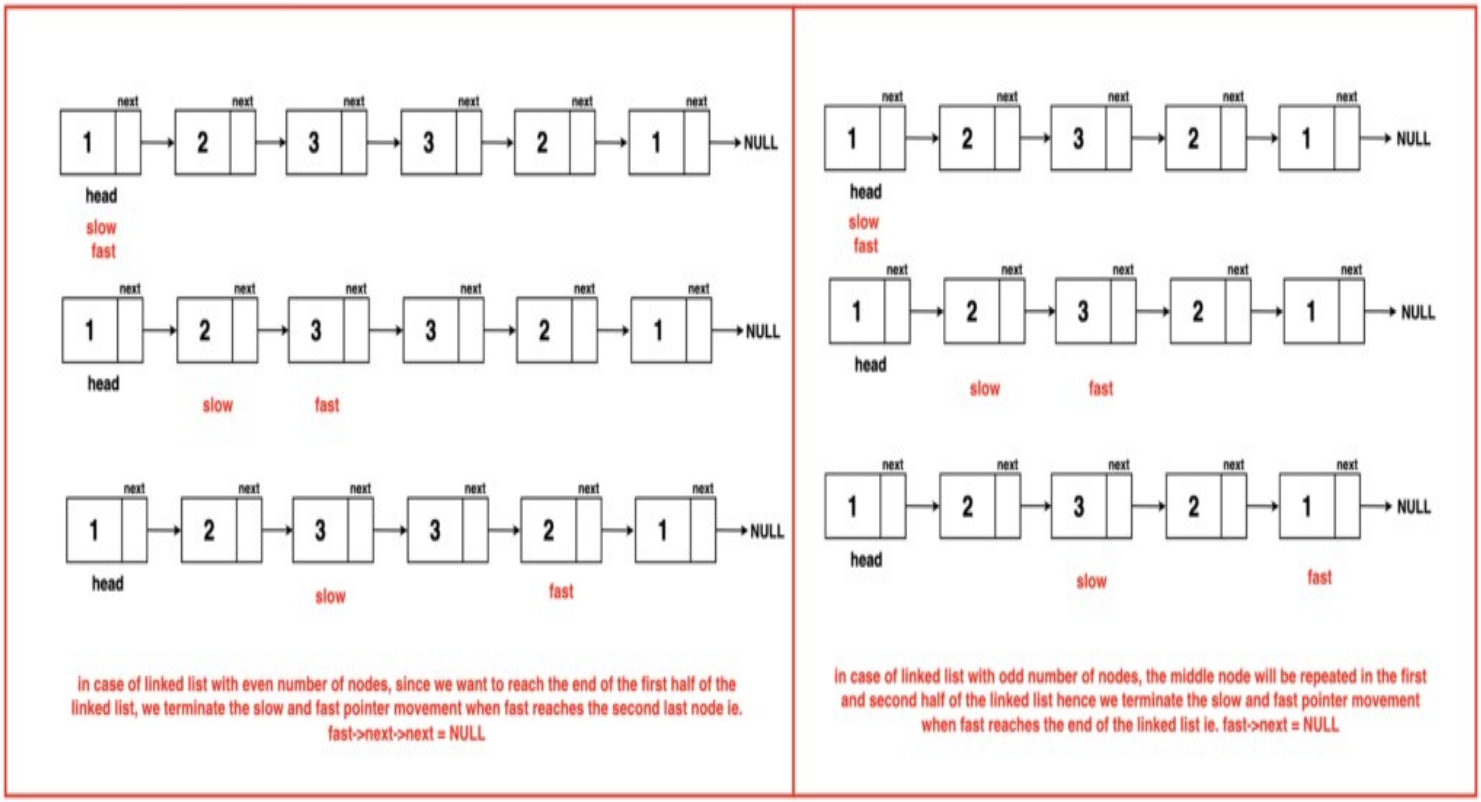


To implement this in-order reversal of the second half and its comparison with the first half has to be done in phases. The first step is dividing the first and second half of the linked list by recognizing the middle node using the **Tortoise and Hare Algorithm.**

Algorithm:

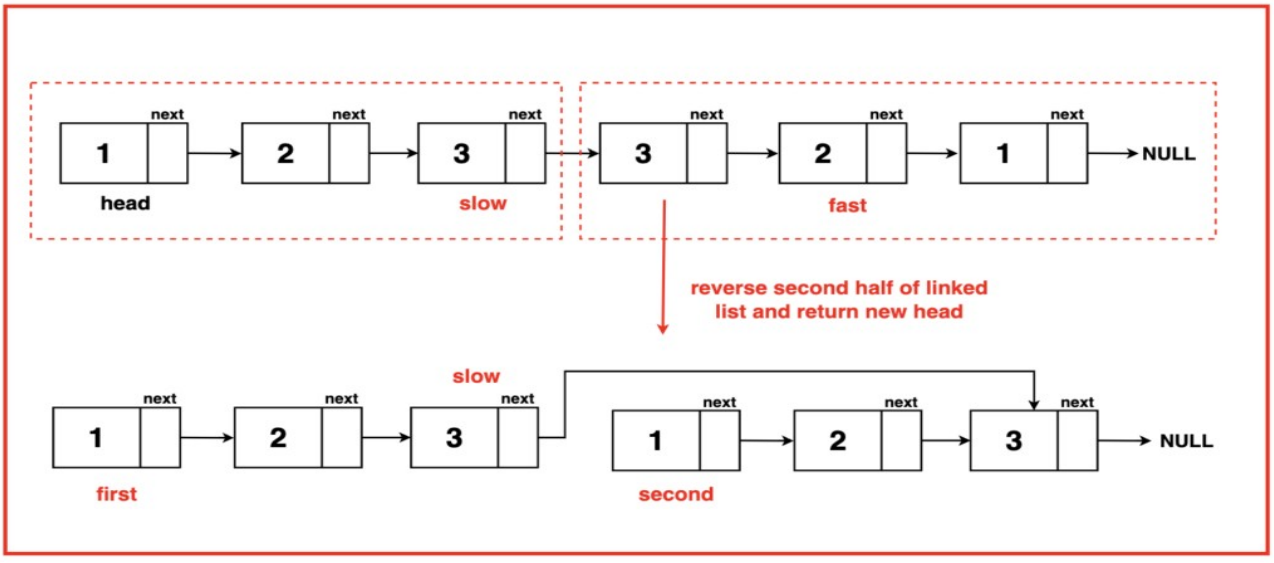
Step 1: Check if the linked list is empty or has only one node. If that's the case, it is a palindrome by definition, so return **true**.

Step 2: Initialise two pointers, **‘slow’** and **‘fast’**, to find the middle of the linked list using the Tortoise and Hare Algorithm. The **‘slow’** pointer advances by one step at a time, while the **‘fast’** pointer advances by two steps at a time. Continue this until the **‘fast’** pointer reaches the end of the list or is the second last on the list. The ‘slow’ pointer will now be in the middle of the linked list.

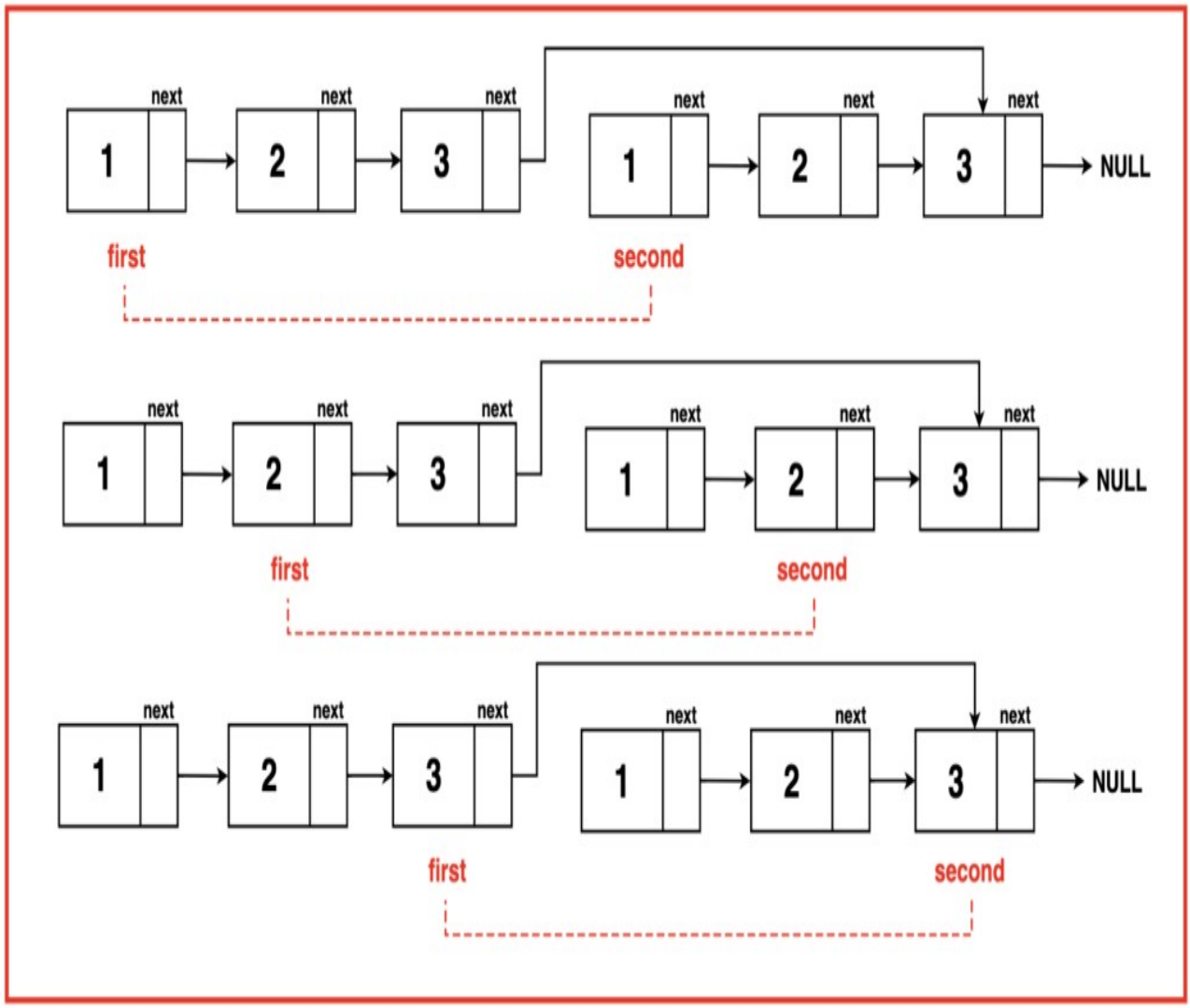


Step 3: Reverse the **second half** of the linked list starting from the middle (the **‘slow->next’** node). This is done by calling the reverse linked list function and returning the head of the new reversed linked list.

Step 4: Create two pointers, **‘first’** and **‘second’**, where **‘first’** points to the head of the linked list, and **‘second’** points to the new head of the reversed second half.



Step 5: Compare data values of nodes from both halves. If the values do not match, it means the list is not a palindrome. In this case, return **'false'**. Continue moving both **'first'** and **'second'** pointers through their **respective halves**, comparing the data values until one of them reaches the end of the list.



Step 6: After the comparison, **reverse** the **second half** back to its original state using the reverse linked list function and **join back** the linked list to its original state. Since all the values matched in the first half and reversed in the second half, return **true**. In case it does not match, return **false.**

Code

```
import java.util.Stack;

// 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 data, Node next) {
        this.data = data;
        this.next = next;
    }

    // Constructor with only data as
    // a parameter, sets next to null
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
    // Function to reverse a linked list
    // using the recursive approach
    public Node reverseLinkedList(Node head) {
        // Check if the list is empty or has only one node
        if (head == null || head.next == null) {

            // No change is needed;
            // return the current head
            return head;
        }

        // Recursive step: Reverse the remaining
        // part of the list and get the new head
        Node newHead = reverseLinkedList(head.next);

        // Store the next node in 'front'
        // to reverse the link
        Node front = head.next;

        // Update the 'next' pointer of 'front' to
        // point to the current head, effectively
        // reversing the link direction
        front.next = head;

        // Set the 'next' pointer of the
        // current head to 'null' to
        // break the original link
        head.next = null;

        // Return the new head obtained
        // from the recursion
        return newHead;
    }

    public static boolean isPalindrome(Node head) {
        // Check if the linked list is
        // empty or has only one node
        if (head == null || head.next == null) {
            // It's a palindrome by definition
            return true;
        }

        // Initialize two pointers, slow and fast,
        // to find the middle of the linked list
        Node slow = head;
        Node fast = head;

        // Traverse the linked list to find the
        // middle using slow and fast pointers
        while (fast.next != null && fast.next.next != null) {
            // Move slow pointer one step at a time
            slow = slow.next;
```

```
        // Move fast pointer two steps at a time
        fast = fast.next.next;
    }

    // Reverse the second half of the
    // linked list starting from the middle
    Node newHead = reverseLinkedList(slow.next);

    // Pointer to the first half
    Node first = head;

    // Pointer to the reversed second half
    Node second = newHead;
    while (second != null) {
        // Compare data values of
        // nodes from both halves

        // If values do not match, the
        // list is not a palindrome
        if (first.data != second.data) {

            // Reverse the second half back
            // to its original state
            reverseLinkedList(newHead);

            // Not a palindrome
            return false;
        }

        // Move the first pointer
        first = first.next;

        // Move the second pointer
        second = second.next;
    }

    // Reverse the second half back
    // to its original state
    reverseLinkedList(newHead);

    // The linked list is a palindrome
    return true;
}


// Function to print the linked list
public static void printLinkedList(Node head) {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    // Create a linked list with
    // values 1, 5, 2, 5, and 1 (15251, a palindrome)
    Node head = new Node(1);
    head.next = new Node(5);
    head.next.next = new Node(2);
    head.next.next.next = new Node(5);
    head.next.next.next.next = new Node(1);
```

```
        // Print the original linked list
        System.out.print("Original Linked List: ");
        printLinkedList(head);

        // Check if the linked list is a palindrome
        if (isPalindrome(head)) {
            System.out.println("The linked list is a palindrome.");
        } else {
            System.out.println("The linked list is not a palindrome.");
        }
    }

}
```

Output: Original Linked List: 1 5 2 5 1 The linked list is a palindrome.

Complexity Analysis

Time Complexity: O (2* N) The algorithm traverses the **linked list twice**, dividing it into halves. During the **first traversal**, it **reverses** one-half of the list, and during the **second traversal**, it **compares** the elements of both halves. As each traversal covers **N/2 elements**, the time complexity is calculated as **O(N/2 + N/2 + N/2 + N/2)**, which simplifies to **O(2N)**, ultimately representing **O(N)**.

Space Complexity: O(1) The approach uses a **constant amount** of **additional space** regardless of the size of the input linked list. It **doesn't allocate** any new data structures that depend on the input size, resulting in a space complexity of O(1).