<https://leetcode.com/problems/step-by-step-directions-from-a-binary-tree-node-to-another/description/>

You are given the root of a **binary tree** with n nodes. Each node is uniquely assigned a value from 1 to n. You are also given an integer startValue representing the value of the start node s, and a different integer destValue representing the value of the destination node t.

Find the **shortest path** starting from node s and ending at node t. Generate step-by-step directions of such path as a string consisting of only the **uppercase** letters 'L', 'R', and 'U'. Each letter indicates a specific direction:

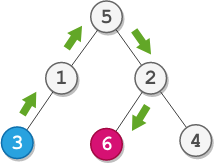
'L' means to go from a node to its **left child** node.

'R' means to go from a node to its **right child** node.

'U' means to go from a node to its **parent** node.

Return the step-by-step directions of the **shortest path** from node s to node t.

**Example 1:**

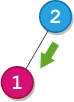


**Input:** root = [5,1,2,3,null,6,4], startValue = 3, destValue = 6

**Output:** "UURL"

**Explanation:** The shortest path is: 3 → 1 → 5 → 2 → 6.

**Example 2:**



**Input:** root = [2,1], startValue = 2, destValue = 1

**Output:** "L"

**Explanation:** The shortest path is: 2 → 1.

**Constraints:**

The number of nodes in the tree is n.

2 <= n <= 10^5

1 <= Node.val <= n

All the values in the tree are unique.

1 <= startValue, destValue <= n

startValue != destValue

**Attempt 1: 2024-07-16**

**Solution 1: LCA + Find path (30 min)**

**Wrong Solution**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public String getDirections(TreeNode root, int startValue, int destValue) {

TreeNode lca = findLCA(root, startValue, destValue);

StringBuilder lcaToStart = new StringBuilder();

StringBuilder lcaToDest = new StringBuilder();

// Find path from lca to startValue

findPath(lca, startValue, lcaToStart);

// Find path from lca to destValue

findPath(lca, destValue, lcaToDest);

// Update LCA to start path by replace all chars to 'U'

String startToLCA = "U".repeat(lcaToStart.length());

return startToLCA + lcaToDest.toString();

}

// Find path to a node from given root

// Note: Don't not return "String" type as its immutable,

// instead return as boolean, and using StringBuilder with

// backtracking will help on building path

private boolean findPath(TreeNode root, int value, StringBuilder sb) {

// Base case: Target value not found

if(root == null) {

return false;

}

// Base case: Target value found

if(root.val == value) {

return true;

}

int len = sb.length();

// Attempt on left branch

sb.append("L");

// If find target value in left branch

if(findPath(root.left, value, sb)) {

return true;

}

// If not find target value in left branch, backtracking by

// removing last move

sb.setLength(len - 1);

// Attempt on right branch

sb.append("R");

// If find target value in right branch

if(findPath(root.right, value, sb)) {

return true;

}

// If not find target value in right branch, backtracking by

// removing last move

sb.setLength(len - 1);

// After all move still not found target value

return false;

}

// Find the Lowest Common Ancestor (LCA) of two nodes

private TreeNode findLCA(TreeNode root, int p, int q) {

if(root == null || root.val == p || root.val == q) {

return root;

}

TreeNode left = findLCA(root.left, p, q);

TreeNode right = findLCA(root.right, p, q);

if(left != null && right != null) {

return root;

}

return left != null ? left : right;

}

}

**Error log**

java.lang.StringIndexOutOfBoundsException: String index out of range: -1

at line 325, java.base/java.lang.AbstractStringBuilder.setLength

at line 91, java.base/java.lang.StringBuilder.setLength

at line 52, Solution.findPath

at line 24, Solution.getDirections

at line 56, \_\_DriverSolution\_\_.\_\_helper\_\_

at line 92, \_\_Driver\_\_.main

**Error out reason refer to ChatGPT**

**The StringIndexOutOfBoundsException occurs because of the following line in the findPath method:**

path.setLength(len - 1);

**This line is problematic because len is the initial length of the path StringBuilder before appending 'L' or 'R'. When the initial length len is zero (i.e., at the root or very beginning of the path), len - 1 becomes -1, which is an invalid index for the setLength method, causing the StringIndexOutOfBoundsException.**

**Correct Solution**

**Style 1: Capture initial 'len'**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public String getDirections(TreeNode root, int startValue, int destValue) {

        TreeNode lca = findLCA(root, startValue, destValue);

        StringBuilder lcaToStart = new StringBuilder();

        StringBuilder lcaToDest = new StringBuilder();

        // Find path from lca to startValue

        findPath(lca, startValue, lcaToStart);

        // Find path from lca to destValue

        findPath(lca, destValue, lcaToDest);

        // Update LCA to start path by replace all chars to 'U'

        String startToLCA = "U".repeat(lcaToStart.length());

        return startToLCA + lcaToDest.toString();

    }

    // Find path to a node from given root

    // Note: Don't not return "String" type as its immutable,

    // instead return as boolean, and using StringBuilder with

    // backtracking will help on building path

    private boolean findPath(TreeNode root, int value, StringBuilder sb) {

        // Base case: Target value not found

        if(root == null) {

            return false;

        }

        // Base case: Target value found

        if(root.val == value) {

            return true;

        }

        // 'len' is the initial length of the path StringBuilder before appending 'L' or 'R'

        int len = sb.length();

        // Attempt on left branch

        sb.append("L");

        // If find target value in left branch

        if(findPath(root.left, value, sb)) {

            return true;

        }

        // If not find target value in left branch, backtracking by removing last move

// Instead of setting the length to len - 1, you should set it back to len,

// the original length before the recent append.

        sb.setLength(len);

        // Attempt on right branch

        sb.append("R");

        // If find target value in right branch

        if(findPath(root.right, value, sb)) {

            return true;

        }

        // If not find target value in left branch, backtracking by removing last move

// Instead of setting the length to len - 1, you should set it back to len,

// the original length before the recent append.

        sb.setLength(len);

        // After all move still not found target value

        return false;

    }

    // Find the Lowest Common Ancestor (LCA) of two nodes

    private TreeNode findLCA(TreeNode root, int p, int q) {

        if(root == null || root.val == p || root.val == q) {

            return root;

        }

        TreeNode left = findLCA(root.left, p, q);

        TreeNode right = findLCA(root.right, p, q);

        if(left != null && right != null) {

            return root;

        }

        return left != null ? left : right;

    }

}

**Style 2: No capture initial 'len'**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public String getDirections(TreeNode root, int startValue, int destValue) {

        TreeNode lca = findLCA(root, startValue, destValue);

        StringBuilder lcaToStart = new StringBuilder();

        StringBuilder lcaToDest = new StringBuilder();

        // Find path from lca to startValue

        findPath(lca, startValue, lcaToStart);

        // Find path from lca to destValue

        findPath(lca, destValue, lcaToDest);

        // Update LCA to start path by replace all chars to 'U'

        String startToLCA = "U".repeat(lcaToStart.length());

        return startToLCA + lcaToDest.toString();

    }

    // Find path to a node from given root

    // Note: Don't not return "String" type as its immutable,

    // instead return as boolean, and using StringBuilder with

    // backtracking will help on building path

    private boolean findPath(TreeNode root, int value, StringBuilder sb) {

        // Base case: Target value not found

        if(root == null) {

            return false;

        }

        // Base case: Target value found

        if(root.val == value) {

            return true;

        }

        //int len = sb.length();

        // Attempt on left branch

        sb.append("L");

        // If find target value in left branch

        if(findPath(root.left, value, sb)) {

            return true;

        }

        // If not find target value in left branch, backtracking by

        // removing last move

        sb.setLength(sb.length() - 1);

        // Attempt on right branch

        sb.append("R");

        // If find target value in right branch

        if(findPath(root.right, value, sb)) {

            return true;

        }

        // If not find target value in right branch, backtracking by

        // removing last move

        sb.setLength(sb.length() - 1);

        // After all move still not found target value

        return false;

    }

    // Find the Lowest Common Ancestor (LCA) of two nodes

    private TreeNode findLCA(TreeNode root, int p, int q) {

        if(root == null || root.val == p || root.val == q) {

            return root;

        }

        TreeNode left = findLCA(root.left, p, q);

        TreeNode right = findLCA(root.right, p, q);

        if(left != null && right != null) {

            return root;

        }

        return left != null ? left : right;

    }

}

**Refer to chatGPT**

Leetcode 2096 - "Step-By-Step Directions From a Binary Tree Node to Another" involves finding a path from one node to another in a binary tree.

**Problem Explanation**

Binary Tree Directions: Given a binary tree and two nodes, start and destination, we need to determine the shortest path from start to destination. The path should be represented using 'L' for moving to the left child and 'R' for moving to the right child.

Steps to Solve:

Find the Lowest Common Ancestor (LCA) of the two nodes. This ensures we have a common node to start diverging paths from.

Calculate the path from the start node to the LCA.

Calculate the path from the LCA to the destination node.

Combine these paths to get the final result.

**Java Implementation**

Here's a solution in Java:

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public String getDirections(TreeNode root, int startValue, int destValue) {

TreeNode lca = findLCA(root, startValue, destValue);

StringBuilder startToLCA = new StringBuilder();

StringBuilder lcaToDest = new StringBuilder();

findPath(lca, startValue, startToLCA);

findPath(lca, destValue, lcaToDest);

// Convert path to LCA for start node into "U" steps

String startToLCAPath = "U".repeat(startToLCA.length());

return startToLCAPath + lcaToDest.toString();

}

private TreeNode findLCA(TreeNode root, int p, int q) {

if (root == null || root.val == p || root.val == q) {

return root;

}

TreeNode left = findLCA(root.left, p, q);

TreeNode right = findLCA(root.right, p, q);

if (left != null && right != null) {

return root;

}

return left != null ? left : right;

}

private boolean findPath(TreeNode root, int value, StringBuilder path) {

if (root == null) {

return false;

}

if (root.val == value) {

return true;

}

path.append('L');

if (findPath(root.left, value, path)) {

return true;

}

path.setLength(path.length() - 1);

path.append('R');

if (findPath(root.right, value, path)) {

return true;

}

path.setLength(path.length() - 1);

return false;

}

}

**Explanation**

Finding the LCA:

findLCA function is a standard approach to find the Lowest Common Ancestor in a binary tree.

It checks if the current node is either p or q. If yes, it returns the current node.

Otherwise, it recursively searches in the left and right subtrees.

If both left and right searches return non-null values, the current node is the LCA.

Finding the Path:

findPath function uses a depth-first search (DFS) approach to find the path from the LCA to a target node.

It appends 'L' or 'R' to the path while traversing the tree.

If the target node is found, it returns true, indicating the path is successfully built.

If not found in a subtree, it backtracks by removing the last appended character.

Combining the Paths:

startToLCAPath is created by converting each character in startToLCA to 'U', indicating upward movement from the start node to the LCA.

The final path is the concatenation of startToLCAPath and lcaToDest.

**Complexity Analysis**

Time Complexity: O(n), where n is the number of nodes in the binary tree. Both findLCA and findPath functions traverse the tree, resulting in a linear time complexity.

Space Complexity: O(h), where h is the height of the tree. This is the space used by the recursion stack, which in the worst case (a completely unbalanced tree) is O(n), but on average for a balanced tree is O(log n).

This solution effectively finds the directions from the start node to the destination node in a binary tree by leveraging the properties of the Lowest Common Ancestor and pathfinding.

**Refer to**

<https://leetcode.com/problems/step-by-step-directions-from-a-binary-tree-node-to-another/solutions/1612123/this-actually-contains-several-tree-questions-nice-one/>

get lowest common ancestor

find the path from ancestor to start / dest

So this is a very good tree question

lowestCommonAncestor + find path + calculate depth

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public String getDirections(TreeNode root, int startValue, int destValue) {

TreeNode ancestor = lowestCommonAncestor(root, startValue, destValue);

List<String> toStart = new ArrayList<>();

getDirection(ancestor, startValue, toStart);

List<String> toDest = new ArrayList<>();

getDirection(ancestor, destValue, toDest);

StringBuilder sb = new StringBuilder();

for (int i = 0; i < toStart.size(); i++) sb.append("U");

for (String s : toDest) sb.append(s);

return sb.toString();

}

private TreeNode lowestCommonAncestor(TreeNode root, int p, int q) {

if (root == null || p == root.val || q == root.val) return root;

TreeNode left = lowestCommonAncestor(root.left, p, q);

TreeNode right = lowestCommonAncestor(root.right, p, q);

if (left != null && right != null) return root;

else return left == null ? right : left;

}

private boolean getDirection(TreeNode ancestor, int value, List<String> steps) {

if (ancestor == null) return false;

if (ancestor.val == value) return true;

steps.add("L");

if (getDirection(ancestor.left, value, steps)) return true;

steps.remove(steps.size() - 1);

steps.add("R");

if (getDirection(ancestor.right, value, steps)) return true;

steps.remove(steps.size() - 1);

return false;

}

}

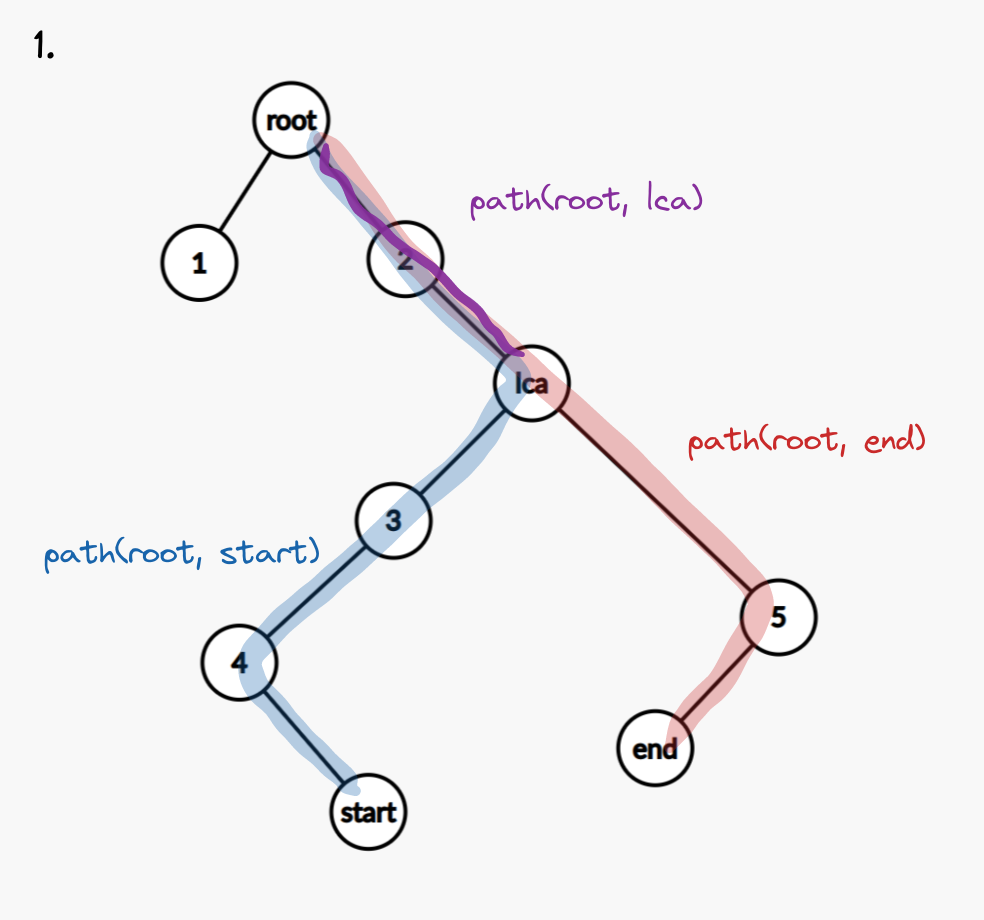
**Refer to**

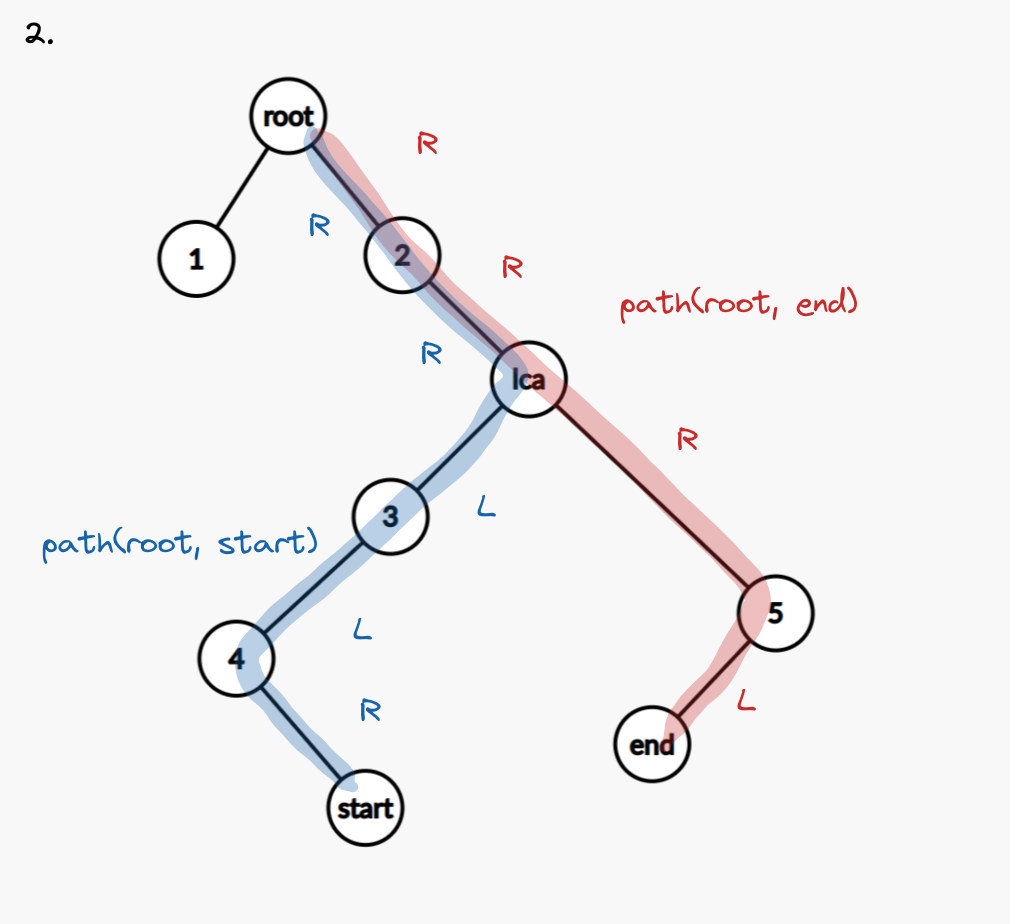
<https://algo.monster/liteproblems/2096>

Let path(node1, node2) denote the path from node1 to node2.

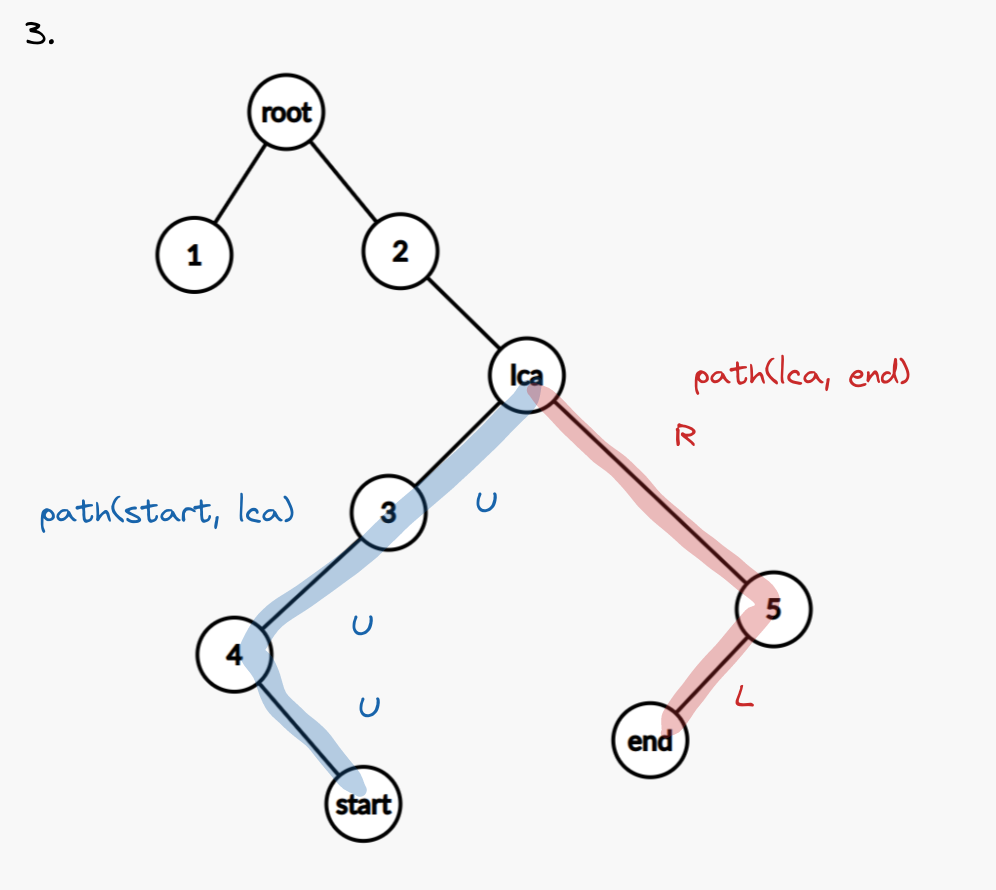
First consider the case where path(start, end) goes through the root. Let's split this into path(start, root) + path(root, end). We can perform a DFS (depth first search) to get path(root, end). This path consists of 'L's and 'R's. We can do another DFS to get path(root, start). Replacing the 'L's and 'R's of path(root, start) with 'U's gives us path(start, root). Now we can concatenate path(start, root) and path(root, end) to get the answer.

In the general case, path(start, end) may not go through the root. Notice that this path goes up a non-negative number of times ("U"s) before going down a non-negative number of times ("L"s or "R"s). The highest node in this path is known as the LCA (lowest common ancestor) of start and end.





Here, path(root, start) = "RRLLR" and path(root, end) = "RRRL". Let's remove their longest common prefix, which is "RR". We have path(LCA, start) = "LLR" and path(LCA, end) = "RL".



Then we replace all characters in path(root, start) with "U"s to obtain path(start, lca) = "UUU". Finally, we get path(start, end) = path(start, LCA) + path(LCA, end) = "UUU" + "RL" = "UUURL".

Time complexity

Each DFS takes O(n) and our string operations never happen on strings exceeding length O(n). The time complexity is

O(n).

Space complexity

The strings never exceed length O(n). The space complexity is O(n).

**Refer to**

[L1740.Find Distance in a Binary Tree (Ref.L2096,L236)](note://WEB1c998eebdd0f0ba3db6a7da7b17e012f)

[L236.Lowest Common Ancestor of a Binary Tree (Ref.L865,L235)](note://E191ABBC6A9B4A3C989AF0136CABCFA4)