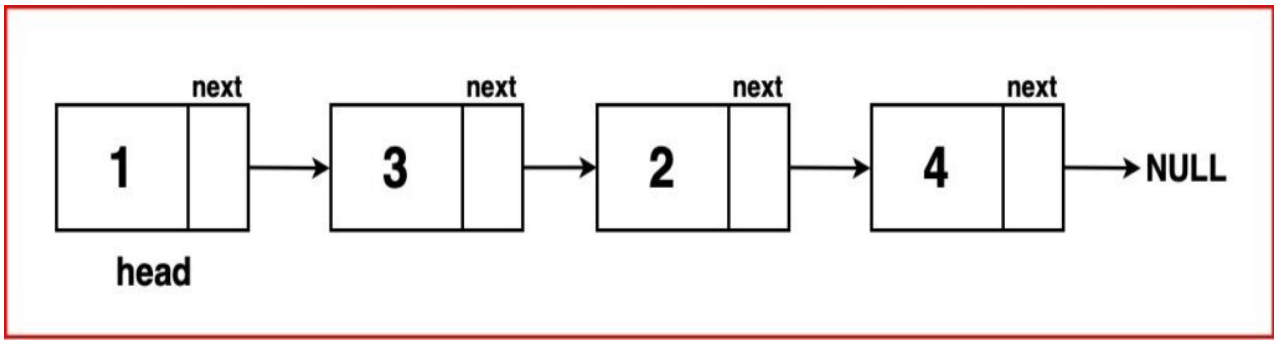
Reverse a Linked List

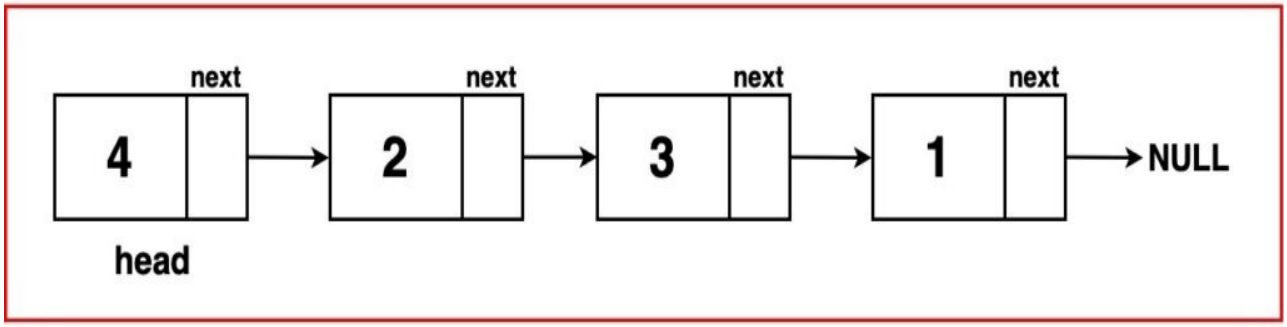
Problem Statement: Problem Statement: Given the **head** of a singly linked list, write a program to reverse the linked list, and return the **head** pointer to the **reversed list**.

Examples

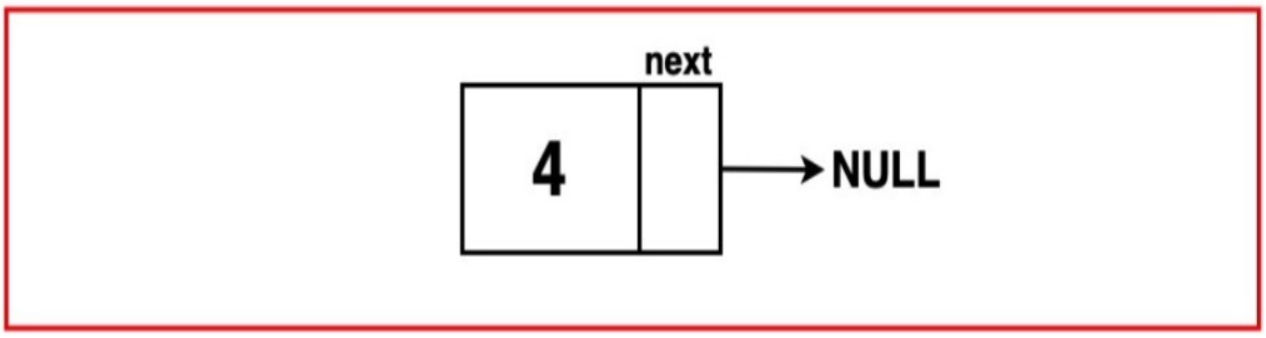
Example 1:
Input Format:
LL: 1 3 2 4



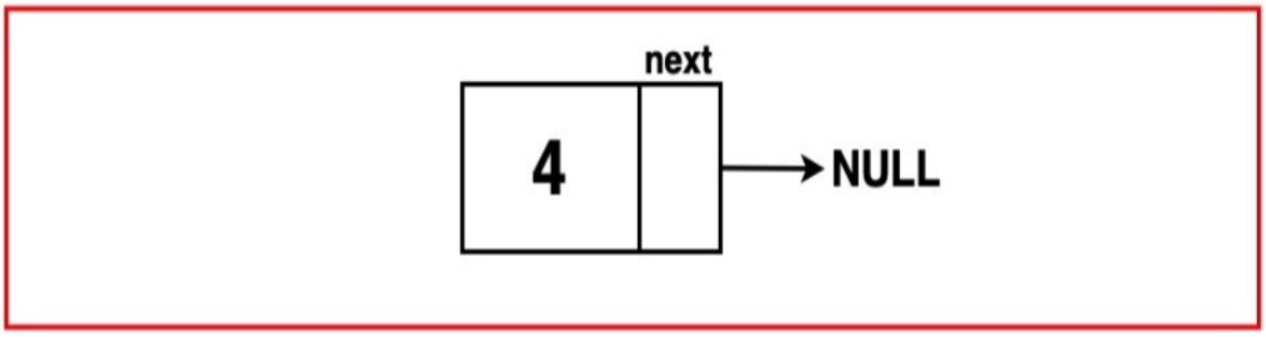
Output: 3



Explanation: After reversing the linked list, the new head will point to the tail of the old linked list.
Example 2:
Input Format:
LL: 4



Output: 4



Explanation: In this example, the linked list contains only one node hence reversing this linked list will result in the same list as the original.

Brute Force Approach
Algorithm / Intuition

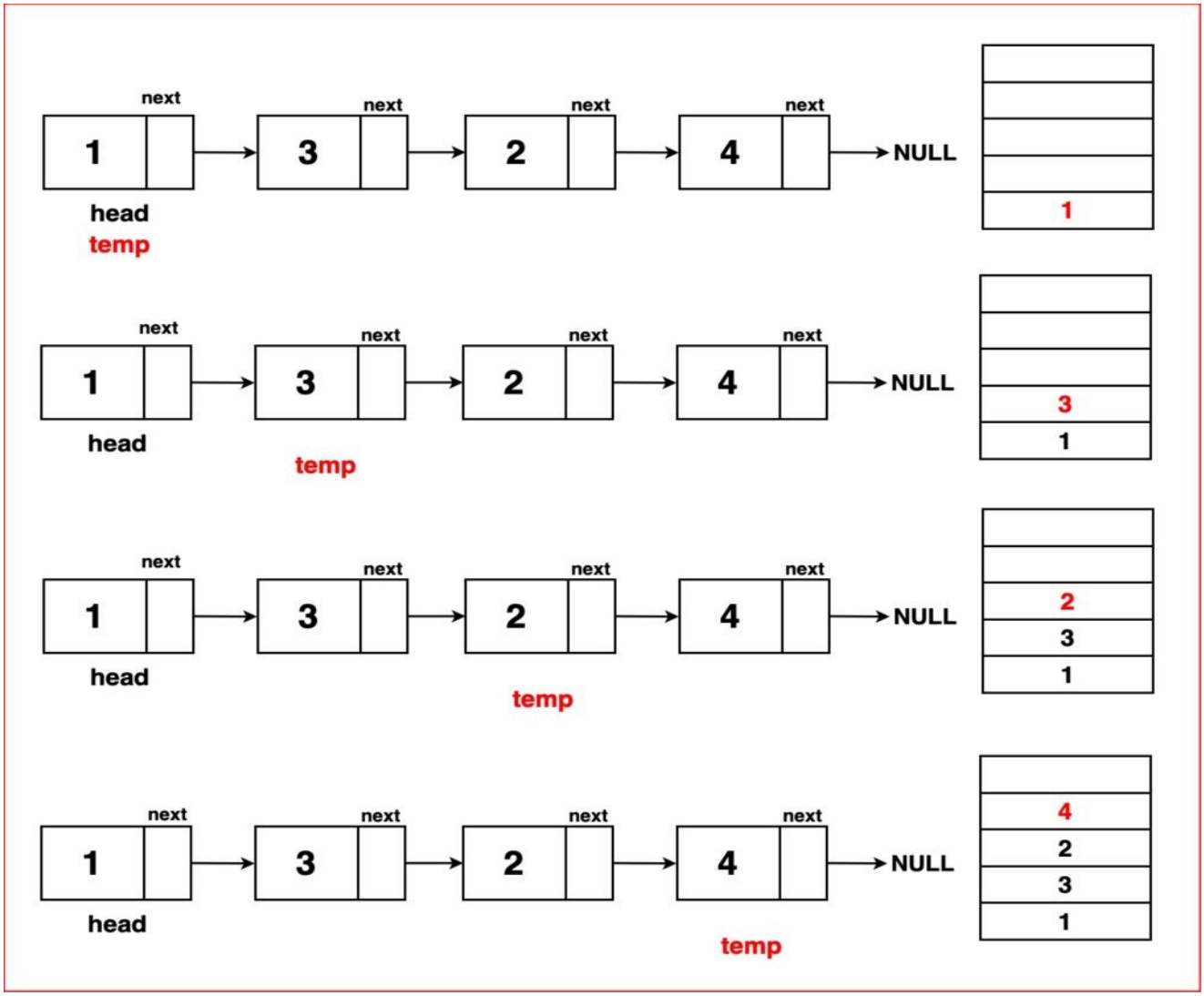
Approach 1 : Brute Force

A straightforward approach to reversing a singly linked list requires an **additional data structure** to temporarily store the values. We can use a **stack** for this. By pushing each node onto the stack as we move through the list, we effectively **reverse the order** of the nodes. Once all the nodes are stored in the **stack**, we rebuild the reversed linked list by **popping nodes** from the stack and **assigning** them to the nodes. The result is a new linked list with the elements in the **opposite order** of the original list.

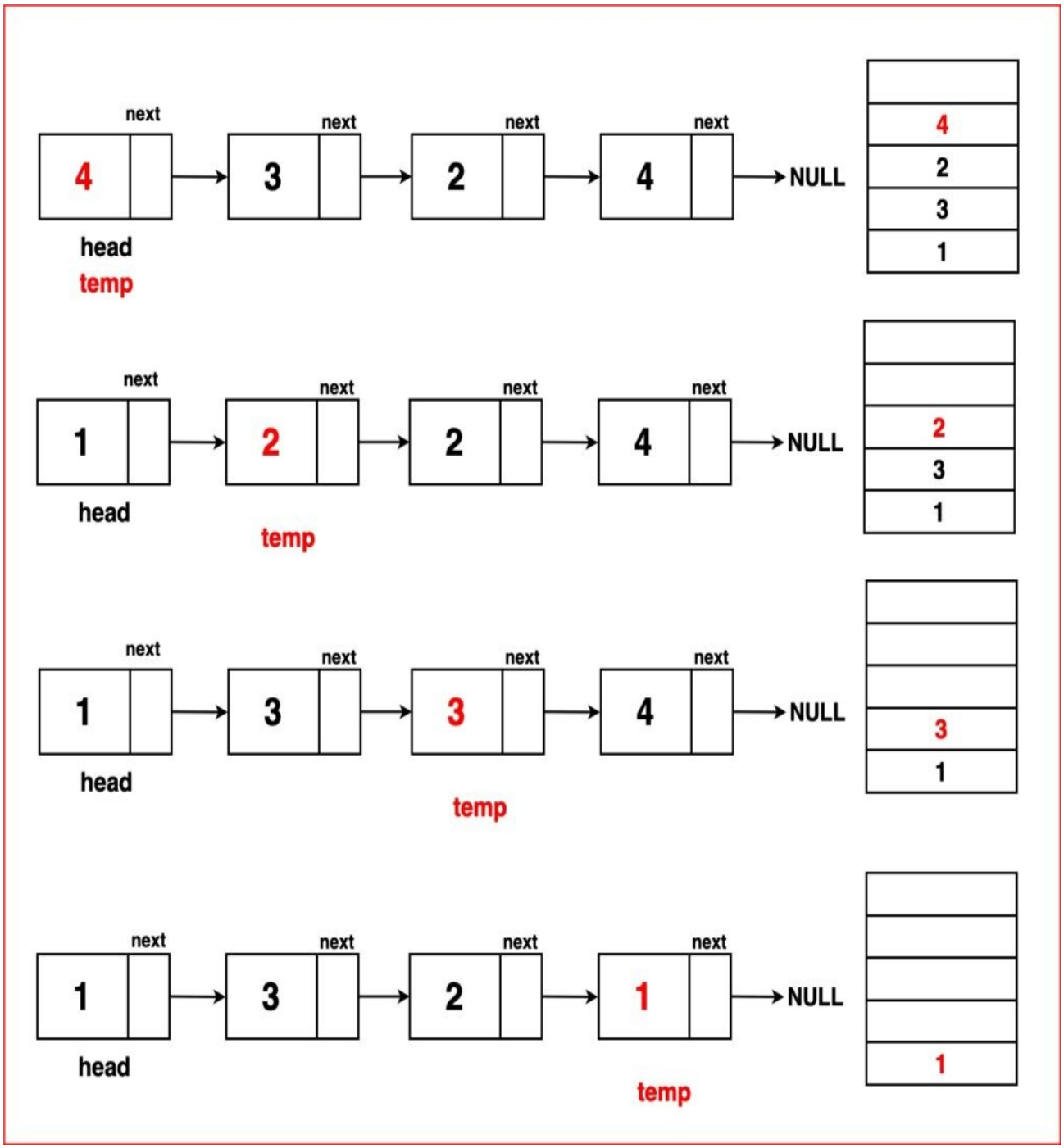
Algorithm:

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, set the **value** at the **temp** node to the value at the **top** of the stack. **Pop** the stack and move **temp** to the **next node** till it reaches null.



Step 4: Return the **head** as the new head of the **reversed** linked list.

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 class ReverseLinkedListUsingStack {

    // Function to reverse the
    // linked list using a stack
    public static Node reverseLinkedList(Node head) {
        // Create a temporary pointer to
        // traverse the linked list
        Node temp = head;

        // Create a stack to temporarily
        //store the data values
        Stack<Integer> stack = new Stack<>();

        // Step 1: Push the values of the
        // linked list onto the stack
        while (temp != null) {
             // Push the current node's
             // data onto the stack
            stack.push(temp.data);
             // Move to the next node
             // in the linked list
            temp = temp.next;
        }
         // Reset the temporary pointer
         // to the head of the linked list
        temp = head;

        // Step 2: Pop values from the stack
        // and update the linked list
        while (temp != null) {
            // Set the current node's data
            // to the value at the top of the stack
            temp.data = stack.pop();
```

```
            // Move to the next node
            // in the linked list
            temp = temp.next;
        }
        // Return the new head of
        // the reversed linked list
        return head;
    }

    // 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, 3, 2, and 4
        Node head = new Node(1);
        head.next = new Node(3);
        head.next.next = new Node(2);
        head.next.next.next = new Node(4);

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

        // Reverse the linked list
        head = reverseLinkedList(head);

        // Print the reversed linked list
        System.out.print("Reversed Linked List: ");
        printLinkedList(head);
    }
}
```

Output: Original Linked List: 1 3 2 4 Reversed Linked List: 4 2 3 1

Complexity Analysis

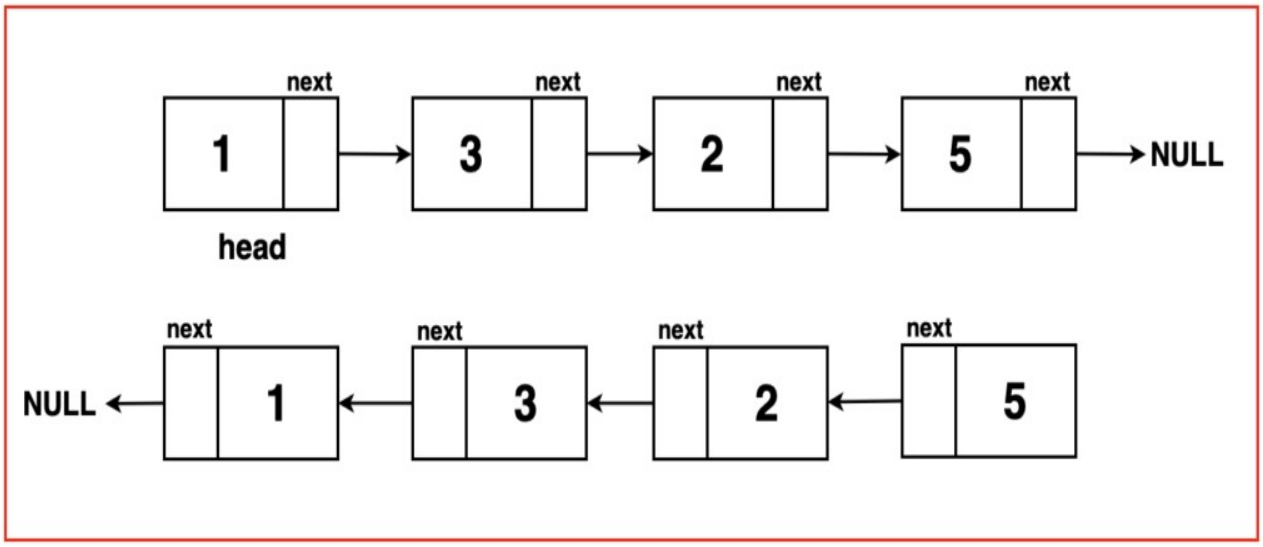
Time Complexity: O(2N) This is because we **traverse** the linked list **twice**: once to push the values onto the stack, and once to pop the values and update the linked list. Both traversals take O(N) time, hence time complexity O(2N) ~ O(N).

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 1
Algorithm / Intuition

Approach 2: Reverse Links in place (Iterative)

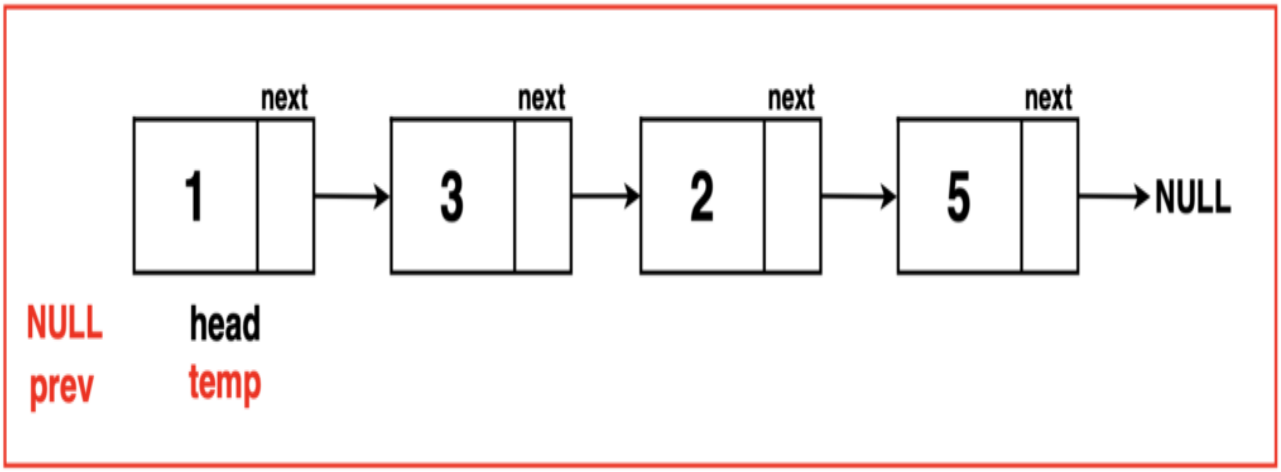
The previous approach uses O(N) addition space which can be avoided by interchanging the connecting links of the nodes of the linked list in place.



The main idea is to **flip** the order of connections in the linked list, which changes the **direction** of the **arrows**. When this happens, the **last element** becomes the **new first element** of the list. This **in-place reversal** allows us to efficiently transform the original list **without using extra space.**

Algorithm:

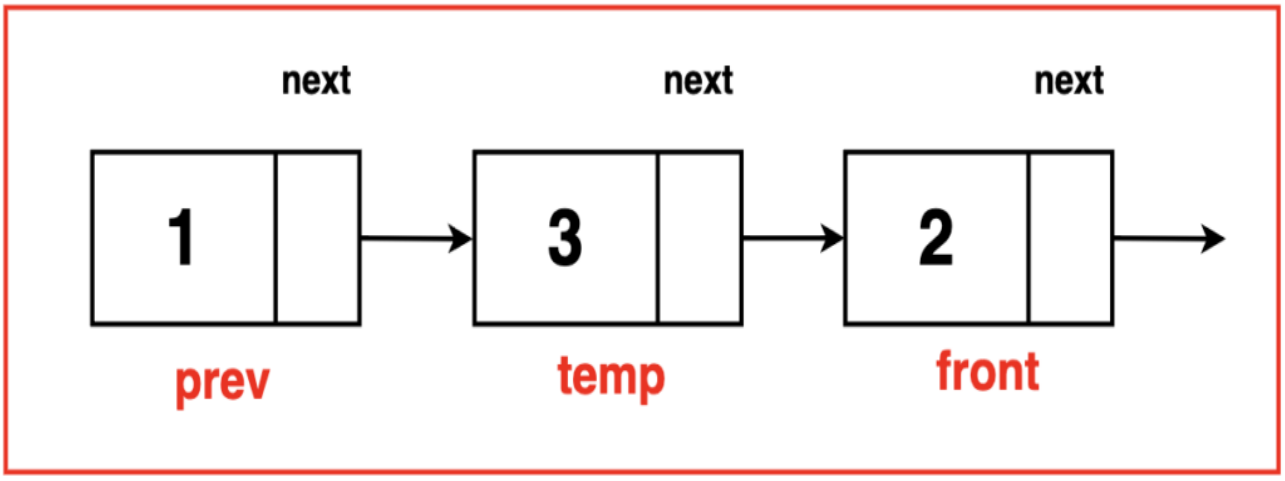
Step 1: Initialise a '**temp'** pointer at the head of the linked list. This pointer will be used to traverse the linked list. And initialize the pointer **'prev'** to **'NULL'** to keep track of the previous node. This will be used to reverse the direction of the **'next'** pointers.



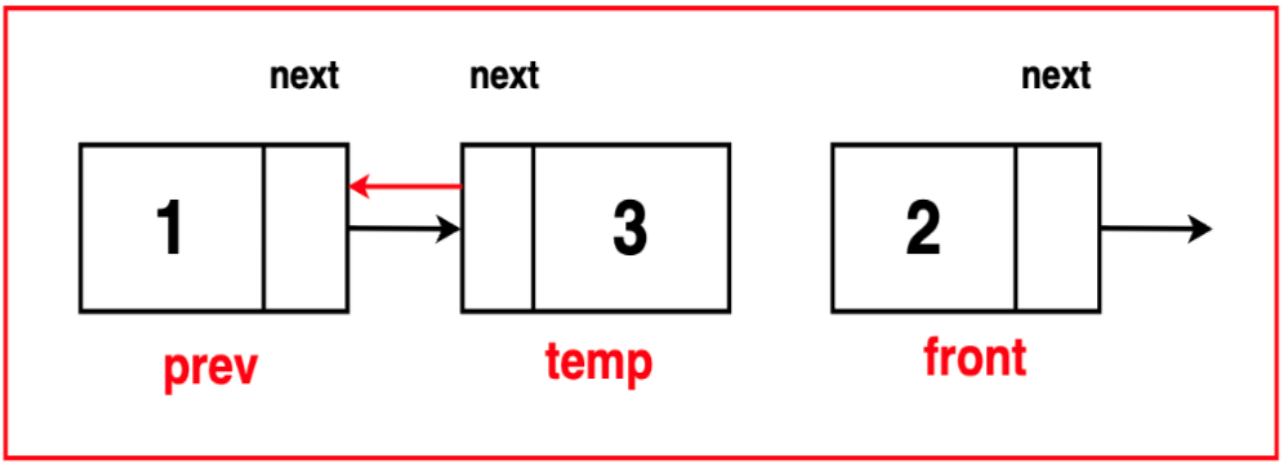
Step 2: Traverse the entire linked list by moving through each node using the **'temp'** pointer until it reaches the end (marked as **'NULL'**).

At each iteration within the traversal,

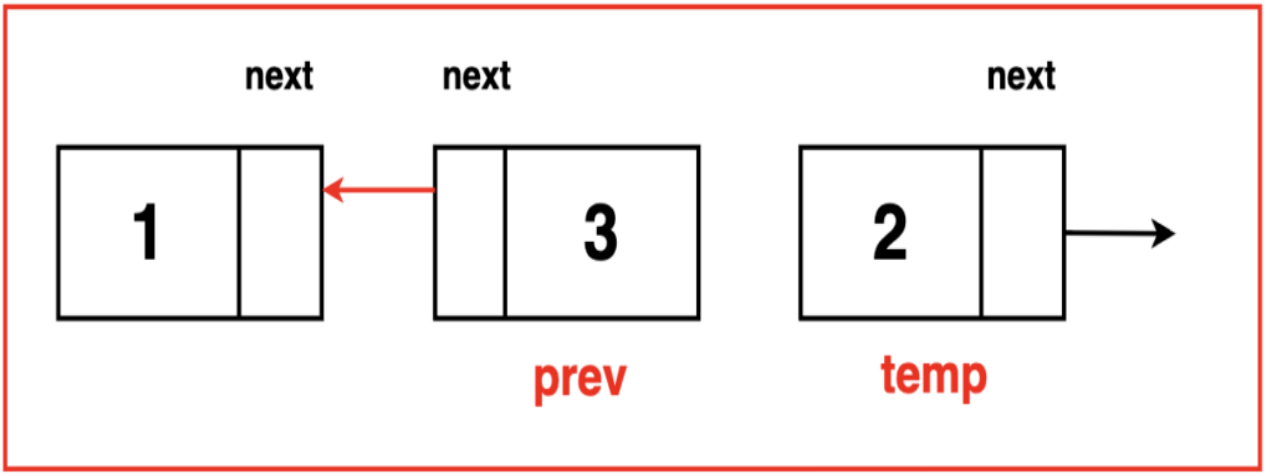
1. Save the reference to the next node that **'temp'** is pointing to in a variable called **'front'**. This helps retain the link to the subsequent node before altering the **'next'** pointer.



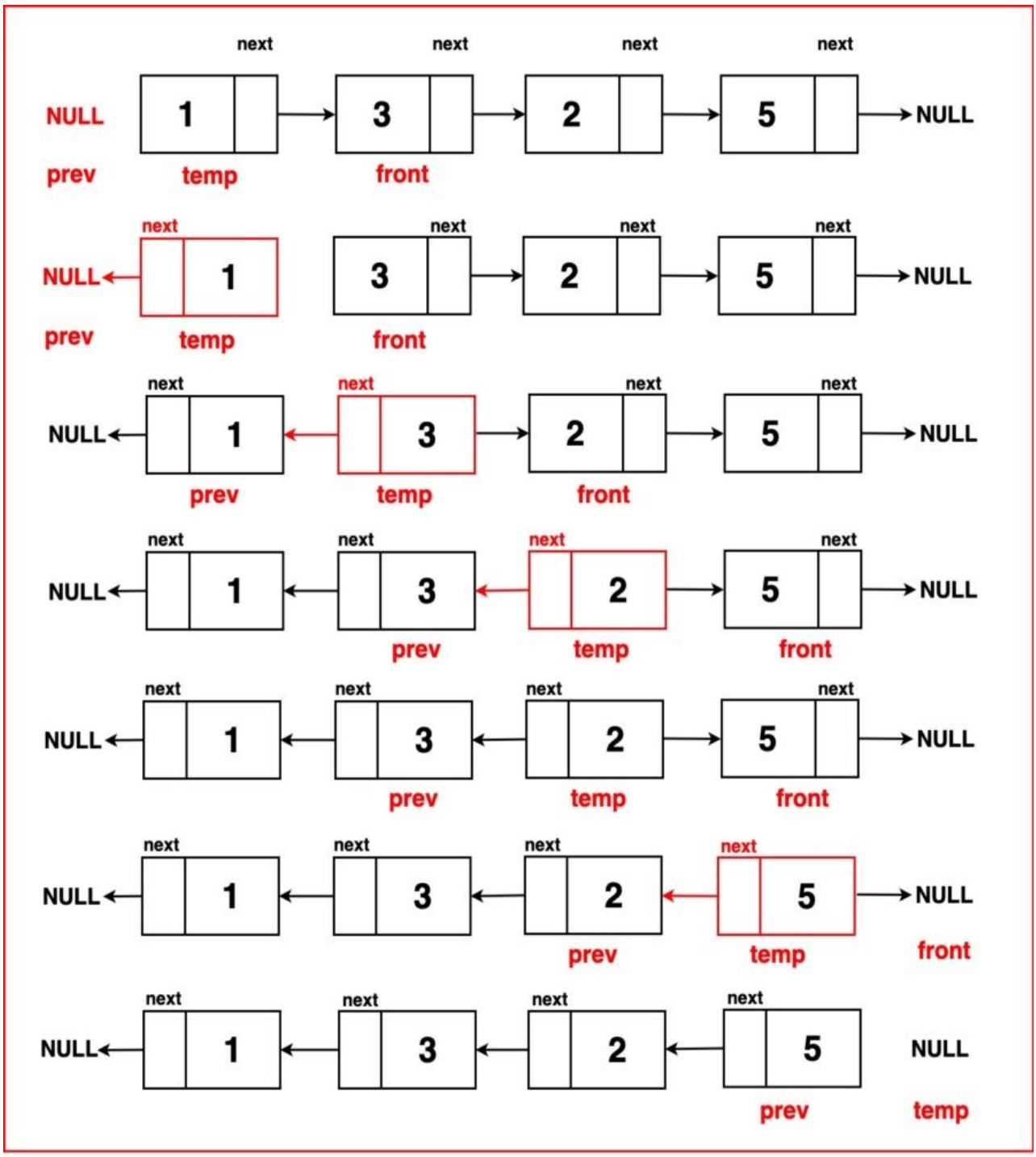
2. Reverse the direction of the **‘next’** pointer of the current node (pointed to by **‘temp’**) to point to the **‘prev’** node. This effectively reversed the direction of the linked list, making the current node point to the previous node.



3. Move the **‘prev’** pointer to the current node. This sets up the **‘prev’** pointer for the next iteration of the loop.
4. Move the **‘temp’** pointer to the **‘front’** node. This advances the traversal to the next node in the original order.



In summary:



Step 3: Keep traversing through the linked list using the **'temp'** pointer until it reaches the **end**, thereby reversing the entire list. Once the **'temp'** pointer reaches the end, return the **new head** of the reversed linked list, which is now indicated by the **'prev'** pointer. This **'prev'** pointer becomes the first node in the newly reversed list.

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 3-pointer approach
public class ReverseLinkedListUsingStack(Node head) {

   // Initialize'temp' at
   // head of linked list
   Node temp = head;

   // Initialize pointer 'prev' to NULL,
   // representing the previous node
   Node prev = null;

   // Traverse the list, continue till
   // 'temp' reaches the end (NULL)
   while(temp != null){
       // Store the next node in
       // 'front' to preserve the reference
       Node front = temp.next;

       // Reverse the direction of the
       // current node's 'next' pointer
       // to point to 'prev'
       temp.next = prev;

        // Move 'prev' to the current
        // node for the next iteration
       prev = temp;

        // Move 'temp' to the 'front' node
        // advancing the traversal
       temp = front;
   }

   // Return the new head of
   // the reversed linked list
   return prev;
```

```
    }

    // 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, 3, 2, and 4
        Node head = new Node(1);
        head.next = new Node(3);
        head.next.next = new Node(2);
        head.next.next.next = new Node(4);

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

        // Reverse the linked list
        head = reverseLinkedList(head);

        // Print the reversed linked list
        System.out.print("Reversed Linked List: ");
        printLinkedList(head);
    }
}
```

Output: Original Linked List: 1 3 2 4 Reversed Linked List: 4 2 3 1

Complexity Analysis

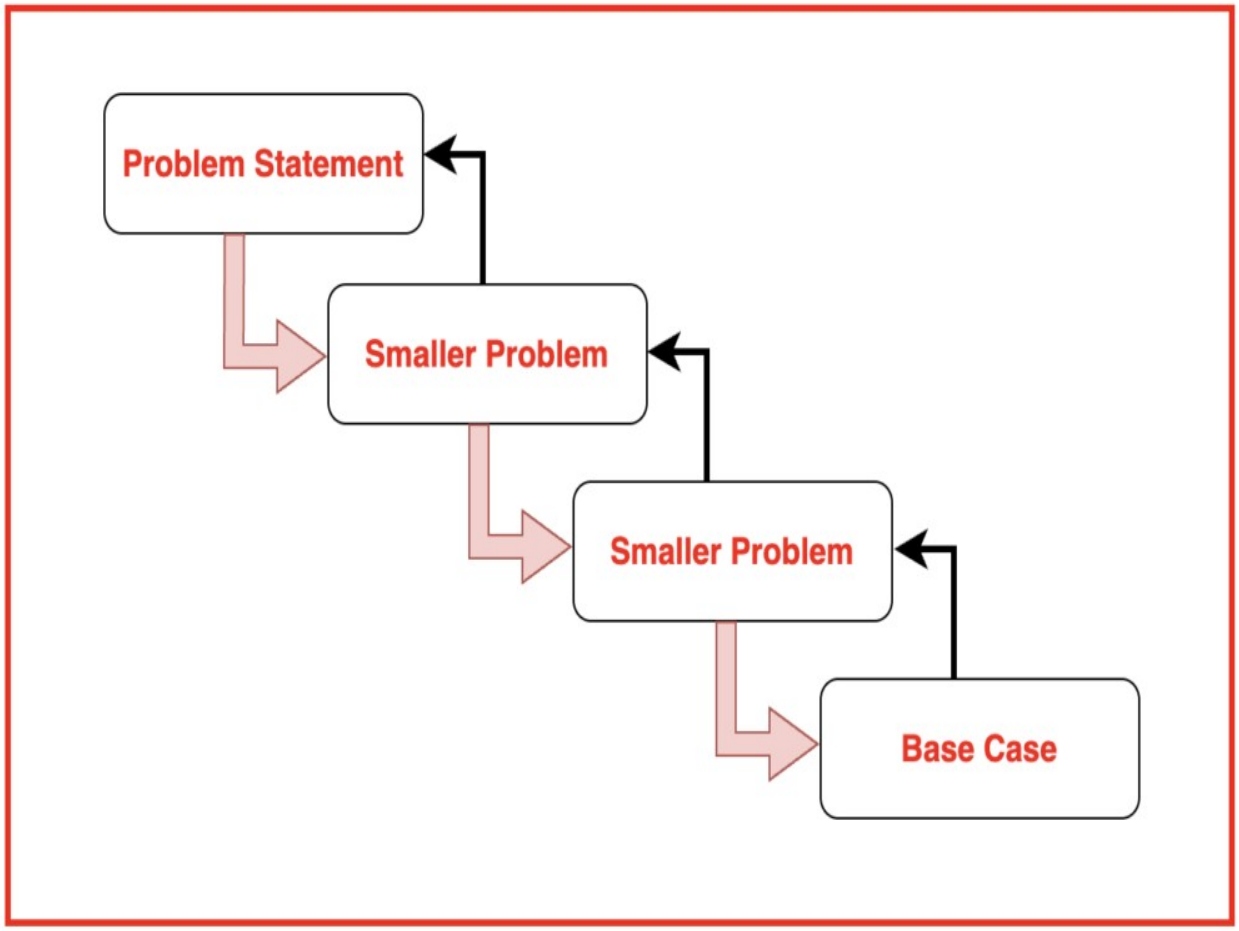
Time Complexity: O(N) The code **traverses** the **entire linked list** once, where 'n' is the number of nodes in the list. This traversal has a **linear time complexity**, O(n).

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 three pointers (**prev, temp** and **front**) to reverse the list without any significant extra memory usage, resulting in **constant space** complexity, O(1).

Optimal Approach II
Algorithm / Intuition

Approach 3: Recursive

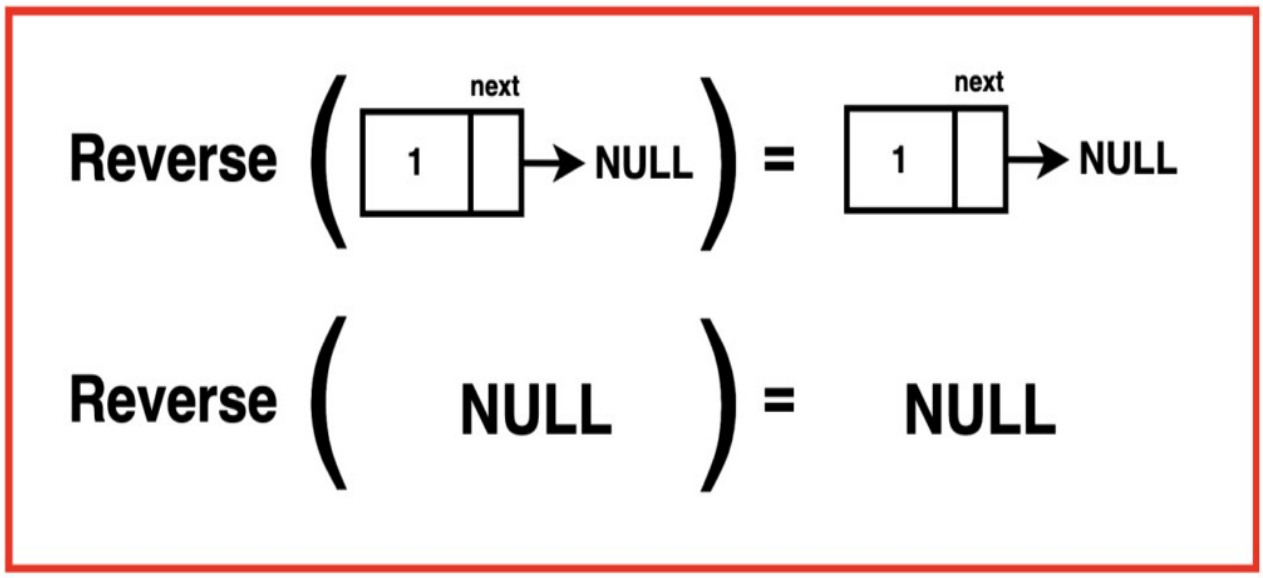
Recursion gives the ability to **break down** a problem statement into **small problems** that can be solved **incrementally**. It keeps solving these smaller problems until it reaches the smallest possible solution, often referred to as the base case, from where it can start combining the results of those smaller solutions to ultimately solve the original, larger problem.



In this case, tackling the **larger problem** involves reversing a linked list with N = 4 nodes. **Recursion** allows us to break this task down into **progressively smaller subproblems**, starting with the case of 3 nodes, then the last 2 nodes, and ultimately reaching the base case where only 1 node remains. In the **base case**, reversing the linked list is straightforward, as a list with just one node is already in its reversed form, and we can simply return it as is.

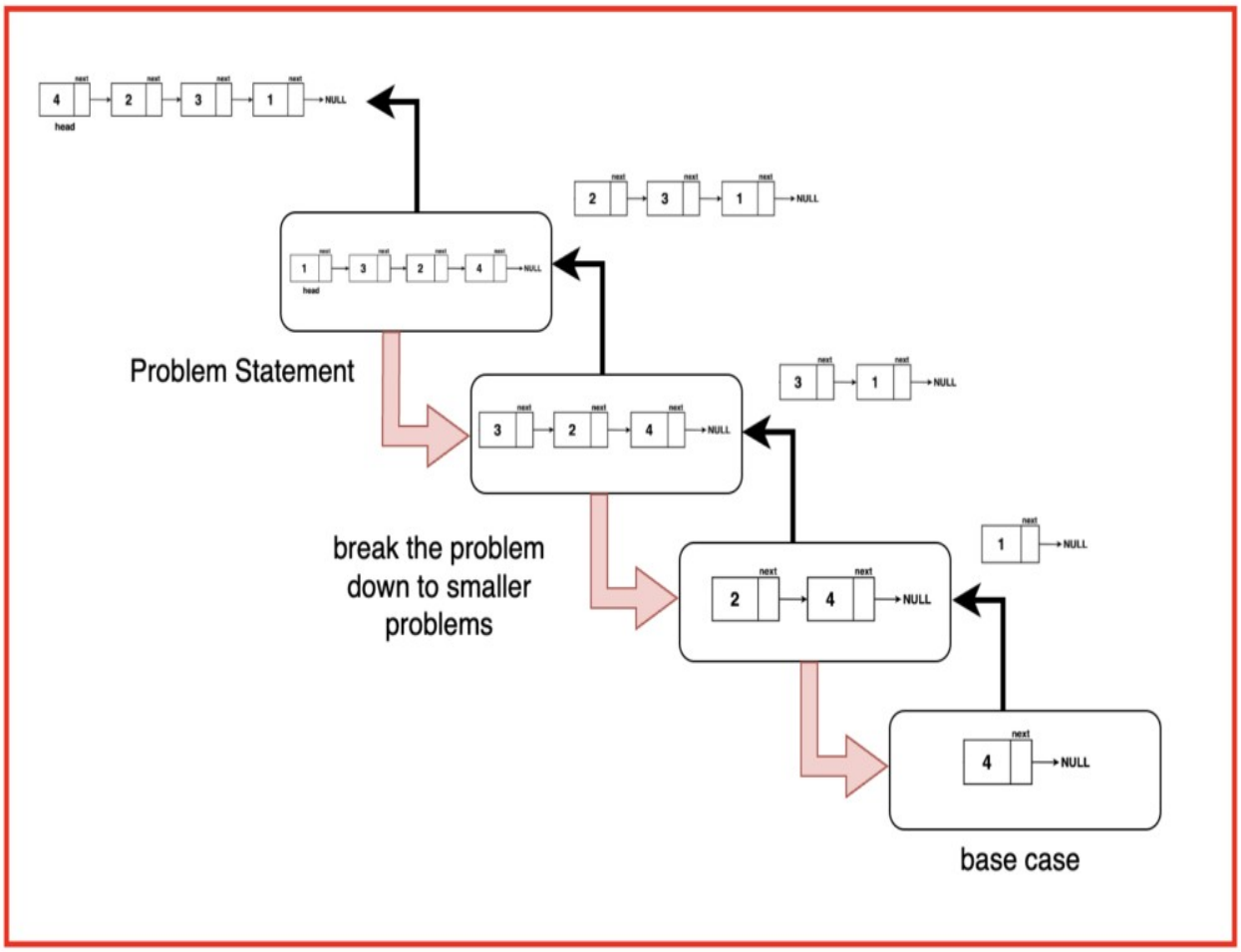
Algorithm:

Base Case:
Check if the linked list is empty or contains only one node. Return the head as it's already reversed in these cases.



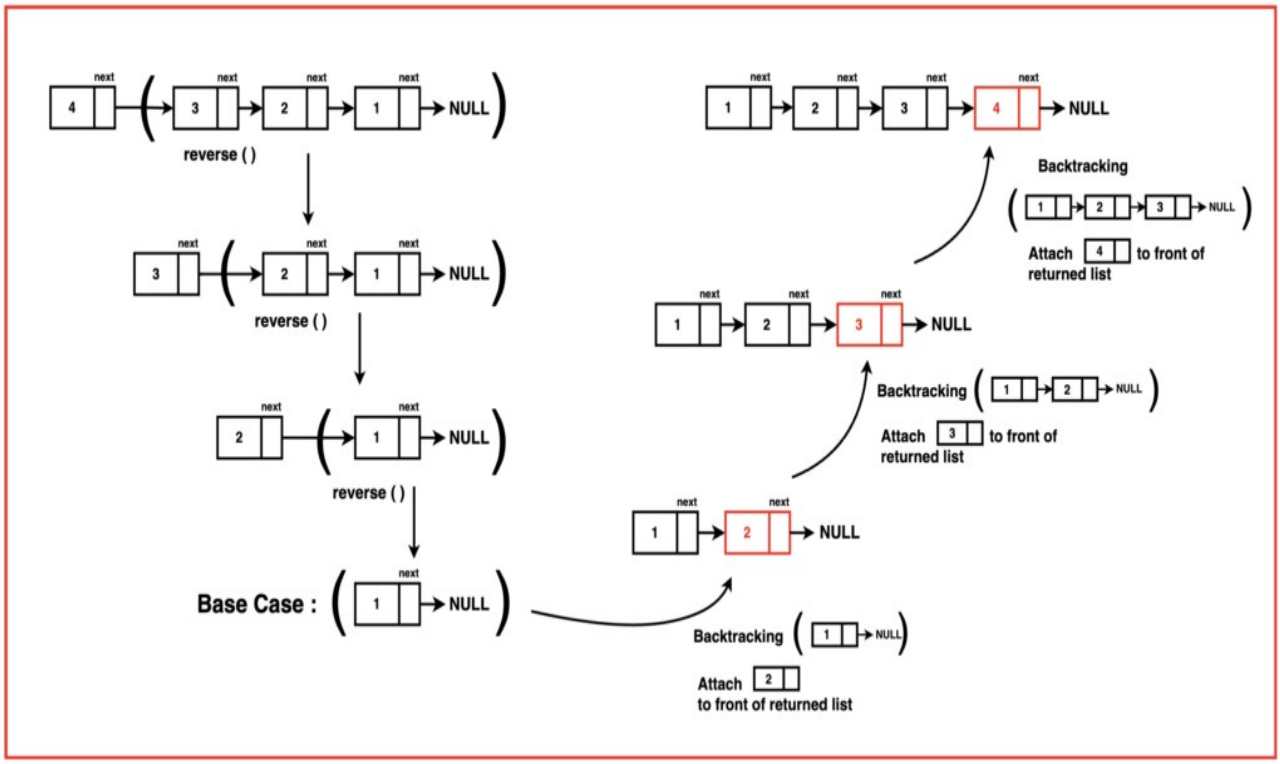
Recursive Function:
The core of the algorithm lies in implementing a recursive function responsible for reversing the linked list. This function operates based on the following principle:

If the base case conditions are not met, the function **invokes itself recursively**.
This **recursion** continues until it reaches the base case, gradually reversing the linked list starting from the **second node** (node after it) onward.



Return

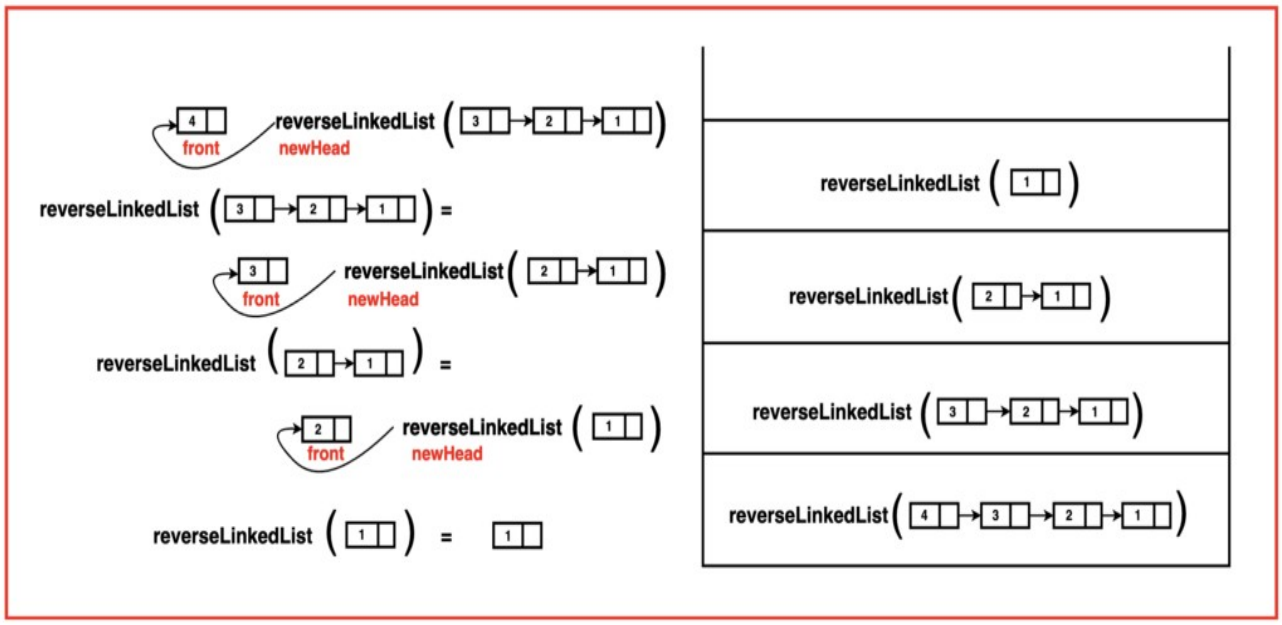
Following the recursion, the function **returns** the **new head** of the reversed linked list. This head marks the **last node** of the original list before reversal, now the **first node** in the **reversed sequence**.



Steps:

Step 1: Establish Base Case Conditions: Check if the linked list is either empty or contains only one node. If so, the list is already reversed; hence, return the head as is.

Step 2: Recursively Reverse the List: Begin the recursive step by reversing the linked list, starting from the second node. Utilise a recursive call to the reverse linked list function, passing the next node as an argument.



Step 3: Preserve Access to Remaining Nodes: To maintain access to the rest of the linked list while reversing the order, store a reference to the node following the current 'head' node. This step ensures continuity in the link sequence during reversal.

Step 4: Reverse Link Direction: Adjust the 'front' node to point to the current 'head' node in the reversed order. This action effectively reverses the link between the 'head' node and the 'front' node.

Step 5: Prevent Cyclic References: Break the link from the current 'head' node to the 'front' node to prevent any cyclic formations. Set 'head->next' to 'NULL' to ensure the reversed segment of the list does not create a loop.

Step 6: Return the New Head: Finally, return the 'newHead,' which signifies the new head of the reversed linked list. This 'newHead' was initially the last node in the list before the reversal commenced.

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 class ReverseLinkedListUsingStack {


    // Function to reverse a singly
    // linked list using a recursion
    public static Node reverseLinkedList(Node head) {
        // Base case:
        // If the linked list is empty or has only one node,
        // return the head as it is already reversed.
        if (head == null || head.next == null) {
            return head;
        }

        // Recursive step:
        // Reverse the linked list starting
        // from the second node (head.next).
        Node newHead = reverseLinkedList(head.next);

        // Save a reference to the node following
        // the current 'head' node.
        Node front = head.next;

        // Make the 'front' node point to the current
        // 'head' node in the reversed order.
        front.next = head;

        // Break the link from the current 'head' node
        // to the 'front' node to avoid cycles.
        head.next = null;

        // Return the 'newHead,' which is the new
        // head of the reversed linked list.
        return newHead;
    }


    // 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, 3, 2, and 4
        Node head = new Node(1);
        head.next = new Node(3);
        head.next.next = new Node(2);
        head.next.next.next = new Node(4);

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

        // Reverse the linked list
        head = reverseLinkedList(head);
```

```
        // Print the reversed linked list
        System.out.print("Reversed Linked List: ");
        printLinkedList(head);
    }
}
```

Complexity Analysis

Time Complexity: O(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 update the linked list. Both traversals take O(N) time.

Space Complexity : O(1) No additional space is used explicitly for **data structures** or **allocations** during the linked list reversal process. However, it's important to note that there is an implicit use of **stack space** due to **recursion**. This recursive stack space **stores function calls** and **associated variables** during the recursive traversal and reversal of the linked list. Despite this, no extra memory beyond the program's existing execution space is allocated, hence maintaining a space complexity of **O(1)**.