Dear reader, welcome to the article on the problem named ‘[**Transform to Normal from Left-Cloned Tree**](https://www.pepcoding.com/resources/online-java-foundation/binary-tree/transform-to-normal-from-left-cloned-tree-official/ojquestion)’.

If you have not solved the previous problem: ‘[Transform To Left-Cloned Tree](https://www.pepcoding.com/resources/online-java-foundation/binary-tree/transform-to-left-cloned-tree-official/ojquestion)’, then I recommend first solve it, and then jump into this problem.

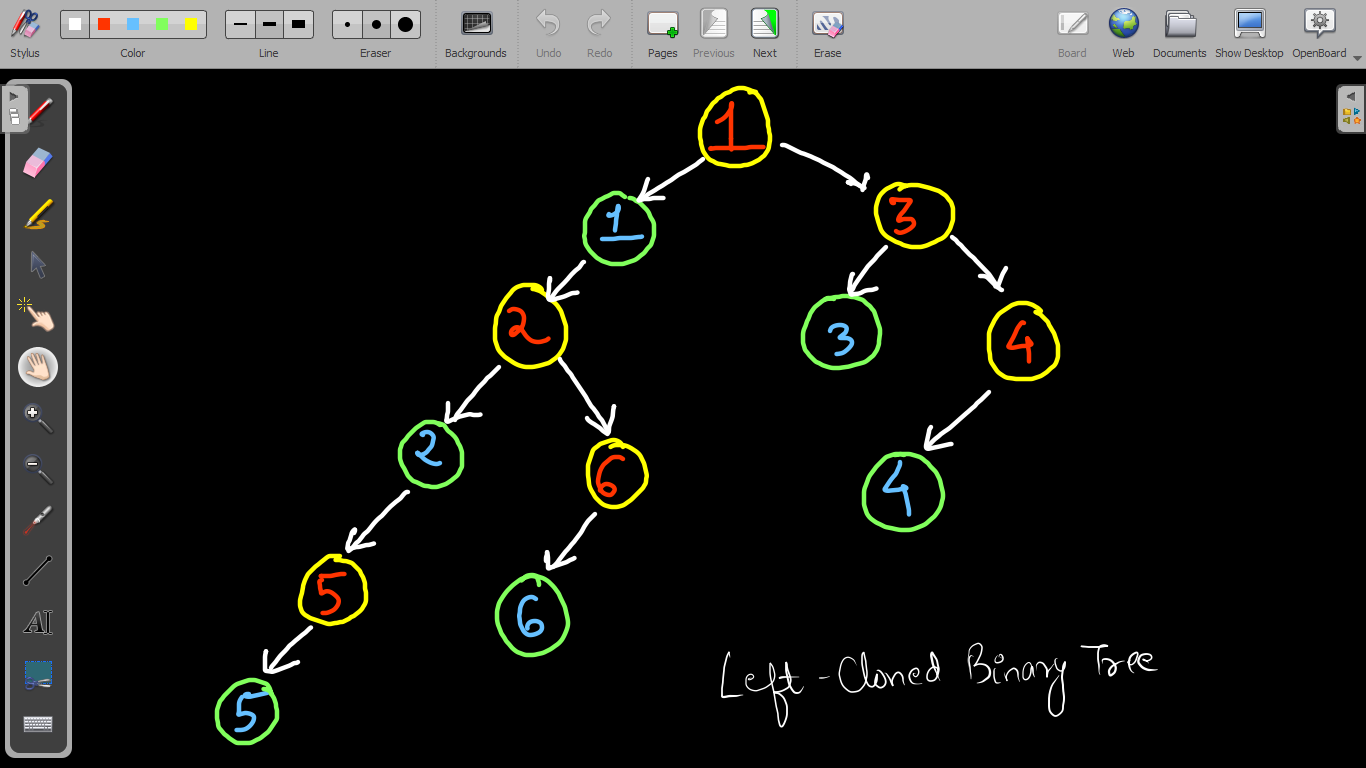
***Problem Statement:***

* You are given a partially written BinaryTree class. (Input and Output is managed for you.)
* You are required to complete the body of the ***transBackFromLeftClonedTree*** function. The function is expected to convert a left-cloned tree back to its original form.
* The left cloned tree is discussed in the previous question. In a left-clone tree a new node for every node equal in value to it is inserted between itself and it's left child.
* The input will always represent a valid left-cloned tree, i.e. all nodes will have a duplicate node as their left child.

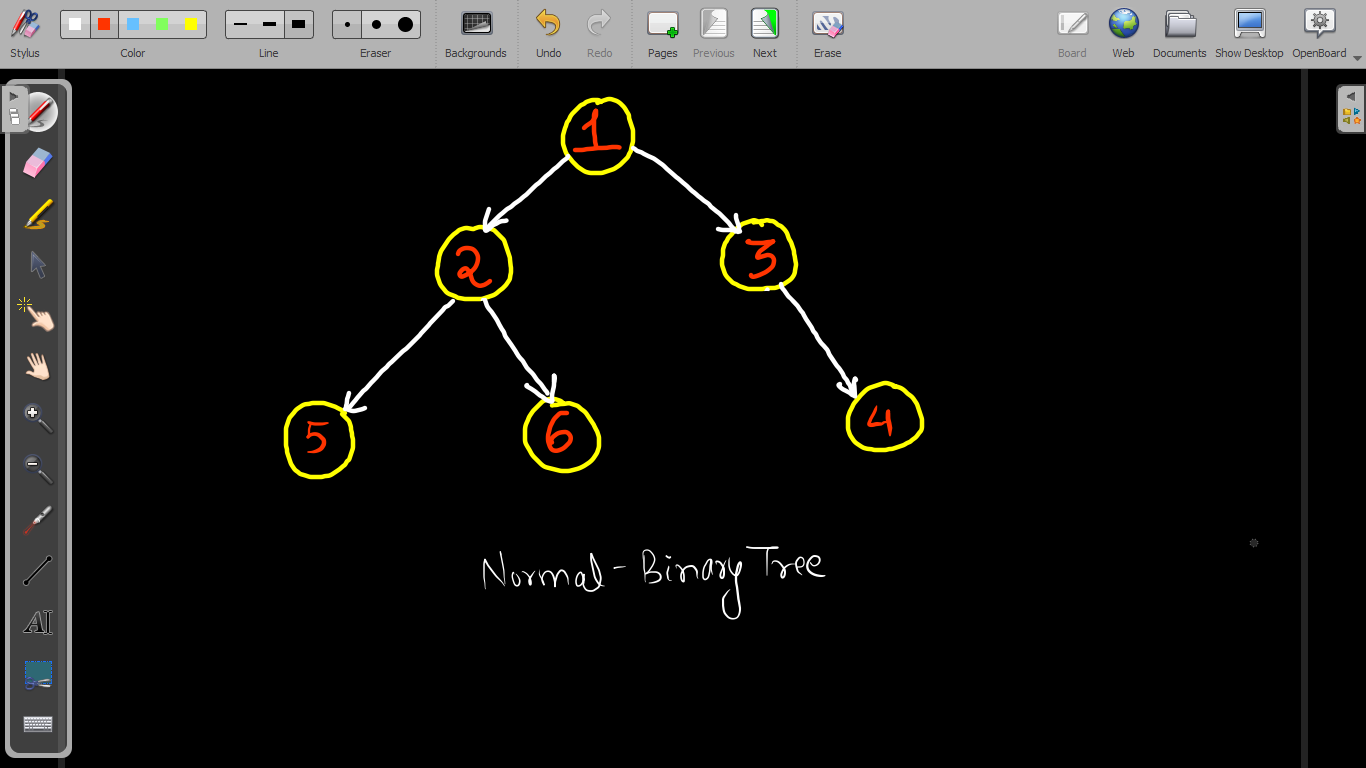
For more details, check out the [question video](https://www.youtube.com/watch?v=YCd-jyAF0CM&list=TLGGs5zZF5wvfYIwNTA3MjAyMQ).

***Example:***

Input:



Output:



***Solution:***

So, what did we perform in the previous problem? It was our heaviest weaponry, and that is ***RECURSION***.

Try to think and do ***reverse engineering***, so that you can get back the normal binary tree. We need to delete all the duplicate nodes that are present in the left-cloned binary tree.

So, let us try to find the recursive solution using the ***low-level thinking*** of defining expectations, faith and meeting expectation with faith.

**Expectation**: We expect that the entire left-cloned binary tree rooted at the *node* will transform back to a normal binary tree.

***transBackFromLeftClonedTree (node)***

**Faith**: We will keep on the smaller subproblems, i.e. we will have complete belief that the left-cloned *left subtree* and the right-cloned *right subtree* know how to transform back to a normal binary tree.

But, there is a slight twist here. What should be the left child of the current node. Should we take left as node.left?

No, since the direct left child of the root node is it’s duplicate node only, hence the left subtree is not *node.left* but *node.left.left*.

Try to think about it for a minute, why node.left.left will give us back the original left child and how we will be able to get rid of the duplicate node easily.

***node.left = transBackFromLeftClonedTree (node.left.left)***

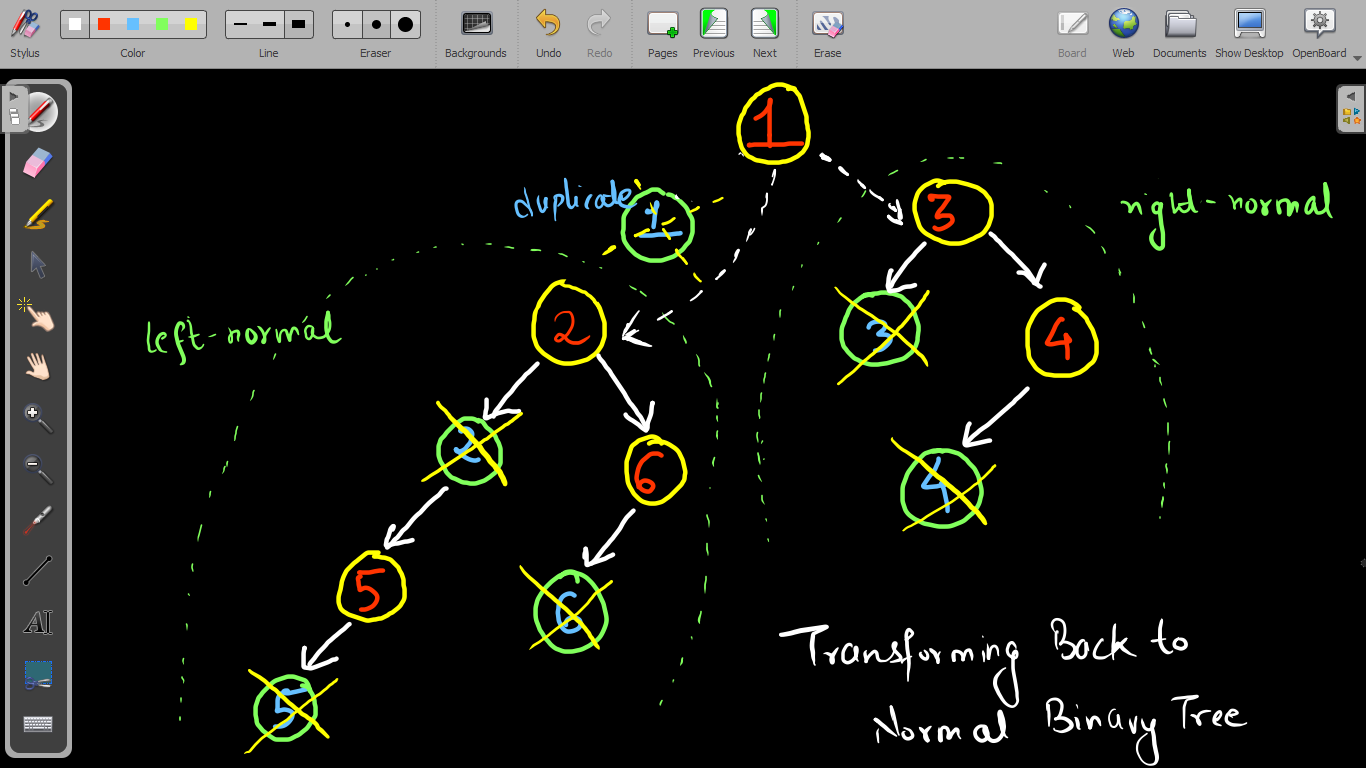
***node.right = transBackFromLeftClonedTree (node.right)***

**Meeting Expectation with Faith**: We needed to delete the duplicate node of the current root, which is present as the direct left child.

By updating node’s left as normal tree of node.left.left, we have automatically deleted the duplicate node of the root.

***Why***? It is because Java has an automatic garbage collection mechanism, so we need not worry about the deallocation of memory.

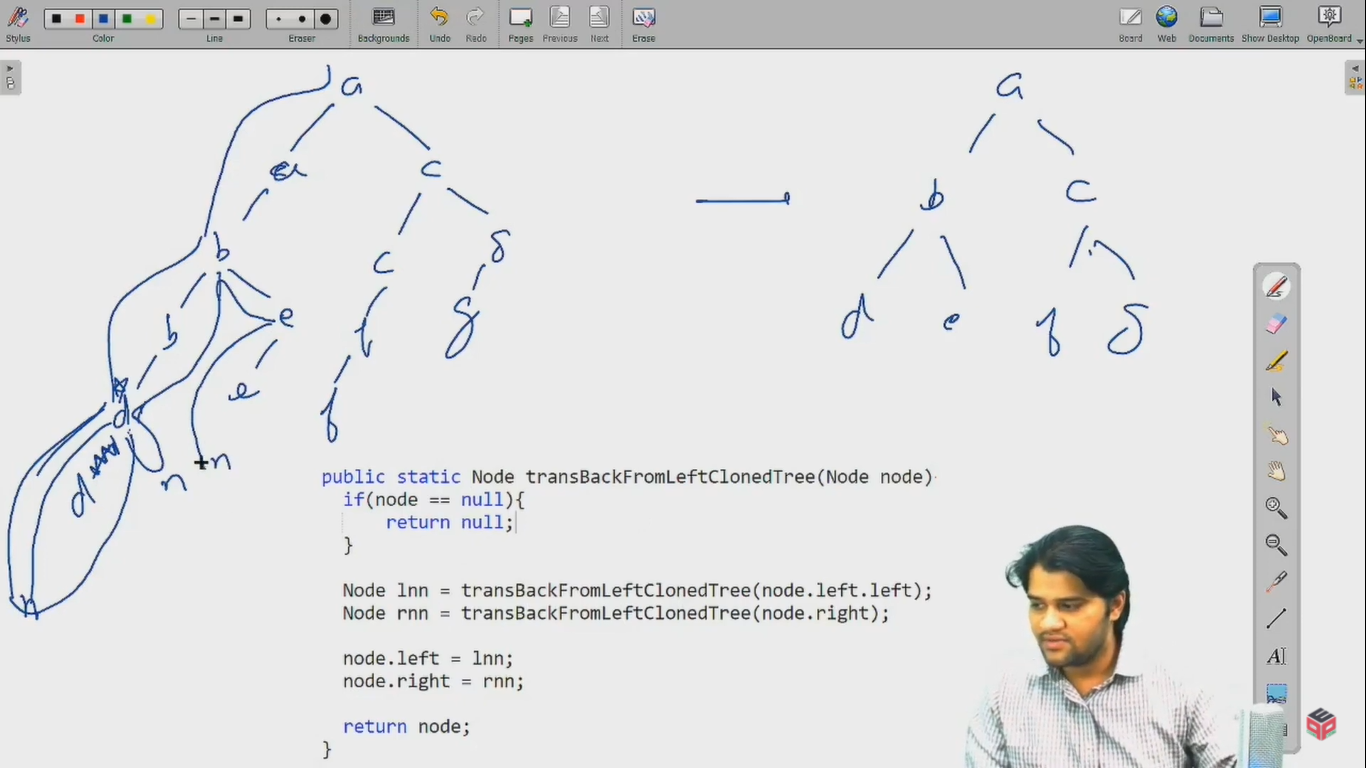
Thus, we can simply return the root node: **return node**

****

What should be the ***base case***?

If the root node is a ***null*** node, then simply return null, as a null node can be considered as a normal binary tree (of size 0).

***Pseudo Code/ Algorithm***



***Implementation***

*Note*: Before reading the Code, we recommend that you must try to come up with the solution on your own. Now, hoping that you have tried by yourself, here is the Java code.

import java.io.\*;

import java.util.\*;

public class Main {

public static class Node {

int data;

Node left;

Node right;

Node(int data, Node left, Node right) {

this.data = data;

this.left = left;

this.right = right;

}

}

public static class Pair {

Node node;

int state;

Pair(Node node, int state) {

this.node = node;

this.state = state;

}

}

public static Node construct(Integer[] arr) {

Node root = new Node(arr[0], null, null);

Pair rtp = new Pair(root, 1);

Stack<Pair> st = new Stack<>();

st.push(rtp);

int idx = 0;

while (st.size() > 0) {

Pair top = st.peek();

if (top.state == 1) {

idx++;

if (arr[idx] != null) {

top.node.left = new Node(arr[idx], null, null);

Pair lp = new Pair(top.node.left, 1);

st.push(lp);

} else {

top.node.left = null;

}

top.state++;

} else if (top.state == 2) {

idx++;

if (arr[idx] != null) {

top.node.right = new Node(arr[idx], null, null);

Pair rp = new Pair(top.node.right, 1);

st.push(rp);

} else {

top.node.right = null;

}

top.state++;

} else {

st.pop();

}

}

return root;

}

public static void display(Node node) {

if (node == null) {

return;

}

String str = "";

str += node.left == null ? "." : node.left.data + "";

str += " <- " + node.data + " -> ";

str += node.right == null ? "." : node.right.data + "";

System.out.println(str);

display(node.left);

display(node.right);

}

public static Node transBackFromLeftClonedTree(Node node){

if(node == null){

return null;

}

node.left = transBackFromLeftClonedTree(node.left.left);

node.right = transBackFromLeftClonedTree(node.right);

return node;

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n = Integer.parseInt(br.readLine());

Integer[] arr = new Integer[n];

String[] values = br.readLine().split(" ");

for (int i = 0; i < n; i++) {

if (values[i].equals("n") == false) {

arr[i] = Integer.parseInt(values[i]);

} else {

arr[i] = null;

}

}

Node root = construct(arr);

root = transBackFromLeftClonedTree(root);

display(root);

}

}

This code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=AvXVChZoZkU&list=TLGGT07f8UkL-twwNTA3MjAyMQ). Do check it out to understand the concept completely.

* What is the ***time complexity*** of the above code?

We are traversing only n/2 nodes (skipping the duplicate nodes), hence total time complexity will be O(n/2) = ***O(n)***.

* What is the ***space complexity*** of the above code?

We are not taking any extra space, in fact we are freeing up some space by deleting duplicate nodes. Hence, the solution takes ***O(1) auxiliary space***.

Although, there is still recursion call stack space of ***O(d)*** where d = maximum depth of the tree.

Hope that you liked the article on *Transform to Normal from Left-Cloned Tree*.

Subscribe to Pepcoding’s youtube channel for more such amazing video content on Data Structures & Algorithms.

You can suggest any improvements to the article on our telegram channel, or on the youtube channel’s comment section.

Article Contributed by: [Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)