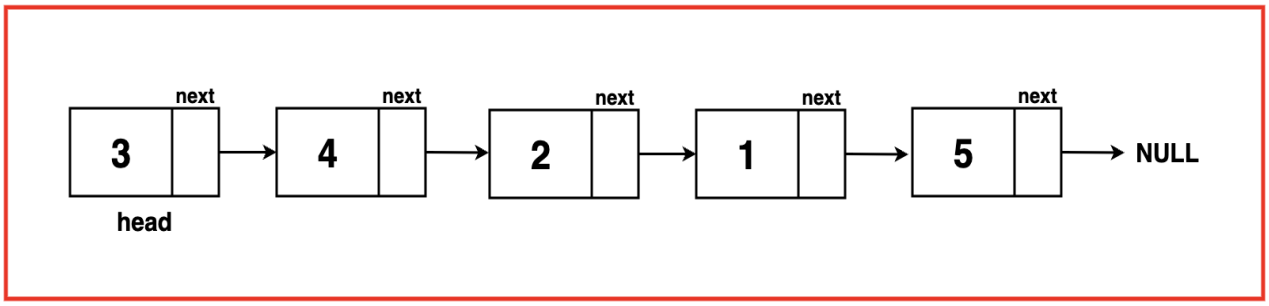
Sort a Linked List

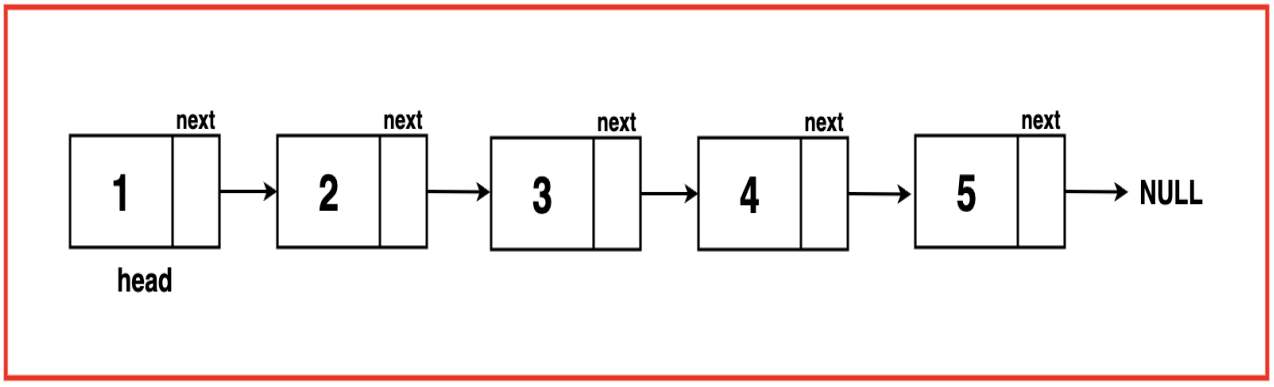
Problem Statement: Given a linked list, sort its nodes based on the data value in them. Return the head of the sorted linked list.

Examples

Example 1:
Input: Linked List: 3 4 2 1 5

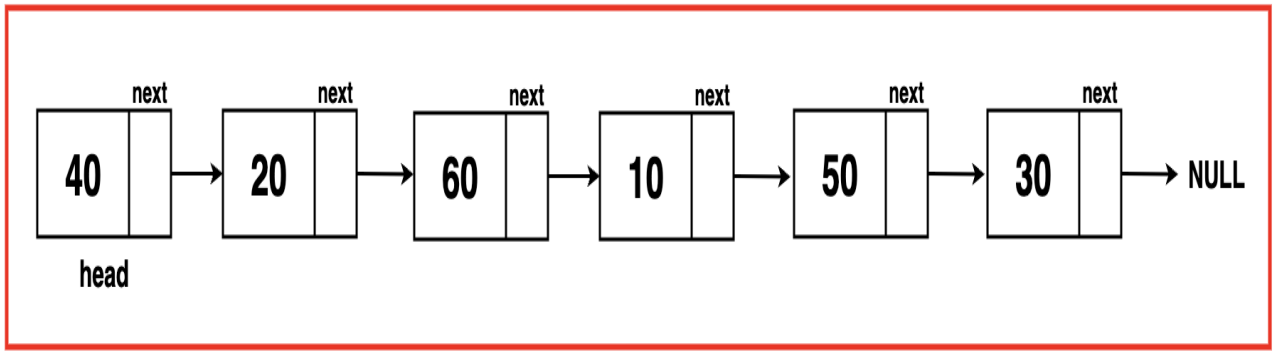


Output: Sorted List: 1 2 3 4 5

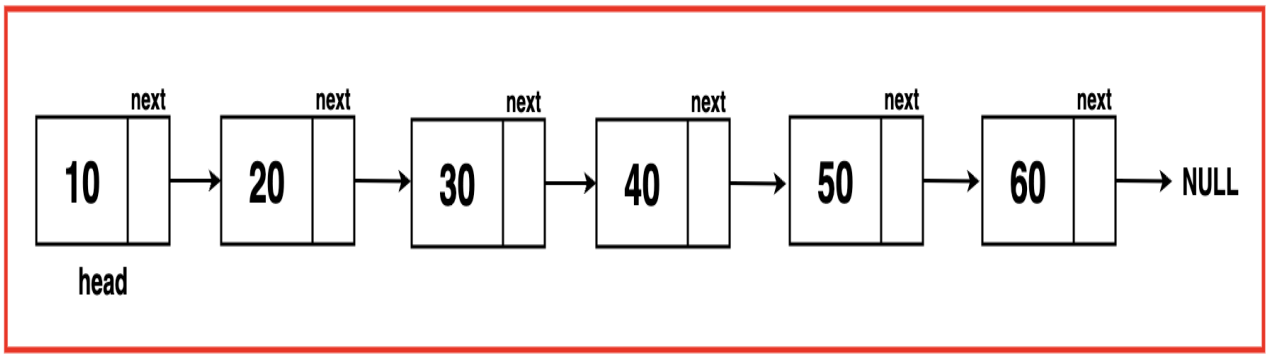


Explanation: The input linked list when sorted from [3, 4, 2, 1, 5] results in a linked list with values: [1, 2, 3, 4, 5].

Example 2:
Input: List: 40 20 60 10 50 30



Output: Sorted List: 10 20 30 40 50 60



Explanation: The input linked list when sorted from [40, 20, 60, 10, 50, 30] results in a linked list with values: [10, 20, 30, 40, 50, 60].

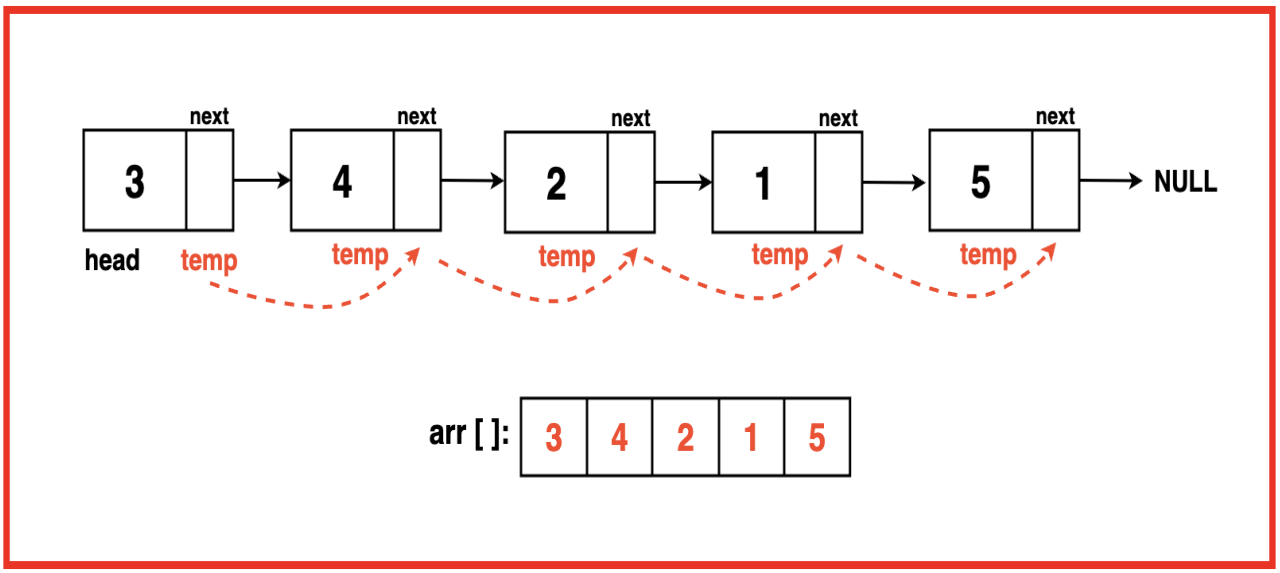
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Brute Force Approach
Algorithm / Intuition

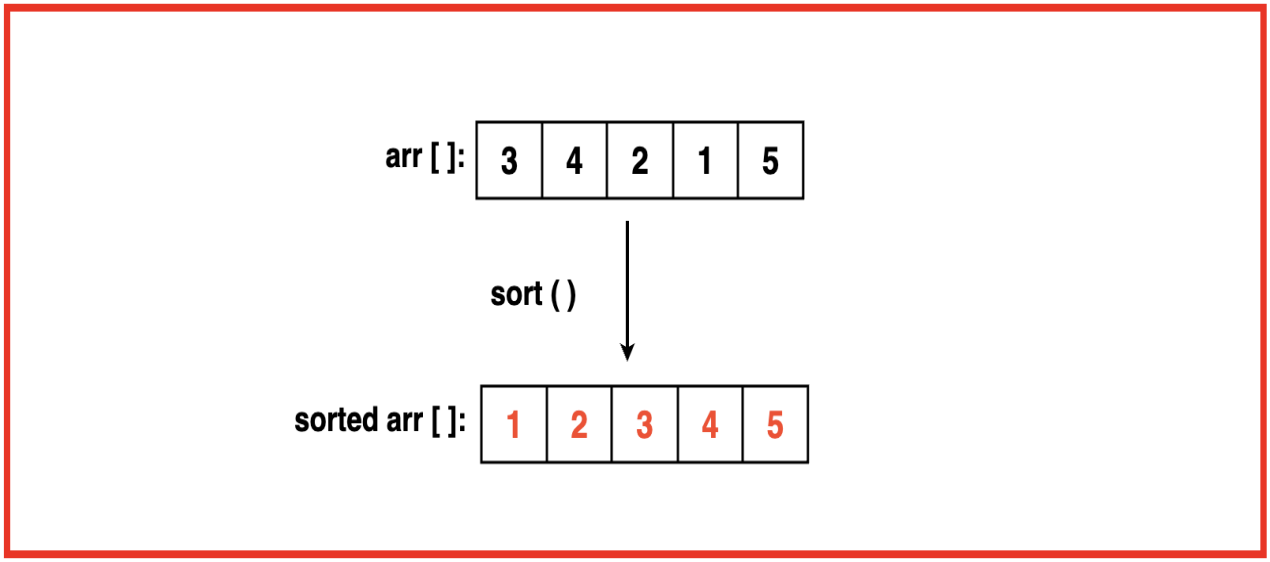
A naive solution could be converting the linked list into an array, sorting the array, and then creating a new linked list from the sorted array's values.

Algorithm

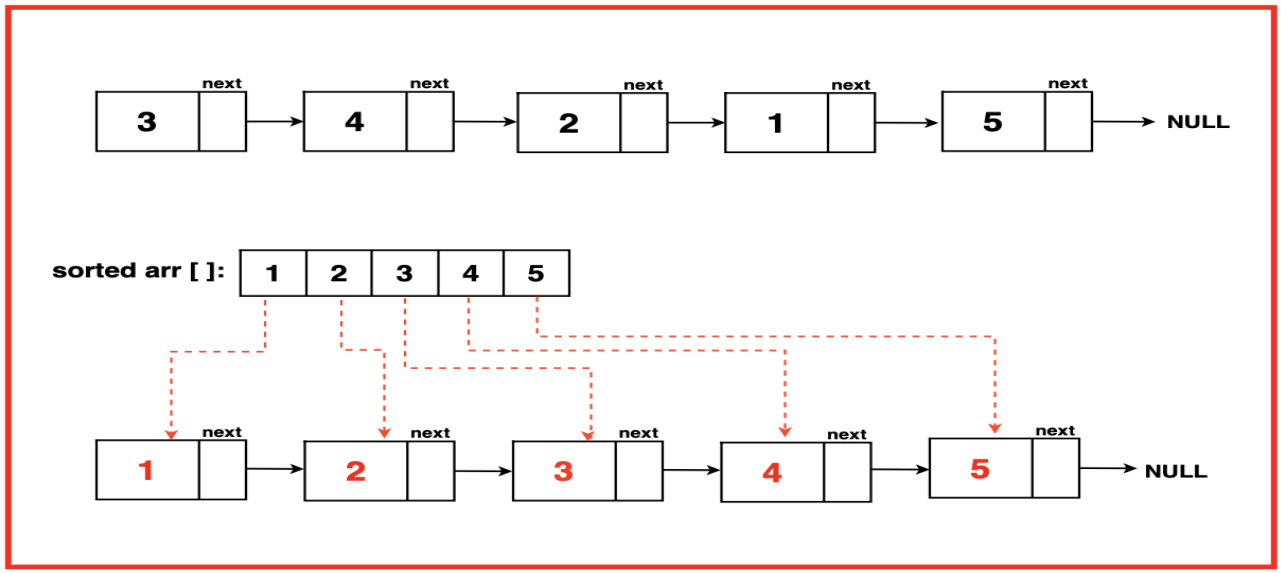
Step 1: Create an empty array to store the node values. Iterate the linked list using a temp pointer to the head and push the value of temp node into the array. Move temp to the next node.



Step 2: Sort the array containing node values in ascending order.



Step 3: Convert the sorted array back to a linked list reassigning the values from the sorted array and overwriting them sequentially according to their order in the array.



Code

```
import java.util.*;

// 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;
    }
}

// Class to perform operations on Linked List
public class Main {

    // Function to sort a linked list
    // using Brute Force approach
    public static Node sortLL(Node head){
        // Create a list to
        // store node values
        List<Integer> arr = new ArrayList<>();

        // Temporary pointer to
        // traverse the linked list
        Node temp = head;

        // Traverse the linked list and
        // store node values in the list
        while(temp != null){
            arr.add(temp.data);
            temp = temp.next;
        }

        // Sort the list
        // containing node values
        Collections.sort(arr);

        // Reassign sorted values to
        // the linked list nodes
        temp = head;
        for(int i = 0; i < arr.size(); i++){
            // Update the node's data
            // with the sorted values
            temp.data = arr.get(i);
            // Move to the next node
            temp = temp.next;
        }

        // Return the head of the
        // sorted linked list
        return head;
    }

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

    public static void main(String[] args) {
        // Linked List: 3 2 5 4 1
        Node head = new Node(3);
        head.next = new Node(2);
```

```
        head.next.next = new Node(5);
        head.next.next.next = new Node(4);
        head.next.next.next.next = new Node(1);

        System.out.print("Original Linked List: ");
        printLinkedList(head);

        // Sort the linked list
        head = sortLL(head);

        System.out.print("Sorted Linked List: ");
        printLinkedList(head);
    }
}
```

Output: Original Linked List: 3 2 5 4 1

Sorted Linked List: 1 2 3 4 5

Complexity Analysis

Time Complexity: O(N) + O(N log N) + O(N)where N is the number of nodes in the linked list.

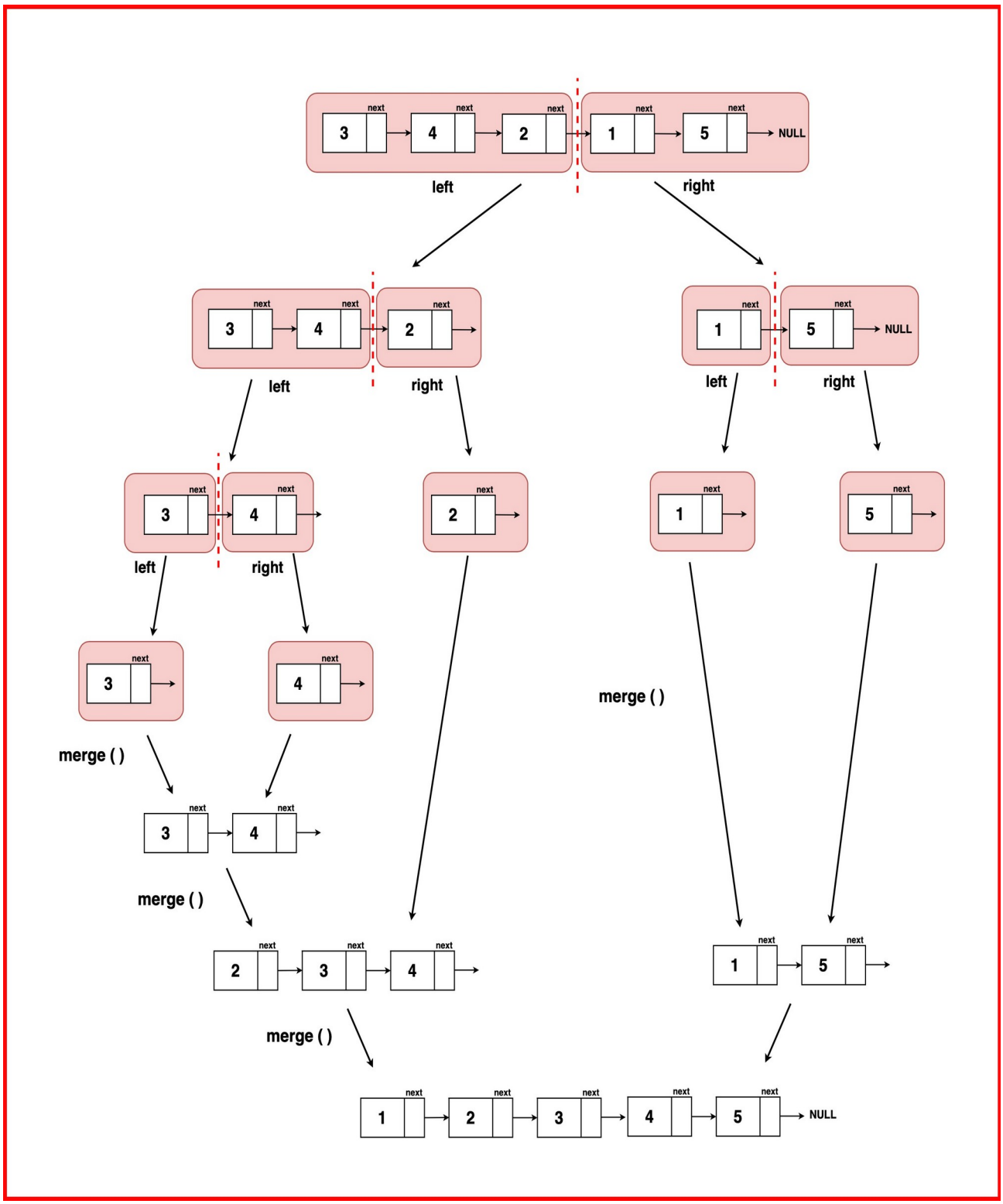
1. O(N) to traverse the linked list and store its data values in an additional array.
2. O(N log N) to sort the array containing the node values.
3. O(N) to traverse the sorted array and convert it into a new linked list.

Space Complexity : O(N)where N is the number of nodes in the linked list as we have to store the values of all nodes in the linked list in an additional array to sort them.

Optimal Approach
Algorithm / Intuition

Instead of using an external array to store node values, we can employ a sorting algorithm without using any extra space. An in-place sorting algorithm like Merge Sort or Quick Sort adapted for linked lists can achieve this.

A modified version of merge sort can operate directly on the linked list without using any additional space. This algorithm would divide the linked list into halves recursively until single nodes remain. These sorted halves of the linked list are merged back together in a sorted order.



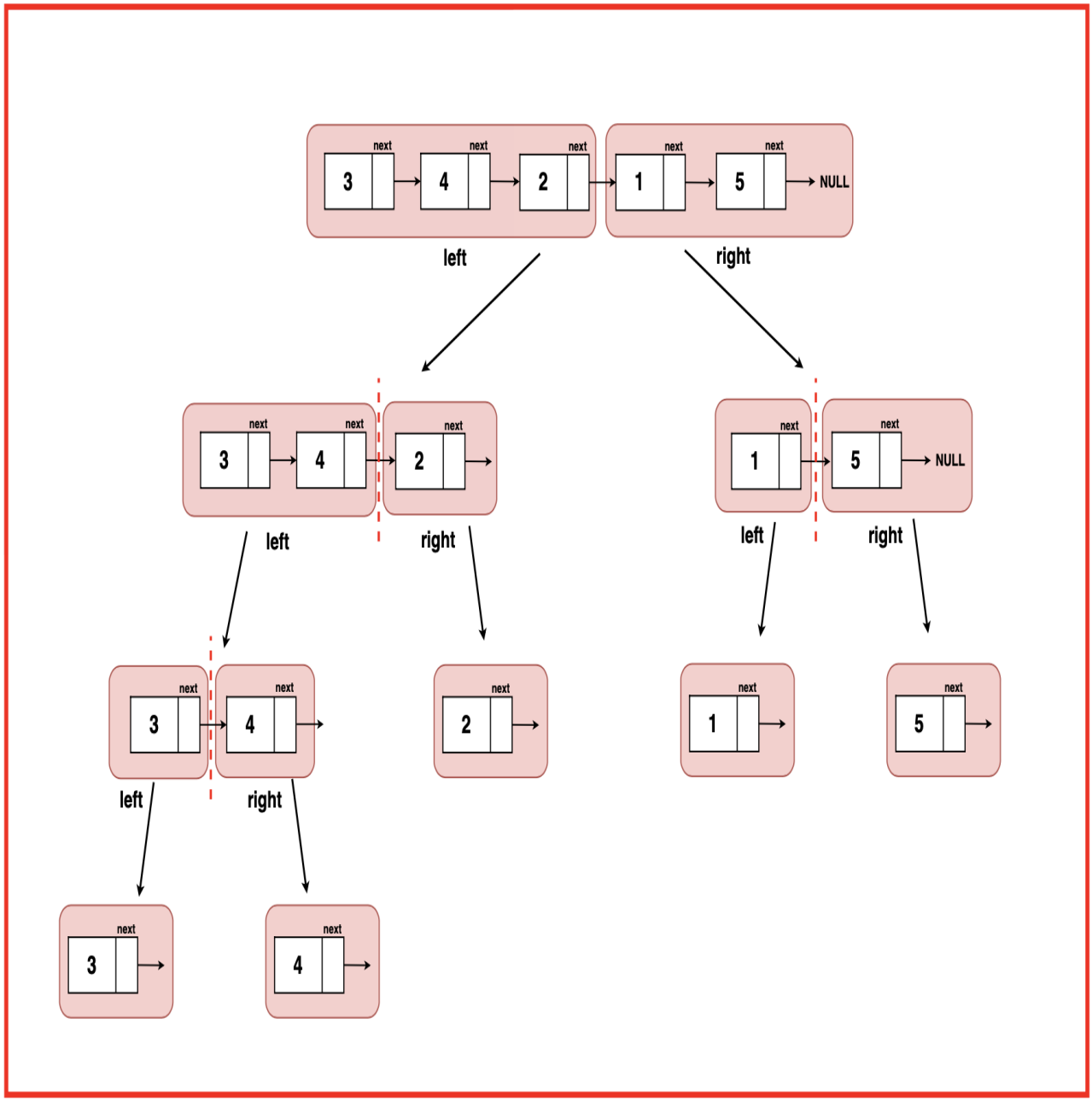
This approach employs the divide-and-conquer strategy:

1. Divides the linked list into smaller parts until they become trivial to sort (single node or empty list).
2. Merges and sorts the divided parts while combining them back together.

Algorithm

Step 1: Base Case If the linked list contains zero or one element, it is already sorted. Return the head node.

Step 2: Split the List Find the middle of the linked list using a slow and a fast pointer. Split the linked list into two halves at the middle node. The two halves will be left and right.



Step 3: Recursion Recursively apply merge sort to both halves obtained in the previous step. This step continues dividing the linked list until there's only one node in each half.

Step 4: Merge Sorted Lists Merge the sorted halves obtained from the recursive calls into a single sorted linked list. Compare the nodes from both halves and rearrange them to form a single sorted list. Update the head pointer to the beginning of the newly sorted list.

Step 5: Return Once the merging is complete, return the head of the sorted linked list.

Code

```
// 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;
    }
}

// Function to merge two sorted linked lists
Node mergeTwoSortedLinkedLists(Node list1, Node list2) {
    // Create a dummy node to serve
    // as the head of the merged list
    Node dummyNode = new Node(-1);
    Node temp = dummyNode;

    // Traverse both lists simultaneously
    while (list1 != null && list2 != null) {
        // Compare elements of both lists and
        // link the smaller node to the merged list
        if (list1.data <= list2.data) {
            temp.next = list1;
            list1 = list1.next;
        } else {
            temp.next = list2;
            list2 = list2.next;
        }
        // Move the temporary pointer
        // to the next node
        temp = temp.next;
    }

    // If any list still has remaining
    // elements, append them to the merged list
    if (list1 != null) {
        temp.next = list1;
    } else {
        temp.next = list2;
    }
    // Return the merged list starting
    // from the next of the dummy node
    return dummyNode.next;
}

// Function to find the middle of a linked list
Node findMiddle(Node head){
```

```
    // If the list is empty or has only one node
    // the middle is the head itself
    if (head == null || head.next == null) {
        return head;
    }

    // Initializing slow and fast pointers
    Node slow = head;
    Node fast = head.next;

    // Move the fast pointer twice
    // as fast as the slow pointer
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // When the fast pointer reaches the end,
    // the slow pointer will be at the middle
    return slow;
}

// Function to perform merge sort on a linked list
Node sortLL(Node head){
    // Base case: if the list is empty or
    // has only one node, it is already
    // sorted, so return the head
    if (head == null || head.next == null) {
        return head;
    }

    // Find the middle of the list
    // using the findMiddle function
    Node middle = findMiddle(head);

    // Divide the list into two halves
    Node right = middle.next;
    middle.next = null;
    Node left = head;

    // Recursively sort the left and right halves
    left = sortLL(left);
    right = sortLL(right);

    // Merge the sorted halves using the
    // mergeTwoSortedLinkedLists function
    return mergeTwoSortedLinkedLists(left, right);
}

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

public static void main(String[] args) {
    // Linked List: 3 2 5 4 1
    Node head = new Node(3);
```

```
    head.next = new Node(2);
    head.next.next = new Node(5);
    head.next.next.next = new Node(4);
    head.next.next.next.next = new Node(1);

    System.out.print("Original Linked List: ");
    printLinkedList(head);

    // Sort the linked list
    head = sortLL(head);

    System.out.print("Sorted Linked List: ");
    printLinkedList(head);
}
```

Output: Original Linked List: 3 2 5 4 1

Sorted Linked List: 1 2 3 4 5

Complexity Analysis

Time Complexity: O(N log N)where N is the number of nodes in the linked list. Finding the middle node of the linked list requires traversing it linearly taking O(N) time complexity and to reach the individual nodes of the list, it has to be split log N times (continuously halve the list until we have individual elements).

Space Complexity : O(1) as no additional data structures or space is allocated for storage during the merging process. However, space proportional to O(log N) stack space is required for the recursive calls. THe maximum recursion depth of log N height is occupied on the call stack.