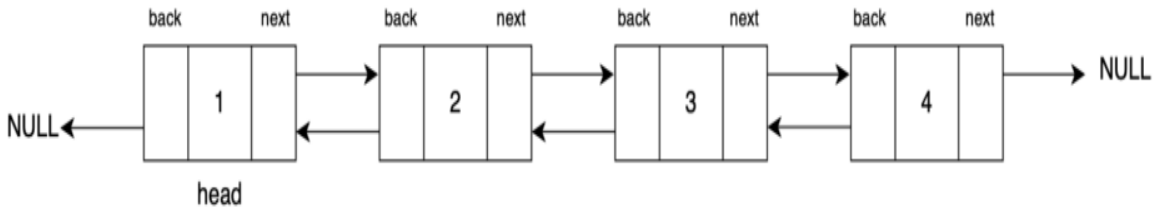
Reverse a Doubly Linked List

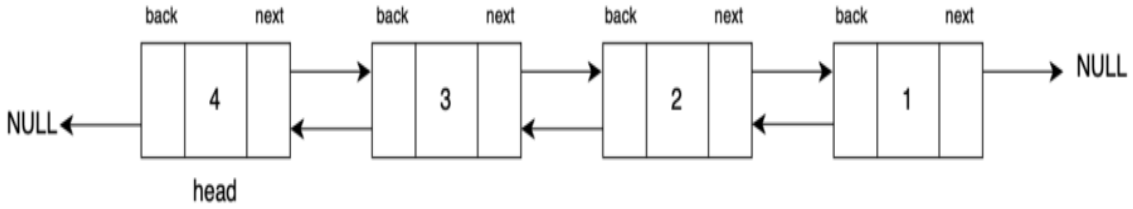
Problem Statement: Given a doubly linked list of size **'N'** consisting of positive integers, your task is to **reverse** it and return the head of the modified doubly linked list.

Examples

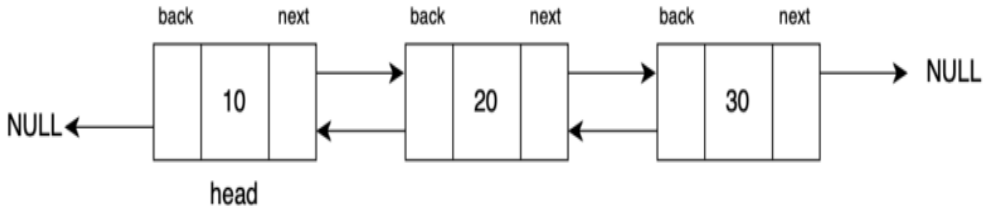
Example 1:
Input Format:
DLL: 1 <-> 2 <-> 3 <-> 4



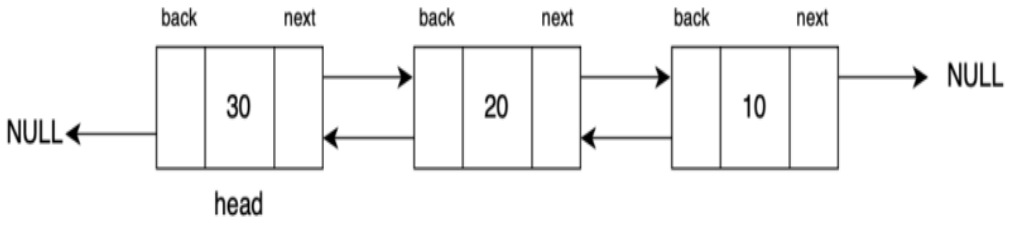
Result: DLL: 4 <-> 3 <-> 2 <-> 1



Explanation: The doubly linked list is reversed and its last node is returned at the new head pointer.
Example 2:
Input Format:
DLL: 10 <-> 20 <-> 30



Result: DLL: 30 <-> 20 <-> 10



Explanation: In this case, the doubly linked list is reversed and its former tail is returned as its new head.

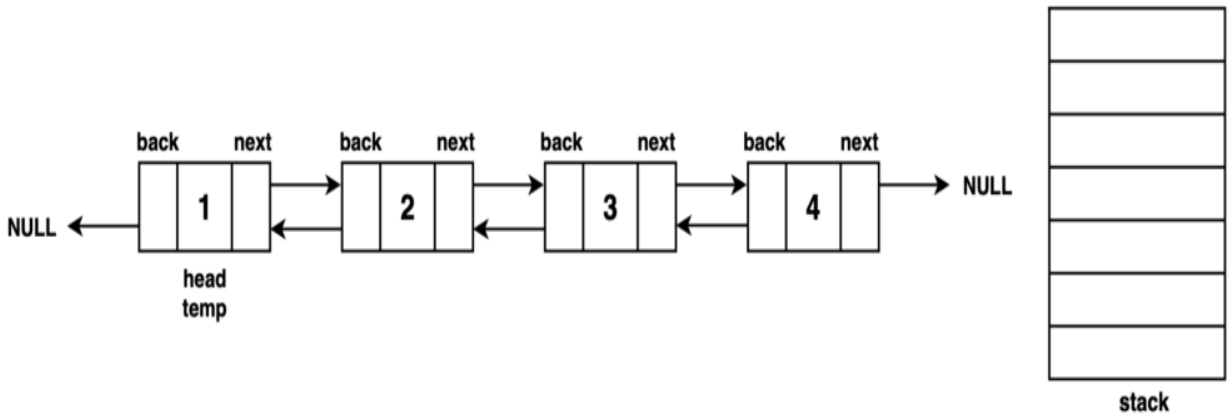
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Brute Force Approach
Algorithm / Intuition

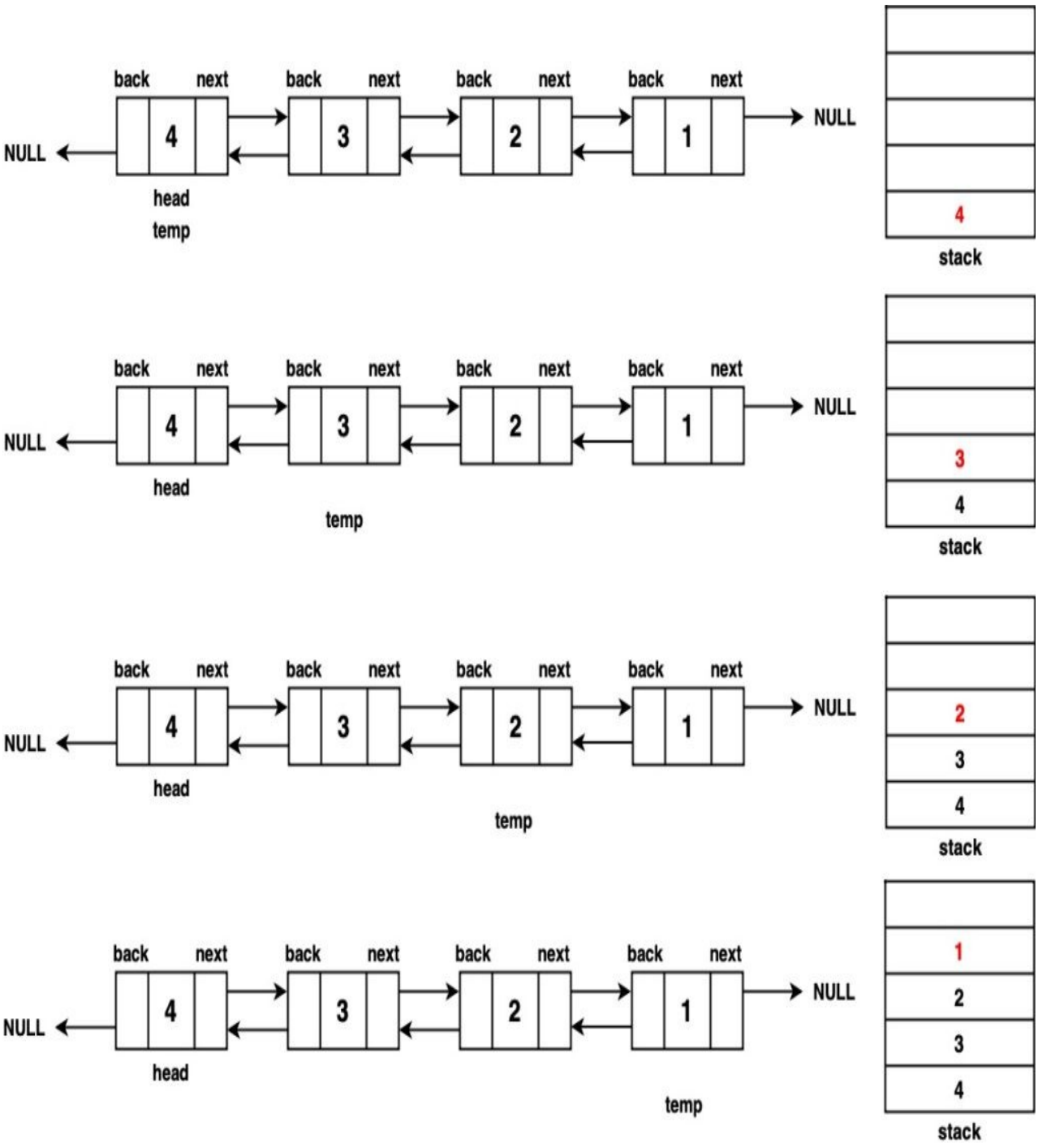
A brute-force approach involves replacing data in a doubly linked list. First, we traverse the list and store node data in a stack. Then, in a second pass, we assign elements from the stack to nodes, ensuring a reverse order replacement since stacks follow the **Last-In-First-Out (LIFO)** principle.

Algorithm:

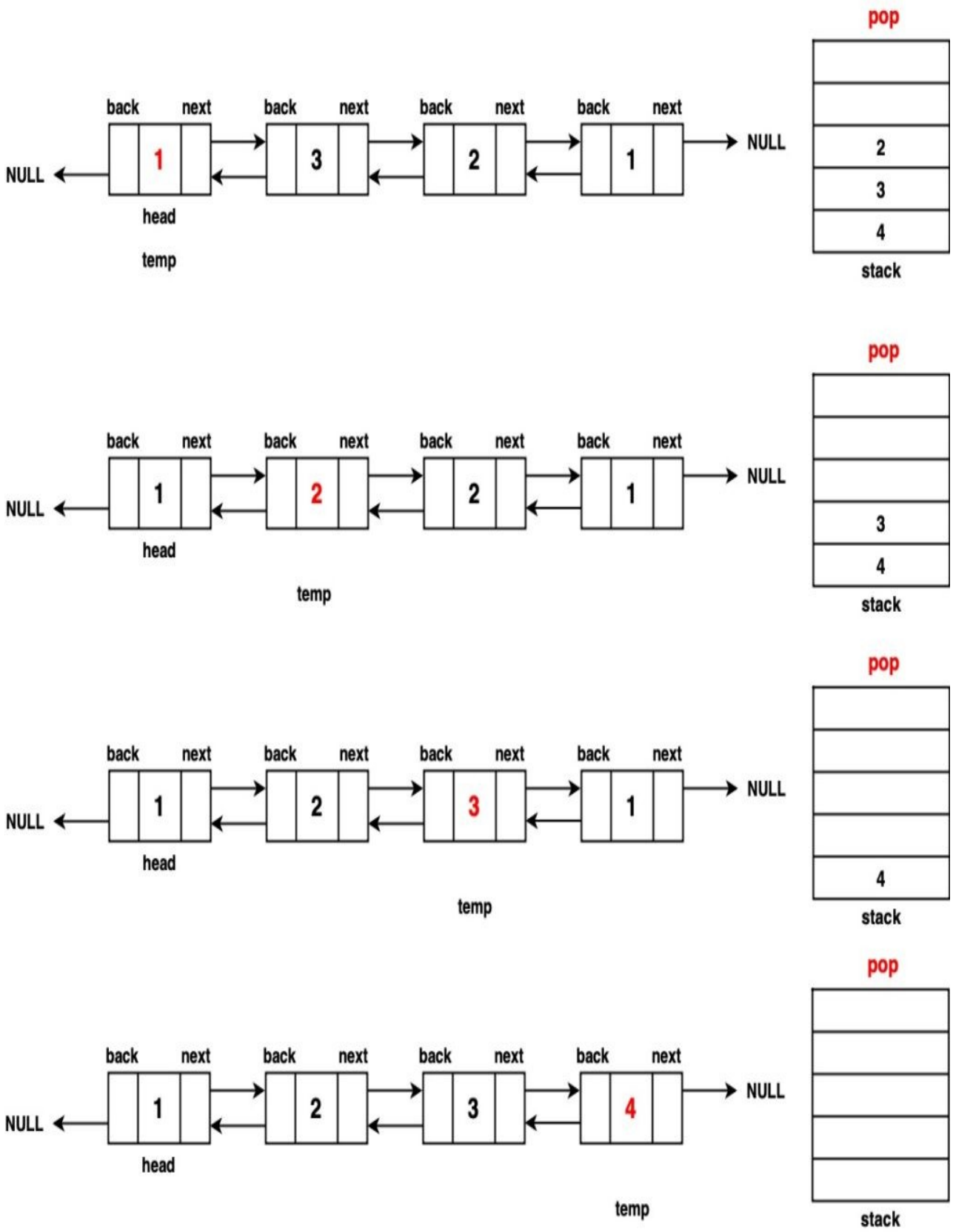
Step 1: Initialization a **temp** pointer to the **head** of the doubly linked list and a **stack** data structure to store the values from the list.



Step 2: Traverse the doubly linked list with the **temp** pointer and while traversing push the value at the **current node temp** onto the stack. Move the **temp** to the **next node** continuing **until temp reaches null** indicating the end of the list.



Step 3: Reset the **temp** pointer back to the **head** of the list and in thissecond iteration **pop the element** from the stack, replace the data at the **current node** with the popped value from the **top** of the stack and move temp to the next node. Repeat this step until temp reaches **null** or the **stack** becomes **empty.**



Code

```
import java.util.Stack;

public class DLinkedList {
    public static class Node {
        // Data stored in the node
```

```
    public int data;
    // Reference to the next node
    //in the list (forward direction)
    public Node next;
    // Reference to the previous nod
    //in the list (backward direction)
    public Node back;

    // Constructor for a Node with both data,
    //a reference to the next node, and a
    //reference to the previous node
    public Node(int data1, Node next1, Node back1) {
        data = data1;
        next = next1;
        back = back1;
    }

    // Constructor for a Node with data,
    //and no references to the next and
    //previous nodes (end of the list)
    public Node(int data1) {
        data = data1;
        next = null;
        back = null;
    }
}

private static Node convertArr2DLL(int[] arr) {
    // Create the head node with the
    //first element of the array
    Node head = new Node(arr[0]);
    // Initialize 'prev' to the head node
    Node prev = head;

    for (int i = 1; i < arr.length; i++) {
        // Create a new node with data from
        // the array and set its 'back' pointer
        // to the previous node

        Node temp = new Node(arr[i], null, prev);
        // Update the 'next' pointer of the
        // previous node to point to the new node

        prev.next = temp;
        // Move 'prev' to the newly created node
        //for the next iteration
        prev = temp;
    }
    // Return the head of the doubly linked list
    return head;
}

private static void print(Node head) {
    while (head != null) {
        // Print the data in the current node
        System.out.print(head.data + " ");
        // Move to the next node
        head = head.next;
    }
    System.out.println();
}

    // Initialise a stack st
Stack<Integer> st = new Stack<>();
```

```
    // Initialise the node pointer temp at head
    Node temp = head;

    // Traverse the doubly linked list via the temp pointer
    while(temp!=null){
        // insert the data of the current node into the stack
        st.push(temp.data);
        // traverse further
        temp = temp.next;
    }

    // Reinitialise temp to head
    temp = head;

    // Second iteration of the DLL to replace the values
    while(temp!=null){
        // Replace the value pointed via temp with
        // the value from the top of the stack and pop it
        temp.data = st.pop();

        // Traverse further
        temp = temp.next;
    }

    // Return the updated doubly linked
    // where the values of nodes from both ends
    // has been swapped
    return head;

    }



    public static void main(String[] args) {
        int[] arr = {12, 5, 6, 8, 4};
        // Convert the array to a doubly linked list
        Node head = convertArr2DLL(arr);

        // Print the doubly linked list
        System.out.println(“Doubly Linked List Initially:  “);
        print(head);

        System.out.println("Doubly Linked List After Reversing :");

        head = reverseDLL(head);
        print(head);

    }
}
```

Output: Doubly Linked List Initially: 12 5 6 8 4 Doubly Linked List After Reversing : 4 8 6 5 12

Complexity Analysis

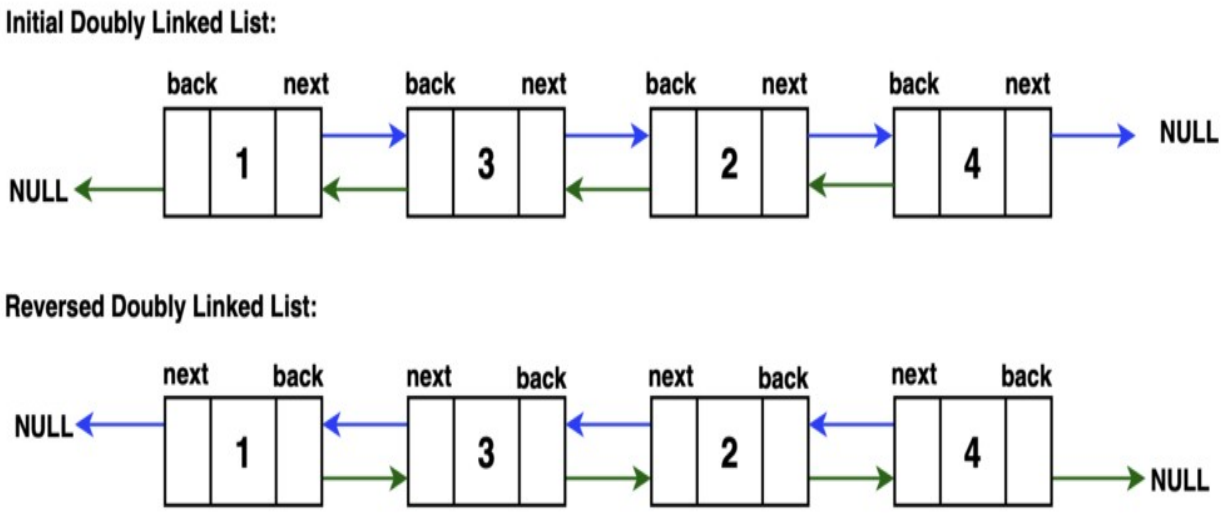
Time Complexity : O(2N) During the first traversal, each node's value is pushed into the stack once, which requires O(N) time. Then, during the second iteration, the values are popped from the stack and used to update the nodes. Space Complexity : O(N) This is because we are using an external stack data structure. At the end of the first iteration, the stack will hold all N values of the doubly

linked list therefore the space required for stack is directly proportional to the size of the input doubly linked list.

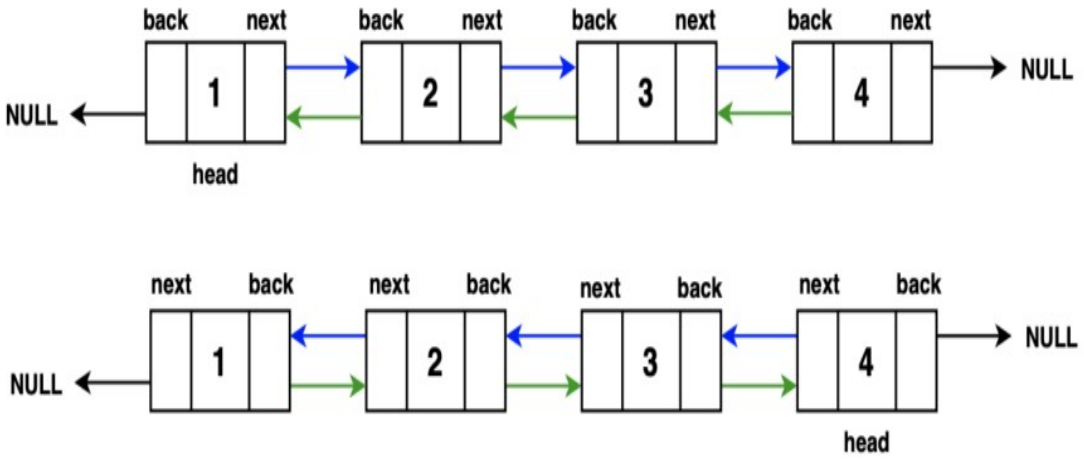
Optimal Approach
Algorithm / Intuition

Reverse the Links in a Single Traversal

Instead of performing two separate traversals of the linked list and storing its node values in an external data structure, we can **optimize** our approach by directly **modifying the links between the nodes** within the doubly linked list **in place**, as visualized below:

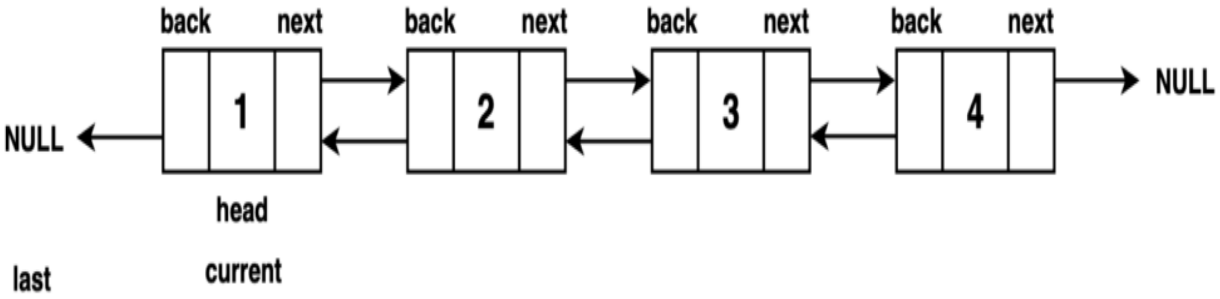


We need to traverse on every node, and for every node change the next pointer and back pointer. If we can do this for all nodes, at the end of traversal, the doubly linked list will be reversed.



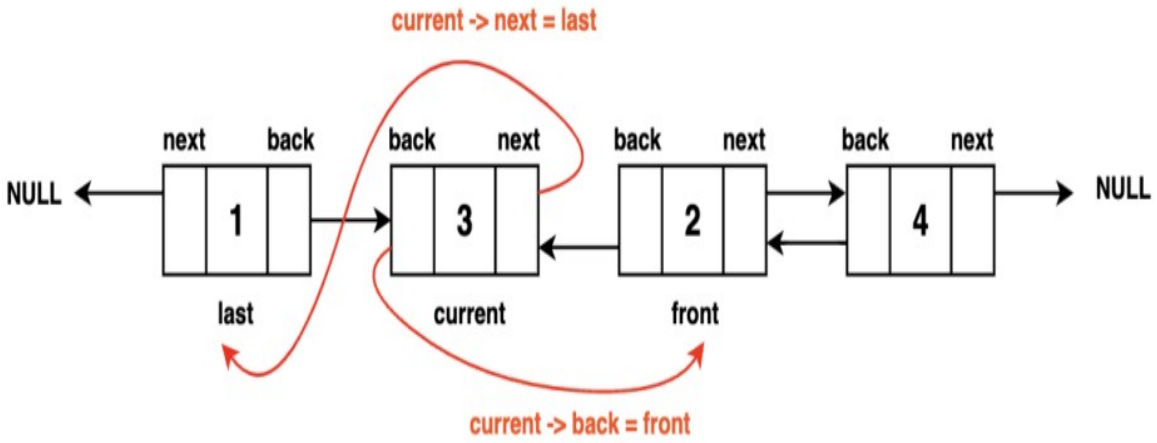
Algorithm:

Step 1: Initialise **two pointers** that are needed for the reversal. Initialize a **current** pointer to the **head** of the linked list. This pointer will **traverse** the list as we **reverse** it. Initialize a second pointer **last** to null. This pointer will be used for **temporary storage** during **pointer swapping,** as we need a third variable while swapping two data**.**

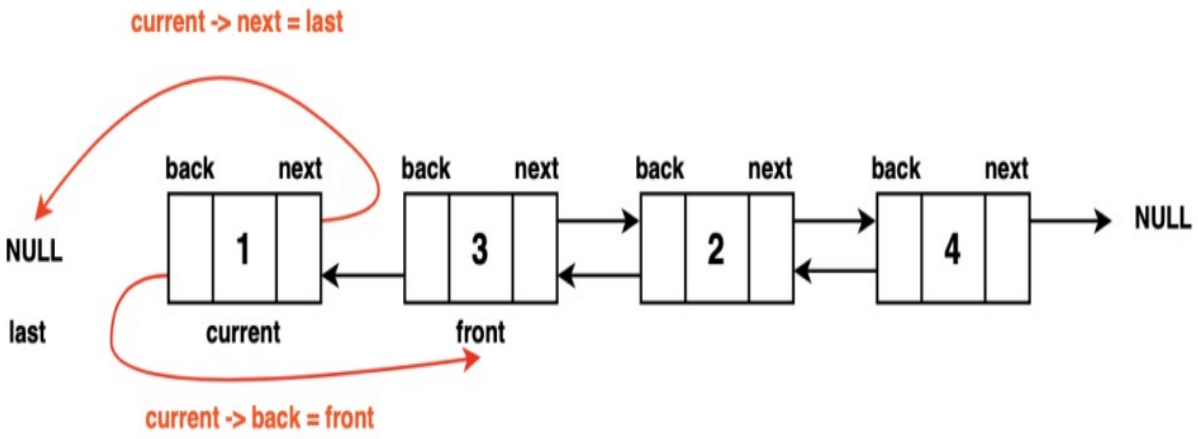


Step 2: Traverse through the DLL by looping over all the nodes..

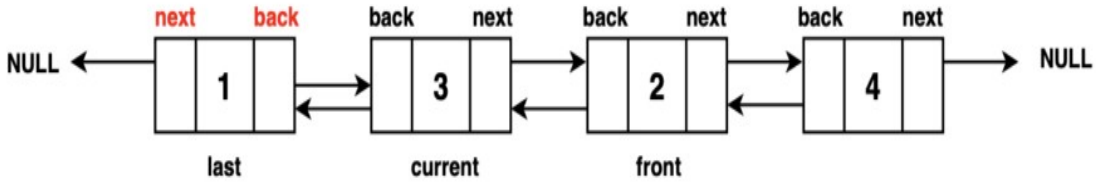
Step 3: While iterating over all nodes in the linked list, we make changes to set the **backward** pointer of a node to the **next** changing its **previous link**. Along with this, the **forward** pointer is adjusted to point to the **previous** node, reversing the **next link**. To prevent losing the last node in this process, we use a **reference** to the **last node** to retain it.



- Update the **current node's back** pointer to point to the **next node** (c**urrent->back = current->next**). This step reverses the direction of the **backward** pointer.



- Update the **current node's next** pointer to point to the **previous node** (current->next = last). This step r**everses the direction of the forward pointer.**

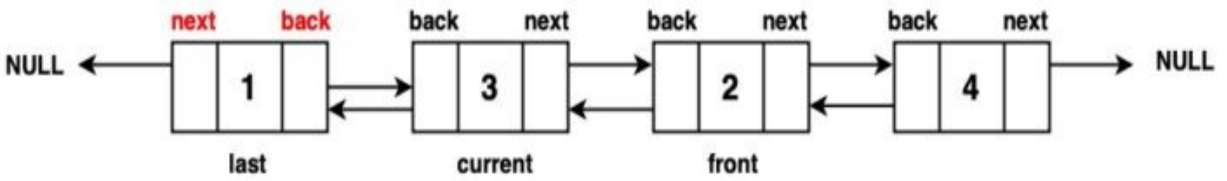


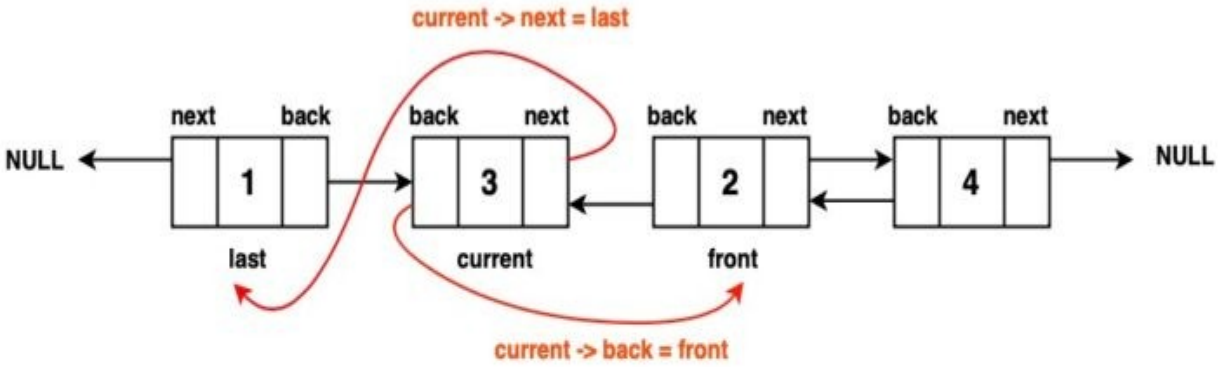
- Move the current pointer **one step forward** (current = current->back). This allows us to continue the reversal process.

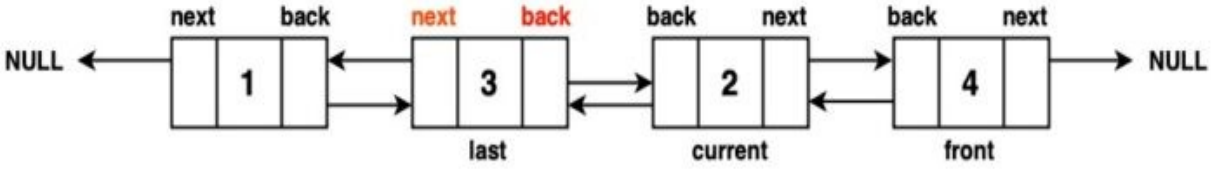
Step 4: After completing the traversal, the **last** node ends up at the **second node** in the reversed doubly linked list. To obtain the **new head** of the reversed list, we simply use the **backward pointer** of the **last node**, which points to the **new head**.

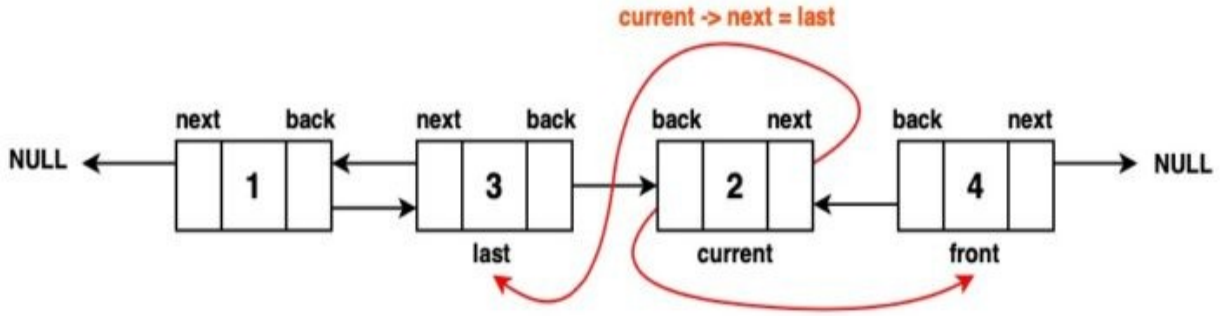
To ensure that we handle the case where the traversal ended at the original list's end (i.e., the last pointer is not null), we **update** the **head pointer** to point to the **new head** of the reversed list, which is stored in the **last pointer**.

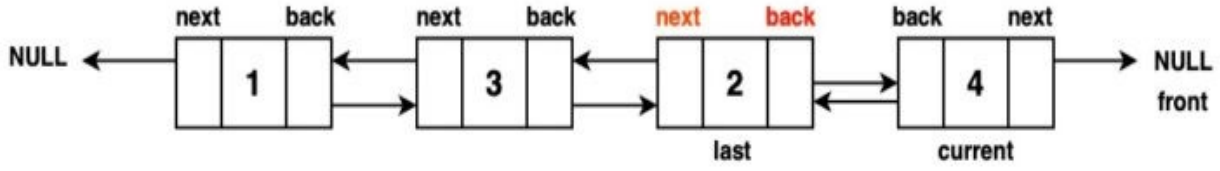
Finally, we return the **head pointer**, now pointing to the head of the fully reversed doubly linked list.

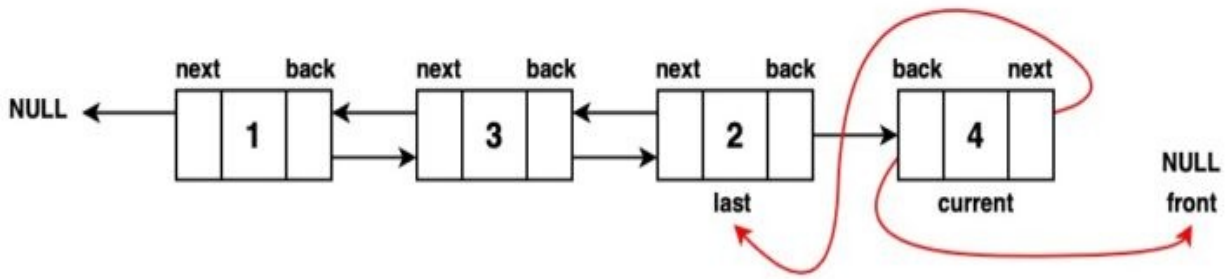

next
back
back
next
back
next
back
next
NULL
1
3
2
4
NULL
last
current
front

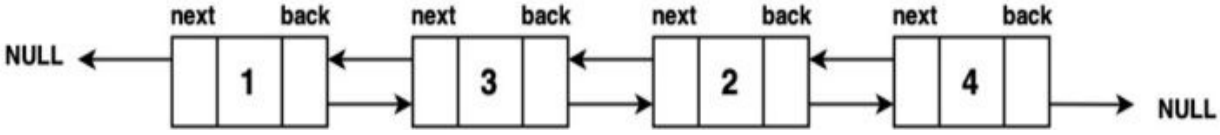

current -> next = last
next
back
back
next
back
next
back
next
NULL
1
3
2
4
NULL
last
current
front
current -> back = front


next
back
next
back
back
next
back
next
NULL
1
3
2
4
NULL
last
current
front


current -> next = last
next
back
next
back
back
next
back
next
NULL
1
3
2
4
NULL
last
current
front


next
back
next
back
next
back
back
next
NULL
1
3
2
4
NULL
front
last
current


next
back
next
back
next
back
back
next
NULL
1
3
2
4
NULL
front
last
current


next
back
next
back
next
back
next
back
NULL
1
3
2
4
NULL

Code

```
import java.util.Stack;

public class DLinkedList {
    public static class Node {
        // Data stored in the node
        public int data;
        // Reference to the next node
        //in the list (forward direction)
        public Node next;
        // Reference to the previous nod
        //in the list (backward direction)
        public Node back;

        // Constructor for a Node with both data,
        //a reference to the next node, and a
        //reference to the previous node
        public Node(int data1, Node next1, Node back1) {
            data = data1;
            next = next1;
            back = back1;
        }

        // Constructor for a Node with data,
        //and no references to the next and
        //previous nodes (end of the list)
        public Node(int data1) {
            data = data1;
            next = null;
            back = null;
        }
    }

    private static Node convertArr2DLL(int[] arr) {
        // Create the head node with the
        //first element of the array
        Node head = new Node(arr[0]);
        // Initialize 'prev' to the head node
        Node prev = head;

        for (int i = 1; i < arr.length; i++) {
            // Create a new node with data from
            // the array and set its 'back' pointer
            // to the previous node

            Node temp = new Node(arr[i], null, prev);
            // Update the 'next' pointer of the
            // previous node to point to the new node

            prev.next = temp;
            // Move 'prev' to the newly created node
            //for the next iteration
            prev = temp;
        }
        // Return the head of the doubly linked list
        return head;
    }

    private static void print(Node head) {
        while (head != null) {
            // Print the data in the current node
            System.out.print(head.data + " ");
```

```
        // Move to the next node
        head = head.next;
    }
    System.out.println();
}

private static Node reverseDLL(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;
    }

     // Initialize a pointer to
     // the previous node
    Node prev = null;

    // Initialize a pointer to
    // the current node
    Node current = head;

    // Traverse the linked list
    while (current != null) {

        // Store a reference to
        // the previous node
        prev = current.back;

        // Swap the previous and
        // next pointers
        current.back = current.next;

        // This step reverses the links
        current.next = prev;

        // Move to the next node
        // in the orignal list

        current = current.back;
    }

    // The final node in the original list
    // becomes the new head after reversal
    return prev.back;
}



public static void main(String[] args) {
    int[] arr = {12, 5, 6, 8, 4};
    // Convert the array to a doubly linked list
    Node head = convertArr2DLL(arr);

    // Print the doubly linked list
    System.out.println(“Doubly Linked List Initially:  “);
    print(head);

    System.out.println("Doubly Linked List After Reversing :");

    head = reverseDLL(head);
    print(head);
```

```
    }
}
```

Output: Doubly Linked List Initially: 12 5 6 8 4 Doubly Linked List After Reversing : 4 8 6 5 12

Complexity Analysis

Time Complexity : O(N) We only have to **traverse** the doubly linked list **once**, hence our time complexity is O(N).

Space Complexity : O(1), as the reversal is done in place.