Clone Linked List with Random and Next Pointer

Problem Statement: Given a linked list where every node in the linked list contains two pointers:

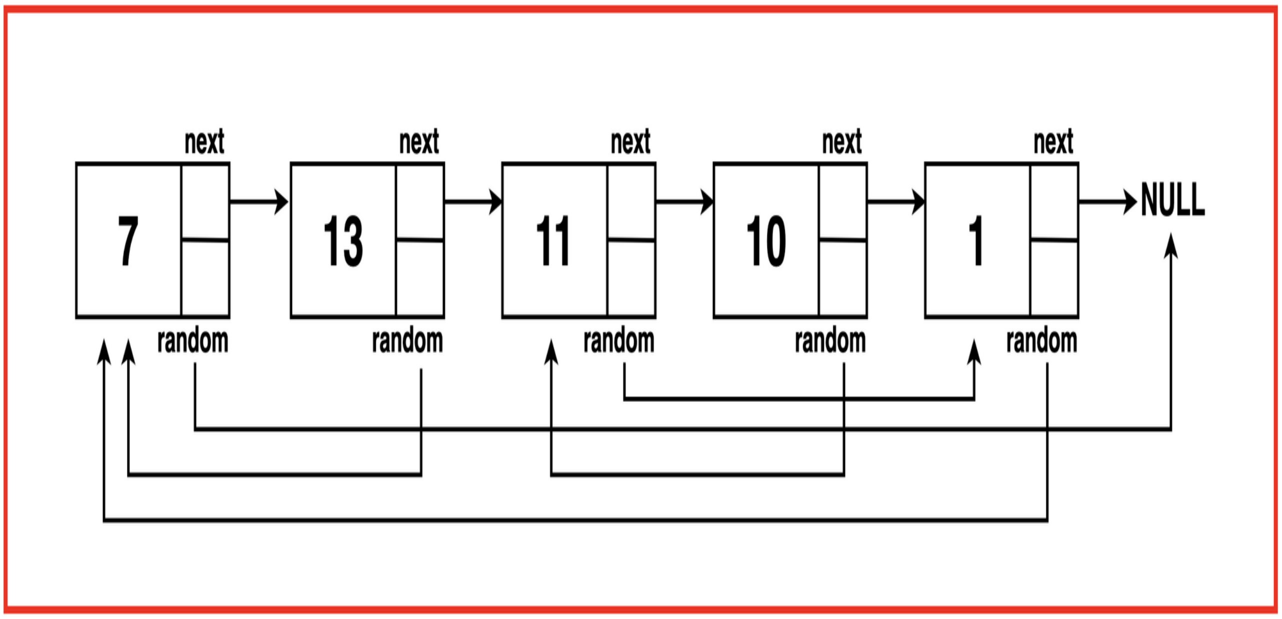
1. ‘next’ which points to the next node in the list.
2. ‘random’ which points to a random node in the list or ‘null’.

Create a ‘deep copy’ of the given linked list and return it.

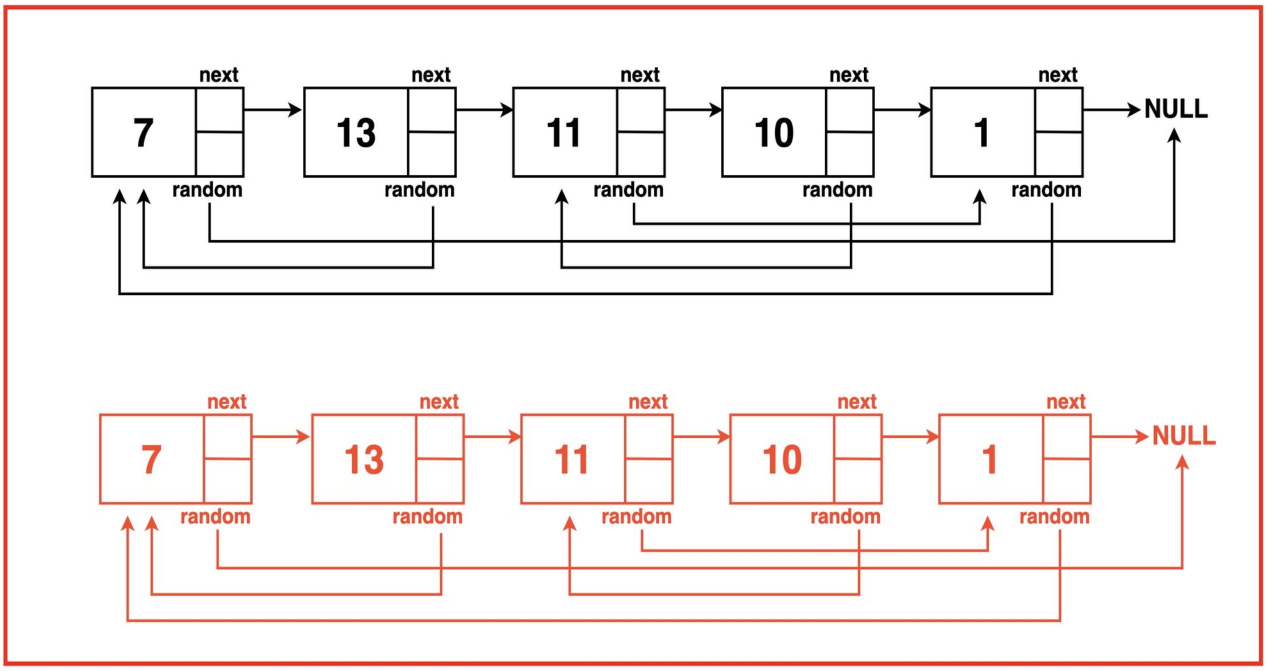
Examples

```
Example 1:

Input:
```

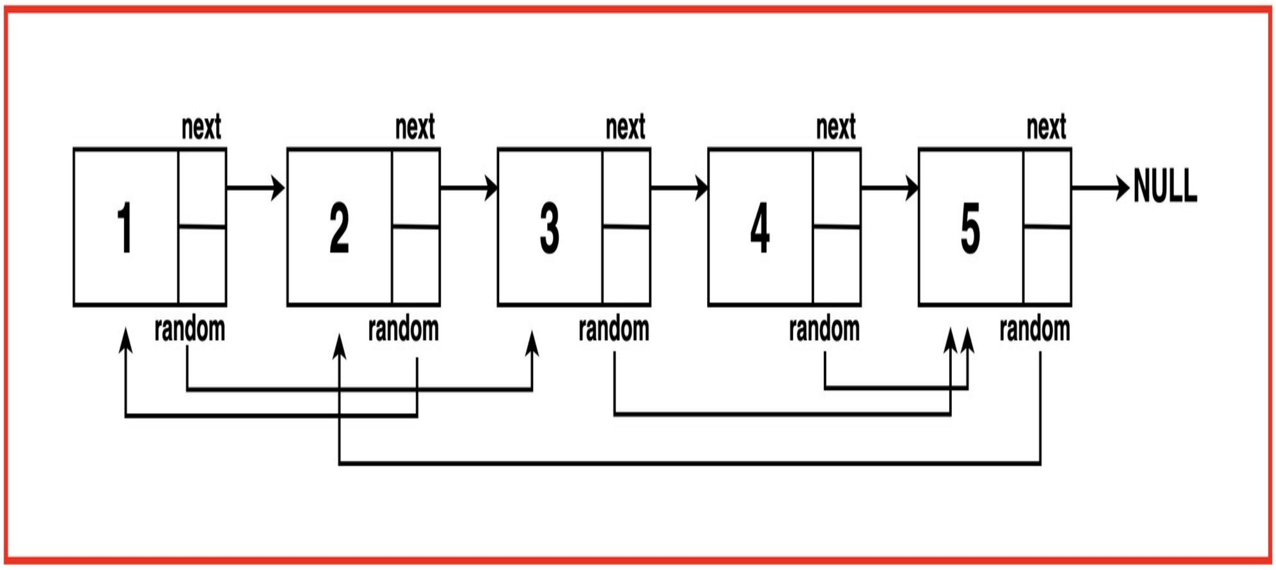


```
Output:
```

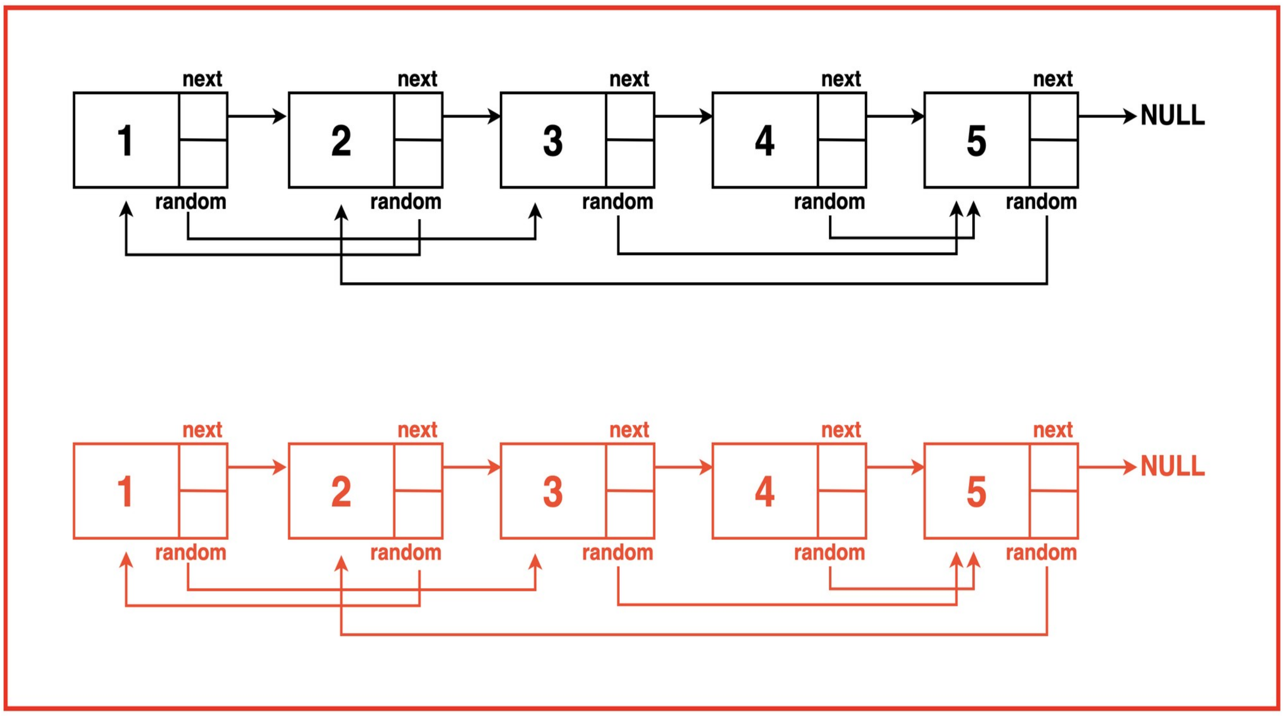


Explanation: A deep copy of the linked list has to be created while maintaining all ‘next’ and ‘random’ pointers to the appropriate new nodes. Additional memory allocation is done while creating a duplicate set of nodes and managing their pointer relationships.

Example 2:
Input:



Output:



Explanation: A deep copy of the linked list has to be created while maintaining all ‘next’ and ‘random’ pointers to the appropriate new nodes. Additional memory allocation is done while creating a duplicate set of nodes and managing their pointer relationships.

Brute Force Approach
Algorithm / Intuition

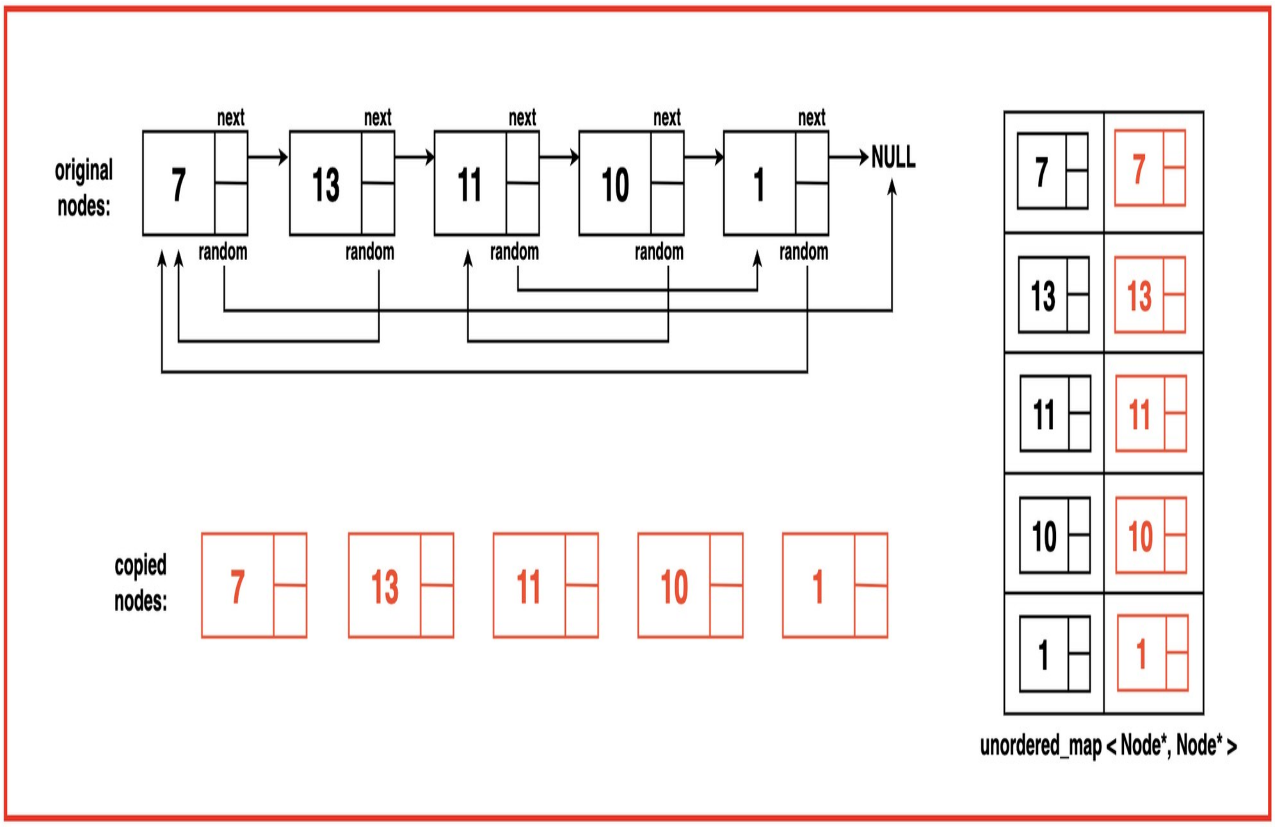
To create a deep copy of the original linked list we can use a map to establish a relationship between original nodes and their copied nodes.

We traverse the list first to create a copied node for each original node then traverse and establish the correct connections between the copied nodes similar to the arrangement of next and random pointers of the original pointers. In the end, return the head of the copied list obtained from the map.

Algorithm

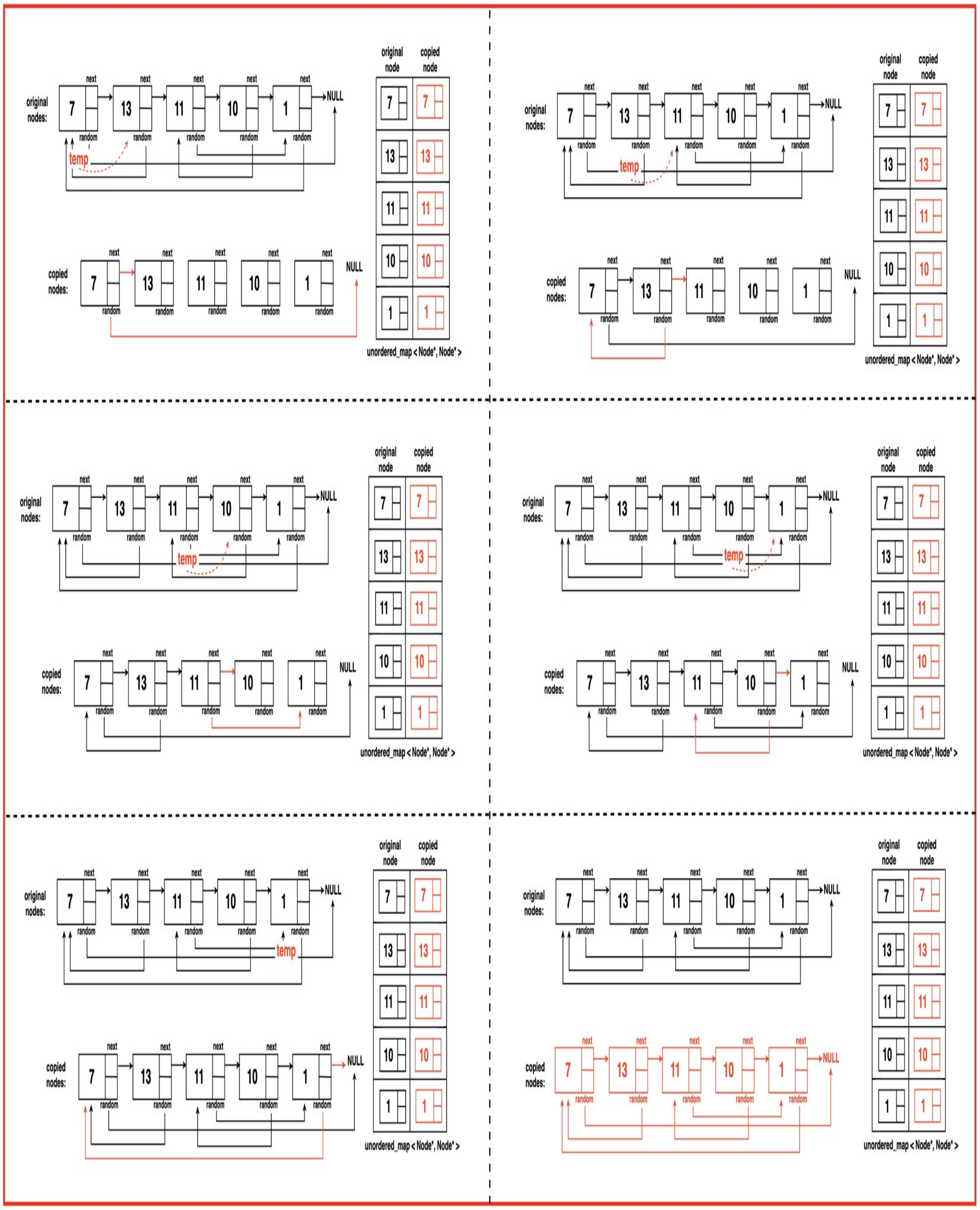
Step 1: Initialise variables 'temp' as a pointer to the head of the original linked list to traverse it. Create an empty unordered_map, to map original nodes to their corresponding copied nodes.

Step 2: Iterate through the original linked list and for each node in the linked list create a new node with the same data value as the original data. Map the original node to its copied node in the map.



Step 3: Iterate through the original list again but this time connect the pointers of the copied nodes in the same arrangement as the original node.

1. Get the copied node corresponding to the original node using the map.
2. Set the next pointer of the copied node to the copied node mapped to the original node's next node.
3. Set the random pointer of the copied node to the original node's next node copied from the map.



Step 4: Return the head of the deep copied list which is obtained by retrieving the copied nodes mapped to the original head from the map.

Code

```
import java.util.HashMap;

// Node class to represent
// elements in the linked list
class Node {
    // Data stored in the node
    int data;
    // Pointer to the next node
    Node next;
    // Pointer to a random node in the list
    Node random;

    // Constructors for Node class
    Node() {
        this.data = 0;
        this.next = null;
        this.random = null;
    }

    Node(int x) {
        this.data = x;
        this.next = null;
        this.random = null;
    }

    Node(int x, Node nextNode, Node randomNode) {
        this.data = x;
        this.next = nextNode;
        this.random = randomNode;
    }
}

public class Main {
    // Function to clone the linked list
    public static Node cloneLL(Node head) {
        Node temp = head;
        // Create a HashMap to map original nodes
        // to their corresponding copied nodes
        HashMap<Node, Node> map = new HashMap<>();

        // Step 1: Create copies of each
        // node and store them in the map
        while (temp != null) {
            // Create a new node with the
            // same data as the original node
            Node newNode = new Node(temp.data);
            // Map the original node to its
            // corresponding copied node in the map
            map.put(temp, newNode);
            // Move to the next node in the original list
            temp = temp.next;
        }

        temp = head;
        // Step 2: Connect the next and random
        // pointers of the copied nodes using the map
        while (temp != null) {
            // Access the copied node corresponding
            // to the current original node
            Node copyNode = map.get(temp);
            // Set the next pointer of the copied node
            // to the copied node mapped to the
            // next node in the original list
            copyNode.next = map.get(temp.next);
            // Set the random pointer of the copied node
            // to the copied node mapped to the
            // random node in the original list
            copyNode.random = map.get(temp.random);
            // Move to the next node in the original list
            temp = temp.next;
        }

        // Return the head of the
        // deep copied list from the map
        return map.get(head);
    }

    // Function to print the cloned linked list
    public static void printClonedLinkedList(Node head) {
        while (head != null) {
            System.out.print("Data: " + head.data);
            if (head.random != null) {
                System.out.print(", Random: " + head.random.data);
            } else {
                System.out.print(", Random: nullptr");
            }
            System.out.println();
            // Move to the next node in the list
            head = head.next;
        }
    }

    // Main function
    public static void main(String[] args) {
        // Example linked list: 7 -> 14 -> 21 -> 28
        Node head = new Node(7);
        head.next = new Node(14);
        head.next.next = new Node(21);
        head.next.next.next = new Node(28);

        // Assigning random pointers
        head.random = head.next.next;
        head.next.random = head;
        head.next.next.random = head.next.next.next;
        head.next.next.next.random = head.next;

        System.out.println("Original Linked List with Random Pointers:");
        printClonedLinkedList(head);

        // Clone the linked list
        Node clonedList = cloneLL(head);

        System.out.println("\nCloned Linked List with Random Pointers:");
        printClonedLinkedList(clonedList);
    }
}
```

Output: Original Linked List with Random Pointers: Data: 7, Random: 21 Data: 14, Random: 7 Data: 21, Random: 28 Data: 28, Random: 14 Cloned Linked List with Random Pointers: Data: 7, Random: 21 Data: 14, Random: 7 Data: 21, Random: 28 Data: 28, Random: 14

Complexity Analysis

Time Complexity: O(2N) where N is the number of nodes in the linked list. The linked list is traversed twice, once for creating copies of each node and for the second time to set the next and random pointers for each copied node. The time to access the nodes in the map is O(1) due to hashing.

Space Complexity : O(N)+O(N)where N is the number of nodes in the linked list as all nodes are stored in the map to maintain mappings and the copied linked lists takes O(N) space as well.

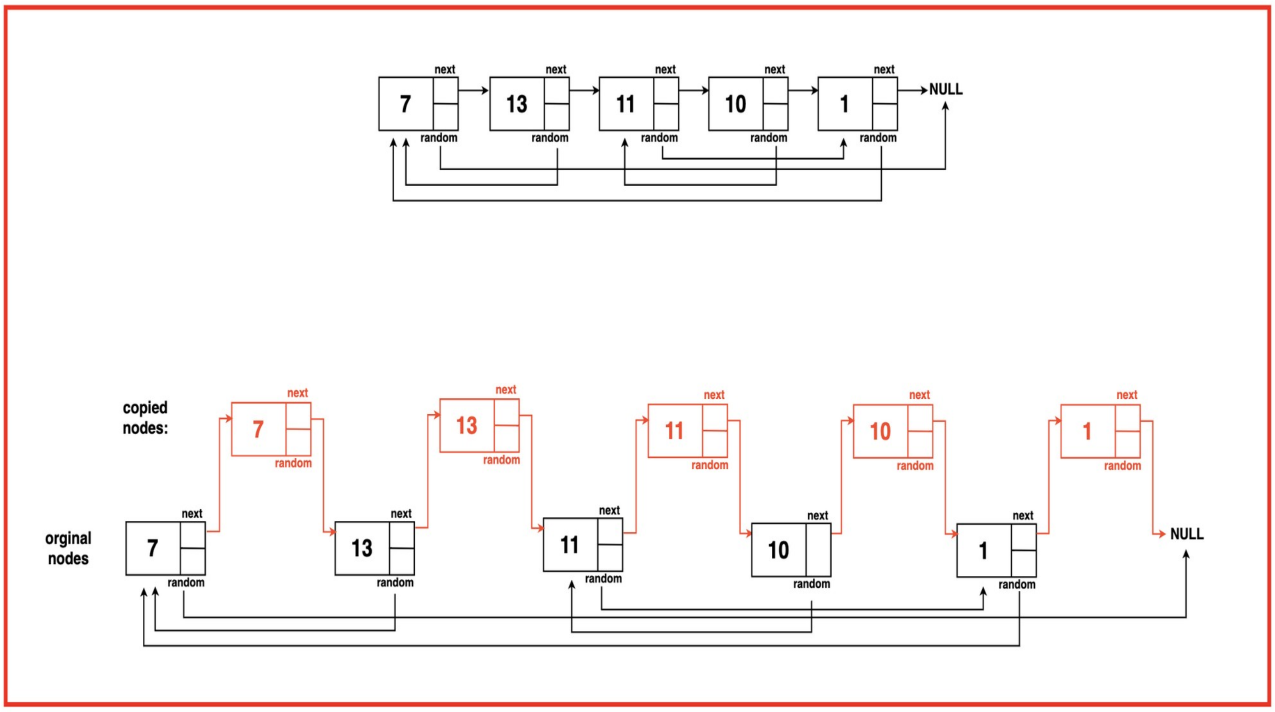
Optimal Approach
Algorithm / Intuition

The previous approach uses an extra space complexity of creating mappings between the original and copied nodes. Instead of creating duplicate nodes and storing them in a map, insert it in between the original node and the next node for quick access without the need for additional space.

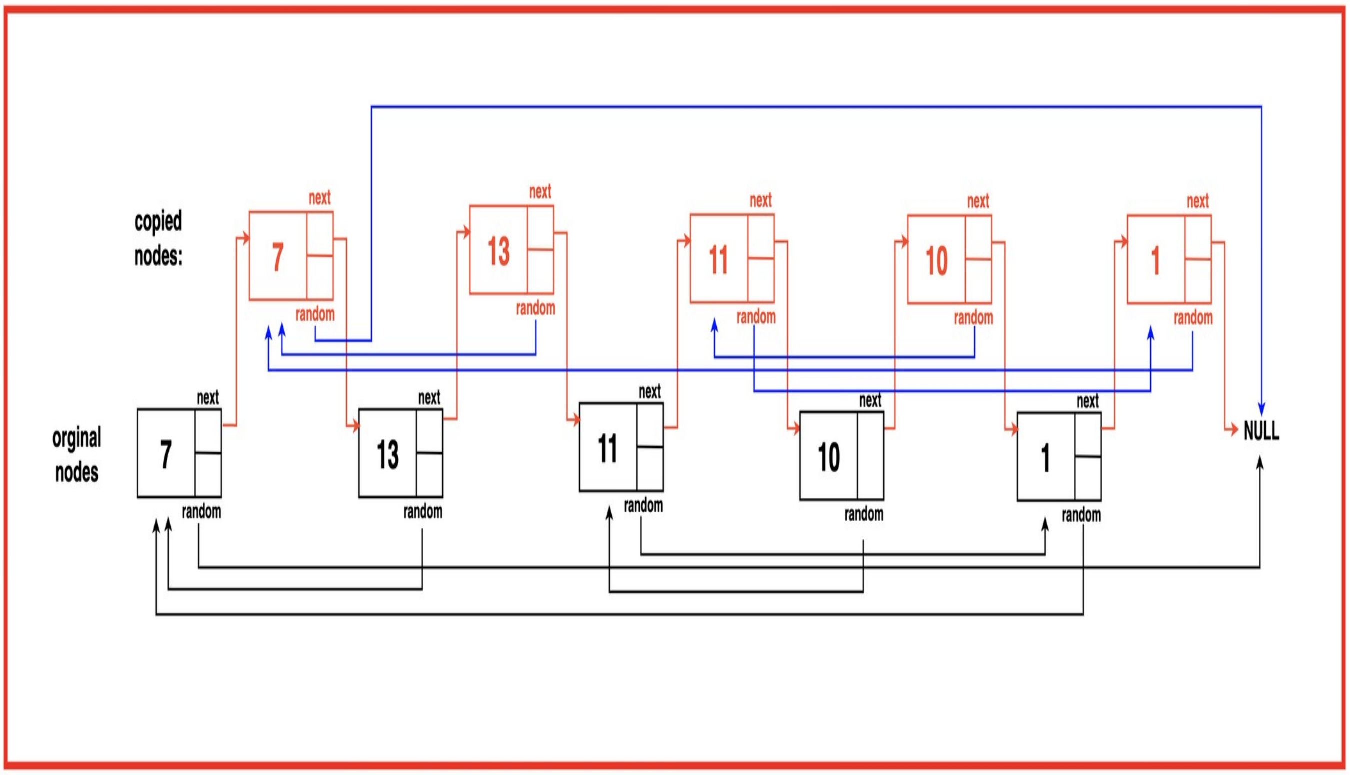
Traverse the list again to set the random pointer of copied nodes to the corresponding copied node duplicating the original arrangement. As a final traversal, separate the copied and original nodes by detaching alternate nodes.

Algorithm

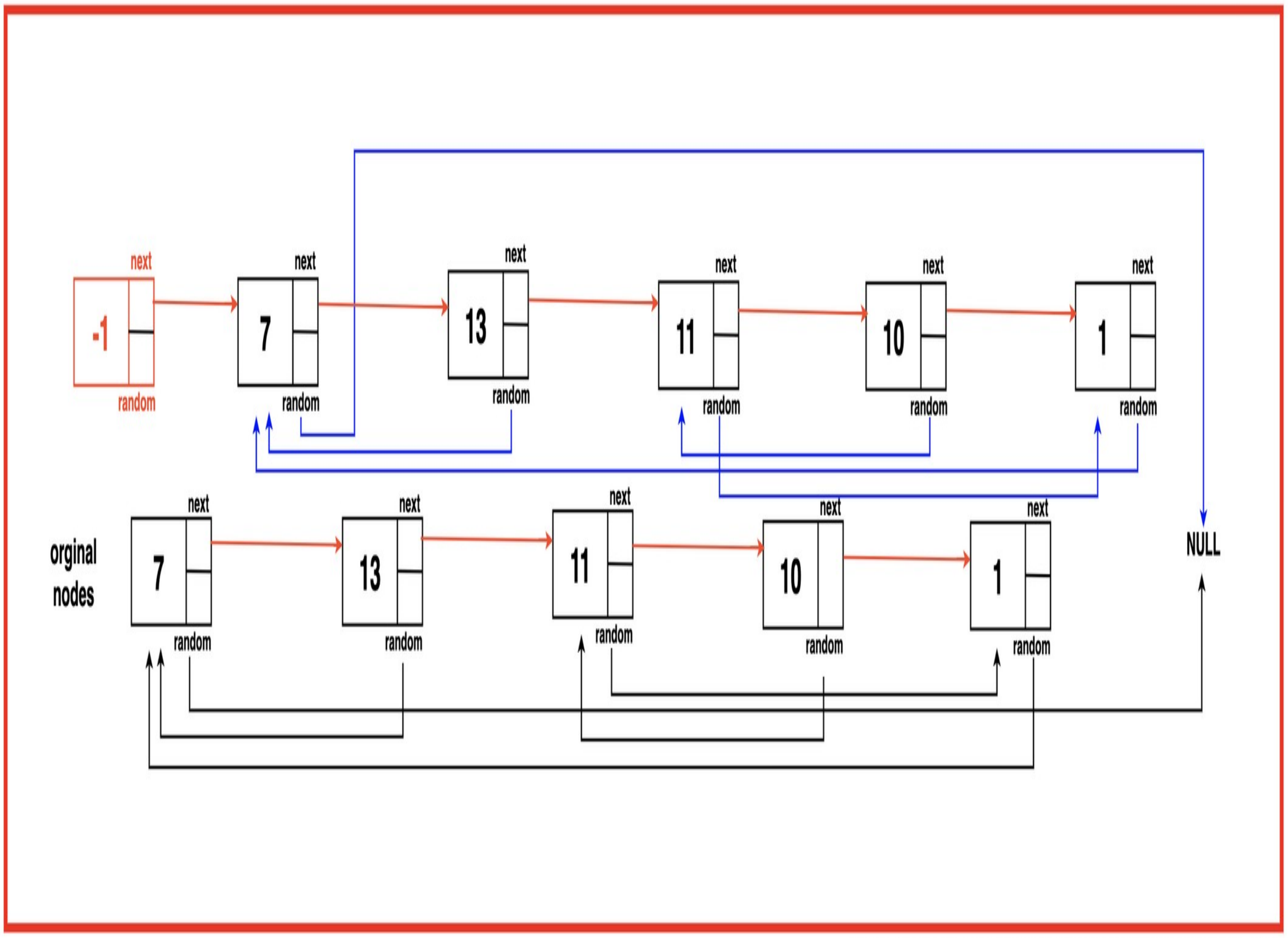
Step 1: Traverse the original node and create a copy of each node and insert it in between the original node and the next node.



Step 2: Traverse this modified list and for each original node that has a random pointer, set the copied node's random pointer to the corresponding copies random node. If the original node's random pointer is full, set the copied node's random pointe to null as well.



Step 3: Recursion Traverse the modified list again and extract the coped nodes by breaking the links between the original nodes and the copied nodes. Revert the original list to its initial state by fixing the next pointers.



Step 4: Return the head of the deep copy obtained after extracting the copied nodes from the modified list.

Code

```
// Node class to represent
// elements in the linked list
class Node {
    // Data stored in the node
    int data;
    // Pointer to the next node
    Node next;
    // Pointer to a random
    // node in the list
    Node random;

    // Constructors for Node class
    Node() {
        // Default constructor
        this.data = 0;
        this.next = null;
        this.random = null;
    }

    Node(int x) {
        // Constructor with data
        this.data = x;
        this.next = null;
        this.random = null;
    }

    Node(int x, Node nextNode, Node randomNode) {
        // Constructor with data,
        // next, and random pointers
        this.data = x;
        this.next = nextNode;
        this.random = randomNode;
    }
}

// Function to insert a copy of each
// node in between the original nodes
void insertCopyInBetween(Node head) {
    Node temp = head;
    while (temp != null) {
        Node nextElement = temp.next;
        // Create a new node with the same data
        Node copy = new Node(temp.data);

        // Point the copy's next to
        // the original node's next
        copy.next = nextElement;

        // Point the original
        // node's next to the copy
        temp.next = copy;

        // Move to the next original node
        temp = nextElement;
    }
}

// Function to connect random
// pointers of the copied nodes
void connectRandomPointers(Node head) {
    Node temp = head;
    while (temp != null) {
        // Access the copied node
        Node copyNode = temp.next;

        // If the original node
        // has a random pointer
        if (temp.random != null) {
            // Point the copied node's random to the
            // corresponding copied random node
            copyNode.random = temp.random.next;
        } else {
            // Set the copied node's random to
            // null if the original random is null
            copyNode.random = null;
        }

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

// Function to retrieve the
// deep copy of the linked list
Node getDeepCopyList(Node head) {
    Node temp = head;
    // Create a dummy node
    Node dummyNode = new Node(-1);
    // Initialize a result pointer
    Node res = dummyNode;

    while (temp != null) {
        // Creating a new List by
        // pointing to copied nodes
        res.next = temp.next;
        res = res.next;

        // Disconnect and revert back to the
        // initial state of the original linked list
        temp.next = temp.next.next;
        temp = temp.next;
    }

    // Return the deep copy of the
    // list starting from the dummy node
    return dummyNode.next;
}

// Function to clone the linked list
Node cloneLL(Node head) {
    // If the original list
    // is empty, return null
    if (head == null) return null;

    // Step 1: Insert copy of
    // nodes in between
    insertCopyInBetween(head);
    // Step 2: Connect random
    // pointers of copied nodes
    connectRandomPointers(head);
    // Step 3: Retrieve the deep
    // copy of the linked list
    return getDeepCopyList(head);
}

// Function to print the cloned linked list
void printClonedLinkedList(Node head) {
    while (head != null) {
        System.out.print("Data: " + head.data);
        if (head.random != null) {
            System.out.print(", Random: " + head.random.data);
        } else {
            System.out.print(", Random: null");
        }
        System.out.println();
        // Move to the next node
        head = head.next;
    }
}

// Main function
public static void main(String[] args) {
    // Example linked list: 7 -> 14 -> 21 -> 28
    Node head = new Node(7);
    head.next = new Node(14);
    head.next.next = new Node(21);
    head.next.next.next = new Node(28);

    // Assigning random pointers
    head.random = head.next.next;
    head.next.random = head;
    head.next.next.random = head.next.next.next;
    head.next.next.next.random = head.next;

    System.out.println("Original Linked List with Random Pointers:");
    printClonedLinkedList(head);

    // Clone the linked list
    Node clonedList = cloneLL(head);

    System.out.println("\nCloned Linked List with Random Pointers:");
    printClonedLinkedList(clonedList);
}
```

Output: Original Linked List with Random Pointers: Data: 7, Random: 21 Data: 14, Random: 7 Data: 21, Random: 28 Data: 28, Random: 14 Cloned Linked List with Random Pointers: Data: 7, Random: 21 Data: 14, Random: 7 Data: 21, Random: 28 Data: 28, Random: 14

Complexity Analysis

Time Complexity: O(3N)where N is the number of nodes in the linked list. The algorithm makes three traversals of the linked list, once to create copies and insert them between original nodes, then to set the random pointers of the copied nodes to their appropriate copied nodes and then to separate the copied and original nodes.

Space Complexity : O(N) where N is the number of nodes in the linked list as the only extra additional space allocated it to create the copied list without creating any other additional data structures.