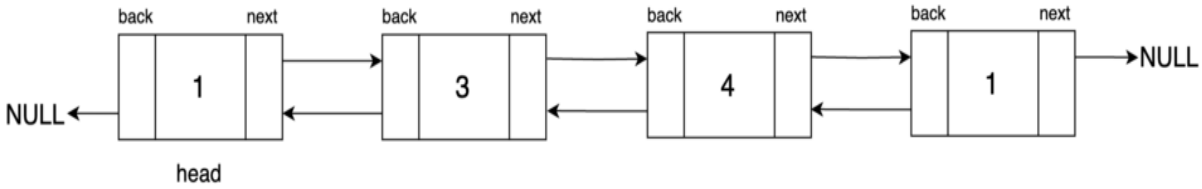
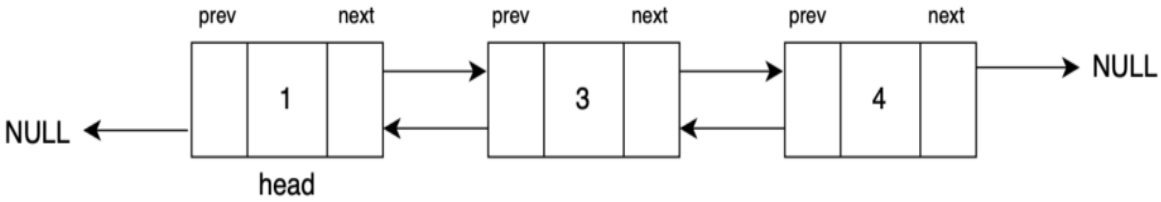
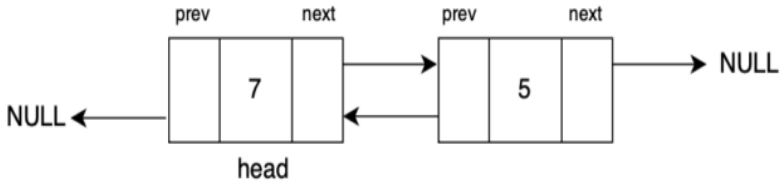
Delete Last Node of a Doubly Linked List

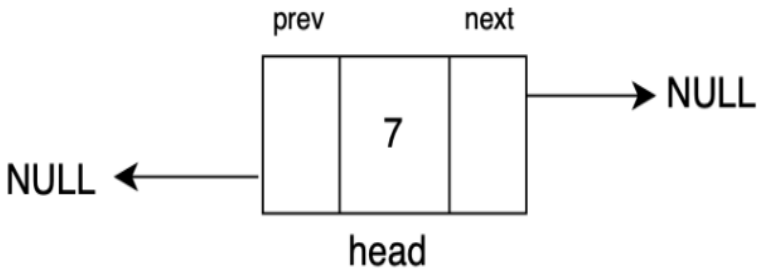
Problem Statement: Given a Doubly Linked List. Delete the last of a Doubly Linked List.

Examples









Solution:

Approach:

To delete the tail of a doubly linked list, we update the linkage between its last node and its second last node. Since a doubly linked list is bidirectional, we set the second last node's next pointer and the last node's back pointer to null. Then, we return the head as the result.

Two edge cases to consider are:

1. If the input doubly linked list is empty, we return null.
2. If there is only one node in the list, that node itself will be the tail and we return null after deleting that node.

Algorithm

Step 1: Traverse the doubly linked list to the last node and keep track of it using the tail pointer.

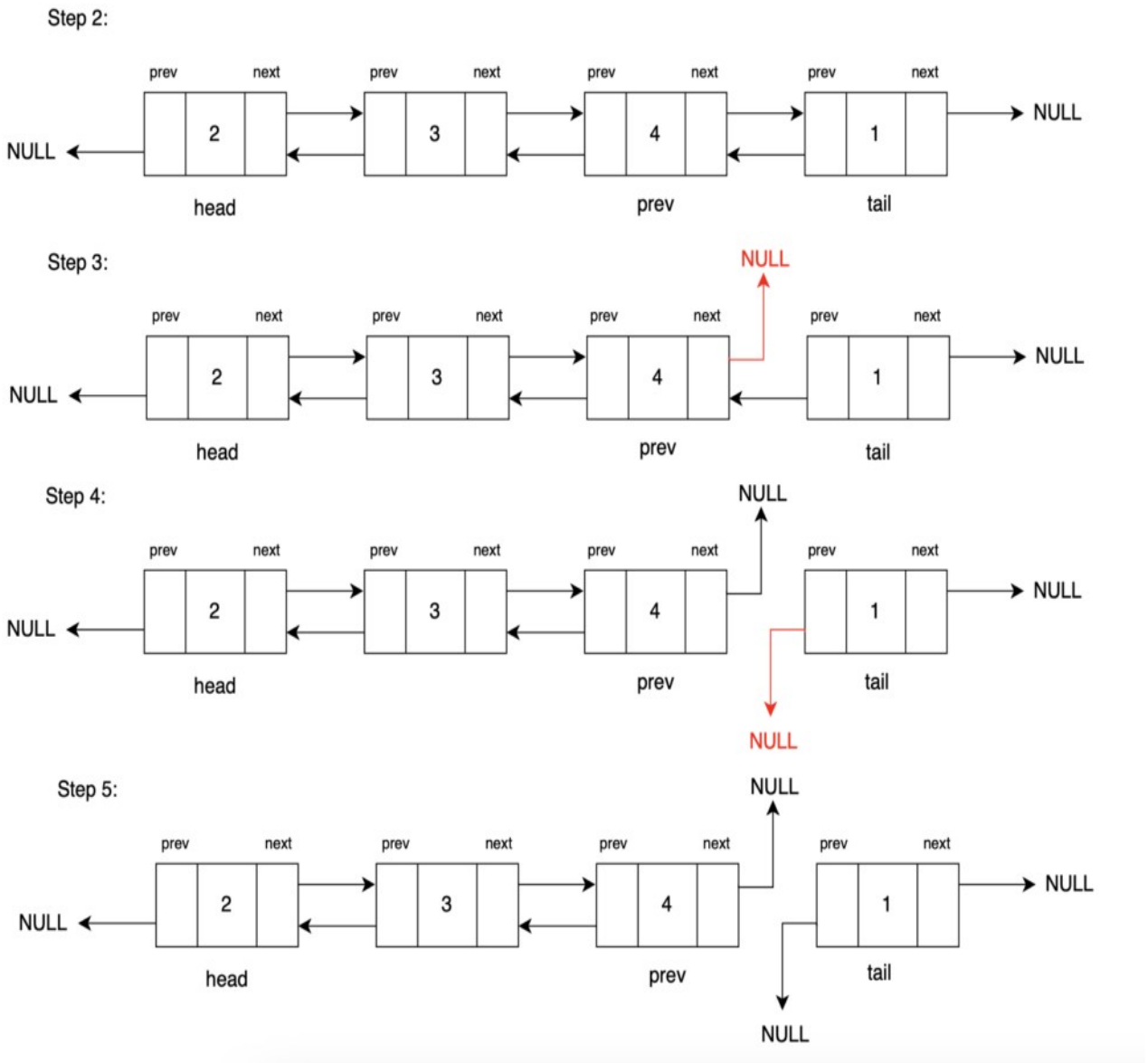
We start from the head of the doubly linked list and iterate through the list using a while loop until we reach the last node. The **tail** pointer is used to keep track of this last node.

Step 2: Obtain the second last node using the tail's back pointer, and name it the **prev** pointer.

Step 3: Set the 'next' pointer of the prev node to null. This step effectively disconnects the initial tail node from the list in the forward direction, making **prev** the new tail node.

Step 4: Set the 'back' pointer of the tail node to null. This ensures that the **tail** node no longer points back to the **prev** node, as it is now the last node in the list.

Step 5: Return the head of the doubly linked list as the result. Now that we have updated the doubly linked list, the list is now one node shorter than before.



Step 6: Delete tail (C++ Only)

Note that in C++, it's essential to explicitly **delete the previous tail** to **free memory**. In Java, **memory management** is automatic, handled by the **garbage collector**, which cleans up unreferenced objects.

Code:

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

        // 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;
        }
    }

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

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

    // Function to delete the tail of the doubly linked list
    private static Node deleteTail(Node head) {
        if (head == null || head.next == null) {
            return null; // Return null if the list is empty or contains only
one element
        }

        Node tail = head;
        while (tail.next != null) {
            tail = tail.next;
        }

        Node newtail = tail.back;

        newtail.next = null;
        tail.back = null;

        return head;
    }

    // Function to delete the head of the doubly linked list
    private static Node deleteHead(Node head) {
        if (head == null || head.next == null) {
            return null; // Return null if the list is empty or contains only
one element
        }

        Node prev = head;
        head = head.next;

        head.back = null; // Set 'back' pointer of the new head to null
        prev.next = null; // Set 'next' pointer of 'prev' to null

        return head;
    }
```

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

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

        System.out.println("Doubly Linked List after deleting tail node: ");
        head = deleteTail(head);
        print(head);
    }
}
```

Output:

12 5 8 7
After deleting tail node:
12 5 8

Time Complexity: O(1) Removing the head of a doubly linked list is a quick operation, taking constant time because it only involves updating references.

Space Complexity: O(1) Deleting the head also has minimal memory usage, using a few extra pointers without regard to the list's size.