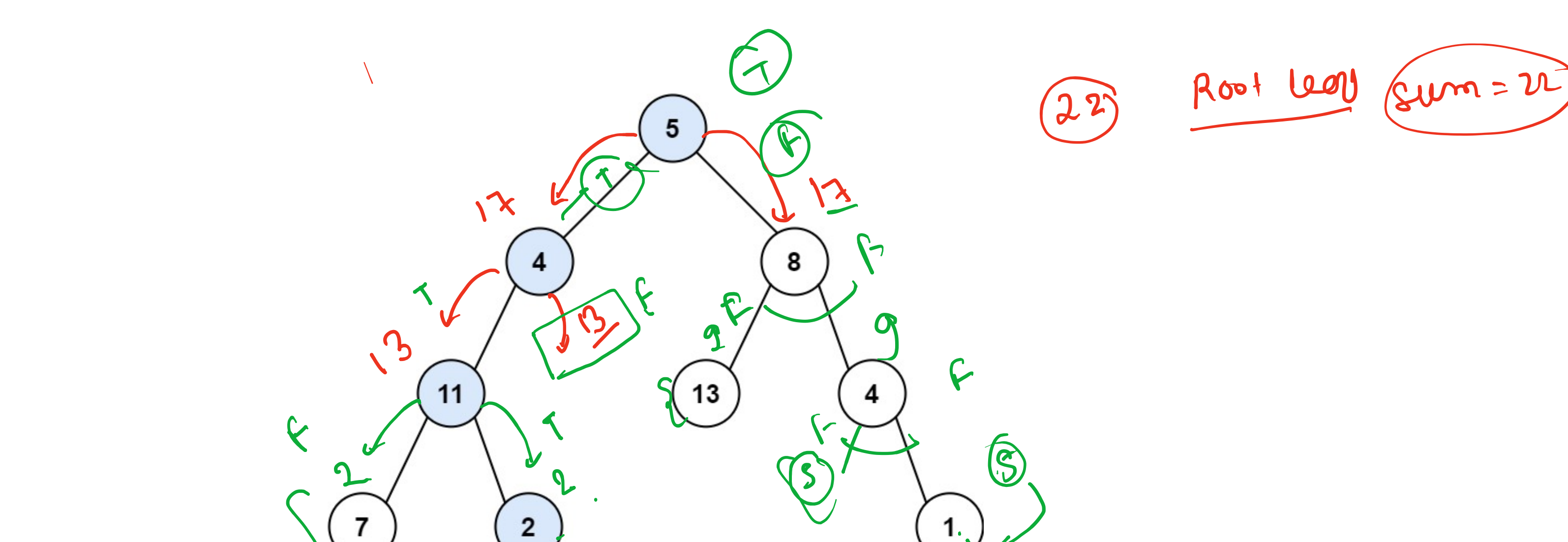
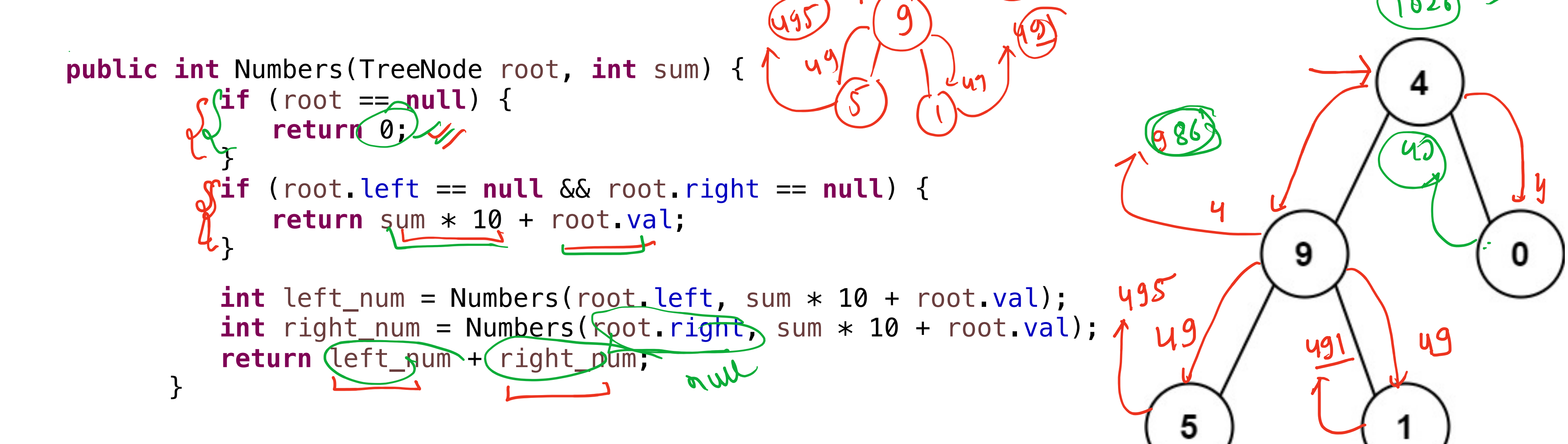


```

public int Numbers(TreeNode root, int sum) {
    if (root == null) {
        return 0;
    }
    if (root.left == null && root.right == null) {
        return sum * 10 + root.val;
    }
    int left_num = Numbers(root.left, sum * 10 + root.val);
    int right_num = Numbers(root.right, sum * 10 + root.val);
    return left_num + right_num;
}

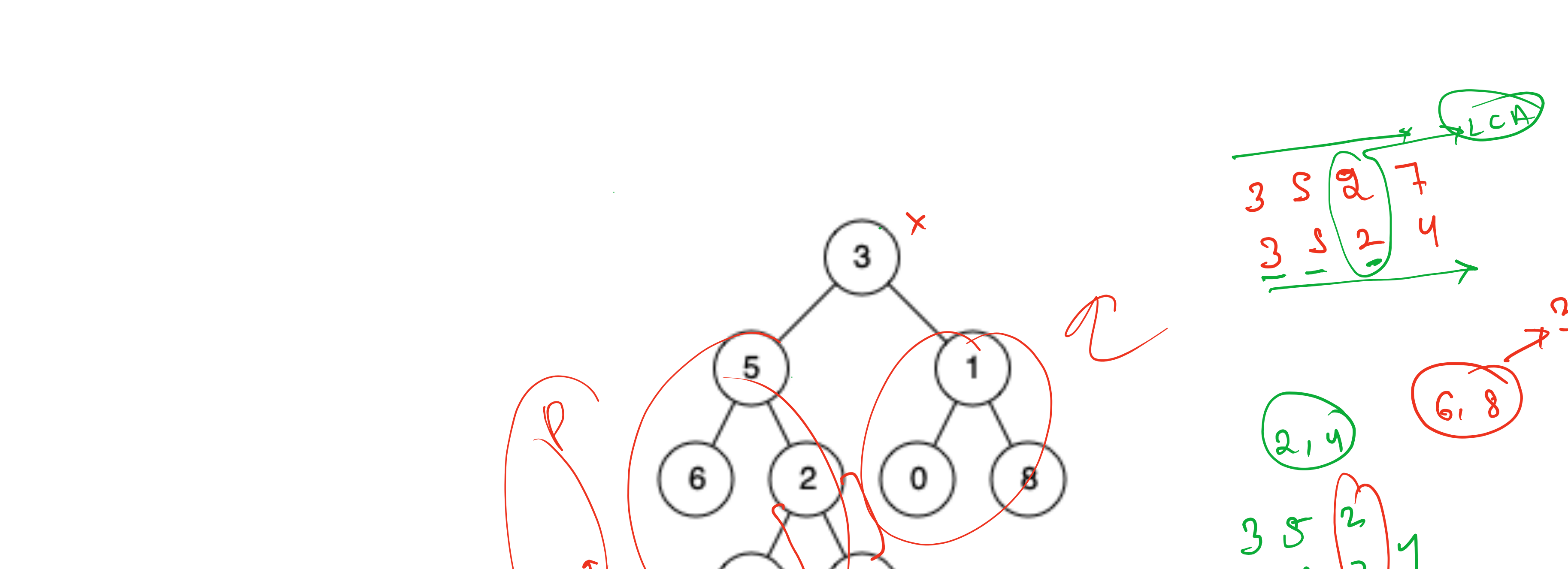
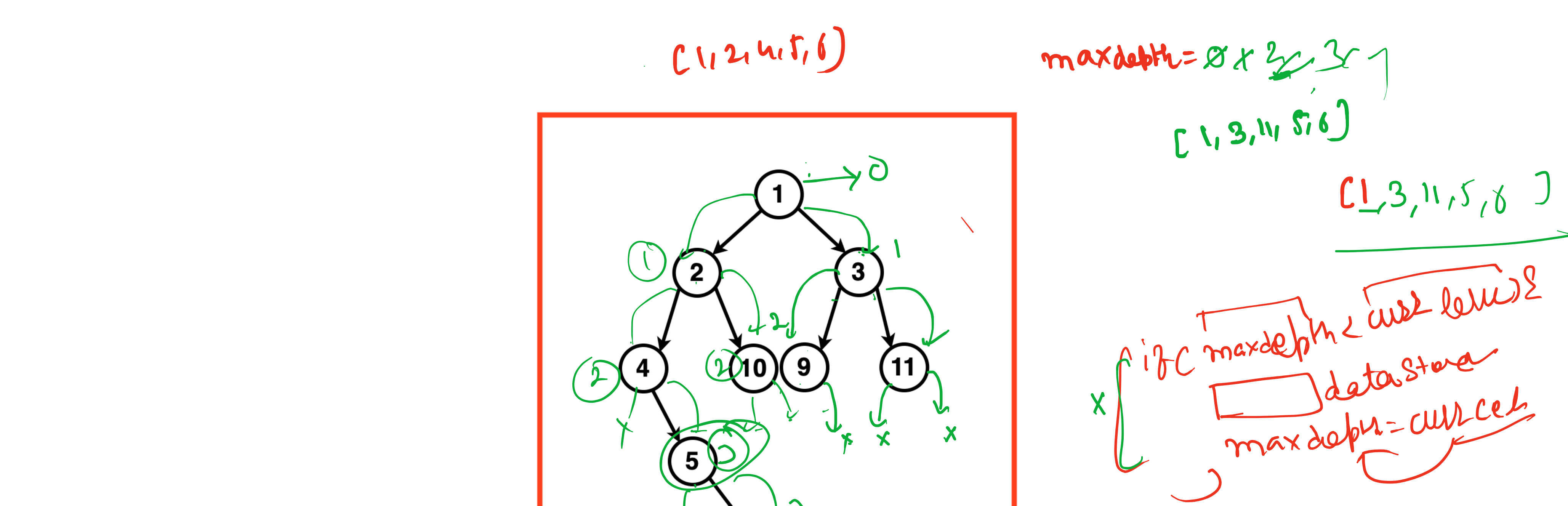
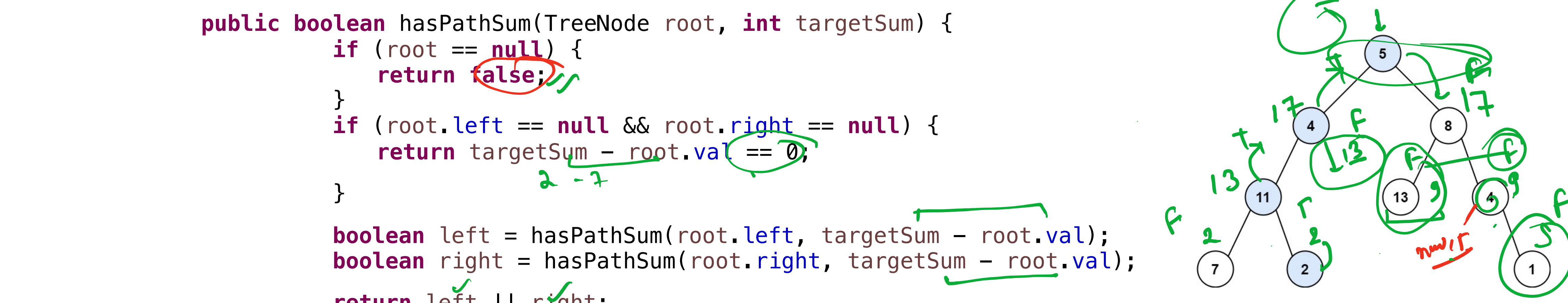
```



```

public boolean hasPathSum(TreeNode root, int targetSum) {
    if (root == null) {
        return false;
    }
    if (root.left == null && root.right == null) {
        return targetSum - root.val == 0;
    }
    boolean left = hasPathSum(root.left, targetSum - root.val);
    boolean right = hasPathSum(root.right, targetSum - root.val);
    return left || right;
}

```



The diagram illustrates a binary tree structure and the recursive logic for finding the Lowest Common Ancestor (LCA) of two nodes, p and q.

Tree Structure:

- Root node: 3
- Node 3's left child: 5
- Node 3's right child: 1
- Node 5's left child: 6
- Node 5's right child: 2
- Node 1's left child: 0
- Node 1's right child: 8

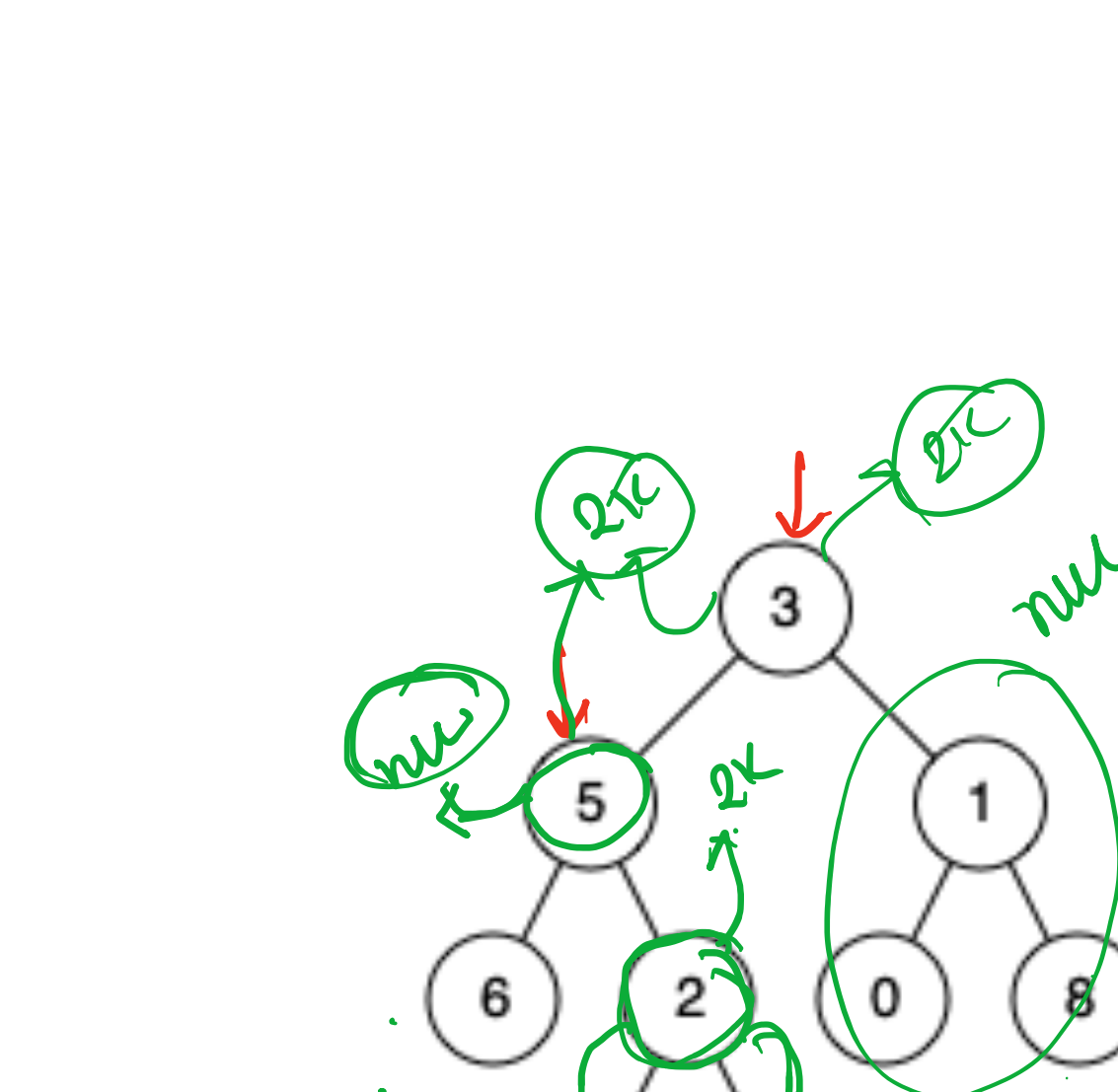
Handwritten Annotations:

- Green circles highlight nodes 3, 5, 1, 2, and 0.
- Red arrows indicate the path from the root (3) down to node 2 (3 → 5 → 2).
- Green arrows indicate the path from node 2 up to the root (2 → 5 → 3).
- Green circles around nodes 5 and 1, and the arrow between them, represent the recursive step where the LCA is found by comparing the left and right subtrees.

Code Snippet:

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null) {
        return null;
    }
    if (root == p || root == q) {
        return root;
    }
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);
}
```

The code implements a recursive algorithm to find the LCA. It checks if the current node is null, or if it is one of the target nodes p or q. If not, it recursively searches the left and right subtrees and returns the node where the paths to p and q converge.



```

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null) {
        return null;
    }
    if (root == p || root == q) {
        return root;
    }
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);
    if (left != null && right != null) {
        return root;
    }
    else if (left == null) {
        return right;
    }
    else {
        return left;
    }
}

```

